Datorsystem VT 2022

F5

Instruction Set Architecture(ISA)

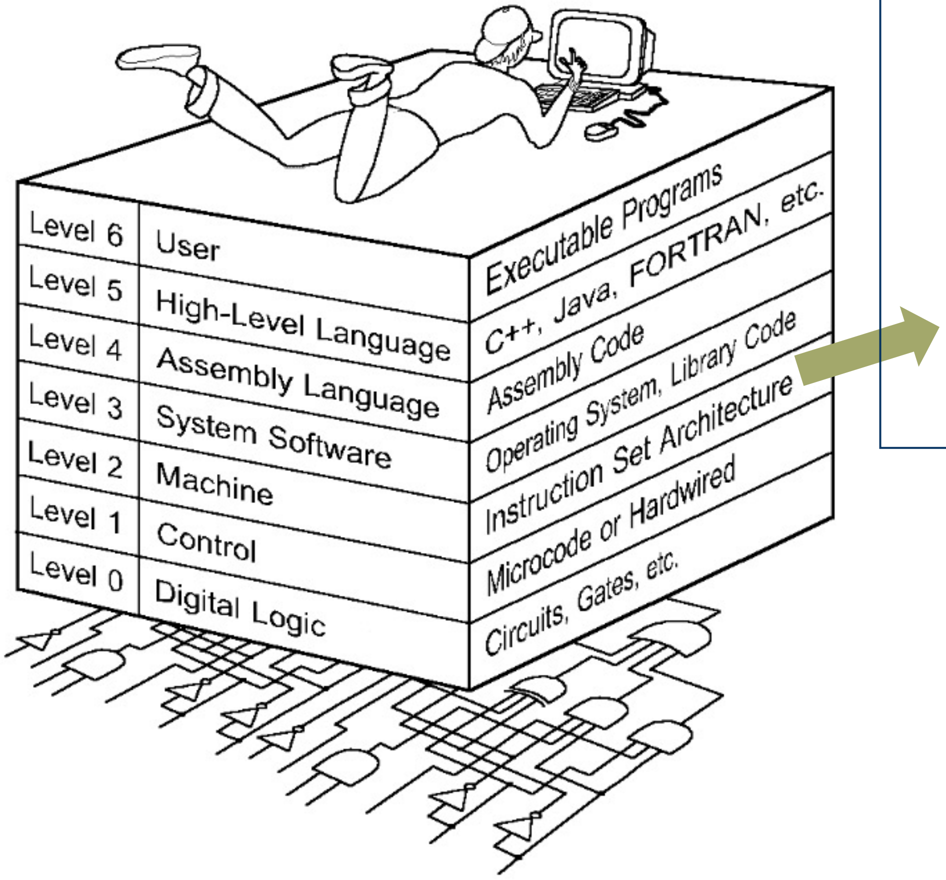
Stockholm
University

SUM Program

This program represents the formulas x = 2, y = 5, x + y = z

#	Machine Code	Assembly Code	Description
0	001 1 000010	LOAD #2	Load the value 2 into the Accumulator
1	010 0 001101	STORE 13	Store the value of the Accumulator in memory location 13
2	001 1 000101	LOAD #5	Load the value 5 into the Accumulator
3	010 0 001110	STORE 14	Store the value of the Accumulator in memory location 14
4	001 1 001101	LOAD 13	Load the value of memory location 13 into the Accumulator
5	011 0 001110	ADD 14	Add the value of memory location 14 to the Accumulator
6	010 0 001111	STORE 15	Store the value of the Accumulator in memory location 15
7	111 0 000000	HALT	Stop execution

The Computer Level Hierarchy



Level 2: Machine Level

- Also known as the Instruction Set Architecture (ISA) Level.
- Consists of instructions that are particular to the architecture of the machine.
- Programs written in machine language need no compilers, interpreters, or assemblers.

Outline

- Instruction Formats
- Instruction types
- Addressing
- Instruction Pipelining
- Example

Instruction Formats

Instruction sets are differentiated by the following:

- Number of bits per instruction.
- Stack-based or register-based.
- Number of explicit operands per instruction.
- Operand location.
- Types of operations.
- Type and size of operands.

Instruction Formats

Instruction Set Architectures are measured according to:

- Main memory space occupied by a program.
- Instruction complexity.
- Instruction length (in bits).
- Total **number of instructions** in the **instruction set.**

Instruction Formats

In designing an instruction set, consideration is given to:

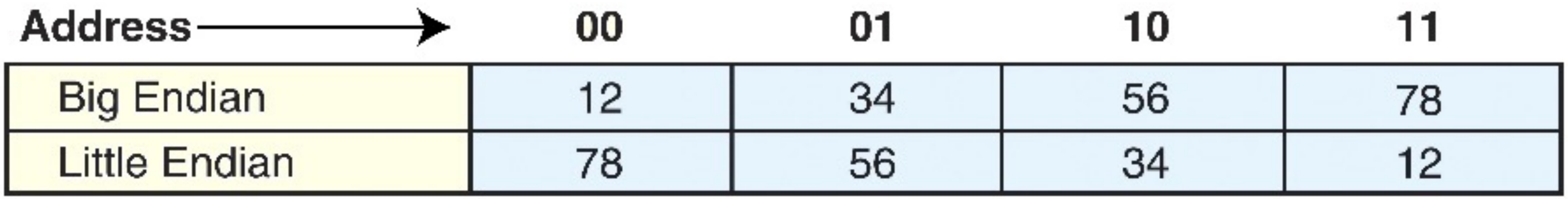
- Instruction length.
  - Whether short, long, or variable.
- Number of operands.
- Number of addressable registers.
- Memory organization.
  - Whether byte- or word addressable.
- Addressing modes.
  - Choose any or all: direct, indirect or indexed.

Instruction Formats

- Byte ordering, or *endianness*, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the **least** significant byte is followed by the **most** significant byte or vice versa.
  - In *little endian* machines, the **least s**ignificant byte is followed by the **most** significant byte.
  - *Big endian* machines store the most significant byte first (at the lower address).

Instruction Formats

- As an example, suppose we have the hexadecimal number 12345678.
- The big endian and small endian arrangements of the bytes are shown below.



Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

Instruction Formats

- **Big endian**:
  - Is more natural.
  - The sign of the number can be determined by looking at the byte at address offset 0.
  - Strings and integers are stored in the same order.
- **Little endian**:
  - Makes it easier to place values on non-word boundaries.
  - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

Instruction Formats

- The next consideration for architecture design concerns **how the CPU will store data.**
- We have **three choices**:

  1. A **stack** architecture
  2. An **accumulator** architecture
  3. A **G**eneral **P**urpose **R**egister(**GPR**) architecture.

- In choosing one over the other, the **tradeoffs** are simplicity (and cost) of hardware design with execution speed and ease of use.

Instruction Formats

- In a **stack** architecture, instructions and operands are implicitly taken from the stack.
  - **A stack cannot be accessed randomly**.
- In an **accumulator** architecture, one operand of a binary operation is implicitly in the accumulator.
  - One operand is in **memory**, creating lots of bus traffic.
- In a **General Purpose R**egister **(GPR)** architecture, registers can be used instead of memory.
  - Faster than accumulator architecture.
  - Efficient implementation for compilers.
  - **Results in longer instructions**.

Instruction Formats

- Most systems today are GPR systems.
- There are **three** types:
  - **Memory-memory** where two or three operands may be in memory.
  - **Register-memory** where at least one operand must be in a register.
  - ***Load-store*** where no operands may be in memory.
- The number of operands and the number of available registers has a direct affect on **instruction length.**

Instruction Formats

- **Stack machines** use one - and zero-operand instructions.
- **`LOAD`** and **`STORE`** instructions require *a single memory address operand*.
- Other instructions use operands from the stack implicitly.
- **`PUSH`** and **`POP`** operations involve only the stack's top element.
- Binary instructions (e.g., **`ADD`**, **`MULT`**) use the top two items on the stack.

Instruction Formats

- Stack architectures require us to think about *arithmetic expressions* a little differently.
- We are accustomed to writing expressions using *infix* notation, such as: Z = X + Y.
- Stack arithmetic requires that we use *postfix* notation: Z = XY+.
    - This is also called *reverse Polish notation*, (some what) in honor of its Polish inventor, Jan Lukasiewicz (1878 - 1956).

Instruction Formats

- The principal advantage of postfix notation is that parentheses are not used.
- For example, the infix expression,

$$Z = (X \times Y) + (W \times U),$$

becomes:

$$Z = X\ Y \times W\ U \times +$$

in postfix notation.

Instruction Formats

- Example: Convert the infix expression (2+3) - 6/3 to postfix:

2 3+ - 6/3	The sum 2 + 3 in parentheses takes precedence; we replace the term with 2 3 +.

Instruction Formats

- Example: Convert the infix expression (2+3) - 6/3 to postfix:

2 3+ - 6 3/

The division operator takes next precedence; we replace 6/3 with 6 3 /.

Instruction Formats

- Example: Convert the infix expression (2+3) - 6/3 to postfix:

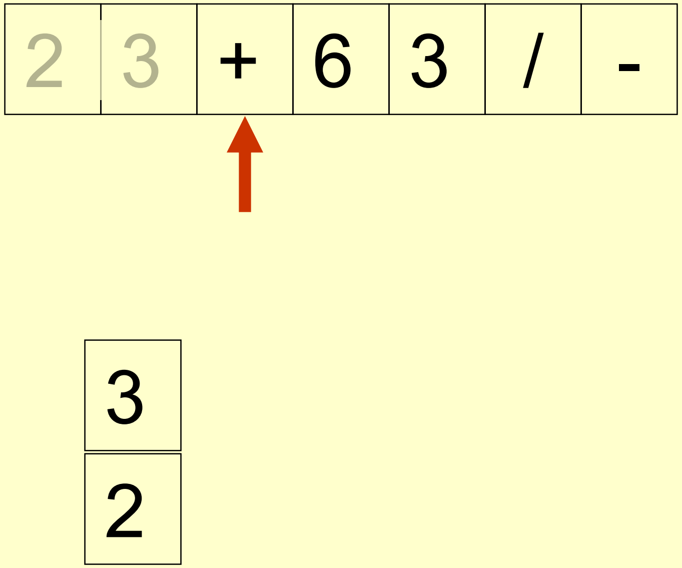
2 3+ 6 3/ -

The quotient 6/3 is subtracted from the sum of 2 + 3, so we move the - operator to the end.

Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

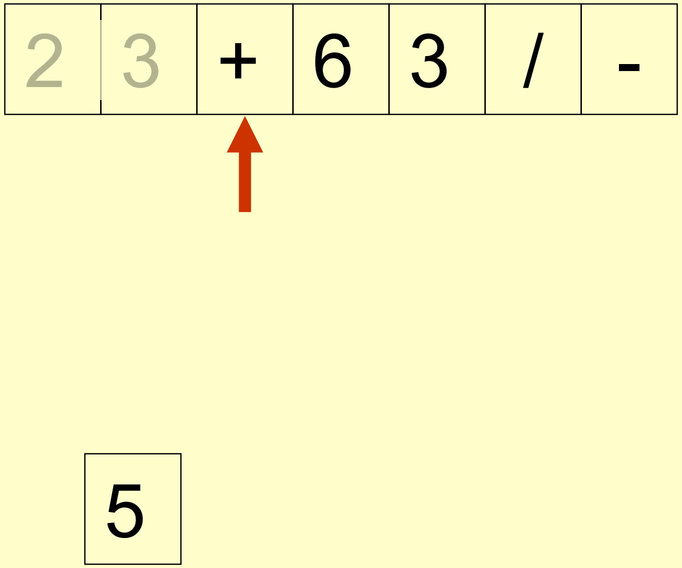
Scanning the expression from left to right, push operands onto the stack, until an operator is found



Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Pop the two operands and carry out the operation indicated by the operator. Push the result back on the stack.



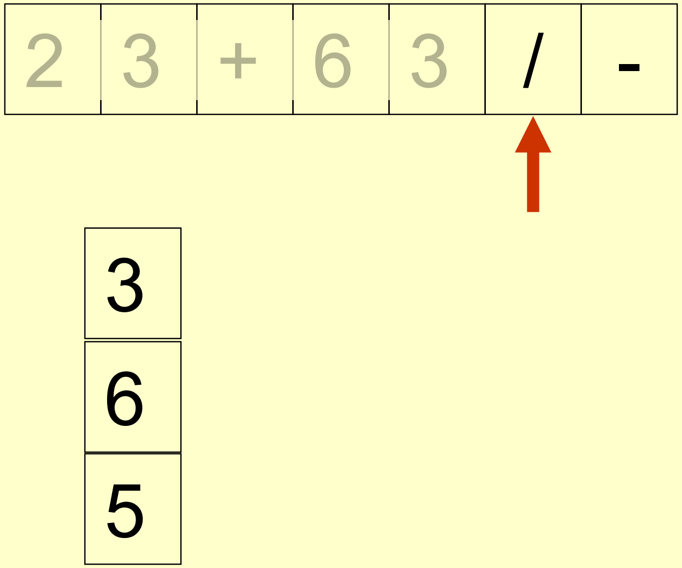
2	3	+	6	3	/	-

5

Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Push operands until another operator is found.



