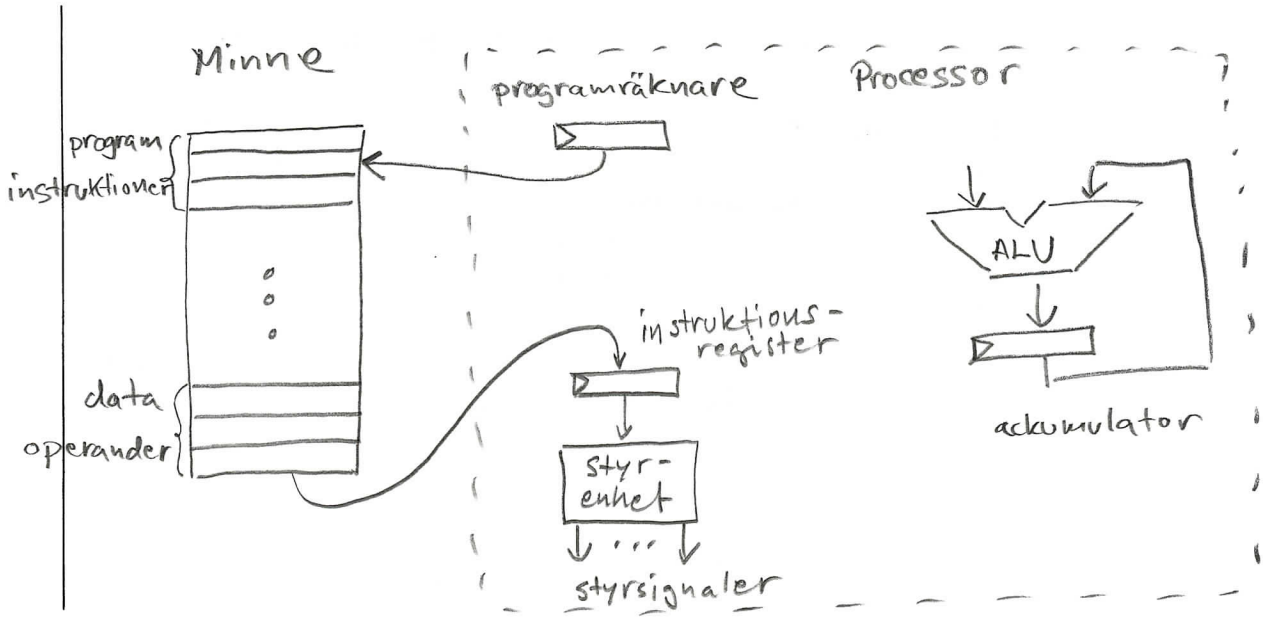
8 Mikrokod

Begrepp i denna föreläsning: *Styrautomat,* NEXT- *och* CTL-*fält, mikrokod, märkfält, mikroarkitektur, DMA, optimerad mikrokod.*

8.1 Styrsignaler

Den första, enkla, processormodellen såg ut så här:



i vilken vi kunde följa instruktionsflödet från instruktionshämtningen tills att den hamnade i instruktionsregistret. På något — hittills — magiskt sätt skapades sedan styrsignaler för resten av hårdvaran och instruktionen exekverades. Vi undvek då helt att beröra dessa styrsignalers ursprung och verksamhet. Av bilden kan man tro att styrsignalerna bara behövs *efter* att instruktionen avkodats och skall exekveras. Det är dock en förenklad bild av verkligheten: Styrsignalerna behövs redan för att överhuvudtaget kunna läsa in instruktionen till instruktionsregistret!

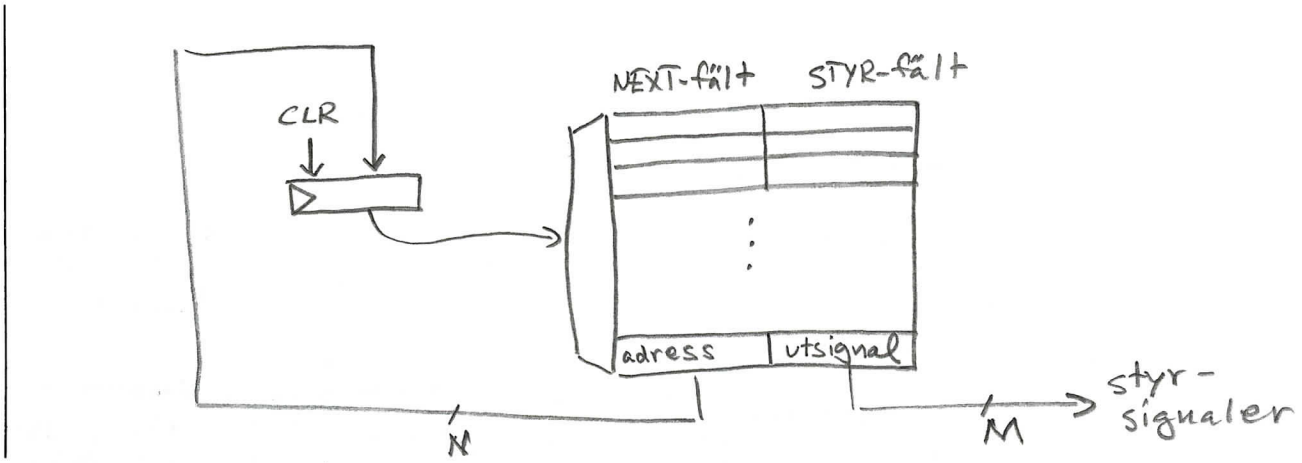
Utan styrsignaler kan vi inte utföra ens de mest grundläggande momenten i hämtfasen. Vi behöver något som

- släpper ut programräknaren till att adressera minnet,
- säger åt minnet att vi vill läsa ut instruktionen, samt
- något som får instruktionsregistret att läsa in instruktionen.

Detta "något" är styrsignalerna som kommer ur "?"-blocket. Problemet är nu hur ska vi få dem att uppträda i rätt ordning och vid rätt tillfälle? Från digitaltekniken kommer vi emellertid ihåg att detta är just vad en tillståndsmaskin gör.

8.2 Styrautomat

Utgående från ett tillståndsdiagram kan man med vippor och logik realisera sekvenser av lämpliga styrsignaler. Med digitalteknikens angreppsmetod vet vi att det är relativt lätt att åstadkomma detta för kortare sekvenser men metoden blev svårhanterlig med ökande antal utsignaler och tillstånd. I digitalteknikkursen presenterades dock en alternativ metod att med minne realisera en tillståndsmaskin — med en *styrautomat*. En styrautomat är ett lätt utbyggbart sekvensnät som enklast realiseras med ett register och ett minne.[1]



Det är viktigt att förstå hur styrautomaten fungerar för att senare kunna utöka den till att bli en väsentlig beståndsdel av en processor. Funktionen är som följer:

1. Vid spänningspåslag, eller om CLR sätts aktiv, nollställs registret. Registret pekar därmed ut adress 0 i minnet.
2. På rad 0 innehåller minnet fältet NEXT den adress som utgör nästa tillstånd. Denna adress ligger sedan omedelbart[2] på registrets ingång.
3. Vid nästa klockflank kommer denna nya adress att låsas av registret och visa sig som ny adress in till minnets NEXT-fält, vilken utgör minnets adress vid nästa klockflank, och så vidare.

Styrautomaten är alltså en realisering av ett sekvensnät med den fördelen att det är enkelt att lägga till ytterligare tillstånd — det är bara en fråga om att lägga in nya hoppadresser i NEXT-fältet[3] — och man ser att strukturen medger godtyckligt komplicerade

[1] Vid andra genomläsningen: Observera att detta minne **inte** är samma minne som används för assemblerprogrammen. Detta minne ligger normalt dolt inuti processorn — oåtkomligt för andra än tillverkaren. Nästan alltid är minnet också av en sort som bara kan läsas från, aldrig skrivas till.

[2] Vi bortser från den lilla tidsfördröjning som minnet innebär. Ointressant just nu.

[3] En form av programmering, väl?

hopp i en tillståndsgraf. Antalet tillstånd begränsas av antalet rader i minnet och med en bredd hos NEXT-fältet lika med N blir största antalet tillstånd 2^N.

Ett sekvensnät brukar ha utsignaler specifika för respektive tillstånd och vår styrautomat är inget undantag. Samtidigt som styrautomaten genomlöper sina tillstånd kan den, för varje tillstånd, ge upp till M stycken utsignaler. Utsignaler som kan användas för att styra yttre hårdvara, och, som det skall visa sig, även ändra i styrautomatens egna beteende.

Den beskrivna styrautomaten har flera fördelar. Bland annat är det en enkel konstruktion som dessutom är enkel att modifiera. För att visa på det senare skall här redovisas hur den kompletteras för att

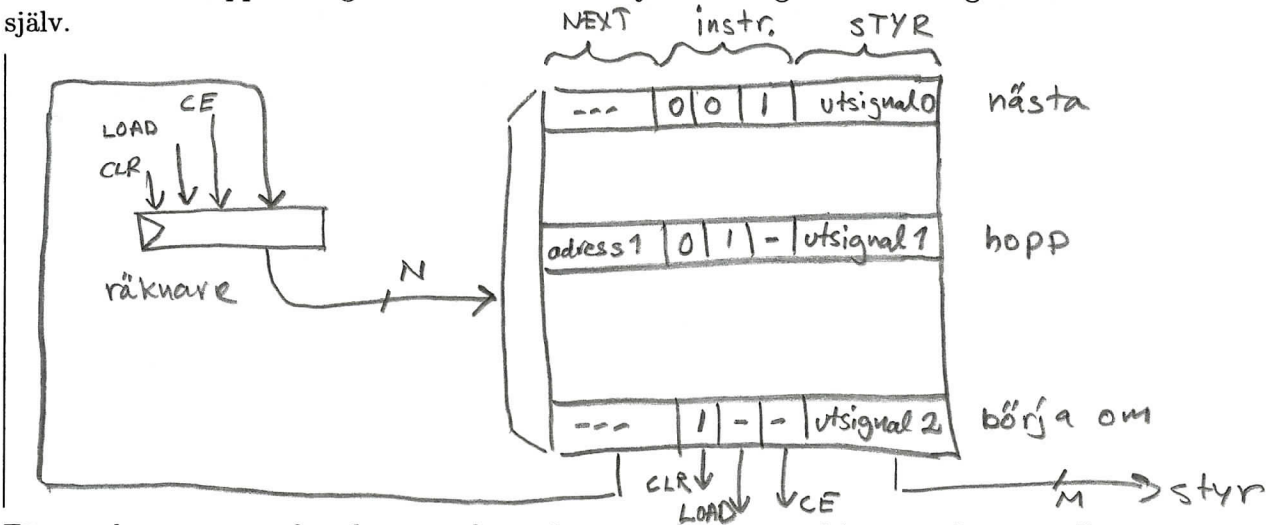
1. bli lättare att programmera, och
2. kunna genomföra *villkorliga* hopp beroende på externa insignaler.

Vi angriper punkterna i tur och ordning och börjar alltså med den första.

8.2.1 Modifierat NEXT-fält

I nuläget måste NEXT-fältet *alltid* innehålla adressen A till nästa tillstånd. Detta är bra om man ständigt måste hoppa överallt i minnet, men är annars onödigt. Med lite eftertanke kan man ofta lägga tillstånden efter varann, förutom i de fall hopp verkligen är nödvändiga.

För att förenkla programmeringen inför således vi en N-bits *räknare* istället för registret. Vi låter räknaren automatgenerera nästa adress, $A+1$, med varje klockpuls, och slipper alltså programmera NEXT-fältet i många fall. NEXT-fältet behöver nu bara innehålla en adress då adressuppräkningssekvensen behöver brytas. I övriga fall klarar sig automaten själv.



För att kunna genomföra hopp utökas minnet med ett nytt fält *control*, CTL, vilket innehåller en en-bits signal som styr räknarens LOAD-ingång, dvs i praktiken bestämmer om hopp ska ske eller inte.

För att ytterligare poängtera den frihetsgrad en styrautomat ger konstruktören låter vi CTL-fältet i figuren även innehålla en CLR-signal, som kan användas för att nollställa

styrautomaten programmässigt, dvs helt oberoende av en extern *RESET*-signal. Oavsett vilken signal som används kommer styrautomaten att börja om från rad 0. Den yrvakna styrautomaten kan inte avgöra vilken av de två signalerna som orsakade omstarten.

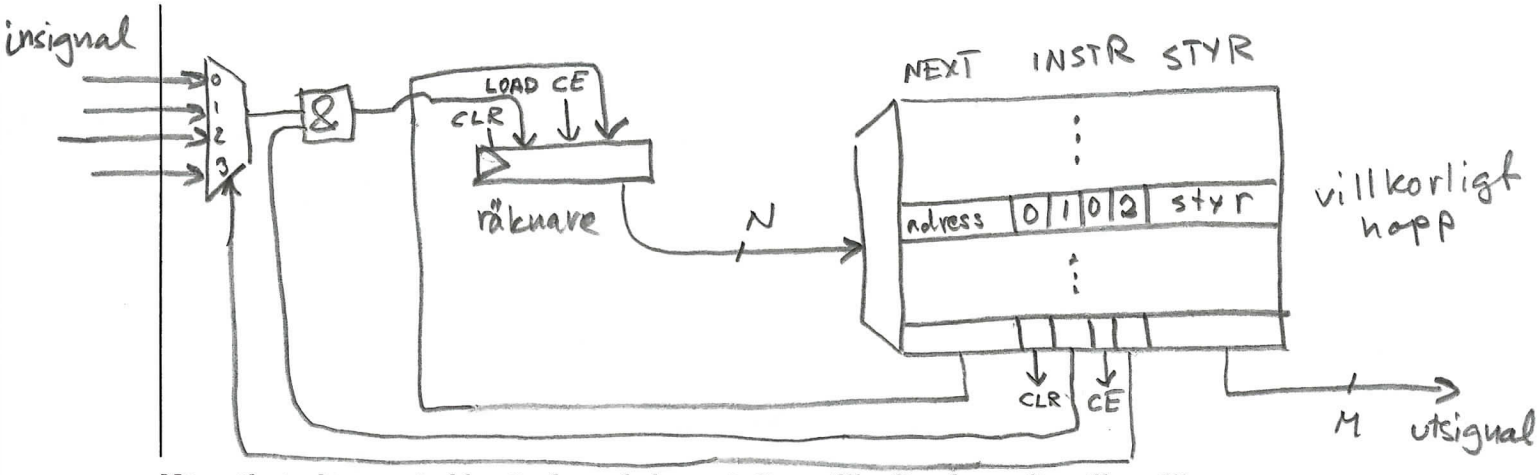
Det kan vara förnuftigt att programmera alla rader i minnet som inte motsvarar "legala" tillstånd med CLR. Då vet man i alla fall att den försöker starta om sig, om den av någon anledning gått över styr. Det är förstås katastrofalt att låta styrautomaten fortsätta hoppa mellan diverse slumpmässiga tillstånd om den en gång klivit ur sin tillståndsgraf. Det är säkrare att starta om den eller kanske helt stänga av den.

8.2.2 Villkorliga hopp

För att möjliggöra att en eller flera yttre signaler utgör villkor för hopp i styrautomatens sekvens måste konstruktionen dessutom modifieras för att kunna

- skilja ut ett — av möjligen flera — villkor, och
- hoppa om detta villkor är uppfyllt.

Lösningen är enkel. Låt ytterligare bitar i CTL-fältet peka ut rätt villkor och låt dessa bitar påverka en multiplexer, *MUX*. Den senare komponenten används sedan för att låta den yttre signalen direkt[4] påverka LOAD-ingången på räknaren.



När väl strukturen är klar är det enkelt att införa villkorliga hopp för vilka villkor som helst. Vi kan exempelvis låta statusregistrets bitar utgöra insignaler till muxen och har plötsligt möjliggjort för exempelvis instruktionerna JMPZ, JMPN, JMPC och JMPV. Muxen kan faktiskt användas för ovillkorligt hopp också: Låt en insignal till muxen alltid vara hög!

