

TSEA28 Datorteknik Y (och U)

Föreläsning 11

Kent Palmkvist, ISY



Dagens föreläsning

- Kort titt på 68000 implementationen
- Generella egenskaper hos CISC
- Alternativ struktur: RISC
- Introduktion till pipelining

Praktiska kommentarer

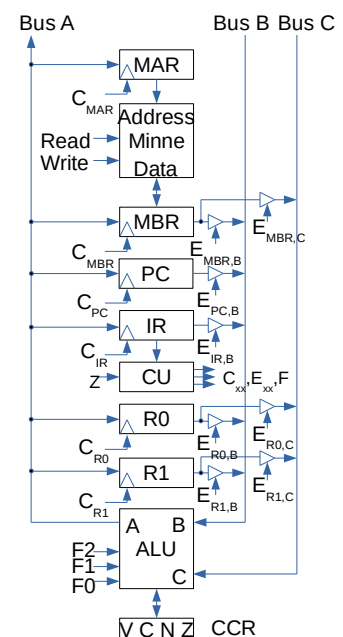
- Labutrustning i MUX1 har tagits bort
 - Labbet används i andra kurser
 - Distansutrustning kommer finnas kvar i MUX2
 - Extra redovisningstillfälle lab 1-3 10/6 kl 9 i mux1
- Laboration 4 simuleringslabb
 - Arbeta ensam eller i grupp om max 2 personer
 - Redovisning i labb, kräver dokumenterad lösning
 - Exempelvis via excelblad från hemsidan
 - Räcker inte med binärdata i lma
 - Körs på linux
 - Asgård, olympen, egypten, SU25, mux1, mux2, thinlinc
 - lma kräver stor skärm eller sänkt skalningsfaktor

Processorarkitektur inkl CU

- Exempel på maskininstruktioner som ska kunna utföras

Op-code	Namn	Operation
0 0 0	LOAD R0,M	[R0] <- [M]
0 0 1	LOAD R1,M	[R1] <- [M]
0 1 0	STORE M,R0	[M] <- [R0]
0 1 1	STORE M,R1	[M] <- [R1]
1 0 0	ADD R1,R0	[R1] <- [R1] + [R0]
1 0 1	SUB R1,R0	[R1] <- [R1] - [R0]
1 1 0	BRA T	[PC] <- T
1 1 1	BEQ T	Om [Z] = 1 då [PC] <- T

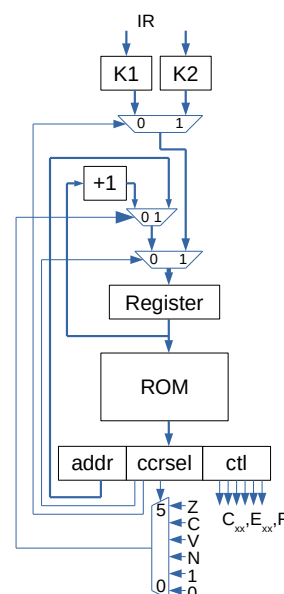
- CU innehåller mikroprogrammet som genererar alla styrsignaler
- Addresseringsmode och opcode specialhanteras i CU



Kontrollenhetens (CU) inre

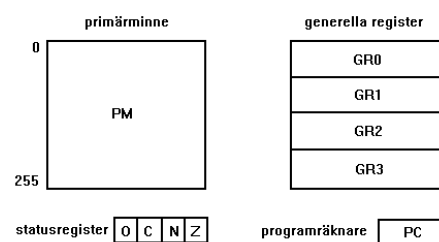
uaddr	AddrSel	kontrollsignal	Addr	
0	+1	$E_{PC,B} = 1, F2F1F0 = 0,0,0, C_{MAR}$	-	; Fetch
1	+1	$E_{PC,B} = 1, F2F1F0 = 0,1,0, C_{PC}$	-	
2	+1	Read=1, C_{MBR}	-	
3	+1	$E_{MBR,B} = 1, F2F1F0 = 0,0,0, C_{IR}$	-	
4	K1	-	-	
k (5)	+1	$E_{R0,B} = 1, F2F1F0 = 0,0,0, C_{MBR}$	-	; STORE
k+1 (6)	+1	$E_{IR,B} = 1, F2F1F0 = 0,0,0, C_{MAR}$	-	
k+2 (7)	1	Write=1	0	
l (8)	Z	-	l+2	; BEQ
l+1 (9)	1	-	0	
l+2 (10)	1	$E_{IR,B} = 1, F2F1F0 = 0,0,0, C_{PC}$	0	