2 3 + 6 3 / -

3

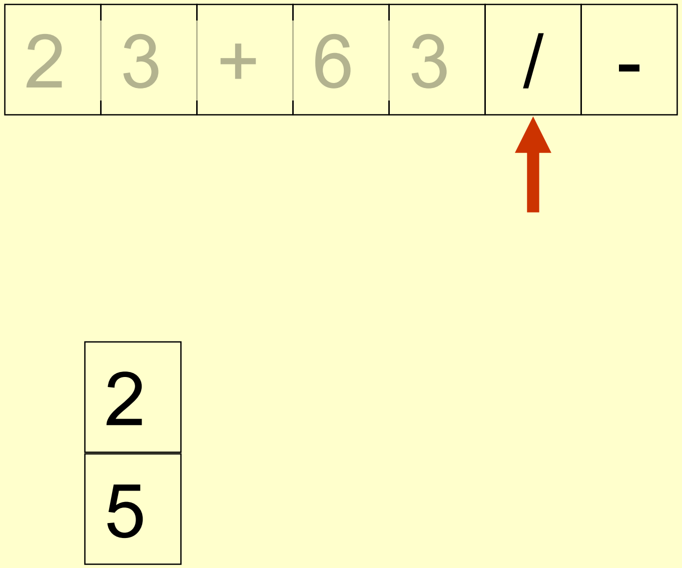
6

5

Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Carry out the operation and push the result.



2	3	+	6	3	/	-

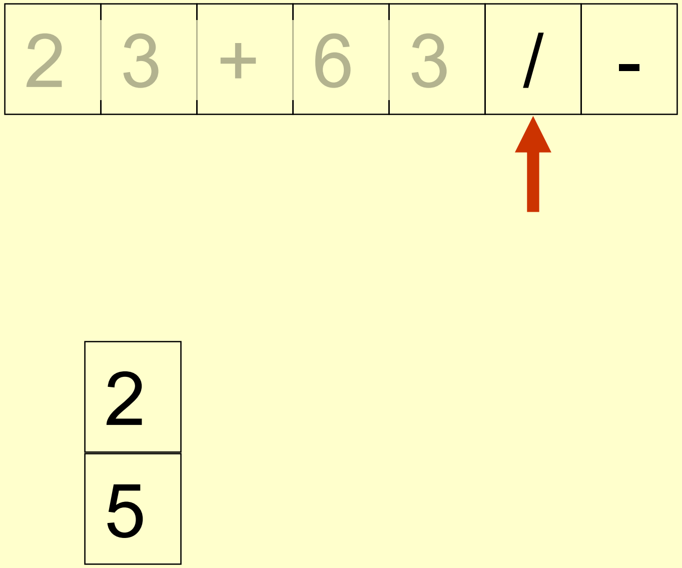
2

5

Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Carry out the operation and push the result.



2	3	+	6	3	/	-

2
5

Instruction Formats

- Let's see how to evaluate an infix expression using different instruction formats.
- With a three-address ISA, (e.g.,mainframes), the infix expression,

  Z = X × Y + W × U

  might look like this:

```
MULT R1,X,Y
MULT R2,W,U
ADD  Z,R1,R2
```

Instruction Formats

- In a two-address ISA, (e.g.,Intel, Motorola), the infix expression,

  Z = X × Y + W × U

  might look like this:

```
LOAD  R1,X
MULT  R1,Y
LOAD  R2,W
MULT  R2,U
ADD   R1,R2
STORE Z,R1
```

Instruction Formats

- In a one-address ISA, like MARIE, the infix expression,

  Z = X × Y + W × U

  looks like this:

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
ADD TEMP
STORE Z
```

Instruction Formats

- In a stack ISA, the postfix expression,

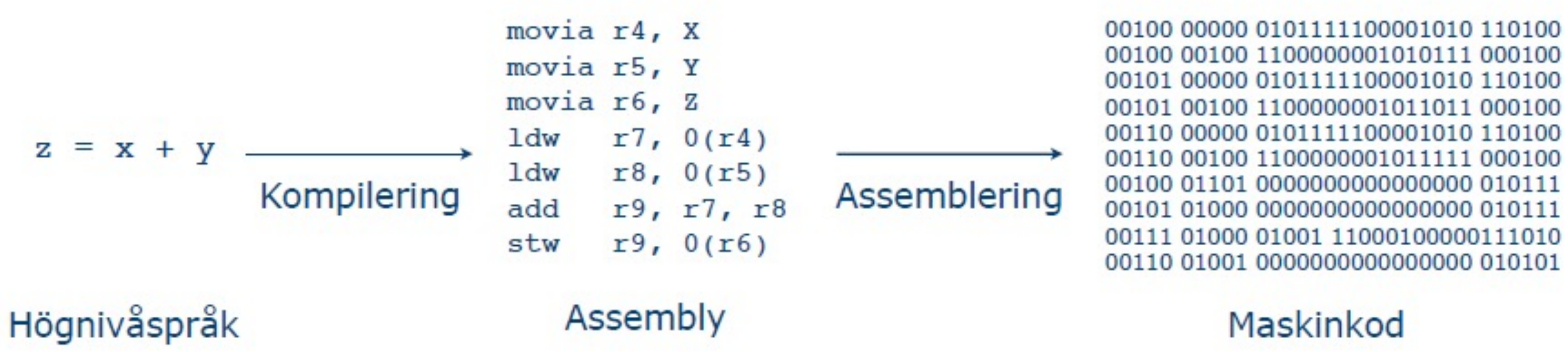
  Z = X Y × W U × +

  might look like this:

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
POP Z
```

Would this program require more execution time than the corresponding (shorter) program that we saw in the 3-address ISA?

Programmering på olika nivåer



```
z = x + y
```

```
movia r4, X
movia r5, Y
movia r6, Z
ldw   r7, 0(r4)
ldw   r8, 0(r5)
add   r9, r7, r8
stw   r9, 0(r6)
```

```
00100 00000 0101111100001010 110100
00100 00100 1100000001010111 000100
00101 00000 0101111100001010 110100
00101 00100 1100000001011011 000100
00110 00000 0101111100001010 110100
00110 00100 1100000001011111 000100
00100 01101 0000000000000000 010111
00101 01000 0000000000000000 010111
00111 01000 01001 11000100000111010
00110 01001 0000000000000000 010101
```

Instruction Formats

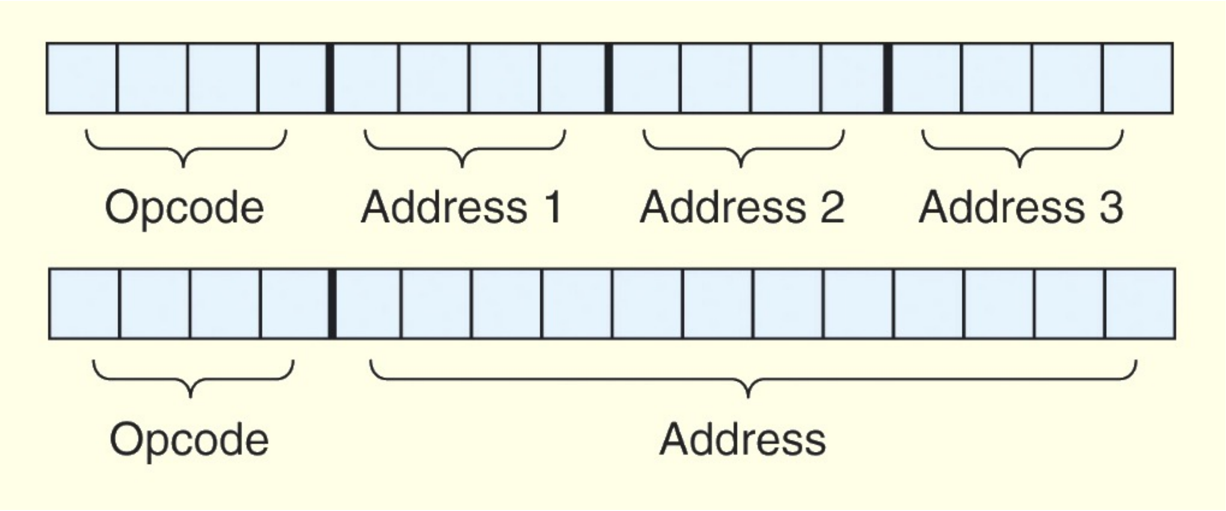
- We have seen how instruction length is affected by the number of operands supported by the ISA.
- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as **`HALT`**, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use **expanding opcodes.**

Instruction Formats

- A system has 16 registers and 4K of memory.

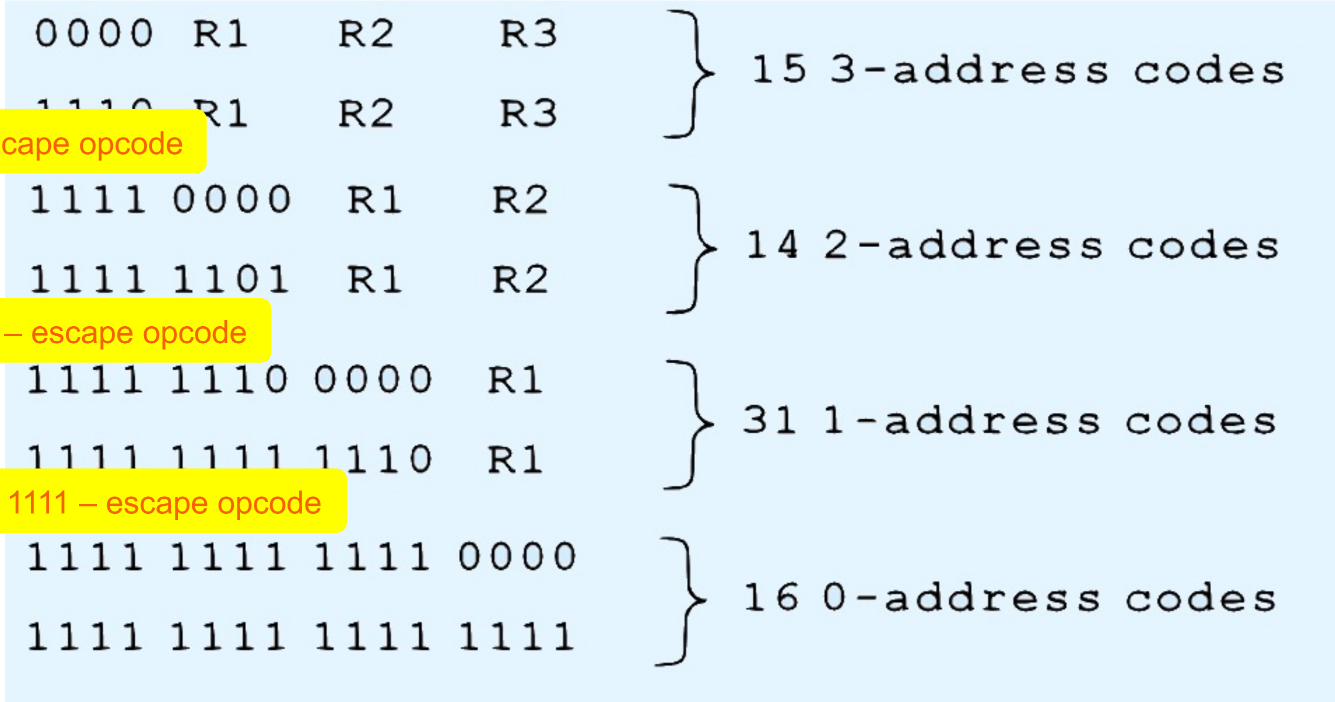
16 registers -> 2^4 , 4k memory =4000 → 2^{12} =4096

- We need 4 bits to access one of the registers. We also need 12 bits for a memory address.
- If the system is to have 16-bit instructions, we have two choices for our instructions:



Instruction Formats

- If we allow the length of the opcode to vary, we could create a very rich instruction set:



```
0000 R1   R2   R3
                     15 3-address codes
1110 R1   R2   R3
```

1111 – escape opcode

```
1111 0000 R1   R2
                     14 2-address codes
1111 1101 R1   R2
```

1111 1110 – escape opcode

```
1111 1110 0000 R1
                     31 1-address codes
1111 1111 1110 R1
```

1111 1111 1111 – escape opcode

```
1111 1111 1111 0000
                     16 0-address codes
1111 1111 1111 1111
```

Stockholm University

Instruction Formats

- Example: Given 8-bit instructions, is it possible to allow the following to be encoded? 2^8 ->256
  - – 3 instructions with two **3-bit** operands.
  - – 2 instructions with one **4-bit** operand.
  - – 4 instructions with one **3-bit** operand.

