Datorteknik för högskolans ingenjörsutbildningar

Datorteknik för högskolans ingenjörsutbildningar

ISBN 91-89280-03-2

Datorteknik för högskolans ingenjörsutbildningar
Art Nr: 110-00
©1993-2000 Roger Johansson och Rolf Snedsbøl
Nionde utgåvan.
Tredje tryckningen.

Kopieringsförbud

Detta verk är skyddat av upphovsrättslagen.

Kopiering är förbjuden utöver vad som anges i avtalet om kopiering i högskolorna (UFB 4).

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@gbgmv.se
Internet: http://www.gbgmv.se

Bildleverantörer i alfabetisk ordning:
Apple Computer, Digital Equipment, GMV, Hewlett-Packard, IBM

Förord

Detta läromedel är avsett att användas vid tekniska ingenjörsutbildningar på högskolenivå såväl som på vissa gymnasieprogram och inom kommunal vuxenutbildning.

Avsikten är att läromedlet ska utgöra ett *basmaterial* i ämnet Datorteknik för högskolans ingenjörsutbildningar. Som sådant bör det också fungera bra inom Kommunal Vuxenutbildning i program med inriktning mot Datorteknik/Informationsteknologi På gymnasienivå kan läromedlet lämpligen användas för fördjupning i ämnet *mikroprocessorteknik* (elprogrammet).

Läroboken revideras kontinuerligt. I denna version (9) har huvudsakligen typografiska ändringar gjorts sedan version 7.

I läromedlet ingår också en arbetsbok för laborationer med mikrodatorer *MC68*/*MD68k* och laborationskortet *ML4* avsedd för inledande laborationsövningar och en arbetsbok för laborationer med *ML5, ML15 och ML23* för en påföljande laborationsserie, dessutom finns *Datorteknik, övningsbok* innehållande ett stort antal övningsuppgifter, facit till uppgifterna och en instruktionslista. Alla dessa häften anknyter direkt till läroboken.

Till läromedlet finns underlag för stordia (Microsoft Power Point format), se GMV's Internetadress.

Göteborg i januari 2000, Roger Johansson och Rolf Snedsbøl

Inför den andra tryckningen i augusti 2000 har en del typografiska fel korrigerats.

Innehåll

Introduktion

1.1 Olika datortyper 10
1.2 Talsystem 14
1.2.1 Talomvandling 17
1.3 Program och programmering 20
1.3.1 Algoritmer 22
1.4 Sammanfattning 23

Mikrodatorn

2.1 Mikrodatorns olika block 25
2.1.1 Mikrodatorsystemets bussar 26
2.1.2 Mikrodatorsystemets minnen 29
2.1.3 Primärminnets uppbyggnad 31
2.1.4 Mikroprocessorns uppbyggnad 33
2.1.5 Mikroprocessorns arbetssätt. 34
2.1.6 Maskininstruktionen 35
2.1.7 Användning av bussarna 40
2.2 In- och utenheter 46
2.3 Sammanfattning 48

Datoraritmetik

3.1 Binära koder 51
3.1.1 Det binära talsystemet 51
3.1.2 Binärkodade decimaltal 53
3.1.3 Alfanumerisk kod 54
3.2 Tecken-Beloppsrepresentation 55
3.3 Tvåkomplementsrepresentation 57
3.4 Implementering 63
3.4.1 Lång addition 63
3.4.2 Tvåkomplementering 64
3.4.3 Subtraktion 66
3.4.4 Vänsterskift (multiplikation med 2) 68
3.4.5 Högerskift (division med 2) 69
3.4.6 Multiplikation 71
3.4.7 Division 74
3.5 Förskjuten binärkod 77
3.6 Flyttal 80
3.6.1 IEEE - flyttalsstandard 83
3.6.2 Aritmetiska operationer på flyttal 84
3.6.3 Avrundningsmetoder 86
3.6.4 Flyttalet noll 86
3.6.5 Denormaliserade tal 86
3.6.6 Oändligheter 86

3.6.7 Icke-tolkningsbara resultat 86
3.6.8 Flyttalsmultiplikation och Division 87
3.6.9 Flyttalstest och jämförelser 87
3.6.10 Sammanfattning IEEE-flyttalsstandard 88

Mikroprocessorn

4.1 MC68000s interna block 93
4.2 MC68000:s arkitektur 94
4.2.1 Programmerarens modell (bild) 94
4.2.2 Datatyper 96
4.2.3 MC68000, instruktionsformat 98
4.3 Processorns arbetssätt 109
4.3.1 En busscykel - läscykel. 110
4.3.2 Subrutiner och Stack. 113
4.3.3 Extern styrning 120
4.4 MC68000:s anslutningar 121
4.4.1 Ett MC68000 minimisystem 124
4.5 Sammanfattning 127

Assemblerprogrammering

5.1 Introduktion till assemblerprogrammering 131
5.2 MC68000 - programmerarens bild 140
5.2.1 Supervisor/User Mode 140
5.2.2 Registermodell 140
5.2.3 Användning av register 144
5.2.4 Instruktionsgrupper 147
5.2.5 Adresseringssätt 152
5.3 Programmering i assembler 162
5.3.1 Inledning 162
5.3.2 Specifikation och formell beskrivning 163
5.3.3 Tilldelningar 165
5.3.4 Operatorer 168
5.3.5 MC68000 flaggsättning vid aritmetiska operationer 168
5.3.6 Aritmetiska operatorer 170
5.3.7 Logiska instruktioner 179
5.3.8 Skiftoperatorer 181
5.3.9 Jämförelser och test 185
5.3.10 Bitoperationer 192
5.3.11 If - konstruktioner 192
5.3.12 If/else konstruktioner 194
5.3.13 While-konstruktioner 196
5.3.14 Subrutiner (funktioner) 197
5.3.15 Parameteröverföring 199
5.3.16 Positionsoberoende kod 207
5.3.17 Exception hantering 208
5.4 Kombinerad programmering 218

5.4.1 X68c - korskompilator ... 219
5.4.2 Minnesdisposition ... 219
5.4.3 X68c, minnesallokering ... 222
5.4.4 Lokala variabler, minnesallokering ... 224
5.4.5 Parameteröverföring ... 227
5.4.6 Maskinnära programmering i C ... 230

Input Output och adressavkodning

6.1 Adressavkodning ... 233
6.1.1 Ofullständig adressavkodning ... 236
6.2 Parallell in- och utmatning ... 240
6.2.1 Skrivarport med register. ... 241
6.2.2 Skrivarport med register och READY-signal. ... 243
6.2.3 Skrivarport med register DAV och STROBE-signal. ... 245
6.2.4 Skrivarport med register READY, DAV och STROBE-signal. ... 246
6.2.5 Avslutande resonemang kring printerporten. ... 248
6.3 Seriell I/O ... 252
6.3.1 Synkron överföring ... 257
6.3.2 Asynkron överföring ... 258
6.4 Datanät ... 264
6.5 Sammanfattning ... 268

Avbrott

7.1 Generellt avbrott, en avbrottskälla. ... 272
7.1.1 System med flera avbrottskällor ... 277
7.2 Vektoravbrott ... 280
7.3 Avbrott med MC68000 ... 283
7.3.1 MC68000 externa avbrottsingångar. ... 286
7.3.2 Avbrottsprioriteter ... 288
7.3.3 Vektoravbrott med MC68000. ... 290
7.3.4 Autovector-avbrott med MC68000 ... 294
7.3.5 Buss Error ... 296
7.3.6 MC68000 exception vector table ... 298

Periferikretsar

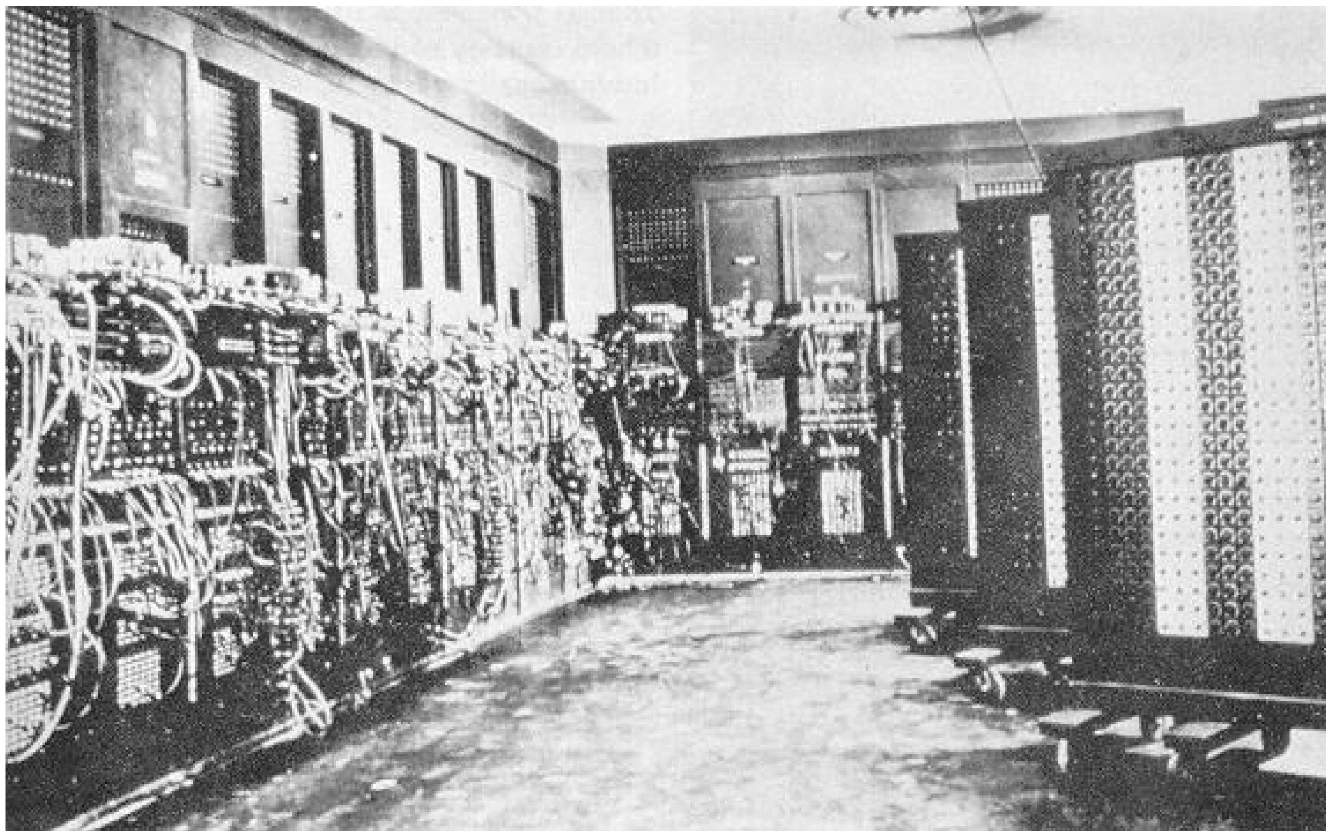
8.1 Varför används periferikretsar ? ... 301
8.1.1 En generell periferikrets. ... 302
8.1.2 En skrivaranslutning. ... 303
8.2 PI/T - MC68230 ... 306
8.2.1 Programmering av PI/T:ns portar ... 308
8.2.2 Registeruppsättning för PI/T:n portar. ... 313
8.2.3 Sammanfattning över PI/T:ns port-register ... 319
8.2.4 Vektoravbrott och PI/T:ns portar ... 320
8.3 PI/T:ns räknare ... 323
8.3.1 Räknarens registermodell ... 324

8.3.2 PI/T:n som realtidsklocka. 327
8.4 Sammanfattning av MC68230 PI/T 331
8.5 DUART - MC68681 332
8.5.1 Seriekommunikation och RS232 332
8.5.2 Översikt över DUART MC68681 334
8.5.3 Enkel användning av DUART 338
8.5.4 Beskrivning av kretsens register 346
8.5.5 Vektoravbrott med DUART. 357
8.6 Sammanfattning 361

Mikrodatorn MC68 och MC68340

9.1 Mikrodatorn MC68 363
9.1.1 Blockdiagram 363
9.2 Minnessystem 365
9.3 MC68's bussar. 369
9.3.1 I/O Buss 369
9.3.2 Expansionsbuss 370
9.4 MC68340 371
9.4.1 System Integration Module (SIM40) 374
9.4.2 SIM40 Registerbeskrivning 380
9.4.3 CPU32 388
9.4.4 I/O-block 394
9.4.5 Seriemodul 395
9.4.6 Räknarmodul 397
9.5 Sammanfattning 404

INTRODUKTION



ENIAC, DEN FÖRSTA ELEKTRONISKA DATORN I DECEMBER 1945

Användning av datorer har under de senaste decennierna kommit att få fler och fler tillämpningsområden. Som följd av detta har också fler och fler människor kommit att påverkas av datorer på ett eller annat sätt. Datorn har utvecklats från en enkel räknemaskin till ett modernt och snabbt arbetsredskap för ett stort antal yrkeskategorier. Ofta har detta varit av godo, i bland av ondo, men oavsett vad man tycker om datorernas intrång erbjuder tekniken såväl utmaningar som möjligheter. Denna lärobok har som syfte att introducera dig till denna teknik.

I detta första kapitel introduceras en rad grundläggande datortekniska begrepp.

Ordet "dator" är en härledning av latinets "datum"

Datorer finns i praktiskt taget alla hem. Det är numera vanligt med så kallade *hemdatorer* avsedda för såväl nytta som nöje. Nästan alla har vi också, i bland utan att ens tänka på det, stött på datorer eller rättare sagt *mikrodatorer* i någon annan form, exempelvis i en mikrovågsugn, en tvättmaskin eller en videobandspelare. Några av oss har säkerligen 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 mikrovågsugn
- 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:

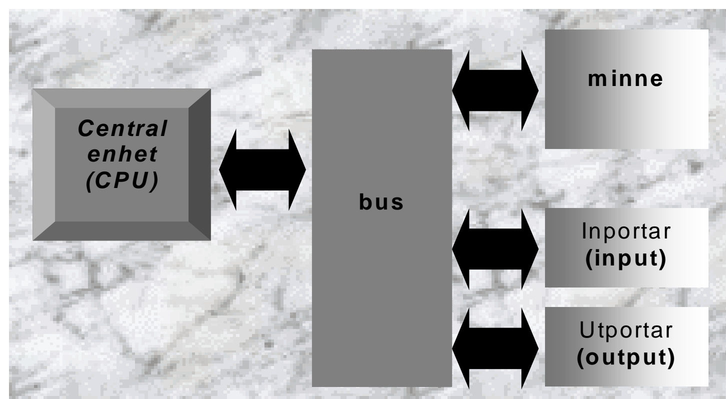
- bildskärmen eller högtalarna till en persondator
- lampor och lysdioder
- sifferpaneler

SONY "PLAYSTATION" SPELDATOR

PERSONDATOR FRÅN HEWLETT-PACKARD (1999)

En dator kan 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 styra mekaniken i exempelvis en videobandspelaren med en sekvens av instruktioner som följd av ett kommando den tagit emot från någon inenhet (*inport*). Enheten som tolkar och utför kommandona kallas

centralenhet (CPU: *Central Processing Unit*) eller vanligtvis helt enkelt *processor*. Utöver CPU'n behövs också ett *minne* där styrinformation (instruktionerna) och data kan lagras. Enheterna är sammankopplade med signalledningar som används för att överföra styrinformation och data mellan CPU, minne och portar. Flera signalledningar med liknande funktion brukar gemensamt kallas *buss.* Vanligen talar man om tre olika bussar, nämligen *styrbuss*, *databuss* och *adressbuss*.



EN DATOR'S PRINCIPIELLA UPPBYGGNAD

KONTROLLPANEL PÅ MIKROVÅGSUGN MED SIFFERPANEL (FÖR UTMATNING), KNAPPAR OCH VRED (FÖR INMATNING)

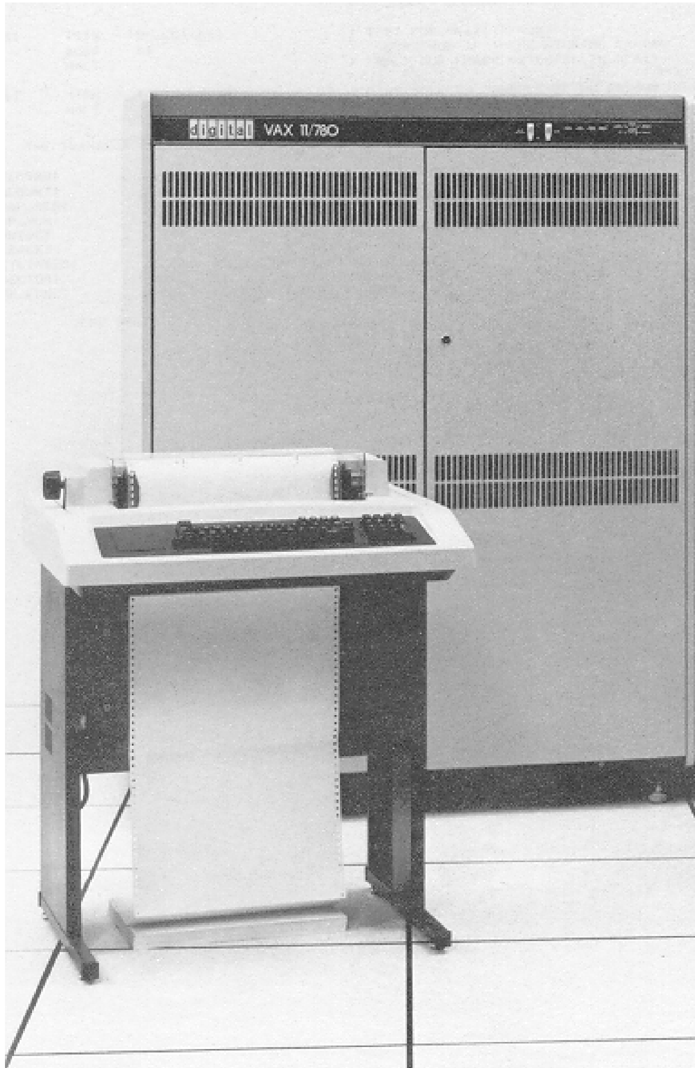
Hur centralenheten ska uppföra sig för ett bestämt kommando avgörs av det *program* som finns lagrat i datorns minne. Ett program kan sägas vara en sekvens av *instruktioner* som centralenheten utför för att åstadkomma en önskad funktion.

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.

1.1 Olika datortyper

Datorsystem klassificeras ofta med begreppen *stordator*, *minidator* och *mikrodator*. Det *principiella arbetssättet* för dessa olika datortyper är vanligtvis det samma, det som skiljer dem åt är mängden av data de kan bearbeta på en viss tid. Kapaciteten (prestanda) bestäms av den arbetstakt de har (*klockfrekvensen*), hur stora program de kan arbeta med och hur många *hårdvarukretsar* som arbetar samtidigt i centralenheten. Det är vanligt att ange en dators prestanda efter hur snabbt instruktionsutförandet sker. Enheten kallas MIPS (*Million Instructions Per Second*) och anger hur många miljoner instruktioner som datorn förmår utföra under en sekund. Även om detta prestandamått är diskutabelt kan det vara intressant vid historiska jämförelser.

En *stordator* används i dag för att utföra omfattande beräkningar och simuleringar, exempelvis simulering av luftströmmarna kring ett flygplan, beräkningar för att ge en detaljerad väderprognos etc. Stordatorer används också i bankcentraler där de bearbetar mycket stora mängder data. Flera tusen *terminaler* eller *arbetsstationer* (se nedan) kan vara anslutna till en stordator.

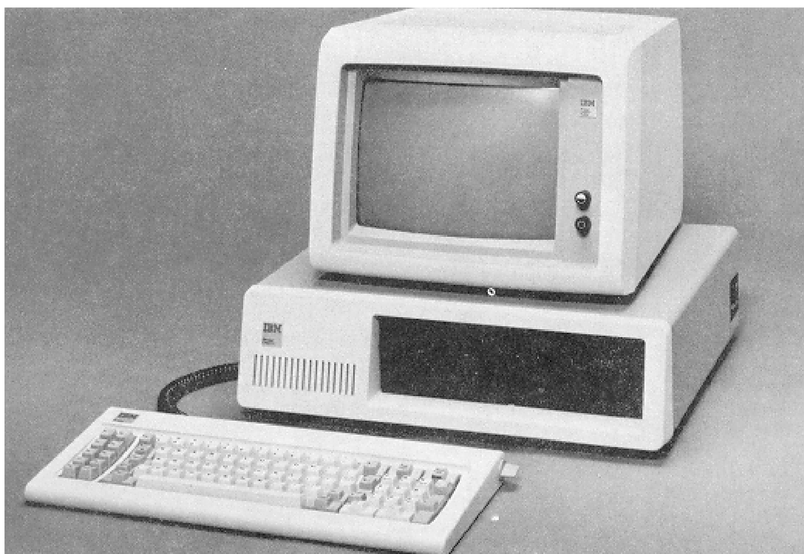


Minidator VAX 780.

Minidatorer kan också användas i banksammanhang, men då oftast lokalt, exempelvis på ett bankkontor, med 8-64 terminaler anslutna. Som regel finns det en förbindelse mellan huvuddatorn (en stordator på centralbanken) och minidatorn. Ett annat användningsområde för en minidator kan exempelvis vara att styra ett antal robotar vid det löpande bandet i en fabrik.

Gruppen *mikrodatorer* kan indelas i generella mikrodatorer och mikrodatorer för specialtillämpningar. De generella mikrodatorerna kan vara av typ arbetsstationer, persondatorer eller hemdatorer. Dessa började utvecklas först under senare delen av 1970-talet. De består av systemenhet med processor och arbetsminne, tangentbord och bildskärm (som ibland är en vanlig TV). De var främst avsedda för enkla hemtillämpningar som ordbehandling, bokföring och registerhantering samt kanske framförallt spel. Programmen lagrades i första hand på massminnen av typ vanliga ljudkassetter.

Persondatorn fick sitt första stora genombrott när **IBM** presenterade sin **PC** (*Personal Computer*) år 1981. Den bestod i princip av en förbättrad hemdator med massminnen av typ flexskivor (disketter) och en egen bildskärm. Ganska snart startade flera mindre datorföretag tillverkning av PC-kopior med lägre pris och ofta bättre prestanda än orginalet.



DEN FÖRSTA IBM-PC'N.

Eftersom PC'n och de så kallade "IBM PC kompatibla" kopiorna använde samma processor och samma systemprogramvara, *PC-Disc Operating System* (PC-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* som kunde lagra mycket program och data med kort åtkomsttid. PC'n är en typisk enanvändardator som började som en enkel maskin med kapacitet att utföra endast ett program åt gången (PC-DOS) Den har dock utvecklats mycket snabbt, via nya systemprogramvaror (*Microsoft Windows*) och med prestandaökning från c:a 0,5 MIPS till flera hundra MIPS, till en kraftfull fleranvändarmaskin för krävande uppgifter.

Utvecklingen avspeglas inte enbart prestandamässigt. PC'n har också blivit mindre och speciella varianter, s.k "laptops" och "note-books" utgör nu exempel på enkla bärbara datorer med pretanda som vida överglänser den första PC'n.

NOTEBOOK-DATOR MED OPERATIVSYSTEM WINDOWS CE

En annan viktig gren på persondatorträdet skapades när **Apple** introducerade **Macintosh** år 1984. Den viktigaste nyheten med “Mac’en” var den användarvänliga "fönsterhanteringen" med menyer och användning av pekanordning som kallas för mus. Eftersom Mac’en använder en annan processor än IBM PC och konstruktionen i övrigt är helt annorlunda kan program inte enkelt flyttas mellan de två typerna av persondatorer.

APPLE MAC-INTOSCH CLASSIC

IBOOK – BÄRBAR PERSONDATOR FRÅN APPLE

På grund av sin användarvänlighet blev Mac’en snabbt populär i brancher där persondatorer nu började göra sitt intåg. Men trots att den genomgått samma snabba tekniska utveckling och, inte minst, vad gäller utförande alltid befunnits en bra bit framför PC-kompatiblerna utgör andelen persondatorer av typen Macintosh i dag endast en bråkdel av det totala beståndet.

APPLE MACINTOSH G3

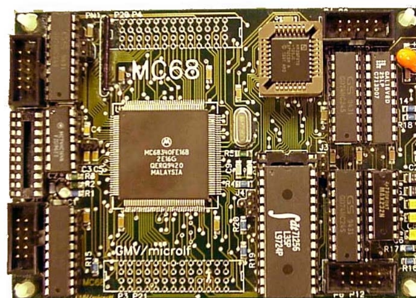
BILDSKÄRM – MODERNT SNITT, SIGNERAD APPLE

IMAC – PERSONDATOR FRÅN APPLE MED ORIGINELL OCH SPÄNNANDE DESIGN

Arbetsstationer kallas de mest avancerade och kraftfulla generella mikrodatorerna. De har närmast sitt ursprung i gårdagens minidatorer och uppstod genom att miniatyriseringen av datorelektoniken gjorde det möjligt att placera all elektronik och övriga komponenter i en liten systemenhet, liknande hem- och persondatorns. Systemprogramvaran och övriga program som utförs på arbetsstationer är normalt hämtad från minidatorvärlden och är därför betydligt dyrare än PC-programvaran. Prestandamässigt har skillnaden mellan högpresterande PC och vanliga arbetsstationer nästan försvunnit. En arbetsstation används oftast av en eller ett fåtal användare.

SUN SPARC 20 ARBETSSTATION

Mikrodatorer för specialtillämpningar är ofta inbyggda i utrustning exempelvis för videospel, tvättmaskiner, videobandspelare, förarlösa truckar, navigations-utrustning, industrirobotar, bilar, etc. Kompletta mikrodatorer kan i dag byggas i en enda kapsel vilket gör dessa väl lämpade för exempelvis styrsystem vilka har krav på liten storlek och låg strömförbrukning.



MIKRODATOR MC68 ÄR AVSEDD FÖR GENERELLA STYRTILLÄMPNINGAR

1.2 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", just tio symboler används förmodligen 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 fortsättningen behandlar vi enbart talsystem som är positionssystem.

Man säger att det decimala talsystemet har *basen* tio eftersom det använder tio olika siffersymboler (0 - 9). Eftersom basen är det antal siffersymboler som används är basen 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 och har blivit mycket viktigt i tekniska sammanhang.

Låt oss illustrera hur man kodar ett talvärde i ett positionssystem. Eftersom vi behärskar det decimala talsystemet, är det lämpligt att använda detta.

EXEMPEL

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

fyran är värd fyra hundratal	(4 . 102)
trean är värd tre tiotal	(3 . 101)
femman är värd fem ental	(5 . 100)
sjuan är värd sju tiondelar	(7 . 10-1)
tvåan är värd två hundradelar	(2 . 10-2)

Talet kan uttryckas som summan

4 . 102 + 3 . 101 + 5 . 100 + 7 . 10-1 + 2 . 10-2

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

1.1

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

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

Koefficienterna d_k ($k = -m, ..., n-1$) representerar siffror i talsystemet. Antalet heltalssiffror är n och antalet bråktalssiffror är m. På samma sätt som för det decimala talsystemet kodar man ett talvärde N i ett godtyckligt talsystem med sifferföljden

$$N_r = d_{n-1}d_{n-2}...d_0.d_{-1}...d_{-m}$$

Sifferföljden tolkas som summan ovan 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. least 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 sjutton första heltalen i några olika talsystem visas. Vi kommer senare att speciellt behandla de binära och hexadecimala talsystemen.

Binärt	Hexadecimalt	Decimalt	Oktalt
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

Tabell 1.1 Talvärden för tal i olika positionssystem

Då 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 parantes och ange talsystemets bas som index till parentesen.

EXEMPEL

Skrivsätt för att ange tal i olika talsystem

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

1.2

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 noll markerar ett oktalt tal (bas 8)
`2934`, att första siffran är skild från 0 (noll) markerar ett decimalt tal (bas 10)
`0x3AF5`, prefixet “0x” (noll-x) markerar ett hexadecimalt tal (bas 16)

1.3

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 (bas 10)
`$3AF5` Prefixet $ markerar ett hexadecimalt tal (bas 16)

1.4

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 de följande avsnitten kommer flera olika kodningssätt att behandlas.

1.2.1 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 företrädesvis *heltal* (Vi återkommer till bråktal i kapitel 3). Ekv 1.1 får då formen

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

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

$(N)_{10} = 1*2^5 + 0*2^4+1*2^3+1*2^2+1*2^1+0*2^0 =$
$32+ 0 + 8+ 4+ 2+ 0 = 46$

DVS $(101110)_2 = (46)_{10}$

1.5

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

1.6

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:

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

Högerledet består av en kvot Q_0 och en rest R_0 där

$$Q_0 = d_{n-1} * r^{n-2} + d_{n-2} * r^{n-3} + ... + d_1 \qquad R_0 = \frac{d_0}{r}$$

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

1.7

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

1.8

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

Tabell 1.2 De 16 olika hexadecimala siffrorna

EXEMPEL

Omvandla binärtalet $(1011001011)_2$ till ett 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 2.2 finner vi nu de hexadecimala motsvarigheterna:

2 C B

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

1.9

EXEMPEL

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

Varje hexadecimal siffra ger direkt fyra binära siffror

A	8
1010	1000

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

1.10

Denna typ av omvandlingar återkommer du ständigt till då du arbetar med datorteknik på maskinnära nivå. Du bör därför lära dig att utföra dessa omvandlingar utantill.

1.3 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. Instruktionerna, eller mer egentligt *maskininstruktionerna* består av en sekvens nollor och ettor (bitsträng). Varje unik sekvens har också en unik betydelse, dvs CPU'n kan entydigt bestämma vilken instruktion som skall utföras baserat på vilken bitsträng som behandlas.

EXEMPEL

Bitsträngen:

```
0100 1110 0111 0001
```

motsvarar instruktionen `NOP` (*No Operation*) hos mikroprocessorn Motorola MC68000

1.10

Översättningen från högnivåprogram (C-program) till maskinprogram utförs ofta i två steg, *kompilering* och *assemblering*. Som nybörjarprogrammerare märker man sällan dessa steg då man vanligtvis trycker på en eller ett par funktionstangenter eller kanske "klickar med musen" för att kompilera, assemblera och starta programmet. Figur 1.2 visar exempel på en sekvens ur ett C-program.

```
...
  sum = sum + amount;
  tax = sum * 0.25;
  total = sum + tax;
```

Som vi ser i figur 1.3 består maskinprogrammet endast av nollor och ettor. Varje rad visar en maskininstruktion, observera att de kan ha olika längd oftast beroende på hur komplexa operationer som skall utföras av processorn.

```
00110011111111000001001000110100000000000010101000011001000
1101000010000010
00000000000000110000000000000101
010011101111110010000000000011000000000000001000
0100111001110000
```

FIGUR 1.3 EXEMPEL PÅ ETT MASKINPROGRAM MED 5 MASKININSTRUKTIONER.

Av figur 1.3 framgår det också klart att maskinprogram är praktiskt taget oläsliga för oss vanliga människor. Vi kan normalt inte säga vad varje enskild nolla eller etta har för mening i maskinprogrammet. Man har därför hittat på ett annat skrivsätt för maskinprogram som är mera anpassat för oss människor. Detta skrivsättet kallas för *assemblerspråk* och innebär att man skriver varje maskininstruktion som en bokstavsförkortning. Förkortningen är lätt att tyda för den som har lärt sig det aktuella assemblerspråket. Ett exempel på assemblerspråk visas i figur 1.4.

```
MOVE.L     (amount).L,D0
ADD.L      D0,(sum).L
MOVE.L     (sum).L,D0
DIVS.L     #4,D0
MOVE.L     D0,(tax).L
ADD.L      (sum).L,D0
MOVE.L     D0,(total).L
```

FIGUR 1.4 EXEMPEL PÅ ASSEMBLERPROGRAM.

Kompilatorn översätter högnivåspråk till assemblerkod.

Assemblatorn översätter assemblerkod till maskinkod.

Såväl kompilator som assemblator är specifik för den använda datorn.

När du ändrar ditt C program måste det först översättas till maskinprogram anpassat för den dator du för närvarande använer. Om du försöker flytta enbart maskinprogrammen och köra dessa på "fel" dator skulle det inte fungera.

När programmet kompileras, genereras först ett *assemblerprogram*. Detta måste översättas (*assembleras*) till ett maskinprogram för att kunna tolkas av processorn. Ett assemblerprogram är alltså läsbart för oss människor, fast det kan verka lite kryptisk i början. En rad i ett assemblerprogram motsvarar (som regel) en maskininstruktion (en rad) i exemplet ovan. Observera att assemblerprogram precis som maskinprogram, är olika för olika processorer. Varför då programmera i assembler när det finns högnivåspråk? Jo, vissa processor- och systemberoende operationer *kan* inte programmeras i något högnivåspråk. En assembler-programmerare har en mycket större programmeringsfrihet än en högnivåprogrammerare har. Däremot krävs

att han har full kontroll på variabler och data. Tidigare hette det att man kunde få mycket snabbare maskinprogram då tidigare kompilatorer genererade en mängd onödiga (maskininstruktioner) assemblerinstruktioner. Dagens kompilatorer genererar mycket bra kod, så detta skäl att programmera i assembler är föråldrat.

1.3.1 Algoritmer

Den sekvens av direktiv vi anger för att programmera videobandspelaren för inspelning kallas för *program.* För att beskriva den uppgift programmet ska utföra använder vi text och eventuellt figurer med symboler. En sådan beskrivning kallas en *algoritm***.** Algoritmen beskriver alltså vad programmet gör. Samma algoritm kan som regel användas oberoende av vilken typ av videobandspelare vi använder.

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

Detta är en förhållandevis "grov" algoritm dvs en algoritm på hög nivå. Om vi exempelvis inte tidigare använt denna videobandspelare behöver vi förmodligen 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 skriva algoritmen på en hög nivå först för att senare förfina den i flera steg. Eftersom algoritmen förfinas allt eftersom fler detaljer infogas blir det 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. Här finns olika standarder för hur ett sådant skall se ut med speciella symboler som anger att data skall hämtas in eller skrivas ut, om valmöligheter önskas osv.

För att algoritmen skall kunna användas måste den först översättas till en sekvens av kommandon (*styrinformation*) som videobandspelaren kan förstå. Vi har därefter *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.

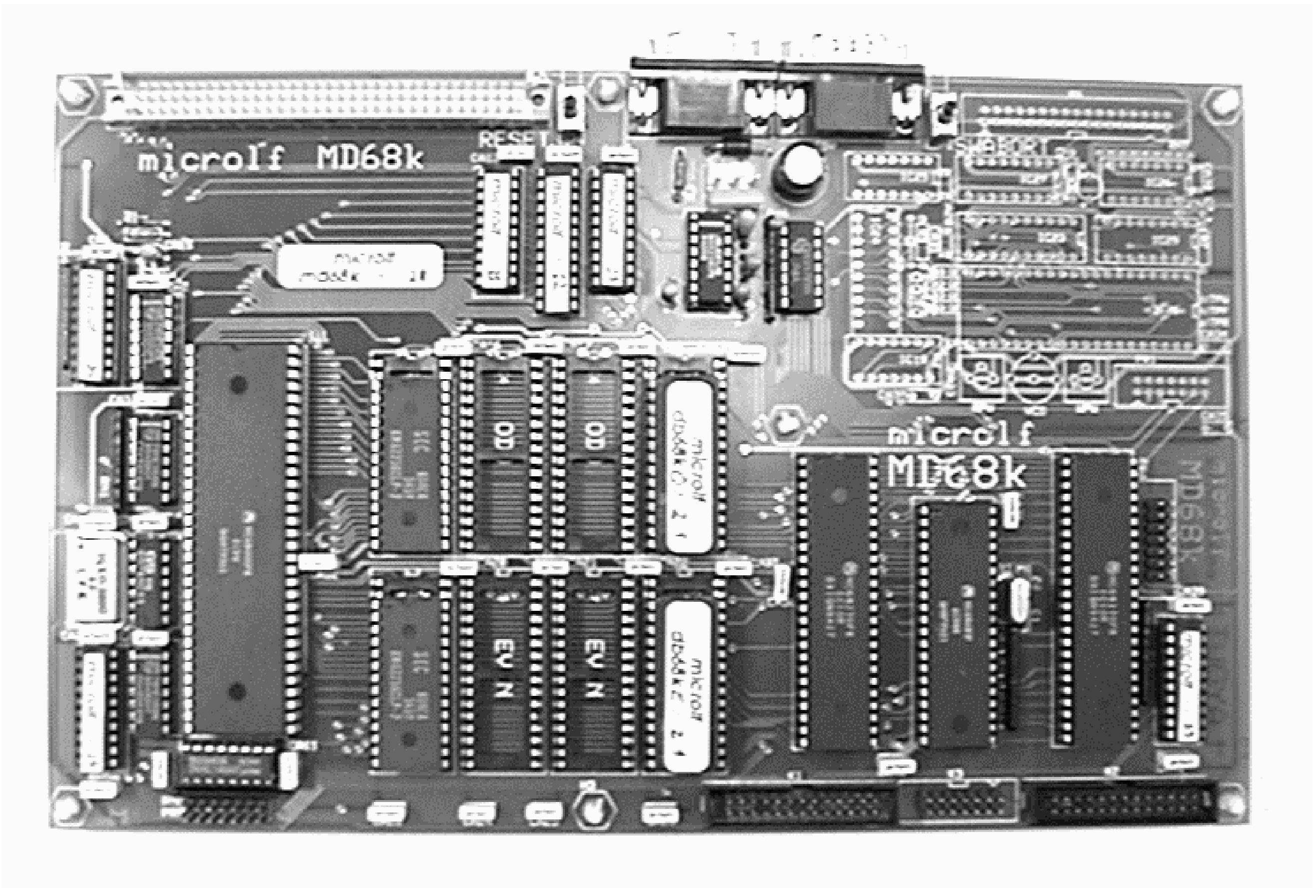
Centralenheten tolkar och utför även felaktiga kommandon och vi får då antagligen ett oönskat resultat. Tänk om vi anger “datum” då videobandspelaren förväntar sig “TV-kanal”....

Detta kursmaterial skall bland annat ge dig kunskaper om hur man programmerar i assembler och peka på varför olika instruktioner ser ut som de gör. Vidare beskriver kursmaterialet en dators uppbyggnad och funktionssätt. Förutom programmet (*mjukvaran*) krävs en mängd fysiska komponenter (*hårdvara)*. Vi behandlar bland annat begrepp som minne, klocka, operativsystem, hårddisk, chip, data, en massa nollor och ettor, filsystem, I/O, ALU, COprocessor, bildskärm, Ethernet-controller, tangentbord, transistorer, register, diskett, mm mm. Alltså, en uppsjö av olika begrepp och termer kan "rymmas" i datorbegreppet beroende på vem vi samtalar med. Vi kommer att redogöra för en mängd sådana termer som ingår i begreppet "dator" i denna lärobok.

1.4 Sammanfattning

- De grundläggande enheterna i ett datorsystem är: *centralenhet* (CPU), *in-/ut-portar* och *bussar*
- Talvärden kan uttryckas i flera olika talsystem, för människan är talsystemet med basen tio (det decimala) naturligt. I andra sammanhang är andra talsystem bättre lämpade och det är därför viktigt att man snabbt kan omvandla talvärden mellan de olika talsystemen.
- En CPU utför sekvenser av kommandon som kallas *program*
- En beskrivning av vad som utförs under ett program kallas *algoritm*
- Ett program skrivet i ett *högnivåspråk* beror oftast ej av den använda CPU:n
- Ett *assemblerprogram* uttrycker maskininstruktioner på en läslig form

MIKRODATORN

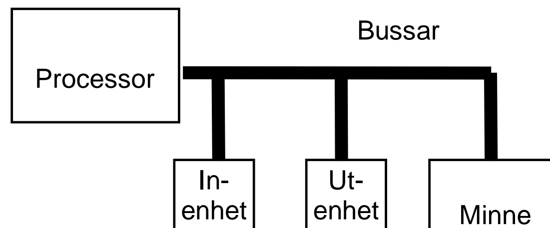


KRAFTFULLA MIKRODATORER KAN I DAG ENKELT KONSTRUERAS PÅ ETT KRETSKORT. PÅ KORTET FINNS DÅ SÅVÄL CPU SOM IN-/ UT- MATNINGSENHETER OCH PRIMÄRMINNE

I detta kapitel studerar vi de byggstenar (block) som bygger upp en dator. Vi ger en introduktion till mikroprocessorn Motorola MC68000. Vi beskriver mikroprocessorns arbetssätt. Slutligen ger vi exempel på några enkla maskininstruktioner och hur dessa utförs.

2.1 Mikrodatorns olika block

En mikrodator består av minst fem principiellt olika delar. Dessa är: processor, inenhet, utenhet, minne och bussar. Bussarnas uppgift är att koppla samman de övriga delarna (se figur 2.2).



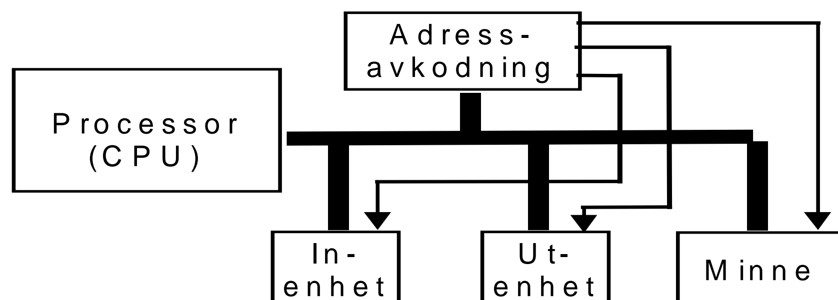
FIGUR 2.2 PROCESSOR MED MINNE, IN- OCH UT-ENHETER.

Maskinprogram och data finns i mikrodatorns minne Maskinprogrammet består av en lång sekvens av maskininstruktioner. Processorn läser en makininstruktion åt gången från minnet, därefter tolkas instruktionen varefter den utförs. Proceduren upprepas oupphörligt. Processorns uppgift är alltså att "*hämta*" (läsa/tolka) och *utföra instruktioner*.

"Address Bus"
"Data Bus"
"Control Buses"

Hurvida en maskininstruktion eller någon variabel hämtas från minnet bestäms av bussarnas värden. Traditionella datorsystem har tre bussar: Dessa är *adress-*, *data-* och *styr-* bussar (*Address-, Data- and Control Buses*). Förloppet att genomföra *en* överföring på bussarna sker under en *busscykel*.

För att välja om data skall överföras mellan processor och inenhet, minnet eller utenhet är ett mikrodatorsystem utrustat med ett logikblock som kallas *adressavkodningslogik*. Logikblockets uppgift är att avkoda adressbussen och skicka en aktiveringssignal antingen till minnet, inenheten eller utenheten (se figur 2.3).



FIGUR 2.3 PROCESSOR MED MINNE, IN- OCH UT-ENHETER OCH ADRESSAVKODNINGSLOGIK

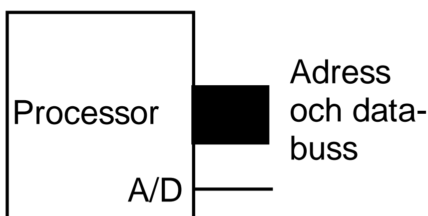
Vi övergår nu till att beskriva minnet, processor, bussarna och mikrodatorsystemets arbetssätt mer ingående. Vi börjar med bussarna.

2.1.1 Mikrodatorsystemets bussar

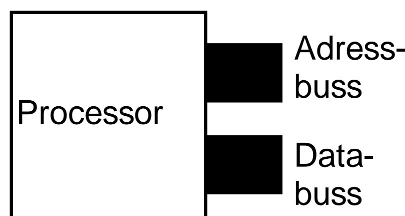
Bussarnas uppgift är att överföra information mellan processor, minne, in- och utenheter. På detta sätt kopplas de olika blocken i mikrodatorsystemet samman. Information kan vara: data, adresser eller styrsignaler av olika slag. Bussarna består av signalledningar (eller trådar).

Vi beskrev tidigare att det finns tre bussar i ett traditionellt mikrodatorsystem. Dessa är adressbussen som används av processorn för att adressera (peka ut) olika block i mikrodatorsystemet. Databussen används för att överföra data mellan olika block och för att överföra instruktioner in till processorn. Slutligen styrbussen som används för att överföra styrsignaler t ex. klocksignaler eller en signal som anger hurvida data skall skrivas till minnet eller läsas från minnet.

Man talar om *multiplexade* och *icke multiplexade bussar.* Vissa mikroprocessorer har multiplexade adress- och databussar. Med detta menas att de växelvis kan användas för att överföra adresser och data. Figur 2.4 visar en processor med en multiplexad buss och figur 2.5 en processor med icke-multiplexad buss.

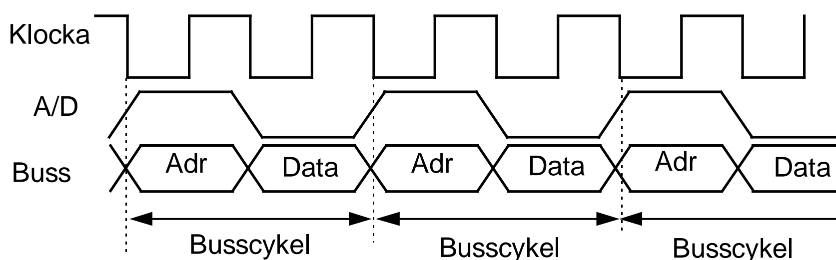


FIGUR 2.4 MULTIPLEXAD BUSS



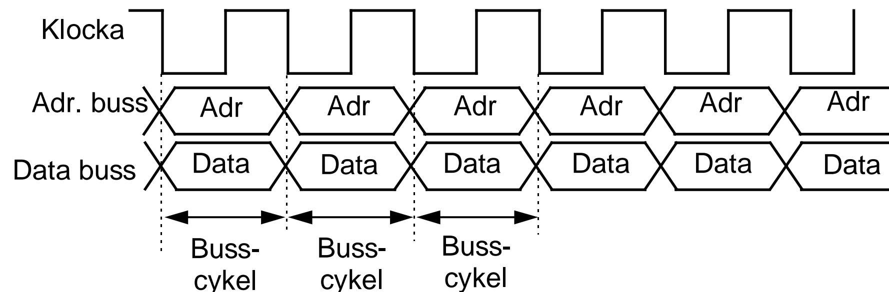
FIGUR 2.5 ICKE-MULTIPLEXAD BUSS

Studera figur 2.6. Som figuren visar finns en signal A/D som anger hurvida den multiplexade bussen används för att överföra data eller adresser. Fördelen med en multiplexad buss är att det inte krävs så många pinnar på processorn jämfört med en processor som har en icke-multiplexad buss.



FIGUR 2.6 ÖVERFÖRING PÅ EN MULTIPLEXAD BUSS

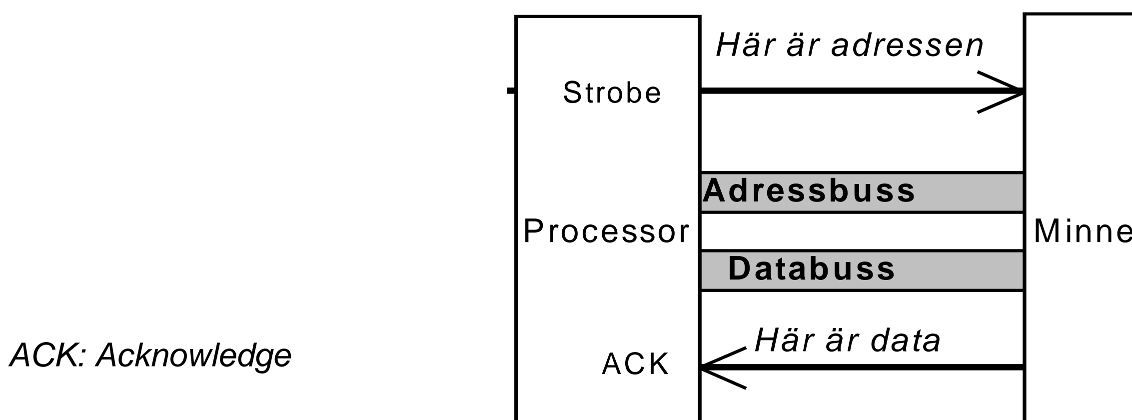
Nackdelen med en multiplexad buss jämfört med en icke-multiplexad är att den är långsammare och att det ofta krävs logik utanför processorn som kan separera data och adress. Sättet att överföra information på en icke-multiplexad buss illustreras i figur 2.7. I detta fall finns alltså en separat adressbuss och en separat databuss.



FIGUR 2.7. ÖVERFÖRING MED EN ICKE-MULTIPLEXAD BUSS

Mikrodatorsystem utnyttjar *asynkrona* eller *synkrona* bussar (bussprotokoll). Kortfattat kan man säga att vid ett asynkront bussprotokoll utnyttjas handskakningssignaler mellan de block som data skall överföras mellan, medan vid ett synkront protokoll används inga handskakningssignaler (se figurer 2.8 och 2.9 nedan).

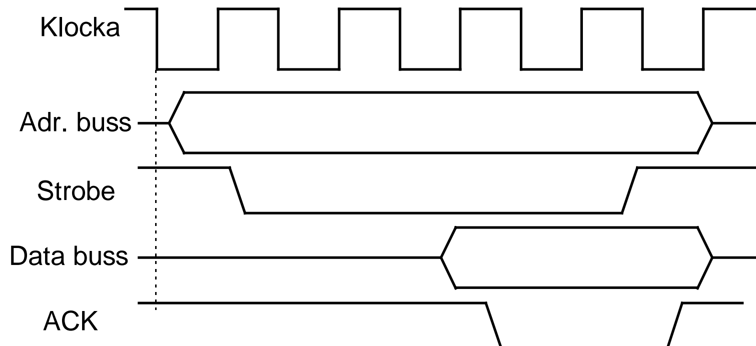
Observera att processorn är utrustad med en "Strobe"-signal som indikerar att adressen ut från processorn är giltig och en "ACK"-signal. Under en busscykel (läsning i minnet) placerar först processorn adressen på adressbussen, sedan aktiverar processorn signalen Strobe. Från och med nu så inväntar processorn "ACK"-signalen från minnet.



FIGUR 2.8 ASYNKRONT BUSSPROTOKOLL

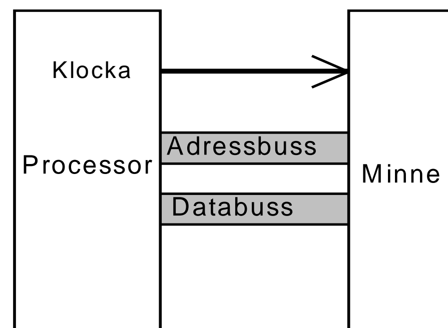
När minnet upptäckt "Strobe"-signalen kommer det att avkoda adressbussen, leta upp data och så småningom placera data på databussen. Därefter aktiverar minnet ACK. När processorn upptäckt en aktiv "ACK"-signal läser den in data från databussen. Processorn

indikerar sedan att den läst databussen genom att återställa Strobe. Därefter avlägsnar den adressen på adressbussen. Minnet som ser att Strobe inte längre är aktiv avlägsnar ACK och det data som är placerat på databussen.



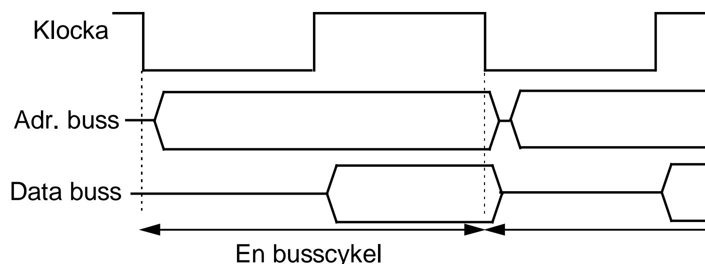
FIGUR 2.9. ÖVERFÖRING PÅ EN ASYNKRON BUSS

Vid synkront bussprotokoll används inga handskakningssignaler, utan en klocksignal. Se figur 2.10 och figur 2.11. Denna klocka är oftast långsammare än den klocka visas i figuren med asynkront protokoll. Vid ett synkront protokoll förutsätter processorn att minnet klarar att tillhandahålla data i tid.



FIGUR 2.10 SYNKRONT PROTOKOLL

Detta medför att klockan måste anpassas för det långsammaste blocket i mikrodatorsystemet. Denna nackdel uppträder ej i ett asynkront protokoll där man kan använda block med olika svarstid (snabbhet) och där varje enskilt block "svarar" så snabbt det kan. Fördelen med ett synkront bussprotokoll är att det inte krävs handskakningssignaler och därmed mindre hårdvara.



FIGUR 2.11. ÖVERFÖRING PÅ EN SYNKRON BUSS

2.1.2 Mikrodatorsystemets minnen

Minnets uppgift är att lagra program och data

Mikrodatorsystemets minne kan delas upp i *primärminne* och *sekundär-minne.* I primärminnet lagras de program (maskinprogram) och de data som för tilllfället används. Sekundärminnet är som namnet säger, en sekundär lagringsplats för program och data.

En av de viktigaste orsakerna till att det finns två olika minnen är att kostnaden per lagrad bit är mycket högre för primärminne än för sekundärminnet. Däremot är primärminnet mycket snabbare är sekundärminnet. Med detta menas tiden det tar att hämta variabler och maskininstruktioner från primärminnet är mycket kortare än den tid det tar att hämta motsvarande från sekundärminnet.

Om endast ett program utförs åt gången räcker det med att det snabba (och mer kostsamma) primärminnet är tillräckligt stort för att detta program får plats. Däremot är det bra om sekundärminnet är så stort som möjligt så att man kan få plats för alla program man kan tänka sig utnyttja.

***1 byte** = 8 bitar*

***1 Kbyte** = 1 024 bytes*

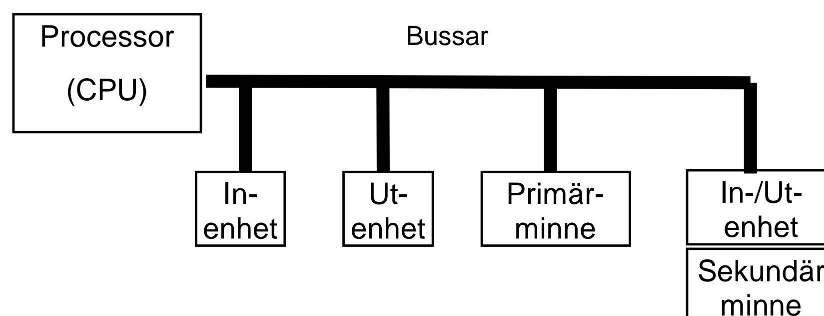
***1 Mbyte** = 1.024 kbyte*
= 1.048.576 bytes

***1 Gbyte** = 1.024 Mbytes*
= 1.048.576 kbytes

Minnets storlek anges i *kbyte* (kilo byte, 10^3 byte egentligen 1024 bytes, men man säger endast kilo; 1000) eller *Mbyte* (Mega byte = 10^6 byte) där en byte innehåller 8 bitar (nollor eller ettor). Större minnen innehåller flera *Gbyte* (Giga byte= 10^9 byte)

Exempel på sekundärminnen är flexskivestationer (disketter) och hårddiskar. En diskett i dag rymmer mellan 360 kbyte till 1.44 Mbyte beroende på vilken typ man använder. En hårddisk kan rymma hundratals Mbyte. Riktigt stora diskar innehåller flera *Gbyte* (Giga byte= 10^9 byte)

Som figur 2.12 visar så måste mikrodatorsystemet utrustas med en speciell kombinerad in- och utenhet för att ansluta sekundärminnet till mikrodatorsystemets processor. Data kan därefter överföras blockvis mellan primärminne och sekundärminnet Storleksordningen på ett block varierar från något 100-tal byte till flera kilobyte beroende på typ av datorsystem. Observera att adressavkodningslogiken (se figur 2.3) är utelämnad i figuren 2.12.



FIGUR 2.12 DATORSYSTEM MED PRIMÄR OCH SEKUNDÄRMINNE

Vissa datorsystem som är utvecklade för speciella ändamål, exempelvis för att styra en enklare robotarm, är inte utrustade med sekundärminne utan har alla sina program i primärminne. Mera generella datorsystem, som exempelvis persondatorer har både primär- och sekundärminne. Vissa program finns alltid i primärminne medan andra program kopieras till primärminne från sekundärminnet vid behov.

Mängden primärminne en dator är bestyckad med anges i kbyte (kilo byte) eller Mbyte (Mega byte). Primärminne är vanligen integrerade kretsar (IC-kretsar) som kan anslutas direkt till processorns bussar.

Primärminnet består vanligen av PROM (*Programmable Read Only Memory*) och RWM (*Read Write Memory*). En fördel med PROM är att när det väl har programmerats så behåller det sin information oberoende av om datorsystemet är spänningssatt eller ej. Nackdelen är att informationen inte kan ändras vid programexekvering. Innehållet i RWM kan däremot ändras under exekvering. RWM mister sin information vid spänningsbortfall.

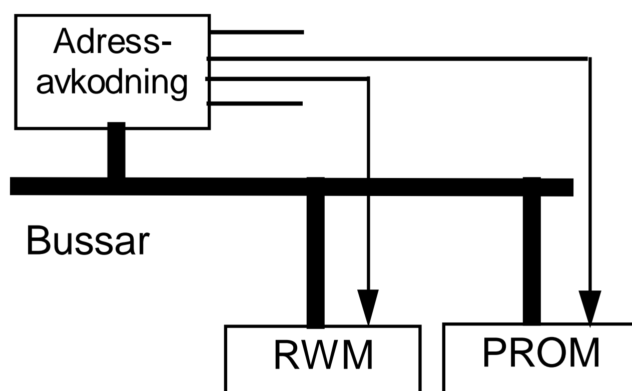
PROM: *Programmerbart minne som inte mister sin information vid spänningsbortfall.*

”Non Volatile Memory” (= icke flyktigt minne)

RWM: *Skriv och läsminne som förlorar sin information vid spänningsbortfall.*

”Volatile Memory” (= flyktigt minne)

Figur 2.13 nedan visar ett blockdiagram av primärminne med PROM och RWM. Vi kan för enkelhetens skull anta att minnet innehåller två IC-kretsar: en RWM-krets och ett PROM. Observera att kretsarna aktiveras av var sin signal från adressavkodningslogiken.



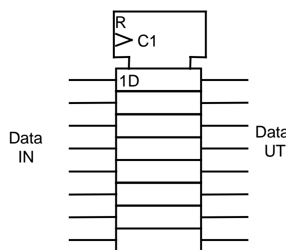
FIGUR 2.13 DATORSYSTEMETS PRIMÄRMINNE

En dator för exempelvis ett självständigt styrsystem är vanligen bestyckad med förhållandevis mycket PROM för att kunna rymma ett större program. Ett sådant system bestyckas samtidigt med förhållandevis lite RWM eftersom det vanligtvis inte omfattar speciellt stora datamängder som ändras under exekvering. Vid spänningstillslag startas ett program som ligger i PROM vilket initierar nödvändiga dataareor i RWM innan själva styrningen börjar.

En persondator däremot, är bestyckad med lite PROM och mycket RWM. Vid spänningstillslag startas enkla program som finns i PROM. Dessa enkla program läser in större program från sekundärminnet till primärminnet (till RWM). Om dessa större program startas automatiskt säger vi att vi "bootar" systemet. ("Bootar" är svengelska och kommer från *Bootstrapping*)

2.1.3 Primärminnets uppbyggnad

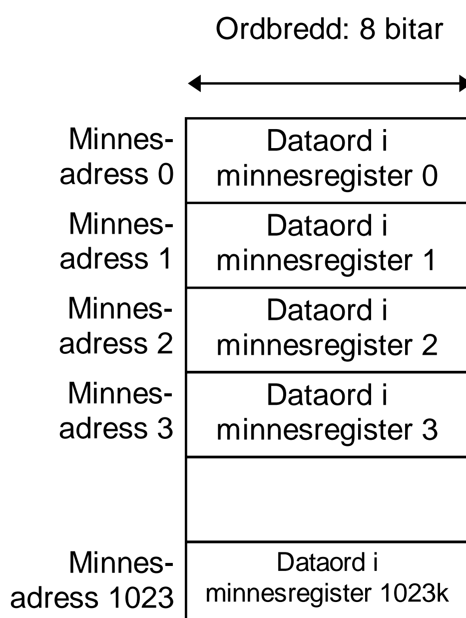
Vi kan jämföra ett primärminne med en hög byrå med tusentals lådor staplade ovanpå varandra. Man kan när som helst dra ut vilken låda som helst, och i vilken ordning som helst, och titta vad som finns i den. Man kan däremot inte öppna och titta i två lådor samtidigt.



FIGUR 2.14 ETT 8-BITARS D-REGISTER SOM ETT MINNESORD ELLER MINNESREGISTER.

I varje låda finns ett dataord bestående av ett visst antal nollor och ettor. Man talar här om minnets (eller processorns) *ordlängd.* Vanligen (i mikrodator-sammanhang) används 8 eller 16 bitars ordlängd. Det finns även datorer med mer udda ordlängder som 4, 12, 24 bitar. Kraftfullare processorer i dag utnyttjar 32 eller 64 bitars ordlängd.

Vi förutsätter för ett ögonblick 8-bitars ordlängd, ett så kallad "byte-minne" innehållande, säg 1024 minnesord. Detta minnet kan realiseras med 1024 vanliga 8-bitars D-register (se figur 2.14). Vi får då 1024 minnesregister.

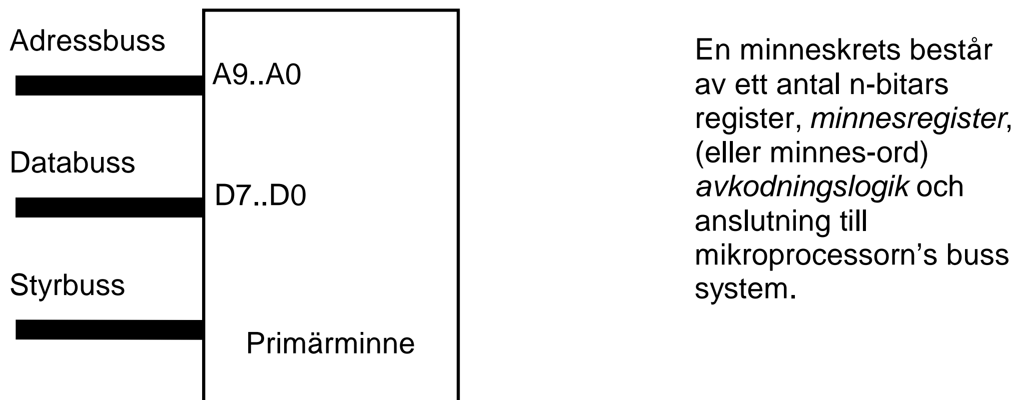


FIGUR 2.15 LOGISK LAYOUT AV PRIMÄRMINNET

Vi numrerar varje minnesregister med start från noll. Numren kallas för minnesadress. I figur 2.15 ser vi att det finns totalt 1024 minnesadresser, numrerade från 0 till 1023. För att kunna adressera (välja vilket dataord vi vill komma åt) detta minnet krävs 10 adressledningar ty $2^{10} = 1024$.

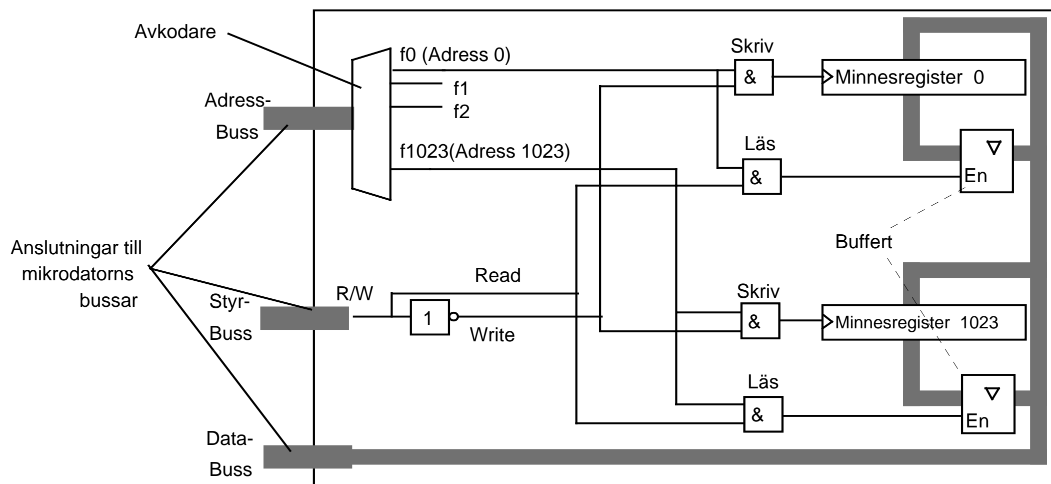
Ett litet resonemang: Om vi har en adressledning kan denna inta värdet noll eller ett och därför adressera två olika minnesregister. Om vi har två adressledningar kan dessa inta värdet 00, 01, 10 och 11. Detta är fyra olika tillstånd och vi kan då adressera 4 olika minnesregister. Testa själv med tre, fyra och fem adressledningar. I det allmänna fallet gäller att med n adressledningar kan vi adressera 2^n = antal dataord i minnet.

Dessa 1024 minnesregister (eller *minnesord*, eller bara *ord*) ryms i en integrerad krets och vi exemplifierar nu med en sådan minneskrets. I figur 2.16 nedan, visas ett blockdiagram över en sådan krets med tillhörande anslutningar. Adress-, data- och styr- bussen kopplas till motsvarande signalledningar på processorns bussar. I vårt exempel med 1024 minnesregister, är alltså 10 adressledningar kopplade till primärminnet. Eftersom vi valde byte-minne omfattar databussen 8 dataledningar. Vidare är även styrbussen ansluten. Signalledningar som ingår i styrbussen är bland andra skriv- och läs-signal som anger huvida det skall placeras ett nytt dataord i ett minnesregister eller om ett minnesregister skall läsas av processorn



FIGUR 2.16 BLOCKDIAGRAM ÖVER EN MINNESKRETS.

Vad finns då inuti en sådan minneskrets? Att 1024 minnesord finns i kretsen är klart, men det måste också finnas en avkodningslogik som avkodar den 10-bitars adressbussen och väljer ut ett unikt minnesregister. Studera figur 2.17 som visar kretsens interna uppbyggnad.



FIGUR 2.17 PRICIPIELL UPPBYGGNAD AV EN MINNESKRETS MED 1024 ORD.

R/W: Read/Write: skriv och lässignal

Figuren visar att en avkodare är ansluten till (den 10-bitars) adressbussen. Från avkodaren kommer 1024 aktiveringssignaler (f_0-f_{1023}) där endast en signal är aktiv åt gången. Varje aktiveringssignal är ansluten till två OCH-grindar, en märkt "Skriv" och en märkt "Läs". Om Read-signalen är aktiverad kommer OCH-grinden märkt "Läs" att öppna bufferten och dataordet i minnesregistrets överförs till den interna databussen. En skrivning går till på liknande sätt. Om Write-signalen är aktiverad kommer minnesregistrets klockingång att aktiveras via grinden "Skriv" och värdet på den interna databussens kommer att klockas in i registret.

Data *kopieras* från minnet när processorn läser från en adress i minnet. Data finns alltså kvar (oförändrat) i minnet.

Observera att R/W-signalen (Read/Write) antingen är noll eller ett. När den är ettställd kommer alla OCH-grindarna märkt "läs" att väljas, men endast den OCH-grinden som får en aktiveringssignal från avkodaren kommer att aktivera "sin" buffert.

Vid en läsning av minnet, *Read*, kopieras minnesregistrets innehåll till databussen. Processorn i sin tur läser databussen.

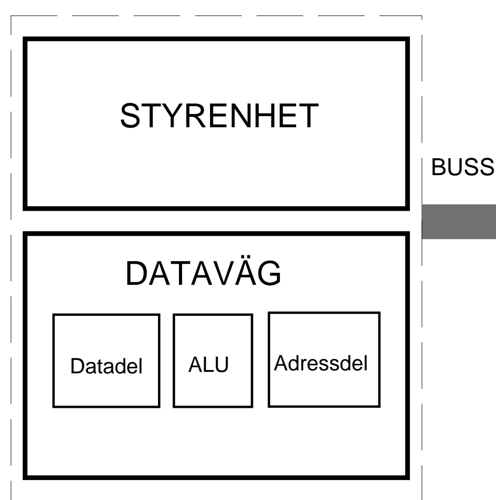
Endast ett minnes-

Vid en skrivning till minnet, *Write,* förutsätts det att processorn skriver ett värde på databussen. Detta värde är tillgängligt på alla minnesregisters ingångar. Då ett minnesregister klockas kommer registret att få databussens värde.

2.1.4 Mikroprocessorns uppbyggnad

Mikroprocessorns uppgift är att hämta maskininstruktioner från minnet, tolka och utföra dessa.

Processorn kan internt delas upp i två block: *dataväg* och *styrenhet*. Datavägens uppgift är att bearbeta data och styrenhetens uppgift är att styra datavägen (se figur 2.18).

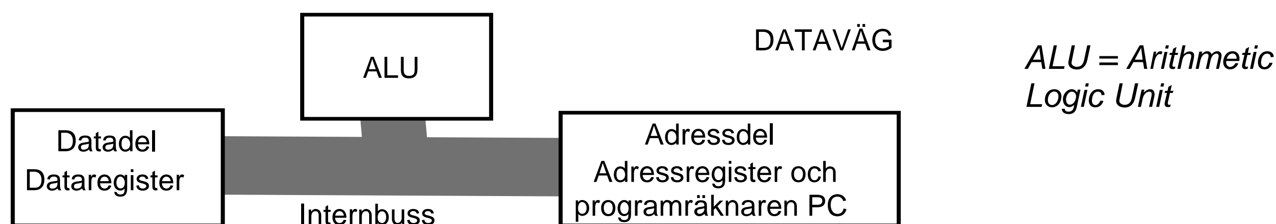


FIGUR 2.18 EN MIKROPROCESSORS INTERNA BLOCK

Datavägen kan i sin tur delas in i en datadel, en adressdel och en *ALU*-del (*Arithmetic Logical Unit*) som används vid beräkningar. Datadelen innehåller ett antal arbetsregister (eller *dataregister*) och dessa används för att mellanlagra data som skall bearbetas. Termerna 8-bitars processor, 16 eller 32-bitars processor anger hur stora arbetsregistren är.

Även adressdelen innehåller ett antal register (*adressregister*). Dessa används för att kunna peka ut (*Adressera*) data, variabler och maskininstruktioner i minnet. Här finns ett speciellt register som kallas

programräknaren (*Program Counter, PC*) ett speciellt register som används för att peka ut vilken instruktion som står i tur att hämtas. Vi återkommer till detta register senare.



ALU = Arithmetic Logic Unit

FIGUR 2.19 DATAVÄGEN I EN MIKROPROCESSOR

Observera också att datadelen, ALU'n och adressdelen i datavägen är förbundna via en intern buss. Styrenheten genererar styrsignaler för den nödvändiga överföringen mellan dessa block.

Styrenheten styr datavägen och processorns bussar för att läsa in maskininstruktioner och data. Beroende på vilken instruktion den läser genererar den olika styrsignaler till datavägen. Ett exempel kan vara en instruktion som anger att innehållen i två dataregister skall adderas. Styrenheten kommer då att generera styrsignaler så att innehållet i dessa två dataregister överförs till ALU:n som utför en addition. Slutligen placeras resultatet i ett dataregister innan nästa instruktion hämtas från minnet

2.1.5 Mikroprocessorns arbetssätt.

Arbetssättet hos en mikroprocessor är väldefinerat och beskrivs detaljerat, oftast i handböcker och datablad utgivna av tillverkaren. Att som nybörjare försöka förstå tekniken genom att sätta sig ner och läsa ett datablad om en processor är dock knappast möjligt, bland annat därför att det här ofta finns en mängd obekanta termer och begrepp.

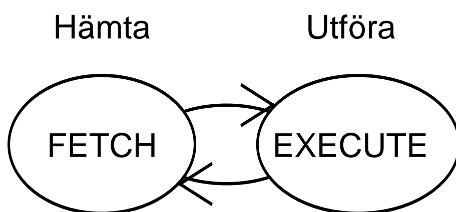
Vi nöjer oss, än så länge, med en förenklad beskrivning av mikroprocessorns's funktion och arbetssätt och formulerar följande förutsättningar:

- En *maskininstruktion* är en sekvens nollor och ettor
- Maskininstruktioner *lagras i sekvens* i datorns primärminne.
- En mikroprocessor kan självständigt *hämta* in och *utföra* maskininstruktioner, dock endast en åt gången.
- Mikroprocessorns *programräknare* (PC) pekar hela tiden ut den instruktion som står i tur att utföras.
- Mikrodatorns *primärminne* består av ett antal minnesregister som vardera *innehåller* en komplett eller en del av en *maskin-instruktion*.

- Innehållet (ett fixt antal nollor och ettor) i varje minnesregister kallas ett *ord*.
- Förloppet att hämta innehållet i ett minnesregister sker under en *busscykel*.
- En *klocksignal* (från en kristall) styr processorns arbetstakt.

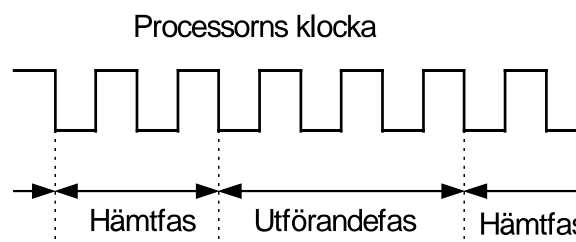
"Execute"= exekvera,utföra

Processorn kan alltså självständigt läsa in och utföra maskininstruktioner. Förloppet betecknas *hämtfas* (*FETCH)* och *utförandefas* (*EXECUTE*), se figur 2.20. Under hämtfasen läser processorn in all den information den behöver för att senare under utförandefasen exekvera instruktionen.



FIGUR 2.20 EN PROCESSORS TVÅ OLIKA FASER

Beroende på maskininstruktionens längd (hur många ord den upptar i minnet) tar det ett visst antal klockcykler att läsa in den (se figur 2.21). Beroende på maskin-instruktionens komplexitet,dvs hur "svår" den är att utföra, tar det ett visst antal klockcykler att exekvera den.



FIGUR 2.21 HÄMT- OCH UTFÖRANDEFASER RELATERAD TILL PROCESSORNS KLOCKA

När en utförandefas är slutförd startas en ny hämtfas och på detta sätt kommer mikroprocessorn att oupphörligt hämta och utföra maskininstruktioner.

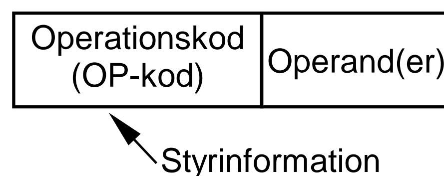
2.1.6 Maskininstruktionen

En maskininstruktion är den enklaste operationen en mikroprocessor kan utföra. En maskininstruktion utförs i sin helhet då den väl påbörjats

En maskininstruktion består av ett antal nollor och ettor. Olika processorer karakteriseras av att de också har olika instruktionsuppsättningar och detta är också anledningen till att ett maskinprogram avsett för exempelvis en Intel-processor normalt inte kan exekveras på en processor från Motorola. Däremot väljer olika fabrikanter vanligtvis att dela upp maskininstruktionerna för en processor i *instruktionsgrupper*. Oftast delas dessa in i grupper efter tillhörighet. Några exempel på olika instruktionsgrupper är:

- Instruktioner för dataöverföring
- Aritmetiska instruktioner
- Logiska instruktioner
- Programflödesinstruktioner
- Övriga instruktioner

Instruktionsformatet beskriver maskininstruktionens uppbyggnad. Olika instruktionsgrupper kan ha olika instruktionsformat. Gemensamt för alla instruktions-format är dock *operationskoden* (OP-kod). Av operationskoden framgår dels vilken typ av operation som ska utföras dels instruktionsformatet, vilket i sin tur också (entydigt) bestämmer operandinformationen. Av operandinformationen framgår också var eventuella resultat skall lagras.



FIGUR 2.22 INSTRUKTIONSFORMAT.

I följande exempel visas en sekvens assemblerinstruktioner med en instruktion hämtad från varje instruktionsgrupp ovan. (Observera att programmet inte gör något vettigt.)

*Notera att vi skriver hexadecimala tal med prefixet **$** och binära tal med prefixet **%**.*

EXEMPEL

Assemblerprogram för MC68000

```
MOVE.W      #$1234,($543210).L
ADD.L       D2,D0
ORI.B       #5,D3
JMP         ($300008).L
RESET
```

2.1

Exemplet är typisk som ett assemblerprogram för vilken processor som helst. Till vänster skrivs en bokstavskombination som anger vad instruktionen skall utföra och till höger operandinformation. Observera att när operandinformationen innehåller två operander anges först *källan* och sist *destinationen.*

Den första instruktionen

```
MOVE.W      #$1234,($543210).L
```

Data Movement Instructions

är ett exempel på en instruktion som utför en dataöverföring och flyttar talet $1234 *till* adress $543210 i minnet (till minnesregistret på adress $543210). (Från källan till destinationen).

Nästa instruktionen ingår i gruppen för aritmetiska instruktioner och utför en addition.

Arithmetic Instruction

```
ADD.L      D2,D0
```

Instruktionen adderar innehållet i dataregister D2 *till* dataregister D0. (*Från* D2 *till* D0) Resultatet placeras i dataregister D0 (destinationen till höger om kommatecknet).

Sedan följer en logisk instruktion

Logical Instruction

```
ORI.B      #$5,D3
```

Den utför logisk ELLER på värdet 5 (%0000 0101) och innehållet i dataregister D3. Resultatet placeras i dataregister D3 (till höger om kommatecknet).

Instruktionen

Program Flow Instruction

```
JMP        ($300008).L
```

tillhör gruppen hoppinstruktioner. Den utför ett ovillkorligt hopp till adress $30 00 08 i primärminnet. Instruktionen har endast en operand.

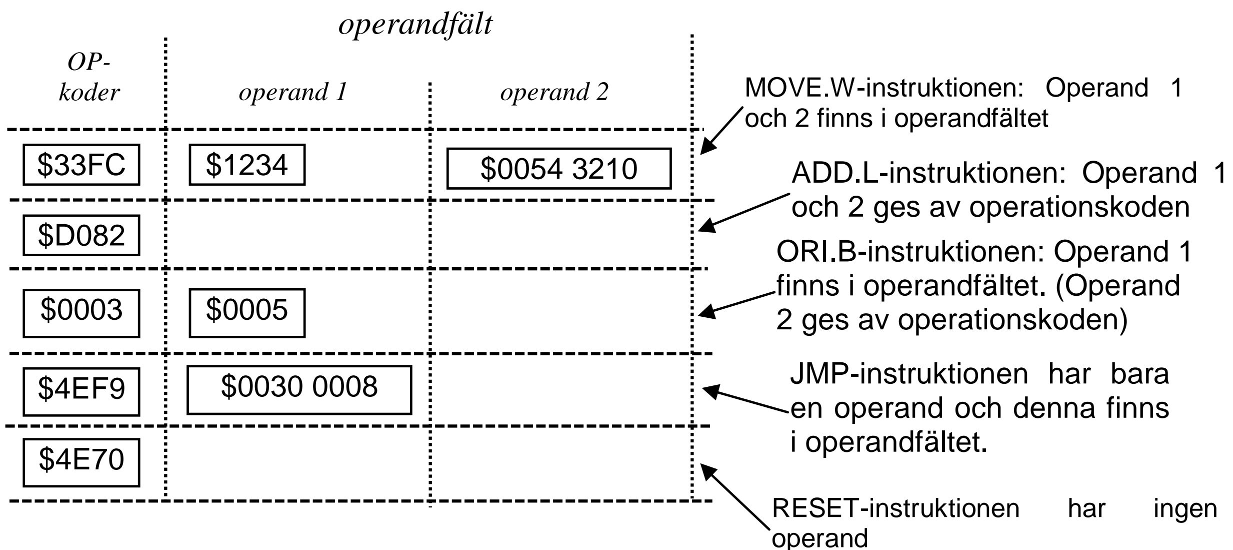
Slutligen visas en instruktion som tillhör gruppen "övriga instruktioner"

System Control Instruction

```
RESET
```

Instruktionen utför en *återstart* av datorsystemets In- och Ut-enheter. Denna instruktion har inte någon operand alls.

Vi har nu sett ett assemblerprogram och studerat varje instruktion i programmet. Vi frågar oss nu: Hur ser tillhörande maskininstruktion ut? Se figur 2.23.



FIGUR 2.23 EXEMPEL PÅ MASKINPROGRAM.

Vi har tidigare nämnt att en maskininstruktion består av en sekvens nollor och ettor. Vi väljer här att representera med hexadecimala tal. Överst i figuren visas MOVE.W-instruktionen. Den består av operationskod (OP-kod) och två operander. OP-koden är $33FC som binärt är 0011001111111100. OP-koden anger att instruktionen har två operander i operandfältet, källoperanden (data-operand) $1234 som skall flyttas till destinationsoperanden (adress-operand) $543210 i minnet.

Nästa instruktion är ADD.L-instruktionen med OP-kod $D082. Denna OP-kod anger att operanderna finns i processorns dataväg (Register D2 och D0) och därför behövs inget ytterligare operandfält.

ORI.B-instruktionen har OP-kod $0003 som anger att ena operanden finns i processorns dataväg (register D3) och att den andra operanden ($5) finns i operandfältet.

Instruktionen JMP med OP-kod $4EF9 har en enda operand ($300008) och denna finns i operandfältet.

Slutligen visas RESET-instruktionen med OP-kod $4E70 som saknar operander helt och hållet.

Hur detta maskinprogram (eller dessa maskininstruktioner) ser ut i mikrodatorsystemets minne visas i figur 2.24. Vi förutsätter här att programmet är placerat på adress $4000 och framåt i minnet. Eftersom vi exemplifierar med maskinprogram för MC68000 visas här 16-bitars minne och endast jämna adresser.

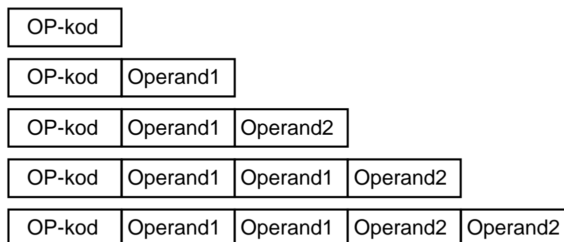
Adress-angivelse	OP-koder och Operander	
$4000	$33FC	MOVE.W #$1234,($543210).L
$4002	$1234	
$4004	$0054	
$4006	$3210	
$4008	$D082	ADD.L D2,D0
$400A	$0003	ORI.B #$5,D3
$400C	$0005	
$400E	$4EF9	JMP ($300008).L
$4010	$0030	
$4012	$0008	
$4014	$4E70	RESET
$4016		HÄR BÖRJAR NÄSTA INSTRUKTION

FIGUR 2.24 MASKINPROGRAM LAGRAT PÅ ADRESS $4000 OCH FRAMÅT I PRIMÄRMINNET.

Studera instruktionen `MOVE.W #$1234,($543210).L` i figur 2.24. Det framgår att instruktionen upptar fyra 16-bitars ord i minnet från adress $4000 till adress $4008. Det första ordet $33FC är instruktionens operationskod som anger att detta är en MOVE.W-instruktion och att instruktionen upptar fyra ord (åtta bytes) i minnet där den första operanden är data och den andra en adress. Vi känner igen den första operanden $1234 på följande adress i minnet. Denna ska placeras på adress $54 32 10. Den andra operanden, adressen, hittas på två olika adresser $4004 och $4006 i minnet. $0054 och $3210 bildar tillsammans den önskade adressen.

De tre första instruktionerna i figur 2.24 har alla två operander. Den första har båda sina operander i anslutning till operationskoden. Nästa instruktion har också två operander, men här är båda operanderna i processorns register. Den tredje instruktionen har en operand i anslutning till operationskoden och en i processorns register.

Instruktion fyra har enbart en operand och den sista instruktionen har ingen operand alls. Figur 2.25 visar några olika instruktionsformat.



FIGUR 2.25 OLIKA INSTRUKTIONSFORMAT.

Vi har nu studerat hur assemblerprogram och assemblerinstruktioner ser ut, hur tillhörande maskininstruktion ser ut och hur ett maskinprogram är lagrat i mikrodatorns primärminne. Låt oss nu se hur mikroprocessorn utnyttjar bussarna när den arbetar.

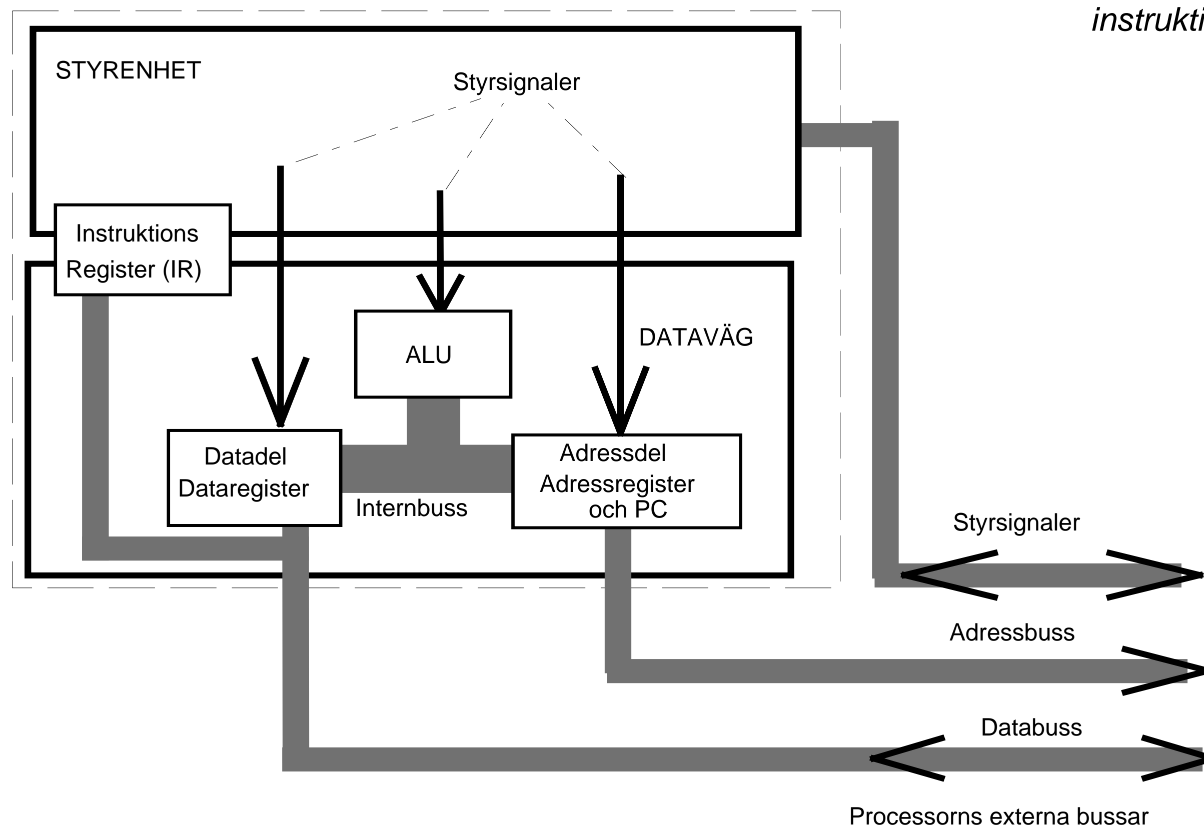
2.1.7 Användning av bussarna

Vi ska nu ge en kortfattad beskrivning av hur processorn utnyttjar bussarna när den arbetar och när den hämtar och utför instruktioner. Först ska vi dock studera processorn lite mer i detalj. I processorn finns förutom dataväg och styrenhet även ett *instruktionsregister,* **IR**, vilket används för att avgöra vilken instruktion som skall utföras. Under hämtfasen av en instruktion läser processorn operationskoden (OP-koden) från primärminnet och placerar denna i instruktionsregistret (IR). Därefter avkodas OP-koden av styrenheten och den kan då generera de nödvändiga styrsignalerna till datavägen för att få aktuell instruktion utförd. Se figur 2.26.

Bussarnas funktion är att överföra information mellan mikroprocessor, minne, in- och utprotar.

OP-koden överförs till IR

PC pekar på nästa instruktion



FIGUR 2.26 EN MIKROPROCESSOR MED DATAVÄG, INSTRUKTIONSREGISTER OCH STYRENHET.

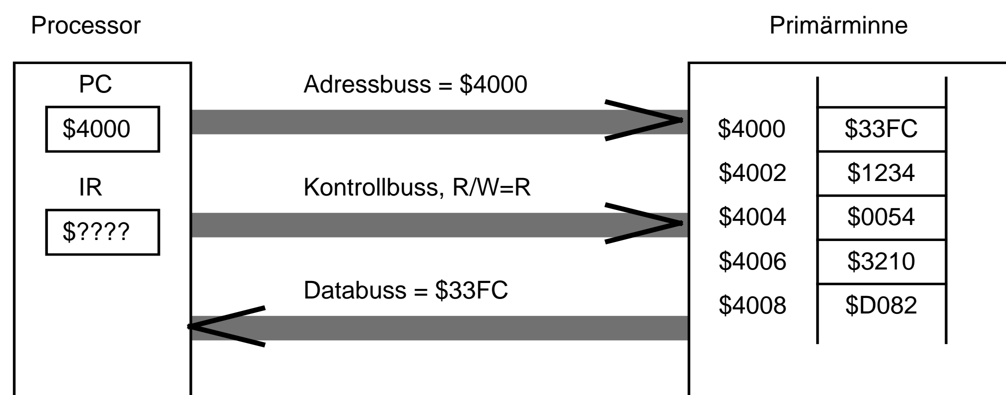
Observera att datavägen även innehåller temporära data-och adressregister som är osynliga för användaren. Dessa register använder processorn för att mellanlagra data och adresser under utförandet av *en* instruktion. Dessa register kan vi som assemblerprogrammerare inte adressera på samma sätt som vid exempelvis `ADD.L  D2,D0`. Figur 2.26 visar ett sådant dolt register, nämligen IR.

Minnes Adr.	OP- och Operand	Instruktion
$4000	$33FC	MOVE.W ...
$4002	$1234	
$4004	$0054	
$4006	$3210	
$4008	$D082	ADD.L ...
$400A	$0003	ORI.B ...
$400C	$0005	
$400E	$4EF9	JMP ...
$4010	$0030	
$4012	$0008	
$4014	$4E70	RESET
$4016		

FIGUR 2.27 MASKINPROGRAM LAGRAT PÅ ADRESS 4000 OCH FRAMÅT I PRIMÄRMINNET.

Nu till exemplet. Vi skall studera hur mikroprocessorn utnyttjar bussarna när den hämtar och utför maskininstruktioner. Vi använder maskinprogrammet i figur 2.27. Programmet visas även i sin helhet i figur 2.24.

Vi förutsätter nu att programräknaren ***PC*** i MC68000 har värdet $4000 och att FETCH-fasen startar. Under **busscykel 1** i vårt exempel lägger processorn ut ***PC*** på adressbussen för att adressera adress $4000 i primärminnet. Samtidigt ger processorn en lässignal till primärminnet via styrbussen. Se figur 2.28.

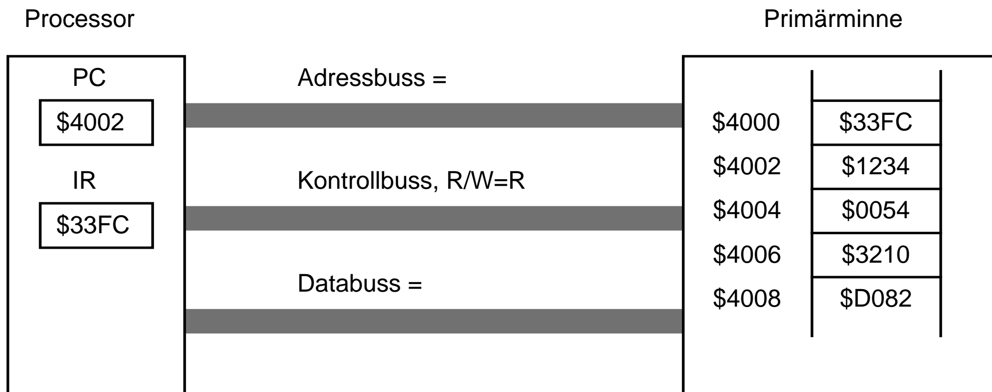


FIGUR 2.28 BUSSCYKEL 1

Primärminnet avkodar adressbussen och lägger ut innehållet ($33FC) i minnesregister $4000 på databussen. (Principen för detta visades i figur 2.17.) När busscykeln avslutas placerar processorn databussens värde ($33FC) i sitt instruktionsregister. Nu är busscykel 1 slut. Processorn har nu läst in OP-koden.

Processorn avkodar nu innehållet i instruktionsregistret. Se figur 2.29. Bitmönstret i instruktionsregistret, OP-koden $33FC, anger att en MOVE.W-instruktion skall utföras. Instruktionen upptar 4 ord i primärminnet vilket innebär att tre ord till måste läsas från

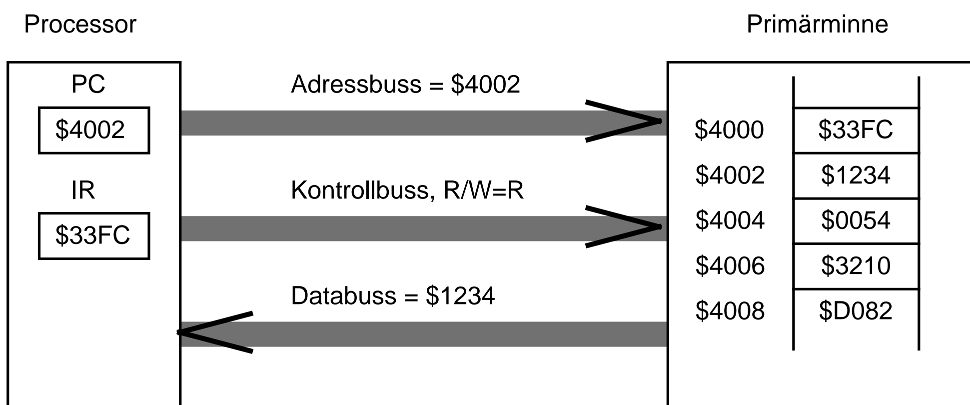
primärminnet innan maskininstruktionen kan utföras. Vidare anger OP-koden att operand 1 skall skrivas till primärminnet på den adress som operand 2 anger. Samtidigt som instruktionsregistret avkodas uppdateras ***PC*** enligt ***PC*** = ***PC***+2 (= $4002) för att *peka ut* nästa ord i minnet.



FIGUR 2.29 BUSSCYKEL 1

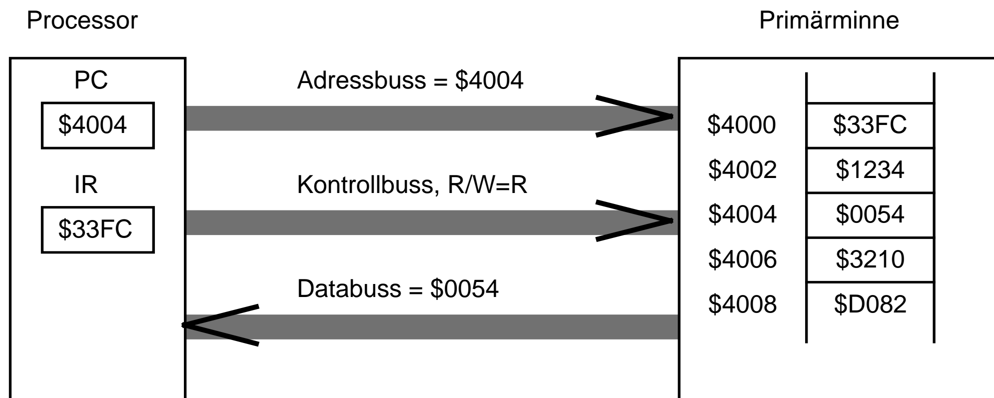
Under **busscykel 2** adresserar ***PC*** på nytt primärminnet samtidigt som en lässignal ges för att läsa in ett nytt minnesord. Se figur 2.30. Adressbussen värde är $4002 och därför läggs innehållet i minnesregister $4002 ut på databussen. Processorn läser databussen ($1234) och placerar värdet i ett temporärt register. ***PC*** uppdateras på nytt med två till $4004. Processorn har nu läst in operanden som senare skall skrivas tillbaka till minnet. Observera att programmet visas i marginalen.

Minnes Adr.	OP- och Operand	Instruktion
$4000	$33FC	MOVE.W ...
$4002	$1234	
$4004	$0054	
$4006	$3210	
$4008	$D082	ADD.L ...
$400A	$0003	ORI.B ...
$400C	$0005	
$400E	$4EF9	JMP ...
$4010	$0030	
$4012	$0008	
$4014	$4E70	RESET
$4016		



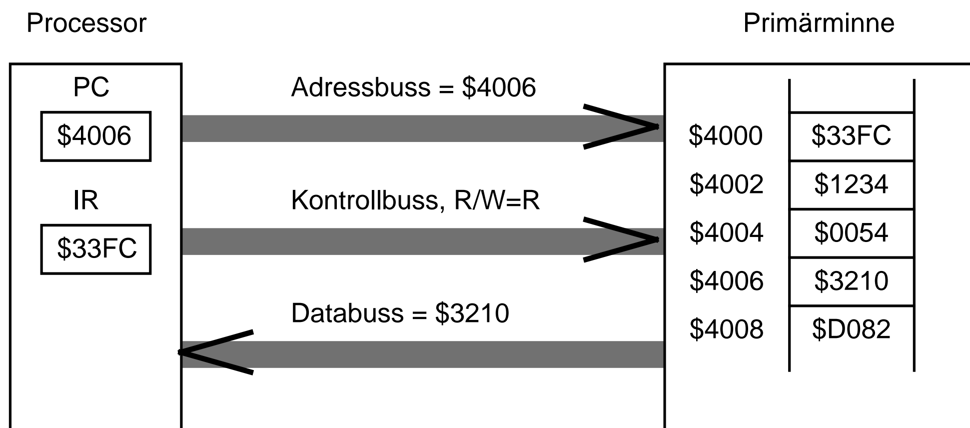
FIGUR 2.30 BUSSCYKEL 2

Under **busscykel 3** adresserar ***PC*** ännu en gång primärminnet samtidigt som en lässignal ges för att läsa in ett nytt minnesord. Se figur 2.31. Adressbussen värde är $4004 och därför läggs innehållet i minnesregister $4004 ut på databussen. Processorn läser databussen ($0054) och placerar värdet i ett nytt temoprärt register. ***PC*** uppdateras på nytt med två till $4006. Processorn har nu läst in höga delen av operand två som senare skall bilda en adress till primärminnet.



FIGUR 2.31 BUSSCYKEL 3

FETCH-fasen avslutas med **busscykel 4** som adresserar primärminnets adress $4006 ty ***PC*** är $4006 samtidigt som en lässignal ges för att läsa in $3210. Se figur 2.32. ***PC*** uppdateras på nytt med två till $4008 för att nu peka ut *nästa instruktion* i primärminnet. Processorn läser databussen ($3210) och placerar värdet i ännu ett temporärt register. Processorn läser nu in låga delen av operand två som så småningom skall bilda en adress till primärminnet.



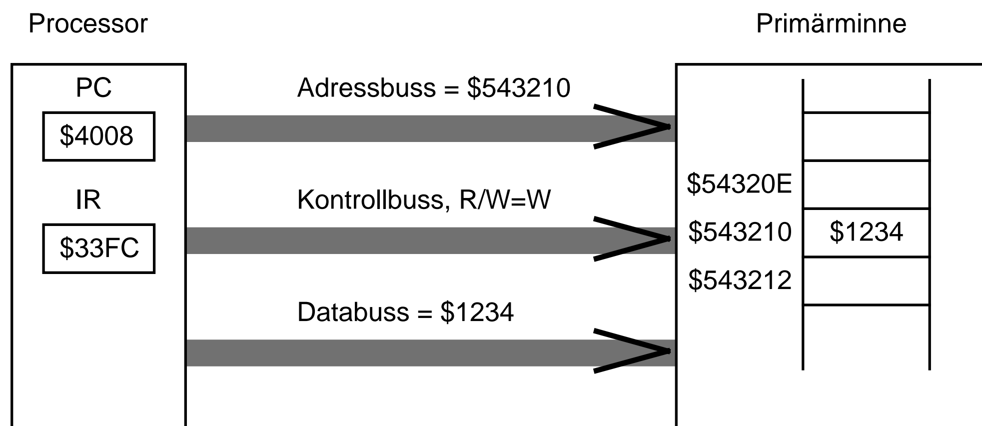
FIGUR 2.32 BUSSCYKEL 4

PC pekar på nästa instruktion

Nu är FETCH-fasen avklarad. Hela instruktionen är inläst och ***PC*** är uppdaterad till $4008 och *pekar ut* nästa instruktion i primärminnet. Nu

följer EXECUTE-fasen av vår *move*-instruktion. Det första som händer är att processorn slår ihop den höga ($0054) och den låga ($3210) delen av operand två, på så sätt bildas $54 32 10 som skall adressera minnet.

Under **busscykel 5** lägger processorn ut den inlästa adressen på adressbussen. Se figur 2.33. På så sätt adresseras minnet med adress $54 32 10. Processorn lägger även ut operand 1 ($1234) på databussen samtidigt som en skrivsignal ges på styrbussen. Detta medför att det adresserade minnesregistret på adress $54 32 10 kommer att få ett nytt värde, nämligen $1234, och EXECUTE-fasen är därmed klar.



FIGUR 2.33 BUSSCYKEL 5

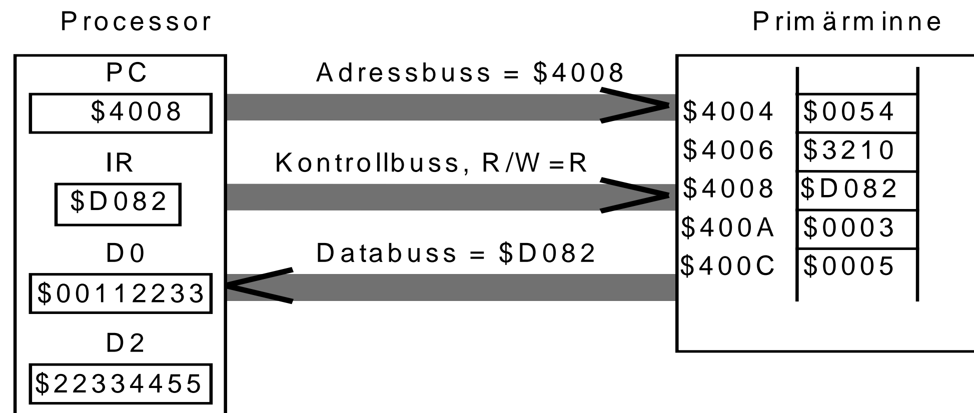
Processorn har nu avkodat och utfört instruktionen

```
MOVE.W     #$1234,($543210).L
```

Då EXECUTE-fasen är klar startas direkt en ny FETCH-fas som läser in det som **PC** adresserar ($4008) till instruktionsregistret.

Under nästa busscykel (busscykel ett för nästa instruktion) kommer därför $D082 att läsas till instruktionsregistret. (Figur 2.34). Detta är operationskoden för `ADD.L D2,D0.` Processorn avkodar detta bitmönster och kommer fram till att denna instruktionen är endast ett ord lång och att innehållet i dataregister D2 skall adderas till innehållet i dataregister D0.

Styrenheten i processorn förutsätter nu att PC pekar på början av en instruktion.



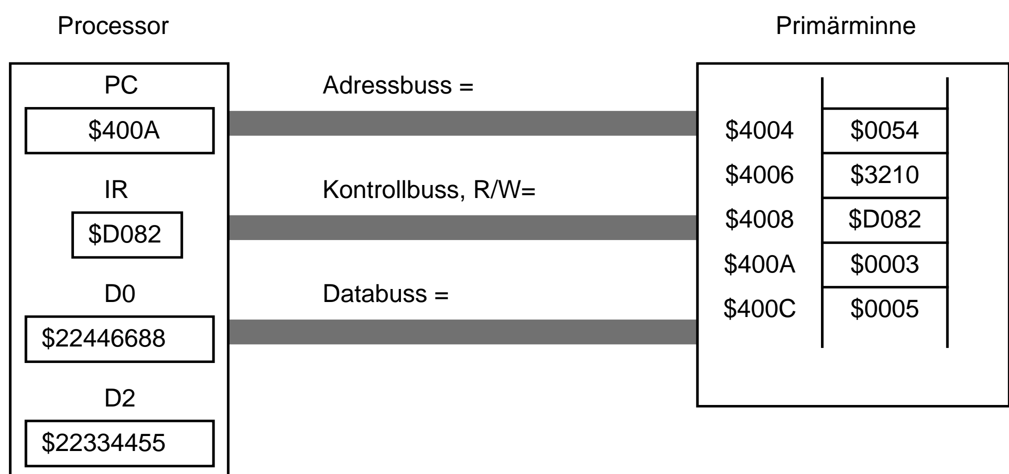
FIGUR 2.34 BUSSCYKEL 1 FÖR ADD-INSTRUKTIONEN

Minnes Adr.	OP- och Operand	Instruktion
$4000	$33FC	MOVE.W ...
$4002	$1234	
$4004	$0054	
$4006	$3210	
$4008	$D082	**ADD.L ...**
$400A	$0003	ORI.B ...
$400C	$0005	
$400E	$4EF9	JMP ...
$4010	$0030	
$4012	$0008	
$4014	$4E70	RESET
$4016		

Förutsätt nu att register D0 innehåller $00112233 och register D2 innehåller $22334455, (se vidstående figur). Additionen utförs och resultatet, $22446688 placeras i register D0, dessutom uppdateras **PC** till $400A som på detta sätt pekar ut nästa instruktion (ORI.B-instruktionen). Se figur 2.35.

D0 *$00112233*
D2 + *$22334455*

D0 = *$22446688*



FIGUR 2.35 RESULTATET EFTER ADD-INSTRUKTIONEN

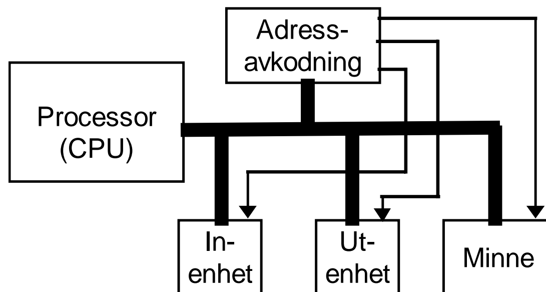
Processorn har nu utfört instruktionen `ADD.L D2,D0` och påbörjar ännu en ny FETCH-fas. ***PC*** placeras på adressbussen och operationskoden för ORI.B-instruktionen läses in. På så sätt kommer

processorn att hämta in och utföra instruktion för instruktion i exemplet. Observera att detta är en principiell beskrivning på hur en generell mikroprocessor fungerar fast vi har exemplifierat med MC68000. (MC68000 *är* en generell mikroprocessor) En del detaljer har utelämnats då de inte är nödvändiga för förståelsen av processorns arbetssätt. Vi återkommer till detaljer i senare kapitel.

2.2 In- och utenheter

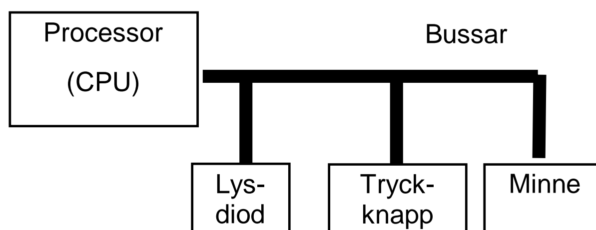
En *tryckknapp* är ett enkelt exempel på en inenhet. En enkel utenhet kan vara en *lysdiod*. Vi skall nu ansluta dessa två enkla in- och utenheter till en mikroprocessor med minne.

Vi har tidigare visate ett komplett blockdiagram över ett mikrodatorsystem. Se figur 2.36. Figur 2.37 visar ett blockdiagram över vårt mikrodatorsystem med lysdiod och tryckknapp. Observera att adressavkodningslogiken är utelämnad i denna figur.



FIGUR 2.36 PROCESSOR MED MINNE, IN- OCH UT-ENHETER OCH ADRESSAVKODNINGSLOGIK

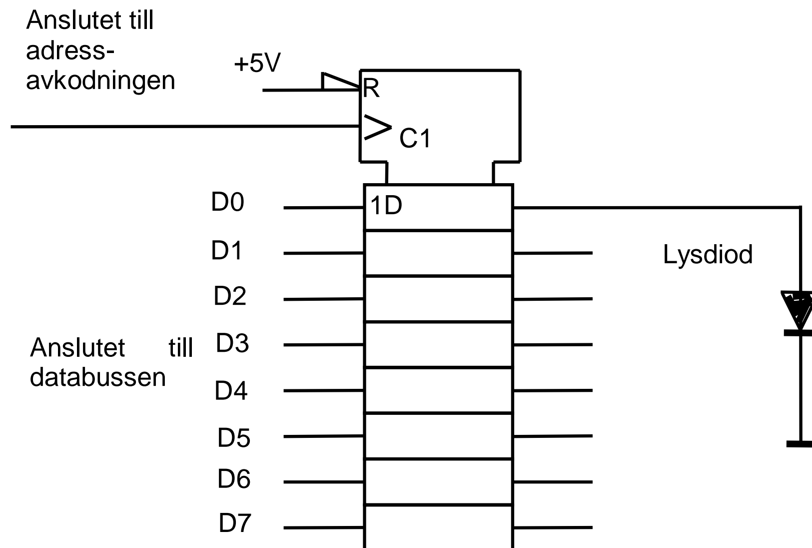
Först studerar vi utporten och ansluter en lysdiod. Om vi önskar tända lysdioden krävs det att processorn skriver en “etta” (en hög signal, +5V) till lysdioden. Önskar vi släcka den skriver vi en “nolla” till den.



FIGUR 2.37 MIKRODATORSYSTEM MED MINNE, TRYCKKNAPP OCHLYSDIOD.

Men, kan lysdioden anslutas direkt till mikrodatorns databuss? Nej, då databussen används för att hämta in instruktioner kommer den att ha olika värden för varje busscykel. Det krävs därför ett register (liknande ett minnes-register) vi kan skriva till där registrets utgång är ansluten till lysdioden.

Vidare krävs det att registret aktiveras (klockas) för en bestämd adress så att innehållet i registret endast ändras när processorn skriver till lysdioden. Se figur 2.38. Som figuren visar är lysdioden ansluten till databit noll (D0) på databussen.

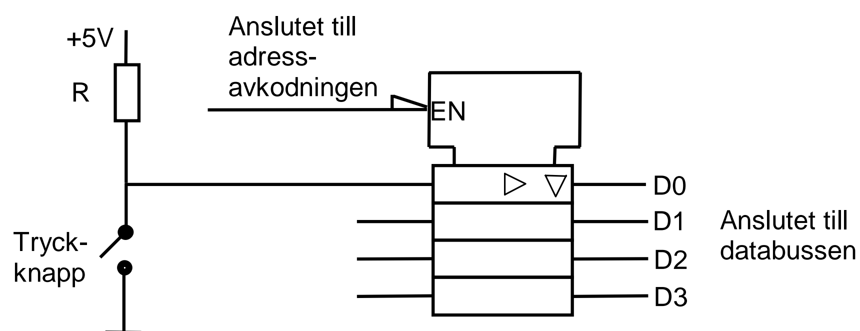


FIGUR 2.38 ETT 8-BITARS D-REGISTER 74HC273 SOM UTPORT FÖR LYSDIODEN.

När vi nu utrustat utporten med ett register kommer "ettan" eller "nollan" som tänder respektive släcker lysdioden att låsas i registret och ge en konstant utsignal tills ett nytt värde klockas in i registret.

Samma förfarande måste användas för inporten. Dvs, det måste finnas en krets som skiljer vår inenhet från mikroprocessorns bussar. I detta fallet använder vi en buffert av typen 74HC241. Se figur 2.39. Observera hur tryckknappen är ansluten till databit noll (D0).

När processorn nu läser adressen till tryckknappen kommer adressavkodningslogiken att aktivera signalen EN på bufferten och denna lägger då ut tryckknappens värde till databussen som processorn läser.



FIGUR 2.39 ETT 8-BITARS D-REGISTER 74HC241 SOM INPORT FÖR TRYCKKNAPPEN.

Avslutningsvis visar vi nu ett programexempel som läser tryckknappen och tänder lysdioden när tryckknappen är sluten.

Studera figur 2.39 som visar inporten. När tryckknappen är sluten kommer databit noll (D0) att vara noll när inporten läses. För att tända lysdioden i figur 2.38 måste D0 vara ett. Programmet måste då läsa D0 från inporten, invertera denna signal och skriva resultatet till utporten.

Vi förutsätter att tryckknappen (bufferten) är placerad på adress \$3F0010 och att lysdioden (registret) är placerad på adress \$3F0020. Studera nu följande exempel, den första instruktionen läser tryckknappens värde till processorns dataregister D3. Nästa instruktion inverterar innehållet i dataregistret och den sista skriver innehållet i D3 till lysdioden. Den sista instruktionen utför ett hopp till början av programmet.

EXEMPEL

Program för MC68000 som läser tryckknappen och tänder lysdioden.

```
Start
      MOVE.B      ($3F0010).L,D3      Läs inport
      NOT.B       D3                  Invertera
      MOVE.B      D3,($3F0020).L      Skriv till utport
      JMP         (Start).L           Hoppa till start
```

2.1

Vi har nu en programsnurra som hela tiden läser inporten, inverterar det inlästa värdet och skriver resultatet till utporten. Med detta avslutar vi kapitlet om mikrodatorns olika block.

2.3 Sammanfattning

I detta kapitel har vi studerat mikrodatorns olika block: processor, bussar, minnen, in- och utportar.

Processorn består av styrenhet och dataväg, där styrenheten styr datavägen. Processorn är antingen i FETCH-fas eller i EXECUTE-fas. Programräknaren, PC, innehåller adressen till nästa instruktion i minnet.

Processorns arbetssätt är att under FETCH-fasen läsa in en OP-kod från minnet och koda av denna. På så sätt undersöker den hur många bytes instruktionen upptar i minne och vad som skall utföras. Processorn

fortsätter med att läsa in hela instruktionen från minnet för att sedan byta tillstånd till EXECUTE. Därefter utförs instruktionen.

Syftet med de tre bussarna (adress, data och styrbuss) är att sammankoppla de olika blocken i mikrodatorn och att överföra information (data). Att överföra *ett* (data-)ord på bussarna sker under en busscykel.

Det finns multiplexade och icke multiplexade bussar. Dessa kan utnyttjas antingen för asynkron eller synkrona bussprotokoll.

Minnet i ett mikrodatorsystem delas in i primärminne och sekundärminne och räknas i kilo-, Mega eller Gigabyte. Primärminnet består av PROM och RWM medan sekundärminnet består oftast av någon form för hårddisk eller diskettstation.

Enklaste formen för in- och utenheter är tryckknappar och lysdioder. Dessa ansluts till mikrodatorsystemets bussar via buffertar och register.

DATORARITMETIK

Vi ska i detta kapitel redogöra för hur beräkningsenheten i en dator utför de vanliga räknesätten och hur talvärden lagras i datorns minne. Vi kommer att visa hur man kan välja talrepresentation för såväl positiva som negativa talvärden och hur resultatet av en aritmetisk operation, under vissa omständigheter, kan bli felaktigt. Vi behandlar olika talsystem, talrepresentationener, räknelagar och olika operationer som utförs av en vanlig processor. Slutligen beskriver vi en standard (IEEE) för flyttalsrepresentation.

Tänk dig att du kör en gammal bil och observerar vägmätaren. Bilen har nu gått 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 för att lagra talvärden och för att utföra beräkningar. Därför kan ett liknande 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 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.

Hur kan man utrycka *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 den negativa talaxeln (negativa tal). En bitsträng av nollor och ettor kan då *tolkas* till ett bestämt talvärde där *olika* tolkningar ger olika talvärden

3.1 Binära koder

Det finns en mängd binära koder (BC) av varierande betydelse och för olika ändamål, exempelvis *binärt kodade decimaltal*, *Graykod*, *streckkod*, *alfanumerisk koder* mm. I detta kapitel behandlas speciellt koder som används för talrepresentation och den så kallade *ASCII-koden* som används för att representera sådana bokstäver, siffror och övriga tecken som exempelvis finns på ett tangentbord.

