Olika datorarkitekturer och pipelining

RISC/CISC

Harvard/von Neumann

Målsättning med föreläsningen

Bli bekant med

- hur processorn är uppbyggd
- de viktigaste byggstenarna i processorn
- begreppet dataväg
- idén bakom pipelining

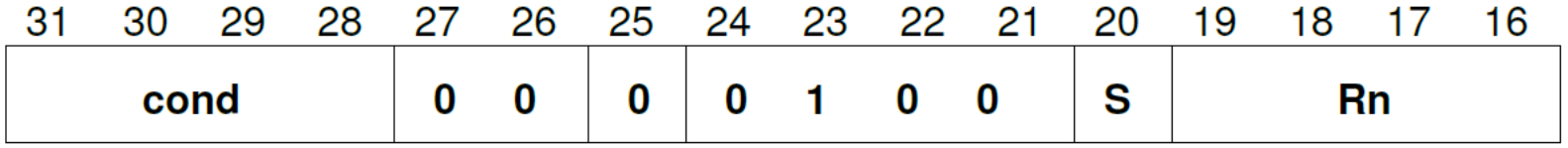
Innehåll

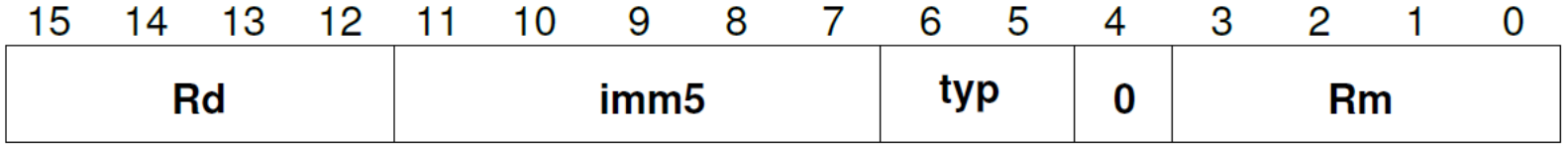
- Processorns byggstenar
- Sammankoppling av byggstenar: Datavägar
- Pipelining
- Problematik vid pipelining
- Hopplucka (delay slot)

Vad händer med instruktionerna i CPU:n?

- Instruktionerna avkodas med logik
  (jfr digitalteknik)

- Olika delar av instruktionen styr utförande av olika uppgifter





Vad gör processorn?

1. Hämta instruktion

2. Avkoda instruktion

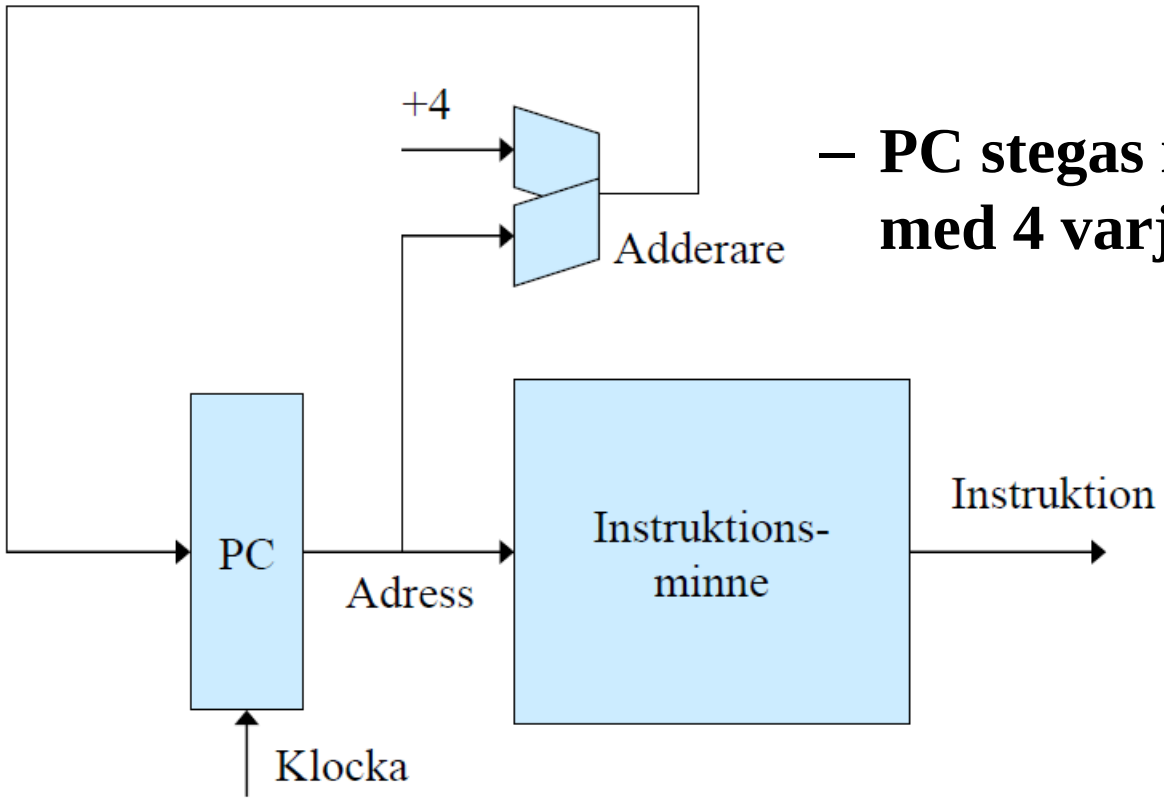
3. Hämta instruktionens operander:
   Från register eller direkt ur instruktionen (om immediate typ)

4. Utför instruktionen, t ex addera, jämföra, hoppa …

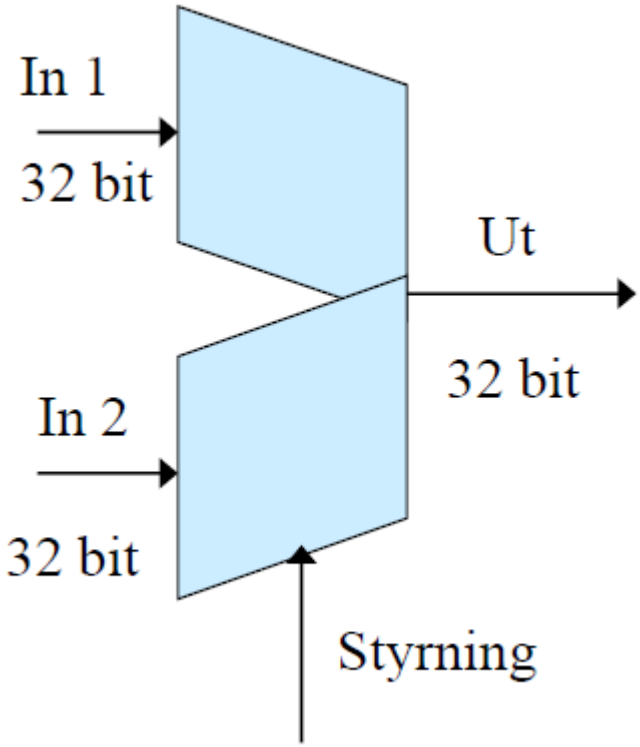
5. Skriva resultat till destinationen:
   -register (**ADD**, **SUB**, **LDR**…)
   -minne (**STR**)
   -programräknaren (hoppinstruktioner)

