Gunnar Markesjö

Mikrodatorns ABC

Elektroniken i ett mikrodatorsystem

Gunnar Markesjö

Mikrodatorns ABC

Elektroniken i ett mikrodatorsystem

ESSELTE STUDIUM

Förord

Programmerad elektronik har blivit ett av de viktigaste områdena inom den moderna elektroniken. Programmerad elektronik kallas de elektroniska kretsar vars funktion kan styras genom programmering.

Programmerad elektronik har tidigare inte kunnat realiseras i lågpristillämpningar på grund av datorkostnad, effektkonsumtion och utrymmesbehov. Med LSI-kretsarnas ankomst har samtliga dessa begränsningar plötsligt försvunnit och datorkraften kan nu utnyttjas i alla tillämpningar där man på något sätt kan ha nytta av den programmerade elektroniken.

Programmerade elektroniska kretsar avsedda för en viss tillämpning kan byggas in direkt i en kristall. Man talar då om kundkretsar (custom design circuits). Man får i detta fall den önskade funktionen inrymd i det minsta antalet kapslar och med kontroll över kritiska tider etc. Nackdelen är att priset blir högt vid små serier. Kundkretsar behandlas i boken "Mikroprocessor i CMOS" (Markesjö, Esselte Studium, ISBN 91-24-27562-X).

Programmerad elektronik kan även realiseras med "standardkretsar", dvs med mikroprocessorer, minnen och periferikretsar av standardtyp. I denna bok ska vi studera ett mikrodatorsystem uppbyggt av standardkretsar. Det är först när man kommer upp på denna "systemnivå" man får det rätta perspektivet på modern elektronik.

Att arbeta med mikrodatorsystem kräver kunskap såväl om kretsar (maskinvara) som om programmering (programvara). Boken tar därför upp det viktiga samspelet mellan elektroniken och de program som styr elektronikkretsarnas funktion.

De första fyra kapitlen behandlar grunderna, dvs datorns funktion i princip, de elektroniska byggblocken och ett tillämpningsexempel av enklaste slag (trafikljus). Med detta som utgångspunkt analyseras i de efterföljande tre kapitlen ett generellt mikrodatorsystem (ABC80) med avseende på buss-struktur, interfacekretsar och programmering.

De första tre kapitlen kan överhoppas av den som redan känner till de vanligaste elektroniska byggblocken och datorns principiella funktion. Dessa tre kapitel är inte avsedda som nybörjarlitteratur i elektronik utan snarare som en snabb repetition (refresh) och inträning på den terminologi som används i efterföljande kapitel.

Det har under arbetets gång påpekats att läroböcker som "Mikroprocessor i CMOS" och "Mikrodatorns ABC" inte passar in i gymnasieskolans läroplan. De är heller inte skrivna för att passa in i äldre kursplan, de är skrivna för att fylla ett utomordentligt viktigt utbildningsbehov. Jag hoppas kursplanerna successivt revideras för att ge

mer plats åt den för vår tekniska utveckling så väsentliga programmerade elektroniken.

Böckerna Mikroprocessor i CMOS och Mikrodatorns ABC förutsätter enbart ett grundkunnande i elektronik och innehåller många praktiska exempel. De är därmed väl lämpade för utbildning på olika nivåer - även på gymnasiet. Största behållningen av Mikrodatorns ABC får man om man under läsningen har tillgång till mikrodatorn ABC80 och därmed kan testa typprogrammen och experimentera med egna program och kringkretsar.

Boken Mikrodatorns ABC är resultatet av ett nära årslångt samarbete mellan ABC80:s konstruktörer och författaren. Utrymmet medger inte en listning av alla bidragsgivare. Jag vill emellertid speciellt tacka Örjan Lindblom för all den tid han ägnat åt att gå igenom och för mig förklara kretsarnas funktion. Jag vill tacka Johan Finnved för hans tålmodiga arbete att lära mig programvarans finesser i ABC80. Björn Ahlén har hjälpt mig med trafikstyrningsprogrammen. Claes Jennel har rensat ut en rad oklarheter ur korrekturet.

Till alla medarbetare på LUXOR, DIAB och SCANDIA METRIC, samt SATTCO, TEXAS INSTRUMENTS och INTERELCO som hjälpt mig med material och information vill jag rikta ett varmt tack. Det är det gedigna kunnandet hos alla dessa medarbetare som jag har haft förmånen att förmedla till mina läsare.

Sist men inte minst vill jag tacka min medarbetare och hustru Friede som har förvandlat oläsliga handskrivna sidor till ett läsbart manus och ej misströstat trots otaliga manusrevisioner. Utan detta stimulerande samarbete hade boken aldrig blivit till.

St Aspö i augusti 1978

Gunnar Markesjö

Teckningar: Nils Svensson
Fotografier: Erwin Bennedich, Luxor, Sattco, Scandia Metric

Första upplagan, första tryckningen
ISBN 91-24-29008-4
© 1978, Gunnar Markesjö och Esselte Studium AB
Esselte Herzogs, Nacka 1978

Innehåll

ABC OM MIKRODATORER 1

1. EN DATOR I AKTION 2

1. Trafikljusens uppgift 2
2. Tre principlösningar 4
3. Interface-kretsar 4
3.1 Ingångskretsen 7
3.2 Utgångskretsen 7
4. Datorns uppgift 8

2. ELEKTRONISKA BYGGBLOCK 9

1. Fyra grundkretsar 9
1.1 Inverteraren 10
1.2 Grinden 11
1.3 Tristate-kretsen 13
1.4 Vippan 14
2. Kombinationskretsar 17
2.1 Grindnät 17
2.2 Avkodare 20
2.3 Läsminnen 21
2.4 Multiplexern 25
3. Sekvenskretsar 26
3.1 Register 26
3.2 Räknare 28
3.3 Skriv/läs-minne 29

3. DATORNS FUNKTION I PRINCIP 33

1. Utgångspunkten 33
2. Bussarna 35
3. Mikroprocessorn 37
3.1 CPU:ns register 38
3.2 Styrenheten 41
3.3 Mikroprogram 42
3.4 ALU:n 44
3.5 CPU:ns flaggor 53
3.6 Stackpekaren 55
3.7 En subrutin 56
3.8 CPU:ns uppbyggnad 58
4. Trafikljusprogrammet 62
4.1 Huvudprogrammet 62
4.2 Subrutinen LJUS 64

4. SYSTEM FÖR TRAFIKLJUSSTYRNING 66

1. Hårdvaran 66
1.1 Kontrollbussen 66

1.2 En generell datorstruktur 68
1.3 Ett minimalsystem 69
1.4 Egen mikrodator eller MCB? 72
2. Mjukvaran 74
2.1 MEMO-koden för Z80 74
2.2 Assemblerformat 76
2.3 Hur skriver vi ett assemblerprogram? 78
2.4 Ett enkelt källprogram 79
2.5 Källprogrammet assemblerat 86
2.6 Ett professionellt skrivet program 86

5. ELEKTRONIKEN I ABC80 89

1. Ett generellt system 89
2. Blockschemat 91
3. Minnen 97
3.1 ROM 98
3.2 RAM 100
4. Ljudgeneratorn 105
4.1 Teknologier 105
4.2 Ljudgeneratorns funktion 107
5. PIO-kretsen 114
5.1 Avbrott 115
5.2 PIO:ns blockschema 118
5.3 PIO:ns inkoppling i ABC80 120
6. Busskontakten 125
6.1 Anslutning av yttre minnen 126
6.2 Anslutning av yttre IO-enheter 129

6. PERIFERIENHETER

1. Tangentbordet 133
1.1 Koder från tangentbordet 135
1.2 Elektroniken i tangentbordet 137
2. Videointerfacet 143
2.1 TV-bilden 144
2.2 Räknarna 145
2.3 Bildadress och minnesadress 146
2.4 Teckengenerering 150
2.5 Tecken, graf och markör 152
2.6 Grafiken i ABC80 154
2.7 Teckenmod och grafmod 155
2.8 Specialkoder 157
2.9 Bildminnet och CPU:n 158
2.10 Videosignalen 160
3. Kassettinterfacet 160
3.1 Inspelningsmetoder 160
3.2 Kretsarna i kassettinterfacet 164
3.3 ABC80:s kassettprogram 167

4. ABC-bussen 168
4.1 Principen 170
4.2 Minnesbussen 171
4.3 Exempel på ett minneskort 172
4.4 IO-bussen 174
4.5 Exempel på ett IO-kort 175
4.6 ABC-bussens praktiska uppbyggnad 179

7. PROGRAMVARA 181

1. Initialisering 182
1.1 Tre rutiner 182
1.2 Övriga nollställningar 184
1.3 PIO:ns programmering 184
1.4 Pekare 184
1.5 Minnet 185
2. Basic-tolken 186
2.1 Radbuffertern 186
2.2 Kommando eller BASIC-sats 187
2.3 Satsbufferten BUF2 188
2.4 Tre typer av variabler 189
2.5 Internkodskompilatorn 190
2.6 Exekvering eller lagring 191
2.7 Programexekvering 192
2.8 Kassettrutinerna 193
3. Demonstrationsprogram 196
3.1 Ljudgeneratorns styrning 196
3.2 Bithantering 200
3.3 Motorstyrning 202
3.4 Tangentbordet 203
3.5 Minnet 205
3.6 Bildminnet 207
3.7 V24-snittet 208
3.8 Effektstyrning 211
4. ABC80:s programvara 213
4.1 Benchmark 213
4.2 ABC80:s grafik 214
4.3 Flyttalsvariabler 215
4.4 Slumptalsgeneratorn 216
5. ABC-bussens programmering 218
6. Programexempel 221
6.1 Realtidsklockan i ABC80 221
6.2 Simulering 224

Appendix A Z80 Instruction Set 226
Appendix B Sjusegmentklocka 228
Appendix C Trafiksimulering 230
Sakregister 231
Litteratur 234

ABC om mikrodatorsystem

Avancerade mikrodatorsystem tränger in i vårt vardagsliv. Datorsystem som för bara något decennium sedan kostade en förmögenhet kan man i dag köpa till ett pris jämförbart med en färg-TV:s. Datorer kan alltså bli var mans egendom. Men för att kunna utnyttja datorkraften måste vi förstå hur ett datorsystem fungerar.

Målet med denna bok är dels att ge läsaren en inledande beskrivning av en dators funktion ur teknisk synvinkel och dels att beskriva uppbyggnaden av ABC-datorn, ett generellt datorsystem med många användningsområden. Fig 0.1.

Fig 0.1 ABC-datorn

1. En dator i aktion

Både den engelska termen "computer" och den gamla svenska termen "matematikmaskin" leder tankarna till att en dator är en "räknemaskin" och därmed någonting matematiskt och svårbegripligt. Detta är - tack och lov - felaktigt. En dator är en universalmaskin och dess funktion är tämligen enkel, den utför de instruktioner vi ger den. Vi kan ge datorn instruktioner för en komplett arbetsuppgift - de instruktionerna bildar ett program. Konsten att använda en dator är därmed att kunna programmera den.

För att visa hur en dator kan fungera ska vi nu studera ett datorsystem i aktion vid en vägkorsning. Vårt exempel har alltså inget med matematik att göra!

Ändamålet med en dator är ju att den ska utföra en uppgift. Det är ju uppgiften - problemets lösning - som alltid är det väsentliga. Därför ska vi börja med uppgiften och inte med datorn!

1. Trafikljusens uppgift

Med datorer kan man lösa mycket komplicerade trafikproblem. Eftersom vi här vill diskutera datorns funktion (och inte trafikteori) ska vi behandla ett mycket enkelt trafikproblem, korsningen mellan en huvudled och en lokalgata (biväg), fig 1.1.

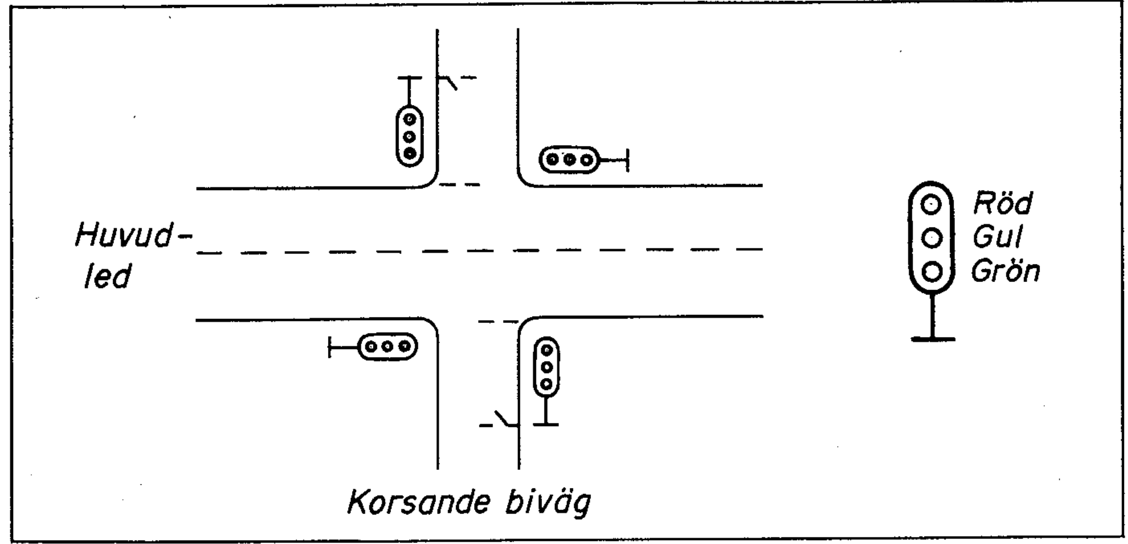


Fig 1.1 Enkel vägkorsning

En grundregel för all rationell verksamhet är att man först av allt måste bestämma sig för vad man vill åstadkomma och därefter hur man vill att uppgiften ska lösas. Politiker talar härvid ofta om mål (eller målsättning) och tekniker om specifikation (spec). Vad är alltså vårt mål för eller vår specifikation för trafikljusen i vägkorsningen i fig 1.1?

Vi specificerar här följande:

- o Normalt ska huvudleden ha grönt ljus och bivägen rött.
- o En bil som inkommer på bivägen (från norr eller söder) ska ge upphov till en ljusväxling enligt fig 1.2. Bivägens infarter till korsningen antas innehålla sensorer i vägbanan som magnetiskt avkänner när en bil anländer på bivägen .
- o Trafikljusen på huvudleden måste visa grönt minst 75 % av tiden.

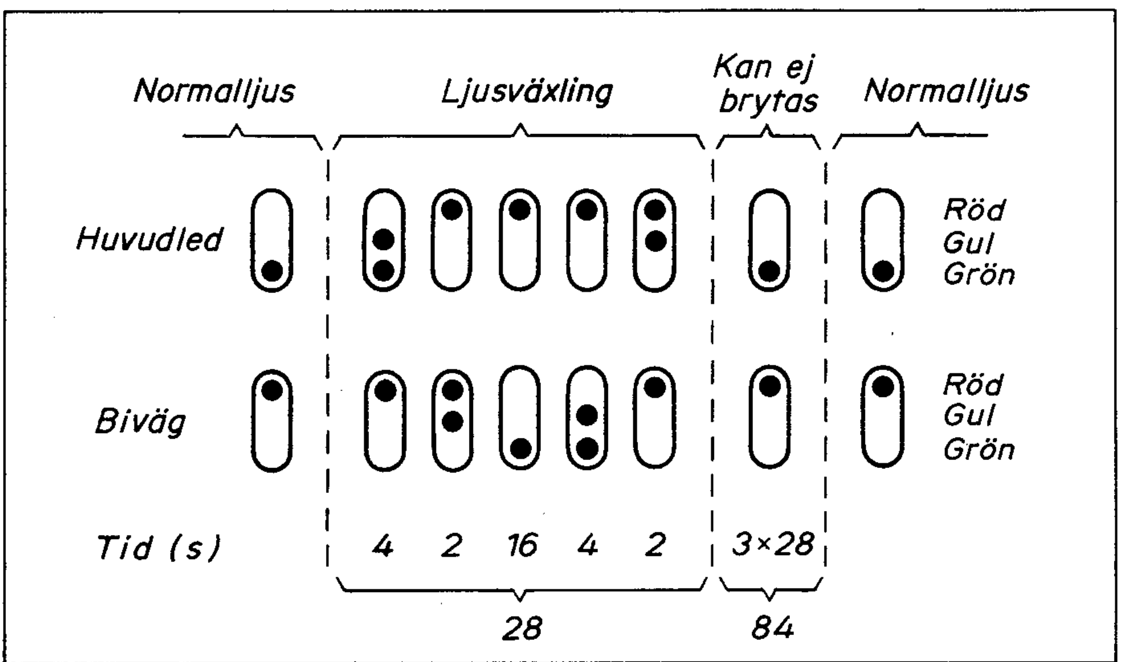


Fig 1.2 Specifikation av ljus-cykeln i vägkorsningen i fig 1.1.

Vi har nu bestämt målet och vi antar här att det kan förverkligas. (Vilket inte alltid är fallet!)

I fig 1.3 har vi skisserat ett tänkbart styrsystem för trafikljusstyrning. In till systemet kommer signaler från två sensorer i bivägen. De är så utformade att en väntande bil på någon av bivägens infarter lägger motsvarande switch i till-läge.

Ut från systemet kommer spänningar som matar 12 signallampor. Eftersom ljusen på huvudled och på biväg i vårt fall är lika i båda riktningar har de parallellkopplats parvis.

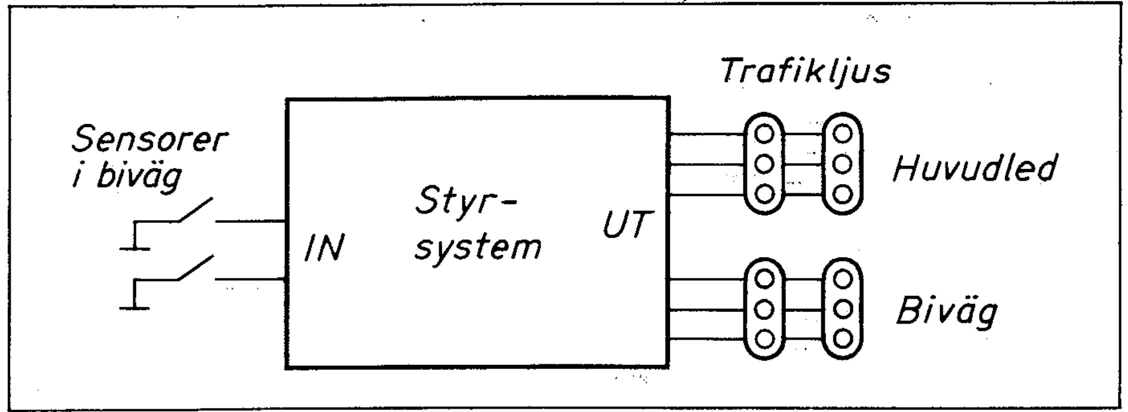


Fig 1.3 Styrsystem för trafikljus

2. Tre principlösningar

Styrsystemet i fig 1.3 kan byggas upp på tre i princip olika sätt: elektromekaniskt, med fast logik eller med dator.

Systemet kan byggas elektromekaniskt med hjälp av reläer. Kommer det damm i någon av kontakterna så som hände i Stockholm för några år sedan blir det emellertid en otrolig röra av trafiken! Det är bl a därför man nu går över till elektronisk styrning.

Systemet i fig 1.3 kan byggas upp helt elektroniskt. Tidigare byggdes styrsystem av integrerade logikkretsar. (Vi ska återkomma till vad en logikkrets är senare). Det kallas "fast logik" eftersom funktionen är inbyggd i hoplödningen av kretsarna. Vill man ändra funktionen måste alltså stora delar av systemet byggas om.

Styrsystemet i fig 1.3 kan byggas upp kring en mikrodator och då kommer funktionen att bestämmas av ett program som lagras i en integrerad krets. Fig 1.4 visar en sådan minneskrets som innehåller 8192 bitar programminne vilket motsvarar ca 2000 lagrade decimala siffror eller ca 1000 bokstäver . Om vi ändrar funktionen hos trafikljuset i vår vägkorsning behöver vi enbart byta programminnet. (Minnet i fig 1.4 kan vi "radera" genom att belysa fönstret med UV-ljus. Efter raderingen kan vi programmera minnet på nytt).

Vi ska givetvis i fortsättningen studera den intressantaste och mest flexibla systemlösningen med en dator.

3. Interface-kretsar

Digitala elektronikkretsar (som logikkretsar och datorer) arbetar binärt. En transistorswitch kan ju endast ligga i två olika tillstånd, "till" (on) eller "från" (off).

För att man enkelt ska kunna koppla ihop olika binära kretsar har man standardiserat två spänningsnivåer, hög (H) och låg (L). Insignaler till (och utsignaler från) en dator eller ett logiksystem brukar vanligen anpassas till TTL-nivåerna, H ≈ 3, 3 V och L ≈ 0, 3 V.

TTL är förkortning för transistor transistor logic och är namnet på den vanligaste serien av digitala integrerade kretsar.

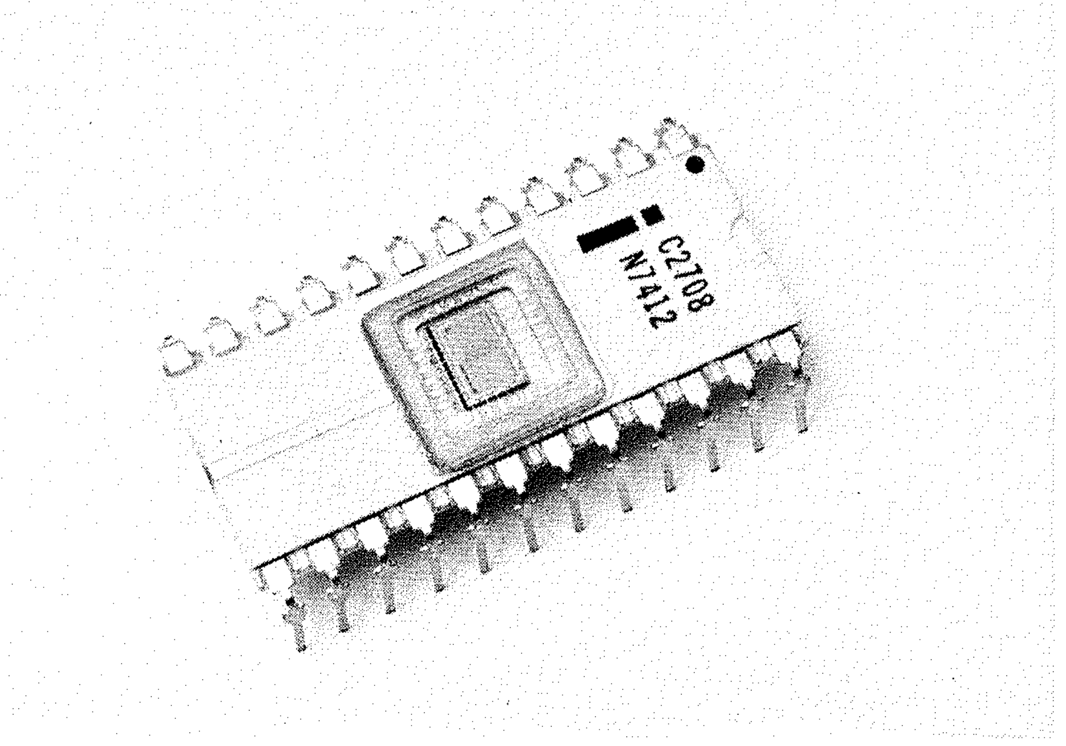


Fig 1.4 Program-minne (EPROM) innehållande 1024x8 bitar

Även drivförmågan (dvs hur många mA utgången ska kunna avge) och belastningen (dvs hur många mA ingången "konsumerar") är specificerade i TTL systemet.

För att kunna ansluta sensorswitcharna till ingången på ett logik- eller datorsystem måste tydligen sensorernas lägen omformas till spänningsnivåer (TTL-nivåer). På utgången kan inte en enkel logikkrets driva 65 W trafikljuslampor. Utgångarna måste alltså förses med drivkretsar som omformar TTL-nivåerna till lämpliga effekter för trafikljuslamporna.

I fig 1.5 har vi delat systemet i tre delar. En anpassningskrets på ingången, själva datorn (som arbetar utåt med TTL-nivåer) och en anpassningskrets på utgången. Mellan anpassningskretsarna och datorn har vi ett antal hopkopplingspunkter som bildar ett snitt (interface). I vårt fall skulle man kunna kalla detta snitt för TTL-snitt. Anpassningskretsarna kallas enligt svensk standard in-ut-kretsar (IO-devices). En vanligare benämning är interface-kretsar.

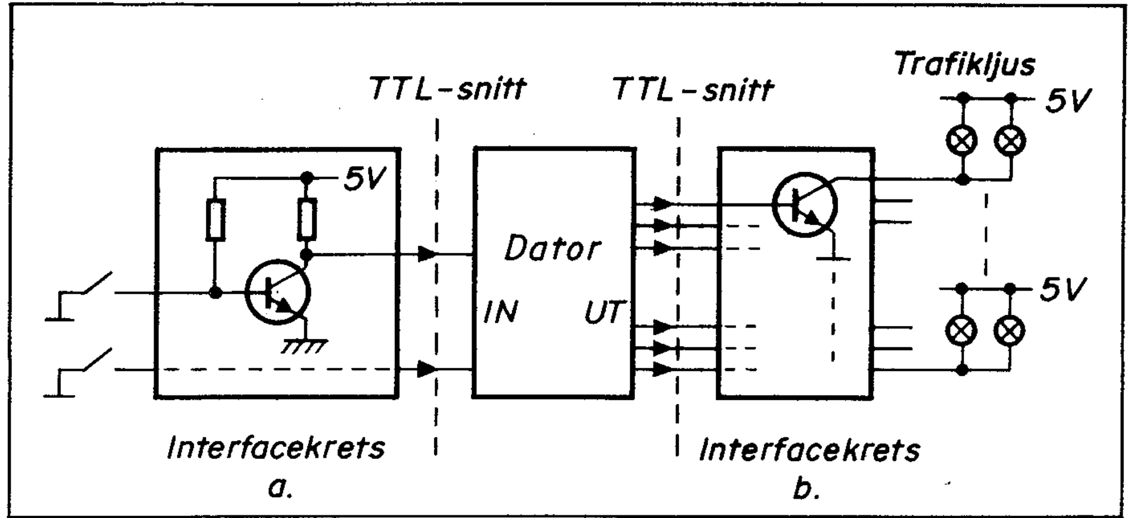


Fig 1.5 In-utkretsarnas uppbyggnad

För alla som i framtiden kommer att arbeta med elektronik (och det blir inte enbart elektronikspecialister) är det två arbetsuppgifter som blir viktigare än alla andra.

- Den ena är att bygga (eller köpa) lämpliga in-utkretsar (ofta kallade interfacekretsar)
- Den andra är att programmera datorn att utföra den önskade funktionen.

(There are two things you can do with a microcomputer. You can interface it with other circuits and you can program it).

Själva datorerna (med minnen etc) tillverkas numera integrerade i en enda kristall och till ett pris av några tior. Där lönar det sig alltså inte längre att konstruera nya.

Kanske borde vi här vända på frågeställningen och formulera den viktigaste av alla framtidsuppgifter inom elektroniken sålunda:

För att kunna utnyttja modern teknologi måste vi både kunna bygga interface-kretsar och programmera datorer. Därmed kan vi också ställa nya mål där vi aktivt styr teknikens utveckling och inte passivt låter oss manipuleras av experter.

Det låter kanske som en högtidlig programförklaring och det är just vad det är! In-utkretsar är ett utomordentligt viktigt område. Och programmeringen ska vi återkomma till senare.

Att in-utkretsar är viktiga betyder inte att de behöver vara invecklade eller svåra att förstå. I fig 1.6 ges ett förslag till uppbyggnad av in-utkretsar för våra trafikljus. Vi tänker oss här en modell av vägkorsningen där sensorerna utgörs av enkla tryckknappar och lamporna (röda, gula och gröna lysdioder) lyser tillräckligt vid 5 V och 30 mA.

3.1 Ingångskretsen

Det finns ingen vedertagen svensk terminologi för anpassningskretsar av det slag vi sett på ingång och utgång i fig 1.5. Vi kommer därför omväxlande att kalla dem in-utkretsar eller interface-kretsar (eller kortare "interface"). Om vi någon gång kallar en sådan krets i bestämd form för "interfacet", så är därmed inte avsikten att göra våld på svenska språket utan snarare att få texten att "flyta" litet bättre.

Ingångskretsen (fig 1.5a) utgörs av två enkla transistorswitchar. När vi trycker in en tryckknapp får motsvarande transistor basspänningen 0 på ingången och därmed blir transistorns utström noll (man säger att transistorn "stryps" eller ligger i "frånläge").

Normalt ligger tryckknapparna i "frånläge". Då får transistorns bas ungefär strömmen 0,5 mA (Ohms lag, 5 V och 10 kΩ) via basmotståndet på 10 kΩ. Transistorn ligger då i "till-läge" med utspänningen noll (nåväl, mindre än 0,2 V för den som är noggrann). Detta gäller så länge vi inte drar mer ström än 25 mA (dvs ca 50 ggr basströmmen) från kollektorn.

Vad vi ovan sagt om transistorns sätt att ligga "från" vid basspänningen noll och "till" när man matar in ström på basen - det är i stort sett allt man behöver veta om transistorer för att kunna förstå funktionen hos flertalet interface-kretsar.

Transistorns egenskaper att kunna växla mellan "till"- och "frånläge" är utförligt genomgånget i de flesta elektronikläroböcker. Där finns också många räkneexempel för den som vill ha fler exempel att tänka igenom!

3.2 Utgångskretsen

Utgångskretsen (fig 1.5b) består liksom på ingången av enkla transistorswitchar. Här ska vi driva två lampor med vardera 25 mA (dvs totalt 50 mA) och då måste man kontrollera att den transistortyp vi använder klarar belastningen utan att behöva mer inström från datorn än vad som är tillåten belastning (dvs specificeras av TTL-snittet). Vi antar i vårt exempel att utkretsarna på datorn kan "sänka" strömmen 2 mA vid låg utgång och då klarar vi oss med en enkel drivtransistor (som ger ca 2x25 dvs 50 mA vid bottning).

Låt oss sammafatta: In-utkretsar är viktiga kretsar och de utgör en anpassning mellan omgivningen (i vårt fall switchar och lampor) och datorn, så att datorn får signaler av specificerad typ (vanligen "TTL-kompatibla").

4. Datorns uppgift

Vi har nu sett hur en dator kan sättas in för att styra ett trafikljus. In till datorn kommer två signalledningar. Normalt ligger spänningen på dessa låg. Om en bil inkommer på någon av bivägens två infarter till vägkorsningen får motsvarande signalledning hög signal. Då reagerar datorn med utsignal som styr trafikljusen enligt den specificerade ljussekvensen.

Både in- och utsignaler är standardiserade (TTL-snitt) till storlek.

Fig 1.6 visar en modell av vår vägkorsning. Till vänster om vägkorsningen ser vi en mikrodator (Z80-MCB). Den är uppbyggd på ett kretskort. För att använda mikrodatorn behöver vi egentligen bara känna till två saker.

- o Var vi ska ansluta in- och utsignalledningarna
- o Vilket program (dvs vilka instruktioner) datorn måste ges för att utföra den önskade uppgiften.

Hur detta kan göras i vårt fall antyds i fig 1.6. Till vänster på kortet är in- och utsignaler anslutna och upptill ser vi en minneskrets (PROM) som innehåller programmet för trafikstyrningen. Men fig 1.6 säger oss inget om hur systemet fungerar.

För att ge en antydan om hur datorn kan utföra den önskade uppgiften ska vi i kapitel 2 bekanta oss med de viktigaste elektroniska kretsarna i en dator. I kapitel 3 ska vi sedan studera hur dessa kretsar samarbetar i datorn. I kapitel 4 ska vi slutligen se hur ett enkelt program för trafikstyrning kan se ut.

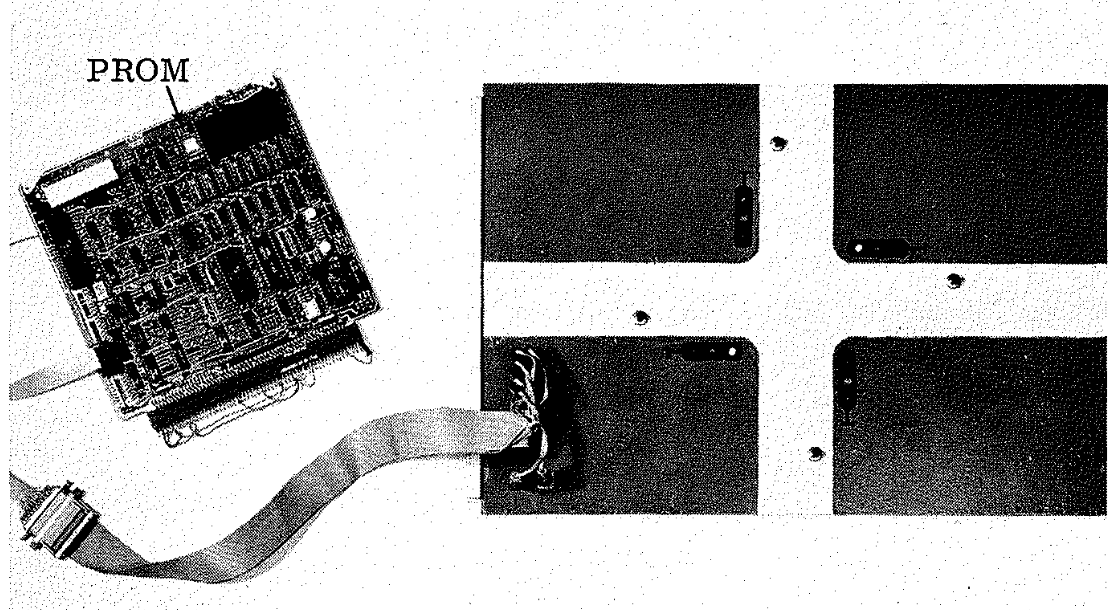


Fig 1.6 Datorstyrda trafikljus
Till höger modell av korsningen (inklusive interfacekretsar)
Till vänster mikrodatorn (Z80-MCB)
Överst programminnet (2708)

2. Elektroniska byggblock

Vi har i föregående kapitel sett exempel på in-utkretsar, dvs elektronikkretsar som förbinder en dator med dess omgivning. Själva datorn är uppbyggd av en mängd elektroniska kretsar. De är integrerade i en enda eller ett fåtal kristaller och därför kan vi inte (och behöver inte heller) studera dessa kretsar i detalj. Man får emellertid ett mycket bättre grepp om hur datorn fungerar om man lär sig förstå principen för de viktigaste elektroniska funktionsblocken. De kallas

CPU (centralenhet)
ROM (läsminne)
RAM (skriv/läsminne) och
IO-kretsar (in-utkretsar).

Vi ska börja detta kapitel med att beskriva fyra viktiga grundkretsar som utgör byggstenarna i alla digitala elektroniksystem. Med hjälp av dessa grundkretsar ska vi sedan bygga upp några viktiga funktionsenheter, bl a avkodare, läsminne (ROM), register och skriv/läsminne (RAM).

Alla kretsar vi behandlar i detta kapitel är digitala kretsar. De arbetar med binära signaler och detta innebär att kretsarna alltid tolkar en inspänning som "hög" eller "låg" (som etta eller nolla) och alltid avger "hög" eller "låg" utspänning. (Vi ska återkomma till tolkningen av signaler som binära tal i kap 3).

I detta kapitel kommer många nya termer och begrepp. Om du inte har läst elektronik tidigare så kommer en hel del att verka obegripligt första gången du läser kapitlet. Det är normalt! Stanna bara inte upp utan läs vidare. Det går bra att hoppa över de avsnitt som tar emot och återkomma till dem senare. Elektronik lär man sig successivt. Första steget är att vänja sig vid de många nya orden!

1. Fyra grundkretsar

Vi ska i detta avsnitt studera fyra grundläggande elektroniska kretsar. Eftersom dessa kretsar utgör byggstenarna i alla digitala system är de väl värda en närmare bekantskap. Vi ska här enbart ägna oss åt principer, detaljstudier kan göras i olika läroböcker om digitala kretsar.

En inverterare (inverter) kan se ut som interface-kretsen i fig 1.5a. Fig 2.1a visar en vanligare variant. Vi ska inte här ge en detaljbeskrivning, det intressanta är kretsens funktion: Hög inspänning ger låg utspänning och vice versa. Man säger att kretsen inverterar insignalen.

I fig 2.1b ser vi två vanliga grafiska symboler för inverteraren. IEC-symbolen är internationell standard men USA-symbolen är den vanligast förekommande. Vi använder i fortsättningen enbart USA-symboler. Funktionen kan även beskrivas med bokstavssymboler. Om inspänningen kallas A betecknar man utspänningen med $\overline{A}$ ("A-streck") eller A* ("A-stjärna").

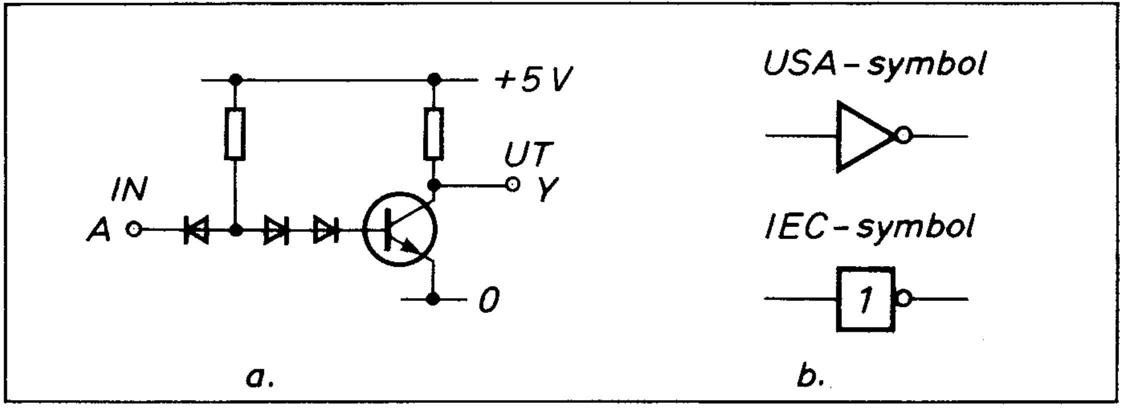


Fig 2.1 Inverterare
a. Kretsschema
b. Två vanliga grafiska symboler

I fig 2.2a jämförs symbolerna för en buffert och en inverterare. Den enda skillnaden i de grafiska symbolerna är inverterarens ring. Ringen är en vanlig symbol för invertering.

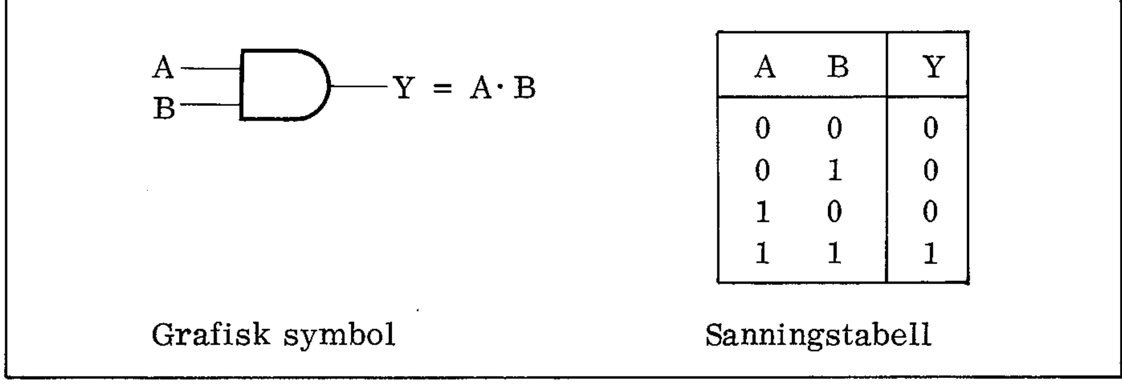
En buffert ändrar inte signalens värde men den tål stor belastning. Den inkopplas därför ofta mellan en IC-krets (som ej tål alltför stor belastning) och efterföljande kretsar. Buffertens och inverterarens funktion kan beskrivas med "sanningstabeller" som anger sambandet mellan in- och utsignal, fig 2.2b.

	a. Grafisk symbol	b. Sanningstabell	c. Beteckning
Buffert buffer, driver	A → Y	A Y 0 0 1 1	Y = A
Inverterare inverter	anger invertering A → Y	A Y 0 1 1 0	$Y = \overline{A}$ eller $Y = A^*$

Fig 2.2 Buffert och inverterare

1.2 Grinden

Fig 2.3 visar en AND-grind med två ingångar samt tillhörande sanningstabell. Det är endast en kombination av inspänningar (etta på både A och B) som ger en etta på utgången. Man brukar kalla kretsar av detta slag (dvs som ger bestämda utsignaler för givna kombinationer av inspänningar) för kombinationskretsar. AND-funktionen brukar betecknas med punkt. $Y = A \cdot B$



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Fig 2.3 AND-grinden och dess sanningstabell

Fig 2.4 visar hur vi kan använda AND-kretsen som en signalgrind. En av ingångarna använder vi som signalingång och den andra som styringång. Enbart då styringången ges värdet ett kommer grinden att släppa igenom signalen. Det är denna användning som givit upphov till benämningen grind (gate).

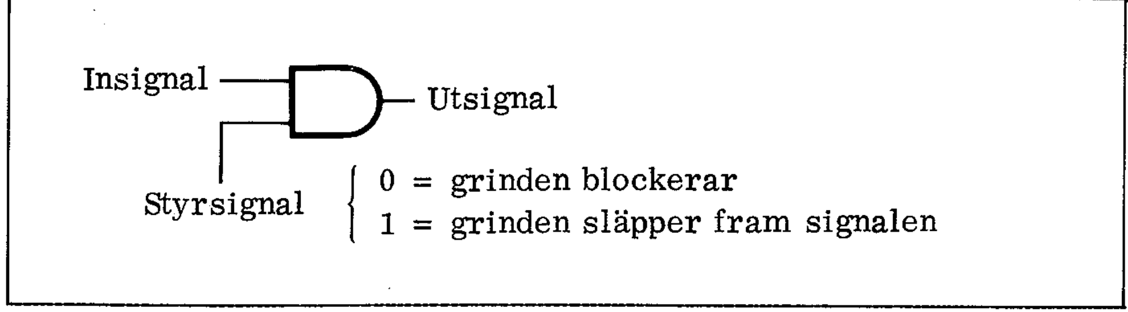


Fig 2.4 AND-grind använd som signalgrind

En AND-grind kan ha många ingångar (men bara en utgång). Fig 2.5 visar en fyra ingångars AND-grind. Utgången får värdet ett endast för en enda insignalkombination - när samtliga ingångar antar värdet ett. AND-grinden fungerar alltså som ett kombinationslås med "kombinationen" fyra ettor.

A
B
C
D

$Y = A \cdot B \cdot C \cdot D$

Fig 2.5 AND-grind med fyra ingångar

Av praktiska skäl kombinerar man ofta en AND-grind med en inverterare. Man får då en NAND-grind. Ett exempel på kretsschemat för en NAND-grind framgår av fig 2.6. Det är den vanligaste av alla grindar, TTL-grinden 7400. TTL betyder "transistor transistor logik" och 7400 är typgrinden i den s k 74-serien. Kapseln i fig 2.6a kallas DIL-kapsel (DIL = dual in line) eftersom uttagen sitter i två rader.

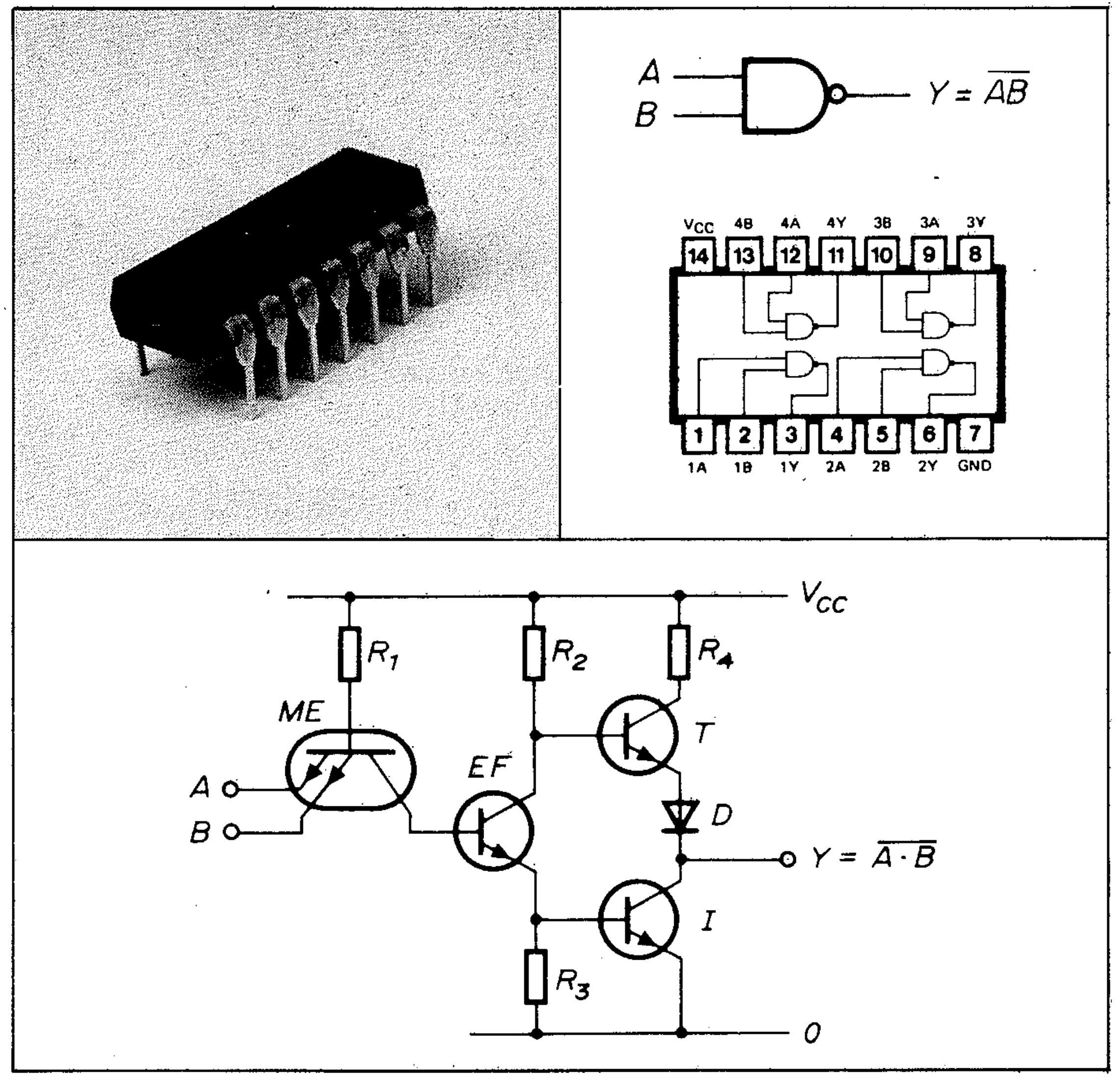


Fig 2.6 NAND-grinden 7400
a. DIL-kapseln
b. Grafisk symbol och uttagsplacering
c. Kretsschema

ME = multiemittertransistor
EF = emitterföljare
I = inverterare
T = pull up transistor
V_{CC} = 5 V
R_1 = 4 kΩ R_3 = 1 kΩ
R_2 = 1,6 kΩ R_4 = 130 Ω

1.3 Tristate-kretsen

Flera grindingångar kan kopplas in till en och samma signalledning. Inspänningen på samtliga dessa grindingångar kommer då att bestämmas av den inkommande signalen.

I många fall vill man styra en enda utgående ledning från flera grindar - men det går inte lika enkelt.

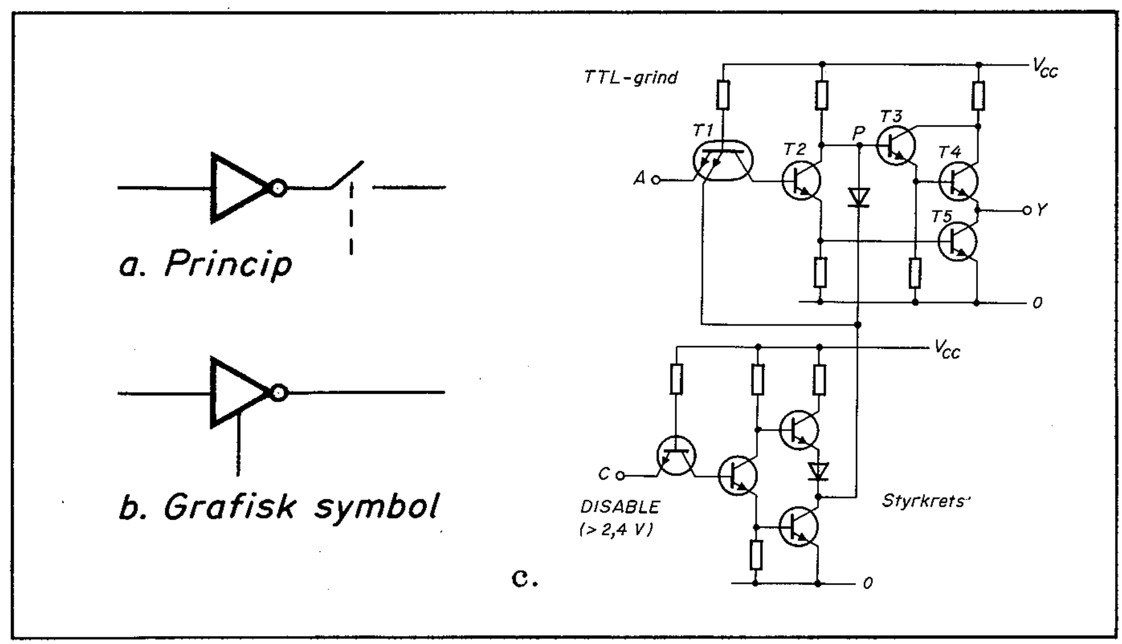


Fig 2.7 Tristate-grind

Normalt får grindar inte hopkopplas på utgången. Om en av grindarna vill lägga en etta på utgående ledning medan en annan vill lägga en nolla på samma ledning så kan systemet inte längre fungera. För att undvika detta har man infört grindar med s k tristate-utgång. Det betyder att utgången kan frånkopplas (bli "högimpediv" eller "flytande"). Frånkopplingen av utgången sker med hjälp av en extra styringång (ofta kallad enable eller tristate). Fig 2.7a visar principen och fig 2.7b den grafiska symbolen. För den som är intresserad av kretsdetaljer återges i fig 2.7c ett exempel på en integrerad tristate-krets uppbyggd i s k TTL-teknik.

Tristate-principen kan givetvis tillämpas på alla typer av grindar. "Tristate" innebär ju endast att en utgång "frikopplas" och därmed inte påverkar den på utgången anslutna signalledningen.

1.4 Vippan

En vippa består i princip av två korskopplade grindar. Fig 2.8. På grund av korskopplingen låser grindarna varandra till ett av två möjliga lägen (Q hög eller Q* hög). Vippan kan ställas om (triggas) mellan dessa lägen med hjälp av olika insignaler. I fig 2.8 är dessa betecknade med S och R. Normalt ska båda dessa ingångar ligga höga. Om $\overline{S}$-ingången triggas - dvs om den ställs låg ett ögonblick - kommer Q-utgången att bli hög (vippan står i läge 1). Om $\overline{R}$-ingången triggas blir Q-utgången låg (vippan står i läge 0).

En vippa är ett minneselement. Vi kan inom loppet av några nanosekunder (1 ns = 10^{-9} s) ställa om vippan från noll till ett eller vice versa. När man bygger vippor i integrerad form får man rum med 16.000 vippor på en kiselkristall av ca 30 mm^2 yta. Det är inte undra på att vippor blivit viktiga byggbitar i all elektronisk utrustning!

När man bygger samman ett stort system är det högst väsentligt att alla kretsar ställs om (klockas) samtidigt. Annars kan det uppträda en rad fenomen där kretsar på ett okontrollerbart sätt påverkar varandra (racing). Detta är orsaken till att varje dator har en "klocka". Kretsarna i datorn förbereds för omställning av olika signaler i systemet, men själva omställningen sker först när klockpulsen kommer.

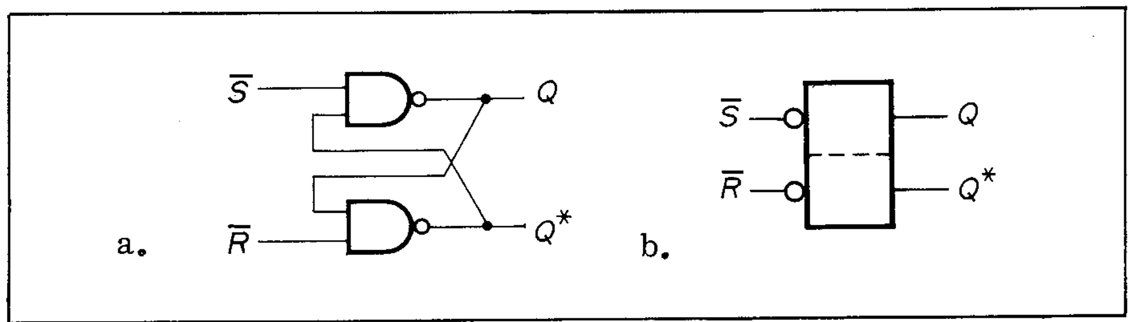


Fig 2.8 Vippa uppbyggd av två NAND-grindar
a. Schema
b. Symbol

En vippa kan göras "klockad" med hjälp av ett par signalgrindar. Fig 2.9 visar vår tidigare RS-vippa men här med signalgrindar på båda ingångarna. S- och R-ingångarna ska i detta fall normalt ligga låga. När man vill ettställa vippan läggs först S-ingången hög som en förberedelse för den efterföljande triggningen - som sker när den positiva klockpulsen anländer.

SR-vippan har fått sitt namn av ingångarna, S = set (dvs ettställ) och R = reset (noll-ställ).

När man köper en integrerad vippa i plastkapsel får man som regel en avsevärt mer komplicerad krets än den vi studerat i fig 2.9. (Det är typiskt för den integrerade elektroniken, det är ju kristallens storlek och plastkapseln som kostar pengar helt oavsett hur många transistorer man har integrerat i kristallen).

Den vanligaste av alla vippor kallas MS-vippan (M = mästare, S = slav) och den består i själva verket av två hoppbyggda vippor, en på ingången (mästaren) och en på utgången (slaven). Man kan på detta sätt göra in- och utsignaler oberoende av varandra och det är, som vi ska se senare, en viktig egenskap.

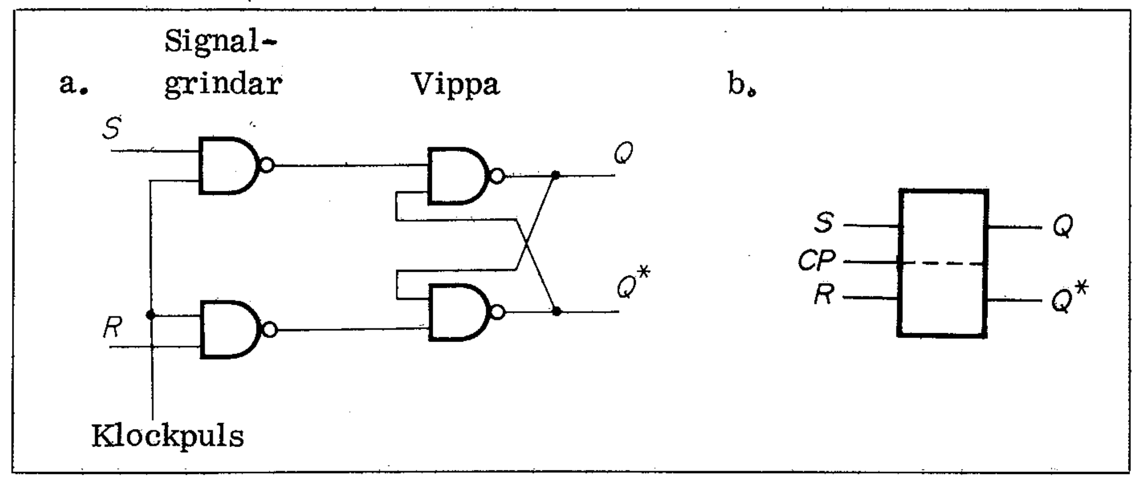


Fig 2.9 Klockad SR-vippa
a. Schema
b. Symbol

Fig 2.10c visar ett exempel på en annan vanlig typ av vippa, D-vippan. Den triggas av klockans positiva flank och lagrar då signalen på dataingången (D-ingången).

Eftersom man får plats med uttagen till två D-vippor i en standardkapsel (med 16 ben) får man här två vippor till samma pris som en ensam.

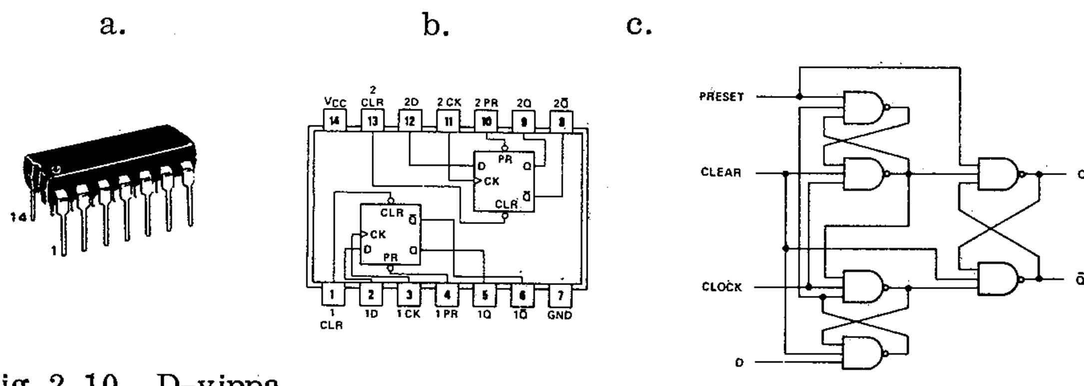


Fig 2.10 D-vippa
a. DIL-kapsel
b. Uttagsplacering
c. Kretsschema

Sammanfattning:

Vi har nu bekantat oss med de fyra byggstenarna för all digital elektronik:

- o inverteraren
- o grinden
- o tristate-kretsen
- o vippan

Tristate-kretsen ska kanske inte betraktas som en grundkrets i samma mening som de tre övriga. Tristate-kretsen är snarare en "frikopplingsanordning" som kan byggas in i alla typer av kretsar. Tristate-principen är emellertid så väsentlig att vi här har tagit med den som en av de fyra "grundfunktionerna".

Fig 2.11 sammanfattar de fyra byggstenarna. Vi har här medtagit en NAND-grind som grindexempel och en tristate-NAND-grind som exempel på tristate-funktion. Som exempel på vippor visas en MS-vippa. Klockingången ska här normalt ligga hög och klockning sker på låg puls (till skillnad från fig 2.9). Detta framgår av inverteringsringen på klockingången.

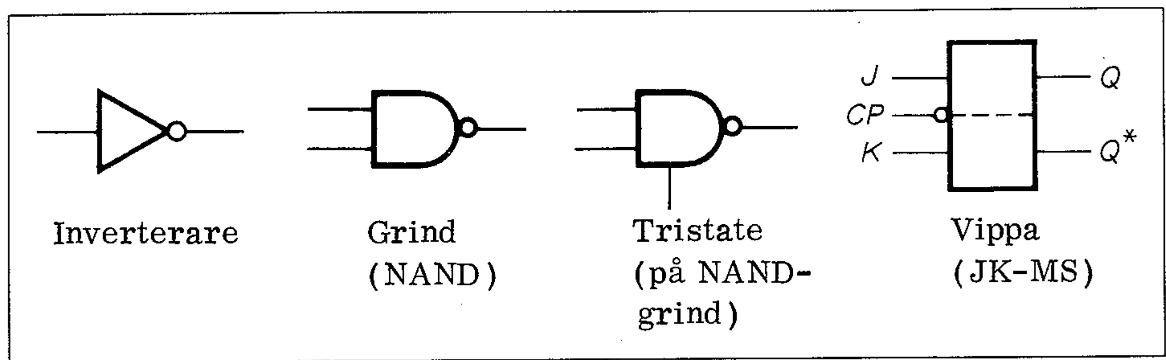


Fig 2.11 De fyra digitala "byggstenarna"

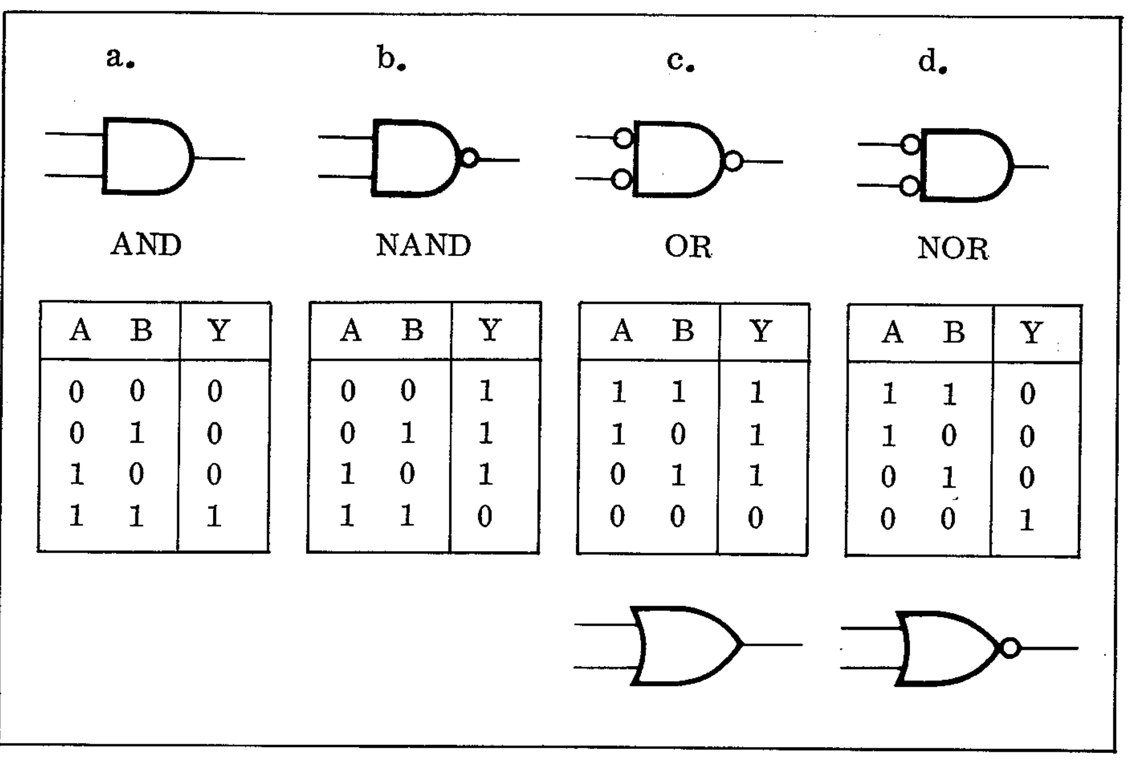
2. Kombinationskretsar

En kombinationskrets har i stort samma egenskaper som ett kombinationslås. Det krävs rätt kombination på ingången för att man ska få önskat resultat på utgången.

Grindar är typiska kombinationskretsar. AND-grinden kräver ju som vi sett ettor på samtliga ingångar för att ge en etta ut. Vi ska nedan visa några vanliga kopplingar med grindar. Vi ska med hjälp av grindar bygga upp en avkodare och slutligen ett läsminne (ROM= read only memory).

2.1 Grindnät

Med AND-grindar och inverterare kan man bygga upp alla tänkbara typer av kombinationsfunktioner. Fig 2.12 visar några funktioner som ofta används och som därför begåvats med egna namn.



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Y
1	1	1
1	0	1
0	1	1
0	0	0

A	B	Y
1	1	0
1	0	0
0	1	0
0	0	1

Fig 2.12 Fyra vanliga logikfunktioner

Med en inverterare på utgången av en AND-grind får vi den från fig 2.6 bekanta NAND-grinden. NAND-grindens funktion får vi genom att byta nollor mot ettor (och vice versa) på utgången i AND-funktionen, dvs i Y-kolumnen.

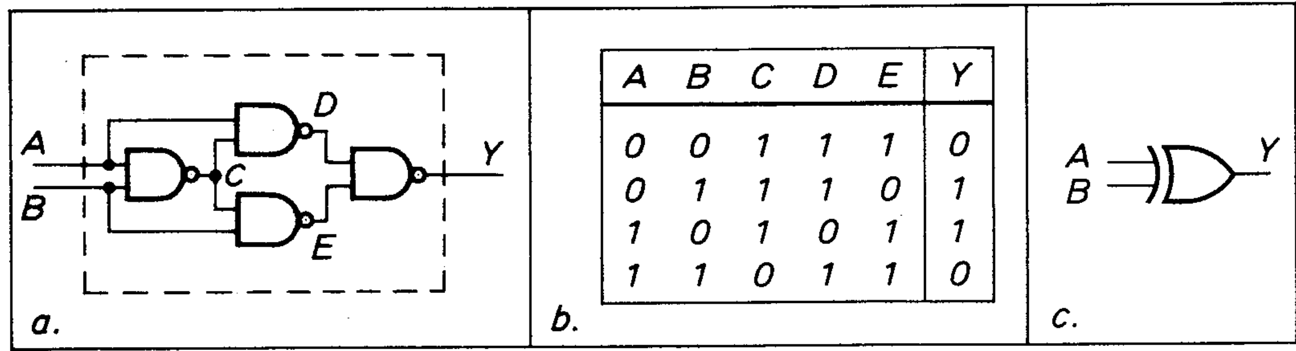
Sätter vi (enligt fig 2.12c) in inverterare på NAND-grindens ingångar och byter nollor och ettor i NAND-funktionens A- och B-kolumn får vi en OR-grind. Den ger en etta på utgången så snart A eller B eller båda har en etta. OR-grinden har fått egen grafisk symbol som utritats under sanningstabellen.

Två seriekopplade inverterare ger tillbaks den ursprungliga signalen. Att sätta en inverterare på en OR-grind så som framgår av NOR-symbolen (underst i fig 2.12d) eller ta bort utgångsinverteraren från kretsen i fig 2.12c överst ger alltså samma resultat. Vi får den mycket vanliga NOR-kretsen.

AND, NAND, OR och NOR är mycket vanliga grindar i digitala system. De kan köpas som komponenter i DIL-kapslar av samma typ som visats i fig 2.6. AND och OR kallas OCH och ELLER på svenska NAND och NOR kallas mera sällan NOCH och NELLER. Det blir komplicerat att införa svenska termer för alla kretsar och därför använder man i praktiken ofta de engelska originaltermerna.

Fig 2.13 ger exempel på en annan mycket viktig grindfunktion. I fig 2.13a är den uppbyggd av fyra NAND-grindar. Funktionen framgår kanske inte omedelbart av fig 2.13a. Det är emellertid enkelt att ta reda på funktionssambandet om man går systematiskt tillväga.

För varje kombination av insignaler (A och B) kan vi successivt bestämma signalerna i noderna allt längre mot höger (först C och sedan D och E samt slutligen utsignalen Y). Med hjälp av NAND-funktionen i fig 2.12b kan vi alltså successivt fylla i kolumnerna C, D, E och Y i fig 2.13b. Det är en nyttig övning att på egen hand härleda funktionen hos kretsen i fig 2.13a. Gör ett försök!



A	B	C	D	E	Y
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	0

Fig 2.13 XOR-funktionen
a. Uppbyggd av NAND-grindar
b. Sanningstabell
c. Grafisk symbol

Funktionen i fig 2.13b kallas "exclusive or" eller kortare XOR. Den kan användas exempelvis för att jämföra signaler på två inkommande ledningar. Om signalerna är lika visar utgången noll. Kretsen har vidsträckt användning och har därför fått en egen symbol, fig 2.13c.

Med hjälp av enkla grindar kan man på detta sätt fortsätta och bygga upp alltmer komplexa nät. Det finns väl utvecklade teoretiska metoder (bl a Boolesk algebra och Karnaugh-diagram) för att behandla grindnät.

Det finns, som nämnts i anslutning till fig 2.1, två symbolserier: IEC-symbolerna (internationell standard från International Electrotechnical Commission) och USA-symbolerna. Fig 2.14 jämför några vanliga symboler från USA och IEC.

Eftersom USA-symbolerna fortfarande används i praktiskt taget alla datablad och i 90 % av den engelskspråkiga facklitteraturen så använder vi dem även här. De flesta europeiska elektronikföretag använder numera internt IEC-symbolerna och ska man arbeta med digitala kretsar måste man vänja sig vid båda symbolserierna.

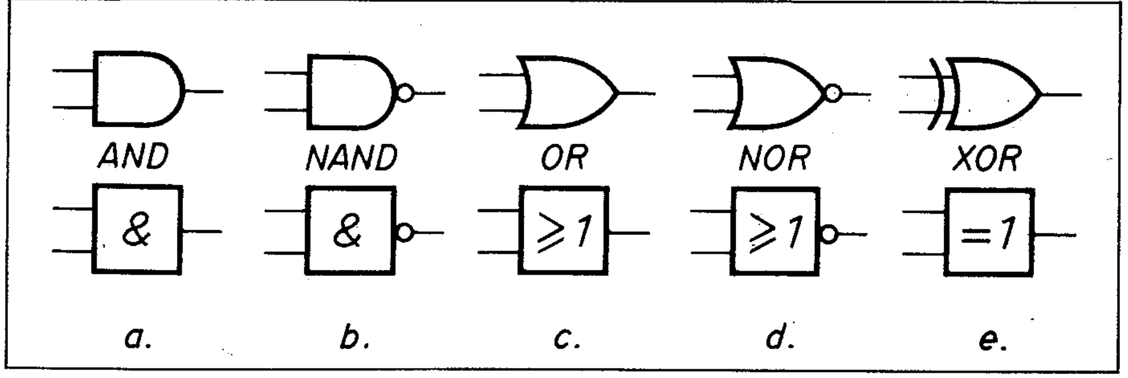


Fig 2.14 USA- och IEC-symboler för de digitala grundkretsarna

2.2 Avkodare

En AND-grind ger utsignal för en insignalkombination som enbart består av ettor. Med hjälp av inverterare på lämpliga ingångar kan man emellertid få grinden att reagera för en godtycklig insignalkombination. I fig 2.15 har vi byggt upp ett grindnät av åtta AND-grindar, vardera med tre ingångar. Genom att placera inverterare

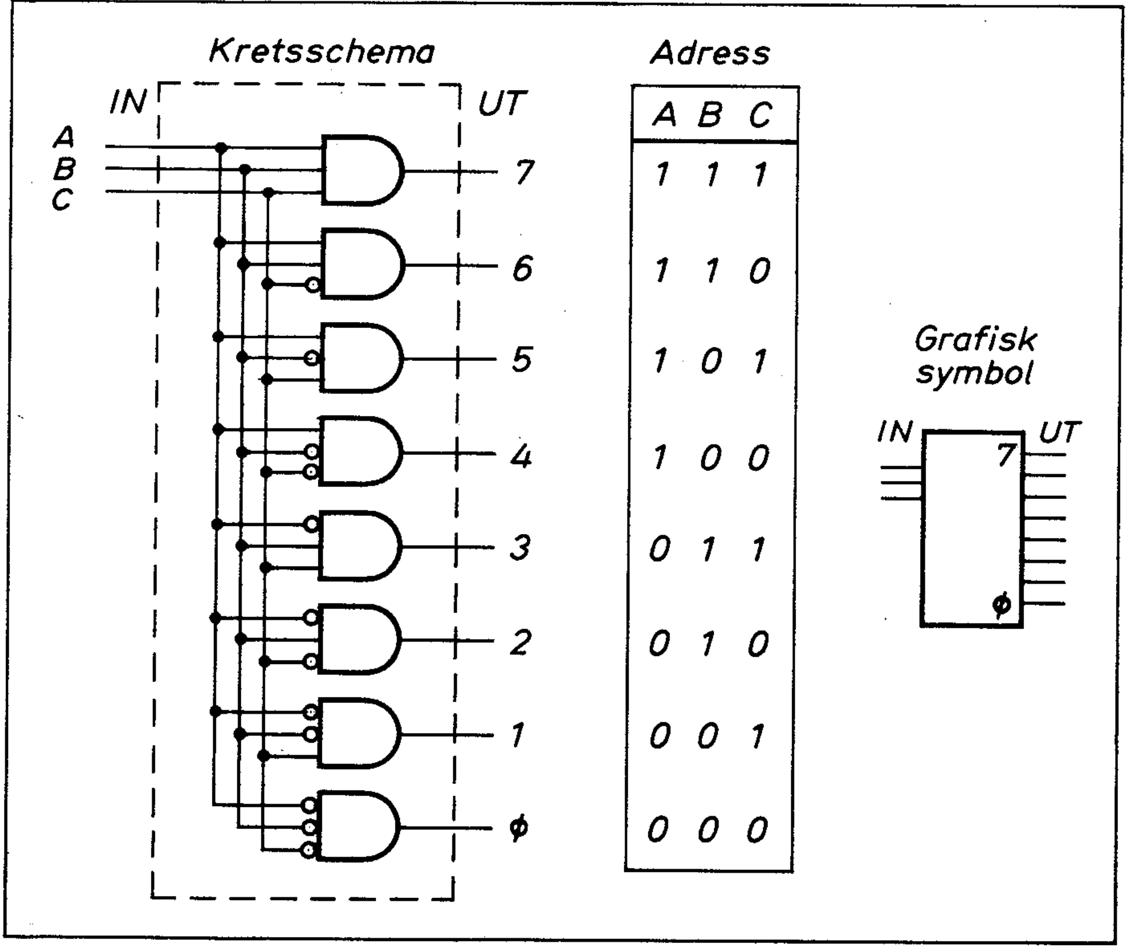


Fig 2.15 "En av åtta"-avkodare

(ringar i fig 2.15) så som figuren visar kommer enbart en grind att reagera för var och en av de åtta möjliga insignalkombinationerna.

Den med streckmarkering inramade kretsen i fig 2.15 brukar kallas en avkodare. En på ingången pålagd signalkombination "avkodas" eller "dechiffreras" och ger motsvarande utsignal. Ingångarna (ledningarna AØ, A1 och A2) brukar kallas adressingångar. En viss insignalkombination, dvs en viss adress, pekar ju ut en bestämd utgång.

Avkodaren är ett väsentligt byggblock i alla datorer. Den gör det möjligt att med ett begränsat antal ledningar peka ut många adresser. Vår avkodare i fig 2.15 har tre adressledningar och kan peka ut $2 \cdot 2 \cdot 2 = 2^3 = 8$ olika adresser. Många mikrodatorer arbetar med 16 adressledningar och kan därmed peka ut $2^{16} = 65536$ olika adresser. Tack vare avkodaren kan vi alltså hålla reda på stora datamängder med jämförelsevis få ledningar.

2.3 Läsminnen

I digitala system hanterar man data i form av "ord". Varje ord består av ett bestämt antal bitar. De flesta mikrodatorer arbetar exempelvis med åtta bitars ordlängd. 8 bitar kallas 1 byte (engelskt uttal!).

Läsminnet är en krets där man permanent kan lagra ett antal ord. Engelska termen för läsminne är ROM (=read only memory).

Fig 2.16 visar principen för ett ROM. Till vänster ser vi den tidigare i fig 2.15 behandlade avkodaren, men nu med inverterare på utgångarna. Adresserad utledning ligger alltså låg (≈ 0 V) medan samtliga övriga ledningar ligger höga (≈ 5 V).

Avkodarens utgångar är utdragna till ett linjemönster och vi kallar dessa ledningar "ordledningar". Ovanför ordledningarna (och alltså isolerade från dem) går fyra lodräta "bitledningar". Bitledningarna är via motstånd på 10 kΩ anslutna till +5 V matningsspänning. Eftersom bitledningarna är isolerade från ordledningarna kommer samtliga bitledningar att ligga höga. Bitledningarna fungerar som datautgång. Data ut kommer alltså att vara fyra ettor. Eftersom vi i fig 2.16 har fyra bitledningar blir ordlängden i detta fall fyra bitar.

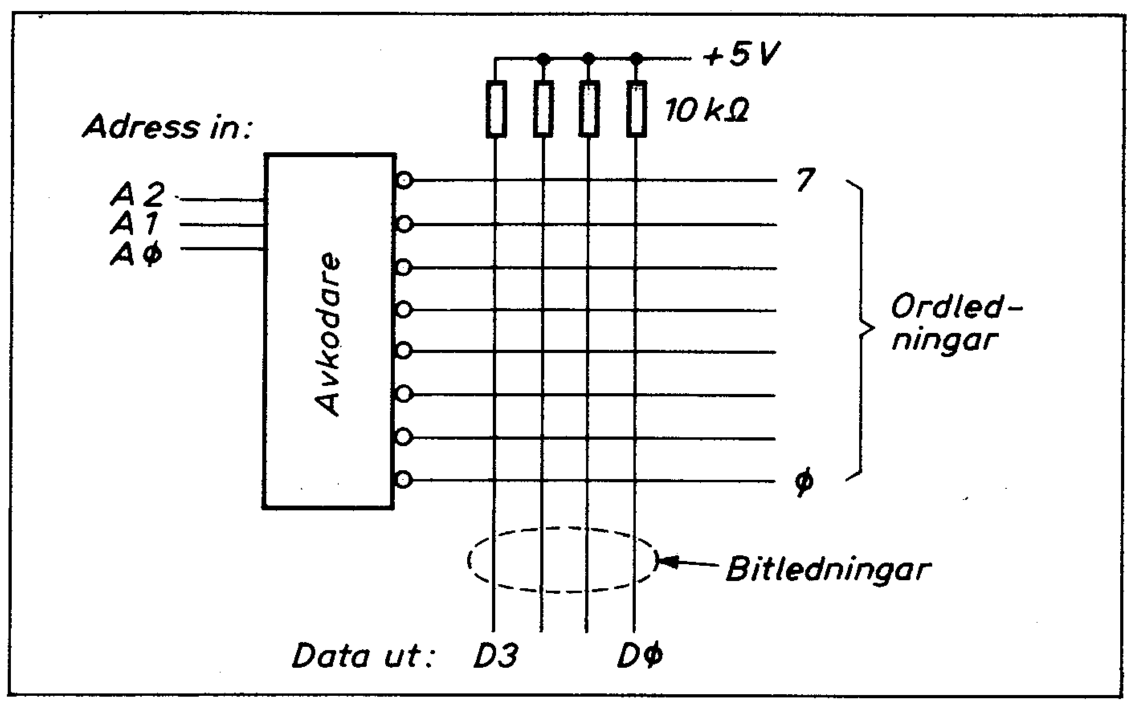


Fig 2.16 Läsminne som ännu ej programmerats

Oavsett vilken adress vi pålägger adressingångarna i fig 2.16 kommer data ut att visa fyra ettor. Bitledningar och ordledningar är ju isolerade från varandra. Men nu kommer finessen - dvs programmeringen av de åtta ord som vi permanent kan lagra i vårt läsminne.

Antag att vi vill ha det binära ordet 0101 lagrat på binära adressen 111. Lägger vi på adress 111 på adressingångarna kommer översta ordledningen i fig 2.16 att gå låg. För att få ordet 0101 på datautgångarna måste vi "sänka" spänningarna på bitledningarna D3 och D1. Det kan vi göra om dioder inkopplas enligt fig 2.17.

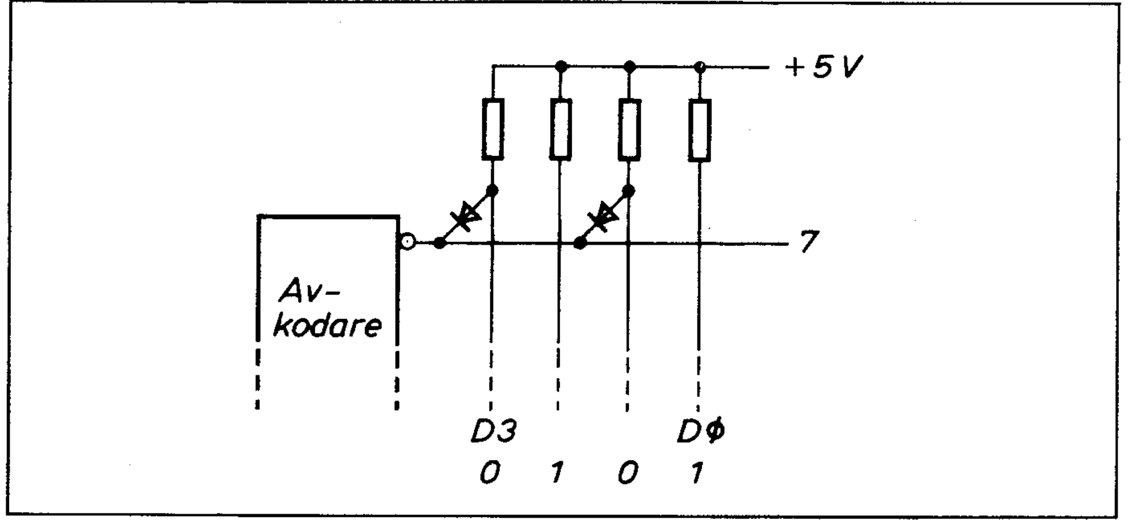


Fig 2.17 Ord nr 7 programmerat med innehållet 0101

Genom att löda in dessa dioder kan vi alltså programmera in ordet 0101 på adress 111 i läsminnet. Dioderna är nödvändiga för att ordledningarna inte ska påverka varandra. När ordledning nr 7 ej är adresserad ligger den hög och då spärrar dess dioder. Andra ordledningar kan därmed ta kommandot över bitledningarna.

Fig 2.18a visar ett programmerat läsminne. Vi har här ritat cirklar runt de krysspunkter som försetts med dioder.

Fig 2.18b visar hur data ligger lagrade på de åtta adresserna.

Fig 2.18c visar slutligen en minnestabell och här har vi helt släppt kontakten med de elektroniska kretsarna. Fig 2.18 visar det nära sambandet mellan de elektroniska kretsarna och motsvarande minnestabell.

Minnet i fig 2.18 innehåller 8x4 = 32 bitar och vi har programmerat det genom inlödning av dioder. En vanlig typ av läsminnen innehåller 1024x8 = 8192 bitar. Det programmeras vid tillverkningen genom att ett diodmönster (i praktiken ett transistormönster) överförs till kristallen via en fotografisk mask. Ett sådant ROM kallas "maskprogrammerat". Fig 2.19a.

Dioderna i fig 2.18 kan ersättas med fälteffekttransistorer vars styrelektroder är väl isolerade. Dessa styren kan laddas med hjälp av spänningspulser och urladdas genom belysning med ultraviolett ljus. Ett sådant läsminne kallas EPROM (=erasable programmable read only memory). Fig 2.19b visar ett vanligt EPROM (typ 2708) som innehåller 1024x8 bitar.

En typ av läsminnen tillverkas med samtliga dioder (eller transistorer) på plats. I fig 2.18 skulle detta motsvaras av ringar i samtliga krysspunkter.

Genom att adressera ord efter ord och samtidigt köra strömstötar genom bitledningarna kan man smälta av de smala diodledningarna (fusable link). Ett sådant minne kan alltså programmeras men inte raderas och kallas därför PROM (programmable read only memory). Fig 2.19c visar ett exempel.

+5V

Adress in

A 2
A 1
A 0

Avkodare

7

Ordled-
ningar

ϕ

Bitledningar

Data ut: D3 Dϕ

a.

Adress in

A 2
A 1
A 0

Avkodare

7	0 1 0 1
	0 1 1 1
	0 1 0 1
	0 1 0 0
	0 1 1 0
	0 1 1 0
	0 1 1 0
ϕ	1 0 1 0

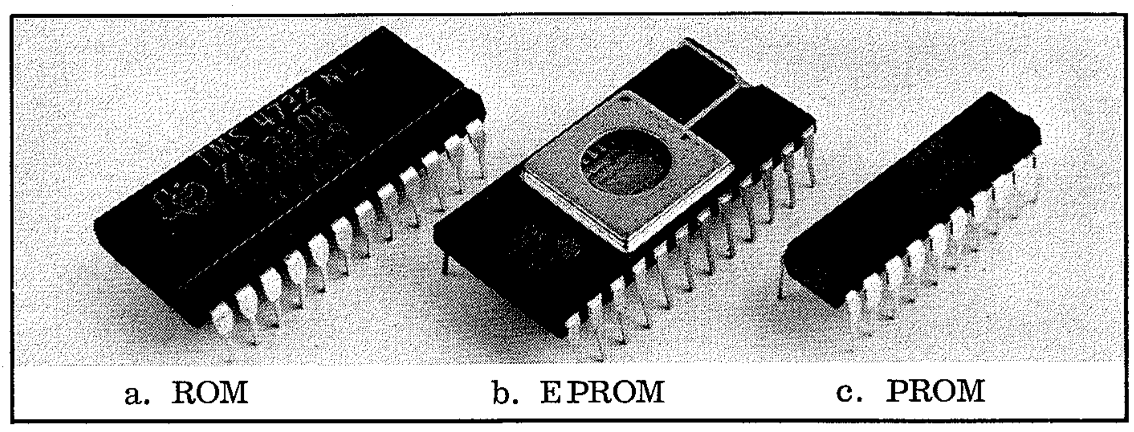
Data ut

b.

Adress	Data
1 1 1	0 1 0 1
1 1 0	0 1 1 1
1 0 1	0 1 0 1
1 0 0	0 1 0 0
0 1 1	0 1 1 0
0 1 0	0 1 1 0
0 0 1	0 1 1 0
0 0 0	1 0 1 0

c.

Fig 2.18 Programmerat läsminne
a. Schema
b. Princip
c. Minnestabell



a. ROM b. EPROM c. PROM

Fig 2.19 Tre vanliga typer av läsminnen

Innan vi lämnar kombinationskretsarna ska vi kort omnämna ytterligare en viktig krets som kallas multiplexer, eller kortare MUX. Principen för en MUX framgår av fig 2.20a.

En multiplexer är en "omkopplare" med vars hjälp man kan "välja" en av ingångarna A eller B. Den kan byggas upp med två AND-grindar och en OR-grind på det sätt som visas i fig 2.20b. När styrsignalen A/B ligger hög är tydligen ingång A inkopplad och när styrsignalen är låg inkopplas ingång B.

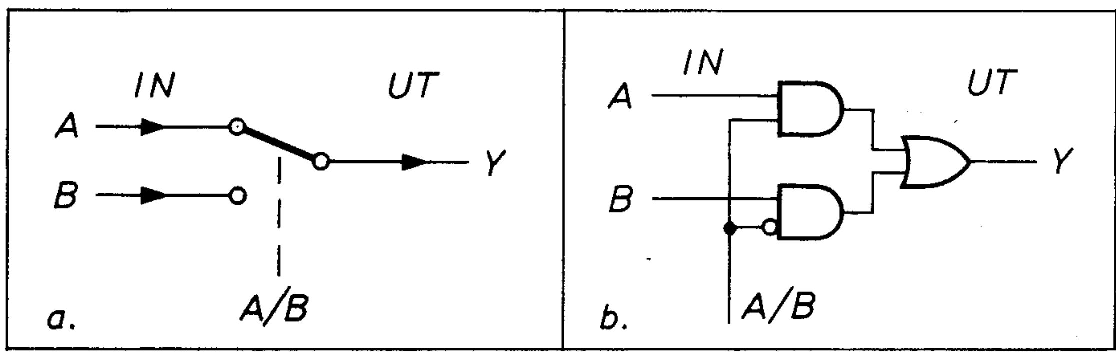


Fig 2.20 Principen för en multiplexer

Fig 2.21 visar en praktisk MUX. Den innehåller 4 st kretsar liknande den i fig 3.20. Med två MUX:ar av detta slag kan man tydligen välja en av två inkommande åtta bitars bussar. Vi ska se en sådan användning senare i mikrodatorn ABC80.

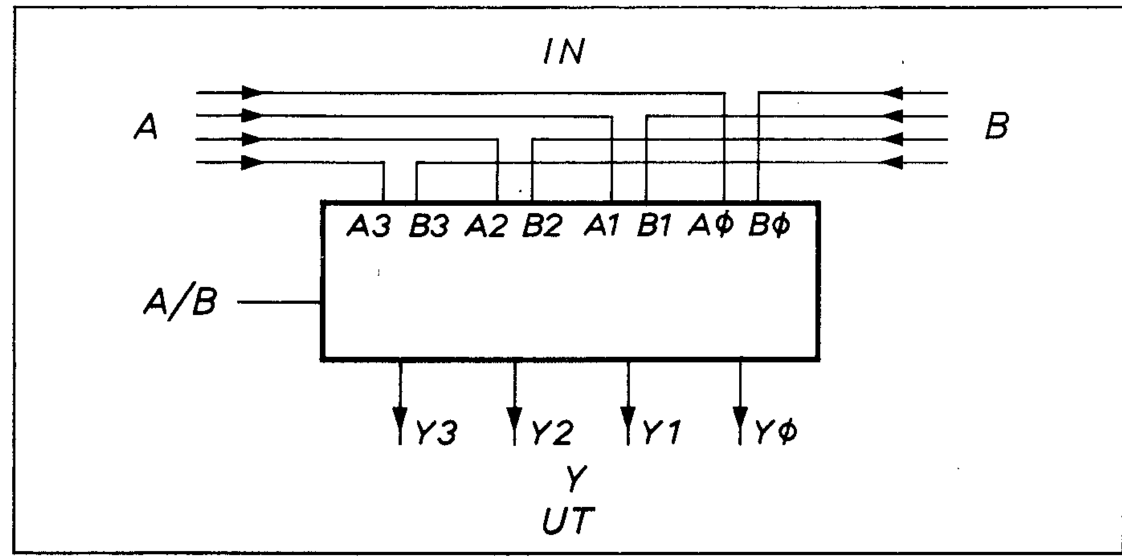


Fig 2.21 Fyra bitars multiplexer (74LS257)

3. Sekvenskretsar

Kretsar som innehåller minneselement (exempelvis vippor) brukar kallas sekvenskretsar.

De flesta mikrodatorer arbetar med åtta bitars ordlängd. Det krävs därför åtta vippor för att lagra ett ord. Åtta vippor brukar därför byggas ihop till ett register och användas för mellanlagring av ord. När man ska lagra stora datamängder kan man använda en stor mängd vippor som integrerats i en och samma kristall. Skriv/läs-minne kallas ofta oegentligt för RAM (random access memory). Termen "random access" betyder att man kan nå alla minnescellerna lika snabbt oberoende av var de är placerade. Även vårt tidigare ROM är därför i princip ett "random access"-minne.

3.1 Register

Fig 2.22 visar principen för ett fyra bitars register. Registret har fyra dataingångar (D-ingångar), en nollställningsingång (RESET) samt fyra datautgångar. Register av denna typ kallas ofta för latch (datalatch) och de ingående vipporna för D-vippor. Vipporna är uppbyggda på liknande sätt som i fig 2.10.

