

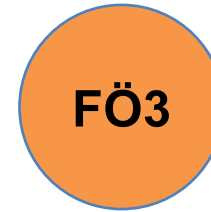
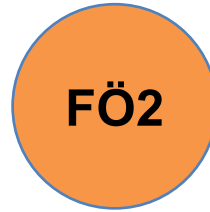
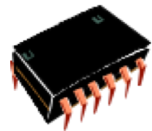
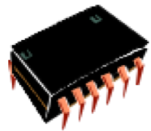
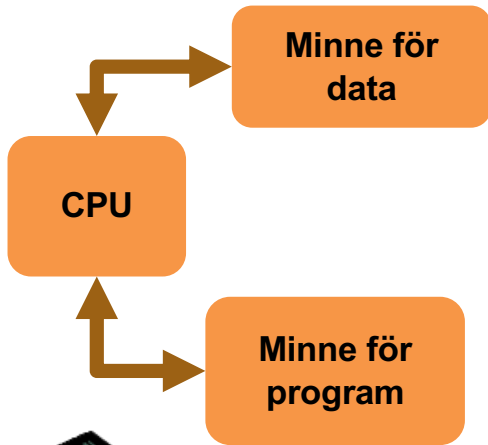
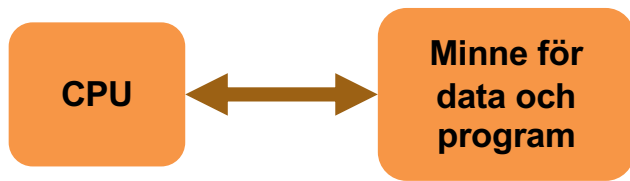


LUNDS
UNIVERSITET

Datorsteknik

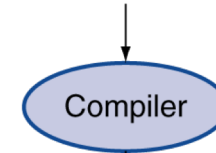
ERIK LARSSON





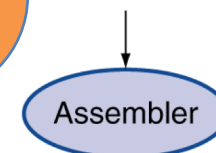
High-level language program (in C)

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



Assembly language program (for MIPS)

```
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
```

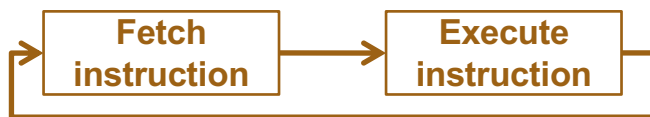
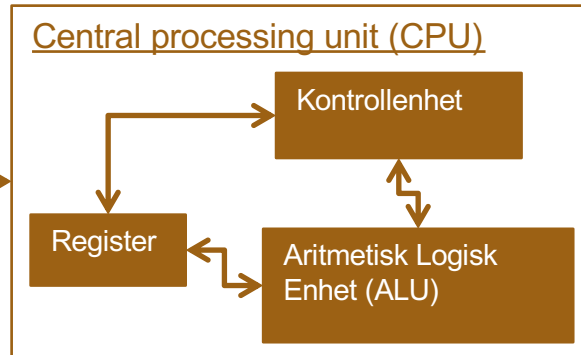


