

TSEA28 Datorteknik Y (och U)

Föreläsning 12

Kent Palmkvist, ISY

Dagens föreläsning

- Flaggor (hjälp till lab4)
- Latens/genomströmning
- Pipelining
- Konflikter
- Stall/bubblor
- Datakonflikt
- Branch prediction

Praktiska kommentarer

- Laboration 4 simuleringslabb
 - Redovisning i labb, kräver dokumenterad lösning
 - Lämpligen mha excelblad från hemsidan
 - Räcker inte med binärdata i lmia
 - Lmia kräver stor skärm
- Sorteringstävling deadline Måndag 8/5 kl 23.59.
 - Skicka in bidrag via mail (bifoga sparad mia-fil)
 - Se websida om tävling för mer info om hur jag testar
 - Jag ska kunna ladda design, ändra värden i PM adress E0-FF, trycka på regs=0, nollställ klockcykler, och sedan Kont.
 - Programmet ska stanna när sorteringen är klar.
 - Sista maskinkodsinstruktionen ska vara "HALT" från 1:a delen av labben (eller använda SEQ-fältet).

Praktiska kommentarer lab4

- Sorteringsalgoritm använder $A > B$ i flödesschemat
- Exemplet på instruktion är BGE motsvarande $A \geq B$
- Testa med likadana värden i tabellen
 - Byter plats på lika värden => sortering stannar aldrig
- Tal är tvåkomplementstal
 - Både positiva och negativa tal är möjliga

Praktiska kommentarer lab 4, forts.

- Vad säger flaggor vid subtraktion A-B?
 - Subtraktion implementeras som $A+(-B)$, dvs vänd tecken på B och lägg ihop. -B fås genom att vända alla bitar i B och lägga till 1 i LSB (minnessiffra till höger)
 - Z: om subtraktionen gav noll som svar, dvs $A=B$
 - C: om subtraktionen gav carry ut, gäller bara positiva heltalsberäkningar
 - N: differensen är negativ (om differensen kan representeras i tillgängligt antal bitar)
 - O: spill (overflow) när svaret beräknades (för 2-komplementstal)

$$A - B = A + (-B) = A + \overline{B} + 1$$

- Exempel (antag 4-bitars tal => talområde -8 till +7),

- $6-4=2$ $4-6=-2$ $-4-6=-10$ $4-(-6)=10$ $-8-7=-15$ $7-(-8)=15$

11111	11	1 11	1 11	1 1	1111
0110	0100	1100	0100	1000	0111
+1011	+1001	+1001	+0101	+1000	+0111
-----	-----	-----	-----	-----	-----
0010	1110	0110	1010	0001	1111
C=1	C=0	C=1	C=0	C=1	C=0
N=0	N=1	N=0	N=1	N=0	N=1
O=0	O=0	O=1	O=1	O=1	O=1

OBS: 5 bitar

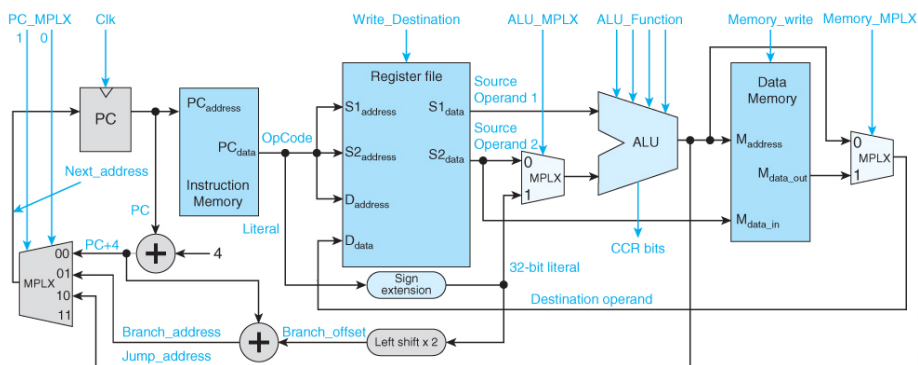
2 = 00010	-2 = 11110
4 = 00100	-4 = 11100
6 = 00110	-6 = 11010
7 = 00111	-7 = 11001
8 = 01000	-8 = 11000
10=01010	-10=10110
15=01111	-15=10001

Notera $O=N$ respektive $O<>N$

Enkel RISC-baserad struktur

- Enkelt exempel från kursboken (avsnitt 7.2)
 - LOAD-STORE, alla operation mellan register i registerfil

FIGURE 7.14 Adding control signals to the computer



© Cengage Learning 2014

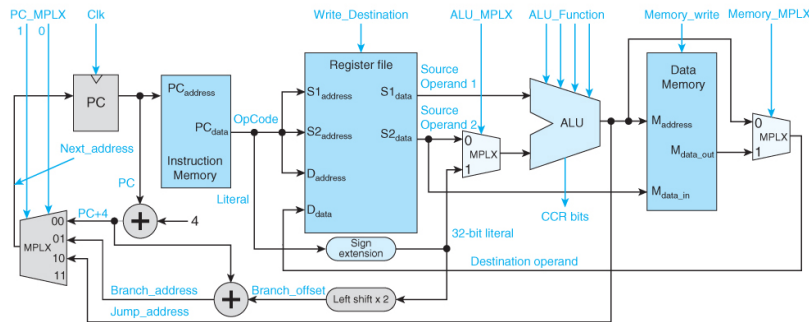
Chapter 7 Handout från
<http://alanclements.org/supplements.html>

Enkel RISC-baserad struktur, forts.