8.3 Mikrokod

På samma sätt inses att man med ytterligare hårdvara och fler styrsignaler i minnet kan skräddarsy en mycket komplicerad apparat runt en styrautomat. Med dussinet eller fler

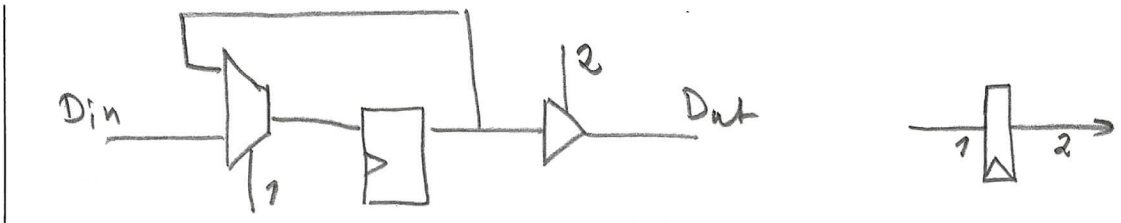
[4]Möjligen via en synkroniseringsvippa om signalen inte redan är synkron med systemklockan!

utsignaler kan styrautomaten exempelvis utgöra den enhet som dirigerar dataflödet i en processor. Och det är faktiskt precis det vi vill göra. Det ligger en styrautomat bakom alla styrsignaler i en processor.[5] Innehållet i styrautomatens minne kallas i sådana fall *mikrokod*. Med förståelse för hur styrautomaten fungerar och vad som ska åstadkommas — hämta, avkoda, utför — är det trivialt att bygga ihop en processor!

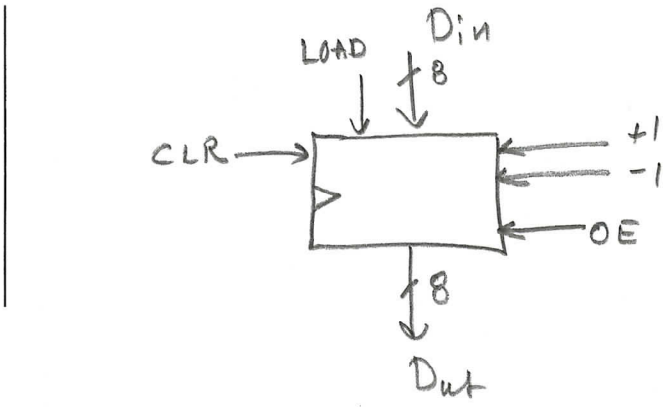
8.4 Byggelement

Innan vi bygger processorn ska vi friska upp minnet på de huvudsakliga digitala byggelement vi har att tillgå, utöver enklare grindar som AND, OR och så vidare. För enkelhets skull väljer vi så "generella" komponenter som möjligt. Allt enligt digitaltekniken:

8.4.1 Register



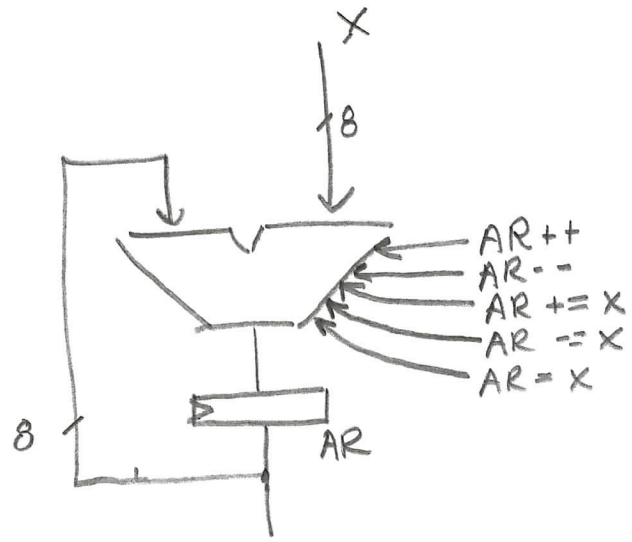
8.4.2 (Universella) räknare



[5] Detta är numera inte *helt* sant, principen gäller däremot fortfarande. Och det är ett mycket strukturerat sätt att bygga en mikroprocessor på.

8.4.3 Aritmetisk Logisk Enhet, *ALU*

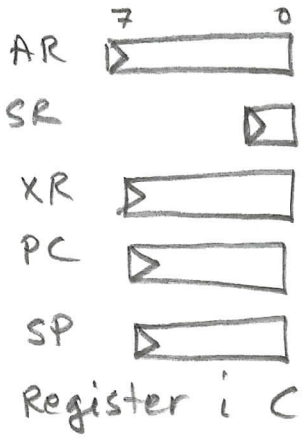
+ackumulator

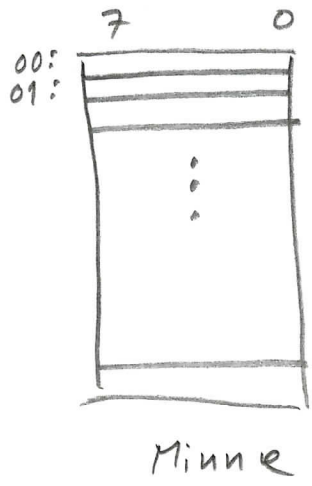


8.5 Programmerarmodell

Processorn måste ha en programmerarmodell. För att förenkla framställningen bestämmer vi nu att inte genomföra en full modelldator. Vi väljer att tillverka en dator med enbart 8-bitars register. Detta får till följd att programmen bara kan bli $2^8 = 256$ rader långa, men det duger för oss här.

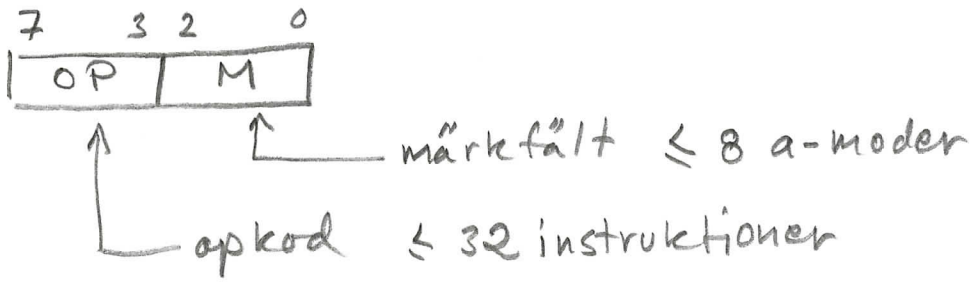
Alltså, programmerarmodellen blir...



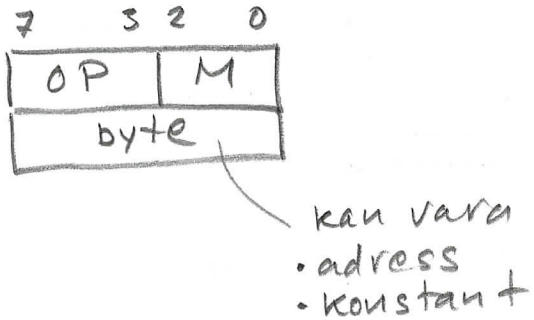


... och vi börjar med att bara stödja två instruktionsformat:

- Ett enbytes...



- ...och ett tvåbytes:



Märkfältet

Det nya i instruktionsformatet är att vi har infört ett *märkfält*, M, som anger vilken adresseringsmod instruktionen skall använda. Med ett trebitars märkfält har vi plats för åtta adresseringsmoder men vi väljer att bara tillåta dessa:

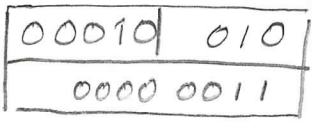
M	Mod	Exempel	EA
000	Absolut	`LDA` *addr*	*addr*
001	Indirekt	`LDA` (*addr*)	$M(addr)$
010	Indexerad	`LDA` *disp*`,(XR)`	$XR + disp$
011	Relativ	`JMP` *disp*	$PC + 2 + disp$
100	Omedelbar	`LDA #`n	$PC + 1$
101	Underförstådd	`INCA/INC`	—

Instruktionerna

Med ett fem bitar brett fält kan vi ha högst 32 instruktioner. Vi tillåter fältet OP att innehålla någon av följande instruktioner:

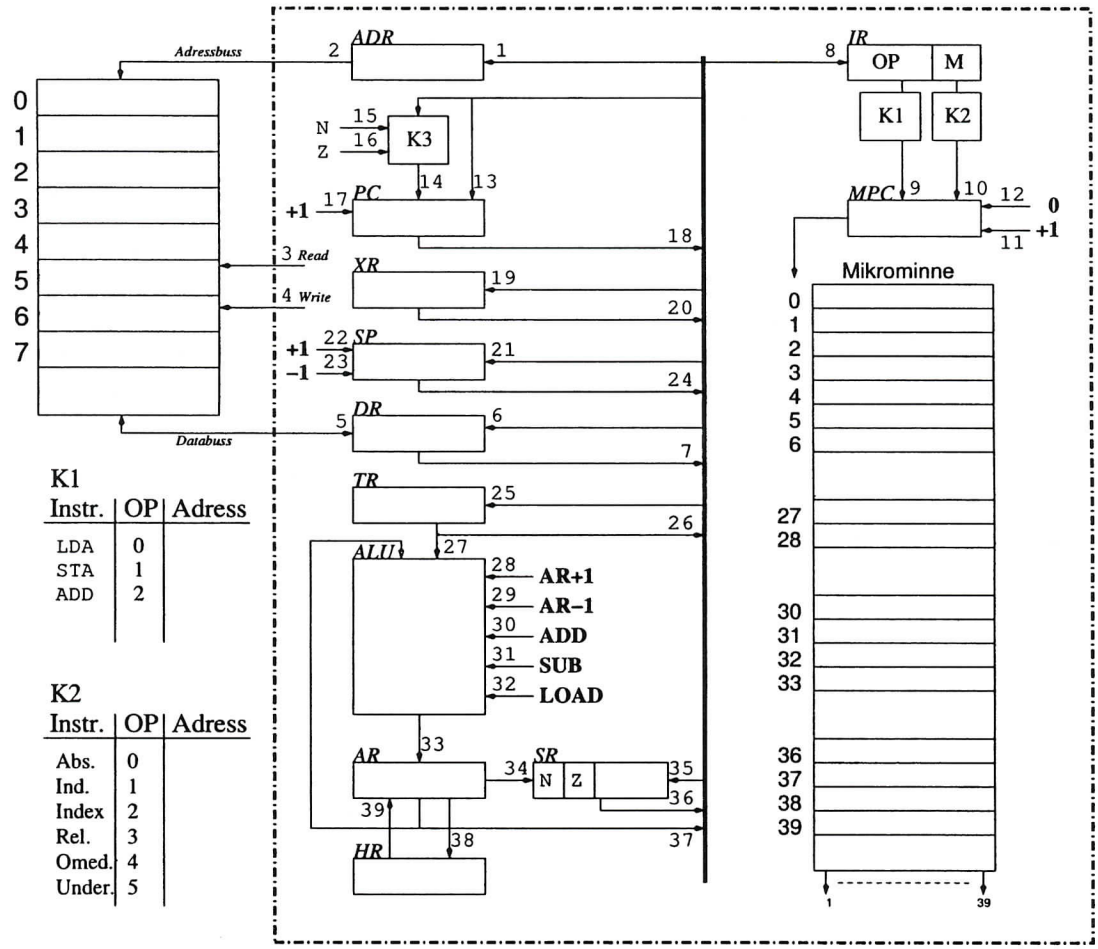
Instruktion	Verkan	Status			
		N	Z	C	V
`LDA` *addr*	$AR := M(addr)$	*	*	-	0
`STA` *addr*	$M(addr) := AR$	-	-	-	0
`ADD` *addr*	$AR := AR + M(addr)$	*	*	*	*
`SUB` *addr*	$AR := AR - M(addr)$	*	*	*	*
`INCA`	$AR := AR + 1$	*	*	*	*
`DEC`	$AR := AR - 1$	*	*	*	*
`CMP` *addr*	$AR - M(addr)$	*	*	*	*
`CLRA`	$AR := 0$	0	1	0	0
`ASRA`	$AR := AR/2$	*	*	*	-
`ASLA`	$AR := AR \cdot 2$	*	*	*	*
`LSRA`	logiskt högerskift av AR	0	*	*	-
`AND` *addr*	$AR := AR$ and $M(addr)$	*	*	-	0
`OR` *addr*	$AR := AR$ or $M(addr)$	*	*	-	0
`JMP` *addr*	$PC := addr$	-	-	-	-
`JMPN` *addr*	$PC := addr$ om $N = 1$	-	-	-	-
`JMPZ` *addr*	$PC := addr$ om $Z = 1$	-	-	-	-
`JMPC` *addr*	$PC := addr$ om $C = 1$	-	-	-	-
`JMPV` *addr*	$PC := addr$ om $V = 1$	-	-	-	-
`IN`	$AR := IN$	*	*	-	0
`OUT`	$UT := AR$	-	-	-	0

Exempelvis kommer instruktionen `ADD 3,(XR)`[6] med indexerad adressmod att, i programminnet, komponeras enligt:



8.6 Mikromaskinen

Då är vi klara att sätta ihop hela processorn. Färdigritad kan den se ut som denna, där själva mikromaskinen är speciellt markerad:



Vi identifierar flera kända processorbeståndsdelar: Processorregistren PC, AR, SP, m fl är kända från programmerarmodellen men mikromaskinen innehåller ytterligare register

[6]Som utför $AR := AR + M(ea)$ där $ea = XR + 3$.

Steg 3. Exekvera (verkställ) instruktionen Detta avslutande steg utför instruktionens "nyttolast", den egentliga datamanipulationen. Det kan handla om att utföra additionen i `ADD`-instruktionen, eller att flytta ackumulatorinnehållet till en adress i det yttre minnet (`STA`), till exempel. Nätet K_1 översätter mellan opkod och den adress i mikrokodsminnet där instruktionens mikrokod ligger.

8.6.2 Normal arbetsgång — alla detaljer

Vi tillverkar nu den mikrokod som krävs för att köra följande assemblerprogram:

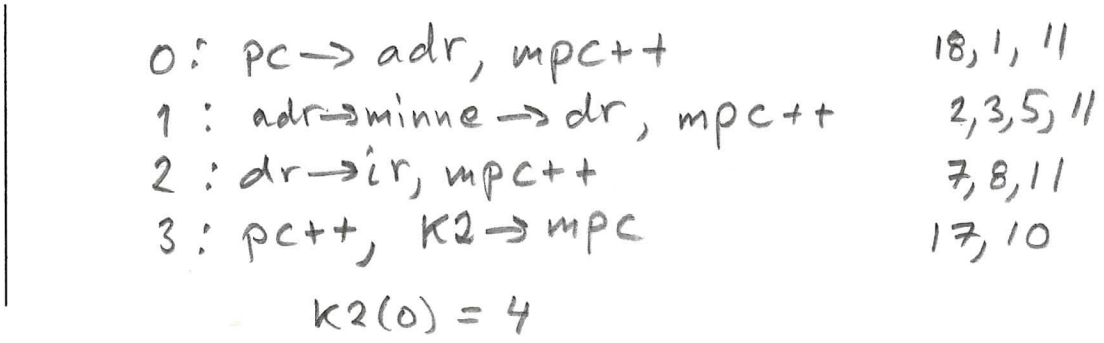
Instruktion	Betydelse
`LDA 12`	$AR := M(12)$
`ADD #7`	$AR := AR + 7$
`STA 13`	$M(13) := AR$

Det vill säga vi skall tillverka mikrokod för dessa instruktioner (`LDA`, `ADD` och `STA`) samt de adresseringsmoder som används. Observera att när vi en gång gjort mikrokod för en adresseringsmod kan andra instruktioner med samma adresseringsmod dra nytta av samma mikrokod. På samma sätt kan exempelvis `LDA`-instruktionen ovan användas så fort `LDA` önskas utförd *oberoende av adresseringsmod.*

Innan vi börjar stega igenom mikrokoden måste vi trycka på *RESET* för att få hela processorn till ett känt utgångsläge, med alla register nollställda.

