

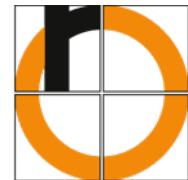


# Prof. Dr. Florian Künzner

Technical University of Applied Sciences Rosenheim, Computer Science

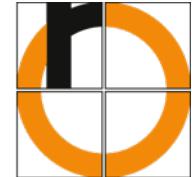
## CA 7 – Processor 4

The lecture is based on the work and the documents of Prof. Dr. Theodor Tempelmeier



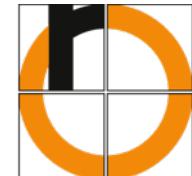
# Goal





# Modern CPU techniques

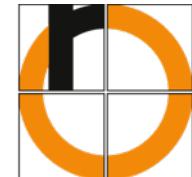
Why are modern (single core) processors **so fast** nowadays?



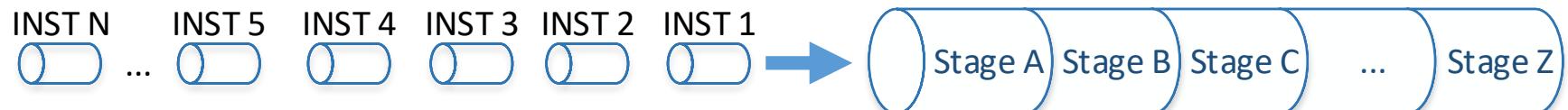
# Goal

## CA::Processor 4

- Pipelining
- Instruction scheduling
- Superscalar architecture
- VLIW
- Out-of-order memory access



# Pipelining

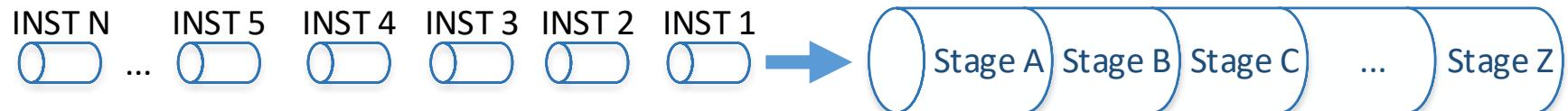


Idea of pipelining:

Pipelining is done in hardware!



# Pipelining



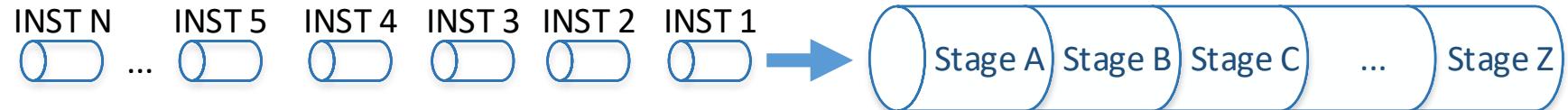
## Idea of pipelining:

- Splitting an instruction processing cycle into individual stages
- Processing of these stages in subsequent hardware stations
- Instructions run through the individual stages
- Each stage does a partial processing of an instruction
- Time overlap of instruction processing!

Pipelining is done in hardware!



# Pipelining



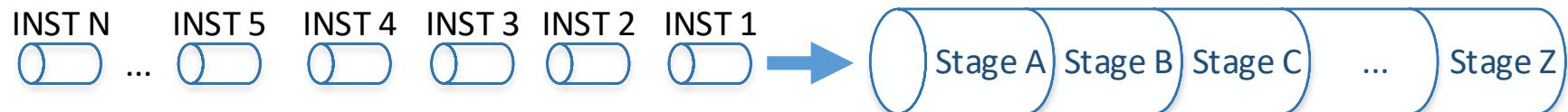
## Idea of pipelining:

- Splitting an instruction processing cycle into individual stages
- Processing of these stages in subsequent hardware stations
- Instructions run through the individual stages
- Each stage does a partial processing of an instruction
- Time overlap of instruction processing!

Pipelining is done in hardware!



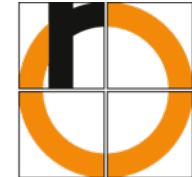
# Pipelining



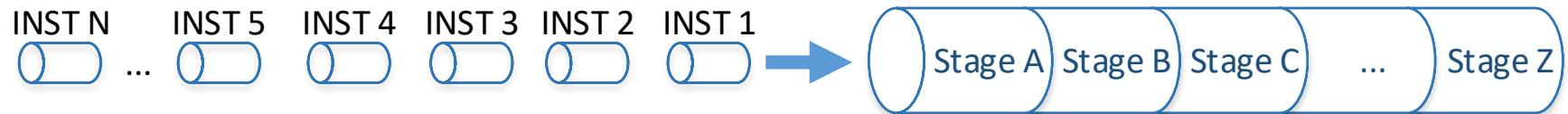
## Idea of pipelining:

- Splitting an instruction processing cycle into individual stages
- Processing of these stages in subsequent hardware stations
- Instructions run through the individual stages
- Each stage does a partial processing of an instruction
- Time overlap of instruction processing!

Pipelining is done in hardware!



# Pipelining



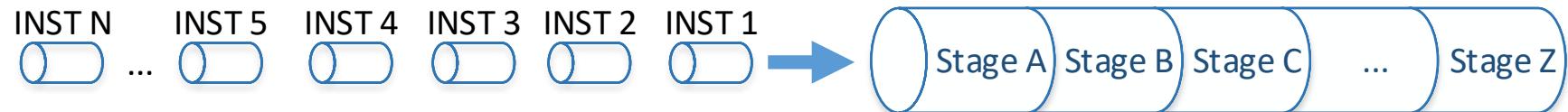
## Idea of pipelining:

- Splitting an instruction processing cycle into individual stages
- Processing of these stages in subsequent hardware stations
- Instructions run through the individual stages
- Each stage does a partial processing of an instruction
- Time overlap of instruction processing!

Pipelining is done in hardware!



# Pipelining



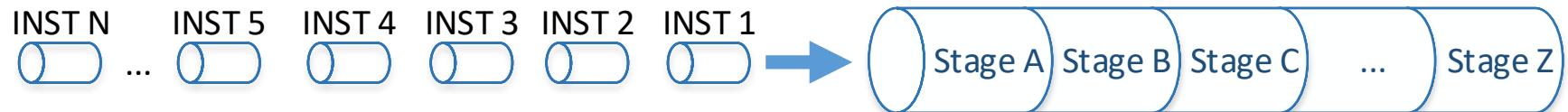
## Idea of pipelining:

- Splitting an instruction processing cycle into individual stages
- Processing of these stages in subsequent hardware stations
- Instructions run through the individual stages
- Each stage does a partial processing of an instruction
- Time overlap of instruction processing!