We need:

$$2^3 x 3 \times 2^3 = 192 \text{ bits for the 3-bit operands}$$

$$2 \times 2^4 = 32 \text{ bits for the 4-bit operands}$$

$$4 \times 2^3 = 32 \text{ bits for the 3-bit operands.}$$

Total: 256 bits.

Instruction Formats

- With a total of 256 bits required, we can exactly encode our instruction set in 8 bits!

We need:

2^3x$3 \times 2^3 = 192$ bits for the 3-bit operands

$2 \times 2^4 = 32$ bits for the 4-bit operands

$4 \times 2^3 = 32$ bits for the 3-bit operands.

Total: 256 bits.

One such encoding is shown on the next slide.

Instruction Formats

```
00   xxx xxx    }  3 instructions with two
01   xxx xxx    }  3-bit operands
10   xxx xxx    }
11 - escape opcode
1100 xxxx       }  2 instructions with one
1101 xxxx       }  4-bit operand
1110 - escape opcode
1111 - escape opcode
11100 xxx       }
11101 xxx       }  4 instructions with one
11110 xxx       }  3-bit operand
11111 xxx       }
```

Instruction types

Instructions fall into several broad categories that you should be familiar with:

- Data movement
- Arithmetic
- Boolean
- Bit manipulation
- I/O
- Control transfer
- Special purpose

Addressing

- **Addressing modes** specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- The actual location of an operand is its *effective address*.
- Certain addressing modes allow us to determine the address of an operand dynamically.

Addressing

- *Immediate addressing* is where the data is part of the instruction.
- *Direct addressing* is where the address of the data is given in the instruction.
- *Register addressing* is where the data is located in a register.
- *Indirect addressing* gives the address of the address of the data in the instruction.
- *Register indirect addressing* uses a register to store the address of the address of the data.

Addressing

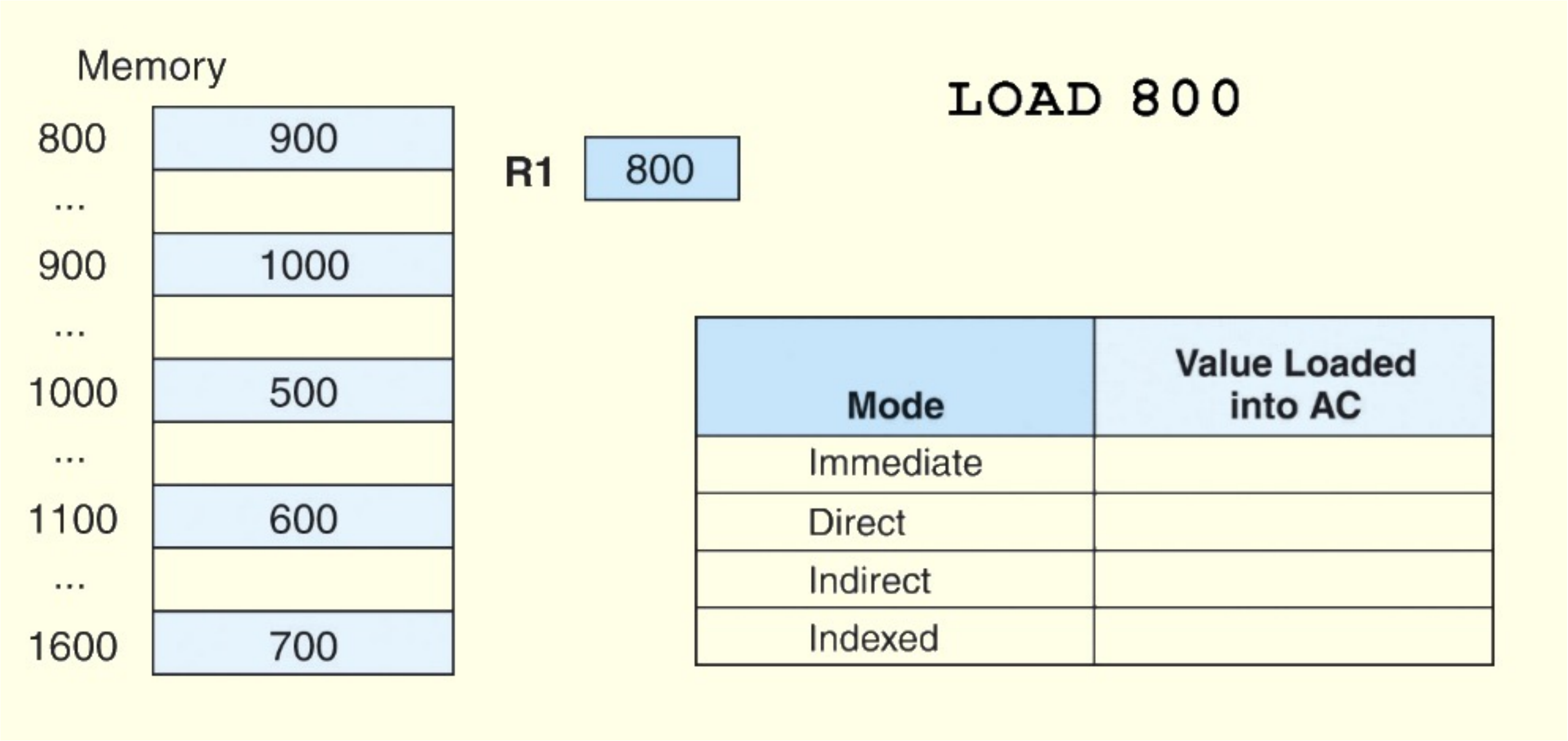
- *Indexed addressing* uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- *Based addressing* is similar except that a base register is used instead of an index register.
- The difference between these two is that an index register holds an offset relative to the address given in the instruction, a base register holds a base address where the address field represents a displacement from this base.

Addressing

- In *stack addressing* the operand is assumed to be on top of the stack
- There are many variations to these addressing modes including:
  - Indirect indexed
  - Base/offset
  - Self-relative
  - Auto increment - decrement
- We won’t cover these in detail

Addressing

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?



Memory

800	900
...	
900	1000
...	
1000	500
...	
1100	600
...	
1600	700

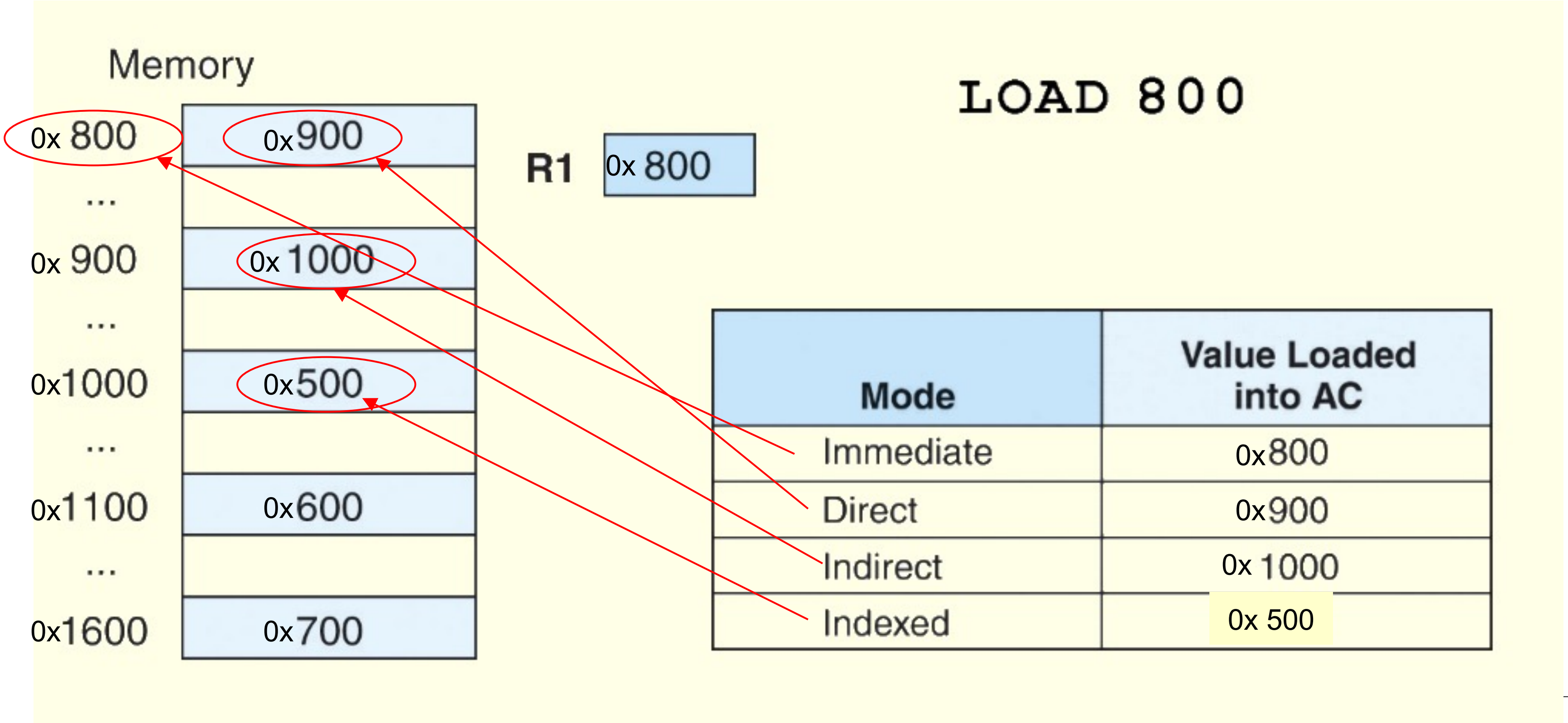
R1 800

```
LOAD 800
```

Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

Addressing

- These are the values loaded into the accumulator for each addressing mode.



Memory

0x 800	0x900
…	
0x 900	0x 1000
…	
0x1000	0x500
…	
0x1100	0x600
…	
0x1600	0x700

LOAD 800

R1 0x 800

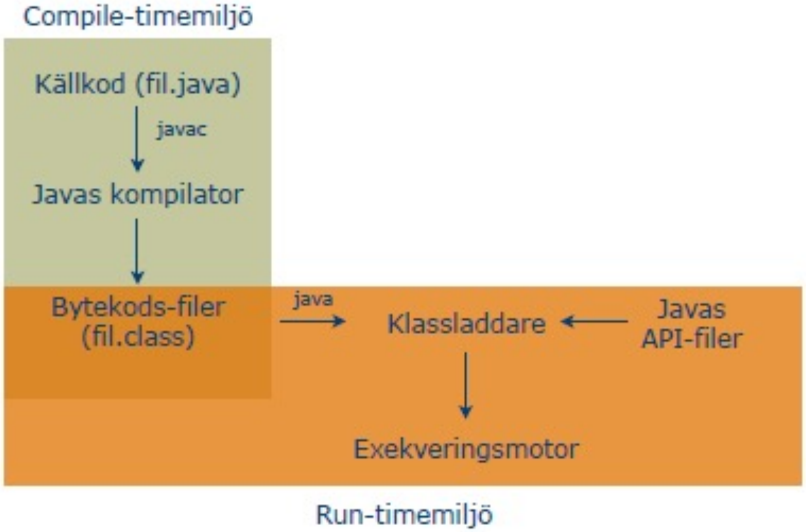
Mode	Value Loaded into AC
Immediate	0x800
Direct	0x900
Indirect	0x 1000
Indexed	0x 500

Interpreterade språk

- Istället för att kompilera ett program till maskinkod kompileras det till ett mellannivåspråk
- Den virtuella maskinen tar mellannivåprogrammet och kör på processorn
- Inetrpreterade språk innebär i regel minskade presentanda och ökad portabilitet
- Ex: Java, Python, C# och etc…

Java

- Släpptes 1995 av Sun Microsystems.
- ”Write once, run anywhere”
- Använder en virtuell maskin, JVM.
- Källkoden kompileras till bytekod.

