Föreläsningsanteckningar
4. Pipelining

Olle Seger 2012
Anders Nilsson 2016

1 Inledning

Denna föreläsning handlar om pipelining, som är den helt dominerande processorarkitekturen i dag. Man kan säga att alla processorer bygger på användande av pipelining och cacheminnen.

Källor är i första hand [1, 2], men även [3, 4]

2 OR-datorn

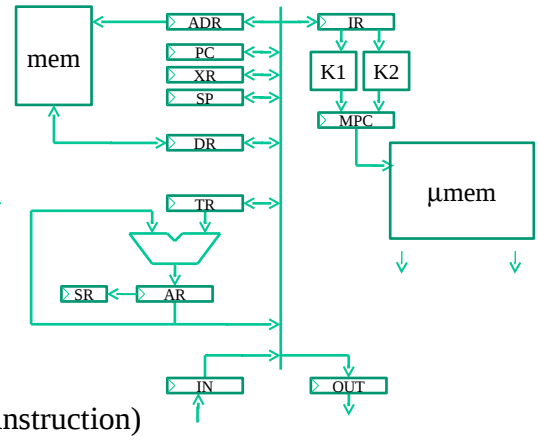
OR-datorn (Olle Roos), som är en mikroprogrammerad dator, har behandlats i de två första föreläsningarna. En liknande dator, lite mer avancerad, är BL-datorn (Björn Lindskog), som används i en laboration. Båda dessa är lätta att komma igång med. Vi kan, helt utan verktyg, implementera instruktioner och adresseringsmetoder.

Men OR-datorn är alldeles för långsam. Instruktionen `LDA 13` tar exempelvis 11 klockcykler att exekvera. Alldeles för många klockcykler slösas bort på ingenting.

OR-datorn är för långsam!

- LDA 13 (exempelvis)
  - Hämta : 3 CP
  - K2 1 CP
  - Absolut,K1: 3 CP
  - EXE: 4 CP

Summa: 11 CP



Alltså 11 CPI (clocks per instruction)
Vi siktar på 1 CPI!

3 Pipelining

3.1 Vad bör göras?

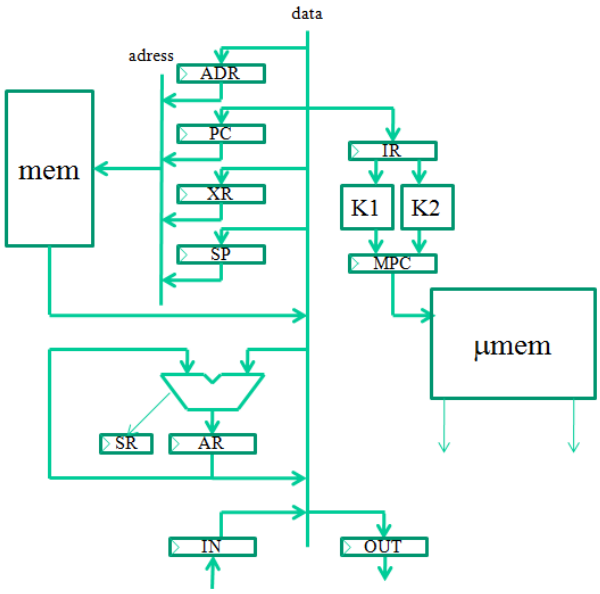
Det enklaste och mest rättframma sättet att snabba upp en dator är att införa pipelining.Pipelining är en sorts paralellism, där flera instruktioner exekveras samtidigt. I ett visst ögonblick kan en instruktion hämtas, för en annan pågår adresseringsmodsberäkning och en tredje instruktion exekveras. I detta sammanhang pratar man ofta om RISC (reduced instruction set computer). Typiska RISC-egenskaper är

- alla instruktioner har samma instruktionsformat och finns i ett minnesord.
- alla instruktioner tar lika många klockcykler att exekvera.
- få adresseringsmoder. Typiskt finns bara indexering, där adressen till operanden finns i ett register.
- Minnet accessas bara genom två instruktioner, `LOAD` och `STORE`.

- Aritmetiska/logiska instruktioner använder bara register.
- Ganska många register, som är generella, dvs kan innehålla data eller adress.
- skilda minnen för program och data (Harvard-arkitektur).

Ovanstående lista snabbar nog inte upp datorn, men nu går det att bygga en pipeline och den gör datorn snabbare! Det är ganska enkelt att närma sig 1 färdig instruktion per klockcykel.

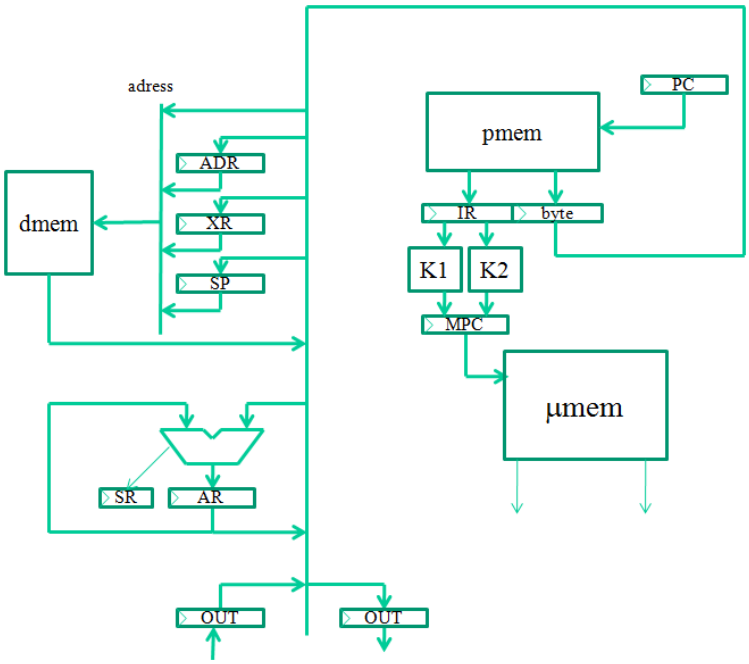
Vi börjar nu att bygga om OR-datorn. Första steget blir att få till kortare datorvägar.