Pipelining is done in hardware!



# Pipelining

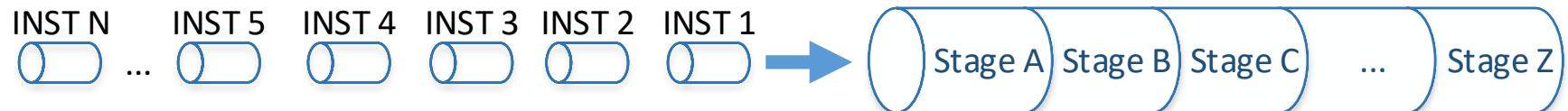


## Idea of pipelining:

- Splitting an instruction processing cycle into individual stages
- Processing of these stages in subsequent hardware stations
- Instructions run through the individual stages
- Each stage does a partial processing of an instruction
- Time overlap of instruction processing!

Pipelining is done in hardware!

# Pipelining



## Idea of pipelining:

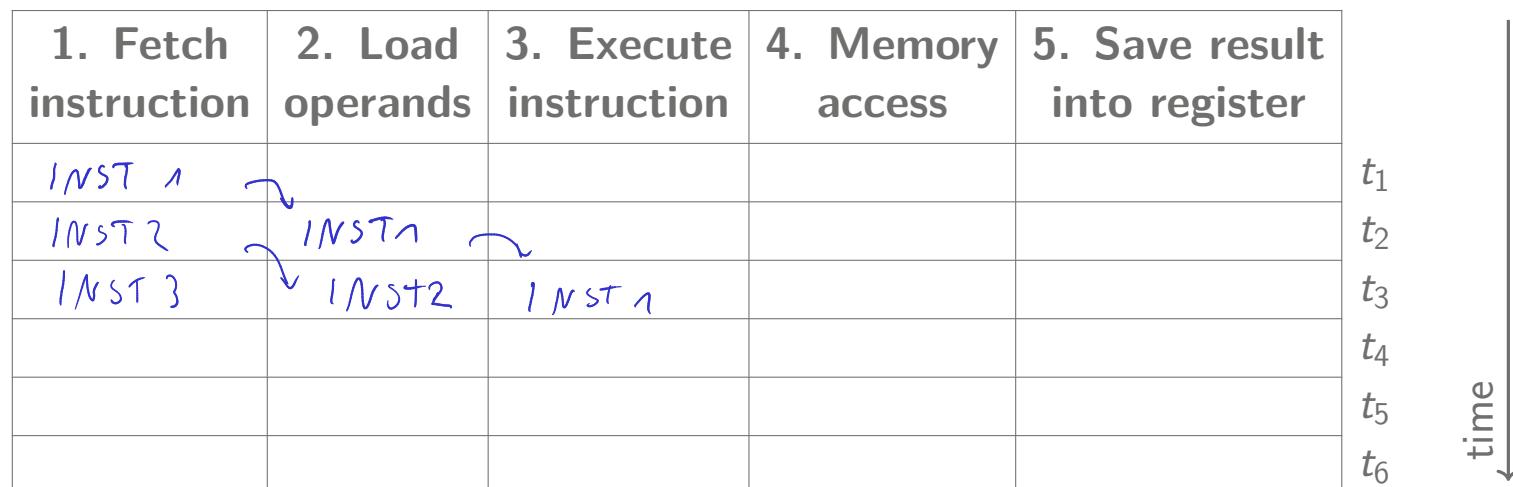
- Splitting an instruction processing cycle into individual stages
- Processing of these stages in subsequent hardware stations
- Instructions run through the individual stages
- Each stage does a partial processing of an instruction
- Time overlap of instruction processing!

**Pipelining is done in hardware!**



# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.



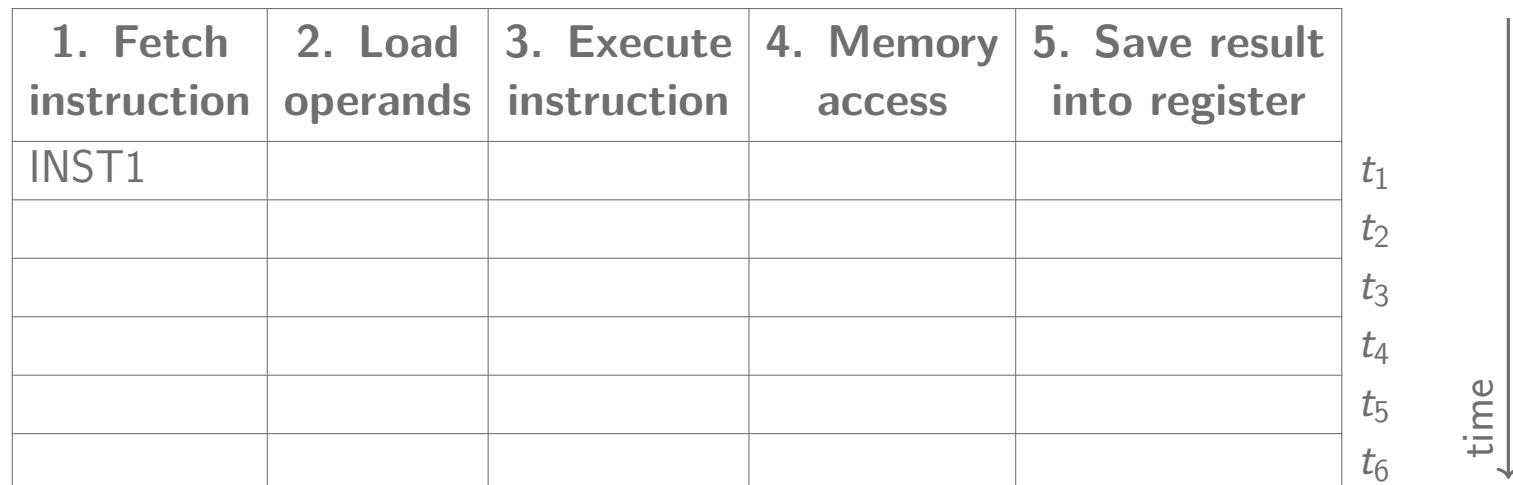
## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended



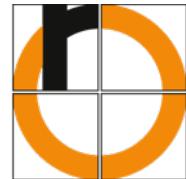
# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.