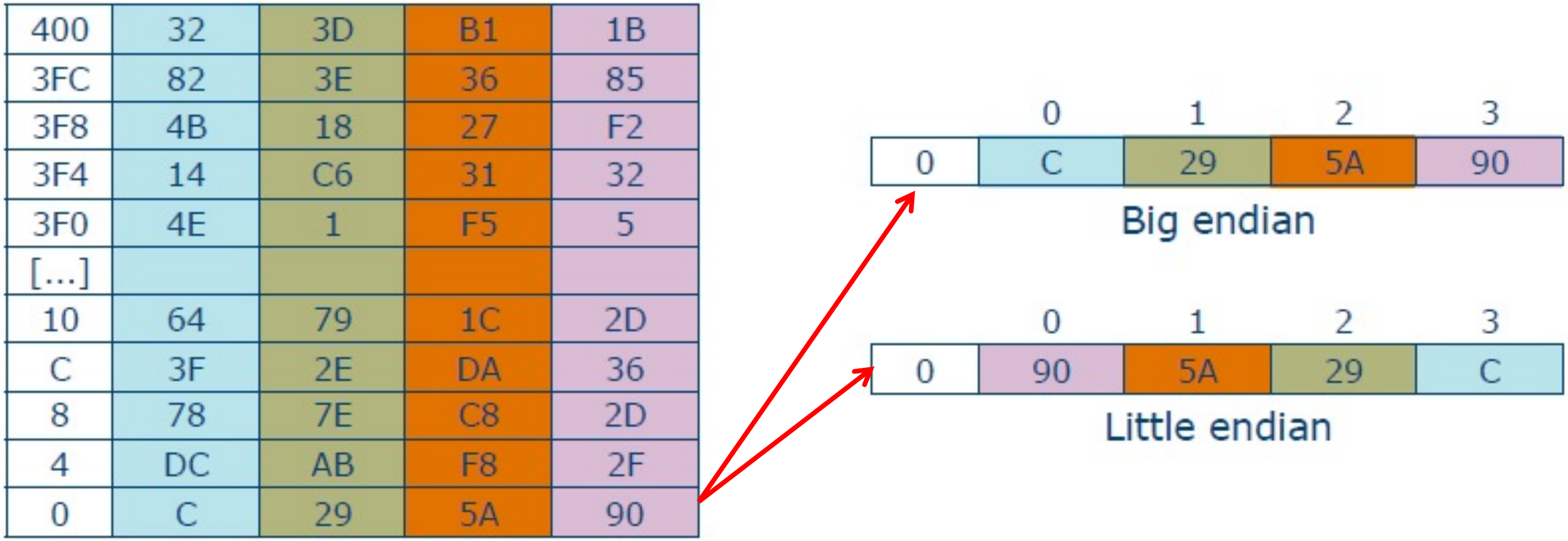


Instruktionsarkitektur i NIOS II

Skillnader mellan instruktionsarkitektur i NIOS II med andra arkitektur

- Antal bitar
- Operandersnas placering
- Instruktioner
- Antal register
- Antal operander
- Endianness

Endianness



400	32	3D	B1	1B
3FC	82	3E	36	85
3F8	4B	18	27	F2
3F4	14	C6	31	32
3F0	4E	1	F5	5
[...]				
10	64	79	1C	2D
C	3F	2E	DA	36
8	78	7E	C8	2D
4	DC	AB	F8	2F
0	C	29	5A	90

	0	1	2	3
0	C	29	5A	90

Big endian

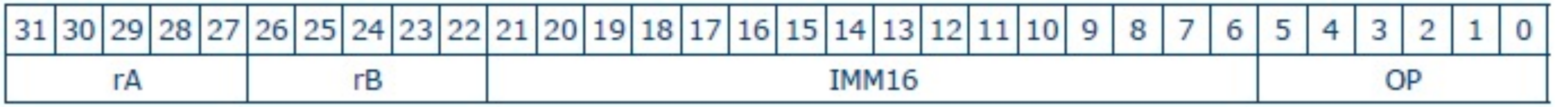
	0	1	2	3
0	90	5A	29	C

Little endian

Instruktionernas längd

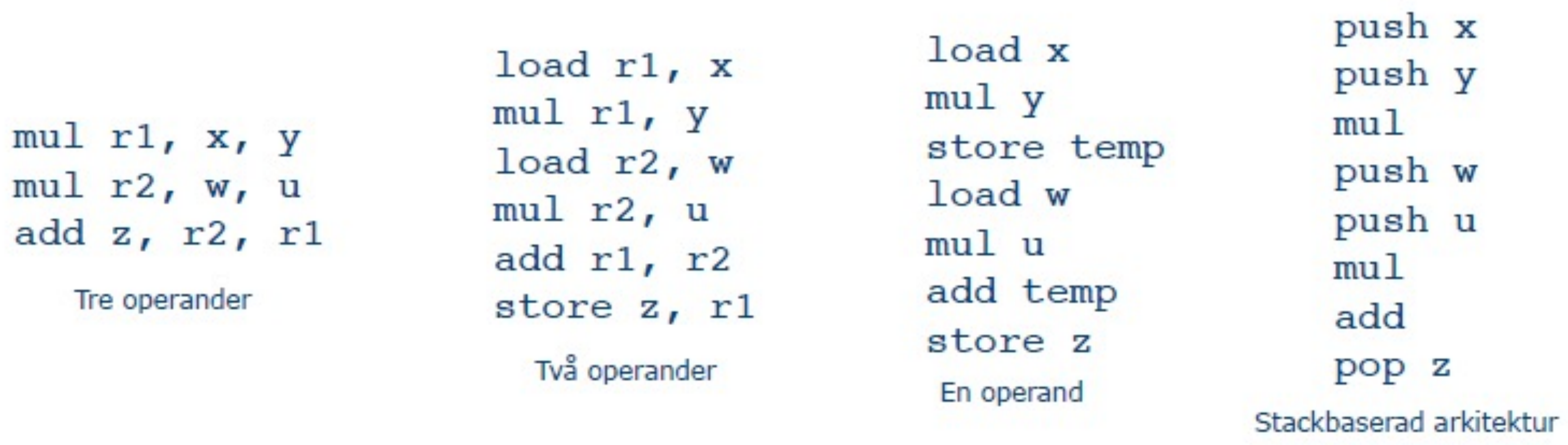
- OP-kodens längd
  - Fast längd
  - Variabel längd
- Registerfält
  - Antal
  - Längd
- Immediatefältets längd

I-Type



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rA					rB					IMM16																OP					

Antalet operander



```
mul r1, x, y
mul r2, w, u
add z, r2, r1
```

Tre operander

```
load r1, x
mul r1, y
load r2, w
mul r2, u
add r1, r2
store z, r1
```

Två operander

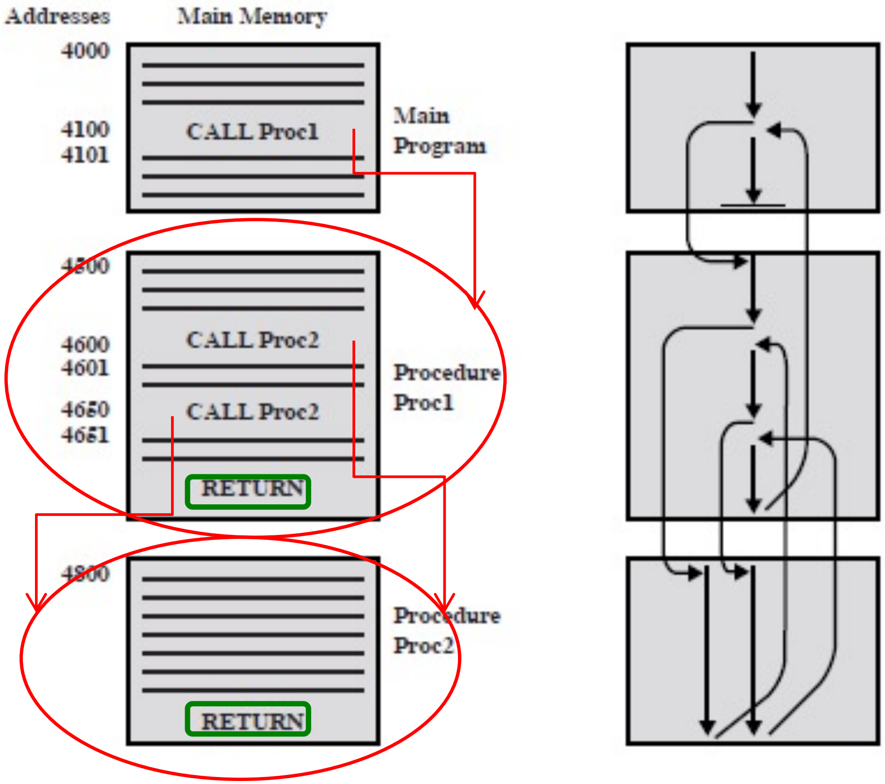
```
load x
mul y
store temp
load w
mul u
add temp
store z
```

En operand

```
push x
push y
mul
push w
push u
mul
add
pop z
```

Stackbaserad arkitektur

Return


Addresses
Main Memory
4000
4100
4101
CALL Proc1
Main Program
4500
4600
4601
4650
4651
CALL Proc2
CALL Proc2
RETURN
Procedure Proc1
4800
RETURN
Procedure Proc2

Ex: Return

```
def add(a, b):
        return a + b


def main():
        addition = add(1, 2)
        subtraktion = addition - 4
```

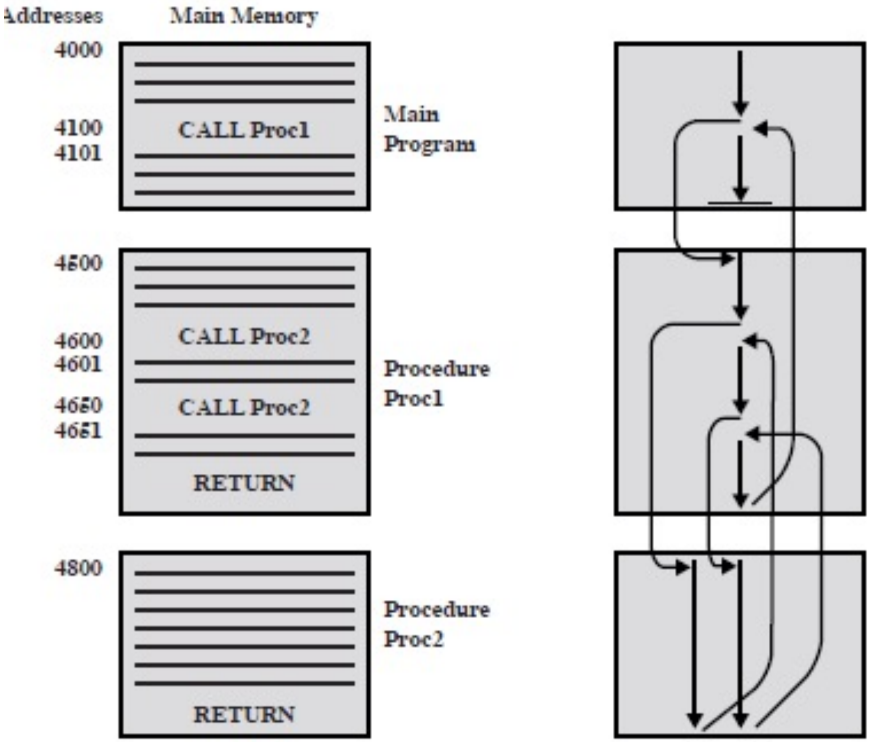
Python-syntax

```
_start:
        movi r2, 1
        movi r3, 2
        call add
        subi r8, r4, 4
add:
        add r4, r2, r3
        ret
```