Instruktionshämtning

- **PC innehåller adressen till nästa instruktion**
- **PC stegas normalt upp med 4 varje gång**



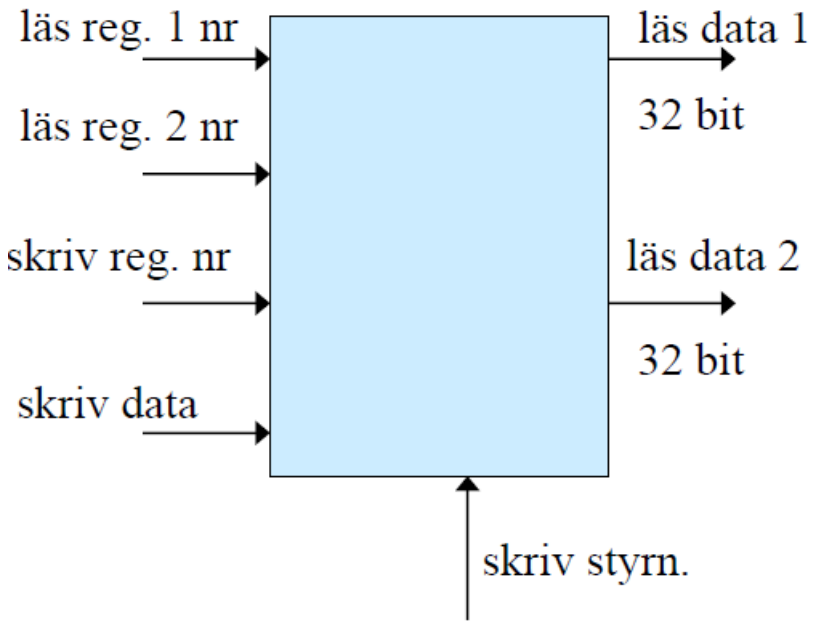
Aritmetisk-logisk enhet



Med styrsignalerna kan funktionen hos ALU:n väljas. Exempel:

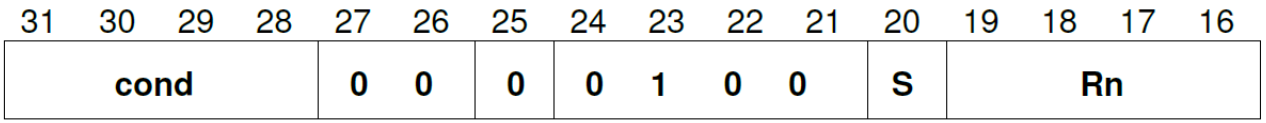
Styrsignal	Funktion
0000	AND
0010	OR
0100	ADD
…	
1100	SUB
1110	….

