

TSEA83 : Datorkonstruktion

Fö4

Pipelining

Fö4 : Agenda

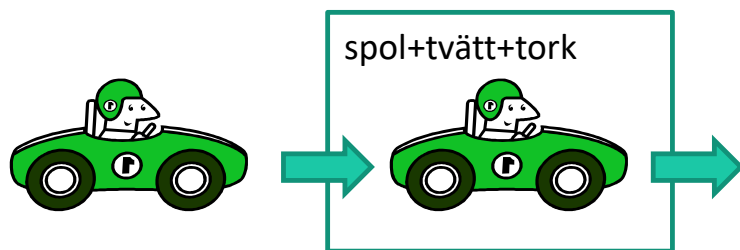
- Hur bygger man en pipeline-CPU?
 - Med utgångspunkt från OR-datorn
- Klassisk 5-steps pipeline
 - IF, RR, EXE, MEM, WB
- Klassisk 5-steps pipeline
 - Problem...
- Lab2
 - Pipelining

Hur bygger man en pipeline-CPU?

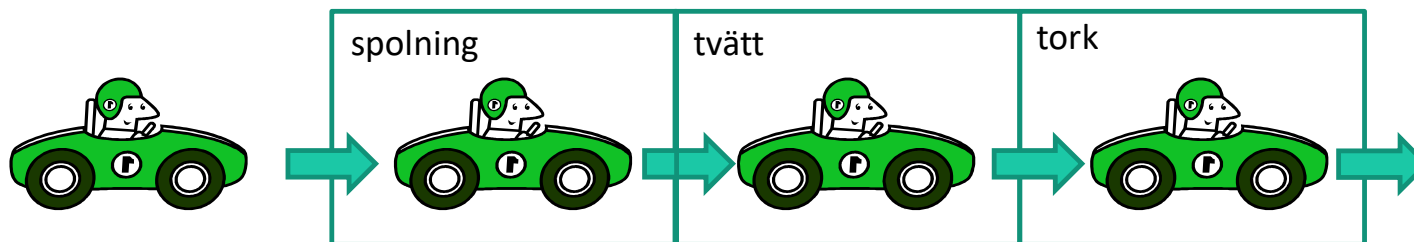
Med utgångspunkt från OR-datorn

Pipelining

"Komplex" biltvätt (tvättbågen är komplex)



Pipelinerad biltvätt



- De tre momenten (spol, tvätt, tork) tar lika lång tid
- Alla bilar går igenom samma program



Väntetid 1/3
Genomströmning 3

OBS, vi får inte uppsnabbningen gratis.
En del utrustning måste finnas på flera ställen!

Pipelining

Alternativa arkitekturer eller riktiga datorer

- kritik av OR-datorn
 - - för många klockcykler,
går att dock att snabba upp
 - + går att göra komplicerade instruktioner:
sortering, matrisinvertering, ...
- En pipelinad processor RISC
RISC = reduced instruction set computer
enkla, lika instruktioner => pipelining möjligt

Mikromaskinen

"Olle Roos – datorn"



= register



= minne

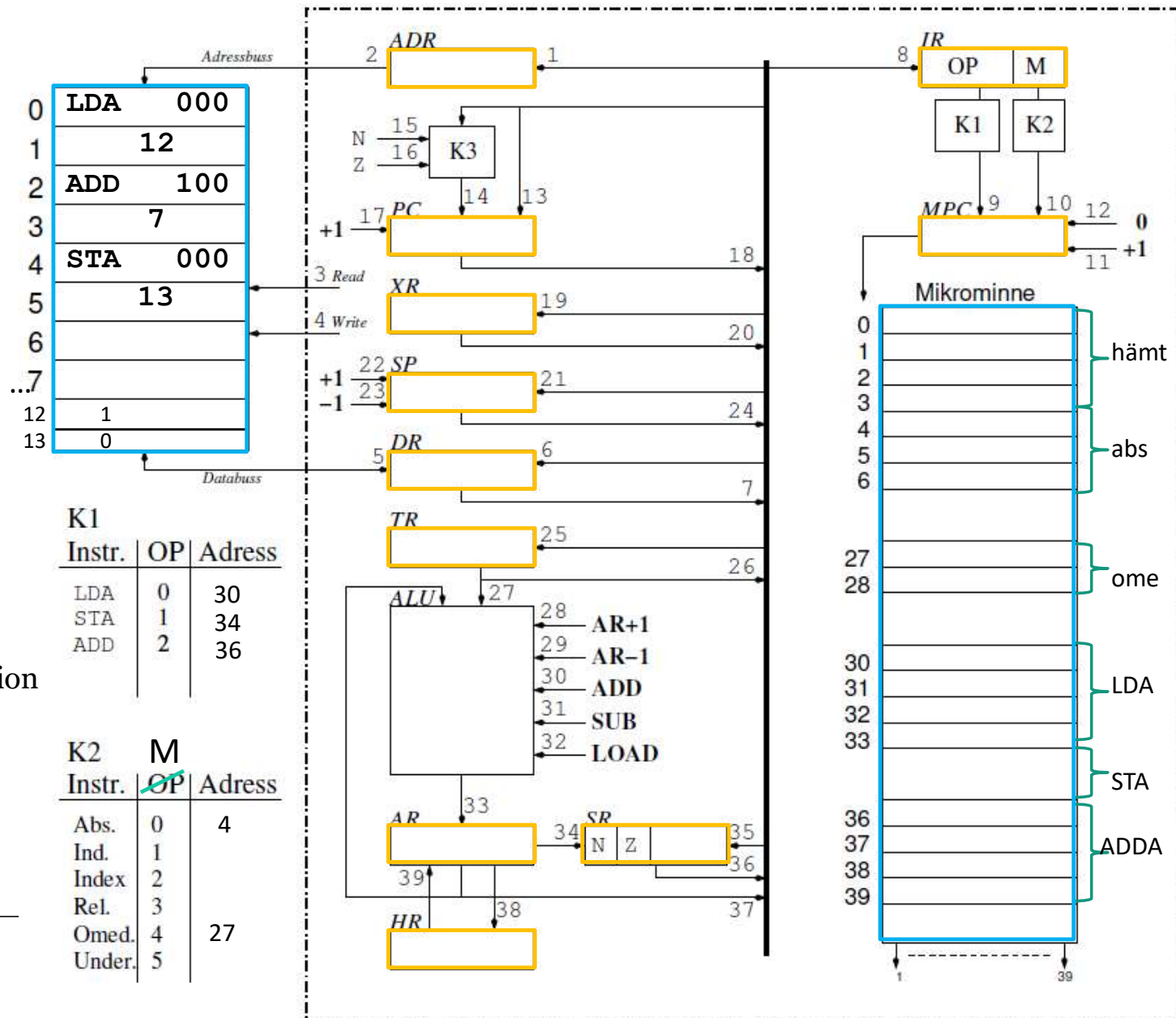


= kombinatorik

Det går åt ganska många klockcykler för att utföra en enda instruktion.

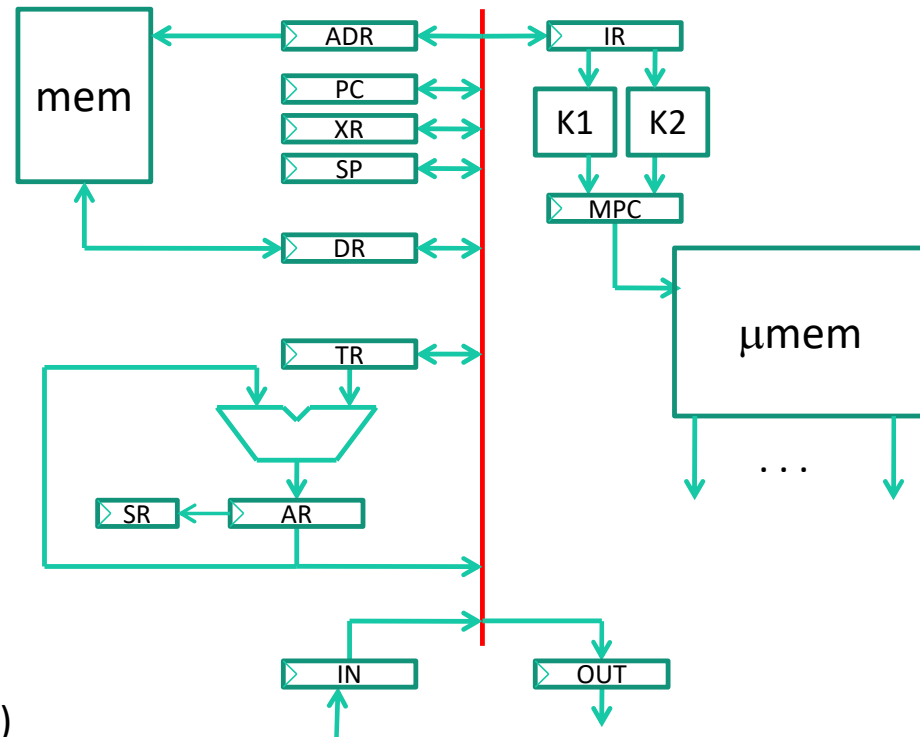
Vilket beror på att varje varje instruktion är uppdelad i flera (många) steg där (nästan) inget sker parallellt.

RESET



OR-datorn är för långsam!

- LDA 12 (exempelvis)
 - Hämta : 3 CP
 - K2 1 CP
 - Absolut: 3 CP
 - EXE: 4 CP
 - Summa: 11 CP



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

Hur kan OR-datorn förbättras?

- Parallellism
 - Tryck ihop mikrokoden, dvs gör flera μ op samtidigt
- Förhämtning:
 - hämta nästa instruktion under exekvering av pågående instruktion

```
adr->M->dr  
dr->tr  
tr->ar  
status
```

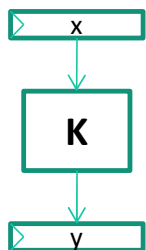
Exe av LDA

```
pc++->adr  
adr->M->dr  
dr->ir
```

Hämta nästa
instruktion

```
mpc++  
mpc++  
mpc++  
K2->mpc
```


Apropos klockfrekvens



Leta rätt på längsta
tidsfördröjningen
mellan två register.
Kalla den T.
Då gäller
 $f < 1/T$

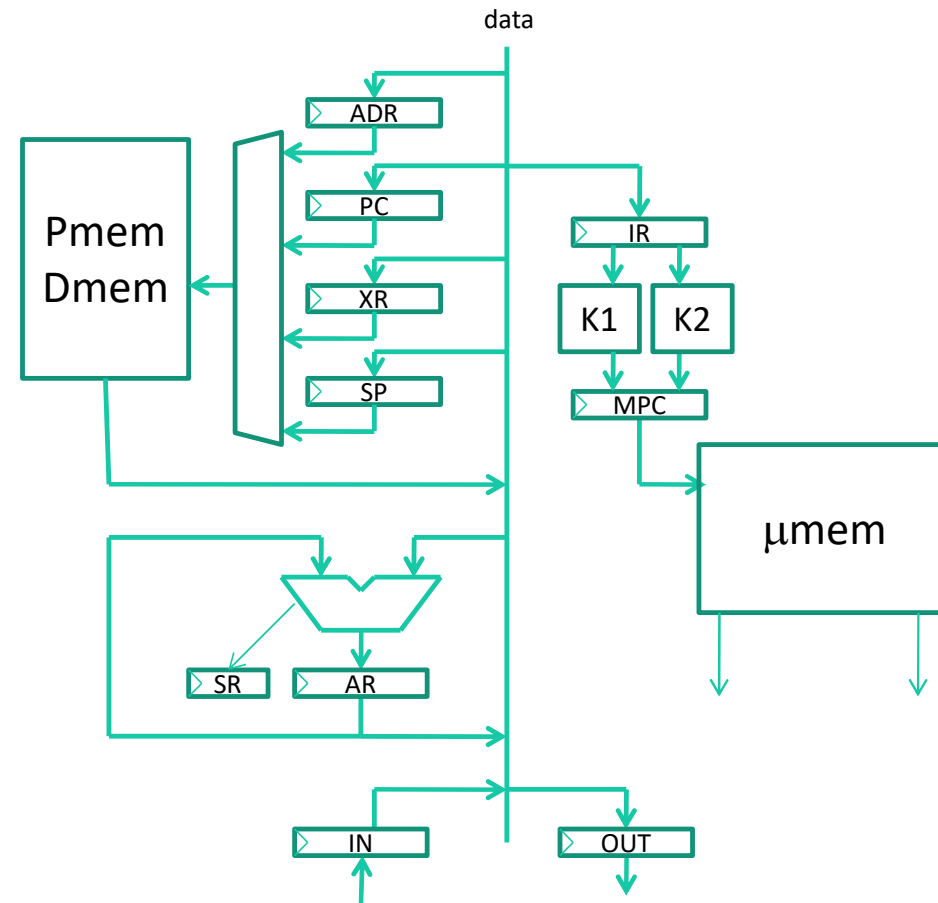
Brukar kallas kritisk väg

Steg 1

- + Omplacering av register
- + Ta bort onödiga register
- + Förbättrat SR
- Lång kritisk väg

Nya LDA 12

- | | |
|-----------|------|
| – Hämta | 1 CP |
| – K2 | 1 CP |
| – Absolut | 1 CP |
| – EXE | 1 CP |