- Ingen mikrokod

- Maskininstruktionens bitar styr enheterna direkt (Write_Destination, ALU Function etc.) utgående från opcode och registerfält
- 32-bitar instruktion (4 byte) => öka PC med 4 vid varje instruktion

FIGURE 7.14 Adding control signals to the computer

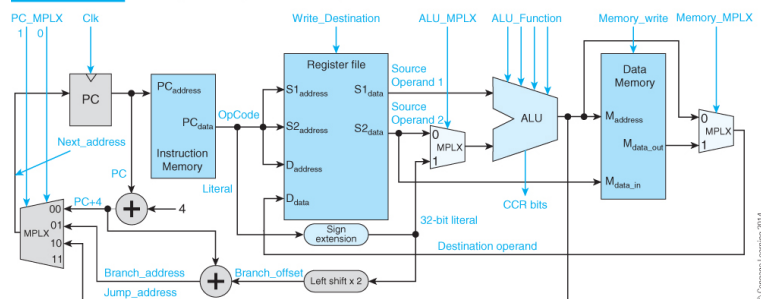


Enkel RISC-baserad struktur, forts.

- Styrsignalerna genereras mha uppslagstabell/logic baserad på instruktionsord från instruktionsminnet (tabell 7.8 i boken)

- Ex: ADD R0,R1,R2 =>
PC_MPLX=00 Writedest=1
ALU_MPX=0 ALU_FUNC=add
Memorywrite=0
Memory_MPLX=0
- Ex: SUB R0,R1,#23 =>
PC_MPLX=00 Writedest=1
ALU_MPX=1 ALU_FUNC=sub
Memorywrite=0
Memory_MPLX=0
- Ex: STORE R0,[R1,#0x12] =>
PC_MPLX=00 Writedest=0
ALU_MPX=1 ALU_FUNC=add
Memorywrite=1
- Ex: BRA 0x1234 => PC_MPLX=01
Writedest=0 Memorywrite=0

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_{mem}

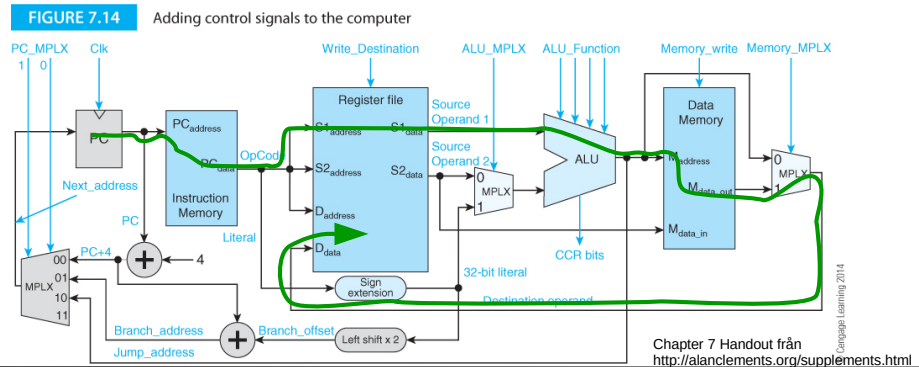
+ t_{RF}

+ t_{ALU}

+ t_{Dmem}

+ t_{MPLX}

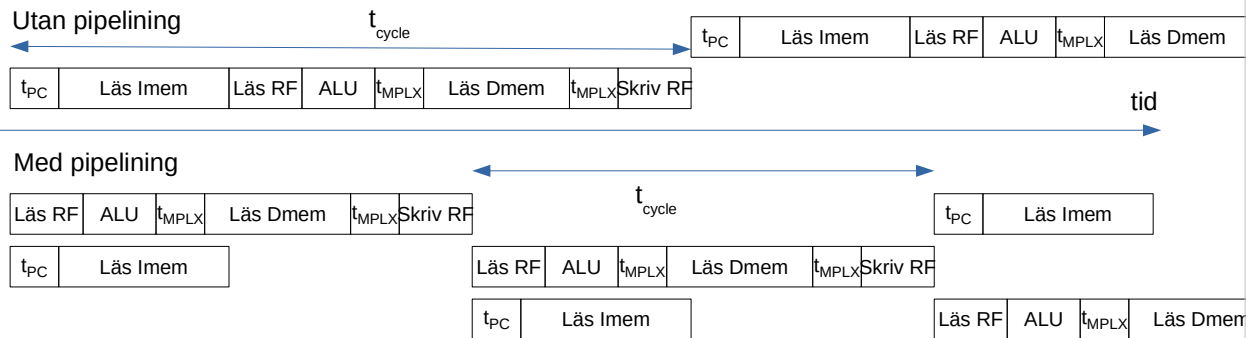
+ t_{RF}



Pipelining

Pipelining, 1:a försöket

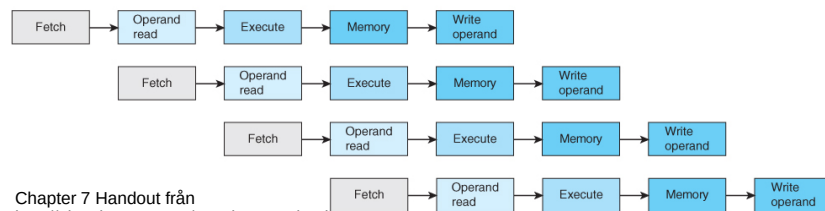
- Korta ned t_{cycle} genom att läsa nästa instruktion innan föregående är slut
 - Kräver ett register mellan Instructionmemory och resten av processorn



Pipelining: Fler steg

- Kan dela upp instruktionen ytterligare (Ex 5 steg)
 - Hämta instruktion
 - Läs operand
 - Beräkna (ALU-operation)
 - Minnesaccess
 - Skriv till registerfil
- Figuren är lite missvisande
 - Slutet på tidigare instruktioner till vänster
 - 5 parallella aktiviteter