3.1.1 Det binära talsystemet

Talvärdet N för en bitsträng (n-bitar) av nollor och ettor skrivs

$$N = \sum_{i=0}^{n-1} x_i 2^i$$

Det binära talsystemet, som vi i någon 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 utgörs normalt av binära tal. 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:

Addition i binära talsystemet

```
                  1
 0     0     1     1
+0    +1    +0    +1
 0     1     1     0
                   ▲
              Additionen
              resulterar i
              en minnes-
              siffra
```

Subtraktion i binära talsystemet

```
      10  (1 överstruken)
 0     0     1     1
-0    -1    -0    -1
 0     1     1     0
       ▲
      Vid denna
      subtraktion
      måste vi låna
```

Låt oss nu, med exempel, illustrera *fyrabitars* binär aritmetik

EXEMPEL

Addition (R=X+Y)

```
      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
```

3.1

EXEMPEL

Subtraktion (R=X-Y)

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

3.2

Vi återkommer till multiplikation och division i binära talsystemet senare i detta kapitel.

3.1.2 Binärkodade decimaltal

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 3.1 NBCD-koden

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)! ≈ 2,9 x 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 3.1), 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

$$9756 = \underbrace{1001}_{9}\ \underbrace{0111}_{7}\ \underbrace{0101}_{5}\ \underbrace{0110}_{6}$$

3.3

De binära talen för 10, 11, 12, 13, 14 och 15 har här ingen innebörd. (se Tabell 3.1). Några processorer har speciella instruktioner 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 ord blir större än 9.

EXEMPEL

Utför additionen 5+7 där talen är kodade på NBCD-form:

$5 = (0101)_{NBCD}$, $7 = (0111)_{NBCD}$, ställ upp additionen:

```
  1 1 1
  0 1 0 1
 +0 1 1 1
  -------
  1 1 0 0   ←   Resultatet > 9 (ej NBCD-kod), vi tvingas därför
                decimaljustera, dvs, addera 6 till resultatet
 +0 1 1 0
  -------
1 0 0 1 0
```

Om vi tolkar resultatet som NBCD-kod får vi alltså:

0001 0010 = 12 som är det förväntade resultatet

3.4

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.

3.1.3 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, ***A****merican* ***S****tandard* ***C****ode for* ***I****nformation*.

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	[	7B	{ ä
1C	FS	3C	<	5C	\ Ö	7C	| ö
1D	GS	3D	=	5D	]	7D	} å
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

Tabell 3.2 ASCII-tabellen

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 också speciella funktioner för de 32 första tecknen (i tabell 3.3). Dessa används företrädesvis i datakommunikationsutrustning. Det är exempelvis med denna kod som tecken inslagna på en dataterminal med skrivmaskins-liknande tangentbord överförs till ett digitalt system.

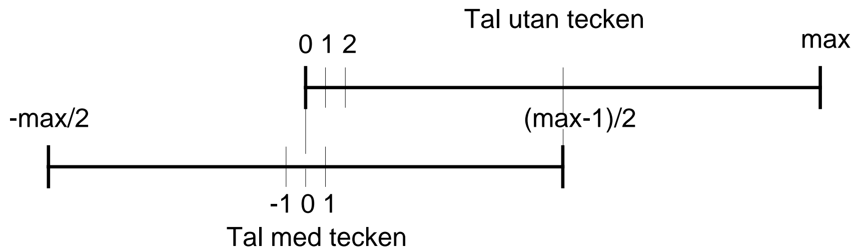
BEL	Bell	HT	Horizontal Tabulation
BS	Backspace	LF	Line Feed
CAN	Cancel	NAK	Negative Acknowlege
CR	Carriage Return	NUL	Null
DC	Device Control	RS	Record Separator
DEL	Delete	SI	Shift-In
DLE	Data Link Escape	SO	Shift-Out
EM	End of Medium	SOH	Start of Heading
ENQ	Enquiry	SP	Space
EOT	End of Transmission	STX	Start of Text
ESC	Escape	SUB	Substitute
ETB	End of Transmission Block	SYN	Synchronous Idle
ETX	End of Text	US	Unit Separator
FF	Form Feed	VT	Vertical Tabulation
FS	File Separator		

Tabell 3.3 Speciella styrtecken i ASCII-koden

3.2 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 måste då koda talen så att vissa bitsträngar representerar negativa tal och andra bitsträngar representerar positiva tal. 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 endast representera (till beloppet) hälften så stora tal jämfört med om vi önskar representera tal utan tecken (se figur 3.1 nedan). Som vi ser så är ungefär hälften av talen positiva och hälften är 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, se tabell 3.4.

Talvärdet N för bitsträngen x, med n-bitar, vid tecken-beloppsrepresentation skrivs:

$$N = (1 - 2x_{n-1}) \sum_{i=0}^{n-2} x_i 2^i$$

EXEMPEL

Tecken/Belopp

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

3.5

Då aritmetik utförs på tal med denna representation måste tecknen betraktas speciellt. Tabell 3.5 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.

>	
≥	
≥	
>	
>	
A,B<0	

Tabell 3.5 Räkneregler för addition av tecken-beloppsrepresenterade tal

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	-7
1110	14	-6
1101	13	-5
1100	12	-4
1011	11	-3
1010	10	-2
1001	9	-1
1000	8	-0

Tabell 3.4 Tecken/ Beloppsrepresentation

Av räknereglerna från tabell 3.5 inser vi att ett logiknät för att utföra operationen blir ganska omfattande. Ett sådant logiknät måste klara av såväl addition som subtraktion och teckenöverläggning.

3.3 Tvåkomplementsrepresentation

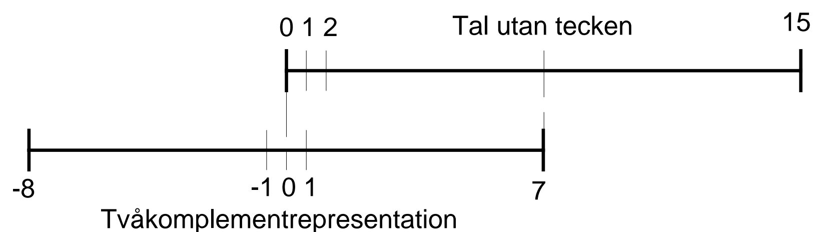
Vi vill självfallet att samma räknelagar skall gälla både för tal med och utan tecken. På så sätt är det tillräckligt med *en* digital enhet som utför samma operationer på bitmönstren oberoende av om vi representerar tal med eller utan tecken. Vi exemplifierar med fyra bitars ord. Detta innebär att vi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

- 2^n ger antal olika kombinationer, där *n* är antal bitar i talet och $2^4 = 16$ olika tal
- Vi *väljer* nu representation enligt tabell 3.6 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

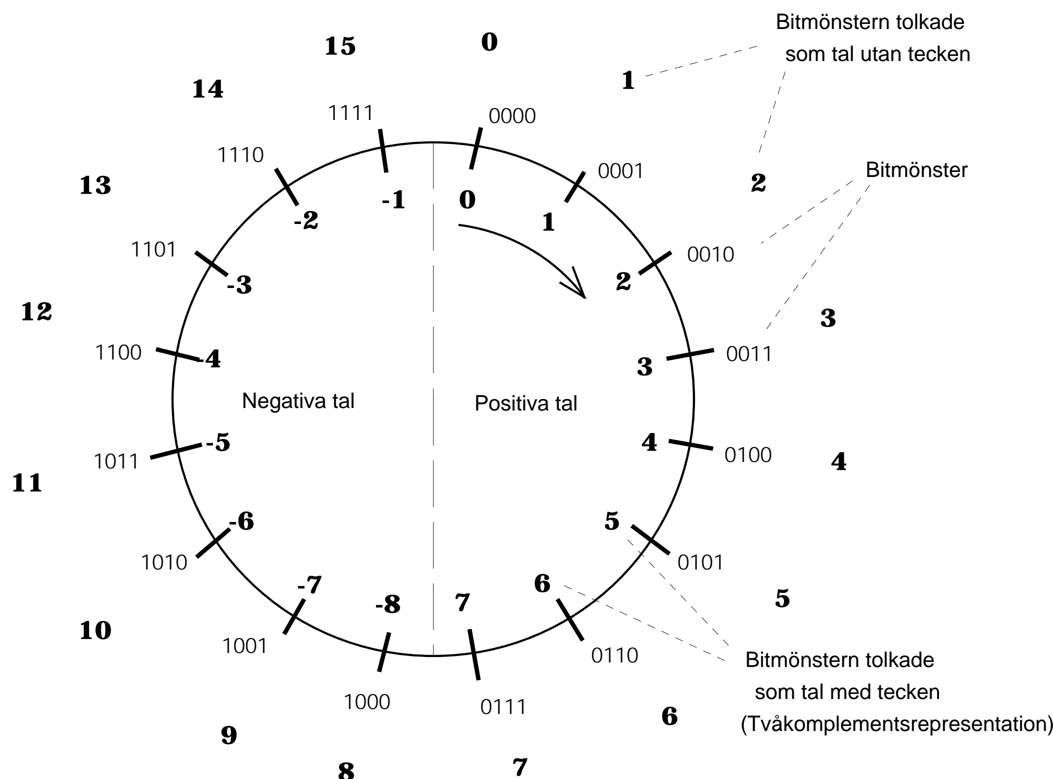
Tabell 3.6 Tvåkomplements-representation

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, se figur 3.2.



FIGUR 3.2 TALOMRÅDE FÖR FYRABITARS TVÅKOMPLEMENTSTAL OCH TAL UTAN TECKEN

Ett annorlunda sätt att grafiskt visa tvåkomplements-representationen än som i figur 3.2 är med en så kallad *talcirkel*, figur 3.3. Figuren visar bitmönstren som lagras i datorn och vad dessa motsvarar om vi väljer att tolka dessa som tal med- respektive utan tecken.



FIGUR 3.3 TALCIRKEL FÖR FYRA BITAR.

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

Vi ställer nu upp några additionsexempel som visar hur de vanliga räknereglerna fungerar då vi tolkar bitmönstren som tal med eller som tal 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

3.6

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.

Låt 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

3.7

"spill" - uppstår då resultat av en aritmetisk operation ej kan anges för den givna talrepresentationen

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 med fyrabitars tvåkomplementsrepresentation (jfr tabell 3.6). 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, detta kallas *spill*.

Det finns fler sätt att överskrida talområdet, betrakta följande exempel.

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

3.8

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

3.9

Försök själv förklara varför det blir fel (använd figur 3.3).

Vi kan nu alltså, genom att använda *tvåkomplementsrepresentation*, 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 kunna veta om operationen har resulterat i spill?

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 undkomma problematiken har man infört *statusbitar* (eller *flaggbitar*) som ger kompletterande information om resultatet av operationen. Dessa flaggbitar sätts av den del av datorns centralenhet (ALU'n) som utför all aritmetik i datorn.

Flaggbitarna sätts enligt reglerna för två-komplements-aritmetik.

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 3.7 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 ALUn.

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

3.10

C-flaggan anger spill vid aritmetik på tal utan tecken.

C-flaggan har inte denna betydelse vid aritmetik på tal med tecken

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). Resultatet blir 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, om dessa siffror är `1 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 blir noll, indikerande att resultatet är korrekt för tal med tecken.

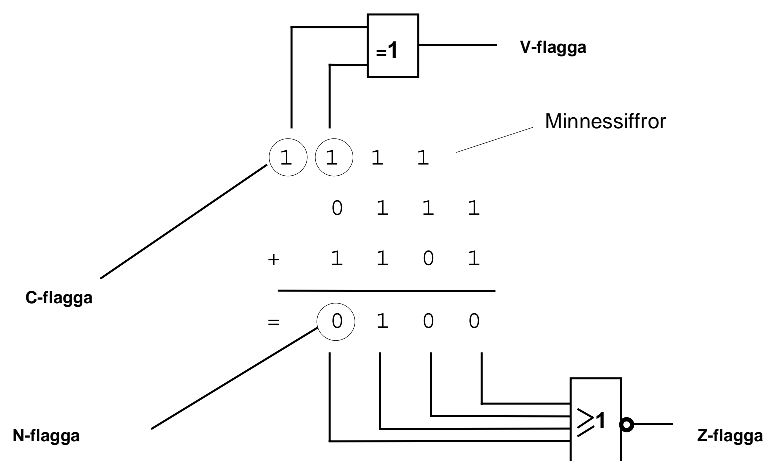
V-flaggan har ingen betydelse vid aritmetik på tal utan tecken.

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

Biten på position 3 i resultatets bitmönster (mest signifikanta biten) anger om talet är negativt eller inte, jämför med tabell 3.6. 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.

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

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



FIGUR 3.4 GENERERING AV FLAGGBITARNA

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 och får ett resultat med motsatt tecken så ettställs V-flaggan. Vid addition av två tal med olika tecken nollställs V-flaggan.

Adderas två positiva tal tecken (talens mest signifikanta bitar är 0) och resultatet blir negativt (resultatets mest signifikanta bit är 1) 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 mest signifikanta bitar (teckenbitar) och där *R* = *X*+*Y*. Då vi adderar tal med olika tecken kan spill inte uppträda.

Additionen i vårt föregående exempel gav alltså upphov till följande *flaggpåverkan*:

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

Studera nu nästa exempel.

EXEMPEL

4 bitars addition av %0111 och %0101

```
Bitmönstren tolkade     Bitmönster       Bitmönstren tolkade
som tal utan tecken                      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
```

3.11

Resultatet blir korrekt för tal utan tecken men fel för tal med tecken. 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 minnessiffrorna i position C och position 3 är olika, dvs V=1. Detta innebär att resultatet inte ryms för tal med tecken. Studerar vi de båda 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. Detta är ett annat sätt att konstatera att V-flaggan ettställs.

- Den mest signifikanta biten i resultatet är 0, alltså sätts N-flaggan till noll.
- Slutligen så ser vi att Z-flaggan är noll ty resultatets bitmönster innehåller minst en etta.

Additionen i exemplet ovan gav alltså upphov till följande flaggpåverkan:

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

Det är mycket viktigt att förstå hur flaggbitarna påverkas. Speciella, så kallade *villkorliga instruktioner* utförs *olika* beroende på flaggbitarnas värden. Vi återkommer till detta längre fram.

3.4 Implementering

Vi skall nu närmre studera hur operationer som addition, subtraktion, multiplikation och division kan utföras av en processor. Utgångspunkten är att processorn använder en ALU, med en fix ordlängd, som klarar av binär addition med flaggsättning enligt tvåkomplementsaritmetik så som beskrivits i tidigare avsnitt. Vi inleder med att visa hur vi hanterar situationer där ALUns ordlängd inte räcker. Därefter visar vi hur subtraktion kan utföras med hjälp av en additionsoperation. För att klara detta måste man kunna *negera* (byta tecken) på ett tvåkomplementstal, vi beskriver detta och hur subtraktion kan utföras. För att klara enkla multiplikations- divisions-algoritmer krävs dessutom att ALU'n kan utföra så kallat "skift" och vi visar därför hur detta går till. Avslutningsvis demonstreras några enkla algoritmer för multiplikation och division där endast additions- och skift- operationer ingår.

3.4.1 Lång addition

Som vi sett adderar processorns ALU bitmönstren som binära tal oberoende av om vi arbetar med tal med eller utan tecken. Efter en operation kan vi inspektera flaggbitarna **C** och **V** i processorns statusregister (eller flaggregister) för att undersöka om resultatet är korrekt eller inte.

Problem kan emellertid uppstå när vi vill addera större tal än de vår processor är konstruerad för. Vi fortsätter här att exemplifiera med en fyrabitars ALU och visar nu hur vi enkelt kan utvidga våra resonemang till tal med större ordbredd. Låt oss illustrera detta med följande exempel.

EXEMPEL

Addition av 8 bitars tal med 4 bitars ALU

Vi vill addera två st 8-bitars (R = X + Y) tal med enfyrabitars ALU. Vi gör då två additioner i sekvens, där den första additionen utför

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

sedan följer addition nummer två

$$R_{HIGH} = X_{HIGH} + Y_{HIGH} + \textit{Carry}\text{-flaggan}$$

(från föregående operation).

Vi inser vad som händer om vi sätter upp operationen 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}). 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 alltså talet negativt och N-flaggan satt.

3.12

Vanliga processorer har två *olika* additionsinstruktioner, en som adderar *med* C-flaggan och en som adderar *utan* C-flaggan.

3.4.2 Tvåkomplementering

Vi ska nu visa hur man enkelt kan *negera* (byta tecken på) ett tal med tecken. (Att negera ett tal utan tecken har förstås ingen mening).

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

3.13

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 -5 och fått resultatet 5, jämför med tabell 3.6.

EXEMPEL

Negera tvåkomplementstalet %0101

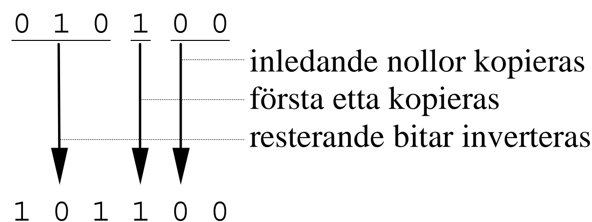
	Bitmönster	Bitmönstren tolkade som tal med tecken
Tal att negera	0101	+5
Invertera varje bit	1010	
Addera 1	+ 0001	
	= 1011	= -5

3.14

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



3.15

Negering av binära tal är en standardoperation som du bör lära dig utantill. Pröva själv med olika fyra bitars tal och jämför med tabell 3.6.

3.4.3 Subtraktion

Vid subtraktion får C-flaggan en "omvänd" betydelse (lånesiffra eller *Borrow*), betrakta följande exempel:

EXEMPEL

Utför subtraktionen X-Y där X=2 och Y=7

Subtraktionen X-Y utförs enligt:

Bitmönster	Bitmönstren tolkade som tal: *med* tecken	*utan* tecken
~~1~~0 ~~1~~0 ~~1~~0 ~~1~~0 10		
0 0 ~~1~~ 0	2	2
- 0 1 1 1	- 7	- 7
= 1 0 1 1	=-5	=11

3.16

Resultatet är korrekt för tal med tecken men kan inte representeras av tal utan tecken. Observera speciellt hur vi här tvingas låna av en (fiktiv) siffra till vänster om den mest signifikanta positionen, vi säger då att operationen genererar *Borrow*. Detta är helt enkelt motsvarigheten till *Carry* vid addition, dvs en spillindikator för tal utan tecken. Man kan då förledas tro att ALU'n måste förses med ytterligare en flagga för att indikera spill vid subtraktion. Detta är dock inte nödvändigt och vi ska strax visa varför det är så.

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=2 och Y=-7

Först tvåkomplementeras (negeras) Y enligt:

	Bitmönster	Bitmönstren tolkade som tal med tecken
Tal att negera	0111	+7
Invertera varje bit	1000	
Addera 1	+ 0001	
	= 1001	= -7

Sedan utförs additionen X+ (-Y) enligt

Bitmönster	Bitmönstren tolkade som tal med tecken
0 000	
0010	2
+ 1001	+ -7
= 1011	= -5

3.17

$\overline{Carry} = Borrow$

$Borrow = \overline{Carry}$

Resultatet är, precis som tidigare korrekt för tal med tecken, men naturligtvis fel för tal utan tecken. Observera dock att minnessiffran i *position 4* (dvs, den bit som utgör Carry-flaggan vid addition) här blir 0, vilket skulle kunna få oss att tro att resultatet *är* korrekt för tal utan tecken trots att det uppenbarligen är fel.

då subtraktion utförs genom addition av tvåkomplement.

Detta gäller alltid!!

Orsaken är att då vi gör subtraktionen på detta sätt, alltså genom att tvåkomplementera en operand och därefter *addera*, får C-flaggan en speciell betydelse, den sätts till *inversen* av den vänstra minnessiffran. Vid denna subtraktion representerar C-flaggan en lånesiffra (*Borrow*) och flaggbitarna blir således:

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

Precis som vid addition uppstår problem när vi vill subtrahera större tal än de vår processor är konstruerad för. Vi löser dock detta på samma principiella sätt som vid addition.

EXEMPEL

Subtraktion av 8-bitars tal med 4-bitars ALU

Vi vill subtrahera två st 8-bitars (R = X - Y) tal med en fyrabitars ALU. Vi gör då två subtraktioner i sekvens, där den första additionen utför

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

sedan följer subtraktion nummer två

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

(från föregående operation).

Vi ser vad som händer om vi sätter upp operationen på följande sätt:

	00111 ◄──	**1**100 ← minnessiffror
X_{HIGH} X_{LOW}	0011	0100
$-Y_{HIGH}$ Y_{LOW}	+0010	+1101
$=R_{HIGH}$ R_{LOW}	=0110	0001

3.18

Observera hur C-flaggan från den första operationen måste propagera till den andra operationen. Vanliga processorer har två olika subtraktionsinstruktioner, en som adderar C-flaggan och en som inte adderar C-flaggan.

3.4.4 Vänsterskift (multiplikation med 2)

Om bitmönstret *skiftas* ett steg åt vänster, och en nolla läggs i den minst signifikanta positionen innebär detta 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

3.19

Vid ett vänsterskift fyller vi på med nollor från höger. Betraktar 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, är den däremot ett innebär detta att spill uppstått.

För tal med tecken ser vi att spill uppstår när siffran i position tre byter tecken. Vi har då multiplicerat ett positivt tal med två och fått ett negativt resultat. V-flaggan ska i sådant fall ettställas. 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.

EXEMPEL

Ett 8-bitars tal, X, ska multipliceras med 2

Vi gör 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.)

Vi ser vad som händer om vi sätter upp räknestycket på följande sätt:

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

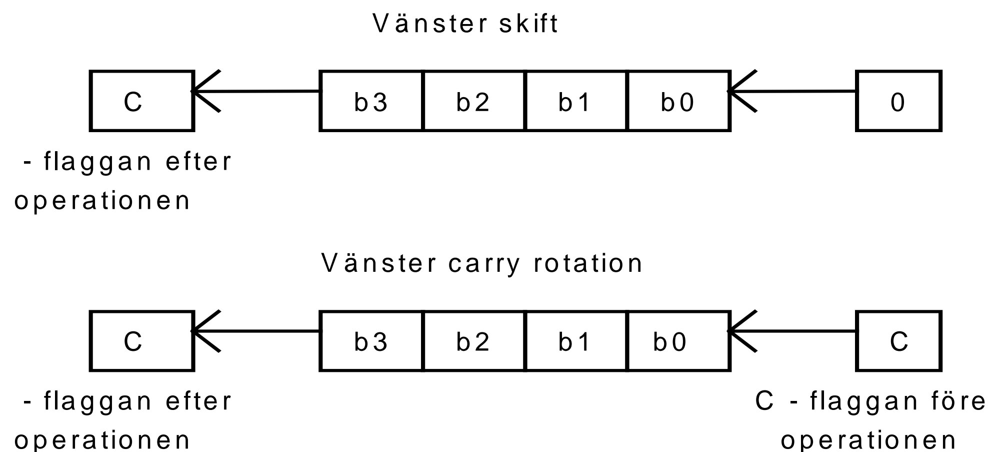
3.20

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.

Processorer är vanligtvis försedda med två *olika* vänsterskift-instruktioner, en som skiftar in en nolla från höger (*skift*) och en som skiftar in C-flaggan (*Carry-rotation*). Betrakta figur 3.5, 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 alltså 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)

Då bitmönstret skiftas ett steg till höger innebär detta att talvärdet divideras med 2. Beroende på att den mest signifikanta biten (tecknenbiten) 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

3.21

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. (Detta är inte fallet i exemplets sista skift där tre dividerat med två blir ett). Den högra biten skiftas till C-flaggan som på detta sätt indikerar spill.

Vi har en annorlunda situation om bitmönstret 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 högerskift av tal med tecken. Instruktionen måste skifta in en kopia av siffran i den mest signifikanta positionen.

EXEMPEL

Högerskift, tal med tecken

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

3.22

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

Vi utför division av tal som är större än vår ALU genom att kombinera olika skiftinstruktioner.

EXEMPEL

Vi vill dividera ett 8-bitars (R = X/2) tal med två i en fyrabitars ALU.

Vi utför två skiftningar i sekvens, där det första skiftet utför

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

sedan följer:

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

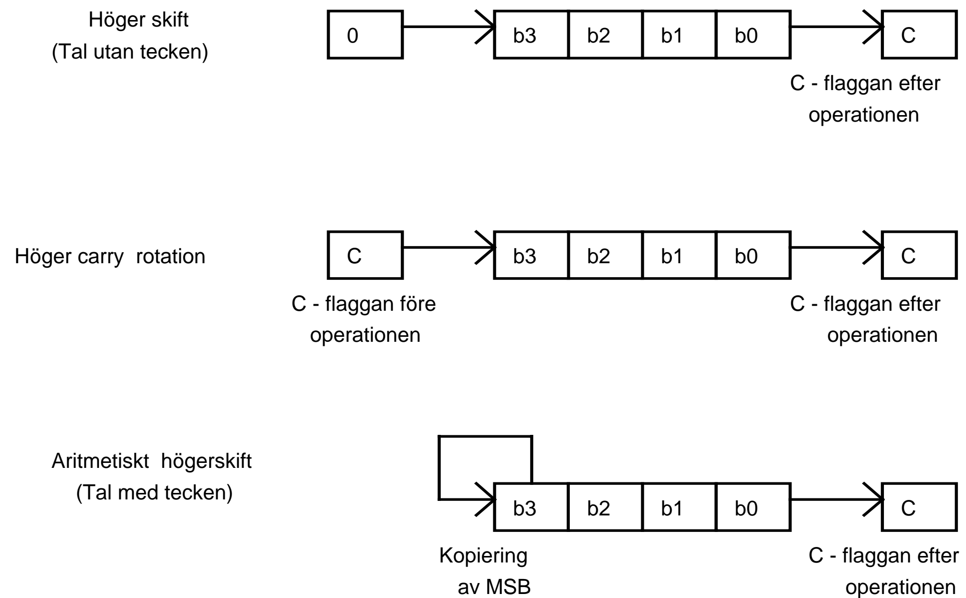
3.23

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 nedan. 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

3.4.6 Multiplikation

Multiplikation av binära tal kan utföras på samma sätt som vid decimala tal. Allmänt skriver vi P=X*Y där:

- P är produkten av multiplikationen
- X är multiplikator
- Y är multiplikand

Multiplikation i binära talsystemet

```
0 * 0 = 0
0 * 1 = 0
1 * 0 = 0
1 * 1 = 1
```

Det finns åtskilliga algoritmer för binär multiplikation. I huvudsak skiljs dom åt genom sin snabbhet. Vi kommer inte här att ge en fullständig framställning

Reglerna för multiplikation i binära talsystemet är mycket enkla, se marginalen. Vi tillämpar dessa och samma metod som då vi ställer upp, och multiplicerar, decimala tal. (I bland kallas detta “papper och penna-metoden”).

EXEMPEL

Multiplikation (P=X*Y), X=5, Y=6, "papper och penna-metoden"

```
            Decimalt      Binärt

      X         5             0101
     *Y       * 6            *0110
     =P       = 30            0000
                             0101
                            0101
                          + 0000
                          =   0011110=2^4+2^3+2^2+2^1=30
```

3.24

Om vi speciellt betraktar uppställningen av binärmultiplikation i exemplet ser vi att samma resultat kan fås enbart genom att använda operationerna addition och högerskift. Med denna metod bildas produkten genom en iterativ procedur som resulterar i partialprodukter (PP). Algoritmen kan kortfattat beskrivas som:

> Partialprodukt 0, PP(0) = 0
>
> med början på multiplikatorns LSB (x_0)
>
> för varje bit *i* hos multiplikatorn
>
> > Om x_i=1 addera Y till nästa partialprodukt
> >
> > annars addera 0 till nästa partialprodukt
>
> tills alla bitar inspekterats

EXEMPEL

Multiplikation (P=X*Y), X=5, Y=6,med användning av addition/skift

```
X=5=x3x2x1x0  =0101
Y=6=          =0110

PP(0)           0000          x0=1⇒ADD Y
              + 0110
                0110          skifta
PP(1)           0011 0        x1=0⇒ADD 0
              + 0000
                0011 0        skifta
PP(2)           0001 10       x2=1⇒ADD Y
              + 0110
                0111 10       skifta
PP(3)           0011 110      x3=0⇒ADD 0
              + 0000
                0011 110      skifta

P=PP(4)=        00011110 = 2^4+2^3+2^2+2^1=30
```

3.25

Den beskrivna algoritmen fungerar som synes för positiva tal. Vi har också all anledning att tro att algoritmen fungerar även om Y är ett negativt tal (på tvåkomplementform). Vi måste dock modifiera algoritmen och använda aritmetiskt skift för att inte förlora någon teckenbit.

EXEMPEL

Multiplikation (P=X*Y), X=5, Y=-6,med användning av addition/skift

```
X=5=x3x2x1x0  =0101
Y=-6=         =1010

PP(0)           0000         x0=1⇒ADD Y
              + 1010
                1010         skifta aritmetiskt
PP(1)           11010        x1=0⇒ADD 0
              + 0000
                11010        skifta aritmetiskt
PP(2)           111010       x2=1⇒ADD Y
              + 1010
               1100010       skifta aritmetiskt
PP(3)           1100010      x3=0⇒ADD 0
              + 0000
                1100010      skifta aritmetiskt

P=PP(4)=        11100010=-(00011110) = -30
```

3.26

Anmärkning: Vid additionen av PP(2) till Y ser vi att vi får spill från den mest signifikanta positionen. Med de givna förutsättningarna (Y<0) sker detta endast då vi har en partialprodukt som är mindre än 0. Spillet kan alltså ses som en kopia av teckenbiten. Eftersom vi skiftar aritmetiskt återställs denna på korrekt sätt.Den använda algoritmen måste dock modifieras om X<0, då vi bildar den sista partialprodukten, dvs slutprodukten, markerar talet X's teckenbit att -Y ska adderas (i stället för Y). Detta kan enkelt visas i det allmänna fallet (vi gör det inte här). Vi har då algoritmen för:

Robertsons metod

P=X*Y X är ett n-bitars tal, Y godtyckligt antal bitar

X och Y är tvåkomplementstal

PP(0) = 0

för k=0,1,2, ...,n-2

$PP(k+1) = (PP(k)+Y X_k)$

för k= n-1

$P=PP(n)=(PP(k)-Y X_{n-1})$

Observera att tillfälligt spill kan uppkomma då algoritmen tillämpas. Resultatet återföres dock till det tillåtna talområdet av det efterföljande skiftet. För att åstadkomma detta måste man dock införa en extra teckenbit (kopia) som används under uträkningarna men som ignoreras i resultatet.

Produkten av två godtyckliga tvåkomplementstal med *i* och *j* bitar kan maximalt vara (i+j-1). Detta inses av att *i* bitar ska skiftas *j* gånger (totalt i+j bitar), men samtidigt krävs endast en teckenbit för produkten (dvs i+j-1). Vanligtvis gäller att ordlängden hos multiplikator och multiplikand är densamma, $i=j=n$. Alltså krävs maximalt $2n$-1 bitar för att representera produkten av två n-bitars tal.

3.4.7 Division

Precis som för multiplikation, kan binär division utföras enligt “papper och penna-metod”. I detta avsnitt ska vi illustrera hur detta går till. Metoden kan endast användas på positiva heltal. Vi behandlar inte metoder för division av tvåkomplementtal i det generella fallet även om sådana metoder naturligtvis finns.

En division kan skrivas som:

$$\frac{X}{Y} = Q + \frac{R}{Y}$$

Där:

- X är dividend
- Y är divisor
- Q är kvot
- R är resten

Av sambandet framgår att resten kan uttryckas:

$$R = X - QY$$

Av exempel 3.27 nedan ser vi hur partialresterna bildas genom att succesivt subtrahera (största möjliga) kvotsiffra multiplicerad med Y som fortfarande ger en *positiv* partialrest. Kvotsiffran fås alltså genom *prövning*.

EXEMPEL

Decimal division, återställning av resten, 3967/15

X = 3967
Y = 15

```
                       R  =  X  - Q*Y
   0264,4
15 |3967,0  3697=3697-0*15                              Utgångsläge
  -0
   3967,0  3967=3967-(0*10^3)*15                           steg 1
  -30
    967,0   967=3697-(0*10^3+2*10^2)*15                    steg 2
  - 90
     67,0    67=3697-(0*10^3+2*10^2+6*10^1)*15             steg 3
  -  60,0
      7,0     7=3697-(0*10^3+2*10^2+6*10^1+4*10^0)*15      steg 4
  -   6,0
      1,0     1=3697-(0*10^3+2*10^2+6*10^1+4*10^0+4*10^-1)*15 steg 5

DVS    3967/15 = 264,4 + 10/15 * 10^-1
```

3.27

Låt oss nu studera ett exempel på binär division med användning av samma metod. Som vi ska se blir detta faktiskt enklare eftersom endast två kvotsiffror (0 och 1) kan förekomma. Prövningen av Q*Y för positiv partialrest kräver inte någon egentlig multiplikation.

EXEMPEL

Binär division X/Y, med återställning, X=13, Y=5

X = %1101
Y = %0101

```
                    R  =  X  - Q*Y
       0010
0101|1101       1101=1101- 0*0101                        Utgångsläge
  -0000
       1101     1101=1101-(0*2^3)*0101                        steg 1
   -0000
       1101     1101=1101-(0*2^3+0*2^2)*0101                  steg 2
    -0101
       0011       11=1101-(0*2^3+0*2^2+1*2^1)*0101            steg 3
     -0000
       0011       11=1101-(0*2^3+0*2^2+1*2^1+0*2^0)*0101      steg 4

DVS    1101/0101 = 0010 + 0011/0101 (2 + 3/5)
```

3.28

Den visade metoden kan formuleras i en algoritm där vi endast använder operationerna vänsterskift och addition. Efter subtraktion av Y från aktuell partialrest sätts kvotsiffran till 1 om skillnaden blir positiv, om skillnaden blir negativ sätts kvotsiffran till 0 och partialresten *återställs* genom addition av Y, därefter fortsätter man med nästa steg. Om partialresten blir 0 har divisionen "gått jämnt ut" och operationen är klar. Ofta är detta inte fallet och man kan då fortsätta om man är villig att acceptera decimalsiffror i svaret (jämför med föregående exempel).

Algoritm: Division med återställning

$R = X - Q*Y$

$Q=q_0q_1q_2...q_{n-1}$

n=antal kvotbitar att beräkna

$R_0 = X$

$R_1 = R_0-q_0*Y$ $\quad$ $q_0=1$ om $R_1 \geq 0$, $q_0 = 0$ annars

för i = 2..n

$R_i=2*R_{i-1} -q_i*Y$ $\quad$ $q_i=1$ om $R_i \geq 0$, $q_i = 0$ annars

Anmärkning: Antalet kvotbitar vi önskar bestämmer antalet partialrester som ska beräknas och därmed också det antal vänsterskift som måste utföras. Eftersom partialrest *n* inte ska skiftas får vi *n-1* vänsterskift.

EXEMPEL

Utför binär division av 13/5 med återställningsmetod.
Svaret ska anges med 4 kvotbitar och 4 restbitar. Varje steg i algoritmen ska redovisas.

Lösning:

X=13 = 1101
Y=5 = 0101
-Y=-5= 1011
n=4 (3 skift ska utföras)

Utgångsuppställningen:

```
R0 = X        0001101 ← 3 bitar ska skiftas
+(-Y)         1011
```

steg 1:

```
R0-Y          1100101     <0⇒q0=0⇒Återställ
+(Y)          0101
R1            0001101
2* R1  0011010
```

steg 2:

```
        +(-Y)       1011
        2*R1-Y 1110010     <0⇒q1=0⇒Återställ
        +(Y)        0101
        R2          0011010
        2* R2  0110100
```

steg 3:

```
        +(-Y)       1011
        2*R2-Y 0001100     ≥0⇒q2=1
        R3          0001100
        2* R3  0011000
```

steg 4:

```
        +(-Y)       1011
        2*R3-Y 1110000     <0⇒q3=0⇒Återställ
        +(Y)        0101
        R4          0011000
```

n=4 och algoritmen terminerar här

Q= $q_0q_1q_2q_3$ =%0010=2
R= =%0011=3
DVS: 13/5 = 2 + 3/5

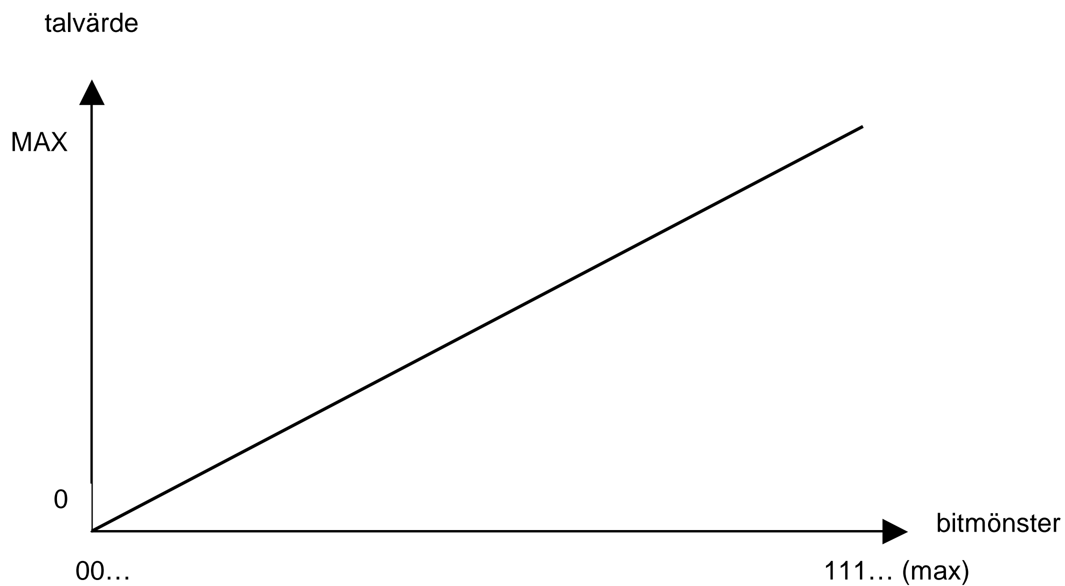
3.29

Det finns åtskilliga olika metoder att utföra såväl binär multiplikation som binär division, vi kan dock inte, inom ramen för detta läromedel behandla fler. Vi kan dock redan här notera komplexiteten i att utföra dessa operationer, det är avsevärt mycket mer tidsödande att till exempel utföra en division med 2 enligt någon divisionsalgoritm än att utföra samma operation med en enkel skiftinstruktion.

3.5 Förskjuten binärkod

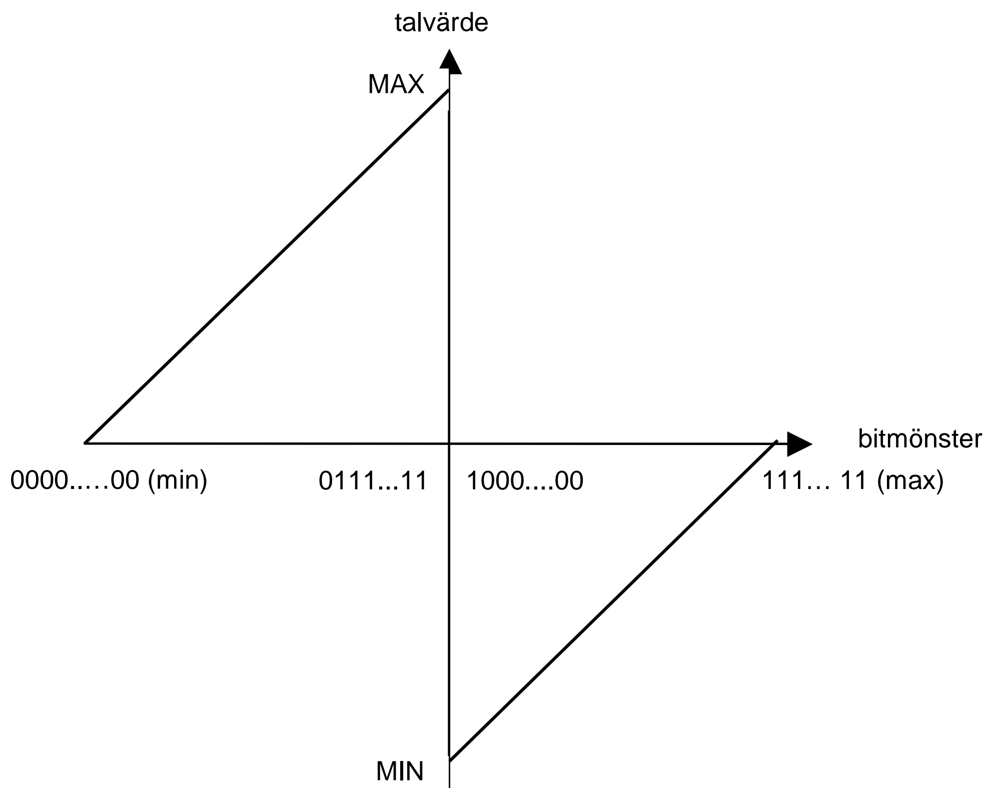
Vi har tidigare diskuterat några olika sätt att representera heltal med hjälp av binär kodning. Vi kan illustrera dessa sätt *grafiskt* genom att rita en funktion som anger sambandet mellan binärt tal (bitmönster) och talvärde. En sådan funktion är alltså unik för den använda representationen.

Talvärdet som funktion av bitmönster vid kodning enligt det binära talsystemet blir:



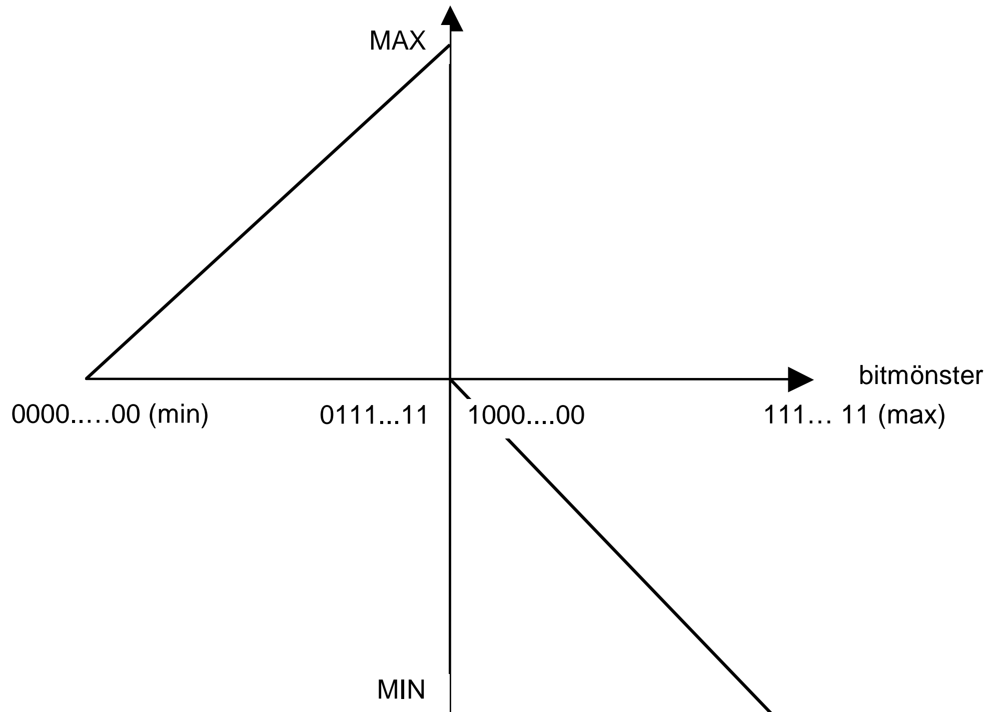
FIGUR 3.7 ILLUSTRATION AV TALVÄRDE VID BINÄR KODNING

På motsvarande sätt blir talvärdet som funktion av bitmönster vid kodning enligt tvåkomplementsformen:



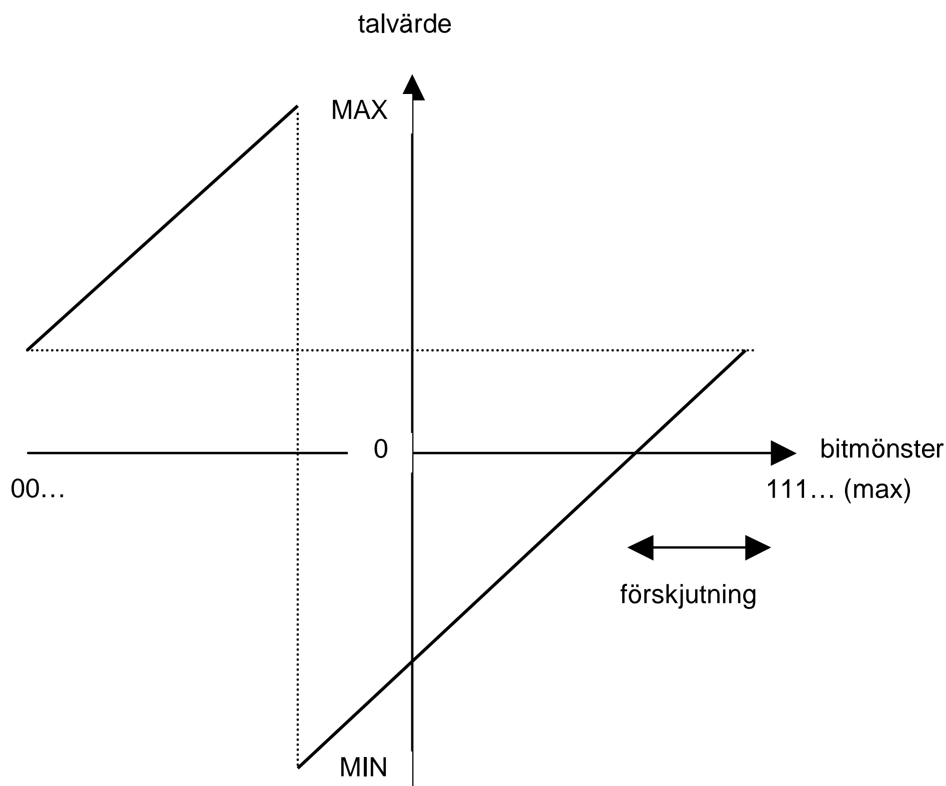
FIGUR 3.8 ILLUSTRATION AV TALVÄRDE VID TVÅKOMPLEMENTSFORM

Vid tecken-beloppsrepresentation får vi följande funktion.



FIGUR 3.9 ILLUSTRATION AV TALVÄRDE VID TECKEN/BELOPPSFORM

En viktig kodningsform är binär förskjutning (*offset binary coding*) eller *excess-n* kod. Metoden utgår ifrån kodning av tal på tvåkomplementsform men man flyttar helt enkelt talvärdet 0 till utefter x-axeln.



FIGUR 3.10 ILLUSTRATION AV TALVÄRDE VID BINÄR OFFSET

Talvärdet vid *excess-n* kodning fås alltså enkelt genom att förskjutningen *n* adderas till talvärdet för tvåkomplements kodning. Speciellt gäller att talvärdet för excess-0 kodning är densamma som för tvåkomplement.

Formen exess (2^{n-1}) för *n*-bitars tal är speciellt intressant. Den ger oss en rätlinjig kontinuerlig funktionsgraf av tal som ursprungligen kodats i tvåkomplementsform.

EXEMPEL

För 8-bitars bitmönster och *excess-128* kodning bestäms talvärdena enligt följande:

Bitmönster	Decimalform	Excess(128) talvärde = Excess(0) talvärde + 128
00000000	0	(00000000)E128 = 128
10000000	-128	(10000000)E128 = 0
11111111	-1	(11111111)E128 = 127

3.30

Förskjuten binärkod används ofta av AD-omvandlare men, som vi strax ska kommer den också till användning i samband med representation av flyttal.

3.6 Flyttal

Vi har tidigare uteslutande behandlat kodning av heltal. I många sammanhang är detta dock inte tillräckligt exempelvis då vi vill uttrycka mycket små tal så som avstånd mellan atomer eller vi vill uttrycka mycket stora tal så som avstånd mellan galaxer i universum. Redan i den inledande behandlingen av olika talsystem antydde vi att en bitsträng kan förses med en ”tänkt” binärpunkt och tolkningen av talvärdet göras därefter. Vi kallar detta allmänt för *fixtal* och menar då en fast binärpunkt någonstans i bitsträngen.

EXEMPEL

Bitsträngen 11011100 kan förses med en ”tänkt” binärpunkt enligt 1101.1100, tolkningen av talet blir då:

$$2^3+2^2+2^0+2^{-1}+2^{-2} = 8+4+1+0{,}5+0{,}25 = 13{,}75$$

3.31

Fixtals aritmetik kan vara användbart i vissa sammanhang men det löser dock knappast det grundläggande problemet dvs att, med samma representation, kunna ange såväl mycket små som mycket stora tal. För att komma till rätta med detta kan vi införa en *flytande* binärpunkt, dvs information om var, i talet, binärpunkten är placerad finns inkodat i talet.

EXEMPEL

Talet 132 kan skrivas om som en produkt av två tal enligt:

$132\ 10^0 = 13{,}2\ 10^1 = 1{,}32\ 10^2$ osv..

vi säger att vi delat upp talet i mantissa och exponent. Under förutsättning att exponentens bas är den samma som mantissans radix gäller att exponenten anger decimalkommats placering i talet.

3.32

Om det gäller att mantissan är i intervallet

$$\frac{1}{r} \leq M < 1$$

sägs den vara *normaliserad.*

EXEMPEL

Uttryck talet 132 på normaliserad form.

Av föregående exempel framgår att det finns åtskilliga sätt att uttrycka ett tal, det finns dock endast en normaliserad form nämligen:

$0{,}132\ 10^3$

ty endast denna form uppfyller:

$0{,}1 \leq 0{,}132 < 10$

3.33

Ovanstående resonemang kan naturligtvis tillämpas oberoende av talsystem.

EXEMPEL

Uttryck

a) $\mathbf{(1101.011)_2}$

b) $\mathbf{(1E.0A)_{16}}$

på normaliserad form.

Lösning:

Skriv talen som mantissa och exponent:

a) $(1101.011)_2 = (1101.011)_2\ 2^0$

b) $(1E.0A)_{16} = (1E.0A)_{16}\ 16^0$

”Flytta” binärpunkten så att mantissan uppfyller vilkoret för normaliserad form. För varje steg vi flyttar binärpunkten till vänster adderar vi 1 till exponenten.

a) $(1101.011)_2 = (0.1101011)_2\ 2^4$
b) $(1E.0A)_{16} = (0.1E0A)_{16}\ 16^2$

Eftersom talen uttrycks på normaliserad form vet vi var binärpunkten är placerad och kan därför utelämna denna:
a) $(1101.011)_2 = (1101011)_2\ 2^4$ (normaliserat)
b) $(1E.0A)_{16} = (1E0A)_{16}\ 16^2$ (normaliserat)

3.34

Ett flyttal uttrycks allmänt som:

$$(-1)^S\ M\ 2^E$$

där:

S (*sign*) är teckenbiten för talet

S=0 anger ett positivt tal ty (-1)0 =1

S=1 anger ett negativt tal ty (-1)S = -1

M (*fractional part*) utgör talets mantissa

E (*exponent*) utgör talets exponent.

EXEMPEL

Skriv talet $(2,52)_{10}\ 10^4$ på formen $(M)^2\ 2\ (E)^2$ normaliserat

Lösning:
Omvandla först hela talet till binär form på känt sätt:

$2,52\ 10^4 = 25200 = (1100\ 0100\ 1110\ 000)_2$

normalisera talet:

$=(0.1100\ 0100\ 1110\ 000)_2\ \ 2^{15}$

3.35

3.6.1 IEEE – flyttalsstandard

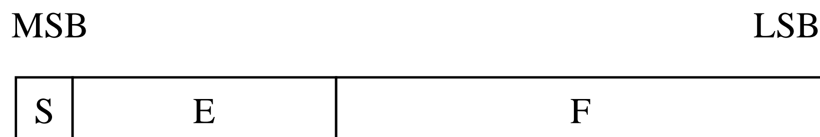
IEEE-flyttalsstandard specificerar

- flyttalsformat
- noggrannhet i resultat från aritmetiska operationer
- omvandling mellan heltal och flyttal
- omvandling till/från andra flyttalsformat
- avrundning
- undantagshantering vid operationer på flyttal, exempelvis division med 0 och resultat som ej kan representeras av flyttalsformatet.

Standarden definierar fyra olika flyttalsformat:

- *Single format*, totalt 32 bitar
- *Double format*, totalt 64 bitar
- *Single extended format*, antalet bitar är implementations-beroende
- *Double extended format*, totalt 80 bitar

Ett IEEE-flyttalsformat delas upp i tre fält enligt följande:



där:

- **F** (*fractional part*) kallas också *signifikand*, är den normaliserade mantissan * 2, dvs den första (implicita) ettan i mantissan utelämnas i representationen och mantissan skiftas ett steg till vänster. På så sätt uppnår vi ytterligare noggrannhet eftersom vi får ytterligare en siffra i det lagrade talet.
- **E** (*exponent*) är exponenten uttryckt på *excess(n)* format, n beror på vilket av de fyra formaten som avses.
- **S** (*sign*) är teckenbit för F.

Följande tabell anger hur de olika formaten disponeras anligt standarden:

Format	**S**	**E**	**F**
Single	1 bit	8 bitar excess(127)	23 bitar
Double	1 bit	11 bitar excess(1023)	52 bitar
Extended	1 bit	15 bitar excess(2047)	64 bitar

EXEMPEL

Skriv talet $(2{,}52)_{10}\ 10^4$ som ett IEEE-single format flyttal

Lösning:
Vi har tidigare kommit fram till resultatet:
$(2{,}52)_{10}\ 10^4 = (0.1100\ 0100\ 1110\ 000)_2\ \ 2^{15}$

Betrakta nu F som vi får ur den normaliserade mantissan om vi skiftar talet ett steg till vänster och fyller på med nollor efter den minst signifikanta siffran så att F totalt får 23 bitar:

F = 1000 1001 1100 0000 0000 000

Exponenten uttrycks som excess(127) kod dvs:

E' = E+127

I den normaliserade ursprungsformen hade vi 2^{15} (exp.=15)
och får nu (eftersom vi skiftat F ytterligare ett steg)
E=14 varför E' = 141 = $(1000\ 1101)_2$

eftersom talet är positivt får vi
S = 0

Vi sammanställer nu resultatet i 32-bitars form (single) och får:

$(2{,}52)_{10}\ 10^4 = (0100\ 0110\ 1100\ 0100\ 1110\ 0000\ 0000\ 0000)_{SFP}$

3.36

3.6.2 Aritmetiska operationer på flyttal

Aritmetiska operationer på flyttal blir avsevärt mer komplicerade än operationer på heltal. Anledningen är att operationerna måste utföras i flera steg.

3.6.2.1. Flyttalsaddition/subtraktion

Vid addition eller subtraktion av flyttal utförs egentligen ett flertal operationer där mantissa och exponent måste behandlas var för sig, dessutom tillkommer teckenöverläggning eftersom mantissan ju är given på tecken-beloppsform. Vid operation på mantissorna måste vi först se till att båda talen har samma exponent. Addition/subtraktion av IEEE-flyttal beskrivs av följande algoritm:

1. Bestäm talens mantissor ur F (dvs lägg till en etta framför den mest signifikanta biten i respektive F).
2. Bestäm talens exponenter på tvåkomplementsform.

3. Beräkna exponentskillnaden
4. Skifta mantissan för talet med *minst* exponent *höger* det antal gånger som exponentskillnaden anger (mins att exponenten anger binärpunkten i flyttalet).
5. Utför addition (subtraktion) av mantissorna efter tecken-överläggning.
6. Normalisera resultatet genom att skifta resultatmantissan samtidigt som resultatexponenten korrigeras.

EXEMPEL

Visa additionen av 2,52 10^4 + 2,52 10^3 av flyttal givna enligt IEEE-single format form:
Lösning:
Omvandla talen till binärformat:
A=$(2,52)_{10}$ 10^4=$(01000110110001001110000000000000)_{SFP}$
B=$(2,52)_{10}$ 10^3=$(01000101000111011000000000000000)_{SFP}$

1. och 2. Dela upp talen i tecken, exponent och mantissa:
A = $(-1)^{S_A}$ * MA * EA och B = $(-1)^{S_B}$ * MB * EB

$S_A = 0$
$M_A = (1.F)_A =$ (1.10001001110000000000000)
$E_A = E'_A - 127 =$ (00001110) (dvs 141-127 = 14)

$S_B = 0$
$M_B = (1.F)_B =$ (1.00111011000000000000000)
$E_B = E'_B - 127 =$ (00001011) (dvs 138-127 = 11)

3. Exponentskillnaden (14-11) är 3.
4. Skifta talet med minst exponent (M_B) tre steg höger

M_B = (0.00100111011000000000000000)
observera att vi tillåter större upplösning hos mantissan under utförande av operationen.

5. Utför additionen, båda talen positiva:
 1.10001001110000000000000
+0.00100111011000000000000000
=1.10110001001000000000000

6. Normalisera resultatet, i detta fall är resultatet redan i normaliserad form:
$S_R = 0$
M_R = 1.10110001001000000000000
$E_R = 14$
Vilket nu ger oss representationen:
F_R = (10110001001000000000000)
$E_R' = 127+14 = (10001101)_2$
Vi kan slutligen sätta samman resultatet:
R = $(01000110110110001001000000000000)_{SFP}$

3.37

3.6.3 Avrundningsmetoder

Observera speciellt att operationen kan utföras med större precision än formatet tillåter. Som konsekvens av detta kan vi i bland tvingas avrunda resultatet. Standarden tillåter fyra olika avrundningsmetoder:

- *Round to nearest*, resultatet avrundas till det representerbara värdet som ligger närmst det verkliga värdet. Om det verkliga värdet ligger mitt emellan två representerbara värden avrundas resultatet till ett jämnt tal.
- *Round to zero*, resultatet trunkeras till rätt precision, dvs avrundningsbitarna ignoreras.
- *Round towards minus infinity*, resultatet avrundas nedåt till närmsta representerbara värde.
- *Round towards plus infinity*, resultatet avrundas uppåt till närmsta representerbara värde.

3.6.4 Flyttalet noll

Det finns inget sätt att ange talet noll på normaliserad form. Standarden säger i stället att såväl exponent som mantissa här ska vara noll. Observera att detta ger frihet att koda såväl plus som minus noll beroende på hur flyttalets teckenbit sätts.

3.6.5 Denormaliserade tal

Denormaliserade tal uppträder som resultat av operationer där exponenten vid normalisering av resultatet inte kan minskas längre, dvs man uppnått den (till beloppet) största negativa exponent som kan representeras. Detta innebär att mantissan inte kan normaliseras. I stället för att avrunda ett sådant resultat tillåter här standarden ett icke-normaliserat resultat. Det denormaliserade talet indikeras genom att exponenten sätts till noll men mantissan är skild från noll.

3.6.6 Oändligheter

Då resultatet av en flyttalsoperation överskrider det representerbara talområdet, dvs exponenten anger ett tal som inte kan normaliseras indikeras detta som *infinity*. *Infinity* kan, precis som talet noll, ha tecken och det indikeras genom att samtliga exponentens bitar sätts till ett och mantissan sätts till noll.

3.6.7 Icke-tolkningsbara resultat

IEEE-standarden inbegriper också en klass av resultat kallade "NaN", (Not A Number). Detta utgör resultatet av en operation som inte har någon matematisk tolkning, exempelvis "oändlighet dividerat med oändlighet" eller "division med noll". Samtliga exponentens bitar sätts här till ett och mantissan är skild från noll. För en operation där minst en av operanderna är NaN produceras alltid resultatet NaN.

3.6.8 Flyttalsmultiplikation och Division

En flyttalsmultiplikation/division utförs betydligt enklare än addition och subtraktion. Vid flyttalsmultiplikation multipliceras mantissorna medan exponenterna adderas, vid flyttalsdivision divideras mantissorna medan dividendens exponent subtraheras från divisorns exponent, detta kan kortare skrivas som:

$A*B = 2^{(E_A+E_B)} * F_A * F_B$ respektive

$A/B = 2^{(E_A-E_B)} * F_A/F_B$

3.6.9 Flyttalstest och jämförelser

En test av ett IEEE-flyttal kan ge följande resultat:

- normaliserat
- denormaliserat
- plus noll
- minus noll
- negativt
- plus oändligheten
- minus oändligheten
- plus *Not A Number*
- minus *Not A Number*

Detta kan jämföras med de testresultat vi kan få då ett vanligt heltal testas (*Zero* eller *Negative*). En ALU för flyttalsaritmetik har därför ytterligare en uppsättning flaggbitar som är avsedda att återspegla de speciella resultat som fås vid en flyttalstest.

En flyttalsjämförelse ska, enligt standarden, kunna testa vilkoren:

- Equal To
- Greater Than
- Less Than
- Unordered

Tack vare kodningen blir dessa jämförelseoperationer enkla att implementera.

- *Equal To*, indikerar att operanderna är identiska
- *Greater Than/Less Than*, indikerar att operand A är större/mindre än operand B, detta inbegriper en teckenöverläggning och om talen har samma tecken kan resterande del av operanderna jämföras på samma sätt som vid heltalsjämförelse. Detta är en av fördelarna med att koda exponenten på *excess*-form i stället för tvåkomplementsform.
- *Unordered* innebär att minst ett av talen är "Not A Number".

3.6.10 Sammanfattning IEEE-flyttalsstandard

Låt oss som avslutning bestämma talområden och upplösning hos de olika IEEE-formaten

Single format, (32 bitar) har 23 bitars signifikand, 8 bitars exponent och 1 teckenbit. Det minsta normaliserade tal som då kan representeras är: E'=1 och F = 0 vilket ger

E = -126

M = 1.00.....0

dvs: $1.0 * 2^{-126} \cong 1,2\ 10^{-38}$

Det minsta *denormaliserade* tal som kan representeras får vi då E'=0 och endast den minst signifikanta biten i F är 1enligt

E = -126 och

M= 00000000000000000000001

dvs: $2^{-126} * 2^{-23} \cong 1,4\ 10^{-45}$

Det största normaliserade tal som kan representeras får vi då E'=254 och F = 11111111111111111111111, dvs

E = 127

M = 1. 11111111111111111111111

vilket ger:

$1.11111111111111111111111 * 2^{127} \cong 2 * 2^{127} \cong 3,4\ 10^{38}$

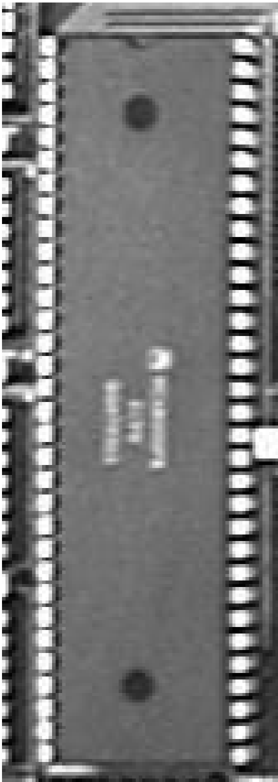
Upplösningen ges av värdet hos den minst signifikanta biten i F ty detta värde anger den minsta skillnad mellan två flyttal vi kan representera. Vikten hos denna bit ger oss alltså upplösningen enligt:

$2^{-23} \cong 1,19\ 10^{-7} = 0,000000119$

Mot bakgrund av att all aritmetik utförs med högre precision kan man utgå från att den första åtskiljande siffran alltid avrundas till ett korrekt värde. Vi ser då att vi kan utgå från att vi har minst 7 decimala siffror's noggrannhet.

MIKROPROCESSORN

I detta kapitel kommer vi att studera mikroprocessorn MC68000 i detalj. Inledningsvis ger vi en översikt av samtliga processorer ur 68000-familjen. Vi beskriver processorns arkitektur, med detta menas vilka register processorn har, vilka datatyper den kan arbeta med, vilka instruktionsformat som används, mm. Efter detta beskrivs processorns arbetssätt och här visas bland annat hur en enstaka busscykel utförs. Kapitlet avslutas med en detaljerad beskrivning av processorns anslutningar (pinnar) följt av ett schema över ett minimalt men ändock komplett mikrodatorsystem.

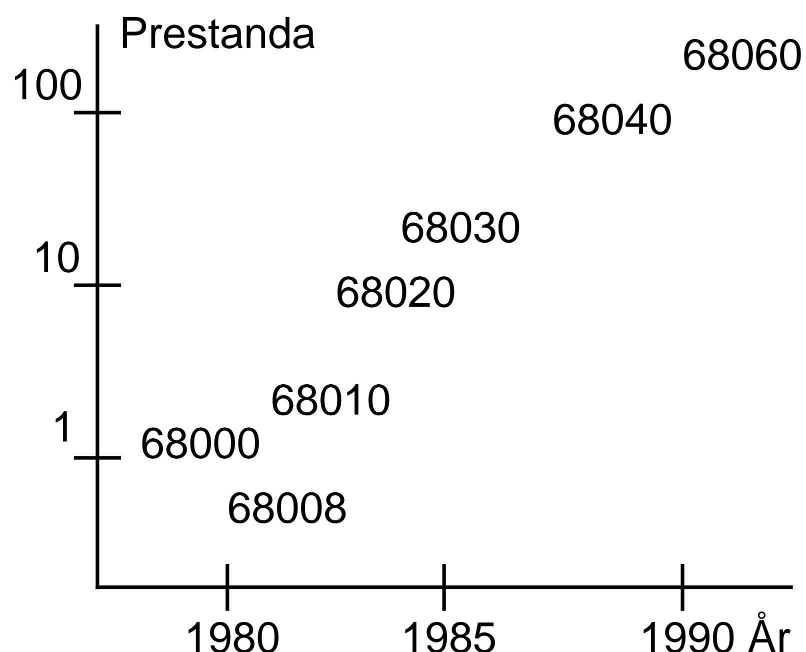


MC68000

Motorola MC68000 var den första mikroprocessorn i en lång rad processorer med liknande uppbyggnad. Man talar om "68k-familjen" och menar då såväl MC68000 som dess efterföljare.

MC68000, som introducerades 1979, blev snabbt populär och har genom åren använts i otaliga tillämpningar exempelvis laser-skrivare, *Macintosh*-datorer, *SUN*-arbetsstationer, *VME*-system (styr och regler utvecklingssystem för industrin), bara för att nämna några. MC68000 fick en rad namnkunniga efterföljare; MC68008, MC68010, MC68020, MC68030, MC68040 och MC68060, utvecklade i denna ordning, är kanske dom mest kända.

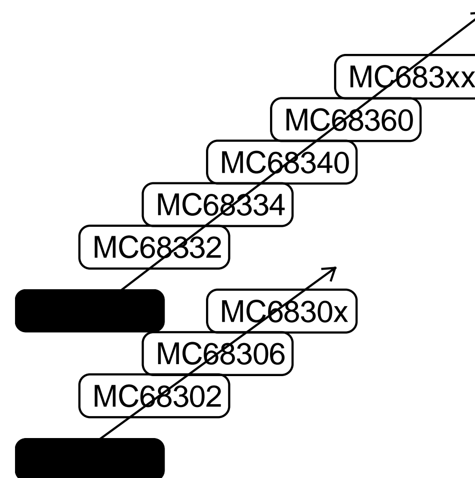
Man säger att dessa processorer tillhör “samma processorfamilj” eftersom dom fungerar på liknande sätt ur en programmerares synvinkel. Då nya processorer utvecklades kunde man lägga till nya egenskaper (dock inte ta bort några gamla). Följaktligen kunde program som utvecklats för "anfadern" MC68000 även utföras, som regel utan modifikation, av nyare och bättre efterföljare. Denna egenskap, att alla (maskin-) program som kan utföras av MC68000 även kan utföras av de övriga processorerna kallas *nedåt (program-) kompatibilitet* hos de senare processorerna. Motsatsen, *uppåt kombatibel*, gäller däremot som regel inte, ty de mer kraftfulla processorerna är utrustade med fler register och kan utföra fler instruktioner. De sist utvecklade (MC68040 och MC68060) karakteriseras av prestanda som man kanske inte ens vågade drömma om då MC68000 introducerades, dom är mer än två tiopotenser så kraftfulla, (se figur 4.1) klarar flera instruktioner och adresseringssätt och har dessutom inbyggda specialenheter för flyttalsaritmetik.



FIGUR 4.1 68K-FAMILJENS UTVECKLING

Beteckningen *68000* kommer ifrån att MC68000 innehåller ca 68000 transistorer. Jämför detta med över en miljon transistorer i MC68060. De första processorerna använde sig av en arbetstakt (*klockfrekvens*) på 4 MHz. De senaste kan användas med klockfrekvenser runt 100 MHz.

Vid sidan av processorer har Motorola även utvecklat ett antal kringkretsar (*peripheral interfaces*) som är anpassade till processorfamiljens bussystem. Exempel på kringkretsar kan vara parallell I/O, räknare, seriell kommunikation, hårddiskanslutning, mm. Det finns också kretsar som utför flyttalsberäkningar och speciell minneshantering (*coprocessorer*). Hårdvarumässigt är alla dessa kretsar mycket enkla att ansluta till processorns bussar vilket är ännu ett skäl till varför processorfamiljen blivit så populär.



FIGUR 4.2 OLIKA MICROCONTROLLERS

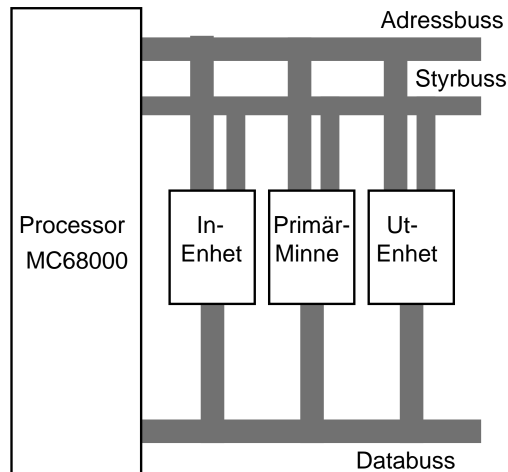
Motorola har även integrerat kringkretsar och processorer i så kallade *microcontrollers.* En microcontroller innehåller förutom processor exempelvis parallella in- och utportar, räknarkretsar, minne, etc. Figur 4.2 visar två grenar med microcontrollers. Den ena med en processorkärna baserad på MC68000 och den andra, med en processorkärna som kallas CPU32. CPU32 är en modifierad MC68020, dvs är bakåt kompatibel med MC68000 dock ej uppåt kompatibel med MC68020. Familjen microcontrollers med CPU32-kärna betecknas ofta MC68300-familjen. Inom MC68300-familjen finns ett antal medlemmar. Några är anpassade för seriell tele- och datakommunikation och innehåller ett flertal standardiserade kommunikationsprotokoll. Vissa är anpassade för motorstyrning och innehåller då en så kallad "*Time Processing Unt*", TPU. Exempelvis är MC68340 anpassad för persondatorändamål.

Orsaken till att denna processorfamilj blivit så populär är troligen MC68000:s *arkitektur*. När processorn lanserades i slutet på 70-talet så hade den (med dåtidens mått mätt) stora och många register. Detta har bidragit till att det har varit enkelt att konstruera mer kraftfulla efterföljare med bibehållen programkompabilitet.

MC68340

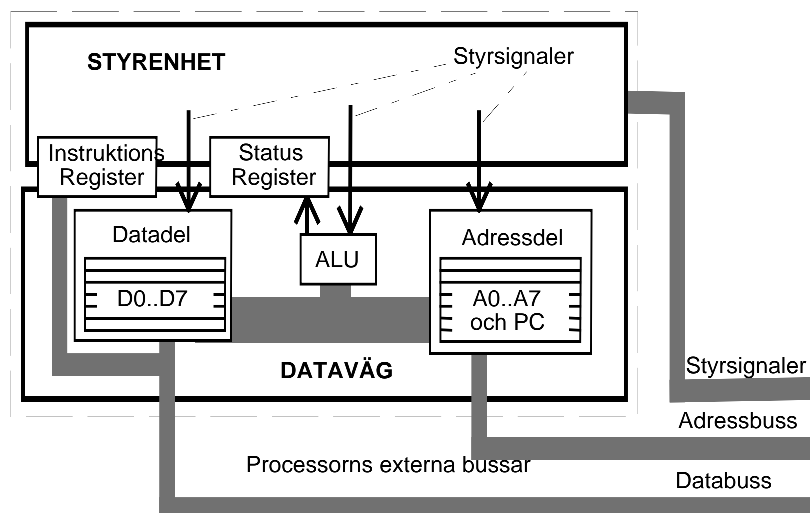
4.1 MC68000s interna block

Vi har i tidigare kapitel sett hur ett mikrodatorsystem består av bland annat processor, minne, och in / utportar (I/O). Dessa är sammankopplade med bussar som processorn styr för att hämta instruktioner från minnet och för att överföra data. Vi kan direkt sätta in MC68000 i figuren som visades redan i kapitel 2, se figur 4.3 som även visar bussarna var för sig.



FIGUR 4.3 MC68000 MED MINNE OCH I/O

Vi såg också hur en traditionell processor är internt uppbyggd av styrenhet, datadel och adressdel, så är också MC68000 uppbyggd (se figur 4.4). Observera att figuren också visar ett statusregister för bland annat flaggbitarna N, Z, V och C som vi beskrev i kapitlet om aritmetik.



FIGUR 4.4 MC68000 INTERNA BLOCK

Vi beskrev hur processorns arbetssätt var att oupphörligt hämta och utföra instruktioner. Under en hämtfas används innehållet i PC (programräknaren) för att adressera primärminnet så att en operationskod kan läsas från minnet till IR (instruktionsregistret). Processorns styrenhet avkodar denna och styr datavägen så att maskininstruktionen utförs

4.2 MC68000:s arkitektur

MC68000 kallas ibland *16/32 bit processor*. Med detta menas att processorn *externt* arbetar med 16 databitar men internt bearbetas 32 bitar åt gången. Alltså, den kan endast läsa in 16 bitar per *busscykel* fast processorns data och adressregister är 32 bitar långa. Den utför full 32 bitars aritmetik, exempelvis addition av innehållet i två 32-bitars dataregister.

Med en processors arkitektur *så menas bland annat*

- *hur många register den har*
- *hur många bitar registren innehåller*
- *vilka datatyper den kan arbeta med*
- *vilka operationer den kan utföra på datatyperna*
- *på vilka olika sätt den kan adressera minnet mm.*

Processorns ALU, data och adressregister ugör processorns *dataväg*. Datavägen styrs av *styrenheten* som i sin tur styrs av den *operationskod* (instruktion) som finns i *instruktionsregistret*. Innehållen i datavägens register kan överföras till processorns data och adressbuss. Dessa bussar är *asynkrona* och *icke-multiplexade*.

4.2.1 Programmerarens modell (bild)

När man talar om en processors arkitektur ingår begreppet *Programming Model*. Med detta menas (assembler-) programmerarens bild av processorn. Vad är det för kännedom han behöver ha om processorn? Inte är det om bussarna är synkrona eller asynkrona, (information som en hårdvarukonstruktör måste känna till) utan han vill veta vilka register som finns i datavägen och hur stora dessa är. Vidare måste han veta hur flaggorna från ALU:n påverkas och var han kan läsa denna information. Han måste veta om ALU:n arbetar med tvåkomplementstal eller inte. Slutligen vill han veta vilka operationer (maskininstruktioner) han kan få datavägen att utföra.

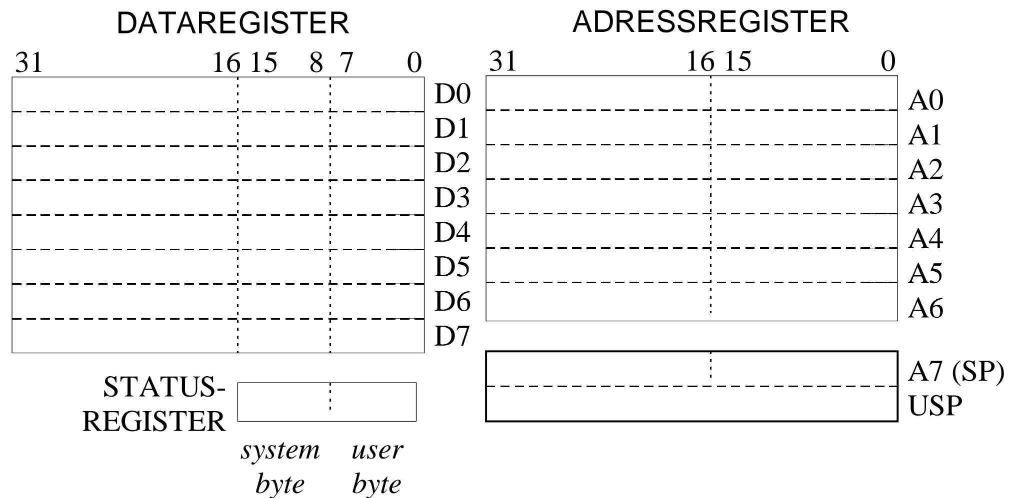
Figur 4.5 nedan visar programmerarens bild av MC68000. Observera att denna bild inte innehåller information om bussar, ALU och dylikt. Däremot så hittas dataregister, adressregister och statusregister. Vi skall nu beskriva dessa register.

Det finns åtta dataregister benämda **D0**..**D7** där alla är 32 bitar långa. Vi ser även att bitarna i registren är numrerade från [31..0]. Dessa register används för bearbetning av 32 bitars data. Vidare finns det ett streck mella bit 16 och 15 respektive 8 och 7. Detta innebär att processorn även kan bearbeta 8 eller 16 bitar av ett dataregisterinnehåll. Vi säger att processorn kan arbeta med *8 bitars ord*, *16 bitars ord* och *32 bitars ord*.

Använda termer

- byte *för 8 bitars ord*
- word *för 16 bitars ord*
- long *för 32 bitars ord*

Figur 4.5 visar också processorns sju adressregister **A0..A6**. Dessa används för adressbearbetning av både 32 och 16 bitars adresser. Vidare ser vi programräknaren (**PC**) som också är på 32 bitar. Detta register användes för att peka ut instruktionen som står i tur att exekveras.

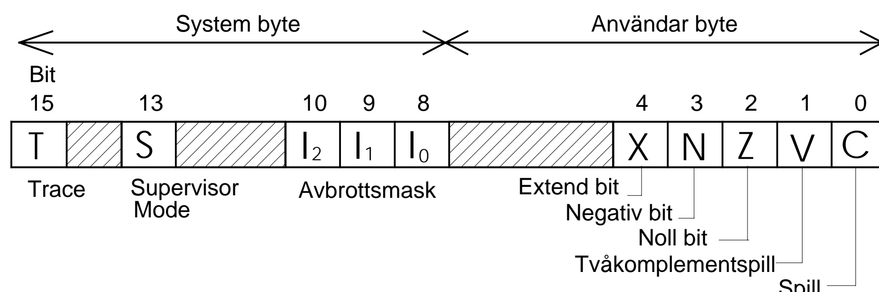


FIGUR 4.5 PROGRAMMERARENS BILD AV MC68000

Det finns även två 32 bitars register som betecknas **USP** och **SP**. Observera att dessa båda register *kan* ha beteckningen **A7**. Registren fungerar som stackpekare och används flitigt av en van assemblerprogrammerare. Registren används för att peka ut en i systemet nödvändig dataarea som kallas *STACK*. Vi återkommer till dessa begrepp ett antal gånger längre fram.

Slutligen finns processorns statusregister (**SR**) som visas i figur 4.6. Vi ser att det är 16 bitar långt och består av en *användar-byte* och en *system_byte*. En assemblerprogrammerare tolkar detta som att processorn kan vara i två moder där vissa operationer (instruktioner) *inte* är tillåtna i användarmod utan enbart i system mod (eller *Supervisor Mode*).

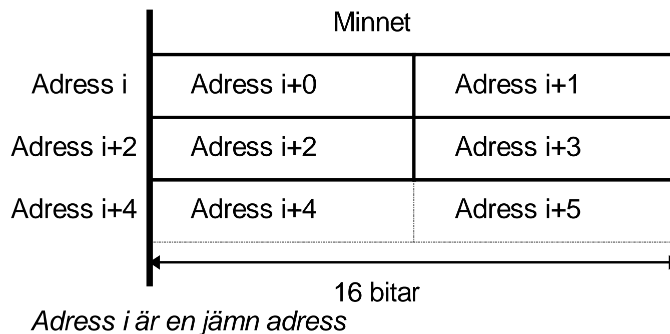
Figur 4.6, som programmeraren också måste bekanta sig med, visar statusregistrets innehåll. Bitarna N, Z, V och C längst till höger känner vi igen sedan tidigare. Dessa anger flaggbitarna från resultatet av en operation i datavägens ALU. De övriga bitarna utelämnas här men beskrivs detaljerat i kapitel 5.



FIGUR 4.6 MC68000 STATUSREGISTER.

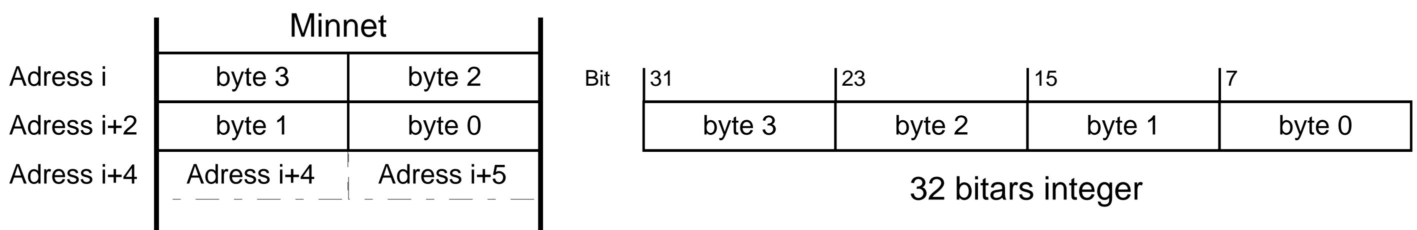
4.2.2 Datatyper

MC68000 använder sig av så kallad *byte*-orienterad addressering. Med detta menas att den kan adressera en enskild byte i minnet, trots att processorn har en 16-bitars databuss. Innan vi visar hur *bytes, word* och *long* lagras i minnet visar vi i figur 4.7 hur minnet adresseras med udda och jämna adresser.



FIGUR 4.7 MINNESADRESSER I ETT MC68000-SYSTEM

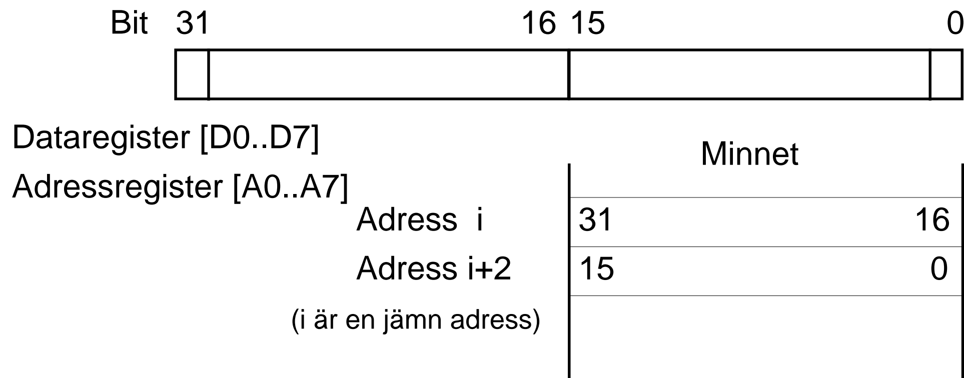
Studera även nästa figur (Figur 4.8.) som visar hur ett 32 bitars heltal (*integer*) lagras i minnet. Som vi sett tidigare placeras den mest signifikanta biten (**MSB**) till vänster och den minst signifikanta (**LSB**) till höger.



FIGUR 4.8 MINNESLAYOUT AV EN 32 BITARS INTEGER

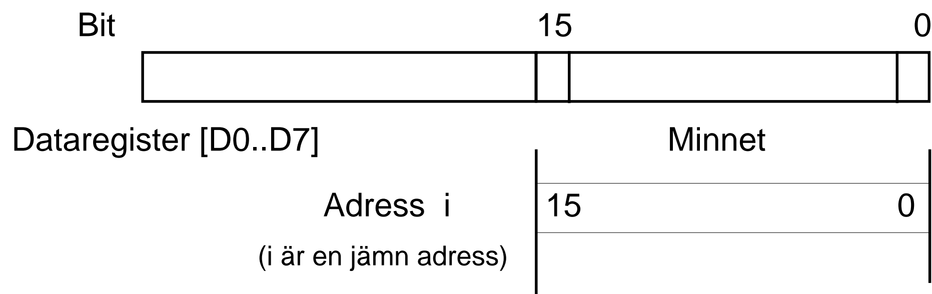
Processorn kan arbeta med *byte, word* och *long*, dessutom kan processorn bearbeta enstaka bitar och NBCD-tal i ett register, men detta påverkar inte var datatyperna lagras i minnet och i dataregistren när de överförs mellan processor och minne. Vi visar nu hur datatyperna *long, word* och *byte* placeras i register och minne.

Observera att datatyperna word eller long inte *kan adresseras med en udda adress. Processorn förutsätter att dessa alltid börjar på en jämn adress.*



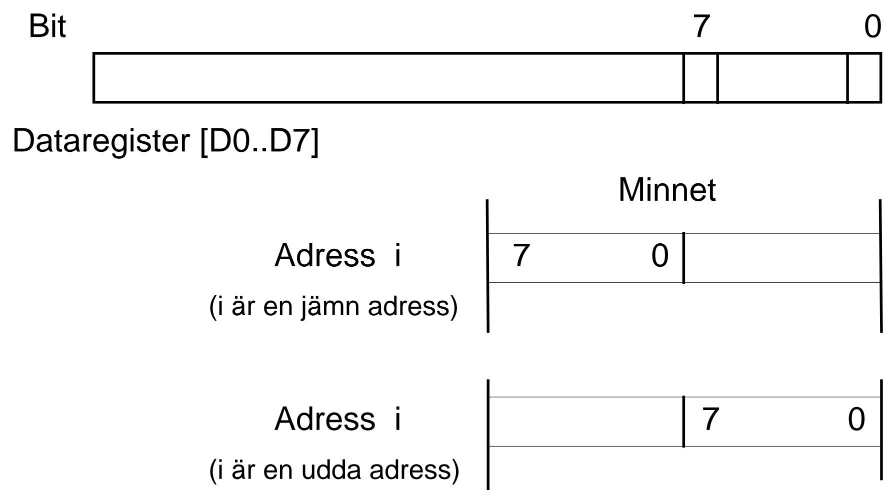
FIGUR 4.9 MINNESLAYOUT AV LONG

Figur 4.9 visar hur en *long* lagras i minnet. Både innehållet i adressregister och dataregister kan överföras mellan processor och minne. Observera att en *long* lagras med start på en jämn minnesadress



FIGUR 4.10 MINNESLAYOUT AV WORD

Figur 4.10 visar hur ett *word* lagras i minnet. Både innehållet i adressregister och dataregister kan överföras mellan processor och minne. Observera att registrets lägsta *word* överförs och att ett *word* lagras med start på en jämn minnesadress

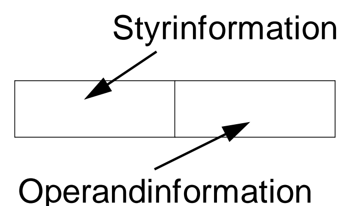


FIGUR 4.11 MINNESLAYOUT AV BYTE

Figur 4.11 visar hur en *byte* lagras i minnet. Endast innehållet i ett dataregister kan här överföras mellan processor och minne. Observera att registrets lägsta *byte* överförs och att en *byte* kan lagras med start på både en jämn och en udda minnesadress

4.2.3 MC68000, instruktionsformat

I kapitel 2 visade vi hur ett generellt instruktionsformat består av styrinformation och operandinformation, (se även figur 4.12). Styrinformationen indikerar exempelvis att en addition skall utföras av operanderna angivna av operandinformationen. Denna styrinformation kallas *operationskod* (*OP-kod*).



FIGUR 4.12 INSTRUKTIONSFORMAT.

Vissa instruktioner kan sakna operandinformation eftersom OP-koden innehåller all information som krävs för att utföra instruktionen. Ett exempel på detta är instruktionen `NOP` (No OPeration) som inte utför någonting och instruktionen `RTS` (Return from Subrutine) som läser en *long* till PC från en minnesadress utpekad av register A7. Alltså instruktioner som inte använder operander alls eller instruktioner som arbetar med operander som bestäms direkt av OP-koden.

OP-koden indikerar:

- vad som skall utföras
- antal *word's* instruktionen upptar
- om instruktionen använder ingen, en eller två operander
- var eventuella operander eller operandinformation finns

Exempel på instruktioner som använder en operand är CLR (CLeaR), där operanden kan vara ett av processorns register eller ett minnesinnehåll (minnesregister; vi lämnar nu termen minnesregister. I fortsättningen används minnesinnehåll eller bara minne.)

Instruktionsgruppen MOVE använder sig av två operander, en källoperand (*Source*) och en destinationsoperand (*Destination*). En källoperand kan vara ett registerinnehåll som skall till en destinationsadress i minnet. Destinationsadressen kan finnas i ett adressregister eller den kan ingå i instruktionen.

I processorns datablad hittar vi tabellen *OP CODE MAP* som visar vilka operationskoder som processorn accepterar. (Figur 4.13 visar en delmängd av tabellen.) Instruktionerna är indelade i 16 grupper och de fyra mest signifikanta bitarna i operationskoden (bit [15..12]) anger grupp. De övriga bitarna i OP-koden kan innehålla operandinformation och/eller ange vilken instruktion inom gruppen som menas.

Lägg märke till att MOVE instruktionerna har var sin grupp (0001, 0010 och 0011) för *byte, word* och *long*. Observera också att det finns en grupp med diverse instruktioner (0100). Den sista gruppen (1111) är reserverad för flyttalsinstruktioner som MC68000 inte kan utföra själv, men däremot kan den signalera till en så kallad flyttalsprocessor (*CO-processor*) som kan utföra dessa. Vidare finns en grupp (1010) för att emulera andra processorers instruktioner.

Bit 15..12	OPERATION
0000	Bit Manipulation / MOVEP / Immediate
0001	MOVE Byte
0010	MOVE Long
0011	MOVE Word
0100	Miscellaneous
0101	ADDQ / SUBQ / Scc / DBcc
0110	Bcc (Branch)
0111	MOVEQ
1000	OR / DIV / SBCD
1001	SUB / SUBX
1010	(Unassigned)
1011	CMP / EOR
1100	AND / MUL / ABCD / EXG
1101	ADD / ADDX
1110	Shift / Rotate
1111	(Unassigned)

Figur 4.13 MC68000:s Operationskoder

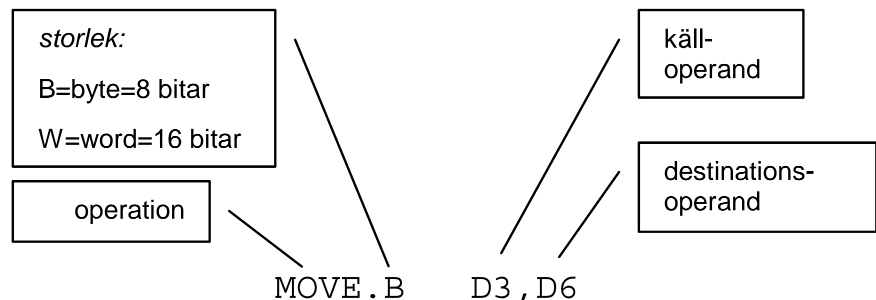
Innan vi i detalj studerar OP-kodens övriga bitar måste vi klargöra innebörden av *effektiva adressen* (*effective address, EA*). EA är "adressen" till data. Om önskad data-lagringsplats (skriv eller läs) är i minnet på adress \$12 3A BC, är EA = \$12 3A BC. Om däremot lagringsplatsen är i exempelvis dataregister 5 är EA = D5 osv.

Om vi önskar flytta A till B, är A *källan* (*Source*) eller *källoperanden* och B destinationen eller *destinationsoperanden*. A och B kan vara register och/eller minnesadress. Figur 4.14 och 4.15 illustrerar en assemblerinstruktion.

```
MOVE    A , B
```

INSTRUKTION KÄLLA , DESTINATION

FIGUR 4.14 FORMAT FÖR ASSEMBLERINSTRUKTION



FIGUR 4.15 EXEMPEL PÅ MC68000 ASSEMBLERINSTRUKTION

EXEMPEL 4.1

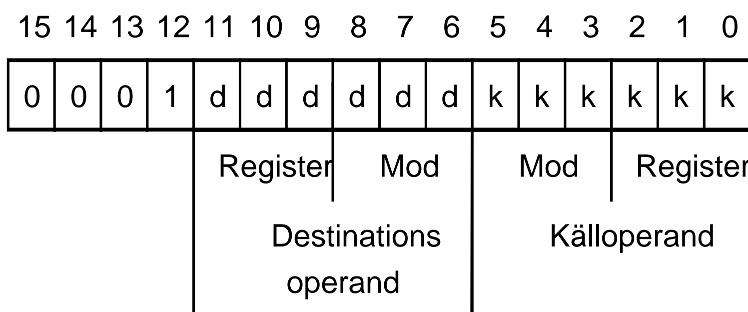
Några olika assemblerinstruktioner.

```
        MOVE.B      D3,D6
*       kopiera en byte (8 bitar) från D3 till D6

        MOVE.B      #$12,D5
*       placera en byte i register D5

        MOVE.W      #$1234,D5
*       placera ett word (16 bitar) i register D5

        MOVE.W      D5,($123456).L
*       kopiera 16 bitar från D5 till adress $123456

        MOVE.W      D2,(A5)
*       kopiera ett word från D2 till adressen som finns i A5
```

Vi skall nu studera några instruktioners OP-koder. Vi kan använda en assemblator som översätter assemblerprogrammet till maskinprogram och studera OP-koderna men här väljer vi att utföra assembleringen för hand. Vi utgår från föregående exempel så att vi känner igen instruktionerna och kan då bestämma OP-kodernas fyra första bitar enligt figur 4.13. Vi skall nu ange operandinformationen som bestämmer vart operanden/ operanderna finns.

Studera den första instruktionen `MOVE.B D3,D6` i exemplet. Instruktionen kopierar en byte från källan **D3** till destinationen **D6**. Studerar vi figur 4.13 hittar vi för instruktionsgruppen MOVE *byte* de första fyra bitarna (0001). För att ange var denna byte skall läsas och vart den skall skrivas så måste vi även ange att register D3 är källan och register D6 är destinationen. Vad vi önskar att ange är den *effektiva adressen*.

Studera figur 4.16 som visar ett typisk format på en OP-kod för en MOVE.B-instruktion. Sex bitar anger var källoperanden finns (**k**-bitarna bit 5..0), och sex bitar anger var destinations-operanden finns (**d**-bitarna bit 11..6).



FIGUR 4.16 INSTRUKTIONSFORMAT FÖR MOVE.B-INSTRUKTION.

Figuren visar att de fyra mest signifikanta bitarna är 0001 vilket indikerar att det är en MOVE.B-instruktion. Vidare ser vi att den resterande delen av OP-koden är indelad i två fält, destinationsoperand och källoperand. Det är i dessa fält vi skall ange var data skall läsas och var data skall skrivas. *Vi skall bilda EA*. För att klara av detta måste vi veta hur dessa fält kodas för att kunna ge **d**- och **k**-bitarna. Se figur 4.17 som visar EA-bildning, EA-bitfält och deras beteckning: *Adresseringsmoden*. Adresseringsmoden anger hur EA bildas och beskrivs utförligt i kapitel 5.

Adresseringsmod	EA-bildning	EA-bitfält Mod Register
Dataregister Direkt	EA = Dn	0 0 0 Register Nummer
Adressregister Direkt	EA = An	0 0 1 Register Nummer
Adressregister Indirekt	EA = (An)	0 1 0 Register Nummer
Adressregister Indirekt, postinkrement	EA = (An)+	0 1 1 Register Nummer
Adressregister Indirekt, predekrement	EA = -(An)	1 0 0 Register Nummer
Adressregister Indirekt med 16 bitars offset	EA = (An)+ d16	1 0 1 Register Nummer
Adressregister Indirekt med index och 8 bitars offset	EA = (An)+(Xn)+d8	1 1 0 Register Nummer
Absolut kort	EA = nästa 16-bitars ord	1 1 1 0 0 0
Absolut lång	EA = två nästföljande 16 bitars ord	1 1 1 0 0 1
Programräknarrelativ med 16 bitars offset	EA = (PC)+d16	1 1 1 0 1 0
Programräknarrelativ med index och 8 bitars offset	EA = (PC)+(Xn)+d8	1 1 1 0 1 1
Omedelbar	EA = (PC)+2	1 1 1 1 0 0

FIGUR 4.17 ADRESSERINGSMODER OCH EA-BILDNING

Till vänster i figur 4.17 ser vi vad adresseringsmoderna heter och i mitten hur EA (Effektiva adressen) bildas. **EA-bitfältet** består av sex bitar som anger adresseringsmod och register. Observera att mod 111 är en gemensam beteckning för *flera* adresseringsmoder och att Register Nummer-bitarna då bestämmer *vilken* adresseringsmod som menas.

I vårt exempel (`MOVE.B D3,D6`) är både källoperanden och destinationsoperanden ett dataregister. Det innebär att vi skall välja **EA=Dn** både för källoperanden och destinationsoperanden i kolumnen EA-bildning. I tillhörande EA-bitfält läser vi att:

Mod = 000 Register Nummer.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	1
				Register			Mod			Mod			Register		
				Destinations-operand						Källoperand					

Instruktion i minnet: $1C03

FIGUR 4.18 INSTRUKTIONEN `MOVE.B D3,D6`

Detta innebär att för källoperanden blir bitarna 000 för Mod och 011 för dataregister 3. För destinationoperanden blir också Mod = 000 och Register = 110 eftersom dataregister 6 används. Figur 4.18 visar den fullständiga operationskoden.

Figuren visar att bit [5..0] anger var källoperanden är lagrad och bit [11..6] anger var destinationen är. Sättet att hitta eller beräkna EA delas in i så kallade *adresseringsmoder* (*Addressing Mode*). Fallet med vårt exempel kallas *Dataregister Direkt* (*Data Register Direct*) då data finns i registeren. Därmed har vi assemblerat vår första assemblerinstruktion (till maskininstruktion).

Studera på nytt figur 4.17 som beskriver hur delar av OP-koden ser ut och observera kolumnen som anger EA-bildning. Här anges också hur många word instruktionen upptar i minnet (hur lång instruktionen är). Orden (word:en) utöver OP-koden kallas **Extensionsord** (*Extension Words*). För exempelvis adresseringsmoden *Absolut Lång* står det "EA = två nästföljande 16-bitars ord". Med detta menas att instruktionen har två *extensionsord* vilka är lagrade efter OP-koden i minnet.

Maximalt kan en instruktion bestå av fem word's i minnet:

- *Operationskod, 1 word*
- *Extensionord för källa: 2 word's*
- *Extensionord för destination: 2 word's*

Studera övre deln av figur 4.19 som visar den generella utformingen av en instruktion med extensionsord. Slutligen visas nederst i figuren extensionsordet som används vid Adressregister Indirekt (och Programräknarrelativ) med index och 8 bitars offset. (Detta extensionsord kan tolkas som en *extra* operationskod som ingår i instruktionen.)

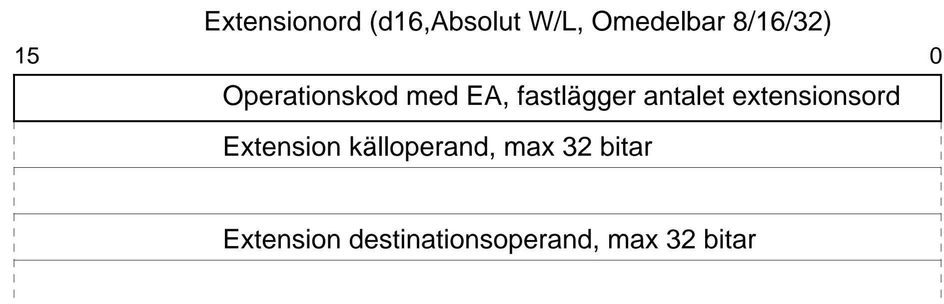
Nästa instruktion i exemplel 4.1 använder ett extensionsord. Instruktionen `MOVE.B #$12,D5` överför källan (#$12) till destinationen (D5). Enda skillnaden mot det förra exemplet är att källoperanden nu finns tillsammans med instruktionen som ett extensionord. EA är nu nästa word i programminnet relativt OP-koden. Denna adresseringsmoden kallas för Omedelbar (*Immediate*). Studera figur 4.17 som visar att omedelbar adresseringsmod har

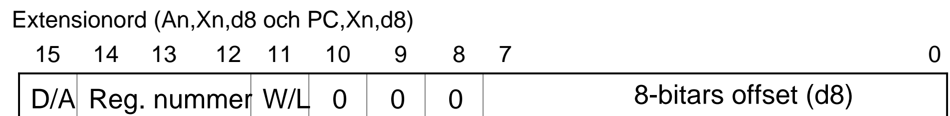
EA-bitfältet **111 100**

Källoperandfältet blir således 111 100 och destinations-operanden som i det förra exemplet bortsett från att nu används D5, bitfältet blir: 000 101, där 000 anger Dataregister Direkt och 101 anger D5 (se figur 4.20). Observera att extensionsordet i minnet upptar ett *word* trots att data är endast en *byte* långt, se även figur 4.19.).

Nästa instruktion `MOVE.W #$1234,D5` är som exemplet innan fast nu överförs ett word till D5 i stället för *byte*. Källoperand och destinationsoperand blir därför som förut. Det enda som skiljer är att instruktionen tillhör en annan instruktionsgrupp. Bit [15..12] blir nu 0011. Instruktionen visas i figur 4.21. Här används (i motsättning till

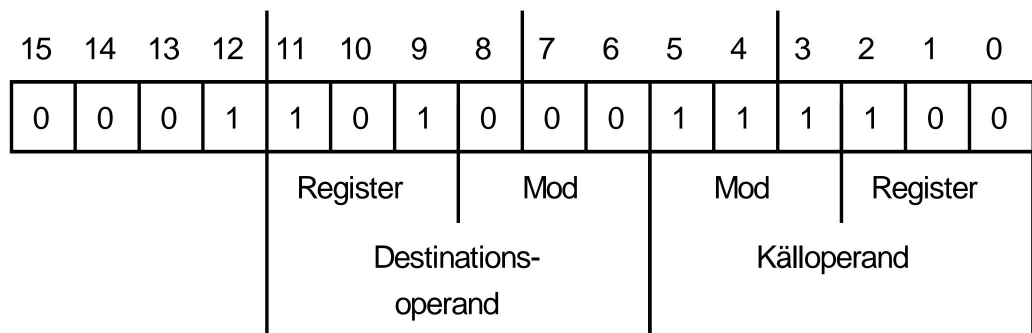
förra exemplet) ett helt 16-bitars extensionsord ty instruktionen tillhör gruppen move word.



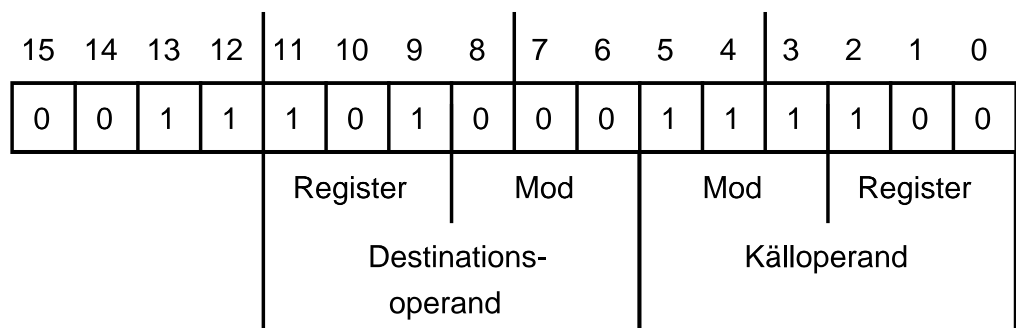


D/A: Indexregister Xn, 0 om dataregister, 1 om adressregister
W/L: Indexformat, 0 om "word", 1 om "long"

FIGUR 4.19 BITFÄLT FÖR EXTENSIONSORD

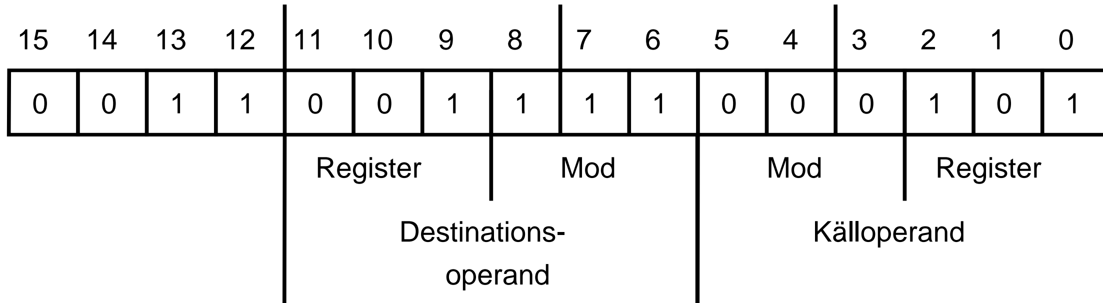


FIGUR 4.20 INSTRUKTIONEN `MOVE.B #$12,D5`



FIGUR 4.21 INSTRUKTIONEN `MOVE.W #$1234,D5`

Nästa instruktion kopierar ett word från källan D5 till minnesadress $123456 (destination) `MOVE.W D5,($123456).L`. Källoperandens EA är Dataregister Direkt vilket ger mod 000 och registernummer blir 101. Destinationsoperanden finns i minnet och adresseringsmoden för denna instruktion kallas *Absolut lång*. Studera figur 4.17, som ger mod 111, och Register 001. Observera att mod 111 innehåller flera adresseringsmoder och att registerindikationen då väljer vilken adresseringsmod. Se figur 4.22 som visar instruktionen. Observera också att instruktionen består av en OP-kod och ett 32-bitars extensionsord (en long adressangivelse).

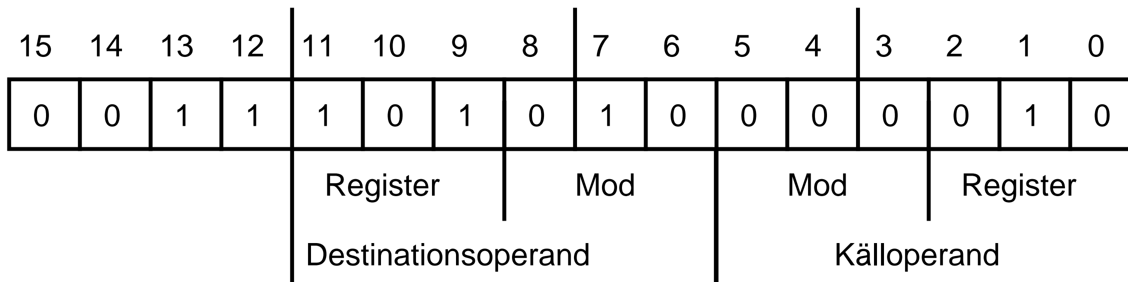


Instruktionen i minnet: $33C5 0012 3456

FIGUR 4.22 INSTRUKTIONEN `MOVE.W D5,($123456).L`

Instruktionen `MOVE.W D2,(A5)` kopierar ett word från källan D2 till destinationen som utpekas av A5. Ett adressregister inom parantes tolkas som att registrets innehåll skall adressera minnet. Instruktionsgruppen blir 3 (0011) i vanlig ordning för en MOVE.W-instruktion.

Källoperanden är D2 vilket innebär mod 000 med register 010 Vidare har vi en ny adresseringsmod för destinationen, studera på nytt figur 4.17 Vi hittar EA-bildningen EA = (An) som ger moden 010 Adressregistret som används är A5 vilket ger registerbitarna 101. Observera att all operandinformationen ryms i OP-koden. Se figur 4.23.



Instruktion i minnet: $3A82

Figur 4.23 Instruktionen `MOVE.W D2,(A5)`

Studera instruktionen `MOVE.W D7,($34,A3,D2.L)` sist i exemplet. Detta är ett exempel på adresseringsmoden *Adressregister*

Indirekt med index och 8-bitars offset. Instruktionen kopierar källan D7 till destinationsadressen EA som fås genom additionen:

EA = (An) + (Xn) +d8

I vårt fall är adressregistret An = A3, Indexregistret Xn=D2 och slutligen offset = $34. EA blir således

EA= (A3) + (D2) + $34

Studera Figur 4.24 nedan som visar operationskod med extensionsord. Källoperand blir mod 000 med register D7. Sedan anger adresseringsmoden för destinationen mod 110 och register A3 ger registerbitarna 011. Slutligen anger extensionsordet vilket *Indexregister* som skall användas.

OP-kod

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	1
				Register			Mod			Mod			Register		
				Destinationsoperand						Källoperand					

Extensionord

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	0	0	0	0	1	1	0	1	0	0
D/A	Register			W/L				8-bitars offset							

Instruktion i minnet $3787 2834

FIGUR 4.24 INSTRUKTIONEN `MOVE.W D7,($34,A3,D2.L)`

Indexregistret är ett dataregister vilket medför att bit 15 (D/A) är 0. Då vi har valt D2 blir bit [14..12] 010. Vidare har vi angett *long* adressangivelse för D2 vilet ger biten W/L=1. Slutligen har offset-fältet värdet $34.

Slutligen skall vi visa exempel på *programräknarrelativt adressering* eller kortare, ***PC-relativ*** adressering. PC-relativ adressering används vid villkorliga hopp, mestadels för att kunna utföra villkorliga programkonstruktioner. I en sådan konstruktion finns det olika vägar programmet kan fortsätta efter.

EXEMPEL

Användning av villkorliga hopp

```
if   (P   !=   Q){
        Satser_A;
}else{
        Satser_B;
}
```

Om P är skilt från Q
utför Satser A
annars
utför Satser B
slut

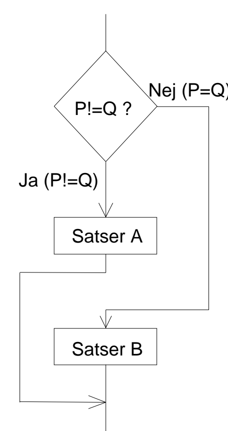
4.2

CMP (Compare): Jämför

branch = gren

Villkorliga hopp, eller *brancher*, placeras direkt efter att jämförelsen (P != Q) är utförd för att antingen fortsätta med *Satser_A* eller "hoppa" till *Satser_B*. När vi använder branch-instruktioner i ett assemblerprogram kan vi ange läget dit vi vill hoppa, exempelvis till "else" eller "end".

Själva jämförelsen utförs med en så kallad *compare*-instruktion. Det enda instruktionen utför är en subtraktion som påverkar flaggbitarna i statusregistret (P-Q). Själva hoppet och testen (om P är skilt från Q) utförs med branch-instruktionen som beroende av flaggbitarnas värde utför ett hopp eller fortsätter med nästa instruktion.



FIGUR 4.25 FLÖDESDIAGRAM

EXEMPEL

Vi fortsätter med föregående exempel. En sekvens assemblerinstruktioner som utför C-satserna visas här.

```
        MOVE.L       (P).L,D1
        CMP.L        (Q).L,D1
        BEQ          else          hoppa till "else" om lika
        ...
        utför Satser_A
        ...
        BRA          end_if        hoppa till end_if

else    ...
        utför Satser_B
        ...

end_if---------    fortsätt med annat arbete
        ...
```

4.3

BEQ: Branch EQual*:*

”Hoppa” om lika

BRA: BRanch Always:

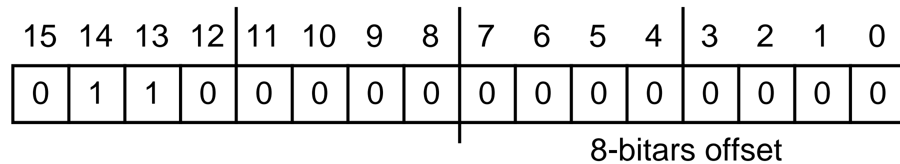
”Hoppa” alltid

Det finns flera olika branch-instruktioner som används för att utföra programhopp för olika villkor, exempelvis:

=, <, >, osv.

Beroende på vilka testvillkor vi har så väljer vi olika branchinstruktioner. I vårt exempel önskar vi utföra "Satser A" när P!=Q och Satser B när P=Q. Vi väljer därför ett villkorligt hopp som utförs när jämförelsen ger resultatet "likhet". Denna branchinstruktion heter *Branch EQual*, BEQ `else` (Hoppa om lika). När alla "Satser_A" är utförda skall ett hopp till "end_if" utföras. Instruktionen `BRA end` utför detta (*BRanch Always*). Se flödesplanen i figur 4.25. Programkonstruktioner och val av branch:er behandlas grundligt i kapitel 5. Vad vi vill visa på här är en relativ adresseringsmod och exemplifierar här med branch.

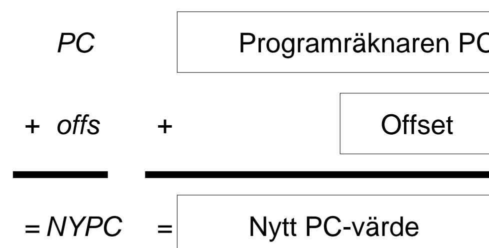
Vi försöker nu att hitta bra-instruktionens OP-kod. I figur 4.13 ser vi att de första fyra bitarna är 0110 då dessa bestämmer instruktionsgruppen *brancher*. Studerar du nu databladen hittar du en figur liknande 4.26. Observera att instruktionen har ett extensionord för offset (märkt yyyy i figuren). Låt oss nu se hur denna bestäms.



Instruktion i minnet: $6000 yyyy

FIGUR 4.26 INSTRUKTIONEN `BRA <EA>`

Vid relativ adresseringsmod bildas EA efter någon form av addition av **PC** och någon **offset**. Vid brancher där man önskar ett programhopp kan detta uttryckas: PC = PC + offset. Offset är ett *avstånd* räknat i *bytes* från där PC pekar när instruktionens OP-kod är läst. Studerar vi vårt exempel för beq-instruktionen skulle detta innebära att offset bör vara lika antal bytes som instruktionerna CMP.W och MOVE.W upptar i minnet för att bilda ett PC-värde som motsvarar början på else-delen av programmet. När processorn utför en branch-instruktion adderar den offset till sitt PC enligt Figur 4.27.



FIGUR 4.27 BERÄKNING AV EA

Vi exemplifierar här instruktionen `BRA end_if.` För att hitta motsvarande maskininstruktion måste vi känna till "hur långt" det är mellan instruktionens OP-kod och end_if. För att ta reda på detta studeras programmets övriga maskininstruktioner. Förutsätt att

programmet är lagrad på adress $4000 i minnet. Se figur 4.28. Förutsätt vidare att variabeln P är lagrad på adress $00 12 34 56 och att variabeln Q är lagrad på adress $00 34 56 78. Observera att instruktionerna BRA och BEQ båda består av OP-kod och ett extensionsord (offset).

Adr	Minne	Kommentar	Instruktion
$4000	$3239	OP-kod	`MOVE.W (P).L,D1`
$4002	$0012	Operand P	
$4004	$3456		
$4006	$B279	OP-kod	`CMP.W (Q).L,D1`
$4008	$0034	Operand Q	
$400A	$5678		
$400C	$6700	OP-kod	`BEQ else`
$400E	$xxxx	Offset för BEQ	
$4010	$????	Första OP-koden i Satser A	
$4120	$????	Sista ord i Satser A	
$4122	$6000	OP-kod	`BRA end_if`
$4124	$yyyy	Offset för BRA (***PC***)	
$4126	$????	Första OP-koden i Satser B	`else`
$4128	$????		
$4404	$????	Sista ord i Satser B	
$4406	$????	Första Op-koden for efter-följande program (***NY PC***)	`end_if`

FIGUR 4.28. VILLKORSSATSER SOM MASKINPROGRAM

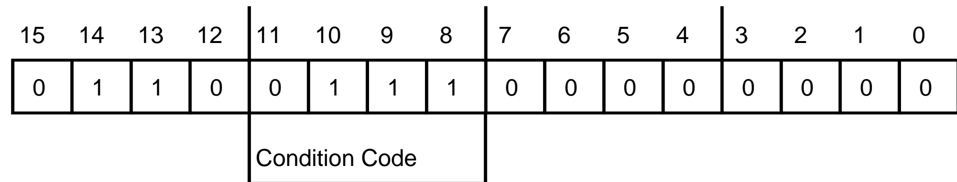
Vi skall nu bestämma extensionsord (offset) till branch-instruktionen `BRA end_if` som hittas på adress $4122 och $4124 i minnet. När OP-koden är inläst pekar PC på extensionsordet på adress $4124 (*PC* i figur programmet ovan). När instruktionen är klar skall PC ha värdet $4406 (*NYPC*). Vi kan enkelt utföra subtraktionen $4406 - $4124 och få resultatet $2E2, se marginalen. Branchinstruktionen får alltså extensionsordet $02E2 som är visat som $yyyy i figur 4.28. Den kompletta instruktionen blir således $6000 02E2

NYPC:	*$4406*
- PC:	*$4124*
= Offset:	*$02E2*

Studerar vi BEQ-instruktionen finner vi att offset blir $118, vilket ger extensionordet $0118, Detta visas som $xxxx i figur 4.28, se även marginalen.

