Mikrodatorteknik för högskolans ingenjörsutbildningar

LOMEK – Läromedel på elektronisk form

Mikrodatorteknik för högskolans ingenjörsutbildningar
©1997,1998,1999,2000,2001 Roger Johansson och Rolf Snedsbøl

Begränsad kopieringsrätt

Detta verk är skyddat av upphovsrättslagen.

Kopiering är tillåten inom ramen för det avtal som träffats mellan GMV och användare.

Den som bryter mot lagen om upphovsrätt kan åtalas av allmän åklagare och dömas till böter eller fängelse i upp till två år samt bli skyldig att erlägga ersättning till upphovsman/rätts-innehavare.

Göteborgs Mikrovaror

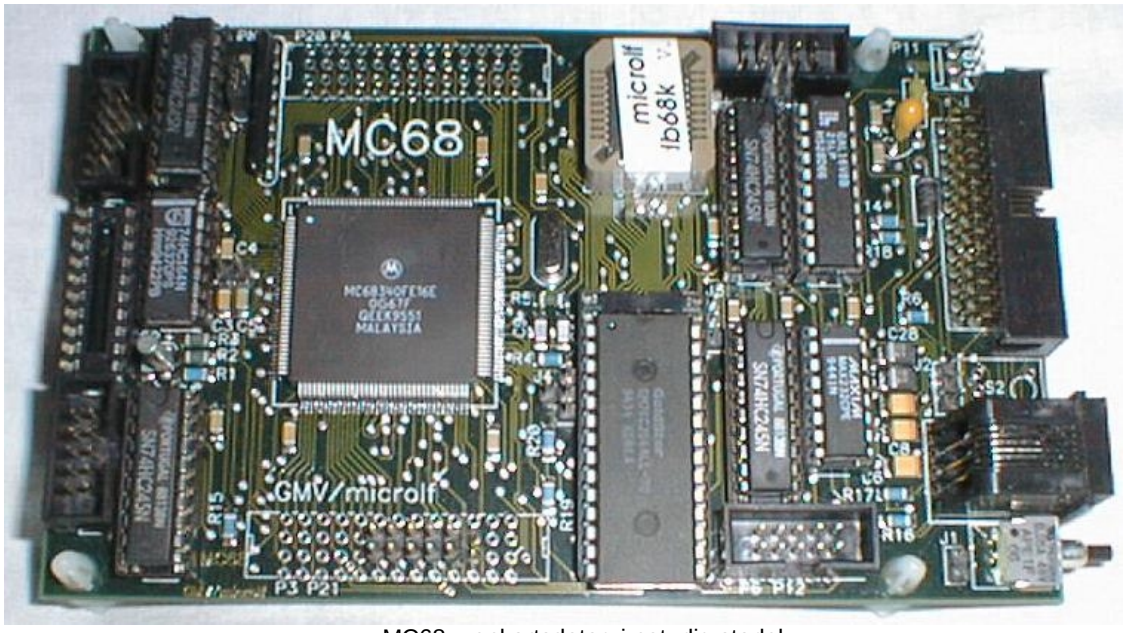
Tel: 031/778 93 10
Fax: 031/778 93 11
e-post: info@gmv.nu
Internet: http://www.gmv.nu

Förord

Denna bok utgör läroboken i ett material avsett för kurser i grundläggande mikrodatorteknik för icke data-/elektroteknik- inriktade ingenjörs-utbildningar. Läromedlet förutsätter inga speciella förkunskaper (utöver allmänna behörighetskrav för högskolestudier) men en genomförd grundkurs i digitalteknik kan underlätta instuderingen.

Innehåll:

1. Introduktion till mikrodatorn



MC68 – enkortsdator, i naturlig storlek.

Med ordet “dator” (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. En dator är ett system som består av flera komponenter där mikroprocessorn spelar en central roll. Detta inledande kapitel syftar till att introducera många av de begrepp och de speciella benämningar (dvs den terminologi) som är nödvändig för fortsatta studier av datorns funktionssätt och dess användning.

1.1 Datortyper och användningsområden

De flesta människor har helt säkert, i bland utan att ens tänka på det, stött på datorer eller mikrodatorer i någon form, exempelvis i en mikrovågsugn, en tvättmaskin eller en videobandspelare. Någon gång har du kanske själv använt en speldator med något spelprogram. Bland det första man då kommer i kontakt med är de delar av datorn som används för att ge datorn "kommandon". En sådan del av systemet kallas *inmatningsenhet*. Exempel på inmatningsenheter är:

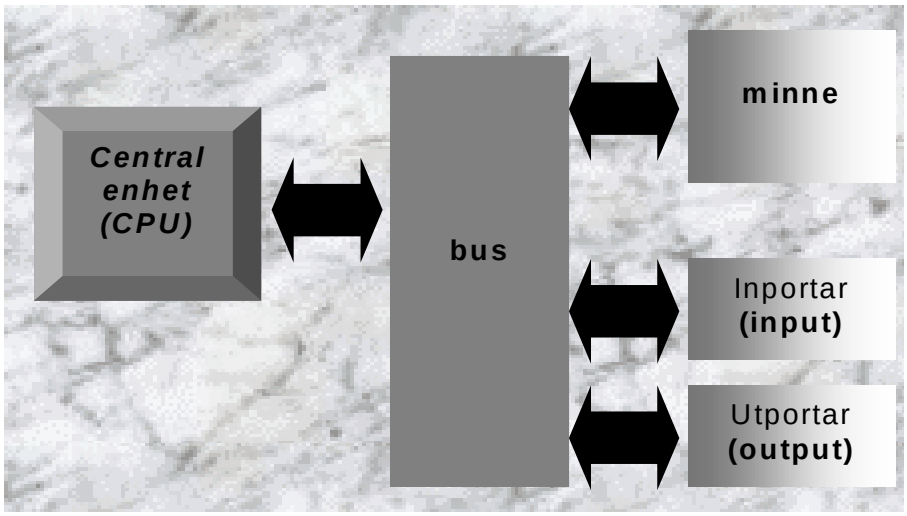
- tangentbordet eller pekdonet ("musen") till en persondator
- tangenterna på en videobandspelare
- knappsatsen till ett TV-spel

Kommandon som ges via inmatningsenheten tolkas och utförs av datorn. Resultatet återges vanligtvis som en utskrift på någon form, text eller bild, men det kan också vara ljud, via en utmatningsenhet från datorn. Exempel på utmatningsenheter är:

- bildskärmen eller högtalarna till en persondator
- en lampa eller ett vred som indikerar att tvätten är klar

Modern persondator för multimedia-tillämpningar

En dator kan alltså innehålla flera olika typer av av inmatningsenheter och flera olika typer av utmatningsenheter, dessutom måste det alltid finnas en enhet som tolkar och utför olika kommandon. En dator kan alltså påverka sin omgivning via sina utenheter (*utportar)* och därigenom exempelvis styra mekaniken i en videobandspelaren enligt en sekvens av kommandon som den har tagit emot från någon inenhet (*inport*). Enheten som tolkar och utför kommandona kallas *centralenhet* (eng: *Central Processing Unit CPU*) eller vanligtvis helt enkelt *processor*. Utöver CPU'n behövs också ett *minne* där styrinformation och data kan lagras. Enheterna är sammankopplade med signalledningar som används för att överföra styrinformation (kommandon) och data. Flera signalledningar med liknande funktion brukar gemensamt kallas *buss* (eng *bus).* Vanligen talar man om tre olika bussar, nämligen *styrbuss*, *databuss* och *adressbuss*.

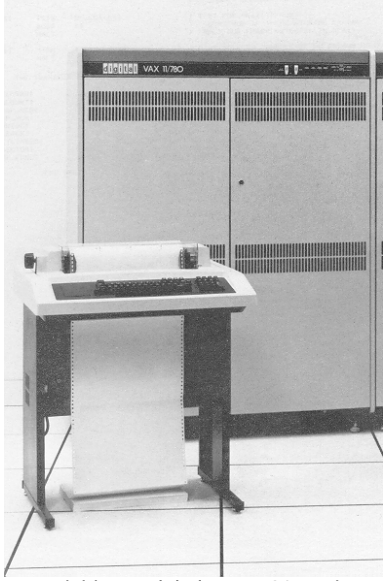


Figur 1.1 En dator's principiella uppbyggnad

Hur centralenheten ska uppföra sig för ett bestämt kommando avgörs av det *program* som finns lagrat i datorns minne. Ett program kan sägas vara en sekvens av *instruktioner* som centralenheten utför för att åstadkomma önskad operation.

Programmets storlek (antal instruktioner) kan vara från ett fåtal till flera miljoner beroende på uppgiftens omfattning och svårighetsgrad. Detta medför att vissa datorsystem i dag kräver lite minne medan andra kräver mycket stort minne för att lagra programmet.

Ett datorsystem *klassificeras* ofta med hjälp av begreppen *stordator*, *minidator* och *mikrodator*. Observera att det *principiella arbetssättet* för dessa olika datorklasser är det samma och att uppbyggnaden (inportar, utportar, CPU och minne) kan vara likartad. Det som skiljer dem åt är antalet portar, storleken på minnet, arbetstakten hos CPU'n etc och därmed också mängden av data de kan bearbeta på en viss tid.



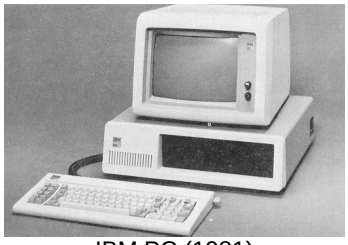
Minidator *Digital VAX 780* med skrivande terminal (tangentbord och printer)

En *stordator* används i dag för att utföra omfattande beräkningar och simuleringar, exempelvis simulering av luftströmmarna kring ett flygplan, beräkningar för att ge detaljerade väderprognoser etc. Stordatorer används också exempelvis i bankcentraler där de bearbetar mycket stora mängder data. Flera tusen arbetsplatser (bildskärms-terminaler) eller *arbetsstationer* (se nedan) kan vara anslutna till en stordator. Detta innebär då att åtskilliga användare kan använda stordatorn samtidigt.

Gruppen *minidatorer* har, från att under 1970-talet och första delen av 1980-talet spelat en viktig roll, allt mer minskat i betydelse. En minidator kan, precis som en stordator, nyttjas i exempelvis banksammanhang, men då lokalt på ett enskilt bankkontor, med flera arbetsplatser anslutna. Det finns då någon form av förbindelse mellan huvuddatorn (en stordator på centralbanken) och minidatorn. Ett annat exempel som illustrerar användningsområdet för en minidator kan vara att styra ett antal robotar vid det löpande bandet i en fabrik. En minidator kan användas av flera användare, typiskt 8-32 stycken, samtidigt.

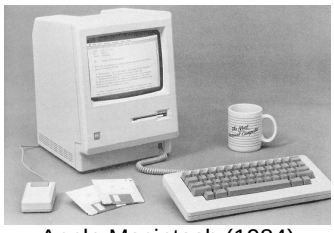
Gruppen *mikrodatorer* uppstod genom att miniatyriseringen av datorelektroniken gjorde det möjligt att placera all elektronik och övriga komponenter i en systemenhet som fick plats i *en* mindre låda. Mikrodatorer utnyttjas vanligtvis av en användare åt gången och kan indelas i *generella mikrodatorer* och *mikrodatorer för special-tillämpningar*. De generella mikrodatorerna är av typ *hemdatorer*, *persondatorer* eller *arbetsstationer*.

Hemdatorerna började utvecklas under senare delen av 1970-talet. De bestod av systemenhet med processor och arbetsminne, tangentbord och bildskärm. Till hemdatorerna kunde ofta någon form av enkel kasettbandspelare anslutas, via kasettband kunde olika program distribueras. Hemdatorer var främst avsedda för enkla tillämpningar som exempelvis ordbehandling, bokföring och registerhantering men kanske framför allt spel.


IBM PC (1981)

Persondatorn föddes när IBM presenterade sin *PC* (*Personal Computer*) år 1981. Den bestod i princip av en förbättrad hemdator med tangentbord och en egen bildskärm. Den hade också försetts med en så kallad *flexskivestation* som ersatte kasettbandspelaren som lagringsmedium för program. Ganska snart startade flera mindre företag tillverkning av PC-kopior till lägre pris och ofta bättre prestanda än originalet.

Eftersom PC'n och de så kallade "IBM PC kompatibla" kopiorna använde samma processor (Intel 8088) och samma systemprogramvara (PC-DOS, MS-DOS) uppstod en marknad för programvaror som kunde säljas i stora upplagor till lågt pris. Ett par år efter introduktionen av PC presenterades *IBM PC XT* som också hade försetts med en *hårddisk* för lagring av program och data.

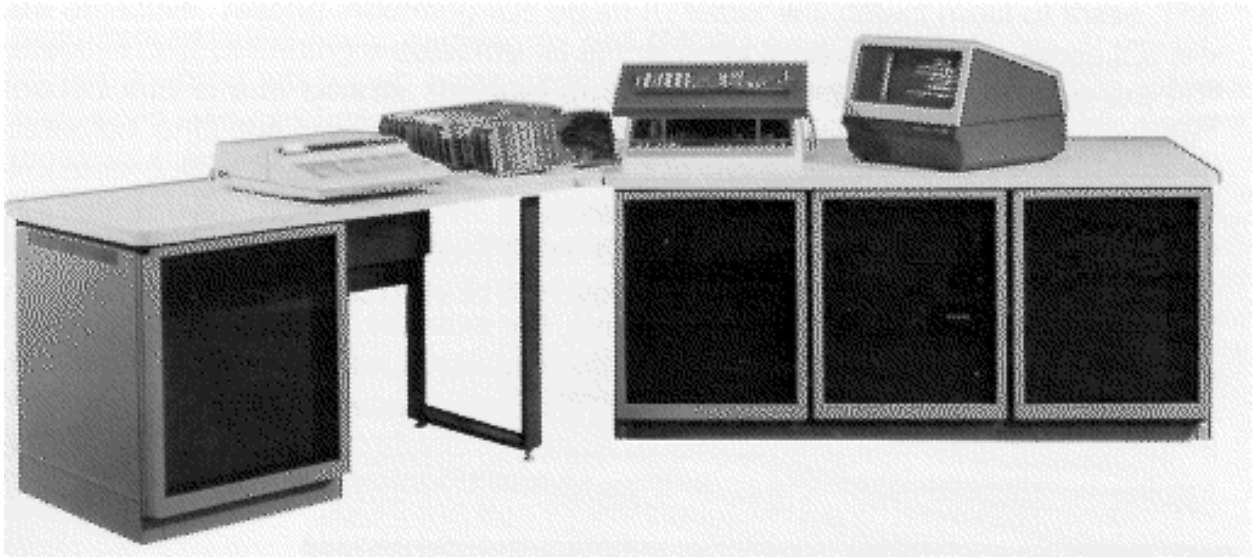

Apple Macintosh (1984)

En annan viktig gren på persondatorträdet skapades när Apple introducerade *Macintosh* år 1984. Den viktigaste nyheten med Macintosh var den användarvänliga "fönsterhanteringen" med menyer och användning av pekanordning som kallas för "mus". Eftersom Macintosh använder en annan processor (Motorola MC68000) än IBM PC och konstruktionen i övrigt är mycket annorlunda kan program inte enkelt flyttas mellan dessa båda typer av persondatorer.


Sun SPARC arbetsstation (1992)

De mest avancerade och kraftfulla generella mikrodatorerna kallas ofta arbetsstationer. Systemprogramvaran och övriga program som utförs på arbetsstationer är normalt hämtad från minidatorvärden och är därför betydligt dyrare än PC-programvaran. Prestandamässigt har skillnaden mellan högpresterande PC och vanliga arbetsstationer i dag nästan försvunnit. En arbetsstation används normalt av en eller möjligtvis ett fåtal användare.

Mikrodatorer för specialtillämpningar är ofta inbyggda i utrustning exempelvis för videospel, tvättmaskiner, videobandspelare, förarlösa truckar, navigationsutrustning, industrirobotar, bilar, mätutrustningar etc.


Mätutrustning från Hewlett Packard, tidigt 80-tal

1.2 Program och programmering

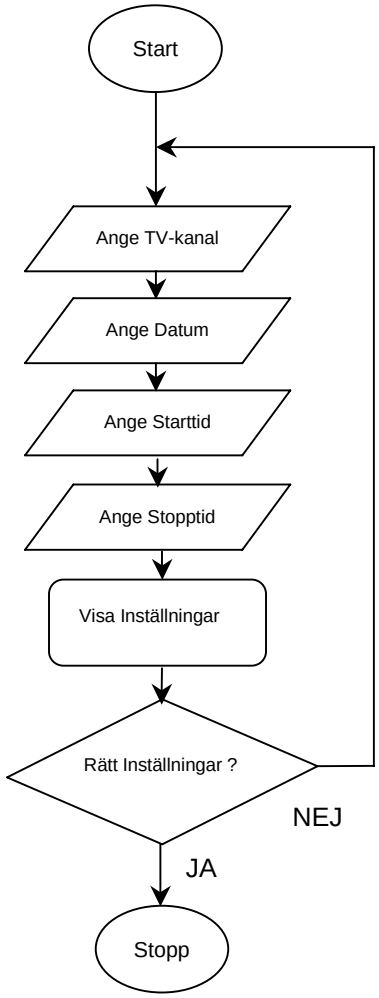
Ett program för en dator skrivs i något *programspråk.* Exempel på programspråk är bland andra *Pascal, BASIC, C, ADA, FORTRAN* osv. I varje programspråk finns speciella regler för *vad* man kan skriva och *hur* det ska skrivas. Hur kan då samma program fås att fungera på olika typer av datorer trots att dessa kan vara uppbyggda av olika kretsar och olika typer av centralenheter?

Vi kan faktiskt jämföra detta med ett musikstycke som beskrivs av ett antal noter. Musikstycket kan spelas på exempelvis en klarinett eller en saxofon, där klarinetten då skulle motsvara en dator av någon typ och saxofonen motsvarar en dator av någon annan typ. Innehållet på notbladet motsvarar programmet. Personen som spelar måste kunna översätta noterna till tangentnedtryckningar anpassade till det instrument han spelar på. Resultatet är en följd av toner med frekvenser som skall vara lika i båda fallen.

På samma sätt är det med datorer av olika typer, raderna i programmet (*Pascal, Basic* eller *C* osv) måste översättas till korrekta instruktioner för den aktuella centralenheten.

1.2.1 Algoritmer

Den sekvens av kommandon (de instruktioner) vi anger för att programmera exempelvis en videobandspelaren för inspelning kallas alltså *program*. För att beskriva den uppgift programmet ska utföra använder vi text och i bland även symboler. En sådan beskrivning kallas en *algoritm.* Algoritmen beskriver alltså vad programmet gör och kan som regel användas *oberoende av vilken typ av videobandspelare* vi har.



EXEMPEL
Algoritmen för en förinställd inspelning skulle kunna vara:

- ✓ Ange TV-kanal
- ✓ Ange datum
- ✓ Ange klockslag då inspelningen ska påbörjas
- ✓ Ange klockslag då inspelningen ska avbrytas
- ✓ Kontrollera inställningar:
- ✓ Om dessa är rätt så är vi klara,annars börjar vi om

Jämför algoritmen med flödesdiagrammet (figur i marginalen). Observera speciellt hur textramarna försetts med olika former beroende på textens mening.

Detta är en förhållandevis "grov" algoritm eller en algoritm på *hög* nivå. Om vi inte tidigare använt denna videobandspelare behöver vi förmodligen också en beskrivning av *hur* man anger TV-kanal, *hur* man anger datum, osv. Den grova algoritmen kan då stegvis kompletteras med sådan information, vi kallar detta att gradvis förfina algoritmen. Den *slutgiltiga* algoritmen kan på detta sätt bli mycket detaljerad.

Då man vill konstruera en välstrukturerad och korrekt algoritm är det vanligtvis enklast att först skriva algoritmen på en “hög” nivå för att

senare förfina den i flera steg. Eftersom algoritmen förfinas genom att fler detaljer infogas blir det också större och större textmassa, denna kan då snabbt bli oöverskådlig. I sådana fall kan det vara bra att använda sig av *figurer* (symboler) som exempelvis ett flödesdiagram. Det finns olika standarder för hur ett sådant skall se ut med speciella symboler som anger att data skall hämtas in eller skrivas ut, om valmöjligheter önskas osv.

För att algoritmen skall kunna användas måste den först översättas till en sekvens instruktioner som videobandspelaren kan förstå, så kallade *maskininstruktioner*. Då vi utfört denna översättning säger vi att vi har *implementerat* algoritmen i ett språk som är avsett för videobandspelaren.

Oberoende av på vilken nivå vi arbetar så kommer eventuella fel vi gör när vi skriver algoritmen eller när vi implementerar denna, att ge upphov till att en felaktig sekvens av kommandon utförs. Centralenheten tolkar och utför även felaktiga kommandon och vi kan då få såväl oväntade som oönskade resultat. Vad kan exempelvis hända om vi anger “Datum” då videobandspelaren förväntar sig “TV-kanal”? Av detta framgår vikten av att så väl algoritmer som implementeringen av dessa måste ägnas stor tankemöda och omsorg.

1.2.2 Syntax och semantik

I beskrivningar av algoritmer använder man oftast ett så kallat *formellt språk*. Detta är i flera avseenden förenklat i jämförelse med vardagsspråk. Anledningen till detta är att man till varje pris vill undvika missförstånd mellan den som *skrivit* algoritmen och den som *läser* algoritmen, man eftersträvar alltså *entydighet* med ett formellt språk. Precis som för vårt vardagsspråk finns det då speciella *regler* för hur vi kan uttrycka oss i ett formellt språk, dessa regler är dock i allmänhet betydligt mer stränga. Vi skiljer på *syntaktiska regler* (syntax), dvs reglerna för *hur* vi uttrycker oss, och *semantiska regler*, dvs innebörden i det vi säger. Att vi använder rätt ordföljd är alltså en fråga om syntax, men om åhöraren *förstår vad vi menar* är en fråga om semantik. För en mikroprocessor är semantiken klar för varje enskild instruktion. Då det gäller syntaxen är mikroprocessor'n däremot betydligt förvirrad. En mikroprocessor utför slaviskt instruktionföljder oavsett om dessa är vettiga eller inte. För en dator i allmänhet måste såväl syntax som semantik vara *entydig* (inte grundad på erfarenhet) och därmed också, i allmänhet, mycket formell.

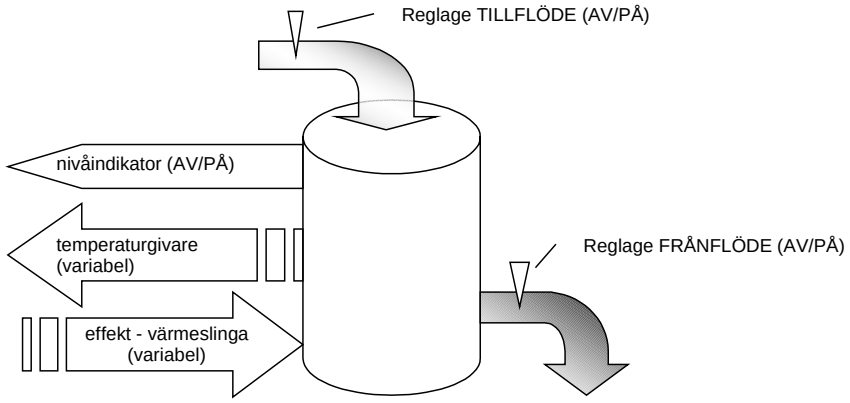
1.3 Datorn som styrsystem

Ett stort och mycket viktigt användningsområde för mikroprocessorer är att övervaka och styra olika typer av mekaniska (elektro-mekaniska) system. Denna grupp av datorsystem kallas ofta *styrsystem*. Det finns en lång rad exempel på sådana och det kan vara allt ifrån små, enkla till mycket stora, komplicerade system. Vad menar man då egentligen med begreppet styrsystem? Alldeles uppenbart finns det någonting som vi av någon anledning vill styra. Detta föremål, oavsett vad det är kallas

styrobjekt. Vi använder dessutom benämningarna *kontrollerande system*, för själva datorsystemet och *kontrollerat system* för styrobjektet.

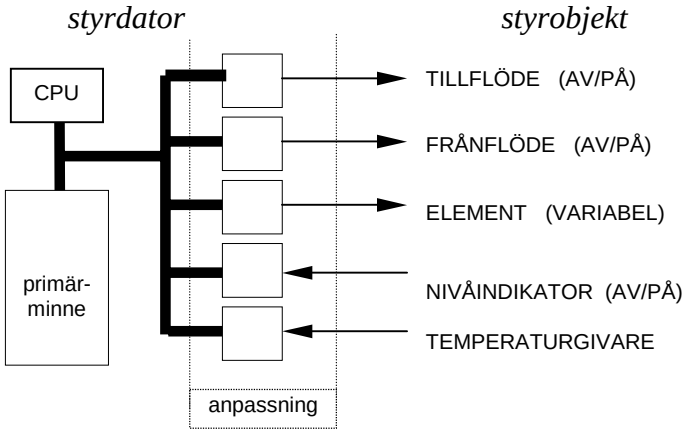
Som exempel på ett enkelt styrsystem kan vi ta en varmvatten-beredare. Den består av en vattentank, ett *värmeelement*, en *nivå-indikator*, ett *tillflöde* och *utflöde* (se figur nedan).

För enkelhets skull antar vi följande funktionssätt: vattentanken skall fyllas med kallt vatten, detta skall sedan värmas till 60°C, slutligen töms tanken på varmvatten. Tillflödet och utflödet styrs via ventiler. En sådan konstruktion kan styras av en enkel styrdator.



Figur 1.1 Illustration till exemplet på en varmvattenberedare

Vi kan illustrera såväl *styrdator* som *styrobjekt* med en enkel *schematisk* bild av hela systemet och utifrån denna bild diskutera de ingående komponenterna.



1.3.1 Enkel reglering AV/PÅ

Ventilerna för vattenflödet till (eller från) tanken är av enkel typ och kan öppnas eller stängas. Det finns också ventiler som kan strypas succesivt och därmed reglera *mängden* vatten som strömmar genom ventilen, men här tänker vi oss enkla ventiler som kan vara *av* eller *på*.

1.3.2 Variabel reglering

En värmeslinga där vi kan styra värmen genom att påföra mer eller mindre effekt kan utgöra exempel på *variabel* reglering. En kontinuerligt variabel reglering kan åstadkommas med hjälp av bland annat en så kallad D/A-omvandlare (digital till analog omvandlare). Vi återkommer till sådana kretsar i kapitel 6.

1.3.3 Enkel givare AV/PÅ

En enkel givare kan exempelvis användas för att indikera om vattentanken är full eller inte. Då vattnet i tanken når upp till givaren kommer den att slå på, då vattnet sjunker under givaren kommer den att slå av. Denna typ av givare kan alltså inte ge oss information om *hur mycket* vatten vi har i tanken, bara om vattnet i tanken når en viss nivå.

1.3.4 Variabel givare

I vårt exempel använder vi en termometer för att mäta temperaturen på vattnet. Denna kan ses som exempel på en *variabel* givare, dvs vi får direkt information om temperaturen. Sådana givare kan byggas upp med bland annat så kallade A/D-omvandlare (analog till digital omvandlare). Vi behandlar även dessa i kapitel 6.

1.3.5 Styrdatorns funktion

Vårt styrsystem är ännu inte färdigt. Vi måste (mycket detaljerat) steg för steg, beskriva vad styrdatorn ska göra. Vi gör detta i form av ett *program* för styrdatorn. Varmvattenberedarens funktion har beskrivits tidigare. När denna är fylld med 60°-igt vatten ska detta tömmas ut och därefter ska beredaren på nytt fyllas med kallt vatten.

Låt oss inte angripa hela problemet på en gång. Detta kan bli onödigt komplicerat. I stället försöker vi bryta ned det i så små delar som möjligt. Vi utgår från att tanken från början är tom, att ventilerna är stängda och att värmeregleringen står på noll. Vi börjar nu med att fylla upp tanken med vatten. För att göra detta måste vi stänga FRÅNFLÖDET och öppna TILLFLÖDET. Samtidigt måste vi försäkra oss om att tanken inte överfylls. Vi kan ge en kortfattad men precis beskrivning på hur detta ska gå till:

```
Algoritm FYLL_TANK(1)

        STÄNG FRÅNFLÖDE
        ÖPPNA TILLFLÖDE
        NÄR TANKEN ÄR FULL, STÄNG TILLFLÖDE
```

Är denna algoritm tillräckligt detaljerad? Vi kontrollerar:

STÄNG FRÅNFLÖDE, detta är inte speciellt svårt. Vi åstadkommer det genom att ge styrsignalen AV till frånflödes-ventilen. Eftersom vi bara har en frånflödesventil är anvisningen också entydig, vi riskerar alltså inte att stänga fel ventil då instruktionen följs. Anvisningen är därför tillräcklig. Med samma resonemang kan vi konstatera att anvisningarna ÖPPNA TILLFLÖDE och STÄNG TILLFLÖDE är tillräckligt detaljerade. Detta gäller dock *inte* anvisningen *NÄR* TANKEN ÄR FULL. Varför är det så? Jo, vi har visserligen en nivåindikator i tanken men den indikerar ju faktiskt bara *ATT* tanken är full. För att avgöra *NÄR* tanken fyllts upp måste vi därför hela tiden kontrollera nivåindikatorn. Vi kan åstadkomma detta genom att använda en repetitions-sats kombinerad med ett villkor i algoritmen:

```
Algoritm FYLL_TANK(2)

        STÄNG FRÅNFLÖDE
        ÖPPNA TILLFLÖDE
REPETERA:
        OM TANK ÄR FULL STÄNG TILLFLÖDE
TILLS:  TILLFLÖDE STÄNGT
```

Är detta nu tillräckligt? Vi kontrollerar genom att utföra algoritmen " i huvudet". Med den beskrivna regleringen och de specificerade givarna kan vi implementera algoritmen, det verkar inte speciellt svårt, men hur *bra* är egentligen denna algoritm? Tar algoritmen hänsyn till alla möjliga situationer eller fungerar den bara under vissa förutsättningar? Låt oss föreställa oss *alla* tänkbara situationer för varmvattenberedaren

1. Tanken är från början tom.
2. Tanken är från början *delvis* fylld.
3. Tanken är från början fylld.

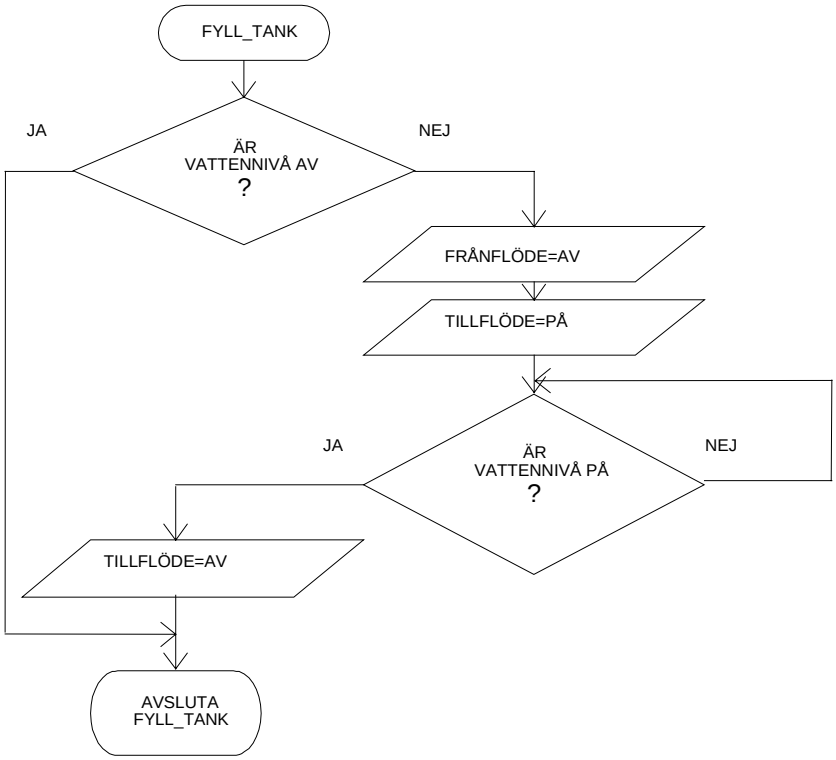
Finns det fler förutsättningar för algoritmen?

Vi sluter oss till att vi tagit hänsyn till alla tänkbara förutsättningar för vår algoritm. Studera nu version 3 (FYLL_TANK(3)), här har vi format om algoritmen (eller *förfinat* den) så att den ännu klarare kan kopplas till styrobjektets (det kontrollerade systemets) komponenter. Jämför med föregående version, kan denna algoritm sägas vara *tillräckligt* detaljerad?

```
Algoritm FYLL_TANK(3)

OM VATTENNIVÅ=AV          ( ej full tank)
        UTFÖR:
                FRÅNFLÖDE=AV            (stäng frånflöde)
                TILLFLÖDE=PÅ            (öppna tillflöde)
                REPETERA:
                TILLS: VATTENNIVÅ=PÅ    (kontrollera nivå)
                TILLFLÖDE=AV            (stäng tillflöde)
AVSLUTA FYLL_TANK                       (klar ...)
```

Ur implementeringssynpunkt kan vi konstatera att vi har tillräcklig information av algoritmen. Vi ska längre fram (kapitel 4) se hur algoritmens instruktioner kan implementeras direkt för en mikroprocessor, dvs varje instruktion i algoritmen svara mot en (eller möjligtvis ett fåtal) maskininstruktioner för den processor vi använder. Låt oss (för

fullständighetens skull) uttrycka samma algoritm med hjälp av ett flödesdiagram:



Figur 1.2 Flödesdiagram FYLL_TANK(3)

Låt oss nöja oss med algoritmen för att fylla upp tanken och nu värma upp vattnet....

```
Algoritm VÄRM_VATTEN

        OM TEMPERATUR ÄR MINDRE ÄN 60 grader
                UTFÖR:
                SÄTT ELEMENT TILL FULL_EFFEKT
                        REPETERA:
                        TILLS: TEMPERATUR = 60 grader
                SÄTT ELEMENT TILL NOLL_EFFEKT
        AVSLUTA VÄRM_VATTEN
```

Allgoritmen för att tömma tanken blir enkel...

```
Algoritm TÖM_TANK

        TILLFLÖDE=AV
        FRÅNFLÖDE=PÅ
        AVSLUTA TÖM_TANK
```

Avslutningsvis sätter vi samman delarna i en algoritm för ett "Huvudprogram"

```
Algoritm VARMVATTENBEREDARE

        START
        FYLL_TANK
        VÄRM_VATTEN
        TÖM_TANK
        STOPP
```

Diskutera gärna de algoritmer och det system som användes i exemplet. Fundera speciellt över frågorna:

- Algoritmen visar endast *en* uppvärmning, men oftast vill man ju ha ständig tillgång till varmvatten. Hur kan man ändra i huvudprogrammet för att uppnå detta ?
- Vi har ett problem i algoritmen TÖM_TANK, nämligen: hur kan vi veta när tanken verkligen är tom?

Det visade exemplet illustrerar någraviktiga aspekter då det gäller mikroprocessorbaserade styrsystem. Vi vill uppenbarligen kunna uttrycka systemets funktionssätt med hjälp av entydiga (och begripliga) algoritmer för olika förutsättningar. Vi vill samtidigt kunna översätta dessa algoritmer till instruktioner för en (billig) mikroprocessor, dvs implementera algoritmerna, på ett enkelt sätt som inte kan misstolkas. Det räcker då inte med att ha en fullständig förståelse för det kontrollerade objektet, dvs styrobjektets natur och egenskaper. Vi måste också förstå mikroprocessorn, dess arbetssätt och framför allt dess användbarhet. Dessa kunskaper omsättes då vi konstruerar verkliga system bestående av elektroniska och mekaniska komponenter, så kallade *mekatroniska system*.

1.4 Talsystem

Vi människor använder normalt det decimala talsystemet för att ange talvärden. I grundskolans matematik lär man sig först addition och subtraktion av decimala tal. Därefter lär man sig multiplikation och division. För oss är det självklart att använda det decimala talsystemet för att representera talvärden och beräkna talvärden. Det decimala talsystemet kallas också det arabiska talsystemet eftersom det använder de arabiska siffersymbolerna 0 t o m 9. Ordet decimal kommer ifrån det latinska ordet "decem" med betydelsen "tio". Tio symboler används eftersom människan har tio fingrar och använder dem att räkna på. Engelskans "*digit*" för siffra har sitt ursprung i det latinska ordet "digitus" för finger.

Det decimala talsystemet är ett s k *positionssystem*. I ett sådant system bestäms värdet (vikten) av varje siffra, som ingår i ett tal, av dess plats i talet (positionen). Vikten hos en siffra ökar ju längre till vänster den står i talet. I denna framställning förutsätts i fortsättningen att alla använda talsystem är positionssystem.

Man säger att det decimala talsystemet har *basen* tio eftersom det använder tio olika siffersymboler (0 - 9). Då basen är det antal siffersymboler som används är den *alltid* ett heltal. Man kan konstruera ett talsystem för varje bas som är två eller större. Det talsystem som bygger på två siffersymboler (basen två) kallas det binära talsystemet. Detta talsystem har blivit mycket viktigt i allmänna tekniska sammanhang och är centralt i mikroprocessortekniken.

Binärt	Hexa-deci-malt	De-ci-malt	Ok-talt
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	8	8	10
1001	9	9	11
1010	A	10	12
1011	B	11	13
1100	C	12	14
1101	D	13	15
1110	E	14	16
1111	F	15	17
10000	10	16	20
10001	11	17	21
10010	12	18	22
10011	13	19	23
10100	14	20	24
10101	15	21	25
10110	16	22	26
10111	17	23	27
11000	18	24	30
11001	19	25	31
11010	1A	26	32
11011	1B	27	33
11100	1C	28	34
11101	1D	29	35
11110	1E	30	36
11111	1F	31	37
100000	20	32	40
100001	21	33	41

Tabell 1.1 Talvärden för tal i olika positionssystem

EXEMPEL
Man kodar ett decimalt tal som en följd av siffror, exempelvis 435,72. Detta tolkas som att

fyran är värd fyra hundratal	$(4 \cdot 10^2)$
trean är värd tre tiotal	$(3 \cdot 10^1)$
femman är värd fem ental	$(5 \cdot 10^0)$
sjuan är värd sju tiondelar	$(7 \cdot 10^{-1})$
tvåan är värd två hundradelar	$(2 \cdot 10^{-2})$

Talet kan uttryckas som summan

$$4 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0 + 7 \cdot 10^{-1} + 2 \cdot 10^{-2}$$

Varje siffras vikt (*signifikans)* bestäms av dess position i talet relativt decimalkommat. Heltalsdelen skrivs till vänster om kommatecknet och bråktalsdelen till höger.

Ett talvärde N i ett positionssystem med bas *r* (även kallat *radix*) kan alltså uttryckas som

$$N = d_{n-1} * r^{n-1} + ... + d_0 * r^0 + ... + d_{-m} * r^{-m} \quad \text{(Ekv 1.1)}$$

Koefficienterna d_k (k = -m, ..., n-1) representerar siffror i talsystemet. Antalet heltalssiffror är *n* och antalet bråktalssiffror är *m*. Sifferföljden tolkas som summan *N* där varje siffra får en potens av basen som vikt. Heltalsdel och bråktalsdel åtskiljs av en punkt (**.**) (Kommat används bara i det decimala talsystemet). Är punkten utelämnad består talet enbart av en heltalsdel. Siffran längst till vänster d_{n-1} sägs vara den mest signifikanta (eng:*Most Significant Digit*, *MSD*) och siffran längst till höger d_{-m} den minst signifikanta (eng:*Last Significant Digit , LSD*).

Om talsystemets bas *r* är mindre än 10 används de *r* första av de arabiska symbolerna 0-9 som siffror. Är basen större än 10 kompletteras siffrorna 0-9 med stora bokstäver ur alfabetet med början på A. Detta illustreras i Tabell 1.1, där de trettiofyra första heltalen i några olika talsystem visas. Vi kommer i denna framställning att speciellt behandla de binära och hexadecimala talsystemen.

Eftersom de arabiska siffrorna förekommer i flera positionssystem, kan man ofta inte avgöra i vilket talsystem ett tal är kodat genom att enbart betrakta talets siffror. För att undvika missförstånd brukar man därför innesluta talet i en parentes och ange talsystemets bas som index till parentesen.

EXEMPEL Skrivsätt för att ange tal i olika talsystem

$(1001)_2$	bas 2
$(1072)_8$	bas 8
$(2934)_{10}$	bas 10
$(3AF5)_{16}$	bas 16

I källtexter till datorprogram är det inte möjligt att använda parenteser (eftersom dessa oftast har speciell betydelse) eller index. Man brukar därför istället ge talen ett *prefix*.

EXEMPEL Prefix för tal i olika talsystem i programspråket C

01072	Att första siffran är 0 (noll) markerar ett oktalt tal	(bas 8)
2934	Att första siffran är skild från 0 (noll) markerar ett decimalt tal	(bas10)
0x3AF5	Prefixet "0x" (noll-x) markerar ett hexadecimalt tal	(bas16)

EXEMPEL Prefix för olika talsystem i Motorola assemblerkod

%1001	Prefixet **%** markerar ett binärt tal	(bas 2)
@1072	Prefixet @ markerar ett oktalt tal	(bas 8)
2934	Inget prefix markerar ett decimalt tal	(bas10)
$3AF5	Prefixet **$** markerar ett hexadecimalt tal	(bas16)

Det finns alltså flera olika sätt att koda talvärden. Siffror kan på liknande sätt användas för att koda annat än talvärden. Man kan t.ex. använda siffror för att koda bokstäver, skiljetecken, aritmetiska operationer o dyl. I senare avsnitt kommer andra kodningssätt att behandlas.

1.4.1 Talomvandling

De aritmetiska operationerna addition (+), subtraktion (-), multiplikation (*) och division (/) har en egen definition i alla talsystem. Om man har två talvärden och vill addera, subtrahera, multiplicera eller dividera dem måste de därför vara givna i samma talsystem. Detta medför att man ibland måste "översätta" talvärden från ett talsystem till ett annat. Översättning *från* godtyckligt talsystem till det decimala talsystemet ges allmänt av Ekv 1.1. I den fortsatta framställningen behandlar vi nu *heltal* (inte bråktal). Ekv 1.1 får då formen

$$N = d_{n-1} * r^{n-1} + d_{n-2} * r^{n-2} + ... + d_0 \quad \text{(Ekv 1.2)}$$

där N är ett talvärde uttryckt med basen *r*, *n* är antalet siffror.

EXEMPEL Omvandla binära talet $(101110)_2$ till decimaltal.

Vi tillämpar Ekv 1.2 direkt. Antalet binära siffror (n) är 6, dvs

$(N)_{10} = 1*2^5 + 0*2^4+1*2^3+1*2^2+1*2^1+1*2^0 =$

$32+ 0 + 8+ 4+ 2+ 0 = 52$

DVS $(101110)_2 = (52)_{10}$

EXEMPEL Omvandla hexadecimala talet $(D3)_{16}$ till decimaltal

Vi tillämpar Ekv 1.2 direkt. Antalet hexadecimala siffror (n) är 2, dvs

$(N)_{10} = (D)_{16} * (10)_{16} + (3)_{16} =$

$(13)_{10}*(16)_{10}+(3)_{10} = (211)_{10}$

DVS $(D3)_{16} = (211)_{10}$

Låt oss nu se hur vi översätter *från* det decimala talsystemet till ett godtyckligt talsystem. Ekv. 1.2 ska nu gälla för talomvandlingen. Siffrorna d_{n-1} ... d_0 fås genom upprepade divisioner, av talet som ska omvandlas, med *r*. Första steget (divisionen) blir:

$$\frac{N}{r} = d_{n-1} * r^{n-2} + d_{n-2} * r^{n-3} + ... + \frac{d_0}{r} \quad \text{(Ekv 1.3)}$$

Högerledet består av en kvot Q_0 och en rest R_0 där

$$Q_0 = d_{n-1} * r^{n-2} + d_{n-2} * r^{n-3} + ... + d_1$$

och

$$R_0 = \frac{d_0}{r}$$

Siffran d_0, i det nya talsystemet, fås ur resten R_0. Nästa steg, som utförs på samma sätt, ger oss siffran d_1.

$$\frac{Q_0}{r} = d_{n-1} * r^{n-3} + d_{n-2} * r^{n-4} + ... + \frac{d_1}{r}$$

vilket ger:

$$Q_1 = d_{n-1} * r^{n-3} + d_{n-2} * r^{n-4} + ... + d_2$$

och

$$R_1 = \frac{d_1}{r}$$

Förfarandet upprepas tills $Q_n = 0$, dvs divisionen inte resulterar i en heltalsdel.

EXEMPEL Omvandla decimaltalet 211 till ett binärtal:

Q/r		kvot		rest	
211/2	=	105	+	1/2	$d_0 = 1$
105/2	=	52	+	1/2	$d_1 = 1$
52/2	=	26	+	0	$d_2 = 0$
26/2	=	13	+	0	$d_3 = 0$
13/2	=	6	+	1/2	$d_4 = 1$
6/2	=	3	+	0	$d_5 = 0$
3/2	=	1	+	1/2	$d_6 = 1$
1/2	=	0	+	1/2	$d_7 = 1$

DVS: $(211)_{10} = (11010011)_2$

EXEMPEL Omvandla decimaltalet 211 till hexadecimaltal:

Q/r		kvot	rest	
211/16	=	$(13)_{10} = (D)_{16}$	3/16	$d_0=3$
13/16	=	0	13/16	$d_1=D$

DVS $(211)_{10} = (D3)_{16}$

Det hexadecimala (och det oktala) talsystemet är i princip ett förkortat skrivsätt av det binära. Varje grupp om 4 binära siffror motsvarar exakt en hexadecimal siffra. Tabell 1.2 i marginalen (som är ett utdrag ur Tabell 1.1) återger de 16 olika hexadecimala siffrorna samt deras binära och decimala motsvarigheter.

Hexa-deci-malt	Binärt	Deci-malt
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Tabell 1.2 De 16 olika hexadecimala siffrorna

EXEMPEL Omvandla binärtalet $(1011001011)_2$ till hexadecimalt tal

Gruppera de binära siffrorna fyra och fyra, börja från höger:

10 1100 1011

Eftersom antalet bitar inte gick jämnt ut fyller vi på från vänster med nollor tills även denna grupp består av fyra binära siffror:

0010 1100 1011

Ur tabell 1.2 finner vi nu de hexadecimala motsvarigheterna:

2 C B

DVS: $(1011001011)_2 = (2CB)_{16}$

EXEMPEL Omvandla hexadecimaltalet $(A8)_{16}$ till binärt tal.

Varje hexadecimal siffra ger direkt fyra binära siffror (Tabell 1.2)

A 8

1010 1000

DVS: $(A8)_{16} = (10101000)_2$

Denna typ av omvandlingar återkommer du ständigt till då du arbetar med datorteknik på maskinnära nivå. Det är en bra id'e at lära sig utföra dessa omvandlingar utantill.

1.4.2 Binärkodade decimaltal

Istället för att omvandla ett decimalt tal till det binära talsystemet kan man binärkoda varje decimal siffra som ingår i talet. Detta kan göras med en mängd olika koder som med ett gemensamt namn kallas decimala binärkoder eller BCD-koder (eng:*Binary Coded Decimal*). Det krävs minst fyra bitar för att binärkoda de decimala siffrorna.

Kodning med fyra bitar kan ske på $16!/(16-10)! \approx 2{,}9 \times 10^{10}$ olika sätt. Varje fyrabitars BCD-kod har sex kombinationer av 0 och 1 utan innebörd. I den vanligaste koden representeras varje decimal siffra naturligt nog med sitt ekvivalenta tal i det binära talsystemet, (se Tabell 1.3), och kallas därför den naturliga decimala binärkoden eller NBCD-koden (eng Natural Binary Coded Decimal).

EXEMPEL Det decimala talet 9756 representeras i NBCD-kod som

9756 = 1001 0111 0101 0110

9 7 5 6

Decimal siffra	NBCD-kodord
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Tabell 1.3 NBCD-koden

De binära talen för 10, 11, 12, 13, 14 och 15 har här ingen innebörd. (se Tabell 1.3). Många processorer har något stöd för att räkna med NBCD-kod. Vissa har additionsinstruktioner speciellt för NBCD-tal, medan andra har en instruktion som *justerar* resultatet efter en vanlig binäraddition i fall summan av två fyrabitars ord blir större än 9. Det är mycket viktigt att förstå skillnaden mellan binära tal och tal uttryckta med decimal binärkod**.** I vardera fallet består talen av ett antal bitar och kan bearbetas med t ex aritmetiska operationer i ett digitalt system. De aritmetiska operationerna utförs olika beroende på om talen är binära tal eller om de är binärkodade decimala tal. Det är därför ytterst viktigt att man har klart för sig om en följd av 0'or och 1'or representerar ett binärt tal eller någon annan binärkod.

1.4.3 Alfanumerisk kod

När de symboler som skall kodas inte enbart är siffror utan även bokstäver (A,B,C,...), speciella operationssymboler (+,-,/,?,(,...) mm krävs en expanderad kod. Sådana koder benämns *alfanumeriska koder* och bildas ofta ur numeriska koder genom att antalet bitar i kodorden utökas. För att representera tio decimala siffror, 29 stora och små bokstäver och diverse specialtecken krävs åtminstone 7-bitars kodord. Alfanumeriska tecken kan kodas på många sätt, vilket kan ställa till problem när flera digitala system arbetande med olika koder önskas sammankopplas. Man har därför försökt standardisera koder relaterade till alfanumeriska tecken. Den i särklass mest använda alfanumeriska koden är ASCII-koden.

