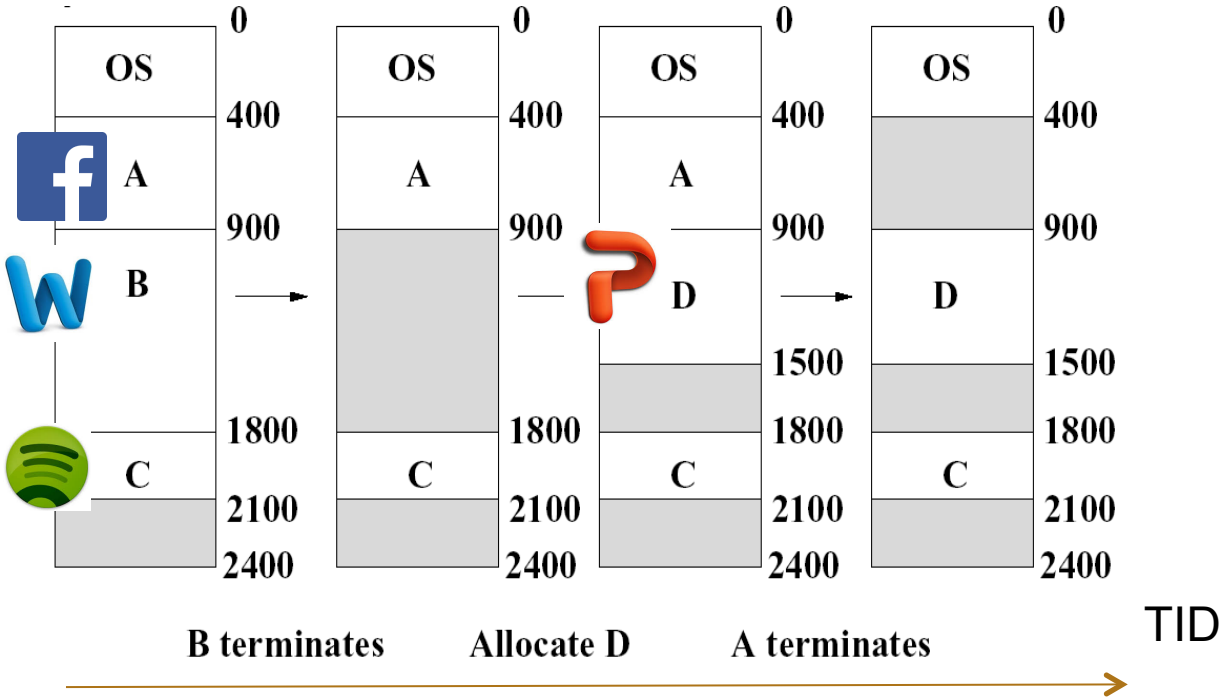
LUNDS UNIVERSITET

Datorteknik

ERIK LARSSON

Närliggande allokering

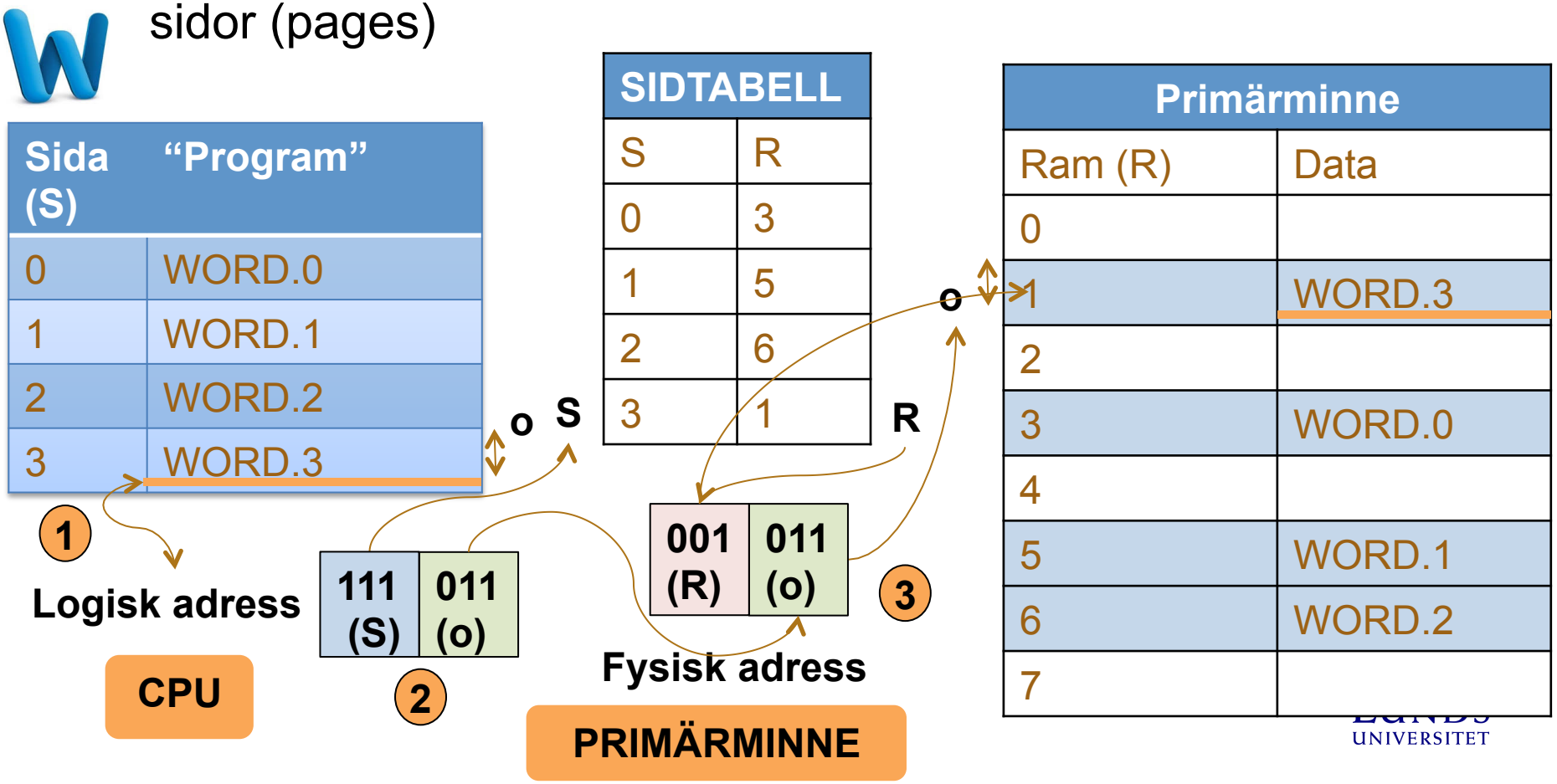


- Problem: Minnet blir fragmenterat

LUNDS UNIVERSITET

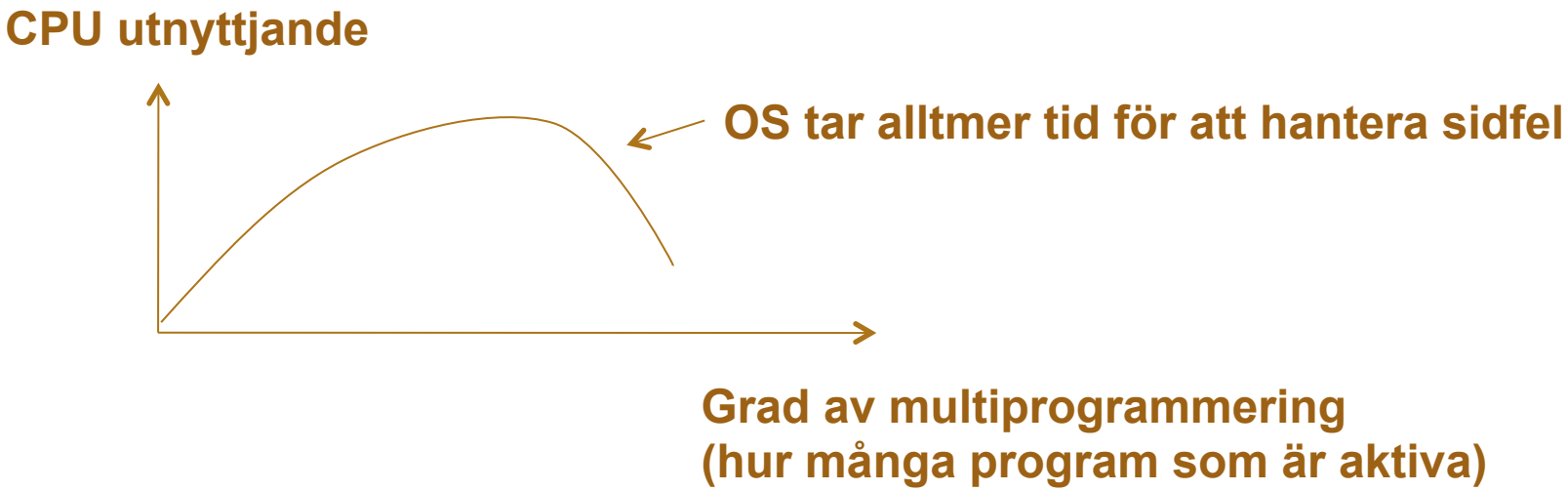
Paging

- Dela upp primärminnet i ramar (frames) och program i sidor (pages)



Demand paging

- Ladda endast de pages som behövs till primärminnet



Virtuellt minne

- Använd primärminne som “cache” för sekundärminne (hårddisk)
  - Hanteras med hårdvara och operativsystem
- Program delar primärminnet
  - Varje program får sin virtuella adressrymd
  - Skyddas från andra program
- CPU och OS översätter virtuella adresser till fysiska adresser
  - Ett “block” kallas för sida (page)
  - “Miss” kallas för sidfel (page fault)

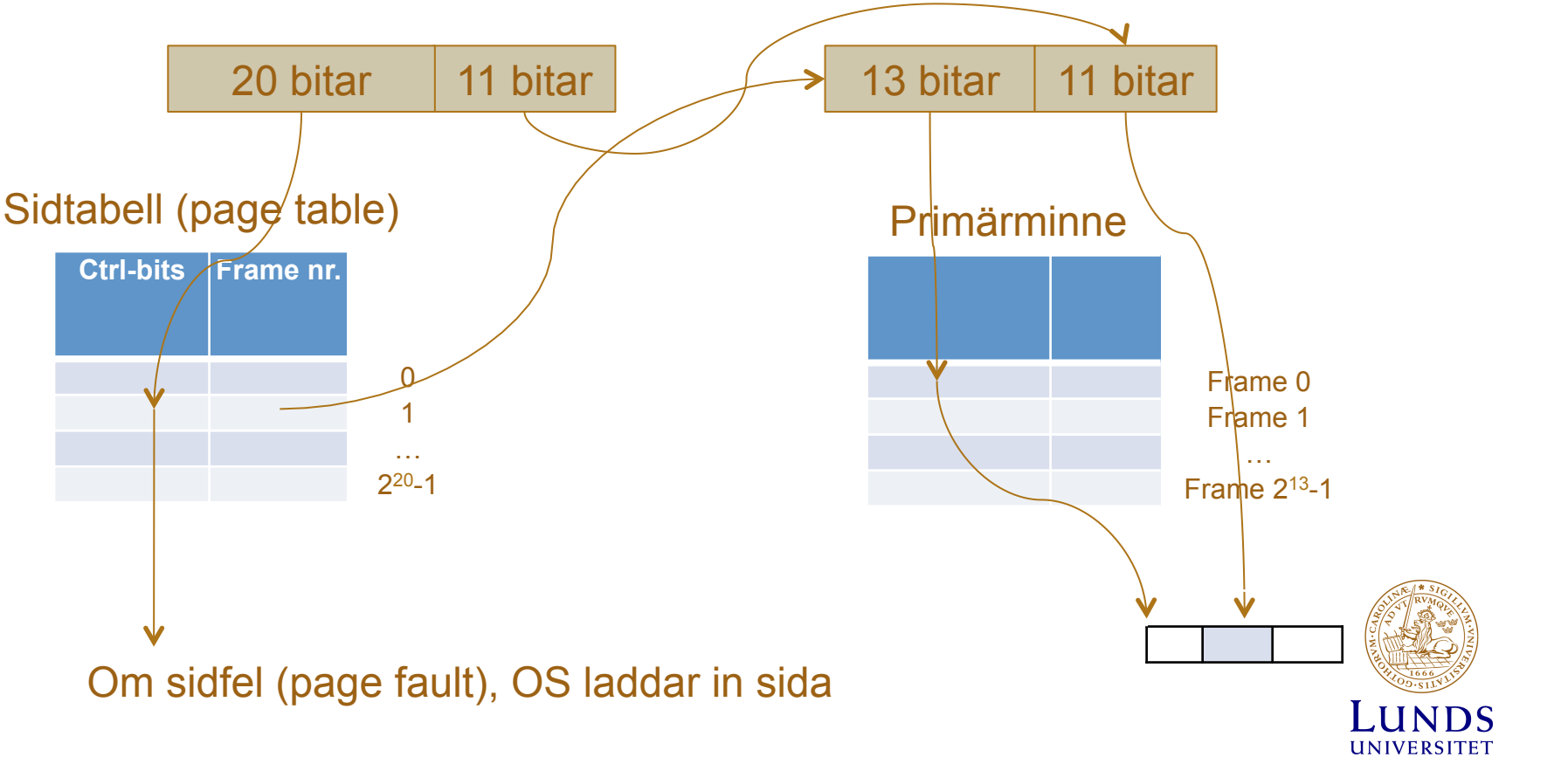
LUNDS UNIVERSITET

Virtuellt minne

Memory Management Unit (MMU)