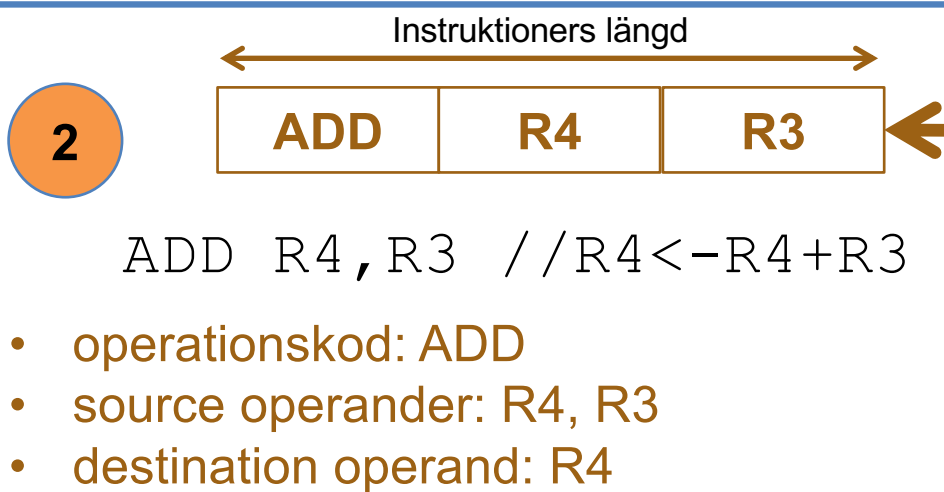
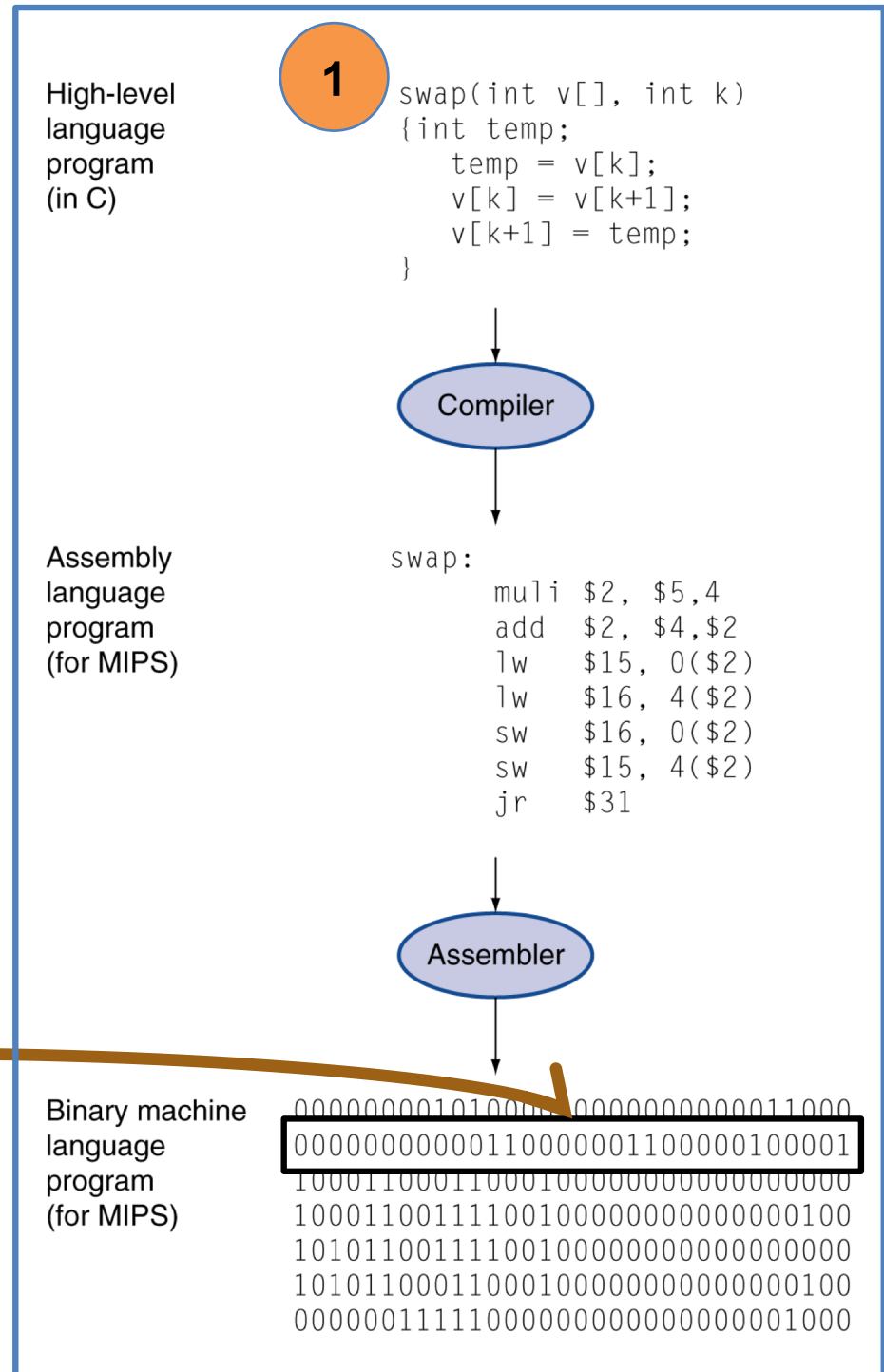
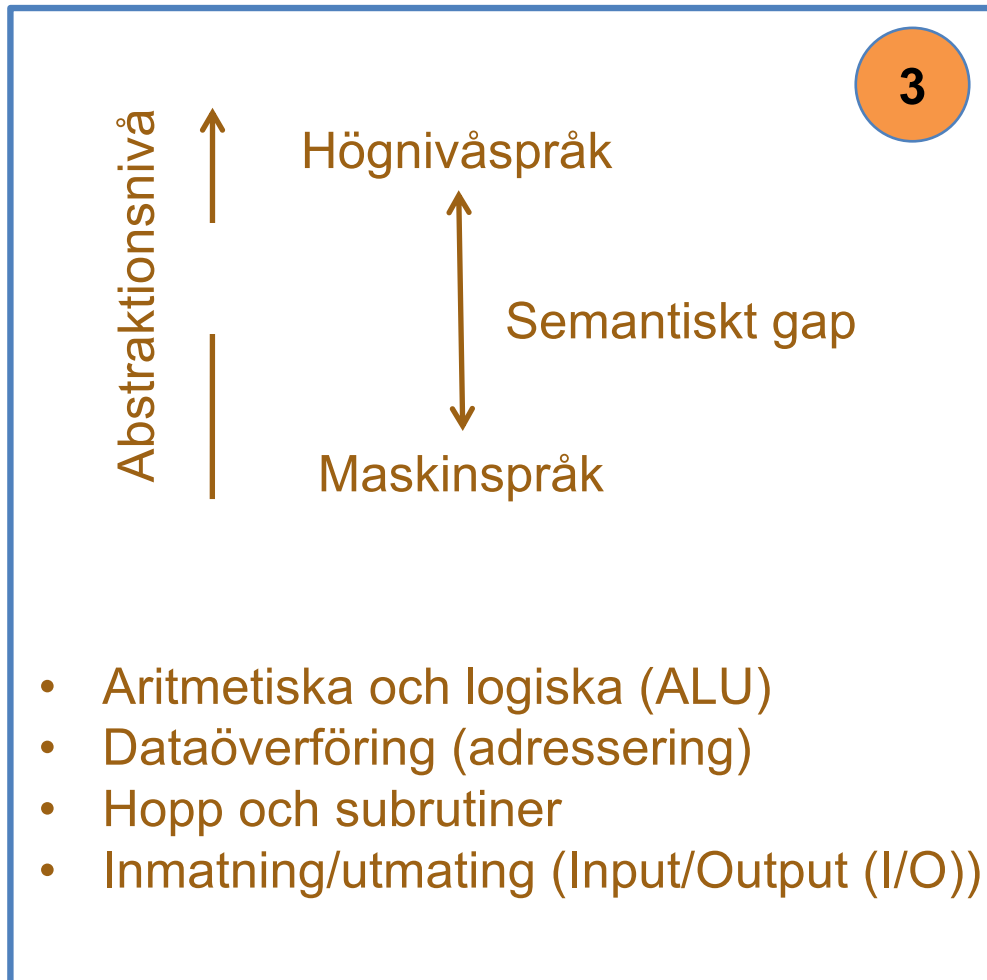
Binary machine language program (for MIPS)

```
000000001010000100000000000011000
000000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

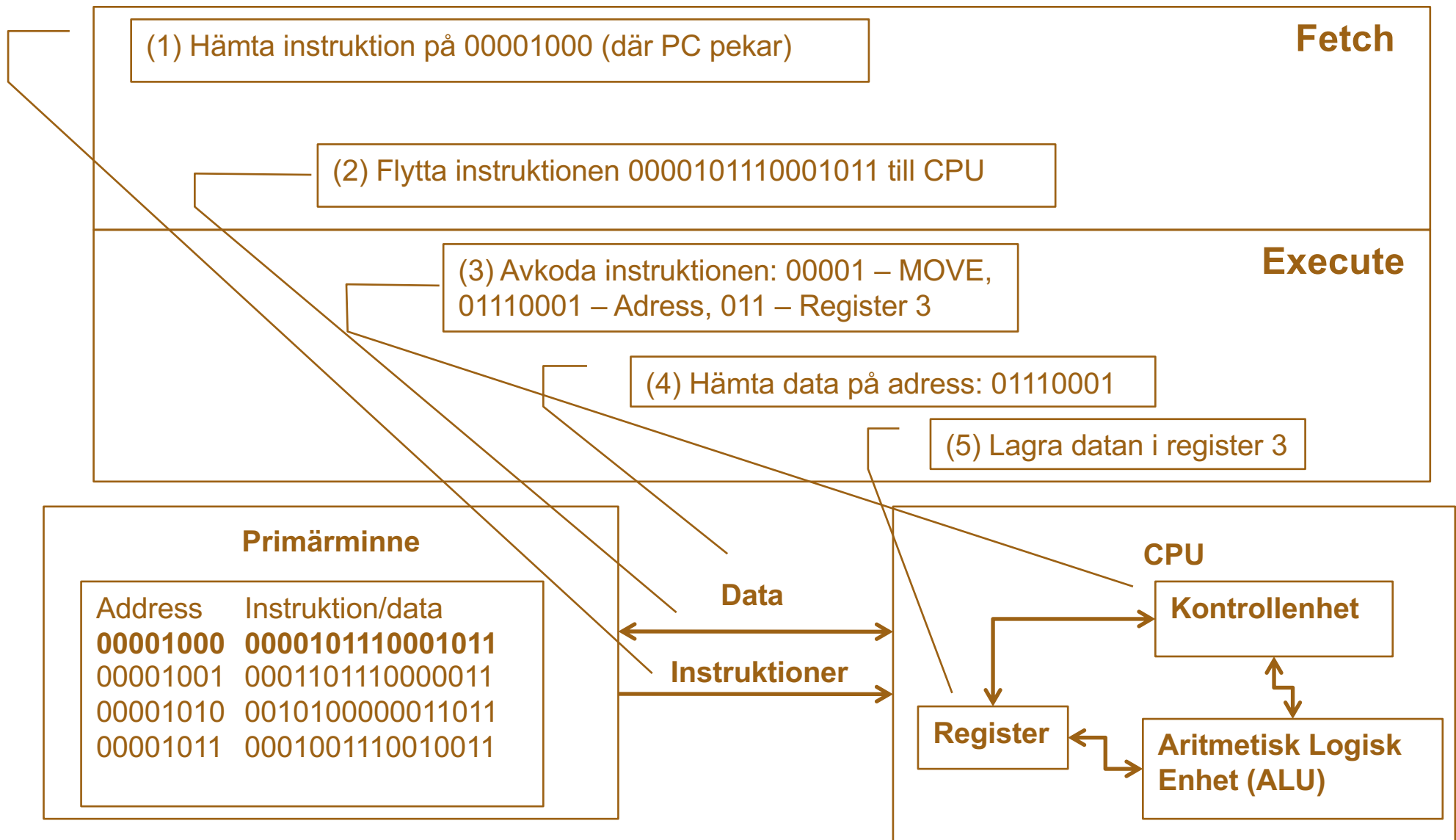
Primärminne
Instruktioner och data

Address	Instruction
00001000	0000101110001011
00001001	0001101110000011
00001010	0010100000011011
00001011	0001001110010011

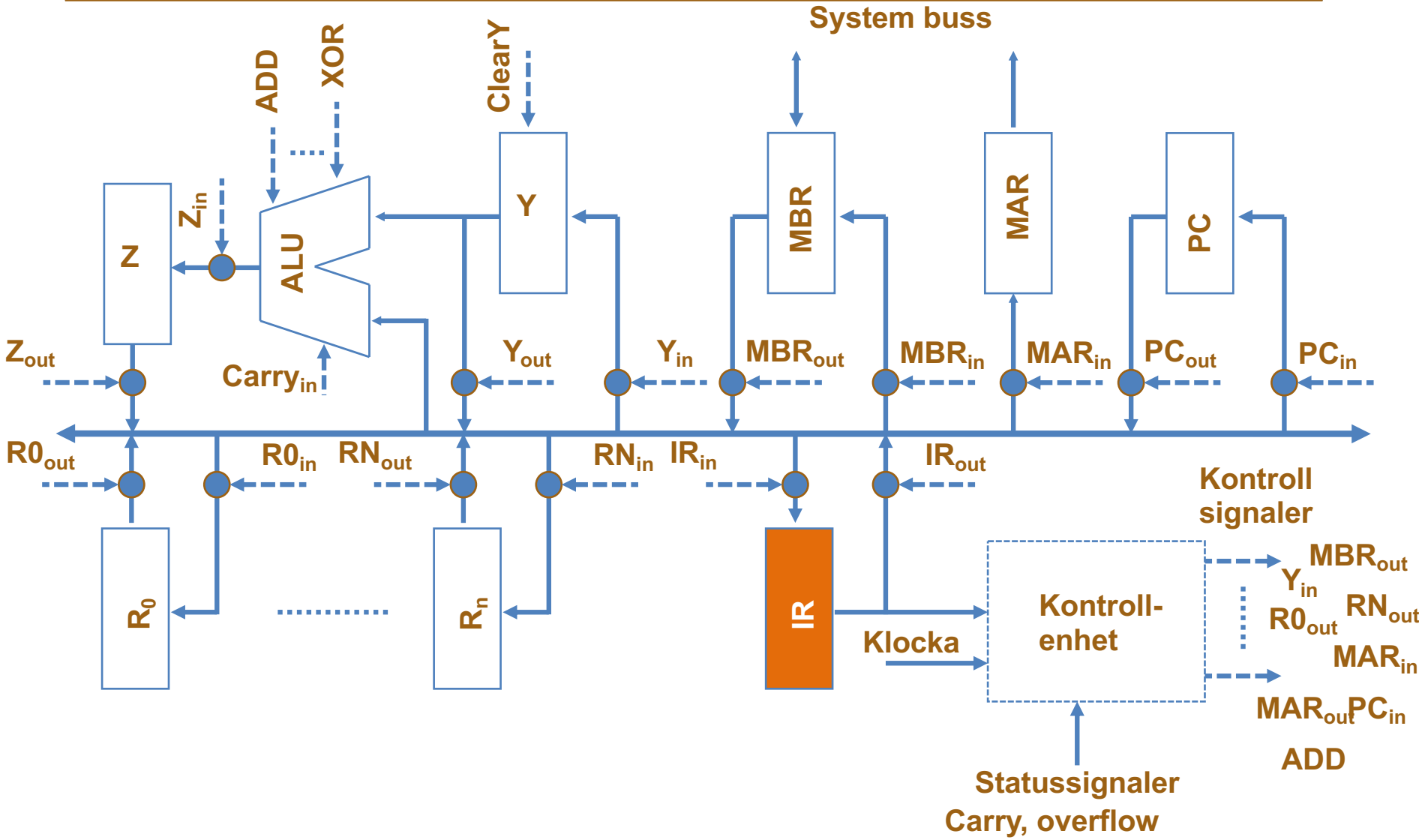




Exekvering av en instruktion



Kontrollenhet



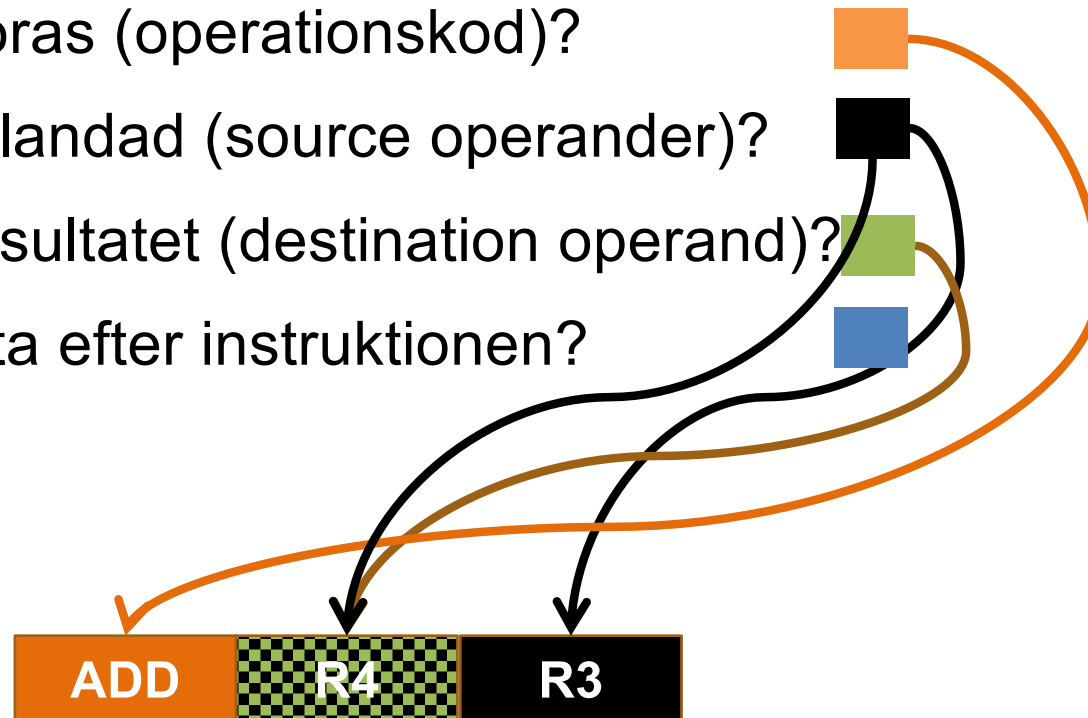
Exekveringstid

- Instruktion:
 - ADD R1, R3 // R1 \leftarrow R1 + R3
- Kontrollsteg = Klockcykler per instruktion (CPI)
 1. PC_{out}, MAR_{in}, Read, Clear Y, Carry-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



Maskininstruktioner

- Definitioner:
 - Vad ska göras (operationskod)?
 - Vem är inblandad (source operander)?
 - Vart ska resultatet (destination operand)?
 - Hur fortsätta efter instruktionen?



Maskininstruktioner

- Att bestämma:
 - Typ av instruktioner (operander och operationer)
 - Antal adresser
 - Instruktionsformat
 - » Fixed eller flexibelt



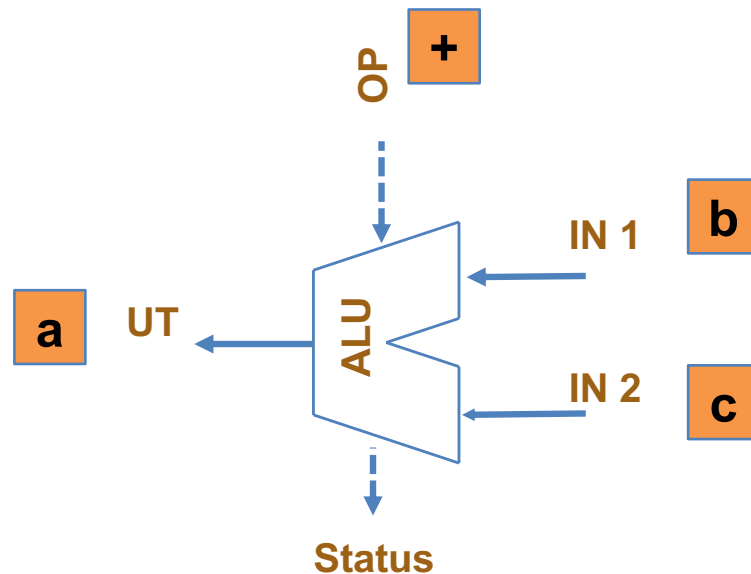
Maskininstruktioner

- Typer av instruktioner:
 - Aritmetiska och logiska (ALU)
 - Dataöverföring(adresseringsformat)
 - Hopp
 - In- och utmatning



Aritmetiska operationer

- Addition (+) och subtraktion (-)
 - Två källor (sources) och en destination (destination)
`add a, b, c // a = b + c`
- Alla aritmetiska funktioner följer detta mönster



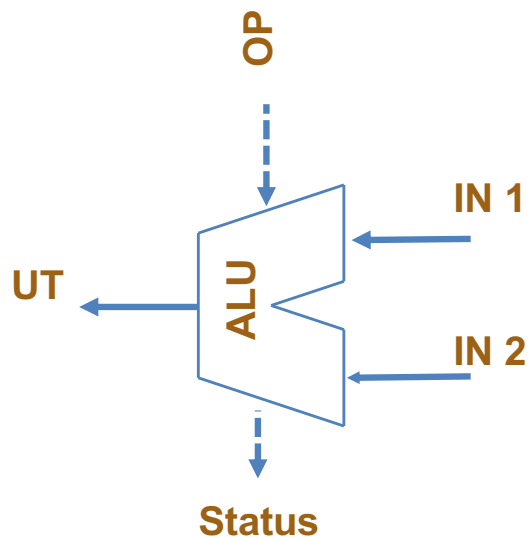
Aritmetiska operationer

- C kod:

```
a=b+c+d;
```

- Kompileras till:

```
ADD a, b, c //adderar b och c, resultat i a  
ADD a, a, d //adderar a (som är b+c) med d,  
             resultat i a
```



Maskininstruktioner

- Aritmetiska och logiska (ALU)
- Dataöverföring (adressering)
 - Flytta data från ett ställe till ett annat
- Hopp och subrutiner
- Inmatning/utmating (Input/Output (I/O))



Adressering

- Immediate adressering

ADD R4, #3 // R4 ← R4 + 3

Operanden finns direkt i instruktionen

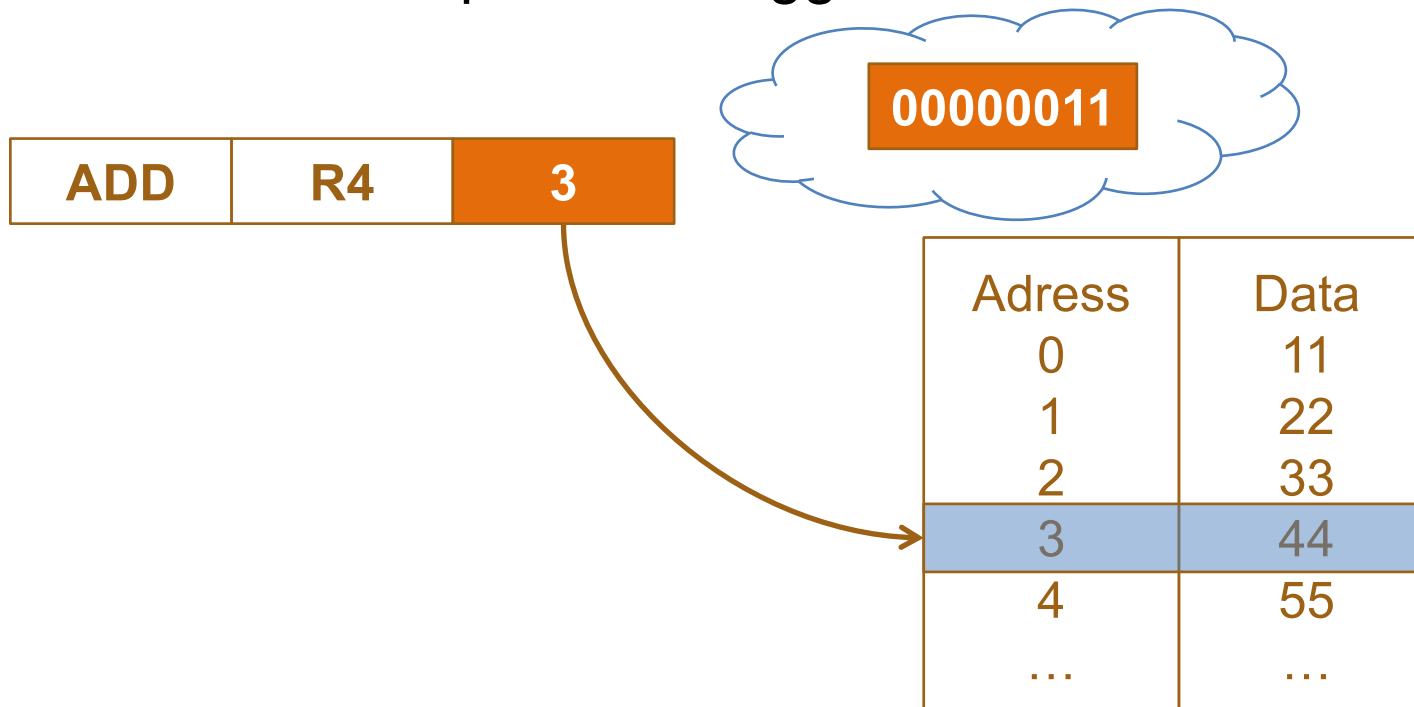


Adressering

- Direkt adressering

ADD R4, **3** //R4←R4+[**3**]

Adressen till operanden ligger i instruktionen



Adressering

- Register adressering

ADD R4, R3 //R4←R4+R3

Liknar direkt adressering men istället för att peka ut i minnet så pekas ett register ut



Register	Data
0	11
1	22
2	33
3	44
4	55
...	...

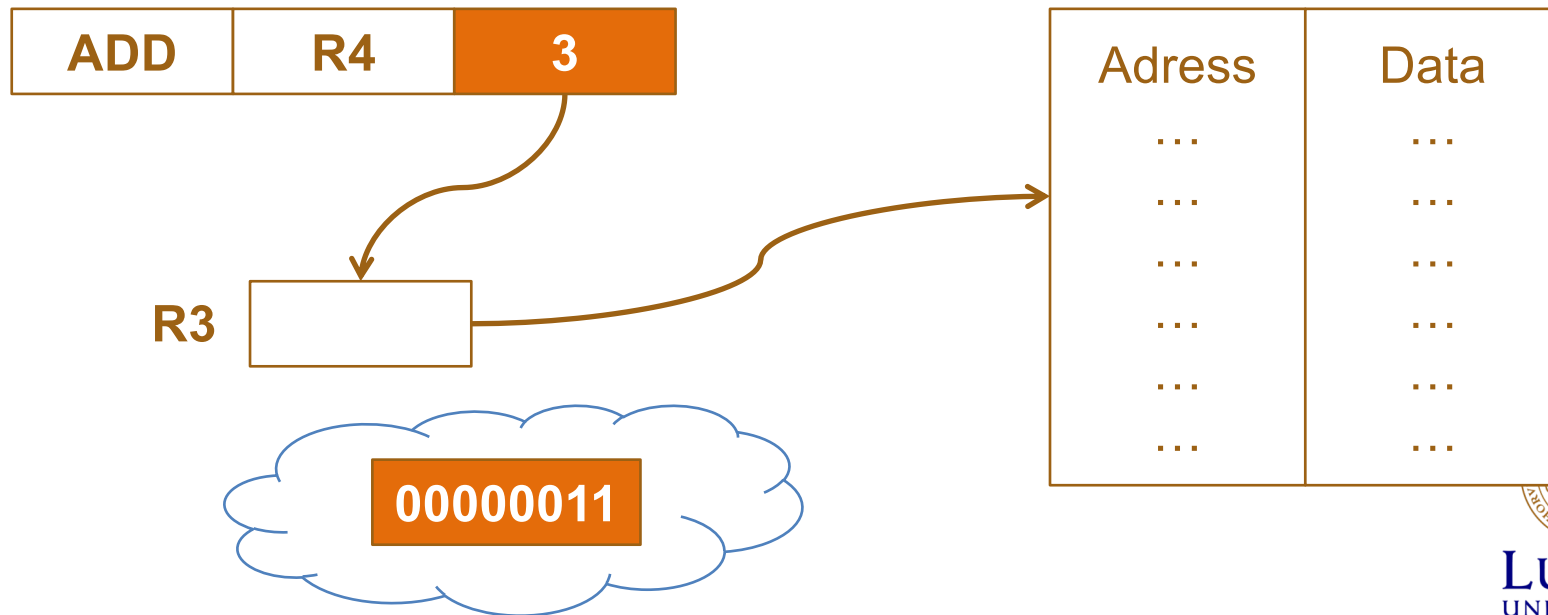


Adressering

- Register indirect

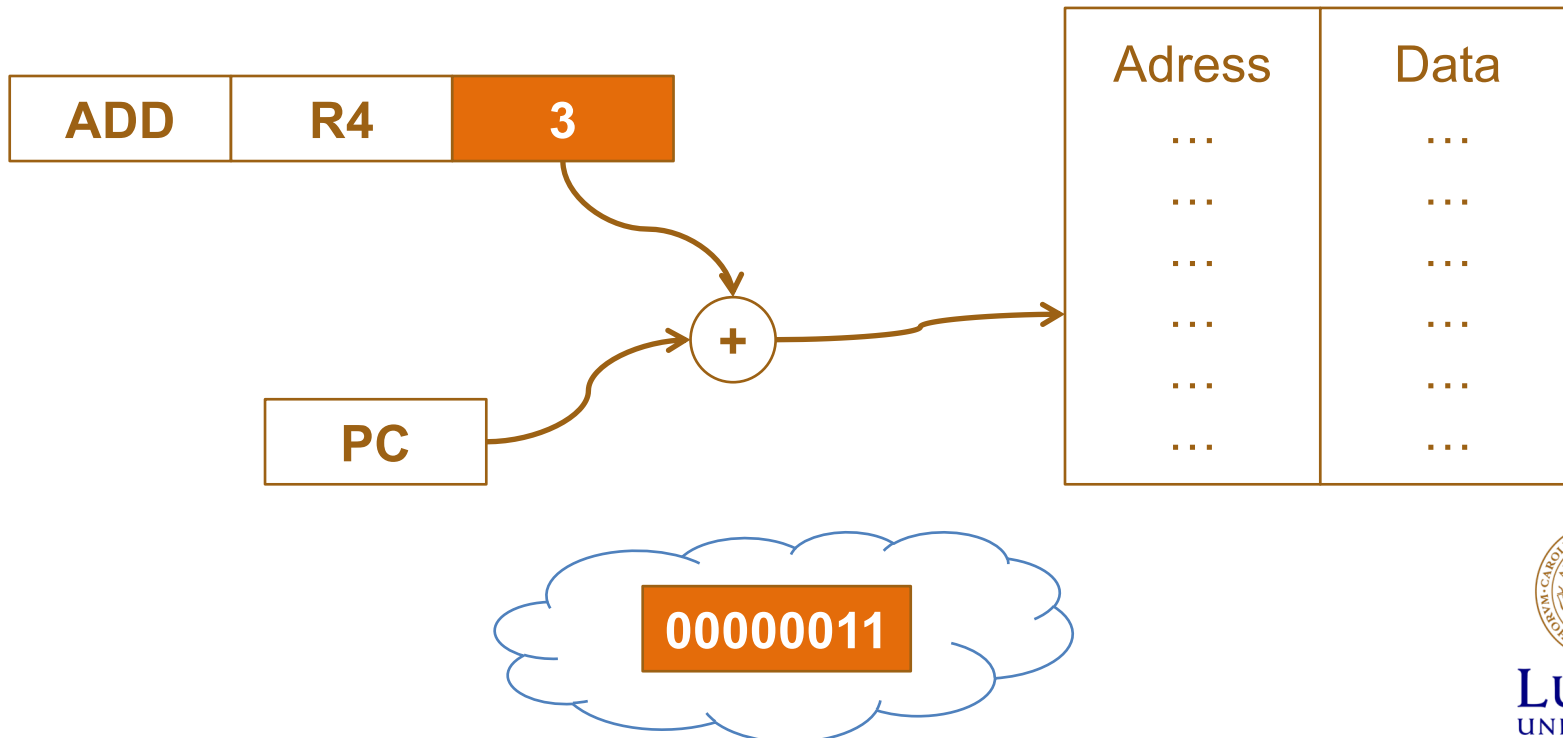
ADD R4, (R3) $R4 \leftarrow R4 + [R3]$

Liknar indirekt adressering men instruktion pekar ut register istället för minnesplats



Adressering

- Relativ adressering
 - Använd t ex PC för att skapa adress. Enbart liten bit finns i instruktionen



Varför olika sätt att adressera?

- Öka flexibiliteten vid programmering, vilken förenklar programmering
- Antal bitar för att specificera en adress är begränsat.

– Exempel: Om 8-bitar används:



kan 2^8 adresser pekats ut.

Minne



Extra
tid (jmf
ovan)

– Men, om 8-bitar används att peka ut ett register som innehåller adress kan 2^{16} adresser pekats ut

Minne



R1 Adress (16-bitar)



LUNDS
UNIVERSITET

Adressering: avvägningar

- Få och enkla adresseringsmöjligheter
 - Stora adressfält i instruktion för att kunna täcka stor adressrymd
 - Begränsar programmering
 - Snabbare adressberäkning
 - Mindre komplex processor
- Många och avancerade adresseringsmöjligheter
 - Tillåter mindre adressfält utan att påverka adressrymd
 - Flexibel programmering
 - Kan ge långsam adressberäkning
 - Ökar komplexitet av processorn



Maskininstruktioner

- Aritmetiska och logiska (ALU)
- Dataöverföring (adressering)
- Hopp och subrutiner
- Inmatning/utmating (Input/Output (I/O))



Hoppinstruktioner

Exempel 1