Steg 0. Initiera processorn Tryck på *RESET*. Alla register nollställs i och med detta. Speciellt noterar vi $0 \rightarrow MPC$ och $0 \rightarrow PC$.

Steg 1. Instruktionshämtning Här gäller det att styra "slussarna" så att instruktionen kommer in i IR. PC är nollställd första gången.



Steg 2. Avkoda använd adressmod Vi behöver absolut adressering först så vi börjar med den. Eftersom nätet K_2 översätter mellan vald adresseringsmod och motsvarande rutins början i mikrominnet låter vi nu $K_2(0) = 4$, så att den "nollte" adresseringsmoden (absolut adressering) börjar på rad 4 i mikrominnet.

```
4:  pc→adr, mpc++          18,1,11
5:  adr→minne→dr, mpc++    2,3,5,11
6:  dr→adr, K1→mpc, PC++   7,1,9,17

K1(0) = 30
```

När adressmodsfasen är klar ligger *effektiv adress* i ADR. Nätet K_1 träder nu in och hoppar i mikrominnet till rätt instruktion. Glöm alltså inte att fylla i K_1 med rätt adress.

Steg 3. Exekvera instruktionen Instruktionen `LDA`:s datamanipulering börjar. Vi lägger den på rad 30 i mikrominnet.

```
30: adr→minne→dr, mpc++    2,3,5,11
31: dr→tr, mpc++           7,2,5,11
32: tr→ar, mpc++           27,32,33,11
33: status, 0→mpc          34,12
```

I och med detta är exekveringsfasen klar. Instruktionen är nu korrekt inhämtad, avkodad och utförd. PC pekar redan nu på nästa *assembler*instruktion — det fixade vi redan i slutet av adressmodsfasen.

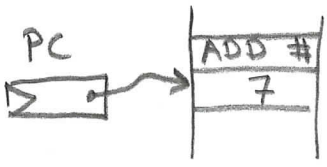
Den första assemblerinstruktionen är klar. Fortsätt med nästa, det blir tre steg en gång till:

Steg 1 (igen). Hämta nästa instruktion I och med att MPC nollställdes påbörjas en ny hämtfas, den då instruktionen `ADD #7` ska hämtas. Hämtfasen utförs på exakt samma sätt som ovan så vi behöver inte skriva ny mikrokod för denna.

Steg 2. Avkoda adressmoden En skillnad är dock nu att adresseringsmoden är ändrad, `ADD #7` innebär moden "omedelbar operand". Vi placerar mikrokoden för denna mod på rad 27 i mikrominnet[8] och går dit:

```
27: pc→adr, mpc++    18,1,11
28: PC++, K1→mpc     17,9

K1(2) = 36
```



När EA befinner sig i ADR är adressmodsfasen över.

[8] Och vi glömmer inte att uppdatera K_2 samtidigt

Steg 3. Exekvera instruktionen Vi skall nu fortsätta med mikrokoden för ADD-instruktionen. Vi lägger den på mikrorad 36:

```
36 : adr -> minne -> dr, mpc++      2,3,5,11
37 : dr -> tr, mpc++                7,25,11
38 : ar+tr -> ar, mpc++             27,33,30,11
39 : status, 0 -> mpc               34,12
```

Nu är *PC* ånyo klar för nästa assemblerrad och det är fritt fram att angripa den sista instruktionen, STA 13. Utför den själva! (Vilken adresseringsmod handlar det om?)

- Hämtfas finns redan
- Absolut a-mod finns redan
- Exekvering

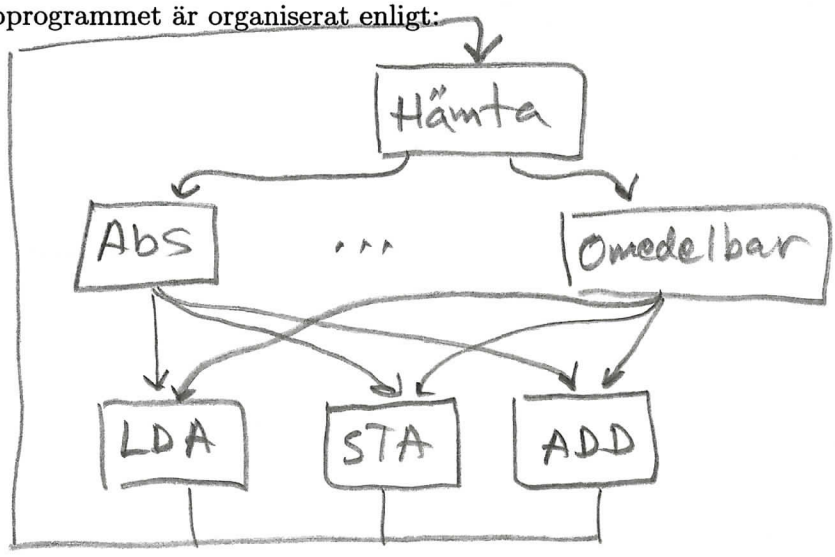
K1(1) = 34

```
34 : ar -> dr, mpc++                37,6,11
35 : adr -> minne <- dr, 0 -> mpc   2,4,5,12
```

8.7 Vad hände egentligen?

Innan vi fortsätter kan det vara läge att göra en snabb återblick på vad som egentligen hänt. Vi har skrivit en massa mikrokod — men inte huller om buller som det kanske kan tyckas. Mikrokodens upplägg i mikrominnet är väl organiserat. Det är lätt att inte se strukturen på grund av alla siffror och manipuleringar.

Mikroprogrammet är organiserat enligt:

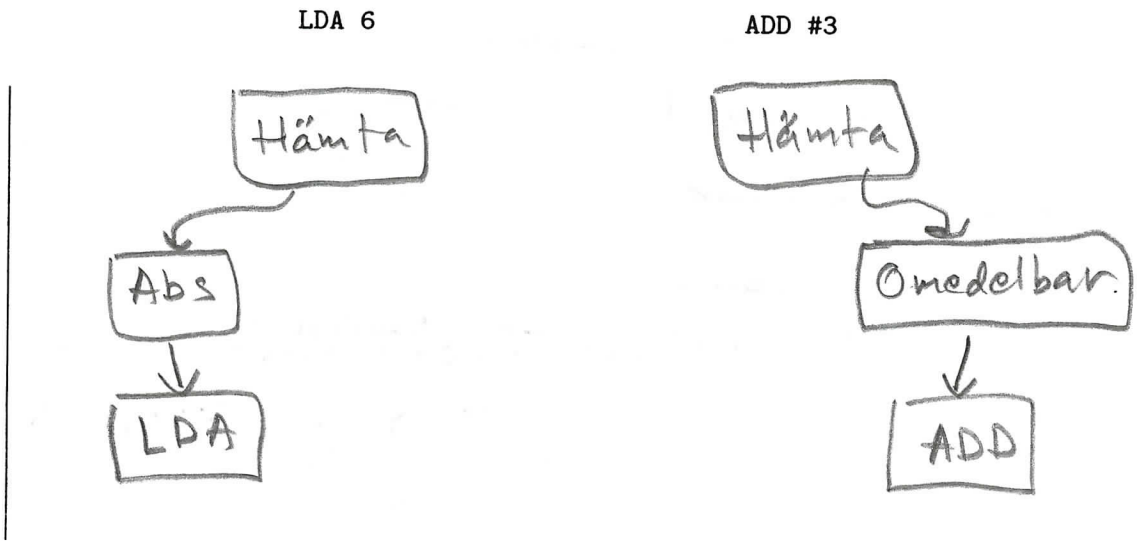


Där man speciellt ska se att de tre nivåerna överensstämmer med våra steg 1, 2 och 3: *hämtfas, adressmodsfas* och *exekveringsfas.*

Olika instruktioner använder olika delar av mikroprogrammet beroende på

- vilken instruktion det handlar om, och
- vilken adressmod som använts.

Exempel



■

Man kan alltså se att olika operationer beroende på adressmod går olika vägar i mikroprogrammet på sin väg till exekvering och att slutligen $0 \rightarrow MPC$ för att hämta en ny instruktion. Och så vidare, och så vidare, och så vidare — till strömmen bryts. Håll ordning och skilj på assemblerprogram å ena sidan och mikrokod å andra. *Maskinkod* är de instruktionsord mikromaskinen ser, dvs de binära mönster som bygger upp opkoden.

8.8 Mer mikrokod — I/O-instruktioner

Vi måste komplettera vår modelldator med möjligheter för in- och utmatning av yttre data. Vi har hittills diskuterat två metoder att orkestrera I/O:

- programstyrd I/O (*pollning*), och
- avbrott

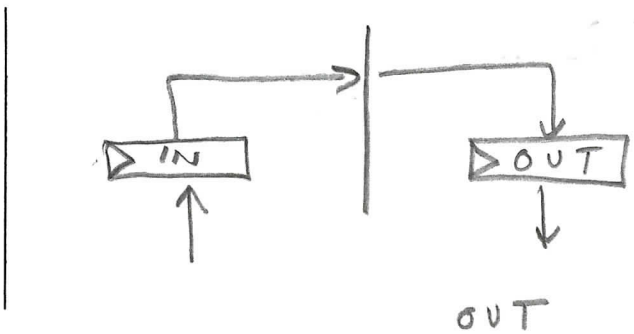
Det finns även ett tredje sätt, DMA, *Direct Memory Access*, vilket innebär att processorn frånsäger sig bussen och låter någon annan enhet använda hela bussen för att överföra data. Om processorn skulle vara inblandad i dataöverföringen måste den läsa in datat till något register innan den strax efteråt ska spara datat någon annanstans. Det tar uppenbarligen en hel del tid. Om processorn däremot inte lägger sig i bussaktiviteten kan den yttre enheten använda hela bussen[9] ensam för att köra över data till minnet eller annan hårdvara. Vi förstår att detta är ett snabbare sätt att forsla data i stora mängder, och inser samtidigt att det mycket väl kan vara krångligt att själv administrera denna DMA-hantering. Till vår oförställda glädje upptäcker vi därför att det finns speciella DMA-kretsar framtagna för detta.

I vår nuvarande processor tar dock vi bara hänsyn till I/O med två instruktioner, de tidigare presenterade, `IN` och `OUT`, vars funktion antas vara enligt nedan.

Instruktion	Betydelse
`IN`	$AR := IN$
`OUT`	$OUT := AR$

Vi måste förse vår konstruktion med 8-bits in- och utenheter samt införa mikrosignaler för att styra dessa. Vi kallar mikrosignalerna 43 och 44 för IN repektive OUT.

Vi måste naturligtvis ha mikroprogram som styr dessa. Vi väljer att lägga det första, för `IN`, på rad 50[10], och mikroprogrammet för `OUT` omedelbart därefter:



A-mod underförstådd
29 : K2 → mpc 9

IN
50: in → tr, mpc++ 43,25,11
51: tr → ar, mpc++ 27,32,33,11
52: status, 0 → mpc 34, 12

OUT
53: ar → out, 0 → mpc 37,44,12

8.9 Avbrott

Avslutningsvis skall vi också implementera avbrott i vår processor. Ett (yttre) avbrott kan anlända när som helst. För att göra det enkelt för oss väljer vi att inte hoppa till avbrottsrutinen förrän pågående assemblerinstruktion slutförts. Så gör man också i alla andra processorer.[11]

Vi vet sedan tidigare att processorn vid avbrott utför i princip ett subrutinanrop. Det är upp till processorkonstruktören — oss, i detta fall — att avgöra exakt vad som ska hända. Den första frågan man som konstruktör måste ställa sig är *när* man skall lyssna efter avbrott. Eftersom processorn slaviskt följer mönstret *hämta–avkoda–utför* kan man

[9] Och det handlar nu förstås om den *yttre bussen*.

[10] Vad, i mikroprocessorn, avgör att det ligger på rad 50? Hur vet mikroprocessorn om det?

[11] Fundera på vad som skulle krävas av hårdvaran om ett avbrott skulle kunna avbryta pågående mikroinstruktion.

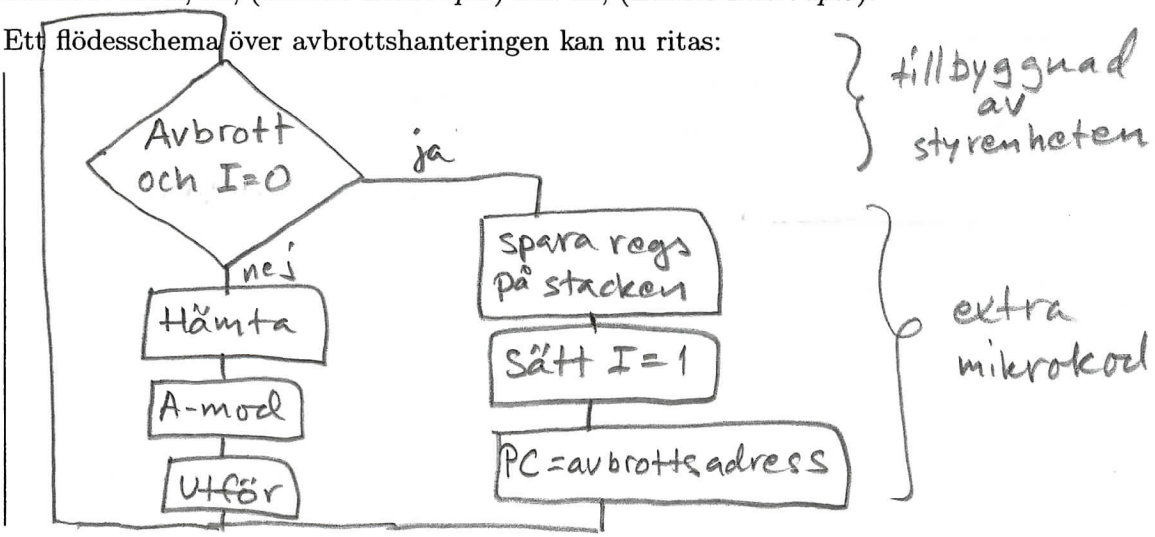
tänka det är naturligt att kolla om avbrott inträffat innan nästa assemblerinstruktion påbörjas, d v s innan nästa instruktion hämtas.

Här bestämmer vi att denna processor hanterar avbrott enligt följande schema, i tur och ordning:

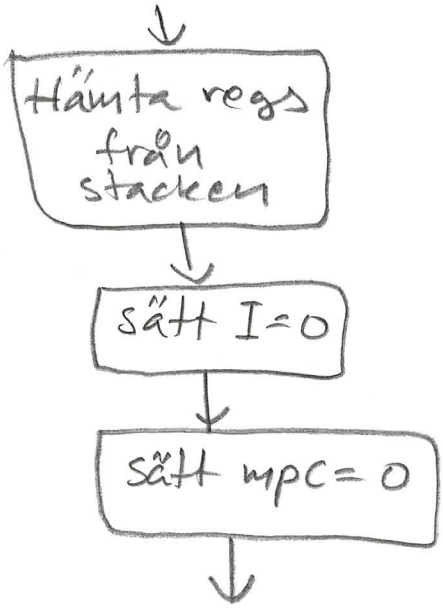
1. Avsluta aktuell instruktion.
2. Spara processorns *inre tillstånd*. Som mikroprogrammerare kan vi här välja att underlätta för assemblerprogrammeraren genom att inte bara spara undan programräknaren, PC, och statusregistret, SR, utan även t ex indexregistret, XR, och ackumulatorregistret, AR.
3. Sätta spärrvippan, så att ytterligare avbrott förhindras.
4. Slutligen, hoppa till avbrottsrutinen, som ligger på en bestämd adress i *assembler*minnet.