NYPC:	*$4126*
- PC:	*$400E*
= Offset:	*$0118*

För att avgöra instruktionens OP-kod måste vi studera Bcc-instruktionsgruppens *Condition Code*. Det är dessa bitar som bestämmer vilken typ av branch som väljs (hoppa om lika, hoppa om större, hoppa om negativt, mm) Se figur 4.29. BEQ-instruktionen har bitarna [11..8] satt till 0111 vilket innebär “hoppa om lika.”

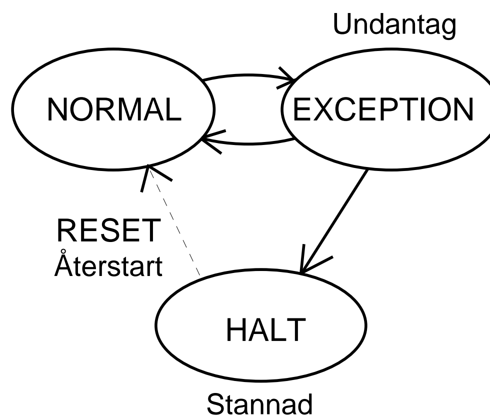


Instruktion i minnet: $6700 0118

FIGUR 4.29 INSTRUKTIONEN `BEQ <EA>`

4.3 Processorns arbetssätt

Vi skall nu beskriva hur processorn arbetar. Vi kommer att se hur handskakningen mellan processor och minne till under en läscykel går till. Vi ger också exempel på hur processorn utnyttjar stacken vid subrutinanrop. Slutligen beskrivs kort hur vi externt kan styra processorns arbete. Först beskrivs processorns olika tillstånd och hur den startar.



FIGUR 4.30 MC68000:S OLIKA TILLSTÅND

Som vi tidigare påpekat är processorn sysselsatt med att hämta och utföra instruktioner. (*FETCH, EXECUTE)*. Vi säger då att processorn är i tillståndet *Normal*. Se figur 4.30. Övriga tillstånd som processorn kan vara i är *Exception* (undantag) och *Halt* (avstannad).

Exeption: Undantag

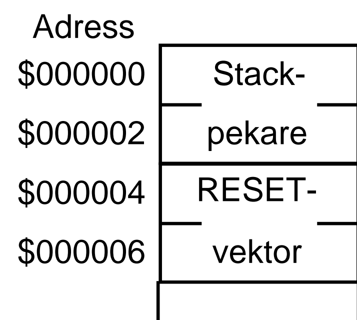
Halted: Avstannad

Processorns normala tillstånd är alltså att hämta instruktioner från minnet och utföra dessa. Det är detta tillstånd vi skall koncentrera oss vid i detta kapitel. Processorn byter tillstånd till “Exception” när

exempelvis en kringkrets aktiverar en av processorns avbrottsingångar (IPL-ingångarna) för att på detta sätt snabbt starta upp speciella avbrottsprogram. Processorn byter även tillstånd till “Exception” när fel uppträder i systemet. Detta kan exempelvis vara adressering av minnet med en felaktig adress, alltså, du har skrivit ett felaktigt program. Speciella program startas även här.

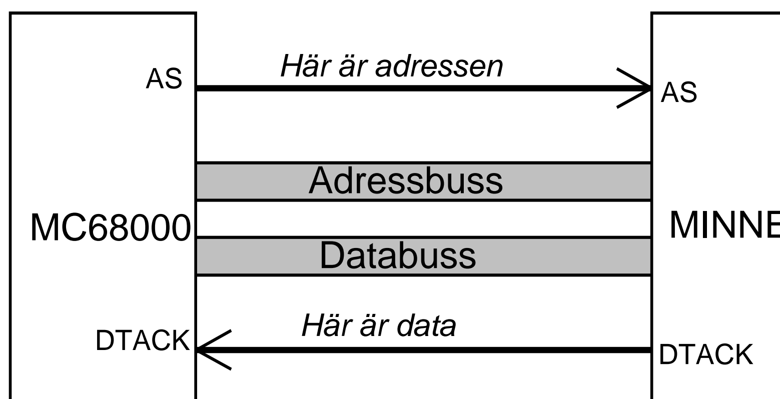
Slutligen finns Halt-tillståndet som inträder när allvarliga fel uppstår. Processorn förutsätter då att den inte kan kommunicera med sitt minne och sina kringkretsar och *stannar*. Har processorn kommit till Halt-tillståndet kan den endast återstartas med *RESET* vilket startar om mikrodatorsystemet från början.

Vi har ännu inte föklarat *hur* en dator startar. Du har säkert tryckt på en *Reset*-knapp på en persondator eller liknande när "det har hängt sig" för att sedan starta dina program på nytt. Vad händer? Jo, knappen är ansluten till en av processorns pinnar, *RESET*. Aktiveras denna signal så läser processorn en *long* i minnet från adress 0 och placerar denna i stackpekar-register A7 (SP). (Vi återkommer till stackpekar-registret.) Efter detta läses en *long* från adress 4 och placeras i PC. Se figur 4.31. Denna inlästa *long* kallas RESET-vektor och tolkas alltså som systemets startadress. Efter detta övergår processorn till sina vanliga FETCH EXECUTE tillstånd. Den som programmerat systemet måste därför se till att placera startadressen för systemets program på adress 4. Observera att denna minnesarea därför måste bestyckas med PROM.



FIGUR 4.31 SYSTEMETS STACKPEKARE OCH RESET-VEKTOR

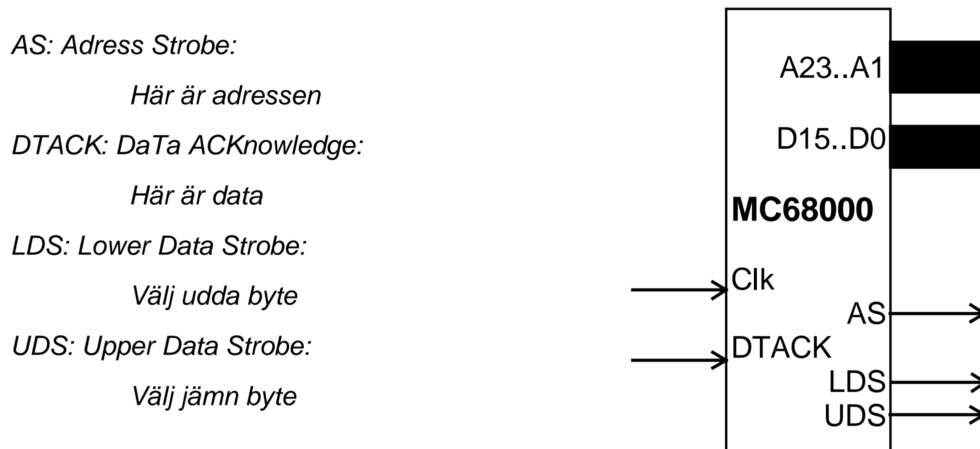
4.3.1 En busscykel - läscykel.



FIGUR 4.32 HANDSKAKNING UNDER EN LÄSCYKEL

I kapitel 2 beskrevs olika busssystem och hur en generell processor använder bussarna i ett mikrodatorsystem. Se figur 4.32. Vi skall nu studera en busscykel i detalj och beskriva hur processorn handskakningssignaler används.

MC68000 utnytjar ett asynkront bussprotokoll men har även stöd för synkront. Vi beskriver här det asynkrona protokollet. Figur 4.33 visar processorns handskakningssignaler som utnyttjas. Vidare visas den 16-bitars databussen och den 23-bitars adressbussen vilka inte är multiplexade.



FIGUR 4.33 HANDSKAKNINGS-SIGNALER FÖR DET ASYNKRONA BUSSPROTOKOLLET.

För att beskriva hur en busscykel fungerar visar vi ett exempel där processorn utför en läsning i minnet. Processorn läser kanske en OP-kod, en del av en instruktion eller data från primärminnet.

a) En busscykel startas med att processorn placerar den önskade adressen på adressbussen och att sätta **R/W**-signalen till läs (*read*). Om processorn önskar läsa ett *word* är både **LDS** och **UDS** aktiverade, om en *byte* skall läsas aktiveras endast en av signalerna beroende på om en jämn eller udda adress skall läsas.

b) Därefter indikerar processorn att adressbussen har ett giltigt värde och att styrsignaler är korrekta genom att aktivera utsignalen **AS**. Signalen **AS** kan populärt översättas till: *Här är adressen!* Processorn övergår nu i ett väntetillstånd där den oavbrutet undersöker insignalen **DTACK** från minnet.

c) Minnet som hela tiden undersöker AS ser nu att denna är aktiv,adresserar nu korrekt minnesregister med adressbussens värde. Då R/W-signalen indikerar en läsning kommer innehållet i

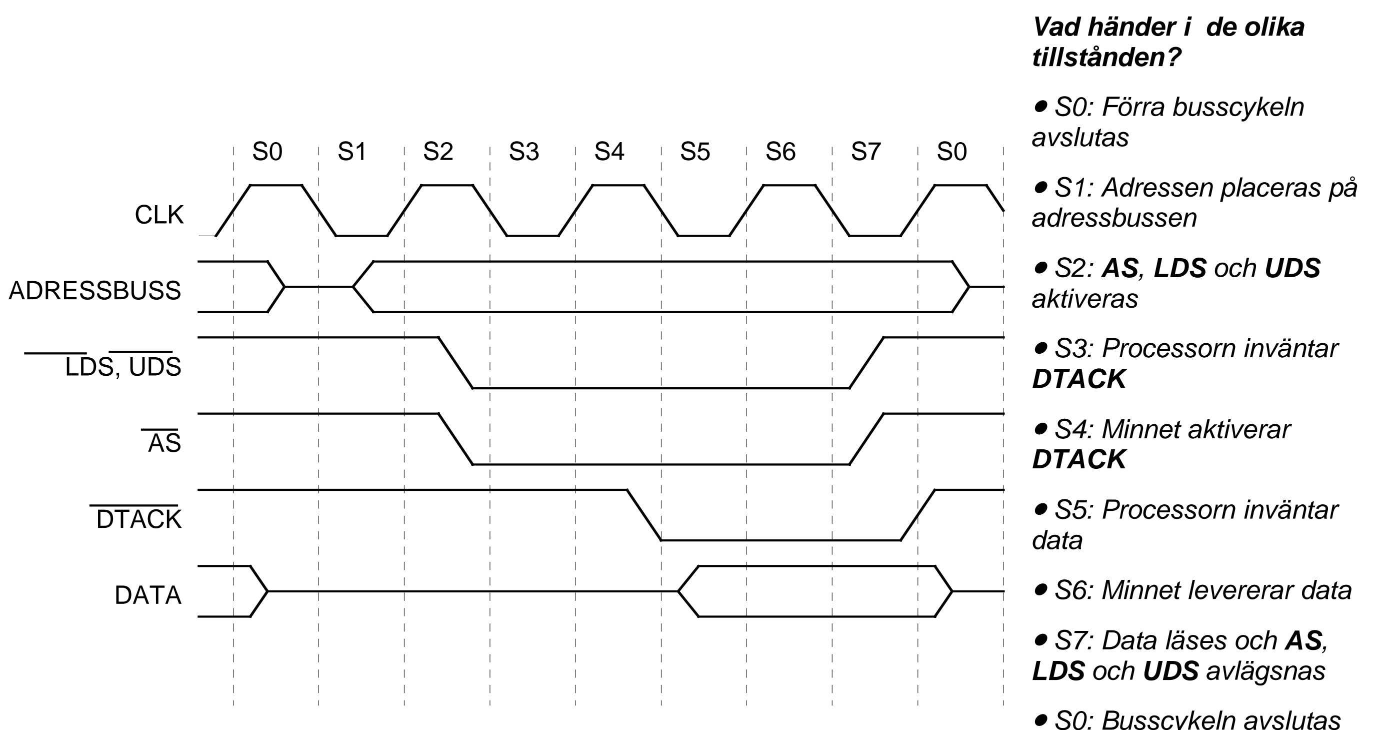
adresserat minnesregiset att överförs till databussen. När detta är klart signalerar minnet till processorn genom att aktivera signalen DTACK. Signalen DTACK kan populärt översättas till: *Här är data!*

d) Processorn lämnar sitt väntetillstånd när DTACK aktiveras. Den läser databussens värde , avlägsnar AS och R/W signalerna och slutligen avlägsnas den utlagda adressen på adressbussen.

e) Minnet upptäcker nu att AS inte är aktiv och avlägsnar därför DTACK och det utlagd data på databussen. Med detta är busscykeln slut och en ny kan påbörjas.

Vi har nu visat på hur en läsning (en läscykel) utförs med processorns asynkrona bussar. En skrivning utförs på liknande sätt.

När de olika signalerna och bussarna är aktiva och stabila visas enklast med ett tidsdiagram, se figur 4.34. Observera systemklockan överst i figuren. En busscykel indelas i åtta tillstånd [S0..S7] som alla är en halv klockcykel.



FIGUR 4.34 TIDSDIAGRAM ÖVER EN MC68000 LÄSCYKEL

Vi sa tidigare att signalen DTACK kunde tolkas som "här är data". Efter att ha studerat tidsdiagrammet i figur 4.34 borde en korrekt tolkning vara "data kommer i tid".

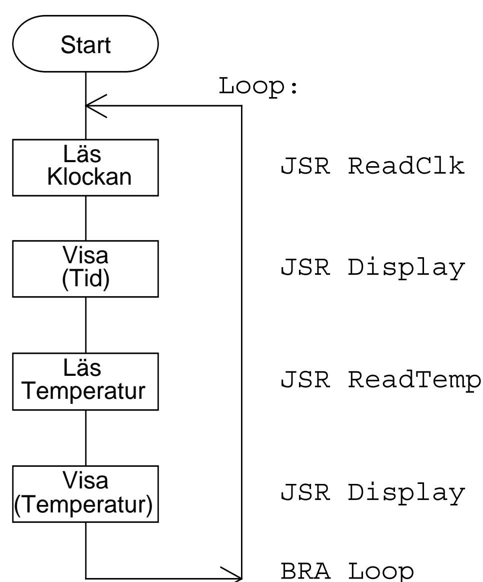
Om DTACK uteblir i tillstånd S4, vilket skulle innebära att data *inte* kommer i tid, (detta kan vara orsakat av fel adress eller hårdvarufel) så kommer processorn att förbli i sittväntetillstånd S4. Processorn sätter in ett antal så kallade Wait States, väntetillstånd för att invänta DTACK från ett *långsamt* minne. Processorn räknar exempelvis upp enligt S0, S1, S2, S3, S4, W, W, W, W, S5, S6, S7 om DTACK uteblir en viss tid.

Om DTACK uteblir helt och hållet, kommer systemet att hänga sig. För att komma ur detta tillstånd kan man alltid aktivera RESET men då processorn är utrustad med insignalen BERR (Bus ERRor) används denna tillsammans med en liten nedräknere. Räknaren har en utsignal som aktiveras när den räknat ner till noll. Denna utsignal kopplas till BERR på processorn. Räknare omstartas varje busscykel av AS signalen (som indikerar en ny busscykel).

Om räknaren räknar ned till noll indikerar detta att den inte är omstartad av ännu en AS signal (en ny busscykel) och skall då aktivera BERR på processorn. Buss fel (Bus Error) ingår i processorn Exception-tillstånd. Detta beskrivs ingående i ett senare kapitel.

4.3.2 Subrutiner och Stack.

En *stack* är ett *minnesutrymme* som kan användas för att överföra parametrar (variabler) till och från programrutiner, för att spara återhoppsadresser och för att lagra temporära variabler. (Dessa begrepp beskrivs i kapitel 5). Vi skall här introducera begreppet stack och studera hur återhoppsadresser lagras på stacken. Vid ett *subrutin anrop* sparas en återhoppsadress. Vi beskriver därför stacken enklast med ett exempel där vi studerar ett subrutin anrop (ett hopp till- och återhopp från en subrutin).



FIGUR 4.35 EXEMPEL PÅ HUVUDPROGRAM OCH SUBRUTINANROP.

En *subrutin* är ett avgränsat programavsnitt (*kod*) som utför en speciell uppgift. Ett program delas ofta upp i subrutiner och en stor fördel är att programmet då blir välstrukturerat. Studera figur 4.35 som visar ett programflöde över ett program som visar tid och temperatur på en "display" liknande det du kan se på husfasader. Varje rektangel i figur 4.35 kan tänkas motsvara ett avgränsat programavsnitt som placerats i en subrutin. Själva huvudprogrammet blir då ett fåtal rader som är enkelt att överskåda.

Vi skall nu studera hur ett subrutinanrop går till och hur återhoppsadressen lagras på-, och hämtas från stacken. För att visa detta måste vi känna till maskinprogrammet och var det är placerad i minnet. Förutsätt att programmet är placerad på adress $0020 4000, att subrutin ReadClk finns på adress $0034 5678, att *Display* är på adress $0066 2200 och slutligen *ReadTemp* finns på adress $0085 7310. Studera följande exempel.

Maskinprogram för huvudprogram och subrutiner

Exempel *Huvudprogram med subrutinanrop*

```
Loop
        JSR   ReadClk
        JSR   Display
        JSR   ReadTemp
        JSR   Display
        BRA   Loop
```

Adress   Minne   Instruktioner

Huvudprogram

```
$0020 4000  $4EB9   JSR   ReadClk
$0020 4002  $0034
$0020 4004  $5678
$0020 4006  $4EB9   JSR   Display
$0020 4008  $0066
$0020 400A  $2200
$0020 400C  $4EB9   JSR   ReadTemp
$0020 400E  $0085
$0020 4010  $7310
$0020 4012  $4EB9   JSR   Display
$0020 4014  $0066
$0020 4016  $2200
$0020 4018  $6000   BRA   Loop
```

Subrutin ReadClk

```
$0034 5678  $4E71   NOP
$0034 567A  $4E71   NOP
$0034 567C  $4E75   RTS
```

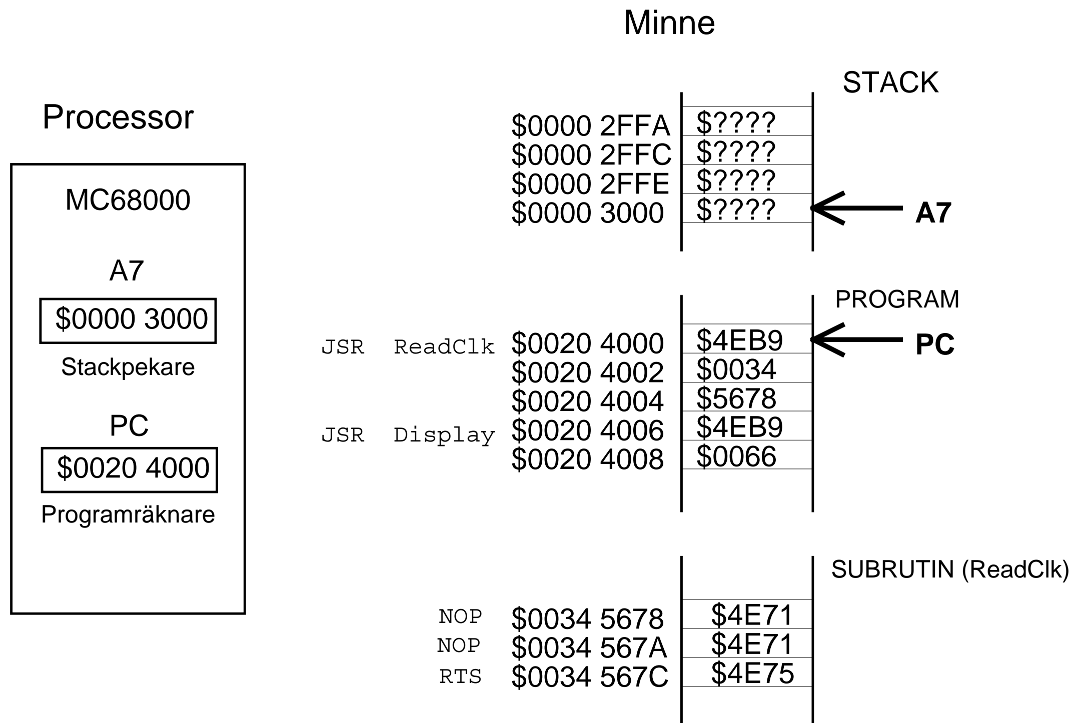
Subrutin Display

```
$0066 2200  $4E71   NOP
$0066 2202  $4E71   NOP
$0066 2204  $4E75   RTS
```

Subrutin ReadTemp

```
$0085 7310  $4E71   NOP
$0085 7312  $4E71   NOP
$0085 7314  $4E75   RTS
```

När den första instruktionen i huvudprogrammet (`JSR ReadClk`) står på tur att utföras ser processorns register ut som figur 4.36 visar. Till höger i figuren visas brottstycken av minnet och dess innehåll. Överst hittas stacken med oinitierat minnesinnehåll, i mitten hittas huvudprogrammet och nederst en subrutin.

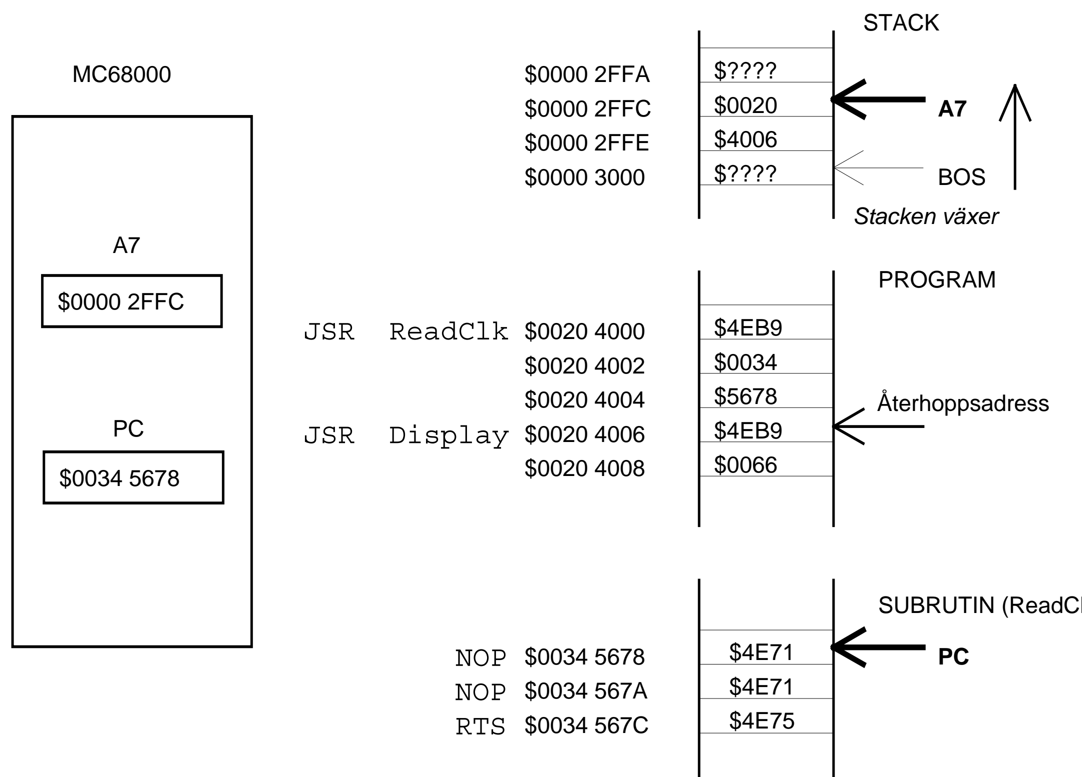


FIGUR 4.36 PROCESSORNS REGISTER FÖRE SUBRUTINANROPET.

Observera att stacken "växer" mot adress noll

Processorns register A7 används som stackpekare och då registret nu innehåller $0000 3000 förutsätter vi att stacken har denna startadress. *Observera att stacken växer mot adress noll.* Stackens maximala adress ($0000 3000) kallas Bottom Of Stack (BOS). Det översta på stacken kallas Top Of Stack (TOS). Nu är TOS = BOS. Processorn har alltså här sin *stack* från adress $0000 3000 och mot adress noll i minnet.

När den första instruktionen `JSR ReadClk` utförs fås ett hopp till subrutinen ReadClk. Studera stack och registerinnehåll i figur 4.37. Observera att TOS nu är $0000 2FFC

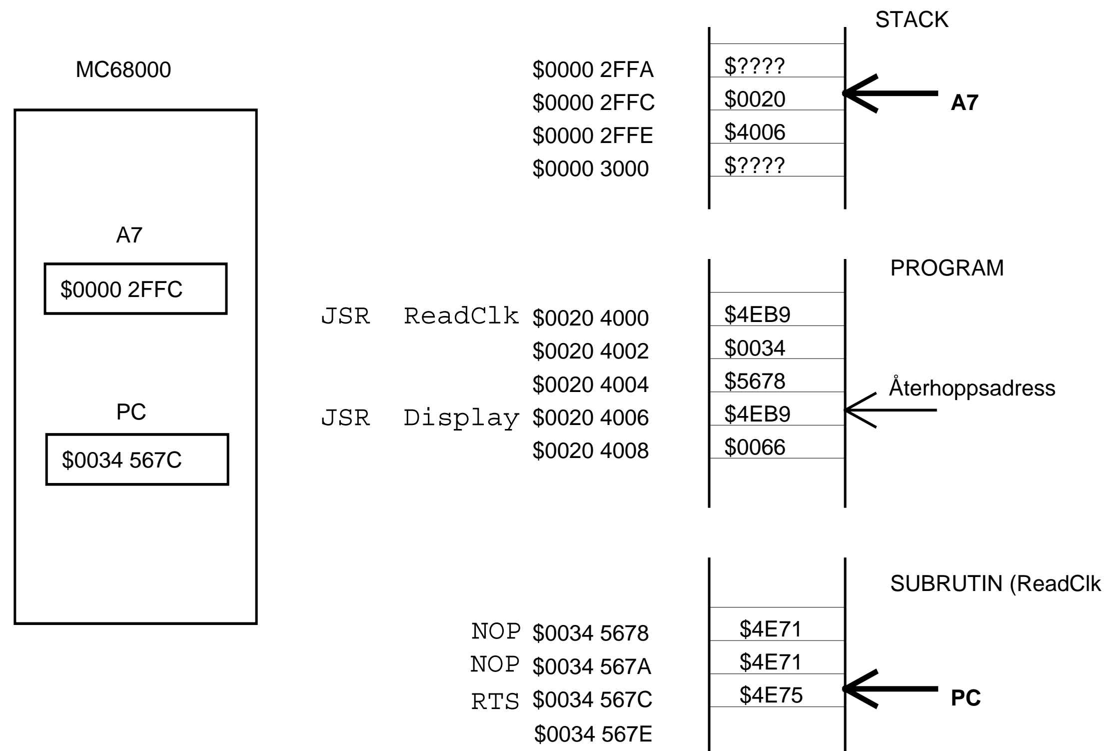


FIGUR 4.37 PROCESORNS REGISTER EFTER SUBRUTINANROPET.

Vid JSR utförs följande:

1. *hela instruktionen läses och PC uppdateras så att den pekar på nästa instruktion* `JSR Display` *(adress $0020 4006)*
2. *register A7 (*stackpekaren*) minskas med 2.*
3. *PC_{LOW WORD} ($4006) sparas undan på den adress som A7 pekar ut ($0000 2FFE)*
4. *register A7 minskas på nytt med 2.*
5. *PC_{HIGH WORD} ($0020) sparas på den adress som A7 pekar ut ($0000 2FFC)*
6. *den inlästa adressen ($0034 5678) placeras i PC.*

Subrutinen "ReadClk" kommer nu att exekveras. I vårt exempel består den endast av `nop` instruktioner. När den sista instruktionen (`RTS`) i subrutinen skall utföras har processorns register följande innehåll. Se figur 4.38.



FIGUR 4.38 PROCESSORNS REGISTER FÖRE RTS-INSTRUKTIONEN.

Vid RTS-instruktionen utförs följande:

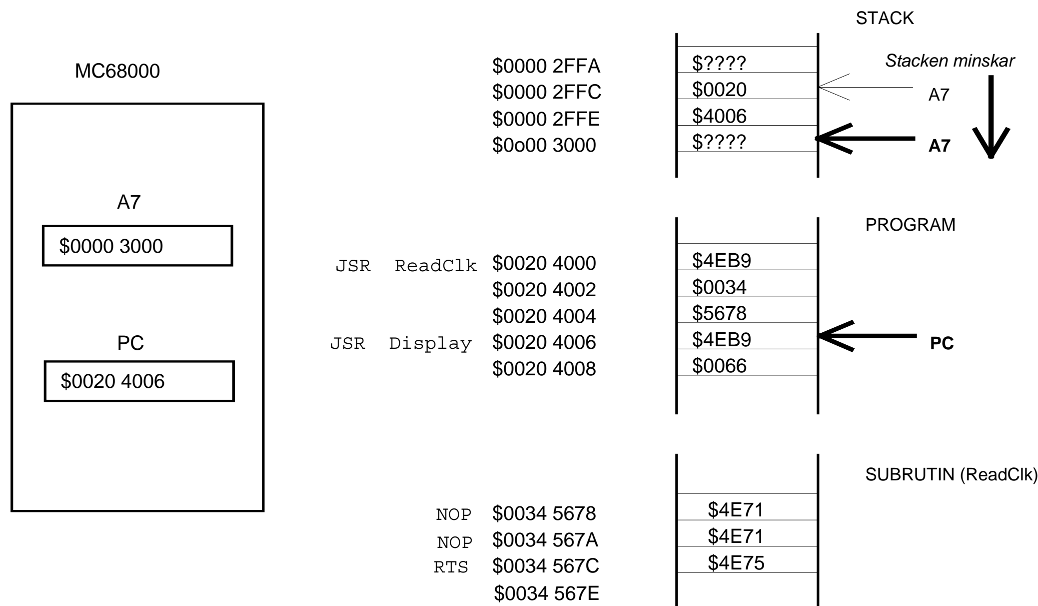
*1) processorn adresserar minnet med stackpekarens (**A7**) innehåll och läser ett* word *($0020) till* ***PC**$_{HIGH\ WORD}$*

*2) register **A7*** (stackpekaren*) ökas med 2.*

3) processorn adresserar minnet på nytt med stackpekarens innehåll och läser ett word *($4006) till* ***PC**$_{LOW\ WORD}$*

*4) register **A7*** (stackpekaren*) ökas med 2.*

RTS-instruktionen utförs och på detta sätt har ett återhopp från subrutinen till huvudprogrammet utförts. Observera att PC nu pekar på instruktionen `JSR Display`, se figur 4.39.



FIGUR 4.39 PROCESSORNS REGISTER DIREKT EFTER RTS-INSTRUKTIONEN.

Observera att stackpekaren på nytt har värdet $0000 3000. Vi har utnyttjat ett minnesarea på stacken och återlämnat detta. TOS är på nytt lik BOS.

Öva själv på de återstående subrutinanropen och rita en bild av stacken och stackpekaren.

Det är fullt möjligt att anropa en subrutin när man är *i* en subrutin. Betrakta följande exempel där den ursprungliga "ReadClk" är utökat med ett anrop av "Subrutin2" som hittas på adress $0038 2240 i minnet. Detta förfarande kallas nästlade subrutiner.

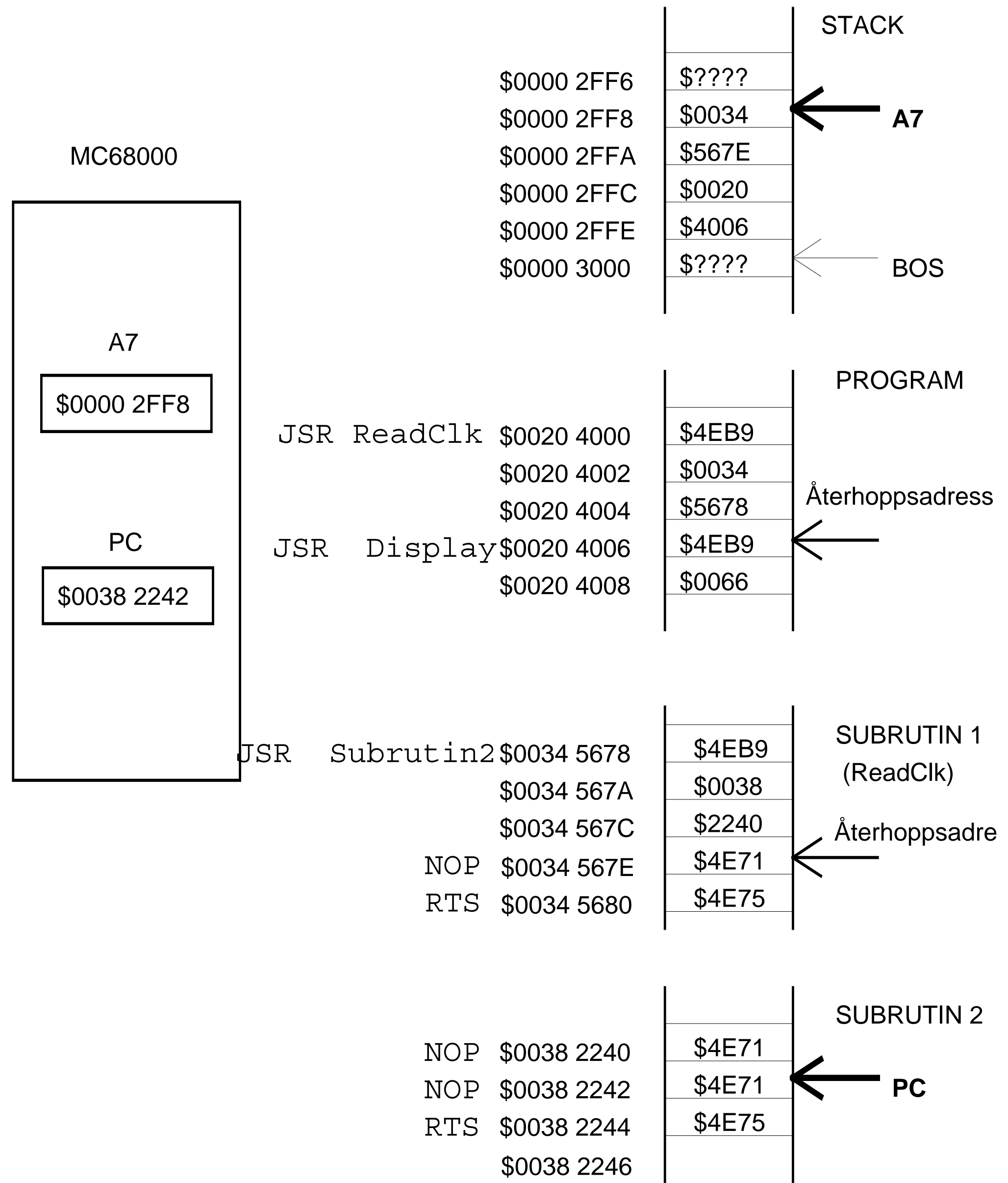
EXEMPEL

Nästlade subrutiner

```
ReadClk:
        JSR     Subrutin2     hoppa till nästa subrutin
        NOP
        RTS

Subrutin2:
        NOP
        NOP     . . .     Studera stackens utseende här
        RTS
```

4.1

Studera nu figur 4.40 som visar stacken och registerinnehåll när processorn exekverar `Subrutin2`. Observera speciellt stackens utseende som nu innehåller två olika återhoppsadresser. TOS är nu $0000 2FF8



FIGUR 4.40 STACKENS UTSEENDE EFTER TVÅ SUBRUTINANROP

- *Stacken växer mot adress noll*
- *Stacken minskar med ökande adress*

När RTS-instruktionen i Subrutin2 utförs hämtas återhoppsadress2 och vi får ett återhopp till ReadClk. När denna rutins RTS-instruktion utförs fås slutligen ett återhopp till huvudprogrammet.

Stackanvändning och subrutiner som är två viktiga begrepp i datortekniken behandlas ett antal gånger i nästa kapitel.

4.3.3 Extern styrning

Vi har tidigare behandlat *programmerade* "hopp", där det är vi som skriver programmen som bestämmer när processorn skall utföra en programflödesändring ("hopp"). Detta gör genom att välja olika hoppinstruktioner, som exempelvis *Jump To Subroutine*.

Exempel på enheter som kan kopplas till processorns avbrottsingångar är

- *en tryckknapp*
- *"klar"-signal från en hårddisk*
- *"data-byte-har-anlänt"-signal från ett datanät*
- *mm*

Här skall vi nu ge en kort översikt av hur vi kan åstadkomma programhopp genererade av externa händelser utanför processorn. Detta betecknas extern styrning. Med extern styrning menas signalpåtvingade programhopp.

Genom att aktivera en av processorns avbrottsinsignaler (pinnar på processorn) avbryts den normala programexekveringen och en ny programrutin (en *avbrottsrutin*) startas. Vidare är avbrottsrutinen som startas, avslutad med en speciell instruktion (`RTE`) som utför ett återhopp till det ställe det första programmet blev avbrutet. Extern styrning eller *avbrott* beskrivs i detalj i ett senare kapitel. Här nedan ges dock en översikt.

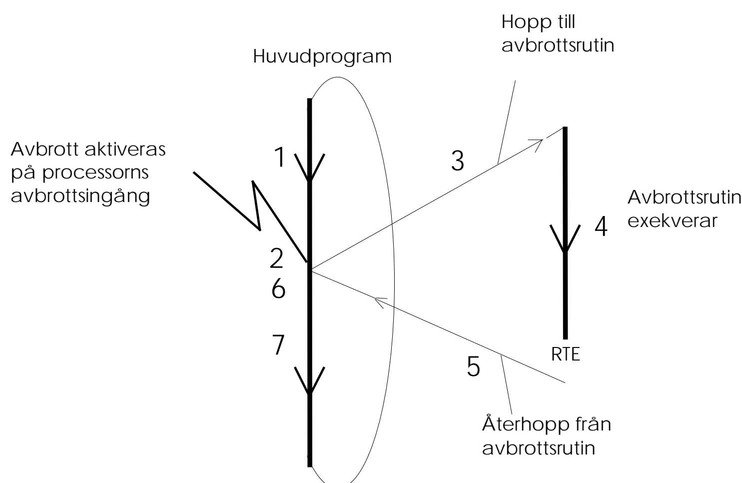
Exeption = Undantag

Interrupt = Avbrott

RTE = ReTurn from Exeption, återhopp från avbrottsrutin (jfr RTS)

IPLx: Interrupt Priority Level

Vanligen är programmet som ett mikrodatorsystem exekverar utformat som en oändlig snurra och kallas för *huvudprogram*. I tillägg till programsnurran så kan det finnas programdelar som ligger utanför snurran. Dessa programdelar kallas avbrottsrutiner och kan startas upp när processorns avbrottsingångar [**IPL0..IPL2**] aktiveras. Se figur 4.41. Om exempelvis **IPL1** aktiveras med en tryckknapp kommer processorn att spara undan sin programräknare (PC). Detta är ju adressen till den instruktion som står i tur att exekveras i huvudprogrammet om inte avbrott aktiverats. Detta är för *återhoppsadressen*. På samma sätt om vid subrutinanrop spara återhoppsadressen på stacken. Efter det läser processorn startadressen för avbrottsrutinen från ett förutbestämt ställe och lägger detta i PC (liknande förfarande som vid RESET).

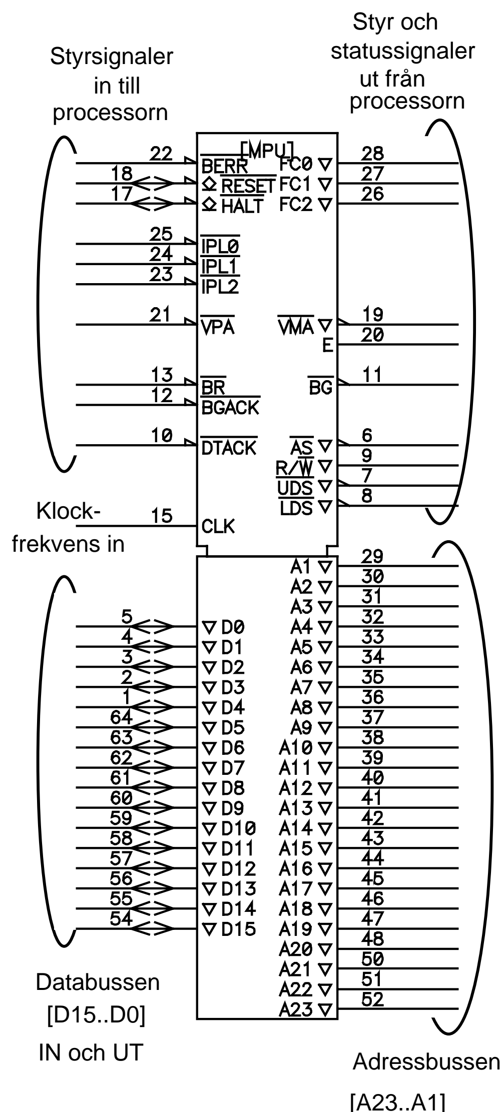


FIGUR 4.41 EXEMPEL PÅ EXTERN (PROGRAM-) STYRNING

På så sätt har processorn nu fått en ny adress i PC och i dagligt tal så säger vi att processorn "hoppar" till avbrottsrutinen. Processorn fortsätter med det enda den kan, nämligen: hämta och utföra instruktioner. Avbrottsrutinen exekveras tills processorn läser in instruktionen `RTE` som avslutar avbrottsrutinen. När processorn utför denna instruktion så hämtas helt enkelt den undanstoppade återhoppsadressen från stacken som läggs i PC. På så sätt kommer nu huvudprogrammet att exekvera vidare från den plats där det blev avbrutet.

Vi har här gett en kortfattad beskrivning på hur avbrott går till. Utöver det vi beskrivit ovan så är MC68000 utrustad med sju olika avbrottsingångar som har inbördes prioriteter. Med tre bitar i processorns statusregister (I_2, I_1, I_0) som kallas *avbrottsmask,* kan avbrott under en viss prioritet att stängas ute. Avbrott beskrivs utförligt i senare kapitel.

4.4 MC68000:s anslutningar



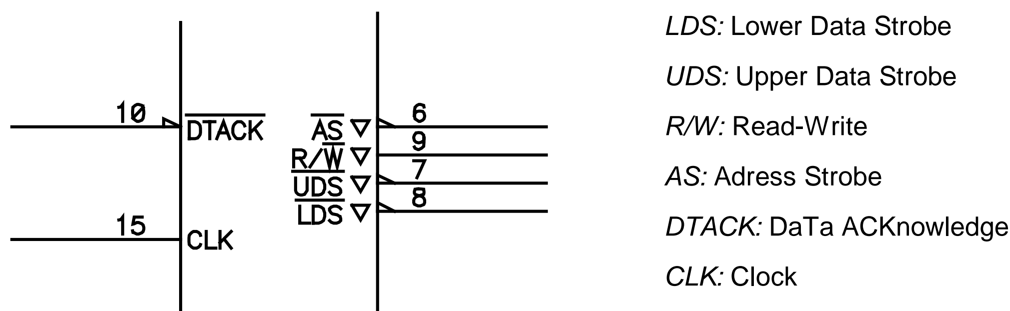
- FIGUR 4.42 PINPLACERING PÅ MC68000

Vi skall nu beskriva processorns “pinning” och sedan visa ett schema över ett minimisystem bestyckad med en MC68000-processor. I figur 4.42 visas processorns alla anslutningar bortsett från 4 anslutningar till +5V och GND. Processorn har totalt 64 anslutningar varav 16 utgörs av databussen, 23 av adressbussen och de övriga är styr och handskakningssignaler med minnet och kringkretsar. Vi skall raskt studera dessa anslutningar.

Databussen [D15..D0] är 16 bitar bred och används för att överföra data till och från processorn, och instruktioner till processorn. Då processorn stödjer så kallad *byte* orienterad adressering, används vid *byte* adressering endast bussens ena halva [D15..D8] eller [D7..D0]. Observera att den lägre delen av databussen [D7..D0] är ansluten till udda adresser och den högre till jämna adresser. Processorn använder sig av signalerna UDS (*Upper Data Strobe*) och LDS (*Lower Data Strobe*) för att adressera *byte* i minnet, Se nedan.

Adressbussen [A23..A1] är 23 bitar bred och kan på därför adressera 2^{23} *word* vilket motsvarar 8388608 *word* eller 8 M*word*. I dagligt tal anger man hur många

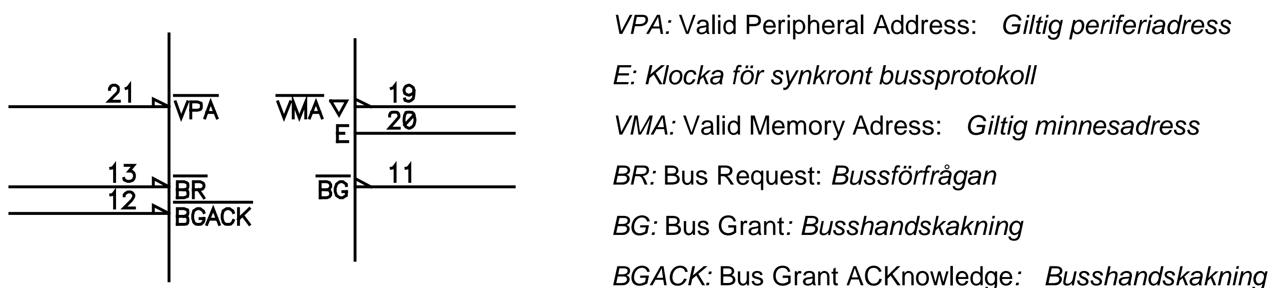
M*byte* en processor kan adressera, och i vårt fall blir det då 16 M*byte*. Processorn använder sig av signalerna UDS och LDS för att adressera *byte* i minnet. Dessa två signaler kan tillsammans sägas bilda adressbit A0.



FIGUR 4.43 HANDSKAKNINGS-SIGNALER FÖR DEN ASYNKRONA DATABUSSEN

Figur 4.43 visar styrsignaler som används för den asynkrona databussen. LDS och UDS (*Lower* and *Upper Data Strobe*) anger om den lägre och/eller den högre *byten* skall adresseras. När LDS är aktiv adresseras en udda adress. R/W (*Read/Write*) signalen anger om data skall läsas eller skrivas i minnet. Signalen AS (*Address Strobe*) indikerar att adressbussen har en giltigt adress. Slutligen DTACK (*Data Transfer Acknowledge*) är en signal från minnet som indikerar att minnet har utfört sin operation korrekt (skrivit in data i ett minnes register eller läst ett minnesregister och placerat data på databussen [D15..D0].

Till pinnen CLK (*Clock*) ansluts en klock signal genererad av en kristall för att uppnå en hög noggranhet i klockfrekvensen. Denna insignal anger processorns arbetstakt. De första MC68000 kunde klockas med 4 MHz och i dag kan de köras på 16 MHz. Observera att denna klocksignal kallas ofta för systemklocka och ansluts till de flesta I/O kretsar (periferikretsar) för att kunna synkronisera sig mot processorns bussprotokoll.



FIGUR 4.44 STYRSIGNALER FÖR BUSPROTOKOLL

Figur 4.44 visar handskaknings-signaler med kringkretsar som *inte* har en asynkron buss och därför inte är direkt anpassade för processorns bussystem. När en sådan kringkrets adresseras skall insignalen VPA (*Valid Peripheral Address*) aktiveras. Processorn som känner av signalen VPA kommer nu att synkronisera bussen till klocksignalen E. Processorn indikerar detta vid att aktivera signalen VMA (*Valid Memory Adress*).

De övriga signalerna används när kringkretsar självständigt önskar använda bussarna. Dessa används exemplevis i DMA sammanhang för att överföra data mellan en snabb hårddisk och minnet. Insignalen BR (*Bus Request*) är en fråga till processorn: Kan jag låna bussarna för att själv överföra data? Utsignalen BG (*Bus Grant*) är ett svar som säger: Ja. det får du. Slutligen indikerar insignalen BGACK (*Bus Grant ACKnowledge*) : Tack, nu börjar jag självständigt att överföra data mellan hårddisken och minnet.

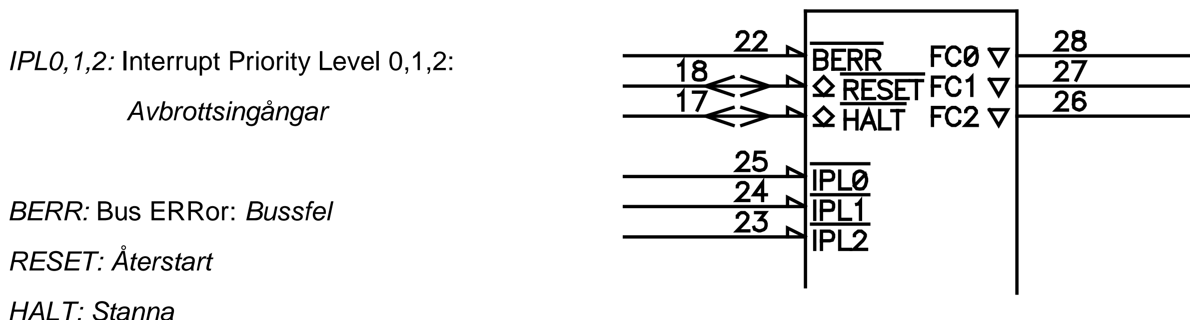
De återstående signalerna används för att externt styra vilka program processorn skall köra. Se figur 4.45. Detta tas upp i senare sammanhang, men i korthet innebär det att processorn avbryter sin normala programexekvering om någon av pinnarna IPL0,1,2 aktiveras (*Input Priority Level 0, 1* och *2*) för att starta ett programavsnitt knutet den externa enheten som aktiverade IPL-ingångarna. När programavsnittet är klart fortsätter den normala programexekveringen från den plats den blev avbruten. Detta förfarande kallas avbrott (*Interrupt*).

IPL0,1,2: Interrupt Priority Level 0,1,2: *Avbrottsingångar*

BERR: Bus ERRor: *Bussfel*

RESET: Återstart

HALT: Stanna



FIGUR 4.45 STYRSIGNALER FÖR EXTERN STRYRNING

Överst i figur 4.45 visas insignalerna BERR, RESET och HALT. Dessa insignaler används också för att externt kunna styra processorn. De innebär omstart av processorn (RESET) och stopp av den (HALT). Observera att_RESET har en dubbelriktad funktion (både ut- och insignal). När den fungerar som utsignal så är det processorn som *begär* att systemets övriga enheter skall utföra RESET.

Även HALT är dubbelriktad. När den är en utsignal är det processorn själv som önskar stanna systemet. Om signalen BERR (*Bus ERRor*) aktiveras (av exempelvis minnet) tolkas detta som att en felaktig adress

är genererad av processorn (*programmeringsfel*) och den startar ett speciellt programavsnitt (*exception processing*).

FC0, FC1, FC2:

(Function Code)

Processorns status-signaler

Slutligen finns signalerna FC0, FC1 och FC2 (*Function Code*) som är processorns statussignaler. Signalernas betydelse kan exempelvis vara *användarmod, systemmod* och *avbrott.*

4.4.1 Ett MC68000 minimisystem

Vi avslutar detta kapitel med att visa ett schema över ett MC68000 minimisystem. Systemet är bestyckat med 64 kbyte RWM, 64 kbyte PROM, en 8-bitars parallell inport och en 8-bitars parallell utport. Schemat visas i figur 4.46, 4.47 och 4.48 uppdelat i *styrlogik*, *minne* och *in- ut-portar*.

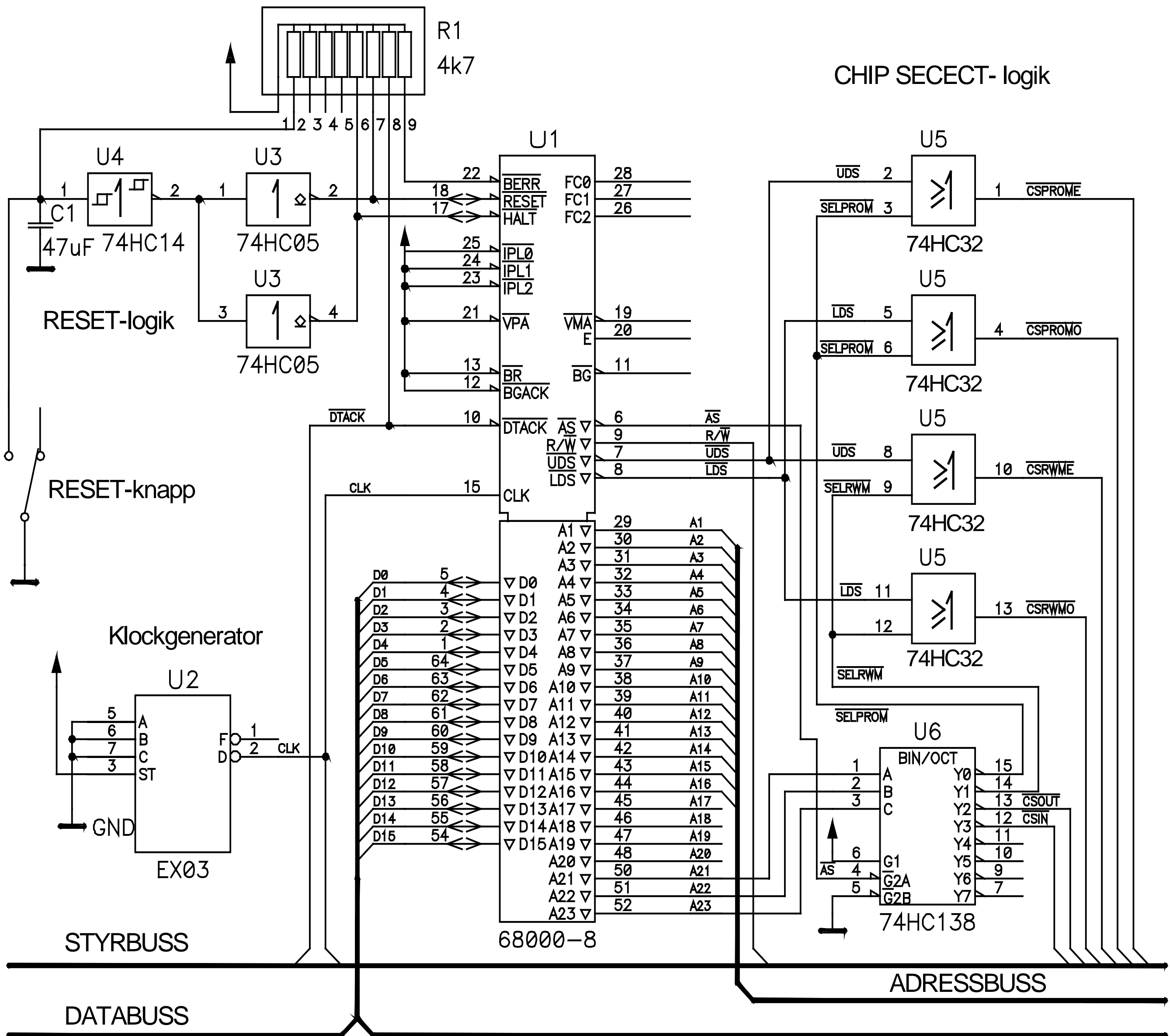
Styrlogik

Figur 4.46 visar processorblocket.Observera 3:8-avkodaren som ingår i adressavkodningslogiken som väljer om processorn adresserar PROM, RWM eller in och utportarna (CSIN och CSOUT). Signalerna SELPROM och SELRWM avkodas ytterligare tillsammans med LDS och UDS för att välja om udda byte, jämn byte eller både och skall adresseras (CSPROMO, CSPROME, CSRWMO och CSRWME). Adresser för respektive krets ges i marginalen. Vi återkommer till adressavkodning i ett senare kapitel.

Krets	*Start-adress*	*Slut-adress*
PROM	*000000*	*00FFFF*
RWM	*200000*	*20FFFF*
Utport	*400001*	
Inport	*600001*	

För fullständig RESET kräver processorn att HALT och RESET-ingångarna går höga samtidigt. Därför finns två “open collector-grindar” (U3) anslutna till RESET-knappen via smitt-triggaren U4.

Som klocka används kretsen U2 som kan byglas för att generera olika klockfrekvenser.

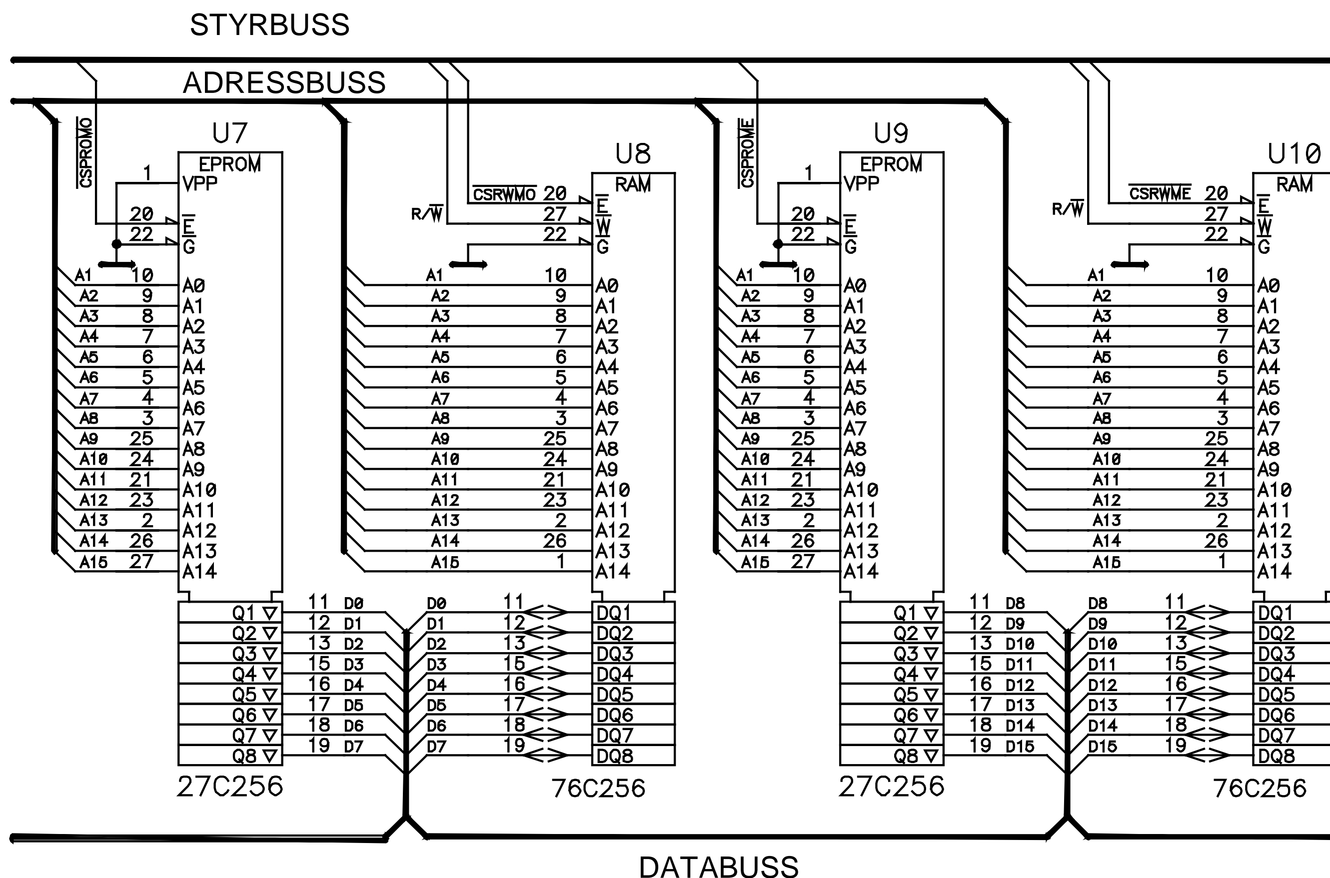


FIGUR 4.46 PROCESSOR OCH RESET- OCH ADRESSAVKODNINGSLOGIK

Minnet

Till vänster i figur 4.47 visas PROM och RWM för den lägre delen av databussen (udda bytes).Till höger visas jämna bytes. Varje minneskrets innehåller 32 kbytes.

Observera aktiveringssignalerna (chip select signalerna) överst på varje minneskrets (CSPROMO, CSRWMO, CSPROME och CSRWMO). Dessa kommer från adressavkodnings-logiken. Vidare är skriv och läs signalen R/W endast ansluten till RWM-kretsarna.



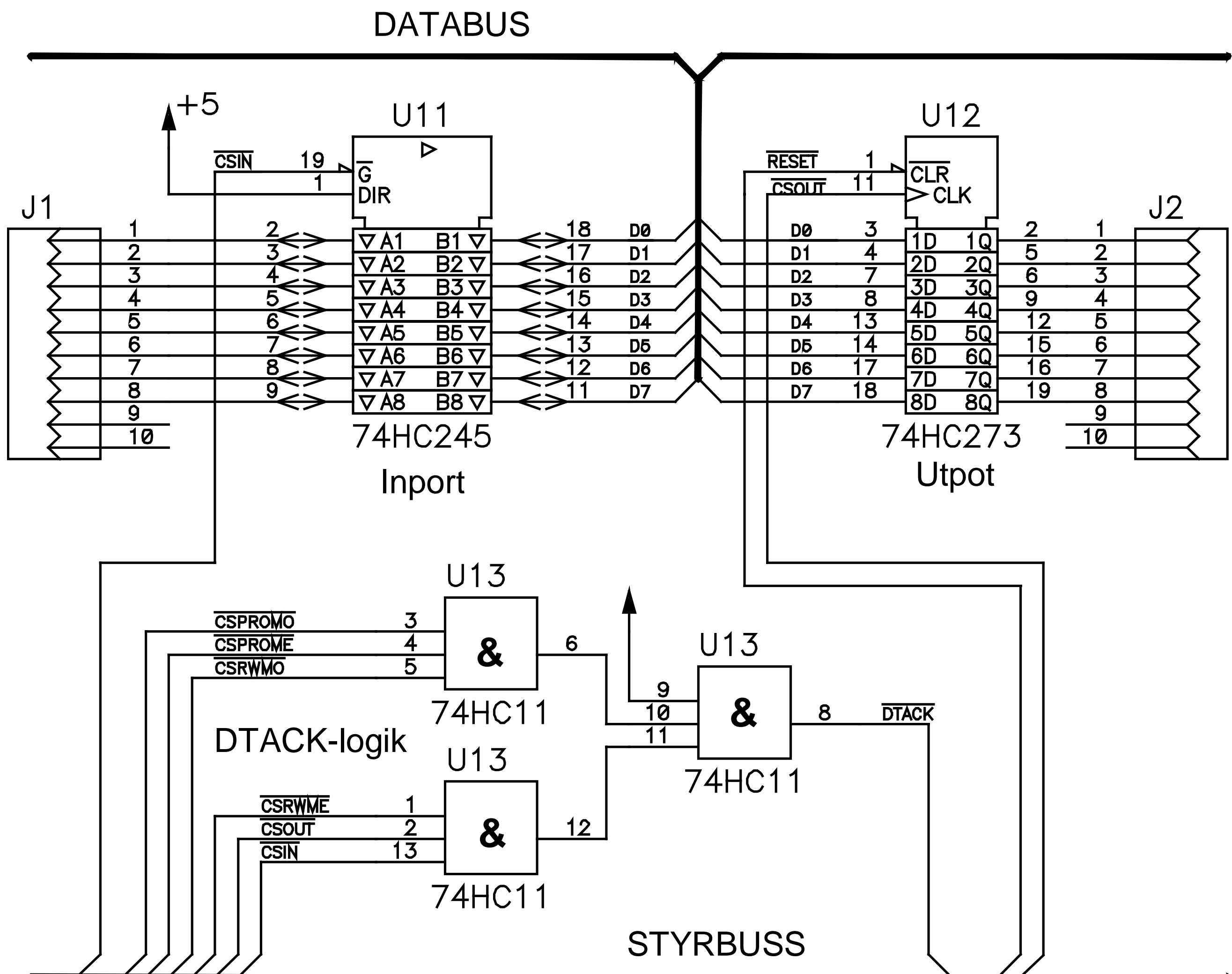
FIGUR 4.47 MIKRODATORSYSTEMETS MINNE

In- Ut- Portar

Överst i figur 4.48 visas in och utporten med tillhörande kontakter. Här används den låga delen av databussen. Detta medför att adresserna för portarna blir udda. Utporten får då adress $400001 och inporten $600001.

Nederst i figuren visas logikblocket som genererar DTACK till processorn. I ett minimisystem som detta kan alla chip-select signaler grindas tillsammans för att generera DTACK. Vi har då förutsatt att vi har tillräckligt snabba minnen. (Portarna dvs registret och bufferten, är

tillräckligt snabba.) Om någon av chipselectsignalerna aktiveras genereras DTACK och processorn avslutar sin busscykel.



FIGUR 4.48 IN OCH UTPORT OCH DTACK-LOGIK

Vi har här utelämnat vissa block som oftast ingår i 68000-system. Dessa är exempelvis: avbrott, busfel, synkront bussprotokoll mm. Vi återkommer till dessa block längre fram i senare kapitel.

4.5 Sammanfattning

Detta kapitel har beskrivit MC68000:s arkitektur, dess arbetssätt och anslutningar.

MC68000 har en 32-bitars intern arkitektur. Den har åtta 32-bitars dataregister och sju 32-bitars adressregister förutom programräknaren. Då processorn kan arbeta i antingen användarmod eller systemmod finns två olika stackpekarregister (SP och USP). Slutligen finns statusregistret som är uppdelad i en användarbyte och en systembyte.

Processorn kan arbeta med byte, word och long word. Word och long word lagras i minnet med början på en jämn adress.

En typisk instruktion för MC68000 består av OP-kod och eventuell operandinformation. OP-koderna kan delas in i 16 olika grupper där bit [15..12] anger grupp. Operandinformationen kan bestå av källoperand och destinationsoperand. Var operanderna hittas anges i OP-kodens EA-bitfält som är uppdelat i Register- och Modbitar.

En instruktion upptar ett till fem word i minnet. Dessa betecknas som OP-kod och eventuella extensionsord.

Processorn befinner sig i ett av tre tillstånd: Normal, Exception (Undantag) eller Halt (Stannad). Processorn lämnar normalt tillstånd när den upptäcker interna fel, exempelvis division med noll, och övergår till Exception. Tillståndsändring fås även genom att påverka processorns avbrottsingångar. Om inga fel inträffar under tillståndet Exception återgår processorn till Normalt tillstånd, annars stannar den (Halted).

Återstart av processorn sker genom att aktivera dess RESET-ingång. Då läses två long word från adress 0 och 4. Dessa placeras i stackpekar registret (A7) och i programräknaren (PC).

MC68000 stöder både synkront och asynkront bussprotokoll. Det asynkrona protokollet utnyttjar handskakningssignalerna AS (Adres Strobe), LDS (Lower Data Strobe), UDS (Upper Data Strobe) och DTACK (Data Acknowledge). AS anger att adressbussen är giltig och DTACK anger att databussen är giltig. LDS och UDS anger hurvida [D7..D0] och/eller [D15..D8] används.

Vid subrutinanrop används instruktionerna JSR och RTS. Återhoppsadressen lagras på stacken och stackpekarregistret A7 pekar ut den senaste lagrade återhoppsadressen.

Processorn har 64 anslutningar, varav adressbussen utgör 23 pinnar och databussen 16. Processorns styrsignaler kan grupperas enligt: styrsignaler för bussprotokoll, styrsignaler för extern styrning, styrsignaler för bussbyte och statussignaler.

ASSEMBLERPROGRAMMERING

I detta kapitel behandlas maskinnära programmering av MC68000-baserade mikrodatorer. Tonvikten läggs på grunderna för assemblerprogrammering och kopplingen till motsvarande programkonstruktioner i programspråket 'C'. Kapitlet är indelat i fyra delar:

- ***Introduktion till assemblerprogrammering** ger ett kort inledande exempel på assemblerprogram.*
- ***MC68000 - programmerarens bild** presenterar MC68000 ur programmerarens synvinkel. Du bör läsa avsnitter översiktligt utan att fördjupa dig i detaljer som kan vara oklara första gången. Du får säkert anledning att återkomma till detta avsnitt då du studerar de två påföljande avsnitten.*
- ***Programmering i assembler**, behandlar assemblerprogrammering av MC68000.*
- ***Kombinerad programmering** behandlar såväl **C** som MC68000-assembler och vad du behöver veta då du samtidigt utnyttjar båda programmeringsspråken.*

Assemblator: *Översätter assemblerprogram till maskinprogram.*

I det första avsnittet (5.1) beskriver vi hur man skriver ett komplett assemblerprogram för MC68000. Vi inleder med att behandla några av de direktiv som vi kan styra assemblatorn med.

I avsnitt 5.2 behandlas *programmerarens bild*, MC68000's *instruktionsgrupper* med en snabb översikt av instruktions-uppsättningen. Avsnittet avslutas med en utförlig beskrivning av processorns olika *adresseringssätt*.

Kapitlet domineras av avsnitt 5.3 som beskriver ett strukturerat sätt att programmera MC68000 i assembler. Här behandlas bland annat aritmetiska operationer, jämförelser och test, och hur du styr programflödet. Efter en första genomläsning kan du utnyttja detta avsnitt som ett uppslagsverk för att hitta svaren på hur du löser olika problem då du konstruerar dina assemblerprogram.

Kapitlet avslutas med ett avsnitt (5.4) som behandlar "kombinerad programmering", dvs hur du blandar kod skriven i programspråket 'C' med dina assemblerprogram. Här beskrivs bland annat hur parameteröverföring utförs mellan olika funktioner (procedurer) i C.

5.1 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.

Det finns strikta 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 kan enbart användas för att ange en "etikett". Man väljer då ett *symboliskt namn* och kan därefter använda detta namn för att ange (referera) denna position i programmet. Anledningen till att man använder sådana symboliska namn är att man då slipper skriva *absoluta* minnesadresser i programmet.

EXEMPEL

Assemblerradens olika fält

start:	MOVE.B	($FFFFF011).L,D0	kommentar...
symbol fält	*instruktion eller direktiv*	*operandfält*	*ev. kommentarer*

5.1

Assemblerradens andra fält ska innehålla antingen en instruktion för mikroprocessor'n *eller* ett direktiv till assemblatorn. Det är viktigr att förstå skillnaden mellan en assemblerinstruktion och ett assemblerdirektiv. Direktivet instruerar assemblatorn att göra någonting vid assembleringstillfället medan instruktionen översätts till maskinkod för att så småningom utföras av processorn vid exekveringen av programmet.

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

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

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

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 vanligtvis *inte* får förekomma i symbolnamn

EXEMPEL

Tillåtna symbolnamn

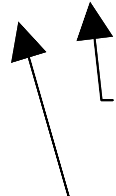
```
start
stopp
prog10
_prog10
```

5.2

EXEMPEL

Definition av symbol

```
start:
```



Kolon (:), kan, men *behöver inte* anges, det tolkas aldrig som en del av symbolnamnet.

Symbolnamnet ("start") måste börja i radens första position för att assemblatorn ska tolka det som en symbol.

5.3

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.

EXEMPEL

Felaktig (multipel) definition av symbol

```
start:

start:
```

5.4

Några vanliga assemblerdirektiv

Assemblerdirektiv används för att instruera assemblatorn på olika sätt. Det finns flera olika direktiv men här behandlar vi bara de vanligaste. Du kan läsa om assemblerdirektiv i handboken för din assemblator.

EXEMPEL

Några vanliga assemblerdirektiv

```
        ORG     $3000

        DC.B    2

        DS.B    2

InPort  EQU     $FFFFF011
```

"`ORG`" (origin) används för att ange *startadress* för kod eller data, i laborationsdatorns primärminne.

"`DC.B`" (*define constant byte*) används för att placera en konstant (data) i laborationsdatorns primärminne.

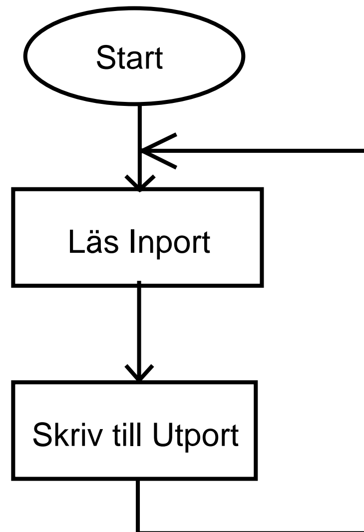
"`DS.B`" *(define storage byte*) används för att reservera utrymme för variabler (data) i laborationsdatorns primärminne

Direktivet "`EQU`" (equate) används för att ge ett symboliskt namn för någon konstant. Ofta är det en fast adress i minnet som exempelvis in-/ut-portar. Symbolen "`InPort`" kan i detta exempel senare användas i stället för att skriva den absoluta adressen $FFFFF011.

5.5

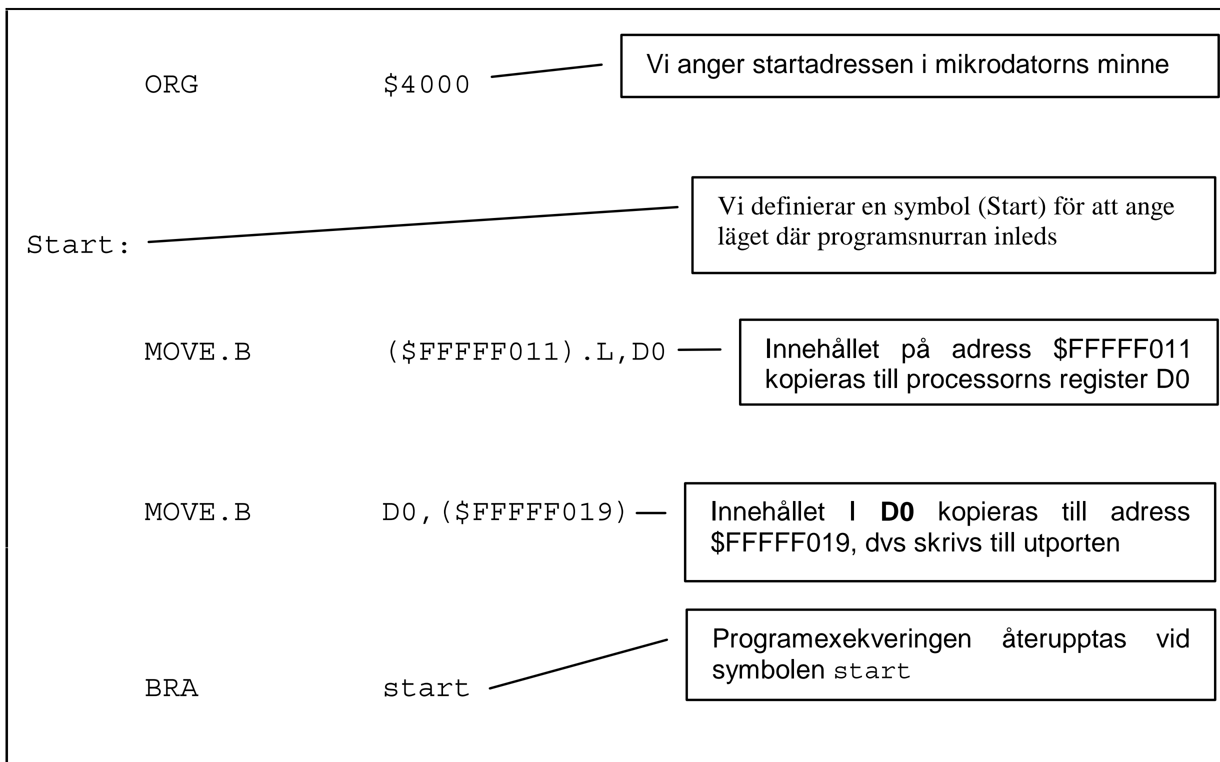
Första assemblerprogrammet

Vårt första assemblerprogram visar exempel på hur vi kan läsa data (8 bitar) från en inport placerad på adress $FFFFF011 i minnet, vi skriver därefter samma data till en utport på adress $FFFFF019 i minnet, detta upprepas i en "oändlig slinga" (Se flödesplan i figur 5.1).



FIGUR 5.1 FLÖDESPLAN FÖR FÖRSTA ASSEMBLERPROGRAMMET

Assemblerkod, det första assemblerprogrammet:



Observera hur portadresserna anges 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. För att undvika dessa problem kan vi i stället definiera adresserna symboliskt med EQU-direktivet. Därefter

kan assemblatorn ersätta symbolerna med rätt värden och en ändring blir enkel att utföra. Vi skriver därför om assemblerprogrammet och samlar då alla EQU-satser (EQU-direktiv) i *början* av programmet så att de blir lättöverskådliga och enkla att hitta.

```
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 förstå vad det utför.

Kommentarsrader

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

EXEMPEL

Första assemblerprogrammet i färdig form

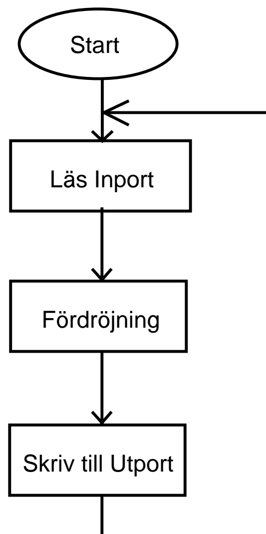
```
* Programmet läser från inport och kopierar till utport

InPort EQU          $3E0013
OutPortEQU          $3E0011

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

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

Fälten separeras med TAB eller SPACE

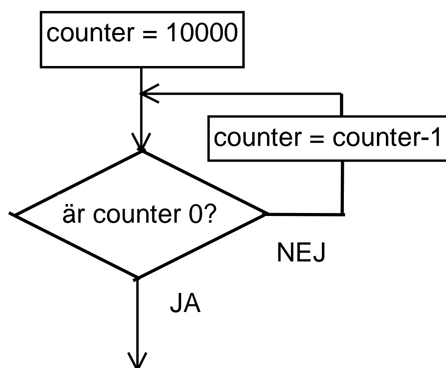
5.6

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 i figur 5.2.



FIGUR 5.2 FLÖDESPLAN MED FÖRDRÖJNING

Fördröjningen kan enkelt åstadkommas genom att vi skapar en *villkorlig* programslinga (se figur 5.3). 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.



FIGUR 5.3 FÖRDRÖJNING KAN ÅSTADKOMMAS MED EN VILLKORLIG PROGRAMSLINGA

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

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

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

EXEMPEL

Jämförelseinstruktionen ”compare”

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

Operation:

0 subtraheras från innehållet i register D0, flaggorna i CCR påverkas av operationen dvs: “jämför innehållet i D0 med 0”. Innehållet i D0 påverkas inte.

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

5.7

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

- Testar villkoret mot innehållet i flaggregistret (**CCR**)
- Om resultatet av testen är SANT, utförs instruktionen (programhoppet sker)
- 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 program:

```
      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
```

Exempel 5.8 visar en sekvens instruktioner som skapar en fördröjning.

EXEMPEL

```
* 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
```


5.8

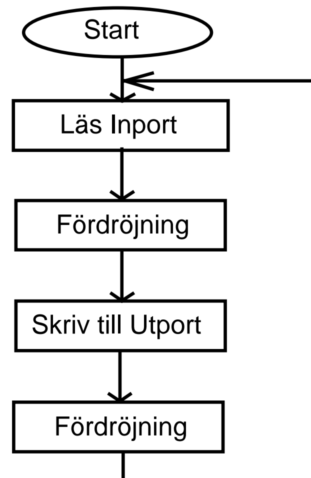
Kombineras fördröjningen med med vårt tidigare program fås följande program (exempel 5.9):

EXEMPEL

```
* Litet program som läser från en inport
* och kopierar värdet till en utport

CODE        EQU         $4000
InPort      EQU         $3E0013
OutPort     EQU         $3E0011

            ORG         CODE
Start:
            MOVE.B      (InPort).L,D0 Läs
            MOVE.L      #10000,D1
delay:      CMPI.L      #0,D1           färdig ?
            BEQ         delay_exit      avbryt i så fall
            SUBI.L      #1,D1           annars ..
            BRA         delay           fortsätt

delay_exit: MOVE.B      D0,(OutPort).L  skriv
            BRA         Start           börja om
```

5.9

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*). Vi avslutar detta avsnitt med att modifiera vårt program enligt flödesplanen i figur 5.4. Det färdiga programmet visas i exempel 5.10.



FIGUR 5.4 FLÖDESPLAN FÖR PROGRAM MED FLERA FÖRDRÖJNINGAR

EXEMPEL

```
* Litet program som läser från en inport
* och kopierar värdet till en utport

CODE        EQU       $4000
InPort      EQU       $3E0013
OutPort     EQU       $3E0011

            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  avbryti så fall
            SUBI.L    #1,D1       annars ..
            BRA       delay1      fortsätt
delay_exit: RTS
```

5.10

5.2 MC68000 – programmerarens bild

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

5.2.1 Supervisor/User Mode

MC68000 arbetar alltid i en av två "moder". *Supervisor Mode*, i vilken samtliga register är tillgängliga och samtliga instruktioner kan utföras. Dessutom finns den något begränsade *User Mode*, i vilken vissa instruktioner inte längre är möjliga att utföra. Processorn använder olika stackregister i de olika moderna. I programmet kallas visserligen alltid stackpekaren A7 (eller SP) men processorn skiljer hårdvarumässigt på dessa register. I vissa texter kallas A7 (*user mode*) för A7' för att indikera skillnaden, men i assemblerprogrammet ser man ingen skillnad på dessa register. Då processorn är i *supervisor mode*, kan A7', dvs den stackpekare som ska användas i *user mode* manipuleras, den kallas då USP.

Vi återkommer till processorns moder vid behandlingen av exceptions. För närvarande kan vi utgå från att processorn alltid arbetar *i supervisor mode*.

5.2.2 Registermodell

MC68000 har totalt 17 st 32-bitars register, en 24-bitars programräknare och ett 16-bitars statusregister. Programräknaren (**PC**) som uppdateras av processorn, anger vilken adress som pekas ut för nästa instruktionshämtning eller eventuellt data till den aktuella instruktionen. Statusregistret **SR** är uppdelat i två delar, *system byte* och *user byte*. **SR** speglar processorns status och såväl *system byte* som *user byte* är både läs- och skrivbara. Utöver detta finns 17 mer eller mindre generella register, av dessa utgörs 8 st av *dataregister*, de övriga 9 av *adressregister*. Figur 5.5 nedan visar registeruppsättningen.

Statusregister

I MC68000:s statusregister (SR) (Figur 5.6) hålls information om processorns tillstånd. SR innehåller 16 bitar, men alla bitar används inte. Hela **SR** är alltid läsbart, men de 8 mest signifikanta bitarna (*system byte*) kan endast skrivas då processorn är i *supervisor mode*. Processorn kan också vara i *user mode* och skillnaden mellan dessa är att vissa av processorns instruktioner ej kan utföras i *user mode*. Om ett sådant försök

Speciella instruktioner för SR

i supervisor mode

MOVE SR,datareg

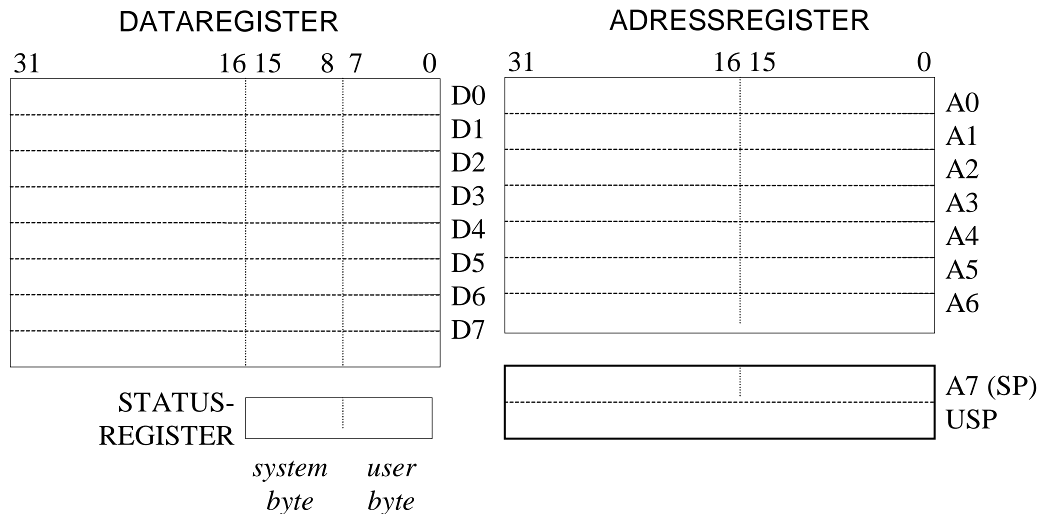
MOVE datareg,*SR*

i user mode

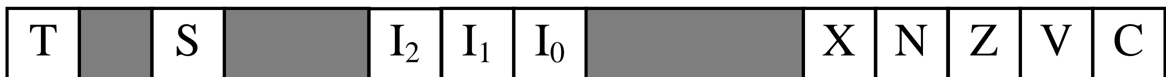
MOVE CCR,datareg

MOVE *datareg*,CCR

görs, kommer processorn att försättas i *supervisor mode* och därefter börja utföra en fördefinierad rutin. Med denna möjlighet kan man implementera olika typer av skydd. Exempelvis, skydd mot att ett felaktigt skrivet program förstör viktiga datastrukturer i systemet.



FIGUR 5.5 REGISTERUPPSÄTTNING HOS MC68000



FIGUR 5.6 MC68000 STATUSREGISTER (S**R**)

- **C**: *Carry*-bit, sätts om ALU-operation genererat carry.
- **V:** *Overflow*-bit, sätts om resultatet av en ALU-operation är s.k tvåkomplementspill,
- **Z:** *Zero*-bit, sätts om resultatet av en ALU-operation är noll.
- **N:** *Negative*-bit, sätts om resultatet av ALU-operationen är ett negativt tal, dvs resultatets mest signifikanta bit är 1.
- **X:** *Extend*-bit, har samma funktion som C-biten med den viktiga skillnaden att X inte påverkas av lika många instruktioner som C. Detta gör att resultatet av en aritmetisk operation sparas i X-biten under exekvering av flera instruktioner som kan ändra C-biten.

- I_2 I_1 I_0: *Interrupt Priority Level* Utgör processorns *avbrottsprioritetsnivå*. 7 olika avbrottsnivåer existerar. För att ett avbrott ska betjänas måste avbrottet ha högre prioritet än processorns prioritetsnivå. Om processorns avbrottsmask I_2 I_1 I_0 är satt till den högsta nivån (`111`), kommer dock avbrott med prioritet 7 att betjänas. Avbrottsprioritet 7 kallas därför också *non-maskable*

interrupt, eftersom dessa avbrott inte kan maskas bort i processorns avbrottsmask. Om processorns avbrottsmask är satt till `000`, kommer alla avbrott att betjänas. Avbrottshantering behandlas detaljerat i kapitel 7 och 8.

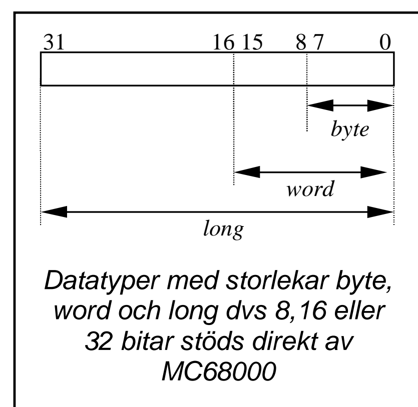
- **S:** *Supervisor mode*, om denna bit är 1, är processorn i *supervisor mode*. Då biten sätts till 0 försätts processorn i *user mode*. Då processorn är i *user mode* kan inte **SR** *system byte* skrivas. Processorn försätts i *supervisor mode* igen exempelvis genom försök att utföra en otillåten instruktion, exekvering av speciell instruktion eller då ett *avbrott* uppträder.

- **T:** *Trace*, om denna bit är 1 kommer processorn att utföra *en* instruktion och därefter uppta exekveringen på en förutbestämd adress. Trace-biten sätts vanligtvis bara av speciella *debugger*-program Vid normal exekvering är biten 0.

De 8 minst signifikanta bitarna i **SR**, (*user byte*) utgör i sig ett register, *condition code register* (**CCR**). Detta register innehåller speciella flaggor, som påverkas av processorns ALU vid instruktionsexekvering. Flaggorna används för att spegla resultatet av en operation, (speciella testinstruktioner, aritmetiska instruktioner, skiftinstruktioner etc). Olika tester används därefter av så kallade *villkorliga instruktioner*. Exempelvis kan *overflow*-flaggan testas efter en addition av två tal för att se om resultatet av multiplikationen är korrekt eller om spill uppträtt med felaktigt resultat som följd. Vi återkommer till flaggsättning och tester längre fram.

Dataregister

De mest generella registren i MC68000:s registeruppsättning utgörs av *dataregister*. Instruktioner kan operera på hela eller delar av dataregister. Exempelvis kan aritmetik utföras med användning av de 8 eller 16 minst signifikanta bitarna i registret såväl som på samtliga 32 bitar. Därutöver finns speciella instruktioner i instruktions-uppsättningen vilka opererar på enskilda bitar i ett godtyckligt dataregister.



Datatyper med storlekar byte, word och long dvs 8,16 eller 32 bitar stöds direkt av MC68000

Begreppet datatyper introduceras ofta med användningen av högnivåspråk. Exempel på högnivåspråk är **C**, **Pascal**, **ADA**, **Cobol**, **LISP**, **APL** etc. Även om begreppet datatyp är en abstraktion är det

viktigt att komma ihåg vad som motiverar begreppet. Grundläggande för alla datatyper är den representation som kan ges typen i den underliggande hårdvaran. Den i särklass vanligaste datatypen är *heltal*. Maskinvaran sätter gränser för hur små eller hur stora heltal som enkelt kan representeras, detta avspeglas direkt utav det antal bitar som används vid exempelvis aritmetiska operationer. För MC68000 gäller att full 32-bitars aritmetik kan utföras som enkla operationer (instruktioner) då det gäller addition och subtraktion. För multiplikation och division är situationen annorlunda. MC68000 tillhandahåller instruktioner för 16-bitars multiplikation respektive 16-bitars division. Att implementera dessa operationer på 32-bitars tal kräver speciella programrutiner. Inte desto mindre ter det sig naturligt att välja 32-bitars representation för heltal i MC68000-baserade mikrodatorsystem.

De 8 dataregistren, betecknade **D0**,**D1**,**D2**,**D3**,**D4**,**D5**,**D6** och **D7** är alltså typiskt avsedda för att hålla variabler av olika datatyper men eftersom full 32-bitars aritmetik finns tillgänglig *kan* även dataregister användas exempelvis för beräkning av adresser etc.

EXEMPEL

Instruktionen:

```
CLR.B D0
```

(*clear byte D0*) nollställer de 8 minst signifikanta bitarna i dataregister D0. Övriga bitar lämnas opåverkade av instruktionen.

Instruktionen:

```
ADD.W D0,D0
```

(*add word D0 to D0*) adderar innehållet i **D0** till sig självt, dvs multiplicerar innehållet med två, och placerar resultatet på nytt i **D0**. Det är emellertid endast de 16 minst signifikanta bitarna som påverkas av instruktionen.

5.11

Adressregister

Det finns 8 olika adressregister tillgängliga för programmeraren. Ett av dessa register har en mycket speciell betydelse (register **A7**) medan de övriga sju används som generella *pekar-register*. Adressregistren benämns **A0**, **A1**, **A2**, **A3**, **A4**, **A5**, **A6** och **A7**. Adressregister **A7** är systemets *stackpekare*. Stackpekaren har en central roll vid utförandet av ett program. Den utgör en direkt referens till olika adresser i programmet där det finns data lagrat eller eventuellt adresser där exekvering skall utföras. Eftersom MC68000 kan vara i ett av två tillstånd (en av två *moder*), tillhandahålls två olika register för

stackpekare. Normalt betecknas stackpekaren som **A7** (eller det synonyma **SP**), men det angivna registret utgörs av två fysiskt olika register. Om processorn är i så kallad *supervisor mode of operation* (*system mode*) används **A7'**. Om däremot processorn är i *user mode of operation* (*användar mode*), används **A7''**. För programmeraren har denna skillnad vanligtvis ingen betydelse, rätt register väljs av hårdvaran i MC68000. Det finns emellertid ett specialfall där en åtskillnad måste göras av programmeraren: Om processorn befinner sig i *supervisor mode* och man avser ändra den stackpekare som aktiveras i *user mode*, måste en speciell instruktion användas (**move USP**), i detta fall betecknas stackpekaren **USP**.

5.2.3 Användning av register

I programspråket 'C' finns datatypen `int` (integer). En variabeldeklaration enligt:

```
int   var_A;
```

innebär i ett MC68000-baserat system (som regel) att 32 bitar reserveras för variabeln `var_A`. En 32-bitars datatyp stöds av MC68000 med storleken *long*. För att bättre tillvarata programmerarens avsikter infördes tidigt i programspråket C en kortare datatyp för heltal, `short int`. Denna datatyp kan användas då programmeraren med säkerhet vet att 16 bitars representation räcker för den variabel han tänkt använda. Datatypen `short int` stöds direkt av MC68000 med storleken *word*. Exempelvis utförs en multiplikation av två variabler, typ `short int`, med en enda instruktion, medan en full 32-bitars multiplikation kräver åtskilliga maskininstruktioner (exempel på full 32-bitars multiplikation ges senare i denna lärobok). En annan datatyp, av stor betydelse är `char`**.** Den är till för att representera *ett* tecken (maximalt 8 bitar) som kan tas emot från eller skrivas till en enkel bildskärm (terminal). Denna datatyp stöds av MC68000 och storleken är *byte*. För såväl `short` som `int` och `char` gäller att dessa kan förgås av bestämningen `unsigned`. Begreppet påverkar inte representationen i MC68000 men väl hur instruktionsföljder väljs och vilka flaggor i statusregistret som testas av villkorliga instruktioner.

En variabel av pekartyp, innehåller en pekare till något objekt. Med MC68000 är det lämpligt att använda adressregister för pekartyper. Typen är då 32-bitar lång. Observera att alla pekartyper har samma storlek, oberoende av vilken typ av objekt de pekar på.

EXEMPEL

Grundläggande datatyper med MC68000

```
char        c;        /* 8-bitars datatyp, storlek byte */
 short      s;        /* 16-bitars datatyp, storlek word */
 int        i;        /* 32-bitars datatyp, storlek long */
 char       *cptr;    /* pekar (32 bitar) på 8-bitars datatyp */
 short      *sptr;    /* pekar (32 bitar) på 16-bitars datatyp */
 int        *iptr;    /* pekar (32 bitar) på 32-bitars datatyp */
```

Generella adressregister (A0-A6) används för pekartyper och **Dataregister (D0-D7)** används för övriga typer

5.11

Utöver de ovan nämnda datatyperna har vi en som kan bestå av ett godtyckligt antal grundläggande datatyper. I **C** kallas en sådan `struct`. Datatypen `struct` stöds av MC68000 med *speciella adresseringsmoder*. Avsikten är att en pekare till det första elementet i `structen` placeras i något adressregister och utgör basadress, en konstant offset adderas till denna basadress. Konstanten kan bestämmas vid kompileringen eftersom storleken av alla datatyper som ingår i `structen` är kända. Om den aktuella variabeln är en *vektor* kan ett speciellt indexregister anges för att markera det speciella element i structens vektor som avses.

Instruktionsformer

I MC68000:s instruktionsuppsättning finns instruktioner med *ingen, en* eller *två* operander.

Den generella formen för en MC68000 instruktion med *två* operander är

mnemonic ***källoperand,destinationsoperand***

där *mnemonic* anger vilken instruktion det är frågan om. Denna svarar precis mot en *maskininstruktion* (jfr kapitel 4). Operanderna anger *var* data skall läsas/och eller skrivas, av instruktionen.

EXEMPEL

Instruktionsform med två operander

```
ADD.L   D0,D1
```

mnemonic ***källoperand*** ***destinationsoperand***

5.12

Såväl destinationsoperand som källoperand ingår i själva operationen. Resultatet av operationen placeras, ofta men inte alltid, i destinationsoperanden. Undantag från detta gäller exempelvis s.k jämförelseoperationer, där resultatet av operationen endast påverkar processorns flaggregister (**CCR**) och destinationsoperanden alltså är opåverkad.

EXEMPEL

Antag innehåll i D0=10, innehållet i D1=5 och att följande instruktion utförs:

```
ADD.L       D0,D1       Addera innehåll i register
```

Operationen som utförs: (D1)+(D0)→D1
dvs: D1 innehåller efter operationen 15. Flaggor i CCR sätts.

Källoperanden påverkas inte av operationen medan destinationsoperandens innehåll dels används i operationen, dels modifieras av operationen.

5.13

Formen för instruktioner med *en* operand är:

mnemonic *operand*

EXEMPEL

Instruktionsform med en operand

```
NOT.B   D0
```

mnemonic ***operand (källoperand och destinationsoperand)***

5.14

För de flesta en-operandsinstruktioner gäller att operanden läses, operationen utförs och operanden uppdateras med det nya resultatet. Även här finns dock undantag i form av testinstruktioner vilka endast påverkar flaggorna i **CCR**.

EXEMPEL

Antag de 8 minst signifikanta bitarna i D0 är binärt: `11110000`
Instruktionen:

```
NOT.B D0
```

inverterar de 8 minst signifikanta bitarna i D0
Instruktionen modifierar operanden.
Efter instruktionen innehåller de 8 minst signifikanta bitarna i D0 alltså (binärt) `00001111`.

5.15

Det finns slutligen instruktioner som *saknar* operand, exempel på sådana är exempelvis `NOP (No Operation)`, `RTS (ReTurn from Subroutine)`, `RESET` (som utför RESET på processorns omgivning), mm.

`NOP`	som inte utför någonting
`RTS`	återhopp från subrutin
`RESET`	återställ periferikretsar

5.2.4 Instruktionsgrupper

MC68000:s instruktionsuppsättning består av ett antal maskininstruktioner som använder en eller två operander eller eventuellt saknar operand. Eftersom maskininstruktionen utgör den minsta (odelbara) operation maskinen kan utföra är det viktigt att varje instruktion konstruerats med rätt funktionalitet.

Det finns en rad omständigheter som påverkar såväl valet av instruktioner ur en instruktionsuppsättning som de operander som kan användas till instruktionerna. I detta avsnitt redogörs kortfattat för de *grupper* av instruktioner som man kan identifiera i MC68000:s instruktions-uppsättning. För en fullständig beskrivning av varje instruktion (varianter, operander, flaggsättning mm.) hänvisas till *instruktionslistan*.

Instruktioner för dataförflyttning

`MOVE` (*move data*) används för att kopiera data från en plats i minnet till en annan plats. Instruktionen är mycket generell och flera varianter finns: `moveq` för små konstanter, `movea` för adresser (pekare), `movep` för periferikretsar och `movem` för att kopiera till respektive från flera register, i en instruktion. Utöver detta finns speciella move-instruktioner för att kopiera data till/från registren **SR**,**CCR** och **USP**.

`EXG` (*exchange*) används för att skifta innehåll mellan register. Såväl data- som adressregister kan anges, exempelvis innebär

```
EXG  D4,A3
```

att innehållet i **A3** placeras i **D4** samtidigt som innehållet i **D4** placeras i **A3**.

SWAP-instruktionen (*swap operand*) utförs på ett dataregister. Instruktionen skiftar innehållen i de 16 mest signifikanta bitarna respektive 16 minst signifikanta bitarna.

LEA (*load effective address*) används för att ladda ett adressregister.

PEA (*push effective address*) placerar innehållet i ett adressregister på stacken och minskar stackpekaren.

Instruktionerna LINK (*link and allocate*) och UNLK (*unlink*) används för att skapa så kallade aktiveringsposter på stacken. Aktiveringsposter innehåller exempelvis parametrar till funktioner och lokala variabler.

Aritmetiska operationer på heltal

Det finns en rad instruktioner för aritmetiska operationer på heltal. Flera av instruktionerna används dessutom i olika varianter.

ADD (*add binary*) och SUB (*subtract binary*) används för addition respektive subtraktion. Varianterna ADDI (*add immediate*) och SUBI (*subtract immediate*) används då källoperanden är ett heltal, varianterna ADDQ (*add quick*) och SUBQ (*subtract quick*) används då källoperanderna är små heltal. quick-varianterna är kortare och snabbare än immediate-varianterna. ADDA (*add address*) och SUBA (*subtract address*) används vid pekararitmetik på adressregister. ADDX (*add with extend*) och SUBX (*subtract with extend*) är varianter som används vid addition (subtraktion) av tal större än 32 bitar.

Aritmetiska operationer måste kunna utföras på såväl tal med tecken som tal utan tecken. Addition och subtraktion utförs (binärt) lika oavsett man arbetar med naturliga tal (0,1,2 ...MAX) eller heltal (-MIN ...-1,0,1...MAX). Detta gäller dock inte division och multiplikation. Instruktions-uppsättningen innehåller därför instruktioner för division och multiplikation av naturliga tal (divu och mulu) och instruktioner för division och multiplikation av heltal

För division används DIVU (*unsigned divide*) för division av naturliga tal eller varianten DIVS (*signed divide*) för division av heltal. Motsvarande instruktioner för multiplikation är MULU (*unsigned multiply*) och MULS (*signed multiply*).

CLR-instruktionen (*clear an operand*) används för att nollställa en operand. För att teckenutvidga resultat från 16-bitars aritmetik till 32 bitar används instruktionen EXT (*sign extend*). Instruktionen NEG (*negate*) bestämmer tvåkomplementet av operanden, dvs byter tecken. NEGX (*negate with extend*) är en variant där X-biten används i tvåkomplementbildningen.

Jämförelseoperationen CMP (*compare*) jämför operander och påverkar flaggorna i **CCR**. Observera att dessa operationer inte modifierar destinationsoperanden. Varianten CMPI (*compare immediate*) används då källoperanden är en konstant. Varianten CMPA (*compare address*) används då destinationsoperanden är ett adressregister.

För att testa en operand används TST (*test an operand*). Resultatet av testen sätter flaggorna i **CCR**.

TAS-instruktionen (*test and set*) är en speciell testinstruktion som testar och samtidigt ettställer en bit i operanden. Instruktionen är avsedd för datorer där flera processorer delar samma primärminne (multiprocessorsystem).

Logiska operationer

AND (*and logical*), OR (*inclusive or logical*) och EOR (*exclusive or logical*) används för att utföra bitvis logiska operationer. varianterna ANDI (*and immediate*), ORI (*inclusive or immediate*) och EORI (*exclusive or immediate*) används då källoperanderna är konstanter. NOT-instruktionen (*logical complement*) bildar ett-komplementet av operanden, dvs *inverterar* bitmönstret hos operanden.

Logiska operatorer (AND, OR, EOR) ska kunna utföras på flera datatyper, det finns dock knappast behov av att utföra logiska operationer på en *adress*. Logiska operationer kan därför inte utföras på innehållet i ett adressregister.

Skiftoperationer

Tre typer av skift-instruktioner förekommer, *aritmetiskt skift*, *logiskt skift* och *rotation*. Skillnaden mellan dessa typer av skift ligger i hur C-flaggan respektive X-flaggan används vid skiften. Såväl vänsterskift som högerskift kan utföras. Utskiftade bitar placeras i C- respektive X-flaggorna. Aritmetiskt skift används då operanden är ett heltal. Studera även kapitel 2 som behandlar skift.

Vid ASR (*arithmetic shift right*) kopieras den mest signifikanta biten så att talets tecken bibehålls.

Vid ASL(*arithmetic shift left*) skiftas alltid en nolla in på den minst signifikanta positionen.

Vid logiska skift LSL (logical shift left) och LSR (*logical shift right*) är den inskiftade biten alltid noll.

Vid rotation ROL (*rotate left without extend*) respektive ROR (*rotate right without extend*) är den inskiftade biten samma som den utskiftade biten.

Slutligen, vid ROXL (*rotate left with extend*) och ROXR (*rotate right with extend*) ingår X-flaggan i skiftet. Denna skifttyp kallas ibland *carryskift*.

Bitoperationer

Bitoperationer används för att manipulera enskilda bitar i en operand. Operanden kan ha två storlekar: 32 bitar om destinationsoperanden är ett dataregister, 8 bitar annars.

`BTST` (*test a bit*) testar en bit och sätter Z-flaggan i **CCR** om biten är 0. Om däremot den testade biten är 1 nollställs Z-flaggan.

`BSET` (*test a bit and set*) utför först samma test som `BTST` men ettställer också den bit som testats.

`BCLR` (*test a bit and clear*) utför först samma test som `BTST` och nollställer därefter den testade biten.

`BCHG` (*test a bit and change*) utför först samma test som `BTST` och inverterar därefter den testade biten.

BCD-operationer

För tal kodade på BCD-form (*binary coded decimal*) (jämför kapitel 3) finns instruktionerna `ABCD` (*add decimal with extend*), `SBCD` (*subtract decimal with extend*) och `NBCD` (*negate decimal with extend*). I operationerna ingår alltid X-biten som carryflagga. Instruktionen `nbcd` kan användas för att bilda såväl 10-komplement som 9-komplement.

Villkorliga operationer

Villkorliga instruktioner används framför allt för att styra programflödet, exempelvis `Bcc` (*branch on condition*) och `DBcc` (*test, decrement and branch on condition*), men det finns också en speciell instruktion, `sCC` (*set according to condition*), med vilken man exempelvis kan styra tilldelning av variabler. Gemensamt för dessa villkorliga operationer är att flaggregistrets innehåll testas och att den fortsatta exekveringen beror på de olika flaggornas värden.

Ovillkorliga flödeskontroll operationer

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

Instruktioner för systemkontroll

Dessa instruktioner är speciella, ofta (men inte alltid) så kallade *priviligierade instruktioner*, vilket innebär att de endast utförs om processorn är i *supervisor mode*. Om ett försök att utföra en priviligierad instruktion utförs då processorn är i *user mode* kommer detta försök att avbrytas och speciell så kallad *exception processing* vidtas. Processorn hämtar en ny startadress från adress $20 i minnet (*privilege violation vector*) och börjar exekvera programmet på denna adress.

Instruktionen `RESET` (*reset peripherals*) aktiverar processorns reset-signal, och tvingar därmed alla periferienheter till ett starttillstånd. Processorns tillstånd påverkas ej i övrigt utan exekveringen fortsätter därefter med nästa instruktion.

`RTE` (*return from exception*) används för att avsluta *exception processing*.

`STOP` har en 16 bitars operand som placeras i **SR**, programpekaren **PC** pekar därefter på nästa instruktion men exekveringen forsätter inte förrän RESET-signalen aktiveras eller ett avbrott med samma eller högre prioritet än avbrottsmasken i **SR** uppträder. Instruktionen kan användas för att synkronisera exekveringen av program i olika MC68000-datorer.

`TRAP`-instruktionen har en operand, ett tal 0-15, talet anger en *trap exception vector*. Vid exekvering av `TRAP`-instruktionen inleds *exception processing* på adress som anges av vektor-numret, dvs operanden.

`TRAPV`-instruktionen inleder *exception processing* om V-biten i **CCR** är 1. Instruktionen medger ett enkelt sätt att registrera spill exempelvis vid multiplikation.

Med `CHK` kontrolleras att innehållet i ett dataregister finns i ett visst intervall. Om detta inte är fallet inleds *exception processing*.

Speciella move-instruktioner används för att läsa eller skriva i **SR**. Det finns också instruktioner som utför logiska operationer på innehåll i **SR** respektive **CCR**.

Övriga instruktioner

Instruktionen `NOP` (*no operation*) utför ingen operation. Instruktionen kan verka överflödig men finns av tradition med i instruktionsuppsättningen.

5.2.5 Adresseringssätt

MC68000 tillhandahåller 12 olika adresseringssätt (*adresseringsmoder*). Med adresseringssätt menas det sätt på vilket *effektiva adressen* (EA), bestäms. Med effektiva adressen menas adressen till data som ska användas av instruktionen. Var och en av MC68000:s adresseringsmoder kommer nu att beskrivas i detalj med exempel på hur effektiva adressen bestäms. Följande tabell sammanfattar de tillgängliga adresseringsmoderna:

	Adresseringsmod	**Syntax**	**Effektiv adress**
	Register direkt		
A	Data	Dn	EA=Dn
B	Adress	An	EA=An
	Register indirekt		
F	Adress	(An)	EA=(An)
G	Adress med postinkrement	(An)+	EA=(An); An=An+*size* (*byte,word,long*)
H	Adress med predecrement	-(An)	An=An-size; EA= (An)
I	Adressregister med offset	(d16,An)	EA=d16+(An)
J	Adressregister indirekt med index och 8-bitars offset	(d8,An,Xn)	EA=d8+(An)+(Xn)
K	Programräknare indirekt med offset	(d16,PC)	EA=d16+(PC)
L	Programräknare indirekt med index och 8 bitars offset	(d8,PC,Xn)	EA=d8+(PC)+Xn
E	Absolut adress kort	(xxx).W	EA=xxx (teckenutvidgat)
D	Absolut adress lång	(xxx).L	EA=xxx
C	Omedelbar	#<data>	EA=implicit i instruktionen

Anmärkning:

- med skrivsättet “d16+(An)” menas d16 (en konstant med max 16 bitar) adderas till innehållet i An.
- med ”Xn” avses ett data- *eller* adress-register
- Bokstäverna till vänster i tabellen använder du som söknyckel för att studera de olika adresseringssätten nedan.

A: *Dataregister direkt* Dn

Dettaa adresseringssätt innebär att operanden finns i ett dataregister. Dataregister anges som **D0**, **D1**, **D2** osv t.o.m **D7**. Instruktionen anges oftast med ett prefix som talar om hur många bitar av dataregistret som används, dvs den avsedda *datatypen*.

EXEMPEL

I instruktionen:

```
NOT.B   D0
```

utgörs effektiva adressen (EA) av register **D0**, och data finns i register **D0**.

5.16

Exempel 5.17 visar hur innehållet i **D0**,**D1** och **D2** påverkas av instruktionen NOT beroende på i vilken form den används.

EXEMPEL

Instruktion		EA
NOT.B D0 *komplementerar de* 8 minst signifikanta *bitarna i* ***D0***		EA: D0
	före	0011 1010 0101 1100 0000 1111 1111 0000
	efter	0011 1010 0101 1100 0000 1111 0000 1111
NOT.W D1 *komplementerar de* 16 minst signifikanta *bitarna i* ***D1***		EA: D1
	före	0011 1010 0101 1100 0000 1111 1111 0000
	efter	0011 1010 0101 1100 0000 1111 0000 1111
NOT.L D2 *komplementerar* samtliga *32 bitar i* ***D2***		EA: D2
	före	0011 1010 0101 1100 0000 1111 1111 0000
	efter	1100 0101 1010 0011 1111 0000 0000 1111

5.17

B: *Adressregister direkt* An

Detta adresseringssätt används då man avser innehållet i ett adressregister. Effektiva adressen, dvs adressen till data, för denna adresseringsmod är ett adressregister. Adressregister anges som **A0**, **A1**, **A2** osv t.o.m **A7**. I stället för **A7** kan **SP** användas. Innehållet i ett adressregister kan inte manipuleras på samma sätt som innehållet i ett dataregister. Eftersom adressregister är avsedda att innehålla minnesadresser, kan bara storleken *word* eller *long* användas.

I exempel 5.18 nedan används adresseringssättet dataregister direkt för källoperanden och adressregister direkt för destinationsoperanden.

EXEMPEL

`ADDA.W`	`D3,A0`	adderar de 16 minst signifikanta bitarna i D3 till register A0.
	EA för destinationsoperand är A0	
`ADDA.L`	`D4,A1`	adderar samtliga bitar i D4 till register A1.
	EA för destinationsoperand är A1	

5.18

C: *Omedelbar* `#<data>`

Omedelbar addressering (immediate) används för att ange en konstant som operand för instruktionen. I exemplet visas hur innehållen i **D0** respektive **A0** ändras av move-instruktioner. Båda exemplen utnyttjar omedelbar adressering för källoperanden. Det första exemplet utnyttjar dataregister direkt som destinationsoperand och det andra adressregister direkt. Notera också att instruktionsformen MOVEA måste användas då destinationsoperanden är ett adressregister.

EXEMPEL

	EA: D0	
före	1010 1010 1111 0000 1010 0011 0000 0000	**MOVE.B #$FE,D0**
efter	1010 1010 1111 0000 1010 0011 **1111 1110**	*Hexadecimala värdet FE placeras i **D0**:s minst signifikanta* byte.
	EA: A0	**MOVEA.L #$4000,A0**
före	0011 1010 0101 1100 0000 1111 1111 0000	
efter	**0000 0000 0000 0000 0100 0000 0000 0000**	*Hexadecimala värdet 4000 placeras i **A0.***

5.19

D: *Absolut adress lång* `(xxx).L`

Med adresseringssättet *Absolut adress lång* (eller bara *absolut long*) anges en 32-bitars adress som utgör EA i instruktionen. Den erhållna adressen är den effektiva adressen och anger alltså var data läses/skrives.

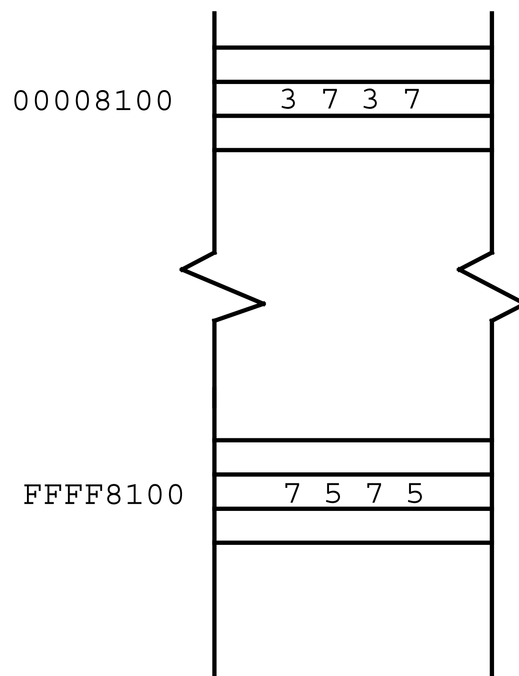
EXEMPEL

Antag att vi har minnesinnehåll enligt figur 5.7 och att instruktionen nedan utförs.

```
MOVE.W      ($FFFF8100).L,D1
```

Placerar hexadecimala värdet 7575 i **D1**. Adresseringssättet för källoperanden är *absolut lång* och **EA** är $FFFF8100. För destinationsoperanden utnyttjas *data register direkt*.

5.20



FIGUR 5.7 ILLUSTRATION AV MINNESINNEHÅLL FÖR EXEMPEL 5.20

E: *Absolut adress kort* `(xxx).W`

Vid adresseringsmoden *absolut adress kort* (eller bara *absolut kort*) anges en 16-bitars adress som utgör EA i instruktionen. Dessa 16 bitar teckenutvidgas till 32 vid adressbildningen.

Observera att den korta formen endast kan användas för att addressera två begränsade intervall, nämligen:

```
$00000000 - $00007FFF
```

respektive

```
$FFFF8000 - $FFFFFFFF
```

Den erhållna teckenutvidgade adressen är den effektiva adressen och anger alltså var data läses/skrives.

EXEMPEL

Antag att vi har minnesinnehåll enligt Figur 5.7. Instruktions-sekvensen:

```
MOVE.W      ($8100).W,D0
MOVE.W      ($8100).L,D1
```

Placerar hexadecimala värdet 7575 i **D0**, ty adressen teckenutvidgas innan data läses. Den andra instruktionen placerar hexadecimala värdet 3737 i **D1**, ty här ske ingen teckenutvidgning av adressen.

Adresseringsmoden för källoperanden i den första instruktionen är *absolut kort* och **EA** är $FFFF8100. Destinationsoperanden utnyttjar dataregister direkt.

Adresseringsmoden för källoperanden i den andra instruktionen är *absolut lång* och **EA** är $8100. Destinationsoperanden utnyttjar dataregister direkt.

5.21

F: *Adressregister indirekt* (An)

I *adressregister indirekt* finns effektiva adressen i ett adressregister. Dvs ett av registren **A0-A7** (**SP**) innehåller adressen till data. Längden (*byte*, *word* eller *long*) anges av instruktionen.