```
a=a+b;  
c=a-b;  
d=a*f;
```

**Instruktionen
här görs flera
gångar**

**Instruktionerna
utförs i ordning**

Exempel 2

```
a=0;  
while (a<10)  
    a=a+1;  
if (b==5)  
    c=10;  
else  
    c=0;  
d=a*f;
```

**Beroende på
villkor görs olika
saker (olika
instruktioner
exekveras)**



Hoppinstruktion

- Nästa instruktion är normalt nästa i programmet.

`PC=PC+1;`

Programräknaren (program counter) räknas upp och nästa instruktion hämtas

- För att göra hopp till annan del av kod, ladda programräknaren med nytt värde

`PC=dit hopp ska sek`

- Två typer av hopp
 - Ovillkorliga hopp
 - Villkorliga hopp



Ovillkorliga hopp

- Exempel:

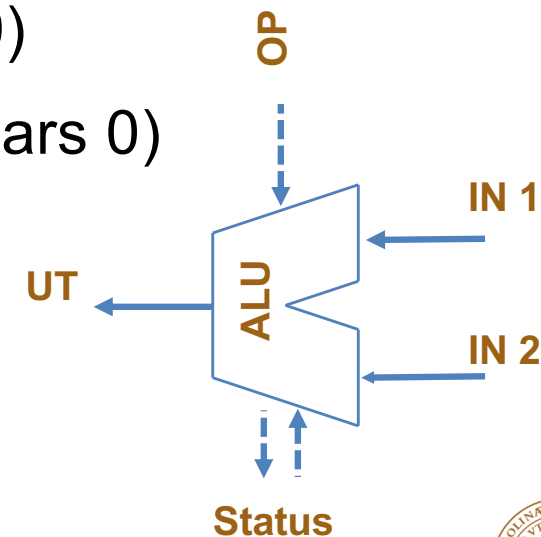
Adress	Instruktion	Kommentar
.....	
01011	Instruktion 29	//Instruktion 29, PC=PC+1
01100	BR 10101	// PC = 10101 , PC=PC+1
01101	Instruktion 31	//Instruktion 31, PC=PC+1
.....	
10101	Instruktion 89	//Instruktion 89, PC=PC+1
.....	

Programräknaren får nytt värde och hämtar nästa instruktion på nytt ställe



Villkorliga hopp

- Villkor bestäms av flaggor i statusregister
- De vanligaste flaggorna är:
 - N: 1 om resultatet är negativt (annars 0)
 - Z: 1 om resultatet är noll (annars 0)
 - V: 1 om aritmetiskt “overflow” (annars 0)
 - C: 1 om “carry” (annars 0)



Villkorliga hopp

- Exempel:

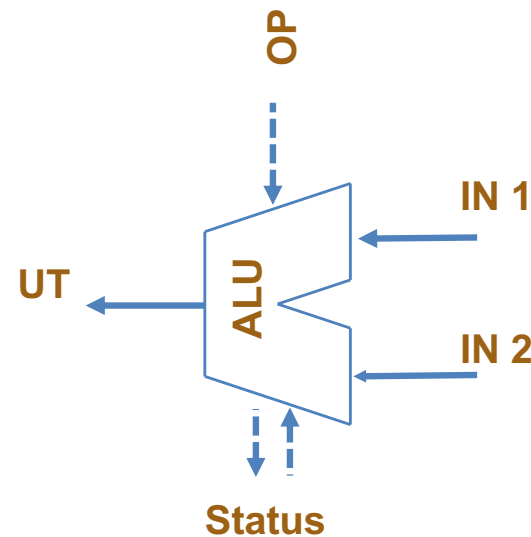
```
SUB R1, #1      // R1=R1-1
BEZ TARGET     // IF Z=1 THEN PC=TARGET
```

Kan påverka statusflagga

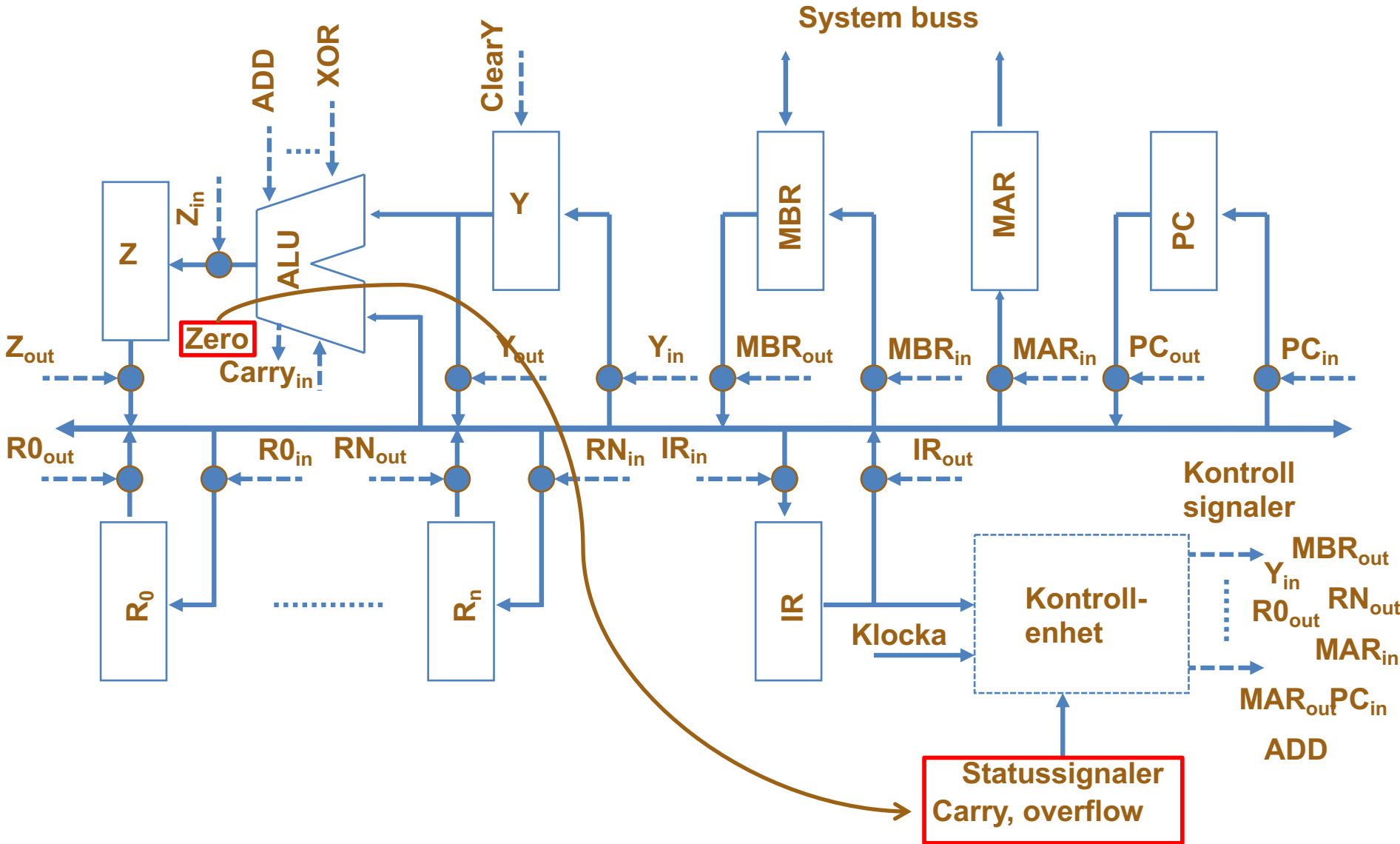
Kollar statusflagga



Statusregister



Villkorliga hopp



Subrutiner och funktioner

Program:

```
void main(void) {  
  int a, b, c;  
  a=5;  
  b=6;  
  c=my_add(a,b);  
}  
  
int my_add(int x, int y)  
{  
  return x+y;  
}
```

Exekvering:

```
a=5    //instruktion 1  
b=6    //instruktion 2
```

funktionsanrop+
parameteröverföring