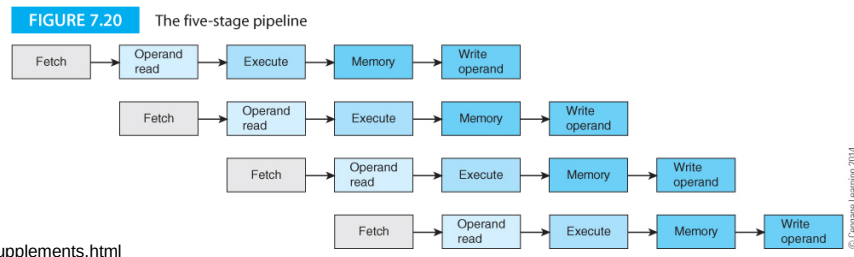
FIGURE 7.20 The five-stage pipeline



Chapter 7 Handout från <http://alanclements.org/supplements.html>

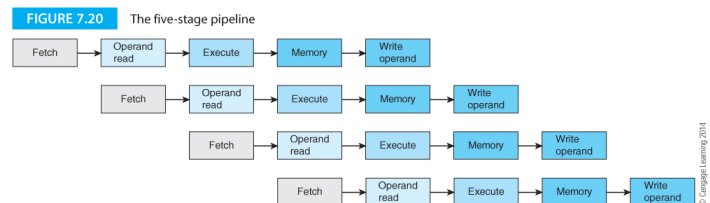
Pipelining: Fler steg, forts.

- 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



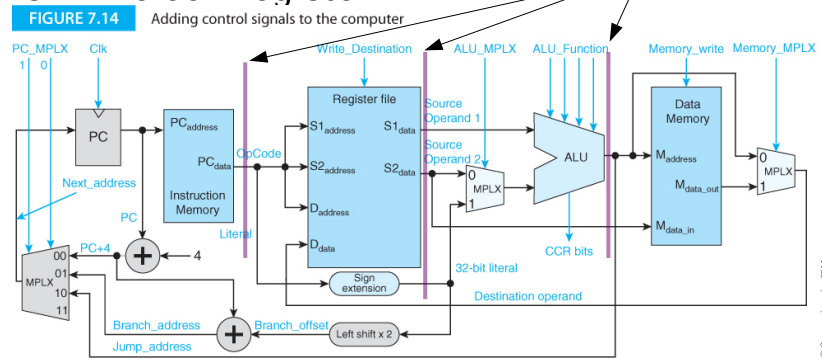
Pipelining: Fler steg

- Borde ge upp till 5 gånger högre klockfrekvens
 - Om alla steg tar lika lång tid
 - Operand read kan vara väldigt enkel => behöver inte så lång tid
 - Minnesläsning/skrivning kan ta lång tid
 - Ignorerar just nu eventuella databeroden etc.
 - Hämtar instruktioner innan föregående instruktioner slutförts
 - Läser register innan föregående instruktion skrivit register



Hur pipelining adderas till strukturen

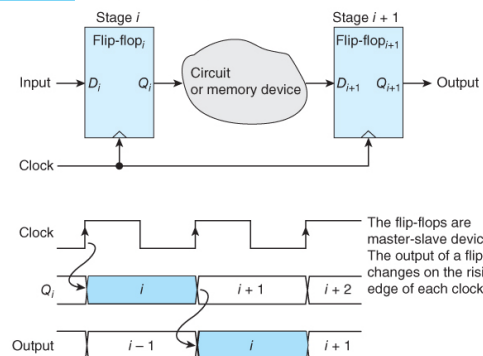
- Lägg in register i dataflödet
 - Varje signal som passerar strecket ska sparas i ett register
- IR mellan instruktionsminne och registerfil
- Register mellan registerfil och ALU



Timing hos pipeline

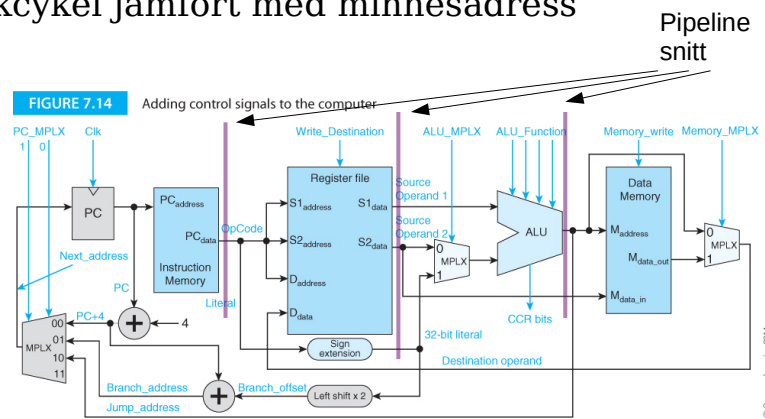
- Resultat från krets/minne passerar nästa register efter stigande flank hos klockan
 - Hela klockcykeln tillgänglig för beräkning/access
 - Fördröjer resultat 1 klockcykel per steg
 - Indikera vilken instruktion aktiv vid varje tidpunkt i respektive register

FIGURE 7.24 The pipelined stage



Flera klockcyklers fördröjning

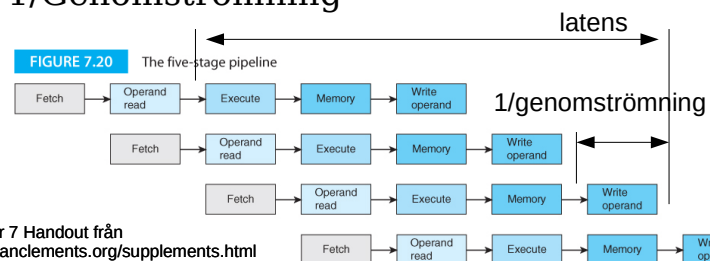
- $S2_{data}$ fördröjs 2 klockcykler innan minnet
- Måste få data i rätt klockcykel jämfört med minnesadress
- Styr signaler fördröjs
 - måste hänga ihop med instruktionerna



Chapter 7 Handout från
<http://alanclements.org/supplements.html>

Begrepp: latens, genomströmning

- Latens: Hur lång tid det tar att få utföra en komplett instruktion
- Genomströmning: Hur många instruktioner per sekund som kan utföras
- $1/\text{Genomströmning} = \text{hur ofta en ny instruktion startas}$
- För pipelining: Latens $> 1/\text{Genomströmning}$