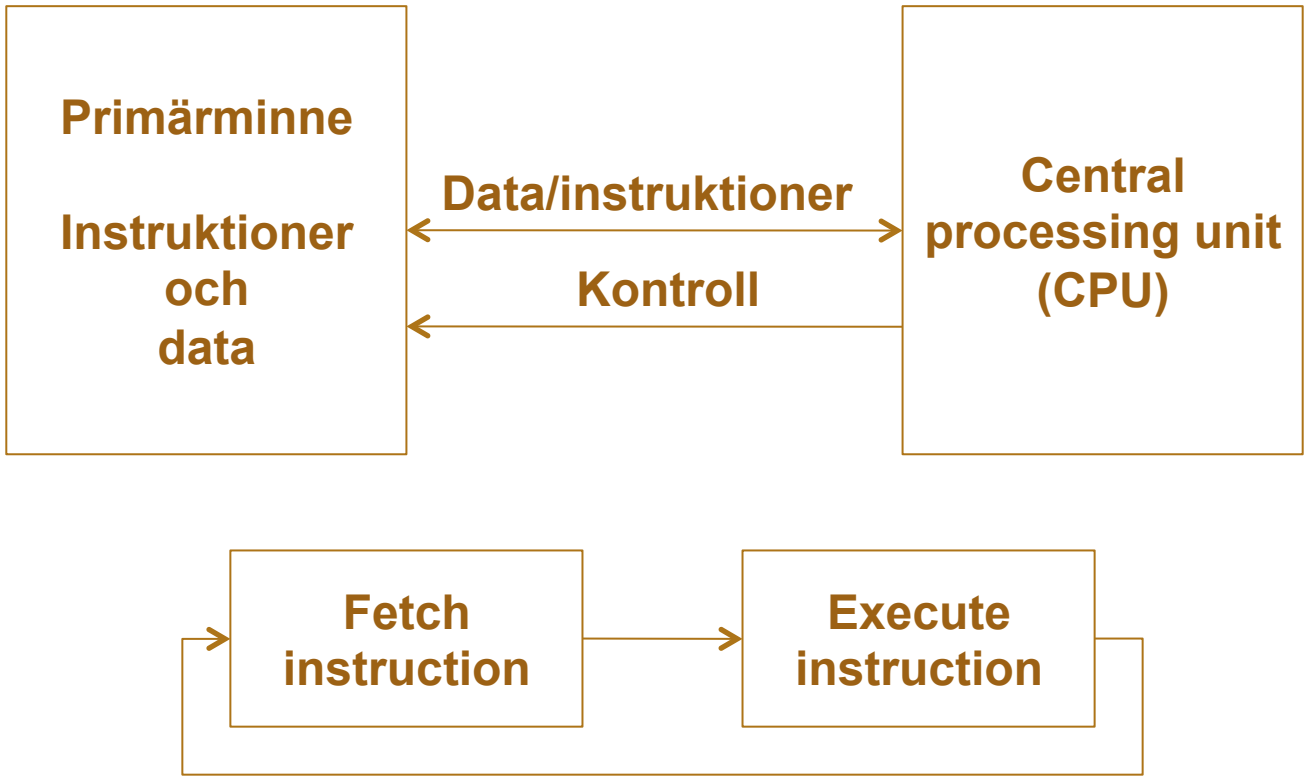
- Virtuell adress: 31 bitar

Fysisk adress: 24 bitar



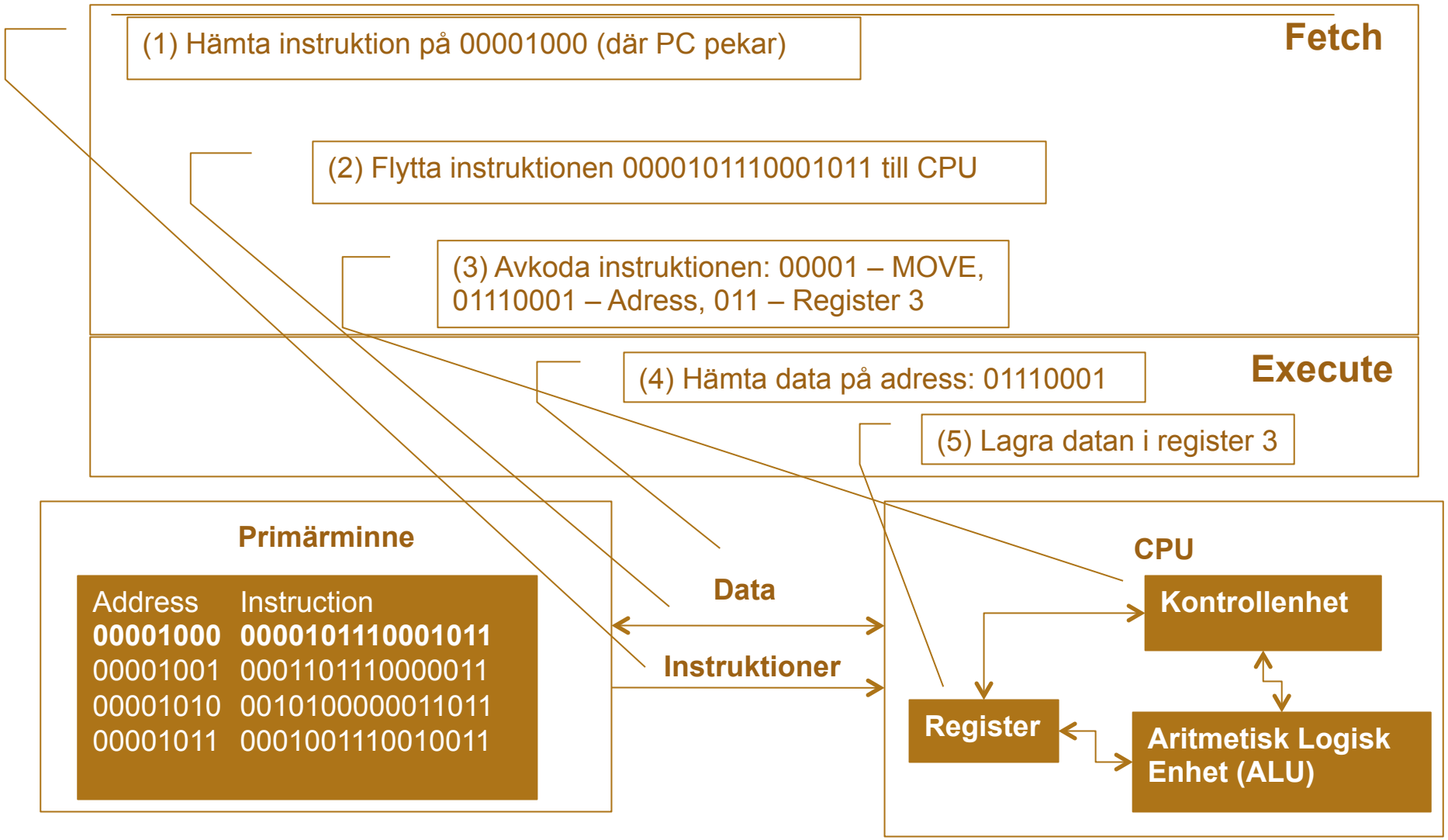
LUNDS UNIVERSITET

Dator



LUNDS UNIVERSITET

Programexekvering



Fetch-Execute

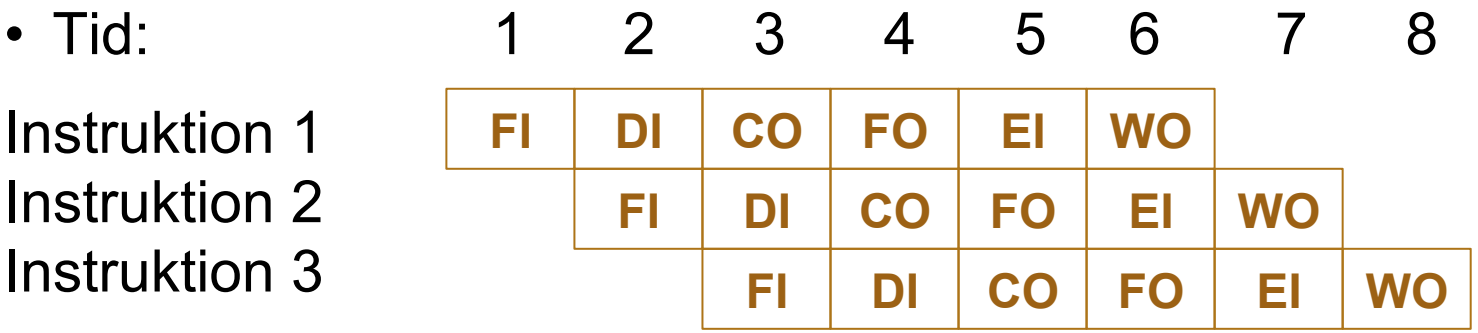
- Utan pipelining:
  - Tid: 1 2 3 4 5 6

- Med pipelining:
  - Tid: 1 2 3 4

LUNDS UNIVERSITET

Pipelining

- 6-stegs pipeline:
  - Fetch instruction (FI)
  - Decode instruction (DI)
  - Calculate operand address (CO)
  - Fetch operand (FO)
  - Execute instruction (EI)
  - Write operand (WO)
- Tid:



LUNDS UNIVERSITET

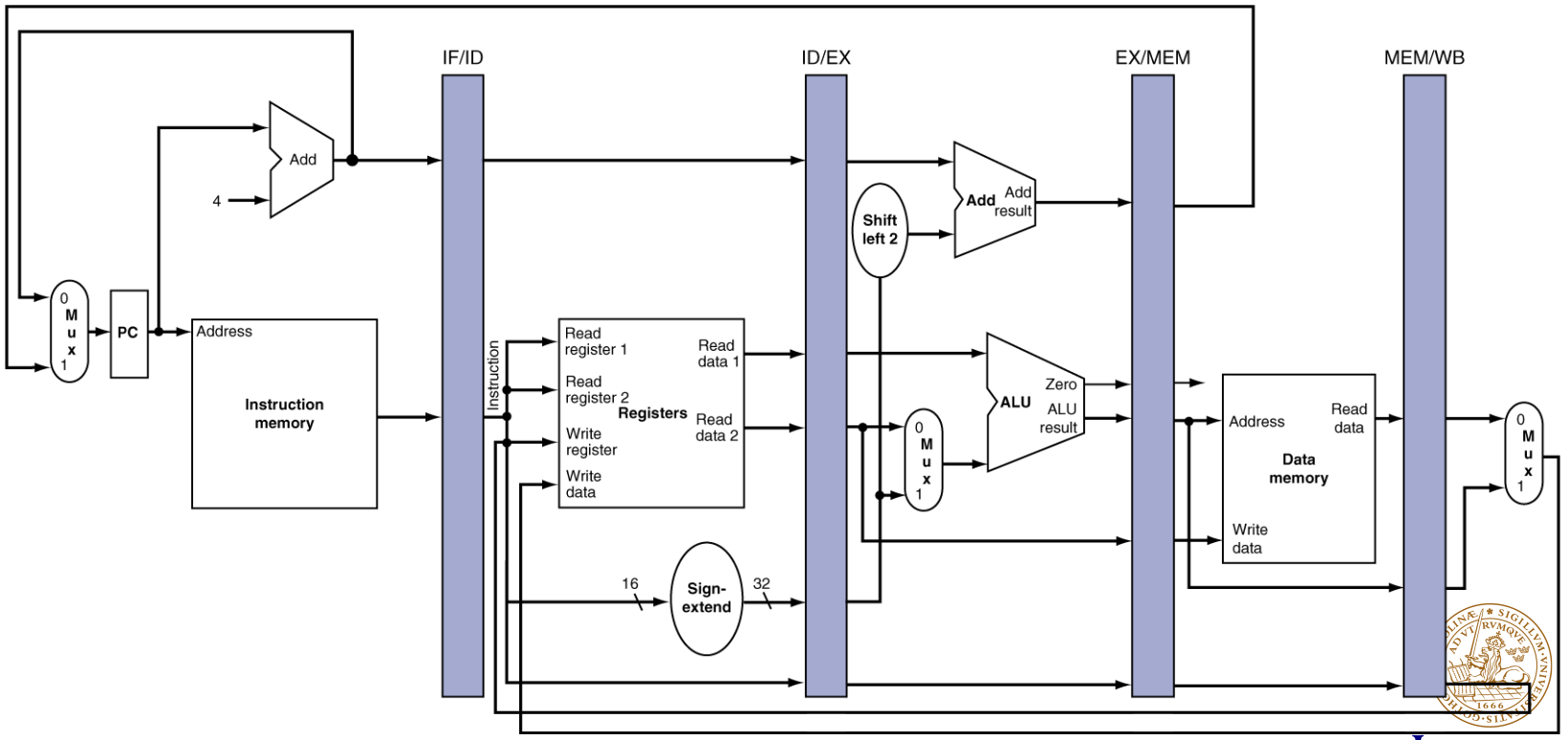
Pipelining

- Fler pipeline steg ger bättre prestanda, men
  - Mer overhead att hålla koll på pipeline
  - Komplexare processor
  - Svårt att hålla pipelinen full pga pipeline hazards
- 80486 och Pentium:
  - Fem-stegs pipeline för heltal (integer) instruktioner
  - Åtta-stegs pipeline för flyttals (float) instruktioner
- PowerPC:
  - Fyra-stegs pipeline för heltal (integer)
  - Sex-stegs pipeline för flyttal (float)

LUNDS UNIVERSITET

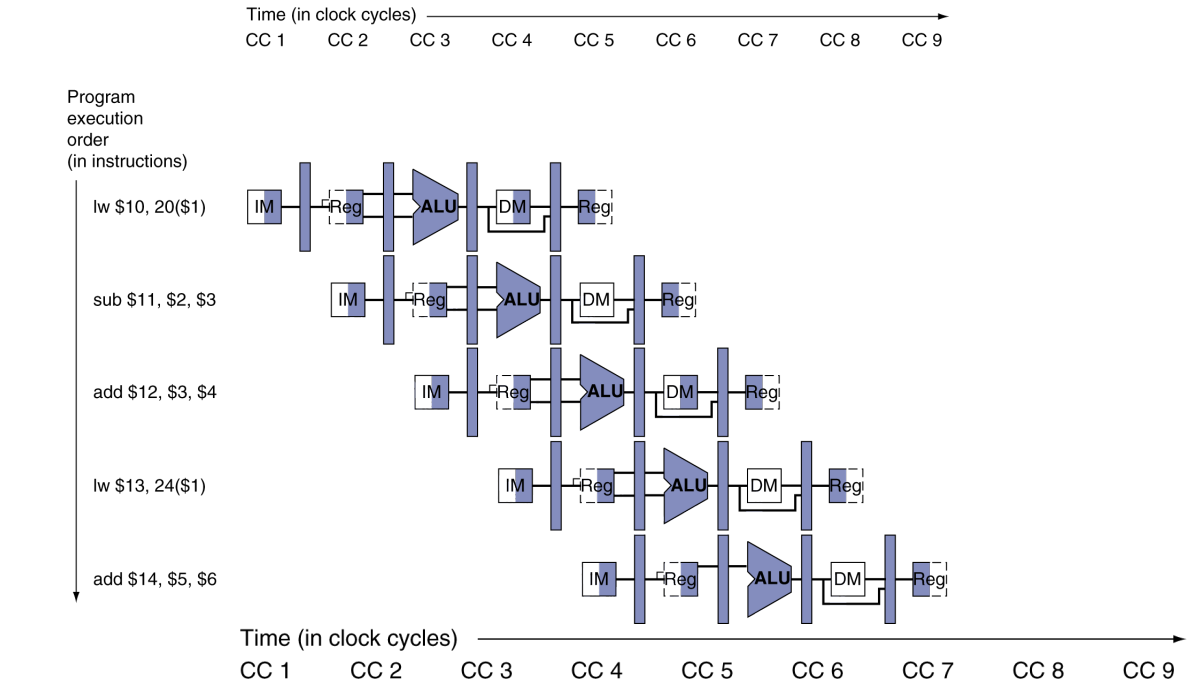
Pipeline diagram (vid en viss tidpunkt)

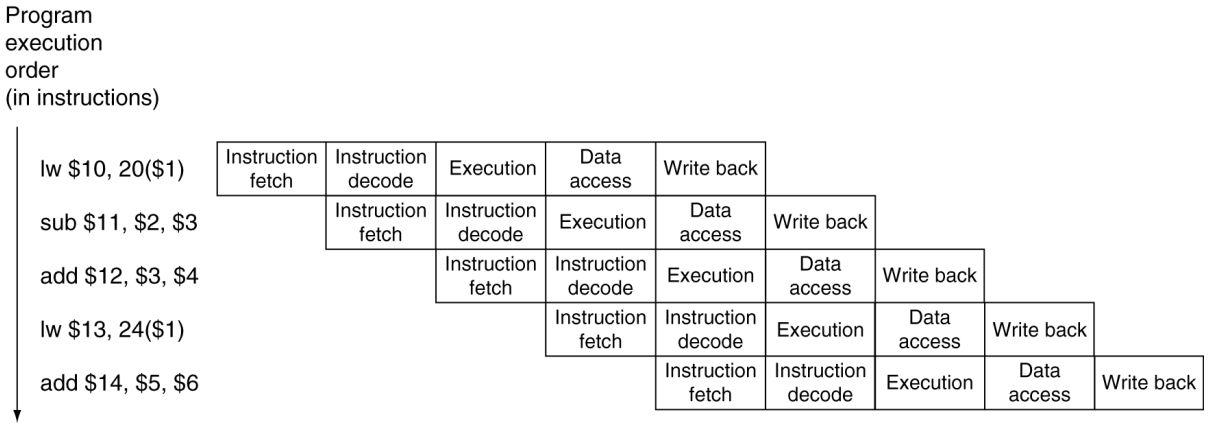
add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