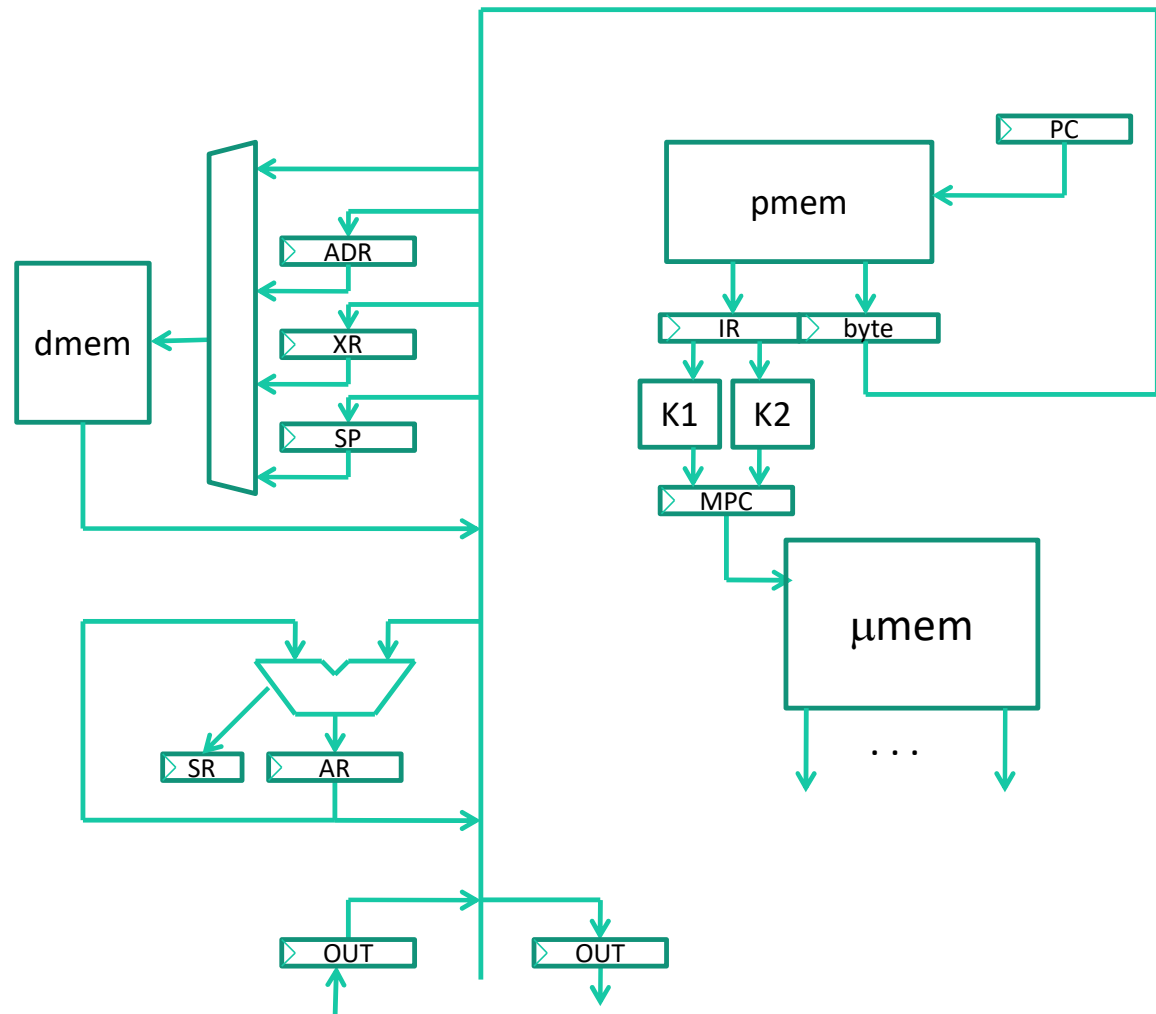


Steg 2

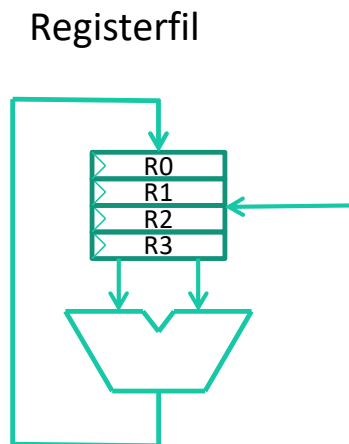
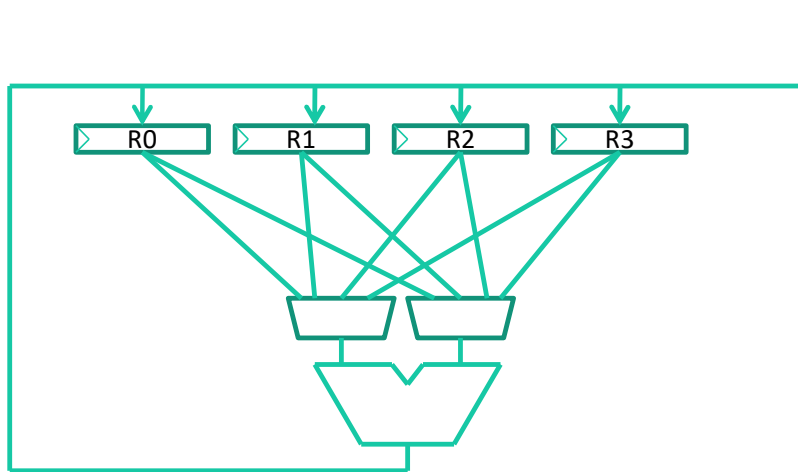
- Skilda program- och dataminnen
- Bredare programminne, så att hela instruktionen kan hämtas på 1 CP



- Nya LDA 12
 - Hämta 1 CP
 - K2 1 CP
 - Absolut
 - EXE 1 CP



Flera register!



Exempelvis:

ADD R3,R2,R1 ; $R3 = R2 + R1$

RISC = Reduced Instruction Set Computer

- Vi avskaffar XR,SP och inför generella register
- Vi avskaffar flera a-moder per instruktion
 - ADD Rx,Ry,Rz inga andra adr-moder!
 - LD Rx,(Ry) enda sättet att läsa i minnet!
 - ST (Rx),Ry enda sättet att skriva i minnet!
- Vi avskaffar MPC och mikroprogrammering
 - Mikroprogrammen försvinner inte utan finns på annan form i maskinen

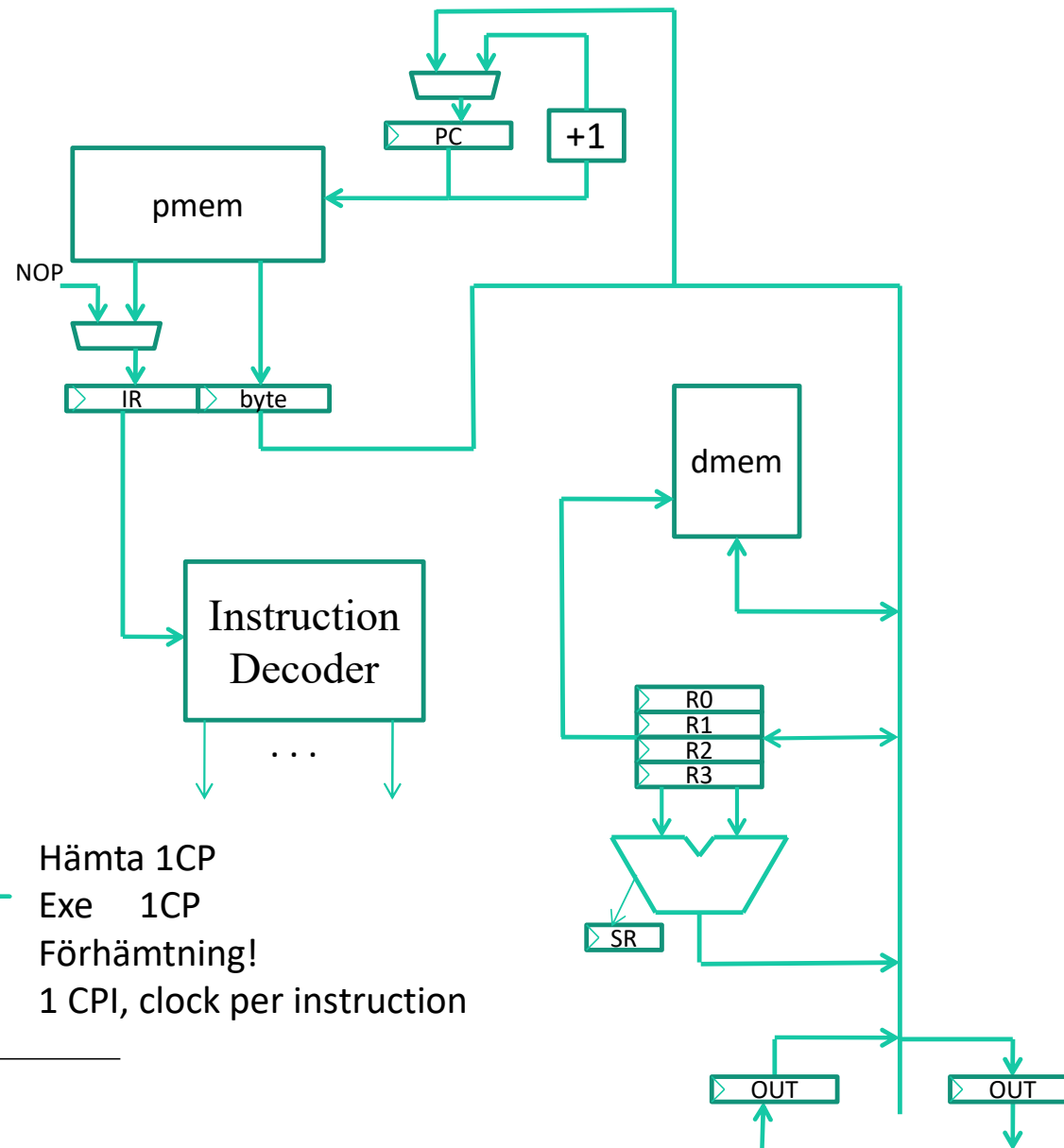
Steg 3

Hämtfasen har blivit HW

Instruction Decoder innehåller 1-rads μ prog

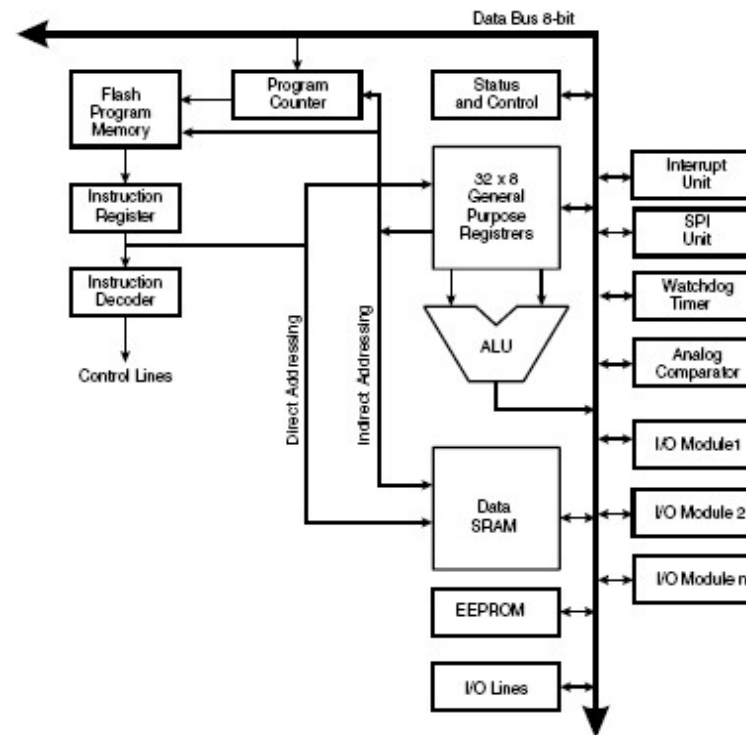
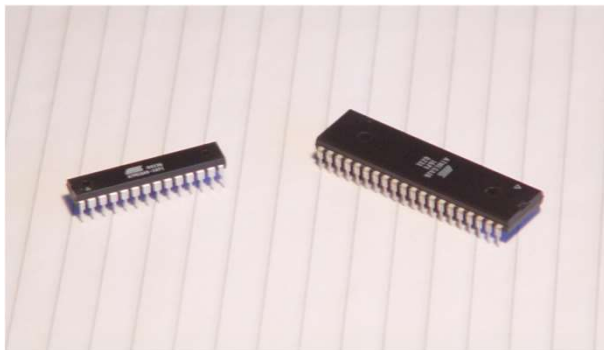
- ADD Rx,Ry,Rz
- LD Rx,(Ry)
- ST (Ry),Rx
- JMP N
- ADDI Rx,Ry,K

Hämta 1CP
Exe 1CP
Förhämtning!
1 CPI, clock per instruction



Datorkonstruktion Atmel AVR

- Trevlig 8-bitars controller
- C-kompilator finns: avr-gcc
- Massor med kul I/O



Pipelinediagram

```

0: LD  R1,(R0)    ; R1:=DM(R0)
1: ADD R3,R2,R1   ; R3:=R1+R2
2: JMP 0          ;
3: XXX