Chapter 7 Handout från
<http://alanclements.org/supplements.html>

Pipelining är inte perfekt lösning

- Pipelining tillåter högre klockfrekvens
 - Ökar maximalt antal operationer per sekund
 - Minskar INTE tid för en enskild operation att slutföras
- Databeroende mellan operationer begränsar hur användbar ökning av klockfrekvens är
 - Villkorliga hopp behöver flaggorna satta innan hopp kan göras
 - Behöver NOP (No-operations) läggas till mellan compare och villkorligt hopp?
 - Tillåter flaggorna att beräknas innan villkorliga hoppet avgör om hopp ska tas

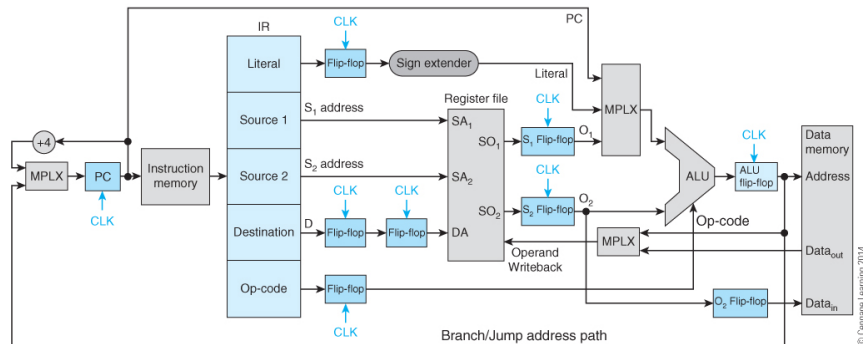
Exempel på olika pipelinedjup

- ARM11 (2002)
 - 8 steg
- Intel Pentium 4 (2004)
 - 31 stegs pipeline
- AMD 64 (2004)
 - 12 stegs pipeline
- i7 Haswell pipeline (2013)
 - 14 till 19 steg

Placera in register (flipflops)

- Även IR är klockad
 - Olika delar i instruktion behövs vid olika tidpunkter

FIGURE 7.32 Extending the pipelined architecture by adding data memory



Timingexempel Store instruktion

- Läs instruktion, hämta operander, beräkna adress, skriv i minnet
 - Nästa Store instruktion kan starta direkt

FIGURE 7.33 Timing of a store memory access

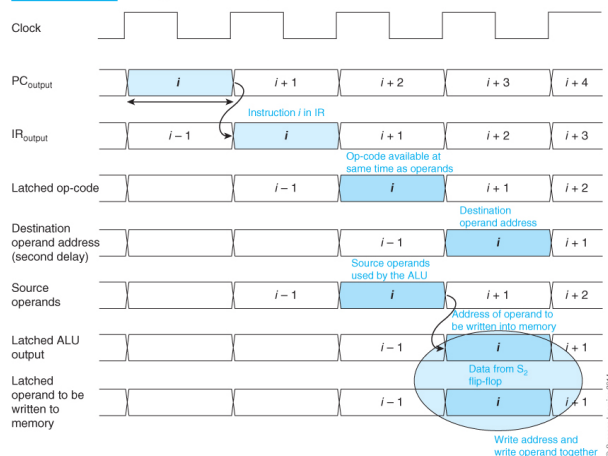
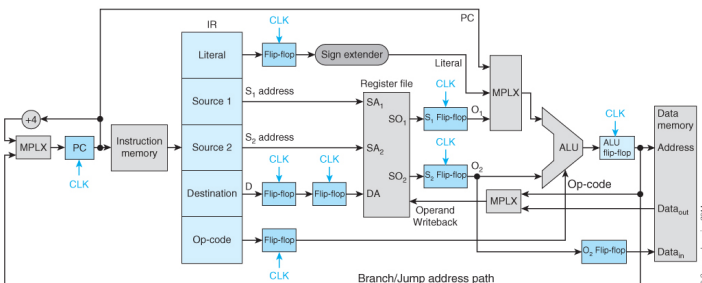


FIGURE 7.32 Extending the pipelined architecture by adding data memory



Problem vid load (minnesläsning)

- Måste ge registerfil tid att spara värde från minne
 - Måste hålla destinationsadress en cykel längre (1 cykel för skrivning i registerfil)
- Lösning: Stall
 - stoppa aktivitet i alla andra pipelinesteg

FIGURE 7.32 Extending the pipelined architecture by adding data memory

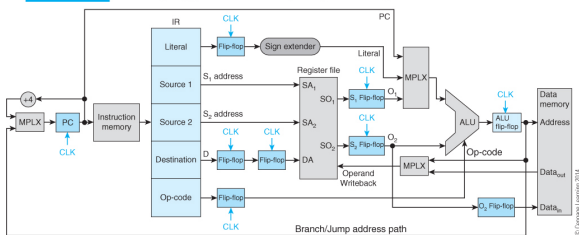
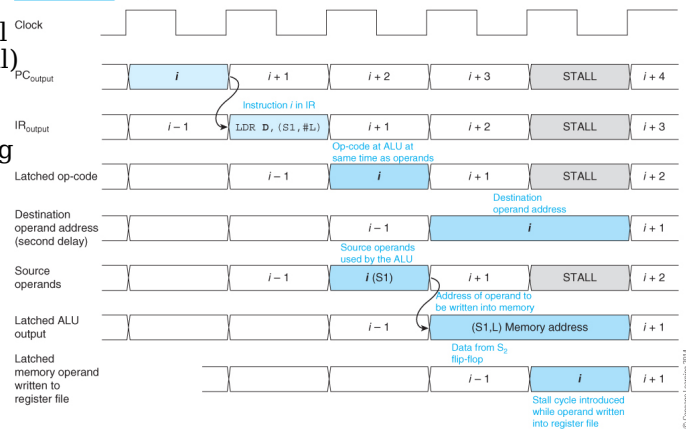


