Datorteknik Y

Daniel Wiklund

Institutionen för Systemteknik
Linköpings universitet

14 april 2005

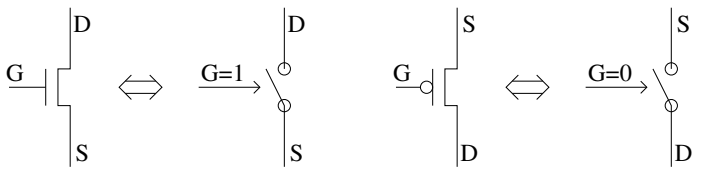
Innehåll

Kapitel 1

Introduktion

1.1 Grundläggande CMOS

Den vanligaste tekniken för att tillverka integrerade kretsar kallas för CMOS, *complementary metal oxide semiconductor*. Kretsarna byggs upp av två komplementära transistortyper, PMOS och NMOS. En modell av en ideal MOS-transistor (för digitala ändamål) består enbart av en spänningsstyrd strömbrytare. Denna spänningsstyrda strömbrytare har då tre anslutningar. Den första är *gate*, G, som styr om switchen ska vara på eller av. Denna ingång kallas ofta *styre* på svenska. De övriga två kallas *source*, S, och *drain*, D. Dessa är helt enkelt anslutningarna till de två ändarna på strömbrytaren och är i praktiken ekvivalenta.



(a) NMOS (b) PMOS

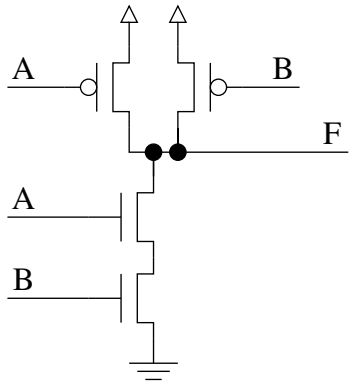
G	NMOS	PMOS
0	Av	På
1	På	Av

(c) Funktion

Figur 1.1: Ideal funktion hos MOS-transistorer

Figur 1.1(a) visar symbolen för en NMOS-transistor till vänster och den ideala modellen för transistorn till höger. För en NMOS-transistor är strömbrytaren sluten (d.v.s. den är "på") om insignalen G är 1 och öppen om G är 0. En PMOS-transistor markeras med hjälp av en cirkel på gate-anslutningen och har den motsatta funktionen jämfört med en NMOS. Figur 1.1(b), visar symbol och ideal modell för en PMOS där stömbrytaren är sluten för G=0.

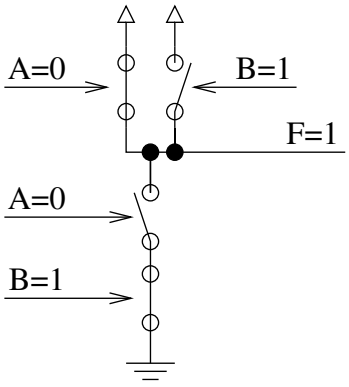
När man vill skapa en mer avancerad funktion med hjälp av MOS-transistorer använder man PMOS för att ansluta utgången till matningsspänningen och NMOS för att



(a) NAND-grind

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

(b) Funktionstabell



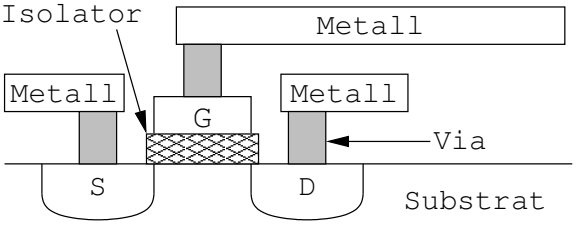
(c) Ekvivalent för A=0, B=1

Figur 1.2: NAND-funktion i CMOS

ansluta utgången till jord. Dessa två kretsdelar, vanligtvis kallade P-nätet och N-nätet, har komplementära funktioner. Detta innebär att när det finns en väg från utgången till jord genom N-nätet så finns det inte någon väg från matning till utgången via P-nätet och tvärtom. Ett exempel visas i figur 1.2 där en NAND-grind har byggts upp av fyra transistorer. En snabb analys av uppkopplingen visar att om ingången A är 0 så kommer den övre NMOS-transistorn att motsvara en öppen strömbrytare. Den vänstra PMOS-transistorn kommer däremot att vara sluten och därmed koppla utgången till matningen vilket get en hög utsignal. Insignalen B har då ingen verkan på utsignalens värde. Samma tankesätt går att använda om B är 0. Den enda fallet där utsignalen kopplas till jord och blir låg är om både A och B är 1. Då kommer PMOS-transistorerna att vara “av” och NMOS-transistorerna att vara “på”.

1.2 NMOS-transistorns funktion

En NMOS-transistor i genomskärning ser ut ungefär som figur 1.3 visar. Tillverkningen utgår ifrån ett substrat, vanligtvis en skiva av rent kisel där man har infört regelbunda orenheter i form av bor. Detta ger ett så kallat P-dopat substrat, d.v.s. där det saknas elektroner. I substratet dopas två områden kraftigt med fosfor vilket get N-dopade regioner (d.v.s ett överskott på elektroner) för source och drain. Ett lager kiseloxid läggs som isolator över mellanrummet mellan source och drain. Ovanpå denna isolator läggs sedan ett lager kraftigt dopat kisel som gate. Transistorns funktion är att när en spänning läggs på gateanslutningen så skapas ett elektriskt fält över isolatorn som attraherar elektroner i substratet och på så sätt bildar en kanal mellan source och drain. När spänningen på gaten tas bort försvinner elektronerna ut i substratet igen och



Figur 1.3: Genomskärning av MOS-transistor

kanalen försvinner. Eftersom substratet är P-dopat så kommer det inte vara möjligt att leda ström mellan de två N-dopade regionerna utan att kanalen finns.

1.3 Tillverkning av MOS-kretsar

Tillverkningen av en IC-krets är synnerligen komplicerad och därför har beskrivningen här förenklats avsevärt. Det är ett stort antal steg involverat i tillverkningen av integrerade kretsar. Efter att den dopade substratskivan har tillverkats börjar man med att lägga på ett lager av isolerande kiseloxid över hela skivan. Sedan beläggs denna oxid med ett lager fotokänsligt material som belyses på de ställen där source och drain på transistorerna ska vara. Efter belysningen etsas den del av oxiden som har blivit belyst bort. Detta gör att substratet återigen är åtkomligt på dessa ställen och kan N-dopas.

Nästa steg påbörjas sedan på samma sätt. Ett nytt lager oxid och fotokänsligt material läggs på och belyses med nästa mönster för att bana väg för gate-kislet.[1] Dessa steg upprepas sedan om och om igen för att bygga upp alla lager av olika material som utgör kretsen.

De mönster som används för varje belysningssteg kallas för masker. Varje mask är ett noggrannt framtaget orginal som oftast är förstorat 10 gånger i förhållande till kretsen. Idag är det inte ovanligt att det krävs ett trettiotal masker för en enda design. Dessa uppsättningar med masker kallas för maskset och kostar avsevärda summor att tillverka. För en modern process kan det kosta bortåt 5 miljoner USD för ett maskset vilket gör att uppstartskostnaden för en kretsdesign ofta är för hög för mindre tillverkningsserier.

1.4 Moores lag

En nuförtiden mycket välkänd person vid namn Gordon E Moore, en av grundarna till företaget Fairchild Semiconductor, skrev år 1965 en artikel med titeln “Cramming

[1] I själva verket läggs gate-kislet på före dopningen av source och drain nuförtiden. Detta för att få så bra precision som möjligt i kanterna mellan dessa regioner.

more components onto integrated circuits”[2]. I den artikeln beskriver han hur man kan förvänta sig exponentiell tillväxt i antalet komponenter (d.v.s. transistorer) som får plats på en enda integrerad krets. Den tillväxt han förutsåg var ungefär en fördubbling av antalet per år. Det har visat sig att Moores förutsägelse var mycket bra sånär som på det exakta tiden för fördubbling av antalet komponenter. Det har visat sig att en fördubbling sker på ungefär 18 månader. Moores förutsägelse är numera känd som *Moores lag*. Den utveckling som Gordon E Moore förutsåg är helt och hållet den som har givit möjlighet att bygga persondatorer, mobiltelefoner, digital-TV, osv.

Moore gick för övrigt vidare i livet och var med att grunda Intel under 1970-talet.

1.5 Termer och annat språkrelaterat

Ett problem med teknisk litteratur inom elektronik- och dataområdet är att det i mångt och mycket saknas vedertagen svensk nomenklatur. Därför kommer denna text att försöka nyttja svenska ord där dessa finns och är vedertagna. De engelska uttrycken för dessa kommer även att ges. I övrigt kommer de engelska uttrycken att användas och läsaren får ha överseende med den “svengelska” som därmed uppstår.

[2]Gordon E Moore, “Cramming more components onto integrated circuits”, Electronics Magazine, Volume 38, nummer 8, 19 April 1965.

Kapitel 2

Minnen

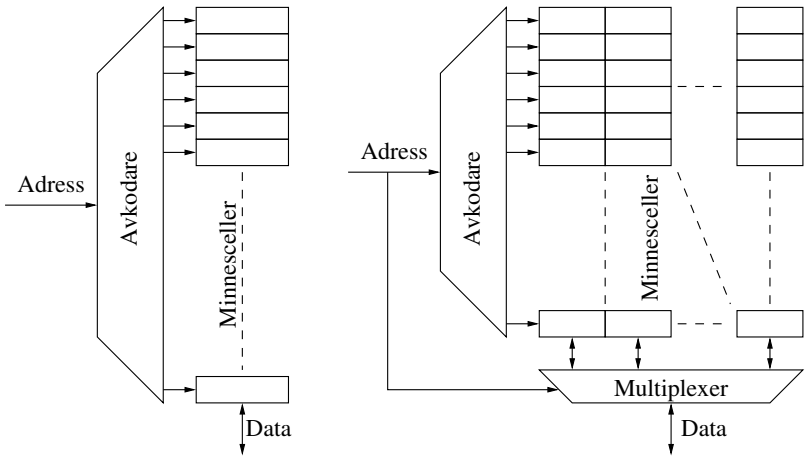
Minnen är en central del i ett datorsystem. För att kunna utföra mer komplicerade uppgifter så finns det i regel ett betydligt större behov av lagringsutrymme än det som finns tillgängligt i form av register. Och redan för enkla uppgifter så krävs det ett utrymme för det program som körs. Detta kapitel diskuterar olika typer av minnen baserade på halvledarteknik samt organisationen hos dessa.

2.1 Minnesorganisation

Från användarens perspektiv ser minnen i regel ut som höga och smala matriser med minnesceller, se figur 2.1(a). Man brukar kalla detta för ett minnes *externa organisation*. Denna organisation är i praktiken väldigt otaktisk när man bygger ett minne då en väldigt stor avkodare skulle behövas för adresseringen. I stället används en någorlunda kvadratisk matris med celler där delar av adressen används för att välja vilken rad och delar av adressen används för att välja vilken kolumn som ska användas enligt figur 2.1(b). Detta brukar kallas för minnets *interna organisation*. Fördelen i form av avkodare är uppenbar då ett minne på en megaord behöver en avkodare med 20 insignaler och 1048576 utsignaler. I ett kvadratiskt minne behövs istället en avkodare av storlek 10 in och 1024 ut samt en multiplexer för 1024 till 1. Den sammanlagda komplexiteten för dessa är avsevärt mindre än för den förstnämnda lösningen.

2.2 Minnestyper

Minnen kan grovt klassificeras i två grupper. Dessa är ROM, *read only memory*, och RWM, *read/write memory*. I den första kategorin ingår minnen som under normala förhållanden bara kan läsas medan den andra kategorin utgörs av minnen där det även går att skriva. Exempel på ROM är mask-ROM och fabrikspressade CD-skivor. Exempel på RWM är RAM, hårddiskar och magnetband. Ett mellanting mellan ROM och RAM utgörs av så kallade WORM-media, *write once/read many*. Ett typiskt exempel på WORM-media är CD-R.