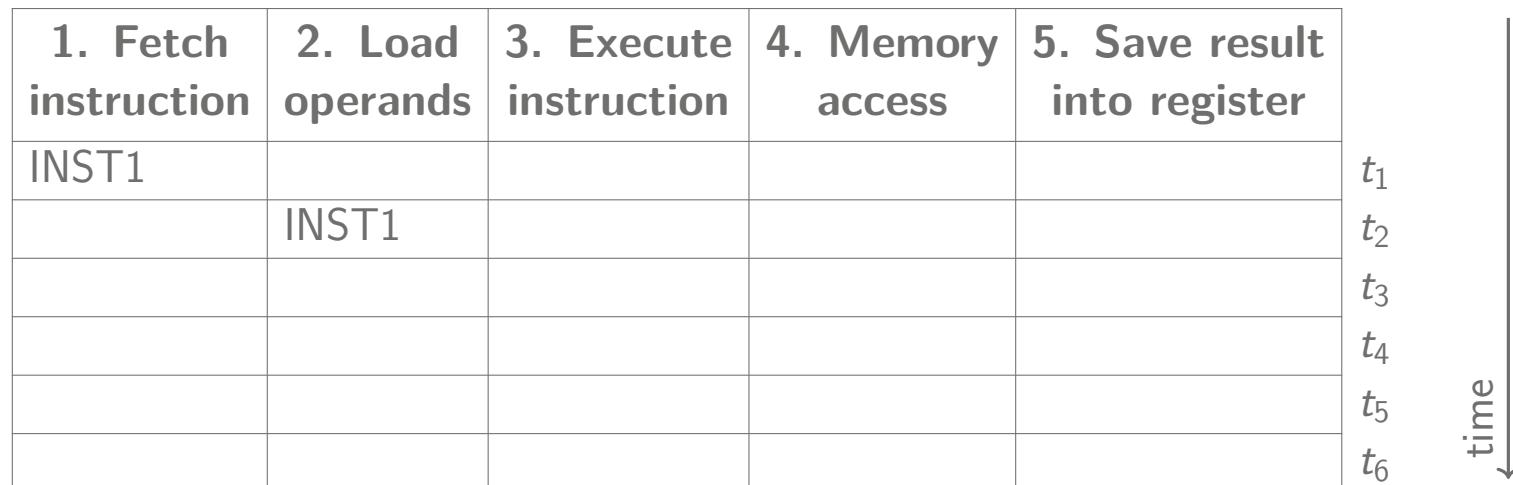
## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended



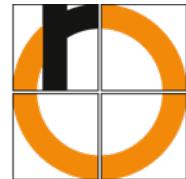
# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.



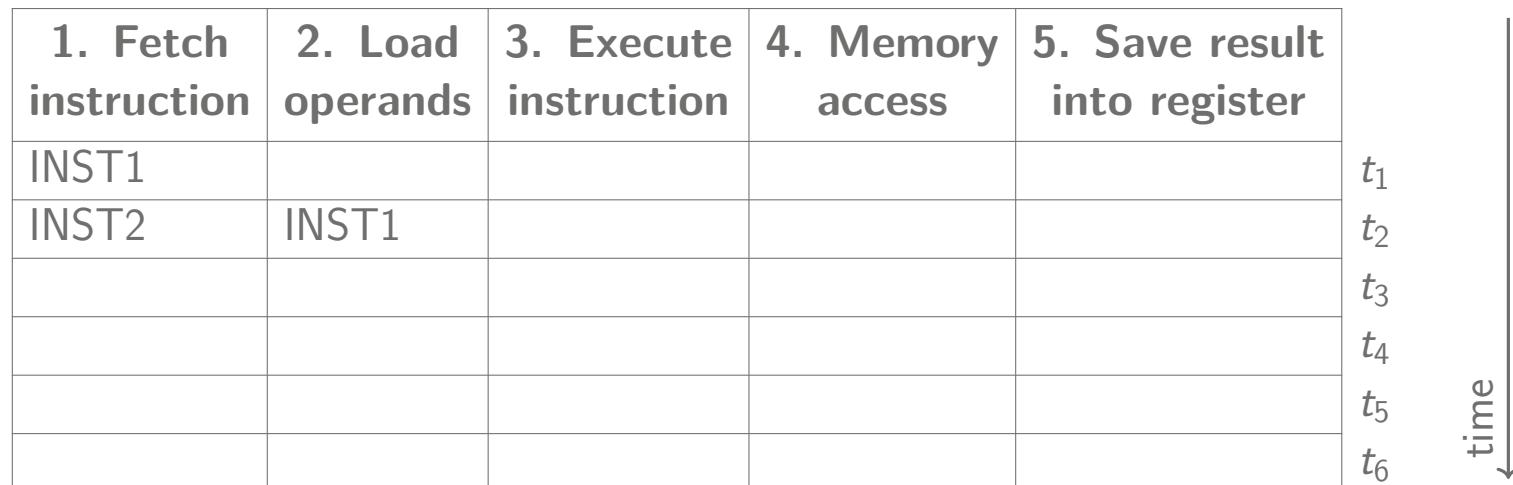
## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended



# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.



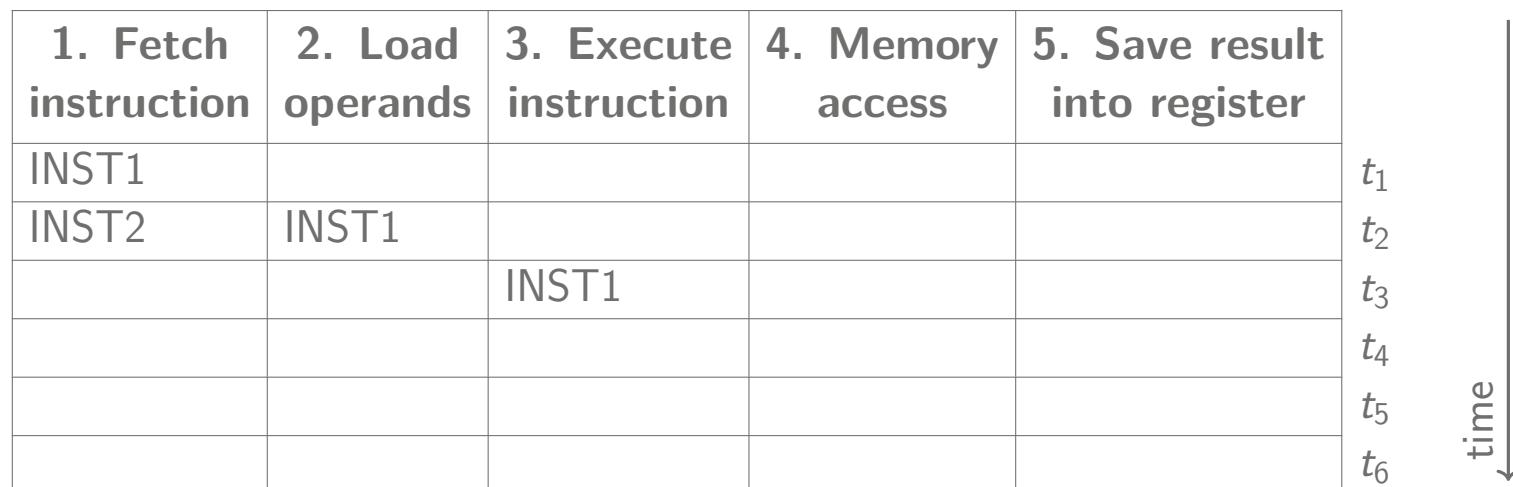
## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended



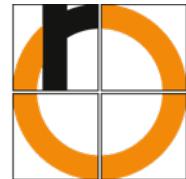
# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.