```
c=my_add(a,b);
```

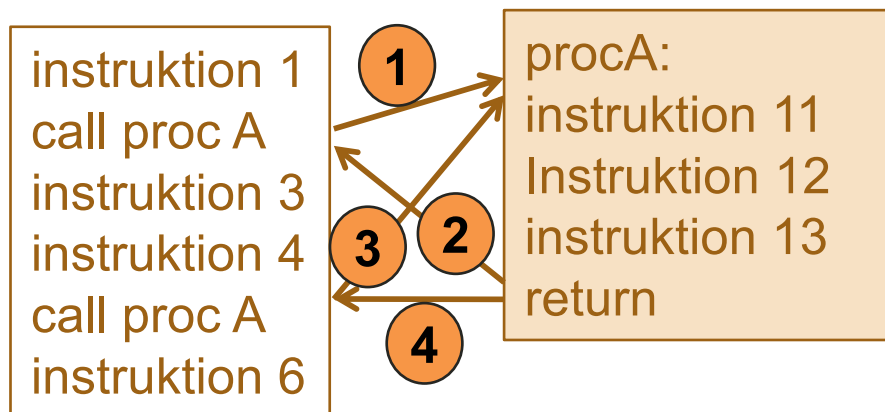
instruktion 11

återhopp+parameteröverföring



Subrutiner och funktioner

- Problem
 - Hopp till subrutin
 - Återhopp från subrutin
 - Parameteröverföring
 - Nästlade subrutinanrop

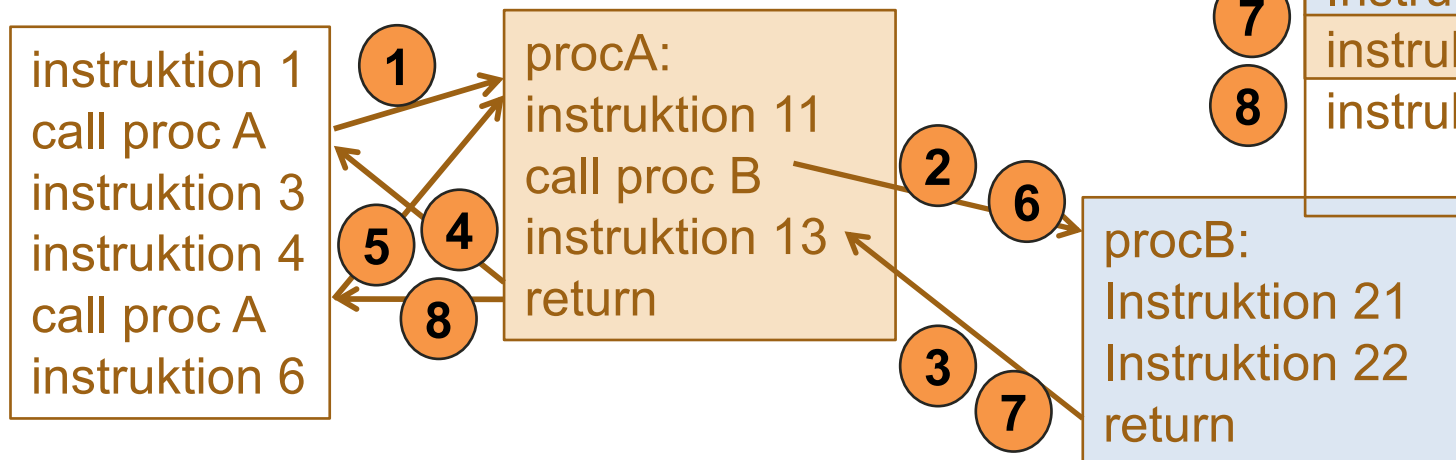


Exekvering:	
1	instruktion 1
2	instruktion 11 //Proc A
	Instruktion 12 //Proc A
	Instruktion 13 //Proc A
3	instruktion 3
	instruktion 4
4	instruktion 11 //Proc A
	Instruktion 12 //Proc A
	instruktion 13 //Proc A
	instruktion 6



Subrutiner och funktioner

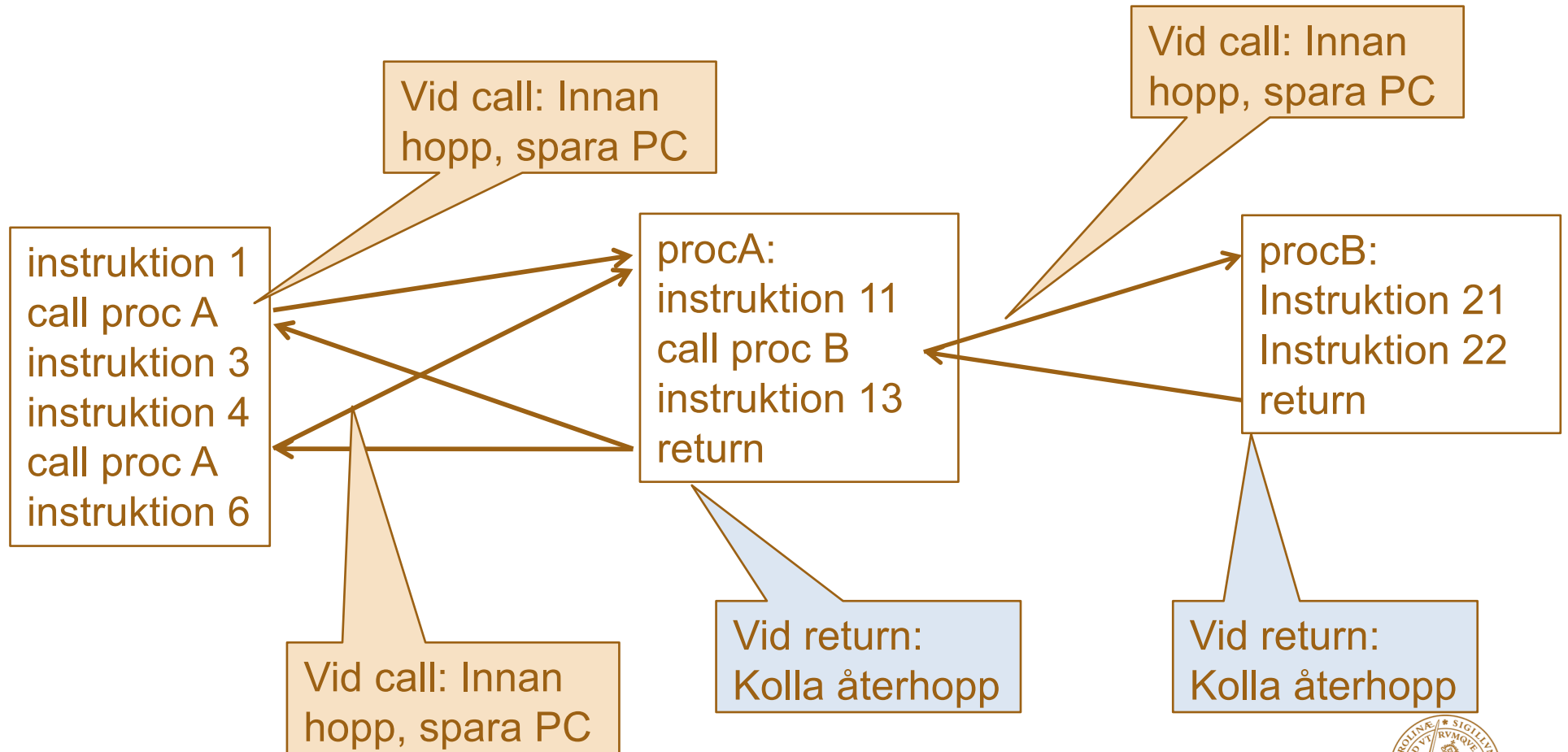
- Problem
 - Hopp till subrutin
 - Återhopp från subrutin
 - Parameteröverföring
 - Nästlade subrutinanrop



Exekvering:	
1	instruktion 1
2	instruktion 11 //Proc A
3	Instruktion 21 //Proc B
4	Instruktion 22
5	instruktion 13 //Proc A
6	instruktion 3
7	instruktion 4
8	instruktion 11 //Proc A
9	Instruktion 21 //Proc B
10	Instruktion 22
11	instruktion 13 //Proc A
12	instruktion 6



Subrutiner och funktioner



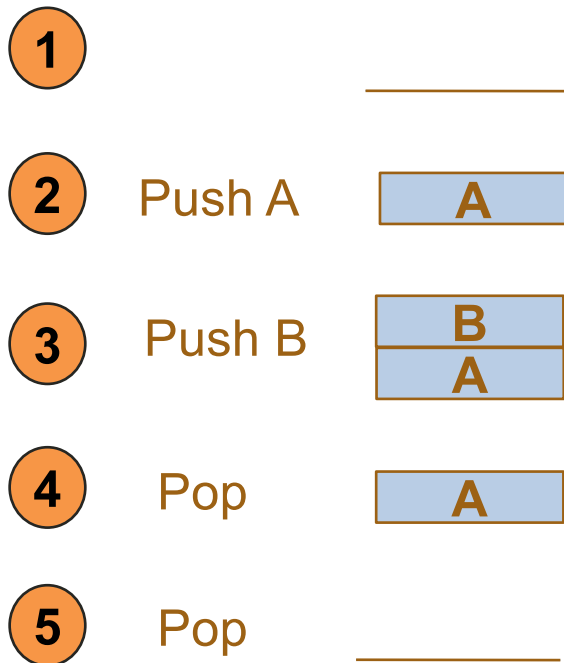
Återhopp och parameteröverföring

- Register
 - Använd ett antal register för att spara återhoppadress och för parameteröverföring
 - Fördel: snabbt. Nackdel: register behövs
- Stack
 - Reservera en del av minnet och skapa en kö som fungerar enligt Last-In First-Out
 - Fördel: stort. Nackdel: långsamt (jmf register)



Stack

- Två operationer: push och pop



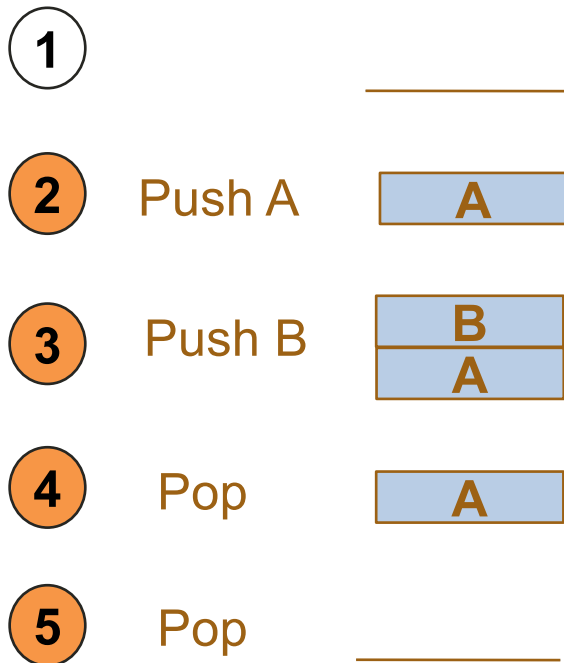
1 Stackpekare



Adress	Byte	Data
0	0	1111 0000
1	1	1010 0101
2	2	1100 0011
3	3	0011 0011
4	4	1111 1111
5	5	0000 1111
6	6	1111 0000
7	7	1010 1010

Stack

- Två operationer: push och pop

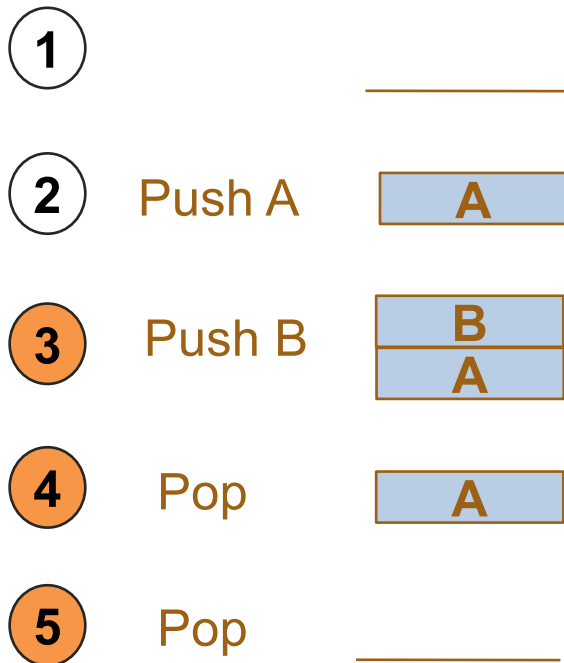


② Stackpekare →

Adress	Byte	Data
0	0	1111 0000
1	1	1010 0101
2	2	1100 0011
3	3	0011 0011
4	4	A
5	5	0000 1111
6	6	1111 0000
7	7	1010 1010

Stack

- Två operationer: push och pop

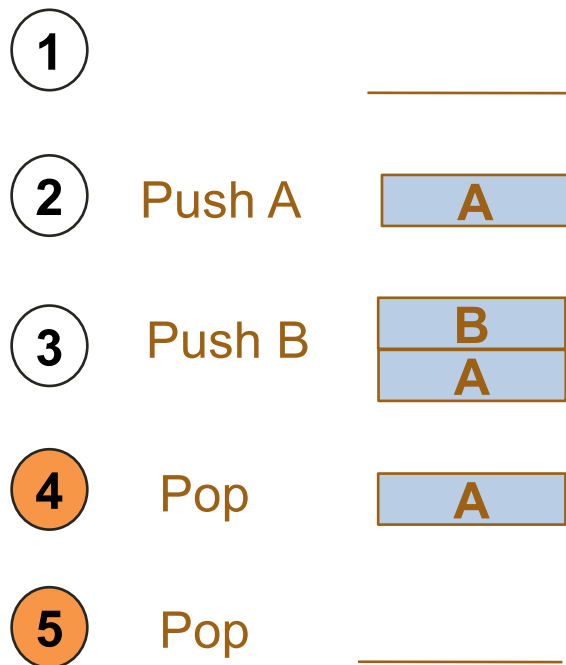


3 Stackpekare →

Adress	Byte	Data
0	0	1111 0000
1	1	1010 0101
2	2	1100 0011
3	3	0011 0011
4	4	A
5	5	B
6	6	1111 0000
7	7	1010 1010

Stack

- Två operationer: push och pop

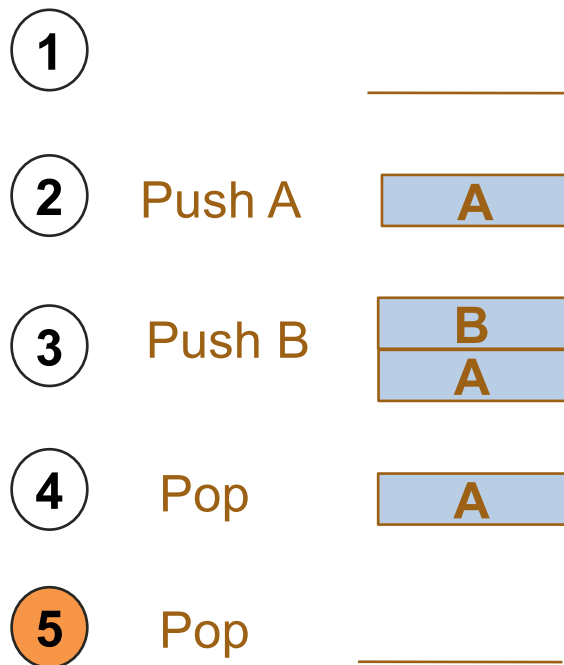


4 Stackpekare →

Adress	Byte	Data
0	0	1111 0000
1	1	1010 0101
2	2	1100 0011
3	3	0011 0011
4	4	A
5	5	B
6	6	1111 0000
7	7	1010 1010

Stack

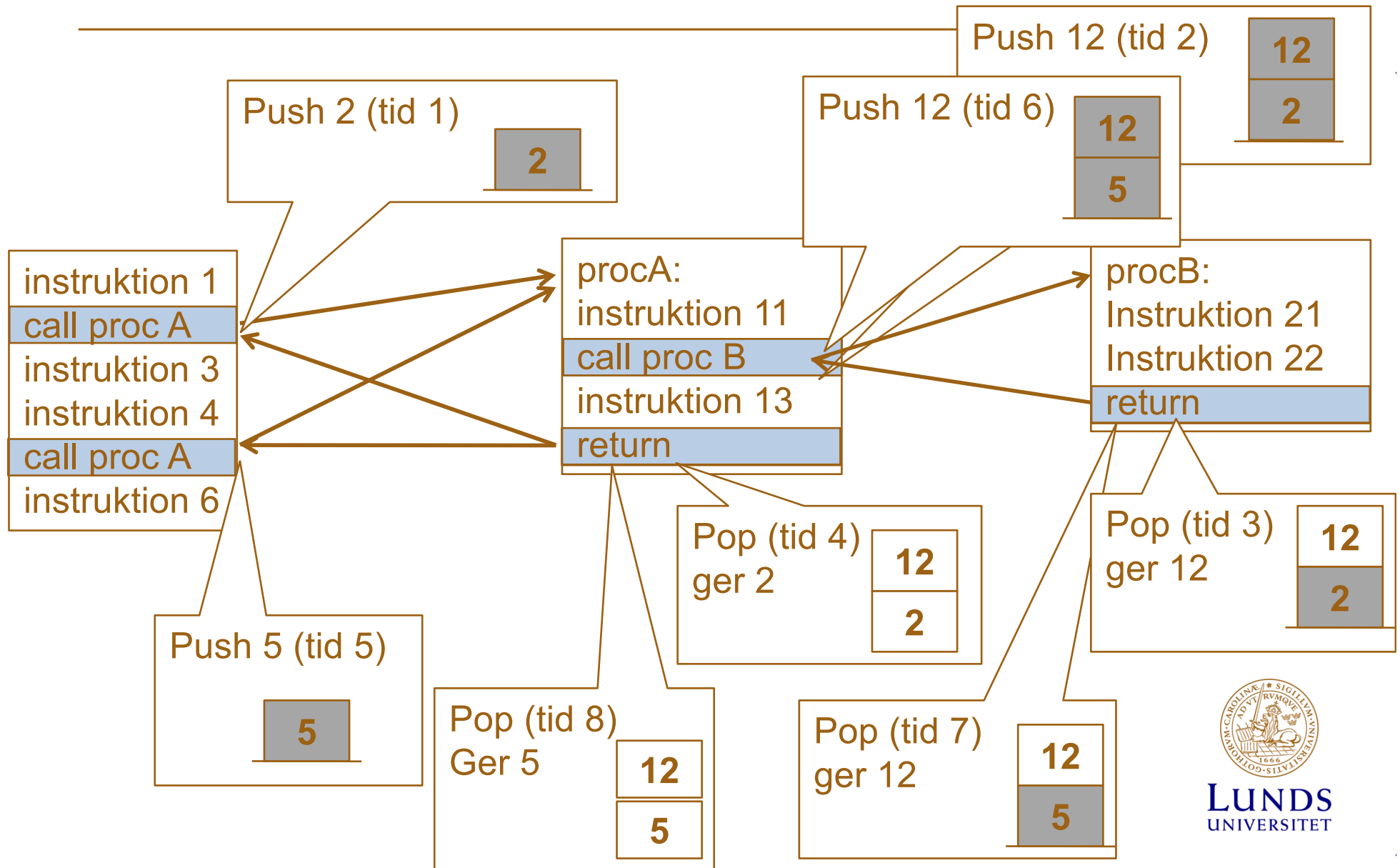
- Två operationer: push och pop



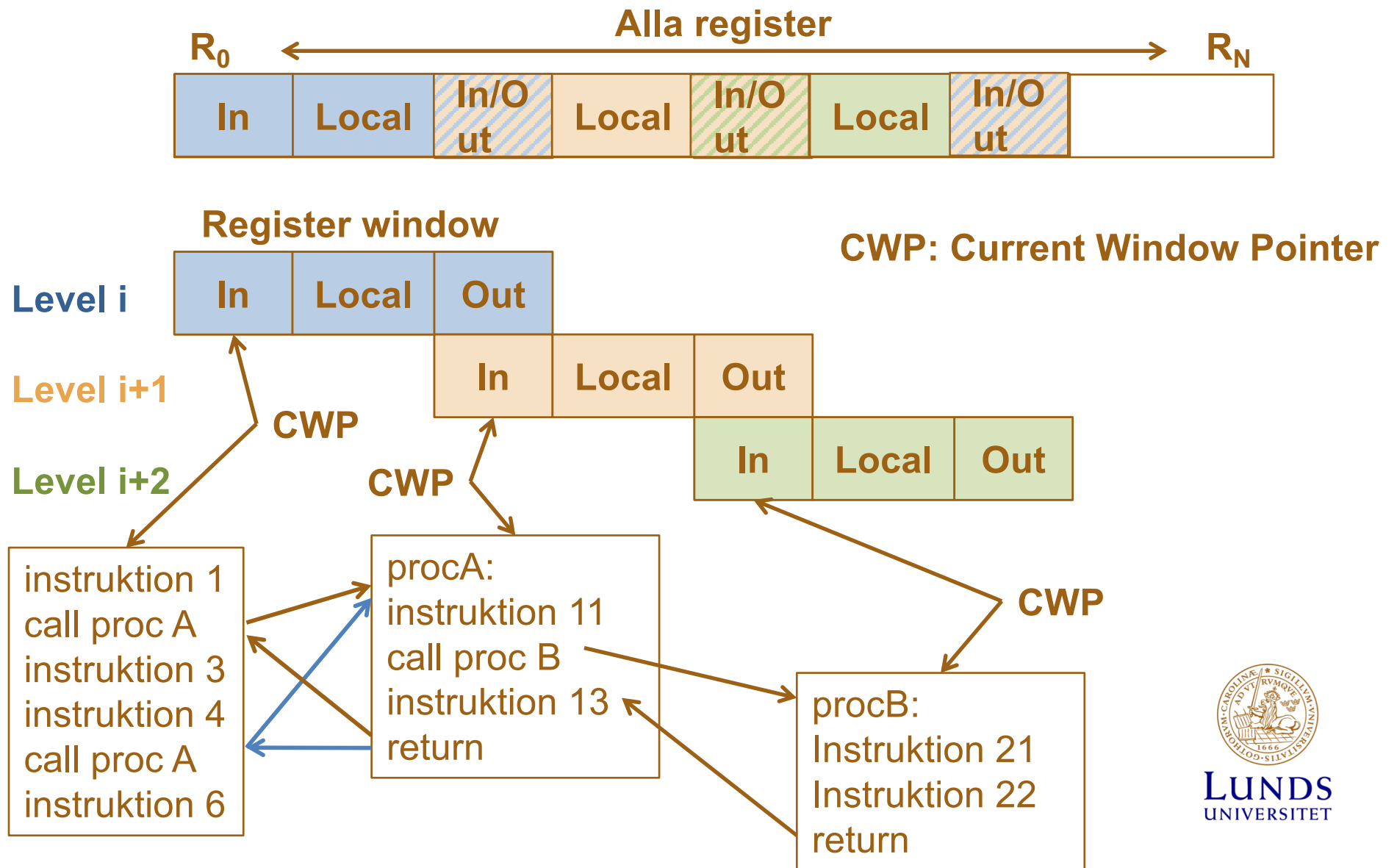
5 Stackpekare →

Adress	Byte	Data
0	0	1111 0000
1	1	1010 0101
2	2	1100 0011
3	3	0011 0011
4	4	A
5	5	B
6	6	1111 0000
7	7	1010 1010

Stack för subrutiner och funktioner



Register för subrutiner och funktioner



Återhopp och parameteröverföring

- Register
 - Använd ett antal register för att spara återhoppadress och för parameteröverföring
 - Fördel: snabbt. Nackdel: register behövs
- Stack
 - Reservera en del av minnet och skapa en kö som fungerar enligt Last-In First-Out
 - Fördel: Minnet är stort. Nackdel: långsamt (jmf register)

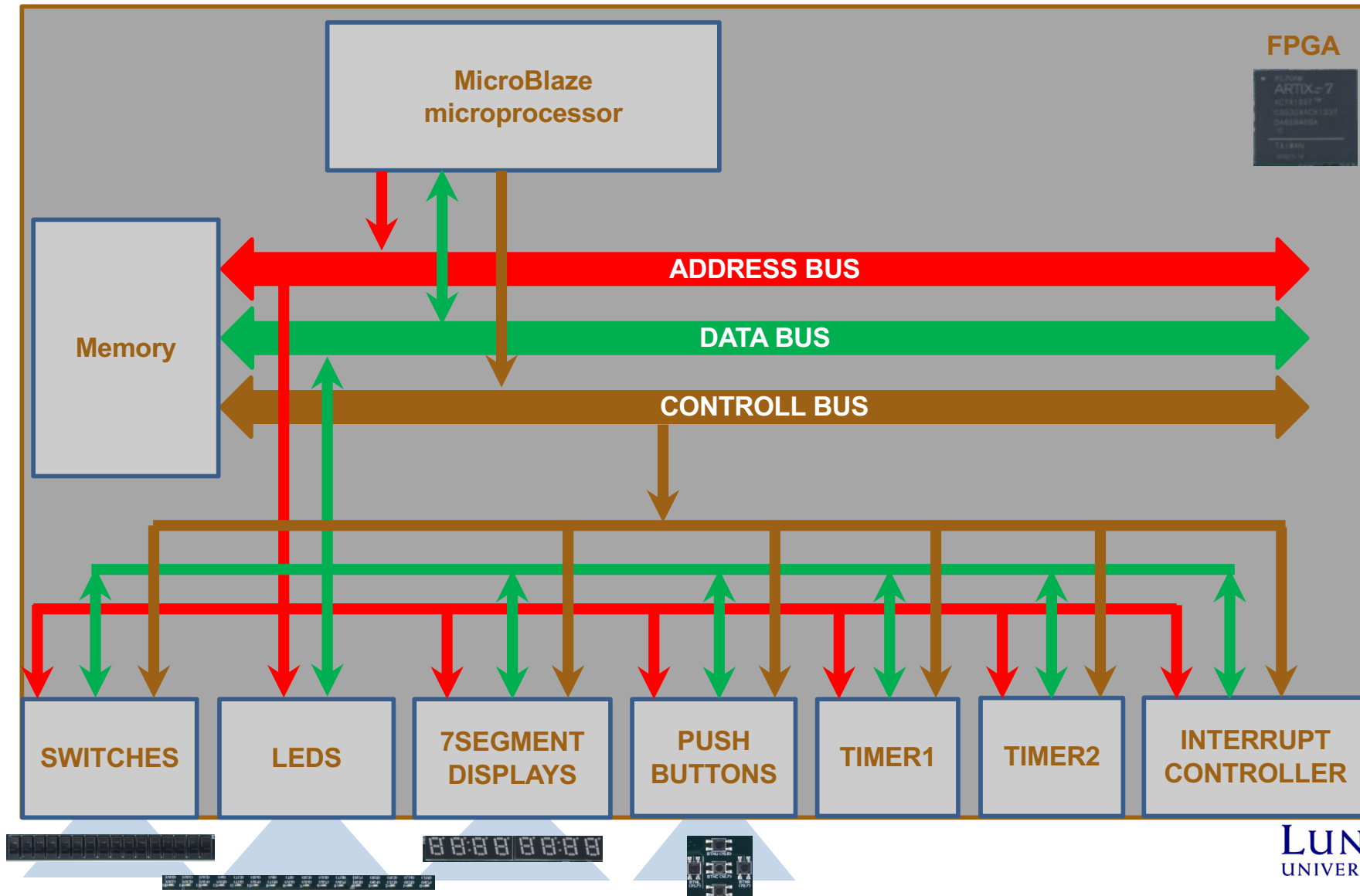


Maskininstruktioner

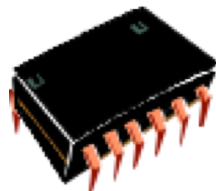
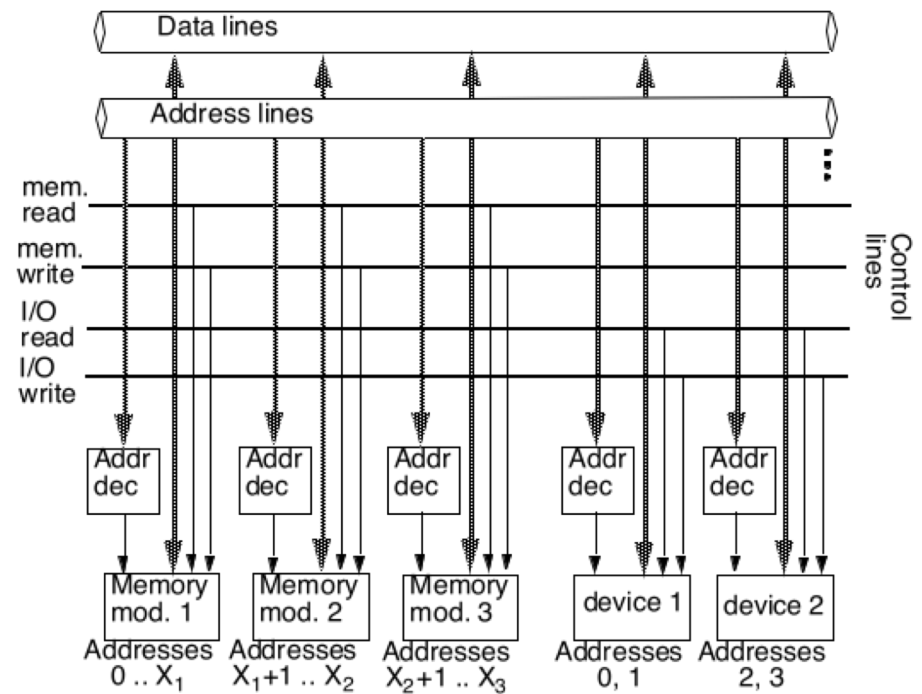
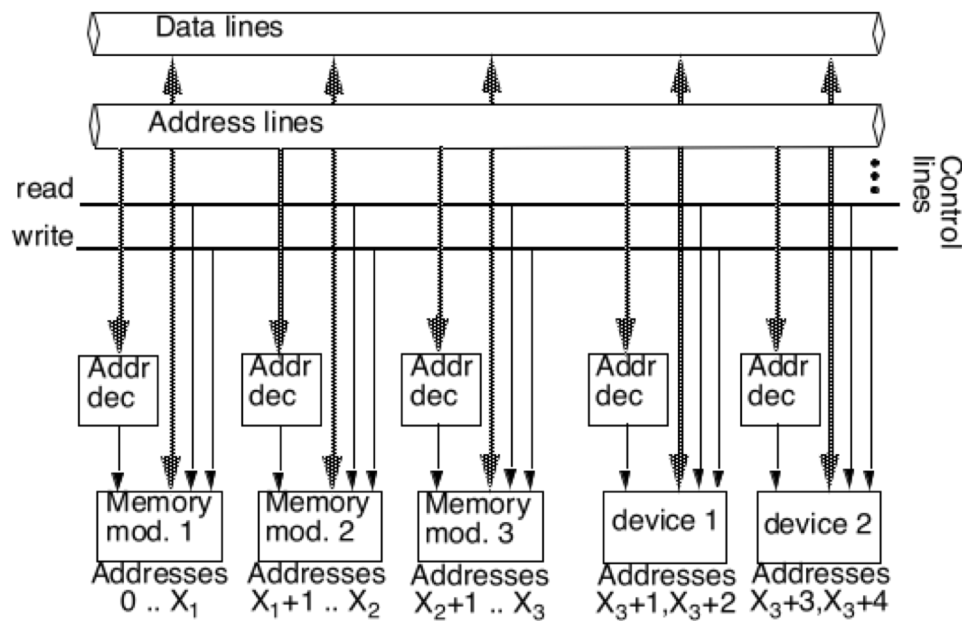
- Aritmetiska och logiska (ALU)
- Dataöverföring (adressering)
- Hopp och subrutiner
- Inmatning/utmating (Input/Output (I/O))