När registret klockas (negativ puls på klockingången) kommer de insignalvärden som ligger på dataingångarna att läsas in och lagras

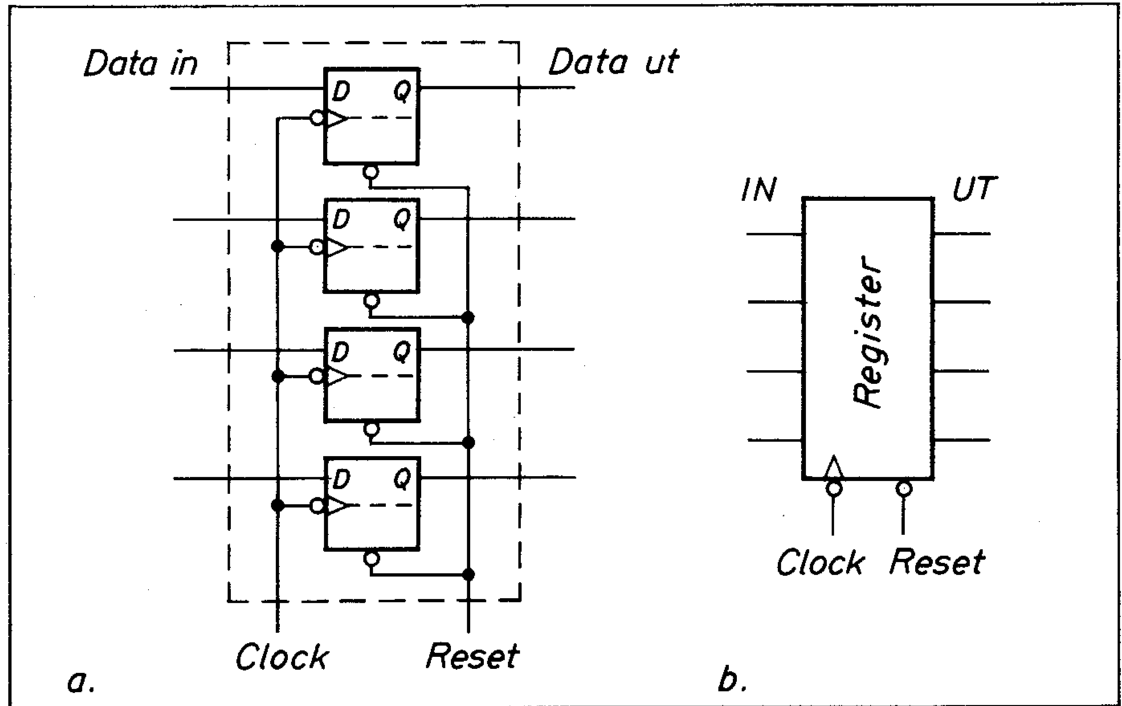


Fig 2.22 Principen för ett fyra bitars register

i vipporna. Utgångarna kommer därefter att visa ingångarnas inställning vid ögonblicket för klockningen. Detta värde behålls alltså tills nästa klockning ställer vipporna i nya lägen.

Om vi använder MS-vippor i registret i fig 2.22 och om vi dessutom kopplar ihop vipporna inbördes på lämpligt sätt får vi ett skiftregister. Fig 2.23 visar exempel på två sådana register. Det vänstra betecknas PISO (= parallell in och serie ut) och det högra SIPO (= serie in och parallell ut).

Med en klockpuls kan vi ladda PISO-registret med ett ord (om fyra bitar). Med fyra successiva skiftpulser kan vi därefter flytta ut en bit i taget på serieutgången.

SIPO-registret fungerar omvänt. För varje skiftpuls läses en bit in från serieingången medan övriga bitar flyttas ett steg åt höger. Om registren skiftas med samma hastighet (skiftpulsfrekvens) kan man tydligen överföra dataord i serieform mellan de två registren. Det är på detta sätt en terminal (exempelvis en teletype) kommunicerar med sin dator.

Mikrodatorfabrikanter bygger som regel generellt användbara register som "kringkretsar" mellan mikroprocessorn och dess omgivning. Kretsar som sänder ut eller tar emot data i serieform och kommunicerar med mikroprocessorn i parallell form brukar kallas SIO (eller ASIA) och fungerar i princip som registren i fig 2.23.

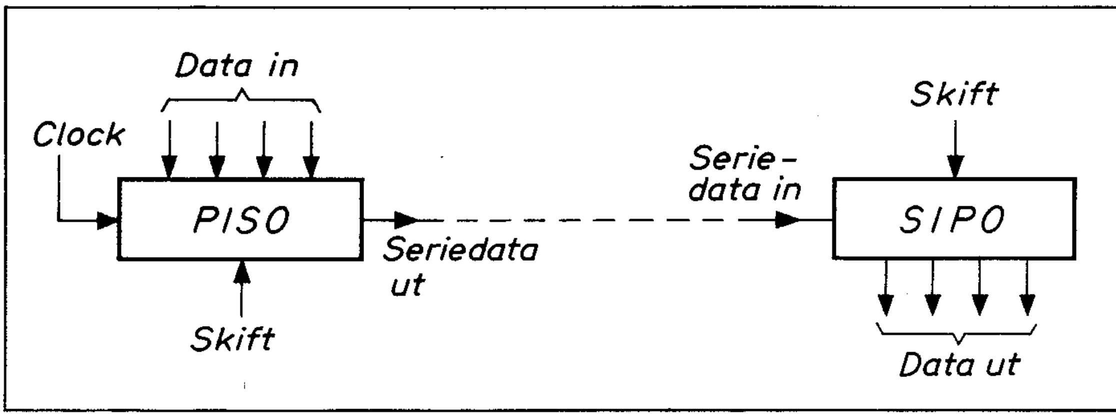


Fig 2.23 Serie- och parallellmatning av register

Beteckningen IO hänför sig till input output och PIO betyder alltså parallell inutkrets. SIO betyder på motsvarande sätt seriell inutkrets.

Fig 2.24 visar de generella PIO- och SIO-kretsarna till Zilogs mikrodatorsystem Z80.

"Kringkretsar" eller "periferikretsar" av den typ som visas i fig 2.24 är generellt användbara och "programmerbara" för att kunna utföra en rad olika uppgifter. Vid den första bekanskapen med dessa kretsar är det emellertid enklast att betrakta dem som parallellt eller seriellt arbetande register.

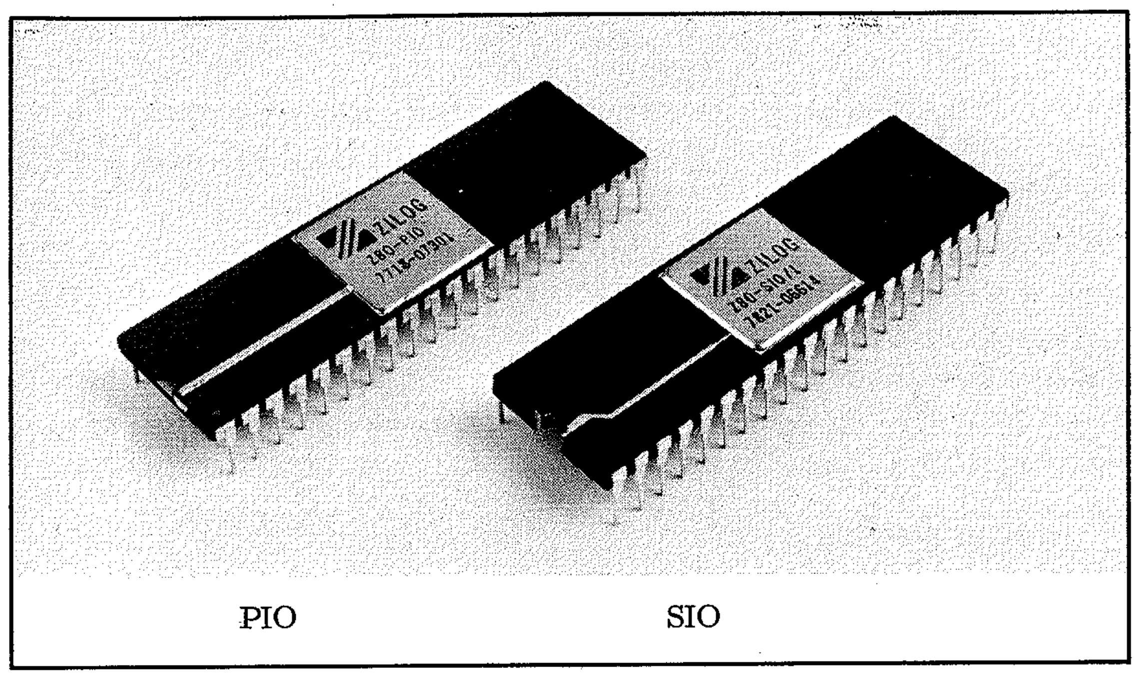


Fig 2.24 Generella kringkretsar till Z80-systemet

3.2 Räknare

Vippor kan kopplas ihop till räknare. Fig 2.25a visar ett exempel på två JK-vippor som bildar en fyra-räknare. När ingången påförs pulser antar utgångarna (A och B) som framgår av fig 2.25b successivt fyra olika värden:

	A	B
	0	0
puls 1→	0	1
puls 2→	1	0
puls 3→	1	1
puls 4→	0	0
puls 1→	0	1
etc	etc	

Kretsar liknande den i fig 2.25a kan användas som räknare eller som frekvensdelare. Utfrekvensen på utgång B är som framgår av fig 2.25b fjärdedelen av infrekvensen. Kretsen delar alltså frekvensen med fyra (divide by four).

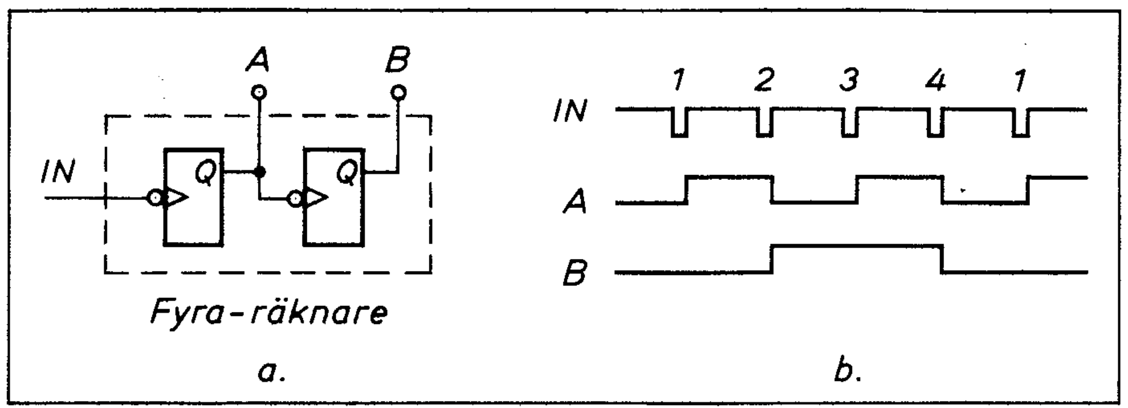


Fig 2.25 Räknare uppbyggd av vippor
(Vippornas J- och K-ingångar antas ligga höga)

3.3 Skriv/läs-minne

Skriv/läs-minnets princip är enkel. Vi sätter in vippor i samtliga krysspunkter mellan ord- och bitledningar i fig 2.16 (istället för dioder). Detaljerna med inkopplingen är inte väsentliga för att förstå principen. Vi har nu möjlighet att ställa vipporna i godtyckliga lägen och därmed kan vi snabbt (inom bråkdelen av en mikrosekund) skriva in data på önskad plats i minnet.

Fig 2.26 visar principen. Dataingången (överst) står i förbindelse med vippornas D-ingångar och bestämmer därigenom vad som ska skrivas in. Datautgången (nederst) står i förbindelse med vippornas Q-utgångar och vi kan därmed avläsa deras inställning. Givetvis får vi endast läsa eller skriva ett ord i taget. Ordledningarna kopplar in vipporna med hjälp av dioder på liknande sätt som vi tidigare mött i läsminnet (fig 2.17). På dataingångarna överst och datautgångarna nederst sitter tristate-buffertar som kopplas in med styrledningarna skriv (överst) respektive läs (nederst). Vi kan alltså med adresseringen välja ut ett ord (dvs en rad) och därefter skriva in via dataingångarna eller avläsa innehållet på datautgångarna.

Fig 2.26b visar ett principschema där man använder samma ledningar för data in och data ut. Eftersom samma ledningar här används för både data in och data ut måste man ha en kontroll-ledning (skriv/läs-kontroll) med vars hjälp vi talar om för minnet om vi vill skriva in data eller avläsa innehållet i adresserat ord.

Av praktiska skäl låter man ofta en minneskapsel innehålla enbart enbitsord (men desto fler!). Vanliga kapslar innehåller exempelvis 4096 vippor organiserade som 4096 ord om vardera 1 bit. För att adressera 4096 ord behövs 12 adressledningar. Vill man bygga ett minne som innehåller 4096 ord om vardera 8 bitar så styr man 8 sådana kapslar med samma adressledningar, fig 2.27.

Data in
Skriv
Adress in
Avkodare
D
Enable
Q
D-vippa med tristate på in- och utgång
Läs
Data ut
a.

Adress in
Avkodare
Minnes-matris
Skriv/läs kontroll
Kontroll
Data in/ut
b.

Fig 2.26
Skriv/läsminne
a. Schema
b. Blockschema

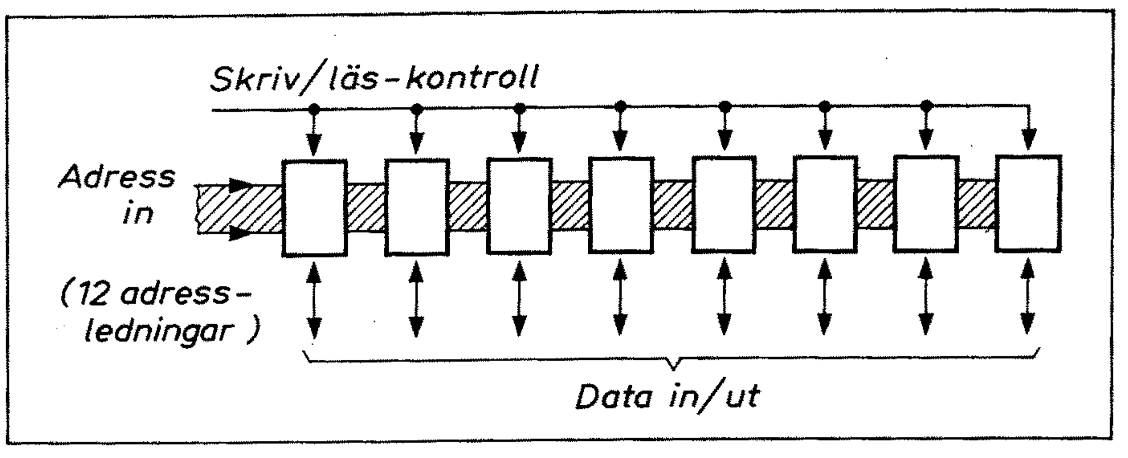


Fig 2.27
Halvledarminne uppbyggt av 4K-kapslar (4096 ord om 8 bitar)

Vi har ovan beskrivit ett "statiskt" halvledarminne. Det kvarhåller den inskrivna informationen så länge matningsspänningen ligger kvar oförändrad. Vid strömavbrott tappar emellertid vipporna sitt innehåll och därför säger man att ett statisk minne är "volatilt" eller flyktigt.

Ett kärminne är i princip uppbyggt som fig 2.16 men med små ferritkärnor (ringar) i varje krysspunkt. Genom "knepiga" kretsar kan dessa kärnor magnetiseras i en av två möjliga riktningar. Den inställda magnetiseringen kan dessutom avläsas (med hjälp av strömpulser genom kärnorna). Ett kärnminne tappar inte minnet vid strömavbrott. Det fungerar ju som ett antal magneter. Det är allstå inte "volatilt". Men kärnminnen är långsammare, dyrare, tyngre och tar mer plats än halvledarminnen. Därför har de konkurrerats ut i flesta vanliga tillämpningar.

En ny magnetisk minnestyp som kallas bubbelminne håller på att utvecklas. Det är emellertid långsamt och ej heller av "random access"-typ varför det ej utgör något alternativ till snabba halvledarminnen (men väl till vissa typer av massminnen som "trummor" och "diskar").

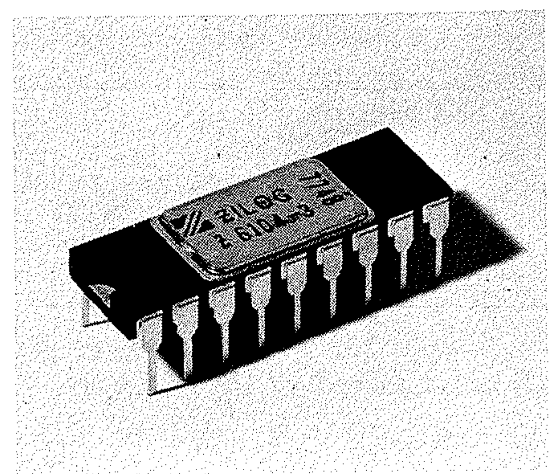
Den kanske viktigaste typen av halvledarminnen är det "dynamiska" minnet. Det är av "random access"-typ och i princip uppbyggt enligt fig 2.16. I krysspunkterna har det dynamiska minnet kondensatorer som är väl isolerade från omgivningen och därmed kan hålla laddningen flera millisekunder. För att ett dynamiskt minne ska kunna bibehålla informationen måste laddningarna i minnets kondensatorer underhållas med periodisk "refreshing". Det kräver både extra kretsar och refresh-signaler.

Kondensatorerna i ett dynamiskt minne kan laddas eller avkännas med hjälp av fälteffekttransistorer som inkopplas vid adresseringen. Fördelen med ett dynamiskt minne är att det endast krävs en transistor per bit jämfört med sex transistorer per bit i ett statiskt minne (där transistorerna är kopplade som vippor). Det dynamiska minnet är (liksom alla halvledarminnen) volatilt, dvs tappar informationen vid spänningsbortfall.

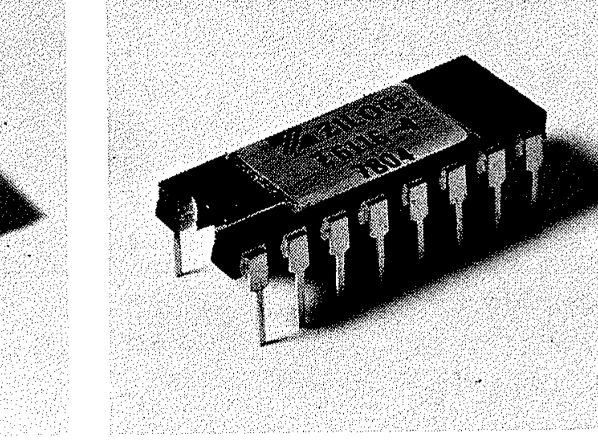
År 1978 kan man köpa kapslar som innehåller 4096 bitar statiskt minne (kallas ofta 4K där K = 1024). Dynamiska minnen finns emellertid tillgängliga som innehåller 64K bitar.

Man räknar med att utvecklingen ska gå vidare mot ca 1 miljon bitar i en och samma kapsel (och till ett pris av storleken "några tior").

Fig 2.28 visar ett statiskt 4K-minne och ett dynamiskt 16K-minne.



4K statiskt minne
(typ Z6104)



16K dynamiskt minne
(typ Z6116)

Fig 2.28 Exempel på halvledarminnen

Sammanfattning

Kombinationskretsar och sekvenskretsar utgör de två huvudgrupperna av logikkretsar. En kombinationskrets avger alltid en och samma utsignal (eller kombination av utsignaler) för varje kombination av insignaler (dvs för varje adress in). En sekvenskrets däremot kan lagra information och här beror utsignalerna på vilken information som tidigare upplagrats.

Ett läsminne (ROM) är exempel på en kombinationskrets för det visar ju ständigt samma information på sina olika adresser. Ett skriv/läs-minne (RAM) är däremot en typisk sekvenskrets.

Vi har tidigare mött enklare typer av kombinationskretsar (grinden, avkodaren och multiplexern) och enklare typer av sekvenskretsar (register och räknare).

3. Datorns funktion i princip

I kap 1 har vi ställts inför en väl specificerad uppgift - styrningen av trafikljusen i en vägkorsning. I kap 2 har vi bekantat oss med en rad elektroniska byggblock, bl a signalgrindar, avkodare, register och läsminnen. Vår uppgift i detta kapitel blir att koppla samman de elektroniska funktionsenheterna så att de kan utföra den specificerade arbetsuppgiften. Vi ska koppla samman elektroniska byggblock till en enkel dator.

1. Utgångspunkten

Vår utgångspunkt är alltså ett problem - eller en arbetsuppgift. Fig 3.1 visar den "apparat" som vi vill sätta i funktion. Apparatens funktion har vi tidigare specificerat i ord. Ofta är det bättre att konkretisera uppgiften i ett schema och fig 3.2 visar ett "flödesschema" som i princip är den arbetsplan efter vilken "apparaten" i fig 3.1 ska fungera.

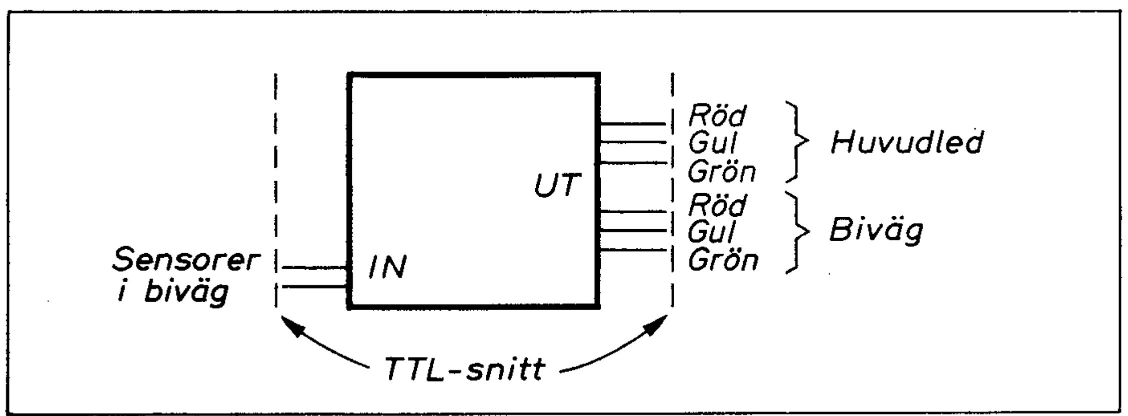


Fig 3.1 "Apparaten" som ska styra trafikljusen

I fig 3.2 har vi använt samma grafiska symboler som programmerare brukar använda för att planera sitt arbete. Symbolerna kan ju användas helt generellt. Startsymbolen har avrundade hörn för att vi ska veta var vi ska börja. Rektanglar anger deluppgifter som ska utföras. Romber anger beslutspunkter. Här måste man bestämma sig för olika alternativ inför det fortsatta arbetet.

Med dessa symboler kan vår arbetsuppgift konkretiseras i fyra block (utöver startsymbolen).

1. Ställ normalljus
2. Läs insignalerna
3. Besluta (på grundval av insignalerna) om ljusväxling ska ske eller ej:

   Om ingen bil inkommit så återgå till ny avläsning av insignalerna.

   Om bil inkommit - gå vidare till ljusväxling
4. Ljusväxling (enligt specifikation). Därefter återgång till ny avsökning av insignaler.

En flödesplan (flow graph) av den typ som vi ovan beskrivit kan man ha nytta av i många sammanhang som inte alls har med datorer att göra!

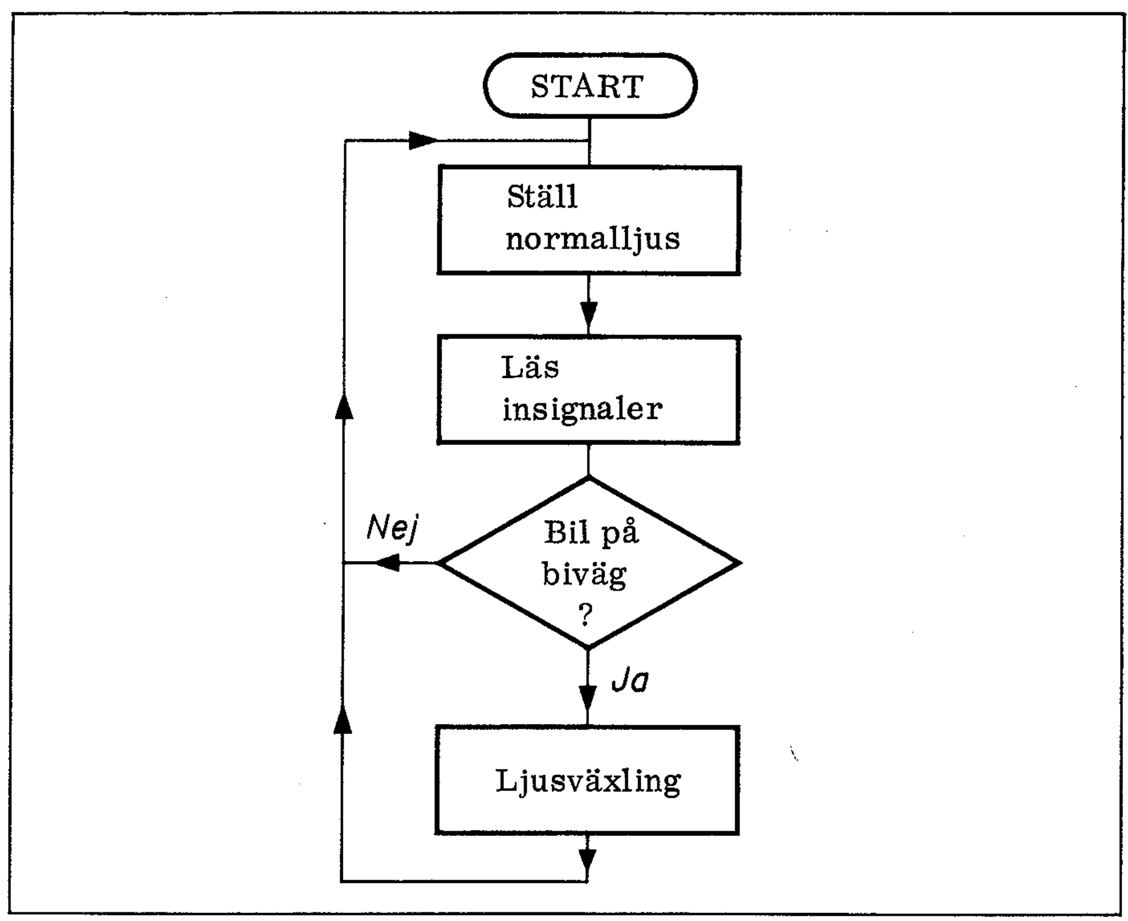


Fig 3.2 "Arbetsuppgiften" presenterad som ett flödesschema

2. Bussarna

Inom elektroniken avser begreppet "buss" en eller flera ledningar, som används av många olika funktionsenheter. Det är alltså fråga om "gemensamma" ledningar, men givetvis måste alla som använder en buss rätta sig efter vissa regler. Den viktigaste regeln är att endast en i taget får styra signalerna på bussen.

Varför är då bussar så utomordentligt viktiga? Svaret är enkelt, om alla signaler skulle få egna ledningar skulle antalet ledningar bli enormt. Det är ekonomiskt omöjligt att ha "privatvägar" för all trafik.

Fig 3.3 visar ett första försök att lägga in en "buss" i vår apparat från fig 3.1. Vi använder en 8 ledningars buss och numrerar bussledningarna från Ø-7 (och inte från 1 till 8). I datorsammanhang skrivs ofta noll med snedstreck för att man enklare ska kunna skilja nollan från bokstaven O.

Insignalerna måste vi ansluta till bussen via tristate-grindar (exempelvis av den typ vi sett tidigare i fig 2.7). Det är ju flera kretsar än ingångarna som vill lägga signaler på bussen. Med hjälp av tristate-kontrollen till vänster i fig 3.3 kan vi alltså släppa in insignalerna på bussen i det ögonblick vi har behov av dem. Vid andra tidpunkter kan de vara bortkopplade (tristate) så att andra enheter får lägga data på bussen.

Utsignalerna kan vi lagra i ett utgångsregister (en latch) och det gör vi enkelt genom att ge registret till höger i fig 3.3 en klockpuls vid lämplig tidpunkt. Därefter kan vi alltså i lugn och ro avläsa utsignalerna även om bussen snabbt övergår att förmedla andra och interna signaler.

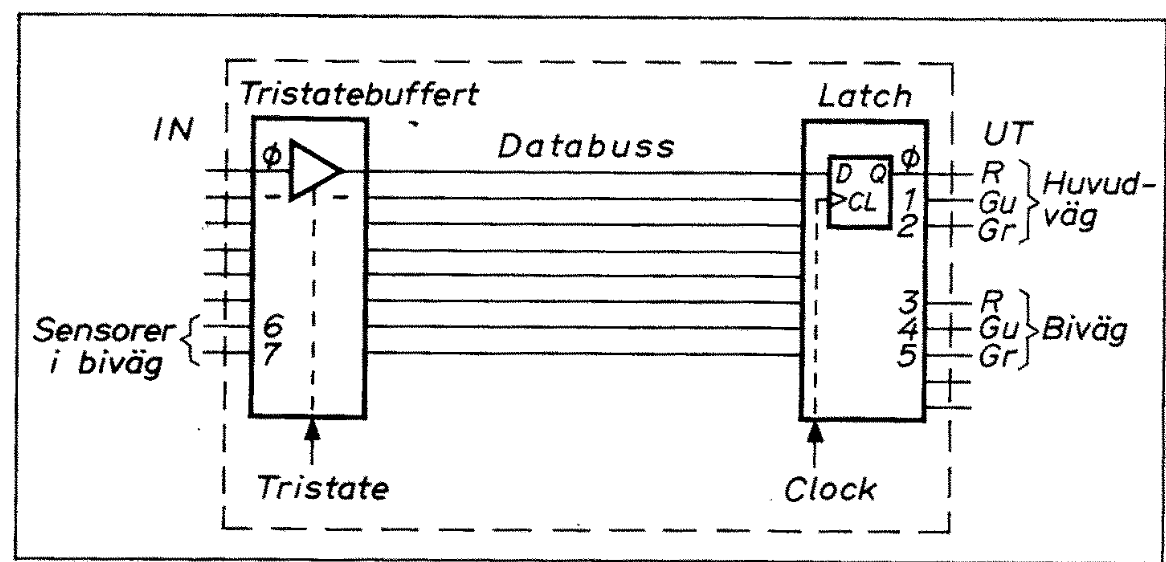


Fig 3.3 Databussens uppgift: Att förmedla data

Låt oss koppla in ingångarna till bitarna 6 och 7 och utgångarna till bitarna ∅, 1 och 2 (huvudväg) respektive 3, 4 och 5 (biväg).

Den buss vi beskrivit i fig 3.3 samlar in data från ingången och sänder ut data till utgången och vi kallar den därför databuss.

Frågan är nu hur vi ska kunna styra trafiken på databussen. Och det är här avkodare kommer in i bilden! I fig 3.4 har vi lagt in ytterligare en buss som vi kallar adressbuss samt dessutom två avkodare.

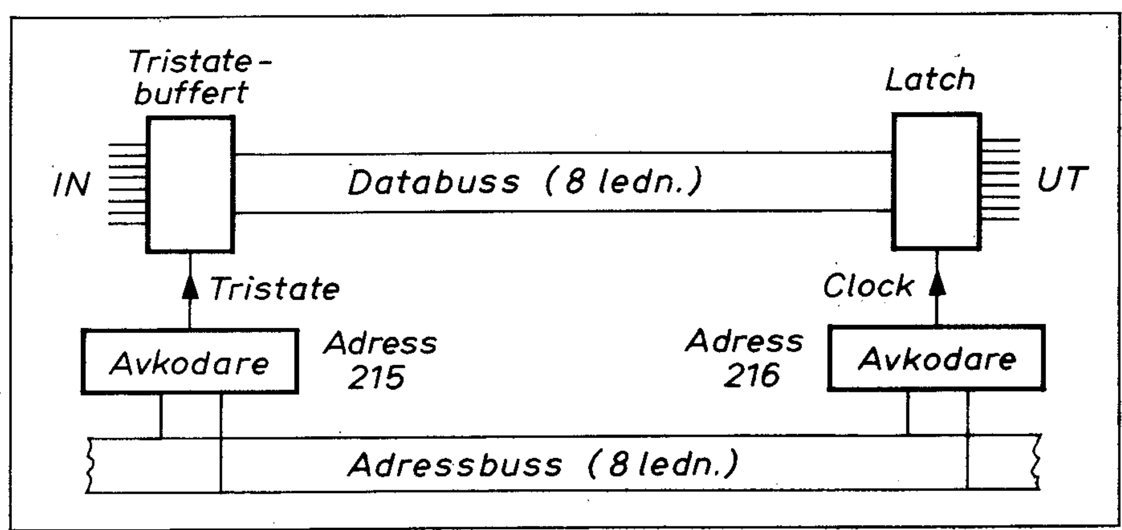


Fig 3.4 Adressbussens uppgift: Att tillfälligt ansluta olika "adressater" till adressbussen

De flesta mikrodatorer har 16 bitars adressbuss och kan som vi sett tidigare särskilja på $2^{16} = 65536$ olika adresser. I fig 3.4 har vi endast ritat ut 8 adressledningar vilket alltså ger $2^8 = 256$ olika adresser (∅ till och med 255). Det är fullt tillräckligt för våra behov just nu.

Låt oss lägga tristate-bufferten på ingången på adress 215 och utgångslatchen på adress 216. Det kan vi göra med hjälp av AND-grindar med lämpligt inkopplade inverterare (fig 2.14).

Saken är klar. Vill vi släppa in insignalerna på bussen ska vi lägga ut adressen (dvs signalkombinationen) 215, dvs binärtalet 1101 0111 på adressbussen. Vill vi vid ett annat tillfälle lagra bussens innehåll i utgångslatchen så lägger vi adress 216 på adressbussen. 216 kan vi skriva binärt som 1101 1000. Frågan är nu: Vem ska lägga ut adresserna på adressbussen?

3. Mikroprocessorn

Varje dator har en centralenhet eller CPU (= central processing unit). Mikroprocessorn utgör mikrodatorns CPU (kallas även MPU eller μP). Mikroprocessorn har många uppgifter, bl a att styra trafiken på databussen. Det gör CPU:n genom att i tur och ordning lägga ut lämpliga adresser på adressbussen. Vi ska följa CPU:ns arbete när den utför den andra deluppgiften i flödesschemat i fig 3.2, dvs när CPU:n avläser signalerna på ingången. Vi ska senare justera programmet så att även första deluppgiften blir utförd.

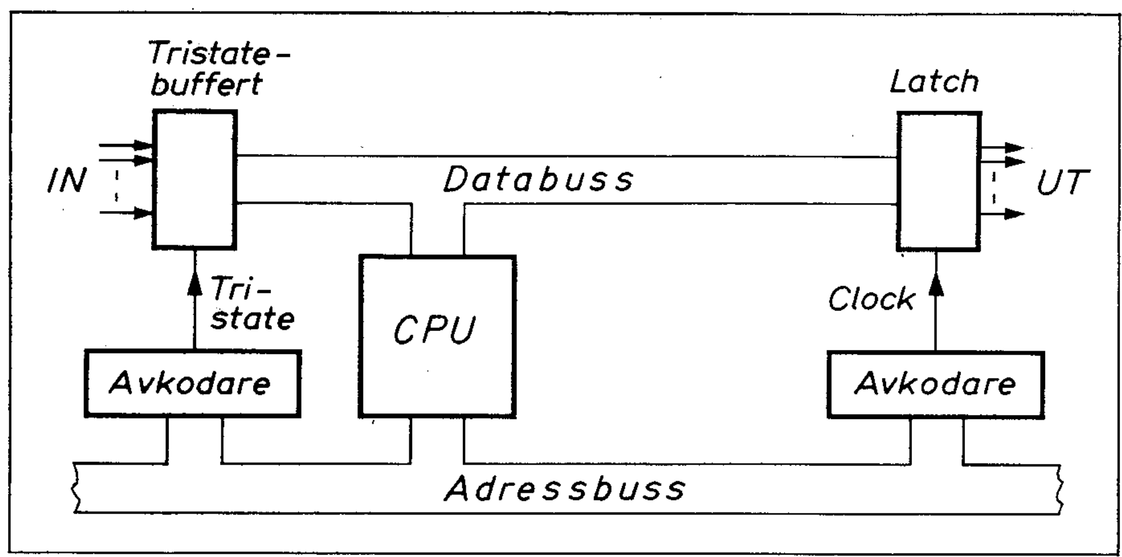


Fig 3.5 Mikroprocessorns uppgift: Att behandla data och kontrollera dataflödet på databussen

CPU:n innehåller flera register. Ett kallas ackumulator och ett annat adressregister. När CPU:n ska läsa data från någon enhet lägger den enhetens adress i adressregistret och klockar sedan (med en viss fördröjning så att data hinner läggas ut på databussen) in data till ackumulatorn.

Fig 3.6 visar principen och med hjälp av siffrorna i figuren kan vi följa förloppet. Adressregistret styr adressbussen. När vi lägger avkodarens adress i adressregistret (1) och därmed på adressbussen (2) så kommer avkodaren att reagera med en klarsignal till tristate-bufferten (3). Insignalerna passerar då bufferten (4) och hamnar på databussen. Om det nu kommer en klockpuls (5) till ackumulatorregistret så lagras insignalernas värden i ackumulatorn.

Frågan är nu, hur vet CPU:n vad den ska göra. Hur ger vi den instruktion om att utföra inläsningssekvensen i fig 3.6?

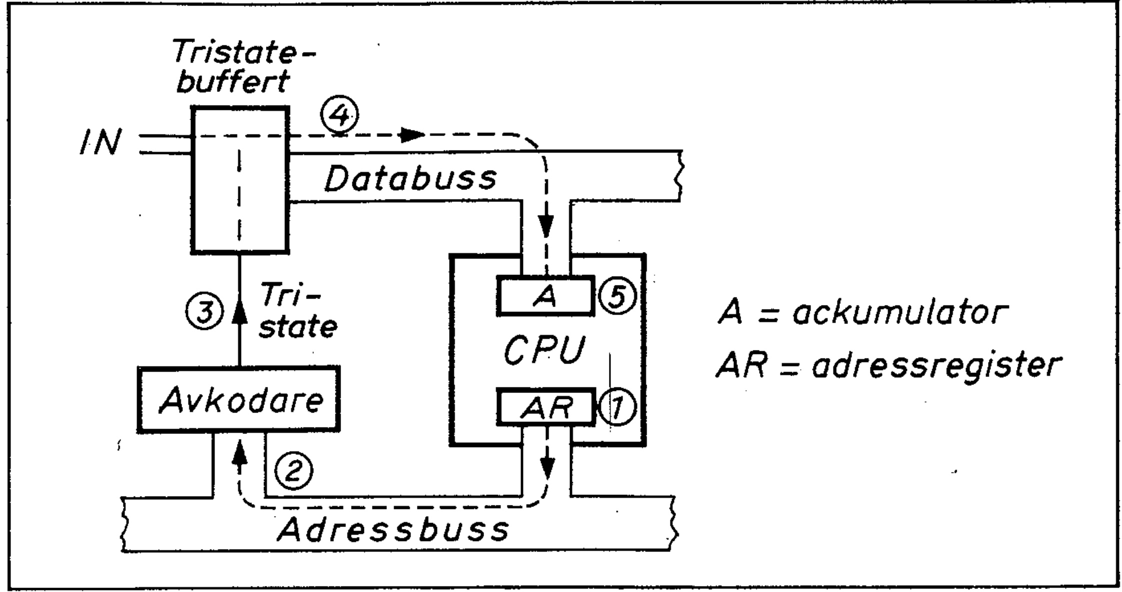


Fig 3.6 CPU:n läser indata

Finessen med en dator är att CPU:n hämtar sina instruktioner (arbetsuppgifter) från minnet. Frågan är bara var i minnet datorn söker sin första instruktion.

En väsentlig ingång på en CPU är RESET-ingången (nollställning). När CPU:n fått RESET-signal börjar den hämta innehållet från en bestämd adress (vanligen adressen Ø) i minnet och vad som där står tolkar CPU:n som en instruktion. När denna instruktion har utförts hämtar CPU:n innehållet från efterföljande minnesadress och tolkar detta innehåll som nästa instruktion.

Om vi har placerat lämpliga instruktioner i minnet på de adresser där CPU:n förväntar sig instruktioner kommer tydligen CPU:n att se till att arbetet blir utfört.

3.1 CPU:ns register

För att klara sina uppgifter måste CPU:n utöver ackumulator och adressregister innehålla ytterligare två register. De kallas programräknare och instruktionsregister.

- Programräknaren PC (programpekare, program counter) har till uppgift att peka ut adressen till nästa instruktion i minnet.
- Instruktionsregistret IR (instruction register) har till uppgift att lagra denna instruktion så att styrenheten ska kunna tolka den och utföra den.

Fig 3.7 visar de fyra registren i en enkel CPU. Figuren visar dessutom hur dessa register står i förbindelse med de två bussarna, databussen och adressbussen. Data kan överföras via 8 numrerade grindar (som var och en innehåller 8 signalledningar).

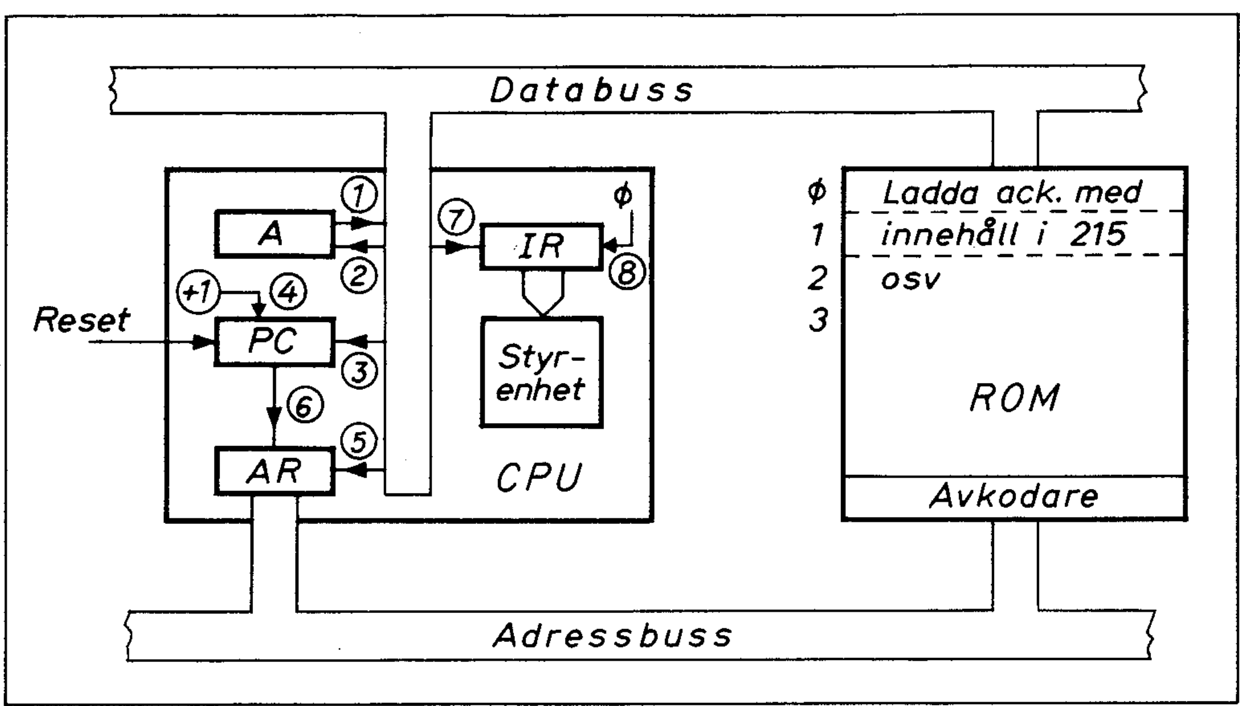


Fig 3.7 Numrerade signalgrindar i CPU:n nödvändiga för exekvering av instruktionen "ladda ackumulator från adress ..."

Till höger i fig 3.7 ser vi programminnet och där har vi på adress ∅ och 1 skrivit in instruktionen

∅: "ladda ackumulatorn med"
1: "innehållet på adress 215"

Instruktionen tar två byte i anspråk och för att i fortsättningen enklare kunna hänvisa till den ska vi beteckna den med förkortningen

LDA
(215)

Förkortningar av detta slag kallas MEMO-koder (mnemonics). De har valts för att vara enkla att komma ihåg. LDA syftar på "load accumulator" eller "ladda ackumulatorn". Parentesen omkring 215 antyder att det är innehållet i cell 215 som avses (och inte talet 215).

Instruktionen LDA (215) ska alltså utföra den arbetsuppgift som vi diskuterat i anslutning till fig 3.6. Hur ska nu detta gå till? Låt oss följa ett tänkbart förlopp.

Instruktionshämtning (FETCH):

Vår utgångspunkt är alltså att CPU:n har fått RESET-signal så att programräknaren PC blivit nollställd. Detta är högst väsentligt för i annat fall kan ju programräknaren ha ett godtyckligt värde.

Med en styrsignal (eller klockpuls) på grind (6) kan vi lägga PC:s innehåll (som alltså är Ø) på adressbussen. Därvid kommer den adresserade minnescellen i läsminnet att lägga sitt innehåll på databussen. Det innehållet är just koden för instruktionen "ladda ackumulatorn". Databussens innehåll kan nu placeras i instruktionsregistret IR med hjälp av grinden (7).

Instruktionshämtningen avslutas med att programräknaren PC "inkrementeras" dvs ökas med ett. PC kommer därmed att peka ut efterföljande adress i programminnet. För att kunna beskriva förloppet enkelt antar vi att PC kan inkrementeras med hjälp av en styrsignal på grind (4) i fig 3.7.

Nu är det styrenhetens tur att avkoda instruktionen och därefter verkställa den.

Verkställande (EXECUTE):

Styrenheten vet nu att den aktuella instruktionen är "ladda ackumulatorn" med innehållet i den adress som anges av efterföljande ord i programminnet. Det gäller alltså att först hämta adressen och lägga den i adressregistret samt därefter att läsa in databussen i ackumulatorn.

EXECUTE-förloppet börjar alltså med att PC (som tidigare inkrementerats) läggs ut på adressbussen genom signal på (6). Därmed får vi cell 1 adresserad och dess innehåll hamnar på databussen. Innehållet utgör koden för tal 215 dvs adressen till ingångsbufferten. För att kunna adressera ingången måste denna adress läggas i adressregistret AR via grind (5). (Här behövs eventuellt en extra buffert eftersom AR ska lägga ut adress 1 till programminnet och samtidigt ta emot innehållet från programminnet. Men dessa detaljer överhoppas här!)

Nu är alltså adressen till ingångsbufferten (215) utlagd på adressbussen. Tristate-bufferten på ingången är adresserad och därmed ligger insignalerna tillfälligt på databussen. Vi lagrar insignalerna i ackumulatorn med hjälp av grind (2). Därmed är den önskade instruktionen exekverad.

Att utföra en instruktion av den typ ovan beskrivit består som vi sett enbart av enkelt "trafikarbete". Det gäller att i rätt ordningsföljd lägga ut signalvägar på bussarna på ett sådant sätt att uppgiften kan utföras. Det måste givetvis - som i all trafik - vara ordning och reda på de inre signalerna. Alla vägar får ju inte öppnas samtidigt! Men det sköter styrenheten om.

Det är givetvis inte nödvändigt att i detalj förstå hur en styrenhet fungerar. Principen är emellertid så enkel och samtidigt så elegant att det vore synd att missa själva kärnpunkten i datorns funktion - datorns mikroprogram.

Låt oss studera en förenklad styrenhet som kan tolka upp till 16 instruktioner. För att särskilja 16 olika instruktioner behövs 4 bitar ($2^4 = 16$). Dessa fyra bitar ska ingå i instruktionskoden och alltså "tolkas" av styrenheten. Detta ska ge upphov till en bestämd sekvens av styrsignaler till de signalgrindar som vi numrerat från (1) till (8) i fig 3.7. Låt oss vidare anta att de flesta arbetsuppgifterna kan klaras på en sekvens av fyra delförlopp. Det behövs då ytterligare två bitar för "timing" eller "sequencing", dvs för att dela upp instruktionen i tidsmässigt åtskilda delförlopp.

Hela styrenheten kan nu byggas upp mycket enkelt. Vi låter de fyra bitarna från instruktionsregistret IR samt ytterligare två bitar från en fyra-räknare tillföras adressingångarna på ett ROM enligt fig 3.8.

Läsminnet i fig 3.8 kommer att innehålla $2^6 = 64$ ord. Om vi behöver 8 styrsignaler ut låter vi ordlängden bli just 8 bitar. Om vi först tillför IR en instruktionskod och därefter successivt stegar fram räknaren (Ø, 1, 2, 3) så får vi tydligen ut fyra efterföljande ord om vardera åtta bitar från läsminnets datautgång. Eller uttryckt på annat sätt: Vi får på datautgången fyra successiva styrsignalkombinationer. Man kallar en sådan sekvens av styrsignalkombinationer för ett mikroprogram.

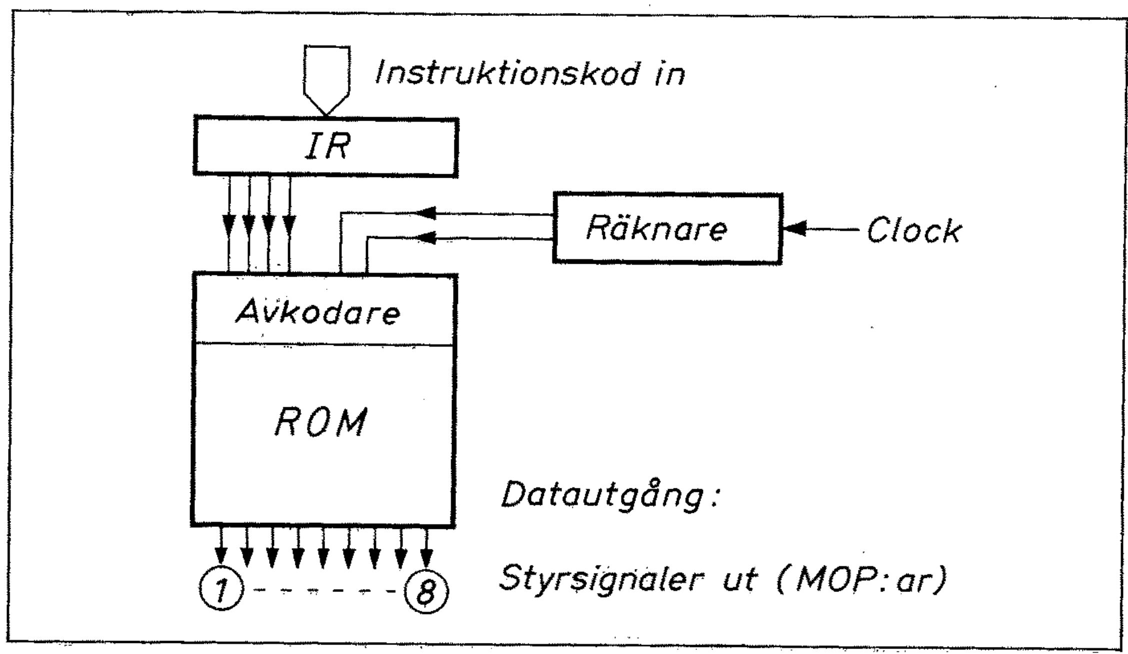


Fig 3.8 Styrenheten: Ett mikroprogramminne och en räknare

3.3 Mikroprogram

Ett mikroprogram styr alltså det inre arbete som utförs etappvis av CPU:ns styrenhet. Denna följd av "mikrooperationer" (MOP:ar) ligger lagrad i mikroprogramminnet i fig 3.8.

Den instruktionskod som vi tillför datorn (exempelvis ladda ackumulatorn) görs alltså om i styrenheten till en följd av mikrooperationer. Vi har tidigare sett hur dessa mikrooperationer måste se ut för att fullgöra uppgiften och nu återstår enbart att programmera in dem i läsminnet i fig 3.8. Den konsten brukar kallas "mikroprogrammering" och utförs normalt av datorfabrikanten. Det finns emellertid mikrodatorer (exempelvis typ 2901) som man själv kan mikroprogrammera.

För den instruktion vi ovan diskuterat skulle ett mikroprogram i princip kunna se ut enligt fig 3.9 (där styrsignalernas nummer hänför sig till fig 3.7). Pilar indikerar datavägar. PC→AR betyder alltså att PC:s innehåll laddas i AR. PC+1→PC betyder att PC inkrementeras (dess innehåll ökas med ett).

Vi har i fig 3.9 förutsatt att IR nollställs vid RESET och att mikroprogrammet därför startar på adress Ø där instruktionshämtningsrutinen (FETCH) ligger.

Om vi antar att instruktionskoden för vår instruktion är 3 (0000 0011) kommer mikroprogrammet för vår speciella instruktion "ladda A" att börja på adress 3Ø.

	Adress	Funktion	MOP nr
FETCH	Ø Ø	PC→AR	6
	1	DB→IR	8
	2	PC+1→PC	4
	3	—	-
	1 Ø		
	1		
	2		
	3		
	2 Ø		
	1		
	2		
	3		
EXECUTE	3 Ø	PC→AR	6
	1	DB→AR	3
	2	PC+1→PC	4
	3	DB→A, Ø→IR	2, 8

Fig 3.9
Mikroprogram (i princip) för "ladda A från adress". Styrsignalnummer enl fig 3.7

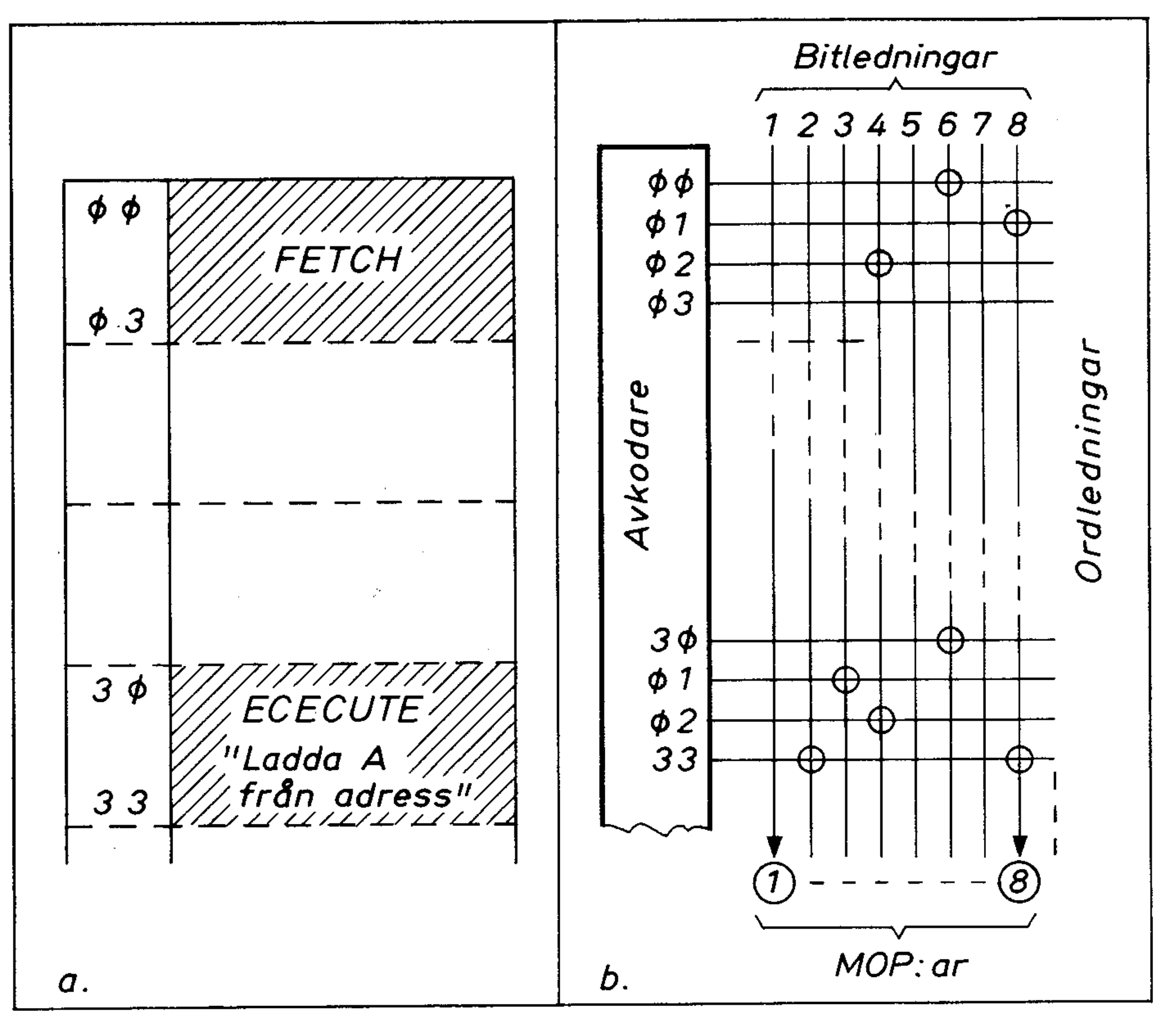


Fig 3.10 Mikroprogramminnets innehåll för FETCH och EXECUTE enligt fig 3.9 (jämför fig 2.17)

FETCH-sekvensen är gemensam för alla instruktioner. EXECUTE-sekvenserna blir däremot placerade i mikroprogramminnet på de adresser som anges av respektive operationskoder.

Den gemensamma FETCH-sekvensen är inlagd på adress ØØ - Ø3 och EXECUTE-sekvensen för den aktuella instruktionen ligger på adress 3Ø - 33. (På adress 1Ø - 23 ligger mikroprogram för andra instruktioner).

Det är flera moment i vårt mikroprogram som kräver speciella kretslösningar. Exempelvis förutsätter DB→AR att databussen hinner avläsas innan den nya adressen lagts i adressregistret. Meningen med vårt enkla mikroprogram är endast att antyda principen och inte att visa alla detaljlösningar.

Fig 3.10 visar hur den önskade sekvensen av styrsignaler kan inprogrammeras i läsminnet. (Jämför med fig 2.17)

Fig 3.7 - 3.10 visar den fundamentala princip efter vilken alla datorer arbetar. Vi har visserligen bara behandlat en enda instruktion (av normalt 50 - 250 olika) och endast utnyttjat 8 styrsignaler eller MOP:ar (av normalt 50 - 100). Det är emellertid den enkla principen vi ska komma ihåg. Då blir en dator inte längre "hokus pokus" utan en efter enkla principer arbetande maskin - även om antalet instruktioner kan vara stort och även om varje instruktion i vissa fall kan vara komplicerad.

Repetera fig 3.7 till 3.10 och gå igenom mikroprogrammet i fig 3.9 ännu en gång! Utgångspunkten är alltså att CPU:n har fått RESET och att programräknaren därmed är nollställd!

Den sista styrsignalen (MOP:en) i fig 3.9 är Ø→IR (dvs nr 8). Den medför alltså återgång till adress Ø i mikroprogramminnet och därmed hämtning av efterföljande instruktion.

Vi har nu bekantat oss med hur datorn fungerar i princip. Vi har diskuterat en ytterligt förenklad dator men vi har inte behandlat hur en dator kan "räkna". I nästa avsnitt ska vi behandla ALU:n, dvs datorns räknecentrum. Vi måste då också diskutera "det binära talsystemet" och hur binära tal kan skrivas i mer koncentrerad form med hjälp av oktala eller hexadecimala siffror.

Detta med olika koder, oktala och hexidecimala siffror och aritmetik upplevs ofta av nybörjaren som svårbegripliga områden. Det krävs också övning om man ska bli van att hantera hextal. För att förstå hur en dator arbetar i princip behöver vi inte all denna detaljkunskap och därför kan vi nu ta litet lättare på fortsättningen. Hoppa över de stycken av avsnitt 3.4 där det känns absolut obegripligt och gå tillbaks dit senare om det behövs. Många nya begrepp behöver tid att mogna.

Datorn förstår sig bara på nollor och ettor och att vi nu inför hextalen är bara ett enklare skrivsätt för dessa nollor och ettor. Så fort vi börjar använda hextalen praktiskt så lär vi oss dem automatiskt och det är ju dessutom ett vettigare sätt att lära sig på än att "läsa torr teori".

3.4 ALU:n

De flesta mikrodatorer arbetar med styrning och reglering, dvs arbetsuppgifter av den typ vi mött i vårt exempel. I dessa tillämpningar behövs som vi sett inte mycket räknearbete.

Matematiska beräkningar är ett annat stort arbetsområde för mikrodatorer. Många datorsystem arbetar huvudsakligen med beräkningsarbeten. Exempel på detta är moderna kassaapparater (point of sale terminals).

Generella mikrodatorer som 8080, 6800 och Z80 kan arbeta effektivt inom båda dessa arbetsfält. För att klara av räkneuppgifter krävs att CPU:n innehåller en aritmetisk logisk enhet, eller kortare ALU. En ALU fungerar som en "räknemaskin", fig 3.11. Om ALU:n tillförs innehållet i två register (A och B i fig 3.11) så kan den som "resultat" avge exempelvis summan eller skillnaden av registrens innehåll, beroende på vilken styrsignal ALU:n ges. ALU:n kan inkrementera eller dekrementera ett tal (dvs öka eller minska talet med ett). I detta fall behövs alltså bara ett register på ingången. I fig 3.7 har vi med grind 4 avsett ALU:ns arbete med inkrementering.

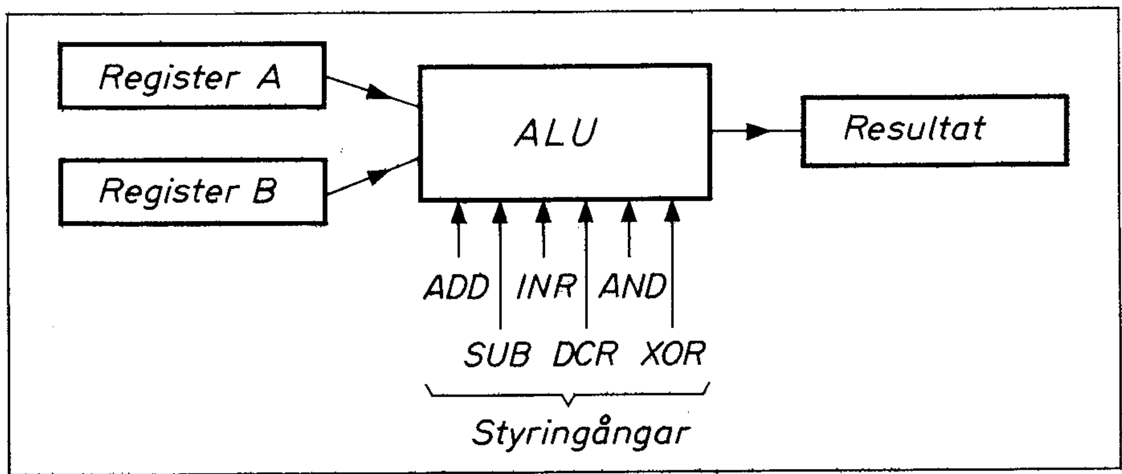


Fig 3.11 ALU:ns funktion i princip

En ALU kan även utföra logiska funktioner. Med XOR kan ALU:n jämföra två tal och med AND kan ALU:n "maska bort" vissa bitar av ett tal och därmed möjliggöra "bithantering". Ett exempel på detta utgör våra insignaler i fig 3.1. Om vi enbart vill ha med signaler från bitarna 6 och 7 kan vi göra AND med masken 1100 0000.

Fig 3.12 visar hur detta tillgår. Bitarna körs parvis genom AND-grindar och de bitar i insignalen som paras med "nollor" i masken kommer därvid att "maskas bort".

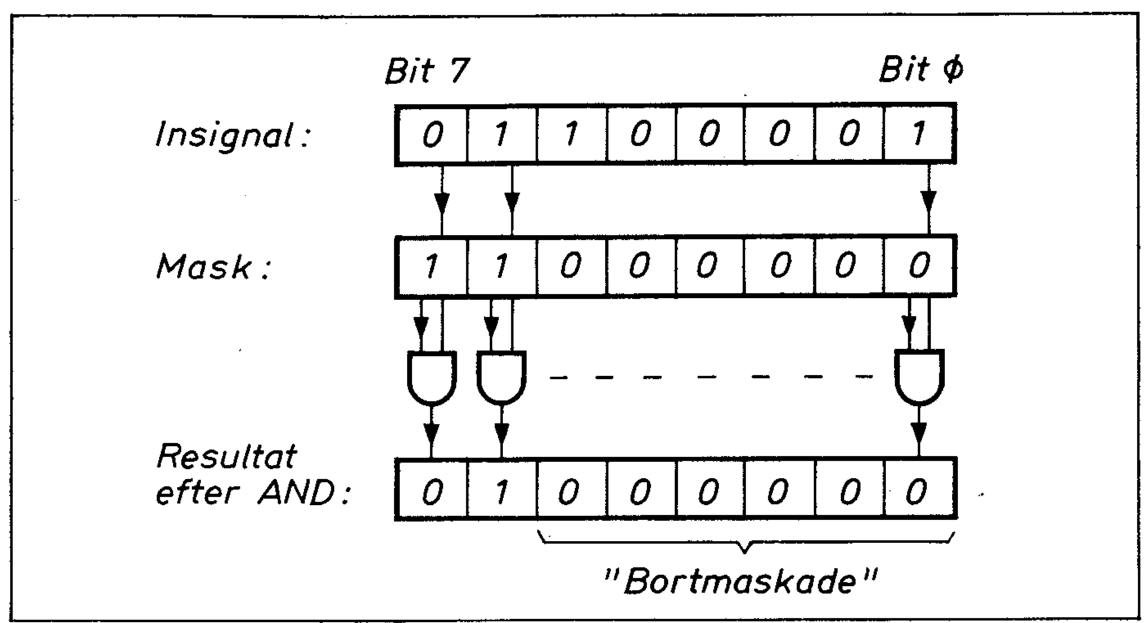


Fig 3.12 "Maskning" med hjälp av AND-funktionen

Binära tal

För att kunna addera binära tal måste man först definiera ett binärt talsystem. Vi arbetar med åtta bitars databuss och vi kan särskilja 256 (= 2^8) olika kombinationer (av nollor och ettor) på de åtta ledningarna. Givetvis skulle vi kunna numrera dessa kombinationer godtyckligt från 1 till 256. Men det är ju alltid bättre med tanke på fortsättningen att vara systematisk och därför använder vi oss av det binära talsystemet. Det är uppbyggt efter samma "positionsprincip" som vårt vanliga decimala system. I stället för det decimala talsystemets ental (10^0), tiotal (10^1), hundratal (10^2), tusental (10^3) etc har det binära talsystemet ental (2^0), tvåtal (2^1), fyratal (2^2), åttatal (2^3) etc. Positionsvärdena blir alltså 1, 2, 4, 8, 16 etc i det binära systemet.

Låt oss ta ett exempel. På databussen inkommer talet

bit nr 7 ... ∅

0 0 0 1 1 0 1 0

Vilket tal motsvarar detta i decimala talsystemet? Det är enkelt att ta reda på med hjälp av "bitvikterna"

binär siffra:	0	0	0	1	1	0	1	0
bitvikt:	128	64	32	16	8	4	2	1

0 + 0 + 0 + 16 + 8 + 0 + 2 + 0 = 26

Det binära systemet är faktiskt enklare än decimalsystemet där vi måste multiplicera varje siffra med respektive viktfaktor. Eftersom vi i det binära systemet enbart har noll eller ett behöver vi bara addera bitvikterna för ettorna. Någon kanske har funderat på varför bitarna vanligen numreras från ∅ till 7 och inte från 1 till 8. En anledning är just att bitvikterna med den vanliga numreringen kan tecknas 2^N där N är bitnummer. Högra biten (dvs bit nr ∅) får alltså bitvikten

$$2^0 = 1.$$

Binär aritmetik

För att kunna räkna decimalt måste vi utantill lära oss bortåt 1000 olika talkombinationer i additions- och multiplikationstabellerna (3 + 4 = 7, 8 x 5 = 40 etc). Additionsreglerna för binära tal är däremot enkla. För binär addition gäller följande enkla regler:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 samt minnessiffra (carry)

En sanningstabell för binär addition ser därför ut på följande sätt:

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

där C = carry och S = summa.

Både C- och S-kolumnerna har vi mött tidigare i fig 2.12 och 2.13. Det är ju AND- och XOR-funktionerna. En adderare kan alltså byggas på det enkla sätt som framgår av fig 3.13.

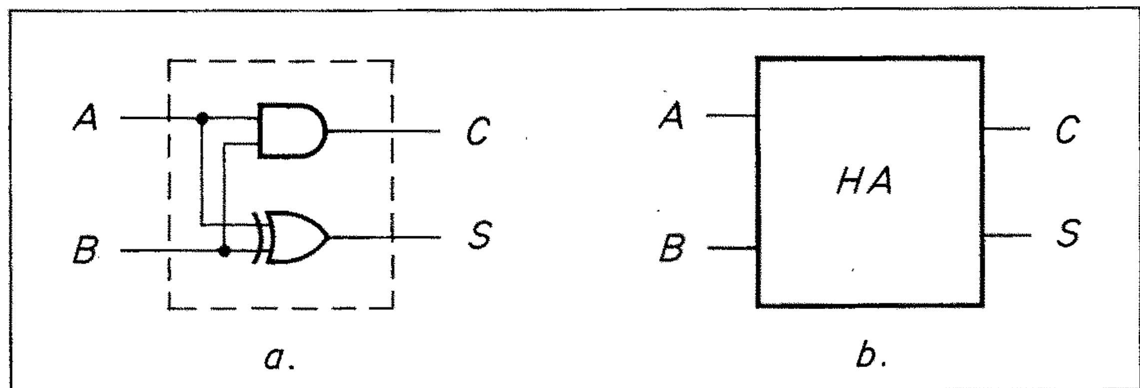


Fig 3.13 Halvadderare (half adder)

När vi adderar siffrorna i två tal (låt oss ta två åttabitars tal) så uppstår ofta "carry":

position		7	6	5	4	3	2	1	0
carry		1	1	1				1	
tal A		0	0	1	1	1	0	0	1
tal B	+	0	1	1	1	0	1	0	1
summa		1	0	1	0	1	1	1	0

I exempel ovan fick vi carry i position 1, 5, 6 och 7. I dessa positioner måste vi alltså addera tre binära siffror. Det klarar inte adderaren i fig 3.13 och därför kallas den ofta halvadderare (half adder). Fig 3.14 visar hur enkelt vi kan utföra addering med en heladderare (full adder). Heladderaren är uppbyggd av två halvadderare enligt fig 3.13 samt en OR-grind.

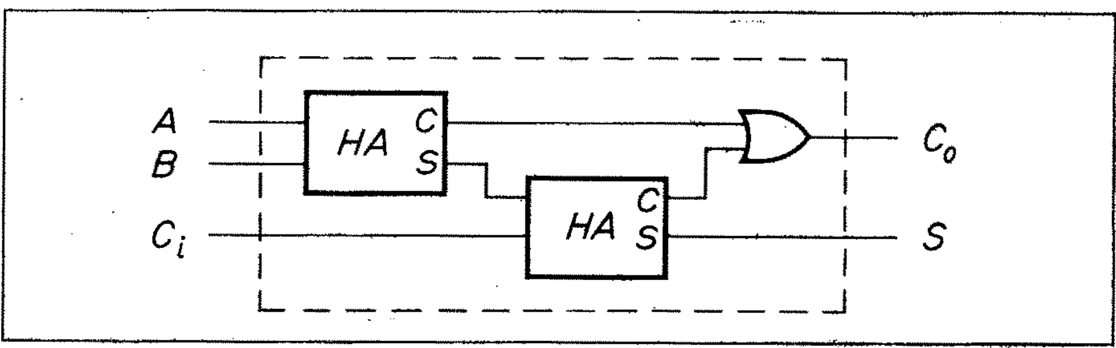


Fig 3.14 Heladderare (full adder)
C_i = carry in
C_o = carry out
S = summa

Om ALU:n i fig 3.11 ges styrsignal på ADD-ingången inkopplas heladderare analogt med AND-kretsarna i fig 3.12. Därvid måste samtidigt carry överföras till närmast högre sifferpositioner. Vi ska emellertid inte här ge oss in på detaljerna med carry.

Oktala och hexadecimala tal

Det är opraktiskt att avläsa och hantera binära tal. Det åtgår ju mer än tre gånger så många siffror som i decimala talsystemet. Binära tal brukar därför skrivas ut i oktal eller hexadecimal form.

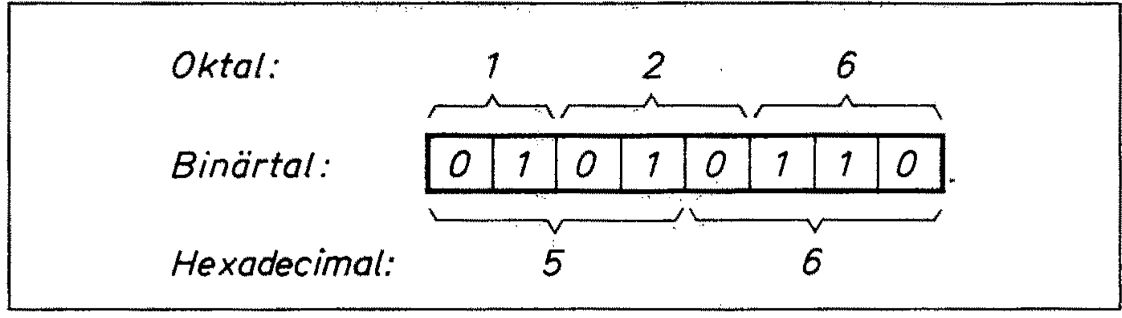


Fig 3.15 Oktaltal och hextal

Principen framgår enklast av fig 3.15. Om man klumpar ihop bitarna i det binära talet i grupper om vardera tre bitar och betecknar varje sådan grupp med dess värde (0 - 7) får man talet utskrivet i oktal form. Man kommer därvid enbart att använda siffrorna 0 - 7 och bitvikterna i det oktala systemet blir potenser av åtta, 1, 8, 64, 128 etc. För att skilja ett oktalt tal från ett decimalt brukar man använda ett index.

$$126_8 = 86_{10}$$

Ovanstående siffror ska alltså tolkas på följande sätt:

Oktaltalet "ett två sex" är lika med decimaltalet "åttiosex". Observera att man inte kan säga oktaltalet "etthundratjugosex" eftersom det inte ingår vare sig hundratal eller tiotal i oktala tal!

Om man klumpar ihop bitarna fyra och fyra (istället för tre och tre) räcker det med två siffror för att ange innehållet i en byte (åtta bitar). Det är förklaringen till att det hexadecimala talsystemet blivit så populärt. Man får givetvis problem med att enkelt ange innehållet i varje klump om fyra bitar eftersom de decimala siffrorna inte räcker till. Man har därför måst införa sex nya siffersymboler. För att kunna använda skrivmaskiner har man därvid tagit bokstäverna A-F istället för att uppfinna helt nya siffror. I fig 3.16 är decimaltalen 0-18 utskrivna i oktal och hexadecimal form.

Decimal	Binär	Oktal	Hextal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	13
etc	etc	etc	etc

Fig 3.16 Decimaltalen 0-18 utskrivna i oktal och hexadecimal form

Det finns givetvis metoder för omformning av decimaltal till oktal- eller hextal och vice versa. Vi kommer inte att slösa tid på sådan drill här, sådana talomvandlingar sköter givetvis mikrodatorn åt oss. Vi kommer däremot ofta att använda hextal för att ange åtta bitars tal och 16 bitars adresser. Vill man programmera en mikrodator på enklaste sätt kan man använda hextangenter (0-F). Fig 3.17 visar en terminal som kan användas för programmering och körning av mikrodatorn Z80 SBC som vi tidigare sett i fig 1.6.

Normalt används versalerna A-F som hextal. Dessa versaler kan emellertid inte återges med sjusegmentdisplay och därför ser man ofta de sex största hexsiffrorna tecknade A b C d E F på sjusegmentdisplay.

Speciella koder

Det binära talsystemet utnyttjar alla bitar effektivt och tillåter enkla aritmetiska operationer. Nackdelen är emellertid uppenbar - den passar inte för oss människor som sedan tusentals år räknat på fingrarna och därmed tänker decimalt.

Fig 3.17 En terminal med hextangenter

En kod som delvis utnyttjar binära talsystemets fördelar men ändå är anpassad för decimalsystemet kallas BCD (binary coded decimal). I BCD-koden används fyra bitar för kodning av en decimal siffra. Man utnyttjar alltså enbart 10 av de 16 kombinationerna. Men man får enkel avkodning till decimala tal.

BCD-koden används som regel i fickkalkylatorer (räknedosor). Den första mikroprocessorn som uppfanns (Intel 4004) var avsedd för fickkalkylatorer och den hade därför fyra bitars ordlängd och instruktioner med vars hjälp man enkelt kunde addera BCD-tal.

Binärkoden har vissa lägen där samtliga bitar ändras, exempelvis mellan 0111 och 1000. Detta kan ställa till problem vid kodning av en vinkelskiva eller vid digital/analog-omvandling. Man har därför gjort en speciell kod där endast en bit i taget ändras och den koden kallas GRAY-kod. I fig 3.18 jämförs binär kod, BCD-kod och Gray-kod för decimaltalen Ø till 15.

Vi har ovan enbart behandlat "numeriska" koder, dvs sifferkoder. En dator ska ju även kunna hantera bokstäver och därför behövs "al-

Decimal	Binär	BCD		GRAY
0	0000		0000	0000
1	0001		0001	0001
2	0010		0010	0011
3	0011		0011	0010
4	0100		0100	0110
5	0101		0101	0111
6	0110		0110	0101
7	0111		0111	0100
8	1000		1000	1100
9	1001		1001	1101
10	1010	0001	0000	1111
11	1011	0001	0001	1110
12	1100	0001	0010	1010
13	1101	0001	0011	1011
14	1110	0001	0100	1001
15	1111	0001	0101	1000

Fig 3.18 BCD- och GRAY-kod jämförda med binärkod

HEX-ASCII TABLE

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

Fig 3.19 HEX-ASCII-tabell

fanumeriska" koder, dvs koder som samtidigt ska kunna hantera såväl siffror som bokstäver (gemener och versaler) samt ett antal olika tecken och kommandon. Den utan jämförelse vanligaste koden härför är ASCII-koden (American standard code for information interchange). Den finns i olika varianter (exempelvis med-och utan paritetsbit) men fig 3.19 ger en lista av de vanligaste tecknen uttryckta som hextal. "Tecken" (character) är ett sammanfattande ord för siffror, bokstäver, skiljetecken m m.

3.5 CPU:ns flaggor

Det blev ett långt avsnitt om ALU:n. Det enda väsentliga vi behöver komma ihåg är att ALU:n sköter "räknandet" och "logiken" i en dator. För vårt trafikljusexempel behövs endast inkrementering och dekrementering (öka och minska med ett) när det gäller räkning. Av logiken kan vi behöva använda AND-funktionen om vi vill "maska bort" eventuella signaler på de 6 ingångar som vi inte använder (men som kan innehålla annan information).

Vi har alltså nu avverkat andra lådan (det kanske är formellt riktigare att säga rutan eller rektangeln) i flödesschemat i fig 3.2. Insignalerna är inlästa till ackumulatorn.

Nästa arbetsuppgift är att fatta beslutet "ja" eller "nej" som svar på frågan "bil på biväg?" Det är tecknat som en romb i fig 3.2. Vi står här inför en av de viktigaste egenskaperna hos en dator, förmågan att fatta enkla beslut. Det sker inte efter "övervägande" eller "omdöme" utan beror enbart på olika "flaggors" inställning.

Flaggor kallas de vippor i CPU:n som ettställs eller nollställs som följd av olika operationer. Olika mikroprocessorer har olika typer av flaggor och dessa flaggor sätts olika vid olika typer av instruktioner. Det är alltså mycket väsentligt att sätta sig in i flaggornas funktion hos den speciella mikroprocessor man använder.

Den enkla CPU som vi tidigare behandlat har bara en flagga, Z-flaggan. Den ettställs när ackumulatorns innehåll är noll. När ackumulatorns innehåll avviker från noll nollställs Z-flaggan. Eventuella bilar på bivägen ger sig tydligen tillkänna genom att någon av bitarna 6 eller 7 blir ettställd, dvs att Z-flaggan blir nollställd. Vilken av bitarna 6 eller 7 som har åstadkommit detta spelar i vårt exempel ingen roll.

Vi har i anslutning till fig 3.7 i detalj följt exekveringen av en instruktion där data flyttas från en given adress och till CPU:ns ackumulator.

Flaggor möjliggör en annan viktig typ av instruktioner som kallas "villkorliga hopp". Instruktionerna i programminnet är ju placerade

i följd efter varandra. Genom inkrementeringen av programräknaren får man automatiskt efterföljande instruktion när den föregående exekverats. Om det däremot kommer en "villkorligt hopp"-instruktion blir förhållandet annorlunda.

Låt oss använda instruktionen "hoppa om ackumulatorn är noll till adress....". För enkelhets skull använder vi MEMO-koden:

JZ = "hoppa om noll"
Ø = "till adress noll"

Enligt flödesschemat i fig 3.20a ska vi hoppa till början av programmet och därför har vi satt in adress Ø ovan.

Vårt program har nu två instruktioner som var och en upptar två byte i programminnet:

Ø	LDA	"ladda A"
1	(215)	"med innehållet i 215"
2	JZ	"hoppa om A = Ø"
3	Ø	"till adress Ø"
4	...	

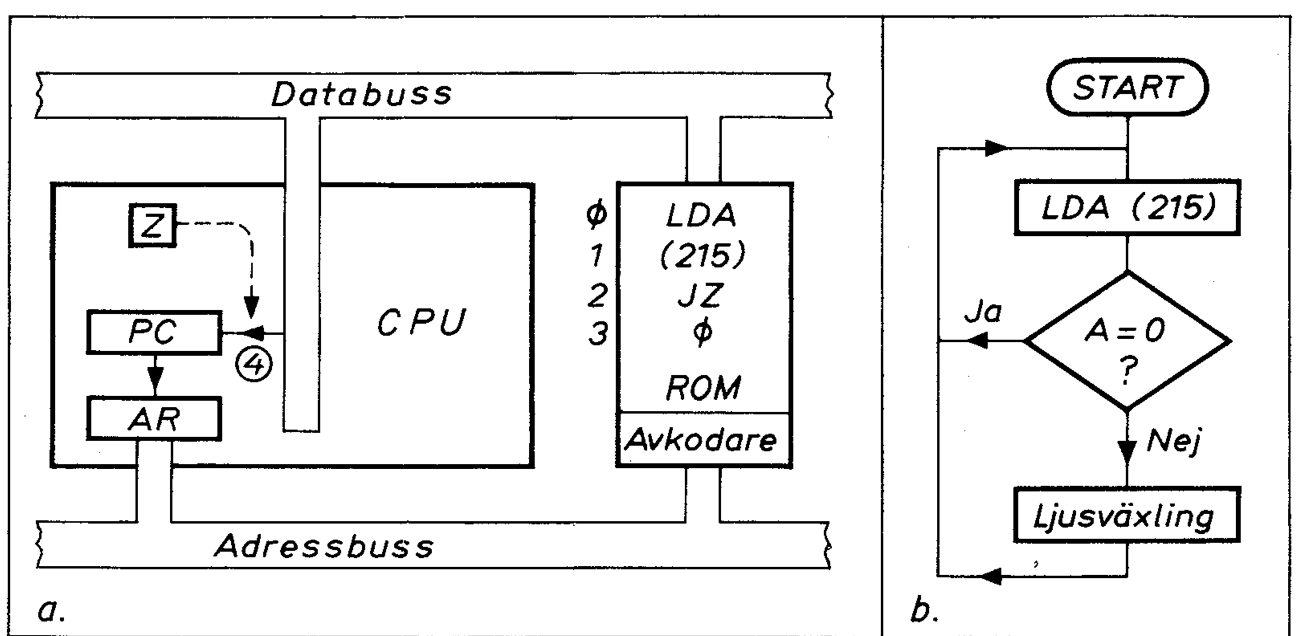


Fig 3.20 "Hoppa om A = Ø"-instruktionen
a. CPU och ROM
b. Flödesplan

Hoppinstruktionens mikroprogram är intressant att studera. Det har vissa likheter med LDA-instruktionens. Fig 3.20.

LDA hämtar adressen (215) och lägger den i adressregistret AR.

JZ hämtar likaledes en adress (Ø) som läggs på databussen. Därefter kommer det viktiga beslutet: Om hoppvillkoret är uppfyllt placeras denna adress i PC. I annat fall bibehåller PC sitt tidigare värde.

Tack vare Z-flaggan är beslutet enkelt att genomföra elektroniskt. Man låter helt enkelt MOP nr 4 koppla in en grind mellan databussen och PC. Denna grind styrs av Z-flaggan. Z = 1 (dvs A = Ø) medför på detta sätt att PC laddas med hoppadressen.

Fig 3.20 sammanfattar situationen. Vi har laddat A från ingångsbufferten (adress 215) och undersöker nu innehållet i ackumulatorn. Vi befinner oss alltså i beslutsromben i flödesschemat (fig 3.20b). Om A = Ø ska vi hoppa tillbaks till adress Ø i programminnet. Om A ≠ Ø ska vi gå vidare till adress 4 där instruktionerna för ljusväxling tar vid.

3.6 Stackpekaren

Vi har ovan sett hur CPU:n kan exekvera ett hopp i programmet och därmed lämna den successiva sekvens av instruktioner som programräknaren annars utpekar.

Det är ofta praktiskt att skriva delar av ett program i subrutiner som kan lagras på annan plats i minnet än huvudprogrammet. Ska man utnyttja en sådan subrutin måste vi alltså göra ett hopp liknande det vi tidigare beskrivit. Sedan subrutinen är utförd måste vi emellertid komma tillbaks till huvudprogrammet och det ställer speciella krav på CPU:n. Det är bl a av denna orsak de flesta mikroprocessorer antingen har en stack (dvs ett extra minne) eller en stackpekare (dvs en pekare som pekar ut en speciell plats i det normala skriv/läs-minnet).

Ofta säger man att man "kallar på en subrutin" och motsvarande instruktion har ibland MEMO-koden CALL.

Instruktionen CALL liknar i mycket vår tidigare hoppinstruktion och den kan vara förenad med liknande villkor. I vårt program behöver vi inte ställa villkor för subrutinanropet och vi ska därför bara behandla ett "ovillkorligt" CALL.

Hur bär sig då CPU:n åt för att komma ihåg återhoppsadressen när subrutinen är exekverad och det är dags att återvända till huvudprogrammet? Saken är enkel. CALL-instruktionens mikroprogram placerar programräknarens innehåll (som pekar på efterföljande instruktion) på "stacken", dvs på den minnesadress som utpekas av stackpekaren. Därefter dekrementeras stackpekaren och är därmed beredd att ta emot fler hoppadresser om det skulle behövas.

Subrutinen måste avslutas med en "return"-instruktion. Dess mikroprogram inkrementerar stackpekaren och hämtar den tidigare lagrade adressen från stacken samt placerar den i programräknaren. Exekveringen återgår därför till det ställe i huvudprogrammet där den tidigare avbröts med subrutinanropet CALL.

Ett program som innehåller subrutiner måste alltid börja med att stackpekaren laddas med den adress man vill ha stacken placerad på. Ofta sätts denna adress högt upp i minnet och stacken växer då mot lägre adresser. Programmet i övrigt använder normalt minnet nerifrån och upp. Den av programmet använda minnesarean växer därför mot högre adresser.

3.7 En subrutin

Vi har nu kommit fram till lådan "ljusväxling" i flödesschemat i fig 3.20b. Vi ska i detta avsnitt visa hur vi kan ställa en ljusbild och hur vi kan införa en fördröjning med hjälp av en subrutin.

För att tända lampor i trafikljusen måste vi sända ettor till motsvarande bitpositioner i utgångslatchen (adress 216). Fig 3.21 visar "bitmönster" för olika "ljusbilder" i vägkorsningen. Under varje ljusbild har vi uttryckt motsvarande bitmönster i hexkod. Subrutinens uppgift är alltså att i tur och ordning - och med angivna intervall - lägga ut de olika hexkoderna på utgångslatchen.

All kommunikation med vår mikrodator sker via ackumulatorn. När vi avläste ingångarna (med instruktionen LDA) placerades resultatet i ackumulatorn. När vi nu ska mata ut hextal (signalljuskombinationer) till utgångslatchen så måste dessa hextal först placeras i ackumulatorn.

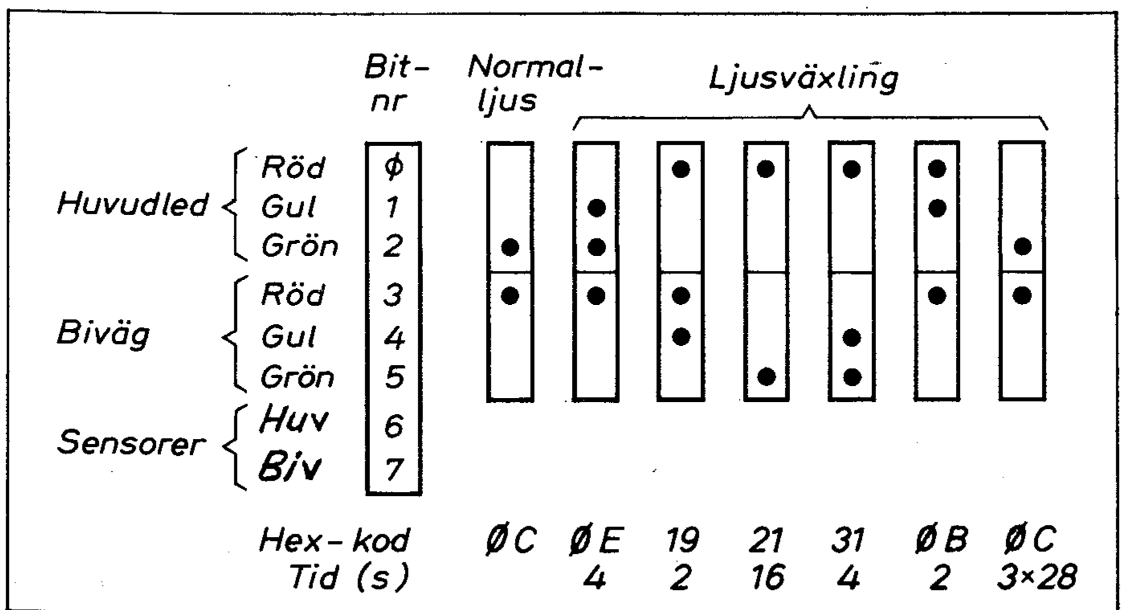


Fig 3.21 HEX-koder för ljusbilder vid ljusväxling

Vi kan ladda tal från programminnet till ackumulatorn med hjälp av en instruktionstyp (eller adresseringsmod) som brukar kallas "immediate".