LUNDS UNIVERSITET

Pipeline diagram

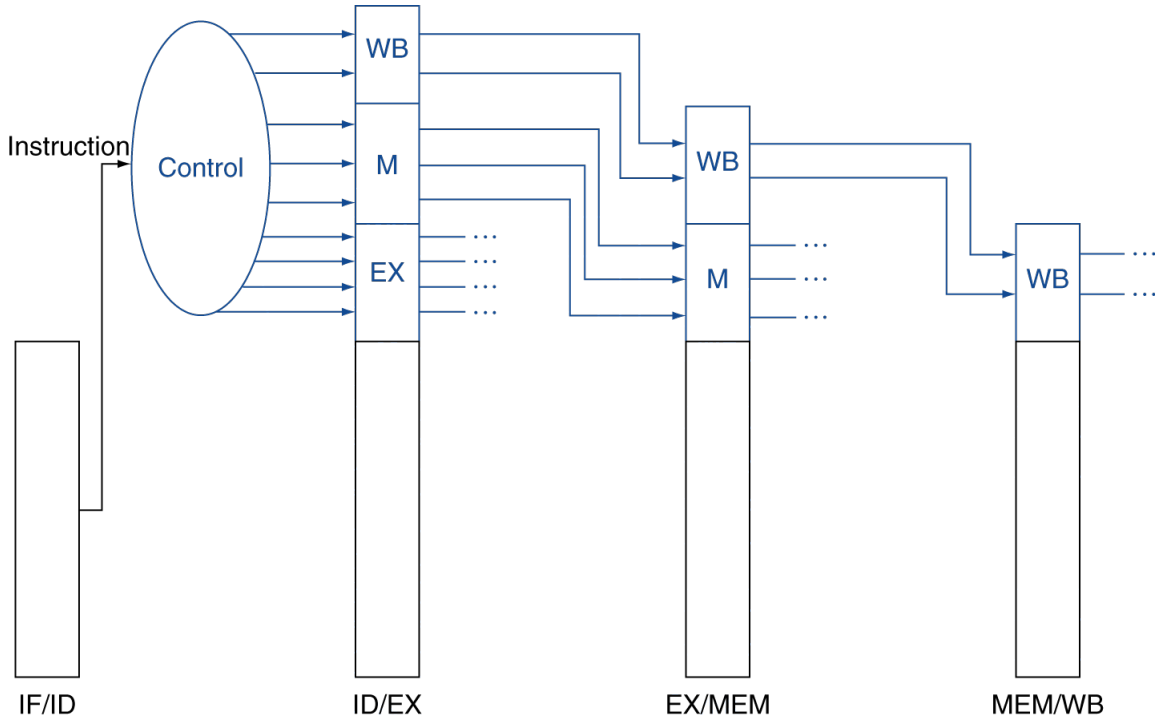




LUNDS UNIVERSITET

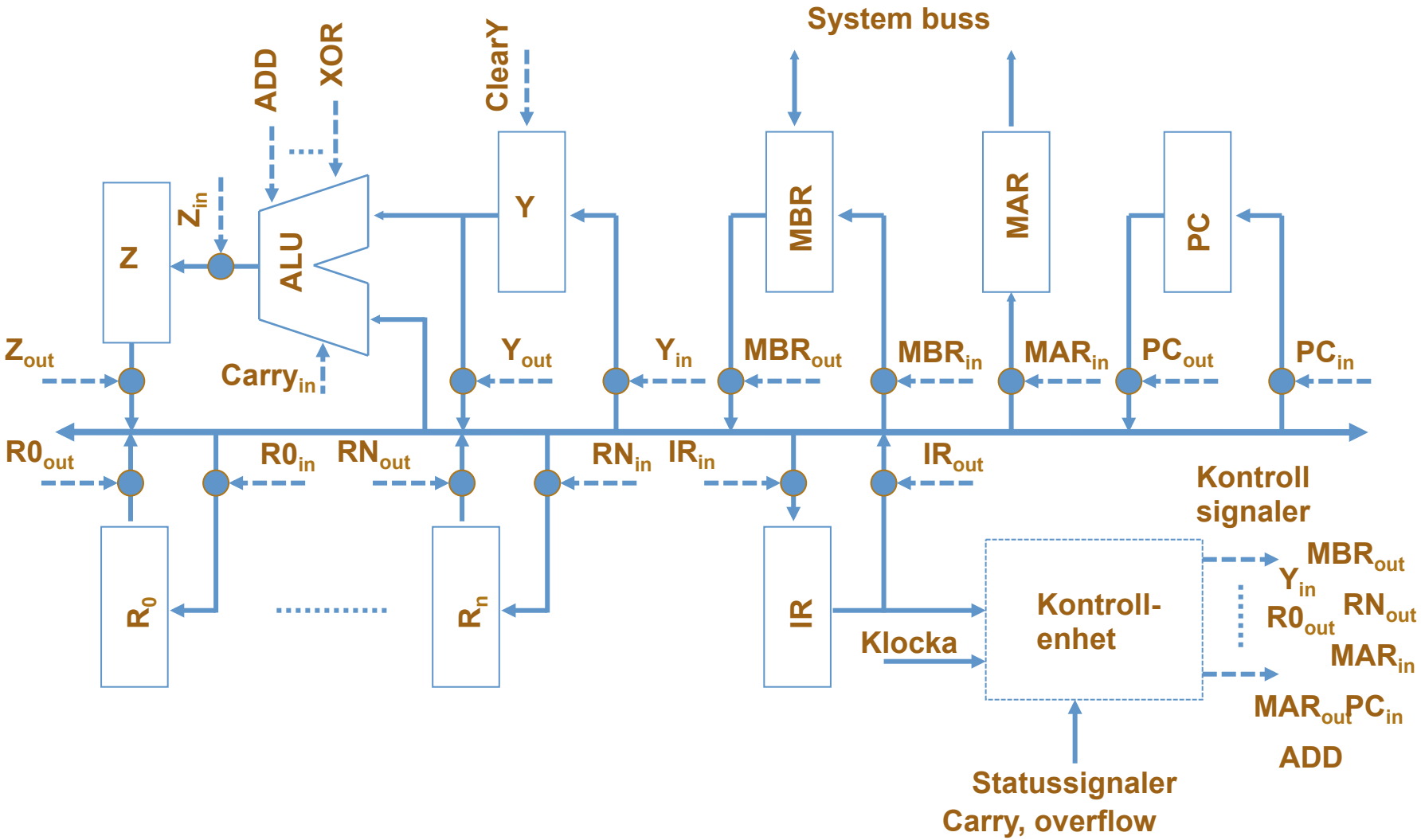
Pipeline kontroll

- Kontrollsignaler fås från instruktion



LUNDS UNIVERSITET

Kontrollenhet utan pipeline (FÖ1)



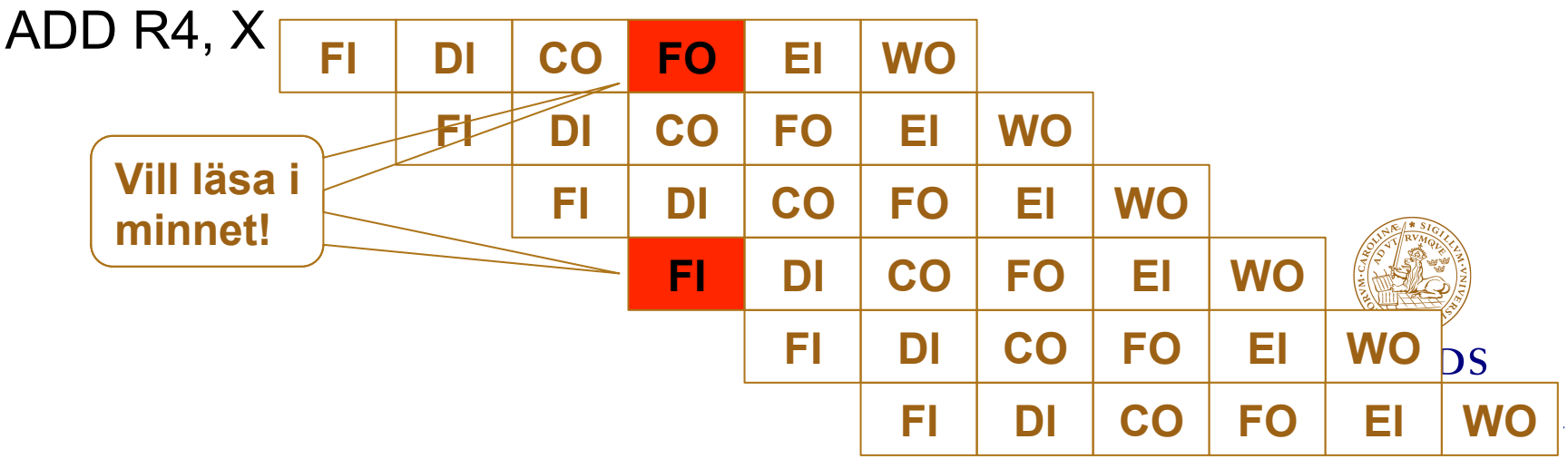
Kontrollenhet utan pipeline (FÖ1)

- Instruktion:
  - ADD R1, R3 // R1 <- R1 + R3
- Kontrollsteg:
  1. $\underline{PC_{out}, MAR_{in}, Read, Clear\ Y, Carry\text{-}in, Add, Z_{in}}$
  2. Z_{out}, PC_{in}
  3. MBR_{out}, IR_{in}
  4. $R1_{out}, Y_{in}$
  5. $R3_{out}, Add, Z_{in}$
  6. $Z_{out}, R1_{in}, End$

LUNDS UNIVERSITET

Strukturella hazards

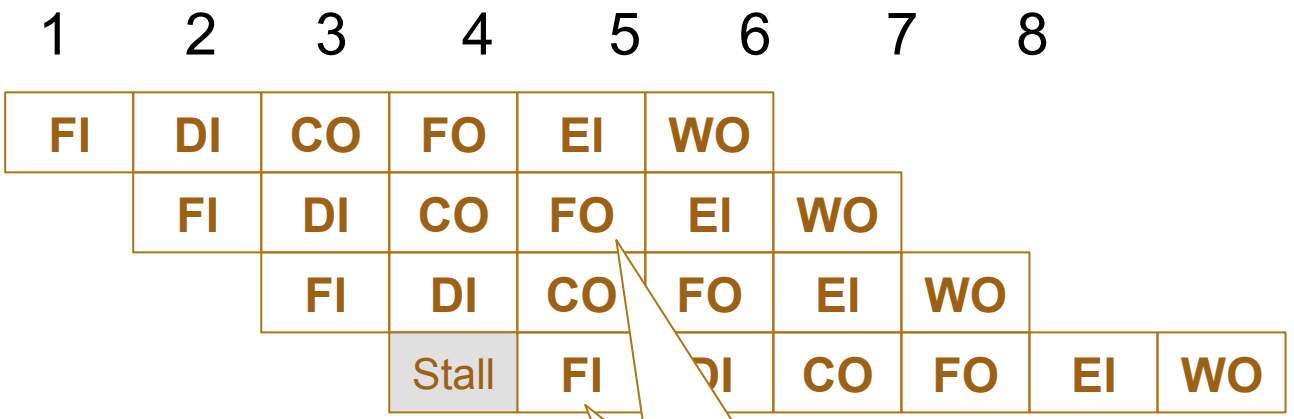
- Resurs (minne, funktionell enhet) behövs av fler än en instruktion samtidigt
- Instruktionen ADD R4, X – R4 <- R4 + X
  - I steget FO hämtas operanden X från minnet. Minnen accepterar en läsning i taget
- Tid: 1 2 3 4 5 6 7 8



Strukturella hazards

- Tid:

ADD R4, X
ADD R1, R2

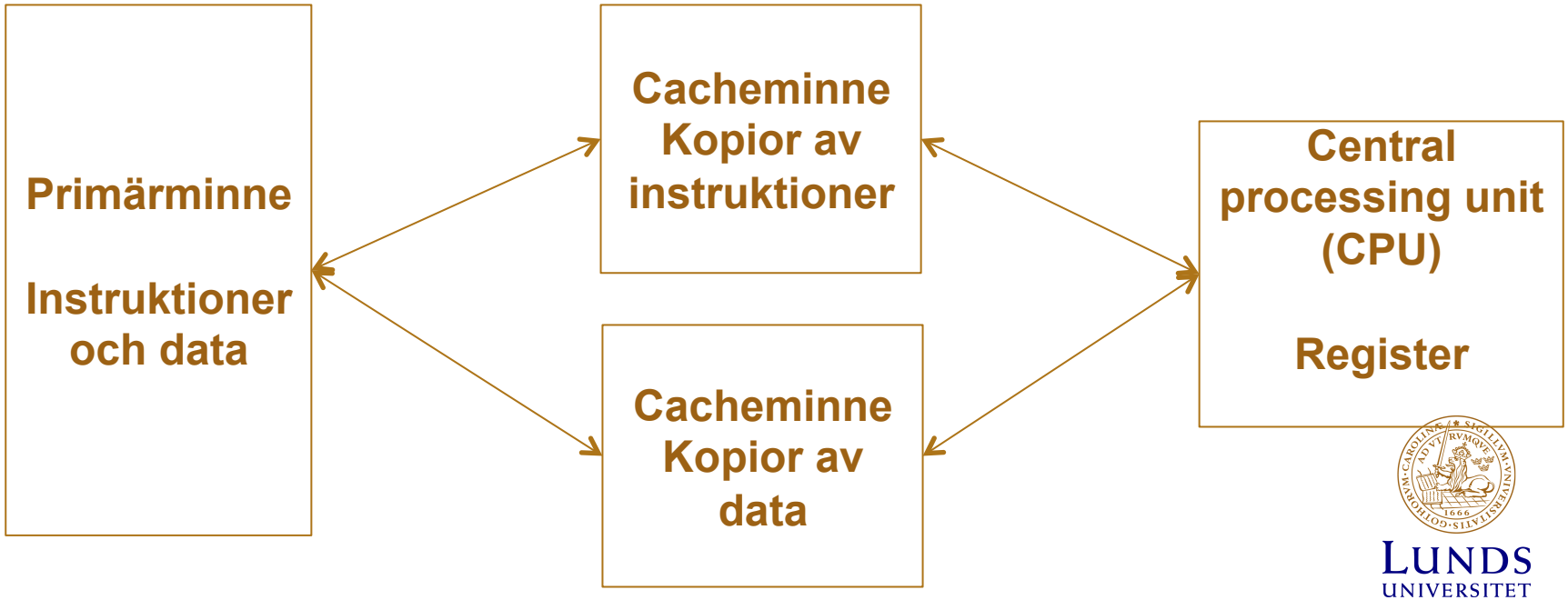


- Penalty: 1 cykel

Men här då? Inget problem då instruktionern (ADD R1, R2) använder register

Strukturella hazards

- För att minska strukturella hazards – öka antalet resurser.
- Ett klassiskt sätt att undvika hazards vid minnesaccess är att ha separat data och instruktions cache

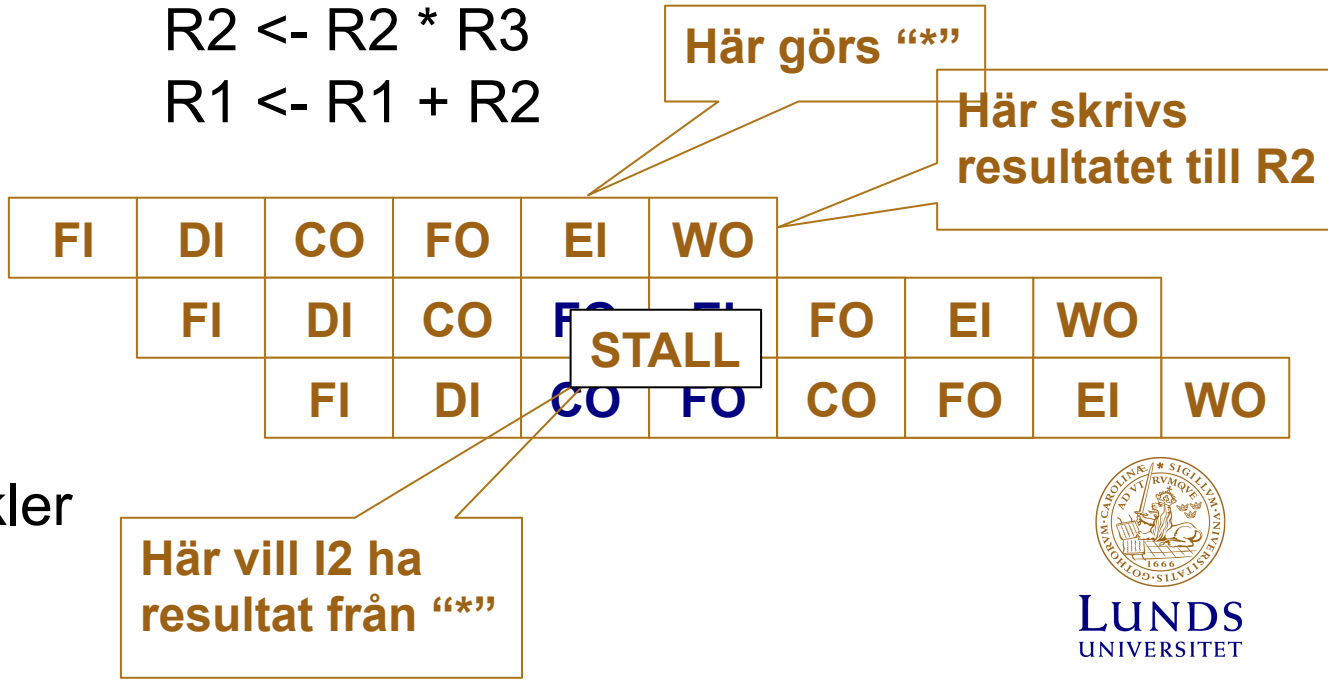


Data hazards

- Antag två instruktioner, I1 och I2. Utan pipeline, så är I1 helt färdig innan I2 startar. Men, så är det inte med pipelines. Det kan bli så att I2 behöver resultat från I1 men I1 är inte klar