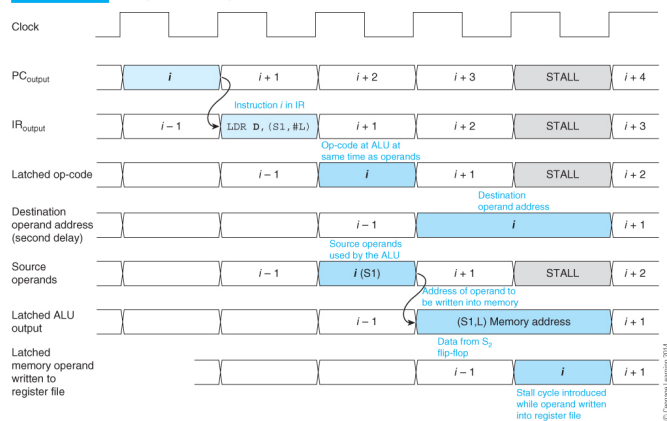
FIGURE 7.34 Fixing the load timing



Konflikter (hazards)

- Flera instruktioner exekveras delvis samtidigt
 - Resultat från en instruktion ofta indata till nästa
 - Resultat från en instruktion är inte alltid tillgänglig till nästa instruktion
- Vid konflikter kan pipeline behöva stannas
 - Jfr den inte hämtade instruktionen vid LOAD
 - Kallas pipeline stall eller bubblor

FIGURE 7.34 Fixing the load timing



Konflikter, datakonflikt

Typer av datakonflikt (data hazard)

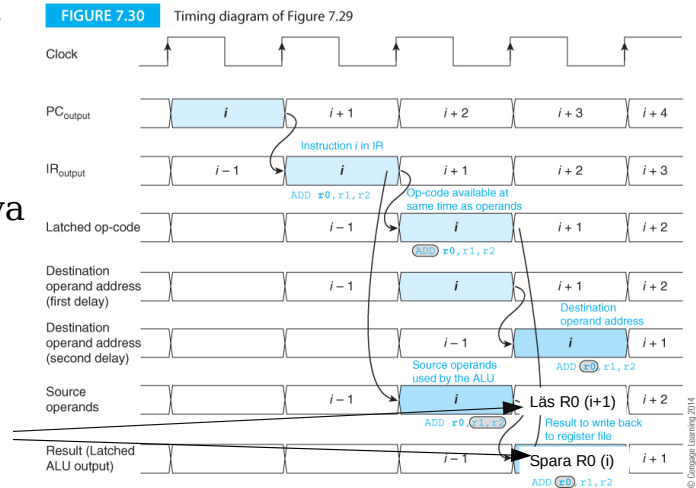
- Konflikt pga databeroende inte uppfyllt mellan instruktioner
- RAW (Read After Write)
 - Läs efter skriv
 - Föregående instruktion har inte sparat resultat som ska användas i nuvarande instruktion
- WAR (Write After Read)
 - Skriv efter läsning
 - Vanligen inget problem (kommer i senare föreläsning)
- WAW (Write After Write)
 - Skriv efter skrivning
 - Vanligen inget problem (kommer i senare föreläsning)

Data hazard exempel (RAW)

Ex: Add r0,r1,r2 ; i (r0=r1+r2)
Add r3,r1,r0 ; i+1 (r3=r0+r1)

- R0 uppdateras i instruktion i, läses i instruktion i+1
 - R0 uppdateras i samma klockcykel som andra instruktionen vill läsa R0
- Registerfilen klarar inte skriva och läsa samma adress i en klockcykel
 - Kräver stall?

Går inte läsa och skriva i RF på samma adress samtidigt



Chapter 7 Handout från
<http://alanclements.org/supplements.html>

Hantering av RAW (mindre lämplig)

- Det är inte en bugg, det är en feature...
- Lös problemet genom att kräva minst en instruktioner mellan varje databeroende
 - Programmeraren/kompilatorn måste hålla reda på databeroende och lägga till NOP (no operation) => sänkt prestanda
 - Bättre om annan instruktion utan kan placeras in istället (inte alltid möjligt)
- Om pipelinedjup ändras måste programmet skriva om
 - Kompilerad kod inte portabel!

RAWOp1:

```
ADD R0,R1,R3 ; R0 = R1+R3
ADD R4,R0,#3 ; R4 = R0+3
ADD R2,R0,#4 ; R2 = R0+4
```

OKOp1:

```
ADD R0,R1,R3 ; R0 = R1+R3
NOP
ADD R4,R0,#3 ; R4 = R0+3
ADD R2,R0,#4 ; R2 = R0+4
```

Hantering av RAW (bättre?)

- Enkelt: STALL (ger "bubblor" i pipeline)
 - Tvinga nästa instruktion vänta på data (sänker fortfarande prestanda vid konflikt)
 - Programmeraren behöver inte hålla reda på databeroende
- Exempel
 - 2 stall mellan instruktion 2 och 3 => två ej utnyttjade möjligheter att utföra instruktioner motsvarande 2 NOP inlagda mellan instruktion 2 och 3

ADD Rx,Ry,Rz ; Rx = Ry+Rz

Klockcykel	1	2	3	4	5	6	7	8	9
ADD R0,R1,R2	IF	OF	OE	OS					
ADD R3,R4,R5		IF	OF	OE	OS				
ADD R2,R3,R4			IF	S	S	OF	OE	OS	
ADD R6,R1,R3						IF	OF	OE	OS

IF = Instruction Fetch OF = Operand Fetch
 OE = Operation Execution OS = Operand Store
 S = Stall

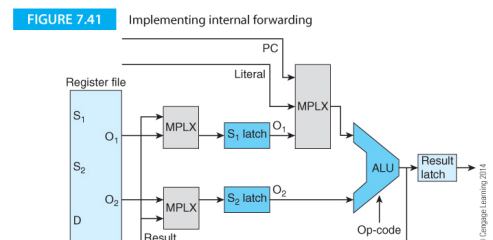
Hantering av RAW (bäst)

