



LUNDS  
UNIVERSITET

# Dator teknik

---

ERIK LARSSON



# Så här långt....

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

FÖ2

Lab1

Compiler

RISC/CISC

swap:

```
muli $2, $5,4
add $2, $4,$2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```

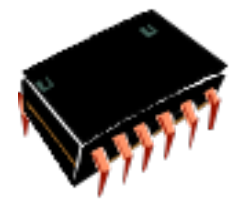
Assembler

FÖ3

Lab3

```
000000001010000100000000000011000
000000000000110000001100000100001
100011000110001000000000000000000
100011001111001000000000000000100
101011001111001000000000000000000
101011000110001000000000000000100
000000111110000000000000000000100
```

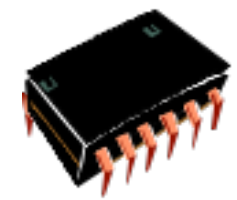
Binary machine language program (for MIPS)



I/O

Lab2

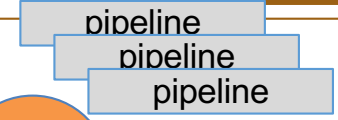
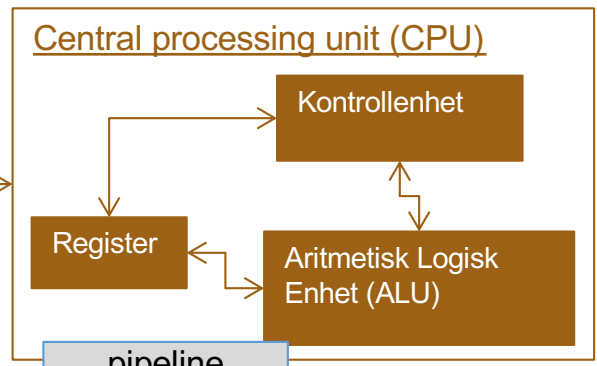
FÖ1



Assembly language program (for MIPS)

Primärminne  
Instruktioner och data

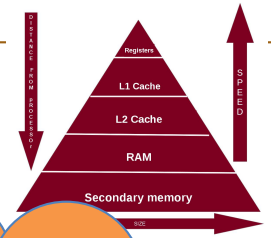
Address	Instruction
00001000	0000101110001011
00001001	0001101110000011
00001010	0010100000011011
00001011	0001001110010011



FÖ4

Lab4

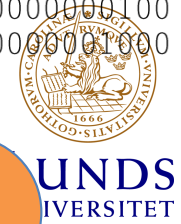
FÖ2



FÖ5 FÖ6



FÖ2



# Översikt

---

- Parallellism
- Parallellism på instruktions-nivå
  - Superscalar processors
  - Very Long Instruction Word processors
- Parallellism på tråd-nivå

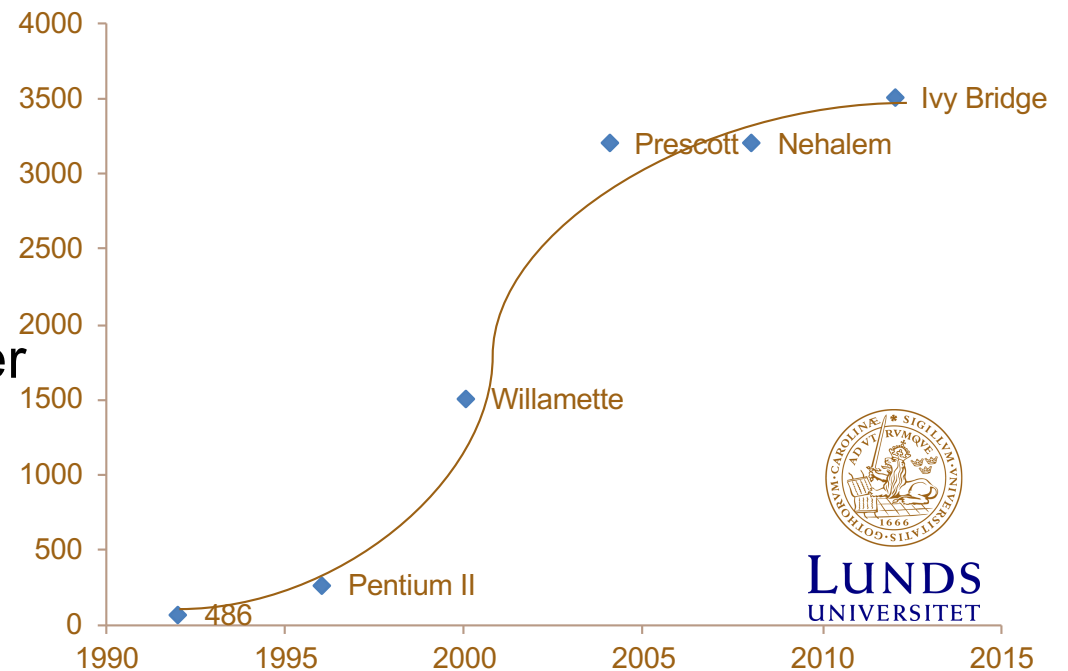


# Inledning

---

- Konstant behov av högre prestanda
- Prestanda har uppnåtts genom:
  - Utveckling inom halvledartechnik
  - Tekniker som:
    - » Cacheminne
    - » Flera bussar
    - » Pipelining
    - » Superscalar arkitekturer

Ökning av klockfrekvens över tiden



# Klassificering av datorarkitekturer

---

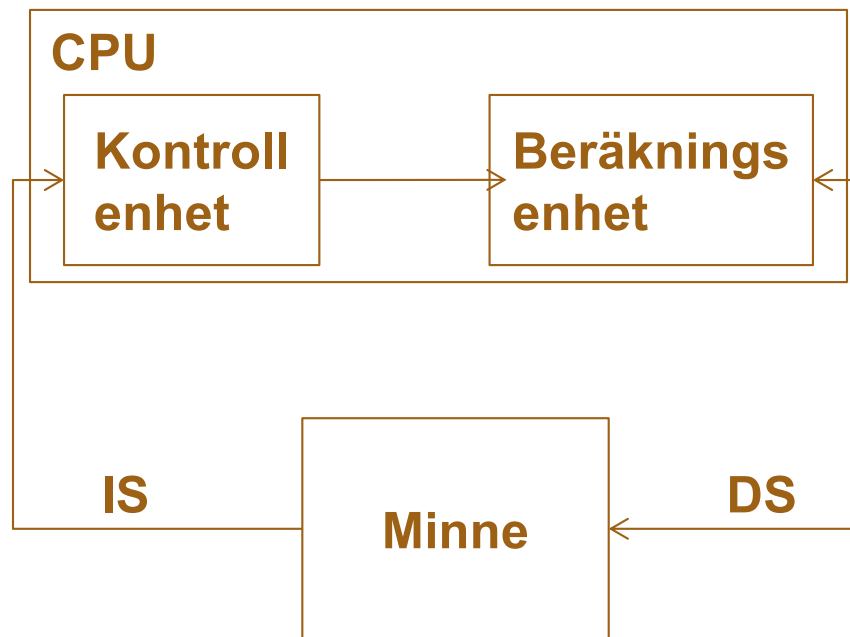
- Baserat på instruktionsflödet och dataflödet :
  - Single-instruction single-datastream (SISD)
  - Single-instruction multiple-datastream (SIMD)
  - Multiple-instruction single-datastream (MISD)
  - Multiple-instruction multiple-datastream (MIMD)
- Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, Vol. C-21, pp. 948, 1972.