```

PC	0	1	2	3	0	1
Hämta		LD	ADD	JMP	NOP	LD
Exekvera			LD	ADD	JMP	NOP

1 inst/CK

JMP ger en NOP i pipelinen

Klassisk 5-steps pipeline

IF *Instruction Fetch*

RR *Register Read*

EXE *Execute*

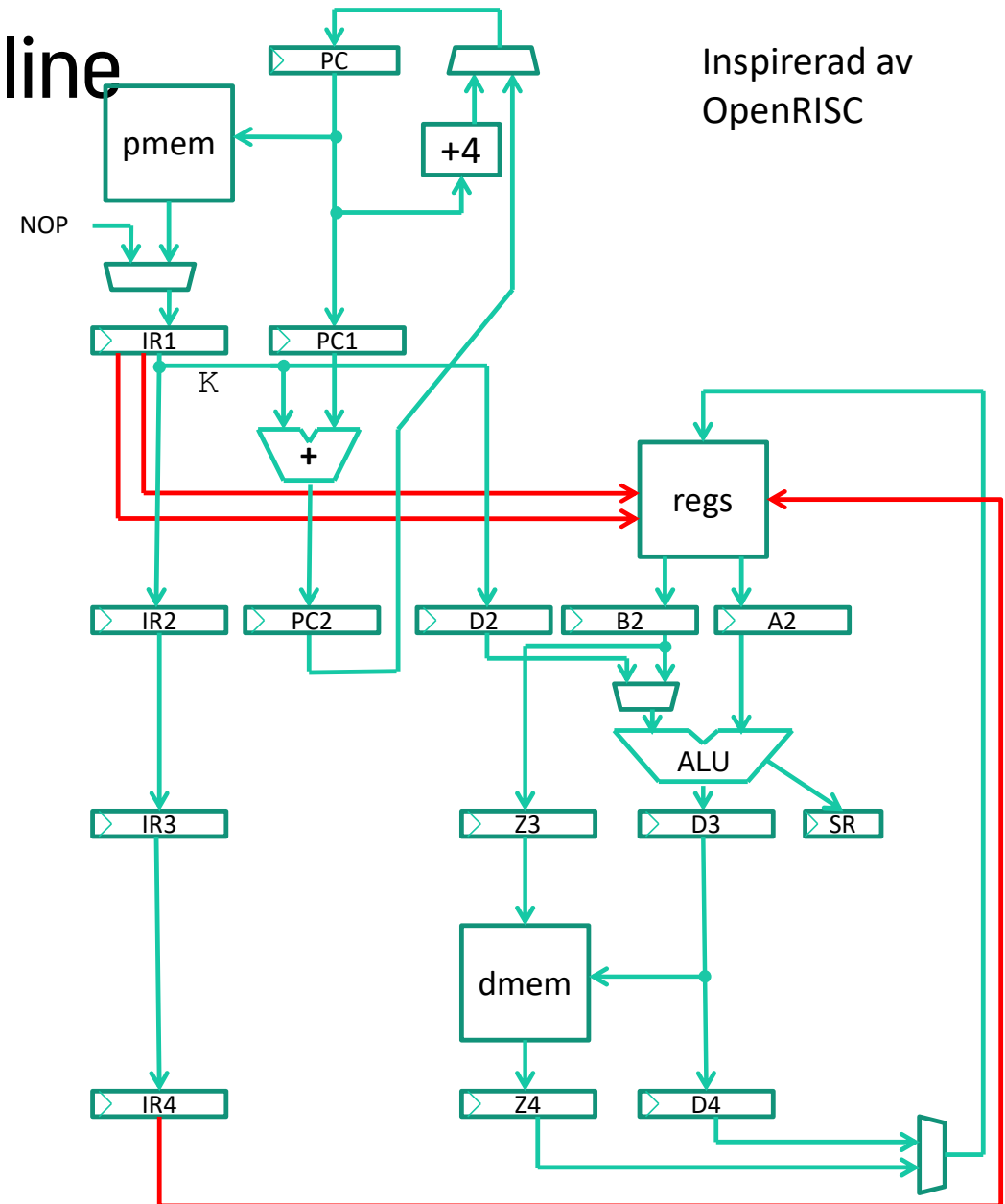
MEM *Memory access*

WB *Write Back*

Klassisk 5-steps pipeline

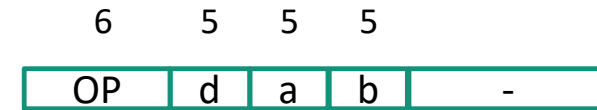
Inspirerad av
OpenRISC

- **IF:** instruction fetch
hämta instr och ny PC
- **RR:** register read
läs reg/beräkna hopp
- **EXE:** execute
kör ALU
- **MEM:** read/write dmem
läs/skriv/ingenting
- **WB:** write back register
skriv reg/ingenting

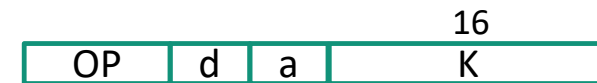


Några instruktioner, alla 32 bitar

ADD Rd, Ra, Rb ; Rd=Ra+Rb



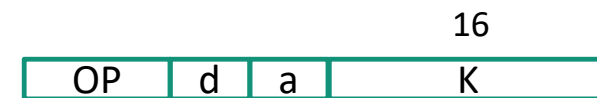
ADDI Rd, Ra, K ; Rd=Ra+K



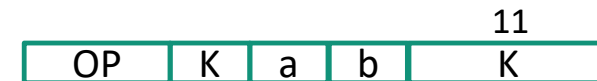
MOVHI Rd, K ; RdH=K, RdL=0



LD Rd, K(Ra) ; Rd=dmem(Ra+K)



ST K(Ra), Rb ; dmem(Ra+K)=Rb



SFEQ Ra, Rb ; F = (A==B)?1:0



BF K ; PC = F ? PC+K : PC+4



JMP K ; PC = PC+K



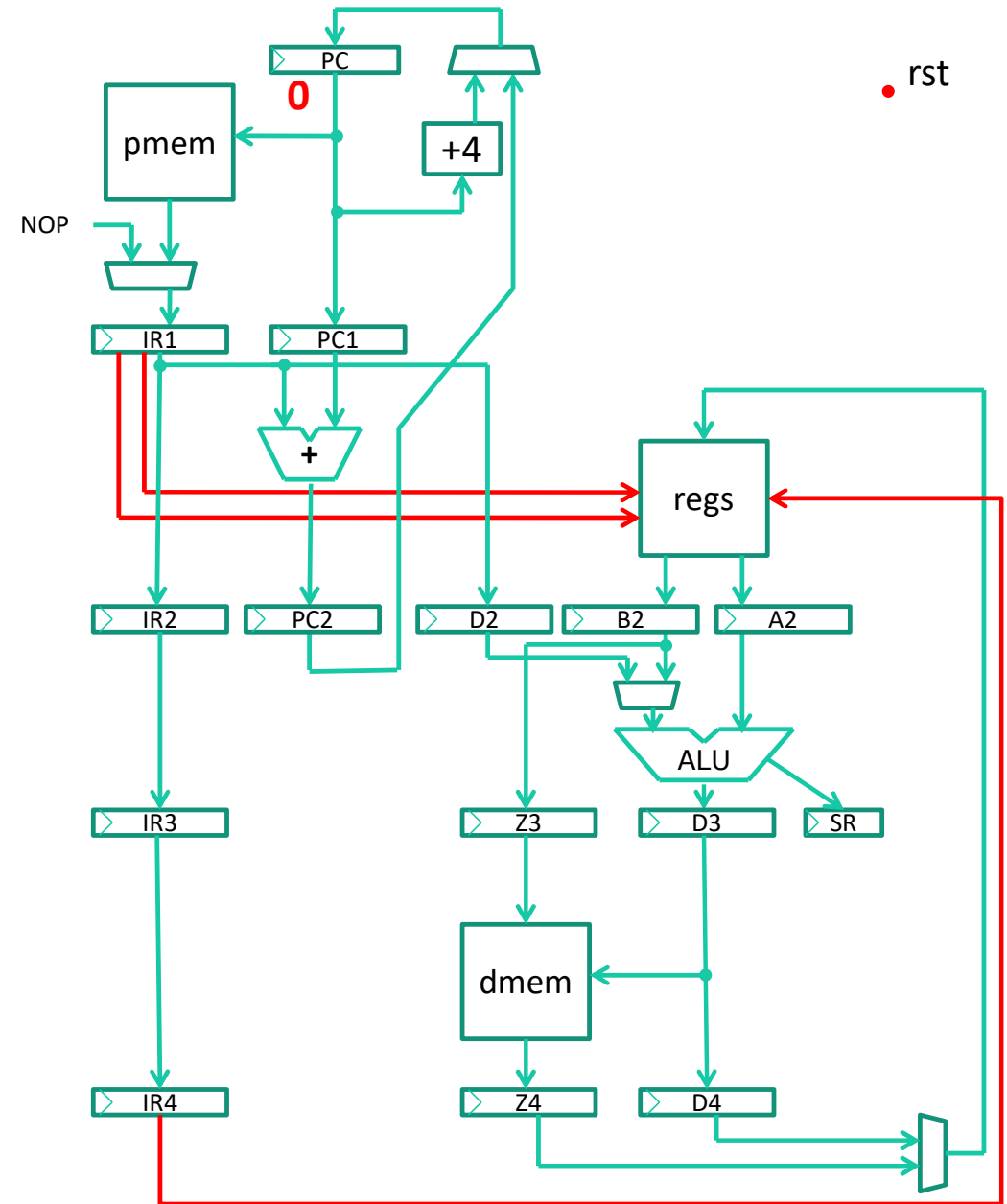
Några "snälla" instruktioner

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

I exemplet:

$R_n = n$

$K = 9$



Några "snälla" instruktioner

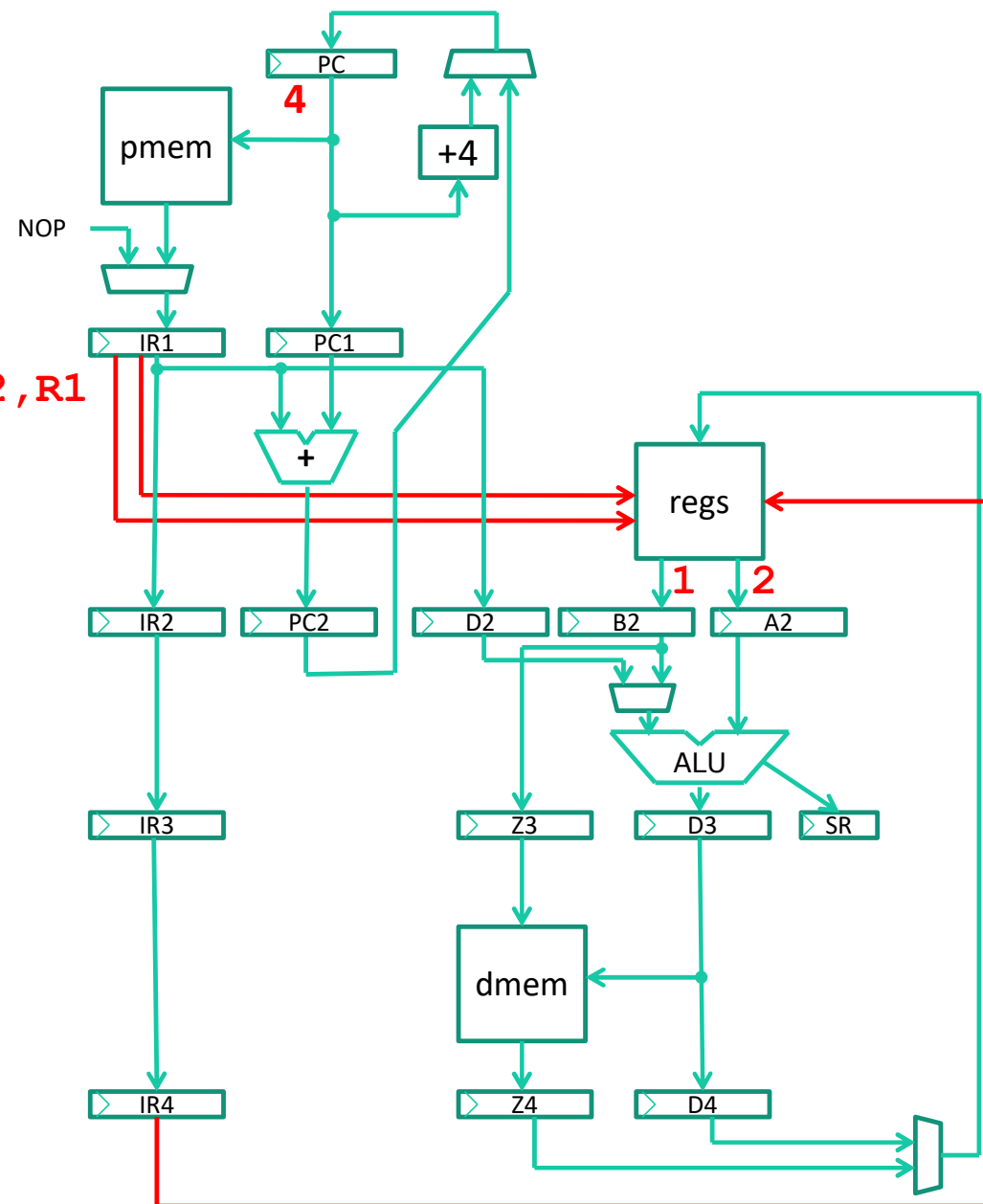
0: ADD R3, R2, R1
 4: LD R6, K(R5)
 8: SFEQ R7, R8
 C: XXX
 10: YYY
 14: ZZZ

I exemplet:

R_n = n

K = 9

ADD R3, R2, R1



Några "snälla" instruktioner

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

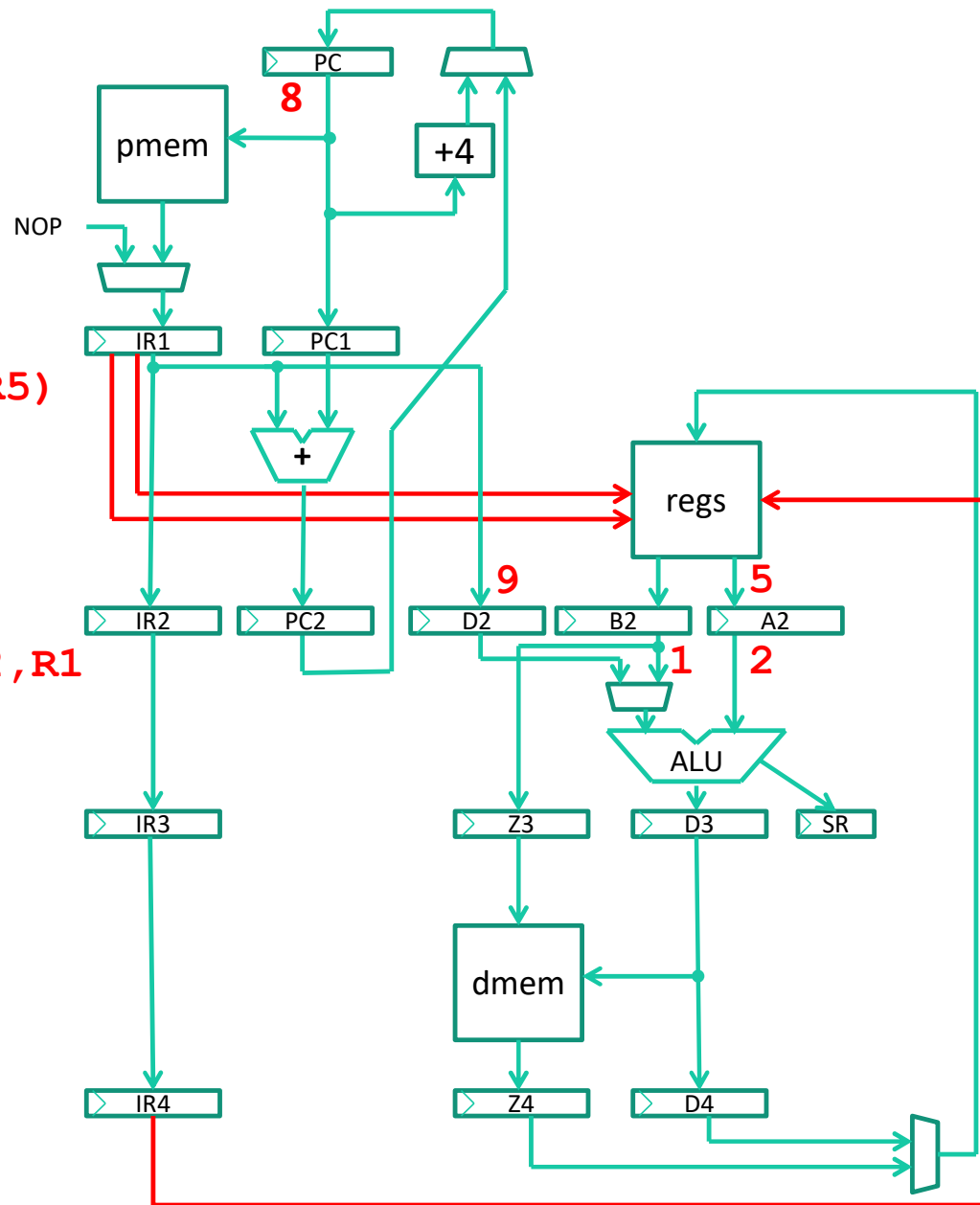
I exemplet:

R_n = n

K = 9

LD R6, K(R5)

ADD R3, R2, R1



Några "snälla" instruktioner

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

I exemplet:

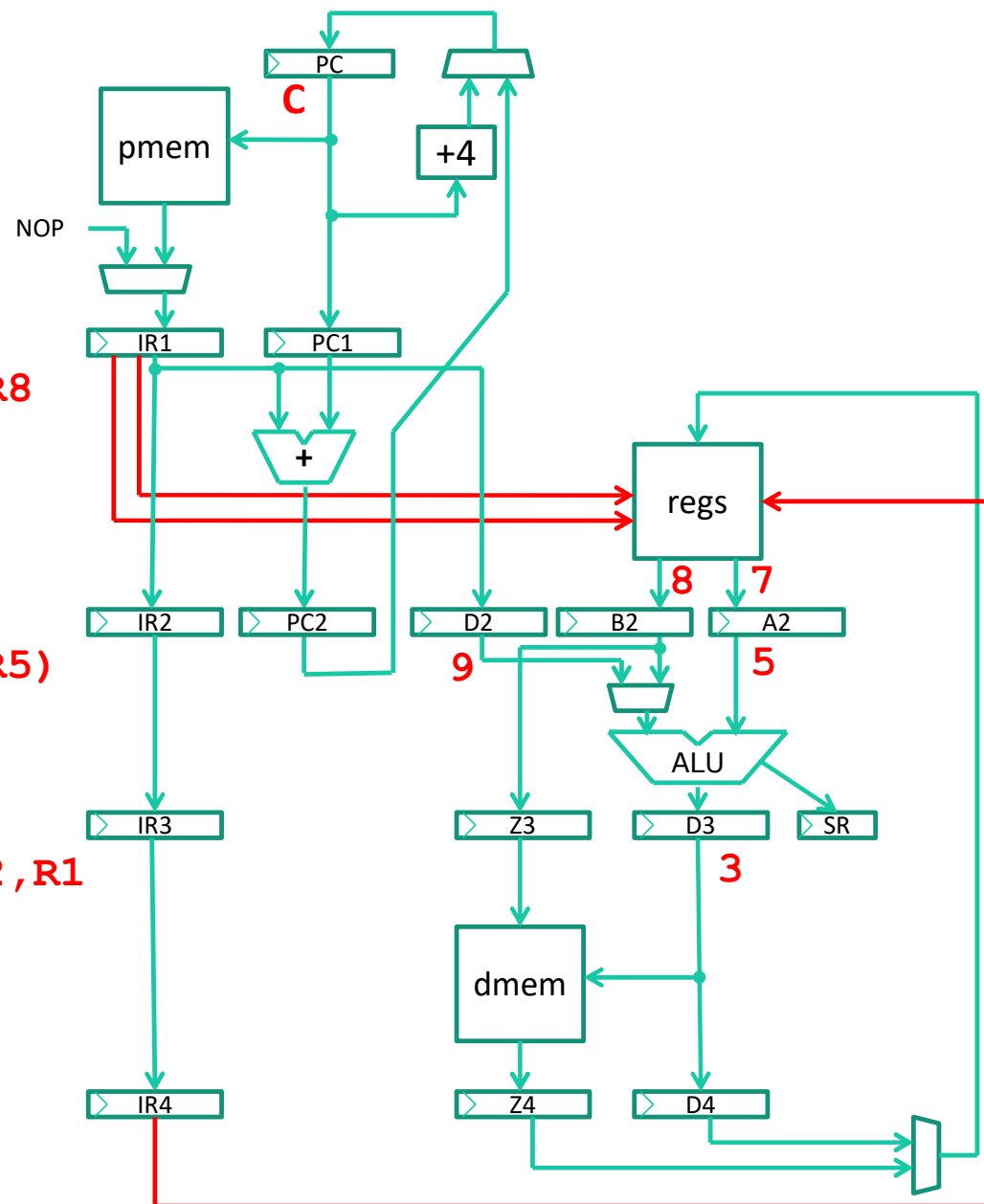
Rn = n

K = 9

SFEQ R7,R8

LD R6,K(R5)

ADD R3,R2,R1



Några "snälla" instruktioner

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

I exemplet:

Rn = n

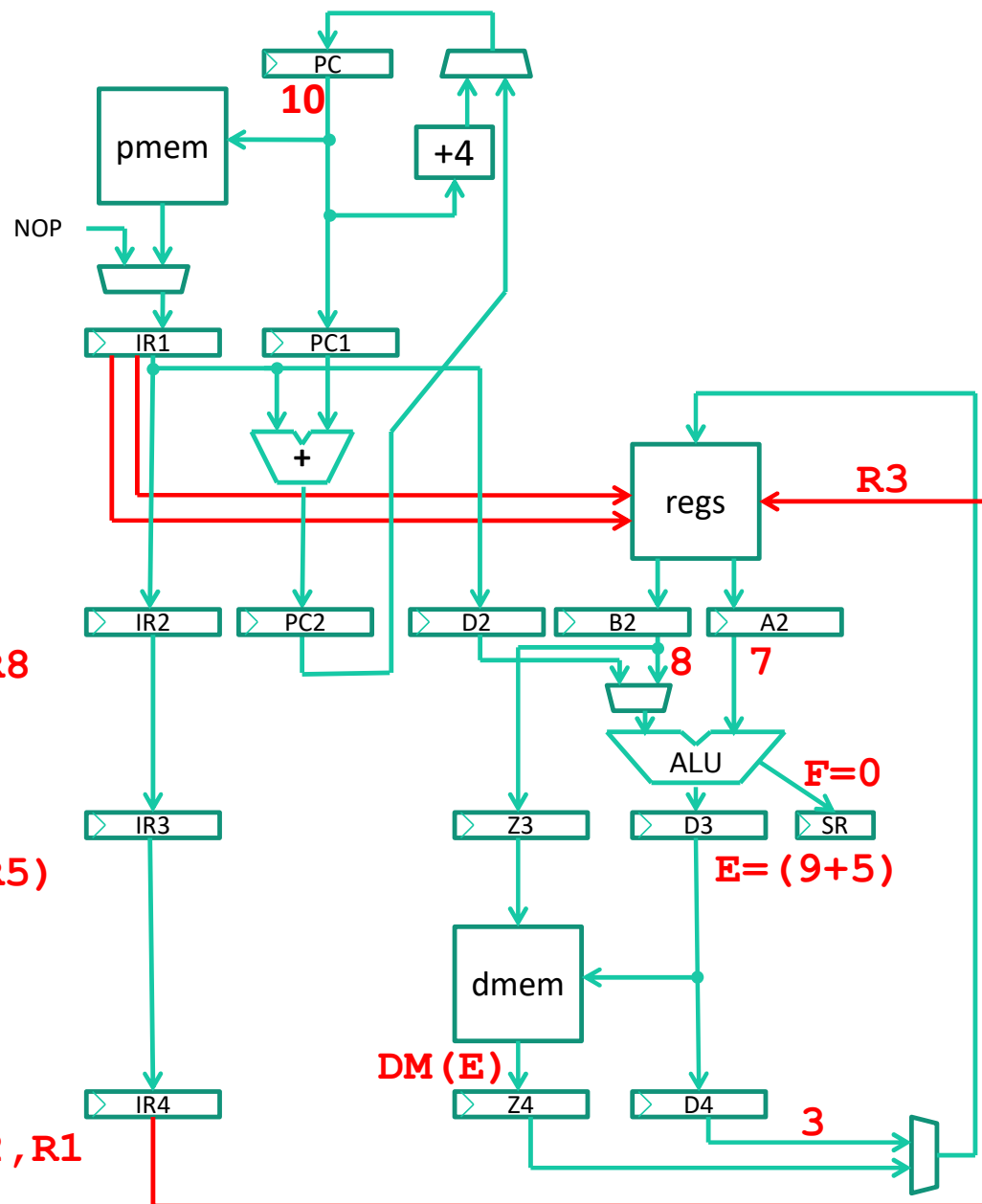
K = 9

XXX

SFEQ R7,R8

LD R6,K(R5)

ADD R3,R2,R1



Några "snälla" instruktioner

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

I exemplet:

R_n = n

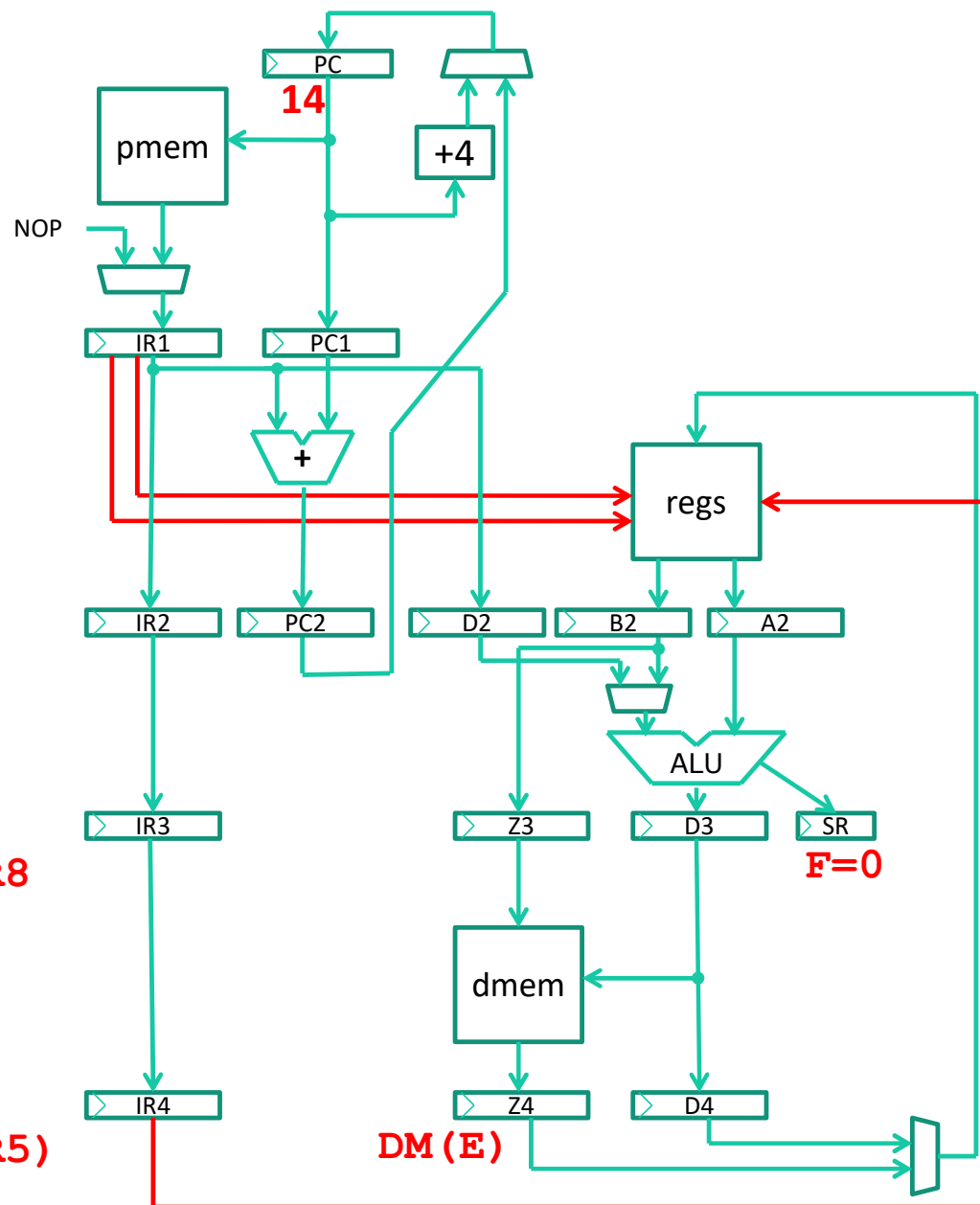
K = 9

YYY

XXX

SFEQ R7, R8

LD R6, K(R5)



Klassisk 5-steps pipeline

Problem ...

Problem ...

1. **Hopp**, Öönskade instr kommer ibland in i pipelinen

- >Instruktionen efter ett hopp exekveras alltid
- >Ytterligare en instr. därefter ersätts med NOP

2. **Databeroenden**, Samma reg som läses i steg 2, skrivs också i steg 5

- >**data forwarding**

3. Pipelinen måste i vissa lägen stängas av, **pipeline stall**

Problem 1: Hopp

Antag följande program

```
0:SFEQ ... ; sätter flaggan F
4:BF L      ; hoppar eventuellt till 20
8:XXX       ; instr. exekveras alltid
C:YYY
...
20:ZZZ
```

Hoppet **BF K**, utförs först i exekveringssteget, dvs om **F=1** och vi ska hoppa så kommer **XXX** att hinna in i pipen innan vi hoppar.