- Forwarding
 - Skicka data direkt från resultat till operand (ej via ALU resultatregister)
 - Skriver även i registerfilen i nästa klockcykel
 - Jämför destination för resultat hos nuvarande instruktion med källregister för nästa instruktion (styr mplx före s1 och s2 latch)
 - Kräver mer logik (jämförare mellan destination och source samt multiplexer)
 - Förlorar ingen klockcykel

Klockcykel 1 2 3 4 5 6 7

ADD R0,R1,R2	IF	OF	OE	OS			
ADD R3,R4,R5		IF	OF	OE	OS		
ADD R2,R3,R4			IF	OF	OE	OS	
ADD R6,R1,R3				IF	OF	OE	OS

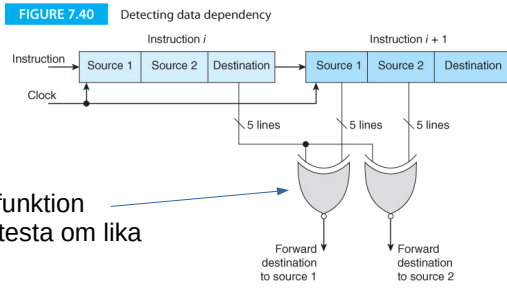
IF = Instruction Fetch OF = Operand Fetch
 OE = Operation Execution OS = Operand Store



Forward, timing

- Exempel

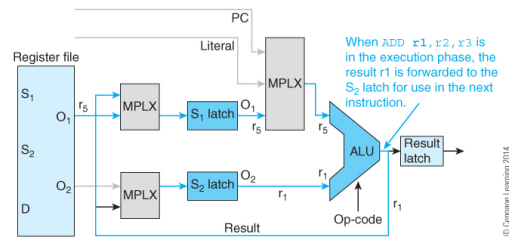
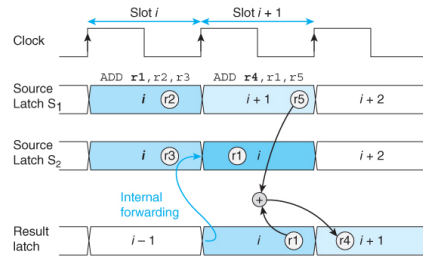
ADD R1,R2,R3 ; Destination R1
 ADD R4,R1,R5 ; Källa R1



xor-funktion
 dvs testa om lika

© Cengage Learning 2014

FIGURE 7.42 Implementing internal forwarding: timing diagram and data flow

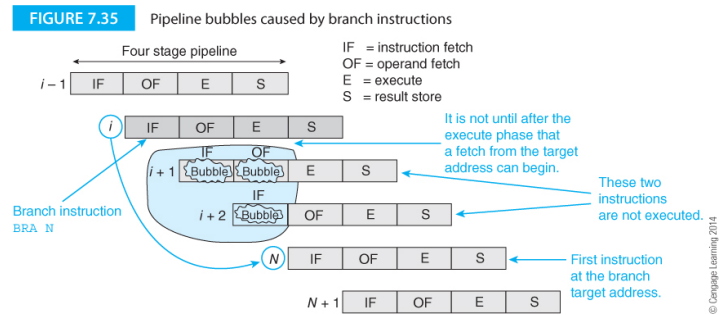


© Cengage Learning 2014

Styrkonflikt

Styrkonflikt

- Konflikt därför att nästa instruktion inte är rätt hämtad
- Nästa instruktion hämtas innan hopp avkodats
 - Execute påverkar programräknare (PC)
 - Måste låta bli utföra två instruktioner
 - Kasta bort hämtade instruktioner
- Gäller även subrutinanrop, avbrott etc.
- Villkorliga hopp som inte hoppar kostar inget extra



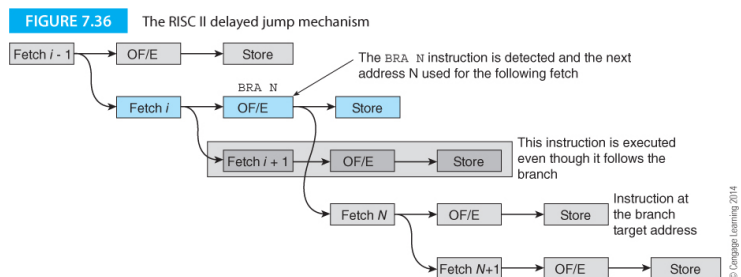
Styrkonflikt (mindre lämplig lösning)

- Det är ingen bugg, det är en feature...
 - Räkna med att alla hopp även utför en eller möjligen två instruktioner efter hoppinstruktionen
 - Känt som "Delayed branch"
 - Kan behöva lägga till NOP (=> sänkt prestanda)

```

mov r1,#1
bra next
mov r2,#2 ; utförs också
:
next:
add r3,r2,r4

```



Styrkonflikt (bättre lösning)

- Kräv inte att programmeraren tar hänsyn till problemet
 - Undvik krav på kännedom om hur processorn ser ut inuti
 - Ändrad arkitektur kan ge andra krav...
 - Försök korrigera/hantera problemet internt i processorn
- Försök undvika kostnaden för hopp
 - Minska antal ej utnyttjade klockcykler
 - Svårt då adress till nästa instruktion inte känd i förväg

Hopp, kostnad

- Instruktion i är BRA N
 - Hoppet avkodad i klockcykel 4 (OE), instruktion hämtas i klockcykel 5
 - Två bubblor, i+1 och i+2

Klockcykel	0	1	2	3	4	5	6	7	8	9	10	11
Instruktion i-2	IF	OF	OE	OS								
i-1		IF	OF	OE	OS							
i			IF	OF	OE	OS						
i+1				IF	OF	OE	OS					
i+2					IF	OF	OE	OS				
N						IF	OF	OE	OS			
N+1							IF	OF	OE	OS		
N+2								IF	OF	OE	OS	
N+3									IF	OF	OE	OS