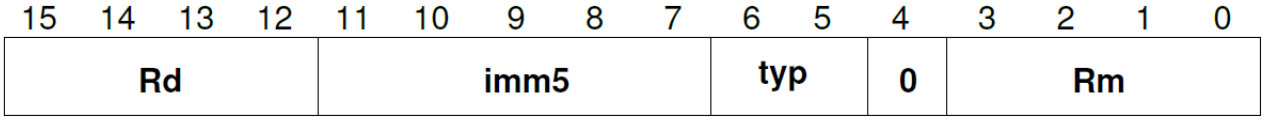
Registerbanken



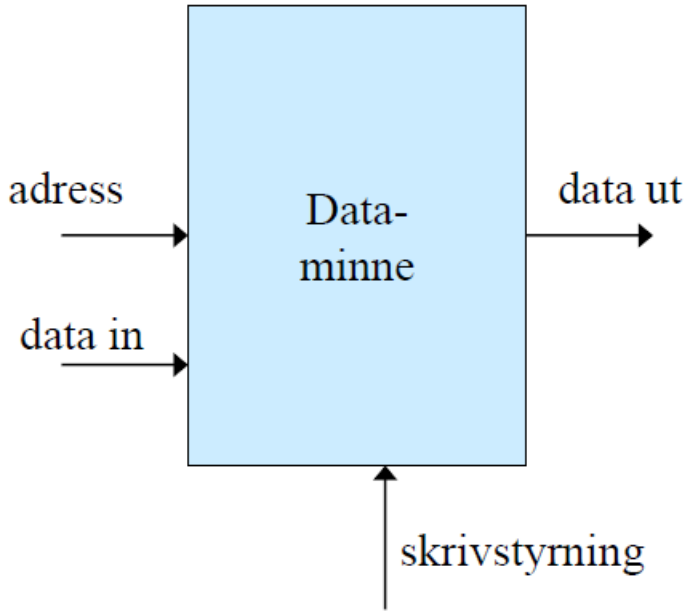
Registernummer: 4 bitar

Registren är organiserade som 32 parallella bitar (D-vippor) per register





Dataminnet

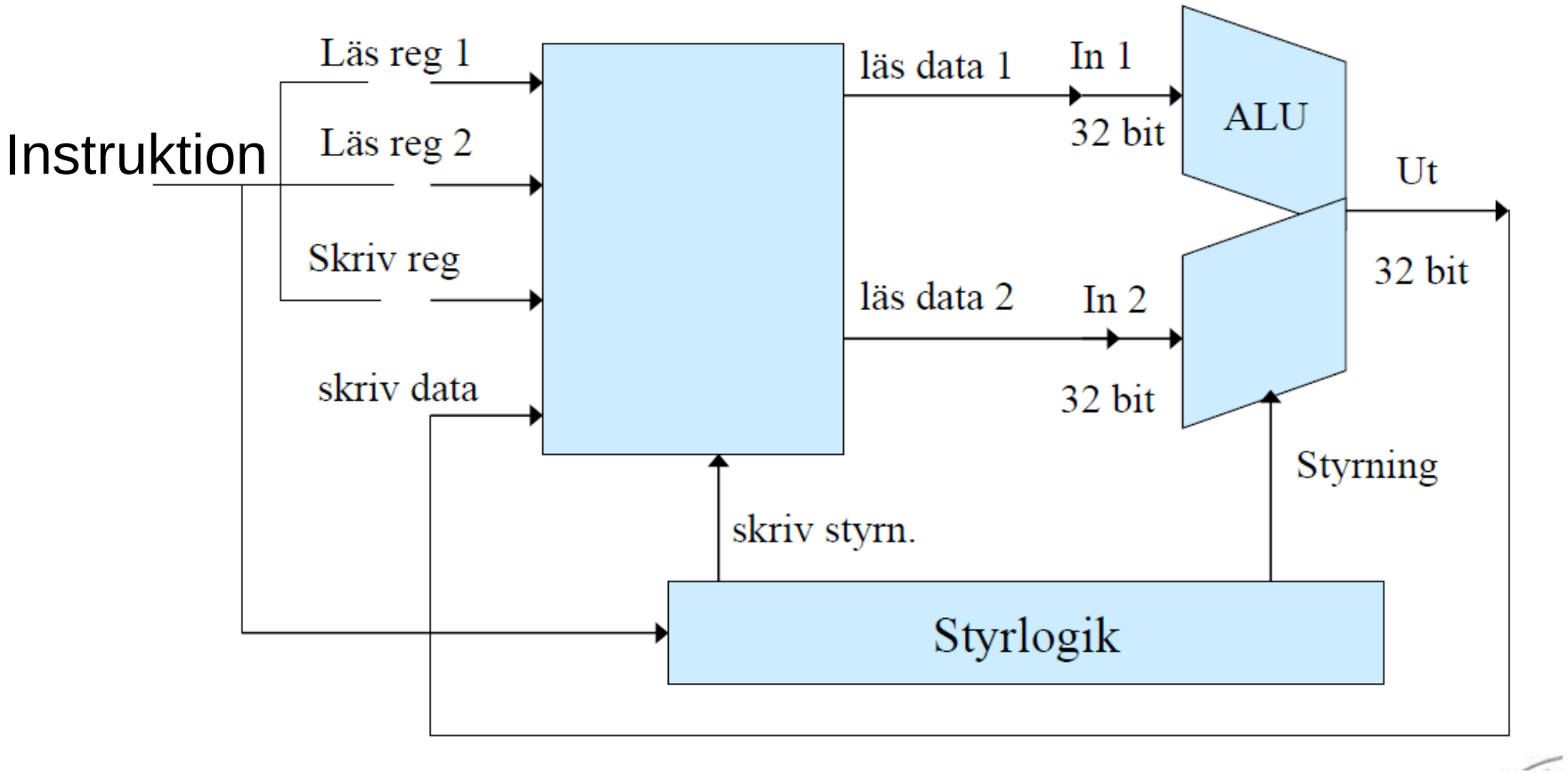


Skrivstyrningen väljer om data ska skrivas till eller läsas från minnet

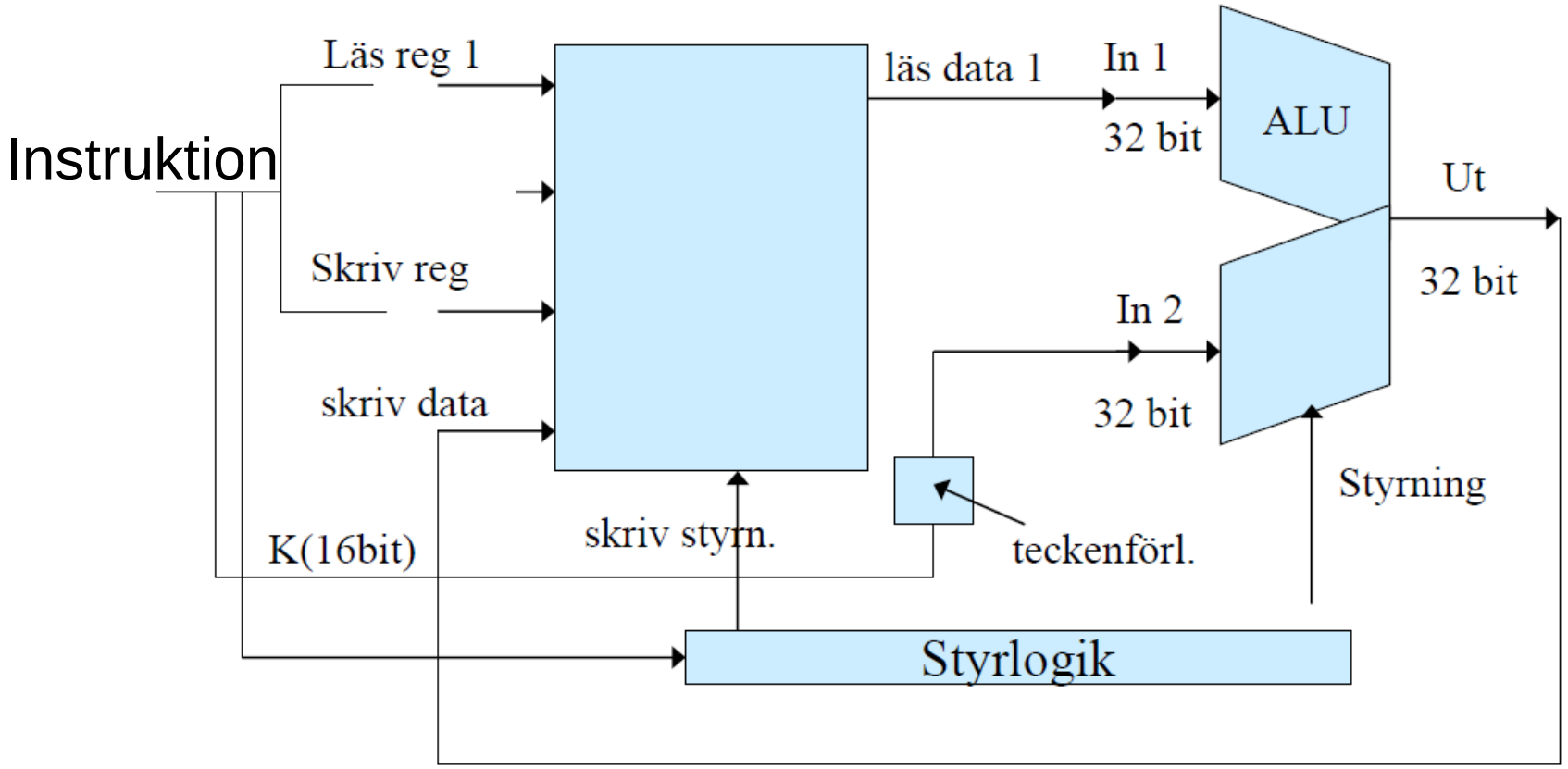
Datavägar

- Processorblocken binds ihop med så kallade datavägar, dvs möjlighet för data att transporteras mellan de olika delarna
- Vi använder några olika instruktioner för att illustrera hur datavägarna ser ut