För spärrvippans skull inför vi nu också två nya instruktioner som kan hindra respektive tillåta avbrott, `DI`, (*Disable Interrupts*) och `EI`, (*Enable Interrupts*).

Ett flödesschema över avbrottshanteringen kan nu ritas:



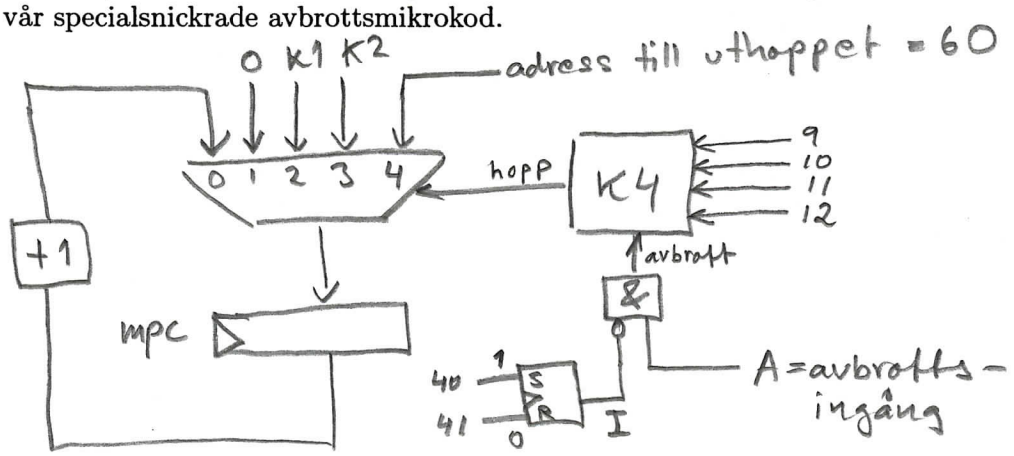
Parat med hoppet till avbrottsrutinen inför vi också återhoppsinstruktionen `RTI`, (*Return from Interrupt*), som måste vara omvändningen till flödesschemat ovan:



8.10 Hårdvarumodifikation för avbrott

Innan vi kan börja mikrokoda måste hårdvaran förädlas ytterligare lite. Vi måste införa en *avbrottsingång* och en spärrvippa, SV, för att kunna slå av respektive slå på ytterligare avbrott. Vi använder konventionen att en ett-ställd vippa hindrar ytterligare avbrott. För att inte missa ett avbrott inför vi också en *avbrottsvippa* som är ett-ställd om någon ryckt i avbrottsingången och nollställd annars.

Vid nollställningen av MPC, d v s vid början av en ny hämtfas, är det alltså lämpligt att låta avbrottsvippan ha sitt inflytande över exekveringen av mikrokoden. Vi inför nätet K_4 för detta. K_4 använder avbrottsvippesignalen, den "gamla" nollställ-MPC-signalen och information från spärrvippan för att avgöra om MPC skall nollställas (=inget avbrott tillåts) eller laddas med en speciell mikrokodsadress som styr över exekveringen till vår specialsnickrade avbrottsmikrokod.



9	10	11	12	a/b	hopp
1	0	0	0	–	2
0	1	0	0	–	3
0	0	1	0	–	0
0	0	0	1	0	1
0	0	0	1	1	4
övrigt					–

Skilj på avbrott i mikrokod och assembler. I båda fallen styrs exekveringen över till en avbrottsrutin, men den ligger i ena fallet i mikrokod och i det andra i assembler.

K_4 ger normalt bara nollställkommandot till MPC men om

- spärrvippan är nollställd *och*
- avbrottsignalen är ettställd *och*
- hämtfas är på gång

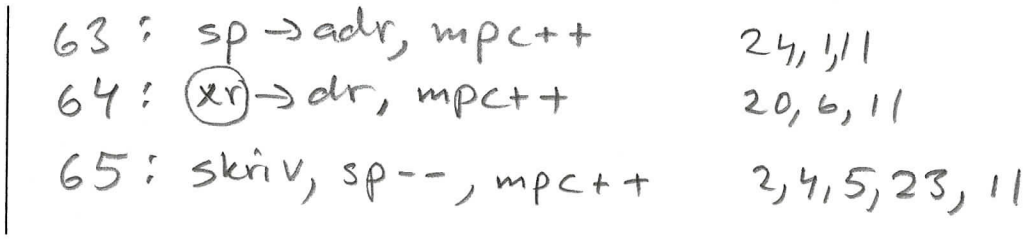
levererar nätet mikroadressen till vår avbrottsrutin, som vi väljer att lägga på adress 60 i mikrokoden.

Om avbrott sker kommer alltså mikrokodsexekveringen att styras över till rad 60. Där ligger nu mikrokoden för avbrottet, nämligen:

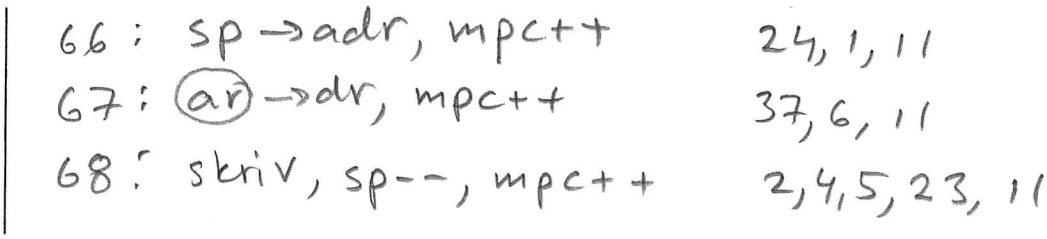
1. Lagra undan nuvarande PC

```
60 : sp→adr, mpc++      24,1,11
61 : (PC)→dr, mpc++     18,6,11
62 : skriv, sp--, mpc++ 2,4,5,23,11
```

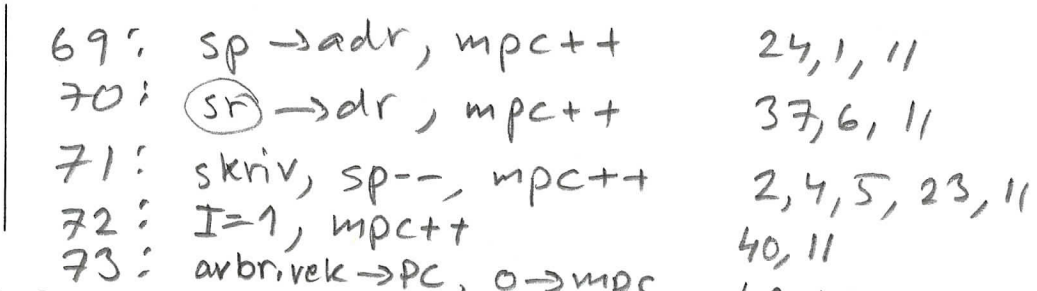
2. Lagra undan XR



3. Lagra undan AR



4. Lagra undan SR



8.11 Optimerad mikrokod

En intressant observation i detta sammanhang är att man kan optimera mikrokod för så vitt datorns arkitektur gör detta fördelaktigt. Så länge de önskade mikrooperationerna inte medför kollision kan de utföras samtidigt. Exempelvis kan mikroinstruktionerna på rad 71 och 72 slås ihop till en enda, som då placeras helt på rad 71:

Rad	Mikrooperation	Styrsignal
71:	skriv, $SP := SP - 1$, DI, $MPC := MPC + 1$	2, 4, 40, 5, 12, 42, 23

8.12 Återhoppsinstruktionen `RTI`

Mikrokoden för assemblerkommandot `RTI` fungerar på omvänt sätt:

- Återställ processorns inre tillstånd, d v s hämta tillbaka det från stacken,
- Möjliggör avbrott på nytt,
- Börja en hämtfas, vi är då tillbaka i det normala programmet igen.

Mikrokoden för `RTI` lämnas som en övning för läsaren. Glöm inte att även fylla i näten K_i i tillämpliga delar.

9 Alternativa arkitekturer

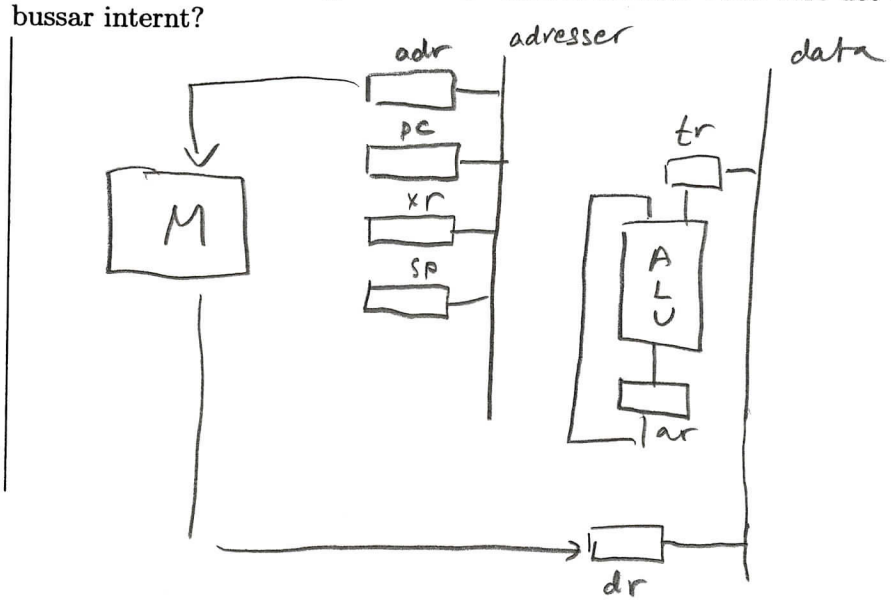
Begrepp i denna föreläsning: *Instruktionsfrekvens, RISC, överlappande* FETCH- *och* EXECUTE-*faser, prefetch, instruktionskö, pipeline, cacheminne,* locality of reference, *direktmappad cache, associativt minne, utbytesalgoritmer, stackprocessor, signalprocessor.*

9.1 Inledning

Vaddå alternativa? Har vi inte lärt oss rätt hittills? Frågan kan besvaras med ett "nja". Det som beskrivits hittills är inte fel. Det är i huvudsak så som en dator eller mikroprocessor såg ut från 1953 och framåt. 1953 uppfanns mikrokoden av Maurice Wilkes i England. Det var en revolutionerande idé — kanske för revolutionerande, det tog bortåt tio år innan IBM slutligen anammade mikrokod fullt ut. Men det visade sig att det var så oerhört mycket enklare att programmera med mikrokod än att inte göra det, så metoden med mikrokod blev den absolut förhärskande fram tills början av 1980-talet då nya tankar vann insteg.

Innan de nya tankarna släpps fram vill jag visa på hur vår modelldator har brister i några avseenden:

- Vi har skrivit tillräckligt mycket mikrokod vid det här laget för att förstå att det är bra om man kan utföra flera mikroperationer samtidigt. D v s ha många siffror på samma rad i mikrokoden. Då kan man få mycket gjort parallellt i processorn. Modelldatorns arkitektur är inte konstruerad med avseende på att utföra parallella (mikro)instruktioner.
- Det finns bara en buss och alla data måste trafikera denna, det kan inte ske samtidigt. Enbart en kan vara sändare på bussen vid varje tillfälle annars blir det krock: Bussen vet inte vilken signal den ska vidarebefordra. Visst vore det bra med flera bussar internt?

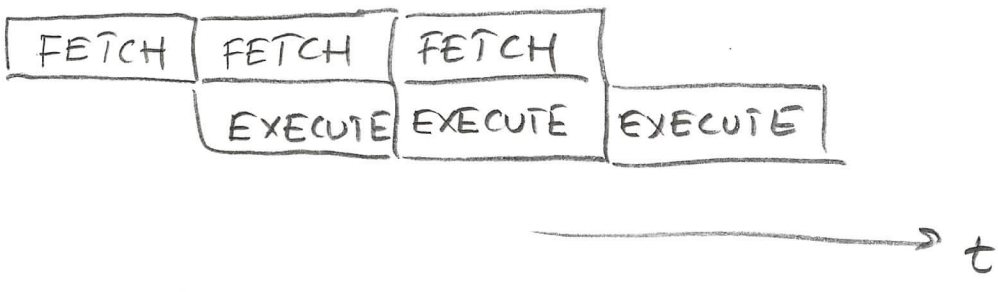


- Studera vilka adresseringsmoder som behövs. Vilka använder kompilatorn? Välj bara de enklaste och snabbaste och de mest använda.
- Välj en bredare instruktionsbredd, 16 eller 32 bitar eller t o m ännu mer. Detta drar ner antalet minnesåtkomster både för operander och instruktioner.
- Välj ett (1) instruktionsformat, detta förenklar hårdvaran som sköter avkodningen av instruktionerna.
- Skippa mikrokoden så långt det är möjligt. Avkoda direkt med logik. Frånvaron av många instruktioner och adresseringsmoder gör att detta plötsligt känns realistiskt. (Kanske ska hämtfasen vara i mikrokod men inget annat eller så?).

Anammar man punkterna 1 till och med 4 ovan kommer man rätt nära vad forskare på Berkeley-universitetet gjorde i början på 1980-talet. Studierna resulterade i en processor som de kallade RISC I (*Reduced Instruction Set Computer*). Ett namn som sedan dess har stått för en egen filosofi inom datorarkitetkturen. För att kallas RISC bör processorn ha de egenskaper som angavs ovan.[4]

RISC 1 hade en 2-stegs pipeline:
1) hämta
2) utför

RISC-ettan hade också en mycket enkel finess som gjorde att prestanda dubblades, nästan. Man *överlappade* FETCH- och EXECUTE-faserna.[5]

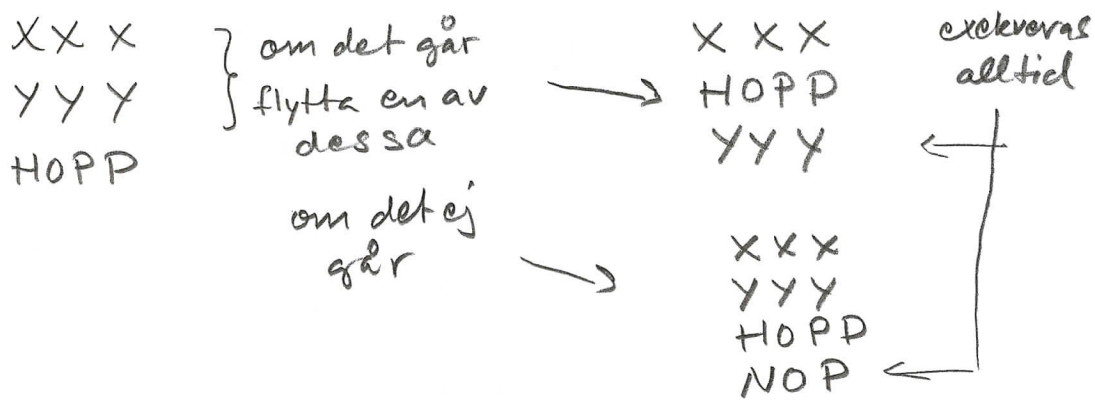


Prestanda fördubblas eftersom EXECUTE-hårdvaran har något att göra i varje klockcykel. Det ställer dock till problem vid hopp. Nästa instruktion är ju redan hämtad när hopp meddelas. Var ska vi göra av den instruktionen? Den är redan inne i processorn och

[4] Dessutom I. Hade den instruktioner av typen ADD r1, r2, r3 för att minska minnestrafiken. Dessutom II. Hade den överlappande registerbanker, dvs gott om register internt.