I ovanstående blockschema har ett antal register plockats bort. De register som innehåller minnesadresser (PC,XR,SP och ADR) har grupperats så att de direkt kan adressera minnet. `LDA 13` tar nu bara 4 klockcykler.

fas	cykler
hämta	1
K2	1
absolut	1
exekvera	1

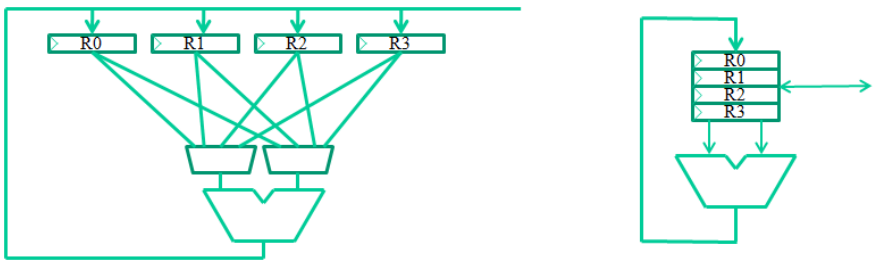
Nu inför vi skilda program- och dataminnen. Programminnet får dubbel bredd, så att hela instruktionen kan hämtas på 1 klockcykel.



Nu tar `LDA 13` 3 klockcykler.

fas	cykler
hämta	1
K2	1
absolut	-
exekvera	1

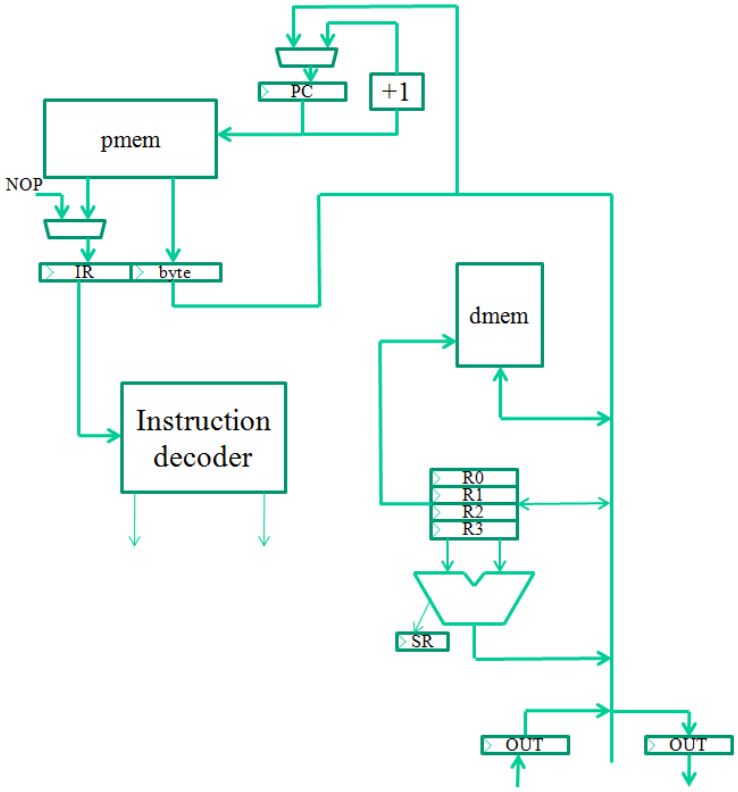
Endast ett ackumulatorregister känns begränsande. Vi inför istället en registerfil (flera register). Den är utformad så att på varje klockcykel går det att läsa 2 register och skriva 1 register.



Vi går vidare med omkonstruktionen av OR-datorn.

- Vi avskaffar XR,SP och inför generella register.
- Vi avskaffar att en instruktion kan ha flera a-moder.
  - `ADD Rx,Ry,Rz` använder endast register.
  - `LD Rx,(Ry)` är det enda sättet att läsa i minnet.
  - `ST (Rx),Ry` är det enda sättet att skriva i minnet.
  - Vi avskaffar MPC och mikroprogrammering
  - Mikrominnet blir kvar, men bara som uppslagstabell. Man kan säga att de mikroprogramsnuttar vi hade förut nu alla tar 1 CP.

Vi är nu klara och här är resultatet:



Datorn har en tvåstegs pipeline. Man brukar inte tala om pipelining i det här fallet utan istället säga förhämtning. Datorn är inspirerad av processorfamiljen AVR från Atmel. Dessa processorer är mestadels enkla mikrocontrollers. En tillämpning kan exempelvis vara garageportsöppning.