IF = Instruction Fetch OF = Operand Fetch
OE = Operation Execution OS = Operand Store

Alternativa sätt att beskriva händelser i CPU

- Samma händelser, olika sätt beskriva beroende på axlar

- Axel är aktivitet eller instruktion
- Tidsaxel vågrätt eller lodrätt

```

i   ADD R0,R1,R3 ; R0 = R1+R3
i+1 NOP
i+2 ADD R4,R0,#3 ; R4 = R0+3
i+3 ADD R2,R0,#4 ; R2 = R0+4
  
```

Klockcykel	IF	OF	OE	OS
0	i	i-1	i-2	i-3
1	i+1	i	i-1	i-2
2	i+2	i+1	i	i-1
3	i+3	i+2	i+1	i
4	i+4	i+3	i+2	i+1

IF = Instruction Fetch OF = Operand Fetch
 OE = Operation Execution OS = Operand Store
 S = Stall

Klockcykel	1	2	3	4	5	6	7
ADD R0,R1,R3	IF	OF	OE	OS			
NOP		IF	OF	OE	OS		
ADD R4,R0,#3			IF	OF	OE	OS	
ADD R2,R0,#4				IF	OF	OE	OS

IF = Instruction Fetch OF = Operand Fetch
 OE = Operation Execution OS = Operand Store
 S = Stall

Hopp, alternativ vy

- Vid hopp uppstår bubblor i pipeline

- Antag instruktion i är BRA N
- I utförandefasen är den nya adressen beräknad
- I exemplet uppstår två tomma klockcykler (i+1 och i+2)

Cykel	Fetch instr.	Hämta operand	Utför	Spara resultat
0	i-2	i-3	i-4	i-5
1	i-1	i-2	i-3	i-4
2	i (BRA)	i-1	i-2	i-3
3	i+1	i (BRA)	i-1	i-2
4	i+2	i+1	i (BRA)	i-1
5	N			i (BRA)
6	N+1	N		
7	N+2	N+1	N	
8	N+3	N+2	N+1	
9	N+4	N+3	N+2	N+1

Hämtade men ej utförda instruktioner

Förbättra hantering av hopp

- Detektera hopp redan vid operandhämtning
 - Endast en klockcykel förloras
 - Ett hopp har ofta inga operander som behöver hämtas
 - BRA label ; offset lagrad i IR som literal i instruktionen

Cykel	Fetch	Hämta	Utför	Spara
	Instr.	operand		resultat
2	i (BRA)	i-1	i-2	i-3
3	i+1 (bubbla)	i (BRA beräknad)	i-1	i-2
4	N	i+1 (bubbla)	I	i-1
5	N+1	N	i+1 (bubbla)	i
6	N+2	N+1	N	i+1 (bubbla)
7	N+3	N+2	N+1	N
8	N+4	N+3	N+2	N+1

Typer av hopp

- Olika typer
 - Ovillkorliga hopp
 - Hopp utförs alltid
 - Villkorliga hopp tillbaks i loopar
 - Hopp tas ofta
 - Andra villkorliga hopp
 - Lika sannolikt hopp inte tas som att det tas
- Viktigt försöka minska påverkan av villkorliga hopp som tas

Kostnad för hopp

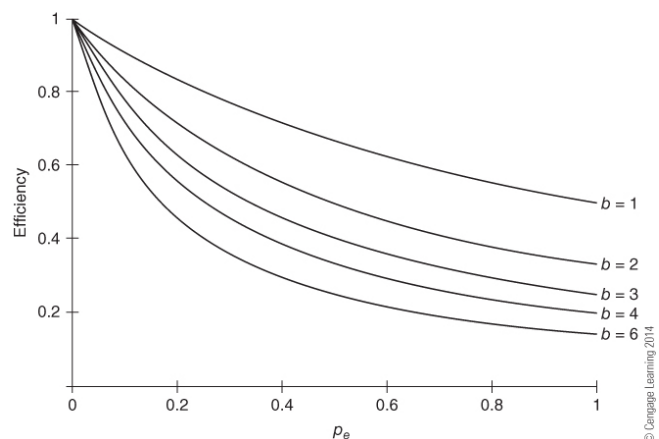
- Antaganden
 - Vanliga instruktioner tar 1 klockcykel
 - Sannolikhet för en hoppinstruktion är p_b
 - Sannolikhet för att hopp tas är p_t
 - Hopp som tas kostar b klockcykler extra
 - Hopp som inte tas kostar inget extra (tar 1 klockcykel)
- Medelvärde på antal klockcykler per instruktion

$$T_{ave} = (1 - p_b) \cdot 1 + p_b \cdot p_t \cdot (1 + b) + p_b \cdot (1 - p_t) \cdot 1 = 1 + p_b p_t b$$

Effektivitet beroende på antal tagna hopp

- $p_e = p_b p_t$, dvs hur sannolikt är det att instruktionen är ett villkorligt hopp som tas
 - Djup pipeline blir dyr när hopp tas
 - Hopp som tas kostar b klockcykler extra

FIGURE 7.43 The efficiency of a RISC processor as a function of p_e



Försök att minska antal hopp

- Skapa instruktioner som tillåter villkorliga operationer
 - ARM har t ex ADDEQ R1,R2,R3
 - Utför instruktionen endast om Z-flagga = 0
 - Möjliggör sekvensiell kod med villkor utan hopp
 - HP-PA (och AVR) har skip-next instruktion (villkorliga)
 - Utför inte nästa instruktion om villkor sant
 - Kan skapa if then else kod utan hopp

Branch prediction

Försök förutsäga hopp

- Om instruktionshämtaren i förväg vet att hopp kommer göras kan den hämta instruktioner från andra adresser
 - Fördröjning kan då undvikas då rätt instruktioner kan hämtas och börja avkodas
 - Omöjligt att förutsäga villkorliga hopp
 - Kan gissa, mer eller mindre rätt
- Om rätt gissning görs förhindras förlorade klockcykler
- Kan i vissa processorer ange i instruktionen om hoppet oftast förväntas tas eller inte
 - Statisk hopp prediktering
 - Bestäms vid kompilering av koden