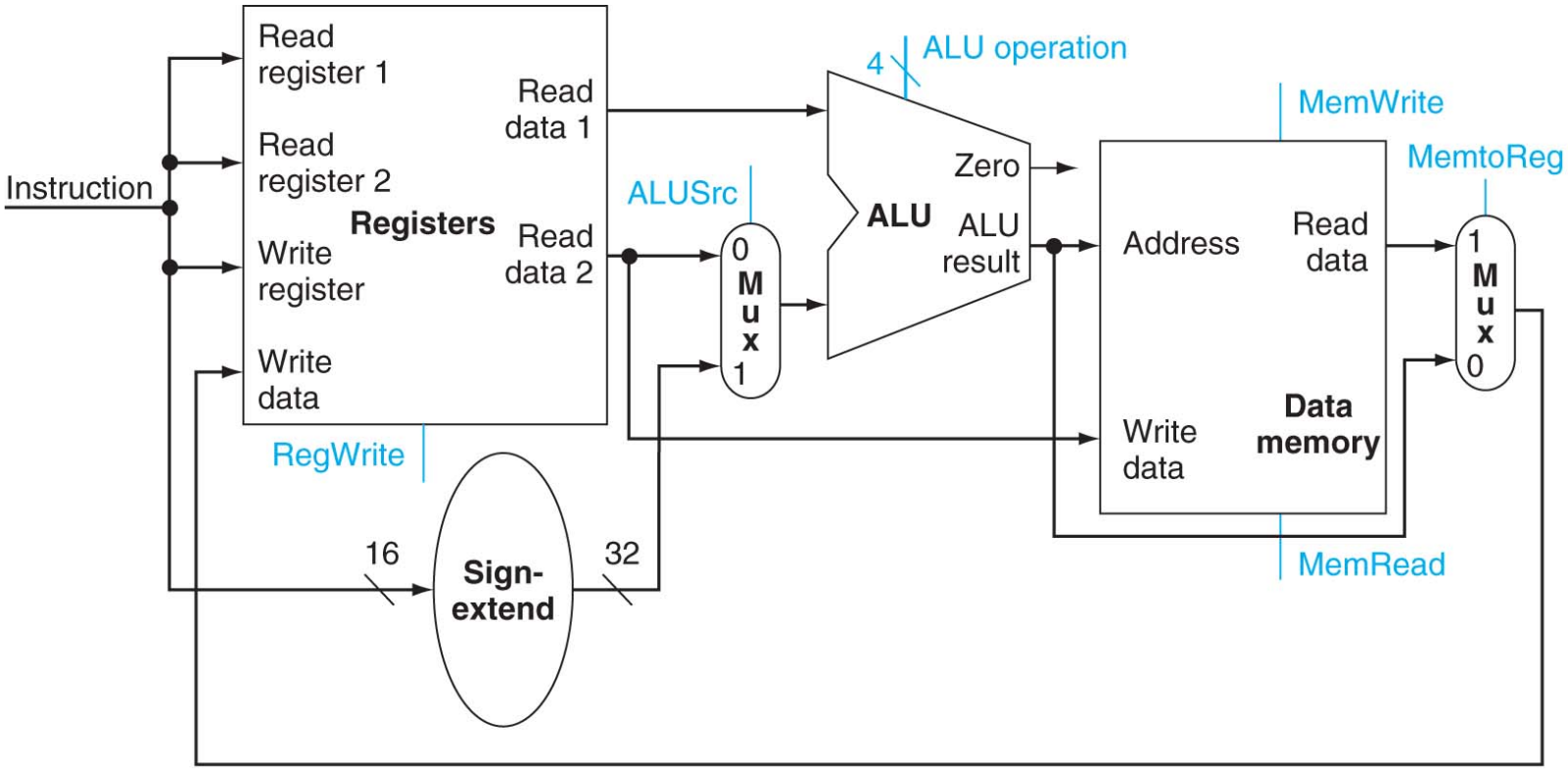
Dataväg – `ADD r0,r1,r2`



Dataväg – `ADD r0,r1,#K`



Dataväg – `LDR r0,[r1,#K]`



Datavägar -reflektioner

- Samma block används hela tiden
- Blocken används i samma ordning
- Att vissa block kopplas ihop med olika beroende på instruktion löses med multiplexrar som styrs av styrlogiken.

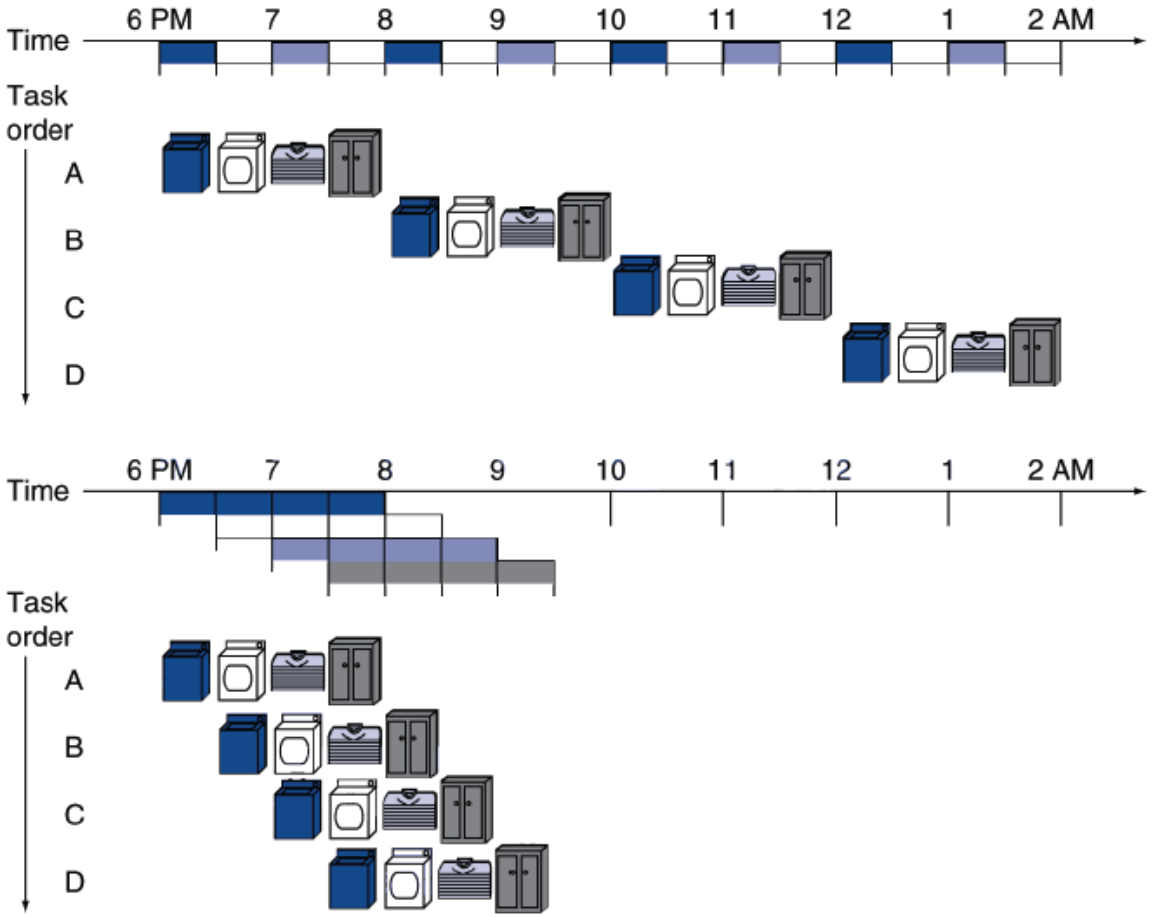
Idé till listigare implementation

- Man skulle kunna använda blocken för att köra en instruktion i taget.

- Eftersom byggblocken är klart separerade och används i samma sekvens skulle olika instruktioner kunna nyttja olika block samtidigt!

Pipeline - Analogi

- Stegen som en instruktion ska gå igenom kan jämföras med aktiviteterna i en tvättstuga.