ASCII=
A*merican* ***S****tandard*
C*ode for* ***I****nformation*
I*nterchange*

Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII
0	NUL	20		40	@	60	‘
1	SOH	21	!	41	A	61	a
2	STX	22	"	42	B	62	b
3	ETX	23	#	43	C	63	c
4	EOT	24	$	44	D	64	d
5	ENQ	25	%	45	E	65	e
6	ACK	26	&	46	F	66	f
7	BEL	27	’	47	G	67	g
8	BS	28	(	48	H	68	h
9	HT	29	)	49	I	69	i
A	LF	2A	*	4A	J	6A	j
B	VT	2B	+	4B	K	6B	k
C	FF	2C	,	4C	L	6C	l
D	CR	2D	-	4D	M	6D	m
E	SO	2E	.	4E	N	6E	n
F	S1	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[Ä	7B	{ ä
1C	FS	3C	<	5C	\ Ö	7C	| ö
1D	GS	3D	=	5D	] Å	7D	} å
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

Tabell 1.4 ASCII-kod

Observera att endast 7 bitar används i ASCII koden. Det finns även nationella varianter där exempelvis Sverige har bytt ut nationella tecken (*ÅÄÖ*) mot befintliga tecken. Det finns också varianter på ASCII-koden

som använder 8 bitar i stället för 7, på så sätt kan man koda ytterligare 128 tecken. Den 7-bitars ASCII-koden innehåller flera speciella funktioner. Dessa används speciellt i datakommunikationsutrustning. Det är exempelvis med denna kod som tecken inslagna på en dataterminal med skrivmaskins-liknande tangentbord överförs till ett digitalt system.

1.5 Digital teknik

Digitaltekniken är grundläggande för datortekniken. Digitala kretsar används för att bygga de komplexa funktioner av logiknät som krävs exempelvis för en mikroprocessor. Denna framställning, som inte är på något sätt är avsedd att vara komplett, vänder sig till dig som inte tidigare studerat digitalteknik och syftar bara till att ge tillräcklig förståelse av digitaltekniken för att du ska kunna följa den fortsatta framställningen i denna lärobok.

1.5.1 Boolesk algebra

En Boolesk variabel kan anta något av värdena 0 eller 1. Följande operationer kan utföras :

ELLER	(OR)	logisk summa, beteckas +
OCH	(AND)	logisk produkt, betecknas •
ICKE	(NOT)	logisk invers, betecknas A’ (icke A)

För operationerna gäller följande *postulat*:

1. 0+0 = 0
2. 1•1 = 1
3. 1+1 = 1
4. 0•0 = 0
5. 0+1 = 1+0 = 1
6. 1•0 = 0•1 = 0
7. 0’ = 1
8. 1’ = 0

Som en direkt följd av dessa postulat får vi följande räknelagar för en boolesk variabel A.

1. A+A = A
2. A•A = A
3. A+A’ = 1
4. A•A’ = 0
5. A+1 = 1
6. A•0 = 0
7. A+0 = A
8. A•1 = A
9. (A’)’ = A

Operationerna kan sammanfattas i så kallade *sanningstabeller* för de olika operationerna (se marginalen föregående sida). I tabellerna anger anger A och B booleska variabler. Precis som i vanlig algebra utelämnar man tecknet • för logisk produkt om inget missförstånd kan uppstå, dvs

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Sanningstabell *logiskt ELLER*

A	B	A•B
0	0	0
0	1	0
1	0	0
1	1	1

Sanningstabell *logiskt OCH*

A	A’
0	1
1	0

Sanningstabell *logisk INVERS*

A•B = AB

För två eller flera booleska variabler gäller bl.a:

associativitet

A+(B+C)=(A+B)+C
A(BC) = (AB)C

kommutativitet

A+B = B+A
AB = BA

distributivitet

A(B+C) = AB+AC
A+BC = (A+B) (A+C)

absorption

A+AB = A
A(A+B) = A

Samtliga räknelagar kan enkelt visas med hjälp av sanningstabeller.

EXEMPEL
Visa att för de booleska variablerna A och B gäller:
A+AB = A

Lösning:
Vi har två variabler som kan anta värdena 0 och 1. Alltså behöver vi undersöka fyra fall.

A	B	Vänstra ledet A+AB	Högra ledet A
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	1

VL = HL vilket skulle visas.

Vi kan visa samma sak med algebraisk manipulation och andra räknelagar:
A+AB = A(1+B) = A (ty 1+B = 1 enligt postulat).

1.5.2 De Morgans lag

De Morgans lag ger ett enkelt samband för övergång mellan OCH respektive ELLER-funktioner:

A'B' = (A+B)' (1
A'+B' = (AB)' (2

Detta visas enkelt med en sanningstabell:

A	B	A'B'	(A+B)'	A'+B'	(AB)'
0	0	1	1	1	1
0	1	0	0	1	1
1	0	0	0	1	1
1	1	0	0	0	0

De Morgans lag kan generaliseras.

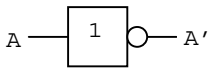
EXEMPEL:
Visa att A'B'C' = (A+B+C)'

Lösning:
Sätt A+B = X. Detta ger för högra ledet:
(X+C)' = X'C' (enligt de Morgans lag 1)
= (A+B)'C' (tillämpa samma lag igen)
= (A'B')C' = A'B'C'
vilket skulle visas.

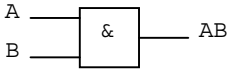
1.5.3 Grindar

Digitala komponenter, uppbyggda kring transistorer, kan realisera de logiska funktionerna. Följande symboler används i detta läromedel:

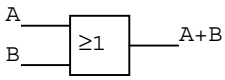
logisk INVERS **NOT**



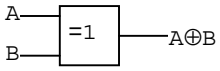
logiskt OCH **AND**



logiskt ELLER **OR**



logiskt EXKLUSIVT ELLER
EXCLUSIVE OR

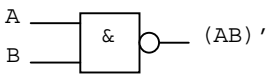


A	B	A•B
0	0	0
0	1	1
1	0	1
1	1	0

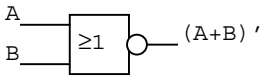
Sanningstabell *logiskt EXKLUSIVT ELLER*

OR respektive AND-grindar finns som standardkomponenter med upp till 12 ingångar. I detta läromedel förekommer också följande typer:

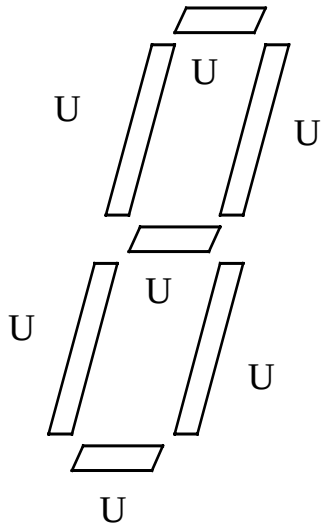
inverterad AND, **NAND**



inverterad ELLER **NOR**



1.5.4 Grindnät



7-sifferindikator, förenklad beskrivning

Flera grindar kan kombineras i ett nät för att bygga upp mer eller mindre komplexa funktioner. Vi kan oftast betrakta ett sådant nät som en "svart låda" med insignaler, utsignaler (booleska variabler) och någon transformation. Låt oss bara illustrera detta med ett enkelt exempel. Vi tar som exempel en så kallad "7-sifferindikator" som används i bland annat radioapparater med digital visning, armbandsur osv. Indikatorn består av sju stycken segment (därav namnet) som kan tändas och släckas individuellt. Antag nu att utsignalen $U_n = 1$ representerar att segment *n* är tänt och utsignalen $U_n = 0$ representerar ett släckt segment (se figur i marginalen). Vi vill nu konstruera ett grindnät som gör det möjligt att direkt visa de binära siffrorna (0000-1001) som "0", "1" .."9" på indikatorn. För binära insignalen 0000 vill vi då tända samtliga segment utom U_4 osv. Vi sammanfattar utsignalerna för de första fyra fallen i följande tabell:

b3	b2	b1	b0	U1	U2	U3	U4	U5	U6	U7
0	0	0	0	1	1	1	0	1	1	1
0	0	0	1	0	0	1	0	0	1	0
0	0	1	0	0	1	1	1	1	0	1
0	0	1	1	0	1	1	1	0	1	1

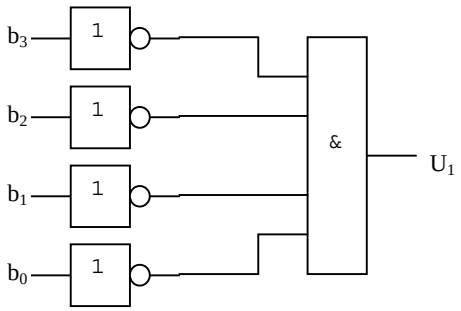
Låt oss nu betrakta utsignalerna en i taget. För U_1 ser vi att denna utsignal skall vara 1 *endast* om bitarna b_3,b_2,b_1 och b_0 är 0. Vi kan skriva detta som:

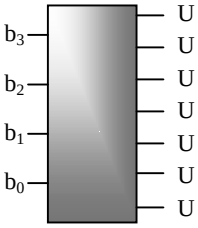
$$U_1 = b_3' \, b_2' \, b_1' \, b_0'$$

Av tabellen ser vi vidare att U_2 skall vara 1 för tre olika kombinationer av insignalerna. Vi kan skriva detta som:

$$U_2 = b_3' \, b_2' \, b_1' \, b_0' + b_3' \, b_2' \, b_1 \, b_0' + b_3' \, b_2' \, b_1 \, b_0$$

På samma sätt tecknas uttryck för samtliga signaler U_n. Implementeringen kan nu göras direkt från dessa uttryck, för U_1 får vi exempelvis:





Vi kan beskriva hela konstruktionen med en enkel figur (se marginalen) kompletterad med tabellen ovan som utökats med uttryck för utsignalerna för samtliga förekommande värden på insignalerna.

1.6 Övningsuppgifter

Uppgift 1: Utför följande omvandlingar mellan talsystemen där talbasen anges med prefix för Motorola assembler.

```
a) 4       = %___________=$_____
b) 8       = %___________=$_____
c) 7       = %___________=$_____
d) 15      = %___________=$_____
e) 30      = %___________=$_____
f) 128     = %___________=$_____

g) %110  = $________=________
h) %1100 = $________=________
i) %1110 = $________=________
j) %10101= $________=________
k) %11010= $________=________
l) %11111= $________=________

m)$5     = %___________=______
n)$9     = %___________=______
o)$A     = %___________=______
p)$12    = %___________=______
q)$BB    = %___________=______
r)$C0    = %___________=______
```

Uppgift 2: Ange binärt de NBCD-tal som motsvaras av följande decimala tal.

```
a) 22 = %(NBCD) ___________
b) 18 = %(NBCD) ___________
c) 30 = %(NBCD) ___________
d) 75 = %(NBCD) ___________
e) 54 = %(NBCD) ___________
f) 69 = %(NBCD) ___________
```

Uppgift 3: Ange, hexadecimalt, koden för följande ASCII-tecken.

```
a) a = $_____
b) Z = $_____
c) ! = $_____
d) 7 = $_____
e) , = $_____
f) ? = $_____
```

Uppgift 4: Ange decimala motsvarigheten till följande tal givna på *binärform*.

```
a) %01010101 = _______
b) %00111100 = _______
c) %10101010 = _______
```

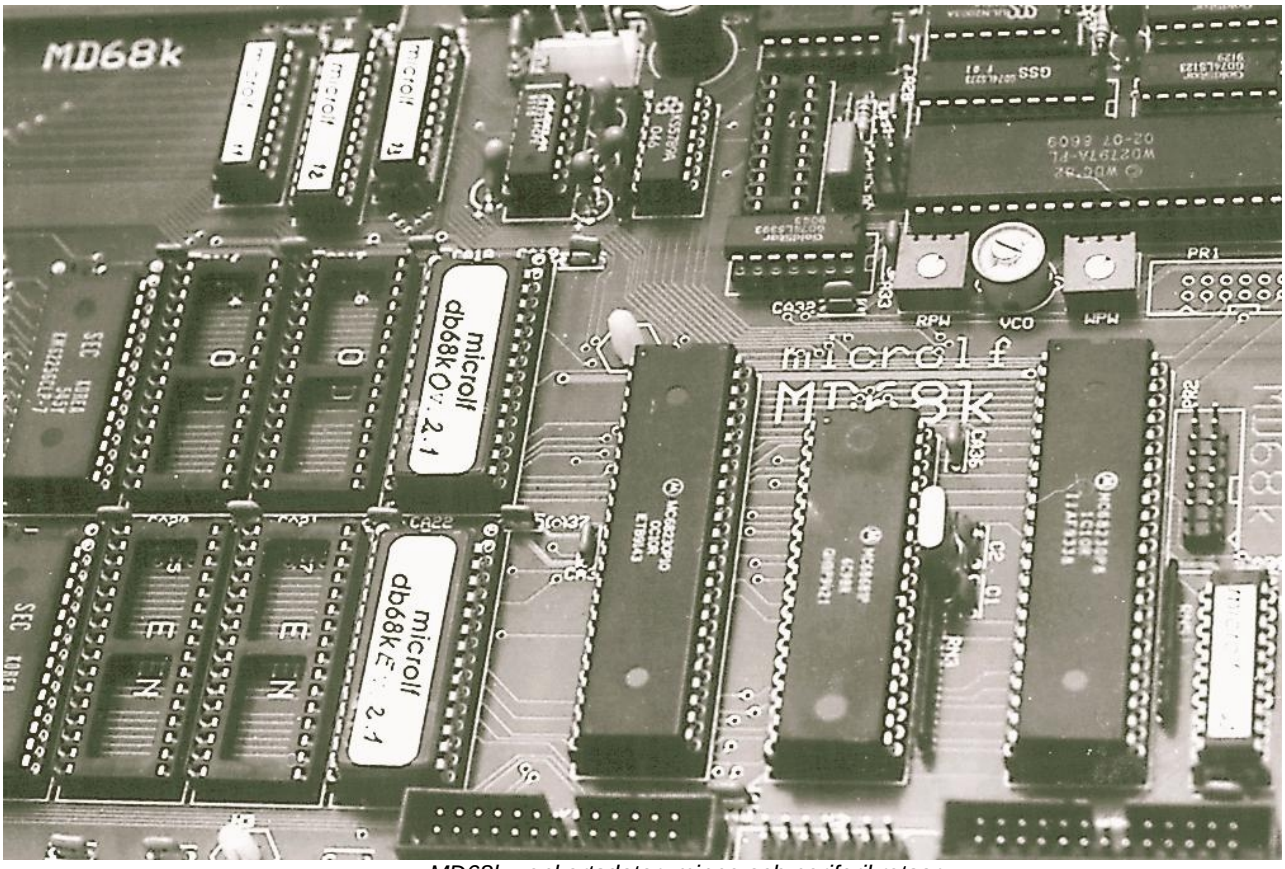
Uppgift 5: Visa att för de booleska variablerna A och B gäller A+BC=(A+B)(A+C).

Uppgift 6: Visa att för de booleska variablerna A och B gäller A(B+C)=AB+AC.

Uppgift 7: Ange utsignalerna U_n för insignalerna A,B och C (följande tabell) på boolesk form.

A	B	C	U1	U2	U3	U4	U5	U6	U7
0	0	0	1	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0
0	1	0	0	1	0	1	0	0	1
0	1	1	0	0	1	1	0	0	1
1	0	0	1	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	1	0

2. Mikrodatorns uppbyggnad



MD68k - enkortsdator, minne och periferikretsar

Inledningsvis nämnde vi några av de block som används för att bygga upp ett datorsystem. Vi har sett hur exempelvis ett styrsystem kan illustreras med sådana block. I detta kapitel ska vi titta närmare på dessa block och behandla de grundläggande principerna för hur de sätts samman till ett fungerande system.

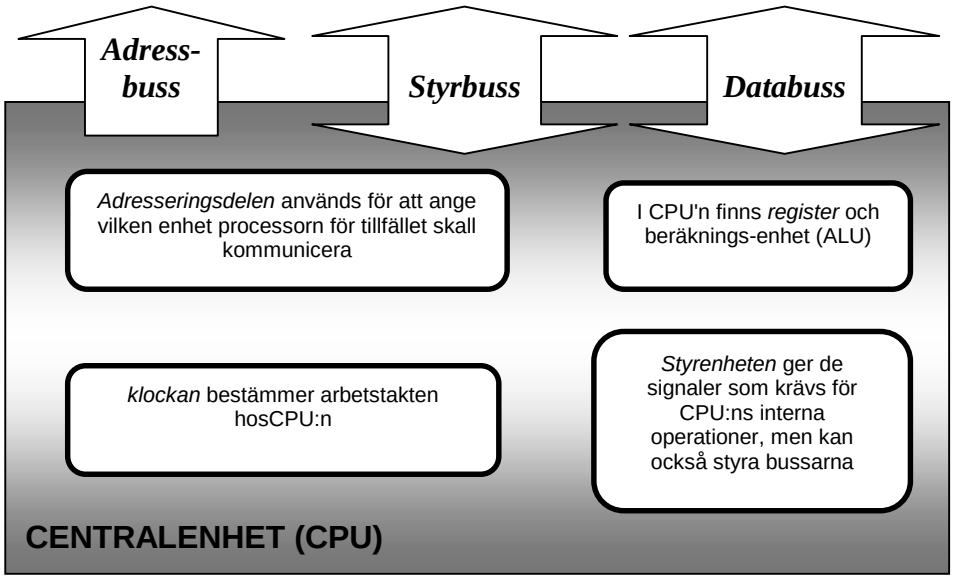
Av detta kapitel kan du lära dig:

- ange de viktigaste delarna i en CPU
- förklara användningen av databuss, adressbuss respektive styrbuss.
- känna till begreppen asynkron, synkron respektive multiplexbuss
- kortfattat redogöra för funktionen hos en minneskrets
- förklara begreppen RWM/ROM, flyktigt respektive icke-flyktigt minnesinnehåll
- förklara begreppen byte, kilobyte och megabyte.
- kortfattat beskriva uppbyggnaden av en dators minnessystem
- redogöra för olika typer av programmerbara minneskretsar

2.1 Centralenheten

Från centralenheten kontrolleras och styrs det mesta som sker i systemet. Centralenheten utför de sekvenser av enkla operationer som programmet i datorsystemet anger. Centralenhetens arbetstakt styrs av en extern *klocka* (kristalloscillator). Arbetstakten bestäms framför allt de använda komponenterna. Ju högre arbetstakt man kräver, desto dyrare blir som regel komponenterna i systemet.

Internt är centralenheten uppbyggd av en styrenhet, en databearbetningsdel och en adresseringsdel. Styrenheten hämtar instruktioner från minnet och utför dessa. Databearbetningsdelen innehåller register för att mellanlagra data och en så kallad ALU (Aritmetisk-Logisk Enhet) som kan utföra additioner subtraktioner, mm. Slutligen finns adresseringsdelen som ser till att adressera de övriga delarna anslutna till busssystemet när dataöverföring skall ske. Figur 2.1 visar en första bild av hur en centralenhet är uppbyggd.



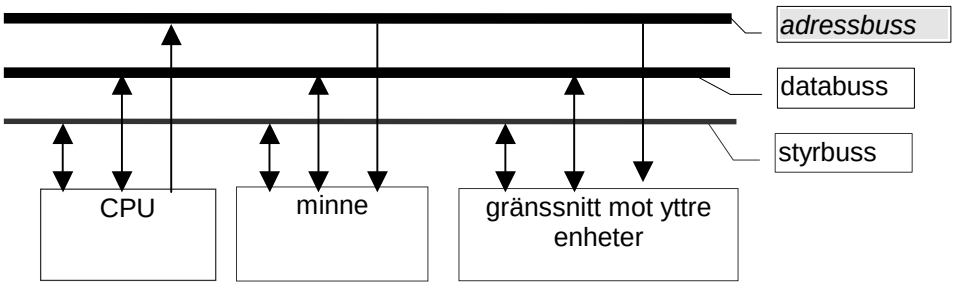
Figur 2.1 *Centralenheten*

Vi återkommer till en detaljerad beskrivning av centralenheten och dess arbetssätt i kapitel 4.

2.2 Bussarna

Datorn är uppbyggd av ett antal enheter som kan kommunicera med varandra via elektriska ledare. En grupp av sådana ledare kallas för *buss*. Ofta talar man om ett datorsystems bussar och menar då just ledarna som kopplar samman CPU, minne och gränssnitten. Beteckningen gränssnitt är en gemensam beteckning på enheter som in- och utportar som inte (elektriskt) direkt kan anslutas till CPU:s busssystem.

Allt informationsutbyte som sker i datorsystemet görs via tre bussar: *databuss, adressbuss* och *styrbuss*.



Figur 2.2 *Datorns bussar*

Databuss
Databussen används, precis som namnet säger, för att överföra data mellan CPU och minne eller mellan CPU och in-/ut-matningsenhet. Den används också för att överföra instruktioner från minnet till CPU'n. Eftersom data kan överföras både *till* och *från* CPU'n via databussen är denna dubbelriktad.

Adressbuss
Adressbussen används av CPU'n för att peka ut (*adressera*) olika delsystem i datorn. Exempelvis för att adressera datorsystemets inport när CPU'n skall hämta indata. Observera att CPU'n styr denna buss och pilen är därför riktad *ut* från CPU:n.

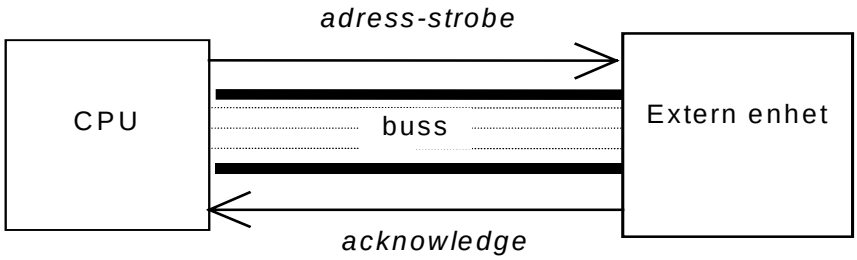
Styrbuss
I denna grupp ingår samtliga styrsignaler som krävs för kommunikationen mellan enheterna anslutna till bussystemet. Ett exempel på en styrsignal är skriv- och lässignalen (*R/W, Read/Write*) där CPU'n anger att den önskar att exempelvis läsa (hämta data från) minne eller gränssnitt. Mestadels är det CPU:n som styr denna buss, men styrsignaler kan även genereras av de övriga enheterna i datorsystemet. Som Figur 2.2 visar är styrbussen därför dubbelriktad.

Vi skall nu beskriva några *olika typer* av busssystem och hur kommunikationen (dataöverföringen) går till. En buss kan vara *asynkron* eller *synkron*. Det finns också så kallade *multiplexade* busssystem.

2.2.1 Asynkron Buss

Med en *asynkron* buss används så kallade *handskakningssignaler* för kommunikationen (dataöverföringen). Med detta menas att CPU:n exempelvis kan signalera till de övriga enheterna att en adress nu finns på bussen genom att “slå på” en speciell signal (*adress-strobe*). Denna signal kan tolkas som “Nu finns det en giltig adress på adressbussen. Känner någon igen denna?”. Nu kan de övriga enheterna läsa av adressbussen och därefter kan den enhet som känner igen adressen generera en annan signal

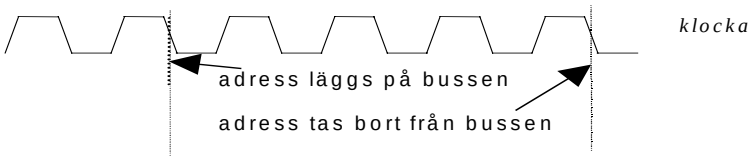
tillbaks till CPU:n (*acknowledge*), för att på så sätt tala om för CPU:n att adressen är igenkänd. Denna signal kan tolkas som “OKAY, det är min adress. Nu överför vi data på databussen.”. Följakligen inväntar CPU:n svar från den enhet den önskar kommunicera med.



Figur 2.3 *Asynkron Buss*

2.2.2 Synkron Buss

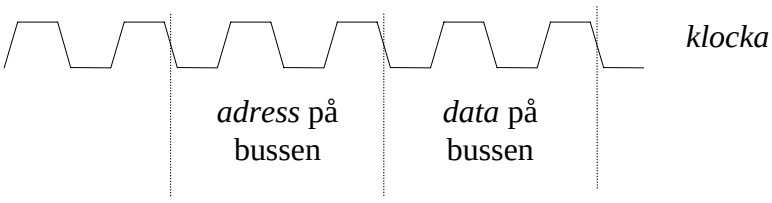
En buss sägs arbeta *synkront* om signalerna överförs vid en förutbestämd tidpunkt. Fördelen med synkron buss är att handskakningssignaler inte behövs längre. Nackdelen är att det kan vara svårt att blanda olika typer av minnen etc, eftersom dessa har olika tidsegenskaper. Vid dataöverföring på en synkron buss *förutsätter* CPU:n att den andra enheten hinner med i den arbetstakt som används. Detta innebär att det är konstruktören av systemet som har ansvaret för att alla komponenter som anslutits till busssystemet verkligen hinner med i CPU:ns arbetstakt.



Figur 2.4 *Synkron buss*

2.2.3 Multiplex buss

Multiplex-buss kallas en buss där *samma* elektriska ledare används för att både adressera och överföra data mellan CPU och övriga enheter. Det krävs då en del extra logik för att kunna bestämma exakt när bussen anger en adress respektive när bussen anger data. Fördelen med denna typ av buss är förstås att den kräver betydligt färre ledare, och därmed färre anslutningar (pinnar) på kretsarna. Nackdelen är att kommunikationen blir långsammare och extra elektronik krävs. Vid överföring på en mulitplexad buss överförs alltså *först* adressen, *sedan* data.

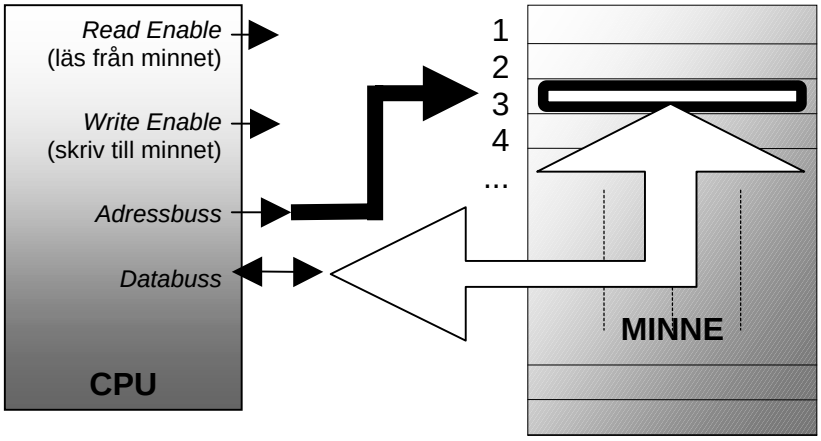


Figur 2.5 *Multiplex buss*

2.3 Minnet

Ett minne skulle kunna jämföras med en väldigt stor byrå. I byrån finns ett stort antal lådor. Vi kan numrera dessa lådor, exempelvis 1,2,3,4 osv. Varje låda har alltså ett *unikt* nummer. Om vi vill kan vi kalla detta nummer för lådans *adress*. I varje låda (som inte är så stor) kan vi placera ett mycket begränsat innehåll. Varje låda rymmer exakt lika mycket. Vi kan bara öppna *en* låda i taget.

Utifrån denna bild kan vi betrakta kommunikationen mellan centralenhet och minne. Vid en *läsning* från minnet måste alltså centralenheten först generera en adress (ange låda), därefter en signal som talar om för minnet att lådan ska öppnas för läsning av innehållet. Minnet kan då plocka fram innehållet i lådan och placera detta på *databussen* så att centralenheten kan läsa in detta. På motsvarande sätt kan centralenheten också generera en signal som säger minneskretsen att ett minnesinnehåll ska *modifieras* dvs, ges ett nytt innehåll. Man säger då att centralenheten skriver i minnet.



Figur 2.6 *Centralenhet - Minne*

Datorsystemets minne består av *primärminne* och *sekundärminne*. I primärminnet lagras det program (dvs de maskininstruktioner) med tillhörande dataareor som för tilllfället används. Sekundärminnet är, som namnet antyder, en sekundär lagringsplats för program och data. Primärminnet kan bestå av olika typer av minneskretsar, medan sekundärminnet oftast är en hårddisk.

När det gäller minnets storlek så brukar man ange detta i antal *bytes*. Med en *byte* menar man 8 databitar. Om vi här förutsätter att det ryms en byte på varje adress i minnet (”en byte per låda”) och att centralenheten kan bearbeta en byte åt gången, har vi ett 8-bitars system. Man säger då att *ordbredden* (eller *ordlängden*) i systemet är 8 bitar. De vanligase datorsystemen i dag använder 8, 16 eller 32-bitars ordbredd men större ordbredder förekommer också.

I dag är det vanligt med minneskretsar som rymmer 4-16 miljoner bytes Hårddiskar finns med en lagringskapacitet på flera tusen miljoner bytes.

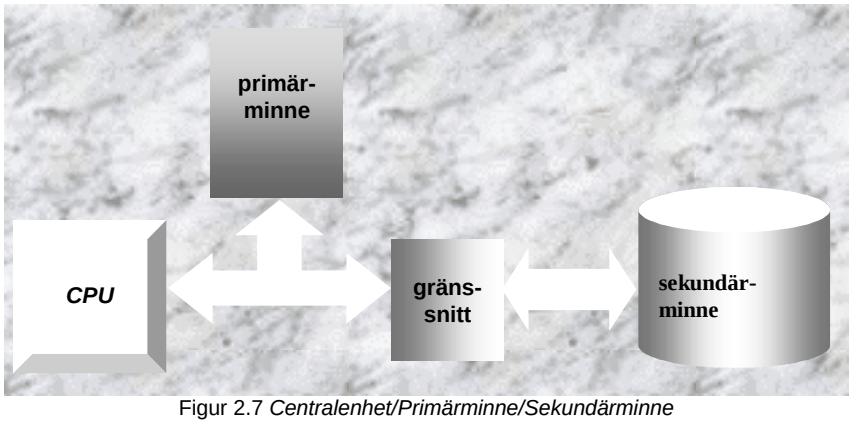
1 byte	=8 bitar		
1 Kbyte	=1.024 bytes	=8.192 bitar	
1 Mbyte	=1.000 kbyte	=1.048.576 bytes	
1 Gbyte	=1.000 Mbyte	=1.048.576 kbytes	=1.073.741.824 bytes

Ofta används (oegentligt) beteckningen "RAM" för det vi här kallar RWM. Beteckningen RAM står för Random Access Memory vilket egentligen bara betyder att åtkomsttiden i minnet är den samma oavsett vilken minnescell (låda) som adresseras.

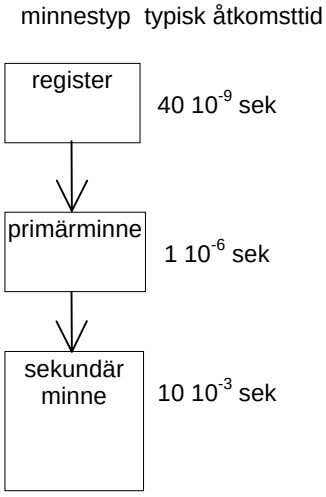
Primärminnet består vanligen av *PROM* (*Programmable Read Only Memory*) och *RWM* (*Read Write Memory*). Fördelen med *PROM*:et är att när det väl har blivit programmerat så behåller det sin information oberoende om datorsystemet är spänningssatt eller ej. Nackdelen är att informationen inte kan ändras under vanlig programkörning. Innehållet i *RWM* kan däremot ändras av programmet men *RWM* mister sin information vid spänningsbortfall.

2.3.1 Minnessystem

Såväl program (instruktioner) som data kan lagras på flera olika sätt: internt i centralenheten (register), primärminnet (RWM-/ROM- kapslar) men också på så kallat sekundärt minne (hårddisk). Organisationen av minnesenheter i ett datorsystem kallas *minneshierarki*.



Figur 2.7 *Centralenhet/Primärminne/Sekundärminne*



Med minneshierarki menas att man delat in minnestyper efter *snabbhet*. Den snabbaste åtkomsten av data får man då data finns i register (internt i centralenheten), därför placerar man register (som minnestyp) överst i minneshierarkin.

Ett minne som är direkt åtkomligt via bussen för processorn kallas alltså primärminne. Primärminnen är idag normalt tillverkade av halvledare. I datorernas barndom förekom magnetiska *kärnminnen*.

De minnen som inte är direkt åtkomliga kallas sekundär-minnen och nås av processorn via in- och ut-portar. Priset för att lagra en bit på dessa är bara en bråkdel av priset för primärminneslagring. När vi talar om ett datorsystems minne menar vi i första hand *primärminnet*.

Minnets tillgänglighet, dvs möjligheten att läsa eller skriva data, kännetecknas av metoden som används för läsning eller skrivning och av den tid som åtgår (åtkomstid, *access time*). Med *direkt åtkomst* (*Random*

Access) menas att åtkomstiden är densamma för varje minnesord oberoende av *var* i minnet dessa finns.

Minnen kännetecknas också av sin förmåga att behålla den lagrade informationen. Om minnesinnehållet försvinner vid en läsning kallas detta för ett raderande läsminne (*destructive readout*). Motsatsen kallas för ett icke raderande läsminne (*non-destructive read out*). Vid spänningsbortfall försvinner innehållet i ett flyktigt minne (*volatile*), medan ett icke-flyktigt minne (*non-volatile*) behåller sitt innehåll. En tredje karakteristik hos minnen är effektförbrukningen som varierar för olika typer av minnen.

2.3.2 Sekundärminne

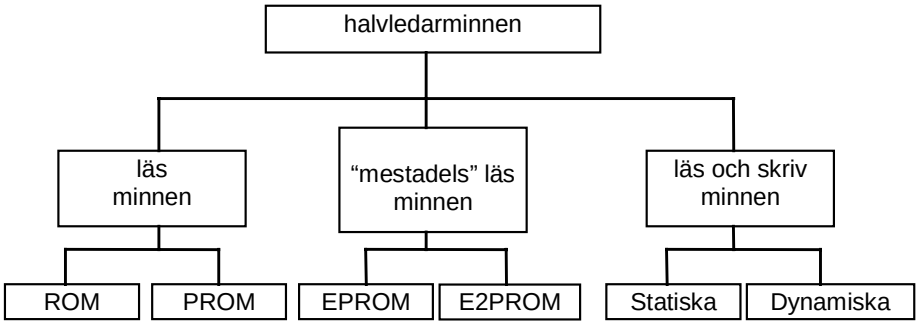
Sekundärminnet räknas vanligtvis till datorns periferiutrustning. Det fungerar både som inmatningsenhet och utmatningsenhet. En speciell egenskap med sekundärminnen är att data lagras i *block* av ord. Detta innebär att, till skillnad från primärminnet, enstaka ord inte kan läsas. I stället måste ett helt block överföras från sekundärminne till primärminne för att därefter bearbetas. På samma sätt gäller att endast hela block kan *skrivas* till sekundärminnet.

2.3.3 Cacheminne

Ofta placerar man ett så kallat *cacheminne* (ung: *fickminne*) mellan primärminne och centralenheten i minneshierarkin. Detta minne fungerar precis som RWM i primärminne med den skillnaden att åtkomsttiden är betydligt lägre. Block av data flyttas mellan primärminne och cacheminne av en så kallad *cache-controller*. På detta sätt kan den totala tiden för läsningar i minnet reduceras avsevärt och därmed förbättras också systemets prestanda.

2.3.4 Primärminnen

Dagens primärminnen består i huvudsak av minneskretsar av halvledartyp. I gruppen halvledarminnen ingår flera olika "familjer" med olika funktion och prestanda. I Figur 2.8 ges en översiklig bild av de olika familjerna.



Figur 2.8 *Halvledarminnen*

I figuren introduceras en rad termer som är vanliga i datorsammanhang. Låt oss diskutera dessa var och en för sig.

Läsminnen

Gruppen "läsminnen" kan, som namnet antyder, endast läsas. Den engelska benämningen är Read Only Memory (ROM). Dessa minnen programmeras vid tillverkningen och innehållet kan därefter inte ändras. Vid tillverkningen av ROM skapas minnets innehåll, dvs ettor och nollor, med hjälp av en fotografisk mask, vars utseende bestäms av användaren. Alltså måste beskrivningen på hur minnet ska programmeras lämnas till tillverkaren som först tillverkar masken och därefter minneskretsarna. ROM-kretsar tillverkas oftast i stora serier (mer än 5000 enheter).

En annan variant kan programmeras av användaren, då med speciell utrustning som är avsedd för ändamålet, så kallad “PROM-brännare”. Detta är de så kallade *Programmable Read Only Memories*. Då kretsen programmerats kan dess innehåll inte längre ändras.

Mestadels läsminnen

Minneskretsar som kan *raderas* och därefter programmeras på nytt kallas EPROM (*Erasable Read Only Memories*). Det finns flera olika typer av EPROM-minnen och den största skillnaden är sättet att radera minnets innehåll. En vanlig typ är UV-EPROM, vars innehåll kan raderas med ultra-violett strålning. Över själva kretsen sitter en genomskinlig platta. Det tar cirka 10 minuter att fullständigt radera en krets och detta kan utföras med speciella UV-raderare. Efter programmering är det viktigt att täcka över den genomskinliga plattan, exempelvis med en pappersetikett, så att inte minnet, eller delar av minnet, ofrivilligt raderas av solljus eller UV-strålningen från ett lysrör. EPROM är enkla att programmera med hjälp av relativt billig utrustning som exempelvis kan anslutas till en persondator. EPROM kretsar är den billigaste typen av raderbara/programmerbara kretsar, de finns dessutom i en rad olika storlekar (8,16,32,64,128 och 512 kByte) och de är därför mycket vanliga.

Så kallade “E-två-PROM” (E2PROM) eller egentligen EEPROM, som står för *Electric Erasable Programmable Read Only Memories* kan raderas och programmeras elektriskt och fodrar därför inte någon speciell utrustning vare sig för radering eller programmering. Kretsen är betydligt dyrare än EPROM men den korta tiden att radera och programmera gör att E2PROM är lämplig för prototyper.

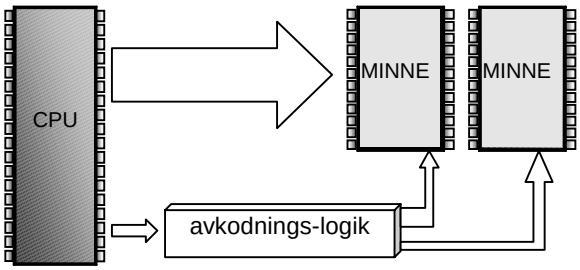
Den senaste typen av icke-flyktigt minne är FLASH. Minnet är, precis som E2PROM, elektriskt raderbart, men betydligt billigare. FLASH-minnen är på god väg att bli den dominerande typen av icke-flyktigt minne. Innehållet kan raderas av en speciell programsekvens och därefter enkelt programmeras. Ingen speciell programmeringsutrustning krävs för detta.

Läs och skriv minnen

Läs- och skrivminnens innehåll är flyktigt, två olika typer av RWM-minnen är vanliga idag: *statiska minnen* och *dynamiska minnen*.

2.4 Adressavkodning

Vi har tidigare sett hur centralenheten kan kommunicera med såväl en minneskrets som ett gränssnitt via systemets bussar. Vad vi inte diskuterat är hur man kan vara säker på att rätt minneskrets eller rätt gränssnitt adresseras. Oftast består ju datorn av flera minneskapslar, och vi såg i exemplet med varmvattensberedaren att flera gränssnitt typiskt används. I detta avsnitt ska vi redogöra för hur ett komplett datorsystem, med centralenhet, två minneskretsar och två gränssnitt kan åstadkommas. Vi gör detta genom att konstruera ett block bestående av grindar. Detta logikblock kallas *adress-avkodningslogik*.

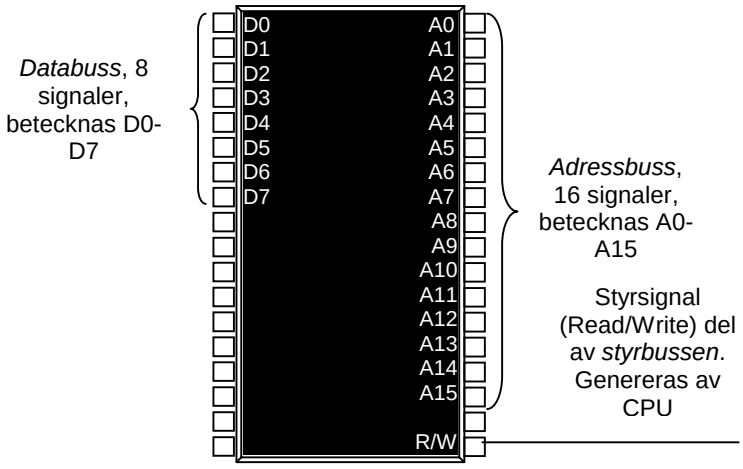


Figur 2.9 *Adressavkodningslogikens placering*

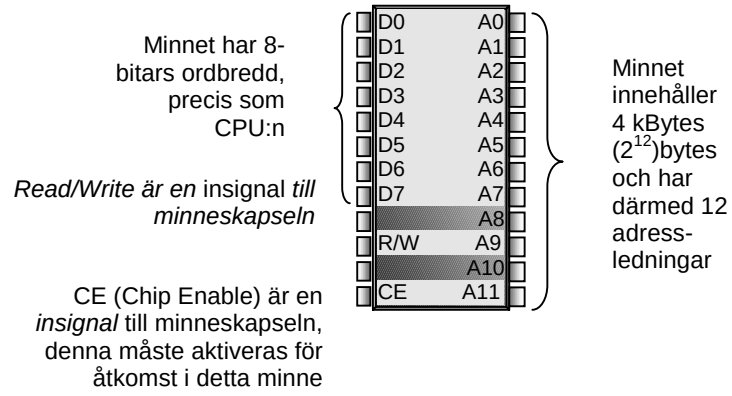
För exemplet antar vi att vi har tillgång till följande komponenter.

- Centralenhet, 64kByte adressrum, 8-bitars ordlängd.
- Minneskapsel RWM 4 kByte
- Minneskapsel ROM 4 kByte
- Register, 8 bitar som inport.
- Register, 8 bitar som utport.
- Standardkretsar med diverse grindar.

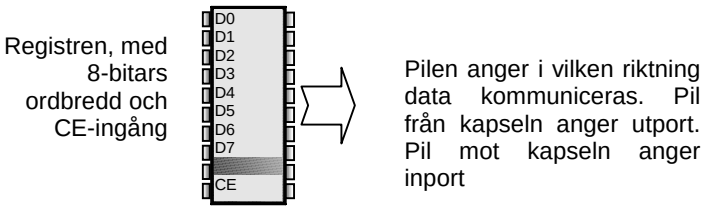
Med de givna förutsättningarna kan vi ge något förenklade bilder av de ingående komponenterna. I dessa bilder utelämnar vi signaler och komponenter som inte behövs för adressavkodningen.



Figur 2.10 *Förenklad bild av centralenheten*



Figur 2.11 *Förenklad bild av RWM. ROM-kapseln är snarlik, den saknar dock R/W-signalen*



Figur 2.12 *Förenklad bild av Register*

Minnesdisposition

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

Kapsel	Startadress	Slutadress
Minneskapsel RWM 4 kByte	0000	1FFF
Minneskapsel ROM 4 kByte	C000	DFFF
Register, 8 bitar som inport	A000	A000
Register, 8 bitar som utport	8000	8000

Tabell 2.1 *Minnesdisposition*

Av minnesdispositionen i Tabell 1.1 framgår att:

- RWM-minnet ska aktiveras om CPU:n genererar någon av adresserna 0-3FFF.
- ROM-minnet ska aktiveras om CPU:n genererar någon av adresserna C000-DFFF.
- Inporten ska aktiveras om CPU:n genererar adress A000.
- Utporten ska aktiveras om CPU:n genererar adress 8000.

Tabell 2.2 visar de värden adressbussens signaler får anta för respektive kapsel. ”X” i tabellen anger ”don’t care” dvs signalen kan vara antingen 0 eller 1.

Kapsel	Adressbuss															
	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
RWM	0	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X
ROM	1	1	0	0	X	X	X	X	X	X	X	X	X	X	X	X
inport	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
utport	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabell 2.2 *Adressbussens nivåer för respektive kapsel*

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

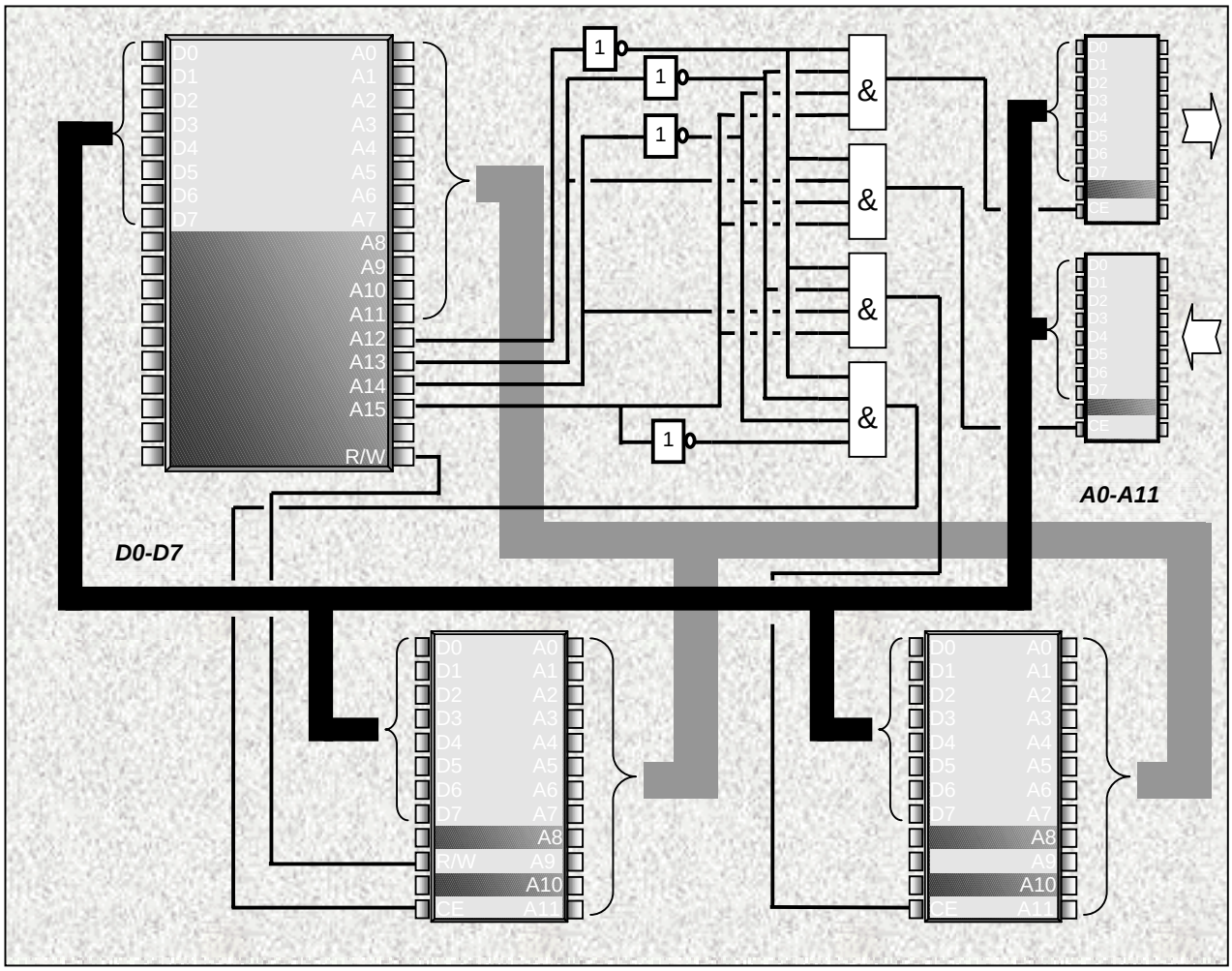
Av Tabell 2.2 framgår att vi kan göra en unik CE-signal för varje kapsel genom att betrakta endast A15, A14 och A13. För exempelvis RWM-kapseln ska dessa tre adressignaler vara 0, för ROM-kapseln ska vi ha A15=1, A14=1 och A13=0, osv. Om vi väljer att låta bara dessa signaler ingå i adressavkodningen får vi minsta möjliga kretslogik. Detta är dock en *ofullständig adressavkodning* eftersom vi då lämnar A12 som ”don’t care”. Resultatet blir att CE-signalen blir aktiv för ett större adressintervall än det som avsatts för respektive kapsel.

I vårt exempel väljer vi en kompromiss. Vi avkodar adresser till de båda minneskapslarna fullständigt men nöjer oss med en ofullständig adressavkodning för portarna. I så fall kan vi göra denna avkodning på samma sätt som för minneskretsarna. Nackdelen är dock att inporten inte längre har *en* unik adress utan aktiveras för *varje* adress i detta adressrumsintervall. Tabell 2.3 visar vårt slutliga val av adressavkodning

Kapsel	Adressbuss				
	A15	A14	A13	A12	Adressintervall
RWM	0	0	0	0	0000-1FFF
ROM	1	1	0	0	C000-DFFF
inport	1	0	1	0	A000-BFFF
utport	1	0	0	0	8000-9FFF

Tabell 2.3 *Adressignaler som ingår i adressavkodningen*

Figur 2.13 nedan visar kopplingarna för adressavkodninglogiken. Dataledningarnas anslutningar (D0 till D0, D1 till D1 osv) är markerade av en svart buss. På motsvarande sätt har adressledningar A0-A11 markerats av en grå buss. Utöver dessa adressledningar, databussen och CE-signalerna ansluts även R/W mellan CPU och RWM.



Figur 2.13 *Kopplingar för adressavkodningslogiken*

2.5 Övningsuppgifter

Uppgift 8: Beskriv den principiella skillnaden mellan en *asynkron* buss och en *synkron* buss.

Uppgift 9: Hur kommuniceras adress respektive data på en *multiplexbuss*?