In- och utmatning



Minnesmappad och isolerad I/O



Exempel på minnesmappad I/O

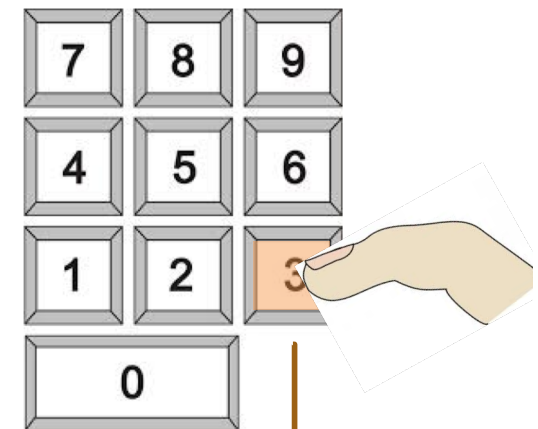
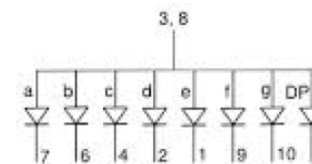
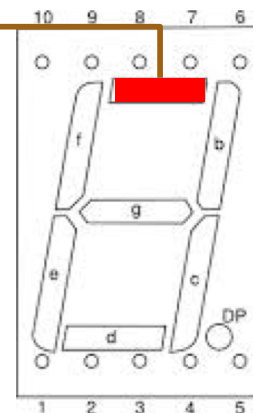
LOAD R1, 7

//Ladda R1 med värde på adress 7

STORE R2, 6



//Lagra värdet i R1 på adressplats 6

Adress	Instruktion/ Data
0	1111 0000
1	1010 0101
2	1100 0011
3	0011 0011
4	1111 1111
5	0000 1111
6	0010 0000
7	0000 0100



Exempel på minnesmappad I/O

- Uppgift: Har någon tryckt på knapp 3?
 - Finns i variabel `x` (unsigned char)
- Om någon tryckt knapp 3 innehåller `x`: 0000 0100
(men det kan också vara så att någon tryckt på knapp 3 och 4, dvs `x=0000 1100`)
- Lösning: maska bort det som inte är intressant

`x & 0b000000100)`
OCH  
Denna bit är intressant



Exempel på minnesmappad I/O

- Uppgift: Tänd övre segment (och andra segment ska vara som de är) som sitter i position 4.
- Övre segment finns i variabel x (unsigned char)

x=x | 0b0010000



ELLER

Denna bit är intressant. Den ska sättas till 1 och alla andra bitar i x ska vara som innan



Maskininstruktioner

- Att bestämma:
 - Typ av instruktioner (operander och operationer)
 - Antal adresser
 - Instruktionsformat
 - » Fixed eller flexibelt



Antal adresser i instruktion

- För att beräkna: $X = (A+B)*C$

- 4-adress instruktioner:

– Op A1, A2, A3, A4

//A1 <- [A2] op [A3], nästa instruktion = A4

I1: ADD X, A, B, I2

I2: MUL X, X, C, I3

I3:



- 3-adress instruktioner:

– Op A1, A2, A3

//A1 <- [A2] op [A3]

I1: ADD X, A, B

I2: MUL X, X, C

I3:



Antal adresser i instruktion

- För att beräkna: $X = (A+B)*C$
- 2-adress instruktioner:

– Op A1, A2

//A1 <- [A1] op [A2]

MOVE A, X
ADD X, B
MUL X, C



- 1-adress instruktioner:

– Op A1

//Acc <- [Acc] op [A1]

LOAD A
ADD B
MUL C
STORE X



Antal adresser i instruktion

- Vanligast med 2 och 3 adresser i instruktioner
- 4 adress instruktioner är opraktiskt.
 - Nästa instruktion antas vara nästa instruktion. PC räknas upp “av sig själv”. Måste ha hopp-instruktioner
- 1 adress instruktion är ganska begränsande
- Exempel, antag 32 register (5 bitar behövs):
 - 3-adress format: 15 bitar
 - 2-adress format: 10 bitar
 - 1-adress format: 5 bitar
- Få adresser, få instruktioner förenklar processorn men gör den mer primitiv



Maskininstruktioner

- Att bestämma:
 - Typ av instruktioner (operander och operationer)
 - Antal adresser
 - Instruktionsformat
 - » Fixed eller flexibelt
 - Alla instruktioner är lika långa -> enklare kontrollenhet
 - Instruktioner olika längd -> flexibilitet, adressrymd

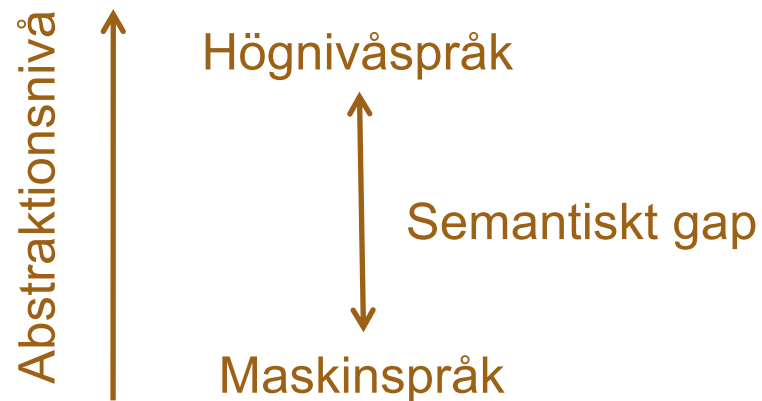


Instruktionsformat

- Hur definiera fälten i instruktioner?

Opcode	Operand1	Operand2
---------------	-----------------	-----------------

- Hur ska högnivåspråk kompileras och exekveras effektivt?



Intel x86 ISA

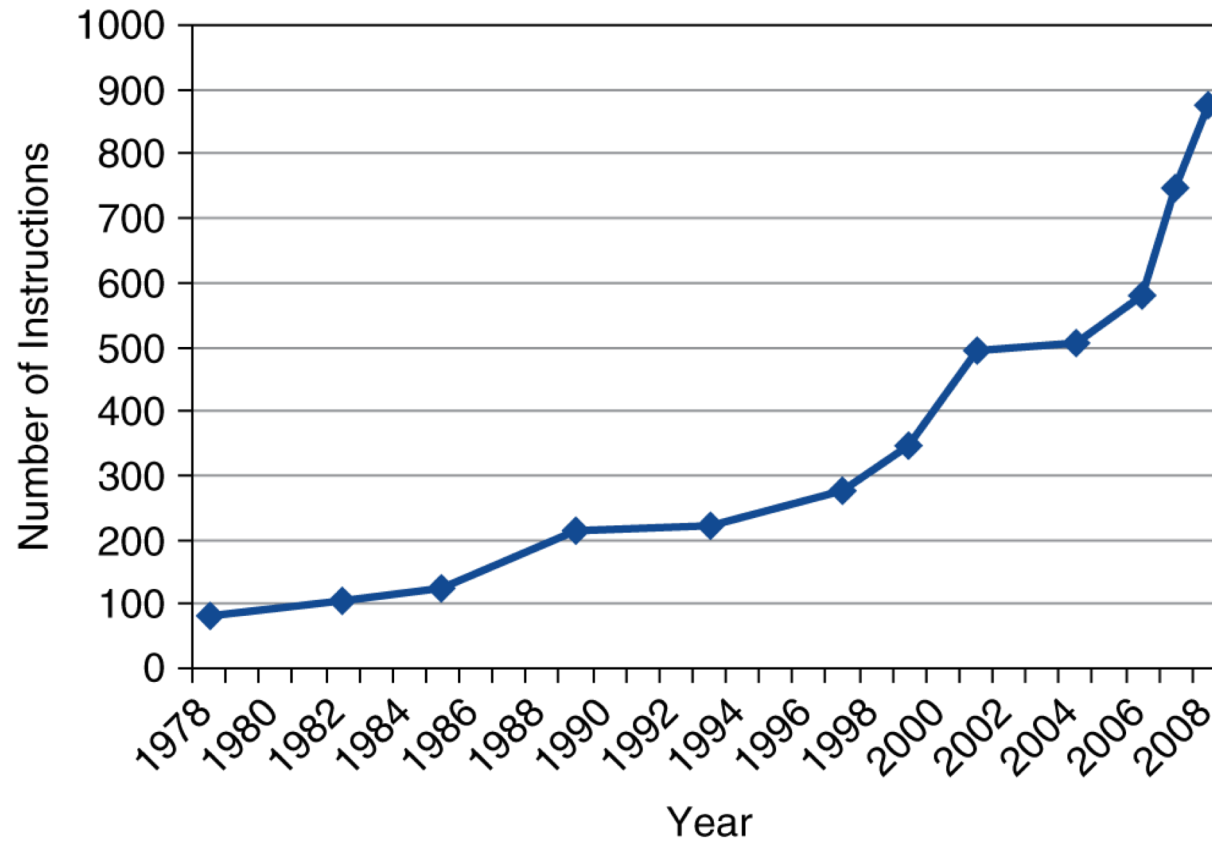
- Utveckling med backward compatibility
 - 8080 (1974): 8-bit microprocessor
 - » Accumulator, plus 3 index-register pairs
 - 8086 (1978): 16-bit extension to 8080
 - » Complex instruction set (CISC)
 - 8087 (1980): floating-point coprocessor
 - » Adds FP instructions and register stack
 - 80286 (1982): 24-bit addresses, MMU