I1: MUL R2, R3 R2 <- R2 * R3
I2: ADD R1, R2 R1 <- R1 + R2

MUL R2, R3
ADD R1, R2

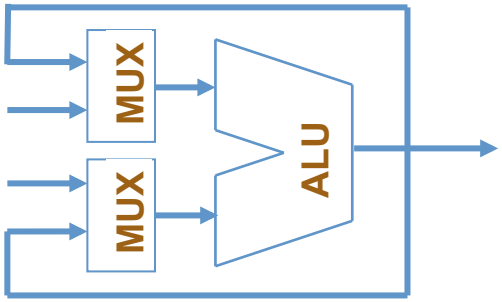


- Penalty: 2 cykler

LUNDS UNIVERSITET

Data Hazards

- För att minska “penalty” kan forwarding/bypassing användas
- Resultat från ALUn kopplas direkt (forward/bypass) till ingångar. Man behöver alltså inte vänta på WO-steget

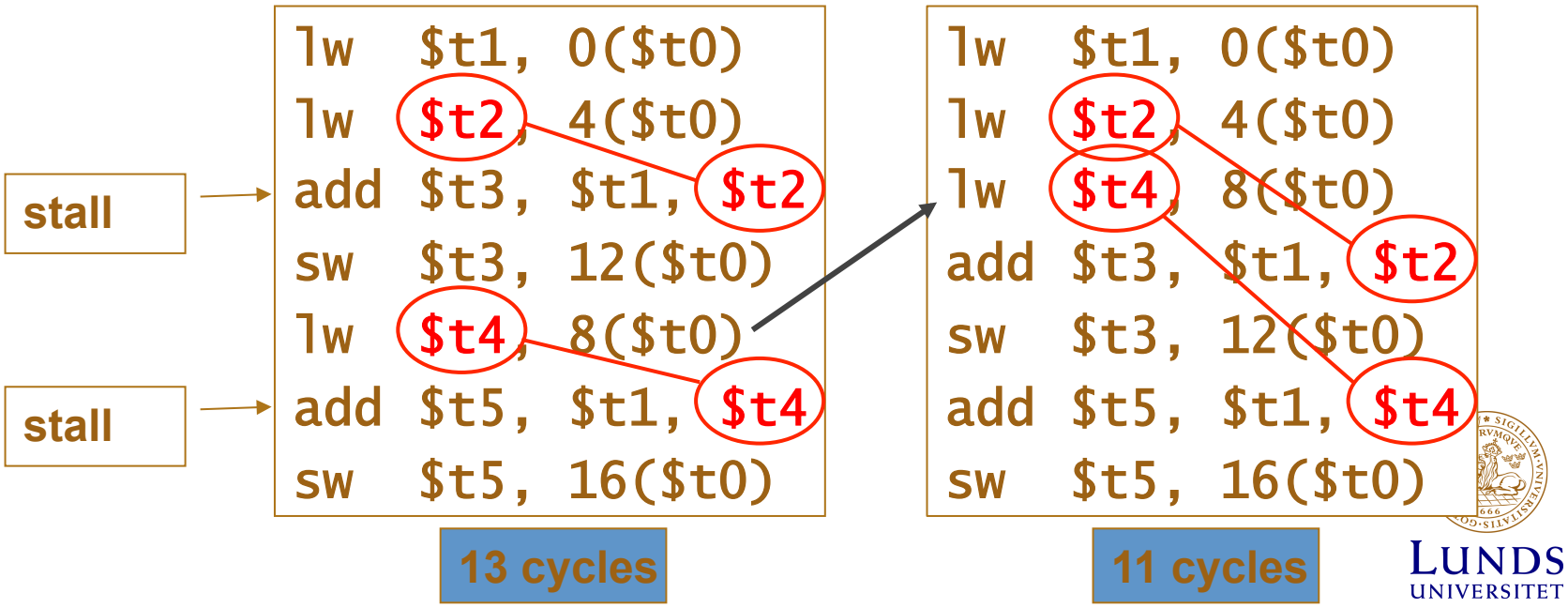


MUL R2, R3
ADD R1, R2

FI	DI	CO	FO	EI	WO		
	FI	DI	CO	Stall	FO	EI	WO

Data Hazards

- Ändra ordning på instruktioner (re-schedule) för att undvika att använda resultat från load i direkt följande instruktion
- C kod: A=B+E;
  C=B+F;



LUNDS UNIVERSITET

Kontroll hazards

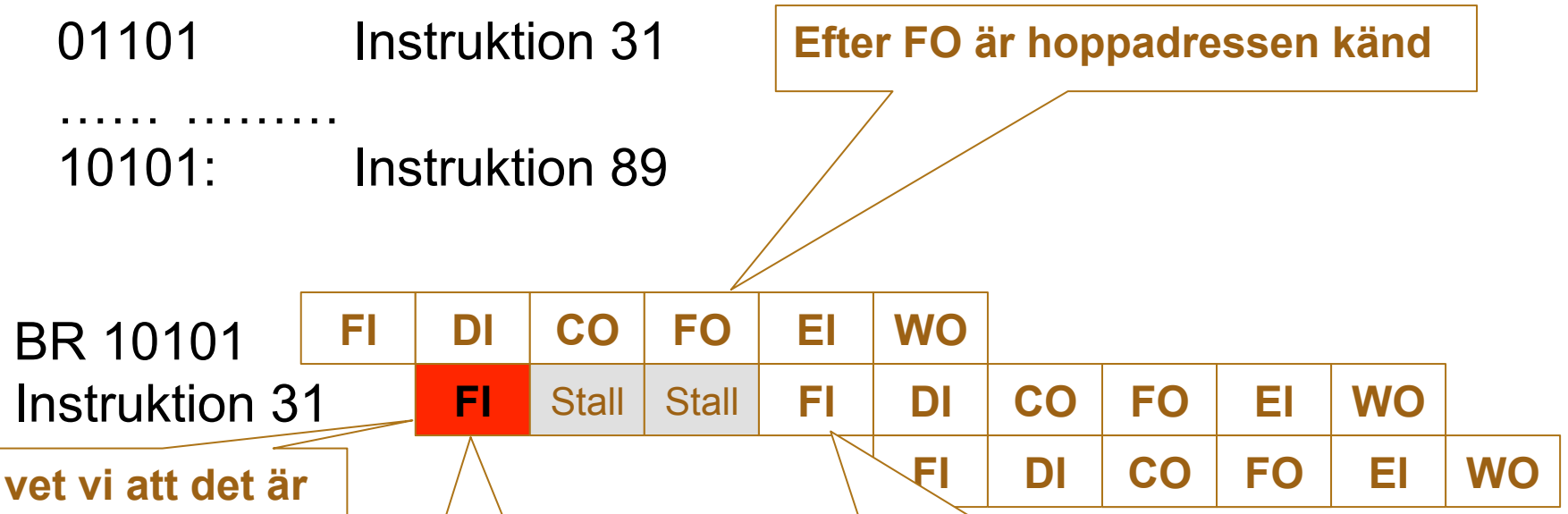
- Kontroll hazards uppkommer på grund av hopp (branch) instruktioner
- Ovillkorliga hopp (unconditional branch)

```
Adress      Instruktion      Kommentar
.....  ..........
01011       Instruktion 29   //Instruktion 29, PC=PC+1
01100       BR 10101         // PC = 10101,  PC=PC+1
01101       Instruktion 31   //Instruktion 31, PC=PC+1
.....  ..........
10101:      Instruktion 89   //Instruktion 89, PC=PC+1
.....  ..........
```

LUNDS UNIVERSITET

Kontroll hazards

```
01011          Instruktion 29
01100          BR 10101
01101          Instruktion 31
…… ……
10101:         Instruktion 89
```



- Penalty: 3

LUNDS UNIVERSITET

Kontroll hazards

- Villkorliga hopp (conditional branch)

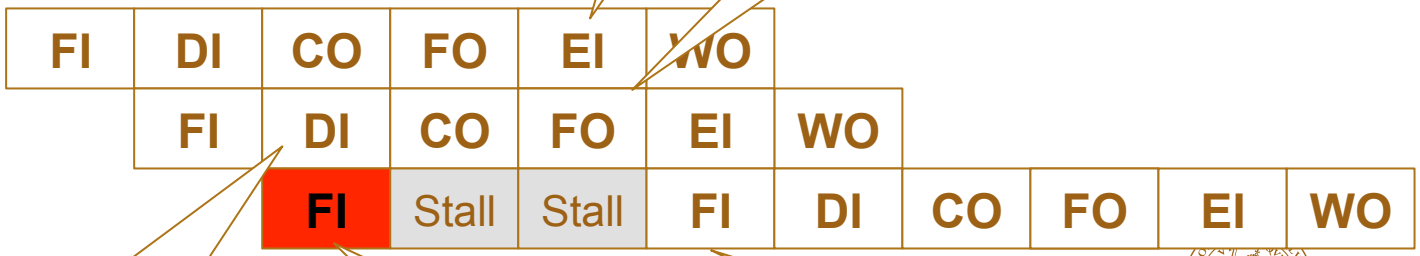
  ADD R1, R2 //R1 <- R1 + R2
  BEZ Target //Branch if zero
  instruktion 31
  ……

Target: instruktion 89

- Alternativ: Hoppet görs (Branch taken)

  ADD R1, R2
  BEZ Target
  instruktion 31

- Penalty: 3 cykler



Här görs additionen som påverkar Z-flaggan (om hoppet ska göras eller inte.