EXEMPEL

Figur 5.8 visar innehållet i ett fragment av minnet.
Nu utförs följande instruktioner:

```
MOVEA.L     #$4000,A0
MOVE.B      (A0),D0
MOVE.W      (A0),D1
MOVE.L      (A0),D2
```

Adress	Innehåll
00003FF8	
00004000	5465
00004002	3243

FIGUR 5.8

I **D2** finns efter instruktionsföljden: 54 65 32 43
I **D1** finns efter instruktionsföljden: XX XX 54 65
I **D0** finns efter instruktionsföljden: XX XX XX 54
där XX anger att innehållet i dessa bitar är opåverkade av instruktionen, dvs de behåller sitt gamla värde.

5.22

För de tre sista instruktionerna gäller att adresseringssättet för källoperanden är adressregister indirekt. EA är i alla tre fall innehållet i **A0** vilket är $4000 (som ges av den första instruktionen). Destinationsoperanden utnyttjar dataregister direkt.

G: *Adressregister indirekt med postautoincrement* (An)+

Detta adresseringssätt liknar det föregående men i detta fall ökas innehållet i adressregistret efter (*postautoincrement*) att data läses (skrives). Ökningen beror av instruktionen, för *byte* ökas 1, för *word* ökas 2 och för *long* är ökningen 4.

EXEMPEL

Med minnesinnehåll enligt Figur 5.8 (se exempel 5.22) och instruktionssekvensen:

```
MOVEA.L     #$4000,A0
MOVE.B      (A0)+,D0
```

blir det nya innehållet i **D0**: `XX XX XX 54` då innehållet i **A0** först används som EA och sedan ökas med ett.
A0 innehåller efter instruktionen `00 00 40 01`

Hade vi i stället haft instruktionen:

```
MOVE.W      (A0)+D0
```

hade nya innehållet i **D0** blivit `XX XX 54 65` då innehållet i **A0** först används som EA och sedan ökas med två.
A0 innehåller efter instruktionen `00 00 40 02`

Adresseringssättet för källoperanden är i båda *fallen adressregister indirekt med postautoincrement*. EA är i båda fallen innehållet i register **A0** vilket är $4000. Destinationsoperanden utnyttjar *dataregister direkt* i båda fallen.

5.23

H: *Adressregister indirekt med preautodecrement* -(An)

Detta adresseringssätt liknar det föregående men i detta fall minskas först (*preautodecrement*) innehållet i adressregistret före data läses (skrives). Minskningen beror av instruktionen, för *byte* minskas 1, för *word* minskas 2 och för *long* är minskningen 4.

EXEMPEL

Med minnesinnehåll enligt Figur 5.8 och instruktionssekvensen:

```
MOVEA.L     #$4002,A0
MOVE.B      -(A0),D0
```

blir det nya innehållet i **D0**: `XX XX XX 65` då **A0** först minskas med ett.
A0 innehåller efter instruktionen `00 00 40 01`

Hade vi i stället haft instruktionen:

```
        MOVE.W        -(A0),D0
```

hade nya innehållet i **D0** blivit `XX XX 54 65` då **A0** först minskas med två. **A0** innehåller efter instruktionen `00 00 40 00`

Adresseringssättet för källoperanden är i båda fallen *adressregister indirekt med preautodecrement*. EA är i båda fallen innehållet i register **A0.** I det första fallet är EA $4001 och i det andra fallet $4000. Destinationsoperanden utnyttjar *dataregister direkt* i båda fallen.

5.23

Auto decrement/increment är användbara då datastrukturer som vektorer ska genomsökas. Dessutom används de för att implementera stackar i minnet. Eftersom ett godtyckligt adressregister kan användas med dessa adresseringssätt kan man i princip använda vilket adressregister som helst för implementering av en datastack. Observera dock att register **A7** (**SP**) har en speciell funktion som stackpekare vid exempelvis subrutinanrop. Denna funktion är låst av hårdvaran och kan inte ändras. Man kan följaktligen inte *ersätta* **A7** med ett annat adressregister, däremot implementera ytterligare en stackfunktion.

I: *Adressregister indirekt med konstant offset* `(d16,An)`

Med detta adresseringssätt bestäms den effektiva adressen genom att innehållet i något adressregister, **A0-A7** (eller **SP**) adderas till en konstant offset. Offseten, betecknad *d16,* kan vara högst 16 bitar men teckenutvidgas till 32 bitar innan den adderas till adressregistrets innehåll vilket innebär att det adressintevall som kan nås relativt detta innehåll blir

$$-32768 \leq \textit{innehåll i adressregister} \leq 32767$$

EXEMPEL

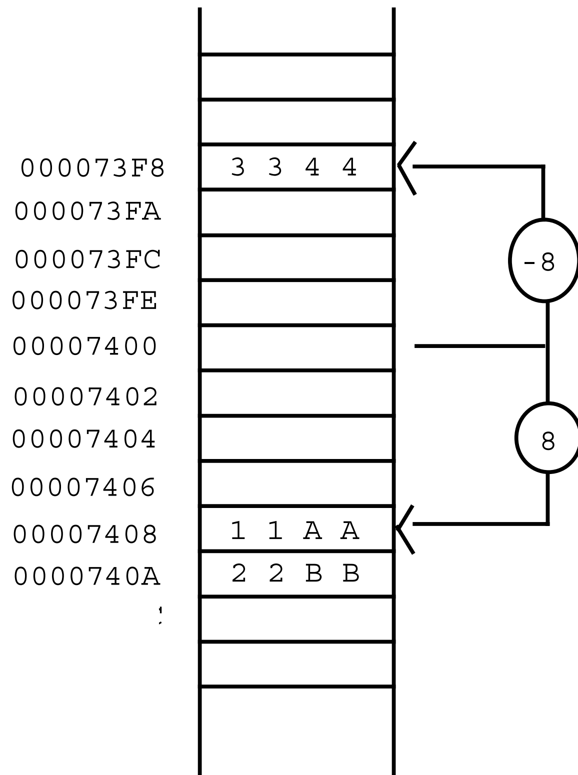
Med minnesinnehåll enligt figur 5.9 nedan kommer instruktionsföljden

```
        MOVEA.L       #$7400,A0
        MOVE.L        (8,A0),D0
        MOVE.W        (-8,A0),D1
```

att ge nya registerinnehåll enligt:

D0: 11 AA 22 BB
D1: XX XX 33 44

5.24



FIGUR 5.9 ILLUSTRATION AV MINNESINNEHÅLL TILL EXEMPEL 5.24

J: *Adressregister indirekt med konstant offset och offset i register* `(d8,An,Xn)`

Detta adresseringssätt tillåter såväl konstant offset (*displacement*) om 8 bitar som en variabel offset angiven av ett register. Den variabla offseten finns i det så kallade indexregistret, betecknat Xn. Indexregistret kan vara såväl adress som dataregister. Indexregistret kan (men behöver inte) anges med storlek.

EXEMPEL

Om hela innehållet i indexregistret ska användas för EA-beräkningen ska detta anges.

```
(d8,An,D0.L)
```

anger index som D0 med storlek *long*

Om endast 16 bitar av indexregistrets innehåll används kommer detta att teckenutdvidgas före EA-beräkningen.

```
(d8,An,D2.W)
```

anger index som D2 med storlek *word*

Om storlek för indexregistret utelämnas, tolkas detta som Dn.W

```
(d8,An,D1)
```

anger index som D1 med storlek *word*

5.25

EXEMPEL

Antag att följande instruktionerna utförts:

```
MOVEA.L     #$7400,A0
MOVE.L      #$10,D0
```

Bestäm effektiva adressen (EA) för källoperanden i instruktionen:

```
MOVE.L      (4,A0,D0.W),D1
```

Lösning:

Adressregister **A0**:		$7400
d8:		$ 4
Indexregister **D0**:	+	$ 10
Effektiva adressen:		$7414

5.26

Om indexregistret angetts med storlek *long* används innehållet i registret direkt i beräkningen av effektiv adress. Om däremot storleken *word* angetts, tas de 16 minst signifikanta bitarna i registret, teckenutvidgas till 32 bitar och används därefter i beräkningen av effektiv adress. Det samma gäller om ett adressregister används som indexregister. Den konstanta offseten *d8* teckenutvidgas också vid adressberäkningen. Detta innebär att:

$$-129<d8<128$$

EXEMPEL

Antag att följande instruktionerna utförts:

```
MOVEA.L     #$7400,A0
MOVE.L      #$8001,D0
```

Bestäm effektiva adressen för destinationsoperanden i:

```
MOVE.B      D1,($7F,A0,D0.W)
```

Lösning:

Adressregister **A0**:	$00007400
d8:	$0000007F
D0, teckenutvidgas eftersom .W angetts:	$FFFF8001
Effektiva adressen	$FFFFF480

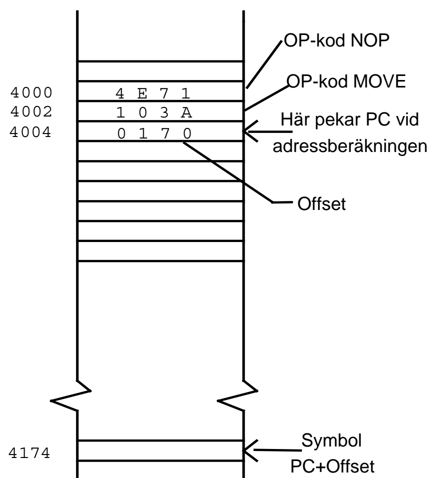
5.27

K: *Programräknarrelativ indirekt med konstant offset*

(d16,PC)

Detta adresseringssätt liknar mycket *adressregister indirekt med konstant offset*. I detta fall används dock programräknaren **PC** i stället för ett generellt adressregister som basadress för beräkningen av effektiv adress. För adressberäkningen används adressen till instruktionens operationskod +2. Figur 5.10 visar ett exempel på hur adressberäkningen utförs.

EXEMPEL



FIGUR 5.10

För exemplet antar vi att följande program placerats i minnet

```
       ORG          $4000
       NOP
       MOVE.B       (Symbol,PC),D0
       ...
       ...
       ORG          $4174
Symbol:DS.B         1
```

Beräkningen av effektiv adress går till på följande sätt:

EA = PC + offset

dvs: Offseten som anges av innehållet i det ord som följer instruktionen (i exemplet 0170) adderas till *adressen* till det ord som följer instruktionens operationskod (i exemplet 4004). För exemplet får vi då 4174 som effektiv adress. Se även figur 5.10

5.28

Observera att *offset* bestäms vid assemblering av programmet. Det finns alltså ingen information om *absoluta* adresser kodade i instruktionen. Detta innebär att det assemblerade programmet, dvs maskinkoden, inte längre är beroende av *var* det placeras i primärminnet. Trots att vi skrivit ORG $4000 i början av programmet, kan det alltså placeras på godtycklig (jämn) adress i primärminnet. Förutsättningen är dock att förhållandet mellan den refererade symbolen och instruktionen som refererar symbolen bibehålls vid en sådan flyttning. Adresseringssättet sägs därför stödja konstruktion av så kallad *positionsoberoende kod* (*Position Independent Code, **PIC***).

Positionsoberoende kod underlättar i hög grad *relokering* av ett program i minnet. Relokering innebär att kod flyttas från en position i minnet till en annan position. Vid relokering måste alla absoluta adresser räknas om.

L: ***Programräknarrelativ indirekt med konstant offset och offset i register*** **`(d8,PC,Xn)`**

Addresseringssättet liknar *adressregister indirekt med konstant offset och offset i register* med skillnaden att **PC** används som basadress i stället för ett adressregister. Beräkningen av Effektiv adress utförs alltså, precis som i föregående exempel med PC som basadress. Vid adressberäkningen innehåller PC instruktionens opkods adress +2. Offseten teckenutvidgas från 8 till 32 bitar. Om Indexregistret anges med storleken *word* teckenutvidgas även detta innehåll innan det adderas till basadressen.

5.3 Programmering i assembler

5.3.1 Inledning

Assemblerprogrammering skiljer sig på flera sätt ifrån programmering i exempelvis något högnivåspråk. Assemblerprogrammeraren måste ha en ingående kunskap om den maskin (processor) som ska programmeras vilket i sig innebär en kunskap om datorteknik i allmänhet som en "vanlig" programmerare knappast behöver besitta. Dessutom bör assembler-programmeraren behärska andra typer av utvecklingssystem. Det är exempelvis vanligt med så kallade kors-utvecklingssystem vilket innebär att programvara för en typ av maskin (målsystemet), utvecklas och simuleras på en helt annan typ av maskin (värdsystemet).

Assemblerprogrammerarens kanske viktigaste verktyg är *assemblatorn*. Assemblatorn är ett program (i värdsystemet) som läser en *källtextfil* med instruktioner (för målsystemet), översätter dessa till maskinkod (för målsystemet) på ett format som gör det möjligt att överföra denna maskinkod från värdsystem till målsystem (*laddfil*). Då maskinkoden överförts till måldatorn kan den testas med hjälp av någon, i målsystemet, inbyggd *monitor/debugger*.

Arbetsgången vid utveckling av assemblerprogram liknar utveckling av program i högnivåspråk. En stor skillnad ligger dock i *testfasen*. Ett program skrivet i högnivåspråk testas vanligtvis genom att man utför (*exekverar*) programmet ett antal gånger, kanske varierar indata och analyserar resultatet, dvs utdata, från programmet. Att tillämpa samma testmetod på ett nyskrivet assemblerprogram är vanligtvis både tidsödande och dumt. Ett assemblerprogram testas, steg för steg, med instruktionsvis exekvering. Resultatet av varje instruktion kontrolleras och jämförs med den avsikt man hade då man skrev instruktionen.

Programmet organiseras i *subrutiner* (funktioner) som noggrannt testas. Efterhand byggs fler lager med subrutiner och varje ny subrutin testas.

Tumregler då du programmerar i assembler ...

- *Modularisera programmen*, dvs dela in det i lämpliga subrutiner, långa assemblerprogram blir snabbt oöverskådliga. Skriv aldrig enstaka subrutiner vars källtext sträcker sig över flera A4-sidor.
- *Testa noggrannt*, små subrutiner är enklast att testa. Försök också "utföra" subrutinen med papper och penna innan du testar den i måldatorn. Försök förutse indata och se vilka utdata som genereras. Speciellt viktigt brukar det vara att kontrollera så kallade specialfall, dvs sådana fall då oväntade indata skickas till subrutinen.
- *Undvik trickprogrammering*, assemblerprogrammering ger möjlighet att skriva snabb och mycket effektiv kod. Detta får emellertid aldrig ge upphov till svårförståeliga programkonstruktioner eller så kallade spagetti-program. Ett sådant program kan visa sig helt oanvändbart då komplexiteten i programvaran växer. Skriv klart, enkelt och strukturerat.
- *Kommentera väl*, assemblerprogram måste kunna underhållas, dvs eventuella fel rättas, programmet måste också kunna vidareutvecklas. Ofta sker detta av olika personer och en annan programmerare måste snabbt kunna sätta sig in i hur koden är avsedd att fungera. Försök därför hålla dina kommentarer korta och kärnfulla. Skriv inga självklarheter utan belys funktionaliteten.

5.3.2 Specifikation och formell beskrivning

Då du konstruerar ett assemblerprogram utgår du i från en specifikation, dvs en beskrivning av vad programmet skall göra. Denna specifikation är ofta formulerad i meningar på ett sätt som gör att du, som människa, kan förstå och föreställa dig vad programmet bör uträtta (*funktionalitet*). Din uppgift är nu att med hjälp av dina utvecklingsverktyg översätta denna beskrivning till ett språk (*maskinkod*), och därefter kontrollera att denna översättning är korrekt, dvs testa programmet och bekräfta att det svarar mot den ursprungliga specifikationen.

I bland kan det hända att du upptäcker att specifikationen är ofullständig, dvs det uppkommer situationer (ofta specialfall) som den ursprungliga specifikationen inte behandlar. Gå då tillbaks till denna och se om du kan, i första hand *komplettera*, i andra hand *ändra* specifikationen. Ett första steg i översättningsprocessen är att *formalisera* specifikationen.

Exempelvis kan man inleda med att skriva ett programförslag i något högnivåspråk. Detta programförslag behöver inte vara ett fullständigt program, utan används bara som mellansteg. Formaliserandet innebär ett viktigt steg när det resulterar i konstruktioner som kan översättas till sekvenser av assemblerinstruktioner.

Vi väljer en uppgift där vi skall översätta stora bokstäver (VERSALER) till små (gemena).

EXEMPEL

Formell beskrivning

Specifikation: Skriv en programsekvens som konverterar alla stora bokstäver (versaler) till små bokstäver (gemena) i en mening.

Formalisering: Vi konstaterar att en mening kan bestå av flera tecken. Antalet tecken är okänt, men vi vet att en mening avslutas med punkt. Alltså har vi start- och slut-villkor.

```
Från( meningens första tecken) till (tecken är "punkt)
        För varje tecken:
                Om( tecken är Versal)
                        Konvertera tecken till gemen;
        nästa tecken;
```

Vi kan gå längre mot en formell beskrivning, exempelvis enligt följande:

```
int     index;
char    tecken;
char    mening[128];

        index = 0;
        tecken = mening[ index ];                   /* från första */
        while( tecken != '.'){                      /* till "punkt" */
                if ( isupper( tecken) )             /* om Versal */
                        tecken = tolower(tecken);   /* konvertera*/
                mening[index] = tecken;             /* ersätt */
                index = index + 1;
                tecken = mening[index];             /* nästa */
        }
```

Vi har i själva verket här formaliserat specifikationen i form av ett C-program. I programmet har vi antagit att textsträngen som ska konverteras tidigare placerats i strängvariabeln `mening`.

5.29

Det enkla exemplet visar hur en specifikation kan formaliseras, i detta fall genom att använda programspråket 'C', men programspråket i sig är oftast mindre intressant. Det viktiga är att vi formaliserar med hjälp av konstruktioner av den typ som ofta finns i högnivåspråk. I exemplet använder vi en *heltalsvariabel* (`index`), en *strängvariabel* (`mening`), språkkonstruktionerna `while` och `if`, dessutom tog vi tillfället i akt att *modularisera* genom att använda färdiga funktioner `isupper` och `tolower`.

Låt oss nu gå vidare och se hur vi översätter deklarationer och programkonstruktioner givna i programspråket ’C’ till motsvarande assemblerkod.

5.3.3 Tilldelningar

```
index = 0;
tecken = mening[ index ];                 /* från första */
while( tecken != '.'){                    /* till "punkt" */
    if ( isupper( tecken) )               /* om Versal */
        tecken = tolower(tecken);         /* konvertera*/
    mening[index] = tecken;               /* ersätt */
    index = index + 1;
    tecken = mening[index];               /* nästa */
}
```

Tilldelningar utgör en enkel konstruktion i ett högnivåspråk. I assembler motsvaras detta ofta av enkla move-instruktioner. Studera programmet i marginalen. Den globala variabeln `index`, tilldelas värdet 0.

Först måste vaiabeln `index` placeras på någon plats i minnet. Den måste också kunna hålla 4 bytes (storleken av en `int`). Det är lämpligt att placera kod och data på olika ställen i minnet. Detta åstadkommes med hjälp av assemblerdirektivet "org". Vi förutsätter här att vi kan placera data på adress \$5000 och uppåt, och att koden ska placeras med start på adress \$4000. Vi får då följande:

```
           ORG         $5000
index:     DS.L        1
```

dvs: Placera variabeln som kallas `index`, med start på adress \$5000 i minnet, reservera 4 bytes (*define storage long*).

Längre fram i programmet skriver vi koden för själva tilldelningen. Vi placerar koden med början på adress \$4000:

```
           ORG         $4000
start:     MOVE.L      #0,(index).L
```

Om `index` skall användas, som i vårt exempel, som en *räknare* och *dessutom bara som en temporär variabel*, kan man i stället upplåta (allokera) ett dataregister. Det är då inte nödvändigt att allokera något extra minne för variabeln. Vi får då *i stället*:

```
           ORG         $4000
* registeranvändning:
* register D0 = variabel ‘index’
start:     MOVE.L      #0,D0       variabel
```

Vi såg också exempel på en strängvariabel (`mening`) deklarerad enligt:

```
char mening[128];
```

Deklarationen översätts enkelt till assemblerkod enligt:

```
mening:    DS.B       128
```

dvs reservera 128 *bytes* i minnet för en textsträng, och kalla startadressen för denna textsträng `mening`.

Då vi vill behandla tecknen i textsträngen gör vi detta genom att använda variabeln `index` som offset till början av strängen. Vi har reserverat register **D0** för `index` och använder adressregister **A0** för startadressen till `mening`. De nödvändiga initieringarna kan då utföras enligt:

```
          ORG        $4000
*  registeranvändning:
*  register D0 = variabel 'index'
*             A0 = startadress till 'mening'

      MOVE.L      #0,D0
      MOVEA.L     #mening,A0
```

Det är också lämpligt att upplåta ett register för variabeln `tecken`, säg register **D1.** Tilldelningen

```
tecken = mening[index];
```

kan då exempelvis utföras enligt:

```
      MOVE.B      (0,A0,D0),D1
```

```
index = 0;
tecken = mening[ index ];             /* från första */
while( tecken != '.'){                /* till "punkt" */
    if ( isupper( tecken) )           /* om Versal */
        tecken = tolower(tecken);     /* konvertera*/
    mening[index] = tecken;           /* ersätt */
    index = index + 1;
    tecken = mening[index];           /* nästa */
}
```

Observera att strängvariabeln innehåller 8-bitars tecken vilket gör det lämpligt att bara utnyttja de 8 minst signifikanta bitarna i register **D1**. Detta sätt att adressera strängen innebär att innehållet i **A0** ej får ändras medan innehållet i **D0** ska räknas upp med 1 för varje varv i programslingan `while`, `(index = index+1);`.

Samma funktion, dvs *tecken för tecken* i textsträngen kan vi få på ett annat sätt genom att använda *postautoincrement mode*:

```
MOVE.B      (A0)+,D1
```

dvs pekaren **A0** räknas upp med ett, (ty storleken är byte) av instruktionen.

Under nästa varv kommer **A0** att peka på nästa tecken i strängen, osv. Programmet kan då i bland göras kortare. För vårt exempel använder vi dock inte denna konstruktion.

Låt oss nu genomföra ytterligare ett steg i översättningsprocessen enligt det inledande exemplet. Vi kan här tillåta oss att blanda delar av den formella beskrivningen med sekvenser av assemblerinstruktioner.

Vi sammanfattar förutsättningarna för fortsättningen:

- Register D0 används för `index`
- Register D1 används för `tecken`
- Register A0 används som pekare till textsträngens första tecken (`mening`)

Nästa steg av vår översättning skulle då kunna skrivas:

```
/* deklarationer */
mening            DS.B        128

/* program */
MOVE.L      #0,D0                      initieringar ..
MOVEA.L     #mening,A0
MOVE.B      (A0,D0),D1                 första tecknet i strängen
while( D1 != '.'){
      if( isupper(D1) )
            D1=tolower(D1);
      MOVE.B      D1,(A0,D0)           /* ersätt */
      D0 = D0+1;                       /*  index = index + 1; */
      MOVE.B      (A0,D0),D1           /* nästa */
}
```

För den fortsatta översättningen behöver vi använda såväl *operatorer* som *jämförelser* och *subrutinanrop*. Detta behandlas generellt i de kommande avsnitten. Vi återkommer dock, med jämna mellanrum till översättningen av detta program.

5.3.4 Operatorer

MC68000 tillhandahåller en rad instruktioner för operationer på heltal. De kanske vanligaste operationerna utgörs av aritmetik och omfattar två operander.

En aritmetisk operation kan också utföras på *en* operand. Exempelvis definieras operationen **-A** som **0-A** och operationen +**A** som **0**+**A**. Eftersom nollan är underförstådd i båda fallen utesluter vi den och har alltså bara en operand. Eftersom +**A**=**A** utesluter vi också +operatorn i detta fall. Operatorer som opererar på en operand kallas *unära operatorer*.

Operationerna:

A + B

A - B

A * B

A / B

representerar de fyra räknesätten.

*Operatorerna +, - , * och / kallas* binära *eftersom de används med* två *operander (**A** och **B**).*

5.3.5 MC68000 flaggsättning vid aritmetiska operationer

Praktiskt taget alla MC68000 instruktioner för aritmetik påverkar flaggorna i **CCR**. Undantag från detta utgörs av instruktioner där ett adressregister utgör destinationsoperand. Flaggsättningen är en viktig del av instruktionsexekveringen. Speciellt viktigt blir det naturligtvis då programflöden ska bestämmas med utgångspunkt från resultat som räknats fram under tidigare exekvering. Flaggsättningen följer nästan undantagslöst de regler som redovisats i kapitel 3 och kan sammanfattas enligt följande:

N-flaggan sätts alltid till samma värde som den mest signifikanta biten i resultatet. Om operandens storlek är *byte*, kopieras bit 7 till **N**-flaggan. Om operandens storlek är *word*, kopieras bit 15 till **N**-flaggan. Om operandens storlek är *long*, kopieras bit 31 till **N**-flaggan. **N**-flaggan är alltså *bara intressant om vi betraktar våra ingående operander som tal med tecken* (tvåkomplement-form).

Z-flaggan sätts till 1 om resultat av operation är 0. Om operanderna har storleken *byte*, innebär detta att de 8 minst signifikanta bitarna i resultatet ska vara 0 för att **Z**-flaggan ska sättas till 1. För operander med storleken *word*, ska på motsvarande sätt, de 16 minst signifikanta bitarna i resultatet vara noll för att **Z**-flaggan ska sättas till 1. Slutligen, för operander med storleken *long*, måste alla 32 bitar i resultatet vara noll för att **Z**-flaggan ska sättas till 1. I alla övriga fall kommer **Z**-flaggan att sättas till 0.

V-flaggan används som indikator på att *tvåkomplementsspill* har uppstått (overflow). **V**-flaggan är alltså, precis som **N**-flaggan av intresse bara om vi betraktar operanderna som tvåkomplements-tal. **V**-flaggan sätts vid addition och subtraktion om:

*Ingående operanders tecken är lika **OCH***

Resultatets tecken skiljer sig från operandernas

För multiplikation och division gäller andra regler.

C-flaggan används som indikator på att ett beräknat resultat inte ryms i destinationsoperanden. Denna innebär därför en *spill-indikator* för addition av tal utan tecken. På motsvarande sätt kan **C**-flaggan betraktas som *lånesiffra* till den mest signifikanta positionen vid subtraktion av tal utan tecken.

X-flaggan (extend) kan ses som en extra **C**-flagga. **X**-flaggan sätts på samma sätt som **C** vid aritmetiska operationer men påverkas i allmänhet inte av andra typer av instruktioner. Användningen av **X**-flaggan kan vid en första betraktelse verka diffus, men det visar sig att i vissa speciella fall kan en mycket kompakt och effektiv kodning åstadkommas genom att man betraktar **X**-flaggan i stället för **C**-flaggan.

EXEMPEL

Redogör för flaggsättningen hos MC68000 då följande instruktioner utförs:

```
MOVE.W      #$7000,D0
ADDI.W      #$9000,D0
```

Lösning: `MOVE`-instruktionen påverkar flaggorna enligt följande (se instruktionslistan)

N=1 om data är negativt, annars 0
Z=1 om data är noll, annars 0
V=Nollställs alltid
C=Nollställs alltid
X=Påverkas ej av instruktionen

Eftersom data ($7000) är positivt och skilt från noll kommer flaggbitarna att ha följande värde efter `MOVE`-instruktionen: **N=0, Z=0, V=0, C=0, X="förra värdet".** Register **D0** innehåller nu: `xxxx xxxx xxxx xxxx 0111 0000 0000 0000` där "x" anger tidigare värde. Nu utförs `ADDI`-instruktionen, dvs:

```
           minnessiffror       1  1 1 1 0  0 0 0 0  0 0 0 0  0 0 0
 x x x x  x x x x  x x x x  x x x x  0 1 1 1  0 0 0 0  0 0 0 0  0 0 0 0
+x x x x  x x x x  x x x x  x x x x  1 0 0 1  0 0 0 0  0 0 0 0  0 0 0 0
=x x x x  x x x x  x x x x  x x x x  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0
```

Observera att de 16 mest signifikanta bitarna hela tiden är opåverkade. Eftersom resultatet ($ 0001 0000) ej kan representeras med 16 bitar, genererar additionen av operandernas bit 15 en Carry i stället för att påverka bit 16. Vi kan nu redogöra för flaggsättning efter `ADDI`:
Z-flaggan sätts till 1, ty de 16 minst signifikanta bitarna är noll.
N-flaggan sätts till 0, ty bit 15 i resultatet är noll
C-flaggan sätts till 1, ty Carry genererades av operationen
X-flaggan sätts på samma sätt som C-flaggan
V-flaggan sätts till 0, ty villkoret för tvåkomplementsspill uppfylldes ej.

5.30

Det finns flera saker att tänka på då det gäller flaggsättningen. De viktigaste är:

- *Operandernas storlek*, tänk på att såväl 8- ,16- som 32-bitars aritmetik kan utföras beroende på om operationen angetts som *byte*, *word* eller *long*.
- *Tal med eller utan tecken*, det finns inget sätt för MC68000 att avgöra om Du betraktar dina operander som tal med eller utan tecken. Av denna anledning tillhandahålls såväl V- som C-flagga. Du måste tänka på detta då Du väljer villkorliga instruktioner, (behandlas senare), eftersom dessa utför tester för *antingen* tal med tecken *eller* tal utan tecken..

Man kan alltid analysera flaggsättningen för en given instruktionssekvens om operanderna är kända. Olika instruktioner kan dock påverka flaggorna i **CCR** på olika sätt. Information om flaggsättning hittar du i MC68000:s instruktionslista.

5.3.6 Aritmetiska operatorer

I följande avsnitt kommer MC68000:s instruktioner för aritmetik att behandlas. Olika instruktionsformer presenteras och användningen exemplifieras.

Instruktioner för addition

Det finns inte mindre än sex olika instruktionsformer att välja bland då två operander ska adderas:

`ADD`	binär addition, grundform
`ADDA`	addition till adressregister
`ADDQ`	addition med liten konstant (1-8)
`ADDI`	addition med konstant, godtycklig storlek
`ADDX`	addition där X-flaggan ingår som carry
`ABCD`	addition av tal på 10-komplementsform.

Vilken form som *ska* användas bestäms av vilka operander som ingår. Om en källoperand är en liten konstant (större än 0 men mindre än 9) bör instruktionsformen `ADDQ` användas. I övrigt gäller för de fall där destinationsoperanden är ett adressregister att instruktionsformen `ADDA` *måste* användas. Om en källoperand är *en konstant* och destinationsoperanden *inte* är ett adressregister, *måste* någon av formerna `ADDI` eller `ADDQ` användas.

Vid instruktionen `ADDX` används X-flaggan i stället för C-flaggan vid additionen. Detta gör `ADDX`-instruktionen användbar i de fall där

programkoden måste skrivas så att C-flaggan påverkas av andra instruktioner emellan additionerna.

Instruktionen ABCD används speciellt då man arbetar med BCD-tal (se kapitel 2, *Datoraritmetik*).

Den generella additionsformen add, används i alla övriga fall. Operandernas storlek kan vara *byte* (ADD.B), *word* (ADD.W) eller *long* (ADD.L). Ett flertal adresseringsmoder kan användas, dock måste alltid minst en operand vara ett dataregister.

EXEMPEL

Användning av instruktioner för addition

```
        ADDQ.L      #1,A2
```

adderar 1 till innehåll i **A2.** Formen "quick" kan användas då konstanten är i intervallet 1-8.

Konstanten 10 ska adderas till innehållet i ett *adressregister*. I detta fall måste formen ADDA användas, dvs:

```
        ADDA.L      #10,An
```

(där n anger registernummer)

```
        ADDQ.L      #4,D6
```

adderar 4 till innehåll i **D6**

Innehållen i D0 och D1 ska adderas, resultatet ska placeras i D1:

```
        ADD.B       D0,D1               D1+D0→D1
```

Observera att instruktionen använder (och påverkar) bara de 8 minst signifikanta bitarna av dataregistret. Flaggsättningen baseras på 8-bitars aritmetik, dvs

N = D1 bit 7
C = 1 om resultatet ej ryms med 8 bitar, 0 annars
Z = 1 om de 8 minst signifikanta bitarna i resultatet är 0, 0 annars.
X = C
V = 1 om 8-bitars 2-komplementsspill, 0 annars

5.31

Instruktioner för subtraktion

Precis som för addition finns det sex olika instruktionsformer att välja bland då två operander ska subtraheras:

`SUB`	binär subtraktion, grundform
`SUBA`	subtraktion till adressregister
`SUBQ`	subtraktion med liten konstant (1-8)
`SUBI`	subtraktion med konstant, godtycklig storlek
`SUBX`	subtraktion där X-flaggan ingår som carry
`SBCD`	subtraktion av tal på 10-komplementsform.

Användning av subtraktionsinstruktioner är helt analog med additions-instruktionerna. Vilken form som *ska* användas bestäms av vilka operander som ingår. Om en källoperand är en liten konstant (större än 0 men mindre än 9) bör instruktionsformen `SUBQ` användas.

I övrigt gäller för de fall där destinationsoperanden är ett adressregister att instruktionsformen `SUBA` *måste* användas. Om en källoperand är *en konstant* och destinationsoperanden *inte* är ett adressregister, *måste* någon av formerna `SUBI` eller `SUBQ` användas. Vid instruktionen `SUBX` används X-flaggan i stället för C-flaggan vid additionen. Detta gör `SUBX`-instruktionen användbar i de fall där programkoden måste skrivas så att C-flaggan påverkas av andra instruktioner emellan additionerna.

Instruktionen `SBCD` används speciellt då man arbetar med BCD-tal (se kapitel 3, *Datoraritmetik*).

EXEMPEL

En konstant ska subtraheras från innehållet i ett adressregister. Om konstanten är mindre än 9 kan formen:

```
        SUBQ.L      #konstant,An
```

användas. Om konstanten är större än 8 ska formen

```
        SUBA.L      #konstant,An
```

användas. ’n’ anger registernummer.

En konstant ska subtraheras från innehållet i ett dataregister. Om konstanten är mindre än 9 kan formen:

```
        SUBQ.L      #konstant,Dn
```

användas. Om konstanten är större än 8 ska formen

```
        SUBI.L      #konstant,An
```

användas. ’n’ anger registernummer.

Den generella additionsformen `SUB`, används i alla övriga fall. Operandernas storlek kan vara *byte* (`SUB.B`), *word* (`SUB.W`) eller *long* (`SUB.L`). Ett flertal adresseringsmoder kan användas, dock måste alltid minst en operand vara ett dataregister.

Följande illustrerar fallet då innehållet i D0 ska subtraheras från innehållet i D1. Resultatet placeras i D1:

```
SUB.B        D0,D1
```
D1-D0→D1

Instruktionen använder (och påverkar) bara de 8 minst signifikanta bitarna. Flaggsättningen baseras på 8-bitars aritmetik, dvs

N = D1 bit 7
C = 1 om lånesiffra genereras, 0 annars
Z = 1 om de 8 minst signifikanta bitarna i resultatet är 0, 0 annars.
X = C
V = 1 om 8-bitars 2-komplementsspill, 0 annars

5.32

Instruktioner för multiplikation

Eftersom multiplikation utförs på olika sätt beroende på om operanderna betraktas som tal med eller utan tecken, finns det två olika multiplikations-instruktioner:

`MULU` multiplikation av tal utan tecken
`MULS` multiplikation av tal med tecken

EXEMPEL

Antag **D0**=$xxxx 9333 och instruktionen

```
MULU    #2,D0
```

utförs. Resultatet i **D0** efter operationen blir då $0001 2666

5.33

Till skillnad från exempelvis addition och subtraktion kan multiplikation endast utföras på operander med storleken *word*. Operationen utförs alltså alltid på två 16-bitars operander. Observera dock att resultatet av denna operation kan bli större än 16-bitar. Instruktionernas destinationsoperand är alltid ett dataregister, och resultatet av operationen påverkar samtliga 32-bitar i detta register (trots att operandernas storlek är *word*).

EXEMPEL

Koda C-programsekvensen:

```
a = b + (c * d);
```

i MC68000 assembler. a,b,c och d är globalt deklarerade enligt:

```
unsigned short a,b,c,d;
```

hänsyn till spill behöver ej tas, register D7 får användas för mellanlagring.

Lösning:

Variabler av typ `unsigned short` (16-bitar) innebär att *word*-format ska användas. Observera att parentesen måste evalueras först:

```
MOVE.W    (d).L,D7    variabel d, nu i D7
MULU      (c).L,D7    (c*d) nu i D7
ADD.W     (b).L,D7    b+(c*d) nu i D7
MOVE.W    D7,(a).L    resultatet nu i  a
```

5.34

Instruktioner för division

Divisionsinstruktionerna `DIVU` och `DIVS` utför division, såväl resultatets heltalsdel som restdelen placeras i destinationsoperanden. Allmänt kan vi skriva divisionen:

a/b = A B/b

där vi kallar:

a för täljare (divisor)

b för nämnare (dividend)

A för heltalsdel (integer part)

B för rest (remainder)

För instruktionerna gäller att operand **a** är 32-bitar, operand **b** är 16-bitar. Heltalsdelen **A** är 16 bitar och placeras i destinationsoperandens minst signifikanta *word.* Resten **B** är 16 bitar och placeras i destinationsoperandens mest signifikanta *word.* Destinationsoperanden är alltid ett dataregister.

EXEMPEL

Instruktionen
```
        MOVE.L      #25,d0
        DIVU        #10,d0
```
utför heltalsdivisionen:

25/10 = 2 5/10

dvs, D0 kommer efter divisionen att ha värdet $ 0005 0002 där de minst signifikanta 16 bitarna innehåller *heltalsdelen* av resultatet och de 16 mest signifikanta bitarna innehåller *resten*. Observera att flaggsättningen vid division avviker något från den generella flaggsättningen.

5.35

EXEMPEL

Koda C-programsekvesen `a=b/c;` där deklarationerna `unsigned short a,b,c;` är globala.

Lösning:
```
        CLR.L       D0          alla bitar måste
*                               nollställas
        MOVE.W      (b).L,D0
        DIVU        (c).L,D0
        MOVE.W      D0,(a).L    spara heltalsdelen
```

5.36

EXEMPEL

Koda C-programsekvesen `a=b%c;` där deklarationerna `unsigned short a,b,c;` är globala.

Lösning:
```
        CLR.L       D0          alla bitar
*                               måste nollställas
        MOVE.W      (b).L,D0
        DIVU        (c).L,D0
        SWAP        D0          skifta h/l word
        MOVE.W      D0,(a).L    spara "resten"
```

Instruktionen `SWAP` skiftar innehållen i de *mest* signifikanta 16 bitarna med de *minst* signifikanta 16 bitarna, dvs ger oss *resten* inför den avslutande move-instruktionen

5.37

EXEMPEL

Koda C-programsekvensen

```
	carr[1]=carr[2]+4;
```

där :

```
	char  carr[10];
```

har deklarerats som global variabel. Register A0 och D7 får användas för mellanlagring.

Lösning:

Vektorn `carr` innehåller dataelement av typen `char`, dvs *byte*:

```
	MOVEA.L		#carr,A0	adress till första element
*					nu i A0
	MOVE.B		(2,A0),D7	element carr[2] nu i D7
	ADDQ.B		#4,D7		carr[2]+4
	MOVE.B		D7,(1,A0)	carr[1]=carr[2]+4
```

5.38

EXEMPEL

Koda samma sekvens som i föregående exempel, men där `carr` deklarerats enligt:

```
	short		int	carr[10];
```

Lösning:

```
	MOVEA.L		#carr,A0		adress till första
*						element nu i A0
	MOVE.W		(2*2,A0),D7		element carr[2]
*						nu i D7
	ADDQ.W		#4,D7			carr[2]+4
	MOVE.W		D7,(1*2,A0)		carr[1]=carr[2]+4
```

5.39

EXEMPEL

Koda samma sekvens som i föregående exempel, men där `carr` deklarerats enligt:

```
	int		carr[10];
```

Lösning:

```
	MOVEA.L		#carr,A0
* adress till första element nu i A0

	MOVE.L		(2*4,A0),D7
* element carr[2] nu i D7

	ADDQ.L		#4,D7			carr[2]+4
	MOVE.L		D7,(1*4,D0)	carr[1]=carr[2]+4
```

5.30

EXEMPEL

Koda C-programsekvensen:

```
a = b+c-(10+d);
```

i MC68000 assembler. a,b,c och d är globalt deklarerade enligt:

```
int   a,b,c,d;
```

hänsyn till spill behöver ej tas. Register D7 och D6 får användas för mellanlagring.

Lösning: Variabler av typ `int` (32 bitar) innebär *long*. Parentesen evalueras först.

```
MOVE.L      (d).L,D7        variabel d nu i D7
ADDI.L      #10,D7          (10+d)
MOVE.L      (c).L,D6        variabel c nu i D6
SUB.L       D7,D6           c-(10+d) nu i D6
ADD.L       (b).L,D6
MOVE.L      D6,(a).L
```

5.31

EXEMPEL

Skriv en sekvens som adderar två 64-bitars tal, N och M, utan tecken. Resultatet skall ersätta N i minnet, dvs N'=N+M. Talen N och M är definierade enligt:

```
N      DS.L      2
M      DS.L      2
```

Register D7 får användas för mellanlagring

Lösning:

```
        MOVE.L      (N).L,D7
        ADD.L       (M).L,D7      minst sign. 32 bitar
*                                 adderade
        MOVE.L      D7,(N).L            skriv tillbaks
        MOVE.L      (N+4).L,D7
        ADDX.L      (M+4).L,D7          mest sign. 32bitar
*                                 +"Carry" från ovan
        MOVE.L      D7,(N+4).L          skriv tillbaks
```

Obs: MOVE-instruktionen påverkar C-flaggan men inte X-flaggan, härav användningen av instruktionen ADDX

5.32

Övriga instruktioner för aritmetik

För unära operatorer - (*unärt minus*) och ~ (*komplementering*) används instruktionerna `NEG` (*negate*) respektive `NOT` (*logical complement*). `CLR` (*clear*) används för att *nollställa* operanden och instruktionen `EXT` (sign extend) används för *teckenutvidgning*. Alla dessa instruktioner har *en* operand.

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

EXEMPEL

Uttrycket `a = -b;` kan kodas:

```
        MOVE.L      (b).L,D0
        NEG.L       D0
        MOVE.L      D0,(a).L
```

eller alternativt:

```
        MOVE.L      (a).L,(b).L
        NEG.L       (b).L
```

5.33

Det logiska komplementet, dvs 1-komplementet, bildas på motsvarande sätt genom att instruktionen `NEG` bytes mot instruktionen `NOT`.

EXEMPEL

Bitvis negation

Instruktionsföljden

```
        MOVE.B      #%10101010,D0
        NOT.B       D0
```

ger i D0:

```
        xxxx xxxx xxxx xxxx xxxx xxxx 0101 0101
```

5.34

Operandens storlek kan vara *byte*, *word* eller *long* och flertalet adresseringsmoder finns tillgängliga.

EXEMPEL

Koda C-programsatsen

```
        a = ~b;
```

i MC68000 assembler. `a` och `b` är globala variabler av typen `int`. Använd register D7 för mellanlagring.

Lösning:

```
        MOVE.L      (b).L,D7
        NOT.L       D7
        MOVE.L      D7,(a).L
```

5.35

Teckenutvidgning

För att omvandla ett 16-bitars tal *med tecken* till ett 32-bitars tal *med tecken* (typkonvertering) måste teckenutvidgning tillgripas. Instruktionen `EXT` teckenutvidgar. Betrakta följande exempel:

EXEMPEL

Koda C-programsekvesen `a=b/c;` där deklarationerna

```
      int         a;
      short       b,c;
```

är globala.

Lösning:

```
      CLR.L       D0    alla bitar måste nollställas
      MOVE.W      (b).L,D0

* Divisionen utförs med  tecken. Resultatet
* (16-bitar) kan vara negativt
      DIVS        (c).L,D0

* Resultatet i D0 måste teckenutvidgas innan detta
* skrivs till den 32-bitars variabeln
      EXT.L       D0          teckenutvidga
      MOVE.L      D0,(a).L    spara heltalsdelen
```

5.36

5.3.7 Logiska instruktioner

Ett antal instruktioner finns tillgängliga för logiska operationer. Den unära operationen NOT, som utför bitvis negation av operanden har vi tidigare behandlat.

De binära operationerna AND, OR och EOR finns i tre olika varianter. Den första varianten förutsätter att källoperanden är en konstant. Storleken kan vara byte, word eller long:

`ANDI`	logiskt "och"
`ORI`	logiskt "eller"
`EORI`	logiskt "exklusivt eller"

EXEMPEL

logiskt "och"($\wedge$)

Instruktionsföljden

```
      MOVE.B      #%10101010,D0
      ANDI.B      #%10000010,D0
```

ger i D0: xxxx xxxx xxxx xxxx xxxx xxxx 1000 0010

5.37

EXEMPEL

logiskt "eller" ($\vee$)
Instruktionsföljden

```
MOVE.B      #%10101010,D0
ORI.B       #%01011010,D0
```

ger i D0:

```
xxxx xxxx xxxx xxxx xxxx xxxx 1111 1010
```

5.38

EXEMPEL

logiskt "exklusivt eller" ($\oplus$)
Instruktionsföljden

```
MOVE.B      #%10101010,D0
EORI.B      #%11110000,D0
```

ger i D0:

```
xxxx xxxx xxxx xxxx xxxx xxxx 0101 1010
```

5.39

Instruktionerna finns också för mer generell adressering, en av operanderna *måste* dock alltid vara ett dataregister:

```
AND   <ea>,Dn
AND   Dn,<ea>
OR    <ea>,Dn
OR    Dn,<ea>
EOR   Dn,<ea>
```

Storleken kan vara *byte*,*word* eller *long*.

EXEMPEL

Koda C-programsatsen

```
b = a | 0x10;
```

i MC68000 assembler. `a` och `b` har deklarerats enligt `int a,b;`. Använd register D7 för mellanlagring.

Lösning:

```
MOVE.L      (a).L,D7
ORI.L       #$10,D7
MOVE.L      D7,(b).L
```

5.40

EXEMPEL

Koda C-programsatsen:

```
a = a & 0xffff0000L;
```

i MC68000 assemblerkod, `a` har deklarerats enligt `int a;`

Lösning:

```
ANDI.L      #$ffff0000,(a).L
```

5.41

Slutligen finns instruktionsformer för manipulering av processorns statusregister (**SR**) eller flaggregistret (**CCR**). Instruktioner som ändrar **SR**, dvs påverkar alla 16 bitar är *priviligierade*, (behandlas senare), medans instruktioner som bara påverkar de 8 minst signifikanta bitarna alltid kan utföras.

Källoperanden för instruktioner som manipulerar processorns statusregister är alltid en konstant. Storleken är alltid *word.*

```
ANDI        #<data>,SR
ANDI        #<data,CCR
ORI         #<data>,SR
ORI         #<data>,CCR
EORI        #<data>,SR
EORI        #<data>,CCR
```

EXEMPEL

Ettställ C-flaggan, nollställ övriga flaggor i flaggregistret.
Lösning

```
ANDI        #0,CCR
ORI         #1,CCR
```

Nollställ flaggregistret
Lösning:

```
ANDI        #0,CCR
```

5.42

5.3.8 Skiftoperatorer

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

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

Medan B måste vara en variabel, kan A vara såväl en konstant som någon (annan) variabel.

Det finns åtskilliga olika tillämpningar av skiftoperationer. Exempelvis kan multiplikation med en konstant 2^x utföras genom att operanden skiftas vänster x antal positioner. På motsvarande sätt kan en division med en konstant 2^y utföras genom att operanden skiftas höger y antal positioner.

EXEMPEL

Operationen:
```
number * 128
```
kan skrivas som
```
number << 7
```
vilket ger en betydande tidsvinst vid exekveringen eftersom skiftoperationen är avsevärt mycket snabbare än multiplikationsoperationen.

5.43

Skiftinstruktioner

MC68000 stödjer skiftoperationer med fyra olika instruktioner:

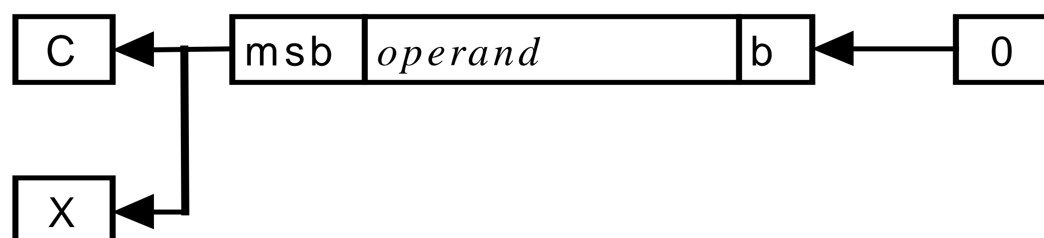
logiskt skift	(LS)
aritmetiskt skift	(AS)
rotation med Carry	(RO)
rotation med Extend	(ROX)

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

1. ***instruktion*** **`<ea>`** , "minnesshift", <ea> anger en minnescell, innehållet i denna skiftas 1 steg. Operandens storlek är här alltid *word*
2. ***instruktion*** **`#<count>,Dn`** "statiskt skift", innehållet i dataregister skiftas det antal steg som anges av källoperanden. storlek kan då vara *byte*, *word* eller *long*.
3. ***instruktion*** **`Dx,Dy`** "dynamiskt skift" innehållet i något dataregister (Dy) skiftas det antal steg som anges av register Dx.

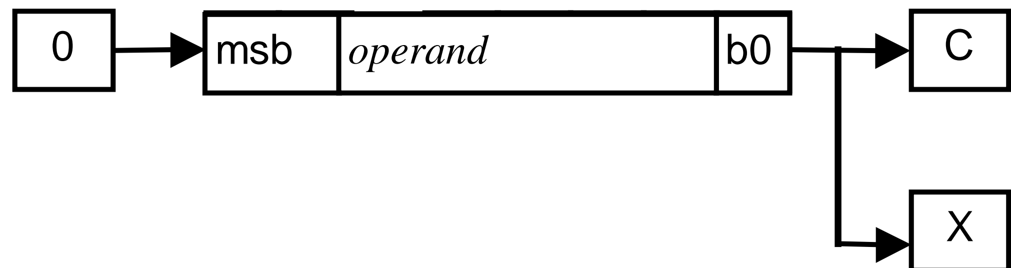
`LSL` *logical shift left*

Operandens mest signifikanta bit kopieras till såväl C som X flaggan.
En nolla skiftas in från höger



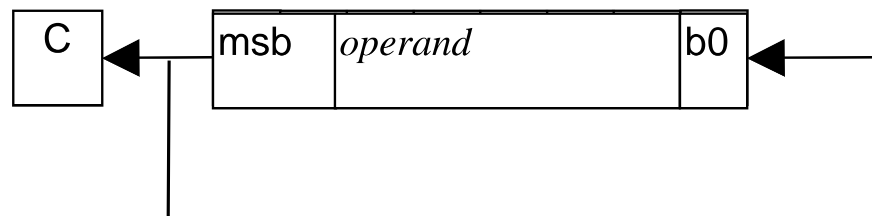
`LSR` ***logical shift right***

Operandens minst signifikanta bit kopieras till C respektive X. En nolla skiftas in i den mest signifikanta positionen



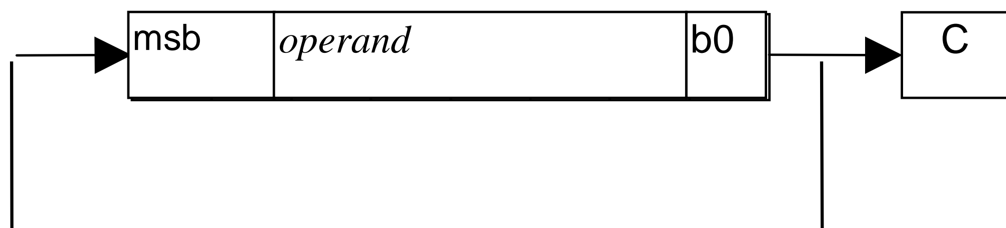
`ROL` ***rotate left***

Operanden “roteras” åt vänster eftersom den mest signifika biten kopieras och blir ny minst signifikant bit



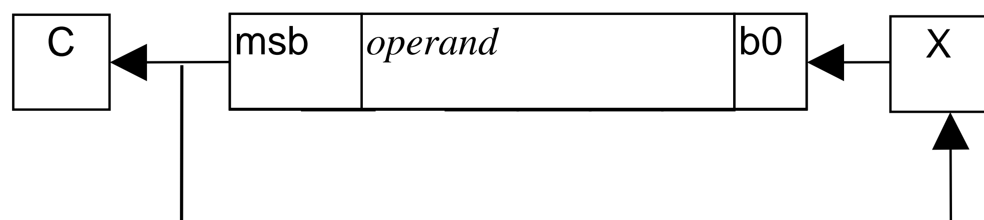
`ROR` ***rotate right***

Operanden “roteras” åt höger eftersom den minst signifika biten kopieras och blir ny mest signifikant bit



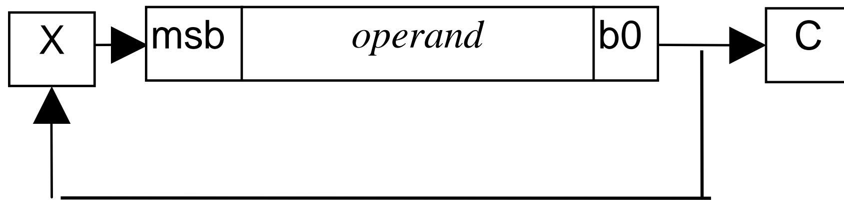
`ROXL` ***rotate left with Extend***

Här deltar X flaggan i rotationen av operanden



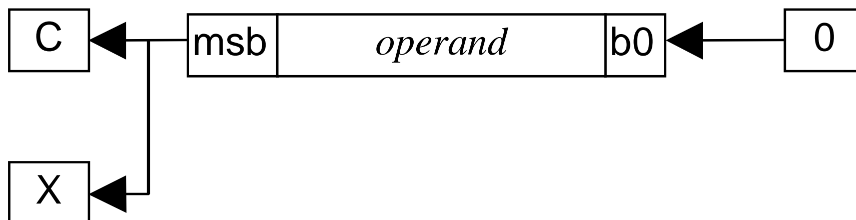
`ROXR` ***rotate right with Extend***

Här deltar X flaggan i rotationen av operanden



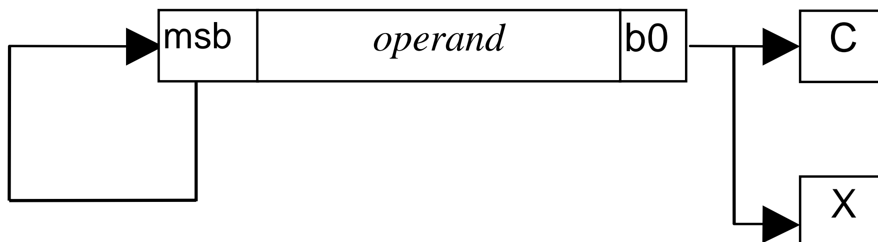
`ASL` ***arithmetic shift left***

Denna instruktion är densamma som `LSL`



`ASR` ***arithmetic shift right***

Aritmetiskt högerskift innebär att teckenbiten behålls i talet.



EXEMPEL

Koda C-uttrycket `result = number << count ;` om:
`number`, `result` är `int` och `count` är en `short int`

Lösning:

```
        MOVE.W      (count).L,D1
        EXT.L       D1
        MOVE.L      (number).L,D0
        LSL.L       D1,D0
        MOVE.L      D0,(result).L
```

5.43

5.3.9 Jämförelser och test

Villkorliga programflöden, eller så kallade *val* görs utgående från exempelvis *test* av en variabel. Betrakta följande programkonstruktion: (`flag` kan vara noll (falskt) eller ett (sant))

```
if( flag ){
      trueStmt;
} else {
      falseStmt;
}
```

Variabeln “flag” testas, om denna är skild från noll utförs program-sekvensen omedelbart efter “if”-satsen (*trueStmt*), annars utförs “else”-satsen, (*falseStmt*).

Boolesk komplement

0	*1*
1	*0*
flag	*! flag*
! flag	*flag*

Vi kan arrangera om programkonstruktionen genom att ändra testen. Följande programsekvens har samma funktion som ovanstående:

```
if( ! flag ){
      falseStmt;
} else {
      trueStmt;
}
```

“!”-operatorn (not) innebär att vi skapar det boolska komplementet av variabeln *innan* själva testen utförs.

test *är en* unär *operation*

Till skillnad från *test*, som använder *en* operand är *jämförelse* en binär operation. Vi använder operatorer för att ange *likhet*, *olikhet* och andra relationer.

Observera att för några enkla (men vanliga) specialfall kan en *test* ersätta en *jämförelse*. Typiskt gäller detta vid jämförelse med 0, dvs följande konstruktioner är likvärdiga:

`if(flag)`	är likvärdig med	`if(flag != 0)`
`if(flag == 0)`	är likvärdig med	`if( ! flag)`

Instruktioner för jämförelser och test

MC68000 tillhandahåller tre speciella instruktioner vars funktion är att utföra test respektive jämförelser. Instruktionerna *påverkar* inte någon av operanderna utan ställer endast flaggor i **CCR** beroende på testens/jämförelsens utfall.

`TST` *test an operand*

kan användas med datastorlek *byte, word* eller *long*. Flaggorna Z respektive N sätts beroende på operanden medan C och V alltid nollställs.

EXEMPEL

Antag innehållet i D0: `%11110100`

```
TST.B        D0
```

Flaggsättning:

$1 \rightarrow$ N, ty b7 i D0 är 1
$0 \rightarrow$ Z, ty D0.b är skilt från 0
$0 \rightarrow$ V (alltid)
$0 \rightarrow$ C (alltid)
X påverkas ej av instruktionen

5.44

Om vi vill, kan vi utföra test av *enskilda bitar* med hjälp av `BTST` *test a bit*. Instruktionen påverkar *bara* Z-flaggan i **CCR**, övriga flaggor behåller sitt föragående värde. Instruktionen används *huvusakligen* i någon av följande två former:

- Testa en bit på någon adress i minnet
- Testa en bit i ett dataregister

I det första fallet kan en av 8 bitar testas, storleken är alltså `BTST.B`, i det andra fallet kan en godtycklig bit i dataregistret testas, dvs en av 32, och storleken är här `BTST.L`. Biten som ska testas kan, i båda fallen anges antingen som en konstant (*static*) med *immediate* adresseringsmod, *eller* som en variabel (*dynamic*), ett dataregister används då för att ange biten som ska testas.

EXEMPEL

Testa *minst signifikanta bit* på adress $A000.

```
        BTST.B          #0,($A000).L
```

Testa *mest signifikanta bit* på adress $A000.

```
        BTST.B          #7,($A000).L
```

Testa den bit som anges av register **D0**, på adress $A000. Endast de tre minst signifikanta bitarna i **D0** används

```
        BTST.B          D0,($A000).L
```

Testa den *minst signifikanta biten* i **D0**

```
        BTST.L          #0,D0
```

Testa den *mest signifikanta biten* i **D0**

```
        BTST.L          #31,D0
```

Testa den bit i **D0** som anges av register **D1**, endast de 5 minst signifikanta bitarna i **D1** används.

```
        BTST.L          D1,D0
```

5.45

För *jämförelser* finns `CMP` (*compare*) instruktionerna. `CMP` utför en subtraktion, på samma sätt som `SUB`-instruktionen men till skillnad från `SUB` påverkas endast flaggorna. Destinationsoperanden lämnas alltid opåverkad. Instruktionen finns i tre olika former beroende på operanderna:

- `CMPA` då destinationsoperanden är ett adressregister
- `CMPI` då *källoperanden* är en konstant
- `CMP` i övriga fall

EXEMPEL

```
        CMP.B           D0,($A000).L
```

Operation: innehåll **D0** – innehåll ($A000) → **CCR**

Endast de 8 minst sign. bitarna ingår i operationen

```
        CMPI.W          #400,D0
```

Operation: innehåll **D0** – 400 → **CCR**

Endast de 16 minst sign. bitarna ingår i operationen

```
        CMPA.L          #$2000,A0
```

Operation: innehåll **A0** – $2000 → **CCR**

5.46

Villkorliga instruktioner

Villkorliga instruktioner används, som namnet antyder, för att utföra en eller flera instruktioner då någon förutsättning är uppfylld. Ofta kallas dessa instruktioner för hopp instruktioner eller *branch*-instruktioner (*branch*: gren). I instruktionslistan anges ett antal *Bcc*-instruktioner som utför ett programhopp om hoppvillkoret är uppfyllt, (*Bcc: branch conditionally*). Villkorliga instruktioner används alltså alltid tillsammans med någon (omedelbart föregående) instruktion som åstadkommit flaggsättning. Ofta är detta någon av instruktionerna `cmp`, `tst` eller `btst` men det kan också vara i kombination med någon aritmetisk instruktion. Det är därför viktigt att du alltid kontrollerar hur flaggorna sätts av instruktionen som föregår branch-instruktionen.

En villkorlig (branch-) instruktion:

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

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

Ett villkor består av termerna N,C,V och Z (flaggorna i **CCR**) ingående i ett sanningsuttryck:

Ett sanningsuttryck kan bestå av en *enkel* term som:

Z är SANT om Z-flaggan är 1

eller en enkel *negerad* term som

$\overline{Z}$ (“icke Z”) är SANT om Z-flaggan är 0

Men vi kan också forma sanningsuttryck med flera termer och logiska operatorer

$C \vee V$ är SANT on C-flaggan är 1 ELLER V-flaggan är 1

På motsvarande sätt har *varje* villkor ett *komplementärt* villkor så att testerna kan utföras på såväl heltal som på naturliga tal. (Jämför Kapitel 3).

EXEMPEL

Den villkorliga hoppinstruktionen BEQ (*branch* equal) utförs endast om Z-flaggan är 1 i annat fall fortsätter exekveringen med *nästa* instruktion.

```
    CMPI.L  #100,D0
```
* Om innehållet i D0 är 100 kommer denna
* instruktion att sätta Z-flaggan till 1

```
    BEQ     Om_100
```
* Hoppet utförs endast om Z-flaggan är 1

```
Inte_100:
```
denna sekvens utförs alltså
om innehållet i D0 är *skilt* från 100

```
Om_100:
```
denna sekvens utförs alltså
om innehållet i D0 är *lika med* 100

5.47

EXEMPEL

Vi har ett *komplementärt* villkor i instruktionen BNE (*branch not equal*), dvs hoppet utförs om Z-flaggan är 0, i annat fall fortsätter exekveringen med nästa instruktion.

```
    CMPI.L        #100,d0
    BNE           Inte_100

Om_100:
```

denna sekvens utförs om innehållet i D0 är *lika* med 100

```
Inte_100:
```
denna sekvens utförs om innehållet i D0 är *skilt* från 100

5.48

Följande tabell sammanfattar de olika villkoren och respektive mnemonic för *branch*-instruktionen. (se även instruktionslistan).

mnemonic	villkor	sanningsuttryck
`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	
`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	$Z \vee N \wedge \overline{V} \vee \overline{N} \wedge V$

I tabellen visas hur flaggorna C,Z,N och V i CCR används för att bilda respektive villkor

Låt oss resonera lite grand kring var de olika villkoren kan komma till användning. Självfallet handlar det om då vi vill testa tal, men frågan är om vi betraktar de testade talen som

naturliga tal (0,1,2,3 ..;MAX)

eller om vi betraktar den som

heltal (-MIN,...-1,0,1,...MAX).

dvs. som tal *med* eller *utan* tecken.

Betrakta följande exempel:

EXEMPEL

Vi jämför instruktionerna `BHI` (*branch high*) och `BGT` (*branch greater than*). Antag att vi vill testa om innehållet i D0 är *större* än -2, vi prövar med kodsekvensen

```
        CMPI.B      #-2,d0
        BHI         St_n2
        ...     denna kod ska utföras om innehållet i D0 är mindre än eller
                lika med -2
St_n2:
        ...     denna kod ska utföras om innehållet i D0 är större än -2
```

Vi antar att D0:s innehåll är 3, vilket bör resultera i att instruktionen (hoppet) *utförs* ty 3 > -2. Vi utför nu operation och flaggsättning:

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

Observera speciellt att en *Borrow* genereras vid operationen

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

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

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

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

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

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

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

5.49

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

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

Av tabellen framgår hur operator *tillsammans* med den aktuella datatypen får avgöra villken villkorlig instruktion som väljs.

5.3.10 Bitoperationer

Vi har tidigare sett hur MC68000 stödjer datatyperna `char`, `int`, `short int` och *pekartyper,* genom att tillhandahålla operandstorlekar *byte, word, long* och speciella adressregister med lämpliga adresseringsmoder. I olika sammanhang har man därutöver behov av så kallade *boolska variabler* dvs en datatyp som kan representera ett SANT eller FALSKT värde.

Enklaste representationen av en sådan datatyp är en enstaka bit, där exempelvis 0 betecknar FALSKT medan 1 betecknar SANT. För en sådan datatyp tillhandahåller MC68000 speciella *bit*-instruktioner. Vi har tidigare sett exempel på hur enskilda bitar kan testas, men vi kan också *ettställa, nollställa* eller *ändra* enskilda bitar med hjälp av instruktionerna `BSET` (*bit set*), `BCLR` (*bit clear*) och `BCHG` (*bit change*).

EXEMPEL

```
bool  DS.B        1
...
      BCLR.B      #0,(bool).L
* nollställ minst signifikanta bit

      BSET.B      #7,(bool).L
* ettställ mest signifikanta bit

      BCHG.B      #1,(bool).L
* ändra ("toggla") bit 1
```

5.50

Destinationsoperanden kan också vara ett dataregister. I detta fall är storleken *long* och alltså kan en av de 32 bitarna i registret påverkas.

Bitoperationerna används också ofta vid in- och ut-matning (se kapitel 6) då man *testar, ettställer* eller *nollställer* enstaka bitar i ett register.

5.3.11 If - konstruktioner

If-konstruktionen innebär en villkorlig operation, dvs om ett bestämt villkor är uppfyllt, skall en speciell *exekveringsväg* väljas, annars utelämnas exekveringen av koden i den exekveringsvägen. En rad olika villkor kan uppträda och det finns som bekant inte mindre än 14 st olika så kallade *villkorliga branch-instruktioner*.

EXEMPEL

Genom att använda en kombination av *jämförelse*- och *branch*-instruktioner kan en If-sats enkelt kodas i assembler.

```
if(a == 4)
      villkorlig kod;

gemensam kod;
```

kan kodas

```
        CMPI.L      #4,(a).l
        BNE         gemensam kod
villkorlig kod:
        ...
        ...
gemensam kod:
        ...
```

5.51

I exemplet jämförs först innehållet i variabeln `a`, med konstanten 4. Denna jämförelse påverkar innehållet i **CCR**. Om `a` innehåller 4 kommer Z-flaggan att sättas till 1, för alla andra resultat av jämförelsen sätts Z-flaggan till 0 (`a` är skilt från 4). Den efterföljande *branch* instruktionen, `BNE` (*branch not equal*), testar innehållet i **CCR**. Instruktionen utför ett hopp endast om Z-flaggan är 0. Följaktligen exekveras bara *villkorlig kod* i exemplet om villkoret i if-satsen är uppfyllt.

Då vi testade *likhet* använde vi `BNE` i exemplet ovan. Hade vi i stället velat testa *olikhet* hade instruktionen `BEQ` (*branch equal*) varit lämpligare. Det kan synas märkligt att associera *likhet* med `BNE`, respektive *olikhet* med `BEQ`, men observera att det handlar om att koda if-satsen på ett lämpligt sätt. Ofta kan samma funktionalitet nås med en alternativ kodning, i detta fall exempelvis:

EXEMPEL

```
        CMPI.L      #4,(a).l
        BEQ         villkorlig kod
        BRA         gemensam kod
villkorlig kod:
        ...
        ...
        ...
gemensam kod:
        ...
```

5.52

Dvs, samma funktionalitet men med användning av andra och fler instruktioner än i det förra fallet. I detta fallet tillkommer den ovillkorliga BRA (*branch always*).

Vi är nu mogna att fortsätta översätta vårt inledande programexempel, men först bör vi uppmärksamma en speciell egenskap hos ASCII-representationen. Versaler karakteriseras utav att bit 5 alltid är 0, medan för motsvarade gemena är bit 5 alltid 1. Exempelvis gäller att ASCII-tecknet för 'A' är $41 (% 0100 0001) medan ASCII-tecknet för 'a' är $61 (%0110 0001). Detta gäller alla tecken i alfabetet och ger oss möjlighet att utföra testen **isupper(D1)** med hjälp av en bit-test instruktion.

```
        BTST.L      #5,d1
```

testar bit 5 i register **D1** och sätter Z-flaggan till 1 om biten är 0, annars sätts Z-flaggan till 0. Observera att vi kan placera en villkorlig branch direkt efter denna instruktion eftersom branch-instruktionen testar Z-flaggan. Om vi översätter funktionen **isupper(D1**) får vi följande:

```
        /* deklarationer */
mening      DS.B  128

        /* program */

        MOVE.L      #0,D0
        MOVEA.L     #mening,A0
        MOVE.B      (A0,D0),D1

        while( D1 != ' '){
        BTST.L      #5,D1
        BNE         lowercase
                D1=tolower(D1);
lowercase:
        MOVE.B      D1,(A0,D0)
        ADDI.L      #1,D0
        MOVE.B      (A0,D0),D1
        }
```

```
index = 0;
tecken = mening[ index ];               /* från första */
while( tecken != '.'){                  /* till "punkt" */
    if ( isupper( tecken) )             /* om Versal */
        tecken = tolower(tecken);       /* konvertera*/
    mening[index] = tecken;             /* ersätt */
    index = index + 1;
    tecken = mening[index];             /* nästa */
}
```

5.3.12 If/else konstruktioner

If/else konstruktionen är en utvidgning av if-konstruktionen. Den används då man vill välja *endast* en av två möjliga exekveringsvägar.

EXEMPEL

Koda följande satser i assembler

```
if( a == 100)
      villkorlig kod 1;
else
      villkorlig  kod 2;
gemensam kod;
```

Lösning

```
        CMPI.L      #100,(a).L
        BEQ         villkorlig kod 1
villkorlig kod 2:
        ...
        ...
        BRA         gemensam kod
villkorlig kod 1:
        ...
        ...
gemensam kod:
        ...
```

5.52

Vi ser att vi kan skriva motsvarande kod genom att använda den *komplementära* instruktionen BNE, enligt:

EXEMPEL

```
        CMPI.L      #100,(a).L
        BNE         villkorlig kod 2
villkorlig kod 1:
        ...
        ...
        BRA         gemensam kod

villkorlig kod 2:
        ...
gemensam kod:
        ...
```

5.53

5.3.13 While-konstruktioner

While-konstruktionen är en sätt att åstadkomma upprepningar (iterationer) i ett program. Konstruktionen har vissa likheter med if-, men skiljer sig på en väsentlig punkt. Medan if-villkoret endast testas en gång måste While-villkoret testas inför varje iteration.

EXEMPEL

Koda följande satser i assembler

```
        while( a != 100) {
                iteration;
        }
        villkorslös kod;
```

Lösning:

```
test:
        CMPI.L      #100,(a).L
        BEQ         villkorslös kod
iteration:
        ...
        BRA         test
villkorslös kod:
        ...
```

5.54

Vi kan nu bygga vidare på den översättning vi började på i de inledande avsnitten med "while" konstruktionen. Efter att ha ersatt i den formella beskrivningen får vi:

```
        /* deklarationer */
mening      DS.B  128
        /* program */
        MOVE.L      #0,D0
        MOVEA.L     #mening,A0
        MOVE.B      (A0,D0),D1
test:
        CMPI.B      #'.',D1
        BEQ         end
* iteration...
        BTST.L      #5,D1
        BNE         lowercase
                D1=tolower(D1);
lowercase:
        ADDI.L      #1,D0
        MOVE.B      (A0,D0),D1
        BRA         test
end:
```

```
index = 0;
tecken = mening[ index ];                 /* från första */
while( tecken != '.'){                    /* till "punkt" */
    if ( isupper( tecken) )               /* om Versal */
        tecken = tolower(tecken);         /* konvertera*/
    mening[index]  =  tecken;             /* ersätt */
    index = index + 1;
    tecken = mening[index];               /* nästa */
}
```

5.3.14 Subrutiner (funktioner)

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

- Subrutinens Indata
- Subrutinens Funktion
- Subrutinens Utdata

Om dessa egenskaper dokumenterats väl är det fullt möjligt för en annan programmerare att använda subrutinen utan att samtidigt tvingas sätta sig in i den detaljerade funktionen. Det är därför av yttersta vikt att denna dokumentation finns med och att den utformats på rätt sätt.

MC68000 stöder modularisering bland annat med instruktionerna `BSR` (*branch to subroutine*) och `RTS` (*return from subroutine*). `BSR` kan ses som en ovillkorlig hopp-instruktion, men till skillnad från exempelvis `BRA`, gäller att exekveringen ska fortsätta direkt efter `BSR` då subrutinen utförts. Följakligen måste *returadressen*, dvs adressen till instruktionen omedelbart efter `BSR` sparas på något sätt. På motsvarande sätt måste adressen kunna placeras i **PC** då instruktionen `RTS` exekveras. MC68000 använder stacken för att spara returadresser, dvs vid:

`BSR` placeras adressen till nästa instruktion på stacken, stackpekaren minskas med 4 bytes, adressen till subrutinen placeras i **PC**

och vid:

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

EXEMPEL

Skriv en subrutin, `tolower`, som konverterar en ASCII-versal till motsvarande gemen. Visa hur subrutinen kan användas om ASCII-tecknet skickas till rutinen i register D1, och det konverterade tecknet returneras i samma register.

Lösning: Skriv först en så kallad "header" för subrutinen:

```
****
* Subroutine tolower
* Converts ASCII-uppercase to ASCII-lowercase
*     Input:      D1.B        ASCII to be converted
*     Output:     D1.B        converted ASCII
```

Skriv nu själva subrutinen:

```
tolower:
      ANDI.B      #%11011111,D1     clear bit 5
      RTS
```

Anrop av subrutinen:

```
      MOVE.B      (tecken).L,D1
      BSR         tolower
```

5.54

Vi använder nu rutinen `tolower` för att avsluta översättningen av vårt inledande exempel. Låt oss då samtidigt modifiera den ursprungliga specifikationen till en specifikation av subrutinen `scanstr` enligt:

```
***
* Subroutine scanstr
* scans an ASCII-textstring until period.
* Converts uppercase characters to lowercase
*     Input:     A0    Address of textstring
*     Output     None
*     Registers D0 and D1 are modified

scanstr:
      MOVE.L     #0,D0             index variable
      MOVE.B     (A0,D0),D1        first character

scanstr_1:
      CMPI.L     #'.',D1
      BEQ        scanstr_2

      BTST.L     #5,D1             if lowercase
      BNE        scanstr_3
      BSR        tolower           char in D1
      MOVE.B     D1,(A0,D0)        replace with lowercase

scanstr_3:
      ADDI.L     #1,D0             next ...
      MOVE.B     (A0,D0),D1        ... character
      BRA        scanstr_1

scanstr_2:
      RTS
```

Observera valet av symbolnamn för adresser *interna* för `scanstr` (`scanstr_1`, `scanstr_2` och `scanstr_3`). Detta är ett bättre val än mer generella namn som `loop`, `iteration`, `exit` osv som lätt kan associeras till *flera* olika subrutiner.

Ett huvudprogram som använder subrutinen `scanstr` kan nu utformas på följande sätt:

```
*
*     MAIN program
*
      ORG         $5000
mening:
      DS.B        128

      ORG         $4000
      MOVEA.L     #mening,A0
      BSR         scanstr
forever:
      BRA         forever
```

5.3.15 Parameteröverföring

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

EXEMPEL

Parametrar i register

Om vi har följande (globala) deklarationer:

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

så kan funktionsanropet

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

översättas till:

```
      MOVE.L      (la).L,D0
      MOVE.L      (lb).L,D1
      MOVE.L      (lc).L,D2
      BSR         dummyfunc
```

Då vi kodar subrutinen dummyfunc vet vi (på grund av våra regler) att den första parametern skickas i **D0**, den andra i **D1** och den tredje i **D2** (osv).

5.55

Det visade exemplet indikerar dock en rad problem för mer generella fall. Exempelvis kan parameterlistor vara långa och hur gör vi om inte processorns register räcker till?... Ett annat problem kan vara att vi redan använt register för mellanlagring av temporära resultat och därför inte kan upplåta dessa för parameteröverföring.

I de följande avsnitten kommer vi därför också att behandla några andra metoder för parameteröverföring till och från subrutiner. I avsnitten om *kombinerad programmering* dvs hur man konstruerar program i *såväl* assemblerkod som ett högnivåspråk, återkommer vi till detta och visar vilken metod som vanligtvis används av C-kompilatorer.

Stacken används för temporär lagring

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

Då vi sparar ett registerinnehåll på stacken (detta kallas av tradition "push") kan vi föreställa oss att vi lägger detta, överst, på en "hög" (eng. stack), då vi återställer (kallas av tradition "pop") innebär detta att vi tar, det som ligger överst, på "högen". Jämför detta med hur MC68000 placerar *returadressen* vid subrutinanrop, på stacken (se kapitel 4) där register **A7** (eller **sp**) utgör stackpekare. Av resonemanget framgår vikten av att vi *lägger* och *plockar* på och från stacken i rätt ordning.

MC68000 stödjer stackhantering med adresseringsmoderna

preautodecrement `-(SP)`,

och

postautoincrement `(SP)+`.

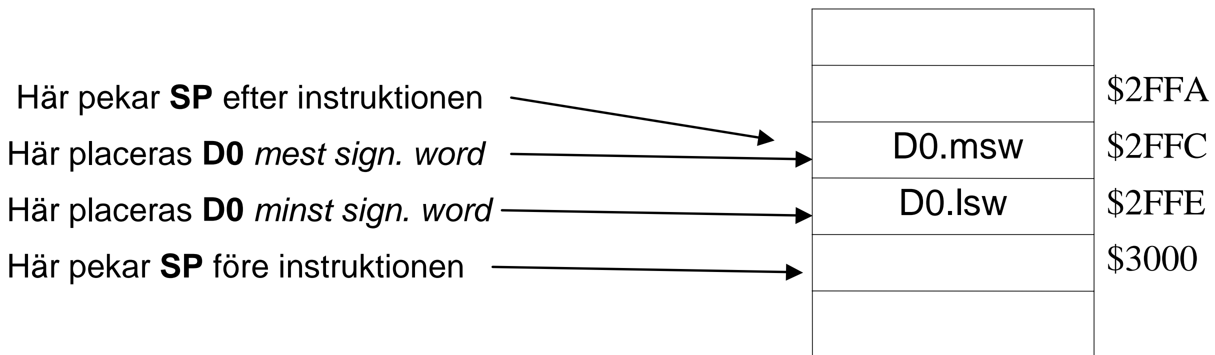
Dessutom finns `MOVEM`- instruktionen (*move multiple registers*). Instruktionen kan användas såväl med storleken *word* som storleken *long* . I det förra fallet sparas 2 bytes på stacken och i det senare fallet sparas 4 bytes på stacken.

Spara registerinnehåll på stacken

Antag att stackpekaren (register **SP**) har värdet $3000 och att vi utför nu instruktionen:

```
MOVEM.L    D0,-(SP)
```
"push **D0**"

Innehållet i register **D0** sparas på stacken. Detta kan illustreras av följande:



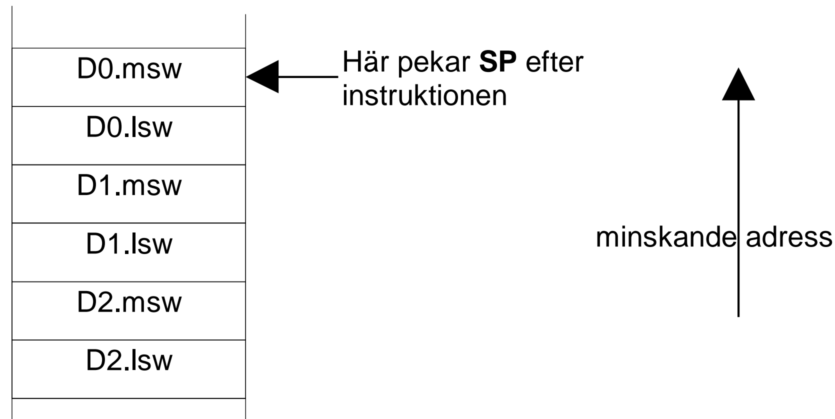
Flera register kan anges. Sekventiella register betecknas med bindestreck.

EXEMPEL

Spara register D0,D1 och D2 på stacken

```
MOVEM.L         D0-D2,-(SP)
```

Stackens utseende efter instruktionen blir:



5.56

Vi kan även ange *icke-sekventiella* register och det går också utmärkt bra att spara innehållet i adressregister. Vi anger en *icke-sekventiell* lista genom att skilja registerbeteckningarna med '/'.

EXEMPEL

Spara register D0,D2,A0,A2

```
MOVEM.L         D0/D2/A0/A2,-(SP)
```

5.57

På liknande sätt som vi "lagt på högen", dvs sparat registerinnehåll, kan vi "plocka från högen" vilket innebär att vi *återställer* registerinnehåll. Då vi anger en lista av register, har det ingen betydelse i vilken ordning vi anger dessa. Om vi däremot delar upp instruktionerna (se följande exempel) måste detta utföras i rätt ordning. Registerinnehållen som sparats enligt tidigare exempel kan återställas med instruktionen:

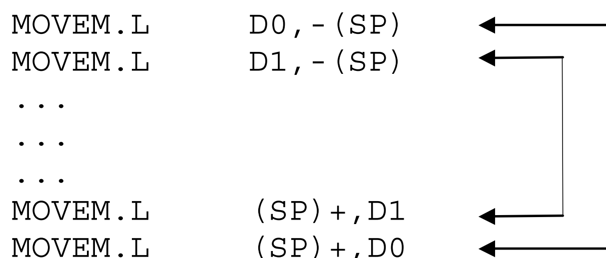
```
MOVEM.L            (SP)+,A0/A2/D2/D0
```

Det har alltså ingen betydelse *i viken ordning* registernamnen räknas upp.

Om vi i stället använt *flera instruktioner* är det viktigt att vi återställer i motsvarande ordning:

EXEMPEL

Om vi *sparat* registerinnehåll med instruktionsföljden så måste dessa återställas enligt "stack-principen", dvs, det som sist lades dit, plockas först upp:



5.58

Vi kan alltså komplettera vårt inledande exempel med instruktioner för att spara respektive återställa registerinnehåll, och på detta sätt *bevara* dessa över subrutinanrop:

```
MOVEM.L    D0-D2,-(SP)      spara register-innehåll
MOVE.L     (la).L,D0
MOVE.L     (lb).L,D1
MOVE.L     (lc).L,D2
BSR   dummyfunc
MOVEM.L    (SP)+,D0-D2      återställ registerinnehåll
```

Parameteröverföring via register kan vi alltså använda då vi med säkerhet vet vilka register som används av det anropande programmet *och hur* parametrar respektive returvärdet behandlas.

EXEMPEL

I detta exempel ska vi se hur 32-bitars multiplikation av tal *utan* tecken kan genomföras av programvara. MC68000 stöder inte 32-bitars multiplikation i hårdvara. Däremot finns en instruktion för 16 bitars multiplikation av tal utan tecken. I detta exempel konstruerar vi därför en *subrutin* som kan användas för 32-bitars multiplikation. Subrutinen används enligt:

```
            MOVE.L      {Operand a},D0
            MOVE.L      {Operand b},D1
            JSR         ULongMul
*     Resultatet finns nu i register D0
      ...
```

Lösning:

Låt oss börja med att dela upp operanderna **a** och **b** i 16-bitars tal, dvs vi skriver:

$\mathbf{a} = \mathbf{ah} * 2^{16} + \mathbf{al}$

där **ah** är de 16 mest signifikanta bitarna av **a,** och **al** är de 16 minst signifikanta bitarna av **a**, och:

$\mathbf{b} = \mathbf{bh} * 2^{16} + \mathbf{bl}$

där **bh** är de 16 mest signifikanta bitarna av **b** och **bl** är de 16 minst signifikanta bitarna av **b**.

Multiplikationen kan då skrivas:

$\mathbf{a}*\mathbf{b}=(\mathbf{ah}*2^{16}+\mathbf{al}) * (\mathbf{bh}*2^{16}+\mathbf{bl})$

multiplicerar vi ihop högerledet får vi:

$\mathbf{a}*\mathbf{b}=(\mathbf{ah}*\mathbf{bh})2^{32} + (\mathbf{al}*\mathbf{bh}+\mathbf{ah}*\mathbf{bl})2^{16}+\mathbf{al}*\mathbf{bl}$

Vi konstaterar att den första termen i högerledet aldrig kan rymmas i resultatet (ty denna är alltid större än 32 bitar, och resultatet skulle rymmas i **D0**) och stryker därför denna term. Vår formel för multiplikation av två 32-bitars tal blir:

$\mathbf{a}*\mathbf{b}=(\mathbf{al}*\mathbf{bh}+\mathbf{ah}*\mathbf{bl})2^{16}+\mathbf{al}*\mathbf{bl}$

```
ULongMul:
* spara parametrar, dom behövs flera gånger
            MOVEM.L     D1,-(SP)           spara b
            MOVEM.L     D0,-(SP)           spara a
* nu gäller följande offseter:
bl          EQU         6
bh          EQU         4
al          EQU         2
ah          EQU         0
            MOVE.W      (ah,SP),D1
            MULU        (bl,SP),D1         ah * bl -> d1
            MOVE.W      (al,SP),D0
            MULU        (bh,SP),D0         al * bh -> d0
            ADD.L       D0,D1              (ah*bl)+(al*bh) -> d1
            SWAP        D1
            CLR.W       D1                 ((ah*bl)+(al*bh)) 2^16
            MOVE.W      (al,SP),D0
            MULU        (bl,SP),D0         al*bl -> d0
            ADD.L       D1,D0   ((ah*bl)+(al*bh)) 2^16 + al*bl -> d0

*     Resultatet finns nu i D0
            ADDQ.L      #8,SP              återställ stacken
            RTS
```

5.59

Nu kan vi enkelt fortsätta med en programrutin som multiplicerar två 32-bitars tal *med* tecken. Strategin är enkel, vi kontrollerar först tecknen hos operanderna, om tecknen är *lika* ska resultatet ha positivt tecken,

om tecknen är *olika* ska resultatet ha negativt tecken. Vi försäkrar oss därefter om att båda operanderna är positiva (ev. med hjälp av tvåkomplementering), nu kan vi använda den tidigare programrutinen för multiplikation av tal utan tecken, slutligen förser vi resultatet med rätt tecken.

Vi har samma förutsättningar som tidigare, dvs **Operand a** i register **D0** och **Operand b** i register **D1**, dock använder vi nu också register **D2** som temporär variabel (för att "minnas" tecknet).

EXEMPEL

32-bitars multiplikation av tal med tecken

```
LongMul:
      CLR.W       D2          d2=0 -> innebär positivt resultat
      TST.L       D0
      BGE         LongMul1
      NEG.L       D0          absolutvärde...
      NOT.W       D2          d2!=0 -> innebär negativt resultat

LongMul1:
      TST.L       D1
      BGE         LongMul2
      NEG.L       D1          absolutvärde...
      NOT.W       D2          byt tecken

LongMul2:
*     nu är båda operanderna större än 0...
      JSR   (ULongMul).L

*     nu sätter vi tecken på resultatet...
      TST.W       D2
      BEQ         LongMul3
      NEG.L       D0          byt tecken

LongMul3:
      RTS
```

5.60

De visade exemplen illustrerar hur vi kan skapa *konventioner,* dvs *regler* för att konstruera och använda subrutiner i programpaket. En svaghet, så här långt, är ju förstås att vi inte kan använda *valfritt* antal parametrar.

Parametrar överförda via stacken

Det mest generella sättet att överföra parametrar är via stacken. Metoden har fördelen att *antalet* parametrar inte är beroende av antalet register i processorn. Ett subrutinanrop föregås då av ett antal instruktioner som placerar parametrarna på stacken. Efter

subrutinanropet måste stacken återställas. I subrutinen refereras parametrarna via den offset de får i förhållande till stackpekaren.

EXEMPEL:

Antag att följande funktion definierats:

```
dummyfunc(int x, int y, int z);
```

la,lb och lc har deklarerats globalt. Översätt funktionsanropet

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

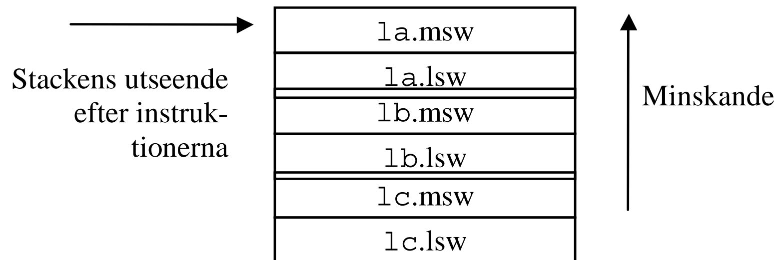
till ett anrop där parametrarna överförs via stacken. Visa hur parametrarna refereras i funktionen.

Lösning:

Vi behandlar parameterlistan från höger till vänster. Dvs placerar i tur och ordning parametrarna lc,lb och la på stacken. Detta kan göras på *flera* sätt men det enklaste är:

```
MOVE.L      (lc).L,-(SP)
MOVE.L      (lb).L,-(SP)
MOVE.L      (la).L,-(SP)
```

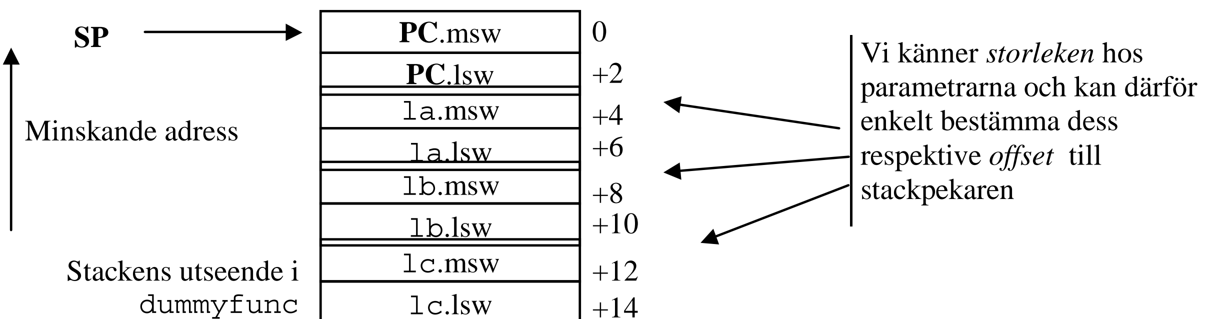
Vi har nu följande stack:



Nu utförs instruktionen

```
BSR         dummyfunc
```

Eftersom *instruktionen* BSR *sparar återhoppsadressen* på stacken växer denna ytterligare. I funktionen dummyfunc har vi därför följande stack:




```
MOVE.L      (8,SP),D0        värdet av lb nu i D0
MOVE.L      (4,SP),D1        värdet av la nu i D1
MOVE.L      (12,SP),D2       värdet av lc nu i D2
```

Parametrarna kan nu användas i subrutinen. Efter subrutinanropet måste stacken återställas, vi har inte längre någon användning av parametrarna så vi "städar" helt enkelt med instruktionen:

```
ADDA.L      #12,SP
```

5.61

Parametrar i programkod (In Line)

Ett annat sätt att överföra parametrar är *direkt i koden*. Metoden förutsätter då att parametrarna är konstanta. Denna metod används nästan undantagslöst bara i kombination med speciella instruktioner.

EXEMPEL

"In line" parameteröverföring

```
      BSR   dummyfunc
      DC.W  10
      NOP
```

I `dummyfunc` måste nu återhoppsadressen (på stacken) modifieras. Annars kommer konstanten 10 att tolkas som en instruktion.

5.61

Temporär lagring på stacken

Vid sidan av parameteröverföring kan vi också använda stacken för lagring av *temporära* (lokala) variabler. Lokala variabler har den egenskapen att de bara "finns" i en subrutin och kan alltså bara användas i denna. Principen påminner om metod för parameteröverföring via stack men här måste vi återställa stacken *i subrutinen*.

EXEMPEL

Betrakta följande definition av en C-funktion:

```
void  dummyfunc(void)
{
      int   la,lb;
      la = 1;
      lb = 3;
}
```

Vi *allokerar* utrymme för variablerna genom att *subtrahera* deras totala storlek från stackpekaren:

```
      SUBQ.L      #(4+4),SP
```

Vilken *offset* vi nu väljer för respektive lokal variabel är godtyckligt men det är lämpligt att vi alltid använder samma metod, exempelvis börja med den först deklararerade variabeln, osv. I så fall får vi stackoffset 0 för la och stackoffset 4 för lb.

Hela subrutinen ska då kodas:

```
dummyfunc:
      SUBQ.L      #8,sp
```

```
        MOVE.L      #1,(0,SP)
        MOVE.L      #3,(4,SP)
        ADDQ.L      #8,SP
        RTS
```

Det är förstås mycket viktigt att SP har *samma värde* då RTS-instruktionen utförs som vid ingången av subrutinen. Om inte, kommer fel återhoppsadress att placeras i PC och programmet "spårar ur".

5.62

5.3.16 Positionsoberoende kod

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

```
        ORG         $4000
main:
        NOP                     4E71
        JMP         main        4EF9 0000 4000
```

Den *absoluta* adressen finns i instruktionen.

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

Betrakta nu följande kod i stället, observera att *funktionen* är den samma:

```
        ORG         $4000
main:
        NOP                     4E71
        BRA         main        6000 FFFC
```

Adressen anges som en offset till programräknaren (PC-relativ)

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

Det finns ytterligare exempel på hur olika instruktioner kan väljas då man vill att programkoden ska vara positionsoberoende, dessutom finns de speciella PC-relativa adresseringsmoderna som är användbara då man vill konstruera positionsoberoende kod.

Egenskaper hos positionsoberoende kod kan speciellt utnyttjas av *operativsystem*, dessa kan flytta sådana program i primärminnet utan att modifiera koden. Det kan också vara användbart då man utvecklar nya datorsystem där man använder flera processorer och de olika processorerna använder olika adressavkodning. Vi återkommer till detta i senare avsnitt.

5.3.17 Exception hantering

Processorn MC68000 befinner sig alltid i något av följande tillstånd:

- *normal*
- *exception*
- *halt*

Så här långt har vi förutsatt att processorn befunnit sig i *normal* tillståndet, dvs, processorn hämtar och utför kontinuerligt nya instruktioner. Det finns emellertid en rad händelser som gör fortsatt instruktions-hämtning /exekvering omöjlig.

I det följande visas några exempel på *exceptions* (undantag)

Instruktionen:

```
DIVS        D0,D1
```

dividerar innehållet i **D1** med innehållet i **D0** och placerar resultatet i **D1**. Men om innehållet i **D0** är 0, kan instruktionen inte utföras ty operationen är odefinierad.

Adress \$00 00 00 14 i minnet hos ett MC68000-baserat mikrodatorsystem innehåller adressen till den rutin som skall utföras om processorn någon gång försöker utföra division med 0. Adress \$00 00 00 14 är exception-vector för Zero-Divide.

En instruktion försöker läsa data från en ogiltig adress, dvs en adress där det inte finns något fysiskt minne i systemet, förmodligen som följd av ett programmeringsfel. Hårdvaran rapporterar minnesfel eftersom ingen DATA ACKNOWLEDGE (DTACK) signal fås från minnet. Instruktionen kan därför inte utföras.

Som resultat av ett programfel har processorn hämtat "nästa instruktion" från en minnesarea som i själva verket innehåller data. Det inhämtade ordet utgör ingen operationskod för MC68000 som alltså inte vare sig kan avkoda eller utföra instruktionen.

Exemplen ovan visar bara några av de *undantags*-situationer som kan uppstå vid exekveringen av ett program. Situationer där processorn inte kan fullfölja utförandet av en instruktion kallas med ett gemensamt namn *exceptions*.

För att kunna skriva stabil programvara måste så många sådana här undantagssituationer som möjligt förutses. Dessutom måste programmeraren tillhandahålla någon form av *undantags-hantering* (*exception handling*), dvs programkod som fångar upp undantagen och behandlar dem på lämpligt sätt. I MC68000 har man förutsett en rad *exceptions* och tillhandahåller så kallade *exception-vectors* för olika typer av undantag. En exception-vector är en fördefinierad adress i datorns minne. För varje typ av exception finns precis en adress och *innehållet på denna adress utgör startadressen för undantagshanteringen*.

Längre fram i detta avsnitt ska vi återkomma till *vad* man lämpligen bör vidta för åtgärder vid olika typer av exceptions, dvs utformningen av själva exception-rutinerna, men låt oss först ge en översikt av de olika typer av exceptions som kan förekomma. Motorola delar upp samtliga typer av exceptions i två grupper: *traps* och *interrupts* (avbrott).

Med ett *avbrott* avses en händelse som genererats utanför processorn och som kräver en omedelbar hantering. Typiskt innebär detta en periferienhet som är klar att överföra data till eller från processorn. Med en *trap* avses en intern händelse, dvs ingen periferikrets är inblandad och ingen speciell avbrotts-signal är aktiverad hos processorn. I ett MC68000-system är adresserna 0-$3FF reserverade för olika exception-vektorer. Denna minnesarea innehåller såväl vektorer för trap-hantering som vektorer för avbrottshantering. Adresserna 0-$3FF kallas *Exception Vector Table* som är organiserad på följande sätt:

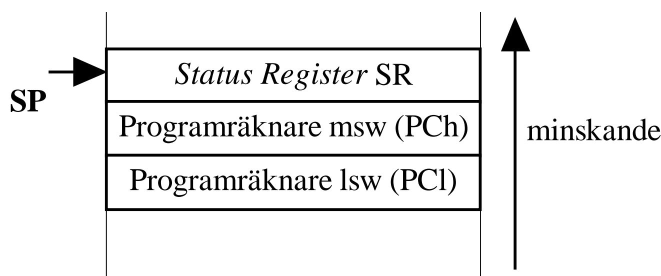
Vektor nr	Adress (hex)	Funktion
0	000	Initial stackpekare
1	004	Initial programräknare
2	008	Bus Error (ex: referens till adress där minne/periferi ej finns)
3	00C	Adress Error (ex: referens till udda adress med *word* operand)
4	010	Illegal instruktion (icke-definierad operationskod)
5	014	Division med 0
6	018	Trap vektor för instruktionen CHK
7	01C	Trap-vektor för instruktionen TRAPV
8	020	Privilege Violation, försök att utföra *supervisor*-instruktion i *user mode*
9	024	Trace, en-instruktions exekvering
10	028	Line 1010, reserverad operationskod
11	02C	Line 1111, reserverad operationskod
12	030	Reserverad för framtida bruk
13	034	Reserverad för framtida bruk
14	038	Reserverad för framtida bruk
15	03C	Avbrott från enhet som ej tillhandahållit avbrottsnummer
16-23	040-05F	Reserverade vektorer
24	060	Icke-identifierat avbrott
25	064	Autovektor avbrottsnivå 1
26	068	Autovektor avbrottsnivå 2
27	06C	Autovektor avbrottsnivå 3
28	070	Autovektor avbrottsnivå 4
29	074	Autovektor avbrottsnivå 5
30	078	Autovektor avbrottsnivå 6
31	07C	Autovektor avbrottsnivå 7
32-47	080-0BF	Trap vektor för instruktionen TRAP #<vektor_nummer>
48-63	0C0-0FF	Reserverade vektorer
64-255	100-3FF	Användardefinierade vektorer

Låt oss, innan vi övergår till en detaljerad beskrivning av exceptions, studera den generella exception-hanteringen, dvs vilka moment som utförs av processorn vid ett exception:

Vid Exception:

- Programräknarens innehåll sparas på stacken
- Statusregistrets innehåll sparas på stacken
- Processorn försätts i *supervisor mode*
- Processorn hämtar aktuell exception vektor från *Exception Vector Table*.
- Adressen till exception-hanteringen laddas i programräknaren.

Stackens utseende då exception-hantering rutinen börjar utföras är:



Notera att all exception-hantering utförs av processorn *i supervisor mode*. Observera stackens utseende i den hanteringsrutin som startas vid undantaget. Jämför med stackens utseende efter ett `BSR`-anrop, då endast program-räknarens innehåll sparas på stacken. För att återuppta programexekvering efter det att exception-hantering utförts måste processorn:

- Återställa innehåll i statusregistret från stacken
- Återställa innehåll i programräknaren från stacken.

Observera skillnaden mot `RTS` (*return from subroutine*) som används för subrutiner. För att avsluta exception-hantering används i stället instruktionen `RTE` (*return from exception*).

Den generella hanteringen gäller samtliga exceptions utom *Bus Error* och *Address Error* då ytterligare information sparas på stacken för att hanteringsrutinen ska kunna avgöra vilken adress som genererade felet. Dessa fall behandlas separat nedan.

Vi kan alltså likna den generella exception-hanteringen vid ett subrutin-anrop, dock med den skillnaden att: vid subrutin-anrop sparas endast innehållet i programräknaren på stacken, men vid exception-hantering sparas även innehållet i statusregistret på stacken. För att utnyttja processorns mekanism för exception-hantering måste programmeraren tillhandahålla exception-rutiner.

Varje exception-rutin är som regel unik och utför någon exception-specifik hantering. Startadresser till respektive exception rutin skrivs till motsvarande exception-vector, detta görs endast en gång och på ett tidigt stadium. Observera att ett oinitierat exception, om detta upträder, kan få processorn att ta en obestämd exekveringsväg ("spåra ur") och eventuellt skriva över såväl programkod som data på ett icke önskvärt sätt.

Illegal Instruction-trap

Illegal-instruction trap uppträder vanligtvis då processorn har "spårat ur" dvs programräknaren har av någon anledning (programfel) fått fel värde och pekar på någon data-area i stället för en instruktion. Det finns här inte så mycket man kan göra för att rätta till felet, vi kan nämligen bara avgöra exakt *var* exekveringsfelet uppstod vilket inte nödvändigtvis betyder att detta var den första instruktionen som exekverades fel. En data-area *kan* ju exempelvis mycket väl innehålla data som tolkas som instruktioner och utförs av processorn.

En hanteringsrutin för Illegal Instruction trap bör därför först och främst rapportera felet, exempelvis genom en utskrift till bildskärmen, därefter återuppta exekvering från någon känd (säker) punkt i

programmet. Oftast innebär detta omstart av det inbyggda monitorprogram som systemet har.

Följande kod visar (i princip) hur Illegal-Instruction-trap behandlas av monitor/debuggern'n *db68*.

EXEMPEL

Hantering av ogiltig operationskod

```
* Initiera "Illegal-Instruction-trap handler"
      MOVEA.L  #Illegal_Instruction_Handler,($10).L
      ....
```

Initieringen av *samtliga* exception-vektorer görs i en följd på ett tidigt stadium efter RESET av systemet.
Hanteringsrutinen skriver ut såväl statusregistrets som programräknarens innehåll som sparats på stacken av processorn då undantaget upptäcktes

```
Illegal_Instruction_Handler:
      MOVE.W      (0,SP),D0   SR vid trap
      MOVE.L      (2,SP),A0   PC vid trap
      JSR   print_message_ill_inst
      JMP   db68_restart
```

5.62

Zero Divide-trap

Vid försök till division med 0, kan man tänka sig två olika hanteringssätt. Ett vanligt sätt är att avbryta med ett felmeddelande, hanteringen blir då praktiskt taget identisk med hanteringen vid Illegal-Instruction-Trap. Men man kan också tänka sig att ersätta (det oidentifierade resultatet) med exempelvis det största tal som kan representeras i maskinen. Denna hantering blir något mer omfattande. Med ledning av den instruktion som genererade undantaget kan destinationsoperandens dataregister bestämmas. (Observera att destinationsoperanden alltid är ett data-register för denna instruktion). Därefter han det förutbestämda värdet placeras i detta dataregister. Hanteringen är nu klar och processorn kan återgå till exekveringen av det program där undantaget uppstod.

EXEMPEL

"Tyst" hantering av division med noll

```
* Initiera trap-vektor
      MOVE.L      #Zero_Div_Handler,($14).L
...
...
```

Observera att hanteringsrutinen måste spara innehållen i de register som används för att programexekveringen senare ska kunna fortsätta från den punkt där undantaget uppstod

```
Zero_Divide_Handler:
* Spara register som används,
* stackoffset till PC nedan påverkas
     MOVEM.L      register_lista,-(SP)
     MOVE.L       (2+x,SP),A0        PC till DIV-instruktion
     MOVE.W       (A0),D0            op-kod för DIV till D0
* extrahera det bit-fält som anger vilket dataregister som
* utgör destinationsoperand
* utför         MOVE.L      #NÅGOT_TAL,Dn
*
* Hanteringen nu avklarad, förbered återhopp
* Återställ register som använts
     MOVEM.L      (SP)+,register_lista
     RTE                             Åter till program
```

5.63

CHK Instruction-trap

CHK-instruktionen används för att kontrollera att innehållet i något dataregister är inom ett bestämt intervall. Om detta inte gäller, kommer denna trap att genereras. Hanteringsrutinen är helt avhängig i vilket sammanhang CHK-instruktionen används.

En CHK-hanteringsrutin kan exempelvis sätta en global variabel till 1. Ett program som använder CHK-instruktionen kan därefter undersöka denna flagga för att se om någon undantagshantering uppträtt.

EXEMPEL

Användning av "CHK"-instruktionen

```
     MOVE.L       #Check_Trap,($18).L           init
     CLR.L        (check_trap_detected).L       init
...
...

Check_Trap:
     MOVE.L       #1,(check_trap_detected).L    sätt flagga
     RTE

Ett program kan använda denna hantering, exempelvis enligt:

     CHK          #100,D0            kontrollera att 0 ≤ D0 ≤ 100
     CMPI.L       #1,(check_trap_detected).L
     BEQ          D0_out_of_range
*    här fortsätter exekveringen om värdet i D0 var o.k
```

5.64

TRAPV instruction-trap

TRAPV-undantagshantering kan användas då man önskar en generell hantering av tvåkomplements-spill vid aritmetiska operationer. Man kan tänka sig flera olika sätt att hantera denna situation. Exempelvis kan beräkningen som utförs vara kritisk för fortsättningen av programmet, dvs ett korrekt resultat måste bestämmas om den fortsatta exekveringen överhuvudtaget skall vara meningsfull.

I så fall kan man tänka sig en hantering liknande den vi visade för Illegal-Instruction trap, dvs, hanteringsrutinen skriver ut undantaget, i detta fall Trap-On-Overflow, och fortsätter därefter exekveringen från en känd (säker) punkt. Detta innebär då att exekveringen av det program där undantaget uppstod avbryts.

I många fall kan man emellertid tänka sig att programexekveringen kan fortsätta även om tvåkomplenentspill uppträder. Under sådana omständigheter kan man använda sådan undantagshantering som visades för CHK-instruktionen ovan.

Följande visar ett enkelt hanteringsförfarande för sådana fall där korrekt beräkning inte är kritisk för fortsättningen, men man ändå vill fånga upp situationen i programmet, och ta hänsyn till den, för att kunna fortsätta.

EXEMPEL

```
      MOVE.L      #TrapV_Handler,($01C).L initiera TRAPV-vektor
      CLR.L       (TrapV_occured)         initiera flagga
...
...
*     Exempel på program som använder TRAPV ...
      ADD.L       (A0),D2           utför addition
      TRAPV                         om spill, fånga ...
      CMPI.L      #1,(TrapV_occured).L    kontrollera spill ...
      BEQ         handle_overflow
*     Inget spill, fortsätt
continue:
      ...
      ...
handle_overflow:
*     Här utförs programmets hantering av spill-situationen
      ....
      CLR.L       (TrapV_occured).L       detta spill hanterat
      BRA         continue                fortsätt som vanligt
```

Hanteringsrutinen blir praktiskt taget identisk med CHK-fallet:

```
TrapV_Handler:
      MOVE.L      #1,(TrapV_occured).L
      RTE
```

5.65

Privilege Violation-trap

En *privilege-violation-trap* genereras om en instruktion som är förbehållen *supervisor mode*, försöks exekveras då processorn är i *user mode*. Hanteringen av en sådan situation beror naturligtvis helt och hållet på vilken typ av programsystem man håller på att bygga upp.

I en enkel monitor kanske man helt enkelt ignorerar situationen och ställer om S-biten i processorn statusregister för att därefter återuppta exekveringen vid (den "otillåtna") instruktionen. I större system kan man emellertid inte göra detta efterson det skulle kunna innebära att programmet i fråga manipulerar periferikretsar eller kanske speciella globala data i systemet som bara får ändras av det övervakande operativsystemet.

EXEMPEL

"Tyst" hantering av Privilege violation-trap

```
* initiera exception vektor
      MOVE.L      #P_Viol_Handler,($20).L
      ...
      ...
P_Viol_Handler:
* ettställ S-bit i användarens SR
      ORI.W       #$2000,(0,SP)
      RTE                             tillbaks ...
```

5.66

Trace-trap

Trace-trap används speciellt för att möjliggöra en-instruktions exekvering. I princip går det till så att en stack skapas där man först placerar ett innehåll som är avsett för processorns statusregister, T-biten är här satt till ett. Därefter placerar man adressen till den instruktion man vill exekvera i stacken och slutligen låter man processorn exekvera RTE. Nu utförs en enda instruktion, därefter kommer hanteringsrutinen för *Trace-trap* att startas. Den typiska funktionen hos denna hanteringsrutin är då att skriva ut innehållet i processorns register till bildskärmen och därefter överlämna kontrollen till något monitor-program.

Line 1010 Emulator-trap

Line 1010-emulator-trap kan liknas vid *Illegal-Instruction-trap*. Skillnaden är att här har Motorola tänkt sig ett enkelt sätt att implementera instruktions-emulering, dvs, om man önskar implementera en instruktion som inte finns i MC68000:s instruktionsuppsättning kan man förse denna med en operationskod vars fyra första bitar är 1010, resten av operationskoden kan användas fritt. Hanteringsrutinen ska då utföra någon specifik, egendefinierad, funktion som alltså är helt beroende av den emulering man tänkt sig.

Line 1111 Emulator-trap

Line 1111 Emulator trap, liksom *Line 1010 Emulator-Trap* används för emulering av instruktioner. För *Line 1111* gäller att denna operationskod reserverats för Motorolas co-processorer (flyttal och minneshanterare). Följaktligen kan man bygga upp ett programbibliotek som emulerar exempelvis instruktionsuppsättningen hos flyttals-processorn MC68882 och göra systemet oberoende av om den fysiska flyttals-processorn faktiskt är ansluten till systemet eller ej.

TRAP Instruction Vectors

För att utföra en ovillkorlig, programstyrd *trap*, kan instruktionen

```
        TRAP        #<VECTOR>
```

användas. Det finns sexton olika vektorer (0-15). Typiskt används dessa för att implementera så kallade systemanrop, där ett användarprogram använder sig av program som exempelvis finns i den inbyggda monitor/debuggern för att utföra någon funktion. Sådana systemanrop kan exempelvis vara in-/utmatning från/till någon periferikrets. Det kan vara mer komplexa funktioner som till exempel att skriva/läsa datablock till/från en flexskiveenhet. I db68 används dessutom instruktionen `TRAP #<15>` som brytpunkt. Den tillhörande hanteringsrutinen skriver ut processorns registerinnehåll till bildskärmen och överlämnar därefter kontrollen till db68.

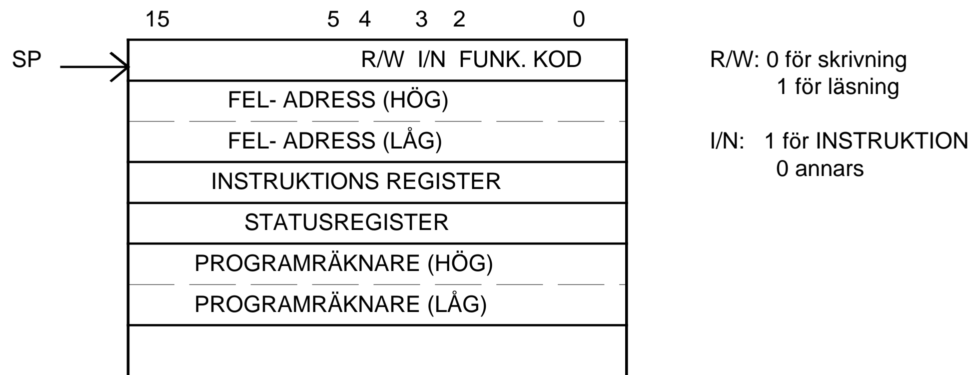
Bus Error-trap

Bus Error-trap är resultatet av en otillåten adressering av minnet. Det vanligaste exemplet är ett program som "spårat ur" av någon anledning och plötsligt genereras en adress till något obefintligt minne. Eftersom minnessystemet i datorn inte genererar en så kallad DTACK inom föreskriven tid (128 processorcykler), avbryts instruktionen i fråga, Bus-Error vektorn hämtas från Exception Vector Table och programexekveringen fortsätter i hanteringsrutinen.

Address Error

Address Error-trap genereras då processorn instrueras att göras en minnes-åtkomst som inte är möjlig. Typiskt innebär detta att processorn instrueras att läsa eller skriva ett *word*, eller en *long* på en udda adress.

För såväl *Bus Error-trap* som *Address Error trap* gäller att stackens utseende avviker från den generella exception-hanteringens stackutseende. Vid Bus/Adress-Error har stacken följande utseende:



Som framgår av figuren sparas avsevärt mer information vid Bus Error och Address Error. Hanteringsrutinen kan använda denna information för en utförligare felutskrift som bättre kan leda programmeraren vid felsökning. Observera att det aldrig kan vara aktuellt att återgå till exekvering av ett program som genererat Bus- eller Address Error eftersom detta bara skulle leda till en oändlig slinga av utskrifter från hanteringsrutinen. Hanteringsrutinen ska därför avsluta med att överlämna kontrollen till det övervakande programmet.

Avbrottshantering

Avbrottshantering i MC68000-baserade system kan på flera sätt jämföras med trap-hanteringen. På motsvarande sätt kan man här reservera olika vektorer för olika avbrottskällor i systemet. Det finns två principiellt olika sätt att anvisa exception-vektorer för avbrott (eller kortare: avbrottsvektorer). Man kan antingen använda *vectored interrupt* eller *autovector interrupt*.

Observera att den metod som används är förberedd i hårdvaran, oftast via någon möjlighet till bygling på datorkortet. Det finns inget sätt att programmässigt styra vilken metod som används av systemet för att generera avbrottsvektorer, däremot är det som regel möjligt att kombinera de olika metoderna i ett och samma system genom att exempelvis använda några prioritetsnivåer för autovector interrupt och de resterande prioritetsnivåerna för vectored interrupt. Principer för autovector interrupt respektive vectored interrupt beskrivs i kapitel 7.

5.4 Kombinerad programmering

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

Programspråket *C* skapades av Dennis Ritchie (Bell Laboratories) i början av 1970-talet. Även om *C* är ett generellt användbart språk har det traditionellt använts som systemprogramspråk. Speciellt är operativsystemet *UNIX* skrivet i programspråket *C*. Under senare delen av 1980-talet och fram till i dag har användningen av *C* ökat närmast explosionsartat. *C*-kompilatorer finns till praktiskt taget alla operativsystem, varianter av *C* (*C++, objektorienterad C*) har utvecklats och, inte minst viktigt, språket har standardiserats.

Den ursprungliga versionen av *C* blev snabbt populär, skälen till detta var flera: *C* tillhandahåller programkonstruktioner som gör det enkelt att implementera algoritmer på ett effektivt sätt. Alla vanliga datatyper finns representerade såväl som pekare och strängar. Det finns en rikhaltig uppsättning operatorer och ett "standard I/O" (input/output) bibliotek som täcker in- och utmatning till filer och terminaler. *C*-program är "effektiva", *C*-operatorer och programflödes-konstruktioner är nära relaterade till instruktioner som tillhandahålls av flertalet processorer. Ett annat sätt att uttrycka det: Det *semantiska gapet* mellan *C* och datorns hårdvara är litet. *C* skapade stora möjligheter att skriva portabla program, dvs applikationer som enkelt kunde kopieras till nya system.