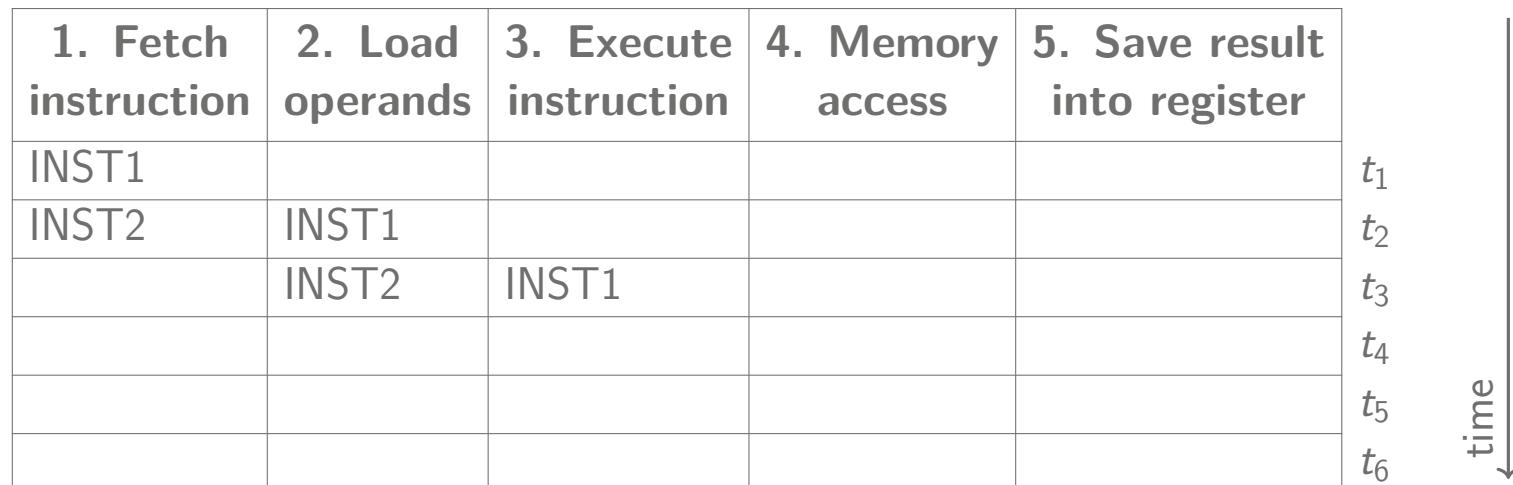
## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended



# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.



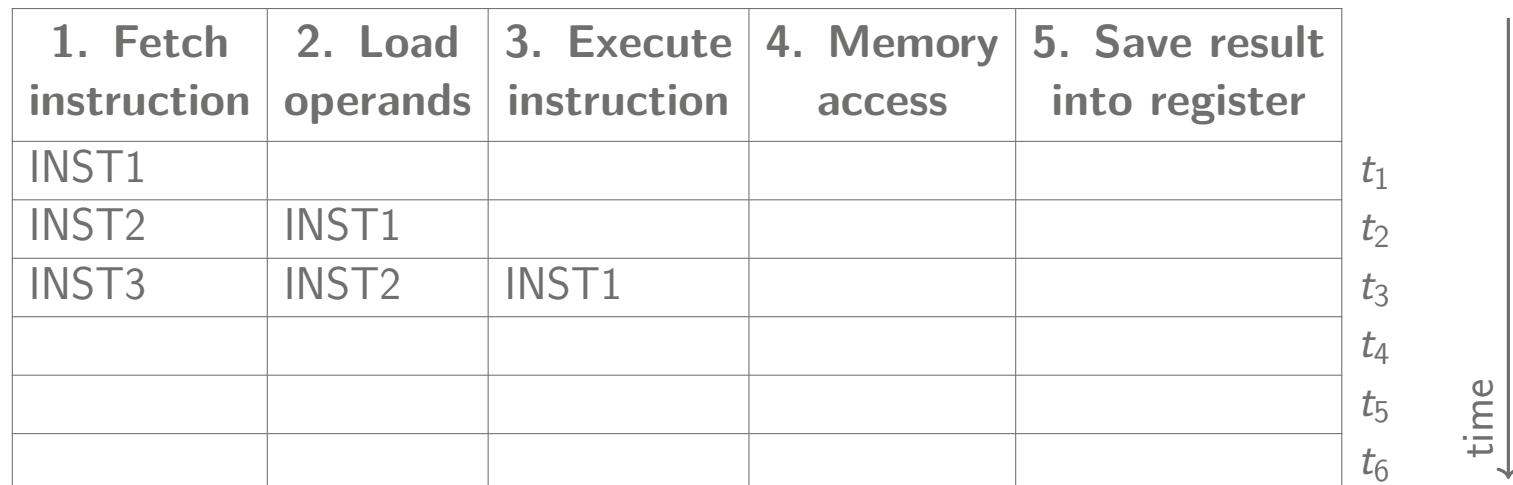
## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended



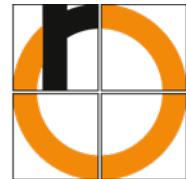
# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.