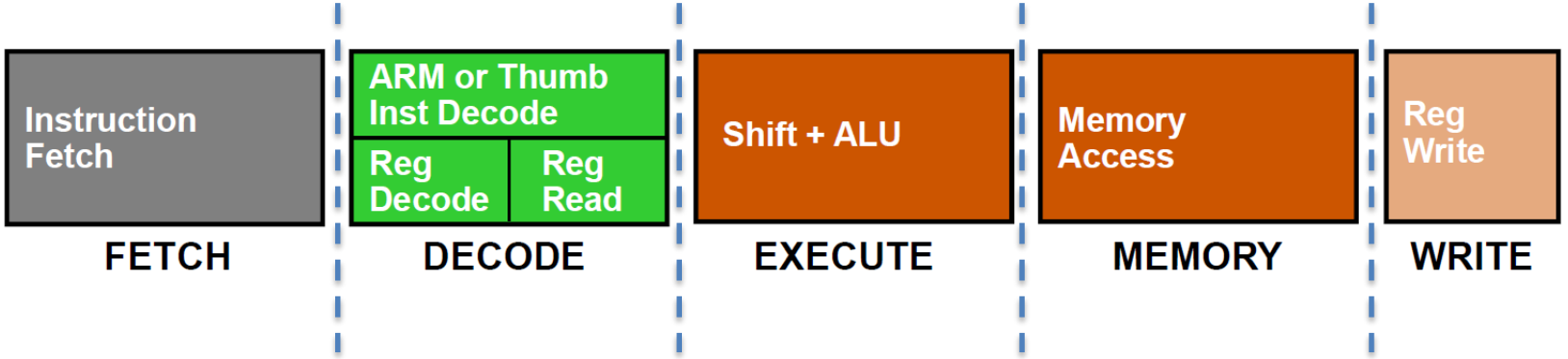


Pipelining

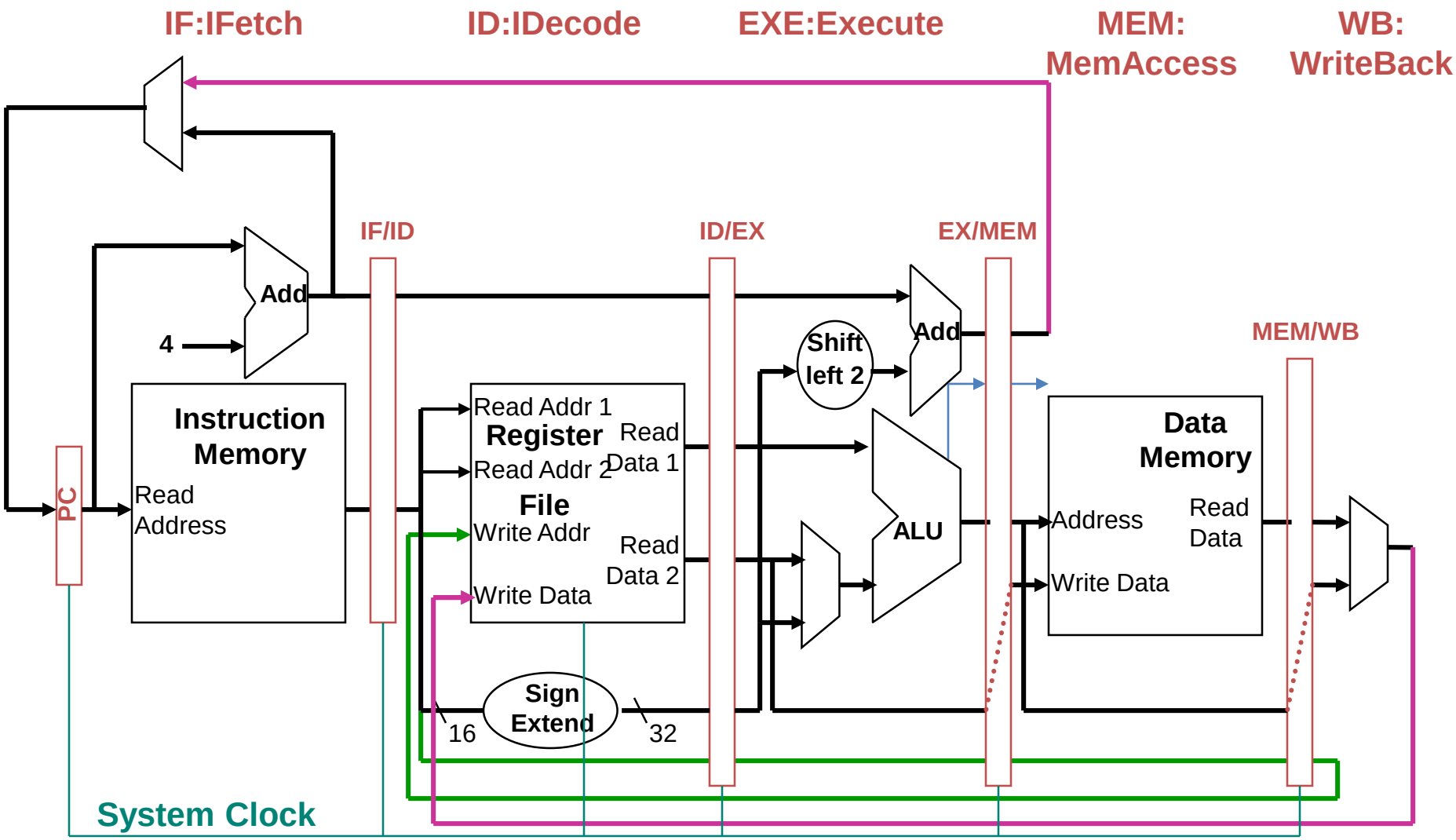
- Ett antal distinkta steg i exekveringen
  (fem i detta exempel):
  - **IF - instruction fetch** (instruktionshämtning)
  - **ID – instruction decode** (avkodning och registerläsning)
  - **EXE – execute** (beräkning av resultat eller adress i ALU)
  - **MEM – memory access** (minnesaccess)
  - **WB – write back** (skriv resultat i register)

Pipelining (forts)