Här är hopp-adressen känd

Här vet vi att det är en hoppinstruktion

Instruktion 31 kan vara fel

Kan hämta instruktion 89

Kontroll hazards

- Villkorliga hopp (conditional branch)

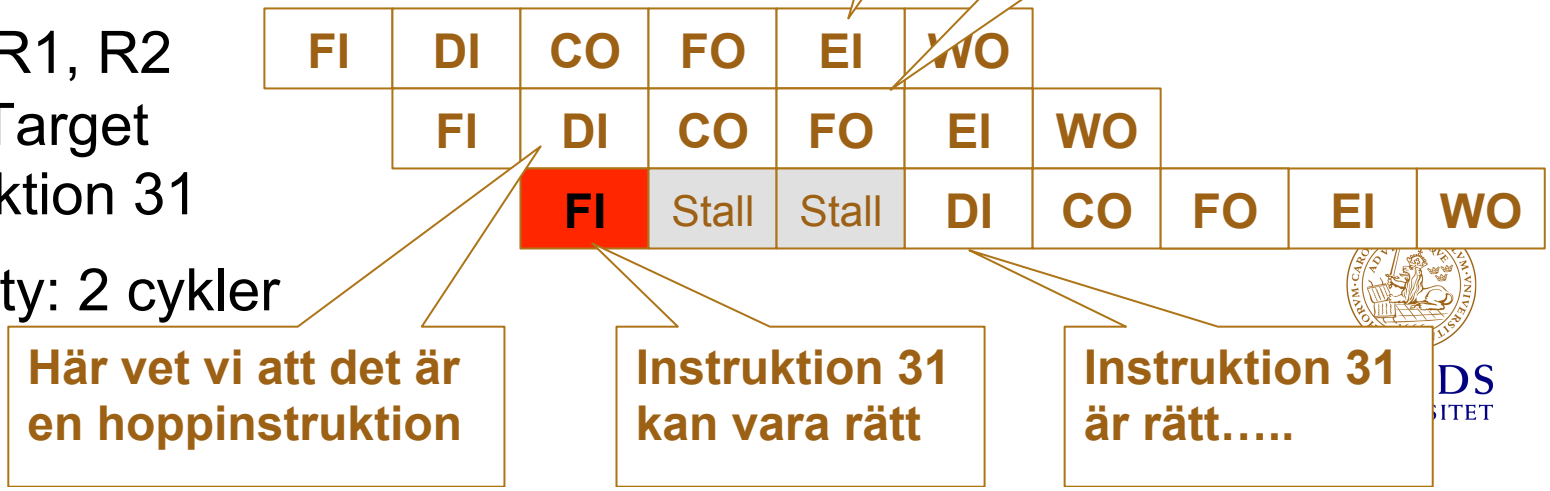
  ADD R1, R2 //R1 <- R1 + R2
  BEZ Target //Branch if zero
  instruktion 31
  ……

Target: instruktion 89

- Alternativ: Hoppet görs inte (Branch not taken)

  ADD R1, R2
  BEZ Target
  instruktion 31

- Penalty: 2 cykler



Minska penalties vid hopp

- Hoppinstruktioner påverkar prestandan av en pipeline
- Vanligt med hoppinstruktioner:
  - 20%-35% av alla instruktioner är hoppinstruktioner (villkorliga och ovillkorliga)
  - Villkorliga hopp är vanligare än ovillkorliga hopp (mer än dubbelt så många villkorliga hopp som ovillkorliga hopp)
  - Mer än 50% av villkorliga hopp tas
- Viktigt att minska penalties som uppstår pga hoppinstruktioner

LUNDS UNIVERSITET

Delayed branching

- Vid villkorliga hopp undersökte vi fallen: hopp görs och görs inte (se tidigare exempel)
- Alternativ: Hoppet görs (Branch taken)

ADD R1, R2
BEZ Target
instruktion 31

Penalty: 3 cykler

- Alternativ: Hoppet görs inte (Branch not taken)

ADD R1, R2
BEZ Target
instruktion 31

Penalty: 2 cykler

LUNDS UNIVERSITET

Delayed branching

- Tanken med delayed branching är att låta processorn göra något istället för att göra stalls
- Med delayed branching så exekverar processorn alltid den instruktion som kommer efter hoppinstruktionen, och sedan kan programflödet ändra riktning (om nödvändigt)
- Instruktionen som hamnar efter ett hopp men som exekveras oavsett utgången av hoppet, sägs ligga i branch delay slot

LUNDS UNIVERSITET

Delayed branching

- Detta har programmeraren skrivit:

```
MUL R3, R4          R3<-R3*R4
SUB #1, R2          R2<-R2-1
ADD R1, R2          R1<-R1+R2
BEZ TAR             Branch om zero
MOVE #10, R1        R1<-10
```

- Detta är det som kompilatorn (assemblatorn) genererar:

```
SUB #1, R2          R2<-R2-1
ADD R1, R2          R1<-R1+R2
BEZ TAR             Branch om zero
MUL R3, R4          R3<-R3*R4
MOVE #10, R1        R1<-10
```

Den här instruktionen påverkar inte det som händer fram till hoppet

Den här instruktionen exekveras bara om hoppet inte tas

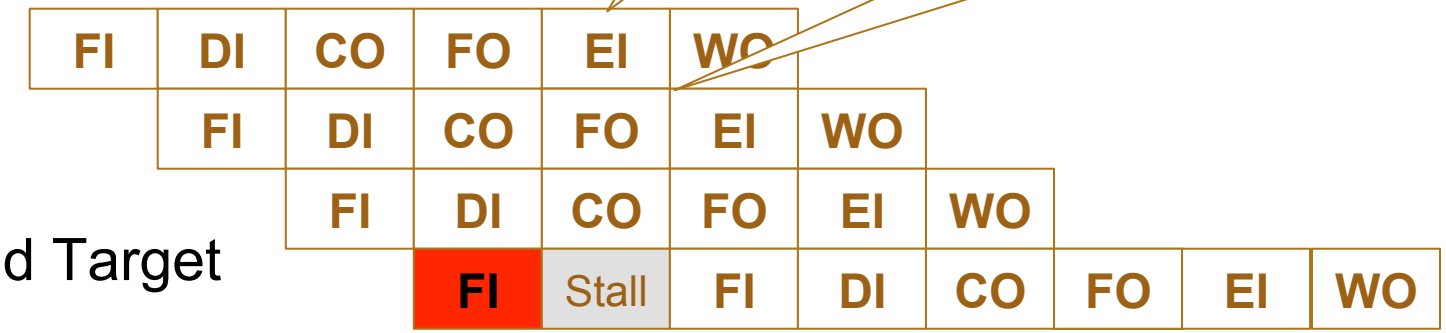
Den här instruktionen kommer exekveras oavsett om hoppet tas eller ej

Den här instruktionen kommer exekveras bara om hoppet inte tas

Delayed branching

- Alternativ: Hoppet görs (Branch taken)

ADD R1, R2
BEZ Target
MUL R3, R4
Instruktion vid Target

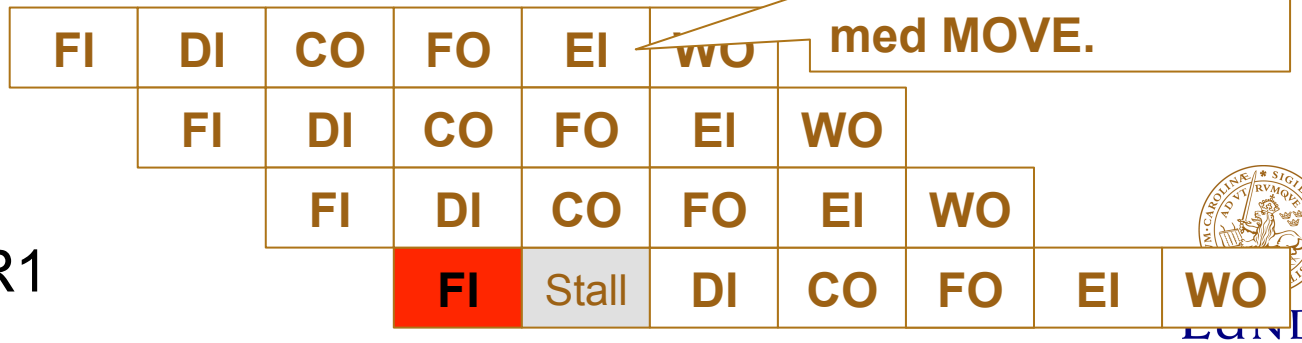


Här görs additionen som påverkar Z-flaggan (om hoppet ska göras eller inte).

Här är hopp-adressen känd

- Penalty 2 cykler (innan 3)
- Alternativ: Hoppet görs inte (Branch not take

ADD R1, R2
BEZ Target
MUL R3, R4
MOVE #10, R1



Här är det klart om vi ska fortsätta med MOVE.

- Penalty: 1 cykler (innan 2)

Delayed branching

- Om kompilatorn inte hittar en instruktion att flytta till branch delay slot, så läggs en NOP (no operation) instruktion in.

MUL **R2**, R4	R2<-R2*R4
SUB #1, R2	R2<-R2-1
ADD R1, R2	R1<-R1+R2
BEZ TAR	Branch om zero
MOVE #10, R1	R1<-10

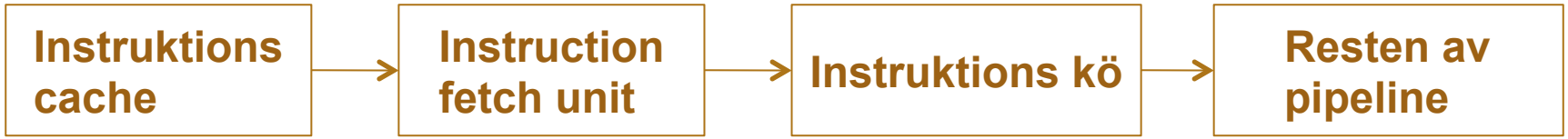
Nu kan inte MUL flyttas eftersom resultatet påverkar instruktionerna efter

- I 60%-85% av fallen med hoppinstruktioner, kan en kompilator hitta en instruktion som kan flyttas till branch delay slot

LUNDS UNIVERSITET

Instruction fetch unit och instruktionskö

- De flesta processorer har fetch unit som hämtar instruktioner innan de behövs
- Dessa instruktioner lagras i en instruktionskö



- Fetch unit kan känna igen hoppinstruktioner och generera hoppadress. “Penalty” för ovillkorliga hopp minskar. Fetch unit kan alltså hämta instruktion enligt hopp.
- För villkorliga hopp är det svårare då man måste veta om hopp ska tas eller inte

LUNDS UNIVERSITET

Branch prediction

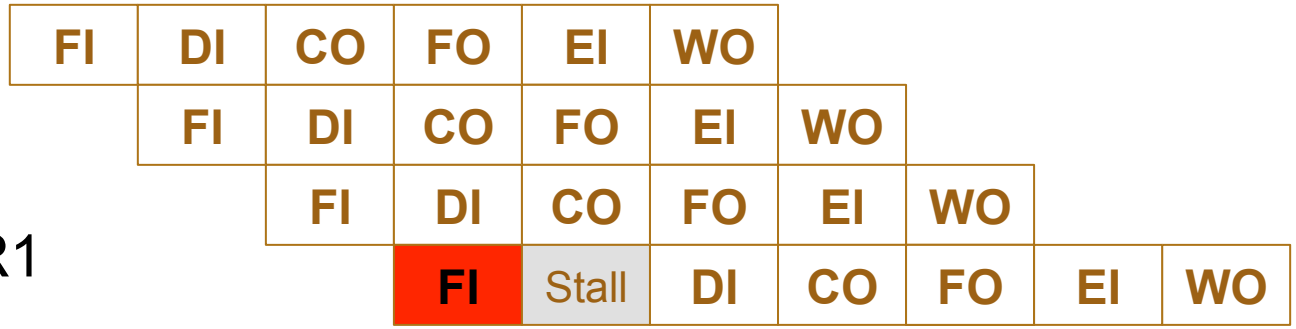
- I förra exemplet såg vi att i fallet branch not taken – där vi fortsatte med nästa instruktion fick följande:
  - Branch penalty:
    - » 1 om branch **not** taken (prediction fullfilled)
    - » 2 om branch is taken (prediction is not fullfilled)
- Vi kan anta motsatsen, dvs branch taken. Då gäller följande (se detaljer på nästa slide):
  - Branch penalty:
    - » 1 om branch is taken (prediction fullfilled)
    - » 2 om branch is **not** taken (prediction is not fullfilled)

LUNDS UNIVERSITET

Branch prediction

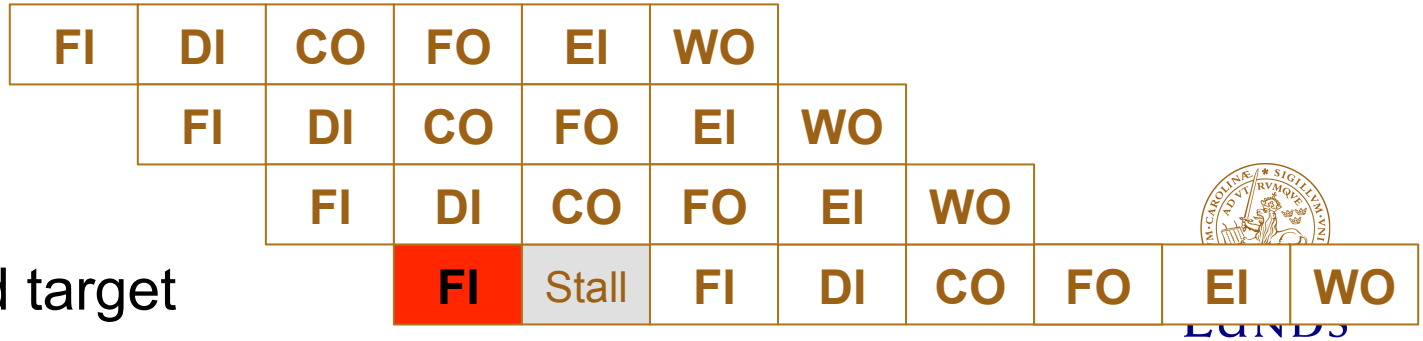
- Antagande (prediction): Nästa instruktion exekveras (inget hopp)
- Alternativ: Hoppet görs inte (Branch not taken) (om vi gissat rätt!)