(a) Extern vy (b) Intern organisation

Figur 2.1: Extern vy respektive intern organisation av ett minne

2.3 ROM

Den första och enklaste varianten på minne är ett ROM, *read only memory*. Det är precis som namnet antyder ett minne som inte går att skriva vid normal användning. En stor fördel med ROM jämfört med ett vanligt RAM (se nedan) är att minnesinnehållet blir kvar oavsett om det finns matningsspänning till minnet eller inte. Detta innebär att ett ROM är permanent (*permanent* eller *persistent*) och att ett RAM är flyktigt (*volatile*). Minnen av ROM-karaktär används vanligen till permanent eller semi-permanent lagring av data och program i ett system, typiskt i form av ett BIOS, *basic input/output system*, som används för att starta en dator.

Den mest fundamentala varianten av ROM är maskprogrammerat. En sådan variant går ut på att det finns en komplett design av ett ROM där en utseendet på en mask kan varieras på kretsen i ett tillverkningssteg. Genom att utlämna vissa anslutningar i denna mask och ta med andra går det att lagra “ettor” och “nollor”. Nackdelen med maskprogrammerade ROM är att det krävs en stor serie för att motivera den kostnad det innebär att påverka kretstillverkningen samt att ett program i mask-ROM är väldigt permanent då det inte finns någon möjlighet att ändra det som är lagrat i ROMet. Fördelen är att det blir extremt billigt vid väldigt stora serier.

Varianter på ROM är PROM, *programmable ROM*, och EPROM, *erasable programmable ROM*. Ett PROM är ett ROM där det går att bränna av anslutningar med hjälp av höga strömmar/spänningar och därmed erhålla samma funktion som i ett maskprogrammerat ROM. När ett PROM nyttjas i en design har tillverkaren den fördelen att innehållet i ROMet kan ändras under en tillverkningsserie. Dock är det som en gång skrivits in i ett PROM permanent och kan inte ändras. Eftersom ett PROM går

att skriva endast en gång kallas det även för *one-time programmable* eller OTP-minne.

EPROM är en ytterligare utveckling där det går att radera innehållet med hjälp av ultraviolett ljus. Dessa kretsar har ett karakteristiskt särdrag i det glasfönster som sitter i kapslingen för att släppa in UV-ljuset till själva kretsen. Raderbarheten har dock ett pris då den kapsling med fönster som krävs måste göras av ett keramiskt material i stället för plast. Denna keramiska kapsling är mycket dyr.

Ytterligare exempel på ROM är CD- eller CD-R-skivor. Dessa kommer dock inte att behandlas i denna text.

2.3.1 Elektriskt raderbara ROM

Det finns två typer av elektriskt raderbara ROM. Dessa är EEPROM, *electrically erasable PROM*, och flashminne. Båda dessa typer använder tekniker som gör att de är omprogrammerbara enbart med hjälp av elektricitet (d.v.s. man slipper UV-belysningen). En skillnad från användarens perspektiv är att EEPROM i regel kan radera en cell i taget medan ett flashminne oftast raderar större block. Blockstorleken för flashminnen minskar dock så att skillnaden mellan typerna minskar.

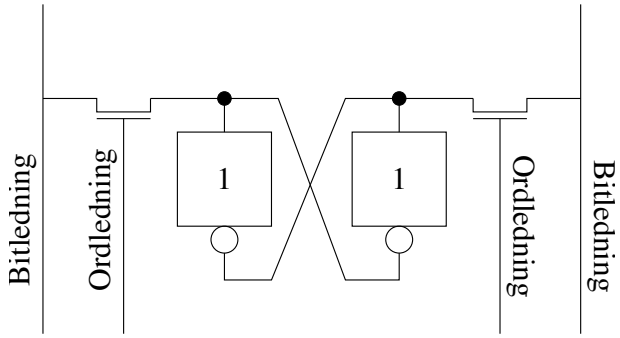
Ett elektriskt raderbart ROM ligger i gränslandet mellan ROM och RWM det det går att se ett elektriskt raderbart ROM som en typ att RWM. Dock har man vissa begränsningar jämfört med ett generellt RWM då skrivtiderna i regel är betydligt längre i ett raderbart ROM och att det finns en begränsning på hur många gånger man kan skriva till minnet innan det slutar fungera. Detta på grund av att man skriver EEPROM och Flash genom att lägga på en spänning över en isolator som är så hög att det blir genomslag. Upprepade sådana genomslag sliter på isolatorn som till slut inte längre klarar av att isolera de vanliga spänningar som används. Normalt sett ligger gränsen vid maximalt $10^4 - 10^5$ skrivningar per cell.

2.4 RWM

RWM eller *read/write memory* är som namnet antyder ett både läs- och skrivbart minne. Den vanligaste typen av RWM är RAM, *random access memory*, som tillåter både läsning och skrivning av godtycklig minnescell och där både läsning och skrivning tar (ungefär) lika lång tid.

2.4.1 SRAM

Ett statiskt RAM (SRAM) utgörs av ett antal minnesceller av vipp-liknande konstruktion, se figur 2.2. Minnesinnehållet hålls stabilt av de två korskopplade inverterarna och kan läsas genom att lägga en lämplig spänning på ordledningarna. Innehållet kommer då att kopplas ut på bitledningarna och kan avkännas. En skrivning går till så att fix spänning läggs differentiellt på bitledningarna, t.ex den vänstra på +5V och den högra på 0V. Om det nya värdet är lika som det gamla kommer inget att hända men



Figur 2.2: En minnescell i ett statiskt RAM

om det nya värdet är inversen av det gamla så kommer cellen att slå om. Detta sker under förutsättning att drivstyrkan hos bitledningarna är högre än drivstyrkan hos de korskopplade inverterarna.

Dessa celler består av ett relativt litet antal transistorer jämfört med en vippa (se digitaltekniken!) för att hålla kvar samt möjliggöra läsning och skrivning av data. Totalt används i regel sex transistorer per cell. Det krävs dock en viss mängd kringelektronik inne i SRAM-kretsen för att generera de signaler som krävs för läsning och skrivning. Det krävs även förstärkare för de signalnivåer som finns vid läsning för att dessa ska vara användbara. Samtlig kringelektronik går dock att använda för många minnesceller så kostnaden för denna blir relativt liten per cell i stora minnen.

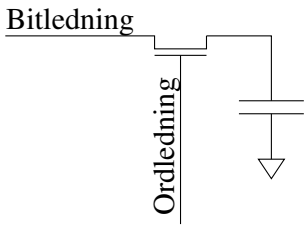
2.4.2 DRAM

Till skillnad från ett statiskt RAM är ett dynamiskt RAM (DRAM) baserat på att en laddning lagras i en kondensator, se figur 2.3. Fördelen är att DRAM-cellen är mycket mindre än SRAM-cellen och därför går det att göra mycket större minnen på samma kretsyta med DRAM-teknik. Detta gör att DRAM är avsevärt mycket billigare än SRAM.

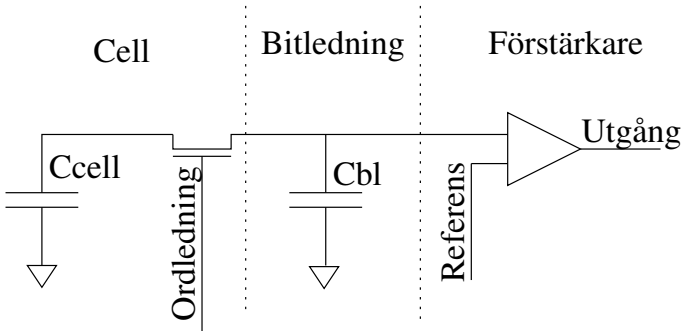
I en ideal värld så skulle ett dynamiskt RAM bete sig väldigt likt ett SRAM. På grund av vissa icke-ideala fenomen, t.ex. läckage i transistorn och kondensatorn, så kommer laddningen i cellen att gradvis försvinna. För att motverka detta krävs periodisk återskrivning av den information som är lagrad i cellen, s.k. *refresh*. Maximal tid mellan två återskrivningar av en cell är typiskt några tiotals millisekunder.

Ett annat relaterat problem är att det inte finns någon drivförmåga i cellen. Detta ställer till problem vid utläsning av data då det enbart finns den lagrade laddningen att tillgå. I ett SRAM finns det en aktiv drivförmåga i de inverterare som utgör cellen.

Vid utläsningen av ett DRAM måste man ta hänsyn till omgivningen i mycket större utsträckning än i ett SRAM. I figur 2.4 visas en förenklad uppkoppling för utläsningen av en DRAM-cell. Cellen har en specifik kapacitans C_{cell} som är kopplad till bitledningen via en transistor som antar är ideal. Om vi dessutom antar att bitledningen



Figur 2.3: En minnescell i ett dynamiskt RAM

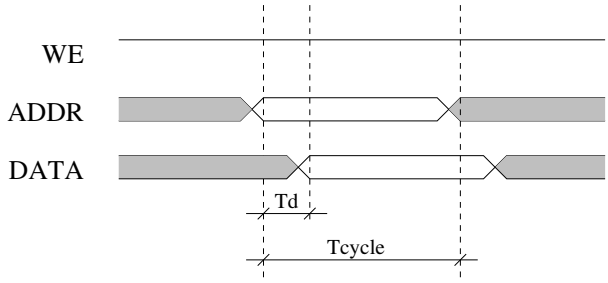


Figur 2.4: Modell för utläsning av data i ett dynamiskt RAM

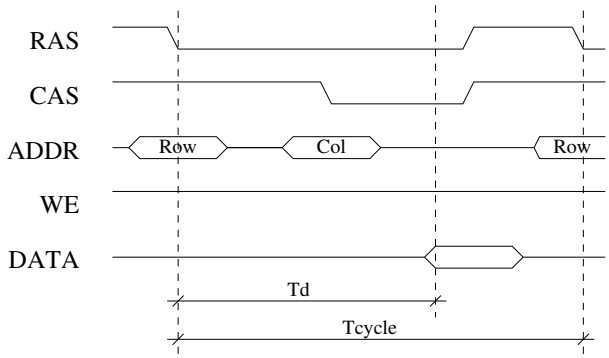
är ideal kommer spänningen över C_{cell} att kopplas ut oförändrad till förstärkaren vid utgången. I verkligheten har bitledningen en s.k. parasitisk kapacitans till omgivande delar av kretsen. Denna går att modellera som en kapacitans till jord av storleken C_{bl}. Förhållandet mellan dessa två kapacitanser är ungefär $C_{cell}/C_{bl} \approx 0.25$. Detta innebär att den skillnad i spänning som upplevs vid förstärkarens ingång endast är 20% av skillnaden i spänning mellan bitledningen och cellen vid utläsningens start på grund av laddningsdelningen mellan kapacitanserna. Om kretsen har en matningsspänning på 2,5 V och bitledningen förspänns till halva denna kommer förändringen vid förstärkarens ingång att uppgå till maximalt 0,25 V. Detta är dock inte helt sant då det förutsätter en helt "oläckt" cell. I ett verkligt fall kan läckaget i cellen sänka denna förändring till under 0,1 V. Den förstärkare som används är av en återkopplad typ som är väldigt lik en SRAM-cell. Detta innebär att när förstärkaren har detekterat cellinnehållet kommer den att driva bitledningen till detta värde och på så sätt återskriva cellinnehållet. En refresh utgörs alltså av en läsning där resultatet på utgångarna från förstärkarna ignoreras.