# Single-instruction single-datastream (SISD)

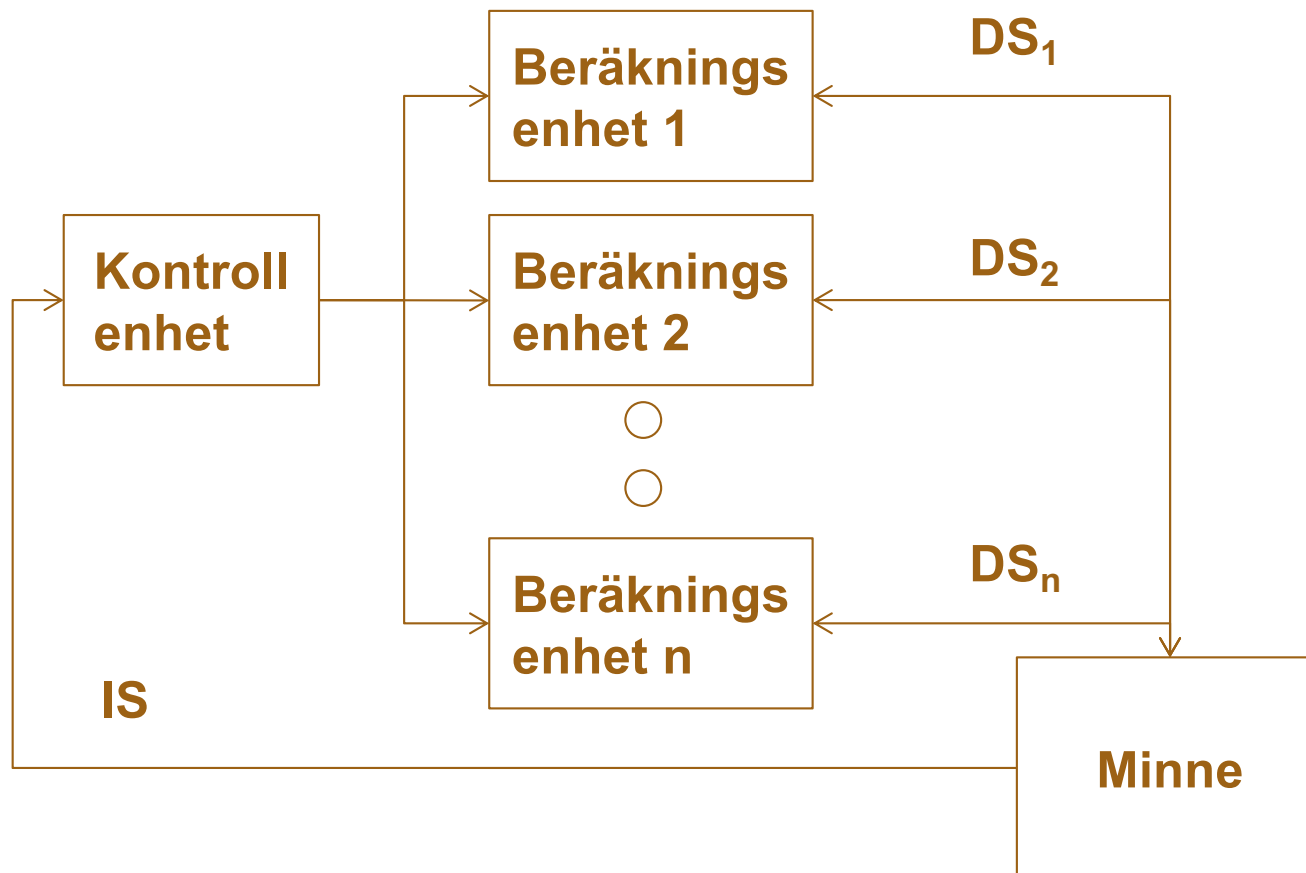
---

- En ström av instruktioner (IS) och en ström av data (DS)



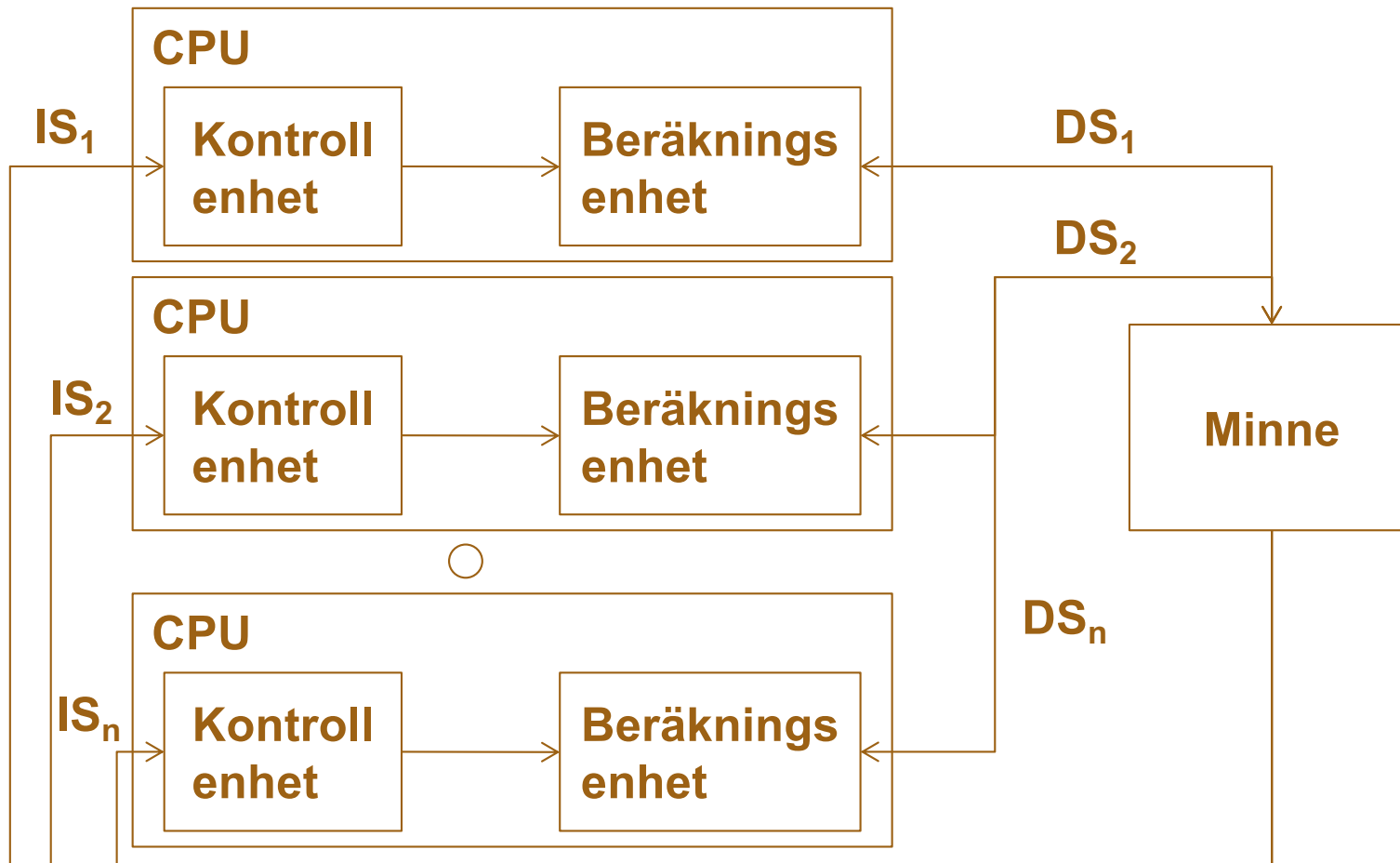
# Single-instruction multiple-datastream (SIMD)

- En ström av instruktioner (IS) och flera strömmar av data (DS)



# Multiple-instruction multiple-datastream (MIMD)

- Flera strömmar av instruktioner (IS) och flera strömmar av data (DS)





# Prestanda på parallella arkitekturer

---

- Hur snabbt kan arkitekturer maximalt exekvera?
- Hur snabbt exekverar arkitekturer ett normalt program?
- Hur mäta prestanda på en parallell arkitektur?
- Hur mäta prestanda om ytterligare funktionella beräkningsenheter läggs till?



# Prestanda på parallella arkitekturer

---

- Peak rate: den teoretiska gränsen på hur snabbt en arkitektur kan exekvera. Ofta har peak rate begränsat intresse eftersom det är svårt att uppnå peak rate
- Speedup (**S**): mäter vinsten med en parallell arkitektur jämfört med sekventiell exekvering:

$$\mathbf{S} = \mathbf{T}_s / \mathbf{T}_p$$

där **T<sub>s</sub>** är exekveringstid vid sekventiell exekvering och  
**T<sub>p</sub>** är exekveringstid med parallell exekvering

- Efficiency (**E**) relaterar speedup till antal processorer, **E = S/p** där **S** är speedup och **p** är antal processorer.
  - Idealt (teori) ska: **S = p** vilket leder till **E = 1**



# Amdahls lag

- Antag att ett program (algoritm) implementeras så att **f** är den del som *måste* exekveras sekventiellt. Då kan resten (**1-f**) exekveras parallellt.