ADD R1, R2
BEZ Target
MUL R3, R4
MOVE #10, R1



- Penalty 1 cykler
- Alternativ: Hoppet görs (Branch taken)(om vi gissat fel!)

ADD R1, R2
BEZ Target
MUL R3, R4
Instruktion vid target

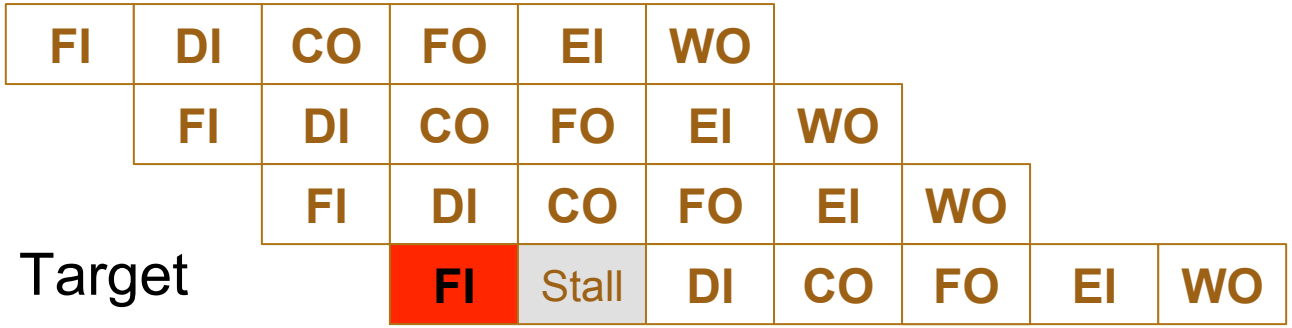


- Penalty: 2 cykler

Branch prediction

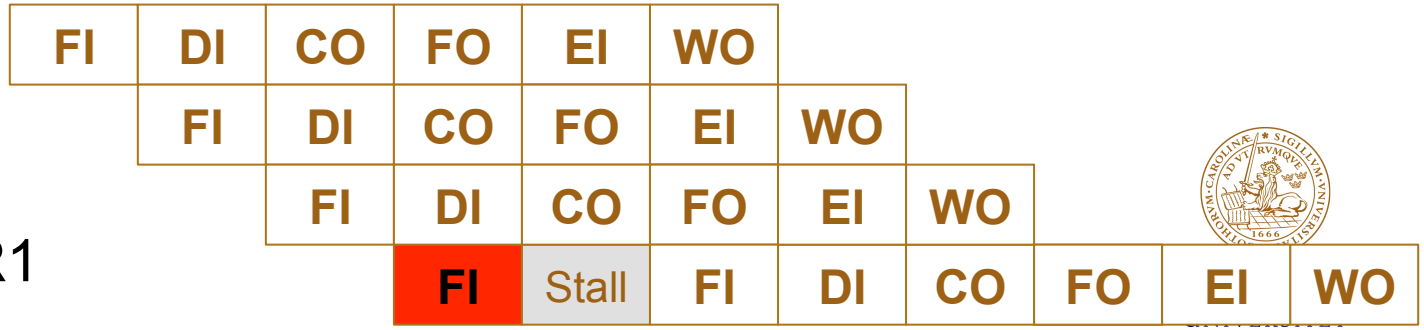
- Antagande (prediction): Instruktion vid target exekveras (hopp görs)
- Alternativ: Hoppet görs (Branch taken)(Om vi gissar rätt)

ADD R1, R2
BEZ Target
MUL R3, R4
Instruktion vid Target



- Penalty 1 cykler
- Alternativ: Hoppet görs inte (Branch not taken)(Om vi gissar fel)

ADD R1, R2
BEZ Target
MUL R3, R4
MOVE #10, R1



- Penalty: 2 cykler

Branch prediction

- Rätt branch prediction är viktigt
- Baserat på prediction, kan respektive instruktion hämtas (och näst följande) för att placeras i instruktionskön.
- När hoppvillkoret är bestämt, kan exekveringen fortsätta (enligt gissningen (prediktering).
- Om gissningen är fel, måste hämtning av “rätt” instruktioner göras.
- För att utnyttja branch prediction maximalt, kan vi påbörja exekveraing – innan hoppvillkoret är bestämt – spekulativ exekvering

LUNDS UNIVERSITET

Speculative execution

- Med speculativ exekvering menas att instruktioner exekveras innan processorn vet om det är rätt instruktion som ska exekveras.
- Om gissningen (spekuleringen) var rätt, kan processorn fortsätta. Annars, för man göra om (hämta rätt instruktion)
- Branch prediction strategies:
  - Statisk prediktering
  - Dynamisk prediktering

LUNDS UNIVERSITET

Static branch prediction

- I statisk branch prediktering tas ingen hänsyn till exekverings historiken
- Statiska tekniker:
  - Predict never taken – antar att hoppet inte kommer tas (Motorola 68020)
  - Predict always taken – antar att hoppet alltid kommer tas
  - Predict beroende på riktning (Power PC 601):
    - » Predict branch taken för tillbaka hopp
    - » Predict branch not taken för framåt hopp

LUNDS UNIVERSITET

Dynamisk branch prediction

- I dynamisk branch prediction tas hänsyn till exekverings historik
- En bit för prediktion
  - Sparar utfall från förra gången hoppinstruktionen användes och man predikterar (gissar) att samma sak ska hända som förra gången. Om hoppet inte togs förra gången, gissar man att man inte tar det (och vice versa)
- Nackdel:

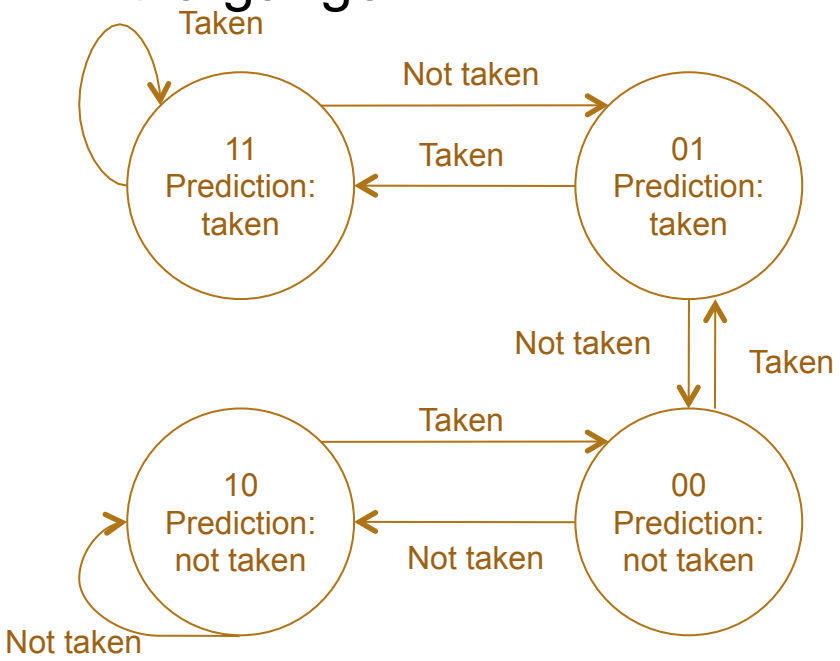
```
LOOP
    .....
    BNZ LOOP
```

- När loopen exekverades förra gången, noterades att hoppet inte ska tas. Vid ny exekvering, tas inte hoppet (det är fel gissat). Sedan gissas rätt, fram tills slutvillkor är uppfyllt. Totalt 2 fel.
- Vid statisk prediction (backwards tas alltid) ges ett fel

LUNDS UNIVERSITET

Tvåbitars prediktering

- Med tvåbitars prediktering görs gissning baserat på de senaste två fallen av en instruktion.
- Vanligt att ändra "gissning" (prediction) när man gissat fel två gånger.



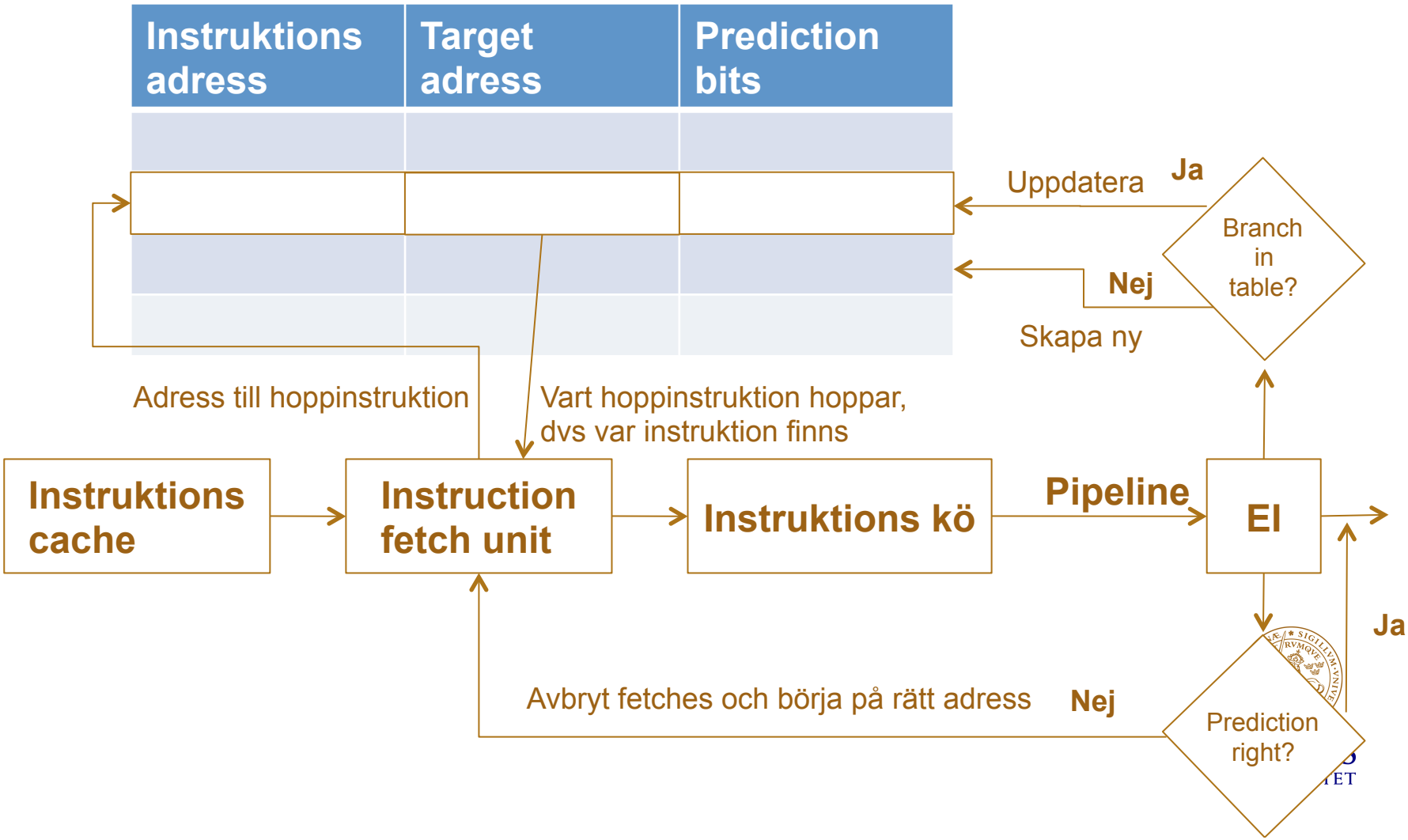
- LOOP

  …..