0	00000 0000 000001 01000000 000 00	}	Varje rad i ROM
1	00000 0000 000100 01000000 010 00		5 bit addr
2	00000 0000 000000 00000000 000 10		4 bit ccrsel
3	00000 0000 001000 10000000 000 00		6 bit Cxx
4	00000 1000 000000 00000000 000 00		8 bit Exx
5	00000 0000 000010 00010000 000 00		1 bit Read
6	00000 0000 000001 00100000 000 00		3 bit Fx
7	00000 0001 000000 00000000 000 01		1 bit Write
8	01010 0101 000000 00000000 000 00		
9	00000 0001 000000 00000000 000 00		
10	00000 0001 000100 00100000 000 00		



Processorarchitektur Laboration 4 (MIA)

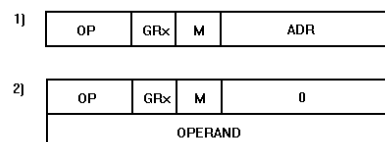
- Simuleringsmodell för övning på mikroprogrammering
- Fördefinierad datorarkitektur med en databuss
- Maskinspråksmodell
 - Primärminne (256 ord a 16 bitar)
 - 4 generella register
 - PC (Program Counter)
 - 4 flaggor (OCNZ)
- 16-bitars databuss, 8-bitars adressbuss



MIA maskininstruktionsformat (instruktioner i PM)

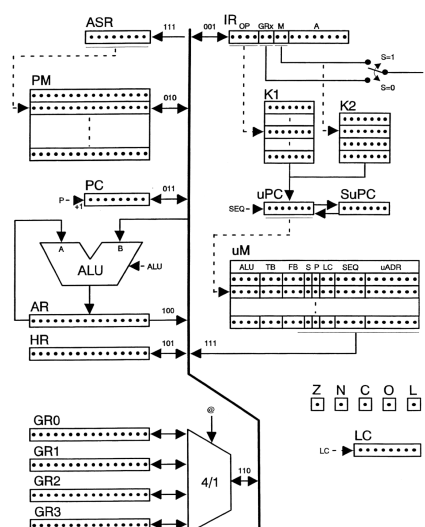
- 2 Instruktionsformat
 - Ett 16-bitars ord
 - Ett 16-bitars ord samt en 16 bitars operand
 - Inte samma som i exemplen tidigare!
 - Uppbyggnad av maskininstruktionsordet
 - OP anger operation (4 bit)
 - GRx angiva vilket generellt register ska användas (2 bit)
 - M anger adresseringsmode (2 bit)
 - ADR minnesaddress (8 bit)
- | | |
|----|----|
| 1) | OP |
|----|----|

2)	OP



Laboration intern arkitektur mikroprogrammering (MIA)

- 1 delad databuss
 - 1 sändare och 1 mottagare per överföring på bussen
 - Vertikal mikroprogrammering
- Inget MBR register
- IR innehåller aktuell maskininstruktion som utförs
- Mikroprogram i uM
 - Genererar styrsignaler till hela strukturen
- En loopräknare LC för mikrokodsloopar
 - Text för LSR i uppgift 1 i lab 4



Mikroinstruktionsformat

- ALU styr ALU:ns funktion
- TB anger källan till bussens värde
- FB anger vart bussens värde sparas
- S Välj M eller GRx fält i maskininstruktionen
- P Räkna upp PC
- LC Loopräknare i mikrokoden
- SEQ styr eventuella hopp i mikrokoden
- UADR måladdress vid vissa hopp i mikrokoden