Denna del måste exekvera sekventiellt

Denna del kan exekvera parallellt

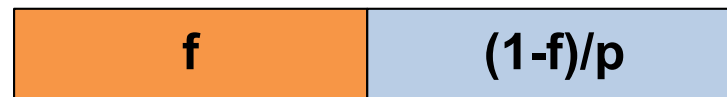
- En CPU:



$$T_s = f \cdot T_s + (1-f) \cdot T_s$$



- p CPUer:



$$T_p = f \cdot T_s + (1-f) \cdot T_s / p$$



# Amdahls lag

---

- Exekveringstid:  $T_p = f \cdot T_s + (1-f) \cdot T_s / p$
- Speedup:  $S = T_s / T_p$
- Efficiency:  $E = S / p$
- Exempel: om  $f = 20\%$  och arkitekturen har 4 processorer, fås:
  - Speedup  $S = 1 / (0.2 + (1-0.2)/4) = 2.5$
  - Efficiency  $E = S / p = 2.5 / 4 = 0.625$
- Även en liten bit *sekventiell exekvering* ( $1/f$ ) leder till begränsningar på speedup och efficiency
  - För att öka speedup och efficiency måste mer exekveras parallellt



# Översikt

---

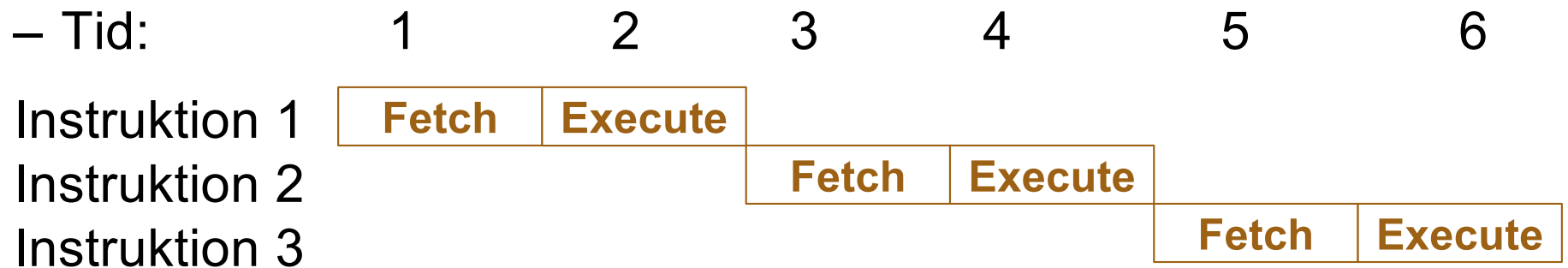
- Parallellism
- Parallellism på instruktions-nivå
  - Superscalar processors
  - Very Long Instruction Word processors
- Parallellism på programs tråd-nivå



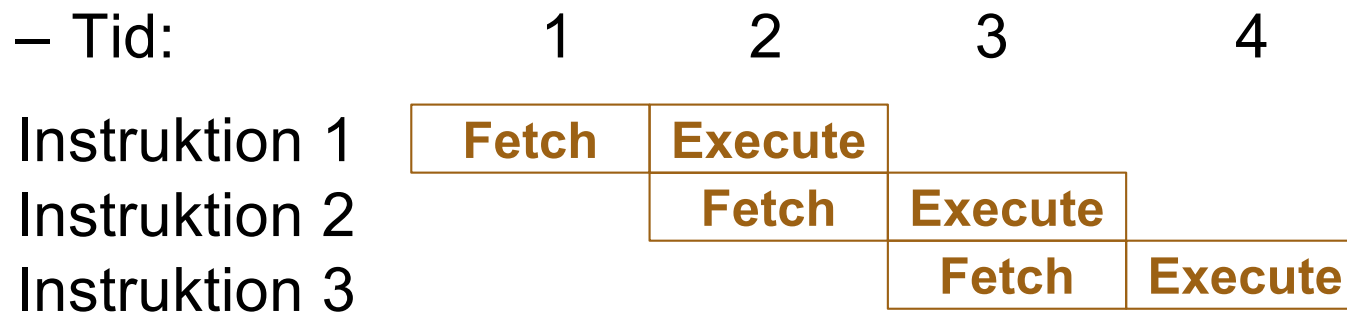
# Pipelining

---

- Utan pipelining:



- Med pipelining:



# Superscalar architecture

---

- En superscalar architecture (SSA) tillåter att mer än en instruktion initieras samtidigt och exekveras oberoende av varandra.
  - Skalär: heltal, flyttal
  - Icke-skalär: array, matris, vektor
- I pipelining är flera instruktioner aktiva men de är alla i olika pipeline steg
- Utöver pipelining, tillåter SSA att flera instruktioner exekverar samtidigt (i samma pipeline steg). Det är möjligt att initiera flera instruktioner i samma klockcykel; instruction level parallelism (ILP)



# Superscalar architecture (SSA)

---

- Ett exempel, tre parallella enheter (pipelines):
  - en flyttalsinstruktion och två heltalsinstruktioner kan exekveras samtidigt (som tidigare kan flera instruktioner vara aktiva i en pipeline)

Flyttal:	FI	DI	CO	FO	EI	WO
Heltal:	FI	DI	CO	FO	EI	WO
Heltal:	FI	DI	CO	FO	EI	WO
	FI	DI	CO	FO	EI	WO
	FI	DI	CO	FO	EI	WO
	FI	DI	CO	FO	EI	WO





# Instruktionsfönster

---

- Instruktionsfönster (instruction window) är de instruktioner som processorn har att välja mellan vid en viss tidpunkt.
- Instruktions fönstret ska vara så stort som möjligt för att ge möjlighet att effektivt välja instruktioner som ska exekveras
- Problem:
  - Svårt att hämta instruktioner i hög hastighet
  - Konflikter i pipeline (branches, struktural, data)



# Problem med parallell exekvering

---

- Problem som förhindrar parallell exekvering i SSA liknar de problem som skapade problem i pipelines
- Konflikter i pipelines:
  - Resource (resurs) konflikter
  - Kontrollkonflikter
  - Datakonflikter
    - » True data dependency
    - » Output dependency
    - » Anti-dependency



# True data dependency

---

- True data dependency (data konflikt/hazard) uppkommer när resultat (output) från en instruktion behövs som indata i följande instruktion:

MUL <b>R4</b> , R3, R1	$R4 \leftarrow R3 * R1$
-----	
ADD R2, <b>R4</b> , R5	$R2 \leftarrow R4 + R5$



# True data dependency

MUL R4, R3, R1  
 ADD R2, R4, R5

FI	DI	CO	FO	EI	WO				
FI	DI	CO	FO	EI	FO	EI	WO		
	FI	DI	STALL		CO	FO	EI	WO	
	FI	DI	CO	FO	CO	FO	EI	WO	

Här görs "\*"

Här skrivs  
 resultatet till R4

Här vill I2 ha  
 resultat från "\*"



# Output dependency

---

- Output dependency existerar om två instruktioner skriver till samma plats. Om den senare instruktionern exekveras före den första blir det fel:

MUL **R4**, R3, R1      $R4 \leftarrow R3 * R1$

-----

ADD **R4**, R2, R5      $R4 \leftarrow R2 + R5$



# Anti-dependency

---

- En anti-dependency existerar om en instruktion använder en plats (minne/register) medan någon nästföljande instruktion skriver till den platsen.
- Om den första instruktionen inte är klar medan en efterföljande instruktion skriver på samma plats, blir det fel:

MUL R4, <b>R3</b> , R1	R4 ← R3 * R1
-----	
ADD <b>R3</b> , R2, R5	R3 ← R2 + R5



# Register renaming: Output dependency och anti-dependency

- Output dependency

- Givet var:

MUL R4, R3, R1     **R4** ← R3\*R1

-----

ADD R4, R2, R5     **R4** ← R2+R5

- Med nya register:

MUL R4, R3, R1     **R4** ← R3\*R1

-----

ADD R7, R2, R5     **R7** ← R2+R5

- Anti dependency

- Givet var:

MUL R4, R3, R1     R4 ← **R3**\*R1

-----

ADD R3, R2, R5     **R3** ← R2+R5

- Med nya register:

MUL R4, R3, R1     R4 ← **R3**\*R1

-----

ADD R6, R2, R5     **R6** ← R2+R5



# Parallell exekvering

---

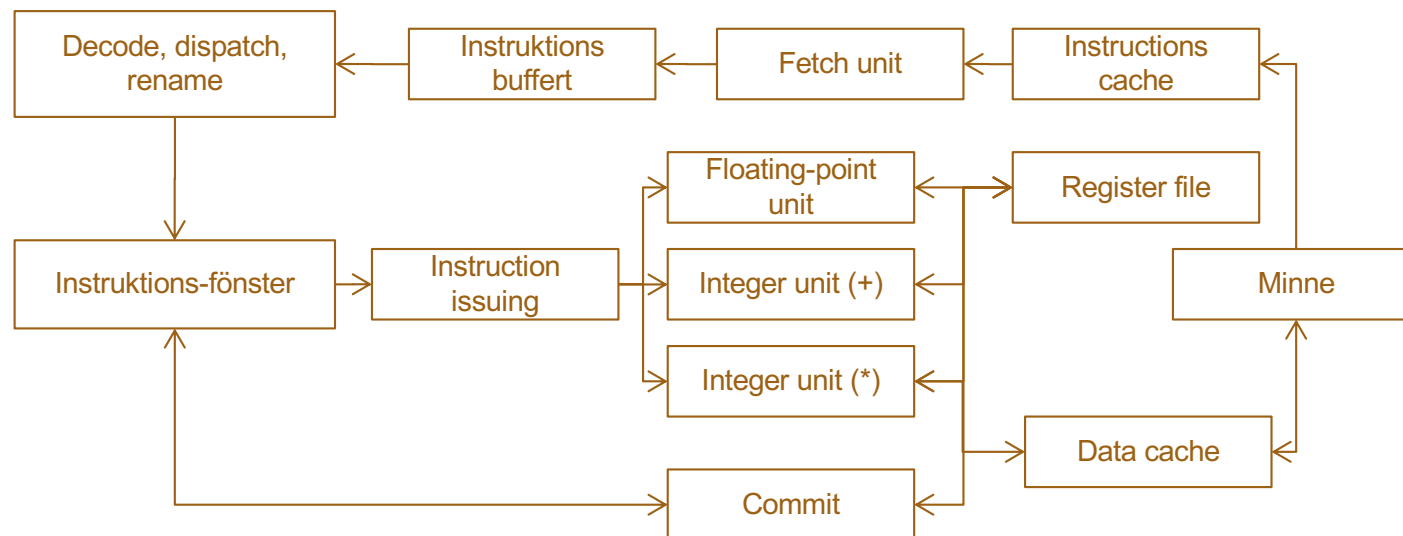
- Instruktioner kan alltså exekveras i en annan ordning än den som är given i programmet. Detta är inget problem så länge resultatet blir rätt.