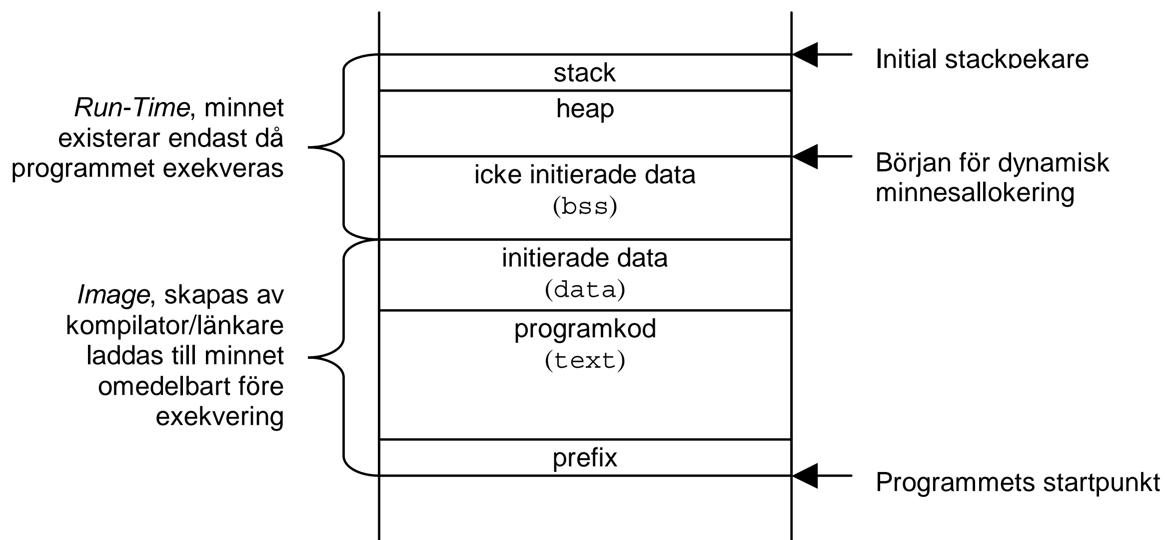
Populariteten hos *C* innebar dock att områden som inte hade beaktats av Kernighan/Ritchie blottlades, dvs brister hos språket identifierades och åtgärdades, ofta lokalt. Som en direkt följd skapades flera olika "dialekter" av språket. Utvecklingen *av UNIX System V* (AT and T) respektive *Berkeley UNIX*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

5.4.1 X68c – korskompilator

X68c är en utökad *KR/C* korskompilator för mikroprocessorer ur MC68000-familjen. *X68c* utgår från den ursprungliga definitionen men implementerar även flera ANSI-C-specifika egenskaper. I själva verket består *X68c* av flera delar: En *preprocessor* som hanterar alla preprocessor-direktiv i C, en *kodgenerator* som kontrollerar syntaxen i C-programmet och genererar assemblerkod för MC68000, en *assembler* som assemblerar koden och genererar relokerbar kod i objektfiler och slutligen en *länkare* som kombinerar flera olika objektfiler till en och skapar en laddfil som kan laddas i en MC68000-baserad dator. Under detta moment *relokeras* koden, dvs alla symboliska namn ersätts med med absoluta adresser i måldatorns minne.

5.4.2 Minnesdisposition

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



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

prefix

Prefixet, eller som det också kallas, *startup*, placeras alltid först. Detta är nödvändigt för att programmet ska ha en känd startpunkt. Den enklaste formen av prefix utgörs helt enkelt av ett subrutinanrop:

```
JSR   (_main).L
```

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

finns tillsammans med kompilatorn. Prefixet länkas automatiskt först i varje C-program av kompilatorn.

programkod

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

initierade data

Deklarationer som

```
int a = 2;

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

osv kan användas för globala variabler. Innehållet är definierat från start, men kan komma att ändras under exekvering. Kompilatorn måste göra en "image" av detta segment för att dessa initialvärden ska kunna laddas till minnet före exekvering.

EXEMPEL

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

Satsen

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

ger liknande resultat på bildskärmen som:

```
char reftext[]={"Denna text ..."};

      printf("%s", reftext);
```

dvs en textsträng skrivs ut.

Kompilatorn betraktar dock textsträngarna på helt olika sätt. I det första fallet är det en konstant sträng, som inte kan refereras av programmet från någon annan punkt än just i printf-satsen. Eftersom den inte kan refereras kan den heller inte ändras, textsträngen är därför *read-only*, och placeras i **text**-segmentet (just det..trots att det inte är programkod).

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

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

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

5.67

oinitierade data

Deklarationer som:

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

osv, kan användas för globala variabler. Eftersom variablerna inte har något definierat innehåll från start behöver kompilatorn bara hålla reda på var, i minnet dessa hamnar. Det behövs alltså ingen "image".

stack

Stacken används av program under exekvering. Storleken hos detta segment bestäms som regel av operativsystemet.

HEAP

"Heapen" benämns ofta det minnesutrymme som reserverats för programmets dynamisk minneshantering `malloc()`, `free()` etc. Även storleken av detta segment bestäms som regel av operativsystemet

Låt oss sammanfatta detta. Vid kompilering av en källtextfil skapas en objektmodul med följande information/innehåll:

- `text`-segment innehållande en "image" av programkoden
- `data`-segment innehållande en "image" av initierade data
- storleken av `bss`-segmentet
- Symboltabell innehållande alla globala symbolers relativa adresser (offset till segmentets början) i respektive segment. Observera att alla symboler är relokerbara, dvs absoluta adresser har ännu ej bestämts.

Då programmet ska exekveras utförs följande:

1. Prefix adderas till textsegmentet
2. Minnesbehov för segmenten text, data och bss bestäms
3. Segmenten relokeras med hjälp av symboltabellen
4. Minnesbehov för stack och heap bestäms (av operativsystemet)
5. Totala minnesbehovet är nu känt och tillräckligt primärminne kan reserveras för programmet.
6. Programmets initierade segment ("images") kopieras till sin respektive plats i primärminnet.
7. Stackpekare initieras och programmet startas (i prefix).

Observera speciellt hur förfarandet förutsätter att denna procedur upprepas inför varje exekvering av programmet. Då man arbetar i en kors-utvecklingsmiljö, som med X68c, har man som regel inget operativsystem utan bara en enkel debugger i målsystemet. Detta innebär att moment som normalt utförs enligt någon strategi bestämd av operativsystemet, nu måste utföras manuellt. Följande punkter är speciellt viktigt att iaktta:

- Stackpekare måste initieras (eventuellt görs detta av debuggern)
- Det finns ingen dynamisk minneshantering tillgänglig
- Programmet måste laddas, från utvecklingssystem till måldatorsystem mellan varje exekvering, oavsett om det har ändrats eller ej.

5.4.3 X68c, minnesallokering

I detta avsnitt skall vi studera hur *X68x* genererar kod för några enkla variabeldeklarationer i **C**.

Globala variabler, minnesallokering

Betrakta följande **C**-program bestående enbart av deklarationer av globala variabler:

```
short shortint;
long        longint;
int         justint;
int         intvec[10];

struct {
      int         s1;
      char        s2;
      char*       s3;
} komplex;
```

```
      text
      bss
      align
      define _shortint
_shortint:
      ds.b   2
      align
      define _longint
_longint:
      ds.b   4
      align
      define _justint
_justint:
      ds.b   4
      align
      define _intvec
_intvec:
      ds.b   40
      align
      define _komplex
_komplex:
      ds.b   10
```

Koden kompileras nu till assemblerkod, den genererade filen får extension <filnamn>.S. Se

marginalen, först genereras några inledande rader med text där *X68x* skriver ut aktuell version av kompilatorn (visas ej i figuren). Dessa rader ignoreras av assemblatorn eftersom de inleds med '`*`'. Därefter genereras direktiv för assemblern:

```
text
```

`text` läggs alltid ut som inledningsdirektiv eftersom kompilatorn normalt förväntar *program-kod*. I detta fall inleds dock texten med deklarationer av globala variabler. Dessa ska placeras i **BSS** och därför följer omedelbart `bss`.

Eftersom variablerna är deklarerade globala sker alla referenser till dem med *namn* så att *samtliga* referenser (även från andra källtextfiler) kan lösas upp vid den slutliga länkningen. Det är ju normalt först då som alla globala variabler är kända. Direktivet:

```
define     _shortint
```

innebär att variabelnamnet får ett *globalt scope*. Observera också den inledande understrykningen. Alla globala namn, såväl funktioner som variabler förses med detta av kompilatorn. På så sätt undviks namnkonflikter mellan exempelvis rutiner som skrivs i assembler och placeras i programbibliotek, och funktioner som definierats i ett C-program.

`shortint` är en variabel av typen `short`.

ANSI-C definitionen av denna typ är:

> "Typen är synonym för: `short int, signed short` och `signed short int`. Det är ett heltal med tecken som kan representeras med 16 bitar."

Efter label:n `_shortint` har *X68c* placerat direktivet

```
ds.b 2,
```

vilket alltså reserverar 2 bytes för variabeln.

På motsvarande sätt ser vi att variabeln `_longint` av typen `long` tilldelas 4 bytes i segmentet och att variabeln `_justint` av typen `int` även den tilldelas 4 bytes. För *X68c* gäller alltså att typen `long` och typen `int` är likvärdiga. Observera dock att detta inte gäller generellt för **C**. Datatypen `int` kan implementeras som 2-bytes i vissa system. Detta är exempelvis fallet för MS-DOS och IBM-PC. Konsekvensen av detta är att man bör tänka sig för vid deklaration av `int`-typer, dvs om man har anledning att tro att variabeln i fråga kan tänkas tilldelas större värden än vad som ryms i en `short` så ska den deklareras som `long`. Denna tumregel måste följas som man vill skapa *portabla* program, dvs **C**-källkod som kan kompileras om i olika miljöer, av olika maskiner, och ändå fungera som avsett.

Variabeln `intvec` är en vektor bestående av 10 komponenter, var och en av typen `int`. Följaktligen tilldelas variabeln `_intvec` 10*4 = 40 bytes minnesutrymme.

Variabeln `komplex` är en `struct`, dvs en datatyp som är komponerad av flera grundläggande typer. Här beräknar kompilatorn det sammanlagda minnesbehovet för variabeln och genererar därefter ett direktiv som åstadkommer detta. `_komplex` består av en variabel `s1` av typen `int`, en variabel `s2` av typen `char` och en variabel `s3` av typen: *pekare till* `char`.

Vi ser att minnesbehovet bestämts till 10 bytes och kan enkelt kolla detta: En datatyp `int` kräver 4 bytes, en datatyp `char` kräver 1 byte, en pekartyp (ovidkommande vad den pekar på) kräver 4 bytes. Det totala behovet är alltså 9 bytes vilket *inte* överensstämmer med direktivet:

```
_komplex:
      ds.b  10
```

Förklaringen ligger i att den aktuella processorn, MC68000, inte kan referera en variabel av pekartyp på udda adress. Följaktligen måste kompilatorn försäkra sig om att `s3` hamnar på jämn adress. Enda sättet att göra detta är att fylla ut med en (icke använd) byte efter variabeln `s2` (som ju bara kräver 1 byte).

5.4.4 Lokala variabler, minnesallokering

Utrymme för lokala variabler allokeras annorlunda. Dessa variabler ska bara finnas under exekvering av den funktion i vilken de deklarerats och det gör det onödigt att placera dem i ett **BSS**-segment eftersom de bara ska vara åtkomliga under en begränsad tid i exekveringen av programmet. Därför allokeras utrymme för lokala variabler på stacken. Då rutinen exekverats färdigt återställs stacken och minnesutrymmet för dessa variabler kan återanvändas under den fortsatta exekveringen.

Samma deklarationer som användes ovan placeras nu i en funktion "main" enligt följande. Programmet kompileras och kod genereras (se figur).

```
main()
{
      short  shortint;
      long         longint;
      int          justint;
      int          intvec[10];
      struct {
            int          s1;
            char         s2;
            char         *s3;
      } typen;
}
```

```
      text
      align
      define
      _main
_main:
      link  a6,#-60
L1:
      unlk  a6
      rts
```

Direktivet `text` instruerar assemblern att det nu kommer programkod. Direktivet `align` instruerar assemblatorn om att nästa byte ska placeras på en jämn adress eftersom MC68000 inte kan läsa en operationskod på udda adress.

Raden:

```
define      _main
```

säger att symbolen `_main` skall föras in som global i symboltabellen. Detta är nödvändigt eftersom alla funktioner i **C** är globala, dvs kan anropas från moduler i andra källkodsfiler.

Därefter inleds det vi känner igen som assemblerkod för MC68000, nämligen ett lägesnamn. I det här fallet `_main` som vi deklarerade som funktion. Direkt efter lägesnamnet följer instruktionen

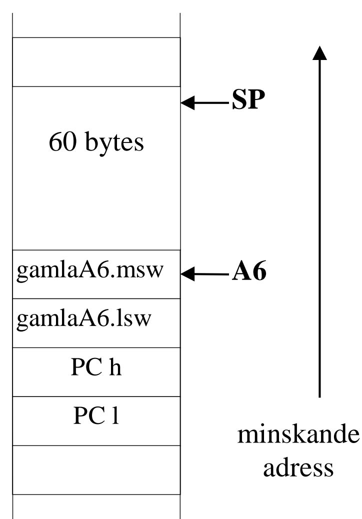
```
link        a6,#-60
```

dvs:

- **A6** placeras på stacken
- stackpekaren kopieras till **A6**
- stackpekaren minskas med 60 bytes.

Följaktligen har vi:

- sparat gamla innehållet i **A6**.
- placerat en ny pekare i **A6**
- reserverat 60 bytes på stacken



Det totala minnesbehovet för de deklarerade variablerna är alltså 60 bytes vilket vi enkelt kan kontrollera genom att jämföra med föregående exempel.

Variablerna refereras därefter genom att ange en offset relativt till adressregistret **A6**, denna offset är alltså *alltid negativ* eftersom stacken "växer nedåt" i minnet (se figur) och stackpekaren efter instruktionen således har ett mindre värde än innehållet i **A6**.

Vid utträde ur funktionen återställs stacken med instruktionen:

```
unlk        a6
```

dvs:

- **A6** kopieras till stackpekaren, 60 bytes deallokeras
- **A6** återställs från stacktopppen

Såväl **A6** som **SP** (stacken) är därefter återställd till det värde den hade före `link`-instruktionen. Överst på stacken ligger nu returadressen för anropet till rutinen.

Observera att det sätt *X68c* använder register **A6** på innebär att detta register inte får modifieras av någon annan kod, exempelvis assemblerkod som länkas med programmet. En vanlig benämning på ett register som används på detta sätt är *frame-pointer*. Den del av stacken som pekas ut av **A6** kallas *aktiveringspost*.

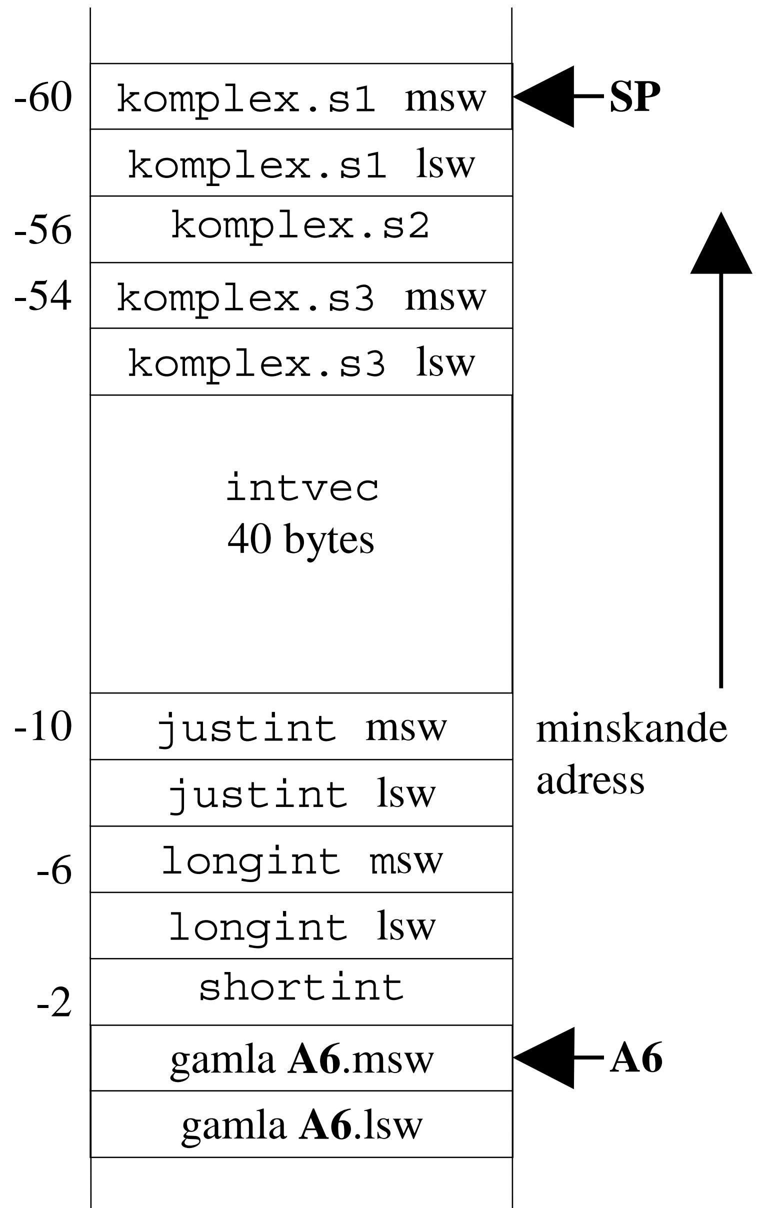
Kompilatorn måste självfallet hålla ordning på var någonstans i aktiveringsposten respektive lokal variabel är placerad. Detta har ingenting att göra med `link` instruktionen som bara skapar aktiveringsposten.

Betrakta följande exempel som visar hur de lokala variablerna refereras:

```
main()
{
      ....
      shortint = 1;
      longint = 2;
      justint = 3;
      komplex.s1 = 4;
      komplex.s2 = 5;
      komplex.s3 = (char *) 6;
}
```

```
      text
      align
      define      _main
_main:
      link  a6,#-60
      move.w      #1,(-2,a6)
      move.l      #2,(-6,a6)
      move.l      #3,(-10,a6)
      move.l      #4,(-60,a6)
      move.b      #5,(-56,a6)
      move.l      #6,(-54,a6)
_1:
      unlk  a6
      rts
```

*Figuren visar hur aktiveringsposten skapats på stacken av link, dvs var **A6** respektive **SP** pekar. Av figuren framgår också hur de lokala variablerna förhåller sig till **A6**.*



5.4.5 Parameteröverföring

Parameteröverföring har tidigare behandlats i generella ordalag. Detta avsnitt handlar om hur *X68c* översätter funktionsanrop och hur rutiner skrivs i assemblerspråk för att fungera tillsammans med rutiner skrivna i **C**.

Tidigare har vi visat hur variabler, såväl globala som lokala hanteras. När det gäller överföring av parametrar kan detta liknas vid lokala variabler, dvs deras "livslängd" begränsas av den tid (under exekvering) som den anropade funktionen använder sig av dom. Parametrar överförs via stacken och det gäller för den anropade funktionen (subrutinen) att korrekt referera sina parametrar.

Allmänt gäller för *X68c* att listan av parametrar i ett funktionsansrop behandlas "bakifrån". Betrakta följande exempel på funktionsanrop:

```
int   a,b;
main()
{
      callfunc( a,b );
}
```

X68c genererar följande kod:

```
      text
      bss
      align
      define     _a
_a:
      ds.b       4
      align
      define     _b
_b:
      ds.b       4
      text
      align
      define     _main
_main:
      link       a6,#0
      move.l     (_b).l,-(a7)
      move.l     (_a).l,-(a7)
      jsr        (_callfunc).l
      addq.l     #8,a7
_1:
      unlk  a6
      rts
```

Vi ser hur kompilatorn genererar kod för att:

- placera värdet av variabeln "b" på stacken
- placera värdet av variabeln "a" på stacken
- utför anropet av funktionen "callfunc"
- adderar 8 bytes till stackpekaren, dvs återställer denna

Vid `jsr` läggs återhoppsadressen (4 bytes) på stacken och i subrutinen `_callfunc` kan vi enkelt bestämma den offset från stackpekaren som gäller för de överförda parametrarna.

Denna offset kombineras med den offset som genereras av eventuella lokala variabler, det vill säga: referens till parametrar kan sker via **A6** som *positiv* offset, eftersom de placerats på stacken före subrutinanrop och allokering för lokala variabler.

Vi exemplifierar detta genom att låta kompilatorn generera kod för "dummy"-funktionen `callfunc`.

```
callfunc( x , y )
{
      x = 1;
      y = 2;
}
```

X68c genererar följande kod:

```
      text
      align
      define      _callfunc
_callfunc:
      link        a6,#0
      move.l      #1,(8,a6)
      move.l      #2,(12,a6)
L1:
      unlk        a6
      rts
```

Observera hur parametrarna refereras via **A6**.

Om vi hade skrivit rutinen i assemblerspråk hade vi antagligen refererat parametrarna direkt via stackpekaren. Eftersom vi inte har några lokala variabler är ju egentligen `link/unlink` konstruktionen onödig och koden skulle då (bättre) ha skrivits enligt föjande:

```
callfunc:
      move.l #1,(4,sp)
      move.l #2,(8,sp)
      rts
```

EXEMPEL

Antag att `func_1` deklareras enligt:

```
func_1( x )
 int  x;
 {
 int  a,b;

  ......
 }
```

Visa med assemblerinstruktioner hur tilldelningarna:

```
      a=x;
      b=a;
```

sker i `func_1`.

Lösning:
Rita aktiveringsposten i `func_1`

-8	b (4 bytes)
-4	a (4 bytes)
0	**A6** (4 bytes)
+4	**PC** (4 bytes)
+8	x (4 bytes)

Referenserna blir då:

```
a=x; ⇒ move.l     (8,a6),(-4,a6)
b=a; ⇒ move.l     (-4,a6),(-8,a6)
```

5.68

5.4.6 Maskinnära programmering i C

Låt oss avsluta detta kapitel med att illustrera hur man faktiskt kan utföra viss maskinnära programmering direkt i *C*.

Absolut adressering

Portadresser i minnet kan enkelt adresseras. Man tillämpar bara en typkonvertering på konstanten, minns att typdeklarationer "läses bakifrån" och betrakta följande:

```
*((char *)Portadress)
```

- `Portadress`, är den fysiska adressen (en konstant)
- `(char *)` anger att `Portadress` är adressen till en 8-bitars port, om det är en 16-bitars port använder vi konverteringen `(short int *)`, för en 32-bitars port används `(long int *)`.
- Slutligen, `*` framför yttersta parentesen anger helt enkelt att det är innehållet på denna adress som avses.

Betrakta nu följande exempel:

EXEMPEL

Antag att vi har ett 8-bitars register på adress $8B00. Antag vidare att registret har dubbla funktioner, vid skrivning fungerar det som ett styrregister, vid läsning fungerar det som ett statusregister.

Vi definierar symboler för portarnas adresser:
/* Port definitioner */
```
#define Status         0x8B000
#define Control        0x8B000
```

Nästa steg blir att definiera "macros" för läsning respektive skrivning:

```
#define read_control   *((char *) Status)
#define set_control(x) *((char *) Control)=x
```

Antag nu vidare att bit 0 i statusregistret indikerar signalen från en givare (sensor) och att bit 0 i styrregistret är en signal till en motor som öppnar en dörr, det ger oss följande definitioner:

```
#define    SENSOR      0x01
#define    OPEN_DOOR   0x01
```

Vi kan nu använda våra macros till att läsa från respektive skriva till portarna, exempelvis på följande sätt:

```
if( read_control & SENSOR )
      set_control(OPEN_DOOR);
```

5.69

IO-ENHETER OCH ADRESS-AVKODNING

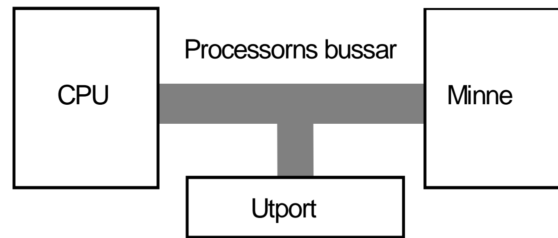
I detta kapitel kommer vi att diskutera hur input/output (I/O) fungerar och på vilka olika sätt detta kan utföras. Vi studerar parallell och seriell överföring mellan ett mikrodatorsystem och omvärlden och diskuterar problem som uppstår när en snabb mikrodator kommunicerar med en förhållandevis långsam yttre enhet. Vi diskuterar också några olika typer av datanät där vi bland annat ser hur det kan vara bättre att skicka data per bil eller tåg mellan Göteborg och Stockholm snarare än att utnyttja datanät.

För att kunna ansluta en en I/O-port till en mikroprocessor måste det finnas speciell, så kallad, "adressavkodningslogik". Kapitlet inledes därför med en studie över hur minneskretsar och I/O-portar tilldelas speciella adresser i mikroprocessorns adressrum.

Kapitlet är grundläggande för bokens återstående delar som bland annat kommer att behandla "avbrott" respektive "periferikretsar".

6.1 Adressavkodning

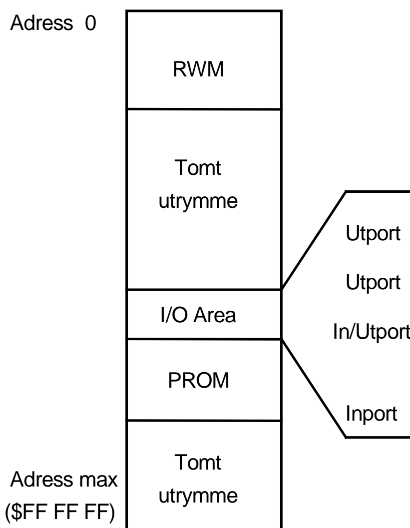
I de inledande kapitlen beskrev vi hur processorn kommunicerar med minnet via bussarna (jämför figur 6.1), vi visade hur en busscykel går till och hur minnet är uppbyggt.



FIGUR 6.1 ETT DATORSYSTEM MED EN UTPORT.

Figuren antyder att *hela* det tillgängliga adressrummet användes, dvs för alla adresser som processorn kan generera finns det ett minnesregister. Man säger då att hela processorns adressrum är bestyckat. Det är alltså adressbussens bredd som bestämmer hur stort primärminne som kan anslutas. MC68000 har exempelvis en 23 bitars adressbuss vilket innebär att den kan adressera 2^{23} = 8 Mword = 16 Mbyte, medan moderna mikroprocessorer har ett avsevärt större adressrum. I själva verket används inte hela processorns adressrum för fysiskt minne, vanligtvis delas adressrummet upp i olika adressintervall för de olika typer av kretsar som ska anslutas i systemet:

- *ROM-area*, avsett för icke-flyktiga minnen
- *RWM-area*, skriv- och läsbart minne
- *IO-area*, här finns kretsar som är avsedda för systemets kommunikation med omvärlden, eller mer korrekt, i princip alla kretsar som inte är av primärminnestyp.



FIGUR 6.2 MINNESLAYOUT

Det är inte vanligt att processorns hela adressrum är utbyggt med kretsar, i stora delar av processorns adressrum finns varken minne eller I/O. Figur 6.2 visar exempel på minnesdisposition (*minnesmap*). När minne och IO blandas, och behandlas (styrs via bussarna) på samma sätt säger man att systemet har *minnesmappad I/O* (*Memory Mapped I/O*). Andra, exempelvis INTELS processorfamilj *80- 2/3/486*, och *Pentium*, behandlar minne och I/O på olika sätt, detta kallas *I/O Mapped*. Här finns speciella handskakningssignaler som aktiveras endast när så kallade I/O-instruktioner exekveras. Vi behandlar här endast minnesmappad I/O vilket innebär att samma instruktioner och samma buss-signaler används för åtkomst av såväl som I/O.

I figur 6.2 visas alltså en möjlig disposition av adressrummet. Observera att RWM är placerat på låga adresser, från adress \$0 och uppåt till exempelvis \$f ff ff. Detta innebär att när processorn antigen hämtar instruktioner (FETCH) eller överför data (MOVE) i detta adressområdet så har adressbussen ett värde mellan \$0 och \$f ff ff (= 1.048.575 = 1 MByte).

Betraktar vi nu hela adressbussen ser vi att vissa av adressbitarna inte ändrar värde oberoende av vilken adress som väljs i RWM:s adressområde, se figur 6.3 nedan.

Adressbussen olika bitar

A23	A22	A21	A20	A19	A18		A1	A0	
0	0	0	0	0	0		0	0	Adress \$0
0	0	0	0	0	0		0	1	Adress \$1
0	0	0	0	0	0		1	0	Adress \$2
0	0	0	0	0	0		1	1	Adress \$3
0	0	0	0	1	1		1	0	Adress \$FFFFE
0	0	0	0	1	1		1	1	Adress \$FFFFF
(0	0	0	1	0	0		0	0	Adress \$100000)

Konstanta | Varierar

En adress utanför aktuell adressarea

FIGUR 6.3 ADRESSBUSSENS BITAR FÖR OLIKA ADRESSER

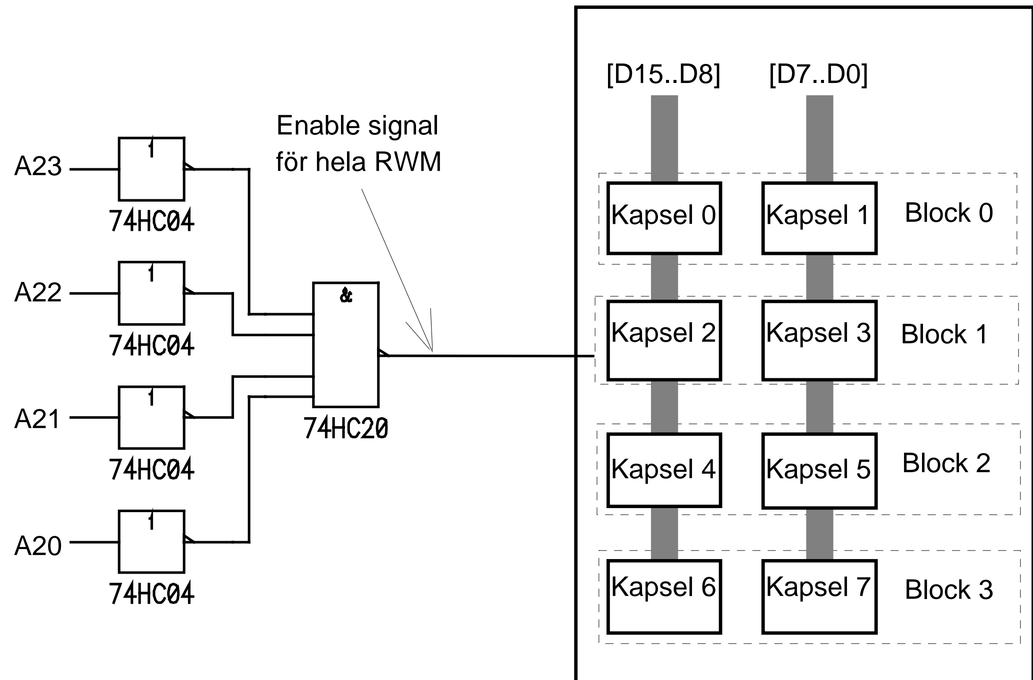
Just detta faktum, att vissa adressbitar är konstanta för hela adressintervallet, utnyttjar vi vid konstruktion av adress-avkodningslogiken. Om vi betraktar figuren ser vi att adressbitarna A23, A22, A21 och A20 samtliga är noll för alla adresser i RWM-minnesarean och vi behöver därför bara avkoda en del av adressbussen [A23..A20] enligt figur 6.4. En adress utanför detta område ska inte generera "Enable-signalen" för RWM. På samma sätt bestäms adressareorna för PROM och IO i systemet.

Efter att ha specificerat dessa areor måste man nu betrakta de speciella kretsar som ska kunna användas inom minnesblocken (RWM och PROM), dvs hur *stora* minneskretsar kan man välja?

I vårt exempel är adressrummet för RWM 512 kword stort (Eftersom 20 adressbitar [A19..A0] tillåts variera inom minnesarean bestäms minnesstorleken enkelt enligt 2^{20} = 1M byte). Antagnu att vi väljer 128k bytes RWM kretsar till vårt system. Vi behöver då 8 RWM-kretsar (8*128k = 1M byte) till systemet, där fyra kretsar är anslutna till

udda adresser [D7..D0] och fyra är anslutna till jämna adresser [D15..D8].

RWM har då indelats i fyra block där varje block består av två byte-minnen som ansluts till den 16-bitars databussen. Vi har tidigare specificerat adressavkodningslogik för hela RWM-arean (se figur 6.4) och nu återstår hur adressavkodningslogiken *internt*, för denna area, skall konstrueras. Studera tabellen nedan och observera att [A19,A18] på adressbussen är konstanta för varje minnes block (0, 1, 2 och 3).



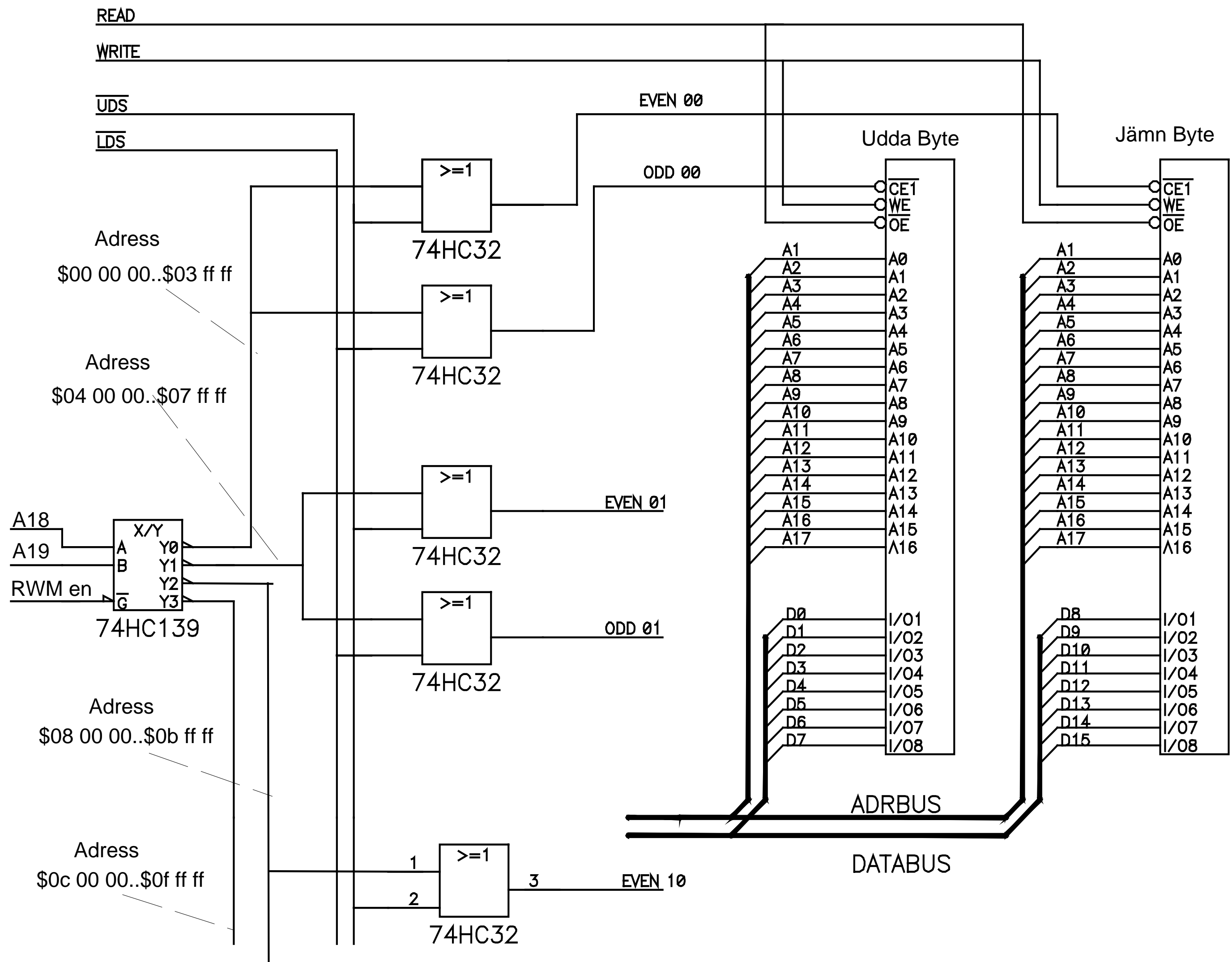
FIGUR 6.4 ADRESSAVKODNING FÖR RWM

Block	Start adress	Slut adress	A19	A18	Minneskrets Jämn adress [D15..D8]	Minneskrets Udda aress [D7..D0]
0	00 00 00	03 FF FF	0	0	Kapsel 0	Kapsel 1
1	04 00 00	07 FF FF	0	1	Kapsel 2	Kapsel 3
2	08 00 00	0B FF FF	1	0	Kapsel 4	Kapsel 5
3	0C 00 00	0F FF FF	1	1	Kapsel 6	Kapsel 7

Figur 6.5 nedan visar hur RWM-arean avkodas, adressbussens [A19,A18] avkodas i 2:4 avkodaren och signalen **RWMen** ("RWM enable") aktiverar avkodaren (denna signale betecknades i figur 6.4 "Enable signal för hela RWM:et"). På detta sätt fås utgångarna från avkodaren [Y0..Y3] att aktivera *var sitt minnesområde* inom RWM-arean.

Processorns signaler **LDS** och **UDS** väljer den högre eller den lägre delen av databussen. Används ordadressering är båda signalerna aktiva, dvs hela databussen (16 bitar) används samtidigt. Adresseras en udda adress är **LDS** aktiv och adresseras en jämn adress är **UDS** aktiv. I de sistnämnda fallen används bara 8 bitar av databussen. **LDS** och **UDS**

grindas med utsignalerna från 2:4-avkodaren för att aktivera udda eller jämn byte.



FIGUR 6.5 ADRESSAVKODNING INOM RWM-AREAN.

På motsvarande sätt avkodas kretsar inom PROM eller I/O-arean. Man bestämmer först vilken typ av kretsar som ska användas och konstruerar därefter den logik som krävs.

6.1.1 Ofullständig adressavkodning

Man talar om *fullständig* och *ofullständig adressavkodning* av processorns adressrum. Vid fullständig adressavkodning aktiveras ett minnesregister vid endast en unik adress, genererad av processorn. Vid ofullständig adressavkodning aktiveras ett minnesregister för flera olika adresser. Observera att två minnesregister aldrig får aktiveras för samma adress. Exemplen på adressavkodningen av RWM arean ovan är utförda som fullständig adressavkodning. Enklast ser man det på att

samtliga adressbussens signaler ingår i adressavkodningen på ett eller annat sätt. [A23..A20] väljer minnesarea, [A19,A18] väljer RWM krets och [A17.. A1] väljer minnesregister internt i RWM kretsen.

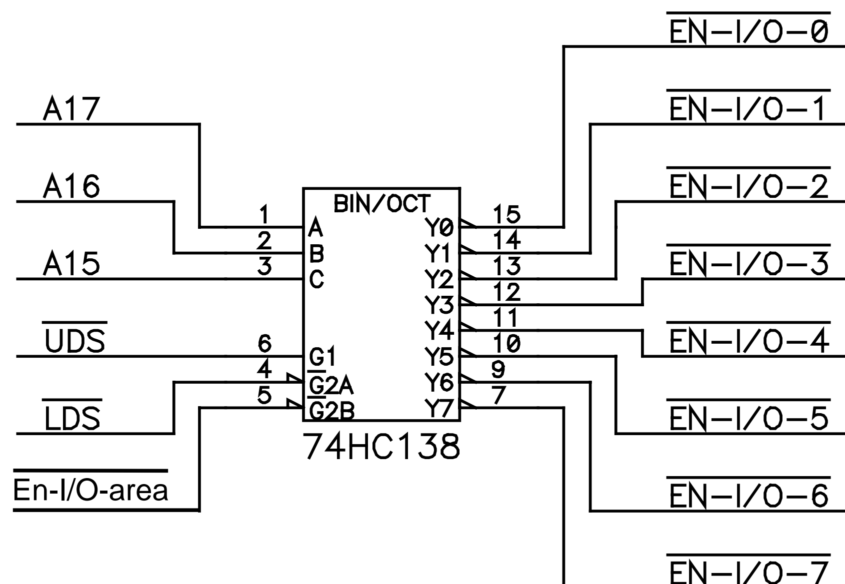
Vi illustrerar ofullständig adressavkodning genom att visa hur IO-arean internt kan avkodas. Antag att vi önskar maximalt 8 olika IO-enheter och att en signal som aktiverar arean (på samma sätt som för RWM-arean), finns tillgänglig. Vi förutsätter vidare denna area är 128k ord stor.

Antag nu att arean valts med start på adress $400000, dvs att signalen "En-I/O-area" är aktiv för adressområdet [$400000..$43ffff]. Inom denna area används då [A17..A15] för adressavkodningslogiken. (Vi önskar 8 st I/O-enheter och därför används tre adressbitar ty $2^3 = 8$). Studera följande tabelle som visar IO-areans olika adresser och respektive "enable"-signaler [EN-I/O-0..EN-I/O-7].

Merparten in- och utkretsar, periferi-kretsar, *som finns på marknaden i dag har 8-bitars dataregister. Detta har en historisk orsak då de första periferikretsarna var anpassade för just 8-bitars system och därför blev också deras dataregister 8 bitar breda. Av prisskäl kan man även i dag välja att konstruera 8-bitars system även om de ingående komponenterna tillåter*

I/O-enhet	Start adress	A17	A16	A15	Signal
0	40 00 00	0	0	0	EN-I/O-0
1	40 80 00	0	0	1	EN-I/O-1
2	41 00 00	0	1	0	EN-I/O-2
3	41 80 00	0	1	1	EN-I/O-3
4	42 00 00	1	0	0	EN-I/O-4
5	42 80 00	1	0	1	EN-I/O-5
6	43 00 00	1	1	0	EN-I/O-6
7	43 80 00	1	1	1	EN-I/O-7

Adressavkodningslogiken konstrueras enklast med en 3:8-avkodare, se figur 6.6. Observera att även **LDS** och **UDS** är anslutna till 3:8 avkodaren, **LDS** skall vara låg och **UDS** ska vara hög för att endat udda adresser ska väljas ([D7..D0]). Vi har här förutsatt användandet av 8-bitars IO-kretsar.



FIGUR 6.6 AVKODNING INOM I/O AREAN

Vi har nu genererat *Enable*-signaler för våra olika IO-kretsar. Antag att IO-kretsen som skall anslutas till signalen EN-I/O-2 är en inport bestående av endast en buffert. Fundera nu vidare på för hur stort adressutrymme som aktiveras för signalen EN-I/O-2. Vi sa tidigare att denna signal var aktiv för hela 16k ord. Detta innebär att alla adresser mellan $41 00 00 och $41 7f ff aktiverar inporten (se figur 6.7). När man anger inportens adress väljer man oftast $41 00 00 som betecknas *basadressen* för porten.

Figur 6.3 och 6.6 angav att [A23..A20] respektive [A17..A15] användes i adressavkodningen för signalerna [EN-I/O-0..EN-I/O-7]. Detta innebär att A18 och A19 inte ingår i avkodningen vilket visas som "x" (don't care) i figur 6.7. Dessa adressbitar kan alltså ha olika värden utan att påverka avkodningen. Exempelvis kommer då signalen EN-I/O-2 att aktiveras för adress $41 00 00, $45 00 00, $49 00 00 och $4D 00 00.

Adressbussen olika bitar

A23	A22	A21	A20	A19	A18	A17	A16	A15	A14..A1	
0	1	0	0	x	x	0	0	0		Adress $40 0000
0	1	0	0	x	x	0	0	1		Adress $40 8000
0	1	0	0	x	x	0	1	0		Adress $41 0000
0	1	0	0	x	x	0	1	1		Adress $41 8000
0	1	0	0	x	x	1	0	0		Adress $42 0000
0	1	0	0	x	x	1	1	1		Adress $43 8000
Konstanta				Varierar						

FIGUR 6.7 ADRESSBUSSENS BITAR FÖR OLIKA ADRESSER

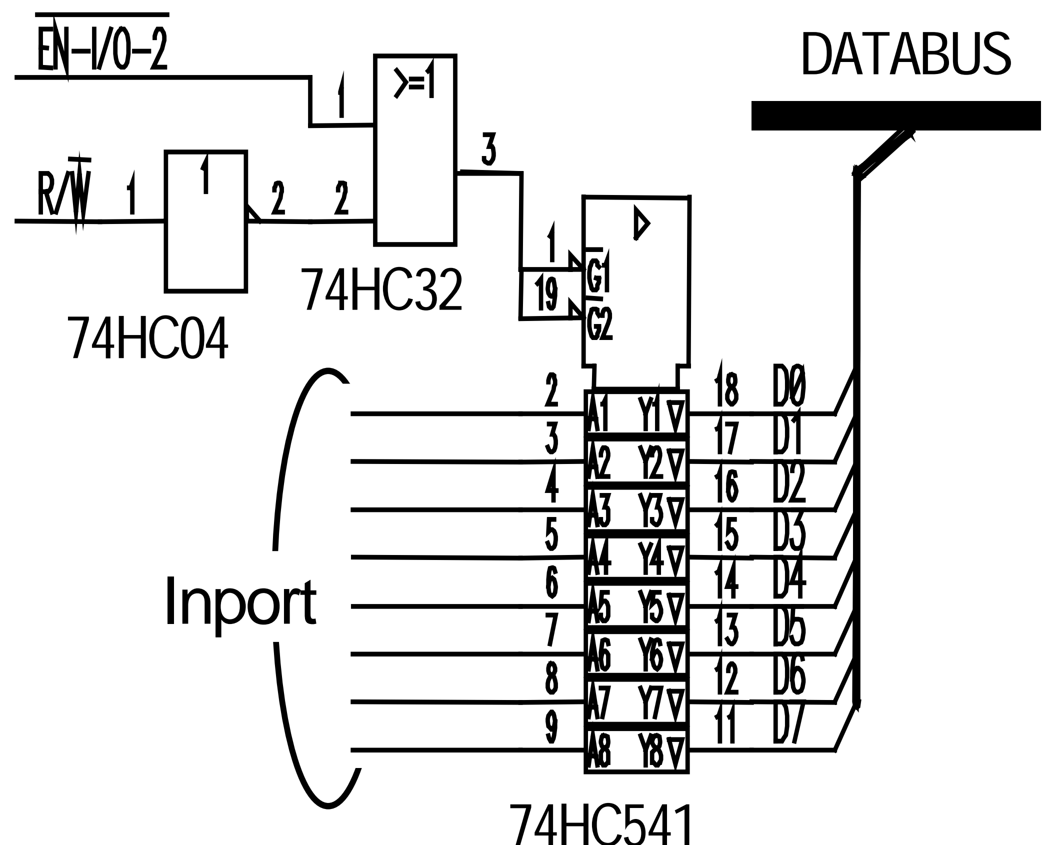
Detta är precis vad ofullständig adressavkodning går ut på, nämligen att förenkla avkodningen och använda mindre kretsar. En effekt av detta blir då att olika adresser aktiverar samma minnesutrymme.

Att använda ofullständig adressavkodning är ett val som konstruktören gör efter att ha läst specifikationerna på hur systemet skall se ut. Programmeraren får angivet basadressen för porten och använder denna adress utan att bry sig om hur adressavkodningen ser ut. Den enda nackdelen som programmeraren kan märka med ofullständig adressavkodning är om han programmerar fel och lyckas generera en adress som ligger i intervallen $45 00 00 till $457f ff. Han kommer då att adresseras porten utan avsikt.

Från konstruktörens synvinkel underlättar det att använda ofullständig adressavkodning då konstruktionen blir enklare. En nackdel

konstruktören upplever är den dag han skall expandera sitt system med fler portar eller mer minne än det den ofullständiga adressavkodningen var dimensionerad för från början. Då kan det tänkas att han måste konstruera helt ny adressavkodningslogik.

Avslutningsvis visar vi i figur 6.8 hur signalen EN-I/O-2 används för att aktivera en 8-bitars inport. Observera att signalen är grindad med R/W innan bufferten aktiveras. Om inte detta görs kan en "busskrock" uppstå i fall processorn skriver på inportens adress. Då kommer både bufferten (inporten) och processorn att lägga ut ett värde på databussen. Tänk dig då att processorn skriver ut enbart ettor på bussen och bufferten nollor så sker en kortslutning mellan dessa kretsar vilket *kan* innebära att processorns interna databuss buffertar förstörs.

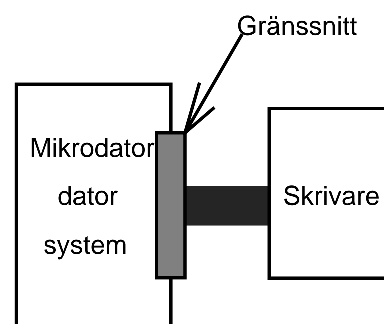


FIGUR 6.8 OFULLSTÄNDIG ADRESSAVKODNING FÖR EN INPORT

En utport konstrueras på samma sätt som inporten i figur 6.8. Enda skillnaden är att då används ett register i stället för en buffert och att R/W-signalen inte skall inverteras då porten skall aktiveras för en skrivning i stället för en läsning.

6.2 Parallell in- och utmatning

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 de problem och fördelar som finns med olika lösningarna.



FIGUR 6.9 *GRÄNSSNITT TILL SKRIVARE*

Du har säkert använt något *print fil* kommando när du arbetat med datorer tidigare och fått din utskrift på papper. Hur gick det där egentligen till? Det måste finnas ett antal olika programrutiner som utför bland annat följande:

- Läser av datorns tangentbord.
- Avkodar det inmatade *print* kommandot .
- Letar upp var den önskade filen *fil* finns på hårddisken.
- Läser in filen *fil* till processorns primärminne.
- Skriver ut primärminnesinnehållet (filen *fil*) till skrivaren.

Vi kommer nu att behandla det sista steget ovan, att skriva ut ett primärminnesinnehåll till skrivaren. Vi skall studera såväl mjukvaru- som hårdvarukonstruktioner (se figur 6.9). Mjukvarulösningarna är programrutiner som utförs av mikrodatorsystemet och hårdvaru-konstruktionerna finns i gränssnittet. Snittet som är placerat i mikrodatorsystemet har som uppgift att anpassa arbetstakten i det snabba mikrodatorsystemet mot den långsamma skrivaren. Vidare kan snittet även fungera som en *förstärkare* för den förhållandevis långa kabeln mellan skrivare och mikrodatorsystem.

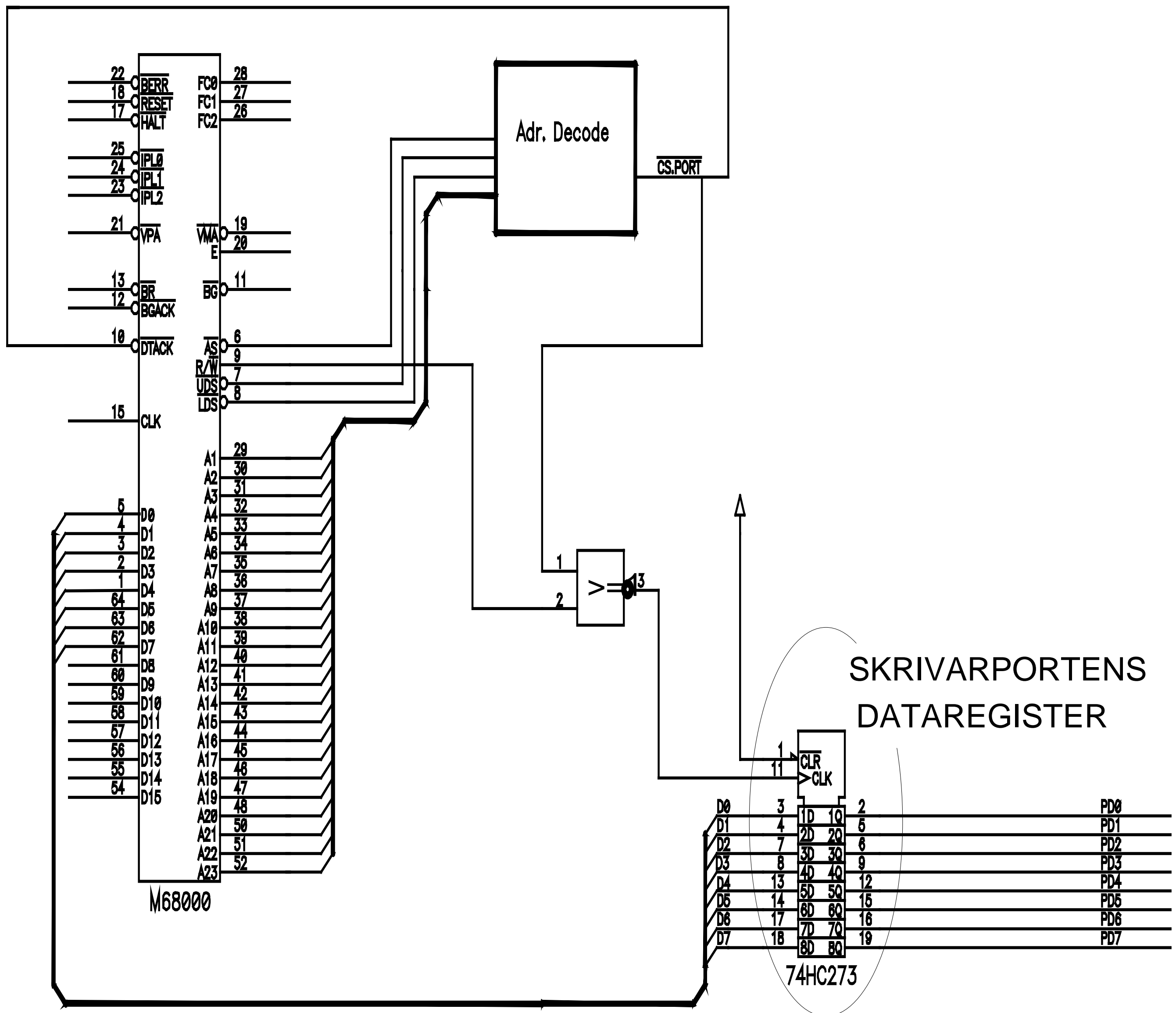
Varför behöver vi överhuvud taget en utport (ett gränssnitt) för att ansluta skrivaren? Man skulle kunna tänka sig att ansluta skrivarens buss direkt till processorns bussar. En sådan koppling skulle sannolikt inte fungera om skrivarkabeln blir för lång. Ett mikrodatorsystems bussar är i storleksordningen 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. Vidare skulle det bli betydligt mer störningskänsligt med så långa bussledningar om man inte kompenserade genom att använda kraftfulla bussdrivare för varje enskild krets.

Vi måste ge vissa förutsättningar för skrivaren. 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 handskakningssignaler** 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**.

6.2.1 Skrivarport med register.

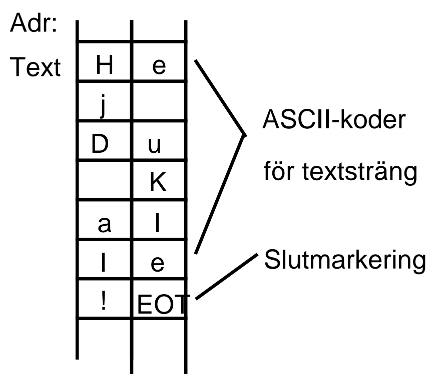
Den enklaste formen av en utport består av ett register anslutet till processorns bussar. Den nödvändiga hårdvarukopplingen för en utport mot en sådan skrivare visas i figur 6.10. Det enda som behövs är ett 8-bitars register för att mellanlagra tecknet som processorn skriver ut. Detta finns sedan tillgängligt på skrivarbussen så att skrivaren kan skriva ut tecknet.



FIGUR 6.10 SKRIVARPORT MED REGISTER

Innan vi studerar ett tänkbart program ger vi vissa förutsättningar för hur texten som skall skrivas ut lagras i minnet se figur 6.10. 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 markeringen **EOT** (End Of Text) som har

ASCII-koden \$04. EOT kommer att användas av programrutinen för undersöka när det sista skrivbara tecknet skickats till skrivaren:



FIGUR 6.11. TEXTSTRÄNG I MINNET

Följande programsekvens kan användas för att mata ut textsträngen till skrivaren via porten:

```
PRINTER     EQU         $418001         skrivarporten

      MOVEA.L     #Text,A1              Pekare till text
loop  MOVE.B      (A1)+,(PRINTER).L     Skriv ett tecken
      BRA         loop
```

Register **A1** används för att peka ut "nästa" tecken som skall skrivas. Under varje varv i slingan skrivs ett ASCII tecken (en *byte*) ut. 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 tredje mikrosekund. Detta innebär att vi med vårt program skriver ut ungefär $300 \cdot 10^3$ 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:

```
PRINTER     EQU         $418001         skrivarporten

      MOVEA.L     #Text,A1              Pekare till text

loop  MOVE.B      (A1)+,(PRINTER).L     Skriv ett tecken
      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

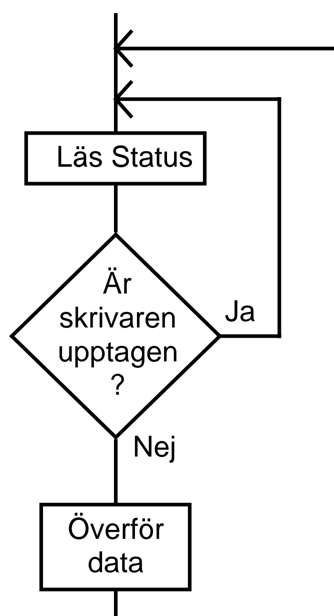
skrivaren är klar att ta emot ett tecken precis när det första skrivs ut. Startögonblicket måste *synkroniseras*. Om vi testar denna utskriftsrutin på en lång utskrift så kommer det definitivt att bli fel då det är mycket svårt (omöjligt) att få två olika klockor (en i skrivaren och en i datorsystemet) att gå synkront (gå lika fort). Felet kommer att uppträda som missade tecken eller dubletter. Vårt exempel skulle på papperet kunna se ut som följer, *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 skall nu övergå till olika typer 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.

6.2.2 Skrivarport med register och READY-signal.

Vi förutsätter nu att skrivaren är utrustad med en utsignal (**READY**) som indikerar om skrivaren är redo att ta emot ett nytt tecken eller inte.



FIGUR 6.12 *BUSY WAIT*

READY = 1 (hög nivå) indikerar att skrivaren är klar att mottaga ett nytt tecken.

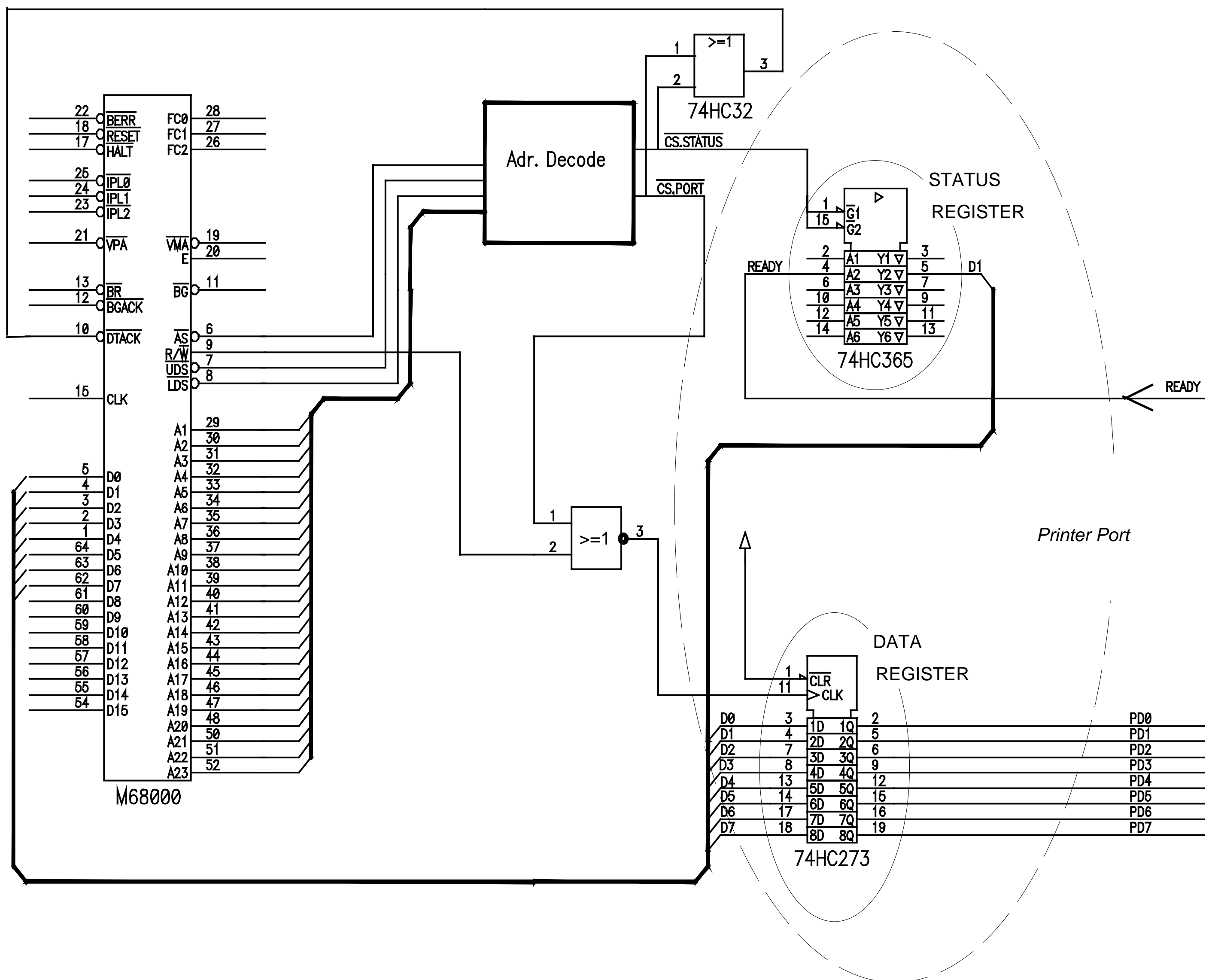
READY-signalen kan läsas via ett register (Status Register) av processorn. På så sätt kan en programkonstruktion liknande vad som visas i figur 6.12 användas.

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**.

Studera figur 6.13 som visar skrivarporten med busy-signalering. Ett programförslag som är anpassat till printerporten med busy-signal visas nedan. 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    $418001              Adress till Data Reg
STATUS       EQU    $418003              Adress till Status Reg
             MOVEA.L      #Text,A1                   Pekare till
text
busy         BTST.B       #1,(STATUS).L              Testa READY bit
             BEQ          busy                       Hoppa om noll

             MOVE.B       (A1)+,(PRINTER).L         Skriv ett tecken
wait         BTST.B       #1,(STATUS).L              Testa READY bit
             BNE          wait                       Hoppa om ett
             BRA          busy
```



FIGUR 6.13. PRINTERPORT MED BUSY-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.

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

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.

6.2.3 Skrivarport med register DAV och STROBE-signal.

Vi anger nu nya förutsättningar för skrivaren och utrustar den med en insignal och skrivarporten med en utsignal **DAV** (Data Available). Vidare förutsätter vi att skrivaren är utrustad med en utsignal (**STROBE**) som aktiveras när skrivaren läser tecknet från skrivarportens dataregister.

Studera hårdvarukopplingen i figur 6.14. Observera att när dataregistret skrivs så klockas samtidigt FLIP-FLOP:en som genererar **DAV**, på så sätt "ser" skrivaren att ett tecken finns i dataregistret. Skrivaren signalerar med **STROBE** att den läser dataregistret. Signalen används här för att nollställa **DAV** signalen.

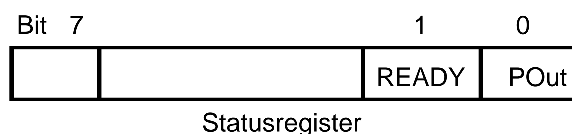
FIGUR 6.14 SKRIVARPORT MED DAV-OCH STROBE-SIGNAL

En sådan hårdvarukonstruktion skulle kunna fungera utmärkt med vårt tidigare programförslag som utnyttjade en fördröjningsrutin för varje tecken som skrivs ut. Observera att om skrivaren klarar av att skriva ett tecken var 10:e ms. så bör datorn skriva ut tecknen något långsammare. Detta är antytt i koden (studera programmet som följer nedan) med att ha en fördröjning på 12 ms.

6.2.4 Skrivarport med register READY, DAV och STROBE-signal.

Olyckligtvis kan vi förvänta oss problem om skrivaren får slut på papper eller "papperskvadd". I sådana fall märker datorsystemet inget och fortsätter att skriva ut tecken. För att lösa detta problem konstruerar vi vår skrivarport med både **READY**, **DAV** och **STROBE** signal, se figur 6.15. Observera att **READY** har samma funktion som i det tidigare exemplet.

Skrivaren i figur 6.15 är utrustad med en signal som indikerar *Paper Out*. Signalen är ansluten till portens statusregister och kan testas på samma sätt som *Ready* i exemplet vi studerat tidigare.



FIGUR 6.15 SKRIVARPORT MED DAV, READY OCH STROBE SIGNAL

Studera nu följande programexempel för skrivarporten i figur 6.15. Observera att vi nu använder symbolnamn för *READY* och *POut* (slut på papper). *READY* är ansluten till D1 och *POut* är ansluten till D0 i figur 6.15.

```
* Programförslag för printerport med DAV, READY och STROBE-signal:
* Definitioner

Ready       EQU         1                   Bit for Ready
POut        EQU         0                   Bit for Paper Out
Printer     EQU         $418001             Data Register
PStat       EQU         $418003             Status Register

* Programrutin
            MOVEA.L     #Text,A0            Läs startadress
            BTST.B #POut,(PStat).L          Är papperet slut
            BNE         PaperOut            Hoppa om Ja

NextChar:
NotRdy:
            BTST.B #Ready,(PStat).L         Skrivaren klar?
            BEQ         NotRdy              Nej, Not Ready
            MOVE.B (A0)+,D0                 Läs tecken
            CMPI.B #EOT,D0                  Sista tecken?
            BEQ         PEnd                Hopp om Ja.

            MOVE.B D0,(Printer).L           Skriv tecken
            BRA         NextChar            Skriv nästa
PEnd        ...
            ...

PaperOut    ...               Rutin som tar hand om papperskvadd
            ...
```

Vi har nu konstruerat hårdvara och mjukvara som klarar av en utskrift till skrivaren. Konstruktionen klarar av att skriva ut text utan att det blir dubletter och utan missade tecken på papperet. Både hårdvaran och programvaran är enkelt uppbyggda och överskådliga.

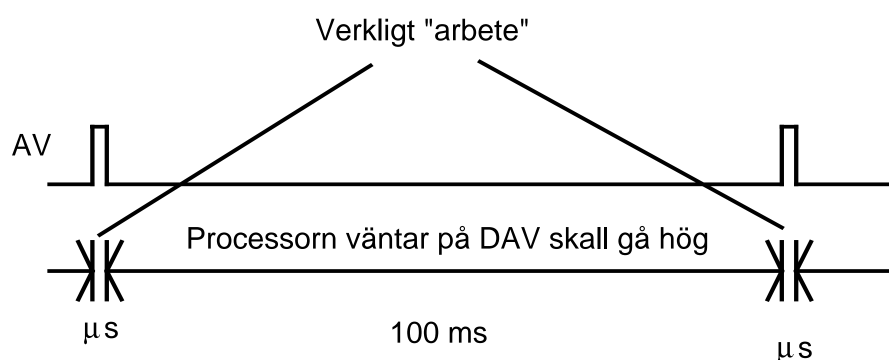
I vårt exempel är skrivarporten utrustad med en vippa som sköter **DAV** signalen som sätts automatiskt när datainregistret skrivs av processorn. **DAV** kunde like gärna funnits i ett styrregister (utregister) som processorn kunde skrivit en etta i efter att ha gett ett nytt tecken:

```
MOVE.B (A0)+,(Printer).L          Skriv tecken
MOVE.B #DAV,(PControl).L          Sätt DAV
       ...
       ...
MOVE.B #0,(PControl).L            Nollställ DAV
```

`PControl` skulle alltså vara ett register där processorn (eller programmet) har möjlighet att skriva **DAV** signalen till skrivaren. Observera att här måste processorn själv nollställa **DAV** signalen, och då dyker problemet upp, hur länge skall **DAV** vara aktiv?. Detta beror ju på hur skrivaren är konstruerad och vilka specifikationer den har. Den förra lösningen, där skrivarens *Strobe* signal nollställer **DAV** är mycket mer elegant ty där löses problemet i hårdvara.

6.2.5 Avslutande resonemang kring printerporten.

Konstruktionen fungerar, om det är intet tvivel, tecken skrivs endast till skrivarens dataregister då den är redo att ta emot ett nytt och skrivaren läser endast dataregistret efter att ett nytt tecken är gett. Frågan vi bör ställa oss nu är hur effektivt processorn utnyttjas under den tid en utskrift pågår.



FIGUR 6.16 FÖRHÅLLANDET MELLAN VERKLIGT ARBETE OCH VÄNTAN

Funderar vi lite över vilka instruktioner som utförs i de givna programförslagen, och hur lång tid de olika program-snurrorna tar att exekvera jämfört med den tid det tar att skriva ut ett tecken så kommer vi fram till att datorsystemet väntar i 99.99% av tiden. Se figur 6.16. Från vår synvinkel utför ju processorn nyttigt arbete endast när den skickar ett tecken till skrivaren och inte när den inväntar att skrivaren skall bli redo att ta emot nästa tecken.

Vi har följaktligen konstruerat ett system som är fruktansvärt ineffektivt. Processorn är mesta tiden upptagen med att undersöka om skrivaren kan ta emot ett nytt tecken. Om skrivaren klarar att skriva ut 10 tecken per sekund undersöker processorn **DAV** signalen ca 30 000 gånger per utskrivet tecken. Detta beror på att vi valde konstruktionen **Busy Wait** vilken kan vara praktiskt användbar i en del tillämpningar men knappast till vår printer port där vi har *fullständig handskakning* med **DAV**, **STROBE** och **READY** signalerna.

Studera följande huvudprogram som är tänkt att fungera med denna konstruktion (busy wait).

```
* HUVUDPROGRAM
            ...                 Olika initieringar
            ...

mainloop    ...
            ...                              Annat arbete som
            ...                              processorn utför
            ...
            JSR         Print                Skriv ut en fil
            ...
            BRA         mainloop
```

Vi kan förmoda att det finns någon form av programslinga i huvudprogrammet som exekveras oupphörligt. Annat arbete som processor utför kan vara att läsa en temperaturgivare, beräkna vilken temperatur värdet motsvarar, påverka en värmekälla och att skriva temperaturen på en display mm. Skulle vi nu hamna i en lång utskrift medför detta att det blir ett "avbrott" i temperatur mätningen som i vissa sammanhang inte kan tolereras.

I stället för att i utskriftsrutinen `(Print)` undersöka om skrivaren är redo (Busy) att ta emot ett nytt tecken kan vi tänka oss att placerad denna test i huvudprogrammet. Rutinen som skriver ut en textfil minskas då till att endast skriva ut ett tecken. Se nedanstående huvudprogram och subrutin. Programmet är utökat med en flagga (`Flag`) som indikerar om det överhuvudtaget pågår någon utskrift. Om ej så är fallet fortsätter programmet *utan* att testa READY signalen då utskrift ej pågår.

```
* HUVUDPROGRAM
     ...                 Olika initieringar
     ...
mainloop          ...
     ...          Annat arbete som ...
     ...          ....    processorn utför
     ...
     BTST.B       #0,(Flag).L         Pågår Utskrift
     BEQ          NoPrint             Hoppa om Nej

     BTST.B       #Ready,(PStat).L    Skrivaren Redo?
     BEQ          NotRdy              Hoppa om Nej

     JSR          Print1Char          Skriv nästa tecken

NotRdy                                Fortsätt om ej klar
NoPrint
     ...
     ...
     BRA          mainloop
```

```
* SUBRUTIN
Print1Char
       MOVEA.L      (TextPek).L,A0          Adress till tecken
       MOVE.B       (A0)+,(Printer).L       Skriv tecken
       MOVE.L       A0,(TextPek).L          Spara nästa Adr
       RTS

* Variabler
TextPek       DS.L   1      Pekare för textsträng
Flag          DS.B   1      Flagga som indikerar om utskrift pågår
*                           Flagga bit 0 = 0 : ingen utskrift pågår
*                           Flagga bit 0 = 1 : utskrift pågår
```

Programkonstruktionen vi valt här kallas **Polling** ("rundfrågning) som går ut på att *ibland* undersöka om en yttre enhet önskar någon form av service. I huvudprogrammet testar vi (exempelvis) en gång varje varv i programloopen om skrivaren är redo att ta emot ett nytt tecken (om någon utskrift pågår).

Allmänt, tar polling mycket datorkraft. Tänk dig ett antal terminaler som är anslutna till *ett* datorsystem på ett företag. Tänk dig vidare *hur ofta* de anställda använder tangentbordet på sin terminal. Om detta datorsystem använder sig av polling för att periodiskt undersöka om någon av användarna trycker på tangentbordet åtgår åtskillig processorkraft.

Processorn som arbetar efter polling- principen frågar väldigt ofta i onödan om de olika tangentborden är aktiverade. Polling principen skall *inte* förkastas då den är lätt att förstå, enkel att implementera och därför mycket använd, men kan inte användas urskiljningslöst som exemplet visade.

Om vi för att ögonblick återgår till vår printerport i figur 6.15 och studerar signalerna från skrivaren ser vi att signalen *Ready* är intressant. Det är ju när denna signal går hög som vi skall utföra något arbete. Tänk dig att denna signal kopplas direkt till processorn styrenhet för att avbryta processorns normala exekvering av huvudprogrammet. Tänk dig vidare att processorn själv kan:

- spara undan sin PC
- starta en rutin som skriver ut *ett* tecken till skrivaren för att
- sedan återstarta huvudprogrammet

Detta förfarande liknar mycket på vad vi studerar tidigare, nämligen

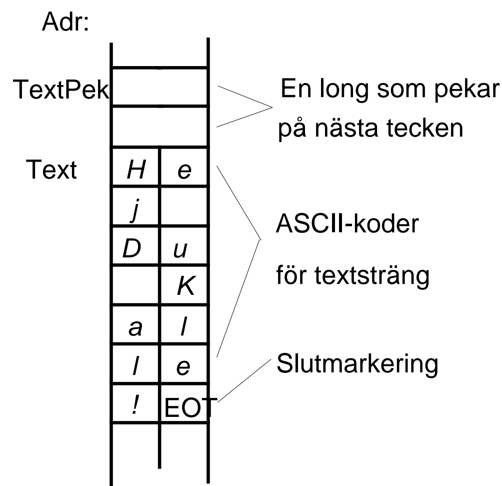
```
       JSR          print Skriv ett tecken
```

Nu däremot, är det en hårdvarusignal direkt in till processorn och inte en instruktion (JSR) som får datorsystemet att skriva ett tecken till skrivaren. En sådan konstruktion kallas för *avbrott* (**Interrupt**) och har stora fördelar jämfört med *Polling* och *Busy Wait*.

Vid avbrott påkallas processorn uppmärksamhet *endast* efter det att den yttre enheten (skrivaren) signalerat att den önskar service (är redo att ta emot ett nytt tecken). Ett system som använder sig av avbrott kallas ett *avbrotts drivet system*. Vidare kallas programmet som exekveras under ett avbrott för *avbrottsrutin.*

För att beskriva principen för en avbrotts driven printerport ger vi först förutsättningen för hur text och en pekarvariabel placeras i minne, vidare påpekar vi att signalen *Ready* i figur 6.15 är ansluten direkt till processorns avbrottsingång.

Betrakta nu figur 6.17. En *long* pekarvariabel *TextPek* anger adressen till nästa tecken som skall skrivas ut. Denna variabel måste läses av avbrottsrutinen för att hitta tecknet. Vidare måste variabeln uppdateras för att peka ut nästa tecken vid nästa avbrott. Nedan visas lämplig kod som sköter utskrift av en textfil. Principer och teknik för avbrott behandlas detaljerat i kapitel 7.



FIGUR 6.17. TEXTSTRÄNG I MINNET

```
* AVBROTTSRUTIN
PrintIRQ
    MOVEA.L    (TextPek).L,A0        Adress till tecken
    MOVE.B     (A0)+,(Printer).L     Skriv tecken
    MOVE.L     A0,(TextPek).L        Spara nästa Adr
    RTE

TextPek
    DS.L       1                     Pekarvariabel
```

6.3 Seriell I/O

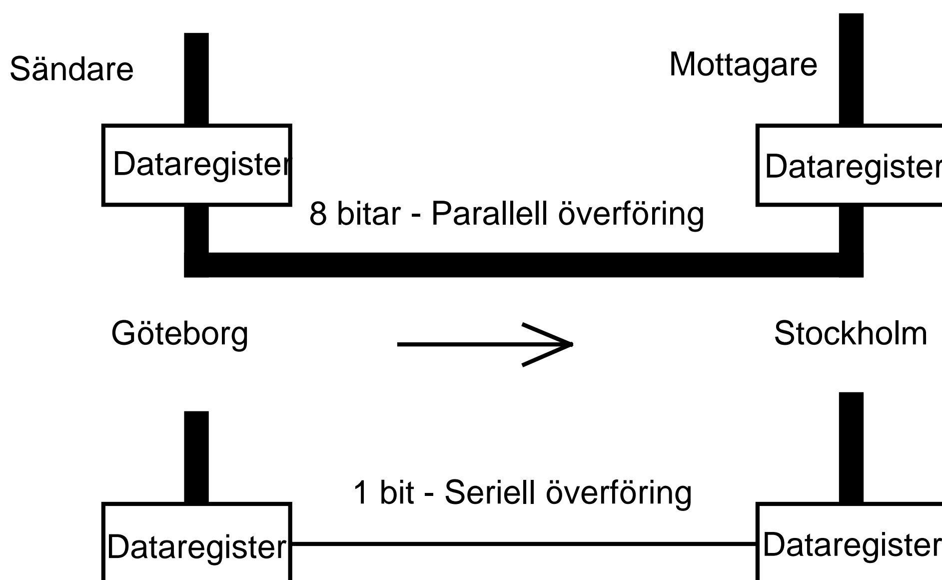
Vid seriell I/O överförs databitarna efter varandra på en och samma signalledning. Detta minskar antalet signalförande ledningar mot exempelvis den parallella skrivarporten där åtta parallella signalledningar används mellan datorsystem och skrivare. Den principiella skillnaden är enkel: I stället för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

Vi skall i de följande avsnitten beskriva grundläggande begrepp som *synkron* och *asynkron* överföring, *överföringskapacitet*, *protokoll*, mm som är utmärkande för seriell I/O. Vidare ger vi en introduktion till *datanät*. Överhuvudtaget, för kommunikation mellan platser belägna på stora avstånd från varandra och där prestandakraven, dvs svarstiderna, är mindre kritiska väljs ofta seriell överföring av data.

Exempel på sådana tillämpningar är världsomfattande nät för flygbokningssystem, men även nationella nät som exempelvis sammankopplingen av bankomater till en centraldator, nyttjar seriell överföring. Ytterligare exempel på seriell överföring är: överföring mellan dator och terminalindustridatanät som PC-NET, LANTASTIC och ETHERNET.

Inledningsvis definierar vi 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, hur handskakningsförloppet skall gå till mm.

Vid "långa avstånd" mellan två system används som sagt oftast seriell överföring eftersom det kostnadsmässigt är mycket billigare. Med *långa avstånd* menas här från "någon meter" och uppåt. Se figur 6.18.



FIGUR 6.18 SERIELL OCH PARALLELL ÖVERFÖRING

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.

Vidare så menar man oftast med parallell överföring att det finns någon form av handskakning med **READY** och **DAV** liknande det som visades för printerporten. Detta innebär att när sändaren i Göteborg har skickat ett tecken och en **DAV**-signal 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:

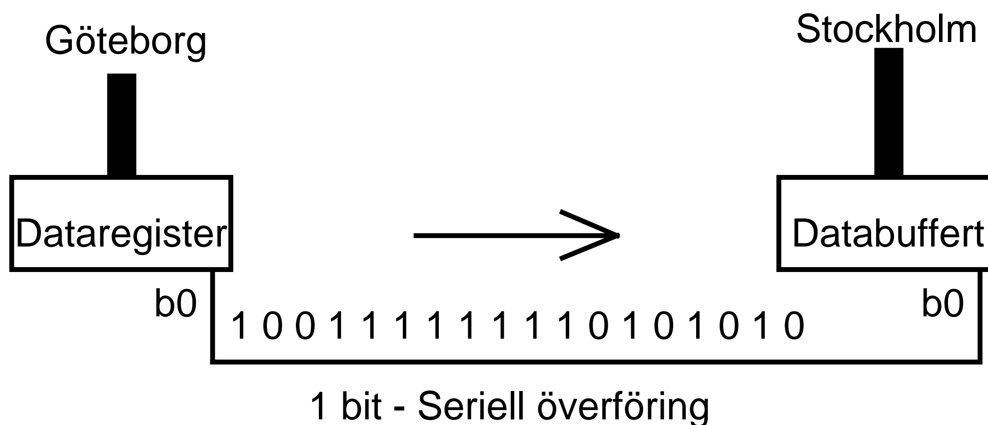
$$t = \frac{l}{c} = \frac{50 \cdot 10^4}{3 \cdot 10^8} = 1{,}66 \qquad ms$$

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.)

Vad man *i stället* väljer är 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 6.19 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).

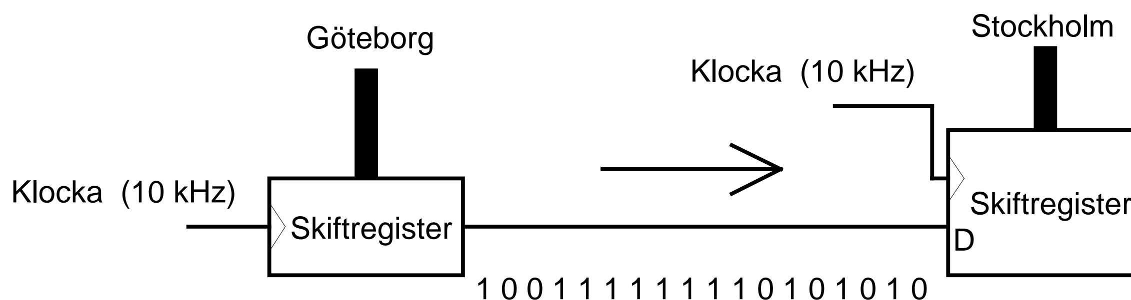


FIGUR 6.19 SERIELL ÖVERFÖRING MED DATAREGISTER.

```
*   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
```

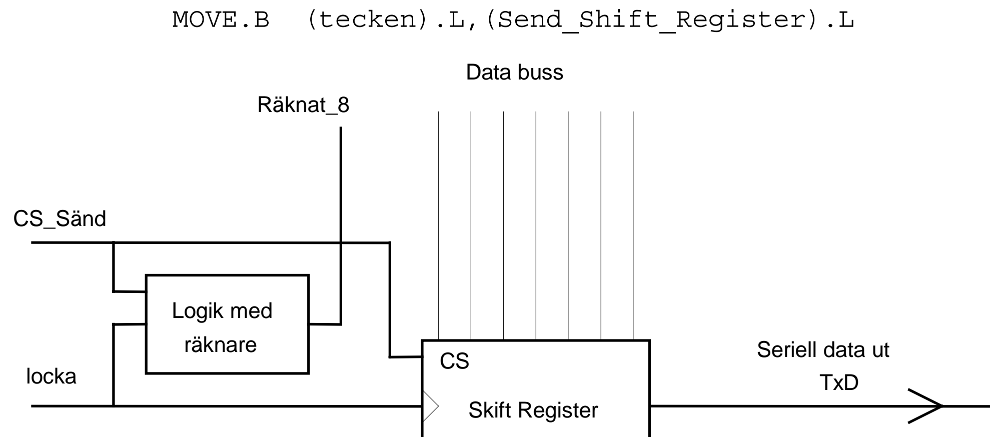
Nackdelen med denna form av konstruktion är att sändaren *programmässigt* måste skicka varje enskild bit. Ett enklare förfarande är att utnyttja *skiftregister* istället för data register och buffert. Vidare ersätts subrutinen i programexemplet ovan med en klocka som levererar en frekvens på 10 kHZ. (se figur 6.20). 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 6.20 SERIELL ÖVERFÖRING MED SKIFTREGISTER.

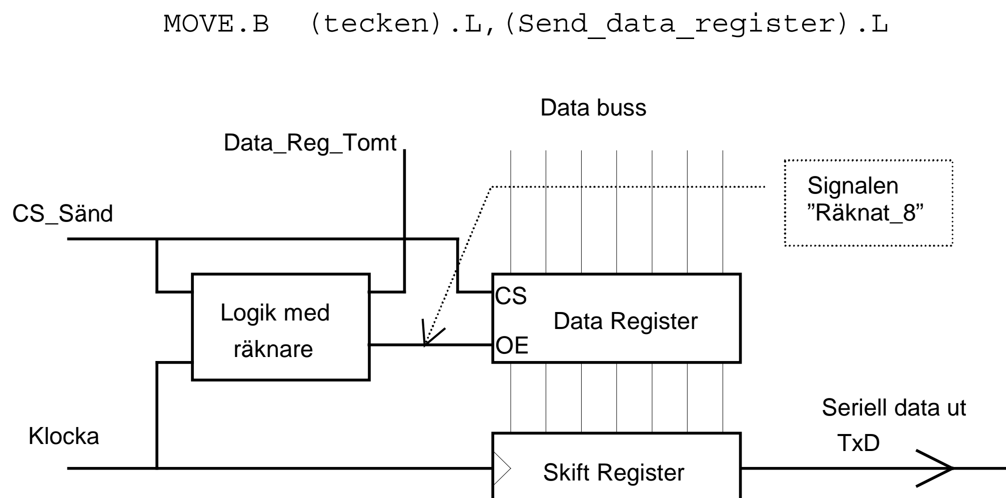
För att sändaren programmässigt inte skall behöva räkna perioder på 0,8 ms för att veta när nästa byte skall skickas kan sändardelen utökas med en räknare och ett logikblock i anslutning till skiftregistret. Studera figur 6.21.



FIGUR 6.21 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 ovanstående problem så används en teknik som kallas dubbelbuffring. Det går ut på att koppla in ett dataregister i anslutning till skiftregistret. Se figur 6.22.

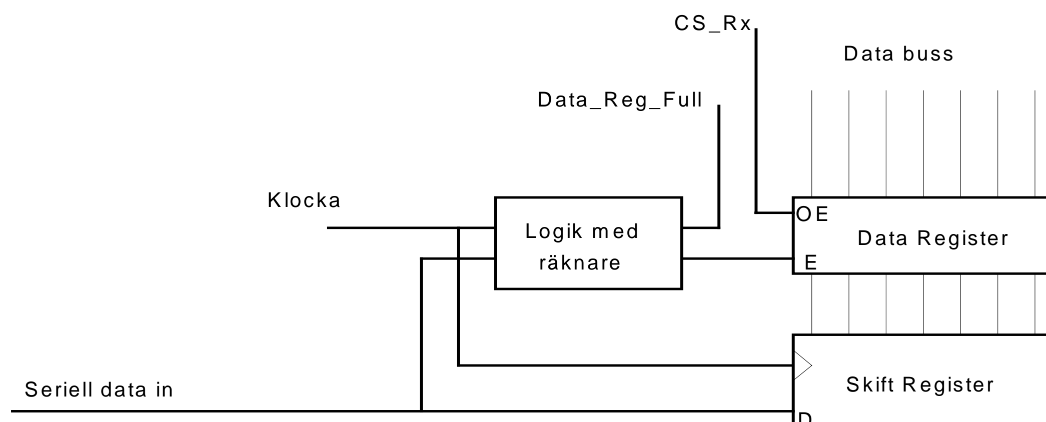


FIGUR 6.22 DUBBELBUFFRAD SÄNDNING.

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.

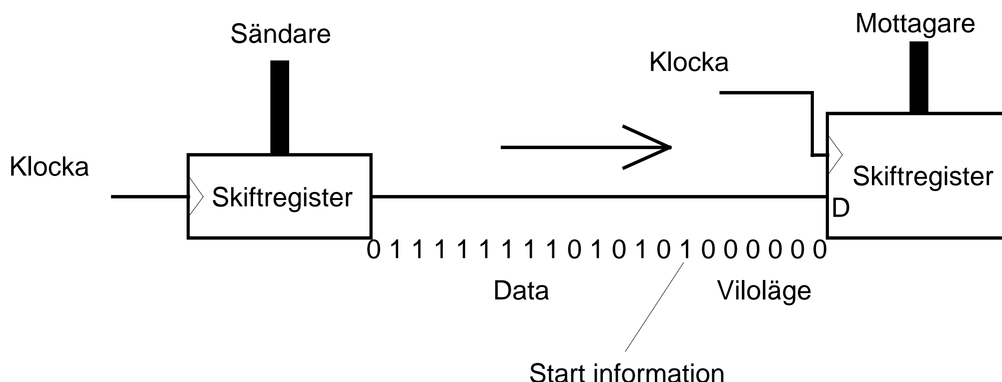
Liknande dubbelbuffring hittas på mottagarsidan. I anslutning till motagarens skiftregister finnns ett data register (se figur 6.23). När skiftregistret har klockat in 8 bitar överförs skiftregistrets innehåll automatiskt till dataregistret med hjälp av logikblocket. Samtidigt sätts någon Data_Register_Full signal som mottagaren kan polla för att undersöka om data har anlänt.

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



FIGUR 6.23 DUBBELBUFFRAD MOTTAGNING

Vi skall nu i detalj studera vad som händer på mottagarsidan. Se figur 6.24 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 6.24 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.

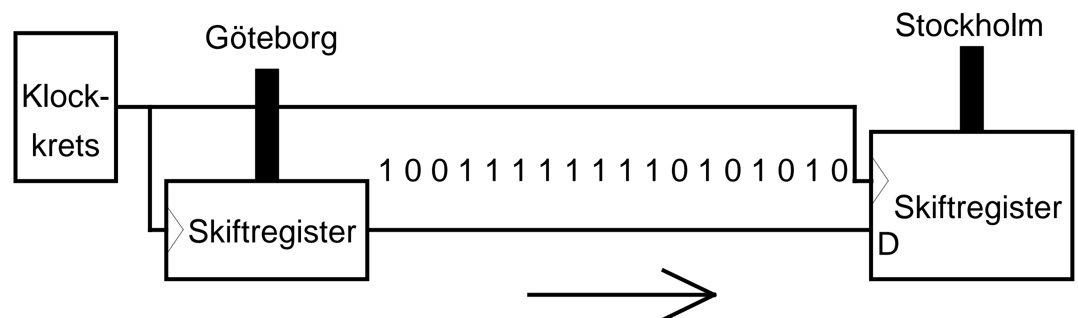
Med samma princip som i vårt första enkla programförslag för seriell I/O skulle vi ju kunna känna av att nivån på signalledningen ändras från 0V till +5V och vice versa under de första bitarna som anländer till mottagaren då dessa består växelvis av nollor och ettor. Men vad händer med de följande bitarna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

6.3.1 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. Denna klocksignal används av mottagaren för att skifta in databitarna i skiftregister. Se figur 6.25.

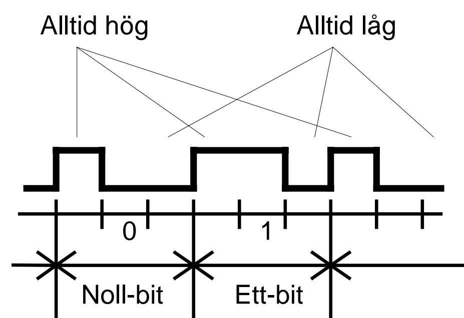


FIGUR 6.25 SYNKRON ÖVERFÖRING MED SEPARAT KLOCKSIGNAL.

När data och klocksignal överförs på samma signalledning säger man att databitarna är kodade (med klockinformation). Manchester-kodning

är ett exempel på detta vilket visas i figur 6.26. 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.



FIGUR 6.26 MANCHESTERKODNING

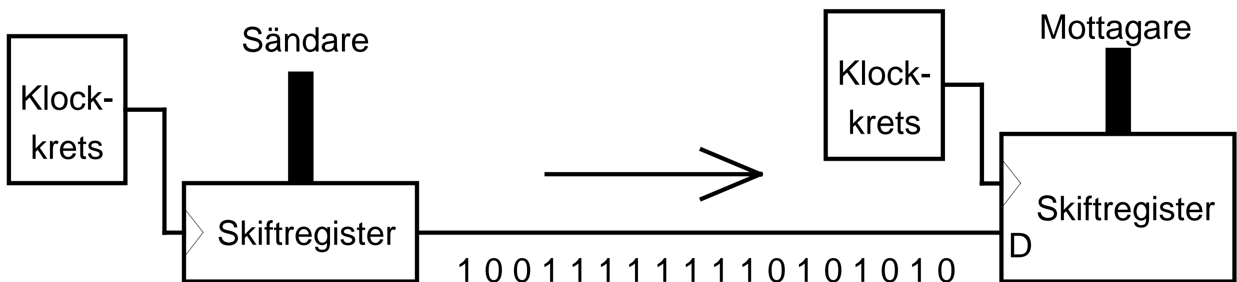
6.3.2 Asynkron överföring

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 6.27. 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.

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.



FIGUR 6.27 ASYNKRON ÖVERFÖRING

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.

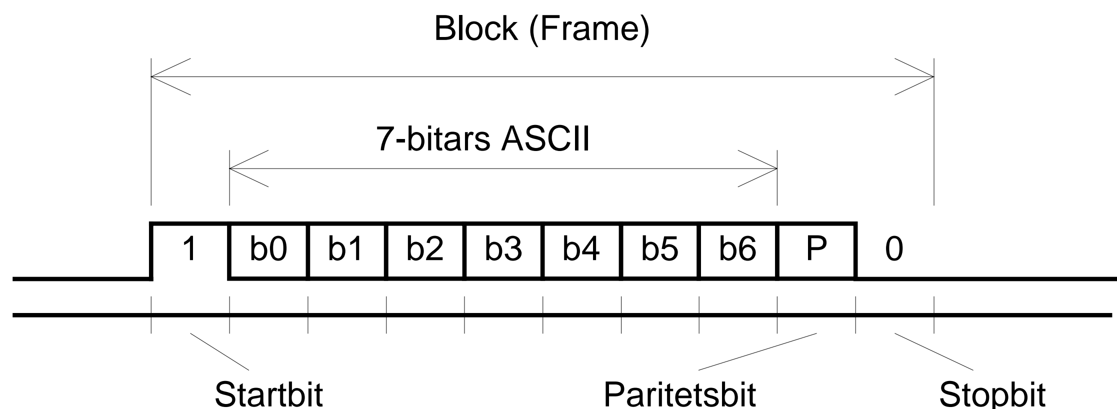
Protokollet **RS232** är ett exempel på ett protokoll som används mycket mellan terminal och dator. Vad som ingår i detta protokoll varierar lite beroende på vilka personer man diskuterar med, men i dagligt tal menar man att man utnyttjar någon form för USART och överför ett ASCII tecken per block. Vidare att man använder en speciell kontakt som kallas D-SUB och att spänningsnivåerna på serieledningen är förutbestämda att variera mellan ±12V, mm. (COM1 och COM2 portarna på en IBM/PC använder 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 6.28 visar en logisk bild av protokollet som har 7 databitar, en startbit, en paritetsbit och en stoppbit. Vi skall här koncentrera oss på hur en mottagare kan ta emot (klocka in) ett sådant format.

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.

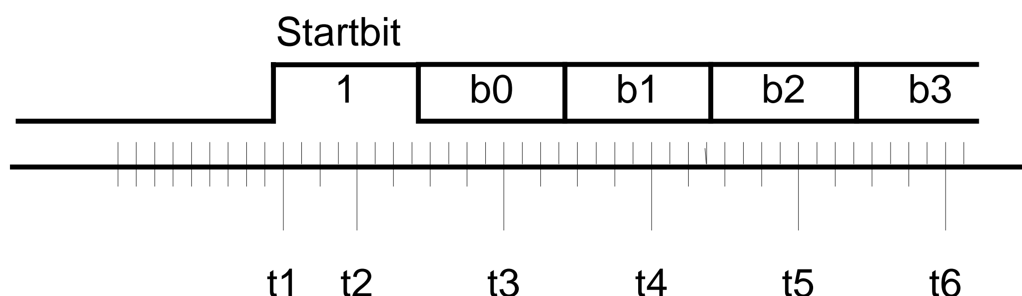
Databitarna beror av överförd data och kan således anta olika värden. Paritetsbiten kan defineras som *udda* eller *jämn*. Med detta menas att summan av antal logiska ettor i överföringen skall anta ett udda eller ett jämt värde. Paritetsbiten *sätts* därför *av sändaren* (till noll eller ett) för

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.



FIGUR 6.28 SERIEFORMAT MELLAN TERMINAL OCH DATOR.

Vi ska nu studera synkroniseringsproblem 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 6.29.



FIGUR 6.29 SERIEFORMAT MELLAN TERMINAL OCH DATOR.

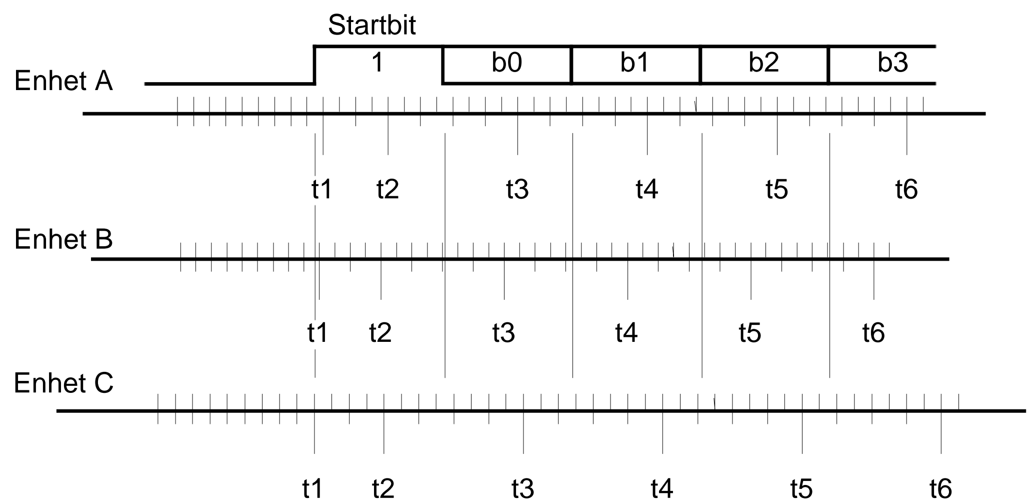
Sändaren skickar sin start bit följt av databitarna osv enligt sin klocka på 1 kHz. När en hög nivå detekteras av mottagaren (vid t1 i figur 6.29) tolkas detta som början på en startbit och en räknare i mottagaren initieras nu att räkna 4 st 125μs pulser. På så sätt har vi kommit ungefär till mitten 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 6.30. 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.

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).



FIGUR 6.30 SYNKRONISERINGSPROBLEM MELLAN ENHETER..

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 önskar man att ange data 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). I forskningssammanhang experimenterar man även med Gbit-överföring (10^9 bitar per sekund) mellan exempelvis Stockholm och Göteborg. Med BAUD menas antal överförda bitar (data, start, stop, övriga styrbitar) per sekund.

En typisk överföringshastighet mellan en terminal och ett datasystem i dag är 9600 BAUD. I vanligt tal säger man att man "kör seriellt på nio och sex". Detta innebär alltså att man kan överföra ca 900 ASCII tecken per sekund. En äldre teletype (liknande en gammal telex apparat) klarar en överföringshatighet på 110 BAUD. Generellt kan man säga att ju kortare avstånd och ju bättre (störokänsligare) kabel ju högre hastighet kan man använda.

Kretsar (UART, USART, DUART, ACIA, mf) som innehåller skiftregister med bubbelbuffring, klockor för både sändning och mottagning, paritetskontroll, mm har funnits på marknaden sedan lång tid. Dessa kan anslutas direkt till processorns buss-system och programmeras för en viss BAUD RATE, för udda eller jämn paritet mm. En konstruktör bygger därför enkelt sin seriekommunikation med någon sådan kretsa. En av MOTOROLAS seriekretsar anpassade för M680x0-familjen betecknas MC68681 och beskrivs i ett senare kapitel.

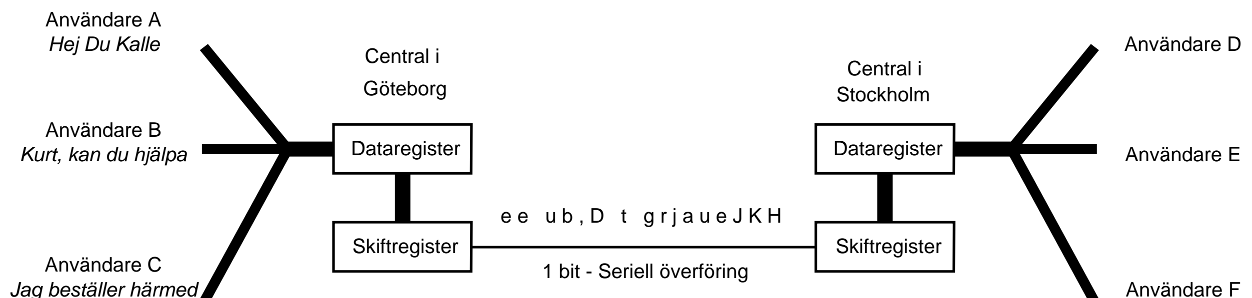
Då det gäller kommunikation mellan terminal och dator har vi tidigare nämt protokollet **RS232**. Detta specificerar de spänningsnivåer och anslutningar som används vid serieöverföringen. Dessutom beskrivs handskaknings-signaler med modem, om sådant används.

Vad vi hittills diskuterat, då det gäller seriell I/O, är *punkt till punkt-förbindelse* mellan en *sändare* och en *mottagare* (en användare i varje ända av kabeln) Om varje företag som har kontor både i Göteborg och Stockholm skulle dra egna kablar mellan kontoren skulle det innebära avsevärda kostnader för varje enskilt företag. Man vill då hellre hyra in sig på en befintlig förbindelse som exempelvis televerket har kopplat upp . Flera användare kan utnyttja samma förbindelse mer eller mindre parallellt. En användare (ett företag) kan välja mellan en *fast förbindelse* eller *uppkopplingsbar förbindelse*.

En uppkopplingsbar förbindelse fungerar på samma sätt som ett telefonnät. Användaren lyfter luren, slår ett nummer och kommer i kontakt med mottagaren. Om det är hög belastning på nätet så kan användaren riskera att inte komma i kontakt med mottagaren och måste eventuellt försöka senare. Användaren betalar "per samtal". En fast förbindelse är ständigt uppkopplad mellan två eller flera användare och "kostar pengar" oberoende av om den används eller ej.

Då fler användare skall överföra data på en och samma kabel mellan exempelvis Göteborg och Stockholm måste man uttnyttja överföringskapaciteten för kabeln på bästa sätt. Det är orimligt att varje användare med en fast förbindelse skall vara ensam on EN kabel. Vad man då gör är att på något sätt tidsdela ("*multiplexa"* från engelskans *multiplexing*) användningen av *en* kabel. Tidsmultiplex innebär att sändarcentralen (tidsmässigt) fördelar kabelns kapacitet mellan användarna.

Studera figur 6.31 nedan, som visar tidsmultiplex seriell överföring. Sändarcentralen måste här buffra upp data från de olika användarna A, B och C och portionera ut det i lagom stora block på kabeln om det är hög belastning. På mottagarsidan är det ett liknande förfarande, mottagaren måste dela upp inkommande data och skicka rätt datablock till rätt mottagare D, E eller F.



FIGUR 6.31 TIDSMULTIPLEXAD ÖVERFÖRING

Figuren exemplifierar hur centralen i Göteborg skickar en byte från varje användare A, B och C i turordning. Användare A skickar texten "Hej Du Kalle..." till användare D, användare B skickar "Kurt, kan du hjälpa...." till användare E osv. Sändaren Skickar således först ett “H” från användare A, sedan ett “K” från användare B, sedan ett “J” från användare C. Nu är ett tecken skickat från varje användare och då är det på nytt A:s tur. Sändaren fortsätter därför med att skicka ett “e” från A sedan ett “u” från B, ett “a” från C osv.

Det första som anländer Stockholm är således ett “H” från användare A, vidare ett “K” från användare B, ett “J” från användare C, sedan ett “e” från användare A, osv. Mottagaren samlar in data och distribuerar detta till rätt användare.

En annan möjlighet är *frekvensmultiplex* där användarnas data moduleras med olika frekvenser på en bärvåg. Vi går inte vidare in på detta här.

Låt oss roa oss med att fundera på hur många databitar som kan "vara i luften" samtidigt i rikning Stockholm. Vi förutsätter en klockfrekvens på 1 MHz vilket motsvarar ett bitintervall på 1 µs.

$$Antal\ bitar = \frac{L}{C \cdot \frac{1}{f}} = 1,7 \cdot 10^3\ bitar$$

där L= 50 mil, C = $3 \cdot 10^8$ och f = 1 Mhz.

6.4 Datanät

Vi ska nu ge en kort introduktion till *datanät*. Med ett datanät menas ett antal datorer eller noder (**Nodes**) sammankopplade via ett media (en kabel). Dessa noder kan på ett eller annat sätt komunicera med varandra över mediat så att varje användare kan utnyttja *en* eller *flera datorer* som är anslutna till datanätet. I datanätsammanhang är det underförstått att noderna kommunicerar seriellt.

Datanäten grupperas i *lokala datanät* (**LAN, Local Area Network**) och kontinentala (**WAN, Wide Area Network**). När man vill beskriva hur datanät fungerar eller vad de kan användas till (vilka "konster" de kan utföra) hänvisar man ofta till den 7-nivåers OSI/ISO modellen.

De sju nivåerna i OSI-modellen beskriver olika typer av tjänster eller specifikationer. På nivå ett beskrivs exempelvis hur anslutningar ser ut och vilka spänningsnivåer som skall användas på seriekabeln. På en högre nivå beskrivs vilken "data-väg" ett meddelande skall färdas för att överföras exempelvis mellan Göteborg och Tokyo. På den högsta nivån, sju, beskrivs exempelvis hur användar snittet (**MMI, Man Machine Interface**) för elektronisk post (eng: **Electronic Mail** eller bara **E-Mail**) skall fungera. Vi koncentrerar oss här dock bara på de lägsta nivåerna.

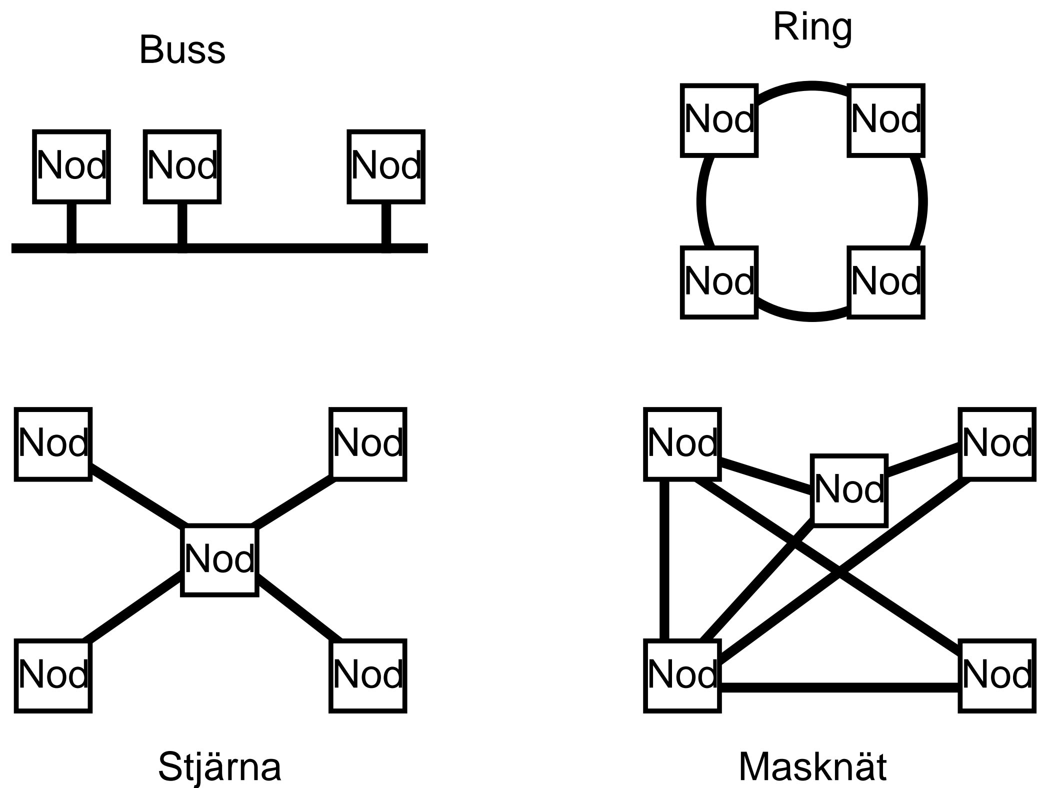
I datanätssammanhang talar man om *media*, *topologi* och *acessmetoder*. Med media menar man den *kabeltyp* man använder. De vanligaste är: **tvinnad partråd**, **koaxialkabel** och **optisk kabel** (fiberoptik). Jämför man dessa inbördes med avseende på överföringshastighet (kapacitet), flexibilitet (hur enkelt är det att ansluta en extra dator till ett befintligt system), kontaktering (hur dyra kontakter, kablage och verktyg behövs) och störokänslighet kan resultatet sammanfattas i följande tabell:

	Tvinnad partråd	Koaxialkabel	Fiberoptik
Kapacitet	Låg	Medium	Mycket hög
Flexibilitet	Mycket hög	Medium	Låg
Kontaktering	Billig	Medium	Mycket dyr
Störokänslighet	Låg	Bra	Mycket hög

Med *topologi* menas hur nätets struktur ser ut. De vanligaste är:

- buss
- stjärna
- ring
- masknät

Figur 6.32 visar principerna för olika topologier. Masknätet liknar de kontinentala telefonnäten och är mycket flexibla. Om en "dataväg" slås ut kan en alternativ väg användas.



FIGUR 6.32 *TOPOLOGIER.*

Det vanliga och flexibla *Ethernet* använder busstopologi. En nackdel med detta kan vara att bussen kommer att utgöra en "flaskhals" då belastningen stiger över en viss nivå.

Stjärntopologin, med en *master* centralt, är enkel att använda. Ring-nät har hög överföringskapacitet. En nackdel med såväl stjärn- som ring-topologier är nätens stora sårbarhet. Hela nätet slås exempelvis ut om *en* nod i ringnätet slås ut. Dessa topologier används därför enbart för lokala nät där den geografiska utsträckningen är liten.

Med *acessmetoder* menar man den *policy* som används när en nod skickar data. De vanligaste är:

- polling
- CSMA
- TDMA
- "token"

Vid polling finns det en "master" som efter en bestämd algoritm frågar noderna i nätet om de önskar överföra data.

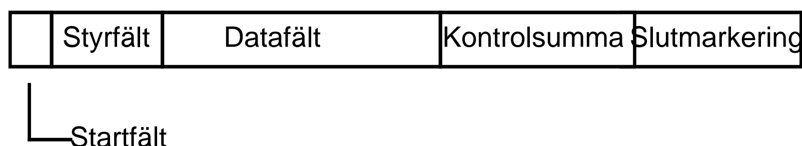
Med CSMA (Carrier Sense Multipple Access) kan vilken nod som helst börja sända om ingen annan överför data. Detta ger oftast upphov till kollisioner på nätet då två noder börjar sända (ungefär) samtidigt och därför kan man inte *garantera* att en viss nod får sitt meddelande överfört inom en viss tid. *Ethernet* använder denna princip.

Vid TDMA (Time Division Multiple Access) finns det någon gemensam tid i systemet som är indelat i lämpliga tidsluckor. Varje nod i systemet tillåts att sända i förutbestämda tidsluckor. Här kan inga kollisioner förekomma och man kan därför garantera att en nod får sända data inom en viss tid. Nackdelen är att en nod som önskar skicka mycket data måste dela upp detta på olika tidsluckor och invänta att tidsluckorna till alla andra av nätets noder har passerat oberoende av om dessa har behov av att skicka data eller inte.

Med "token"-principen så cirkulerar detta (token) som ett *objekt* från nod till nod i nätet. Endast den nod som för tillfället *har token* tillåts att skicka data. En nod som inte önskar skicka data skickar direkt "token" vidare. På så sätt blir väntetiderna mindre än vid TDMA. Nackdelar med detta är att om "token" försvinner, orsakat exempelvis av störningar på nätet, så krävs komplexa algoritmer för att generera en (och endast en) *ny* "token" i nätet. IBM har utvecklat ett ringnät (Camebridge Ring) som använder *token-ring* principen. Nätet uppvisar goda prestanda på såväl överföringskapacitet som tidskrav.

För att säga något om överföringshastigheter kan vi nämna att ett typisk *Ethernet* så som det används i lokala datanät har en överföringshastighet omkring 1-10 Mbits. En datakabel över (eller rättare sagt i) Atlanten, till USA, har en kapacitet på 64 kbit/s. Ett modernare höghastighetsnät kallat *TAXI* har en överföringshastighet på 300 Mbits/s.

Ett typisk datameddelande som överförs i ett datanät visas i figur 6.33. Meddelandet (eller blocket) består av *startfält*, *styrfält*, *datafält*, *kontrollsumma* och *slutmarkering*. Beroende på typ protokoll kan vissa av fälten utelämnas. Syftet med de olika fälten beskrivs nedan.



FIGUR 6.33 UPPBYGGNAD AV EN "FRAME"

- **Startfältet** används för att indikera början av ett datablock och att synkronisera mottagaren med sändaren. Beroende på vilka

synkroniseringsmekanismer som finns mellan sändare och mottagare så kan detta fältet innehålla en eller flera bitar.

- **Styrfältet** anger exempelvis hur stort datafältet är (detta kan ha variabel storlek). Vidare innehåller styrfältet antingen någon form för "till-adress" eller "från-adress" eller både och. I vissa datanät används inte adresser, utan man anger typen av data som överförs istället. Typisk storlek på kontrollfältet är från några bitar till ett tiotal bytes. Detta fält kan även innehålla handskakningsinformation, exempelvis "jag förstod inte det du skickade till mig senast - var god sänd om", eller "jag förstod och har utfört det du begärde senast" osv.

- **Datafältet** överför data och dess längd kan variera från ett fåtal bytes till någon kbyte. I vissa sammanhang kan datafältet saknas. I så fall indikerar styrfältet i blocket någon speciell information

- **Kontrollsumman** beräknas av sändaren utifrån det överförda datablocket. Mottagaren kontrollerar att denna kontrollsumma stämmer. På så sätt kan mottagaren vara säker på att erhållet data har mottagits korrekt. En ofta använd kontrollsumma är *Cyclic Redundancy Check* (**CRC**) som är i storleksordningen några byte. Det finns även kontrollsummor som innehåller information om vilken eller vilka bitar i ett datablock som är felaktigt överförda . På så sätt kan mottagaren eventuellt korrigera för överföringsfel.

- **Slutmarkeringen** används för att indikera att datablocket är slut.

Slutligen bör vi belysa begreppet *överföringskapacitet*. Med detta menas *hur mycket data per tidsenhet* som kan överföras. (Oftast diskuterar man samtidigt till vilket pris.) Allmänt kan man, om serie kommunikation, säga att det är ett billigt sätt att överföra mindre mängder av data förhållandevis snabbt.

Vad vi måste ha i åtanke är att inom mindre geografiska områden, ett kontor eller en industri, där man har tillgång till ett lokalt datanät, kan man med dagens teknologi överföra förhållandevis stora datamängder till en rimlig kostnad. Även att överföra några kilobyte data mellan Göteborg och Stockholm är i dag billigt och det går dessutom snabbt om vi exempelvis förutsätter att överföringstakten är 1 MHz. Att överföra stora mängder data, exempelvis 100 Gbyte, kommer dock att ta mycket lång tid eftersom datanäten, i dag, inte är anpassade för detta.

I följande exempel är det troligt att det mest kostnadseffektiva och snabbaste överföringssättet är att transportera data via databand (kassetter) exempelvi med bil eller tåg.

100 Gbyte skall överföras från Göteborg till Stockholm på en linje med klockfrekvensen 1 Mhz:

$$t = \frac{ant.bitar}{f} = \frac{100 \cdot 10^9 \cdot 8}{1 \cdot 10^6} = 8 \cdot 10^5 s \approx 9{,}3 dagar$$

6.5 Sammanfattning

Vi har visat hur ett datorsystem kan vara uppbyggt av *processor*, *minne* och *in-/ut- enheter*. In- och utenheter kan exempelvis bestå av enkla register anslutna till processorns bussar. Vi såg också exempel på olika programkonstruktioner för hantering av in- och utmatning.

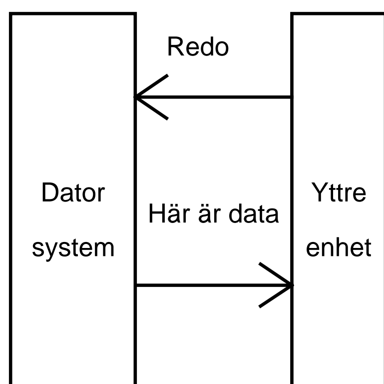
Vi diskuterade olika typer av överföring mellan ett mikrodatorsystem och dess omgivning och exemplifierade denna med en skrivarport. Vi redogjorde för *ovillkorlig* såväl som *villkorlig* överföring med olika typer av handskakning.

Principen för ovillkorlig överföring är att sändaren (datorsystemet) skickar data utan att överhuvud taget bry sig om mottagaren (skrivaren) är redo att ta emot data medan vid villkorlig överföring används handskakning för att både sändaren och mottagaren skall vara överens om överföringen. På så sätt synkroniseras sändare och mottagare till en gemensam överföringstakt.

Vi diskuterade olika sätt att synkronisera enheterna till varandra. Vi beskrev *busy wait* och *polling* där "busy wait" innebär att rutinen, för exempelvis en utskrift, exekveras oavbrutet tills hela utskriften är klar. Detta medför att processorn låses för utskriftsrutinen utan att kunna utföra andra uppgifter under tiden utskriften pågår. I de flesta system är detta oacceptabelt.

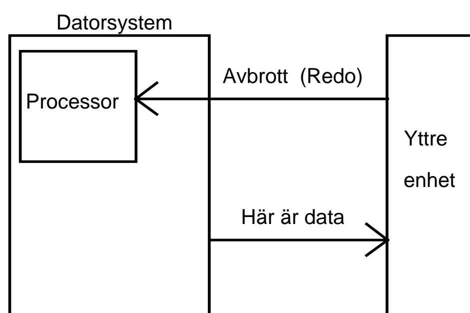
"Polling" innebär att processorn med jämna mellanrum undersöker om skrivaren är redo att ta emot ett nytt tecken. På så sätt låses inte processorn till utskriftsrutinen som vid "busy wait" utan kan utföra andra uppgifter samtidigt som skrivaren är upptagen med att skriva ut tecken. Tyvärr kan detta också visa sig vara ett ineffektivt arbetssätt eftersom processorn ofta kommer att undersöka den yttre enheten väldigt många gånger i onödan.

AVBROTT



FIGUR 7.1 *DATORSYSTEM OCH YTTRE ENHET.*

I kapitel 6 behandlade vi hur ett datorsystem kommunicerar med sin omvärld. Speciellt såg vi exempel på hur man kan synkronisera mikroprocessorns höga arbetstakt med den förhållandevis långsamma omvärlden. Vi använde två olika metoder, *busy wait* respektive *polling* (jämför med figur 7.1). Båda dessa metoder kännetecknas av att processorns tvingas utföra "onödigt arbete", i första fallet genom att en test-slinga utförs ända tills den yttre enheten är beredd, i det andra fallet genom att processorn, med jämna mellanrum testar om den yttre enheten är beredd.



FIGUR 7.2 *DATORSYSTEM OCH YTTRE ENHET SOM BEGÄR AVBROTT*

Det finns emellertid en helt annan metod som kan användas för att synkronisera datorsystemet med omvärlden. Vi låter den yttre enheten leverera en speciell *avbrottssignal* (*interrupt request*) för att påkalla uppmärksamhet. Då processorn uppfattar denna avbrottsignal utförs en speciell program-sekvens för att betjäna den yttre enheten (jämför figur 7.2). Avbrottssignalen får då processorn att avbryta sin normala exekvering och utföra en *avbrottsrutin* som ger den yttre enheten service (exempelvis skriver ut ett nytt tecken till skrivaren) för att sedan återgå till normal programexekvering. Detta förlopp kallas att *betjäna* avbrottet.

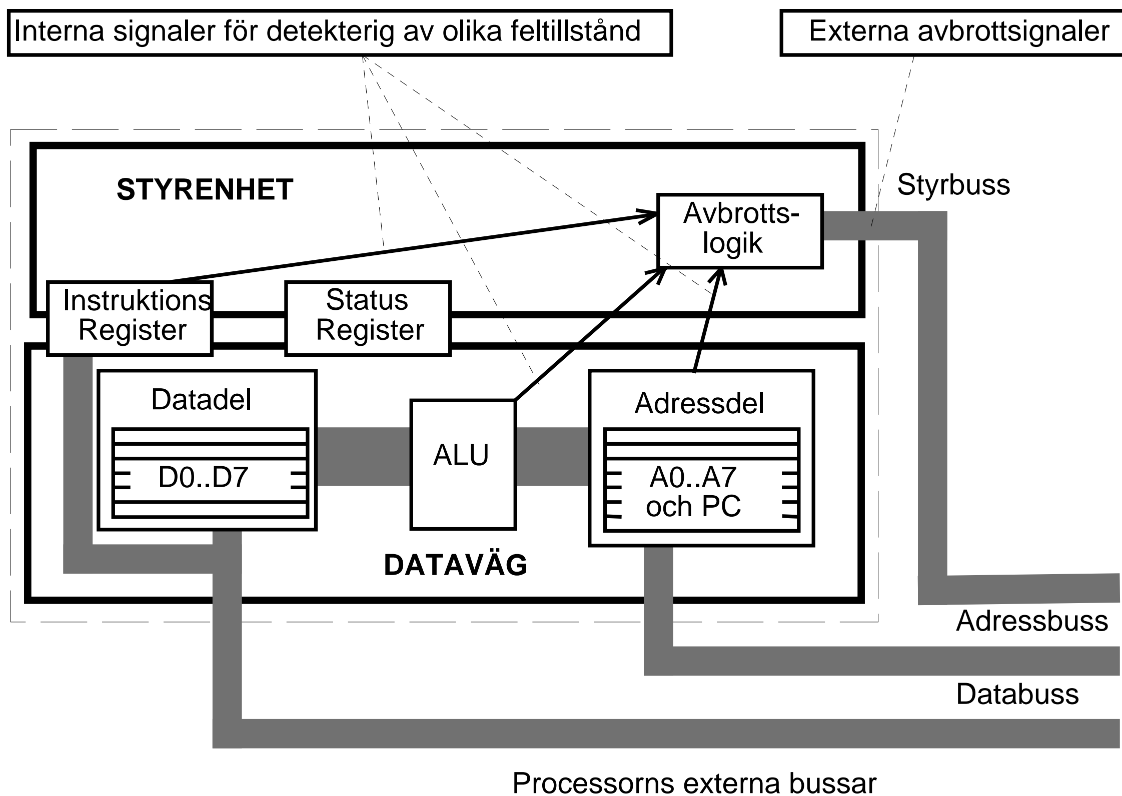
Olika typer av processorer kan ha från en upp till 32 olika avbrottsingångar. Ofta förses dessa med olika prioritet och man talar om *avbrotts-prioritetsnivå.*

Avbrott innebär därför att processorn varken blir låst till en yttre enhet som vid busy-wait eller måste fråga i onödan som vid polling. Detta är en av de största fördelarna med avbrott. Däremot kostar det lite extra när växlingen mellan normal programexekvering till avbrottsbetjäning sker. Innehållen i flera av processorns register måste sparas (jämför med subrutinanrop), PC måste ges ett nytt värde (adressen till avbrottsrutinen) etc. Detta kallas *context switch.* Då avbrottsrutinen utförts måste programexekveringen återupptas från exakt det tillstånd det blev avbrutet, dvs registervärden måste återställas igen. För varje betjänat avbrott har vi alltså åtminstone två "context switches".

En avbrottssignal skickas alltså direkt in till processorn. I kapitel 4 beskrev vi hur en processor var uppbyggd av dataväg och styrenhet, och, styrenhetens uppgift är bland annat att känna av olika avbrottssignaler. En mycket speciell typ av avbrott (*interna avbrott*) uppstår när fel upptäcks inne i processorn. Vi har tidigare, i kapitel 5, behandlat dessa under namnet *undantag* (*exceptions*), exempelvis felaktig operationskod, när processorn försöker dividera med noll,

felaktig adressbildning, mm. Enklare processorer saknar ofta interna avbrottsmöjligheter medan de kraftfullare har försetts med ett antal interna avbrott.

Figur 7.3 nedan illustreras hur avbrottslogiken kan placeras i processorns styrenhet.

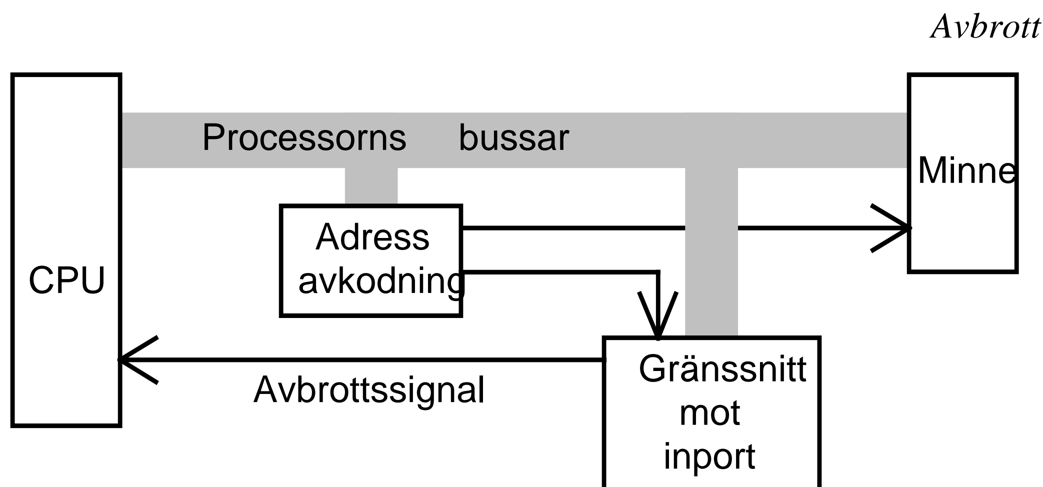


Figur 7.3 *Exempel på interna block*

Fortsättningsvis, i detta kapitel, behandlar vi *externa avbrott*, dvs avbrott som genereras genom att en speciell avbrottsignal aktiveras på processorns styrbuss.

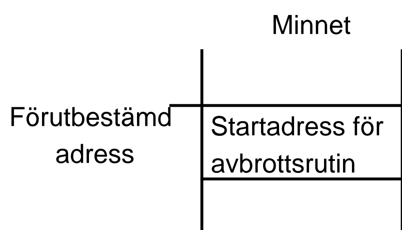
7.1 Generellt avbrott, en avbrottskälla.

Figur 7.4 visar ett datorsystem med minne och en yttre enhet (en inport) som kan generera en avbrottsignal till processorn. Den yttre enheten signalerar då till processorn att ett nytt tecken finns klart att läsas.



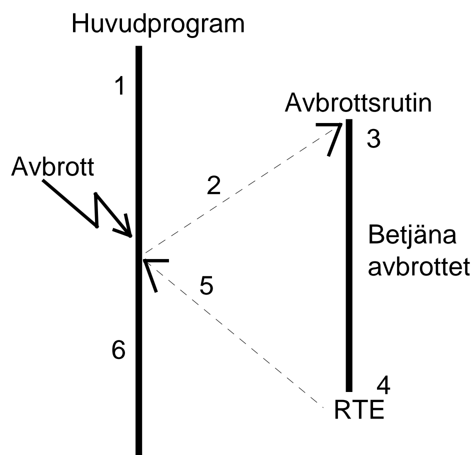
FIGUR 7.4 OLIKA PERIFERIKRETSAR ANSLUTNA TILL PROCESSORNS AVBROTTSINGÅNG

Processorn avbryter då sin normala programexekvering och startar en speciell avbrottsrutin. Denna avbrottsrutin läser data från inporten och återupptar därefter den normala programexekveringen. Observera att processorn nu endast befattar sig med inporten först när avbrottssignalen kommer och det är dags att läsa in data. Jämför detta med principerna för pollning och busy-wait där processorn är sysselsatt med att undersöka om den yttre enheten *önskade* att överföra data.



FIGUR 7.5 AVBROTTSVEKTOR

Hur lokaliseras då avbrottsrutinen för en speciell avbrottsignal? Till varje enskild avbrottsignal associeras en *avbrottsvektor*. Avbrottsvektorn har en förutbestämd adress i minnet och avbrottsvektorns innehåll tolkas som startadress för avbrottsrutinen. Se figur 7.5 som visar en avbrottsvektor i minnet. Programmet måste måste själv skriva in startadressen för avbrottsrutinen (initiera avbrottsvektorn) på den förutbestämda adressen innan det första avbrottet initieras.



FIGUR 7.6 AVBROTTSHANTERING

Förloppet för ett avbrott visas i punktform nedan och i figur 7.6. Vi har här förutsatt att en korrekt avbrottsvektor har initierats.

1. Huvudprogram exekveras när ett avbrott aktiveras
2. Hopp till avbrottsrutin
3. Avbrottsrutin startar
4. Avbrottsrutin avslutas med en speciell instruktion, *return from exception* (RTE)
5. Återhopp till huvudprogram
6. Huvudprogrammet fortsätter.

Eftersom ett avbrott kan inträffa i princip när som helst kan man heller inte i förväg veta vilken del av huvudprogrammet som exekveras då avbrottet uppträder. Detta medför att när återhoppet från avbrottsrutinen ska utföras så måste det finnas information om var huvudprogrammet blev avbrutet. Processorn sparar därför återhoppsadresen på stacken, precis som vid subrutinanrop. Vid subrutinanrop är det en instruktion (JSR eller BSR) som får processorn att spara återhoppsadressen på stacken och i avbrottssammanhang är det en hårdvarusignal (IRQ, *interrupt request*) som aktiverar förloppet.

En ytterligare komplikation är innehållet i processorns register då den blir avbruten. Studera följande exempel som belyser problematiken

EXEMPEL