Uppgift 10: Vad menas med begreppen *flyktigt* (RWM) respektive *icke-flyktigt* (ROM) minne?

Uppgift 11: Hur många *bytes* är:

a) 2 kByte
b) 4 kByte
c) 8 kByte

Uppgift 12: Hur stor adressbuss (dvs hur många adressledningar) krävs för att en CPU ska kunna adressera:

a) 8 kByte
b) 64 kByte
c) 1 MByte

3. Grundläggande datoraritmetik

En dator kan bara behandla “nollor” och “ettor”. eller mer precist, en låg signalnivå eller en hög signalnivå. Eftersom den större delen av uppgifter en dator utför är beräkningar är det därför lämpligt att kunna utföra dessa beräkningar, enbart med hjälp av “nollor” och “ettor”. I detta kapitel behandlar vi binär aritmetik, dvs sådana metoder som används för att utföra beräkningar i en dator.

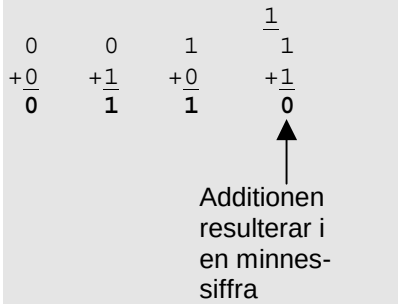
3.1 Inledning

Tänk dig att du kör en gammal bil och studerar vägmätaren. Bilen har nu gått exempelvis 99995 km. du kör några kilometer längre och plötsligt står mätaren på 00000 km. Detta är uppenbarligen fel, och orsaken är självklar: Antalet siffror på vägmätaren räcker inte för att representera 100000 km. Det finns ingen sifferposition för ettan längst till vänster och den faller därför bort. Om man bara tittar på vägmätaren kan man därför luras att tro att bilen inte har körts alls.

Binära ord med ordlängden 8 bitar och 4 bitar är mycket vanliga. Ett 8-bitars ord kallas en ***byte*** och ett 4-bitars ord kallas en ***nibble*** . Dessutom används beteckningen ***word*** för ett 16-bitars ord och beteckningen ***long*** för ett 32-bitars ord

I datorer används dataregister med fixt antal siffror för att lagra talvärden och för att utföra beräkningar. Därför kan samma fenomen inträffa i datorer. Ökar man innehållet i ett av datorn's dataregister med ett, tillräckligt många gånger så kommer resultatet till slut att bli så stort att det inte ryms i dataregistret. En dator arbetar alltså med ett fixt antal siffror (nollor och ettor), exempelvis 8, 16 eller 32 st. Vi säger att den har en *ordlängd* på 8, 16 eller 32 siffror. Dataregistren rymmer alltså lika många siffror som ordlängden anger.

Addition i binära talsystemet

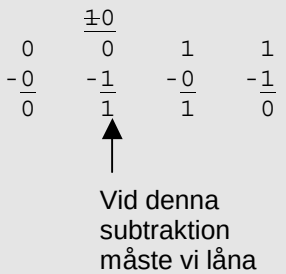


Hur kan man uttrycka *negativa* tal i en dator? I processorn förekommer som sagt endast två olika siffror (noll och ett), det finns inget minustecken. Vi ska se hur man kan låta vissa kombinationer av nollor och ettor representera den positiva talaxeln (positiva tal) och andra kombinationer representera den negativa talaxeln (negativa tal). En bitsträng av nollor och ettor kan då *tolkas* till ett bestämt talvärde där *olika* tolkningar ger olika talvärden för en given bitsträng.

3.2 Det binära talsystemet

Det binära talsystemet, som vi i viss utsträckning redan behandlat, är ett positionssystem med basen två och siffrorna 0 och 1. Det används för att koda siffror och talvärden i andra talsystem till binära tal, så att de kan bearbetas i ett digitalt system. Även mätvärden för fysikaliska storheter erhållna genom analog-digitalomvandling (se kapitel 6) utgörs normalt av binära tal.

Subtraktion i binära talsystemet



I stället för termen binär siffra används ofta ordet *bit* (eng. binary digit). Det binärkodade talet $(10111)_2$ sägs exempelvis ha fem bitar.

I digitala sammanhang kallas en grupp binära siffror ofta för ett *binärt ord* eller bara ett *ord*. Ofta föregås det också av en del som identifierar ordets innehåll, t ex *kod-*, *data-*, *instruktions-* eller *minnes-* ord. Antalet bitar i ett ord kallas *ordlängden*.

Aritmetiska operationer utförs i det binära talsystemet enligt i princip samma regler som i det decimala. Eftersom det bara finns två siffror är additions- och subtraktions- tabellerna mycket enkla, se marginalen.

EXEMPEL Addition (R=X+Y), X=7, Y=5

```
      Decimalt     Binärt
                              Minnes-
                   1  1  1  ← siffror
 X       7         0  1  1  1
+Y     + 5         0  1  0  1
---    ---         -------
=R     =12         1  1  0  0
```

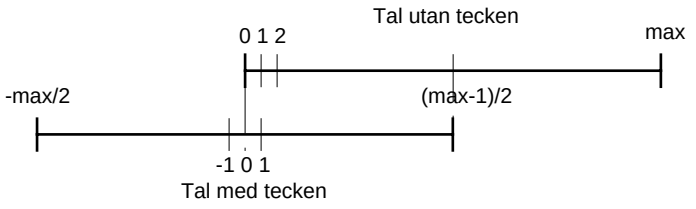
EXEMPEL Subtraktion (R=X-Y), X=6, Y=5

```
      Decimalt           Binärt
                                  Låne-
                            10 ← siffror
       X       6          0  1  1  0
      -Y     - 5          0  1  0  1
      ---    ---          -------
      =R     = 1          0  0  0  1
```

Vi kommer inte att behandla multiplikation och division i binära talsystemet i detta läromedel.

3.3 Tecken-Beloppsrepresentation

I datorer kan vi enbart använda nollor och ettor för att representera tal, någon binärkod är därför naturlig. Detta medför att om vi vill representera tal med *tecken,* så måste vi på något sätt indikera om talet är negativt eller positivt. Vi kan exempelvis koda talen så att vissa representerar negativa tal och andra positiva. Detta innebär att hälften av de binära bitmönstren går åt för att representera negativa tal. Om vi väljer att representera tal med tecken så kan vi därför endast representera (till beloppet) hälften så stora tal jämfört med om vi önskar representera tal utan tecken. Se figur 2.1. Som vi ser så är ungefär hälften av talen positiva och hälften negativa när vi önskar representera tal med tecken. Då vi väljer att tolka en bitsträng, uppbyggd av ettor och nollor, har vi också valt en *talrepresentation*.



Figur 3.1 Talområde för tal med och utan tecken

Ett enkelt sätt är att låta bitsträngens mest signifikanta bit representera talets tecken, denna bit kallas av denna anledning *teckenbit*. Teckenbiten är 0 om talvärdet är större eller lika med noll, teckenbiten är 1 om talvärdet är mindre än noll. Resterande bitar anger talets belopp.

EXEMPEL:

Bitsträng	Talvärde-Tecken-Belopp
0101	+5, ty teckenbiten är 0, talvärdet av bit 0-2 är 5
1101	-5, ty teckenbiten är 1, talvärdet av bit 0-2 är 5

Då aritmetik utförs på tal med denna representation måste tecknen betraktas speciellt. Tabell 3.1 anger hur additionen A+B utförs för olika tecken på talen A och B. För enkelhets skull förutsätter vi att A och B, till beloppet är så små, att spill inte uppträder i resultatet. Den aritmetiska operationen som utförs beror av talens tecken.

Relation A,B	Operation A+B
A,B≥0	|A| + |B|
A≥0, B<0, |A| > |B|	|A| - |B|
A≥0, B<0, |A| < |B|	-(|B| - |A|)
A<0, B≥0, |A| > |B|	|B| - |A|
A<0, B≥0, |A| < |B|	-(|A| - |B|)
A,B<0	-(|A| + |B|)

Tabell 3.1 *Räkneregler för addition av tecken-beloppsrepresenterade tal*

Av räknereglerna ser vi att ett logiknät för att utföra operationen blir ganska omfattande. Ett sådant logiknät måste klara av såväl addition som subtraktion och teckenöverläggning.

3.4 Tvåkomplementsrepresentation

Vi vill självfallet att samma räknelagar skall gälla både för tal med och utan tecken. På så sätt är det tillräckligt med *en* digital enhet som utför samma operationer på bitmönstren oberoende av om vi representerar tal med eller utan tecken, *tvåkomplementsform* är en sådan representation. Vi exemplifierar med fyra bitars ord. Detta innebär att vi kan representera 16 olika tal, ty:

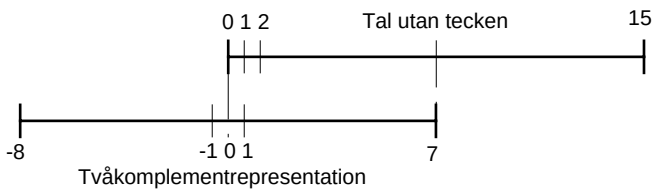
2^n ger antal olika kombinationer, där n är antal bitar i talet.
$2^4 = 16$ olika tal

Vi väljer nu representation enligt Tabell 3.2 i marginalen.

Bit-mönster	Utan tecken	Med tecken
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0
1111	15	-1
1110	14	-2
1101	13	-3
1100	12	-4
1011	11	-5
1010	10	-6
1001	9	-7
1000	8	-8

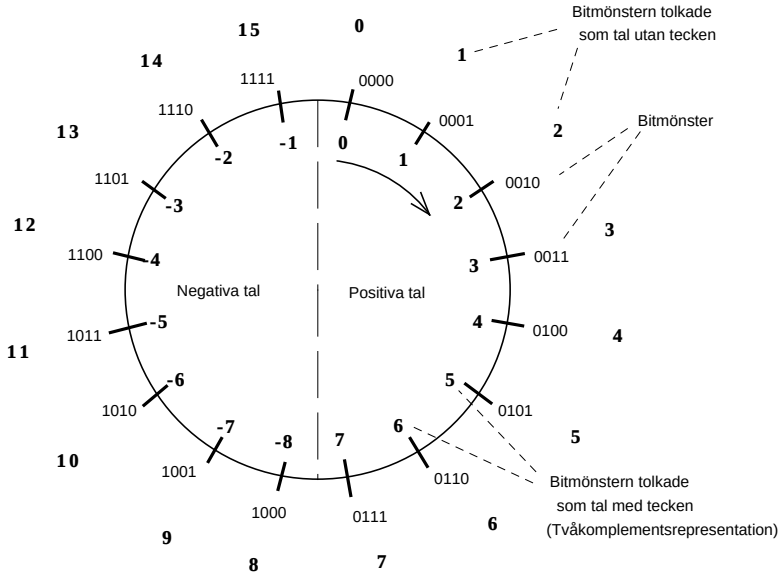
Tabell 3.2 *Tvåkomplements-representation*

Observera att med fyra bitar (n=4) kan vi representera talområdet [0..15] för tal utan tecken (naturliga tal) och talområdet [-8..7] för tal med tecken (heltal). Antal negativa tal vi kan representera är åtta [-8,-1] och antal positiva tal är sju. Med fyra bitar kan man representera 16 olika tal och vi har bara 15. Den siffran som saknas är "nollan" vilket tolkas som ett positivt tal i tvåkomplementsrepresentationen.



Figur 3.2 *Talområde för fyrabitars tvåkomplementstal och tal utan tecken*

Ett annorlunda sätt än som i Figur 3.2 är att grafiskt visa tvåkomplementsrepresentationen med en så kallad talcirkel (se Figur 3.3). Figuren visar bitmönstren som lagras i datorn och vad dessa motsvarar om vi väljer att tolka dessa som tal med- och utan tecken.



Figur 3.3 *Talcirkel för fyra bitar.*

Vi sätter nu upp några additionsexempel som visar att samma räkneregler fungerar oberoende av om vi tolkar talen med eller utan tecken.

EXEMPEL 4-bitars addition av %0010 och %0011

Bitmönstren tolkade som tal utan tecken	Bitmönster	Bitmönstren tolkade som tal med tecken
	1 ← *Minnessiffror*	
2	0010	2
+ 3	+ 0011	+ 3
= 5	= 0101	= 5

Vi har adderat efter räknelagarna som gäller för binär aritmetik. Som vi ser så blev resultatet korrekt oberoende av om vi tolkar talen med eller utan tecken. Detta beror på att vi i detta exemplet har adderat tal som ryms inom talområdet för *båda* representationerna. Jämför med Figur 3.3 och observera att talet 3 adderas till talet 2 i positiv riktning i cirkeln.

Man kan formellt bevisa att de normala räknelagarna gäller även för tvåkomplements-representationen.

Lår oss nu se ett exempel på hur man *överskrider* talområdet.

EXEMPEL 4-bitars addition av %0111 och %0101

Bitmönstren tolkade som tal utan tecken	Bitmönster	Bitmönstren tolkade som tal med tecken
	111 ← *Minnessiffror*	
7	0111	7
+ 5	+ 0101	+ 5
=12	= 1100	=-4

Här ser vi att resultatet blev korrekt för tal utan tecken men fel för tal med tecken. Detta beror på att resultatet 12 inte kan representeras i fyrabitars tvåkomplementsrepresentation (se Tabell 3.2). Studera även Figur 3.3 och speciellt då tal med tecken inne i cirkeln. Adderas ett tal till 7 så kommer vi direkt över på den negativa halvan.
Här följer ytterligare ett exempel på hur man överskrider talområdet.

EXEMPEL 4-bitars addition av %0111 och %1101

Bitmönstren tolkade som tal utan tecken	Bitmönster	Bitmönstren tolkade som tal med tecken
	111 ← *Minnessiffror*	
7	0111	7
+13	+ 1101	+-3
= 4	= 0100	= 4

Här ser vi att resultatet blev korrekt för tal med tecken men fel för tal utan tecken. Detta beror på att resultatet 20 inte kan representeras med fyra bitar. Figur 3.3 visar också detta ty när 13 adderas till 7 så kommer vi att passera "nollan" i cirkeln för tal utan tecken.

Nästa exempel visar hur resultatet kan bli fel för tal både med och utan tecken.

EXEMPEL 4 bitars addition av %1000 och %1100

Bitmönstren tolkade som tal utan tecken	Bitmönster	Bitmönstren tolkade som tal med tecken
8	1000	-8
+ 12	+ 1100	+ -4
= 4	= 0100	= 4

Fundera själv ut varför det blev fel genom att studera Figur 3.3.

"spill" - Uppstår då resultat av aritmetisk operation ej kan anges för en given talrepresentation

Vi kan nu alltså, genom att välja så kallad *tvåkomplements-representation*, även representera negativa tal binärt. Vi har sett hur vanliga räkneregler för addition kan tillämpas och vi har också sett exempel där så kallat "spill" uppkommer. Hur skall vi då veta om resultatet blev korrekt eller resulterat i spill?

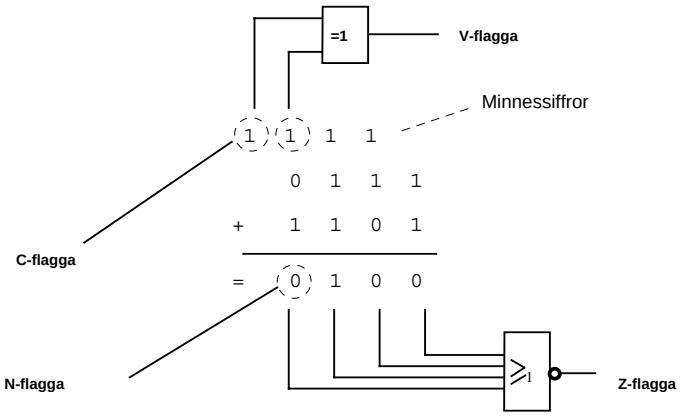
Flaggbitarna sätts enligt reglerna för tvåkomplementsaritmetik.

Det enda vi ser är ett resultat som i bland visar sig vara korrekt och i bland fel. Den digitala enhet som utför additionen har ingen som helst uppfattning om hur vi tolkar talen, som tal med eller utan tecken. För att lösa detta problem har man infört *statusbitar* (eller *flaggbitar*) som ger kompletterande information om resultatet av additionen. Dessa flaggbitar sätts av den del av datorns centralenhet, ALU'n, som utför all aritmetik i datorn.

Z	*Zero*	Biten indikerar om resultatet av en operation blev noll.
C	*Carry*	Biten indikerar om resultatet av en operation mellan tal utan tecken resulterade i spill.
V	*Overflow*	Biten indikerar om resultatet av en operation mellan tal med tecken resulterade i spill.
N	*Negative*	Biten indikerar om resultatet av en operation blev negativt (tal med tecken).

Tabell 3.3 *Statusflaggor från ALU:n*

Då vi utfört en aritmetisk operation kan vi kontrollera ALU'ns flaggbitar (statusbitar) för att få reda på om resultatet av operationen blev korrekt eller inte. Dessa finns tillgängliga i processorn i ett speciellt register som kallas "flaggregistret" eller "statusregistret". Vi återkommer till mer detaljer om detta längre fram och koncentrerar oss i detta kapitel på hur dessa flaggor sätts av ALU'n.



Figur 3.4 *Generering av flaggbitarna*

EXEMPEL: 4 bitars addition av %0111 och %1101

Bitmönstren tolkade som tal utan tecken	Bitmönster	Bitmönstren tolkade som tal med tecken
	4 3 2 1 0 ← position	
	1 1 1 1 ← minnessiffror	
7	0 1 1 1	7
+ 13	1 1 0 1	+ -3
= 4	0 1 0 0	= 4

Resultatet blir korrekt för tal med tecken men fel för tal utan tecken. Detta beror på att resultatet 20 inte kan representeras med fyrabitars tvåkomplementsrepresentation. Observera speciellt additionens minnessiffror. Minnessiffran (i position 4) från additionen av position tre i exemplet indikerar om resultatet ryms för tal utan tecken. Man kallar denna minnessiffra för *Carry* eller *Carry-flaggan.* C-flaggan är en kopia av den vänstra minnessiffran (i position 4 i vårt exempel). Se även Tabell 3.3.

C-flaggan anger spill vid aritmetik på tal utan tecken.
C-flaggan har inte denna betydelse vid aritmetik på tal med tecken

V-flaggan har ingen betydelse vid aritmetik på tal utan tecken.

N-flaggan har ingen betydelse vid aritmetik på tal utan tecken.

Z-flaggan gäller vid aritmetik på tal med- och utan tecken.

Resultatet ovan blev korrekt för tal med tecken vilket kan avläsas av *Overflow-flaggan* (**V**). V-flaggan ettställs om de två vänstra minnessiffrorna (i position 3 och 4) är olika. Alltså, när dessa siffror är1 0 eller 0 1 kommer V-flaggan att ettställas. Detta kan realiseras med en EXOR-grind, se Figur 3.4. I vårt exempel är dessa minnessiffror lika (1 1) vilket innebär att V-flaggan är noll och därför är resultatet korrekt.

Den vänstra biten (position 3) i resultatets bitmönster anger om talet är negativt eller inte. Vårt resultat har en nolla i position 3 och därför är resultatet positivt. *Negativ-flaggan* (**N**) är en kopia av position 3 i talet vi studerar.

Slutligen har vi *Zero-flaggan* (**Z**) som indikerar om alla bitarna i resultatet är noll.

Adderas två positiva tal och resultatet blir negativt så ettställs V-flaggan. På samma sätt ettställs flaggan om två negativa tal adderas och resultatet blir positivt. Detta kan mera formellt uttryckas:

$$= \overline{N_X}\,\overline{N_Y} N_R + N_X N_Y \overline{N_R}$$

där N anger talen R, X och Y:s N-flaggor (teckenbitar) och där R = f(X,Y). Då vi adderar tal med olika tecken kan spill inte uppträda.

I stället för att kontrollera om de två vänstra minnessiffrorna är olika för att bestämma V-flaggan kan vi studera bitmönstren som ingår i additionen. Om vi adderar två tal med samma tecken (talens N-flaggor är lika) och får ett resultat med motsatt tecken så ettställs V-flaggan. Vid addition av två tal med olika tecken nollställs V-flaggan. Additionen i vårt föregående exempel gav alltså upphov till följande *flaggpåverkan*:

C=1 V=0 N=0 Z=0

Studera nästa exempel.

EXEMPEL: 4 bitars addition av %0111 och %0101

Bitmönstren tolkade som tal utan tecken	Bitmönster	Bitmönstren tolkade som tal med tecken
	C 3 2 1 0← position	
	0 1 1 1 ← minnessiffror	
7	0 1 1 1	7
+ 5	0 1 0 1	+ 5
= 12	1 1 0 0	= -4

Förståelsen för hur flaggbitarna påverkas är mycket viktig eftersom dessa används i flera vanliga program-konstruktioner där man jämför och testar variabler. Vi återkommer till detta längre fram.

Här ser vi att resultatet blev korrekt för tal utan tecken men fel för tal med tecken. Detta beror på att resultatet 12 inte kan representeras i fyrabitars tvåkomplementsrepresentation. Denna information kan vi läsa ut genom att studera flaggbitarna efter operationen. Som vi ser är den vänstra minnessiffran noll (C-flaggan är noll), vilket innebär att resultatet rymdes för tal utan tecken.

Vidare ser vi att de två vänstra minnessiffrorna är olika (0 1). Detta innebär att (V=1) resultatet inte rymdes för tal utan tecken. Studerar vi de två bitmönstren som adderas ser vi att båda talen är positiva (den vänstra biten, N-flaggan, är noll), däremot är den vänstra biten ett (negativt) i resultatet. Också detta innebär att V-flaggan ettställs. Slutligen så ser vi att Z-flaggan är noll ty resultatets bitmönster innehåller minst en etta. Additionen ovan gav alltså upphov till följande flaggpåverkan:

C=0 V=1 N=1 Z=0

3.4.1 Lång addition

Som vi sett tidigare adderar processorns ALU bitmönstren som binära tal oberoende av om vi arbetar med tal med eller utan tecken. Vi studerar enbart flaggbitarna **C** och **V** i processorns status register (eller flaggregister) för att undersöka om resultatet är korrekt eller inte.

Problem uppstår när vi önskar addera större tal än det vår processor är konstruerad för. Vi fortsätter att exemplifiera med fyrabitars aritmetik, som alltså utförs av en fyrabitars ALU. Vi kan dock enkelt utvidga resonemangen till tal med större ordbredd. Låt oss illustrera detta med följande exempel.

Tänk dig att vi önskar addera två st 8-bitars (R = X + Y) tal med vår fyrabitars ALU. Vi gör då två additioner i sekvens, där den första additionen utför

$R_{LOW} = X_{LOW} + Y_{LOW}$

sedan följer addition nummer två

$R_{HIGH} = X_{HIGH} + Y_{HIGH}$ + *Carry*-flaggan
(från föregående operation).

Detta verkar självklart om vi sätter upp räknestycket på följande sätt:

```
                    0 0111 ◄─── 1100 ← minnessiffror
  XHIGH XLOW            0011       0100
+ YHIGH YLOW      +     0010       1101
= RHIGH RLOW      =     0110       0001
```

Observera C-flaggan från den första additionen ($X_{LOW} + Y_{LOW}$) måste ingå i den andra additionen ($X_{HIGH} + Y_{HIGH}$). På grund av att vi adderar talens mest signifikanta bitar sist kommer dessa att påverka flaggsättningen. Är den vänstra biten satt i resultatet så är altså talet negativt och N-flaggan satt. Vanliga processorer har två olika additionsinstruktioner, en som adderar med C-flaggan och en som inte gör det. Vidare kan det finnas speciella additionsinstruktioner som för exempelvis NBCD-tal.

3.4.2 Tvåkomplementering

Vi ska nu visa hur man enkelt kan negera (byta tecken på) ett tal med tecken.

EXEMPEL Negera tvåkomplementstalet %1011

		Bitmönster	Bitmönstren tolkade som tal med tecken
Tal att negera		1011	-5
Invertera varje bit		0100	
Addera 1	+	0001	
	=	0101	= +5

Arbetsgången är att invertera varje bit i bitmönstret vi önskar byta tecken på. Därefter adderas ett och vi får resultatet. Operationen kallas även att *tvåkomplementera* ett tal. Vi har med denna operation negerat talet -5 och fått talet 5.

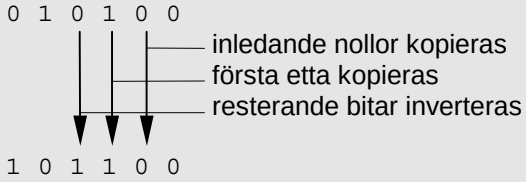
EXEMPEL Negera tvåkomplementstalet %0101

		Bitmönster	Bitmönstren tolkade som tal med tecken
Tal att negera		0101	+5
Invertera varje bit		1010	
Addera 1	+	0001	
	=	1011	= -5

Som vi ser stämmer resultatet.

Ett ännu snabbare sätt att tvåkomplementera ett binärt tal är att med början från minst signifikanta bit till mest signifikanta bit: Kopiera nollor, kopiera första etta, invertera övriga bitar

EXEMPEL: Tvåkomplementera talet %010100



Pröva själv med olika fyra-bitars tal tal och jämför med Tabell 3.2.

3.4.3 Subtraktion

Processorer har vanligtvis inga speciella logiknät för att utföra subtraktion av binära tal. Operationen X-Y utförs i stället som operationen X+(-Y), dvs addition av Y's tvåkomplement.

EXEMPEL Utför subtraktionen X-Y med operationen X+(-Y)
där X=7 och Y=2

Först tvåkomplementeras (negeras) Y enligt:

	Bitmönster	Bitmönstren tolkade som tal med tecken
Tal att negera	0010	+2
Invertera varje bit	1101	
Addera 1	+ 0001	
	1110	= -2

Sedan utförs additionen X+ (-Y) enligt

	Bitmönster		Bitmönstren tolkade som tal med tecken
1	11		
	0111		7
+	1110	←negerat bitmönster	-2
=	0101		= 5

Studerar vi de två vänstra minnessiffrorna ser vi att de är lika, vilket innebär att V-flaggan är noll och resultatet är korrekt för tal med tecken. Däremot är den vänstra minnessiffran ett vilket skulle kunna få oss att tro att resultatet inte är korrekt för tal utan tecken. Resultatet *är* emellertid korrekt. Orsaken är att vid subtraktion har C-flaggan en speciell betydelse, då sätts C-flaggan till *inversen* av den vänstra minnessiffran. Vid subtraktion representerar C-flaggan en lånesiffra (*Borrow*). Flaggbitarna blir således:

N = 0, Z = 0, V = 0, C = 0

i visar också ett exempel där resultatet blir negativt (R=X-Y= 3-5). Observera att dessa tal kan representeras både som tal med- och utan tecken med fyra bitar.

EXEMPEL Utför subtraktionen X-Y med operationen X+(-Y)
där X=3 och Y=5

Först tvåkomplementeras (negeras) Y enligt:

	Bitmönster	Bitmönstren tolkade som tal med tecken
Tal att negera	0101	+5
Invertera varje bit	1010	
Addera 1	+ 0001	
	1011	= -5

Sedan utförs additionen X+ (-Y) enligt

Bitmönstren tolkade som tal utan tecken	Bitmönster	Bitmönstren tolkade som tal med tecken
	0 011	
3	0011	3
+11	+ 1011 ←negerat bitmönster	-5
= 14	= 1110	= -2

Resultatet stämmer för tal med tecken men inte för tal utan tecken (självklart ty resultatet som är mindre än noll kan överhuvudtaget inte representeras som ett tal utan tecken). Vi kan direkt avläsa detta av flaggsättningen. Studerar vi de två vänstra minnessiffrorna ser vi att de är lika, vilket innebär att V-flaggan är noll och resultatet är korrekt för tal med tecken. Däremot är den vänstra minnessiffran noll vilket innebär att C-flaggan är ettställd (inverterad vid subtraktion). Detta indikerar som bekant att vi har "lånat" att resultatet inte är korrekt för tal utan tecken. Flaggbitarna blir:

N = 1, Z = 0, V = 0, C = 1

Precis som vid addition uppstår problem när vi önskar subtrahera större tal än det vår processor är konstruerad för. Vi fortsätter att exemplifiera med fyrabitars aritmetik. Vi önskar subtrahera två st 8-bitars (R = X - Y) tal med en fyrabitars processor. Vi gör då två subtraktioner i sekvens, där den första subtraktionen utför

$R_{LOW} = X_{LOW} + (-Y_{LOW})$

sedan följer

$R_{HIGH} = X_{HIGH} + (-Y_{HIGH}) +$ *Carry-flaggan*

Vi sätter upp räknestycket på följande sätt:

	0 0111 ◄——	**1**100 ← minnessiffror
X_{HIGH} X_{LOW}	0011	0100
$-Y_{HIGH}$ Y_{LOW}	+0010	+1101
$=R_{HIGH}$ R_{LOW}	=0110	0001

Observera C-flaggan från den första additionen ($X_{LOW} + (-Y_{LOW})$) måste ingå i den andra additionen ($X_{HIGH} + (-Y_{HIGH})$).

3.4.4 Vänsterskift (multiplikation med 2)

Att skifta ett binärt tal ett steg till vänster innebär att talet multipliceras med 2.

EXEMPEL: Vänsterskift

Bitmönstren tolkade som tal utan tecken	Bitmönster	Bitmönstren tolkade som tal med tecken
	3210← position	
3	0011	3
6	0110	6
12	1100	-4
8	1000	-8

Vid ett vänsterskift fyller vi på med nollor från höger. Studerar vi först talen utan tecken ser vi att vi kan skifta tills ettor försvinner ut till vänster. Utskiftad bit motsvarar C-flaggan och är denna noll innebär det korrekt skift (multiplikation med två) för tal utan tecken. Funderar vi lite ser vi att en addition av talet med sig själv ger upphov till ett vänsterskift.

För tal med tecken ser vi att fel uppstår när siffran i position tre byter tecken, vi har multiplicerat ett positivt tal med två och erhåller ett negativt resultat. V-flaggan ettställs i sådana fall. Observera det nedersta skiftet (-4*2 = -8). Ett vänsterskift fungerar även för negativa tal.

Precis som tidigare uppstår problem när vi vill skifta större tal än det vår ALU är konstruerad för. Vi fortsätter att exemplifiera med fyrabitars aritmetik. Tänk dig att vi önskar multiplicera ett 8-bitars (R = 2*X) tal med två. Vi måste då göra två skiftningar i sekvens, där det första skiftet utför

$R_{LOW} = 2 * X_{LOW}$ (där en nolla skiftas in från höger)

sedan följer

$R_{HIGH} = 2 * X_{HIGH}$ (där *Carry*-flaggan från föregående operation skiftas in från höger.)

Detta verkar självklart om vi sätter upp räknestycket på följande sätt:

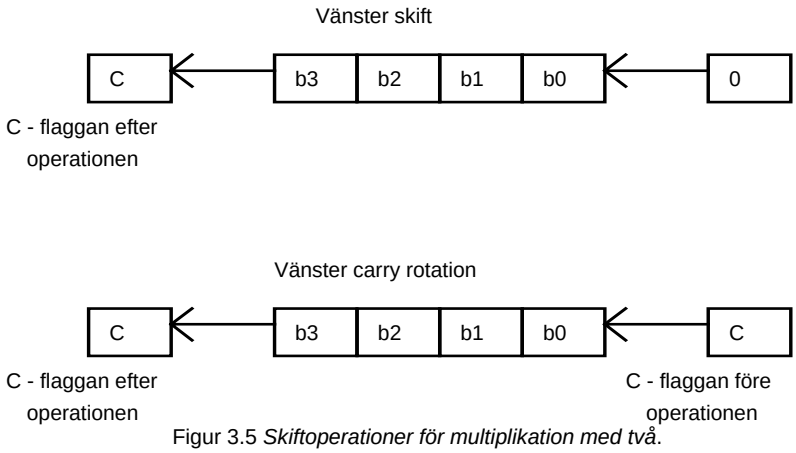
X_{HIGH} X_{LOW}	0010	1101
R_{HIGH} R_{LOW}	0101	1010

Observera C-flaggan från det första skiftet (X_{LOW}) måste skiftas in vid det andra skiftet (X_{HIGH}). På grund av att vi skiftar talets mest signifikanta bitar sist kommer denna att påverka flaggsättningen.

De flesta processorer är utrustade med två olika vänsterskift-instruktioner, en som skiftar in en nolla från höger (*skift*) och en som skiftar in C-flaggan (*Carry-rotation*), studera Figur 3.5 nedan. Observera speciellt carry-rotationen som skiftar in den "gamla" C-flaggan in till bit 0 för att sedan skifta bit 3 in i C-flaggan. På så sätt skiftas totalt 5 bitar vid en rotation.

Vårt exempel (R = 2 * X) ovan kan altså utföras i en processor som *i sekvens* gör:

- vänster skift (X_{LOW})
- vänster carry rotation (X_{HIGH})



Figur 3.5 *Skiftoperationer för multiplikation med två.*

3.4.5 Högerskift (division med 2)

En skiftning av bitmönstret ett steg till höger innebär att talet divideras med 2. Eftersom den mest signifikanta biten (teckenbiten) ingår i skiftet måste tal med- och utan tecken behandlas var för sig. Vi studerar först tal utan tecken.

EXEMPEL Högerskift, tal utan tecken

Bitmönstren tolkade som tal utan tecken	Bitmönster
	3210← position
12	1100
6	0110
3	0011
1	0001

Vid ett högerskift fyller vi på med nollor från vänster. Studerar vi exemplet ovan ser vi att vi kan skifta tills ettor försvinner ut till höger. Utskiftad bit motsvarar C-flaggan och är denna noll innebär det korrekt skift (division med två) för tal utan tecken. Studera det sista skiftet ovan där tre delat med två blir ett. Den högra biten skiftas till C-flaggan som på detta sätt indikerar fel resultat (spill).

Tänk dig att bitmönstret ovan representerar ett tal med tecken. Skiftar vi då in nollor från vänster skulle alla skift av negativa tal bli fel, eftersom teckenbiten ändras från ett till noll. Det krävs därför en speciell skiftinstruktion som utför division av tal med tecken. Instruktionen måste skifta in en kopia av siffran i position tre.

EXEMPEL Högerskift, tal med tecken

Bitmönster	Bitmönstren tolkade som tal med tecken
3210← position	
1000	-8
1100	-4
1110	-2
1111	-1

Utskiftad bit till höger motsvarar C-flaggan och är denna noll innebär det korrekt skift (division med två) för tal med tecken.

Precis som tidigare uppstår problem när vi önskar dividera större tal med två än det vår processor är konstruerad för. Tänk dig att vi önskar dividera ett 8-bitars (R = X/2) tal med två i en fyrabitars processor. Vi måste göra två skiftningar i sekvens, där det första skiftet utför

$R_{HIGH} = X_{HIGH}/2$ (där det skiftas in en nolla från vänster vid division av tal utan tecken och en kopiering av bitposition 3 till bitposition 2 för tal med tecken)

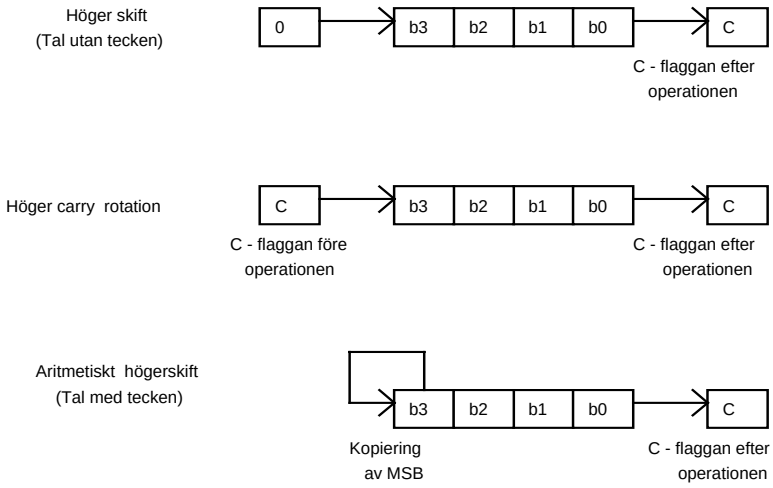
sedan följer

$R_{LOW} = X_{LOW}/2$ (där C-flaggan skiftas in från vänster)

De flesta processorer är utrustade med tre olika högerskift-instruktioner, en som skiftar in en nolla från vänster (skift), en som skiftar in C-flaggan (rotera) och en som skiftar in en kopia av den mest signifikanta biten (aritmetiskt skift) Se Figur 3.6. Observera speciellt carry-rotationen som skiftar in den "gamla" C-flaggan in till bit 3 för att sedan skifta bit 0 in i C-flaggan. På så sätt skiftas totalt 5 bitar vid en rotation.

Vårt exempel (R = 2/X) ovan kan alltså utföras i en processor som *i sekvens* utför:

- höger skift (X_{HIGH})
- höger carry rotation (X_{LOW})



Figur 3.6 *Skiftoperationer för division med två*

3.5 Övningsuppgifter

Uppgift 13: Ange följande decimala tal på *tvåkomplementsform*. (Ledning, bestäm först binär form av talet utan tecken, tvåkomplementera därefter)

```
a) -10 = % -(__________) = __________
b) -38 = % -(__________) = __________
c) -42 = % -(__________) = __________
d) -56 = % -(__________) = __________
```

Uppgift 14: Genomför följande additioner (R=X+Y) av tal givna på *binärform*. Ange dessutom hur flaggorna C och Z sätts respektive nollställs av operationerna. Det tillgängliga talområdet begränsas av 8 binära siffror.

```
a) X = 00100100  Y= 01001010
b) X = 10111100  Y= 01000100
c) X = 10000001  Y= 10000001
```

Uppgift 15: Genomför följande additioner av tal givna på *tvåkomplementsform*. Ange dessutom hur flaggorna V och Z sätts respektive nollställs av operationerna. Det tillgängliga talområdet begränsas av 8 binära siffror.

```
a) X = 00100100  Y= 01001010
b) X = 10111100  Y= 01000100
c) X = 10000001  Y= 10000001
```

Uppgift 16: Utför följande subtraktioner av tal givna på binär form. Redogör speciellt för hur eventuell lånesiffra (”borrow”) genereras vid operationen.

```
a) 126 – 80
b) 80 – 126
```

Uppgift 17: Utför följande subtraktioner av tal givna på tvåkomplementform med hjälp av addition. Redogör speciellt för hur flaggor sätts av operationen.

```
a) 126 – 80
b) 80 – 126
```

Uppgift 18: Ange resultatet av följande skiftoperationer, ange speciellt om (och i så fall under vilka förutsättningar) spill uppstår vid operationen.

```
a) vänsterskift (0101)
b) vänsterskift (1000)
```

Uppgift 19:

```
a) högerskift (0101)
b) högerskift (1000)
```

Uppgift 20:

```
a) aritmetiskt högerskift (0101)
b) aritmetiskt högerskift (1000)
```

4. Mikroprocessorns uppbyggnad och arbetssätt

MC68340 - Microcontroller från Motorola

En mikroprocessor är en mycket komplex elektronisk komponent och som sådan också svår att till fullo förstå. Komplexiteten ger dock samtidigt mikroprocessorn en annan egenskap, den blir faktiskt enklare att använda…

I detta kapitel ska vi närma oss mikroprocessorn med målsättningen att så snart som möjligt kunna använda den. Vi kommer att introducera flertalet nya begrepp och facktermer som förekommer speciellt i samband med assemblerprogrammering.

Kapitlet är organiserat i tre delar:

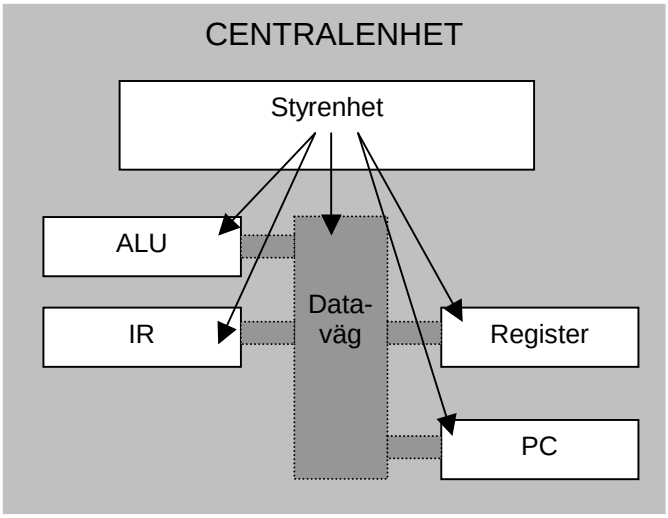
- Mikroprocessorn’s (eller rättare ”centralenhetens”) uppbyggnad beskrivs mycket kortfattat.
- Mikroprocessorn’s arbetssätt illustreras.
- Vi ger ett exempel på en mycket enkel mikroprocessor och hur den kan programmeras.

4.1 Mikroprocessorns uppbyggnad

Utan att gå in på en detaljerad beskrivning av en mikroprocessor, kan dess arbetssätt beskrivas på ett mycket enkel sätt:

1. Hämta en instruktion
2. Utför denna instruktion
3. Upprepa förfarandet

I mikroprocessorn finns centralenheten, eller CPU:n, som "hjärtat" i datorsystemet. Härifrån utförs instruktionshämtning och här utförs instruktionen. Låt oss titta lite närmare på hur en CPU kan vara uppbyggd.



Figur 4.1 *Centralenhet*

Register, här lagras data som ska bearbetas av CPU:n. Data kan läsas *från* primärminnet till register, eller skrivas *till* primärminnet från register. Det kan finnas flera register och de kan också ha olika funktion i centralenheten.

ALU, (*Arithmetic Logical Unit*), här utförs all aritmetik, dvs räknesätt som addition, subtraktion, division och multiplikation. Även logiska operationer som AND, OR och EOR kan utföras av ALU:n

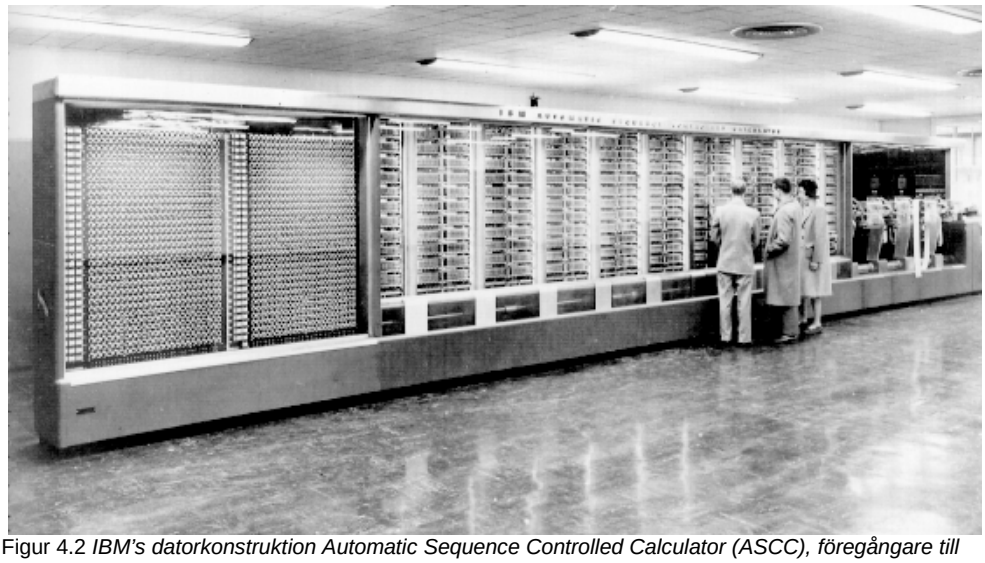
Styrenheten skapar de interna signaler som krävs för att exempelvis koppla samman olika register med ALU:n, få ALU:n att utföra önskad operation, Läsa från primärminnet, skriva till primärminnet, etc.

Dataväg kallas de *interna bussar* som används för att koppla samman register med ALU, register med minne, etc. Datavägen styrs alltså av Styrenheten.

4.2 Mikroprocessorns arbetssätt

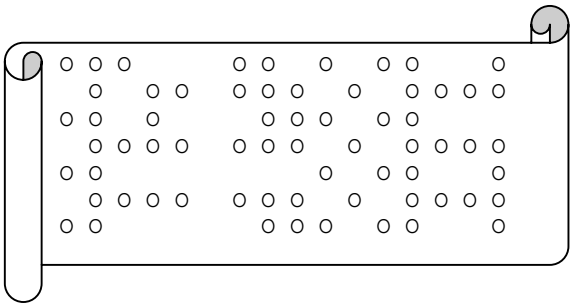
4.2.1 Det lagrade programmets princip

För att programmera de allra första datorerna (på 1940-talet) användes ett antal kopplingskablar. Dessa kopplades upp för att ange en sekvens av styrsignaler (nollor och ettor) till processorn. För att datorn skulle utföra ett annat program fick man koppla om alla sladdarna på nytt. Detta var en mycket omständigt arbete.



Figur 4.2 *IBM's datorkonstruktion Automatic Sequence Controlled Calculator (ASCC), föregångare till dagens fick-kalkylatorer. Till vänster syns primärminnet av s.k kärnminnestyp, därefter följer flera rader med kopplingsplintar för programmeringen. Maskinen var 18 meter lång, 3 meter hög och arbetade med en enkel reläteknik. Installerades hos Harvard University i februari 1944. (Källa: IBM)*

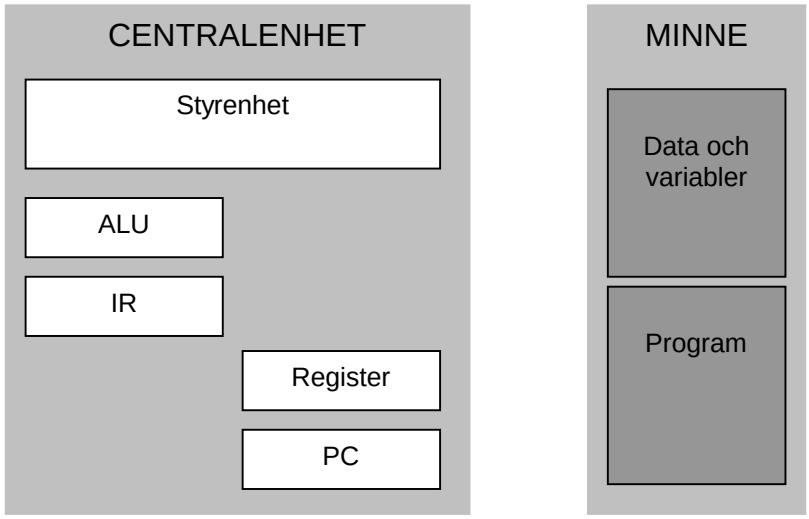
En amerikan, von Neuman, insåg att styrinformationen kunde placeras i *sekvens* i datorns minne på samma sätt som data och variabler (som minnet ursprungligen var ägnat för). På så sätt slapp man koppla upp programmet med sladdar. I stället överförde man programmet från exempelvis en hålad pappersremsa till datorns minne. Vi säger att vi *läser in* ett program till datorns minne. Kombinationen av hål i pappersremsan gav den sekvens av nollor och ettor som bildade de instruktioner som processorn så småningom skulle utföra.



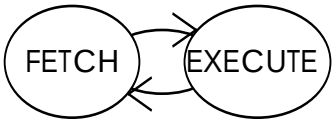
Figur 4.3 *Kombinationen av "hål", "icke-hål" omvandlas till nollor och ettor av en hålkortsläsare.*

Med den hålade pappersremsan kunde man snabbare byta program i datorn och utföra ("köra" eller *exekvera*) ett nytt program.

Just denna princip, att lagra program och data i samma minne tillskrivs alltså von Neuman. Datorer som konstruerats för att arbeta på detta sätt, *det lagrade programmets princip*, kallas därför ofta, kort och gott, *von Neumann-datorer*. Hur kan då processorn i en von Neumann-dator skilja ut vad som är program och vad som är data i ett och samma minne. Jo, minnet kan sägas vara uppdelat i två delar där den ena delen innehåller programmet och den andra innehåller data och variabler. Vidare finns det ett speciellt register i processorn som pekar ut den nästa styrinformationen (instruktionen) som står i tur att utföras. Detta speciella register kallas för *programräknaren* (*Program Counter* eller *PC*). Dessutom måste ett speciellt register, där en instruktion kan lagras, upplåtas. Vi kallar detta register *instruktionsregister* (IR).



Figur 4.4 *Illustration av Centralenhet och Minne*



Centralenhetens tillstånd

Arbetsprincipen för en sådan dator är enkel:

1. Läs en instruktion från program-minnet, adressen anges av PC, placera instruktionen i IR, uppdatera så att PC innehåller adressen till nästa instruktion.
2. Styrenheten avkodar (tolkar) och utför instruktionen i IR

Centralenheten är alltså alltid i ett av två tillstånd: *FETCH* (hämta instruktion) eller *EXECUTE* (tolka och utför instruktion).

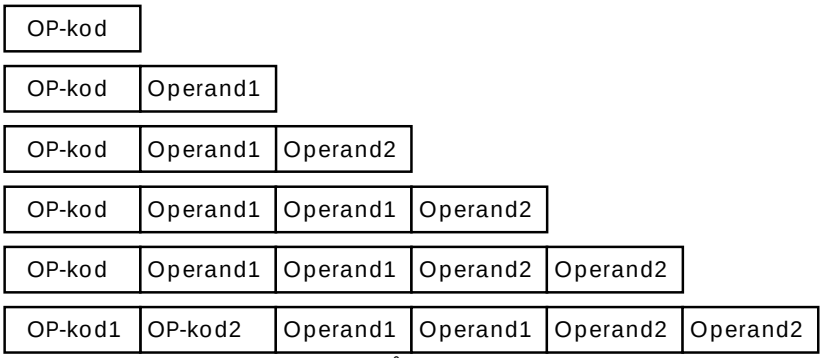
4.2.2 Maskininstruktionen

En maskininstruktion är en uppsättning nollor och ettor. Beroende på hur komplex en instruktion är och hur mycket information som följer med en maskininstruktion så kan den bestå av *olika* antal nollor och ettor (se Figur 4.5 nedan).

```
0011001111111100000100100011010000000000010101000011001000010000
1101000010000010
0000000000000011000000000000101
0100111011111001000000000011000000000000000001000
0100111001110000
```

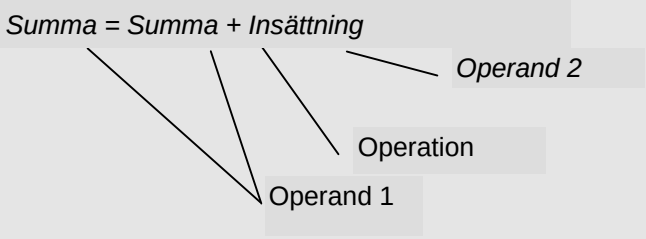
Figur 4.5 *Exempel på ett maskinprogram med 5 maskininstruktioner*.