- Exekveringspolicies:
  - In-order issue with in-order completion
  - In-order issue with out-of-order completion
  - Out-of-order issue with out-of-order completion





# Parallell exekvering

- Antag en superscalar processor:
  - Två instruktioner kan hämtas (fetch) och avkodas (decoded)
  - Tre funktionella enheter kan arbeta parallellt; en flyttalsenhet, en heltalsadderare, och en heltalsmultiplicerare
  - Resultat från två instruktioner kan skrivas tillbaka (write back) samtidigt



# Parallell exekvering

I3 och I4 är i konflikt om samma funktionella enhet (\*)

- Antag ett program:

I1: ADDF R12, R13, R14

I2: ADD R1, R8, R9

I3: MUL R4, R2, R3

I4: MUL R5, R6, R7

I5: ADD R10, R5, R7

I6: ADD R11, R2, R3

R12 ← R13 + R14 (float. pnt.)

R1 ← R8 + R9

R4 ← R2 \* R3

R5 ← R6 \* R7

R10 ← R5 + R7

R11 ← R2 + R3

I5 behöver resultatet från beräkningen i I4

I2, I5 och I6 är i konflikt om samma funktionella enhet (+)

- Alla instruktioner kräver 1 klockcykel för exekvering förutom I1 som kräver två klockcykler



# In-order issue with in-order completion

I1: ADDF R12, R13, R14      R12 ← R13 + R14 (flyttalsberäkning)  
 I2: ADD R1, R8, R9          R1 ← R8 + R9  
 I3: MUL R4, R2, R3          R4 ← R2 \* R3  
 I4: MUL **R5**, R6, R7        R5 ← R6 \* R7  
 I5: ADD R10, **R5**, R7        R10 ← R5 + R7  
 I6: ADD R11, R2, R3         R11 ← R2 + R3

Decode/Issue		Execute			Writeback/complete		Cykel
		FP	ADD	MUL			
I1	I2						1
I3	I4	I1	I2				2
I5	I6	I1				STALL	3
				I3	I1	I2	4
				I4	I3		5
			I5		I4		6
			I6		I5		7
					I6		8



# In-order issue med out-of-order completion

I1: ADDF R12, R13, R14  
 I2: ADD R1, R8, R9  
 I3: MUL R4, R2, R3  
 I4: MUL **R5**, R6, R7  
 I5: ADD R10, **R5**, R7  
 I6: ADD R11, R2, R3

$R12 \leftarrow R13 + R14$  (flyttalsberäkning)  
 $R1 \leftarrow R8 + R9$   
 $R4 \leftarrow R2 * R3$   
 $R5 \leftarrow R6 * R7$   
 $R10 \leftarrow R5 + R7$   
 $R11 \leftarrow R2 + R3$

Decode/Issue		Execute			Writeback/complete		Cykel
		FP	ADD	MUL			
I1	I2						1
I3	I4	I1	I2				2
I5	I6	I1		I3	I2		3
				I4	I1	I3	4
			I5		I4		5
			I6		I5		6
					I6		7
							8

Out-of-order completion



# Out-of-Order Issue (OOI) med Out-of-Order Completion (OOC)

I1: ADDF R12, R13, R14  
 I2: ADD R1, R8, R9  
 I3: MUL R4, R2, R3  
 I4: MUL R5, R6, R7  
 I5: ADD R10, R5, R7  
 I6: ADD R11, R2, R3

$R12 \leftarrow R13 + R14$  (flyttalsberäkning)  
 $R1 \leftarrow R8 + R9$   
 $R4 \leftarrow R2 * R3$   
 $R5 \leftarrow R6 * R7$   
 $R10 \leftarrow R5 + R7$   
 $R11 \leftarrow R2 + R3$

I6 kan starta före I5 och parallellt med I4. Programmet tar bara 6 cykler (jämfört med 8 i fallet med in-order issue and in-order completion)

Decode/Issue		Execute			Writeback/complete		Cykel
		FP	ADD	MUL			
I1	I2						1
I3	I4	I1	I2				2
I5	I6	I1		I3	I2		3
			I6	I4	I1	I3	4
			I5		I4	I6	5
					I5		6
							7
							8



# Konflikthantering

---

- Med in-order issue med in-order-completion behöver processorn bara hantera *true data dependency*, men inte *output dependency* och *anti-dependency*.
- Med out-of-order issue med out-of-order completion behöver processorn hantera *true data dependency*, *output dependency* och *anti-dependency*

– Output dependency

MUL R4, R3, R1	R4 ← R3 * R1
----	
ADD R4, R2, R5	R4 ← R2 + R5

– Antidependency

MUL R4, R3, R1	R4 ← R3 * R1
----	
ADD R3, R2, R5	R3 ← R2 + R5



# Sammanfattning

---

- Experiment har visat:
  - Att enbart lägga till extra enheter (typ pipelines) är inte effektivt;
  - Out-of-order är väldigt viktigt; det tillåter look ahead för “independent” instruktioner;
  - Register renaming kan förbättra prestanda med mer än 30%; I detta fall blir prestanda bara begränsat av true dependencies.
  - Viktigt med fetch/decode där instruktionsfönstret är tillräckligt stort



# Översikt

---

- Parallellism
- Parallellism på instruktions-nivå
  - Superscalar processors
  - Very Long Instruction Word processors
- Parallellism på programs tråd-nivå





# Superscalar processorer

---

- Bra:
  - Hårdvaran löser allt:
    - » Hårdvara detekterar potentiell parallellism av instruktioner
    - » Hårdvara försöker starta exekvering (issue) av så många instruktioner som möjligt parallellt
    - » Hårdvara löser register renaming
  - Kompatibla binärer:
    - » Om en ny funktionell enhet läggs till i en ny version av arkitekturen eller några andra förbättringar göra (utan att ändra instruktionsuppsättningen) så kommer äldre program (kompilerade för äldre arkitektur) att kunna dra nytta av förbättringar eftersom den nya hårdvaran kan exekverar det gamla programmet (dess ordning av instruktioner) mer effektivt.



# Superscalar processorer

---

- Problematiskt:
  - Väckigt komplext:
    - » Behövs mycket hårdvara för "run-time" detektering av möjlig parallellism
    - » Effektförbrukningen kan bli hög, pga mycket hårdvara
  - Begränsat instruktionsfönster:
    - » Det gör att det blir svårare att hitta instruktioner som har möjlighet att exekveras parallellt.