```
* Del av ett huvudprogrammet
        ...
       MOVE.L      (summa).L,D0
       ADD.L       (belopp).L,D0
       MOVE.L      D0,(summa).L
```

Om avbrottet inträffar precis innan additionsinstruktionen:

```
       ADD.L    (belopp).L,D0
```

i programsekvensen utförs och om avbrottsrutinen använder sig av register **D0** kommer registerinnehållet att ändras. Detta medför att vår summa kommer att bli fel när denna skrivs sist i programexemplet. Alltså, för att undvika detta, och relaterade problem så måste som regel också registerinnehållen sparas vid avbrott. *Dessa lagras på stacken.*

7.1

Det finns olika strategier beroende på processortyp, vissa processorer sparar automatisk alla registerinnehåll på stacken vid vissa typer av avbrott. Andra processorer sparar endast programräknaren (återhoppsadressen) och innehållet i statusregistret (så att inte aktuell flaggsättning ska gå förlorad).

EXEMPEL

```
* Del av ett huvudprogrammet
        ...
       CMPI.L      #1,D0
       BNE   ...
```

Om avbrottet inträffar precis innan den villkorliga instruktionen (BNE) så kan flaggsättningen komma att ändras av avbrottsrutinen med ett felaktigt programflöde som följd. Vid avbrott sparas därför såväl programräknarens som statusregistrets innehåll på stacken.

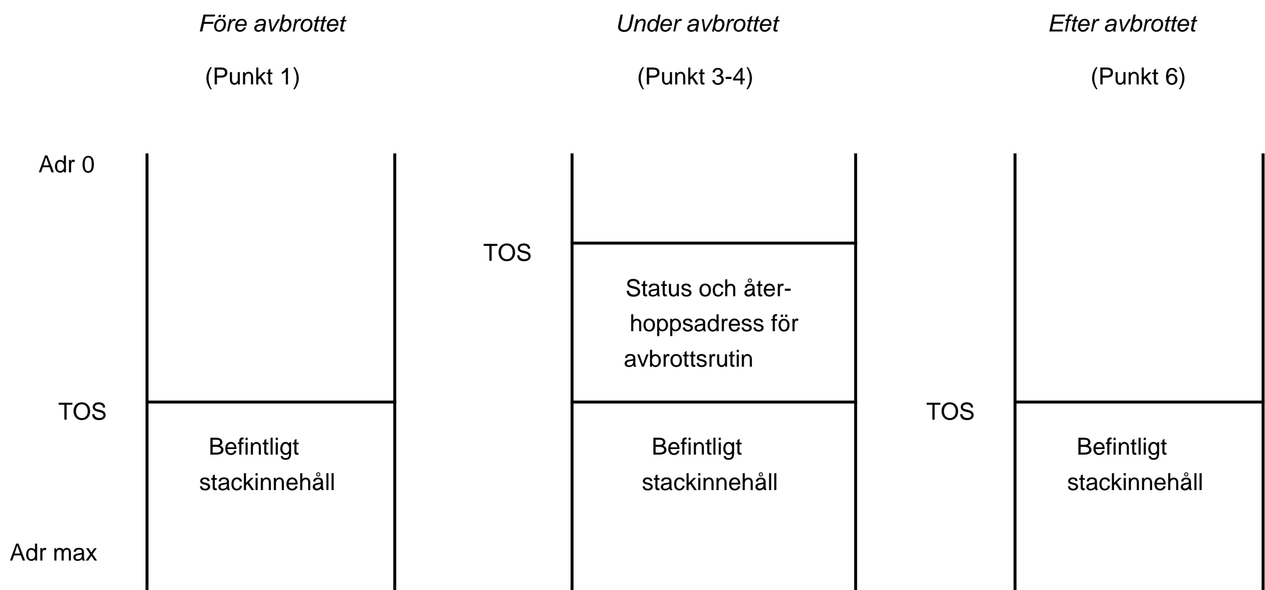
7.2

Vi förutsätter nu att processorn sparar alla registerinnehåll på stacken. I vårt exempel innebär detta att vid punkten 2 i figur 7.6 sparas alla registerinnehåll (*save context*) på stacken och vid punkten 5 återhämtas de (*restore context*).

Vi summerar händelseförloppet vid ett avbrott.

1. Processorn känner att IRQ är aktiverad och slutför utförandet av pågående instruktion.
2. Processorn sparar huvudprogrammets återhoppsadress och övriga registerinnehåll på stacken, *save context*. Därefter läser processorn startadressen för avbrottsrutinen från den förutbestämda adressen i minnet. Denna startadress placeras i PC
3. Avbrottsrutinen startas
4. Avbrottsrutinen avslutas med instruktionen RTE som får processorn att utföra *restore context*, dvs registerinnehållen återställs från stacken.
5. Återhopp till huvudprogram.
6. Därmed återstartas huvudprogrammet där det blev avbrutet.

Figur 7.7 illustrerar stackens utseende och stackpekare (betecknat TOS, *top of stack*) före, under och efter avbrottet.

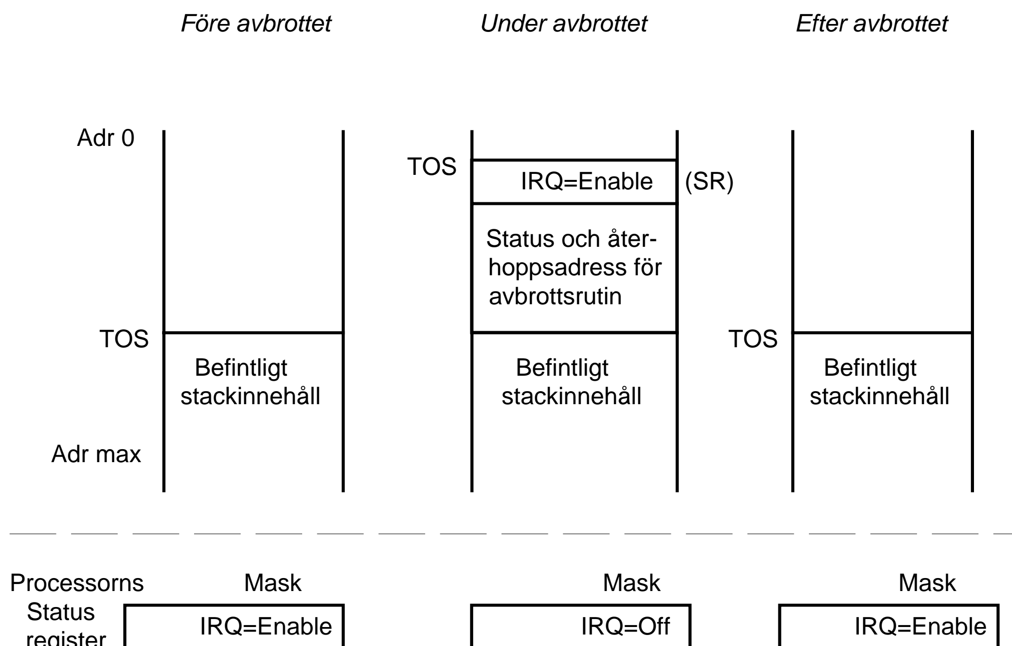


FIGUR 7.7 STACKENS UTSEENDE NÄR AVBROTTSRUTINEN EXEKVERAS

I avbrottsrutinen är det då fullt möjligt att använda processorns register utan att information går förlorad eftersom programmets status är lagrat på stacken.

Vi övergår nu till att beskriva de handskakningsmekanismer som krävs mellan processor och yttre enhet vid avbrott. Den yttre enheten signalerar alltså med en avbrottsignal, IRQ, till processorn. Denna signal måste dock avlägsnas på något sätt under utförandet av avbrottsrutinen annars är den fortfarande aktiv när processorn återgår till huvudprogrammet. Detta skulle leda till upprepade avbrott från samma avbrottskälla. På något sätt måste därför avbrottsrutinen *kvittera avbrottet* mot den yttre enheten så att denna kan inaktivera sin avbrottsbegäran. I bland sker detta automatisk då processorn läser från inporten (alternativt skriver till en utport), vilket den yttre enheten då uppfattar som en kvittens. En annan möjlighet är att processorn skriver i ett speciellt register i den yttre enheten för att kvittera avbrottet.

I processorns statusregister finns ofta flera statusbitar som har en viktig uppgift i avbrottssammanhang. Genom att manipulera dessa bitar kan programmeraren nämligen tillåta eller stänga ute ("maska bort") avbrott, s.k. *avbrottsmask*. Processorn ändrar automatiskt denna avbrottsmask vid avbrottsbetjäning. När börjar utföra avbrottsrutinen (punkt 2 i figur7.6) är fortfarande avbrottsbegäran aktiv från den yttre enheten. Alltså, vid punkten 2 i figur 7.6 *efter* det att processorn sparat registerinnehållen, inklusive statusregistret, så ändrar den sin avbrottsmask i statusregistret. På så sätt så stängs fortsatta avbrott från samma avbrottskälla ute. Vid punkten 5 *efter* det att avbrottsrutinen har kvitterat avbrottsbegäran mot den yttre enheten, återställs registerinnehållen från stacken och på så sätt tillåts nytt avbrott, se figur 7.8.

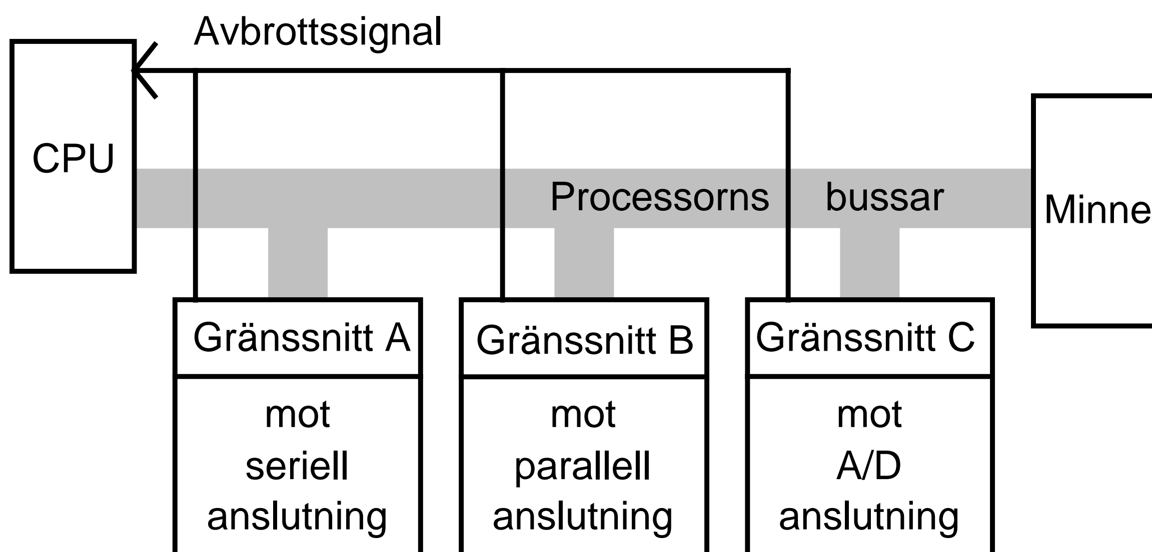


FIGUR 7.8 STACKENS UTSEENDE NÄR AVBROTTSRUTINEN EXEKVERAS

7.1.1 System med flera avbrottskällor

Vi ska nu studera system med *flera* yttre enheter som kan generera avbrott. Betrakta figur 7.9 som visar ett datorsystem med tre enheter (A, B och C) som kan ge avbrott till processorn. Avbrottsignalerna är anslutna till samma insignal på processorn. Kopplingen fungerar så att oavsett vilken yttre enhet som aktiverar sin avbrottsignal så startas samma avbrottsrutin.

Det finns nu inget sätt för processorn att avgöra vilken av enheterna A, B och C som genererat avbrottet. I en sådan koppling måste därför avbrottsrutinen undersöka vilken enhet som genererat avbrottet. Detta utförs exempelvis med polling-teknik.

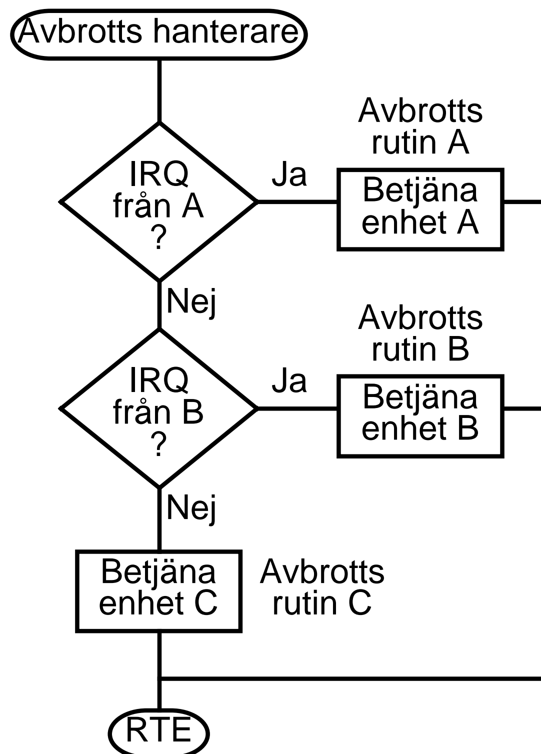


FIGUR 7.9 OLIKA YTTRE ENHETER ANSLUTNA TILL PROCESSORNS AVBROTTSINGÅNG

Vid sådana konstruktioner kallas den programsekvens som undersöker vilken enhet som genererat avbrott för *avbrotts-hanterare* (*interrupt handler*) medan programsekvensen som betjänar en bestämd enhet kalls *avbrotts-service-rutin* (*interrupt service routine*).

Avbrottshanteraren kommer att ge de olika enheterna (A, B och C) en viss inbördes prioritet. Prioritetsordningen beror på den *ordningsföljd* som enheterna undersöks, betrakta figur 7.10 som illustrerar hur enhet A ges prioritet och C lägst. Om avbrott genereras *samtidigt* från exempelvis B och C kommer först B att betjänas eftersom denna undersöks först enligt figuren. I och med att avbrottshanteraren (avbrottsrutinen) startar kommer masken i statusregistret att förhindra *nytt avbrott* tills återhopp från *detta* avbrott sker (när B's avbrottsrutin exekverat klart). När processorn försöker att starta upp huvudprogrammet på nytt känner den av att avbrottsbegäran

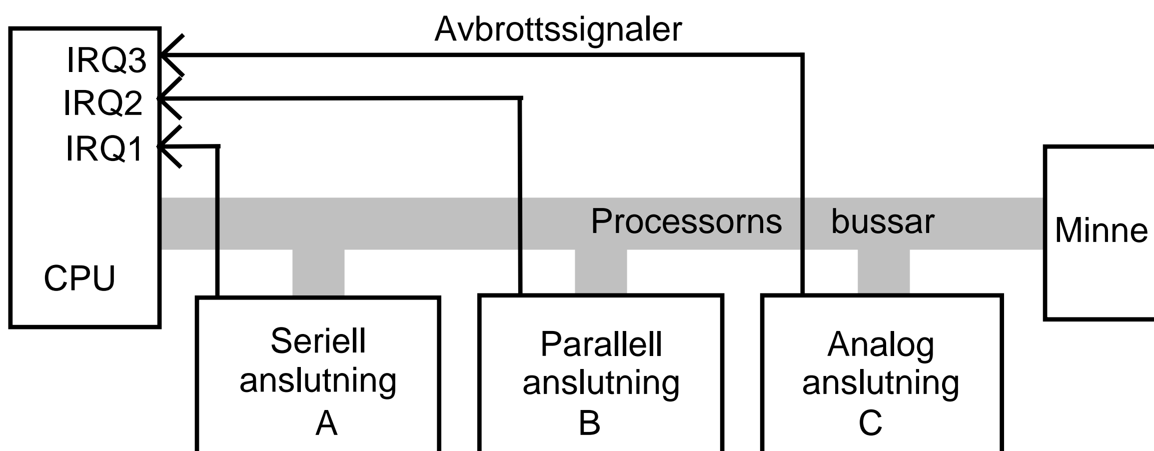
(fortfarande) är aktiv, denna gång från enhet C. Avbrottshanteraren startas på nytt och därmed också avbrottsservicerutinen för enhet C.



Figur 7.10 *Polling i en avbrottsrutin*

De flesta processorer har flera olika avbrottsingångar med inbördes olika prioriteter. I nästa exempel visas en processor med tre avbrottsingångar IRQ1, IRQ2 och IRQ3, där IRQ3 har högst prioritet och IRQ1 lägst.

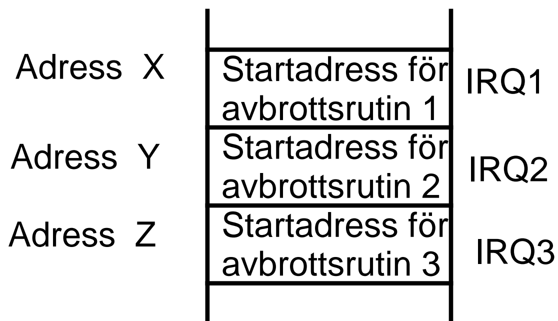
Studera figur 7.11. Kopplingen antyder att ett “avbrott bli avbrutet, dvs att *betjänandet* av ett avbrott på nivå två (IRQ2) kan avbrytas av en avbrottsbegäran på nivå tre (IRQ3) men inte av nivå ett (IRQ1).



FIGUR 7.11 PROCESSORN MED TRE AVBROTTSINGÅNGAR

I system med flera avbrottsnivåer krävs det en tabell med flera avbrottsvektorer. Detta är samma princip som tidigare, skillnaden är att det finns ett antal (tre i vårt exempel) startadresser till avbrottsrutiner lagrade på varandra i minnet.

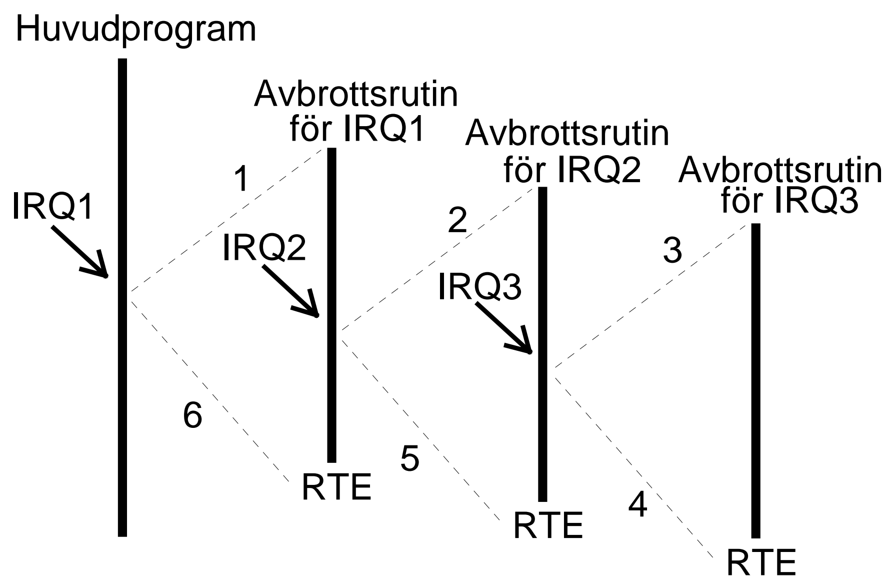
Se figur 7.12, adresserna X, Y och Z är beroende av villken processor vi använder. Olika processorer är utrustade med olika antal avbrottsvektorer och tabellens *(interrupt vector table*) placering kan skilja mellan olika processorfamiljer.



FIGUR 7.12 AVBROTTSVEKTORER

Beroende på vilken av dessa tre avbrottsingångar IRQ1, IRQ2 eller IRQ3 som aktiveras kommer processorn att läsa en av startadresserna till programräknaren och på så sätt starta tillhörande avbrottsrutin. Observera att det är processortillverkaren som väljer och bestämmer adresserna X, Y och Z och att det är programmets uppgift att initiera avbrottsvektorerna med avbrottsrutinernas startadresser.

På samma sätt som när ett huvudprogram blir avbrutet så måste status lagras även när en avbrottsrutin blir avbruten. I figur 7.13 visas vad som sker om tre olika avbrott med olika prioriteter kommer på varandra utan att avbrottsrutinerna har exekverats klart.

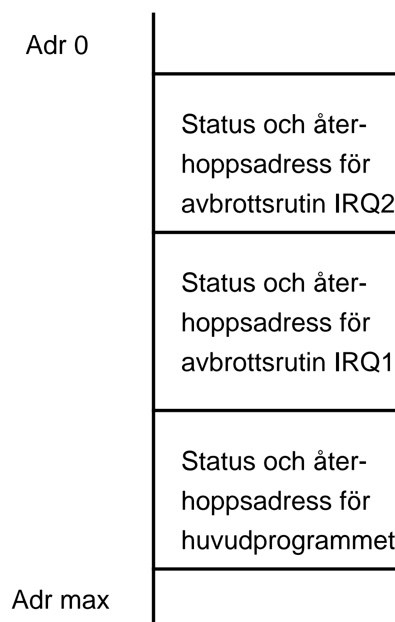


FIGUR 7.13 HUVUDPROGRAM OCH TRE OLIKA AVBROTTSNIVÅER.

Figuren visar att IRQ1 som har lägst prioritet blir avbruten av IRQ2 som i sin tur avbryts av IRQ3 som har högsta prioritet.

Förloppet som visas i figur 7.13 kan beskrivas i punktform:

1. Statuslagring (*save context*) av huvudprogrammet. Maskning av avbrott i SR för nivå 1. Start av avbrottsrutinen för IRQ1.
2. Statuslagring av avbrottsrutinen för IRQ1. Maskning av avbrott i SR för nivå 2. Start av avbrottsrutinen för IRQ2.
3. Statuslagring av avbrottsrutinen för IRQ2. Maskning av avbrott i SR för nivå 3. Start av avbrottsrutinen för IRQ3.
4. Avbrottsrutinen IRQ3 avslutas. Status för avbrottsrutin 2 återhämtas från stacken. Avbrottsrutin IRQ2 återstartas.
5. Avbrottsrutinen IRQ2 avslutas. Status för avbrottsrutin 1 återhämtas från stacken. Avbrottsrutin IRQ1 återstartas.
6. Avbrottsrutinen IRQ1 avslutas. Status för huvudprogrammet återhämtas från stacken. Huvudprogrammet återstartas.



Figur 7.14 *Stackens utseende när avbrottsrutin IRQ3 exekveras*

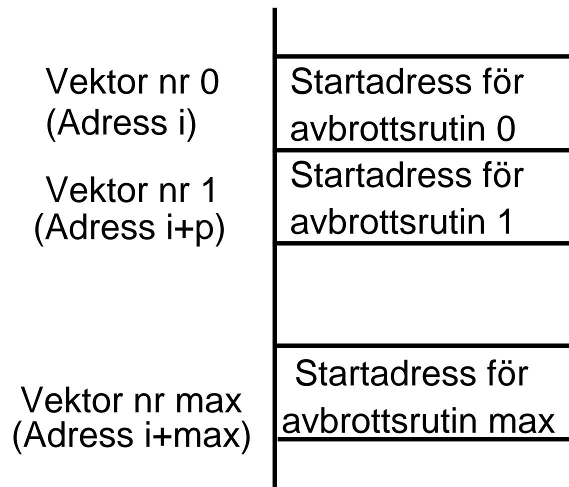
Figur 7.14 visar stackens utseende när avbrottsrutin för IRQ3 exekveras enligt figur 7.13.

På processorer med flera avbrottsingångar brukar den högsta nivån vara icke-maskbar för programmeraren, dvs den inte kan maskas bort i statusregistret, detta kallas *non maskable interrupt*. Denna avbrottsingång på processorn är då flanktriggad och inte nivåtriggad som avbrottsingångar i allmänhet brukar vara.

Vi har nu behandlat en avbrottsmetod som kan användas tillsammans med förhållandevis enkla yttre enheter. Metoden kallas *autovektoravbrott* eftersom avbrottsvektorn är hårdvarumässigt bestämd av avbrottsprioritetsnivån. En nackdel med denna metod är att man som regel, programvarumässigt, måste undersöka vilken enhet, av alla de som kopplats till en specifik prioritetsnivå, som begärt avbrott. Med en annan metod, *vektoravbrott* som beskrivs i nästa stycke slipper man detta, i stället krävs mer komplex yttre hårdvara.

7.2 Vektoravbrott

Principen vid *vektoravbrott* är att varje enkild avbrottskälla också tilldelas en unik avbrottsvektor. Medan autovektoravbrott endast kräver ett fåtal avbrottsvektorer lagrade i minnet (tre i vårt förra exempel) kan det vid vektoravbrott finnas flera hundra avbrottsvektorer. Se figur 7.15.



FIGUR 7.15 AVBROTTSVEKTORER

Detta innebär dock inte att det finns lika många avbrottsingångar, (IRQ1, IRQ2, ...) på processorn. I stället är det den yttre enheten som tillhandahåller ett vektornummer som processorn läser via databussen. Man säger att vid vektoravbrott *identifierar den yttre enheten sig med ett vektornummer.* Processorn översätter detta vektor-nummer till en adress i minnet där processorn läser startadressen till avbrottsrutinen.

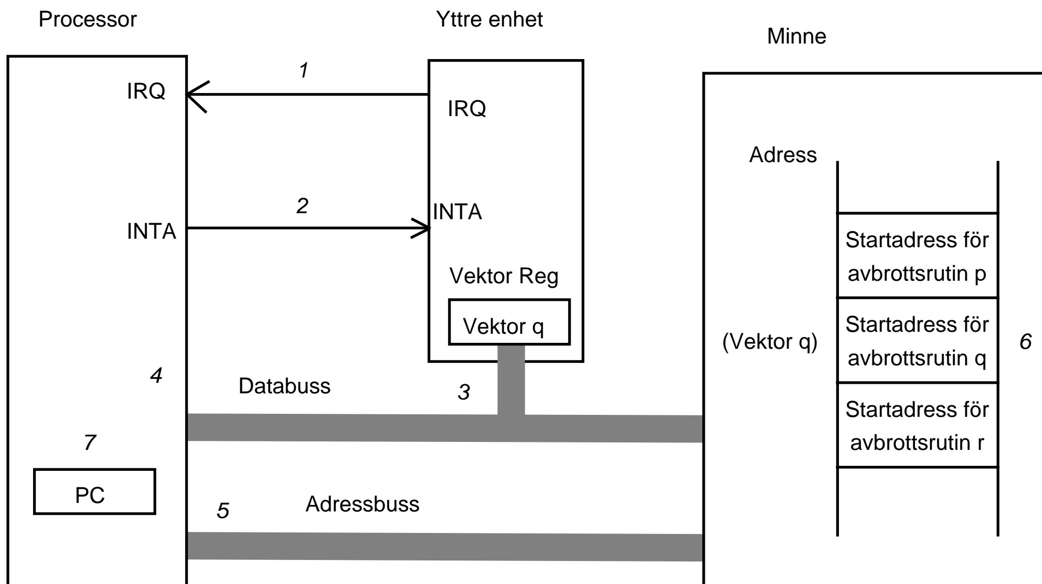
En sådan konstruktion innebär att åtskilliga yttre enheter kan anslutas till en och samma avbrottsingång på processorn. När en av dessa yttre enheterna begär avbrott skickar processorn helt enkelt en speciell signal "INTA" (*interrupt acknowledge*) tillbaka som indikerar:

> *"Jag har sett att någon begär avbrott, var god att identifiera dig med ditt vektornummer!"*.

Den yttre enheten svarar med sitt vektornummer via databussen vilket processorn översätter till en avbrottvektor och därmed också den minnesadress där avbrottsrutinens startadress är lagrad. Se figur 7.16 nedan.

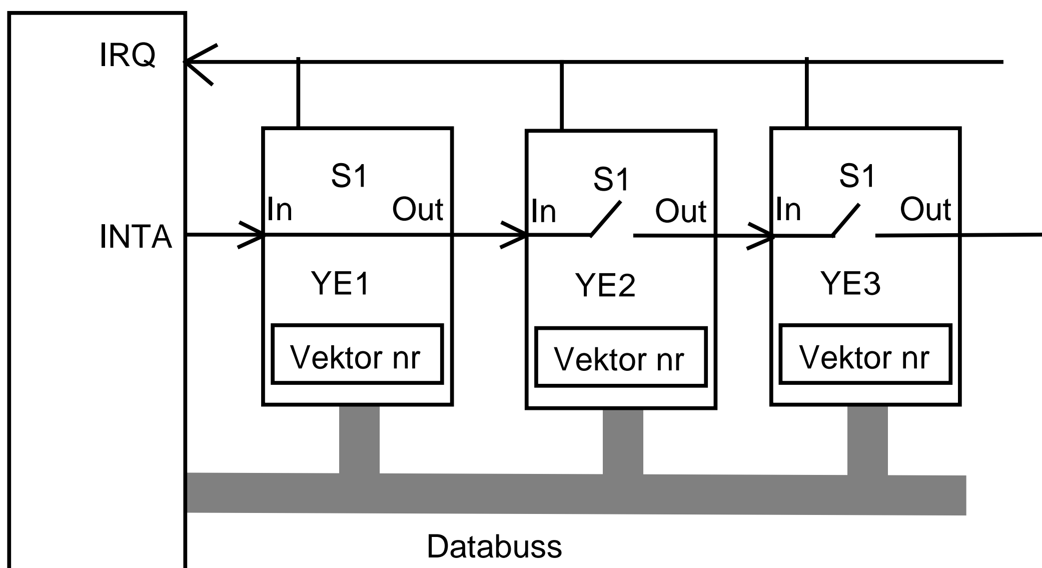
Handskakningsförloppet vid vektoravbrott enligt figur 7.16 mellan processor och en yttre enhet beskrivs i punktform nedan. (Observera att statuslagring utelämnas i detta exempel.)

1. IRQ. Den yttre enheten begär avbrott.
2. Om avbrottsmasken i processorns statusregister (SR) medger avbrott, skickas INTA (annars fortsätter normal exekvering)
3. Den yttre enheten lägger ut sitt vektornummer q på databussen
4. Processorn läser detta nummer och översätter till avbrottsvektor.
5. Processorn adresserar minnet med avbrottsvektorn och.....
6. läser startadressen för avbrottsrutin q som placeras i PC
7. Avbrottsrutinen startas



FIGUR 7.16 PRINCIP FÖR VEKTORAVBROTT

Flera yttre enheter kan alltså vara anslutna till en och samma avbrottsingång på processorn vid vektoravbrott. En frågeställning blir då: Vad händer när två yttre enheter begär avbrott samtidigt och processorn skickar INTA tillbaka? Problemet uppstår då de de två yttre enheterna båda skall lägga ut sitt vektornummer på databussen (punkt 3 i exemlet ovan). Detta får inte förekomma eftersom det skulle innebära en busskrock på databussen. Problemt löses med en mekanism som kallas *daisy chain*. Se figur 7.17 som visar en principskiss av förfarandet.



FIGUR 7.17 VEKTORAVBROTT OCH DAISY CHAIN

Figuren visar hur yttre enhet två och tre (YE2 och YE3) har begär avbrott. Detta innebär att YE2 och YE3 båda har öppnat sina strömbrytare S1. Processorn känner av avbrottsbegäran och utför statuslagring, sätter sin avbrottsmask i statusregistret och svarar med signalen INTA. De yttre enheterna måste vara utrustade med både en ingång och en utgång för INTA signalen. Om den yttre enheten inte har begärt avbrott kopplas ingången direkt till utgången via strömbrytaren S1. På så sätt kommer INTA signalen direkt till yttre enhet 2 som begärt avbrott. Signalen kommer inte till yttre enhet 3 då strömbrytaren är öppen i enhet 2. När YE2 känner INTA signalen placerar den innehållet i sitt vektor nummer register på databussen. Processorn läser detta och startar tillhörande avbrottsrutin.

Avbrottsrutinen för enhet 2 kvitterar avbrottet och avbrottsrutinen avslutas i vanlig ordning. Enhet 3, som ännu inte har fått någon service har fortfarande sin avbrottssignal aktiv. Detta medför att så fort processorn försöker återstarta sitt huvudprogram kommer den att på nytt känna av IRQ signalen och skickar därför en INTA-signal. Nu är strömbrytaren i enhet 2 sluten i och med att den inte begärt avbrott och därför kopplas signalen nu direkt till enhet 3 som nu svarar med sitt vektornummer.

Observera att prioritetsordningen mellan de yttre enheterna nu är bestämd i hårdvara eftersom prioritetsordningen är beroende av hur INTA-signalen är kopplad mellan enheterna.

7.3 Avbrott med MC68000

MC68000 stödjer såväl vektoravbrott (*vectored interrupt*) som vanligt avbrott (*autovector interrupt*). Det finns sju olika avbrottsprioritetsnivåer numrerade från IRQ1 till IRQ7, där IRQ7 har den högsta prioriteten. Alla avbrottsnivåer utom nivå 7, är maskbara i processorn statusregister, IRQ7 kallas därför också *non maskable interrupt*. Varje prioritetsnivå kan konfigureras för antingen autovektor- eller vektor-avbrott. Man kan alltså blanda metoderna för olika prioriteter. Konfigurering för de olika metoderna beskrivs senare i detta kapitel.

MC68000 kan försättas antingen i *användarmod* (*UM, user mode*) eller i *systemmod* (*SM, supervisor mode*) som ibland oskså kallas *kernel* mode. Skillnaden mellan UM och SM märks framför allt i att:

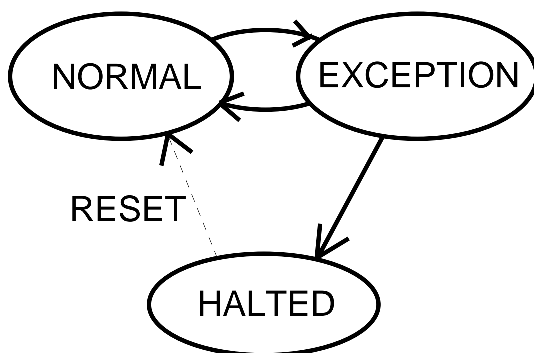
- I SM kan samtliga MC68000 instruktioner utföras.
- Ett antal så kallade *priviligierade* instruktioner, kan *inte* utföras i UM. Om *försök* att utföra sådan instruktion görs då processorn är i UM, kommer speciell *exception- hantering* att vidta.

Införandet av två olika moder, på detta sätt, ger en möjlighet att "bygga in" en *övervakningsfunktion* (*supervisor*) för att skapa stabil programvara som tar hand om de eventuella felsituationer som kan uppträda på grund av felaktiga program etc. Detta förutsätter dock att den programvara som utförs då processorn är i SM noga testats och kontrollerats.

Processorn är dessutom alltid i ett av följande tre tillstånd, nämligen:

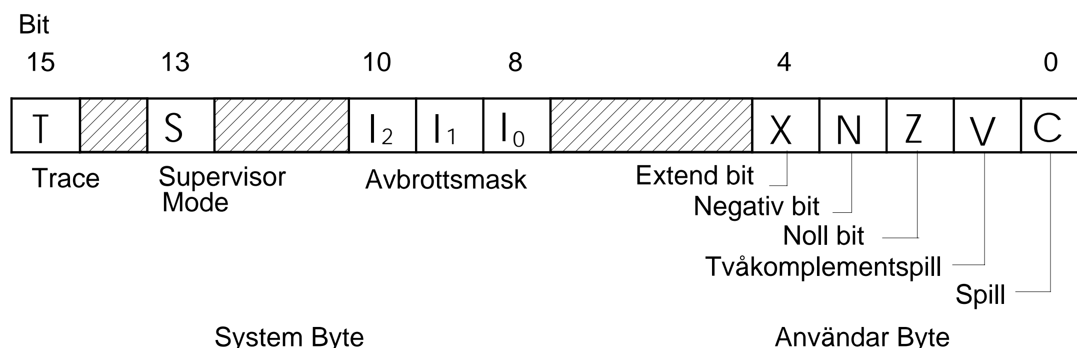
- Normal
- Exception
- Halted.

Se figur 7.18, i tillståndet *NORMAL* kan den vara antingen i UM *eller* i SM men i tillståndet *EXCEPTION* är den *alltid* i SM. Observera skillnaden mellan termerna *mode* och *tillstånd* så som de används här. När processorn utför ett avbrott kan den alltså innan avbrottet vara i antingen UM eller SM. Under själva avbrottssekvensen är den dock alltid i *SM*.



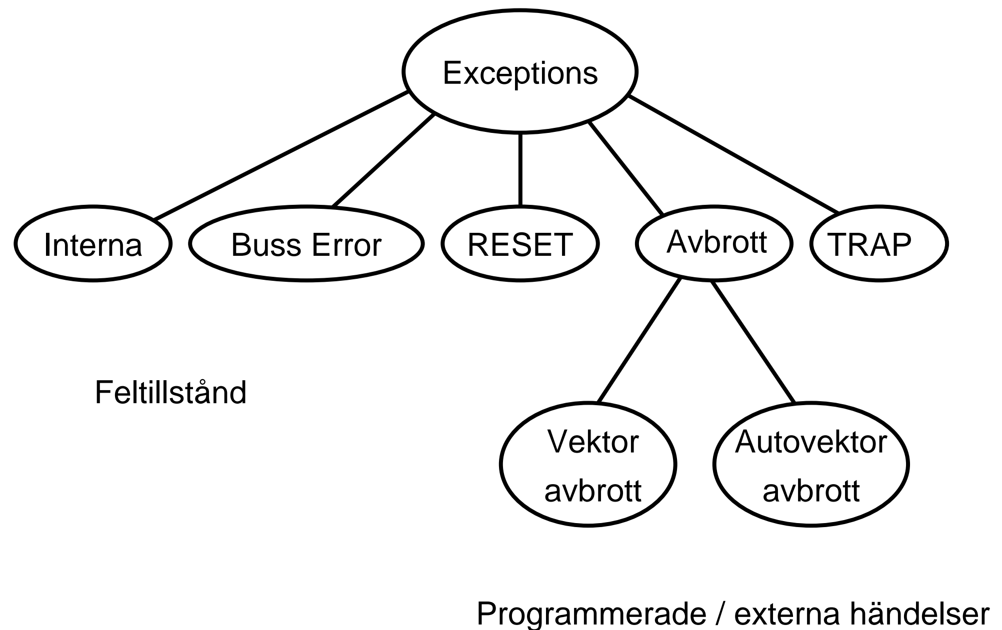
FIGUR 7.18 MC68000:S OLIKA TILLSTÅND

Statusregistret innehåller en bit (S) som indikerar vilken mod processorn är i, se figur 7.19 som även visar avbrottsmasken (I_2,I_1,I_0), vi återkommer till denna längre fram (Jämför även med kapitel 5).



FIGUR 7.19 MC68000 STATUSREGISTER

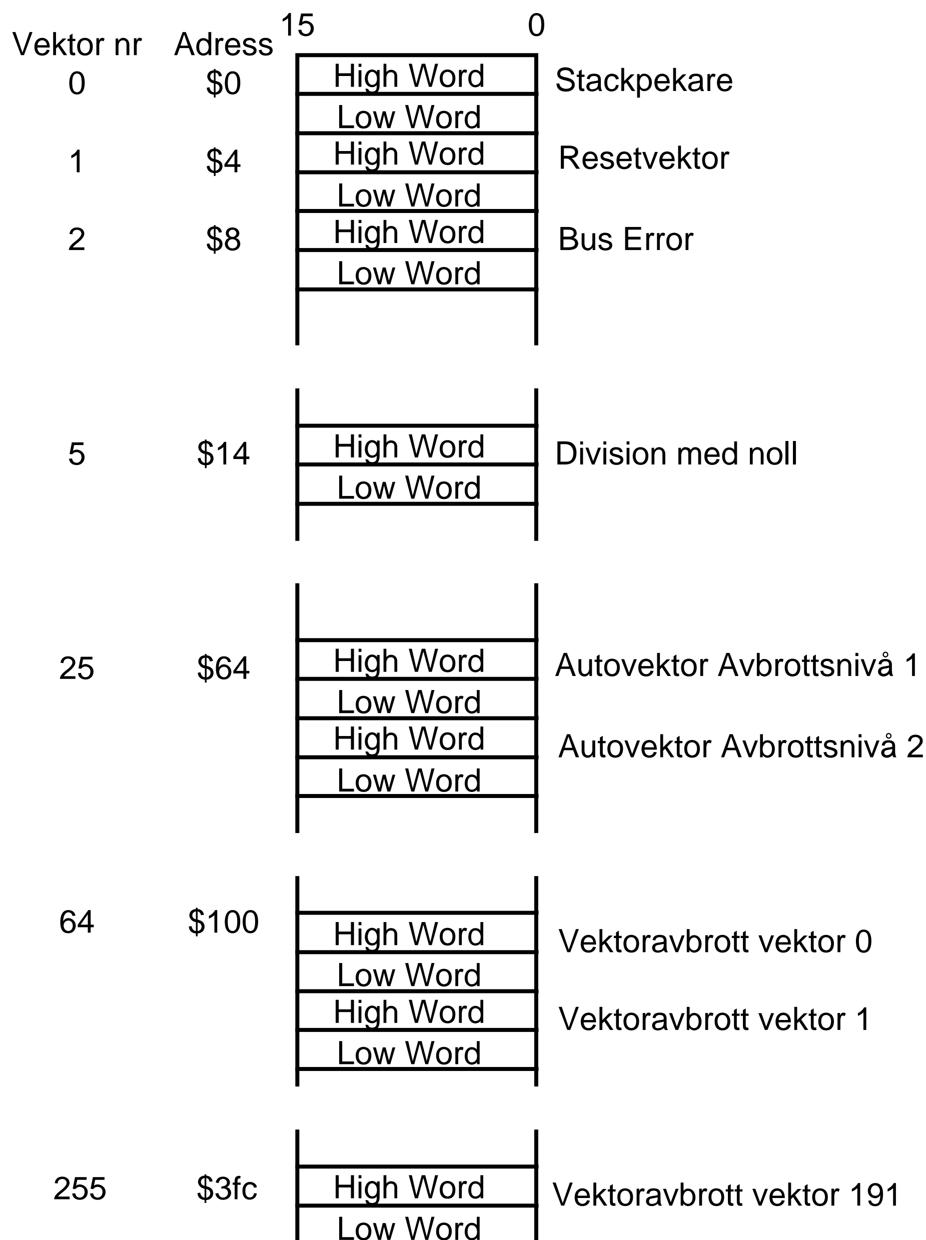
Motorola använder den gemensamma beteckningen *EXCEPTIONS* (*undantag*) för *alla sorters händelser som tar processorn ut ur tillståndet "normal"*. Dessa är indelade i externa och interna händelser, se figur 7.20.



FIGUR 7.20 OLIKA TYPER AV EXCEPTIONS.

Externa händelser är exempelvis avbrott från yttre enheter och externa insignaler som RESET och BUS ERROR. Interna händelser kan genereras av normal programexekvering (exempelvis TRAP-instruktionen) men också av felaktig instruktionsexekvering (exempelvis division med noll, fel adressberäkning, mm).

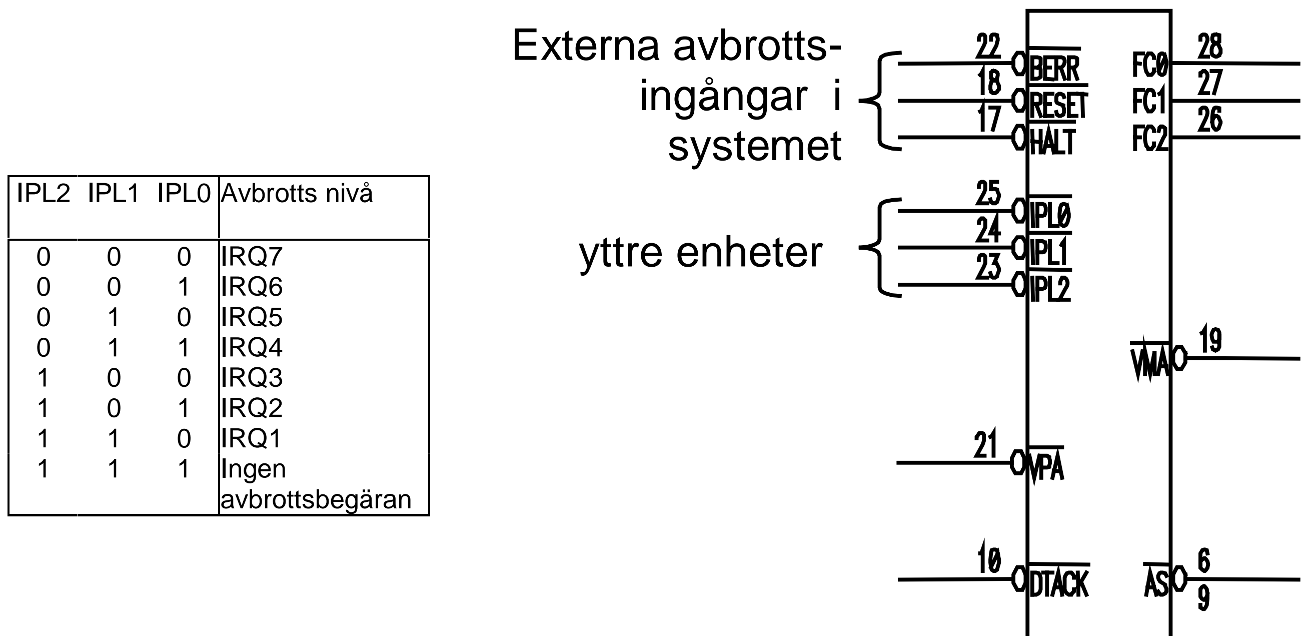
Hos MC68000-familjen finns 256 avbrottsvektorer samlade i *exception vector area*, vilket innebär att det ryms 254 startadresser för avbrottsrutiner, en adress för RESET och slutligen en för en initial stackpekaren vid återstart. Vissa av avbrottsvektorerna används inte av MC68000, dessa är reserverade för efterföljare som *MC68020*, och *MC68030*. Eftersom varje avbrottsvektor är 4 bytes lång innebär detta att hela avbrottsvektorarean är 1 kbyte (se figur 7.21 nedan). Sist i detta kapitel finns en komplett beskrivning av MC68000's vektorarea (*exception vector table*).



FIGUR 7.21 MC68000:S VEKTORAREA.

7.3.1 MC68000 externa avbrottsingångar.

De anslutningar på MC68000 som används för exceptions visas i figur 7.22 nedan. De externa avbrottsingångarna *BUS ERROR* (**BERR**), *RESET*, *HALT* och *INTERRUPT PRIORITY LEVEL* [**IPL2..IPL0**] återfinns överst till vänster i figuren.



IPL2	IPL1	IPL0	Avbrotts nivå
0	0	0	IRQ7
0	0	1	IRQ6
0	1	0	IRQ5
0	1	1	IRQ4
1	0	0	IRQ3
1	0	1	IRQ2
1	1	0	IRQ1
1	1	1	Ingen avbrottsbegäran

FIGUR 7.22 MC68000:S AVBROTTSPINNAR

IPL-pinnarna är anslutna till yttre enheter som serie- och parallellportar och anger vilken prioritetsnivå ett avbrott har. Observera att processorn läser IPL-pinnarna som ett binärtal. En tabell över processorns olika avbrottsnivåer visas i marginalen. Vanligtvis används en 8/3 avkodare för att översätta 8 olika avbrottsignaler till en *interrupt priority level* som anbringas på dessa ingångar

RESET är en dubbelriktad signal. Om denna aktiveras utifrån försätts processorn i återstartstillstånd, MC68000 läser då den så kallade RESET-vektorn (från vektornummer 1 i *exception vector table*) till PC. Vektornummer 0 ger värdet av den initiala stackpekaren som alltså läses till SP. Därefter startas instruktionsexekveringen på den adress som anges av PC. Signalen kan också aktiveras inifrån processorn genom att den utför en RESET-instruktion. Alla övriga kretsar i systemet kommer då att återstartas medan processorn fortsätter med att exekvera nästa instruktion.

BUS ERROR beskrivs detaljerat i ett senare avsnitt i detta kapitel.

Om HALT ingången aktiveras stoppas all instruktionsexekvering och alla buss-aktiviteter. Signalen kan exempelvis användas i multiprocessorkonstruktioner där flera processorer konkurrerar om ett delat minne.

Överst till höger i figur 7.22 visas tre signaler, *Function Codes* [**FC2..FC0**]. Dessa signaler ger en detaljerad information om processorns aktivitet.

Processorn anger, för varje buss-cykel, en specifik adressrymd. Adressrymden bestäms av den mode processorn befinner sig i (SM eller UM) och den typ av åtkomst som utförs (program, dvs instruktion, eller data). Utöver detta finns en speciell adressrymd (CPU Space) som används vid kommunikation med så kallade coprocessorer, dvs minneshanteringskretsar, flyttalsprocessorer mm. Denna adressrymd indikerar också att en avbrottssekvens, vektoravbrott eller autovektoravbrott startar och kan därför tolkas som en *interrupt acknowledge* (IACK) signal.

Följande tabell sammanfattar hur funktionskoderna sätts av MC68000 för olika typer av interna aktiviteter:

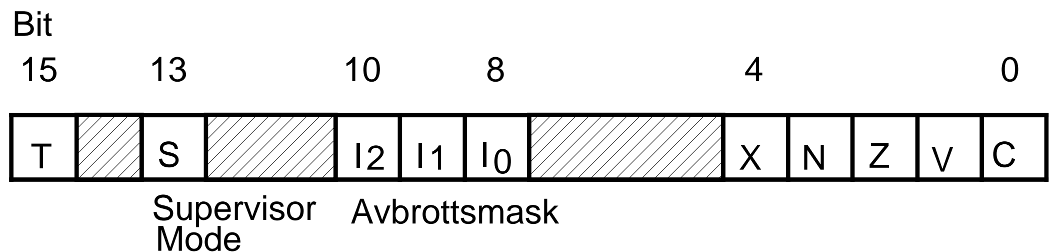
FC2	FC1	FC0	Adressrymd
0	0	0	Reserverad för framtida bruk
0	0	1	User Mode Data
0	1	0	User Mode Program
0	1	1	Reserverad: Användardefinierad
1	0	0	Reserverad för framtida bruk
1	0	1	Supervisor Mode Data
1	1	0	Supervisor Mode Program
1	1	1	Interrupt Acknowledge (CPU Space)

Observera också insignalerna DTACK och VPA. Dessa insignaler indikerar för processorn om den skall utföra autovektoravbrott eller ett vektoravbrott. AS-signalen används som handskakningssignal precis som vid en vanlig läs-cykel när vektornumret skall läsas från den yttre enheten.

7.3.2 Avbrottsprioriteter

För att MC68000 ska acceptera ett avbrott och utföra avbrottshantering krävs att avbrottets prioritet är högre än processorns aktuella prioritet. Medan externa enheters prioritet är bestämd av hårdvarukonfigurationen, kan man programstyrt ändra processorns prioritet. Detta görs genom att sätta avbrottsmasken i processorns SR. (Se figur 7.23 och följande tabell). Studera också de påföljande exemplen som visar hur processorns avbrottsnivå kan ändras.

Prioritet	Trigg nivå	Högsta avbrotts mask i SR för att avbrott ska betjänas	Lägsta avbrotts mask i SR för att avbrott *inte* ska betjänas	När ett avbrott betjänas sätts mask i SR till
7(Högst)	Flank	Betjänas alltid	Betjänas alltid	111
6	Nivå	101	110	110
5	Nivå	100	101	101
4	Nivå	011	100	100
3	Nivå	010	011	011
2	Nivå	001	010	010
1(Lägst)	Nivå	000	001	001



FIGUR 7.23 STATUSREGISTER

EXEMPEL

Instruktionen:

```
ORI    #%0000011100000000,SR
```

sätter processorns avbrottsmask till 111, dvs prioritetsnivå 7 vilket innebär att endast avbrott med prioritet 7 betjänas. Övriga bitar i SR lämnas opåverkade.

Instruktionen

```
ANDI   #%1111100011111111,SR
```

sätter processorns avbrottsmask till 000, dvs prioritetsnivå 0 vilket innebär att alla avbrott betjänas. Övriga bitar i SR lämnas opåverkade.

Instruktionsföljden

```
ANDI   #%1111100011111111,SR
ORI    #%0000010100000000,SR
```

sätter om processorns avbrottsmask till prioritetsnivå 5, oavsett vilken prioritetsnivå processorn hade före instruktionsföljden. Efter detta kommer processorn att betjäna varje avbrottsbegäran med prioritet 6 eller 7.

Observera att instruktioner som påverkar SR endast utförs om processorn är i supervisor mode, dvs bit 13 i SR är 1.

Instruktionen

```
ANDI         #%1101100011111111,SR
```

försätter processorn i *user mode* med avbrottsprioritet 0.
Då processorn försatts i *user mode* kan den återgå till supervisor mode endast genom exception-hantering. (Se även kapitel 5 - Exception hantering).

7.3

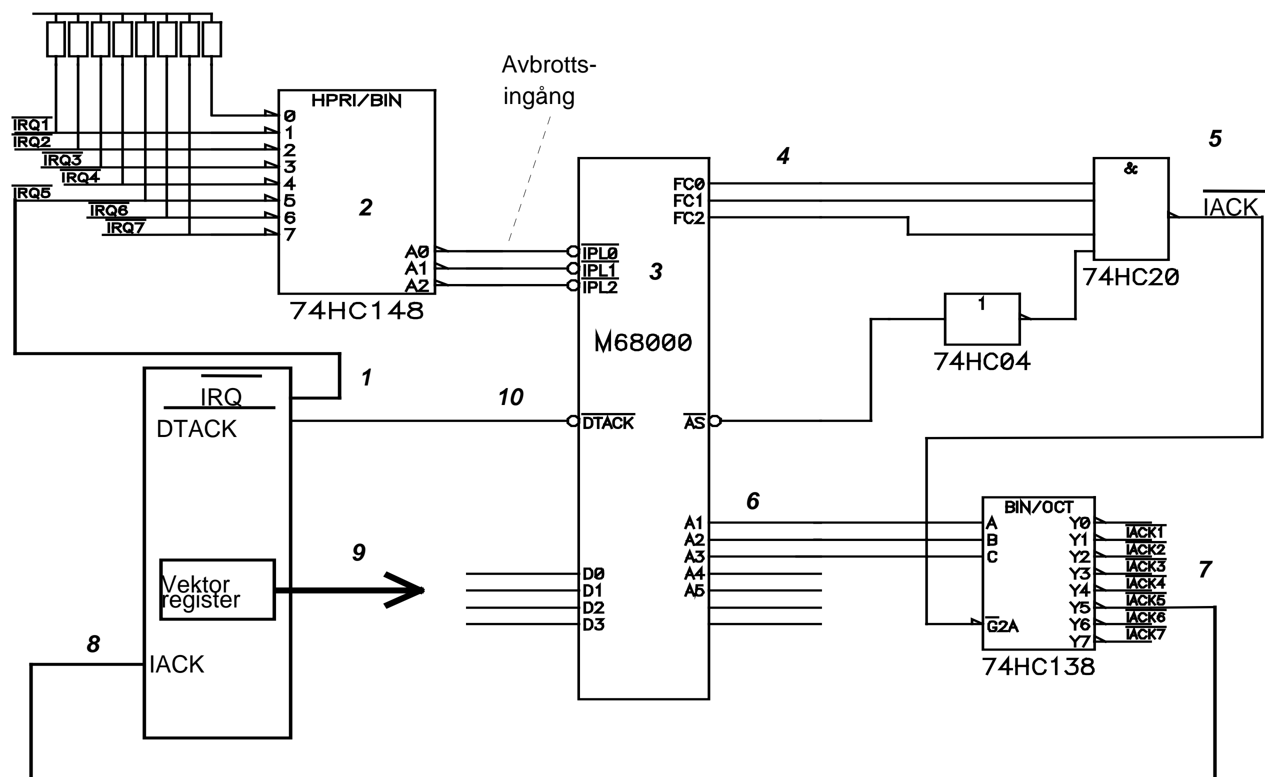
7.3.3 Vektoravbrott med MC68000.

För att MC68000 skall utföra ett vektoravbrott så måste tre villkor vara uppfyllda:

1. En binär avbrottsbegäran (aktiv låg) in till processorn på pinnarna [**IP0..IP2**]
2. Avbrottsmasken i **SR** måste vara lägre än aktuell avbrottsbegäran.
3. Kretsen som begär avbrott måste kunna identifiera sig med ett vektornummer.

Som vi beskrivit tidigare förväntar sig processorn under vektoravbrott att den avbrottsbegärande yttre enheten kan identifiera sig med ett vektornummer som processorn läser. Detta nummer använder processorn för att bilda den adress där startadressen för tillhörande avbrottsrutin är lagrad. Avbrottsrutinen startas och exekveras klart i och med att instruktionen **RTE** utförs.

Figur 7.24 visar en konfiguration för vektoravbrott med en MC68000 och en port. Förutsätt att porten är anpassad till **M680x0** familjens bussystem och skall begära avbrott på nivå fem (**IRQ5**). Förutsätt vidare att portens vektor register innehåller $43. En beskrivning av handskakningsförloppet mellan processor och porten visas nedan. Handskakningsförloppet diskuteras längre fram.



FIGUR 7.24 AVBROTTSSIGNALERING MED MC68000

Eftersom processorns avbrottsingångar [IPL2.. IPL0] aktiveras av ett binärtal som anger avbrottsnivå måste vi externt tillhandahålla en binärkodare, se överst till vänster i figuren. Även signalen IACK måste genereras externt. Den bildas genom att man grindar [FC2..FC0].

Händelseförloppet skildras nu i punktform, se figur 7.24, därefter följer en detaljerad beskrivning:

1. IRQ5 genereras då porten begär avbrott
2. IRQ5 omvandlas till ett binärtal
3. Inkommande avbrottsnivå jämförs med avbrottsmask i **SR**. (Om avbrott tillåts exekverar processorn klart sin instruktion.)
4. [FC2..FC0] sätts till 111
5. IACK genereras
6. Processorn placerar avbrottsnivån (5) på A1, A2, A3
7. IACK5 genereras
8. Porten känner Interrupt Acknowledge,
9. placerar vektornumret ($43) på databussen
10. och indikerar detta med DTACK för att indikera vektoravbrott
11. Processorn multiplicerar vektornummret med 4 (4*$43 = $10C) och läser startadressen för portens avbrottsrutin från adress $10C.

Punkt 1: Figur 7.24 visar att porten (den yttre enheten) begär avbrott. Avbrottsignalen är ansluten till ingång 5 på en prioritetsavkodare (74_HC148) vars utgångar är anslutna till processorns avbrottsingångar [IPL0..IPL2]. På så sätt är porten hårdvarumässigt ansluten till processorns avbrottsnivå 5 (IRQ5).

Punkt 2: När en av prioritetsavkodarens insignaler aktiveras, exempelvis IRQ5 kommer den att leverera bitmönstret 010 (Ingång 5 $\Rightarrow$ 101 $\Rightarrow$ 010 inverterat, ty aktiv låg) på sina utgångar. Interrupt Priority Level pinnarna [IPL0..IPL2] på processorn får nu ett värde skilt från 111. Detta tolkar processorn som att en avbrottsbegäran på nivå fem är aktiverad.

När vi studerar figur 7.24 kan vi ställa oss frågan vad som händer om flera avbrott kommer samtidigt till prioritetsavkodaren, säg IRQ2**,** IRQ3 och IRQ5. Prioritetsavkodaren fungerar så att den negligerar

IRQ2 när IRQ3 aktiveras och dess utsignal blir 100 (observera att signalerna är aktivt låga) för avkodarens utgångar A2, A1 och A0. Om även IRQ5 aktiveras negligeras de lägre nivåerna och kretsens utsignal blir 010. På så sätt kommer alltid den högsta prioriteten att kopplas till pinnarna [IPL0..IPL2] på processorn.

Punkt 3. Processorn registrerar ett värde skilt från 111 på [IPL0..IPL2]-ingångarna. Om nu avbrottsmasken i **SR** är lägre än 5 kommer processorn att acceptera avbrottet efter det att pågående instruktion har exekverat klart. (Vi förutsätter här att avbrottet betjänas direkt.)

Punkt 4. Processorn ettställer *Function Code* signalerna [FC0..FC2] till '111' för att indikera att den accepterar avbrottet, interrupt acknowledge**,** visas som en signal IACK i figuren (**Punkt 5**).

Punkt 6. Samtidigt som Punkt 4 signalerar processorn med adressbitarna [A3..A1] *på vilken avbrottsnivå* som som signalen (IACK) aktiveras för. Processorn har känt av ett avbrott på nivå 5 och kopplingen skall därför returnera IACK5. I vårt exempel har därför adressbitarna nu värdet 101 (5).

Punkt 7. Binäravkodaren (74HC138) avkodar adressbitarna och aktiverar slutligen IACK5 efter det att signalen IACK aktiverats. Observera även att processorns signal AS ingår i bildandet av denna signal. AS indikerar som bekant början på en traditionell buscykel. Detta verkar rimligt då processorn nu skall läsa ett vektornummer.

Vid en första anblick kan det verka konstigt att processorn använder sig av adressbitarna [A3..A1] för att indikera vilken avbrottsnivå som skall behandlas. Det är dock helt i sin ordning eftersom adressbussen saknar traditionell betydelse när processorn skall läsa kretsens vektornummer. Eftersom ingen av signalerna LDS eller UDS aktiveras kommer inte heller någon minneskapsel att adresseras. Motorola har då valt att duplicera funktionen hos dessa adresspinnarna [A3..A1] i stället för att utrusta kretsen med flera pinnar.

Punkt 8. Portens IACK-ingång aktiveras. Detta medför att den placerar innehållet i sitt vektorregister ($43) på databussen (**Punkt 9**). Vidare indikerar porten detta med att aktivera DTACK. Observera att sättet som processorn erhåller räknarens vektornummer på (punkt 8-10) liknar en "vanlig" läscykel bortsett från att nu "adresseras" räknaren med signalen IACK5 istället för det sedvanliga adresseringssättet.

Punkt 10. Processorn som inväntat DTACK läser nu det 8-bitars vektornummret ($43 i vårt exempel) från databussen [D7..D0].

Punkt 11. Innan processorn adresserar minnet för att läsa startadressen för avbrottsrutinen multipliceras vektornummret med 4. Detta medför att processorn adresserar minnet på adress $10C. (Detta är adressen för vektornummer $43. Processorn multiplicerar nämligen vektornummret med fyra för att erhålla korrekt adress i minnet. Detta är logiskt när vi tänker på att varje startadress som lagras i minnet upptar fyra bytes. Se för övrigt figur 7.20 som visar att vektor nummer ett är på adress $004, vektor nummer två på adress $008, osv.) På så sätt har processorn läst in startadressen för avbrottsrutinen.

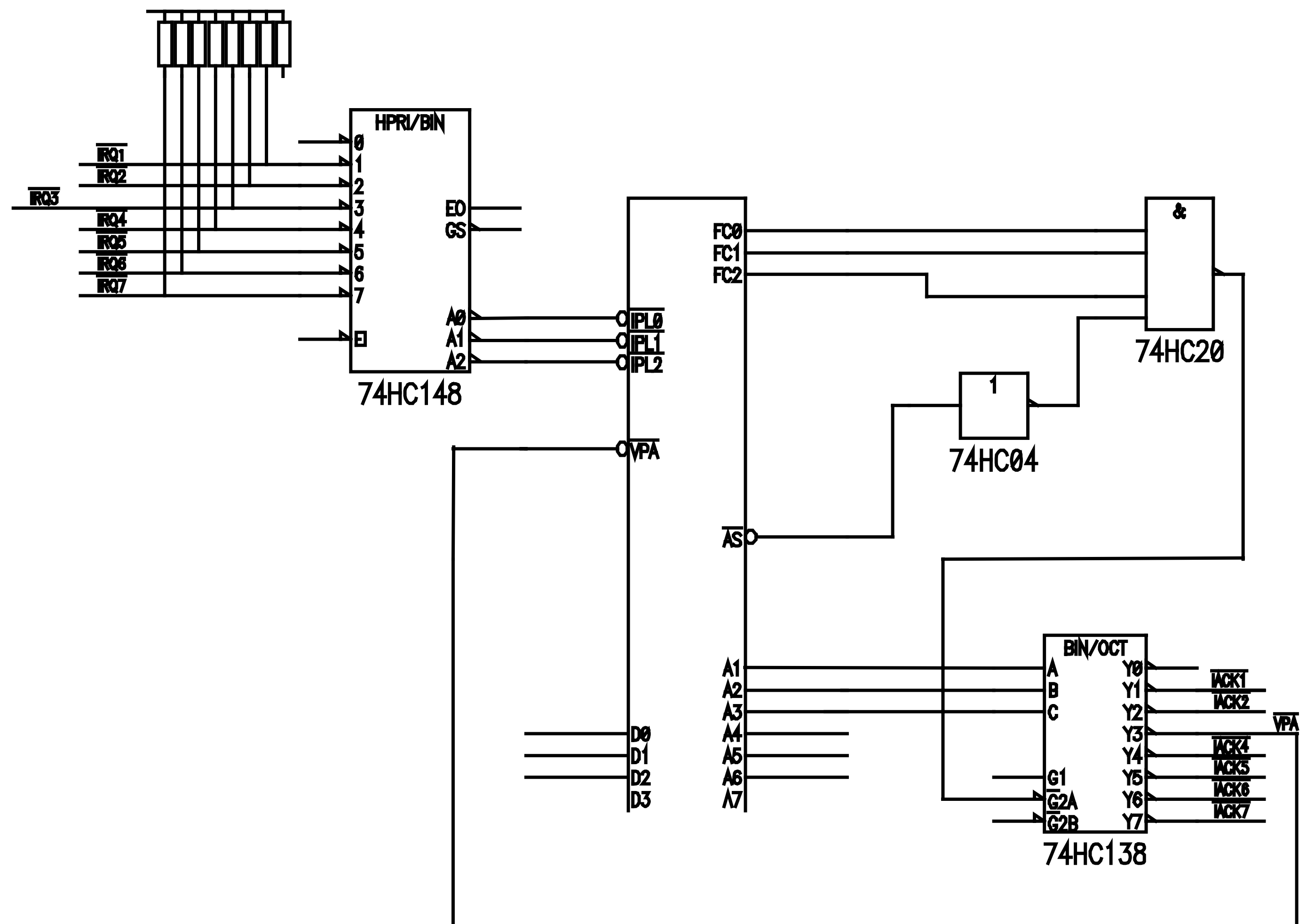
Processorn sparar återhoppsadress och statusregister på stacken. Sedan kommer processorn att ändra avbrottsmasken i SR till nivå fem så att avbrott på nivå fem inte accepteras i fortsättningen. Processorn övergår också till *supervisor mode*, dvs S-biten i SR sätts till 1. Studera figur 7.25.

FIGUR 7.25 SR OCH STACKENS UTSEENDE DIREKT FÖRE OCH EFTER AVBROTTET.

7.3.4 Autovector-avbrott med MC68000

Beteendet vid autovektoravbrott påminner om vektoravbrott. Enda skillnaden är att processorn direkt läser startadressen för avbrottsrutinen utan att först läsa något vektornummer. Detta medför att den yttre enheten (porten) vi ansluter kan vara enklare och behöver följakligen inte vara utrustad med ett vektorregister.

Huruvida processorn skall utföra ett vektor- eller ett autovektoravbrott för en viss prioritetsnivå bestäms vid konstruktionen av systemet, detta visas i figur 7.26 nedan. Observera att vid vektoravbrott genererar IACK-signalen så småningom DTACK till processorn medan i autovektorfallet genereras signalen VPA (*valid peripheral address*) till processorn i stället (jämför med figur 7.24). Figur 7.26 visar hur autovektor avbrott används på avbrottsnivå tre och vektoravbrott för övriga nivåer.



FIGUR 7.26 MC68000 I AUTOVEKTORKONFIGURATION.

Om processorn först känner DTACK efter att den aktiverat IACK så läser den ett vektornummer från databussen, medan om VPA aktiveras först så läser den tillhörande autovektor i minnet. Figur 7.27 visar autovektorarean.

Vektor nr	Adress	15 – 0	
25	$64	High Word	Autovektor Avbrottsnivå 1
		Low Word	
26	$68	High Word	Autovektor Avbrottsnivå 2
		Low Word	
27	$6C	High Word	Autovektor Avbrottsnivå 3
		Low Word	
28	$70	High Word	Autovektor Avbrottsnivå 4
		Low Word	
29	$74	High Word	Autovektor Avbrottsnivå 5
		Low Word	
30	$78	High Word	Autovektor Avbrottsnivå 6
		Low Word	
31	$7C	High Word	Autovektor Avbrottsnivå 7
		Low Word	

FIGUR 7.27 MC6800:S AUTOVEKTORER.

I figur 7.26 ser vi att konstruktionen nu blir betydligt enklare, det krävs heller inte något krångligt handskakningsförfarande med periferikretsen som därför kan vara av enkelt slag. Låt oss illustrera användningen av denna koppling med följande exempel:

EXEMPEL

En inport är ansluten till ett MC68000-system. Porten som är ansluten till avbrottsnivå 3 är avsedd för autovektoravbrott.

I programmets inledning måste följande initiering startadressen för avbrottsrutinen utföras:

```
* Initiering av avbrottsvektor
      MOVE.L           #IrqRut3,($06c).L
```

dvs, exceptionvektor nummer 27 (adress $06C) som är autovektor för prioritetsnivå 3 (se även figur 7.27), initieras med adressen till vår avbrottsrutin (IrqRut3).

Avbrottsmasken i SR måste ändras till en nivå under 3, exempelvis nivå noll. Vi förutsätter här att den tidigare var satt till nivå 3 eller högre (se även figur 7.23).

```
* Sätt avbrottsmask till nivå 0
      ANDI             #%1111100011111111,SR
```

En avbrottsrutin för nivå tre kan se ut som visas nedan.

```
IrqRut3:
       MOVEM.L     D0-D2,-(SP)
       ...
       ...   Betjäna och kvittera avbrottet
       ...
       ...
       MOVEM.L     (SP)+,D0-D2
       RTE
```

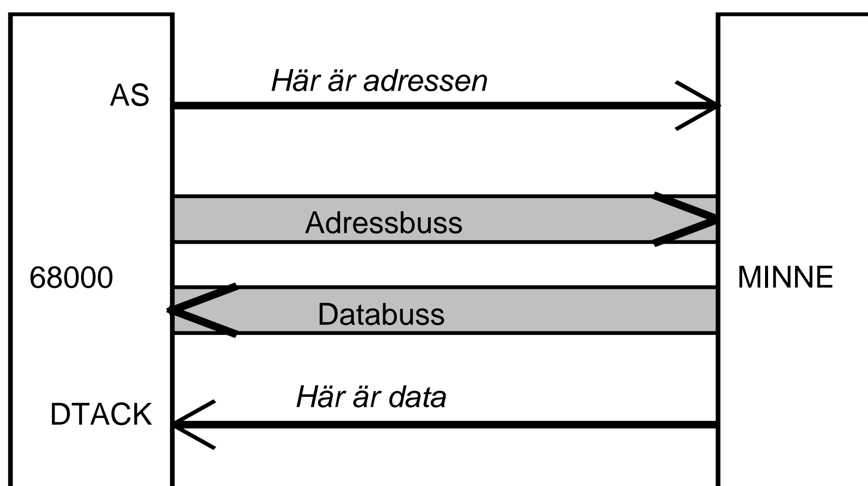
Vi har här antagit att register D0,D1 och D2 används under avbrottsbetjäningen men detta är naturligtvis högst varierande. Vi har också utgått i från att den yttre enheten initierats på ett sådant sätt att den genererar avbrott och att avbrottet kan kvitteras, vi återkommer till detta i nästa kapitel.

7.4

Med detta avslutar vi behandlingen av MC68000's olika avbrottshanteringar i detta kapitel. Vi återkommer med åtskilliga tillämpningar av avbrott i kapitel 8 (periferikretsar).

7.3.5 Buss Error

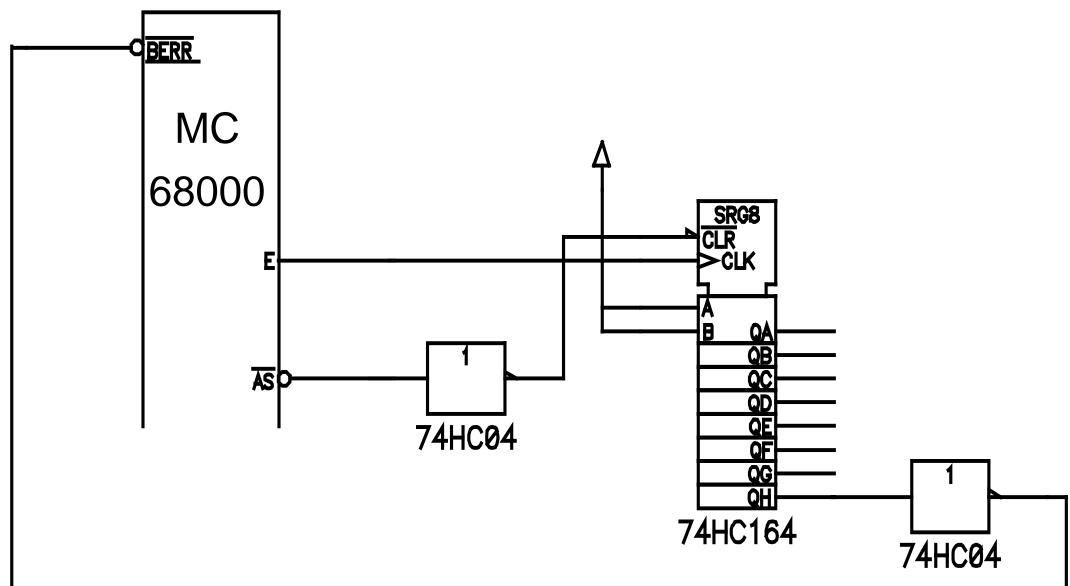
BUSS ERROR är en speciell typ av avbrott som kan användas exempelvis för att upptäcka programmeringsfel. Kortfattat kan man säga att BUSS ERROR uppträder om vi adresserar ett obestyckat adressrum.



FIGUR 7.28 HANDSKAKNING UNDER EN LÄSCYKEL

Handskakningsförloppet för en busscykel är som bekant att processorn först genererar AS och därefter inväntar DTACK. Se figur 7.28 (jämför även kapitel 4). Om ingen DTACK genereras på grund av att inget minne eller yttre enhet finns på denna adress kommer processorn att tvingas vänta i all framtid, systemet stannar helt enkelt.

Man kan därför utnyttja processorns insignal BERR där man ansluter en räknare, se figur 7.29. Processorn AS i början av en busscykel. Denna signal används då också till att nollställa en räknare. Om nu busscykeln termineras korrekt med en DTACK kommer direkt en ny AS att nollställa räknaren på nytt då en ny busscykel startas.

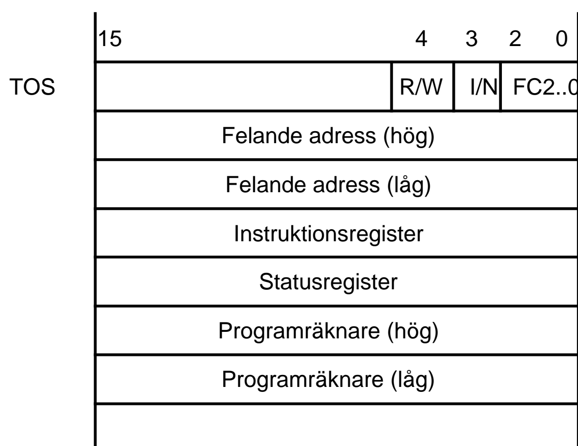


FIGUR 7.29. BUSSERROR-LOGIK.

Skulle AS utebli och alltså indikera att DTACK uteblivit, kommer räknaren att klockas med klocksignalen E från processorn. Så småningom kommer räknares utgång QH, som i sin tur aktiverar processorns insignal BERR att aktiveras. På så sätt kommer processorn att tvingas utföra BUSS ERROR hantering.

Det är nu upp till denna avbrottsrutin att informera användaren om att ett program refererat en felaktig adress. Dessutom kan användaren informeras *var* i programmet felet uppstod. Avbrottsrutinen kan tillhandahålla denna information genom att undersöka innehållet på stacken. Se figur 7.30

MC68000 sparar den felande adressen, vilken typ av instruktion som exekverades, innehållet i statusregistret och programräknaren på stacken. Vidare sparas ett ord till som innehåller information om det var en skrivning eller läsning som utfördes när felet inträffade. Detta ordet innehålller även *processorns Function Code* bitar och information om hurvida en instruktion utfördes eller inte.



FIGUR 7.30. STACKENS UTSEENDE EFTER ETT BUSSERROR.

Med hjälp av informationen på stacken kan alltså en välskriven *exception-hanteringsrutin* för *bus-error* generera en utskrift som gör programmeraren uppmärksam på var, i programmet, felet uppstod.

7.3.6 MC68000 exception vector table

Som avslutning på detta kapitel visar vi MC68000:s alla avbrottsvektorer, figur 7.31 nedan. Samtliga avbrottsvektorer finns samlade i *Exception Vector Table* (Se även kapitel 5 - Exception hantering). Autovektor-avbrott har tilldelats vektornumren 25 till 31. För vektoravbrott från yttre enheter används vektornumren 64 till 255. Övriga vektornummer är fördefinerade för exempelvis *RESET*, *Bus Error*, *Trace*, *Zero Divide* osv. Ett antal är också fördefinerade för att exempelvis kunna stödja flyttalsprocessorer, andra är reserverade för framtida bruk. (Dessa används av de senare utvecklade processorerna i 680x0 familjen, exempelvis MC68030 och MC68040.). Notera också speciellt vektorerna 15 och 24. Dessa används endast för vektoravbrott (saknar mening vid autovektor-avbrott).

Vektor nr	Adress (hex)	Funktion
0	000	Initial stackpekare
1	004	Initial programräknare
2	008	Bus Error (ex: referens till adress där minne/periferi ej finns)
3	00C	Adress Error (ex: referens till udda adress med *word* operand)
4	010	Illegal instruktion (icke-definierad operationskod)
5	014	Division med 0
6	018	Trap vektor för instruktionen CHK
7	01C	Trap-vektor för instruktionen TRAPV

8	020	Privilege Violation, försök att utföra *supervisor*-instruktion i *user mode*
9	024	Trace, en-instruktions exekvering
10	028	Line 1010, reserverad operationskod
11	02C	Line 1111, reserverad operationskod
12	030	Reserverad för framtida bruk
13	034	Reserverad för framtida bruk
14	038	Reserverad för framtida bruk
15	03C	Avbrott från enhet som ej tillhandahållit avbrottsnummer
16-23	040-05F	Reserverade vektorer
24	060	Icke-identifierat avbrott
25	064	Autovektor avbrottsnivå 1
26	068	Autovektor avbrottsnivå 2
27	06C	Autovektor avbrottsnivå 3
28	070	Autovektor avbrottsnivå 4
29	074	Autovektor avbrottsnivå 5
30	078	Autovektor avbrottsnivå 6
31	07C	Autovektor avbrottsnivå 7
32-47	080-0BF	Trap vektor för instruktionen TRAP #<vektor_nummer>
48-63	0C0-0FF	Reserverade vektorer
64-255	100-3FF	Användardefinierade vektorer

FIGUR 7.31. AVBROTTSVEKTORER

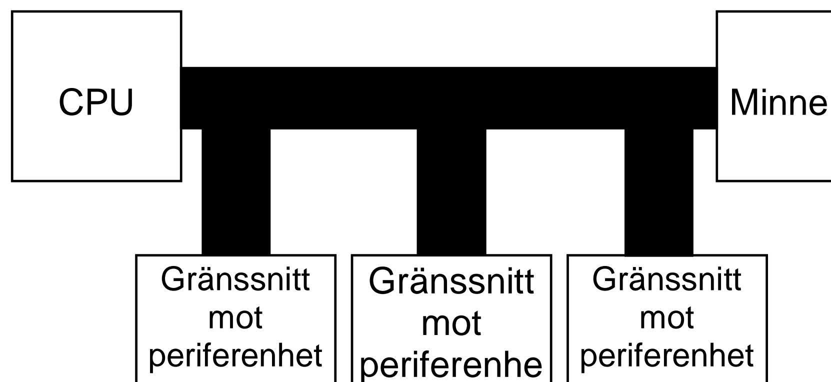
PERIFERIKRETSAR

I detta kapitel fortsätter vi diskussionen kring datorns koppling mot omvärlden. Vi utvecklar denna diskussion till ett exempel där vi använder en periferkrets för att åstadkomma en skrivaranslutning.

Speciellt studerar vi, därefter, MOTOROLAS parallellkrets MC68230 respektive seriekretsen MC68681, dessa beskrivs i något av "databladsform". Ett viktigt syfte med kapitlet är att visa på hur man läser och tyder just datablad.

Tidigare har vi visat hur ett datorsystem är uppbyggt av processor, minne, in- och utenheter. In och utenheter kan vara enkla register anslutna till processorns bussar. Sådana enkla enheter kan dock vara komplicerade att använda i sammanhang där det är nödvändigt med olika former av *handskakning*. Enklare enheter kräver i allmänhet också betydligt mer och framför allt, komplex, programvara för att fungera tillfredsställande. Så länge det är *generella* in och utenheter som önskas är det vanligtvis både billigare och enklare att använda så kallade *periferikretsar* (kringkretsar) som då utgör processorns *gränssnitt* (*interface*) mot omvärlden. Färdiga periferikretsar finns för exempelvis parallell in och utmatning, för seriekommunikation, för anslutning av en flexskivestation, för anslutning av en bildskärm, för datanät, mm.

gränssnitt : interface



FIGUR 8.1 SCHEMATISK BILD: ANSLUTNING AV FLERA PERIFERIKRETSAR

Vi inleder med att diskutera varför periferikretsar används och hur en generell periferikrets ser ut. Vi kommer att visa hur en pereferikrets används som en skrivaranslutning innan vi övergår till en detaljerad beskrivning av parallellkretsen MC68230 och seriekretsen.

8.1 Varför används periferikretsar ?

Periferikretsarna ansluts direkt till processorns buss-system. Den delen som vetter mot processorn ser därför lika ut för alla periferikretsar, medan de delar som ansluts till "omvärlden" kan ha mycket varierande utseende.

Periferikretsar används för att det är ett hårdvarumässigt billigt och enkelt sätt att utrusta datorsystemet med, ofta avancerade, gränssnitt mot omvärlden. Processortillverkare levererar mestadels periferikretsar till sina egna processorer, eller till sin egen processorfamilj. Dessa är då ofta enkla att ansluta till processorns bussar eftersom de normalt är utrustade med de handskakningssignaler som används för processorns busskommunikation.

De billigaste kretsarna kostar från några tior och uppåt och tillverkas i mycket stora upplagor. Dessa är ofta mycket generella och kan programmeras för att användas i olika konstruktioner där det krävs

någon form av parallell eller seriell kommunikation. En konstruktör väljer ofta en generell periferikrets till sin konstruktion och den programmeras därefter för önskad funktion.

Periferikretsar används också av synkroniseringsskäl. Det gäller att synkronisera processorns arbetstakt till omvärlden. För exempelvis en *matrisskrivare*, som skriver ca 100 tecken i sekunden (ett tecken var 10:de millisekund) är i allmänhet processorns arbetstakt alldeles för hög. En millisekund är, å andra sidan, för processorn en väldig lång tid om det bara gäller att mata ut ett enstaka tecken. Syftet med ett *skrivargränssnitt* är att tillhandahålla tecknet som skall skrivas under en tillräckligt lång tid för att den förhållandevis långsamma skrivaren ska kunna läsa det i egen takt. Å andra sidan är det viktigt att processorn inte, i onödan, tvingas vänta på att skrivaren ska hinna med. Oftast finns det någon signal (handskakningssignal) från skrivaren som indikerar att ett tecken är läst och att ett nytt tecken kan läggas in i gränssnittet. De flesta periferikretsar är utrustade med (status-) register där processorn kan utläsa tillståndet på sådana handskakningssignaler.

MOTOROLA har utvecklat en periferikrets som kallas PI/T MC68230. Den är utrustad med tre stycken 8-bitars parallella portar som, oberoende av varandra, kan programmeras för att vara in eller utportar. Det finns också en räknarkrets i periferikretsen som kan användas som realtidsklocka eller för andra former av tidmätningar. Dyrare kretsar, som exempelvis MC68824 innehåller all logik som krävs för att kommunicera enligt Token Bus protokollet, ett mycket komplext protokoll, som används i vissa datanät. När en periferikrets utför ett sådant protokoll säger man att protokollet är implementerat i hårdvara.

8.1.1 En generell periferikrets.

Figur 8.2 nedan visas en bild av en generell periferikrets till någon processorfamilj. Den består av anslutningar mot processorns buss system och mot den yttre enheten (periferibuss). Internt består en periferikrets av *styrlogik*, *beräkningsdel* och *register*.

FIGUR 8.2 EN GENERELL PERIFERIKRETS

Anslutningen mot processorn är anpassad till processorns buss-system och uppdelad på databuss och styrbuss, där styrbussen används för handskakning. Samma sak gäller för periferibussen, ett antal anslutningar på periferikretsen kan användas för handskakning med den

yttre enheten. Databussen på periferisidan kan ha olika bredd (1, 2, 8, 12 bitar) beroende på tillämpning.

Vissa periferikretsar är utrustade med en beräkningsdel exempelvis för att; bestämma udda eller jämn paritet, bestämma CRC-summan på ett överfört datafält eller att omvandla en analog insignal till ett binärt värde.

CR:
Control Register
SR:
Status Register
TxData:
Transmitt Register
RxData:
Receive Register

Periferikretsar är i allmänhet utrustade med statusregister, styrregister (*control register*) och dataregister. Beroende på funktion och komplexitet kan det finnas flera uppsättningar av registren. Status registren han innehålla information om att data från den yttre enheten är korrekt mottaget, om handskakningssignaler från enheten är aktiva (exempelvis för att indikera att papperet är slut i en skrivare, mm). Syftet med dataregistren är att uppnå ett lagringsställe (buffert) mellan datorsystemet och den yttre enheten. Ofta talar man om *sändar* och *mottagarregister* (*Transmit and Receive Data Register*).

Slutligen finns styrlogiken. Denna används tillsammans med styrregistren för att få önskad funktion i en tillämpning. Styrregistren måste ges värden när man startar datorsystemet, vi säger att vi *initierar* periferikretsen. Observera att en periferikrets *kan* initieras om under programexekvering till olika funktioner, men det normala är att det görs endast en gång vid systemstart.

I bland kan två register vara placerade *på samma adress*. Vanligt är exempelvis att statusregistret och styrregistret har samma adress. Registren skiljs åt med R/W-signalen eftersom man knappast har intresse av att *skriva till status registret* eller *läsa styrregistret*. På detta sätt kan kretsen utrustas med färre pinnar (adresspinnar) för att adressera de olika registren.

En periferikrets *avlastar* processorn. Den utför alltså ett arbete som processorn *i princip* lika gärna kunnat göra. En seriekrets exempelvis, klockar in databitar seriellt och presenterar data i form av 8 eller 16-bitars dataord för processorn. Vi kunde (om den seriella datatakten inte är för snabb) skriva program för processorn, så att den läst den seriella insignalen, detekterat att nästa databit kan läsas, samlat in 8 eller 16 bitar till ett dataord och lagrat ordet i minnet. Detta medför visserligen massa extra arbete för processorn men det viktigaste är kanske att processorn oavbrutet måste undersöka den seriella insignalen ifall något skulle skickas till datorsystemet.

8.1.2 En skrivaranslutning.

Låt oss betrakta en vanlig typ av gränssnitt mot skrivare som kallas *Centronics Interface*. Det är ett parallellt gränssnitt och används bl.a. i persondator sammanhang. En skrivare som via ett sådant gränssnitt är ansluten till ett mikrodatorsystem kan betraktas på olika sätt. Vi kan

enkelt identifiera åtminstone tre olika grupper som på något sätt kommer i kontakt med skrivaren.

Applikationsprogrammeraren, eller "***den vanlige datoranvändaren***". Från denna nivå är skrivaren *en logisk enhet* som omvandlar text och figurer, synliga på bildskärmen, till text och figurer på papper. Medan användaren behöver någon form av *print (fil)* kommando, kanske applikations-programmeraren kräver möjlighet att formatera utskrifter med enkla satser som: `putchar(tecken,skrivare)` och `putstring( sträng, skrivare)`.

Systemprogrammeraren. På denna nivå ser man två snitt, dels uppåt mot applikationsprogrammeraren och dels nedåt, mot hårdvaran. Systemprogrammerarens uppgift är alltså att tillhandahålla de funktioner i form av putstring, putchar och vad som kan anses nödvändigt. Detta kräver detaljerad kunskap om hårdvaran utan att för den skull inbegripa design och konstruktion av densamma. Ofta skriver man så kallade *drivrutiner*, med ett generellt snitt uppåt men för den speciella hårdvara som skall användas.

Hårdvarukonstruktören måste ha ingående kunskaper om olika typer av periferikretsar, framför allt för att kunna välja ut den mest lämpade, men också, självfallet, för att rent fysiskt kunna ansluta den aktuella kretsen till datorsystemet.

Applikationsprogrammerare

Systemprogrammerare

Hårdvarukonstruktör

Applikationsprogrammeraren "ser" alltså skrivaren endast som de anrop/direktiv (*print fil)* som systemprogrammeraren sedan tidigare tillhandahållit. Följaktligen behöver han ingen som helst kännedom om hur periferikretsen är beskaffad, vilka adresser den finns på, vilka data och statusregister mm den har. Förmodligen räcker det med att han vet att av skrivarens två kablar, skall en till datorsystemet och en till kraftförsörjningen från vägguttaget.

En systemprogrammerare som skall skriva drivrutiner för skrivaranslutningen måste veta en hel del om själva periferikretsen. Han måste veta vilka adresser dataregistren för ut och inmatning har (data ut respektive status från skrivare), vad bitarna i statusregistret indikerar, mm. Jämför detta med en assemblerprogrammerares bild av processorn. Studera figur 8.3 som visar *systemprogrammerarens* bild av MOTOROLAS parallellkrets **MC68230.**

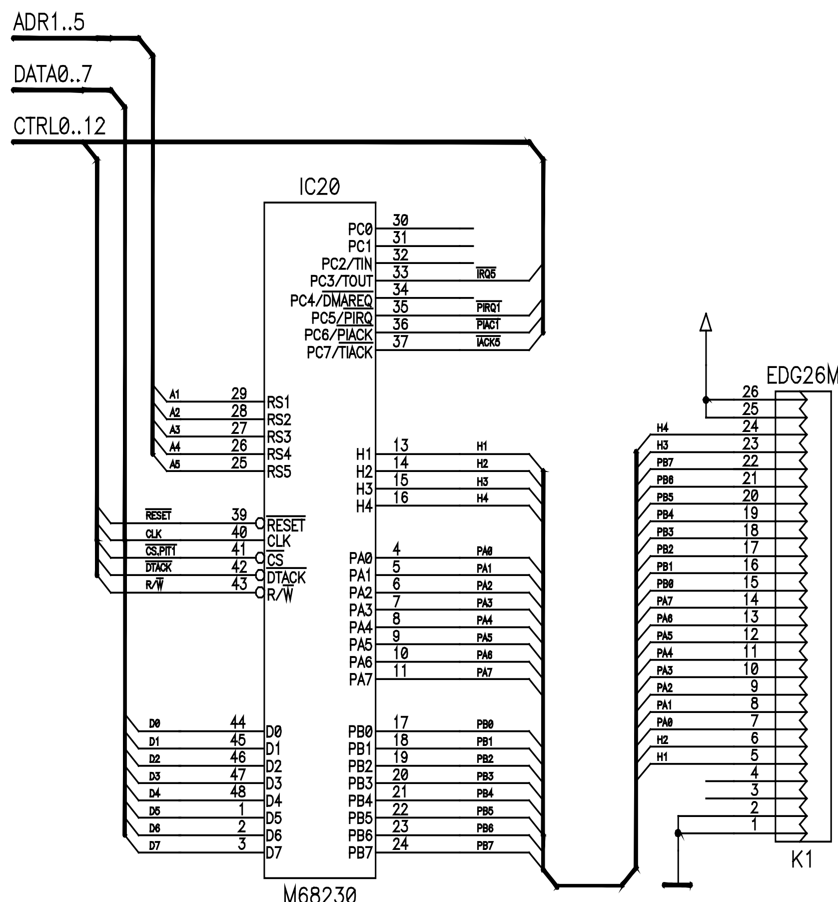
Vi ser där status och dataregistren (nederst). Systemprogrammeraren måste veta hur kretsen ska initieras. Överst i figuren visas ett styrregister med vars hjälp man ger kretsen en övergripande funktion, vidare följer två riktningsregister som bestämmer om kretsens dataportar skall vara ut eller inportar. Troligen önskar systemprogrammeraren att använda någon form för avbrottsstyrd utmatning till skrivaren. I så fall måste han även initiera *Interrupt Vector Register*. Slutligen finns två register (*Port A* och *B Control Register*) som

används för att erhålla önskad handskakning med skrivaren, exempelvis, "Klar att ta emot nytt tecken", "Slut på papper", mm.

7	6	5	4	3	2	1	0	
Mode		H34	H12	H4	H3	H2	H1	Port General Control Register
I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	Port A Data Direction Register
I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	Port B Data Direction Register
Interrupt vector number						*	*	Port Interrupt Vector Register
PA mode		H2 control			H2ie	H1en	H1st	Port A Control Register
PB mode		H4 control			H4ie	H3en	H3st	Port B Control Register
PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	Port A Data Register
PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	Port B Data Register
H4L	H3L	H2L	H1L	H4S	H3S	H2S	H1S	Port Status Register

FIGUR 8.3 SYSTEMPROGRAMMERARENS BILD AV PRINTERPORTEN

Slutligen har vi hårdvarukonstruktörens bild av printerporten. (Se figur 8.4) Han måste veta vad pinnarna på kretsen har för funktion och hur dessa skall anslutas till processor och skrivare. Vidare måste han veta om kretsen skall ha möjlighet att generera avbrott (*Interrupt*) i systemet. Vad som inte visas i denna figur är eventuella buffertar och drivkretsar för Centronics gränssnittet vilket han också måste känna till. Slutligen måste han veta hur kablaget för en standard parallell printerport ser ut, och kunna ansluta detta.



FIGUR 8.4 HÅRDVARUKONSTRUKTÖRENS BILD AV PRINTERPORTEN

8.2 PI/T - MC68230

PI/T, *Parallell Interface/Timer*, är en generell parallellkrets med inbyggd räknare. Parallellkretsen kan användas som gränssnitt mot skrivare, tangentbord, relä, display, SCSI-adapter (*SCSI = Small Computer System Interface)* dvs en buss för anslutning av hårdiskar, bandstationer, mm, osv. Räknarkretsen kan användas som en (realtids-) klocka, pulsräknare, pulsgenerator, frekvensmätare, mm. Fortsättningsvis betecknas parallellkretsen MC68230 som *PI/T*.

PI/T:n kännetecknas av:

- *MC68000 buss kompatibel*
- *Olika in och ut moder*
- *Programmerbar handskakning*
- *Olika räknarmoder för den 24bitars programmerbara räknardelen.*
- *5 separata avbrottsvektorer*
- *Separata avbrottssignaler från I/O och räknare*

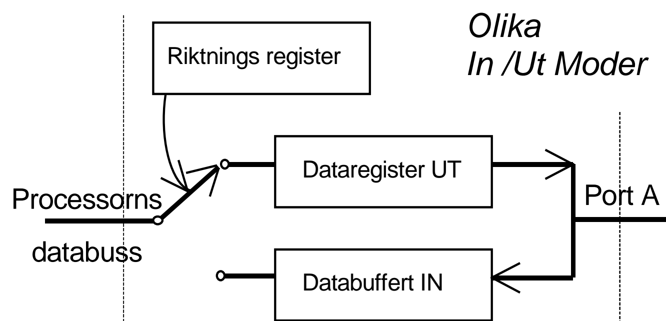
PI/T:n har 48 pinnar (anslutningar) varav 18 skall anslutas till processorns buss system. Det finns ett antal pinnar med dubbelriktad funktion som antingen kan anslutas till processorns bussar eller till den yttre enheten. Övriga pinnar ansluts till den yttre enheten för dataöverföring eller i handskakningssyfte.

Buss kompatibel

Kretsen är *MC68000 buss kompatibel*. Med detta menas att den direkt kan anslutas till bussarna på en processor ur M68xxx familjen. Kretsen använder sig av samma bussprotokoll som MC68000 och genererar de handskakningssignaler som krävs.

Olika In /Ut Moder

Kretsen kan initieras för att arbeta i olika *moder*. Med olika in och utmoder menas att kretsen kan arbeta med bitvis in och utmatning, där bitarnas riktning (in eller ut) kan bestämmas oberoende av varandra. Se figur 8.5. Vidare kan porten arbeta som en 16 bitars port eller två 8 bitars portar. Den kan också initieras som en bidirektionel port, dvs en port som både kan skrivas och läsas (dataöverföring i båda riktningar).



FIGUR 8.5. PRINCIPBILD AV PI/T:NS PORTAR

Programmerbar handskakning

Kretsen är utrustad med handskakningssignaler för porten och dessa kan oberoende av varandra initieras för olika typer av handskakning.

Olika Räknarmoder

Räknardelen på PI/T'n kan initieras för olika räknarmoder. Räknaren består av 24 bitar vilket innebär att den kan räkna upp till $16 \cdot 10^6$. De olika moderna kan vara pulslängdsmätning, som en realtidsklocka, pulstågsgenerator, mm.

Olika avbrott

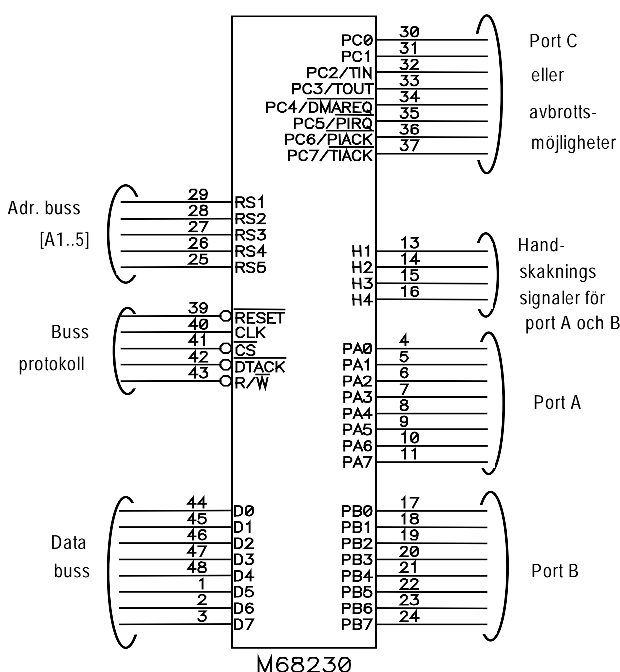
Kretsen har fem olika avbrotts vektorer där två är knutna till varje 8 bitars port och en till räknaren. Kretsen har separata avbrottssignaler för porten och räknaren. Detta innebär att man har möjlighet att välja olika avbrottsnivåer för port och räknare.

PI/T'n kan också användas i DMA sammanhang. Kretsen är utrustad med en pinne som kan initieras för att bergära DMA-överföring.

Alla register i kretsen är direkt åtkomliga för programmeraren (alla register har unik adress). Figur 8.6 visar *programmerarens bild* av PI/T'n.

Offset	Register namn	Förkortning
$0	Port General Control Register	PGCR
$1	Port Service Request Register	PSRR
$2	Port A Data Direction Register	PADDR
$3	Port B Data Direction Register	PBDDR
$4	Port C Data Direction Register	PCDDR
$5	Interrupt Vector Register	IVR
$6	Port A Control Register	PACR
$7	Port B Control Register	PBCR
$8	Port A Data Register	PADR
$9	Port B Data Register	PBDR
$A	Port A Alternate Register	PAAR
$B	Port B Alternate Register	PBAR
$C	Port C Data Register	PCDR
$D	Port Status Register	PSR
$10	Timer Control Register	TCR
$11	Timer Interrupt Vector Register	TIVR
$13	Counter Preload Register High	CPRH
$14	Counter Preload Register Mid	CPRM
$15	Counter Preload Register Low	CPRL
$17	Count Register High	CRH
$18	Count Register Mid	CRM
$19	Count Register Low	CRL
$1A	Timer Status Register	TSR

FIGUR 8.6 PROGRAMMERARENS BILD AV MC68230



FIGUR 8.7 MC68230 PI/T

Studera nu figur 8.7. Databussen återfinns längst ner till vänster i figuren. Observera att databussen är 8 bitar bred och ansluts här till den nedre halvan [D7..D0] av processorns databuss.

Nederst till höger finns port B och port A. Här ansluts de yttre enheter som kretsen skall arbeta tillsammans med. Vanligtvis behövs också någon handskakning med den yttre enheten, då används signalerna [H1..4].

Port C har dubbel betydelse, antingen fungerar den som en generell in och utport eller så används den i avbrotts sammanhang.

Adressbussens anslutning visas överst till vänster i figuren. Vi ser att denna är 5 bitar bred, [RS1..5]. Detta indikerar att kretsen kan innehålla högst 32 olika register ($2^5 = 32$). I figur 8.6 ser vi att kretsen i själva verket innehåller 23 st 8-bitars register åtkomliga för programmeraren. Vi återkommer senare till hur dessa register kan användas. Figur 8.7 visar också styrsignaler och handskakningssignaler för processorns bussprotokoll.

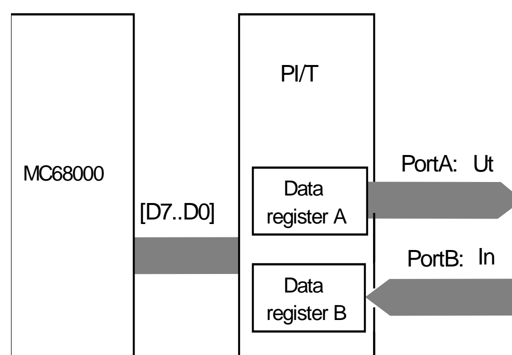
I inledningen sa vi att PI/T'n kunde fungera både som in eller utport. Önskad funktion fås genom att initiera kretsen. En mycket förenklad bild av hur kretsen kan fungera som antingen in eller utport visas i figur 8.5.

Processorns databuss kan via ett styrregister i kretsen anslutas till ett dataregister som är riktat ut från PI/T:n eller mot en databuffert in mot PI/T'n. Genom att skriva ettor i riktningsregistret ansluts processorns databuss till dataregistret (utport) och genom att skriva nollor, ansluts processorns databuss till databufferten (inport).

8.2.1 Programmering av PI/T:ns portar

Låt oss nu illustrera användningen av PI/T:n med ett programexempel. Antag att vi önskar *skicka ut* 8 bitar parallell data på Port A och *ta emot* 8 bitars parallell data på Port B. Vi har alltså en bild enligt figur 8.8.

Programmässigt kan vi då betrakta porten enligt exempel 8.1.



FIGUR 8.8. PI/T SOM UT OCH INPORT

EXEMPEL

PI/T'ns adresser och in-och utmatning via portarna

Med dessa enkla instruktioner kan vi alltså läsa data från inporten (dataregister Port B i PI/T:n) och skriva data till utporten (dataregister Port A i PI/T:n)

```
PADR     EQU  $3e0011   Fysisk adress i systemet
PBDR     EQU  $3e0013   Fysisk adress i systemet
OutPort  EQU  PADR      Adress Port A PI/T OutPort
InPort   EQU  PBDR      Adress Port B PI/T InPort

*  Del av programmet
      ...
      MOVE.B    (InPort).L,D0     Läs data
      MOVE.B    D0,(OutPort).L    Skriv data
      ...
```

8.1

Innan PI/T'n kan användas på detta sätt måste kretsen initieras så att Port A verkligen fungerar som en utport och Port B som en inport. För att kunna initiera kretsen måste vi veta hur den fungerar, i detalj. PI/T'n är en komplex krets med ett antal olika *moder*, där kretsen kan initieras för att arbeta i både dubbelriktad (*bidirectional*) och enkelriktad (*unidirectional*) mod.

Fyra huvudmoder :

- *Mod 0: Enkelriktad 8 bitars*
- *Mod 1: Enkelriktad 16 bitars*
- *Mod 2: Dubbelriktad 8 bitars*
- *Mod 3: Dubbelriktad 16 bitars*

Submoder:

- *Submod 00: Pin definierbar dubbel buffrad ingång eller enkel buffrad utgång*
- *Submod 01: Pin definierbar dubbel buffrad utgång eller icke-klockadingång (Non Latched Input)*
- *Submod 1X: Bitvis I/O (Pin definierbar enkel buffrad utgång eller icke-klockad ingång (Non Latched Input)*

PI/T:n kan arbeta i fyra olika moder. Arbetssättet väljs vid initieringen. Vilken mod som skall användas bestäms av innehållet i *Port General Control Register* (PGCR). Vi återkommer till detta register längre fram. Vilken *submod* som används väljs i *Port A* eller *Port B Control Register* (PACR och PBCR).

Alltså kan kretsen *buffra* eller *inte buffra* data på sina in och utgångar. Kretsen kan arbeta med 8 eller 16 bitars in och utmatning. Dessutom kan handskakningssignalerna [H1..H4] (se figur 8.7) initieras för ett antal olika arbetssätt. Vi kan, i detta inledande exempel, bortse från en rad egenskaper hos kretsen eftersom vi använder den enkla uppkopplingen (funktionen) som visas i figur 8.8.

Som på de flesta periferikretsar, finns även på PI/T'n en anslutning till RESET. När denna signal aktiveras får kretsen ett väl definierat *begynnelsetillstånd.* Efter detta är kretsen enkel att initiera. Vid RESET nollställs de flesta register av betydelse. Detta innebär att vi kan förutsätta att en bestämd funktion hos kretsen är definierad.

I figur 8.9 visas kretsens *riktnings-* respektive *data-*register. Observera att alla register är 8 bitar breda. Med de två första registren, riktningsregister A och riktningsregister B (Port A och B Data Direction Register) bestäms om porten skall vara ut eller inport. Skriver vi ettor i ett av dessa register kommer motsvarande port att fungera som en *utport* och skriver vi nollor får vi en *inport*.

Register Select Bits	7	6	5	4	3	2	1	0	
00010	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	Port A Data Direction Register
00011	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	Port B Data Direction Register
01000	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	Port A Data Register
01001	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	Port B Data Register

FIGUR 8.9 PI/T:NS RIKTNING OCH DATAREGISTER.

Vi vill initiera Port A som utport (jämför figur 8.8), det är då tillräckligt att ändra Port A's riktningsregister PADDR (*Port A Data Direction Register*). Detta skall i så fall initieras med ettor för att åstadkomma en utport. Se exempel 8.2 och figur 8.10.

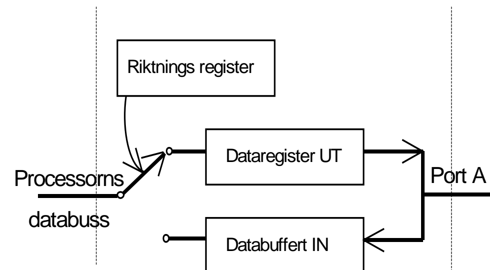
EXEMPEL

Initiera PortA som utport

```
MOVE.B  #$FF,(PADDR).L  PortA=Out Port
```

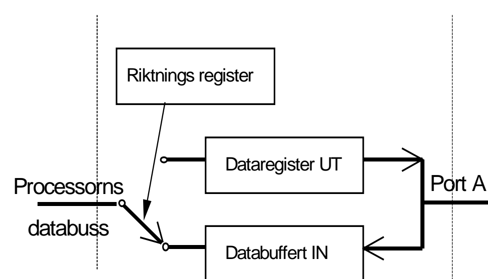
8.2

Då vi förutsätter att RESET nyligen är utfört, är kretsen i *Mod 0* och *Submod 00* ty registren nollställs vid RESET. Mod 0 innebär att kretsen är initierad för enkelriktad 8 bitars överföring. Submod 00 innebär att en utport är enkelbuffrad. Kretsen är därför initierad korrekt efter RESET för denna konfiguration och det enda vi behöver göra är att initiera riktningsregistret (PADDR) enligt ovan.



FIGUR 8.10. PORT A = UT

Port B ska initieras som en inport. Efter RESET är Port B's riktningsregister PBDDR (*Port B Data Direction Register*) nollställt och Port B är därmed en inport. I initieringsrutiner brukar man även ta med denna initiering för att det förtydligar kretsens funktion och initiering när man studerar en programlistning. Se exemepl 8.3 och figur 8.11.



FIGUR 8.11. PORT B = IN

EXEMPEL

Initiera PortB som inport

```
MOVE.B      #$00,(PBDDR).L    PortB=In Port
```

8.3

Vi väljer att även ta med initiering av inporten (fast det i detta exempel är onödigt). Efter RESET placeras kretsen alltid i *submod 00* vilket alltså innebär har dubbel buffrade ingångar. För att välja bort buffrad ingång använder vi därför *Submod 01*. Detta väljs i Port B's styrregister PBCR (*Port B Control Register*). En initiering av Port B som *obuffrad* ingång visas i exempel 8.4.

EXEMPEL

Initiera port B som obuffrad inport

```
MOVE.B #$00,(PBDDR).L       Port B=inport
MOVE.B #%01000000,(PBCR).L  PortB=Non Latched Input
```

8.4

Med dessa få rader har vi nu initierat kretsen för en önskad funktion. Notera att ej initierade register är *nollställda* efter RESET.

Vi har fram till nu behandlat både Port A och Port B:s riktningsregister (PADDR och PBDDR), vidare portarnas styrregister (PACR och PBCR) och slutligen deras dataregister (PADR ochPBDR). Figur 8.12 visar en bild över de register vi hitills har använt.

Register Select Bits	7	6	5	4	3	2	1	0	
00010	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	Port A Data Direction Register
00011	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	Port B Data Direction Register
00110	PA mode		H2 control			H2ie	H1en	H1st	Port A Control Register
00111	PB mode		H4 control			H4ie	H3en	H3st	Port B Control Register
01000	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	Port A Data Register
01001	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	Port B Data Register

FIGUR 8.12 DE REGISTER I PI/T:N VI HITTILLS ANVÄNT

Funktionen hos riktnings-registren var enkel, vi skrev in ettor där vi önskade utport och nollor där vi önskade en inport. När det gäller styrregistret och dess olika bitar återkommer vi till dessa längre fram.

En fullständig typisk initiering (enligt vårt exempel) av en periferikrets visas i exempel 8.5. Subrutinhuvudet beskriver kortfattat hur kretsen initieras. En assemblerprogrammerare skall genom att läsa detta få tillräckligt med information för att kunna använda kretsen. Endast i undantagsfall skall han vara tvingad läsa radkommentarerna i koden.

EXEMPEL

Fullständig initiering av PI/T:ns parallellportar.