Instruktionen

LDA	"ladda ackumulatorn"
24H	"med hextal 24"

utför den önskade uppgiften. H anger här att det är fråga om hextal. Lägg märke till skillnaden mellan instruktionerna LDA (24H) och LDA 24H. Instruktionen LDA (24H) laddar ackumulatorn med innehållet i den minnescell som har adressen 24H i skriv/läs-minnet. Instruktionen LDA 24H laddar ackumulatorn med talet 24H (som alltså är lagrat i programminnet).

I engelskspråkig litteratur kallas den förra adresseringsmoden för "direct" och den senare för "immediate". Eftersom de flesta assemblermanualer är skrivna på engelska är det meningslöst att införa svenska ord för dessa begrepp. Den som ska arbeta med datorer och programmering måste vänja sig vid engelska termer.

Trafikljusen kan ställas in till "normalljus", dvs grönt ljus på huvudvägen och rött på bivägen, med hjälp av följande instruktioner

LDA	"ladda ackumulatorn"
ØCH	"med talet ØCH"
STA	"sänd ut A:s innehåll"
(D8H)	"till utporten D8

Den första instruktionen har vi diskuterat ovan. Den senare lagrar innehållet i ackumulatorn på den adress som anges av efterföljande byte. MEMO-koden STA är en förkortning av "store accumulator".

Eftersom vi nu är tränade på hextal har vi i instruktionen STA (D8H) uttryckt den tidigare adressen 216_{10} som hextal. $216_{10} = D8_{16}$ (eller D8H som det ofta skrivs).

Nu återstår bara en rutin. Den ska hålla reda på den tid de olika ljusbilderna ska ligga ute. En sådan rutin kan vi göra genom att låta datorn ligga i en "loop" och dekrementera ett stort tal. När talet minskat till noll hoppar datorn ur loopen (med instruktionen JZ). Vi ska återkomma till detaljerna hur detta kan utföras i nästa kapitel. Här antar vi nu att vi har tillgång till en subrutin som vi kallar "vänta". Om vi lägger ett tal i ackumulatorn och kallar på "vänta" håller datorn på att räkna runt precis det antal sekunder som anges av ackumulatorns innehåll.

Instruktionerna

```
LDA 4H
CALL VÄNTA
```

medför alltså 4 sekunders fördröjning innan datorn fortsätter arbetet med efterföljande instruktion.

Talet som vi här laddat i ackumulatorn brukar kallas för en parameter som vi på detta sätt överför från huvudprogrammet till subrutinen.

3.8 CPU:ns uppbyggnad

Genom att studera hur datorn exekverar delar av ett program för styrning av trafikljus har vi kunnat följa CPU:ns arbete.

Vår CPU från fig 3.7 har nu vuxit och omfattar sex register, en flagga, en ALU, en räknare och ett ROM samt dessutom ett antal signalgrindar som styrs av MOP:ar. Fig 3.22.

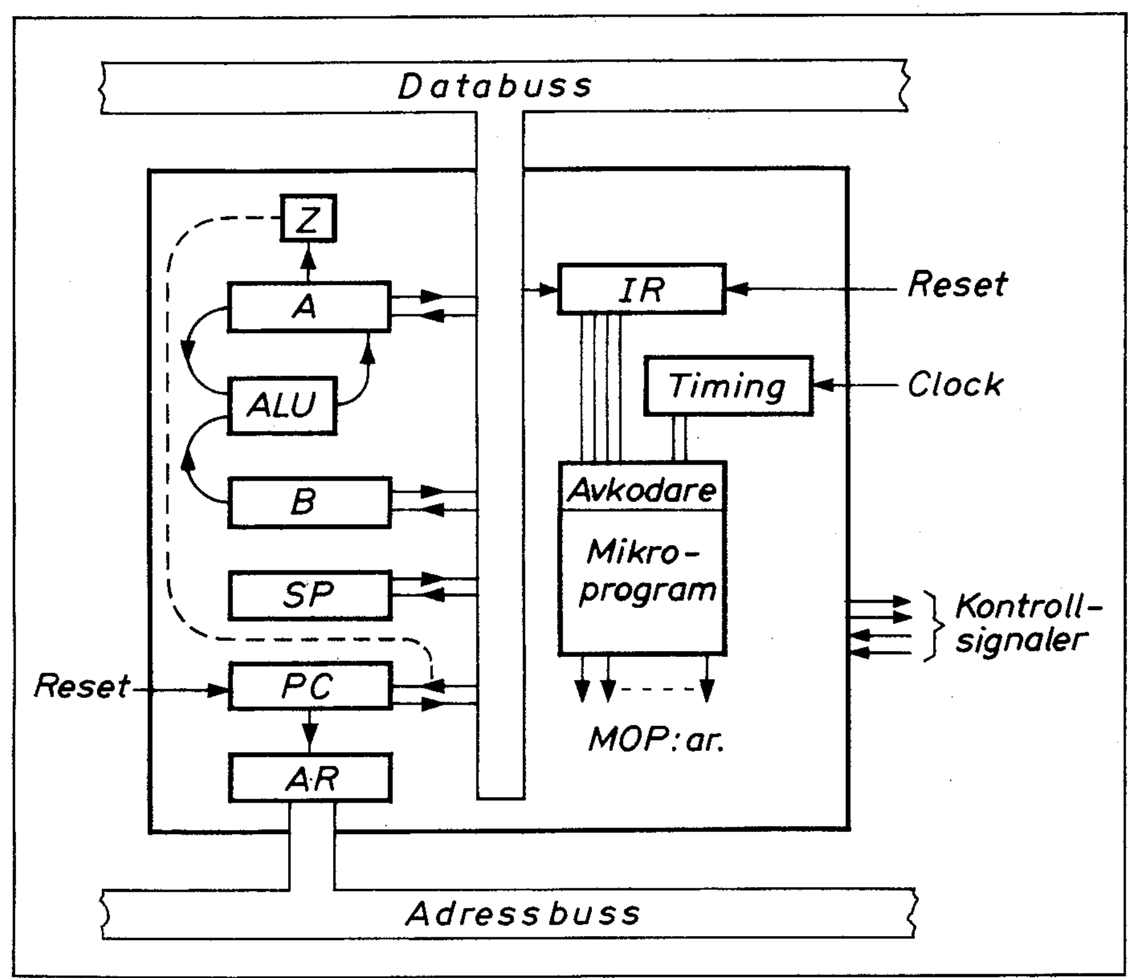


Fig 3.22 CPU:ns uppbyggnad i princip
A = ackumulator
B = extra register
ALU = aritmetisk logisk enhet
SP = stackpekare
PC = programräknare
AR = adressregister
IR = instruktionsregister
Z = nollflagga

Mikroprocessortillverkare brukar aldrig visa detaljerna i sina CPU:er. Fig 3.23 visar som exempel hur Zilog anger uppbyggnaden av sin CPU Z80.

Till vänster ser vi instruktionsregistret, instruktionsavkodaren och styrsignalerna "CPU CONTROL" som motsvarar våra tidigare MOP:ar. Styrenheten har 13 st yttre styrledningar som kallas "CPU AND SYSTEM CONTROL SIGNALS". Det är styrsignaler (exempelvis av typen "läs" och "skriv") som vi inte behandlat tidigare. Motsvarande styrledningar brukar kallas "kontrollbuss" (control bus).

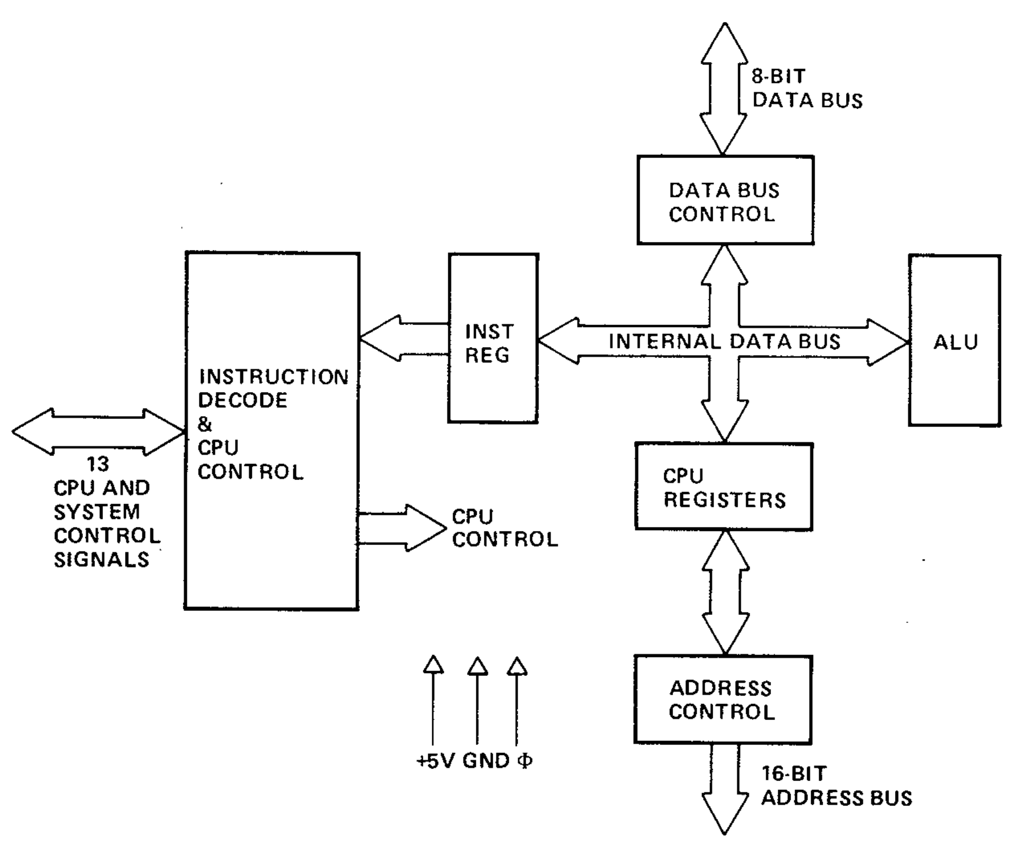


Fig 3.23 Blockschema för CPU:n Z80

Hur en CPU är uppbyggd är visserligen intressant att veta men användaren kommer aldrig i direkt kontakt med många av CPU:ns inre kretsar. Det är egentligen enbart de delar av CPU:n som användaren kommer åt genom olika instruktioner (dvs via programmet) som är väsentliga att känna till. Därför specificerar fabrikanter alltid alla register som nås med instruktioner. Fig 3.24 visar hur Zilog presenterar registren i Z80. Det är sammanlagt 18 st 8 bitars register och 4 st 16 bitars register.

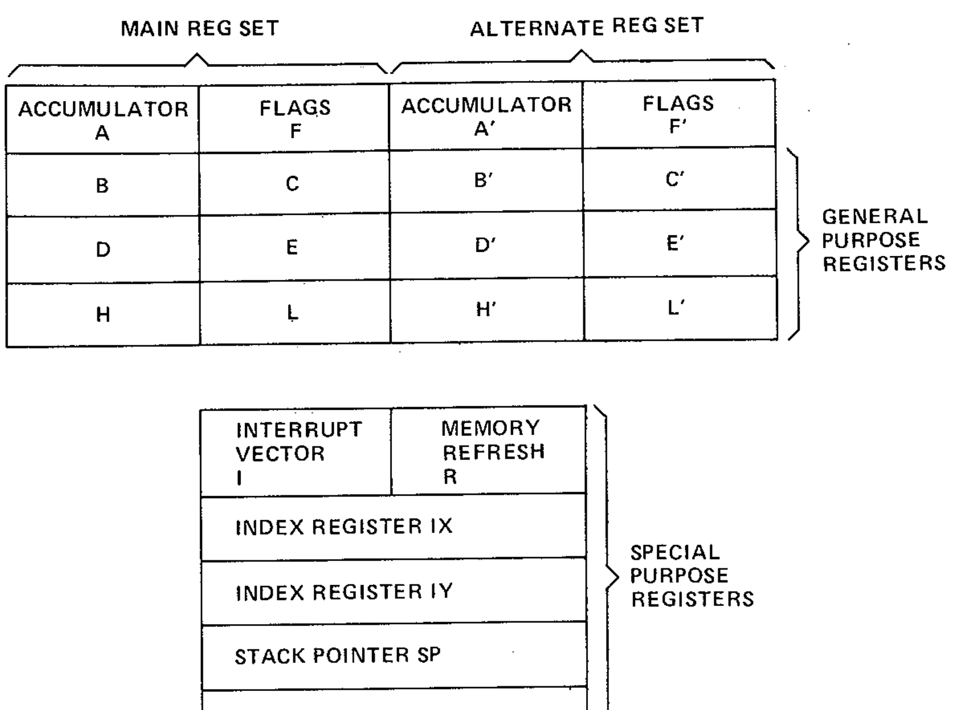


Fig 3.24 Registren i CPU:n Z80

Registren är uppdelade i tre grupper.

- o Main register set
- o Alternate register set
- o Special function registers

Flaggorna har ställts samman till två flaggregister (FLAGS). Fig 3.25. Varje sådant flaggregister innehåller sex flaggor av vilka fyra kan användas för hoppinstruktioner av den typ vi tidigare behandlat.

Vår avsikt är inte att här ge en komplett beskrivning av Z80. Den som är intresserad av de olika registrens funktion ska givetvis läsa de manualer som CPU-tillverkare alltid tillhandahåller. För Z80 återfinner man en teknisk beskrivning i "Z80-CPU Technical Manual" och en detaljerad beskrivning av alla instruktioner (och assembler) återfinns i "Z80-Assemby Language Programming Manual".

Fig 3.26 visar slutligen en närbild på Z80 samt uttagens funktion.

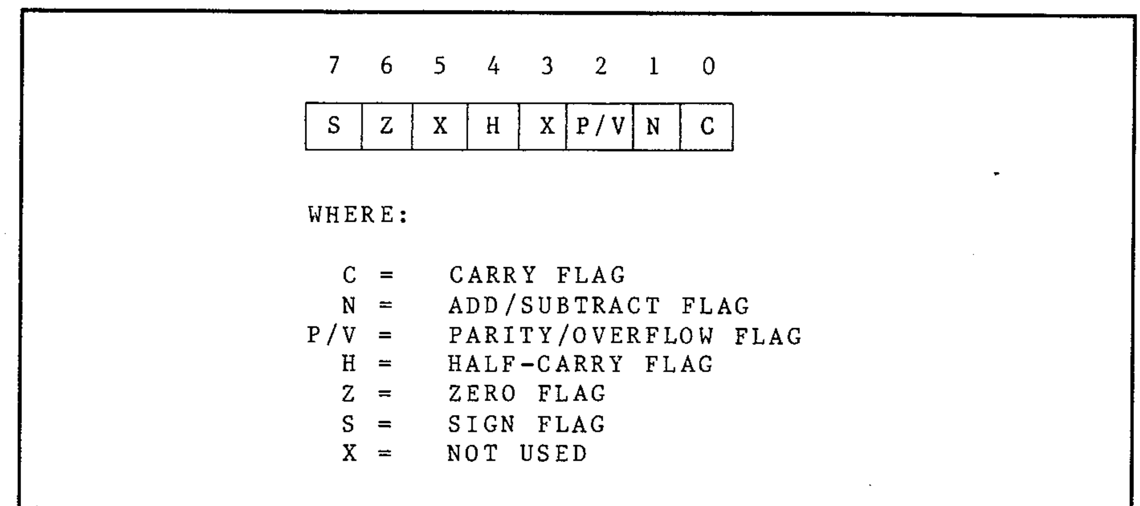


Fig 3.25 Innehållet i ett av Z80:s två lika flaggregister

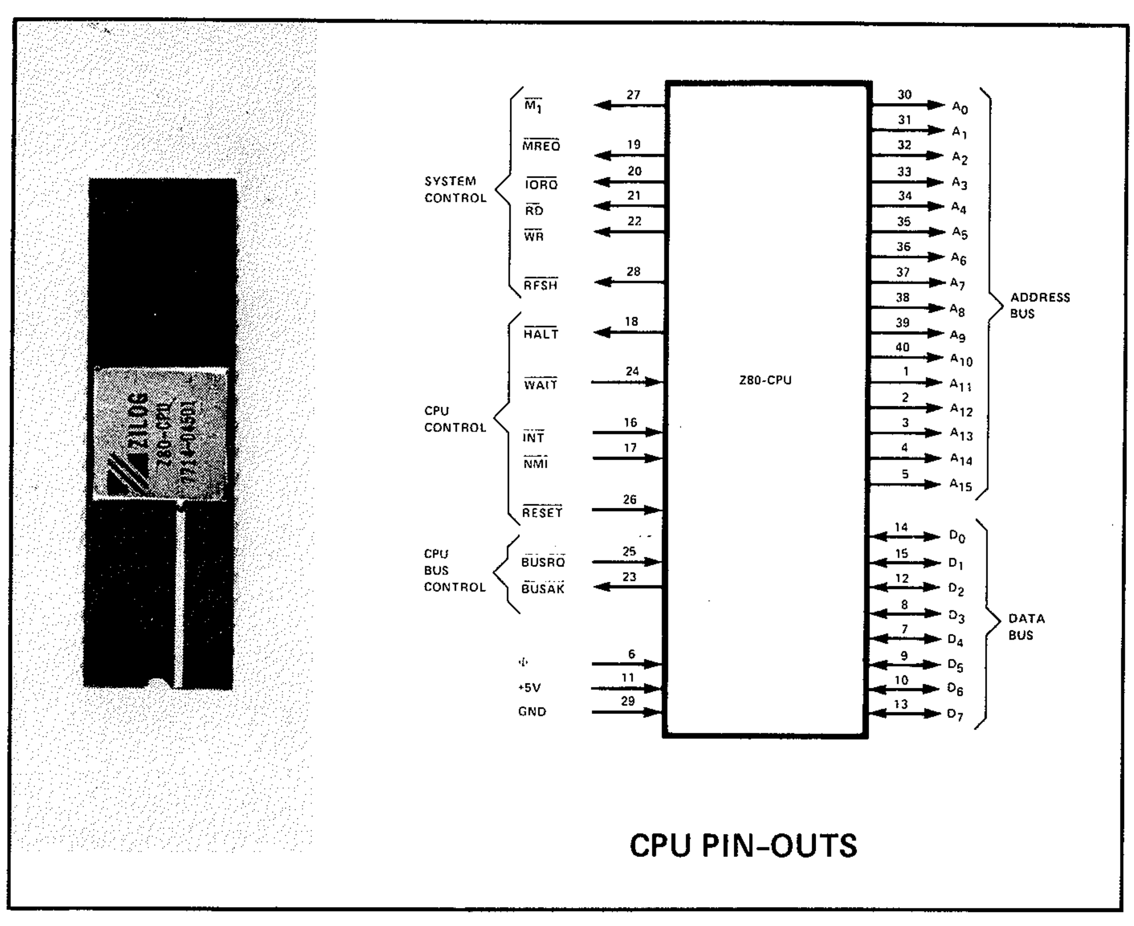


Fig 3.26 Mikroprocessorn Z80 och dess uttagsplacering

4. Trafikljusprogrammet

Vi ska nu sammanfatta datorns funktion vid styrningen av trafikljusen i vår vägkorsning. En sådan sammanfattning gör vi enklast med utgångspunkt från flödesschemat i fig 3.27.

4.1 Huvudprogrammet

Programmet börjar med att stackpekaren ställs på en hög adress (FFH). Det första vi måste göra med trafikljusen är att lägga ut "normalljus", dvs rött på biväg och grönt på huvudväg. Normalljuset har som vi sett i fig 3.21 hexkoden ØC. Första lådan (normalljus) i huvudprogrammet kan vi nu förverkliga med hjälp av två instruktioner

HEX-adress	MEMO-kod	Kommentar
Ø, 1	LDS FFH	ladda stackpekaren med adressen FFH
2, 3	LDA ØCH	ladda A med hextalet ØC
4, 5	STA (D8H)	lagra A på hexadress D8

Till vänster anges HEX-adresserna till de celler i programminnet där instruktionerna har lagrats.

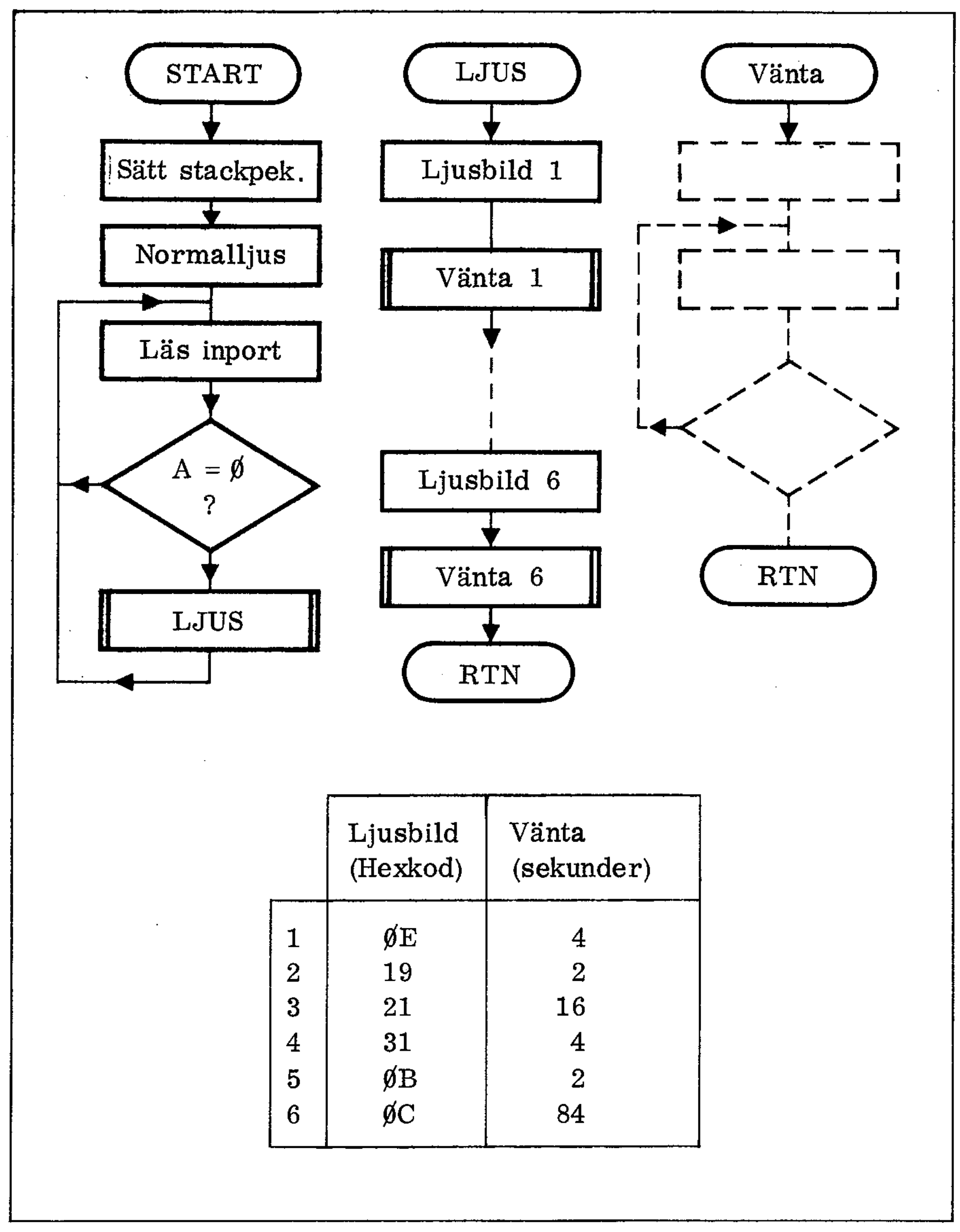
LDA-instruktionen lägger hextalet ØC i ackumulatorn och STA-instruktionen lagrar ackumulatorns innehåll på HEX-adressen D8. Därmed har vi fått normalljus i vägkorsningen.

Lådan "läs inport" och romben "A = Ø?" har vi tidigare diskuterat. De kan förverkligas med instruktionerna

HEX-adress	MEMO-kod	Kommentar
6, 7	LDA (D7H)	Ladda A med innehållet i adress D7H
8, 9	AND CØH	Maska fram bit 6 och 7
A, B	JZ Ø4H	Hoppa om A = 0 till adress Ø4H

Ljusväxlingen har vi lagt som en subrutin och den kan programmeras på följande sätt

HEX-adress	MEMO-kod	
C, D	CALL LJUS	Lagra PC på stacken och lägg därefter adressen "LJUS" i PC
E, F	JMP Ø4H	Hoppa till Ø4H



	Ljusbild (Hexkod)	Vänta (sekunder)
1	∅E	4
2	19	2
3	21	16
4	31	4
5	∅B	2
6	∅C	84

Fig 3.27 Flödesschema för ett trafikljusprogram

Instruktionen CALL lägger en ny adress (som vi ovan kallat "LJUS") i programräknaren PC. När subrutinen är exekverad fortsätter datorn med den efterföljande instruktionen som är JMP ∅4H, ett ovillkorligt hopp till hexadressen ∅4H.

Därmed är huvudprogrammet genomgånget.

Subrutinen måste lägga ut sex olika ljusbilder med olika tidsintervall. Ljusbilder och tider anges nertill i fig 3.27.

Låt oss följa programmet för första ljusbilden.

HEX-adress	MEMO-kod	
1Ø, 11	LDA ØEH	Ladda A med talet ØEH
12, 13	STA (D8H)	Lagra A på adress D8H

lägger ljusbild nr 1 i vägkorsningen. Nu måste datorn vänta 4 sekunder innan ljusbild nr 2 får läggas ut. Denna väntan kan åstadkommas med instruktionerna

HEX-adress	MEMO-kod	
14, 15	LDA 4H	Ladda A med talet 4H
16, 17	CALL VÄNTA	Lagra PC på stacken och lägg därefter adressen "VÄNTA" i PC.

VÄNTA är här adress till en subrutin. Denna subrutin ger en fördröjning med det antal sekunder som anges av ackumulatorns innehåll.

En ljusbild tar tydligen 8 byte programminne i anspråk. Våra sex ljusbilder upptar därmed 48 byte av programminnet. Ljusbild nr 6 är normalljuset och måste ligga ute utan avbrott under 84 sekunder för att tillförsäkra huvudleden grönt ljus under minst 75 % av tiden.

Slutet av subrutinen LJUS kan därmed programmeras på följande sätt:

HEX-adress	MEMO-kod	Kommentar
38, 39	LDA ØCH	Ladda A med talet ØCH
3A, 3B	STA (D8H)	Lagra A på adress D8H
3C, 3D	LDA 54H	Ladda A med talet 54H
3E, 3F	CALL VÄNTA	Lagra PC på stacken och ladda därefter adressen "VÄNTA" i PC
4Ø	RTN	Hämta återhoppsadress på stacken och ladda den i PC.

Sista instruktionen, RTN som betyder RETURN (återvänd till huvudprogrammet), är högst väsentlig. Mikroprogrammet till RTN tar reda på återhopps-adressen som tidigare lagrats på stacken av CALL-instruktionen. Återhoppsadressen placeras i PC och därmed har exekveringen återgått till huvudprogrammet.

Sammanfattning

Vi har i detta kapitel studerat en principlösning av vår trafikljusstyrning. Vi har använt en "datorarkitektur" enligt fig 3.28 och vi har diskuterat CPU:ns roll för trafikdirigeringen på databussen. CPU:n hämtar sina instruktioner från ett ROM och exekverar dessa instruktioner i tur och ordning. Vi har sett hur CPU:n måste utföra varje instruktion i två huvudmoment, FETCH och EXECUTE. Varje sådant moment är uppdelat på deltider under vilka olika datavägar kopplas upp av CPU:ns mikroprogram.

För att lösa trafikljusproblemet behövde vi enbart utnyttja 8 olika instruktioner. Fig 3.29. Vi har studerat olika delar av ett enkelt program och hur det kan delas upp på subrutiner. En sådan subrutin har vi studerat i detalj.

Vi har därmed följt vår princip-dators funktion vid exekveringen av ett trafikljusprogram. I nästa kapitel ska vi studera ett verkligt program som kan köras på Z80.

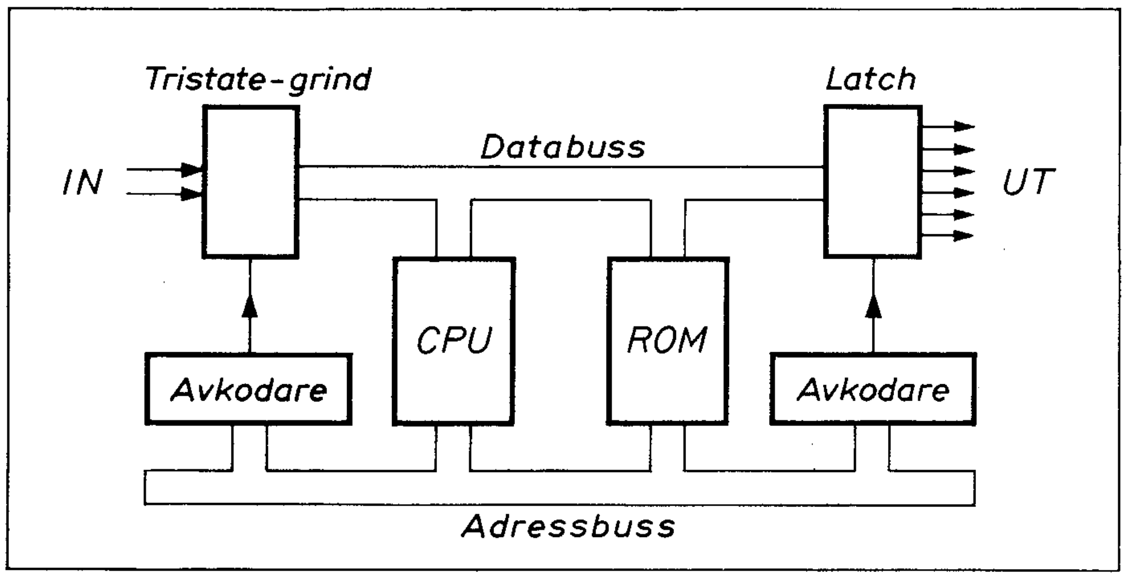


Fig 3.28 En enkel dator

LDS	ladda stackpekaren
LDA	ladda ackumulatorn
STA	lagra ackumulatorn
JMP	ovillkorligt hopp
JZ	hoppa om A = ∅
CALL	hopp till subrutin
RTN	återhopp
AND	AND-funktion (bitvis)

Fig 3.29 Exempel på vanliga instruktioner och MEMO-koder

4. System för trafikljusstyrning

Vi ska nu läma principresonemangen och ge oss i kast med verkliga mikrodatorer och deras programmering. Av praktiska skäl måste vi begränsa oss till en av de många mikroprocessorer som nu finns på marknaden. Vi har valt Z80, som är en av de mest avancerade mikroprocessorerna av åtta bitars ordlängd.

1. Hårdvaran

För att kunna skriva ett riktigt program måste vi först bestämma "hårdvaran" (maskinvaran) dvs hur systemet är uppbyggt med avseende på kretsar och komponenter. Vi har tidigare studerat ett mycket enkelt system som bestod av en inport (ingångsgrinden i fig 3.5), en utport (utgångslatchen i fig 3.5), en CPU och ett ROM innehållande programmet. I fig 4.1 har vi ritat vårt tidigare system med vedertagna bussymboler. Pilarna på bussarna visar signalriktning.

I vårt exempel styrde CPU:n alltid adressbussen och därför är pilen för CPU:ns anslutning till adressbussen riktad mot adressbussen. CPU:ns databuss kan däremot riktas i båda riktningar (men givetvis enbart i en riktning i taget!) Anslutningen till databussen har därför pil i båda riktningar.

När man inte är speciellt intresserad av innehållet i en ingångskrets (i vårt exempel var ingångsgrinden av tristate-typ) eller en utgångskrets (datalatchen) brukar man helt enkelt tala om inportar och utportar. En inport är alltså en ingång till datorn och den kan innehålla ett antal ledningar (dvs bitar) och motsvarande gäller för en utport.

1.1 Kontrollbussen

I den enkla dator vi tidigare diskuterat (fig 4.1) har vi inte använt några kontrollsignaler (utöver RESET vid start). När vi adresserade ROM eller inport var det enbart för läsning, dvs CPU:ns data-

buss var riktad in mot CPU:n. När vi adresserade utporten var det enbart för skrivning, dvs CPU:ns databuss var riktad utåt från CPU:n. Det blev på detta sätt aldrig några oklarheter om databussens riktning.

I verkliga system ingår normalt RAM (skriv/läs-minnen). Ett sådant minne måste (som vi sett tidigare i fig 2.26) utöver adressen även få en signal som talar om huruvida CPU:n har för avsikt att läsa data eller skriva in data i minnet. Hos Z80 sker detta med två separata ledningar, en skrivkontroll ($\overline{WR}$ = write) och en läskontroll ($\overline{RD}$ = read). Normalt ligger dessa ledningar höga. Om CPU:n vill skriva i adresserad minnescell lägger den $\overline{WR}$-ledningen låg. Vill CPU:n vid ett annat tillfälle läsa adresserad cell lägger den $\overline{RD}$-ledningen låg. Beteckningen $\overline{WR}$ antyder att det är "WR:s inverterade värde", dvs låg, som ger önskad funktion. Man kallar en sådan signal för "aktiv låg".

Skriv- och läskontrollerna utgör två av de 13 kontrolledningar som hos Z80 bildar kontrollbussen (eller styrbussen), på engelska control bus. Funktionerna antyds i fig 3.26 och vi ska här beskriva de viktigaste. Tre av dessa har vi redan behandlat

$\overline{WR}$ = skriv, aktiv låg
$\overline{RD}$ = läs, aktiv låg
$\overline{RESET}$ = nollställning (bl a av PC), aktiv låg.

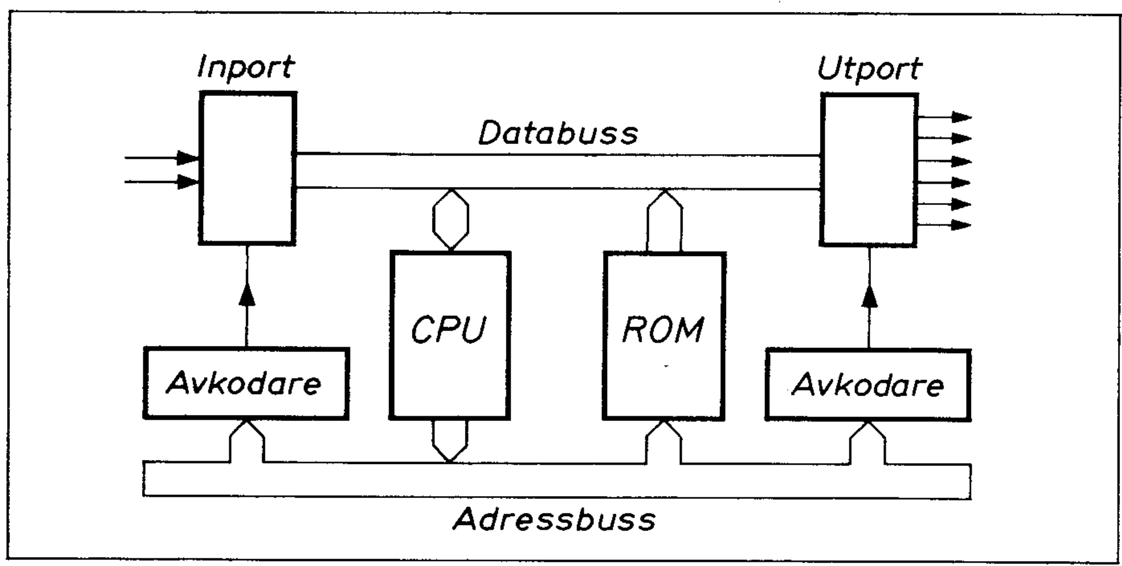


Fig 4.1 Enkel datorstruktur

Två kontrolledningar från CPU:n kallas $\overline{MREQ}$ och $\overline{IORQ}$. REQ eller RQ är förkortning av "request" (begäran).

Med $\overline{\text{MREQ}}$ låg talar CPU:n om för systemet att den adresserar ett minne (M) med hjälp av adressbussens 16 bitar. Det ger en minnesarea av 65536 minnespositioner.

Med $\overline{\text{IORQ}}$ låg talar CPU:n om att den adresserar en IO-enhet. Eftersom man normalt endast har ett begränsat antal IO-enheter (inportar och utportar etc) används enbart 8 bitar av adressbussen för IO-adressering. Man vinner på detta sätt fördelen av enklare avkodning för IO-enheter. Att använda olika adressering för minnen och IO-enheter brukar kallas "separatadresserad IO" eller "IO-mapping"

I vår tidigare principdator skilde vi inte på IO-enheter och minnen vid adresseringen. Man kan givetvis lika bra låta inutportar adresseras på samma sätt som minnen. Den metoden används också under namnet "memory mapping" i flera mikroprocessorer, bl a i Motorola 6800. En fördel med memory mapping är att man kan använda samma instruktioner såväl för minnen som för IO-enheter. Vi använde ju exempelvis LDA-instruktionen vid läsning av inporten.

Man skulle kunna betrakta $\overline{\text{MREQ}}$ och $\overline{\text{IORQ}}$ som två extra adressledningar som ger större adressarea och samtidigt bekvämare avkodning.

Vi ska omnämna ytterligare en ledning i kontrollbussen. Den kallas $\overline{\text{M1}}$ vilket är en förkortning av "maskincykel nr 1". Vi har tidigare sett att varje instruktion måste hämtas (fetch) för att sedan kunna utföras (execute). CPU:n Z80 meddelar att instruktionshämtning pågår genom att lägga $\overline{\text{M1}}$ låg.

Olika mikroprocessorer kan ha olika typer av styrsignaler. Här gäller det alltså att studera manualen för den speciella CPU som man ska använda.

1.2 En generell datorstruktur

En generell datorstruktur kan innehålla ett flertal block, som alla är anslutna till de tre bussarna:

databussen
adressbussen
kontrollbussen

Fig 4.2 visar ett blockschema för en sådan generell datorstruktur. Hade vi inte tidigare diskuterat funktionen hos de ingående blocken hade schemat i fig 4.2 enbart utgjort en (möjligen dekorativ) samling av fyrkanter. Nu vet vi att CPU:n har kommandot över adressbussen och styr hela systemet genom att i tur och ordning avläsa och utföra instruktioner som är permanent lagrade i ROM. IN och UT är systemets kontakt med omvärden och i RAM kan data skrivas in och läsas ut snabbt under arbetets gång.

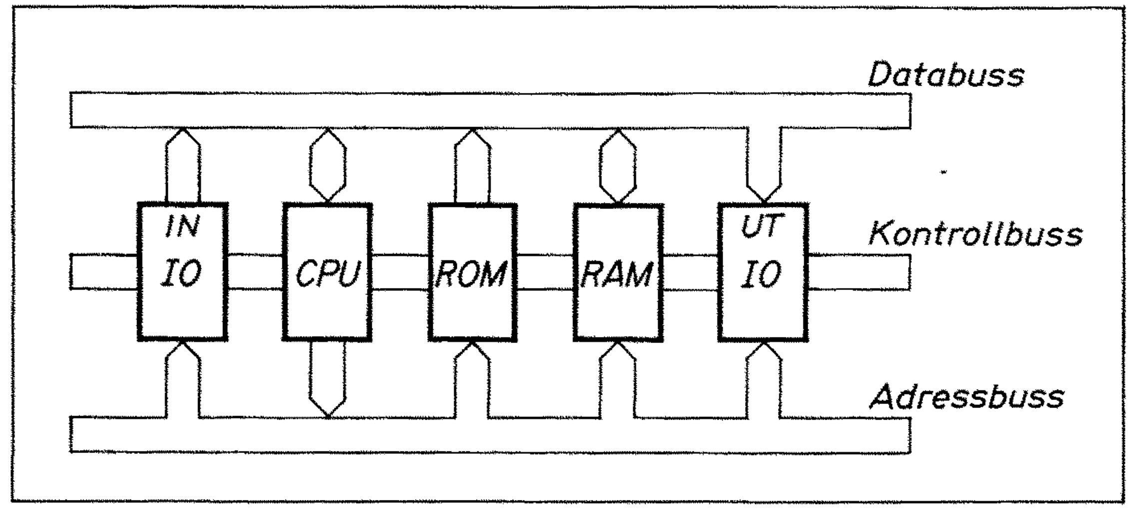


Fig 4.2 Blockschema för generell datorstruktur

1.3 Ett minimalsystem

Alla datorsystem är uppbyggda enligt den generella princip som visats i fig 4.2. Det som skiljer en stordator från en liten specialdator (dedicated computer) är bussarnas storlek och sammansättning samt antalet minnen och kringkretsar som är anslutna.

Vi har ju en speciell uppgift som ska lösas. Den kan lösas med hjälp av ett mycket litet system, exempelvis av en mikrodator i en enda kristallbricka (single chip computer). Exempel härpå är mikrodatorerna 8048 från Intel och Z8 från Zilog.

I nästa kapitel ska vi bygga upp ett kraftfullt datorsystem med stor minneskapacitet kring CPU:n Z80. Vi ska därför som förövning även bygga upp den enkla trafikstyrningen kring Z80. Fig 4.3.

Istället för de separata in- och utportarna i vår principdator i fig 3.5 använder vi i systemet i fig 4.3 en PIO-krets. Den innehåller två åttabitars portar (port A och port B) som var och en kan programmeras att fungera bl a som enbart utgång (mode Ø), enbart ingång (mode 1) eller en godtycklig bitkombination av in- och utgångar (mode 3).

För vår trafikstyrning behöver vi enbart använda en av portarna och vi programmerar därför port A att ha bitarna Ø-5 som utgångar och bitarna 6-7 som ingångar.

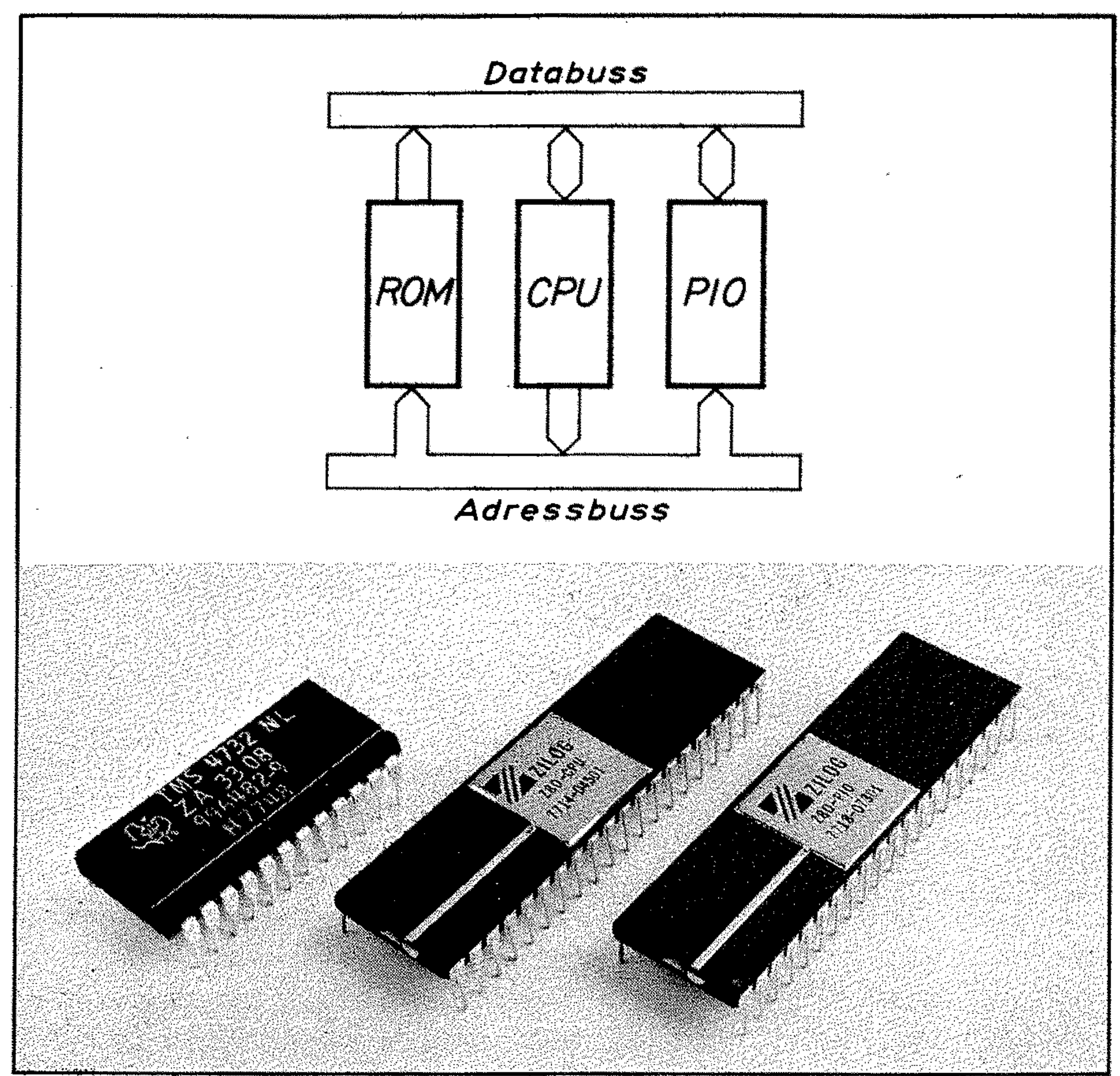


Fig 4.3 Minimalsystem för trafikstyrning
a. Blockschema
b. Foto på kretsarna (skala 1:1)

Z80-manualen ger ett exempel på ett "minimumsystem" som vi skulle kunna använda för vår trafikstyrning. Fig 4.4. Låt oss se hur det är hopkopplat.

Till vänster i fig 4.4 ser vi CPU:n Z80 och RESET-ingången. Genom att kortsluta kondensatorn (med en switch som tyvärr inte utritats i schemat) får vi en säker manuell nollställning fri från kontaktstudsar.

Ovanför CPU:n sitter en klocka som avger en fyrkantvåg (exempelvis 3 MHz). Om man inte behöver stabil frekvens, kan den byggas upp av ett par inverterare och en RC-krets. I annat fall kan RC-kretsen ersättas med en kristall och då får man en kristallklockas noggrannhet i frekvens.

Upptill höger ser vi kraftförsörjningen (power supply). Systemet kräver enbart 5 V om man inte använder ROM som fordrar andra matningsspänningar.

Adressbussen pekar på ett ROM innehållande 8K bit dvs 1K byte (1024x8 bitar). ROM:et har ytterligare två ingångar märkta $\overline{CE}_1$ och $\overline{CE}_2$. CE betyder chip select. Båda $\overline{CE}$-ingångarna måste vara låga för att kretsen (chip = kristallbricka) ska kunna adresseras (select). Man kan därför betrakta $\overline{CE}$-ingångarna som två extra adressingångar. Det intressanta för oss är att se hur de används:

$\overline{CE}_1$ är kopplad till $\overline{MREQ}$. ROM:et kan därmed enbart adresseras vid $\overline{MREQ}$-signal, dvs med minnesinstruktioner.

$\overline{CE}_2$ är kopplad till $\overline{RD}$. ROM:et kan därmed enbart adresseras för läsning (read). Ett ROM är ju också som vi sett tidigare enbart ett läsminne (read only memory).

Nu återstår endast parallellkretsen Z80-PIO. Den har två portar, port A och port B. Enligt schemat i fig 4.4 antas port A vara programmerad som utport och port B som inport. Men som vi sett ovan är portarnas användning helt en programmeringsfråga.

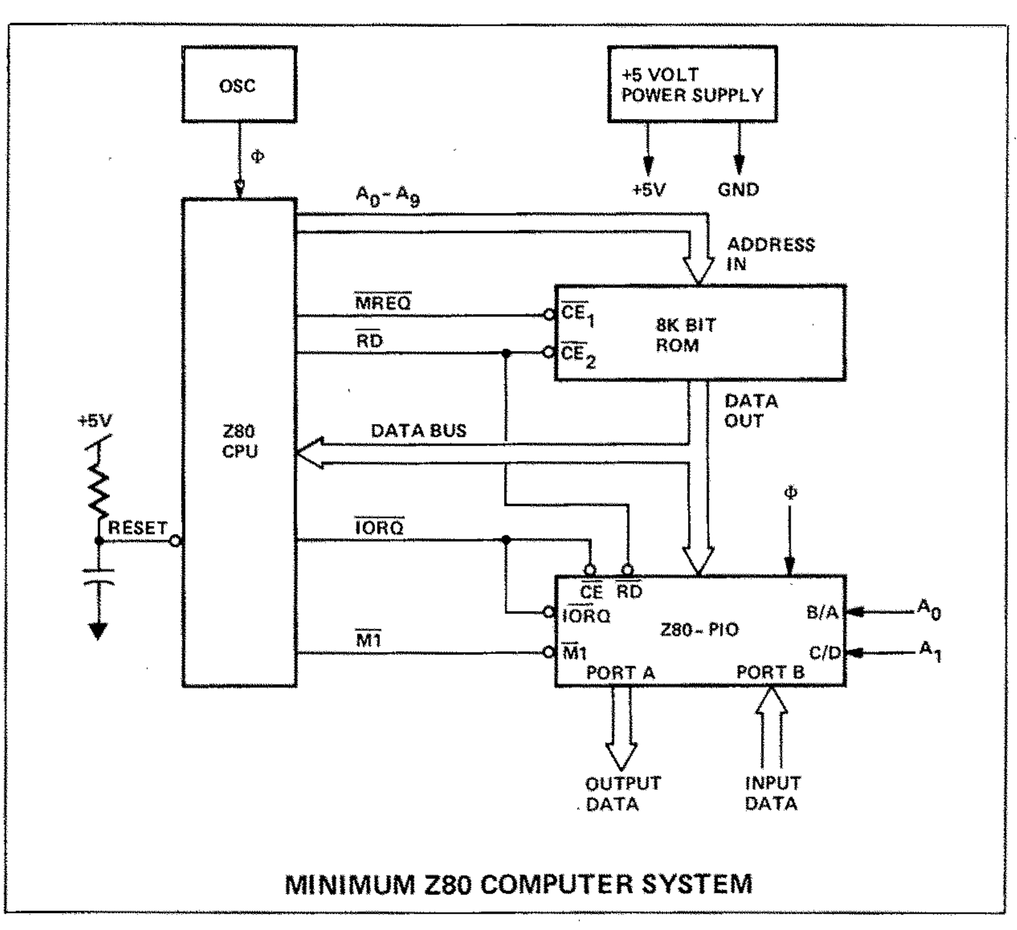


Fig 4.4 Minimalsystem med Z80 (Zilog)

Man frågar sig kanske här varför inte adressbussen är avkodad för adressering av PIO:n. Svaret är att Z80 använder separata IO-instruktioner. När en sådan instruktion exekveras läggs kontrolledningen $\overline{\text{IORQ}}$ låg. Eftersom inga andra IO-kretsar är inkopplade tycker man kanske att detta borde vara tillräckligt som adressering av PIO-kretsen. Nu måste vi emellertid komma ihåg att en PIO är en mycket komplex krets. Den innehåller två portar som vardera innehåller en åtta bitars datalatch. Portarna styrs från två kontrollregister. Det krävs fyra adresser för att skilja dessa åt. Vi ser också att av de åtta möjliga adressledningarna (AØ - A7) är PIO:n ansluten till två (AØ - A1). Med AØ bestämmer man vilken av de två portarna man vill komma i kontakt med och med A1 kan man välja kontrollregistret eller datalatchen. Ska vi använda port A (låg B/A) blir binära adressen till A-portens datalatch ØØ och binära adressen till motsvarande kontrollregister 1Ø.

Z80-PIO måste matas med ytterligare två signaler, klockan Ø och M1 (som anger att ny instruktion hämtas in).

1.4 Egen mikrodator eller MCB?

De kretsar som fordras för att bygga ett minimalsystem (fig 4.3) är relativt billiga. Ska man bygga upp ett enda eller några få system så blir det emellertid relativt dyrt att prova ut kopplingen, tillverka kretskort och felsöka (debug = avlusa).

Om vi är storfabrikanter av trafikljus lönar det sig givetvis att göra en egen och "optimal konstruktion" med minsta möjliga antal komponenter. Konstruktionskostnaderna kan ju då slås ut på ett stort antal producerade och likadana system och apparatkostnaden blir låg.

Om vi ska bygga ett enda styrsystem för att prova principen med mikroprocessorstyrning av trafikljus blir förhållandet annorlunda. Då lönar det sig att köpa en färdig mikrodator. Det finns många sådana system på marknaden. Vi väljer här den tidigare i fig 1.6 visade mikrodatorn Z80-MCB. Vi slipper därvid allt konstruktionsarbete och behöver enbart göra programmet (som sätts på kortet i form av ett ROM), ansluta in- och utgångar samt ansluta matningsspänningen (5 V). Vi ska inte reta oss på de kretsar som vi inte utnyttjar på MCB-kortet. För att kunna tillverkas i stora serier och därmed bli billigt görs MCB-kortet generellt användbart. Det innehåller därmed fler kretsar än som behövs för vårt lilla minimalsystem.

Fig 4.5 visar de kretsar i MCB Z80 som vi kommer att använda. Eftersom kortet är generellt fordras avkodade IO-kretsar. (Den

enkla avkodning som används för PIO-kretsen i fig 4.5 blockerar 64 av de 256 möjliga adresserna).

MCB-kortet innehåller buffertkretsar för både adress- och databuss. Högintegrerade kretsar tål inte stor belastning. Tack vare buffertarna (bus drivers) slipper man tänka på dessa problem.

När man använder ett MCB-kort får man givetvis rätta sig efter de adresser som fabrikanten utnyttjat för olika enheter. Port A ligger exempelvis på hexadressen D8H.

Det finns plats för fyra 16-benskapslar till vänster på MCB-kortet. Vi kan här koppla upp egna kretsar, exempelvis en buffert till PIO-kretsen eller löda på en sockel för anslutning av flatkabel till PIO-kretsen.

Vi har därmed bestämt hårdvaran för vårt system. Vi gick den bekväma vägen och valde ett färdigt MCB-kort. Nu återstår programmet.

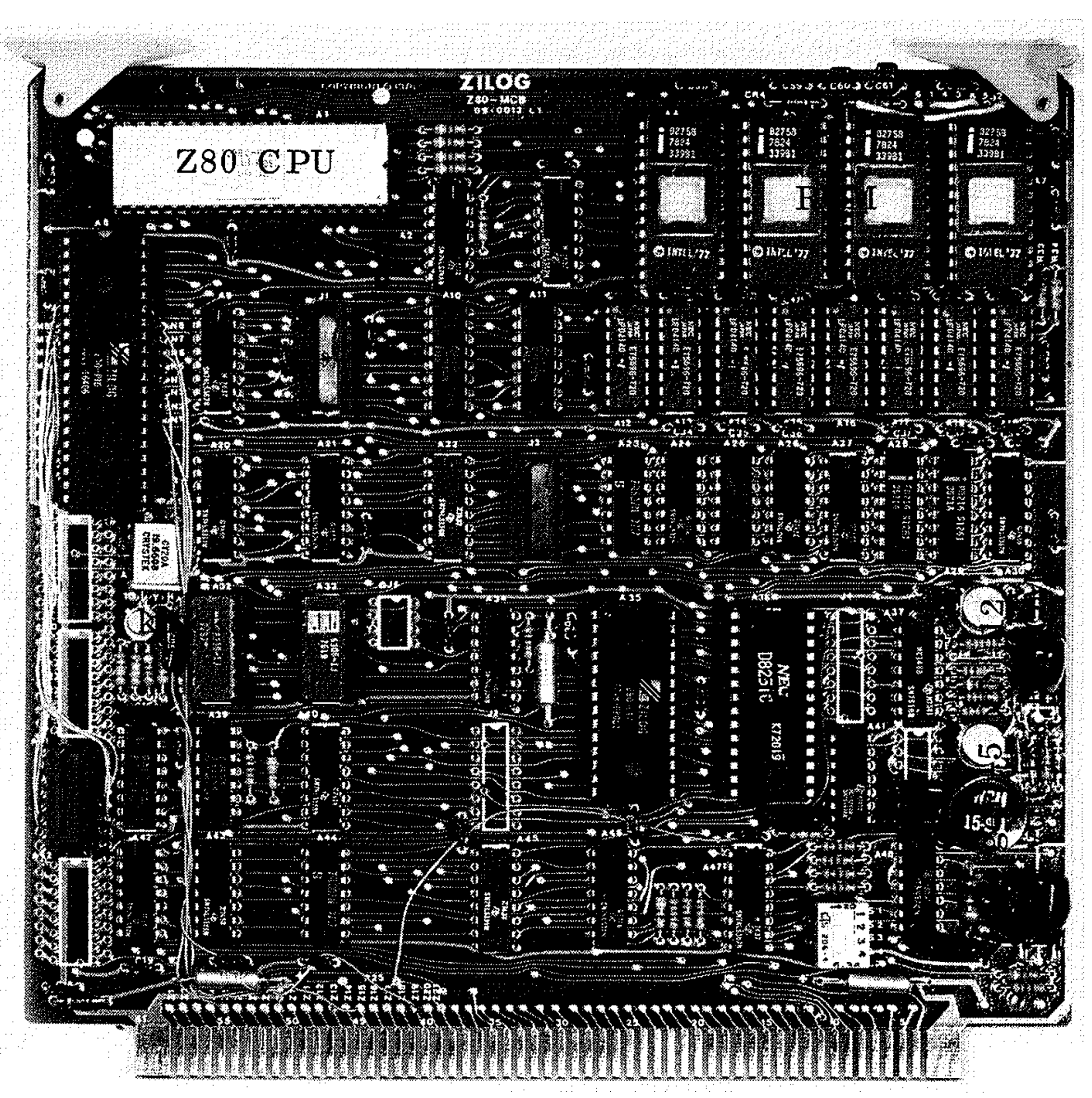


Fig 4.5 Z80-MCB

2. Mjukvaran

Hardware och software (hårdvara och mjukvara) är två viktiga begrepp. På svenska heter de maskinvara och programvara. Vi har tidigare i detta kapitel behandlat kretsarna för vårt system - dvs hårdvaran. Vi ska nu ge oss i kast med programmet, dvs mjukvaran. (Program dokumenteras ju normalt på papper och är därmed "mjuka" till skillnad från de "hårda" kretsarna).

Mikroprocessorn Z80 har 158 olika instruktioner. Många av dessa har ett flertal varianter. Många av Z80:s instruktioner är synnerligen kraftfulla och detta gör det möjligt att lösa komplexa uppgifter med få instruktioner och på kort tid. Vår avsikt är emellertid inte här att visa finesser och "smarta" program. Vi ska använda samma enkla typer av instruktioner som vi tidigare diskuterat. Har man väl lärt sig principen är det enkelt att gå vidare (med hjälp av programmeringsmanualen) och successivt använda allt mer avancerade instruktioner.

Vi ska nu bekanta oss med hur instruktionerna ska skrivas formellt korrekt. Det är en förutsättning för att vi ska kunna använda datorn för översättning av MEMO-koderna till maskinspråk (med ettor och nollor). Denna översättning kallas assemblering.

2.1 MEMO-koden för Z80

Det är fabrikanten (Zilog) som tillhandahåller assemblern. Det är därför också fabrikanten som specificerar hur MEMO-koden i detalj ska skrivas för att kunna köras i assemblern. Handboken "Z80-Assembly Language Programming Manual" ger oss en utförlig beskrivning.

Z80 har en mycket systematisk och enkel MEMO-kod. Vi har tidigare använt instruktioner av typen LDA (ladda ackumulatorn) och STA (lagra ackumulatorn). Z80 använder genomgående koden LD för alla dataförflyttningar (load, store, move etc).

LD A, B	betyder ladda register A (dvs ackumulatorn) med innehållet i register B.
LD C, 3FH	betyder ladda register C med hextalet 3F.
LD A, (3FØØH)	betyder ladda register A med innehållet i minnescellen med hexadressen 3FØØ (dvs 16 bitars adress).

Z80 har dessutom en rad instruktioner för förflyttning av 16 bitars ord. Exempelvis laddar instruktionen LD HL, (3FØØH) registerparet HL, som ofta används som adresspekare, med data från cellerna 3FØØH (till L) och 3FØ1H (till H).

Z80 har 12 olika instruktioner för IO-operationer. Vi använde tidigare i vår "principdator" LDA (215) för att överföra ingångssignalerna på adress 215 till ackumulatorn. När vi nu har en separat adressarea för IO-enheter måste vi använda instruktionen IN A, (n) för motsvarande operation. Med (n) menas innehållet i den IO-enhet som har adressen n (8 bitar). Motsvarande utinstruktion heter OUT (n), A. Den motsvarar vår tidigare instruktion STA (215).

LD, IN och OUT kallas operationskod eller kortare opkod (opcode).

A, B
A, (n)
(n), A } kallas operander

Observera att data flyttas från höger operand till vänster operand (från "source" till "destination"). I de tre uttrycken är de högra operanderna B, (n) resp A källor och de vänstra operanderna A, A resp (n) destinationer.

MEMO-koden för ovillkorligt hopp är JP nn. Det villkorliga hoppet, "hoppa om A=Ø", har MEMO-koden JP Z, nn. Beteckningen nn avser en 16 bitars hoppadress.

JP Z, nn har vi tidigare kallat "hoppa om A = Ø". Det är en grov förenkling, eftersom hoppet sker om nollflaggan är satt (Z = 1 i fig 3.25) och inte om ackumulatorn = Ø. Det är mera korrekt att tolka JP Z, nn som "hoppa om Z = 1". För att veta om Z = 1 måste vi kontrollera hur tidigare instruktioner behandlat nollflaggan. Det ligger alltså mycket information bakom varje MEMO-kod och vi måste gå till assemblermanualen för att få reda på hur den ska tolkas.

MEMO-kod ska man givetvis inte lära sig utantill. Arbetar man tillfälligt med en mikroprocessor slår man upp koden i manualen. Arbetar man mycket med en viss processor lär man sig koden automatiskt under arbetets gång.

Appendix A visar en sammanfattning av Z80:s 158 instruktioner och MEMO-koder.

Det väsentliga är att man är noga med detaljerna och formatet - och det lär man sig snart med hjälp av de felutskrifter man i annat fall får av assemblern!

Det program vi skriver med hjälp av MEMO-koder kallas källprogram (source program, source code). För att assemblern ska kunna tolka vårt program måste det skrivas i ett bestämt format. Assemblern är beredd att ta emot fyra grupper av information på varje rad. De engelska termerna för dessa grupper är:

LABEL	OPCODE	OPERANDS	COMMENT
LOOP:	LD	A, VALUE	; GET VALUE

Exemplet under grupprubrikerna visar hur datorn skiljer grupperna åt.

- o Label betyder etikett och används här istället för adress. I källprogrammet ingår inga absoluta adresser - som i våra tidigare programexempel. Vissa adresser är väsentliga för hopp av olika slag och då skriver vi en label istället för adress. Vi använde två lablar, "LJUS" och "VÄNTA", i vårt tidigare program utan att närmare kommentera deras funktion.

Vid assembleringen ger man assemblern ett kommando (pseudokod eller directive) som talar om för assemblern på vilken adress i minnet vi vill att programmet ska börja. Assemblern beräknar då automatiskt alla efterföljande adresser.

- o Assemblern observerar det kolon som avslutar labeln och vet därmed att nästa grupp är opkoden. (Om man inte har någon label i början av raden räcker det att göra ett eller flera mellanslag före opkoden).

  Opkoden måste utgöras av någon av de MEMO-symboler som ingår i Z80:s instruktionslista, appendix A.

- o Mellanslag anger att själva opkoden är slut och därmed kommer en grupp bestående av en eller två operander, åtskilda av kommatecken. Allt som har att göra med programmet är därmed avslutat.

  Vi har tidigare sett nyttan av en kommentar som vägledning för den som läser programmet. Ofta glömmer programmeraren själv bort vad hon (han) hade för avsikter med vissa instruktioner. Kommentarer är då värdefulla!

- o Assemblern tolkar ett semikolon (;) som en inledning till en kommentar. Kommentarer är enbart till för läsaren av källprogrammet (dvs som dokumentation) och behandlas ej av assemblern. Man kan börja en rad direkt med semikolon och då uppfattar assemblern hela raden som en kommentar.

- o Nedtryck av CR-tangenten (CR = carriage return) indikerar ny rad. Assemblern är därmed inställd på att ta emot en ny label.

Assemblerprogram - liksom all annan programvara - använder engelska termer och förkortningar. Det kan därmed vara befogat att återge den sida i assemblermanualen som behandlar formatet.

RULES FOR WRITING ASSEMBLY STATEMENTS (SYNTAX)

An assembly language program (source program) consists of labels, opcodes, operands, comments and pseudo-ops in a sequence which defines the user's program.

There are 74 generic opcodes (such as LD), 25 operand key words (such as A), and 694 legitimate combinations of opcodes and operands in the Z80 instruction set.

ASSEMBLER STATEMENT FORMAT:

Statements are always written in a particular format. A typical Assembler statement is shown below:

```
LABEL      OPCODE     OPERANDS  COMMENT
LOOP:      LD         HL,VALUE  ;GET VALUE
```

In this example, the label, LOOP, provides a means for assigning a specific name to the instruction LOAD (LD), and is used to address the statement in other statements. The operand field contains one or two entries separated by one or more commas, tabs or spaces. The comment field is used by the programmer to quickly identify the action defined by the statement. Comments must begin with a semicolon and labels must be terminated by a colon, unless the label starts in column No. 1.

Låt oss exemplifiera det ovan beskrivna assemblerformatet med några instruktioner:

```
LÄS:  IN A, (D8H)   ; SIGNAL FRÅN BIVÄG
      AND ØCØH      ; MASKA UT BIT 6 OCH 7
      JP Z, LÄS     ; HOPPA TILL LÄS OM A = Ø
      CALL LJUS     ; HOPPA TILL SUBRUTINEN LJUS
```

I ovanstående exempel har vi endast en label, LÄS. Övriga rader börjar med mellanslag (space) och därmed vet assemblern att dessa rader ej börjar med label.

AND-instruktionen har operanden CØH, där C är en hexsiffra. För att assemblern ska veta att operanden är en siffra (och inte en label) så måste alla tal börja med någon av siffrorna Ø-9. Vi skriver därför ØCØH istället för CØH.

Vi har ovan sett hur man ska skriva ett källprogram i en sådan form (eller med sådant format) att en assembler kan läsa det och assemblera det.

Även om vi för tillfället inte utnyttjar ett assemblerprogram utan gör assembleringen (dvs översättningen av MEMO-koder till binär- eller hextal och utskrivning av alla adresser) för hand så grundlägger det en god vana att alltid hålla sig till det korrekta assemblerformatet. Många program kan assembleras för hand och sedan laddas direkt i en mikrodator med hjälp av siffertangenter (oktal- elle hex). Vi har i fig 3.17 sett ett exempel på en terminal som kan användas på detta sätt.

En assembler är ett datorprogram. Det finns två typer av assemblers. Den ena typen är skriven i något lämpligt högnivåspråk (Fortran eller Basic) och kan laddas in i alla datorer som har det aktuella högnivåspråket. Assemblers av denna typ brukar kallas "cross assembler".

Mikrodatorn ABC80 kan exempelvis laddas med en crossassembler (från bandspelare eller floppy disk) och därmed användas för skrivning och assemblering av källprogram.

Konstruktörer som i sin dagliga verksamhet arbetar med mikroprocessorer använder speciella utvecklingssystem. Fig 4.6 visar ett exempel. Kärnan i systemet i fig 4.6 är en mikrodator uppbyggd kring Z80. I systemets programvara ingår olika hjälpmedel för arbete med källprogram, bl a en "editor" för skrivning och korrigering av text. Givetvis ingår även en assembler. Det skrivna källprogrammet lagras i form av en "textfil" och kan sedan assembleras med ett enkelt kommando. Denna typ av assembler kallas "resident assembler" och kan köras direkt i det givna utvecklingssystemet.

En assembler kontrollerar alla formella fel (syntax) och ger felutskrift på de rader där fel uppträder. Om programmet är formellt riktigt kan det direkt överföras till ett PROM via en i utvecklingssystemet inbyggd PROM-programmerare.

Textskrivning och korrigering med hjälp av editor, lagring av källkoder i form av textfiler samt assemblering är endast en bråkdel av de många arbetsuppgifter som enkelt kan klaras av med ett utvecklingssystem, fig 4.7.

Det tar förstås litet tid att lära sig ett utvecklingssystems editor och kommandon. Sedan man väl lärt sig systemet har man ett kraftfullt hjälpmedel till sitt förfogande för de flesta arbeten med mikrodatorsystem.

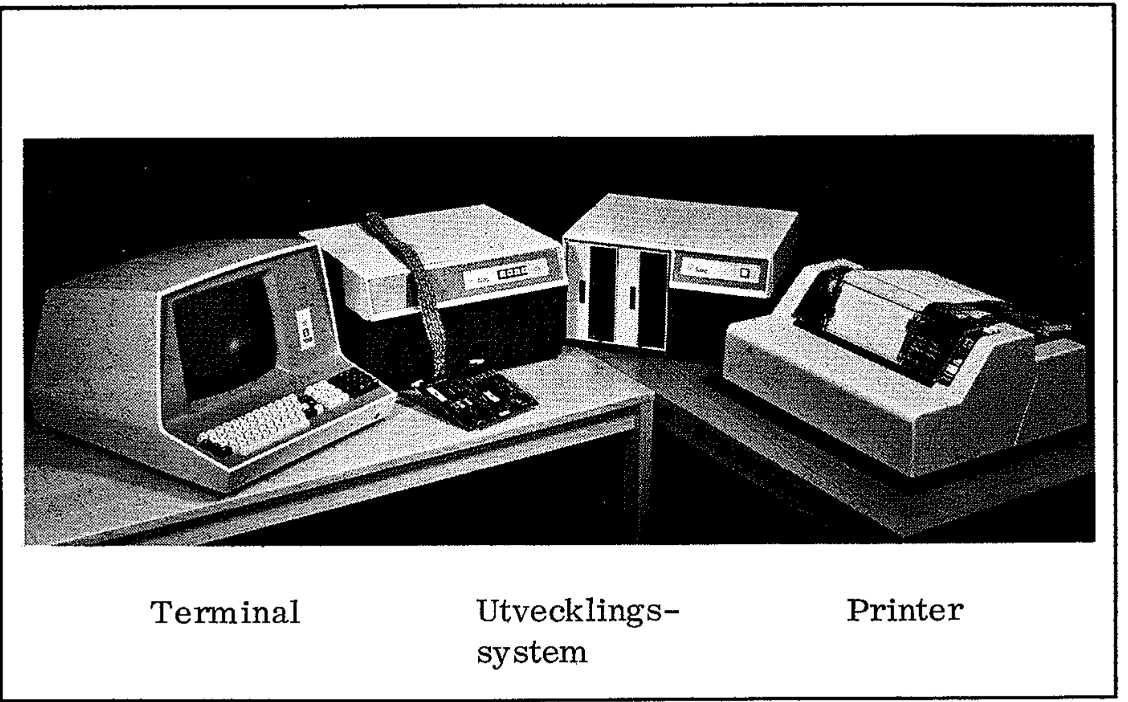


Fig 4.6 Utvecklingssystem för Z80

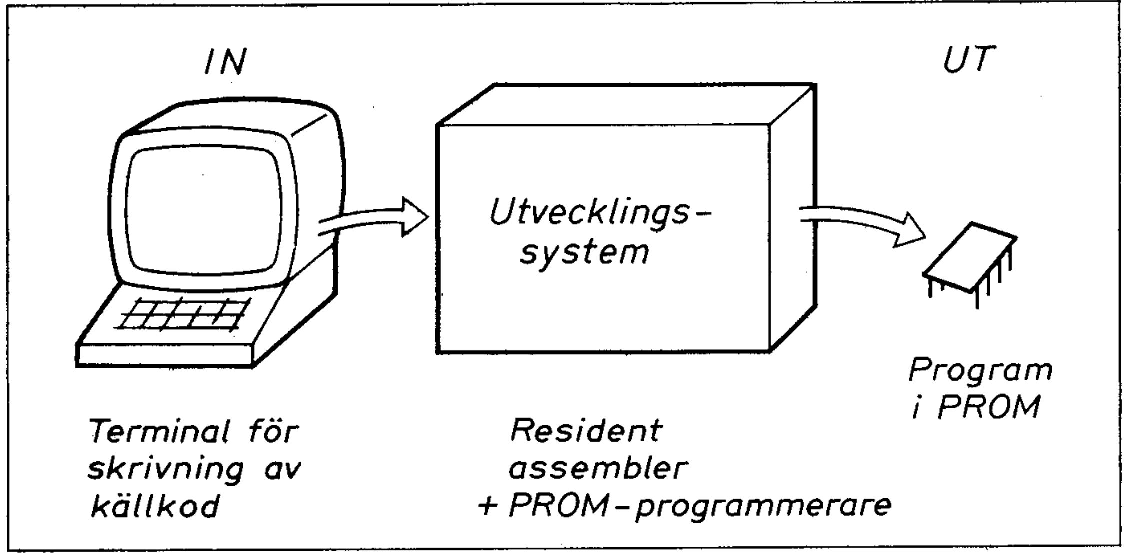


Fig 4.7 Programutveckling med utvecklingssystem

2.4 Ett enkelt källprogram

Källprogrammet (source program) i fig 4.8 är skrivet på terminalen till ett av Zilogs utvecklingssystem. Programmet är uppbyggt i fyra block med rubrikerna

- o Initiera PIO och sätt stackpekare
- o Huvudprogrammet
- o Subrutinen LJUS
- o Subrutinen WAIT

Frånsett det första blocket (initiera PIO och sätt stackpekare) följer programmet uppläggning fig 3.27. Subrutinen VÄNTA i fig 3.27 har fått namnet ändrat till WAIT beroende på att det amerikanska utvecklingssystemet inte accepterar bokstäverna Å, Ä och Ö. Men det är ju inget större problem för oss! Låt oss gå igenom programmets fyra block.

2.4.1 Adresser

Vid all programmering i lågnivå-språk (assemblerprogram och maskinprogram) måste man noga hålla reda på alla adresser. Vi måste alltså först ställa upp en minnesplan (address map).

På Z80-MCB finns det plats för fyra PROM med vardera 1K byte programminne. Avkodningen till de fyra PROM-hållarna blir följande:

0000 - 03FF	PROM-plats 1 (MMT monitorprogram)
0400 - 07FF	PROM-plats 2
0800 - 0BFF	PROM-plats 3
0000 - 0FFF	PROM-plats 4

(Om man vill använda terminalen i fig 3.17 för Z80-MCB måste man placera tillhörande monitorprogram på PROM-plats 1. Förutom detta krävs vissa extra ledningar på MCB-kortet och vissa kopplingar i kortkontakten).

På Z80-MCB, strax bredvid PROM-hållarna, sitter 4K byte RAM som avkodas för hexadresserna

1000 - 1EFF	(Användar-RAM)
1F00 - 1FFF	(Utnyttjas av MMT-monitorprogram)

På Z80-MCB finns både en parallellkrets (PIO) och en seriekrets (SIO). Vi ska enbart använda port A i PIO:n. Den har IO-adresserna

D8	PIO	DATA	port A
DA	PIO	Control	port A

I fig 4.9 visas de ovan nämnda adresserna utritade på en adresskarta (eller minnesplan). Observera att vi har två adressareor, en för minnesadresser och en för IO-adresser (Z80-MCB använder "IO-mapping").

```
COPY TRAFIKLJUS.S $TTY
; TRAFIKLJUSPROGRAMMET
; ********************

; Initiera PIO, port A och stackpekare
        LD      A,0FFH          ; Mod 3 till PIO A
        OUT     (0DAH),A        ; till kontrollreg.
        LD      A,11000000B     ; Bit 6 och 7 in
        OUT     (0DAH),A        ; till kontrollreg.
        LD      SP,1FFFH        ; Sätt stackpekaren

;HUVUDPROGRAMMET
        LD      A,21H           ; Normalljus
        OUT     (0D8H),A        ; t. PIO:s dataport
BIL?:   IN      A,(0D8H)        ; Läs PIO A
        AND     10000000B       ; Maska fram bit 7
        CALL    Z,LJUS          ; Om A=0: hoppa till sub
        JP      BIL?            ; Testa igen

; SUBRUTIN LJUS
LJUS:   LD      A,31H           ; Ljusbild 1
        OUT     (0D8H),A        ; till PIO A
        LD      A,04H           ; Antal sekunder
        CALL    WAIT            ; Hoppa till subrutin

        LD      A,0BH           ; Ljusbild 2
        OUT     (0D8H),A        ; till PIO A
        LD      A,04H           ; Antal sekunder
        CALL    WAIT            ; Hoppa till subrutin

        LD      A,0CH           ; Ljusbild 3
        OUT     (0D8H),A        ; till PIO A
        LD      A,16H           ; Antal sekunder
        CALL    WAIT            ; Hoppa till subrutin

        LD      A,0EH           ; Ljusbild 4
        OUT     (0D8H),A        ; till PIO A
        LD      A,04H           ; Antal sekunder
        CALL    WAIT            ; Hoppa till subrutin

        LD      A,19H           ; Ljusbild 5
        OUT     (0D8H),A        ; till PIO A
        LD      A,04H           ; Antal sekunder
        CALL    WAIT            ; Hoppa till subrutin

        LD      A,21H           ; Ljusbild 6
        OUT     (0D8H),A        ; till PIO A
        LD      A,10H           ; Antal sekunder
        CALL    WAIT            ; Hoppa till subrutin

        RET                     ; Tillbaka till huvudpgm

; SUBRUTIN WAIT

WAIT:   LD      D,A             ; Antal sek. till D-reg

LOOP:   LD      BC,50156D       ; Tidskonst. f. 1 sek.
LOOP1:  DEC     BC              ; Minska räknaren med 1
        PUSH    BC
        POP     BC
        LD      A,B             ; Hämta B-reg till A
        OR      C               ; OR:a med C-reg
        JP      NZ,LOOP1        ; Hoppa om ej=0

        DEC     D               ; Minska sekundräknaren
        JP      NZ,LOOP         ; Hoppa om ej=0
        RET                     ; Tillbaka till huvudpgm
```

Fig 4. 8 Trafikljusprogrammet (källprogram)

Hex-adress	Minne
1FFF	RAM 4KB
1ØØØ	
ØAØØ	PROM 4 1KB
Ø8ØØ	PROM 3 1KB
Ø4ØØ	PROM 2 1KB
ØØØØ	PROM 1 1KB

Hex-adress	IO
DA	PIO: Kontroll Port A
D8	Data Port A
Ø	

Fig 4.9 Minnesplan för Z80-MCB

2.4.2 Initiera PIO och sätt stackpekare

På MCB-kortet (fig 4.5) finns en PIO och vi ska som sagt använda port A både som utgång (bit Ø-5) och ingång (bit 6-7). För att kunna programmera en PIO-krets måste man noggrant studera manualen ("Z80-PIO, Zilog Product Specification" eller "Z80-MCB Hardware User's Manual"). Där kan man läsa att man väljer arbetsmod genom att ge kontrollregistret (i vårt fall adress DA) en av följande kontrollsignaler (hex-kod):

3F:	Mod Ø.	Enbart utgångar
7F:	Mod 1.	Enbart ingångar
BF:	Mod 2.	Dubbelriktade (bidirectional)
FF:	Mod 3.	Bitmod, dvs blandade in- och utgångar.

Vi ger alltså port A:s kontrollregister signalen FFH (mod 3). Enligt PIO-manualen förväntar sig nu kontrollregistret ytterligare en kontrollsignal som anger vilka bitar som ska vara inportar och utportar. Kontrollordets bitpositioner ska innehålla ettor för inportar och nollor för utportar.

I vårt exempel får kontrollsignalen hexvärdet CØ vilket framgår av fig 4.10.

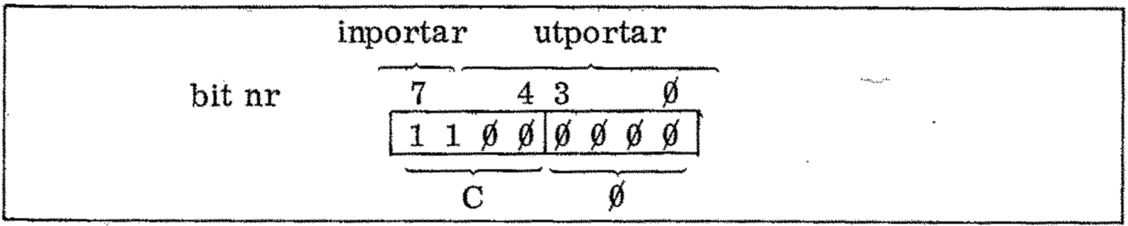


Fig 4.10 Kontrollsignal till PIO för inställning av bitarna i en inutport

Initieringen av PIO port A kan göras med följande programsekvens:

```
LD A,ØFFH        ; kontrollord nr 1 till A
OUT (ØDAH), A    ; A till PIO:s kontrollregister
LD A,ØCØH        ; kontrollord nr 2 till A
OUT (ØDAH), A    ; A till PIO:s kontrollregister
```

När man skriver källkoden på ett utvecklingssystem behöver man aldrig göra den talomvandling vi gjorde i fig 4.10. Som framgår av första blocket i fig 4.8 är det andra kontrollordet skrivet direkt i binär kod (vilket markeras med B):

```
LD A,    11ØØØØØØB
```

(Om man däremot ska ladda motsvarande program med handterminalen i fig 3.17 måste man givetvis omvandla binärtalet till hextal).

Nu är port A på PIO:n programmerad enligt våra önskemål. Det återstår emellertid ytterligare en väsentlig detalj innan vi kan börja det egentliga programmet. Vårt program ska innehålla subrutiner och då måste vi ha en stackpekare (stack pointer). Stackpekaren dekrementeras varje gång man lägger data på stacken. Vi ska därför sätta stackpekarens begynnelseadress högst upp i RAM-minnet. Vi väljer enligt minnesplanen (fig 4.9) hexadressen 1FFF. (Eftersom stackpekaren dekrementeras innan man lagrar data på stacken hade vi här kunnat välja RAM-adressen 2ØØØ som ligger omedelbart ovanför RAM-arean).

Med instruktionen

```
LD SP,1FFFH
```

laddar vi stackpekaren direkt med den önskade adressen.

Det är lätt gjort att glömma bort stackpekaren när man skriver assemblerprogram! Programmet "spårar då ur" vid första subrutin och man förvånas över att det i övrigt välskrivna programmet inte fungerar. Det är som sagt detaljerna som är viktiga vid all programmering!

2.4.3 Initiering

Att vara "initierad" betyder att man är insatt i den aktuella frågeställningen. En dator kan bli initierad (initialized) om vad den ska utföra vid initieringen. Vi har just sett exempel på en sådan initiering. Den gav PIO:n order om vilka bitar som skulle vara in- respektive utgångar och vidare ställdes stackpekaren på lämplig adress.

Finessen med programmerbara kretsar (som PIO:n) är att man kan ställa om deras funktion med några instruktioner i programmet. Man behöver inte löda!

Typiskt för de flesta programsystem är att de börjar med en initiering. Efter RESET vet ju inte alla kringkretsar hur de ska fungera och därför fordras alltid initiering efter start (eller återstart). Vid initieringen "sätter man upp" systemet till önskad "konfiguration". Ja, det finns många sätt att uttrycka sig på! Nu har vi i varje fall sett exempel på en initiering eller INITA som det ofta förkortas i programmets kommentarer.

2.4.4 Huvudprogrammet

Huvudprogrammet i fig 4.8 avviker inte från det typprogram vi tidigare diskuterat. Om vi skulle be fem programmerare skriva programmet (efter det i fig 3.27 specificerade flödesschemat) skulle vi få fem olika program! Det är det som gör programmeringen till en skön konst!

Subrutinen LJUS

Även subrutinen LJUS känner vi igen från vårt tidigare typprogram (Avsnitt 4.2 i kap 3). En van programmerare skulle förmodligen göra en "loop" eller slinga för att slippa uppräkningen av de sex snarlika ljusbilderna i subrutinen. Det finns ett stort antal "smarta" instruktioner man skulle kunna utnyttja och därmed göra programmet kort (billigare PROM). Ska en produkt massproduceras är detta väsentliga faktorer.

I de allra flesta fall är dock den utan jämförelse väsentligaste punkten att man skriver programmet så att det är lätt att förstå och lätt att ändra. Programmet ska vara väl dokumenterat. Det bör vara försett med tydliga kommentarer och uppdelat på lätt urskiljbara subrutiner.

Man talar ofta om "moduluppbyggda program" (dvs fristående programmoduler som kan bearbetas separat) och "strukturerad programmering" (ett diffusare begrepp som bl a avser god dokumentering och en lättförstådd struktur).

Subrutinen WAIT

Subrutinen WAIT (fig 4.8) ger oss en fördröjning med det antal sekunder som anges av ackumulatorns innehåll. Subrutinen börjar med att flytta över antalet sekunder från ackumulatorn till D-registret. Vi får därmed ackumulatorn ledig. (Vi ska strax använda OR-funktionen och den arbetar alltid med ena talet i ackumulatorn).

Först laddas registerparet BC (som tillsammans utgör ett 16 bitars register) med decimaltalet 50156. Det sker i Z80 med en enda instruktion

```
LD BC, 50156D
```

Vi använder här ett decimaltal (betecknas med D) för att slippa omräkningar. Talet ska vara så stort att den efterföljande loopen (LOOP1) tar precis en sekund att fullborda.

LOOP1 börjar med att registerparet BC dekrementeras (DEC BC). Nu följer instruktionerna PUSH BC och POP BC. Deras enda uppgift är att förlänga exekveringstiden för LOOP1 och de inverkar inte på funktionen i övrigt (om de placeras i rätt ordningsföljd). Därefter laddas B-registret till ackumulatorn. Och nu kommer den aviserade OR-funktionen:

OR C

ger bitvis OR-funktion mellan ackumulatorn (som har samma innehåll som B-registret) och C-registret. Finns det nu någon enda etta i C-registret eller ackumulatorn så kommer den ettan med som resultat av OR-funktionen. (Resultatet placeras i ackumulatorn). Endast i det fall att både C-registret och ackumulatorn (dvs B-registret) innehåller enbart nollor får vi värdet noll som resultat. Vi kan alltså på detta sätt avkänna nollställning i ett 16 bitars register.

Efterföljande instruktion är "hoppa om A ≠ 0" med MEMO-koden ≠

JP NZ, LOOP1

Vi hoppar tillbaka till adressen LOOP1 om och om igen, ända tills dekrementeringen har reducerat innehållet i registerparet BC till noll.

Nu är det bara att slå upp i assembler-manualen och se hur lång tid som ett varv tar i loopen. För bekvämlighets skull finns tiden angiven i mikrosekunder under förutsättning att man har 4 MHz-klocka.

DEC BC	1,5 μs
PUSH BC	2,75 μs
POP BC	2,5 μs
LD A, B	1,0 μs
OR C	1,0 μs
JP NZ, LOOP1	2,5 μs
Summa	11,25 μs

Z80-MCB har 2,457 MHz-klocka. Vid denna klockfrekvens tar LOOP1 tiden 19,94 μs. För 1 s krävs $10^6/19,94 = 50156$ varv i LOOP1. Det är orsaken till att vi ovan laddade BC-registret just med talet 50156D.

Den yttre loopen (LOOP) dekrementerar antalet sekunder (som ligger i D-registret) för varje fullbordad sekundcykel hos LOOP1. När antalet sekunder räknats ner till noll sker återhopp med "return"-instruktionen RET.

Därmed har vi gått igenom källkoden till trafikstyrningsprogrammet.

2.5 Källprogrammet assemblerat

Källprogrammet i fig 4.8 (som efter skrivning ligger lagrat på en floppy-disk i utvecklingssystemet) kan nu med ett enkelt kommando assembleras. Man får då utskriften i fig 4.11. (Vi förutsätter här att alla fel har korrigerats).

Till vänster ser vi adresserna (från begynnelseadressen ØØØØ hex) och maskinkoden utskriven som hextal. Den tredje kolumnen innehåller ett R för vissa positioner.

Vårt program ska ha startadressen noll för då startar programmet automatiskt vid RESET. Ofta vill man emellertid ha programmet inlagt på andra adresser. Då anger man med ett enkelt kommando på vilken adress man vill ha programmet placerat och vilka ytterligare programmoduler som ska ingå. Utvecklingssystemets "linking loader" sätter då automatiskt ihop de olika programdelarna och beräknar samtliga adresser. Hoppadresser (som markerats med R = relokerbar) måste därvid omräknas av utvecklingssystemet.

Den som arbetar med ett utvecklingssystem har så småningom samlat ihop ett helt "bibliotek" med programmoduler. Tillsammans med tillverkarens programbibliotek utgör detta en erfarenhetsbank för framtida konstruktioner. Med utvecklingssystemets hjälp behöver konstruktören endast skriva kortare huvudprogram som utnyttjar tidigare utvecklade subrutiner. Att arbeta på detta sätt ger givetvis en helt annan säkerhet och bättre ekonomi än om alla program alltid skulle skrivas på nytt (från"scratch"). Kostnaden är här väsentlig eftersom man ofta räknar med att en programmerare i snitt skriver en (eller några få) instruktioner per timme. Då inräknas givetvis tiden för fel ("bugs") och felsökning ("debugging" = avlusning).

Typiskt för utvecklingen på datorområdet är att hårdvaran tack vare de integrerade kretsarna blir allt billigare. Mjukvaran blir däremot dyrare allteftersom kraven på komplexa funktioner ökar. Hjälpmedel som subrutinbibliotek och utvecklingssystem för bekväm programutveckling blir därför alltmera väsentliga.

2.6 Ett professionellt skrivet program

Som tidigare påpekats finns det lika många "programmeringsstilar" som det finns programmerare. Man kan alltså inte säga att ett visst program är mera professionellt än ett annat. Fig 4.12 visar en version av trafikljusprogrammet som skrivits av en van programmerare.

