Föreläsningsunderlag
TSEA49 Datorteknik D Del 2

Michael Josefsson

10 december 2008

Innehåll

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*

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

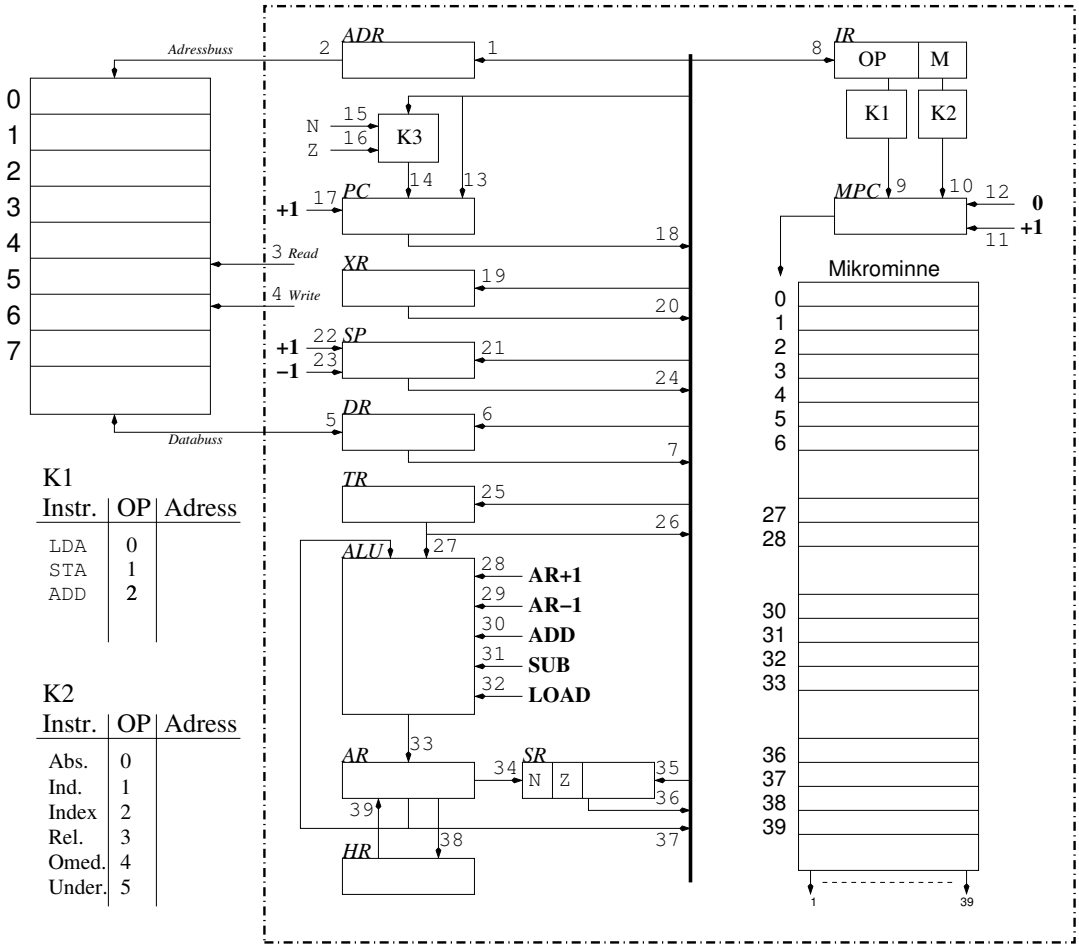
Med ett fem bitar brett fält kan vi ha högst 32 instruktioner. Vi tillåter fältet OP att innehålla någon av följande instruktioner:

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 $ea = XR + 3$.

som normalt är osynliga och oåtkomliga för assemblerprogrammeraren. Till exempel har ackumulatorn, AR, en kompanjon i *hjälpregistret*, HR. Instruktionregistret IR syns normalt inte heller utåt, men vi vet sedan tidigare att ett sådant behövs.

För att hantera det yttre minnet finns två register: *Adressregistret* ADR, som kan peka ut en rad i assemblerminnet, och det dubbelriktade *dataregistret* DR som läser/skriver in motsvarande data. Det finns också ett inre minne, *mikrominnet* som innehåller det mikroprogram som skapar de trettionio styrsignaler som denna mikromaskin kräver. För att peka ut nuvarande programrad i mikrominnet finns en *mikroprogramräknare*, MPC.

Vi ser också att det finns en hel del signalvägar med nummer på sig. Dessa hör ihop med motsvarande utsignal från mikrominnet.

Sammanfattningsvis: Det är mikroprogrammets uppgift att styra signalflödet i mikromaskinen så att den *utåt* ser ut som programmerarmodellen och uppför sig korrekt för varje assemblerinstruktion.

8.6.1 Normal arbetsgång — översikt

Med processorn definierad enligt ovan är det bara att slå på spänningen till den. Vad vill vi ska hända då? Den skall naturligtvis hämta första instruktionen, avkoda den och exekvera den, för att sedan hämta nästa instruktion och så vidare.

Precis som med andra tillståndsmaskiner måste även denna vakna upp i ett väldefinierat tillstånd. För den skull har processorn en $RESET$-knapp som nollställer alla register i den. Speciellt noterar vi att programräknaren, PC:n, nollställs. Den kan då peka ut rad noll i assemblerminnet.[7]

Den normala arbetsgången hos processorn blir nu:

Steg 1. Hämta instruktionen För att hämta in instruktionen från assemblerminnet till instruktionsregistret måste vi göra följande steg

- Släppa fram PC:ns innehåll till minnet (via adressbussen)
- Släppa fram minnets innehåll på den utpekade adressen till processorns dataregister, DR
- Överföra (kopiera räcker) dataregistret, DR, till instruktionsregistret, IR.

Steg 2. Avkoda instruktionen Med avkodning av instruktionen menas att med hänsyn till använd adresseringsmod — som ju anges av märkfältet — beräkna instruktionens effektiva adress. När detta steg är klart är alla eventuella beräkningar för att få fram adressen till operanden gjorda och slutresultatet är just denna adress. Nätet K_2 innehåller översättningen från märkfältets värde till startposition för den mikrokodsrutin som handhar adressavkodningen för respektive adressmod.

[7] Just det! I assemblerminnet. Det är viktigt att inte blanda ihop de olika nivåer av program vi nu hanterar. Dels har vi assemblernivån och under den, mer knutet till hårdvaran, mikroprogramnivån.

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.

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.

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:

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:

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?)

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

```
LDA 6                    ADD #3
```


■

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:

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`, (*Re-Turn 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.

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

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`

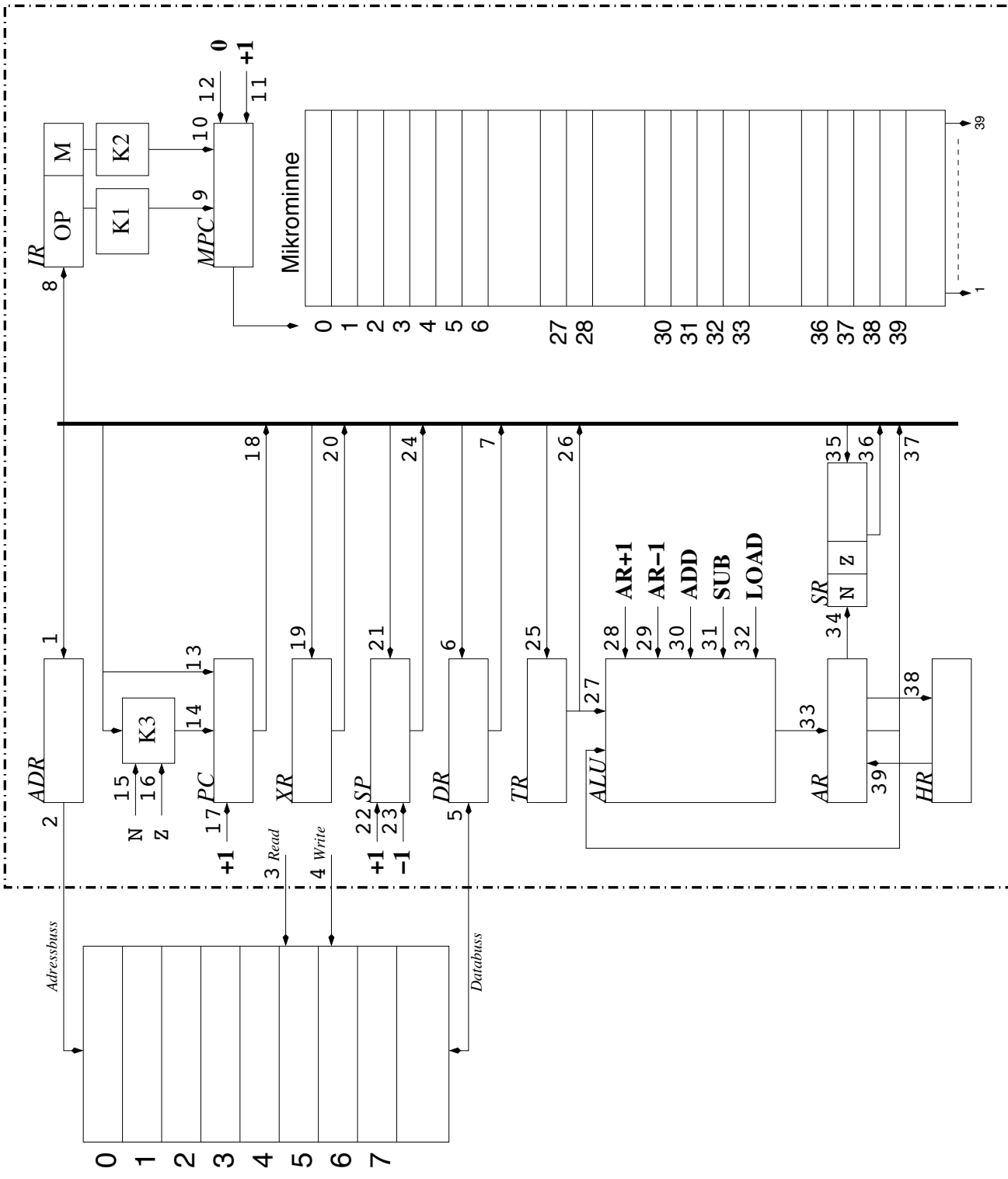
Mikrokoden för assemblerkommandot `RTI` fungerar på omvänt sätt:

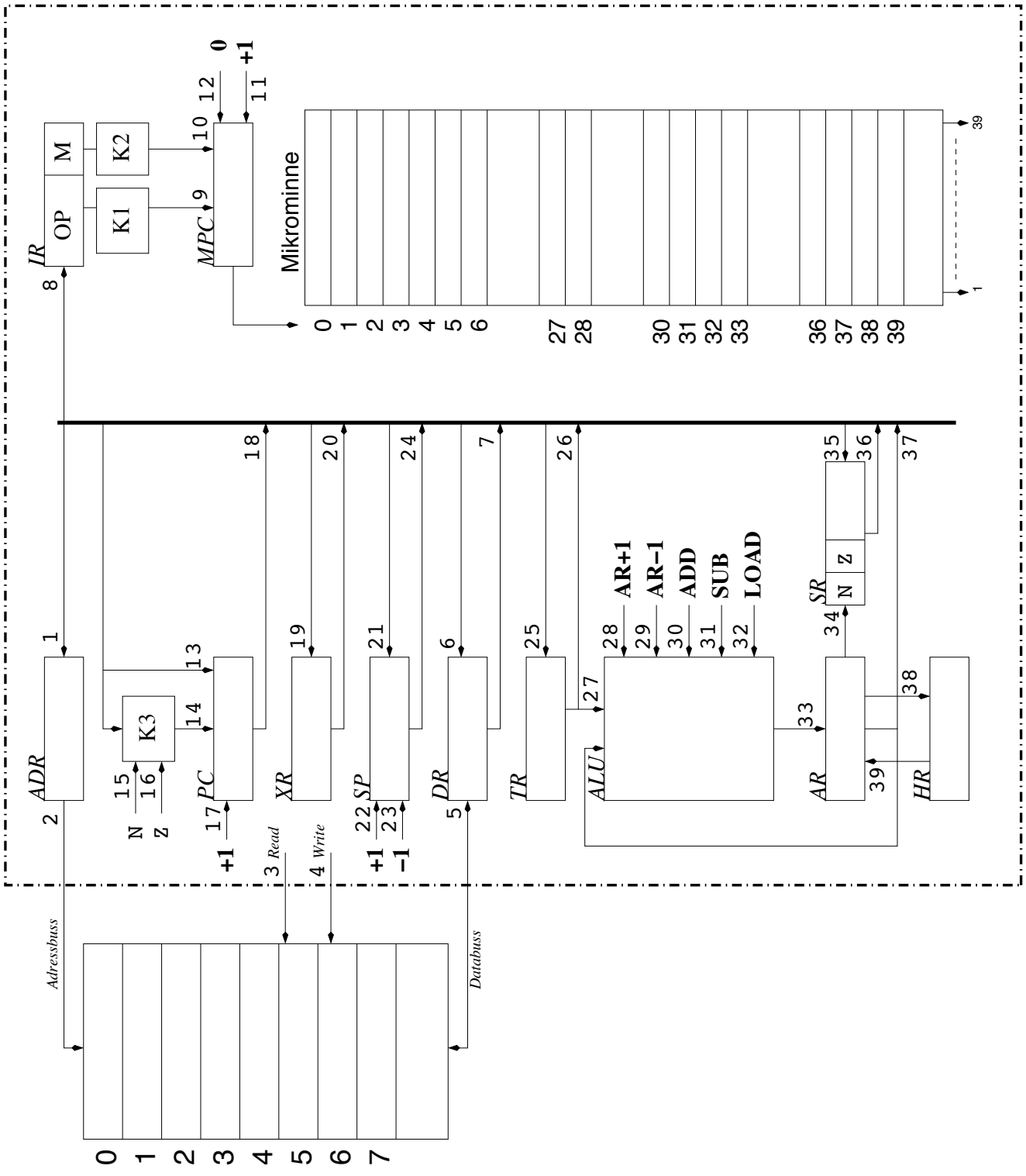
- Återställ processorns inre tillstånd, d v s hämta tillbaka det från stacken,
- Möjliggör avbrott på nytt,
- Börja en hämtfas, vi är då tillbaka i det normala programmet igen.

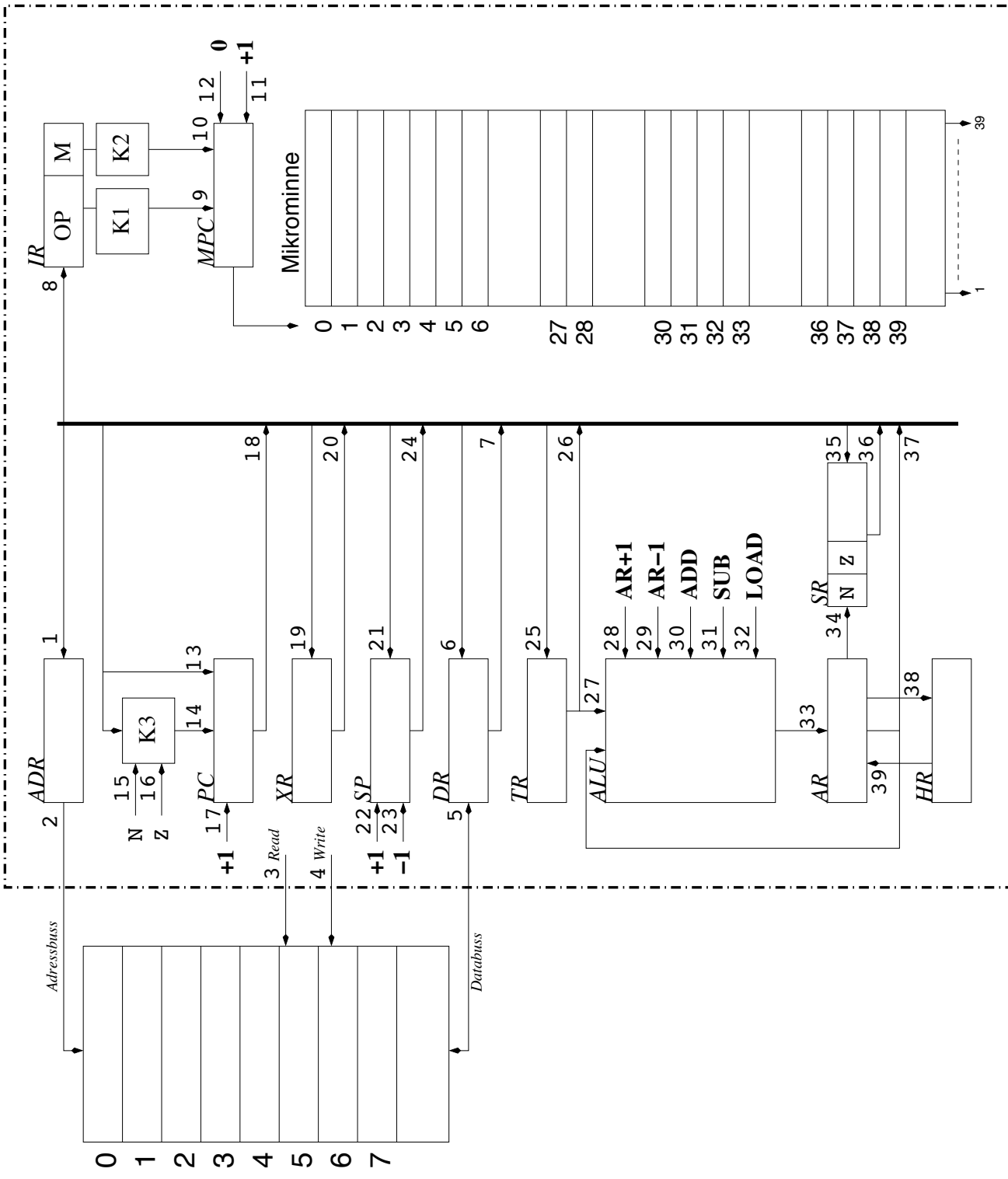
Mikrokoden för `RTI` lämnas som en övning för läsaren. Glöm inte att även fylla i näten K_i i tillämpliga delar.

8.13 Egna kompletteringar

Det återstår naturligtvis att skriva mikrokod för alla andra instruktioner och adresseringsmoder. Välj någon eller några och peta ihop mikrokoden själv. Försök optimera mikrokoden så långt som möjligt. Klura också ut vad signalerna 14, 15 och 16 och nätet K_3 har för uppgift.


Mikrominne
IR
OP
M
K1
K2
MPC
0
+1
Adressbuss
Databuss
ADR
PC
XR
SP
DR
TR
ALU
AR
SR
HR
K3
N
Z
+1
–1
Read
Write
AR+1
AR–1
ADD
SUB
LOAD


Mikrominne
IR
OP
M
K1
K2
MPC
0
+1
ADR
PC
XR
SP
DR
TR
ALU
AR
SR
HR
K3
N
Z
+1
–1
Read
Write
AR+1
AR–1
ADD
SUB
LOAD
Adressbuss
Databuss


Mikrominne
IR
OP
M
K1
K2
MPC
0
+1
ADR
PC
XR
SP
DR
TR
ALU
AR
SR
HR
K3
N
Z
+1
–1
Read
Write
Adressbuss
Databuss
AR+1
AR–1
ADD
SUB
LOAD

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?

- Att dessutom behöva gå genom temporärregistret för att kunna utföra många ALU-operationer är ju rätt onödigt.
- Det finns säkert fler brister...

När vi väl identifierat flaskhalsarna kan vi göra något åt dom. Det är svårt att göra förändringar enbart i mikrokoden. Eftersom mikrokoden intimt hänger samman med den befintliga hårdvaran får vi börja med att smida om hårdvaran. För modelldatorn gör vi först en speciell adressbuss internt skild från databussen, då kan adresser och data köras samtidigt, det möjliggör ökad parallellism även i mikrokoden. Om vi tar bort temporärregistret försvinner den flaskhalsen men samtidigt kanske vissa instruktioner omöjliggörs. Vi kan också börja överväga om alla instruktioner verkligen behövs. Och hur är det med adresseringsmoderna? Behövs verkligen alla varianter av dom?

Av detta förstår vi att det kan vara svårt att ändra i en befintlig konstruktion. Det är antagligen lättare att börja om från början.

9.2 Mål

På samma sätt som att vi måste identifiera flaskhalsarna innan de kan åtgärdas måste vi ha någon metod att avgöra när vi optimerat vår konstruktion tillräckligt. När är det dags att sluta finlira med arkitekturen och börja bygga?

Vi måste ställa oss frågan ”Hur snabb kan en processor bli”? I vårt fall borde vi vara nöjda om processorn kunde utföra en assemblerinstruktion per klockcykel. Även med ytterligare bussar och andra modifieringar vi kan tänka oss, kan vi inte tvinga vår processor att bli mycket snabbare. Inget händer utan klockpuls och den minsta klockpuls vi kan tänka oss är förstås en vanlig klockflank. Nu behövs en klockcykel för varje mikrokodsrad och vi vet att redan hämtfasen drar flera cykler. Kan vi komma dithän att en hel instruktion utförs per klockflank måste vi betrakta konstruktionen som optimal. Vi måste antagligen fortfarande traska igenom en hel del mikrokod för att utföra även de mest enkla operationerna, men genom en ökad parallellism i arkitekturen kan mikrokodsraderna dras ner till ett minimum.

Vi säger att målet för oss är detta: En instruktion per klockcykel.[1] Vill man göra mer kan vi tänka oss parallella behandlingsenheter (ALU o s v) i samma processor, men då måste det till en hel del logik för att se till att alla används effektivt och inte kolliderar med varann. Det är ett svårt kapitel[2] så vi nöjer oss med detta mål.

9.3 Vad tar plats i/krånglar till vår modell?

Bland annat detta:

- Instruktionsbredden är så liten att vi måste ibland använda två bytes för att ange vilken op-kod som avses och utöver detta dessutom en adress eller operand.

[1] Även benämningen CPI, *Clocks Per Instruction*, är vanlig.
[2] Snarare en bok i sig!

- Vi har en hel del adresseringsmoder. Kanske för många?
- Den där mikrokodsmaskinen äter ju också lite logik/räknare och slikt i sig. Framförallt är mikrokoden ett isolerande lager, som sinkar oss tidsmässigt, mellan assemblerinstruktionen och processorns grindar. Vi får inte glömma att det är grindarna som verkligen åstadkommer någonting i processorn.

Studier visar att alla våra fina instruktioner inte används så ofta de borde med tanke på den tid och omsorg vi lagt ner på att tillverka dom. Nuförtiden sitter man dessutom inte och handpetar assemblerinstruktioner i någon större utsträckning utan låter *kompilatorer*[3] göra jobbet. Kompilatorer kanske kan kan vara smarta men de är i allmänhet rätt kassa på att variera de genererade instruktionerna efter situationen. De har en handfull av de tillgängliga assemblerinstruktionerna som de använder. Man tjänar allstå inte så mycket på att ha en bred flora av instruktioner att välja från. De flesta används inte i alla fall.

Botemedlen är alltså, för att börja med det sista:

- Försök minimera antalet instruktioner. Studera vilka som verkligen kommer att användas. Det finns ingen anledning att lägga ner 30% jobb på en instruktion som bara används 5% av tiden. Då är det kanske bättre att syntetisera den med hjälp av andra instruktioner. En positiv sidoeffekt är naturligtvis att mikrokodsminnet inte behöver vara så stort, vilket i sin tur påverkar priset på vår processor.

Det finns flera studier över vilka instruktioner som används oftast. Beroende på vilken processor och kompilator som använts blir resultaten något annorlunda, men de avslöjar i varje fall vissa typinstruktioners frekvens:

Instruktion	**Frekvens (%)**
load	22
conditional jump	20
compare	16
store	12
add	8
and	6
sub	5
move reg–reg	4
call	1
return	1
Totalt	96

Andra undersökningar visar liknande resultat. Vi drar slutsatsen att huvuddelen av instruktionerna skall alltså vara enkla, d v s av typen `LDA`, `STA`, `JMP`, ... Vi ser också att många av dessa vanliga instruktioner måste prata med minnet hela tiden. Man kan undvika detta genom att låta data mellanlagras inuti processorn, dvs. vi kan undvika att belasta kanalen till minnet genom att lägga till interna register. För bästa prestanda lägger vi alltså till en registerbank med så många register vi har råd med.

Andra symptom och botemedel är:

[3] Alltså program som översätter våra program till assembler.

- 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-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.

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

- Problem 2. ALU-konflikt

- 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.

$$t_{acc} = h \cdot t_{CM} + (1 - h) \cdot t_{PM}$$

där h är den s k *träffkvoten*. Kvoten, h, är alltså ett mått på hur bra, i genomsnitt, cachen fungerar och kan definieras som

$$h = \frac{N_{CM}}{N_{CM} + N_{PM}}$$

Där N_{CM} och N_{PM} är antalet accesser till respektive minne överhuvudtaget. Om h är nära 1 blir accesstiden för det *totala* minnet ungefär lika med accesstiden för cacheminnet.

9.5.1 Locality of reference

Det visar sig att h faktiskt är stor. 0.95 och mer är inte en orimligt. Hur kan det komma sig?

Jo, det visar sig att program normalt uppvisar några egenskaper som går under samlingsbegreppet *locality of reference*,

- Program innehåller loopar d v s använder samma sekvens av *adresser* om och om igen. Denna egenskap kallas temporal lokalitet, *temporal locality*.
- Program består typiskt av procedurer/subrutiner och slikt, d.v.s. grupper av programrader som använder ungefär samma *data* om och om igen. Detta kallas spatial lokalitet, *spatial locality*.

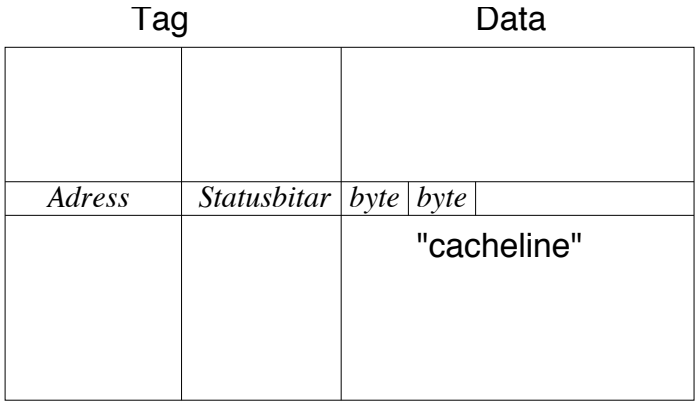
De båda lokaliteterna ovan medför att det faktiskt inte är så oerhört svårt att erhålla ett högt värde på h. Beroende på cacheminnets storlek kan större eller mindre del av ett program få plats i det, men poängen är att redan om bara *en del* av programmet ryms kommer h, i varje fall rätt ofta, att vara nästan 1.

> **Processorn upplever ett minne som är lika stort som primärminnet och nästan lika snabbt som cacheminnet.**

68000-efterföljaren 68020 hade bara 256 rader cacheminne, med redan det medförde en rejäl uppsnabbning. En modern processor à la Pentium4 har möjlighet till tre nivåer cache, L1, L2 och L3, där L1 sitter närmast processorn. L1 är 8kb stort och man har dessutom delat på cache för data och instruktioner (inte vanliga assemblerinstruktioner här, utan en lägre representationsform av redan avkodade instruktioner!). L2-cachen är 256kb stor, och kan nås på sju klockcykler. Sju cykler vid 1.5GHz är lite knappt 5 nanosekunder! Ruskigt snabbt alltså. P4:an har också stöd för en L3-cache som normalt inte implementeras.

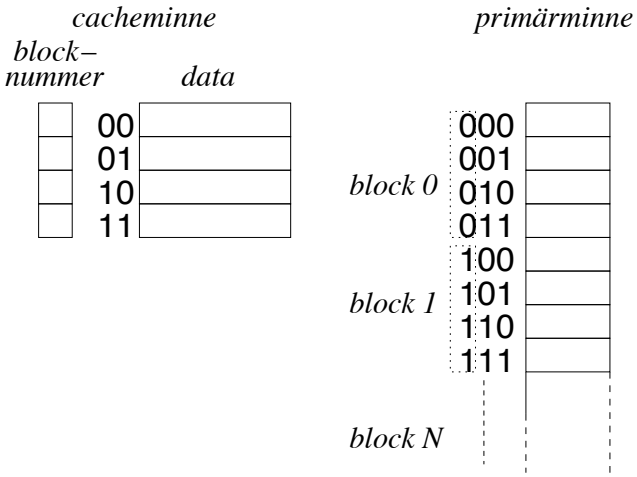
9.5.2 Cachestrategier

Vi ser att en förnuftigt använd cache avsevärt förbättrar processorns prestanda. Ska vi göra en snabb processor kan vi helt enkelt inte konstruera en arkitektur utan cache. Så hur gör man? Oavsett typ av cache delas minnet i block och av varje block gör man en *cacheline* i cachen:



Då kan man komma åt datat med några olika tekniker:

1. Direktmappad cache — den enklaste



Med adress-sekvensen 0,1,4,1, ... sker följande:

- Sök efter adress 0. Finns inte i cachen. Hämta från PM. Placera datat (`0A`) i CM rad 00. Markera vilket block denna rad togs ifrån (block 0).
- Sök efter adress 1. Finns inte i cachen. Hämta från PM. Placera datat (`0B`) i CM rad 01. Markera vilket block denna rad togs ifrån (block 0).
- Sök efter adress 4, dvs rad 00 i block 1. Finns inte i cachen. Hämta från PM. Placera datat (`0E`) i CM rad 00. Markera vilket block denna rad togs ifrån (block 1).

- 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*.

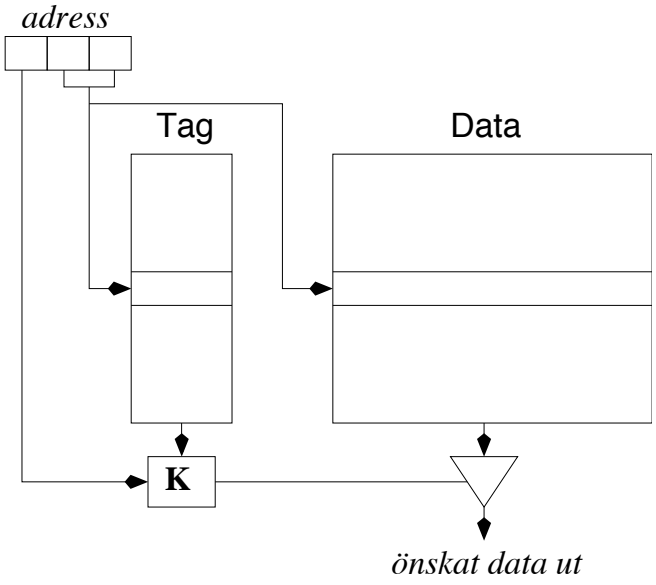
2. Associativt minne

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

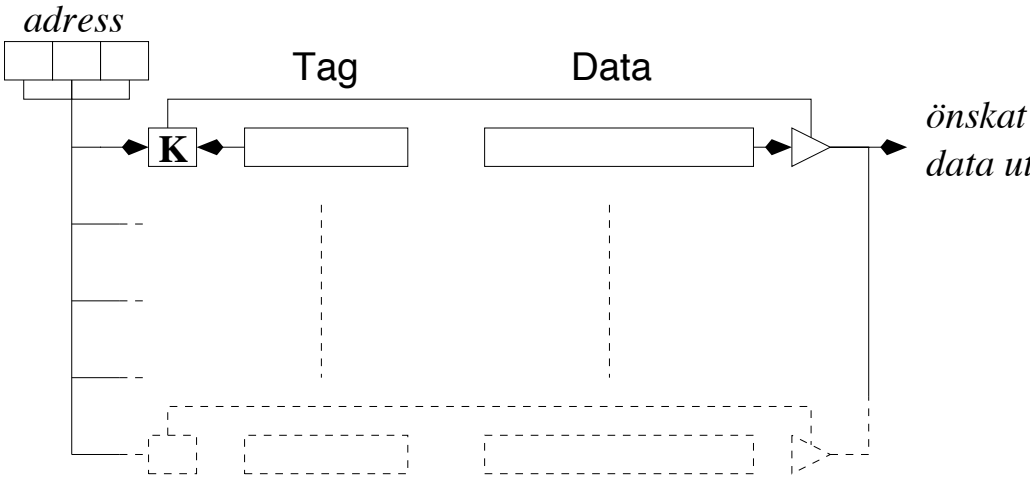
Nu jämförs *hela* adressen med den s.k. ”tag”-en. Det är naturligtvis så här vi vill ha det egentligen. Men det kostar kisel.

Hårdvaran för metod 1 och 2 skissas nedan:

- Hårdvara för direktadresserat minne



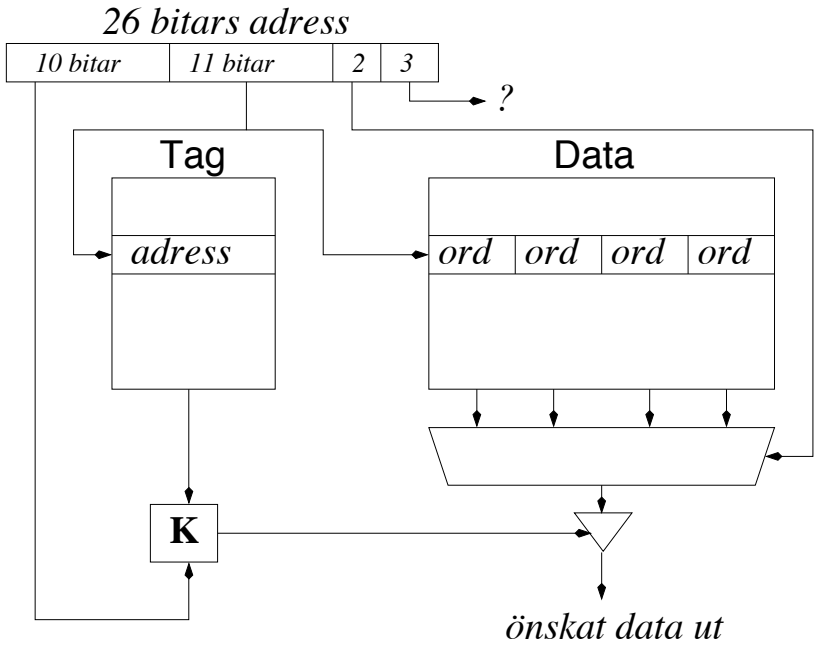
- Hårdvara för associativt minne



Sluligen kommer här ett sifferexempel på direktmappat cacheminne.

Exempel

Siffrorna anger att primärminnet är 64 Mb och cacheminnet 64 kb. Vidare antas en cacheline innehålla fyra ord à 32 bytes och ett ord är 64 bitar d v s 8 bytes:

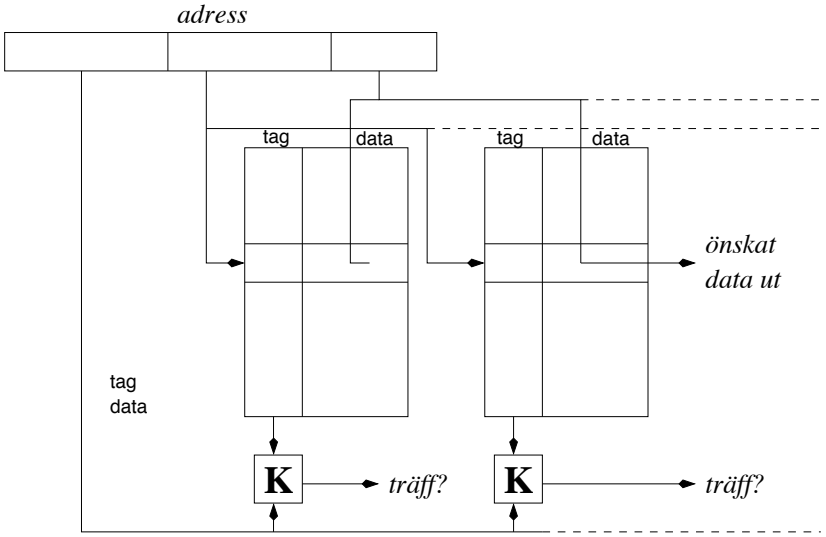


■

Klura själv ut vad de tre minst signifikanta bitarna i adressen betyder!