```
* Definiera adresser för PI/T
pit          EQU   $3e0001     PI/T base address
PADDR        EQU   pit+4       PortA Data Direction
PBDDR        EQU   pit+6       PortB Data Direction
PBCR         EQU   pit+$e      PortBControlRegister
* Subroutine InitPIT initiate PI/T to
*            PortA as outport (Single Buffered)
*            PortB as inport (Non Latched)
*            Timer is not initiated
* No interrupts enabled
* InitPIT assumes no access to PI/T since RESET
InitPIT:
      MOVE.B      #$FF,(PADDR).L       Port A Out
* (Submode 00 after RESET)
      MOVE.B      #$0,(PBDDR).L        Port B In
      MOVE.B      #%01000000,(PBCR).L  Submode 01
      RTS
```

8.5

Där man anger styrord i initieringen kan det vara en god regel att visa dessa som binärtal, då det är enklare att "se var ettorna finns". Om styrorden är uppenbara och består av enbart ettor eller nollor kan det vara bättre att använda hexadecimala tal för läslighetens skull.

Huruvida definitionen av registren skall anges tillsammans med initieringsrutinen eller inte kan man ha olika uppfattning om. Definitionen anger här bland annat på vilka adresser de olika registren hamnar i ett typisk system med en 16 bitars databuss. Då PI/T:n har en 8 bitars databuss och är ansluten till den nedre halvan av processorns buss kommer endast udda adresser att vara aktuella för kretsen. Jämför deklarationerna med figur 8.6.

Med denna initiering av PI/T:n vi visat ovan kan vi nu enkelt skriva de rutiner som krävs för att *läsa* data från parallellporten och *skriva* data till parallellporten. Se exempel 8.6.

EXEMPEL

Subrutiner för att läsa från, och skriva till, parallellportarna

```
* Subroutine InPIT
* Reads 8 bits of data from parallell inport
* Returns data in D0.(byte)
InPIT:
       MOVE.B (InPort).L,D0
       RTS

* Subroutine OutPIT
* Writes 8 bits of data to parallell outport
* D0.(byte) holds data to be written
OutPIT:
       MOVE.B D0,(OutPort).L
       RTS
```

8.6

Vi har nu konstruerat de programrutiner som krävs för att *initiera*, *läsa* och *skriva* parallell data från och till en yttre enhet (`InitPIT`, `InPIT` och `OutPIT`). Fortsättningsvis kan vi använda dessa rutiner utan att veta de exakta detaljerna kring periferikretsens funktion. Vi diskuterade tidigare applikations- respektive systemprogrammerarens bild av en printerport. I exempel 8.7 visas ett programexempel med den principiella användningen av dessa rutiner.

EXEMPEL

Principiell användning av de konstruerade subrutinerna.

```
main: JSR   (InitPIT).L       init peripheral
main2: JSR  (InPIT).L         read data from port
*
* do something with data ...
       JSR    (OutPIT).L   output data
       BRA    main2
```

8.7

Att konstruera rutinerna `IniPIT`, `InPIT` och `OutPIT` motsvarar arbetet för en system-programmerare. En applikationsprogram-merare utnyttjar dessa rutinerna enbart med kännedom om deras namn och hur parameter-överföringen går till.

Med detta har vi visat en enkel initiering och användning av kretsen PI/T MC68230. För att studera kretsen mer ingående krävs kännedom om hur kretsens register är uppbyggda och styrbitarnas funktion. Vi fortsätter därför med att beskriva kretsens registeruppsättning.

8.2.2 Registeruppsättning för PI/T:n portar.

I det följande ges en kortfattad beskrivning av de olika registren för portarna. Figur 8.13 visar kretsens alla port-register. Studera även figur 8.6 och 8.7 som visar kretsens portar och anslutningar. Se speciellt Port C, vars pinnar kan vara vanliga ut/in portar eller handskakningssignaler som används i avbrott och DMA sammanhang.

PI/T:n är utrustad med 23 st. 8-bitars register där 14 används för parallellporten, övriga nio används för räknardelen (behandlas senare).

Register Select Bits	7	6	5	4	3	2	1	0	
00000	Mode		H34	H12	H4	H3	H2	H1	Port General Control Register
00001	*	SVCRQ Sel		IPF Sel		IRQ Priority			Port Service Requestl Register
00010	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	Port A Data Direction Register
00011	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	Port B Data Direction Register
00100	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	Port C Data Direction Register
00101	Interrupt vector number						*	*	Port Interrupt Vector Register
00110	PA mode		H2 control			H2ie	H1en	H1st	Port A Control Register
00111	PB mode		H4 control			H4ie	H3en	H3st	Port B Control Register
01000	A7	A6	A5	A4	A3	A2	A1	A0	Port A Data Register
01001	B7	B6	B5	B4	B3	B2	B1	B0	Port B Data Register
01010	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	Port A Alternate Register
01011	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	Port B Alternate Register
01100	C7	C6	C5	C4	C3	C2	C1	C0	Port C Data Register
01101	H4L	H3L	H2L	H1L	H4S	H3S	H2S	H1S	Port Status Register

FIGUR 8.13. REGISTERUPPSÄTTNING FÖR PI/T:NS PORTAR

PGCR

Read/Write

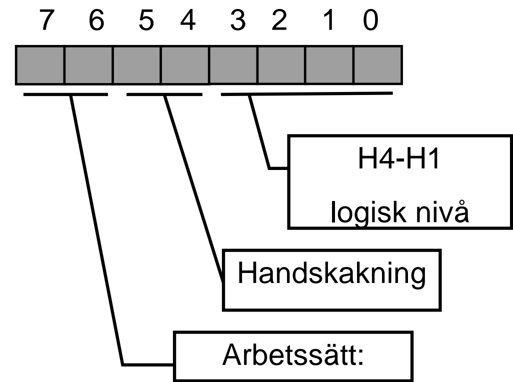
Adress offset

Port General Control Register. (PGCR)

I detta register skrivs ett *överordnat* styrord till kretsen; exempelvis 8 eller 16 bitars mod (Submoder bestäms i Port A och B Control Register) och om handskakningssignalerna [H4..H1] skall vara aktivt höga eller låga. Observera bitarna 4 och 5 som ettställs för att åstadkomma handskakning med [H4..H1]. Dessa bitar måste nollställas om kretsen skall initieras på nytt.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Mode		H34	H12	H4	H3	H2	H1

Bit 7 6	Mode
0 0	Mod 0: Enkelriktad 8 bitars mod
0 1	Mod 1: Enkelriktad 16 bitars mod
1 0	Mod 2: Dubbelriktad 8 bitars mod
1 1	Mod 3: Dubbelriktad 16 bitars mod
Bit 5 4	**Handskakning**
0	H12 och H34, Handskakning används *ej*.
1	H12 och H34, Handskakning används
Bit 3 2 1 0	**Logisk nivå**
0	[H4..H1] är aktiva låga
1	[H4..H1] är aktiva höga



Port Service Request Register. (PSRR)

PSRR

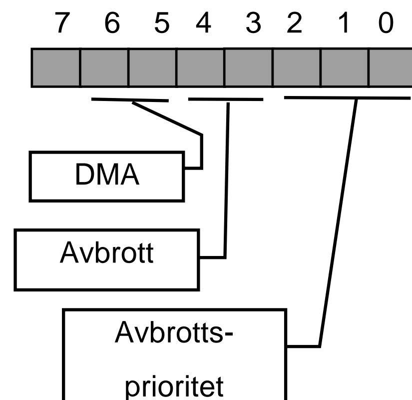
Read/Write

Adress offset $1

Ett generellt register som används för att ange typ av avbrott och DMA-överföring om dessa används. Observera att beroende på initiering så kommer pinnar tillhörande Port C som har dubbel betydelse att användas vid handskakning i samband med avbrott eller DMA-överföring.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used	SVCRQ Select		Operation Select		Port Interrupt Priority Control		

Bit 6 5	SVRC Select: DMA mod			
0 X	Ej DMA			
1 X	DMA mod, bit 5 anger *vilken* mod			
Bit 4 3	**Operation Select: Interrupt**			
0 0	Ej avbrott			
0 1	Autovektor avbrott			
1 1	Vektoravbrott			
Bit 2 1 0	**Hög**	**Prioritet**		**Låg**
0 0 0	H1S	H2S	H3S	H4S
0 0 1	H2S	H1S	H3S	H4S
0 1 0	H1S	H2S	H4S	H3S
0 1 1	H2S	H1S	H4S	H3S
1 0 0	H3S	H4S	H1S	H2S
1 0 1	H3S	H4S	H2S	H1S
1 1 0	H4S	H3S	H1S	H2S
1 1 1	H4S	H3S	H2S	H1S



Bitarna anger prioriteten på avbrotten när fler än en av signalerna [H4..H1] kan generera avbrott

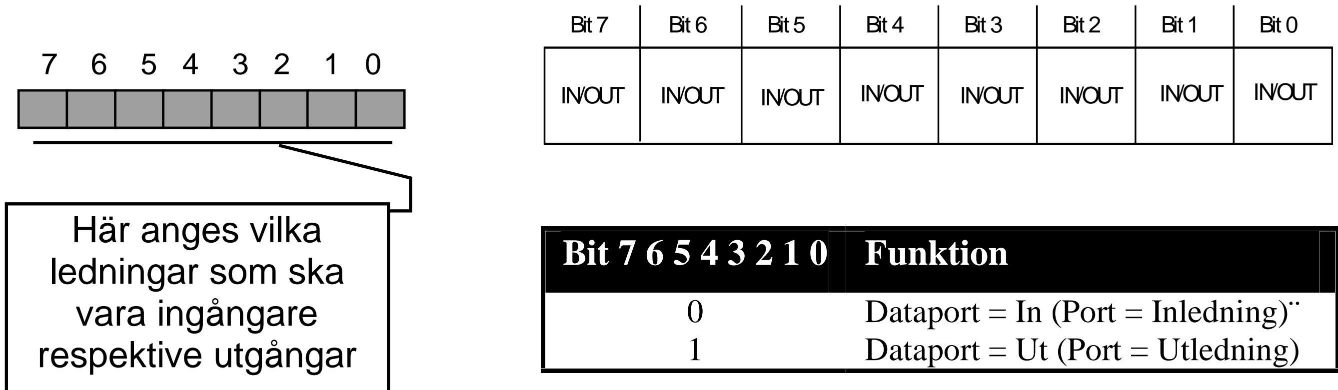
PADDR

Read/Write

Adress offset $2

Port A Data Direction Register (PADDR)

Bestämmer om Port A skall vara in eller utport. Port A (PADDR) och Port B (PBDDR) initieras på samma sätt.



Bit 7 6 5 4 3 2 1 0	Funktion
0	Dataport = In (Port = Inledning)¨
1	Dataport = Ut (Port = Utledning)

PBDDR

Read/Write

Adress offset $3

Port B Data Direction Register (PBDDR)

Här anges om Port B skall vara in eller utport. Port A (PADDR) och Port B (PBDDR) initieras på samma sätt.

PCDDR

Read/Write

Adress offset $4

Port C Data Direction Register (PCDDR)

Bestämmer om Port C skall vara in eller utport. Noll ger ingång och etta ger utgång. Observera att vissa av Port C:s pinnar kan användas som handskakningssignaler då avbrott och/eller DMA-överföring används. Se även PADDR.

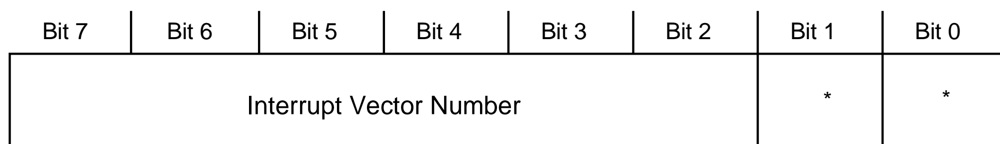
PIVR

Read/Write

Adress offset $5

Port Interrupt Vector Register (PIVR)

Detta register skall initieras med portarnas avbrotts vektornummer.



Observera att bit 1 och 0 inte kan initieras. Dessa två bitar anger vilken av signalerna [H4..H1] som genererat avbrott. Om exempelvis registret är initierat till $40 (64) kommer vektornumren att bli:

Avbrott från		Avbrottsvektor
H1;	(bit 1 och 0 är 00)	$40
H2;	(bit 1 och 0 är 01)	$41
H3;	(bit 1 och 0 är 10)	$42
H4;	(bit 1 och 0 är 11)	$43

Efter RESET innehåller registret $0F vilket är vektornumret (15, adress $03C) som enligt **MOTOROLA** är "Avbrott från enhet som ej tillhandahållit avbrottsnummer".

Port A Control Register. (PACR)

PACR

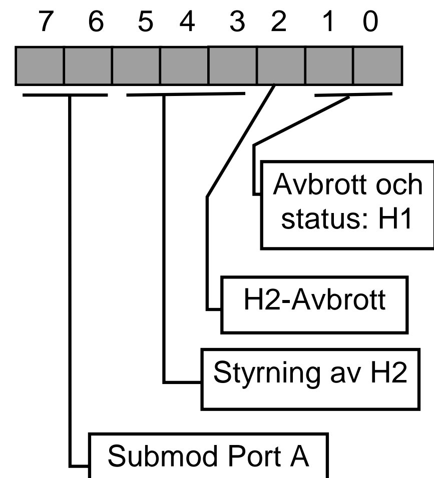
Read/Write

Adress offset $6

Registret, i samband med Port General Control Register, bestämmer funktionen för Port A i detalj. PACR och PBCR har samma funktion.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Port A Submode		H2 Control			H2 Interrupt Enable	H1 SVCRQ Enable	H1 Status Control

Bit 7 6	Submoder
0 0	Pin definierbar dubbel buffrad ingång eller enkel buffrad utgång
0 1	Pin definierbar dubbel buffrad utgång eller icke klockadingång (Non Latched Input)
1 X	Bitvis I/O (Pin definierbar enkel buffrad utgång eller icke klockad ingång (Non Latched Input)



Beroende på vilken Submod som väljs har registrets resterande bitar olika betydelse. Nedan visas funktionen hos registrets övriga bitar **när Mod 0 och Submod 00 är vald**.

Bit 5 4 3	Styrning av H2; påverkan av Port Status register
0 X X	H2 = Ingång; H2S ettställs vid aktiv flank på H2.
1 0 0	H2 = Utgång; H2 nollställs, H2S alltid noll.
1 0 1	H2 = Utgång; H2 ettställs, H2S alltid noll.
1 1 0	H2 = Utgång; fullständig handskakning, H2S alltid noll.
1 1 1	H2 = Utgång; ofullständig handskakning, H2S alltid noll.
Bit 2	**H2, avbrottsmöjligheter**
0	Avbrott EJ möjligt via H2
1	Avbrott möjligt via H2

Bit 1	H1 avbrott och DMA möjligheter
0	Avbrott och DMA EJ möjligt via H1
1	Avbrott och DMA möjligt via H1
Bit 0	**H1 Status**
X	H1S ettställs alltid när data finns i den dubbel buffrade inporten. Denna bit saknar betydelse i denna mod/submod

PBCR

Read/Write

Adress offset $7

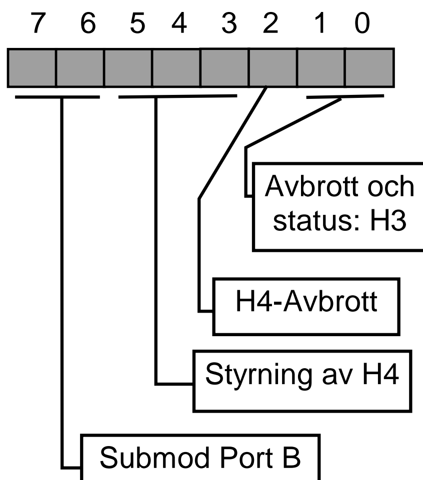
Port B Control Register. (PBCR)

Registret, i samband med Port General Control Register, bestämmer funktionen för Port B i detalj. PACR och PBCR har samma funktion.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Port B Submode		H4 Control			H4 Interrupt Enable	H3 SVCRQ Enable	H3 Status Control

Bit 7 6	Submoder
0 0	Dubbel buffrad ingång / enkel buffrad utgång
0 1	Dubbel buffrad utgång / icke-klockad ingång (non latched input)
1 X	Enkel buffrad utgång / icke-klockad ingång (non latched input)

Beroende på vilken Submod som väljs har registrets resterande bitar olika betydelse. Nedan visas funktionen hos registrets övriga bitar **när Mod 0 och Submod 00 är vald**.



Bit 5 4 3	Styrning av H4; påverkan av Port Status register
0 x x	H4 = Ingång; H4S ettställs vid aktiv flank på H4.
1 0 0	H4 = Utgång; H4 nollställs, H4S alltid noll.
1 0 1	H4 = Utgång; H4 ettställs, H4S alltid noll.
1 1 0	H4 = Utgång; fullständig handskakning, H4S alltid noll.
1 1 1	H4 = Utgång; ofullständig handskakning, H4S alltid noll.
Bit 2	**H4, avbrottsmöjligheter**
0	Avbrott EJ möjligt via H4
1	Avbrott möjligt via H4
Bit 1	**H3 avbrott och DMA möjligheter**
0	Avbrott och DMA EJ möjligt via H3
1	Avbrott och DMA möjligt via H3
Bit 0	**H3 Status**
x	H3S ettställs alltid när data finns i den dubbel buffrade inporten. Denna bit saknar betydelse i denna mod/submod

Port A Data Register. (PADR)

PADR

Read/Write

Adress offset $8

Registret används som dataregister för Port A. PADR och PBDR har samma funktion. Registret kan läsas eller skrivas oberoende av portens riktning. Observera att skriv-ning respek-tive läsning kan påverka hand-skakningssignalerna vid dubbel buffring av data.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

Port B Data Register. (PBDR)

PBDR

Read/Write

Adress offset $9

Registret används som dataregister för Port B. PADR och PBDR har samma funktion. Registret kan läsas eller skrivas oberoende av portens riktning. Observera att skriv-ning respek-tive läsning kan påverka hand-skakningssignalerna vid dubbel buffring av data.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

Port A Alternate Register. (PAAR)

PAAR

Read

Adress offset $A

Detta register är direkt kopplat till Port A:s pinnar. Registret kan endast läsas. PAAR och PBAR har samma funktion. Nivån på portens pinnar kan läsas i detta register obero-ende av vilken mod, submod, in, ut, mm. kretsen har är i..

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

Port B Alternate Register. (PBAR)

PBAR

Read

Adress offset $B

Detta register är direkt kopplat till Port B:s pinnar. Registret kan endast läsas. PAAR och PBAR har samma funktion. Nivån på portens pinnar kan läsas i detta register obero-ende av vilken mod, submod, in, ut, mm. kretsen har är i..

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

PCDR

Read/Write

Adress offset $C

Port C Data Register. (PCDR)

Registret används som dataregister för Port C.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

Registret kan läsas eller skrivas oberoende av portens riktning. Observera att vissa av portens bitar kan ingå i handskakningen i avbrotts och DMA- sammanhang.

PSR

Read/Write

Adress offset $D

Port Status Register. (PSR)

Registret innehåller information om handskakningen för [H4..H1].

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
H4 Level	H3 Level	H2 Level	H1 Level	H4S	H3S	H2S	H1S

Avbrott avlägsnas från respektive [H4..H1] genom att en etta skrivs till respektive bit

Bit 7 6 5 4	H4 H3 H2 H1 Level
x	Handskakningssignalernas nivå
Bit 3 2 1 0	**H4 H3 H2 H1 Level**
x	En aktiv signal på [H4..H1] (oberoende om aktiv låg eller hög) indikeras med en etta i detta bitfält.

8.2.3 Sammanfattning över PI/T:ns port-register

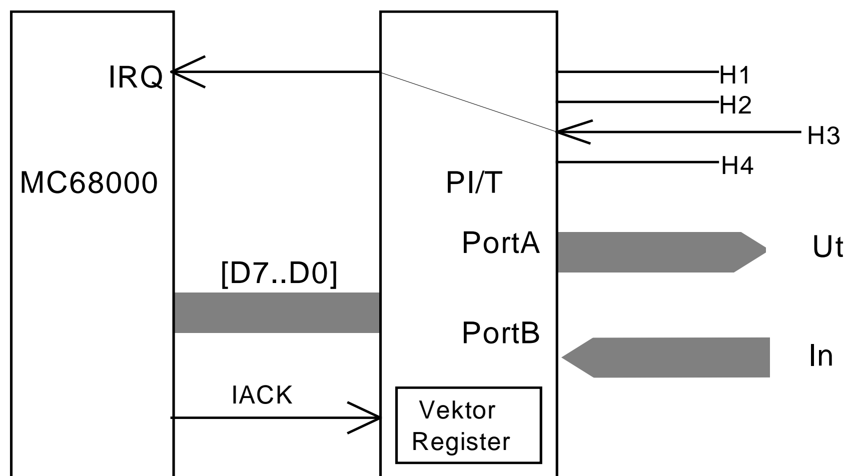
I figur 8.14 ges en sammanställning av PI/T:ns port-registren.

Offset	Register namn	Förkortning
$0	Port General Control Register	PGCR
$1	Port Service Request Register	PSRR
$2	Port A Data Direction Register	PADDR
$3	Port B Data Direction Register	PBDDR
$4	Port C Data Direction Register	PCDDR
$5	Interrupt Vector Register	IVR
$6	Port A Control Register	PACR
$7	Port B Control Register	PBCR
$8	Port A Data Register	PADR
$9	Port B Data Register	PBDR
$A	Port A Alternate Register	PAAR
$B	Port B Alternate Register	PBAR
$C	Port C Data Register	PCDR
$D	Port Status Register	PSR

FIGUR 8.14 PROGRAMMERARENS BILD AV MC68230

8.2.4 Vektoravbrott och PI/T:ns portar

Vi har tidigare diskuterat olika former för avbrott. Vi skall nu se hur **PI/T**:n kan användas i avbrotts sammanhang. I följande exempel använder vi oss av vektoravbrott. Studera figur 8.15. Figuren visar också vektorregistret i kretsen. Beroende på vilken av handskakningssignalerna ([H1..H4]) som genererar avbrott kommer kretsen att addera noll, ett, två eller tre till sitt vektornummer innan processorn läser kretsens vektor register.



Vektoravbrott innebär att periferikretsen identifierar sig med ett vektornummer som processorn läser via databussen. Kretsen har handskakningssignalerna H1, H2, H3 och H4 som kan initieras för att generera avbrott. Dessa kan vidarekopplas genom PI/T:n för att generera en avbrottssignal mot processorn.

FIGUR 8.15. PI/T SOM EN AVBROTTSSTYRD UT OCH INPORT.

I vårt exempel skall vi initiera kretsen där Port A är utgång och Port B är ingång. Vidare skall kretsen generera avbrott för en *negativ flank* på insignalen (handskakningssignalen) H3. Slutligen väljer vi kretsens vektornummer i detta exempel till $80 (128).

Att initiera portarna som in och utgångar är känt sedan tidigare och utförs genom att ge riktningsregistren sina värden enligt exempel 8.8.

EXEMPEL

Initiera PI/T för : Port A = UT och Port B = IN

```
      MOVE.B      #$ff,(PADDR).L    Port A = Outport
      MOVE.B      #0,(PBDDR).L      Port B = Inport
```

8.8

Vi måste initiera styrregistren till korrekt Mod och Submod. Här skall även avbrott tillåtas för H3-signalen. Observera att signalen ansluten till H3 signalen är aktiv låg och att H3-ingången är flanktriggad. Vårt exempel innebär Mod 0, dvs enkelriktad 8 bitars mod. Se exempel 8.9. Vidare väljs Submod 00 för Port A och Submod 1X för Port B.

Observera att kretsens generella styrregister (PGCR) skall initieras sist för att påverka bit 5 (H34). Deta visas i exempel 8.10. Portens mod får inte ändras då denna är ettställd. Här anges att kretsen skall arbeta i Mod 0 (bit 7 och 6 sätts till 00) och att H3's aktiva nivå är noll. Därför nollställs bit 2 (H3) i PGCR.

EXEMPEL

Initiera PI/T för : Avbrott för H3 = IN, negativ flank

```
    MOVE.B   #0,(PACR).L            Port A=Submode 00
    MOVE.B   #%10000010,(PBCR).L    Port B
*                                   b76  Submode 1X
*                                   b1   IRQ for H3
    MOVE.B   #%00100000,(PGCR).L    b76  Mode 0
*                                   b5   H34 Active
*                                   b2   H3 active low
```

8.9

Initieringen av Port A's styrregister är ej nödvändig om vi förutsätter att registret inte är ändrat sedan kretsen fått RESET. Registret är då nollställt vilket innebär Submod 00. Port B's styrregister däremot skall ha Submod 1X vilket sätts i bit 7 och 6. Vidare skall kretsen generera avbrott för en aktiv signal på H3 ingången, bit 2 ettställs (H3en).

För att slutföra initieringssekvensen för kretsen initierar vi nu portarnas avbrotts vektornummer och den prioritet som handskakningssignalerna [H1..H4] inbördes skall ha. Detta görs enligt följande:

EXEMPEL

Initiera PI/T för : Vektornummer = $80, Prioritet H3,H4,H1,H2

```
   MOVE.B #%00011100,(PSRR).L   Service Request
*                               b43  Vector Interrupt
*                               b2-0 Priority H3,4,1,2
   MOVE.B #$80,(PIVR).L         Port Interrupt Vector
```

8.10

Bit 4 och 3 i Port Service Request Register *ettställs eftersom vi önskar vektor avbrott. Vidare anger vi också den inbördes prioriteten för signalerna [H1..H4] för de fall där fler än H3 (i vårt exempel) fungerar som avbrottsingångar. Bit 2, 1 och 0 sätts till 100 för att uppnå prioriteten H3, H4, H1 och H2, där H3 har högst prioritet och H2 den lägsta. Slutligen initieras* Port

Observera nu att när H3 genererar avbrott kommer avbrotts vektorn som processorn läser att vara $82. Detta beror på att kretsen adderar 0,1,2 eller 3 till sin vektor beroende på vilken av H1, H2, H3 eller H4 som genererat avbrottet. Processorn kommer därför att läsa vektornummer 130 ($82) i minnet.

Vi måste alltså initiera detta vektornummer med startadressen till vår avbrottsrutin. Vektornummer $82 (130) hittas på adress $208 i minnet. Processorn multiplicerar ju det lästa vektornummret med fyra innan startadressen läses ($82 * 4 = $208). Exempel 8.11 visar detta.

EXEMPEL

Initiera PI/T för : Avbrottsvektorn med avbrottsrutinens startadress

```
   MOVE.L   #IrqRut,(($80+2)*4).L
```

8.11

Vi förutsätter här att `#IrqRut` är startadressen för avbrottsrutinen vilken skall placeras på adress ($80+2)*4. $80 var kretsens vektornummer, tvåan beror på att det är H3 som genererar avbrott och slutligen måste vi multiplicera med fyra för att få korrekt adress.

När väl porten har genererat ett avbrott, pga att H3 aktiverades, måste denna avbrottskälla (IRQ) avlägsnas från PI/T'n mot processorn. För att nollställa IRQ-signalen krävs att vi skriver en etta i portens statusregister för den av [H1..H4] signalerna som genererade avbrottet enligt exempel 8.12.

EXEMPEL

Nollställavbrottet från PI/T:ns signal H3

```
   MOVE.B   #%00000100,(PSR).L
```

8.12

Den enklaste formen för en avbrottsrutin som avlägsnar avbrottet visas i exempel 8.13.

EXEMPEL

Nollställavbrottet från PI/T:ns signal H3

```
IrqRut:
      MOVE.B      #%00000100,(PSR).L
      ---
      ---         GÖR  NÅGOT ARBETE
      ---
      RTE
```

Minns att en avbrottsrutin måste avslutas med *return*

8.13

Med detta har vi nu visat hur **PI/T**:n fungerar i ett sammanhang med vektoravbrott. Vid autovektoravbrott används stort sett samma initiering. Det enda som skiljer är ett nytt styrord för *Port Service Request Register* och att kretsen inte levererar något vektornummer.

Vi avslutar därmed behandlingen av PI/T'ns portar och övergår till att beskriva räknarkretsen.

8.3 PI/T:ns räknare

Tillämpningar:

tidmätningar av en yttre insignal

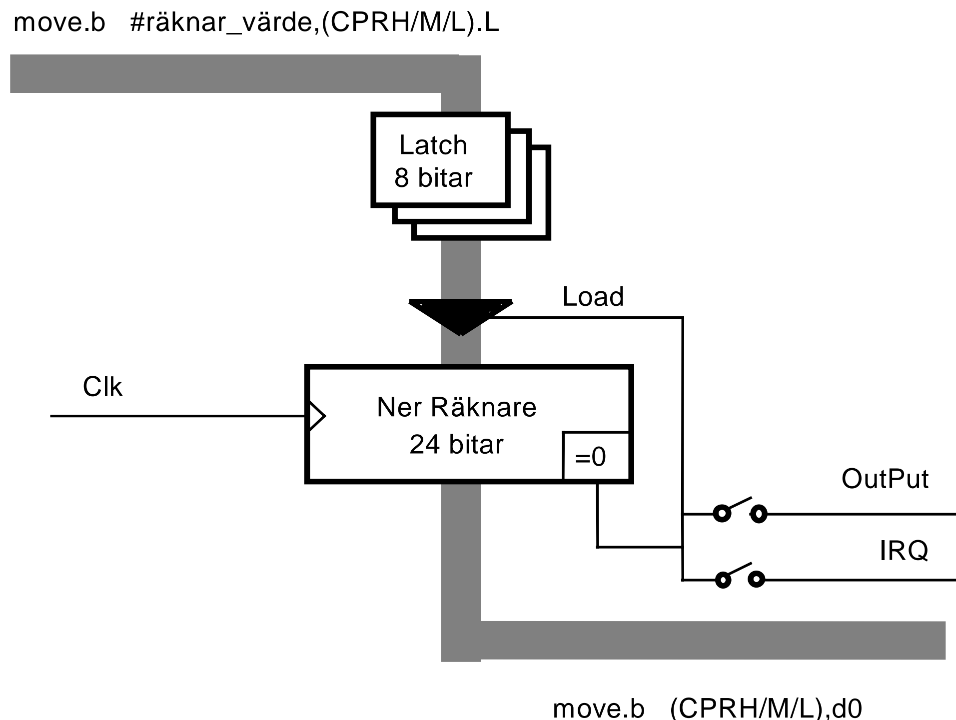
realtidsklocka

pulsgenerator

som en övervakare, mm.

PI/T'n är också bestyckad med en räknare som kan användas i många olika tillämpningar. Räknaren består av en 24 bitars synkron ner-räknare. Ner-räknaren kan klockas (räknas ner) med system klockan eller en yttre klocka som ansluts till en av ingångarna på kretsen.

Studera figur 8.16 nedan, som visar ett blockdiagram över räknaren. När vi initierar räknaren med ett startvärde läggs detta i en latch och transporteras därefter till nerräknaren. Beroende på hur vi initierat räknaren kan innehållet i latch flyttas över till nerräknaren så fort denna blivit noll (kontinuerlig mod)

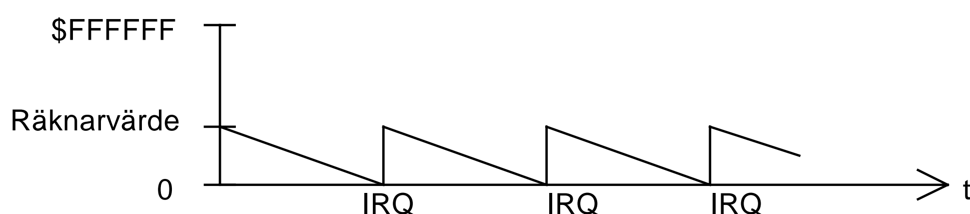


FIGUR 8.16. BLOCKDIAGRAM ÖVER PI/T:NS RÄKNARE.

När räknaren blivit noll kan avbrott genereras för att uppnå en typisk realtidsklocka. Beteckningen *OutPut* i figuren kan vara en periodisk fyrkantvåg eller en puls av en bestämd längd.

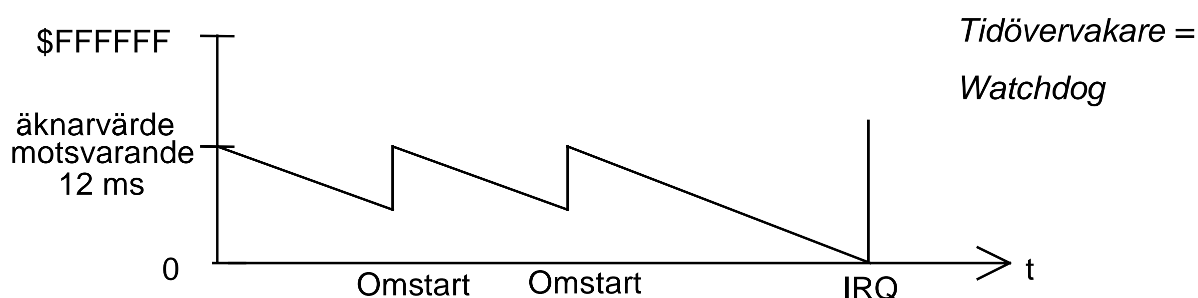
De vanligaste tillämpningarna som räknaren används i är som tidövervakare och för att generera ett periodisk klockavbrott. När räknaren programmeras för att generera klockavbrott initieras kretsen med ett räknarvärde vilket räknas ner med systemklockan. När räknaren blivit noll initieras räknaren automatiskt med sitt startvärde, fortsätter att räkna ner detta samtidigt som ett avbrott generas. På så sätt kommer

räknaren oupphörligt att räkna ner och generera periodiska avbrott. Se figur 8.17.



FIGUR 8.17 PERIODISKA KLOCKAVBROTT.

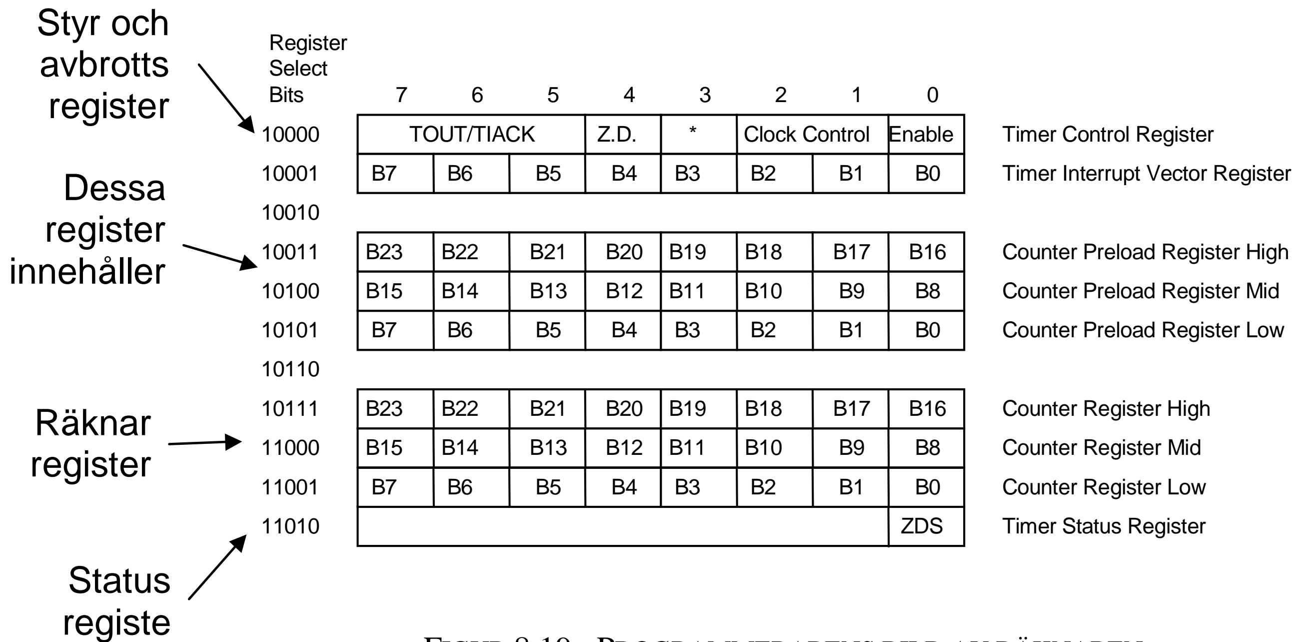
Som tidsövervakare gäller det att starta om räknaren med jämna mellanrum *innan* den genererar avbrott. Tänk dig ett kommunikationsprotokoll där mottagaren förväntar sig ett tecken var 10 ms. Om ett tecken uteblir innebär det att ett fel har uppstått i dataöverföringen vilket måste behandlas på något sätt. I sådana fall kan vi tänka oss att varje gång mottagaren tar emot ett tecken så initierar den räknarkretsen för att generera avbrott efter, säg, 12 ms för att ha god marginal. Figur 8.18 visar detta förloppet. Figuren visar att två tecken är mottagit och det tredje tecknet inte mottagits i tid och därför genererar räknaren ett avbrott.



FIGUR 8.18. RÄKNAREN SOM WATCHDOG

8.3.1 Räknarens registermodell

Räknarens registeruppsättning består av tre block där två av blocken består av 23-bitars räknarregister och det tredje består av styr, status och vektoravbrotts register. Se figur 8.19. Det ena av de två blocken som innehåller räknar register används för att *räkna ner* ett värde och det andra blocket används för att lagra ett startvärde för räknaren, se exemplet om periodiska klockavbrott i texen ovan.



FIGUR 8.19 PROGRAMMERARENS BILD AV RÄKNAREN

TCR

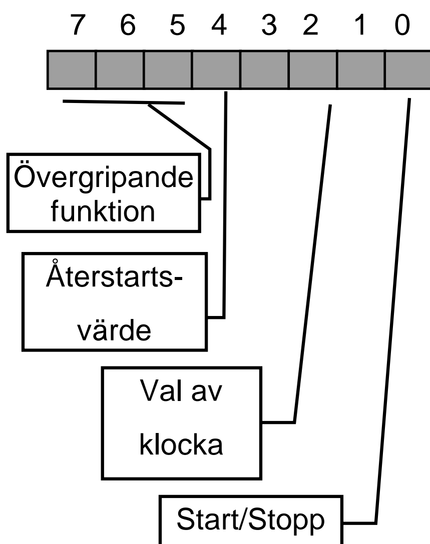
Read/Write

Adress offset $10

Timer Control Register (TCR)

Det 8-bitars styrregistert, *Timer Control Register,* **TCR**, innehåller alla styrbirarna för räknaren. Här bestämms hurvida räknaren skall generera avbrott och om detta skall vara vektoravbrott eller autovektor avbrott. Vidare bestämms om räknaren skall initieras automatiskt då den blivit noll. Slutligen bestämms vilken klock-källa som skall användas.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TOUT / TIACK Control			Z D Control	*	Clock Control		Timer Enable



Bit 7 6 5	*TOUT/TIACK Control*
0 0 x	Pinnarna PC3/TOUT och PC7/TIACK är PORT C-pinnar
0 1 x	Pinne PC3/TOUT följer TOUT-funktion och används för att generera fyrkantsvåg. Pinne PC7/TIACK är PORT C-pinne

Bit 7 6 5	*TOUT/TIACK Control*
1 0 0	Ej avbrott. Pinne PC7/TIACK följer TIAC-funktion
1 0 1	Vektor avbrott. Pinne PC7/TIACK följer TIAC-funktion
1 1 0	Ej avbrott. Pinne PC7/TIACK är PORT C-pinne
1 1 1	Autovektor avbrott. Pinne PC7/TIACK är PORT C-pinne

När bit 7 är ettställd följer alltid pinne PC3/TOUT TOUT-funktion.

Räknaren är utrustad med prescaler som används i tre av fallen. Räknaren minskas, laddas med värdet i preload-registren eller laddas med värdet $FF FF FF

Bit 4	*Zero Detect Control*
0	Räknaren laddas från Counter Preload Register (CPRH/M/L) när den räknat ner till noll.
1	Räknaren laddas med $FF FF FF när den räknat ner till 0
Bit 2 1	***Clock Control***
0 0	Pine PC2/TIN är PORT C-pinne Klockas av CLK-pinnen.
0 1	Pinne PC2/TIN är styrsignal (ENABLE) för räknaren. När TIN är låg är räknaren i HALT. Klockas av CLK-pinnen.
1 0	Klockas av PC2/TIN-pinnen (positiv flank).
1 1	Räknaren klockas av PC2/TIN-pinnen (positiv flank). (*Ingen prescaler*)
Bit 0	***Timer Enable***
0	Räknaren klockas inte, HALT
1	Räknaren klockas, ENABLE

Timer Interrupt Vector Register. (TIVR)

TIVR

Read/Write

Adress offset $11

Detta 8-bitars register skall innehålla avbrottsvektor när räknaren skall användas i avbrotts-sammanhang där vektoravbrott används. Registret som kan skrivas och läsas när som helst initieras vid RESET med $0F.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interrupt Vector Number							

Timer Status Register. (TSR)

TSR

Read/Write

Adress offset $1A

Detta register innehåller endast en bit (Bit 0) som har en FLIP-FLOP-funktion. Då räknaren har blivit eller passerat noll sätts FLIP-FLOP'en och statusbiten ettställs. Registret kan alltid läsas utan att dess innehåll ändras. Statusbiten är ettställd tills kretsen får RESET eller en skriv-access ges till detta register. Detta medför att statusbiten sätts till noll.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
*	*	*	*	*	*	*	ZDS

CPRH/M/L
Read/Write
Adress offset:
$13,14,15

Counter Preload Register High/Mid/Low (CPRH/M/L)

Detta (eller dessa) register utgör räknarens indata. Registret som utgör 24 bitar är uppdelade på tre bytes (tre adresser) CPRH, CPRM, CPRL, initieras med räknarens startvärde.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CPRH $13	D23	D22	D21	D20	D19	D18	D17	D16
CPRM $14	D15	D14	D13	D12	D11	D10	D9	D8
CPRL $15	D7	D6	D5	D4	D3	D2	D1	D0

CRH/M/L
Read
Adress offset:
$17,18,19

Counter Register High/Mid/Low (CRH, CRM, CPL)

Detta (eller dessa) register utgör räknaren. Registret som utgör 24 bitar är uppdelade på tre bytes (tre adresser) CRH, CRM, CRL, och innehåller räknarens aktuella värde. Registren kan läsas när som för ett momentant räknarvärde. En skrivning till dessa adresser innebär en normal buss cykel men ej någon ändring av registerinnehållen

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CRH $17	D23	D22	D21	D20	D19	D18	D17	D16
CRM $18	D15	D14	D13	D12	D11	D10	D9	D8
CRL $19	D7	D6	D5	D4	D3	D2	D1	D0

8.3.2 PI/T:n som realtidsklocka.

I detta avsnitt kommer vi att ge exempel på en enkel användning av räknarkretsen i PI/T'n. Vi gör detta genom att implementera ett enkelt stoppur. Funktionen kan enkelt beskrivas:

> Räknarkretsen ska *avbryta* processorn 10 gånger per/sekund, varje gång utförs en *avbrottsrutin* som uppdaterar en *klockvariabel*
>
> Vi ska (programvarustyrt) kunna *starta* och *stanna* klockan

Av detta framgår att vi måste tillhandahålla (i vart fall) *fyra* programrutiner:

Initiering av räknarkretsen (Exempel 8.17)

Rutin för att *starta* räknarkretsen (Exempel 8.18)

Rutin för att *stoppa* räknarkretsen (Exempel 8.19)

Avbrottsrutin för att *uppdatera* klockan och *kvittera* avbrott (Exempel 8.20)

Vi skriver några användbara *equate*-satser för att göra programmet tydligare. Studera exempel 8.14.

EXEMPEL

Definition av räknarens register för en MC68230 i ett typiskt MC68000-baserat system. Med16-bitars databuss och basadress $3E0000 har vi:

```
PITBASE equ   $3E0001       timer 1 base address
TCR     equ   PITBASE+$20   timer control register
TIVR    equ   PITBASE+$22   timer interrupt vector register
CPRH    equ   PITBASE+$26   timer preload HIGH
CPRM    equ   PITBASE+$28   timer preload MIDDLE
CPRL    equ   PITBASE+$2A   timer preload LOW
TSR     equ   PITBASE+$34   timer status register
```

8.14

Exempel 8.15 visar de styrord som krävs för att initiera räknaren. Vi bestämmer nu vilken *avbrottsvektor* vi ska tilldela räknarkretsen. Vi anger en *initieringsbyte* för räknarkretsen, dvs det *värde* vi måste skriva i kontrollregistret för att få önska funktion (periodiskt avbrott). Vi bestämmer också *intervall* för avbrott.

Vektornumret (TIMINT) är det *värde* som skall anges som MC68230 räknarkrets' avbrottsnummer. Motsvarande *adress* blir (alltid) *vektornummer* * 4, dvs, i detta fall 4*$40=$100

EXEMPEL

Styrord för att initiera räknaren

```
*      Timer operation parameters
TIMINT       equ    $40    timer interrupt vector number *
                           (first user supplied)
TIMADDR      equ    $100   timer interrupt vector address
TIMMODE      equ    $A0    timer operating mode
TBASEH       equ    $0     time base high byte
TBASEM       equ    $64    time base middle byte
TBASEL       equ    $00    time base low byte
```

Initieringsbyte för styrregistret TCR.

På de följande sidor beskrivs de olika styrorden..

8.15

Initieringsbyte ($A0) för styrregistret TCR (TIMMODE):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TOUT / TIACK Control			Z D Control	*	Clock Control		Timer Enable
1	0	1	0	0	0	0	0

bit 7,6,5 = 1 0 1: ger *vektoravbrott*

bit 4 =0, dvs räknaren laddas om från *Counter Preload Register* efter varje avbrott. På detta sätt bibehålls rätt räknarperiod.

bit 3 = 0, men detta är oväsentligt ty denna bit används ej.

bit 2,1 = 0 0 dvs räknaren klockas av systemets CLK-signal för vårt exempel antar vi att denna är 8 Mhz.

bit 0 = 0; dvs räknaren sätts i HALT-tillstånd

Räknarens initialvärde ($006400): TBASEx:

Tillsammans utgör alltså dessa tre värden det tidsintervall mellan vilka avbrott ska begäras. Låt oss nu se hur dessa bestäms. Vad vi anger är *det antal klockpulser* som ska utföras mellan varje avbrott. *Dessutom* har vi en faktor (32) att multiplicera detta med ty vi har valt en *pre-scaling* funktion av klockpulserna för att enkel kunna ange ett större tidsintervall i realtid. Beräkningen visas i exempel 8.16.

EXEMPEL

Beräkna tiden mellan varje avbrott och initiera räknarregistren

*Räknarregister * Klockperiod * 32*

I vårt fall:

$006400 * 1/8 Mhz * 32 =
25600 * 125 10^{-9} * 32 = 0,1024 sekund

vilket är ungefär 0,1 sekund (100 millisekunder)

Observera att detta värde skrivs som tre st bytes till de tre registren enligt:

```
MOVE.B      #0,(CPRH).L
MOVE.B      #$64,(CPRM).L
MOVE.B      #0,(CPRL).L
```

8.16

De olika subrutinerna visas i exempeln nedan.

Adressen till vår *avbrottsrutin* "ntirq" (se nedan) skrivs till adress $100 i vektor-tabellen

Vektornummer ($40) för adress $100 skrivs till

EXEMPEL

Initiera räknarkretsen

```
timinit:

   MOVE.B #TBASEH,(CPRH).L   räknarvärde bitar 16-23
   MOVE.B #TBASEM,(CPRM).L   räknarvärde bitar 15-8
   MOVE.B #TBASEL,(CPRL).L   räknarvärde bitar 7-0


   MOVE.L #ntirq,(TIMADDR).L vektoravbrott
   MOVE.B #TIMINT,(TIVR).L
   MOVE.B #TIMMODE,(TCR).L   init styrregister


   MOVE   #$2000,SR          allow interrupts
   RTS
```

Processorns *avbrottsnivå* sänks så att den kommer att betjäna avbrott från

8.17

Starta räknarkretsen

En bit i styrregistret (den minst signifikanta) bestämmer om räknarkretsen ska räkna ned vid varje klockpuls, eller ej. Att *starta* den initierade kretsen visas i exempel 8.18.

EXEMPEL

Starta räknaren

```
starttim:
    MOVE.B  #TIMMODE+1,(TCR).L  starta räknarkrets
    RTS
```

8.18

Stoppa räknarkretsen

För att stanna räknaren nollställes helt enkelt den minst signifikanta biten i styrregistret. Se exempel 8.19.

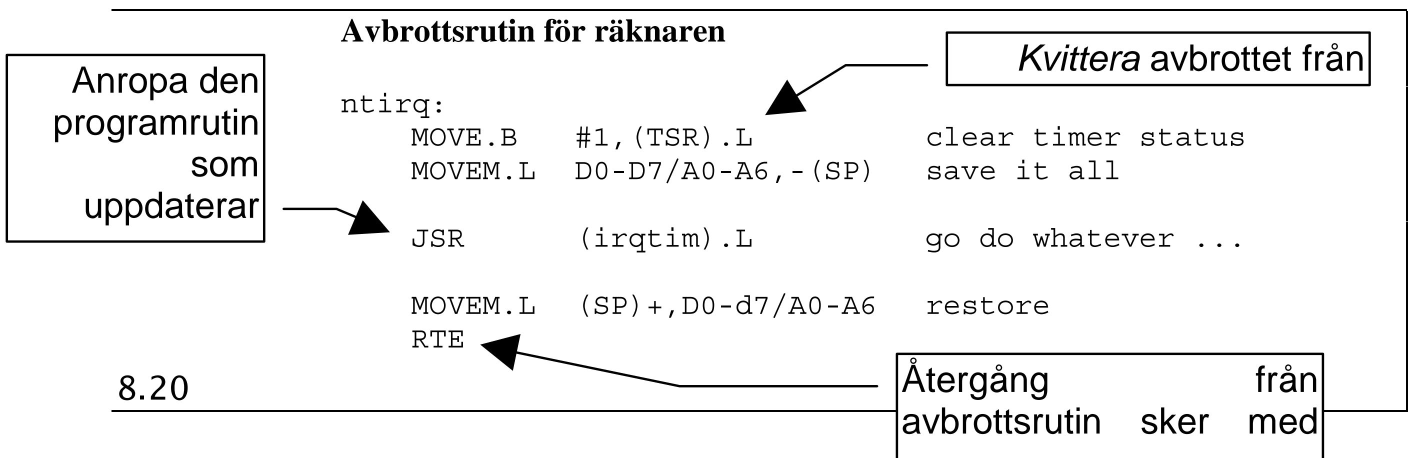
EXEMPEL

Stanna räknaren

```
stoptim:
    MOVE.B   #TIMMODE,(TCR).L   stanna räknarkrets
    RTS
```

8.19

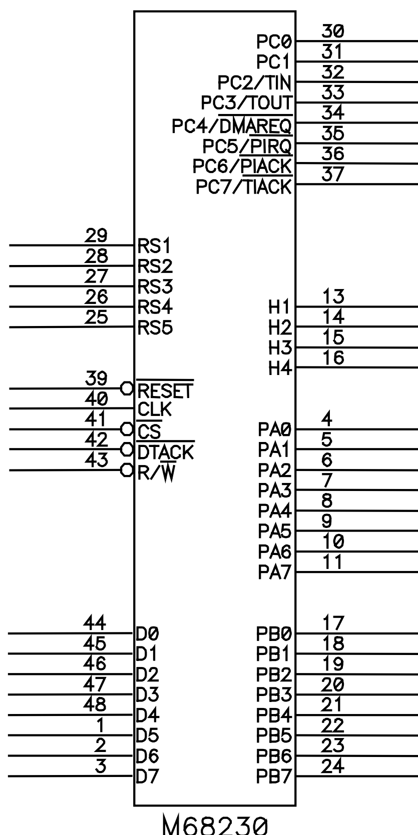
Avbrottsrutin för räknaren



För att göra stoppuret komplett måste vi naturligtvis lägga till några programrutiner. Detta lämnar vi emellertid som övning till läsaren.

8.4 Sammanfattning av MC68230 PI/T

Vi sammanfattar MC68230 PI/T:



MC68230 består av en parallell I/O-port och en räknare.

Parallellporten är uppdelat i Port A och Port B som båda kan initieras som in eller utport, dessutom finns Port C som även kan initieras för att ingå i processorns bussprotkoll.

Räknaren är en nerräknare på 24 bitar. Räknaren kan styras externt eller av mikrodatorsystemets egen klocka.

Alla register i kretsen är direkt åtkomliga för programmeraren dvs alla register har unik adress.

Kretsens anslutningar mot processorn är [RS1..5] till adressbussen, databussen [D0..7] och ett antal styrsignaler. Se figur 8.20. Vissa av styrsignalerna är placerade på Port C och måste därför prorrammeras för korrekt funktion. Kretsens anslutningar mot yttre enhet är port A, Port B och Port C.

Det finns 4 så kallade handskakningssignaler [H1..4].

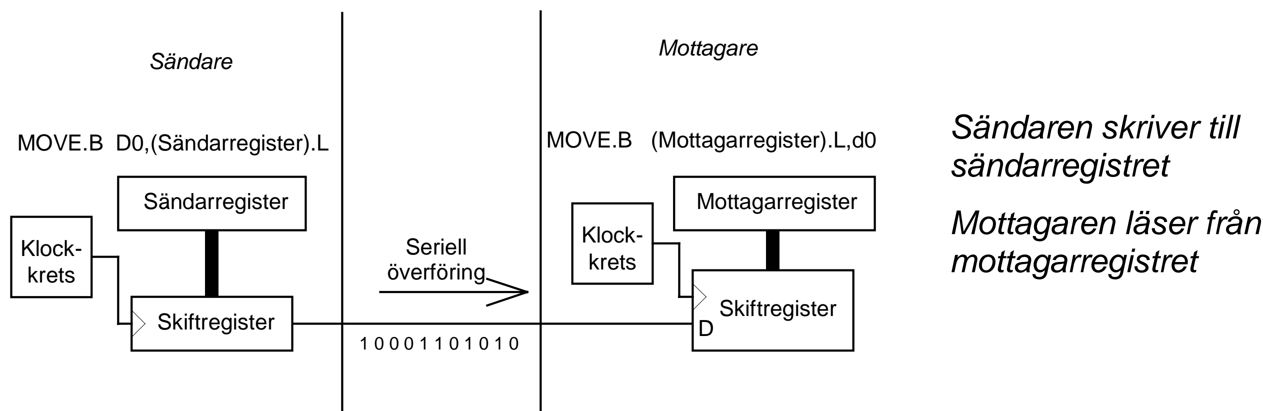
FIGUR 8.20 MC68230 PI/T

8.5 DUART – MC68681

I de följande avsnitten skall vi beskriva hur seriekommunikationskretsen *MC68681* fungerar. Kretsen tillhör MC68000-familjens periferikretsar och är därför anpassad för MC68000-buss system. Innan vi går in på hur kretsen fungerar i detalj gör vi en snabb repetition av seriell överföring. Eftersom MC68681 är mycket väl lämpad att användas för det seriella formatet RS232 skall vi dessutom studera detta lite närmre.

8.5.1 Seriekommunikation och RS232

Seriell överföring används mestadels mellan system som fysiskt är placerade med ett stort inbördes avstånd. Som vi diskuterade i kapitel 6 skulle kostnaden för att ansluta en traditionell 8- eller 16-bitars bred databuss mellan Stockholm och Göteborg bli mycket stor. Visserligen skulle vi kunna överföra data parallellt på denna buss, men det är seriekommunikation som framför allt av kostnadsskäl, väljs i dag. Se figur 8.21.



FIGUR 8.21 SERIELL ÖVERFÖRING

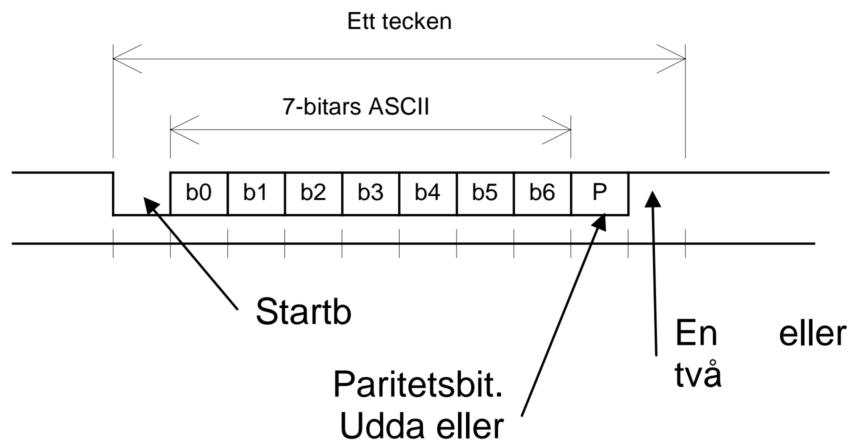
Då sändare och mottagare är placerade på stora inbördes avstånd väljs *asynkron* kommunikation. Detta innebär att sändaren skickar ut sitt seriella meddelande i sin *egen* takt, och mottagaren, som har blivit initierad med att överföringstakten skall vara *inom vissa toleranser*, försöker läsa in mottaget data så gott det går.

Då detta förfarande kan ge upphov till överföringsfel brukar man utöka överfört data med någon form av kontrollinformation. Som vi såg i kapitel 6 kan detta vara en paritetsbit ellen en så kallad CRC (Cyclic Redundancy Check) som används i datanätsammanhang.

Det seriella överföringsformatet RS232 visas nedan. Ett tecken (en byte) överförs i taget. Varje tecken är omslutet av en startbit, en eller två stoppbitar och eventuellt en paritetsbit. Se figur 8.22. Pariteten kan vara bestämd som udda eller jämn. Pariteten används av mottagaren för

att kontrollera om det mottagna tecknet kan vara felaktigt. Vidare används en högre spänning (±10V) för den seriella signalen. Detta är för att vara mer okänslig mot störningar. Se figur 8.23. Överföringshastigheten anges i birat per sekund, BAUD. Se exempel 8.21.

Observera att paritetskontrollen upptäcker alla enkelbitsfel. *Med enkelbitsfel menas att en bit i meddelandet är skickat som en "etta" och mottaget som en "nolla" eller vice versa. Den överförda biten har blivit "störd" under överföringen. Mottagaren som exempelvis är initierad till seriell överföring med udda paritet kommer nu direkt att uptäcka att ett fel inträffat då den erhåller ett jämnt antal ettor i stället för ett udda antal. Fun-dera nu själv på vad som hän-der om två, tre, fyra osv bitar störs i det överförda meddelan-det. Vilka fel upptäcks?*



FIGUR 8.29. SERIEFORMAT MELLAN TERMINAL OCH DATOR.

Med *jämn paritet* menas att antal ettor i det överförda meddelandet skall vara en *jämt* tal (2, 4, 6 osv.). Sändaren sätter då paritetsbiten till noll eller ett för att uppfylla att antal ettställda överförda bitar i meddelandet är jämt. Mottagaren kan på så sätt kontrollera det mottagna meddelandet genom att antalet ettor är ett jämnt antal. Vid *udda paritet* skall antal mottagna databitar vara udda.

EXEMPEL

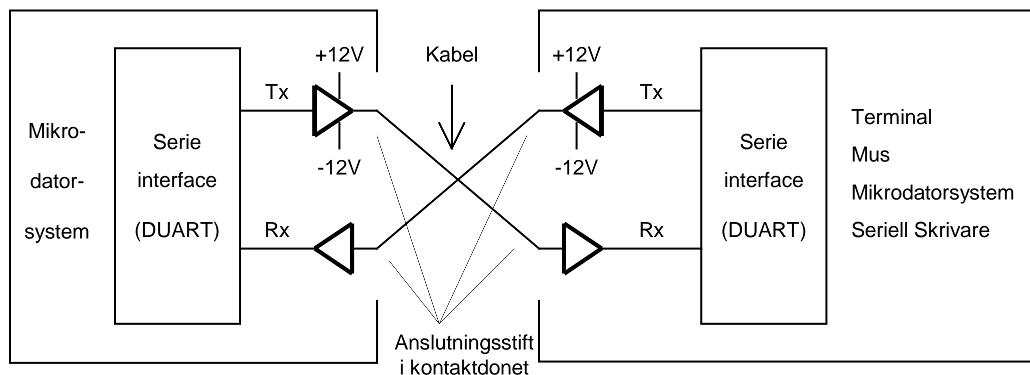
Beräkna antal bitar per sekund

Hastigheten som *bitströmmen* i det överförda meddelandet betecknas med antal bitar per sekund eller *BAUD-RATE*. När man anger att BAUD-RATE exempelvis är 4800 BAUD så innebär detta att det totala antalet bitar som överförs är 4800 bitar per sekund, paritetsbitar och dylikt medräknat. Detta innebär att periodtiden för en bit blir:

$$t = \frac{1}{4800\, {}^{bitar}\!/_{s}} = 2.08 \cdot 10^{-4}\, s$$

8.21

Dagens användning av RS232 exemplifieras av COM-portarna på en persondator. En sådan är bestyckad med en 9- eller 25-polig hankontakt som kallas D-SUB. Här ansluter man exempelvis: modem, mus, seriell skrivare, mm eller rent generellt där man önskar seriell överföring som inte kräver allt för höga prestanda.

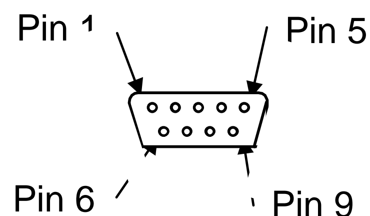


FIGUR 8.23 BLOCKDIAGRAM ÖVER RS232-ÖVERFÖRING MELLAN ETT MIKRODATORSYSTEM OCH ANNAN UTRUSTNING

Pinkonfigurationen för en 9-polig D-SUB anslutning visas i figur 8.24 nedan och i figur 8.25 visas pinplaceringen i anslutningen. Anslutningarna DCD, RTS och CTS är handskakningssignaler som ibland används i samband med seriell överföring. Dessa kallas även "modem"-signaler då dom används i de fall överföring via modem används. Då fungerar dessa signaler som handskakningssignaler mellan modem och datautrustning både på sändarsidan och på mottagarsidan.

Pinplacering för9 polig D-SUB RS232		
Pin	Funktion	Kommentar
1	DCD	Data Carrier Detect
2	RxDB	Receive Data
3	TxDB	Transmitt Data
5	GND	Signaljord
7	RTS	Request to Send
8	CTS	Clear to Send

FIGUR 8.24 PINPLACERING FÖR EN D-SUB



FIGUR 8.25 PINPLACERING FÖR EN D-SUB HONA ANSLUTNING SEDD FRÅN LÖDSIDAN

Att överföra enligt formatet RS232 har använts under mycket lång tid. Det var mycket vanligt under den tid då *hårdvaran* var kostsam och ett företag endast hade *en dator* som alla användare var anslutna till. Man använder då RS232-överföring mellan datacentralen och de olika kontorsrummen i företaget.

8.5.2 Översikt över DUART MC68681

DUART MC68681 innehåller två seriegränssnitt eller *kanaler* för asynkron seriekommunikation. Kanalerna kan arbeta oberoende av varandra. Dessutom finns en räknare som kan användas för klockning av seriegränssnittens skiftregister (användas för generering av korrekt BAUD-RATE).

Figur 8.26 visar ett blockdiagram över MC68681. Kretsen är anpassad till MC68000's bussprotokoll och kan därför enkelt anslutas till processorns bussar.

Kretsen kan indelas i ett antal block för seriekanal A (Channel A), seriekanal B (Channel B) och två parallella portar [IP5..IP0] och [OP7..OP]. Slutligen finns ett block som består av en räknare som används för att generera korrekt BAUD-RATE.

DUART kännetecknas av följande egenskaper:

MC68000 bus kompatibel

Två oberoende asynkrona sändar/mottagar kanaler. Kanal A och B

Fyra bytes kan buffras vid mottagning

Två bytes kan buffras vid sändning

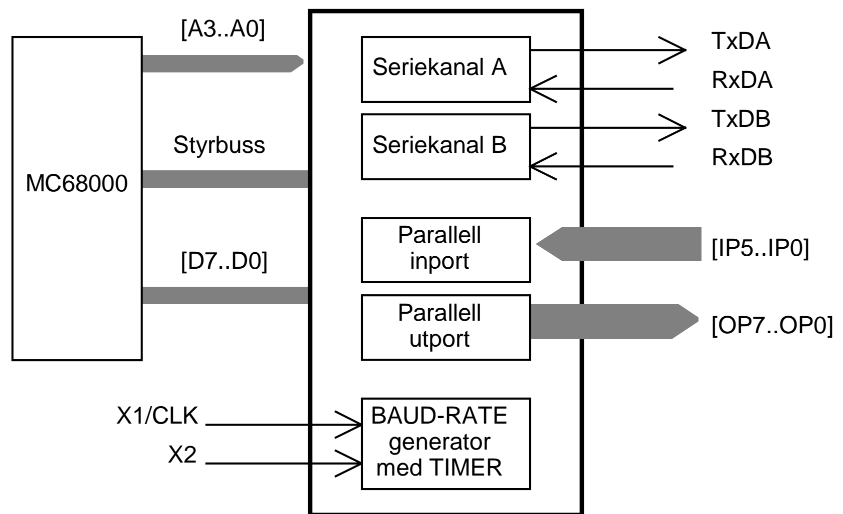
Programmerbar BAUD-RATE

Programmerbart seriellt dataformat

Feldetektering- och indikering för fel seriellt överförings-format.

Multi function utport för avbrott och seriella styrsignaler, [OP7..OP0].

Multi function inport för seriella status- och



FIGUR 8.26 BLOCKDIAGRAM ÖVER MC68681

Registeruppsättningen för DUART MC68681 visas i figur 8.27. Som vi påpekade när vi beskrev PI/T:n så är detta *programmerarens* bild av DUART MC68681.

I tabellen hittas några register (adresser) utmärkta med punkt (●). Dessa adressangivelser innehåller två olika register. Hur man skiljer registren åt och hur man adresserar dessa anges i registerbeskrivningen längre fram i detta avsnitt.

Observera speciellt de register som är utmärkta med två punkter (●●). Dessa används av MOTOROLA under uttestning av nyproducerade kretsar. Att läsa eller skriva i något av dessa register ger (eller kan ge) upphov till ändrad funktion hos kretsen och att den seriella överföringen störs. Därför, adressera *aldrig* dessa register.

Som vi ser av programmerarens bild av DUART:en och av figur 8.26 så finns två seriekanaler A och B som båda har en seriell sändarutgång och mottagaringång. Båda kan buffra fyra bytes vid mottagning och två bytes vid sändning. De två seriekanalerna kan initieras för att arbeta på två olika sätt och oberoende av varandra.

Pinnarna på de två parallella portarna har dubbla funktioner. De kan användas som handskakningssignaler eller klockingångar för den

seriella kommunikationen eller som generella in- eller ut- signaler. Slutligen kan pinnarna på de parallella portarna användar i samband med räknarkretsen i BAUD-RATE blocket.

Offset	**Register namn och förkortningar**	
	READ	WRITE
$0	Mode Register A• (**MR1A,MR2A**)	Mode Register A• (**MR1A,MR2A**)
$1	Status Register (**SRA**)	Clock Select Register A (CSRA)
$2	Får ej användas••	Command Register A (CRA)
$3	Receiver Buffer A (**RBA**)	Transmitter Buffer A (**TBA**)
$4	Input Port Change Register (**IPCR**)	Auxiliary Control Register(ACR)
$5	Interrupt Status Register (**ISR**)	Interrupt Mask Register (IMR)
$6	Current MSB of Counter (**CUR**)	Counter/Timer Upper Register (CTUR)
$7	Current LSB of Counter (**CLR**)	Counter/Timer Lower Register(CTLR)
$8	Mode Register B (**MR1B,MR2B**)	Mode Register B (**MR1B,MR2B**)
$9	Status Register B (**SRB**)	Clock Select Register B (CSRB)
$A	Får ej användas••	Command Register B (CRB)
$B	Receiver Buffer B (**RBB**)	Transmitter Buffer B (**TBB**)
$C	Interrupt Vector Register (**IVR**)	Interrupt Vector Register (**IVR**)
$D	Input Port (**IP**)	Output Port Configuration Reg(OPCR)
$E	Start Counter Command (**STAC**)	Output Port Register, bit set (OPR)
$F	Stop Counter Command (**STOC**)	Output Port Register, bit reset (OPR)

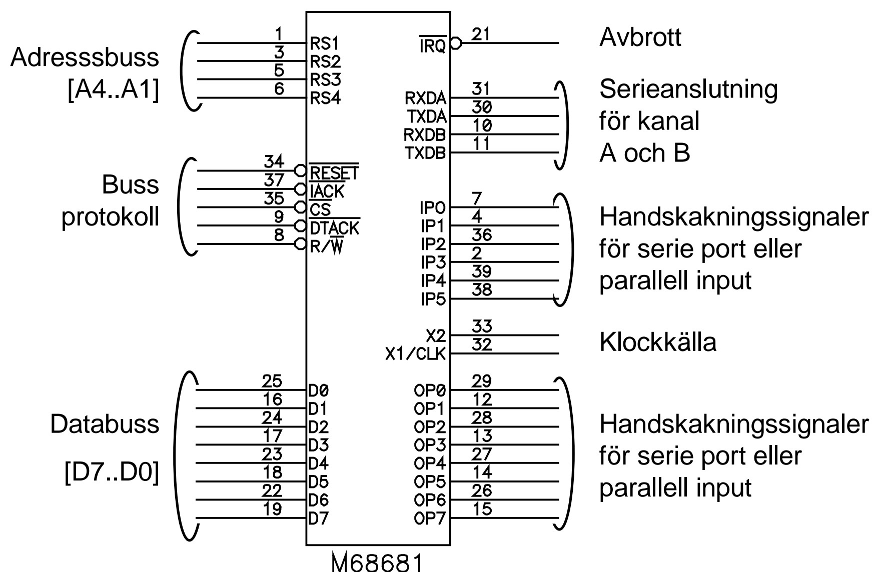
• På dessa adresser finns i själva verket 2 olika register (se nedan)

•• Dessa register används för *fabrikstest* och får *inte* vare sig läsas eller skrivas.

FIGUR 8.27 PROGRAMMERARENS BILD AV MC68681

I blocket med BAUD-RATE generatoren genereras klocksignalerna som klockar skiftregistren i seriekanalerna. Räknarkretsen kan användas i detta sammanhang.

Om vi jämför räknaren den som finns i **PI/T** MC68230, så är räknaren i BAUD-RATE blocket mindre (16 bitar mot **PI/T**:ns 24 bitar). Vidare är registeruppsättningen annorlunda. Annars är den funktionella användningen av räknaren den samma som för **PI/**T:n eftersom den kan användas som en generell räknare i sammanhang som: realtidsklocka, vakthund (Watch Dog), mm. Av blockdiagrammet i figur 8.26 kan vi se att olika grupper av anslutningar till DUART.

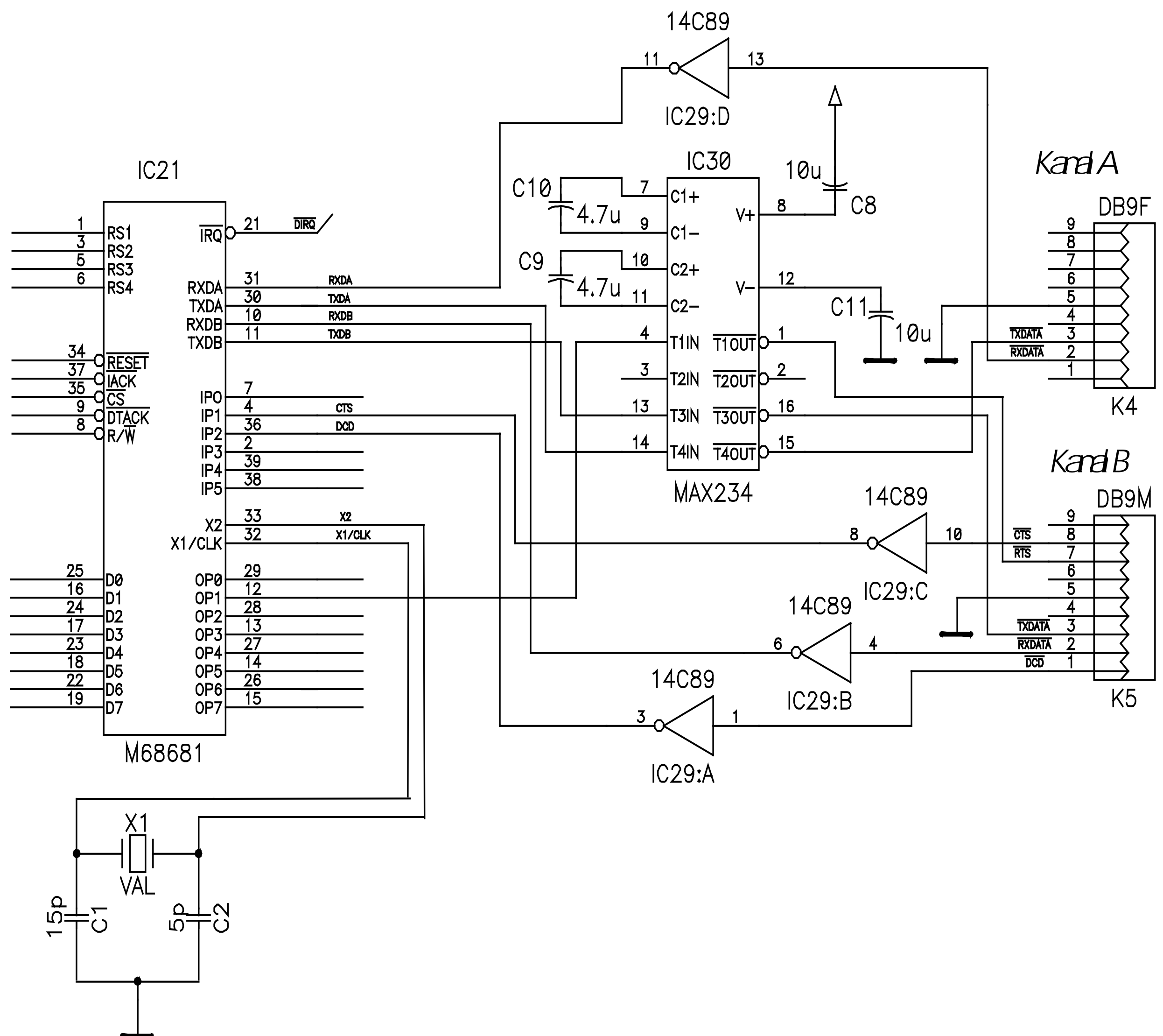


FIGUR 8.28 MC68681'S PINPLACERING

Anslutningarna visas i sin helhet i Figur 8.28. Figuren visar hur kretsen ansluts med fyra adress-ledningar mot processorn. Vidare känner vi igen signalerna son ingår i MC68000's bussprotokoll. Som övriga periferikretsar så används en 8 bitars bred databuss för kommunikation med kretsen.

Anslutningarna för den seriella kommunikationen är TxDA och RxDA för seriekanal A och TxDB/RxDB för B-kanalen. Vidare finns ett antal in- och ut pinnar [IP5..IP0] och [OP7..OP0] som antingen används som handskakningssignaler för den seriell kommunikationen eller som generella in och utsignaler. Handskakningssignaler för den seriella kommunikationen betecknas *modemsignaler* i vissa sammanhang. Som klockkälla använd antingen en kristall eller en klocksignal.

Om vi kort studerar ett hårdvaruschema över ett seriellt gränssnitt där MC68681 ingår ser vi exempel på hur kanal A används utan handskakningssignaler och kanal B utnyttjar handskakningssignaler för seriekommunikation. Studera figur 8.29.



FIGUR 8.29. MC68681 OCH RS232 ANPASSNING

Observera att drivkretsar (MAX234) och buffertar (14C89) krävs för att uppnå anpassning mellan RS232's spänningsnivåer och vad som används för DUART:en. MAX234 har intern späningsomvandlare och kan därför generera ca 10 V's signalnivå för seriekommunikationen. Studera även figur 8.30 som visar en skematisk bild över RS232 snittet.

Observera också att i denna koppling används en kristall som klockkälla. Lämplig frekvens på kristallen är 3.6864 Mhz.

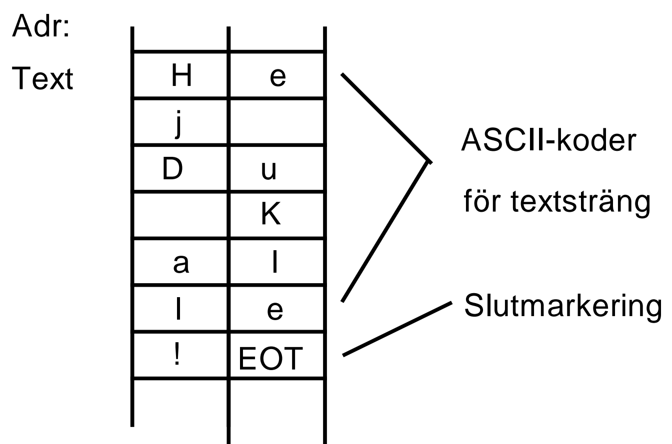
Observera att en del MOTOROLAS microcontrollers, exempelvis MC68340, har en "DUART MC68681" på chip'et. Denna DUART:n är "nästan" kompatibel med MC68681, bortsett från att räknaren (TIMER) saknas och dessutom är registrens adresser ändrade. Du kan dock, med mycket små ändringar, använda samma program för att kommunicera med en fristående DUART som den som finns i MC68340 så länge inte räknaren används.

Här skall vi inrikta oss på hur den seriella delen av kretsen fungerar och hur man kan kommunicera seriellt med hjälp av kretsen. Vi kommer därför att i nästa kapitel att visa på en enkel användning av DUART.

8.5.3 Enkel användning av DUART.

Som vi såg i blockdiagrammet i figur 8.26 består DUARTEN av två seriekanaler. Båda två fungerar på samma sätt och vi ska nu ge ett par exempel på hur man med enkla programrutiner kan skicka och ta emot seriellt data med kretsen.

Vi förutsätter till en början att kretsen är initierad för ett bestämt seriellt kommunikationsformat och att vi utnyttjar seriekanal A. Först studerar vi hur man kan *skicka* seriellt data med kretsen och förutsätter då att det finns en dataarea i minnet som skall skickas via seriekanal A. Se figur 8.30. Dataarean som är placerad på adressen *Text* i minnet innehåller en textsträng med ASCII-koder. Vidare är textsträngen avslutad med tecknet End Of Text (*EOT*) som indikerar att textsträngen är slut. Betrakta nu programmerarens bild av kretsen i figur 8.27. Kanals A's sändar register (Transmitter Buffer A - **TBA**) hittas i figuren på offset 3.



FIGUR 8.30 DATAAREA I MINNET

Hur ska nu ett program som sänder denna textsträngen via DUART:en se ut? Vårt första försök illustrerar en *felaktig* lösning. Se exempel 8.22. Principen i programutkastet är emellrtid riktig. Vi använder en slinga för att läsa ett ASCII-tecken från minnet som skrivs i kanal A:s sändar-

register i DUART:n. Det är bara det att snurran genomlöps på några få µs. På denna korta tid hinner inte DUART skifta ut en byte på serieanslutningen, och tecknen i sändar registret kommer därför att skrivas över.. Vi måste se till att synkronicera processorns utmatningstakt till vad DUART:n klarar av. Studera därför programmerarens bild av DUART och se speciellt statusregistret för kanal A. Registret visas även i figur 8.31.

EXEMPEL

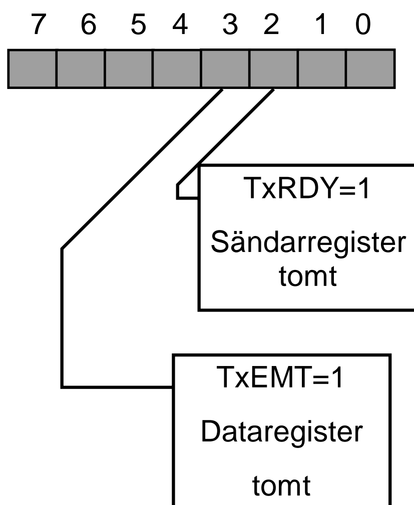
Utskriftsrutin utan att ta hänsyn till att tecknet har skiftats ut ur skiftregistret!

```
EOT    EQU        $04                 Slutmarkering

       MOVE.L     #Text,A0            läs startadress
loop:
       MOVE.B     (a0)+,D0            läs tecken
       CMPI.B     #EOT,D0             sista tecken?
       BEQ        exit
       MOVE.B     D0,(duart_TBA).L    sänd tecken
       BRA        loop
exit:
       ---
```

8.22

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Received Break	Framing Error	Parity Error	Overrun Error	TxEMT	TxRDY	RxFIFO FULL	RxRDY

FIGUR 8.31. DUART:NS STATUSREGISTER



Registrets fyra mest signifikanta bitar innehåller statusbitar som indikerar felaktig seriell överföring. Studera exempelvis bit 5 som indikerar paritetsfel. De fyra minst signifikanta bitarna innehåller information om mottagarregistren innehåller seriell mottaget data som inte ännu är läst av processorn, eller, om sändarregistren kan initieras med ny data som skall skickas seriellt.

I vårt exempel önskar vi att undersöka om sändardelens dataregister är redo att ta emot ett nytt ASCII-tecken som skall skickas. Vi kan gissa av figur 8.31 att bit 3 och 2 har med sändning att göra då dessa bitar är betecknade med Tx (Transmit). Bitarnas betydelse visas nedan.

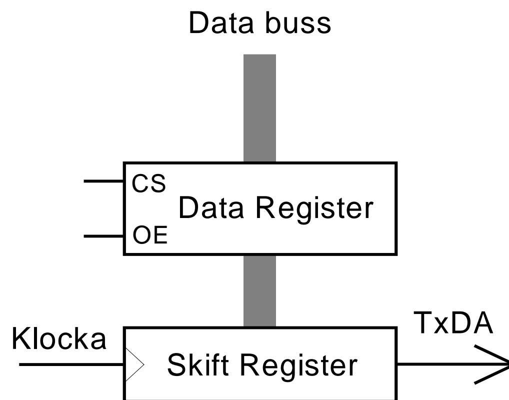
Bit 3	*TRANSMITTER EMPTY*
0	Sändardelen (dataregister och skiftregister) innehåller data att sändas
1	Sändardelen (dataregister och skiftregister)är tom och kan laddas med nytt data

Biten nollställs när processorn skriver data till dataregistret

Bit 2	*TRANSMITTER READY*
0	Sändarregistret (dataregistret) innehåller data
1	Sändarregistret (dataregistret) är tomt

Biten nollställs när processorn skriver data till dataregistret

För att förstå skillnaden på dessa statusbitar kan det vara lämpligt att studera ett blockdiagram över sändardelen. Som figur 8.32 visar är sändardelen buffrad. Detta innebär att när ett tecken överförs från dataregistret till skiftregistret kan processorn direkt skriva in ett nytt tecken till sändarens dataregister.



FIGUR 8.32. DUART:NS SÄNDARDEL

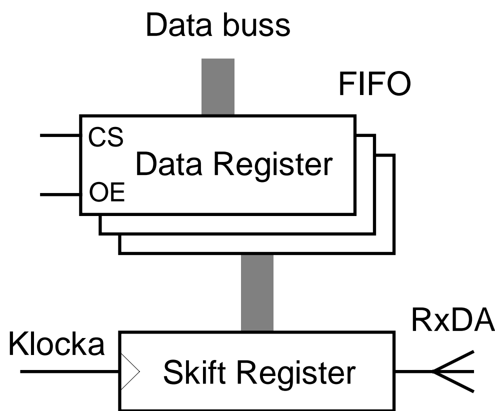
För vårt exempel behöver vi inte beakta bit 3 ty denna indikerar att *både* skiftregister och dataregister är tomma. Vi observerar i stället att bit 2, TxRDY; indikerar att dataregistret är tomt när biten är ettställd. Vi måste utöka vårt program med en "busy-wait" snurra så att vi anpassar processorns utmatningstakt till DUART:ns. Se exempel 8.23.

EXEMPEL

Utskriftsrutin som inväntar att tecknet har skiftats ut ur skiftregistret!

```
EOT   EQU         $04          Slutmarkering

      MOVE.L      #Text,A0     läs startadress
loop:
      BTST.B      #2,(duart_SRA).L  testa TxRDY
      BEQ         loop
      MOVE.B      (A0)+,D0     läs tecken
      CMPI.B      #EOT,D0      sista tecken?
      BEQ         exit
      MOVE.B      D0,(duart_TBA).L  sänd tecken
      BRA         loop
exit:
      ----
```

8.23



FIGUR 8.33. DUART'S MOTTAGARDEL.

Om vi studerar hur mottagning med DUART:n går till så är principen den samma som för sändning. Vi måste även här testa en statusbit som indikerar om något tecken är mottaget och alltså finns tillgängligt i mottagarens dataregister. Om vi studerar blockdiagrammet över mottagardelen i figur 8.33 så ser vi att det finns ett litet FIFO. Detta består av tre dataregister (tre bytes) utöver skiftregistret. Detta innebär att mottagardelen kan ta emot fyra tecken utan att "tappa" mottaget data.

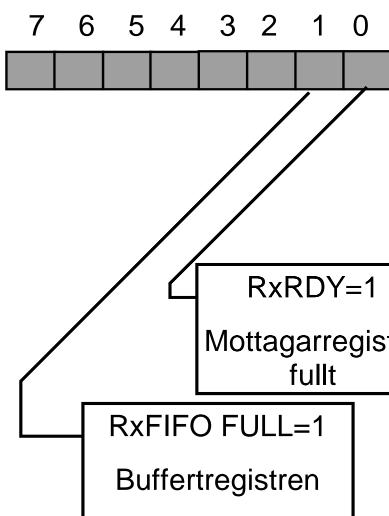
Om vi studerar statusregistret i figur 8.31 på nytt ser vi att bit 0 (RxRDY) och bit 1 (FFULL) berör mottaget data. Betydelsen av bitarna är följande:

Biten nollställs när processorn tömt mottagardelen på data

Bit 1	*RECEIVER FIFO FULL*
0	Mottagardelen är inte full
1	Mottagardelen är full

Biten nollställs när processorn tömt mottagardelen på data

Bit 0	*RECEIVER READY*
0	Mottagardelen är tömd på data
1	Data finns att läsa i mottagardelen



Bit 1 (FFULL) indikerar alltså att mottagardelen är *fylld* eller inte. Detta innbär att alla dataregistren i FIFO och skiftregistret innehåller data, och skulle ytterligare data skickas till DUART:n så kommer detta att förloras.

Bit 0 (RxRDY) däremot indikerar om det finns *någon* data i mottagardelen. Är denna biten ettställd innebär det att det finns åtminstonde ett tecken som processorn kan läsa från mottagar delen.

Ett program som läser in en mottagen ASCII-sträng från DUART:n visas i exempel 8.24. Vi placerar den mottagna strängen som *skall vara* avslutad med slutmarkeringen EOT på adress `InText` i minnet. Vi väljer här i programmet att testa bit 0 (RxRDY) i statusregistret då denna bit indikerar om *något* tecken har anlänt.

EXEMPEL

Läs en textsträng från duart till minne

```
EOT    EQU          $04                  Slutmarkering

       MOVE.L       #InText,A0           läs startadress
loop:
       BTST.B       #0,(duart_SRA).L     testa RxRDY
       BEQ          loop
       MOVE.B       (duart_RBA),D0       läs tecken
       MOVE.B       D0,(A0)+             spara tecken
       CMPI.B       #EOT,D0              sista tecken?
       BNE          loop
       ----
```

8.24

Nu har vi visat två programrutiner, en där vi skickar data och en där vi tar emot data. Nästa del blir att studera hur vi skall initiera DUART:n för att kommunicera enligt ett bestämt seriellt överföringsformat. Studera exempel 8.25. Vad som ingår, eller kan ingå, i ett bestämt överföringsformat är exempelvis:

antal databitar i varje överförd tecken

antal stopbitar i varje tecken

om udda, jämn eller ingen paritet ska användas

hurvida modemhandskakningssignaler ska användas eller inte

BAUD-RATE (överföringshastighet)

Vilken klock-källa skall användas för att generera korrekt BAUD-RATE

EXEMPEL

Vi visar här en typisk initi-eringsrutin för DUART (ej avbrott). Rutinen initierar kretsen för: ingen paritet; 8 databitar; stoppbitens längd =1.000; 9600 BAUD

```
IniDuart:
       MOVE.B       #$13,(duart_MR1A).L
       MOVE.B       #$07,(duart_MR2A).L
       MOVE.B       #$BB,(duart_CSRA).L
       MOVE.B       #$05,(duart_CRA).L
       RTS
```

8.25

Den första instruktionen initierar det första mod-registret (MR1A) till %00010011 (=$13). Det innebär att bitarna 4, 1 och 0 är ettställda och

de övriga nollställda. Det är i detta register tillsammans med det andra mod-registret (MR2A) man anger kretsens mod. Studera figur 8.34 nedan som visar registrets innehåll.

Styrord = $13

0 0 0 10 0 11

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Rx RTS Control	Rx IRQ Select	Error Mode	Parity Mode		Parity Type	Bits per Character	

FIGUR 8.34. MODE REGISTER 1

Vid initieringen ettställer vi bit 4 och nollställer bit 3. Se nedan.

Detta innebär att vi här väljer "ingen paritet" för den seriella

Bit 4 3	*PARITY MODE*
0 0	Addera udda eller jämn paritet
0 1	Addera hög eller låg paritetsbit
1 0	Ingen paritet
1 1	Multidrop Mode

Vi ettställer även bit 0 och 1. Dessa anger antal databitar som skall ingå i varje överfört tecken.

Tabellen visar att vi har valt 8 databitar.

Bit 1 0	*BITS-PER-CHARACTER*
0 0	5 databitar per tecken
0 1	6 databitar per tecken
1 0	7 databitar per tecken
1 1	8 databitar per tecken

De övriga bitarna, som är nollställda, innebär inget i vår *enkla* initiering av kretsen. Självfallet har de betydelse och för den vetgirige hänvisar vi till registerbeskrivningarna längre fram i detta kapitel.

Nästa instruktion initierar Mode Register 2 (MR2A) till %00000111 (=$07). Detta innebär att vi väljer *normal mod* och en viss längd på stoppbiten. Se figur 8.35 och bitarnas betydelse längre fram. Observera bit 6 och 7. Bitarnas betydelse visas nedan.

Styrord = $07

00 0 0 0111

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Channel Mode		Tx RTS Control	CTS Enable Transmitter	Stop Bit Length			

FIGUR 8.35 MOD REGISTER 2

Stoppbitens längd bestäms av ett binärtal som anges i bitarna 3-0. Är dessa bitar nollställda, eller mer korrekt, är *binärtalet* noll så erhålls den

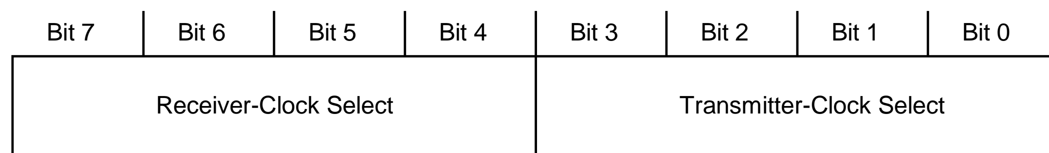
kortast möjliga stoppbiten (0.563~½ stoppbit). Den längsta stoppbiten vi kan få är när binärtalet har sitt högsta värde %1111.

Bit 7 6	***CHANNEL MODE SELECT***
0 0	Normal mod. Sändare/mottagare arbetar oberoende av varandra
0 1	Automatisk eko-mod. Sändaren skickar direkt eventuell mottaget data
1 0	Diagnostik mod. Local loop-back
1 1	Diagnostik mod. Remote loop-back
Bit 3 2 1 0	***STOP BIT LENGTH***
0 0 0 0	Stoppbitens längd blir 0.563
0 1 1 1	Stoppbitens längd blir 1.000
1 1 1 1	Stoppbitens längd blir 2.000

Vi anger normal mod.

Då vi anger $7 i detta fält erhåller vi en stopp-bit med längden 1.0.

Efter att ha initierat mod-registren skall vi bestämma korrekt BAUD-RATE. Detta görs i Clock Select Register A (CSRA). Registret är indelat i två halvor där vi bestämmer mottagarklockan i den ena halvan och sändarklockan i den andra. Dessa två halvor är nästan identiska och vi visar därför här endast den ena halvan. Se figur 8.36.



Styrord = $BB

1011 1011

FIGUR 8.36 CLOCK SELECT REGISTER

Bit 3 2 1 0	**ACR bit 7 = 0**	**ACR bit 7 = 1**
0 0 0 0	50 BAUD	75 BAUD
0 0 0 1	110 BAUD	110 BAUD
0 0 1 0	134.5 BAUD	134.5 BAUD
0 0 1 1	200 BAUD	150 BAUD
0 1 0 0	300 BAUD	300 BAUD
0 1 0 1	600 BAUD	600 BAUD
0 1 1 0	1200 BAUD	1200 BAUD
0 1 1 1	1050 BAUD	2000 BAUD
1 0 0 0	2400 BAUD	2400 BAUD
1 0 0 1	4800 BAUD	4800 BAUD
1 0 1 0	7200 BAUD	1800 BAUD
1 0 1 1	9600 BAUD	9600 BAUD
1 1 0 0	38.4k BAUD	19.2k BAUD
1 1 0 1	TIMER	TIMER
1 1 1 0	IP3-16X	IP3-16X
1 1 1 1	IP3-1X	IP3-1X

Av tabellen ser vi att vi valt 9600 BAUD då vi initierat detta register med %1011 1011 (=$BB). Ur tabellen ser vi att för vissa bitmönster så kan två olika BAUD-RATE erhållas. Vilken BAUD-RATE som väljs beror på bit 7 i register ACR.

Slutligen initieras Command Register A (CRA) med %00000101 (=$05). Detta register används för att starta och stanna både sändare och mottagare. Studera figur 8.37 och bitarnas betydelse nedan.

Styrord = $05

0 000 01 01

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NOT USED	Miscellaneous Commands			Transmitter Commands		Receiver Commands	

FIGUR 8.37 COMMAND REGISTER

Vi ettställer här bit 2 och 0 för att aktivera sändare och mottagare.

Bit 3 2	*TRANSMITTER COMMAND*
0 0	Ingen ändring av nuvarande mod
0 1	Aktivera sändaren
1 0	Stanna sändaren
1 1	Ej tillåtet
Bit 1 0	***RECEIVER COMMAND***
0 0	Ingen ändring av nuvarande mod
0 1	Aktivera mottagaren
1 0	Stanna mottagaren
1 1	Ej tillåtet

Med denna enkla initiering har vi alltså initierat kanal A på DUART:n för seriell överföring med följande egenskaper:

Normal mod

BAUDRATE :	*9600*
Antal databitar :	*8*
Antal stoppbitar :	*1.000*
Paritet:	*Ingen paritet*

Vidare bör här påpekas att DUART inte är initierad för att generera avbrott mot processorn. Ett exempel på avbrottsinitiering ges längre fram i kapitlet.

Innan exempel på avbrottsinitiering ges så beskrivs först kretsens alla register. Vi föreslår att du åtminstonde läser den kortfattade översiktliga registerbeskrivningen som ges direkt efter varje figur på de följande sidorna.

8.5.4 Beskrivning av kretsens register

Observera att beskrivningen på de följande sidorna behandlar kanal A's registeruppsättning. Kanal A's och kanal B's register är i de flesta fall identiska och för att initiera kanal B så hänvisas till kanal A's registeruppsättning. I de fall skillnader uppträder så framgår dessa av texten.

Mode Register 1A (MR1A)

MR1A

Read/Write

Adress offset $0

Detta registret tillsammans med MR2A bestämmer kretsens arbetsmod och formatet på den seriella överföringen.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Rx RTS Control	Rx IRQ Select	Erroor Mode	Parity Mode		Parity Type	Bits per Character	

Bit 7	***RECEIVER REQUEST-TO-SEND***
0	Pinne OP0 följer register OPR bit 0
1	Pinne OP0 har funktion RTS
Bit 6	***RECEIVER INTERRUPT SELECT***
0	Avbrott vid mottaget ett tecken (RxRDY)
1	Avbrott vid full mottagarbuffert (FFULL)
Bit 5	***ERROR MODE SELECT***
0	Signallera för fel efter varje mottaget tecken
1	Signallera för fel efter varje mottaget block
Bit 4 3	***PARITY MODE***
0 0	Addera udda eller jämn paritet
0 1	Addera hög eller låg paritetsbit
1 0	Ingen paritet
1 1	Multidrop Mode
Bit 2	***PARITY TYPE***
0	(Bit 4 3 = 0 0) Jämn paritet (Bit 4 3 = 0 1) Låg paritetsbit (Bit 4 3 = 1 1) Data; Multidrop Mode
1	(Bit 4 3 = 0 0) Udda paritet (Bit 4 3 = 0 1) Hög paritetsbit (Bit 4 3 = 1 1)Adress; Multidrop Mode
Bit 1 0	***BITS-PER-CHARACTER***
0 0	5 databitar per tecken
0 1	6 databitar per tecken
1 0	7 databitar per tecken
1 1	8 databitar per tecken

Observera att detta register är placerat på samma adress som Mode Register 2A (MR2A). Registren skiljs åt med en Mode Register Pointer (MR Pointer) i Command Register A (CRA). Efter RESET pekar MR Pointer på MR1A och när MR1A läses eller skrivs kommer pekaren MR Pointer att peka på MR2A. För att på nytt peka ut MR1A ges ett nytt styrord till Command Register A (CRA)

Se ovan: MR1A bit 4 och 3

MR2A

Read/Write

Adress offset $0

Mode Register A (MR2A)

Detta registret tillsammans med MR1A bestämmer kretsens arbetsmod och formatet på den seriella överföringen.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Channel Mode		Tx RTS Control	CTS Enable Transmitter	Stop Bit Length			

Observera att detta register är placerat på samma adress som Mode Register 1A (MR1A). Registren skiljs åt med en Mode Register Pointer (MR Pointer) i Command Register A (CRA). Efter RESET pekar MR Pointer på MR1A och när MR1A läses eller skrivs kommer pekaren MR Pointer att peka på MR2A. När MR2A skrivs eller läses ändras inte MR Pointer.

Bit 7 6	*CHANNEL MODE SELECT*
0 0	Normal mod. Sändare/mottagare arbetar oberoende av varandra
0 1	Automatisk eko-mod. Sändaren skickar direkt mottaget data
1 0	Diagnostik mod. Local loop-back
1 1	Diagnostik mod. Remote loop-back
Bit 5	***TRANSMITTER REQUEST TO SEND CONTROL***
0	RTS påverkas ej
1	RTS (OPR bit 0) nollställs automatisk när sändarregistret, eller sädnar bufferten är tömd
Bit 4	***CLEAR TO SEND CONTROL***
0	CTS påverkar ej sändaren
1	Sändning av nytt teccen fördröjs till CTS (IP0) aktiveras (går låg)

Bitarna bestämmer hur lång stoppbiten skall vara. Längden ges som ett binärtal och är beroende av antal databitar

Bit 3 2 1 0	*STOP BIT LENGTH*
0 0 0 0	ger 0.563 vid 6-8 databitar; 1.063 vid 5 databitar
0 1 1 1	ger 1.000 vid 6-8 databitar; 1.500 vid 5 databitar
0 0 0 0	ger 2.000 vid 6-8 databitar; 2.000 vid 5 databitar

SRA

Read

Adress offset $1

Status Register (SRA)

Registret innehåller statusbitar som indikerar felaktig seriell överföring och om seriell överföring är utförd till/från dataregistren.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Received Break	Framing Error	Parity Error	Overrun Error	TxEMT	TxRDY	RxFIFO FULL	RxRDY

Bit 7	*RECEIVED BREAK*
0	Ej NULL-tecken
1	NULL-tecken

Indikerar om ett NULL-tecken utan stopbit har mottagits. Biten är endast giltig när RxRDY (SRA bit 0) är ettställd.

Bit 6	*FRAMING ERROR*
0	Ej FRAMING ERROR
1	FRAMING ERROR
Bit 5	***PARITY ERROR***
0	Ej paritetsfel
1	Paritetsfel
Bit 4	***OVERRUN ERROR***
0	Ej tappade tecken i mottagar delen
1	Tappade tecken i mottagardelen

Biten är endast giltig när RxRDY (SRA bit 0) är ettställd.

Biten är endast giltig när RxRDY (SRA bit 0) är ettställd

Bit 3	*TRANSMITTER EMPTY*
0	Sändardelen (buffer och skiftregister) innehåller data att sändas
1	Sändardelen (buffer och skiftregister)är tom och kan laddas med ny data
Bit 2	***TRANSMITTER READY***
0	Sändarregistret innehåller data
1	Sändarregistret är tomt

Biten nollställs när processorn skriver data till sändarregistret

Biten nollställs när processorn skriver data till

Bit 1	*RECEIVER FIFO FULL*
0	Mottagardelen är inte full
1	Mottagardelen är full
Bit 0	***RECEIVER READY***
0	Mottagardelen är tömd på data
1	Data finns att läsa i mottagardelen

Biten nollställs när processorn läst data från

Clock Select Register A (CSRA)

CSRA

Write

Adress offset $1

Registret innehåller bitar för att välja öveföringshastighet (BAUD-RATE) på den seriella kommunikationen.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Receiver-Clock Select				Transmitter-Clock Select			

Bit 7 6 5 4	ACR bit 7 = 0	ACR bit 7 = 1
0 0 0 0	50 BAUD	75 BAUD
0 0 0 1	110 BAUD	110 BAUD
0 0 1 0	134.5 BAUD	134.5 BAUD
0 0 1 1	200 BAUD	150 BAUD
0 1 0 0	300 BAUD	300 BAUD
0 1 0 1	600 BAUD	600 BAUD
0 1 1 0	1200 BAUD	1200 BAUD
0 1 1 1	1050 BAUD	2000 BAUD
1 0 0 0	2400 BAUD	2400 BAUD
1 0 0 1	4800 BAUD	4800 BAUD
1 0 1 0	7200 BAUD	1800 BAUD
1 0 1 1	9600 BAUD	9600 BAUD
1 1 0 0	38.4k BAUD	19.2k BAUD
1 1 0 1	TIMER	TIMER
1 1 1 0	IP4-16X	IP3-16X
1 1 1 1	IP4-1X	IP3-1X

Bit 3 2 1 0	ACR bit 7 = 0	ACR bit 7 = 1
0 0 0 0	50 BAUD	75 BAUD
0 0 0 1	110 BAUD	110 BAUD
0 0 1 0	134.5 BAUD	134.5 BAUD
0 0 1 1	200 BAUD	150 BAUD
0 1 0 0	300 BAUD	300 BAUD
0 1 0 1	600 BAUD	600 BAUD
0 1 1 0	1200 BAUD	1200 BAUD
0 1 1 1	1050 BAUD	2000 BAUD
1 0 0 0	2400 BAUD	2400 BAUD
1 0 0 1	4800 BAUD	4800 BAUD
1 0 1 0	7200 BAUD	1800 BAUD
1 0 1 1	9600 BAUD	9600 BAUD
1 1 0 0	38.4k BAUD	19.2k BAUD
1 1 0 1	TIMER	TIMER
1 1 1 0	IP2-16X	IP5-16X
1 1 1 1	IP2-1X	IP5-1X

Beroende på bit 7's värde i Auixilary Control Register (ACR bit7) så fås en av två uppsättningar av BAUDRATE.

Command Register A (CRA)

CRA

Write

Adress offset $2

I detta register ger vi olika kommandon till DUART:n. Exempel på kommandon är RESET, starta eller stanna sändardelen (eller mottagardelen)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NOT USED	Miscellaneous Commands			Transmitter Commands		Receiver Commands	

Bit 6 5 4	*MISCELLANEOUS COMMANDS*
0 0 0	Inget kommando
0 0 1	Sätt MR Pointer till Mode Register 1 (MR1)
0 1 0	RESET mottagaren
0 1 1	RESET sändaren
1 0 0	RESET ERROR STATUS
1 0 1	RESET BREAK avbrott
1 1 0	Start BREAK
1 1 1	Stop BREAK
Bit 3 2	***TRANSMITTER COMMAND***
0 0	Ingen ändring av nuvarande mod
0 1	Aktivera sändaren
1 0	Stanna sändaren
1 1	Ej tillåtet
Bit 1 0	***RECEIVER COMMAND***
0 0	Ingen ändring av nuvarande mod
0 1	Aktivera mottagaren
1 0	Stanna mottagaren
1 1	Ej tillåtet

Receiver Buffer A (RBA)

RBA

Read

Adress offset $3

Registret innehåller seriellt mottaget data

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

Transmitter Buffer A (TBA)

TBA

Write

Adress offset $3

Registret innehåller seriell data som skall sändas

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

IPCR
Read
Adress offset $4

Input Port Change Register (IPCR)

Via detta register kan den parallella inporten läsas. Vidare indikerar registret vilka ingångar som har ändrats efter den senaste läsningen av porten.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Delta Detect IP3	Delta Detect IP2	Delta Detect IP1	Delta Detect IP0	Level IP3	Level IP2	Level IP1	Level IP0

Bitarna nollställs när registret läses. Aktiv flank (hög-till-låg eller låg-till-hög) bestämms i Auxiliary Control Register (ACR) bit 3-0

Bit 7 6 5 4	***CHANGE OF STATE IP3, IP2, IP1, IP0***
x x x x	Dessa statusbitar indikerar en aktiv flank på respektive inputpinne IP3-IP0.
Bit 3 2 1 0	***CURRENT STATE IP3, IP2, IP1, IP0***
x x x x	Dessa bitar anger aktuell nivå på respektive inpinne. Observera att data på dessa inpinnar inte är latchade.

ACR
Write
Adress offset $4

Auxiliary Control Register (ACR)

I detta register ges mer styrord till DUART:n när denna skall användas i mer speciella tillämpningar. Vidare anges här hur räknaren skall användas.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
BRG SET Select	Counter / Timer Mode and source			Delta IP3 IRQ	Delta IP2 IRQ	Delta IP1 IRQ	Delta IP0 IRQ

Bit 7	***BAUD-RATE GENERATOR SET SELECT***
0	Set noll väljs
1	Set ett väljs

Bit 6 5 4	***COUNTER/TIMER MODE, CLOCK SOURCE***	***MODE***
0 0 0	Extern källa (IP2)	Counter
0 0 1	TxCA-1X; klocka för kanal A's sändare	Counter
0 1 0	TxCB-1X; klocka för kanal B's sändare	Counter
0 1 1	Kristall eller extern källa (X1 / Clk)	Counter
1 0 0	Extern källa (IP2)	Timer
1 0 1	Extern källa, delas med 16	Timer
1 1 0	Kristall eller extern källa (X1 / Clk)	Timer
1 1 1	Kristall eller extern källa (X1 / Clk), delas med 16	Timer

Bit 3 2 1 0	*CHANGE OF STATE INTERRUPT ENABLE IP3..0*
x x x x	En ettställd bit innebär att ett avbrott kan genereras (bit 7 i Interrupt Status Register, ISR, ettställs) under följande förutsättningar att bit 7 i Interrupt Mask Register (IMR) är ettställd att en flank uppträder på tillhörande inpinne Ipx. När en flank uppträder på en inpinne IPx vars tillhörande bit i ACR är nollställd sker ingen ändring av ISR bit 7.

Interrupt Status Register (ISR)

ISR

Read

Adress offset $5

I detta statusregister kan vi läsa orsaken till eventuella avbrott. En ettställd bit i detta register förorsakar att ett avbrott genereras om och endast om tillhörande bit i Interrupt Mask Register (IMR) är ettställd.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Input Port Change	Delta Break B	RxRDYB/ FFULLB	TxRDYB	Counter / Timer Ready	Delta Break A	RxRDYA/ FFULLA	TxRDYA

Bit 7	*INPUT PORT CHANGE*
x	Indikerar om mågon av de i ACR[3..0] aktiverade inpinnarna IP3, IP2, IP1, IP0 har detekterad en flank.
Bit 6	***DELTA BREAK CHANNEL B***
x	Indikerar att kanal B har mottagit en BREAK.
Bit 5	***RECEIVER READY OR FIFO FULL, CHANNEL B***
x	Indikerar att det finns tecken att läsa i mottagaren eller att mottagaren är full och måste läsas. Skillnaden bestämms i MR1B bit 6.
Bit 4	***TRANSMITTER READY, CHANNEL B***
x	Biten är en kopia av bit 2 i SRB
Bit 3	***COUNTER / TIMER READY***
x	I Counter mode innebär en ettställd bit att räknaren nått sitt slutvärde.
Bit 2	***DELTA BREAK CHANNEL A***
x	Indikerar att kanal A har mottagit en BREAK.
Bit 1	***RECEIVER READY OR FIFO FULL, CHANNEL A***
x	Indikerar att det finns tecken att läsa i mottagaren eller att mottagaren är full och måste läsas. Skillnaden bestämms i MR1A bit 6.
Bit 0	***TRANSMITTER READY, CHANNEL A***
x	Biten är en kopia av bit 2 i SRA

Biten nollställs när processorn läser Input Port Change Register

Biten nollställs när processorn utför ett RESET BREAK kommando

Biten nollställs när processorn läser mottaget data

Biten nollställs med kommandot ”Stanna räknaren”

Biten nollställs när processorn utför ett RESET BREAK kommando

Biten nollställs när processorn läser mottaget data

IMR

Write

Adress offset $5

Interrupt Mask Register (IMR)

Registret indikerar vilka händelser som *tillåts* generera avbrott. En ettställd bit indikerar att avbrott tillåts om en viss händelse (i ISR) har inträffat.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Input Port Change IRQ	Delta Break IRQ	RxRDYB/ FFULLB IRQ	TxRDYB IRQ	Cntr/Timer Ready IRQ	Delta Break A IRQ	RxRDYA/ FFULLA IRQ	TxRDYA IRQ

CUR/CLR

Read

CUR Adress offset $6

CLR Adress offset $7

Current Counter (CUR/CLR)

Dessa register innehåller mest signifikanta respektive minst signifikanta byten av den 16-bitars räknaren. Registret kan läsas enbart när *counter mod* är vald och endast när räknaren är stannad.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
C/T(15)	C/T(14)	C/T(13)	C/T(12)	C/T(11)	C/T(10)	C/T(9)	C/T(8)
C/T(7)	C/T(6)	C/T(5)	C/T(4)	C/T(3)	C/T(2)	C/T(1)	C/T(0)

CTUR/CTLR

Write

CTUR Adress offset $6

CTLR Adress offset $7

Counter/Timer Register (CTUR/CTLR)

Detta register, tillsammans med registret nedan, (CTLR) innehåller mest signifikanta respektive minst signifikanta byten av den 16-birars räknarens initialvärde. Observera att det minsta räknarvärde som kan laddas i räknaren är $0002.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
C/T(15)	C/T(14)	C/T(13)	C/T(12)	C/T(11)	C/T(10)	C/T(9)	C/T(8)
C/T(7)	C/T(6)	C/T(5)	C/T(4)	C/T(3)	C/T(2)	C/T(1)	C/T(0)

MR1B

Read/Write

Adress offset $8

Mode Register B (MR1B)

Registret innehåller styrbitar för det seriella förmatet.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Rx RTS Control	Rx IRQ Select	Erroor Mode	Parity Mode		Parity Type	Bits pe Character	

Bitarnas funktion är identiska med MR1A bortsett i från att detta registret styr kanal B. Vi hänvisar därför till register MR1A för bitarnas funktion. Observera att detta register är placerat på samma adress som

Mode Register 2B (MR2B). Registren skiljs åt med en Mode Register Pointer (MR Pointer) i Command Register B (CRB). Efter RESET pekar MR Pointer på MR1B och när MR1B läses eller skrivs kommer pekaren MR Pointer att peka på MR2B. För att på nytt peka ut MR1B ges ett nytt styrord till Command Register B (CRB)

Mode Register B (MR2B)

MR2B

Read/Write

Adress offset $8

Bitarnas funktion är identiska med MR2A bortsett från att detta register styr kanal B. Vi hänvisar därför till register MR2A för bitarnas funktion. Registret innehåller styrbitar för den seriella kommunikationen. Observera att detta register är placerad på samma adress som Mode Register 1B (MR1B). Registren skiljs åt med en Mode Register Pointer (MR Pointer) i Command Register B (CRB). Efter RESET pekar MR Pointer på MR1B och när MR1B läses eller skrivs kommer pekaren MR Pointer att peka på MR2B. När MR2B skrivs eller läses ändras inte MR Pointer.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Channel Mode		Tx RTS Control	CTS Enable Transmitter	Stop Bit Length			

Status Register B (SRB)

SRB

Read

Adress offset $9

Registret innehåller statusbitar som indikerar felaktig seriell överföring och om seriell överföring till/från dataregistren har utförts.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Received Break	Framing Error	Parity Error	Overrun Error	TxEMT	TxRDY	RxFIFO FULL	RxRDY

Bitarnas funktion är indentisk med SRA bortsett från att detta register styr kanal B. Vi hänvisar därför till register SRA för funktion.

Clock Select Register B (CSRB)

(CSRB)

Write

Adress offset $9

Bitarnas funktion är indentisk med CSRA bortsett från att detta register styr kanal B. Vi hänvisar därför till register CSRA för att studera bitarnas funktion.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Receiver-Clock Select				Transmitter-Clock Select			

CRB
Write
Adress offset $A

Command Register B (CRB)

Bitarnas funktion är indentisk med CRA bortsett från att detta register styr kanal B. Vi hänvisar därför till register CRA bitarnas funktion.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NOT USED	Miscellaneous Commands			Transmitter Commands		Receiver Commands	

RBB
Read
Adress offset $B

Receiver Buffer B (RBB)

Registret innehåller mottaget seriell data för kanal B

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

TBB
Write
Adress offset

Transmitter Buffer B (TBB)

Registret innehåller data som skall skickas på kanal B

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

IVR
Read/Write
Adress offset $C

Interrupt Vector Register (IVR)

Registret innehåller avbrottsvektor för vektoravbrott. Vid RESET initieras detta register till $0F som enligt MC68000's exception-vektorer innebär oinitierad avbrottsvektor.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interrupt Vector Number							

IP
Read
Adress offset $D

Input Port (IP)

Bitarna indikerar vad som för tillfället finns på inportens pinnar IP5..IP0. Observear att bit 7 alltid läses som etta. Bit 6 läses som noll eller ett beroende på IACK-signalen

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

Output Port Configuration Reg (OPCR)

OPCR

Write

Adress offset $D

Registret definierar funktionen hos utpinnarna OP7..OP2 hurvida dessa skall följa innehållet i *output port register* (OPR) eller ingå som handskakningssignaler i den seriella kommunikationen.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OP7	OP6	OP5	OP4	OP3		OP2	

För bitarna OPCR7..OPCR4 gäller att när en bit är nollställd kommer tillhörande OPx-pinne att följa innhållet i Output Port Register (OPR). När en bit är ettställd får tillhörande OPx pinne följande funktion:

Bit	**Funktion**
7	OP7 följer statusbiten Tx RDYB
6	OP6 följer statusbiten Tx RDYA
5	OP5 följer statusbiten Tx RDYB/FFULLB
4	OP4 följer statusbiten Tx RDYAB/FFULLA
Bit 3 2	**Styr funktion hos OP3**
0 0	OP3 följer Output Port Register
0 1	OP3 är en utsignal från Counter/TIMER. Vid TIMER mod fås en fyrkantvåg med den programmerade frekvensen på OP3 Vid Counter mod är OP3 hög till räknarens värde nås, vilket medför att OP3 går låg. OP3 återställs hög när räknaren stannas.
1 0	OP3 är en kopia av sändar klockan som skiftar ut data på seriekanal B
1 1	OP3 är en kopia av mottagarklockan som skiftar in data från seriekanal B
Bit 1 0	**Styr funktion hos OP2**
0 0	OP2 följer Output Port Register
0 1	OP2 är en kopia av sändar klockan som skiftar ut data på seriekanal A. OP2 kan vara 16X eller 1X av sändarklockan beroende på CSRA[3..0]
1 0	OP2 är en kopia av sändar klockan som skiftar ut data på seriekanal A
1 1	OP3 är en kopia av mottagarklockan som skiftar in data från seriekanal A

Start Counter Command (STAC)

STAC

Read

Adress offset $E

Detta är inget register i traditionell mening utan en läsning på denna adress får *counter /TIMER* att starta

STOC
Read
Adress offset

Stop Counter Command (STOC)

Detta är heller inget register i traditionell mening utan en läsning på denna adress får *counter /TIMER* att stanna.

OPR
Write
Adress offset $E

Output Port Register, bit set (OPR)

Detta register, är uppdelat på två adresser. För att ettställa bitar i OPR skrivs till OPR BIT SET, och för att nollställa bitar skrivs till OPR BIT RESET (se nedan). Registrets innehåll fås som det inverterade värdet hos de bitar som är initierade som OP-pinnar.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OP7	OP6	OP5	OP4	OP3	OP2	OP1	OP0

OPR
Write
Adress offset $F

Output Port Register, bit reset (OPR)

När en nolla skrivs till detta register kommer tillhörande bit att behålla sitt gamla värde oberoende om detta var en etta eller en nolla. För att nollställa biten skrivs en etta.

8.5.5 Vektoravbrott med DUART.

Vi ska nu visa hur kretsen används med vektoravbrott. Vi förutsätter att du själv studerar registerbeskrivningen i de fall det är nödvändigt.

Vi använder kanal B's port för att generera avbrott då ett tecken tas emot.

avbrott skall genereras när seriell data mottages av kretsen.

avbrottsvektor = 64

Som seriellt format väljer vi:

9600 BAUD

ingen paritet

1 stopbit

Då det i vissa tilllämpningar kan vara så att DUART:n redan används för en viss överföring och det handlar om att initiera den för ett *nytt* arbetssätt visar vi här hur detta utförs.

Studera initieringsrutinen i exempel 8.26 (läs kommentarerna). Det är uppdelat i olika block. Först initieras vektornummer, sedan ges RESET till kanal B för att slutligen initiera avbrott och kommunikations-parametrar på nytt.

EXEMPEL

Initieringsrutin för duart med avbrott

```
DIRQNO   EQU  64   Vector Number for DUART IRQ

init_duart_irq:
* Sätt duart interrupt vector
   MOVE.B  #DIRQNO,(duart_IVR).L
   MOVE.L  #duart_irq,(DIRQNO*4).L

* RESET Kanal B
   MOVE.B  #$30,(duart_CRB).L    reset transmitter
   MOVE.B  #$20,(duart_CRB).L    reset receiver
   MOVE.B  #$10,(duart_CRB).L    reset MR pekare

* Kommunikationsparametrar
   MOVE.B  #$BB,(duart_CSRB).L   9600 baud
   MOVE.B  #$13,(duart_MR1B).L   8,N,1
   MOVE.B  #$7,(duart_MR2B).L    Normal mode
   MOVE.B  #5,(duart_CRB).L      enable Tx/Rx
   MOVE.B  #0,(duart_OPCR).L
   MOVE.B  #$20,(duart_IMR).L    pass irq
   RTS
```

8.26

När vi nu beskriver initieringsrutinen ovan så bör du samtidigt studera registerbeskrivningen tidigare i detta kapitel. Försök förstå innebörden av varje bit som används i initieringen.

Sätt DUART interrupt vektor

Först initierar vi kretsen med vektornummer (DIRQNO) genom att skriva till DUART:ns vektor register (IVR adress offset $C) Sedan placerar vi avbrottssutinens startadress på den första lediga avbrottsvektorn i minnet (på adress 64*4).

RESET Kanal B

Vi måste ge kanal B ett RESET-kommando eftersom vi här inte vet om kretsens kanal B initierats tidigare och använts i något annat sammanhang. För att återställa kanal B ges ett kommando till *command register*, CRB. Detta register kan initieras med ett antal olika kommandon. Studera betydelsen av bitarna 4, 5 och 6 i registret. Observera speciellt det sista kommandot vi ger när vi återställer en "adresspekare" till register MR1. Detta är ett måste och beror på att två register, MR1B och MR2B, har samma adress.

Kommunikations parametrar

När vi har stannat kanal B är det nu dags att initiera den för det kommunikationsformat vi önskar. Vi valde 9600 BAUD och därför

9600 BAUD

initierar vi både sändar- och mottagarklockan till 9600 BAUD. Studera register CSRB.

8 bitar

Ingen paritet

1 stoppbit

Vidare besämmer vi dataformatet i register MR1B och MR2B. Studera först MR1B. Vi har valt "ingen paritet" (bit 4 och 3) och 8 databitar per tecken (bit 1 och 0). Observera när avbrott nu skall användas, att vi här väljer om avbrott skall genereras vid *ett* mottaget tecken eller när FIFO *är full* (bit 6). Om avbrott skall användas bestämms i register IMR.

När vi initierar MR2 bestämmer vi *normal arbetsmod* med bitarna 6 och 7 och *en stoppbit* (med längden 1.000) med bitarna 3-0.

Aktivera kanalen

När kommunikationsparametrarna är givna kan vi nu aktivera kanal B på nytt. Detta görs genom att skriva $5 *till command register B,* CRB.

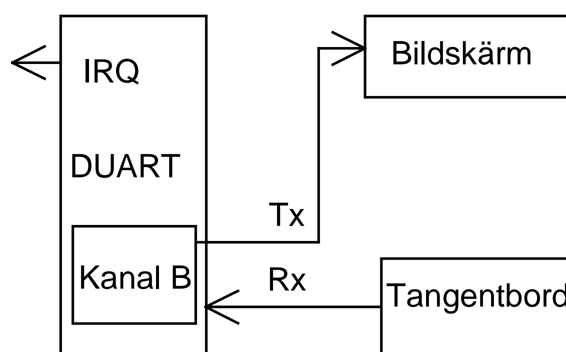
Sedan nollställs Output port Configuration Register i fall dessa pinnar tidigare ingick i handskakningen för den seriella kommunikationen.

Tillåt avbrott

Slutligen i initieringen så anger vi *vilken händelse* som ska tillåtas generera avbrott mot processorn. Vi väljer här att initiera *interrupt mask register* med $20. Detta innebär att vi önskar ett avbrott när RxRDYB eller FFULLB ettställs i kanalens statusregister.

Vi valde tidigare i initieringen att RxRDYB (och inte FFULLB) skulle generera avbrott *om* avbrott skulle användas. Vi har med detta slutfört vår initiering av DUART.

Som avslutning på detta kapitel visar vi nu exempel 8.27 där DUART används för att kommunicera mellan ett mikrodatorsystem, en bildskärm och ett tangentbord (på samma sätt som en terminal är ansluten till ett datorsystem). Se figur 8.39.



Uppgiften består i att läsa tecken från tangentbordet, översätta små bokstäver till stora och skicka tecknen till bildskärmen. Eftersom vi inte vet *när* någon aktiverar tangetbordet är det lämpligt att använda avbrott för mottagna tecken. Vi använder *inte* avbrott på sändarsidan.

FIGUR 8.39. TERMINALUPPKOPPLING MED DUART

Programmet är enkelt, det mottagna tecknet i en global variabel (`buffert`) av avbrottsrutinen. Tecknet NULL ($00) får här representera "inget tecken". Vi förutsätter att DUART initierats enligt föregående exempel.

EXEMPEL

Kommunikation mellan en mikrodator, tangentbord och bildskärm med hjälp av en duart.