ALU	TB	FB	S	PLC	SEQ	uADR
-----	----	----	---	-----	-----	------

Styrenhetens (CU) styrsekvenser

- 3 faser för utförande av en maskininstruktion
 - 1: hämta instruktion till IR
 $IR = PM(PC++)$
 - 2: Beräkna effektiv adress
 Beroende på M-bitar i IR (dvs adresseringsmode):
 - 00 (direktadressering) $ASR = ADR$
 - 01 (omedelbar) $ASR = PC++$
 - 10 (indirekt) $ASR = PM(ADR)$
 - 11 (indexerad) $ASR = GR3 + ADR$
 - 3: Utför operationen
 - En av 16 olika möjliga sekvenser (bestäms av OP-fältet i IR)

Praktiska kommentarer, förberedelsetid

- Min lösning av sorteringsalgoritmen i lab4 tog mig 3-4h att förbereda
 - Jag är van, räkna med mer tid för att lösa uppgiften
- Metodik (samma som för assembly programmering typ lab1-3)
 - Beskriv flödesdiagram (speciellt för mer avancerade instruktioner)
 - Beskriv funktion med pseudokod
 - (Kompaktera kod, dvs sätt flera aktiviteter i samma instruktion)
 - Placera i sekvens, bestäm adresser för varje instruktion
 - Översätt till mikrokod (använd excelblad på labhemsidan)
- Min 1:a sorteringsalgoritm med dataexemplet från labanvisningarna: 40391 klockcykler
 - Går att göra mycket bättre (klaras ni det?)

Praktiska kommentarer, sorting

- Behöver sortera baserat på värde enligt 2-komplementsrepresentation
 - Bit längst till vänster i registret (N-flaggan) anger om talet är negativt
 - Kräver att summan/differensen får plats i ett register för att N-flaggan ska stämma
 - Kan få spill (svar som går utanför möjligt talområde)
- Ex: 8-bitars register, addition talen $0x42 + 0x41$
 - 8-bitars talområde tvåkomplement $\Rightarrow [-128, 127]$
 - $0x42 = 66$, $0x41 = 65$, $0x42 + 0x41 = 66 + 65 = 131$! Utanför talområde
 - $0x42 + 0x41 = 0x83$. Tolkas som värdet $-128 + 2 + 1 = -125$ enligt 2-komplement
 - Flaggor visar $N=1$ (MSB bit = 1) samt flagga $O=1$ visar på spill

Hur beskriva mikrokod på tentan (och lab)

- Ange alltid källa och mål för alla dataflyttar
 - Mindre risk för fel om t ex `ASR:=buss`, `buss:=PC` istället för `ASR:=PC`
- Ange vad som gäller för uPC (+1 eller annat)
- Ange alltid om ALU-operationer ska påverka flaggorna (kan styras i vissa fall)
- Glöm inte S-biten (om GRx används)
- Lägg till kommentarer
 - Kan ge fler poäng på uppgiften även om koden inte riktigt korrekt
- Titta på svaren till gamla tentor
 - Finns på webben

Jämförelse maskinkod och mikrokod

- Programmeringsmässigt ganska lika
 - Kod tolkas steg för steg, hopp kan göras i sekvensen, alla instruktioner ligger på en adress i ett minne
- Mikrokod mer parallell till sin natur
 - Kan få flera saker att utföras i samma klockcykel
- Maskinkod mer generell
 - Arkitekturen internt i processorn kan ändras utan att maskinkoden behöver skrivas om
 - Kan återanvända all tidigare skriven och kompilerad programvara
 - Färre detaljer att hålla reda på

RISC och CISC

Designmål för tidiga mikroprocessorer (< 1990)

- Hårdvara var dyrt, speciellt minne var dyrt
 - IBM PC: Mer än 640 kByte RAM behöver man inte...
 - Instruktionsorden och registren var korta (8 eller 16 bitar långa)
- Programmering gjordes ofta i assembler
 - Låt varje instruktion göra så mycket som möjligt
 - Förenkla programmering om möjligt
 - Numera används högnivåspråk (kompileras)
- Få register, enkel dataväg, få olika instruktioner
 - Avkodning inte allt för svår
- Bakåtkompatibla
 - Tidigare skriven kod fungerade även på nya processorer