Vanligen talar man om olika *instruktionsformat* och menar då hur en instruktion är uppdelad i *operationskod* och *operander*. Med Operationskod (OP) menas den styrinformation som processorn läser för att undersöka *vad* den skall utföra. OP-koden anger också *hur många operander* som finns och *var* dessa finns.



Figur 4.6 *Exempel på olika instruktionsformat*

EXEMPEL: operation och operander



Som exemplet visar finns här två operander, *Summa* och *Insättning* som skall adderas. Operationskoden blir därför *addera.* Maskininstruktionen består av tre fält, nämligen *Operationskod, Operand 1* och *Operand 2*.

Vissa processorer har en förhållandevis enkel uppbyggnad av operationskoderna. Exempelvis kan de första bitarna ange instruktionstyp och de följande ange var eventuella operander finns. Ofta nöjer man sig med två operandfält. För binära operationer innebär då detta att destinationsfältet samtidigt utgör ett källoperandsfält. Situationen illustreras i exemplet ovan av att destinationen "Summa" dessutom återfinns i högerledet, dvs utgör en av källoperanderna.

EXEMPEL: Instruktionsformat

Här kan exempelvis bit7 till bit4 bestämma instruktionstyp, förslagsvis:

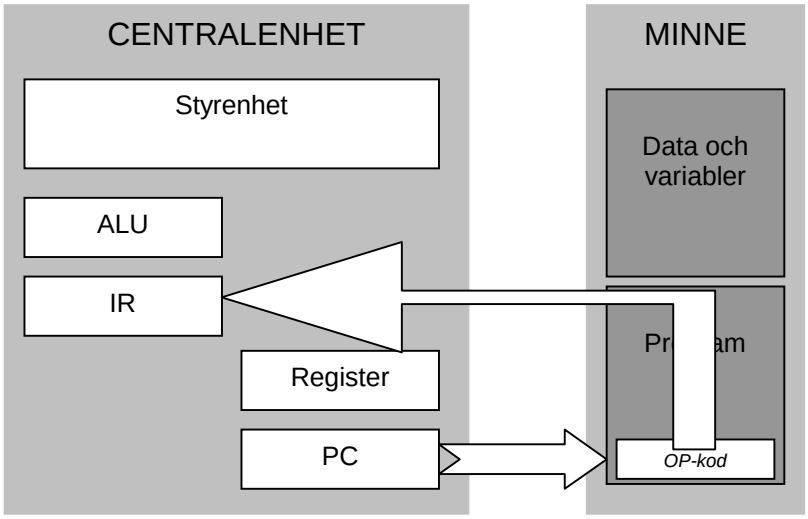
0 0 0 0 betyder dataförflyttning
0 0 0 1 betyder addition
0 0 1 0 betyder subtraktion
osv.

Vidare kan bit3 och bit2 ange var *källan* finns. Källan motsvaras av variabeln Insättning i exemplet ovan. *Destinationen* motsvaras av variabeln Summa och dess adress ges av bit1 och bit0. (se nedan).

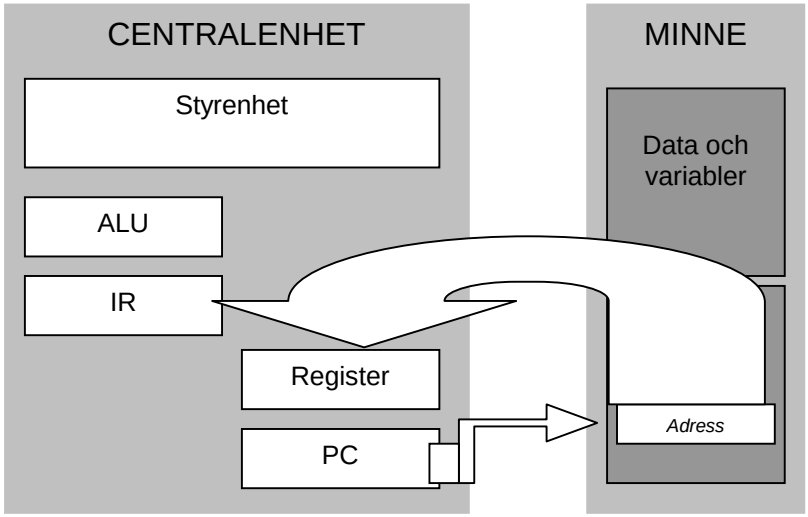
0 0 betyder att operanden följer direkt efter operationskoden i minnet (ej i destinationsfältet)
0 1 betyder att *operandens minnesadress* följer direkt efter operationskoden i minnet
1 0 betyder att operanden finns i ett dataregister
osv..

4.2.3 Instruktionsutförande

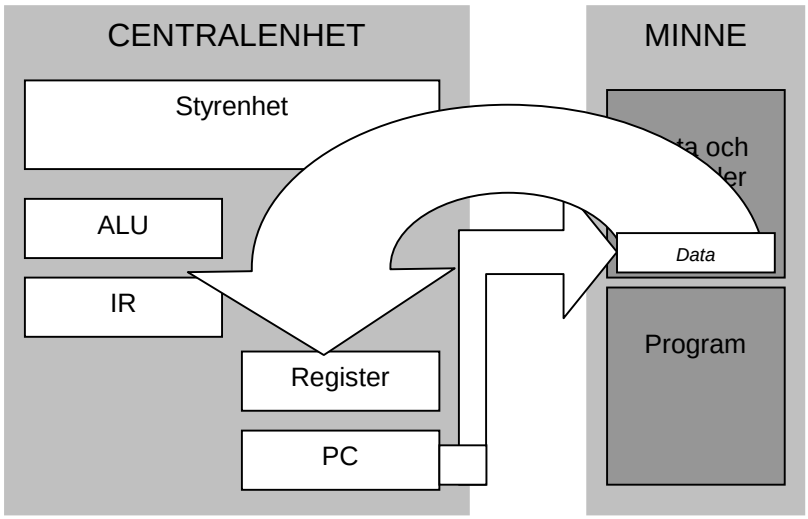
Programmet som består av maskininstruktioner är alltså lagrat i sekvens i datorns minne. Programräknaren - PC pekar ut den instruktion som står i tur att utföras. Instruktionsregistret – IR, används för att överföra OP-koden till processorns styrenhet.



Figur 4.7 *När processorn är i tillståndet "FETCH" används adressen i PC för att adressera minnet och en operationskod läses in till IR. När operationskoden är inläst avkodas den av styrenheten. Vidare ökas nu PC för att peka ut "nästa" instruktion i minnet.*



Figur 4.8 *Beroende på vad OP-koden anger så kommer nu styrenheten exempelvis att koppla om bussarna internt i processorn så att en adressangivelse som följer direkt efter OP-koden kan läsas in till ett register.*



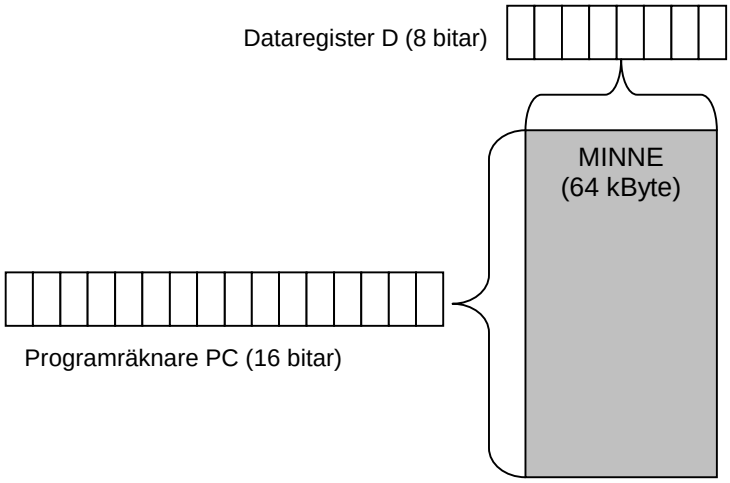
Figur 4.9 *Avslutningsvis kan data på angiven adress läsas till samma, eller eventuellt ett annat register.*

Då instruktionen och dess operand (eller operander) lästs in till centralenheten övergår styrenheten till EXECUTE-tillståndet. Beroende på instruktion kopplas nu de register som innehåller operanderna till ALU'n som utför såväl aritmetiska som logiska operationer. Då detta är klart följer eventuellt ytterligare en minnesoperation. Denna gång kan det vara resultatet av operationen som skrivs ut till någon minnesadress.

4.3 En enkel mikroprocessor

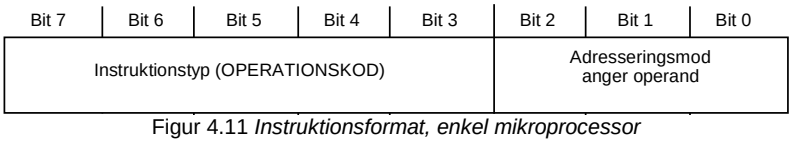
Låt oss betrakta en mycket enkel mikroprocessor. Vi låter denna processor ha 3 olika register; *programräknare* (PC), *instruktionsregister* (IR) och *dataregister* (D).

Vårt programminne består av maximalt 65536 st 8-bitars ord (64 kByte), därmed har vi också bestämt storleken hos de ingående registren. Det är lämpligt att dimensionera ett dataregister efter den ordbredd vi använder, dvs 8 bitar. Programräknaren måste kunna "peka ut" varje ord i primärminnet. Eftersom 2^{16}=65536, finner vi att detta register måste vara 16 bitar brett. Det är nu lämpligt att välja ett instruktionsformat som passar in i detta. Uppenbarligen bör en instruktion rymmas av 8 bitar, det räcker då med *en* läsning i minnet för att *hämta* instruktionen.



Figur 4.10 *Registermodell, enkel mikroprocessor*

Av instruktionens 8-bitar låter vi 4 bitar bestämma operationen, 4 bitar använder vi för att ange operander enligt instruktionsformatet i Figur 4.11.



Figur 4.11 *Instruktionsformat, enkel mikroprocessor*

Operationerna skulle kunna kodas enligt:

Bitfält	Operation
0 0 0 0	Läs data till register D
0 0 0 1	Skriv data från register D
0 0 1 0	Addera data till register D
. . .	
. . .	Dessa används ännu ej ...
1 1 1 1	

Sätten att ange operand, *adresseringssätten* skulle kunna vara:

Bitfält	Adresseringssätt
0 0	Data finns i nästa minnesord
0 1	Adress till data finns i 2 påföljande minnesord
1 0	Data finns i register D
1 1	Används ännu ej

4.3.1 Datakopiering i minnet

Med denna enkla mikroprocessor ska vi nu skriva en sekvens instruktioner som adderar ett tal, lagrat i primärminnet på adress 100, med en konstant (1) och därefter skriver resultatet till adress 101 i primärminnet. Vi kan *symboliskt* skriva denna sekvens av operationer som:

```
LD      100,D  "load D", läs data från adress 100 till
                       register D
ADD     #1,D   "add D" addera 1 till innehållet i
                                        register D

ST      D,101  "store D", skriv data i register
               D till adress101 i primärminnet
```

Vi använder detta tecken (#) för att skilja *konstanten* 1 från *adressen* 001

Med ledning av tabellerna och *programsekvensen* ovan, kan vi nu skriva de maskininstruktioner som måste lagras i programminnet. För exemplet antar vi att programmet placerats på adress 0 i primärminnet:

Adress	Minnesinnehåll	Kommentar
0	00001001	**LD** (adress följer) till register D
1	00000000	adress första byte
2	01100100	adress andra byte=100
3	00101000	**ADD** # (konstant följer) till register D
4	00000001	konstanten 1
5	00011001	**ST** (adress följer) från register D
6	00000000	adress första byte
7	01100101	adress andra byte =101

Ett symboliskt namn för en maskininstruktion kallas *mnemonic* och är avsevärt enklare att komma ihåg än den aktuella maskin-instruktionen (nollor och ettor). För varje instruktion används en unik mnemonic. Mnemonics bygger upp ett så kallat *assemblerspråk*, som ger oss möjlighet att skriva ett program av maskininstruktioner på ett (någorlunda) läsbart sätt. Detta *assemblerprogram* kan översättas till maskinkod av ett speciellt programverktyg som kallas *assemblator*.

På liknande sätt kodas alla de olika instruktioner processorn ska kunna utföra. Notera speciellt processorns uppträdande. Innehållet på adress 0 läses in, *detta förutsätts vara en instruktion*. Efter avkodning hämtas operanden till instruktionen och PC uppdateras, dvs dess innehåll ökas med 2 eller 3 beroende på operandens storlek. Då denna instruktion utförts hämtas nästa instruktion och förfarandet upprepas.

4.3.2 Programflödesinstruktioner

Instruktioner placeras ofta i olika *grupper* beroende på dess funktion. Exempelvis kallas instruktioner som LD och ST, i exemplet ovan, för *instruktioner för dataförflyttning* eller *Load/Store*. Vi har också sett exempel på en instruktion ur gruppen *aritmetiska* instruktioner (ADD). En annan viktig grupp av instruktioner är de som styr *programflödet*, dvs bestämmer olika vägar genom programmet. Exempelvis kan man konstruera en *oändlig slinga* med hjälp av en så kallad "hopp"-instruktion. Instruktionen ska, i stället för den sekvensiella exekveringen, ladda en ny adress (för nästa instruktion) i PC. Låt oss komplettera vår processor med instruktionen:

```
JMP         adress      jump to address
```

Instruktionen medför att PC laddas med *adress* och alltså avbryts den sekvensiella exekveringen. Detta är ett exempel på en *ovillkorlig ändring av programflödet*.

Låt oss nu använda vår processor i kombination med den konfiguration vi byggde i kapitel 2. Här hade vi ett system med en inport (adress $A000), en utport (adress $8000) och primärminne. Vi kan exempelvis konstruera en kontinuerlig reglering där vi *läser* inportens värde adderar något tal, säg 10, och skriver detta till utporten. För exemplet antar vi att programmet placerats med start på adress 0 i primärminnet:

Skrivsättet
$A000
anger att adressen givits på hexadecimal form.

```
adress  mnemonic      operander     kommentarer
0       LD            $A000,D       Läs inporten
3       ADD           #10,D         Addera talet 10
5       ST            D,$8000       Skriv till utporten
8       JMP           0             Upprepa (oändlig slinga)
```

4.3.3 Villkorlig programflödeskontroll

Det visar sig dock snart att det inte är tillräckligt att på detta sätt alltid utföra *ovillkorlig* ändring av programflödet. Vi måste också kunna åstadkomma en *villkorlig ändring av programflödet*. Villkoret för vilka instruktioner i programmet som ska utföras bestäms då av någon aritmetisk operation. Vi ser då behovet av att spara, exempelvis i form av *flaggor*, resultatet av en tidigare operation. Vi inför nu ytterligare ett register i vår processor. Vi kallar detta *flaggregister*. Registret har tre olika bitar som vardera reflekterar resultatet av en aritmetisk operation. Vi kallar flaggorna Z, C och V och definierar dom enligt följande: (jämför kapitel 3)

Z (zero), flaggan sätts till 1 om instruktionen medförde att resultatet blev 0, annars sätts flaggan till 0.

C (carry), flaggan sätts till 1 om en aritmetisk operation åstadkommer spill då de ingående operanderna betraktas som *tal utan tecken*, annars sätts flaggan till 0.

V (overflow), flaggan sätts till 1 om en aritmetisk operation åstadkommer spill då de ingående operanderna betraktas som *tal med tecken*, annars sätts flaggan till 0.

EXEMPEL:

```
        LD      #0,D
```

innebär att resultatet i register D är 0. Z-flaggan sätts därför till 1.

EXEMPEL:

```
        LD      #200,D
        ADD     #200,D
```

Resultatet (400) kan inte representeras med 8 bitar (ordbredden hos register D). C-flaggan sätts därför till 1.

EXEMPEL:

```
        LD      #100,D
        ADD     #100,D
```

kommer att resultera i tvåkomplementsspill ty resultatet (200) kan *inte* representeras som 8-bitars tvåkomplementstal

Låt oss illustrera användningen av flaggorna med en programsekvens som ska:
1) Läsa indata från inporten
2) Vänta tills indata är *skilt från* 100

Vi lägger nu till ytterligare en instruktion till vår processors instruktionsuppsättning:

`JEQ`	`adress`	*jump if equal zero*

dvs programflödesändringen, eller "hoppet", utförs bara om Z-flaggan i flaggregistret är 1, annars uppdateras PC som vanligt och nästa instruktion utförs.

Vår programsekvens blir då:

```
0     LD    $A000,D     Läs inportens värde
3     ADD   #-100,D     Subtrahera 100
5     JEQ   0           Om indata är 100,resulterar ADD
                        instruktionen i att Z-
                        flaggan sätts till 1, den
                        villkorliga programflödes-
                        instruktionen gör då att
                        exekveringen återupptas på
                        adress 0,annars fortsätter
                        exekveringen med nästa instruktion.
```

På liknande sätt kan vi införa den *komplementära* instruktionen:

`JNE`	`adress`	*jump if not equal*

dvs programflödesändring utförs om Z-flaggan är 0, annars fortsätter exekveringen med nästa instruktion

För att kunna testa om spill uppstått efter aritmetiska instruktioner inför vi även villkorliga programflödesinstruktioner som testar bitarna C och V:

`JCS`	`adress`	*jump if carry is set*
`JCC`	`adress`	*jump if carry is clear*
`JVS`	`adress`	*jump if overflow is set*
`JVC`	`adress`	*jump if overflow is clear*

4.3.4 Stackoperationer

Vi har nu, hos vår processor, *exempel på instruktioner för dataförflyttning, aritmetiska* operationer och *programflödes-instruktioner*. Detta är emellertid fortfarande inte tillräckligt. Vid uttrycksevaluering tvingas man ofta spara vissa deluttryck.

EXEMPEL: En korrekt evaluering av uttrycket
(x+y)-(z+w)
innebär att de båda parenteserna beräknas först, därefter kan subtraktionen utföras.

Vi behöver här en *temporär* lagringsplats där vi kan spara det första delresultatet. Det vanligaste sättet att åstadkomma en sådan temporär lagringsplats är att avdela något minnesutrymme till en så kallad "stack". En *stackpekare* är ett register som pekar på någon minnesadress i detta minnesutrymme. Stackpekaren har två olika operationer:

`PSH`	*push data onto stack*
`PUL`	*pull data from stack*

Ordet “stack” betyder, översatt, “hög”, och det är också så vi ska betrakta operationerna. Dvs, för varje PUSH-instruktion *lägger* vi något överst på högen. För varje PULL-instruktion *plockar vi av det som ligger överst* från högen. Vi ska nu införa en sådan stack-funktion i vår processor, för detta behöver vi ett 16-bitars register (ty detta ska ju innehålla en adress). Vi måste också ha vissa instruktioner för detta registret.

Vi lägger till ett nytt 16-bitars register **S**, (stackregister) hos vår processor. Vi inför också en ny instruktion:

`LD`	*`operand`*`,S`	*load operand into stackpointer*

Stackpekaren kan nu initieras till att peka på toppen av en minnesarea i primärminnet som vi reserverat för temporär lagring av data och adresser enligt:

```
LD    #$2000,S
```

Vi definierar nu PUSH-operationen enligt följande:

PSH *8-bit-register*:

S = S-1	minska värdet hos register S
(S)=register	placera innehållet i *register* på den adress som nu pekas ut av S

Men vi måste också kunna återställa innehållet i ett register från stacken.

PUL *8-bit-register*:

register=(S)	placera innehållet på den adress som pekas ut av S i *register*.
S=S+1	öka värdet hos register S

EXEMPEL: Användning av stacken för att spara och återställa registerinnehåll D

```
LD    #$2000,D    stackpekare initieras
LD    #10,D       värdet 10 nu i D
PSH   D
ADD   #20,D       värdet 20 nu i D
...               gör något med detta...
PUL   D           värdet 10 nu i D
```

Med detta klarar vi alltså av att spara och återställa registerinnehåll. Det löser dock inte problemet med delresultat vi diskuterade tidigare. Det visar sig att vi måste kunna komma åt data som finns på stacken och använda dessa data som operander i våra aritmetiska instruktioner. Vi inför därför

ytterligare ett adresseringssätt som vi kallar *stackregister-indexerad*. Vi skriver detta som:

`(n,S)` där *n* anger en konstant (maximalt 8 bitar) och *S* anger stackregistret. Operandens adress bestäms genom att index (n) adderas till innehållet i S.

Antag att resultatet från vårt tidigare exempel som krävde evaluering av deluttryck ska sparas på adress $300 i minnet. Detta skulle vi då kunna koda:

EXEMPEL: Evaluering av deluttryck

```
0       LD      #$2000,S        sätt upp stackregistret
3       LD      z,D             operand 'z' i deluttryck 2
6       ADD     w,D             adderas till 'w' , resultatet nu i D
9       PSH     D               deluttryck sparas på stacken, dvs adress $1FFF
                                Innehållet i S är nu $1FFF (se även figur i
                                marginalen)
10      LD      x,D             operand 'x' i deluttryck 1
13      ADD     y,D             adderas till 'y', resultatet nu i D
16      SUB     (0,S),D         deluttryck 2 subtraheras från deluttryck 1,
                                resultatet placeras i D
18      ST      D,$300          spara resultatet i minnet
21      PUL     D               återställ ursprungligt innehåll i D och S
```

adress	innehåll
$2000	xx
$1FFF	a+b
$1FFE	xx
$1FFD	xx

4.3.5 Subrutiner

Man vill ofta “modularisera” programmen, dvs sekvenser som ska utföras många gånger i programmet placeras i så kallade *subrutiner* som kan utföras efter ett *anrop* från ett huvudprogram. För att kunna utföra en subrutin krävs att vi på något sätt kan spara adressen till den instruktion som ska utföras *efter* subrutinen. Vi kan använda temporärminne från stacken där vi sparar adressen till denna instruktion. För att kunna återgå, efter subrutinen måste vi också kunna *återställa* denna adress och placera den i PC.

Vi definierar nu två nya instruktioner där stacken används enligt:

`JSR` *adress* — *jump to subroutine* =

```
S = S-1;
(S) = PC(minst sign. byte);
S = S-1;
PC = adress
(S) = PC(mest sign. byte);
```

och instruktionen

`RTS` — *return from subroutine* =

```
PC(mest sign. byte) = (S);
S = S+1;
PC(minst sign. byte) = (S);
S = S+1;
```

EXEMPEL: Anrop av subrutin på adress 60:

```
0       LDS    #$2000       S måste ha ett värde
3       JSR    60           Återhoppsadress (6) sparas på
                            stacken (adress $1FFE). S pekar
                            nu på denna adress.
                            Adress 60 placeras i PC.

6       nästa instruktion   här fortsätter exekvering efter
                            subrutinen
        ....
        ....


60      någon instruktion
        ....
        ....
        RTS                 Återhoppsadress (6) tas från
                            stacken och placeras i PC
                            S pekar därefter på minnescell
                            $2000 igen.
```

Följande exempel illustrerar ytterligare hur stacken kan användas:

EXEMPEL:
Skriv ett program som *räknar* hur många gånger inportens värde är 100 och kontinuerligt skriver ut detta *antal* till utporten.

Lösning:
Vi delar upp programmet i huvudprogram och subrutin.
Vi skapar ett huvudprogram, med början på adress 40, i form av en "oändlig slinga" och en subrutin på adress 60:

```
40      LD     #$2000,S     Initiera stackpekare
43      LD     #0,D         används som räknare
45      JSR    60           vänta tills inport=100
48      ADD    #1,D         lägg till 1
50      ST     $8000,D      skriv till utport
53      JMP    45           upprepa ...
Subrutinen ...
60      PSH    D            Spara innehåll i register D
61      LD     $A000,D      Läs inportens värde
64      ADD    #-100,D      Subtrahera 100
66      JNE    61           Vänta tills inport =100
69      PUL    D            Återställ register D
70      RTS                 Åter från subrutin
```

Låt oss nu, i återstoden av detta kapitel, ägna oss åt att försöka fullständiga instruktionsuppsättningen hos en mycket enkel mikroprocessor.

4.3.6 Mikroprocessorn MP1

Vår enkla mikroprocessor "MP1" har mycket gemensamt med de exempel vi visat tidigare. För att bättre kunna hantera minnesadresser och adressberäkningar har vi infört ett nytt (16-bitars) adressregister, vi kallar detta register 'A'. Vi har då följande registeruppsättning:

Registermodell MP1:

Dataregister D - 8 bitar							
b7	b6	b5	b4	b3	b2	b1	b0

Adressregister A - 16 bitar															
b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

Stackregister S - 16 bitar															
b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

Flaggregister F - 8 bitar varav 4 används							
				N	Z	V	C

- N negative flag - kopia av den mest signifikanta biten (b7 resp b15) i register D eller A.
- Z zero flag - anger om ett resultat är noll eller ej
- V overflow flag - anger om tvåkomplementsspill uppstått
- C carry flag - anger om spill uppstått

Adresseringssätt

Beroende på instruktion kan olika sätt (adresseringssätt) användas för att ange operand/operander. De adresseringssätt som kan användas för en speciell instruktion framgår av instruktionslistan nedan. Följande adresseringssätt tillhandahålls av MP1:

Adresseringssätt	*Skrivsätt*	*Beskrivning*
Immediate data:	`#<data>`	Adressen till data ges implicit i instruktionen. Data följer omedelbart efter operationskoden
Adress:	`<adress>`	Adressen till data ges här direkt
Adressregister indexerad indirekt:	`(n,A)`	Adressen till data bildas genom att en positiv offset (0-255) adderas till innehållet i adressregister A.
Stackregister indexerad indirekt:	`(n,S)`	Adressen till data bildas genom att en positiv offset (0-255) adderas till innehållet i stackregister S.

Instruktionsuppsättningen innehåller instruktioner för att påverka såväl registerinnehåll som minnesinnehåll på olika sätt.

Följande tabell sammanfattar instruktionsuppsättningen, tillåtna adresseringssätt och flaggpåverkan.

Instruktion	Dest. Reg.	#< >	addr	(n,A)	(n,S)	Kort Beskrivning	Flaggor N	Z	V	C
LD	D	X	X	1	1	Load register	*	*	0	0
LD	A	X	X	1	1					
LD	S	X	X	1	1					
ST	D		X	1	1	Store register	-	-	-	-
ST	A		X	1	1					
ST	S		X	1	1					
LSL	D					Logical shift left	*	*	*	*
LSL	A									
LSR	D					Logical shift right	*	*	0	*
LSR	A									
ASR	D					Arithmetic shift right	*	*	0	*
ASR	A									
JSR			X	X		Jump to subroutine	-	-	-	-
JMP			X	X		Jump to address	-	-	-	-
PSH	D					Push register contents	-	-	-	-
PUL	D					Pull register contents	-	-	-	-
PSHH	A					Push high register contents	-	-	-	-
PULH	A					Pull high register contents	-	-	-	-
PSHL	A					Push low register contents	-	-	-	-
PULL	A					Pull low register contents	-	-	-	-
RTS						Return from subroutine	-	-	-	-
JEQ			X			Jump if equal	-	-	-	-
JNE			X			Jump if not equal	-	-	-	-
JCS			X			Jump if carry set	-	-	-	-
JCC			X			Jump if carry clear	-	-	-	-
JVS			X			Jump if overflow set	-	-	-	-
JVC			X			Jump if overflow clear	-	-	-	-
JLT			X			Jump if less than	-	-	-	-
JHI			X			Jump if higher	-	-	-	-
ADD	D	X	X	X	X	Add binary	*	*	*	*
ADD	A	X	X	X	X					
ADDC	D	X	X	X	X	Add binary with carry	*	*	*	*
SUB	D	X	X	X	X	Subtract binary	*	*	*	*
SUB	A	X	X	X	X					
SUBC	D	X	X	X	X	Subtract binary with carry	*	*	*	*
AND	D	X	X	X	X	Logical AND	*	*	-	-
OR	D	X	X	X	X	Logical OR	*	*	-	-
EOR	D	X	X	X	X	Logical EXCLUSIVE OR	*	*	-	-
CMP	D	X	X	X	X	Compare	*	*	*	*
CMP	A	X	X	X	X					

Anm. Flaggor:

*	sätts beroende på operationens resultat
-	påverkas inte
0	nollställs alltid

Instruktionerna beskrivs detaljerat i *instruktionslistan* i nästa avsnitt.

4.3.7 Instruktionslista för MP1

LD *load register*

Laddar ett register med data från effektiva adressen.

Former:
```
LD   <EA>,D    register D
LD   <EA>,A    register A
LD   <EA>,S    stackpekare S
```

	#<data>	<adress>	(n,A)	(n,S)
EA	x	x	x	x

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Nollställs alltid
V	Nollställs alltid
N	Sätts om resultatet är negativt, nollställs annars

ST *store register*

Skriver innehållet i ett register till effektiva adressen.

Former:
```
ST   D,<EA>    register D
ST   A,<EA>    register A
ST   S,<EA>    stackpekare S
```

	#<data>	<adress>	(n,A)	(n,S)
EA	-	x	x	x

Flaggpåverkan	
Z	Påverkas ej
C	Påverkas ej
V	Påverkas ej
N	Påverkas ej

ADD *add to register*

Adderar innehållet i register till innehållet på effektiva adressen och placerar resultatet i register.

Former:
```
ADD  <EA>,D    (D)+(EA)->register D
ADD  <EA>,A    (A)+(EA)->register A
```

	#<data>	<adress>	(n,A)	(n,S)
EA	x	x	x	x

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Sätts om operationen resulterade i spill (tal utan tecken) nollställs annars
V	Sätts om operationen resulterade i spill (tal med tecken) nollställs annars
N	Sätts om resultatet är negativt, nollställs annars

ADDC *add with Carry to register*

Adderar carryflaggan till innehållet i register och innehållet på effektiva adressen och placerar resultatet i register.

Former:

`ADDC <EA>,D` Carry+(D)+(EA)->register D

	#<data>	<adress>	(n,A)	(n,S)
EA	x	x	x	x

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Sätts om operationen resulterade i spill (tal utan tecken) nollställs annars
V	Sätts om operationen resulterade i spill (tal med tecken) nollställs annars
N	Sätts om resultatet är negativt, nollställs annars

SUB *subtract from register*

Subtraherar innehållet på effektiva adressen från innehållet i register och placerar resultatet i register.

Former: `SUB <EA>,D` (D)-(EA)->register D

`SUB <EA>,A` (A)-(EA)->register A

	#<data>	<adress>	(n,A)	(n,S)
EA	x	x	x	x

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Sätts om lånesiffra använts för operationen (tal utan tecken) nollställs annars
V	Sätts om lånesiffra använts för operationen (tal med tecken) nollställs annars
N	Sätts om resultatet är negativt, nollställs annars

SUBC *subtract with Carry from register*

Subtraherar innehållet på effektiva adressen och carryflaggan från innehållet i register och och placerar resultatet i register.

Former:

`SUBC <EA>,D` (D) - Carry - (EA)->register D

	#<data>	<adress>	(n,A)	(n,S)
EA	x	x	x	x

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Sätts om lånesiffra använts för operationen (tal utan tecken) nollställs annars
V	Sätts om lånesiffra använts för operationen (tal med tecken) nollställs annars
N	Sätts om resultatet är negativt, nollställs annars

AND *logical and to register*

Utför logiskt "AND" mellan innehållet i register och innehållet på effektiva adressen och placerar resultatet i register.

Former: `AND <EA>,D` (D)∧(EA)->register D

	#<data>	<adress>	(n,A)	(n,S)
EA	x	x	x	x

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Nollställs alltid
V	Nollställs alltid
N	Sätts om resultatet är negativt, nollställs annars

OR *logical or to register*

Utför logiskt "OR" mellan innehållet i register och innehållet på effektiva adressen och placerar resultatet i register.

Former: `OR <EA>,D` (D)∨(EA)->register D

	#<data>	<adress>	(n,A)	(n,S)
EA	x	x	x	x

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Nollställs alltid
V	Nollställs alltid
N	Sätts om resultatet är negativt, nollställs annars

EOR *logical exclusive OR to register*

Utför logiskt "EXCLUSIVE OR" mellan innehållet i register och innehållet på effektiva adressen och placerar resultatet i register.

Former: `AND <EA>,D` (D)⊕(EA)->register D

	#<data>	<adress>	(n,A)	(n,S)
EA	x	x	x	x

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Nollställs alltid
V	Nollställs alltid
N	Sätts om resultatet är negativt, nollställs annars

CMP *compare from register*

Subtraherar innehållet på effektiva adressen från innehållet i register. Registerinnehållet ändras ej av instruktionen. Endast flaggorna i statusregistret påverkas.

Former: `CMP <EA>,D` (D)-(EA)->flaggor
`CMP <EA>,A` (A)-(EA)->flaggor

	#<data>	<adress>	(n,A)	(n,S)
EA	x	x	x	x

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Sätts om lånesiffra använts för operationen (tal utan tecken) nollställs annars
V	Sätts om lånesiffra använts för operationen (tal med tecken) nollställs annars
N	Sätts om resultatet är negativt, nollställs annars

LSL *logical shift left*

Skiftar innehållet i register ett steg till vänster. Operandens mest signifikanta bit skiftas till Carry-flaggan. En nolla skiftas alltid in till den minst signifikanta biten.

Former: `LSL D`
`LSL A`

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Mest signifikanta bit av operanden före skiftet
V	Sätts om C och mest signifikant bit i operanden (efter skiftet) är olika.
N	Sätts om resultatet är negativt, nollställs annars

LSR *logical shift right*

Skiftar innehållet i register ett steg till höger. Minst signifikant bit skiftas till Carry. En nolla skiftas in i den mest signifikanta positionen.

Former: `LSR D`
`LSR A`

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Sätts om minst signifikant bit av operanden (före skiftet) är 1, nollställs annars
V	Nollställs alltid
N	Sätts om resultatet är negativt, nollställs annars

ASR *arithmetic shift right*

Skiftar innehållet i register ett steg till höger. Den mest signifikanta biten skiftas in som ny mest signifikant bit. Talet behåller alltså sitt tecken.

Former:
```
ASR  D
ASR  A
```

Flaggpåverkan	
Z	Sätts om resultatet är 0, nollställs annars
C	Sätts om operationen resulterade i spill (tal utan tecken) nollställs annars
V	Nollställs alltid
N	Sätts om resultatet är negativt, nollställs annars

JSR *jump to subroutine*

Utför subrutin. Adressen till nästa instruktion placeras på stacken. <EA> placeras i PC, därefter fortsätter instruktionsexekveringen

Former:
```
JSR  address
JSR  (n,A)
```

	#<data>	<adress>	(n,A)	(n,S)
EA	-	x	x	-

Flaggpåverkan	
Z	Påverkas ej
C	Påverkas ej
V	Påverkas ej
N	Påverkas ej

JMP *jump*

Utför programflödesänding. <EA> placeras i PC, därefter fortsätter instruktionsexekveringen

Former:
```
JMP  address
JMP  (n,A)
```

	#<data>	<adress>	(n,A)	(n,S)
EA	-	x	x	-

Flaggpåverkan	
Z	Påverkas ej
C	Påverkas ej
V	Påverkas ej
N	Påverkas ej

Jcc *jump conditionally*
Utför villkorlig programflödesänding. Villkoret (*condition codes*) testas först. Om detta är *sant* placeras <EA> i PC, därefter fortsätter instruktionsexekveringen om inte fortsätter exekveringen vid nästa instruktion.

Form: `Jcc address`

Mnemonic	Villkor	Flaggvillkor
JEQ	*Equal*	Z=1
JNE	*Not Equal*	Z=0
JCS	*Carry set*	C=1
JCC	*Carry clear*	C=0
JVS	*Overflow set*	V=1
JVC	*Overflow clear*	V=0
JLT	*Less than (signed)*	$N \wedge \overline{V} \vee \overline{N} \wedge V$
JHI	*Higher (signed)*	$N \wedge V \wedge \overline{Z} \vee \overline{N} \wedge \overline{V} \wedge \overline{Z}$

Anm: Flaggvillkoren för de båda sista instruktionerna har utformats för att ge jämförelser av tal med tecken.

Flaggpåverkan	
Z	Ingen
C	Ingen
V	Ingen
N	Ingen

Operationskoder
Det återstår nu att bestämma operationskoder för instruktions-uppsättningen och adresseringssätten. Valet av operationskoder kan göras på en mångfald olika sätt. I detta fall delar vi in instruktionerna i sex olika format och tilldelar dom operationskoder enligt följande mall

7	6	5	4	3	2	1	0	<-***Bitnummer***
1	x	x	x	x	D/A	x	x	Aritmetiska/Logiska
0	1	1	1	x	x	x	x	Load
0	1	1	0	x	x	x	x	Store
0	1	0	0	x	x	x	x	Skift/JSR/JMP
0	0	0	0	1	x	x	x	Villkorlig flödeskontroll
0	1	0	1	0	x	x	x	Övriga

Anm:
- X betyder "don' care" dvs kan vara 0 eller 1 i respektive grupp
- D/A är 1 om dataregistret ingår i operationen, 0 om adressregistret ingår i operationen

4.3.8 Instruktioner/Opkoder MP1

Följande tabell sammanfattar samtliga tillåtna instruktioner/ adresseringssätt och anger dessutom hur många bytes respektive instruktion upptar. Tabellen innehåller också ett nytt adresseringssätt, "inherent" (Inh), vilken innebär att eventuella operander framgår direkt av operationskoden. Samtliga värden i tabellen ges på hexadecimal form.

Instruktion	Dest. Reg.	Imm OP	#	Addr OP	#	Ind,A OP	#	Ind,S OP	#	Inh OP	#
LD	D	70	2	71	3	72	2	73	2		
LD	A	74	3	75	3	76	2	77	2		
LD	S	78	3	79	3	7A	2	7B	2		
ST	D			61	3	62	2	63	2		
ST	A			65	3	66	2	67	2		
ST	S			69	3	6A	2	6B	2		
LSL	D									40	1
LSL	A									41	1
LSR	D									42	1
LSR	A									43	1
ASR	D									44	1
ASR	A									45	1
JSR				46	3	48	2				
JMP				47	3	49	2				
PSH	D									50	1
PUL	D									51	1
PSHH	A									52	1
PULH	A									53	1
PSHL	A									54	1
PULL	A									55	1
RTS										56	1
JEQ				10	3						
JNE				11	3						
JCS				12	3						
JCC				13	3						
JVS				14	3						
JVC				15	3						
JLT				16	3						
JHI				17	3						
ADD	D	80	2	81	3	82	2	83	2		
ADD	A	84	3	85	3	86	2	87	2		
ADDC	D	A8	2	A9	3	AA	2	AB	2		
SUB	D	88	2	89	3	8A	2	8B	2		
SUB	A	8C	3	8D	3	8E	2	8F	2		
SUBC	D	B0	2	B1	3	B2	2	B3	2		
AND	D	C0	2	C1	3	C2	2	C3	2		
OR	D	C8	2	C9	3	CA	2	CB	2		
EOR	D	D0	2	D1	3	D2	2	D3	2		
CMP	D	90	2	91	3	92	2	93	2		
CMP	A	94	3	95	3	96	2	97	2		

4.3.9 Programexempel för MP1

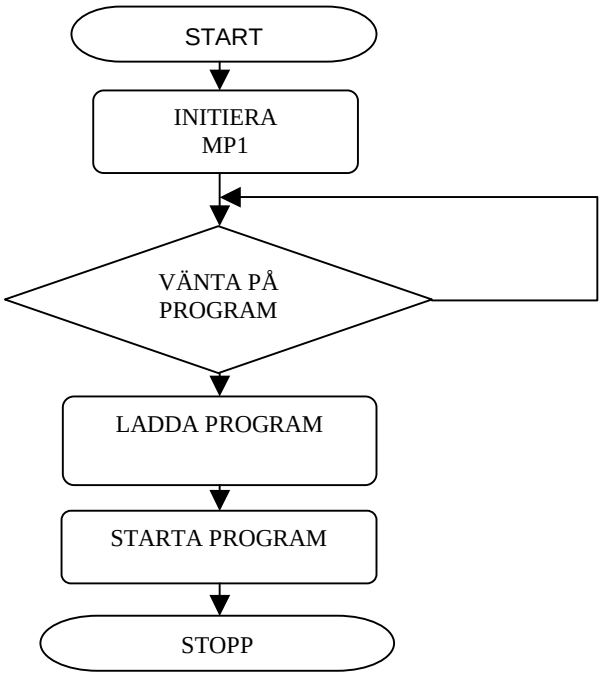
Låt oss nu illustrera användningen av en mikrodator bestyckad med MP1 med några enkla exempel. För exemplen utgår vi i från en konfiguration som den som beskrevs i kapitel 2. Vi rekapitulerar förutsättningarna:

- RWM-minnet ska aktiveras om MP1 genererar någon av adresserna 0-3FFF.
- ROM-minnet ska aktiveras om MP1 genererar någon av adresserna C000-DFFF.

- Inporten ska aktiveras om MP1 genererar adress A000.
- Utporten ska aktiveras om MP1 genererar adress 8000.

4.3.10 Programmering av MP1

Vi ska nu illustrera hur denna dator, uppbyggd kring vår mikroprocessor "MP1" kan förses med enkel, grundläggande programvara. Vi kommer att konstruera en enkel laddningsprocedur med vars hjälp vi kan överföra ett program för MP1 från någon annan dator, via MP1's portar till MP1's RWM-minne. Denna laddningsprocedur måste alltså först ha placerats i MP1's ROM-minne. Då MP1 startar vid spänningspåslag ska följande utföras.



Vi definierar nu följande protokoll för överföring av program mellan MP1 och den andra datorn.

1. MP1 skriver värdet 2 till utporten, detta tolkas av den andra datorn som att MP1 är klar att ta emot program
2. MP1 läser nu, byte för byte, från inporten. MP1 förväntar att ett program inleds med det hexadecimala värdet $EE (en byte).
3. Därefter följer programmets startadress, dvs den adress det ska placeras på i MP1's primärminne. Denna adress är två bytes och den mest signifikanta byten kommer först.
4. Efter startadressen följer programmets slutadress på samma sätt.
5. Efter slutadressen kommer ett antal bytes (*slutadress-startadress+1*) som ska placeras konsekutivt med början på *startadress*.

Vi måste också införa ett protokoll för överföring av enstaka bytes. Om vi inte gör detta riskerar vi att MP1 inte hinner med den andra datorns arbetstakt. Vi definierar protokollet för överföring av en enstaka byte enligt följande:

1. MP1 skriver värdet 1 till utporten. Den andra datorn tolkar detta som att MP1 är klar att ta emot en byte.
2. MP1 skriver värdet 0 till utporten. Den andra datorn tolkar det som att MP1 tagit emot data.

Subrutin 1 "getByte"

Subrutinen ska läsa en byte som överförts från den andra datorn via MP1's inport. Denna byte ska finnas i register D då subrutinen avslutas. Funktionen blir:

1. Sätt bit 1 på MP1's utport till 1 som en signal till en yttre enhet att MP1 är beredd att läsa en byte från MP1's inport.
2. Läser en byte från MP1's inport, denna byte ska subrutinen returnera i D-registret.
3. Nollställer bit 1 på MP1's utport som en signal till en yttre enhet att data har lästs från MP1's inport.

Anm. Vi vet ännu inte var (i MP1's primärminne) koden för denna subrutin ska placeras. Vi skriver därför koden med symboliska adresser. En symbolisk adress anges genom att symbolen skrivs först på en rad. Instruktioner kan sedan referera till symbolen:

```
getByte:    LD    #1,D
            ST    D,$8000      "klar att ta emot"
            LD    $A000,D      ta emot data
            PSH   D            spara temporärt
            LD    #0,D
            ST    D,$8000      "tagit emot data"
            PUL   D            återställ databyte till D
            RTS
```

Huvudprogram

Vi bygger nu upp vårt huvudprogram enligt flödesschemat:

```
            ORG   $C000        förutsätt startadress för detta program
START:

INIT:       LD    $4000,S      initiera stackpekare "top of RWM"
            LD    #2,D
            ST    D,$8000      signalera "klar att ta emot program"
WAIT:
            JSR   getByte      läs en byte
            CMP   #$EE,D
            JNE   WAIT         vänta tills "start of program"
LOADPROG:
            JSR   getByte      startadress hög byte
            PSH   D            spara på stacken
            JSR   getByte      startadress låg byte
            PSH   D            spara på stacken

            JSR   getByte      slutadress hög byte
            PSH   D            spara på stacken
            JSR   getByte      slutadress låg byte
            PSH   D            spara på stacken
```

```
* Vi är nu klara att börja överföra data från den andra datorn. Startadress för data har vi
* på stacken (16 bitars värde), offset 2.  Slutadressen ges av ett 16-bitars värde på
* stacktoppen dvs offset 0

            LD      (2,S),A         startadress nu i A
LOADDATA:
            JSR     getByte
            ST      (0,A)
            CMP     (0,S),A         sista databyte?
            JEQ     STARTPROG
            ADD     #1,A
            JMP     LOADDATA

STARTPROG:
            LD      (2,S),A         startadress nu i A
            JMP     (0,A)           starta det laddade programmet
STOPP:
            JMP     START
```

Vi har nu konstruerat en så kallad *bootstrap-procedur*, dvs ett program som inte har någon annan uppgift än att ladda och starta ett annat (mer avancerat) program. I exemplet använder vi oss av en "annan dator" som källa för det andra programmet men det kan lika gärna vara någon form av icke flyktigt sekundärt minne som till exempel en flexskiveenhet eller hårddisk.

4.3.11 Sammanfattning

Vi har nu illustrerat en rad viktiga egenskaper hos en mikroprocessor. Vi har gjort detta med hjälp av en “påhittad” konstruktion av mycket enkel typ. Fortsättningsvis kommer vi att behandla en *verklig* mikroprocessor som skiljer sig en del från den vi illustrerat här. Du kan dock använda de resonemang vi fört i dessa avsnitt, kring instruktionsuppsättning och adresseringssätt.

4.4 Övningsuppgifter

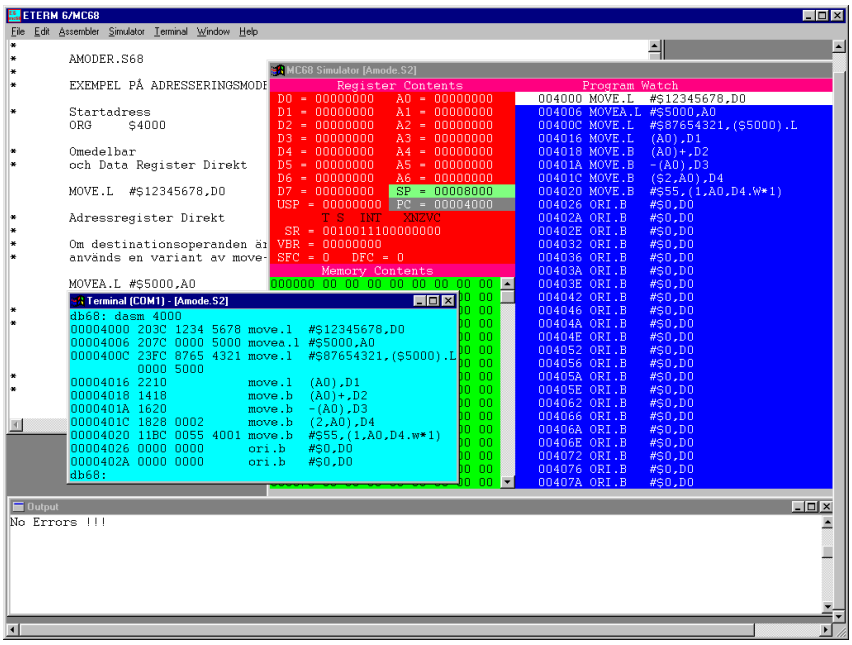
Uppgift 21: Skriv, för MP1, en sekvens instruktioner som två-komplementerar innehållet i register D.

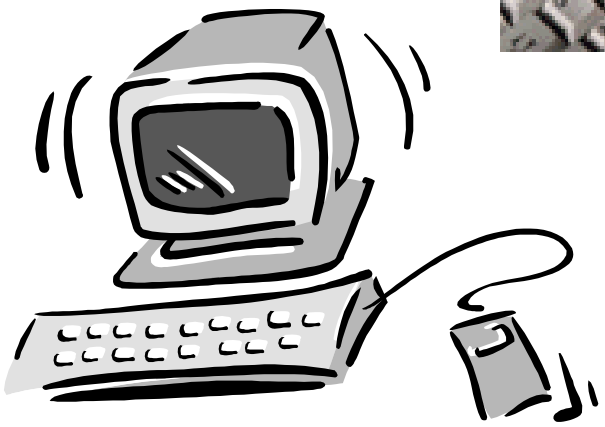
Uppgift 22: Skriv, för MP1, en sekvens instruktioner som adderar de två översta 8-bitars talen på stacken och ersätter dessa med resultatet.

Uppgift 23: Skriv en subrutin för MP1 som utför addition av två 8-bitars operander vilka är placerade konsekutivt i minnet. Adressregister A innehåller adressen till den första operanden. Resultatet ska finnas i register D efter det att subrutinen utförts.