2.4.3 Timing hos minnen

I ett vanligt asynkront SRAM är timingen hos signalerna relativt enkel. Figur 2.5 visar ett typiskt timingdiagram för läsning från ett SRAM. De viktigaste tiderna är medtagna i diagrammet. Den fördröjning i kretsen som uppstår från att en adress är stabil på



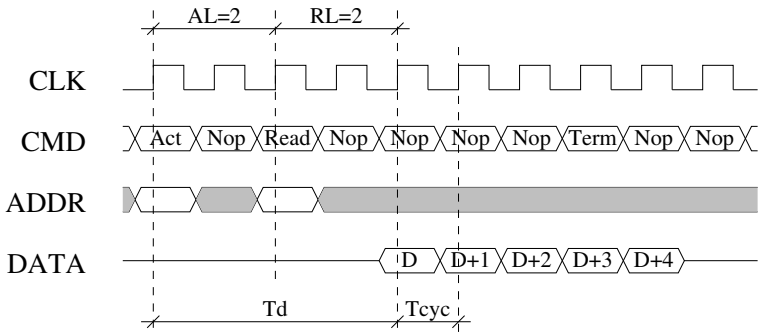
Figur 2.5: Åtkomsttid och cykeltid för ett asynkront SRAM



Figur 2.6: Typiskt timingdiagram för ett DRAM

ingångarna till den lagrade informationen finns tillgänglig på utgångarna är markerad med T_d. Denna tid kallas åtkomsttid eller *accesstid* och är normalt sett relativt kort, i storleksordningen enstaka nanosekunder. Den andra tiden är cykeltiden T_{cycle}, d.v.s. den kortaste tid som en adress måste vara stabil för att minnet ska fungera. Idag finns det minnen att tillgå där cykeltiden ligger under tio nanosekunder. För ett minne som integreras på ett chip tillsammans med andra kretsar går det att komma ner i enstaka nanosekunder.

När DRAM blev populärt i elektronik såsom de tidiga persondatorerna uppstod ett problem. De kapslingar som fanns att tillgå hade inte tillräckligt många anslutningar för att både adress, data och kontrollsignaler skulle få plats för de - med den tidens mått mätt - stora minnena. Lösningen på detta var att dela upp adressen i två delar och på så sätt minska antalet anslutningar för adressen till hälften. Dessa två delar speglar indirekt det relativt kvadratiska utförandet av minnesmatrisen och benämns radadress (*row address*) och kolumnadress (*column adress*). Genom uppdelningen måste dessa adresser därför presenteras till kretsen efter varandra. Figur 2.6 visar ett timingdiagram för ett klassiskt DRAM. De två signalerna RAS, *row address strobe*, och CAS, *column address strobe*, används för att ladda in de två adresshalvorna till minnet. En stund efter att kolumnadressen har laddats in så kommer data att vara tillgängligt på dataanslut-



Figur 2.7: Funktionen hos ett SDRAM

ningarna. Den begränsande tiden för ett klassiskt DRAM är den tid som måste fortlöpa mellan två fallande flanker på RAS-signalen. Denna tid är i storleksordningen minst 60 nanosekunder på kommersiellt tillgängliga DRAM.

2.4.4 SDRAM

Vartefter utvecklingen av elektroniken har fortgått så har minnen blivit en flaskhals i systemen. Detta på grund av att minnenas hastighet inte har skalats i samma utsträckning som logiken. Ett av de primära problemen med tidigare DRAM-teknik var att det inte var möjligt att göra flera läsningar eller skrivningar direkt efter varandra med enbart accesstiden som begränsning. Detta problem uppstår delvis på grund av att gränssnittet mellan minnet och omgivningen är asynkront, d.v.s. den saknar en gemensam tidsreferens och måste därför designas på ett "säkert" sätt.

För att motverka detta specificerades ett synkront gränssnitt för minnen. Denna typ av minnen kallas för synkront (*synchronous*) dynamiskt RAM, SDRAM. Detta gränssnitt tillåter sådant som att flera läsningar kan köas upp i väntan på att data ska vara tillgängligt. Denna metod ger avsevärt mycket högre användbar bandbredd hos minnet samt att det är enkelt att tillåta läsningar på konsekutiva adresser utan att tid behöver ödslas på att nya läskommandon ges till minnet. Figur 2.7 visar hur detta går till. Först skickas kommandot[1] *activate* för att väcka minnet till liv och låsa radadressen. Detta kommandot är i stort sett ekvivalent med RAS-signalen i ett klassiskt DRAM. En stund senare skickas kommandot *read* tillsammans med kolumnadressen. När minnet har hämtat den information som fanns lagrad vid denna adress så presenteras denna på databussen. Finessen här är dock att minnet inte bara hämtar den information som efterfrågas direkt utan hämtar samtlig information som finns inom ett block. Ett sådant block utgörs ofta av en rad i minnet. Detta innebär att en mängd data, t.ex. 256 ord, kan läsas direkt efter varandra nyttjandes endast en klockcykel per ord. När denna läsning är slut avbryts den med ett *terminate*-kommando. Typiska tider för ett standard

[1] I ett SDRAM har läs- och skrivsignalerna ersatts med en fyra-bitars kommandobuss.

SDRAM är en cykeltid på 6-7 nanosekunder och en accesstid, i synkrona system normalt kallat latens (*latency*), på fyra till sex cykler. För en läsning utgörs denna latens av AL, *activation latency*, och RL, *read latency*. I nyare versioner av SDRAM (t.ex. DDR, *double data rate*) så har cykeltiden minskat medan latenserna har ökat.

2.4.5 FRAM

FRAM, *Ferroelectric RAM*, är en ny lovande minnestyp baserad på elektromagnetiska fenomen som har dykt upp under de senaste åren. I ett FRAM lagras informationen i en magnetisk kristall inbyggd i minnescellen.[2] Det är magnetiseringens riktning som bestämmer minnescellens innehåll. Läsning och skrivning till minnet sker på elektromagnetisk väg genom de fält som uppstår då ström passerar genom en ledare invid kristallen.

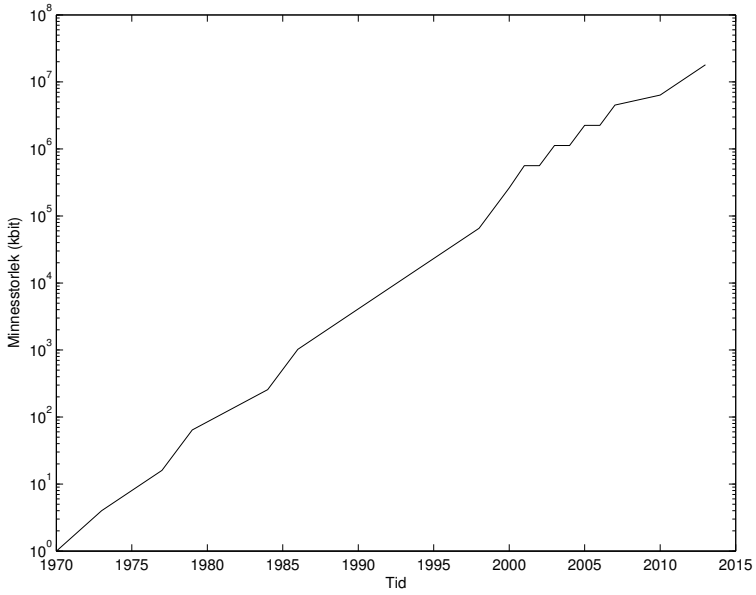
Genom att minnesinnehållet lagras som en magnetisk polarisering i FRAM så blir minnet oberoende av matningsspänning för att hålla kvar informationen. Dessutom behöver ett FRAM inte använda någon av de metoder för att skriva som normalt sett sliter ut ett EEPROM/Flash. Detta gör att begränsningen för antalet skrivningar blir avsevärt mycket högre, i dagsläget upp emot 10^{10} skrivningar. En annan fördel gentemot EEPROM och Flash är att skrivning till minnet inte tar längre tid än läsning från minnet.

Än så länge är FRAM förhållandevis små men utvecklingen pekar på betydande framsteg inom de närmaste åren. Fördelarna med FRAM gör att utvecklingen på sikt mycket väl kan bli att FRAM slår ut EEPROM och Flash för lagring av semipermanent data. När detta skrivs har både Ramtron och Fujitsu presenterat FRAM med storleken 1 Mbit (128k x 8).

2.5 Storlekar och utveckling

Den mängd minnesceller som får plats i en IC-krets skalar enligt Moores lag. Detta innebär att den möjliga minnesmängden dubbleras ungefär var 18 månad. En graf över storleken hos DRAM visas i figur 2.8. De värden som ligger efter år 2000 kommer från *international technology roadmap for semiconductors*[3], en publikation från ett av halvledarindustrins samarbetsorgan. Här förutses en transistorstorlek av endast 18 nanometer år 2013. Detta innebär att ett minne på 64 Gbit, d.v.s. mer än 68 miljarder minnesceller ska få plats i en enda IC-krets.

[2] http://www.ramtron.com
[3] http://public.itrs.net



Figur 2.8: Minnesstorlek för en DRAM-krets

2.6 Jämförelse med andra typer av minnen

Förutom dessa halvledarbaserade minnen så finns det en uppsjö av olika minnen där minnet inte baseras på en elektrisk laddning eller ström. Dessa andra minnen är vanligtvis baserade på magnetisk polarisering av ett magnetiskt medium. Typiska exempel är magnetband och hårddisk.

En hårddisk består av ett antal roterande skivor som är belagda med ett magnetiskt lager. Genom forskning och utveckling har tillverkarna av hårddiskar lyckats åstadkomma en relativt hög packningstäthet av "minnescellerna". Skivorna ger en hårddisk har en avsevärt mycket större area för lagring än ett halvledarbaserat minne. För att ta ett exempel så har en hårddisk med fyra skivor med en diameter på sju cm en total yta på ca 300 cm^2. Detta ska jämföras med ett halvledarminnes maximala 4-5 cm^2. Genom denna storleksfördel kan en hårddisk hålla mycket mer information än halvledarminnen vid samma kostnad. Detta har dock ett annat pris i form av accesstid och bandbredd som hos hårddisken är avsevärt mycket lägre än för ett halvledarminne. En hårddisk har typiskt accesstid av millisekund-storlek.

Kapitel 3

Minnesarkitekturer

3.1 Bakgrund

Det finns två primära arkitekturer för minnesanslutning till en processorkärna. Skillnaden mellan dessa två är om program och data ligger i samma minne eller är åtskilda. Som vi ska se senare så är det möjligt att blanda dessa två arkitekturer om det finns en minneshierarki i systemet.

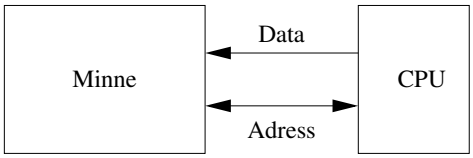
3.1.1 von Neumann

En tidig förespråkare av datorer med lagrat program var den mycket framstående matematikern John von Neumann. Den tidigaste minnesarkitekturen för datorsystem har därför uppkallats efter honom. I denna arkitektur nyttjas ett gemensamt minne för både program och data. Figur 3.1(a) visar ett typiskt von Neumann-system.

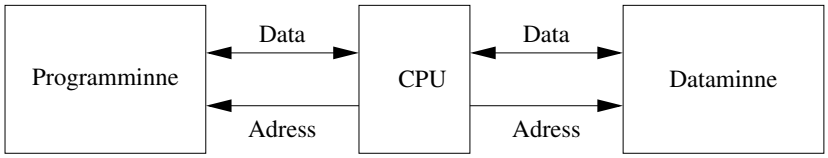
Den främsta fördelen med en von Neumann-arkitektur är att det bara finns en minnesrymd. För en generell dator där konstruktören inte vet vilka program som kommer att köras är detta en fördel. För vissa program kan behovet av programminne vara stort medan det knappt krävs något dataminne. För andra program kan det istället vara tvärtom. Den stora nackdelen är att det inte går att hämta instruktioner och data parallellt.

3.1.2 Harvard

En Harvard-arkitektur, se figur 3.1(b), har till skillnad mot von Neumann-arkitekturen separata minnen för data och program. Den stora fördelen här är att det går att hämta både instruktioner och operander samtidigt. Nackdelen är naturligtvis att förhållandet mellan minnenas storlek är fastställt i och med tillverkningen av systemet.