Associativt minne kostar kisel. Varför man inte sällan löser det hela med ett flervägs associativt minne (*set-associative memory*) som utgör en blandning av 1 och 2 ovan. Man kan till exempel ha två direktmappade minnen bredvid varann. Då blir risken för onödig överskrivning lite mindre.

3. Flervägs associativt minne



Observera allmänt att *hela* cachelinen måste uppdateras vid cachemiss. Oundvikligen kommer något att skrivas över, vi hoppas bara att det inte var något viktigt...

Om det bara är *instruktioner* i cachen vet vi att nya kan hämtas från PM om så behövs. Det är ett större problem om cachen innehåller *variabler/data* — som vi ändrat på. Då måste dom sparas och baxas ut till sina respektive platser i PM innan vi kan skriva över cachelinen. Man brukar välja mellan två strategier i det senare fallet:

- *write-thru*, uppdatera CM men skriv också omedelbart till PM, då är det inte farligt att skriva över cachelinen närhelst man vill det, och
- *write-back*, uppdatera CM, men skriv inte till PM förrän cachelinen ska skrivas över.

Det är klart önskvärt att veta om en speciell cacheline innehåller modifierad data, därför förser man varje cacheline för data med en flagga som anger om just denna line modifierats eller inte.

9.5.3 Utbytesalgoritmer

När vi ändå behandlar associativt cacheminne kan vi passa på att nämna några olika strategier för vilken cacheline man ska offra om man behöver återanvända platsen. Den oftast använda strategien kallas

- LRU, *Least Recently Used*. Man väljer den cacheline som använts mest sällan, och det verkar väl rimligt? Varje cacheline måste då förses med en räknare eller annan mekanism för att man ska kunna veta vilken som använts minst. Denna räknare ryms då inom cacheminnets *statusbitar*.
- En enklare strategi är att helt enkelt slumpa sig fram till en cacheline och det ger egendomligt nog inte så tokiga resultat det heller!
- Man kan också använda den cacheline som är äldst. Med temporal lokalitet i åtanke är det kanske inte ett dumt så förslag?

9.6 Stackprocessorer

Vi såg att RISC-processorns minnesbredd var stor bland annat p.g.a. att den måste lagra både två *source*-register och ett *destination*-register i samma ord, för att det hela skulle få plats, operationerna var ju av typen `ADD r2, r3, r4`. I en stackprocessor kommer man från detta och vinner dessutom att man lätt kan ha fler register utan att instruktionen behöver innehålla dessas adress!

Man kan generellt skilja på 0-, 1- respektive 2-operandsprocessorer. Vår modellprocessor är en 1-operandprocessor ty den innehåller alltid (tja, nästan) *en* operand eller adress, "`STA 12`", medan 68000-är en typisk 2-operandprocessor, "`MOVE.B D0,A1`". En stackprocessor är i allmänhet en *0-operandprocessor*. Endast i undantagsfall måste en operand specificeras. Oftast ligger operanden redan på stacken.

Vi har redan stött på stackbegreppet i samband med returstacken. I stackprocessorfallet används en stack även för parametrar och beräkningar. För beräkningar på en dylik parameterstack används RPN, *Reverse Polish Notation*, omvänd polsk notation, efter dess polske uppfinnare, matematikern Lukasiewicz.[7]

Exempel

Beräkna uttrycket $(219/34 + 3) \cdot 7$ med omvänd polsk notation:

```
219 34 / 3 + 7 *
```

På stacken ligger nu det önskade resultatet.

■

Detta skrivsätt brukar kallas RPN är faktiskt vanligare än man kan tro — i många mjukvarusammanhang är RPN en mellanprodukt, exempelvis vid utvärdering av algebraiska uttryck.

Frånvaron av källa och destination i instruktionen och närvaron av en stack för parametrar resulterar i flera tydliga fördelar hos en stackprocessor:

[7] Jämför med laboration 3, Portning av FORTH-kompilator.

- Programmen är typiskt små, till mycket små, i jämförelse med motsvarande kod skriven för traditionell processor.
- Mycket snabb exekvering. Det är relativt lätt att åstadkomma en instruktion per klockcykel. Enklare instruktioner kompenseras av att man kan klocka igenom fler per tidsenhet.
- Mycket snabbt subrutinhopp. Endast 1 – 2 klockor långt. Returadresser lagras på en annan stack, returstacken. Parametrar/operander överförs via parameterstacken.
- Förutsägbart beteende vid interrupt. Inga register behöver sparas undan vid interrupt. Allt finns på stacken och kan vara kvar där till avbrottsrutinen är klar.

Det sista är väsentligt vid realtidstillämpningar. Vi har ofta inte tid att låta avbrottet vänta kanske 50 – 60 klockcykler, som i 68000, på att börja exekveras — det måste ske bums! Med en sådan s k *interrupt latency* på bara någon klockcykel är stackprocessorernas avbrottsmekanism snabbare än någon annan processor.

Sammanfattningsvis kan man alltså säga att stackprocessorer är:

- Jämförelsevis enkla.
- Extremt effektiva.
- Har förutsägbar interrupt latency, d v s tiden mellan att interrupt signaleras tills att första raden i avbrottsrutinen exekveras är bestämd på förhand, och mycket liten.

9.7 Digitala signalprocessorer

En annan vanlig sorts specialprocessor är den digitala signlprocessorn, DSP:n. Typiska egenskaper hos en DSP:er är:

- *Multiply-accumulate*-instruktion, dvs en instruktion som multiplicerar och adderar i samma instruktion. Används typiskt i många DSP-algoritmer för filtrering av signaler och transformberäkningar inklusive den ”fnabba” Fouriertransformen, FFT.
- *Multiple-access memory architecture*, dvs möjlighet att läsa flera data ur dataminnet under en och samma instruktion.
- Speciella adresseringsmoder, används ofta då man hanterar arrayer av data och lite ovanligare datastrukturer än en generell processor har stöd för (*fifo*-buffrar m.m.).
- Speciell exekveringskontroll, för att kunna hantera loopar och annat iterativt på ett effektivt sätt. Många algoritmer i signalbehandlingsvärlden är iterativa.
- Speciella instruktioner för att stödja någon sorts bråktal. Det finns definitiva fördelar att använda bråktal vid signalbehandling jämfört med exempelvis heltal eller flyttal.

10 Minne

Begrepp i denna föreläsning: *ROM, PROM, EPROM, EEPROM, RWM, statiskt och dynamiskt RAM, minneselement, refresh, adressavkodning, minnesmapp, lineärt minne, segmenterat minne, virtuellt minne, minnesskydd, multipla program, tasks.*

10.1 Inledning

Vi har hittills betraktat minnet som en svart låda som givet en adress in levererar rätt data ut. Minnet har använts till både program och data utan vidare kommentarer. Det visar sig att det finns en hel mängd olika sorters minnen med några olika egenskaper.

I huvudsak kan man dock urskilja två sorters minnen: enbart läsminnen, *Read Only Memory*, ROM, och sådana som både kan läsas från och skrivas till, *Read Write Memory*, RWM. Någon har förhoppningsvis skrivit in något innehåll i ROM:en också men det är inget man kan upprepa flera gånger.

10.2 ROM

ROM är enbart läsminnen. Skrivning sker naturligtvis, men bara en gång. ROM behåller sitt innehåll vid spänningsfrånslag. Det finns ett antal undertyper bl a:

10.2.1 ROM, Read Only Memory

Dessa minnen tillverkas och programmeras en gång för alla direkt på fabrik. Det är till och med så att minnesinnehållet byggs in i konstruktionen av kretsen. Detta innebär naturligtvis att minnessorten enbart tillåter läsning hädanefter. Metoden att bygga in minnesinnehållet i kretsen lönar sig bara för mycket stora serier (tiotusentals kretsar och uppåt) och tar lång tid i anspråk (månader), från beställning till leverans. Sen gäller det naturligtvis också att hela minnesinnehållet är korrekt från början, det går inte att be om ändringar halvvägs in i produktionen.

10.2.2 PROM, Programmable Read Only Memory

PROM liknar ROM vid användningen, men skiljer sig vid programmeringen. Minnena är användarprogrammerbara en (1) gång och kan bara läsas därefter. Programmeringen sker genom att fysiskt bränna av små trådar i kretsen och man stelopererar på sätt minnesinnehållet för komponentens resterande livslängd.

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.

Som representant får ett typiskt statiskt minne återfinns CY6264 mot slutet av kapitlet.

10.3.2 DRAM, Dynamic Random Access Memory

I ett SRAM går det åt förhållandevis många transistorer för att tillverka den vippa som ska lagra informationen. Det är onödigt, eftersom en enda kondensator kan användas för att lagra en spänning eller inte. I det *dynamiska* RAM:et görs precis så.

Då en kondensator kan göras mycket liten i förhållande till en transistor och det visar sig att det bara behövs en kondensator och transistor per minnescell blir minnena mycket kompakta. Alla minnen med någon högre kapacitet idag utförs som DRAM. Stora DRAM är dessutom väsentligt mycket billigare att tillverka än stora SRAM.

För att kunna ha många adressignaler in till minnesytan utan att ha en stor kapsel använder man adressbenen på ett dubbelt sätt. Beroende på signalerna RAS, (*Row Address Strobe*) och CAS, (*Column Address Strobe*), anges om det är en rad- eller kolumnadress som ligger på adressbenen för tillfället. Den kompletta adressen delas alltså upp i två delar innan den används.

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:

- 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:

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,

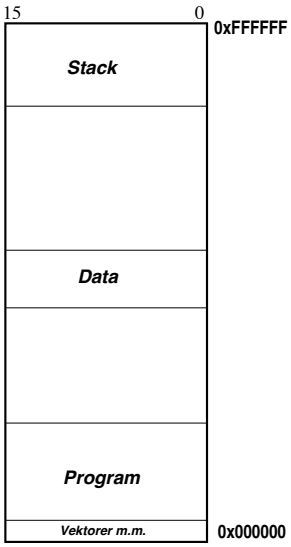
[4] Det program som i labbarna skriver `TUTOR 1.3 >`.

[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.

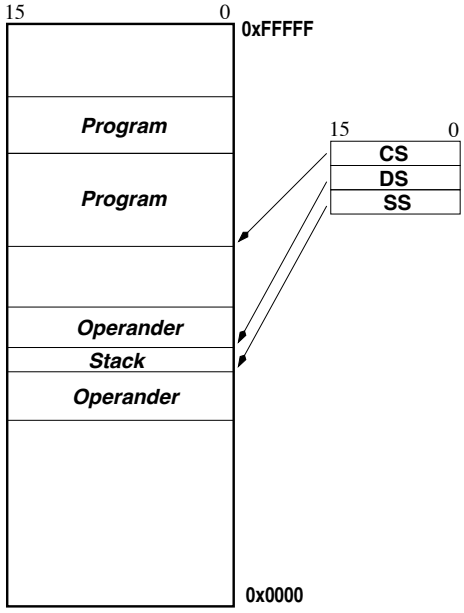
Resten av funktionen får ni fundera ut själva. Det kan då vara bra att känna till följande signaler: Signalen *AS*, *Address Strobe*, är aktiv då processorn lagt ut en giltig adress på sin adressbuss. $R/\overline{W}$, *Read/Write*, är hög vid läsning och låg vid skrivning. $\overline{DTACK}$, *Data 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.



10.4.2 Segmenterad minnesmodell

Även om den lineära minnesmodellen är intuitiv och enkel finns det fördelar med att inte låta minnet vara sammanhängande på det sättet. Motorola har av tradition en lineär minnesmodell medan Intel av tradition har bevarat en *segmenterad* minnesmodell. Den första processorn från Intel med segmenterad minnesmodell hette 8086.



I en segmenterad minnesmodell återfinns typiskt:

- ett eller flera programsegment
- ett stacksegment
- ett datasegment

För att kunna hålla reda på var i hela minnesrymden dessa segment befinner sig för tillfället finns extra interna register i processorn. I Intels fall kallas de CS, SS och DS beroende på om de pekar på ett CODE-, STACK- eller DATA-segment. Det vi kallar programräknare, PC, heter hos Intel *Instruction Pointer*, IP, och pekar ut var i CODE-segmentet nuvarande programrad är belägen. En komplett adress för kod är då kombinationen CS:IP.

DATA-segmentet kan bara innehålla data eftersom IP inte kan peka in i detta segment. STACK-segmentet innehåller den vanliga returstacken som naturligtvis också används för att mellanlagra parametrar.[6] Endast ett segment av varje sort kan vara aktivt vid varje enskilt tillfälle.

[6]Speciellt används den för parameteröverföring vid funktionsanrop i högnivåspråk.

På samma sätt måste man naturligtvis peka ut data och stack genom att, om nödvändigt, ange full adress med både ”lilla” pekaren och ”stora” segmentpekaren. Om man vill börja köra en annan del av sitt program, som ligger i ett annat segment, måste CS bytas ut och IP modifieras så att den pekar rätt. Eftersom IP är 16 bitar bred kan denna processor hantera segment som är 64 kB som störst. Om vårt program är större än 65536 programrader assembler måste vi använda ytterligare segment. Det är naturligtvis lite extra bök men inget orimligt företag.

Varför segmenterat minne?

Den största poängen, definitivt i Intels fall, är bakåtkompatibilitet! De program som var skrivna för Intels 8080, en 8-bitars processor med en absolut största minnesmängd på 64 kilobyte, kan[7], köras på en 8086:a. Bakåtkompatibiliteten blev än viktigare för Intel då de utvecklade efterföljaren 80286. Den tidigare så framgångsrika 8086-arkitekturen från Intel satt vid det här laget i varje PC som tillverkades. Det vore förstås mycket synd att gå miste om denna enorma marknad...

Men allt är inte bara tvång från ”marknaden”. En segmenterad minnesmodell kan ge vissa tekniska fördelar också. När Intels processorvariant 80286 kom ut på marknaden 1982 visade det sig vad man haft i åtanke när man höll fast vid sin idé med segmenterat minne och inte övergivit den för den kanske mer naturliga valet med en helt lineär minnesrymd. En segmenterad minnesmodell delar som nämnts upp minnet i bitar *beroende på användning* — och detta kan vändas till en fördel.

Genom att dela upp minnet beroende på användning kan hålla isär sådana minnesytor som inte har någon logisk kontakt med varann. Man kan också tillåta flera *program*-areaor i minnet utan att dessa ens vet om varann. Det går utmärkt att på så sätt hindra exekvering av innehållet i tabeller eller innehållet i de minnessegment som innehåller stackvärden. Denna distinktion av vad man får göra med respektive minnesyta hanteras av flaggor i s.k. *deskriptorer* som vi snart ska se. Med dessa flaggor kan vi markera segment som *icke-körbara*, dvs bara läsbara inte exekverbara, *enbart stacksegment*, dvs får bara manipuleras med `JSR`/`RTS` samt pushas till och poppas från. Användarprivilegier implementeras relativt lätt med hjälp av deskriptorer.

Även ett lineärt minne kan styckas upp i segment. Motorola åstadkommer detta med en yttre *minnehanteringskrets* som tittar på vilka adresser processorn genererar och jämför med de segmentgränser som finns i denna krets’ interna register. Innehållet i

[7] Visserligen efter omkompilering.

dessa register kan naturligtvis ändras under programmens gång. På senare processorer är denna MMU, *Memory Management Unit*, inbyggd.

MC68008 MINIMUM CONFIGURATION SYSTEM

INTRODUCTION

This application note demonstrates the design of a simple high-performance MC68008 system that uses the MC68681 Dual Universal Asynchronous Receiver Transmitter (DUART) to interface with external devices. The MC68008 is an excellent low-cost alternative to the MC68000 and features an 8-bit data bus while maintaining software compatibility with the rest of the M68000 Family. The MC68681 DUART is an M68000 Family data communications chip that features:

Two independent asynchronous serial channels,
A programmable 16-bit counter/timer,
A 6-bit parallel input port, and
An 8-bit parallel output port.

Emphasis in this design concept is placed upon performance, expandability, and low chip count.

The M68000 system design principles demonstrated in this application note include:

Interrupt hardware,
Peripheral interfacing,
Memory interface techniques,
Memory refresh arbitration in an M68000 system, and
Efficient serial I/O software.

The system, described in this design concept, features the following hardware:

An 8 MHz MC68000 microprocessor,
16K bytes of ROM,
64K bytes of dynamic RAM with no wait states, and
An MC68681 DUART.

The following paragraphs describe the hardware required for a high-performance, expandable, low chip count MC68008 system followed by a description of the software necessary to initialize and drive the MC68681 DUART.

HARDWARE REQUIREMENTS

The MC68008 has an asynchronous bus structure in which bus cycles are initiated by the assertion of address strobe ($\overline{AS}$) by the processor and are terminated by the assertion of data transfer acknowledge ($\overline{DTACK}$) by the peripheral or memory device being addressed. Figures 1-4 show the minimum hardware necessary for an MC68008 system consisting of:

Address decode logic,
$\overline{DTACK}$ generation logic,
Reset logic,
Bus error generation logic,
System memory,
Interrupt handling logic, and
An MC68681 interface.

The following paragraphs detail the required hardware as applied to the design concept described in this application note.

Address Decode Logic

The only tricky part of address decoding for an MC68008 system is that the system ROM must be mapped to address $00000 at reset. It would be impractical to fix the ROM at the bottom of the address map, as this would not allow for dynamic programming of interrupt vectors. To provide dynamic mapping of these interrupt vectors, an SN74LS164 shift register (U28) is used to generate a signal, $\overline{MAP}$, which is low for the first eight memory cycles after reset (the number of cycles necessry to fetch the reset vector and stack pointer). U28 is reset along with the processor and is clocked by the rising edge of $\overline{AS}$. The $\overline{MAP}$ signal generated by U28 is used by the address decoding circuitry to force selection of ROM when $\overline{MAP}$ is low and to allow normal memory decoding when $\overline{MAP}$ is high.

In the design given in this application note, address decoding is accomplished by a PAL16L8 (U22). This PAL is programmed to generate eight chip-select signals from ten input signals. The inputs to the PAL are the upper eight address lines (A12-A19), $\overline{IACK}$ (the NAND of the MC68008 function code lines, FC0-FC2), and the $\overline{MAP}$ signal. Four of the PAL-generated chip-select lines are used in this design to locate RAM at the address $00000, ROM at $A0000, and the MC68681 at $F0000. The four remaining chip-select lines are available for future system expansion.

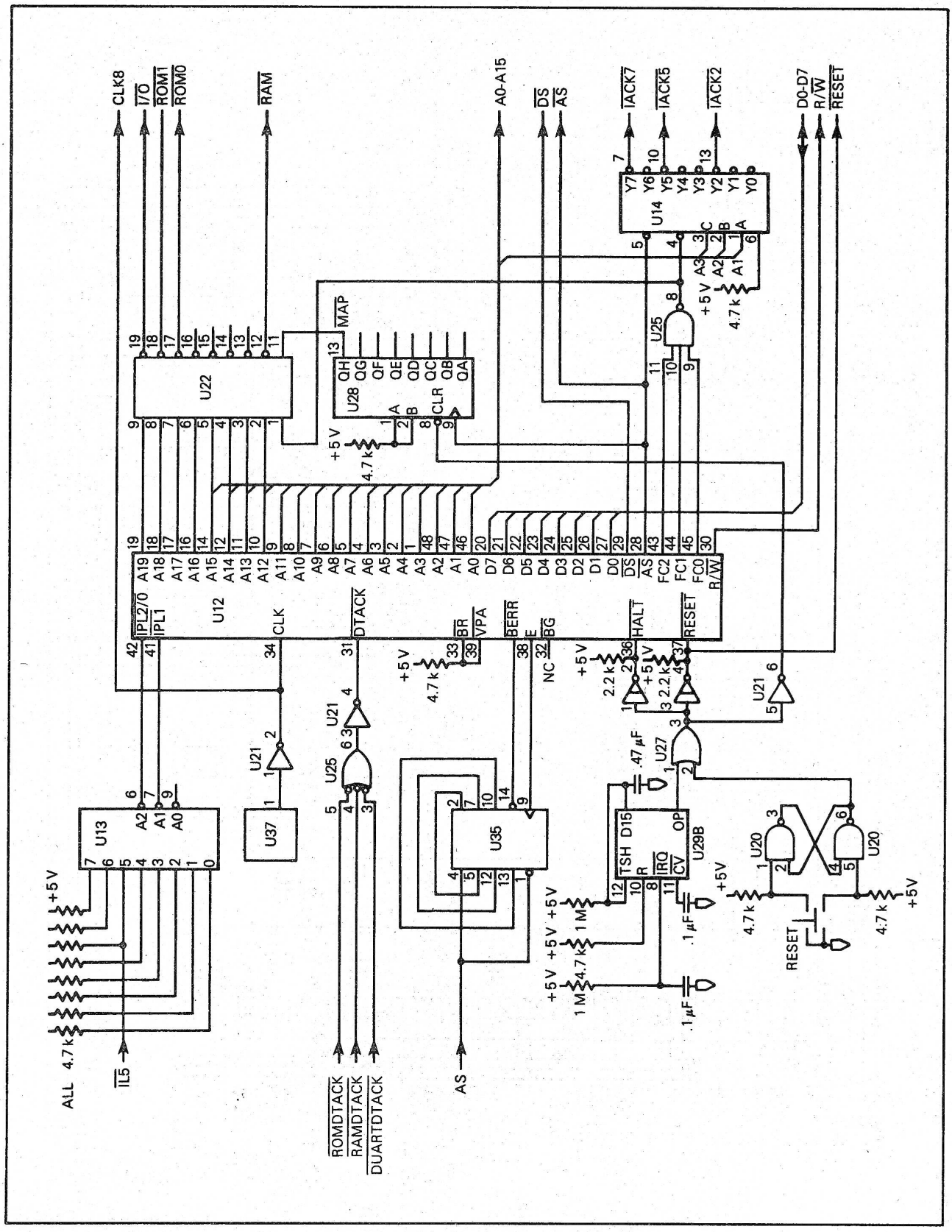


FIGURE 1 — MC680008 and Interrupt Hardware

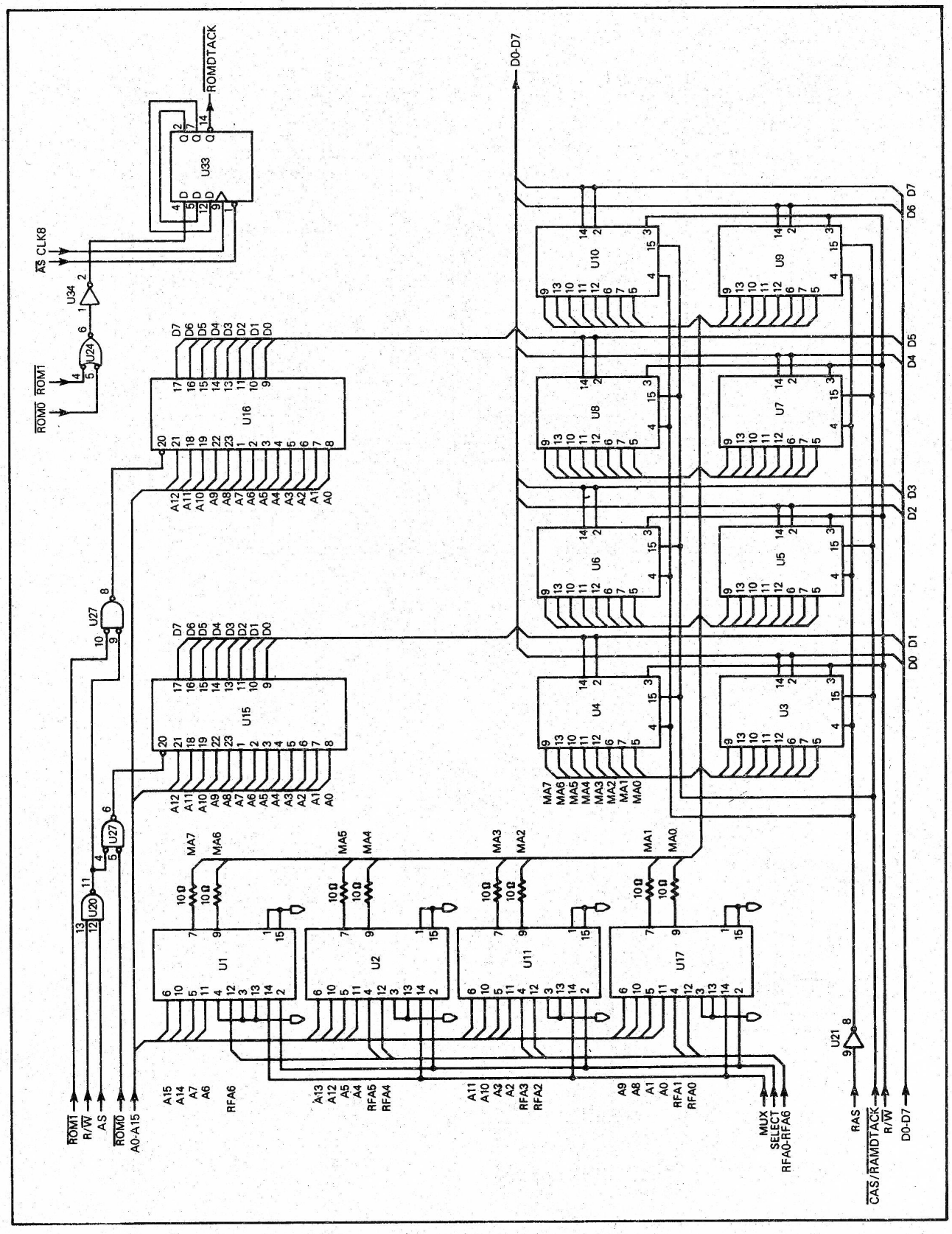


FIGURE 2 — RAM and ROM

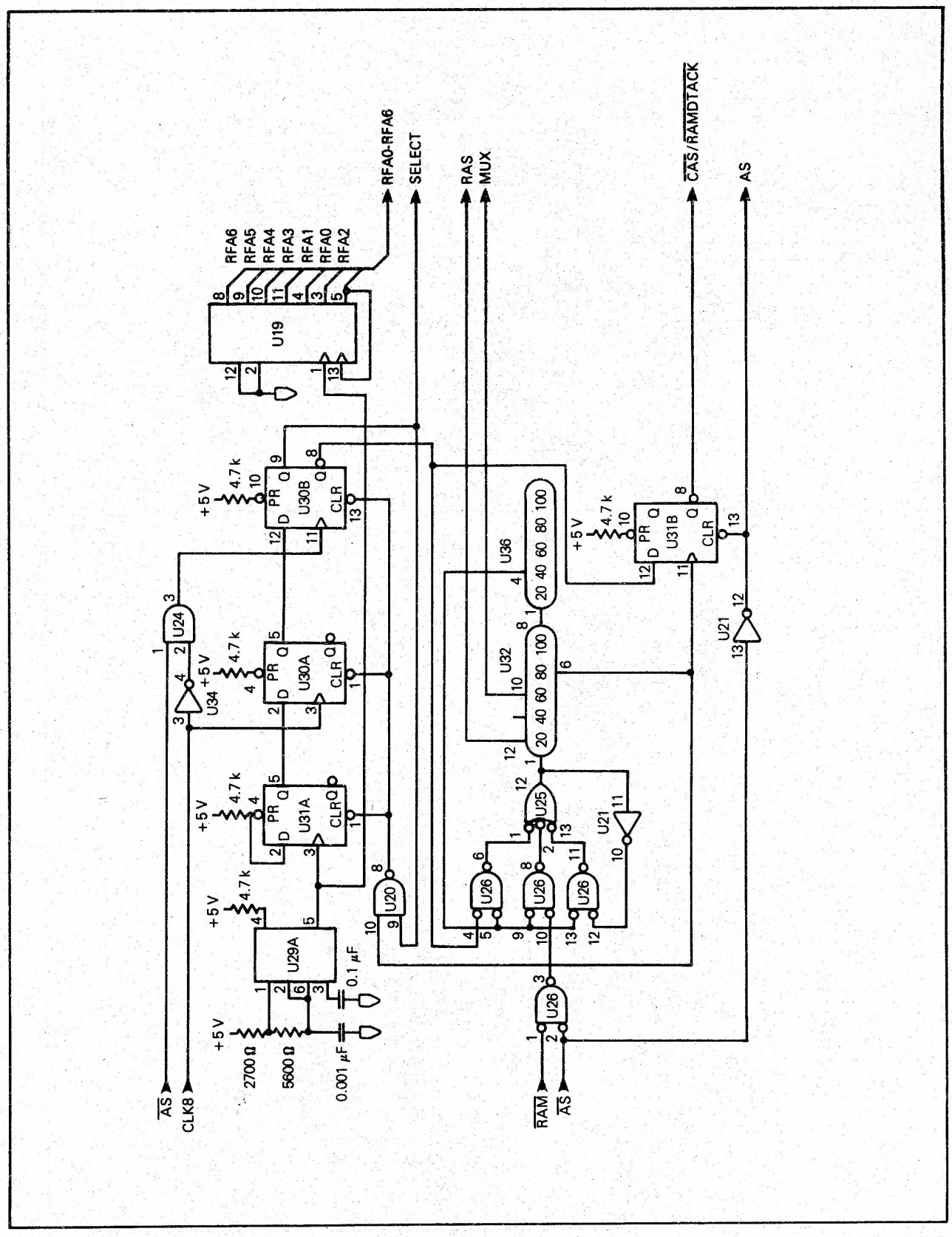
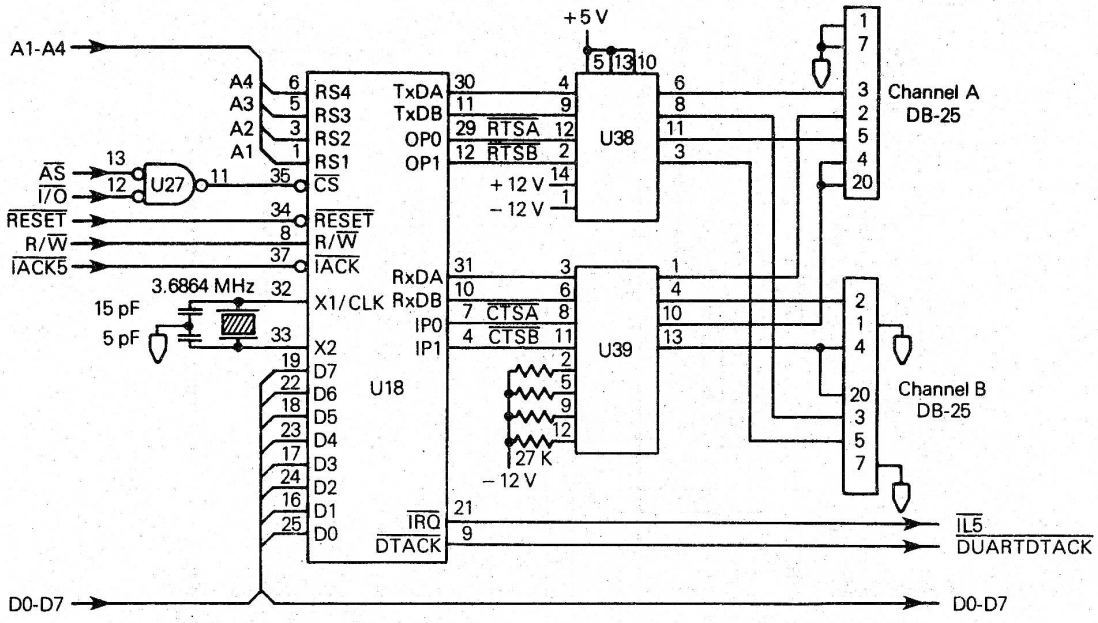
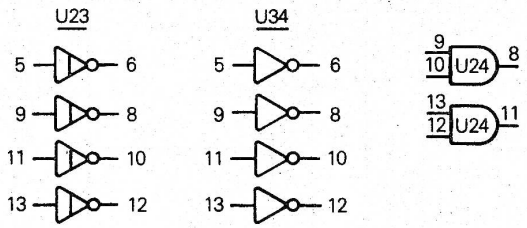


FIGURE 3 — Dynamic RAM Controller



EXTRA GATES



Parts

U1, U2, U11, U17	SN74S153
U3-U10	MCM6665-15
U12	MC68008L8
U13	SN74LS148
U14	SN74LS138
U15, U16	MCM68766
U18	MC68681
U19	SN74LS393
U20	SN74LS00
U21, U34	SN74LS04
U22	PAL16L8A
U23	SN74LS05
U24	SN74LS08
U25	SN74S10
U26, U27	SN74532
U28	SN74LS164
U29	MC3456
U30, U31	SN74F74
U32, U36	DDU-4-5100
U33, U35	SN74LS175
U37	K1114A, 8 MHz OSC
U38	MC1488
U39	MC1489

FIGURE 4 — MC68681 DUART

DTACK Generation Logic

There are three possible sources of $\overline{DTACK}$:

DUART $\overline{DTACK}$,

RAM $\overline{DTACK}$, and

ROM $\overline{DTACK}$.

The DUART generates its own $\overline{DTACK}$, $\overline{DTACK}$ for RAM is generated by the RAM control circuitry, and $\overline{DTACK}$ for ROM is generated by an SN74LS175 quad flip-flop (U33). These three $\overline{DTACK}$ sources are NANDed together by U25 and U21 to generate one processor $\overline{DTACK}$.

Reset Logic

There are two sources of system reset:

Power-up reset, and

Pushbutton reset.

Power-up reset is generated by the timer (U29B) which produces an active high pulse of approximately one-half second duration. The pushbutton reset, which allows the user to reset the system without powering down, is generated by a debounced switch. These two reset signals drive $\overline{RESET}$ and $\overline{HALT}$ through SN74LS05 open-collector drivers (U23).

Bus Error Generation Logic

The bus error signal, $\overline{BERR}$, is generated by an SN74LS175 quad flip-flop (U35). U35 counts clock pulses that occur after $\overline{AS}$ becomes asserted. If $\overline{AS}$ is still asserted after four rising edges of the E clock (between 5 and 6.5 microseconds), U35 will generate $\overline{BERR}$.

System Memory

The MC68008 system presented here has a system memory that consists of 16K bytes of ROM and 64K bytes of dynamic RAM (see Figure 2). Because system performance is critical in this design, a fairly complicated, but fast, dynamic RAM control circuit has been designed (see Figure 3). This circuit uses two delay lines to sequence RAS and to address the MUX, $\overline{CAS}$, and $\overline{DTACK}$ signals. Delay lines are necessary in order to optimize memory cycle times and make it possible to design the memory controller such that the system can operate without wait states.

A description of RAM refresh request synchronization and arbitration is given in the following paragraphs. Note that, for now, a signal called SELECT is assumed which initiates refresh cycles. The principle requirements of this signal are that it occurs periodically and that it becomes asserted only while $\overline{AS}$ is negated. In addition, the RAM decode signal is qualified with $\overline{AS}$ in order to create a RAM request signal. Either the SELECT signal or the RAM request signal may initiate a RAM cycle.