Därför bestämmer vi:

Instruktionen

XXX

exekveras alltid

Fördröjt hopp: **XXX** kan vara

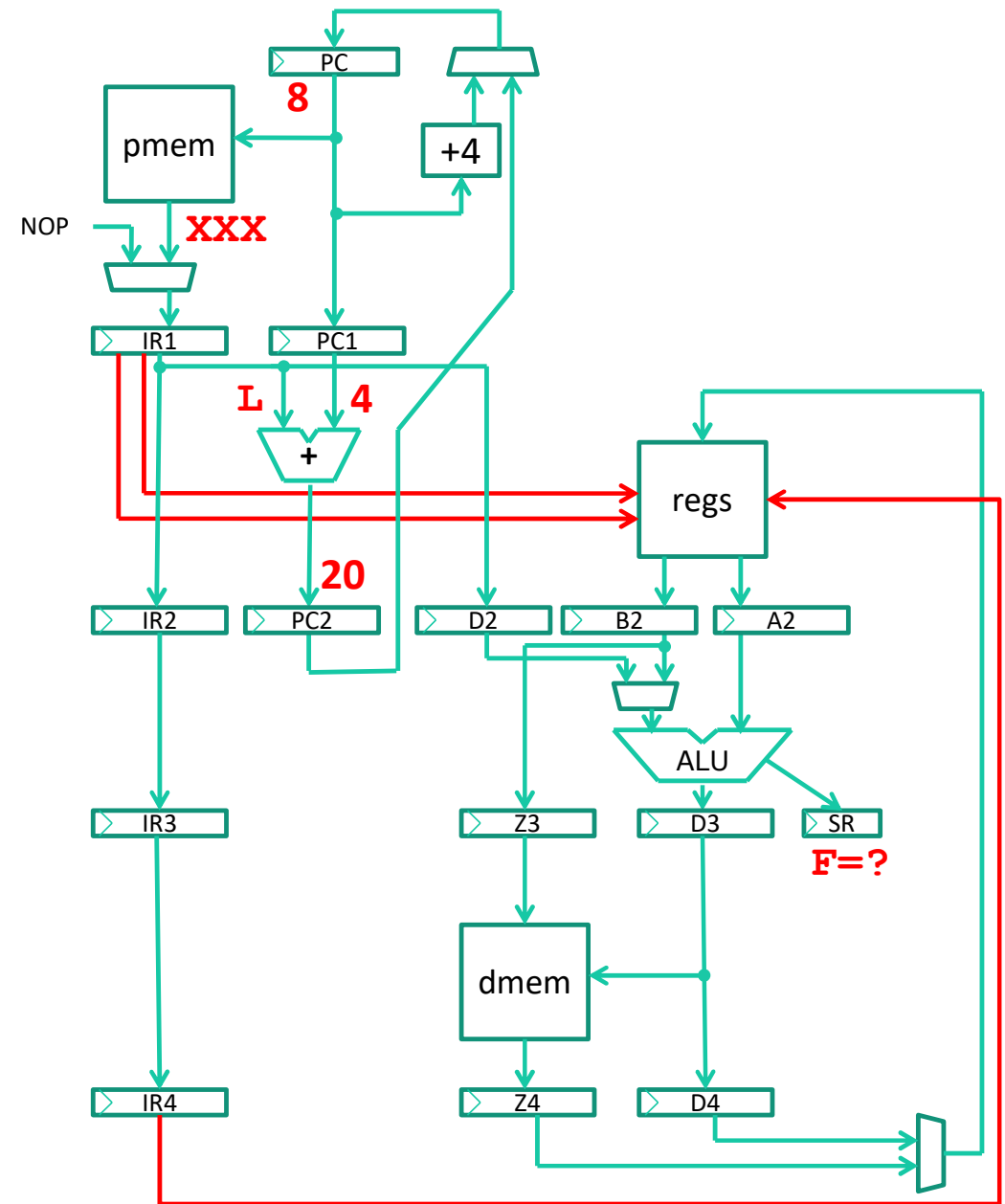
- en nyttig instr flyttad hit av kompilatorn
- (software) NOP

Problem 1: Hopp

0: SFEQ ...
 4: BF L
 8: XXX
 C: YYY
 ...
 20: ZZZ

BF L

SFEQ ...



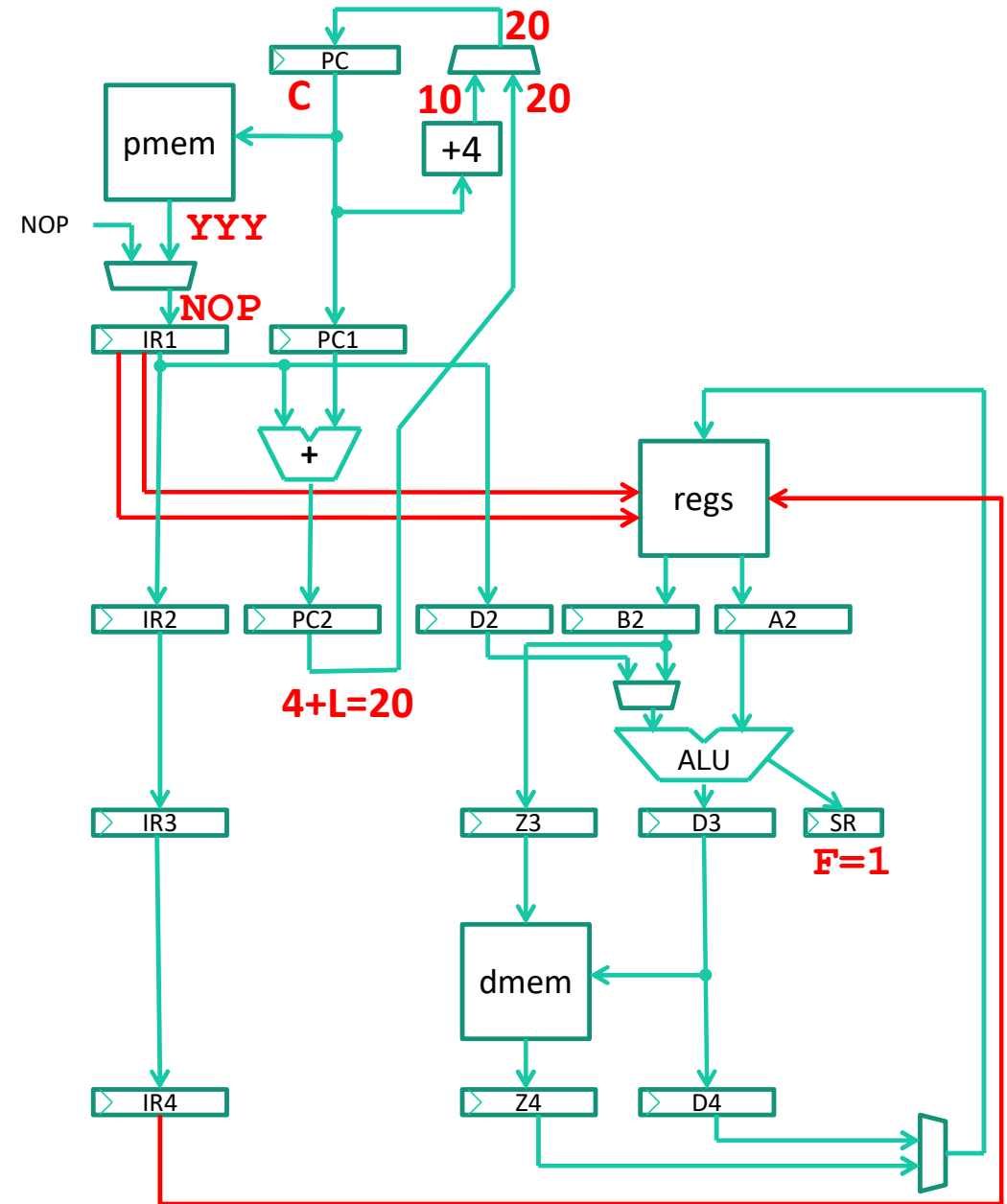
Problem 1: Hopp

0: SFEQ ...
 4: BF L
 8: XXX
 C: YYY
 ...
 20: ZZZ

XXX

BF L

SFEQ ...



Problem 1: Hopp

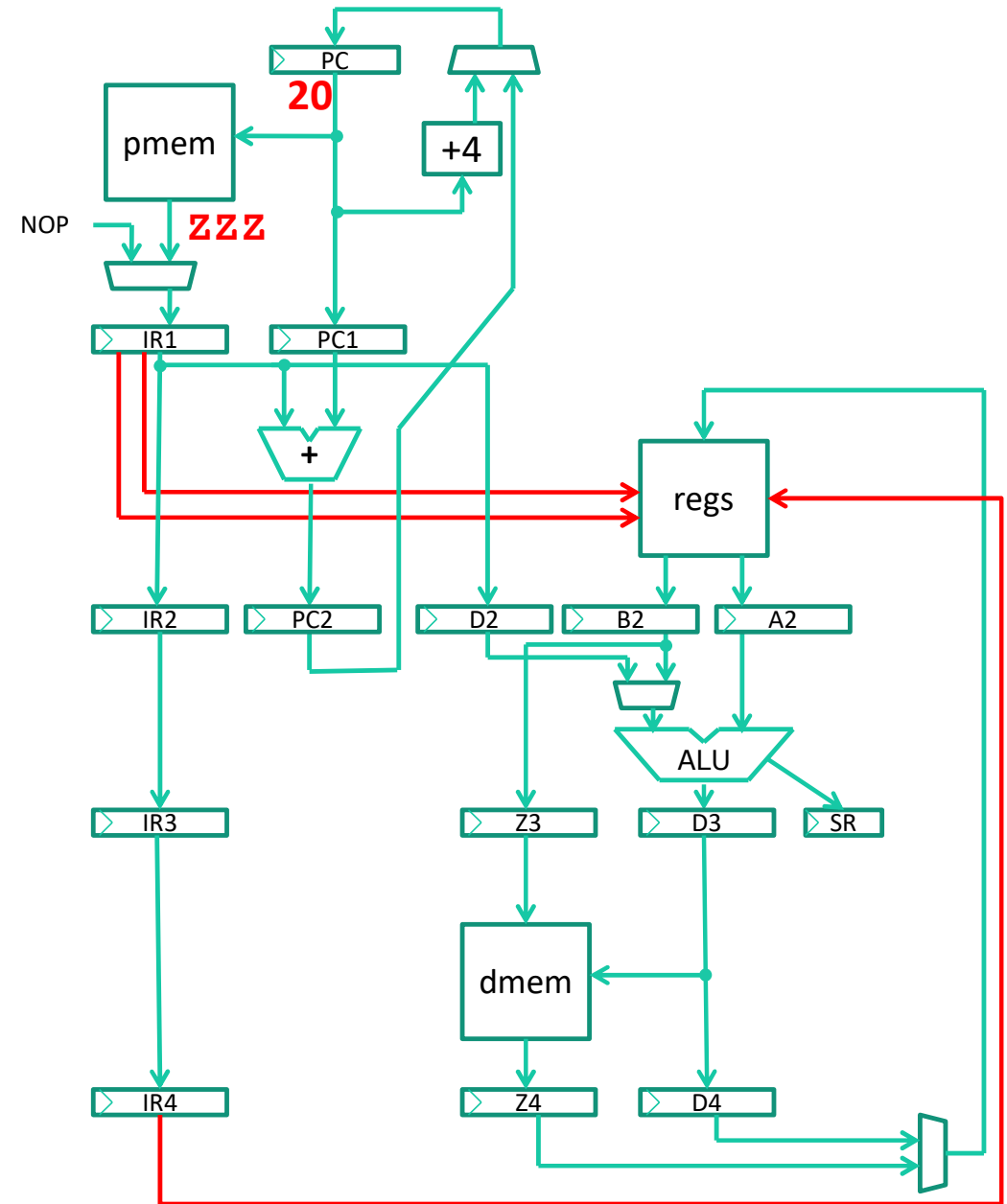
0 : SFEQ ...
 4 : BF L
 8 : XXX
 C : YYY
 ...
 20 : ZZZ

NOP

XXX

BF L

SFEQ ...



Problem 1: Hopp

Pipelinediagram

PC	IR1	IR2	IR3	IR4
0				
4	SFEQ			
8	BF	SFEQ		
C	XXX	BF	SFEQ	
20	NOP	XXX	BF	SFEQ

Instr efter hoppet
exekveras alltid

Vid taget hopp måste
en NOP muxas in

Diskussion: Hur ska de två muxarna i första steget styras?

if ((IR2.op==BF and F==1) or (IR2.op==J)) /* taget hopp */

IR1 = NOP; "ena muxen"

PC = PC2; "andra muxen"

else // annan instr, F=0

IR1 = pmem; "ena muxen"

PC = PC+4; "andra muxen"

6	5	5	5	
OP	d	a	b	-

Problem 2: Databeroende

0 : ADD R5 , R2 , R1

4 : ADD R8 , R5 , R5

8 : MUL R3 , R5 , R5

Farlig situation, som måste
åtgärdas direkt! Data Hazard

Problemet beror på att det tar flera klockcykler
innan registret R5 uppdateras!