(a) von Neumann-arkitektur



(b) Harvard-arkitektur

Figur 3.1: Traditionella minnesarkitekturer

3.1.3 Arkitekturval

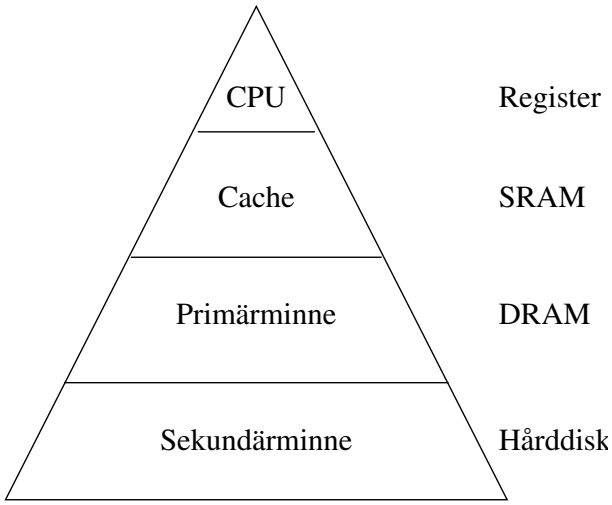
De ovan beskrivna skillnaderna mellan von Neumann- och Harvard-arkitekturerna ger en förståelse för vilka system som väljer vilken arkitektur. En vanlig PC använder en von Neumann-arkitektur[1] då det inte går att förutse vilka program som kommer att köras. Detta är även giltigt för större datorsystem som (massivt parallella) superdatorer.

Ett typiskt system som nyttjar Harvard-arkitekturen är digitala signalprocessorer (DSP). Dessa har dessutom i regel flera dataminnen för att tillåta maximal beräkningshastighet. En typisk uppgift för en DSP är ett FIR-filter. Här hämtas två operander som multipliceras för att sedan adderas med en summa i en ackumulator. För att utföra en sådan "multiply-and-accumulate"-instruktion i en cykel krävs en access till programminnet och två separata accesser till dataminnen. Andra system som nyttjar Harvard-arkitekturen är små datorer (microcontrollers) som används för enklare inbyggnadssystem.

3.2 Minneshierarkier

I ett mer avancerat system där det minne som integrerats på processorkretsen inte är tillräckligt nyttjar man i regel en hierarki av olika minnestyper. Dessa är ordnade så att närmast processorn sitter de snabbaste minnena som dock är väldigt dyra per lagrad bit. Längre ifrån processorn finns de minnen som är långsammare och billigare. På detta

[1] Detta är inte helt sant. På hög nivå ser en modern PC ut att använda en von Neumann-arkitektur men på nivån närmast processorn nyttjas en Harvard-arkitektur med separata cacheminnen för instruktioner och data.



Figur 3.2: Minneshierarki

sätt kan man få en ekonomiskt sett bättre situation än att bara nyttja dyra och snabba minnen till priset av något försämrad prestanda. Iden bygger på att minnesåtkomster i regel är "lokala", d.v.s. att konsekutiva accesser är nära varandra i minnet (*locality of access*). Det är också vanligt med loopar i program vilket gör att samma programsnutt körs om och om igen. Därför kan ett relativt litet minne rymma tillräckligt mycket program och data för att kunna köra en stund.

Figur 3.2 visar en typisk minneshierarki i en PC. Längst upp finns processorkärnan med tillhörande register. Dessa är snabba, dyra och få. Nästa nivå är cache-minnet som utgörs av SRAM. Detta SRAM är inte lika snabbt och dyrt som registren samt kan vara betydligt större. Det kan även finnas flera nivåer av cache-minne som då benämns *level 1*, *level 2* osv. med lägsta numret närmast CPUn.

Efter cacheminnet kommer man till primärminnet (arbetsminnet). Detta utgörs av DRAM som nu för tiden är mycket långsammare än processorn och mycket billigare än SRAM.

Den nedersta nivån i figur 3.2 är sekundärminnet (*secondary memory* eller *secondary storage*), vanligtvis i form av en hårddisk. Sekundärminnet används för att lagra program och data permanent mellan körningar men även för att utöka primärminnet. När sekundärminnet nyttjas för att utöka primärminnet talar man om virtuellt minne (*virtual memory*) och att systemet växlar (*swap*)[2]

Det kan även finnas ytterligare en nivå i minneshierarkin. Denna benämns då tertiärminne (*tertiary storage*). Typiskt utgörs detta av någon form av säkerhetssystem i form av magnetband.

Vi kan se fyra tydliga kriterier till varför denna pyramid är en gångbar lösning på problemet med önskvärda minnesstorlekar resp minneshastigheter. Desto längre ner i

[2]För mer information om virtuellt minne se Olof Roos, "Grundläggande datorteknik", kapitel 8.

hierarkin man kommer desto:

1. lägre kostnad per lagrad bit.
2. större kapacitet.
3. större åtkomsttid.
4. lägre åtkomstfrekvens.

De tre första kriterierna uppfylls direkt genom de respektive minnenas natur i dessa frågor. Det fjärde kriteriet motsvarar den tidigare nämnda åtkomstlokaliteten (*locality of access*).

3.3 Cache-minnen

En nackdel med stora minnen är att de tenderar att bli långsammare än små minnen. Detta uppkommer på grund av två primära faktorer. Den första anledningen är att stora minnen alltid realiseras som DRAM istället för SRAM. Den andra anledningen är att ett större minne kräver mer komplicerade kringkretsar för att göra läsningar och skrivningar möjliga vilket ytterligare drar ner hastigheten på minnet.

På 1970-talet gjordes ett antal undersökningar av minnesåtkomster i program där slutsatsen var att det är en stor grad av lokalitet i minnesaccesserna, alltså att programmen tenderar att läsa upprepade gånger inom ett litet område i minnet. Över tiden flyttar sig det lilla området av minnesaccesser. För att kunna utnyttja denna egenskap av lokalitet infördes ett lager av minnen mellan processorns register och det DRAM som utgör primärminnet. Detta nya lager av minne i hierarkin kallas cacheminnen och tar sitt namn från det engelska ordet cache som ungefär betyder en säker lagringsplats eller en plats för att dölja förråd eller proviant.

Ett cacheminne utgörs alltså av ett litet och snabbt SRAM som ligger både fysiskt och logiskt nära processorn. Ett datorsystem kan ha flera lager av cacheminnen men lagret närmast processorn ligger i regel i samma integrerade krets som processorn. En processor som kör med hög hastighet - flera gigahertz - har ett stort prestandagap mellan de interna registrens hastighet och det externa minnet. Att hämta instruktioner eller data i DRAM kan ta åtskilliga hundra klockcykler vilket drastiskt drar ner prestanda hos systemet. Avsnitt 2.4.4 visar att det tar ca fyra klockcykler att hämta det första dataordet i ett första generationens SDRAM. För ett “400 MHz” DDR-SDRAM så körs klockan i 200 MHz[3] och då har “read latency” (RL) ökat till fyra klockcykler vilket ger en total latens på sex klockcykler för en läsning[4]. Detta gör att minnet tar 30 ns på sig för en läsning. Om man jämför detta med en processor på 3 GHz motsvarar detta 90

[3]DDR, *double data rate*, innebär att man överför data på både stigande och fallande klockflanker. Detta gör att marknadsföringsfolket gav DDR dubbelt så hög “MHz” mot vad de egentligen körs med.

[4]Källa: www.micron.com

klockcykler. Till detta ska läggas den (avsevärda) tid som går åt till att undersöka innehållet i ett par lager cacheminnen samt den tid det tar för en minnesaccess att komma igenom alla lager av bussar och olika kretsar mellan processorn och primärminnet.

För mer detaljerad information om implementering av ett cacheminne hänvisas till Olof Roos, “Grundläggande datorteknik”, kapitel 9.

Kapitel 4

RISC

4.1 Bakgrund

När kompilatorerna för högnivåspråk (t.ex. C och Pascal) blev vanligare på 1960- och 1970-talen gjordes mycket forskning på hur effektiv kompilerad kod var på olika datorarkitekturer. Forskarna delades snabbt in i två läger där de ena lägret förespråkade mer komplicerade instruktioner så att kompilatorn skulle ha mer att välja på och på så vis förenkla kompilatorns jobb. Det andra lägret ansåg att instruktionerna skulle hållas enkla för att förenkla instruktionsvalet i kompileringen. Denna andra metod ledde fram till de så kallade RISC-processorerna. RISC eller *reduced instruction set computer* innebär att processorerna görs (i viss utsträckning) så enkla som möjligt.

4.2 RISC och CISC

Tabell 4.1 visar en jämförelse mellan RISC och CISC[1] vad gäller typiska karakteristika. Idén med RISC-processorer är att hålla konstruktionen så minimal och enkel att allt som den kan göra ska gå att göra på en instruktionscykel[2]. Genom att använda ett fast instruktionsformat för RISC-processorn så kan man garantera att en instruktion går att utföra inom en instruktionscykel. Fast instruktionsformat innebär helt enkelt att alla instruktioner är lika stora (typiskt 16 eller 32 bitar) inklusive eventuella operander.

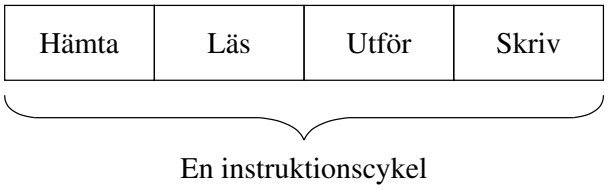
En instruktion i en godtycklig dator (vare sig det är en CISC- eller RISC-maskin) består i princip av fyra faser, se figur 4.1. Först hämtas instruktionsordet från instruktionsminnet och avkodas till någon form av styrsignaler till processorns delar. Denna avkodning kan ske till exempel genom ett mikroprogram i en CISC-maskin. Den andra fasen utgörs av att processorn hämtar de rätta operanderna. Fas nummer tre är själva

[1] CISC står för *complex instruction set computer* och innebär i praktiken en mikroprogrammerad processor.

[2] En instruktionscykel (maskincykel) är en fast tid som alla instruktioner tar på sig, t.ex. fyra klockcykler. Olof Roos, "Grundläggande datorteknik", kapitel 10 hävdar felaktigt att instruktionen ska utföras inom en *klockcykel*.

Tabell 4.1: Jämförelse av CISC och RISC

	CISC	RISC
Antal klockcykler per instruktion	10-200	1-4 (fixt)
Variabelt instruktionsformat	Ja	Nej
Antal adresseringsmoder	10-30	1-4
Indirekt adressering	Ja	Nej
ALU-operander i minnet	Ja	Nej
Antal register	8-32	32-512



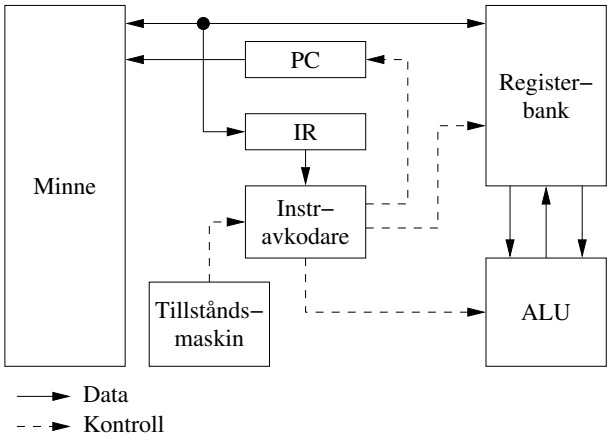
Figur 4.1: En RISC-instruktionscykel

utförandet av instruktionen (t.ex. en beräkning) och det hela avslutas med att resultatet sparas. I en CISC-dator så kan var och en av faserna ta (i princip) godtycklig tid medan en RISC-dator har begränsat funktionen så att varje fas har en fast tid (t.ex. en klockcykel). "Straffet" för den fasta instruktionscykeltiden och det fasta instruktionsformatet är att en RISC-processor inte kan utföra lika avancerade funktioner som en CISC-processor.

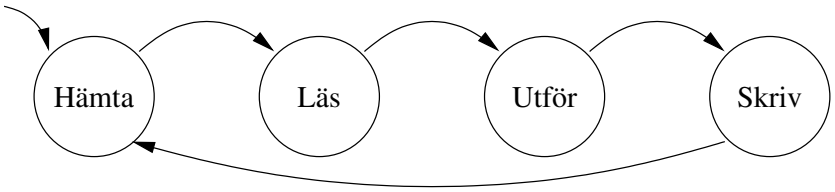
Det har visat sig att det läger som förespråkade en enkel arkitektur är det mest framgångsrika vad gäller kompilerad kod. Anledningen är att instruktionerna i ett högnivåspråk inte reflekterar instruktionerna i en processor. Detta gör att kompilatorn har stora problem att utnyttja de komplicerade instruktioner som processorn erbjuder[3] och att endast en delmängd av dem kommer att användas.