Programmet kan användas för självstudier (med hjälp av assemblermanualen). Men stoppa inte upp här om det tar emot. "Smarta" lösningar kan ofta vara svåra att förstå och de hör inte till livets väsentligheter.

```
                                    TRAFIKLJUS                          PAGE    1
LOC     OBJ CODE M STMT SOURCE STATEMENT                                ASM 5.0

                      1   ; TRAFIKLJUSPROGRAMMET
                      2   ; ********************
                      3
                      4   ; Initiera PIO, port A och stackpekare
0000    3EFF          5           LD      A,0FFH          ; Mod 3 till PIO A
0002    D3DA          6           OUT     (0DAH),A        ; till kontrollreg.
0004    3EC0          7           LD      A,11000000B     ; Bit 6 och 7 in
0006    D3DA          8           OUT     (0DAH),A        ; till kontrollreg.
0008    31FF1F        9           LD      SP,1FFFH        ; Sätt stackpekaren
                     10
                     11   ;HUVUDPROGRAMMET
000B    3E21         12           LD      A,21H           ; Normalljus
000D    D3D8         13           OUT     (0D8H),A        ; t. PIO:s dataport
000F    DBD8         14   BIL?:   IN      A,(0D8H)        ; Läs PIO A
0011    E680         15           AND     10000000B       ; Maska fram bit 7
0013    CC1900   R   16           CALL    Z,LJUS          ; Om A=0: hoppa till sub
0016    C30F00   R   17           JP      BIL?            ; Testa igen
                     18
                     19   ; SUBRUTIN LJUS
0019    3E31         20   LJUS:   LD      A,31H           ; Ljusbild 1
001B    D3D8         21           OUT     (0D8H),A        ; till PIO A
001D    3E04         22           LD      A,04H           ; Antal sekunder
001F    CD5000   R   23           CALL    WAIT            ; Hoppa till subrutin
                     24
0022    3E0B         25           LD      A,0BH           ; Ljusbild 2
0024    D3D8         26           OUT     (0D8H),A        ; till PIO A
0026    3E04         27           LD      A,04H           ; Antal sekunder
0028    CD5000   R   28           CALL    WAIT            ; Hoppa till subrutin
                     29
002B    3E0C         30           LD      A,0CH           ; Ljusbild 3
002D    D3D8         31           OUT     (0D8H),A        ; till PIO A
002F    3E16         32           LD      A,16H           ; Antal sekunder
0031    CD5000   R   33           CALL    WAIT            ; Hoppa till subrutin
                     34
0034    3E0E         35           LD      A,0EH           ; Ljusbild 4
0036    D3D8         36           OUT     (0D8H),A        ; till PIO A
0038    3E04         37           LD      A,04H           ; Antal sekunder
003A    CD5000   R   38           CALL    WAIT            ; Hoppa till subrutin
                     39
003D    3E19         40           LD      A,19H           ; Ljusbild 5
003F    D3D8         41           OUT     (0D8H),A        ; till PIO A
0041    3E04         42           LD      A,04H           ; Antal sekunder
0043    CD5000   R   43           CALL    WAIT            ; Hoppa till subrutin
                     44
0046    3E21         45           LD      A,21H           ; Ljusbild 6
0048    D3D8         46           OUT     (0D8H),A        ; till PIO A
004A    3E10         47           LD      A,10H           ; Antal sekunder
004C    CD5000   R   48           CALL    WAIT            ; Hoppa till subrutin
                     49
004F    C9           50           RET                     ; Tillbaka till huvudpgm
                     51
                     52   ; SUBRUTIN WAIT
                     53
0050    57           54   WAIT:   LD      D,A             ; Antal sek. till D-reg
                     55
0051    01ECC3       56   LOOP:   LD      BC,50156D       ; Tidskonst. f. 1 sek.
0054    0B           57   LOOP1:  DEC     BC              ; Minska räknaren med 1
0055    C5           58           PUSH    BC
0056    C1           59           POP     BC
0057    78           60           LD      A,B             ; Hämta B-reg till A
0058    B1           61           OR      C               ; OR:a med C-reg
0059    C25400   R   62           JP      NZ,LOOP1        ; Hoppa om ej=0
                     63
005C    15           64           DEC     D               ; Minska sekundräknaren
005D    C25100   R   65           JP      NZ,LOOP         ; Hoppa om ej=0
0060    C9           66           RET                     ; Tillbaka till huvudpgm
```

Fig 4.11 Utskriften av det assemblerade trafikljusprogrammet

```
                                   TRAFIK                                   PAGE   2
LOC     OBJ CODE M STMT SOURCE STATEMENT                                     ASM 5.0

                       7  *E
                       8  ; Initiera PIO, port A
0000    3EFF           9            LD      A,0FFH            ; Mod 3
0002    D3DA          10            OUT     (0DAH),A          ; till PIO kontrollreg.
0004    3E80          11            LD      A,80H             ; Bit 7 input
0006    D3DA          12            OUT     (0DAH),A          ; till PIO kontrollreg.
0008    0ED8          13            LD      C,0D8H            ; Adress t. PIO dataport
                      14
000A    212C00    R   15  OMIGEN:   LD      HL,LJUSTAB        ; Pekare för OUTI
000D    EDA3          16            OUTI                      ; Ge normalljus
000F    ED78          17  BIL?:     IN      A,(C)             ; LÄs PIO A o sÄtt flaggor
0011    E20F00    R   18            JP      PO,BIL?           ; Loopa till 1 bit Ändras
                      19
0014    0606          20  LJUS:     LD      B,6               ; Antal ljusbilder
0016    EDA3          21  LBILD:    OUTI                      ; NÄsta ljusbild
0018    28F0          22            JR      Z,OMIGEN          ; Klart efter 6 bilder
001A    7E            23            LD      A,(HL)            ; HÄm antal sek.
                      24
001B    DD210000      25  SEKUND:   LD      IX,0              ; Tidskonst. 1 sekund
001F    110100        26  LOOP:     LD      DE,1              ; StegvÄrde
0022    DD19          27            ADD     IX,DE             ; Startv. + stegvÄrde
0024    30F9          28            JR      NC,LOOP           ; Loopa till > FFFFH
0026    3D            29            DEC     A                 ; Sekund-rÄknare
0027    20F2          30            JR      NZ,SEKUND         ; Loopa till klart
0029    23            31            INC     HL                ; Peka pÅ nÄsta ljusbild
002A    18EA          32            JR      LBILD             ; och kör vidare
                      33
002C                  34  LJUSTAB:DB   21H,31H,4H,0BH,4H,0CH,16H
0033                  35            DB   0EH,04H,19H,4H,21H,10H
                      36  ; (Normalljus + ljusbild + tid , ljusbild + tid, osv.)
```

Fig 4.12 Ett trafikljusprogram skrivet av en van programmerare

Sammanfattning

I detta kapitel har vi knutit samman maskinvara och programvara. Vi kan inte skriva program förrän hårdvaran är bestämd. Och vi kan inte utforma hårdvaran utan att samtidigt tänka på hur arbetsuppgifterna programmässigt ska lösas. Ofta har man ett val: Man kan lösa ett delproblem antingen med kretsar (wired logic) eller med en programrutin (programmed logic). Vi har exempelvis använt en programrutin för fördröjning - vi hade lika väl (och kanske bättre) kunnat utnyttja en på MCB-kortet inbyggd räknare.

Typiskt för allt arbete med mikrodatorsystem är just det ömsesidiga beroendet mellan hårdvara och mjukvara (eller för att prata mera svenskt mellan maskinvara och programvara). Det duger inte längre att vara enbart programmeringsspecialist eller enbart kretsdesigner. För att arbeta med mikrodatorsystem krävs både och.

5. Elektroniken i ABC80

I föregående fyra kapitel har vi behandlat ett speciellt mikrodatorsystem (dedicated microcomputer). Det utförde en specifik uppgift, trafikljusstyrningen. Fördelen med mikrodatorstyrningen var bl a att man enkelt kunde modifiera funktionen. Systemet var speciellt så tillvida att ingångarna bestod enbart av switchar och utgångarna enbart av lampor. Systemets uppgift begränsades till att visa olika ljuskombinationer vid olika insignaler.

I detta kapitel ska vi bekanta oss med kretsarna i mikrodatorn ABC80 som är ett generellt användbart mikrodatorsystem. Vi ska börja med frågan om vad som kännetecknar ett generellt system. Vi ska sedan - med utgångspunkt från blockschemat - studera de viktigaste blocken i ABC80-systemet.

Vi kommer i detta kapitel i kontakt med en rad olika kretsar och en mängd signaler. I många fall ger vi oss nu in på detaljer - det är nödvändigt för förståelsen av ABC80:s många kontrollsignaler. Låt dig ej avskräckas av signalernas mångfald! Det är först nu elektroniken blir verkligt spännande! Om vi bara har en grundförståelse för de kretsar vi gick igenom i kapitel 2 kan vi med gott mod ge oss in på samspelet mellan kretsar och signaler. Det viktigaste är kanske nu att vänja sig vid och hålla reda på vissa typer av signaler och hur de påverkar olika funktioner i ABC80.

Vissa saker tar emot när man första gången läser om dem. Det gäller då att inte fastna på detaljerna. Först måste man vänja sig vid orden (alla signalbenämningar exempelvis). Sedan kan man steg för steg lära sig funktionen och sammanhanget. Detta kapitel är alltså inte avsett att streckläsas utan mera att avsmakas bitvis. Då först kan man riktigt avnjuta alla de enkla och fina kretslösningar som ingår i ABC80.

1. Ett generellt system

Ett generellt system fungerar efter samma enkla principer som vårt trafikstyrningssystem. Ett generellt system ska emellertid kunna behandla mer komplexa signaler, exempelvis siffror, bokstäver och

grafiska symboler. Ett sådant system måste innehålla ett stort minne och kunna byggas ut med yttre massminnen, exempelvis bandminnen och skivminnen ("diskar").

Systemet bör dessutom ha en buss som är åtkomlig. Via bussen kan systemet byggas ut godtyckligt.

Ett generellt system måste ha en praktiskt användbar systemprogramvara så att det är lätt att arbeta med. I denna programvara bör ingå ett (eller flera) högnivåspråk och enkelt användbara rutiner för att skriva, lagra och exekvera program.

Såväl beträffande maskinvara som programvara är ABC80 ett generellt datorsystem. Fig 5.1.

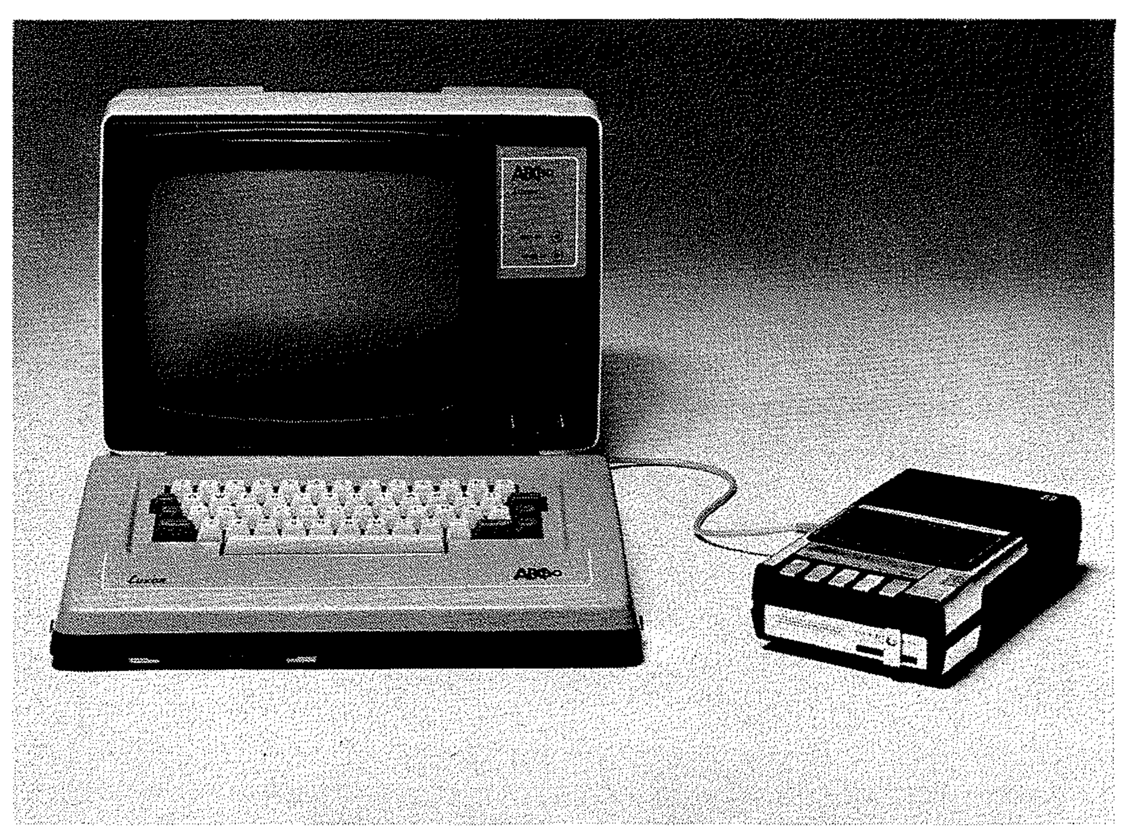


Fig 5.1 ABC80
1. Videomonitor med strömförsörjning
2. Mikrodator med tangentbord
3. Kassettbandspelare

2. Blockschemat

I detta kapitel ska vi studera en del av elektroniken i ABC80. Komponenterna som till större delen utgörs av halvledarkretsar sitter monterade på fyra kretskort. Fig 5.2. För att vi ska få en översikt över systemets funktion innan vi ger oss i kast med alla kretsarna ska vi börja med ett blockschema.

Fig 5.2 Elektroniken i ABC80

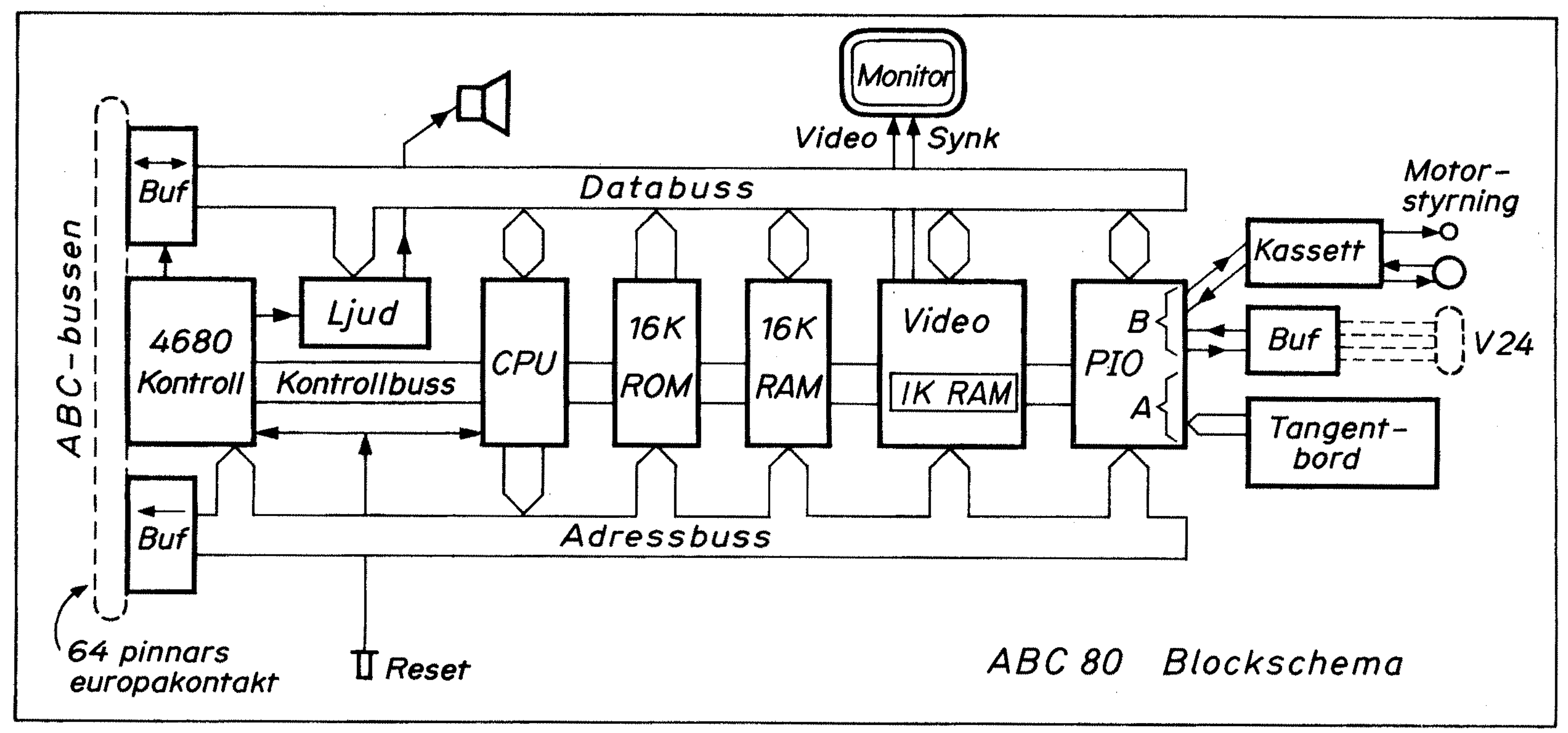


Fig 5.3 Blockschema för ABC80

I fig 5.3 återges blockschemat för ABC80. Schemat är snarlikt det blockschema vi tidigare diskuterat i anslutning till fig 4.2.

I fig 5.3 - liksom tidigare i vår principdator - är det CPU:n som har kontrollen över bussarna. Via adressbussen pekar CPU:n ut de minnesceller eller IO-enheter den vill komma i kontakt med för läsning eller skrivning.

ROM utgör ett fast programminne på 16K byte. Det innehåller alla nödvändiga rutiner samt dessutom tolken (interpretatorn) till programspråket Basic (Beginners All-purpose Symbolic Instruction Code).

RAM är ett dynamiskt läs-skrivminne på 16K byte. Det kan användas både för programlagring och datalagring.

LJUD är en utport som styr en ljudgenerator. Vi kan alltså få datorn att "låta", dvs avge olika ljud.

VIDEO är en IO-enhet som avkodar bildminnet och matar bildskärmen med signaler så att vi kan se innehållet i bildminnet.

Bildminnet är ett statiskt RAM på 1K byte. CPU:n kan både skriva och läsa i detta bildminne och detta samtidigt som bildminnets innehåll presenteras på bildskärmen av videoenheten.

Längst till höger på bussen (fig 5.3) ser vi en PIO. Den känner vi igen från tidigare! I fig 5.3 används PIO:n till tre olika uppgifter.

- o Port A tar emot signalerna från tangentbordet.
- o Bitarna 5, 6 och 7 i port B går till kassettinterfacet. Bit 5 driver ett start/stopp-relä. Bit 6 är utgång och bit 7 ingång (från kassett till CPU).
- o Bitarna Ø-4 i port B kan användas för ett V24-snitt. V24 (eller RS-232C) är ett internationellt standardiserat snitt för anslutning av exempelvis terminaler (fjärrskrivmaskiner). I V24-snittet är bitarna Ø-2 ingångar och 3-4 utgångar. Med V24-snittet (och tillhörande programvara) kan ABC80 användas som terminal.

Längst till vänster på bussen (fig 5.3) ser vi drivkretsar till omgivningen. Adressbussens drivkretsar är enkelriktade (utåt från ABC80). Databussens drivkretsar är dubbelriktade (bidirectional) och inriktningen styrs av CPU:n.

Vissa ledningar avkodas och bildar (tillsammans med adress- och dataledningar) en speciell yttre buss som kallas "ABC-bussen". Det är en generell buss och till denna buss finns ett komplett byggsystem av färdigbyggda moduler för olika uppgifter. Upp till 64 olika IO-enheter kan anslutas till denna buss. På detta sätt kan ABC80 utgöra kontrollenhet för ett mätsystem, ett ordbehandlingssystem och många liknande system.

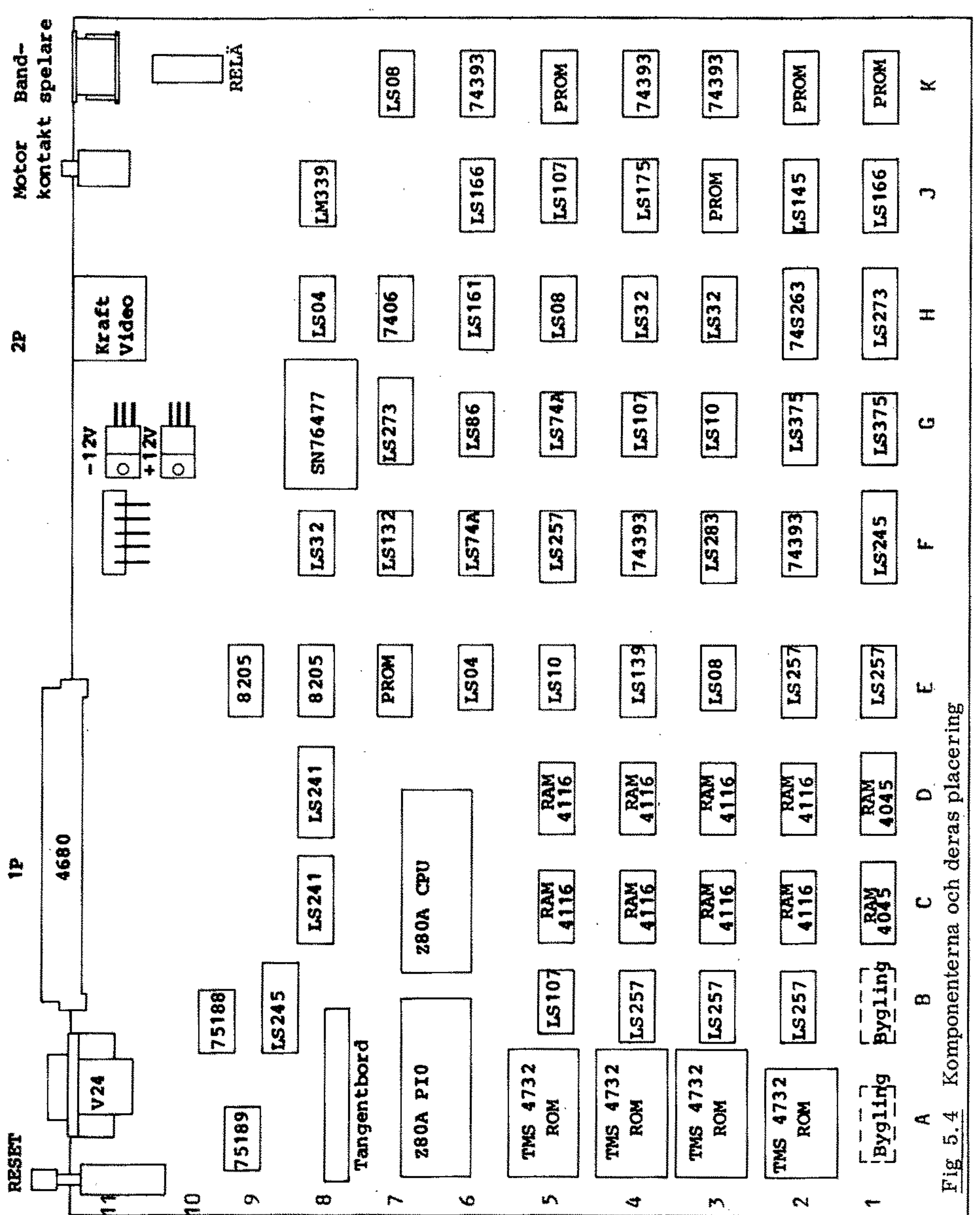


Fig 5.4 Komponenterna och deras placering

Som vi sett tidigare måste man vara noga med adresserna för både minnen och IO-enheter. Fig 5.6 visar en adressplan för minnen och IO-enheter. För att minska kostnaderna är avkodningen av IO-adresser mycket förenklad. Exempelvis avkodas PIO-kretsen enbart med bit fyra (hög). Med bitarna ∅ och 1 väljs därefter A- eller B-port respektive kontroll- eller dataregister.

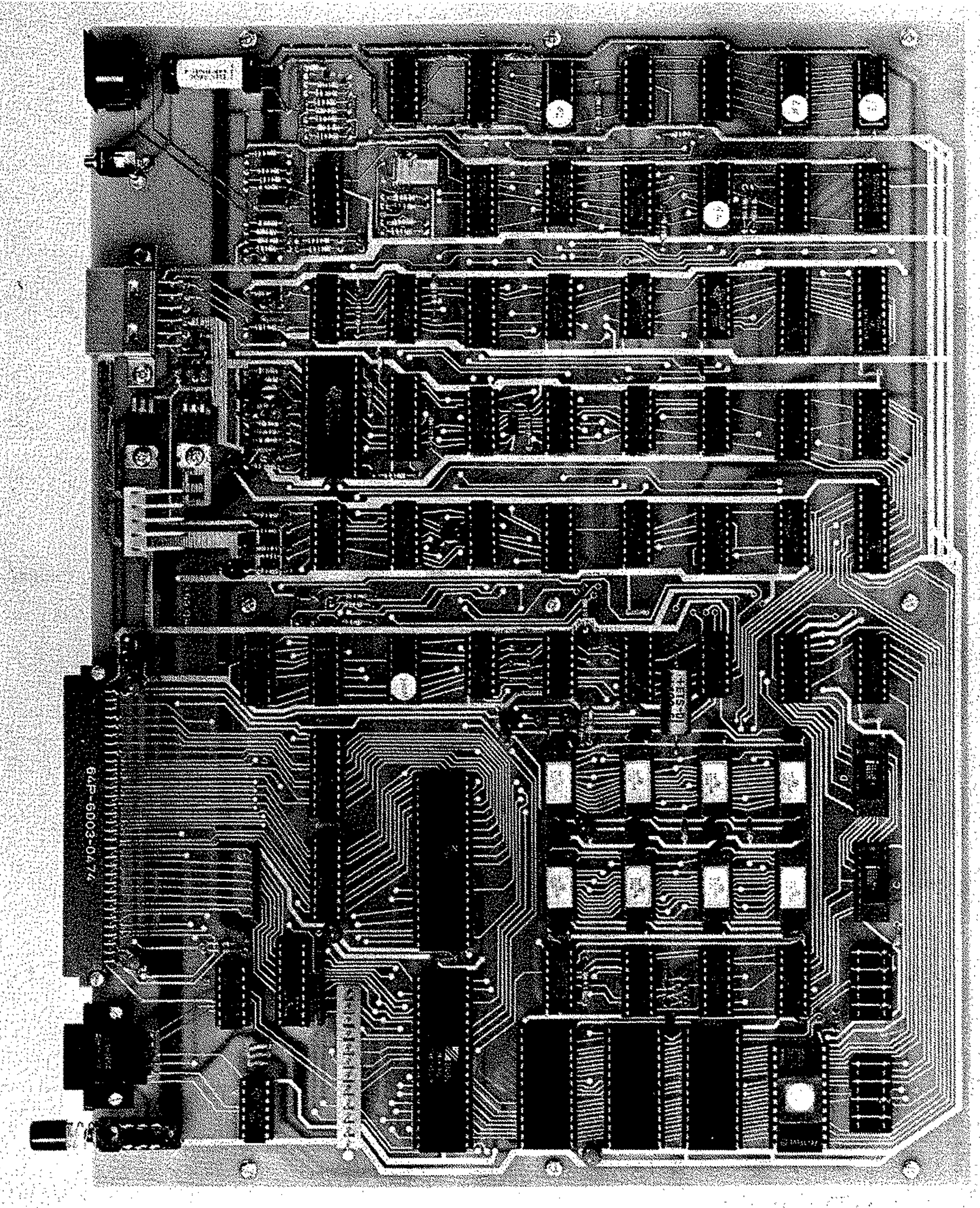


Fig 5.5 Foto på ABC80:s kretskort

I systemprogramvaran används följande adresser för PIO:n

PIO:n		
hexadress:	funktion:	
38	data	port A
39	kontroll	port A
3A	data	port B
3B	kontroll	port B

Minne

Hex-adress (decimal)	Användning		
FFFF (65535)	RAM 16KB — Stack och användarprogram		16K
CØØØ (49152)	RAM 16KB — Externt minne (option)		16K
8ØØØ (32768)	RAM 1KB — Bildminne		16K
7CØØ (31744)	ROM 1KB — Printer (option)		
78ØØ (30720)	1KB — Fritt (ROM eller RAM)		
74ØØ (29696)	ROM 1KB — IEC (option)		
7ØØØ (28672)	ROM 4KB — FLOPPY (option)		
6ØØØ (24576)	ROM 8KB — Fritt		
4ØØØ (16384)	ROM 16KB — BASIC interpreter m m		16K
ØØØØ			

In - utportar

PIO

Hex-adress (decimal)	Användning
3B (59)	kontroll port B
3A (58)	data port B
39 (57)	kontroll port A
38 (56)	data port A

IN-portar

Hex-adress (decimal)	Användning
7	Reset
1	Status IN
Ø	Data IN

UT-portar

Hex-adress (decimal)	Användning
6	Ljudgenerator
5	Kommando C4
4	Kommando C3
3	Kommando C2
2	Kommando C1
1	Kanalval
Ø	Data ut

Fig 5.6 Adressplan för minnen och IO-portar i ABC80

Utöver dessa adresser finns det alltså ytterligare ett antal adresser (alla med bit 4 hög) som pekar på PIO:n. Eftersom dessa adresser inte utnyttjas för andra IO-enheter uppstår aldrig någon tvetydighet.

Ljudgeneratorn styrs av en latch med adressen 6.

ABC-bussen använder enligt fig 5.6 nio IO-adresser: 3 inportar och 6 utportar.

Vi har ovan bekantat oss med ABC80:s blockmässiga uppbyggnad. Vi ska nu gå igenom ABC80:s minnen, ljudgenerator, PIO-kretsen och busskontakten. I nästa kapitel ska vi behandla tengentbordet, videoenheten, kassett-interfacet och ABC-bussen.

3. Minnen

Z80 kan med sina 16 adressledningar adressera 64K byte minne. I fig 5.6 har vi sett hur denna adressarea används. Ofta talar man om "höga adresser" och "låga adresser". I fig 5.6 har adressplanen därför ritats med adressen noll nertill och "högsta adress" upptill. (Ibland ritas adressplaner med högsta adress nedåt och då kan det bli tvetydigt att tala om högsta adress).

ABC80:s minnesarea (till vänster i fig 5.6) innehåller fyra block om vardera 16K.

De övre två blocken är avsedda för RAM-minnen: Det översta blocket (adresserna CØØØH-FFFFH) utgörs av ett i ABC80 inbyggt dynamiskt RAM. Det näst översta blocket (adresserna 8ØØØH-BFFFH) är reserverat för den som vill ansluta ett yttre RAM ("option") via busskontakten.

De två nedre 16K-blocken är - så när som på 1K bildminne - avsedda för permanent programlagring i ROM.

Det lägsta 16K-blocket innehåller systemprogrammen (Basic-interpreter).

Det näst lägsta 16K-blocket är uppdelat i 6 mindre grupper som alla är reserverade för speciella ändamål. Den övre IK-gruppen inom detta block används som nämnts tidigare för ett inbyggt bildminne. Det utgörs av ett statiskt RAM om 1K byte.

Vi ska nedan beskriva de olika typer av minnen som ingår i ABC80. För permanent programlagring används ROM eller EPROM och för datalagring med snabb access används statiska RAM för bildminnet och dynamiska RAM för arbetsminnet.

ABC80 har som nämnts tidigare en interpretator eller tolk för högnivåspråket Basic. Tillsammans med olika rutiner för hantering av video m m ligger denna tolk lagrad i fyra kapslar om vardera 4K byte. Dessa ROM-kapslar utgörs normalt av maskprogrammerade ROM av typen 4732 (Texas) och sitter i socklar till vänster på kretskortet i fig 5.5. Under en utvecklingsperiod kan man använda raderbara EPROM (2532). Man har då möjlighet att radera dessa med UV-ljus och programmera dem på nytt.

Utvecklingen på minnesområdet går snabbt och socklarna har plats för 8K ROM. Genom att byta ut de fyra ROM:en (på vardera 4K byte) mot tre stycken 8K ROM (dvs totalt 24K byte) får man plats med en mer avancerad Basic i en senare version. Att på detta sätt ha funktionen placerad i ROM som lätt kan bytas gör ABC80 utvecklingsbar. Program i ROM brukar kallas "firmware" eftersom det är ett mellanting mellan "software" och "hardware".

Vissa ROM-adresser är reserverade för yttre kringutrustningar, exempelvis ett IEC-bussinterface för mätändamål, ett "floppy-disk"-interface och en printer. Dessa ROM byggs in i interfacekretsarna på de yttre IO-enheterna.

Fig 5.7 visar uttagsplaceringen (pin configuration) för läsminnena TMS 2532 (EPROM) och TMS 4732 (ROM). Observera att dessa minnen är "ordorganiserade", dvs de innehåller samtliga 8 bitar för varje adresserat ord.

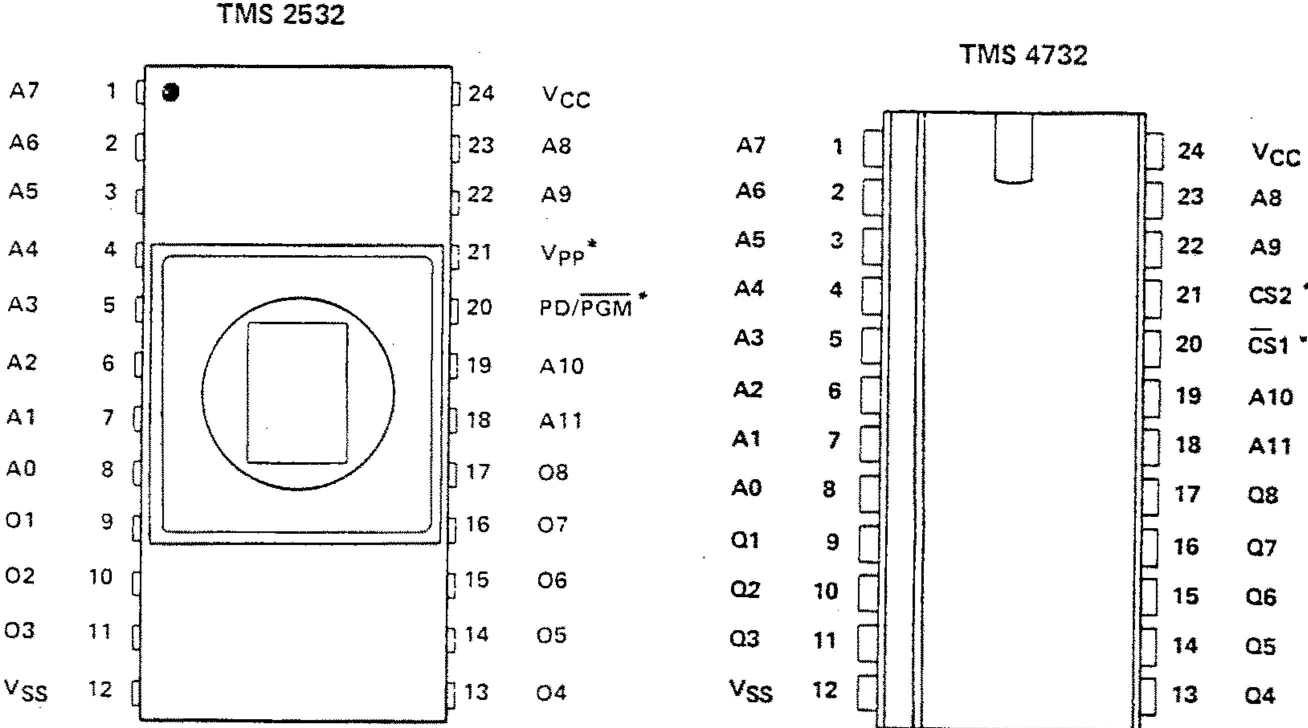


Fig 5.7 4K byte EPROM och ROM med samma funktion

Adressering

Adresseringen av ett minne sker dels med adressingångar och dels med kapselval (chip select). Fig 5.8 visar i princip hur de fyra ROM-kapslarna i ABC80 adresseras.

Varje kapsel innehåller 4K ord och därför har den 12 adressingångar. Eftersom fyra kapslar ska arbeta parallellt krävs det dessutom ett kapselval och det görs med $\overline{CS}$-ingången. För detta val används en avkodare (en av fyra) som matas med adressbitarna A12 och A13. Det är kapsel E4 på kretskortet (fig 5.5).

De fyra ROM-kapslarna upptar endast ett av de fyra 16K-blocken i adressplanen. Därför måste en "blockavkodning" göras med adressbitarna A14 och A15. I ABC80 har detta utförts med hjälp av ett PROM (position E7) och en MUX (position F5) som vi ska återkomma till i anslutning till fig 5.10. Blockavkodningen styr (som framgår av fig 5.8) den tidigare omnämnda "en av fyra"-avkodarens enable-ingång.

Det krävs ytterligare en avkodning som utelämnats i fig 5.8. När CPU:n vill läsa i minnet lägger den ut $\overline{MREQ}$ (för att skilja mellan minnesadress och IO-adress) och $\overline{RD}$ (för att skilja mellan skriv och läs). Dessa två signaler kombineras (AND) i ABC80 till signalen $\overline{MRD}$. Vi återkommer till detta i fig 5.10.

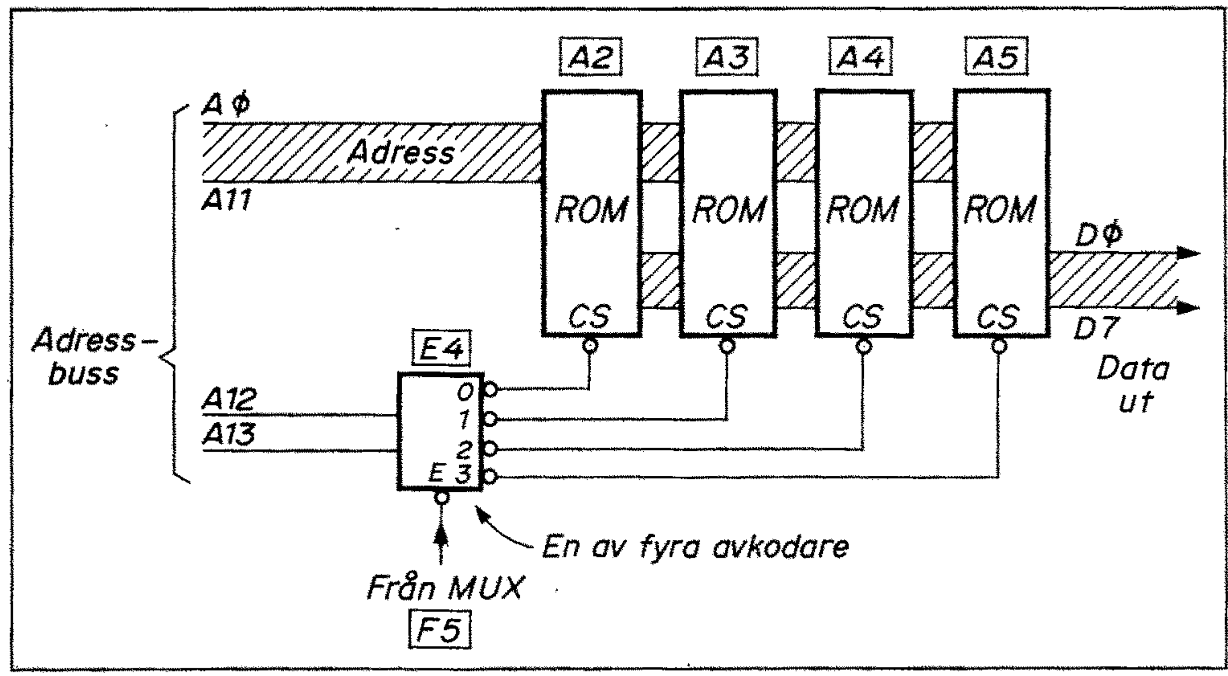


Fig 5.8 Adressering av ROM på ABC80

ABC80 innehåller 16K byte dynamiskt minne och 1K byte statiskt minne. Låt oss först beskriva det statiska bildminnet.

3.2.1 Statiskt RAM

Bildskärmen innehåller 24 rader och varje rad rymmer 40 tecken (characters). Eftersom ett tecken kräver en byte vid lagring i ASCII-kod krävs ett bildminne på 1K byte, dvs 8K bit. Här har man valt ett statiskt minne för ABC80.

Den vanligaste (1978) statiska RAM-kapseln heter 2102. Den innehåller 1K bit organiserade i 1024 ord om 1 bit. Det skulle tydligen krävas 8 kapslar av 2102 för att bygga upp ABC80:s bildminne.

För att minska antalet kapslar har man valt det statiska minnet 4045 som innehåller 4K bit organiserade i ord om 4 bitar. Med två kapslar får man tydligen den erforderliga kapaciteten på 1K byte. Som framgår av fig 5.9 har 4045-kapseln 18 ben. De används för 10 adressingångar, 4 data in/utgångar, skriv/läskontroll ($\overline{WE}$), chip select ($\overline{CS}$) samt matningsspänning och jord.

Användningen av ett statiskt RAM är enkel i princip: Man lägger på tre typer av signaler

- o adress (10 bitar för 1K)
- o kapselval ($\overline{CS}$ = chip select eller $\overline{CE}$ = chip enable)
- o skriv/läskontroll

Bildminnet i ABC80 kan dels användas för skrivning och läsning av CPU:n och dels för matning av data till bildskärmen. Det senare sker utan att CPU:n behöver vänta. För att åstadkomma detta har man i ABC80 använt speciella kretslösningar som vi inte här ska detaljstudera. $\overline{WR}$-signalen tillförs minneskapslarnas $\overline{WE}$-ingångar via en MUX (position B2) som styrs av de ovan nämnda kretsarna.

PIN CONFIGURATION

Ben	Signal	Ben	Signal
1	A_6	18	V_{CC}
2	A_5	17	A_7
3	A_4	16	A_8
4	A_3	15	A_9
5	A_0	14	I/O_1
6	A_1	13	I/O_2
7	A_2	12	I/O_3
8	$\overline{CS}$	11	I/O_4
9	GND	10	$\overline{WE}$

2114

LOGIC SYMBOL

A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 — I/O_1 I/O_2 I/O_3 I/O_4 — WE CS

PIN NAMES

A_0–A_9	ADDRESS INPUTS	V_{CC} POWER (+5V)	
$\overline{WE}$	WRITE ENABLE	GND GROUND	
$\overline{CS}$	CHIP SELECT		
I/O_1–I/O_4	DATA INPUT/OUTPUT		

Fig 5.9 RAM-minnet (Intel 2114 eller Texas 4045)

RAM-minnet upptar 1K byte och avkodningen av adressbitarna A10-A15 är något knepigare än den tidigare visade ROM-avkodningen i fig 5.8. I ABC80 har man löst adresseringen med hjälp av ett bipolärt PROM som är anslutet till adressbitarna A10-A15. Fig 5.10 visar principen.

Detta PROM avkodar samtliga 1K-kombinationer. Man kan därför programmera en av dess fyra utgångar att ge utsignal enbart för den 1K-grupp som avser bildminnet. En av de övriga utgångarna är så programmerad att den ger utsignal för samtliga 1K-grupper som utpekar det interna RAM:et på 16K. Ytterligare en utgång används för avkodning av externt minne.

En MUX är ansluten till två av utgångarna på PROM:et. MUX:en styrs av CPU-signalerna $\overline{MREQ}$ och $\overline{RD}$. Från MUX:en kommer den tidigare omtalade kontrollsignalen $\overline{MRD}$, den tidigare omtalade blockavkodningen till ROM-avkodaren i fig 5.8 samt dessutom signalerna $\overline{XM}$ och $\overline{XIN}$. $\overline{MRD}$ är som nämnts en kombination av CPU-signalerna $\overline{MREQ}$ och $\overline{RD}$. $\overline{XM}$ och $\overline{XIN}$ är kontrollsignaler som vi snart ska återkomma till.

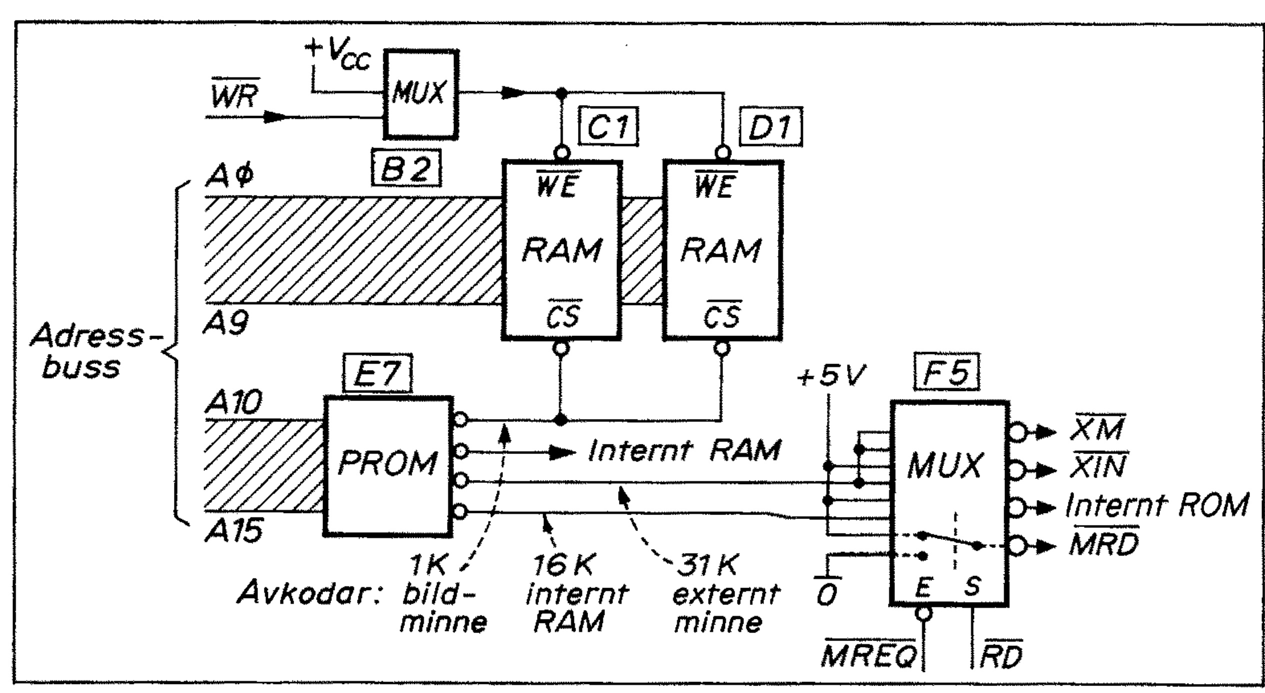


Fig 5.10 Adressering av bildminnet

3.2.2 Dynamiskt RAM

På ABC80:s kretskort sitter åtta RAM-kapslar (4116) som vardera innehåller 16384 bitar (16K bit) dynamiskt skriv/läsminne. Kapslarna är "bitorganiserade", dvs varje kapsel innehåller endast en bit på varje adress. För att få ett ord om åtta bitar måste man alltså använda åtta kapslar på det sätt vi tidigare sett i fig 2.27.

Två nya problem möter oss när vi ska använda dynamiska RAM-minnen av typ 4116 och liknande. Det ena är "multiplexing" av adressledningar och det andra är "refreshing".

Multiplex av adressledningar

För att adressera 16K behövs 14 adressledningar ($2^{14} = 16384$). Kapseln tar en stor del av kostnaden för en halvledarkrets och det gäller därför för fabrikanten att använda 16 bens standardkapslar och undvika dyra kapslar med många uttag. 16 ben är inte tillräckligt för alla adressledningar (inklusive $\overline{CS}$ och $\overline{WE}$) och matningsspänningar. För att kunna använda prisbilliga kapslar "multiplexas" adressingångarna på många dynamiska minnen. Principen framgår av fig 5.11.

Inkommande adressledningar (14 st för 16K byte) är i fig 5.11 uppdelade i två grupper, AØ-A6 respektive A7-A13. Multiplexern MUX bestämmer vilken av dessa grupper som ska tillföras de 7 adressingångarna på vårt dynamiska RAM. (I fig 5.11 är endast två av de 8 RAM-kapslarna utritade).

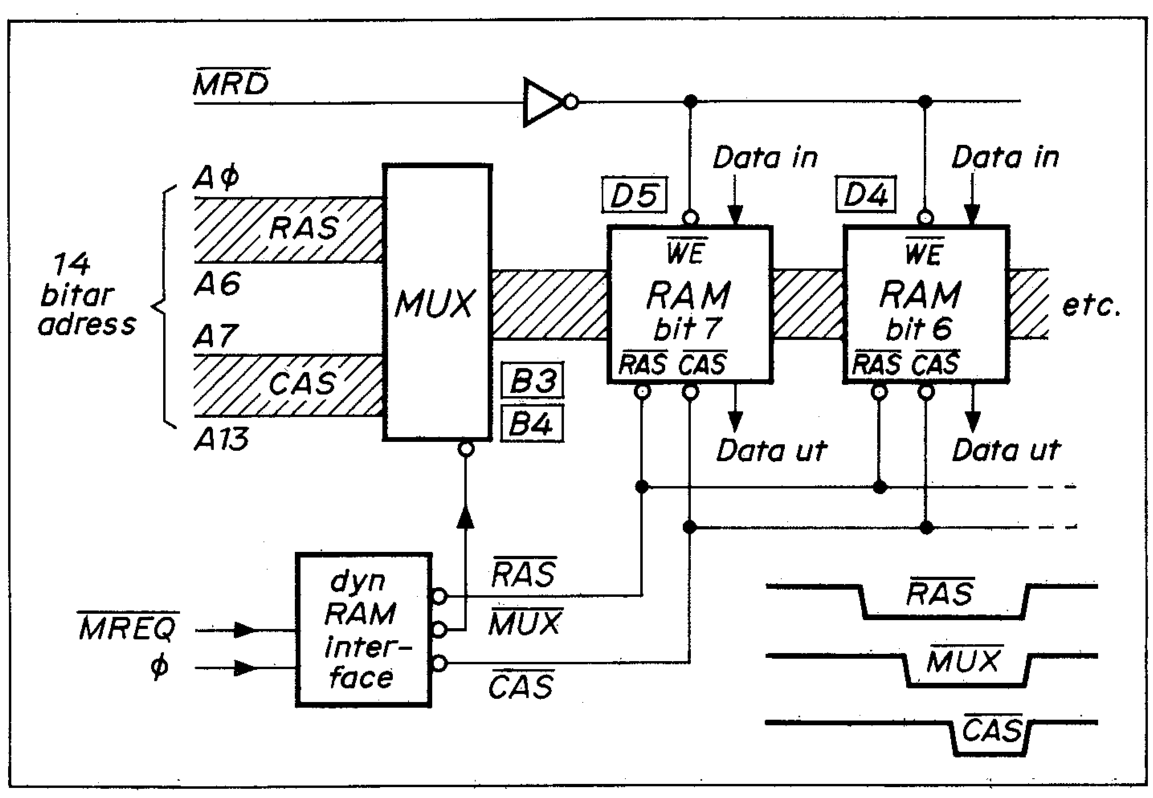


Fig 5.11 Principen för multiplex av adressledningar

Förutom adress är det två signaler som vanligen styr ett minne. $\overline{MREQ}$ talar om att en minnesinstruktion exekveras och $\overline{RD}$ visar om det är läsning från ($\overline{RD}$ låg) eller skrivning till minnet. (I ABC80 är $\overline{MRD}$ en kombination av dessa två signaler).

Eftersom adresserna i vårt fall ska multiplexas krävs det fler kontrollsignaler och därför fordras en speciell "RAM-interface-krets". Den är utritad som ett block nere till vänster i fig 5.11 och matas med $\overline{\text{MREQ}}$ och klockan Ø. Interfacekretsen är så utformad att den i rätt tidsföljd avger tre signaler,

$\overline{\text{RAS}}$ = radadresstrob (Row Address Strobe)
$\overline{\text{MUX}}$ = multiplexersignal
$\overline{\text{CAS}}$ = kolumnadresstrob (Column Address Strobe)

Utformningen av ett sådant interface kan göras på många sätt och beskrivs i "application notes" från minnesfabrikanterna (Exempelvis Zilog Application Note: Interfacing 16 Pin Dynamic RAMs to the Z80 Microprocessor).

Nere till höger i fig 5.11 ser vi exempel på tidsdiagram för $\overline{\text{RAS}}$, $\overline{\text{MUX}}$ och $\overline{\text{CAS}}$.

Först kommer alltså $\overline{\text{RAS}}$ (triggas direkt av $\overline{\text{MREQ}}$) och därmed inläses adresserna AØ-A6 i minneskapseln och lagras i en intern adresslatch. Därefter kommer $\overline{\text{MUX}}$-signalen som lägger om multiplexern så att adresserna A7-A13 tillförs minneskapseln.

Slutligen (när signalerna på ledningarna hunnit stabiliseras) kommer $\overline{\text{CAS}}$-signalen och då läses adresserna A7-A13 in till minneskapseln.

Kontrollsignalen $\overline{\text{MRD}}$ inverteras och tillförs minnets $\overline{\text{WE}}$-ingång och bestämmer därigenom om adresserad bit ska läsas ($\overline{\text{MRD}}$ låg) av CPU:n eller skrivas in i minnet. För många styrpulser liknande $\overline{\text{MRD}}$ är tiderna (timing) kritiska. Invertering och frånslag av $\overline{\text{MRD}}$-pulsen sker därför i ABC80 med en klockad vippa och inte på det enkla sätt (via en inverterare) som visas i principschemat i fig 5.11.

I fig 5.11 har vi utelämnat en väsentlig detalj, blockavkodningen (eller chip select). PROM:et i fig 5.10 var programmerat att avge signalen "internt minne" för alla 1K-grupper som avser det dynamiska RAM:et. Denna signal skulle normalt tillföras RAM-kapslarnas $\overline{\text{CS}}$-ingångar. Nu räcker inte benen till för $\overline{\text{CS}}$-ingångar på 4116 (se fig 5.12). Därför körs $\overline{\text{RAS}}$ och $\overline{\text{CAS}}$ genom två signalgrindar (NAND-grindar, position E5) som styrs av signalen "internt minne" från PROM-avkodaren. Även dessa detaljer (om än så väsentliga) är utelämnade i fig 5.11.

Fig 5.12 visar uttagsplacering (pin connections) för ett minne av typ 4116. Dynamiska minnen av denna typ kräver tre matningsspänningar, +12 V, +5 V och -5 V.

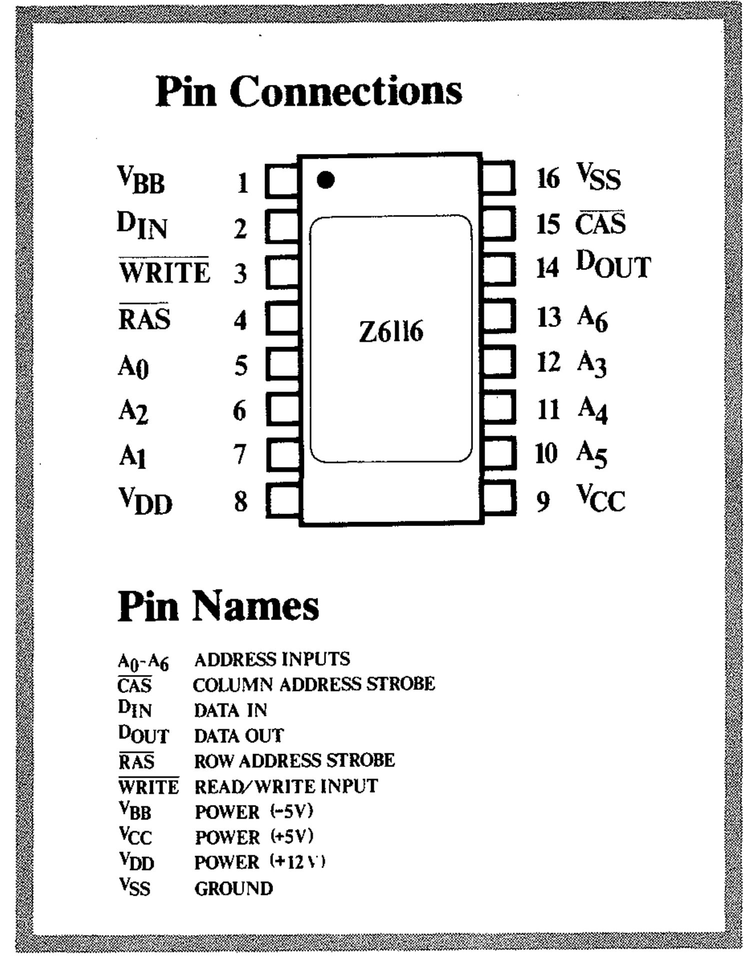


Fig 5.12 Dynamiska minnet 6116 (Zilog 6116, Texas 4116)

Refresh

RAM-minnet 4116 är dynamiskt och laddningarna på de 16536 minneskondensatorerna måste fyllas på (refresh) minst en gång för varje intervall på 2 ms. Refresh sker automatiskt vid radadressering ($\overline{RAS}$) och det gäller därför att läsa minnets samtliga 128 rader minst varannan millisekund.

Z80 har speciella kretsar för automatisk refresh. Denna refresh utförs vid varje fetchcykel. Som vi sett tidigare indikeras fetch av CPU-signalen $\overline{M1}$.

När instruktionen har hämtats in måste den avkodas och då är minnet "ledigt". Z80 passar då på att lägga ut en radadress på adressledningarna AØ-A7 och samtidigt en kontrollsignal $\overline{RFSH}$.

Interfacekretsen i fig 5.11 kan utformas så att den avkänner signalen $\overline{RFSH}$ (vilken inträffar i slutet på varje M1-cykel) och därvid automatiskt refreshar minnet. I Z80 finns ett speciellt register (R-registret i fig 3.24) som innehåller aktuell radadress för refresh och som automatiskt inkrementeras för varje M1-cykel. Fig 5.13 visar principen för refresh av minne med multiplexade adressingångar.

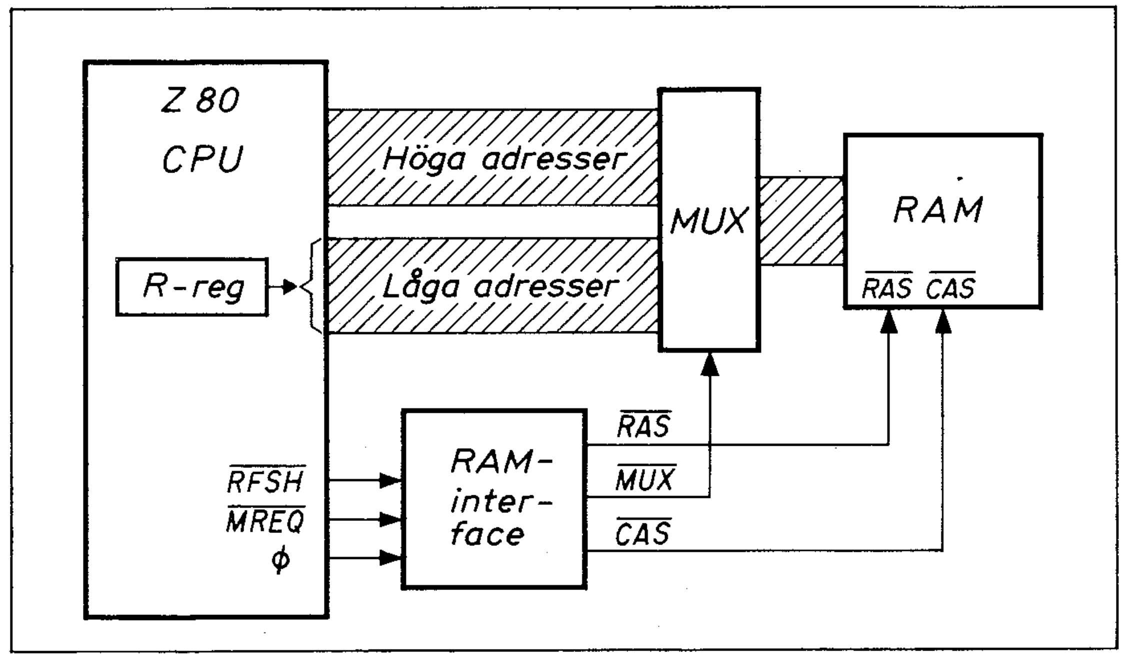


Fig 5.13 Automatisk refresh av dynamiskt minne med multiplexade adressingångar

4. Ljudgeneratorn

ABC80 innehåller en ljudgenerator. Den är uppbyggd i en mycket avancerad "teknologi" som kallas I^2L. Ordet teknologi syftar i detta sammanhang på tillverkningsteknik och kristallstruktur. Vi ska därför börja detta avsnitt med att omnämna några olika teknologier som finns representerade i kapslarna på ABC80:s kretskort.

Därefter ska vi visa blockschemat och användningen av ljudgeneratorn. Till skillnad från tidigare behandlade kretsar innehåller ljudgeneratorn både analoga och digitala funktionsblock.

4.1 Teknologier

I ABC80-systemet finns kretsar utförda i olika teknologier.

CPU:n och PIO:n är LSI-kretsar av MOS-typ. LSI betyder "large scale integration" och MOS "metal oxide semiconductor". MOS-kretsarna innehåller fälteffekttransistorer av N-kanaltyp (N-channel MOS transistor).

Minnena (ROM och RAM) är tillverkade NMOS-teknik med kiselstyre (N-channel Silicon Gate Technology).

Som avkodare utnyttjas PROM av "fusable link"-typ (typ 7611). De är i princip uppbyggda enligt fig 2.16 - 2.18 och innehåller vanliga bipolära transistorer (NPN-transistorer av "planar" typ).

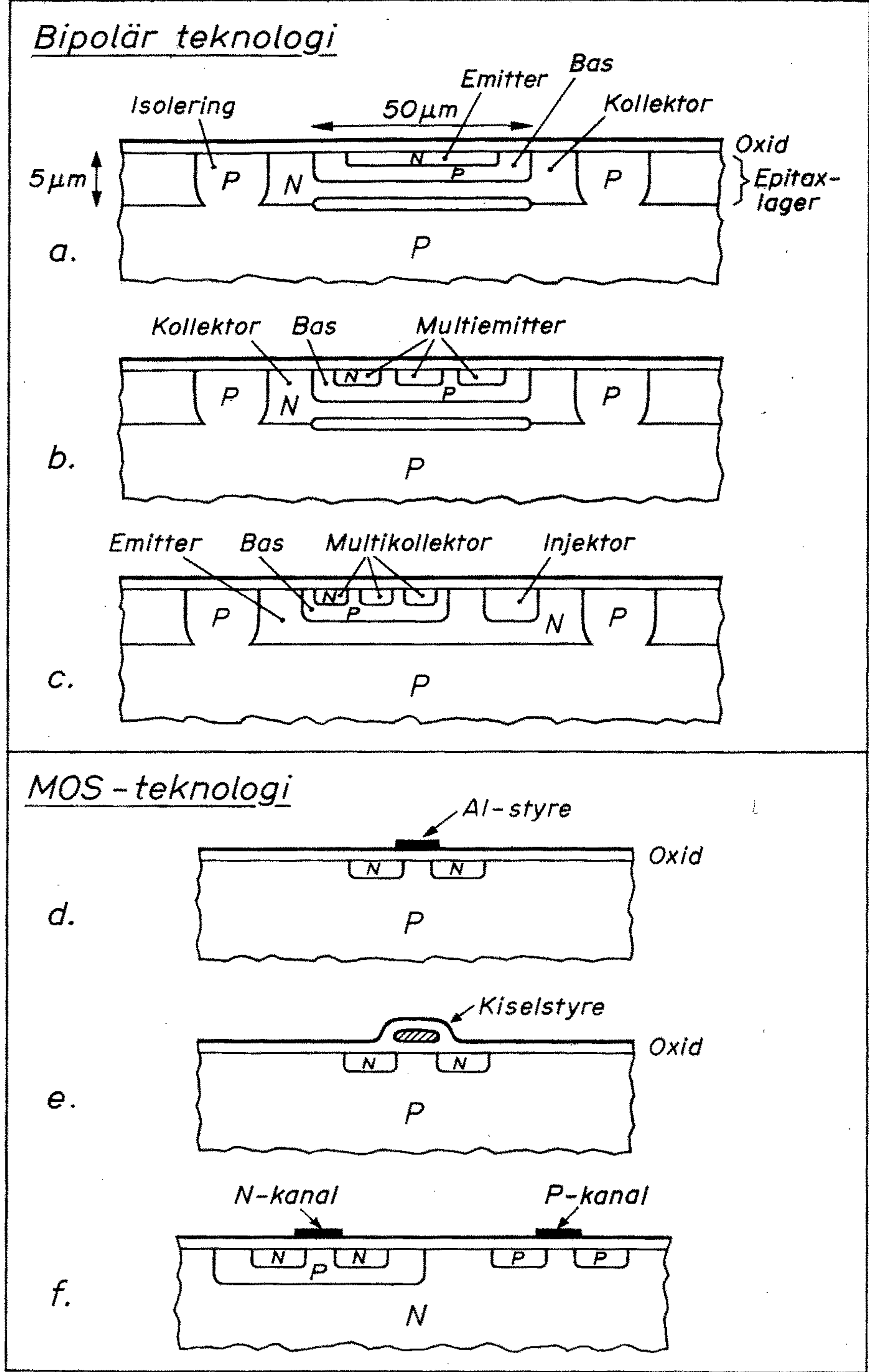


Fig 5.14 Kristallstruktur (i princip) för
a. Planartransistor
b. Multiemittertransistor (TTL)
c. Multikollektortransistor (I^2L) med separat injektor
d. MOS-transistor med metallstyre (metal gate)
e. MOS-transistor med kiselstyre (silicon gate)
f. CMOS-struktur

De flesta plastkapslar på kretskortet innehåller bipolära TTL-kretsar (Transistor-Transistor-Logic). Det finns olika varianter av TTL-kretsar. Standardkretsar betecknas med ett nummer som börjar med 74. De flesta av TTL-kretsarna i ABC80 är av typen "Low Power Schottky" och betecknas i fig 5.4 med LS (Low Power Schottky) och ett serienummer. LS-kretsarna är lika snabba som standard TTL men drar bara 20 % av standardkretsarnas effekt.

Ljudgeneratorn (SN 76477N) i ABC80 är utförd i en bipolär teknologi som kallas I^2L (Integrated Injection Logic). Det är en ny teknologi som man väntar sig mycket av i framtiden. I^2L möjliggör en blandning av analoga och digitala byggblock i samma kristallbricka (chip) och I^2L är dessutom mera komponenttät än andra jämförbara teknologier.

Slutligen finns CMOS-teknologin (CMOS = komplementär MOS) representerad i en MUX på tangentbordets kretskort. CMOS-kretsar är mycket högimpediva och effektsnåla.

Fig 5.14 visar några vanliga kristallstrukturer i princip. Bipolakretsar (fig 5.14a, b och c) kräver isolationslager mellan kretselementen. Bipolära kretselement tar därför större plats än MOS-teknologins kretselement.

I de flesta typer av halvledarkretsar är kretselementen så små att mellan 5000 och 50.000 element får rum i en kristall med ca 50 mm^2 area.

4.2 Ljudgeneratorns funktion

Principen för ljudgeneratorns funktion är enkel och framgår av fig 5.15. Till vänster i figuren ser vi tre generatorer.

- o VCO (voltage controlled oscillator) är en spänningsstyrd oscillator som genererar sinusvåg.
- o NOISE är en brusgenerator som generar vitt brus, dvs brus som innehåller alla hörbara frekvenser.
- o SLF (super low frequency) är en långsam fyrkantgenerator, typiskt 0,1 - 5 Hz.

Dessa tre signaler kan var och en för sig eller i olika kombinationer släppas fram genom MIXER (mixer = blandare) till ENVELOPE-blocket till höger. Detta block formar transienterna (in- och ursvängningsförloppen) så att utsignalen får önskad karaktär. SLF-generatorn kan dessutom styra VCO:n med triangelvåg.

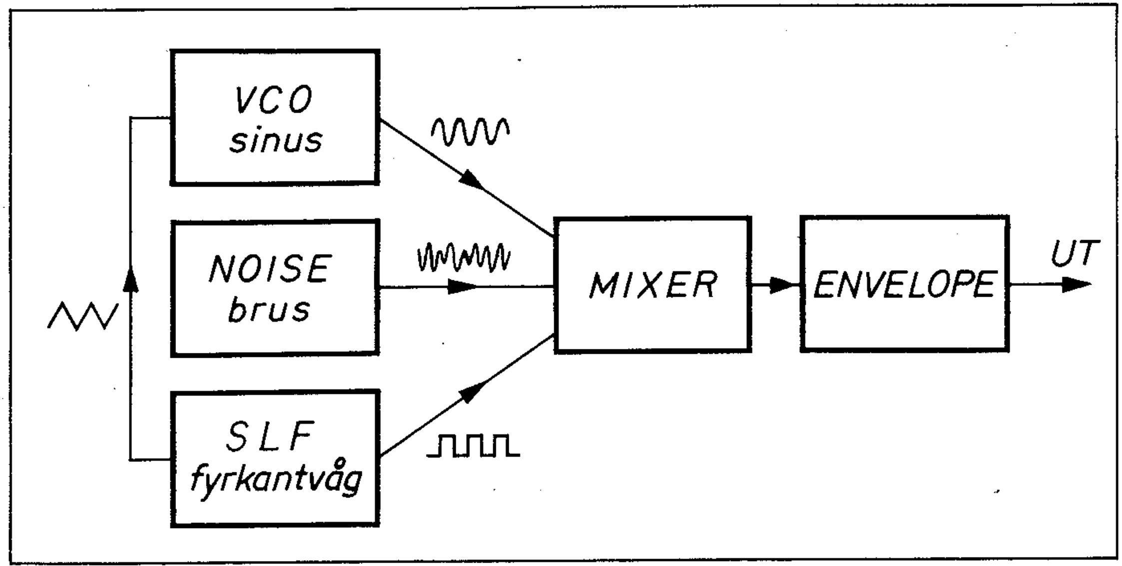


Fig 5.15 Principen för ljudgeneratorn

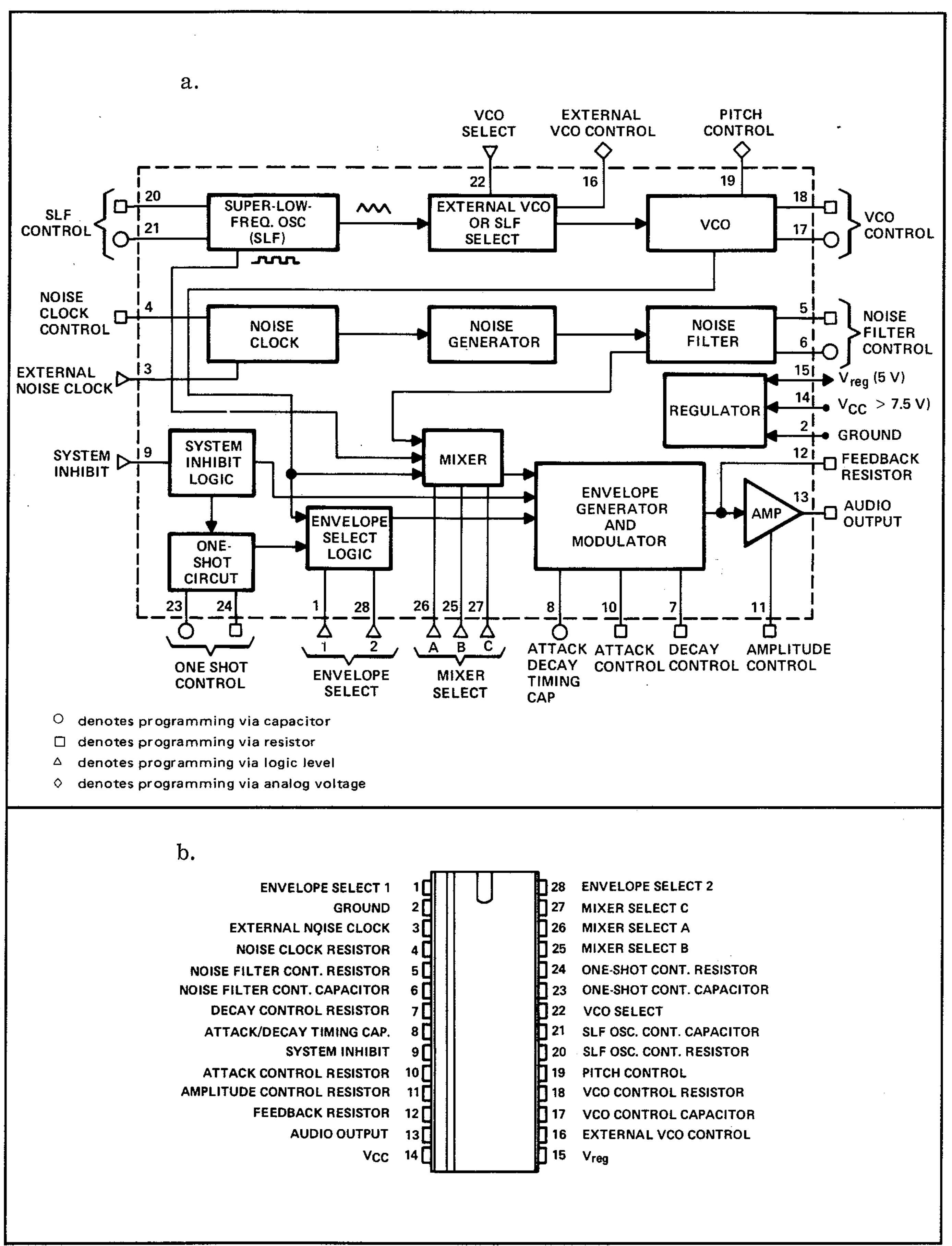
Fig 5.16a visar fabrikantens blockschema (Texas Instruments, SN 76477N, Complex Sound Generator). Som framgår av texten under figuren finns det fyra möjligheter att styra funktionerna i de olika blocken.

○ □ { anger analog styrning genom anslutning av yttre kondensator eller resistor.

◇ anger analog styrning med spänning.

△ { anger digital styrning (med binära ingångar, dvs två logiska nivåer per ingång).

Alla dessa styrsignaler medför att kapseln behöver ha många ben (totalt 28 ben). Uttagsplaceringen framgår av fig 5.16b.



<u>Fig 5.16</u> Ljudgeneratorn 76477N
a. Blockschema
b. Kapsel

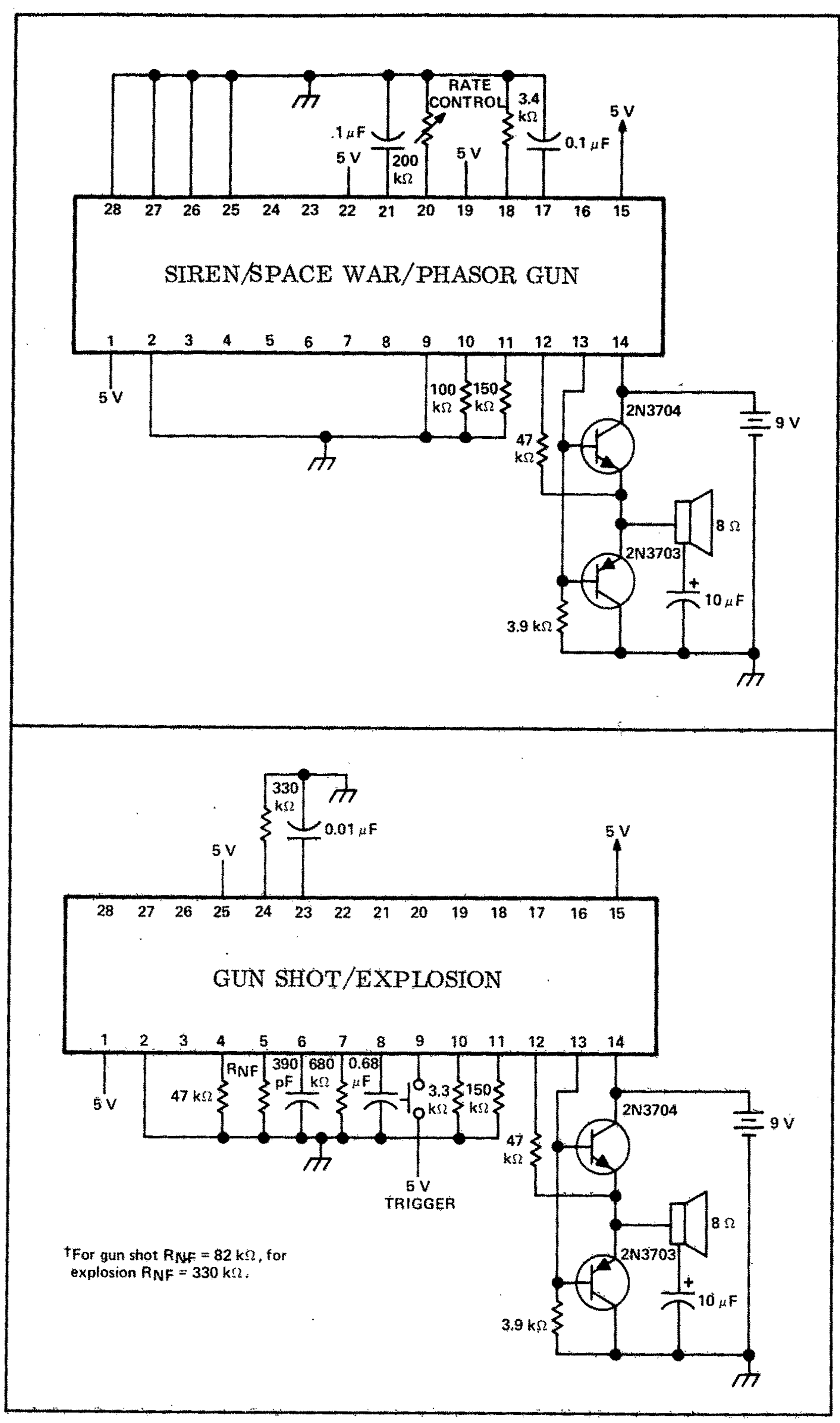


Fig 5.17 Exempel på ljudgenerering med kretsen 76477

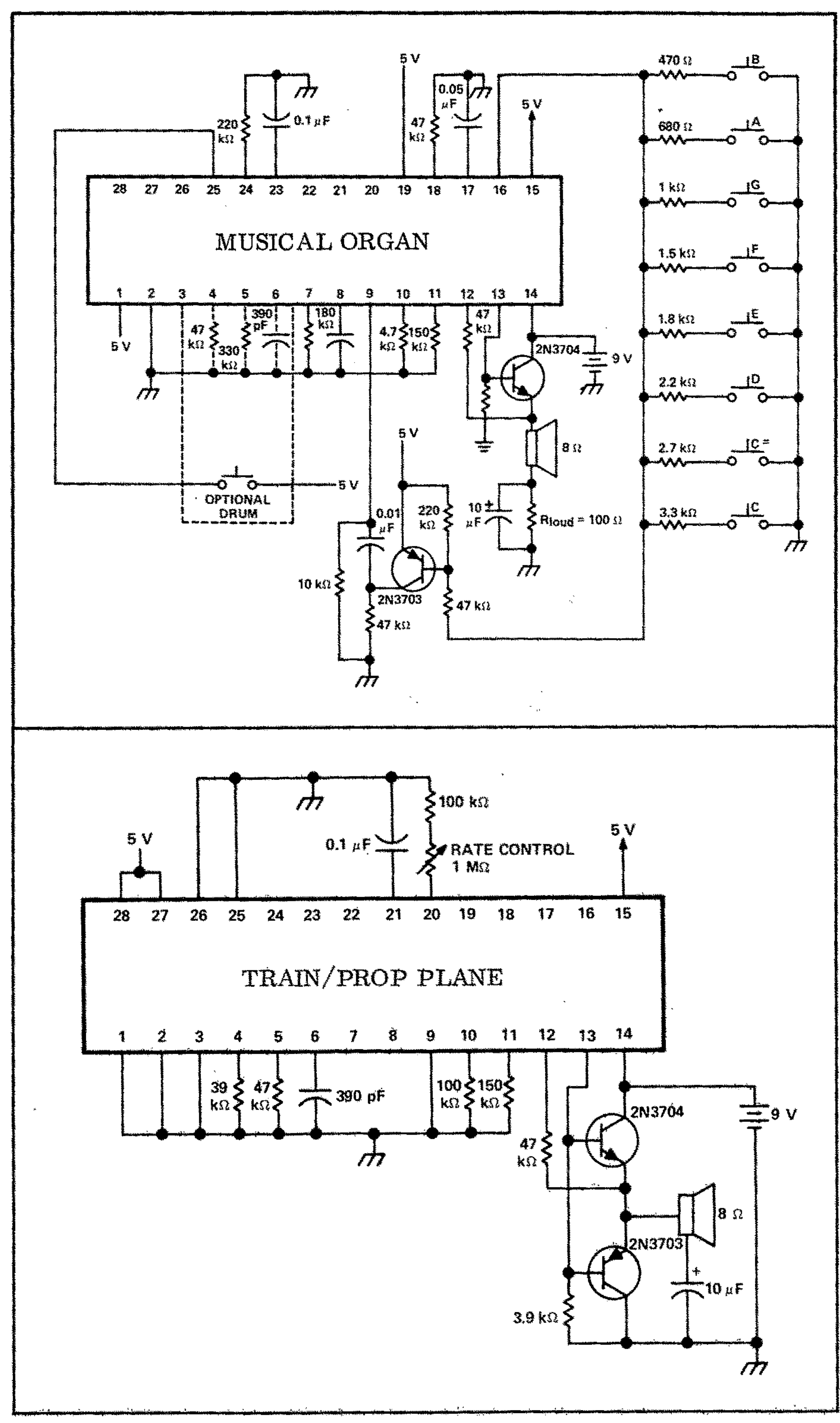


Fig 5.17 forts.

Många ljud kan genereras med kretsen SN 76477N. Fig 5.17 visar fyra exempel från databladet ("typical applications").

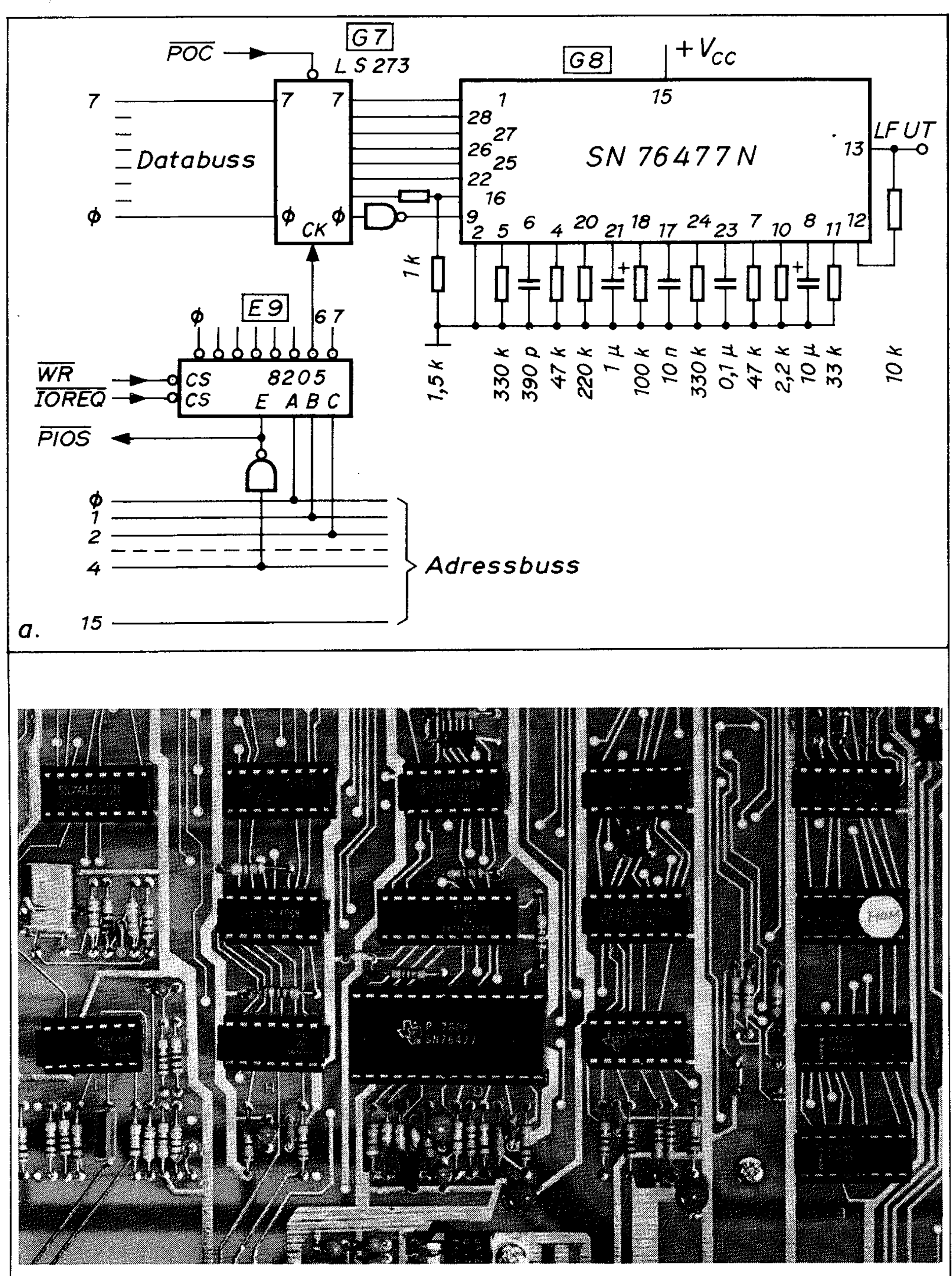
För att tillfullo kunna utnyttja ljudgeneratorn i ABC80 skulle man behöva koppla in en hel rad interface-kretsar, bl a DA-omvandlare (DA = digital-analog) för de analoga styrfunktionerna. För att hålla kostnaderna nere har man i ABC80 gjort en enklare inkoppling som visas i fig 5.18.

De flesta styringångar har här fixa värden på C och R via anslutna kondensatorer och resistorer. Inspänningar tas från spänningsdelare.

Ljudgeneratorn är via latchen 74LS273 ansluten till databussen på följande sätt:

Databuss	Ljudgenerator
bit ∅	ben 9 (via inverterare
1	16 (via spänningsdelare)
2	22
3	25
4	26
5	27
6	28
7	1

Latchens nollställningsingång är kopplad till $\overline{\text{POC}}$ (processor clear) som ger nollställning vid systemreset. Latchens klockingång får signal från utgång nr 6 på adressavkodaren 8205 (position E9) som är en av de två "en av åtta"-avkodare (E8 och E9) som ger kontrollsignaler på busskontakten. Avkodaren i position E9 selekteras av $\overline{\text{WR}}$ och $\overline{\text{IORQ}}$ låga samt bit 4 på adressbussen låg. "En av åtta"-valet sker därefter med bitarna ∅, 1 och 2 på adressbussen.



b.

Fig 5.18 Ljudgeneratorns inkoppling i ABC80
a. Schema
b. Placering på kretskortet

Med denna inkoppling blir ljudgeneratorn en utport med IO-adressen 6. Valet av funktion sker med ett styrord (liknande våra tidigare ljussignaler i trafikljussystemet) som placeras på databussen. Funktionerna framgår av fig 5.19. Med styrordets bit ∅ kan vi stänga av ljudgeneratorn, med bit 1 och 2 kan vi variera VCO (hög frekvens, låg frekvens eller triangelstyrd "polisbilssiren"). Med bitarna 3, 4 och 5 kan vi göra olika kombinationer av VCO, NOISE och SLF. Med bitarna 6 och 7 kan vi slutligen välja en av fyra enveloper.

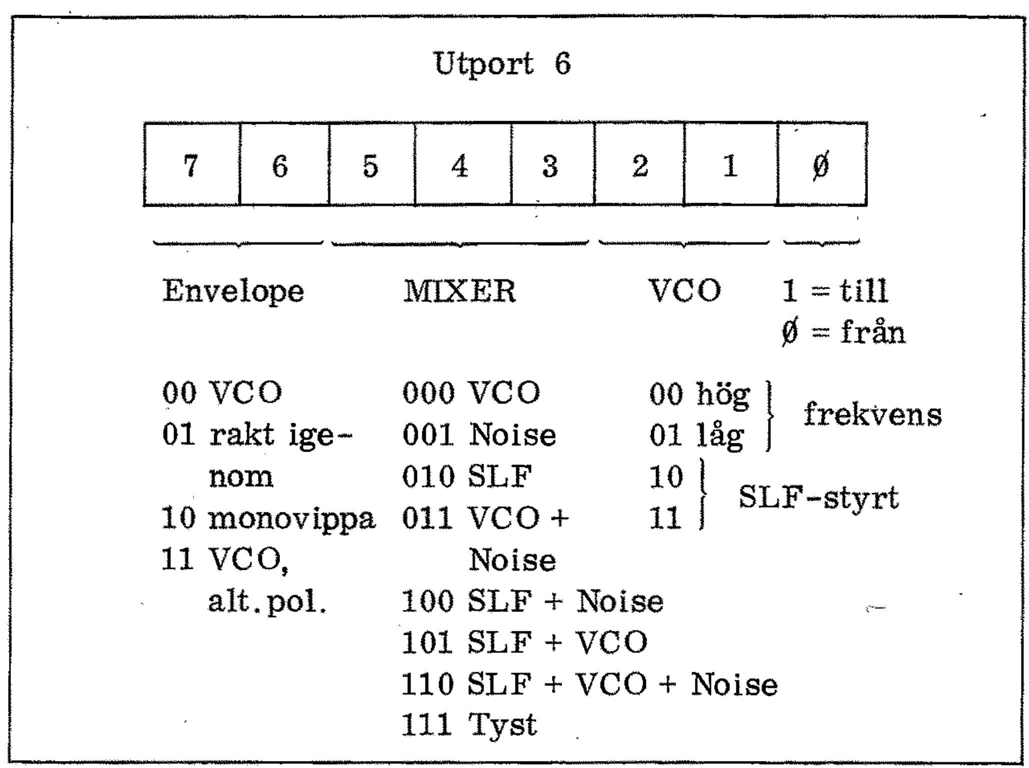


Fig 5.19 Programmerbara funktioner hos ljudgeneratorn i ABC80

Ljud ska höras och kan inte tillfullo avnjutas i text. Vi ska senare göra ett program som demonstrerar ljudgeneratorns möjligheter.

5. PIO-kretsen

Till höger i blockschemat i fig 5.3 ser vi PIO-kretsen. Port A används för att ta emot signaler från tangentbordet. Port B används för två olika uppgifter, dels för kassettinterface och dels för ett allmänt interface som kan programmeras bl a för V24-snitt.

Vi har redan tidigare utnyttjat en PIO för trafikljusen i avsnitt 2.4 i kap 4. Fördelen med en PIO är bl a att man kan programmera den att utföra olika uppgifter utan att man behöver använda lödkolven när man ska ändra arbetsuppgift.

I ABC80 används en metod för inkoppling av IO-enheter som kallas "interrupt" eller på svenska "avbrott". Vi ska börja detta avsnitt med att beskriva vad begreppet interrupt innebär.

Vi ska därefter se hur PIO:ns blockschema ser ut och hur de olika kretsarna inkopplats till PIO:n i ABC80. Programmeringen ska vi återkomma till i ett senare kapitel.

5.1 Avbrott

I vårt tidigare trafikljusprogram låg datorn i en programslinga (loop) och väntade på signal från trafik på bivägen. Det är ju slöseri med CPU:ns kapacitet. CPU:n hade kunnat utföra mängder med beräkningar eller annat nyttigt arbete under väntetiden. Istället för att ligga i vänteslinga hade CPU:n exempelvis kunnat programmeras att göra statistik över trafiken.

Om man vill utnyttja en CPU för visst arbete (bakgrundsprogram) men samtidigt vara beredd att när som helst ta emot en signal utifrån kan man använda interrupt eller avbrott. Det tillgår i stort sett på följande sätt.

- o Signalen från en IO-enhet (exempelvis från en sensor i bivägen eller från ett tangentbord) tillförs CPU:ns interruptingång.
- o CPU:n avslutar då den pågående instruktionen i bakgrundsprogrammet och lägger programräknarens innehåll på stacken.
- o CPU:n svarar nu med signalen "interrupt acknowledge".
- o Programräknaren kan nu laddas med en hoppadress som i detta sammanhang brukar kallas vektor.
- o Hoppadressen (eller vektorn) pekar ut en plats i minnet där en rutin för avbrottshanteringen lagrats.
- o Avbrottshanteringen kan exempelvis bestå i att utföra ljusväxling eller läsa in ett värde (dvs koden för nedtryckt tangent) till viss plats (en "mjukvaru-buffert") i minnet.
- o När avbrottsrutinen är slutförd återgår CPU:n till sitt bakgrundsprogram genom att först hämta tillbaka programräknaren från stacken.
- o Om vissa register används av avbrottsrutinen måste även deras innehåll lagras på stacken vid avbrott och hämtas från stacken när bakgrundsprogrammet återupptas (Z80 har dubbla uppsättningar av register. Man kan därmed hoppa över från den ena registerbanken till den andra vid avbrott).