Problem 2: Databeroende

0: **ADD** R5, R2, R1

4: **ADD** R8, R5, R5

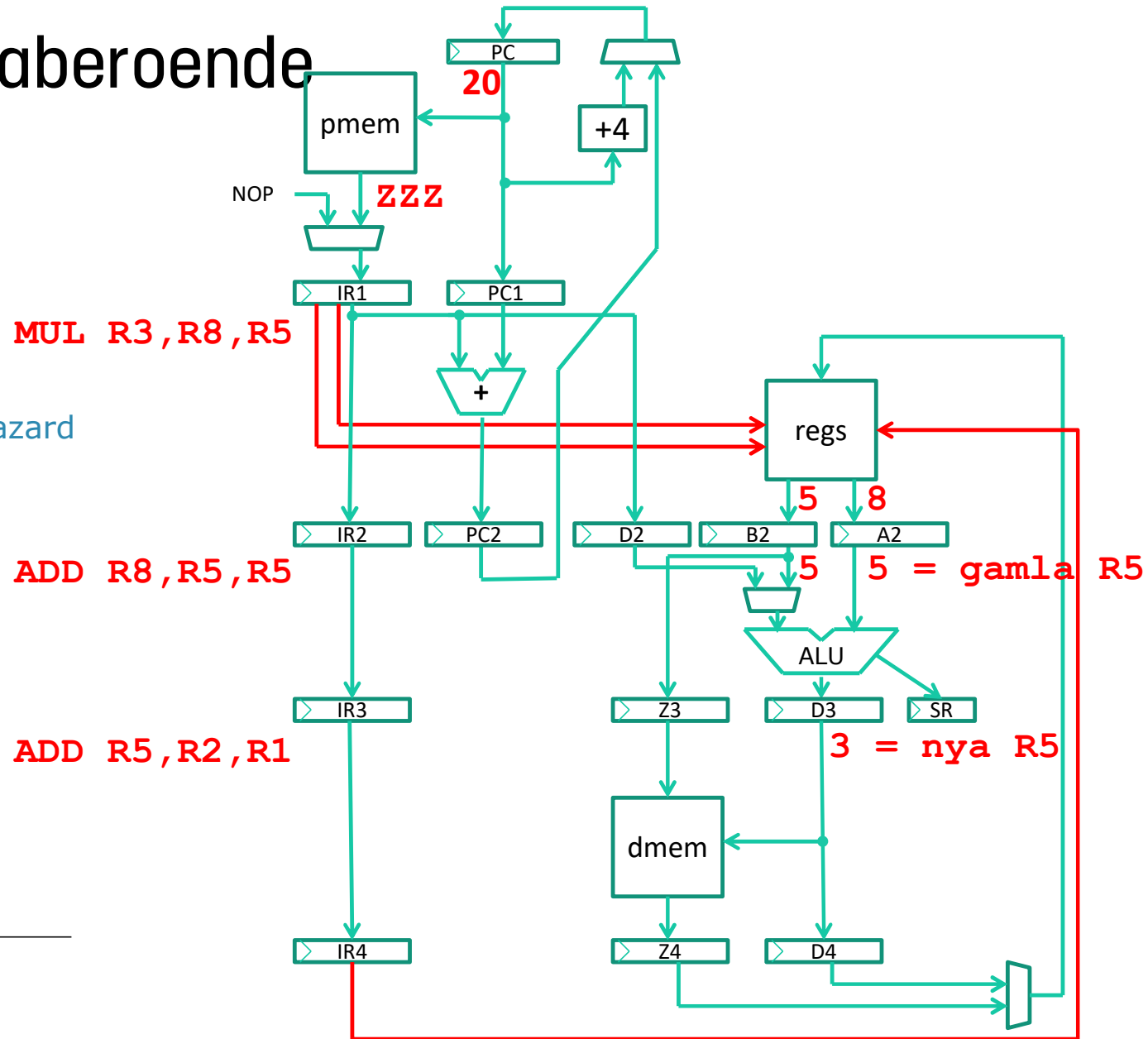
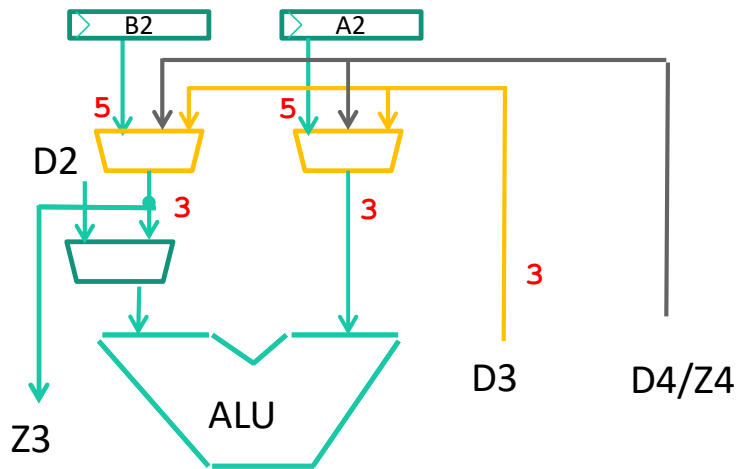
8: **MUL** R3, R5, R5

Antag $R_n = n$

Registerfilen kan innehålla gamla data!

Nya resultat finns i D3, D4/Z4, **Data hazard**

Löses med **Data forwarding**



Datorkonstruktion

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

4:ADD R8,R5,R5

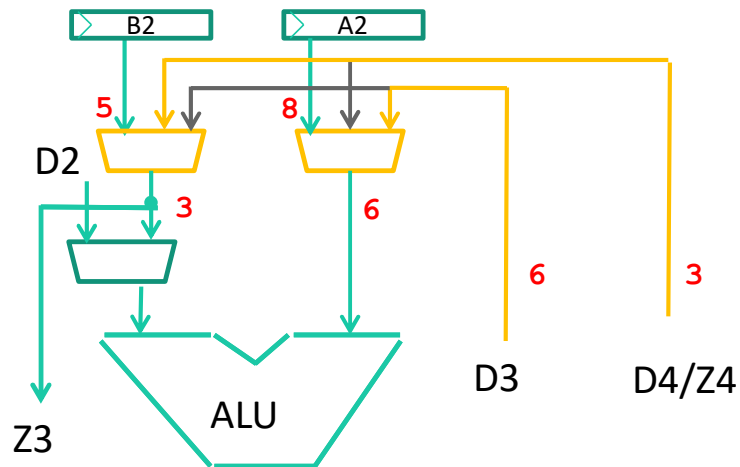
```
8: MUL  R3, R5, R5
```

Antag $R_n = n$

Registerfilen kan innehålla gamla data!

Nya resultat finns i D3, D4/Z4, **Data hazard**

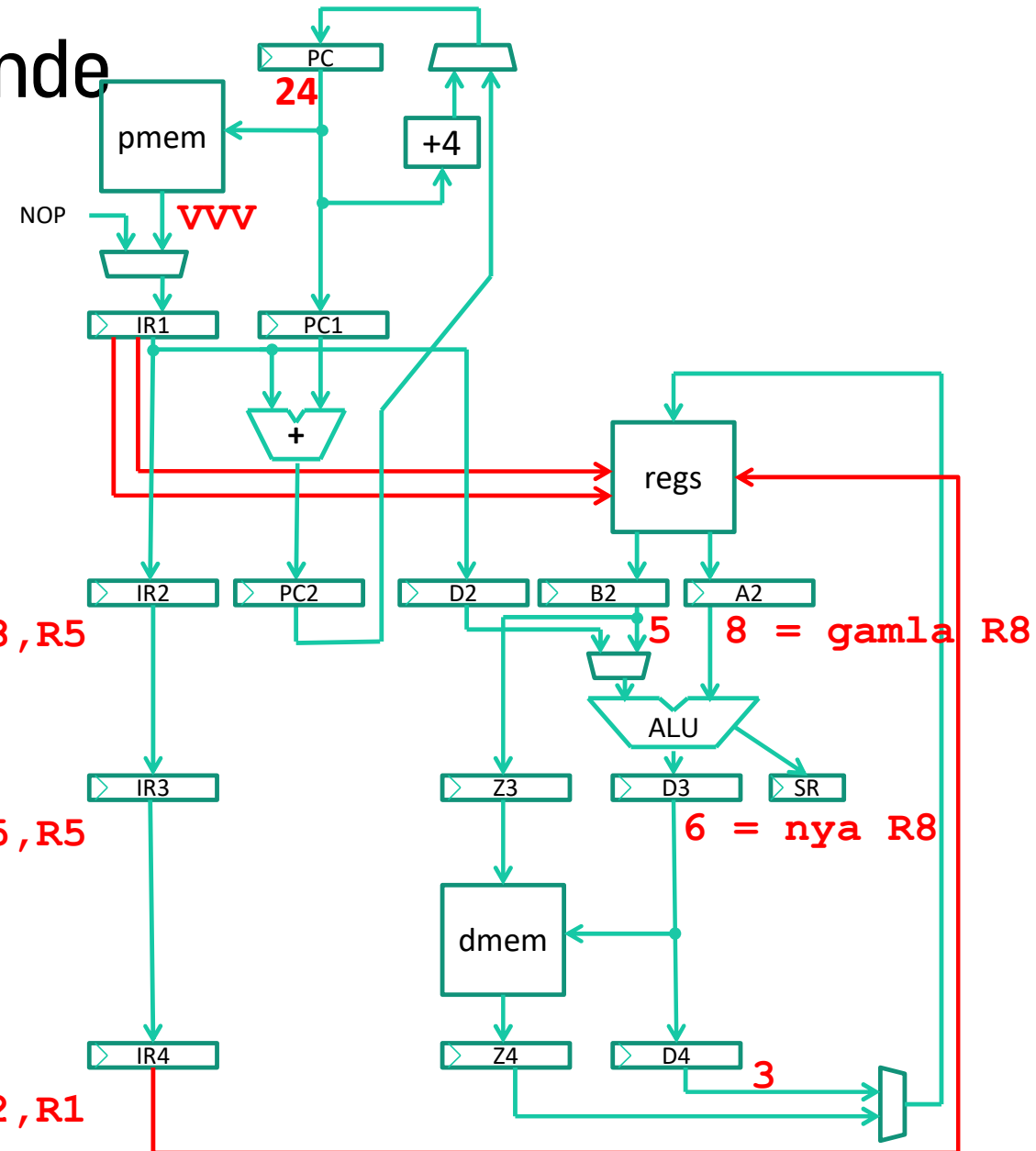
Löses med Data forwarding



MUL R3,R8,R5

ADD R8,R5,R5

ADD R5,R2,R1



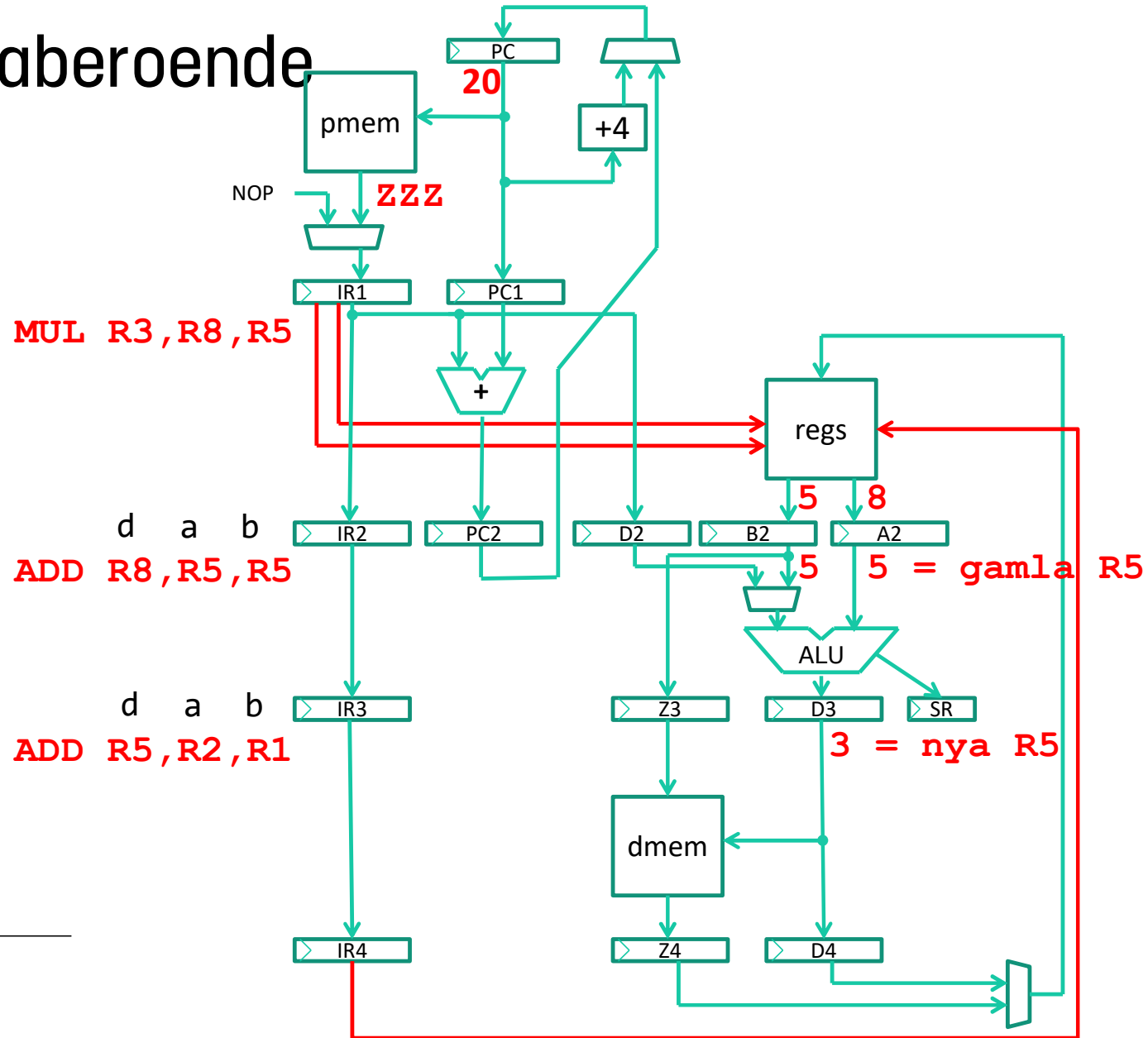
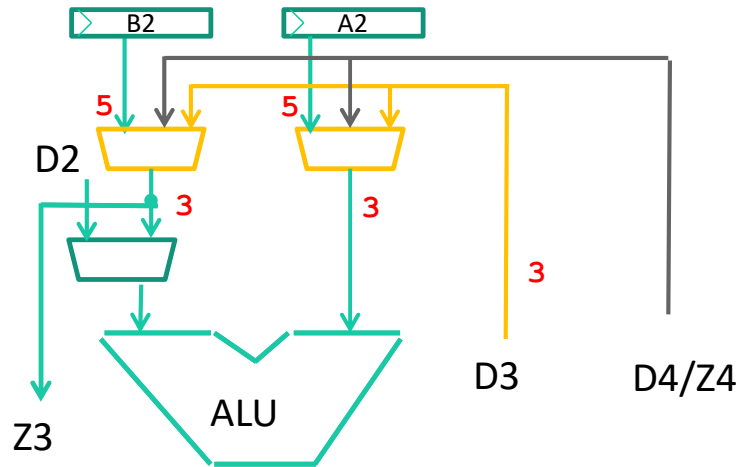
```
0:ADD  R5,R2,R1
4:ADD  R8,R5,R5
8:MUL  R3,R5,R5
```