NIOS II Assembly-syntax

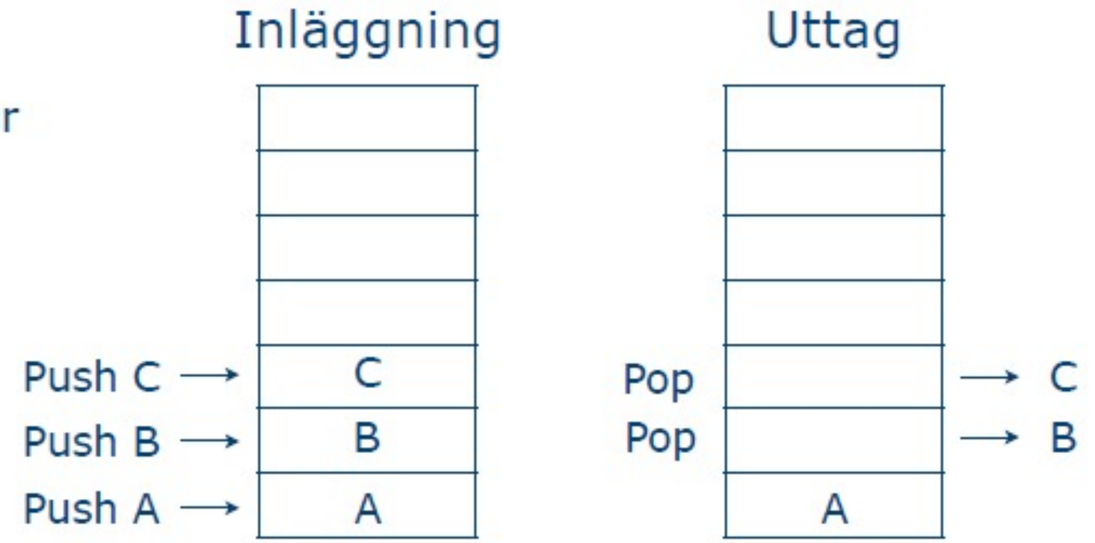
Returnadress

- Returadressregister
  -r31 på Nios II
- Spara returadressen på stacken

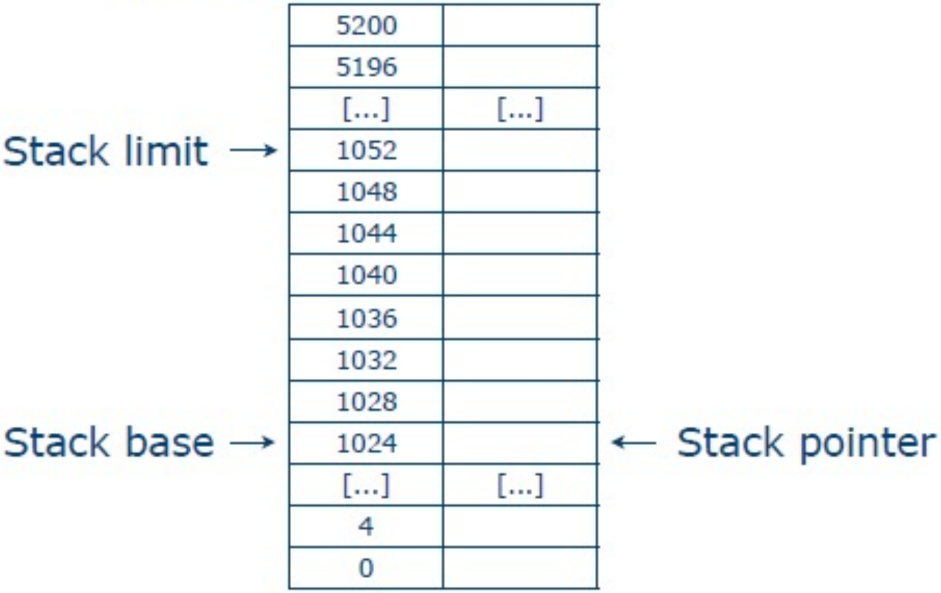


Stack

- Last In First Out (LIFO)-struktur
- Två operationer
  - -Push
  - -Pop



Stacken i minnet

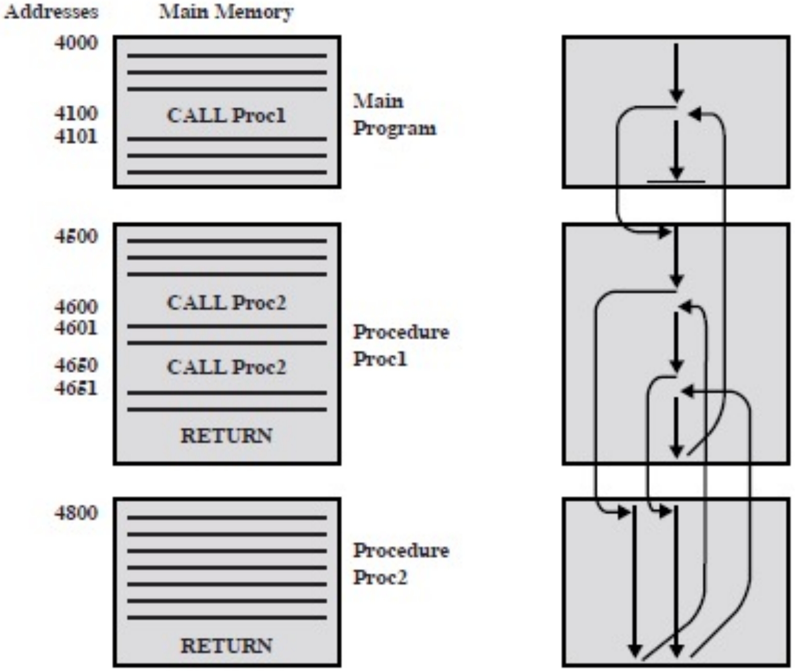


Stack limit →

Stack base →

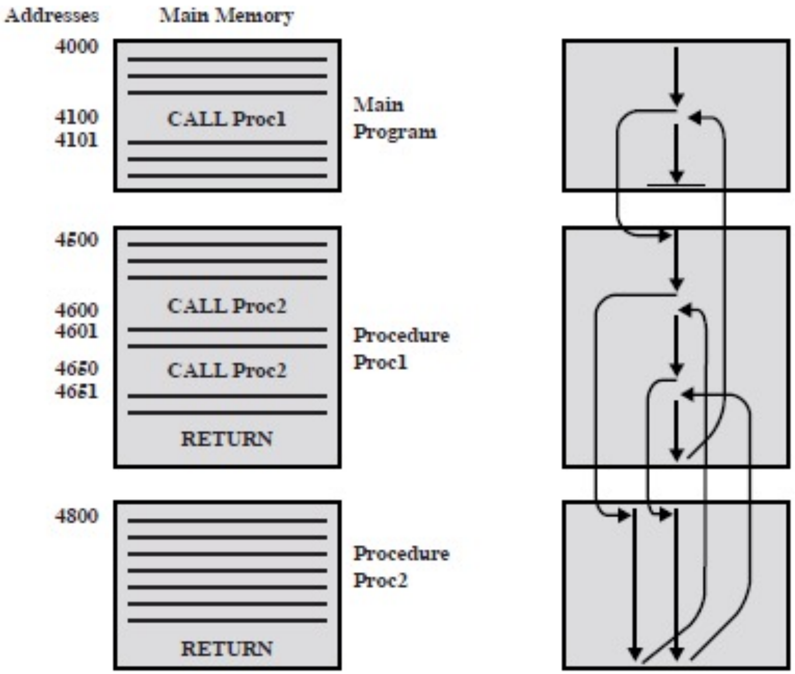
← Stack pointer

5200	
5196	
[...]	[...]
1052	
1048	
1044	
1040	
1036	
1032	
1028	
1024	
[...]	[...]
4	
0	



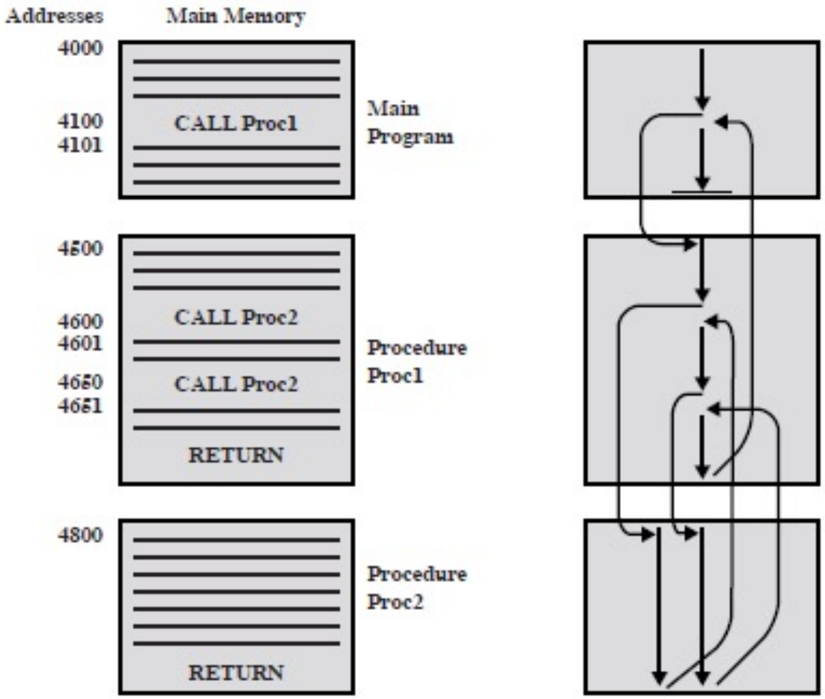
Stacken i minnet

	4096		
	4092		
	[...]	[...]	
Stack limit →	1052		
	1048		
	1044		
	1040		
	1036		
	1032		
	1028		← Stack pointer
Stack base →	1024	4100	
	[...]	[...]	
	4		
	0		



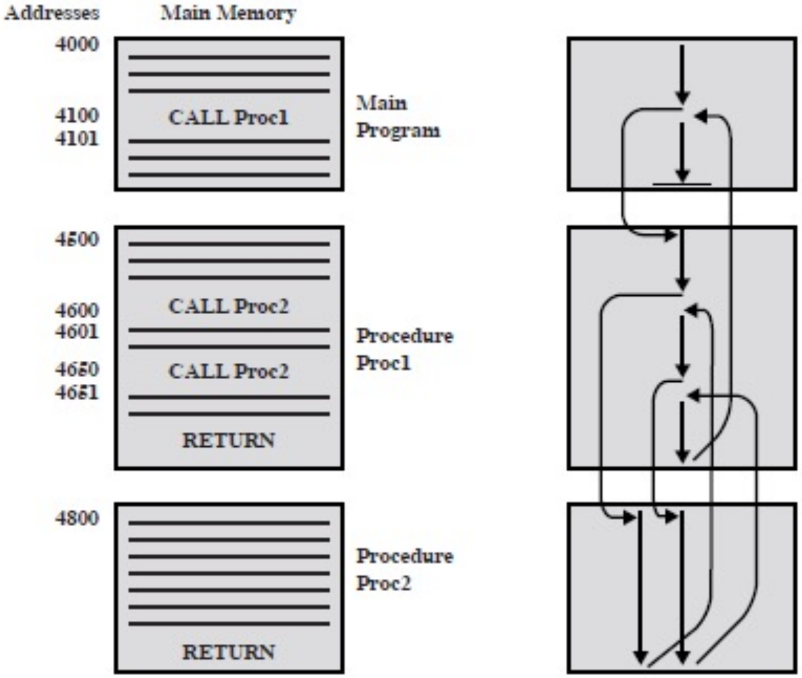
Stacken i minnet

	4096		
	4092		
	[...]	[...]	
Stack limit →	1052		
	1048		
	1044		
	1040		
	1036		
	1032		← Stack pointer
	1028	4600	
Stack base →	1024	4100	
	[...]	[...]	
	4		
	0		



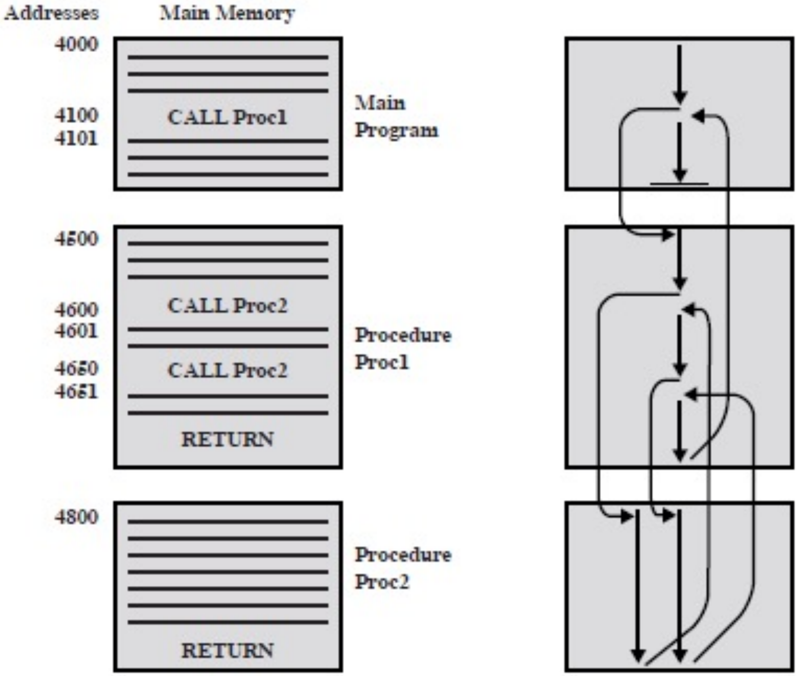
Stacken i minnet

	4096		
	4092		
	[...]	[...]	
Stack limit →	1052		
	1048		
	1044		
	1040		
	1036		
	1032		
	1028		← Stack pointer
Stack base →	1024	4100	
	[...]	[...]	
	4		
	0		