5.1.1 6800systemets avbrottsrutiner

Olika mikroprocessorer har olika metoder för avbrottshantering. 6800 har exempelvis fyra fasta avbrottsadresser (vektorer) högst upp i minnet. Fig 5.20.

Address	
FFFF	Reset Vector (low order address)
FFFE	Reset Vector (high order address)
FFFD	Non-Maskable-Interrupt Vector (low order address)
FFFC	Non-Maskable-Interrupt Vector (high order address)
FFFB	Software Interrupt Vector (low order address)
FFFA	Software Interrupt Vector (high order address)
FFF9	Interrupt Request Vector (low order address)
FFF8	Interrupt Request Vector (high order address)

Fig 5.20 Adresser till de fyra avbrottsrutinerna i 6800-systemet

I 6800-systemet har man fyra typer av avbrott (RESET kan räknas som en typ av avbrott).

- o Signal på $\overline{\text{INT}}$-ingången
- o Programstyrt avbrott (instruktionen SWI = software interrupt).
- o Signal på $\overline{\text{NMI}}$-ingången
- o $\overline{\text{RESET}}$

$\overline{\text{INT}}$ ger vanligt avbrott som kan maskeras (dvs inhiberas, disablas eller bortkopplas) med en instruktion i programmet.

$\overline{\text{NMI}}$ (non maskable interrupt) är ett avbrott som ej kan maskeras. Det används exempelvis vid spänningsavbrott (för lagring av data innan spänningen helt försvunnit) och får därför ej hindras genom maskering.

SWI är en instruktion som ger avbrott (och kan användas som HALT-instruktion).

$\overline{\text{RESET}}$ är en nollställning av systemet och ger startadress för programexekvering.

Ett avbrott i 6800-systemet leder alltså till ett hopp till den adress som utpekas av innehållet i en av de i fig 5.20 angivna minnespositionerna (dvs till den adress som utpekas av en av de fyra lagrade vektorerna). Initialiseringen av 6800 måste alltså alltid innehålla laddning av hoppadresser åtminstone i en av dessa positioner.

5.1.2 Z80-systemets avbrottsrutiner

Z80-systemets avbrottsrutiner är avsevärt mer generella är 6800-systemets. Vi kan därför inte här gå in på alla olika möjligheter som står till buds. Bland de olika rutinerna finns givetvis även maskerbart avbrott ($\overline{\text{INT}}$) och icke maskerbart avbrott ($\overline{\text{NMI}}$). I Z80-systemet kan emellertid IO-enheterna själva lägga ut vektorerna till önskade avbrottsrutiner (vector interrupt) och därmed har man obegränsat antal avbrottsvektorer.

PIO:n i Z80-systemet kan exempelvis lagra en avbrottsvektor för vardera av de två portarna. Avbrottssignalen kan genereras inom PIO:n genom en programmerbar avbrottsmask (interrupt mask). Man kan därvid logiskt jämföra inportens signaler med avbrottsmasken och därmed generera avbrott för en enda kombination (AND) eller ett givet antal kombinationer (OR) mellan insignal och avbrottsmask.

Avbrottssignalerna från flera PIO-kretsar kan seriekopplas ("Daisy chain") så att den krets som ligger närmast CPU:n får högsta prioritet dvs behandlas först vid samtidig avbrottsbegäran på flera PIO-kretsar.

Avbrottssignalerna från port A och port B i PIO:n sänds via inbyggda programstyrda grindar till PIO:ns $\overline{\text{INT}}$-utgång varifrån $\overline{\text{INT}}$-signalen kan kopplas vidare till CPU:n.

När CPU:n fått $\overline{\text{INT}}$-begäran svarar den med signalen "interrupt acknowledge". Eftersom antalet pinnar är begränsade till 40 får man i Z80 inte plats med ett speciellt uttag för "interrupt acknowledge". I Z80 ges därför denna signal med en $\overline{\text{IORQ}}$-signal under $\overline{\text{MI}}$-pulsen (där $\overline{\text{IORQ}}$ normalt inte ska förekomma). Kombinationen av $\overline{\text{MI}}$ och $\overline{\text{IORQ}}$ avkodas internt i PIO:n som därefter lägger ut en vektor på databussen. Z80 kombinerar denna vektor (8 bitar) med innehållet i I-register till en 16 bitars adress som indirekt pekar ut avbrottsrutinen på det sätt som visas i fig 5.21.

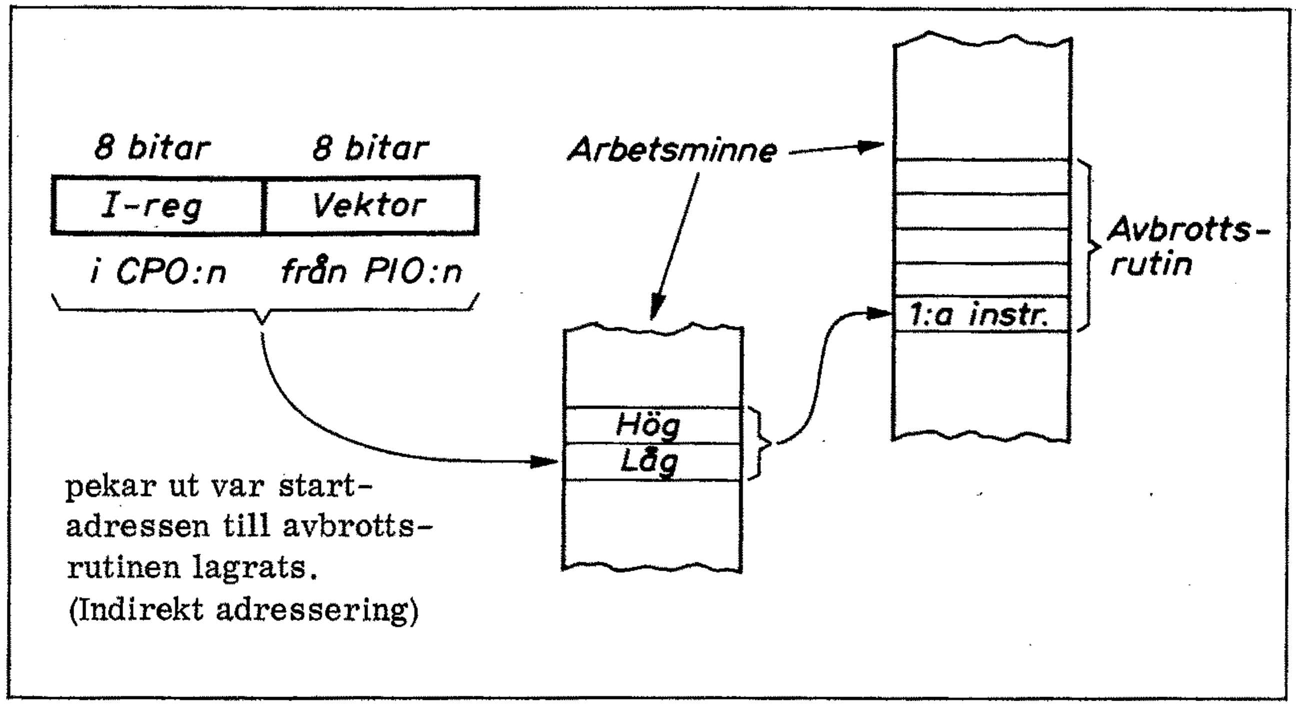


Fig 5.21 Vektor-avbrott i Z80

5.2 PIO:ns blockschema

Fig 5.22a visar hur fabrikanten presenterar PIO:ns blockschema. Till vänster ser vi databuss och kontrollbuss (CPU interface).

Till höger ligger de två portarna med sina åtta dataledningar och två ledningar för "handskakning". Av dessa är den ingående ledningen en "strob" med vars hjälp man exempelvis kan klocka inläsningen till porten. Den utgående ledningen ger "färdig"-signal som talar om att porten är beredd att ta emot ny information.

Nederst i fig 5.22a ser vi avbrottskontrollen. Den består av tre ledningar, en utgående $\overline{\text{INT}}$-signal samt in- och utgång för "Daisy-chain".

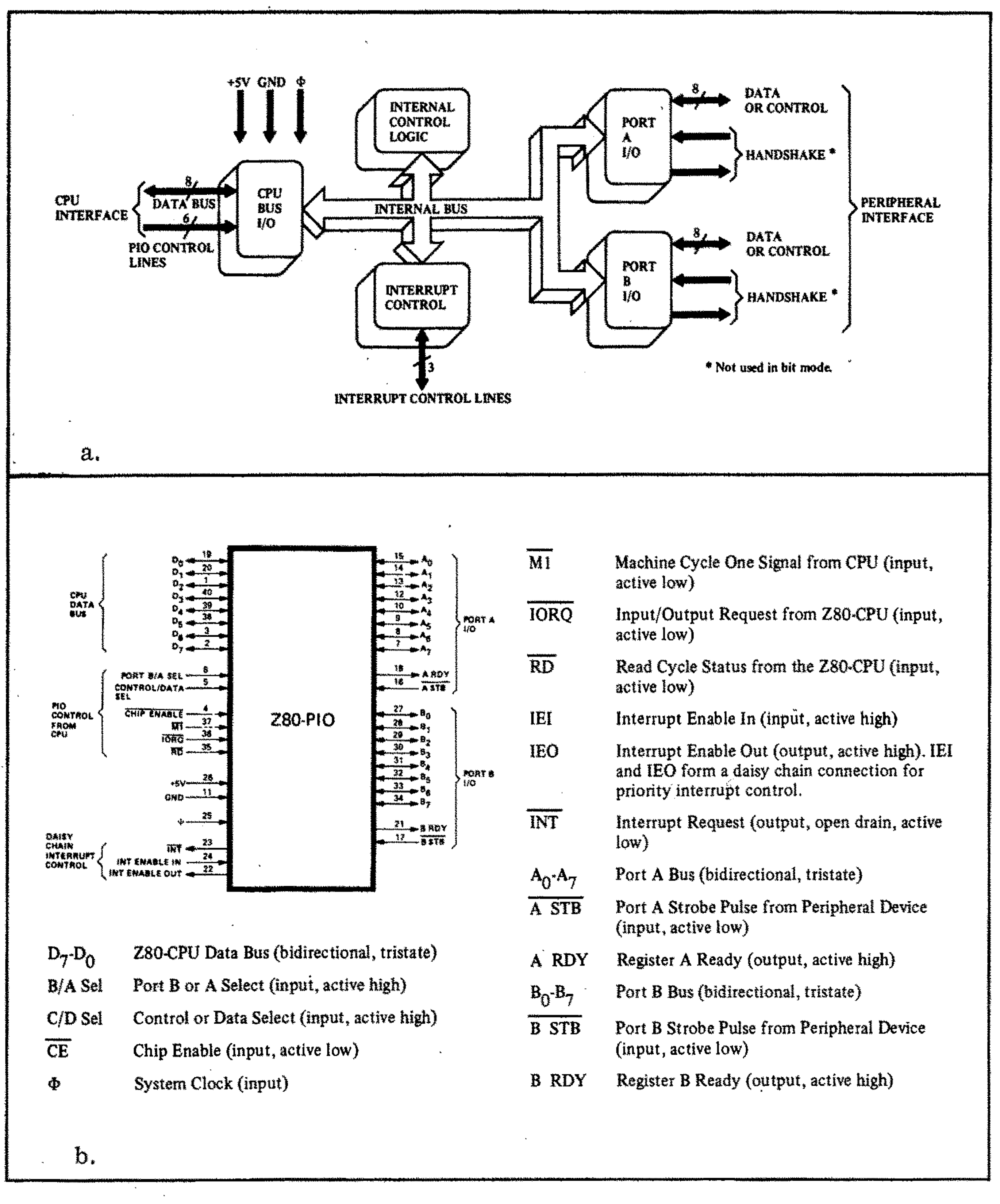


Fig 5.22 Z80-PIO
a. Blockschema
b. Uttagsplacering och funktionsbeteckning

Fig 5.22b visar uttagsplacering och beskrivning av de olika benens funktion. Man måste givetvis noga studera fabrikantens datablad när man ska använda PIO-kretsen, här diskuterar vi ju enbart principer.

5.3 PIO:ns inkoppling i ABC80

Fig 5.23 visar hur PIO:n kopplats in i ABC80. Adresseringen sker som tidigare nämnts med bit 4 hög ($\overline{CE}$ = chip enable). Val av port och val av kontroll- eller dataregister sker med bit Ø respektive 1.

Databussens samtliga ledningar har "pull up"-motstånd på 10 kΩ inkopplade till matningsspänningen V_{CC} (+5 V). Om det inte kommer några signaler (exempelvis om man adresserar en minnesposition där det inte finns någon minneskrets inkopplad) ger dessa motstånd svaret FFH (= samtliga ledningar höga). Detta utnyttjas som en avbrottsvektor (FFH) vid avbrottsbegäran från ABC-bussen.

PIO:n behöver flera styrsignaler. $\overline{RD}$ behövs för valet mellan läsning och skrivning. $\overline{IORQ}$ behövs både för IO-adressering och "interrupt acknowledge".

PIO:ns $\overline{INT}$-utgång går till Z80:s $\overline{INT}$-ingång. Även här används "pull up"-motstånd och därigenom kan flera ingångar inkopplas enligt principen "wired OR". Busskontakten har exempelvis en $\overline{INTØ}$-ingång som kan sänka spänningen på CPU:ns $\overline{INT}$-ingång. I fig 5.23 har principen för $\overline{INTØ}$-ingången markerats med en seriediod. I ABC80 används en mer komplex krets (LM339) som ger inställbar nivå och bättre isolation mellan busskontakt och CPU.

5.3.1 RESET

I ABC80 finns en rad kretsar inkopplade för att ge säker $\overline{RESET}$ i olika lägen. I fig 5.23 visar vi principen för några av dessa kretsar. I busskontakten finns en ledning för generell nollställning (exempelvis när man startar systemet). Den kallas $\overline{RESIN}$ och den påverkar CPU:n ($\overline{RES}$), PIO:n (via $\overline{M1}$) samt övriga kretsar via signalen $\overline{POC}$ (processor clear). $\overline{RESIN}$ ger även en $\overline{RST}$-signal på busskontakten för nollställning av alla yttre IO-kretsar.

Utöver ovanstående RESET-kretsar finns en automatisk systemreset vid spänningstillslag. Denna har utelämnats i fig 5.23.

Antalet ben på PIO:n räcker inte till för en separat $\overline{RESET}$-ingång. M1-signal ska normalt vara kombinerad med $\overline{RD}$ eller $\overline{IORQ}$. Om $\overline{M1}$ ges utan samtidig $\overline{RD}$- eller $\overline{IORQ}$-signal tolkar PIO:n den ensamma $\overline{M1}$-signalen som $\overline{RESET}$. Detta är anledningen till att $\overline{M1}$-ingången på PIO:n via en OR-grind inkopplats till både CPU:ns $\overline{M1}$-ingång (normal inkoppling) och $\overline{POC}$ (för $\overline{RESET}$).

Till ingången $\overline{RESIN}$ i busskontakten är även en manuell nollställningsknapp inkopplad. Om ABC80 av någon anledning "går ur kontroll" kan man ge en system-reset genom att trycka på denna knapp som återfinns till vänster baktill (dvs position A11 på kretskortet).

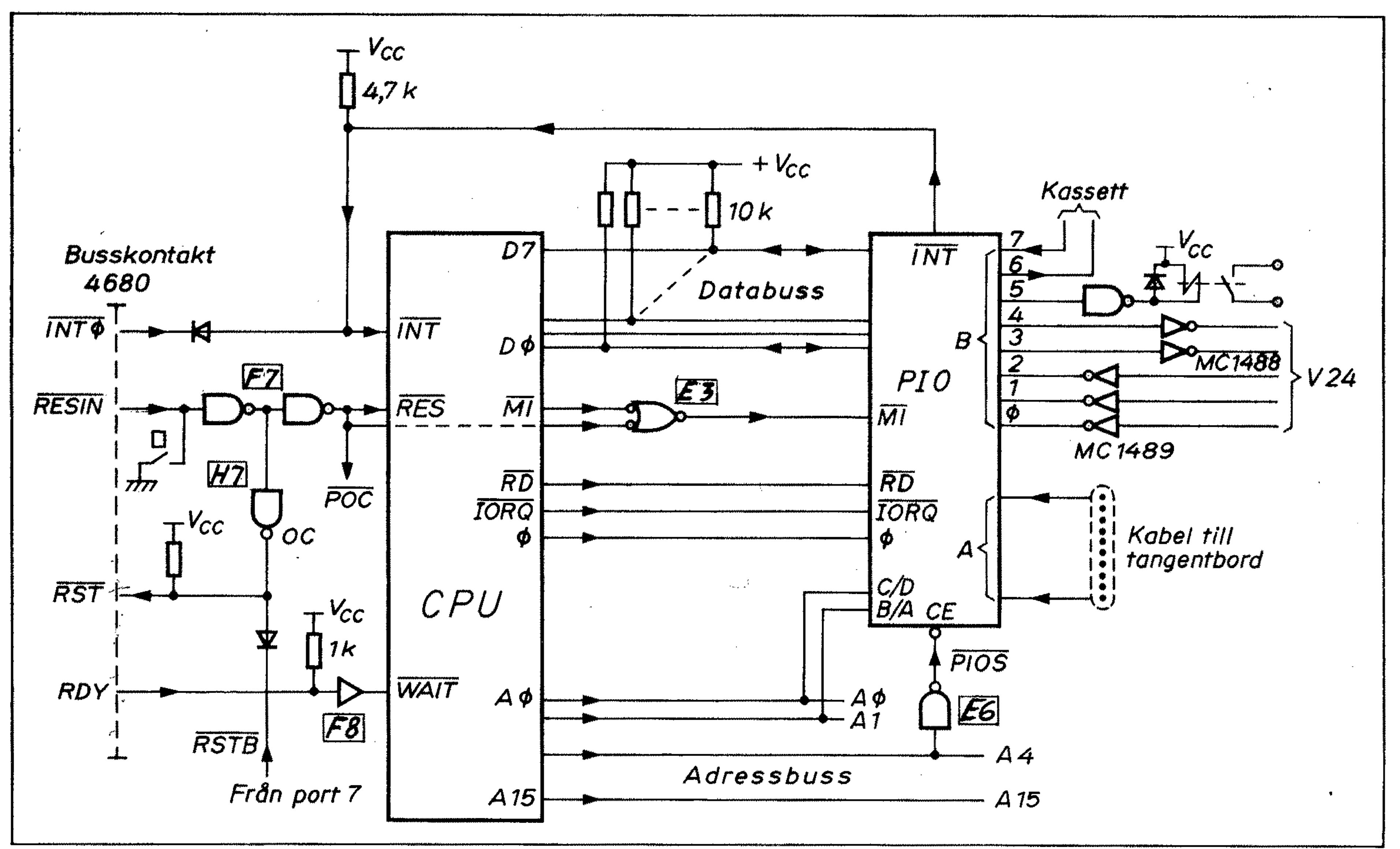


Fig 5.23 Principen för PIO:ns inkoppling i ABC80

IO-kretsarna kan nollställas under programkontroll genom adressering av inport 7 vilket ger $\overline{RSTB}$-signal. Som framgår av fig 5.23 (till vänster) kommer därvid enbart IO-enheter att nollställas via den utgående $\overline{RST}$-ledningen i busskontakten.

5.3.2 Port A

Av PIO:ns två portar, port A och port B, används port A till tangentbordet. Det sker via en kabelkontakt i kretskortet (position A8) och en flatkabel.

5.3.3 Port B

Port B används både för ett V24-snitt och kassettbandspelaren. Bitarna Ø-4 i port B är via speciella kommunikationskretsar anslutna till ett 9-poligt uttag i ABC80. Bitarna 5-7 används för kassettbandspelaren.

V24-snittet

Bitarna Ø, 1 och 2 i port B är ingångar och här sitter linjemottagare (line receivers) av typ MC 1489 (position A9). Dessa mottagare har "hysteres-ingång" som gör att de blir mindre känsliga för störningar. Inimpedansen till mottagarkretsarna är ca 5 kΩ. Normalt ska bit Ø vara dataingång som arbetar med hög hastighet medan bit 1 och bit 2 är kontrollingångar.

MC 1488 motsvarar Texas 75188 och MC 1489 Texas 75189.

Mottagarkretsarna har en speciell ingång (response control) för störfiltrering. Här är kondensatorer inkopplade (470 pF till dataingången, bit Ø och 10 nF till kontrollingångarna, bit 1 och 2).

Bit 3 och 4 är linjeutgångar. PIO:n förmår inte själv driva en ledning och den tål inte heller de störningar som en direkt anslutning till en ledning skulle medföra. Därför är linjedrivkretsar (line drivers) inkopplade mellan PIO:n och V24-kontakten (position B9). Drivkretsarna heter MC1488 och matas med ± 9 V. De ger en utspänning av ca ± 7 V och har strömbegränsning till ca 10 mA. Utimpedansen är 300 Ω. Fig 5.24a visar linjekretsarnas uttagsplacering enligt databladet.

MC1488

QUAD MDTL LINE DRIVER
RS-232C
SILICON MONOLITHIC
INTEGRATED CIRCUIT

L Suffix
CERAMIC PACKAGE
CASE 632
TO-116

PIN CONNECTIONS

V_{EE} 1, Input A 2, Output A 3, Input B1 4, Input B2 5, Output B' 6, Gnd 7, Output C 8, Input C2 9, Input C1 10, Output D 11, Input D2 12, Input D1 13, V_{CC} 14

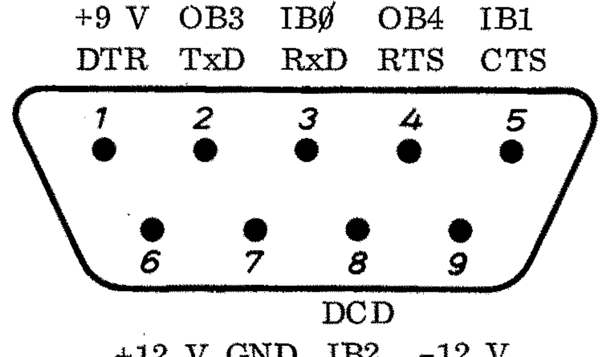
MC1489L
MC1489AL

QUAD MDTL
LINE RECEIVERS
RS-232C

SILICON MONOLITHIC
INTEGRATED CIRCUIT

Input A 1, Response Control A 2, Output A 3, Input B 4, Response Control B 5, Output B 6, Ground 7, Output C 8, Response Control C 9, Input C 10, Output D 11, Response Control D 12, Input D 13, V_{CC} 14

L SUFFIX
CERAMIC PACKAGE
CASE 632
TO-116



Beteckningarna närmast uttagen avser modemkontrollsignaler

OB3: Out Bit 3
IBØ: In Bit Ø
GND: Ground (jord)

Fig 5.24 a. Linjekretsarna till V24-snittet
b. V24-kontakten

Kassett-interfacet

Bit 5 i port B används som utgång och styr ett relä (Position K10). För att få tillräcklig drivström till relät används en inverterare (position H7) som buffert. Över relät sitter en diod för att skydda bufferten från den spänningstransient som i annat fall skulle uppträda vid frånslag i relät.

Fig 5.25 visar de olika uttagen på ABC:s kretskort.

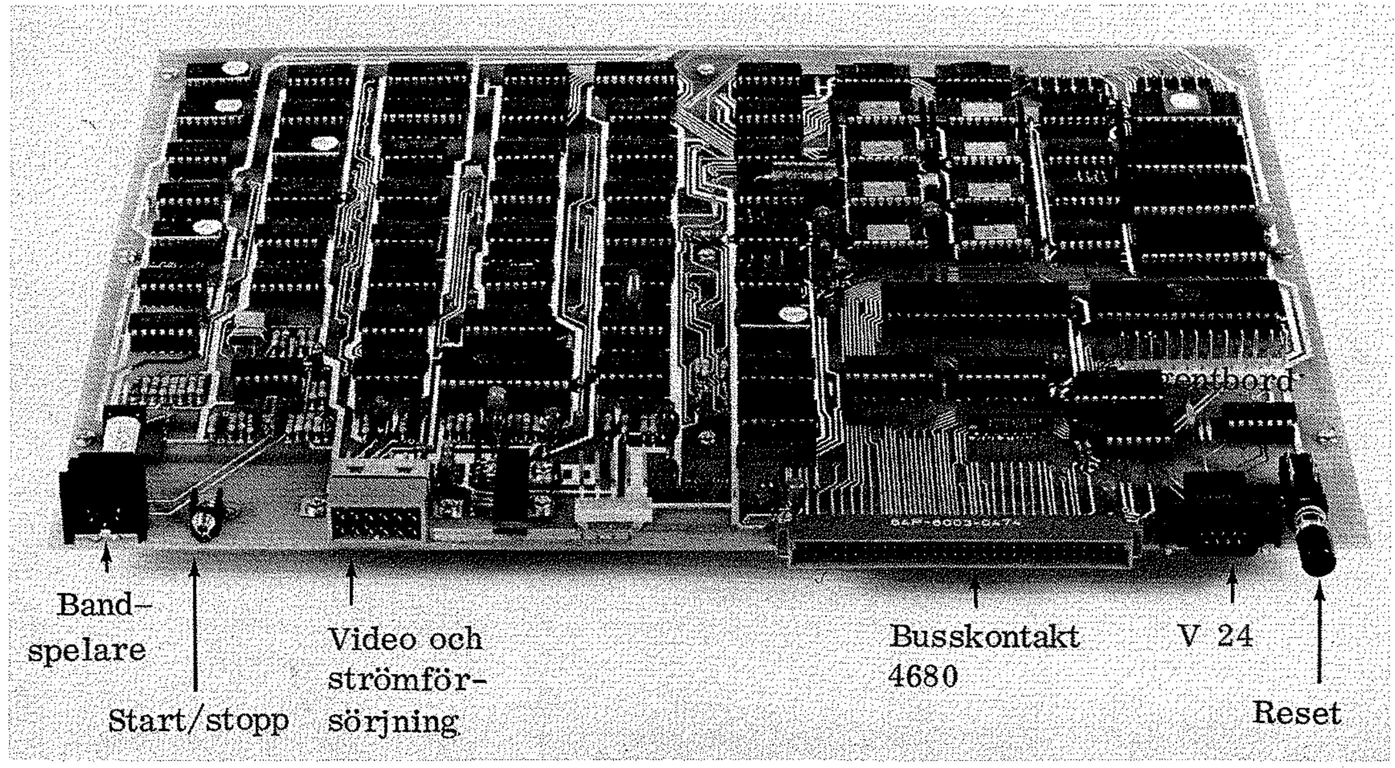


Fig 5.25 Uttagen på ABC80:s kretskort

6. Busskontakten

Vi har vid flera tillfällen tidigare berört busskontakten i ABC80. Den kan användas för anslutning till 4680-bussen och vi har i fig 5.25 sett uttaget baktill på ABC:s kretskort. Det är detta uttag som gör det möjligt att bygga ut ABC80 till ett generellt datorsystem.

Busskontakten har 64 pinnar (DIN 41612) och fig 5.26 ger en sammanställning av hur dessa pinnar används.

Med busskontakten i ABC80 kommer vi tydligen åt adressbuss, databuss och en rad olika kontrollsignaler. Det står givetvis var och en fritt att allt efter egna önskemål använda dessa ledningar.

Anledningen till busskontakten i ABC80 är framför allt att man enkelt ska kunna bygga ut systemet. Till ABC80 hör bl a en printer och en floppy disk som dessutom har plats för extra minne. Dessa enheter kan direkt pluggas in i busskontakten. Fig 5.27.

För den som själv önskar bygga ut ABC80 kan det vara av intresse att lära känna kretsarna som styr signalerna på busskontakten.

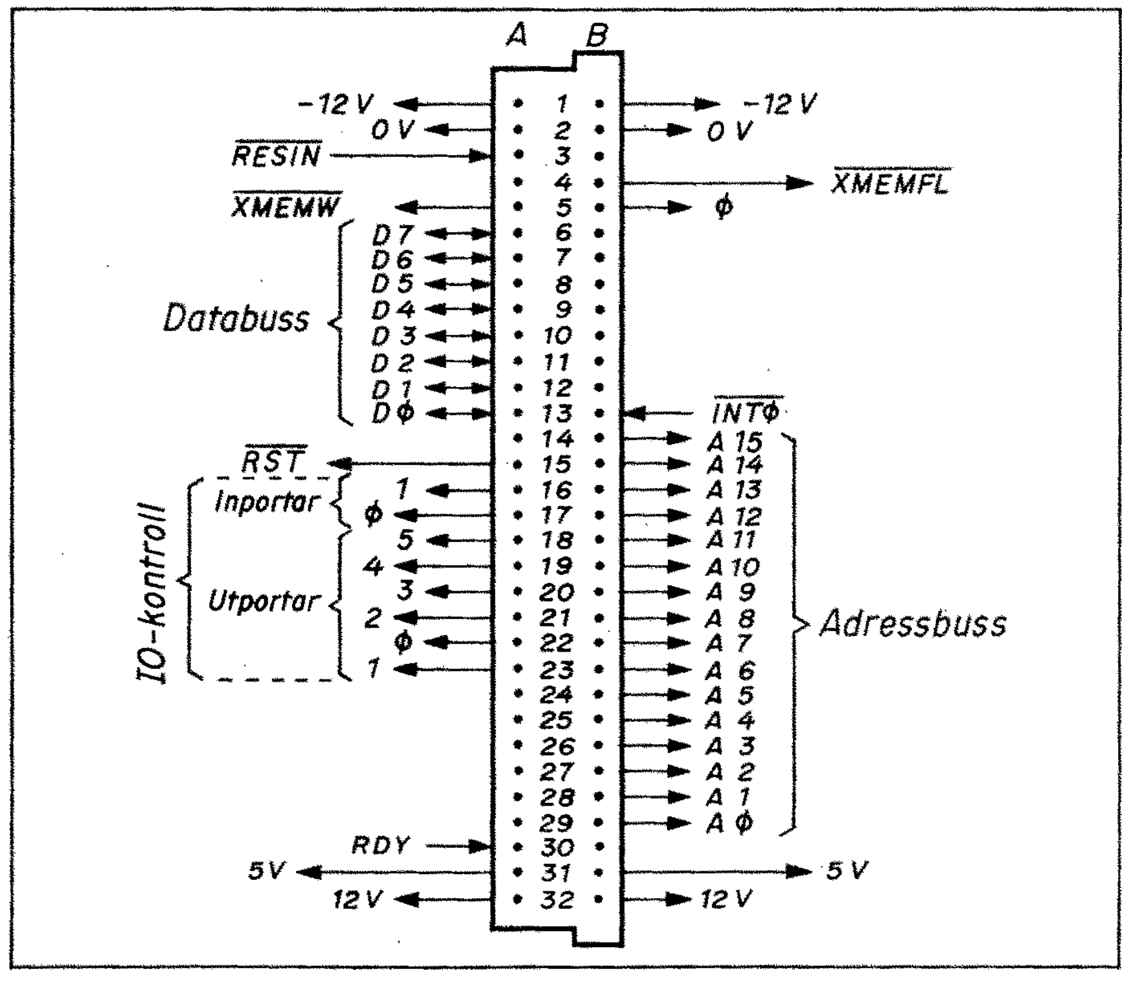


Fig 5.26 4680-kontakten i ABC80 och dess pinfunktioner

Fig 5.27 ABC80 med printer och floppy disk, båda inkopplade via 4680-kontakten

6.1 Anslutning av yttre minne

Via busskontakten får vi kontakt med både adressbussen och databussen i ABC80. Adressbussen är ständigt riktad utåt (från CPU:n) och för att inte ABC80:s adressbuss ska bli för hårt belastad ligger buffertar (typ LS241, position D8 och C8) mellan inre adressbussen och busskontakten i ABC80.

Även databussen är "buffrad" men här fordras en buffert som kan "vändas", dvs vara riktad utåt (från CPU:n) vid skrivning och inåt vid läsning. Här krävs alltså kretsar för styrning av buffertens riktning. Databussens buffert återfinns i position B8 på kretskortet.

Som framgår av minnesplanen i fig 5.6 är det vissa minnesareor som är avsedda för yttre (externa) minnen. För att undvika sammanblandning av yttre och inre minnen har dessa areor avkodats. De kontrollsignaler i busskontakten som avser det yttre minnet läggs enbart ut när externa minnesareor adresseras. Vi ska nu studera hur ABC80 genererar dessa kontrollsignaler.

6.1.1 Busskontaktens kontrollsignaler för minnesaccess

För att inom ABC80 skilja mellan minnesarea och IO-area används signalerna $\overline{\text{MREQ}}$ och $\overline{\text{IORQ}}$. Vi har tidigare sett hur man i ABC80 kombinerar $\overline{\text{MREQ}}$ och $\overline{\text{RD}}$ till signalen $\overline{\text{MRD}}$ (läs i minnet).

ABC80 kan som vi sett använda både internt och externt minne. I ABC80 genereras två speciella kontrollsignaler som motsvarar de interna signalerna $\overline{\text{MRD}}$ och $\overline{\text{WR}}$ men som enbart är avsedda för det externa minnet. ABC80 har alltså två grupper av kontrollsignaler för minnesaccess:

Internt	Externt
$\overline{\text{MRD}}$	$\overline{\text{XMEMFL}}$
$\overline{\text{WR}}$ och $\overline{\text{MREQ}}$	$\overline{\text{XMEMW}}$

Tack vare att externt inkopplade minnesenheter har egna signaler är det omöjligt att med feladresserade (felbyglade) minneskort störa det interna minnet. Detta underlättar givetvis användningen av ABC80.

Vi ska nu se hur signalerna $\overline{\text{XMEMFL}}$ och $\overline{\text{XMEMW}}$ genereras i ABC80. Låt oss gå tillbaks till fig 5.10. PROM:et i position E7 avkodar alla 1K-areor som är avsedda för yttre minnen (totalt 31K byte). Denna signal går vidare till MUX:en i position F5 som inkopplas (enablas) av $\overline{\text{MREQ}}$ och styrs av $\overline{\text{RD}}$.

I detta sammanhang är utsignalerna $\overline{\text{XM}}$ och $\overline{\text{XIN}}$ av intresse. Som framgår av fig 5.10 genereras $\overline{\text{XM}}$ av $\overline{\text{MREQ}}$ (oberoende av $\overline{\text{RD}}$). $\overline{\text{XM}}$ är alltså en signal som motsvarar $\overline{\text{MREQ}}$ men som endast uppträder när yttre minnesareor adresseras.

$\overline{\text{XIN}}$ genereras av $\overline{\text{MREQ}}$ och $\overline{\text{RD}}$ men enbart när yttre minnesarean adresseras. $\overline{\text{XIN}}$ motsvaras därför av interna minnets $\overline{\text{MRD}}$-signal. Även denna genereras ju (som vi tidigare visat i anslutning till fig 5.10) av MUX:en i position E7.

Fig 5.28 visar hur man i ABC80 genererar kontrollsignalerna till externa minnet:

$\overline{\text{XMEMFL}}$ = läs i externt minne
$\overline{\text{XMEMW}}$ = skriv i externt minne

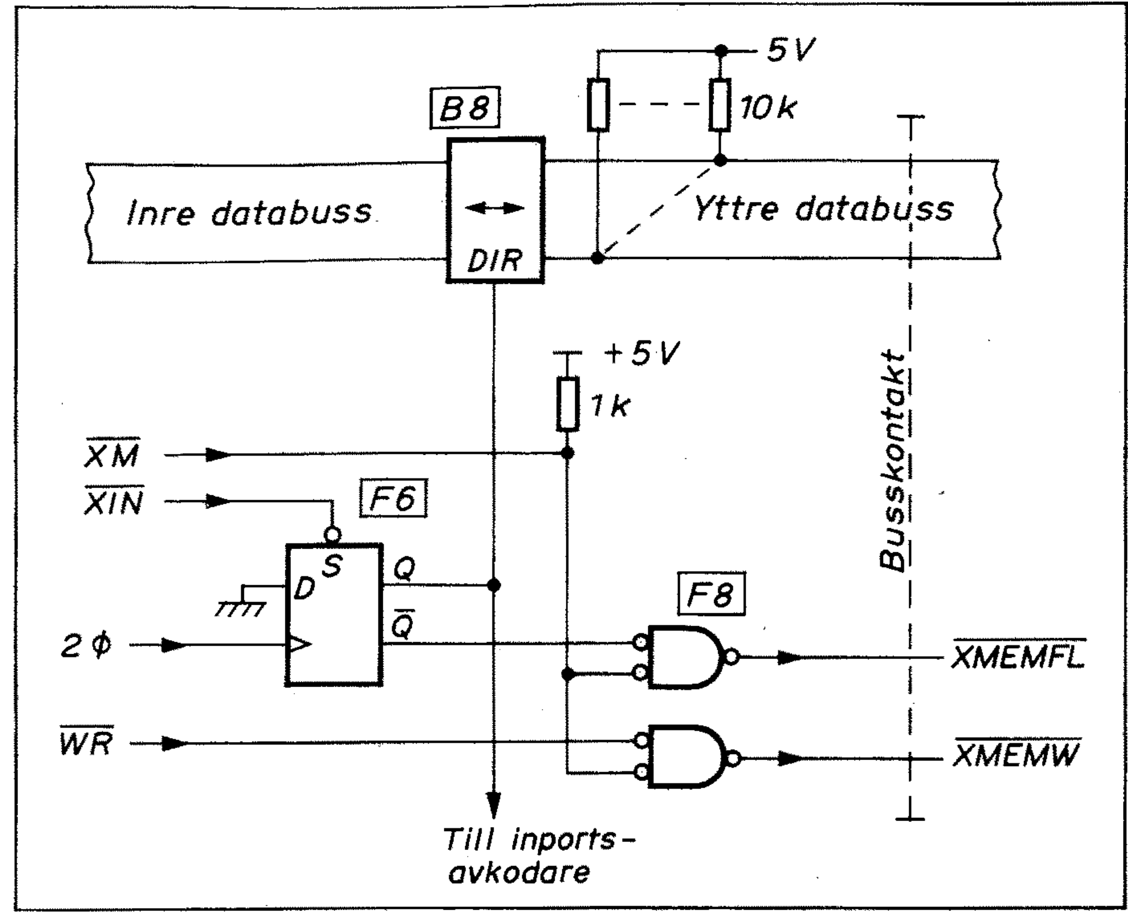


Fig 5.28 Generering av signalerna $\overline{\text{XMEMFL}}$ och $\overline{\text{XMEMW}}$ i ABC80

När det externa minnet adresseras får vi som framgått av fig 5.10 signalen $\overline{\text{XM}}$. Denna signal styr två signalgrindar (position F8). Om nu Z80 ger skrivkommando ($\overline{\text{WR}}$) kommer den nedre av dessa signalgrindar att ge signalen $\overline{\text{XMEMW}}$. Om Z80 istället ger läskommando till minnet $\overline{\text{XIN}}$ kommer den övre av signalgrindarna i fig 5.28 att avge signalen $\overline{\text{XMEMFL}}$.

När man läser i det externa minnet (eller från externa inportar) måste bufferten (bussdrivkretsen) mellan Z80:s databuss och den externa databussen vändas. Härvid är det kritiskt med tiden. För att få säker avläsning av yttre databussen styrs bussdrivkretsens riktningsingång (DIR) inte direkt från $\overline{\text{XIN}}$ utan via en latch (position F6) som klockas med dubbla klockfrekvensen 2Ø (= 6 MHz). Återvändningen efter $\overline{\text{XIN}}$ blir tack vare latchen något fördröjd.

"Timing" av kontrollsignaler i ett mikrodatorsystem är en konst som kräver mycket erfarenhet. Ovanstående är exempel på en av de många kretslösningar som tillgripits för att ABC80 ska fungera tillförlitligt.

6.1.2 Exempel på anslutning av yttre minne

I busskontakten på ABC80 finns som vi sett ovan adressbuss, databuss och de två kontrollsignalerna $\overline{XMEMFL}$ och $\overline{XMEMW}$. Vi kan därmed mycket enkelt ansluta yttre minnen.

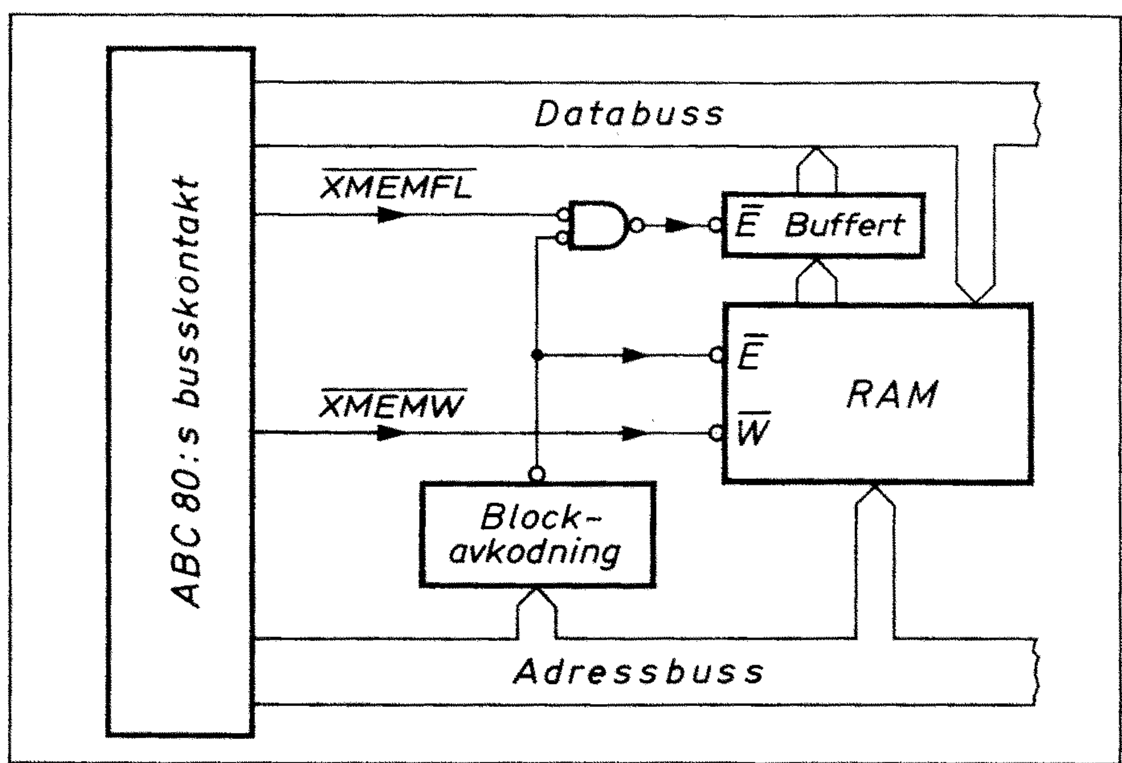


Fig 5.29 Principen för inkoppling av externt minne

6.2 Anslutning av yttre IO-enheter

Vi har tidigare sett hur in- och utportar fungerar. Fig 5.30 ger en sammanfattning. När CPU:n adresserar en in- eller utport lägger den ut dels en adress på adressbussen och dels en kontrollsignal på kontrollbussen. CPU:n Z80 avger kontrollsignalerna $\overline{IORQ}$ (för att särskilja IO-enheter från minnen) kombinerad med $\overline{RD}$ (= läs) eller $\overline{WR}$ (= skriv).

Ingångsavkodaren i fig 5.30 avkodar alltså kombinationen:

- o önskad adress
- o $\overline{IORQ}$
- o $\overline{RD}$

Utgångsavkodaren i fig 5.30 avkodar på motsvarande sätt: adress, $\overline{IORQ}$ och $\overline{WR}$.

Vid dessa signalkombinationer får ingångsgrinden respektive utgångslatchen "klarsignal", en signal som ofta kallas CE (= chip

enable), CS (= chip select) och E (= enable). Vanligen används "aktiv låg". I fig 5.30 framgår detta av ringarna på E-ingångarna till portarna.

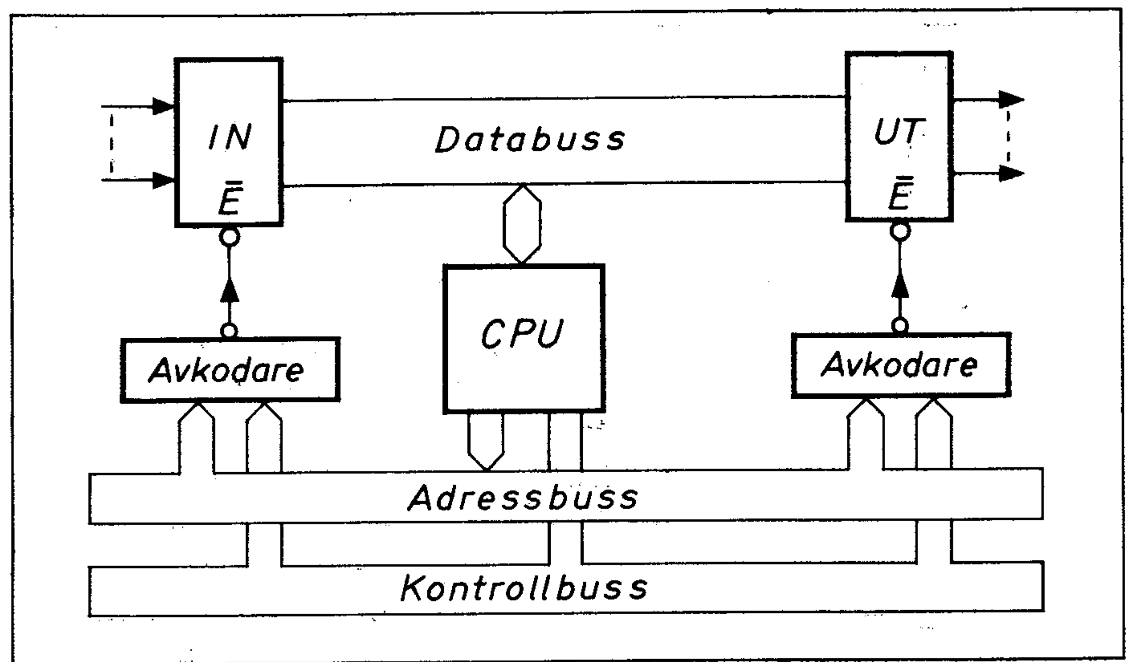


Fig 5.30 Adressering av in- och utportar

Det är tydligen två typer av klarsignaler som kan förekomma, en för inportar och en annan för utportar. Skillnaden mellan dessa signaler är att då inportar adresseras ligger databussen riktad inåt (mot CPU:n) men då utportar adresseras ligger databussen riktad utåt. Vi har tidigare sett att det ligger en buffert mellan interna databussen och busskontakten i ABC80. Det gäller alltså att denna buffert "hänger med" och vänder sig åt rätt håll.

6.2.1 Busskontaktens kontrollsignaler för IO-access

För att vi enkelt ska kunna koppla in IO-enheter till ABC80 innehåller busskontakten åtta klarsignaler, avkodade och färdiga att användas. Två av dessa avkodar inportar och sex avkodar utportar. De är tillgängliga på följande ben i busskontakten:

IN-portar

adress	ben nr
Ø	A17
1	A16

UT-portar

adress	ben nr
Ø	A22
1	A23
2	A21
3	A20
4	A19
5	A18

Det kan vara intressant att studera de kretsar i ABC80 som avger ovanstående klarsignaler (som i samband med ABC-bussen kallas kontrollsignaler).

6.2.2 Generering av kontrollsignaler i ABC80

Fig 5.31 visar hur de åtta klarsignalerna i ABC80:s busskontakt genereras.

Vi har tidigare i fig 5.18a sett hur ljudgeneratorn adresseras av en "en av åtta"-avkodare i position E9. Samma avkodare återser vi nu i fig 5.31. Det är alltså utgångarna Ø-5 från samma avkodare som ger kontrollsignalerna för UT-portarna i fig 5.30. Tack vare att $\overline{\text{PIOS}}$-signalen från inverteraren (position E6) tillförs en av utport-avkodarens E-ingångar undviker vi sammanblandning mellan PIO:n och utportarna i busskontakten.

D-vippan (position F6) i fig 5.31 har vi också mött tidigare (fig 5.28). Som framgår av fig 5.31 kan D-vippan även styras av en "en av fyra"-avkodare (position E4) via ett 220 Ω motstånd. Utsignal från denna avkodare kan endast uppträda vid $\overline{\text{IORQ}}$ (dvs vid IO-instruktioner) och endast vid kombinationen $\overline{\text{PIOS}}$ hög och $\overline{\text{RD}}$ låg (dvs vid läsning av allt utom PIO:n). På detta sätt adresseras enbart IN-portar (utan risk för kollision med PIO:n eller minnet).

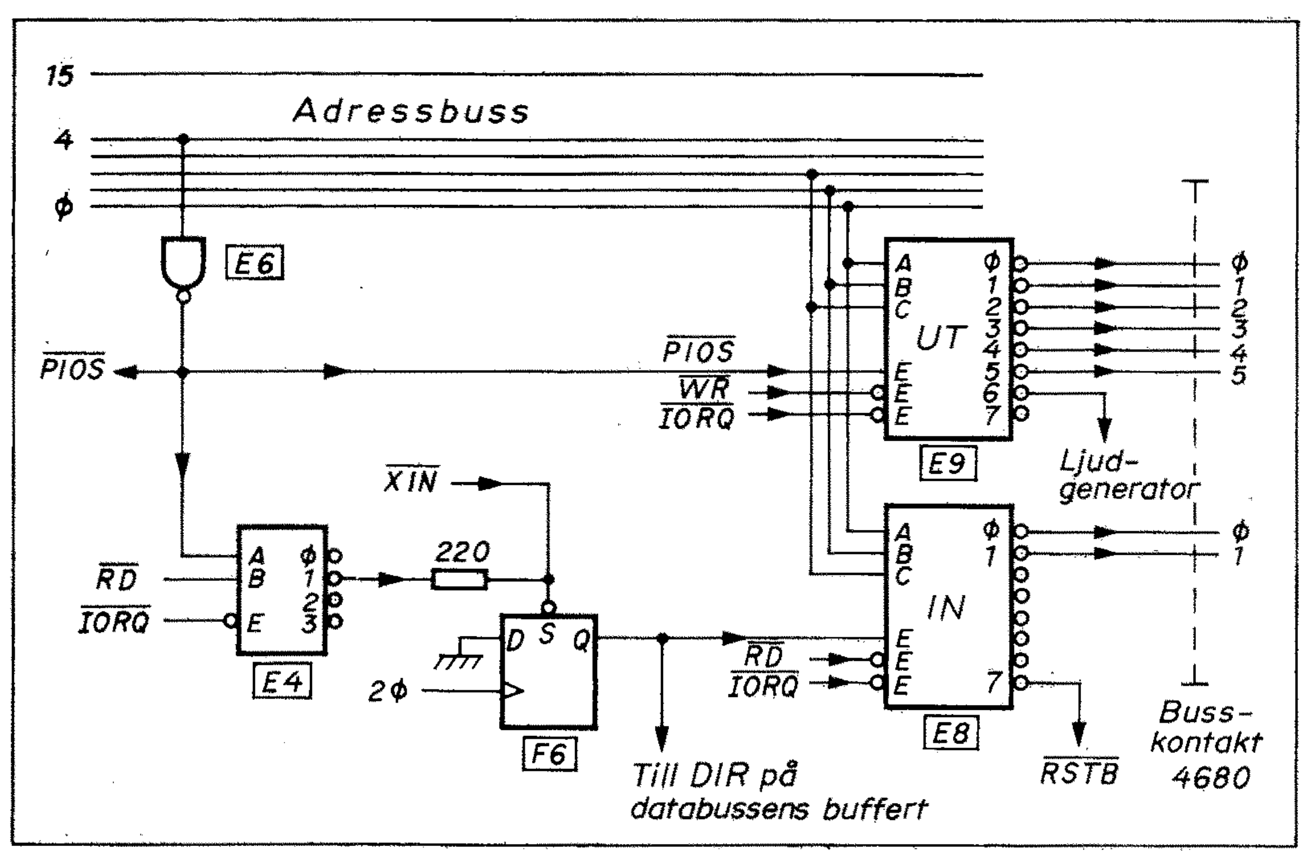


Fig 5.31 Generering av kontrollsignaler för IO-bussen

Med hjälp av kontrollsignalerna i busskontakten är det tydligen mycket enkelt att ansluta 2 inportar och 6 utportar till ABC80. Kontrollsignalerna kan emellertid användas på en mångfald olika sätt allt efter användarens egna önskemål. Om ABC80 ansluts till en "4680-buss" har de in- och utportar vi ovan behandlat speciella funktioner.

6.2.3 Exempel på anslutning av yttre IO-enheter

Fig 5.32 visar hur enkelt två utportar och en inport kan anslutas till ABC80:s busskontakt. Portarnas adresser är avkodade internt av ABC80 och motsvarande klarsignalledningar är tillgängliga i busskontakten.

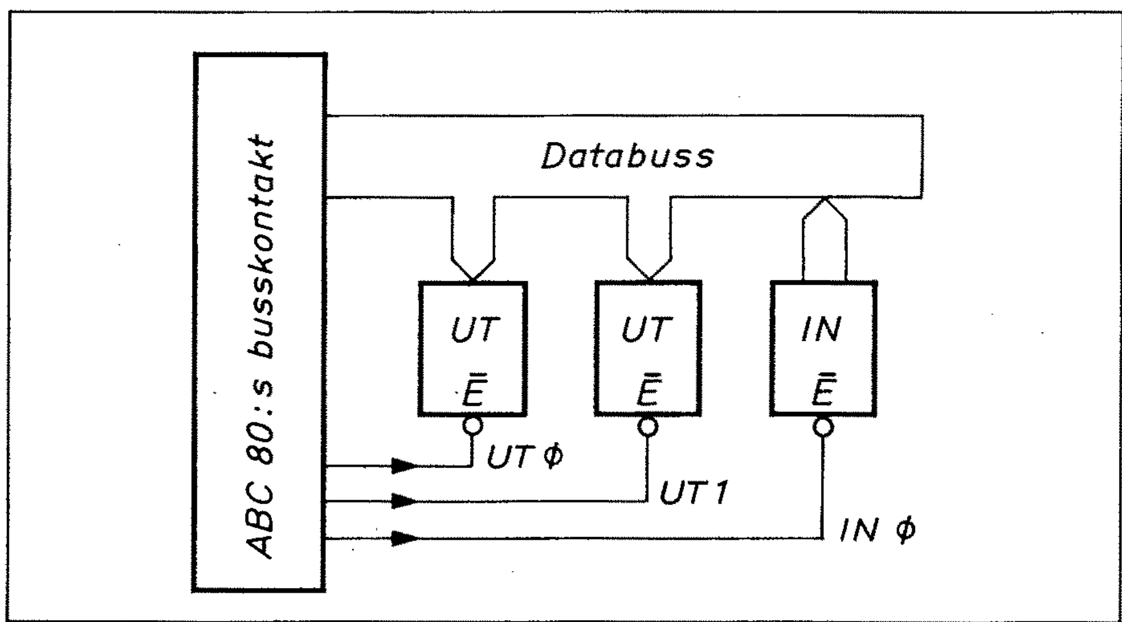


Fig 5.32 Exempel på anslutning av in- och utportar till ABC80:s busskontakt

Sammanfattning

Med utgångspunkt från ett enkelt blockschema med tre bussar (databussen, adressbussen och kontrollbussen) har vi nu bekantat oss med detaljerna kring minnesblocken, ljudgeneratorn och PIO:n. Vi har sett hur busskontakten i ABC80 ger oss möjlighet både att utöka minneskapaciteten och hänga på fler IO-enheter. Vi har också lärt känna flertalet av de kontrollsignaler som styr arbetet såväl internt inom ABC80 som externt via busskontakten.

6. Periferienheter

I flera kapitel har vi studerat hur en dator fungerar internt. Vi har också sett hur man med hjälp av inportar och utportar kan använda en dator till speciella uppgifter - exempelvis trafikstyrning.

I ett generellt datorsystem måste man ha bekväma in- och utkanaler så att man enkelt och effektivt kan programmera och köra datorsystemet. ABC80 har ett mycket avancerat tangentbord och en bildskärm som både kan hantera tecken (siffror och bokstäver) och grafisk information.

När man arbetar med ett datorsystem måste man bekvämt kunna lagra program och data. ABC80 har därför försetts med ett interface för kassettbandspelare.

Ett generellt datorsystem ska kunna anpassas till alla olika typer av IO-enheter. Genom att ansluta ABC80 till 4680-bussen kommer man mycket nära detta mål med ABC80-systemet.

1. Tangentbordet

Fig 6.1 visar tangenternas placering. Tangentbordet innehåller en del tecken som man inte känner igen från skrivmaskinernas tangentbord. Det gäller bl a pilarna nere till höger (högerpil och vänsterpil) samt tecknen "större än" och "mindre än" längst uppe till höger.

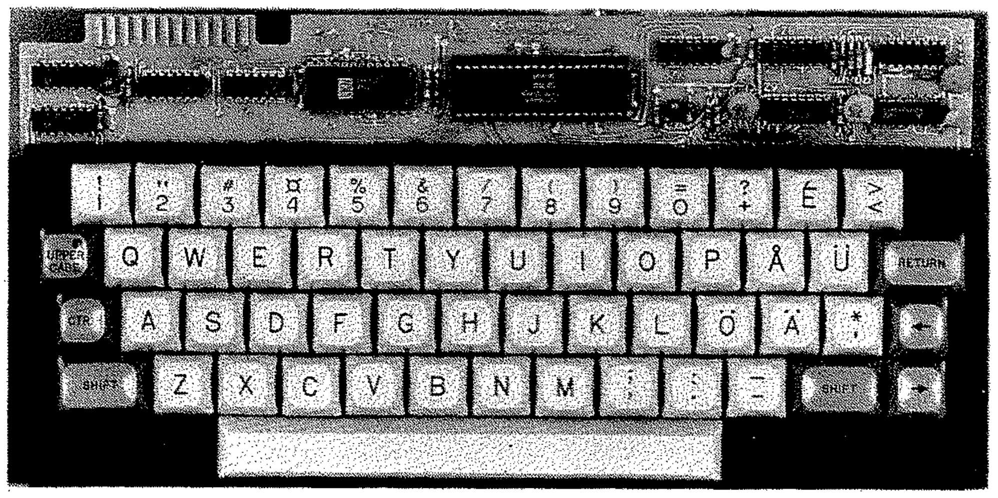


Fig 6.1 Tangentbordet i ABC80

1 2 3 4 5 6 7 8 9 10 11 12 13

19 20 21 22 23 24 25 26 27 28 29 30 31 32

37 38 39 40 41 42 43 44 45 46 47 48 49 50

55 56 57 58 59 60 61 62 63 64 65 66 67

74

a.

b.

KEY NUMBER	UNSHIFT OCT	UNSHIFT HEX	SHIFT OCT	SHIFT HEX	CONTROL OCT	CONTROL HEX	CONTROL-SHIFT OCT	CONTROL-SHIFT HEX
1	061	31	041	21	061	31	041	21
2	062	32	042	22	062	32	042	22
3	063	33	043	23	063	33	043	23
4	064	34	044	24	064	34	044	24
5	065	35	045	25	065	35	045	25
6	066	36	046	26	066	36	046	26
7	067	37	057	2F	067	37	057	2F
8	070	38	050	28	070	38	050	28
9	071	39	051	29	071	39	051	29
10	060	30	075	3D	060	30	075	3D
11	053	2B	077	3F	053	2B	077	3F
12	140	60	100	40	000	00	000	00
13	074	3C	076	3E	177	7F	177	7F
20	161	71	121	51	021	11	021	11
21	167	77	127	57	027	17	027	17
22	145	65	105	45	005	05	005	05
23	162	72	122	52	022	12	022	12
24	164	74	124	54	024	14	024	14
25	171	79	131	59	031	19	031	19
26	165	75	125	55	025	15	025	15
27	151	69	111	49	011	09	011	09
28	157	6F	117	4F	017	0F	037	1F
29	160	70	120	50	020	10	000	00
30	175	7D	135	5D	035	1D	035	1D
31	176	7E	136	5E	036	1E	036	1E
32	015	0D	015	0D	015	0D	015	0D
38	141	61	101	41	001	01	001	01
39	163	73	123	53	023	13	023	13
40	144	64	104	44	004	04	004	04
41	146	66	106	46	006	06	006	06
42	147	67	107	47	007	07	007	07
43	150	68	110	48	010	08	010	08
44	152	6A	112	4A	012	0A	012	0A
45	153	6B	113	4B	013	0B	033	1B
46	154	6C	114	4C	014	0C	034	1C
47	174	7C	134	5C	034	1C	034	1C
48	173	7B	133	5B	033	1B	033	1B
49	047	27	052	2A	047	27	052	2A
50	010	08	010	08	010	08	010	08
56	172	7A	132	5A	032	1A	032	1A
57	170	78	130	58	030	18	030	18
58	143	63	103	43	003	03	003	03
59	166	76	126	56	026	16	026	16
60	142	62	102	42	002	02	002	02
61	156	6E	116	4E	016	0E	036	1E
62	155	6D	115	4D	015	0D	035	1D
63	054	2C	073	3B	054	2C	073	3B
64	056	2E	072	3A	056	2E	072	3A
65	055	2D	137	5F	055	2D	137	5F
67	011	09	011	09	011	09	011	09
74	040	20	040	20	040	20	040	20

Fig 6.2 Tangenternas koder
a. Tangentnumrering
b. Koder (fyra alternativ beroende på SHIFT- och CTRL-tangenterna)

Return (eller vagnretur) är en viktig tangent i alla datorterminaler. Vi ser return-tangenten till höger i fig 6.1. Till vänster sitter en tangent med beteckningen CTRL, dvs kontroll. Den ska kombineras med vissa andra tangenter och ger då (liksom skifttangenterna) speciella funktioner för de tangenter man trycker ner.

På en vanlig skrivmaskin har man ett "skiftlås" så att man kan skriva stora bokstäver (versaler) utan att ständigt trycka på skifttangenten. ABC80 har ett elektroniskt skiftlås, "UPPER CASE". Det är en vippa som omväxlande ställs i det ena eller andra läget. I normalläget är den i tangenten inbyggda lysdioden släckt och i versal-läget är lysdioden tänd. Funktionen avviker från ett enkelt mekaniskt skiftlås. När lysdioden lyser ger alla bokstavstangenter versaler men alla teckentangenter har kvar sin vanliga funktion. UPPER CASE-tangenten fungerar alltså enbart för bokstäver och när dess lysdiod lyser kan man använda de vanliga skifttangenterna för skift av tecken, exempelvis mellan 1 och !.

Tecknet ovanför 4 kallas "sol" och ersätter det vanliga dollartecknet $. Tecknet ovanför 3 kallas vanligen "nummertecken" eller "brädgård" (efter kartverkets symbol för brädgård).

Hur har man egentligen placerat bokstäverna Å, Ä och Ö? ABC80 använder här den nya europeiska standarden (SIS 662241). Så småningom vänjer sig vårt högra lillfinger!

Ja, det är många synpunkter som kan läggas på ett tangentbord. Vi ska begränsa oss till koderna och den elektroniska funktionen.

1.1 Koder från tangentbordet

Tangentbordet i ABC80 avkodas av en speciell avkodarkrets som fabrikanten (KEY TRONIC) kallar microcontroller ("logic engine"). Det är den stora kapseln ovanför tangenterna i fig 6.1. När man använder sig av specialkretsar kan man få komplexa funktioner utförda. Avkodarkretsen på tangentbordet kan ge fyra olika koder för varje tangent

- o utan SHIFT
- o med SHIFT
- o med CTRL
- o med både SHIFT och CTRL

Avkodarkretsen innehåller ett läsminne (ROM) uppdelat i fyra sidor, en för vardera av ovanstående fyra kodserier.

Fig 6.2a visar en numrering av tangenterna och fig 6.2b motsvarande koder.

Tangentbordet i ABC80 lämnar avsevärt fler koder än vad som behövs för ABC80. Detta kostar emellertid inget extra eftersom kapseln masstillverkas med dessa läsminnen inbyggda. Vi ska senare göra ett program med vars hjälp vi kan kontrollera koderna från tangentbordet.

1.2 Elektroniken i tangentbordet

För att minska antalet ledningar brukar tangentbord utföras i form av en matris. Tangenterna påverkar switchar i ledningarnas krysspunkter.

Fig 6.3 visar ett exempel på ett matrisuppbyggt tangentbord. I tur och ordning läggs en av de fyra kolumnledningarna XØ-X3 låg medan övriga hålls höga. Detta kallas ofta avsökning (scanning). Radledningarna YØ-Y3 i fig 6.3 ligger normalt höga på grund av pull upmotstånd till matningsspänningen.

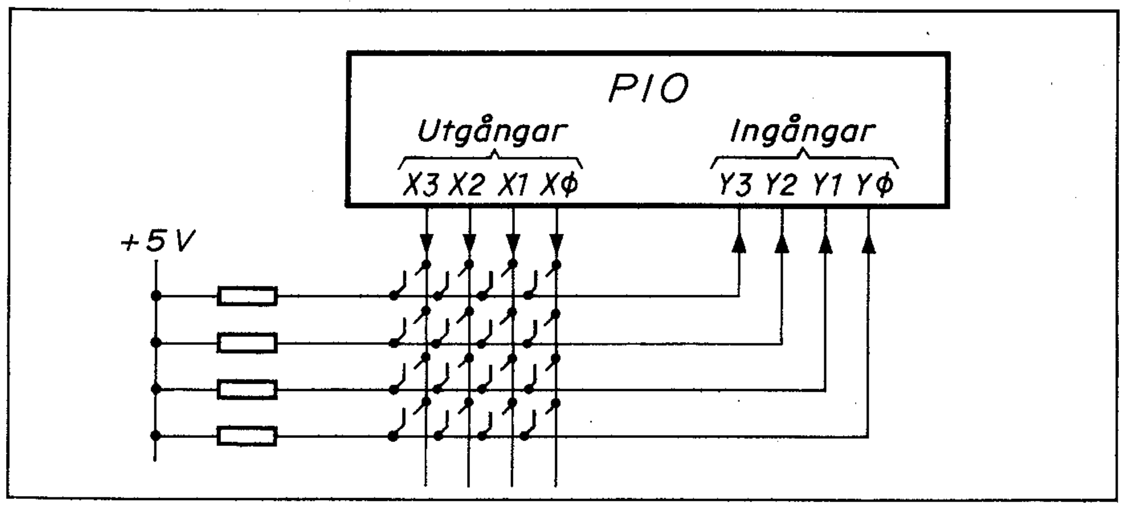


Fig 6.3 Principen för ett tangentbord uppbyggt på en matris

Om en tangent trycks ned kommer motsvarande Y-ledning att gå låg under avsökningspulsen på den X-ledning till vilken tangenten är inkopplad.

Med 8 ledningar (4 scannas och 4 avläses) kan man på detta sätt avkänna 16 tangenter. Med 20 ledningar kan man avkänna hundra tangenter.

Både scanning och avläsning utförs i fig 6.3 med hjälp av en PIO som programmerats på lämpligt sätt. Mekaniska switchar ger kontaktstudsar vid slutning och brytning. För att inte läsa fel låter man ofta PIO:n göra två avläsningar med några millisekunders tidsintervall. Om avläsningarna ger samma resultat anses det avlästa värdet vara korrekt.

Givetvis kostar det tid för en mikrodator att på detta sätt ständigt ligga och sända ut signaler och sedan läsa av i hopp om tangentnedtryck. Ska mikrodatorn dessutom ligga i vänteslinga för att förhindra inverkan av kontaktstudsar (debouncing) tar det ytterligare CPU-tid. I ABC80 har man löst tangentbordsavkodningen med scanning, men här används kapacitiva givare istället för mekaniska switchar och tangentbordet har som nämnts en egen avkodarkrets (microcontroller).

Kapacitiva tangenter

ABC80 har ett "kapacitivt tangentbord". Det betyder att krysspunkterna i matrisen utgörs av kondensatorer som har en mycket låg kapacitans (ca 1 pF) när tangenten är i normalläget (uppe), men som har högre kapacitans (ca 20 pF) när tangenten är nedtryckt. Fig 6.4a visar hur en krysspunkt på tangentbordet utformats som två halvcirkelformiga kondensatorbelägg.

Tangenten påverkar ett tredje isolerat kondensatorbelägg som när tangenten är uppe ligger på ca 2,5 mm avstånd från kretskortet. Kapacitansen mellan de halvcirkelformiga beläggen blir då mycket låg.

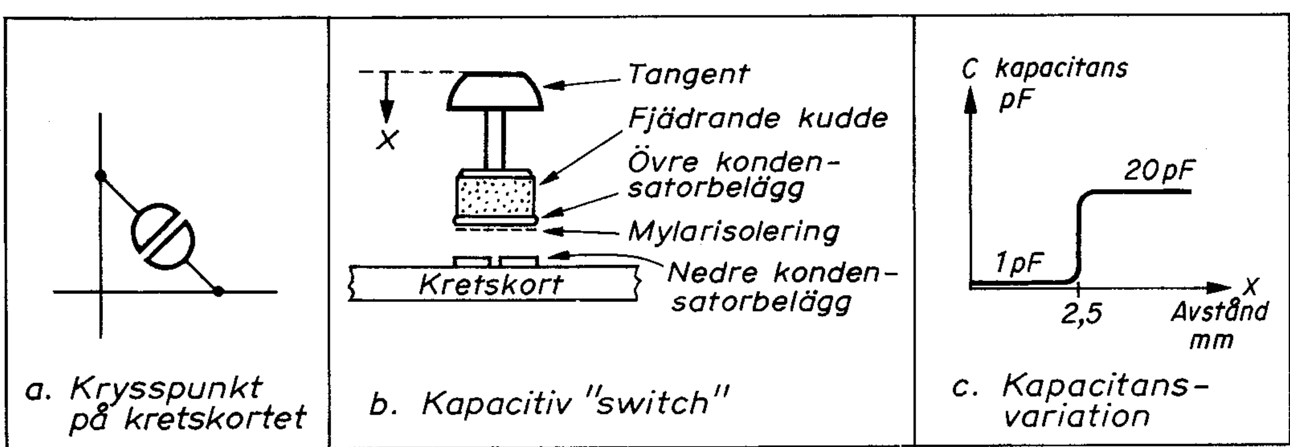


Fig 6.4 Principen för kapacitiva tangenter
a. Krysspunktens utformning
b. Tangent som styr kapacitans
c. Kapacitansvariation vid olika nedtryckning av tangent

Fig 6.4c visar hur kapacitansen växer som funktion av tangentens nedtryck (x mm). När tangenten tryckts ned till 2,5 mm ligger övre kondensatorbelägget mot de två undre och kapacitansen ökar snabbt till 20 pF. Ytterligare nedtryck (dvs $x > 2,5$ mm) tas upp av en liten fjädrande plastkudde som sitter mellan tangenten och övre kondensatorbelägget. Tangenten hålls uppe av en stålfjäder (som ej utritats i fig 6.4b).

När tangenten är nedtryckt utgör den tunna mylar-isolationen på övre kondensatorbelägget dielektrikum (isolation) i två seriekopplade kondensatorer. Resulterande kapacitansen blir ca 20 pF.

En så liten kapacitans som 20 pF kan inte avkännas med långa pulser (likspänning) på det sätt som visades i fig 6.3. Här krävs scanning med korta pulser (spänningssprång) om vi ska kunna avkänna kapacitansen.

Vi ska först visa principen för scanningen av tangentbordet i ABC80 och därefter ska vi se hur själva avsökningspulserna har utformats.

Principen för tangentbordets scanning

Fig 6.5 visar i princip hur kretsarna i ABC80:s tangentbord fungerar. Överst ser vi en 16-räknare som matas med pulser från en klocka.

Räknarens fyra utgångar tillförs en latch. De två lägsta utgångarna (Ø och 1) tillförs dessutom till adressingångarna till en fyra ingångars multiplexer (MUX). De två högsta utgångarna (2 och 3) går till en "en av fyra"-avkodare.

Avkodaren i fig 6.5 fungerar som "scanner" och lägger i tur och ordning ut pulser (dvs låg spänning) på kolumnledningarna XØ-X3.

För varje sådan kolumnpuls ges MUX:en fyra adresser (Ø, 1, 2 och 3).

Vi tänker oss nu först att krysspukterna har mekaniska switchar och Y-ledningarna pull up-motstånd. En nedtryckt tangent ger sig då tillkänna som en puls vid den tidpunkt när räknaren just innehåller tangentens adress. Låter vi nu signalen från MUX:en klocka latchen så får vi tangentens adress lagrad på latchens utgång.

Vill vi nu ha en speciell kod på utgången från tangentbordet låter vi latchen adressera ett ROM. Detta läsminne kan vi programmera att avge godtyckliga kombinationer för varje inadress.

Låt oss slutligen lägga till två extra tangenter som vi kallar SHIFT och CTRL: De får styra var sin adressingång på ROM:et. Vi får på detta sätt fyra olika koder för varje tangent, beroende på hur SHIFT- och CTRL-tangenterna ligger.

Om vi vill ha 7 bitars ASCII-kod på utgången måste vårt ROM (som i fig 6.5 avkodar 16 tangenter) innehålla $2^6 = 64$ adresser och varje adress ett 7 bitars ord. Detta ROM måste alltså ha kapaciteten

$$64 \times 7 = 448 \text{ bitar.}$$

Nu har vi klarat ut principen för scanning av tangentbordet. Om vi ersätter de mekaniska switcharna med kondensatorer måste vi först och främst ändra scannern så att den avger korta branta pulser.

Fig 6.5 Principen för scanning av ABC80:s tangentbord

Flankerna ger då utsignaler på Y-ledningarna av den typ som visas i fig 6.6 visar. MUX:en måste nu väljas med omsorg så att den inte dämpar de små signalerna. För att få säker klockning av latchen måste vi koppla in en känslig förstärkare mellan MUX:en och latchen.

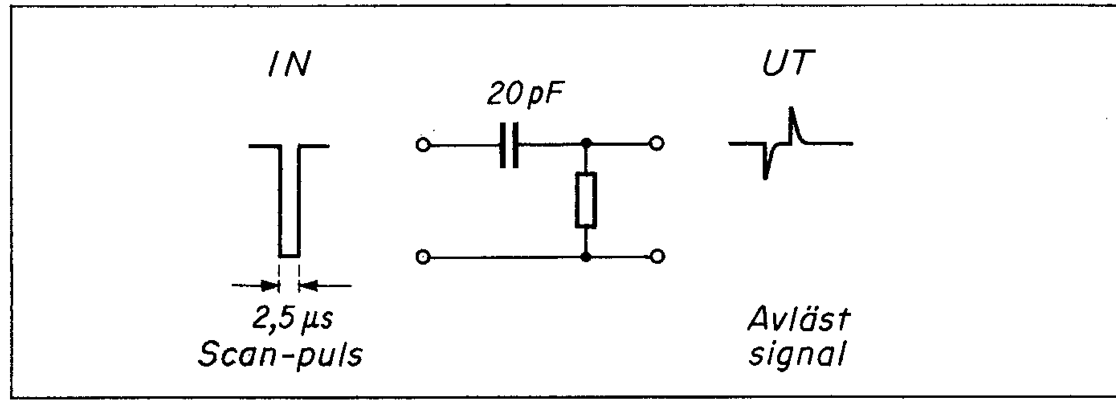


Fig 6.6 Scan-puls och avläst signal i kapacitivt tangentbord

Det praktiska utförandet

Tillverkaren av tangentbordet har valt att lägga elektroniken i kapslar enligt fig 6.7. Avkodarkretsen (som kallas "logic engine") är tillverkad i NMOS-teknik och innehåller klocka, räknare, avkodare, latch och ROM. MUX:en är en CMOS-krets (4051) med ytterst låga läckströmmar. Förstärkaren är en "hemlig" specialkrets. Den är enligt fabrikantens uppgift tillverkad i bipolarteknik, har låg inimpedans och är konstruerad för att avkänna strömpulser. Förstärkaren måste reagera på laddningar av storleken 20 pF x 5 V = 100 pC.

Kunden kan beställa avkodarkretsen med godtyckliga koder inbyggda. Avkodarkretsen avkänner 16 kolumner och för varje kolumn ges 8 radadresser. Det räcker till totalt 8 x 16 = 128 tangenter.

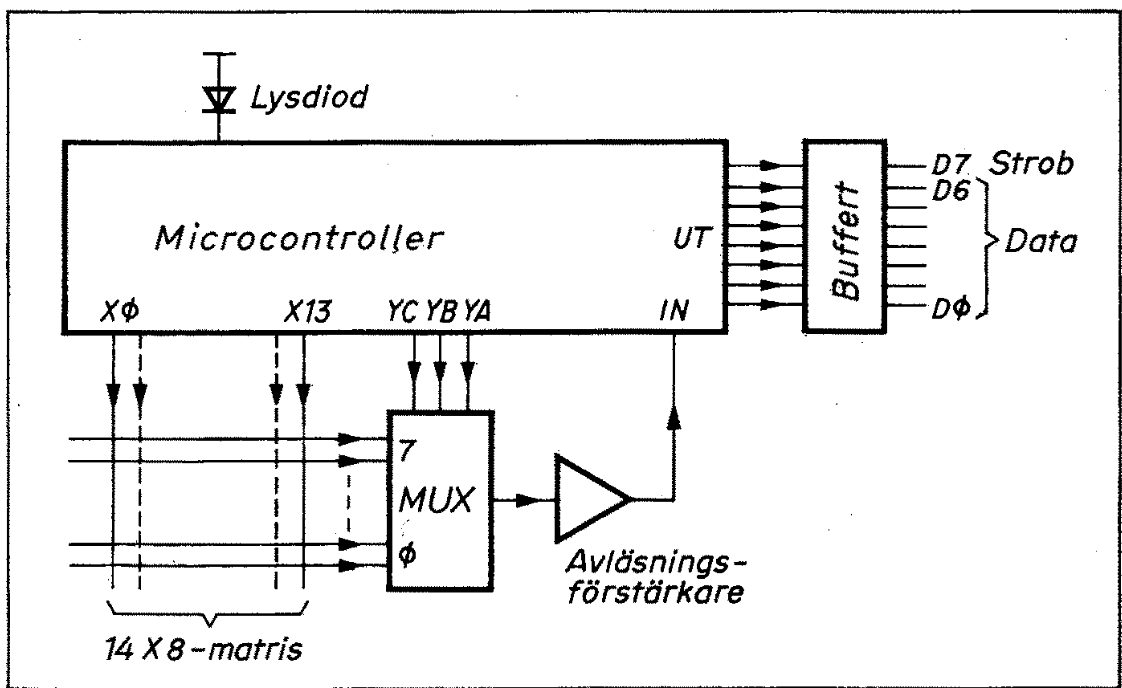


Fig 6.7 De viktigaste kretsarna i ABC80:s tangentbord

I fig 6.8 jämförs avsökningspulser för tangentbord med mekaniska och kapacitiva switchar. Den långa pulsen för de mekaniska switcharna har i ABC80:s tangentbord ersatts av åtta dubbelpulser om vardera 2,5 μs enligt fig 6.8b. En dubbelpuls används för avläsningen av varje rad. MUX:en ger ny adress före varje ny dubbelpuls.

För att få säker avläsning är avkodarkretsen (microcontroller) så utförd att den avger utsignal först sedan två efterföljande avsökningar (vardera med två pulser) givit samma resultat.

Eftersom kontrollkretsen är utförd i NMOS-teknik tål den inte alltför stor belastning. Man har därför lagt in en buffert mellan kontrollkretsen och utgången.

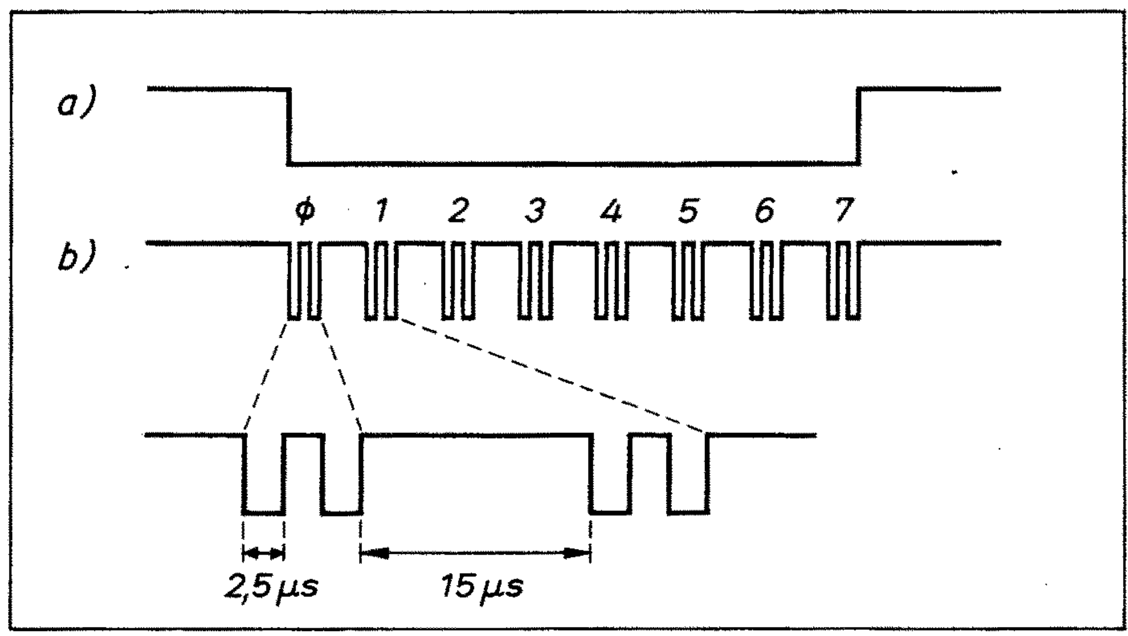


Fig 6.8 Scanpulser i tangentbord med
a. Mekaniska switchar
b. Kapacitiva "switchar"

Fig 6.9 visar kretskortet med kondensatorbelägg för kapacitiva tangenter och med påmonterade kretsar. Tangenterna med de övre kondensatorbeläggen sitter i en plåtram som enkelt kan skruvas loss (för reparation eller utbyte av tangenter). I fig 6.9 är tangenterna borttagna för att vi ska se kondensatorbeläggen i matrisens kryss-punkter.

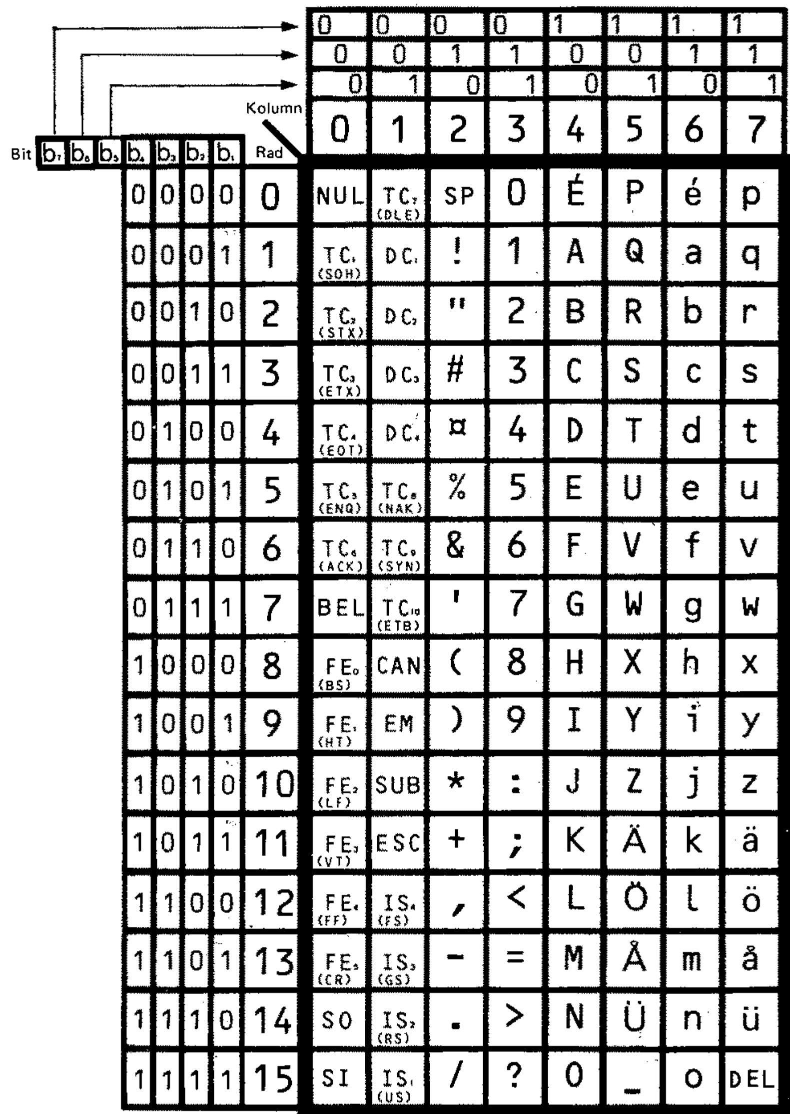
Många äldre tangentbord tappar information om man trycker ner två tangenter i snabb följd. Problem med mekaniska switchar och kontaktstudsar m m gör att man i många sådana tangentbord stoppar scanningen när en tangent nedtryckts. Man måste därför släppa föregående tangent innan nästa nedtrycks. (2-key roll-over).

ABC80:s tangentbord har vad som brukar kallas "N-key roll-over". Då avläses tangentnedtryck utan att scanningen stoppas. På ABC80 skulle man kunna skriva med en hastighet av ca 30 nedslag i sekunden utan att funktionen uteblev (avsökning sker med intervall på 2,5 ms).

Tangentbordet avger 7 bitars ASCII-kod enligt fig 6.10. Dessutom ges en strobsignal (bit 7), som indikerar att någon tangent är nedtryckt och att giltigt data finns på DØ - D6. Databitarna hålls latchade tills nästa tangent avkodats. För att undvika felaktiga avläsningar vid kodväxling, ska avläsning endast ske när strobpuls ges.

Buffert Microcontroller MUX Förstärkare

Fig 6.9 Tangentbordets kretskort med kondensatorbelägg och komponenter



b_7					0	0	0	0	1	1	1	1
b_6					0	0	1	1	0	0	1	1
b_5					0	1	0	1	0	1	0	1
Bit b_4	b_3	b_2	b_1	Rad \ Kolumn	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	TC$_7$ (DLE)	SP	0	É	P	é	p
0	0	0	1	1	TC$_1$ (SOH)	DC$_1$	!	1	A	Q	a	q
0	0	1	0	2	TC$_2$ (STX)	DC$_2$	"	2	B	R	b	r
0	0	1	1	3	TC$_3$ (ETX)	DC$_3$	#	3	C	S	c	s
0	1	0	0	4	TC$_4$ (EOT)	DC$_4$	¤	4	D	T	d	t
0	1	0	1	5	TC$_5$ (ENQ)	TC$_8$ (NAK)	%	5	E	U	e	u
0	1	1	0	6	TC$_6$ (ACK)	TC$_9$ (SYN)	&	6	F	V	f	v
0	1	1	1	7	BEL	TC$_{10}$ (ETB)	'	7	G	W	g	w
1	0	0	0	8	FE$_0$ (BS)	CAN	(	8	H	X	h	x
1	0	0	1	9	FE$_1$ (HT)	EM	)	9	I	Y	i	y
1	0	1	0	10	FE$_2$ (LF)	SUB	*	:	J	Z	j	z
1	0	1	1	11	FE$_3$ (VT)	ESC	+	;	K	Ä	k	ä
1	1	0	0	12	FE$_4$ (FF)	IS$_4$ (FS)	,	<	L	Ö	l	ö
1	1	0	1	13	FE$_5$ (CR)	IS$_3$ (GS)	-	=	M	Å	m	å
1	1	1	0	14	SO	IS$_2$ (RS)	.	>	N	Ü	n	ü
1	1	1	1	15	SI	IS$_1$ (US)	/	?	O	_	o	DEL

Fig 6.10 Svensk variant av ASCII-koden (SIS 63 61 27)

2. Videointerfacet

ABC80 har en bildskärm som ger tecken enligt det nya textsändningssystemet för TV (teletext och wiew data). Bildskärmen har 24 rader och varje rad rymmer 40 tecken. Totalt ger detta 40 x 24 = 960 tecken.

För att bildskärmen ska kunna kommunicera med datorn har man reserverat 1K byte av arbetsminnet som bildminne. Där placerar CPU:n data som ska visas på bildskärmen.

Varje byte i bildminnet lagrar ett tecken. Det är videointerfacets uppgift att läsa bildminnet, avkoda varje tecken och placera det på avsedd plats på bildskärmen. Detta sker med hjälp av en bildsignal och en synksignal. Fig 6.11 sammanfattar videointerfacets uppgift.

Bildsignalen och synksignalen från ABC80 kan enkelt kombineras till en videosignal enligt europeiskt standard. Med en enkel tillsats kan ABC80 därmed använda vanliga TV-minitorer som extra bildskärmar.

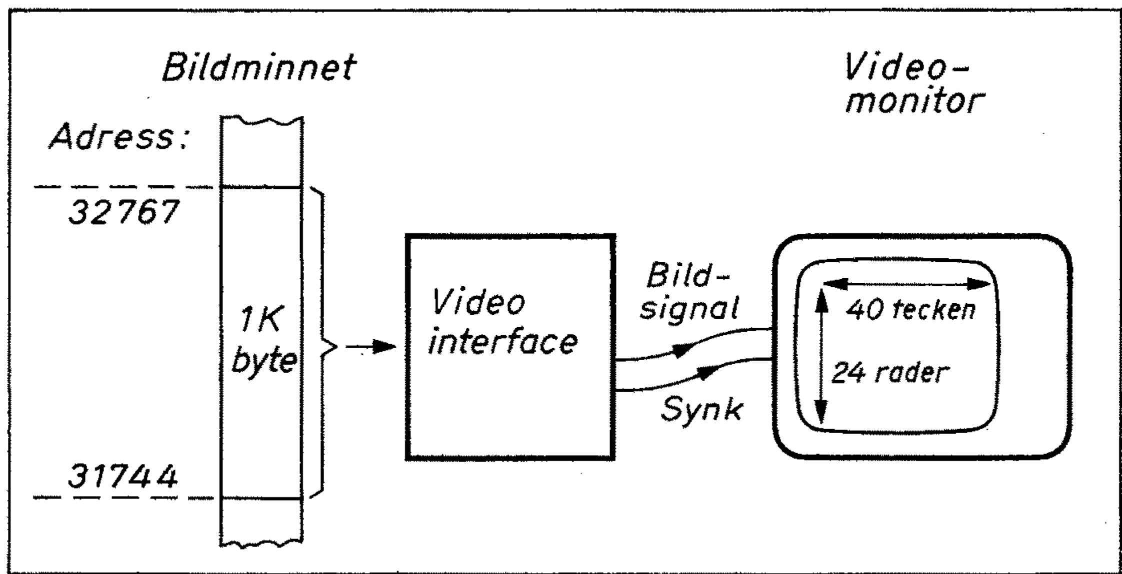


Fig 6.11 Videointerfacets uppgift

Videointerfacet i ABC80 innehåller ett flertal kretsar. Vi ska börja med att beskriva räknarna, som delar ner klockfrekvensen till linje- och bildsignaler. Vi ska sedan se hur videointerfacet adresserar bildminnet och omformar dess innehåll till en videosignal som ger önskade tecken på bildskärmen.