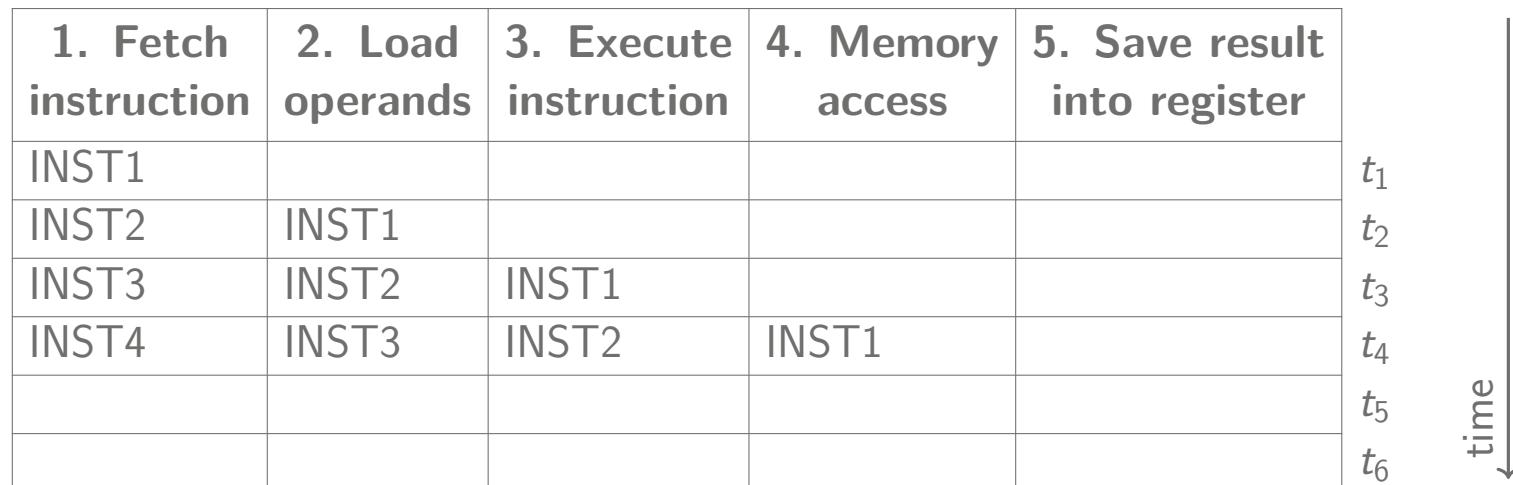
## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended



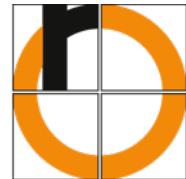
# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.



## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended



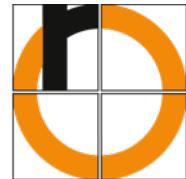
# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.

1. Fetch instruction	2. Load operands	3. Execute instruction	4. Memory access	5. Save result into register	time ↓
INST1					$t_1$
INST2	INST1				$t_2$
INST3	INST2	INST1			$t_3$
INST4	INST3	INST2	INST1		$t_4$
INST5	INST4	INST3	INST2	INST1	$t_5$
					$t_6$

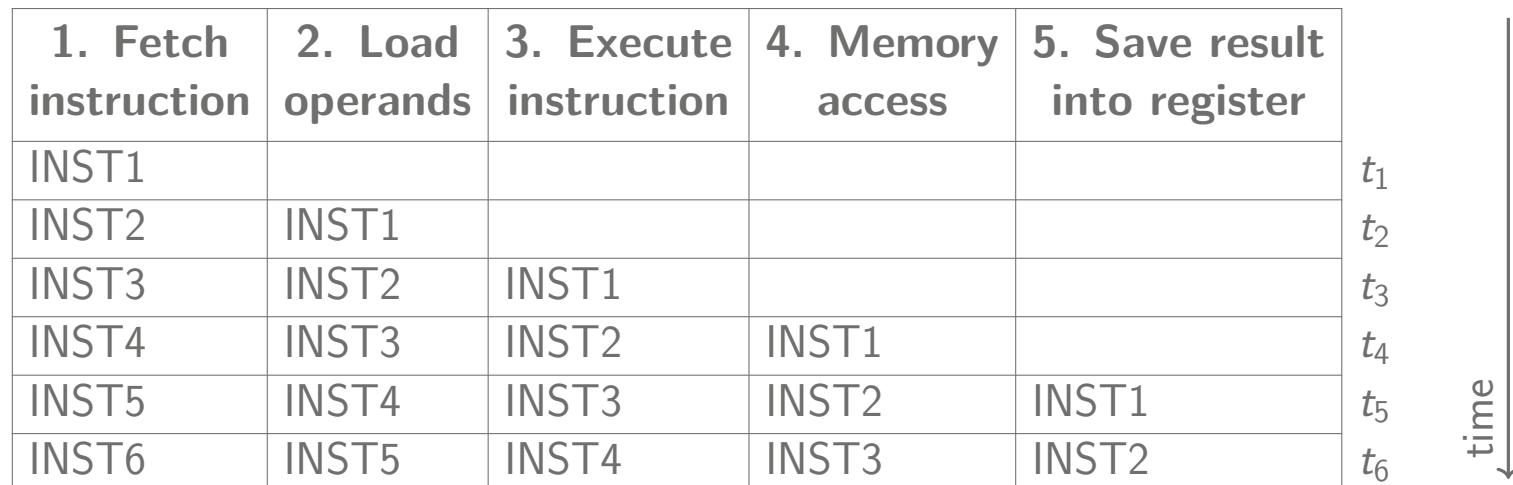
## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended



# Pipelining - example

This is a very simplified 5-stage (classical RISC) pipeline, since modern processors contain many more pipeline stages.



## Processing

- The processing of an instruction takes 5 clock cycles
- On each clock cycle one instruction is started and one ends
- Without pipelining: Every 5 clock cycles, an instruction would be started and ended

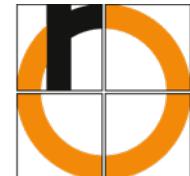


# Pipelining - problems

The execution of instructions in a pipeline is disturbed by:

- Jump instructions
- Interrupts
- Data dependencies (the next instruction requires data from the previous)

Special hardware is often available for this purpose, in order to minimize the disruption of the pipeline as much as possible:



# Pipelining - problems

The execution of instructions in a pipeline is disturbed by:

- Jump instructions
- Interrupts
- Data dependencies (the next instruction requires data from the previous)

Special hardware is often available for this purpose, in order to minimize the disruption of the pipeline as much as possible:

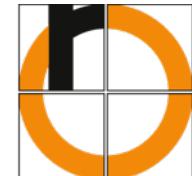


# Pipelining - problems

The execution of instructions in a pipeline is disturbed by:

- Jump instructions
- Interrupts
- Data dependencies (the next instruction requires data from the previous)

Special hardware is often available for this purpose, in order to minimize the disruption of the pipeline as much as possible:



# Pipelining - problems

The execution of instructions in a pipeline is disturbed by:

- Jump instructions
- Interrupts
- Data dependencies (the next instruction requires data from the previous)

Special hardware is often available for this purpose, in order to minimize the disruption of the pipeline as much as possible:



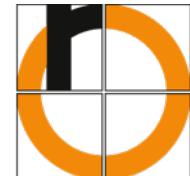
# Pipelining - problems

The execution of instructions in a pipeline is disturbed by:

- Jump instructions
- Interrupts
- Data dependencies (the next instruction requires data from the previous)