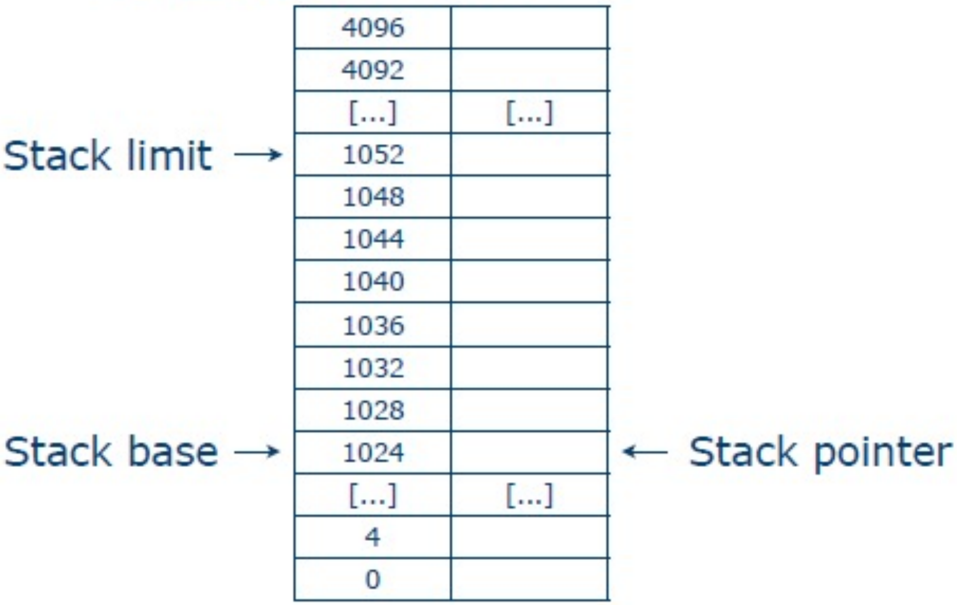


Stacken i minnet

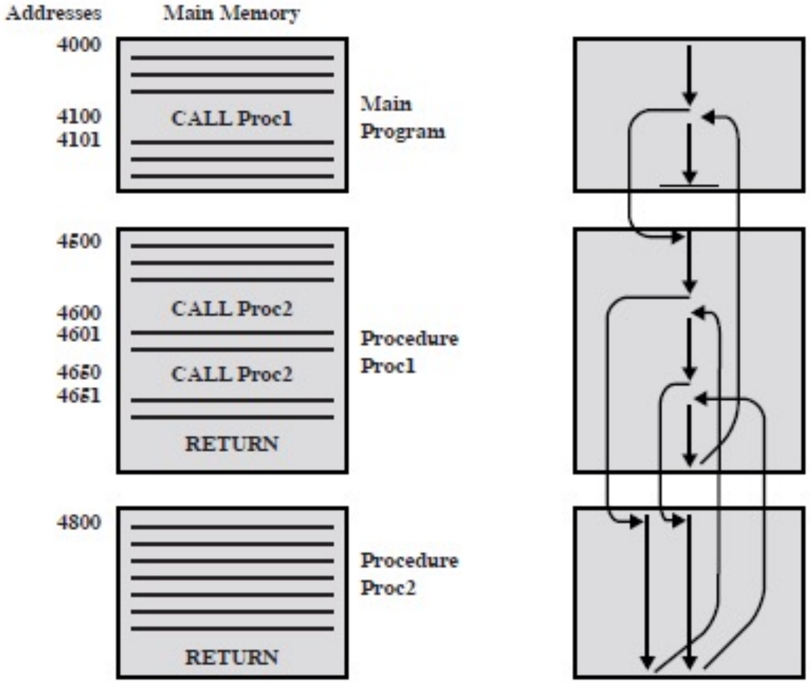
	4096		
	4092		
	[...]	[...]	
Stack limit →	1052		
	1048		
	1044		
	1040		
	1036		
	1032		← Stack pointer
	1028	4650	
Stack base →	1024	4100	
	[...]	[...]	
	4		
	0		



Stacken i minnet



4096	
4092	
[...]	[...]
1052	
1048	
1044	
1040	
1036	
1032	
1028	
1024	
[...]	[...]
4	
0	



Instruction Pipelining

- Some **CPUs divide** the fetch-decode-execute cycle into smaller steps.
- These smaller steps can often be executed in parallel to increase throughput.
- Such parallel execution is called ***instruction pipelining***.
- Instruction pipelining provides for ***instruction level parallelism*** **(*ILP*)**

Instruction Pipelining

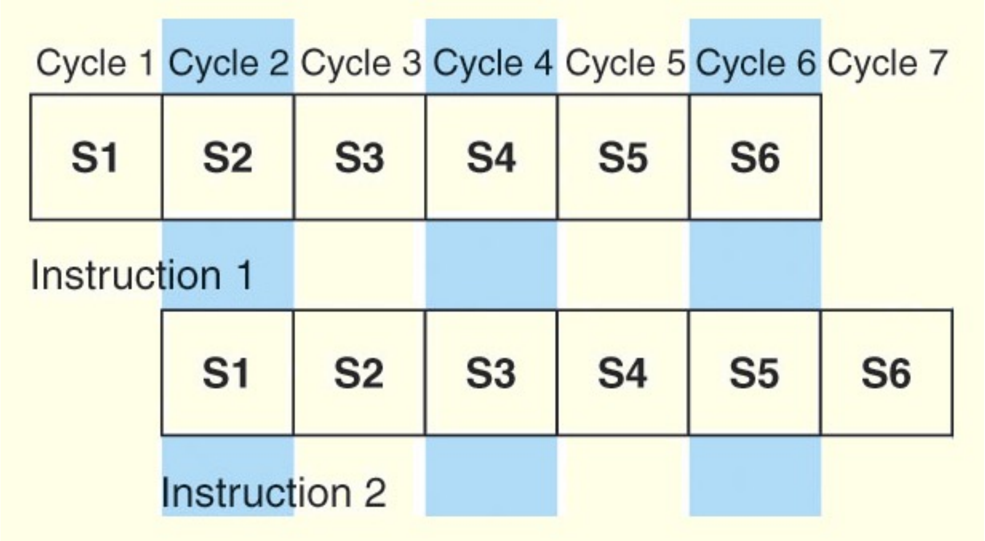
- Suppose a **fetch-decode-execute cycle** were broken into the following smaller steps:

1. **F**etch **I**nstruction(FI).
2. **D**ecode **O**pcode(DO).
3. **C**alculate **E**ffective address of operands(CE)
4. **F**etch **O**perands(FO)
5. **E**xecute **I**nstruction(EI)
6. **S**tore **R**esult (SR).

- Suppose we have a six-stage pipeline. S1 fetches the instruction, S2 decodes it, S3 determines the address of the operands, S4 fetches them, S5 executes the instruction, and S6 stores the result.

Instruction Pipelining

- For every clock cycle, one small step is carried out, and the stages are overlapped.



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
Instruction 1	S1	S2	S3	S4	S5	S6	
Instruction 2		S1	S2	S3	S4	S5	S6

S1. Fetch instruction.
S2. Decode opcode.
S3. Calculate effective address of operands

S4. Fetch operands.
S5. Execute.
S6. Store result

Instruction Pipelining

- The theoretical speedup offered by a pipeline can be determined as follows:

  Let t_p be the time per stage. Each instruction represents a task, T, in the pipeline.

  The first task (instruction) requires $k \times t_p$ time to complete in a k-stage pipeline. The remaining $(n - 1)$ tasks emerge from the pipeline one per cycle. So the **total time** to complete the remaining tasks is $(n - 1)t_p$.

  Thus, to complete ***n* tasks** using a k-stage pipeline requires:

$$(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p$$

Instruction Pipelining

- If we take the time required to complete *n* tasks without a pipeline and divide it by the time it takes to complete *n* tasks using a pipeline, we find:

$$\text{Speedup } S = \frac{n t_n}{(k + n - 1)\, t_p}$$

- If we take the limit of this as *n* approaches infinity, we see that (*k* + *n* - 1) approaches *n*, which results in a theoretical speedup of:

$$\text{Speedup } S = \frac{k t_p}{t_p} = k$$

Instruction Pipelining

- Our next equations take a number of things for granted.
- **First,** we have to assume that the architecture supports fetching instructions and data in parallel.
- **Second**, we assume that the pipeline can be kept filled at all times. This is not always the case. **Pipeline *hazards*** arise that cause pipeline conflicts and stalls.

Instruction Pipelining

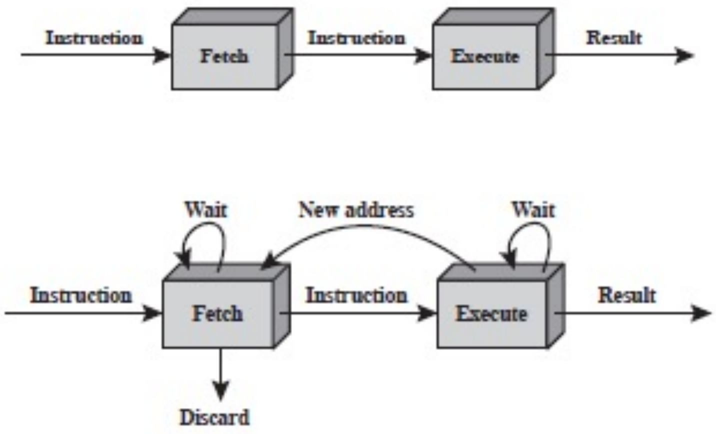
- An instruction **pipeline may stall**, or be flushed for any of the following reasons:
  - **Resource conflicts**
  - **Data dependencies**
  - **Conditional branching**
- Measures can be taken at the software level as well as at the hardware level to reduce the effects of these hazards, but they cannot be totally eliminated.

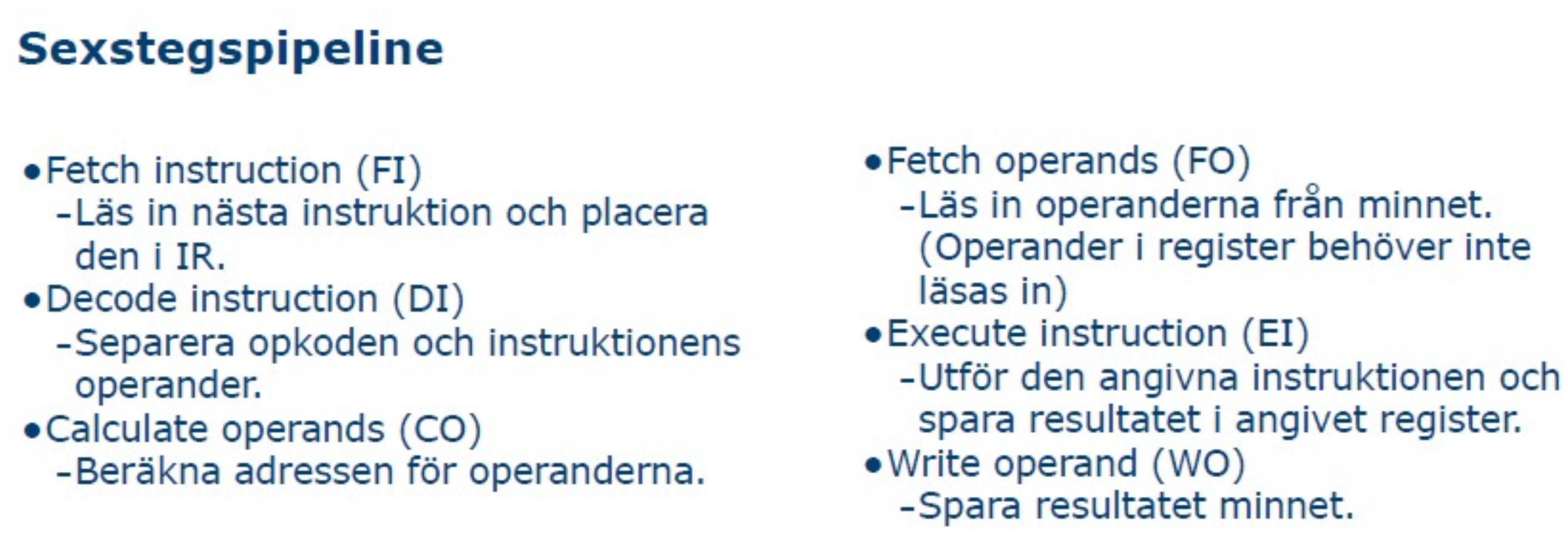
Vilka konflikter kan uppstå i en pipline?