En vanlig europeisk TV-bild avsöks 25 gånger per sekund med 625 linjer per bild. För att TV-bilden inte ska flimra avsöks den i två etapper, först med udda radnummer och sedan med jämna radnummer. Detta kallas radsprång (interlace). Man får alltså 50 bilder per sekund men dessa bilder är var för sig inte "heltäckande" utan täcker bara 312,5 av de 625 linjerna.

När en TV-skärm ska användas för presentation av tecken får man en bättre bild om man enbart använder 312 linjer men kör ut 50 kompletta bilder per sekund (50 bilder/s utan radsprång). Vid normal TV-mottagning är det synkpulserna från TV-sändaren som bestämmer när varje rad ska börja och därmed hur radsprånget utförs. I ABC80 ligger synkpulser och bildsignal så fördelade i tid att man får 50 bilder per sekund utan radsprång.

Eftersom en videomonitor återger linjer i takt med linjesynkpulserna märker den ingen skillnad på signaler med eller utan radsprång. Det behövs alltså ingen omställning eller omjustering av en videomonitor om man omväxlande vill mata den med vanlig TV-bild (25 bilder/s och radsprång) eller med text från ABC80 (50 bilder/s utan radsprång).

Varje tecken i ABC80 avbildas med hjälp av en punktmatris som upptar 5 x 9 punkter. Fig 6.12. För att tecknen ska åtskiljas har man en blank punktkolumn mellan varje tecken och en blank punktlinje mellan varje rad. Varje teckenplats på skärmen upptar därför 6x10 punkter. TV-skärmen måste alltså ha en upplösning av 24x10 = 240 punktlinjer och 40x6 = 240 punktkolumner, dvs totalt 57.600 punkter. Varje sekund ska tydligen 50x57.600 = 2.880.000 punkter placeras på skärmen. Det är räknarnas uppgift att hålla reda på alla dessa punkter!

Som framgår av fig 6.12 till höger är varje teckenplats uppdelad i 6 fält. Dessa kan tändas i olika kombinationer om bildskärmen körs i grafisk mod.

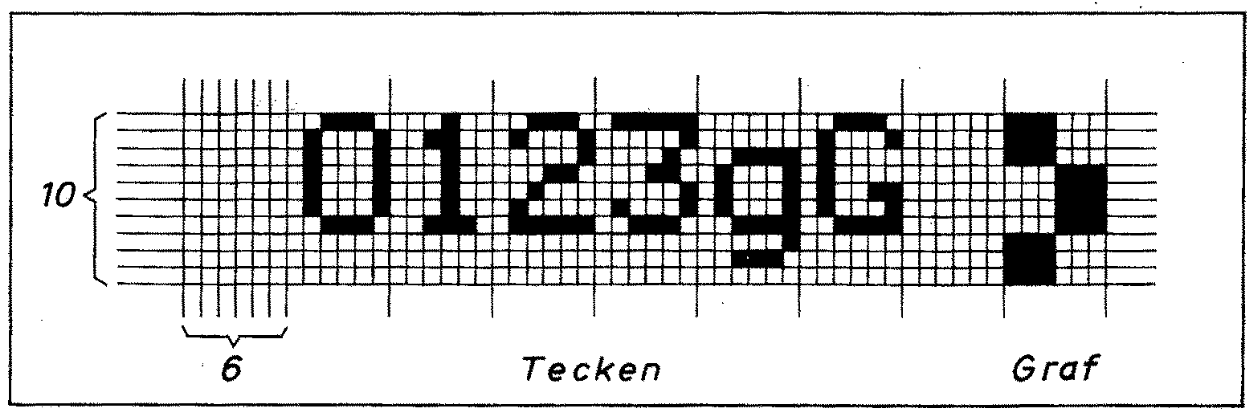


Fig 6.12 Punktmatriser (5x9) för några tecken och en graf

Den som vill se exempel på hur elektroniska räknare kan användas får sitt lystmäte i videointerfacet i ABC80. Principen framgår av fig 6.13. För att förstå räknarnas funktion måste vi först studera bildens uppbyggnad. 50 bilder per sekund och 312 linjer per bild ger oss direkt tiden 64 μs för en linje. Som bekant avsöks (scannas) en TV-bild med vågräta linjer från vänster till höger. När elektronstrålen nått högra bildkanten släcks den. Det tar nu en viss tid för horisontalavlänkningen att återgå till utgångsläget för nästa svep. Därtill återgår ca 12 μs av totala linjetiden på 64 μs.

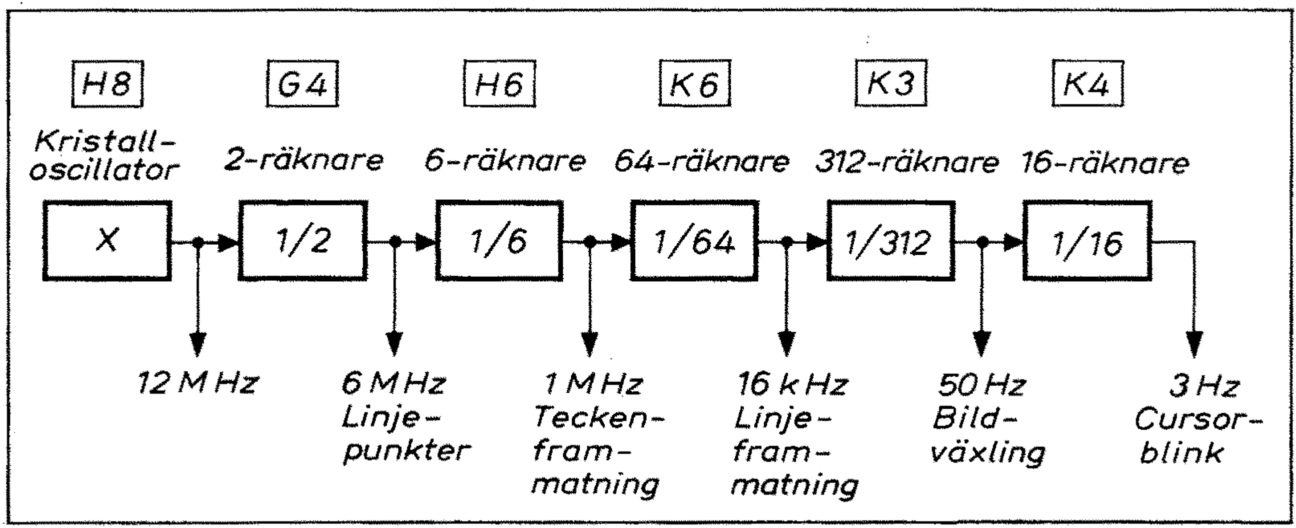


Fig 6.13 Några av räknarna i videointerfacet

När en bild har avsökts och strålen just avverkat den nedersta linjen på bildskärmen tar det på motsvarande sätt viss tid för vertikalavlänkningen att återställa strålen till utgångsläget för nästa bild. I praktiken åtgår det en tid som svarar mot 25 linjer. Av de 312 linjerna blir tydligen endast 287 synliga på skärmen. Vid sändning av teletext (text-TV) används 2 av de 25 osynliga linjerna för textöverföring. Var och en av dessa linjer överför därvid en rad (40 tecken) kodad med 8 bitar per tecken.

Bildskärmen har avrundade hörn och kan inte användas ända ut till kanterna om man vill ha läsbara tecken. Det användbara området omfattar därför den inre streckade rektangeln i fig 6.14. Det ser kanske litet lustigt ut att ange bildens bredd till 40 μs och bildens höjd till 240 linjer, men i vårt fall är det praktiskt!

Kristalloscillatorn till vänster i fig 6.13 ger en frekvensstabil signal på 12 MHz. (För att slippa interferensfenomen med nätfrekvensen vill man ha bildväxlingsfrekvensen exakt 50 Hz och därför har

man lagt kristalloscillatorfrekvensen på 11, 9808 MHz). Det är den högsta frekvensen i ABC80.

Med hjälp av en vippa (position G4) delas frekvensen ned till 6 MHz vilket ger 6 horisontellt placerade punkter för varje linje i ett tecken (fig 6.12).

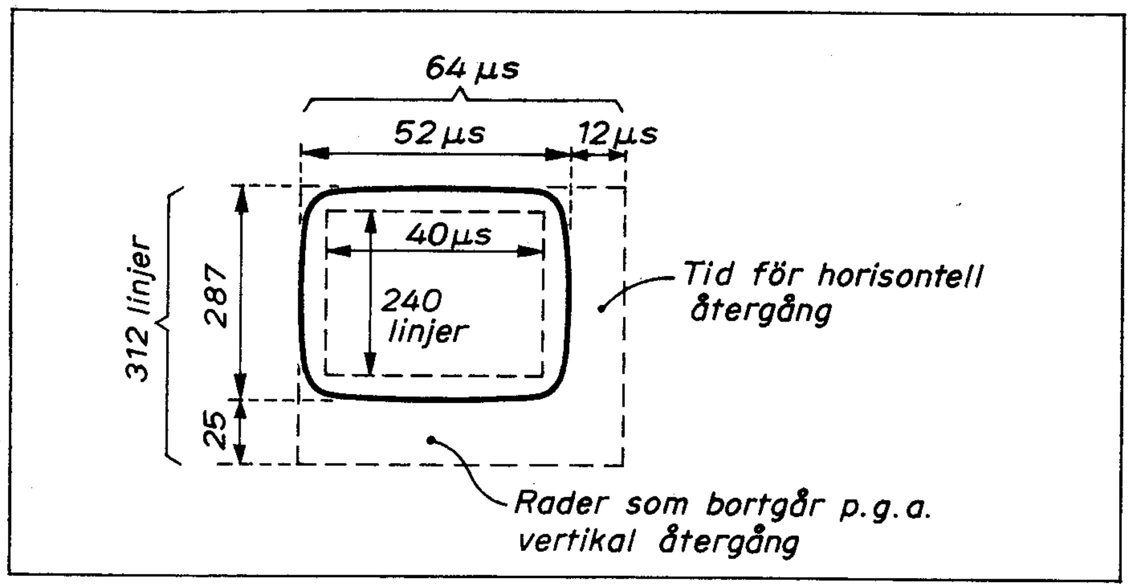


Fig 6.14 TV-bilden med bredden mätt i µs och höjden i linjer

Med en 6-räknare delas frekvensen ytterligare ned till 1 MHz och där har vi nu frammatningen av tecken på skärmen. Eftersom vi ska ha 40 tecken på varje rad och "radlängden" är 40 µs så måste ju varje tecken uppta 1 µs.

1 MHz-signalen i fig 6.13 körs in på en 64-räknare. Den räknar fram de 64 mikrosekunder som enligt 6.14 utgör totala "linjelängden" (återgången inräknad). Här får vi alltså en utsignal på ca 16 kHz och den kan vi använda för att mata fram nya linjer på skärmen.

Bilden ska innehålla 312 linjer (inklusive linjerna som försvinner under återgången) och därmed är efterföljande räknare en 312-räknare. Utgången här ger oss 50 Hz vilket just är bildväxlingsfrekvensen.

Ytterligare en 16-räknare ger utfrekvensen ca 3 Hz som är lagom takt för "blinket" hos bildskärmens markör (cursor).

2.3 Bildadress och minnesadress

Vi kan beskriva var ett tecken är placerat på bildskärmen - dvs bildadressen - med hjälp av radnummer och positionsnummer (det motsvarar y- och x-koordinater i ett vanligt koordinatsystem). Po-

sition (2, 20) på ABC-skärmen anger 3:e raden uppifrån och 21:a positionen från vänster på denna rad.

Bildadressen genereras i ABC80 med hjälp av utgångarna på 64-räknaren (som räknar μs) och 312-räknaren (som räknar linjer). Fig 6.15 visar principen. Man använder här tre PROM. Det vänstra adresseras från 64-räknaren och har fyra utgångar. Den vänstra PROM-utgången styr en signalgrind som släpper fram 1 MHz-signalen till en 40-räknare. En utgång från samma PROM nollställer 40-räknaren när linjen är slut. Ytterligare en utgång ger horisontalsynk till videomonitorn. Med ett PROM kan alla dessa utgångssignaler programmeras godtyckligt inom intervallet Ø-63 μs. Man har alltså möjlighet att starta linjen och avge synk så att man får lämplig vänstermarginal på bildskärmen. Utgången DH (fördröjd horisontalsynk) används för att släcka strålen utanför den önskade bildytan. Kretsdetaljerna för detta har utelämnats i fig 6.15.

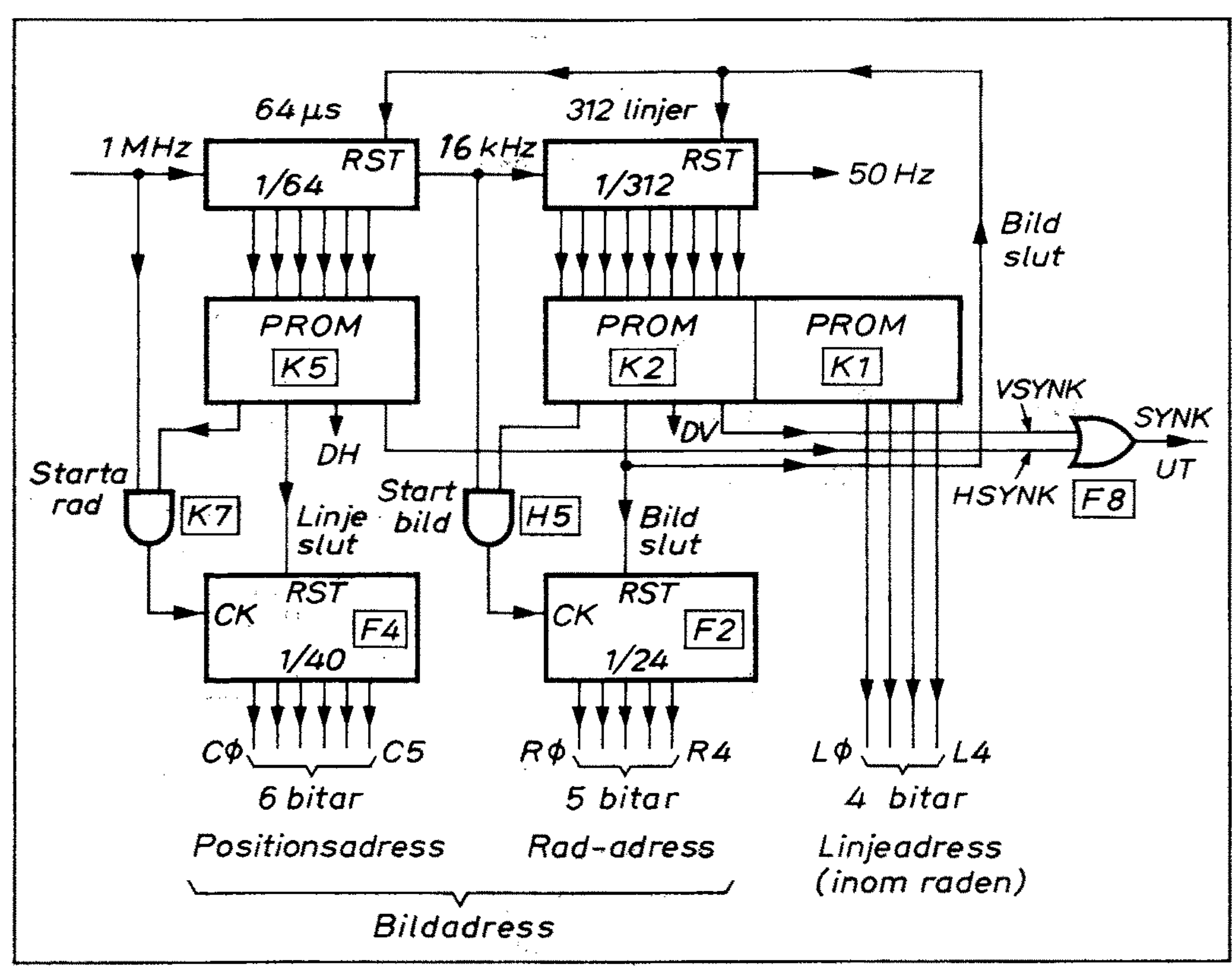


Fig 6.15 Generering av bildadressen (i princip)

312-räknaren adresserar två PROM och här har man åtta godtyckligt programmerbara utgångssignaler tillgängliga. De första fyra används för att starta och nollställa radräknaren (24-räknaren).

RESET-signalen kommer här först när bilden är slut (rad 311) och kan därför användas för nollställning av såväl 64- som 312-räknaren. Vänstra PROM:et under 312-räknaren ger dessutom vertikalsynk. Denna vertikalsynk körs tillsammans med horisontalsynken genom en OR-grind som avger utgående synksignal.

Det högra PROM:et under 312-räknaren adresseras på samma sätt som det vänstra. Utgångarna på detta PROM är programmerade att avge fyra bitars linjeadress. Varje rad är ju som vi sett uppdelad på 10 linjer och därför krävs 4 bitar.

Nu kommer problemet! Fig 6.16. Varje bildadress måste på ett entydigt och enkelt sätt motsvara en viss minnesadress i bildminnet. Om det vore så enkelt att vi hade 32 rader med 32 tecken på varje rad skulle vi direkt kunna använda bildadressen som minnesadress (inom det givna 1K-blocket). Hobbydatorer har av denna anledning ofta videointerface som ger 16 eller 32 rader med 32 eller 64 tecken per rad.

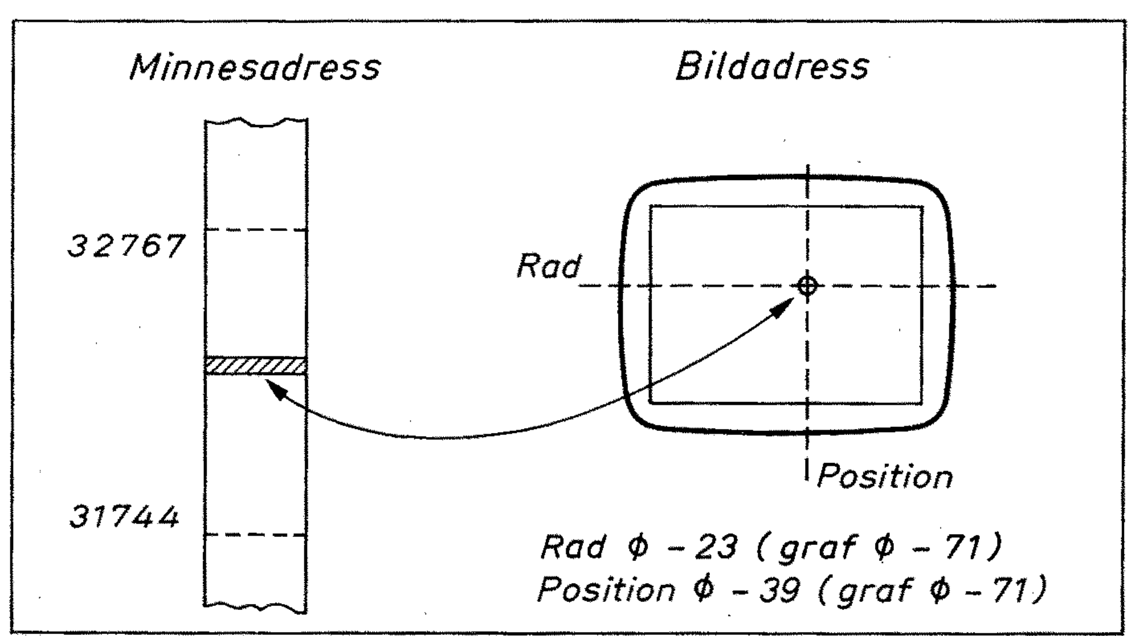


Fig 6.16 Minnesadress och bildadress

Om vi direkt skulle använda 5 bitar radadress och 6 bitar positionsadress som minnesadress skulle det krävas ett bildminne på 2K byte - och hälften av detta skulle vi inte utnyttja! Minnen är billiga så det är inte fråga om kostnad - men vi vill kunna använda hela minnesarean (64K) för nyttiga ändamål. Därför har man i ABC80 gjort en omkodning av den 11 bitar stora bildadressen till en 10 bitars minnesadress. Denna omkodning är enklast att förstå om vi delar in bildskärmen i 15 numrerade rutor enligt fig 6.17a. Varje ruta består av 8 rader med 8 positioner.

Positionsadressen i fig 6.17b är nu uppdelad på 3 bitar (dvs 8 positioner) inom varje ruta och 3 bitar rutnummer (ɸ-8 varav ɸ-4 ut-

nyttjas). Radadressen till höger i fig 6.17b är på motsvarande sätt uppdelad i rader inom rutan (3 bitar) och rutnummer (2 bitar). Minst signifikanta bitarna i fig 6.17b är ritade till vänster för att passa ihop med utgångarna på räknarna i fig 6.15.

Räknar vi nu fram bildadressen steg för steg så kommer vi tydligen att avsöka bildskärmen på normalt sätt linje för linje. Positionsadressens rutnummer räknas hela tiden upp från Ø-4 och radadressens rutnummer från Ø-2.

Genom att addera positionsadressens rutnummer med 5 när radadressens rutnummer blir 1 kommer minnesadressens rutnummer att bli 5. När radadressens rutnummer blir 2 adderas på motsvarande sätt talet 10 till positionsadressens rutnummer. Minnesadressen kommer på detta sätt att bestå av positionsadressen AØ-A2 inom aktuell ruta, rutadressen A3-A6 (dvs ruta Ø-15) samt linjeadressen A7-A9 inom aktuell ruta. Med den enkla adderingen har vi alltså placerat ut rutorna i rad i minnesarean.

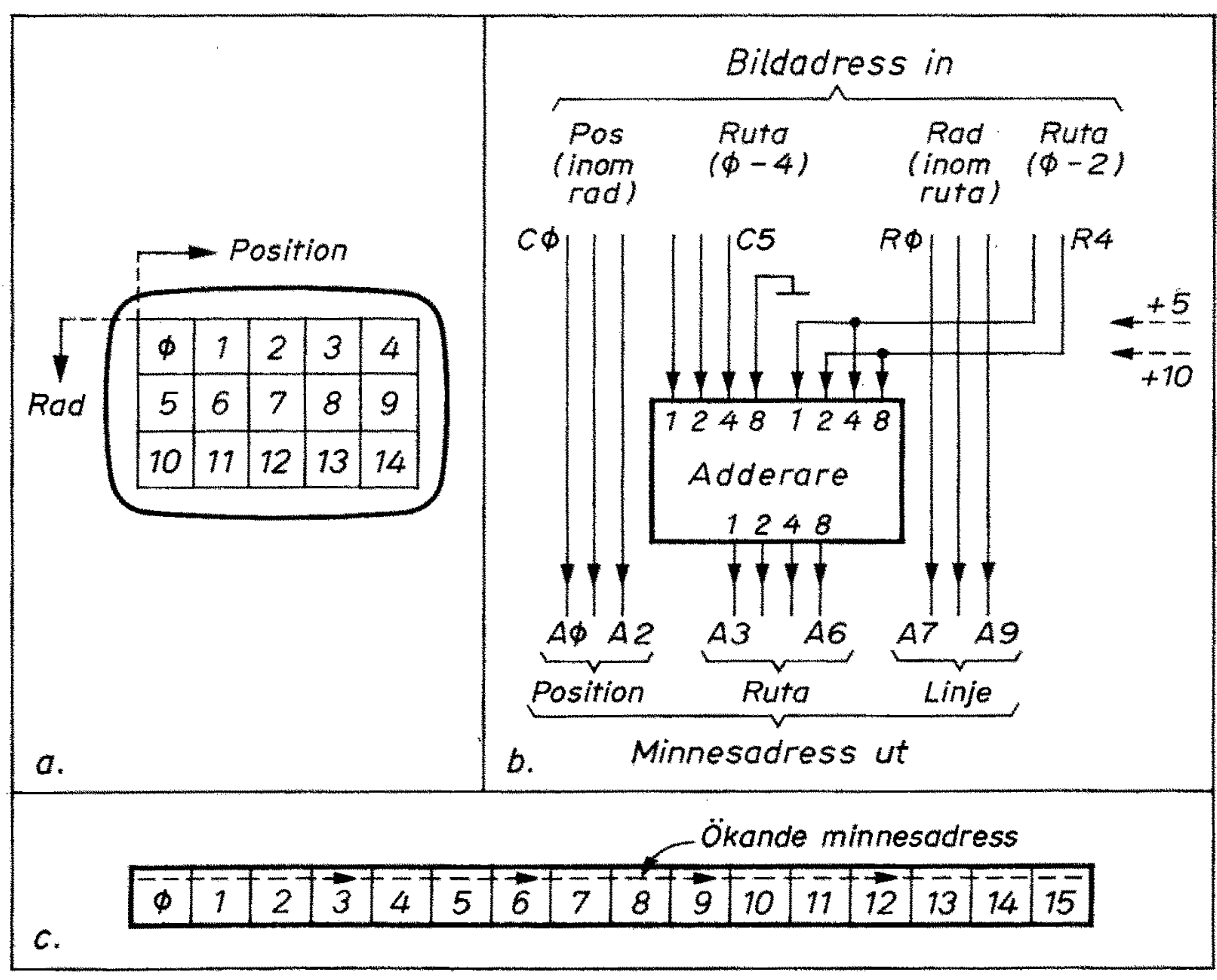


Fig 6.17 Omkodning av bildadress till minnesadress

Stegar vi bildminnet från begynnelseadressen kommer vi att söka av bilden på det sätt som fig 6.17 visar. Nu får bildskärmens innehåll rum på minnesarean 1K! Endast ruta nr 15 är outnyttjat utrymme. Det krävs givetvis motsvarande rutiner i programvaran för att man enkelt ska kunna placera ut tecken radvis på bildskärmen.

Fig 6.18 anger radnummer på bildskärmen (Ø-23) och motsvarande decimala adresser i bildminnet.

Rad	Decimal adress	Outnyttjat minne:
Ø	31744 - 31783	
1	31872 - 31911	
2	32000 - 32039	
3	32128 - 32167	
4	32256 - 32295	
5	32384 - 32423	
6	32512 - 32551	
7	32640 - 32679	
8	31784 - 31823	
9	31912 - 31951	
10	32040 - 32079	
11	32168 - 32207	
12	32296 - 32335	
13	32424 - 32463	
14	32552 - 32591	
15	32680 - 32719	
16	31824 - 31863	31864 - 31871
17	31952 - 31991	31992 - 31999
18	32080 - 32119	32120 - 32127
19	32208 - 32247	32248 - 32255
20	32336 - 32375	32376 - 32383
21	32464 - 32503	32504 - 32511
22	32592 - 32631	32632 - 32639
23	32720 - 32759	32760 - 32767

Fig 6.18 Radernas adresser i bildminnet

2.4 Teckengenerering

Fig 6.19 visar principen för teckengenerering på bildskärmen. Räknarna ger synkpulser så att strålen på bildskärmen går synkront med framräkningen av positions- och linjeadresser.

Minnesadresserna pekar synkront ut motsvarande teckenpositioner i bildminnet.

En rad på skärmen upptar som vi sett i fig 6.18, 40 byte i bildminnet. Eftersom en rad består av 10 linjer kommer dessa 40 byte att läsas 10 ggr. För varje ny läsning ökas linjeadressen ett steg.

I bildminnet har CPU:n placerat ASCII-koden för de tecken vi tidigare skrivit in via tangentbordet (eller som matats fram av ett program). Som framgår av fig 6.19 matas ASCII-koden från bildminnet (7 bitar) och linjeadressen från räknaren (4 bitar) in på adressledningarna till ett ROM. Detta ROM har 5 bitars ordlängd och innehåller 10.240 bitar.

Avläsningen av bildminnet sker med hastigheten 1 MHz. Varje μs får vi alltså ut ett fembitars ord från ROM:et. Detta ord laddas parallellt i ett 6 bitars skiftregister. (Den sjätte biten i detta register laddas ständigt med en nolla).

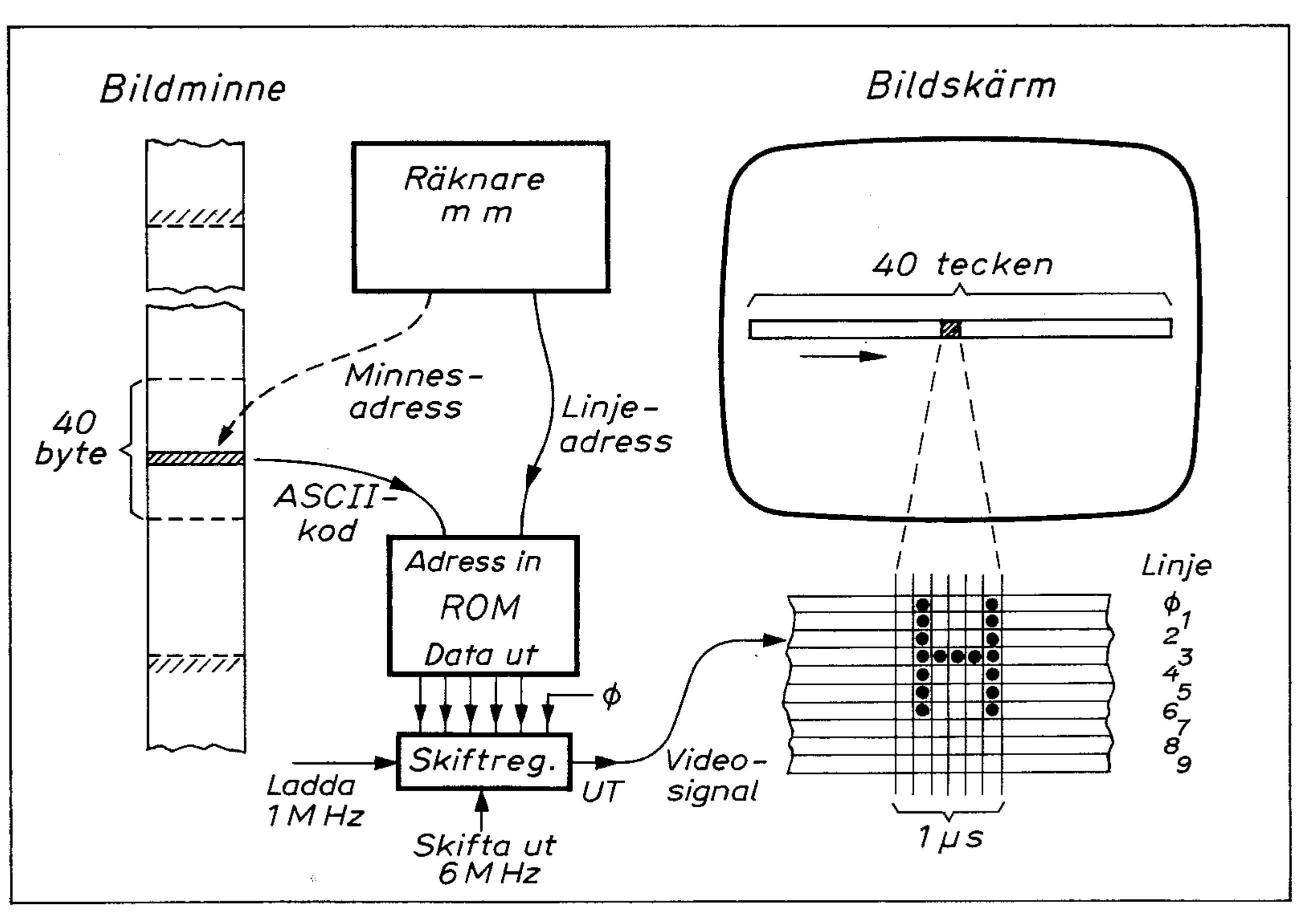


Fig 6.19 Principen för generering av ett tecken på bildskärmen

Genom att skifta ut innehållet ur detta register med skiftfrekvensen 6 MHz placerar vi ut 6 bildpunkter för varje tecken på den aktuella linjen. Den vänstra bildpunkten är alltid svart och utgör mellanrummet i sidled mellan två närbelägna tecken på bildskärmen.

Ofta kallar man ROM:et i fig 6.19 för teckengenerator trots att det endast utgör en av de många kretsar som fordras för teckengenerering.

Vi har nu fått förklaringen på 6 MHz-frekvensen i räknarkedjan. Den klockar alltså ut videosignalen från skiftregistret.

När man tittar på bildskärmen anar man inte vilken ofantlig karusell som pågår i bildminnet och räknekretsarna. För varje bild (och vi får 50 bilder i sekunden) avläses bildminnets samtliga 1024 adresser 10 ggr. Detta pågår ständigt och man kan fråga sig hur CPU:n ska kunna adressera bildminnet utan att stöta ihop med videointerfacets ständigt pågående avläsningar i bildminnet. Vi ska återkomma till detta problem senare.

2.5 Tecken, graf och markör

Överst i fig 6.20 återser vi teckengeneratorn från fig 6.19. Den matas med linjeadress och ASCII-kod och laddar ett skiftregister med 5 bitar. Dessa bitar (samt en första bit som alltid är noll) skiftas ut i serie och bildar en videosignal för generering av tecken på bildskärmen.

ABC80 kan även generera grafiska bilder och det sker i princip på samma sätt som teckengenereringen. Vi ser grafgeneratorn nedanför teckengeneratorn i fig 6.20 och till höger därom grafgeneratorns skiftregister. Med tre grindar (position H5 och H4) kan vi koppla om mellan teckensignal och grafsignal.

Omkopplingen mellan tecken och graf styrs av två ASCII-koder som avkänns av det nedre PROM:et i fig 6.20. Inkommer ASCII-koden 135 ettställs JK-vippan (position J5) och dess utsignal lagras i en latch (position J4). Latchen styr den ovan omtalade omkopplaren så att det övre skiftregistret (teckengeneratorn) inkopplas till videoutgången.

Med ASCII-koden 151 får man på motsvarande sätt grafgeneratorn inkopplad i stället för teckengeneratorn.

Latchen i fig 6.20 håller tre viktiga styrsignaler. En är den ovan beskrivna tecken/graf-omkopplingen. De två andra styrsignalerna kallas MARKÖR (CURSOR) och BLANK. Observera att versaler avkodas av nedre PROM:et (J3) och ger latchen teckensignal även om vippan (J5) står i grafmod.

ASCII-koden upptar 7 bitar och den åttonde (dvs bit nr 7) anger om teckenpositionen på bildskärmen ska "markeras". Man använder

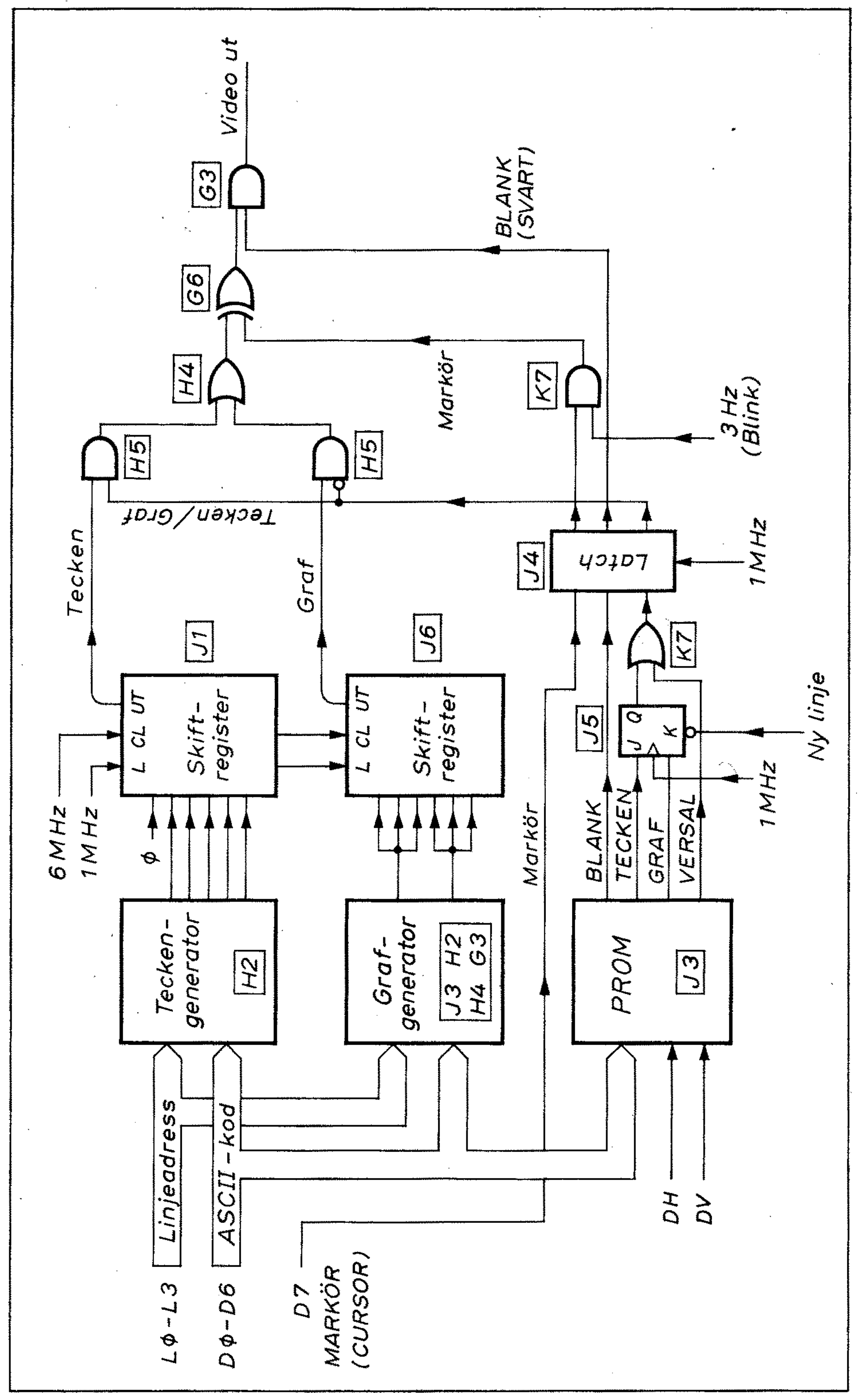


Fig 6.20 Principen för generering av tecken, grafer och markör i ABC80

normalt en markör för att visa var nästa tecken (som man anslår på tangentbordet) ska hamna på bildskärmen.

En etta i bit 7 i bildminnet anger alltså att motsvarande tecken ska markeras på bildskärmen. Markörsignalen lagras i latchen i fig 6.20 och sänds vidare via en AND-grind (position K7) till en XOR-grind (position G6) som passeras av videosignalen. Vid markörsignal kommer AND-grinden att släppa fram 3 Hz (blinkfrekvensen) från slutet av den långa kedjan av räknare. Blinkfrekvensen kommer genom XOR-grinden att omväxlande fasvända videosignalen - så att svart blir vitt och vitt blir svart.

Markören förstör ingen information på bildskärmen. Skriver man in bokstaven A i bildminnet får man ett lysande A på svart botten, skriver man in A med markör får man omväxlande ett svart A i en lysande ruta (som omfattar 10 linjer och 6 punktkolumner) och ett vitt A i en svart ruta.

Den tredje styrsignalen, BLANK, leds vidare till en OCH-grind som videosignalen måste passera. Med BLANK-signalen kan alltså videosignalen stoppas och detta ger svart ruta (blank) på bildskärmen.

Nedre PROM:et i fig 6.20 avkodar kontrolltecken (vänstra kolumnen i fig 3.19), mellanslag och marginalerna på bildskärmen. Det senare sker med hjälp av signalerna DH och DV (= fördröjd horisontal- och vertikalsynk) från räknarkedjan. BLANK-signalen kommer därigenom att släcka bilden dels utanför det avsedda bildutrymmet och dels när ASCII-koden för mellanslag eller kontrolltecken avkodas.

2.6 Grafiken i ABC80

ABC80:s bildskärm rymmer som vi sett ett rutnät med 6x40 = 240 punktpositioner och 24x10 = 240 punktlinjer. För att kunna rita detaljrika bilder på bildskärmen skulle vi kanske önska oss att kunna tända och släcka dessa 57600 punkter på godtyckligt sätt. Varje teckenplats på skärmen rymmer 60 punkter och dessa punkter kan kombineras på $2^{60} \approx 10^{18}$ olika sätt. Bildminnet skulle behöva ha ordlängden 60 bitar (istället för 8) för att klara detta krav. Man kan givetvis tänka sig andra lösningar, exempelvis att varje teckenplats inkodas i 8 byte - dvs att vi utökar bildminnet till 8K byte. Eftersom vi inte kan öka klockfrekvensen nämnvärt skulle det därvid ta 8 ggr längre tid att teckna en komplett bild på bildskärmen.

Av praktiska skäl måste vi tydligen kompromissa med den önskade detaljrikedomen i grafiken. En rimlig kompromiss är att behålla bildminnet som det är och använda 7 bitar för de viktigaste punkt-

kombinationerna inom en teckenplats. Vi väljer förstås de kombinationer som har standardiserats för användning inom det europeiska text-TV-systemet. Dessa kombinationer framgår av fig 6.21.

2.7 Teckenmod och grafmod

En byte i bildminnet kan tydligen tolkas på två sätt:

- o som ett tecken (dvs bokstav, siffra eller specialtecken)
- o som ett grafiskt bildelement

Vi har redan sett hur elektroniken i ABC80 utför denna tolkning. När ABC80 startas ställs JK-vippan i fig 6.20 i läge "tecken". Därefter arbetar alltså bildskärmen på normalt sätt, dvs tolkar bildminnets innehåll som ASCII-kod och skriver ut motsvarande tecken på bildskärmen.

Om vi skriver in koden 151 i en byte i bildminnet kommer funktionen att ändras radikalt. Låt oss följa förloppet.

Vi antar att radräknaren just hoppat över till ny rad och stegar fram längs linje Ø på denna rad. Nu läser PROM:et plötsligt koden 151 som vi placerat ett stycke in på denna rad. Vad händer?

Omedelbart reagerar PROM:et i fig 6.20 på grafkoden och avger en grafsignal till JK-vippan. Denna nollställs och kopplar därmed grafgeneratorn till videoutgången. Efterföljande byte i bildminnet tolkas nu som grafer. Detta fortsätter så länge JK-vippan står kvar i läge "graf". Vad är det då som får JK-vippan att återgå till normalläget, dvs läge "tecken"? Fig 6.20 ger svaret. JK-vippan kan nollställas på två sätt.

Vid varje linjeväxling nollställs JK-vippan automatiskt genom signal på RESET-ingången. Avsökningen av efterföljande linje börjar alltså med teckenmod, men så fort avsökningen når koden 151 ställs vippan åter i grafmod. Så fortsätter avsökningen genom radens samtliga tio linjer. Grafmoden ger alltså grafik på resten av den rad där kommandot placerats men på efterföljande rad återgår bildskärmen automatiskt till teckenmod. Vill vi ställa hela bildskärmen i grafisk mod skriver vi in grafkommandot längst till vänster på varje rad. Första kolumnen blir därmed svart (innehåller det osynliga grafkommandot) och skärmens utrymme i grafisk mod är därmed begränsat till 39x24 rutor (jämfört med 40x24 för teckenmod).

Vi vill förstås kunna använda den grafiska moden inom en begränsad del av en rad. Det kan vi göra genom att både använda grafkommando och teckenkommando på samma rad. Fig 6.22 visar ett exempel. Här har vi satt in ASCII-koden för sekvensen abcdE på tre ställen på samma rad. Mellan första och andra sekvensen har vi satt in ett graf-kommando (151) och mellan andra och tredje sekvensen har vi satt in ett teckenkommando (135). Båda dessa kom-

Kod	T	G	Kod	T	G	Kod	T	G	Kod	T	G
32	Blank		56	8		80	P	P	104	h	
33	!		57	9		81	Q	Q	105	i	
34	”		58	:		82	R	R	106	j	
35	#		59	;		83	S	S	107	k	
36	¤		60	<		84	T	T	108	l	
37	%		61	=		85	U	U	109	m	
38	&		62	>		86	V	V	110	n	
39	’		63	?		87	W	W	111	o	
40	(		64	É	É	88	X	X	112	p	
41	)		65	A	A	89	Y	Y	113	q	
42	*		66	B	B	90	Z	Z	114	r	
43	+		67	C	C	91	Ä	Ä	115	s	
44	,		68	D	D	92	Ö	Ö	116	t	
45	-		69	E	E	93	Å	Å	117	u	
46	.		70	F	F	94	Ü	Ü	118	v	
47	/		71	G	G	95	–	–	119	w	
48	0		72	H	H	96	é		120	x	
49	1		73	I	I	97	a		121	y	
50	2		74	J	J	98	b		122	z	
51	3		75	K	K	99	c		123	ä	
52	4		76	L	L	100	d		124	ö	
53	5		77	M	M	101	e		125	å	
54	6		78	N	N	102	f		126	ü	
55	7		79	O	O	103	g		127	■	

Fig 6.21 Koder tolkade i teckenmod (T) och grafmod (G)

mandon ger svart ruta på bildskärmen. Nu ser vi funktionen klart! Vänstra delen av raden är automatiskt ställd i teckenmod av föregående radväxling. Efter grafkommandot ger ASCII-koderna grafer. Det efterföljande teckenkommandot återställer resten av raden till teckenmod.

Fig 6.22 Funktion hos graf- och teckenkommando

Hur gick det då med versalen E? Som vi sett både i tabellen i fig 6.21 och i schemat i fig 6.20 ger versaler automatiskt tecken-mod. Det är givetvis praktiskt att kunna blanda versaler och grafer utan att behöva ställa om skärmen med tecken- och graf-kommandon.

2.8 Specialkoder

Vi har redan mött två koder som inte ger någon bild på bildskärmen men som trots detta har stor betydelse för ABC80:s funktion. Dessa tecken kan inte heller genereras direkt (med ett enda tangentnedtryck) från tangentbordet eftersom det inte finns tangenter som ger de aktuella koderna. Nedanstående figur ger några av de specialkoder som utnyttjas i ABC80-systemet.

kod (decimal)	funktion
7	bell (ger ljudsignal)
10	radframmatning (LF = line feed)
12	blank skärm (FF = form feed)
13	vagnretur (CR = carriage return)
135	teckenmod (startar teckenmod på en rad)
151	grafmod (startar grafmod på en rad)

Specialkoderna 135 och 151 avkodas som vi sett av ABC80:s hårdvara medan 7, 10, 12 och 13 avkodas och effektueras av programvaran.

2.9 Bildminnet och CPU:n

Vi har nu sett hur bildminnet ständigt avläses av videointerfacet. Bildminnet laddas med ny information av CPU:n. För att inte störa CPU:ns adressbuss när videointerfacet gör sina ständigt pågående avläsningar i bildminnet styr man bildminnets adressingångar via en MUX som normalt ligger i läge "video". Minnesadresser från videointerfacet matas då in på bildminnet och detta svarar med att lägga ut sökta data på dataledningar till videointerfacet. Fig 6.23.

För att inte störa CPU:ns databuss har man lagt in en dubbelriktad drivkrets som kan läggas i tristate mellan databussen och bildminnets dataledningar.

I de flesta mikrodatorsystem med videointerface gör man avbrott i bildavsökningen när CPU:n behöver access till bildminnet. Detta kan i vissa fall bli ganska irriterande.

För att inte behöva avbryta bildavsökningen när CPU:n gör access till bildminnet har man infört en rad elektroniska finesser i ABC80. Vi ska inte här ge oss in i alla kretsdetaljerna utan enbart antyda principerna.

För videointerfacets avläsning av bildminnet använder man "pipelining". Det innebär att videointerfacet ständigt ligger en byte före i tid med adressering och avläsning av bildminnet. Med hjälp av en latch (position H1) i dataledningarna fördröjer man sedan data så att de anländer i rätt tid till bildskärmen.

Om nu CPU:n vill läsa eller skriva i bildminnet så måste den dela den tillgängliga tiden för minnesaccess med videointerfacet. Genom att snabbt mellanlagra data för bildskärmen i en latch (position G1 och G2) och samtidigt fördröja $\overline{\text{MREQ}}$-signalen något kan man skilja på videoaccess och CPU:access.

När bildskärmens data har snabblagrats i vänstra latchen kopplas MUX:en över till adressbussen. CPU:n kan nu göra sin läsning eller skrivning, något försenad men väl inom tidsmarginalen för minnesaccess.

Det krävs en knepig styrkrets för att förskjuta tiderna utan att störa trafiken. Den antyds med ett streckat block i fig 6.24 och består bl a av två vippor och två grindar.

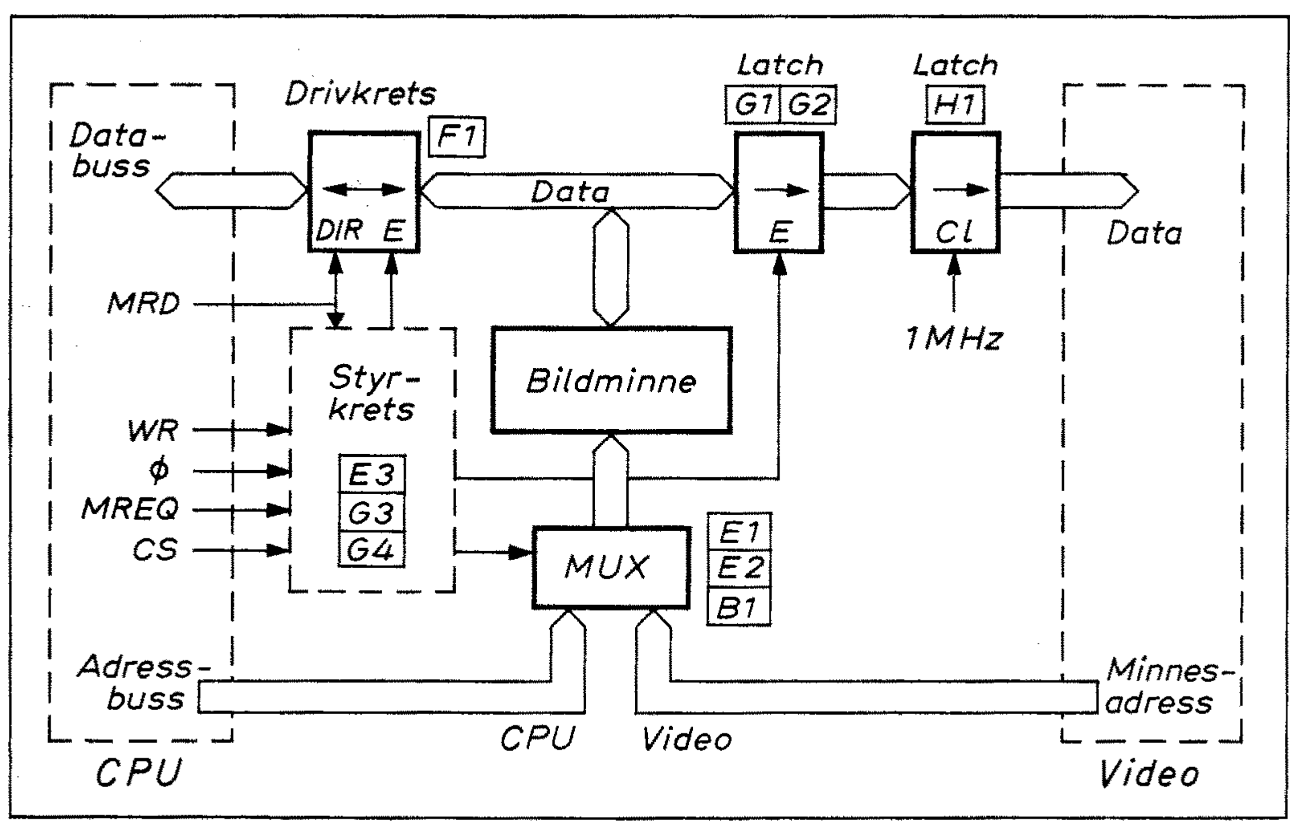


Fig 6.23 Några av kretsarna för delningen av bildminnet

5
64 µs
Linjesynk (H-synk)
En linje
240 bildpunkter
5 12 40 µs
Linjesynk (H-synk)
Bildsynk (V-synk)
1
20 ms
Synk-
signal
En bild
Bild-
signal
240 linjer
1 2 15 ms

Fig 6.24 Synk- och bildsignal i ABC80

2.10 Videosignalen

Videointerfacet avger två signaler, en synksignal och en bildsignal. Fig 6.24 visar ungefärliga tider för dessa signalers inbördes lägen.

Sammanfattning

Videointerfacet ger exempel på en rad intressanta kretslösningar. Avsikten med detta avsnitt har varit att ge en antydan om funktionen utan att därför gå in på alla kretsdetaljer. Principen för ett videointerface är nyttig att känna till, bl a för att bättre förstå hur interfacet hanterar olika koder och fungerar i olika arbetsmoder. Sambandet mellan bildadress och minnesadress kan verka förbryllande om man inte fått en inblick i hur videointerfacet hanterar adresser för att spara minnesutrymme.

Givetvis kommer man snart att kunna köpa större delen av de funktioner vi här beskrivit förpackade i en enda LSI-krets (LSI = large scale integration). När den tiden kommer (och den är delvis redan här) kan det ju vara bra att ha en viss grundförståelse för vad som döljer sig i dessa LSI-kretsars inre!

3. Kassettinterfacet

Den vanliga kassetten (Philips Compact Cassette) har blivit internationell standard inom ljudåtergivningens lågprisområde. Kassetter tillverkas i mycket stora serier och får därmed lågt pris. Man har länge använt samma typ av kassetter för registrering av digital information. ABC80 har ett uttag för kassettbandspelare och ett inbyggt interface som dels kan fjärrstyra kassettbandspelarens motor och dels hantera signaler för in- och avspelning.

Vi ska nedan kort beskriva några vanliga inspelningsmetoder och därefter visa de enkla kretsar som ingår i ABC80:s kassettinterface.

3.1 Inspelningsmetoder

Varken ljudbandspelare eller vanliga ljudkassetter är tillverkade för inspelning av digital information. Vi måste vara väl medvetna om detta faktum när vi nu ska använda dessa lågprisprylar för dataanvändning.

Kassettbandspelare är avsedda för musik och det ligger därför nära tillhands att låta två toner (sinussignaler med olika frekvens) representera de binära siffrorna Ø och 1. Vill vi sedan vara säkra

på att inspelningen blir korrekt måste vi låta tonerna ligga kvar så länge att eventuella störningar eller avbrott i oxidbeläggningen på tonbandet inte inverkar. Och därmed blir inspelningen så långsam att vi får sitta och vänta även vid avspelning av korta programsnuttar.

En metod att använda en ljudbandspelare är att mata den med fyrkantvåg. Man kan då låta varje period av fyrkantvågen innehålla koden för en bit. Vi får därmed snabb in- och avspelning. Ett enda dammkorn, en enda störspik i en kritisk tidpunkt eller ett aldrig så litet avbrott i oxidbeläggningen på bandet ger i detta fall fel vid in- eller avspelning (eller bådadera).

Problemet med vanliga kassettband för datalagring är tydligen att finna den lämpliga kompromissen mellan tillförlitlighet och snabbhet.

3.1.1 Kansas City

1975 samlades en skara hobbydatorfolk i Kansas City (Missouri, USA) och beslöt sig för en "hobbystandard" som fick namnet Kansas City (eller kortare KC). Tanken var att man skulle kunna använda enkla bandspelare och man beslöt sig därför för ton-metoden. För att enkelt kunna generera och räkna perioderna i signalen beslöt man sig för följande kod:

Logisk nolla:	4 perioder	1200 Hz
Logisk etta:	8 perioder	2400 Hz

KC-metoden överför informationen asynkront, dvs varje byte har eget start- och stopptecken och behöver därför inte ligga synkront med någon klocka. KC-standard har startbit och två stoppbitar på samma sätt som signalerna från en vanlig teletype. Fig 6.25.

Överföringshastigheten med KC blir låg, 300 bit/s, dvs bara 3 gånger snabbare än remsläsaren hos en vanlig mekanisk TTY (= teletype).

Kansas City (KC-standard) har fått mycket stor spridning både bland hobbyfolket och för lågprisdatorer.

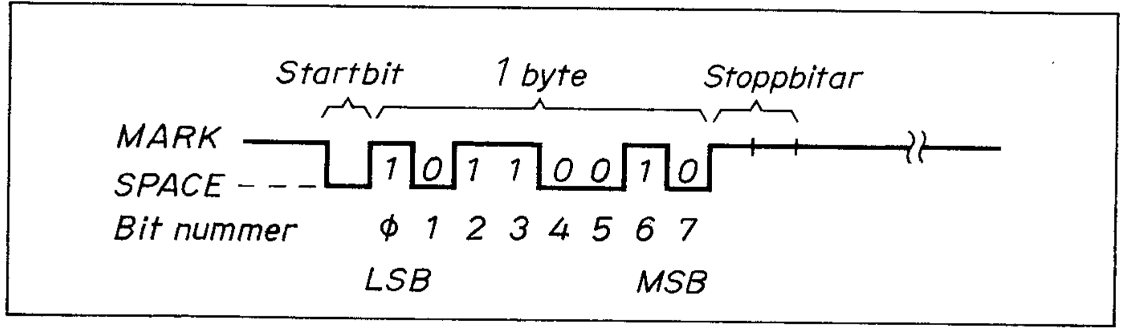


Fig 6.25 Asynkron överföring av en byte med teletype

3.1.2 Tarbell

Mr Tarbell driver ett av de många småföretag i USA som säljer kretskort till S100-bussen (Imsai, Altair m fl hobbydatorer). Han har givit namn till en inspelningsmetod inom hobbydatorområdet som mycket liknar den professionella PE-metoden (PE = phase encoding) enligt ANSI standard biphase encoding. Tarbell använder fyrkantvåg och kör normalt med 1496 bit/s, dvs fem gånger snabbare än KC.

I fig 6.26 jämförs Tarbells metod med Kansas City och ytterligare en metod som kallas FM. Den senare ska vi strax återkomma till.

Enligt PE-metoden låter man signalen ligga hög under halva periodtiden. En nolla ger hög signal under andra halvan medan en etta ger hög signal under första halvan av tiden för en bit.

PE arbetar synkront och självklockande. Med synkront menas att ett stort antal byte körs ut synkront med en klocka. Man slipper därigenom använda start- och stoppbitar för att avgränsa enskilda tecken. Med självklockande menas att man inte behöver en extra ledning (eller kanal) för synkroniseringspulser (vilket var fallet i flera äldre system). Med hjälp av lämpliga kretsar kan klocksignalen återbildas ur PE-signalen.

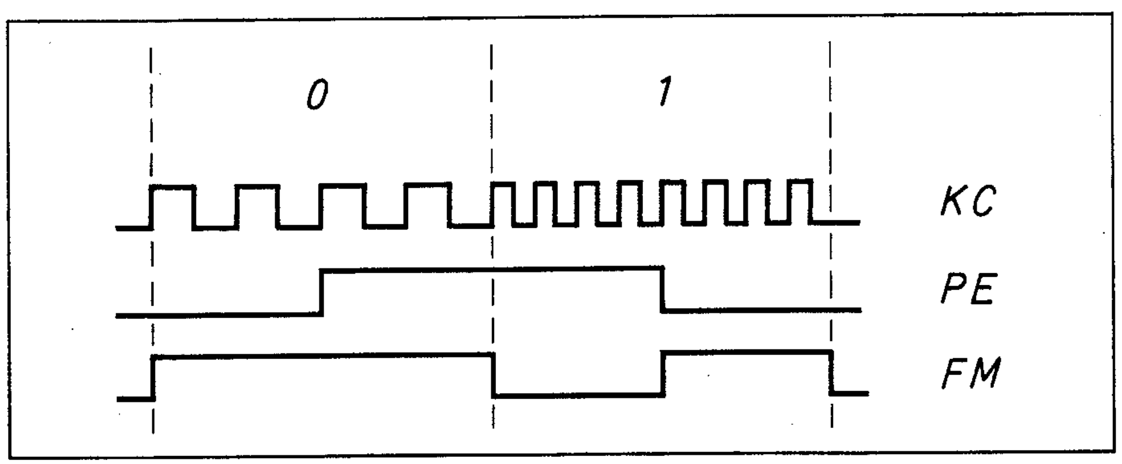


Fig 6.26 Nolla och etta enligt
KC = Kansas City
PE = fasmodulen
FM = frekvensmodulring.

Givetvis ställer PE-metoden stora krav på bandspelare och tonband om man ska få säker överföring. Vid in- eller avspelning används olika typer av "protokoll". Man startar exempelvis med en rad nollor som bygger upp synksignalen. Därefter kommer ett kontrollord och kanske uppgifter om inspelningens längd eller var den ska pla-

ceras i minnet. För att kontrollera om fel uppstått vid in- eller avspelning används en rad olika metoder. När datamängden passerar räknas checksumma (eller CRC-summa) för varje byte (CRC = cyclic redundancy code). Checksumman eller CRC-koden placeras omedelbart efter datablocket vid inspelningen.

Vid avspelning kollas checksumman (eller CRC-koden) av datorn och man får felutskrift om avspelningen ger en avvikande checksumma jämfört med den tidigare inspelade.

3.1.3 Frekvensmodulering

Frekvensmodulering är snarlik PE. Den vanligaste benämningen, FM (eller MFM = modified frequency modulation) är oegentlig. Den syftar förmodligen på att inspelade ettor ger dubbelt så hög frekvens som inspelade nollor.

Tanken bakom FM (liksom även bakom PE) är att man ska köra med fyrkantvåg och avläsa flankerna (eller nollgenomgångarna). I FM växlar man signalnivån i takt med klockan (pulstiden T). Vill man ha en etta inspelad växlar man dessutom signalnivån vid halva pulstiden (T/2). Om dessa "halvtidssprång" saknas tolkas innehållet som nollor. Fig 6.27. FM är liksom PE självklockande.

Inspelning med PE och FM görs normalt med ren fyrkantvåg (direkt från en TTL-grind). Vid avspelning får man givetvis inte tillbaks samma fyrkantformade signal som man tidigare spelat in. Det bästa är att göra inspelningen direkt på inspelningshuvudet och ha minsta möjliga antal förstärkarsteg inkopplade vid avspelning. Att låta den avspelade signalen passera en komplett lågfrekvensförstärkare kan ofta ge sådana fasfel att man tappar informationen.

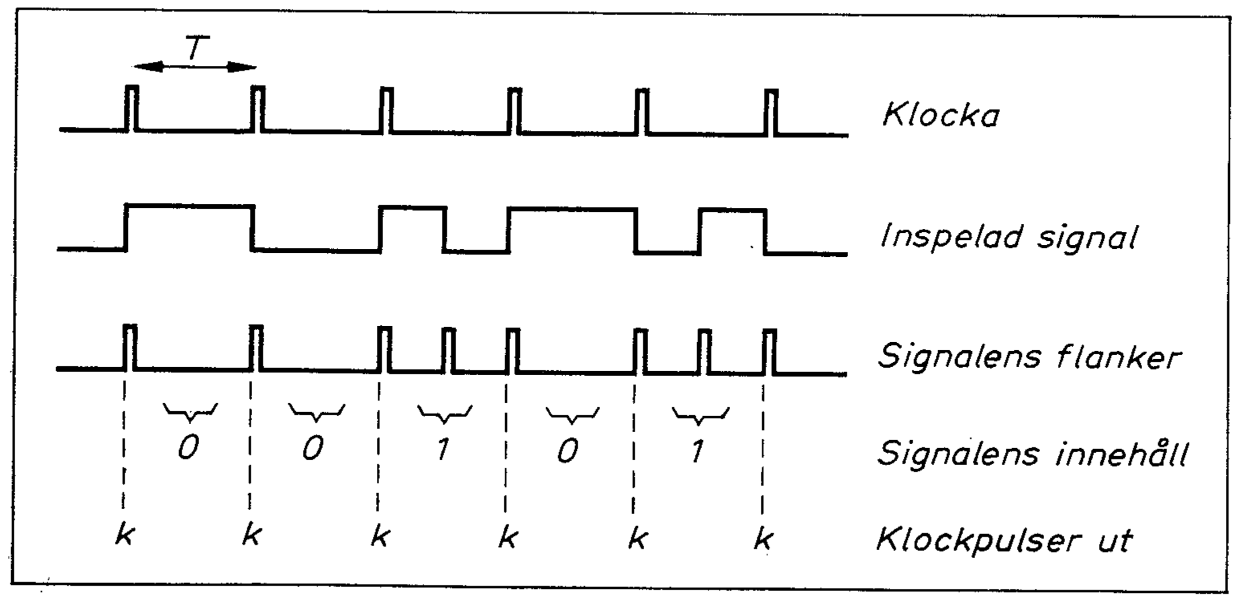


Fig 6.27 Principen för FM (frekvensmodulering)

Nåväl, även med en bandspelare av god kvalitet blir utsignalen avsevärt deformerad. Fig 6.28. Om bara flankerna (eller nollgenomgångarna) behåller sin inbördes tidsplacering kan man emellertid återskapa såväl klocksignal som inspelad information.

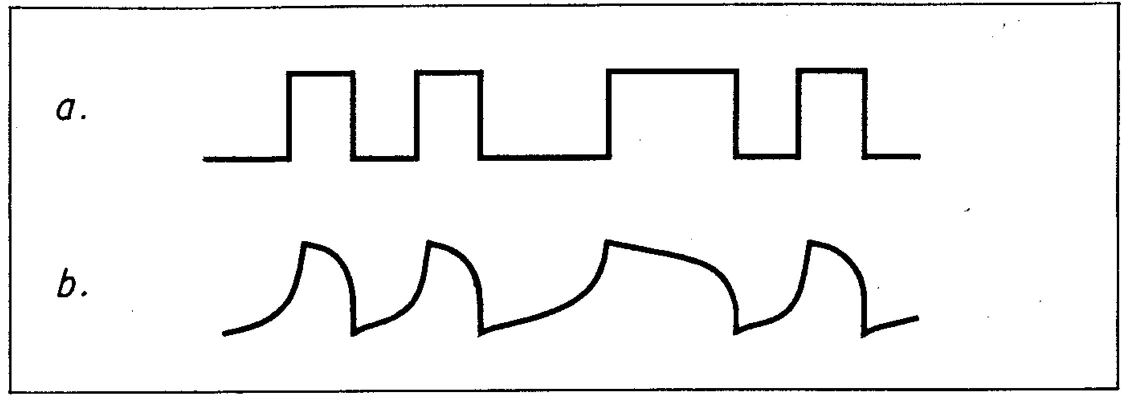


Fig 6.28 a. Inspelad signal
b. Signal vid avspelning

3.2 Kretsarna i kassettinterfacet

Fig 6.29 visar kretsarna i ABC80:s kassettinterface. De är anslutna till bitarna 5, 6 och 7 i port B på PIO:n.

3.2.1 Motorstyrningen

Om bit 5 programmeras som utport kommer en etta på utgången att dra relät (nere till höger i fig 6.29) via inverteraren i position H7. Eftersom det kan ligga en stor kapacitiv last över reläkontakterna har man lagt in en liten induktor i serie. Detta begränsar strömmen så att inte kontakterna i relät "svetsas" ihop av en otillåtet stor strömspik.

Över relälindningen ligger som tidigare nämnts en diod som begränsar spänningstransienten vid frånslag.

3.2.2 Inspelning

En dator arbetar parallellt med ett flertal bitar samtidigt. När vi ska spela in dessa bitar på ett band måste de placeras i serie (om vi inte har tillgång till en mångkanalbandspelare).

Det finns två i princip olika sätt att överföra information från parallell- till serieform, fig 6.30. Antingen använder man kretsar (av typ skiftregister) eller också skriver man ett program som låter ackumulatorns innehåll skiftas ut bit för bit på en önskad utport.

J8 339
5 V
5,6 k
10nF
IN
-0,5 V
oc
10k
+0,5 V
Deriverande krets
Puls-vändare
G5 7474
S
R
Q̄
Latch
Databuss
5 V
220
G6
H7
1µF
UT
B
PIO
A
5 V
H7
Motor-styrning
Relä
Adressbuss

Fig 6.29 ABC80:s kassettinterface

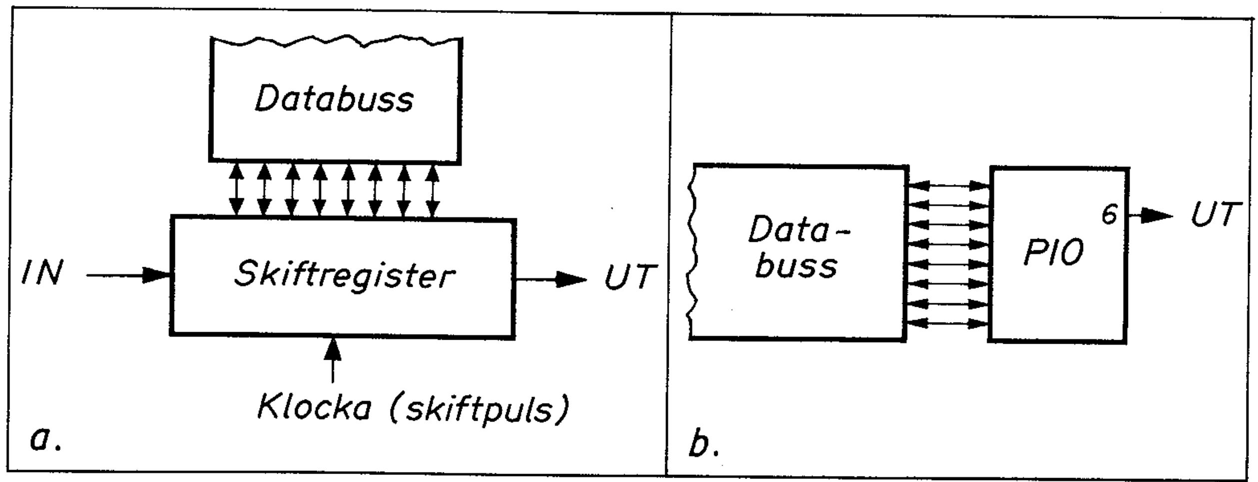


Fig 6.30 Parallell/serie-omvandling
a. I hårdvara (skiftregister)
b. I programvara

Det finns numera specialkretsar för parallell/ serieomvandling. UART:en (universal asynchronous receiver transmitter) var en av de första LSI-kretsarna på marknaden. Den innehåller flera skiftregister och kan automatiskt lägga ut start- och stoppbitar samt kolla paritet.

I de flesta mikrodatorfamiljer ingår en SIO (serial input output) eller ASIA (asynchronous serial input output adapter). Det är i princip en UART som kan programmeras och som har kontrollsignaler som passar ihop med mikrodatorfamiljen i övrigt.

I ABC80 har man löst parallell/serie-omvandlingen med programvara. Bit 6 i PIO:ns port B används som serieutgång. För att avlasta PIO:n sitter det buffrar (position G6 och H7 i fig 6. 29) mellan PIO:n och kassettuttaget.

3.2.3 Avspelning

På ingången (från kassettbandspelaren) sitter en RC-länk (deriverande krets) med tidskonstanten = 0,1 ms. Normalt körs ABC80 med pulser av storleken ca 1 ms. RC-länken släpper enbart igenom flankerna.

På RC-länkens utgång får vi positiva och negativa spikar. Fig 6.31.

Efter RC-länken följer två komparatorer (position J8). De har normalt högohmiga utgångar. Endast när insignalen överskrider referensnivån (som är +0,5 resp -0,5 V) blir utgången ledande mot jord. Komparatorerna är så inkopplade att den ena avkänner positiva spänningar och den andra negativa.

Resultatet blir att vi vänder spikarna från RC-länken och får en negativ puls för varje flank i insignalen. Denna puls ett-ställer en vippa (position G5). Q-utgången från denna vippa matar bit 7 i port B på PIO:n.

Om vippan är nollställd kommer den första flank som uppträder i insignalen att ettställa vippan. Man har nu möjlighet att låta CPU:n avkänna bit 7 och därefter vid lämplig tidpunkt nollställa vippan och därmed åter göra den mottaglig för flanker i insignalen.

ABC80 avger fyrkantsignal vid inspelning och avkänner flankerna i kurvformen vid avspelning. ABC80 kan därmed programmeras för vilken som helst av de tidigare visade inspelningsmetoderna.

Programvaran i ABC80 innehåller en "cassette handler". Det är ett in- och avspelningsprogram som arbetar enligt FM-metoden. Klockfrekvensen är här ca 800 Hz och fig 6.31 visar hur man programmässigt avläser ettor och nollor i ABC80. Jämför med fig 6.29. Låt oss följa förloppet i detalj:

t_1: PIO:ns bit 7 (ingång) går hög och detta tolkas (genom PIO:ns programmering) som en avbrottsbegäran.

t_1-t_2: Avbrottsrutinen räknar tid och efter ca 0,4 ms nollställer avbrottsrutinen latchen (position G5) genom att lägga ut en etta på bit 6 (utgång).

t_2-t_3: ABC80 avvaktar nu eventuell flank i insignalen. Om latchen inte ett-ställs tolkar ABC80 den aktuella biten som en nolla, i annat fall som en etta.

t_3: ca 0,8 ms efter klockpulsen t_1 hoppar ABC80 ut ur avbrottsrutinen.

t_3-t_4: ABC80 arbetar nu med att lagra den inlästa biten i lämplig mjukvarubuffert. När detta är klart avvaktar ABC80 en ny avbrottsbegäran. Denna tolkas som nästa klocksignal.

t_4: Bit 7 (ingång) går hög och förloppet från t_1 upprepas.

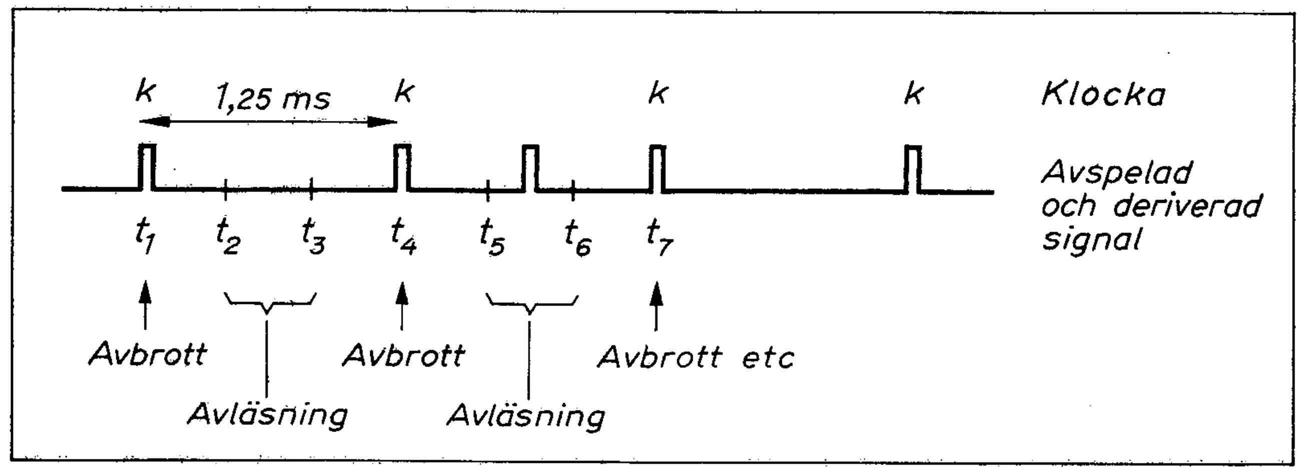


Fig 6.31 Hur ABC80 avkänner signalen från kassettbandspelaren

Digital informationslagring på vanliga kassettbandspelare kräver viss omsorg för att fungera felfritt. Detta gäller både beträffande bandspelare och tonband.

Vi har bekantat oss med tre olika metoder för informationslagring på kassett. Kansas City (KC), Tarbell (PE) och frekvensmodulering (FM). ABC80 har en generell hårdvara som avkänner flankerna i insignalen och därför kan användas till samtliga metoder. ABC80:s programvara arbetar enligt FM-principen.

4. ABC-bussen

DIAB (Data-Industrier AB) som konstruerat ABC80 har tidigare utvecklat en generell buss som kallas 4680-bussen. Till denna buss finns ett stort sortiment med anpassningskort (interface) som saluförs under namnet Databoard av företaget SATTCO.

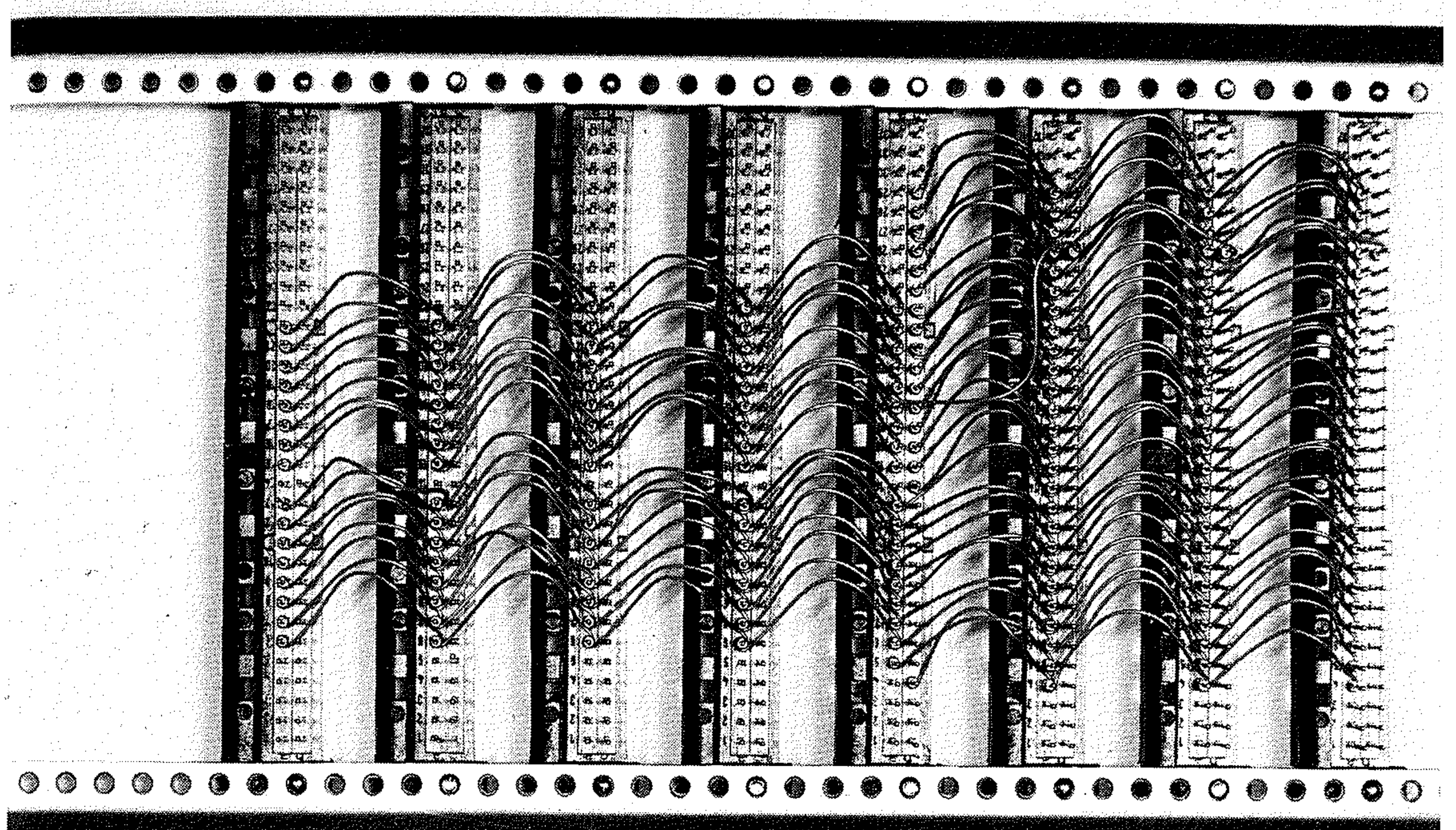
Grundidén med 4680-bussen är att den ska kunna styras av alla vanliga typer av 8 bitars mikroprocessorer. Det finns exempelvis färdiga datorkort som innehåller datorer baserade på Z80 (Zilog), 8080 (Intel), 2650 (Signetics) eller 6502 (Synertec). Oavsett vilket datorkort man använder kan man utnyttja hela Databoard-sortimentet av minnes- och anpassningskort till 4680-bussen.

4680-bussen finns i två varianter. ABC80 kan via busskontakten baktill användas som styrdator till den mindre varianten, som vi därför i fortsättningen kallar ABC-bussen.

Fig 6.32a visar ett foto av en ABC-buss med 9 kontakter (de två vänstra används för strömförsörjningen). Fig 6.32b visar hur ABC-bussen kan användas. Bussen består av två delar, en minnesdel till höger och en IO-del till vänster. Mellan minnes- och IO-delen ansluts datorkortet och det är här vi kopplar in ABC80 med hjälp av en flatkabel.

ABC-bussens minnesdel är avsedd för minneskort. På fig 6.32b är två RAM-kort på vardera 8K byte utritade. På detta sätt kan ABC80:s minneskapacitet utökas från 16K byte till 32K byte. (Se fig 5.6).

I ABC-bussens IO-del kan vi sätta in anpassningskort för olika typer av periferiutrustning. I fig 6.32b sitter anpassningskort till två mycket viktiga utrustningar, en printer och ett floppy-disk-system. Här hade vi också kunnat plugga in ett övergångskort till den internationellt standardiserade IEC-bussen och då kan ABC80 användas som styrdator i ett mätsystem.



a.

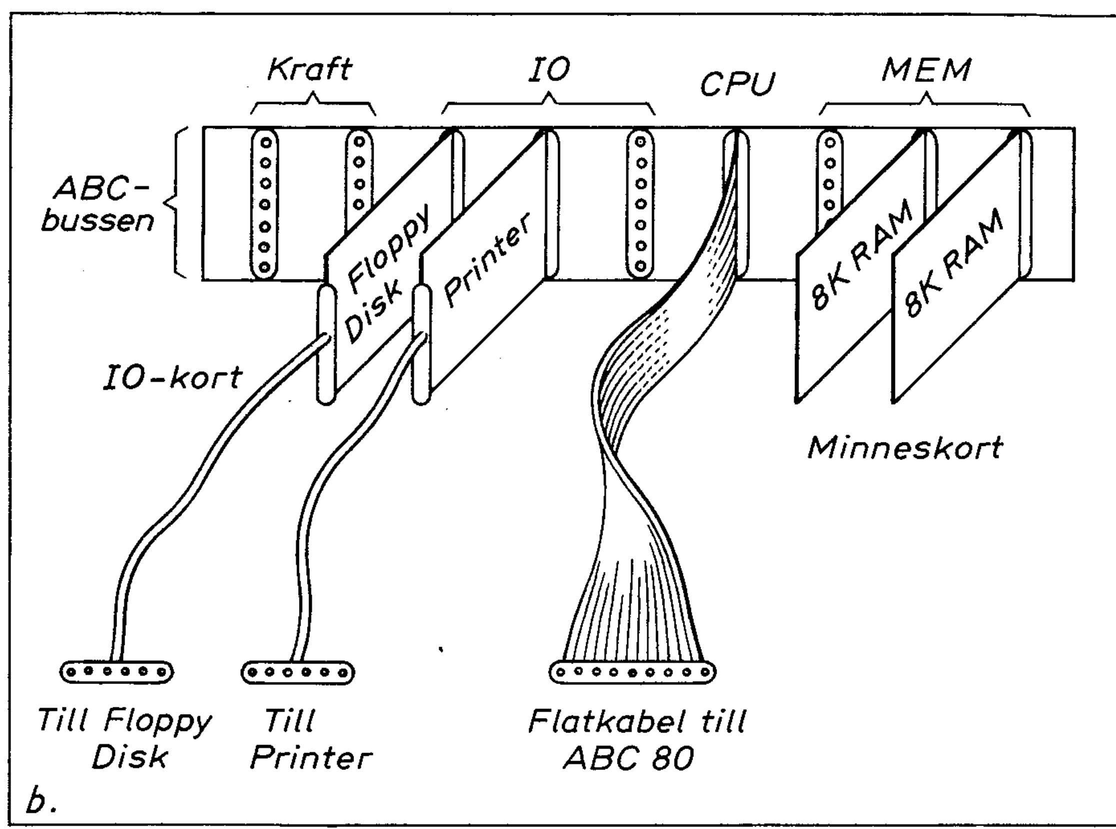


Fig 6.32 ABC-bussen
a. Foto
b. Principen för inkoppling av CPU, IO- och minneskort

ABC-bussens IO-del kan hantera upp till 64 anpassningskort (det är bara att utöka IO-sidan på bussen i fig 6.32). Utbyggnadsmöjligheterna för ABC-systemet är därför praktiskt taget obegränsade.

Vi ska nedan först studera ABC-bussens uppbyggnad och kontrollsignalernas funktion. Därefter ska vi visa exempel på uppbyggnaden av ett minnes- och ett anpassningskort.

4.1 Principen

Principen för ABC-bussen framgår av fig 6.33. Man frågar sig kanske varför ABC-bussen är uppdelad i en minnesdel och IO-del. Orsaken är att olika mikroprocessorer använder olika metoder för IO-adressering. För att kunna använda samma buss för alla typer av mikroprocessorer har man delat upp bussen på en minnesdel (som är tämligen lika för olika mikroprocessorer) och en IO-del som är specialgjord för ABC-bussen och mycket enkel att adressera och använda.

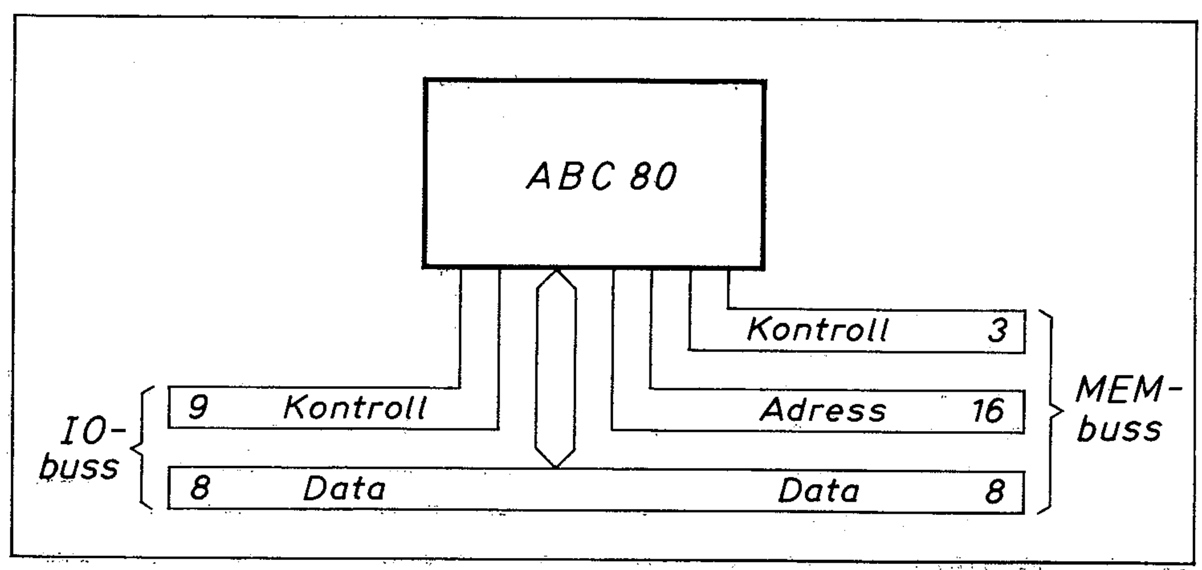


Fig 6.33 Principen för ABC-bussen

Som framgår av fig 6.33 är vissa delar av bussen gemensamma för både minnes- och IO-sidan. Följande pinnar har samma funktion på minnes- och IO-sidan:

Gemensamma ledare		
Ben	**Funktion**	
A6 . . . A13	DØ . . . D7	databuss
A31, B31	+5 V	ström-försörjning
A2, B2	0 V	
A32, B32	+12 V	
A1, B1	-12 V	

4.2 Minnesbussen

ABC-bussens minnesdel innehåller ovanstående gemensamma ledningar samt dessutom en komplett adressbuss (16 bitar) och tre kontrolledningar:

Specifika ledare för minnesbussen	
Ben	**Funktion**
A14 . . . A29	A15 . . . AØ — adressbuss
A4	$\overline{\text{XMEMFL}}$ (= läs)
A5	$\overline{\text{XMEMW}}$ (= skriv)
A30	RDY (= ready)

Kontrollsignalerna $\overline{\text{XMEMFL}}$ och $\overline{\text{XMEMW}}$ har vi behandlat tidigare (avsnitt 6.11).

RDY-ledningen (ben A30 i minnesbussen) styr $\overline{\text{WAIT}}$-ingången på Z80 och hålls normalt hög av ett pull up-motstånd i ABC80 (fig 5.23). Om RDY-ledningen hålls låg läggs Z80 i $\overline{\text{WAIT}}$-läge, dvs all verksamhet avstannar tills RDY åter går hög. Om man vill ha informationen kvar i det dynamiska minnet får inte RDY ligga låg alltför länge, för under denna tid får dynamiska minnet ingen refresh.

RDY-ledningen kan med hjälp av lämpliga kretsar utnyttjas för att ge CPU:n order om ett begränsat antal väntecykler i de fall när man använder långsamma minneskapslar. På ABC-bussen bör enbart snabba minneskapslar användas (med accesstid < 350 ns) och kabeln mellan ABC80 och ABC-bussen bör göras kort (< 1 m).

4.3 Exempel på ett minneskort

Fig 6.34a visar schemat för ett minneskort ur Databoard-serien. Det är ett 8K byte statiskt RAM uppbyggt av samma minneskapslar som vi tidigare sett i bildminnet (4114). Vi har tidigare i fig 5.29 sett principen för minneskortets inkoppling till ABC-bussen och i fig 6.34a kan vi nu studera detaljerna.

4.3.1 Adressering

Adresseringen är uppdelad i tre delar. Eftersom en kapsel innehåller 1K adresser kräver varje kapsel 10 adressledningar (AØ-A9).

Två kapslar ger en 1K byte. För att välja ut rätt par av de 16 minneskapslarna används en "en av åtta"-avkodare (typ LS138 i position B1). Denna utväljer alltså 1K byte inom ett block på 8K byte.

Slutligen måste kortet (som innehåller 8K byte) placeras på önskad adress inom det totala adressutrymmet på 64K. Det är här adressbyglingen längst ner på fig 6.34a kommer in i bilden. Med tre inverterande XOR-kretsar (LS266 i position C1) kan önskat 8K-block avkodas från adressbitarna A13-A15. Vill vi exempelvis utöka minnet i ABC80 med 8K byte ska vi välja det 8K block som gränsar till interna RAM-minnet, dvs som täcker hexadresserna AØØØ - BFFF (fig 5.6). Om alla 3 adressbyglarna är inkopplade får vi startadressen ØØØØ. Om A15-bygeln klipps av får vi startadressen 8ØØØ (dvs övre 32K-arean). Med A13-bygeln avklippt flyttar vi oss ytterligare 8K uppåt i minnet och där har vi nu startadressen AØØØ.

Byglingsavkodningen ger klarsignal (enable) till "en av åtta"-avkodaren och styr signalgrindarna till databuffert och minneskapslarnas $\overline{WR}$-ingångar.