Ena situationen:

```

if IR2.op == "instruktion som läser reg."
    if IR3.op == "instruktion som skriver reg."
        if IR2.a == IR3.d or IR2.b == IR3.d
            gula muxarna = ...;

```

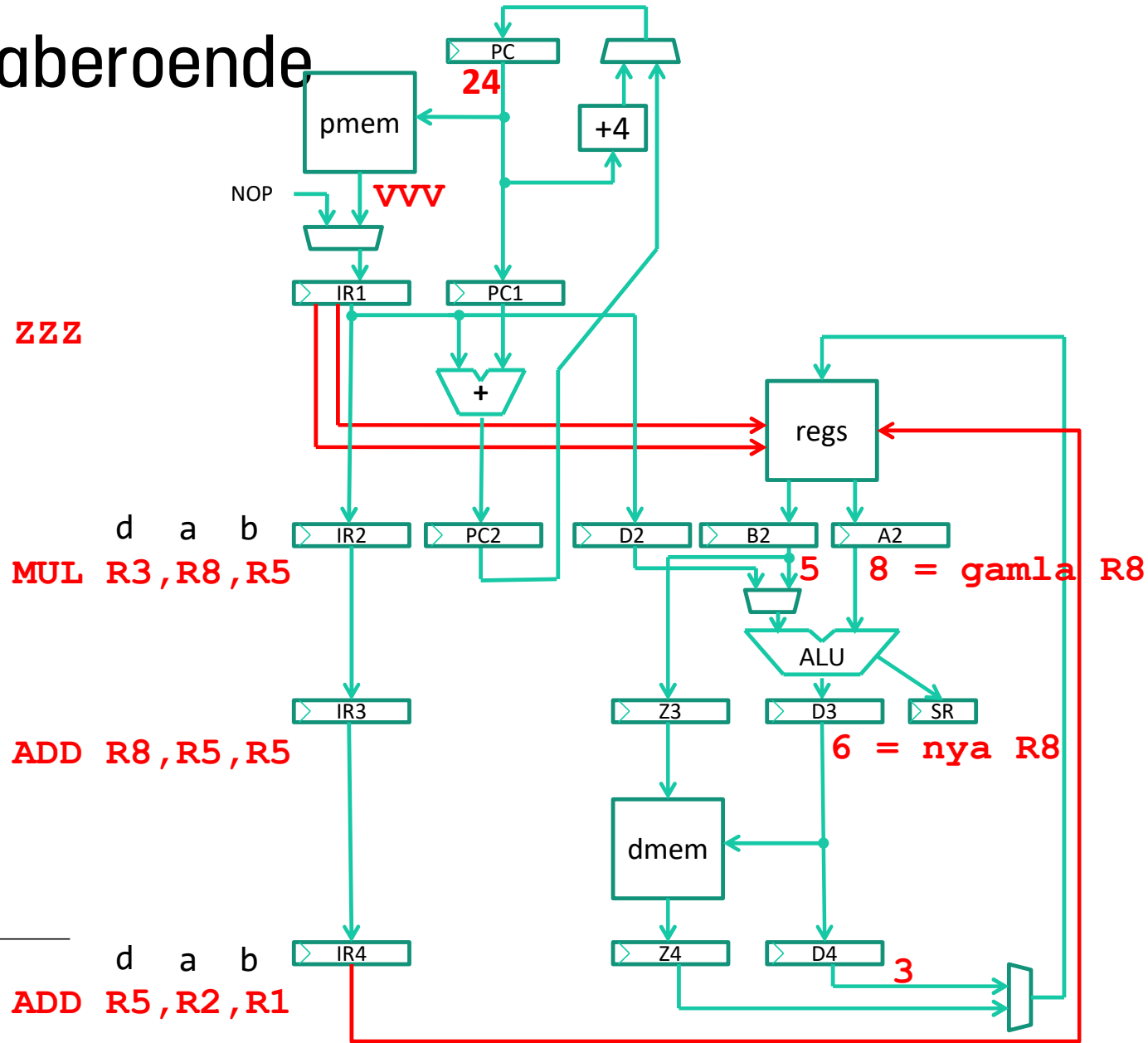
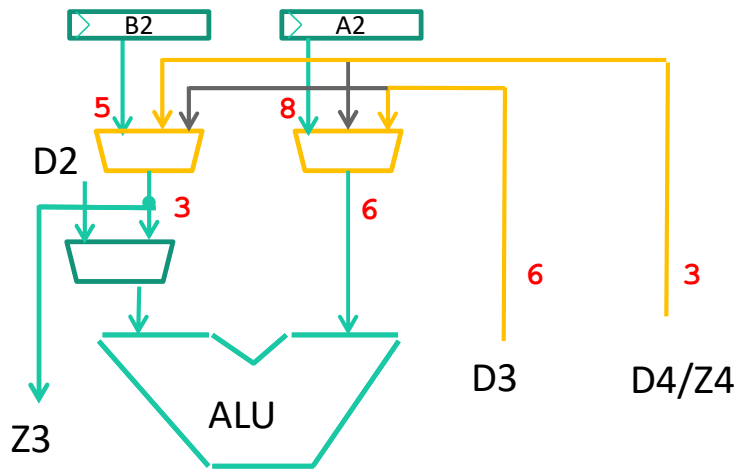


Datorkonstruktion

```
8:MUL R3,R5,R5
```

Andra situationen:

gula muxarna = ...;



Problem 2: Databeroende

Sammantaget:

Hur ska de gula muxarna styras?
Det finns två situationer:

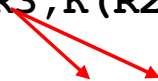
```
if IR2.op == "instruktion som läser reg."  
  if IR3.op == "instruktion som skriver reg."  
    if IR2.a == IR3.d or IR2.b == IR3.d  
      muxarna = ...;
```

```
if IR2.op == "instruktion som läser reg."  
  if IR4.op == "instruktion som skriver reg."  
    if IR2.a == IR4.d or IR2.b == IR4.d  
      muxarna = ...;
```

Problem 3: Minnesberoende

0: LD R3, K(R2) ; läs från minnet

4: ADD R4, R3, R3 ;



Problemet beror på att minnet sitter 1 klockcykel efter ALU-n.

Problem 3: Minnesberoende

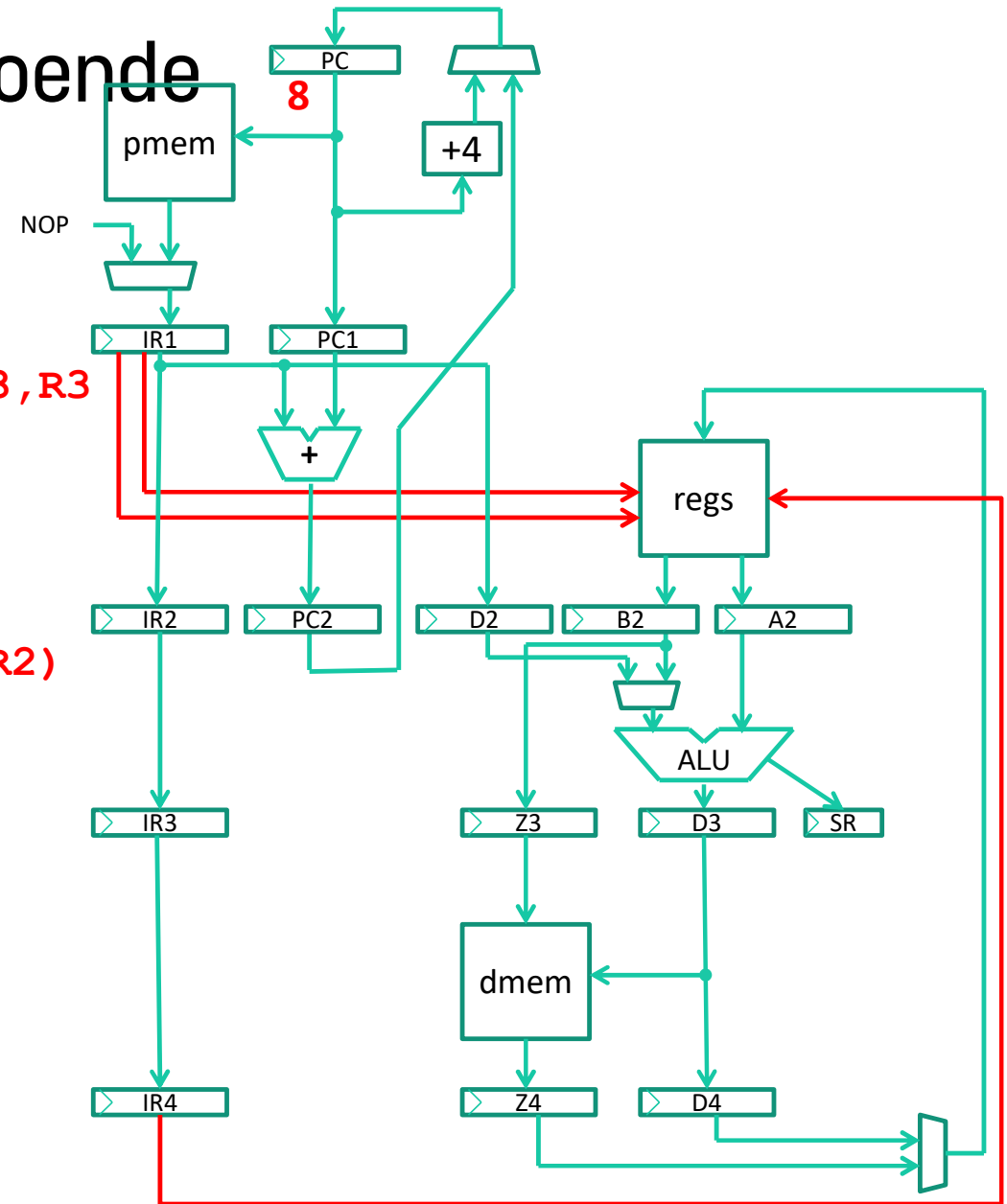
0: LD R3, K(R2)

4: ADD R4, R3, R3

ADD-instruktionen läser gamla R3
LD har ännu inte läst minnet, och
 nytt värde på R3 kan fås (via Z4)
 först två steg senare (i writeback).

ADD R4, R3, R3

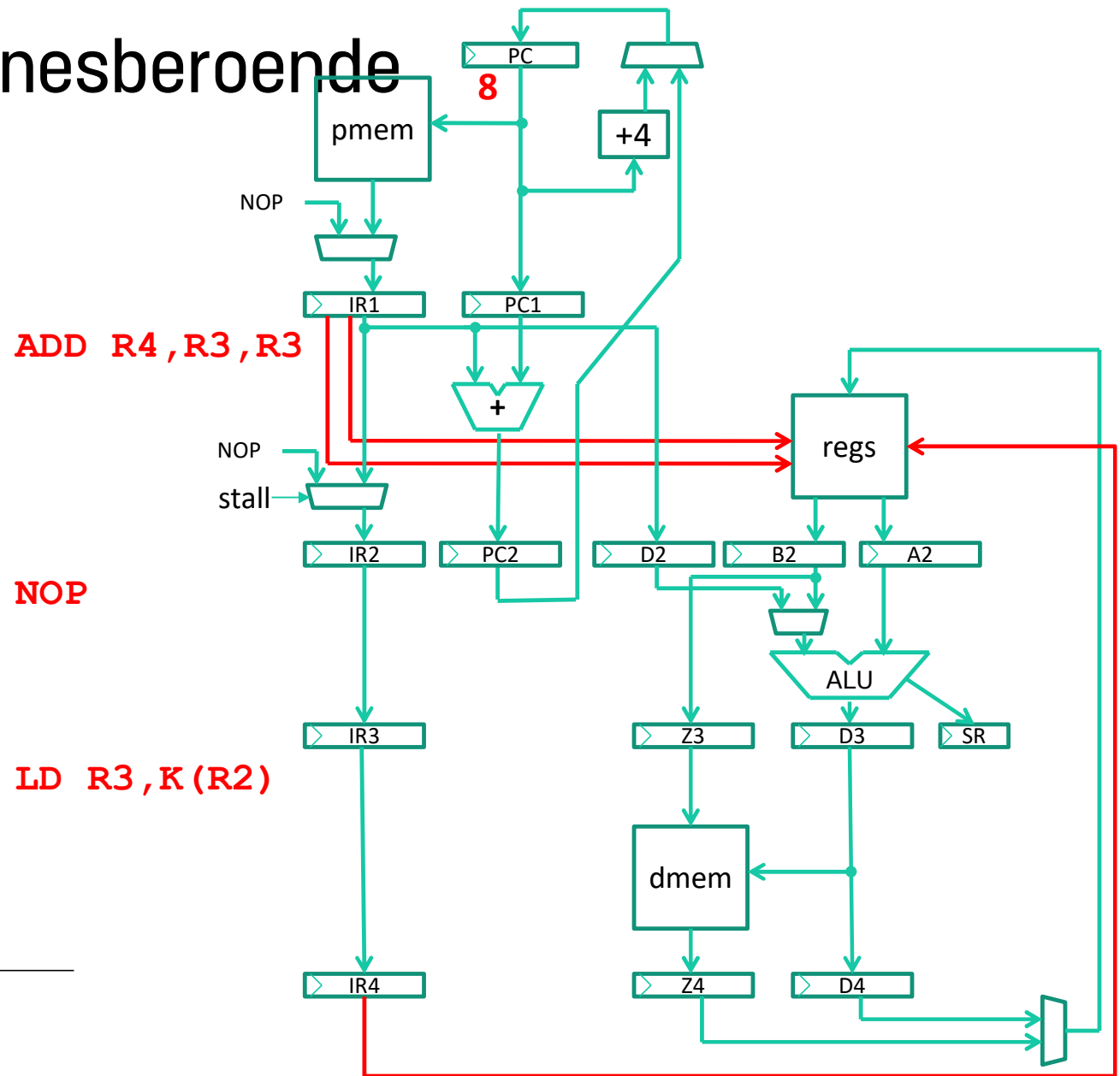
LD R3, K(R2)



Problem 3: Minnesberoende

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

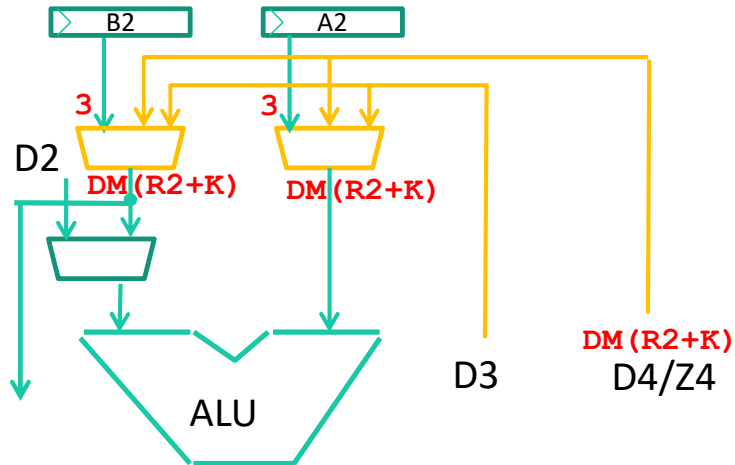
Löses med Pipeline stall
kräver en till mux, som petar in
en **NOP** (mellan **ADD** och **LD**),
samt att **ADD** står stilla (stall).