- **Strukturella konflikter:** Uppstår när 2 pipelinade instruktioner försöker använda en hårdvarukomponent(t.ex. primärminnet)) samtidigt.
- **Data konflikter:** Uppstår när en instruktion påverkas av resultat från en annan instruktion som ännu inte är färdig.
- **Kontrollkonflikter:** Uppstår när en conditional branch instruktion felaktigt hoppas över. Det Kan ske när branch–condition:et -> beror på resultatet från en pipline:ad instruktion som inte är färdig.

Pipelineing

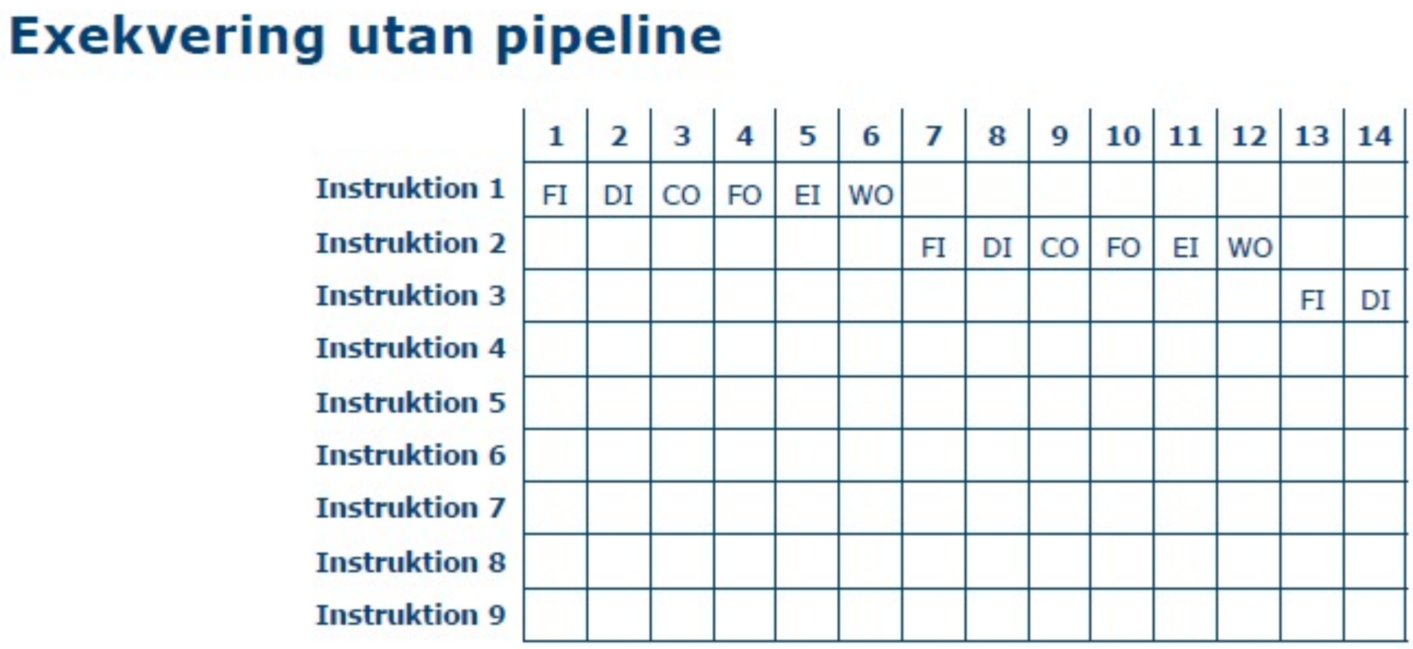
- Ökar prestandan genom att låta processorn utföra flera steg samtidigt.
- Enklaste formen: Processorn utför fetch och execute simultant.
  - Instruction prefetch / fetch overlap
- Hinder för prestandaökning:
  - Execute-steget kommer att ta längre tid än fetch-steget.
  - En villkorad branch innebär att adressen för nästkommande instruktion är okänd.





Sexstegspipeline

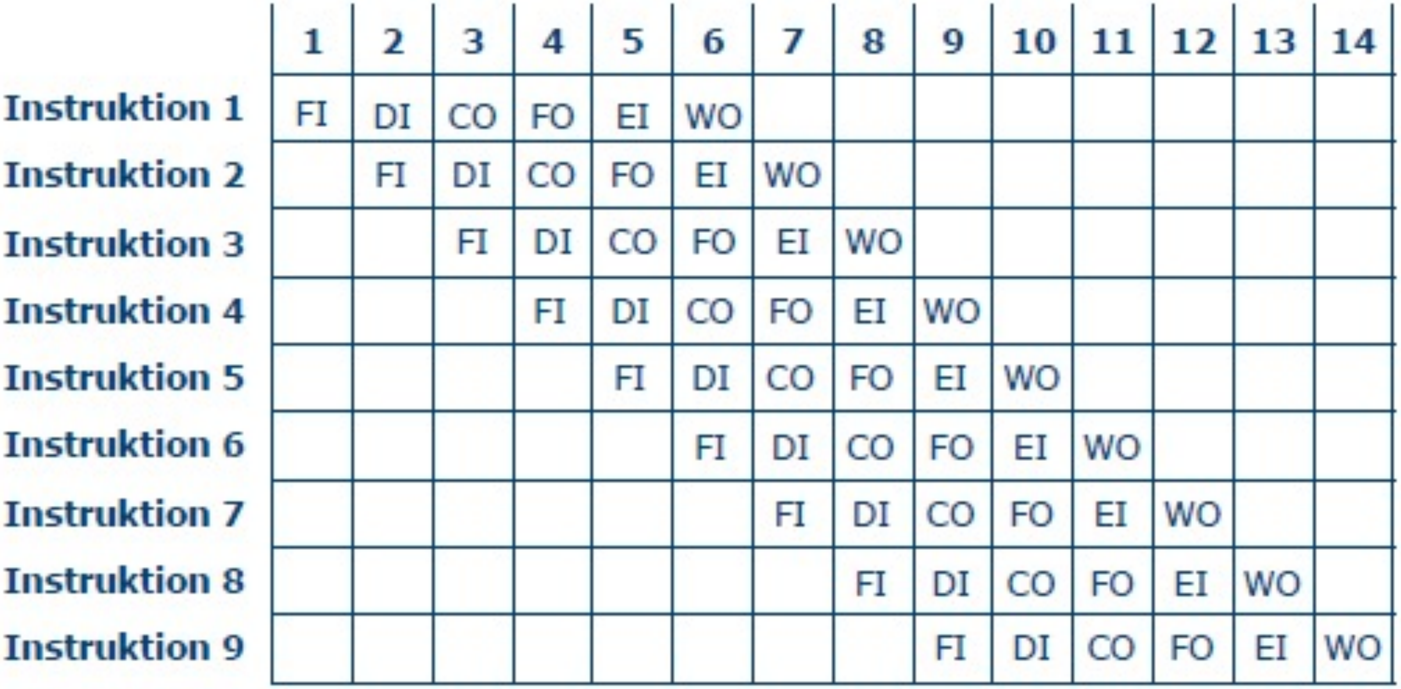
- Fetch instruction (FI)
  - Läs in nästa instruktion och placera den i IR.
- Decode instruction (DI)
  - Separera opkoden och instruktionens operander.
- Calculate operands (CO)
  - Beräkna adressen för operanderna.
- Fetch operands (FO)
  - Läs in operanderna från minnet. (Operander i register behöver inte läsas in)
- Execute instruction (EI)
  - Utför den angivna instruktionen och spara resultatet i angivet register.
- Write operand (WO)
  - Spara resultatet minnet.



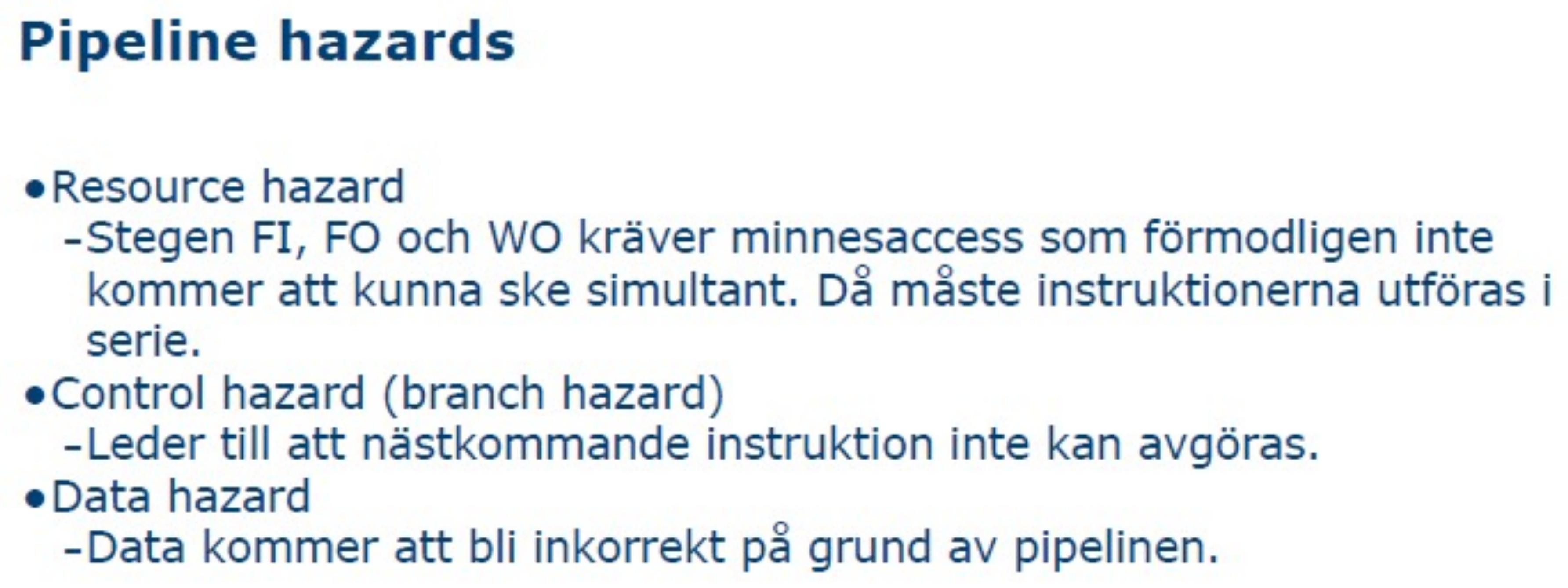
Exekvering utan pipeline

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruktion 1	FI	DI	CO	FO	EI	WO								
Instruktion 2							FI	DI	CO	FO	EI	WO		
Instruktion 3													FI	DI
Instruktion 4														
Instruktion 5														
Instruktion 6														
Instruktion 7														
Instruktion 8														
Instruktion 9														