Special hardware is often available for this purpose, in order to minimize the disruption of the pipeline as much as possible:

- On conditional jumps: instructions for both alternatives are fetched and evaluated
- A kind of cache remembers which alternative from the past seems more likely (**branch prediction**)
- Try to avoid conditional jumps—instead use **conditional instructions** (e.g. CMOV, MOVCC)



# Pipelining - problems

The execution of instructions in a pipeline is disturbed by:

- Jump instructions
- Interrupts
- Data dependencies (the next instruction requires data from the previous)

Special hardware is often available for this purpose, in order to minimize the disruption of the pipeline as much as possible:

- On conditional jumps: instructions for both alternatives are fetched and evaluated
- A kind of cache remembers which alternative from the past seems more likely (branch prediction)
- Try to avoid conditional jumps—instead use conditional instructions (e.g. CMOV, MOVCC)



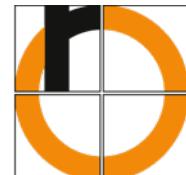
# Pipelining - problems

The execution of instructions in a pipeline is disturbed by:

- Jump instructions
- Interrupts
- Data dependencies (the next instruction requires data from the previous)

Special hardware is often available for this purpose, in order to minimize the disruption of the pipeline as much as possible:

- On conditional jumps: instructions for both alternatives are fetched and evaluated
- A kind of cache remembers which alternative from the past seems more likely (**branch prediction**)
- Try to avoid conditional jumps—instead use **conditional instructions** (e.g. CMOV, MOVCC)



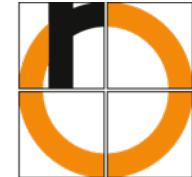
# Pipelining - problems

The execution of instructions in a pipeline is disturbed by:

- Jump instructions
- Interrupts
- Data dependencies (the next instruction requires data from the previous)

Special hardware is often available for this purpose, in order to minimize the disruption of the pipeline as much as possible:

- On conditional jumps: instructions for both alternatives are fetched and evaluated
- A kind of cache remembers which alternative from the past seems more likely (**branch prediction**)
- Try to avoid conditional jumps—instead use **conditional instructions** (e.g. CMOV, MOVCC)



# Instruction scheduling

**Idea: The compiler changes the order of the instructions**

- Change the processing order of the instructions to maximize the pipeline utilisation
- Important: The semantics of the high-level language program remains intact (as-if rule)



# Instruction scheduling

Idea: **The compiler changes the order of the instructions**

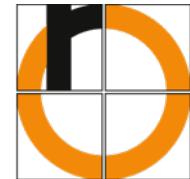
- Change the processing order of the instructions to **maximize the pipeline utilisation**
- Important: The **semantics** of the high-level language program **remains intact** (as-if rule)



# Instruction scheduling

Idea: **The compiler changes the order of the instructions**

- Change the processing order of the instructions to **maximize the pipeline utilisation**
- Important: The **semantics** of the high-level language program **remains intact** (as-if rule)



# Instruction scheduling - example

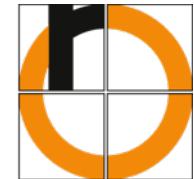
Pipeline (with 4 stages):

Unscheduled:

Nr	Opcode	Operand	Comment	Pseudo C code

[source (with changes): Alsup: Motorola's 88000 Family Architecture, IEEE Micro, June 1990]

Fetch	Load	Execute	Save
$t_1$			
$t_2$			
$t_3$			
$t_4$			
$t_5$			
$t_6$			
$t_7$			
$t_8$			
$t_9$			
$t_{10}$			
$t_{11}$			
$t_{12}$			
$t_{13}$			
$t_{14}$			
$t_{15}$			
$t_{16}$			
$t_{17}$			
$t_{18}$			
$t_{19}$			
$t_{20}$			
$t_{21}$			
$t_{22}$			



# Instruction scheduling - example

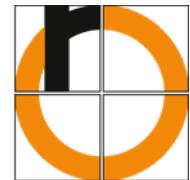
Pipeline (with 4 stages):

	Fetch	Load	Execute	Save
$t_1$				
$t_2$				
$t_3$				
$t_4$				
$t_5$				
$t_6$				
$t_7$				
$t_8$				
$t_9$				
$t_{10}$				
$t_{11}$				
$t_{12}$				
$t_{13}$				
$t_{14}$				
$t_{15}$				
$t_{16}$				
$t_{17}$				
$t_{18}$				
$t_{19}$				
$t_{20}$				
$t_{21}$				
$t_{22}$				

Unscheduled:

Nr	Opcode	Operand	Comment	Pseudo C code
1	LOAD	R7, (R30)	R7 = (R30)	
2	ADD	R7, R7, 1	R7 = R7+1	x++;
3	STORE	R7, (R30)	(R30) = R7	

[source (with changes): Alsup: Motorola's 88000 Family Architecture, IEEE Micro, June 1990]



# Instruction scheduling - example

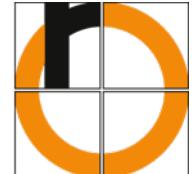
Pipeline (with 4 stages):

	Fetch	Load	Execute	Save
$t_1$				
$t_2$				
$t_3$				
$t_4$				
$t_5$				
$t_6$				
$t_7$				
$t_8$				
$t_9$				
$t_{10}$				
$t_{11}$				
$t_{12}$				
$t_{13}$				
$t_{14}$				
$t_{15}$				
$t_{16}$				
$t_{17}$				
$t_{18}$				
$t_{19}$				
$t_{20}$				
$t_{21}$				
$t_{22}$				

Unscheduled:

Nr	Opcode	Operand	Comment	Pseudo C code
1	LOAD	R7, (R30)	R7 = (R30)	
2	ADD	R7, R7, 1	R7 = R7+1	x++;
3	STORE	R7, (R30)	(R30) = R7	
4	LOAD	R8, (R31)	R8 = (R31)	
5	ADD	R8, R8, 4	R8 = R8+4	y += 4;
6	STORE	R8, (R31)	(R31) = R8	

[source (with changes): Alsup: Motorola's 88000 Family Architecture, IEEE Micro, June 1990]



# Instruction scheduling - example

Pipeline (with 4 stages):

Unscheduled:

Nr	Opcode	Operand	Comment	Pseudo C code
1	LOAD	R7, (R30)	R7 = (R30)	
2	ADD	R7, R7, 1	R7 = R7+1	x++;
3	STORE	R7, (R30)	(R30) = R7	
4	LOAD	R8, (R31)	R8 = (R31)	
5	ADD	R8, R8, 4	R8 = R8+4	y += 4;
6	STORE	R8, (R31)	(R31) = R8	
7	CMP	R10, R6, R2	R10 = R6 < R2	if(R6 < R2)
8	BLT	R10, M	GOTO M if R10 < 0	GOTO M;