Vidareutveckling från 16 bitars datorer och framåt

- Fler interna register
 - Fler styrsignaler behövs
- Fler allt mer komplexa instruktioner och addressmoder (exempel från 68000)
 - MOVM.L D1-D3/A0-A2, -(A7) ; flytta D1-D3, A0-A2 till stacken
 - DBEQ D0,addr ; test condition, if not true decrement register and branch
- Allt mer och längre instruktionssekvenser
 - Större och längre mikrokod
- Totalt sett ger det all mer styrlogik/mikroprogramminne
- Känd som CISC (Complex Instruction Set Computer)

Analys av 68000

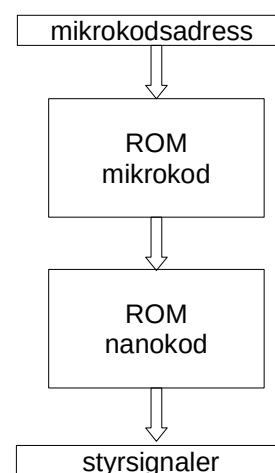
- Processor i Amiga 500
 - Ganska populär hemdator under 80 och 90-talen
 - Samtida ungefär med första IBM PC (1987)
- Processorn användes även av Apple i Macintosh-datorer
- Beskrivning baserad bl a på "reverse engineering"
 - Görs även idag, finns många webbsiter med intressanta bilder och analyser från aktuella designer
- Motsvarande patent och vetenskapliga tidningsartiklar finns även för aktuella processordesigner
- 70% av ytan till avkodning/kontroll av processor (CU-delen)
 - 30% utför beräkningar/flytt av data

Detaljerad case study

- Se separata slides (se föreläsningshemsidan)
 - TSEA28_Lecture11_68000.pdf
 - Sammanställt av Andreas Ehliar (förra föreläsaren i kursen)
- Mycket bilder av fysisk chip (reverse engineered)
- Patent och publikationer innehåller en hel del information
- Främst till för den som är extra intresserad

Exempel på gjorda optimeringar

- Nano-kod
 - Ofta samma kontrollsignal på många platser i mikrokoden
 - Alla möjliga kombinationer används aldrig
 - Olika sekvenser
 - Reducera mängd mikrokod genom två stegs kontrollsignalgenerering
- Exempel: Antag 200 olika maskin-instruktioner, 4 steg/instruktion, 150 bitar bred mikrokod
 - $200 * 4 * 150 = 120\,000$ bitar minne
- Om endast 120 unika mikrokods-kombinationer
 - 7 bitar ersätter varje originalrad i mikrokodsminnet
 - $7 * 200 * 4 + 150 * 120 = 23\,600$ bitars minne



Analys av prestanda

- CISC (Complex Instruction Set Computer)
 - Stor och komplicerad instruktionsavkodning
 - Långa instruktioner
 - Många klockcykler för varje instruktion
 - Långsamma klockcykler
 - Stora minnen används för avkodning
 - Viss parallellism mellan olika interna operationer
 - Flytta mellan register
 - Räkna upp PC
 - Beräkna minnesadresser

Alternativ för bättre prestanda/effektivitet

- RISC (Reduced Instruction Set Computer)
 - Färre och enklare instruktioner än för CISC
 - Minnesaccess ofta begränsad
 - Färre adresseringsmoder
- Exempel på RISC-processorer
 - ARM (finns i smartphones), MIPS (t ex routrar etc.), RISC-V (t ex i hårddiskar)
- Program tar ofta större plats i programminnet för RISC än för CISC
 - Fler men enklare instruktioner
 - Minne har blivit billigt...

Load-Store arkitekturer

- Endast två instruktioner får tillgång till minne
 - LOAD : läs från minnet
 - STORE: Skriv till minnet
- Oftast ytterligare begränsningar
 - Begränsat antal adresseringsmoder
 - Maximalt ett register ändras per instruktion
- Ofta sedda som RISC-arkitekturer
 - RISC har lite olika betydelse
 - Finns ingen exakt definition
- MIA (lab 4) ej Load-Store, ARM nästan Load-Store (har push/pop)

Load-Store arkitektur, forts.