Nackdelen med en RISC-processor är i stället att det krävs ganska många instruktioner som inte gör något "produktivt" (i form av beräkningar och liknande) utan bara är till för att flytta data fram och tillbaka mellan register och minnen. Fördelen är i stället att en RISC-processor går att köra snabbt samt att instruktionscykeln i regel är mycket kortare än för en CISC-processor vilket gör att RISC-processorer oftast ändå har en nettofördel gentemot CISC-processorer.

[3]Denna del av kompilatorprocessen kallas för *instruction selection* och är – som nästan alla andra intressanta problem – NP-komplett, d.v.s. tiden för att hitta en optimal lösning ökar exponentiellt med storleken på problemet. Detta gör att en optimal lösning i praktiken aldrig kan nås.



Figur 4.2: Enkel modell av en RISC-processor



Figur 4.3: Tillståndsmaskin för faser i en RISC-maskin med fyra klockcykler per instruktion

4.3 Direktavkodning

I en RISC-processor används inte ett mikroprogram för att specificera hur instruktioner ska utföras. I stället nyttjas något som kallas för direktavkodning. Detta innebär att instruktionerna avkodas med hjälp av ett kombinatoriskt nät. Figur 4.2 visar en förenklad modell[4] av en RISC-processor. Instruktionsavkodaren tar helt enkelt in innehållet i instruktionsregistret (IR) och avkodar detta – kombinatoriskt – direkt till mikrooperationer/styrsignaler.

En RISC-processor kan antingen använda en eller flera klockcykler för att utföra en instruktion. I en RISC-processor som använder flera klockcykler till en instruktion finns en enkel tillståndsmaskin som beskriver vilken fas det är som körs i det aktuella klockintervallet. Figur 4.3 visar grafen för en sådan tillståndsmaskin som ger fyra klockcykler per instruktion.

[4]Några (viktiga!) detaljer som t.ex. villkorsregistret är utelämnade för överskådlighetens skull.

4.4 Lästips

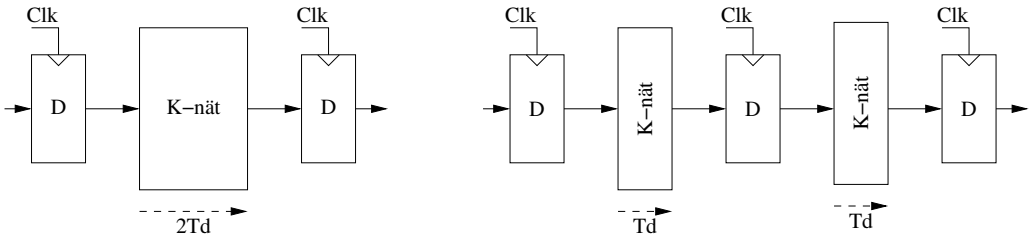
Det finns en mycket kort introduktion till RISC i Olof Roos, “Grundläggande datorteknik”, kapitel 10.

Ett mycket bra avsnitt om RISC-processorer går att finna i boken “Computer organization and architecture” av William Stallings. Här beskrivs både bakgrunden och principerna för RISC-processorerna i detalj.

Kapitel 5

Pipelining

Figur 5.1(a) visar en del av ett sekvensnät. Den kombinatorik som ligger mellan de två registren bestämmer i stort sett den maximala hastigheten hos sekvensnätet.[1] Genom att dela upp kombinatoriken i flera delar och sätta vippor mellan dem går det sålunda att minska den fördröjning som bestämmer den maximala klockfrekvensen. Till exempel kan man dela upp det kombinatoriska blocket i två delar med någorlunda lika fördröjning som figur 5.1(b) visar. Det är uppenbart att klockfrekvensen direkt går att dubblera då varje block nu bara har hälften så lång fördröjning som förut. Men den totala tiden för en "beräkning" att flyta igenom nätet är dock konstant då det nu krävs två klockcykler!



(a) Utan pipelining

(b) Med pipelining

Figur 5.1: Pipelining av ett sekvensnät

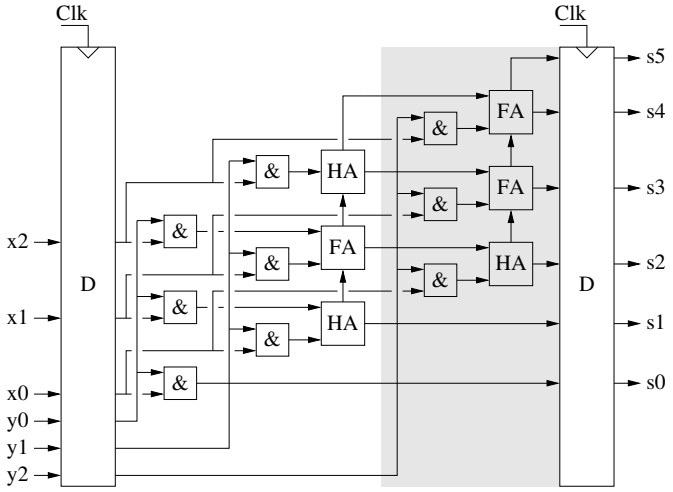
Fördelen är då att två beräkningar kan delbearbetas samtidigt, en i det vänstra steget och en i det högra. Alltså har vi möjligheten att öka den så kallade genomströmningen (*throughput*) i processen. Kostnaden för detta är då att vi har fått en längre fördröjning (*latency*) av beräkningen i antal klockcykler.[2] Principen är alltså väldigt lik det löpande bandet inom tillverkningsindustrin. Det vore synnerligen ineffektivt om hela fabriken

[1] D-vipporna har i realiteten också en viss fördröjning men den ignoreras för enkelhetens skull.

[2] På grund av de tidigare nämnda fördröjningarna i vipporna kommer i själva verket den totala fördröjningen även i absoluta termer att bli längre.

koncentrerar sig på att bygga en enda bil. I stället så byggs bilen bit för bit på olika stationer. Varje station ser då till att en specifik del av bilen läggs till de redan färdiga delarna och till slut har man bilen färdig.

5.1 Ett litet exempel i digitalteknikens värld



Figur 5.2: En-cykels multiplikator för 3-bitarstal

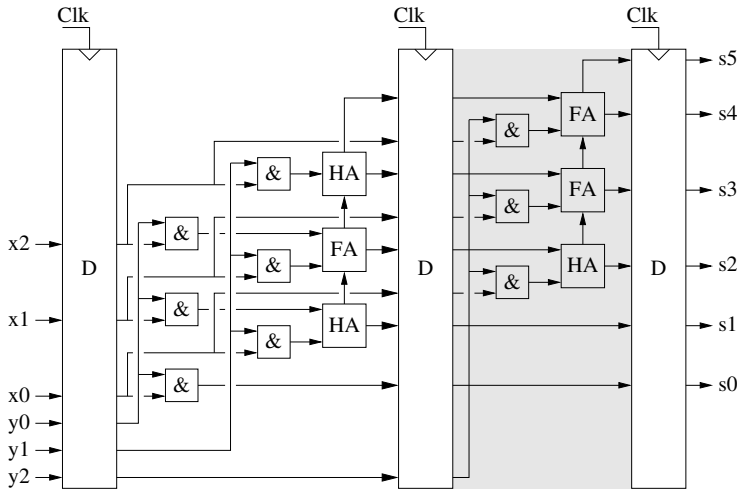
Figur 5.2 visar en enkel 3-bitarsmultiplikator som beräknas på en klockcykel. De två operanderna X=<x_2, x_1, x_0> och Y=<y_2, y_1, y_0> multipliceras med hjälp av summering av delprodukter. Summan S=<s_5, s_4, s_3, s_2, s_1, s_0> blir då:

$$S = Xy_0 + 2Xy_1 + 4Xy_2$$

Det oskuggade området i figuren räknar ut delsumman $S_1 = Xy_0 + 2Xy_1$ medan det skuggade beräknar delsumman $S_2 = S_1 + 4Xy_2$ (d.v.s. slutresultatet S). Om vi antar att fördröjningen genom en halvadderare är samma som för en heladderare (vilket i stort sett är sant) så blir fördröjningarna genom den oskuggade och den skuggade delen lika stora. Den längsta kedjan av fördröjningar består då av en och-grind samt tre adderare i varje del.

Den här kretsen är en definitiv kandidat för pipelining då det redan från början finns en klar linje som delar kretsen i två likvärdiga delar. Om man kapar samtliga ledare som passerar mellan det oskuggade och det skuggade området och lägger in en uppsättning D-vippor så får vi en naturlig pipelining.

Figur 5.3 visar multiplikatorn efter denna ändring. Att klockfrekvensen går att höja för den här kopplingen är uppenbart då den totala fördröjningen mellan efterföljande vippor har delats med två. Att även genomströmningen ökar kan förklaras av att under en klockcykel kan nu de två delsummorna för efterföljande beräkningar skapas samtidigt och helt oberoende.



Figur 5.3: Två-cykels multiplikator

5.2 Pipelining av processorer

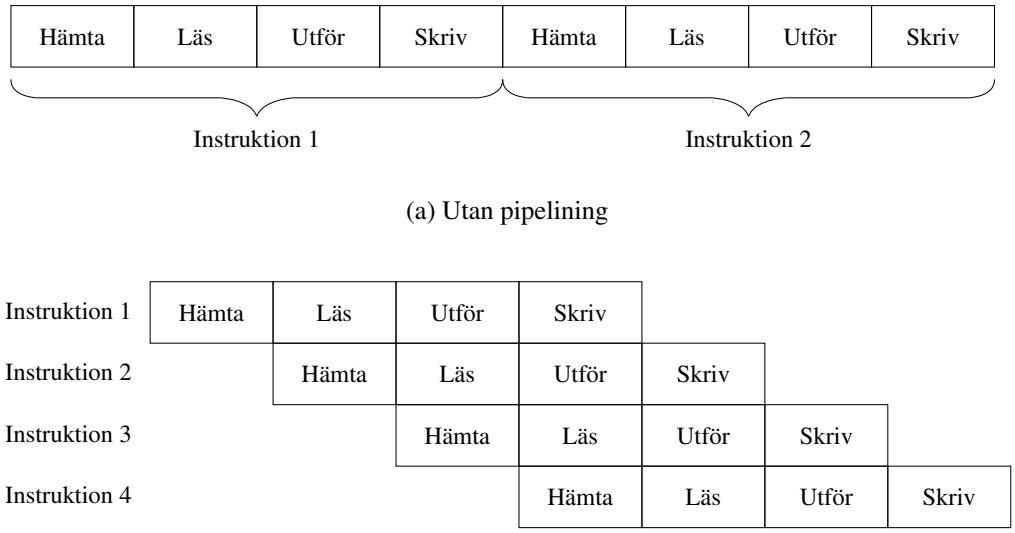
Principen går att applicera på en processor då varje fas i en instruktion i regel endast nyttjar en del av processorn. Om de olika faserna nyttjar helt olika delar går det att köra samtliga faser parallellt – men dock inte för samma instruktion! En processor utan pipelining kör instruktionerna helt sekventiellt. Med pipelining går det istället att överlappa instruktionerna och på så sätt nå högre genomströmning (throughput) i processorn. Skillnaden illustreras tydligt i figur 5.4. I den ej pipelinade versionen, figur 5.4(a), tar två instruktioner åtta klockcykler[3] medan den pipelinade versionen i figur 5.4(b) klarar av fyra instruktioner på sju klockcykler och varje ytterligare instruktion ökar denna tid med bara en klockcykel!