Problem 3: Minnesberoende

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

Nu löser sig resten av problemet med hjälp av data forwarding.

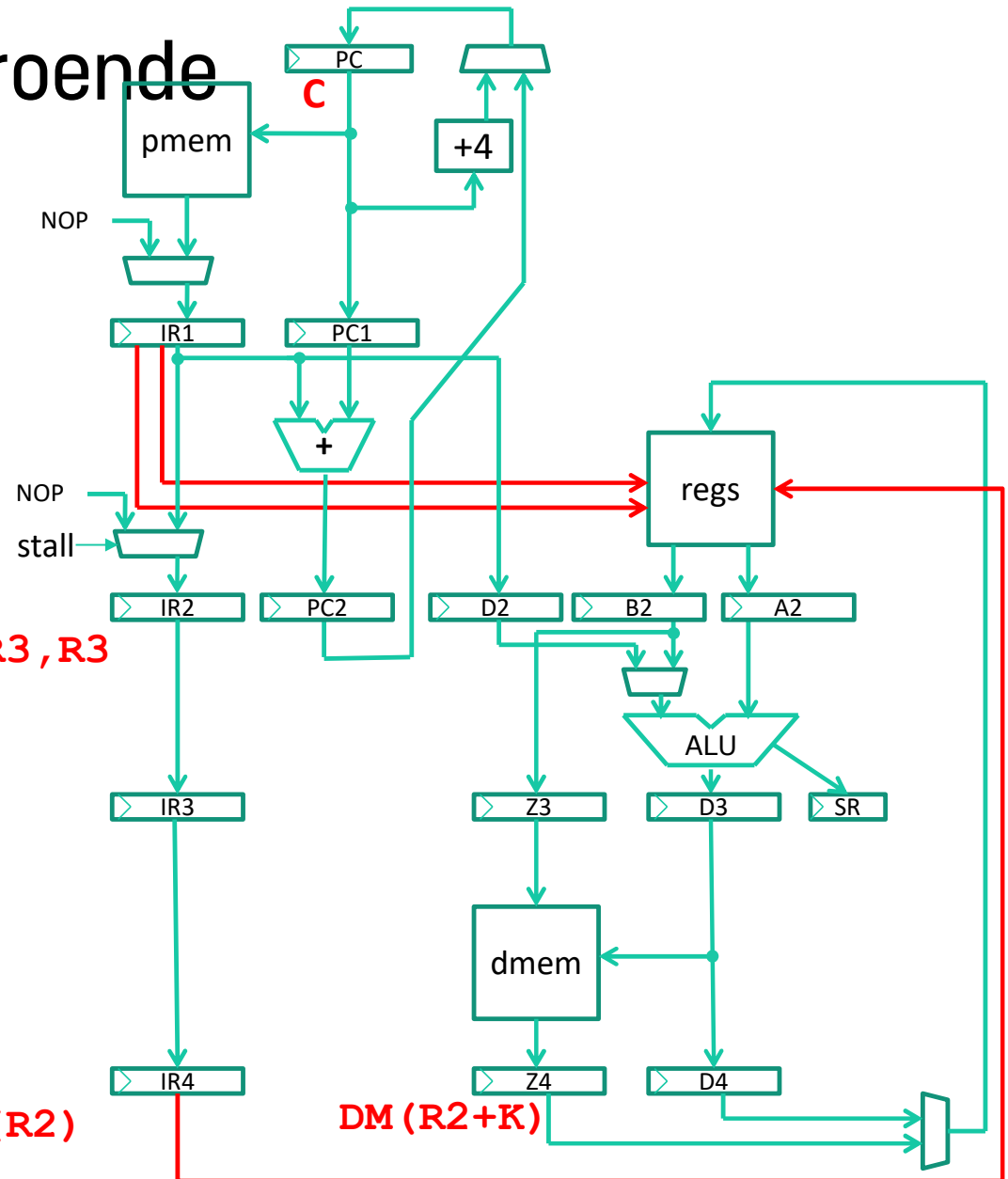


XXX

ADD R4, R3, R3

NOP

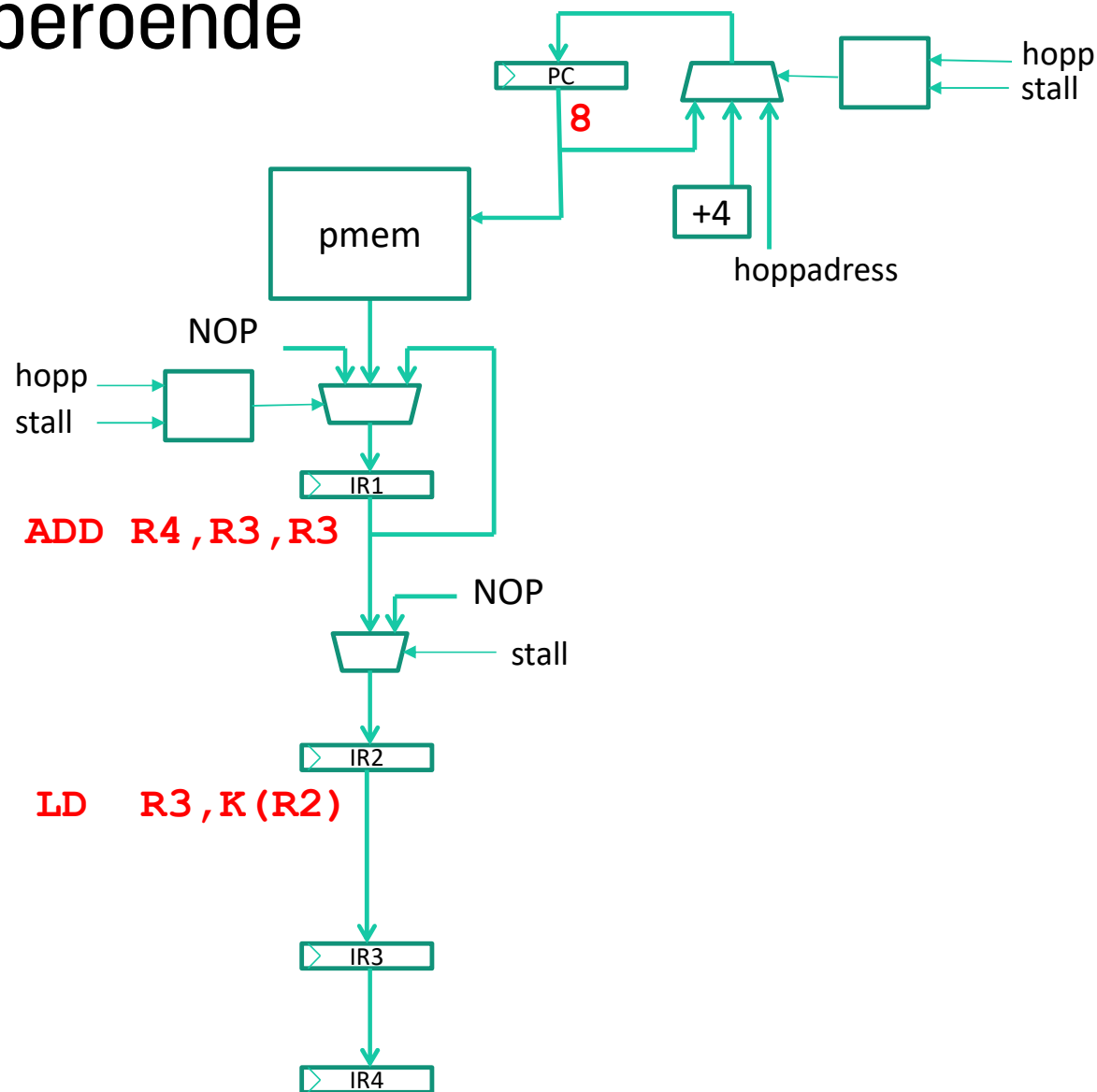
LD R3, K(R2)



Problem 3: Minnesberoende

Vid stall:

- NOP -> IR2
- Behåll IR1
- Behåll PC



Problem : Sammanfattning

1. Hopp
Vid hopp hinner efterföljande instruktion in i pipen innan hoppet tagits.
Lösning: **exekvera alltid instruktionen efter hoppet**
2. Databeroende
Data hinner inte skrivas tillbaka (i WB-steget) innan efterföljande instruktion(er) behöver dom.
Lösning: **data forwarding**
3. Minnesberoende
Läsning från minnet hinner inte göras innan efterföljande instruktion vill använda data från läsningen.
Lösning: **pipeline stalling + data forwarding**

Lab2

Pipelining

Lab2 : Pipelining

Labben baseras på en delmängd av den riktiga processorn OpenRisc OR1200:

https://en.wikipedia.org/wiki/OpenRISC_1200

Utdrag ur instruktionsuppsättning:

ADD rD, rA, rB

Operation: $rD = rA + rB, SR[CY], SR[OV]$

Opkod:

6	5	5	5	11
0x38	D	A	B	-

ADDI rD, rA, I

Operation: $rD = rA + exts(I), SR[CY], SR[OV]$

Opkod:

6	5	5	16
0x27	D	A	I

Lab2 : Naiv Pipelining

För att studera vad som händer kan man använda ett pipelinediagram.
Antag följande programkod:

```
ADDI    R1,R0,1      ; R1:=R0+1
ADD     R2,R1,R1      ; R2:=R1+R1
```

Det medför följande pipelinediagram:

		RR	EXE	MEM	WB	
cykel	PC	IR1	IR2	IR3	IR4	Kommentar
1	0					
2	4	ADDI				
3	8	ADD	ADDI			ADD läser här
4	C		ADD	ADDI		
5	10			ADD	ADDI	ADDI skriver här
6	14				ADD	
7	18					
8	1C					

Instruktionerna måste separeras två steg:

		RR	EXE	MEM	WB	
cykel	PC	IR1	IR2	IR3	IR4	Kommentar
1	0					
2	4	ADDI				
3	8	...	ADDI			
4	C	ADDI		
5	10	ADD	ADDI	Nu funkar det!
6	14		ADD	
7	18			ADD	...	
8	1C				ADD	

Lab2 : Naiv Pipelining

Testprogram, $DM[20..3C] = [1..8]$, $DM[40..5C] = [1..8]$

Programmet beräknar $1*1+2*2+3*3+...+8*8$

```
MOVHI R1,0      ; R1[31..16] = 0
MOVHI R2,0      ; R2[31..16] = 0
ADDIR1,R1,8     ; loopräknare = 8
ADDIR2,R2,20    ; pekare = 20
LOOP: LW R3,0(R2) ; hämta det ena talet
      LW R4,20(R2) ; hämta det andra talet
      MUL R5,R4,R3 ; multiplicera dem
      ADD R6,R6,R5 ; och ackumulera till resultat
      ADDI R2,R2,4 ; pekare++
      ADDI R1,R1,-1 ; loopräknare--
      SFNE R0,R1   ; sätt flagga=1 om loopräknare != 0
      BF LOOP     ; fortsätta?
      SW 0(R0),R6 ; spara resultatet i minnet
      TRAP 0      ; stanna processorn
```

Programmet kommer inte att göra rätt med mindre än att man löser olika beroenden.

Uppgift 3.1 =>

Lös beroenden genom att sätta in NOP-instruktioner, så få som möjligt.

Gör pipelinediagram.

Uppgift 3.2 =>

Ändra ordningsföljden på instruktioner för att lösa beroenden.

Klarar du dig utan NOP:ar i loopen?

Lab2 : Riktig Pipelining

Testprogram, $DM[20..3C] = [1..8]$, $DM[40..5C] = [1..8]$

Programmet beräknar $1*1+2*2+3*3+...+8*8$

```

MOVHI R1,0      ; R1[31..16] = 0
MOVHI R2,0      ; R2[31..16] = 0
ADDIR1,R1,8     ; loopräknare = 8
ADDIR2,R2,20    ; pekare = 20
LOOP: LW R3,0(R2) ; hämta det ena talet
      LW R4,20(R2) ; hämta det andra talet
      MUL R5,R4,R3 ; multiplicera dem
      ADD R6,R6,R5 ; och ackumulera till resultat
      ADDI R2,R2,4 ; pekare++
      ADDI R1,R1,-1 ; loopräknare--
      SFNE R0,R1   ; sätt flagga=1 om loopräknare != 0
      BF LOOP     ; fortsätta?
      SW 0(R0),R6 ; spara resultatet i minnet
      TRAP 0      ; stanna processorn

```

Vi vill ju inte behöva ändra i programmet för att det ska göra rätt.

Uppgift 4.1 =>

Gör pipelinediagram för det ursprungliga programmet. Antag att jump- och stall-logik sätter in NOP:ar på rätt ställen.

Uppgift 4.2 =>

Definiera/programmera jump- och stall-logiken, samt dataforwarding-logiken, så att det ursprungliga programmet fungerar.

Anders Nilsson

www.liu.se