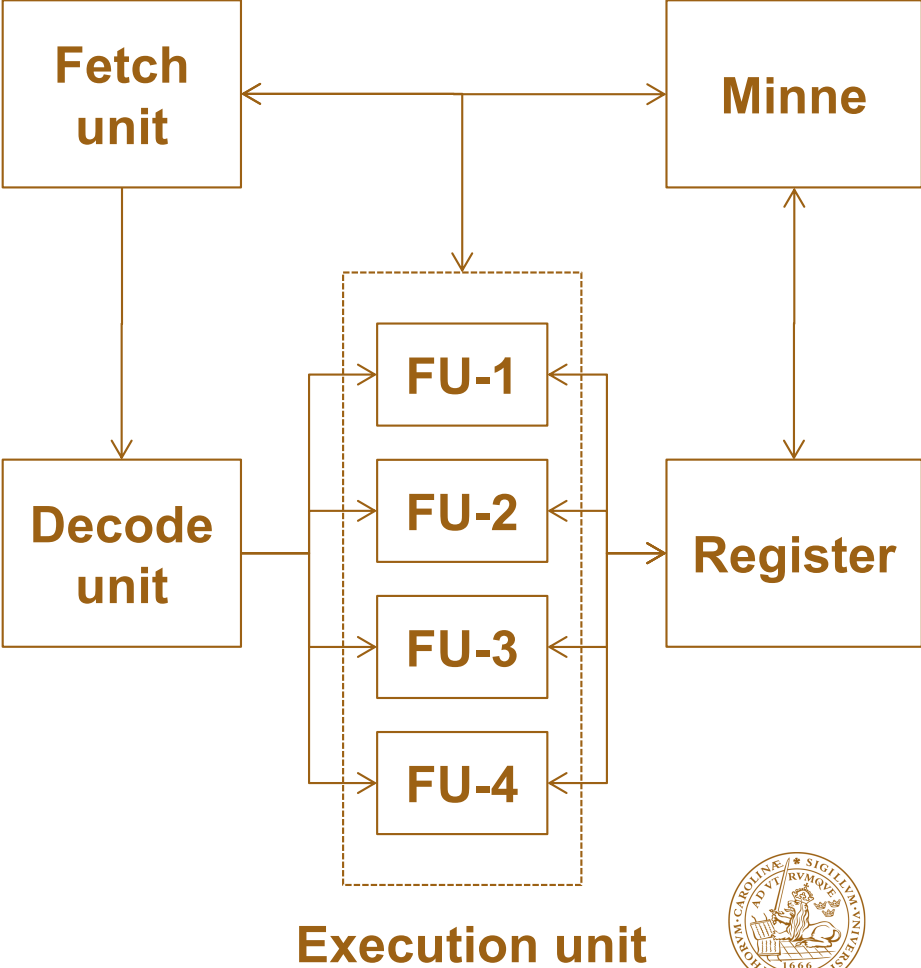
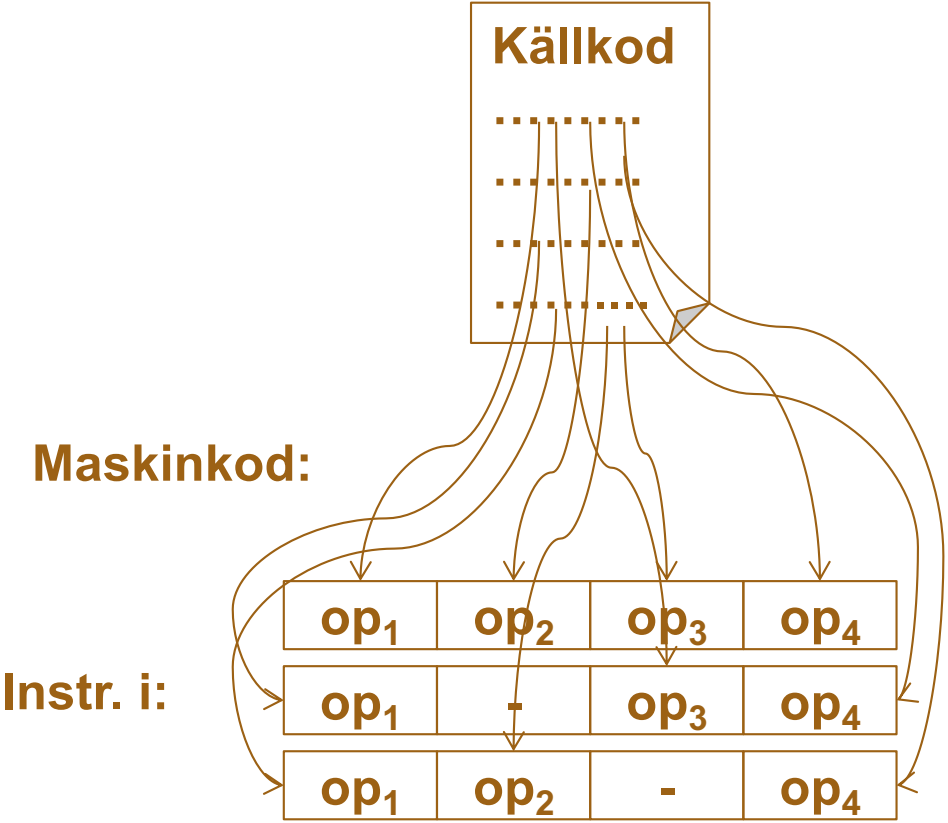
# VLIW processorer

---

- VLIW processorer använder kompilatorn för detektering av parallellism
  - Kompilatorn analyserar och detekterar operationer som kan exekveras parallellt. Dessa operationer packas in i en "large" instruktion
- När en "long" instruktion hämtats, exekveras alla dess operationer parallellt
- Ingen hårdvara behövs för "run time" detektering av parallellism
- Problemet med litet instruktionsfönster är löst; kompilatorn kan vid kompilering använda hela programmet för detektera parallellism.



# VLIW processorer



# VLIW processorer

---

- Fördelar:
  - Enklare hårdvara.
    - » Antalet FU kan ökas utan behov av mer och sofistikerad hårdvara för att detektera parallellism
    - » Effektförbrukning kan reduceras
  - Kompilatorn kan detektera parallellism genom en global analys av hela programmet (inget problem med begränsat instruktions fönster)



# VLIW processorer

---

- Nackdelar:
  - Många register behövs för att hålla alla FU aktiva (lagring av operander och resultat)
  - Stor bandbredd behövs mellan:
    - » FU och register
    - » Register och minne
    - » Instruktions cache och fetch unit (exempel: en instruktion med 7 operationer där varje operation behöver 24, krävs 168 bitar per instruktion)



# VLIW processorer

---

- Nackdelar:
  - Stora program (många instruktioner) pga icke använda operationer i instruktioner