5.3 Problem med pipelining

Det största problemet med pipelining är att det kan finnas beroenden från en instruktion till nästa. Dessa beroenden kan leda till en uppsjö olika problem som gör att vissa instruktioner inte kan fortsätta. Man talar då om en pipelinekonflikt (*pipeline hazard*) och detta måste hanteras på något sätt. Det finns två principer för att hantera pipelinekonflikter. Antingen så hanteras den av hårdvaran i processorn eller så antar processorn att koden är konfliktfri. Att processorn antar att koden är konfliktfri kan generera intressanta resultat om programmeraren inte inser denna begränsning vilket behandlas senare i kapitlet.

Lösning av konflikter i hårdvara kan göras på ett antal sätt, t.ex. genom att proces-

[3] Om vi antar fyra faser och en klockcykel per fas.



(a) Utan pipelining

(b) Med pipelining

Figur 5.4: Pipelining av en RISC-processor

sorn måste *stalla*[4], d.v.s stanna till under tiden som det tar att vänta på den resurs som krävs. Pipelinekonflikter kan delas upp i tre grupper beroende på varför de uppstår.

- **Strukturell konflikt** eller *structural hazard* innebär att två (eller flera) steg i en pipeline samtidigt vill nyttja en gemensam resurs. Detta kan till exempel vara ett minne, en databuss eller ALUn. Eftersom endast en instruktion åt gången kan använda resursen måste de andra stoppas.

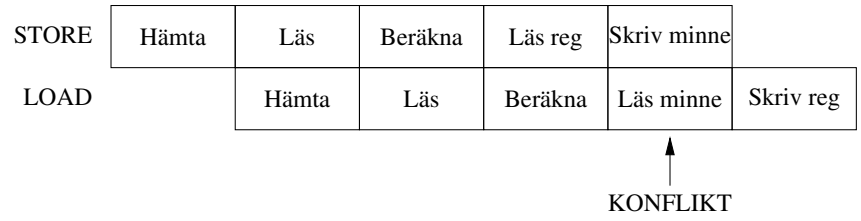
- **Datakonflikt** eller *data dependency hazard* beror på att en instruktion behöver indata som beräknas i en tidigare instruktion. Om instruktionerna "ligger för tätt" kan det bli problem då det data som krävs ännu inte finns tillgängligt.

- **Styrkonflikt** eller *control hazard* uppstår då ett hopp ska utföras. För att fylla pipelinen krävs ett konstant inflöde av instruktioner (det vill säga ett konstant utförande av hämtfaser). När ett hopp ska utföras måste den nya adressen först beräknas innan instruktionerna kan hämtas och följaktligen måste pipelinen stoppas i väntan på nya instruktioner.

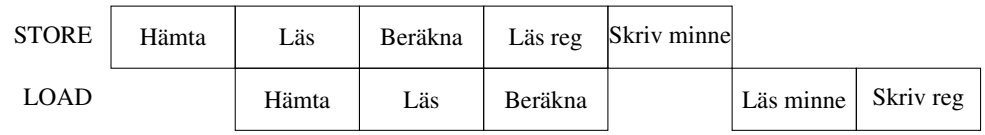
[4]Stalla kommer från engelskans *stall* och betyder att stanna (Merriam-Webster dictionary: "Stall: [noun] to bring to a standstill"). Stalla uttalas ungefär "ståla".

5.4 Strukturella konflikter

Strukturella konflikter uppstår som sagt då två pipelinesteg vill nyttja samma resurs. Figur 5.5 illustrerar detta då två instruktioner (STORE och LOAD) samtidigt vill komma åt minnet. Lösningen som visas i figur 5.5(b) bygger på en cykels stallning av pipelinen vid konflikttillfället.



(a) Strukturell konflikt



(b) Konflikt löst med stall

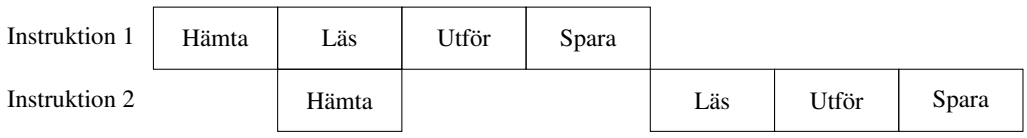
Figur 5.5: Strukturell konflikt med STORE/LOAD

5.5 Datakonflikter

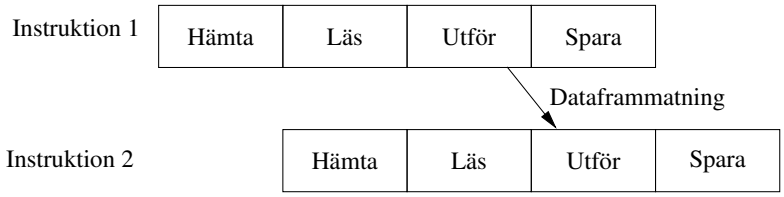
Datakonflikter, det vill säga databeroenden mellan efterföljande instruktioner, kan lösas på två sätt. Det första innebär att man stallar pipelinen tills det efterfrågade datat finns tillgängligt. Den andra principen är att data kan skickas framåt i pipelinen utan att sparas i register (*data forwarding* eller *result forwarding*). Figur 5.6 visar skillnaden mellan dessa två metoder.

5.6 Styrkonflikter

En hoppinstruktion skapar problem i en pipelinad processor då den adress där programmet ska fortsätta exekveringen inte är känd förrän hoppinstruktionen är slutförd. Återigen finns det två primära metoder att lösa detta problem på. Det första är att processorn måste stalla för att ge tid att räkna fram nästa adress. Den andra är att man



(a) Datakonflikt löst med stall



(b) Datakonflikt löst med frammatning

Figur 5.6: Strukturell konflikt med STORE/LOAD

flyttar hoppinstruktionen bakåt i programmet så att ett antal instruktioner efter hoppinstruktionen också utförs innan hoppet görs.

Exempel: I orginalprogrammet nedan körs först instruktionerna `I0-I3` följt av hoppinstruktionen. Beroende på villkoret för hoppet ska antingen `I5` eller `Ix` utföras sedan. Här måste hela hoppinstruktionen utföras innan nästa instruktion (d.v.s `I5` eller `Ix`) kan hämtas. Om fördröjt hopp används så kan flödet fortsätta ohejdat då processorn har flera mellanliggande instruktioner för att beräkna hoppadressen. Dock kan det bli struligt om man vill nyttja resultatet från `I3` för att bestämma villkoret.

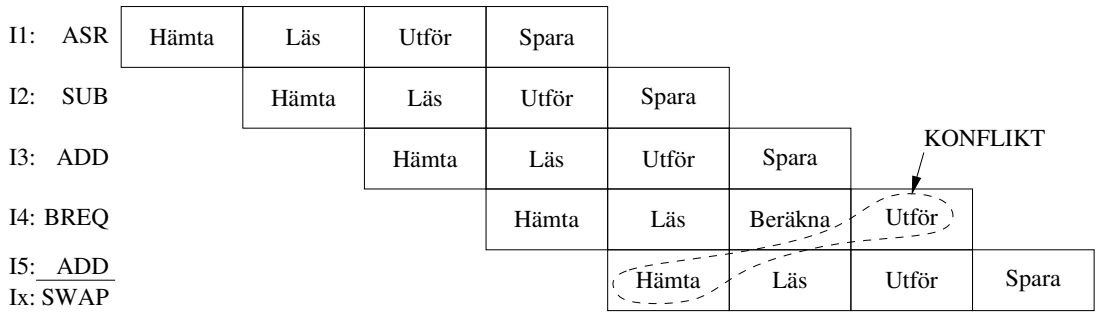
Orginalprogram

```
I0: ADD
I1: ASR
I2: SUB
I3: ADD
I4: BREQ Ix
I5: ADD
  ...
Ix: SWAP
```

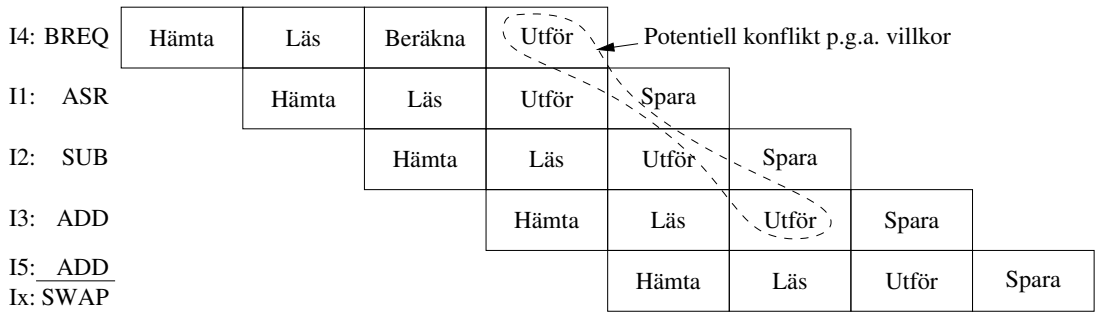
Fördröjt hopp

```
I0: ADD
I4: BREQ Ix
I1: ASR
I2: SUB
I3: ADD
I5: ADD
  ...
Ix: SWAP
```

Denna metod illustreras i figur 5.7. I figur 5.7(a) försöker processorn att utföra ett hopp som fjärde instruktion. I och med att det tar några klockcykler att räkna fram nästa instruktions adress så kommer processorn att bli tvungen att stalla (i tre klockcykler) för att lösa konflikten. Om processorn i stället nyttjar ett fördröjt hopp på tre instruktioner så kommer de tre instruktionerna närmast efter hoppinstruktionen att ut-



(a) Styrkonflikt



(b) Styrkonflikt löst med förskjutet hopp

Figur 5.7: Styrkonflikt på grund av hoppinstruktion

föras före hoppet görs. Figur 5.7(b) visar det omorganiserade instruktionsflödet där hoppinstruktionen är klar när hoppadressen behövs.

Här finns det ett potentiellt problem i att villkoret för hoppet måste vara färdigt när hoppinstruktionen ska utföras. Detta kan dock lösas genom att specificera att instruktionen före hoppinstruktionen ger villkorsflaggorna till hoppet även i det fördröjda fallet. I ett normalt programflöde bestäms hoppvillkoret i regel utav den sista instruktion före hoppet som påverkar flaggorna. I ett fördröjt hopp fungerar inte detta eftersom villkoret måste vara tillgängligt åtminstone före utförandefasen av hoppinstruktionen. Därför går det inte att låta `I3` styra det fördröjda hoppet i exemplet ovan. I stället får `I0` nyttjas för att beräkna villkoret för hoppet. Detta är dock ganska rättframt då effekten blir att villkoret (senast) kommer från instruktionen före hoppinstruktionen.

5.7 Konfliktfri kod

Om processorn antar att koden är fri från konflikter så betyder det att all kontroll av kodens riktighet är upp till programmeraren. Detta gör att det generellt är svårt att programmera sådana processorer i assembler. Om programmeringen i stället utförs i ett högnivåspråk så faller det på kompilatorn att se till att den genererade assemblerkoden är korrekt vilket är betydligt mera praktiskt.

Exempel: En pipelinad processor antar att koden är konfliktfri. ALU-instruktioner tar dock tre klockcykler på sig innan resultatet är skrivet till målregistret medan processorn startar exekveringen av en ny instruktion varje cykel. Anta att vi har följande enkla kodsnutt:

```
1:          add D1, D2
2:          add D0, D1
```

Det logiska beteendet är att efter dessa två instruktioner ska `D0` innehålla resultatet av beräkningen `D0+D1+D2`. Men resultatet av den första instruktionen skrivs inte förrän den andra har läst operanderna vilket gör att resultatet i `D0` kommer att vara `D0+D1`. För att få det "korrekta" beteendet måste man lägga in en eller flera andra instruktioner (t.ex. `nop`) mellan de två additionerna. Hur många instruktioner som krävs mellan additionerna bestäms av pipelinens längd, när operanderna läses från registren samt när resultatet skrivs.