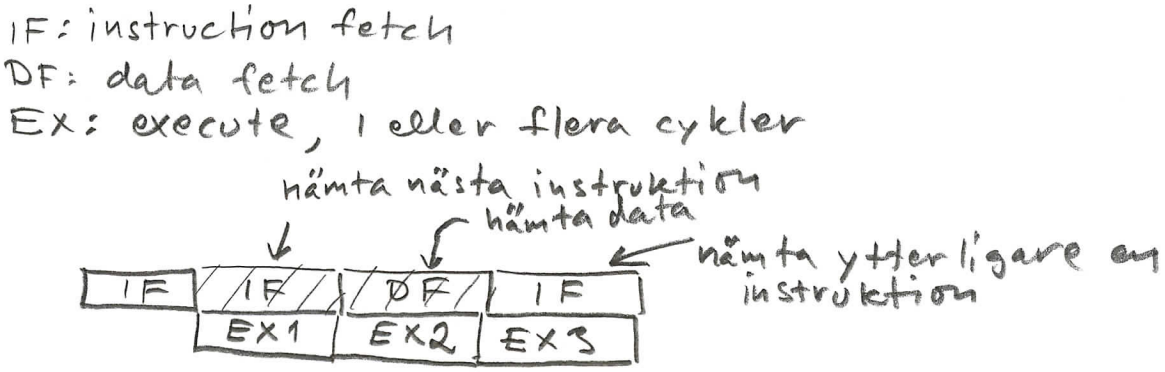
[5] Även 68008 har i viss mån överlappande FETCH och EXECUTE-faser.

kommer att exekveras nästa klockcykel, det går inte att undvika! Paniken breder ut sig! Lösningen i det här fallet är lyckligvis enkel. Byt ordning på hoppinstruktionen och instruktionen före den, och om det inte går, lägg till en NOP innan hoppet. Det här är en enkel sak för en kompilator att ta hänsyn till så vi behöver inte bekymra oss om att det kanske verkar bakvänt.



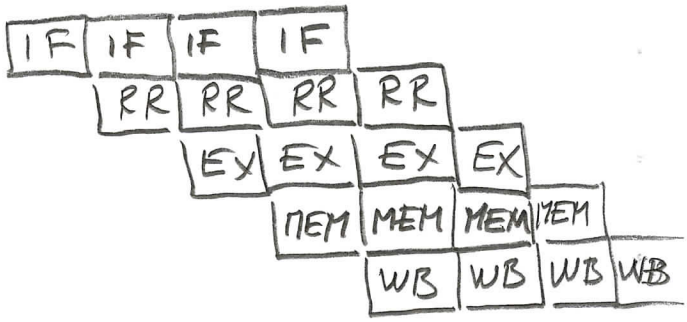
Det här kan också kallas för *prefetch* eller förhämtning. Ty varför ska ALU:n bara användas varannan eller var tredje fas? Det blir uppenbart effektivare om den har något att tugga på i varje fas. Ska vi nå fram till en instruktion per klocka måste varje fas nyttjas maximalt.

Prefetch verkar bra. Men vad händer om vi inte har plats på bussen att hämta en instruktion? (Vi förutsätter en von Neumann-arkitektur så både adresser och data måste dela på samma buss.) På samma sätt kan man tänka sig att processorn ibland inte pratar med yttre minne alls på flera klockcykler. Vad vore bättre än att utnyttja denna tid till att läsa in *flera* av kommande instruktioner? Inget. Naturligtvis gör vi så:



Drar vi det här resonemanget till sin spets kommer vi fram till vad som kallas överlappning eller, vanligare, *pipelining*. Principen och målet är att se till att alla delar av processorn har något att göra vid varje klockpuls. Alltid.

IF = instruction fetch
RR = register read
EX = execute
MEM = memory access
WB = write back reg.



9.4 Problem vid pipelining

Pipelining kommer dock inte ostraffat. Det är lätt att interna kollisioner uppstår. Om t ex två (eller fler) av våra faser vill komma åt minnet eller någon annan resurs samtidigt måste detta hanteras på något sätt. Vi kan lätt urskilja tre scenarion med strul:

- Problem 1. Minneskonflikt

Vi måste ha skilda minnen för
- instruktioner
- operander

Det brukar kallas Harvard-arkitektur

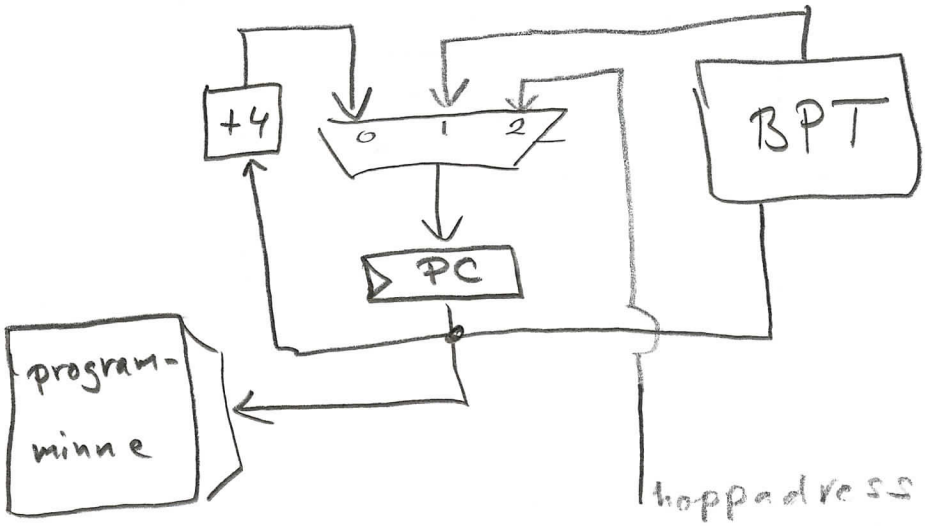
- Problem 2. ALU-konflikt

ALUn kan behövas för
1) vanlig aritmetik mellan operander
2) adressaritmetik för indexerad adressering
3) — " — för relativa hopp

I lab-pipelinen sker 1 och 2 i ALUn och 3 i en separat adderare.

- Problem 3. Villkorliga hopp.

  Detta fall liknar det vid förhämtning men nu vet man inte i förhand om hoppet skall genomföras eller inte. Har vi otur måste hela pipelinen tömmas. Om vi i förväg vet, eller hyggligt bra *kan gissa*, om hoppet ska tas eller inte kan vi undvika tidspillan.

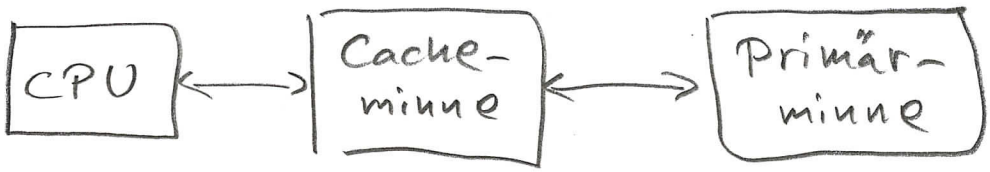


Branch Prediction Table, BPT, löser man enklast med s k *Associativa minnen*

9.5 Cacheminne

Avbrott i exekveringen är alltså en styggelse som måste bekämpas! Vi ser att en BPT kan vara åtminstone en del av lösningen. En annan del av lösningen är cacheminnet (kallas också för fickminne):

Cachen är ett litet minne som placeras mellan processorn och primärminnet:



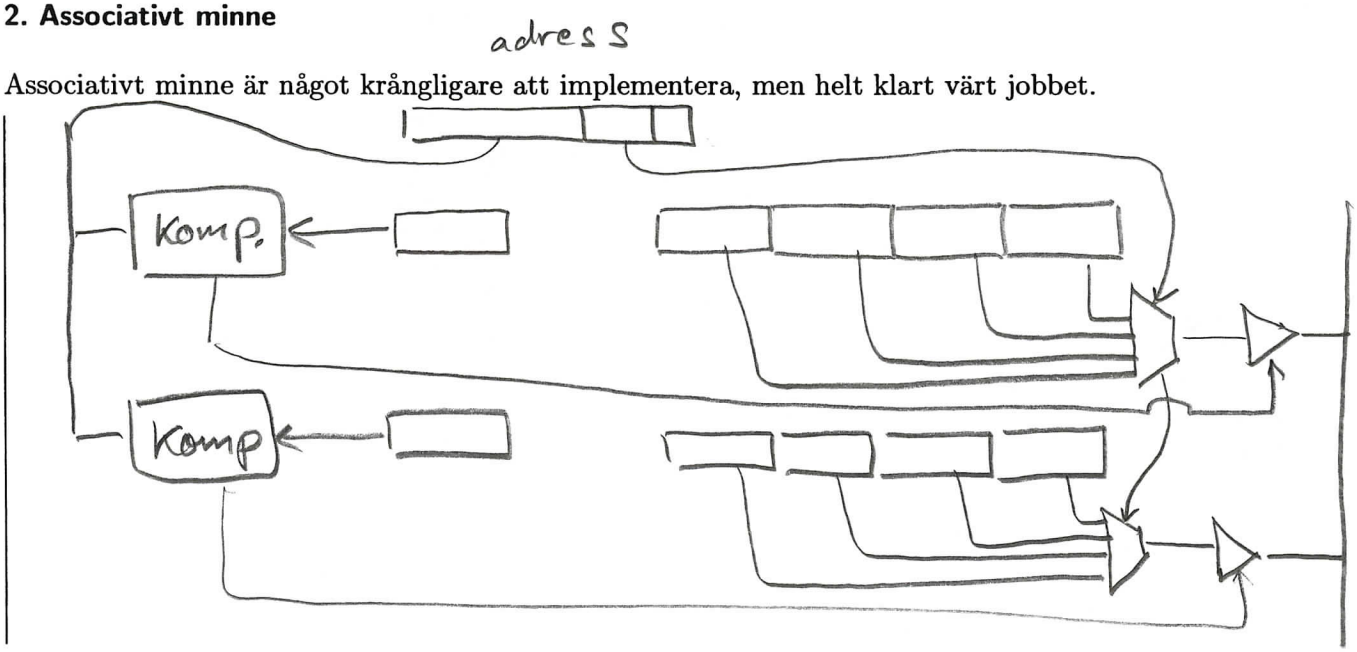
De instruktioner och det data som används mest placeras automatiskt[6] i cacheminnet och är snabba att ta fram igen. Man definierar accesstiden som tiden att komma åt ett datum ur minnet, d v s kombinationen av cacheminne och primärminne, som

[6]Programmeraren kan inte avgöra detta, cacheminnet ingår i processormekanismen och är transparent för användaren.

- Sök efter adress 1. Finns i cachen. Tag detta data.

Det är ju synd att behöva skriva över rad 00 när vi nu så omsorgsfullt hämtat det från PM. Det är naturligtvis inte bra. Det är sannolikt (*locality of reference!*) att adress 0 kommer att behövas snart igen. Då var det ju dumt att skriva över den raden i cachen. Lösningen heter *associativt minne.*

2. Associativt minne

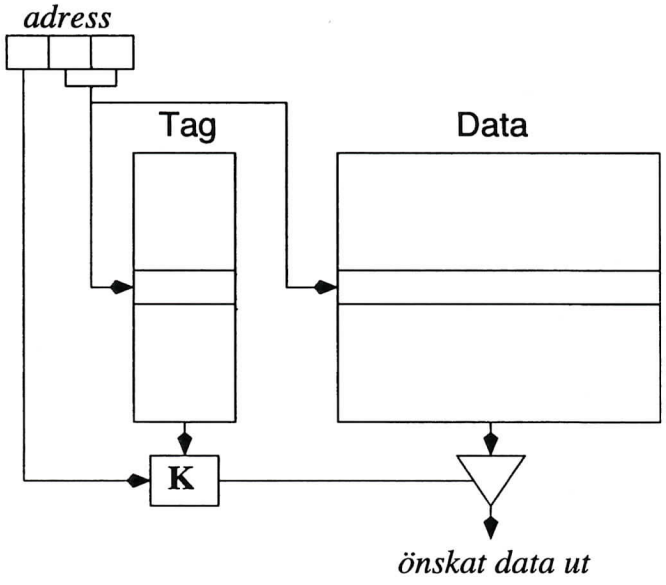


Associativt minne är något krångligare att implementera, men helt klart värt jobbet.

Nu jämförs *hela* adressen med den s.k. "tag"-en. Det är naturligtvis så här vi vill ha det egentligen. Men det kostar kisel.

Hårdvaran för metod 1 och 2 skissas nedan:

- Hårdvara för direktadresserat minne



10.2.3 EPROM, Erasable Programmable Read Only Memory

EPROM är både användarprogrammerbara och användarraderbara. Programmeringen sker genom att ladda upp eller ur kondensatorer i kretsen. Varje minnesbit information kräver sin egen kondensator. Man förstår att det är av högsta vikt att de elektroner som utgör kondensatorladdningen inte läcker ut utan håller sig i sin kondensator. Detta kräver en smått fantastisk grad av isolation mot omgivningen då laddningen numera kan vara endast något dussin elektroner. Det är inte ovanligt att tillverkaren lovar att elektronerna håller sig i kondensatorn i minst 40 år!

Minnena raderas genom belysning med ultraviolett ljus av hög intensitet. Detta sparkar ut elektronerna ur kondensatorn och tömmer minnet (faktiskt '1'-ställer det bitinformationen). Solljus har tillräcklig andel ultraviolett ljus för att radera EPROM, men det krävs veckor av bestrålning. EPROM raderas med ljus från särskilda lysrör, med för oss skadlig strålning, och kännetecknas av det titthål av glas de är begåvade med för att raderljuset ska kunna komma in. Efter programmering brukar man klistra fast en lite täcklapp på "fönstret" för att undvika överraskningar om ströljus trots allt råkar belysa komponenten.

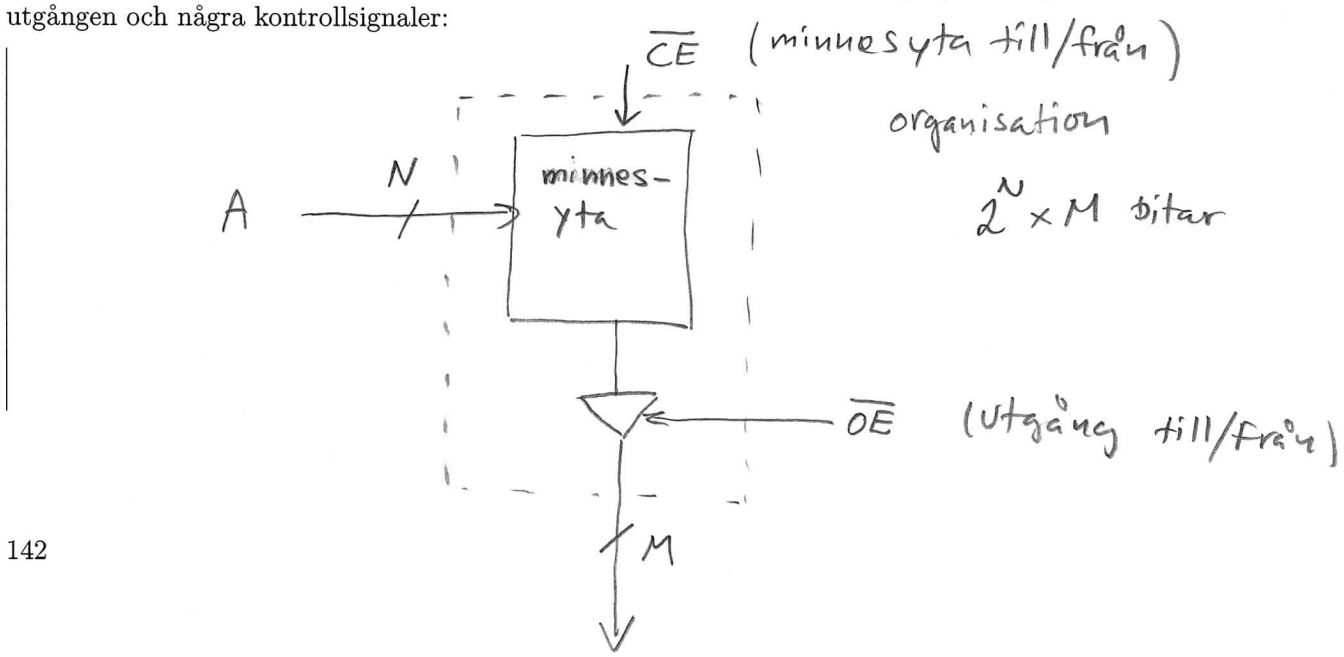
EPROM används numera även också som PROM. Det enda som skiljer är att tillverkaren för de senare väljer en kapsel utan "titthål". Lyckas man borra hål i kapsel kan dock radering medelst belysning ske!