Låt oss köra ett enkelt testprogram

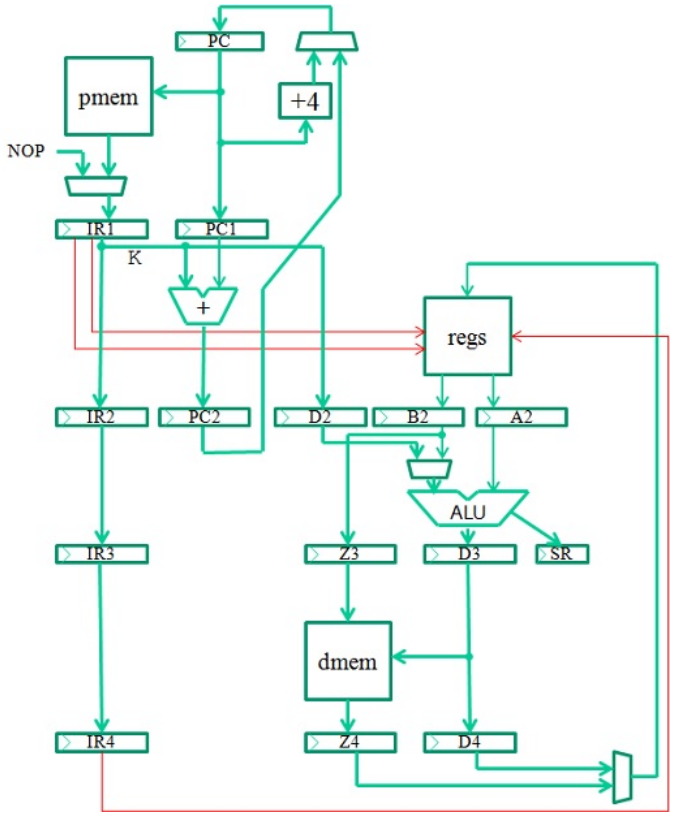
```
0:LD R1,(R0)
1:ADD R3,R2,R1
2:JMP 0
3:XXX
```

Datorn arbetar enligt principen: hämta-utför. När en instruktion hämtas, så utförs föregående. Ett pipeline-diagram illustrerar detta:

PC	IR	register
0	-	-
1	LD	-
2	ADD	R1
3	JMP	R3,PC
0	NOP	-
1	LD	-

4 Modelldatorn

Nedanstående bild visar den dator, som används i laborationen, [1].



Datorn har en 5-stegs pipeline, med följande steg:

1. *IF = Instruction Fetch*. Den instruktion, som pekas ut av PC hämtas till IR1. Samtidigt beräknas ett nytt värde för PC.

2. *RR = Read Registers*. En av följande är rätt scenario

   - 2 register läses till A2 resp B2.
   - 1 register läses till A2 och en konstant kopieras till D2
   - en hoppadress beräknas till PC2

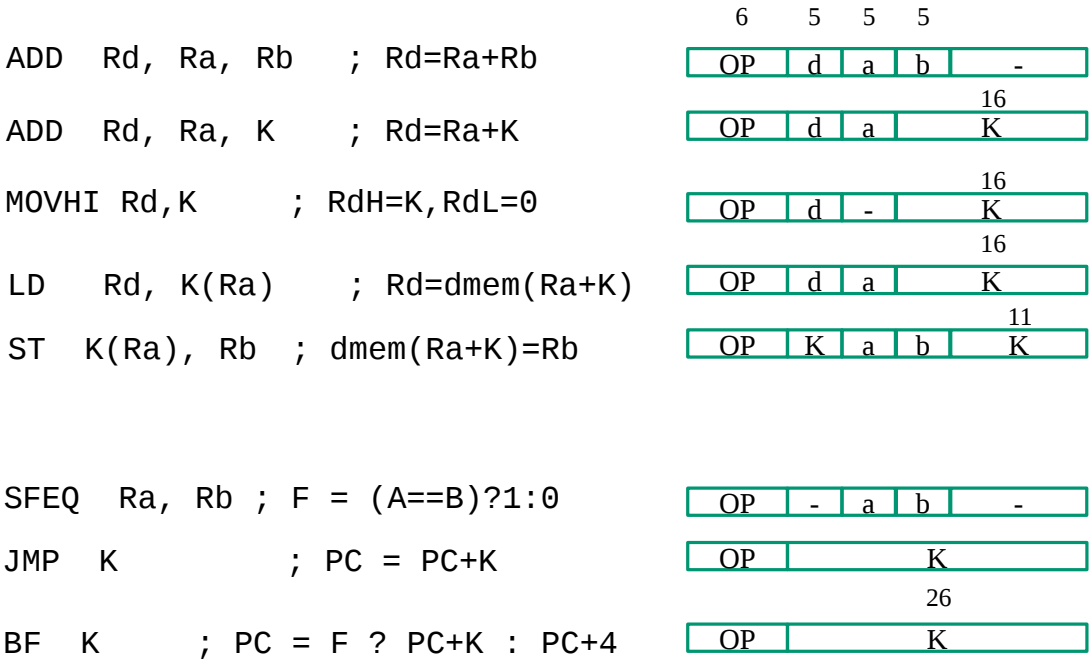
3. *EXE=Execute*. En av följande är rätt scenario:

   - ALUn beräknar ett nytt registervärde ($B2 \oplus A2$) till D3
   - ALUn beräknar en minnesadress ($D2 + A2$) till D3. Eventuellt kopieras B2 till Z3.

4. *MEM=Memory*. En av följande är rätt scenario:
   - *läsning* ur minnet till Z4.
   - *skrivning* till minnet.
   - *kopiering* från D3 till D4.
5. *WB=Write back*. En av följande är rätt scenario:
   - *skriv* Z4 till registerfilen.
   - *skriv* D4 till registerfilen.
   - *ingenting*.