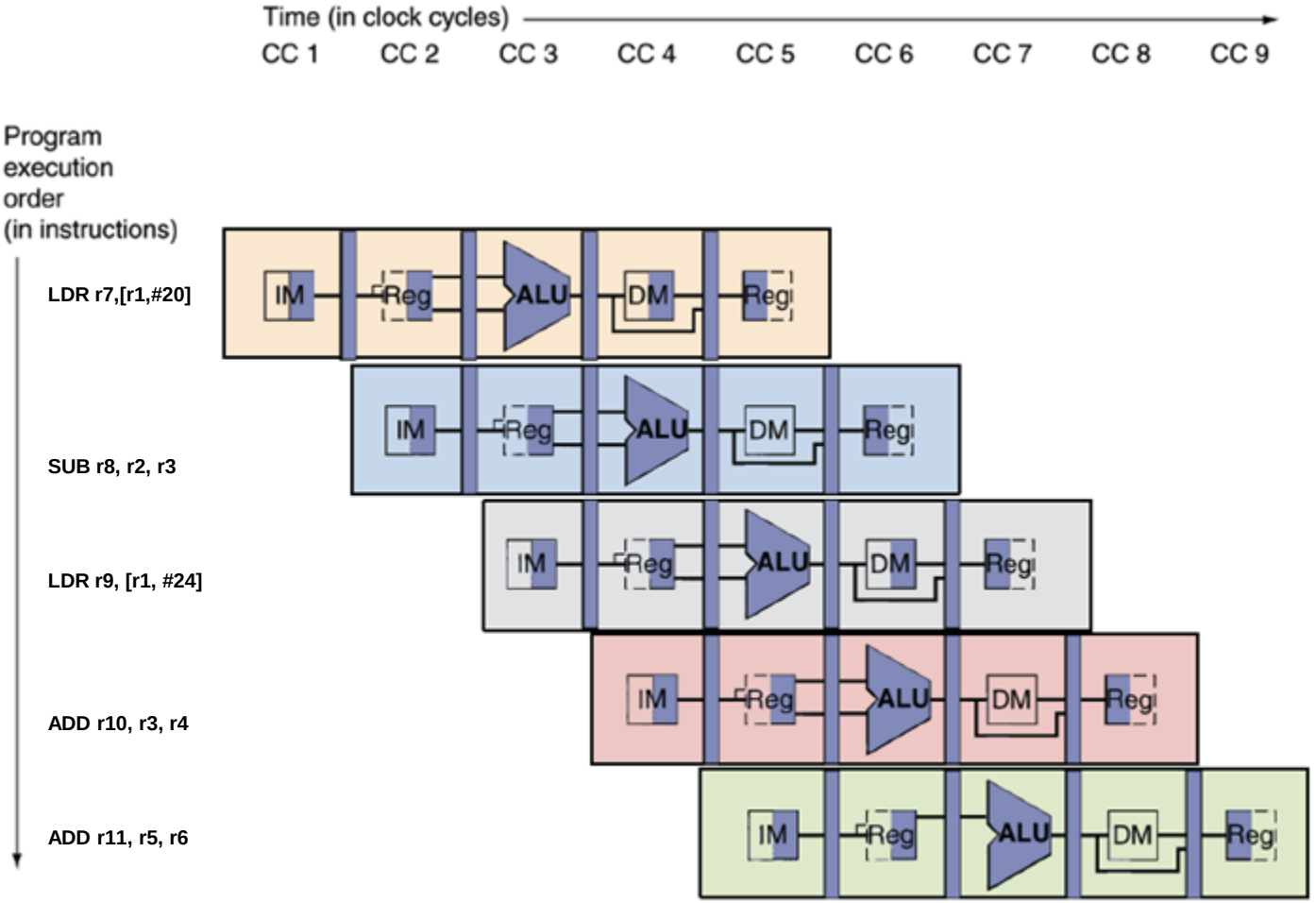
- Exekveringen delas upp i dessa delar
- IF, ID, EXE, MEM, WB
- För att separera delarna finns så kallade ”pipeline-register” mellan varje steg