10.2.4 EEPROM, Electrically Erasable Programmable Read Only Memory

Detta PROM är både elektriskt raderbart och programmerbart. En egenskap som man drar nytta av om man vill programmera om en apparat "i fält". Det finns exempelvis radioapparater som egentligen bara styrs av en stor digital signalprocessor. För att snabbt få genomslag på marknaden görs den helt mjukvaruprogrammerbar. Om den skulle levereras med något fel i programmet kan användaren tanka ner uppdaterad mjukvara från Internet. Modernare processorer för inbyggnad brukar innehålla åtminstone några bytes EEPROM för konfigurationsinformation.

EEPROM används inte bara i minnen: det finns åtskilliga programmerbara logiska kretsar, PLD eller FPGA, som också baserar sig på EEPROM för konfigurationen.

När dessa minnen väl är programmerade består de i huvudsak av minnesyta, buffer på utgången och några kontrollsignaler:



10.3 RWM, Read-Write Memory

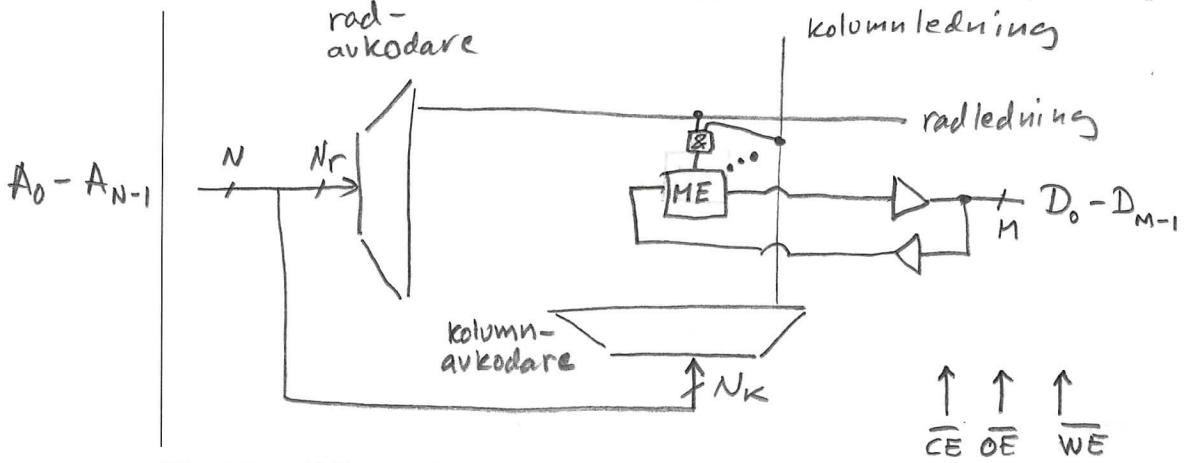
RWM är minne som användaren kan programmera och radera. Radering sker helt elektriskt, man slipper använda ultraviolett ljus. Bland RWM:erna kan främst nämnas olika varianter av RAM, *Random Access Memory*[1].

10.3.1 RAM, (Static) Random Access Memory

När vi tänker oss ett minne är det nog RAM som vi automatiskt har föreställer oss: Lägg på en adress, läs, och datat ploppar ut, eller: lägg på adress och data, skriv, och datat fastnar därinne. Minnet säges vara av typen *random access*, slumpmässig åtkomst, eftersom man inte behöver komma åt datat sekvensiellt, som om datat ligger på magnetband till exempel. På ett magnetband måste ju bandet spolas fram/back till rätt ställe innan läsningen eller skrivningen kan påbörjas.[2]

Det RAM som är enklast att använda är det s k *statiska* RAM:et. Statiskt eftersom det behåller informationen när det väl är ditskrivet, såvida inte spänningen till kretsen förloras, då försvinner datat också. Vid spänningspåslag blir RAM-innehållet helt slumpartat blandade ettor och nollor, det skulle vara för dyrt att införa en generell reset på alla minnescseller — och strängt taget onödigt också.

I det statiska RAM:et, SRAM:et, lagras informationen i en vippa och för att peka ut den individuella minnescellen adresseras dessa vippor av en adressavkodare, en multiplexer:



För att inte få långa avkodningstider, dvs många logiknivåer i adressavkodaren, tillverkar man vipporna i en kvadratisk matris och delar upp adressavkodningen så att man pekar ut rad och kolumn samtidigt.

[1] Innan RAM fanns egentligen bara minnen på magnetband eller exempelvis i långa glasrör fyllda med kvicksilver där man höll en puls studsande mellan väggarna för att markera en logisk etta. Kvicksilver användes för att får rätt akustisk impedans mot glaset så att studsarna blev så tydliga som möjligt. Innan dess använde man bl a efterlysningstiden på en oscilloskopskärm som indikator på etta eller nolla. Tack och lov är den tiden förbi!

[2] Se läroboken för mer om lagring på magnetband.

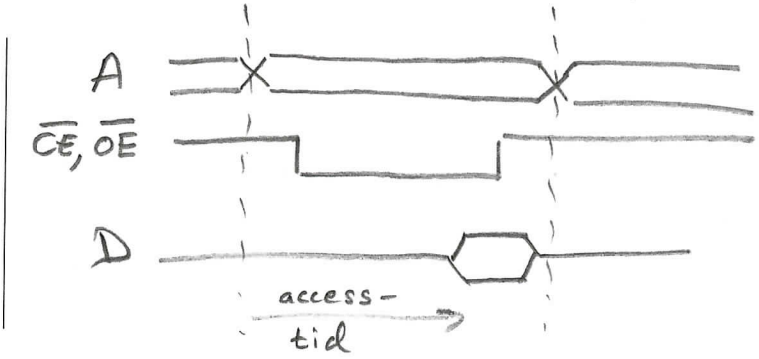
Tyvärr läcker elektronerna ut från kondensatorn efter en stund, på några millisekunder kan det vara så få kvar att man kommit in i den digitala gråzonen där kondensatorspänningen inte räcker för att skilja digital etta från digital nolla. I kretsarna finns därför logik som regelbundet läser av varje minnescell och skriver in innehållet på nytt, en mekanism som kallas *refresh.* Det är vanligt att refreshen måste initieras, eller till och med helt styras, utifrån. Om minnet inte får sin refresh inom föreskriven tid kommer minnesinnehållet att försvinna.

Refresh är lite olyckligt eftersom minnet, i varje fall emellanåt, är upptaget med att genomföra refresh och så att säga "se om sig själv" och kan alltså inte alltid leverera data lika snabbt hela tiden. Lösningen för detta är speciella refresh-kretsar som sprider ut refreshcyklerna i tiden, tjuvlyssnar på minnestrafiken och sticker in en refresh här och var, till exempel när minnet just använts — det är antagligen ingen som vill prata med minnet precis efter en minnescykel. Ett förfarande som kallas *hidden refresh.* Refreshkretsen innehåller även en tidmätare som kan tvångsköra en refreshcykel som en sista åtgärd för att bevara informationen om kretsen inte fått möjlighet att göra en refresh på ett tag.

Både SRAM och DRAM tappar innehållet vid spänningsfrånslag.

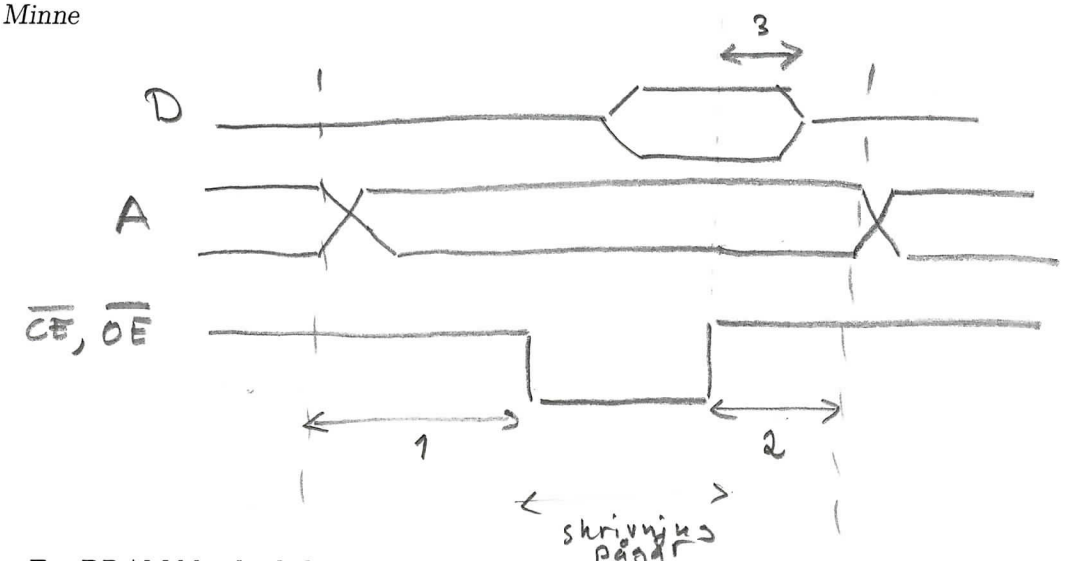
För att läsa/skriva till ett minne måste man göra på olika sätt beroende på om det är ett PROM, SRAM eller DRAM:

- PROM är det enklaste, här finns bara en läscykel:

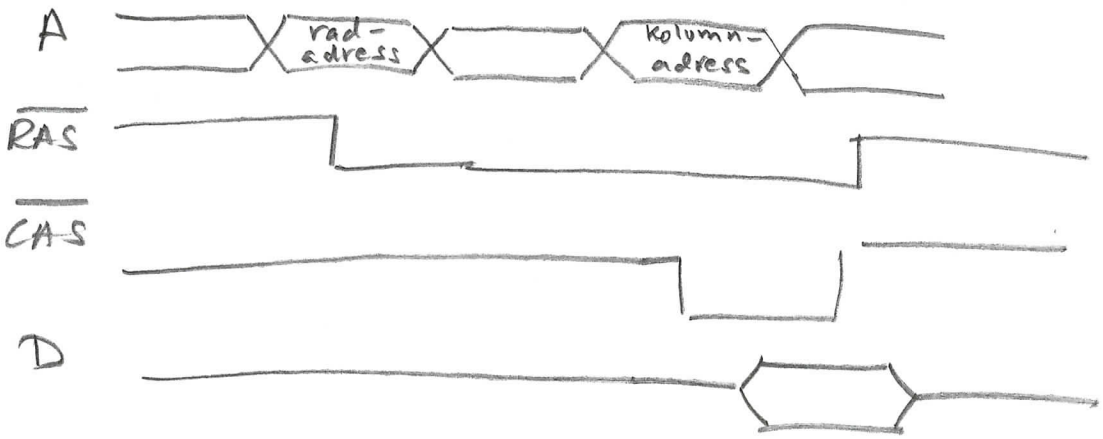


- För SRAM kan man genomföra både en läs- och en skrivcykel:

1, 2, 3 måste alla vara > 0



- För DRAM blir det hela mer komplicerat eftersom adressen delas upp i två delar, en rad- och en kolumnadress, man *multiplexar* adressen:

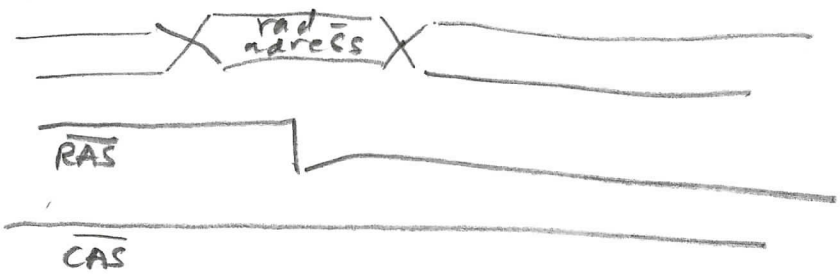


- DRAM har dessutom en refresh-cykel:

Moderna DRAM är burstorienterade, dvs efter första datat kommer nya data på varje klockcykel (typiskt en cacheline)

Typisk refreshtid 64ms ← alla rader!
Minnet refreshas rad för rad

RAS-only med yttre räknare



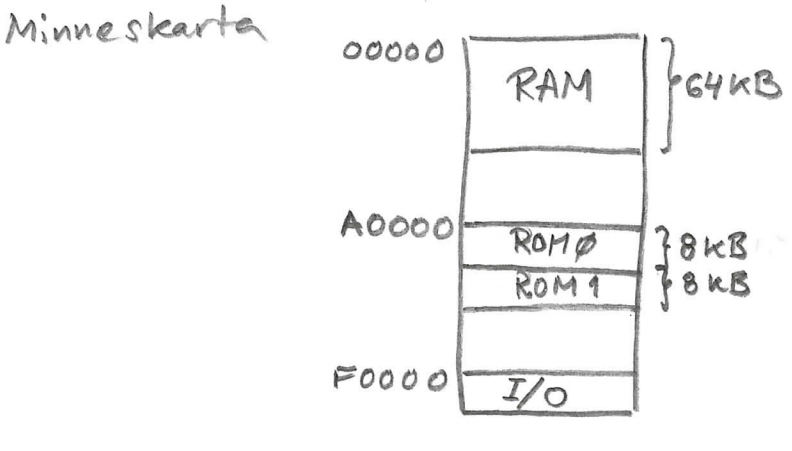
10.4 Ett verkligt exempel

För att visa på hur minnen används och kopplas in ska vi titta på ett faktiskt kopplingsschema på ett 68008-baserat processorkort som det beskrivs i Motorolas *Application Note 897*, AN897, sidan 153. Tillverkarna av kretsar ger ut sådana application notes för att folk ska förstå hur bra kretsarna är och hur de kan användas. I det här fallet beskrivs ett minimalt processorsystem bestående av en 68008-processor, RAM, ROM och en *Dual Universal Asynchronous Receiver Transmitter*-krets, DUART. Den senare gör att man kan koppla en seriell enhet, i detta fall en terminal, till processorn. Nu intresserar vi oss bara för minnet. Det står en massa om asynkron bussöverföring också i texten, vad det är kommer att klarna senare. För tillfället antar vi bara att det fungerar — på något sätt.

För att processorn ska kunna hämta en instruktion måste motsvarande adress läggas ut på adressbussen, precis som i mikrokodsfallet. 68008 har adressbussen A19–A0, där A0 är minst signifikant bit. 68008 pekar alltid ut enbart bytes till skillnad från sin storebror 68000 som också kunde peka ut ett word om 2 bytes.[3] När adressen pekat ut instruktionen hämtas denna från programminnet för vidare bearbetning. Det hela känner vi igen som hämtfasen i mikrokodsfallet.

Det kluriga är nu, hur respektive minneskomponent vet att just den är tillfrågad; att vi ska läsa instruktioner från PROM och inte från RAM, att vi ska skriva en variabel till RAM och inte till PROM o s v.