  - Binär (exekverbar kod) är inte kompatibel. Om en ny processor har fler FU så kan parallellism öka (fler operationer). Problemet är att instruktionerna ändras. Gammal kod kan inte exekvera på den nya processorn. Och om den kan det så utnyttjas inte alla FUs.



# Översikt

---

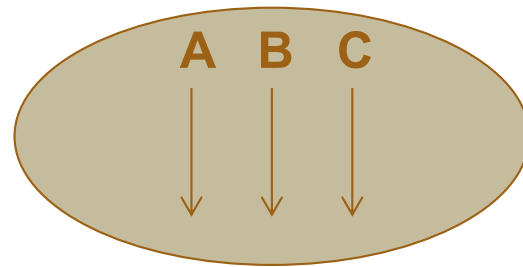
- Parallellism
- Parallellism på instruktions-nivå
  - Superscalar processors
  - Very Long Instruction Word processors
- Parallellism på programs tråd-nivå





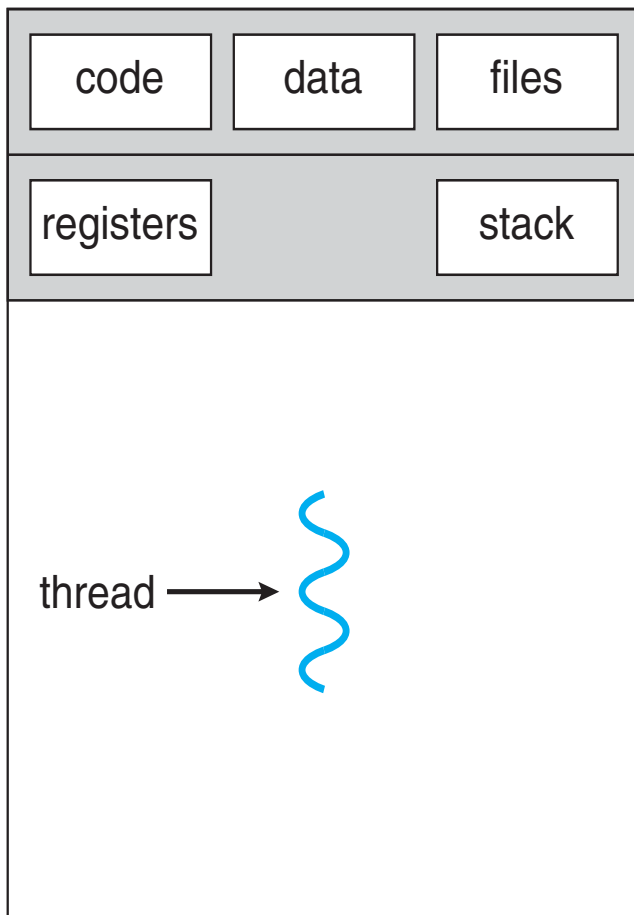
# Processer och trådar

- En process består av en eller flera trådar (threads) där en tråd är sekventiell kod som kan exekvera parallellt med andra trådar

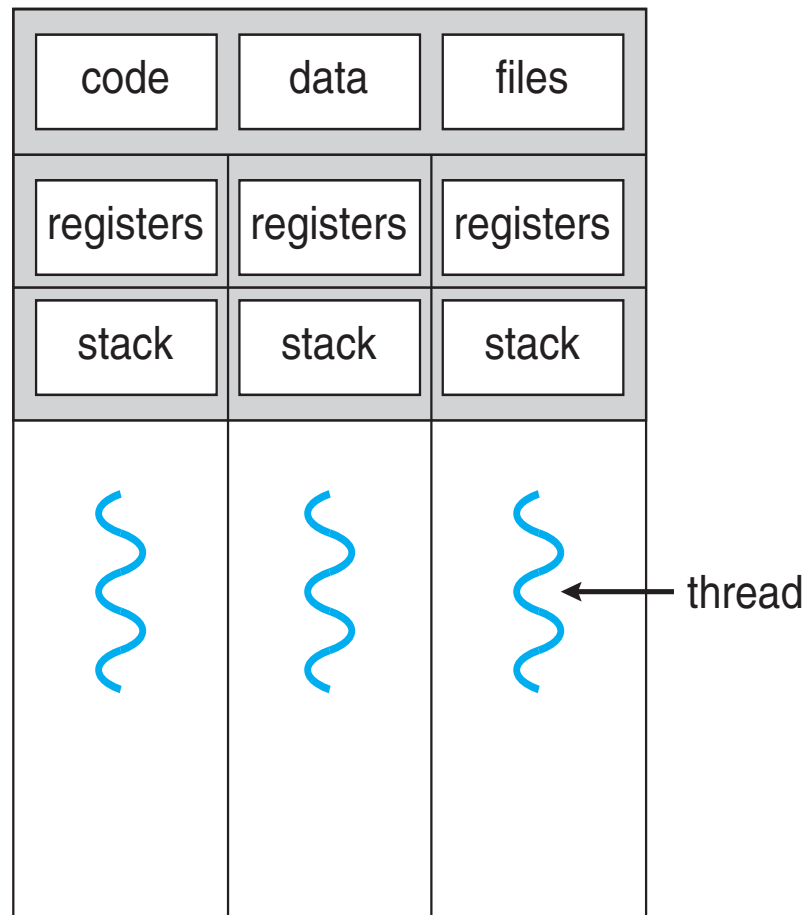


- Alla trådar i en process delar data och stack, vilket gör att byte av tråd är mindre kostsamt än byte av process
- Hårdvarustöd:
  - Programräknare och register per tråd
  - Instruktionshämtning (fetch) på trådbasis
  - Kontextbyte (byte av tråd)
- Effektiv exekvering av program med flera trådar (multithreading)
- Effektivt utnyttjande av processorns resurser

# Single and Multithreaded Processes



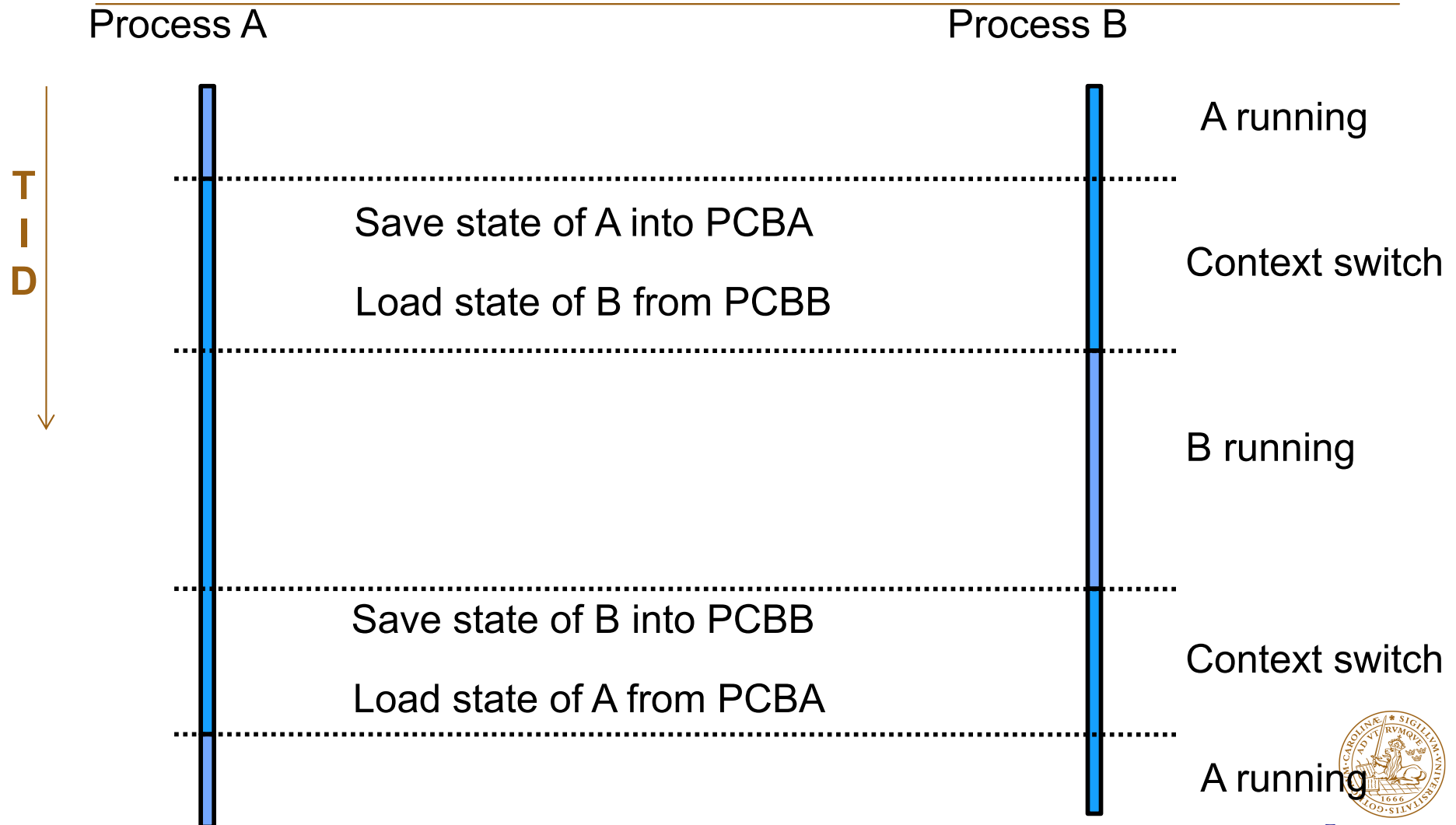
single-threaded process



multithreaded process



# Kontextbyte (context switch)



# Processer och trådar

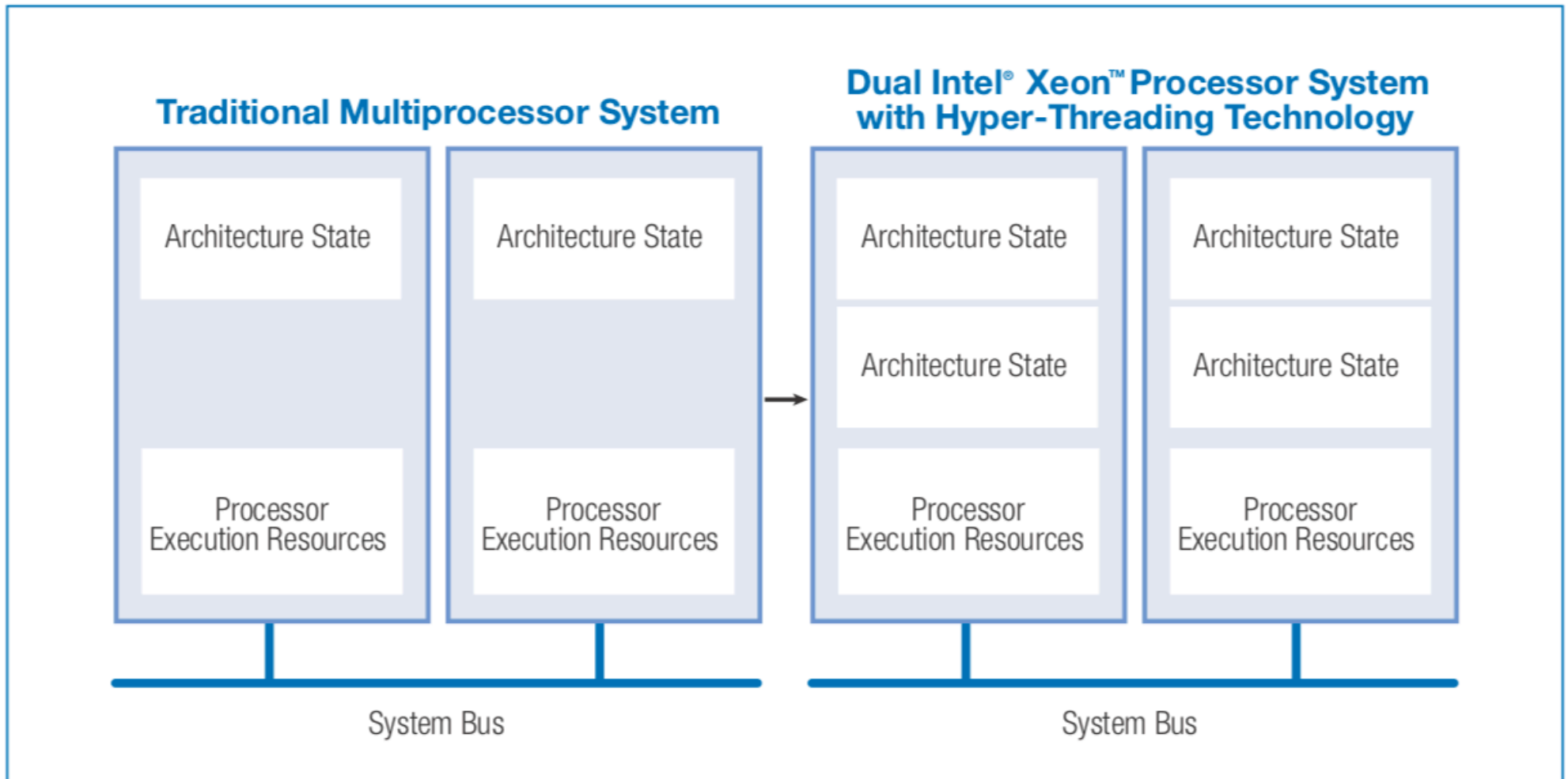
---

- Flera trådar (multithreading)
  - Större chans att kunna exekvera instruktioner parallellt
  - När en tråd blockeras (wait) på grund av till exempel minnesaccess så kan instruktioner från en annan tråd exekvera. Fördelen är att fördröjning på grund av minnesaccess (t ex cache miss), databeroende kan gömmas