Alla instruktioner kodas som 32-bitars minnesord. Nedanstående är en lista på några typiska instruktioner:

Några instruktioner



4.1 Testkörning

Låt oss testa datorn genom att köra ett snällt program:

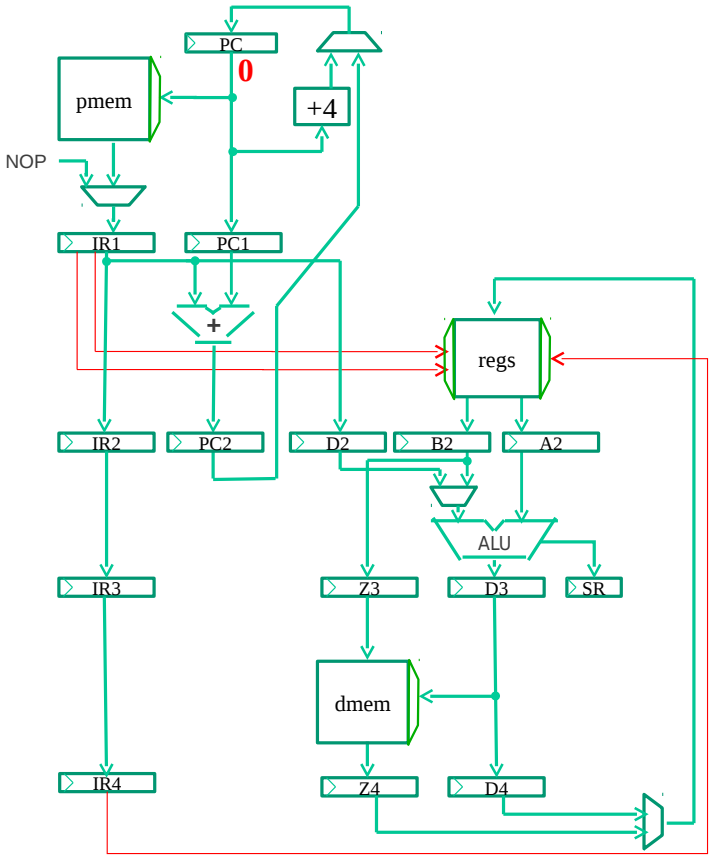
```
0:ADD R3,R2,R1
4:LD R6,0(R5)
8:SFEQ R7,R8
```

Programmet är snällt, eftersom instruktionerna inte har med varandra att göra. För enkelhetens skull antar vi att registren innehåller: R0=0,R1=1, ... Vi kommenterar vad som händer med `ADD`.

Vi trycker på reset-knappen:

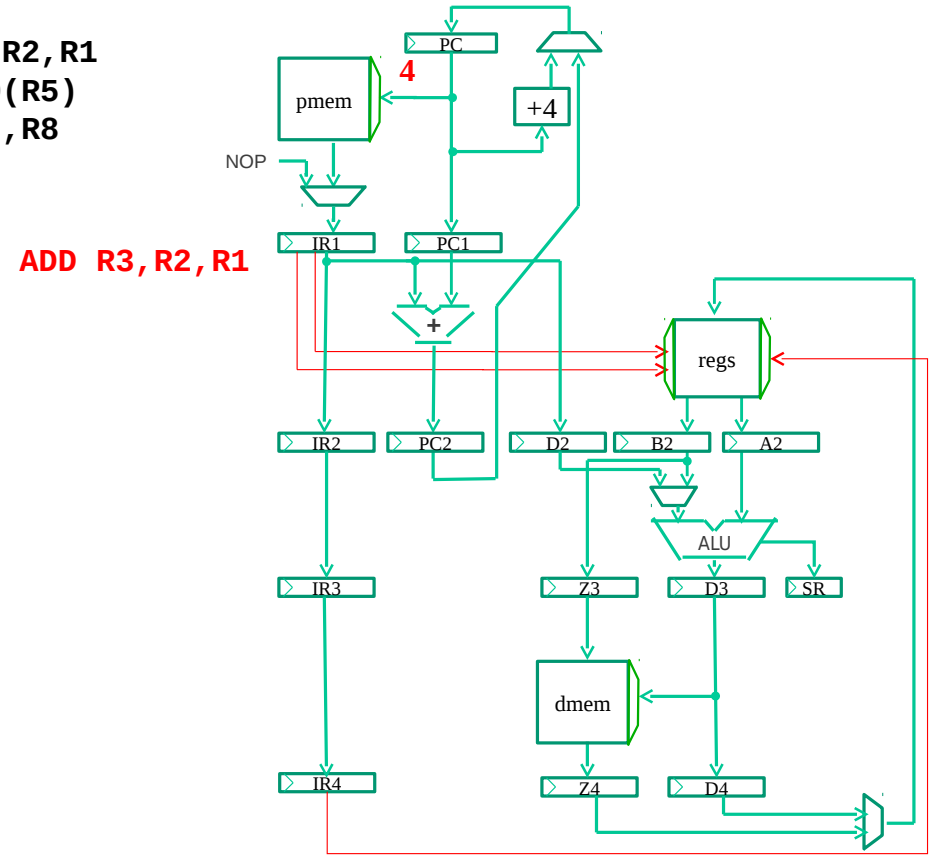
0:ADD R3,R2,R1
4:LD R6,0(R5)
8:SFEQ R7,R8

Några snälla instruktioner



Vi klockar:

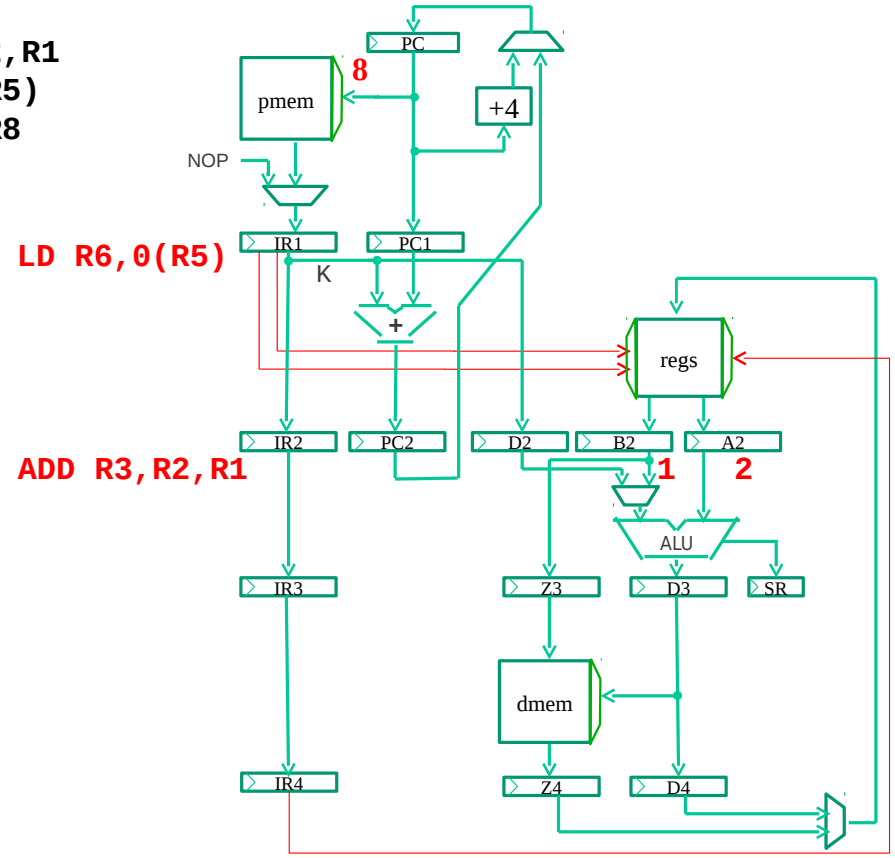
```
0:ADD R3,R2,R1
4:LD R6,0(R5)
8:SFEQ R7,R8
```



`ADD` har hämtats och `PC` har räknats upp.

Vi klockar:

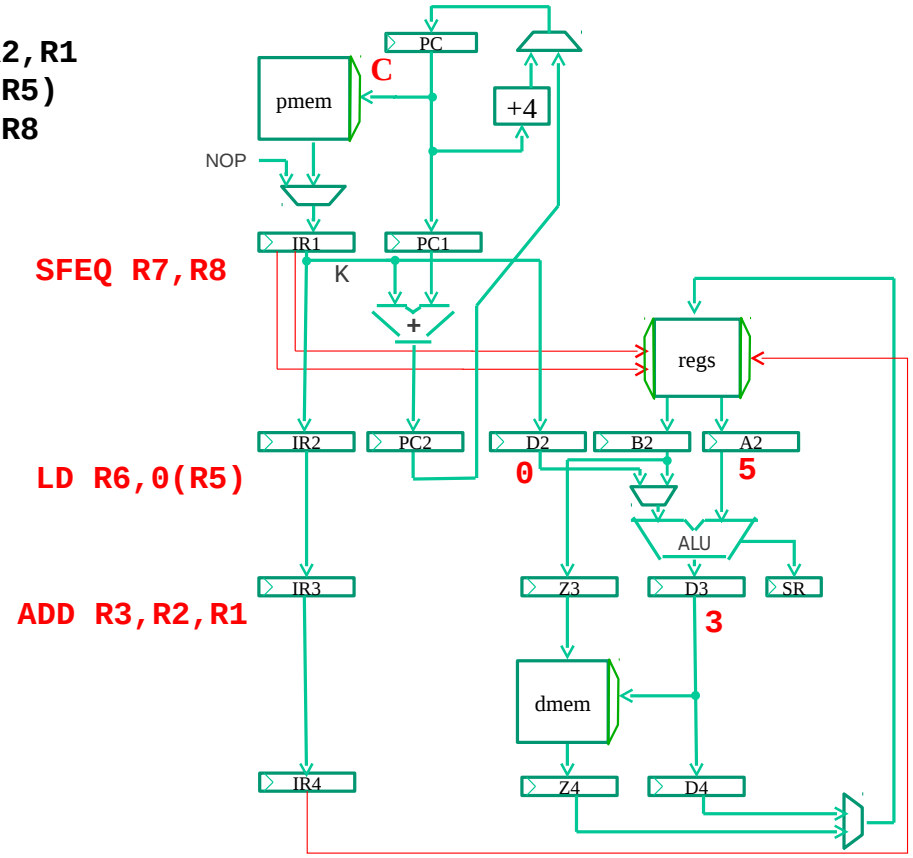
```
0:ADD R3,R2,R1
4:LD R6,0(R5)
8:SFEQ R7,R8
```



Två registervärden har lästs från registerfilen.

Vi klockar:

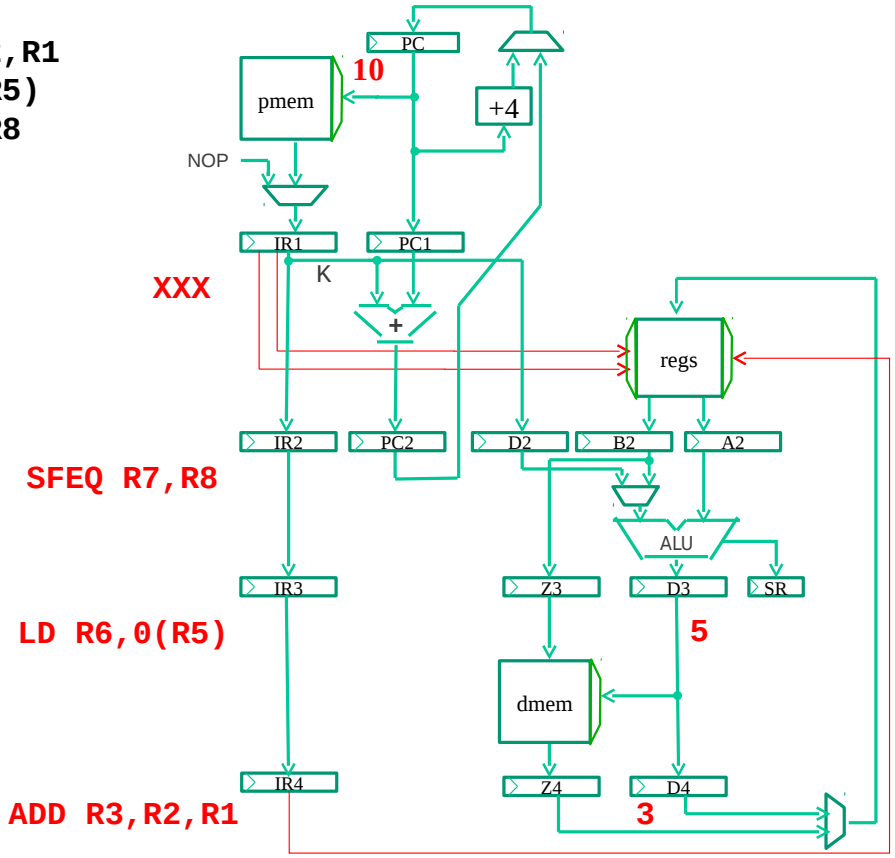
```
0:ADD R3,R2,R1
4:LD R6,0(R5)
8:SFEQ R7,R8
```



Additionen har skett.

Vi klockar:

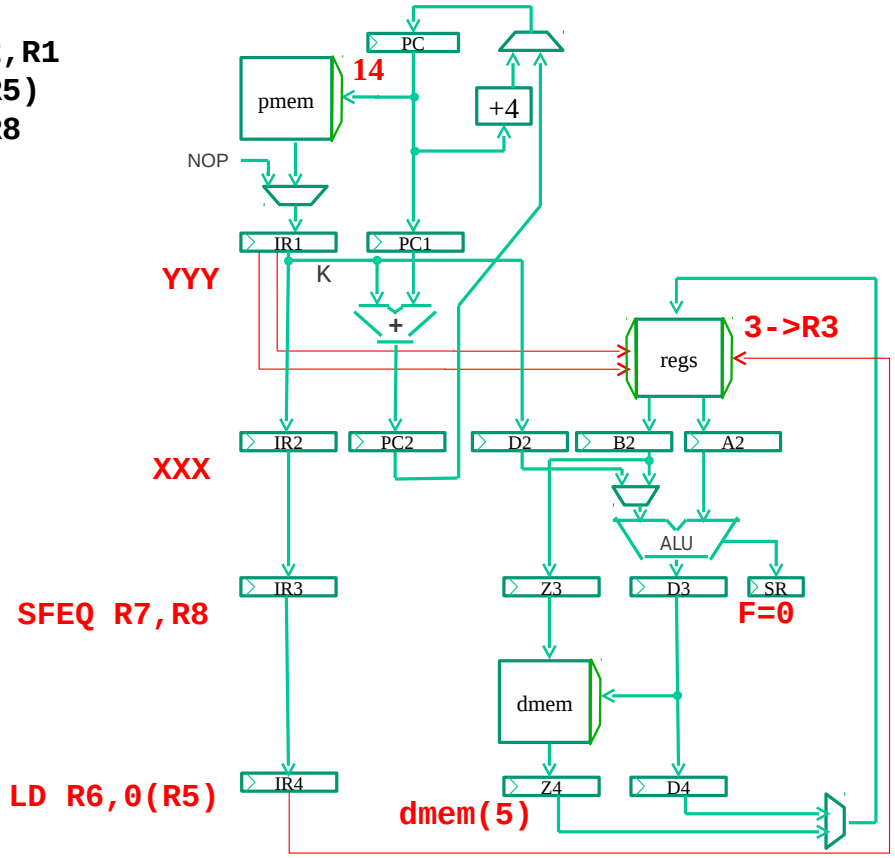
```
0:ADD R3,R2,R1
4:LD R6,K(R5)
8:SFEQ R7,R8
C:XXX
```



Summan kopierades till D4.

Vi klockar:

```
0:ADD R3,R2,R1
4:LD R6,K(R5)
8:SFEQ R7,R8
C:XXX
10:YYY
```



Register R3 uppdaterades.

4.2 Problem

Så långt verkar allt vara ganska enkelt. Det beror på att vi medvetet undvek problem i testprogrammet. Men för ett riktigt program kommer problem att dyka upp.