[source (with changes): Alsup: Motorola's 88000 Family Architecture, IEEE Micro, June 1990]

Fetch	Load	Execute	Save	
1				$t_1$
2	1			$t_2$
2		1		$t_3$
2			1	$t_4$
3	2			$t_5$
3		2		$t_6$
3			2	$t_7$
4	3			$t_8$
4		3		$t_9$
5	4	3		$t_{10}$
5		4	3	$t_{11}$
5			4	$t_{12}$
6	5			$t_{13}$
6		5		$t_{14}$
6			5	$t_{15}$
7	6			$t_{16}$
7		6		$t_{17}$
8	7	6		$t_{18}$
8		7	6	$t_{19}$
8			7	$t_{20}$
		8		$t_{21}$
			8	$t_{22}$
M				

# Instruction scheduling - example

Pipeline (with 4 stages):

	Fetch	Load	Execute	Save	
$t_1$	1				
$t_2$	2	1			
$t_3$	2		1		
$t_4$	2			1	
$t_5$	3	2			
$t_6$	3		2		
$t_7$	3			2	
$t_8$	4	3			
$t_9$	5	4	3		
$t_{10}$	5		4	3	
$t_{11}$	5			4	
$t_{12}$	6	5			
$t_{13}$	6		5		
$t_{14}$	6			5	
$t_{15}$	7	6			
$t_{16}$	8	7	6		
$t_{17}$	8		7	6	
$t_{18}$	8			7	
$t_{19}$	8				
$t_{20}$			8		
$t_{21}$				8	
$t_{22}$	M				

Unscheduled:

Nr	Opcode	Operand	Comment	Pseudo C code
1	LOAD	R7, (R30)	$R7 = (R30)$	
2	ADD	R7, R7, 1	$R7 = R7 + 1$	$x++;$
3	STORE	R7, (R30)	$(R30) = R7$	
4	LOAD	R8, (R31)	$R8 = (R31)$	
5	ADD	R8, R8, 4	$R8 = R8 + 4$	$y += 4;$
6	STORE	R8, (R31)	$(R31) = R8$	
7	CMP	R10, R6, R2	$R10 = R6 < R2$	$\text{if}(R6 < R2)$
8	BLT	R10, M	GOTO M if $R10 < 0$	GOTO M;

[source (with changes): Alsup: Motorola's 88000 Family Architecture, IEEE Micro, June 1990]



# Instruction scheduling - example

Scheduled:

Nr	Opcode	Operand	Old order
1	LOAD	R7, (R30)	1
2	CMP	R10, R6, R2	7
3	LOAD	R8, (R31)	4
4	ADD	R7, R7, 1	2
5	ADD	R8, R8, 4	5
6	STORE	R7, (R30)	3
7	BLT	R10, M	8
8	STORE	R8, (R31)	6

[source (with changes): Alsup: Motorola's 88000 Family Architecture, IEEE Micro, June 1990]

Pipeline (with 4 stages):

Fetch	Load	Execute	Save	
1				$t_1$
2	1			$t_2$
3	2	1		$t_3$
4	3	2	1	$t_4$
5	4	3	2	$t_5$
5		4	3	$t_6$
6	5		4	$t_7$
7	6	5		$t_8$
8	7	6	5	$t_9$
	8	7	6	$t_{10}$
		8	7	$t_{11}$
			8	$t_{12}$
M				$t_{13}$



# Instruction scheduling - example

Scheduled:

Nr	Opcode	Operand	Old order
1	LOAD	R7, (R30)	1
2	CMP	R10, R6, R2	7
3	LOAD	R8, (R31)	4
4	ADD	R7, R7, 1	2
5	ADD	R8, R8, 4	5
6	STORE	R7, (R30)	3
7	BLT	R10, M	8
8	STORE	R8, (R31)	6

[source (with changes): Alsup: Motorola's 88000 Family Architecture, IEEE Micro, June 1990]

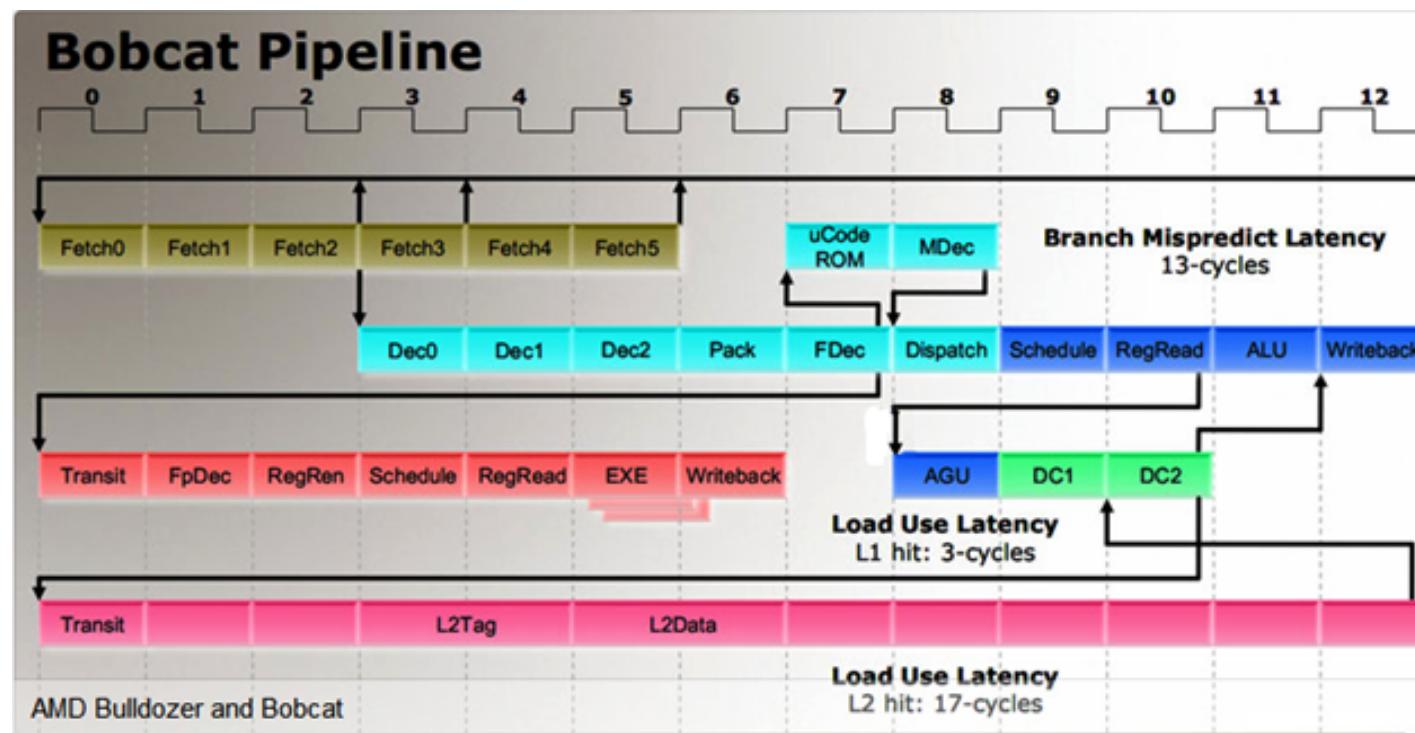
Pipeline (with 4 stages):