- Multithreading kan användas på arkitekturer som är såväl skalära (scalar) och superskalära (superscalar)
- Multithreading:
  - Interleaved multithreading
  - Blocked multithreading
  - Simultaneous multithreading (SMT)



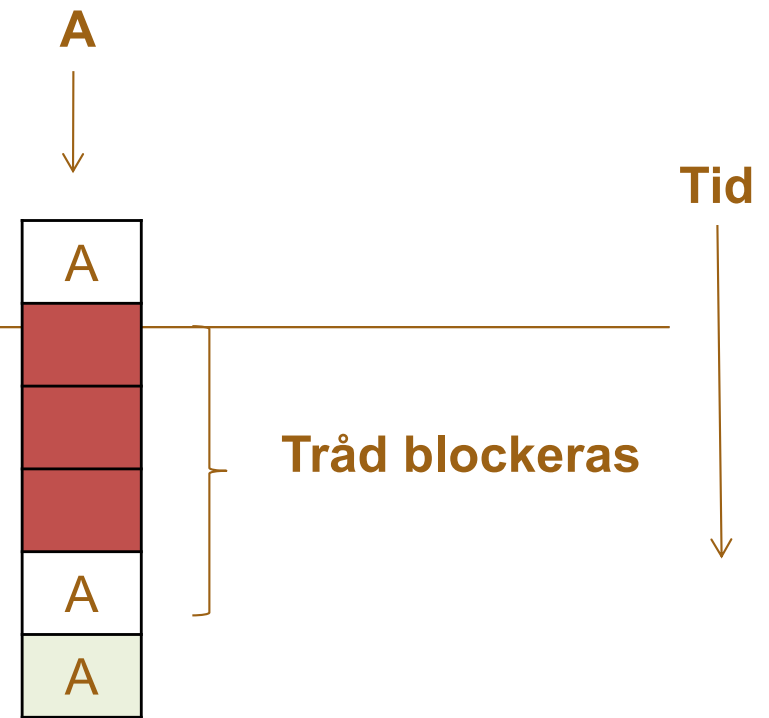
# Hårdvarutrådar-logisk/fysisk CPU/core

---

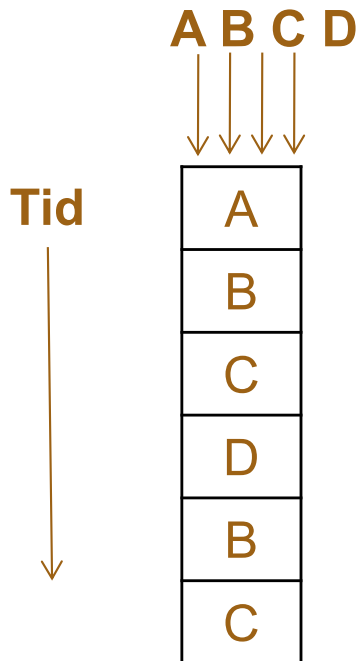


# Multithreading

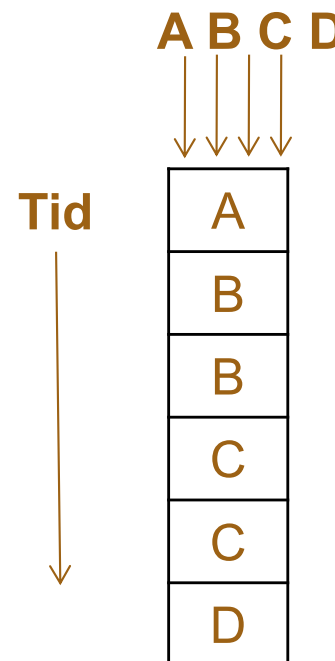
- Scalar (icke-superscalar) processor
  - En tråd och en beräkningsenhet (pipeline)



- Interleaved multithreading

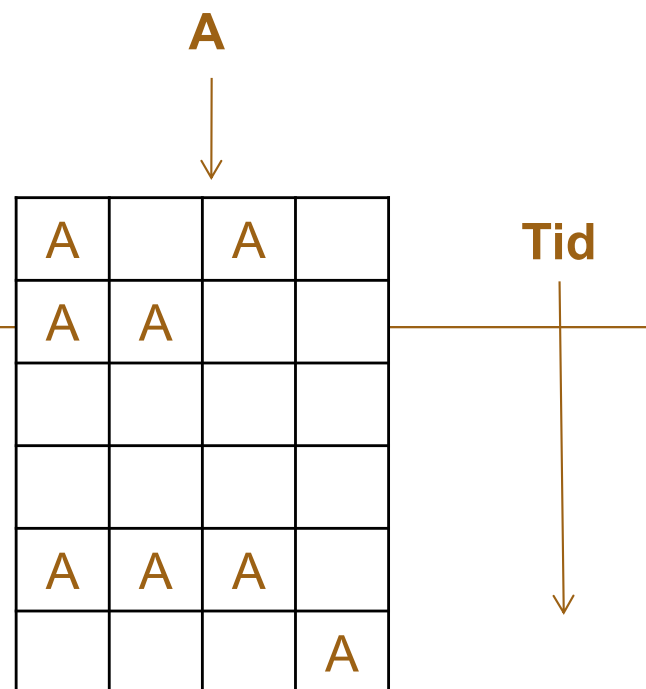


- Blocked multithreading

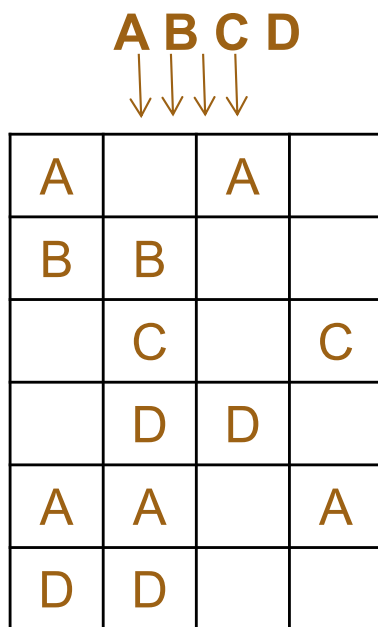


# Multithreading

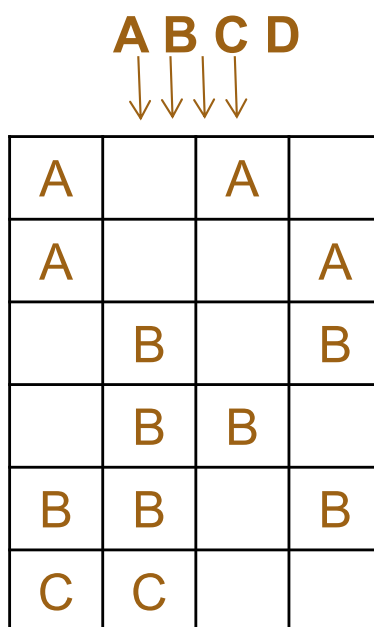
- Superscalar processor
  - En tråd och flera beräkningsenheter (pipelines)



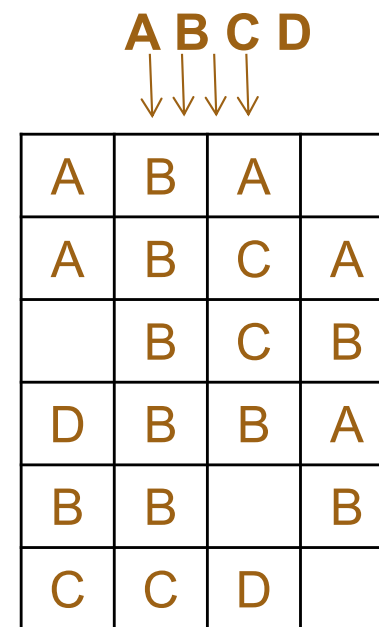
- Interleaved multithreading



- Blocked multithreading



- Simultaneous multithreading (SMT)



# Sammanfattning

---

- Högre prestanda kan inte längre fås med högre klockfrekvens
- I parallella datorer samarbetar flera processorer för att lösa en uppgift
- Parallella program underlättar exekvering i en parallell arkitektur
- Datorer kan klassas som SISD, SIMD och MIMD
- Prestandan beror inte bara på antal beräkningsenheter, utan också hur “parallellt” ett program är
- Multithreading processorer ger via hårdvarustöd möjlighet att exekvera flera trådar





# Sammanfattning

---

- Vem ska hantera parallellism?