- Exempel på instruktioner som INTE finns i en typisk RISC
 - ; Flyttar värde mellan minnesadress 0x10080 till adress 0x4000
`move 0x10080,0x4000` ; två minneåtkomster!
 - ; läs värdet i minnesadress 0x10084, bitvis logisk or med 0x04,
; skriv tillbaks resultat till minnesadress 0x10084
`or 0x10084,#0x04` ; läs och skriv till minne, inte bara skriv eller läs
- Eventuellt möjlig RISC instruktion
`move.b (a0)+,d0` ; två register ändras!

Antal operander

- Färre instruktioner => Färre bitar behövs för opcode-delen av maskininstruktionen
 - Fler bitar tillgängligt för annat
 - Ökad ordbredd på processor ger fler tillgängliga bitar i maskinkodsordet
- En typisk RISC brukar ofta ta tre register som parametrar till instruktionen istället för två
 - 68000: `add.b d0,d1 ; d1 = d1 + d0`
 - ARM: `add r0,r1,r2 ; r0 = r1 + r2`
 - I detta exempel är det omvänd ordning på destination jämfört med 68000

Villkorliga hopp i RISC

- Villkorliga hopp fungerar ofta som vanligt
 - Uppdatera programräknare om flagga/flaggor stämmer med villkor
- Undantag finns
 - Flaggregister kan saknas, villkor kan istället ligga inbakat i hoppinstruktionen
 - Ex: MIPS, RISC-V: `beq r1,r2,label ; hoppa till label om  $r1=r2$`
- Även villkorlig exekvering av andra instruktioner kan finnas
 - Ex: ARM: `ADDEQ r1,r2,r3`; Om $Z = 1$ utför $r1=r2+r3$

Fördelar med villkorliga instruktioner

- Hittills varit tvungna att hoppa i kodsekvensen för att utföra villkorliga instruktioner
 - Ex: Översatt 4-bitarstal till hexadecimalt tal i ASCII-representation


```
AND R0,#0x0F ; Plocka fram 4 minst signifikanta bitarna
ADD R0,#0x30 ; ASCII-kod för '0' är 0x30
CMP R0,#0x3A ; ASCII-kod för 'A' är 0x41, inte 0x3A
BLT Done ; Klar om 0-9
ADD R0,#0x07 ; 0x3A+7 = 0x41
Done:
```
 - Denna lösning kräver hopp, trots att det bara är en instruktion extra i fallen A-F.
- Villkorliga instruktioner (pseudokod, ej riktig ARM-assembly)


```
AND R0,#0x0F
ADD R0,#0x30
CMP R0,#0x3A
ADD.LT R0,#0x07 ; Instruktion avkodas för alla fall, men påverkar bara R0 om A-F
Done:
```

 - Denna lösning snabbare och färre instruktioner. RISC inte alltid större och långsammare kod!

Subrutiner och länkregister i CISC

- I en CISC används oftast stacken vid subrutinanrop
 - JSR sparar återhoppadress på stack (liknar hur avbrott hanteras i Darna)
 - RTS återställer PC mha data från stacken
 - Dessa instruktioner läser/skriver i minnet (stacken) utan att vara rena LOAD/STORE instruktioner
- JSR subrutin (Jump SubRoutine)
 - Kopiera adress till nästa instruktion (PC:s nya värde) till stacken
 - Hoppa till subrutinen (sätt PC till adressen för subrutin)
- RTS (ReTurn from Subroutine)
 - Läs värde från stack, placera i PC