Exekvering med pipeline



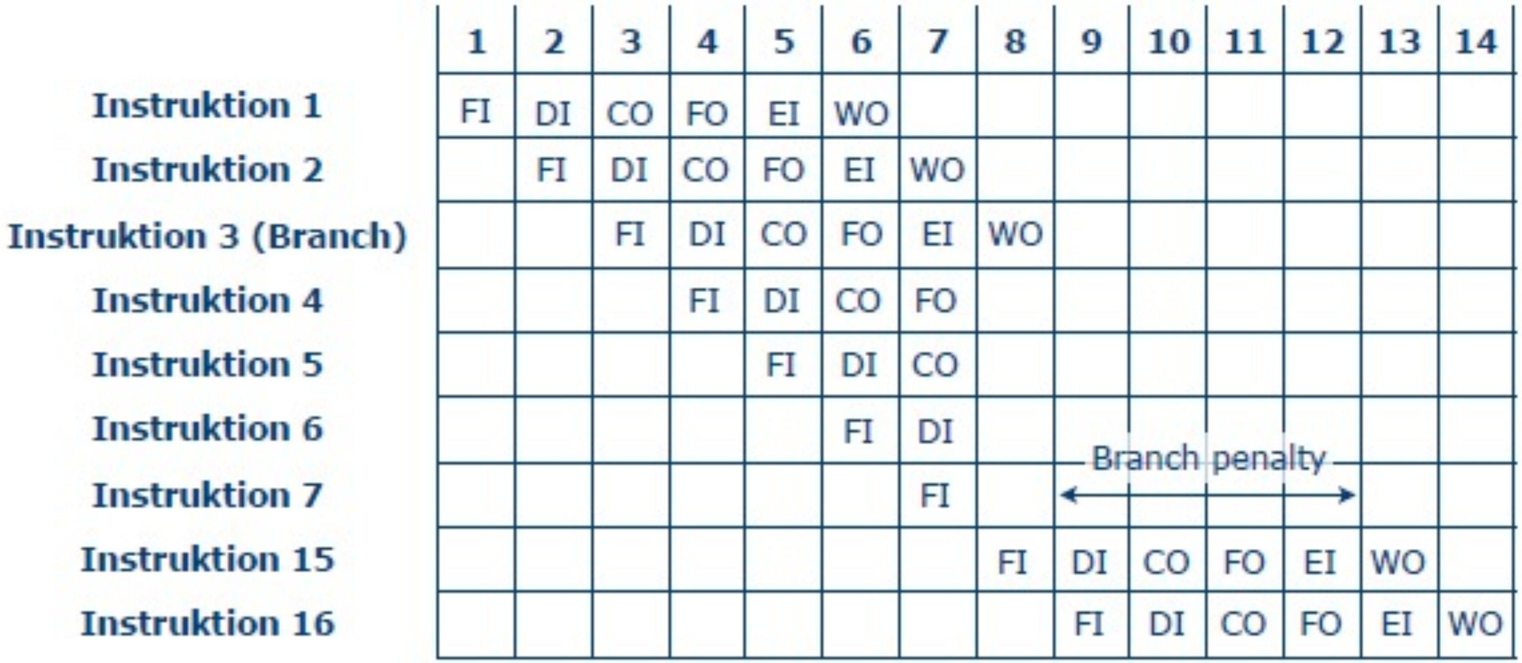
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruktion 1	FI	DI	CO	FO	EI	WO								
Instruktion 2		FI	DI	CO	FO	EI	WO							
Instruktion 3			FI	DI	CO	FO	EI	WO						
Instruktion 4				FI	DI	CO	FO	EI	WO					
Instruktion 5					FI	DI	CO	FO	EI	WO				
Instruktion 6						FI	DI	CO	FO	EI	WO			
Instruktion 7							FI	DI	CO	FO	EI	WO		
Instruktion 8								FI	DI	CO	FO	EI	WO	
Instruktion 9									FI	DI	CO	FO	EI	WO



Pipeline hazards

- Resource hazard
  - Stegen FI, FO och WO kräver minnesaccess som förmodligen inte kommer att kunna ske simultant. Då måste instruktionerna utföras i serie.
- Control hazard (branch hazard)
  - Leder till att nästkommande instruktion inte kan avgöras.
- Data hazard
  - Data kommer att bli inkorrekt på grund av pipelinen.

Pipeline med villkorad branch



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruktion 1	FI	DI	CO	FO	EI	WO								
Instruktion 2		FI	DI	CO	FO	EI	WO							
Instruktion 3 (Branch)			FI	DI	CO	FO	EI	WO						
Instruktion 4				FI	DI	CO	FO							
Instruktion 5					FI	DI	CO							
Instruktion 6						FI	DI							
Instruktion 7							FI							
Instruktion 15								FI	DI	CO	FO	EI	WO	
Instruktion 16									FI	DI	CO	FO	EI	WO

Branch penalty

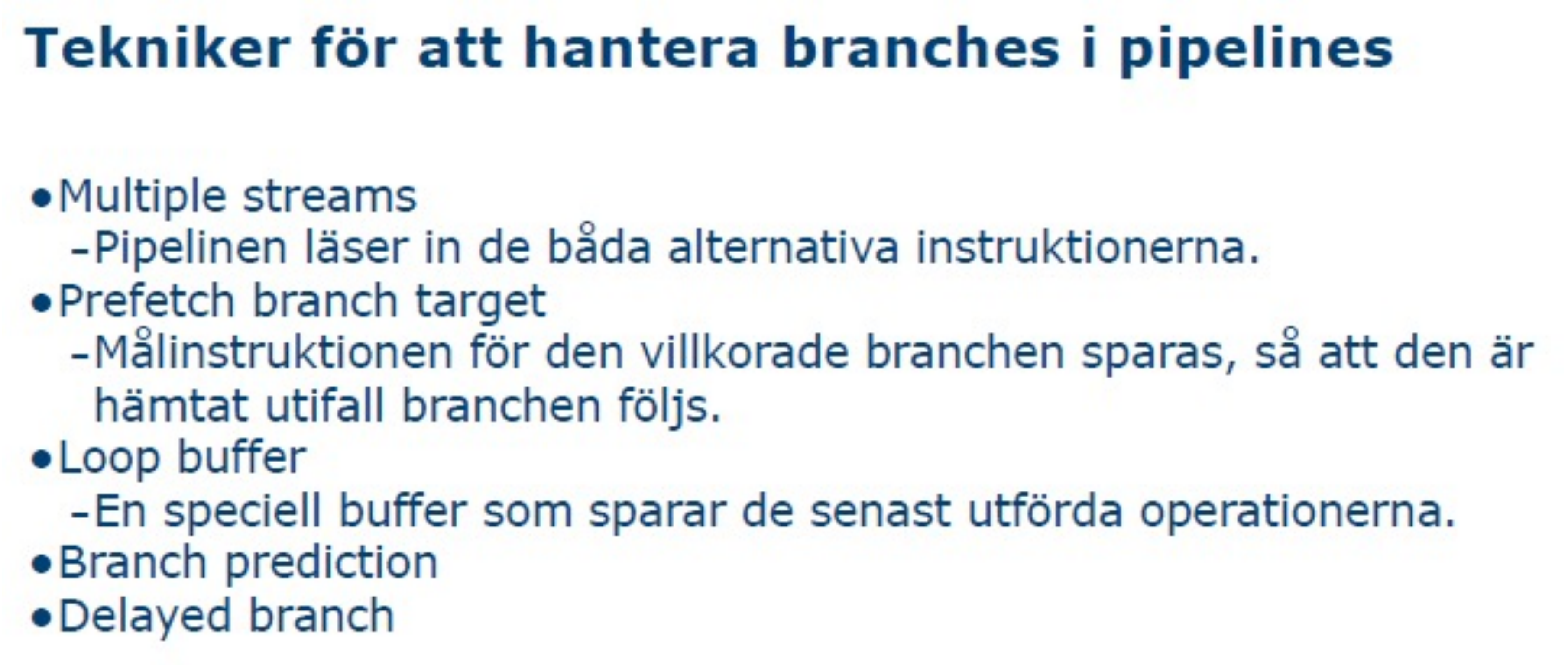
Data hazard

- Read after Write
- Write after Read
- Write after Write

	1	2	3	4	5	6	7
add r4, r5, r6	FI	DI	CO	FO	EI	WO	
sub r7, r4, r8		FI	DI	CO	FO	EI	WO

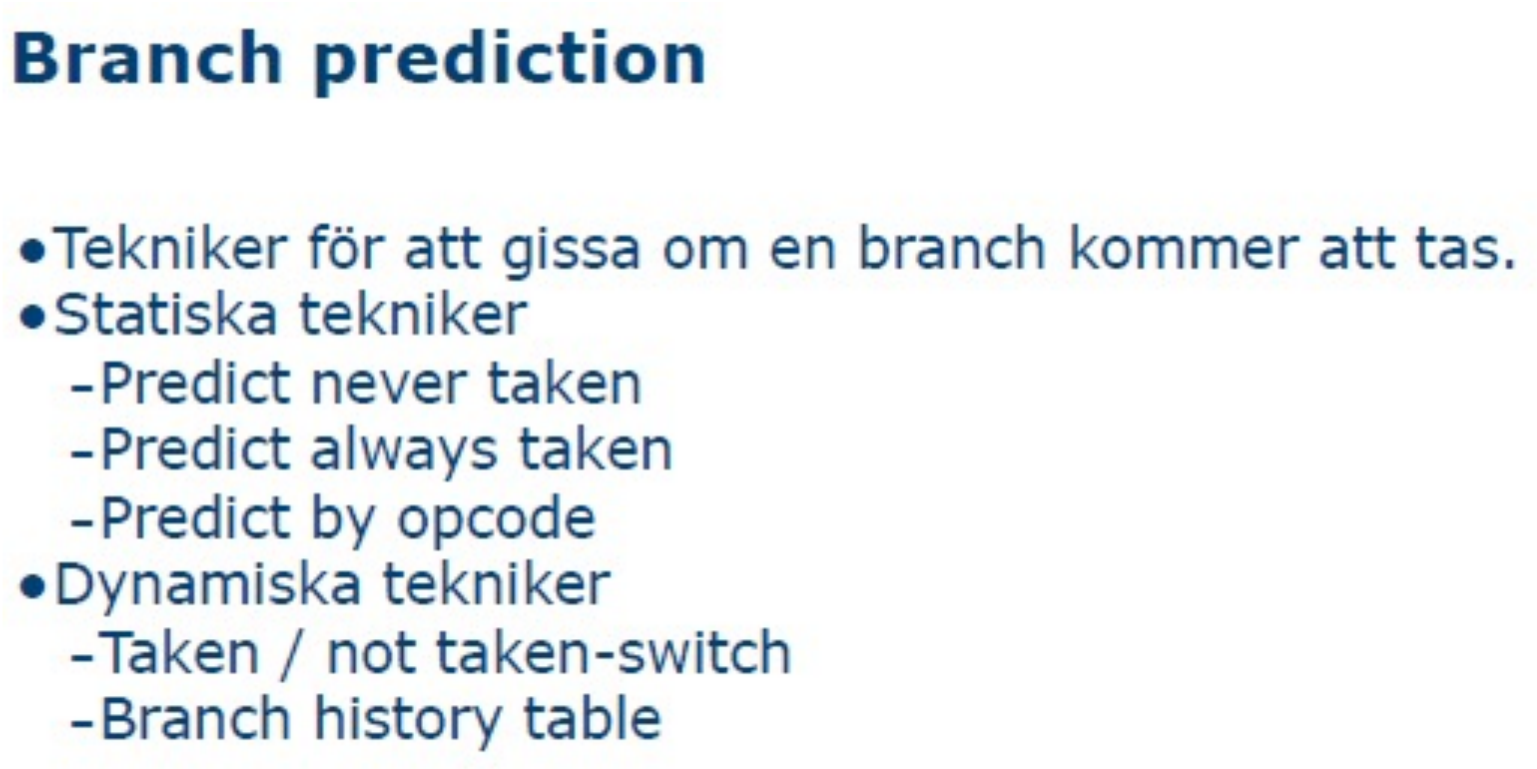
Read after Write

	1	2	3	4	5	6	7	8	9
add r4, r5, r6	FI	DI	CO	FO	EI	WO			
sub r7, r4, r8		FI	DI	CO	Idle		FO	EI	WO



Tekniker för att hantera branches i pipelines

- Multiple streams
  - Pipelinen läser in de båda alternativa instruktionerna.
- Prefetch branch target
  - Målinstruktionen för den villkorade branchen sparas, så att den är hämtat utifall branchen följs.
- Loop buffer
  - En speciell buffer som sparar de senast utförda operationerna.
- Branch prediction
- Delayed branch



Branch prediction

- Tekniker för att gissa om en branch kommer att tas.
- Statiska tekniker
  - Predict never taken
  - Predict always taken
  - Predict by opcode
- Dynamiska tekniker
  - Taken / not taken-switch
  - Branch history table

Exempel på delayed branch

- Under vissa omständigheter kan man öka pipelinens prestanda genom att ändra ordningen som instruktionerna kommer i.

ldb r5, 0(r4)	I	E	D				
addi r5, r5, 1		I	E				
br go_ahead			I	E			
add r5, r5, r6				I	E		
stb r5, 0(r4)					I	E	D

Traditionell pipeline

ldb r5, 0(r4)	I	E	D				
addi r5, r5, 1		I	E				
br go_ahead			I	E			
noop				I	E		
stb r5, 0(r4)					I	E	D

Pipeline med noop

ldb r5, 0(r4)	I	E	D				
br go_ahead		I	E				
addi r5, r5, 1			I	E			
stb r5, 0(r4)				I	E	D	

Delayed branch

Summary

- Instruction Formats
- Instruction types
- Addressing
- Instruction Pipelining
- 4 stegs CPU Konsekvent synkronism
- 6 stegspipeline
- konflikter i pipline
- Data Hazard