Kapitel 6

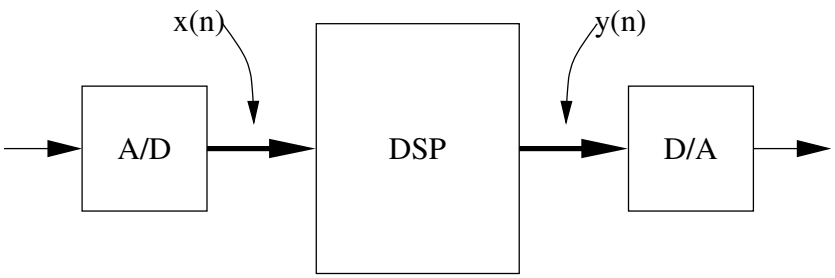
DSP

De processorer som hittils har behandlats i kursen är så kallade generella processorer. Detta innebär att det inte har tagits någon hänsyn till en direkt applikation vid konstruktion av processorer och instruktionsuppsättningar. Generella processorer är skapligt bra i de flesta applikationer men i regel inte extremt bra i någon applikation. Till exempel är en generell processor inte särskilt effektiv vad gäller signalbehandling. I dessa fall får konstruktören antingen använda en ren hårdvarulösning eller en applikationsspecifik processor. Den mest tydliga kategorin av applikationsspecifika processorer är digitala signalprocessorer (*digital signal processor* eller DSP).

6.1 Grundläggande signalbehandling

Ett exempel på applikationsområde där de generella processorerna inte är adekvata är alltså signalbehandling (filter, transformer, etc.). Ett blockschema för ett typiskt signalprocessningssystem återfinns i figur 6.1. En analog insignal samplas i en A/D-omvandlare och samplen skickas till en DSP. Samplen som kommer ut från DSPn skickas via en D/A-omvandlare till en analog signal.

Den primära anledningen till att en generell processor är ineffektiv i denna domän är att dessa applikationer karakteriseras av en stor andel operationer av typen



Figur 6.1: Typiskt signalbehandlingssystem

multiplikation-addition. En sådan applikation är ett FIR-filter (*finite impulse response*) som beskrivs av följande ekvation.

$$y_n = \sum_{i=0}^{N-1} c_i \cdot x_{n-i}$$

Samplen som kommer in till filtret betecknas x_n medan utgående sampel betecknas y_n. c_i är de så kallade filterkoefficienterna. För varje nytt sampel ska alltså den ovanstående ekvationen beräknas vilket innebär N stycken multiplikationer och $N-1$ stycken additioner. Utöver detta krävs det lite kontrollinstruktioner för att styra den loop som lämpligtvis används. För att beräkna ett utsampel skulle programmet kunna se ut som följer.[1]

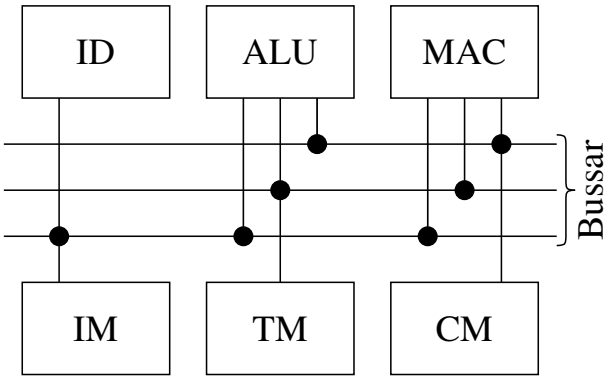
```
1:         clr D1
2:         mov #N, D2
3:  loop:  mov (CMEM+), D0
4:         mul (XMEM+), D0
5:         add D0, D1
6:         dec D2
7:         brne loop
8:         ..... Använd resultatet i D1
```

Varje iteration i loopen utgörs alltså av fem instruktioner varav två är till enbart för loopens skull. Vi ser också att en von Neumann-arkitektur inte kan bli särskilt effektiv då en iteration av summaberäkningen kräver två operander från minnet tillsammans med ett antal instruktionshämtningar. För att motverka dessa problem byggs DSPer med en speciell minnesarkitektur samt med speciella instruktioner och hårdvarustöd för loopar.

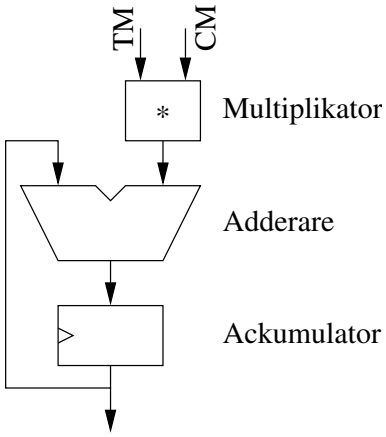
6.2 Hårdvaruarkitektur

För att undvika problem på grund av minnesåtkomster byggs DSPer med flera minnen enligt en (något modifierad) Harvard-arkitektur. Det första steget är att separera instruktionsminne och dataminne. Detta visar sig dock inte vara tillräckligt då operationer av typen multiplicera-addera som används i det ovan visade filtret kräver två "data" per operation. Därför har DSPer i regel minst två separata dataminnen så att två operander kan hämtas per klockcykel. Figur 6.2 visar ett sådant treminnessystem. De tre nedre blocken är minnena. IM är instruktionsminnet, TM är sampelminnet (ofta kallat *tap memory*) och CM är koefficientminnet (*coefficient memory*). De tre övre blocken

[1] Under förutsättning att sampel- och koefficientpekarna, XMEM oh CMEM, är korrekta vid programmets start samt att insamplet har sparats.



Figur 6.2: Typisk arkitektur för en DSP



Figur 6.3: Förenklat blockschema för en MAC-enhet

är funktionella enheter, från vänster instruktionsdekodern, ALUn och multiplikation-additionsenheten (*multiply and accumulate*). Genom att ha separata bussar för samtliga minnen så möjliggörs exekvering med samtidig hämtning av en instruktion och två operander. DSPer byggs (i princip) alltid enligt RISC-principer vilket innebär att alla instruktioner tar lika lång tid samt att hela processorn är pipelinad. Datormodellen för en DSP är med andra ord ungefär samma som för en RISC-processor fast med fler minnen.

6.3 Speciella instruktioner i DSPer

6.3.1 Multiply and accumulate

Som visat i exemplet ovan är multiplikation-addition en vanlig operation. Detta innebär att det är den första uppenbara kandidaten för en speciell instruktion. Denna instruktion kallas vanligtvis för *multiply and accumulate* eller MAC. Eftersom en sådan instruktion inte får ta mer tid än någon annan instruktion så krävs det en hårdvarumultiplikator. Figur 6.3 visar uppbyggnaden av en förenklad MAC-enhet. I en verklig implementation krävs mer hårdvara (främst i form av multiplexrar) för alla extra funktioner såsom separat multiplikation eller addition, nollställning av ackumulatorn, etc.

MAC-instruktionen har tre operander - två faktorer till multiplikationen samt ackumulatorn. För att multiplicera värden från sampelminnet och koefficientminnet samt addera till ackumulatorn kan instruktionen se ut som följer.

```
mac (TM+), (CM+), A
```

6.3.2 Loopinstruktioner

För att undvika det tidigare visade problemet med ett antal extra instruktioner att exekvera i varje loopiteration så används speciella loopinstruktioner med hårdvarustöd i en DSP. Oftast finns det två typer av loopinstruktioner, `Repeat` och `Loop`[2]. Repeat används för en loop med bara en instruktion och kräver bara ett argument för att visa antalet interationer. Loop kan användas för att interationer över mer än en instruktion.

Exempel: Ett FIR-filter enligt ovan med 16 koefficienter kan beskrivas med följande kod:

```
      // Antag att pekarnas värden är korrekta
1:    clr    A               // Nollställ ackumulatorn
2:    repeat #16             // Kör nästa instr 16 ggr
3:    mac    (TM+), (CM+), A // Filter!
```

Dessa tre instruktioner gör nu exakt samma sak som programexemplet för filtret ovan. Skillnaden är alltså att det krävs endast 18 instruktioner för att utföra samma sak som i den generella processorn krävde 82 instruktioner[3]. Dessutom har mängden programminne som krävs minskat från sju till tre ord. Det är mycket vanligt med DSP-system i handhållna apparater, t.ex. mobiltelefoner. I dessa applikationer där låg effektförbrukning är viktigt är båda dessa förbättringar mycket trevliga.

Som en illustration av denna effektbesparing kan vi göra en enkel jämförelse. Effektförbrukningen hos digitala kretsar följer en enkel lag:

$$P = K \cdot f \cdot V_{dd}^2 \tag{6.1}$$

[2] Vissa processorer har bara en instruktion för båda funktionerna.

[3] 16 iterationer i loopen med fem instruktioner plus två instruktioner för förberedelser.

där f är klockfrekvensen och V_{dd} är matningsspänningen. Om vi antar att K är oförändrad[4] så blir effekten linjärt beroende på klockfrekvensen. Behövs det då - som i exemplet ovan - 18 instruktioner istället för 82 så minskar effekt-förbrukningen med ca 79% för detta filter. Dessutom ger den lägre frekvensen möjlighet att minska matningsspänningen till processorn vilket ger ännu större effektbesparing.

6.4 Talområden och precision i MAC-enheten

Som visades vid föreläsningarna i binär aritmetik så åstadkommer en multiplikation av två tal en dubblering av ordlängden. Detta faktum är giltigt oavsett talbas. Till exempel kan talen i den sista raden i följande exempel antingen tolkas som decimala eller binära. Värdet på talet i sista raden är naturligtvis beroende på vilken talbas som väljs. I det binära fallet är faktorerna värda 0.5_{10} och produkten är värd 0.25_{10}.

$$\begin{aligned} 9 \cdot 9 &= 81 \\ 99 \cdot 11 &= 1089 \\ 0.1 \cdot 0.1 &= 0.01 \end{aligned}$$

På samma sätt orsakar en addition en möjlig förlängning av ordlängden med ett[5].

$$\begin{aligned} 9 + 9 &= 18 \\ 99 + 11 &= 110 \\ 0.1_2 + 0.1_2 &= 1.0_2 \end{aligned}$$

Fixtals-DSPer[6] använder i regel ett fraktionellt format där såväl ordlängden som binärpunkten är fasta. Den fasta ordlängden brukar kallas för *native word length*. Det fasta formatet innebär oftast att binärpunkten placeras direkt efter teckenbiten och att den mest signifikanta biten (näst efter teckenbiten) har värdet 0.5_{10}. Om resultatet i ett sådant talsystem bara kapas till den ursprungliga längden kommer det att uppstå avrundningsfel efter multiplikationer och möjlig overflow vid addition.

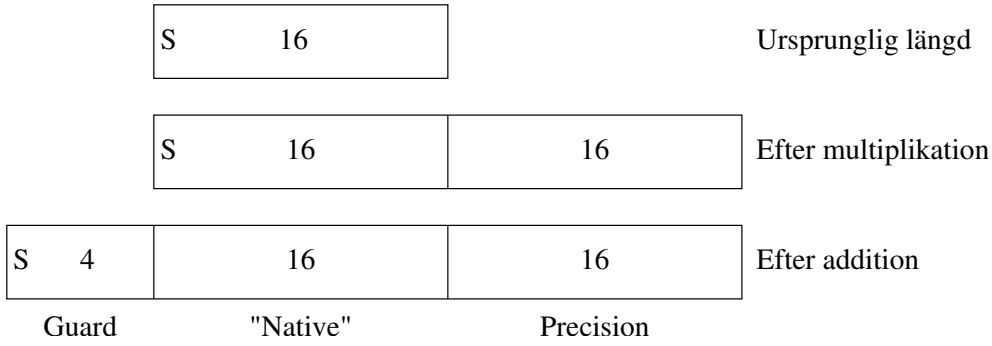
För att motverka dessa problem så nyttjar en fixtals-DSP alltid en större ordlängd internt i MAC-enheten. Dels ger multiplikationen dubbla ordlängden plus att man lägger till några bitar i den mest signifikanta ändan för att undvika overflow efter additionen.