The front end of the RAM controller consists of three OR gates (U26) followed by a three input NAND gate (U25) which in turn feeds into the first delay line (U32). Each of the three OR gates has as one of its inputs a signal from the second delay line (U36) which changes state in the middle of the memory cycle. The other inputs to these OR gates consist of SELECT, RAM request, and an inverted feedback path from the output of the three input NAND. The initiation of a RAM cycle via either SELECT or RAM request causes the output of the NAND gate to go high. The output of the NAND gate is then held high by the inverted feedback path until the feedback from the second delay line forces it low. The purpose of the feedback path from the second delay line is to guarantee that the delay lines will be cleared and ready to begin another RAM cycle at the end of a cycle. The outputs of the first delay line generate the RAS, MUX, and $\overline{CAS}$ signals. Both the RAS and address multiplex signals are asserted during both types of RAM cycles (normal and refresh). $\overline{CAS}$ is generated only during normal cycles and must be held asserted until the processor removes $\overline{AS}$. In order to accomplish this, the appropriate delay tap (80 nanoseconds) is used to clock the SELECT signal through a flip-flop (U31B). This flip-flop is cleared when $\overline{AS}$ is negated. The output of this flip-flop is used for both the $\overline{CAS}$ signal and for the RAM $\overline{DTACK}$. The 8 MHz MC68008 allows the $\overline{DTACK}$ to be asserted up to 90 nanoseconds before data from memory is valid on a read cycle. The specifications for MCM6665L15 dynamic memories guarantee that data is valid 75 nanoseconds after $\overline{CAS}$.

The memory refresh controller operates on the principle of cycle stealing. Refresh requests may only occur between MPU bus cycles. If an MPU RAM cycle request occurs during a refresh cycle, it will not be started until the refresh cycle is finished. At periodic intervals, a free-running clock (U29A) clocks a flip-flop (U31A) to generate a refresh request. This refresh request is synchronized with the MPU clock by two flip-flops (U30).

The MC68008 ac electrical specifications guarantee one falling clock edge during the $\overline{AS}$ high time and that there will be at least a one-half clock period of $\overline{AS}$ high time following that clock edge. Arbitration between MPU and refresh requests occurs during this one-half clock period. The refresh request synchronizer consists of two SN74F74 flip-flops (U30). The first flip-flop (U30A) has as its input the refresh request signal from the refresh request flip-flop (U31A) and is clocked by the MPU clock. The second flip-flop (U30B) has as its input the output of the first synchronizer flip-flop (U30A) and is clocked by the MPU clock qualified by $\overline{AS}$ high. This two-level synchronizer is used to ensure that there will be no risk of the first synchronizer flip-flop (U30A) entering a metastable state due to a missed setup time. The output of the synchronizer flip-flop (U30B) is the SELECT signal. All three flip-flops of the refresh circuitry are cleared after the $\overline{CAS}/\overline{RAMDTACK}$ flip-flop (U31B) has been clocked during the refresh cycle. Address multiplexing for the RAM is done by four SN74S153 multiplexers (U1, U2, U11, and U17) with the appropriate addresses routed to the RAMs by SELECT and MUX. Refresh addresses are generated by an SN74LS393 dual 4-bit counter (U19) which is clocked by the refresh clock.

Interrupt Handling Logic

The interrupt handling logic must prioritize incoming interrupt requests and generate interrupt acknowledge signals back to the interrupt sources. Interrupt prioritization is accomplished with an SN74LS148 8-to-3 priority encoder (U13). The MC68008 supports three of the M68000 interrupt levels (interrupt levels two, five, and seven); therefore, only two of the outputs of U13 are connected to the MC68008. An SN74LS138 3-to-8 demultiplexer (U14) is used to generate $\overline{IACK}$ signals for interrupting devices. The SN74LS138 is enabled when $\overline{AS}$ is asserted and FC0-FC2 are all high (indicating an interrupt acknowledge cycle). Because the MC68681 uses only one of the interrupt levels (interrupt level five), the remaining two levels are available for future system expansion.

The MC68681 Interface

With these logic circuits in place, interfacing the MC68681 to the MC68008 is trivial (see Figure 4). The $\overline{RESET}$, R/$\overline{W}$, and data bus lines (D0-D7) are connected directly between the MC68681 and the MC68008. The I/O chip-select line generated by the address decode logic ($\overline{I/O}$) is connected to the MC68681 chip-select ($\overline{CS}$) pin. These address lines are

used instead of A0-A3 in order to maintain hardware design consistency with the other M68000 Family microprocessors (which do not have address line A0). The MC68681 interrupt ($\overline{IRQ}$) and interrupt acknowledge ($\overline{IACK}$) pins are tied to the $\overline{IL5}$ and $\overline{IACK5}$ lines of the interrupt handling logic, respectively, thus assigning the MC68681 interrupt a level 5 priority. Finally, a 3.6864 MHz crystal is connected between the MC68681 X1/CLK and X2 pins. The crystal is required for the on-chip baud-rate generator. 15 pF and 5 pF shunt capacitors must also be connected between the crystal and ground as shown to ensure proper operation of the oscillator.

The MC68681 serial channels are connected to external devices via RS-232 drivers and DB-25 connectors. The MC68681 $\overline{OP0}$, $\overline{IP0}$, OP1, and $\overline{IP1}$ pins are used as the $\overline{RTSA}$, $\overline{CTSA}$, $\overline{RTSB}$, and $\overline{CTSB}$ handshake lines, respectively; therefore, they too are routed via the RS-232 drivers to their respective connectors.

THE DUART SOFTWARE

This design will use both of the channels and the $\overline{RTS}/\overline{CTS}$ handshake capabilities of the DUART. The interface software required for this design is flowcharted in Figure 5 and is listed at the end of this document. There are three routines: DINIT, INCH, and OUTCH.

DINIT is the DUART initialization routine and is executed upon system power-up. After DINIT initializes the DUART channels, it enables channel A and channel B in normal operation mode. INCH is the input character routine. Upon entry, INCH requires the channel base address in address register A0. Upon return, the lower byte of data register D0 will contain the received character. OUTCH is the output character routine. Upon entry, OUTCH requires the channel base address in address register A0 and the character to be transmitted in the lower byte of data register D0.

SUMMARY

The system presented in this design concept is a low-cost, high-performance minimal chip count system. With the MC68681 DUART alone, this system offers a high degree of interface flexibility. It provides two serial I/O channels, a parallel input port, a parallel output port, a counter/timer, and versatile interrupt capabilities. However, if more peripheral chips are required, they can be interfaced easily. Also, if chip count is of higher importance than performance and/or expandability, the design presented here can be reduced even further by simplifying the RAM controller logic and the interrupt handler logic.

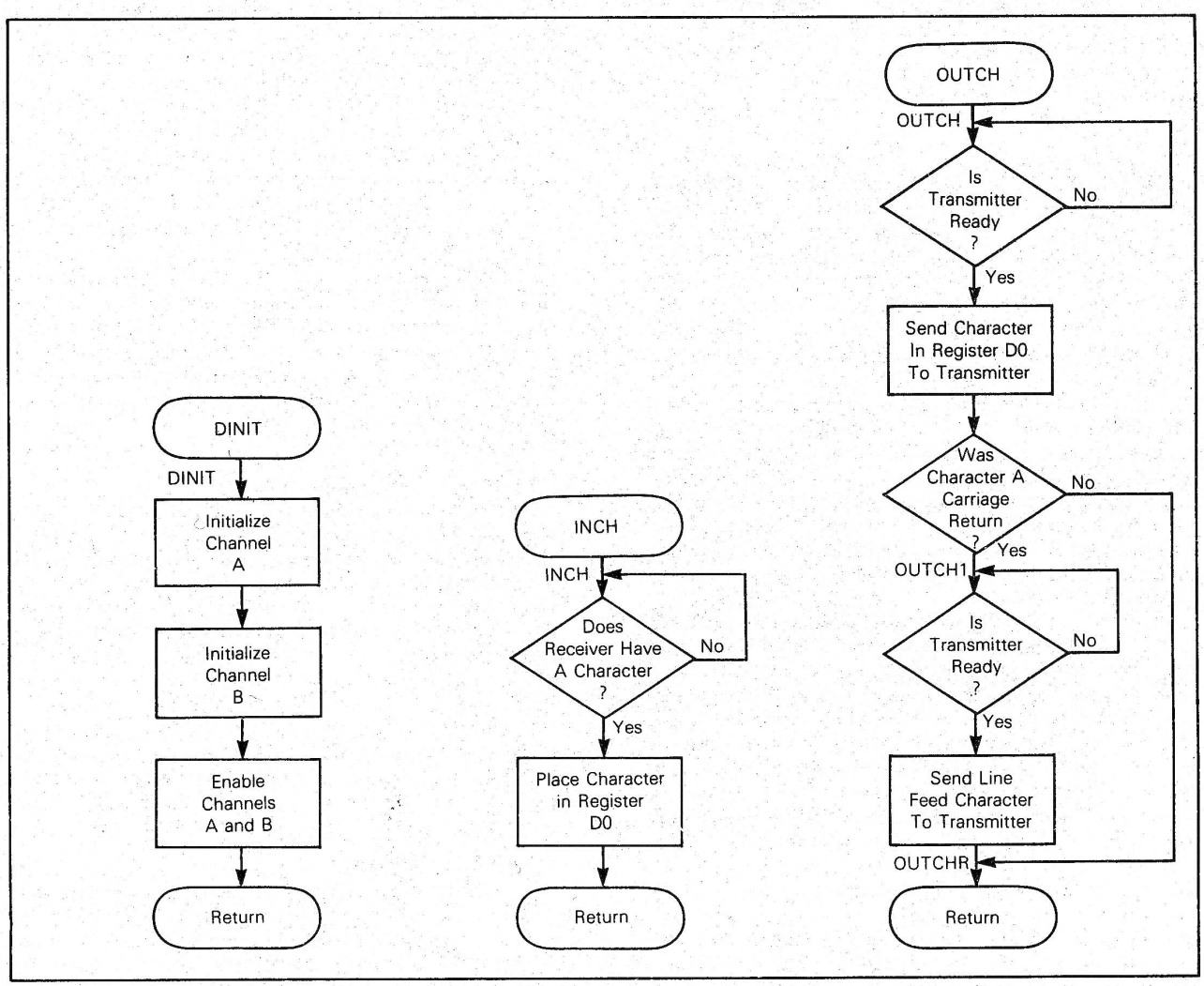


FIGURE 5 — MC68008 Minimum System Software Flowcharts

```
MOTOROLA M68000 ASM VERSION 1.30SYS :   5.        .MINSYSS .SA 01/20/84 13:44:18                PAGE 1 of 4
MINSYSS

                                OPT      FRS,PCS,BRS
                        *
                        * MINSYS   - ROUTINES REQUIRED FOR A 68008-BASED SYSTEM TO INITIALIZE
                        *           AND DRIVE A 68681 DUART:
                        *
                        *                 DINIT - SUBROUTINE TO INITIALIZE DUART
                        *                  INCH - SUBROUTINE TO INPUT A CHARACTER
                        *                 OUTCH - SUBROUTINE TO OUTPUT A CHARACTER
                        *
                        *           AUTHOR    - KYLE HARPER
                        *           DATE      - DECEMBER 22, 1983
                        *           VERSION   - 2
                        *
                        *


                        *
                        * SYSTEM ADDRESSES
                        *

000F0001    DUART       EQU      $0F0001          BASE ADDRESS OF 68681 DUART

000F0001    CHANA       EQU      DUART+0          CHANNEL A BASE ADDRESS
000F0001    MR1A        EQU      DUART+0          MODE REGISTER 1A
000F0001    MR2A        EQU      DUART+0          MODE REGISTER 2A
000F0003    SRA         EQU      DUART+2          STATUS REGISTER A
000F0003    CSRA        EQU      DUART+2          CLOCK-SELECT REGISTER A
000F0005    CRA         EQU      DUART+4          COMMAND REGISTER A
000F0007    RBA         EQU      DUART+6          RECEIVER BUFFER A
000F0007    TBA         EQU      DUART+6          TRANSMITTER BUFFER A

000F0009    IPCR        EQU      DUART+8          INPUT PORT CHANGE REGISTER
000F0009    ACR         EQU      DUART+8          AUXILIARY CONTROL REGISTER
000F000B    ISR         EQU      DUART+10         INTERRUPT STATUS REGISTER
000F000B    IMR         EQU      DUART+10         INTERRUPT MASK REGISTER
000F000D    CMSB        EQU      DUART+12         CURRENT COUNTER/TIMER MOST SIGNIFICANT BYTE
000F000D    CTUR        EQU      DUART+12         COUNTER/TIMER UPPER REGISTER
000F000F    CLSB        EQU      DUART+14         CURRENT COUNTER/TIMER LEAST SIGNIFICANT BYTE
000F000F    CTLR        EQU      DUART+14         COUNTER/TIMER LOWER REGISTER

000F0011    CHANB       EQU      DUART+16         CHANNEL B BASE ADDRESS
000F0011    MR1B        EQU      DUART+16         MODE REGISTER 1B
000F0011    MR2B        EQU      DUART+16         MODE REGISTER 2B
000F0013    SRB         EQU      DUART+18         STATUS REGISTER B
000F0013    CSRB        EQU      DUART+18         CLOCK-SELECT REGISTER B
000F0015    CRB         EQU      DUART+20         COMMAND REGISTER B
000F0017    RBB         EQU      DUART+22         RECEIVER BUFFER B
000F0017    TBB         EQU      DUART+22         TRANSMITTER BUFFER B

000F0019    IVR         EQU      DUART+24         INTERRUPT VECTOR REGISTER
000F001B    IP          EQU      DUART+26         INPUT PORT (UNLATCHED)
000F001B    OPCR        EQU      DUART+26         OUTPUT PORT CONFIGURATION REGISTER
000F001D    STRC        EQU      DUART+28         START-COUNTER COMMAND
000F001D    BTST        EQU      DUART+28         OUTPUT PORT REGISTER BIT SET COMMAND
000F001F    STPC        EQU      DUART+30         STOP-COUNTER COMMAND
000F001F    BTRST       EQU      DUART+30         OUTPUT PORT REGISTER BIT RESET COMMAND

                        *
                        * CONSTANTS
```

```
                                *
                                *
00000000D           CR      EQU     $0D                 ASCII CARRIAGE RETURN
0000000A            LF      EQU     $0A                 ASCII LINE FEED


00002000                    ORG     $002000

                                *
                                * DINIT - DUART INITIALIZATION ROUTINE.
                                *         AFTER INITIALIZING THE DUART CHANNELS FOR
                                *         OPERATION, DINIT ENABLES CHANNEL A AND CHANNEL B.
                                *
                                *         ENTRY CONDITIONS:
                                *
                                *                 (NONE)
                                *
                                *         EXIT CONDITIONS:
                                *
                                *                 CHANNEL A'S RX & TX ARE ENABLED.
                                *                 CHANNEL B'S RX & TX ARE ENABLED.
                                *                 ALL REGISTERS ARE UNALTERED.
                                *
                                *

00002000 13FC0000000F DINIT     MOVE.B  #$00,ACR            USE BAUD RATE GENERATOR SET 1
         0009
00002008 13FC00BB000F           MOVE.B  #$BB,CSRA           A: RX & TX AT 9600 BAUD
         0003
00002010 13FC0082000F           MOVE.B  #$82,MR1A           RX-RTS,NO RX-IRQ,CHAR ER,EVN PRTY,7 CHAR
         0001
00002018 13FC001F000F           MOVE.B  #$1F,MR2A           NORMAL,NO TX-RTS,CTS-TX,2 STOPS
         0001
00002020 13FC00BB000F           MOVE.B  #$BB,CSRB           B: RX & TX AT 9600 BAUD
         0013
00002028 13FC0082000F           MOVE.B  #$82,MR1B           RX-RTS,NO RX-IRQ,CHAR ER,EVN PRTY,7 CHAR
         0011
00002030 13FC001F000F           MOVE.B  #$1F,MR2B           NORMAL,NO TX-RTS,CTS-TX,2 STOPS
         0011
00002038 13FC0005000F           MOVE.B  #$05,CRA            ENABLE A'S RX & TX
         0005
00002040 13FC0005000F           MOVE.B  #$05,CRB            ENABLE B'S RX & TX
         0015
00002048 4E75                   RTS

                                *
                                * INCH - DUART CHANNEL INPUT CHARACTER ROUTINE.
                                *         GETS CHARACTER FROM A DUART CHANNEL AND PLACES IT IN D0.
                                *
                                *         ENTRY CONDITIONS:
                                *
                                *                 CHANNEL BASE ADDRESS IN A0.
                                *                 CHANNEL RX ENABLED.
                                *
                                *         EXIT CONDITIONS:
                                *
                                *                 RECEIVED CHARACTER PLACED IN D0.
                                *                 ALL OTHER REGISTERS UNALTERED.
```

```
MOTOROLA M68000 ASM VERSION  1.30SYS :   5.        .MINSYSS .SA 01/20/84 13:44:18                PAGE 3 of 4
MINSYSS

 109                            *
 110                            *
 111
 112   0000204A 082800000002 INCH     BTST.B   #0,2(A0)              WAIT FOR CHANNEL'S RX TO GET A CHAR
 113   00002050 67F8                  BEQ      INCH
 114   00002052 10280006              MOVE.B   6(A0),D0              GET CHARACTER FROM RECEIVER
 115   00002056 4E75                  RTS
 116
 117                            *
 118                            * OUTCH - DUART CHANNEL OUTPUT CHARACTER ROUTINE.
 119                            *         SENDS CHARACTER IN D0 TO A DUART CHANNEL.
 120                            *         IF CHARACTER IN D0 IS A CARRIAGE RETURN, OUTCH WILL
 121                            *         OUTPUT BOTH A CARRIAGE RETURN & LINE FEED CHARACTER.
 122                            *
 123                            *         ENTRY CONDITIONS:
 124                            *
 125                            *                 CHANNEL BASE ADDRESS IN A0.
 126                            *                 CHARACTER TO BE TRANSMITTED IN D0.
 127                            *                 CHANNEL TRANSMITTER ENABLED.
 128                            *
 129                            *         EXIT CONDITIONS:
 130                            *
 131                            *                 CHARACTER SENT TO CHANNEL'S TX.
 132                            *                 ALL REGISTERS UNALTERED.
 133                            *
 134                            *
 135
 136   00002058 082800020002 OUTCH    BTST.B   #2,2(A0)              WAIT FOR CHANNEL'S TX TO BECOME READY
 137   0000205E 67F8                  BEQ      OUTCH
 138   00002060 11400006              MOVE.B   D0,6(A0)              SEND CHAR TO TRANSMITTER
 139   00002064 0C000000              CMP.B    #CR,D0                WAS IT A CARRIAGE RETURN?
 140   00002068 660E                  BNE      OUTCHR                NO, SKIP NEXT PART
 141   0000206A 082800020002 OUTCH1   BTST.B   #2,2(A0)              YES, WAIT FOR TX TO BECOME READY AGAIN
 142   00002070 67F8                  BEQ      OUTCH1
 143   00002072 117C000A0006          MOVE.B   #LF,6(A0)             SEND A LINE FEED
 144   00002078 4E75          OUTCHR  RTS
 145                                   END

****** TOTAL ERRORS      0--
****** TOTAL WARNINGS    0--
```

```
MOTOROLA M68000 ASM VERSION  1.30SYS :   5.        .MINSYSS .SA 01/20/84 13:44:18          PAGE 4 of 4
MINSYSS

SYMBOL TABLE LISTING

SYMBOL NAME      SECT   VALUE       SYMBOL NAME      SECT   VALUE

ACR                     000F0009     ISR                     000F000B
BTRST                   000F001F     IVR                     000F0019
BTST                    000F001D     LF                      0000000A
CHANA                   000F0001     MR1A                    000F0001
CHANB                   000F0011     MR1B                    000F0011
CLSB                    000F000F     MR2A                    000F0001
CMSB                    000F000D     MR2B                    000F0011
CR                      0000000D     OPCR                    000F001B
CRA                     000F0005     OUTCH                   00002058
CRB                     000F0015     OUTCH1                  0000206A
CSRA                    000F0003     OUTCHR                  00002078
CSRB                    000F0013     RBA                     000F0007
CTLR                    000F000F     RBB                     000F0017
CTUR                    000F000D     SRA                     000F0003
DINIT                   00002000     SRB                     000F0013
DUART                   000F0001     STPC                    000F001F
IMR                     000F000B     STRC                    000F001D
INCH                    0000204A     TBA                     000F0007
IP                      000F001B     TBB                     000F0017
IPCR                    000F0009
```

10.6 80286, två processorer i en kapsel (1982)

80286:an var revolutionerande för Intel, även om få kunde utnyttja 286:ans resurser fullt ut från början. Processorn var egentligen två processorer i en kapsel. 286:an var dels en snabbare 8086, men också en helt ny processor med möjlighet att *adressera*[8] hela 1 GB minne! I den vanliga PC:n användes den nästan uteslutande som en snabbare 8086-processor.

När 80286-processorn strömsätts ser den ut som en vanlig 8086, den har samma minnesmodell o s v, men är något snabbare. Med speciella instruktioner kan den byta skepnad helt och hållet. Den kan fås att övergå i s k *protected mode*, en mod där segmenten blir begåvade med *minnesskydd* och processorn kan adressera ända upp till 1 GB minne.

De processorer från Intel som följde (386/486/Pentium o s v) har många drag gemensamma med 286:an, varför vi ägnar lite tid åt dess egenskaper.

10.6.1 Virtuellt minne

Det tog inte lång tid förrän programmen blev större än att de kunde få plats i datorns minne. Vi vet att programmet måste befinna sig i minnet för att kunna köras och under en övergångstid på några år användes s k *overlay*er. Med det menas att programmet styckades upp i flera delar och varje sådant programstycke, overlay, visste var dess fortsättning var — i en annan overlay.

Detta gjordes när disketter fortfarande användes för att lagra program! Det gällde att vara på plats och byta diskett när programmet ville byta overlay. Ett problem var under alla förhållanden att programräknaren inte längre räckte till att peka ut alla rader i ett stort program. Ett annat problem vid denna tid var att primärminne var dyrt och datorerna inte var utrustade med så mycket, bara kanske 512 kb. Om overlayen inte var tillräckligt liten kanske den överhuvudtaget inte fick plats i det tillgängliga minnet. Alternativen var då att antingen köpa mer minne eller att inte kunna köra programmet.

Även om vi utrustar processorn med 32-bitars programräknare — det borde räcka till — fanns det då inga möjligheter i världen att förse datorn med sådana mängder minne. En annan lösning söktes och den heter *virtuellt minne*. Med virtuellt minne kan 80286 adressera 1 Gb minne och *varje* program kan köras i villfarelsen att det har tillgång till denna mycket stora mängd minne!

Virtuellt minne är ingen uppfinning i och med 80286 utan har i stordatorsammanhang funnit sedan 1960-talet.

Vi tacklar först hur man med hjälp av virtuellt minne på 80286 plötsligt kan adressera hela 1 Gb minne. Det register som förr hette CS kallas i virtuellt minnessammanhang för en *selektor* och används för att peka ut en post, en s k *deskriptor*, i en *deskriptortabell*. Varje segment i minnet representeras av en post i denna deskriptortabell. Den utpekade posten innehåller information om

- segmentets bas(start)adress
- segmentets längd

[8] Lägg märke till att det står adressera, inget annat!

- flaggor, som beskriver vad som får hända med segmentet.

Vi ser att den virtuella adressen (VA) mycket väl kan vara 32 bitar lång men det resulterande, fysiska, minnet som adressen pekar ut är fortfarande bara 16 Mb stort. Den stora *virtuella* minnesrymden mappas alltså om till en mindre *fysisk* minnesrymd. Man kan inte ansluta minneskapslar för 1 Gb minne till kretsen. När man lött fast minne motsvarande 16 Mb finns det inte pinnar kvar på processorn att ansluta mer minne på!

10.6.2 Vad är då poängen med virtuellt minne?

OK, så vi kan adressera mängder med minne men vi kan inte ansluta så mycket minne till processorn. Det är ju bara trixande. Vem försöker vi lura?

- En väsentlig vits, troligen den största, är att man kan köra *godtyckligt stora program* — upp till hela det stora virtuella minnesutrymmet på 1 GB i varje fall — även om man inte har ett fysiskt minne som är så stort. Bara de delar av programmet som behövs för stunden läggs i det fysiska minnet, resten kan ligga kvar på hårddisken tills det behövs. Det liknar metoden med overlay en del, men med skillnaden att alla programdelar ryms *i samma adressrymd*. Man kommer naturligtvis att behöva ”göra plats” i minnet innan mer data överförs men det är inget större programmeringstekniskt problem. Programmet tror sig under alla förhållanden ha tillgång till hela den virtuella minnesmängden för sig själv.

  Man kan här också använda samma resonemang som för cache-minnen: Med en effektiv algoritm s k *ersättningsalgoritm* som utnyttjar *locality of reference*-egenskapen kommer det mindre fysiska minnet att ur processorns synvinkel uppföra sig som det stora virtuella minnet — med en hastighet som närmar sig det fysiska minnets.

- Man kan låta flera program dela på minnet utan att de vet om varann. I det stora fysiska adressområdet får säkert flera program plats. Om ett program behöver mer plats kan systemet allokera in mer minnesyta, kanske genom att flytta ut andra program. Detta sker automatiskt om till exempel exekveringen hoppar till en plats i programmet som inte finns i det fysiska minnet. En bit i deskriptorn avslöjar om det önskade segmentet är i det fysiska minnet eller inte. Om det inte är inladdat

ombesörjer processorn automatiskt denna inladdning, något som kallas *demand-driven virtual memory*.

- Om man ändå delar upp minnet i bitar med deskriptorer[9] har man krattat manegen för att samtidigt på ett smidigt sätt införa *minnesskydd*.

Med minnesskydd menas en processormekanism som begränsar det adressområde ett program kan nå. Under programmets gång jämförs alla adressreferenser programmet gör med dessa gränser. Processorn har alltså möjlighet att under programkörning övervaka vad programmet tar sig för — och kan larma[10] om det försöker göra något otillåtet.

Minneskyddet kan enkelt knytas till minnesområdenas deskriptorer i den segmenterade minnesmodellen. För lineärt minne lagras gränserna i processorregister direkt, men man brukar för enkelhets skull klumpa ihop registerinnehållet i deskriptorer i alla fall för att få en lättarbetad struktur åt mekanismen.

10.6.3 Hur ser en bra ersättningsalgoritm ut?

Om vi tänker tillbaka på cacheminnet förlitade sig cachemekanismen på en algoritm[11] som såg till att h var nära 1. Hemligheten var att hålla de mest använda programraderna eller datana i cacheminnet för snabb åtkomst. Det svåra var att konstruera en algoritm som tillräckligt bra kunde avgöra vilka data som skulle *komma* att användas. På samma sätt måste en ersättningsalgoritm för det virtuella minnet, i idealfallet, kunna se in i framtiden. Algoritmen kan förstås inte se in i framtiden utan måste förlita sig på lokal och temporal lokalitet.

Så länge primärminnet räcker till att husera den använda virtuella minnesmängden behöver ingen ersättningsalgoritm användas. Men när primärminnet är fullt och programmet behöver ytterligare programrader eller data måste några gamla kastas ut från primärminnet. Om de kan skrivas över direkt eller om de måste sparas undan beror naturligtvis på hur enkelt de kan läsas in på nytt (programrader) eller om de blivit ändrade (data). Om de skall sparas görs detta lämpligen på hårddisken.

[9] Det kanske är lättare att föreställa sig detta om minnet redan är segmentindelat, men även 68000-familjen har deskriptorer för VA-hanteringen även om minnet i övrigt är lineärt.
[10] Vi minns undantagshantering i 68000. Även i detta fall kommer en TRAP att genereras.
[11] Visserligen i hårdvara men en algoritm likaväl.

Hur ska algoritmen välja vilket segment som ska bytas ut?

Nedan blandas begreppen ”sida” med ”segment”. För att förenkla framställningen har det inte avlöjats att man nuförtiden inte använder segmentet som den minsta minnesmängden utan den så kallde *sidan*. Men det spelar ingen roll här. En ersättningsalgoritm kan tänkas arbeta mot sida eller segment, principen är densamma.

Medan segment kan vara olika stora är en *sida* alltid lika stor, till exempel i Intels fall 4 kb. Ett segment byggs upp av ett antal sådana sidor. Att byta ut ett segment, som i det generella fallet kan anta vilken storlek som helst, är krångligare än att byta en sida med en (1) bestämd storlek och effektiviteten ökar då man inte måste byta ut ett helt (stort) segment utan bara ett antal sidor med var för sig känd storlek.[12]

- Enklast är förstås att hugga ett segment på måfå. Man riskerar förstås att emellanåt ta bort ett segment som snart skall användas men eftersom antalet segment är många är risken att välja ”precis fel”, rätt liten.
- FIFO, *First-In-First-Out* dvs byt alltid ut den äldsta sidan. En tidräknare behövs då till varje sida. Men om den äldsta sidan är den mest använda då?
- LRU, *Least Recently Used*, som vi kommer ihåg från cacheminnesfallet, väljer den sida som använts minst hittills.
- FINUFO, *First-In Not-Used First-out*, tar hänsyn till nyttjandet av respektive sida och kastar ut den som är äldst och dessutom använts minst. Både en tid- och nyttjanderäknare behövs.

Deskriptortabellen ligger förstås i minnet självt och lider därför av den gamla 64 kB segmentstorleken på så sätt att det inte kan finnas mer än cirka 8000 deskriptorer i ett 286-system. Nu är detta inget egentligt problem, 8000 segment är många.

Med 8000 deskriptorer kan man nu också tillåta flera program att ligga i minnet samtidigt, men bara det program som pekas ut av deskriptorregistret i processorn för tillfället, kommer att köras. Det finns ju bara en instruktionspekare, IP, så även om vi vill, kan vi inte peka ut kod i flera segment samtidigt.

Man kan till och med tänka sig ett överordnat program som med automatik roterar mellan alla dessa olika program, d v s ett system där kanske tio program samtidigt befinner sig i minnet, men endast ett körs åt gången. Precis detta sker numera i alla vanliga operativsystem: Flera program tilldelas processorns resurser för en liten tid, typiskt ett tiotal millisekunder. På detta sätt *verkar* det som om flera program körs samtidigt, vilket är omöjligt eftersom det bara finns en centralenhet. Men upplevelsen är i varje fall att programmen löper samtidigt.

[12] En *swap*-yta/växlingsfil på en hårddisk är enklare att administrera om man vet att det som ska passas in är av en bestämd storlek. Till exempel riskerar man inte att behöva stycka upp segmentet för att få plats med det om segment bara kan ha en storlek. Med variabla segment kan man behöva kompaktera (dvs maka ihop de använda delarna) swap-ytan för att få plats.

10.6.4 Deskriptorflaggorna

Flaggorna i deskriptorn har en del intressanta egenskaper. De ingår i det system av *minnesskydd* hela processorn får då man sätter den i *protected mode*.

Flaggorna utgör vad Intel kallar för segmentets *access rights*, åtkomsträttigheter, och ser lite olika ut beroende på om det är ett kodsegment eller ett datasegment. En bit i access rights-byten avgör dessutom om deskriptorn avser ett kod- eller ett datasegment.

Flaggorna möjliggör bland annat följande funktioner:

- För ett kodsegment:
  - Tillåt bara exekvering av koden, förbjud läsning av koden. Enbart IP kan peka ut adresser i segmentet och få den exekverad. Det går alltså att hindra program från att snoka runt i minnet och kolla på andra program
- För ett datasegment:
  - Tillåt bara läsning av data, förbjud skrivning till segmentet. Man kan således hindra felaktig programkod eller medvetet illasinnat skriven kod att korrumpera data.
  - En bit i flaggregistret avgör om segmentet ska kunna utökas uppåt eller nedåt. En stack är ett datasegment som oftast växer ”nedåt” i minnet. För övriga segment är det däremot naturligt att låta dem utökas uppåt i minnet.

I samtliga deskriptorer finns dessutom två bitar som avgör vilken *privilege level*, privilegienivå, segmentet tillhör. Det är inte bara segmentflaggorna som kan hindra läsning av kod respektive ändring av data. Med *privilegiemekanismen* kan man dessutom hindra hela grupper av program från att överhuvudtaget komma åt andra grupper av segment.

I Intels universum finns fyra privilegienivåer: 0, 1, 2 och 3. Nivå 3 är den lägsta privilegienivån, nivå 0 den högsta. Program som kör i segment med privilegien satt till 0 har tillgång till hela processorn, program i övriga nivåer kan behöva be en högre nivå att utföra vissa operationer åt dem. Speciellt finns det en del systemkritiska instruktioner (HALT-instruktionen till exempel) som bara kan köras av kod i nivå 0.

Om det är välskriven, testad och säker kod, kod som aldrig kraschar, som körs i nivå 0, och alla andra program körs i någon annan, d v s lägre, nivå, kommer systemet som helhet inte att *kunna* krascha. Om någon kod missköter sig kommer processorn att svara med att avbryta programmet med en TRAP-instruktion precis som vi är vana vid från 68000. Programexeveringen förs över till undantagshanteringskoden i nivå 0, som får en möjlighet att reda ut situationen — kanske stoppa och radera det felaktiga programmet — innan processorn lämnar över till nästa program.

Processorns hårdvara har alltså en inneboende förmåga att upptäcka oegentligheter. En förmåga som inte på något sätt *kan* konstrueras med hjälp av programkod utan måste utföras av hårdvara på en lägre nivå än att någon programvara är inblandad — utom möjligen mikrokod inuti processorn förstås. I samtliga fall genereras en TRAP av någon sort, exempelvis:

- Om VA pekar utanför segment => TRAP
- Om VA pekar ut minne som inte finns => TRAP

- Om VA pekar ut minne som inte ingår i tillåten privilegienivå => TRAP
- Om processorn försöker exekvera från ett datasegment => TRAP

10.6.5 Multipla program

Vi har sett att man kan tänka sig att rotera mellan olika program[13] genom att peka ut en deskriptor som i sin tur pekar ut ett kodsegment. Processorn 80286 har en mekanism för att göra detta så smidigt som överhuvudtaget är möjligt. Samma mekanism återfinns i andra processorer. Vi vet att det i ett program ingår åtminstone kod och möjligen även data och en stack. I Intels fall ligger dessa i olika segment, och en deskriptor pekar ut rätt (samhörande) segment utgående från en virtuell adress.

För att stödja multipla processer kan man definiera ett speciellt TASK-segment som innehåller de data som är privata för processen, här lagras bland annat processorns *inre tillstånd* då processen inte körs. Då processen körs ligger naturligtvis det inre tillståndet i processorn.

Med 286:ans upplägg är processbyte enkelt. Det räcker att ändra ett värde i ett speciellt processorregister, TR, *Task Register*, och processorn ombesörjer allt jobb som krävs för att kasta ut den pågående processen, spara processorns inre tillstånd och förse processorn med den nya TASK:en. TASK-registret innehåller, förstår vi, adressen till det TASK-segment som hör till den nu aktiva pågående processen.

[13] Det vi kallar ”ett program som körs” kallas också i vissa sammanhang för process eller uppgift, *task*. Vi gör ingen stor sak av eventuella skillnader.