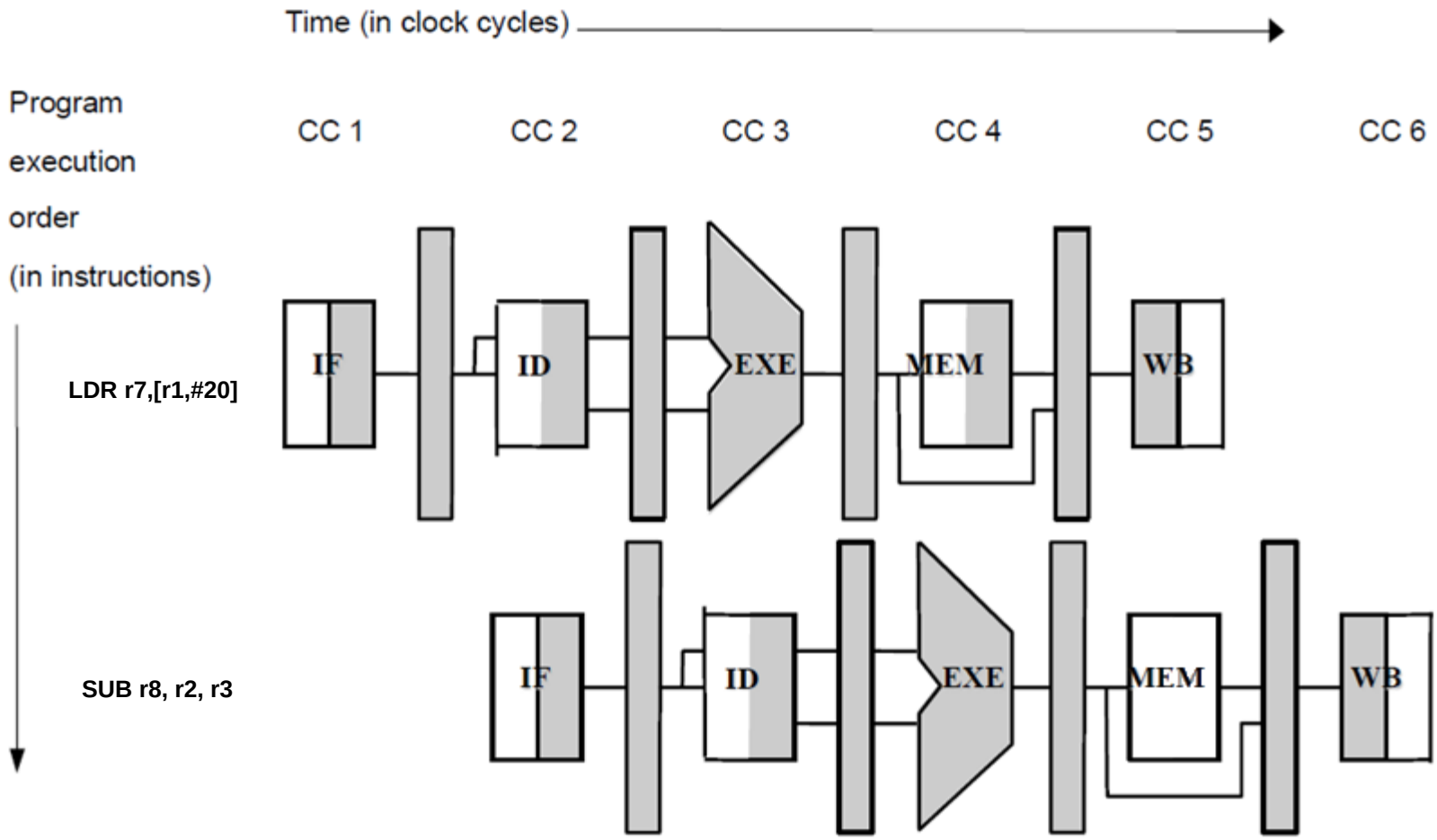
Pipeline med 5 steg



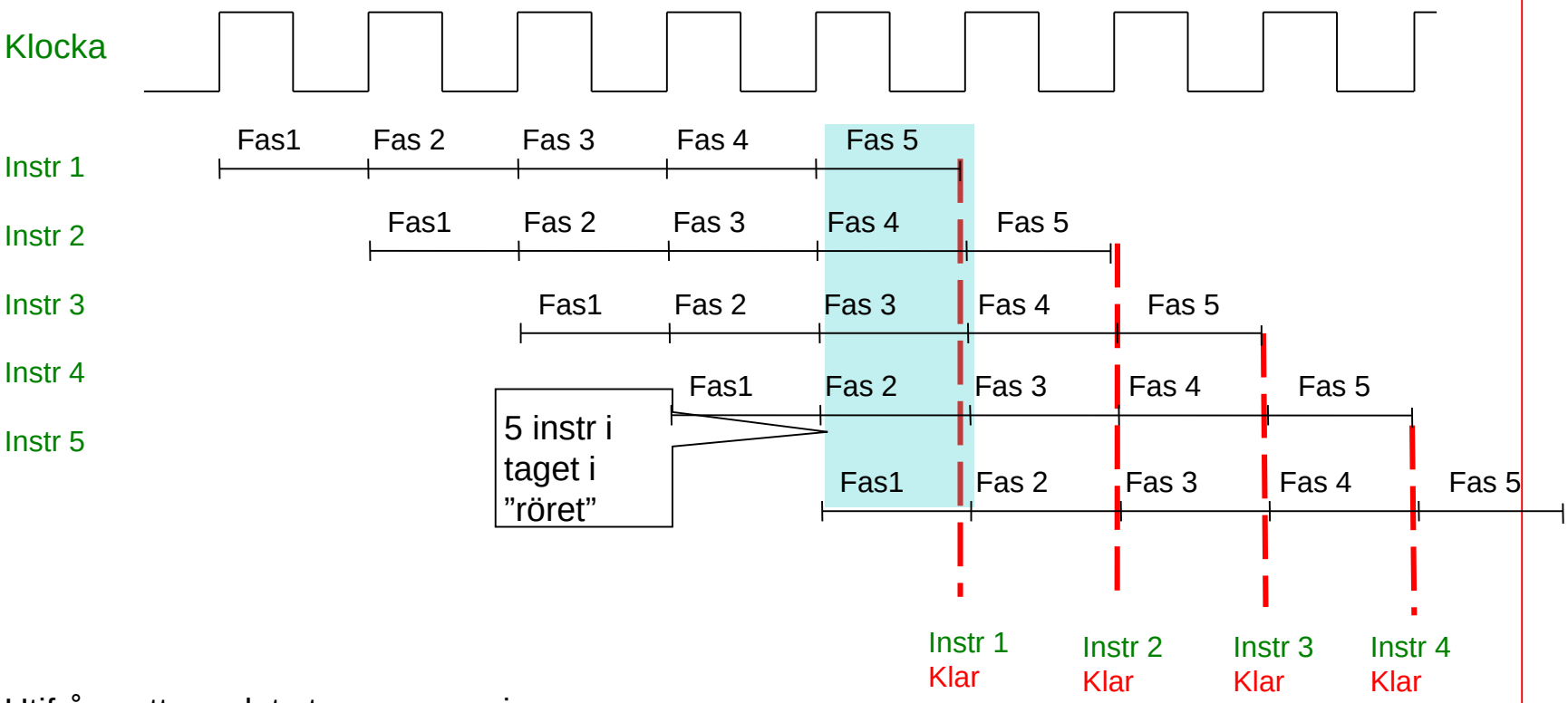
Pipeline, diagram



Pipeline


Time (in clock cycles)
Program
execution
order
(in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
LDR r7,[r1,#20]
IF
ID
EXE
MEM
WB
SUB r8, r2, r3
IF
ID
EXE
MEM
WB

Pipelining ger virtuella single-cycle instruktioner



Utifrån sett ser det ut som om varje instruktion endast tar en klockcykel.

Hoppinstruktioner

- Om vi gör ett hopp vill vi inte att instruktionerna efter hoppinstruktionen i pipen ska köras.
- Då måste pipen tömmas!
- Så kallad ”control hazard” (fara/hinder i styrstrukturen)
- Assemblern stoppar in **`NOP`**

(no operation, instruktion som inte gör något.
På ARM pseudoinstruktion för **`MOV r0,r0`**)

Hoppinstruktioner och pipeline, forts.

- Vi vet om hoppet verkligen görs när det når EXE-steget om det är villkorligt
- För att inte förlora tid kör vi färdigt nästa instruktion också, den är ju redan i ID-steget
- Därför finns **HOPPLUCKAN** (Delay slot) som kan användas för en annan instruktion.

Lite historik och allmänbildning

- I datorteknikens barndom var skillnaderna mellan arkitekturer större och gick lättare att dela in i kategorier.

- Begrepp som finns i datorteknikens värld och som hör till allmänbildningen att kunna följer här:

RISC/CISC

Reduced Instruction Set/Complex Instruction Set

En RISC-processor kännetecknas av

- enkel arkitektur
- många register
- hårdvaruimplementerad styrenhet *(sekvensnät)*
- ”single cycle”-instruktioner (virtuellt genom pipelining)

Kännetecken RISC (forts)

- aritmetiska instruktioner sker *register-till-register*
- minnesaccess endast via instruktionerna *load* och *store*
  (kallas därför ibland även ”load-store”-arkitektur)
- fundamentala operationer (enkla och grundläggande)
- instruktionsformat med konstant längd
- liten instruktionsrepertoar *(ca 60 i ARM)*
- instruktioner använder få adresseringsmöjligheter:
  - Typiskt: register, direkt, register indirekt, displacement