Figur 6.4 visar ett exempel på ordlängder som lagras i ackumulatorn hos en fraktionell fixtals-DSP. Med fraktionella tal så kommer den mest signifikanta delen av resultatet från multiplikationen att motsvara det talområde som faktorerna har. För att

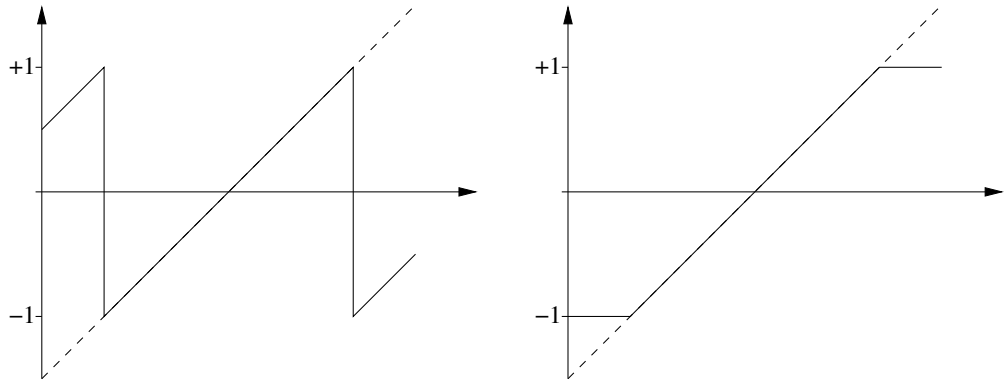
[4] K beror på saker såsom aktivitet, d.v.s. hur många signaler som ändras varje klockcykel, och fysisk konstruktion av kretsen. K är därför i praktiken inte helt oberoende av de övriga parametrarna.

[5] Om man adderar ett godtyckligt antal (N) binära tal med samma ursprunglig ordlängd så kommer den total förlängningen an ordlängden att uppgå till $\lceil log_2 N \rceil$.

[6] Till skillnad från andra typer av DSPer som använder flyttal!



Figur 6.4: Ordlängd hos ackumulatorn i MAC-enheten hos en fraktionell fixtals-DSP. S markerar teckenbitens position.



(a) Talområde med trunkering (b) Talområde med mättnad

Figur 6.5: Talområden med och utan mättnad (saturation)

motverka overflow läggs dessutom ett antal så kallade guardbitar till före den mest signifikanta biten i talet. I detta exempel så blir ackumulatorns total ordlängd 36 bitar inklusive fyra guardbitar.

Vid slutet på en beräkning av till exempel ett filter måste värdet i ackumulatorn tas ut för att skickas vidare eller lagras. Eftersom ackumulatorn är för stor måste talet justeras till den ordinare ordlängdens talområde. Precisionsbitarna hanteras enkelt med en avrundning som utförs med en addition med ett till den mest signifikanta precisionsbiten. Denna addition motsvarar alltså en addition med en halv minst signifikant bit i det ursprungliga talområdet.

Den mest rättframma metoden att bli av med guardbitarna innebär en trunkering (avkapning) av dessa. Detta innebär dock att för stora eller små tal kommer att vikas in i det ordinarie talområdet vilket ger upphov till distortion i den filtrerade signalen.

För att minska detta problem används mättnad (*saturation*) i talområdet där ett tal som är för stort ändras till det största tal som kan representeras och ett tal som är för litet ändras till det minsta tal som kan representeras[7]. Det är naturligtvis fortfarande möjligt att få distortion på grund av att utvärdet är "fel" men felet blir i alla fall så litet som möjligt. Dessa två principer illustreras i figur 6.5. X-axeln visar talområdet för hela talet med guardbitar och y-axeln visar talområdet utan guardbitar. Figur 6.5(a) påvisar problemet där tal som är lite för litet blir maximalt stort och tvärtom medan figur 6.5(b) visar talområdet med mättnad.

Exempel: Antag ett system med en guardbit, en teckenbit och två ordinarie bitar. Invikning av talen kommer att ske vid direkt trunkering av guardbiten. Detta problem minskar med mättnadsfunktion vid borttagningen av guardbiten.

Ursprungsvärde		Trunkering			Mättnad		
Binärt	Decimalt	Binärt	Decimalt	Fel	Binärt	Decimalt	Fel
00.11	0.75	0.11	0.75	0.00	0.11	0.75	0.00
01.11	1.75	1.11	-0.25	2.00	0.11	0.75	1.00
11.11	-0.25	1.11	-0.25	0.00	1.11	-0.25	0.00
10.00	-2.00	0.00	0.00	2.00	1.00	-1.00	1.00

[7] Jämför med A/D-omvandlare av typen reversibel räknare. Vad händer om den analoga insignalen är för hög så att komparatorn aldrig signalerar till räknaren att räkna ner? Om räknaren är av mättnadstyp (det vill säga inte slår om till noll efter det maximala talet) så kommer den att stanna på maxvärdet vilket är betydligt mer rätt än alternativet att slå om till noll för att sedan börja om att räkna upp.

Kapitel 7

Programmeringsmetodik

Det är nästan aldrig enkelt att implementera ett program på ett bra sätt. För att kunna åstadkomma en effektiv, enkel och lättförståelig programkod krävs det att man är någorlunda strikt i sin metodik. Detta är speciellt viktigt när man programmerar på assemblernivå då det blir svårare att ha god översikt över koden. I det här kapitlet presenteras en mycket förenklad metod för att kunna överföra en specifikation till assemblerkod som är lämplig att använda på laborationerna.

7.1 Metodik för assemblerprogrammering

1. **Specifikation:** Bestäm *vad* som ska göras! (Detta är naturligtvis redan gjort för labbarna.)

2. **Design:** Dela upp specifikationen i mindre funktioner och bestäm vad dessa ska göra. (Eftersom labbarna utgörs av relativt små och enkla program är detta steg oftast inte nödvändigt.)

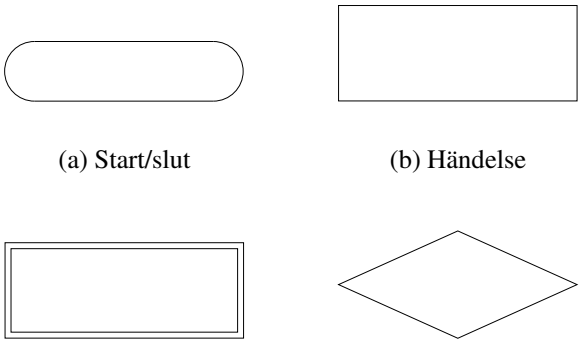
3. **Detaljdesign:** Bestäm kontrollflöde och rita flödesschema för varje funktion.

4. **Implementation:** Översätt varje flödesschema till assemblerkod.

5. **Integration:** "Koppla ihop" samtliga funktioner. (Återigen inte särkilt nödvändigt på labbarna.)

Med kontrollflöde menas hur programmet anropar olika funktioner och hur flödet ser ut inne i funktionerna.

I varje steg - speciellt de två sista - är det viktigt att resultatet kontinuerligt testas mot specifikationen. I ett scenario med lite mer komplicerade program (t.ex. i kursen Elektronikprojekt Y) så är detta extremt viktigt eftersom ett fel någonstans kan påverka resten av programmet på mer eller mindre förutsägbart sätt.



(a) Start/slut (b) Händelse

(c) Hierarkisk händelse (d) Beslut

Figur 7.1: Olika typer av flödesschemasymboler

7.2 Flödesschema

Ett flödesschema är en enkel metod att representera kontrollflödet i en process. Processen kan vara ett industriellt problem, t.ex. hur en maskin fungerar, men det kan också vara ett program. Ett flödesschema byggs upp av olika grafiska "boxar". De vanligaste typerna av "boxar" återfinns i figur 7.1.

Boxarna i flödesschemat kopplas ihop med pilar som visar hoppen mellan boxarna. I de fall där komplicerade flödesscheman ska ritas kan man nyttja boxen för den hierarkiska händelsen. Denna innebär helt enkelt ett hierarkiskt anrop av ett annat flödesschema (på samma sätt som ett subrutinanrop).

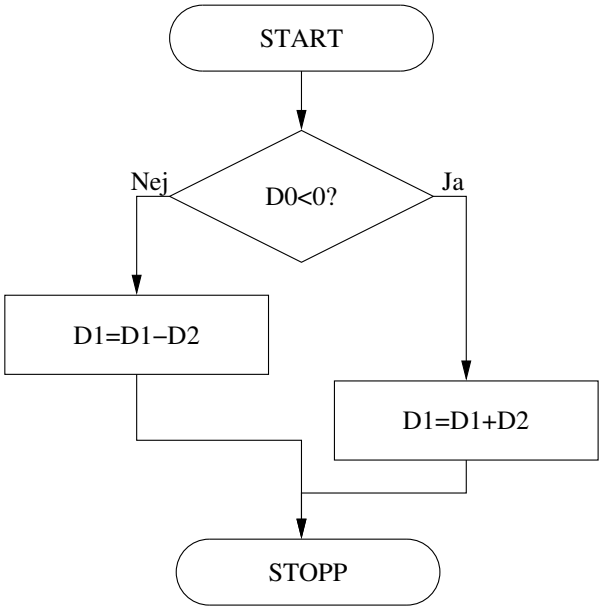
Flödesschemats vara eller inte vara vid mjukvarudesign har debatterats mycket under åren. För lågnivåprogrammering (d.v.s. assemblernivå) är det en överlägsen metod[1]. Flödesscheman används ofta för specifikation av funktioner när den typen av programmering används.

Exempel: Antag att en funktion (subrutin) som antingen adderar eller subtraherar två tal beroende på ett annat tal ska implementeras. Om det styrande talet (som finns i register `D0`) är positivt ska subtraktionen `D1=D1-D2` utföras. Om det är negativt ska additionen `D1=D1+D2` utföras.

Ett flödesschema för detta återfinns i figur 7.2(a). Varje del i flödesschemat kan enkelt överföras till assembler. Ett beslut eller en händelse överförs direkt till motsvarande kod. Pilarna motsvaras på samma sätt av hopp mellan olika assemblerkodsnuttar[2]. Den assemblerkod som motsvarar flödesschemat återfinns i figur 7.2(b).

[1] För högnivåprogrammering är det inte helt lyckat att använda flödesscheman då dessa kan bryta mot de regler som ställs upp i högnivåspråken.

[2] I formella metoder för att översätta flödesscheman till kod skulle samtliga pilar översättas till hopp även om dessa inte skulle behövas. Onödiga hopp skull sedan optimeras bort i ett senare steg.



(a) Flödesschema

```
function:
            tst D0
            bmi addition

            sub D2,D1
            bra funcend
addition:   add D2,D1

funcend:    rts
```

(b) Assemblerkod

Figur 7.2: Exempel på flödesschema

Referenslista

Som bakgrundsmaterial och referenser till detta kompendium har ett antal källor använts. Här kommer en någorlunda kompett lista över dessa i bokstavsording:

- Alfred V. Aho, Ravi Sethi och Jeffery D. Ullman, *Compilers: Principles, techniques, and tools*, Addison Wesley 1988, ISBN 0-201-10088-6
- Stephen A. Campbell, *The science and engineering of microelectronic fabrication*, Oxford university press 1996, ISBN 0-19-510508-7
- Alan Clements, *The Principles of Computer Hardware*, Oxford University Press 2000, ISBN 0198564538
- Per-Erik Danielsson och Lennart Bengtsson, *Digital teknik*, Studentlitteratur 1996, ISBN 91-44-00110-X
- John L. Hennessy och David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann 2002, ISBN 1558605967
- David A. Johns och Ken Martin, *Analog integrated circuit design*, Wiley 1997, ISBN 0-471-14448-7
- Dake Liu, *Design of embedded DSP processors*, Linköpings universitet 2003
- Micron Technology, Inc., Datablad för *MT47H64M4FP-5E*, http://www.micron.com, 2004
- Gordon E Moore, *Cramming more components onto integrated circuits*, Electronics Magazine, Volume 38, nummer 8, 19 April 1965
- Ramtron International Corporation, *About FRAM*, http://www.ramtron.com, 2004
- Olof Roos, *Grundläggande datorteknik*, Studentlitteratur 1995, ISBN 91-44-46651-X
- Semiconductor Indusry Association, *International technology roadmap for semiconductors*, http://public.itrs.net, 2001

- Steven W. Smith, *The scientist and engineer's guide to digital signal processing*, Analog Devices 1999, ISBN 0-9660176-4-1
- William P. Stallings, *Computer Organization and Architecture: Designing for Performance*, Prentice Hall 2002, ISBN 0130493074