10.7 Datablad CY6264

PRELIMINARY

CY6264

8K x 8 Static RAM

Features

- **55, 70 ns access times**
- **CMOS for optimum speed/power**
- **Easy memory expansion with $\overline{CE}_1$, CE_2, and $\overline{OE}$ features**
- **TTL-compatible inputs and outputs**
- **Automatic power-down when deselected**

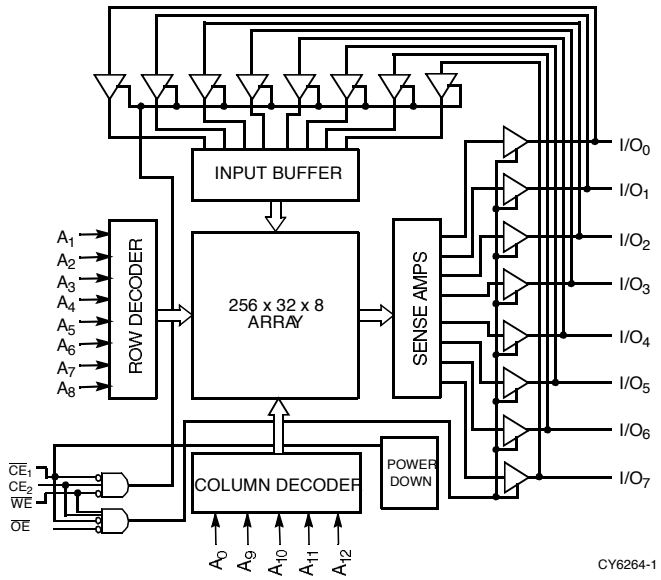
Functional Description

The CY6264 is a high-performance CMOS static RAM organized as 8192 words by 8 bits. Easy memory expansion is provided by an active LOW chip enable ($\overline{CE}_1$), an active HIGH chip enable (CE_2), and active LOW output enable ($\overline{OE}$) and three-state drivers. Both devices have an automatic power-down feature ($\overline{CE}_1$), reducing the power consumption by over 70% when deselected. The CY6264 is packaged in a 450-mil (300-mil body) SOIC.

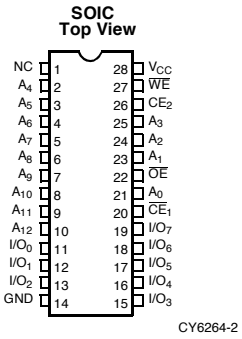
An active LOW write enable signal ($\overline{WE}$) controls the writing/reading operation of the memory. When $\overline{CE}_1$ and $\overline{WE}$ inputs are both LOW and CE_2 is HIGH, data on the eight data input/output pins (I/O_0 through I/O_7) is written into the memory location addressed by the address present on the address pins (A_0 through A_{12}). Reading the device is accomplished by selecting the device and enabling the outputs, $\overline{CE}_1$ and $\overline{OE}$ active LOW, CE_2 active HIGH, while $\overline{WE}$ remains inactive or HIGH. Under these conditions, the contents of the location addressed by the information on address pins is present on the eight data input/output pins.

The input/output pins remain in a high-impedance state unless the chip is selected, outputs are enabled, and write enable ($\overline{WE}$) is HIGH. A die coat is used to insure alpha immunity.

Logic Block Diagram



Pin Configuration



Selection Guide

	CY6264-55	CY6264-70
Maximum Access Time (ns)	55	70
Maximum Operating Current (mA)	100	100
Maximum Standby Current (mA)	20/15	20/15

Shaded area contains advanced information.

Cypress Semiconductor Corporation • 3901 North First Street • San Jose • CA 95134 • 408-943-2600
October 1994 – Revised June 1996

Maximum Ratings

(Above which the useful life may be impaired. For user guidelines, not tested.)

Storage Temperature ..... –65 C to +150 C

Ambient Temperature with Power Applied..... –55 C to +125 C

Supply Voltage to Ground Potential..... –0.5V to +7.0V

DC Voltage Applied to Outputs in High Z State[1]..... –0.5V to +7.0V

DC Input Voltage[1]..... –0.5V to +7.0V

Output Current into Outputs (LOW)..... 20 mA

Static Discharge Voltage ..... >2001V (per MIL-STD-883, Method 3015)

Latch-Up Current..... >200 mA

Operating Range

Range	Ambient Temperature	V_{CC}
Commercial	0 C to +70 C	5V 10%

Electrical Characteristics Over the Operating Range

Parameter	Description	Test Conditions	6264-55		6264-70		Unit
			Min.	Max.	Min.	Max.	
V_{OH}	Output HIGH Voltage	V_{CC} = Min., I_{OH} = –4.0 mA	2.4		2.4		V
V_{OL}	Output LOW Voltage	V_{CC} = Min., I_{OL} = 8.0 mA		0.4		0.4	V
V_{IH}	Input HIGH Voltage		2.2	V_{CC}	2.2	V_{CC}	V
V_{IL}	Input LOW Voltage[1]		–0.5	0.8	–0.5	0.8	V
I_{IX}	Input Load Current	GND ≤ V_I ≤ V_{CC}	–5	+5	–5	+5	µA
I_{OZ}	Output Leakage Current	GND ≤ V_I ≤ V_{CC}, Output Disabled	–5	+5	–5	+5	µA
I_{OS}	Output Short Circuit Current[2]	V_{CC} = Max., V_{OUT} = GND		–300		–300	mA
I_{CC}	V_{CC} Operating Supply Current	V_{CC} = Max., I_{OUT} = 0 mA		100		100	mA
I_{SB1}	Automatic $\overline{CE}_1$ Power–Down Current	Max. V_{CC}, $\overline{CE}_1$ ≥ V_{IH}, Min. Duty Cycle=100%		20		20	mA
I_{SB2}	Automatic $\overline{CE}_1$ Power–Down Current	Max. V_{CC}, $\overline{CE}_T$ ≥ V_{CC} – 0.3V, V_{IN} ≥ V_{CC} – 0.3V or V_{IN} ≤ 0.3V		15		15	mA

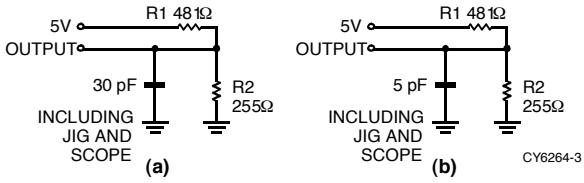
Shaded area contains advanced information.

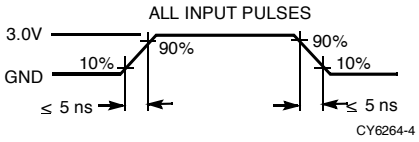
Capacitance[3]

Parameter	Description	Test Conditions	Max.	Unit
C_{IN}	Input Capacitance	T_A = 25 C, f = 1 MHz, V_{CC} = 5.0V	7	pF
C_{OUT}	Output Capacitance		7	pF

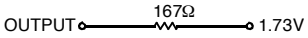
Notes:
1. Minimum voltage is equal to -3.0V for pulse durations less than 30 ns.
2. Not more than 1 output should be shorted at one time. Duration of the short circuit should not exceed 30 seconds.
3. Tested initially and after any design or process changes that may affect these parameters.

AC Test Loads and Waveforms





Equivalent to: THÉVENIN EQUIVALENT



Switching Characteristics Over the Operating Range[4]

Parameter	Description	6264-55		6264-70		Unit
		Min.	Max.	Min.	Max.	
READ CYCLE						
t_{RC}	Read Cycle Time	55		70		ns
t_{AA}	Address to Data Valid		55		70	ns
t_{OHA}	Data Hold from Address Change	5		5		ns
t_{ACE1}	$\overline{CE}_1$ LOW to Data Valid		55		70	ns
t_{ACE2}	CE_2 HIGH to Data Valid		40		70	ns
t_{DOE}	$\overline{OE}$ LOW to Data Valid		25		35	ns
t_{LZOE}	$\overline{OE}$ LOW to Low Z	3		5		ns
t_{HZOE}	$\overline{OE}$ HIGH to High Z[5]		20		30	ns
t_{LZCE1}	$\overline{CE}_1$ LOW to Low Z[6]	5		5		ns
t_{LZCE2}	CE_2 HIGH to Low Z	3		5		ns
t_{HZCE}	$\overline{CE}_1$ HIGH to High Z[5, 6] CE_2 LOW to High Z		20		30	ns
t_{PU}	$\overline{CE}_1$ LOW to Power-Up	0		0		ns
t_{PD}	$\overline{CE}_1$ HIGH to Power-Down		25		30	ns
WRITE CYCLE[7]						
t_{WC}	Write Cycle Time	50		70		ns
t_{SCE1}	$\overline{CE}_1$ LOW to Write End	40		60		ns
t_{SCE2}	CE_2 HIGH to Write End	30		50		ns
t_{AW}	Address Set-Up to Write End	40		55		ns
t_{HA}	Address Hold from Write End	0		0		ns
t_{SA}	Address Set-Up to Write Start	0		0		ns
t_{PWE}	$\overline{WE}$ Pulse Width	25		40		ns
t_{SD}	Data Set-Up to Write End	25		35		ns
t_{HD}	Data Hold from Write End	0		0		ns
t_{HZWE}	$\overline{WE}$ LOW to High Z[5]		20		30	ns
t_{LZWE}	$\overline{WE}$ HIGH to Low Z	5		5		ns

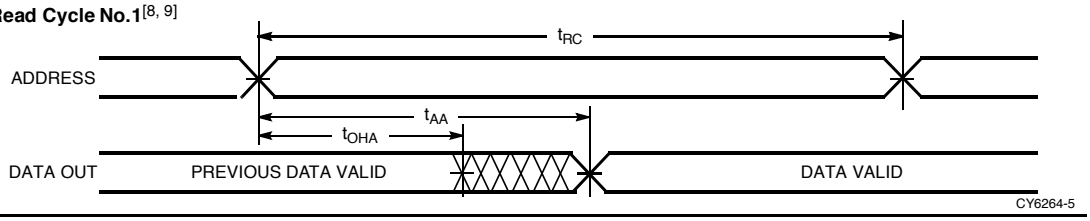
Shaded area contains advanced information.

Notes:

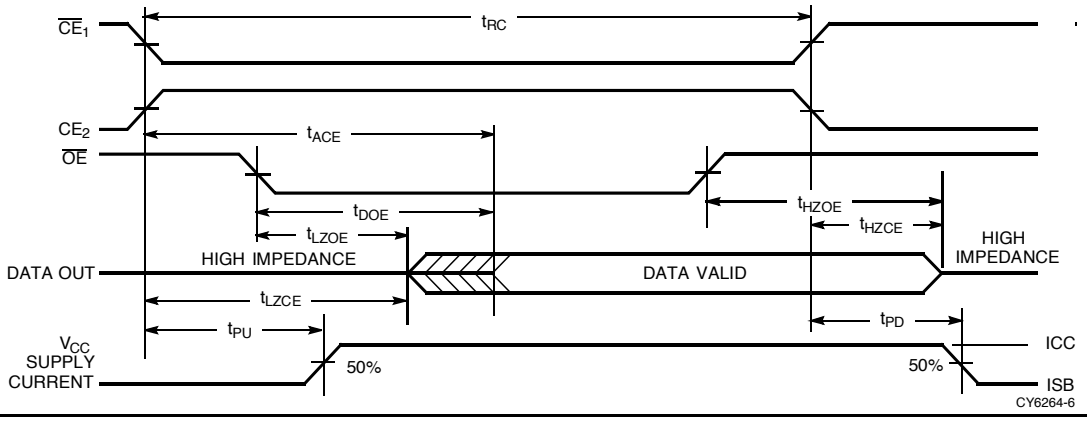
4. Test conditions assume signal transition time of 5 ns or less, timing reference levels of 1.5V, input pulse levels of 0 to 3.0V, and output loading of the specified I_{OL}/I_{OH} and 30-pF load capacitance.
5. t_{HZOE}, t_{HZCE}, and t_{HZWE} are specified with C_L = 5 pF as in part (b) of AC Test Loads. Transition is measured 500 mV from steady-state voltage.
6. At any given temperature and voltage condition, t_{HZCE} is less than t_{LZCE} for any given device.
7. The internal write time of the memory is defined by the overlap of $\overline{CE}_1$ LOW, CE_2 HIGH, and $\overline{WE}$ LOW. Both signals must be LOW to initiate a write and either signal can terminate a write by going HIGH. The data input set-up and hold timing should be referenced to the rising edge of the signal that terminates the write.

Switching Waveforms

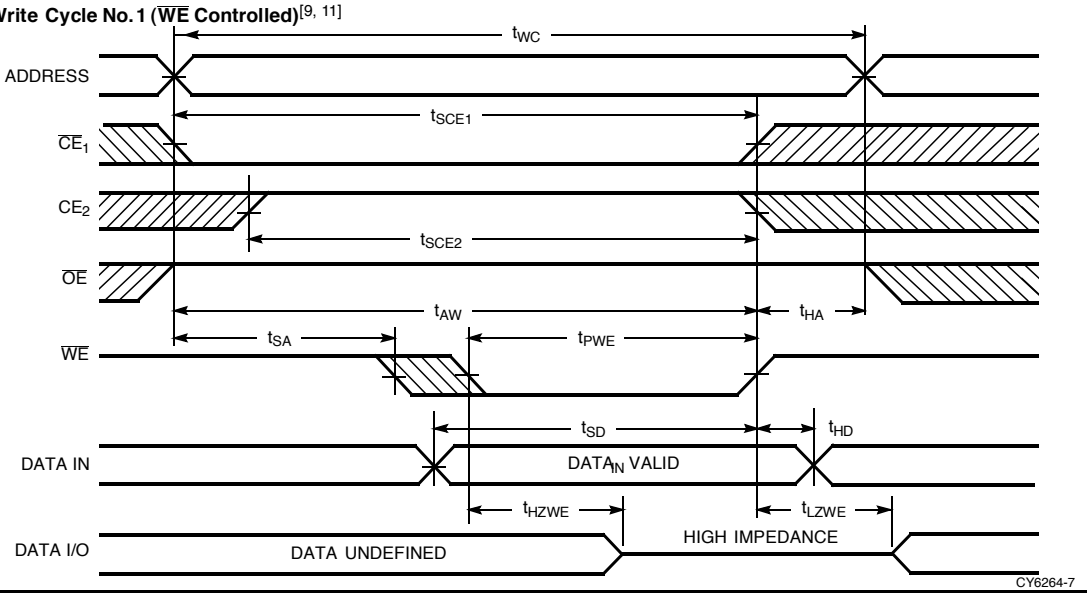
Read Cycle No.1[8, 9]



Read Cycle No. 2[10, 11]



Write Cycle No.1 (WE Controlled)[9, 11]

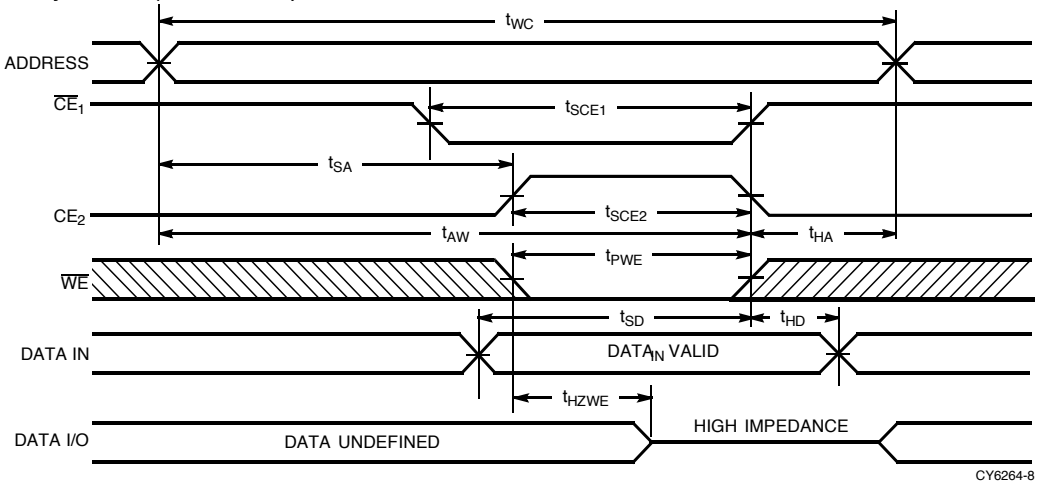


Notes:

8. Device is continuously selected. $\overline{OE}$, $\overline{CE}$ = V_{IL}, CE_2 = V_{IH}.
9. Address valid prior to or coincident with $\overline{CE}$ transition LOW.
10. $\overline{WE}$ is HIGH for read cycle.
11. Data I/O is High Z if $\overline{OE}$ = V_{IH}, $\overline{CE}_1$ = V_{IH}, or $\overline{WE}$ = V_{IL}.

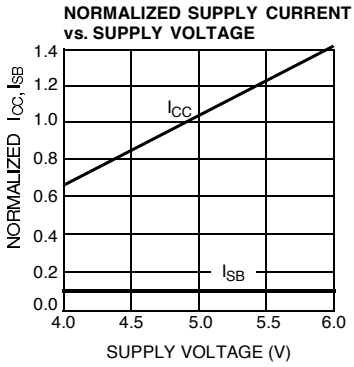
Switching Waveforms (continued)

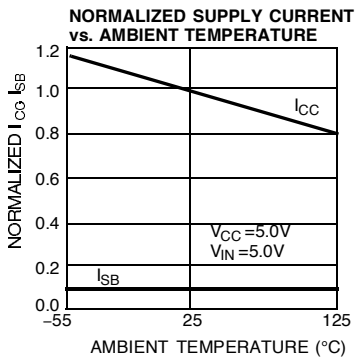
Write Cycle No. 2 ($\overline{CE}$ Controlled)[9, 11, 12]

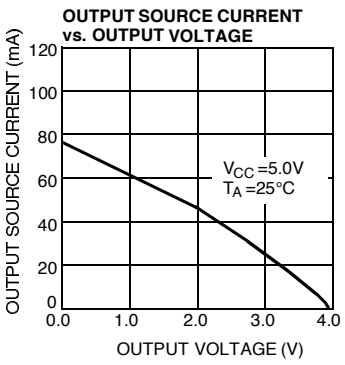


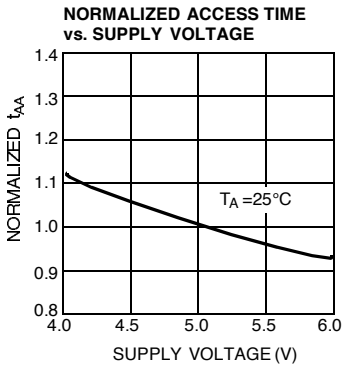
Note:
12. If $\overline{CE}$ goes HIGH simultaneously with $\overline{WE}$ HIGH, the output remains in a high-impedance state.

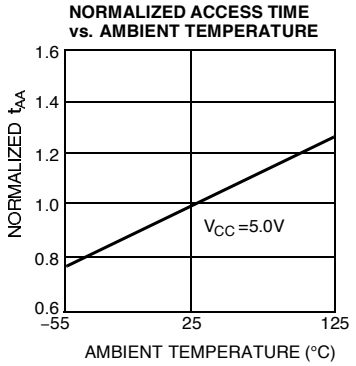
Typical DC and AC Characteristics

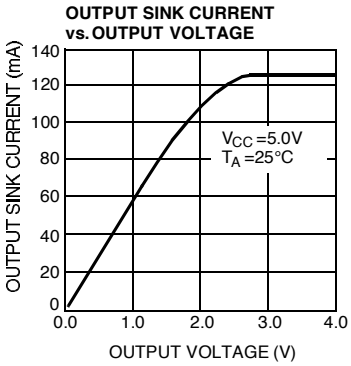




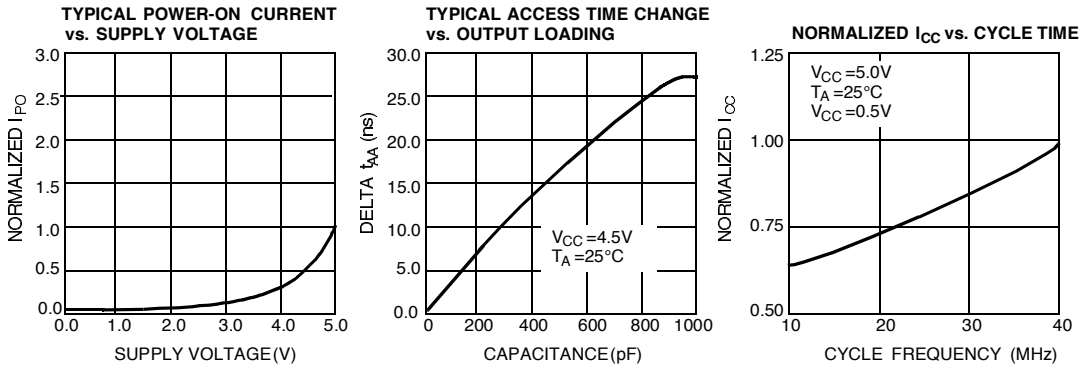








Typical DC and AC Characteristics (continued)



Truth Table

$\overline{CE}_1$	CE_2	$\overline{WE}$	$\overline{OE}$	Input/Output	Mode
H	X	X	X	High Z	Deselect/Power-Down
X	L	X	X	High Z	Deselect
L	H	H	L	Data Out	Read
L	H	L	X	Data In	Write
L	H	H	H	High Z	Deselect

Address Designators

Address Name	Address Function	Pin Number
A4	X3	2
A5	X4	3
A6	X5	4
A7	X6	5
A8	X7	6
A9	Y1	7
A10	Y4	8
A11	Y3	9
A12	Y0	10
A0	Y2	21
A1	X0	23
A2	X1	24
A3	X2	25

Ordering Information

Speed (ns)	Ordering Code	Package Name	Package Type	Operating Range
55	CY6264-55SC	S23	28-Lead 330-Mil SOIC[13]	Commercial
70	CY6264-70SC	S23	28-Lead 330-Mil SOIC[13]	Commercial
55	CY6264-55SNC	S22	28-Lead 300-Mil SOIC	Commercial
70	CY6264-70SNC	S22	28-Lead 300-Mil SOIC	Commercial

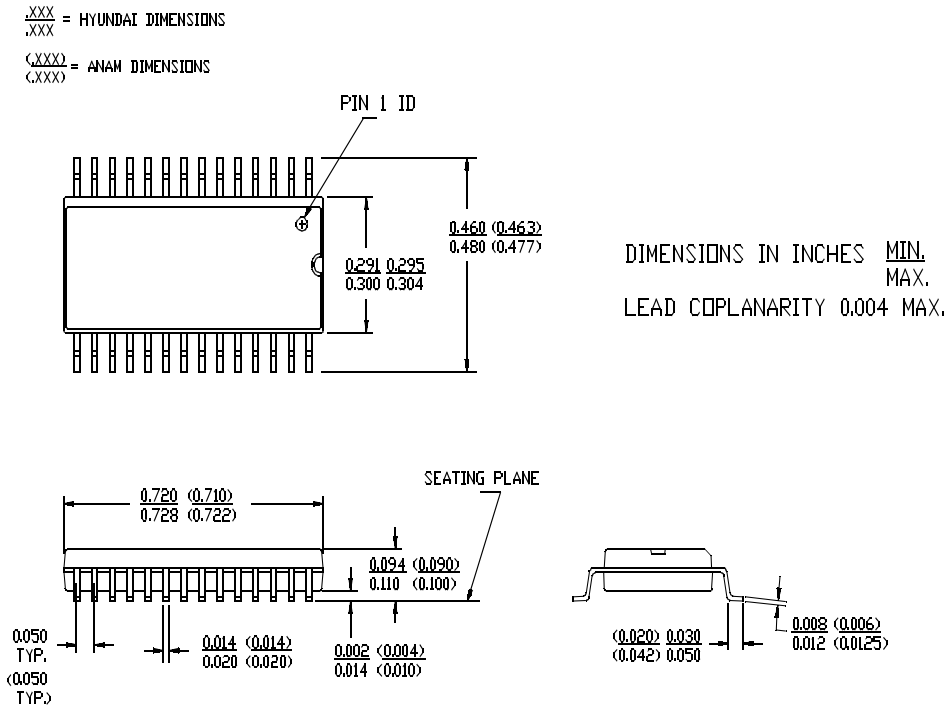
Shaded area contains advanced information.

Note:
13. Not recommended for new designs.

Document #: 38-00425-A

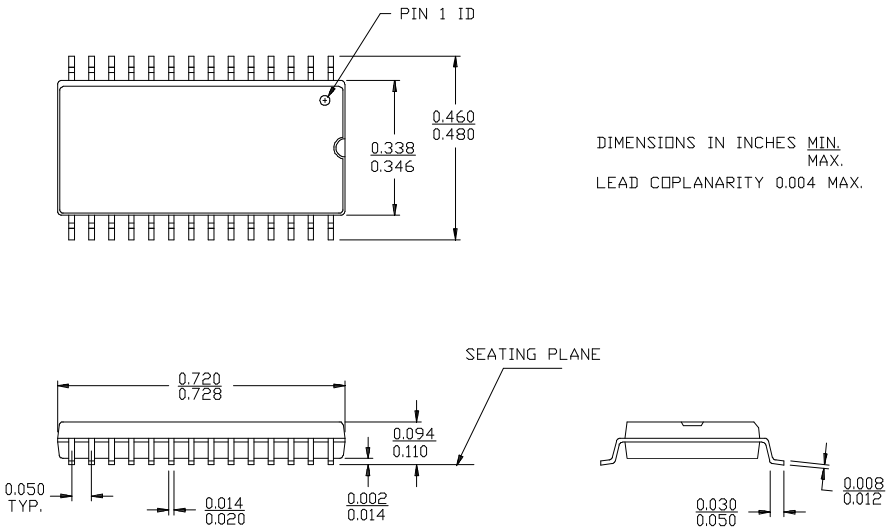
Package Diagrams

28-Lead 450-Mil (300-Mil Body Width) SOIC S22



Package Diagrams (continued)

28-Lead (330-Mil) SOIC S23



11 Hårddiskar

Begrepp i denna föreläsning: *Minneshierarki, ärnminne, cylinder, sektor, spår, MBR, partition, filallokeringstabell, filsystem, rotkatalog, allokeringspolicy, intern och extern fragmentering, FFS, inod, block, cylindergrupp, journalförande filsystem, konkatenering av diskar, RAID-0,1,5.*

11.1 Inledning — minneshierarki

Vi har sett hur data lagras i olika nivår i ett processorsystem. Med början inuti processorn lagrades informationen i register, utanför processorn kunde datat finnas i ett snabbt men litet cacheminne, eller ett större och långsammare primärminne. Det data som inte får plats i primärminnet måste lagras på ett mer permanent sätt. Till detta använder man praktiskt taget uteslutande hårddiskar, som har mycket större lagringskapacitet än något ekonomiskt gångbart minne. Åtkomsttiden för ett data lagrat på en hårddisk är samtidigt mycket större än för något elektriskt minne. Kan man acceptera ännu längre åtkomsttider är lagring på band ett kostnadseffektivt alternativ.

Några exempel på åtkomstider och lagringsstorlekar för en dylik *minneshierarki*[1] ges i tabellen nedan. Dessa siffror är bara typvärden tagna på en höft, så läs inte för bokstavligt.

Minnestyp	Storlek (bytes)	Åtkomsttid (sekunder)
Register	enstaka	10^{-10}
Cache	någon megabyte	10^{-9}
Primärminne	hundratals megabyte	10^{-8}
Hårddisk	tiotals gigabyte	10^{-3}
Band	tiotals terabyte	10^{3}

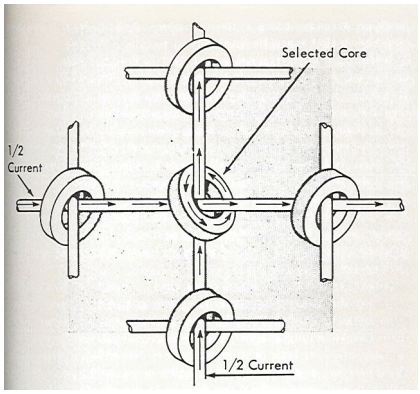
Man ser att likheten med cacheminnet är påfallande. Ett litet, snabbt minne närmast processorn och ett större, något långsammare lite längre bort. Man kan naturligtvis inte låta det större minnet vara hur långsamt som helst utan vill ha det så snabbt som tekniken och portmonnän tillåter.

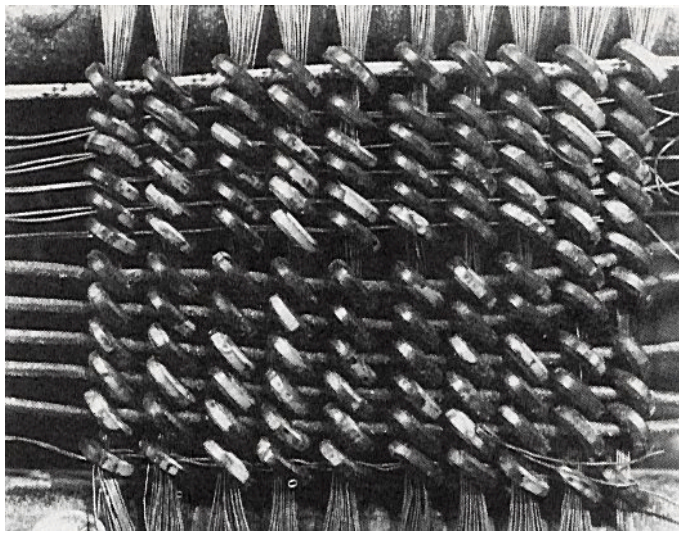
Historiskt använde man kärnminnet som det lilla, snabba minnet och magnetband för masslagring av data och program. Numer används kiselbaserade minnen närmast processorn och hårddiskar för masslagring. Band används fortfarande men bara för ”vilande” datalagring, backup. Hur det kiselbaserade minnet fungerar har vi sett tidigare men kärnminnet, magnetbandet och hårddisken är nya komponenter för oss.

[1] Internet borde också vara med på ett hörn här. Var vore isåfall lämpligt?

11.2 Kärnminne

Jag har hittat en siffra som anger att så sent som 1976 använde 95% av alla stordatorer kärnminnen som primärminne. Bland fördelarna med denna minnesttyp kan nämnas att minnet bevaras vid spänningsfrånslag och att kärnminnnet är tåligt mot strålning. Men det är fysiskt stort och tidsödande att virka ihop... I bilden nedan lagras en bits information i varje liten toroid:





Som alla andra magnetlagringstekniker förlitar sig kärnminnet på den hystereskurva som magnetiska material uppvisar. Den är grunden för alla magnetbaserade minne vare sig det rör sig om kärnminne, magnetband eller hårddisken.

11.3 Hårddisk

Hårddisken är en platt cirkelrund aluminiumskiva som roterar kring sin egen axel. Skivstorleken är vanligen 3,5 tum, men även mindre börjar bli vanliga. Skivan är försedd med ett mycket tunt magnetiskt lager, cirka 10 nm tjockt. Skrivning sker genom lokal magnetisering av det område där *läs-/skrivhuvudet* just är. För att hjälpa läs-/skrivhuvudet att positionera sig finns, förutom själva datainformationen, magnetiserade styrspår på skivan.

Numera är alla typiska lagringsvolymer i storleksordningen flertalet 10-tals gigabyte. Med cirka 500 Gb som en gräns för dagen (2006). Man förstår att läs-/skrivhuvudet är oerhört litet för att kunna fysiskt peka ut varje enskild databit. Packningstätheten är så hög att de magnetiserade områdenas magnetisering flyter ihop och speciella kodningar för informationen behövs så att den lagrade datatakten blir så *låg* som möjligt, dvs magnetiseringens flödesväxlingar kommer så *långt* från varann som möjligt.

Skivans rotationshastighet var under lång tid standardiserad till 3 600 rpm[2], men numera förekommer även 5 400, 7 200 rpm. 10 000 rpm är inte ovanligt. En hög rotationshastighet medför naturligtvis att överföringstakten till och från skivan också blir hög. Även om rotationshastigheten är hög finns dock fysiska begränsningar som hindrar *åtkomsttiden*

[2]rpm=*revolutions per minute*, varv per minut.

att bli mycket snabbare än cirka 5 millisekunder. Läshuvudet sitter mekaniskt på en arm som måste styras mycket noggrant med ett avancerat servosystem. Fysiska trögheten, på grund av de involverade massorna, i detta mekaniska system utgör för närvarande den största begränsningen i vägen för kortare åtkomsttid. Positioneringen tar alltså tid, medan själva överföringen, när läshuvudet ”dammsuger” disken på data, är betydligt snabbare.

I och med att skivan roterar byggs en liten luftkudde upp längs ytan och precis i detta ytskikt kan man få en aerodynamiskt utformat läs/skriv-huvud att sväva. Avståndet till diskens yta är då cirka 70 nm. Det är ett oerhört precisionsjobb att tillverka läshuvudet aerodynamiskt — så lite som avtrycket från ett fingeravtryck förstör de aerodynamiska egenskaperna helt.

Läshuvudet får inte beröra skivan under gång. Med en hastighet om 3 600 rpm och däröver skulle friktionen omedelbart ge en sådan värmeutveckling att läshuvudet bränns upp eller svetsas fast i skivan.

Informationen skrivs i cirklar på skivan och för att lättare kunna hitta informationen är skivan magnetiskt strukturerad i *cylindrar*, *spår* och *sektorer*.

En sektor är typiskt (alltid?) 512 bytes lång medan antalet sektorer per spår, SPT, *Sectors Per Track*, ofta är 64. En sektor är den minsta enhet som kan avläsas från hårddisken, vill man bara läsa en byte måste man ändå läsa hela den sektor där byten är belägen.

För att lokalisera en sektor måste hårddisken alltså adresseras med tripletten CHS, *CylinderHeadSector*. När datorn ska adressera en hårddisk gör den det — historiskt — med 3 bytes d v s 24 bits. De första 10 bitarna anger cylindernummer (numrerade 0–1023), nästföljande 8 bitar pekar ut ett huvud (numrerade 0–255) och de sista 6 bitarna anger antal sektorer (numrerade 1–63)[3].

[3] Ja, de räknas från 1, inte 0.

Exempel

Hur många bytes kan adresseras med CHS enligt detta system?

$$1024 \cdot 256 \cdot 63 \cdot 256 = 8\ 455\ 716\ 864 \text{ bytes, d v s cirka 7.8 Gb.}$$

■

Det finns flera standarder för hur en disk ska uppföra sig. En vanlig är fortfarande den s k IDE-standarden.[4] Den är av senare datum än den ursprungliga CHS-specifikationen och tillåter $C/H/S = 65\ 536/16/256$.

Exempel

Hur stor disk kan adresseras med IDE-standarden?

$$65536 \cdot 16 \cdot 256 \cdot 512 = 128\ Gb.$$

■

Om man nu har en IDE-disk men adresserar den enligt enligt CHS, disken vet ju inte vilket, blir resultatet den minsta gemensamma nämnaren av dessa båda ovanstående, d v s

$$1024 \cdot 16 \cdot 63 \cdot 512 = 504\ Mb.$$

Vilket är anledningen till att många äldre datorer inte kan hantera moderna diskar, d v s de över 504 megabyte. Det är lite snopet, så man har infört ett antal knep för att komma förbi dessa begränsningar. I princip så ljuger man för disken: med hjälp av programvara går man förbi IDE-standarden ur datorns synvinkel och på så sätt få tillgång till stora diskar. Det är möjligt att disken inte har 8 huvuden men om programvaran kan få datorn att *tro det* så har man lyckats. En sådan *ommappning* kan göras så länge diskens storlek inte överskrids:

Exempel

En hårddisk sägs ha CHS=4092/16/63, vilket vi känner igen som en typisk IDE-drive. Utan ytterligare ommappning skulle denna disk bara kunna innehålla 504 Mb, men med stöd i datorn[5] kan efter ommappning någon av följande geometrier också användas:

[4]IDE står för *Integrated Device Electronics*, och var ett stort framsteg på sin tid. Tidigare hade praktiskt taget all styrelektronik befunnit sig utanför själva disken på ett *controllerkort*. I och med IDE mninatyriserades denna elektronik till att kunna rymmas på själva disken vilket gav en väsentligt ökad tillförlitlighet.