  BNZ LOOP…
- Antag: 11 för BNZ. Så länge hopp tas är det rätt. Sista gången tas ej hoppet. Tillstånd flyttas till 01 (men prediction samma)

LUNDS UNIVERSITET

Branch history table (branch target buffer)



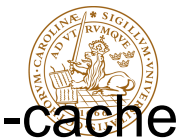
Intel 80486 pipeline

- Intel 80486 är den sista icke-superscalar processorn från Intel. 80486 är exempel på avancerad non-superscalar pipeline
- Fem pipeline steg och ingen branch prediction (always not taken)
  - Fetch: Instruktioner hämtas från cache och placeras i instruktionskö. Instruktionskön organiserad som två pre-fetch buffrar som opererar operoende av varandra för att hålla pre-fetch buffert full)
  - Decode1: Tar de 3 första byten i en instruktion avkodar: opkod, adresserings mode och instruktionslängd
  - Decode2: Avkodar resten av instruktionen och producerar kontrollsignaler. Gör adressberäkning.
  - Execute: ALU operations. Cache access för operander.
  - Write back: Updaterar register, status flaggor.

LUNDS UNIVERSITET

ARM pipeline

- ARM7 pipeline (3-steg)
  - Fetch: hämtar instruktioner från cache
  - Decode: Avkodar
  - Execute: Läser register, gör ALU beräkningar och skriver tillbaka svar
- ARM9 pipeline (5-steg):
  - Fetch: hämtar instruktioner från cache
  - Decode: Avkodar
  - Execute: Läser register, gör ALU beräkningar
  - Data memory access: skriver tillbaka svar till/från D-cache
  - Register write: Skriver till register



Fallgropar

- Pipelining är enkelt
  - Principen är enkel, problemen finns i detaljerna:
    - » T ex detektera kontroll hazards
- Pipelining är teknologioberoende
  - Varför inte använt pipelining innan?
    - » Mer transistorer gör det möjligt med mer avancerade tekniker
- Dålig design av instruktionsuppsättning (ISA) kan göra pipelining svårare
  - Komplex instruktionsuppsättning
  - Komplexa adresseringsmöjligheter

LUNDS UNIVERSITET

Karaktärsdrag hos RISC processorer (FÖ2)

- Enkla och få instruktioner
- Enkla och få adresseringsmöjligheter
- Fixt (fast) instruktionsformat
- Stort antal register
- Load-and-store architecture

LUNDS UNIVERSITET

Karaktärsdrag hos RISC processorer

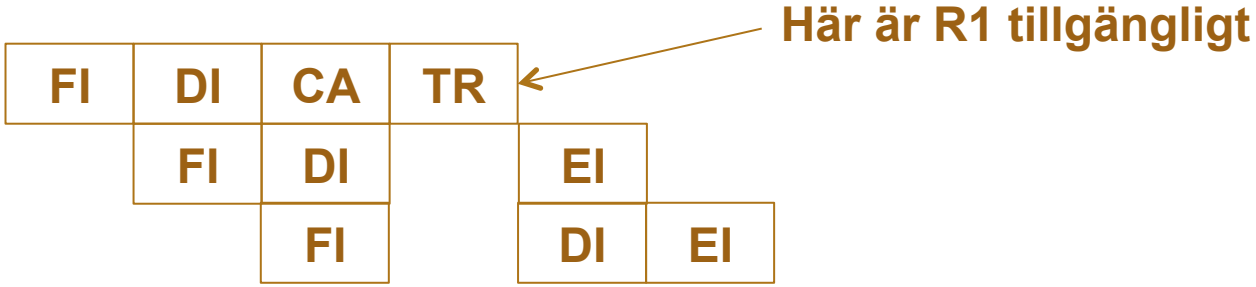
- Load-and-store architecture:
  - Endast LOAD och STORE instruktioner refererar data i minnet (primärminnet). Alla andra instruktioner opererar enbart på register (register-till-register instruktioner). Alltså, endast ett fåtal instruktioner behöver mer än en klockcykel för att exekvera (efter fetch och decode)
  - Pipeline vid minnesreferenser:

  - CA: Compute adress, TR: Transfer

Delayed load problem

- LOAD och STORE instruktioner refererar data i minnet (primärminnet). Alla andra instruktioner opererar enbart på register (register-till-register instruktioner).
- LOAD och STORE instruktioner hinner inte bli exekverade på en klockcykel.

LOAD R1, X
ADD R2, R1
ADD R4, R3



LUNDS UNIVERSITET

Delayed load problem

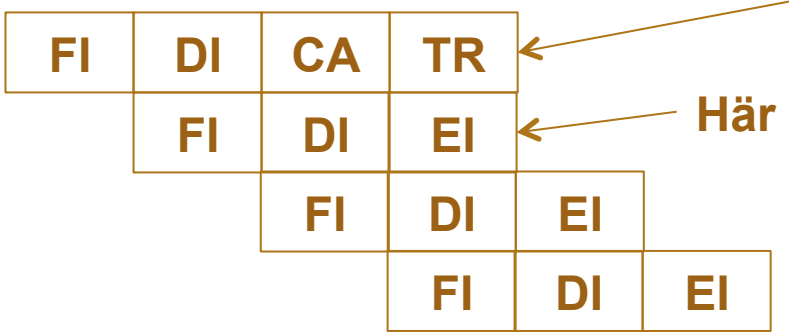
- Två alternativ:
  - Hårdvaran gör delay (stall) av instruktioner som följer efter LOAD
  - Kompilatorn ändrar ordningen genom delayed load (jämför med delayed branching)
- Med delayed load exekveras alltid instruktionen som följer en LOAD instruktion (inget stall). Det är programmerarens (kompilatorns) ansvar att se till att instruktionen direkt efter LOAD inte behöver just det värde som laddas.

LUNDS UNIVERSITET

Delayed load problem

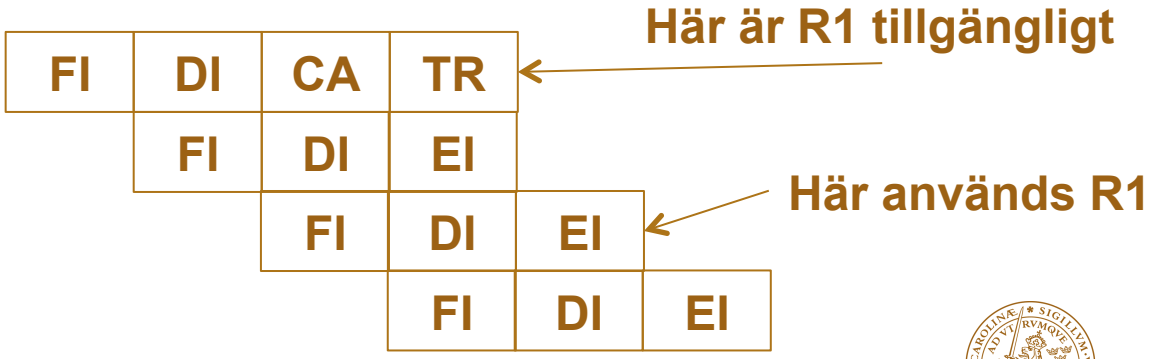
- Om programmet är:

```
LOAD R1, X
ADD R2, R1
ADD R4, R3
SUB R2, R4
```



- Genererar kompilatorn:

```
LOAD R1, X
ADD R4, R3
ADD R2, R1
SUB R2, R4
```

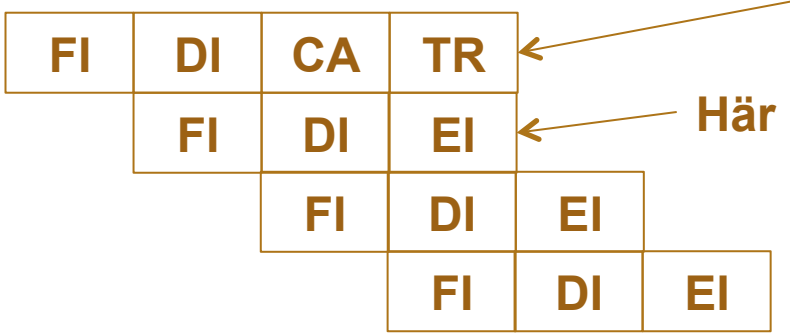


LUNDS UNIVERSITET

Delayed load problem

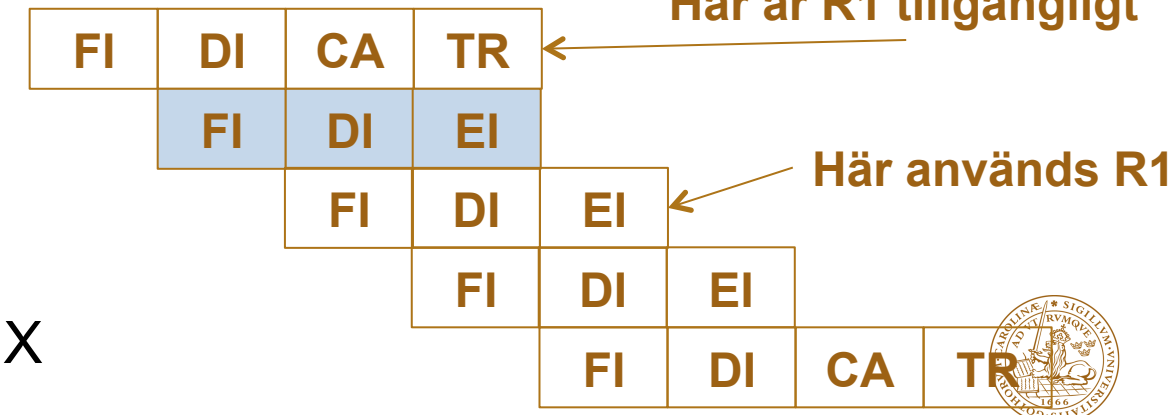
- Om programmet är:

```
LOAD R1, X
ADD R2, R1
ADD R4, R2
SUB R4, X
```



- Genererar kompilatorn:

```
LOAD R1, X
NOP
ADD R2, R1
ADD R4, R2
STORE R4, X
```



- Om kompilatorn inte hittar en instruktion, tas en NOP (no operation) instruktion

LUNDS UNIVERSITET

Sammanfattning

- Instruktioner exekveras som en sekvens av steg; t ex Fetch instruction (FI), Decode instruction (DI), Calculate operand address (CO), Fetch operand (FO), Execute instruction (EI) och Write operand (WO)
- En pipeline består av N steg. Vid ett visst ögonblick kan N instruktioner vara aktiva. Om 6-steg används (FI, DI, CO, FO, EI, WO) kan 6 instruktioner vara aktiva
- Att hålla piplinen full med instruktioner är svårt pga pipeline hazards.
  - Strukturella hazards beror på resurskonflikter.
  - Data hazards beror på detaberoende mellan instruktioner
  - Control hazards beror på hoppinstruktioner

LUNDS UNIVERSITET

Sammanfattning

- Hoppinstruktioner påverkar pipelinen mycket. Viktigt att minska penalties
- Instruktion fetch unit kan känna igen hoppinstruktioner och ta fram vart hoppet går. Genom att hämta snabbt från instruktions cachen och hålla instruktionskön full så kan penalties för ovillkorliga hopp reduceras till noll. För villkorliga hopp är det svårare då man måste veta om hoppet ska tas eller ej
- Delayed branching är en kompilator-baserad teknik för att minska branch penalty genom att flytta instruktioner till branch delay slot
- Viktigt för att minska branch penalty är att ha bra branch prediction teknik. Statiska tekniker tar inte hänsyn till exekverings historik, vilket dynamiska tekniker gör
- Branch history tables används för att lagra utfall av ett hopp och target adress (vart hoppet går)

LUNDS
UNIVERSITET