Man brukar tala om detta:

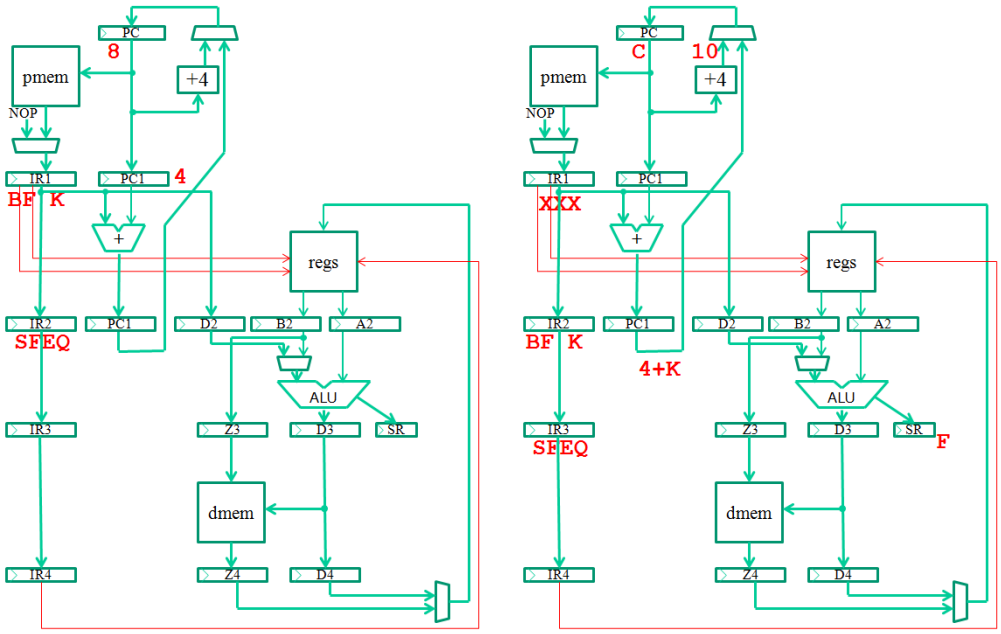
- *Hårdvarukonflikter*. Två pipelinesteg behöver samma hårdvara. Löses genom att duplicera hårdvara. Det är redan fixat i vår maskin, t ex:
  - Två minnen, pmem och dmem.
  - Två adderare. Vi har lagt till en adderare för hoppadresser.
- *Fördröjda hopp*. Pipelinen vill gärna hämta instruktionen efter ett hopp. Det är faktiskt enklast att alltid exekvera den instruktionen.

- *Databeroenden*. Så här är det: registren läses i steg 2 men skrivs i steg 5. Under några klockcykler är värdena i registerfilen gamla. Det rätta värdet finns någonstans i pipelinen.
- *Stall*. Pipelinen måste i vissa lägen stängas av.

4.3 Fördröjda hopp

För att utröna vad som händer vid hopp, så kör vi ett testprogram:

```
0:SFEQ R1,R2
4:BF K       ; hopp till 20 om R1==R2
8:XXX
C:YYY
 ...
20:ZZZ
```



I den vänstra bilden har hoppinstruktionen `BF` hämtats, men flaggan `F` är ännu inte satt. Vi måste därför köra en klockpuls till. Det är då enklast att låta instruktionen `XXX` rinna in i pipelinen.

I den högra bilden har `F` just fått sitt värde. Vi har förstås två fall:

- F=0. Inga problem, hämta instruktionen på adress `C`, dvs `YYY`.
- F=1. Nya värdet för `PC` blir `4+K=20`. För att förhindra `YYY` går in i pipelinen "trycker" vi in en `NOP` i `IR1`.

Sammanfattningsvis har vi fördröjda hopp, dvs instruktion efter hoppet verkställs alltid. Om hoppet sker måste en `NOP` tryckas in i pipelinen. Det är naturligtvis en bortslösad klockcykel.

4.4 Databeroende

Betrakta följande programsnutt:

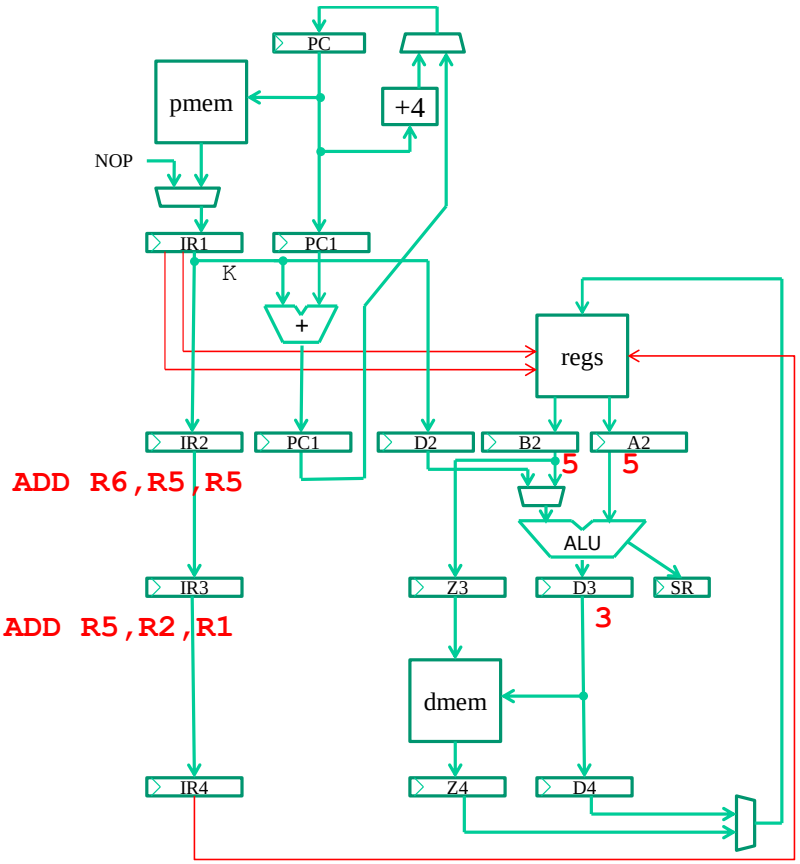
```
0:ADD R5,R2,R1
4:ADD R6,R5,R5
```

Intruktionen på rad 0 uppdaterar `R5`, som sedan används av instruktionen på rad 4. Detta är ett sk databeroende. Att ett program ser ut så här är förstås det normala.