Uppgift 24: Skriv en subrutin för MP1 som utför multiplikation av innehållen i register D och de 8 minst signifikanta bitarna i register A. Efter subrutinen ska resultatet finnas i register A. Eventuellt spill ignoreras.

5. Assemblerprogrammering





I detta kapitel ges en introduktion till assemblerprogrammering för mikrocontrollern MC68340. I Appendix A finner du en fullständig instruktionslista och dessutom beskrivning av de assemblerdirektiv som finns tillgängliga.

5.1 Utvecklingsmiljö

För att kunna skriva och testa program krävs en så kallad *utvecklingsmiljö*. I utvecklingsmiljön ingår *verktyg* som används för att genomföra flera olika moment. De verktyg som används varierar något beroende på vilken typ av programutveckling som ska utföras och man kan identifiera två huvudkategorier.

Den första kategorin omfattar system där programmet kan utvecklas på *samma dator* som det sedan ska användas. Den andra kategorin omfattar system där programmet utvecklas på en typ av dator *för* en helt annan typ av dator. En sådan utvecklingsmiljö kallas för *kors-utvecklingsmiljö*. I korsutvecklingsmiljön ingår en *värddator* och en *måldator*. Värddatorn används för att skapa de program som skall utföras av *måldatorn*. Korsutveckling är vanlig exempelvis då det gäller att konstruera små, inbyggda styrsystem.

Utvecklingsmiljön *ETERM* är speciellt avsedd för korsutveckling där värddatorn är en persondator (IBM-PC) och måldatorn är en enkel enkortsdator som exempelvis *MC68*. En grunduppsättning verktyg består av: Persondator (IBM-PC) med programvara *ETERM*, enkortsdator *MC68*, stabiliserad strömförsörjning 5 Volt och en seriekabel för anslutning mellan persondatorn och enkortsdatorn. Programutvecklingen sker i flera steg:

- Använd ett *textredigeringsverktyg* för att skapa en *källtextfil* med ett assemblerprogram för måldatorn
- Använd en *assemblator* för att översätta källtextfilen till *maskinkod* för måldatorn (assemblera)
- Använd *terminalemuleringsprogrammet* för att ladda maskinkoden till måldatorn och testa programmet i måldatorn, ***eller***
- Använd *simulator* för att testa programmet.

För att kunna utnyttja utvecklingsmiljön på rätt sätt är det viktigt att förstå dess övergripande funktion. Steg 1,2 och 4, ovan, utförs *enbart* med hjälp av värddatorn. Under steg 3 används värddatorn för att överföra maskinkoden till måldatorn därefter används praktiskt taget bara värddatorns tangentbord och bildskärm vid sidan av måldatorn. Vi säger då att värddatorn fungerar som *terminal*.

Figur 5.1 *Mikrodator MC68*

5.2 Mikrodatorn MC68

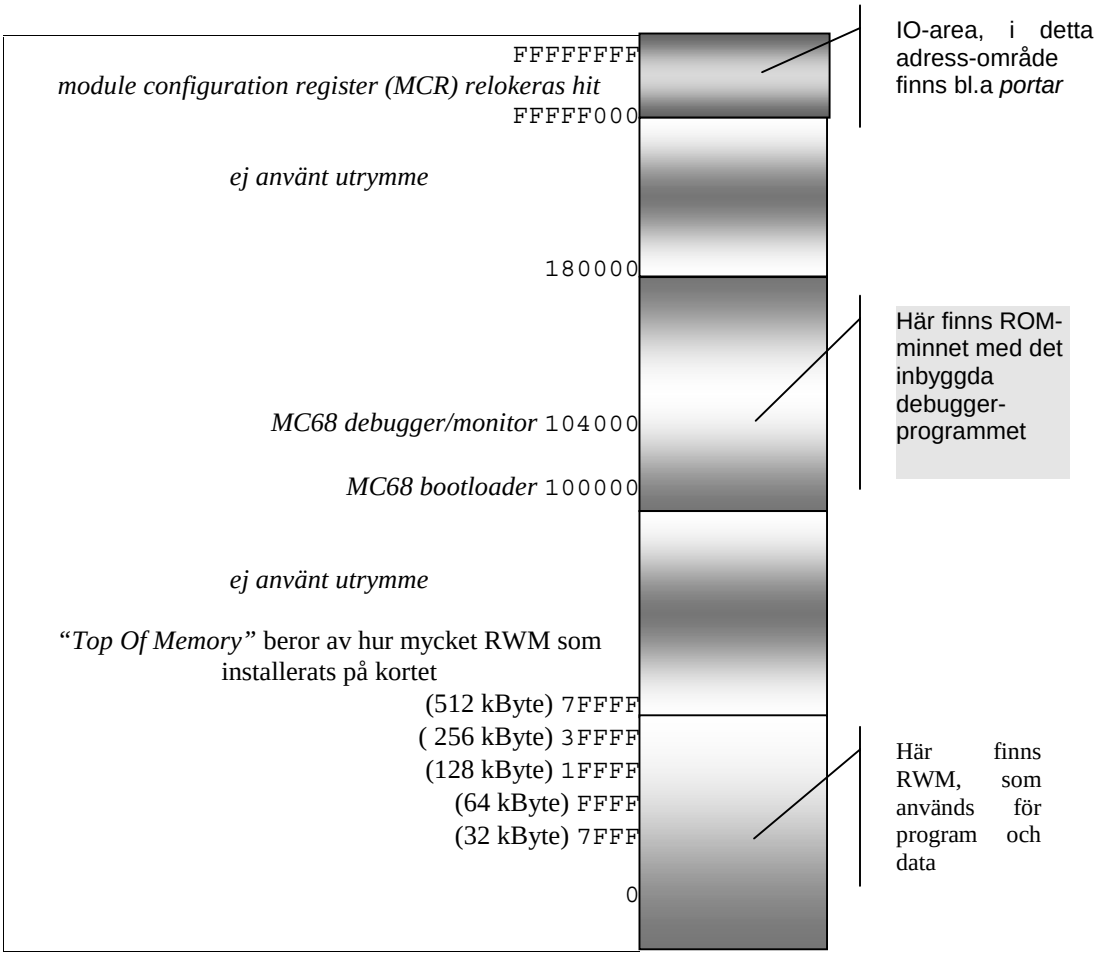
MC68 är en liten enkortsdator som byggts kring Motorolas microcontroller MC68340. På kortet finns plats för maximalt 512 kByte RWM respektive 512 kByte FLASH-PROM men det är i standard-bestyckningen försett med 32 kByte RWM respektive 128 kByte FLASH-PROM.

5.2.1 Mikrocontrollern MC68340

MC68340 är en så kallad *microcontroller* dvs en krets där såväl centralenheten som andra funktioner byggts samman (integrerats) i en kapsel. Centralenheten utgörs av en *processorkärna* som kallas CPU-32 och bland de perifera funktionerna (periferikretsar beskrivs i kapitel 6) märks bl. a två serieportar, av vilka den ena används för kommunikationen värddator/*MC68*, en 8-bitars parallell inport och en 8-bitars parallell utport.

5.2.2 Minnesdisposition

MC68's minnesdisposition anger de adressområden som kan användas och *hur* de kan användas. Dessutom anger minnesdispositionen adresserna till systemets yttre enheter.



Figur 5.2 *MC68 minnesdisposition*

Då du startar *MC68* kontrolleras datorn av det inbyggda *debugger*-programmet (*db68*). Detta är alltså placerat i PROM-minne, dels för att behålla innehållet då spänningen till *MC68* slås av, men också för att andra program inte av misstag ska kunna ändra debugger-programmet.

Under programutvecklingen placeras det program som ska testas i RWM-minnet. Detta sker genom en enkel *laddningsprocedur* där programmet överförs från *värddatorn* till *måldatorns* minne. Efter laddningen kan programmet testas med hjälp av debuggern.

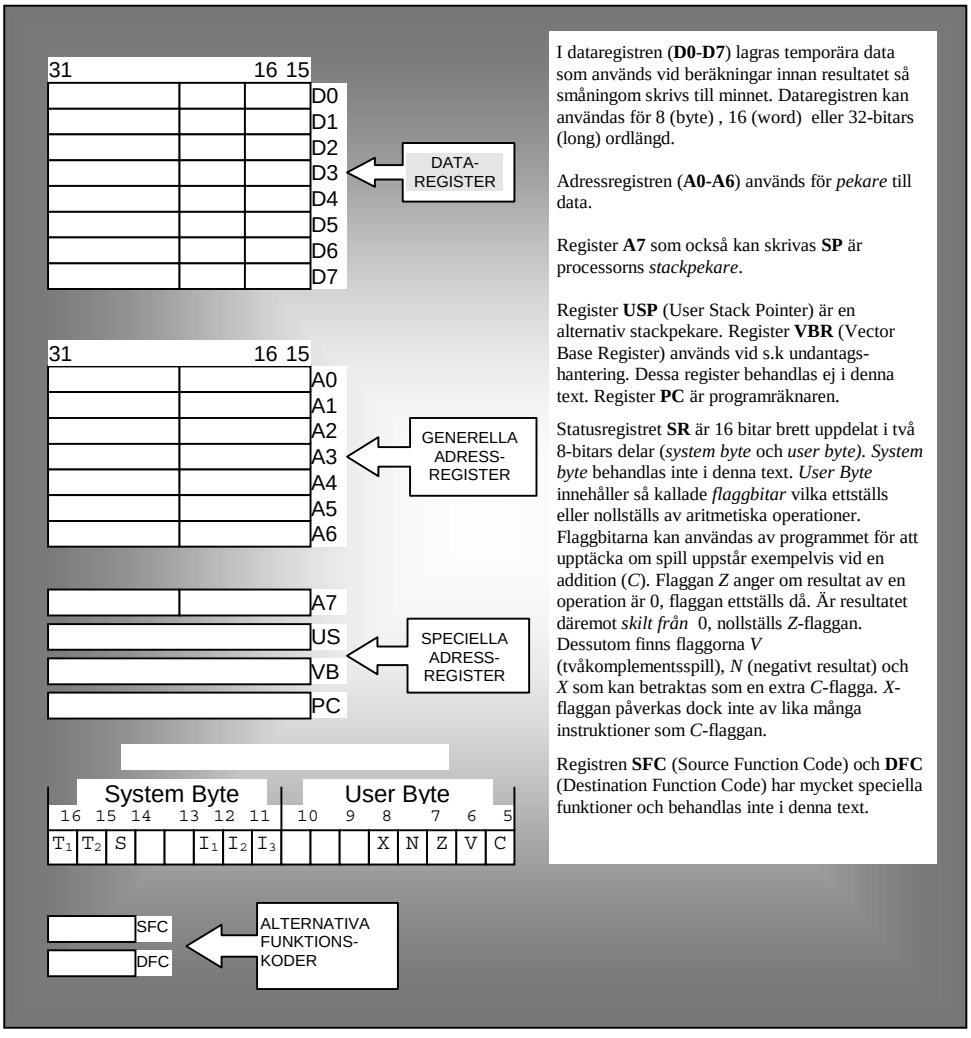
Mikrodatorn kommunicerar med *omvärlden* via in- och ut-portar och dessa har placerats på *bestämda* adresser grupperade i den så kallade *IO-arean*.

5.2.3 Programmerarens bild

Med *programmerarens bild* av en mikroprocessor menar man dess *instruktionsuppsättning, register* och dess *adresseringsätt*. CPU-32 är en avancerad processorkärna med åtskilliga instruktioner och flera olika adresseringssätt. Vi kommer här att behandla processorn's register, de vanligaste instruktionerna och de enklaste adresseringssätten.

5.2.4 Register

Mikrocontrollern MC68340 har flera register som används av programmeraren för tillfällig lagring. Det finns två typer, *dataregister* respektive *adressregister*. Adressregistren används som regel för att lagra *pekare* till data, medan dataregistren, som namnet säger, används direkt för data. Utöver dessa, finns några *speciella* register, vi kommer inte, inom ramen för denna kurs, att hinna med en utförlig beskrivning av hur dessa speciella register används, utan ger här bara snabb översikt. Det viktiga för fortsättningen är att du nu koncentrerar dig på hur dataregistren används, och att du observerar *flaggbitarna* för aritmetiska operationer i *statusregistret*.



Figur 5.3 *MC68340 Registeruppsättning*

Instruktionsuppsättning
Processorns instruktioner kan delas in i grupper där likartade instruktioner samlas inom varje grupp. Det finns bland annat instruktioner för att kopiera data, aritmetiska/logiska instruktioner och instruktioner för att styra programflödet.

Olika instruktioner kan ha olika antal operander. Aritmetiska instruktioner utgör exempel på instruktioner med två operander. Det finns instruktioner som bara använder en operand och det finns instruktioner som saknar operand.

EXEMPEL
Instruktionen
```
      MOVE.B     #1,($5000).L
```
placerar talet 1 på adress $5000. Ettan kallas *källoperand*, adress $5000 är *destinationsoperand*.

EXEMPEL
Instruktionen
```
      ADDI.B     #1,($5000).L
```
adderar 1 till innehållet på adress $5000.

EXEMPEL
Instruktionen
```
      CLR.B            ($3000).L
```
nollställer innehållet på adress $3000

5.2.5 Adresseringssätt
MC68340 erbjuder inte mindre än *14 olika möjligheter* att ange instruktionernas operander, dessa sammanfattas tabellen nedan. Vi kommer att beskriva och i fortsättningen använda endast ett fåtal av dessa, tabellen finns med för att du ska *känna till* att det finns fler möjligheter än de du använder i denna kurs.

Adresseringssätt	Adressbildning	Syntax
Dataregister Direkt	EA = Dn	**Dn**
Adressregister Direkt	EA = An	**An**
Adressregister Indirekt	EA = (An)	**(An)**
Adressregister Indirekt, postinkrement	EA = (An), An=An+SIZE	**(An)+**
Adressregister Indirekt, predekrement	EA = An=An-SIZE, (An)	**-(An)**
Adressregister Indirekt med 16 bitars offset	EA = (An)+ d16	**(d16,An)**
Adressregister Indirekt med index och 8 bitars offset	EA = (An)+(Xn)+d8	**(d8,An,Xn.SIZE)**
Adressregister Indirekt med index och basoffset	EA=(An)+(Xn*SCALE)+disp	**(disp,An,Xn.SIZE*SCALE)**
Absolut kort	EA = nästa 16-bitars ord	**(xxx).W**
Absolut lång	EA = två nästa16 bitars ord	**(xxx).L**
Programräknarrelativ med 16 bitars offset	EA = (PC)+d16	**(d16,PC)**
Programräknarrelativ med index och 8 bitars offset	EA = (PC)+(Xn)+d8	**(d8,PC,Xn.SIZE)**
Programräknarrelativ med index och basoffset	EA=(PC)+(Xn*SCALE)+disp	**(disp,PC,Xn.SIZE*SCALE)**
Omedelbar	EA = (PC)+2	**#data**

Tabell 5.1 *MC68340 adresseringssätt*

Effektiv adressberäkning

Med en operands *effektivadress* (EA) anger man *var* operandens data finns. Man bör inte ta ordet "adress" i uttrycket för bokstavligt. EA kan exempelvis vara en minnesadress, men också ett register. I följande avsnitt kommer vi att beskriva de adresseringssätt (dvs *sätt* att ange EA) som huvudsakligen kommer att användas i denna kurs.

Dataregister direkt

Dataregister direkt används då man vill påverka innehållet i något dataregister. Varje dataregister kan lagra 32 bitar men vi behöver inte nödvändigtvis påverka samtliga dessa bitar. För åtskilliga instruktioner, där dataregister kan användas, kan man också ange *storlek*. Denna kan vara *byte* (8 bitar), *word* (16 bitar) eller *long* (samtliga 32 bitar). Det finns 8 identiska dataregister och de namnges: **D0**, **D1**, **D2** osv till **D7**. Du kan, om du vill använda små bokstäver, dvs det går lika bra att ange registren som **d0**, **d1**, **d2** osv till **d7**.

Några exempel:

```
CLR.L           D4
```

Instruktion: *clear long D4* nollställer *samtliga* 32 bitar i dataregister D4. Innehållet i D4 blir *efter* instruktionen:
`00000000 00000000 00000000`

```
CLR.W           D2
```

Instruktionen *clear word D2* nollställer *de 16 minst signifikanta bitarna* i dataregister D2. Övriga bitar påverkas ej.Om exempelvis D2 från början innehåller
`10100111 11001010 01011100 11010111`
blir innehållet i D2 *efter* instruktionen:
`10100111 11001010 00000000 00000000`

```
CLR.B           D0
```

Instruktionen *clear byte D0* nollställer *de 8 minst signifikanta bitarna* i dataregister D0. Övriga bitar påverkas ej. Om D0 från början innehåller
`10100111 11001010 01011100 11010111`
blir innehållet i D0 *efter* instruktionen:
`10100111 11001010 01011100 00000000`

Omedelbar

Omedelbar (eng. *immediate*) addressering används för att ange en *konstant*. Oftast används också en speciell instruktionsform, exempelvis för instruktionen `ADD`, som finns i flera former. Med detta adresseringssätt används formen `ADDI` (*add immediate*). För att addera en 8-bitars konstant till innehållet i ett dataregister:

```
ADDI.B     #konstant,Dn
```

Tecknet # anger att detta är addresseringssättet *omedelbar*. Konstanten kan vara ett tal:
0-255 (max 8 bitar), n anger ett valfritt dataregister (0-7).

Absolut

Adresseringsmoden *absolut* används då man vill läsa eller skriva från/till en bestämd adress i minnet.

EXEMPEL

```
MOVE.B    ($5000).L,D0
```

Källoperanden är adress $5000, destinationsoperanden är register D0. Efter instruktionen kommer alltså innehållet på adress $5000 att ha kopierats till register D0.

EXEMPEL

```
MOVE.B    D0,($5000).L
```

Innehållet i register D0 kopieras till adress $5000

EXEMPEL

```
MOVE.B    ($5000).L,($5004).L
```

Innehållet på adress $5000 kopieras till adress $5004

Observera att detta adresseringssätt kan användas i två olika former:

Absolute Word `(adress).W`
respektive
Absolute Long `(adress).L`

I den första formen kodas adressen i form av en 16-bitars konstant. Denna konstant teckenutvidgas till 32 bitar vid adressberäkningen. Detta innebär att endast adressområdena

$0000-$7FFF

respektive

$FFFF8000-$FFFFFFFF

kan anges med den kortare formen.

Om du är osäker på vilka absoluta adresser som du kommer att använda så utnyttja den längre formen för säkerhets skull.

Adressregister direkt

För att placera en adress i ett adressregister används *adressregister direkt*. Adresseringsmoden liknar *dataregister direkt* men instruktionens form påverkas som regel. Exempelvis används `MOVEA` (*move to address register*) i stället för `MOVE`.

EXEMPEL

```
MOVEA.L   #$5000,A0
```

placerar adressen $5000 i register **A0**.

Adressregister indirekt

Adressregister indirekt ger möjlighet att adressera via en pekare. Operanden finns på den adress som anges i adressregistret.

EXEMPEL:

```
MOVEA.L   #$5000,A0
MOVE.B    (A0),D0
```

placerar innehållet på adress $5000 i register **D0**.

5.3 Introduktion till assemblerprogrammering

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

Du finner en fullständig instruktionslista och beskrivning av samtliga assemblerdirektiv i Appendix A

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

En rad, i assemblerprogrammet delas in i maximalt 4 fält. Första fältet används enbart för att markera ett läge (en symbol eller en "etikett"). Man väljer då ett *symboliskt namn* och kan därefter använda detta namn som exempelvis operand till instruktioner. Anledningen till att man använder sådana symboliska namn är att man då slipper skriva *absoluta* minnesadresser i programmet. Se nedan

```
start:
```

Symbolnamnet ("start") måste börja i radens första position. Kolon (:) efter symbolen, kan men behöver inte anges

```
start:
        …
        …
                JMP             (start).L
```

Symbolfält

Instruktion *eller* direktiv

Exempel på tillåtna symbolnamn

```
start
stopp
prog10
_prog10
```

Exempel på felaktig (multipel) definition av symbol

```
start:

start:
```

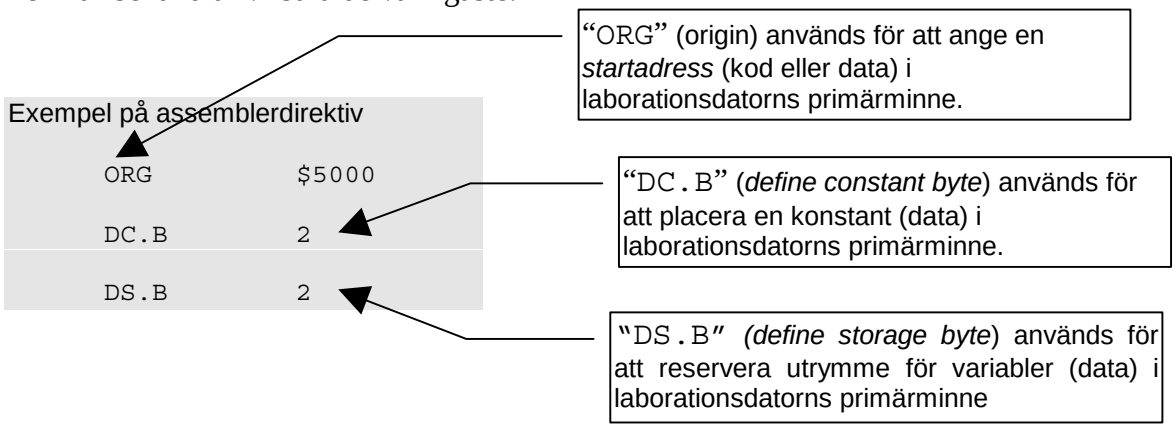
Symboler

Varje symbolnamn måste väljas *unikt* dvs, får bara definieras *en* gång i programmet. Symbolnamnet får vara högst 32 tecken långt. Symbolens *första* tecken måste vara en bokstav (a-z eller A-Z) *eller* en "understrykning". Observera att de svenska tecknen å,ä och ö *inte* får användas i symbolnamn

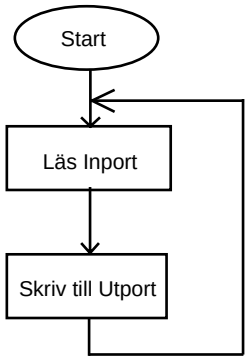
Små respektive stora bokstäver betraktas som *olika* i symbolnamn, alltså kan exempelvis symbolnamnen "`start`" och "`Start`" definieras i samma program. Det är dock olämpligt att göra så eftersom det lätt kan skapa förvirring hos den som läser programmet.

Instruktioner och assemblerdirektiv

Nästa fält i raden kan innehålla en *mnemonic* för någon processorinstruktion *eller* ett assemblerdirektiv. Assemblerdirektiv används för att instruera assemblatorn på olika sätt. Det finns flera olika direktiv men här behandlar vi bara de vanligaste.



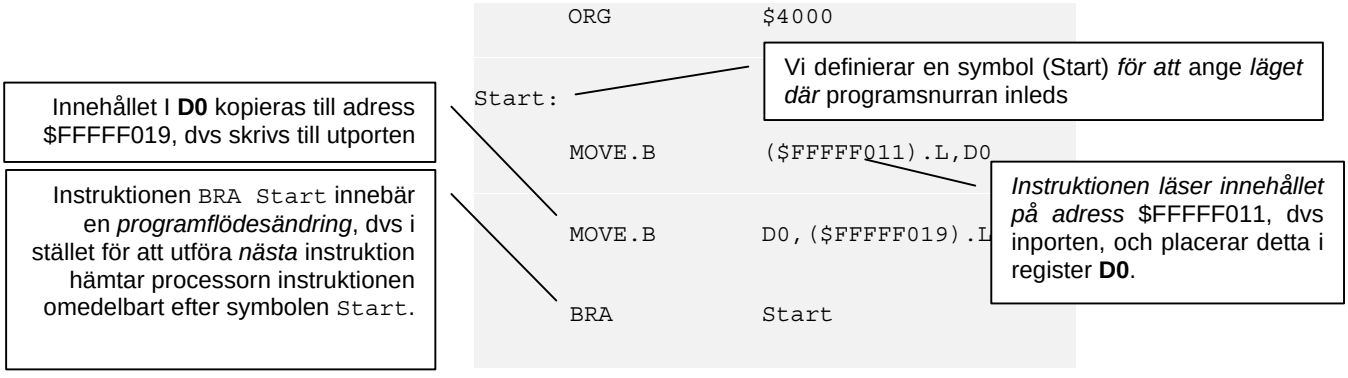
Fältet kan också innehålla en mnemonic, dvs en assemblerinstruktion för laborationsdatorns processor. Vi har tidigare sett exempel på några instruktioner och kommer längre fram i detta kapitel att behandla ytterligare instruktioner i MC68340's instruktionsuppsättning.



Figur 5.4 *Flödesplan för första assemblerprogrammet*

Första assemblerprogrammet

Vårt första assemblerprogram visar exempel på hur vi kan läsa data (8 bitar) från en inport, skriver samma data till en utport och upprepar detta i en "oändlig slinga" (Se flödesplan i marginalen). För exemplet antar vi att inporten är placerad på adress $FFFFF011 och utporten är placerad på adress $FFFFF019 i primärminnet.



I exemplet anges portadresserna numeriskt. Detta är ofta opraktiskt av framför allt två skäl: För det första är risken att skriva *fel* ganska stor, för det andra kan det bli svårt att *ändra* eftersom samma adress säkert förekommer flera gånger. Risken för fel är då överhängande. För att undvika dessa problem kan i stället adresserna definieras som *symboler*. Därefter kan assemblatorn ersätta symbolerna med rätt värden och en

ändring blir enkel att utföra. Assemblerdirektivet `EQU` (*equate*) används för att ange en sådan symbol med ett konstant värde. Vi kan alltså skriva om assemblerprogrammet från vårt första exempel. Vi samlar då ihop alla equ-satser (equ-direktiv) i *början* av programmet så att de blir lättöverskådliga och enkla att hitta:

EQU-direktivet: I stället för att skriva

```
   MOVE.B  ($FFFFF011).L,D0
```

kan du skriva:

```
InPort   EQU     $ FFFFF011
         MOVE.B  (InPort).L,D0
```

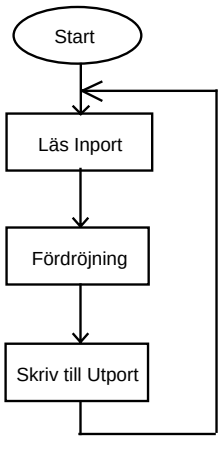
```
CODE            EQU             $4000
InPort          EQU             $FFFFF011
OutPort         EQU             $FFFFF019

                ORG             CODE
Start:
                MOVE.B          (InPort).L,D0
                MOVE.B          D0,(OutPort).L
                BRA             Start
```

Som vi ser kan vi, genom att välja lämpliga symbolnamn, underlätta läsbarheten av programmet. Det blir då enklare att följa programflödet och "se" vad det utför. För att ytterligare öka läsbarheten kan vi införa *kommentarer* i programmet. Det fjärde fältet på raden tolkas av assemblatorn som en kommentar. Vi kan också, genom att ange en *stjärna* (`*`) i radens första position, använda en hel rad för kommentarer.

```
* Programmet läser inport och kopierar till utport

InPort          EQU             $FFFFF011
OutPort         EQU             $FFFFF019

                ORG             $4000
Start:
                MOVE.B          (InPort).L,D0       Läs
                MOVE.B          D0,(OutPort).L      Skriv
                BRA             Start               Börja om
```

Symbolfält, blankt *eller* kommentar	Instruktion *eller* direktiv	Operand(er) till instruktion *eller* argument till direktiv	Kommentar *eller* ingenting

Fälten separeras med `TAB` eller `SPACE`

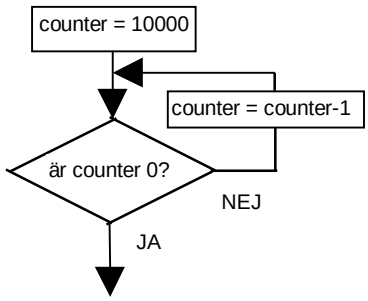


Figur 5.5 *Flödesplan för program med fördröjning*

Låt oss nu skapa en *fördröjning* i vårt program. Fördröjningen placeras mellan inmatning och utmatning. Det enklaste sättet att göra detta är att skapa en *programslinga* som utförs ett stort antal gånger. Vad som utförs i programslingan är oväsentligt, eftersom vi ju bara vill fördröja programmet. Vi utformar programslingan enligt flödesplanen Figur 5.5. Fördröjningen kan enkelt åstadkommas genom att vi skapar en *villkorlig* programslinga (se Figur 5.6 nedan). Genom att minska en räknarvariabel med ett (ett stort antal gånger) "uppehåller" vi processorn en kort stund. I programflödet har vi placerat en *villkorstest*. Villkoret gäller en *räknarvariabel*. Vi har gett räknarvariabeln värdet 10000 från start och för varje *varv* i programslingan minskar vi värdet med 1. Om programslingan utförs tillräckligt många gånger, så har vi också skapat en fördröjning av programmet.

Villkorliga programflöden, eller så kallade *val* görs utgående från en *test* av något villkor. Momentet utförs genom att man kombinerar två instruktioner:

1. Jämförelseinstruktion som påverkar flaggregistret.
2. Villkorlig instruktion (hopp-instruktion) som utför programflödesändring beroende på flaggorna i statusregistret.



Figur 5.6 *Fördröjning kan åstadkommas med en villkorlig programslinga*

För *jämförelser* finns bland annat `CMP` (*compare*) instruktionerna. Instruktionen påverkar endast flaggorna i statusregistret och destinationsoperanden lämnas opåverkad.

```
CMPI.L      #0,D0
```

Operation:
innehåll D0 –0 → SR
dvs: jämför innehållet i D0 med 0

Om innehållet i D0 *är* 0, kommer Z-flaggan att sättas till 1, annars sätts Z-flaggan till 0

Villkorliga instruktioner används, som namnet antyder, för att utföra en eller flera instruktioner då någon förutsättning är uppfylld. En villkorlig instruktion:

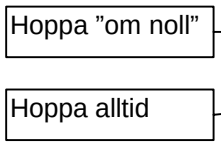
1. Testar villkoret mot innehållet i statusregistret.
2. Om resultatet av testen är SANT, utförs instruktionen (programhoppet sker)
3. Om resultatet är FALSKT fortsätter exekveringen med nästa instruktion (programhoppet sker inte)

14 olika villkor kan anges (vi återkommer till dessa längre fram) här kan vi använda `BEQ` (*branch equal*).

Den villkorliga hoppinstruktionen `BEQ` utförs endast om Z-flaggan är 1 i annat fall fortsätter exekveringen med *nästa* instruktion. Med användande av symboliska adresser får vi alltså för vårt exempel:

```
      CMPI.L      #0,D0
      BEQ         Om_0
Inte_0:
*  denna sekvens utförs om innehållet i D0
*  är skilt  från 0

Om_0:
*  denna sekvens utförs om innehållet i D0
*  är lika med 0
```

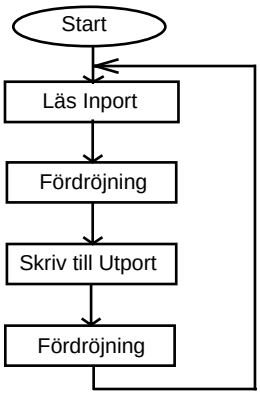
Vi kan nu skriva en sekvens instruktioner som skapar en fördröjningsrutin.




```
* Sekvensen som utför fördröjningen
            MOVE.L      #10000,D1   startvärde
delay:
            CMPI.L      #0,D1             färdig ?
            BEQ         delay_exit        i så fall
            SUBI.L      #1,D1             annars ..
            BRA         delay             fortsätt

delay_exit:
* här fortsätter exekveringen då sekvensen är klar
```

Kombineras fördröjningen med med tidigare program fås följande kod:

```
* Litet program som läser från en inport
* och kopierar värdet till en utport
CODE        EQU         $4000
InPort      EQU         $FFFFF011
OutPort     EQU         $FFFFF019
            ORG         CODE
Start:      MOVE.B      (InPort).L,D0     Läs
            MOVE.L      #10000,D1
delay:      CMPI.L      #0,D1             färdig ?
            BEQ         delay_exit        i så fall
            SUBI.L      #1,D1             annars ..
            BRA         delay             fortsätt

delay_exit:
            MOVE.B      D0,(OutPort).L    Skriv
            BRA         Start             Börja om
```



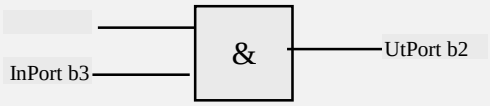
Figur 5.7 *Flödesplan för program med flera fördröjningar*

Ofta har man anledning att organisera programmet i *subrutiner*. Detta har flera fördelar, för det första blir programmet *överskådligare* och för det andra kan subrutinens programkod användas från flera ställen i programmet utan att man behöver upprepa den. En subrutin *anropas* med instruktionen `BSR` (*branch to subroutine*), en subrutin *avslutas* alltid med instruktionen `RTS` (*return from subroutine*). Om vi exempelvis vill modifiera vårt program enligt flödesplanen i Figur 5.7 är det lämpligt att omforma programsekvensen som utför fördröjningen till en subrutin. Vi får då följande utseende på vårt program.

```
* Litet program som läser från en inport
* och kopierar värdet till en utport
CODE        EQU         $4000
InPort      EQU         $FFFFF011
OutPort     EQU         $FFFFF019

            ORG         CODE
Start:      MOVE.B      (InPort).L,D0     Läs
            BSR         Delay             Fördröj
            MOVE.B      D0,(OutPort).L    Skriv
            BSR         Delay             Fördröj
            BRA         Start             Börja om

* Subrutin Delay
Delay:      MOVE.L      #10000,D1   startvärde
delay1:     CMPI.L      #0,D1             färdig ?
            BEQ         delay_exit        i så fall
            SUBI.L      #1,D1             annars ..
            BRA         delay1            fortsätt
delay_exit: RTS
```

Som avslutning på detta kapitel ger vi nu exempel på hur vi kan åstadkomma *booleska funktioner* programmässigt.

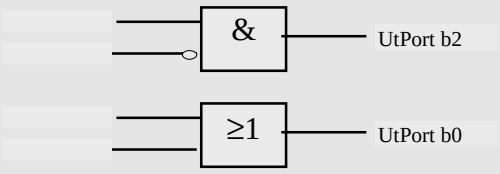
EXEMPEL
Antag en 8-bitars in-port på adress $FFFFF011 och en 8-bitars ut-port på adress $FFFFF019. Skriv ett (ändlöst) program som utför följande logiska villkor:



Lösning:

```
InPort      EQU         $FFFFF011
OutPort     EQU         $FFFFF019
            ORG         $4000
start:
            MOVE.B      (InPort).L,D0
            ANDI.B      #%00001001,D0
            CMPI.B      #%00001001,D0
            BNE         OutNUL
            MOVE.B      #%00000100,(OutPort).L
            BRA         start
OutNUL:     MOVE.B      #0,(OutPort).L
            BRA         start
```

EXEMPEL
Antag en 8-bitars in-port på adress $FFFFF019 och en 8-bitars ut-port på adress $FFFFF011. Skriv ett (ändlöst) program som utför följande logiska villkor:




```
InPort      EQU         $FFFFF011
OutPort     EQU         $FFFFF019
            ORG         $4000
gate_1:
            MOVE.B      (InPort).L,D0
            ANDI.B      #%00001001,D0
            CMPI.B      #%00000001,D0
            BNE         OutNUL1
* Ettställ endast b2
      ORI.B       #%00000100,(OutPort).L
      BRA         gate_2
OutNUL1:
* Nollställ endast b2
      ANDI.B      #%11111011,(OutPort).L
gate_2:
      MOVE.B      (InPort).L,D0
      ANDI.B      #%00010010,D0
      CMPI.B      #0,D0
      BEQ         OutNUL2
* Ettställ endast b0
      ORI.B #%00000001,(OutPort).L
      BRA         gate_1
OutNUL2:
* Nollställ endast b0
      ANDI.B      #%11111110,(OutPort).L
      BRA         gate_1
```

5.4 Övningsuppgifter

Uppgift 25: Ange *innebörden* av följande assemblerdirektiv:

a) `ORG $4000`
b) `DS.B $32`
c) `DC.W $10`

Uppgift 26: Ange minnesadresser och minnesinnehåll som påverkas av följande sekvens assemblerdirektiv:

```
ORG  $5000
DS.B 4
DC.W $4320
DC.L $88776655
```

Uppgift 27: Ange innehållet i register D0 (i hexadecimal form) efter följande instruktionssekvens. Använd 'X' för att ange en okänd hexadecimal siffra.

a) `MOVE.L #$87654321,D0`
b) `MOVE.W #$4321,D0`
b) `MOVE.B #$21,D0`

$5000	$00
$5001	$11
$5002	$22
$5003	$33
$5004	$44
$5005	$55
$5006	$66
$5007	$77
$5008	$88
$5009	$99

Uppgift 28: Antag att en sekvens i minnet har följande innehåll (se tabell i marginalen). Ange innehåll i de register som påverkas, efter det att följande instruktioner utförts:

```
a)  MOVE.B   ($5003).L,D0

b)  MOVE.W   ($5002).L,D0

c)  MOVE.L   ($5006).L,D0

d)  MOVEA.L  #$5006,A0
    MOVE.B   (A0),D0
```

$5000	$00
$5001	$11
$5002	$22
$5003	$33
$5004	$44
$5005	$55
$5006	$66
$5007	$77
$5008	$88
$5009	$99

Uppgift 29: Förutsätt minnesinnehåll enligt tabell i marginalen och ange *nytt* minnesinnehåll för de minnesadresser som påverkas efter respektive instruktion:

```
a)  MOVE.B   #$AA,($5003).L
b)  MOVE.W   #$BBCC,($5002).L
c)  MOVE.L   #$BBCCDDFF,($5002).L
d)  MOVEA.L  #$5000,A0
    MOVE.B   (A0),($5001).L
e)  MOVEA.L  #$5000,A0
    MOVE.W   (A0),($5004).L
f)  MOVEA.L  #$5000,A0
    MOVE.L   (A0),($5004).L
```

Uppgift 30: Skriv *en* assemblerinstruktion som nollställer innehållet från adress $5000 till och med adress $5003 i primärminnet.

Uppgift 31: Skriv *en* assemblerinstruktion som skriver konstanten $21 till en ut-port på adress $FFFFF019. Ut-porten är 8 bitar bred.

Uppgift 32: Skriv en sekvens assemblerinstruktioner som *läser* värdet hos en in-port (adress $FFFFF011), därefter skiftar detta värde ett steg till höger, och sedan skriver det skiftade värdet till en ut-port på adress $FFFFF019. Båda portarna är 8 bitar breda.

Uppgift 33: Skriv ett *program* som kontinuerligt *läser* värdet hos en in-port (adress $FFFFF011), därefter skiftar detta värde ett steg till höger, och slutligen skriver det skiftade värdet till en ut-port på adress $FFFFF019. Båda portarna är 8 bitar breda. Programmet ska placeras med start på adress $4000.

Uppgift 34: Skriv ett *program* som kontinuerligt *läser* värdet hos en In-port (adress $FFFFF011), därefter skiftar detta värde ett steg till vänster, och slutligen skriver det skiftade värdet till en Ut-port på adress $FFFFF019. Båda portarna är 8 bitar breda. Programmet ska placeras med start på adress $4000.

Uppgift 35: Skriv ett *program* som kontinuerligt *läser* värdet hos en in-port (adress $FFFFF011), därefter nollställer den minst signifikanta biten hos detta värde och slutligen skriver värdet till en ut-port på adress $FFFFF019. Båda portarna är 8 bitar breda. Programmet ska placeras med start på adress $4000.

Uppgift 36: Skriv ett *program* som kontinuerligt *läser* värdet hos en In-port (adress $FFFFF011), därefter nollställer den minst signifikanta *och* den mest signifikanta biten hos detta värde och slutligen skriver värdet till en Ut-port på adress $FFFFF019. Båda portarna är 8 bitar breda. Programmet ska placeras med start på adress $4000.

Uppgift 37: Skriv ett *program* som kontinuerligt *läser* värdet hos en In-port (adress $FFFFF011), därefter ettställer den minst signifikanta *och* den mest signifikanta biten hos detta värde och slutligen skriver värdet till en Ut-port på adress $FFFFF019. Båda portarna är 8 bitar breda. Programmet ska placeras med start på adress $4000.

Uppgift 38: Skriv ett *program* som kontinuerligt *läser* värdet hos en In-port (adress $FFFFF011), därefter *inverterar* den minst signifikanta *och* den mest signifikanta biten hos detta värde och slutligen skriver värdet till en Ut-port på adress $FFFFF019. Båda portarna är 8 bitar breda. Programmet ska placeras med start på adress $4000.

Uppgift 39: Skriv en subrutin TestForZero som testar värdet hos en In-port (adress $FFFFF011). Om värdet är 0, ska subrutinen avslutas, annars

forsätter testen. Subrutinen ska kunna användas, exempelvis, på följande sätt.

```
       ORG          $4000
program:
       BSR          TestForZero
       ....         ytterligare instruktioner
       BRA          program
```

Uppgift 40: Skriv en *subrutin* `TestInPort` som testar värdet hos en In-port (adress $FFFFF011). Om den mest signifikanta biten är 0, ska subrutinen avslutas, annars forsätter testen. Subrutinen ska kunna användas, exempelvis, på följande sätt.

```
       ORG          $4000
program:
       BSR          TestInPort
       ....         ytterligare instruktioner
       BRA          program
```

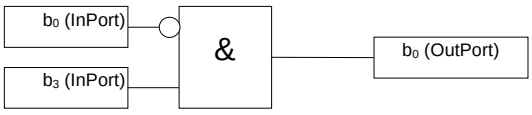
Uppgift 41: Konstruera en *subrutin* `Logical_1` som utför följande logiska villkor:

a) Rita en flödesplan för subrutinen
b) Skriv assemblerkoden.

Subrutinen ska exempelvis kunna anropas enligt:

```
InPort      EQU          $FFFFF011
OutPort     EQU          $FFFFF019
            ORG          $4000
program:
            BSR          Logical_1
            ..... ytterligare instruktioner
            BRA          program
```

Uppgift 42: Konstruera en *subrutin* `Logical_2` som utför följande logiska villkor:

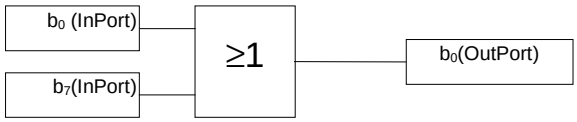


a) Rita en flödesplan för subrutinen
b) Skriv assemblerkoden.

Subrutinen ska exempelvis kunna anropas enligt:

```
InPort      EQU          $FFFFF011
OutPort     EQU          $FFFFF019
            ORG          $4000
program:
            BSR          Logical_2
            ..... ytterligare instruktioner
            BRA          program
```

Uppgift 43: Konstruera en *subrutin* `Logical_3` som utför följande logiska villkor:

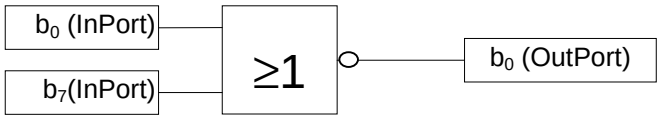


a) Rita en flödesplan för subrutinen
b) Skriv assemblerkoden.

Subrutinen ska exempelvis kunna anropas enligt:

```
InPort      EQU         $FFFFF011
OutPort     EQU         $FFFFF019
            ORG         $4000
program:
            BSR         Logical_3
            ..... ytterligare instruktioner
            BRA         program
```

Uppgift 44: Konstruera en *subrutin* `Logical_4` som utför följande logiska villkor:



a) Rita en flödesplan för subrutinen
b) Skriv assemblerkoden.

Subrutinen ska exempelvis kunna anropas enligt:

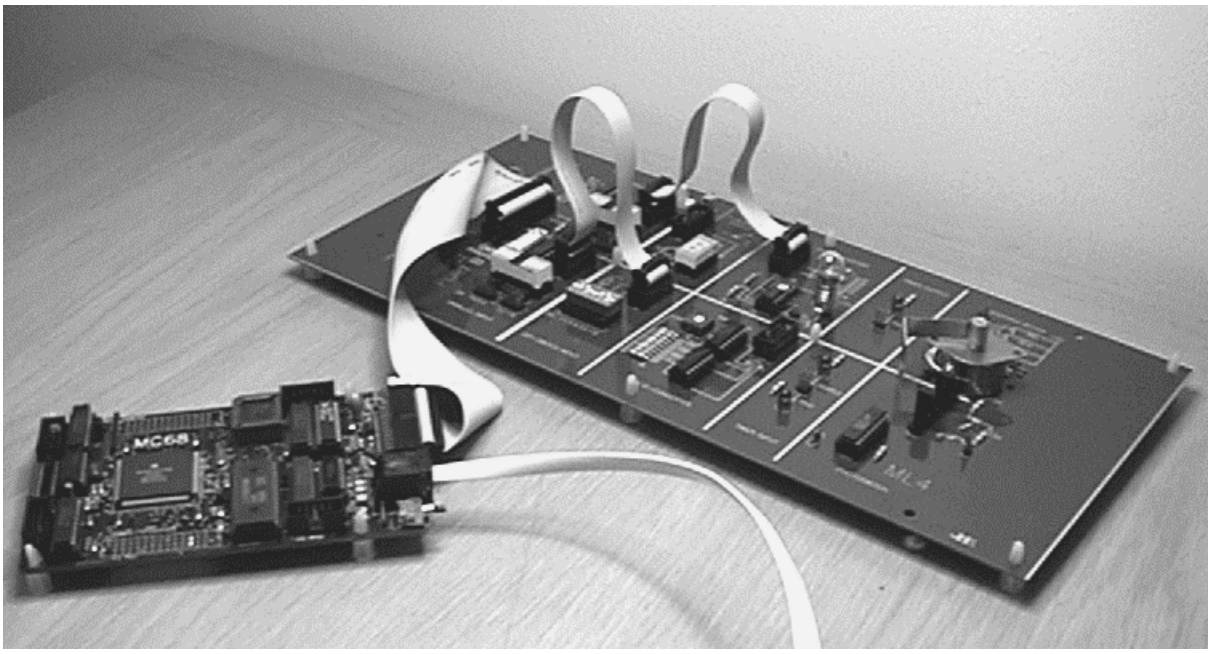
```
InPort      EQU         $FFFFF011
OutPort     EQU         $FFFFF019

            ORG         $4000
program:
            BSR         Logical_4
            ..... ytterligare instruktioner
            BRA         program
```

Uppgift 45: Skriv ett *program* som kontinuerligt utför följande algoritm:

Om b_0 (InPort) = 1
Sätt (b_7) UtPort = 1
annars
Sätt (b_7) UtPort = 0

Programmet ska placeras med start på adress $4000. *InPort* förutsätts ha adress $FFFFF011. *UtPort* förutsätts ha adress $FFFFF019.

6. Mikrodatorns kommunikation med omvärlden



Mikrodator MC68 och tillbehörskortet ML4, med ”yttre enheter”.

En mikrodators ”omvärld” kan vara tämligen mångfacetterad. Vi talar allmänt om ”yttre enheter” och menar då någon godtycklig komponent som kan ge insignaler till, och/eller, styras av, en centralenhet.

Mikrodatorns centralenhet kommunicerar med sådana yttre enheter via speciella så kallade ”periferikretsar”. Det finns en rad olika typer av sådana periferikretsar beroende på vilken yttre enhet de anpassats för. I detta kapitel ska vi beskriva principerna för de vanligaste typerna av kommunikation som kan ske via speciella periferikretsar. Vi kommer att redogöra för

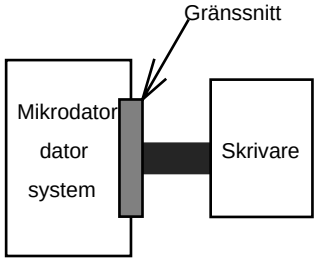
- Parallell kommunikation
- Seriell kommunikation
- Omvandling av analoga signaler till digital form och vice versa

6.1 Parallell kommunikation

Vi skall nu beskriva parallell in- och utmatning. För att göra detta exemplifierar vi med en utport (ett *gränssnitt*) till en vanlig skrivare. Vi kommer att bygga upp porten bit för bit, både hård- och mjukvarumässigt, och studera problem och fördelar med några olika lösningar.

Vi måste ge vissa förutsättningar för skrivaren som ska anslutas. Vår skrivare är från början en "dum" skrivare:

- Den kan endast arbeta med ett tecken i taget. Större skrivare kan hämta in många sidor av ett dokument innan den börjar på utskriften, medan vår hämtar ett tecken i taget för att skriva det innan nästa tecken hämtas.
- Det finns inledningsvis inga handskaknings-signaler från skrivaren som indikerar exempelvis slut på papper (*Paper Out*), klar att ta emot nytt tecken (*Ready*), mm.
- Vi förutsätter att skrivaren klarar av att ta emot och skriva 100 tecken per sekund.



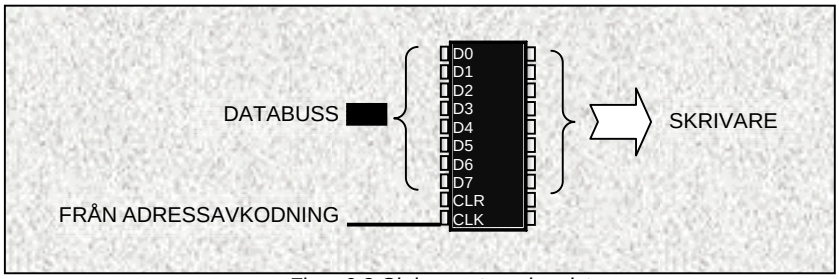
Figur 6.1 *Gränssnitt till skrivare*

Gränssnittet som är placerat i mikrodatorsystemet har som uppgift att anpassa arbetstakten i det snabba mikrodatorsystemet mot den långsamma skrivaren. Gränssnittet fungerar också som en *förstärkare* för den förhållandevis långa kabeln mellan skrivare och mikrodatorsystem.