[5]Genom *drivrutiner* i mjukvara.

4092/16/63 2046/32/63 1023/64/63 511/128/63

Vi ser att ommappningen går till så att man dubblar antalet huvuden medan man halverar antalet spår. I den sista mappningen måste 511.5 rundas neråt till 511 och man tappar alltså lite utrymme på disken. 1023/64/63 är alltså den lägsta ommappning som kan väljas för CHS-adressering om man inte vill offra lagringskapacitet.

■

Tittar vi inuti disken kan vi se att den bara har 4 huvuden, inte 16 som den själv påstår, rimligtvis är alltså geometrin egentligen 16368/4/63. Man har filat på verkligheten. Varför? En anledning kan vara att man vill ha så mycket av disken som möjligt inom de första 1024 cylindrarna. En gammal begränsning medförde nämligen att en vanlig dator inte kunde starta från en cylinder större än 1024, en relikt från det ursprungliga CHS-systemet. Moderna datorer lider dock inte av denna begränsning.

11.3.1 Organisation på disken

På själva skivan ligger alltså datat lagrat i sektorer. Dessa numreras från skivans ytterkant och in mot mitten med sektor 0 som första sektor. Denna första sektor innehåller en s.k. MBR, *Master Boot Record* (egentligen placerad i sektor CHS=0/0/1), och är en, i övrigt vanlig, 512-bytes sektor.

MBR innehåller dels *partitionsinformation* och dels en programsnutt som letar reda på den *aktiva partitionen* och läser in den första sektorn, *startposten*, i denna. Startposten innehåller sedan ett program som drar igång resten av datorns operativsystem.

11.3.2 Partitioner

Partitioner är en ytterligare uppdelning av en hårddisk till en eller flera mindre enheter som var och en ser ut som en hårddisk de med. Man skiljer på *primära* och *utökade* partitioner. MBR innehåller plats för upp till 4 pekare till partitioner. Dessa kan antingen vara 1, 2, 3 eller 4 primära eller upp till 3 primära och en utökad partition. De primära inleds med en startpost medan de utökade inleds med en utökad partitionspost som

- dels pekar ut en startpost för den utökade partitionen,
- dels kan innehålla en framåtpekare till ytterligare en utökad partitionspost.

På så sätt kan man ha godtyckligt många utökade partitioner men högst 3 eller 4 primära partitioner.

Om partitionerna är primära eller utökade spelar roll när datorn vid start går igenom de anslutna diskarna och försöker tilldela enhetsbokstäver i DOS/Windows eller motsvarande i Unix. Den första primära partitionen kallas, i DOS/Windows, C:\, den andra D:\ o s v. När samtliga primära partitioner är tilldelade börjar systemet numrera upp de utökade som tar vid där den förra uppräkningen slutade, dvs E:\ här.

11.4 Filsystem

De data som ligger på hårddisken är organiserade i ett *filsystem*. Utan filsystemet skulle hårddisken bara innehålla en massa ”lösa” sektorer huller om buller. Det är uppenbart opraktiskt och mycket bättre om sektorerna kan struktureras i till exempel filer och kataloger. Det är filsystemet som ser till att dessa begrepp får mening. Det är också filsystemets konstruktion som avgör om ett data är lätt (=snabbt) att hitta eller om det är en omständlig procedur.

Målet för filsystemet är förstås att optimera skriv- och läshastighet till och från disken samt att utnyttja diskens lagringskapacitet maximalt. Till viss del är detta motstridiga krav och ett filsystem innehåller nödvändigtvis kompromisser. Vi kommer att se att en del av vägen till goda prestanda är att minimera intern och extern *fragmentering*.

Det finns många filsystem i användning och vi ska studera två stycken. Dels det s k FAT-filsystemet och dels det s k FFS-filsystemet. Det finns flera andra — HPFS för IBM:s operativsystem OS/2 och NTFS för Microsofts senare operativsystem för att ta några exempel — men då specifikationerna inte är offentliga är deras mekanismer inte helt kända. FAT och FFS är däremot väldokumenterade.

Vi kommer att studera filsystemens *allokeringspolicy*, med vilket menas den mekanism som används för att vid skrivning hitta en ”bra” plats för datat. En optimal allokeringspolicy kan placera ut datat utan några tidsödande beräkningar på ett sådant sätt att

både skrivning och senare läsning blir snabb. Även om hårddisken i sig är snabb, d v s man kan skriva till den i hög takt (tiotals megabyte per sekund), kommer den praktiska skrivhastigheten begränsas av hur snabbt allokeringspolicyn kan placera ut datat på disken (kanske bara några megabyte per sekund).

Hårddiskar tillverkas för övrigt inte speciellt för något filsystem utan är generella i den meningen att användaren kan använda det filsystem han önskar på dem.

11.4.1 FAT-filsystemet

Som representant för ett enkelt filsystem använder vi FAT-systemet (*File Allocation Table*). Det används på alla DOS-disketter och på många hårddiskar också.

I början av hårddisken ligger i FAT-fallet ett antal sektorer med information om resten av disken, detta är själva filallokeringstabellen.[6] Denna information talar om vilka sektorer som är lediga och vilka som används. Dessutom innehåller FAT information om vilka sektorer som hör ihop och tillsammans utgör filer. Dessutom kan FAT innehålla markeringar om vilka sektorer på hårddisken som inte kan användas för att de är felaktiga på nåt sätt.

FAT finns i flera varianter. En vanlig är FAT16 som tillåter 2^{16} sektorer, d v s sektorerna numreras från 0 till 65535. 65536 sektorer à 512 bytes ger en största hårddisk på 32 megabytes. En numera mycket liten hårddisk. För att kunna rymma mer infördes filsystemet FAT32 (med 2^{32} sektorer) och man kunde klumpa ihop flera sektorer till ett *kluster* och adresserade klustren istället. Typiska klusterstorlekar är 1, 2, 4, 8, 16 och 32 kb. En FAT16-hårddisk med klusterstorleken 32 kb kan då rymma $65536 \cdot 32 = 2048$ megabyte.

Man kommer alltid få fragmentering i ett filsystem som baserar sig på någon blockstorlek och att ha 32 kb stora kluster innebär att varje fil — hur liten den än är — lägger beslag på ett helt kluster. Ett slöseri med hårddiskyta. Och en tydlig nackdel med FAT-systemet.

[6] Det finns för övrigt två uppsättningar av filallokeringstabellen ”för säkerhets skull”, men jag har aldrig förstått varför man skulle vara säker på att andra FAT är bättre än första. Med två klockor vet man aldrig vad klockan är.

11.4.2 Rotkatalogen

Utöver filallokeringstabellen innehåller hårddisken även en *rotkatalog*. Denna innehåller information om de filer systemet innehåller, bland annat filnamnet. En post i rotkatalogen kan se ut som:

Där ett fält pekar ut första FAT-posten för filen. När man väl hittat början på filen kan

man använda information i FAT för att gå igenom filen till sista sektorn.

En post i rotkatalogen kan peka vidare till ytterligare katalogposter men rotkatalogens storlek är fix och sätter en gräns på hur många filer och/eller kataloger det kan ligga i ”roten”.

11.4.3 Allokeringspolicy FAT: ”Första bästa lediga plats”

Från början läggs alla filer ut konsekutivt, d v s man använder kluster efter kluster i filallokeringstabellen. Om man tar bort en fil kommer det alltså att bli luckor i filallokeringstabellen och om man lägger till en fil som är större än en sådan lucka kommer filen först att ta ”första bästa” FAT-lucka i anspråk och sedan hoppa bock över de använda FAT-posterna och nyttja första tomma post. Efterson filallokeringstabellen är en avbildning av klustren på skivan resulterar denna policy efter en tid i en kraftig *fragmentering* av filerna. Filerna kommer att ligga utspridda i små bitar — i storleken av ett kluster — på i stort sett hela skivan s k *extern fragmentering*. För att läsa en enda fil kan det alltså behövas hämtas information över praktiskt taget hela diskytan. Varje sökning innebär därmed sannolikt flera cylinderbyten och det är som nämnts en *mycket* långsam operation.[7]

För att undvika denna oreda på partitionen används ofta defragmenteringsprogram som sorterar upp filerna på disken och lägger dom ”i ordning” utan extern fragmentering. Ett resultat av defragmenteringen är att inläsningen av en fil går snabbare, även om det tar lika lång tid att hitta var filen börjar. Även skrivningen till partitionen går snabbare då det sannolikt går att placera nya filer utan fragmentering på den frigjorda ytan.

En annan typ av fragmentering uppstår också *inom* en sektor (eller kluster) om den inte kan nyttjas fullt ut, men detta är möjligen ett mindre problem. Det innebär att *en* 512-bytes multipel av en datafil kanske inte kommer utnyttjas helt, ett slöseri med i snitt 256 bytes oavsett filens längd, s k *intern fragmentering*. Med intern fragmentering inom ett kluster blir förlusterna naturligtvis större med ökande klusterstorlek.

Problemen med fragmentering i FAT-filsystemet och den ökade söktid, som är en direkt följd av detta, gör att FAT-filsystemet är tämligen långsamt. Den enkelhet som ligger bakom konstruktionen och den praktiska avsaknaden av en intelligent allokeringsalgoritm gör systemet enkelt att sjösätta, men dess prestanda är inte höga. Dessa brister blir större ju större datamängder som skall manipuleras.

[7] Relativt processorklockan i alla fall. Tiden för cylinderbyte är i storleksordningen millisekunder medan processorklockan är i storleksordningen mikrosekunder eller mindre. Under den tid det tar att positionera läshuvudet på hårddisken kan åtskilliga miljontals assemblerinstruktioner utföras.

Även om detta kan ses som förödande kritik av filsystemet, måste man hålla i tankarna att när det konstruerades fanns fortfarande inte hårddiskar med en storlek av ens 32 Mb. Det är ett utmärkt filsystem för mindre datamängder — vi har nämnt disketter — men det borde kanske inte ha överlevt 32 Mb gränsen? Definitivt inte om man tycker att ett filsystem skall vara effektivt.

11.5 FFS Unix Fast Filesystem — ett effektivt filsystem

För större datamängder måste mer avancerade metoder användas för att placera ut datat så att skrivning och läsning blir effektiva. Operativsystemet Unix har använts i stordatorsammanhang sedan 70-talet och man har där varit tvungen att optimera prestanda. En väsentlig del av datorsystemets totala prestanda hänger samman med dess filsystem, speciellt i fleranvändarsystem. Nuförtiden kan en webbserver tjäna som exempel på ett fleranvändarsystem med ”mycket igång samtidigt”. Tidigare kunde ett fleranvändarsystem (med total processorkraft lägre än tutorkortets!) hantera kanske hundratalet samtidiga terminalanvändare. Oavsett typ av belastning har sådana system många filer i omlopp och filsystemet utmärker sig som en flaskhals. Ett mål med Unix’ filsystem har varit att eliminera denna flaskhals.

11.5.1 Generellt upplägg FFS

Istället för filallokeringstabell och kluster förlitar sig Unix’ filsystem på *inoder* enligt följande mönster, där en inode typiskt (men inte alltid) motsvarar en fil:

fig 7.1 4.4BSD

Inoden innehåller, bland annat, information om filens:

- mode (läs/skriv-rättigheter bland annat)
- ägare
- tidmarkering för senaste läsning och skrivning
- storlek i byte
- antal referenser till filen (åtminstone 1 om filen existerar, 0 om filen är raderad)
- ett antal pekare till block där data lagrats. Pekarna kan vara direkta (pekar direkt på block), eller enkelt, dubbelt eller t o m trippelt indirekta (pekar på ytterligare inoder).

Data lagras i *block*. *Blockstorleken* bestäms när man skapar filsystemet och sedan inte ändras. En typisk blockstorlek är 4 eller 8 kilobyte. Blocken motsvarar ungefär kluster i FAT-filsystemet, men med det tillägget att ett block kan delas upp i upp till 8 *fragment* om så skulle behövas. Något som inte kunde göras med klustren i FAT-fallet.

Med block och fragment kan man få ett optimalt nyttjande av disken. Om blockstorleken väljs till 4 kb kan varje sektor pekas ut och med 8 kb block kan varje kilobyte pekas ut.

För att hålla ordning på alla block inom en eller flera cylindrar (en *cylindergrupp*) finns en *block map* som indikerar vilka block som används och inte används. I blockmappen används en bit per block för att indikera detta.

På en högre nivå i filsystemet finns även ett *superblock* som innehåller övergripande information, exempelvis när senaste skrivning till filsystemet skedde, antal cylindergrupper och block i filsystemet, storleken på blocken o s v.

Notera att vi hittills inte sett vad filen egentligen heter. Filnamnet lagras separat i särskilda inoder och pekare från dessa inoder pekar ut filens början. Detta ger dessutom en frihet att låta samma data refereras från olika håll! Unix använder detta för s k mjuka och hårda länkar (*soft links*, *hard links*).

(bild på hur filnamn pekar ut en och flera inoder fig 7.7 och 7.8 4.4BSD)

11.5.2 Allokeringspolicy FFS

Det som gör detta filsystem så effektivt är en aggressiv allokeringspolicy. Systemet lägger ner stor möda på att placera ut data på hårddisken på ett smart sätt för att på så sätt erhålla maximala prestanda vid både läsning och skrivning. Detaljerna kan läsas i dokumentationen på sidan 198.

Uppenbart är det effektivt att lägga ut en fil i närliggande områden på disken och FFS försöker således placera en fil inom en cylindergrupp om möjligt. Om det inte är möjligt — filen är för stor — måste flera cylindergrupper tas i anspråk. Till en början väljer dock man cylindergrupper som helt kan rymma filen och först i absolut sista hand genomsöker man alla cylindergrupper på jakt efter en lämplig plats/platser för filen.

Man får data utspritt över disken på detta sätt och den erhållna glesheten gör att det i framtiden är snabbare/lättare att hitta en plats för en ny fil. Observera att det inte är någon prestandanedsättning att filerna ligger spridda över diskytan så länge filerna *i sig* hålls samman. Extern fragmentisering försämrar alltså inte prestanda i detta fall.

För att ge systemet frihet att välja ”bra” platser på disken används aldrig diskens kapacitet helt och hållet. 8 % av den totala diskarean[8] reserveras för att ge allokeringsalgoritmen

[8] Detta är standardinställningen i FreeBSD. Andra operativsystem kan reservera mer eller mindre än detta, upp till 25 procent! Moderna filsystem behöver dock alltid denna typ av extra frihet för att vara effektiva.

denna frihet. Systemanvändaren går förbi denna behränsning så när en ”vanlig” användare ser att disken är 100 % full kan systemanvändaren (root på unix) fortfarande nyttja de återstående 8 %-en. Prestanda avtar väsentligt om denna area görs mindre eller tas bort helt. Den exakta mängden kan bestämmas vid formateringen av disken. Normala värden på fragmenteringen i ett FFS-filsystem är mindre än två procent, d v s endast två procent av systemets filer ligger i oordning. Detta gäller även för tungt lastade diskar efter åratal av användning.

11.5.3 Optimering av prestanda

För att konstruera ett effektivt filsystem är det viktigt att ha någon uppfattning om storleken på de filer som systemet ska hantera. Även sedan upplägget av filsystemet i sig är konstruerat, är det bra om man kan välja blockstorlek med hänsyn till den väntade filstorleken, för att kunna nå våra mål d v s:

- optimera skriv- och läsning med avseende på överföringshastighet
- minimera (interna vad gäller blockstorleken) fragmenteringen på hårddisken.

Exempel

För webbservern `vanheden` ser filstorleken, viktat med antalet läsningar av respektive fil ut som följer:

Filstorlek	Antal åtkomster	%bytes
0	41660	
1 b – 10 b	50	
11 b – 100 b	598	
101 b – 1 kb	18153	0.06 %
1 kb – 10 kb	33001	1.03 %
10 kb – 100 kb	87056	19.51 %
100 kb – 1 Mb	18357	38.53 %
1 Mb – 10 Mb	1688	30.31 %
10 Mb – 100 Mb	71	10.55 %

■

En blockstorlek på 8 eller 16 kb verkar enligt tabellen vettig för detta filsystems blandning av filstorlekar. Anledningen till att blockstorleken ändå är 4 kb — för det är den — är att det fungerar (lathet?), samt att hela filsystemet innehåller andra filer än de som webservern använder.

11.6 Journalförande filsystem — Om strömmen går...

Vad händer om strömmen går eller datorn låser sig?[9]

[9] Om datorn låser sig och det inte beror på hårdvarufel rekommenderas förstås att byta till ett bättre operativsystem:)

För att kunna analysera detta måste vi införa begreppet *metadata* till skillnad från vanligt data. Med metadata menas all den ytterligare information som lagras om en fil förutom själva filinnehållet, d v s filnamn, ägare och så vidare. Metadatat innehåller *pekare* till var motsvarande data ligger.

En normal skrivning till en hårddisk skriver in både data och tillhörande metadata. Problem uppstår om dessa inte båda lyckas skrivas till disken. Man kan tänka sig två fall beroende på i vilken ordning data och metadata skrivs:

1. Man hann skriva in metadata men inget mer d v s det ser ut som om filen finns men pekarna pekar inte på något eller möjligen på helt fel data.
2. Man hann skriva in data men inte metadata, d v s filinnehållet finns på disken men kan inte nås eftersom inget metadata finns som kan peka ut detta.

Man kan lätt inse att det är olyckligt att ha lösa pekare på hårddisken. Filsystemsreparationsprogram kan ofta reparera diaken till ett konsistent tillstånd utan lösa pekare o s v, men med eventuell dataförlust som resultat.

En metod att komma åt detta problem är att använda journalförande filsystem. Det vill säga att föra en journal — på hårddisken — om vad som man tänkt göra i nästa moment och markera om man sedan lyckats göra detta. Om man lyckats med operationen raderas journalanteckningen, men om strömmen plötsligt gick under pågående operation, kan man i journalen se att man börjat skriva på ett visst ställe men inte slutfört skrivningen. Med speciella program kan man då manuellt eller automatiskt antingen

- ta bort den påbörjade operationen och radera i loggen, eller kanske till och med
- fullfölja den avbrutna operationen och radera journalanteckningen.

Oavsett vilken väg man gick har man nu plötsligt ett konsistent filsystem utan lösa pekare. Det kostar naturligtvis lite extra tid och omtanke att skriva till ett journalförande filsystem men den vinst man får vid ett eventuellt strömavbrott e dyl — att hårddisken är direkt användbar nästan omedelbart vid nästa uppstart — överväger ofta eventuella nackdelar.

11.7 Hårddisk-konstellationer

För inte så länge sedan var hårddiskarnas storlek och/eller snabbhet alltför begränsad för att duga i många sammanhang. Man kan tänka sig större databaser, eller lagring av ljud- och videodata som exempel på situationer där en enda hårddisks kapacitet inte räcker till. Vi ska nu titta på olika sätt att kombinera flera hårddiskar för att förbättra lagringssystemets kapacitet med avseende på lagringsvolym, feltolerans och skriv- och läsprestanda.

11.7.1 Konkatenerade diskar

Det första sättet att åstadkomma större lagringsvolym är *konkatenering* av diskarna (*concatenation*). Fördelen med denna metod är att man kan använda diskar av olika storlek eftersom de fylls på efterhand:

Om läsning/skrivning sker jämnt fördelat mellan diskarna kommer bandbredden till/från diskarna att vara större än för en enskild disk. Om all trafik går mot samma disk kommer dock denna disks prestanda vara avgörande för systemets prestanda som helhet.

11.7.2 RAID — Redundant Array of Inexpensive Disks

Under detta begrepp[10] döljer sig ett antal kombinationer av hårddiskar. Metoden är som förut att kombinera de olika diskarna så att prestande förbättras i ett eller annat avseende. Man kan tänka sig situationer där det viktigaste är stor lagringsvolym, andra där tillgängligheten är viktigast o s v. Skriv- och läsprestanda får inte heller vara för begränsande. Samtliga RAID-konstellationer som presenteras här kräver diskar av samma storlek eller i varje fall partitioner av samma storlek.

RAID-0 (striping)

RAID-0 (även kallad *striping*) ger stor lagringsvolym precis som i det konkatenerade exemplet ovan men data lagras annorlunda för att tvinga belastningen att fördela sig jämnt över alla diskarna:

[10] Betyder också Redundant Array of Independent Disks

Med RAID-0 blir det lite jobbigare att lokalisera och manipulera data — man måste ju räkna fram i vilken disk informationen ligger. För att optimera läs- och skrivhastigheten är det viktigt att inte lägga all information på en av diskarna, den kommer då att belastas hårt och utgöra en prestandabegränsning i systemet. Om informationen däremot är strategiskt organiserad över de ingående diskarna kan man förvänta sig en jämn arbetsbelastning och undviker därmed att enskilda diskars prestanda blir tydliga flaskhalsar.

RAID-1 (spegling, mirroring)

Hårddiskar kraschar och dör förr eller senare. Tillförlitligheten har under åren hela tiden ökat men fortfarande är hårddisken den datorkomponent som sannolikt dör först. Och allt efter att diskarna blir större blir förlusten av en hårddisk förstås alltmer kännbar. För att förbättra oddsen kan man tillgripa RAID-1, *spegling* av en disk. Det innebär att man har flera kopior av samma data i systemet. I det enklaste fallet består det av två diskar, som alltså innehåller identiska data:

Med RAID-1 måste varje skrivning ske till båda diskarna. Det blir alltså långsammare att skriva till systemet. Å andra sidan kan man läsa från endera disken. Läsning borde alltså bli något snabbare: om ena disken är upptagen kan man fortfarande läsa alla data ur den andra. I specialfallet att den ena disken slutar fungera kommer användaren inte märka något, systemet fungerar som vanligt utåt. Denna inbyggda redundans är den främsta anledningen till att använda RAID-1.

Uppenbarligen går det åt dubbelt så stort diskutrymme än vad som kan lagras men det kompenseras av den ökade tillgängligheten och pålitligheten.

RAID-5

En annan möjlighet att förbättra tillförlitligheten är att tillgripa paritet. RAID-2, -3, -4 och -5 med flera är varianter på detta tema. Här skall bara nämnas RAID-5. Det handlar hela tiden om att möjliggöra felrättning/felupptäckt med hjälp av paritetsinformation[11] som ligger på en annan disk. RAID-5 innehåller fördelarna med RAID-1 men har även tillräckligt mycket paritetsinformation för att helt kunna *återskapa* en havererad disk:

[11] Med udda paritet menas att antalet ettor i en viss bitposition på alla diskar ska vara udda. Om den plötsligt inte är det vet vi att fel inträffat och med kännedom om två bitar kan vi dessutom också räkna ut den tredje.

Om `disk1` dör kan dess information återskapas med ledning av `disk2`, `disk3` och paritetsinformationen på `disk4` och så vidare. Om `disk4` dör kan paritetsinformationen inte läsas men väl återskapas om någon skulle vilja det. Läsning från en fungerande disk sker som vanligt.

Systemet är mycket tolerant mot hårdvarufel hos någon ingående hårddisk. Den totala lagringsvolymen och datat förblir intakt om en disk skulle stanna. Om däremot fler än en disk skulle haverera rasar hela konfigurationen.

RAID-5 kräver minst 3 diskar, två för datat och en för paritetsinformation. I verkliga system brukar fem diskar användas. Dessutom vill man fördela arbetsbelastningen jämnt på diskarna varför paritetsinformationen också återfinns fördelad på samtliga ingående diskar.

Skrivning blir långsammare eftersom paritetsinformationen måste beräknas och införas på respektive disk. Om en disk stannar kan man å andra sidan, i vissa system, helt enkelt dra ur den felaktiga disken och sätta in en ny — under drift! När systemet lägger märke till den tomma disken uppstår en febril aktivitet för att fylla den med paritetsinformation och återskapa dataareorna.

11.7.3 Teknisk dokumentation

En mer detaljerad inblick i filsystem kan erhållas ur följande tekniska dokumentation.[12] Först beskrivs FFS, *Fast File System*, det optimerade filsystem som härstammar ur UFS, *the Unix File System*. Sedan beskrivs det program som kan reparera ett filsystem som inte längre är konsistent. I den senare texten är dess appendix borttaget då det beskriver de felkoder programmet genererar och som inte är av intresse för oss här. Tillsammans ger de en god inblick i flera av de överväganden som ligger bakom ett modernt filsystem.

Texterna är hämtade ur systemdokumentationen till operativsystemet FreeBSD och omfattas av den s k BSD-licensen:

```
All of the documentation and software included in the 4.4BSD and 4.4BSD-Lite
Releases is copyrighted by The Regents of the University of California.

Copyright 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
        The Regents of the University of California.  All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
```

[12] Närmare bestämt ur *System Manager's Manual, SMM, för FreeBSD.*

A Fast File System for UNIX*

Marshall Kirk McKusick, William N. Joy†,
Samuel J. Leffler‡, Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

A reimplementation of the UNIX file system is described. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies that allow better locality of reference and can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access to large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the programmers' interface are discussed. These include a mechanism to place advisory locks on files, extensions of the name space across file systems, the ability to use long file names, and provisions for administrative control of resource usage.

Revised February 18, 1984

CR Categories and Subject Descriptors: D.4.3 **[Operating Systems]**: File Systems Management – *file organization, directory structures, access methods*; D.4.2 **[Operating Systems]**: Storage Management – *allocation/deallocation strategies, secondary storage devices*; D.4.8 **[Operating Systems]**: Performance – *measurements, operational analysis*; H.3.2 **[Information Systems]**: Information Storage – *file organization*

Additional Keywords and Phrases: UNIX, file system organization, file system performance, file system design, application program interface.

General Terms: file system, measurement, performance.

---

* UNIX is a trademark of Bell Laboratories.

† William N. Joy is currently employed by: Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043

‡ Samuel J. Leffler is currently employed by: Lucasfilm Ltd., PO Box 2009, San Rafael, CA 94912

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

TABLE OF CONTENTS

1. Introduction

This paper describes the changes from the original 512 byte UNIX file system to the new one released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to effect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the facilities that are available to programmers.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. Virtually no constraints other than available disk space are placed on file growth [Ritchie74], [Thompson78].*

When used on the VAX-11 together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications such as VLSI design and image processing do a small amount of processing on a large quantities of data and need to have a high throughput from the file system. High throughput rates are also needed by programs that map files from the file system into large virtual address spaces. Paging data in and out of the file system is likely to occur frequently [Ferrin82b]. This requires a file system providing higher bandwidth than the original 512 byte UNIX one that provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently, users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. Previous work to improve the UNIX file system performance has been done by [Ferrin82a]. The UNIX operating system drew many of its ideas from Multics, a large, high performance

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

* In practice, a file's size is constrained to be less than about one gigabyte.

operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a LISP environment [Symbolics81]. A good introduction to the physical latencies of disks is described in [Pechura83].

2. Old File System

In the file system developed at Bell Laboratories (the "traditional" file system), each disk drive is divided into one or more partitions. Each of these disk partitions may contain one file system. A file system never spans multiple partitions.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to the *free list*, a linked list of all the free blocks in the file system.

Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. An inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in an inode itself*. An inode may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further singly indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A 150 megabyte traditional UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from the file's inode to its data. Files in a single directory are not typically allocated consecutive slots in the 4 megabytes of inodes, causing many non-consecutive blocks of inodes to be accessed when executing operations on the inodes of several files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by staging modifications to critical file system information so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors: each disk transfer accessed twice as much data, and most files could be described without need to access indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system*.

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random, causing files to have their blocks allocated randomly over the disk. This forced a seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of this randomization of data block placement. There was no way of restoring the performance of an old file system except to dump, rebuild, and restore the file system. Another possibility, as suggested by [Maruyama76], would be to have a process that periodically

† By "partition" here we refer to the subdivision of physical space on a disk drive. In the traditional file system, as in the new file system, file systems are really located in logical disk partitions that may overlap. This overlapping is made available, for example, to allow programs to copy entire disk drives containing multiple file systems.

* The actual number may vary from system to system, but is usually in the range 5-13.

reorganized the data on the disk to restore locality.

3. New file system organization

In the new file system organization (as in the old file system organization), each disk drive contains one or more file systems. A file system is described by its super-block, located at the beginning of the file system's disk partition. Because the super-block contains critical data, it is replicated to protect against catastrophic loss. This is done when the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To insure that it is possible to create files as large as 2^{32} bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of a file system is recorded in the file system's super-block so it is possible for file systems with different block sizes to be simultaneously accessible on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization divides a disk partition into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. The bit map of available blocks in the cylinder group replaces the traditional file system's free list. For each cylinder group a static number of inodes is allocated at file system creation time. The default policy is to allocate one inode for each 2048 bytes of space in the cylinder group, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all redundant copies of the super-block. Thus the cylinder group bookkeeping information begins at a varying offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group than the preceding cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-block. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transaction, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before requiring a seek.

The main problem with larger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system block size on the amount of wasted space in the file system. The files measured to obtain these figures reside on one of our

† While it appears that the first cylinder group could be laid out with its super-block at the ''known'' location, this would not work for file systems with blocks sizes of 16 kilobytes or greater. This is because of a requirement that the first 8 kilobytes of the disk be reserved for a bootstrap program and a separate requirement that the cylinder group information begin on a file system block boundary. To start the cylinder group on a file system block boundary, file systems with block sizes larger than 8 kilobytes would have to leave an empty space between the end of the boot block and the beginning of the cylinder group. Without knowing the size of the file system blocks, the system would not know what roundup function to use to find the beginning of the first cylinder group.

time sharing systems that has roughly 1.2 gigabytes of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formatted space.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	Data + inodes, 512 byte block UNIX file system
866.5 Mb	11.8	Data + inodes, 1024 byte block UNIX file system
948.5 Mb	22.4	Data + inodes, 2048 byte block UNIX file system
1128.3 Mb	45.6	Data + inodes, 4096 byte block UNIX file system

Table 1 – Amount of wasted space as a function of block size.

The space wasted is calculated to be the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified at the time that the file system is created; each file system block can optionally be broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space available in a cylinder group at the fragment level; to determine if a block is available, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1 – Example layout of blocks and fragments in a 4096/1024 file system.

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0–5, 10, and 11 are in use, while fragments 6–9, and 12–15 are free. Fragments of adjoining blocks cannot be used as a full block, even if they are large enough. In this example, fragments 6–9 cannot be allocated as a full block; only fragments 12–15 can be coalesced into a full block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remaining fragments of the block are made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would uses two full size blocks and one three fragment portion of another block. If no block with three aligned fragments is available at the time the file is created, a full size block is split yielding the necessary fragments and a single unused fragment. This remaining fragment can be allocated to another file as needed.

Space is allocated to a file when a program does a *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased*. If the file needs to be expanded to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block or fragment to hold the new data. The new data is written into the available space.

2) The file contains no fragmented blocks (and the last block in the file contains insufficient space to hold the new data). If space exists in a block already allocated, the space is filled with new data. If the remainder of the new data contains more than a full block of data, a full block is allocated and the

* A program may be overwriting data in the middle of an existing file in which case space would already have been allocated.

first full block of new data is written there. This process is repeated until less than a full block of new data remains. If the remaining new data to be written will fit in less than a full block, a block with the necessary fragments is located, otherwise a full block is located. The remaining new data is written into the located space.

3) The file contains one or more fragments (and the fragments contain insufficient space to hold the new data). If the size of the new data plus the size of the data already in the fragments exceeds the size of a full block, a new block is allocated. The contents of the fragments are copied to the beginning of the block and the remainder of the block is filled with new data. The process then continues as in (2) above. Otherwise, if the new data to be written will fit in less than a full block, a block with the necessary fragments is located, otherwise a full block is located. The contents of the existing fragments appended with the new data are written into the allocated space.

The problem with expanding a file one fragment at a a time is that data may be copied many times as a fragmented block expands to a full block. Fragment reallocation can be minimized if the user program writes a full block at a time, except for a partial block at the end of the file. Since file systems with different block sizes may reside on the same system, the file system interface has been extended to provide application programs the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The amount of wasted space in the 4096/1024 byte new file system organization is empirically observed to be about the same as in the 1024 byte old file system organization. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of wasted space as the 512 byte block UNIX file system. The new file system uses less space than the 512 byte or 1024 byte file systems for indexing information for large files and the same amount of space for small files. These savings are offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when a new file system's fragment size equals an old file system's block size.

In order for the layout policies to be effective, a file system cannot be kept completely full. For each file system there is a parameter, termed the free space reserve, that gives the minimum acceptable percentage of file system blocks that should be free. If the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter may be changed at any time, even when the file system is mounted and active. The transfer rates that appear in section 4 were measured on file systems kept less than 90% full (a reserve of 10%). If the number of free blocks falls to zero, the file system throughput tends to be cut in half, because of the inability of the file system to localize blocks in a file. If a file system's performance degrades because of overfilling, it may be restored by removing files until the amount of free space once again reaches the minimum acceptable level. Access rates for files created during periods of little free space may be restored by moving their data once enough space is available. The free space reserve must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, the percentage of waste in an old 1024 byte UNIX file system is roughly comparable to a new 4096/512 byte file system with the free space reserve set at 5%. (Compare 11.8% wasted with the old file system to 6.9% waste + 5% reserved space in the new file system.)

3.2. File system parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration-dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is parameterized so that it can be adapted to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be rotationally well

positioned. The distance between ‘‘rotationally optimal’’ blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with an input/output channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks can often be accessed without suffering lost time because of an intervening disk revolution. For processors without input/output channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to service an interrupt and schedule a new disk transfer. Given a block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in the file will come into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the available blocks in a cylinder group at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive. The super-block contains a vector of lists called *rotational layout tables*. The vector is indexed by rotational position. Each component of the vector lists the index into the block map for every data block contained in its rotational position. When looking for an allocatable block, the system first looks through the summary counts for a rotational position with a non-zero block count. It then uses the index of the rotational position to find the appropriate list to use to index through only the relevant parts of the block map to find a free block.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with a rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

The file system layout policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Inodes of files in the same directory are frequently accessed together. For example, the ‘‘list directory’’ command

often accesses the inode for each file in a directory. The layout policy tries to place all the inodes of files in a directory in the same cylinder group. To ensure that files are distributed throughout the disk, a different policy is used for directory allocation. A new directory is placed in a cylinder group that has a greater than average number of free inodes, and the smallest number of directories already in it. The intent of this policy is to allow the inode clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for a particular cylinder group can be read with 8 to 16 disk transfers. (At most 16 disk transfers are required because a cylinder group may have no more than 2048 inodes.) This puts a small and constant upper bound on the number of disk transfers required to access the inodes for all the files in a directory. In contrast, the old file system typically requires one disk transfer to fetch the inode for each file in a directory.

The other major resource is data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all data blocks for a file in the same cylinder group, preferably at rotationally optimal positions in the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Further, using all the space in a cylinder group causes future allocations for any file in the cylinder group to also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The heuristic solution chosen is to redirect block allocation to a different cylinder group when a file exceeds 48 kilobytes, and at every megabyte thereafter.* The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free, otherwise it allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristics that employ only partial information.

If a requested block is not available, the local allocator uses a four level allocation strategy:

1) Use the next available block rotationally closest to the requested block on the same cylinder. It is assumed here that head switching time is zero. On disk controllers where this is not the case, it may be possible to incorporate the time required to switch between disk platters when constructing the rotational layout tables. This, however, has not yet been tried.

2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.

3) If that cylinder group is entirely full, quadratically hash the cylinder group number to choose another cylinder group to look for a free block.

4) Finally if the hash fails, apply an exhaustive search to all cylinder groups.

Quadratic hash is used because of its speed in finding unused slots in nearly full hash tables [Knuth75]. File systems that are parameterized to maintain at least 10% free space rarely use this strategy. File systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random; the most important characteristic of the strategy used under such conditions is that the strategy be fast.