- Parallelism
  - Programnivå: Program skrivna så att de kan exekvera parallellt, t ex många trådar
  - Operativsystemsnivå: Program/trådar fördelas över cores/CPU:er
  - Hårdvarunivå: Instruktionsnivåparallellism



# Framtiden

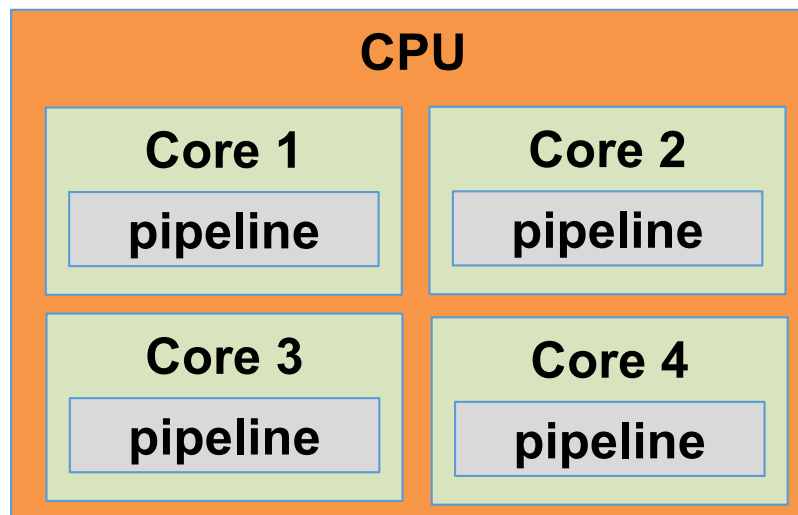
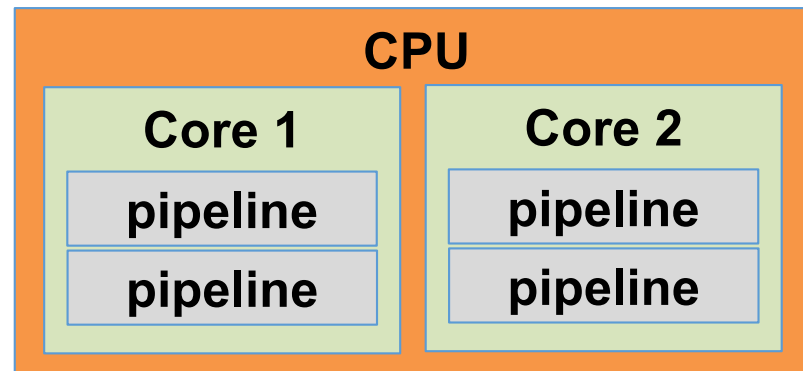
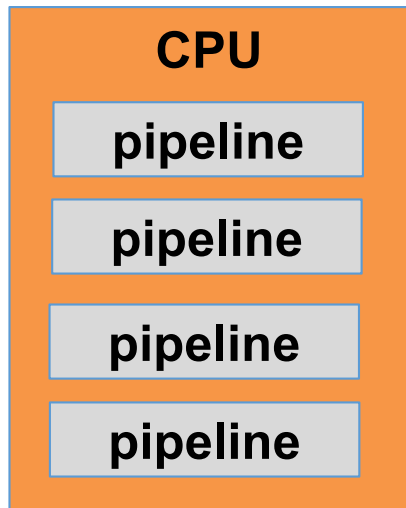
---

- Moores lag
  - 2015: SPARC M7 – 10 miljarder
  - 2017: AMD EPYC – 19 miljarder
- Transistorer blir mindre
  - 2015: SPARC M7 - 20nm
  - 2017: AMD EPYC – 14 nm
  - 2018: AMD EPYC2 – 7nm

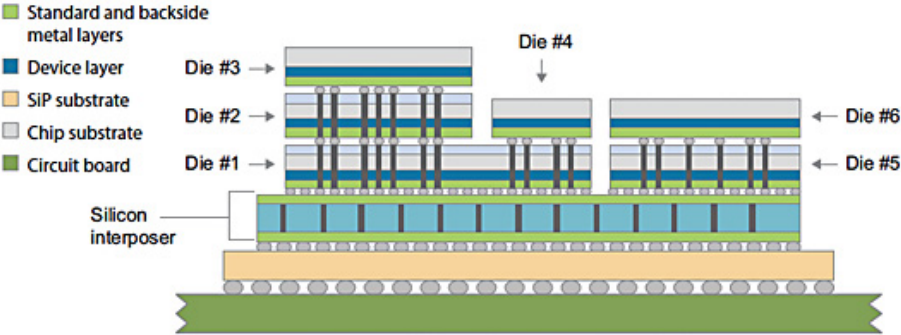
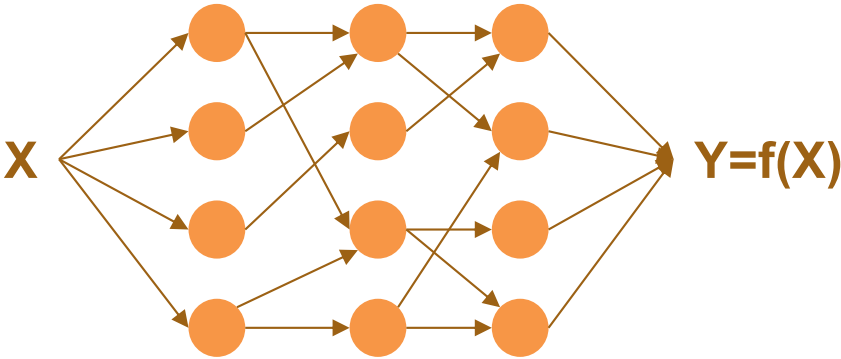
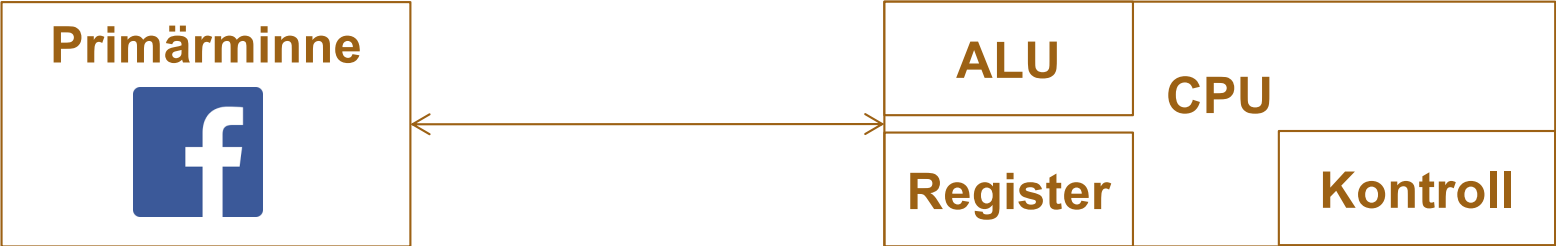
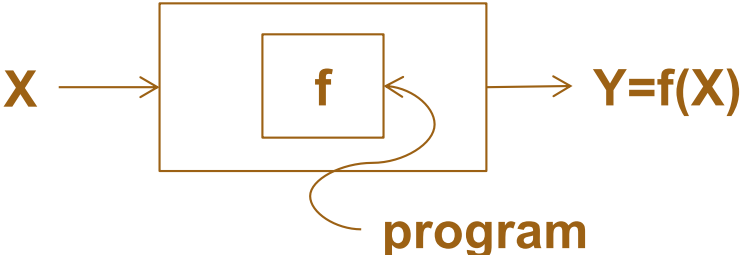


# Framtiden

---



# CISC->RISC->Zero Instruction Set Computers



# Några fortsättningskurser

---

IC-project (ETIN35, ETIN40)

Konstruktion av  
inbyggdasystem -  
EDAN15

DSP-design –  
ETIN45

Minnesteknologi för  
maskininlärning –  
ETIP25

Operating Systems  
– EDAF35

Datorarkitektur  
- EITF20

Digitala  
strukturer på  
kisel – EITF35

Concurrent  
Programming–  
EDAF35

Digital IC-  
konstruktion -  
ETIN20

Kompilatorer  
– EDAN65

Flertrådad  
programmering  
– EDAN26

Digital  
signalbehandling  
– EITF75



# Uppsamlingstillfälle

---

- Jag har missat en (1) dugga
- Jag har inte hunnit redovisa laborationer
- Håll koll på meddelandesidan för instruktioner





**LUNDS**  
**UNIVERSITET**