Det hela löses med hjälp av *minneskartan*, eller *minnesmappen*, som delar upp minnet mellan ROM och RAM. I just detta fall är även DUART:en åtkomlig via minnesadresser. I AN897 har man inte angivit exakt hur minnesmappen ser ut så vi får resonera oss fram:

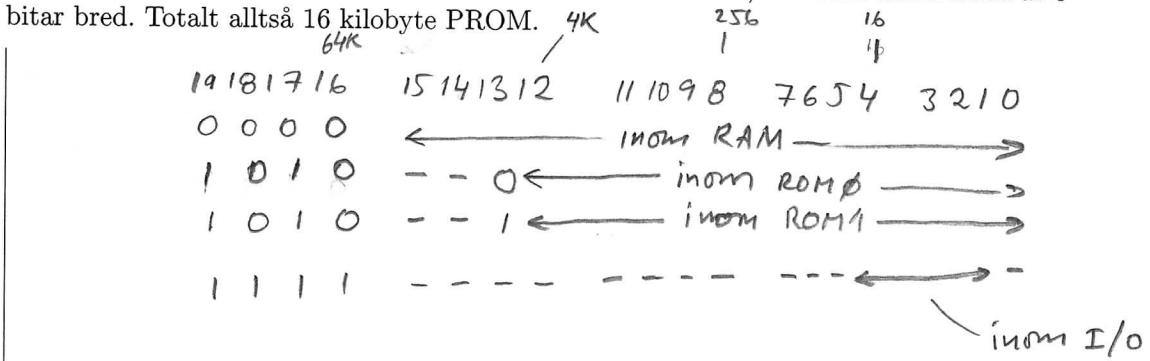


Hemligheten är kretsen U22 som är en programmerbar krets, en PLD (*Programmable Logic Device*). Tricket är att använda de övre bitarna på adressbussen för att peka ut en större minnesyta och de lägre för att peka ut enstaka bytes i denna minnesyta.

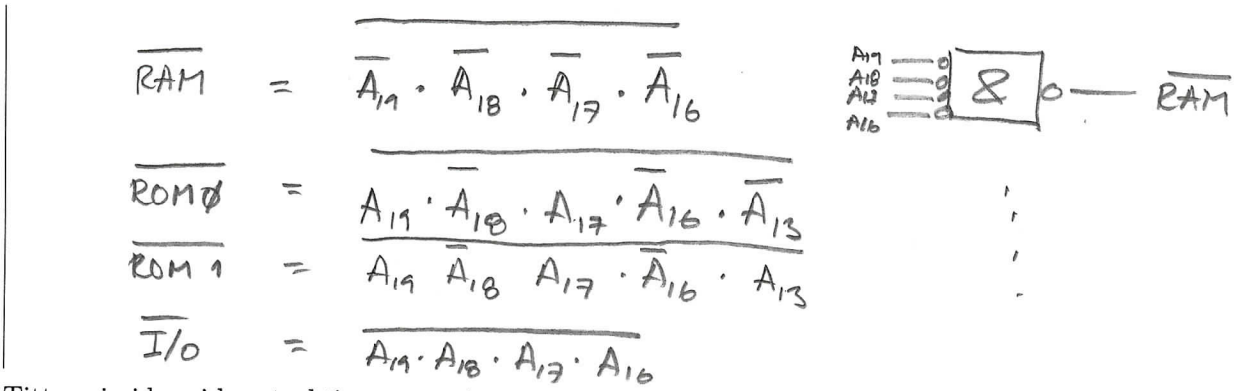
Här är fyra minnesområden avkodade. Ett för RAM, två för ROM och ett för I/O, d v s DUART:en. Övriga pinnar kan användas för att avkoda ytterligare områden i adressrymden för ytterligare minne eller I/O-kretsar.

[3] Vi kommer ihåg att man till och med tagit bort signalen A0 från adressbussen på 68000.

Tittar vi på minnena i sig är PROM:en de lättaste att ge sig in på. U15 och U16 är de PROM-kapslar där processorkortets fasta program[4] är beläget. Vi ser hur en del av adressbussen, 13 bitar, A12–A0 går till var och en av dom. Av detta kan man dra slutsatsen att PROM:en har $2^{13} = 8192$ rader minne à 8 bits, eftersom databussen är 8 bitar bred. Totalt alltså 16 kilobyte PROM.



Båda PROM:en kan få en läsbegäran på sin rad 0 genom att A12–A0 utförs med noll. Kollision på databussen undviks, då signalerna ROM0 och ROM1 inte kan vara aktiva samtidigt, d v s genom att ingångarna på pinne 20 (CS, *Chip Select*) på respektive krets aldrig kan vara aktiva — spänningsmässigt låga — vid samma tidpunkt, se U20 och U27.[5]



Tittar vi vidare i konstruktionen ser vi att man, antagligen av kostnadsskäl, har använt RAM av dynamisk typ här (U3–U10) och för att inte behöva en egen refresh-krets har man tillverkat en förenklad refresh med egen logik. Vi bryr oss inte om denna.

10.4.1 Vilken storlek har RAM-kapslarna?

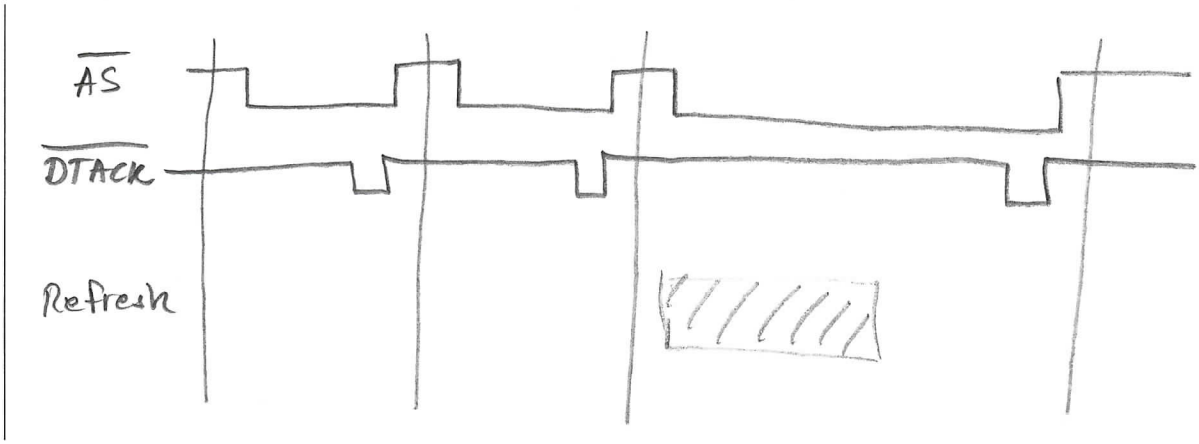
Genom inspektion i schemat ser vi vidare att det bara går ut en databit ur varje kapsel U3–U10, uppenbarligen är kapslarna parallellkopplade för att slutligen erhålla den önskade bitbredden 8 bitar. Precis som förut ser vi antalet rader i varje kapsel genom att titta på vilka adressbitar som används, tydligen 8 stycken, vilket borde ge oss 2^8=256 rader — men då har man inte tagit hänsyn till att minnena är av den dynamiska typen. Förutom de signaler vi känner till sedan förut, går det till varje kapsel två ytterligare signaler: RAS och CAS, *Row Adress Strobe* respektive *Column Adress Strobe*. Detta är,

[4]Det program som i labbarna skriver `TUTOR 1.3 >`.

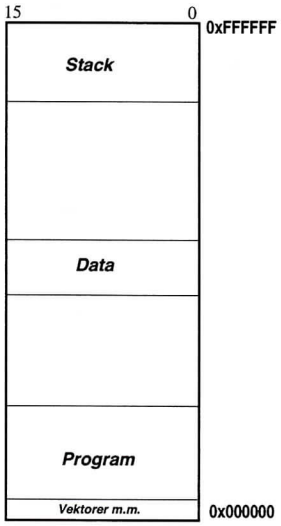
[5]Vad händer om vi inte tar med A15 och A14 i logiken för att peka ut ROM0 och ROM1?

som nämnts, ett vanligt utförande av dynamiska RAM, man *multiplexar* hög och låg byte av en adress. Adressen pekas här ut i *två faser*: först radadressen till minnesmatrisen, sedan kolumnadressen, och man har alltså $2^{8+8} = 65536$ rader i varje kapsel, inte 256 som man skulle kunna tro till en början. (Man blir ju lite misstänksam när kretsarna U1, U2, U11 och U17 sitter iväg för adressbussen på sin väg till RAM-kapslarna, eller hur?) Det är dessa som skickar ut adressen i två omgångar till DRAM:en. Total mängd RAM-minne är alltså 64 kilobyte.

Resten av funktionen får ni fundera ut själva. Det kan då vara bra att känna till följande signaler: Signalen *AS*, *Address Strobe*, är aktiv då processorn lagt ut en giltig adress på sin adressbuss. $R/\overline{W}$, *Read/Write*, är hög vid läsning och låg vid skrivning. $\overline{DTACK}$, *Data Acknowledge*, är en ingång på processorn som används av en yttre enhet för att meddela processorn att den lagt ut data och det är klart för processorn att läsa in datat.

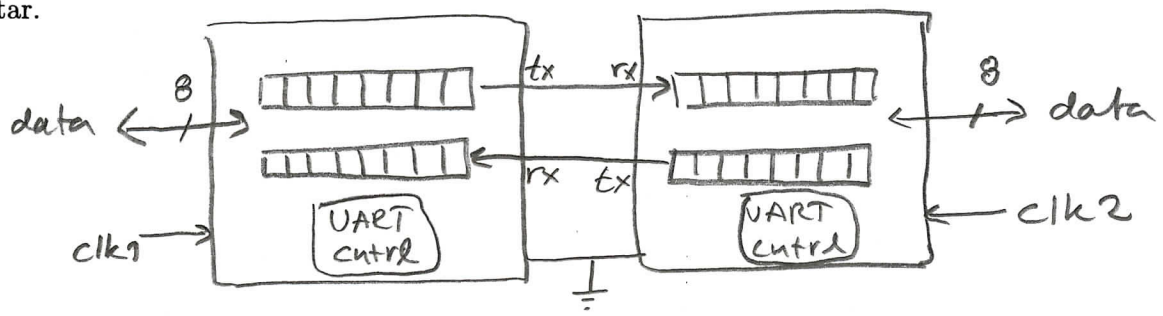


Motorolas minnesmapp är trevlig att ha och göra med. Hela minnet är *lineärt* från adress 0 och uppåt. Det finns dock inget som kräver att minnesmappen skall vara kontinuerligt sammanhängande. Speciellt områden som inte används för program- eller dataminne brukar hänga löst i adressrymden.

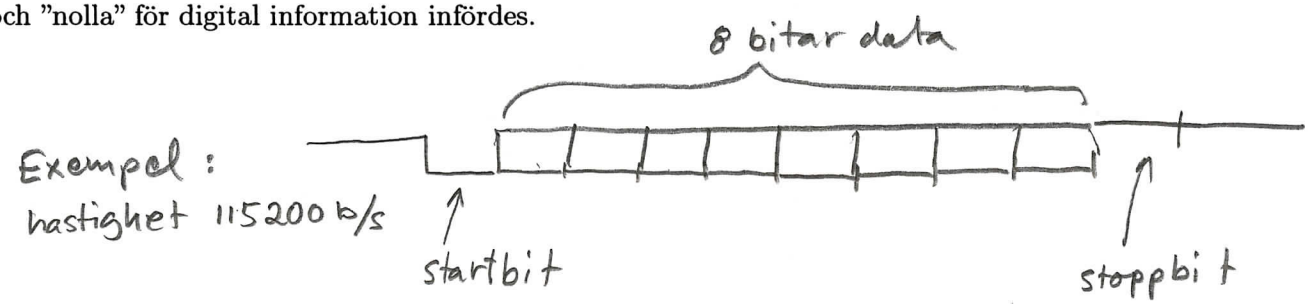


beror på de använda kabellängderna mellan enheterna då en kabel alltid innehåller en viss mängd hindrande kapacitans och induktans vilken till slut omöjliggör en felfri överföring.

Med RS232 överförs informationen bit för bit med möjlighet till paritetskontroll efter 7 eller 8 bitar.



Spänningsnivåerna är +3 – +15V för en SPACE och –3 – –15V för en MARK. MARK och SPACE var etablerade begrepp sedan tidigt 1900-tal långt innan begreppen "etta" och "nolla" för digital information infördes.



Överföringen sker med tecken kodade enligt *ASCII-standarden.* ASCII-standarden (*American Standard Code for Information Interchange*) definierar en teckenuppsättning av alla bokstäver, siffror och en del andra tecken. Den var ursprungligen en lokal USA-standard med en 7-bitars kod med sammanlagt 128 olika tecken men utökades sedan med en bit, till 8 bitar, för att kunna innehålla internationella tecken, bland annat de svenska "åääöÅÄÖ". Medan de första 128 är bestämda "hårt" varierar de övriga beroende på vilket lands teckenuppsättning man använder.

RS232 är en standard för kommunikation på kort håll. Mindre än 15 meter mellan sändare och mottagare rekommenderas, i varje fall vid de högre hastigheterna. Vanliga hastigheter är 4800, 9600, 19200, 57600 och 115200 bitar per sekund.

12.1.2 RS422/485

En senare standard är RS485[3] som genom att införa differentiella (balanserade) spänningar på kabeln lyckas höja prestanda avsevärt. Rekommenderat maximalt avstånd är 1 200 meter och man kan då upprätthålla en datatakt på bortåt 100 kbit/s. Vid kortare avstånd kan datatakten höjas, upp till 10 Mbit/s vid 20 meter. Detta kan man åstadkomma trots att signaleringsspänningen bara är mellan 0 och 5 V tack vare de differentiella spänningarna som ger tydligare flanker på signalen och betydligt högre störmarginal.

[3]Snarlik standarden RS422

stryk

Elektriskt använder RS485, och RS422, ett trådpar i varje riktning, med differentiella spänningar. Med differentiella spänningar menas att det är spänningen mellan trådarna som är informationsbärande, inte någonderas spänning till jord eller 0 V, som i RS232. Genom att använda tvinnad tråd kommer yttre störningar att i det stora hela ta ut varann eftersom störningen induceras lika mycket i varje tråd, och det bara är skillnadsspänningen mellan trådarna som är informationsbärande.

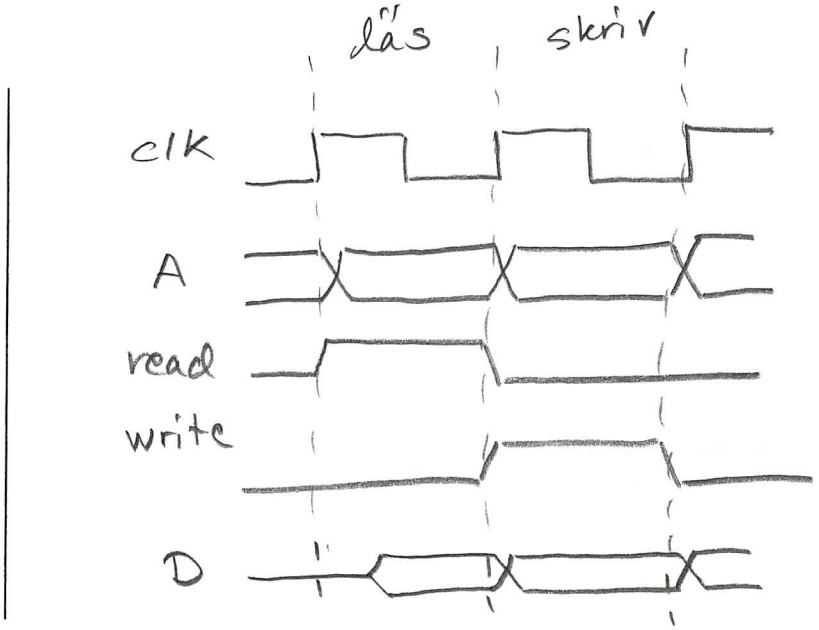
RS485 är konstruerad för 32 enheter på samma buss, med en busstopologi enligt *multidrop*-principen:

stryk

Då det bara finns en (1) buss att signalera på måste ett strikt protokoll följas av alla enheter. Man definierar begreppen *master*, *slave* respektive *sändare* och *mottagare*. Dessa kan kombineras på ett antal sätt och uppfattas ofta som rätt förbryllande. Emellertid fungerar det så här:

En master är den enhet som kan *initiera* en sändning på bussen, alldeles oavsett vilken riktning kommunikationen kommer att utgöra. Mastern adresserar en slav och kan starta en läsning eller skrivning till denna slav. En slav kan aldrig initiera en sändning. Däremot kan den begära att få bli master. Om denna begäran går igenom — nuvarande master signalerar att den släpper kontrollen av bussen eller tråden — kan den efter att ha blivit master få initiera en sändning.