4.3.2 Kontrollsignaler

Minnesbussen har tre kontrollsignaler. Två av dessa är styrsignaler från CPU:n till yttre minnet.

$\overline{XMEMFL}$ = läskommando

$\overline{XMEMW}$ = skrivkommando

Lässignalen $\overline{XMEMFL}$ i fig 6.34a ger klarsignal (E) via en OR-grind till 1K-avkodaren (position B1) och grindas med 8K-avkodningen (byglingsadressen) så att databufferten riktas mot CPU:n. Förutsatt att byglingsadressen stämmer kommer nu adresserad minnesposition att läsas.

Skrivsignalen $\overline{XMEMW}$ riktar på motsvarande sätt databufferten från CPU:n mot minnet, ger klarsignal till 1K-avkodaren och WR-signal till minneskapslarna.

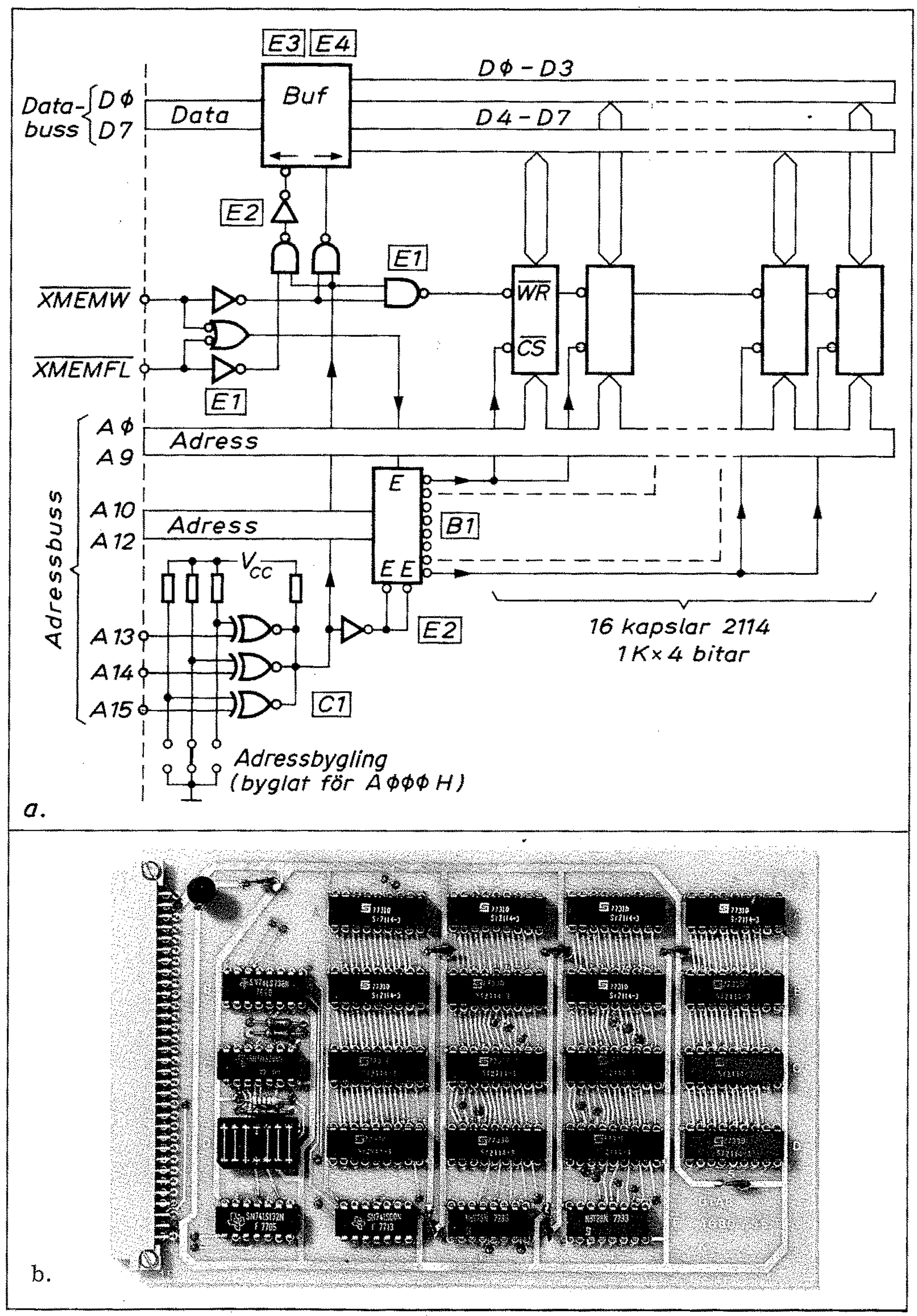


Fig 6.34 Minneskort till ABC-bussen (Databoard)
a. Schema
b. Foto

För att minska störkänsligheten har man valt grindar med hysteres (dvs som har till- och frånslag vid skilda nivåer) i position E1.

Det är en nyttig övning att med utgångspunkt från fig 6.34a tänka igenom de olika kretsarnas funktion när kortet adresseras för läsning eller skrivning.

4.4 IO-bussen

IO-bussen har en funktion som avviker från allt vi tidigare varit i kontakt med. Utöver databussen innehåller IO-bussen nio kontrollledningar. Vi har redan i fig 5.31 sett att åtta av dessa ledningar egentligen är klarsignaler (chip select eller enable) till 6 utportar (adresserna Ø-5) och två inportar (adresserna Ø-1). I ABC-bussen används emellertid dessa kontrolledningar för speciella funktioner och har därför fått följande beteckningar:

Specifika ledare i IO-bussen				
Ben	Port-adress	Beteckning, Funktion		
A22	UTØ	$\overline{OUT}$ = data ut		aviserar utsignaler från CPU till IO
A23	UT1	$\overline{CS}$ = kortval		
A21	UT2	$\overline{C1}$	kommando	
A20	UT3	$\overline{C2}$		
A19	UT4	$\overline{C3}$		
A18	UT5	$\overline{C4}$		
A17	INØ	$\overline{INP}$ = data in		begär insignaler från IO till CPU
A16	IN1	$\overline{STAT}$ = status in		
A15	IN7	RST = programstyrd RESET till IO		

Fig 6.35 visar principen för anpassningskorten till ABC-bussens IO-del. Man frågar sig här hur det kan vara möjligt att adressera 64 olika IO-kort utan att ha tillgång till en adressbuss. En av finesserna med ABC-bussen (eller 4680-bussen) ligger just i adresseringen av IO-korten. Nr A23 i IO-bussen innehåller klarsignal för utport 1. Denna signal kallas $\overline{CS}$, en förkortning av card select (kort- eller kanalval). När signalen läggs ut tolkar alla IO-kort på ABC-bussen databussens innehålla som en adress. Alla IO-korten

jämför då sin egen adress med databussens värde. Finns det ett kort med den på databussen utlagda adressen så kommer card select-elektroniken på detta kort att kopplas in. Kortet blir därmed utvalt och dess in- och utportar kan nu styras med hjälp av. olika signaler på kontrollbussen. Kortet förblir inkopplat ända till nästa kortval signaleras med hjälp av $\overline{CS}$-signal på kontrollbussen.

För kortval utnyttjas endast sex bitar (DØ-D5) av databussens åtta bitar.

Card select-elektroniken är så utformad att den kopplar bort (frikopplar) IO-kortet för alla $\overline{CS}$-signaler med annan adress än kortets egen. Om IO-korten byglas med olika adresser kan alltså endast ett kort i taget vara inkopplat.

I fig 6.35 har vi nu sett principen för IO-korten på ABC-bussen. Låt oss studera ett IO-kort mera i detalj.

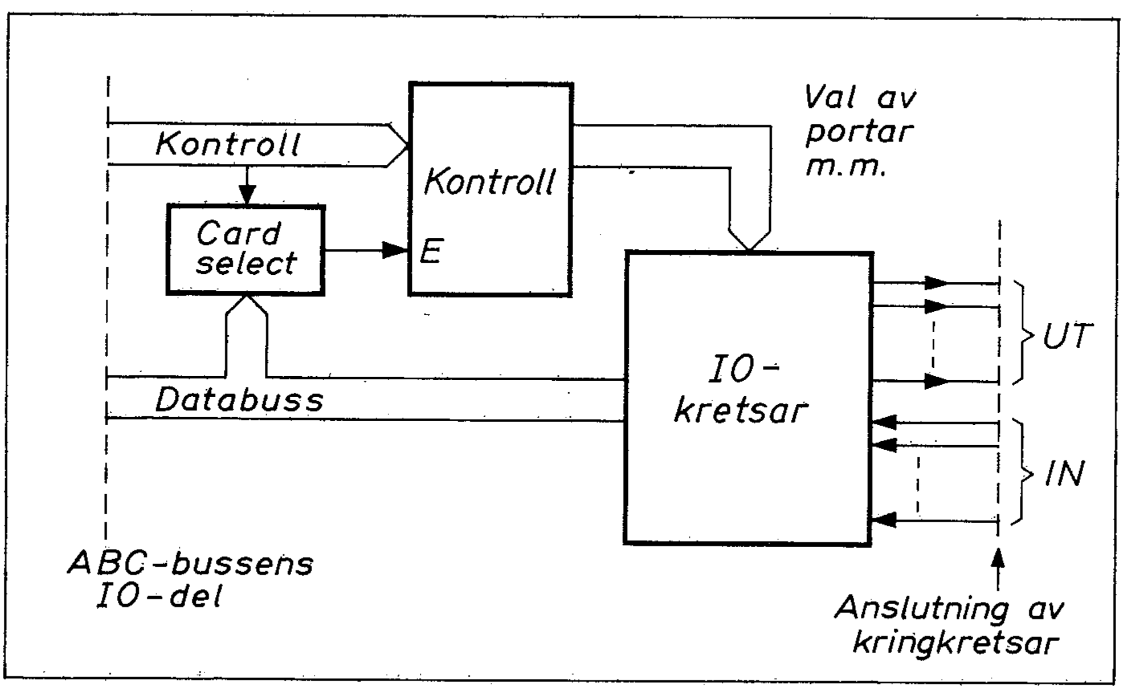


Fig 6.35 Principen för 4680-bussens anpassningskort

4.5 Exempel på ett IO-kort

Fig 6.36a visar schemat för ett IO-kort med 8 reläutgångar. Här kan vi se exempel både på adressering (kanalval) och användningen av några kontrollsignaler i ABC-bussen.

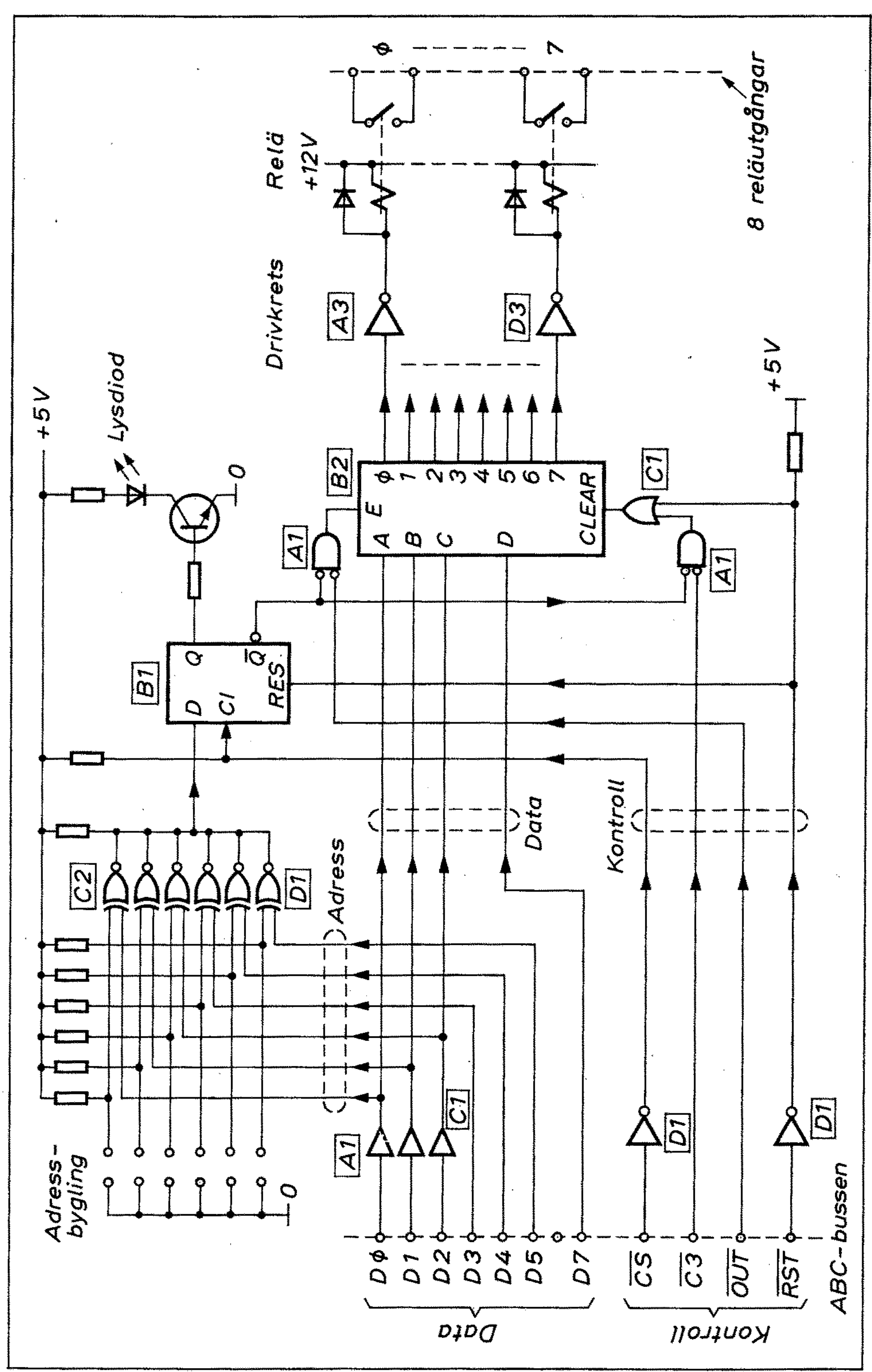


Fig 6.36a IO-kort med åtta reläutgångar

4.5.1 Kanalval

Ett IO-kort på ABC-bussen adresseras med kontrollsignaler $\overline{CS}$ (= card select) och den adress som samtidigt ligger på bitarna Ø-5 på databussen. Om vi följer $\overline{CS}$-ledningen i fig 6.36a hamnar vi på klockingången till en D-vippa (position B1).

Vippans dataingång matas från en adressavkodare av samma typ vi tidigare mött i fig 6.34a (för adressbitarna A13-A15). På IO-kortet i fig 6.36a hämtas adressen från bitarna DØ-D5 och jämförs med adressbyglingen längst uppe till vänster. Om samtliga byglingar är uppklippta (som i fig 6.36a) blir decimala adressen (kanalnumret) 63. Om byglingsadressen och data sammanfaller kommer D-vippan att ettställas.

IO-kortet är därmed utvalt (adresserat) och det är nu mottagligt för kontrollsignaler. För att markera kanalvalet tänds en lysdiod som är kopplad till vippans Q-utgång via en transistor.

4.5.2 Kontrollsignaler

IO-kortet i fig 6.36 innehåller 8 reläer. Varje relä drivs av en inverterande drivkrets (75454 i position A3, B3, C3, D3). Drivkretsarna matas från specialkretsen 74259 (8-bit adressable latch) vilken i princip innehåller en "en av åtta"-avkodare samt åtta D-vippor.

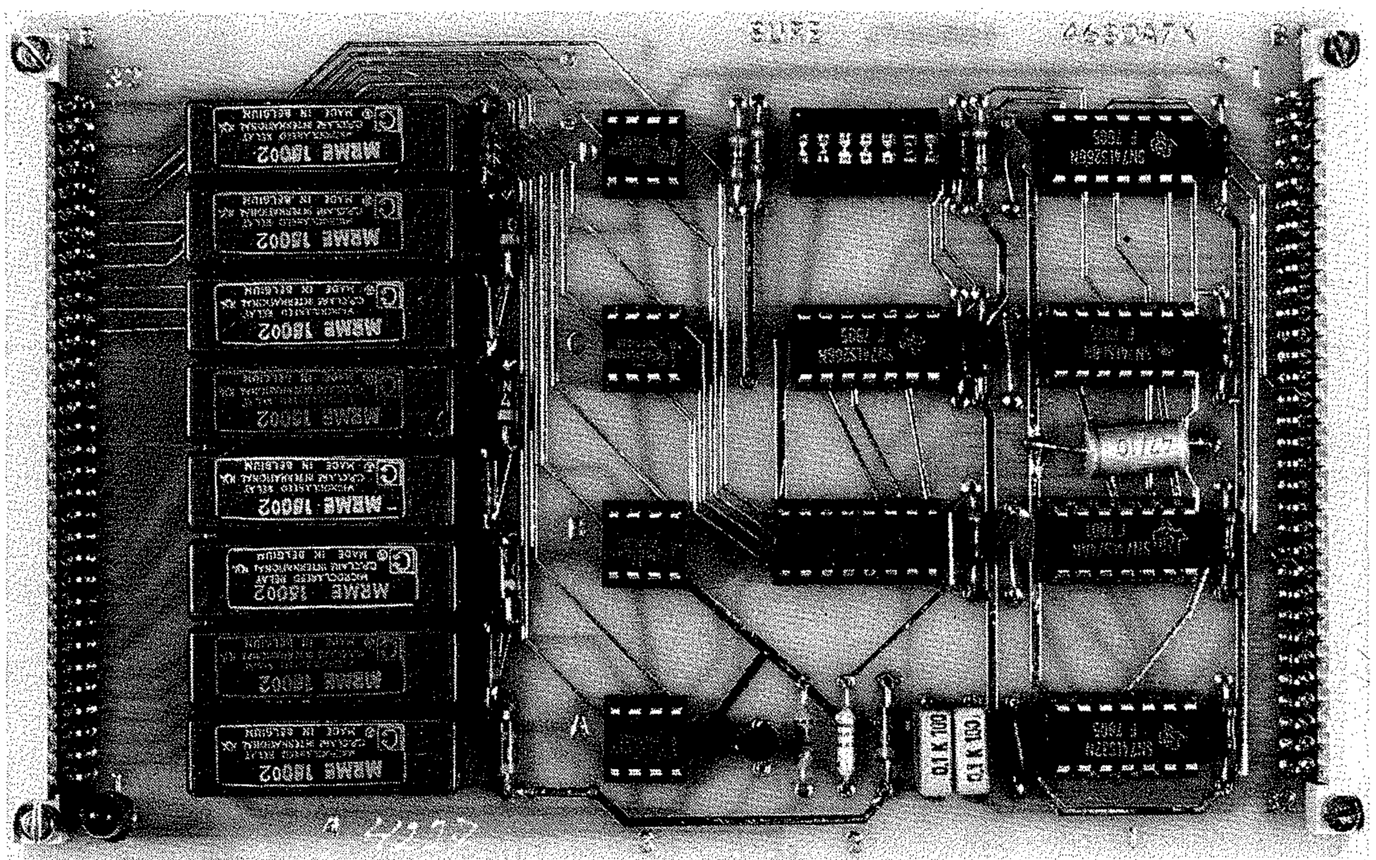


Fig 6.36b Foto av IO-kort

Med ingångarna ABC kan man adressera en av vipporna i taget och denna ställs då enligt villkoret på D-ingången. Med CLEAR-ingången nollställs samtliga vippor omedelbart.

Låt oss nu se hur man kan använda kontrollsignalerna! Utöver $\overline{CS}$ används för IO-kortet i fig 6.36 enbart signalerna $\overline{C3}$, $\overline{OUT}$ och $\overline{RST}$.

När kortet är utvalt kan vi med $\overline{OUT}$-signal ge avkodaren klarsignal (enable). Därmed kommer bitarna DØ-D3 att adressera avkodaren (dvs välja relä) och bit D7 att avgöra om adresserad utgång ska ligga hög eller låg.

Med $\overline{C3}$-signal kan samtliga drivkretsar nollställas. Då släpper alla reläer samtidigt.

Med $\overline{RST}$-signal nollställs såväl avkodaren (dvs samtliga reläer släpper) som D-vippan. Observera att $\overline{RST}$-signal nollställer hela IO-bussen dvs alla anpassningskort som utnyttjar $\overline{RST}$-signal.

Databoard-manualen beskriver kontrollsignalerna på följande sätt:

SIGNALERING MELLAN CPU OCH 4007

INP DATA	-
INP STAT	-
OUT DATA	Styrning av reläer D0-D2 = Binär adress till relä som ska slås till eller från D7 = 1 = drag relä D7 = 0 = släpp relä Ett relä åt gången styrs
OUT C1	-
OUT C2	-
OUT C3	Släpp alla reläer
OUT C4	-
	D3-D6 är ej valida vid OUT DATA. Dessa kan stå hur som helst Med OUT C3 släpps alla reläer ovillkorligen.

För reläkort (Databoard 4007) används tydligen inte någon av följande signaler: $\overline{INP}$, $\overline{STAT}$, $\overline{C1}$, $\overline{C2}$ och $\overline{C4}$.

4.6 ABC-bussens praktiska uppbyggnad

I fig 6.32a såg vi kontaktdonen i en ABC-buss med plats för 3 IO-kort och 3 minneskort. Fig 6.37 visar en låda som inrymmer såväl ABC-bussen från fig 6.32a som transformator och likriktare för strömförsörjningen. Bussen sitter i lådans bakre del. Minnes- och IO-kort skjuts in i ABC-bussens kontaktdon och stagas av styrskenor upptill och nertill. I lådans framkant skymtar IO-kortens kontaktdon för anslutning av kringkretsar exempelvis reläutgångarna i fig 6.36.

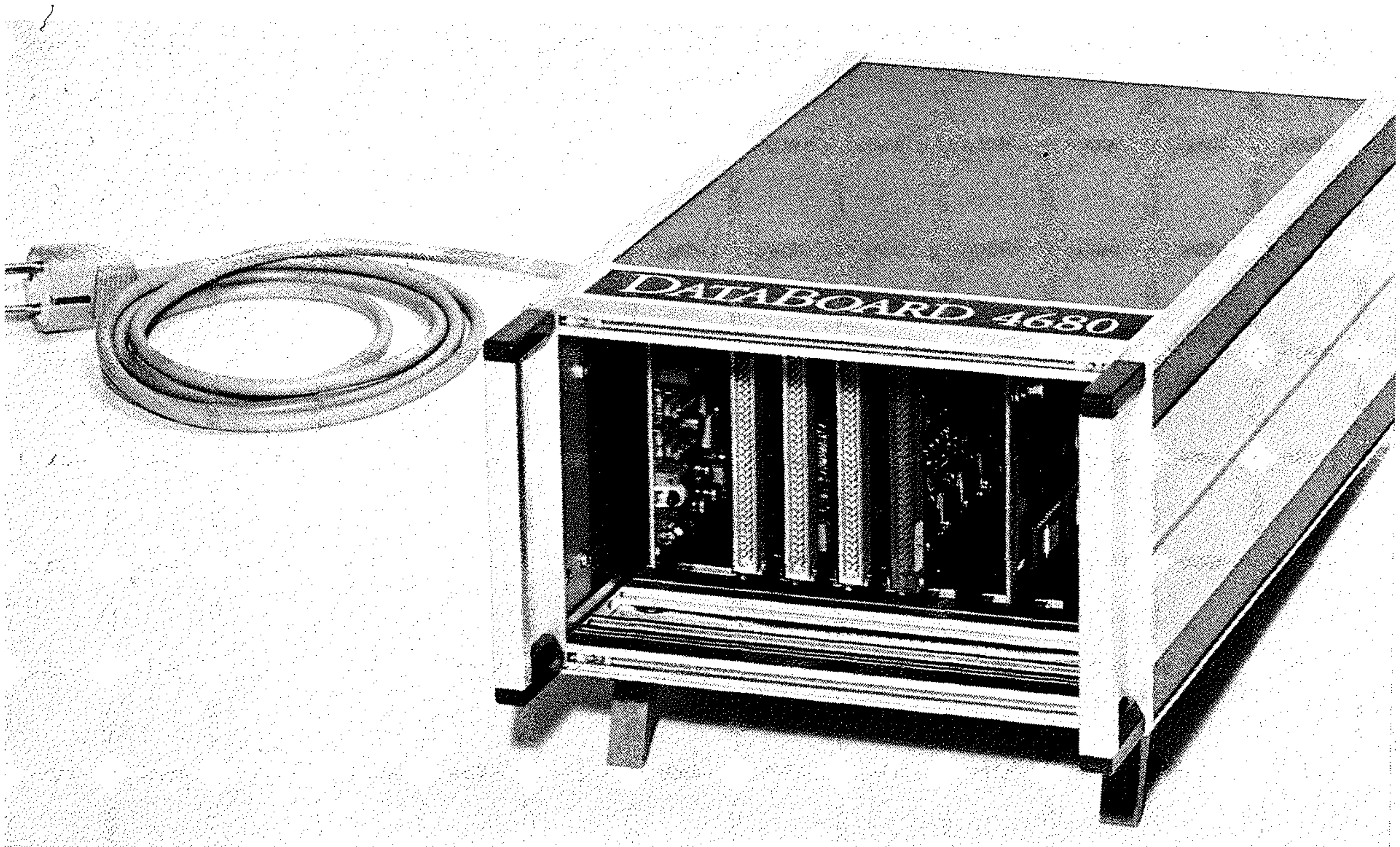


Fig 6.37 Exempel på en låda innehållande ABC-buss och strömförsörjning

Det står givetvis var och en fritt att bygga sina egna minnes- och IO-kort till ABC-bussen. Databoard-serien innehåller ett rikt urval kort för den som är mer intresserad av tillämpningar än av kortkonstruktion. Fig 6.38 visar några exempel.

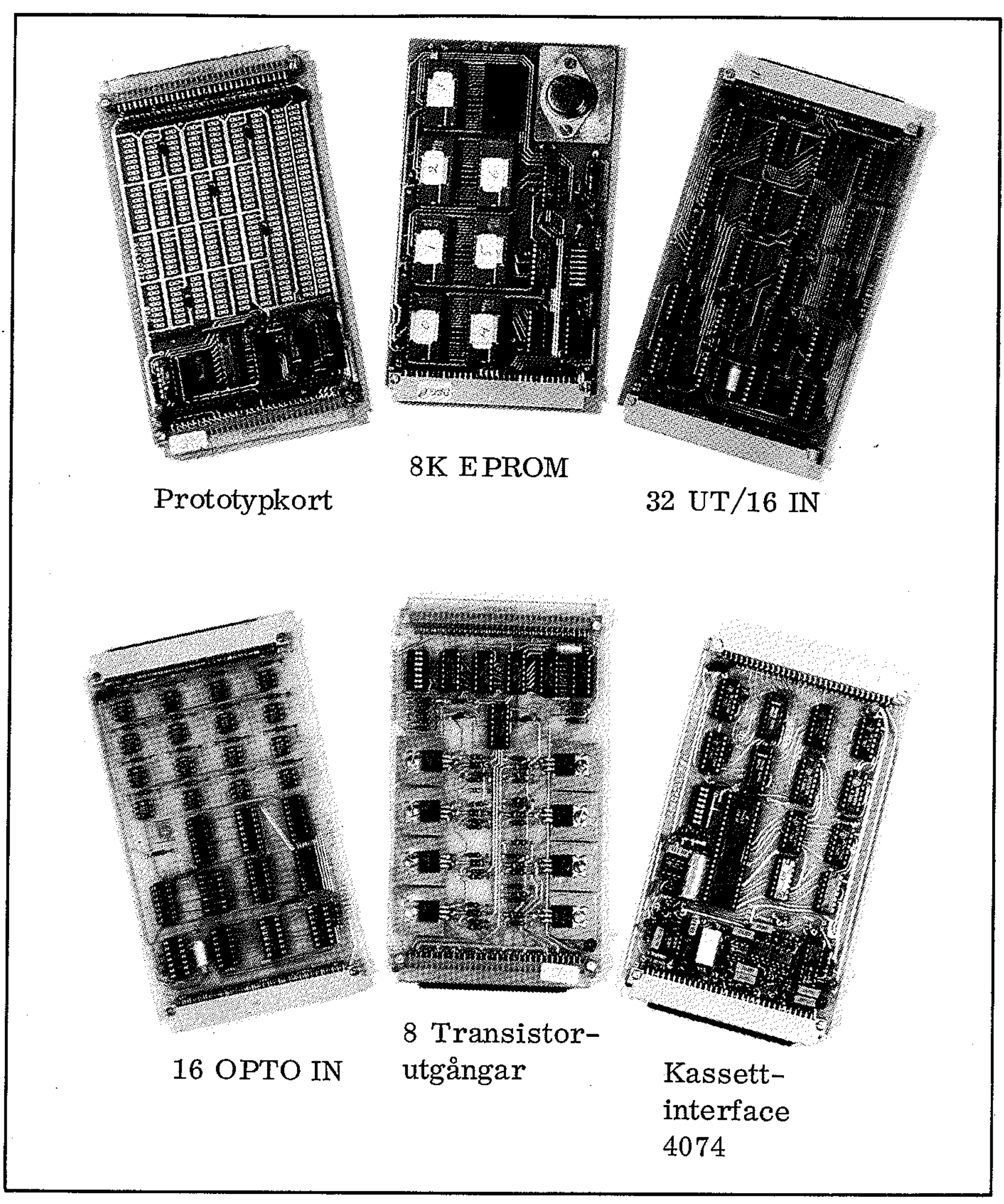


Fig 6.38 Exempel på Databoard-kort för ABC-bussen

Sammanfattning

ABC-bussen gör det möjligt att använda ABC80 för alla tänkbara ändamål. ABC-bussen är alltså en mycket väsentlig del i ABC80-systemet.

För att kunna utnyttja ABC-bussens fulla kapacitet är det nödvändigt att förstå både funktion och signaler hos minnes- och IO-kort. Detta är orsaken till att vi i detta avsnitt relativt utförligt beskrivit både ett minneskort och ett IO-kort till ABC-bussen.

7. Programvara

I kapitel 5 och 6 har vi bit för bit gått igenom elektroniken i ABC80. Hur vårt mikrodatorsystem nu kommer att fungera beror på det program vi laddar in. Var och en skulle i princip kunna sätta in sitt eget programpaket i ABC80 och därmed få sina egna speciella önskemål uppfyllda. Utveckling av programvaran är emellertid ett så omfattande arbete att detta knappast är realistiskt.

Vi har tidigare sett vilken möda som ligger bakom en enda programsida (fig 4.11). Listningen av ABC80:s programvara fyller helt tre A4-pärmar, fig 7.1.

ABC80:s programpaket omfattar 16K byte och ligger lagrat i ROM. ABC80 är därmed direkt färdig att användas. Så fort vi slår på strömbrytaren (baktill på monitorn) så sätts detta programpaket i arbete och ABC80 svarar med klarsignal i övre vänstra hörnet på bildskärmen:

ABC80
☐ (☐ är en blinkande markör)

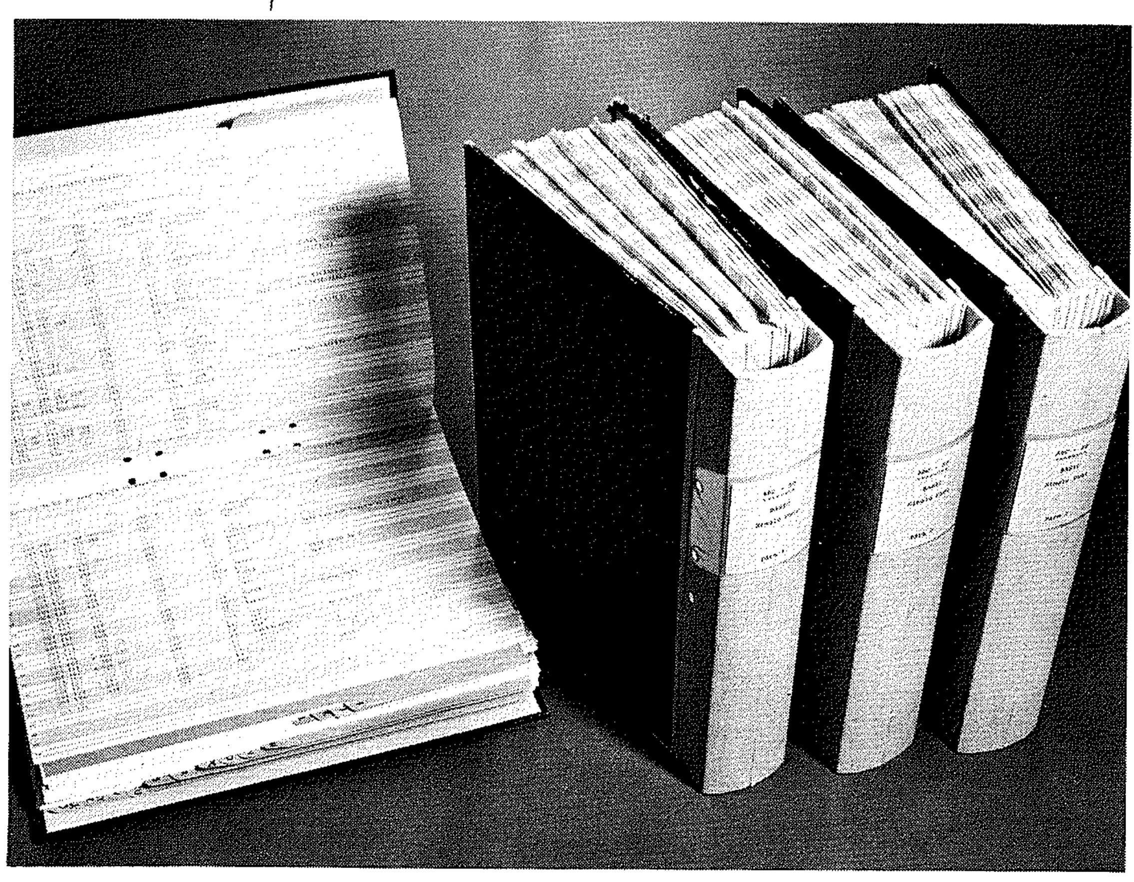


Fig 7.1 Listningen av programvaran i ABC80

Det är nu hög tid att läsa bruksanvisningen (manualen) som medföljer ABC80. Där får vi en beskrivning av vilka kommandon som ABC80 accepterar. Där står också hur vi kan skriva (eller ladda) in program i minnet och hur vi sedan kan exekvera dessa program. Man behöver givetvis inte känna till något som helst om elektroniken i ABC80 för att kunna använda ABC80-systemet. Man måste emellertid noggrant följa den "syntax" som anges i manualen.

För att kunna använda ABC80 måste man alltså följa vissa formella regler. Dessa regler bildar programspråket BASIC och sammanfattas i ABC80-manualen. Det finns ett stort antal läroböcker om BASIC på marknaden. Boken ABC om BASIC är skriven speciellt för ABC80.

Meningen med denna bok är ju att beskriva elektroniken och funktionen hos ABC80. Vi förutsätter här att läsaren har läst manualen och därigenom fått en bakgrund till hur ABC80 fungerar "utifrån sett". Avsikten med detta kapitel är nu att krypa innanför skalet på ABC80 och få en antydan om hur programvaran är uppbyggd och hur den hanterar de kretsar vi tidigare studerat.

Vi ska börja med initialiseringen av systemet, det är ju ett begrepp som vi mött tidigare i kap 4.

Vi ska därefter se hur BASIC-tolken är uppbyggd i princip. Med hjälp av korta programsnuttar ska vi sedan demonstrera programvarans kontakt med elektroniken.

ABC-bussen kräver speciell programmering om vi till fullo ska kunna utnyttja finesserna på Databoard-korten. Vi ska visa ett exempel.

1. Initialisering

När man trycker in nätströmbrytaren på ABC80 vill man givetvis att ABC80 direkt ska vara färdig att användas. Låt oss kort gå igenom några av de rutiner som förbereder ABC80 för att ta emot våra kommandon.

1.1 Tre rutiner

Tre rutiner är här självklara, bildskärmen ska vara rensuddad (raderad), ljudgeneratorn ska vara tyst och PIO:n ska vara så programmerad att bitarna i port A, som tar emot signalerna från tangentbordet, fungerar som ingångar.

Bildskärmen fylls med mellanslag genom att motsvarande ASCII-kod (2Ø H) placeras i hela bildminnet. Ljudgeneratorn blir tyst om vi

nollställer latchen i position G7 och det sker med signalen $\overline{\text{POC}}$ (processor clear). Hur en PIO ska programmeras har vi tidigare sett exempel på i kap 4.

En annan självklar initialiseringsrutin gäller motorstyrningen till bandspelaren. Givetvis vill vi att bandspelarens motor ska slås ifrån vid initieringen. Den ska ju senare kunna startas under programkontroll.

En detalj är viktig när man slår på strömmen på ABC80. Z80 måste ges en $\overline{\text{RESET}}$-signal för att starta. Detta sker automatiskt när vi trycker på nätströmbrytaren (dvs vid "kallstart" eller "power up"). Fig 7.2 visar de detaljer i schemat som ansvarar för kallstarten och som vi tidigare utelämnat i fig 5.23.

Som vi tidigare beskrivit kan Z80 ges $\overline{\text{RESET}}$ antingen manuellt (via knappen bredvid uttaget för ABC-bussen) eller via $\overline{\text{RESIN}}$-signal på ABC-bussen. Detta sker med hjälp av grindarna F7. Den erhållna nollställningssignalen tillförs $\overline{\text{RES}}$-ingången på Z80 och går samtidigt ut som $\overline{\text{POC}}$-signal på ABC-kortet.

Den vänstra grinden F7 i fig 7.2 har två ingångar. Den ena av dessa ingångar har en kondensator (C = 150 μF) till jord och ett motstånd (R = 10 kΩ) till V_{CC}. Detta ger en tidskonstant på RC = 1, 5 s. Vid tillslag av nätströmbrytaren hinner alla matningsspänningar stabiliseras innan kondensatorn hunnit laddas upp så långt att den nivåkännande grinden i position F7 avger $\overline{\text{RESET}}$-signal till Z80.

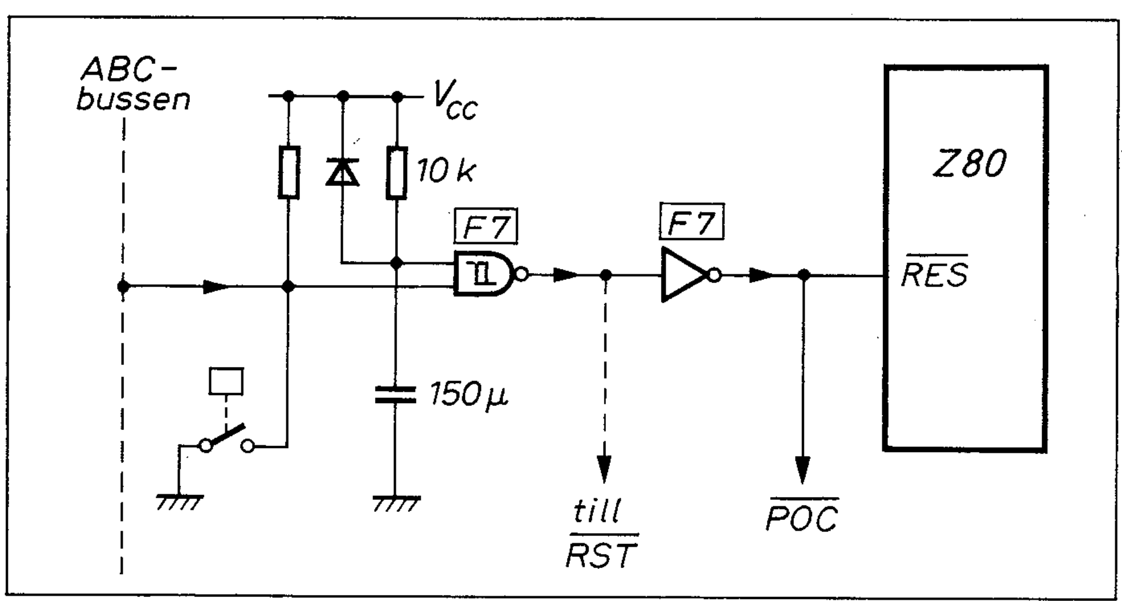


Fig 7.2 Kretsen för kallstart av ABC80

Vid ett kort spänningsavbrott urladdas kondensatorn kvickt via den diod som ligger parallellt med 10 kΩ -motståndet. Även vid korta spänningsavbrott fungerar alltså elektroniken för den fördröjda $\overline{\text{RES}}$-signalen.

1.2 Övriga nollställningar

Det är inte enbart Z80 som ska nollställas vid kallstart. Interface på ABC-bussen måste nollställas med $\overline{\text{RST}}$-signal så att de kopplas bort och inte åstadkommer förvirring på IO-bussen.

Ett flertal flaggor (i detta fall bitar i vissa minnespositioner) måste nollställas. En av dessa flaggor anger om data finns tillgängliga från tangentbordet, en annan flagga anger om felutskrifterna ska göras i klartext (innehållet ska då hämtas från floppy), en flagga (panikflaggan) talar om att tangenterna CTRL och C är nedtryckta samtidigt. Alla dessa flaggor ska alltså nollställas vid kallstart.

Flera mjukvaruregister måste nollställas. Exempel på ett sådant register är den timer som ger tiden och exempelvis upprepar tecken om man håller en tangent nedtryckt under längre tid.

1.3 PIO:ns programmering

Vi har tidigare nämnt att port A ska programmeras att fungera som inport. Enligt fig 5.23 ska port B programmeras för utgångar på bitarna 3, 4 och 5 och med resterande bitar som ingångar. Vidare ska både port A och port B generera avbrottsbegäran ($\overline{\text{INT}}$) på bit 7.

Därför måste PIO:n laddas med masker så att avbrott sker på de önskade bitarna och ger önskad utsignal (hög eller låg).

Port A ska laddas med den låga delen av en lagringsadress (interruptvektorn 34H) och Z80:s I-register ska laddas med den höga adressdelen (I = ØØH).

Vid nedtryckning av en tangent får Z80 avbrottsbegäran ($\overline{\text{INT}}$) och exekverar som svar den rutin vars startadress är lagrad i adressen ØØ34H i programminnet.

1.4 Pekare

Det finns ett flertal pekare, dvs register som innehåller (pekar ut) olika adresser. Dessa pekare måste innehålla vissa utgångsvärden efter initialiseringen. Vi har tidigare behandlat stackpekaren men i ABC80 finns ytterligare ett antal pekare. Fig 7.3.

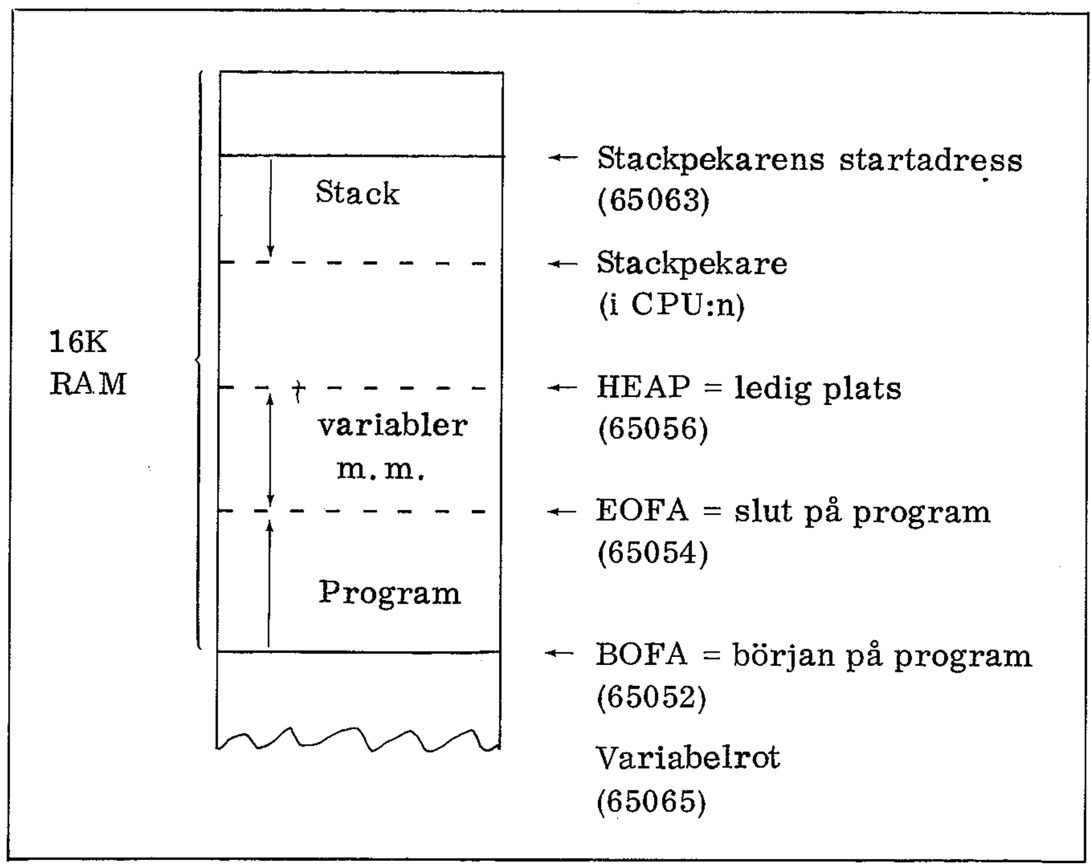


Fig 7.3 Exempel på pekare i ABC80. Decimala pekaradresser inom parentes

BOFA (begin of file address) är en pekare som visar var programmet ska lagras.

EOFA (end of file address) pekar ut slutet av det program som lagrats i minnet. EOFA ska alltså ställas in på samma adress som BOFA vid initialiseringen.

HEAP pekar ut ledigt minnesutrymme.

Det finns dessutom variabelpekare (exempelvis variabelroten).

1.5 Minnet

I ABC80:s initialisering ingår en rutin som avsöker RAM-minnet. Genom att först skriva in ett tal och sedan kolla om talet blivit inskrivet kan ABC80 kontrollera om det existerar ett RAM-minne på den aktuella adressen. På detta sätt scannar ABC80 adressområdet från CØØØH till 8ØØØH i fig 5.6 och ser därvid om vi satt en extra RAM-minne på ABC-bussen.

När ABC80 fått reda på hur mycket RAM-minne som är tillgängligt ställer ABC80 själv in minnespekare (exempelvis BOFA) så att hela det tillgängliga minnesutrymmet utnyttjas.

Vill vi undanta ett område i RAM för egna assemblerrutiner kan vi ställa in BOFA-pekaren "manuellt". Det kan ske med hjälp av en POKE-instruktion som laddar önskad minnesadress med angivet innehåll.

När vi ändrat BOFA-pekaren måste vi ge ABC80 ett NEW-kommando för att EOFA, HEAP och en del andra pekare ska flyttas i relation till BOFA.

På motsvarande sätt kollar ABC80 om det sitter något ROM inkopplat på adressen 6ØØØH. Om ett ROM är inkopplat här antar ABC80 att vi satt in ett floppy-disk-interface på ABC-bussen. Då följer ytterligare en omfattande initialisering för att göra floppy-disken körklar.

Vi kan inte i detalj gå in på hela initialiseringen men klart är att om ett mikrodatorsystem ska vara bekvämt och praktiskt att hantera och använda så måste initialiseringen vara väl genomtänkt.

När initialiseringen är klar ligger ABC80 i en vänteslinga och avvaktar avbrottssignal från tangentbordet. Den blinkande markören visar var det tecken vi anslår på tangentbordet kommer att placeras på bildskärmen.

2. BASIC-tolken

Vi antar nu att ABC80 är initialiserad och visar sitt klartecken (ABC80 och blinkande markör) på bildskärmen. Vad händer när vi trycker ned en tangent?

2.1 Radbufferten BUF1

När vi trycker ned en tangent får Z80 avbrottsbegäran och hoppar till en inläsningsrutin vars startadress är lagrad på adress ØØ34H. ASCII-koden från tangentbordet läses nu in och placeras i en radbuffert (BUF1), som börjar på en viss adress i RAM:et. Om vi överskrider den maximalt tillåtna längden så lägger ABC80 själv till en vagnretur i sista tillåtna positionen. Buffertpekaren återställs till buffertens startadress och vi får börja om från början (vilket ABC80 anger med ERR 11).

Genom att anslå ett antal tangenter får vi en lång rad med tecken på bildskärmen. Vi kallar detta för en teckensträng.

Vi avslutar inläsningen av teckensträngen med RETURN-tangenten (CR = carriage return). Fig 7.4 sammanfattar händelseförloppet.

① markerar att en inläsningsrutin har anropats genom avbrottsbegäran från tangentbordet. Den nedtryckta tangentens ASCII-kod har placerats i en radbuffert som i fig 7.4 betecknas BUF1.

② anger att den inlästa strängen avslutas med RETURN-tecken och att ABC80 trätt i full verksamhet för att tolka teckensträngens innehåll.

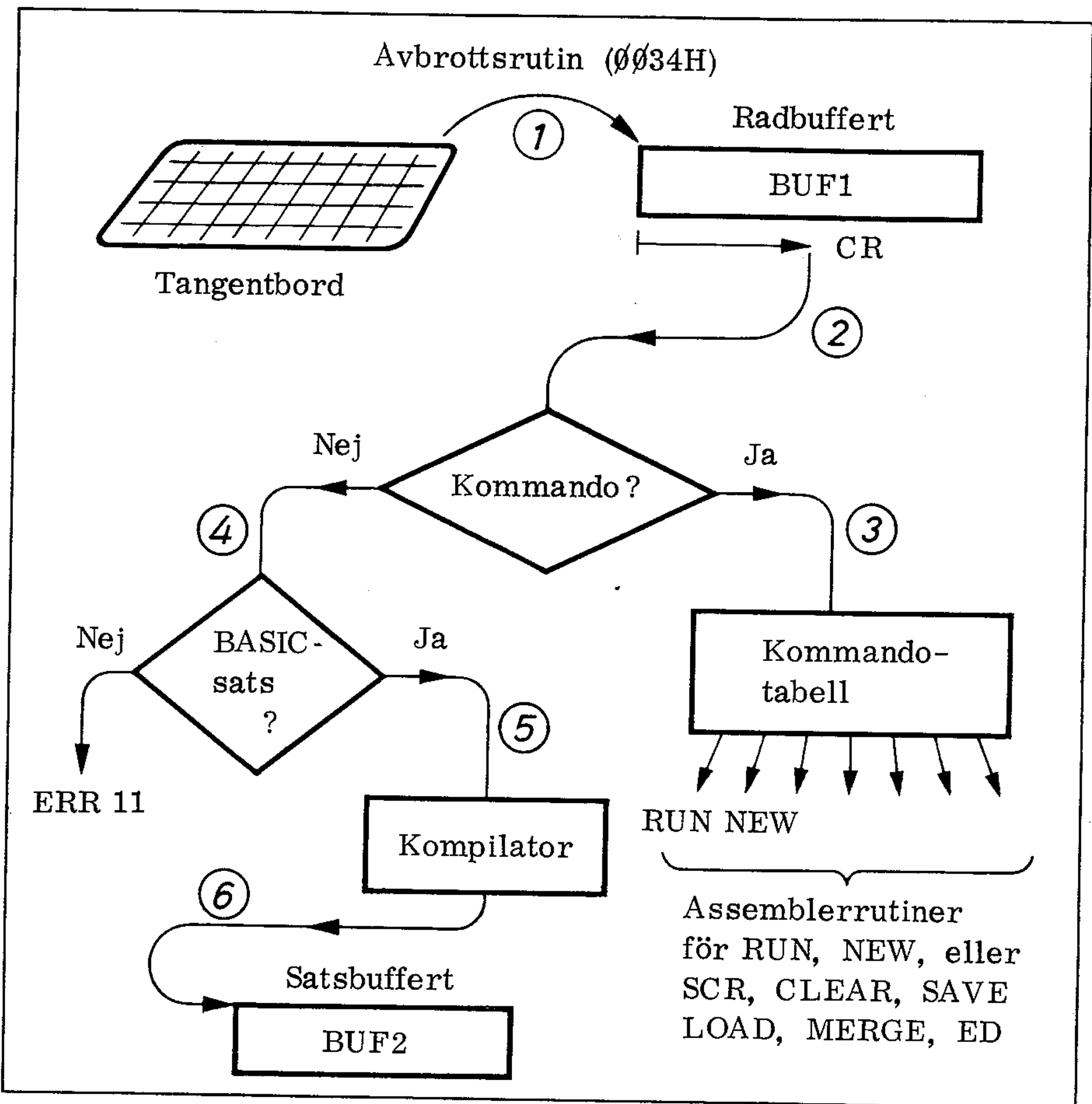


Fig 7.4 Tolkning av innehållet i BUF1

2.2 Kommando eller BASIC-sats

Vi fortsätter nu med fig 7.4 och följer markeringarna.

③ anger att ABC80 indentifierat ett kommando. ABC80 skiljer på två typer av teckensträngar: kommandon och BASIC-satser. ABC80 jämför därför teckensträngen med en kommandotabell. Finner ABC80 att teckensträngen innehåller ett kommando exekveras mot-

svarande kommandorutin. (Exempelvis RUN för att starta ett program eller LOAD för att ladda ett program).

④ anger att ABC80 ej kan finna något kommando i teckensträngen. ABC80 försöker därför tolka strängen som en BASIC-sats. Som framgår av manualen finns det en mängd olika satser att välja på. Om ABC80 exempelvis påträffar strängen

```
PRINT "JOHAN"
```

så tolkar den PRINT som en operationskod och JOHAN som en teckensträng som ska skrivas ut på bildskärmen.

ABC80 är tolerant vad gäller antalet mellanslag. De två teckensträngarna

```
PRINT "JOHAN"
PRINT     "JOHAN"
```

tolkas på samma sätt. Men om ett enda felaktigt tecken ingår i operationskoden förkastas hela teckensträngen omedelbart. Skriver vi

```
PRIN "JOHAN"
```

får vi direkt svaret "pip" och följande utskrift på bildskärmen:

```
ERR 11
ABC80
□
```

ERR 11 betyder enligt manualen "Förstår ej". ABC80 har alltså inte kunnat tolka våra avsikter med teckensträngen och avvaktar därför en ny och mera korrekt teckensträng.

2.3 Satsbufferten BUF2

Vi har nu kommit till kompilatorn nederst i fig 7.4.

⑤ anger att ABC80 tolkat teckensträngen i BUF1 som en BASIC-sats. Det finns flera olika typer av BASIC-tolkar. ABC80 har en avancerad och snabb BASIC och detta beror bl a på att den teckensträng som blivit accepterad som BASIC-sats nu kodas och inplaceras på ett mycket systematiskt sätt i en satsbuffert (BUF2). Detta är en form av kompilering och satsbufferten innehåller alltså en komprimerad BASIC-kod som vid programlagring betecknas ".BAC". Koden är optimerad för att ge snabb exekvering.

När ABC80 lagrar program (med kommandot SAVE) lagrar den alltså inte de textsträngar som vi skrivit in via tangentbordet utan istället en komprimerad kod.

Om vi vill ha en listning av den komprimerade BASIC-koden (med kommandot LIST) genererar ABC80 en läsbar listning av den komprimerade BASIC-koden. Därvid får man BASIC-satserna utskrivna på ett standardiserat sätt (med versaler och mellanslag).

⑥ visar hur den komprimerade koden lagras i satsbufferten BUF2. Vi ska strax visa ett exempel på hur denna lagring av komprimerad BASIC-kod kan se ut.

2.4 Tre typer av variabler

ABC80 kan hantera tre typer av variabler

flyttal, exempelvis A = 3.14
heltal, exempelvis B% = 256
strängar, exempelvis C = "ABC80"

Ett flyttal upptar 5 byte i minnet och lagras enligt fig 7.5. Talområdet sträcker sig därmed från $\pm 0.1 \cdot 10^{-127}$ till $\pm 0.999999 \cdot 10^{127}$.

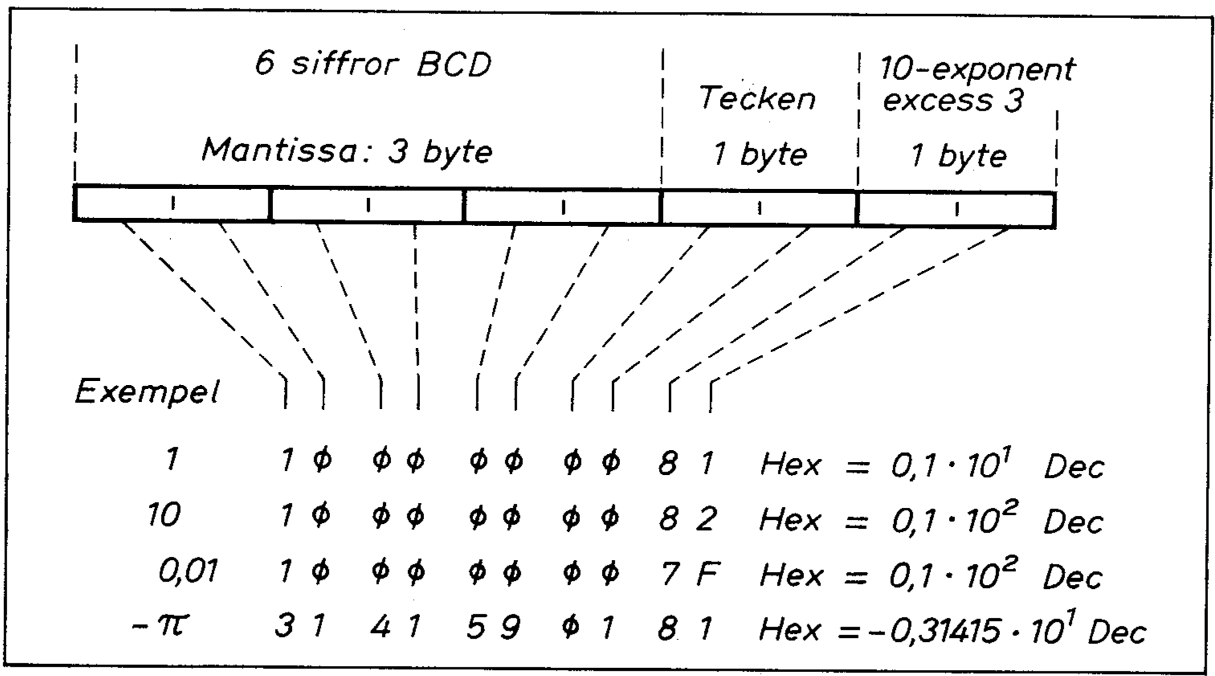


Fig 7.5 Lagring av flyttal

Heltal betecknas med ett efterföljande %-tecken och lagras i binär form i två byte. Mest signifikanta biten anger tecken och därför blir talområdet -32768 till +32767.

För att enkelt kunna ange 16 bitars adresser som decimaltal accepterar ABC80 heltal upp till 65536 men lagrar dem som negativa tal. Satsen PRINT 65065% ger exempelvis utskriften -471. Adressen

65065 kan lika gärna anges med heltalet -471. (Exempel på detta framgår av rad 40 i fig 7.22).

Strängar som inte deklareras får i ABC80 ha maximala längden 80 tecken.

Flyt- och heltalsvariabler samt pekare till stränger lagras i minnet direkt efter programmet. En speciell pekare (variabel-roten) anger adressen till det minnesutrymme som används för lagring av variabler.

2.5 Internkodskompilatorn

När en BASIC-sats har accepterats så kompileras den (fig 7.4) av en internkodskompilator. Vi får som nämnts en komprimerad kod som tar liten plats att lagra och som går snabbt att exekvera.

Internkodskompilatorn är intressant ur många synvinklar. För att ge en antydan om hur kompileringen utförs och hur den komprimerade koden som lagras i BUF2 ser ut ska vi visa ett exempel.

Antag att en BASIC-sats har följande utseende

PRINT 1% + 2%

Satsen består av två delar:

- o PRINT är operationskod och anger att efterföljande uttryck ska beräknas och därefter skrivas ut på bildskärmen.
- o 1% + 2% är det uttryck som ska beräknas.

Kompilatorn omvandlar PRINT till en komprimerad binärkod (1 byte).

Efterföljande uttryck analyseras och placeras i omvänd polsk notation (RPN = Reverse Polish Notation) i BUF2.

Fig 7.6 visar hur PRINT 1% + 2% lagras i BUF2. Lagringen av uttrycket sker som synes på samma sätt som motsvarande tal och plus-tecken skulle slås in på en räknedosa med RPN (exempelvis HP25).

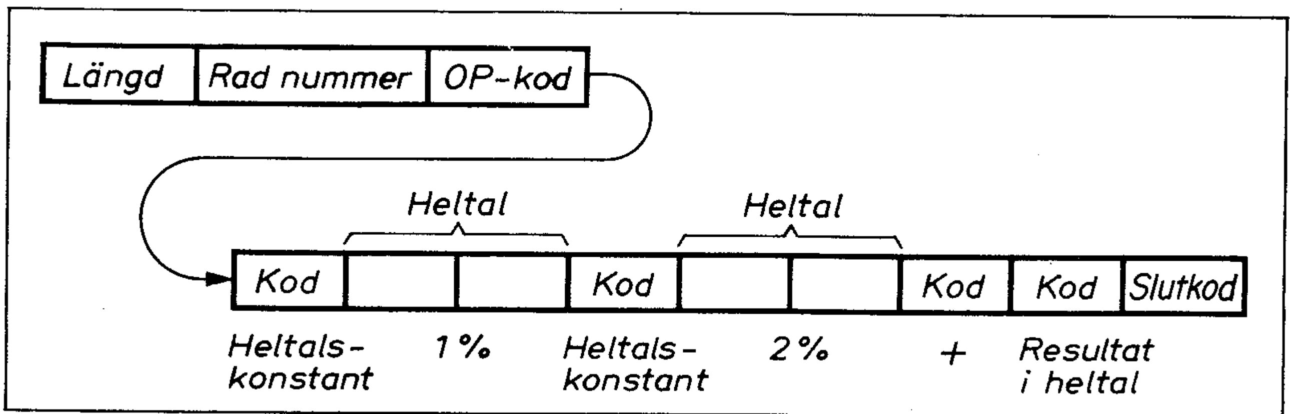


Fig 7.6 Lagring av BASIC-satsen PRINT 1% + 2% i BUF2. Varje ruta upptar 1 byte.

Man frågar sig kanske nu hur innehållet i BUF2 hade sett ut om uttrycket i PRINT-satsen hade bestått av flyttal istället för heltal:

PRINT 2 + 3

Vi har tidigare sett (fig 7.5) att flyttal kräver 5 byte och vårt uttryck i fig 7.6 hade tydligen blivit längre. Totala innehållet i BUF2 hade ökat från 12 byte (fig 7.6) till 18 byte.

Givetvis tar det längre tid och mer minnesutrymme att hantera flyttal än heltal. Enkla BASIC-tolkar avsedda för små hobbydatorer innehåller ofta endast heltal. Exempel härpå är PALO ALTO TINY BASIC (listad i majnumret 1976 av Dr Dobb's Journal of Computer Calisthenics & Orthodontia, Box 310, Menlo Park, California 94025).

2.6 Exekvering eller lagring

Det är nu dags att ta upp den röda tråden från fig 7.4. Vi har tidigare sett hur en BASIC-sats kan kompileras av en internkodkompilator till en komprimerad kod. Denna kod placeras i BUF2 och därvid inplaceras eventuella uttryck som senare ska beräknas i RPN-följd.

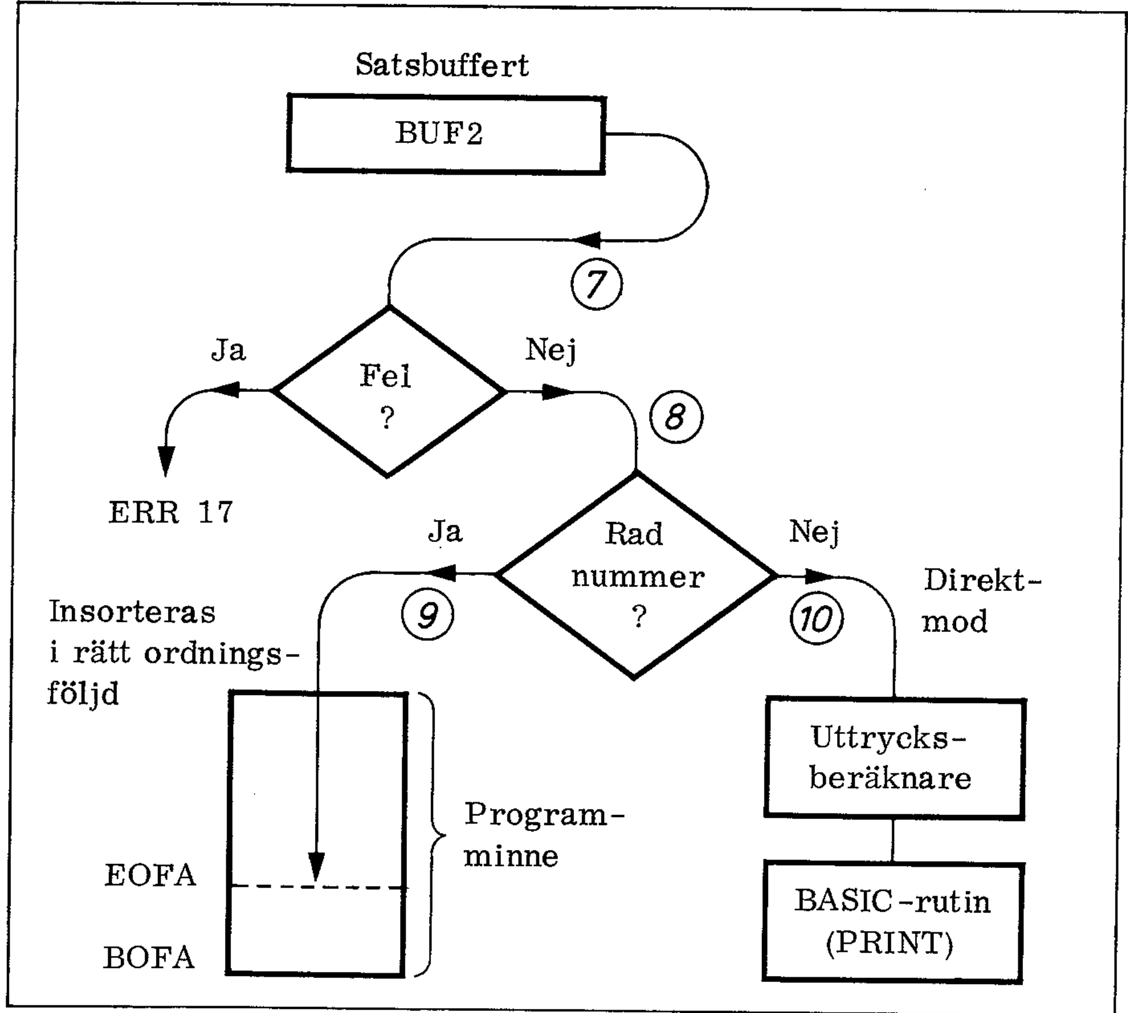


Fig 7.7 Hanteringen av koden i BUF2

Fig 7.7 visar ABC80:s fortsatta hantering av BASIC-satsen. Vi ska följa den steg för steg.

⑦ visar hur koden från BUF2 kontrolleras med hänsyn till syntax. Uttrycket kan exempelvis vara felaktigt uppställt. Olika typer av variabler kan vara sammanblandade på ett otillåtet sätt.

⑧ anger att koden har accepterats av ABC80. Nu kollar ABC80 om BASIC-satsen ska lagras i programbufferten eller om den ska exekveras i direktmod:

20 PRINT 1% + 2%	PRINT 1% + 2%
ska lagras	exekveras direkt

⑨ anger att ABC80 funnit att BASIC-satsen inleds med ett antal siffror. Siffror som placeras framför operations-koden för en BASIC-sats tolkas som radnummer. Den komprimerade koden för en BASIC-sats med radnummer sänds vidare till en programbuffert. En rutin inplacerar satserna i ordningsföljd. Den tidigare nämnda pekaren EOFA pekar på en adress omedelbart efter den sats som har högsta radnummer. Pekaren EOFA uppdateras med hjälp av första byten i koden från BUF2 (fig 7.6).

⑩ visar att BASIC-satsen ej innehåller radnummer och därför kommer den att exekveras i direkt-mod. Först beräknas det uttryck som följer efter operationskoden. Detta arbete utförs av ett omfattande programpaket som vi här kan kalla "uttrycksberäknaren". Därefter tolkas operationskoden och motsvarande rutin hanterar resultatet från uttrycksberäknaren.

I vårt enkla exempel adderade uttrycksberäknaren heltalen 1% och 2%. PRINT-rutinen fick sedan överta resultatet (summan 3%) och skriva ut det på bildskärmen.

Det finns ett antal BASIC-satser som ej är fristående och därmed ej kan exekveras i direkt-mod:

FOR	NEXT	DEF	DIM
GOTO	GOSUB	ON	ON ERROR
DATA	RETURN	STOP	

2.7 Programexekvering

Man brukar skilja mellan kompilerande språk (exempelvis FORTRAN) och interpreterande språk.

BASIC är ett interpreterande språk. Den engelska termen "interpret" betyder "tolka". Vi talar därför om en BASIC-tolk.

En kompilator översätter högnivåspråkets satser till maskinkod (objektkod). Man kompilerar alltså först och det kan ske i en annan dator än den i vilken man senare exekverar den erhållna objektkoden. Fördelarna med en kompilator är att man får en mycket effektiv objektkod som tar liten tid att exekvera. Nackdelarna är att kompilatorn måste kompilera hela programmet (eller i varje fall stora delar) samtidigt. Man kan inte "provköra" en enstaka sats.

En interpretator (som vår BASIC-tolk) exekverar en sats i taget. Vi kan alltså provköra en enda sats och se hur resultatet blir. Ett interpreterande språk är "interaktivt", dvs vi kan sitta vid vår dator och testa programmet sats för sats. Nackdelen är att många interpretatorer är mycket långsamma. När exempelvis en slinga ska köras igenom ett stort antal gånger måste varje sats tolkas på nytt för varje varv som slingan exekveras. Det är här ABC80:s interkodskompilator och komprimerade kod har betydelse. ABC80:s BASIC behöver inte tolka långa textstränger när ett program ska exekveras. ABC80 har en komprimerad kod som går snabbt att exekvera.

När vi ger kommandot RUN kommer ABC80 att hämta och exekvera sats efter sats i programbufferten. De flesta fel är redan avlusade vid vid programskrivningen eftersom satserna kontrolleras före inplaceringen i BUF2. Fel som beror på samspelet mellan satser blir emellertid först upptäckta vid den slutliga exekveringen. Exempel på ett sådant fel är om man glömt NEXT N efter en tidigare sats FOR N = 1 TO 10.

2.8 Kassettrutinerna

Det är många rutiner i ABC80 som skulle vara värda ett närmare studium. Tyvärr ligger detta utanför ramen för denna bok. För att ge en bättre bakgrund till datalagringen på kassettband ska vi emellertid översiktligt beskriva SAVE-rutinen.

SAVE är alltså det BASIC-kommando man använder när man vill lagra programminnets innehåll på kassett. SAVE-kommandot ska åtföljas av ett namn på högst åtta tecken. Med kommandot

SAVE SPADER

lagrar vi programminnets innehåll på en fil på kassettbandet. Kommandot döper samtidigt denna fil till namnet SPADER.

Inspelningsformat

På ABC80:s kassettband ligger filerna lagrade med 5 s mellanrum enligt fig 7.8 överst. På detta sätt är det enklare att lokalisera en fil på bandet.

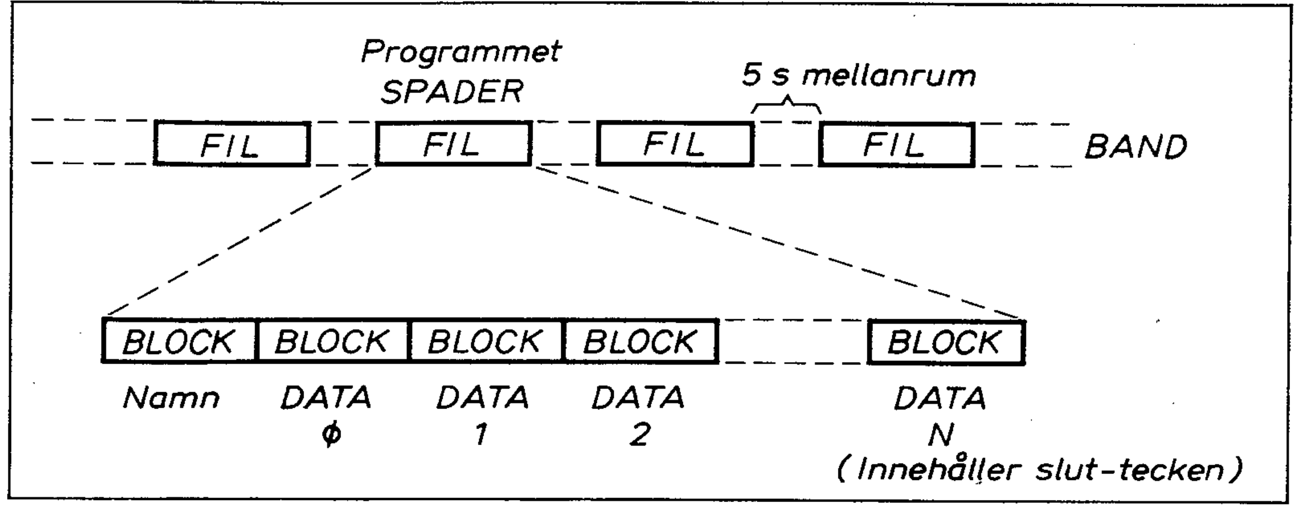


Fig 7.8 Filer och block på ett inspelat kassettband

Varje fil består av ett antal block. Det första av dessa block är ett namnblock som innehåller en "header" samt namnet på filen (i vårt fall SPADER). Efterföljande block är datablock och det sista blocket innehåller ett avslutningstecken.

Alla block har en likformig uppbyggnad och innehåller följande:

Ett block

256 bitar noll
3 st SYNC 16H
1 st STX 2H
256 byte datadel
1 st ETX 3H
2 byte binär checksumma

SYNC-tecknet är så valt att det inte ska kunna förväxlas när det skiftats in i rätt läge i mottagarens buffert. STX betyder "start of transmission" och ETX betyder "end of transmission". Checksumman erhålls genom addition av alla byte i blocket, från och med tecknet efter STX och till och med tecknet ETX.

Datadelen i ett block har olika innehåll beroende på om det är ett namnblock eller ett datablock. Namnblockets datadel har följande innehåll:

Datadel i Namnblock

3 st FFH header
8 byte namn (SPADER)
3 byte extension (.BAC)
resten nollor

Headern består av 3 byte FFH och anger alltså att här startar en ny fil. Namnet kan vara upp till 8 tecken och namndelen upptar alltid 8 byte.

Det finns ytterligare tre byte som kan användas för att närmare beskriva innehållet i filen. Om det exempelvis är en programfil med komprimerad BASIC-kod lagras beteckningen (extension) BAC i dessa tre byte. Om det är en textfil eller en bildfil används beteckningen TXT respektive PIC osv.

Datablockets datadel har följande innehåll:

Datadelen i Datablock
1 byte ØØH
2 byte binärt blocknummer
253 byte användar-data

Blocket börjar med ASCII-tecknet NUL och därefter följer blocknummer (2 byte binärt). Antalet användbara byte per block blir därmed 253.

Inspelningsrutinen

Hur fungerar då rutinen för kommandot

SAVE SPADER

(avslutat med CR)

Enligt ABC80-manualen ska bandspelaren laddas med kassett och framspelas till lämpligt ställe på bandet. Vidare ska kassettbandspelaren ställas in för inspelning. Först därefter får vi alltså ge kommandot

SAVE SPADER.

SAVE-rutinen startar nu motorn på kassettbandspelaren och ligger sedan och väntar i 5 sekunder. Därefter sänds ett namnblock som innehåller namnet SPADER (vilket hämtas från kommandot) och extension BAC.

Efter namnblocket sänds ett antal block som innehåller rader som hämtas från programminnet (med början på adressen BOFA och till och med adressen EOFA). För varje block anges blocknummer. Varje block fylls med så många kompletta rader som möjligt. Resterande del av blocket är odefinierat. Checksumman beräknas successivt för varje block genom addition av varje tecken som spelas in (inklusive den odefinierade biten). Efter 256 byte sänds tecknet STX och därefter checksumman. När sista raden har skrivits läggs ett slutmärke på 1 byte i sista blocket.

Avspelningsrutinen

Vill vi ladda programmet SPADER från kassettbandspelaren till ABC80:s minne ger vi kommandot

LOAD SPADER

(avslutat med CR).

Avspelningsrutinen startar nu motorn (och förutsätter att bandspelaren har rätt kassett inlagd och är inställd för avspelning). Bandet går ca 9 sekunder och om ingen signal inkommer innan denna tid gått ut avbryts sökandet.

Om signal inkommer från bandspelaren börjar ABC80 leta efter en header. När en header påträffats jämförs det efterföljande filnamnet med namnet i kommandot LOAD SPADER. Så fortsätter sökandet fil efter fil. Tack vare att pauserna endast är ca 5 sekunder långa stoppas inte bandspelaren vid filväxling på bandet.

När rätt fil påträffats sker avspelningen på likartat sätt som inspelningen. Checksumman kontrollräknas och kollas för varje block och även blocknumren kontrolleras (så att inget block blir överhoppat). Om allt stämmer ända tills slutmarkeringen mottagits ger ABC80 klartecken och då vet vi att programmet lagrats korrekt i ABC80:s programminne. I annat fall får vi ERR 37 på bildskärmen.

3. Demonstrationsprogram

I detta avsnitt ska vi se några exempel på hur man med BASIC-program kan styra elektroniken i ABC80. Vi ska börja med ljudgeneratorn och motorstyrningen till kassettbandspelaren. Därefter ska vi gå vidare och med hjälp av små "programsnuttar" bl a studera tangentbord, bildskärm och användningen av V24-snittet.

3.1 Ljudgeneratorns styrning

I kap 4 har vi detaljstuderat ett assembler-program som bl a lägger ut data på en utport (fig 4.8 och 4.11). Det var de sex lamporna i trafikljuset som vi kunde tända och släcka på detta sätt.

Vår ljudgenerator är ansluten som utport 6 (dvs med adressen 6). Dataordet Ø7H kan läggas ut på denna port med assembler-instruktionerna

```
LD   A,Ø7H
OUT  (Ø6H),A
```

Den första instruktionen laddar ackumulatorn (A) med hextalet Ø7H och den andra kopierar ackumulatorns innehåll till utporten på adress Ø6H.

BASIC är ett högnivåspråk och där sköter BASIC-tolken om detaljarbetet med att flytta data till och från ackumulatorn etc. Motsvarande BASIC-sats kan skrivas

```
OUT 6, 7
```

Det är manualen som ger oss information om hur olika BASIC-satser måste skrivas för att BASIC-tolken ska kunna utföra dem. OUT 6, 7 betyder enligt manualen att decimaltalet 7 skrivs in på utport 6.

Som vi sett tidigare kan vi använda satsen i direktmod och avsluta den med vagnretur: OUT 6, 7 (CR). Då startar ljudgeneratorn omedelbart. Pröva själv!

Vill vi snabbt tysta ljudgeneratorn (som inte låter särskilt vackert med data ut = 7) kan vi trycka ned en tangent. Den avbrottsrutin som läser in tecken från tangentbordet har för bekvämlighets skull även försetts med en $\overline{\text{RESET}}$ till ljudgeneratorn.

Vi kan även skriva OUT 6, 7 som en sats i programminnet (BUF2) och därefter exekvera den med kommandot RUN.

Vi skriver då exempelvis

```
10 OUT 6, 7 (CR)
RUN (CR)
```

Även här kan vi tysta ljudet genom att anslå en tangent. Pröva själv!

Nu har vi sett principen och är mogna att göra vårt första demonstrationsprogram. Det är givetvis intressant att veta hur ljudgeneratorn låter med olika värden på data till utport 6.

Låt oss göra en räknare som långsamt stegar fram från Ø till 255 och för varje steg laddar utport 6.

Att göra en räknare är mycket enkelt i BASIC. Låt variabeln I vara vår räknare. Vi skriver bara två rader:

```
10 FOR I = 0 TO 255
20 NEXT I
```

Om vi exekverar detta program med kommandot RUN hinner vi inte blinka förrän räknaren I har räknat igenom samtliga 256 värden.

Vi har tidigare i trafikljusprogrammet använt subrutinen WAIT (fig 4.11) för att få en önskad tidsfördröjning. Vi fick då ladda ett register med ett stort tal och därefter lägga oss i en dekrementeringsloop. För varje varv i denna loop kontrollerade vi om registrets innehåll minskat till noll och i så fall hoppade vi ur loopen.

I BASIC gör vi en loopräknare mycket enklare. Vi räknade ovan stegen (I) och kan på samma sätt räkna tiden (T) med rutinen

```
FOR T=0 TO 1000
NEXT T
```

som visar sig ta en sekund att exekvera.

Vårt program ser nu ut på följande sätt:

```
10 FOR I=0 TO 255
20 FOR T=0 TO 1000
30 NEXT T
40 NEXT I
```

Testa programmet! Händer det ingenting? Ha tålamod, programmet tar nu nästan fem minuter att exekvera.

Låt oss nu ladda ljudgeneratorn med talet I från räknaren. Det går enkelt genom att skjuta in satsen

```
15 OUT 6, I
```

Starta med RUN och avnjut under 5 minuter alla ABC80:s olika ljudeffekter.

Nu börjar vi givetvis undra vilka värden på I som ger de mest intressanta ljuden. Det är enkelt att ta reda på. Låt ABC80 skriva ut värdet på I. Skjut in satsen

```
16 PRINT I
```

Testa programmet själv! Vi får som synes en kolumn med siffror till vänster på skärmen.

Låt oss slutligen ge programmet en bättre presentation på bildskärmen. Vi börjar då med att sudda bildskärmen med kommandot

```
        PRINT CHR¤ (12)
eller   ; CHR¤ (12)
```

eftersom PRINT även kan skrivas kortare i form av ett semikolon.

Tecknet ¤ (sol) har införts i Europa istället för dollartecknet ($). Följande utskrifter är gjorda på en DIABLO-printer med dollartecknet istället för sol.

Vi vill kanske ha en rubrik på bildskärmen. Det ser kanske snyggt ut att placera rubriken på rad 3 med början på teckenposition 10. Vi placerar då markören på den önskade positionen med satsen CUR(3, 10) och låter rubriken följa inom citationstecken:

```
5; CUR(3, 10) "Ljudgeneratorn"
```

Nu tycker vi kanske att det ser tokigt ut med sifferkolumnen till vänster. Vi kan "printa ut" variabeln I på önskad plats med hjälp av markören och vi ändrar därför rad 16 till

16; CUR(6,15) I

Med ED-kommandot är det mycket enkelt att ändra i en sats utan att behöva skriva om den helt.

Om vi slutligen ger kommandot REN (= renumbering) så får vi radnummer med början på 10 och med jämna tiotal.

Nu är det dags att spara vårt första demonstrationsprogram på kassett. Vi kanske vill döpa programmet till LJUD. Med bandkassett på plats och bandspelaren inställd för inspelning ger vi kommandot

SAVE LJUD

Vill vi gardera oss för eventuella fel på bandet (avbrott i oxidbeläggningen etc) kan vi göra flera inspelningar efter varandra med upprepade SAVE LJUD-kommandon. Vill vi dessutom hålla reda på våra inspelningar är det nog enklast att endast ha ett eller några få program på varje band. Fig 7.9 visar programmet.

```
ABC80
list
10 ; CHR¤(12)
20 ; CUR(3,10)"Ljudgeneratorn"
30 FOR I=0 TO 255
40 OUT 6,I
50 ; CUR(6,15)I
60 FOR T=0 TO 1000
70 NEXT T
80 NEXT I

ABC80
█
```

Fig 7.9 Programmet LJUD

Ja nu har vi gjort vårt första demonstrationsprogram. En stor del av mödan fick vi ägna åt presentationen på bildskärmen! Det finns bara ett sätt att lära sig programmering och hantering av utskrifter på bildskärmen: Det är att själv göra enkla program och utskrifter! Börja med att modifiera ett givet program - exempelvis ovanstående - och undersök vad som händer!

I följande program ger vi endast sparsamma kommentarer. Det blir alltså rikliga tillfällen att pröva och se hur olika satser fungerar i efterföljande demonstrationsprogram !

3.2 Bithantering

BASIC är inte ett enhetligt och exakt standardiserat språk. Det förekommer i många varianter (eller dialekter). Det finns några BASIC-varianter som kan hantera bitar, dvs testa en bit i en port och ställa en bit i en port. När ABC80:s BASIC skulle klämmas in i 16K programminne fick man inte plats med dessa bithanteringsrutiner. Innan vi ger oss i kast med tangentbord, V24-snitt och motorstyrning ska vi därför visa att vi mycket väl kan hantera bitar med ABC80:s vanliga BASIC-satser. Fig 7.10 visar tre programexempel.

3.2.1 Manuell rutin

I det första programmet i fig 7.10a beräknas bitarnas värden på samma sätt som om man skulle utföra arbetet med papper och penna.

Rad 10 i programmet fig 7.10 raderar skärmen och rad 20 läser in ett tal (ger variabeln A ett värde) från tangentbordet. Vi antar att $0 \leq A \leq 255$.

Raderna 30 till 170 analyserar det inlästa talet "bit för bit". Vi får på känt sätt motsvarande värden för bitarna b7 till och med bØ. Rad 180 skriver ut bitarnas värden. Här har vi alltså fått fram bitarna ur det givna talet. Dessa bitar kan vi testa separat eftersom de har skilda variabelnamn (B7 till BØ).

Det är enklare att gå den motsatta vägen, dvs bilda ett tal när bitarna är givna. Rad 190 visar på känt sätt hur detta går till. På rad 200 skriver vi ut det av bitvärdena beräknade talet på bildskärmen.

Provkör programmet och övertyga dig om att det stämmer.

3.2.2 Loop-rutinen

När stora tal ska bit-analyseras blir programmet kortare om det skrivs med hjälp av loopar som i fig 7.10b. Den första loopen (30-70) beräknar bitvärden, den andra loopen (80-100) skriver ut bitvärden och den tredje loopen (110-140) syntetiserar ett tal av givna bitvärden.

```
10 ; CHR$(12)
20 INPUT A
30 IF A>=128 THEN B7=1 ELSE B7=0
40 IF B7=1 THEN A=A-128
50 IF A>=64 THEN B6=1 ELSE B6=0
60 IF B6=1 THEN A=A-64
70 IF A>=32 THEN B5=1 ELSE B5=0
80 IF B5=1 THEN A=A-32
90 IF A>=16 THEN B4=1 ELSE B4=0
100 IF B4=1 THEN A=A-16
110 IF A>=8 THEN B3=1 ELSE B3=0
120 IF B3=1 THEN A=A-8
130 IF A>=4 THEN B2=1 ELSE B2=0
140 IF B2=1 THEN A=A-4
150 IF A>=2 THEN B1=1 ELSE B1=0
160 IF B1=1 THEN A=A-2
170 B0=A
180 ; B7;B6;B5;B4;B3;B2;B1;B0;
190 Z=B0*1+B1*2+B2*4+B3*8+B4*16+B5*32+B6*64+B7*128
200 ; "    Z=";Z
210 GOTO 20
```

a.

```
10 ; CHR$(12)
20 INPUT A
30 C=128
40 FOR I=7 TO 0 STEP -1
50 IF A>=C THEN B(I)=1 : A=A-C ELSE B(I)=0
60 C=C/2
70 NEXT I
80 FOR I=7 TO 0 STEP -1
90 ; B(I);
100 NEXT I
110 Z=0
120 FOR I=7 TO 0 STEP -1
130 Z=2*Z+B(I)
140 NEXT I
150 ; "    Z=";Z
160 GOTO 20
```

b.

```
10 ; CHR$(12)
20 INPUT A
30 C=128
40 FOR I=7 TO 0 STEP -1
50 IF A AND C THEN B(I)=1 : ; 1; ELSE B(I)=0 : ; 0;
60 C=C/2
70 NEXT I
80 Z=B(0)*1+B(1)*2+B(2)*4+B(3)*8+B(4)*16+B(5)*32+B(6)*64+B(7)*128
90 ; "    Z=";Z
100 GOTO 20
```

c.

Fig 7.10 Program för bithantering
a. "Manuell" rutin
b. Med loopar
c. Med masker

3.2.3 Mask-rutiner

Den elegantaste bithanteringsmetoden är kanske att använda masker. Det kan utföras på i princip samma sätt som vi tidigare sett i assemblerspråk (exempelvis i avsnitt 4.1 i kap 3). I fig 7.10c börjar vi på rad 30 med masken C=128, dvs bitmönstret

10000000

När bit B(7) analyserats på rad 50 (och printats på skärmen) skiftas bitmasken ett steg till höger med satsen C=C/2. Så upprepas analysen bit för bit. Den manuella bitanalysen och printningen i fig 7.10a upptar 15 rader. De tre looparna i fig 7.10b upptar 12 rader. Maskinrutinen i fig 7.10c upptar enbart 5 rader. Men det är ofta svårare att förstå funktionen hos korta och komprimerade program.

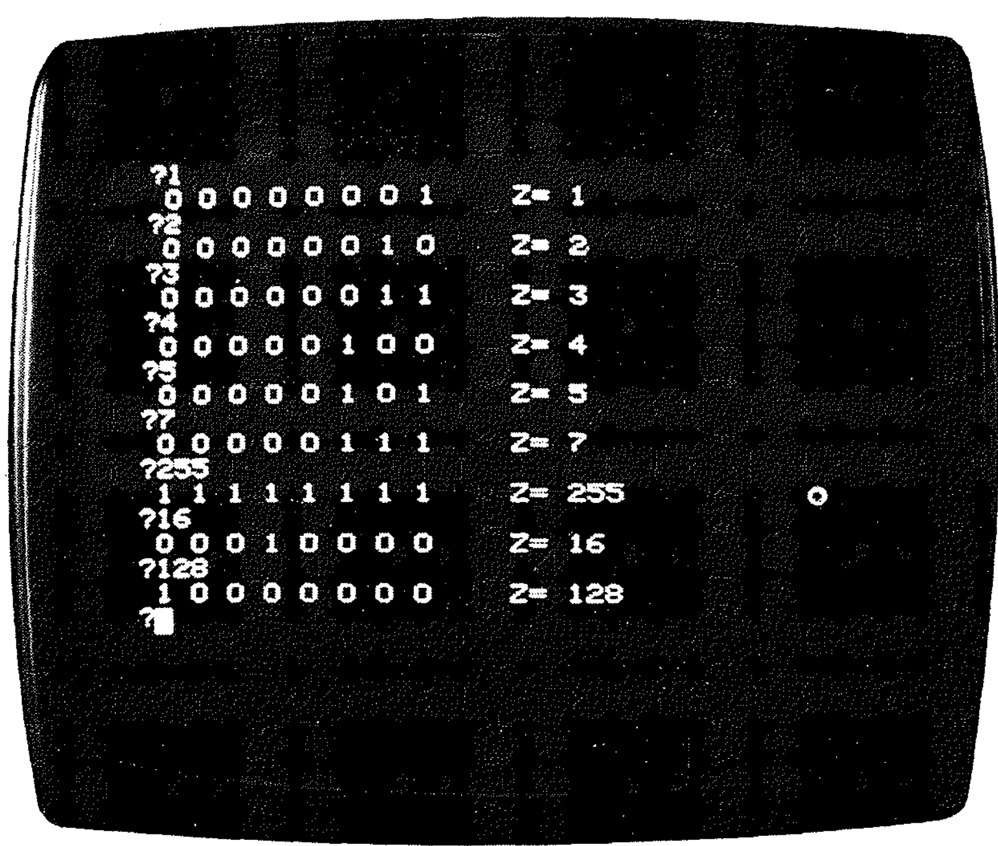


Fig 7.10d Bildskärmens utseende vid exekvering av programmet

3.3 Motorstyrning

Motorn i bandspelaren styrs som vi sett i fig 6.29 av bit 5 i PIO:ns port B. För att kunna styra motorn måste vi alltså kunna ställa bit 5 utan att påverka någon av de övriga bitarna i porten. Fig 7.11a visar ett tänkbart program för motorstyrningen.