Subrutiner och länkregister i RISC

- I en strikt load-store-arkitektur får inte JSR/RTS finnas
 - Dessa instruktioner läser/skriver i minnet (stacken) utan att vara rena LOAD/STORE instruktioner
- Alternativ till JSR/RTS: Länkregister
 - Vid subrutinanrop sparas återhoppssadressen i ett länkregister
 - ARM: $lr = r14$
 - Vid återhopp: Sätt $PC = \text{länkregister}$
- Stack kan fortfarande skapas i en load-store-arkitektur
 - Peka på stack med ett av registren
 - Spara i minnet följt av ökning/minskning av registervärdet, exempel:


```
ADD R15,R15,1    ; öka stackpekare med 1
STORE LR,R15     ; Lägg LR värdet på stacken
```

Subrutin med länkregister, exempel

```
subrutin1:
    push lr          ; Spara undan återhoppssadress
    ....
    bl  subrutin2     ; Hoppa och länka aktuell PC (dvs LR = PC)
    ....
    pop  lr           ; Återställ LR och hoppa
    bx   lr           ; tillbaks (hoppa till adress i LR-register)

subrutin2:           ; Denna subrutin anropar inte någon
                     ; annan subrutin. Alltså behöver inte
                     ; lr sparas undan
    ....
    bx   lr           ; hoppa tillbaks
```

- Mer beskrivning av detta hittas i avsnitt 3.8.1 i kursboken

Enklare instruktionsformat

- Enklare instruktionsformat ger enklare avkodning
 - Jfr MIA-systemet
 - Fasta registerfält (avkodar alltid register på samma sätt)
 - Ofta fix längd på instruktionen i RISC
 - MIA-systemet har två olika längder (1 eller 2 ord)
 - Vissa kombinationer onödiga/har samma funktion
 - and r0,r0,r0
 - or r0,r0,r0
 - Svårighet med längd på Omedelbar operand
 - Kortare än ordlängden
 - Flera möjliga lösningar

31	30	26	25	21	20	16	15	0	
0	Opkod		Dest. Reg		Källa1		Omedelbar operand		
1	Opkod		Dest. Reg		Källa1		Källa2		Ej använt
31	30	26	25	21	20	16	15	11	10
0									

Sammanfattning RISC

- Enklare instruktionsformat => Lättare bygga avkodare
 - Mindre logik/yta för avkodare
- Få instruktioner => Lättare bygga avkodare och styrmodul
- Insparat yta/transistorer kan användas bättre
 - Fler eller snabbare exekveringsenheter
 - Fler register
- Alternativt sparad yta => Billigare chip

Datorprestanda

- Klockfrekvens har dålig koppling till prestanda
 - Allt från 1 klockcykel till > 15 klockcykler per instruktion
 - Antal klockcykler varierar med vilken instruktion det handlar om
- Prestandamått MIPS: Million Instructions Per Second
 - Bättre mått än klockfrekvens
 - Fortfarande otydligt: Vad består en instruktion av?
 - RISC behöver fler instruktioner än CISC?
- Bättre mått: SPEC (Standard Performance Evaluation Corporation)
 - Benchmarkprogram fokuserar på tid för att slutföra en definierad uppgift (källkod finns)
 - Fortfarande inte samma som upplevd hastighet hos dator
 - Flera olika delar testas (heltal, flyttal) med olika deluppgifter

Prestandaexempel + area

- Prestandasiffror från boken "Embedded software development with C"

	MIPS:	Antal transistorer
MC68000:	Ca 1 MIPS vid 8 MHz	68000
MC68020:	Ca 4 MIPS vid 20 MHz	200000
80386DX:	Ca 8.5 MIPS vid 25 MHz	275000
ARM2:	Ca 4 MIPS vid 4 MHz	25000
- Moderna processorer
 - Högre klockfrekvenser (> 5GHz)
 - Högre antal MIPS/Mhz (> 1)
 - Större antal transistorer (miljarder)

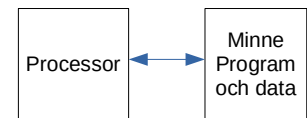
Von Neumann och Harvard

Ytterligare förbättringar

- Analysera sekvens av operationer i en processor
 - Hämta instruktion
 - Avkoda instruktion
 - Hämta operander
 - Utför instruktion (beräkning/flytt)
 - Spara resultat
- De flesta delar i processorn används inte hela tiden
 - Borde gå att hämta nästa instruktion medan föregående instruktion avkodas
 - Inte säkert den kan användas om t ex hopp utförs
 - Idé: Överlappa operationer från olika instruktioner
 - Svårt när allt måste hämtas/skrivas från/till samma minne (instruktion och data)