* The first spill over point at 48 kilobytes is the point at which a file on a 4096 byte block file system first requires a single indirect block. This appears to be a natural first point at which to redirect block allocation. The other spillover points are chosen with the intent of forcing block allocation to be redirected when a file has used about 25% of the data blocks in a cylinder group. In observing the new file system in day to day use, the heuristics appear to work well in minimizing the number of completely filled cylinder groups.

4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empirical studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories (to force the system to access inodes in multiple cylinder groups), the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate at which user programs can transfer data to or from a file without performing any processing on it. These programs must read and write enough data to insure that buffering in the operating system does not affect the results. They are also run at least three times in succession; the first to get the system into a known state and the second two to insure that the experiment has stabilized and is repeatable. The tests used and their results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the CPU or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an AMPEX Capricorn 330 megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured. The same number of system calls were performed in all tests; the basic system call overhead was a negligible portion of the total running time of the tests.

Type of File System	Processor and Bus Measured	Read Speed	Read Bandwidth	Read % CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/983 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/983 22%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/983 24%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/983 47%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/983 47%	54%

Table 2a – Reading rates of the old and new UNIX file systems.

Type of File System	Processor and Bus Measured	Write Speed	Write Bandwidth	Write % CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/983 5%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/983 14%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/983 22%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/983 33%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/983 47%	95%

Table 2b – Writing rates of the old and new UNIX file systems.

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system with a 10% free space reserve. Synthetic work loads suggest that throughput deteriorates to about half the rates given in Table 2 when the file systems are full.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is calculated by multiplying the number of bytes on a track by the number of revolutions of the disk per second. The bandwidth is calculated by comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3–5% of the disk bandwidth, while the new file system uses up to 47% of the bandwidth.

† A UNIX command that is similar to the reading test that we used is "cp file /dev/null", where "file" is eight megabytes long.

Both reads and writes are faster in the new system than in the old system. The biggest factor in this speedup is because of the larger block size used by the new file system. The overhead of allocating blocks in the new system is greater than the overhead of allocating blocks in the old system, however fewer blocks need to be allocated in the new system because they are bigger. The net effect is that the cost per byte allocated is about the same for both systems.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, making the processor unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the write system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers queue up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek distance, the average seek between the scheduled disk writes is much less than it would be if the data blocks were written out in the random disk order in which they are generated. However when the file is read, the read system call is processed synchronously so the disk blocks must be retrieved from the disk in the non-optimal seek order in which they are requested. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

In the new system the blocks of a file are more optimally ordered on the disk. Even though reads are still synchronous, the requests are presented to the disk in a much better order. Even though the writes are still asynchronous, they are already presented to the disk in minimum seek order so there is no gain to be had by reordering them. Hence the disk seek latencies that limited the old file system have little effect in the new file system. The cost of allocation is the factor in the new system that causes writes to be slower than reads.

The performance of the new file system is currently limited by memory to memory copy operations required to move data from disk buffers in the system's address space to data buffers in the user's address space. These copy operations account for about 40% of the time spent performing an input/output operation. If the buffers in both address spaces were properly aligned, this transfer could be performed without copying by using the VAX virtual memory management hardware. This would be especially desirable when transferring large amounts of data. We did not implement this because it would change the user interface to the file system in two major ways: user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow contiguous disk blocks to be read in a single disk transaction. Many disks used with UNIX systems contain either 32 or 48 512 byte sectors per track. Each track holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than 50% of the available bandwidth. If the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up allocations, the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79]. This technique was not included because block allocation currently accounts for less than 10% of the time spent in a write system call and, once again, the current throughput rates are already limited by the speed of the available processors.

5. File system functional enhancements

The performance enhancements to the UNIX file system did not require any changes to the semantics or data structures visible to application programs. However, several changes had been generally desired for some time but had not been introduced because they would require users to dump and restore all their file systems. Since the new file system already required all existing file systems to be dumped and restored, these functional enhancements were introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. Only programs that read directories are affected by this change. To promote portability to UNIX systems that are not running the new file system, a set of directory access routines have been introduced to provide a consistent interface to directories on both old and new systems.

Directories are allocated in 512 byte units called chunks. This size is chosen so that each allocation can be transferred to disk in a single operation. Chunks are broken up into variable length records termed directory entries. A directory entry contains the information necessary to map the name of a file to its associated inode. No directory entry is allowed to span multiple chunks. The first three fields of a directory entry are fixed length and contain: an inode number, the size of the entry, and the length of the file name contained in the entry. The remainder of an entry is variable length and contains a null terminated file name, padded to a 4 byte boundary. The maximum length of a file name in a directory is currently 255 characters.

Available space in a directory is recorded by having one or more entries accumulate the free space in their entry size fields. This results in directory entries that are larger than required to hold the entry name plus fixed length fields. Space allocated to a directory should always be completely accounted for by totaling up the sizes of its entries. When an entry is deleted from a directory, its space is returned to a previous entry in the same directory chunk by increasing the size of the previous entry by the size of the deleted entry. If the first entry of a directory chunk is free, then the entry's inode number is set to zero to indicate that it is unallocated.

5.2. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to use a separate ''lock'' file. A process would try to create a ''lock'' file. If the creation succeeded, then the process could proceed with its update; if the creation failed, then the process would wait and try again. This mechanism had three drawbacks. Processes consumed CPU time by looping over attempts to create locks. Locks left lying around because of system crashes had to be manually removed (normally in a system startup command script). Finally, processes running as system administrator are always permitted to create files, so were forced to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight forward, so a mechanism for locking files has been added.

The most general schemes allow multiple processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to serialize access to a file with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the standard system applications, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the extent of enforcement. A hard lock is always enforced when a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel. With advisory locks the policy is left to the user programs. In the UNIX system, programs with system administrator privilege are allowed override any protection scheme. Because many of the programs that need to use locks must also run as the system administrator, we chose to implement advisory locks rather than create an additional protection scheme that was inconsistent with the UNIX philosophy or could not be used by system

administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process may have an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the lock request will block until the lock can be obtained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process may access the file.

Locks are applied or removed only on open files. This means that locks can be manipulated without needing to close and reopen a file. This is useful, for example, when a process wishes to apply a shared lock, read some information and determine whether an update is required, then apply an exclusive lock and update the file.

A request for a lock will cause a process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to conditionally request a lock is useful to ''daemon'' processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since locks exist only while the locking processes exist, lock files can never be left active after the processes exit or if the system crashes.

Almost no deadlock detection is attempted. The only deadlock detection done by the system is that the file to which a lock is applied must not already have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail).

5.3. Symbolic links

The traditional UNIX file system allows multiple directory entries in the same file system to reference a single file. Each directory entry ''links'' a file's name to an inode and its contents. The link concept is fundamental; inodes do not reside in directories, but exist separately and are referenced by links. When all the links to an inode are removed, the inode is deallocated. This style of referencing an inode does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* similar to the scheme used by Multics [Feiertag71] have been added.

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. In UNIX, pathnames are specified relative to the root of the file system hierarchy, or relative to a process's current working directory. Pathnames specified relative to the root are called absolute pathnames. Pathnames specified relative to the current working directory are termed relative pathnames. If a symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links; seven system utilities required changes to use these calls.

In future Berkeley software distributions it may be possible to reference file systems located on remote machines using pathnames. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create a new version of an existing file typically create the new version as a temporary file and then rename the temporary file with the name of the target file. In the old UNIX file system renaming required three calls to the system. If a program were interrupted or the system crashed between these calls, the target file could be left with only its temporary name. To eliminate this possibility the *rename* system call has been added. The rename call does the rename operation in a fashion that guarantees the

existence of the target name.

Rename works both on data files and directories. When renaming directories, the system must do special validation checks to insure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the descendents of the target directory to insure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of inodes and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Resources are given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more resources while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If users fails to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

Acknowledgements

We thank Robert Elz for his ongoing interest in the new file system, and for adding disk quotas in a rational and efficient manner. We also acknowledge Dennis Ritchie for his suggestions on the appropriate modifications to the user interface. We appreciate Michael Powell's explanations on how the DEMOS file system worked; many of his ideas were used in this implementation. Special commendation goes to Peter Kessler and Robert Henry for acting like real users during the early debugging stage when file systems were less stable than they should have been. The criticisms and suggestions by the reviews contributed significantly to the coherence of the paper. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

References

[Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41

[Ferrin82a] Ferrin, T.E., "Performance and Robustness Improvements in Version 7 UNIX", Computer Graphics Laboratory Technical Report 2, School of Pharmacy, University of California, San Francisco, January 1982. Presented at the 1982 Winter Usenix Conference, Santa Monica, California.

[Ferrin82b] Ferrin, T.E., "Performance Issuses of VMUNIX Revisited", ;login: (The Usenix Association Newsletter), Vol 7, #5, November 1982. pp 3-6

[Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group,

Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Kowalski78] Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978

[Knuth75] Kunth, D. "The Art of Computer Programming", Volume 3 - Sorting and Searching, Addison-Wesley Publishing Company Inc, Reading, Mass, 1975. pp 506-549

[Maruyama76] Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", CACM, 19, 11. Nov 1976. pp 634-642

[Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247

[Pechura83] Pechura, M., and Schoeffler, J. "Estimating File Access Time of Floppy Disks", CACM, 26, 10. Oct 1983. pp 754-763

[Peterson83] Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55

[Powell79] Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42

[Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

[Smith81a] Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117

[Smith81b] Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54

[Symbolics81] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.

[Thompson78] Thompson, K. "UNIX Implementation", Bell System Technical Journal, 57, 6, part 2. pp 1931-1946 July-August 1978.

[Thompson80] Thompson, M. "Spice File System", Carnegie-Mellon University, Department of Computer Science, Pittsburg, PA 15213 #CMU-CS-80, Sept 1980.

[Trivedi80] Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473

[White80] White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

Fsck – The UNIX† File System Check Program

Marshall Kirk McKusick

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This document reflects the use of *fsck* with the 4.2BSD and 4.3BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

File System Check Program (*fsck*) is an interactive file system check and repair program. *Fsck* uses the redundant structural information in the UNIX file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. Unless there has been a hardware failure, *fsck* is able to repair corrupted file systems using procedures based upon the order in which UNIX honors these file system update requests.

The purpose of this document is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by *fsck*. Both the program and the interaction between the program and the operator are described.

Revised October 7, 1996

---

†UNIX is a trademark of Bell Laboratories.

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

TABLE OF CONTENTS

1. Introduction

This document reflects the use of *fsck* with the 4.2BSD and 4.3BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. *Fsck* runs in two modes. Normally it is run non-interactively by the system after a normal boot. When running in this mode, it will only make changes to the file system that are known to always be correct. If an unexpected inconsistency is found *fsck* will exit with a non-zero exit status, leaving the system running single-user. Typically the operator then runs *fsck* interactively. When running in this mode, each problem is listed followed by a suggested corrective action. The operator must decide whether or not the suggested correction should be made.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of deterministic corrective actions used by *fsck* (the Coast Guard to the rescue) is presented.

2. Overview of the file system

The file system is discussed in detail in [Mckusick84]; this section gives a brief overview.

2.1. Superblock

A file system is described by its *super-block*. The super-block is built when the file system is created (*newfs*(8)) and never changes. The super-block contains the basic parameters of the file system, such as the number of data blocks it contains and a count of the maximum number of files. Because the super-block contains critical data, *newfs* replicates it to protect against catastrophic loss. The *default super block* always resides at a fixed offset from the beginning of the file system's disk partition. The *redundant super blocks* are not referenced unless a head crash or other hard disk error causes the default super-block to be unusable. The redundant blocks are sprinkled throughout the disk partition.

Within the file system are files. Certain files are distinguished as directories and contain collections of pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps indicating modification and access times for the file, and an array of indices pointing to the data blocks for the file. In this section, we assume that the first 12 blocks of the file are directly referenced by values stored in the inode structure itself†. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 4096 byte block size, a singly indirect block contains 1024 further block addresses, a doubly indirect block contains 1024 addresses of further single indirect blocks, and a triply indirect block contains 1024 addresses of further doubly indirect blocks (the triple indirect block is never needed in practice).

In order to create files with up to 2 32 bytes, using only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block, so it is possible for file systems of different block sizes to be accessible simultaneously on the same system. The block size must be decided when *newfs* creates the file system; the block size cannot be subsequently changed without rebuilding the file system.

2.2. Summary information

Associated with the super block is non replicated *summary information*. The summary information changes as the file system is modified. The summary information contains the number of blocks, fragments, inodes and directories in the file system.

†The actual number may vary from system to system, but is usually in the range 5-13.

2.3. Cylinder groups

The file system partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Each cylinder group includes inode slots for files, a *block map* describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. A fixed number of inodes is allocated for each cylinder group when the file system is created. The current policy is to allocate one inode for each 2048 bytes of disk space; this is expected to be far more inodes than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for the *i+1*st cylinder group is about one track further from the beginning of the cylinder group than it was for the *i*th cylinder group. In this way, the redundant information spirals down into the pack; any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information stores data.

2.4. Fragments

To avoid waste in storing small files, the file system space allocator divides a single file system block into one or more *fragments*. The fragmentation of the file system is specified when the file system is created; each file system block can be optionally broken into 2, 4, or 8 addressable fragments. The lower bound on the size of these fragments is constrained by the disk sector size; typically 512 bytes is the lower bound on fragment size. The block map associated with each cylinder group records the space availability at the fragment level. Aligned fragments are examined to determine block availability.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. For example, consider an 11000 byte file stored on a 4096/1024 byte file system. This file uses two full size blocks and a 3072 byte fragment. If no fragments with at least 3072 bytes are available when the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file, as needed.

2.5. Updates to the file system

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. The file system stages all modifications of critical information; modification can either be completed or cleanly backed out after a crash. Knowing the information that is first written to the file system, deterministic procedures can be developed to repair a corrupted file system. To understand this process, the order that the update requests were being honored must first be understood.

When a user program does an operation to change the file system, such as a *write*, the data to be written is copied into an internal *in-core* buffer in the kernel. Normally, the disk update is handled asynchronously; the user process is allowed to proceed even though the data has not yet been written to the disk. The data, along with the inode information reflecting the change, is eventually written out to disk. The real disk write may not happen until long after the *write* system call has returned. Thus at any given time, the file system, as it resides on the disk, lags the state of the file system represented by the in-core information.

The disk information is updated to reflect the in-core information when the buffer is required for another use, when a *sync*(2) is done (at 30 second intervals) by */etc/update*(8), or by manual operator intervention with the *sync*(8) command. If the system is halted without writing out the in-core information, the file system on the disk will be in an inconsistent state.

If all updates are done asynchronously, several serious inconsistencies can arise. One inconsistency is that a block may be claimed by two inodes. Such an inconsistency can occur when the system is halted

before the pointer to the block in the old inode has been cleared in the copy of the old inode on the disk, and after the pointer to the block in the new inode has been written out to the copy of the new inode on the disk. Here, there is no deterministic method for deciding which inode should really claim the block. A similar problem can arise with a multiply claimed inode.

The problem with asynchronous inode updates can be avoided by doing all inode deallocations synchronously. Consequently, inodes and indirect blocks are written to the disk synchronously (*i.e.* the process blocks until the information is really written to disk) when they are being deallocated. Similarly inodes are kept consistent by synchronously deleting, adding, or changing directory entries.

3. Fixing corrupted file systems

A file system can become corrupted in several ways. The most common of these ways are improper shutdown procedures and hardware failures.

File systems may become corrupted during an *unclean halt*. This happens when proper shutdown procedures are not observed, physically write-protecting a mounted file system, or a mounted file system is taken off-line. The most common operator procedural failure is forgetting to *sync* the system before halting the CPU.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a non-functional disk-controller.

3.1. Detecting and correcting corruption

Normally *fsck* is run non-interactively. In this mode it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this paper we assume that *fsck* is being run interactively, and all possible errors can be encountered. When an inconsistency is discovered in this mode, *fsck* reports the inconsistency for the operator to chose a corrective action.

A quiescent‡ file system may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system, or computed from other known values. The file system **must** be in a quiescent state when *fsck* is run, since *fsck* is a multi-pass program.

In the following sections, we discuss methods to discover inconsistencies and possible corrective actions for the cylinder group blocks, the inodes, the indirect blocks, and the data blocks containing directory entries.

3.2. Super-block checking

The most commonly corrupted item in a file system is the summary information associated with the super-block. The summary information is prone to corruption because it is modified with every change to the file system's blocks or inodes, and is usually corrupted after an unclean halt.

The super-block is checked for inconsistencies involving file-system size, number of inodes, free-block count, and the free-inode count. The file-system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of inodes. The file-system size and layout information are the most critical pieces of information for *fsck*. While there is no way to actually check these sizes, since they are statically determined by *newfs*, *fsck* can check that these sizes are within reasonable bounds. All other file system checks require that these sizes be correct. If *fsck* detects corruption in the static parameters of the default super-block, *fsck* requests the operator to specify the location of an alternate super-block.

‡ I.e., unmounted and not being written on.

3.3. Free block checking

Fsck checks that all the blocks marked as free in the cylinder group block maps are not claimed by any files. When all the blocks have been initially accounted for, *fsck* checks that the number of free blocks plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

If anything is wrong with the block allocation maps, *fsck* will rebuild them, based on the list it has computed of allocated blocks.

The summary information associated with the super-block counts the total number of free blocks within the file system. *Fsck* compares this count to the number of free blocks it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-block count.

The summary information counts the total number of free inodes within the file system. *Fsck* compares this count to the number of free inodes it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-inode count.

3.4. Checking the inode state

An individual inode is not as likely to be corrupted as the allocation information. However, because of the great number of active inodes, a few of the inodes are usually corrupted.

The list of inodes in the file system is checked sequentially starting with inode 2 (inode 0 marks unused inodes; inode 1 is saved for future generations) and progressing through the last inode in the file system. The state of each inode is checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes must be one of six types: regular inode, directory inode, symbolic link inode, special block inode, special character inode, or socket inode. Inodes may be found in one of three allocation states: unallocated, allocated, and neither unallocated nor allocated. This last state suggests an incorrectly formated inode. An inode can get in this state if bad data is written into the inode list. The only possible corrective action is for *fsck* is to clear the inode.

3.5. Inode links

Each inode counts the total number of directory entries linked to the inode. *Fsck* verifies the link count of each inode by starting at the root of the file system, and descending through the directory structure. The actual link count for each inode is calculated during the descent.

If the stored link count is non-zero and the actual link count is zero, then no directory entry appears for the inode. If this happens, *fsck* will place the disconnected file in the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated. If this happens, *fsck* replaces the incorrect stored link count by the actual link count.

Each inode contains a list, or pointers to lists (indirect blocks), of all the blocks claimed by the inode. Since indirect blocks are owned by an inode, inconsistencies in indirect blocks directly affect the inode that owns it.

Fsck compares each block number claimed by an inode against a list of already allocated blocks. If another inode already claims a block number, then the block number is added to a list of *duplicate blocks*. Otherwise, the list of allocated blocks is updated to include the block number.

If there are any duplicate blocks, *fsck* will perform a partial second pass over the inode list to find the inode of the duplicated block. The second pass is needed, since without examining the files associated with these inodes for correct content, not enough information is available to determine which inode is corrupted and should be cleared. If this condition does arise (only hardware failure will cause it), then the inode with the earliest modify time is usually incorrect, and should be cleared. If this happens, *fsck* prompts the operator to clear both inodes. The operator must decide which one should be kept and which one should be cleared.

Fsck checks the range of each block number claimed by an inode. If the block number is lower than the first data block in the file system, or greater than the last data block, then the block number is a *bad block number*. Many bad blocks in an inode are usually caused by an indirect block that was not written to the file system, a condition which can only occur if there has been a hardware failure. If an inode contains bad block numbers, *fsck* prompts the operator to clear it.

3.6. Inode data size

Each inode contains a count of the number of data blocks that it contains. The number of actual data blocks is the sum of the allocated data blocks and the indirect blocks. *Fsck* computes the actual number of data blocks and compares that block count against the actual number of blocks the inode claims. If an inode contains an incorrect count *fsck* prompts the operator to fix it.

Each inode contains a thirty-two bit size field. The size is the number of data bytes in the file associated with the inode. The consistency of the byte size field is roughly checked by computing from the size field the maximum number of blocks that should be associated with the inode, and comparing that expected block count against the actual number of blocks the inode claims.

3.7. Checking the data associated with an inode

An inode can directly or indirectly reference three kinds of data blocks. All referenced blocks must be the same kind. The three types of data blocks are: plain data blocks, symbolic link data blocks, and directory data blocks. Plain data blocks contain the information stored in a file; symbolic link data blocks contain the path name stored in a link. Directory data blocks contain directory entries. *Fsck* can only check the validity of directory data blocks.

Each directory data block is checked for several types of inconsistencies. These inconsistencies include directory inode numbers pointing to unallocated inodes, directory inode numbers that are greater than the number of inodes in the file system, incorrect directory inode numbers for "**.**" and "**..**", and directories that are not attached to the file system. If the inode number in a directory data block references an unallocated inode, then *fsck* will remove that directory entry. Again, this condition can only arise when there has been a hardware failure.

Fsck also checks for directories with unallocated blocks (holes). Such directories should never be created. When found, *fsck* will prompt the user to adjust the length of the offending directory which is done by shortening the size of the directory to the end of the last allocated block preceeding the hole. Unfortunately, this means that another Phase 1 run has to be done. *Fsck* will remind the user to rerun fsck after repairing a directory containing an unallocated block.

If a directory entry inode number references outside the inode list, then *fsck* will remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "**.**" must be the first entry in the directory data block. The inode number for "**.**" must reference itself; e.g., it must equal the inode number for the directory data block. The directory inode number entry for "**..**" must be the second entry in the directory data block. Its value must equal the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory). If the directory inode numbers are incorrect, *fsck* will replace them with the correct values. If there are multiple hard links to a directory, the first one encountered is considered the real parent to which "**..**" should point; *fsck* recommends deletion for the subsequently discovered names.

3.8. File system connectivity

Fsck checks the general connectivity of the file system. If directories are not linked into the file system, then *fsck* links the directory back into the file system in the *lost+found* directory. This condition only occurs when there has been a hardware failure.

Acknowledgements

I thank Bill Joy, Sam Leffler, Robert Elz and Dennis Ritchie for their suggestions and help in implementing the new file system. Thanks also to Robert Henry for his editorial input to get this document

together. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235. (Kirk McKusick, July 1983)

I would like to thank Larry A. Wehr for advice that lead to the first version of *fsck* and Rick B. Brandt for adapting *fsck* to UNIX/TS. (T. Kowalski, July 1979)

References

[Dolotta78] Dolotta, T. A., and Olsson, S. B. eds., *UNIX User's Manual, Edition 1.1*, January 1978.

[Joy83] Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, M., and Mosher, D. 4.2BSD System Manual, *University of California at Berkeley*, *Computer Systems Research Group Technical Report* #4, 1982.

[McKusick84] McKusick, M., Joy, W., Leffler, S., and Fabry, R. A Fast File System for UNIX, *ACM Transactions on Computer Systems 2*, 3. pp. 181-197, August 1984.

[Ritchie78] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1905-29.

[Thompson78] Thompson, K., UNIX Implementation, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1931-46.

12 Seriella och parallella bussar

> Begrepp i denna föreläsning: *Seriell buss, RS232C, baud, paritet, spänningsnivåer, RS485, differential spänning, multidroptopologi, master/slave, bussarbitrering, I2C, CAN, USB, bandbreddsbudget, strömbudget, FireWire, parallell buss, synkron/asynkron buss, wait-cykel, DTACK, DMA, ISA, portmappad och minnesmappad I/O, PCI, PCI-brygga, burst mode, SCSI, SCSI-controller, kommandokö, överlappande I/O.*

För att fortsätta vandringen genom minneshierarkin fortsätter vi här med några möjliga kommunikationskanaler mellan processor och kringenheter. Vi förstår att det är av central betydelse i ett datorsystem är möjligheten att de olika enheterna kan kommunicera med varann. Även ut mot yttervärlden är kommunikationen viktig, vare sig det gäller en monitor, skrivare eller skanner.

Vi ska betrakta några kommunikationskanaler. Med början i de ursprungligen långsammare seriella bussarna fortsätter vi sedan med de mycket snabbare parallella bussarna. Seriella kommunikationskanaler har funnits länge men att betrakta dem som bussar och införa mer eller mindre komplicerade protokoll för dataöverföringen är relativt nytt.

Man ska inte ledas till den villfarelsen att seriella bussar är omoderna och hör forntiden till. Den moderna miniatyriseringen ställer krav på bland annat att benantalet på en komponent måste minimeras så långt det är möjligt och vi har en hel flora av moderna komponenter (A/D-omvandlare till exempel) som uteslutande meddelar sig med omvärlden via en seriell kanal.

Även hårddiskar med seriell kommunikation finns numera på marknaden.

12.1 Seriella bussar

12.1.1 RS232C

RS232, egentligen RS232C, är en överlevare från 50-talet. RS232 är en standard som definierar hur serieöverföringen ska gå till och vilka spänningar som skall användas.[1]

Datatakten anges i *baud* där baudtalet anger hur många bitar, inklusive paritetsbitar och stoppbitar som kan överföras per sekund. Hastigheten var tidigare i gynnsamma fall 9600 baud — att jämföra med den långsammaste Ethernetstandarden för moderna nätverk som ligger på 10 miljoner bitar per sekund. Numer är även hastigheterna 57 600 och 115 200 baud inte ovanliga för RS232.[2] Den maximalt användbara överföringshastigheten

[1] Innan Internet etablerades rikstäckande nät med hjälp av modem och RS232-kommunikation!

[2] 115 200 baud kan också skrivas 115.2 kbaud, dvs prefixet "*k*", betyder "1000" och inte "1024" som annars är vanligt i datorsammanhang.

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

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:

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.

12.1.3 I2C, Inter-Integrated Circuit

Under 1980-talet utvecklade Philips en seriell buss för att kunna sammankoppla de olika kretsar som ingick i, framförallt, sina TV-apparater. Dess prestanda är modesta med dagens mått mätt, ursprungligen var bittakten satt till högst 100 kbit/s men senare utkom en reviderad specifikation som tillåter upp till 400 kbit/s bittakt. All signalering sker med 0 och 5V spänningar.

Signalering sker även här med två trådar, men dessa två trådar används i båda riktiningarna. Den ena tråden SDA (*Serial DatA*) överför informationen och den andra, SCL (*Serial CLock*), innehåller klockinformation så att den andra enheten vet *när* den kan läsa datat, när klockan än hög.

Varje dataöverföring på I2C-bussen kan uppdelas i fyra delar:

- Etablera ett starttillstånd,
- överför en eller flera 8-bitars ord,
- invänta eller generera en bekräftelse, *acknowledge*, på att bitarna överförts samt slutligen,
- etablera ett stopptillstånd.

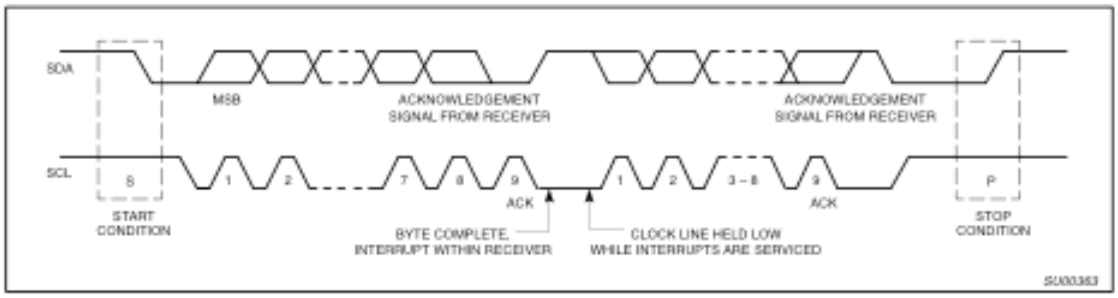


Figure 7. Data transfer on the I²C-bus

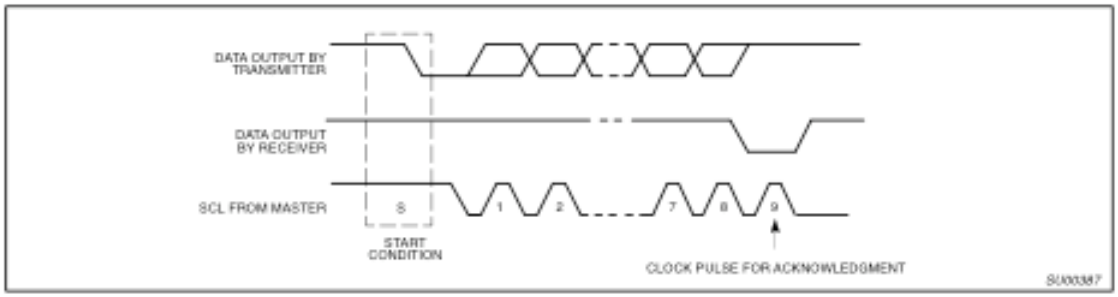


Figure 8. Acknowledge on the I²C-bus

I2C har också ett master/slave-förhållande. Förutom att mastern alltid är den som initierar en kommunikation är det alltid mastern som genererar klocksignalen, SCL. Detta alltså oberoende om mastern är sändare eller mottagare för tillfället.

Varje enhet på I2C-bussen måste ha en unik adress och detta åstadkoms genom att antingen varje komponenttyp har en unik fast identifiering eller att en grupp av komponenter (minnen exempelvis) har en fix gruppadress och ytterligare adress bestäms genom

att löda några ben på kapseln till digital nolla eller etta. I fallet med minnet kan man på så sätt använda flera minnen av samma sort, man måste bara själv se till att de ges en unik adress genom hur man ansluter kapseln.

I2C-bussen är mycket populär och många tillverkare använder samma protokoll utan att explicit nämna varken I2C eller Philips vid namn. Om en komponent stödjer ”en två-trådsbuss” kan man vara rätt säker på att den beter sig som en I2C-komponent.

12.1.4 CAN, Controller Area Networking

För automotiva sammanhang utvecklades av Bosch i Tyskland CAN-bussen. Varje bil har några mil kabel i sig och i ett försök att minska alla dessa kabelstammar konstruerade man denna seriella buss. Detta är liksom I2C en två-trådsbuss med master/slave men signaleringen sker differentiellt och den är därmed väldigt stryktålig mot störningar.

Tanken är att varje enhet i en bil (blinkers, tuta osv) ska sitta ihop med denna enda buss. En centraldator genomför då ”blinka höger” genom att till höger blinkerspar signalera ”tänd”, följt av ett ”släck” en halvsekund senare, följt av ett ”tänd” efter ytterligare en halvsekund och så vidare. Det krävs förstås att varje enhet är försedd med en CAN-mottagarkrets men förutom det behövs bara själva CAN-bussen och batterispänningen. Som vanligt på bilar är chassiet anslutet till batteriets minus-pol så det är, strängt taget, onödigt att även dra en jordledning ut till varje enhet. Vid jordfel i elsystemet kan strömmar uppstå på ställen man minst anar det och man gör trots allt bäst i att dra all spänningsmatning till CAN-systemet separat.

Bussen tillåter upp till 1 024 slavar och 1 master. Datatakten är högst 1 Mbit/s. Många slaveneheter är dock inte konstruerade för denna datatakt, utan följer en reducerad specifikation om en datatakt på 125 kbit/s på avstånd upp till 80 meter.

12.1.5 USB (Universal Serial Bus)

För hopkoppling av datorer och datorprylar krävs i många fall en betydligt högre bandbredd än vad vi hittills berört. Att dessutom åstadkomma denna bandbredd med en seriell buss har inte blivit möjligt förrän mot slutet av 1990-talet.

USB-bussen är lätt att använda: Bara plugga i respektive enhet, även under drift, och den ska fungera. För att få all denna lättanvändhet kräver bussen ett mycket avancerat protokoll. Inte bara ska enheten identifieras, mastern ska även göra upp en *bandbreddsbudget* för hela bussen och kan strypa datatakten för vissa enheter så att andra kan få tillgång till bandbredden. Vissa enheter kan behöva en viss minsta bandbredd för att fungera som väntat. Till exempel kan en webkamera köpslå med mastern och bli garanterad en viss minsta bandbredd så att bilderna inte blir ryckiga.

Somliga USB-enheter har ingen egen strömförsörjning utan måste ta den via bussen från andra enheter. För den skull innehåller bussprotokollet även en *strömbudget*eringsfunktion för att tillfredsställa allas krav så långt det går, eller stänga av vissa USB-enheter om de meddelar att de kräver mer ström än vad systemet kan ge för tillfället. Varje USB-enhet med extern strömförsörjning kan leverera 0.5 ampere till övriga enheter.[4]

[4] Vissa tillverkare har tagit fasta på det och tillverkar tangentbordsbelysning, eltandborstar m m för anslutning till USB-porten.

Prestanda är höga: Standarden stipulerar en maximal bandbredd på 12 Mbit/s. Förutom denna *Full Speed*-mod, FS, definieras också en *Low Speed*-mod, LS, om 1.5 Mbit/s. Totalt kan 127 enheter anslutas till samma buss med högst 5 meter mellan varje enhet.

En USB-enhet spänningsförsörjs med enbart 5 volt och signalnivåerna är 0.8 respektive 2.5 volt. Differentiell signalering används och en minsta spänningsskillnad på 0.2 volt behövs för att detektera en bit.

12.1.6 FireWire (IEEE-1394)

Behovet av bandbredd minskar inte. Speciellt multimediatillämpningar och realtidsvideo kräver mer bandbredd än USB kan leverera och för att täcka det behovet har IEEE och ett 40-tal elektroniktillverkare slagits sig ihop om den nya standarden *FireWire*. FireWire är en antagen industristandard som tillåter 63 enheter att kommunicera på samma buss, varje segment d v s avståndet mellan två enheter, är högst 4.5 meter, där bussen har en bandbredd på 400 Mbit/s. En utökning av brandbredden till 1.2 Gbit/s är planerad.

Liksom USB klarar FireWire att anslutas under drift och man behöver alltså inte starta om datorn för att ansluta en FireWire-enhet.

FireWire är en relativt sen standard, den fastslogs 1999.

12.2 Parallella bussar

Det känns är onekligen lite egendomligt att kalla seriell kommunikation för busskommunikation som vi gjort ovan, men närvaron av komplicerade protokoll och handskakningar gör att även seriell dataöverföring brukar kallas för bussöverföring. Nu ska vi emellertid titta närmare på den traditionella konstruktionen av bussar, där dataöverföringen sker på flera parallella signaler.

Vi har redan, ehuru underförstått, introducerat bussbegreppet bland annat när vi tittade in i mikromaskinens inre och när vi såg hur processorn 68000 matades med data från yttre minnen. Vi kommer se att man i huvudsak skiljer mellan synkrona och asynkrona bussar och att man dessutom kan använda begreppen master och slav även när det gäller parallella bussar.

12.2.1 Synkron buss

Med synkron buss avses en buss där data överförs i takt med systemets klocka.[5] Det är en enkel och mycket vanlig buss hos alla enklare mikroprocessorer som 8080, Z80, 6800 o s v. Busstimingen framgår av bilden nedan och man ser att den yttre enheten, typiskt ett minne, måste vara så snabbt att det, vid läsning, hinner leverera data innan den synkrona busscykeln är slut. Ett analogt resonemang gäller för skrivning.

[5] De seriella bussarna med klocka som ”medlevererades” ovan kallas också synkrona av samma anledning.

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.

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

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

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]

(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 insticksкort, *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.

överföra data.