 - » Segmented memory mapping and protection
 - 80386 (1985): 32-bit extension (now IA-32)
 - » Additional addressing modes and operations
 - » Paged memory mapping as well as segments

**AMD och Intel
bygger för x86
arkitektur men
organiserar på
olika sätt**



Intel x86 ISA

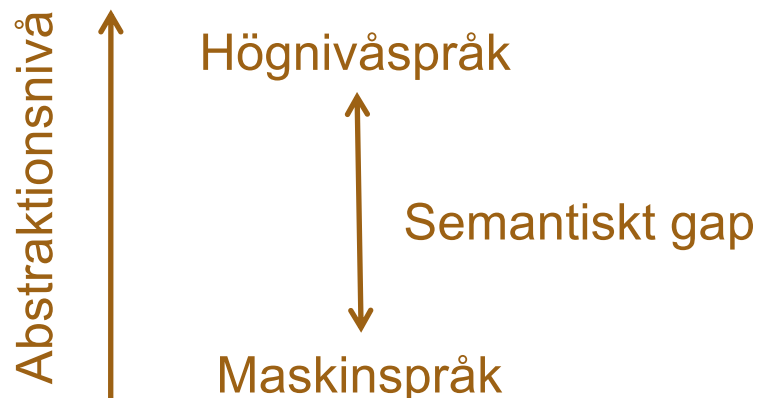
- Bakåtkompabilitet (Backward compatibility)



Semantiskt glapp

Två alternativ:

- CISC (Complex Instruction Set Computers):
 - utveckla komplex arkitektur med många instruktioner och många adresseringsmöjligheter; ta med instruktioner som liknas högnivåinstruktioner.
- RISC (Reduced Instruction Set Computers):
 - förenkla instruktionsuppsättningen och anpassa den till de verkliga kraven ett applikationsprogram har



Utvärdering av applikationsprogram

- Många studier har gjorts för att ta fram karaktärsdrag av maskinkod som genererats från högnivåspråk
- Intressanta aspekter:
 - Hur ofta används varje instruktion?
 - Hur används operander? Och hur ofta?
 - Hur mycket används hopp, loopar, anrop till procedurer/funktioner/subrutiner?



Utvärdering av applikationsprogram

Typ	Instruktioner i högnivåspråk (%)		Instruktioner i maskinspråk (%)		Minnesreferenser (%)	
	Pascal	C	Pascal	C	Pascal	C
Tilldelning	45	38	13	13	14	15
Loop	5	3	42	32	33	26
Call	15	12	31	33	44	45
Villkor (if)	29	43	11	21	7	13
Andra	6	1	3	1	2	1

Många tilldelningar i högnivåspråk men dessa ger få instruktioner i maskinspråk och relativt få minnesreferenser

Relativt få loopar men looparna resulterar i många maskininstruktioner och många minnesreferenser

Utvärdering av applikationsprogram

Typ	Instruktioner i högnivåspråk (%)		Instruktioner i maskinspråk (%)		Minnesreferenser (%)	
	Pascal	C	Pascal	C	Pascal	C
Tilldelning	45	38	13	13	14	15
Loop	5	3	42	32	33	26
Call	15	12	31	33	44	45
Villkor (if)	29	43	11	21	7	13
Andra	6	1	3	1	2	1

Få CALL (anrop till funktioner/procedurer/subrutiner) men de ger många instruktioner i maskinspråk och relativt många minnesreferenser

Relativt många villkor med de ger relativt få maskininstruktioner och relativt få minnesreferenser



Utvärdering av applikationsprogram

- Fördelning av maskininstruktioner (frekvens av användande):
 - Flytta data (move): ~33%
 - Villkorliga hopp: ~20%
 - ALU operationer: ~16%
- Adressering
 - Komplexa adressering: ~18% (memory indirect, indexed+indirect, displacement+indexed, stack)
 - Enkel adressering: vanligast (adress kan beräknas i en klockcykel, register, register indirekt, displacement)



Utvärdering av applikationsprogram

- Typer av operander
 - 74-80% av operander är skalärer (heltal, flyttal, tecken, etc), vilka kan lagras i register (icke-skalär, t ex array)
 - Resterande (20-26%) är arrayer; 90% är globala variabler
 - 80% av skalärer är lokala variabler
- Slutsats: Lagring i register



Utvärdering av applikationsprogram

- Trots att endast 10-15% av högnivå instruktionerna var funktion/procedure/subrutin anrop, så svarar dessa CALL och RETURN instruktioner för:
 - 31-43% av maskininstruktionerna
 - 44-45% av minnesreferenserna
- För procedurer:
 - Endast 1.25% har fler än 6 parametrar
 - Endast 6.7% har fler än 6 lokala variabler
 - Antal nästlingar är ofta få (korta) och mycket sällan längre än 6