Varför behöver vi överhuvud taget en utport (ett gränssnitt) för att ansluta skrivaren? Man skulle kunna tänka sig att ansluta skrivarens buss direkt till processorns bussar. En sådan koppling skulle dock knappast fungera om skrivarkabeln blir för lång. Ett mikrodatorsystems bussar är i storleksordningen ett par decimeter vilket möjliggör en hög arbetstakt på bussarna. Om en skrivare med en meterlång kabel ansluts direkt till processorns bussar skulle detta medföra att processorns arbetstakt på bussarna måste dämpas avsevärt. Det skulle också bli betydligt mer störningskänsligt med så långa bussledningar om man inte kompenserade genom att använda kraftiga drivkretsar för signalerna hos varje enskild krets.

6.1.1 Skrivarport med register.

Den enklaste formen av en utport består av ett register anslutet till processorns bussar. Den nödvändiga hårdvarukopplingen för en utport mot en sådan skrivare visas i Figur 6.2 nedan. Det enda som behövs är ett 8-bitars register för att mellanlagra tecknet som processorn skriver ut. Detta finns sedan tillgängligt på skrivarbussen så att skrivaren kan skriva ut tecknet.



Figur 6.2 *Skrivarport med register*

Innan vi studerar ett tänkbart program måste vi ge vissa förutsättningar för hur texten som skall skrivas ut lagras i minnet. Förutsätt att någon programrutin har placerat ASCII-koderna för texten i minnet och att den är lagrad i sekvens från adressen *Text* och framåt. Exemplet visar texten "*Hej Du Kalle!"* som skall skrivas ut. Observera att textsträngen är avslutad med en nolla. Denna markering används av programrutinen för undersöka när det sista skrivbara tecknet i strängen skickats till skrivaren

* Ett tänkbart program som skriver ut textsträngen är:

```
PRINTER     EQU         $8000       Adress till skrivaren

Text1:      DC          ”Hej Du Kalle!”
            DC.B        0
            ALIGN

Print:
            MOVEA.L     #Text,A1          Pekare till text
loop        MOVE.B      (A1),(PRINTER).L  Skriv ett tecken
            CMPI.B      #0,(A1)
            BEQ         EndPrint
            ADDA.L      #1,A1
            BRA         loop
EndPrint:   ...
```

Programmet är enkelt. Register **A1** används för att peka ut "nästa" tecken som skall skrivas. Varje varv i snurran skriver ut ett ASCII tecken (en *byte*) i taget. Funderar vi lite på hur snabbt (hur ofta) vi skriver ett tecken till skrivaren kommer vi fram till att vi skriver ett tecken i storleksordning var tionde mikrosekund. Detta innebär att vi med vårt program skriver ut ungefär $100 \cdot 10^3$ tecken per sekund till vår enkla skrivare som har en utskriftstakt på 100 tecken per sekund. (vilket motsvarar ett tecken per 10 ms.). Man skulle kunna tänka sig att lägga in en fördröjning i utskriftsrutinen enligt:

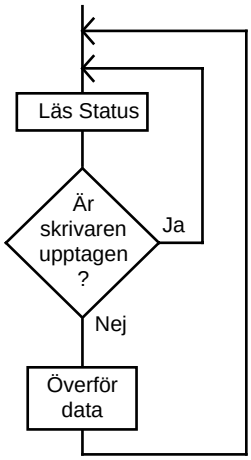
```
            ...
            ADDA.L      #1,A1
            JSR         delay10ms         Vänta i 10 ms.
            BRA         loop
```

Koden ovan skulle kunna fungera då den skriver ut ett tecken var 10:e ms. vilket är skrivarens arbetstakt. Detta är då under förutsättning att skrivaren är klar att ta emot ett tecken precis när det första skrivs ut. Startögonblicket måste *synkroniseras*. Om vi testar denna utskriftsrutin på

en lång utskrift så kommer det snart att bli fel eftersom det är mycket svårt för att inte säga omöjligt, att få två olika klockor (en i skrivaren och en i datorsystemet) att gå *synkront* (gå lika fort). Felet kommer att uppträda som missade tecken eller dubletter. Utskriften från vårt exempel skulle på papperet kunna se ut som: *Hej Du alle!* eller *Hej Duu Kalle!*.

Vi har hittills använt oss av ovillkorlig överföring som kortfattat kan beskrivas som att datorsystemet (*sändaren*) skickar data till skrivaren (*mottagaren*) utan att överhuvud taget ta hänsyn till om mottagaren är redo att ta emot ett (eller *nästa*) tecken. Vi inser att detta är otillräckligt för en säker kommunikation. Vi behöver någon form av villkorlig överföring.

Som vi nu ser så handlar det hela om synkroniseringsproblem. Vi har två olika system: skrivaren och datorsystemet som har helt olika arbetstakter. För att få dessa att arbeta på önskvärt sätt tillsammans måste dessa på något sätt synkroniseras. Detta kan göras helt i hårdvara eller en kombination av hård- och mjukvara.



Figur 6.3 *Busy Waiting*

6.1.2 Skrivarport med register och READY-signal.

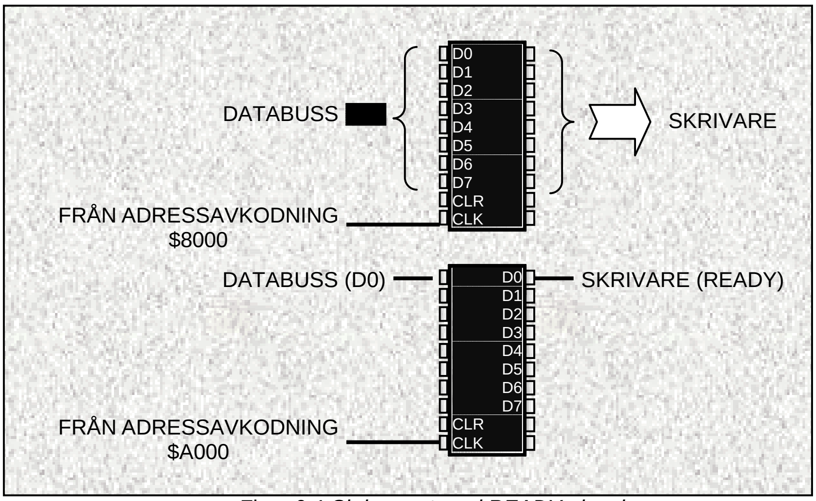
Vi förutsätter nu att skrivaren kan generera en speciell utsignal (*READY*) som indikerar om skrivaren är redo att ta emot ett nytt tecken eller inte. Låt signalen definieras enligt:

- READY = 1 (hög nivå) indikerar att skrivaren är klar att ta emot ett nytt tecken.
- READY-signalen kan läsas via ett register (Status Register) av processorn.

Processorn läser oupphörligt statusregistret och undersöker sedan *READY*-biten (signalen) från skrivaren tills denna indikerar att ett nytt tecken kan skickas till skrivaren. En sådan programkonstruktion kallas *Upprepad Statustest* eller *Busy Waiting* (se Figur 6.3).

Studera Figur 6.4 nedan som visar skrivarporten med busy-signalering. Ett programförslag som är anpassat till printerporten med busy-signal följer. Observera att efter vi skrivit ut ett tecken till skrivaren måste programmet invänta att *READY* går låg för att inte skriva ut ett nytt tecken direkt. Detta medför att programmet består av två inre snurror, en som testar att *READY* går hög och en som testar att *READY*-signalen går låg.

```
PRINTER     EQU    $8000              Adress till Data Reg
STATUS      EQU    $A000              Adress till Status Reg

            MOVEA.L      #Text,A1           Pekare till text

busy        BTST.B       #0,(STATUS).L      Testa READY bit
            BEQ          busy               Hoppa om noll

            MOVE.B       (A1),(PRINTER).L   Skriv ett tecken
            ADDA.L       #1,A1

wait        BTST.B       #0,(STATUS).L      Testa READY bit
            BNE          wait               Hoppa om ett
            CMPI.B       #0,(A1)
            BNE          busy
            ....
```



Figur 6.4 *Skrivarport med READY-signal*

Nu har vi synkroniserat datorsystemet till skrivarens arbetstakt och på så sätt kommer inte skrivaren att varken skriva ut dubletter eller att missa tecken. Lösningen är tyvärr fortfarande felaktig. Tänk dig vad som händer när datorn skriver ut det sista tecknet (utropstecknet) om vi förutsätter att vår enkla skrivare inte kan stoppas med något kommando.

Det sista tecknet som skrivs till printer portens dataregister ligger kvar och kommer att läsas ett antal gånger av skrivaren. Varje gång skrivaren signalerar *READY* kommer den att läsa utropstecknet som ligger kvar i skrivarportens dataregister. I vårt exempel kommer utskriften därför utropstecknet att fylla hela papperet (Se figur i marginalen). Det krävs därför en signal (handskakningssignal) från datorsystemet till skrivaren som indikerar att det *finns* ett tecken att skrivas. Följakligen måste vårt gränssnitt mot skrivaren också förses med någon form av *styrregister* genom vilket signaler från datorsystemet kan nå skrivaren.

Hej Du Kalle !!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!

Ett generellt parallell-gränssnitt består därför ofta av:

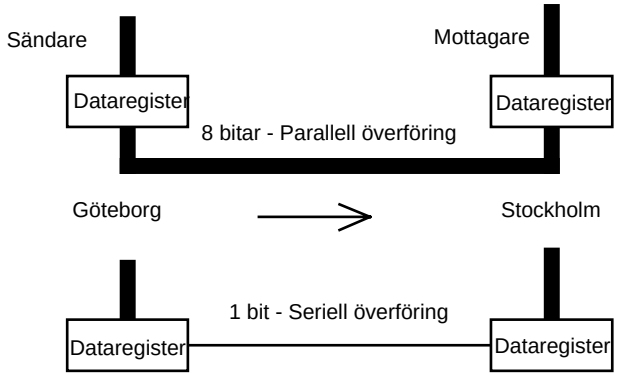
- Styrregister
- Dataregister
- Statusregister

Ofta är riktningen hos dataregistret (från datorsystemet eller mot datorsystemet) programmerbar via styrregistret och man kan i sådana fall utelämna vissa styr- status- register mot omgivningen.

6.2 Seriell kommunikation

Vid seriell I/O överförs databitarna efter varandra på en och samma signalledning. Detta minskar antalet signalförande ledningar mot exempelvis den parallella skrivarporten där minst åtta parallella signalledningar används mellan datorsystem och skrivare. Den principiella skillnaden är enkel, i stället för att använda flera koppartrådar klarar vi oss med en, men till priset av en minst åtta gånger lägre överföringshastighet. Vi måste också införa speciella regler för i vilken ordning bitarna överförs

(och vilken betydelse dessa har) på vår enda signalledning. En uppsättning sådana regler kallas *protokoll*.



Figur 6.5 *Seriell och parallell överföring*

Vi kommer i de följande avsnitten att beskriva grundläggande begrepp som *synkron* och *asynkron* överföring, *överförings-kapacitet*, *protokoll*, som är utmärkande för seriell I/O.

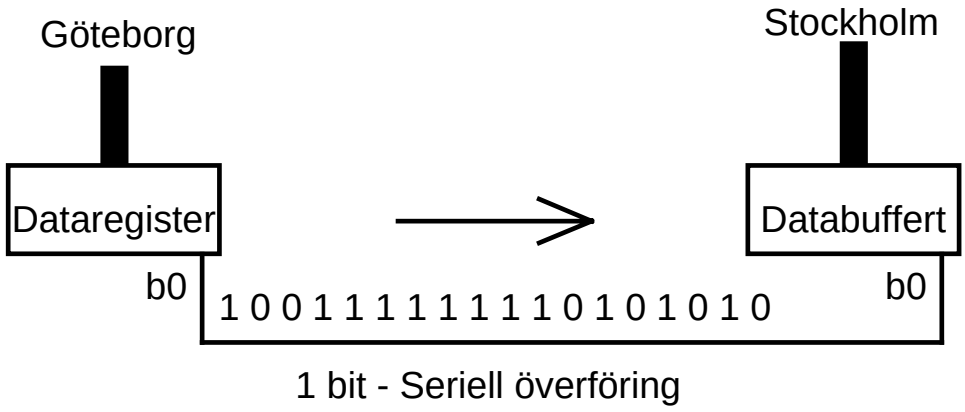
Överhuvudtaget, för kommunikation mellan platser belägna på lite större avstånd från varandra och där prestandakraven, dvs svarstiderna, är mindre kritiska väljs ofta seriell överföring av data. Låt oss inledningsvis diskutera termen *protokoll*. Med protokoll menas *regler* för hur en överföring skall gå till, hur många databitar som skall skickas, hur snabbt databitarna skickas, vilka spänningsnivåer som används och hur handskakningsförloppet skall gå till. Vid "lite större avstånd" mellan två system används alltså oftast seriell överföring eftersom det kostnadsmässigt är mycket billigare. Med *lite större avstånd* menas här från några meter och uppåt.

Tänk dig nu att vi önskar sammankoppla ett datorsystem i Stockholm med ett i Göteborg. En 8-bitars parallell anslutning borde här bli 8 gånger så dyr som en serieanslutning eftersom det används 8 signalledningar i stället för en ledning. Dessutom har vi sett att det oftast med parallell överföring krävs någon form av handskaknings-signaler för att klara av synkroniseringen (jämför med resonemang i föregående avsnitt). Detta innebär att när sändaren i Göteborg har skickat ett tecken så inväntar den någon *READY*-signal innan den skickar nästa tecken. Visserligen fortplantas elektriska signaler (nästan) med ljusets hastighet i kablage, men då avstånden blir stora tar det ur en dators synvinkel faktiskt *mycket* lång tid. Överföringstiden för en enstaka elektrisk signal blir:

$$t = \frac{l}{c} = \frac{50 \cdot 10^4}{3 \cdot 10^8} = 1{,}66 \qquad ms$$

Det tar lika lång tid för *READY* signalen att överföras tillbaks till Göteborg och totalt blir detta alltså 3,3 ms. Med dessa överföringshastigheter överförs då ca 300 byte per sekund, vilket måste betraktas som mycket dåliga prestanda. (Egentligen tar det ännu längre tid eftersom utbredningshastigheten i kablage är lägre än ljushastigheten (c) och att det dessutom tillkommer fördröjningar i förstärkare och dylikt).

Man väljer i stället seriell överföring där man skickar bitarna efter varandra på *en* signalledning *utan* att invänta svar på att varje enskild databit har nått mottagaren. Däremot överför man *ett antal* databitar (ett block, eng *frame*) innan mottagaren sänder någon form av *READY* signal, vilken indikerar att ytterligare data kan skickas.



Figur 6.6 *Seriell överföring med dataregister*

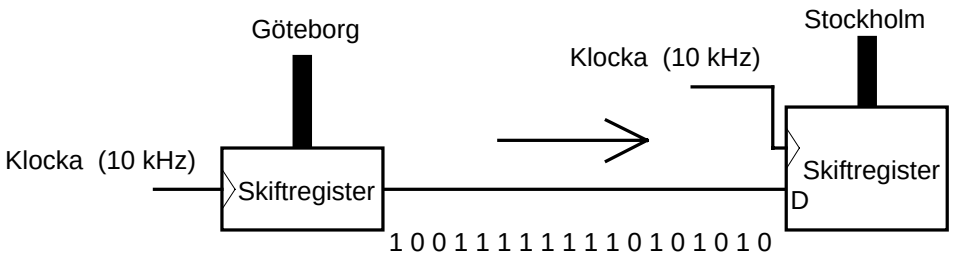
Figur 6.6 visar den enklaste formen av seriell överföring. Detta är samma hårdvaruprincip som vid parallell I/O då konstruktionen har register och buffert (se tidigare i detta kapitel). Däremot används bara *en* signalledning för att sammankoppla enheterna. Ett data register är anslutet till sändaren och en data buffert är anslutet till mottagaren.

Eftersom endast en ledning används måste de olika bitarna i dataregistret skiftas ut (en och en) på signalledningen. Nedan visas ett programexempel som överför data på detta sätt. Vi förutsätter här för enkelhetens skull att den seriella överföringstakten är bestämd till 10.000 bitar per sekund (10 kbits; bits = bitar per sekund).

```
*  Programexempel för seriell överföring med dataregister
       MOVE.B (tecken).L,D0       Läs tecken
       MOVE.B #8,D1               Varvräknare
loop:
       MOVE.B D0,(TxData).L       Sänd bit från "position b0"
       BSR    delay0.1ms          Vänta 0,1 ms
       LSR.B  #1,D0               Skifta fram nästa bit till
*                                        "position b0"
       SUBI.B #1,D1               Minska varvräknare
       BNE    loop                Fortsätt tills det är klart
```

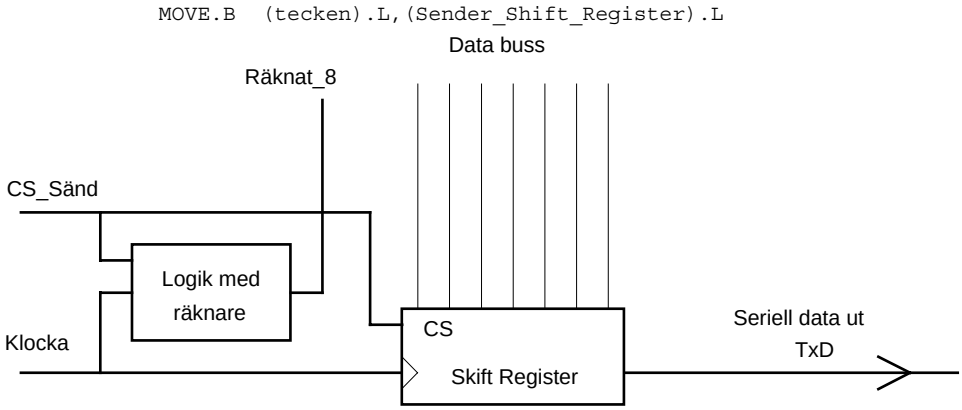
Nackdelen med denna form av konstruktion är att sändaren *programmässigt* måste skicka varje enskild bit. Ett enklare förfarande är att utnyttja *skiftregister* istället för data register och buffert. Vidare ersätts subrutinen i programexemplet ovan med en klocka som levererar en frekvens på 10 kHZ. (se Figur 6.7). Med denna koppling kan sändaren skriva en byte åt gången med 0,8 ms mellanrum (tiden det tar att skifta ut en byte).

$$t = \frac{1}{f} \cdot antal\,bitar = \frac{1}{10kHz} \cdot 8 = 0.8ms$$



Figur 6.7 *Seriell överföring med skiftregister*

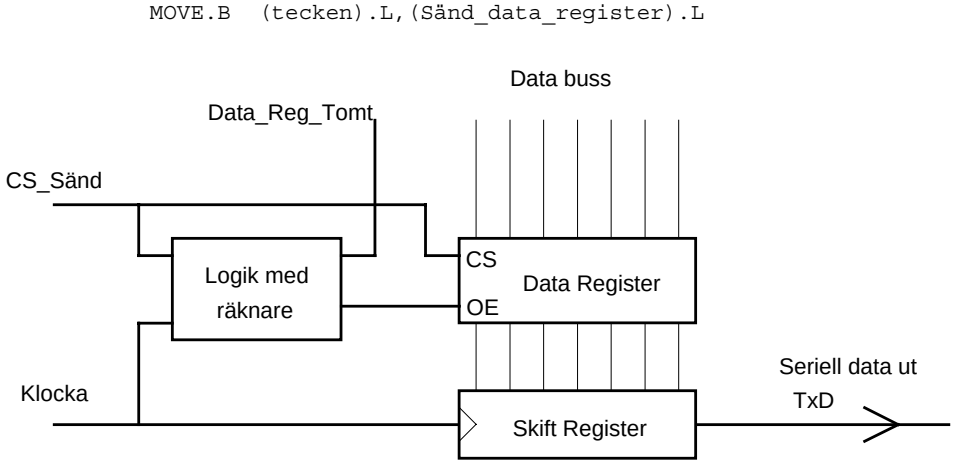
För att sändaren programmässigt inte skall behöva räkna perioder på 0,8 ms för att veta när nästa byte skall skickas kan sändardelen utökas med en räknare och ett logikblock i anslutning till skiftregistret, se Figur 6.8.



Figur 6.8 *Sändarens skiftregister med räknare och logik*

Här startas räknaren när data skrivs (CS_sänd) i skiftregistret. Logikblocket levererar en status signal (Räknat_8) ut som indikerar att alla 8 bitarna är utskiftade. Denna signal kan processorn exempelvis testa i sin sändningsrutin.

Om vi skall uppnå en jämn ström av bitar så måste skiftregistret initieras med en ny databyte i precis rätt ögonblick, dvs då alla tidigare bitar är utskiftade. Detta innebär att processorn direkt måste upptäcka att Räknat_8 signalen har aktiverats. För att undgå detta problem så används en teknik som kallas dubbelbuffring. Det går ut på att koppla in ett dataregister i anslutning till skiftregistret. Se Figur 6.9

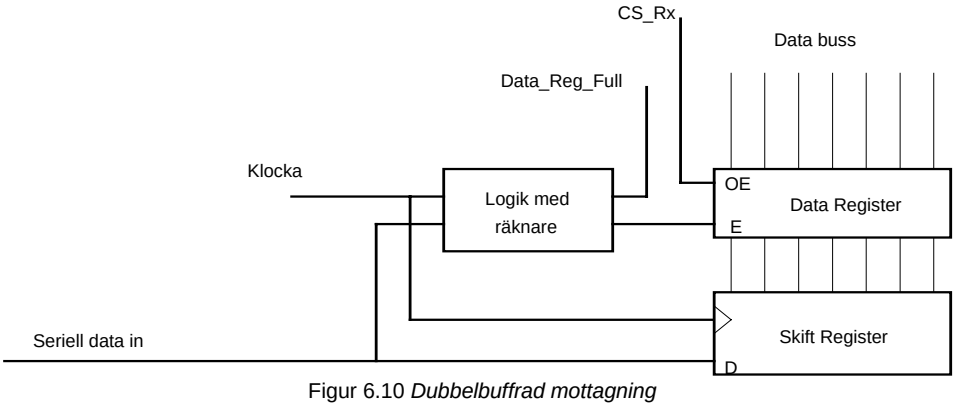


Figur 6.9 *Dubbelbuffrad sändning*

Räknaren startas på liknande sätt (CS) och när den har räknat till 8 genereras OE till det förut initierade dataregistret. På så sätt får skiftregistret ett nytt värde direkt. Vidare så genererar logikblocket signalen Data_Register_Tomt mot processorn som därefter kan överföra en ny byte till dataregistret då skiftregistret för närvarande innehåller en byte som skiftas ut.

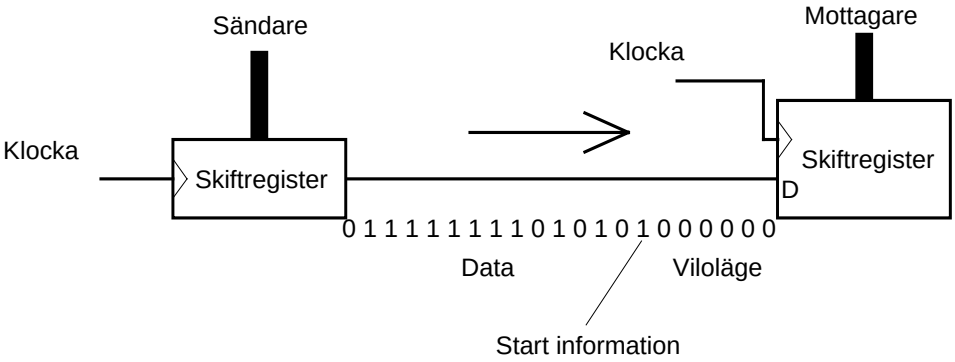
Liknande dubbelbuffring hittas på mottagarsidan. I anslutning till motagarens skiftregister finnns ett data register (se figur Figur 6.10). När skiftregistret har klockat in 8 bitar överförs skiftregistrets innehåll automatiskt till dataregistret med hjälp av logikblocket. Samtidigt sätts någon Data_Register_Full signal som mottagaren kan polla för att undersöka om data har anlänt.

```
MOVE.B  (Rx_data_register).L,(tecken).L
```



Figur 6.10 *Dubbelbuffrad mottagning*

Vi skall nu i detalj studera vad som händer på mottagarsidan. Se Figur 6.11 som visar en ström av bitar på väg mot mottagaren. En fråga här är nu, hur kan mottagaren veta att "den första" biten är på väg? Vi måste definera någon form av startinformation som indikerar att det nu kommer data. Detta kan överföras från sändaren på en separat ledning eller som ett förutbestämt mönster på den seriella data ledningen.



Figur 6.11 *Seriella databitar på väg mot mottagaren*

Om vi skall överföra startinformationen på data ledningen så måste vi bestämma spänningsnivån för kabeln i viloläge (ingen överföring) till exempelvis 0V. Vidare måste det finnas någon logik hos mottagaren som känner av att nivån på kabeln har ändrats till +5V. På så sätt kan vi detektera en **startbit** och vet då att de följande bitar som kommer är databitar.

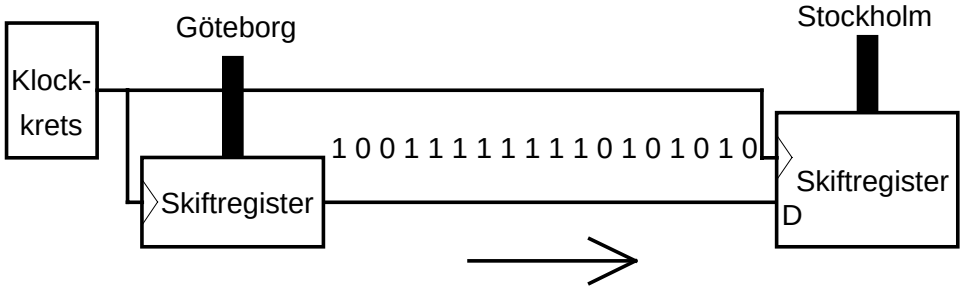
Med samma princip som i vårt första enkla programförslag för seriell I/O skulle vi ju kunna känna av att nivån på signalledningen ändras från 0V till +5V och vice versa under de första bitarna som anländer till mottagaren då dessa består växelvis av nollor och ettor. Men vad händer med de följande bitarna som består av åtta ettor?

Vi förutsätter att en etta motsvarar +5V och en nolla motsvarar 0V. Detta innebär att mottagaren "ser" en +5V-nivå under en lång tid. Skall mottagaren skifta in 7, 8 eller 9 ettor under denna tiden? Vi diskuterar alltså ett synkroniseringsproblem.

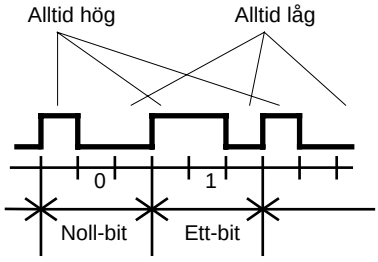
Vi måste på ett eller annat sätt se till att bitarna skiftas in hos mottagaren i samma takt som de skiftades ut av sändaren och vi måste se till att mottagaren börjar med den "första databiten". Man talar om två olika typer av synkronisering vid serieöverföring, nämligen *synkron* och *asynkron* överföring. Vi skall nu studera skillnaden.

6.2.1 Synkron överföring

Kortfattat kan man säga att vid synkron överföring tillhandahåller sändaren en klock-signal som skickas till mottagaren på en separat förbindelse eller tillsammans med databitarna, se Figur 6.12. Denna klocksignal används av mottagaren för att skifta in databitarna i skiftregister .



Figur 6.12 *Synkron överföring med separat klocksignal*



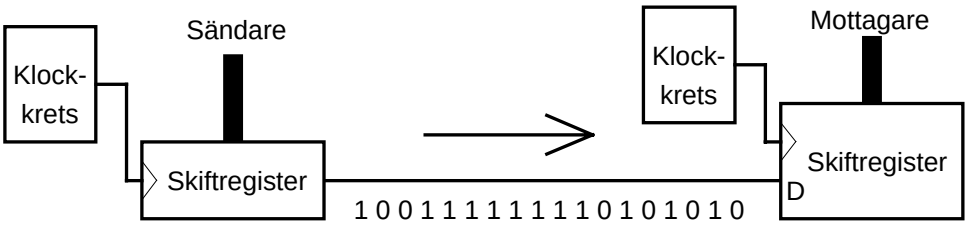
Figur 6.13 *Manchesterkodning*

När data och klocksignal överförs på samma signalledning säger man att databitarna är kodade (med klockinformation). Manchester-kodning är ett exempel på detta, (Figur 6.13) en överförd databit består av tre fält, där det första fältet alltid har en hög nivå och det sista låg. Det mittersta fältet indikerar om det är en noll-bit eller en ett-bit. På så sätt kommer alltid en överförd databit att bestå av en positiv och en negativ puls. En mottagare synkroniserar sig till den positiva flanken som *alltid* uppträder emellan två databitar.

En sådan kodning innebär att den effektiva dataöverföringen sjunker. Vad vi visat här medför att vi utnyttjar 33% av överföringskapaciteten eftersom vi skickar tre bitar för varje databit. Det finns andra kodningsprinciper som är mer effektiva, men vi går inte vidare in på dessa här.

6.2.2 Asynkron överföring

Vid asynkron överföring är sändare och mottagare i förväg överens om en (ungefärligt) bestämd överföringstakt. Denna kan bestämmas redan vid (hårdvaru-) konstruktion av systemet eller med initieringsprogram. Sändaren skickar i sin egen takt till mottagaren som så gott det går ställer in sig efter sändarens takt. Se Figur 6.14. Problem kan uppstå här om *många* bitar skickas efter varandra (se föregående sidor) utan någon form av mellanliggande (regelbunden) synkronisering. Detta beror i så fall på att sändar-klockan och mottagar-klockan inte är tillräckligt väl synkroniserade.



Figur 6.14 *Asynkron överföring*

Beroende på överföringstakten och meddelandets längd (datablockets storlek) används från en till ett hundratal bitar i början av överföringen (startinformation) som enbart har som funktion att synkronisera mottagarklockan till sändareklockan. När datablocket består av ett fåtal bitar, exempelvis ett tiotal, är det tillräckligt med en startbit som synkroniserar mottagar-klockan till sändarklockan.

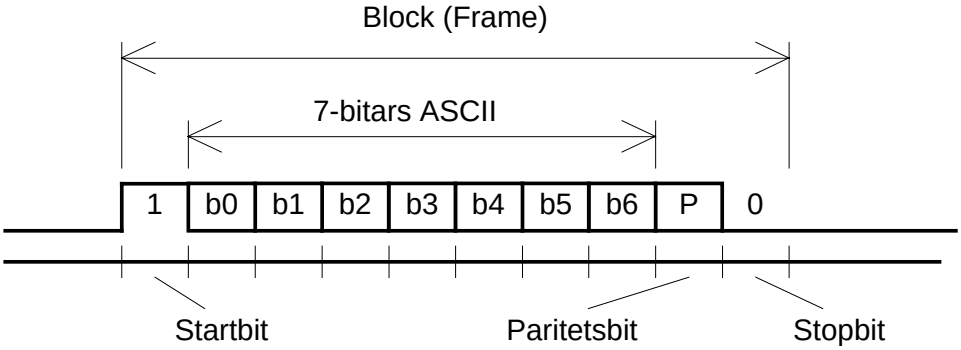
Ett av de vanligaste sätten att komunicera mellan terminal och dator använder exempelvis 10 bitar och en startbit (protokoll). När ett block om 10 bitar skickats, sänds en ny startbit (synkroniseringsbit) tillsammans med nästa block och så vidare. Synkroniseringen (och dataformatet) kan hanteras av en seriell kommunikationskrets som kallas *USART* (Universial Serial Asynchronous Receiver Transmitter). Protokollet är standardiserat.

En typisk USART (eller liknande kretsar som *UART*, *DUART*, mm) innehåller dubbelbuffrade enheter både för sändning och mottagning som vi visat tidigare i detta kapitel. Vidare innehåller kretsarna både styr- och statusregister där användaren kan undersöka om mottagar-dataregistret är fullt, om sändar-dataregistret är tomt mm.

Protokollet *RS232* är ett exempel på ett protokoll som ofta används mellan terminal och dator. Vad som ingår i detta protokoll varierar lite beroende på vilka personer man diskuterar med, men i dagligt tal menar man att man utnyttjar någon form av USART och överför ett ASCII tecken per block. Man använder ett standardiserat kontaktdon (9 eller 25-polig D-SUB Spänningsnivåerna på serieledningen är förutbestämda att variera mellan ± 12V. COM-portarna på en IBM/PC använder exempelvis detta protokoll.

Beroende på version av protokollet RS232 så kan formatet se lite olika ut, exempelvis kan 7 eller 8 databitar överföras i varje block. Figur 6.15 visar en logisk bild av protokollet som har 7 databitar, en startbit, en paritetsbit

och en stoppbit. Vi skall nu speciellt studera hur en mottagare kan ta emot (klocka in) ett sådant format.

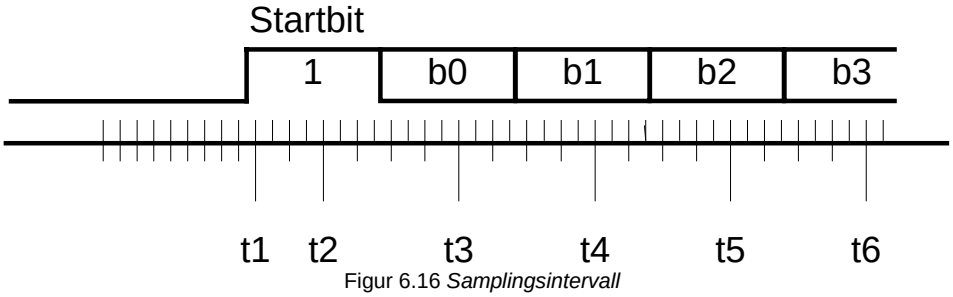


Figur 6.15 *Serieformat mellan terminal och dator*

Syftet med startbiten är att mottagaren skall starta sin mottagarklocka och klocka in databitar seriellt. Syftet med paritetsbiten är att mottagaren skall kunna undersöka om överföringen är korrekt eller inte. Stoppbitens funktion är att signalera att överföringen är slutförd. Både stop och startbiten har definerade värden som mottagaren kontrollerar. Skulle dessa bitar ej ha de förväntade värden tolkas överföringen som felaktig av mottagaren.

Databitarna beror av överförd data och kan således anta olika värden. Paritetsbiten kan defineras som *udda* eller *jämn*. Med detta menas att summan av antal logiska ettor i överföringen skall anta ett udda eller ett jämt värde. Paritetsbiten *sätts* därför *av sändaren* (till noll eller ett) för att ange (den förutbestämda) pariteten. Mottagaren som undersöker pariteten kan på så sätt upptäcka att en bit har ändrat värde under överföringen. Om två bitar ändras under överföringen så upptäcks *inte* felet av mottagaren.

Vi ska nu studera de synkroniseringsproblem som kan uppstå när mottagaren läser (klockar in) data vid en överföring. Förutsätt nu att varje bit skickas med 1 ms mellanrum, vilket motsvarar 1 kHz. När ingen överrföring sker har signalledningen en låg nivå. Mottagaren samplar (undersöker) hela tiden signalledningen med en betydligt högre frekvens, säg 8 gånger högre (8 kHz vilket motsvarar en sampling varje 125 µs) för att upptäcka om en startbit kommer. Detta motsvarar de korta tidsperioderna i Figur 6.16.



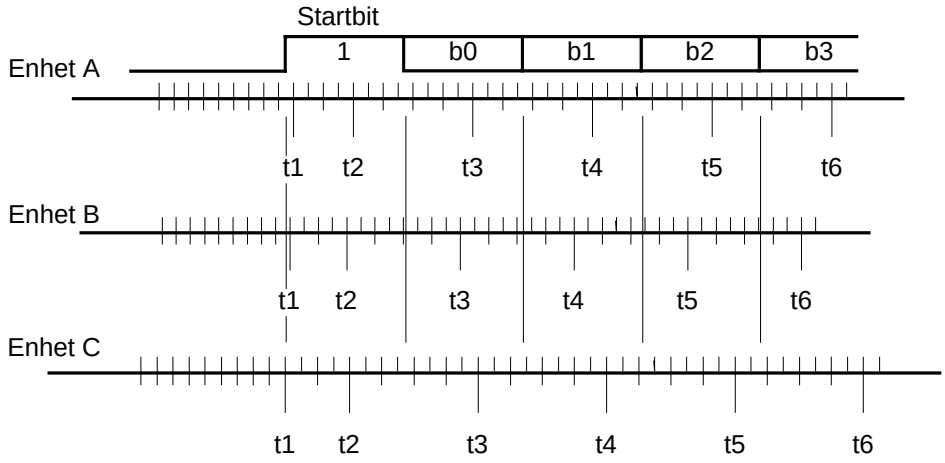
Figur 6.16 *Samplingsintervall*

Sändaren skickar sin start bit följt av databitarna osv enligt sin klocka på 1 kHz. När en hög nivå detekteras av mottagaren (vid t1 i Figur 6.16) tolkas detta som början på en startbit och en räknare i mottagaren initieras nu att

räkna 4 st 125µs pulser. På så sätt har vi kommit ungefär till mitten av startbiten.

Mottagaren avläser signalledningen ännu en gång (vid t2), och om denna fortfarande är hög så tolkas detta som en korrekt startbit för överföringen. Nu initieras räknaren i mottagaren att räkna till 8 med samma samplingsfrekvens (125 µs) innan signalledningen på nytt samplas. På så sätt kommer vi att avläsa signalledningen ungefär mitt i första databiten (vid t3). Räknaren initieras på nytt till 8 för att senare läsa databit ett (vid t4) och så vidare. Vi kommer alltså att "mäta" nivån på signalledningen ungefär mitt varje tidslucka som varje enskild databit upptar.

Låt oss uppskatta den onogrannhet som kan tillåtas i klockfrekvensen mellan mottagare och sändare för detta protokoll. Se Figur 6.17, figuren visar tre enheter, A, B och C där A först sänder data till B och sedan till C. A's klocka blir därför referensklocka i vårt resonemang. B's klocka går för fort och C's klocka för för sakta.



Figur 6.17 *Synkroniseringsproblem mellan enheter*

Enhet A kan troligen kommunicera felfritt både med B och C om skillnaderna mellan klockorna är *under* 5%. A skickar ju 10 bitar varje gång vilket medför en skillnad mindre än ett halvt bitintervall för den sista biten (mottgaren samplar ju mitt i bitintervallet).

Om vi däremot ansluter B och C för att överföra data mellan dessa kommer kommunikation troligen inte att fungera då felmarginalen här verkar vara över 5% vilket motsvarar mer än ett halvt bitintervall för stopinformationen. Kraven på onogrannhet i klockorna bör således vara lägre än, säg, ±2%.

Vid seriekommunikation måste man ange överföringstakten mellan sändare och mottagare. Vid lägre hastigheter används termen *BAUD RATE* och vid högre hastigheter anges hastigheten i kbits (10^3 bitar per sekund) eller Mbits (10^6 bitar per sekund).

En vanlig överföringshastighet mellan en terminal och ett datorsystem är 9600 BAUD. Detta innebär alltså att man kan överföra ca 900 ASCII

tecken per sekund. Generellt kan man säga att ju kortare avstånd och ju bättre (störokänsligare) kabel desto högre hastighet kan man använda.

Speciella periferikretsar (UART, USART, DUART, ACIA, med flera) som innehåller skiftregister med bubbelbuffring, klockor för både sändning och mottagning, paritetskontroll, mm har funnits på marknaden sedan länge. Dessa kan anslutas direkt till processorns buss-system och programmeras för en viss BAUD RATE, för udda eller jämn paritet mm.

6.3 A/D-D/A Omvandling

Redan i kapitel 1 såg vi exempel på två principiellt olika typer av styrobjekt. Den första typen liknade vi vid en brytare som kunde ställas av eller på. Den andra varianten kallade vi helt enkelt *variabel reglering*. Ett styrobjekt med variabel reglering styrs av en *kontinuerlig signal*. Styrobjektets utslag är då *proportionellt* mot denna signal.

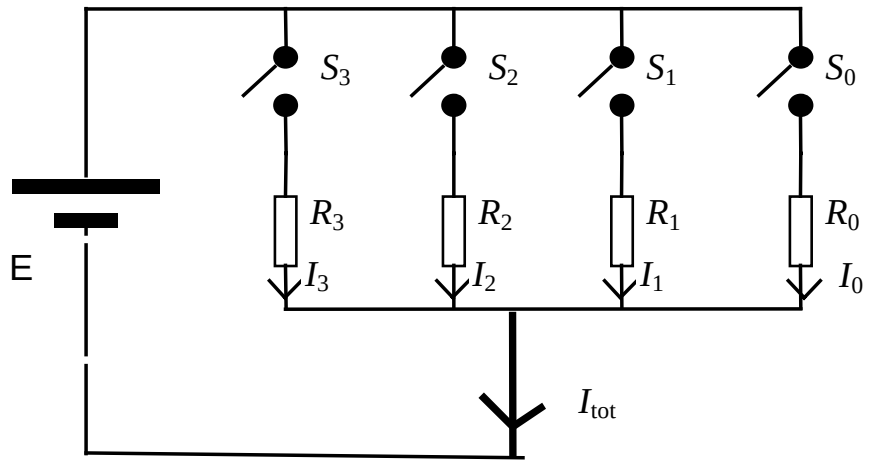
EXEMPEL:
En likströmsmotor kan styras till olika varvtal med hjälp av en variabel spänning. Ju högre pålagd spänning, desto högre varvtal.

Eftersom ett datorsystem arbetar med *digitala* signaler kan dessa inte direkt anslutas till ett styrobjekt med variabel reglering. Vi behöver någon form av *omvandling* av den digitala signalen till en *kontinuerlig* signal som är lämpad för styrobjektet. Den kontinuerliga signalen ska på något sätt motsvara den digitala signalen, dvs den kontinuerliga signalen ska vara *analog* (likvärdig) med det digitala värdet.

I de följande avsnitten ska vi studera hur ett *digitalt* värde, bestämt av en dator, kan omvandlas till en *analog* kontinuerlig spänning (eller eventuellt en ström) som direkt kan styra någon form av givare (styrobjekt). Vi ska också se hur hur *analoga* mätvärden, från *givare* kan omvandlas till *digitala* värden vilka kan bearbetas av datorn.

6.3.1 D/A-Omvandling

En omvandling av en digital signal till en analog utspänning kallas *D/A-Omvandling* och utförs med hjälp av speciella kretsar, så kallade DA-Omvandlare (eng, *DAC, Digital Analog Converter*). De flesta D/A-omvandlare arbetar efter principen med summation av *binärt viktade* strömmar. Principen kan enkelt beskrivas med hjälp av ett resistansnät, brytare och en spänningskälla.



Figur 6.18 *Förenklad bild av D/A omvandlaren*

Studera Figur 6.18 som visar en förenklad bild av D/A-omvandlaren. Utgå från att I_{tot} på något sätt kan mätas och därmed ge en spänning över en belastning (proportionell mot I_{tot}, exempelvis med en operationsförstärkare)

Antag att strömbrytarna S_3,S_2,S_1 och S_0 styrs av ett pålagt bitmönster där "0" motsvarar *öppen* brytare och "1" motsvarar *sluten* brytare. Vi vill välja resistanserna R_3,R_2,R_1 och R_0 så att delströmmarna I_3,I_2,I_1,I_0 ger bidrag enligt följande *förutsättningar*:

Sluten S_3 motsvarar 2^3 (8) → $I_3=8\,I_0$
Sluten S_2 motsvarar 2^2 (4) → $I_2=4\,I_0$
Sluten S_1 motsvarar 2^1 (2) → $I_1=2\,I_0$
Sluten S_0 motsvarar 2^0 (1) → $I_0=1\,I_0$

Ohms lag ger sambanden

$$I_3 = E/R_3,\ I_2 = E/R_2,\ I_1 = E/R_1,\ I_0 = E/R_0,$$

om vi kombinerar dess samband med "förutsättningar" ovan får vi:

$$8I_0=E/R_3,\ 4I_0=E/R_2,\ 2I_0=E/R_1,\ I_0=E/R_0$$

Vi substituerar nu I_0 och får:

$$8\ E/R_0 = E/R_3 \Rightarrow R_3 = R_0/8$$
$$4\ E/R_0 = E/R_2 \Rightarrow R_2 = R_0/4$$
$$2\ E/R_0 = E/R_1 \Rightarrow R_1 = R_0/2$$

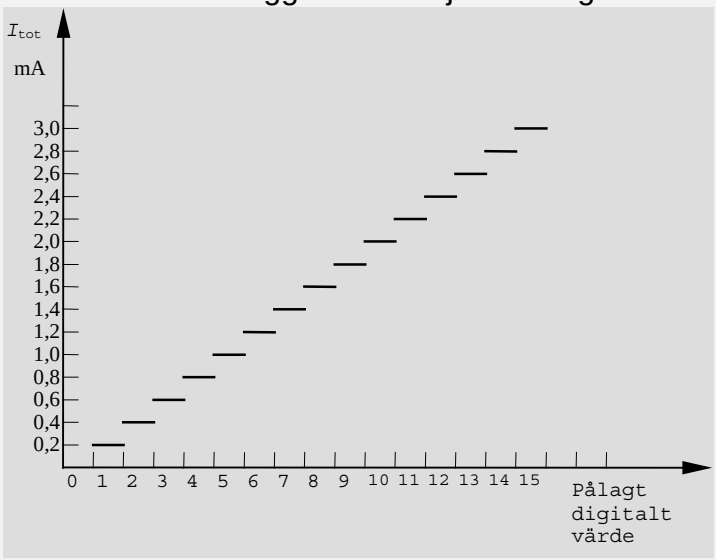
EXEMPEL:
Antag att vi har en referensspänning E =12 Volt. Vi väljer (godtyckligt) R_0 = 60 kΩ vilket ger I_0 = 0,2 mA. Av tidigare visade samband får vi:

R_3 = 7,5 kΩ och I_3 = 1,6 mA
R_2 = 15 kΩ och I_2 = 0,8 mA
R_1 = 30 kΩ och I_1 = 0,4 mA

Följande tabell illustrerar hur strömmen I_{tot} varierar med olika digitala värden, dvs olika inställningar av S_3,S_2,S_1 och S_0.

Digitalt Värde $S_3\,S_2\,S_1\,S_0$	Strömbidrag	I_{tot} (mA)
0 0 0 0		0
0 0 0 1	I_0	0,2
0 0 1 0	I_1	0,4
0 0 1 1	I_1+I_0	0,6
0 1 0 0	I_2	0,8
0 1 0 1	I_2+I_0	1,0
0 1 1 0	I_2+I_1	1,2
0 1 1 1	$I_2+I_1+I_0$	1,4
1 0 0 0	I_3	1,6
1 0 0 1	I_3+I_0	1,8
1 0 1 0	I_3+I_1	2,0
1 0 1 1	$I_3+I_1+I_0$	2,2
1 1 0 0	I_3+I_2	2,4
1 1 0 1	$I_3+I_2+I_0$	2,6
1 1 1 0	$I_3+I_2+I_1$	3,8
1 1 1 1	$I_3+I_2+I_1+I_0$	3,0

Tabellen kan också åskådliggöras av följande diagram:



Av diagrammet ser vi tydligt att vi inte får en *kontinuerlig* utsignal. Detta beror på att vi inte har tillräckligt hög *upplösning* hos D/A-omvandlaren. D/A-omvandlaren upplösning är omvänt proportionell mot antalet bitar *n* enligt:

$$upplösning = 1 / 2^n$$

för vårt fall alltså:

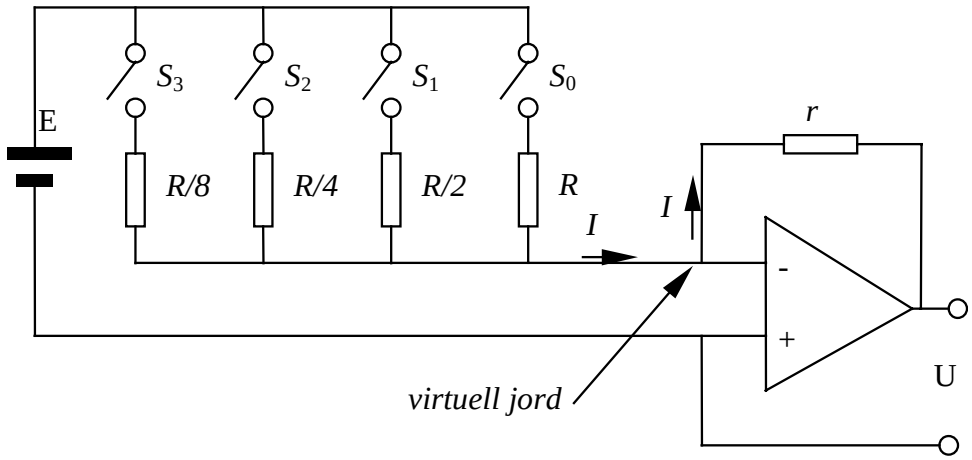
$$upplösning = 1 / 2^4 = 1/16 = \mathbf{6{,}25\ \%}$$

Följaktligen kommer *onoggrannheten* i omvandlingen att vara maximalt 6,25%.

Vanligtvis används D/A-omvandlare för 10-bitars digitala värden. Noggrannheten hos dessa omvandlare blir c:a 0,1% vilket är tillräckligt bra för de flesta användningsområden.

Vi har sett hur ett digitalt värde kan omvandlas till en analog ström. Ofta matas en sådan ström till en ström/spänningsomvandlare uppbyggd med en operationsförstärkare. Resulterande utsignal blir då en spänning som är

analog mot det pålagda bitmönstret. Figur 6.19 visar principen för en sådan koppling:



Figur 6.19 *Princip för D/A omvandling med operationsförstärkare*

Kopplingen kan enkelt analyseras med teori för *ideal operationsförstärkare*.