Man kan undra varför SCSI blev en sådan succé? Mycket beror på att den representerar ett enhetligt sätt att kommunicera mellan hårdvaruenheter. Innan SCSI:s inträde på scenen var alla interface speciella: det fanns ett för hårddisken, ett annat för bandstationen och så vidare. Nu kunde man plötsligt koppla ihop alla enheter på en gemensam buss.

Till varje SCSI-buss hör en *controller*, oftast ett separat instickskort (det jag kallade SCSI-adapter ovan) till datorn. Controllern fungerar som master och kan allokera plats på bussen för 8 (eller i modernare fall 16) enheter. Controllern själv ligger på bussen och äter upp en av dessa positioner, vilket gör att man i praktiken kan ansluta 7 eller 15 andra SCSI-enheter. Man förstår att detta är mycket praktiskt för att till exempel konstruera RAID-system där man kanske kan behöva 5 stycken parallella hårddiskar.

Närvaron av en controller gör också att datorsystemets processor kan lägga ut kommandon till denna i en kommandokö och sedan överlåta åt controllern att fixa detaljerna, dvs optimera kommandoordningen, genomföra själva överföringen och kontrollera checksummor innan den meddelar processorn att den är klar.

Controllern håller också reda på de olika anslutna enheternas maximala överföringshastighet och anpassar sig efter dessa. Man kan alltså ha både långsamma och snabba SCSI-enheter anslutna till samma buss.

SCSI-bussen är ett master/slave-protokoll men enligt SCSI-nomenklatur kallas dessa *initiator* och *target*.

Det som gör SCSI överlägset andra bussprotokoll är den intelligens som ligger i controllern och de andra enheterna. SCSI-bussen stödjer *command queuing* med vilket menas möjligheten att arrangera om I/O-kommandona så att genomflödet blir maximalt. SCSI-bussen kan behandla kommandon parallellt och kan sprida läsningar och skrivningar mellan olika enheter på bussen så att dataflödet maximeras. Man behöver alltså inte göra klart en transaktion innan nästa kan påbörjas, utan använder s.k. *overlapped I/O*. De olika enheterna har dessutom egna buffrar så att, när väl informationen samlats ihop vid en läsning av en hårddisk till exempel, den mycket snabbt kan överföras till rätt mottagare på bussen. Naturligtvis sker överföringen med DMA utan processorns onödiga inblandning.

Den nya standarden SCSI-3 definierar förutom synkron och asynkron överföring även seriell överföring.

Tankenöt: Om man köper en hårddisk som stödjer Ultra160 och har ett modernt controllerkort med stöd för Ultra160 och en snabb dator visar det sig att man trots detta inte lyckas komma upp i 160Mb/s när man flyttar filer på den. Varför det? Vad hindrar?

SCSI är numera en mycket omfattande standard och en översvallande flora av bokstavsförkortningar HVD/SPI/Wide SCSI/QAS... Vilket tillsammans med kravet på terminering[9] gör att ihopkoppling av enheter och samtidigt veta vad man gör är rätt krångligt, vilket torde framgå av det antal getter som måste offras för att få igång sin SCSI-buss:

> ”Getting a SCSI chain working is perfectly simple if you remember that there must be exactly three terminations: one on one end of the cable, one

[9] Terminering behövs, speciellt vid de högre hastigheterna, för att undvika spänningsreflexioner i bussen på grund av impedansförändringar längs densamma.

on the far end and the goat terminated over the SCSI chain with at silver-handled knife whilst burning *black* candles" — Anthony DeBoer

"SCSI is not magic. There are *fundamental technical reasons* why you have to sacrifice a young goat to your SCSI-chain every now and then." — John F. Woods

"But if it is a differential SCSI chain, you need two goats, one black and one white, and two ceremonies: kill the black goat at high noon and the white one at midnight. Same silver knife for both, of course. Otherwise the chain will be unbalanced and things just get worse from there as all the drives do the "washing machine dance". Selfterminating devices merely need to be left with sufficient livestock and they'll take care of the rest." — Graham Reed

13 Mikrokontroller

> Begrepp i denna föreläsning: *Mikrokontroller, egenskaper och inbyggda funktioner, timer, watch dog timer, klockalternativ, black-out, brown-out.*

Hittills har vi studerat traditionella mikroprocessorer och speciellt deras arkitektur och hur de kan vara konstruerade internt med digitalteknikkomponenter och mikrokod. Ser man tillbaka så har vi specialstuderat just mikroprocessorns uppbyggnad och insida, inte några detaljer om den miljö den måste vara i för att få något gjort överhuvudtaget. Även om vi inte betonat det kan inte processorn leva ett eget liv utan yttre hjälpkomponenter, exempelvis:

- Processorn behöver en klocka för att takta den interna digitaltekniken,
- ett yttre programminne för att ha någonstans att hämta sina instruktioner,
- sannolikt ett yttre dataminne för att mellanlagra information.

De få interna register vi behandlat räcker med all säkerhet inte till för all information, ens ett litet, program måste hantera. Sammantaget kan vi förstå att en ensam processor inte är till någon större nytta, den är i hög grad beroende av kringkomponenter för att komma till sin rätt.

Laborationerna i kursen har utförts på ett i många avseenden minimalt mikroprocessorsystem: Tutor-kortet. Detta kort innehåller processorn, PROM (*Programmable Read Only Memory*), RAM (*Random Access Memory*), ”klister”-komponenter i form av enkla logikkretsar, kretsar för att kunna kommunicera med omgivningen m. m. Mindre än så blir inte gärna ett 68000- eller 68008-baserat system. Även om prestanda för systemet är hyggliga är storleken och kostnaden avskräckande för många användningsområden.

Det har sedan datorernas barndom alltid funnnits behov av att minimera den fysiska storleken hos dem för att kunna användas i de mest skilda områden. Ska processorn styra en tvättmaskin kanske Tutors storlek kan acccepteras (men knappast kostnaden), ska processorn styra en datormus är redan storleken ett hinder. Med denna bakgrund är det knappast någon överraskning att man tidigt försökte hitta lösningar där *små* processorer ändå kunde ha resonabla prestanda — åtminstone för vissas speciella tillämpningar.

Med *mikrokontrollern* försöker man lösa dilemmat. En mikrokontroller är en kombination av en processor som vi känner den och de kringkretsar som behövs för att den skall fungera. Sålunda innehåller en mikrokontroller både egen klockkrets, program- och dataminne, ofta flera speciella hårdvaruenheter o. s. v. Tanken är att användaren inte ska behöva ansluta för processorn livsnödvändiga komponenter utan direkt kunna använda mikrokontrollern som ett färdigt byggblock i sitt system.

13.1 En verklig mikrokontroller: PIC16C84

Redan på 1970-talet konstruerades på General Instruments i USA en krets med namnet PIC[1]. Denna var ett försök att konstruera en liten databahandlande enhet för att styra en del kringutrustning (*peripherals*) i större datorer. Kanske skulle ett tangentbord kunna dra nytta av en liten processor? Vilken marknad![2]

Man kan anta att processorn inte blev den succé man väntat sig, GI bestlutade i varje fall att deras halvledarsatsning skulle säljas av och PIC-processorn slutade tillverkas.

I början på 1980-talet köptes resterna av PIC-tillverkningen upp av det då okända företaget Microchip. Microchip fortsatte utvecklingen av arkitekturen. Man införde några konstruktionsändringar, men framförallt såg man till att förenkla konstruktionen så att man genom att utnyttja moderna tillverkningstekniker kunde få ner tillverkningskostnaden. Och med dessa nya grepp började PIC-arkitekturen sprida sig över världen. Det fanns ett uppdämt intresse för denna sorts komponent och de såldes i enorma mängder, hundratals miljoner enheter om året. Flera nya varianter togs fram men samtliga hade ett tydligt arv från General Instruments' arkitektur från tidigt 1970-tal.

För den elektronikintresserade allmänheten kom Microchips genombrott i och med processorn PIC16C84[3]. Denna är en 8-bitars processor med tillräckligt mycket program- och dataminne ombord för att kunna användas till åtskilliga mindre projekt. Dess assembler kanske kan tyckas vara lite "bakvänd", speciellt om man är bortskämd med 68000:s eleganta instruktioner, men den innehåller bara drygt trettio instruktioner och är lätt att lära sig.

Mikrokontrollern 16C84 kompletterades och ersattes efter några år av varianten 16F84. Skillnaden är den minnesteknik som används i programminnet och är inte väsentlig för oss här. Internt ser de båda likadana ut, de har samma programmerarmodell, programminne, instruktioner osv. Det är den tidigare varianten vi ska studera i fortsättningen.

13.2 Datablad PIC16C84

Mikrokontrollers är en balansgång mellan att vara så enkel och billig som möjligt men ändå tillåta användning i så skilda omgivningar som möjligt. Ofta hänger lämpligheten i en viss tillämpning inte på klockfrekvens och instruktionstakt utan på hur mikrokontrollerns innehåll kan undvika ytterligare externa komponenter.

Med de kompromisser Microchip gjorde resulterade konstruktionen i följande mikrokontroller:

[1] Jag har ett datablad på PIC7000 från 1984 om nån är intresserad. Rent akademiskt intresse allstå, den går inte att få tag på numera.

[2] Det visade sig sedan att Intel knep den marknaden. I alla tangentbord till PC finns processorn 8048 från Intel.

[3] Microchip hävdar numer att PIC inte betyder *Peripheral Integrated Controller*, utan bara betyder "PIC".

MICROCHIP

PIC16C84

8-Bit CMOS EEPROM Microcontroller

High Performance RISC CPU Features

- Only 35 single word instructions to learn
- All instructions single cycle (400 ns @ 10MHz) except for program branches which are two-cycle
- Operating speed: DC - 10MHz clock input
  DC - 400 ns instruction cycle
- 14-bit wide instructions
- 8-bit wide data path
- 1K x 14 EEPROM program memory
- 36 x 8 general purpose registers (SRAM)
- 64 x 8 on-chip EEPROM data memory
- 15 special function hardware registers
- Eight-level deep hardware stack
- Direct, indirect and relative addressing modes
- Four interrupt sources:
  - External RB0/INT pin
  - TMR0 timer overflow
  - PORTB<7:4> interrupt on change
  - Data EEPROM write complete
- 1,000,000 data memory EEPROM ERASE/WRITE cycles
- EEPROM Data Retention > 40 years

Peripheral Features

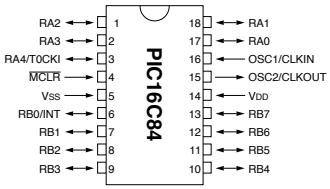
- 13 I/O pins with individual direction control
- High current sink/source for direct LED drive
  - 25 mA sink max. per pin
  - 20 mA source max. per pin
- TMR0: 8-bit timer/counter with 8-bit programmable prescaler

Special Microcontroller Features

- Power-on Reset (POR)
- Power-up Timer (PWRT)
- Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own on-chip RC oscillator for reliable operation
- Code protection
- Power saving SLEEP mode
- Selectable oscillator options
- Serial In-System Programming - via two pins

Pin Diagram

PDIP, SOIC



CMOS Technology

- Low-power, high-speed CMOS EEPROM technology
- Fully static design
- Wide operating voltage range:
  - Commercial: 2.0V to 6.0V
  - Industrial: 2.0V to 6.0V
- Low power consumption:
  - < 2 mA typical @ 5V, 4 MHz
  - 60 μA typical @ 2V, 32 kHz
  - 26 μA typical standby current @ 2V

Detta är första sidan på mikrokontrollerns datablad och innehåller viktiga typdata. Vi ska studera dessa data och bedöma dem utifrån det vi redan vet om mikroprocessorer.

Programminne Kontrollern har maximalt plats för ett program på 1024 rader. Punkt! Det finns inga möjligheter att utöka minnesmängden. Detta är naturligtvis en begränsning, men det visar sig att man med 1024 rader program kan komma förvånansvärt långt. Man kanske till och med kan sträcka sig till att säga att när man fyllt dessa rader så har man antagligen utnyttjat processorns övriga funktioner till bristningsgränsen. Programmet ligger i ett EEPROM, Elektriskt raderbart PROM. Skulle det visa sig att man tänkt fel vid programkonstruktionen kan en ny version av programmet enkelt laddas ner igen. Tack vare denna möjlighet minskar programutvecklingstiden avsevärt.

Dataminne Förutom några interna register — som inte är lämpade för ren datalagring i alla fall — förlitar sig processorn på en s.k. *Register File*, ett internt dataminne på 36 bytes. 36 bytes är inte mycket, inte ens i denna minimala omgivning, och är det någonstans man stöter på en tydlig begränsning är det här[4].

De interna register som nämndes ovan och som inte är lämpade för datalagring används för att komma åt hårdvarufunktioner i processorn, exempelvis I/O-enheter. Även några normala processorregister återfinns som minnesmappade, indexregistret och statusregistret bl. a.

Extra dataminne I vissa fall kan det vara bra att ha lite extra data. Det kan handla om konstanter, kodnycklar, kalibreringsvärden till givare eller kanske rena texter att mata ut på en display. För denna skull innehåller kontrollern också 64 bytes EEPROM. Detta minne är något omständligare att använda än övrigt dataminne men har fördelen att kontrollern själv kan ändra i det. Det behåller även sitt värde efter strömavbrott.

Innan man förleds att använda EEPROM:et för frikostigt måste man notera att det har en begränsad livslängd. Högst 1 000 000 skrivningar till det kan utföras. Utöver detta lovar tillverkaren inte något. Av denna anledning skall man ofta undvika att lagra mätadata osv i EEPROM.[5]

Skrivning till EEPROM tar längre tid än skrivning till de interna registren, typ 10 millisekunder. Läsning är däremot lika snabb som läsning från andra interna register.

Klockfrekvens En komponent som denna (egentligen avhängigt tillverkningsprocessen CMOS) drar ström proportionerligt med klockfrekvensen. Enligt databladet förbrukar den endast 2 mA vid 4 MHz klockfrekvens, så den drar under alla förhållanden lite ström. Men för verkligt portabla och batteridrivna applikationer kan man dra ner klockfrekvensen till ren likström om man skulle vilja. Med en klockfrekvens på 0 Hz utför processorn förstås inga instruktioner men vill man, kan man alltså spara ström genom att stänga av klockan helt. Även med en så låg frekvens kommer de interna registren att behålla sina värden. Vi vet att det betyder att minnestypen måste vara av den statiska typen.

Varianter av processorn finns för maximala klockfrekvenser om 10 MHz. Det har visat sig att processorn välvilligt låter sig *överklockas*, dvs även en variant för maximalt 4 MHz kan oftast köras i 10 MHz eller mer. Men gör man så får man vara beredd på att kanske

[4] Efterföljaren 16F84 uppmärksammade detta problem och var försett med mer dataminne — 68 bytes. Nåja, en liten förbättring var det väl.

[5] Hur lång livslängd har minnet om en skrivning per sekund görs?

vissa instruktioner inte ”hinner med” eller att processorn plötsligt upphör att exekvera kod. Men i allmänhet fungerar det faktiskt bra.

Även om en processor kan klockas med 10 MHz, och därvid göra 2,5 miljoner instruktioner per sekund i det här fallet, är det fullt naturligt och vanligt att klocka den med bara 32 eller 455 kHz. I det förra fallet använder man billiga massproducerade kristaller avsedda för armbandsur och liknande, i det senare utnyttjas keramiska resonatorer egentligen avsedda för mellanfrekvensdelen i radiomottagare. Med en klockfrekvens om 32 kHz exekveras lite drygt 8000 instruktioner per sekund.

Stack För att kunna göra avancerade program med subrutiner och avbrott måste kontrollern vara försedd med en stack. Den stack denna mikrokontroller har skiljer sig från den vi tidigare har beskrivit i det att den inte har någon stackpekare som är åtkomlig för programmeraren. Man kan heller inte temporärlagra data på stacken. Kanske lika bra det eftersom den har en fix storlek på enbart åtta adresser. Fler än åtta (nästlade) subrutinanrop kan man alltså inte tillåta.

Avbrott Ofta är en mikrokontrollers syfte att sitta i mindre inbyggda system som en självständig enhet. Inte sällan innebär detta att den är ensam ansvarig för att hålla samman alla kringenheter. Och för den sakens skull är det ofta kristiskt att den kan svara på och behandla olika avbrott. Avbrottsmekanismen liknar den vi känner till, fast den tillåter här flera olika *orsaker* till avbrott (*avbrottskällor*). Trots detta har den inte prioriterade avbrott som 68000, inte heller har den ett avancerat avbrottssystem med $TRAP$:s som 68000.

Avbrott till processorn kan initieras av

- en förändring på pinnen INT, eller
- förändring på någon av pinnarna $< RB7 : RB4 >$ eller
- om en intern *timer* räknat ner till noll.

Det finns även en fjärde avbrottskälla, som meddelar att skrivning till den interna (data)EEPROM:et slutförts, men den kan inte användas som de tidigare nämnda.

Vid avbrott kommer programexekveringen att överföras till rad 4 i assemblerprogrammet. Denna adress är bestämd av tillverkaren och kan inte förändras. Samtliga olika avbrottskällor ger detta avbrott och det är alltså inte möjligt att låta varje avbrottskälla ha sin egen avbrottsrutin. Det är avbrottsrutinens första uppgift att ta reda på vad som orsakade avbrottet.

Strömförbrukning Som nämnts hänger klockfrekvens och strömförbrukning intimt ihop. Även vid den relativt höga klockfrekvensen (för mikrokontrollers i varje fall) 4 MHz drar den inte mer än 2 mA. Perfekt för batteridrift. Kan man släppa kraven på klockfrekvens och gå ner på 32 kHz drar kretsen endast — närmast obefintliga — 60μA. Man ser också att kretsen ska kunna fungera med matningsspänningar varierande mellan 2.0 och 6.0 volt.

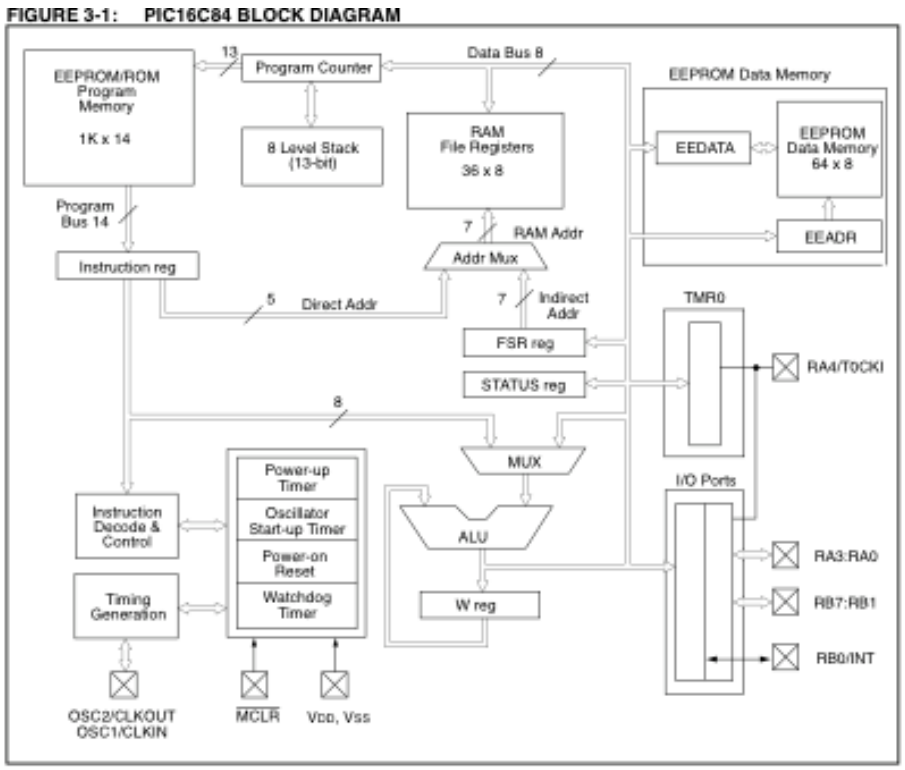
Med en speciell instruktion, `SLEEP`, kan mikrokontrollern ytterligare minska sin strömförbrukning till endast 26μA! Denna instruktion stänger i praktiken av nästan hela processorn, den enda funktion som fortfarande gör något är timern. Ett timeravbrott kan väcka mikrokontrollern ur sin nirvana, så en timer måste fortfarande fungera även om resten av processorn är bortkopplad, annars skulle strömförbrukningen kunna minskas ytterligare.

Processorn kan väckas från sin `SLEEP`-inducerade dvala genom timeravbrott (se avsnittet om vakthunden nedan), yttre `RESET` eller förändring av det logiska tillståndet på någon pinne på port B (se avsnittet om portar).

13.3 Arkitektur

Jämfört med den bekanta M68000-processorn är PIC-processorns arkitektur mycket udda. Detta är främst av två orsaker:

1. Arkitekturen är optimerad att ta så lite hårdvara i anspråk som möjligt, och
2. den ursprungliga arkitekturen är över 25 år gammal. Mycket nya tankar om processorarkitektur har tillkommit sedan dess.



Emellertid känner vi igen flera drag från den enklare arkitekturen MMM som användes på mikroprogrammeringslaborationerna. I blockschemat återfinner vi bl.a. ALU (Arithmetic Logic Unit), statusregistret, programräknare (Program Counter) och instruktionsregistret (Instructin Reg).

Uppe till vänster i schemat ligger det inprogrammerade assemblerprogrammet i EEPROM. Här används ett enda instruktionsformat som är 14 bitar långt. Totalt innehåller detta EEPROM 1024 rader (1 kiloord). Programräknaren används för att peka ut rader i minnet som vanligt. Det finns en (primitiv) möjlighet för programräknaren att lagra sitt värde på en åtta nivåer djup stack. Denna stack fungerar enligt principen *last in-first out*, som en tallriksfjäder i en matbespisning. Övriga stackmanipulationer vi är vana vid kan inte genomföras.

Den utpekade instruktionen leds via en 14-bitars *Program Bus* till instruktionsregister där olika delar av processorns hårdvara ges tillfälle att studera den. En del av instruktionsordet går till *Instruction Decode & Control*, en annan del om fem bitar kan fungera som operand i de fall ett register skall pekas ut.

Internt finns 36 stycken register à 8 bitar, dvs 36 byte totalt läs- och skrivminne. Detta minne är organiserat i vad tillverkaren kallar en *Register File*. Flera tillverkare använder liknande namn då registren är generella och kan adresseras individuellt. Av instruktionsordet kan fem bitar användas för att peka ut något av dessa register.

Processorn är utsvulten på adresseringsmoder och den enda ”tyngre” adresseringsmoden är indirekt adressering. I arkitekturen ser vi att ett register, *FSR*, kan användas för att peka ut ett annat register i RAM-minnet. Det är omöjligt att peka ut kod i programminnet.

Det register vi skulle kalla ackumulator kallar tillverkaren ”W”, för *Working Register*.

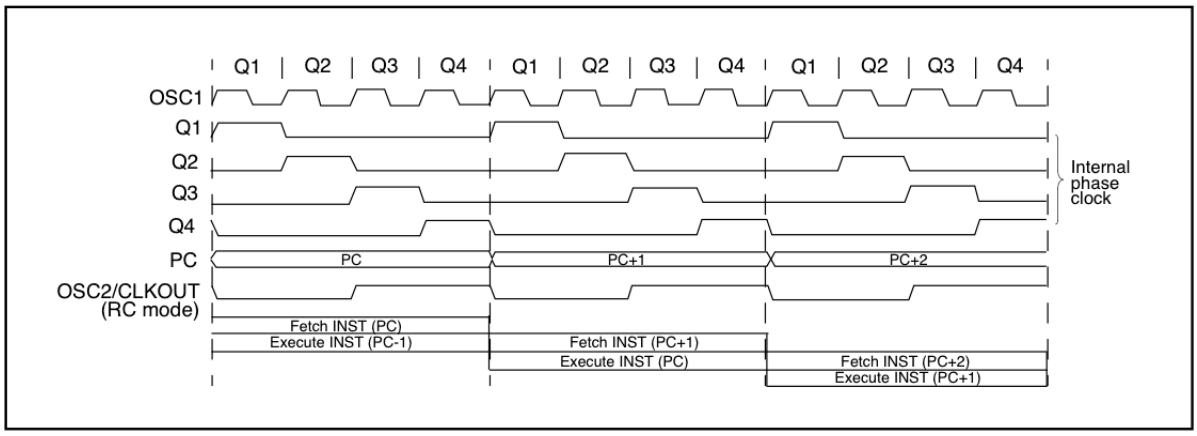
En viktig skillnad mellan denna arkitekturer och den vi känner från MMM, är förekomsten av flera bussar. Man skiljer på data- och programbuss till exempel. Även på andra ställen i arkitekturen används ”specialbussar” för att möjliggöra stor parallellism exekveringsfasen.

13.4 Instruktionsexekvering

Varje instruktion tar fyra klockcykler i anspråk. Detta är möjligt genom att ha enkla instruktionsformat och en genomtänkt arkitektur — exvis de olika interna bussarna. Det enkla instruktionsformatet möjliggör en enkel avkodning av instruktionerna. Så enkel att mikrokod inte behövs. Istället används ren digitalteknik med sekvensnät av klassisk modell för avkodningen. För att möjliggöra detta delas alla instruktioner upp i fyra faser $Q_1, .., Q_4$ enligt figuren nedan.

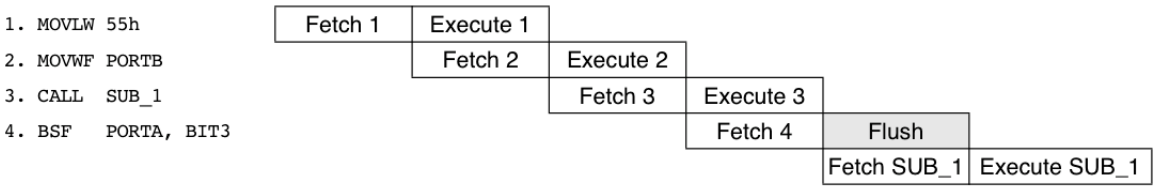
Det är inte dokumenterat exakt vad som händer i respektive fas men kan förstå att närvaron av sammanlagt åtta stycken flanker (upp- och ner på varje Q_i) ger möjlighet till många ”mikroord” för styrning av olika delar av arkitekturen.

FIGURE 3-2: CLOCK/INSTRUCTION CYCLE



Ett sätt att snabba upp instruktionsexekveringen är, som vi vet, att tillgripa *pipeline*. Emellertid kräver detta rätt mycket hårdvara varför denna processor använder den betydligt enklare tekniken med överlappande hämta- och utförfaser (*Overlapping fetch/execute*). Detta innebär (enligt figuren nedan) att en instruktions exekveringsfas överlappar i tid med nästa instruktions hämtfas. I bästa fall kan denna teknik fördubbla processorns hastighet.

EXAMPLE 3-1: INSTRUCTION PIPELINE FLOW



All instructions are single cycle, except for any program branches. These take two cycles since the fetch instruction is “flushed” from the pipeline while the new instruction is being fetched and then executed.

”I bästa fall” innebär här att programmet är skrivet utan några som helst hopp. Sannolikt går man inte iland med att skriva ett program helt utan hoppinstruktioner och detta medför att exekveringshastigheten inte kan bli den dubbla. Problemet med hopp är att: Parallellt med att hoppet utförs (*Execute 3*), läses nästa instruktion in (*Fetch 4*). Den senare instruktionen skall inte utföras eftersom den ligger efter hopp-raden. Åtgärden blir i detta fall att kasta bort (”flusha”) den hämtade instruktionen och genomföra en ny hämtfas på rätt adress (dvs dit hoppet skedde).

En möjlig optimering av situationen är ibland möjlig men tillverkarna har medvetet uteslutit denna: Om det är möjligt ur programfunktionssynpunkt kan hoppinstruktionen och den innan denna byta plats. Detta medför att hoppet hämtas och samtidigt med att hoppet utförs hämtas nästa instruktion vilken kommer att utföras samtidigt med hämtfasen på det nya stället. På så sätt förlorar man inte någon cykel i exekveringen. Men tillverkaren har tyvärr(?) gjort det omöjligt för oss att vara så smidiga.

13.5 Instruktionsuppsättning

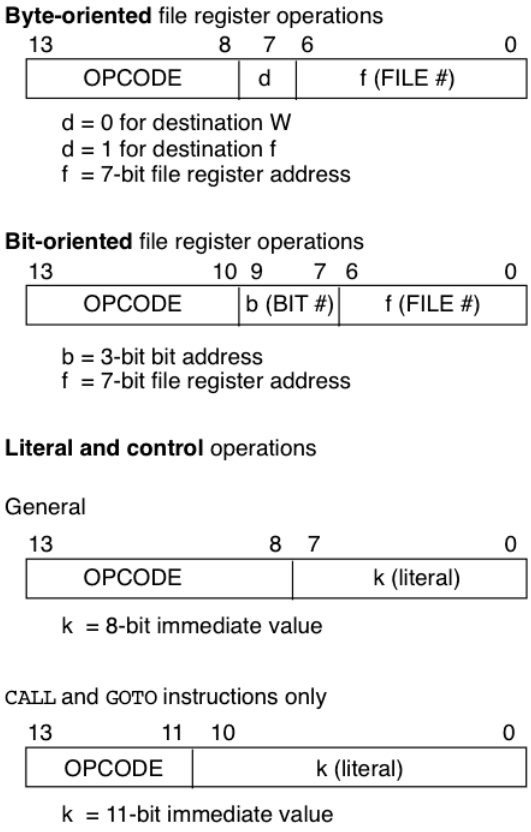
Trots den egendomliga arkitekturen är processorn förvånansvärt kraftfull och de assemblerinstruktioner den använder ger en mycket kompakt kod.

Instruktionslistan är i stora drag självförklarande. Vi hittar kända instruktionstyper: aritmetiska, logiska, subrutin (`CALL/RETURN`) och test-instruktioner, vilka motsvarar `CMP` på 68000. Uppsättningen är minimal i flera avseenden och det tar ett tag att vänja om sig efter att ha blivit van med 68000:s generösa instruktionsuppsättning.

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected
ADDWF	**f, d**	Add W and f	1	00	0111	dfff	ffff	C,DC,Z
ANDWF	**f, d**	AND W with f	1	00	0101	dfff	ffff	Z
CLRF	**f**	Clear f	1	00	0001	lfff	ffff	Z
CLRW	**-**	Clear W	1	00	0001	0000	0011	Z
COMF	**f, d**	Complement f	1	00	1001	dfff	ffff	Z
DECF	**f, d**	Decrement f	1	00	0011	dfff	ffff	Z
DECFSZ	**f, d**	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff	None
INCF	**f, d**	Increment f	1	00	1010	dfff	ffff	Z
INCFSZ	**f, d**	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff	None
IORWF	**f, d**	Inclusive OR W with f	1	00	0100	dfff	ffff	Z
MOVF	**f, d**	Move f	1	00	1000	dfff	ffff	Z
MOVWF	**f**	Move W to f	1	00	0000	lfff	ffff	None
NOP	**-**	No Operation	1	00	0000	0xx0	0000	None
RLF	**f, d**	Rotate left f through carry	1	00	1101	dfff	ffff	C
RRF	**f, d**	Rotate right f through carry	1	00	1100	dfff	ffff	C
SUBWF	**f, d**	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z
SWAPF	**f, d**	Swap nibbles in f	1	00	1110	dfff	ffff	None
XORWF	**f, d**	Exclusive OR W with f	1	00	0110	dfff	ffff	Z
BIT-ORIENTED FILE REGISTER OPERATIONS								
BCF	**f, b**	Bit Clear f	1	01	00bb	bfff	ffff	None
BSF	**f, b**	Bit Set f	1	01	01bb	bfff	ffff	None
BTFSC	**f, b**	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff	None
BTFSS	**f, b**	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff	None
LITERAL AND CONTROL OPERATIONS								
ADDLW	**k**	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z
ANDLW	**k**	AND literal with W	1	11	1001	kkkk	kkkk	Z
CALL	**k**	Call subroutine	2	10	0kkk	kkkk	kkkk	
CLRWDT	**-**	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$
GOTO	**k**	Go to address	2	10	1kkk	kkkk	kkkk	None
IORLW	**k**	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z
MOVLW	**k**	Move literal to W	1	11	00xx	kkkk	kkkk	None
RETFIE	**-**	Return from interrupt	2	00	0000	0000	1001	None
RETLW	**k**	Return with literal in W	2	11	01xx	kkkk	kkkk	None
RETURN	**-**	Return from subroutine	2	00	0000	0000	1000	None
SLEEP	**-**	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$
SUBLW	**k**	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z
XORLW	**k**	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z

Vi ser speciellt att vissa instruktioner kan ta en eller två *cycles* i anspråk. En *cycle* motsvarar fyra klockpulser enligt ovan. Hopp-instruktionerna och de kombinerade ”testa och hoppa”-instruktionerna kan ta två cykler av anledningar vi nu känner till.

Det enhetliga instruktionsformatet är:



13.6 Inbyggd hårdvara

För att göra så god tjänst som möjligt är mikrokontrollrar försedda med mycket hårdvara ombord vilken vanligen återfinns utanför en processor.

I/O-portar Typiskt sitter dessa mikrokontrollrar och skall skapa styrsignaler till någon yttre apparat (diskmaskin/industriprocess m.m.). I laborationerna använde vi en PIA, *Peripheral Interface Adapter*, för att hantera yttre signaler från tryckknappar och pipa ljud i en hörsnäcka. Det är inte förvånande att en mikrokontroller har denna funktion inbyggd.

Just denna mikrokontroller har två sådana portar med skillnaden att bara en av dem är ”komplett” med åtta bitars bredd. Den andra har bara fem bitar. På databladets förstasida och processorns blockschema ser vi dessa portar som RA_i och RB_i. För att maximera kretsens flexibilitet är inte dessa portar identiska utan är försedda med en del ”praktiska finesser”. Här behandlas bara A-portarna. B-portarna är *nästan* identiska med port A:s bitar 3–1. Studera schemana för port A nedan:

FIGURE 5-1: BLOCK DIAGRAM OF PINS RA3:RA0

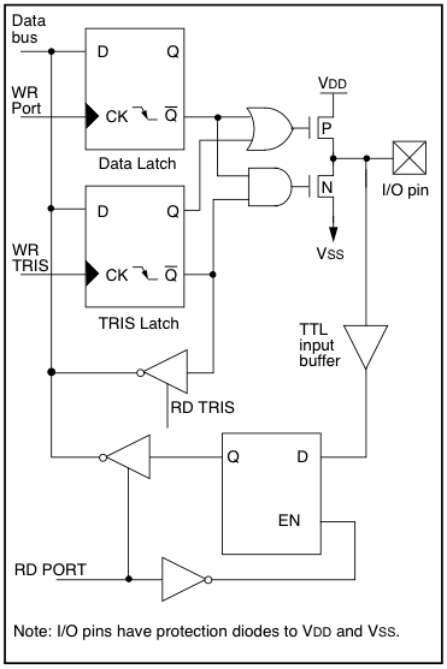
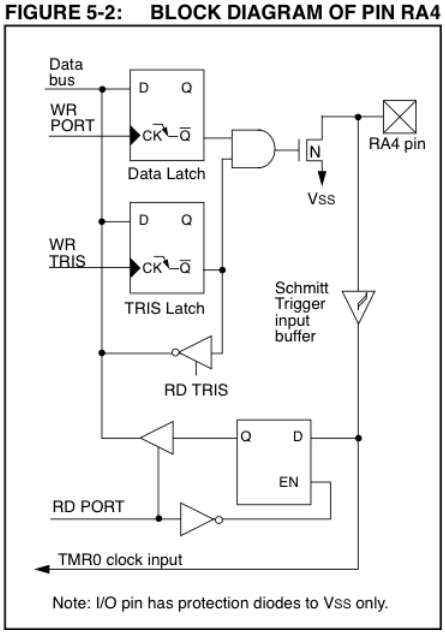


FIGURE 5-2: BLOCK DIAGRAM OF PIN RA4



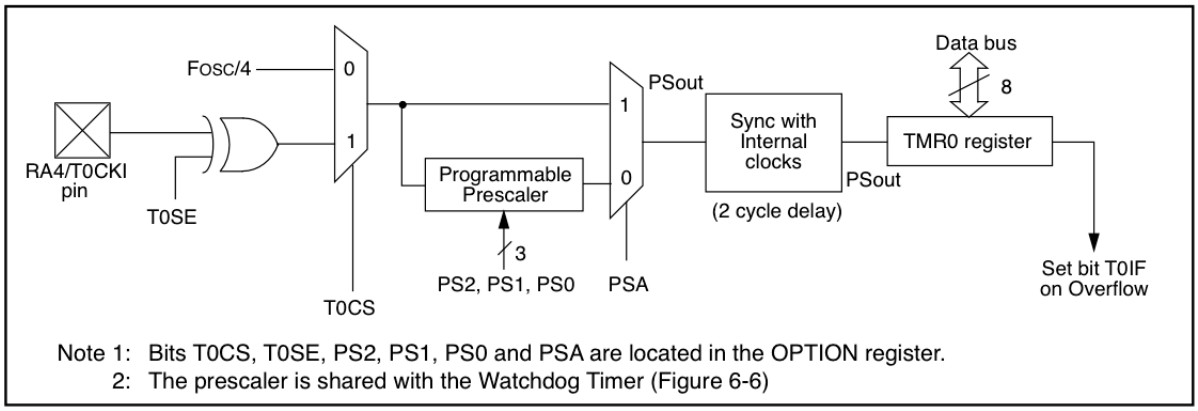
Portarna RA3–RA0 fungerar som man kan förvänta sig. Varje enskild bit kan ställas som in- eller utgång, precis som på PIA:n. Varje enskild bit kan — om den är ställd som utgång — skapa en hög eller låg utsignal på sin pinne.