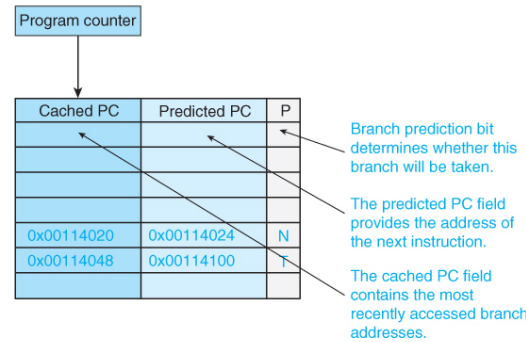
Att förutsäga hopp

- Statisk hopp-prediktering missar många hopp
 - Vid kompilering förutsägs vilka hopp som kommer tas oftast
 - Fler än 15% av alla hopp missas
- Bättre lösning: Dynamisk hoppprediktering
 - Kom ihåg om hoppet togs förra gången koden kördes
 - Sparas internt i processorn för varje villkorlig hoppinstruktion som påträffas
 - Begränsat antal adresser kan komma ihåg
- Ännu bättre förutsägelse om inte bara det senaste hoppet används som förutsägelse
 - Majoritet av visst antal av senaste exekveringar kan ange mest sannolika förutsägelse
 - Hysteres kan användas (enstaka felgissningar påverkar inte förutsägelsen)

Hopp prediktering, implementering med BTB

- Olika hopp har olika gissning
 - Spara gissning för varje enskilt villkorligt hopp
- Dyrt minne (associativt minne)
 - Branch Target Buffer (BTB)
- Kan gissa om nästa instruktion utan att nuvarande instruktion lästs
 - Måste kört koden innan tabellen fyllts

FIGURE 7.48 The simple branch target buffer



Hur branch prediction fungerar

- Fortfarande dyrt att gissa fel
 - Tappar 2 klockcykler i exemplet

TABLE 7.14 Using the BTB to Reduce Pipeline Bubbles

(a) Branch correctly predicted

Cycle	Instruction	Fetch Instruction	Fetch Operand	Execute Instruction	Store Operand
0	Conditional branch	i	$i - 1$	$i - 2$	$i - 3$
1	Predicted instruction	N	i	$i - 1$	$i - 2$
2	Predicted instruction + 1	$N + 1$	N	i	$i - 1$
3	Predicted instruction + 2	$N + 2$	$N + 1$	N	i
4	Predicted instruction + 3	$N + 3$	$N + 2$	$N + 1$	N

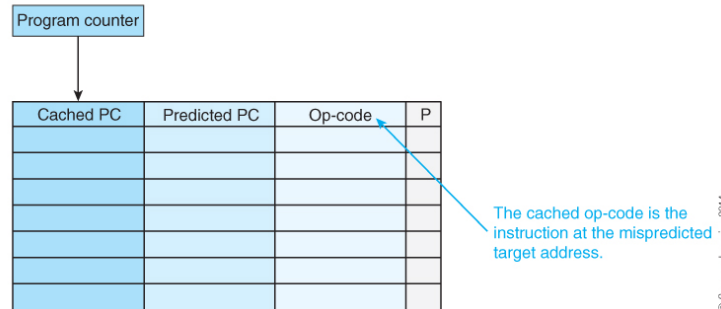
(b) Branch mispredicted (M is the address where execution begins following misprediction)

Cycle	Instruction	Fetch Instruction	Fetch Operand	Execute Instruction	Store Operand
0	Conditional branch	i	$i - 1$	$i - 2$	$i - 3$
1	Predicted instruction	N {bubble}	i	$i - 1$	$i - 2$
2	Predicted instruction + 1	$N + 1$ {bubble}	N {bubble}	i	$i - 1$
3	Mispredicted instruction	M	$N + 1$ {bubble}	N {bubble}	i
4	Mispredicted instruction + 1	$M + 1$	M	$N + 1$ {bubble}	N {bubble}

Förbättrad Branch Target Buffer

- Problem med miss består i att instruktionen inte kan börja avkodas
- Lagra opcode från adress som inte förväntas läsas
 - Om fel förutsägelse så finns opcode redan tillgänglig

FIGURE 7.49 Modified branch target buffer with cached op-code at mispredicted address

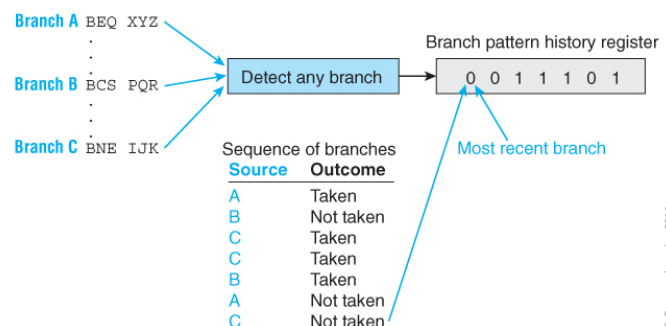


© Cengage Learning 2014

Två nivåers branch predictions

- Resultat från flera villkorliga hopp kan vara korrelerade
 - Vissa mönster kan finnas som kan säga mer om vilka hopp som sannolikt kommer hända
- Branch pattern history
 - En form av "fingeravtryck" för en viss sekvens av hopp

FIGURE 7.50 Recording branch sequences—the branch pattern history



© Cengage Learning 2014

Två nivåers branch prediction

- Använd mönstret av senaste villkorliga hopp för att välja en prediktor som användes senast detta mönster sågs
 - Har troligen gått genom samma sekvens av villkorliga hopp, och kommer då troligen göra samma val framöver

FIGURE 7.51

Using the pattern history to index into a table of individual state machines