Kirchoffs lagar ger:

$$U + rI = 0$$

$$E - IR\left(\frac{S_3}{8} + \frac{S_2}{4} + \frac{S_1}{2} + \frac{S_0}{1}\right) = 0$$

Substitueras I får vi:

$$E + \frac{U}{r}R\left(\frac{S_3}{8} + \frac{S_2}{4} + \frac{S_1}{2} + \frac{S_0}{1}\right) = 0$$

Vilket ger den analoga utspänningen U ur:

$$U = -E\frac{r}{R}\left(\frac{1}{\frac{S_3}{8} + \frac{S_2}{4} + \frac{S_1}{2} + \frac{S_0}{1}}\right)$$

Vad händer om alla strömbrytarna är öppna? Nämnaren blir 0 !!!, dvs U blir ”minus oändligheten”.

Detta beror på egenskaper hos operationsförstärkaren. Nätet måste i själva verket kompletteras genom att man modifierar resistansnätet så att operationsförstärkarens ingång alltid belastas med samma resistans mot jord oavsett i vilka lägen strömbrytarna står.

Vi har sett hur en digital signal kan omvandlas till en analog ström eller spänning. Med hjälp av dessa kan vi styra yttre enheter som kräver någon form av kontinuerlig reglering. Vi har också sett att denna omvandling från den digitala signalen alltid kommer att ge *någon* onoggrannhet.

6.3.2 A/D omvandling

Precis som vissa styrobjekt kräver en kontinuerlig spänning karakteriseras många *givare* av att de levererar en kontinuerlig spänning. För att kunna bearbeta mätvärden från sådana givare krävs det att värdena först översätts till ett digitalt värde. Denna översättning kallas *A/D-Omvandling* (eng, *ADC, Analog to Digital Converter*). Det finns flera olika sätt att åstadkomma A/D-omvandling. Vi kommer här att översikligt beskriva de vanligaste: *Succesiv approximation*, *Rampomvandlare* och *Flash Converter*. Det som skiljer de olika typerna åt är *omvandlingstiden*, dvs den tid det tar att omvandla värdet av en kontinuerlig spänning på ingången till ett digitalt värde på utgången, och *tillverkningspriset*.

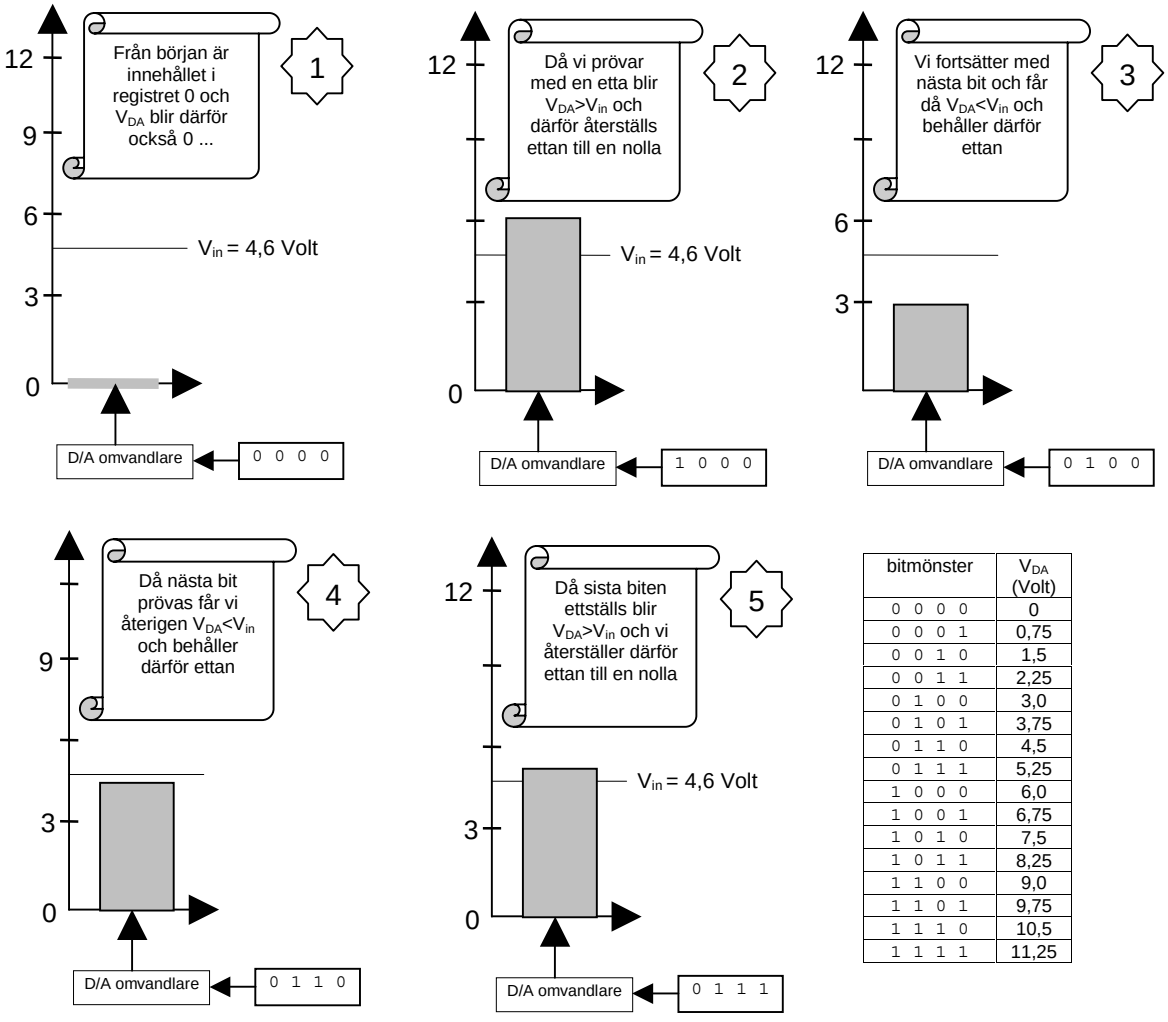
Omvandlare med succesiv approximation

Omvandlare med succesiv approximation är den vanligaste typen. Principen är mycket enkel. Omvandlaren arbetar med ett register som från början är nollställt. Innehållet i registret kopplas till en D/A-omvandlare vars utsignal (V_{DA}) jämförs med insignalen (V_{in}).

1. Den mest signifikanta biten i registret ettställs av logiken i omvandlaren, ett nytt värde, V_{in} , genereras av D/A-omvandlaren.
2. Om V_{in} nu är *större* än V_{DA} kommer den mest signifikanta biten i registret att *nollställas*, annars behålls ettan.
3. Förfarandet upprepas *för varje bit* i registret, i fallande nummerordning, tills även bit 0 testats på detta sätt. Resultatet i registret är nu det omvandlade digitala värdet.

Följande exempel beskriver arbetssättet.

EXEMPEL I detta exempel illustreras principen *succesiv approximation*. Vi använder här en 4-bitars omvandlare. I denna finns en intern D/A-omvandlaren som kan leverera max 11,25 Volt (=$V_{DA\text{-}max}$) och det största värde vi kan omvandla korrekt är därför just 11,25 Volt. Vi visar hur omvandlingen av spänningen V_{in} = 4,6 Volt utförs. Figurerna visar hur V_{in} omvandlas till digitalt värde i flera steg. Ju fler bitar som används, desto större noggrannhet fås. Oftast är det dock bara ett *närmevärde*, och därav namnet: succesiv approximation.

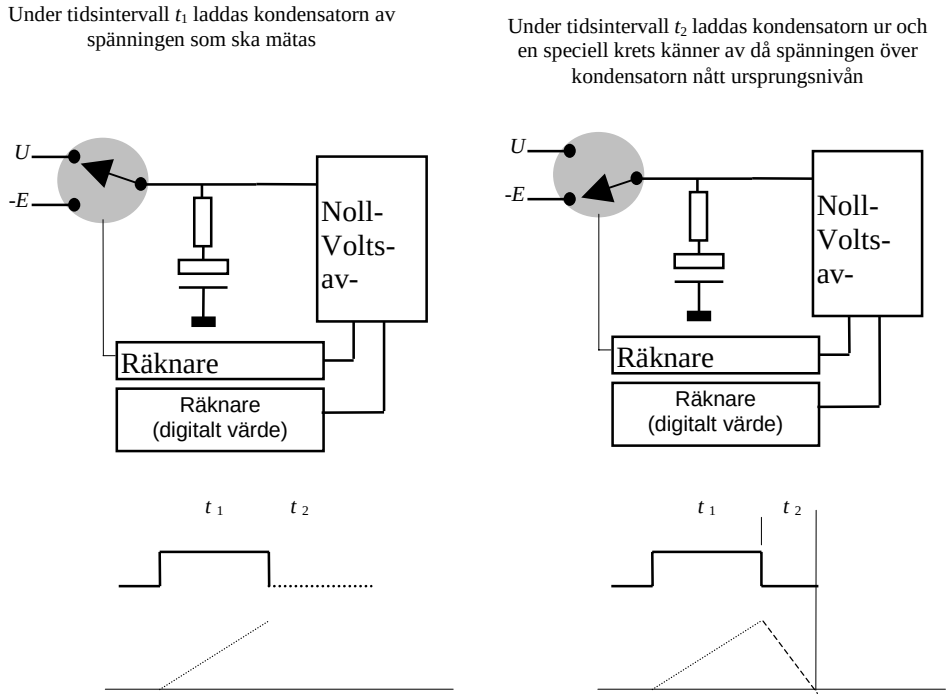


bitmönster	V_{DA} (Volt)
0 0 0 0	0
0 0 0 1	0,75
0 0 1 0	1,5
0 0 1 1	2,25
0 1 0 0	3,0
0 1 0 1	3,75
0 1 1 0	4,5
0 1 1 1	5,25
1 0 0 0	6,0
1 0 0 1	6,75
1 0 1 0	7,5
1 0 1 1	8,25
1 1 0 0	9,0
1 1 0 1	9,75
1 1 1 0	10,5
1 1 1 1	11,25

Rampomvandlare

I en *rampomvandlare* utnyttjas en kondensators förmåga att hålla elektriska laddningar. Förloppet styrs av en krets som bland annat innehåller en *klocka*. Klockan startas av styrkretsen och samtidigt påförs spänningen U (som ska mätas). Under en konstant tid t_1, låter man kondensatorn laddas upp av spänningen U. Därefter slår styrkretsen om och låter kondensatorn *laddas ur* till ursprungsvärdet, samtidigt aktiveras en krets som känner av då spänningen nått detta ursprungsvärde. Under den tid kondensatorn laddas ur (t_2), löper en räknare och innehållet i denna räknare (efter tiden t_2) är det digitala värde som motsvarar den uppmätta spänningen.

Principen kan åskådliggöras av följande figurer:



Om kondensatorn från början är oladdad gäller att den laddas upp (under tiden t_1) till:

$$t_1 \, U / RC$$

Under t_2 avger kondensatorn laddningen

$$t_2 \, (-E) / RC$$

då noll-volts-avkännaren ger signal till räknaren att kondensatorn laddats ur gäller alltså

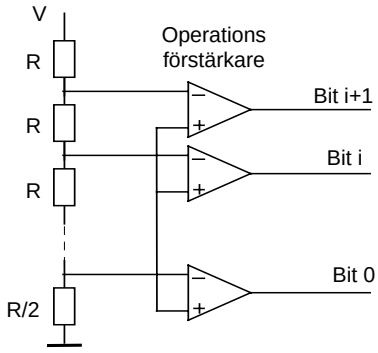
$$t_1 \, U / RC + t_2 \, (-E) / RC = 0$$

vilket vi kan skriva:

$$t_1 \, U = t_2 \, E$$

Av ekvationen ser vi att t_2 är proportionellt mot U och därmed också det digitala värde som räknats fram under tiden t_2.

Förloppet upprepas hela tiden och omvandlaren behöver därför inte nollställas. Observera också att mätnoggrannheten är oberoende av såväl komponenternas tidskonstant (RC-nätet) som den använda klockfrekvensen, dvs med vilken hastighet räknaren klockas.



Flash-Converter

Flash Converter

Den snabbaste typen av A/D omvandlare är en så kallad *Flash-Converter*. Principen för denna är att använda omslagspunkter hos operationsförstärkare med differentialingång. Spänningen som ska mätas appliceras på varje operationsförstärkares plus-ingång, medan omslagspunkterna ges av en resistansstege. Då spänningen som mäts överstiger respektive operations-förstärkares referensspänning (omslags-punkten) kommer op-förstärkaren att leverera en hög nivå ut (1), annars är nivån ut låg (0). En avkodare översätter därefter utsignalerna från operationsförstärkarna till ett digitalt värde.

EXEMPEL

För en 4-bitars Flash-Converter krävs 15 operationsförstärkare och utsignalerna avkodas enligt följande tabell:

b14 b0	Digitalt värde
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1	0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1	0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1	0 1 0 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1	0 1 1 0
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1	0 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	1 0 0 0
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1	1 0 0 1
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1	1 0 1 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1	1 0 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1	1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 1 0 1
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 1 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 1 1 1

Upplösningen hos FlashConvertern begränsas alltså av antalet op-förstärkare enligt:

$$x = 2^n - 1$$

där n är upplösningen i antalet antalet bitar och x, antalet op-förstärkare.

EXEMPEL:

Följande Flash-Converter (se figur nedan) byggs kring 8 st operationsförstärkare, en omkodare och en resistansstege med lika stora resistanser. Referensspänningen E är 12 Volt. Bestäm omslagsspänningarna E_n enligt figuren.

Lösning:

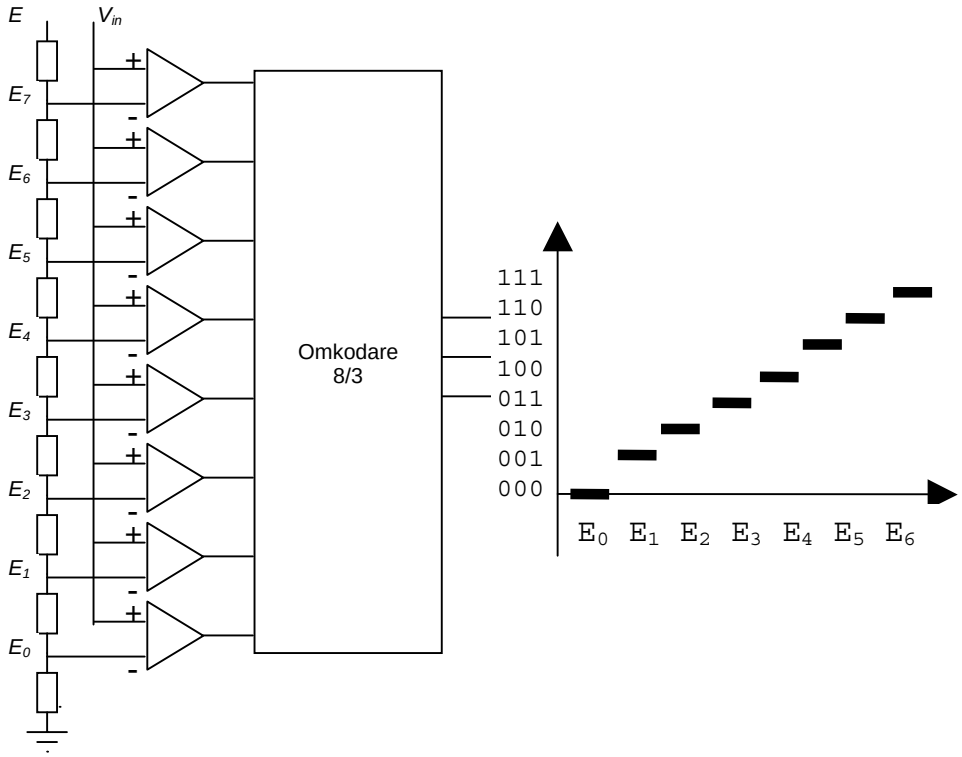
Totala resistansen i stegen är 9R. Med spänningsdelning får vi då:

$$E_n = \frac{12V(n+1)\cdot R\Omega}{9R\Omega}$$

dvs: $E_0 = E\ (R/9R) = E/9 = 1{,}33$ V

$U_1 = E(2R/9R) = 2E/9 = 2{,}66$ V

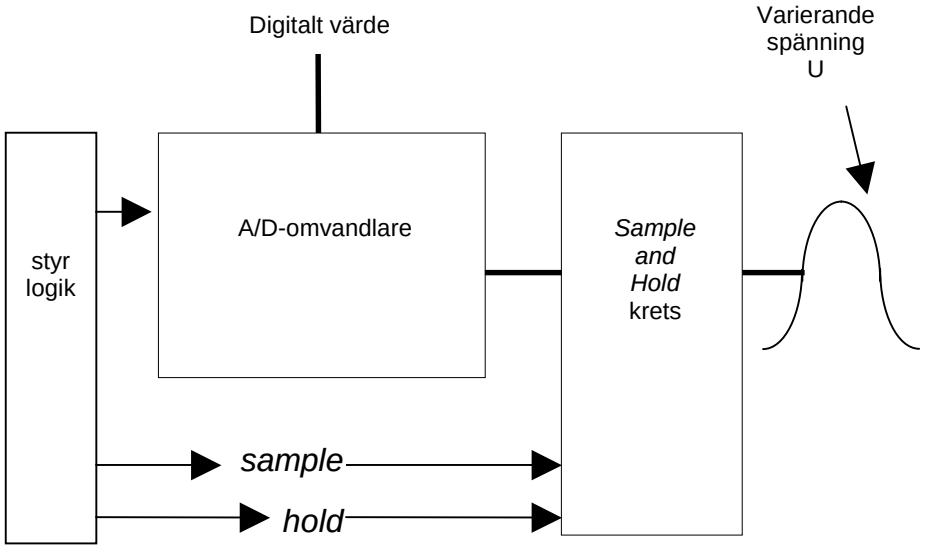
osv. Dvs, vi har inte speciellt god noggrannhet i resultatet. Se Figur 6.20 nedan.



Figur 6.20

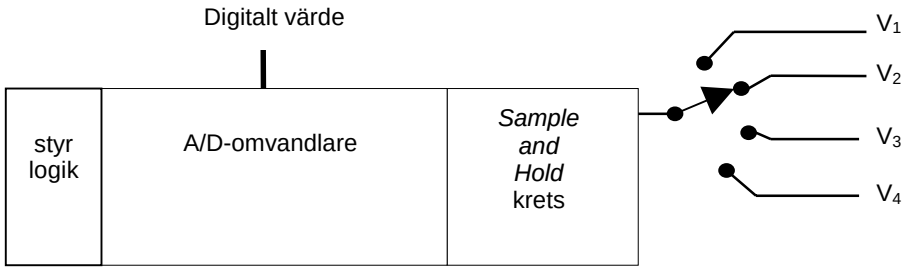
6.3.3 Sample and Hold

Ett problem för långsammare A/D-omvandlare är att spänningen varierar under omvandlingen. Man kan använda en speciell *sample-and-hold* krets för att "låsa" mätvärdet under den tid omvandlingen sker. Då signalen *sample* är aktiv, går den analoga spänningen rakt igenom kretsen och då signalen *hold* är aktiv "låses" signalen:



6.3.4 Multiplexande omvandlare

Multiplexande omvandlare kan användas för att åstadkomma omvandling från flera olika givare med hjälp av en enda A/D-omvandlare. Principen kan liknas vid en flervals-omkopplare som väljer den analoga inspänningen.



6.4 Sammanfattning

Överföring av data mellan olika system kan ske på två principiellt olika sätt, *parallellt* eller *seriellt*. Medan parallell överföring är vanlig då avstånden är korta används seriell överföring vanligtvis då avstånden sträcker sig över några meter.

Kompletta omvandlare såväl D/A som A/D, tillverkas i dag enkelt på ett enda chip. Oftast har dessa upplösningen 10-bitar, men varianter förekommer. Olika typer används beroende på tillämpningen. För exempelvis digitala voltmetrar används såväl rampomvandlare som omvandlare med succesiv approximation. För mer krävande tillämpningar som exempelvis digitala oscilloskop krävs flash-converters.

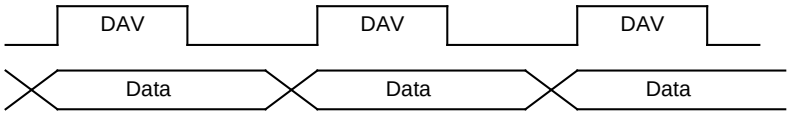
6.5 Övningsuppgifter

Uppgift 46: *Ovillkorlig överföring*. Skriv en subrutin `GetTxt` som tar emot en ASCII-sträng från en 8-bitars inport och placerar texten i minnet med start på adress $1000. Inportens dataregister är placerad på adress $A000. Subrutinen anropas på följande sätt:

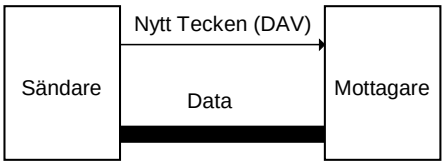
```
MOVEA.L     #$1000,A0
BSR         GetText
```

Textsträngen är avslutad med ASCII-tecknet NUL ($00).

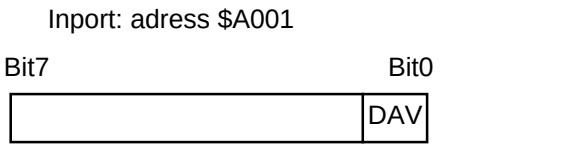
Uppgift 47: *Villkorlig överföring med handskakning*. Ändra subrutinen i föregående uppgift så att villkorlig överföring utnyttjas. Till förfogande finns ytterligare en inport (adress $A001) där bit 0 (*DAV=Data AVailable*) anger huruvida sändaren har skickat ett nytt tecken eller inte. Se *timingdiagrammet*.



Timingdiagram



Signaler på bussen



Statusregister

Uppgift 48: Mellan Malmö och Kiruna är det knappa 150 mil. Hur lång tid tar det för en signal att förflytta sig denna sträcka tur och retur? Använd ljushastigheten $3 \cdot 10^8$ m/s.

Uppgift 49: Hur lång tid tar det att skifta ut 1500 bitar när skiftregistret klockas med 10 kHz?

Uppgift 50: Vad är skillnaden på asynkron och synkron överföring.

Uppgift 51: Vilken utspänning fås från en 8-bitars D/A-omvandlare, som summerar binärt viktade strömma, om det pålagda bitmönstret är `10000000` och referensspänningen är 24 Volt?

Uppgift 52: Ange resultatet vid en 8-bitars A/D-omvandling, enligt principen sucessiv approximation, av spänningen 4,6 Volt. Referensspänningen är 16 Volt. Hur stort blir felet?

Uppgift 53: Bestäm omslagsspänningarna (E0 - E14) för en Flash-Converter (där alla resistanser är lika) med referensspänningen 16 Volt.

Index

Appendix A
Instruktionslista för MC68 (MC68340) Assemblerdirektiv

MOVE "move data from source to destination"
MOVEQ "move quick"
MOVEA "move address"

Syntax:

```
MOVE.<x>    <ea>,<ea>
MOVEQ       #<data>,Dn
MOVEA.<x>   <ea>,An
```

Attribut:
Storlek <x> kan vara: B (*byte*), W (*word*) eller L (*long*) utom för adresseringsmod An, där storlek kan vara W eller L. MOVEQ är alltid *long*.

Beskrivning:
Kopierar källoperand till destination. MOVEQ *kan* användas om källoperanden är en konstant med högst 8 bitar och destinationsoperanden är ett dataregister. Konstanten teckenutvidgas före operationen. MOVEA.<x> *ska* användas om destinations-operanden är ett adressregister.

<ea> som källa			
Dn	**X**	**(xxx).W**	**X**
An	**X**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

<ea> som destination			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	Nollställs alltid

Anm: MOVEA.<x> påverkar ej flaggorna.

MOVE to CCR "move to condition code register"

Syntax:

```
MOVE        <ea>,CCR
```

Beskrivning:
Källoperanden kopieras till processorns flaggregister. Operandens storlek är 16 bitar men endast de 8 minst signifikanta bitarna används.

<ea> som källa			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor: Beror av operanden

MOVE from CCR "move from condition code register"

Syntax:

```
MOVE        CCR,<ea>
```

Beskrivning:
Innehållet i CC registret kopieras till destinationen. Detta är en teckenutvidgande 8 till 16-bitars operation eftersom endast innehållet i CCR (8 bitar) läses.

<ea> som destination			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor: Påverkas ej

MOVE from SR "move from status register"

Syntax:

```
MOVE        SR,<ea>
```

Beskrivning:
Innehållet i statusregistret (16 bitar) kopieras till destinationen.

Anmärkning:
Detta en *priviligierad* instruktion.

<ea> som destination			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor: Påverkas ej

MOVEP "move peripheral data"

Syntax:

```
MOVEP.<x>     Dx,(d16,An)
MOVEP.<x>     (d16,An),Dx
```

Attribut:
Storlek <x> kan vara W (*word*) eller L (*long*)
Beskrivning:
Kopierar data mellan dataregister och udda eller jämna adresser i minnet. Instruktionen kan användas då 8-bitars periferikretsar adresseras på en 16-bitars databuss. Om EA är en jämn adress kommer påföljande jämna adresser att användas. Om EA är en udda adress kommer påföljande udda adresser att användas. Om storleken anges till *long* kommer 4 bytes att kopieras med början på den mest signifikanta byten i dataregistret. Om storleken är *word* kommer två bytes från dataregistret att kopieras med början på den byte som bildas av bitarna 8-15, därefter den byte som bildas av bitarna 0-7.

Flaggor: Påverkas ej

MOVEM "move multiple registers"

Syntax:

```
MOVEM.<x>     <register>,<ea>
MOVEM.<x>     <ea>,<register>
```

Attribut:
Storlek <x> kan vara: W (*word*) eller L (*long*)
Beskrivning:
Godtyckligt antal registerinnehåll kan kopieras till konsekutiva adresser i minnet. Hela registerinnehållet (32 bitar) kan kopieras (MOVEM.L) eller endast de 16 minst signifikanta bitarna (MOVEM.W). Listan av register kan anges på flera sätt. Enskilda register separeras med '/', konsekutiva register separeras med '-'.

EXEMPEL: *lista av register*

```
D0/D1/A0      register d0,d1 och a0
D0-D2/A0-A1   register d0,d1,d2,a0 och a1
```

EXEMPEL: *spara registerinnehåll*
Instruktionen kan användas för att spara registerinnehåll på stacken med:

```
MOVEM.L       D0-D7/A0-A6,-(A7)
```

Den andra formen av MOVEM.<x> kopierar konsekutivt minnesinnehåll till de angivna registren.

EXEMPEL: *återställ registerinnehåll*
Hela registeruppsättningen kan återställas från stacken med:

```
MOVEM.L       (A7)+,D0-D7/A0-A6
```

<ea> som destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**-**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

<ea> som källa			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**-**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor: Påverkas ej

EXG "exchange registers"

Syntax:

```
EXG           Dx,Dy
EXG           Ax,Ay
EXG           Ax,Dy
```

Beskrivning:
Skiftar innehållen (32 bitar) mellan två register. Såväl data- som adressregister kan anges.

Flaggor: Påverkas ej

SWAP "swap register halves"

Syntax:

```
SWAP          Dn
```

Beskrivning:
Instruktionen skiftar innehållen (16 bitar) mellan de mest, respektive minst, signifikanta orden i ett dataregister.

Flaggor	
X	Påverkas ej
N	1 om mest signifikanta biten av det 32-bitars resultatet är 1, 0 annars
Z	1 om det 32-bitars resultatet är 0, 0 annars
V	Nollställs alltid
C	Nollställs alltid

LEA "load effective address"

Syntax:

```
LEA           <ea>,An
```

Beskrivning:
Effektiva adressen (*long*) laddas i angivet adressregister.

<ea> som källa			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**-**		
-(An)	**-**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor: Påverkas ej

PEA "push effective address"

Syntax:

```
PEA          <ea>
```

Beskrivning:
Effektiva adressen (*long*) bestäms och placeras på stacken.

<ea>			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**-**		
-(An)	**-**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor: Påverkas ej

LINK "link and allocate"

Syntax:

```
LINK         An,#<data>
LINK.W       An,#<data>
LINK.L       An,#<data>
```

Beskrivning:
Innehållet i det av källoperanden angivna adressregistret placeras på stacken, stackpekarens nya innehåll kopieras till adressregistret, slutligen adderas konstanten <data> till stackpekaren.
Anmärkning:
LINK och UNLK instruktionerna kan användas för att skapa en länkad lista av lokala data och parametrar.

Flaggor: Påverkas ej

UNLK "unlink"

Syntax:

```
UNLK         An
```

Beskrivning:
Inehållet i det angivna adressregistret kopieras till stackpekaren, därefter placeras adressen överst på stacken i adressregistret och stackpekaren ökas med 4.

Flaggor: Påverkas ej

ADD "add binary"
ADDI "add immediate"
ADDQ "add quick"
ADDA "add address"

Syntax:

```
ADD.<x>      <ea>,Dn
ADD.<x>      Dn,<ea>
ADDI.<x>     #<data>,<ea>
ADDQ.<x>     #<data>,<ea>
ADDA.<x>     <ea>,An
```

Attribut:
Storlek <x> kan vara W eller L då någon operand är ett adressregister, <x> kan vara B, W eller L i övriga fall.
Beskrivning:
Källoperand och destinationsoperand adderas, resultatet placeras i destinationsoperanden.

Anmärkning:
ADDQ.<x> är en kortare form av ADDI.<x> som kan användas om den adderade konstanten är i intervallet 1-8. An *kan* då användas som destination. Om destinationsoperanden är ett adressregister skall annars formen ADDA.<x> användas. Om källoperanden är en konstant (<data>) skall formen ADDI.<x> (alternativt ADDQ.<x>) användas.

<ea> som destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**
<ea> som källa			
Dn	**X**	**(xxx).W**	**X**
An	**X**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor	
X	1 om carry genererats, 0 annars
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om spill genererats, 0 annars
C	1 om carry genererats, 0 annars

Anmärkning:
Instruktionen ADDA.<x> påverkar ej flaggorna.

ADDX "add extended"

Syntax:

```
ADDX.<x>     Dy,Dx
ADDX.<x>     -(Ay),-(Ax)
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Adderar källoperanden och X-bit till destinationsoperanden och placerar resultatet i destinationen.

Flaggor	
X	1 om carry genererats, 0 annars
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om spill genererats, 0 annars
C	1 om carry genererats, 0 annars

CLR "clear an operand"

Syntax:

```
CLR.<x>      <ea>
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Samtliga bitar (8,16 eller 32 beroende på <x>) i operanden nollställs.

<ea>			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	Nollställs alltid
Z	Ettställs alltid
V	Nollställs alltid
C	Nollställs alltid

CMP "compare"
CMPA "compare address"

Syntax:

```
CMP.<x>      <ea>,Dn
CMPA.<x>     <ea>,An
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*) utom då någon operand är ett adressregister då endast W och L är tillåtna.
Beskrivning:
Subtraherar källoperanden från destinationsoperanden. Operationen påverkar endast innehållet i CC-registret.
Anmärkning:
Instruktionsformen CMPI (se nedan) *ska* användas om källoperanden är en konstant.

<ea> som källa			
Dn	**X**	**(xxx).W**	**X**
An	**X**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om spill genererats, 0 annars
C	1 om lånesiffra genererats, 0 annars

CMPI "compare immediate"

Syntax:

```
CMPI.<x>     #<data>,<ea>
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*).
Beskrivning:
Subtraherar källoperanden från destinationsoperanden. Operationen påverkar endast innehållet i CC-registret.

<ea> som destination			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om spill genererats, 0 annars
C	1 om lånesiffra genererats, 0 annars

CMP2 "compare register against bounds

Syntax:

```
CMP2.<x>     <ea>,Rn
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Jämför innehållet i Rn med en övre gräns (UB) respektive undre gräns (LB) och påverkar flaggorna. Effektiva adressen utgör adress till både UB och LB, LB först. Rn kan vara ett adress- eller data-register. Instruktionen är identisk med chk2 bortsett från att chk2-instruktionen kan generera *trap*.

<ea> som källa			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**-**		
-(An)	**-**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor	
X	Påverkas ej
N	Odefinierad
Z	1 om Rn är lika någon gräns, 0 annars
V	Odefinierad
C	1 om Rn utanför gränserna, 0 annars

CMPM "compare memory"

Syntax:

```
CMPM.<x>        (Ay)+,(Ax)+
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Jämför minnesinnehåll och påverkar flaggorna, ökar därefter innehållet i de adressregister som angetts, med 1 om storleken är byte, 2 om storleken är *word* och med 4 om storleken är *long*.

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om spill genererats, 0 annars
C	1 om lånesiffra genererats, 0 annars

EXT "sign extend"

Syntax:

```
EXT.W           Dn
EXT.L           Dn
EXTB.L          Dn
```

Beskrivning:
Teckenutvidgar innehållet i ett dataregister från *byte* till *word* (EXT.W) eller från *word* till *long* (EXT.L) eller från *byte* till *long* (EXTB.L).

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	Nollställs alltid

DIVS "signed divide"
DIVU "unsigned divide"

Syntax:

```
DIVS            <ea>,Dn
DIVS.W          <ea>,Dn     32/16➔16r-16q
DIVU            <ea>,Dn
DIVU.W          <ea>,Dn     32/16➔16r-16q
DIVS.L          <ea>,Dq     32/32➔32q
DIVU.L          <ea>,Dq     32/32➔32q
DIVS.L          <ea>,Dr:Dq
                            64/32➔32r-32q
DIVU.L          <ea>,Dr:Dq
                            64/32➔32r-32q
DIVSL.L         <ea>,Dr:Dq
                            32/32➔32r-32q
DIVUL.L         <ea>,Dr:Dq
                            32/32➔32r-32q
```

Beskrivning:
Dividerar två tal och placerar resultatet i ett eller eventuellt två dataregister. Resultatet är en *rest* (remainder) och en *kvot* (quotient). Teckenbiten för "resten" är densamma som hos dividenden (DIVS) om "resten" är skild från 0.

Två speciella tillstånd kan uppträda:

- Division med 0, vilket föranleder en *trap*
- Spill kan uppträda under instruktionen. Detta detekteras och avspeglas i CC-registret men operanderna påverkas ej.

<ea> som källa			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor	
X	Påverkas ej
N	1 om kvoten negativ, 0 annars, odefinierad om spill
Z	1 om kvoten är 0, 0 annars, odefinierad om spill
V	1 om division spill genererats, 0 annars
C	Nollställs alltid

Anm: DIVS är kortform för DIVS.W, DIVU är kortform för DIVU.W, godtyckligt skrivsätt kan användas.

MULS "signed multiply"
MULU "unsigned multiply"

Operation: Destination * Källa ⇒ Destination
Syntax:

```
MULS    <ea>,Dn
MULS.W  <ea>,Dn     16•16➔32
MULU    <ea>,Dn
MULU.W  <ea>,Dn     16•16➔32
MULS.L  <ea>,DI     32•32➔32
MULU.L  <ea>,DI     32•32➔32
MULS.L  <ea>,Dh-DI  32•32➔64
MULU.L  <ea>,Dh-DI  32•32➔64
```

Beskrivning:
Multiplicerar två tal och placerar resultatet i det eller de dataregister som angetts som destination. MULS förutsätter tal på tvåkomplementform medan MULU utför multiplikation på tal utan tecken.

<ea> som källa			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om spill, 0 annars
C	Nollställs alltid

Anm: Spill kan endast uppkomma då två 32-bitars operander multipliceras och resultatet placeras i *ett* dataregister. MULS är kortform för MULS.W, MULU är kortform för MULU.W, godtyckligt skrivsätt kan användas.

NEG "negate"

Syntax:

```
NEG.<x>      <ea>
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Operanden subtraheras från 0, resultatet placeras i destinationen, med andra ord: tvåkomplementering av operanden.

<ea>			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	1 om carry genererats, 0 annars
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om spill genererats, 0 annars
C	0 om resultatet är 0, 1 annars

NEGX "negate with extend"

Syntax:

```
NEGX.<x>     <ea>
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Operanden och X-bit subtraheras från 0, resultatet placeras i destinationen.

<ea>			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	1 om lånesiffra genererats, 0 annars
N	1 om resultatet negativt, 0 annars
Z	0 om resultatet är skilt från 0, påverkas ej annars
V	1 om spill genererats, 0 annars
C	1 om lånesiffra genererats, 0 annars

SUB "subtract binary"
SUBI "subtract immediate"
SUBQ "subtract quick"
SUBA "subtract address"

Syntax:

```
SUB.<x>      <ea>,Dn
SUB.<x>      Dn,<ea>
SUBI.<x>     #<data>,<ea>
SUBQ.<x>     #<data>,<ea>
SUBA.<x>     <ea>,An
```

Attribut:
Storlek <x> kan vara w eller l då någon av operanderna är ett adressregister. <x> kan vara B, W eller L i övriga fall.
Beskrivning:
Källoperand subtraheras från destinationsoperand, resultatet placeras i destinationsoperanden.

Anmärkningar:
SUBQ.<x> är en kortare form av SUBI.<x> som kan användas om den subtraherade konstanten är i intervallet 1-8. I detta fall kan även ett adressregister anges som destinationsoperand. I övrigt gäller att: om destinationsoperanden är ett adressregister *ska* formen SUBA.<x> användas. Om källoperanden är en konstant *ska* formen SUBI.<x> användas,

<ea> som källa			
Dn	**X**	**(xxx).W**	**X**
An	**X**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**
<ea> som destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	1 om lånesiffra genererats, 0 annars
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om spill genererats, 0 annars
C	1 om lånesiffra genererats, 0 annars

Anmärkning:
Instruktionen SUBA.<x> påverkar ej flaggorna.

SUBX "subtract with extend"

Syntax:

```
SUBX.<x>     Dy,Dx
SUBX.<x>     -(Ay),-(Ax)
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Subtraherar källoperanden och X-bit från destinations-operanden och placerar resultatet i destinationen.

Flaggor	
X	1 om lånesiffra genererats, 0 annars
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om spill genererats, 0 annars
C	1 om lånesiffra genererats, 0 annars

TAS "test and set an operand"

Syntax:

```
TAS          <ea>
```

Beskrivning:
Testar operanden som anges av effektiva adressen. N- och Z-flaggorna sätts beroende på operanden. Den mest signifikanta biten i operanden sätts till 1. Operationen är odelbar, dvs hela instruktionen utförs alltid för att möjliggöra synkronisering i multiprocessor applikationer. Operandes storlek är alltid *byte*.

<ea> som destination			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	Nollställs alltid

TST "test an operand"

Syntax:

```
TST.<x>      <ea>
```

Attribut:
Storlek <x> kan vara W eller L om operanden är ett adressregister. B, W eller L annars.
Beskrivning:
Jämför operanden med 0. Resultatet av jämförelsen påverkar innehållet i CC-registret.

<ea>			
Dn	**X**	**(xxx).W**	**X**
An	**X**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	Nollställs alltid

AND "AND logical"
ANDI "AND immediate"

Syntax:

```
AND.<x>      <ea>,Dn
AND.<x>      Dn,<ea>
ANDI.<x>     #<data>,Dn
```

Attribut:
Storlek <x> kan vara B (*byte*), W(*word*) eller L (*long*)
Beskrivning:
Utför logiskt AND mellan källoperand och destinations-operand. Resultatet placeras i destinationsoperanden.
Anmärkning:
Om källoperanden är #<data> skall instruktionsformen ANDI.<x> användas.

<ea>som källa			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**
<ea>som destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	Nollställs alltid

OR "inclusive OR logical"
ORI "inclusive OR immediate"

Syntax:

```
OR.<x>      <ea>,Dn
OR.<x>      Dn,<ea>
ORI.<x>     #<data>,Dn
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Utför logiskt OR mellan källoperand och destinations-operand. Resultatet placeras i destinationsoperanden.
Anmärkning:
Om källoperanden är #<data> skall instruktionsformen ORI.<x> användas.

<ea>som källa			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**
<ea>som destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	Nollställs alltid

EOR "exclusive OR logical"
EORI "exclusive OR immediate"

Syntax:

```
EOR.<x>     Dn,<ea>
EORI.<x>    #<data>,Dn
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Utför exklusivt OR mellan källoperand och destinationsoperand. Resultatet placeras i destinationsoperanden.
Anmärkning:
Om källoperanden är #<data> skall instruktionsformen EORI.<x> användas.

<ea>som destination			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	Nollställs alltid

NOT "logical complement"

Syntax:

```
NOT.<x>     <ea>
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Utför ett-komplementering av operanden.

<ea>som destination			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	1 om resultatet negativt, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	Nollställs alltid

ASL "arithmetic shift left"
ASR "arithmetic shift right"

Syntax:

```
ASd.<x>     Dx,Dy
ASd.<x>     #<data>,Dy
ASd         <ea>
```

Attribut:
Storlek <x> kan vara B (byte), W (*word*) eller L (*long*)
Beskrivning:
Destinationsoperanden skiftas aritmetiskt i den riktning som anges av instruktionen. <*antal steg*> kan anges på två olika sätt, om källoperanden är ett dataregister anger innehållet i detta register antalet skift. I den andra formen anges antalet skift som en konstant (1-8). I en tredje instruktionsform kan innehållet på en minnesadress skiftas. Detta skift är alltid ett steg och storleken på operanden är alltid *word*.

ASL: Operanden skiftas *vänster*. Den mest signifikanta skiftade biten kopieras till C-biten respektive X-biten i CC-registret. På den minst signifikanta bitens plats skiftas 0 in.

ASR: Operanden skiftas *höger*. Den mest signifikanta skiftade biten kopieras, dvs operandens tecken bibehålls. Den minst signifikanta skiftade biten kopieras till C-biten respektive X-biten i CC-registret.

<ea>			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	se ovan
N	1 om resultatet är 0, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Ettställs om den mest signifikanta biten ändras någon gång under operationen. Nollställs annars
C	se ovan

LSL "logical shift left"
LSR "logical shift right"

Syntax:

```
LSd.<x>      Dx,Dy
LSd.<x>      #<data>,Dy
LSd          <ea>
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Destinationsoperanden skiftas i den riktning som anges av instruktionen. <*antal steg*> kan anges på två olika sätt, om källoperanden är ett dataregister anger innehållet i detta register antalet skift. I den andra formen anges antalet skift som en konstant (1-8). I en tredje instruktionsform kan innehållet på en minnesadress skiftas. Detta skift är alltid ett steg och storleken på operanden är alltid *word*.

LSL: Operanden skiftas vänster. Den mest signifikanta skiftade biten kopieras till C-biten respektive X-biten i CC-registret. På den minst signifikanta bitens plats skiftas 0 in. Operationen är ekvivalent med **ASL**.

LSR: Operanden skiftas höger. Den mest signifikanta biten ersätts av 0. Den minst signifikanta skiftade biten kopieras till C-biten respektive X-biten i CC-registret.

<ea>			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	se ovan
N	1 om resultatet är 0, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	se ovan

ROL "rotate left without extend"
ROR "rotate right without extend"

Syntax:

```
ROd.<x>      Dx,Dy
ROd.<x>      #<data>,Dy
ROd          <ea>
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Destinationsoperanden roteras i den riktning som anges av instruktionen. <*antal steg*> kan anges på två olika sätt, om källoperanden är ett dataregister anger innehållet i detta register antalet skift. I den andra formen anges antalet skift som en konstant (1-8). I en tredje instruktionsform kan innehållet på en minnesadress skiftas. Detta skift är alltid ett steg och storleken på operanden är alltid *word*.

ROL: Operanden skiftas vänster. Den mest signifikanta skiftade biten kopieras till den minst signifikanta skiftade biten och till C-biten i CC-registret.

ROR: Operanden skiftas höger. Den minst signifikanta skiftade biten kopieras till C-biten i CC-registret respektive den mest signifikanta skiftade biten.

<ea>			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	se ovan
N	1 om resultatet är 0, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	se ovan

ROXL "rotate left with extend"
ROXR "rotate right with extend"

Syntax:

```
ROXd.<x>     Dx,Dy
ROXd.<x>     #<data>,Dy
ROXd         <ea>
```

Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Destinationsoperanden roteras i den riktning som anges av instruktionen. <*antal steg*> kan anges på två olika sätt, om källoperanden är ett dataregister anger innehållet i detta register antalet skift. I den andra formen anges antalet skift som en konstant (1-8). I en tredje instruktionsform kan innehållet på en minnesadress skiftas. Detta skift är alltid ett steg och storleken på operanden är alltid *word*.

ROXL: Operanden skiftas vänster. X-biten i CC-registret kopieras till den minst signifikanta skiftade biten. Den mest signifikanta skiftade biten kopieras till C-biten respektive X-biten i CC-registret.

ROXR: Operanden skiftas höger. X-biten i CC-registret kopieras till den mest signifikanta skiftade biten. Den minst signifikanta skiftade biten kopieras till C-biten respektive X-biten i CC-registret.

<ea>			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	se ovan
N	1 om resultatet är 0, 0 annars
Z	1 om resultatet är 0, 0 annars
V	Nollställs alltid
C	se ovan

BTST "test a bit"

Syntax:

```
BTST.L      Dn,Dm
BTST.L      #<data>,Dn
BTST.B      Dn,<ea>
BTST.B      #<data>,<ea>
```

Beskrivning:
En bit hos destinationsoperanden testas, resultatet av testen återspeglas av Z-biten i CC-registret. Destinationsoperandens testade bit påverkas inte. Om destinations-operanden är ett dataregister anger källoperanden den aktuella biten (modulo 32), dvs vilken som av de 32 bitarna i dataregistret kan testas. Om destinations-operanden är en minnesadress kommer 8-bitar att läsas från denna adress. Den testade biten kan anges på två olika sätt, i instruktionens form

```
BTST.B      Dn,<ea>
```

anges bit-numret av innehållet i det specificerade dataregistret. I instruktionens form

```
BTST.B      #<data>,<ea>
```

anges bit-numret av en konstant.

Anmärkning:
Observera att bitnumret anges som bit-namn, dvs den minst signifikanta biten betecknas 0 och den mest signifikanta biten betecknas 31.

<ea> som destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor	
X	Påverkas ej
N	Påverkas ej
Z	1 om den testade biten är 0, 0 annars
V	Påverkas ej
C	Påverkas ej

BSET "test a bit and set"

Syntax:

```
BSET.L      Dn,Dm
BSET.L      #<data>,Dn
BSET.B      Dn,<ea>
BSET.B      #<data>,<ea>
```

Beskrivning:
En bit hos destinationsoperanden testas, resultatet av testen återspeglas av Z-biten i CC-registret. Destinationsoperandens testade bit ettställs därefter. Om destinations-operanden är ett dataregister anger källoperanden den aktuella biten (modulo 32), dvs vilken som av de 32 bitarna i dataregistret kan manipuleras. Om destinations-operanden är en minnesadress kommer 8-bitar att läsas från denna adress, därefter utförs operationen (modulo 8) och slutligen skrivs resultatet tillbaks i minnet. Den testade biten kan anges på två olika sätt, i instruktionens form

```
BSET.B      Dn,<ea>
```

anges bit-numret av innehållet i det specificerade dataregistret. I instruktionens andra form

```
BSET.B      #<data>,<ea>
```

anges bit-numret av en konstant.

Anmärkning:
Observera att bitnumret anges som bit-namn, dvs den minst signifikanta biten betecknas 0 och den mest signifikanta biten betecknas 31.

<ea> som destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	Påverkas ej
Z	1 om den testade biten är 0, 0 annars
V	Påverkas ej
C	Påverkas ej

BCLR "test a bit and clear"

Syntax:

```
BCLR.L      Dn,Dm
BCLR.L      #<data>,Dn
BCLR.B      Dn,<ea>
BCLR.B      #<data>,<ea>
```

Beskrivning:
En bit hos destinationsoperanden testas, resultatet av testen återspeglas av Z-biten i CC-registret. Destinationsoperandens testade bit nollställs därefter. Om destinations-operanden är ett dataregister anger källoperanden den aktuella biten (modulo 32), dvs vilken som av de 32 bitarna i dataregistret kan manipuleras. Om destinations-operanden är en minnesadress kommer 8-bitar att läsas från denna adress, därefter utförs operationen (modulo 8) och slutligen skrivs resultatet tillbaks i minnet. Den testade biten kan anges på två olika sätt, i instruktionens första form

```
BCLR.B Dn,<ea>
```

anges bit-numret av innehållet i det specificerade dataregistret. I instruktionens andra form

BCLR.B #<data>,<ea>

anges bit-numret av en konstant.

Anmärkning:

Observera att bitnumret anges som bit-namn, dvs den minst signifikanta biten betecknas 0 och den mest signifikanta biten betecknas 31.

<ea> som destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	Påverkas ej
Z	1 om den testade biten är 0, 0 annars
V	Påverkas ej
C	Påverkas ej

BCHG "test a bit and change"

Syntax:

```
BCHG.L      Dn,Dm
BCHG.L      #<data>,Dn
BCHG.B      Dn,<ea>
BCHG.B      #<data>,<ea>
```

Beskrivning:

En bit hos destinationsoperanden testas, resultatet av testen återspeglas av Z-biten i CC-registret. Destinationsoperandens testade bit inverteras därefter. Om destinationsoperanden är ett dataregister anger källoperanden den aktuella biten (modulo 32), dvs vilken som av de 32 bitarna i dataregistret kan manipuleras. Om destinationsoperanden är en minnesadress kommer 8-bitar att läsas från denna adress, därefter utförs operationen (modulo 8) och slutligen skrivs resultatet tillbaks i minnet. Den testade biten kan anges på två olika sätt, i instruktionens första form

BCHG.B Dn,<ea>

anges bit-numret av innehållet i det specificerade dataregistret. I instruktionens andra form

BCHG.B #<data>,<ea>

anges bit-numret av en konstant.