Port A:s bit 4 är däremot speciell. Som vi ser i schemat till höger finns det ingen koppling till V_{DD} dvs positiv matningsspänning. Denna bit kan bara ge en låg utsignal, genom att kortsluta RA4-pinnen till V_{SS}, noll volt, jord. Ett vanligt misstag är att tro att denna bit är trasig bara för att den inte kan ge en hög utsignal. Knepet är att använda ett yttre s.k. *pull-upmotstånd* för att dra biten hög när den själv inte vill dra den låg. Jämför med digitalteknikens *wire-and* och *wire-or*.

Timer En *timer* utgörs av ett register som automatiskt räknas upp till noll (nerräknare finns också ibland) och då skapar ett avbrott. Det finns en hel uppsjö av möjligheter att få timern att räkna:

- I takt med varje instruktion, eller
- i takt med yttre signal på RA4-pinnen, eller
- via en *prescaler*.

FIGURE 6-1: TMR0 BLOCK DIAGRAM



Med $F_{OSC}/4$ som klockkälla sker en räkning vid varje instruktion. Med RA4/T0CKI, (*Timer 0, Clock Input*), kan räkning ske med yttre signal. Båda dessa klockpulser kan fås att passera en *prescaler* med vilket menas en ytterligare räknare som i sin tur måste räkna till noll för att en puls ska ges till timer-räknaren. På så sätt kan timern fås att räkna 2, 4, 8, 16, 32, 64, 128 eller 256 gånger långsammare än vad som annars skulle varit fallet.

Med T0SE kan man välja om uppåtgående eller nedåtgående flank skall användas på signalen in på RA4.

Om RA4 används som klockkälla kallas moden inte timer utan *counter*, räknare.

Watch Dog Timer, WDT, Vakthund I kretsen finns en separat oscillator som alltid snurrar, oavsett närvaron av program eller yttre klocksignal eller till och med om processorn gått i dvala genom instruktionen `SLEEP`. Denna oscillator påverkar en *Watch Dog Timer*, en räknare som ger processorn en RESET-signal om den räknat till noll. Oscillatorns noggrannhet är begränsad och kan variera med omgivningstemperatur och matningsspänning. RESET-signalen kommer att tvinga hela processorn till omstart från programmets rad 0.

Syftet är att identifiera om processorn eller programmet gått överstyr och inte är pålitligt längre. Metoden är att tvinga programmet att då och då ”sparka till vakthunden så att den inte somnar”. Så länge programmet gör vad det skall och håller vakthunden vaken sker inget, men om programmet slutat fungera av någon anledning kommer WDT att hinna räkna till noll och en RESET skapas.

Med tanke på att en sådan här mikrokontroller oftast sitter oåtkomligt placerad inuti andra apparater är det bra om den själv kan starta om sig på det här sättet om det skulle behövas.

Klockalternativ För att vara flexibel, återigen, kan processorn erhålla sin klocksignal från flera olika källor. Vilken man väljer beror på önskad klockfrekvens, noggrannhet och pris. Processorn innehåller en oscillatorförstärkare som kan användas i de tre första fallen nedan. Beroende på vilket alternativ man väljer måste denna förstärkare programmeras

(sker genom att vid programmering av kretsen påverka bitar i ett konfigurationsregister) så att försärkning och fasvridning blir lämplig. Processorn stödjer sammanlagt fyra alternativa klockkonfigurationer, i fallande kostnadsordning:

- Kristalloscillator. Med en yttre kristall (piezoelektriskt element med mekanisk vibration samma som önskad klockfrekvens) erhålls hög frekvenstabilitet och noggrannhet. De flesta kristallers noggrannhet är i storleksordningen 10^{-6}, dvs en kristall på 1 MHz håller sin frekvens inom några tiotal hertz. Frekvensen ändras mycket litet om omgivningstemperaturen ändras.[6]
- Resonator. En keramisk resonator är billigare än en kristall och har större onoggrannhet (typ 10^{-2}). Normalt är deras frekvens under 1 MHz men på senare tid har resonatorer för högre frekvens dykt upp på marknaden.
- RC-länk. Men en kondensator och ett motstånd kan en oscillatorfrekvens om cirka en megahertz uppnås. Tillverkningstolernasen hos en normal kondensator är 10 % varför frekvensen blir i stort ”blir vad den blir”. Både motstånd och kondensatorer påverkas mycket av yttertemperaturen varför någon egentlig förbättring inte sker om dyrare kondensatorer används.
- Yttre TTL-signal. Om man redan har den önskade klockfrekvensen i omgivningen kan denna användas direkt.

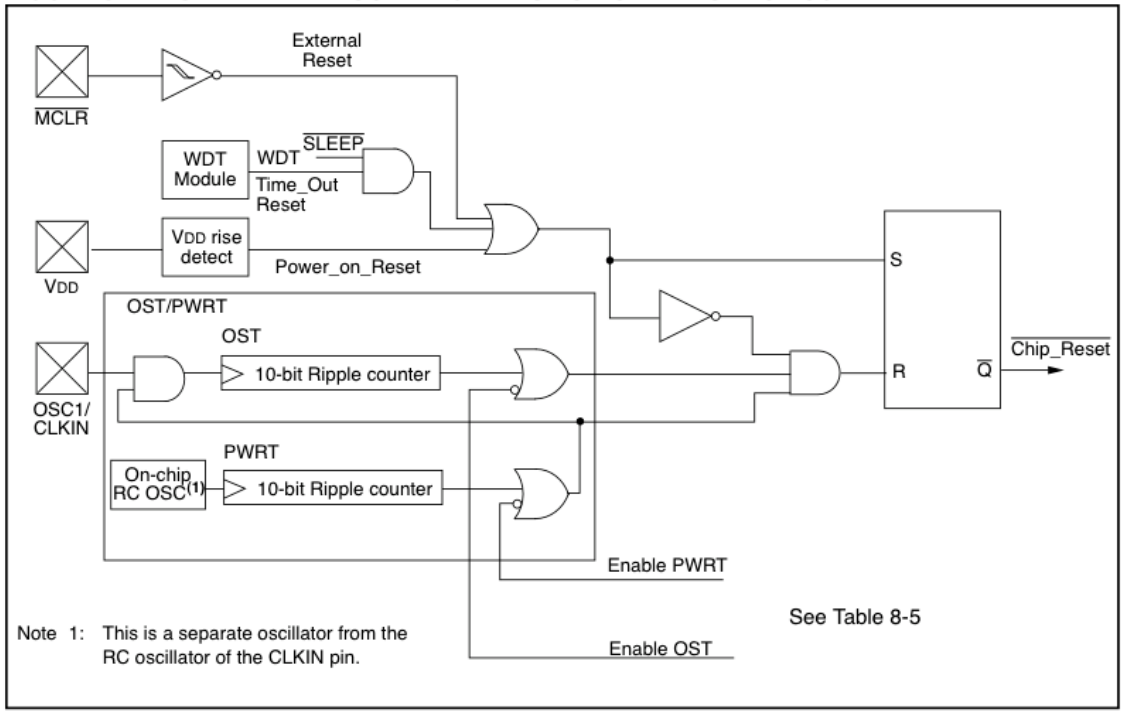
13.7 Reset

För att processorn ska börja exekvera instruktioner vet vi att vissa register måste nollställas. Om denna nollställning misslyckas kommer processorn i ett ”limbo”-tillstånd som den normalt inte själv kan ta sig ur. Mikrokontrollrar har därför ofta ett komplicerat startsystem som skall garantera att den inte hänger sig vid spänningspåslag. I fallet med PIC:en sker följande vid spänningspåslag:

1. När kretsen känner av en spänningsuppgång (strax under 2 volt) sätts en klarsignal till den senare logiken med betydelsen ”jag har märkt att spänningen överstiger min lägsta nivå”.
2. Det tillhör ”god ton” hos användaren att för säkerhets skull också generera en yttre reset genom att låta signalen $\overline{MCLR}$ vara låg ett antal millisekunder efter att spänningen nått sitt normala värde.
3. För att säkerställa att resetten ovan hunnit bli färdig används en separat oscillator ombord vilken måste räkna till 1024 innan den ger sitt klartecken till resten av initieringen.
4. Dessutom måste även den normala klockoscillatorn också hunnit stabilisera sig och även den genererat 1024 klockpulser innan dess klarsignal kommer.

[6] Om riktigt hög noggrannhet önskas kan kristallen läggas i en s.k. kristallugn som håller en reglerad temperatur. Temperaturen hålls något varmare än rumstemperatur då det är lättare att reglera temperaturen uppåt än nedåt.

FIGURE 8-7: SIMPLIFIED BLOCK DIAGRAM OF ON-CHIP RESET CIRCUIT



När samtliga klarsignaler enligt ovan (förutom möjligen extern reset) kommit, vilket tar cirka 80 millisekunder, sker slutligen reset av hela processorn, dvs vissa register förladdas med sina värden och instruktionshämtningen påbörjas (från rad 0).

Brown-Out Protection Om spänningen försvinner helt kallas detta en *black-out*. Efter en black-out kommer processorn vid spänningspåslaget att initieras som ovan, med en fullständig reset. För inbyggda system är det inte ovanligt att den omgivande elektriska störmiljön är så hög att variationer och svackor i matningsspänningen förekommer. Svackor som är så stora att vissa av processorns interna register får för lite spänning för att hålla sin information men som samtidigt inte ger så låg spänning att resetkretsarna tar itu med problemet. En sådan svacka kan vara bråkdelen av någon millisekund lång och ändå få förödande konsekvenser för programexekveringen.

Eftersom processorns register inte är pålitliga längre måste en full reset genomföras. För detta finns yttre kretsar att ansluta, men i PIC-fallet kan motsvarande funktioner erhållas med några få yttre komponenter. Det är också möjligt att processorn (inte denna dock) i ett register meddelar programmet att nuvarande uppstart beror på en brown-out och inte en black-out. Detta ger processorn möjlighet att vidta speciella åtgärder.

14 Operativsystem

Begrepp i denna föreläsning: *Systemanrop, bibliotek, kärna, BIOS, drivrutin, tidsdelning, multitasking (mt), timerinterrupt, processköer, körbar kö, blockerad kö, kooperativ mt, preemptiv mt, virtuellt minne/minneshantering, swap, swappning, tröskning, sidstorlek, sidtyper (aktiva, inaktiva mm), utbytesalgoritmer, sidfel, demand-zero paging, page-fault, process control block (PCB), minnesskydd och praktiska övningar.*

14.1 Inledning

Vi har hittills nämnt ordet operativsystem i lösa termer utan någon egentlig definition. Det har förutsatts att vi har åtminstone något hum om vad det är och någon mer ingående kunskap har inte behövts. Det är dock synd att inte beskriva operativsystem när vi i kursen gått den långa vägen från de absoluta grunderna med digitala grindar och mikroprogrammering ända till bland annat virtuellt minne. Operativsystemet knyter ihop alla delar till en användbar enhet.

Jag tänker inte beskriva något speciellt operativsystem utan koncentrera mig på generella principer på funktioner som kan ingå i ett operativsystem. Det finns mängder med information jag utelämnar men många grundtankar är gemensamma i dagens operativsystem vare sig det gäller Windows i någon inkarnation eller Unix i någon speciell smak. Det fria unixoperativsystemet FreeBSD används som underlag för några exempel bara för att *jag* gärna använder det.

Behövs det operativsystem då? Bättre är att fråga sig *när* ett operativsystem behövs. I mindre mikroprocessorstyrda apparater som mikrovågsugnar, tvättmaskiner och dylikt, klarar man sig säkert bra utan operativsystem. Det är svårt att göra operativsystem små, det krävs rätt stora mängder minne för att lagra det och OS:et behöver ofta en hel del minne för att kunna köras också. I kostnadskänsliga omgivningar kan det vara helt omöjligt att använda ett operativsystem av dessa skäl.

Om man lämnar de allra enklaste apparaterna bakom sig behöver man inte gå upp mycket i komplexitet förrän fördelarna med ett operativsystem visar sig. Ett exempel kan vara ett mätinsamlingssystem som ska känna av ett antal elektriska signaler, sammanställa dessa, kanske utföra någon matematisk operation på de insamlade värdena och sedan skicka resultatet vidare via nätverk. Apparaten ska också kunna styras för att inte alltid mäta alla signaler. Man kanske vill kunna stänga av vissa givare eller nåt. Här kan ett operativsystem underlätta. Både den slutliga funktionen och utvecklingstiden kan tjäna på att låta ett operativsystem vara inblandat. Vid slutet av kapitlet hoppas jag att det ska gå att identifiera vilka funktioner ett operativsystem kan ha i detta hypotetiska mätsystem.

Av det som skrivits ovan skall man också dra slutsatsen att ett operativsystem inte nödvändigtvis bara är det som finns i en vanlig bordsdator, d.v.s. Windows eller nåt annat, utan i många fall återfinns i inbyggda system som inte ens har ett tangentbord anslutet till sig.

14.2 Systemanrop och bibliotek

Det vi vanligen ser av ett operativsystem är bara ytan av ett mycket komplicerat programpaket som både ska tjäna användaren vad gäller användarvänlighet och utseende men också tjäna datorns hårdvara för att alla delar av datorn ska fungera ihop och på ett effektivt sätt få saker gjorda.

För programmeraren tillhandahåller operativsystemet en stor uppsättning funktioner och rutiner som gör programmerarjobbet enklare. I FreeBSD finns något hundratal sådana funktioner och man når dem genom att först ange funktionens nummer och sedan göra systemanrop nummer `0x80`. Ett assemblerprogram som skriver texten ”`Hello, World!`” kan då för FreeBSD se ut så här:

```
section  .text
  global _start                      ;must be declared for linker (ld)
  msg    db   ''Hello, world!'',0xa ;our dear string
  len    equ  $ - msg               ;length of our dear string

  _syscall:
          int 0x80                ;system call
   _start:                        ;tell linker entry point
          push dword len          ;message length
          push dword msg          ;message to write
          push dword 1            ;file descriptor (stdout)
          mov  eax,0x4            ;system call number (sys_write)
          call _syscall           ;call kernel
          add  esp,12             ;clean stack (3 arguments * 4 byte)
          push dword 0            ;exit code
          mov  eax,0x1            ;system call number (sys_exit)
          call _syscall           ;call kernel
                                  ;we do not return from sys_exit,
                                  ;there's no need to clean stack
```

Dessa rutiner använder funktioner i kärnan, en del av operativsystemet som alltid måste ligga i minnet. Kärnan innehåller i lösa termer förutom stöd för systemanropen även allt som behövs för att ”hålla ordning” på datorn och dess användare.

Att kunna komma åt funktioner i kärnan på det här sättet är mycket praktiskt och bidrar naturligtvis till att programmera mer effektivt. Rutinerna i kärnan är förhoppningsvis väl testade och pålitliga och eftersom man anropar kärnan direkt, blir programmet även så snabbt som det överhuvudtaget är möjligt.

Ett problem kan dock uppstå om konstruktörerna av operativsystemet väljer att ändra i dessa rutiner. Man kan då komma i den situationen att ens program inte fungerar längre. För att undvika denna sorts överraskningar används ofta ett extra ”lager” med

isolerande kod mellan användarprogrammens anrop och deras exekvering i kärnan. För programmering i C finns ett standardbibliotek, `libc`, som just innehåller ett standardiserat utseende utåt. Om en ändring i kärnan blir nödvändig räcker det då att också ändra i "ena sidan" av `libc` och ändringar i användarprogrammen kan undvikas.

14.3 BIOS

Vi ser hur OS:et kan tänkas ligga mellan oss och hårdvaran. Speciellt i PC-fallet förlitar sig operativsystemet på ett program som alltid ligger i hårdvaran: BIOS (*Basic Input-Output System*). Ett BIOS är ett program med rutiner för att kunna prata med hårdvaran. Här finns rutiner för att bland annat kunna läsa in från tangentbordet, skriva ut till skärmen, skriva till parallellporten och serieporten och även för att kunna läsa/skriva mot hårddisken. Det är BIOS som vid datorns spänningssättning räknar upp minnet (RAM) i systemet och känner av vilken hårddisk som är ansluten[1]. Det är också i BIOS som det beryktade programmet SETUP befinner sig, varmed man kan välja egna inställningar för hårdvaran, hårddisk med mera.

[1] Och meddelar/piper om minne saknas eller grafikkortet är urtaget. Prova själva!

Vissa operativsystem har egna program, *drivrutiner*, och bibliotek med funktioner som ersätter de i BIOS. Bland de vanligare är till exempel drivrutiner för speciell hårdvara. BIOS-funktionerna är så grundläggande att de inte har stöd för annat än den mest enkla hårdvaran. Grafikkort är exempel på hårdvara som behöver egen programvara för att komma till sin fulla rätt. Men även, bland annat, hårddiskar tjänar på egna drivrutiner. Lägg märke till att dessa drivrutiner förutsätter närvaron av operativsystemet för att kunna fungera. Innan operativsystemet är helt startat efter spänningspåslag måste BIOS användas.

14.4 Tidsdelning och multitasking

I sin enklaste form kan operativsystemet vara avsett för en användare och bara kunna köra ett program åt gången. MS-DOS[2] är exempel på ett sånt. Med detta tidiga operativsystem kunde en användare mata in kommandon till datorn och få resultat på en enkel skärm med 80x25 tecken. Det kommando man matat in eller det program som körde var det som gällde för processorn. En tydlig brist, eftersom man till exempel var tvungen att vänta på att en utskrift blev klar tills man kunde mata in nästa kommando.

Det hade varit mycket lättare i det här fallet om utskriften kunde ske samtidigt, i *bakgrunden*, som användaren fortsatte med annat. Det infördes ett rudimentärt stöd för detta i MS-DOS som i princip gick ut på att en avbrottsbegäran till processorn fick denna att kasta ut det den höll på med och ta in utskriftsprogrammet en stund och sedan byta tillbaka igen.

Denna manöver, kallad tidsdelning eller *time-sharing*, innebär numer att processorn får ett avbrott ungefär 10-100 gånger per sekund. Vid avbrottet anropas operativsystemet, som väljer nästa program som ska köras, och överlämnar detta program till processorn. Och så kör detta program en stund, tills nästa avbrott sker. På så sätt får alla program köra under sin tidslucka vid processorn och det upplevs som om alla program körs samtidigt, fastän de egentligen bara körs en snutt i taget.

14.5 Processköer

Programmeringstekniskt lägger operativsystemet alla program som ska köras i en kö, *körbar*-kön, som sedan betas av allt eftersom avbrotten kommer:

[2] I Windows, välj programmet MS-DOS-prompt för det äkta känslan...

Om användaren har igång sin ordbehandlare men gått på lunch är det ju rätt onödigt att ordbehandlaren ska få tillgång till processorn. Vi vet med säkerhet att ingen tangentnedtryckning kan ske. Överhuvudtaget är det onödigt att ordbehandlaren ska få processortid om ingen tangentnedtryckning[3] är gjord.

Lösningen är att införa en *blockerad*-kö. En kö där program som väntar på någon yttre händelse för att kunna köra vidare. Till exempel läggs det program som väntar på sin tangentnedtryckning i denna kö.

Om operativsystemet senare lägger märke till att en tangentnedtryckning skett flyttas programmet över till körbar-kön i väntan på att få processortid.

Det här kanske låter exotiskt men är det inte. Dessa funktioner finns i alla hyggligt moderna operativsystem. Processerna åker in och ut ur dessa (och flera andra) köer nästan jämt. Exemplet med tangentnedtryckning är enkelt och uppenbart, men samma resonemang gäller vid varje åtkomst av hårddisken, serieportar, mus, eller vilken annan hårdvara som helst. Så fort en process inte kan göra mer av sin tidslucka ska den för effektivitetens skull lämna processorn och ge plats för annan aktivitet.

[3] Med tangentnedtryckning menas här även mus-klickande eller något annat som ger förändring i dokumentet eller programmet. Läs med förstånd!

Den här mekanismen kan skötas mer eller mindre av operativsystemet och man brukar skilja på olika sorters multitasking, *mt*:

- *Non-preemptive* mt (*kooperativ* mt). Här är det upp till varje process att lämna processorn när den inte behöver den längre, till exempel för att processen väntar på något. Processen meddelar operativsystemet att den är klar och OS:et träder in och genomför processbytet. Om processen går fel och inte lämnar processorn hänger sig hela systemet, och måste startas om.
- *Preemptiv* mt. Här bestämmer operativsystemet spelreglerna. En process kan kastas ut när som helst, dvs utan att processen är förberedd på det. Detta sker antingen när
  - den tilldelade tidsluckan är konsumerad, eller
  - när processen anropat någon funktion i operativsystemet (till exempel att skriva en bokstav på skärmen). I det senare fallet får OS:et avgöra om processen bör blockas eller inte.
  - processen får ett avbrott, eller
  - processen frivilligt släpper processorn.

Den del av operativsystemet som ständigt ligger i minnet, den s.k. kärnan, *kernel*n, brukar inte kunna avbrytas (*non-preemptive kernel*). Det är därför viktigt att dess avbrottsrutiner är snabba, och operativsystemet fungerar då ungefär som en stor växel: Antag att process A körs och process B väntar på tangentnedtryckning. En tangentnedtryckning resulterar i ett interrupt. Process A avbryts. Kärnan körs, noterar vilken tangent det var, vilken process som ska ha den och går ur interruptet. Process A fortsätter köra. Nu är förutsättningarna uppfyllda för att flytta över process B från *blocked*- till *ready*-kön. När systemet får nästa *timeravbrott* (10–100 gånger per sekund) tar kärnan över kontrollen igen och flyttar över process B till ready-kön. Sedan kör kärnan nästa process i ready-kön med hänsyn till processernas eventuella prioritet.

Det är viktigt att notera att *kärnan stänger av vidare avbrott då den betjänar ett interrupt*. När det pågående avbrottet har kört klart slås avbrott på igen. Med ett stort och stadigt inflöde av avbrott kan kärnan behöva stänga av avbrott under större delen av tiden. Men i och med att avstängda avbrott betyder att kärnan inte kan svara på ytterligare avbrott kommer systemet då att upplevas som långsamt. Av denna anledning konstrueras kärnan att bara innehålla det mest nödvändiga för att betjäna avbrotten. När avbrottet kommer förbereder kärnan allt som behövs för att något annat program eller systemprocess ska kunna slutföra uppgiften, medan kärnan så fort som möjligt hoppar ur avbrottsrutinen och möjliggör ytterligare avbrott. Uppgiften slutförs sedan som vilket program som helst, dvs det ligger i någon av köerna som vanligt, även om det är kärnan själv som i detta fall startat programmet.

När vi pratar om att processer flyttas mellan olika köer ska det inte tolkas som att hela programmet skyfflas runt. Programmet ligger still i minnet och bara pekare, processens *handtag*, flyttas. Det är lätt att förstå att det är väldigt komplicerat förlopp som ska utföras för att det hela ska fungera (vad händer exempelvis om process B ovan läggs sist i kön när den kommer till *ready*-tillståndet?) och för att hela maskineriet ska vara

effektivt krävs en bra arbetsfördelare, schemaläggare, *scheduler*, hos operativsystemet. Det är schemaläggarens uppgift att avgöra vilken process som ska köras vid varje tillfälle.

Det finns som märks en omfattande nomenklatur runt detta och följande begrepp är bra att kunna:

- Process (*process, task*). En process är lika med ett program för vår del. Man kan programmera ett program att bestå av flera processer men det spelar ingen roll i det vi diskuterar här.
- Processbyte (*context switch*) innebär att processorns inre tillstånd undanlagras och en ny process' inre tillstånd laddas i processorn.
- Körbar-kö (*ready queue*), aktiv kö. Den kö där processer som är klara att köras ligger. De väntar på sin tur vid processorn. Programmen i denna kö betas av allteftersom, då systemet får sina timeravbrott och genomför sina processbyten.
- Blockad-kö (*blocked queue*), inaktiv kö. Den kö där program som väntar på någon händelse ligger. Om händelsen inträffar (DMA-överföring klar eller tangentbordsnedtryckning gjord, till exempel) ska processen flyttas över till körbar-kön.

Man brukar tala om att en process befinner sig i något av tillstånden *ready*, *blocked* eller *running*. Där running uppenbart betyder att den processen just nu får processortid.

Ett system som kan köra flera program samtidigt, enligt denna modell eller annan, kallas för *multitasking* eller ett *multiprogrammerat* system.

14.6 Virtuellt minne/minneshantering

Vi har tidigare sett hur virtuellt minne fungerar ur hårdvarans synvinkel. Medan hårdvaran ger möjlighet till virtuell minneshantering, deskriptorer m.m. *administreras* allt minne av operativsystemet. Det är operativsystemet som bland annat tilldelar en process minne om den behöver det och övervakar tillgänglig minnesmängd.

Exempel

Antag att följande minnesmapp gäller ett nystartat system, dvs bara [nödvändiga delar av] operativsystemet är i minnet:

Vi vill köra programmen A, B och C. Låt oss göra det! Program A laddas in i minnet och körs i några millisekunder, sedan kastar schemaläggaren ut det (vi vet nu att det bara är pekare som flyttas), och laddar program B istället. Program B körs, plötsligt vill det vill ha en tangentnedtryckning. Schemaläggaren märker att ingen tangent är nedtryckt och flyttar över processen till kön över inaktiva processer. Process C står näst på tur — men tyvärr! — processen är för stor för det fysiska minnet. Operativsystemet måste nu göra plats genom att flytta ut någon annan process till hårddisken. Det är synd, för hårddiskar är ofantligt mycket långsammare än minne, men, måste man så måste man. Proceduren att flytta ut minne till hårddisk för att göra plats i primärminnet kallas att swappa[4], *swapping*. Nu finns det plats i primärminnet och processen C laddas och körs.

■

Har vi otur måste vårt system swappa ofta. Har vi ännu mer otur ägnar sig systemet mer åt att swappa data in och ut än att faktiskt köra programmet och situationen kallas tröskning, *thrashing*. Lösningen är förstås att skaffa mer primärminne om möjligt.

Den mekanism som hanterar minnet i ett datorsystem är alltså väsentlig ur prestandasynpunkt. Med hundratals samtidiga processer i minnet[5] måste operativsystemet använda flera knep för att hantera den tillgängliga minnesmängden på ett effektivt sätt. Som exempel ska vi snegla lite på hur FreeBSD gör detta.

[4]På windows kallas ytan på hårddisken den swappar till för *växlingsfil*. Unix har i allmänhet en egen partition för detta.

[5]Bara operativsystemet självt kan behöva något dussin processer för att överhuvudtaget vakna!

FreeBSD placerar minnesområden i olika fack för att snabbt kunna hantera dem. Vid nystart av en dator räknar operativsystemet igenom tillgängligt minne och skapar en lista med referenser till allt fysiskt minne. Observera att vi hanterar pekare hela tiden, det är inget flyttande av dataareor. FreeBSD hanterar dessutom minnet i klumpar om 4 kilobyte, den s.k. sidstorleken[6], *page-size*. Dessa referenser kan sedan befinna sig i något av dessa fyra tillstånd:

- *aktiv* sidor som används av program
- *inaktiv* sidor som inte aktivt används längre men som är skrivna på och måste sparas på hårddisk innan de återanvänds.
- *cache* sidor som inte används aktivt och kan återanvändas med kort varsel. Alla initialt uppräknade sidor ligger här.
- *free* sidor som inte används av någon.

Med detta upplägg blir systemaktiviteten ungefär så här: När ett program refererar en sida på hårddisken läses den in och sidreferensen flyttas från cachen till de aktiva sidorna. Om programmet inte har använt sidan på ett tag flyttar minneshanteraren sidreferensen till inaktiv eller möjligen chace, beroende på om något skrivits på den. Detta sker för att snabbt kunna ha tillgång till mer minne åt andra, nya processer som kan komma att startas. Sidor i tillståndet inaktiv flyttas sakta över tillbaka till hårddisken och referensen flyttas samtidigt till cachen.

Minnet är en viktig resurs och systemet lägger ned en hel del tid på administrera detta så att det alltid ska finnas mer minne om så skulle behövas. En del i detta är den ovannämnda städningen av minnesutrymmet. Det finns flera strategier, *utbytesalgoritmer*, för att hantera minnet, dvs välja vilka sidor som ska användas eller frigöras. Några vanliga, och förhoppningsvis lätta att komma ihåg, är:

- Slumpartat. En enkel strategi och om vi inte har någon statistik om sidanvändningen i system skulle denna strategi duga lika bra som någon annan. Men med lite statistik kan vi välja våra sidor bättre.
- FIFO, *First-in First-out*. Det här är också en enkel strategi. Med varje process' minnesreferenser i en kö kan vi från ”huvudet” ta sidor som använts längst (first-in) och till ”svansen” lägga till nya sidor om processen skulle vilja det. Tyvärr kan det innebära att vi tar sidor som använts intensivt också. Om processen vill använda minnet igen måste det släpas in från hårddisken. Man kan komma undan detta genom att göra som FreeBSD ovan: börja med att *markera* sidorna som inaktiva men ta inte omedelbart bort dom utan var beredd att släppa in dom igen om processen skulle behöva det.
- LRU, *Least Recently Used*. Med statistik om användningen av varje sida kan vi välja att i första hand återanvända de sidor som används sällan. Och bäst är det förstås att välja de äldsta av dessa återanvändningskandidater

[6] Vi har bara berört sidindelning av minne ytligt hittills och kommer inte att studera det noggrannare heller. Det är en finess som alla moderna OS använder sig av. För att undvika någon större fragmentering av minnet hanterar systemet minne i små — lika stora — bitar, typiskt 4kb. Intels processorer har haft stöd för sidindelning sedan 80386.

14.7 Sidfel och demand-zero paging

Eftersom en sida typiskt är mycket mindre än ett program utnyttjas minnet effektivare om man bara tar in de sidor som kommer att användas hellre än att, vid programstart, dra in programmets alla sidor.

Faktum är att man till och med kan göra mindre än detta. När ett operativsystem som stödjer *demand-zero* paging startar ett program laddas *inte* motsvarande sida in från hårddisken alls! Operativsystemet skapar däremot alla andra strukturer som behövs för att köra programmet och sätter slutligen programräknaren på programmets startadress. Pang! Processorn lägger märke till att den refererade sidan inte existerar och signalerar ett avbrott av typen sidfel, *page-fault*, som operativsystemet tar hand om. Operativsystemet ser vad som hänt och hämtar nu den önskade sidan och programmet får fortsätta — ända tills programmet nästa gång referar en sida som inte finns, kanske på grund av ett hopp utanför nuvarande sida eller att programräknaren trillat av kanten på sidan. Pang igen! Och operativsystemet får hämta nästa sida. På detta sätt kommer bara de sidor som verkligen behövs, att flyttas in i minnet från hårddisken. Och med de tidigare nämnda ”städ”-mekanismerna ser operativsystemet till att minnet utnyttjas på ett effektivt sätt.

Operativssystemet håller ordning på varje process genom dess PCB, *Process Control Block*. En datastruktur med information om processen, bland annat:

- Processens nuvarande tillstånd, dvs någon av running, ready eller blocked.
- En unik identifierare för processen, PID, *Process IDentity*, en siffra som entydigt pekar ut processen.
- Processens prioritet. I vissa fall kan man prioritera någon process på bekostnad av andra processer. Man går då förbi mekanismen ovan där processerna fick vänta på sin tur. Vissa processer kan få förtur. Speciellt gäller detta vissa processer som operativsystemet i sig behöver för att fungera effektivt.
- Pekare till processens minnesområden, dvs de områden av kod, data och stack som processen använder.
- Pekare till tilldelade resurser. Om processen skriver en fil till hårddisken har den av operativet fått en handtag, ett nummer som unikt identifierar filen. All åtkomst av

filen måste ske genom detta nummer och det måste sparas mellan processbytena. Även andra resurser har handtag på motsvarande sätt, deskriptorerna till exempel.

- En area för lagring av processens inre tillstånd, då processen inte längre är i tillståndet running.

Med vår kännedom om hur Intels 286:a fungerar vet vi att det finns stöd i processorn för mycket av detta genom TASK-registret och dess TSS, *Task State Segment*. Det är en mycket omständlig operation att genomföra ett processbyte och genom att automatisera detta i processorns hårdvara vinner man mycket tid. Det stödet är ytterligare ett tydligt exempel på hur operativsystemets totala prestanda är kraftigt beroende av stöd från hårdvaran.

14.8 Minneskydd

Ett annat stöd operativsystemet behöver av hårdvaran är minnesskydd och helst privilegienivåer i processorn. Vi har sett hur minneskydd för virtuellt minne kan fungera. Speciellt viktigt är detta när ett operativsystem är inblandat. Ett operativsystem kan med hjälp av minnesskyddsmekanismen i hårdvara vara säker på att inget av det egna datat eller programmen kan bli förstörda om någon lägre priviligierad process skulle råka ur styr. I det senare fallet märker processorn att den underordnade processen försöker komma åt ”fel” minne och meddelar detta förhållande till operativsystemet som kan träda in i sin roll av vaktmästare och ta bort processen från köerna.

14.9 Praktiska övningar

Kör man någon unixvariant kan man faktiskt se och uppleva mycket av det här live. Kör programmet `top`, så visas många processer med uppgifter om deras PID, reserverade minnesmängd och prioritet. Man kan också se vilken process som körs just nu och även se processbyten. `top` visar situationen varje sekund och processbytena sker betydligt oftare än så vilket gör att man missar många detaljer. Man ser dessutom att en process kan befinna sig i många andra tillstånd än de som nämnts här. Men det kan vara intressant ändå. Väl inne i `top` ger tangenten `h` en liten hjälptext.

Kommandot `ps` visar operativsystemets olika pågående processer. I Solaris ger växeln `-fel`, eller ännu ”värre” `-feL`, mycket mer information. Kolla i manualsidan för `ps` för fler växlar.