Datorarkitektur: von Neumann

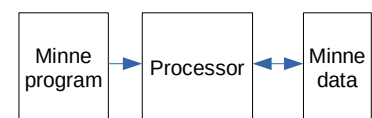
- Idé att överlappa flera instruktioner kallas pipelining
- Förenklas om få olika möjliga sekvenser av styrsignaler
- Förenklas om nästa instruktion kan läsas parallellt med att data läses och skrivs i minnet
 - Går inte med arkitektur som använts hittills: von Neumann.
 - Minnebussen mellan processor och minne är en flaskhals
- Vissa fördelar med von Neumann
 - Enkel programmeringsmodell
 - Behöver inte bestämma i förväg andel program vs data minne
 - Kan hantera program som vilket data som helst



Von Neumann

Datorarkitektur: Harvard

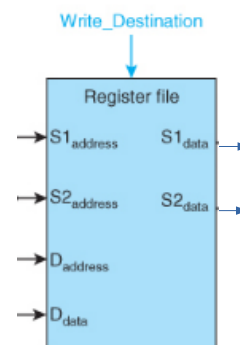
- Ny arkitektur: Dela minnet
 - Programminne för sig
 - Dataminne för sig
 - Känt som Harvard-arkitektur
- Fördelar
 - Kan både läsa programinstruktion samtidigt som data läses
 - Kan inte skriva över programmet av misstag (undviker självmodifierande kod)
- Moderna processorer ibland byggda som Harvard men beter sig som von Neumann
 - Bästa av båda världar



Harvard

Enkel RISC-baserad struktur

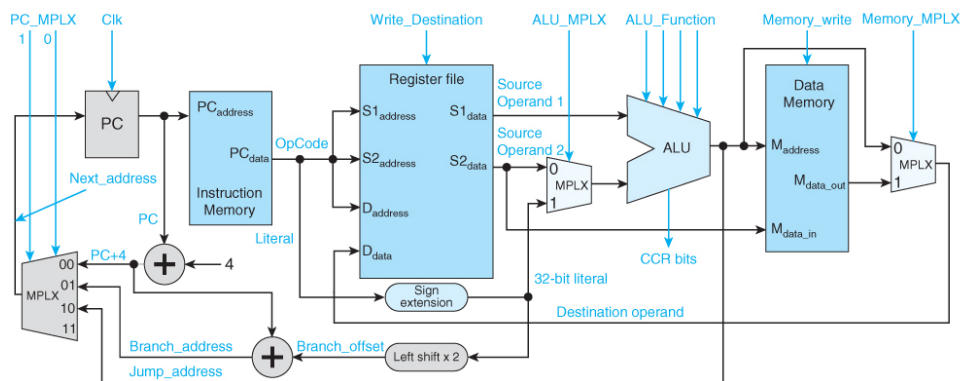
- Ska nu bygga om arkitekturen till RISC-typ
 - Få instruktioner, ingen/nästan ingen mikrokod
 - Load-Store arkitektur, bara 1 minnesaccess per instruktion
 - Använder Harvardarkitektur
- Placera alla generella register i en registerfil (RF)
 - Ett block där alla register ligger i, och två register kan läsas och 1 skrivs per gång, klockad som tidigare
 - Väljer vilka register mha tre adresser
 - Ett datavärde in och två ut
 - En styrsignal avgör om skrivning i register sker



Enkel RISC-baserad struktur, forts.