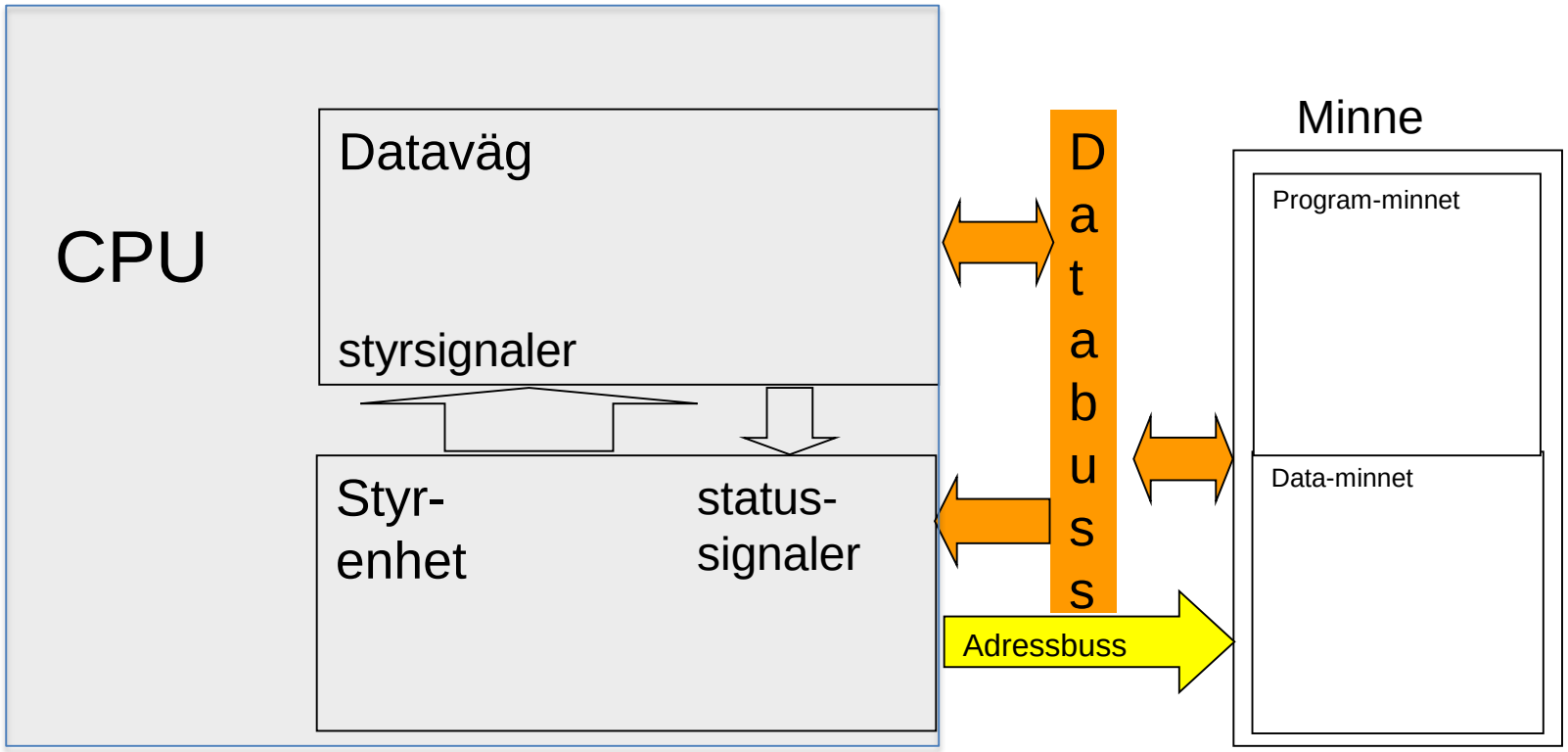
RISC/CISC forts

En CISC-processor kännetecknas av

- komplex arkitektur
- komplexa operationer
- varierande längd på instruktionsformatet *(Intel 1 – 14 byte)*
- stor instruktionsrepertoar *(Intel 32-bit ca 1100)*
- styrenhet med mikroprogram
- en instruktion tar flera klockcykler

Arkitekturtyp - von Neumann

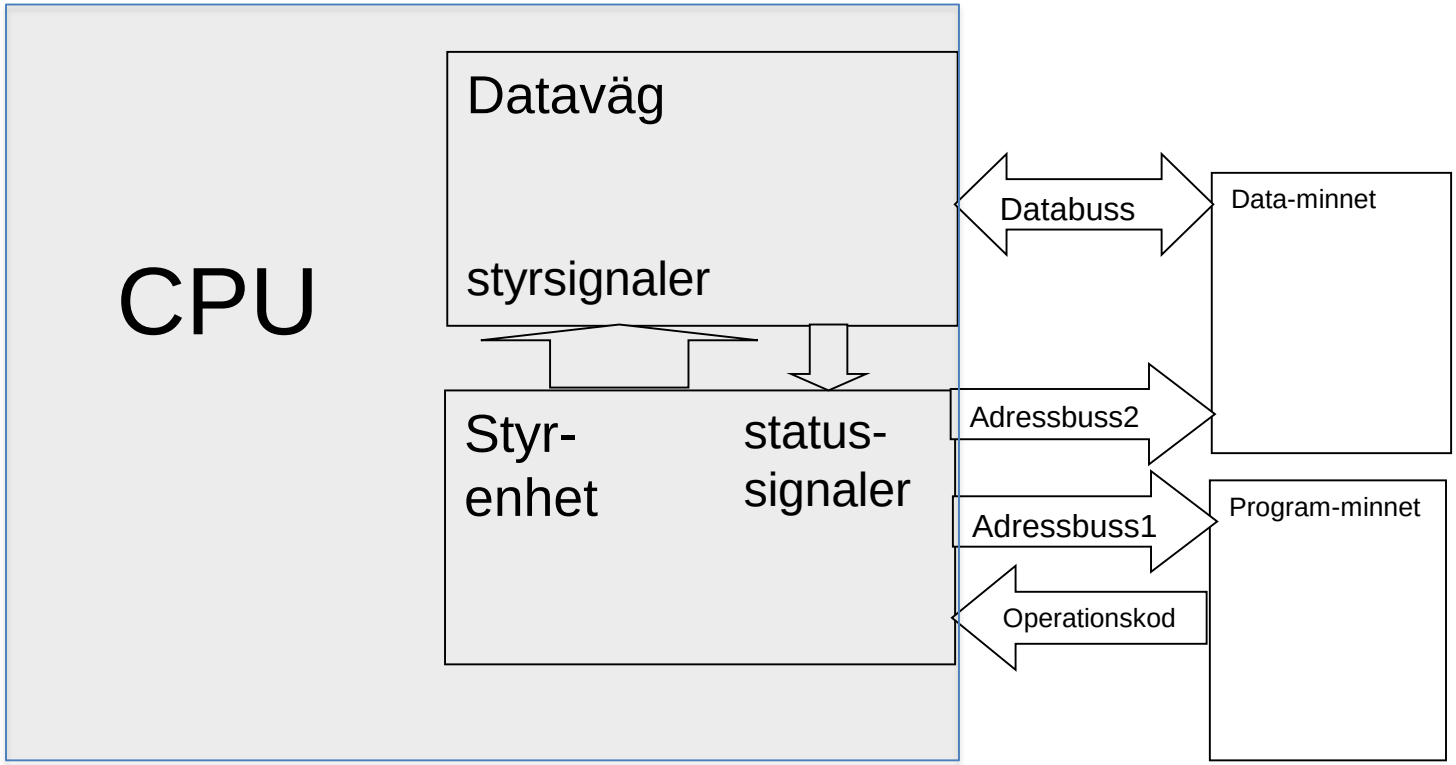
Program och data ligger i samma minne



Fungerar dåligt med pipelining eftersom bussarna mot minnet blir en flaskhals

Arkitekturtyp - Harvard

Data och programkod ligger i olika minnen och adresseras via olika bussar.



ARM i sammanhanget

- ARM är en utpräglad RISC-processor.

- Man kan säga att den är en mix av von Neumann och Harvardarkitektur. Utifrån sett är den von Neuman, men cache-minnet internt är uppdelat i en I-cache och en D-cache, så internt kan den arbeta som en Harvard-processor.

I-cache = instruktionscache

D-cache = datacache

RISC vs CISC

- Svårt att ge entydigt svar vad som är bäst
- Flera prestandajämförelser visar att RISC exekverar snabbare än CISC, men svårt att göra en rättvisande jämförelse
- RISC behöver fler instruktioner för att utföra samma jobb vilket oftast kräver mer minne
- Moderna processorer använder sig av både lite RISC och lite CISC

Andra typer av ”hazards” vid pipelining

- Strukturella ”hazards” (faror/hinder) uppstår om man har en gemensam instruktions och datacache
  - Då måste uppehåll (*stall*) göras när en instruktion ska accessa minne (man kan inte läsa instruktion och läsa/skriva data samtidigt)

- Data ”hazards” uppstår om en instruktion behöver resultat från en föregående instruktion, som ännu inte är klart (databeroende).
  - Kan ibland lösas med så kallad ”data forwarding”, som gör ett resultat tillgängligt vid ALUns ingångar innan det skrivits till register
  - Assemblern försöker undvika problemet genom att möblera om instruktionerna (*instruction reordering*)