```
* Definitioner
    USE   md68.def  Använd fördefinierade adresser
DIRQNO:  EQU   64  första tillgängliga IRQ vektor

   ORG    $4000
****************
* Huvudprogrammets initieringar
main:
   BSR    init_duart_irq   se föregående exempel
   CLR.B  (buffer).L      "inget tecken"
   MOVE   #$2000,sr       tillåt avbrott

* huvudprogrammet
main1:
   ---                    gör något nyttigt
   BSR    echo            ta hand om ev. tecken
   ---                    gör något annat nyttigt
   BRA    main1

*****************
* Subrutin "echo" konverterar gemena till versaler
* och skickar dessa
echo:
   MOVE.B (buffer).L,D0   läs buffert
   BEQ     exit           om inget tecken, avsluta
   CMPI.B #$61,D0         gemena ska konverteras
   BLT    send
   CMPI.B #$7A,D0
   BGE    send
   ANDI.B #%11011111,D0   konvertera och send
send:
   BTST.B (duart_SRB).L   transmitter ledig?
   BEQ    send            om inte: vänta
   MOVE.B D0,(duart_TBB).L     skicka tecken
   CLR.B  (buffer).L      "inget tecken"
exit:
   RTS

****************
* Avbrottsrutin
* Hanterar avbrott från DUART kanal B (receiver)
duart_irq:
   MOVE.B  (duart_RBB).L,(buffer).L   läs och spara
   RTE

****************
buffer  DS.B     1
```

8.27

Med detta har vi beskrivit hur DUART MC68681 kan användas för seriell in och utmatning och hur kretsen används i avbrotts sammanhang.

8.6 Sammanfattning

Detta kapitel har behandlat två periferikretsar, DUART MC68681 och PI/T MC68230 från MOTOROLA. Med dessa kretsar kan vi exempelvis implementera en realtidsklocka, en pulsgenerator, ett *Centronicsinterface* (anslutning mot parallell skrivare), generell parallell I/O, seriekommunikation med ett antal olika enheter, mm.

Vi har alltså sett hur vi kunnat använda *en generell* krets för att åstadkomma olika funktioner. Det finns ett antal olika krets leverantörer som producerar DUART-kretsar och PI/T-kretsar med samma eller liknande funktioner. Efter att ha studerat dessa kretsar är det förhållandevis enkelt att använda liknande kretsar, ty alla har motsvarande funktioner som måste oftast måste initieras på ett eller annat sätt.

MIKRODATORN MC68 OCH MC68340

I detta kapitel behandlar vi enkortsdatorn MC68. Kapitlet beskriver datorns olika block. Vidare beskrivs microcontrollern MC68340 detaljerat. Vi ger programexempel som visar hur kretsen kan initieras och användas.

9.1 Mikrodatorn MC68

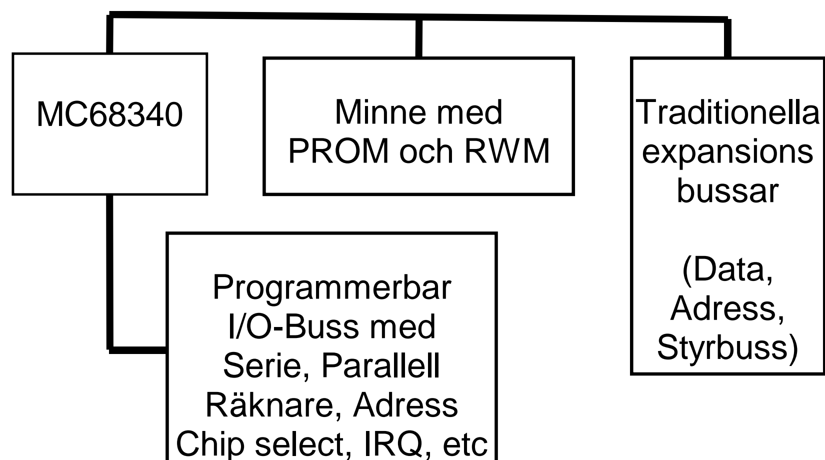
MC68 är en enkortsdator eller "ett controller-kort" (om vi ska använda svengelska). Kontrollerkort är kanske det mest beskrivande eftersom MC68 är uppbyggd kring microcontrollern MOTOROLA MC68340. MC68 kan bestyckas med upp till 1 Mbyte minne på kortet. Vidare finns ett antal periferikretsar (serieportar, räknarkretsar, DMA-kanaler, parallellportar, etc) inbyggda i MC68340.

Utöver de interna periferikretsarna finns också ett stort antal tillbehörskort anpassade för MC68. Här märks bland annat enkla kort som digital I/O så väl som komplexa kort för robotstyrning och datanät.

MC68 är tänkt som en "applikationsmaskin", dvs det ska vara enkelt att programmera datorn för något specifikt styrändamål och därefter, med några enkla handgrepp, förse datorn med ett nytt program, för annat ändamål. MC68 är därför utrustad med FLASH-minne och en speciell programvara som gör att minnet enkelt kan programmeras på nytt utan att minneskapseln flyttas från kortet. MC68 har dessutom getts olika start-upp möjligheter (boot-varianter) via en bygel på kortet. På detta sätt kan olika applikationer startas beroende på hur bygeln har placerats.

9.1.1 Blockdiagram

I princip består MC68 av två block: Microcontroller MC68340 och minne (RWM och FLASH-PROM), se figur 9.1. Utöver detta finns två buss system. Det ena är en programmerbar I/O buss direkt ansluten till MC68340's interna I/O-block. Den andra är en traditionell expansionsbuss bestående av adress, data och styrbuss. Maximalt kan man uppnå en 16 bitars bred databuss och en 32 bitars bred adressbuss.

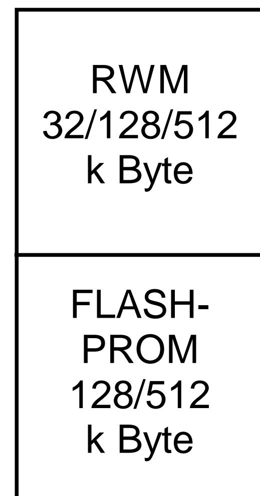


FIGUR 9.1 MC68:S OLIKA BLOCK

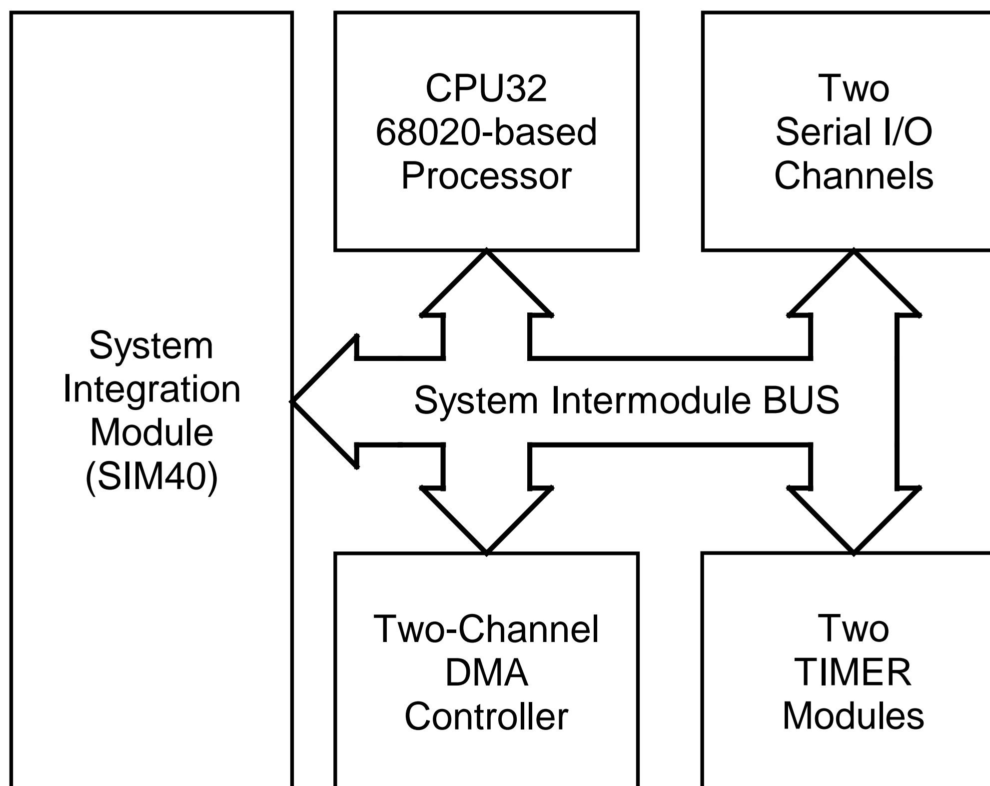
Låt oss börja med en kort sammanfattning av innehållet i de olika blocken. Blocken kommer att beskrivas i detalj längre fram.

Minnesblocket är bestyckat med statiskt RWM (SRAM) och FLASH-PROM, se figur 9.2. Det statiska blocket kan bestyckas med 32 kbyte, 128 kbyte eller 512 kbyte. FLASH-PROM'et kan vara på 128 kbyte eller 512 kbyte.

MC68340 innehåller processor och ett antal periferikretsar. Processorkärnan, som är MC68000 kompatibel, betecknas *CPU32* och påminner en hel del om mikroprocessorn MC68020 (en större släkting till MC68000). Dessutom finns två seriekanaler, en räknare (s.k *periodic timer* för användning som realtidsklocka) och två räknarmoduler (för mer generella räknarapplikationer) två DMA-kanaler, två 8-bitars parallella portar. Vidare finns en internbuss och ett logikblock, SIM40, som används för att initiera MC68340's olika funktioner, se figur 9.3.



FIGUR 9.2 MC68:S MINNESBANK.



MC68000-kompatibel = Program skrivna för MC68000 kan exekveras på CPU32.

DMA = Direct Memory Access

FIGUR 9.3 MC68340:S INTERNA BLOCK.

I/O bussen kan bland annat programmeras för att vara parallell-port eller ytterligare adressbitar i den traditionella adressbussen. Vissa delar av den programmerbara I/O-bussen kan även utnyttjas för "CHIP-SELECT-signaler" för t.ex. minneskretsarna och andra delar av bussen

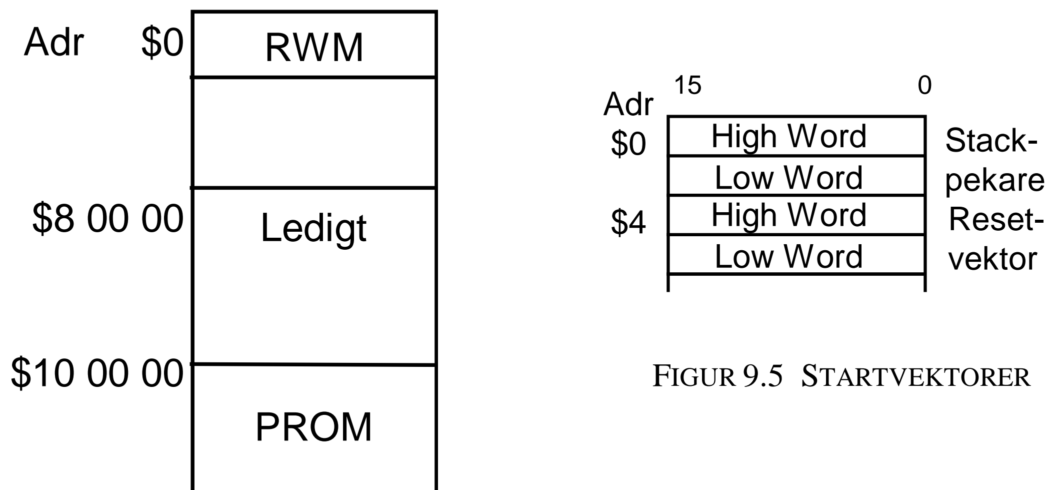
kan också utgöra avbrottssignaler. Alla programmerbara funktioner styrs via SIM40-modulen

Den traditionella expansionsbussen är uppdelad i en enkel och en komplett buss. Den enkla expansionsbussen används för anslutning av laborationskort till MC68 (direkt på kortet via s.k *piggy-back* anslutning) och är då en byte bred och upptar 64 kbyte. Bussen är vald som synkron och innehåller styrsignaler som RESET, Skriv/Läs, Adress Strobe, etc. Den kompletta bussen omfattar resterande signaler från MC68340.

9.2 Minnessystem

I originalutförande är MC68 bestyckad med 32 kbyte RWM och 128 kbytes FLASH-PROM. Maximalt kan ½ Mbyte RWM och ½ MByte PROM placeras på kortet. MC68 använder en 8 bitars bred databuss för att kommunicera med minnet på kortet. 16-bitars brett minne kan anslutas till expansionsbussen.

Skriv och läsminnet är placerat från adress noll och uppåt. Se figur 9.4. Ett ledigt utrymme finns från adress $8 00 00 och framåt. Detta utrymme är avsett för de olika laborationskort som kan anslutas till MC68 via den 16-bitars expansionsbussen. Slutligen är FLASH-PROM placerat med start från adress $10 00 00.



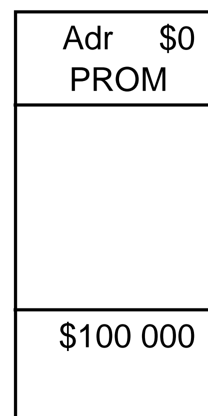
FIGUR 9.5 STARTVEKTORER

FIGUR 9.4 MINNESLAYOUT FÖR MC68

Vid RESET läser MC68340 in två vektorer, stackpekare och programräknare, se figur 9.5. Den läser alltså in en *long* från adress noll och en *long* från adress fyra. Eftersom MC68 är bestyckad med RWM på adress noll så kan det inte lagras några vektorer där, minnesinnehållet försvinner då spänningen slås av. Dessa måste alltså lagras i PROM för att det skall fungera vid spänningstillslag. Utöver

den traditionella adressavkodningslogiken är därför MC68 utrustad med ett litet logikblock som ser till att PROM'et placeras på adress noll vid RESET och på adress $10 00 00 annars. Detta kallas att “mappa ner PROM:et vid RESET”. På så sätt kommer processorn att läsa in stackpekare och programräknare från adress noll och fyra i PROM'et, först därefter flyttas RWM'et till adress 0. Detta sker på följande sätt.

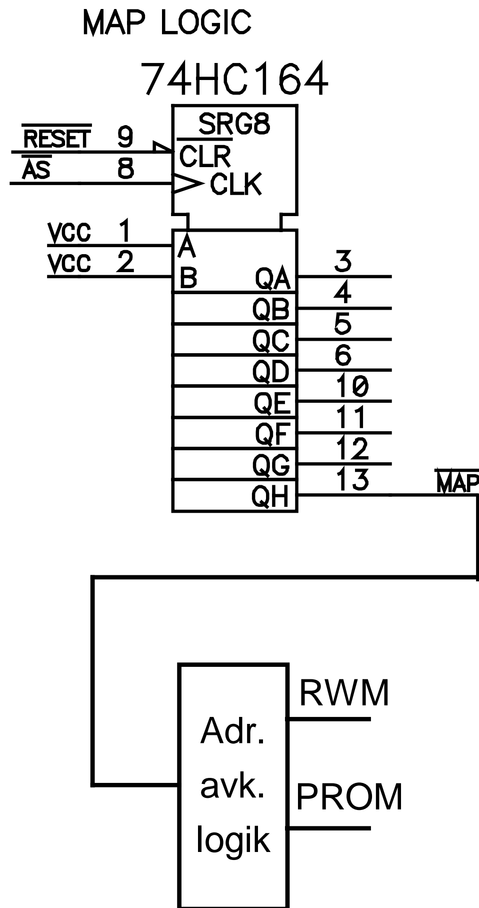
Vid RESET läses totalt 8 bytes (två *long*). Eftersom MC68 använder en 8-bitars bred databuss krävs 8 busscykler för att läsa in vektorerna. Logikblocket på MC68 räknar därför 8 busscykler och aktiverar under denna tid en MAP-signal till adressavkodningslogiken som i sin tur aktiverar PROM:et under denna tid. När sedan MAP-signalen inte är aktiv kommer adressavkodningslogiken att arbeta enligt minneskartan i figur 9.4. Figur 9.6 illustrerar minnesdispositionen under den tid MAP-signalen är aktiv. MAP-logiken visas i figur 9.7 nedan.



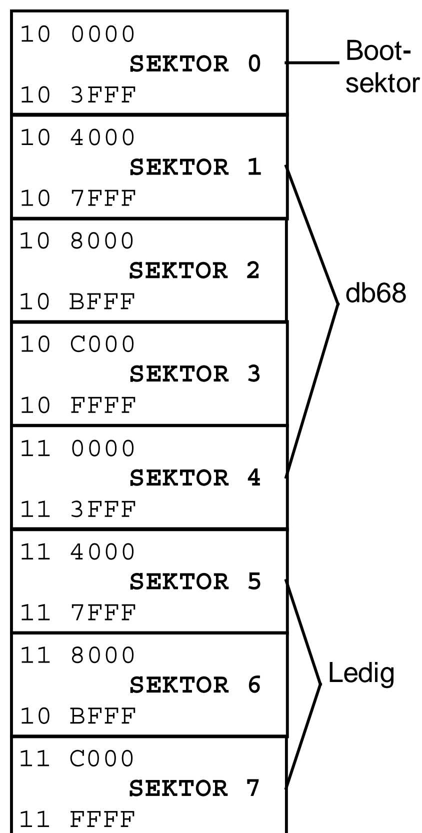
FIGUR 9.6 MINNESLAYOUT VID RESET

Som tidigare beskrivet aktiveras processorns AS-signal (*Adress Strobe*) varje gång en busscykel startar. En räknare (74HC164) klockas därför med AS-signalen. När räknaren har räknat till 8 går MAP-signalen (QH) hög. På så sätt kan adressavkodningslogiken aktivera PROM'et för låga adresser de första 8 busscyklerna, och för adress $10 00 00 annars.

Om MC68 vore utrustad med ett 16-bitars brett minne vid RESET skulle processorn ha läst in 4 word. MAP-signalen skulle då varit kopplat till QD-anslutningen på räknaren som går hög efter fyra klockpulser (Adress Strobar). På så sätt kommer adressavkodningslogiken att aktivera PROM:et för låga adresser de första 4 busscyklerna, och för adress $10 00 00 annars



FIGUR 9.7 MAP-LOGIK MED RÄKNARE.

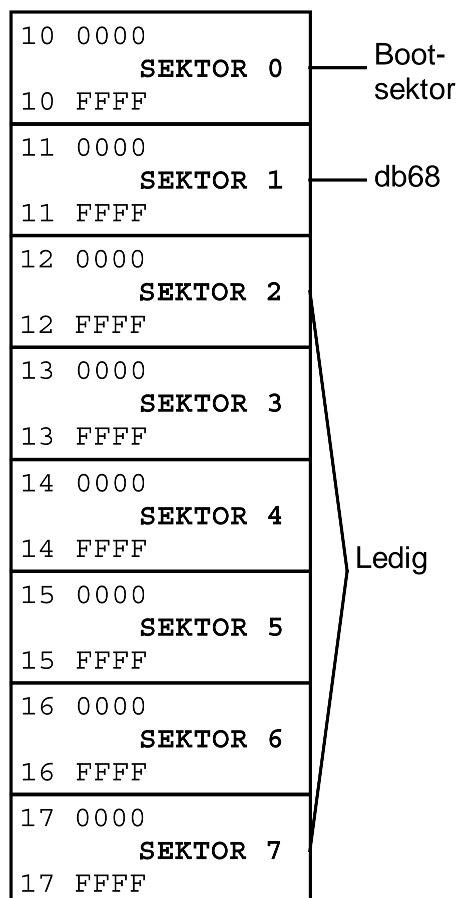


FIGUR 9.8 MINNESKARTA FÖR 128 KBYTES FLASH-PROM.

Vi skall nu studera FLASH-PROM:et på MC68 mer detaljerat. Detta kan skrivas och raderas med befintliga programmerings-rutiner (skrivrutiner) och raderingsrutiner. När man programmerar FLASH-minnet kan man programmera en eller flera bytes, efter eget önskemål. När man däremot raderar så måste man radera ett helt block (eller en *sektor* som det benäms) åt gången.

FLASH-minnet är indelat i åtta sektorer (block). Första sektorn (sektor 0) är reserverad för MC68's uppstartsrutin. Detta kallas *boot-sektor*. Studera figur 9.8. Vid RESET startas boot-rutinen (i boot-sektorn) och läser av en bygel på kortet. Beroende på bygelns läge startas antingen debuggern *db68* i sektor 1, en applikation i första lediga sektor efter debuggern eller en applikation i sektor 7. Man kan också välja att enbart starta boot-rutinen. Den används när man skall underhålla FLASH minnet (radera/programmera sektorer), när man skall bränna in en ny db68 (monitor), etc.

Normalt är MC68 bestyckad med 128 kbyte FLASH. Detta innebär att varje sektor i FLASH'et är 16 kbyte stor. Startadresser för de olika segmenten ges i figur 9.8.



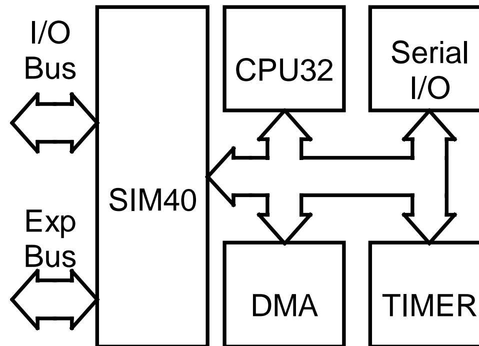
FIGUR 9.9 MINNESKARTA FÖR 512 KBYTES FLASH-PROM.

MC68 kan också förses med 512 kbytes FLASH. Detta är också indelat i 8 sektorer vilket innebär att varje sektor upptar 64 kbyte (se figur 9.9). I vanlig ordning är det första segmentet reserverad för boot-rutinen. Observera att monitorn, *db68*, nu ryms i en sektor. De övriga sex sektorerna kan användas för önskade applikationer. Självfallet kan även monitorn i sektor 1 bytas ut mot egna applikationer, detta gäller för såväl 128- som 512 k's FLASH.

Så långt om minnessystemet för MC68. Önskas mer minne än 1 Mbyte måste detta placeras som expansionskort på MC68.

9.3 MC68's bussar.

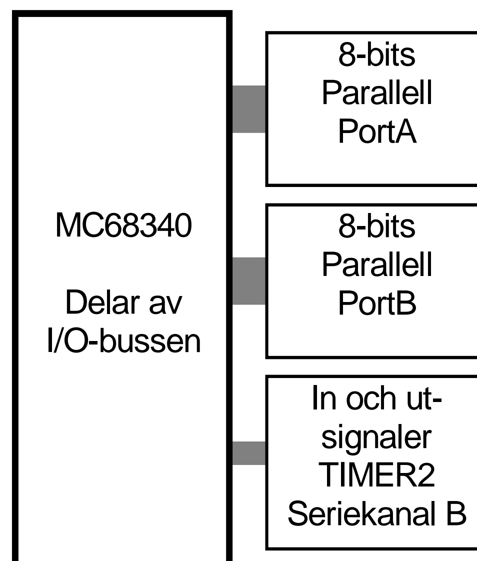
MC68 har två bussystem där den ena är en programmerbar I/O och styrbuss medan den andra är en traditionell adress-, data- och styrbuss. Dessa är åtkomliga via olika kontakter på kortet. Vi beskriver här bussarnas funktion för en normal/default MC68 och hur man kan initiera bussarna enligt egna önskemål när SIM40-modulen i MC68340 beskrivs. Se figur 9.10



FIGUR 9.10 MC68340:S BUSSAR.

9.3.1 I/O Buss

MC68's I/O buss är tillgänglig bland annat via anslutning P5. I normalfallet är detta en anslutning för laborationskortet ML4, bussen innehåller då en 8-bitars parallell utport och en 8-bitars parallell inport (se figur 9.11). Vidare finns in/ut signaler för TIMER2 och seriekanal B. Observera att dessa anslutningar är obuffrade på MC68 (bortsett från serieanslutningen). Övriga anslutningar till de interna I/O-blocken på MC68340 är TIMER1, seriekanal A och DMA.

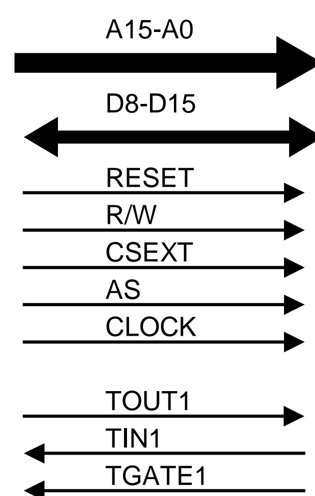


FIGUR 9.11 I/O-BUSS.
ANSLUTNING P5

9.3.2 Expansionsbuss

I normalutförande av MC68 består den traditionella expansionsbussen av 24 adressbitar, 16 databitar och en traditionell styrbuss med klocka, skriv/lässignal, etc. (Internt på MC68 används en 8-bitars databuss och en 21 bitars adressbuss). Expansionsbussen är uppdelad i två delar, en enkel och en kompletterande expansionsbuss. Den enkla är anpassad för MC68:s olika laborationskort.

Anslutningarna för den enkla expansionsbussen utgörs av de fyra 10-poliga anslutningarna, ett i varje hörn på kortet. Bussen som är buffrad har 16-bitars adressbuss och 8 bitars databuss. Observera att databussens övre halva (D8-D15) används internt på MC68 och till den enkla expansionsbussen. Bussen är en synkronbuss och dess styrsignaler visas i figur 9.12.

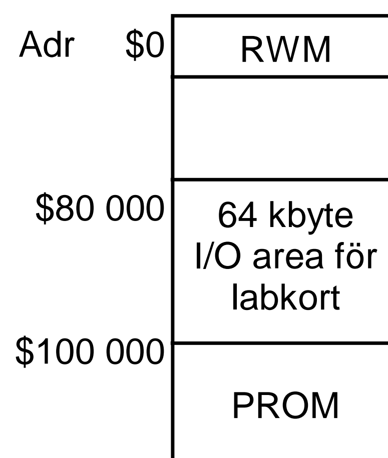


FIGUR 9.12 ENKEL EXPANSIONSBUSS

Signaler som RESET, R/W och AS förutsätts vara bekanta. CLOCK är systemklockan från MC68340. Observera signalen CSEXT (*chip select external*) i den enkla expansionsbussen, det är en aktiveringssignal till adressarean för MC68:s påbyggnadskort. Studera figur 9.13 som visar hela minnesdispositionen för MC68.

Även TIMER1 finns tillgänglig i den enkla expansionsbussen. Signalerna betecknas TIN1, TOUT1 och TGATE1.

Anslutningarna för den kompletterande expansionsbussen är inte inlödda. Bussen innehåller den resterande delen av adressbussen (A16 -A23) och den resterande halvan av databussen (D0-D7). Vidare finns här alla traditionella styrsignaler för en processor i 68000-familjen. Vi går inte i detalj in på vilka signaler detta är utan hänvisar i stället till figur 9.14 nedan, som visar MC68340:s pinplacering.



FIGUR 9.13 MINNESDISPOSITION FÖR MC68

9.4 MC68340

På MC68 används MC68340 i en 144-pinnars ytmonterad kåpa med benämningen MC68340FEXXY. Studera kretsen. FE anger att det är en kåpa som benäms *144-Lead Ceramic Quad Flat Pack*. XX anger max klockfrekvens för denna krets, 16 och 25 MHz används. Sedan anger Y:et i beteckningen ett revisionsnummer.

Det finns också en beteckning som anger vilken programmeringsmask som tillverkaren använde vid produktionen. Slutligen anges var kretsen är tillverkad, vilket år och vilken vecka. Sifferkombinationen 9551 innebär exempelvis att kretsen är tillverkad vecka 51 år 1995. Studera hur gammal just din MC68 är.

Innan vi "öppnar på locket" på MC68340 bör vi studera kretsens pinplacering. Observera att kretsens I/O-moduler upptar ett antal pinnar på kretsen, betrakta figur 9.15 nedan, som visar pinnplaceringen.

Överst till vänster visas styrsignalerna BERR, RESET och HALT. Dessa anses kända. Sedan följer DSACK0 och DSACK1. Dessa är liknande signalen DTACK för MC68000. Här anger de även om det är DTACK för 8-eller 16-bitars bred databuss. Insignalerna BGACK och BR med tillhörande utsignal BG används vid busarbitrering. Dessa har samma funktion som vid MC68000.

Signalerna VCCSYN, XFC, EXTAL och XTAL är för anslutning av kretsens kristall. Klockfrekvensen ut är tillgänglig på pinnen CLKOUT.

Överst till höger hittas processorns Function Code pinnar. Observera att MC38340 har fyra medan MC68000 har tre. Vidare på höger sida hittas traditionella busshandskakningssignaler som AS, DS, R/W, etc. DS är Data Strobe och anger att databussen är giltig vid en skrivcykel. Signalerna SIZ0 och SIZ1 anger hur många bytes som återstår att överföras vid skrivning/läsning.

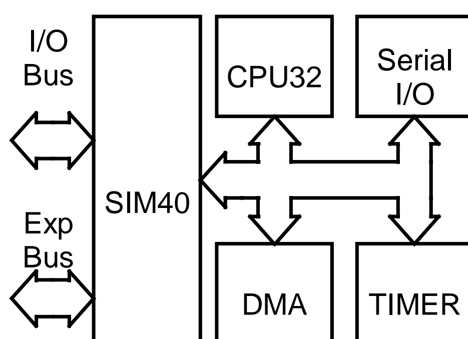
Nedanför signalerna tillhörande EXTBUS i figuren, finns ett antal signaler som används i samband med intern testning av kretsen och vid avlusning (debugging). Dessa signaler behandlas inte vidare.

Sedan återfinns signalerna som ingår i de programmerbara I/O-bussarna vi diskuterade tidigare. Här anges Port-B som inport och Port-A som utport. Vidare förjer in- och utsignaler för serieportarna. Sedan hittas in- och utsignaler fört TIMER och DMA. Vi kommer att behandla delar av dessa signaler längre fram när de olika blocken i MC68340 beskrivs. Längst ner i figuren hittas databuss och adressbuss.

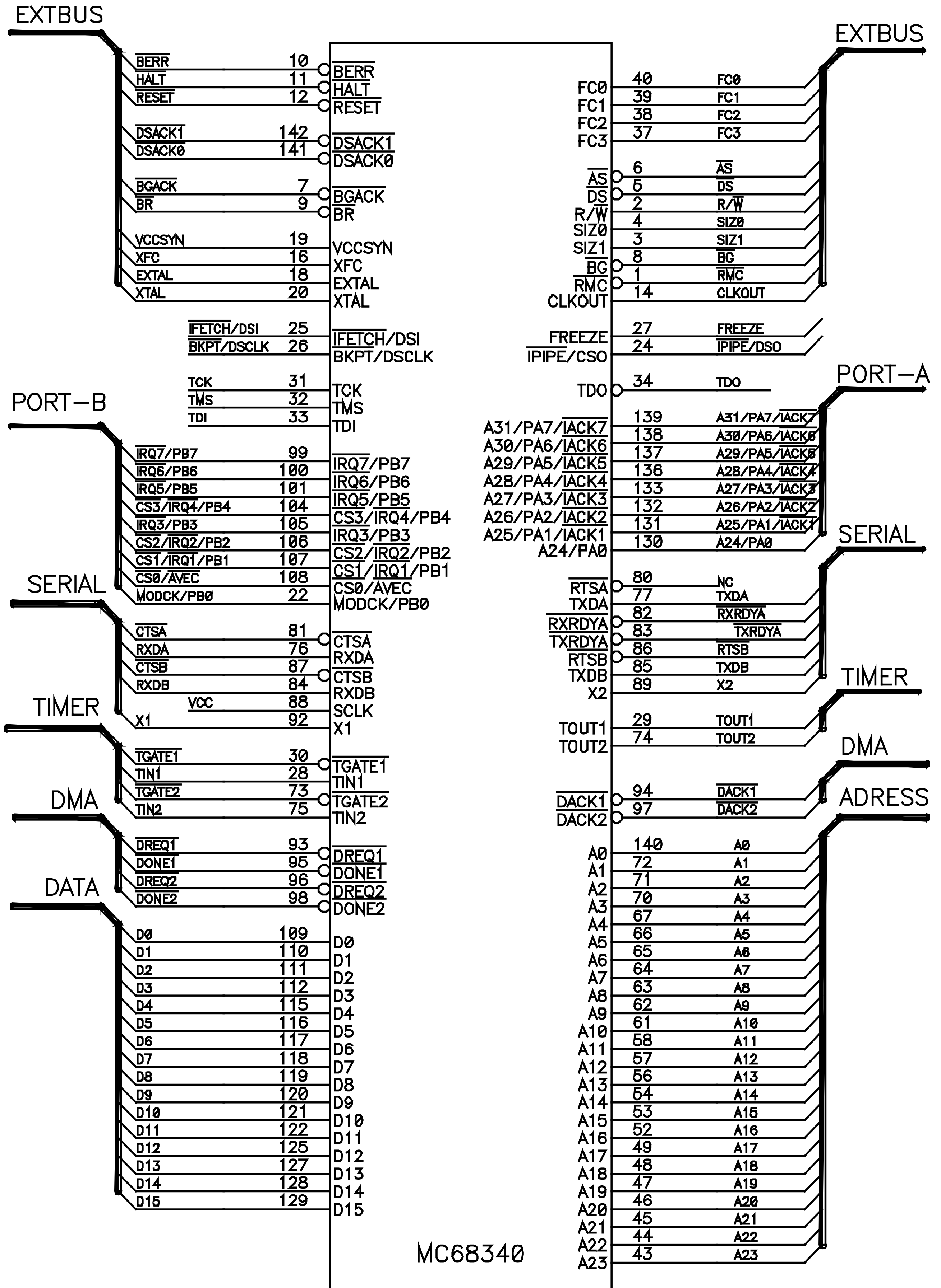
Räknar vi antal pinnar som visas på kommer vi fram till att det finns 113 pinnar. Denna "kåpa" (inkapslingen) hade 144 pinnar. Skillnaden, 31 pinnar, används för +5V och jord. Vissa strömförsörjningsanslutningar används för att driva databussen, medan andra är för adressbussen, några driver vissa styrsignaler och andra är

för olika interna block på MC68340, etc. Denna gruppering är bland annat för att kunna stänga ner delar av- eller hela kretsen för att få ner strömförbrukningen till ett minimum. Vid normaldrift och 5V och 16 MHz är strömförbrukningen knappa 100 mA (500mW). När kretsen är stannad i “Low Power Stop Mode” är strömförbrukningen endast 60 μA.

Vi repeterar det interna blockdiagrammet för MC68340. Kretsen består av processor (CPU32), två seriekanaler, två räknarmoduler och en två-kanalers DMA-enhet. Vidare finns en konfigurerbar I/O-buss och en expansionsbuss. (Figur 9.14) Slutligen finns System Integration Module (SIM40) som kan ses som “spindeln mitt i nätet”. Vi skall därför fortsätta med att studera detta block.



FIGUR 9.14 BLOCKDIAGRAM ÖVER MC68340



FIGUR 9.15 PINPLACERING PÅ MC68340.

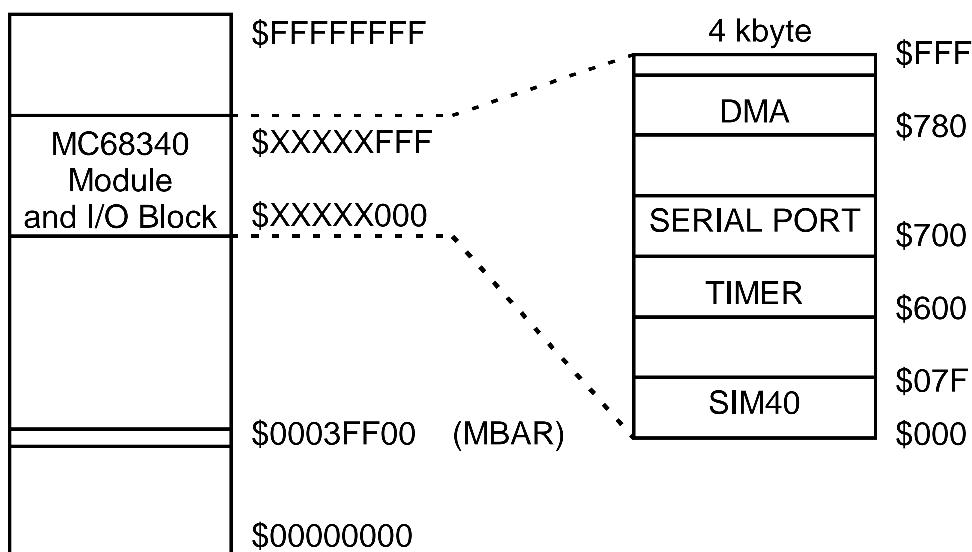
9.4.1 System Integration Module (SIM40)

Gör först en kort tillbakablick och studera de hårdvaruscheman som diskuterades i tidigare kapitel där vi beskrev ett MC68000 minimisystem. Studera vidare kapitlet där vi konstruerade en printerport. Titta även på avsnittet som behandlar adressavkodning. Studeras dessa scheman så hittas en mängd logiska kretsar, register, grindar, buffertar, etc *utöver* processor, minne och periferikretsar. Dessa logikblock kallas *glue logic* (klister-logik) då de krävs för att "limma samman" processor med minne och I/O kretsar.

Det mesta av denna klister-logik finns i System Integration Module (SIM40). Vidare finns här en mängd olika andra logikblock som exempelvis används för att:

- få önskad funktion för I/O-bussen
- få önskad klockfrekvens
- välja adresser för de interna I/O-modulerna
- bestämma minnesmap och Bus-Fels generering
- bestämma Wait-States för minnen
- adressavkodningslogik
- mm, mm, mm.......

SIM modulen innehåller ett antal olika register. Dessa register (tillsammans med registren för de interna DMA-, serie- och räknarmodulerna) kan placeras på valfri plats i minneskartan av programmeraren. Totalt utgör detta minnesområde 4 kbyte.



FIGUR 9.16 ADRESSMAPP FÖR SIM40 OCH I/O BLOCKEN

För att välja basadress för detta block används registret MBAR (*Module Base Address Register*). Registret hittas på adress $0003FF00 i "cpu space" (illustreras strax)

MBAR1
CPU Space only
$0003FF00

MBAR2
CPU Space only
$0003FF02

MBAR1, MBAR2 - Module Base Address Register.

Registret består av 32 bitar. Bitarna BA31-*BA12 (Base Address Bits*) anger de övre 20 bitarna av basadressen för SIM-blocket. Registret kan relokeras till en godtycklig minnesarea, detta anges med XXXXX i figur. Registret kan endast läsas/skrivas då CPU32 är i SM (*supervisor mode*).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BA31	BA30	BA29	BA28	BA27	BA26	BA25	BA24	BA23	BA22	BA21	BA20	BA19	BA18	BA17	BA16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BA15	BA14	BA13	BA12	0	0	AS8	AS7	AS6	AS5	AS4	AS3	AS2	AS1	AS0	V

Den sista biten V-biten, Bit 0, *Valid*, anger om innehållet i MBAR är giltigt eller inte. När V-biten är nollställd kan inte SIM-modulen eller I/O-blocken adresseras. Detta innebär att en skrivning till MBAR är något av det förstasom måste göras efter RESET. Slutligen används Bit AS8-AS0 för att maskera blocket som en extern adress. Följande gäller:

Bit	Mask	Adress Space Function Code
AS8	DMA Space	1xxx
AS7	CPU Space	0111
AS6	Supervisor Program	0110
AS5	Supervisor Data	0101
AS4	Reserved [Motorola]	0100
AS3	Reserved [User]	0011
AS2	User Program	0010
AS1	User Data	0001
AS0	Reserved [Motorola]	0000

Nedan visas en programsekvens som sätter SIM40's basadress i MC68 till $FFFFF000. Observera att CPU Space måste väljas då MBAR skall adresseras. Observera vidare att denna processorn (CPU32) har flera register än MC68000. Dessa register behandlas längre fram.

EXEMPEL

Relokering av SIM40 modul i MC68

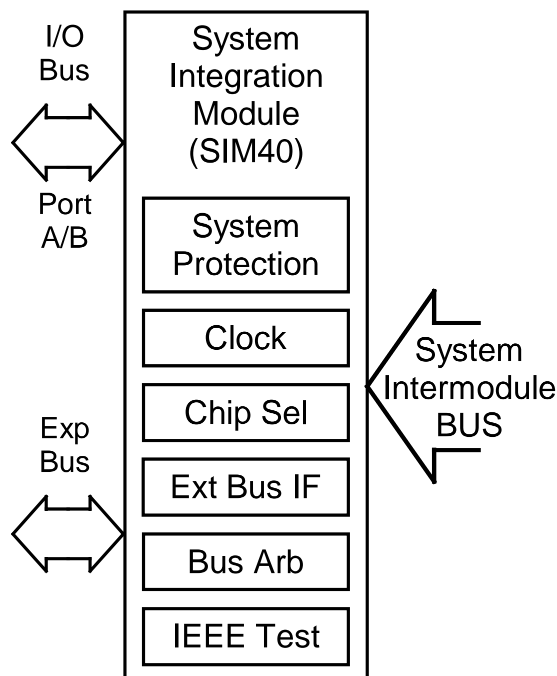
```
* Memory Base Address Register
MBAR  EQU         $3ff00

* Bas adress för SIM40 i MC68
MODBASE     EQU   $fffff000

* initieringssekvens
      MOVE.W      #$2700,SR   Maskera avbrott

* Initiera basadress för SIM40
      MOVEQ       #7,D0
      MOVEC       D0,DFC          Sätt CPU space
      MOVE.L      #MODBASE+1,D0 Sätt basadress
      MOVES.L     D0,(MBAR).L   ...
```

Registret "DFC" beskrivs i avsnittet CPU32

9.1

Efter denna initiering har SIM40 fått basadressen $FFFFF000, seriekretsen har fått basadressen $FFFFF700, räknarkretsen basadress $FFFFF600, etc. (Jämför figur 9.16)

Vi återgår nu till att studera vad som finns internt i SIM40-modulen. Senare ges fler programexempel på hur modulen initieras i MC68. Efter detta kommer vissa register i modulen att beskrivas.



FIGUR 9.17 SIM40:S INTERNA BLOCK

SIM40 innehåller ett antal block. Se figur 9.17 som visar de interna blocken. Överst hittas *System Protection Block*. Detta block innehåller en övervakare för den interna och om önskvärt den externa (utanför MC68340) bussen. Det finns även övervakning för *Double Bus Error* och *Spurious Interrupt*. Vidare finns möjlighet att utnyttja en *Software Watchdog*.

Clock innehåller en räknare som kan användas som realtidsklocka (periodiska avbrott) eller i samband med *Software Watchdog*. Periodtiden kan initieras mellan 122 µs till 15.94 s. Blocket genererar även systemklockan och klockas av en enkel kristall på 32.768 kHz. Systemklockan är programmerbar och default har MC68 en 8 MHz systemklocka.

Nästa block av SIM40 innehåller adressavkodningslogik och om denna initieras så anges här vilka adressområden de olika Chip-Select

utgångarna har. Vidare anges huruvida intern eller extern DSACK-generering (DTACK) skall användas och om intern DSACK utnyttjas, hur många Wait States som skall gälla för just denna Chip Select signalen.

Blocket *External Bus Interface* styr om I/O-Port A och I/O-Port B skall fungera som I/O-portar eller som adressbuss, avbrottssignaler (IRQ och IACK), etc. Vidare anges här om port A och B skall vara ut eller in.

Det näst sista blocket heter *Bus Arbitration* och används i de sammanhang när man har flera enheter som kan styra bussarna. Slutligen finns *IEEE Test-blocket*. Detta är ett användar gränssnitt för att testa MC68340 internt enligt standarden "IEEE 1149.1 Standard Test Access Port and Boundary Scan Architecture". Programmerarens bild av SIM40 ges i figur 9.18.

Adress FFFFF000	S/U	15 8	7 0	
+000	S	MODULE CONFIGURATION REGISTER (MCR)		SYSTEM PROTECTION
+004	S	CLOCK SYNTHESISIZER CONTROL REGISTER (SYNCR)		CLOCK
+006	S	AUTOVECTOR REGISTER (AVR)	RESET STATUS REGISTER (RSR)	SYSTEM PROTECTION
+010	S/U	RESERVED	PORT A DATA (PORTA)	EBI
+012		RESERVED	PORT A DATA DIRECTION (DDRA)	EBI
+014		RESERVED	PORT A PIN ASSIGNMENT 1 (PPRA1)	EBI
+016		RESERVED	PORT A PIN ASSIGNMENT 2 (PPRA2)	EBI
+018		RESERVED	PORT B DATA (PORTB)	EBI
+01A		RESERVED	PORT B DATA (PORTB1)	EBI
+01C		RESERVED	PORT B DATA DIRECTION (DDRB)	EBI
+01E		RESERVED	PORT B PIN ASSIGNMENT (PPARB)	EBI
+020		SW INTERRUPT VECTOR (SWIV)	SYSTEM PROTECTION CONTROL (SYPCR)	SYSTEM PROTECTION
+022	S	PERIODIC INTERRUPT CONTROL REGISTER (PICR)		SYSTEM PROTECTION
+024	S	PERIODIC INTERRUPT TIMING REGISTER (PITR)		SYSTEM PROTECTION
+026		RESERVED	SOFTWARE SERVICE (SWSR)	SYSTEM PROTECTION
+040	S	ADDRESS MASK 1 CS0		CHIP SELECT
+042	S	ADDRESS MASK 2 CS0		CHIP SELECT
+044	S	BASE ADDRESS 1 CS0		CHIP SELECT
+046	S	BASE ADDRESS 2 CS0		CHIP SELECT
+048	S	ADDRESS MASK 1 CS1		CHIP SELECT
+04A	S	ADDRESS MASK 2 CS1		CHIP SELECT
+04C	S	BASE ADDRESS 1 CS1		CHIP SELECT
+04E	S	BASE ADDRESS 2 CS1		CHIP SELECT
+050	S	ADDRESS MASK 1 CS2		CHIP SELECT
+052	S	ADDRESS MASK 2 CS2		CHIP SELECT
+054	S	BASE ADDRESS 1 CS2		CHIP SELECT
+056	S	BASE ADDRESS 2 CS2		CHIP SELECT
+058	S	ADDRESS MASK 1 CS3		CHIP SELECT
+05A	S	ADDRESS MASK 2 CS3		CHIP SELECT
+05C	S	BASE ADDRESS 1 CS3		CHIP SELECT
+05E	S	BASE ADDRESS 2 CS3		CHIP SELECT

FIGUR 9.18 PROGRAMMERARENS BILD AV SIM40.

Följande exempel visar hur initiering av SIM40 utförs *av boot-laddaren* i MC68.

EXEMPEL

```
*
* Utdrag av MC68 Bootstrap Loader

MBAR  EQU   $3ff00      Memory Base Address Register
MODBASE     EQU   $fffff000   Bas adress för SIM40

* SIM40-register, Offset från basadress
MCR   EQU   $00   Module Configuration Register
SYNCR EQU   $04   Clock Synthesizer Register
SYPCR EQU   $21   System Protection Control
CSAM0 EQU   $40
CSBA0 EQU   $44

* Styrord för Chip Select 0
* Adressområde $0 - $7F FFFF
* 3 Wait state, 8 bitars databuss
AM0   EQU   $007FFFFE   Adress mask
BAR0  EQU   $00000001   Basadress $0, Valid

      MOVE.W      #$2700,SR   Maskera avbrott

* Initiera basadress för SIM40
      MOVEQ       #7,D0
      MOVEC       D0,DFC
      MOVE.L      #MODBASE+1,D0
      MOVES.L     D0,(MBAR).L

* Initiera Protection Register
* Aktivera (externa) bussfel efter
* 16 klockcykler
      MOVE.B      #6,(SYPCR+MODBASE).L

* Initiera Clock Synthesizer Register
* Sätt klockfrekvens
      MOVE.W      #$3f00,(SYNCR+MODBASE).L

* Initiera Module Configuration Register
* Ingen Watchdog, Aktivera externa
* bussar vid interna
* operationer, Avbrottsarbitrering
      MOVE.W      #$420F,(MCR+MODBASE).L

* Initiera Chip select
      MOVE.L      #AM0,(CSAM0+MODBASE).L
      MOVE.L      #BAR0,(CSBA0+MODBASE).L
      ---
      ---
```

9.2

Efter denna initiering så initieras seriekretsen. Därefter undersöks byglarna som anger vilken rutin som skall startas. I normalfallet är detta monitorn *db68*. Monitorn är placerad med start i sektor 1 i FLASH:et och det är monitorn som initierar SIM40:ans parallellport. Se följande exempel.

EXEMPEL

```
PPRA1 EQU   MODBASE+$15 Port A Pin Assingment Register
DDRA  EQU   MODBASE+$13 Port A Data Data Direction Register
PPARB EQU   MODBASE+$1F Port B Pin Assingment Register
PORTA EQU   MODBASE+$11 Port A Data Register
PORTB EQU   MODBASE+$19 Port B Data Register
DDRB  EQU   MODBASE+$1D Port B Data Data Direction Register

* Initiera Port A som inport
      MOVE.B      #$FF,(PPRA1).L      Port A är I/O-port
      MOVE.B      #0,(DDRA).L         A är inport
* Initiera Port B som utport
      ORI.W       #$1000,(MCR).L      Ej Chip-Selekt pinnar
      MOVE.B      #0,(PPARB).L        Port B är IO
      MOVE.B      #$FF,(DDRB).L       Port B är utport
```

9.3

Med detta är SIM40 initierad och klar att användas för in- och utmatning tillsammans med exempelvis laborationskortet ML4.

9.4.2 SIM40 Registerbeskrivning

Vi kommer här att beskriva SIM40's register. Här beskrivs registrens viktigaste bitar för normal användning av MC68340. För kompletterande beskrivning så hänvisas till MOTOROLAs "MC68340 USER MANUAL" kapitel 4.

MCR - Module Configuration Register.

MCR

Supervisor Only

$000

Module configuration register används för att bestämma övergripande funktioner. Det kan ändras endast då CPU32 är i SM (*supervisor mode*).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	FRZ1	FRZ0	FIRQ	0	0	SHEN1	SHEN0	SUPV	0	0	0	IARB3	IARB2	IARB1	IARB0

Bit 12, FIRQ; Full Interrupt Request Mode.

1 = Port B är I/O eller 7 avbrottspinnar

0 = Port B är 4 avbrottspinnar och 4 Chip Select-pinnar

Bit 9-8, SHEN1-0; Show Cycle Enable.
Bestämmer hur den externa bussen skall bete sig vid interna busscykler

00 = Interna busscykler visas ej på externbussen, Extern arbitrering

01 = Interna busscykler visas på externbussen, Ej extern arbitrering

10 = Interna busscykler visas på externbussen, Extern arbitrering

Bit 3-0, IARB3-0; Interrupt Arbitration Bits. MC68340's I/O moduler måste ha olika prioriteter men externa avbrottsprioriteter kan vara de samma. Dessa bitar avgör hur bussen arbitreras vid samtidiga avbrott på samma prioritetsnivå, $F ger SIM40 högsta prioritet, $1 lägsta.

SYNCR - Clock Synthesizer Control Register.

SYNCR

Supervisor Only

$004

Med detta register kan systemets arbetstakt regleras. Utifrån en yttre kristall genereras en klockfrekvens som sedan används för att styra busshastigheten. Det kan ändras endast då CPU32 är i SM.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
W	X	Y5	Y4	Y3	Y2	Y1	Y0	RSVD	0	0	SLIMP	SLOCK	RSTEN	STSIM	STEXT

Efter RESET fås en systemklocka (CLOCK) på 8.39MHz när kristallfrekvensen är 32.768 kHz. Klockan kan ställas enligt:

$$F_{SYSTEM} = F_{KRISTALL}[\ 2^{(2+2W+X)}] \times (Y+1)$$

Bit 15, W; Frequency Control Bit.
1 = Öka VCO-hastigheten med 4
0 = Normal

Bit 14, X; Frequency Control Bit.
1 = Dubbla systemklockans frekvens utan att ändra VCO-hastigheten
0 = Normal

Bit 13-8, Y5-0; Frequency Control Bit.
Räknarvärde från 0-63 som används av nerräknaren i klockmodulen.

AVR

Supervisor Only

$006

AVR - Autovector Register.

Registret anger vilka externa avbrottsnivåer som skall vara av typen autovektor. Det kan ändras endast då CPU32 är i SM.

7	6	5	4	3	2	1	0
AV7	AV6	AV5	AV4	AV3	AV2	AV1	0

EXEMPEL

```
MOVE.B      #%00011110,(mc68_avr).L
```

initierar chipet för att hantera avbrottsnivåerna 1,2,3 och 4 som autovektoravbrott.

9.4

RSR

Supervisor Only

$007

RSR - Reset Status Register.

Ett läsregister som anger källan till den senaste RESET-orsaken. Det kan läsas endast då CPU32 är i SM.

7	6	5	4	3	2	1	0
EXT	POW	SW	DBF	0	LOC	SYS	0

PORTA - Port A Data Register.

PORTA

Supervisor/User

$011

Parallellport A, dataregister. Registret är läs-/ skriv- bart då CPU32 är i såväl SM som UM (*user mode*).

7	6	5	4	3	2	1	0
P7	P6	P5	P4	P3	P2	P1	P0

DDRA - Port A Data Direction Register.

DDRA

Supervisor/User

$013

Parallellport A, riktningsregister, används för att ställa PORTA som ingång eller utgång. Registret är läs-/ skriv- bart då CPU32 är i såväl SM som UM.

7	6	5	4	3	2	1	0
DD7	DD6	DD5	DD4	DD3	DD2	DD1	DD0

Bit $i = 1$: Bit i är utgång

Bit $i = 0$: Bit i är ingång

PPARA1 - Port A Pin Assignment Register 1.

PPARA1

Supervisor Only

$015

Anger om Port A skall vara diskret I/O-port eller adressport.

7	6	5	4	3	2	1	0
PRTA7 (A31)	PRTA6 (A30)	PRTA5 (A29)	PRTA4 (A28)	PRTA3 (A27)	PRTA2 (A26)	PRTA1 (A25)	PRTA0 (A24)

Bit $i = 1$: Bit i är I/O-port, styrd av DDRA, PPARA2 Bit i är ej giltig.

Bit $i = 0$: Bit i är adresspinne eller IACK-pinne, Se PPARA2.

PPARA2 - Port A Pin Assignment Register 2.

PPARA2

Supervisor Only

$017

Anger om Bit i Port A skall vara adresspinne eller IACK-pinne om och endast om tillhörande bit är nollställd i PPARA1.

7	6	5	4	3	2	1	0
IACK7 (A31)	IACK6 (A30)	IACK5 (A29)	IACK4 (A28)	IACK3 (A27)	IACK2 (A26)	IACK1 (A25)	IACK0 (A24)

PORTB

PORTB1
Supervisor/User
$019/1B

PORTB - Port B Data Register.

Parallellport B, dataregister som kan nås på två adresser.

7	6	5	4	3	2	1	0
P7	P6	P5	P4	P3	P2	P1	P0

DDRB
Supervisor/User
$01D

DDRB - Port B Data Direction Register.

7	6	5	4	3	2	1	0
DD7	DD6	DD5	DD4	DD3	DD2	DD1	DD0

Parallellport B, riktningsregister.

Bit $i = 1$: Bit i är utgång

Bit $i = 0$: Bit i är ingång

PPARB
Supervisor Only
$01F

PPARB - Port B Pin Assignment Register.

Anger om port B skall vara diskret I/O-port eller avbrotts/Chip Select-pinne. Se även bit FIRQ i MCR.

7	6	5	4	3	2	1	0
PPARB7 (IRQ7)	PPARB6 (IRQ6)	PPARB5 (IRQ5)	PPARB4 (IRQ4)	PPARB3 (IRQ3)	PPARB2 (IRQ2)	PPARB1 (IRQ1)	PPARB0 (MOCK)

Bit $i = 1$: Bit i är avbrotts- eller Chip Select-pinne.

Bit $i = 0$: Bit i är I/O-port, styrd av DDRB

SWIV
Supervisor Only
$020

SWIV - Software Interrupt Vector Register.

Software Watchdog avbrotts vektornummer. Registret initieras till $0F vid RESET.

7	6	5	4	3	2	1	0
SWIV7	SWIV6	SWIV5	SWIV4	SWIV3	SWIV2	SWIV1	SWIV0

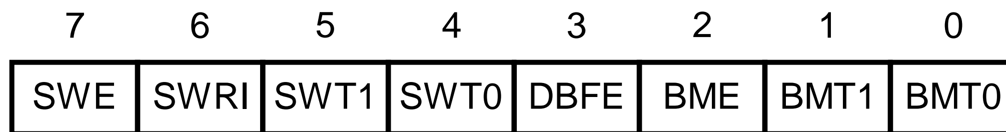
SYPCR - System Protection Control Register.

SYPCR

Supervisor Only

$021

Bitarna 7-4 styr SIM40:ans Watchdog.



7	6	5	4	3	2	1	0
SWE	SWRI	SWT1	SWT0	DBFE	BME	BMT1	BMT0

Bit 3, DBFE; Double Bus Fault Monitor Enable.

1 = Aktivera dubbel-buss-fels övervakaren
0 = Deaktivera dubbel-buss-fels övervakaren

Bit 2, BME; Bus Monitor External Enable.

1 = Aktivera buss övervakaren för en intern-till-extern cykel.
0 = Deaktivera buss övervakaren för en intern-till-extern cykel.

Bit 1-0, BMT1-0; Bus Monitor Timing.

Anger antal klockcykler som skall passera innan buss-fels-övervakaren aktiveras.

00 = 64 klockcykler
01 = 32 klockcykler
10 = 16 klockcykler
11 = 8 klockcykler

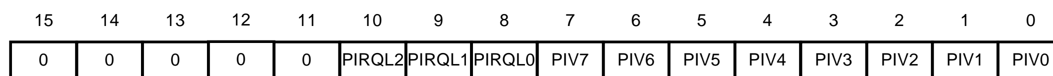
PICR - Periodic Interrupt Control Register.

PICR

Supervisor Only

$022

Används, tillsammans med PITR, för att implementera en realtidsklocka. Se även exempel nedan.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	PIRQL2	PIRQL1	PIRQL0	PIV7	PIV6	PIV5	PIV4	PIV3	PIV2	PIV1	PIV0

Bit 10-8, PIRQL2-0; Periodic Interrupt Request Level.

Anger avbrottsnivå för den periodiska räknaren i SIM40.

000 = Ej avbrott. 001 = Avbrottsnivå 1, 010 = Avbrottsnivå 2, osv.

Bit 7-0, PIV7-0; Periodic Interrupt Vector.

Vektornummer för den periodiska räknaren. Registret är initierad till $0F vid RESET

PITR

Supervisor Only

$024

PITR - Periodic Interrupt Timer Register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	SWP	PTP	PITR7	PITR6	PITR5	PITR4	PITR3	PITR2	PITR1	PITR0

Bit 9, SWP; Software Watchdog Prescale.

1 = Software Watchdog är nerdelad med 512
0 = Normal

Bit 8, PTP; Periodic Timer Prescale Control.

1 = Periodiska räknaren är nerdelad med 512
0 = Normal

Bit 7-0, PITR7-0; Periodic Interrupt Timer Register Bits.

Innehåller räknarvärdet för den periodiska räknaren. $00 anger att stänga av räknaren Periodtiden bestäms enligt:

period = PITRvärde/32768/prescaler värde/4

EXEMPEL

Följande exempel illustrerar användningen av SIM40's periodiska räknare. Initieringssekvens och exempel på avbrottsrutin visas.

```
* Adresser för SIM40 i MC68
mc68_sim40_mcr          equ   $FFFFF000
mc68_sim40_picr         equ   $FFFFF022
mc68_sim40_pitr         equ   $FFFFF024

_start_timer:
* Initiera och starta den periodiska räknaren
* aktivera intern arbitrering
      MOVE.W      #$000F,(mc68_sim40_mcr).L

* installera avbrottsvektor
      MOVE.L      #timer_interrupt,($100).L

* avbrottsnivå 6, avbrottsnummer $40
      MOVE.W      #$0640,(mc68_sim40_picr).L

* tidsbas (10 ms)
      MOVE.W      #$10,(mc68_sim40_pitr).L
      RTS


timer_interrupt:
* anm. avbrott behöver ej kvitteras...
      RTE
```

9.5

SWSR - Software Service Register

SWSR

Supervisor Only

$027

Detta register används endast tillsammans med *watchdog*-funktionen. För att inte watchdogen ska aktiveras måste programvaran, med jämna mellanrum, skriva något i detta register. Vid läsning från detta register returneras alltid 0.

7	6	5	4	3	2	1	0
SWSR7	SWSR6	SWSR5	SWSR4	SWSR3	SWSR2	SWSR1	SWSR0

AMR1, AMR2 - Address Mask Register.

AMR1

Supervisor Only

$040/48/50/58

AMR2

Supervisor Only

$042/4A/52/5A

Med dessa register kan kretsen fås att generera “chip-select”-signaler till externa kretsar. Det finns fyra 32-bitars register, ett för varje Chip Select-pinne. Varje register har följande layout:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AM31	AM30	AM29	AM28	AM27	AM26	AM25	AM24	AM23	AM22	AM21	AM20	AM19	AM18	AM17	AM16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AM15	AM14	AM13	AM12	AM11	AM10	AM9	AM8	FCM3	FCM2	FCM1	FCM0	DD1	DD0	PS1	PS0

Bit 31-8, AM31-8; Address Mask Bits.

Anger storleken på den minnesarea som skall aktiveras.

Bit 7-4, FCM3-0; Function Code Mask Bits.

Maskera Function Code

Bit 3-2, DD1-0; DSACK Delay Bits.

Anger antal Wait States för aktuell Chip Select-pinne

00 = Noll Wait State
01 =En Wait State
10 = Två Wait State
11 = Tre Wait State

Bit 1-0, PS1-0; Port Size Bits.

Anger om intern DSACK-generering skall användas.

00 = Reserverad
01 = Intern DSACK-generering, 16-bitars databuss.
10 = Intern DSACK-generering, 8-bitars databuss.
11 = Extern DSACK-generering.

BAR1
Supervisor Only
$044/4C/52/5C
BAR2
Supervisor Only
$046/4E/56/5E

BAR1, BAR2 - Base Address Register.

Det finns fyra 32-bitars register, ett för varje Chip Select-pinne

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BA31	BA30	BA29	BA28	BA27	BA26	BA25	BA24	BA23	BA22	BA21	BA20	BA19	BA18	BA17	BA16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BA15	BA14	BA13	BA12	BA11	BA10	BA9	BA8	BFC3	BFC2	BCF1	BCF0	WP	FTE	NCS	V

Bit 31-8, BA31-8; Base Address Bits.

Anger startadressen på den minnesarea som skall aktiveras.

Bit 0, V: Valid

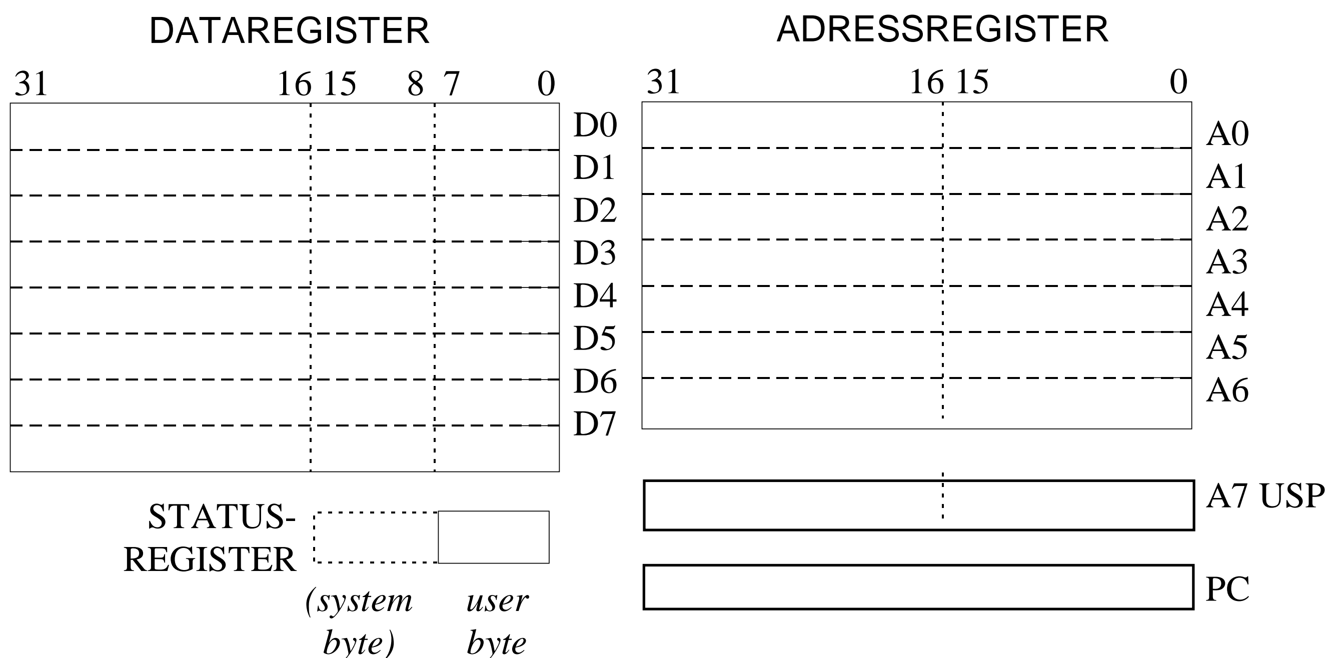
1 = Anger giltigt värde för Chip Select pinne

0 = Anger ej giltigt värde

9.4.3 CPU32

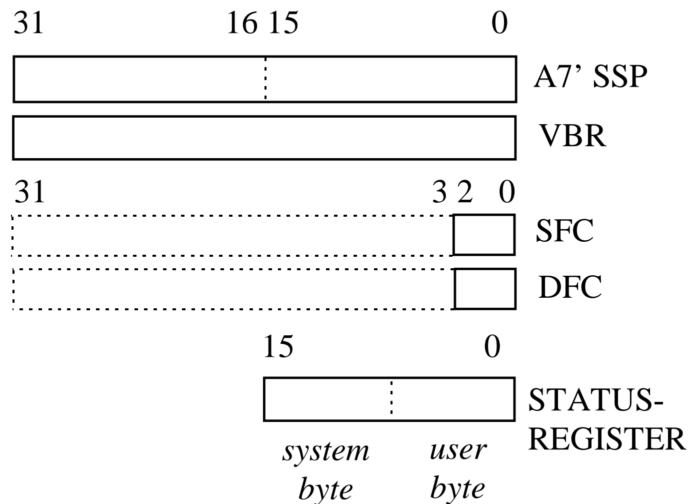
CPU32 är processor-blocket i MC68340. CPU32 är anpassad för att arbeta mot den interna bussen IMB (Inter Module Bus) som finns i ett flertal av kontroller-kretsarna i MC68300-familjen. CPU32 bygger på industristandarden MC68000 och är objektkompatibel med denna. Vidare har CPU32 flera av egenskaperna hos MC68010 och MC68020, så egentligen liknar CPU32 mer på en MC68020 processor än en ren MC68000 processor.

Vi börjar med att studera programmerarens bild i figur 9.19 som visar User Model. Överst i figuren känner vi igen de åtta 32-bitars dataregistren följt av sju adressregister. Vidare finns A7 som är User Stack Pointer och programräknaren PC. Slutligen hittas flaggregistret CCR, där man kommer åt åtta bitar i user mode.



FIGUR 9.19 CPU32 USER PROGRAMMING MODEL.

Utöver registren visade i figuren finns även ett antal register som enbart är åtkomliga i supervisor mode. Dessa visas i figur 9.20. Precis som en traditionell MC68000 processor så hittas här A7' som är Supervisor Stack Pointer och Status Register (CCR). CCR visas i 9.21. Vidare finns tre nya register: VBR, SFC och DFC. Dessa tre sista registren används bland annat vid virtuell minneshantering. CPU32 Supervisor Programming Model är identisk med MC68010:s Programming Mode.



FIGUR 9.20 CPU32 SUPERVISOR PROGRAMMING MODEL

VBR registret (*Vector Base Register*) innehåller basadressen till tabellen med exeptionvektorer i systemet. På så sätt kan denna placeras var som helst i minnet. Registret kan ändras under körning, så att man dynamiskt kan ändra exeptionvektorerna under drift. När processorn får en exeption, exempelvis vektornummer 5 (Divide by Zero) som motsvarar adress $14 kommer processorn att addera $14 till innehållet i VBR innan den läser startadressen för avbrottsrutinen i minnet. (Jämför avsnittet om avbrott.) De sista registren betecknas *Alternate Source* och *Destination Function Code Register* (SFC och DFC) och innehåller en tre-bitars funktionskod. Innehållet i registren kan användas för att koppla en viss funktionskod till en viss adressrymd.

Som en direkt följd av införandet nya register har också en ny instruktion införts, den har två former:

`MOVEC        Rn,Rc` (*move control register*)

kopierar innehållet i ett generellt register (R=Data- eller Adressregister) till ett kontrollregister Rc.

Den andre formen:

```
MOVEC        Rc,Rn
```

kopierar innehållet i ett kontrollregister till ett generellt register.

Införandet av VBR ger utmärkta möjligheter till differentierad exceptionhantering, dvs ett snabbt och enkelt sätt att ändra innehållet i minnesarean med exceptionvektorer. Betrakta följande exempel.

EXEMPEL

Skifta exceptionhantering

Antag att vi definierat och initierat två olika tabeller med exceptionvektorer. Den första tabellen `Default_EH` används under normala omständigheter men vi har också en tabell `Test_EH` som vi vill använda då vi exekverar en subrutin `test`.
Med CPU32 kan vi då skifta exceptionhantering och starta test-programmet enligt följande.

```
      MOVEA.L     #Test_EH,A0       tabell för testprogram
      MOVEC       A0,VBR
      JSR         test              starta testprogram
      MOVEA.L     #Default_EH,A0    återställ ursprunglig tabell
      MOVEC       A0,VBR
```

Vill vi göra samma sak med en MC68000 processor får vi utföra följande:

```
* installera tabell för testprogram
      MOVE.L      #$FFF,D0          tabellens sorlek
      MOVEA.L     #Test_EH,A0       ny tabell
      MOVEA.L     #0,A1             startadress för tabell
_1:
      MOVE.L      (A0)+,(A1)+
      DBRA        D0,_1

      JSR         test              starta testprogram

* återställ ursprunglig tabell
      MOVE.L      #$FFF,D0          tabellens sorlek
      MOVEA.L     #Default_EH,A0
      MOVEA.L     #0,A1             startadress för tabell
_2:
      MOVE.L      (A0)+,(A1)+
      DBRA        D0,_2
```

Införandet av VBR underlättar alltså väsentligt en differentierad exceptionhantering.

9.6

Processorn anger, för varje buss-cykel, en specifik adressrymd. Adressrymden bestäms av den mode processorn befinner sig i (SM eller UM) och den typ av åtkomst som utförs (program, dvs instruktion, eller data). Utöver detta finns en speciell adressrymd (*CPU Space*) som används vid kommunikation med så kallade coprocessorer, dvs minneshanteringskretsar, flyttalsprocessorer mm.

Informationen som kallas *Function Codes* finns tillgänglig som tre pinnar på processorns kapsel. Innebörden av dessa ges i följande tabell:

FC2	FC1	FC0	Adressrymd
0	0	0	Reserverad för framtida bruk
0	0	1	User Mode Data
0	1	0	User Mode Program
0	1	1	Reserverad: Användardefinierad
1	0	0	Reserverad för framtida bruk
1	0	1	Supervisor Mode Data
1	1	0	Supervisor Mode Program
1	1	1	Interrupt Acknowledge (CPU Space)

Signalerna kan användas i minnesavkodningen för att skapa ett enkelt men effektivt minnesskydd, dvs det minne som reserverats för SM kan aldrig överhuvudtaget manipuleras på processorn är i UM. För att kopiera data mellan de olika adressrymderna används en priviligierad instruktion, *Move Address Space* (MOVES) tillsammans med två stycken 3-bitars register (gäller ej MC68000):

```
MOVES Rn,<ea>
```

kopierar innehållet i ett data- eller adress- register (Rn) till adress som anges av <ea> i den adressrymd som anges av registret *Destination Function Codes*, (DFC)

Med formen:

```
MOVES <ea>,Rn
```

kopieras innehållet från adress som anges av <ea> i den adressrymd som anges av registret *Source Function Codes*, (SFC)

i ett data- eller adress- register (Rn) till

Innehållen i SFC och DFC kan läsas/skrivas med MOVEC - instruktionen.

EXEMPEL

Datakopiering mellan olika adressrymder

För att kopiera minnesinnehåll från minne som endast är tillgängligt i SM (SM Data) till minne som är tillgängligt även i UM kan följande sekvens användas:

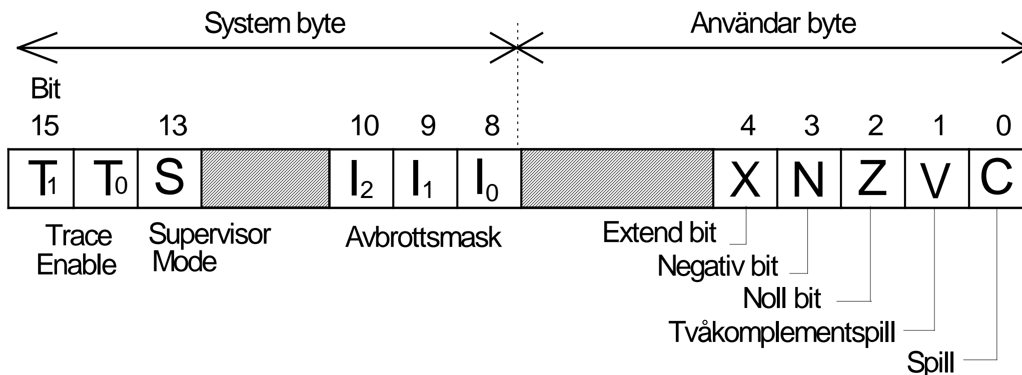
```
MOVE.L      (Source_Data).L,D0
MOVEC       #%001,DFC
MOVES       D0,(Destination).L
```

På analogt sätt kan data kopieras in till skyddat minne enligt:

```
MOVEC       #%001,SFC
MOVES       (Source_Data).L,D0
MOVE.L      (Destination).L,D0
```

9.7

Slutligen finns Status Register, (figur 9.21), som bortsett från bit 14 och 15 är identiskt med MC68000:s statusregister. Bit T1 och T0 anger trace-funktion. Man kan ange att få en exeption efter varje fullgjord instruktion eller efter en programflödesändring.



FIGUR 9.21 CPU32 STATUS REGISTER

Processorn stöder alla adresseringssätt som MC68000 (kompatibilitet). Dessutom har man här infört ytterligare två adresseringssätt.

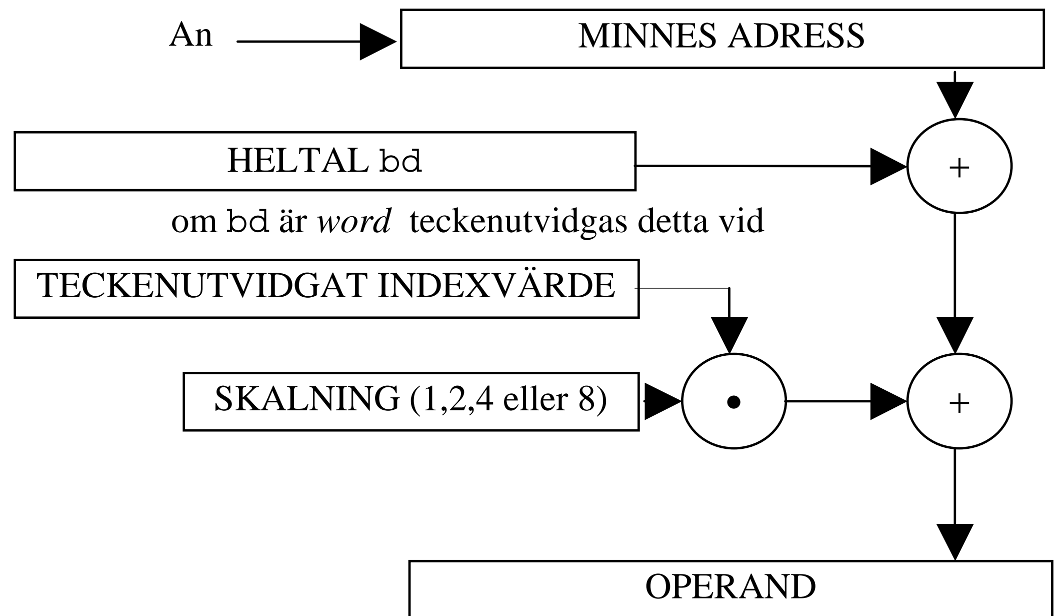
Adress Register Indirect with index (base displacement)

Assemblersyntax: `(bd,An,Xn.s*SCALE)`

Adressen till data bildas genom att den godtyckliga konstanten *bd* och innehållet i indexregistret adderas till innehållet i `An`. Indexregistret, ovan angett med `Xn`, kan vara godtyckligt data eller adress-register. Storleken hos indexet, ovan angett med `s`, kan vara `l` (*long*), dvs innehållet adderas oförändrat vid adressbildningen. Alternativt kan indexet vara `w` (*word*) varvid endast de 16 minst signifikanta bitarna i indexregistret används. Detta teckenutvidgas till 32 bitar innan det adderas vid adressbildningen. (Se även figur nedan)

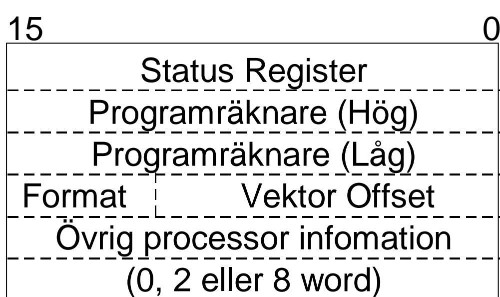
Slutligen multipliceras innehållet i indexregistret med en konstant (1,2,4 eller 8) ovan angett med `SCALE` och adressen till data har bildats. SCALE behöver ej anges och ersätts då med 1.

Alla komponenter i adressbildningen behöver inte anges. En utelämnad komponent ersätts med värdet 0 vid adressberäkningen.



Instruktionsuppsättningen är ungefär den den samma som för MC68020. Vissa grupper saknas dock, exempelvis bitfältsinstruktioner, coprocessor-instruktioner, etc. För ytterligare information hänvisas till instruktionslistan.

En intressant instruktion, implementerad endast i CP32 är LPSTOP, som när den exekveras, får MC68340 till att stanna och gå över i strömsnål mod. I denna mod är strömförbrukningen endast 60 µA. Kretsen återgår till normaldrift efter en avbrottsbegäran eller en RESET beroende på hur den är initierad. Detta gör att kretsen är mycket lämplig för batteridrift.

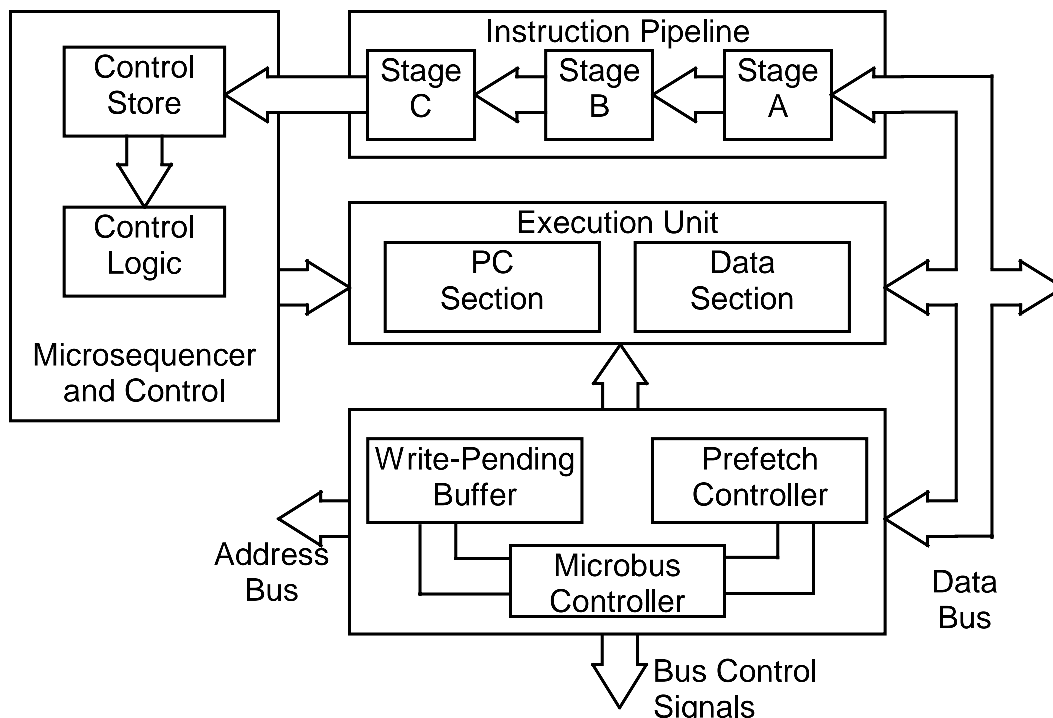


FIGUR 9.22 *EXCEPTION STACK FRAME*

Exception utförs i princip som hos de övriga MC68xxx- processorerna. Vid multipla exeptions utförs dessa beroende på prioritet. Högst prioritet har RESET, sedan kommer adress- och bussfel, sedan kommer en grupp av TRAP-instruktioner etc. och sedan kommer otillåten operationskod, Line A och Line F exeption. Sist på lägsta prioritet kommer trace och vanliga avbrott från periferikretsar. Figur 9.22 visar hur registren placeras på stacken vid exceptions (*Stack Frame*). Observera att det finns flera olika format för sådana Stack Frames. Speciellt finner man skillnader mellan CPU32 och MC68000.

Studerar man den interna uppbyggnaden av CPU32 så hittas fyra block (figur 9.23) . Själva styrenheten hitttas längst till vänster.

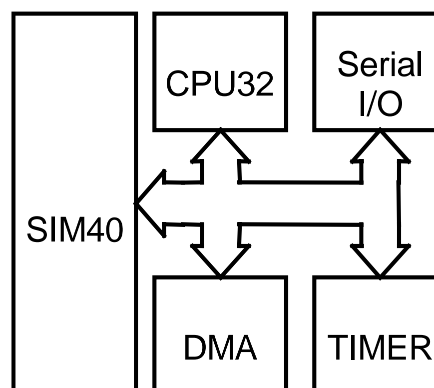
Denna blir "matad" med operationskoder från ett block som har en tre-stegs instruktions-pipeline. Styrenheten styr datavägen i med data-del och adress-del (Execution Unit mitt i figuren). Slutligen finns ett block som styr instruktions-pipelinen och de externa bussarna.



FIGUR 9.23 INTERN UPPBYGGNAD AV CPU32

9.4.4 I/O-block

MC68340 är utrustad med en tvåkanals DMA-krets, två räknarmoduler (utöver den periodiska räknaren i SIM40 och slutligen en tvåkanals seriemodul. Se 9.26. DMA-kanalen behandlas inte vidare. Däremot skall vi studera seriemodulen och räknarmodulen.



FIGUR 9.24 MC68340:S OLIKA I/O-BLOCK

Seriekretsen är egentligen DUART-kretsen MC68681 som beskrevs i kapitlet som behandlar periferikretsar. Vi kommer här att påpeka de skillnader som gäller. Annars hänvisas till avsnittet som behandlar MC68681. Funktionen hos räknarmodulerna är mycket lika det som är beskrivet när MC68230:s räknare behandlades. Denna räknare har dock några fler möjligheter. Vi behandlar seriekretsen först.

9.4.5 Seriemodul

Seriemodulen är i princip MC68681 som beskrevs i ett tidigare kapitel. Följande skillnader finns dock:

- Interrupt Vektor Register är flyttat till Supervisor area och är placerad på en ny adress.
- MR2A MR2B som tidigare hade "dolda" adresser är flyttade till egna adresser.
- Räknardelen är borttagen
- IP-pinnarna saknas på MC68340 vilket medför att vissa bitar i Input/Output Port-registren saknar betydelse.
- Man måste invänta stabil kristallfrekvens vid spänningstillslag.

Nedan följer seriemodulens registeruppsättning. Vi hänvisar till kapitlet som beskriver MC68681 för en utförlig beskrivning av de enskilda registren.

Mcrh	Module Configuration Register, High	R/W	S	$700
mcrl	Module Configuration Register, Low	R/W	S	$701
ilr	Interrupt Level Register	R/W	S	$704
ivr	Interrupt Vector Register	R/W	S	$705
mr1a	Mode Register 1 A	R/W	S/U	$710
sra	Status Register A	R	S/U	$711
csra	Clock Select Register A	W	S/U	$711
cra	Command Register A	W	S/U	$712
rba	Receiver Buffer A	R	S/U	$713
tba	Transmitter Buffer A	W	S/U	$713
ipcr	Input Port Change Register	R	S/U	$714
acr	Auxiliary Control Register	W	S/U	$714
isr	Interrupt Status Register	R	S/U	$715
ier	Interrupt Enable Register	W	S/U	$715
mr1b	Mode Register 1 B	R/W	S/U	$718
srb	Status Register B	R	S/U	$719
csrb	Clock Select Register B	W	S/U	$719
crb	Command Register B	W	S/U	$71A
rbb	Receiver Buffer B	R	S/U	$71B
tbb	Transmitter Buffer B	W	S/U	$71B
ip	Input Port	R	S/U	$71D
opcr	Output Port Control Register	W	S/U	$71D
ops	Output Port Data Register	W	S/U	71E
opc	Output Port Data Register	W	S/U	71F
mr2a	Mode Register 2 A	R/W	S/U	$720
mr2b	Mode Register 2 B	R/W	S/U	$721

Ett nytt register måste dock beskrivas. Det är MCR som är ett styrregister för hela seriemodulen.

Module Configuration Register, High.

7	6	5	4	3	2	1	0
STP	FRZ1	FRZ0	ICCS	0	0	0	0

MCRH

Supervisor Only

$700

Bit 7, STP; Stop Mode Bit.

1 = Seriemodulen är bortkopplad och klockorna är stoppade
0 = Normal

Bit 6-5, FRZ1-0; Freeze

Bestämmer åtgärd när FREEZE-signalen aktiveras. Används för debug ändamål.

Bit 4, ICCS; Input Capture Clock Select

1 = SCLK väljs som Clear-to-send Input Capture Clock
0 = Kristallen väljs som Clear-to-send Input Capture Clock

Module Configuration Register, Low.

MCRL

Supervisor Only

$701

7	6	5	4	3	2	1	0
SUPV	0	0	0	IARB			

Bit 7, SUPV; Supervisor/User.

1 = Modulens register är enbart åtkommliga i supervisor mod.
0 = Modulens register (de märka S/U i registersamman-ställningen) är åtkommliga i supervisor eller användar mod.

Bit 3-0, IARB; Interrupt Arbitration Bits.

Arbitrering vid samtidiga avbrott. $F högsta prioritet, $1 lägsta. MC68340:s I/O-moduler måste ha olika prioriteter.

Nedan följer ett exempel på hur seriemodulen kan initieras. Exemplet är hämtat från boot-laddaren för MC68.

EXEMPEL

Initiering av seriemodulen, port A i MC68

```
*
iniuart:
* Aktivera seriemodulen
    MOVE.B      #$0,(MCRH).L      Normal, Ingen Freeze
    MOVE.B      #$0,(MCRL).L      User mode

* Invänta stabil kristall,behövs endast vid start efter RESET
    MOVE.L      #$2000,D0
wait1:
    BTST.B      #3,(SRA).L        SRA
    NOP
    DBNE        D0,wait1

    MOVE.B      #$20,(CRA).L      Återställ mottagare
    MOVE.B      #$30,(CRA).L      Återställ sändare
    MOVE.B      #$13,(MR1A).L     Ingen Paritet, 8 databitar
    MOVE.B      #$7,(MR2A).L      En stoppbit
    MOVE.B      #$BB,(CSRA).L     9600 BAUD
    MOVE.B      #$0,(ACR).L
    MOVE.B      #$45,(CRA).L
* Sätt RTS, Aktivera sändare/mottagare
    RTS
```

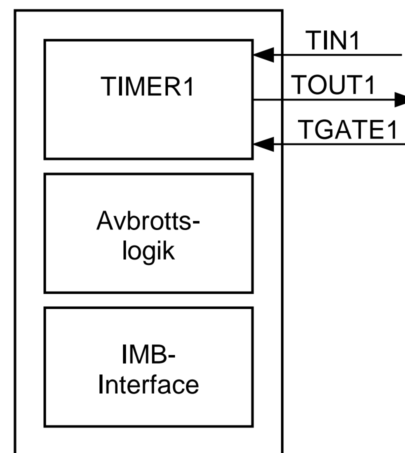
9.8

9.4.6 Räknarmodul

MC68340 innehåller två identiska räknarkretsar som kan användas som

- generell räknarkrets
- pulsgenerator
- fyrkantgenerator med variabel “Duty Cycle”
- pulsräknare
- periodräknare
- pulsbreddsmodulering
- mm.

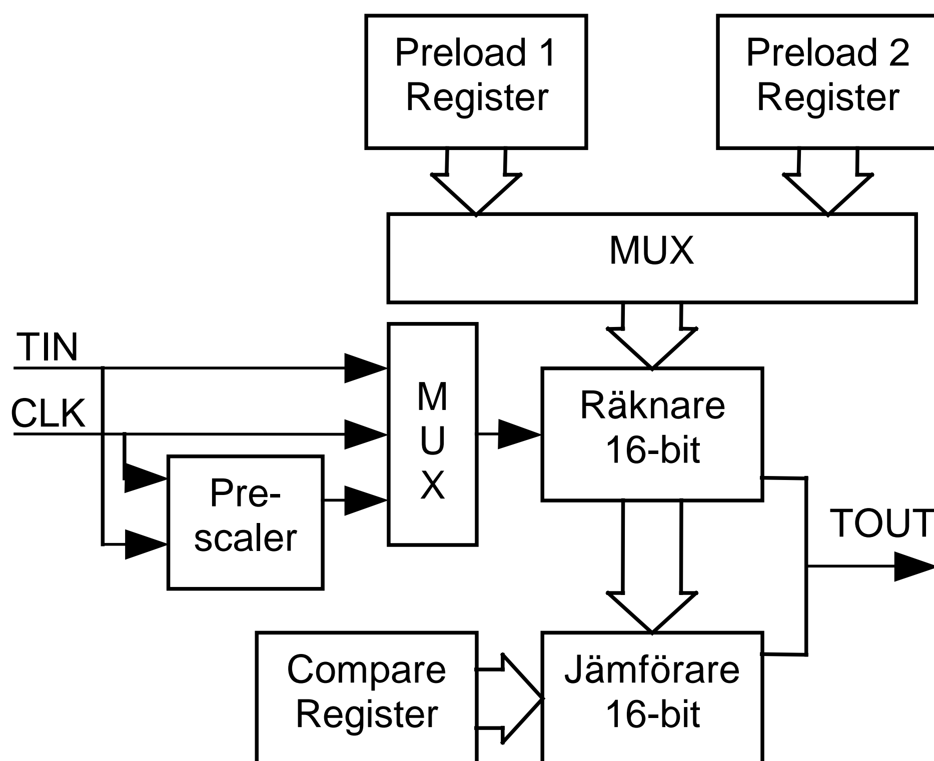
Figur 9.25 visar ett blockdiagram för en av räknarkretsarna i modulen. Internt har räknaren en 16-bitars räknare, TIMER 1. Vidare finns ett block som styr avbrottsgenereringen. Slutligen finns block som utgör gränssnittet mot den interna bussen (IMB) där bland annat systemklockan ingår som kan användas för att klocka räknaren.



FIGUR 9.25 BLOCKDIAGRAM FÖR RÄKNARKRETSEN

Räknaren har tre externa anslutningar. TIN som kan användas som klockingång eller för att styra räknaren. TOUT är räknarens utsignal och på denna anslutning kan man få en puls, en fyrkantvåg, mm. Slutligen finns insiganmlen TGATE som används för att exempelvis starta och stoppa räknaren.

Internt är räknaren uppbyggd kring en 16-bitars ner-räknare. Denna kan utökas till en 24-bitar räknare om det 8-bitars Prescaler-blocket utnyttjas. Via en multiplexer väljs önskad klocka. Se figur 9.26



FIGUR 9.26 RÄKNARENS UPPBYGGNAD

Räknaren får i normalfallet sitt startvärde från Preload 1 Register. Man kan även välja att växelvis hämta startvärden från Preload 1 Register och Preload 2 Register. På så sätt kan man erhålla ett pulstog ut med önskad “Duty Cycle”. Slutligen finns ett block som jämför räknarens

värde mot innehållet i Compare Register. När dessa är lika kan man exempelxis få ett avbrott. Nedan visas räknarnas registeruppsättning. Observera att alla register är på 16 bitar.

S anger att registret enbart är adresserbart i supervisor mode.

S/U anger att registret är adresserbart i supervisor och eller user mode beroende på hur bit 7 i MCR är initierad.

MCR	Module Configuration Register	S	$600	$640
IR	Interrupt Register	S	$604	$644
CR	Control Register	S/U	$606	$646
SR	Status Prescaler Register	S/U	$608	$648
CNTR	Counter Register	S/U	$60A	$64A
PREL1	Preload 1 Register	S/U	$60C	$64C
PREL2	Preload 2 Register	S/U	$60E	$64E
COM	Compare Register	S/U	$610	$650

Nedan kommer en beskrivning av räknarens register och dess mest använda bitar.

MCR

Supervisor Only

$600, $640

MCR - Module Configuration Register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STP	FRZ1	FRZ0	0	0	0	0	0	SUPV	0	0	0	IARB3	IARB2	IARB1	IARB0

Bit 15, STP; Stop Mode Bit.

1 = Seriemodulen är bortkopplad och klockorna är stannade
0 = Normal

Bit 14-13, FRZ1-0; Freeze

Bestämmer åtgärd när FREEZE-signalen aktiveras. Används för debug ändamål

Bit 7, SUPV; Supervisor/User.

1 = Modulens register är enbart åtkomliga i supervisor mod.
0 = Modulens register (de märka S/U i registersamman-ställningen) är åtkomliga i supervisor eller användar mod.

Bit 3-0, IARB3-0; Interrupt Arbitration Bits.

Arbitrering vid samtidiga avbrott. $F högsta prioritet, $1 lägsta. MC68340:s I/O-moduler måste ha olika prioriteter.

IR - Interrupt Register.

IR

Supervisor Only

$604, $644

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	IL2	IL1	IL0	IVR7	IVR6	IVR5	IVR4	IVR3	IVR2	IVR1	IVR0

Bit 10-8, IL2-0; Interrupt Level Bits.

Anger önskad avbrottsnivå. 7 anger högsta nivå.

Bit 7-0, IVR7-0; Interrupt Vektor Bits.

Anger önskad avbrottsvektornummer. Sätts till $0F vid RESET

CR - Control Register.

CR

Supervisor / User

$606, $646

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWR	IE2	IE1	IE0	TGE	PCLK	CPE	CLK	POT2	POT1	POT0	MODE2	MODE1	MODE0	OC1	OC0

Bit 15, SWR; Software Reset.

1 = Normal
0 = Software Reset, Räknaren är stoppad

Bit 14-12, IE2-0; Interrupt Enable

Anger vilken källa (eller vilka källor) som skall generera avbrott.

000 = Polling, Ej avbrott
Bit 14, IE2 = 1: TO (Timer Out) avbrott
Bit 13, IE1 = 1: TG (Timer Gate) avbrott
Bit 12, IE0 = 1: TC (Timer Compare) avbrott

Bit 11, TGE; Timer Gate Enable

1 = TGATE-signalen styr räknaren
0 = TGATE-signalen har ingen inverkan på räknare.

Bit 10, PCLK; Prescaler Clock Select

1 = Prescaler används
0 = Prescaler används inte.

Bit 9, CPE; Counter Prescaler Enable

1 = Vald klocka är aktiv
0 = Vald klocka ettställs och därmed klockas varken räknare eller prescaler

Bit 8, CLK; Clock

1 = TIN används för att klocka räknarmodulen
0 = Systemklockas halva frekvens används för att klocka räknarmodulen

Bit 7-5, POT2-0; Prescaler Output Tap

Anger hur mycket prescaler skall dela ner vald klockingång

000 = Dela ner med 256
001 = Dela ner med 2
010 = Dela ner med 4
011 = Dela ner med8
100 = Dela ner med 16
101 = Dela ner med 32
110 = Dela ner med 64
111 = Dela ner med 128

Bit 4-2, MODE2-0; Operation Mode

Anger vilken mod som skall användas

000 = Jämför / Kontinuerlig mod
001 = Fyrkantgenerator
010 = Fyrkantgenerator, variabel Duty Cycle
011 = Pulsgenerator, Single Shot
100 = Pulsbreddsmätning
101 = Periodtidsmätning
110 = Händelseräkning
111 = Timer bypass (Test mode)

Bit 1-0, OC1-0; Output Control

Anger hur utgången (TOUT) skall ändra värde vid en händelse, exempelvis Time Out.

00 = TOUT är i tri state
01 = TOUT togglar vid en händelse
10 = TOUT nollställs vid en händelse
11 = TOUT ettställs vid en händelse

SR - Status Register.

SR

Supervisor / User

$608, $648

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ	TO	TG	TC	TGL	ON	OUT	COM	PO7	PO6	PO5	PO4	PO3	PO2	PO1	PO0

Bit 15, IRQ; Interrupt Request Bit.

1 = Ett avbrott, TO, TG eller TC har inträffat
0 = Inget avbrott

Bit 14, TO; Timeout Interrupt.

1 = Ett TO avbrott har inträffat, Räknaren har ändrat värde från $0001 till $0000
0 = Inget TO avbrott. Biten nollställs vid RESET eller vid att skriva en etta till den. Skrivs en nolla behålls bitens värde.

Bit 13, TG; Timer Gate Interrupt.

1 = Ett TG avbrott har inträffat,
0 = Inget TG avbrott. Biten nollställs vid RESET eller vid att skriva en etta till den. Skrivs en nolla behålls bitens värde.

Bit 12, TC; Timer Compare Interrupt.

1 = Ett TC avbrott har inträffat, Räknarens värde är identisk med det i COM-registret
0 = Inget TC avbrott. Biten nollställs vid RESET eller vid att skriva en etta till den. Skrivs en nolla behålls bitens värde.

Bit 11, TGL; TGATE Level.

1 = TGATE-signalen är hög
0 = TGATE-signalen är låg

Bit 10, ON; Counter Enable

1 = Räknaren är aktiverad
0 = Räknaren är inte aktiverad

Bit 9, OUT; Output Level

1 = Utgången TOUT är hög
0 = Utgången TOUT är låg eller i tri state

Bit 8, COM; Compare Bit

1 = Nerräknaren har ett lägre värde än värdet i COM-registret
0 = Nerräknaren har ett högre värde än värdet i COM-registret

Bit 7-0, PO7-0; Prescaler Output.

Bitarna anger prescalerens värde.

CR

Supervisor / User

$60A, $64A

CR - Counter Register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT15	CNT14	CNT13	CNT12	CNT11	CNT10	CNT9	CNT8	CNT7	CNT6	CNT5	CNT4	CNT3	CNT2	CNT1	CNT0

Bit 15-0, CNT15-0; Counter bits.

Registret som kan läsas anger räknarens värde.

PREL1

Supervisor / User

$60C, $64C

Preload 1 Register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR1-15	PR1-14	PR1-13	PR1-12	PR1-11	PR1-10	PR1-9	PR1-8	PR1-7	PR1-6	PR1-5	PR1-4	PR1-3	PR1-2	PR1-1	PR1-0

Bit 15-0, PR1-15-0; Preload 1 Register Bits

Här anges normalt räknarens startvärde

PREL2

Supervisor / User

$60E, $64E

Preload 2 Register.

Används tillsammans med PREL1 för att exempelvis ange variabel *duty cycle*.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR2-15	PR2-14	PR2-13	PR2-12	PR2-11	PR2-10	PR2-9	PR2-8	PR2-7	PR2-6	PR2-5	PR2-4	PR2-3	PR2-2	PR2-1	PR2-0

Bit 15-0, PR2-15-0; Preload 2 Register Bits

Här anges räknarens nya startvärde efter det att den räknat ner värdet från PREL1

COM

Supervisor / User

$610, $650

Compare 2 Register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COM15	COM14	COM13	COM12	COM11	COM10	COM9	COM8	COM7	COM6	COM5	COM4	COM3	COM2	COM1	COM0

Bit 15-0, COM15-0; Compare Register Bits

Här anges värdet som räknarens innehåll skall jämföras med.

Vi avslutar med ett exempel på initiering av räknaren.

EXEMPEL

```
* Initiera TIMER 1 för att fungera som en
* fyrkantspulsgenerator

* Stanna timer1
      CLR.W (CR1).L

* Nollställ the TO, TG, and TC bitarna
      CLR.W (SR1).L

* Module configuration register (MCR):
* Timer1 module normal operation, Ingen FREEZE.
* User mode, Interrupt arbitration exempelvis $03.
      MOVE.W      #$0003,(MCR1).L

* Initiera timer1 till avbrottsnivå 2 och
* vektornummer till $0F
      MOVE.W      #$020F,(IR1).L

* Initiera Preloadregistr 1 till 3
      MOVE.W      #$0003,(PREL11).L

* Nollställ compare register
      CLR.W (COM1)

* Styrregister 1:
* Aktivera timer1, Inget avbrott aktiveras
* TGATE signalen utnyttjas ej
* Använd system-klockan och aktivera denna
* Välj fyrkantvåg och toggle TOUT
      MOVE.W      #$8205,(CR1).L
```

9.9

9.5 Sammanfattning

I detta kapitel har vi beskrivit en modern enkortsdator och speciellt studerat den utökande centralenhet (microcontroller) som denna dator byggts kring.

Utvecklingen av tekniken bakom elekronikframställning går rasande snabbt och som ett resultat av detta ser vi hela tiden mer integrerade och komplexa kretsar.

Den ständigt ökande komplexitetsgraden ställer allt högre krav på förnyelse och utveckling hos dom människor som konstruerar system med dessa kretsar.