Vi antar att registervärdena är R0=0, R1=1, ... Vi kör programsnutten på datorn. Betrakta situationen när instruktionen på rad 0 precis har exekverats.

```
0:ADD R5,R2,R1
4:ADD R6,R5,R5
```

Databeroende

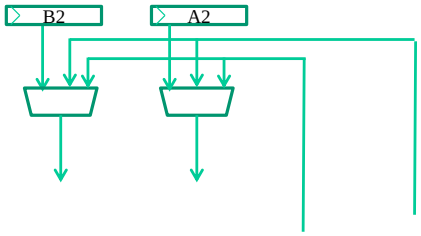


I `D3` ligger 3, som är det nya (rätta) värdet på `R5`. Instruktion på rad 4, har just läst ut de gamla värdet på `R5`. De gamla värdena måste förhindras att gå in i ALUn.

Vi konstaterar att ett nytt värde (som är på väg att skrivas till registerfilen) kan finnas i någon av `D3` eller `D4/Z4`. Ingångarna till ALUn måste förses med muxar, som väljer rätt data.

Databeroende

Registerfilen kan innehålla gamla data!
Var kan nya resultat finnas? D3,D4,Z4



Hur ska muxarna styras?

```
if IR2.op == ”instruktion som läser reg.”
  if  IR3.op == ”instruktion som skriver reg.”
   if IR2.a == IR3.d
      muxarn = …;
   else
```

Muxarna i figuren brukar kallas data forwarding muxes. Vi försöker nu att i ord beskriva styrningen av muxarna för fallet i programsnuttet ovan.

- Innehåller `IR1` en instruktion, som läser (1 eller 2) register?
- Innehåller `IR2` en instruktion, som skriver ett register?
- Är det månne så att det är samma register (obs två fall)?

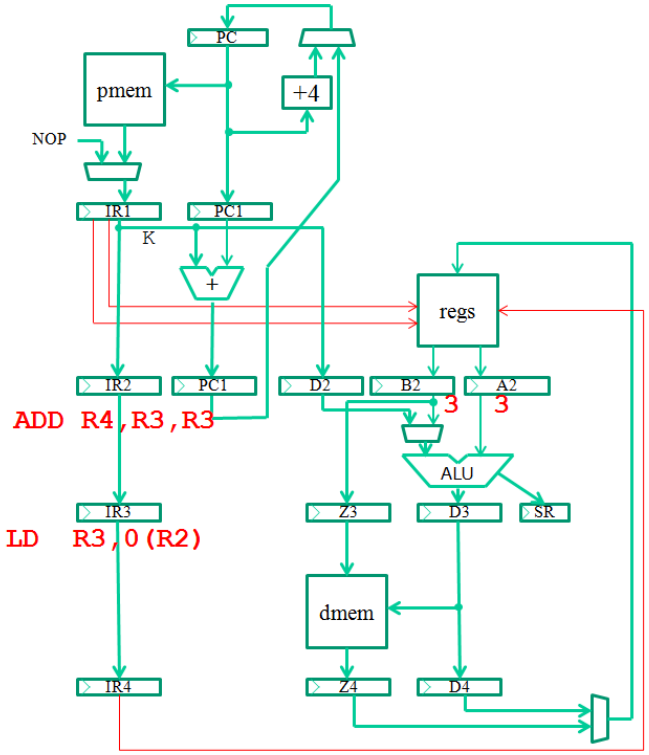
I vårt testsnutt är svaret ja på alla tre frågorna. Muxarna ska styras så att `D3` muxas in i båda ingångarna till ALUn.

4.5 Stall

Tyvärr är det så att ovanstående, alltså data forwarding, inte alltid räcker. Betrakta denna programsnutt:

```
0:LD   R3,K(R2)
4:ADD  R4,R3,R3
```

Vi kör den tills något intressant händer:



Situationen liknar den i föregående avsnitt. Vi har ett databeroende kring `R3`. Problemet är nu att det går inte att muxa in rätt värde. Det värdet är ju ännu inte läst från minnet. Här är goda råd dyra!

Vad bör göras? Jo:

1. Vi stänger av övre delen av pipelinen och låter `LD` fortsätta. Alltså: `PC`, `IR1`, `IR2` behåller sina värden. `IR3` blir `NOP`. `IR4` blir `LD`. Detta kallas pipeline stalling.

2. Nu ser det bättre ut. Rätta värdet på `R3` finns i `Z4`. Vi hoppas nu att hårdvaran vi gjorde i föregående avsnitt fungerar.

Men(!), säger kanske den uppmärksamme läsaren. Kan vi inte göra data forwarding från utgången på minnet (dmem) till ingången på ALUn på samma sätt

som vi gjorde för register D3? I terorin, ja. Men i praktiken så är minnesaccessen relativt långsam jämfört med den övriga kombinatoriken och vi skulle få en lång kritisk väg och tvingas till att dra ned klockfrekvensen på hela processorn. Då blir det istället effektivare att göra stall på pipen och införa en extra klockcykel just för denna situation.

Referenser

[1] Seger, Andersson (2011): *Laborationshandledning TSEA 49 Datorteknik D del 2 : Pipelining*, LiU-tryck.

[2] Patterson, Hennessy (2009): *Computer Organization and Design. The Hardware/Software Interface*, Morgan Kaufmann ISBN: 978-0-12-374493-7.

[3] Josefsson M. (2008): *Föreläsningsunderlag TSEA49 Datorteknik D del 2*, LiU-tryck.

[4] Clements (2008): *The Principles of Computer Hardware*, Oxford ISBN0-19-856453-8.