- Enkelt exempel från kursboken (avsnitt 7.2)
 - Samma byggblock som tidigare

FIGURE 7.14 Adding control signals to the computer



Exekveringstid i exemplet

- Klockfrekvens begränsas av längsta väg från register till nästa register

- Instruktion LDR D,(S1,#L) ; Läs minnescell som (S1) +L pekar på

- $t_{\text{cycle}} = t_{\text{pc}}$

+ t_{Imem}

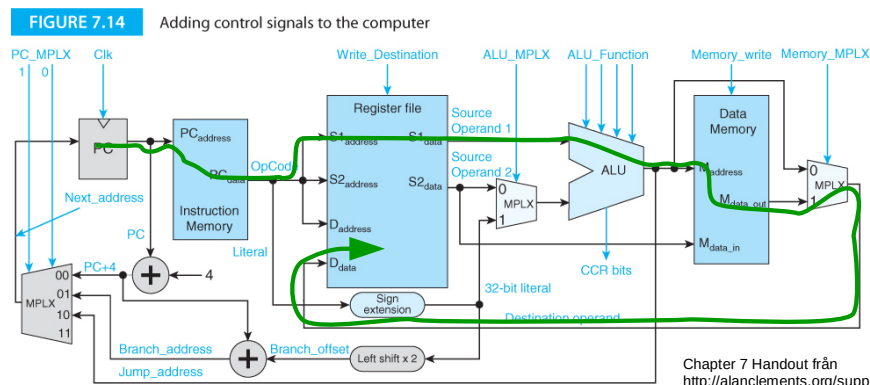
+ t_{RF}

+ t_{ALU}

+ t_{Dmem}

+ t_{MPLX}

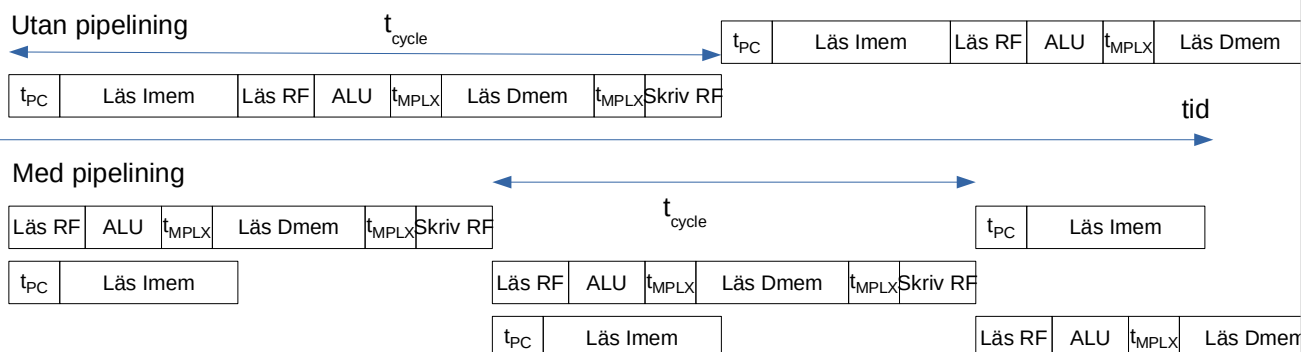
+ t_{RF}



Pipelining, 1:a försöket

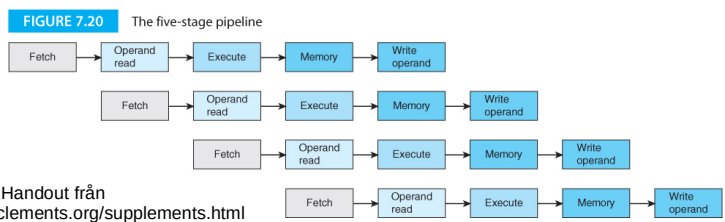
- Korta ned t_{cycle} genom att läsa nästa instruktion innan föregående är slut

- Kräver ett register mellan Imem och resten av processorn



Pipelining: Fler steg

- Kan dela upp instruktionen ytterligare (Ex 5 steg)
 - Hämta instruktion
 - Läs operand
 - Beräkna (ALU-operation)
 - Minnesaccess
 - Skriv till registerfil



Pipelining: Fler steg, forts.

- Borde ge upp till 5 gånger högre klockfrekvens
 - Ger andra problem
 - Hoppaddress beräknas sent
 - Läsning från dataminne till register
 - Användning av registervärde innan värdet tillgängligt