Rad 10-80 raderar bildskärmen och skriver ut text på lämpliga ställen med hjälp av CUR-funktion.

PIO:ns port B har decimala adressen 58. På rad 90 läses värdet från port B till variablen A med satsen 90 A = INP (58). Raderna 100-150 analyserar därefter bitarna i variabeln A och skriver ut dessa bitvärden på bildskärmen. Rad 160 placerar markören framför texten "KOMMANDO".

Satsen GET K¤ på rad 170 läser in ASCII-koden för nedtryckt tangent till strängvariablen K¤.

På raderna 180 och 190 utförs bithanteringen. Om g-tangenten tryckts ned får bit b5 värdet 1 och om s-tangenten nedtryckts får bit b5 värdet Ø.

För att inte påverka värdet hos övriga bitar i PIO:n sammanställs på rad 200 ett tal Z som består av de tidigare inlästa bitarna b7, b6, b4 b3, b2, b1 och bØ samt den "manipulerade biten b5.

Med satsen på rad 210 kopieras Z till PIO:ns port B. Därmed kommer bandspelaren att starta eller stoppa enligt den instruktion vi lagt in i bit 5.

Med satsen 220 GOTO 90 är ABC80 redo att ta emot ett nytt kommando (g eller s) på tangentbordet.

Fig 7.11b visar utskriften på bildskärmen.

I fig 7.12 har vi förenklat programmet och ej medtagit utskriften på det kompletta bitmönstret för port B. Styrprogrammet omfattar här endast 4 rader (80-110) om man frånser texten på bildskärmen.

På rad 90 ettställs bit 5 med en mask (32) och OR-funktion på inläst värde. På rad 100 nollställs bit 5 med den komplementära masken (223) och AND-funktion på inläst värde.

3.4 Tangentbordet

Inläsning från tangentbordet sker som vi nämnt tidigare med hjälp av en avbrottsrutin. Låt oss göra ett program som ständigt avsöker PIO:ns port A och som därmed visar vilka koder ABC80 läser in från tangentbordet. Fig 7.13 visar ett exempel.

Raderna 10-70 suddar och skriver text på bildskärmen. Själva programmet börjar på rad 80 med läsning av PIO:ns port A som har decimala adressen 56.

Raderna 90-140 analyserar och skriver ut bitarna i det inlästa talet.

a.

```
10 ; CHR$(12)
20 ; CUR(3,13)"MOTOR-STYRNING"
30 ; CUR(6,10)"PIO Port B: Adress 58"
40 ; CUR(8,10)"Utgång: bit 5"
50 ; CUR(12,5)"Bitnummer: b7"
60 ; CUR(12,30)"b0"
70 ; CUR(17,5)"KOMMANDO: g = gå (start)"
80 ; CUR(18,15)"s = stopp"
90 A=INP(58)
100 C=128
110 ; CUR(13,16);
120 FOR I=7 TO 0 STEP -1
130 IF A AND C THEN B(I)=1 : ; 1; ELSE B(I)=0 : ; 0;
140 C=C/2
150 NEXT I
160 ; CUR(17,4);
170 GET K$
180 IF K$="g" THEN B(5)=1
190 IF K$="s" THEN B(5)=0
200 Z=B(0)*1+B(1)*2+B(2)*4+B(3)*8+B(4)*16+B(5)*32+B(6)*64+B(7)*128
210 OUT 58,Z
220 GOTO 90
```

b.

```
MOTOR-STYRNING

PIO Port B: Adress 58
Utgång: bit 5

Bitnummer: b7              b0
           1 0 0 0 0 1 1 1

KOMMANDO: g = gå (start)
          s = stopp
```

Fig 7.11 a. Program för styrning av bandspelarens motor
b. Bildskärmens utseende vid programmets exekvering

```
10 ; CHR$(12)
20 ; CUR(3,13)"MOTOR-STYRNING"
30 ; CUR(6,10)"PIO Port B: Adress 58"
40 ; CUR(8,10)"Utgång: bit 5"
50 ; CUR(17,5)"KOMMANDO: g = gå (start)"
60 ; CUR(18,15)"s = stopp"
70 ; CUR(17,4);
80 GET K$
90 IF K$="g" THEN OUT 58,32 OR INP(58)
100 IF K$="s" THEN OUT 58,223 AND INP(58)
110 GOTO 80
```

Fig 7.12 Kort program för motorstyrning

```
10 ; CHR$(12)
20 ; CUR(2,12)"TANGENTBORDET"
30 ; CUR(11,3)"Bitnummer: b7"
40 ; CUR(11,28)"b0"
50 ; CUR(4,14)"PIO Port A"
60 ; CUR(6,8)"Inport med adress 56"
70 ; CUR(8,12)"Data in:"
80 A=INP(56)
90 C=128
100 ; CUR(12,14);
110 FOR I=7 TO 0 STEP -1
120 IF A AND C THEN B(I)=1 : ; 1; ELSE B(I)=0 : ; 0;
130 C=C/2
140 NEXT I
150 ; CUR(8,21)A"  " : ; CUR(8,27)CHR$(A AND 127)
160 GOTO 70
```

Fig 7.13 Tangentbordsprogrammet

Med avslutningssatsen 160 GOTO 70 får vi programmet att löpa i en slinga och därmed ständigt göra nya avläsningar.

Ja, nu kan vi kontrollera om ASCII-koderna från tangentbordet stämmer med specifikationen i fig 6.2. Den som inte vill göra manuell omräkning till hexkod uppmanas skriva en rutin för decimal-hex-omvandling.

3.5 Minnet

Innan vi ger oss i kast med videointerfacet och bildskärmen ska vi göra ett program som "dumpar" minnet så att vi kan läsa dess innehåll på bildskärmen. Fig 7.14 ger ett förslag.

```
10 ; CHR$(12)
20 ; "Minnet"
30 ;
40 ; "Startadress?" : INPUT A
50 ; CHR$(12)
60 ; "Minnet: Startadress ";A
70 ;
80 FOR K=0 TO 7
90 ; TAB(K*4+4);K;
100 NEXT K
110 ;
120 ;
130 FOR N=0 TO 15
140 ; N;
150 FOR K=0 TO 7
160 ; TAB(K*4+4);PEEK(N+K*16+A);
170 NEXT K
180 ;
190 NEXT N
```

Fig 7.14 Program för att läsa i minnet

Det finns två viktiga BASIC-satser med vars hjälp man kan läsa eller skriva i önskad adress i minnet.

PEEK, betyder "kika in i" och läser på angiven adress

POKE, betyder "peta" och petar alltså in önskad data på angiven plats i minnet.

Vi ska använda PEEK-satsen och vi ser den på rad 160 i fig 7.14.

När vi skriver ut minnets innehåll får 8 kolumner (80 FOR K=0 TO 7) och 16 rader (130 FOR N=0 TO 15) lagom plats på bildskärmen. Vi beräknar tillhörande adresser och ställer upp en tabell på bildskärmen. Den visar innehållet i 128 byte. Vi kan välja en godtycklig startadress genom att läsa in startadressen A med INPUT-satsen på rad 40.

Vi har sagt det tidigare: En stor del av arbetet med program ligger i att presentera resultatet. I fig 7.14 är det en tabell. Pröva själv med några andra varianter för presentation av minnets innehåll!

I fig 7.14 har vi använt tabulatorsatsen TAB som flyttar markören önskat antal positioner längs en rad. Tidigare använde vi CUR-funktionen som gav oss möjlighet att välja både rad och kolumn för markören.

Vi har tidigare i fig 6.17 sett hur bildminnet avsöks som en sekvens av 16 rutor vardera innehållande 8x8 tecken. Fig 7.15a visar ett program som demonstrerar både användningen av POKE-satsen och scanningen av bildminnet.

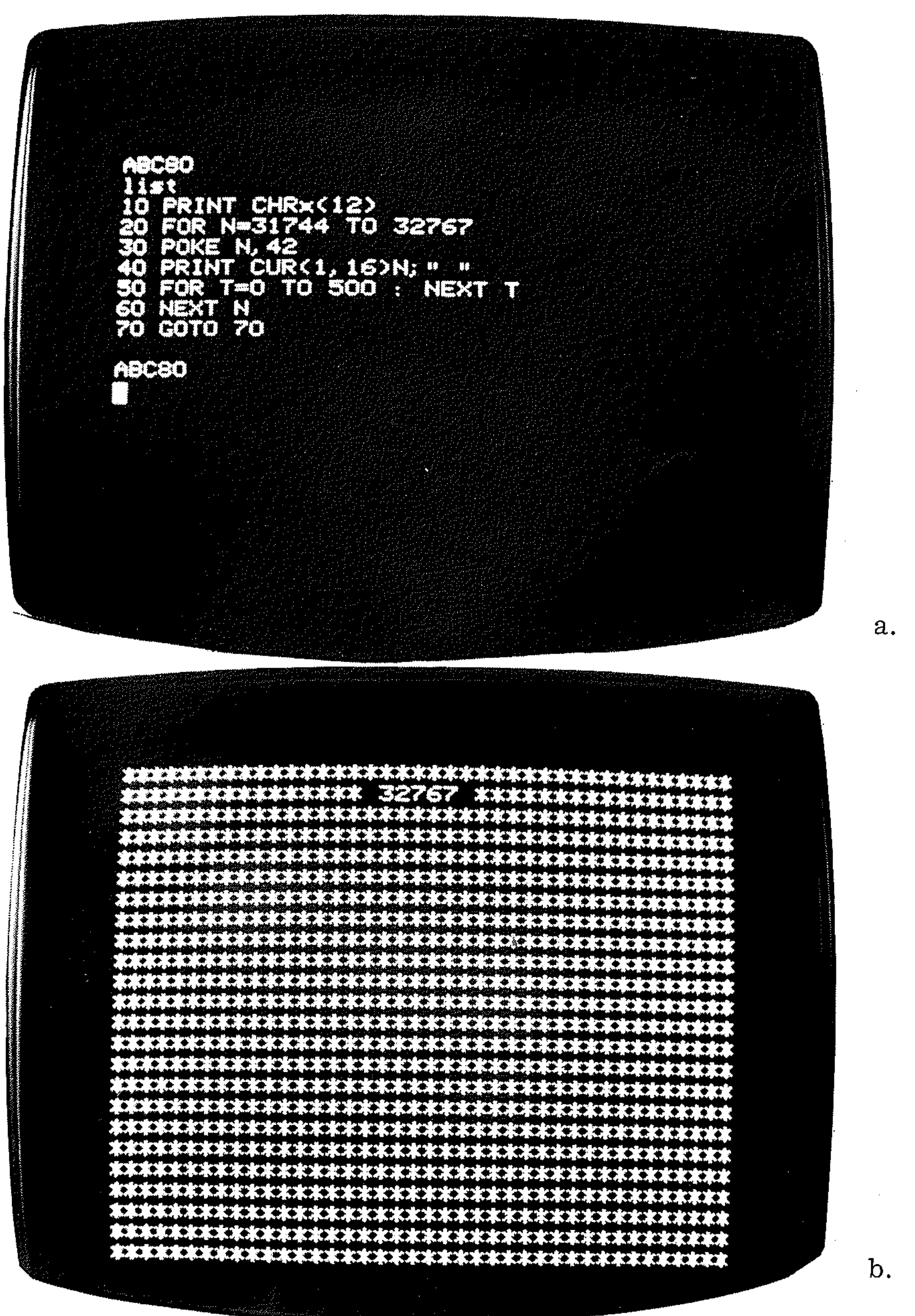


Fig 7.15 a. Program för scanning av bildminnet
b. Foto av resultatet

Bildminnet innefattar området mellan (och inklusive) de decimala adresserna 31744 och 32767. Rad 20 och 60 bildar en räknare som successivt lägger ut dessa adresser.

För att vi bättre ska kunna följa scanningen i bildminnet laddar vi successivt dess minnesceller med stjärnor (ASCII-koden 42) med hjälp av POKE-satsen på rad 30.

För att vi ska veta aktuell minnesadress skriver vi ut minnesadressen med PRINT-satsen på rad 40.

Rad 50 ger tidsfördröjning så att vi hinner följa förloppet och läsa adresserna.

Fig 7.15b visar bildskärmens utseende när programmet nått rad 70 och gått in i den ändlösa loopen 70 GOTO 70.

Det kan nu vara av intresse att köra minnesprogrammet i fig 7.14 och kontrollera vilken ASCII-kod som används för mörka rutor. Bildminnet upptar adresserna 31744 till 32767. Pröva själv!

3.7 V24-snittet

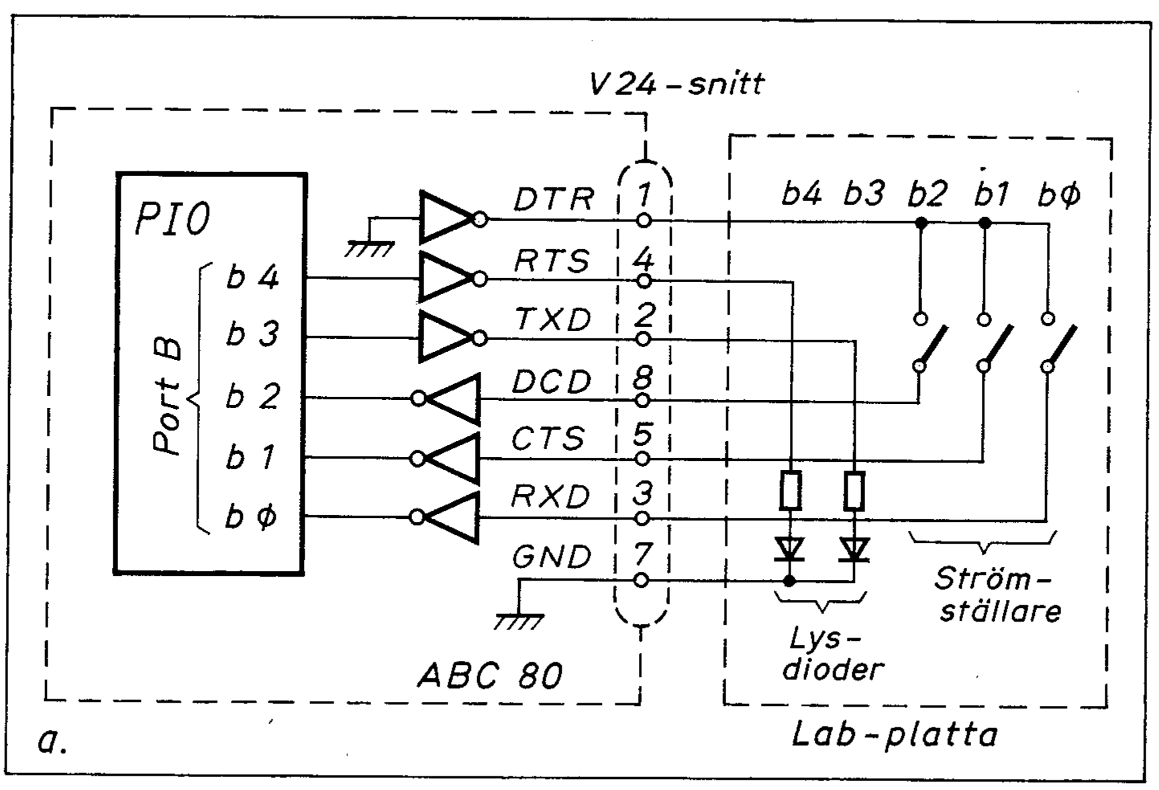
Vi har tidigare (i avsnitt 5.3.3 i kap 5) bekantat oss med V24-snittet och dess niopoliga uttag baktill på ABC80. Uttaget är avsett för anslutning av kommunikationsutrustning men kan givetvis även användas för andra ändamål.

Fig 7.16 visar hur en "labplatta" kan kopplas in till V24-uttaget och vi ska strax göra ett program med vars hjälp vi kan demonstrera funktionen.

Utgångarna från V24-snittet tillförs två lysdioder (via lämpliga seriemotstånd). En lysdiod kräver vanligen 10 mA framström för att lysa någorlunda och 10 mA är just den ström drivstegen (MC1488) kan ge. Eftersom drivstegen är inverterande betyder en "etta" från ABC80 låg utspänning på motsvarande utgång.

Även ingångarna har inverterande kretsar (MC1489) och när inga ledningar är anslutna får vi binära ettor till ABC80. För att läsa in nollor måste vi tydligen lägga på hög inspänning (5 - 10 V) och denna kan vi ta från ben nr 1 i V24-kontakten. Här ligger utgången på en drivkrets som har jordad ingång. Utgången ger ca 9 V vid max 10 mA belastning. Med tre switchar kan vi nu styra insignalerna på inportarna bØ, b1 och b2 till PIO:n.

Fig 7.16b visar ett foto av en enkel labplatta. Nu återstår bara ett program för att få "liv" i labplattan. Låt oss göra ett program som avläser switcharna (ingångarna) och kör ut motsvarande signaler till de två lysdioder som är inkopplade till utgångarna.



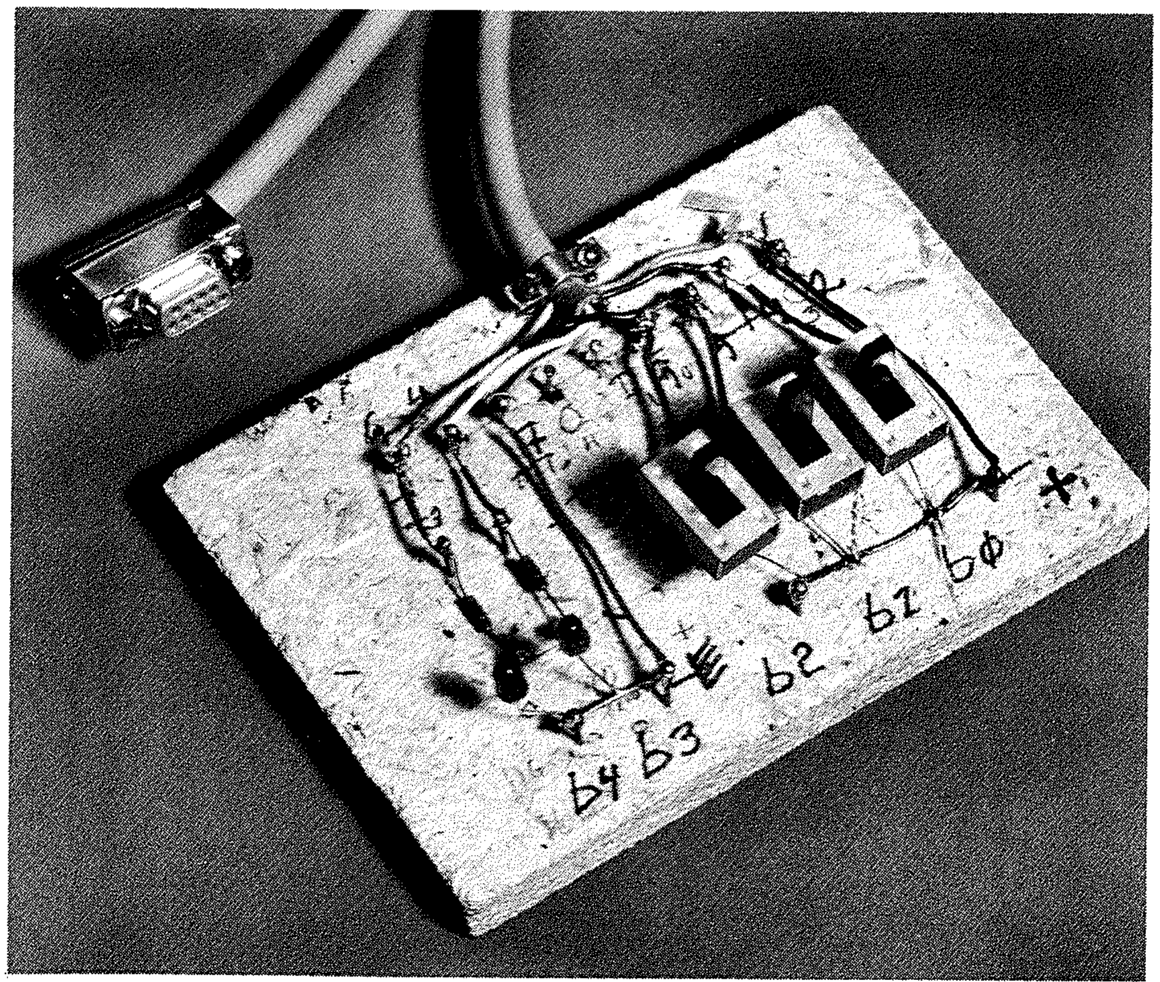


Fig 7.16 V24-snittet
a. Labplattans schema
b. Foto

```
list
10 ; CHR¤(12)
20 ; CUR(3,13)"V24-SNITTET"
30 ; CUR(6,10)"PIO Port B: Adress 58"
40 ; CUR(7,10)"Ingångar: bit 0,1 och 2"
50 ; CUR(8,10)"Utgångar: bit 3 och 4"
60 ; CUR(10,10)"Data:"
70 ; CUR(12,5)"Bitnummer: b7"
80 ; CUR(12,30)"b0"
90 A=INP(58)
100 ; CUR(10,16)A
110 ; CUR(13,16);
120 C=128
130 FOR I=7 TO 0 STEP -1
140 IF A AND C THEN B(I)=1 : ; 1; ELSE B
(I)=0 : ; 0;
150 C=C/2
160 NEXT I
170 Z=B(0)*8+B(1)*16
180 OUT 58,Z
190 GOTO 90

ABC80
█
```

a.

```
              V24-SNITTET

          PIO Port B: Adress 58
          Ingångar: bit 0,1 och 2
          Utgångar: bit 3 och 4

          Data:   159

     Bitnummer: b7              b0
                1 0 0 1 1 1 1 1
```

b.

Fig 7.17 a. Program för styrning av lysdioderna på labplattan i fig 7.16

b. Programmet exekveras

Fig 7.17a visar ett förslag till program. Raderna 10-80 ger förklarande text på bildskärmen. Rad 90 läser inport B. Rad 100-160 analyserar bitarnas innehåll och visar ett bitmönster på skärmen. På rad 170 beräknas det tal som ger utgångarna B(3) och B(4) önskade bitvärden. (Adderar man här talet 32 blir bit 5 hög och bandspelarmotorn startar).

Rad 180 ställer utgångarna B(3) och B(4) till de bitvärden som erhålls från ingångarna B(Ø) och B(1).

Nu är det bara att starta programmet med RUN. Studera vad som händer på skärmen och i lysdioderna när vi ställer switcharna på labplattan i olika lägen!

V24-utgången är den minst känsliga utgången på ABC80 och därför mycket lämplig för olika labexperiment.

3.8 Effektstyrning

I fig 7.16 styrde utgångarna i V24-snittet två lysdioder (ca 10 mA). I många fall vill vi styra större effekter och fig 7.18 visar ett exempel.

Här är två statiska relän inkopplade till utgångarna på V24-kontakten.

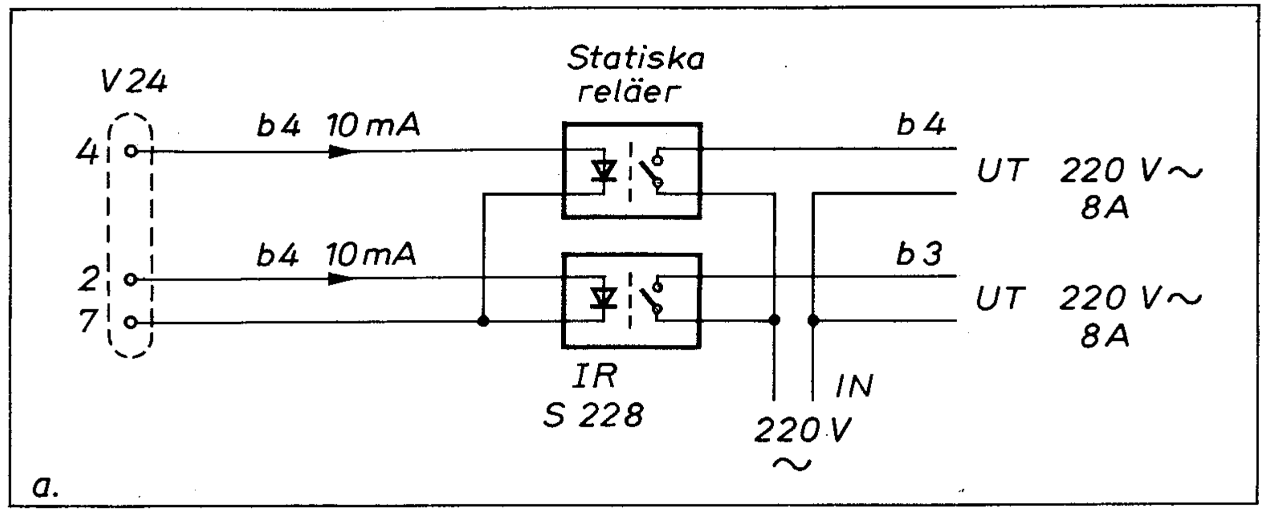
Ett statiskt relä innehåller inga rörliga delar (därav benämningen statisk). Det innehåller istället tyristorswitchar.

De statiska reläerna i fig 7.18 har inbyggda optokopplare. Det betyder att ingångarna utgörs av lysdioder precis som på vår labplatta i fig 7.16. Ljuset från dessa lysdioder styr de inbyggda tyristorerna. Man får på detta sätt ingångarna helt isolerade från nätspänningen. De statiska reläerna i fig 7.18 switchar vid nätspänningens nollgenomgångar och är därför relativt störningsfria.

Fig 7.19 visar ett program som är så utformat att utgångarna kan inkopplas (tändas) eller urkopplas (släckas) med följande kommandon på tangentbordet

3 = tänd på B3
# = släck på B3
4 = tänd på B4
¤ = släck på B4

Programmet visar ett exempel på bithantering där man först läser in (rad 110) och sedan analyserar och skriver ut bitmönstret (rad 120-170). Därefter ställer man bitar i enlighet med givna kommandon (rad 190-230) samt beräknar och kör ut motsvarande talvärde till utporten (rad 240-250).



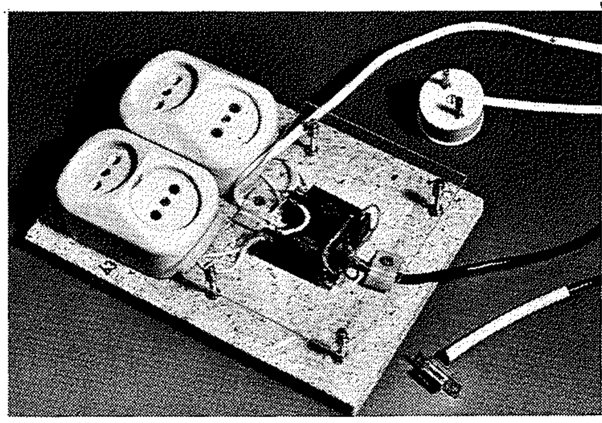


Fig 7.18 a. Statiska reläer inkopplade till V24-utgångarna
b. Foto på labplatta

```
10 ; CHR$(12)
20 ; CUR(3,13)"V24-RELÄ"
30 ; CUR(6,10)"PIO Port B: Adress 58"
40 ; CUR(8,10)"Utgångar: bit 3 och 4"
50 ; CUR(12,5)"Bitnummer: b7"
60 ; CUR(12,30)"b0"
70 ; CUR(17,5)"KOMMANDO: 3=TÄND PÅ BIT 3"
80 ; CUR(18,15)"#=SLÄCK PÅ BIT 3"
90 ; CUR(19,15)"4=TÄND PÅ BIT 4"
100 ; CUR(20,15)"$=SLÄCK PÅ BIT 4"
110 A=INP(58)
120 ; CUR(13,16);
130 C=128
140 FOR I=7 TO 0 STEP -1
150 IF A AND C THEN B(I)=1 : ; 1; ELSE B(I)=0 : ; 0;
160 C=C/2
170 NEXT I
180 ; CUR(17,4);
190 GET K$
200 IF K$="#" THEN B(3)=1
210 IF K$="$" THEN B(4)=1
220 IF K$="3" THEN B(3)=0
230 IF K$="4" THEN B(4)=0
240 Z=B(3)*8+B(4)*16
250 OUT 58,Z
260 GOTO 110
```

Fig 7.19 Program för styrning av reläerna i fig 7.18

4. ABC80:s programvara

ABC80:s programvara är mycket intressant och skulle kunna fylla flera läroböcker med stoff för intressanta detaljstudier. Vi kan här endast ge en lätt provsmakning. Vi ska köra ett "Benchmark"-program, vi ska se hur ABC80 lagrar sina flyttalsvariabler och vi ska kort bekanta oss med slumptalsgeneratorn.

4.1 Benchmark

Ordet benchmark hör man ofta i datorsammanhang. "Benchmark" kallades (i USA) de märken som snickaren skar in i sin hyvelbänk för att ha vissa vanliga mått snabbt tillgängliga.

```
10 ; CHR$(12)
20 ; "Benchmark Program 7"
30 ; "från Kilobaud, #10 1977"
40 ; "Starta på s och ta tiden!"
50 GET S$
60 IF S$="s" GOTO 80
70 GOTO 50
80 PRINT "START"
90 K=0
100 DIM M(5)
110 K=K+1
120 A=K/2*3+4-5
130 GOSUB 310
140 FOR L=1 TO 5
150 M(L)=A
160 NEXT L
170 IF K<1000 THEN 110
180 PRINT "END"
190 ; "   Jämförelsetider:"
200 ; "   LSI-11:              23s"
210 ; "   Zapple Basic:        33s"
220 ; "   OSI (1 MHz klocka): 42s"
230 ; "   PET Basic:           51s"
240 ; "   Altair (8K, ver 4): 52s"
250 ; "   Compal:              66s"
260 ; "   Wang 2200S:          75s"
270 ; "   Micropolis:         146s"
280 ; "   IBM 5100:           172s"
290 ; "   Imsai:              235s"
300 GOTO 320
310 RETURN
320 END
```

Fig 7.20 Benchmark-programmet från tidskriften Kilobaud

Om man har en arbetsuppgift (t ex trafikstyrning) som ska lösas med datorkraft kan man skriva ett för arbetsuppgiften typiskt program och sedan köra detta program på datorer av olika fabrikat. Det visar sig då att vissa datorer löser den aktuella uppgiften snabbare eller kräver mindre minne. Ett testprogram av denna typ brukar kallas "benchmark program" eller kortare "benchmark".

Tidskriften Kilobaud jämförde olika persondatorer med ett "benchmark" i oktobernumret 1977. Programmet återfinns på raderna 80-180 samt 280 i fig 7.20.

Valet av benchmark kommenteras utförligt i tidskriften Kilobaud. (Givetvis har Kilobauds benchmark utsatts för hård kritik av de fabrikanter som hamnat långt ner på ranglistan och som menar att andra benchmark-program hade varit mera rättvisande. Givetvis är tiden enbart en av de många faktorer som avgör hur användbart ett smådatorsystem är i praktiken).

Kör programmet i fig 7.20! ABC80 ligger i toppen med ca 23 s exekveringstid.

Benchmark-program är en "het potatis" och vi ska därför inte diskutera dem vidare här.

4.2 ABC80:s grafik

Vi har i fig 6.21 sett hur ABC80 kan tolka en sjubitars kod på två sätt, som ASCII-tecken eller som grafer. Fig 7.21 visar ett program som ritar fig 6.21 på ABC80:s bildskärm.

R anger rad och C ger decimala koderna för första radens tecken.

Kanske verkar satsen PRINT CHR ¤ (135, R+C) förbryllande. För övre radens första tecken får R värdet Ø och C värdet 16. Därmed får satsen utseendet

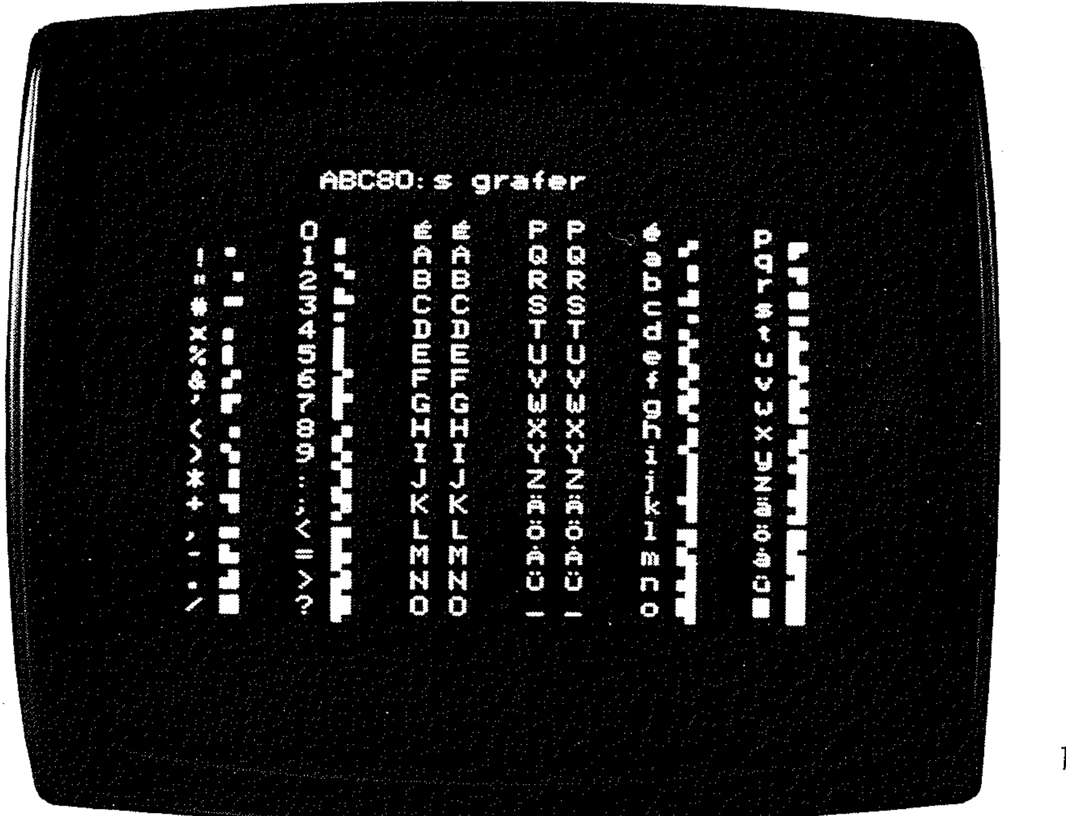
```
PRINT CHR ¤ (135, 16)
```

Detta tolkas av ABC80 på samma sätt som om vi hade skrivit

```
PRINT CHR ¤ (135); CHR ¤ (16).
```

```
10 PRINT CHR$(12)CUR(2,10)"ABC80:s grafer"
20 PRINT
30 FOR R=0 TO 15
40 PRINT "  ";
50 FOR C=32 TO 112 STEP 16
60 PRINT CHR$(135,R+C);
70 PRINT CHR$(151,R+C);
80 PRINT "  ";
90 NEXT C
100 PRINT
110 NEXT R
120 PRINT
140 GOTO 140
```

a.



b.

Fig 7.21 a. Program som tecknar fig 6.21 på bildskärmen
b. Programmet exekverat

4.3 Flyttalsvariabler

När vi övergår från assembler-språk till högnivåspråk avlastas vi arbetet att hålla reda på var alla variabler ska placeras i minnet.

För att visa hur ABC80 lagrar sina flyttal och var de lagras kan vi köra programmet i fig 7.22.

```
10 PRINT CHR$(12)
20 PRINT "FLYTTAL"
30 INPUT Y
40 A%=-471%
50 A%=PEEK(A%)+SWAP%(PEEK(A%+1%))
60 IF A%=0% THEN 200
70 PRINT CHR$(PEEK(A%+1%));": ";
80 PRINT ; : PRINT INT(A%)+2^16%" :";
90 FOR I%=0% TO 4%
100 PRINT TAB(I%*3%+9%);
110 V%=PEEK(A%+I%+4%)
120 V1%=V%/16%
130 FOR J%=1% TO 2%
140 IF V1%<10% THEN PRINT CHR$(V1%+48%); ELSE PRINT CHR$(V1%+55%);
150 V1%=V% AND 15%
160 NEXT J%
170 NEXT I%
180 PRINT
190 GOTO 30
200 END
```

Fig 7.22 Program som visar lagringsprincipen för flyttal i ABC80

Som svar på ABC80:s inputbegäran (dvs frågetecknet på skärmen) kan man slå in olika flyttal (avslutade med CR). Man får som svar variabelns namn (rad 70), variabelns lagringsadress (rad 80) samt innehållet i de fem byte som lagrar variablens värde (rad 90-170).

Rad 40 anger variableroten, dvs den pekare som BASIC-tolken använder för att hitta fram till variablernas lagringsplatser.

Variablerna lagras intill programmet. Om man skjuter in en extra sats (exempelvis 185 PRINT "NÄSTA TAL") kommer lagringsadressen att förskjutas. Man kan på detta sätt undersöka hur stort lagringsutrymme som olika satser kräver. Här finns plats för mycket experimenterande!

Utrymmet tillåter tyvärr inte en noggrann genomgång av alla detaljer i programmet i fig 7.22.

4.4 Slumptalsgeneratorn

Slumptalsgeneratorer är utomordentligt viktiga i många tillämpningar (även om vi bortser från spel av olika slag). I datorer bildas ofta slumptal genom att man genererar en serie som ständigt ger nya talvärden. ABC80 har en slumptalsgenerator av denna typ.

Räkning med slumptal är ett fascinerande område. Lennart Råde på Chalmers har skrivit boken "Att chansa med räknedosan" (Studentlitteratur 1977). Det går ännu bättre att chansa med ABC80 eftersom man här har bättre möjligheter att presentera resultatet på ett överskådligt sätt.

Skriv PRINT RND på tangentbordet så svarar ABC80 med ett slumptal vars värde ligger mellan noll och ett. Gör om försöket så kommer ett nytt slumptal från ABC80:s slumptalsgenerator.

Slumptalsgeneratorn i ABC80 är skriven i assembler och arbetar med 35 bitars ordlängd. För att ge en antydan om dess uppbyggnad visar fig 7.23 ett program för en liknande slumptalsgenerering med en BASIC-rutin. Slumptalen från denna rutin jämförs med ABC80:s slumptal.

I slumptalsgeneratorn spelar tre konstanter A, B och C en viktig roll.

```
10 PRINT CHR$(12)
20 ; "  SLUMPTALSGENERATORER"
30 ;
40 ; " ABC80:    BASIC-rutin:"
50 A$="34359738368"
60 B$="1061851031117"
70 C$="2718281829"
80 X$="0"
90 FOR I=1 TO 18
100 ; RND,
110 X$=MUL$(X$,B$,0)
120 X$=ADD$(X$,C$,0)
130 Y$=DIV$(X$,A$,0)
140 Z$=MUL$(Y$,A$,0)
150 X$=SUB$(X$,Z$,0)
160 IF INSTR(1,X$,"-") THEN X$=ADD$(X$,A$,0)
170 PRINT VAL(X$)/2^35%
180 NEXT I
190 GOTO 190
```

Fig 7.23 En slumptalsgenerator

Talet $A = 2^{35}$ medan B och C är valda så att talen från slumptalsgeneratorn inte ska upprepas (eller rättare att slumptalsserien ska få en lång cykel).

(Merläsning i tidskriften Interface Age, Februari 1977 sid 96-100).

BASIC-rutinen på rad 110-150 består av "de fyra räknesätten" och efter varje operation stryks samtliga decimaler. Om rutinen ger negativt resultat korrigeras detta på rad 160. På rad 170 "normeras" slumptalet till storlek så att det utskrivna slumptalet kommer att ligga mellan noll och ett.

För att kunna utföra beräkningarna med nödvändigt antal siffror används sträng-aritmetik. Detta beskrivs utförligt i BASIC-manualen och i kursboken ABC om BASIC.

När programmet exekveras märker man att slumptalsgenereringen i BASIC tar avsevärt längre tid än den inbyggda slumptalsgenereringen i assembler.

5. ABC-bussens programmering

För att ge en konkret beskrivning av ABC-bussens programmering ska vi köra ett konkret exempel. Vi väljer trafikstyrningen och använder för detta ändamål Databoard-kortet 4005. Detta IO-kort innehåller 16 utgångar och åtta ingångar. Fig 7.24a visar inkoppling av switchar (sensorer) och lampor (trafikljus).

I praktiken används ofta "aktiv låg" för in- och utsignaler. På Databoard-kortets ingångar kan motstånd lödas in antingen till V_{CC} eller jord. Vi väljer att ansluta dessa ingångsmotstånd till $+V_{CC}$. De fungerar då som pull-up motstånd och insignalen "aktiv låg" kan då erhållas från en enkel switch till jord.

Utgångarna är av typ öppen kollektor och kan därmed enbart sänka utspänningen. Även utsignalen blir därmed aktiv låg.

Om vi denna gång vill ha starkare lampor kan vi koppla in två transistorer mellan varje utgång och lampa, på så sätt som visas i fig 7.24a.

IO-kortet specificeras enligt fig 7.24b. Vi byglar IO-kortet till adress Ø. För att få kontakt med IO-kortet måste vi börja med ett kortval, dvs BASIC-satsen

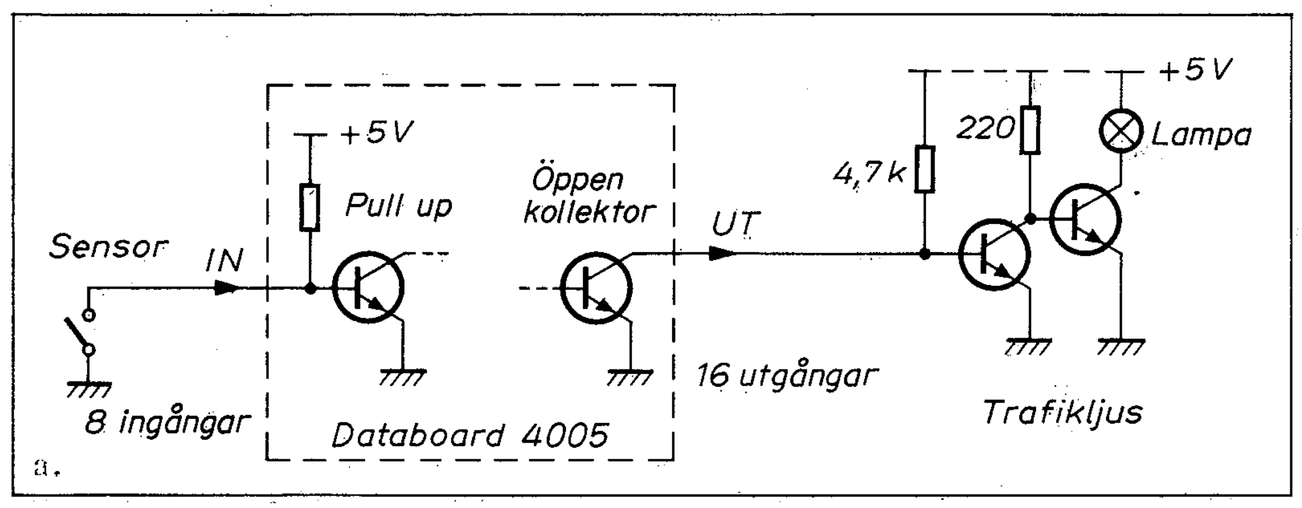
```
OUT 1,0
```

Inläsningen kan sedan göras med BASIC-satsen

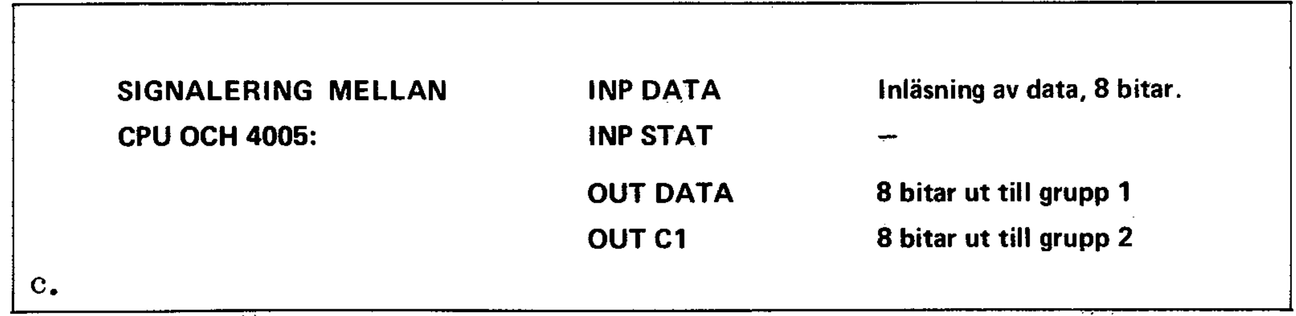
```
INP(0)
```

Som framgår av specifikationen sänder man data till utgångarna 1-8 med OUT DATA, dvs BASIC-satsen

```
OUT 0, DATA
```



8 DATA
CS
CSS
CSB
6
CARD SELECT
BUS INTERFACE
INP
OUT
C1
RST
CONTROL
LATCHED DRIVER
TERMINATED INPUT
1
16
1
8
I/O CONNECTOR
b.



SIGNALERING MELLAN CPU OCH 4005:

INP DATA	Inläsning av data, 8 bitar.
INP STAT	–
OUT DATA	8 bitar ut till grupp 1
OUT C1	8 bitar ut till grupp 2

c.

Fig 7.24 Inkoppling av kretsar till IO-kort för trafikstyrning

För vårt trafikljus behövs endast 6 utgångar. Skulle vi behöva fler än 8 utgångar kan utgångarna 9-16 nås med OUT C1 vilket motsvarar BASIC-satsen

OUT 2, DATA

Eftersom vi nu använder aktiv låg får vi se upp med utsignaler och insignaler. Om vi inverterar alla bitar i fig 3.21 får vi de decimala värden på ljusbilderna som framgår av fig 7.25.

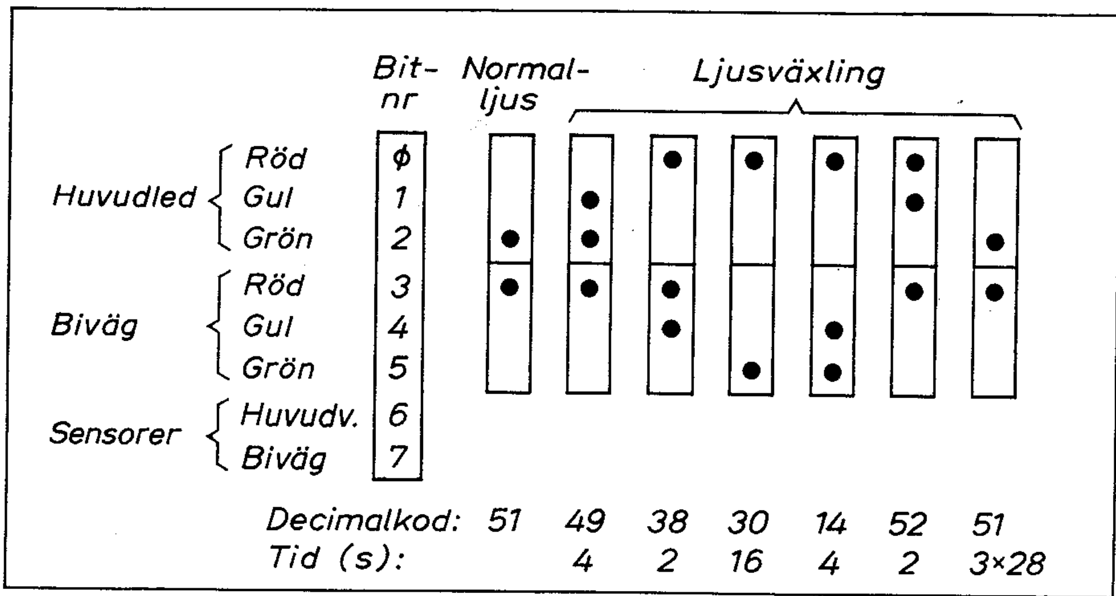


Fig 7.25 Koder för ljusbilder i trafikljus

```
10 ; CHR$(12)
20 ; "TRAFIKLJUS"
30 OUT 1,0 : REM kanalval
40 OUT 0,51 : REM normalljus
50 A%=128 AND INP(0) : ; CUR(1,25)A%" "
60 IF A%=128 THEN 50 ELSE 70
70 OUT 0,49
80 FOR I=0 TO 4000 : NEXT I
90 OUT 0,38
100 FOR I=0 TO 2000 : NEXT I
110 OUT 0,30
120 FOR I=0 TO 16000 : NEXT I
130 OUT 0,14
140 FOR I=0 TO 4000 : NEXT I
150 OUT 0,52
160 FOR I=0 TO 2000 : NEXT I
170 OUT 0,51
180 FOR I=0 TO 84000 : NEXT I
190 GOTO 40
```

Fig 7.26 BASIC-program för trafikstyrning

Ett förslag till BASIC-program framgår av fig 7.26. Ljusväxlingen sker på rad 70-180 och tidsfördröjningen för varje ljusbild är utförd som en enkel loop. Satsen FOR I=Ø TO 1000: NEXT I tar ca 1 s att exekvera.

Rad 30 är väsentlig! Här väljer vi ut IO-kortet, dvs gör kanalvalet med satsen OUT 1, Ø. På rad 40 ställer vi normalljus för att starta med ett känt utgångsvärde.

På rad 50 sker inläsningen av värdet från sensorn i bivägen, dvs bit 7. Vi använder här AND-masken A% = 128 som enbart släpper fram bit 7. För att se vad som händer skriver vi dessutom ut inläst värde på bildskärmen.

På rad 60 testar vi den inlästa biten. Är den hög (dvs A% = 128) går vi tillbaks och gör ny inläsning av sensorn på rad 50. Är bit 7 låg (ELSE 70) exekveras ljusväxlingen.

Ett trafikljusprogram i BASIC är verkligen enkelt och programmeringen av ABC-bussen är som vi sett mycket bekväm.

6. Programexempel

Vi ska avsluta denna bok med att visa exempel på två viktiga användningsområden för ABC80-systemet. Det ena exemplet utnyttjar den inbyggda "realtidsklockan" och det andra är en "simulering" av trafiken i en vägkorsning. Vi kommer inte att gå igenom programmen i detalj eftersom vi här vill diskutera användningsprinciper och inte programmeringsdetaljer.

6.1 Realtidsklockan i ABC80

En detalj i ABC80:s schema har vi tidigare inte omnämnt: Från 312-räknaren i fig 6.13 får vi 50 Hz bildväxlingspulser. dvs en puls var 20:e ms. Bildväxlingspulserna i ABC80 är kopplade till CPU:ns NMI-ingång (NMI = non maskable interrupt). I fig 5.23 har detta utelämnats för att inte figuren ska bli onödigt komplicerad. En avbrottsbegäran på NMI-ingången kan enligt avsnitt 5.1.1 ej hindras (maskeras).

Vid NMI-avbrott hoppar Z80 till en avbrottsrutin för "uppdatering" av realtidsklockan. Denna rutin måste vara så kort att inte exekveringen av övriga program störs. ABC80 innehåller ju exempelvis program som känner tiden vid in- och avspelning från kassett. Avbrottsrutinen måste alltså vara så kort att avbrottstiden är försumbar jämfört med tidsmarginalerna för ABC80:s övriga programhantering.

Realtidsklockan i ABC80 lagrar tiden som ett 24-bitars binärt heltal på adress 65008, 65009 och 65010. Det är lätt att undersöka innehållet i dessa tre byte. Pröva med följande programsnutte:

```
10 ; CHR$(12)
20 ; CUR(3,10)"Realtidsklockan"
30 ; CUR(8,10)PEEK(65010%)" "
40 ; CUR(8,15)PEEK(65009%)" "
50 ; CUR(8,20)PEEK(65008%)" "
60 GOTO 30
```

Realtidsklockan är en tidräknare och dess innehåll dekrementeras var 20 ms med avbrottsrutinen för NMI. Man kanske tycker att det hade varit enklare om klockan inkremerats, men för att optimera avbrottsrutinen för minsta möjliga avbrottstid har man valt dekrementering (med instruktionen DJNZ som inte ändrar innehållet i CPU:ns flaggregister).

Som användare kan man nu utnyttja denna 20 ms-räknare helt efter egna önskemål. I ABC80:s bruksanvisning finns ett program med vars hjälp man dels kan ställa räknaren i ett önskat utgångsvärde (real tid) och dels avläsa tiden. Vi återger programmet i fig 7.27.

Vill man använda ABC80 som klocka behövs större siffror på bildskärmen. Programmet i appendix B utnyttjar grafiken i ABC80 och visar tiden med fyra stora siffror av 7-segmentstyp på ABC80:s bildskärm. Fig 7.28.

Givetvis kan man använda realtidsklockan för att vid önskade tidpunkter anropa program som exempelvis styr de effektswitchar vi diskuterat i anslutning till fig 7.18. Det skulle på detta sätt vara möjligt att exempelvis köra ABC80 som nattklocka med stort lysande display och ha en programrutin som sätter på morgonkaffet 10 minuter innan den tänder lyset och sätter på radion. Om ABC80 inte fått svar efter 5 minuter skulle den kunna starta ljudgeneratorn med lämplig väckningssiren.

Viktigare tillämpningar enligt samma princip är mätvärdesinsamling vid givna tidpunkter eller processtyrning med realtidsklockan som en av givarna.

För att illustrera programmeringstekniken har vi lagt in två rutiner i 7-segmentprogrammet i appendix B. Med den ena rutinen ställer vi in önskad väckningstid (rad 20-60) och med den andra rutinen (rad 200-210) ger ABC80 alarm på den inställda väckningstiden.

```
10 T1%=65008% : ; CHR$(12) : ; : ;
20 PRINT "Vill du ställa klockan? (j/n)"; : GET A$ : ; A$
30 IF A$="j" THEN GOSUB 100
40 PRINT CHR$(12)
50 GOSUB 170
60 PRINT CUR(10,12);"Klockan är nu:";CUR(12,13);
70 ; RIGHT$(NUM$(100+H%),3);":";RIGHT$(NUM$(100+M%),3);
80 PRINT ":";RIGHT$(NUM$(100+S%),3)
90 GOTO 50
100 PRINT "hh,mm,ss"; : INPUT H%,M%,S%
110 Z=H%*3600+M%*60+S%
120 Z1%=Z*50/256
130 Z%= NOT (50*(Z-Z1%/50*256))
140 Z1%= NOT Z1%
150 POKE 65008%,Z%,Z1%,SWAP%(Z1%)
160 RETURN
170 D%=0
180 IF (PEEK(T1%) AND 4%)=0 THEN 170
190 FOR I%=0% TO 2%
200 Z%(I%)=255% XOR PEEK(T1%+I%)
210 NEXT I%
220 Z=((Z%(2)*256)+Z%(1))*5.12+Z%(0)/50
230 IF Z>86400 THEN Z=Z-86400 : GOTO 230
240 H%=Z/3600 : Z=Z-3600*H%
250 M%=Z/60 : S%=Z-60%*M%
260 IF D<>0 THEN GOSUB 110
270 RETURN
```

Fig 7.27 Program för att ställa och läsa realtidsklockan i ABC80

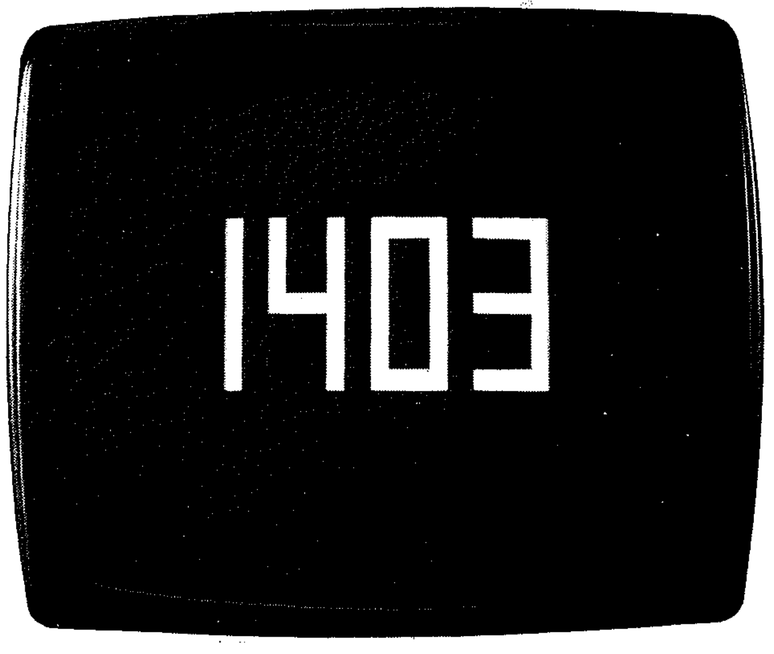


Fig 7.28 Bildskärmens utseende vid exekvering av programmet i appendix B

6.2 Simulering

Simulering (från latinska ordet simulare, efterbilda, låtsa) är ett viktigt tillämpningsområde för datorer. Månfärderna simulerades på datorsystem och därigenom kunde besättningarna tränas i förväg.

Trafikflygbolagen har stora simulatorer där piloterna får träna alla moment under normala trafikflygningar. Man kan emellertid även simulera eld i motorer och andra moment som knappast kan övas under verkliga förhållanden.

Det finns en rad simuleringsprogram för hobbydatorer. Med dessa program kan olika delar av en månlandning simuleras och resultatet av eventuella kraschlandningar åskådliggörs på olika sätt. Givetvis kan en rad spel simuleras. Det finns exempelvis ett luffarschackspel för ABC80 där vi kan spela med ABC80 som motståndare och simulera det rutade papperet med bildskärmen.

Vi har tidigare sett två varianter av trafikljusstyrning. Den ena utfördes med ett assemblerprogrammerat datorkort (fig 4.8) och den andra med ett BASIC-programmerat IO-kort anslutet till ABC-bussen (fig 7.26). Vi använde i båda fallen modeller med små signallampor och switchar (istället för sensorer) för att simulera vägkorsningen.

Vi kan nu gå ett steg längre och simulera hela vägkorsningen på ABC80:s bildskärm. Vi kan dessutom låta slumpgeneratorn styra trafikströmmen.

Fig 7.29 visar grafiken för den vägkorsning vi använder. Vi ser en bil som väntar på bivägen och en som passerar på huvudvägen. Eftersom bildskärmen inte har färg skriver vi ut signalljusens färger på bildskärmen.

Appendix C visar ett tänkbart program. Med hjälp av simuleringen av detta slag kan man gå vidare och studera vilka trafikproblem som uppträder vid olika ljussignaler i komplexa trafikstyrningssystem. Nu har vi kommit så långt bort från elektroniken i ABC80 att det är dags att sätta punkt. Ovanstående simuleringsexempel är enbart avsett som en aptitretare till eget experimenterande med ABC80 och dess många kringkretsar.

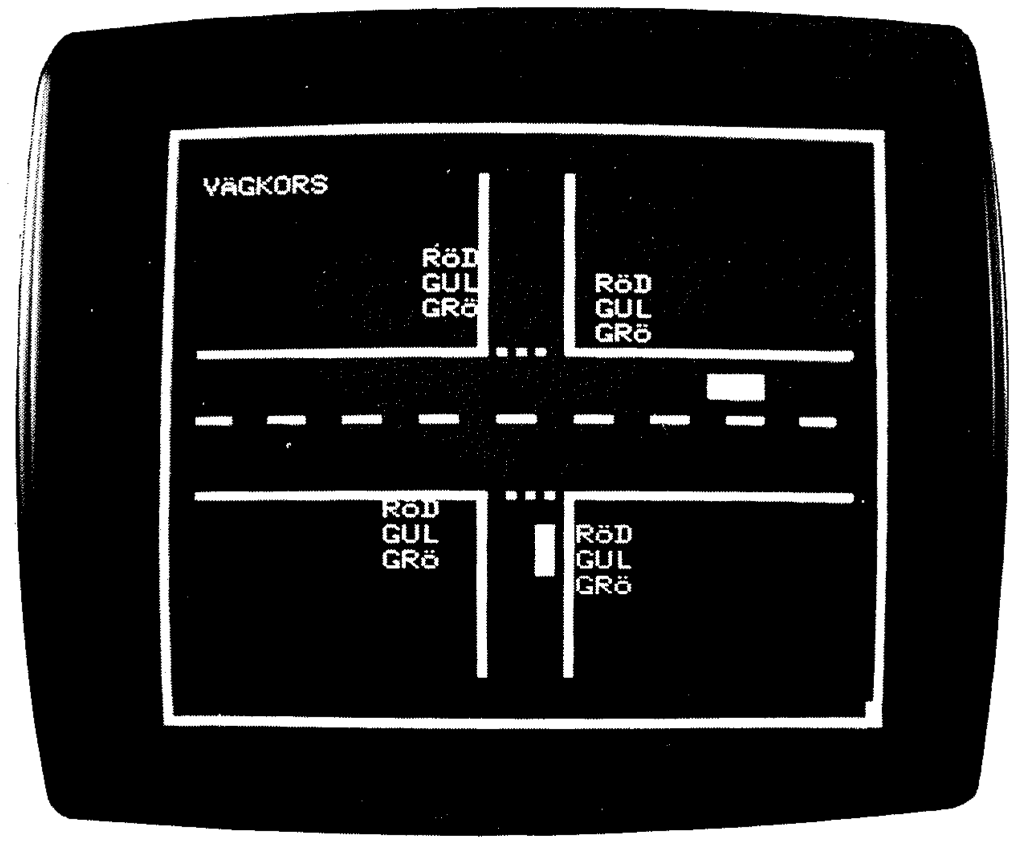


Fig 7.29 Grafiken för simulering av trafiken i vägkorsning

Sammanfattning

Det är programvaran som bestämmer ett systems funktion. Vi har översiktligt beskrivit initieringen och BASIC-tolkens funktion i ABC80. För att mer konkret se hur programvara och elektronik samarbetar har vi kört en rad enkla program som demonstrerar funktionen hos olika block i ABC80. Vi har sett två nya varianter av trafikljuset, en med IO-kort på ABC-bussen och program i BASIC och en simulering. Slutligen har vi sett ett exempel på användningen av den i ABC80 inbyggda realtidsklockan.

Appendix A

Z80 Instruction Set

The following is a summary of the Z80 instruction set showing the assembly language mnemonic and the symbolic operation performed by the instruction. A more detailed listing appears in the Z80-CPU technical manual. The instructions are divided into the following categories:

8-bit loads	Miscellaneous Group
16-bit loads	Rotates and Shifts
Exchanges	Bit Set, Reset and Test
Memory Block Moves	Input and Output
Memory Block Searches	Jumps
8-bit arithmetic and logic	Calls
16-bit arithmetic	Restarts
General purpose Accumulator & Flag Operations	Returns

In the table the following terminology is used.

- b ≡ a bit number in any 8-bit register or memory location
- cc ≡ flag condition code
  - NZ ≡ non zero
  - Z ≡ zero
  - NC ≡ non carry
  - C ≡ carry
  - PO ≡ Parity odd or no over flow
  - PE ≡ Parity even or over flow
  - P ≡ Positive
  - M ≡ Negative (minus)
- d ≡ any 8-bit destination register or memory location
- dd ≡ any 16-bit destination register or memory location
- e ≡ 8-bit signed 2's complement displacement used in relative jumps and indexed addressing
- L ≡ 8 special call locations in page zero. In decimal notation these are 0, 8, 16, 24, 32, 40, 48 and 56
- n ≡ any 8-bit binary number
- nn ≡ any 16-bit binary number
- r ≡ any 8-bit general purpose register (A, B, C, D, E, H, or L)
- s ≡ any 8-bit source register or memory location
- s_b ≡ a bit in a specific 8-bit register or memory location
- ss ≡ any 16-bit source register or memory location
- subscript "L" ≡ the low order 8 bits of a 16-bit register
- subscript "H" ≡ the high order 8 bits of a 16-bit register
- ( ) ≡ the contents within the ( ) are to be used as a pointer to a memory location or I/O port number

8-bit registers are A, B, C, D, E, H, L, I and R
16-bit register pairs are AF, BC, DE and HL
16-bit registers are SP, PC, IX and IY

Addressing Modes implemented include combinations of the following:

Immediate	Indexed
Immediate extended	Register
Modified Page Zero	Implied
Relative	Register Indirect
Extended	Bit

Group	Mnemonic	Symbolic Operation	Comments
8-BIT LOADS	LD r, s	r ← s	s ≡ r, n, (HL), (IX+e), (IY+e)
	LD d, r	d ← r	d ≡ (HL), r (IX+e), (IY+e)
	LD d, n	d ← n	d ≡ (HL), (IX+e), (IY+e)
	LD A, s	A ← s	s ≡ (BC), (DE), (nn), I, R
	LD d, A	d ← A	d ≡ (BC), (DE), (nn), I, R
16-BIT LOADS	LD dd, nn	dd ← nn	dd ≡ BC, DE, HL, SP, IX, IY
	LD dd, (nn)	dd ← (nn)	dd ≡ BC, DE, HL, SP, IX, IY
	LD (nn), ss	(nn) ← ss	ss ≡ BC, DE, HL, SP, IX, IY
	LD SP, ss	SP ← ss	ss = HL, IX, IY
	PUSH ss	(SP-1) ← ss_H; (SP-2) ← ss_L	ss = BC, DE, HL, AF, IX, IY
	POP dd	dd_L ← (SP); dd_H ← (SP+1)	dd = BC, DE, HL, AF, IX, IY
EXCHANGES	EX DE, HL	DE ↔ HL	
	EX AF, AF'	AF ↔ AF'	
	EXX	(BC, DE, HL) ↔ (BC', DE', HL')	
	EX (SP), ss	(SP) ↔ ss_L, (SP+1) ↔ ss_H	ss ≡ HL, IX, IY

Group	Mnemonic	Symbolic Operation	Comments
MEMORY BLOCK MOVES	LDI	(DE) ← (HL), DE ← DE+1 HL ← HL+1, BC ← BC-1	
	LDIR	(DE) ← (HL), DE ← DE+1 HL ← HL+1, BC ← BC-1 Repeat until BC = 0	
	LDD	(DE) ← (HL), DE ← DE-1 HL ← HL-1, BC ← BC-1	
	LDDR	(DE) ← (HL), DE ← DE-1 HL ← HL-1, BC ← BC-1 Repeat until BC = 0	
MEMORY BLOCK SEARCHES	CPI	A-(HL), HL ← HL+1 BC ← BC-1	
	CPIR	A-(HL), HL ← HL+1 BC ← BC-1, Repeat until BC = 0 or A = (HL)	A-(HL) sets the flags only. A is not affected
	CPD	A-(HL), HL ← HL-1 BC ← BC-1	
	CPDR	A-(HL), HL ← HL-1 BC ← BC-1, Repeat until BC = 0 or A = (HL)	
8-BIT ALU	ADD s	A ← A + s	
	ADC s	A ← A + s + CY	CY is the carry flag
	SUB s	A ← A - s	
	SBC s	A ← A - s - CY	s ≡ r, n, (HL) (IX+e), (IY+e)
	AND s	A ← A ∧ s	
	OR s	A ← A ∨ s	
	XOR s	A ← A ⊕ s	

Group	Mnemonic	Symbolic Operation	Comments
8-BIT ALU	CP s	A - s	s = r, n (HL) (IX+e), (IY+e)
	INC d	d ← d + 1	d = r, (HL) (IX+e), (IY+e)
	DEC d	d ← d - 1	
16-BIT ARITHMETIC	ADD HL, ss	HL ← HL + ss	ss ≡ BC, DE HL, SP
	ADC HL, ss	HL ← HL + ss + CY	
	SBC HL, ss	HL ← HL - ss - CY	
	ADD IX, ss	IX ← IX + ss	ss ≡ BC, DE, IX, SP
	ADD IY, ss	IY ← IY + ss	ss ≡ BC, DE, IY, SP
	INC dd	dd ← dd + 1	dd ≡ BC, DE, HL, SP, IX, IY
	DEC dd	dd ← dd - 1	dd ≡ BC, DE, HL, SP, IX, IY
GP ACC. & FLAG	DAA	Converts A contents into packed BCD following add or subtract.	Operands must be in packed BCD format
	CPL	$A \leftarrow \overline{A}$	
	NEG	A ← 00 – A	
	CCF	$CY \leftarrow \overline{CY}$	
	SCF	CY ← 1	
MISCELLANEOUS	NOP	No operation	
	HALT	Halt CPU	
	DI	Disable Interrupts	
	EI	Enable Interrupts	
	IM 0	Set interrupt mode 0	8080A mode
	IM 1	Set interrupt mode 1	Call to 0038_H
	IM 2	Set interrupt mode 2	Indirect Call
ROTATES AND SHIFTS	RLC s	CY ← [7 ← 0] ← (bit 7 to CY and bit 0); s	s ≡ r, (HL) (IX+e), (IY+e)
	RL s	[CY] ← [7 ← 0] ← CY; s	
	RRC s	[7 → 0] → CY (bit 0 to CY and bit 7); s	
	RR s	[7 → 0] → [CY] → bit 7; s	
	SLA s	CY ← [7 ← 0] ← 0; s	
	SRA s	[7 → 0] → CY, bit 7 retained; s	
	SRL s	0 → [7 → 0] → CY; s	
	RLD	A [7 4 3 0], (HL) [7 4 3 0]	
	RRD	A [7 4 3 0], (HL) [7 4 3 0]	

Group	Mnemonic	Symbolic Operation	Comments
BIT S, R, & T	BIT b, s	$Z \leftarrow \overline{s_b}$	Z is zero flag
	SET b, s	$s_b \leftarrow 1$	s ≡ r, (HL) (IX+e), (IY+e)
	RES b, s	$s_b \leftarrow 0$	
INPUT AND OUTPUT	IN A, (n)	A ← (n)	
	IN r, (C)	r ← (C)	Set flags
	INI	(HL) ← (C), HL ← HL + 1 B ← B - 1	
	INIR	(HL) ← (C), HL ← HL + 1 B ← B - 1 Repeat until B = 0	
	IND	(HL) ← (C), HL ← HL - 1 B ← B - 1	
	INDR	(HL) ← (C), HL ← HL - 1 B ← B - 1 Repeat until B = 0	
	OUT(n), A	(n) ← A	
	OUT(C), r	(C) ← r	
	OUTI	(C) ← (HL), HL ← HL + 1 B ← B - 1	
	OTIR	(C) ← (HL), HL ← HL + 1 B ← B - 1 Repeat until B = 0	
	OUTD	(C) ← (HL), HL ← HL - 1 B ← B - 1	
	OTDR	(C) ← (HL), HL ← HL - 1 B ← B - 1 Repeat until B = 0	
JUMPS	JP nn	PC ← nn	cc { NZ PO, Z PE, NC P, C M
	JP cc, nn	If condition cc is true PC ← nn, else continue	
	JR e	PC ← PC + e	
	JR kk, e	If condition kk is true PC ← PC + e, else continue	kk { NZ NC, Z C
	JP (ss)	PC ← ss	ss = HL, IX, IY
	DJNZ e	B ← B - 1, if B = 0 continue, else PC ← PC + e	
CALLS	CALL nn	$(SP-1) \leftarrow PC_H$ $(SP-2) \leftarrow PC_L$, PC ← nn	cc { NZ PO, Z PE, NC P, C M
	CALL cc, nn	If condition cc is false continue, else same as CALL nn	
RESTARTS	RST L	$(SP-1) \leftarrow PC_H$ $(SP-2) \leftarrow PC_L$, $PC_H \leftarrow 0$ $PC_L \leftarrow L$	
RETURNS	RET	$PC_L \leftarrow (SP)$, $PC_H \leftarrow (SP+1)$	
	RET cc	If condition cc is false continue, else same as RET	cc { NZ PO, Z PE, NC P, C M
	RETI	Return from interrupt, same as RET	
	RETN	Return from non-maskable interrupt	

Appendix B

Sjusegment-klocka

```
10 REM SÄTTER VÄCKNING *****************
20 A%=-1%
30 ; "Vill du ha väckning?" : GET A$ : ; A$
40 IF A$<>´j´ AND A$<>´J´ THEN 80
50 ; "Tid (HH,MM) "; : INPUT H%,M%
60 A%=60%*H%+M%
70 REM *********************************
80 LET Q$="    "
90 GOSUB 280
100 LET Q%=M%
110 PRINT CHR$(12)
120 LET U%=INT(H%/10%)
130 LET S1$=CHR$(48%+U%)
140 LET S2$=CHR$(48%+(H%-U%*10%))
150 LET U%=INT(M%/10%)
160 LET S3$=CHR$(48%+U%)
170 LET S4$=CHR$(48%+(M%-U%*10%))
180 LET Z2$=S1$+S2$+S3$+S4$
190 REM GER ALARM *********************
200 IF A%<>H%*60%+M% THEN OUT 6,0 : GOTO 240
210 IF INP(56%) AND 128% THEN A%=-1% : OUT 6%,0%
220 REM ********************************
230 OUT 6%,7%
240 GOSUB 420
250 GOSUB 280
260 LET Q%=M%
270 GOTO 120
280 REM ****** Read System Time *******
290 REM D% Contains number of days
300 REM H% Contains hours.
310 REM M% minutes and S% seconds
320 D%=0
330 IF (PEEK(65008%) AND 4%)=0 THEN 330
340 FOR I%=0% TO 2%
350 Z%(I%)=255% XOR PEEK(65008%+I%) : NEXT I%
360 Z$=ADD$(MUL$(NUM$(256*Z%(2%)+Z%(1%)),´512´,0%),NUM$(Z%(0%)*2%),0%)
370 IF COMP%(Z$,´8640000´)>-1% THEN D%=D%+1% : Z$=SUB$(Z$,´8640000´,0%) : GOTO 370
380 Z=VAL(Z$)/100
390 H%=Z/3600 : Z=Z-3600*H%
400 M%=Z/60 : S%=Z-60%*M%
410 RETURN
420 REM *******************************
430 REM 7-segment display generator
440 REM Z2$ = 4 siffror
450 REM Q$ initieras med 4 space
460 REM Q$ = Z2$ vid RETURN
470 FOR I%=0% TO 23%
480 ; CUR(I%,0%)CHR$(23);
490 NEXT I%
500 Y%=6%
510 X%=5%
520 FOR B%=1% TO 4%
530 RESTORE
540 T$=MID$(Z2$,B%,1)
550 D%=VAL(T$)
560 IF T$=MID$(Q$,B%,1) THEN 670
570 IF D%=0% THEN 610
580 FOR E%=1% TO D%
590 FOR F%=1% TO 7% : READ I% : NEXT F%
600 NEXT E%
610 FOR A1%=1% TO 7%
620 READ I%
630 T%=127% : IF I%<0% THEN T%=32%
640 I%=ABS(I%)
650 ON I%+1% GOSUB 1150,820,870,920,970,1020,1070,1120
660 NEXT A1%
```

Sakregister

A 58
ABC-bussen 125, 168, 179, 218
ackumulator 37, 58
address map 80
adress 36, 100
adressbuss 36, 68
adressbygling 173
adressering 99, 172
adresskarta 80
adressledning 29
adressregister 37, 58
aktiv låg 67
alfanumerisk kod 51
ALU 45, 58
AND-grind 11
anpassningskrets 5
AR 58
aritmetisk logisk enhet 45, 58
ASCII-kod 53, 142
ASIA 27, 166
assembler 76, 78
assemblering 74
assemblerprogram 77
avbrott 115
avbrottshantering 115, 116
avbrottsmask 117
avkodare 20
avlusa 72
avlusning 86
avspelningsrutin 196
avsökning 136

bakgrundsprogram 115
BASIC-sats 187
BASIC-tolk 186
BCD-kod 51
begäran 67
belastning 5
bell 157
benchmark 213
bibliotek 86
bidirectional 93
bildadress 146
bildminne 100, 143, 207
bildsignal 159
bildsynk 159
bildväxling 145
binary coded decimal 51
binär aritmetik 47
binära signaler 9
binära tal 46
binärt 4
binärt talsystem 46
biphase encoding 162
bipolär teknologi 106
bithantering 45, 200
bitledning 21
bitorganiserad 101
bitvikt 46
blandare 107
blank 152
blank skärm 157
block 194
blockavkodning 99
blockschema 91
BOFA 185
bricka 71
brusgenerator 107
brädgård 135
bubbelminne 31
buffert 10, 73
buffrad 126
BUF1 186
BUF2 188
bug 86
bus driver 73
busskontakt 125
bygling 172
byglingsadress 172
byte 21

card select 174
carriage return 76, 157, 186
carry 47
CAS 103
CE 129
centralenhet 37
central processing unit 37
character 53
checksumma 163, 194
chip 71
chip enable 129
chip select 71, 130, 174
CMOS-struktur 106
Column Address Strobe 103
comment 76
Complex Sound Generator 108
computer 2
control bus 59, 67
CPU 37, 58
CR 157, 187
CRC-summa 163
cross assembler 78
CR-tangent 76
CS 130
CTRL 135
CUR 198
cursor 146
cursorblink 145
cyclic redundancy code 163

Daisy chain 117
DA-omvandlare 112
datablock 194
databuss 36, 68
dataingång 26
datalatch 26
dataledning 29
dataväg 42
dator 2
datorarkitektur 65
debouncing 137
debug 72
debugging 86
dedicated computer 69
dedicated microcomputer 89
destination 75
DH 147
digitala kretsar 9
DIL-kapsel 12
D-ingång 26
direct 57
directive 76
divide by four 27
drivförmåga 5
drivkrets 93
dubbelriktad 93
D-vippa 26
dynamiskt minne 31
dynamiskt RAM 101

E 130
editor 78
effektstyrning 211
E-ingång 130
ELLER 18
enable 14, 130, 174
ENVELOPE 107
EOFA 185
EPROM 23, 98
erasable programmable read only memory 23
etikett 76
ettställ 15
ETX 194
exclusive or 19
EXECUTE 40
Explosion 110
extension 195

fast logik 4
felsöka 72
felsökning 86

FETCH 40
FF 157
fil 194
firmware 98
flagga 53
flaggregister 60
FLAGS 60
flow graph 34
flyktigt 31
flytande 14
flyttal 189, 215
flödesplan 34
flödesschema 33
FM 163
form feed 157
frekvensmodulering 163
från 4
frånläge 7
full adder 48
fusable link 23, 105
färdig-signal 118
fördröjd horisontalsynk 147

gate 11
generell dator 89
generellt system 89
graf 152
grafik 154, 214
grafkommando 155
grafmod 155, 157
GRAY-kod 51
grind 11
grundkretsar 9
Gun Shot 110

H 5
half adder 47
halvadderare 47
handskakning 118
hardware 74
header 194
HEAP 185
heladderare 48
heltal 189
hexadecimala tal 48
hextal 49
hoppadress 115
horisontalavlänkning 145
H-synk 147, 159
huvudprogram 55
hysteresingång 122
hårdvara 66, 74
hög 5
hög adress 97
högimpediv 14

IEC-symboler 19
icke maskerbart avbrott 117
immediate 57
indirekt adressering 118
INITA 84
initialisering 182
initialized 83
initiering 83
injektionslogik 107
inport 66
inspelningsformat 193
inspelningsrutin 195
instruction register 38
instruktion 2, 39
instruktionshämtning 40
instruktionsregister 38, 58
INT 117
Integrated Injection Logic 107
interaktivt 193
interface 5
interface-krets 5
interlace 144
internkodskompilator 190
interpretator 192
interrupt 115
interrupt acknowledge 115, 117
interrupt mask 117
in-ut-krets 5
inverter 10
inverterare 10
IO-adress 80
IO-buss 170, 174
IO-device 5
IO-mapping 68
IR 38, 58
I-register 117
I^2L 107

J-ingång 16

K 31
kallstart 183
kanalval 174
Kansas City 161
kapacitivt tangentbord 137
kapselval 99, 100
kassaapparat 44
kassettinterface 124, 160
kassettprogram 167
kassettrutin 193
KC 161
K-ingång 16
kiselstyre 105, 106
klarsignal 37, 129, 174
klocka 15
kolumnadresstrob 103
kombinationskrets 11
kommando 76, 187
kommandotabell 187
kommentar 76
kompilator 192
kompilering 188
konfiguration 84
kontaktstuds 136
kontrollbuss 59, 67, 68
kontrolledning 29
kontrollsignal 66
kortval 174, 218
kraftförsörjning 71
kringkrets 28
källa 75
källprogram 76, 78
kärnminne 31

L 5
label 76
large scale integration 105
latch 26
LDA 39
LF 157
line driver 124
line feed 157
line receiver 122
linjeadress
linjedrivkretsar 124
linjeframmatning 145
linjemottagare 122
linjepunkt 145
linjesynk 159
linjeutgångar 124
linking loader 86
ljudgenerator 107, 196
loop 57, 115, 200
Low Power Schottky 107
LS 107
LSI 105
låg 5
låg adress 97
läskommando 172
läskontroll 67
läsminne 21

markör 146, 152
mask 45
maskerbart avbrott 117
maskinkod 193
maskinvara 66, 74
maskprogrammera 23
matematikmaskin 2
matris 136
MEMO-kod 39, 74
memory mapping 68
metal oxide semiconductor 105
mellanslag 77
metal gate 106
metallstyre 106

MFM 163
mikrooperation 42
mikroprocessor 37
mikroprogram 41, 42
mikroprogrammering 42
minimalsystem 69
minne 205
minnesadress 80, 146
minnesbuss 171
minneskort 172
minnesplan 80
minnessiffra 47
MIXER 107
mjukvara 74
mjukvarubuffert 115
mnemonics 39
modified frequency
modulation 163
modul 84
MOP 41, 42
MOS 105
MOS-teknologi 106
MOS-transistor 106
motorrelä 124
motorstyrning 164, 202
MPU 37
MS-vippa 15
multiemittertransistor
106
multikollektortransistor
106
multiplex 102
multiplexer 25
multiplexersignal 103
MUX 25, 103
mästare 15
μP 37

NAND-grind 12, 18
namnblock 194
NELLER 18
N-key roll-over 141
NMI 117
NOCH 18
NOISE 107
nollflagga 58
nollställ 15
NOR-grind 18
numerisk kod 51
nummertecken 135

objektkod 193
OCH 18
off 4
oktaltal 49
on 4
opcode 75, 76
operand 75, 76
operationskod 75
opkod 75
optokopplare 211
ord 21
ordledning 21
ordlängd 21
ordorganiserad 98
Organ 111
OR-grind 18
OUT-sats 197

panikflagga 184
parameter 58
PC 38, 58
PEEK 206
pekare 184
PE-metoden 162
periferikrets 28
phase encoding 162
Phasor Gun 110
PIO 27
PIO-krets 114
pipelining 158
PISO 27
PISO-register 27
planartransistor 106
POC 120
point of sale terminal 44
POKE 206
positionsadress 147
positionsvärde 46
power supply 71
power up 183
processor clear 120
program 2
program counter 38
programmable read only
memory 23
programmed logic 88
programmerad logik 88
programmodul 84, 86
programpekare 38
programräknare 38, 58
programvara 74, 181
PROM 23
Prop Plane 111
protokoll 162
pseudokod 76
pull up-motstånd 120

racing 15
radadress 147
radadresstrob 103
radbuffert 186
radframmatning 157
radsprång 144
RAM 26
RAM-interface 103
random access 26
random access memory
26
RAS 103
read only memory 21
realtidsklocka 221
refresh 31, 104
refreshing 31
register 26
request 67
reset 15, 120
RESET-ingång 38
resident assembler 78
response control 124
return 135, 186
Reverse Polish Notation
190
R-ingång 16
roll-over 141
ROM 21, 98
Row Adress Strobe 103
RPN 190
R-register 104
RS-232C 93
räknare 28, 145, 197
räknemaskin 2

sanningstabell 11
satsbuffert 188
SAVE 193
scanner 138
scanning 136
scanpuls 141
sekvenskrets 26
select 71
sensorer 3
separatadresserad IO 68
sequencing 41
set 15
SHIFT 135
signalingång 11
silicon gate 105, 106
simulering 224
single chip computer 69
S-ingång 16
SIO 27, 166
SIPO 27
SIPO-register 27
Siren 110
självklockande 162
skiftregister 27
skrivkommando 172
skrivkontroll 67
skriv/läskontroll 100
skriv/läsminne 26, 29
slav 15
SLF 107
slinga 115
slumptalsgenerator 216
slutmärke 195
snitt 5
software 74
sol 135
source 75
source code 76

source program 76
SP 58
space 77
Space War 110
specialdator 69
specifikation 3
SR-vippa 15
STA 57
stack 55
stackpekare 55, 58, 83
185
stack pointer 83
statiskt minne 31
statiskt RAM 100
statiskt relä 211
strob 118
strukturerad programmering 84
stryps 7
sträng 189
STX 194
styrbuss 67
styringång 11
subrutin 55
subrutinbibliotek 86
sudda 198
summa 47
super low frequency 107
SYNC 194
SYNK 147
synkront 162
synksignal 159
syntax 78
systemreset 120
sänka 7

TAB 206
tangent 137
tangentbord 133, 203
Tarbell 162
tecken 53, 152
teckenframmatning 145
teckengenerator 152
teckengenerering 150
teckenkommando 155
teckenmod 155, 157
teckensträng 186
teknologi 105
teletext 143, 145
textfil 78
text-TV 145
till 4
till-läge 7
timing 41, 103, 128
tolk 186, 192
trafikljusprogram 62
Train 111
Transistor-Transistor-Logic 107
trigga 14
tristate 13, 14
TTL 5, 107
TTL-kompatibel 7
TTL-nivå 5

UART 166
universal asynchronous receiver transmitter 166
USA-symboler 19
utport 66
uttrycksberäknare 192
utvecklingssystem 78
vagnretur 135, 157
variabelrot 185, 190
VCO 107
vektor 115
verkställande 40
vertikalavlänkning 145
videointerface 143
view data 143
villkorligt hopp 53
vippa 14
wired logic 88
volatilt 31
voltage controlled oscillator 107
V-synk 147, 159
V24-snitt 93, 208

XMEMFL 127
XMEMW 127
XOR 19
XOR-grind 19

yttre IO-enhet 129
yttre minne 126

Z 58
Z-flagga 53

återgång 146

Litteratur

1. En översikt om digitala kretsar (behandlar även minnen och busskretsar): Markesjö, G: "Digitala kretsar", Esselte Studium 1976 ISBN 91-24-26209-9

2. En beskrivning av en mikroprocessors funktion och uppbyggnad återfinns i boken: Markesjö, G och Höglund, I: "En mikroprocessor i CMOS. Funktion och tillverkning", Esselte Studium 1978 ISBN 91-24-27562-X

3. En förenklad framställning av en dators funktion återfinns i boken: Markesjö, G: "Bli bekant med en minidator", Studentlitteratur 1976 ISBN 91-44-11361-7

4. En bra amerikansk bok om mikrodatorer: Zaks, Rodnay: "Microprocessors, from drips to systems", SYBEX 1977 ISBN 0-89588-001-8

5. En bra amerikansk bok om interfacekretsar: Zaks, Rodnay: "Microprocessor interfacing techniques", SYBEX 1977 ISBN 0-89588-000-8

Mikrodatorns ABC

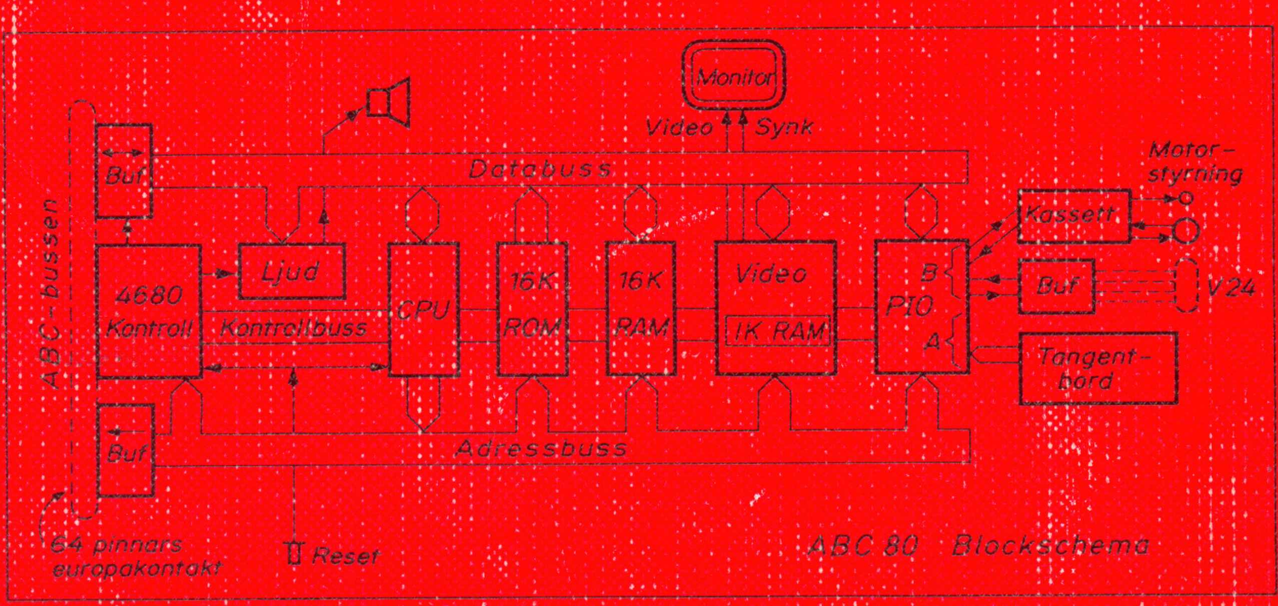
Den här boken är avsedd för dig, som har grundkunskaper i elektronik och nu vill få en orientering om mikrodatorns teknik.

Som tillämpningsexempel har författaren valt den svenska mikrodatorn ABC 80 som är uppbyggd kring mikroprocessorn Z80A.

Först beskrivs komponenter och funktion hos ett enkelt mikrodatorsystem uppbyggt med standardkretsar. Därefter får man lära känna ett generellt mikrodatorsystem med dess viktigaste periferienheter: tangentbordet, bildskärmen och kassettbandspelaren. Författaren ger också en beskrivning av hur mikrodatorsystemet kan expanderas med en generell buss. Boken avslutas med en genomgång av programvaran. Flera program studeras i detalj.

Tidigare utgivna böcker av samma författare:
Mikroprocessor i CMOS (i samarbete med Ingmar Höglund)

Elektronikserien:
Inledande kurs A, Inledande kurs B, Pulskretsar, Digitala kretsar, Analoga kretsar, Effektkretsar (i samarbete med Ingmar Höglund)



ISBN 91-24-29008-4
Best. nr 24-29008-4
(24-29008-4) B

ESSELTE STUDIUM