Anmärkning:

Observera att bitnumret anges som bit-namn, dvs den minst signifikanta biten betecknas 0 och den mest signifikanta biten betecknas 31.

<ea> som destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	Påverkas ej
Z	1 om den testade biten är 0, 0 annars
V	Påverkas ej
C	Påverkas ej

ABCD "add decimal with extend"

Syntax:

```
ABCD        Dy,Dx
ABCD        -(Ay),-(Ax)
```

Beskrivning:

Källoperanden adderas till destinationsoperanden tillsammans med X-biten i CCR. Operationen utförs med användning av BCD-aritmetik. Operanderna kan anges på två olika sätt:

1. data register till data register: dvs operanderna finns i de dataregister som anges av instruktionen.
2. minnesadress till minnesadress: operandernas adress anges av det minskade innehållet i något adressregister.

Operandens storlek är alltid 8 bitar (*byte*).

Flaggor	
X	1 om carry (decimal) genererats, 0 annars
N	Odefinierad
Z	0 om resultatet är skilt från 0, påverkas annars inte
V	Odefinierad
C	1 om carry (decimal) genererats, 0 annars

SBCD "subtract decimal with extend"

Syntax:

```
SBCD        Dy,Dx
SBCD        -(Ay),-(Ax)
```

Beskrivning:

Källoperanden subtraheras från destinationsoperanden tillsammans med X-biten i CCR. Operationen utförs med användning av BCD-aritmetik. Operanderna kan anges på två olika sätt:

1. data register till data register: dvs operanderna finns i de dataregister som anges av instruktionen.
2. minnesadress till minnesadress: operandernas adress anges av det minskade innehållet i något adressregister.

Operandens storlek är alltid 8 bitar (byte).

Flaggor	
X	1 om lånesiffra (decimal) genererats, 0 annars
N	Odefinierad
Z	0 om resultatet är skilt från 0, påverkas annars inte
V	Odefinierad
C	1 om lånesiffra (decimal) genererats, 0 annars

NBCD "negate decimal with extend"

Syntax:

```
NBCD        <ea>
```

Beskrivning:
Operanden och X-biten subtraheras från 0. Resultatet placeras på operandens adress. Operationen utförs med BCD-aritmetik. Instruktionen skapar alltså 10-komplementet av operanden om X är 0, och 9-komplementet av operanden om X är 1. Operandens storlek är alltid 8 bitar (byte).

<ea>			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	1 om lånesiffra (decimal) genererats, 0 annars
N	Odefinierad
Z	0 om resultatet är skilt från 0, påverkas annars inte
V	Odefinierad
C	1 om lånesiffra (decimal) genererats, 0 annars

Bcc "branch conditionally"

Syntax:

```
Bcc.S  <label>      8-bitars offset
Bcc.B  <label>      8-bitars offset
Bcc    <label>      se anmärkning
Bcc.W  <label>      16-bitars offset
Bcc.L  <label>      32-bitars offset
```

Anmärkning:
Med denna form kommer assemblatorn att försöka välja den kortast möjliga offseten.
Beskrivning:
Om det angivna villkoret är uppfyllt utförs hoppet, annars fortsätter exekveringen vid den påföljande instruktionen. Testen ger resultatet av tidigare instruktion som påverkat flaggorna i CCR.

Följande villkor (cc**) kan anges:**

BHI	High	$\overline{C} \wedge \overline{Z}$
BLS	Low or Same	$C \vee Z$
BCC	Carry Clear	$\overline{C}$
BCS	Carry Set	C
BNE	Not Equal	$\overline{Z}$
BEQ	Equal	Z
BVC	Overflow Clear	$\overline{\ }$
BVS	Overflov Set	
BPL	Plus	$\overline{N}$
BMI	Minus	N
BGE	Greater or Equal	$N \wedge V \vee \overline{N} \wedge \overline{V}$
BLT	Less Than	$N \wedge \overline{V} \vee \overline{N} \wedge V$
BGT	Greater Than	$N \wedge V \wedge \overline{Z} \vee \overline{N} \wedge \overline{V} \wedge \overline{Z}$
BLE	Less or Equal	$Z \vee N \wedge \overline{V} \vee \overline{N} \wedge V$

Flaggor: Påverkas ej

DBcc "test condition, decrement and branch"

Syntax:

```
DBcc   Dn,<label>     16-bitars offset
```

Beskrivning:
Instruktionen används för att skapa programslingor. Den har tre parametrar: ett villkor (jämför "branch conditionally"), ett dataregister och en offset. Instruktionen testar först villkoret för att se om detta är uppfyllt. Om detta är sant utförs inget mer. Om termineringsvillkoret inte är sant minskas innehållet i det angivna dataregistret med 1. Om resultatet i dataregistret nu är -1 fortsätter exekveringen med nästa instruktion, om inte fortsätter exekveringen vid den adress som anges av PC+"offset". Första testen ger resultatet av tidigare instruktion som påverkat flaggorna i CCR.

Följande villkor (cc) kan anges:

DBHI	High	$\overline{C} \wedge \overline{Z}$
DBLS	Low or Same	$C \vee Z$
DBCC	Carry Clear	$\overline{C}$
DBCS	Carry Set	C
DBNE	Not Equal	$\overline{Z}$
DBEQ	Equal	Z
DBVC	Overflow Clear	$\overline{\ }$
DBVS	Overflov Set	
DBPL	Plus	$\overline{N}$
DBMI	Minus	N
DBGE	Greater or Equal	$N \wedge V \vee \overline{N} \wedge \overline{V}$
DBLT	Less Than	$N \wedge \overline{V} \vee \overline{N} \wedge V$
DBGT	Greater Than	$N \wedge V \wedge \overline{Z} \vee \overline{N} \wedge \overline{V} \wedge \overline{Z}$
DBLE	Less or Equal	$Z \vee N \wedge \overline{V} \vee \overline{N} \wedge V$
DBRA	False	0
DBT	True	1
DBF	False	0

Anmärkning:
I formen DBF (eller DBRA), dvs villkoret är aldrig uppfyllt, kan instruktionen användas för att åstadkomma ett givet antal iterationer.

EXEMPEL:

```
        MOVE.L #loopcount-1,D0
loop:
        ...
        ...
        DBRA           D0,loop
```

Instruktioner mellan loop och DBRA utförs *loopcount* gånger

Flaggor: Påverkas ej

Scc "set according to condition"

Syntax:

```
Scc          <ea>
```

Beskrivning:
Om det angivna villkoret är uppfyllt sätts operanden till 1, annars sätts operanden till 0. Villkoret är resultat av tidigare instruktion som påverkat flaggorna i CCR. Operandens storlek är alltid *byte*.

Följande villkor (cc) kan anges:

SHI	High	$\overline{C} \wedge \overline{Z}$
SLS	Low or Same	$C \vee Z$
SCC	Carry Clear	$\overline{C}$
SCS	Carry Set	C
SNE	Not Equal	$\overline{Z}$
SEQ	Equal	Z
SVC	Overflow Clear	$\overline{\ }$
SVS	Overflov Set	
SPL	Plus	$\overline{N}$
SMI	Minus	N
SGE	Greater or Equal	$N \wedge V \vee \overline{N} \wedge \overline{V}$
SLT	Less Than	$N \wedge \overline{V} \vee \overline{N} \wedge V$
SGT	Greater Than	$N \wedge V \wedge \overline{Z} \vee \overline{N} \wedge \overline{V} \wedge \overline{Z}$
SLE	Less or Equal	$Z \vee N \wedge \overline{V} \vee \overline{N} \wedge V$
ST	True	1
SF	False	0

<ea>			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor: Påverkas ej

BRA "branch always"

Syntax:

```
BRA.S  <label>      8-bitars offset
BRA.B  <label>      8-bitars offset
BRA    <label>      se anmärkning
BRA.W  <label>      16-bitars offset
BRA.L  <label>      32-bitars offset
```

Anmärkning:
Med denna form kommer assemblatorn att försöka välja den kortast möjliga offseten.
Beskrivning:
Exekveringen fortsätter på den adress som ges av PC+offset. "offset" är ett tal på tvåkomplementform. Vid adresssberäkningen innehåller PC adressen till BRA-instruktionen + 2.
Flaggor: Påverkas ej

BSR "branch to subroutine"

Syntax:

```
BSR.S  <label>      8-bitars offset
BSR.B  <label>      8-bitars offset
BSR    <label>      se anmärkning
BSR.W  <label>      16-bitars offset
BSR.L  <label>      32-bitars offset
```

Anmärkning:
Med denna form kommer assemblatorn att försöka välja den kortast möjliga offseten.
Beskrivning:
Adressen till instruktionen omedelbart efter BSR placeras på stacken. Exekveringen fortsätter på den adress som ges av PC+offset. "offset" är ett tal på tvåkomplement-form. Vid adresssberäkningen innehåller PC adressen till BSR-instruktionen + 2.
Flaggor: Påverkas ej

JMP "jump"

Syntax:

```
JMP    <ea>
```

Beskrivning:
Exekveringen fortsätter på den adress som anges av effektiva adressen.

<ea>			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**-**		
-(An)	**-**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor: Påverkas ej

JSR "jump to subroutine"

Syntax:

```
JSR          <ea>
```

Beskrivning:
Adressen till nästa instruktion placeras på stacken. Exekveringen fortsätter på den adress som angetts av effektiva adressen.

<ea>			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**-**		
-(An)	**-**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor: Påverkas ej

RTR "return and restore condition codes"

Syntax:

```
RTR
```

Beskrivning:
Det 8-bitars ord som ligger överst på stacken kopieras till CC-registret. Stack-pekaren ökas med 2. Därefter placeras den 32-bitars adress som nu ligger överst på stacken i PC och stackpekaren ökas med 4.
Flaggor: Se ovan

RTS "return from subroutine"

Syntax:

```
RTS
```

Den 32-bitars adress som ligger överst på stacken kopieras till PC, stackpekaren ökas med 4.
Flaggor: Påverkas ej

RESET "reset peripherals"

Syntax:

```
RESET
```

Beskrivning:
Processorn aktiverar *reset*-signalen och tvingar därmed periferienheter till ett väldefinierat tillstånd. Processorns tillstånd påverkas ej utan exekveringen fortsätter med nästa instruktion.
Anmärkning:
Instruktionen utförs endast om processorn är i *supervisor mode*, om inte sker en *privilege violation* trap

Flaggor: Påverkas ej

BKPT "breakpoint"

Syntax:

```
BKPT            #<data>
```

Beskrivning:
Instruktionen stöder implementering av brytpunkter i debuggers. För en utförlig diskussion om användningen av BKPT, se MC68340 *user's manual*.

Flaggor: Påverkas ej

RTE "return from exception"

Syntax:

```
RTE
```

Beskrivning:
Det översta ordet (16 bitar) på stacken kopieras till statusregistret. Stackpekaren ökas med 2. Den 32-bitars adress som nu ligger överst på stacken kopieras till programräknaren. Därefter ökas stackpekaren med 4.
Anmärkning:
Instruktionen utförs endast om processorn är i *supervisor mode*, om inte sker en *privilege violation* trap.
Flaggor: Se ovan

RTD "return and deallocate"

Syntax:

```
RTD             #<disp>
```

Beskrivning:
Kopierar värdet överst på stacken till programräknaren, adderar därefter <disp>, teckenutvidgat från 16 bitar till 32, till stackpekaren. Föregående programräknarvärde förloras.
Flaggor: Påverkas ej

STOP "load status register and stop"

Syntax:

```
STOP            #<data>
```

Beskrivning:
Operanden (word) kopieras till statusregistret. Programräknaren pekar på nästa instruktion i koden. Processorn avbryter exekveringen. Exekvering av nästa instruktion återupptas då ett RESET tillstånd, eller ett avbrott med högre prioritet än processorns uppträder
Anmärkning:
Instruktionen utförs endast om processorn är i *supervisor mode*, om inte sker en *privilege violation* trap
Flaggor: Se ovan

ORI to SR "or immediate to status register"
ORI to CCR "or immediate to Condition Code Register"

Syntax:

```
ORI             #<data>,SR
ORI             #<data>,CCR
```

Beskrivning:
Utför logiskt OR mellan <data> och register, resultatet placeras i processorns register. Om destinationsoperanden är CCR kommer de 8 minst signifikanta bitarna i <data> att maskas med det tidigare innehållet i CCR och på nytt placeras i CCR. Om destinationsoperanden är SR, dvs processorns statusregister kommer <data> att maskas med det tidigare innehållet i SR och på nytt placeras i SR. **ORI to CCR** kan alltid utföras medan **ORI to SR** endast utförs om processorn är i *supervisor mode*.
Flaggor: Bestäms av operand och tidigare innehåll

ANDI to SR "and immediate to status register"
ANDI to CCR "and immediate to condition code register"

Syntax:

```
ANDI            #<data>,SR
ANDI            #<data>,CCR
```

Beskrivning:
Utför logiskt AND mellan <data> och processorregister, resultatet placeras i processorns register. Om destinations-operanden är CCR kommer de 8 minst signifikanta bitarna i <data> att maskas med det tidigare innehållet i CCR och på nytt placeras i CCR. Om destinations-operanden är SR, dvs processorns statusregister kommer <data> att maskas med det tidigare innehållet i SR och på nytt placeras i SR. **ANDI to CCR** kan alltid utföras medan **ANDI to SR** endast utförs om processorn är i *supervisor mode*.
Flaggor: Bestäms av operand och tidigare innehåll

EORI to SR "exclusive or immediate to status register"
EORI to CCR "exclusive or immediate to condition code register"

Syntax:

```
EORI            #<data>,SR
EORI            #<data>,CCR
```

Beskrivning:
Utför logiskt EXCLUSIVE OR mellan <data> och processorregister, resultatet placeras i processorns register. Om destinationsoperanden är CCR kommer de 8 minst signifikanta bitarna i <data> att maskas med det tidigare innehållet i CCR och på nytt placeras i CCR. Om destinationsoperanden är SR, dvs processorns statusregister

kommer <data> att maskas med det tidigare innehållet i SR och på nytt placeras i SR. **EORI to CCR** kan alltid utföras medan **EORI to SR** endast utförs om processorn är i *supervisor mode*.
Flaggor: Bestäms av operand och tidigare innehåll

MOVE to SR "move to status register"

Syntax:
```
MOVE        <ea>,SR
```
Beskrivning:
Källoperanden kopieras till processorns statusregister. Operandens storlek är 16 bitar och hela statusregistret påverkas.
Anmärkning:
Instruktionen utförs endast om processorn är i *supervisor mode*, om inte sker en *privilege violation* trap

<ea> som källa			
Dn	**X**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**X**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor: Bestäms av operand.

MOVE USP "move user stack pointer"

Syntax:
```
MOVE        USP,An
MOVE        An,USP
```
Beskrivning:
Innehållet i *user stack pointer* kopieras till ett adressregister respektive: innehållet i ett adressregister kopieras till *user stack pointer*. Instruktionen är priviligierad och användbar endast då processorn är i *supervisor mode* och skall ändra den stackpekare som är aktiv i *user mode* alternativt spara den stackpekare som är aktiv i *user mode*.
Flaggor: Påverkas ej

MOVEC "move control register"

Syntax:
```
MOVEC       Rn,Rc
MOVEC       Rc,Rn
```
Attribut: Inga, storlek är alltid 32 bitar (*long*)
Beskrivning:
Instruktionens första form kopierar innehållet i ett generellt register (32 bitar) Rn (Dn eller An) till ett kontrollregister Rc. Instruktionens andra form kopierar innehållet i ett kontrollregister (32 bitar) till ett generellt register.

Kontrollregister:

SFC	source function code register
DFC	destination function code register
USP	user stack pointer
VBR	vector base register

Flaggor: Påverkas ej

MOVES "move address space"

Syntax:
```
MOVES.<x>   Rn,<ea>
MOVES.<x>   <ea>,Rn
```
Attribut:
Storlek <x> kan vara B (*byte*), W (*word*) eller L (*long*)
Beskrivning:
Instruktionens första form kopierar innehållet i Rn (Dn eller An) till det adressrum som anges av DFC.
Instruktionens andra form kopierar data från det adressrum som anges av SFC till ett register Dn eller An.
Anmärkning:
Instruktionen utförs endast om processorn är i *supervisor mode*, om inte sker en *privilege violation* trap.

<ea> som källa/destination			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor: Påverkas ej

TRAP *"trap"*

Syntax:
```
TRAP    #<Vector>
```
Beskrivning:
Processorn initierar *exception processing*. Vektornumret (0-15) anger vilken *exception* rutin som skall exekveras.
Anmärkning:
I *db68* används TRAP #14 för inbyggda subrutiner och TRAP #15 för brytpunktshantering.
Flaggor: Påverkas ej

TRAPV "trap on overflow"

Syntax:
```
TRAPV
```
Beskrivning:
Om processorns V-flagga (overflow) är 1 som resultat av tidigare operation kommer *exception processing* att vidtas. I annat fall är operationen ekvivalent med *no operation*.
Flaggor: Påverkas ej

TRAPcc "trap on condition"

Syntax:

```
TRAPcc
TRAPcc.W    #<data>
TRAPcc.L    #<data>
```

Beskrivning:
Om angivet villkor är *sant* som resultat av tidigare operation kommer *exception processing* (vektor 7) att vidtas. I annat fall är operationen ekvivalent med *no operation*. data kan användas av en exception-rutin för vidare hantering. Följande former kan användas:

TRAPHI	High	$\overline{C} \wedge \overline{Z}$
TRAPLS	Low or Same	$C \vee Z$
TRAPCC	Carry Clear	$\overline{C}$
TRAPCS	Carry Set	C
TRAPNE	Not Equal	$\overline{Z}$
TRAPEQ	Equal	Z
TRAPVC	Overflow Clear	$\overline{\ }$
TRAPVS	Overflov Set	
TRAPPL	Plus	$\overline{N}$
TRAPMI	Minus	N
TRAPGE	Greater or Equal	$N \wedge V \vee \overline{N} \wedge \overline{V}$
TRAPLT	Less Than	$N \wedge \overline{V} \vee \overline{N} \wedge V$
TRAPGT	Greater Than	$N \wedge V \wedge \overline{Z} \vee \overline{N} \wedge \overline{V} \wedge \overline{Z}$
TRAPLE	Less or Equal	$Z \vee N \wedge \overline{V} \vee \overline{N} \wedge V$
TRAPT	True	1
TRAPF	False	0

Flaggor: Påverkas ej

CHK "check registers against bounds"

Syntax:

```
CHK    <ea>,Dn
```

Attribut: <x> kan vara B (*byte*), W (*word*) eller L (*long*).
Beskrivning:
De 16 minst signifikanta bitarna i Dn jämförs med "övre gräns" som utgörs av operanden på effektiva adressen. Detta tal betraktas som tal på tvåkomplements-form. Om innehållet i Dn är mindre än noll, eller större än operanden på effektiva adressen startas *exception processing*.

<ea> (chk)			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**-**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**-**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**-**

Flaggor	
X	Påverkas ej
N	1 om Dn < 0, 0 om Dn > ea-operand, annars odefinierad
Z	Odefinierad
V	Odefinierad
C	Odefinierad

CHK2 "check registers against bounds 2"

Syntax:

```
CHK2.<x>    <ea>,Rn
```

Attribut: <x> kan vara B (*byte*), W (*word*) eller L (*long*).
Beskrivning:
Jämför värdet i Rn med undre gräns (LB) och övre gräns (UB). Effektiva adressen innehåller LB:UB. Rn kan vara ett data- eller adress-register.

<ea> (chk2)			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**-**		
-(An)	**-**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor	
X	Påverkas ej
N	Odefinierad
Z	1 om Rn är lika med någon av gränserna, 0 annars O
V	Odefinierad
C	1 om Rn utanför gränserna, 0 annars

NOP "no operation"

Syntax:

```
NOP
```

Beskrivning:
Programräknaren ökas och pekar därefter på nästa instruktion.

Flaggor: Påverkas ej.

BGND "enter background mode"

Syntax:

```
BGND
```

Beskrivning:
Processorn upphör med instruktionsexekvering och vidtar *background mode* om denna är tillåten (*enabled*). Om *background mode* ej är tillåten vidtar *illegal instruction exception processing*. Se handbok för respektive microcontroller för ytterligare information om *background mode*.
Flaggor: Påverkas ej.

LPSTOP "low-power stop"

Syntax:

```
LPSTOP    #<data>
```

Beskrivning:
Operanden överförs till statusregistret, programräknaren avanceras och pekar på nästa instruktion, därefter upphör instruktions-hämtningen och processorn försätts i *strömsnål mode*. Instruktionsexekvering återupptas efter en *trace*, *interrupt* eller *reset* exception. Trace-exception kan endast inträffa om T-biten i SR är 1 då instruktionen utförs.

TBLS "table look-up and interpolate signed")
TBLSN
TBLU "table look-up and interpolate unsigned"
TBLUN

Syntax:

```
TBLS.<x>      <ea>,Dx
        Resultat avrundas
TBLSN.<x>     <ea>,Dx
        Resultat avrundas ej
TBLU.<x>      Dym:Dyn,Dx
        Resultat avrundas
TBLUN.<s>     Dym:Dyn,Dx
        Resultat avrundas ej
```

Attribut:
Storlek <x> kan vara B (*byte*) W (*word*) eller L (*long*).
Beskrivning:
TBLS, TBLSN, TBLU och TBLUN används tillsammans med *tabeller* som innehåller styckevis linjära funktioner (formen "table lookup and interpolate" TBLx <ea>,Dn) eller *data i register* (formen "register interpolate" TBLx Dym:Dyn,Dx) för att bestämma funktionsvärden.

För "table lookup and interpolate" innehåller Dx (bit 15-0) den oberoende variabeln X. <ea> pekar ut en tabell, innehållande tal *med* tecken, storleken av dessa tal kan vara *byte*, *word* eller *long*. I tabellen ges en linjäriserad representation av den beroende variabeln Y som funktion av X. I allmänhet består den oberoende variabeln X av en 8-bitars heltalsdel och en 8-bitars bråktalsdel, dvs en *implicit* decimalpunkt mellan bit 7 och 8 i Dx. Heltalsdelen, skalad med storleken, används som offset i tabellen och bråktalsdelen används för att interpolera fram funktionsvärdet ur två konsekutiva poster i tabellen.

För "register interpolate" sker interpolationen mellan värden i register Dym och Dyn i stället för poster i tabellen. I denna form används endast bråktalsdelen och heltalsdelen ignoreras. Denna form kan användas tillsammans med "table lookup and interpolate" för att modellera funktioner av flera oberoend variabler.

Avrundning (TBLS)	
Justerad bråktalsdifferens	Avrundning
≤–1/2	–1
>–1/2 och <1/2	0
≥1/2	+1

Avrundning (TBLU)	
Justerad bråktalsdifferens	Avrundning
≥1/2	+1
<1/2	0

Returvärdet lagras i Dx. Bråktalsdelen placeras alltid i Dx[7..0], om storleken är *byte* placeras heltalsdelen i Dx[15..8], om storleken är *word* placeras heltalsdelen i Dx[23..8], om storleken är *long* placeras de 24 minst signifikanta bitarna av heltalsdelen i Dx[31..8]. För TBLS och TBLSN *teckenutvidgas* heltalsdelen från *byte* och *word* resultat. För TBLU och TBLUN lämnas motsvarande delar av registret opåverkat.

<ea> som källa			
Dn	**-**	**(xxx).W**	**X**
An	**-**	**(xxx).L**	**X**
(An)	**X**	**#<data>**	**-**
(An)+	**X**		
-(An)	**X**		
(d16,An)	**X**	**(d16,PC)**	**X**
(d8,An,Xn)	**X**	**(d8,PC,Xn)**	**X**
(bd,An,Xn)	**X**	**(bd,PC,Xn)**	**X**

Flaggor	
X	Påverkas ej
N	1 om den mest signifikanta biten i resultatet är 1, 0 annars
Z	1 om resultatet är 0, 0 annars
V	1 om heltalsdelen av det icke avrundade långa resultatet är utanför området $-(2^{23}) \leq$ Resultat $\leq (2^{23})-1$, 0 annars
C	Nollställs alltid

Assemblerdirektiv:

ORG **placering av kod och data**

Under assembleringen används en så kallad *location counter* (LC) för att hålla reda på aktuell adress för maskinkoden. Användningen av LC är central för flera assemblerdirektiv. Då ett assembleringspass startas tilldelas LC värdet 0. Detta innebär att kod och data placeras konsekutivt från adress 0 och uppåt. För att påverka placering av kod och data i datorns primär-minne används **ORG** (origin). Detta direktiv instruerar assemblatorn att tilldela LC ett nytt värde. Framåtreferenser är *inte* tillåtna med **ORG**.

Assemblersyntax:

```
        ORG     <adress>        LC = adress
```

EXEMPEL:

```
        ORG     $1000
        ...
```

påföljande kod/data placeras med början på adress $1000.

EXEMPEL:

```
CODE    EQU     $1000
        ORG     CODE
        .. är tillåtet men
        ORG     CODE
CODE    EQU     $1000
```

.. är *inte* tillåtet eftersom inga framåtreferenser tillåts med detta direktiv.

USE **öppna ny källtextfil**

Om fler filer än en skall assembleras tillsammans kan **USE** direktivet användas. **USE** får assemblatorn att spara nödvändig information om den första källtextfilen, öppna den nya filen och börja assembleringen av denna. Då den nya filen assemblerats återställs den gamla och assembleringen fortsätter efter **USE**- direktivet.

EXEMPEL
Antag en källtextfil, *projekt.s68* som innehåller följande direktiv:

```
        ...
        ...
        USE             defs.s68
        USE             init.s68
        USE             last.s68
        ...
        ...
```

Då filen *projekt.s68* assembleras kommer laddfil *projekt.s2* och list-filen *projekt.lst* att skapas. Dessa innehåller nu den kompletta assembleringen av filerna:

projekt.s68 före första **USE**-direktiv
defs.s68, init.s68, last.s68 och
projekt.s68 efter sista **USE**-direktiv

Källtextfilerna påverkas ej av detta. De filer som inkluderats på detta sätt kan också innehålla **USE**-direktiv, dock är den maximala nivån möjliga **USE**-direktiv 10

DC **initiering av dataareor**

DC.B
DC.W
DC.L

Dataareor kan skapas i minnet med *define constant* direktiven. Beroende på vilken typ av data som ska definieras används olika former av direktivet.

DC **definiera teckensträng**

Assemblersyntax:

```
[LABEL]         DC      "<textsträng>"
```

DC-direktivet används för att skapa initierade textsträngar. Strängen tolkas som ASCII, men följande specialtecken kan också anges:

Tecken	Betydelse	ASCII-hex
\n	*new line*	$0A
\d	*kill*	$04
\b	*backspace*	$08
\t	*tabulator*	$0B
\f	*form feed*	$0C
\r	*return*	$0D
\"	*quote*	$22
\\	*backslash*	$5C

DC.<s> **definiera konstant**

Assemblersyntax:

```
[LABEL]   DC.B  <minnesinnehåll (byte)>
[LABEL]   DC.W  <minnesinnehåll (word)>
[LABEL]   DC.L  <minnesinnehåll (long)>
```

DC.<s> direktivet används för att initiera minnet med data. Data kan vara på formen *byte* (8-bitar), *word* (16-bitar) eller *long* (32-bitar).

EXEMPEL:

```
        ORG             $9000
        DC.B            $12
```

placerar $12 på adress $9000

EXEMPEL:

```
        ORG             $9000
        DC.W            $1234
```

placerar $12 på adress $9000 och $34 på adress $9001

EXEMPEL:

```
        ORG             $9000
        DC.L            $12345678
```

placerar $12 på adress $9000, $34 på adress $9001, $56 på adress $9002 och $78 på adress $9003.

DS.<s> **allokera minne statiskt**

Assemblersyntax:

```
[LABEL]         DS.B            <antal bytes>
[LABEL]         DS.W            <antal words>
[LABEL]         DS.L            <antal longs>
```

DS.<s> används för att reservera dataareor i minnet. Minnets innehåll är ej definierat.

EXEMPEL:

```
        DS.B    256             reservera 256 bytes
```

EQU skapa symbolnamn och tilldela värde

För att öka läsbarheten hos ett assemblerprogram är det lämpligt att använda symboler för konstanter, i stället för att skriva dessa med absoluta tal. Direktivet EQU instruerar assemblatorn att tilldela en symbol ett värde. Närhelst denna symbol påträffas i den fortsatta assembleringen kommer värdet att ersätta symbolen.

EXEMPEL:
```
CODE   EQU    $8000
*  symbolen CODE får värde $8000

       ....
       ....

       ORG    CODE   samma som "ORG $8000"
```

EXEMPEL:

```
IRQmask:  EQU $0700 interrupt mask
SMode:    EQU $2000 supervisor mode
```

Uttryck som används med EQU får ej innehålla framåtreferenser.

ALIGN justera LC

Assemblersyntax:
```
ALIGN
```

Eftersom MC68000/CPU32 endast kan referera *word* och *long* på jämn adress kan i bland utfyllnad bli nödvändig. Typiskt inträffar detta efter direktiv som kan initiera udda antal bytes i minnet (**dc** och **dc.b**). Align kommer att placera en utfyllnadsbyte om LC är udda, annars påverkas inte LC.

NOLIST stoppa utskrift till listfil

Assemblersyntax:
```
NOLIST
```

Direktivet har endast betydelse för listfilen. Om listfil skapas kommer detta direktiv att stoppa utskrift, av påföljande rader, till listfilen.

LIST återstarta utskrift till listfil

Assemblersyntax:
```
LIST
```

Direktivet har endast betydelse för listfilen. Om listfil skapas och ett tidigare NOLIST direktiv har stoppat utskrifter till listfilen kommer **LIST** direktivet att starta utskrift igen.

PAGE börja ny sida i listfil

Assemblersyntax:
```
PAGE
```

Direktivet har endast betydelse för listfilen. Om listfil skapas kommer ett *form-feed* tecken att matas ut till den, dvs påföljande text i listfilen börjar på ny sida.

END avsluta assemblering

Assemblersyntax:
```
END
```

Assemblering avbryts efter END-direktivet.

Appendix B
Facit till Övningsuppgifter

1.1

a) 4 = %100=$4
b) 8 = %1000=$8
c) 7 = %111=$7
d) 15 = %1111=$F
e) 30 = %11110=$1E
f) 128 = %10000000=$80

g) %110 = $6=6
h) %1100 = $C=12
i) %1110 = $E=14
j) %10101= $15=21
k) %11010= $1A=26
l) %11111= $1F=31

m) $5 = %101=5
n) $9 = %1001=9
o) $A = %1010=10
p) $12 = %10010=18
q) $BB = %10111011=187
r) $C0 = %11000000=192

1.2

a) 22 = %(NBCD) 0010 0010
b) 18 = %(NBCD) 0001 1000
c) 30 = %(NBCD) 0011 0000
d) 75 = %(NBCD) 0111 0101
e) 54 = %(NBCD) 0101 0100
f) 69 = %(NBCD) 0110 1001

1.3

a) a = $61
b) Z = $5A
c) ! = $21
d) 7 = $37
e) , = $2C
f) ? = $3F

1.4

a) %01010101 = 85
b) %00111100 = 60
c) %10101010 = 170
d) %11000011 = 195

1.5

A	B	C	A+BC	(A+B)(A+C)
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

1.6

A	B	C	A(B+C)	AB+AC
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

1.7 $U_1 = A\bar{B}\bar{C}+AB\bar{C} = B\bar{C}$
$U_2 = A\bar{B}C+AB\bar{C}$
$U_3 = A\bar{B}C$
$U_4 = A\bar{B}C+A\bar{B}C$
$U_5 = A\bar{B}\bar{C}+AB\bar{C}$
$U_6 = A\bar{B}\bar{C}+ABC$
$U_7 = A\bar{B}C+A\bar{B}C$

1.8 Med en asynkron buss används så kallade *handskakningssignaler* (styrbussen) för synkronisering. Med en *synkronbuss* används *tiden* för synkroniseringen.

1.9 En multiplexbuss använder *samma ledningar* (samma buss) för både adress och data. *Först* kommer adress *sedan* kommer data. Bussen är synkron.

1.10 Ett *icke-flyktigt* minne behåller sitt innehåll vid spänningsbortfall. Det är inte fallet för ett *flyktigt* minne.

1.11 a) 2048 bytes
b) 4096 bytes
c) 8192 bytes

1.12 a) 8 kByte = 2^{13} bytes, dvs 13 adressledningar
b) 64 kByte = 2^{16} bytes, dvs 16 adressledningar
c) 1 Mbyte = 2^{20} bytes, dvs 20 adressledningar

1.13

a) -10 = % -(0000 1010) = 1111 0110
b) -38 = % -(0010 0110) = 1101 1010
c) -42 = % -(0010 1010) = 1101 0110
d) -56 = % -(0011 1000) = 1100 1000

1.14 a)
X = 00100100
Y = 01001010
R = 01101110
C = 0 Z = 0

Operationen innebär X=36, Y= 74, R= 110, (36+74=110) Eftersom talen betraktas på *binärform* är talområdet (8 bitar) 0-255 och således resultatet korrekt

b)
X = 10111100
Y = 01000100
R = 00000000
C = 1 Z = 1

Operationen innebär X=188, Y= 68, R= 0, men 188+68=256 !! Eftersom talen betraktas på *binärform* är talområdet (8 bitar) 0-255 och således resultatet utanför talområdet. Carryflaggan sätts därför till 1, dessutom blir i detta fall innehållet i registret 0 varför Z-flaggan också sätts till 1.

c)
```
X = 10000001
Y = 10000001
R = 00000010
C = 1    Z = 0
```
Operationen innebär X=129, Y=129, R=2, men 129+129=258!!. Eftersom talen betraktas på *binärform* är talområdet (8 bitar) 0-255 och således resultatet utanför talområdet. Carryflaggan sätts därför till 1, resultatet i registret är dock skilt från 0 varför Z-flaggan sätts till 0.

1.15 a)
```
X = 00100100
Y = 01001010
R = 01101110
V = 0    Z = 0
```
Operationen innebär X=36, Y= 74, R= 110, (36+74=110). Eftersom talen betraktas på *tvåkomplementsform* är talområdet (8 bitar) -128- +127 och således resultatet korrekt. Inget *spill* alltså och därmed sätts V-flaggan till 0. Resultatet är skilt från 0 och därmed sätts Z-flaggan till 0.

b)
```
X = 10111100
Y = 01000100
R = 00000000
V = 0    Z = 1
```
Operationen innebär X=-68, Y= 68, R= 0, (-68+68=0). Eftersom talen betraktas på *tvåkomplementsform* är talområdet (8 bitar) -128- +127 och således resultatet inom talområdet. V-flaggan sätts därför till 0, dessutom blir i detta fall innehållet i registret 0 varför Z-flaggan sätts till 1.

c)
```
X = 10000001
Y = 10000001
R = 00000010
V = 1    Z = 0
```
Operationen innebär X=-127, Y=-127, R= 2 men -127-127 = -254. Eftersom talen betraktas på *tvåkomplementsform* är talområdet (8 bitar) -128- +127 och således resultatet utanför talområdet. V-flaggan sätts därför till 1, resultatet i registret är skilt från 0 varför Z-flaggan sätts till 0. Slutsatsen att V-flaggan sätts till 1 kan man också komma till på följande sätt: Betrakta operanderna X och Y, dessa har samma teckenbit (den mest signifikanta), den är 1, dvs talen tolkas som *negativa*. Å andra sidan, då vi betraktar det 8-bitars resultatet R ser vi att teckenbiten är 0. Eftersom addition av två negativa tal aldrig kan generera ett positivt resultat måste därför *spill* ha uppstått. Detta indikeras genom att V-flaggan sätts till 1.

1.16

a) 126-80 = 46
```
00 00 00 00 00 00 00 00 00
    0  1  1  1  1  1  1  1  0
  - 0  1  0  1  0  0  0  0  0
    0  0  1  0  1  1  1  1  0
```
Dvs: 126-80 = 46, B=0

b) 80-126 = -46
```
~~10~~ ~~10~~ 10 ~~10~~ 10 ~~10~~ ~~10~~ 10 00
    0  ~~1~~  0  ~~1~~  0  0  0  0
  - 0  1  1  1  1  1  1  1  0
    1  1  0  1  0  0  1  0
```
Dvs: 80-126 = -46, B=1

1.17

a) 126-80 = 126+(-80)
```
1  111
   01111110
  +10110000
  ---------
   00101110
```
Dvs: 126-80 = 46, Flaggor: Z=0, N=0, C=1, V=0

b) 80-126 = 80+(-126)
```
0
   01010000
  +10000010
  ---------
   11010010
```
Dvs: 80-126 = -46, Flaggor: Z=0, N=1, C=0, V=0

SLUTSATSER: Flaggsättning sker enligt enkla regler. Observera speciellt att C sätts som ”icke-B” vid en jämförelse med uppgift 16.

1.18

a) 1010 - spill om talen betraktas på tvåkomplementsform
b) 0000 - spill

1.19

a) 0010 - spill
b) 0100 - inget spill

1.20

a) 0010 - spill
b) 1100 - inget spill

1.21
```
        EOR   #$FF,D
        ADD   #1,D
```

1.22
```
        PUL   D
        ADD   (0,S),D
        ST    D,(0,S)
```

1.23
```
        LD    (0,A),D
        ADD   (1,A),D
        RTS
```

1.24
```
        PSHL  A
        PSHH  A
        LD    #0,A
_1      CMP   #0,D
        JEQ   _2
        ADD   (1,S),A
        SUB   #1,D
        JMP   _1
_2      PUL   D
        PUL   D
        RTS
```

1.25 a) Påföljande kod/data placeras med start på adress $4000.
b) Minne reserveras ($32*1=50 bytes).
c) Minne reserveras ($10*2=16*2=32 bytes)

1.26 "DS.B": Adresser $5000-$5003, odefinierat innehåll
"DC.W": $5004=$43, $5005=$20
"DC.L": $5006=$88, $5007=$77, $5008=$66, $5009=$55

1.27 a) D0=87654321
b) D0=XXXX4321
c) D0=XXXXXX21

1.28 a) D0:$XXXXXX33
b) D0:$XXXX2233
c) D0:$66778899
d) A0: $00005006, D0:$XXXXXX66

1.29 a)

$5000	$00
$5001	$11
$5002	$22
$5003	***$AA***
$5004	$44
$5005	$55
$5006	$66
$5007	$77
$5008	$88
$5009	$99

b)

$5000	$00
$5001	$11
$5002	***$BB***
$5003	***$CC***
$5004	$44
$5005	$55
$5006	$66
$5007	$77
$5008	$88
$5009	$99

c)

$5000	$00
$5001	$11
$5002	***$BB***
$5003	***$CC***
$5004	***$DD***
$5005	***$FF***
$5006	$66
$5007	$77
$5008	$88
$5009	$99

d)

$5000	$00
$5001	***$00***
$5002	$22
$5003	$33
$5004	$44
$5005	$55
$5006	$66
$5007	$77
$5008	$88
$5009	$99

e)

$5000	$00
$5001	$11
$5002	$22
$5003	$33
$5004	***$00***
$5005	***$11***
$5006	$66
$5007	$77
$5008	$88
$5009	$99

f)

$5000	$00
$5001	$11
$5002	$22
$5003	$33
$5004	***$00***
$5005	***$11***
$5006	***$22***
$5007	***$33***
$5008	$88
$5009	$99

1.30

```
        MOVE.L      #0,($5000).L  eller
        CLR.L       ($5000).L
```

1.31

```
        MOVE.B      #$21,($FFFFF011).L
```

1.32

```
        MOVE.B      ($FFFFF019).L,D0
        ASR.B       #1,D0
        MOVE.B      D0,($FFFFF011).L
```

1.33

```
        ORG         $4000
        Startadress
program:
        MOVE.B      ($FFFFF019).L,D0
        ASR.B       #1,D0
        MOVE.B      D0,($FFFFF011).L
        BRA         program
```

1.34

```
        ORG         $4000
program:
        MOVE.B      ($FFFFF019).L,D0
        ASL.B       #1,D0
        MOVE.B      D0,($FFFFF011).L
        BRA         program
```

1.35

```
        ORG         $4000
program:
        MOVE.B      ($FFFFF019).L,D0
        ANDI.B      #%11111110,D0
        MOVE.B      D0,($FFFFF011).L
        BRA         program
```

1.36

```
        ORG         $4000
program:
        MOVE.B      ($FFFFF019).L,D0
        ANDI.B      #%01111110,D0
        MOVE.B      D0,($FFFFF011).L
        BRA         program
```

1.37

```
        ORG         $4000
program:
        MOVE.B      ($FFFFF019).L,D0
        ORI.B       #%10000001,D0
        MOVE.B      D0,($FFFFF011).L
        BRA         program
```

1.38

```
        ORG         $4000
program:
        MOVE.B      ($FFFFF019).L,D0
        EORI.B      #%10000001,D0
        MOVE.B      D0,($FFFFF011).L
        BRA         program
```

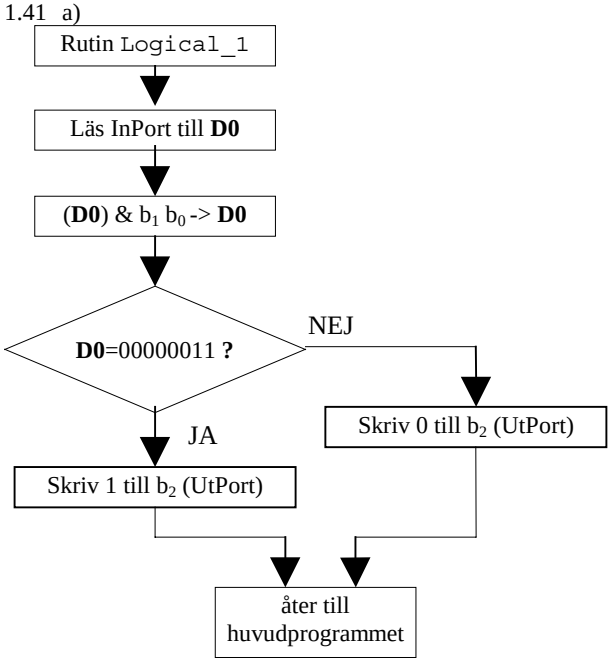
1.39

```
* Subrutin
TestForZero:
        MOVE.B      ($FFFFF019).L,D0
        CMPI.B      #0,D0
        BNE         TestForZero
        RTS
```

1.40

```
* Subrutin
TestInPort:
        MOVE.B      ($FFFFF019).L,D0
        ANDI.B      #%10000000,D0
        CMPI.B      #0,D0
        BNE         TestInPort
        RTS
```

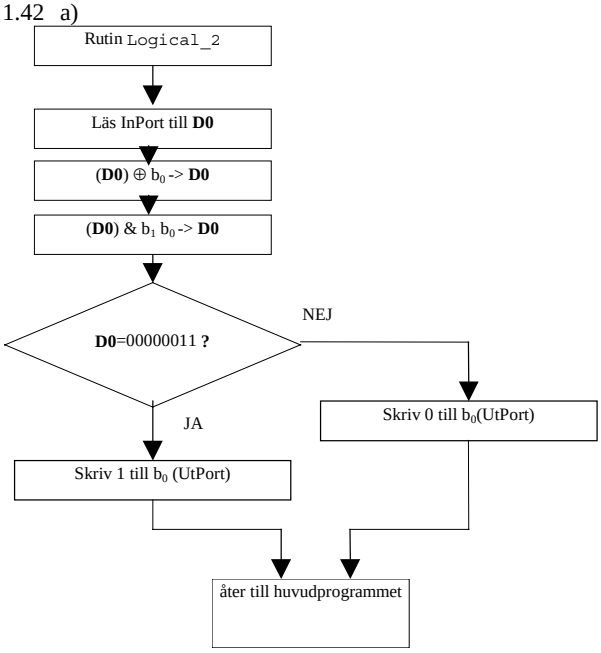
1.41 a)



b)

```
* Subrutin
Logical_1:
        MOVE.B      (InPort).L,D0
        ANDI.B      #3,D0
        CMPI.B      #3,D0
        BNE         Logical_12
        MOVE.B  #%00000100,(OutPort).L
        BRA         Logical_13
Logical_12:
        MOVE.B  #%00000000,(OutPort).L
Logical_13:
RTS
```

1.42 a)

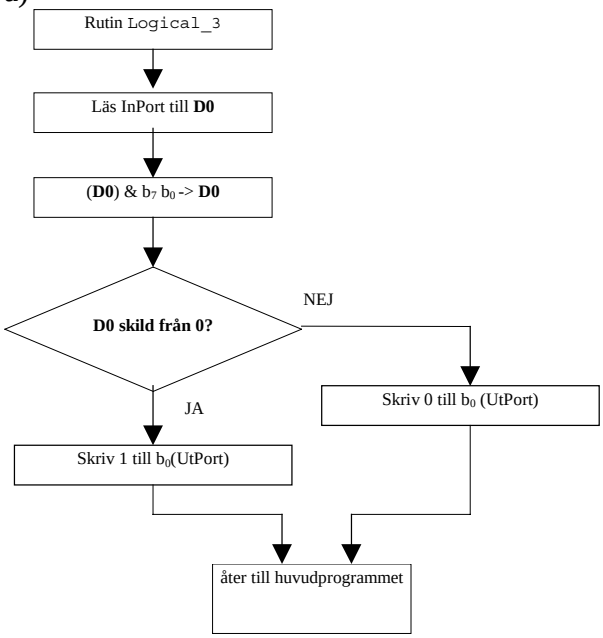


b)

```
* Subrutin
Logical_2:
        MOVE.B      (InPort).L,D0
        EORI.B      #1,D0
        ANDI.B      #3,D0
        CMPI.B      #3,D0
        BEQ         Logical_21
        MOVE.B      #1,(OutPort).L
        BRA         Logical_22
Logical_21:
        MOVE.B      #0,(OutPort).L
Logical_22:
        RTS
```

1.43

a)

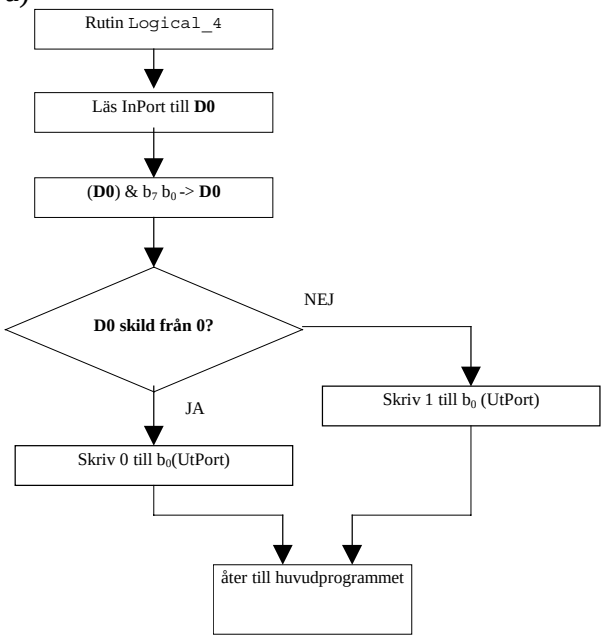


b)

```
* Subrutin
Logical_3:
       MOVE.B        (InPort).L,D0
       ANDI.B        #%10000001,D0
CMPI.B        #0,D0
       BEQ           Logical_31
       MOVE.B        #1,(OutPort).L
       BRA           Logical_32
Logical_31:
       MOVE.B        #0,(OutPort).L
Logical_32:
       RTS
```

1.44

a)



b)

```
* Subrutin
Logical_4:
       MOVE.B        (InPort).L,D0
       ANDI.B        #%10000001,D0
       CMPI.B        #0,D0
       BNE           Logical_41
       MOVE.B        #1,(OutPort).L
       BRA           Logical_42
Logical_41:
       MOVE.B        #0,(OutPort).L
Logical_42:
       RTS
```

1.45

```
InPort        EQU    $FFFFF019
OutPort       EQU    $FFFFF011
              ORG    $4000
program: MOVE.B      (InPort).L,D0
       ANDI.B        #1,D0
       CMPI.B        #1,D0
       BEQ           program2
       MOVE.B        #0,(OutPort).L
       BRA           program
program2:
       MOVE.B        #$80,(OutPort).L
       BRA           program
```

```
* Subrutin GetText
GetText:      MOVEA.L       #$1000,A0
GetText_1:    MOVE.B        ($A000).L,D0
              BEQ           GetText_2
              MOVE.B        D0,(A0)
              ADDA.L        #1,A0
              BRA           GetText_1
GetText_2:    MOVE.B        #0,(A0)
              RTS
```

1.46

```
* Subrutin GetText (2)
GetText:      MOVEA.L       #$1000,A0

GetText_0:
              MOVE.B        ($A001).L,D0
              ANDI.B        #1,D0
              BEQ           GetText_0

GetText_1:    MOVE.B        ($A000).L,D0
              BEQ           GetText_2
              MOVE.B        D0,(A0)
              ADDA.L        #1,A0
              BRA           GetText_1
GetText_2:    MOVE.B        #0,(A0)
              RTS
```

1.47 t = L/c = 2*(1500 * 10^3)/(3*10^8) = 10 ms.

1.48 t = (ant bitar)/f = 1500/(10 10^3) = 150 ms.

1.49 Asynkron överföring: Olika klockor hos sändare och mottagare.
Synkron överföring: Samma klocka hos sändare och mottagare.

1.50 24/2 = 12 Volt

1.51 Upplösningen blir 16/256 = 0,0625 Volt.
4,6/0,0625 = 73,6 men endast heltal kan representeras.
73*0,0625 = 4,5625 Volt.
74*0,0625 = 4,625 Volt.
Principen med succesiv approximation ger att 4,5625 Volt approximerar den mätta spänningen. Felet blir 4,6-4,5625 = 0,0375 Volt.

1.52 För att få 15 omslagsspänningar (E0-E14) krävs 15 operationsförstärkare. Antalet resistorer blir då 16. Eftersom alla resistanser är lika ligger då 1 Volt över varje resistans och vi får omslagsspänningarna 15,14,13,12,...,1 Volt.