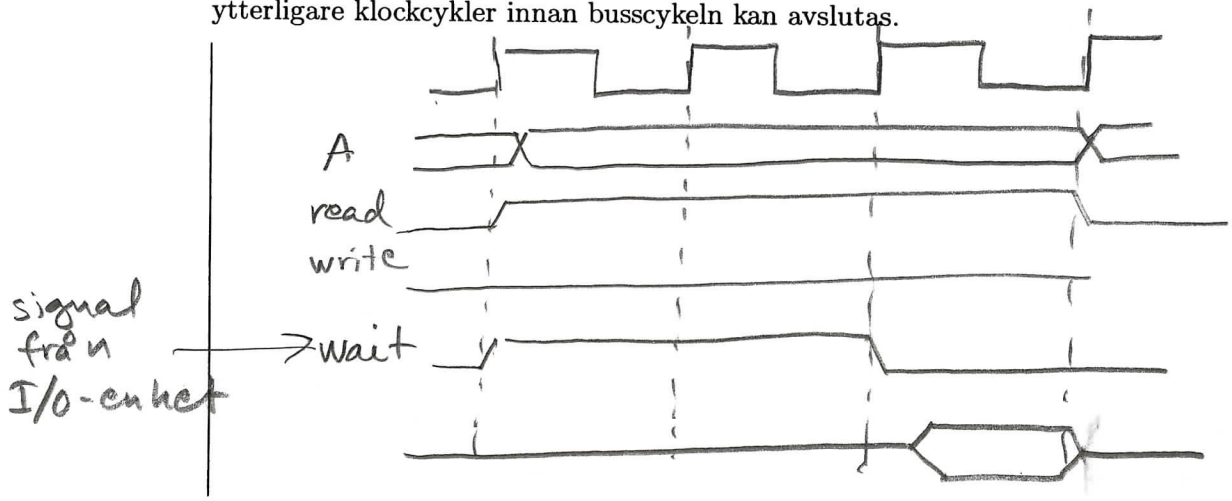
Bussarbitreringen, diskussionen om vem som ska få bli master, är rätt krånglig. Ofta kan man slippa denna om man låter processorn i ett datorsystem att alltid vara master och då se till att den pollar de ingående slavarna tillräckligt ofta för att ingen data ska missas. Man frånhänder sig då naturligtvis möjligheten att slavarna kommunicerar sinsemellan. Informationsflödet går bara från (eller till) mastern och till (eller från) slavarna.



i detta exempel är busscykel = 1 klockcykel

Andra bussar har t ex 1 busscykel = 4 klockcykler

Då snabbare minnen är dyrare än långsamma och andra yttre enheter kanske inte överhuvudtaget finns i de hastighetsklasser som processorns busscykel kräver, finns det intresse att kunna ansluta även dessa långsammare enheter till ett i övrigt snabbt processorsystem. Genom att införa en eller flera vänte-tillstånd, *wait cycles*, kan man låta processorn hänga kvar i busscykeln en extra tid så att den långsammare enheten hinner läsas från/skrivas till. I schemat nedan antas minnet vara så långsamt att det krävs två ytterligare klockcykler innan busscykeln kan avslutas.



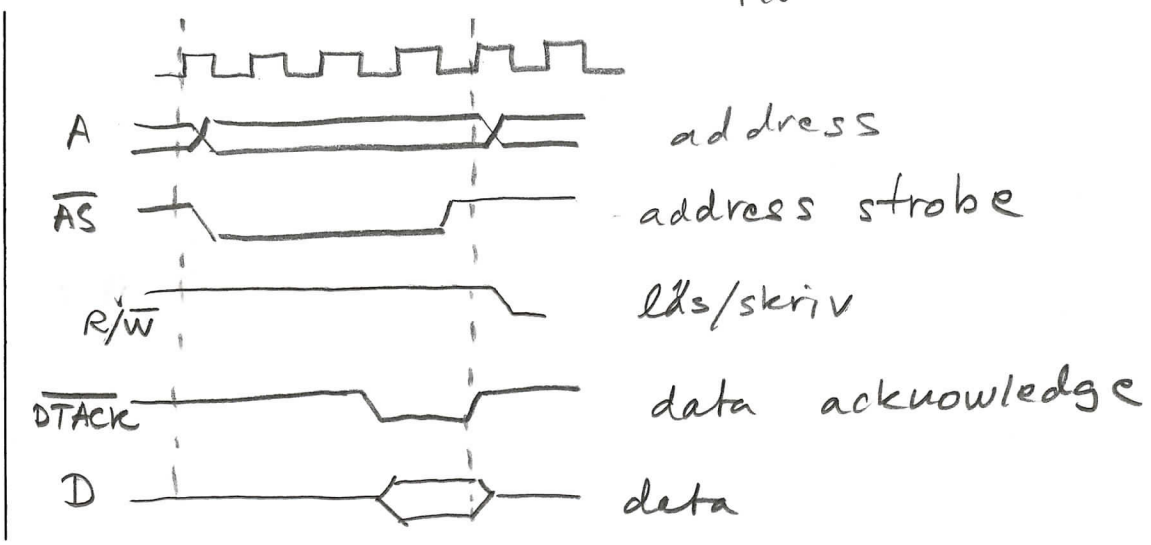
Detta arrangemang ger processorn möjlighet att arbeta i full takt när ingen busstrafik är för handen. Man kan också tänka sig att adressavkodningen, som skiljer ut den yttre enheten, påverkar busscykelns längd så att bara de yttre enheter som verkligen behöver extra wait-states får dem, medan andra, snabbare komponenter, körs med full hastighet. Se schemat på 68008-systemet i kapitlet om minnen.

12.2.2 Asynkron buss

Observera att man ovan bara kan lägga till *hela* wait-states, inte halva. Det är naturligtvis önskvärt att lägga till precis så långa wait-states som behövs, men inte längre. Lösningen på problemet kallas *asynkron buss*. Av namnet kan man förstå att denna buss inte är beroende av en taktgivare i form av en processorklocka som den synkrona bussen.

Konstruktörerna av processorn 68000 ville göra processorn generell. Så ock med busscyklerna. Förutom ett kompatibilitetsläge, där den härmar den synkrona bussen hos 6800[6], har 68000 en fullt asynkron buss. Genom insignalen $\overline{DTACK}$, *DaTa ACKnowledge*, till processorn, har en yttre enhet möjlighet att meddela exakt när den är klar och busscykeln kan slutföras och avslutas av processorn.

Funktionen framgår av tidsdiagrammet nedan.



Signalen $\overline{AS}$ (*Adress Strobe*) är en utsignal från processorn som meddelar att en giltig adress ligger på adressbussen. Den positiva, dvs uppåtgående, flanken på signalerna $\overline{LDS}$ och $\overline{UDS}$ (*Lower* och *Upper Data Strobe*) anger när processorn läser in värdet från databussen.[7] Signalen $R/\overline{W}$ (*Read/Write*) avgör om busscykeln är en läscykel eller skrivcykel. Signalerna FC<0:2> (*Function Code*) är skvallersignaler för i huvudsak yttre MMU, *Memory Management Unit*, för att kunna implementera virtuellt minne. Inget vi bryr oss om just nu.

Om 68000 nu verkligen ska ha en generell bus är det ingen förvåning att upptäcka att bussystemet även tillåter ett master/slave-förfarande. I normalfallet är 68000 bussmaster men det finns signaler från processorn varmed en yttre enhet kan meddela önskan att få ta över bussen. Dessa signaler framgår av databladet men är inget vi tar upp här.

Syftet med att kunna släppa bussen till en yttre enhet är uppenbar om man betänker att stora mängder data ska pumpas från en yttre enhet till processorsystemets minne för vidare behandling av processorn. Det är naturligtvis en onödig omväg att låta processorn först läsa in datat för att sedan omedelbart lägga ut det i minnet igen. Av denna anledning kan man låta processorn koppla loss sig från bussen och överlämna den till en yttre enhet, som då tillåts skriva direkt in i systemets minne utan inblandning av processorn. Resultatet, som kallas DMA, *Direct Memory Access*, är en mycket snabb och praktisk överföring av data där hela bussbandbredden kan användas. Principen framgår av nedanstående figur:

[6] 68000:s föregångare 6800 var en åttabitars processor med enbart synkron buss. Den hade en uppsjö av komponenter anpassade till sig och 68000 försågs med ett kompatibilitetsläge för att kunna dra nytta av dessa äldre synkrona komponenter. PIA:n 6821 är ett exempel på en komponent ur 6800-familjen som ofta används tillsammans med 68000 — och som dessutom har överlevt till våra dagar.

[7] Man kan föreställa sig att den positiva flanken låser datavärdena i interna vippor om det hjälper.

se OH för ett multimastersystem utan speciella signaler. En av dessa masters kan vara en DMA-enhet.

I och för sig kan man åstadkomma DMA utan att processorn har stöd för det, men med stöd blir det mycket lättare.

12.3 Andra bussar

Det finns många andra bussar i användning. Alla bygger dock på någon eller flera av de principer som de analyserade bussarna ovan. Vi tittar här på tre särskilt vanliga, nämligen ISA-bussen, PCI-bussen och den mer avancerade SCSI-bussen som främst är tänkt för servertillämpningar.

12.3.1 ISA-bussen

Det är egentligen inget nytt eller häftigt med ISA-bussen, men eftersom den har varit med sedan tidigt 1980-tal och fortfarande används i många PC-produkter vore det slarvigt att inte nämna den. ISA står för *Industry Standard Architecture* och är från en början 8-bitars buss som sedan utökades till både 16 och 32-bitar. Den senare kallas EISA, *Extended ISA*, men innebär inget revolutionerande nytt, mer än att just bitantalet ökats något. Dessutom verkar EISA-bussen blivit en parentes i datorhistorien: vill man ha prestandan hos en 32-bitars buss är det bättre att gå direkt på PCI-bussen (se nästa rubrik) än att envisas med den gamla ISA-arkitekturen. Nuförtiden måste man hävda att ISA definitivt passerat bäst-före datum, men den används fortfarande, bland annat i inbyggda system då instickskort (I/O-kort m m) med ISA-buss är billiga.

Busskontakten för ISA innehåller bland annat en adressbus om 20 bitar (A19–A0), en 8-bitars databuss (D7–D0) och de fyra kontrollsignalerna $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$ och $\overline{IOW}$. Dessa senare signaler utgör kontrollsignalerna *Read* och *Write* för minne respektive I/O. Man kan med rätta tycka att de borde räckt med enbart en Read- och en Write-signal. Anledningen till dubbleringen är att processortillverkaren Intel alltid försett sina processorer med separata adressområden för data/minne å ena sidan och I/O å andra. Det finns alltså 2^{20} stycken minnesadresser och lika många I/O-adresser på ISA-bussen. Till skillnad från, till exempel, fallet med 68000 där alla enheter trängs i samma adressrymd, s k minnesmappad I/O.

För att en yttre enhet ska kunna påkalla processorns uppmärksamhet finns även ett antal IRQ-singaler, IRQ7–IRQ2. För DMA finns signalerna DRQ3–DRQ1, *DMA Request*, för att begära bussen och signalerna $\overline{DACK3}$–$\overline{DACK0}$, *DMA Acknowledge*, för att överlämna den.

För att se hur enheter kan kopplas in till ISA-bussen visas nedan ett kopplingsschema där två A/D-omvandlare av typen ADC0804 anslutits. Komponenten 16L8 är ett PLD för adressavkodning. Ekvationen för utsignalen $\overline{O1}$ på pinne 19 är

$$\overline{O1} = \overline{A15} \cdot \ldots \cdot \overline{A10} \cdot \overline{A9} \cdot \overline{A8} \cdot \ldots \cdot \overline{A3},$$

och för utsignalen $\overline{O3}$ på pinne 17 är den

$$\overline{O3} = \overline{A15} \cdot \ldots \cdot \overline{A10} \cdot \overline{A9} \cdot \overline{A8} \cdot \ldots \cdot \overline{A3} \cdot \overline{IOR}.$$

se OH för en bild på busscykler för ISA.

Brey's avkodning

ISA-bussen har en maximal klockhastighet på 8MHz, även 32-bitarsvarianten, och för att vara kompatibla med tidigare PC-datorer är det en synkron buss.

12.3.2 PCI

Den nu förhärskande busstypen i bl a persondatorer är PCI, *Peripheral Interconnect Bus.* Namnet beskriver väl vad den är konstruerad för. Sedan något år tillbaka (nu = 2003) förekommer endast PCI-busskontakter på nya processorkort. Den är väsentligt snabbare än de tidigare bussarna med en klockhastighet om 33 eller 66 MHz och en bredd som vanligen är 32-bitar men finns även i en 64-bitars variant.

PCI-bussen är inte knuten till någon enskild processor eller tillverkare, till skillnad från ISA-bussen som är knuten mot Intel, utan ska kunna användas till både Intels processorer och IBMs PowerPC till exempel. För att komma dithän måste man införa en extra komponent, en *PCI-brygga* mellan processorn och PCI-bussenheterna. Detta är inte bara negativt, i och med det kan processorn låta PCI-enheterna gå i sin egen takt och processorn i sin, ända tills data måste överföras.

Prestanda på bussen är så höga att en normal PC även låter ISA-bussen ligga på PCI-bussen. För detta krävs mer hårdvara, en *PCI-ISA-brygga*, men bandbredden på PCI-bussen är tillräcklig för att även kunna hantera eventuella ISA-enheter. För att ytterligare höja prestanda kan data överföras i klump med en enklare handskakning, *burst mode*, än vad vi sett hittills: Man ser bara till att inleda en bussöverföring med en adress så kan man sedan direkt köra över 32- eller 64-bitars data synkront med PCI-klockan.[8]

se OH för en bild
på en bussCykel för PCI.

(bild enligt fig p575 Brey)

För att spara pinnar i busskontakten är adress- och dataledningarna multiplexade, dvs signalerna heter AD31–AD0, och en ytterligare signal används för att meddela om det för tillfället är en adress eller ett data som ligger på bussen.

12.3.3 SCSI

Innan vi slutar ska vi nudda vid en annan vanlig bussarkitektur, SCSI-bussen, där SCSI uttalas som "skassi" eller möjligen "skussi". Oavsett uttalet betyder det *Small Computer Systems Interface*. Den används ofta i servrar där prestanda är viktiga. Om man är kvalitetsmedveten är SCSI även ett alternativ för den vanliga bords-PC:n, men så kvalitetsmedveten kanske man inte är: Jämfört med den vanligaste busstandarden för PC — IDE — är SCSI nämligen betydligt dyrare både vad gäller enheter man kan ansluta och det instickskort, *SCSI-adapter*, man måste ha för att datorn ska kunna prata med SCSI-enheterna.

Från början, 1986, var SCSI en standard för överföring med för den tiden hyggliga prestanda på 5 Mb/s synkront. Med asynkron överföring var prestanda lägre, 1.5 Mb/s. 1990 kom en reviderad standard, SCSI-2, som tillät överföring med en takt av högst 10 Mb/s. Ytterligare revideringar av SCSI-2 har bland annat resulterat i SCSI-3 med en mod kallad Ultra160 som tillåter 160 Mb/s. Elektriskt har bussen utvecklats från obalanserad 5 volt signalspänning till numera balanserad med signalspänningen 3 volt.

Utöver de elektriska specifikationerna innehåller standarden även en hel svit av kommandon för att administrera bussen, RAID-enheter, bandstationer med mera och för att

[8] 64 bitar à 66 MHz = 8 byte à 66MHz = 528 Mb/s! Det går fort på PCI-bussen.