```
int x, y; float z;
void proc3(int a, int b, int c) {
    -----
}
void proc2(int k) {
    int j, q;
    -----
    proc3(j, 5, q);
}
void proc1() {
    int i;
    -----
    proc2(i);
}
main () {
    proc1();
}
```



Karaktärsdrag hos RISC processorer

- Enkla och få instruktioner
- Enkla och få adresseringsmöjligheter
 - Typiskt: register, direkt, register indirekt, displacement
- Instruktioner har fixt (fast) längd och uniformt format
 - Detta förenklar och snabbar upp laddning och avkodning. Processorn behöver inte vänta på att hela instruktionen är laddad innan avkodning kan påbörjas
 - Uniformt format, dvs opcode och adressfält är placerat på samma position för olika instruktioner, gör att avkodning förenklas.



Karaktärsdrag hos RISC processorer

- Stort antal register
 - Register kan användas för att lagra variabler och för “mellanlagring” resultat. Det minskar behov av att använda upprepade load och store med primärminnet.
 - Alla lokala variabler från CALL/RETURN strukturer (anrop till funktioner, procedurer, subrutiner) kan lagras i register.
- Vad händer vid anrop till funktioner och procedurer?
 - Register innehåll måste sparas, parametrar ska föras över, återhopp adresser ska sparas. Relativt mycket minnes access, vilket tar mycket tid.
 - Med många register, kan nya register allokeras vid anrop



Karaktärsdrag hos RISC processorer

- Load-and-store architecture: dela upp instruktioner så att vissa gör Load/Store och andra gör ALU-operation, dvs inga instruktioner finns som läser från minnet OCH gör ALU operation



Karaktärsdrag hos RISC processorer

- Ett litet exempel:
 - Antag ett program med 80% enkla instruktioner och 20% komplexa instruktioner som exekveras på en CISC och en RISC
 - CISC: enkla instruktioner tar 4 klockcykler medan komplexa instruktioner tar 8 klockcykler. Antag klockcykeltid: 100 ns (10^{-7} s)
 - RISC: enkla instruktioner 1 klockcykel medan komplexa instruktioner implementeras som en sekvens av enkla. Antag att det tar i genomsnitt 14 instruktioner för en komplex instruktion. Antag klockcykeltid på 75 ns ($0.75 \cdot 10^{-7}$).
- Hur lång tid tar det att exekvera 1 000 000 instruktioner?
 - CISC: $(10^6 \cdot 0.80 \cdot 4 + 10^6 \cdot 0.20 \cdot 8) \cdot 10^{-7} = 0.48\text{s}$
 - RISC: $(10^6 \cdot 0.80 \cdot 1 + 10^6 \cdot 0.20 \cdot 14) \cdot 0.75 \cdot 10^{-7} = 0.27\text{s}$



Karaktärsdrag hos RISC processorer

- Ett litet exempel (fortsättning):
 - Komplexa instruktioner tar längre tid på en RISC, men det är relativt fler enkla instruktioner
 - På grund av sin (relativa) enkelhet, kan RISC operera med kortare cykeltid (högre klockfrekvens). För CISC bromsas enkla instruktioner av mer komplex data path och mer komplex kontrollenhet (det som inte är kontrollenhet kallas för data path)



RISC eller CISC?

- Ett entydigt svar är svårt att ge
- Flera prestanda jämförelser (benchmarking) visar att RISC exekverar snabbare än CISC
- Men, det är svårt att identifiera vilket karaktärsdrag som är avgörande. Det är svårt att jämföra; t ex olika halvledartechniker, kompilatorer
- Ett argument för CISC: Den enklare RISC konstruktionen gör att program behöver mer minne (fler instruktioner) jämfört med samma program för en CISC processor
- Processorer av idag använder sig ofta av lite RISC och lite CISC



Några exempel

- CISC

- VAX 11/780

- Nr. of instructions: 303

- Instruction size: 2 – 57 bytes

- Instruction format: not fixed

- Addressing modes: 22

- Number of general purpose registers: 16

- Pentium

- Nr. of instructions: 235

- Instruction size: 1 – 11 bytes

- Instruction format: not fixed

- Addressing modes: 11

- Number of general purpose registers: 8 (32-bitar mode), 16 (64-bitar mode)



Några exempel

- RISC
 - Sun SPARC
 - Nr. of instructions: 52
 - Instruction size: 4 bytes
 - Instruction format: fixed
 - Addressing modes: 2
 - Number of general purpose registers: up to 520
 - PowerPC
 - Nr. of instructions: 206
 - Instruction size: 4 bytes
 - Instruction format: not fixed (but small differences)
 - Addressing modes: 2
 - Number of general purpose registers: 32



Några exempel

- ARM (Advanced RISC Machine och Acorn RISC Machine)
 - Antal instruktioner (standard set): 122
 - Instruktions storlek: 4 (standard), 2 (Thumb) byte
 - Instruktions format: fixerat (olika mellan standard och Thumb)
 - Adresserings modes: 3
 - Antal general purpose register: 27 (16 kan användas samtidigt)
 - Dagens ARM processorer kan exekvera både standard set (32 bitars instruktioner och 16 bitars Thumb instruktions set. Thumb består av en delmängd av 32-bitars setet, kodat som 16-bitars instruktioner).





LUNDS
UNIVERSITET