	Fetch	Load	Execute	Save
1				
2	1			
3	2	1		
4	3	2	1	
5	4	3	2	1
5		4	3	2
6	5		4	3
7	6	5		4
8	7	6	5	
	8	7	6	5
		8	7	6
			8	7
M				8

Total clock cycles: 12



# Example: AMD Bobcat pipelining



[Source: AMD Bobcat & Bulldozer Hot Chips Presentations 2011, available on youtube, last access 6.3.2015;  
Presentation: AMD Bobcat]



# Instruction scheduling - C example

**Core 1:** (producer)

```
1 byte data[1000];
2 byte shared_data[1000];
3 bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 flag = true; //data ready
```

**Core 2:** (consumer)

```
1 byte data_copy[1000];
2 byte* shared_data; //attach
3 bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 // data_copy = shared_data;
11 memcpy(data_copy, shared_data, 1000);
```

Any problems with that code?



# Instruction scheduling - C example

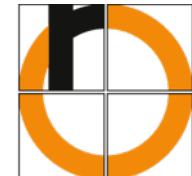
**Core 1:** (producer)

```
1 byte data[1000];
2 byte shared_data[1000];
3 bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 flag = true; //data ready
```

**Core 2:** (consumer)

```
1 byte data_copy[1000];
2 byte* shared_data; //attach
3 bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 // data_copy = shared_data;
11 memcpy(data_copy, shared_data, 1000);
```

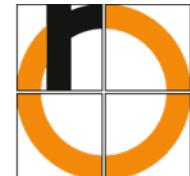
Any problems with that code?



# Instruction scheduling - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the instruction scheduling can cause **unpredictable behaviour** (in concurrent programs, device drivers) when **accessing the same data**.

Prevent compiler from reordering (instruction scheduling)



# Instruction scheduling - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the instruction scheduling can cause **unpredictable behaviour** (in concurrent programs, device drivers) when **accessing the same data**.

## Prevent **compiler from reordering** (instruction scheduling)

- Use of the `volatile` keyword in C
- Forces the compiler to
  
- Notice: The `volatile` keyword does not prevent the hardware to reorder instructions

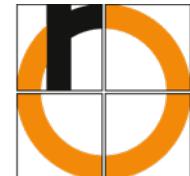


# Instruction scheduling - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the instruction scheduling can cause **unpredictable behaviour** (in concurrent programs, device drivers) when **accessing the same data**.

## Prevent **compiler from reordering** (instruction scheduling)

- Use of the **volatile** keyword in C
- Forces the compiler to
  - Notice: The **volatile** keyword does not prevent the hardware to reorder instructions

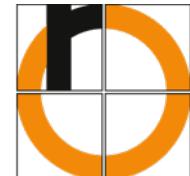


# Instruction scheduling - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the instruction scheduling can cause **unpredictable behaviour** (in concurrent programs, device drivers) when **accessing the same data**.

## Prevent **compiler from reordering** (instruction scheduling)

- Use of the **volatile** keyword in C
- Forces the compiler to
  - not omit reads from and writes to volatile memory locations,
  - nor may it reorder read/writes relative to other such actions for the same volatile location
- Notice: The **volatile** keyword does not prevent the hardware to reorder instructions

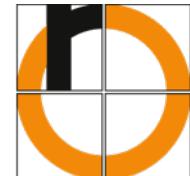


# Instruction scheduling - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the instruction scheduling can cause **unpredictable behaviour** (in concurrent programs, device drivers) when **accessing the same data**.

## Prevent compiler from reordering (instruction scheduling)

- Use of the **volatile** keyword in C
- Forces the compiler to
  - not omit reads from and writes to volatile memory locations,
  - nor may it reorder read/writes relative to other such actions for the same volatile location
- Notice: The **volatile** keyword does not prevent the hardware to reorder instructions



# Instruction scheduling - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the instruction scheduling can cause **unpredictable behaviour** (in concurrent programs, device drivers) when **accessing the same data**.

## Prevent **compiler from reordering** (instruction scheduling)

- Use of the **volatile** keyword in C
- Forces the compiler to
  - not omit reads from and writes to volatile memory locations,
  - nor may it reorder read/writes relative to other such actions for the same volatile location
- Notice: The **volatile** keyword does not prevent the hardware to reorder instructions



# Instruction scheduling - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the instruction scheduling can cause **unpredictable behaviour** (in concurrent programs, device drivers) when **accessing the same data**.

## Prevent **compiler from reordering** (instruction scheduling)

- Use of the **volatile** keyword in C
- Forces the compiler to
  - not omit reads from and writes to volatile memory locations,
  - nor may it reorder read/writes relative to other such actions for the same volatile location
- Notice: The **volatile** keyword does not prevent the hardware to reorder instructions



# Instruction scheduling - C example

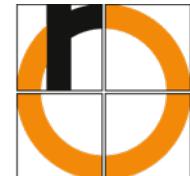
## Core 1: (producer)

```
1 byte data[1000];
2 volatile byte shared_data[1000];
3 volatile bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 flag = true; //data ready
```

## Core 2: (consumer)

```
1 byte data_copy[1000];
2 volatile byte* shared_data; //attach
3 volatile bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 // data_copy = shared_data;
11 memcpy(data_copy, shared_data, 1000);
```

- The use of **volatile** prevents the compiler from reordering or omitting instructions
- There is still a problem, that can occur through **out-of-order memory access**...



# Instruction scheduling - C example

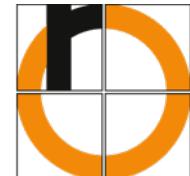
## Core 1: (producer)

```
1 byte data[1000];
2 volatile byte shared_data[1000];
3 volatile bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 flag = true; //data ready
```

## Core 2: (consumer)

```
1 byte data_copy[1000];
2 volatile byte* shared_data; //attach
3 volatile bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 // data_copy = shared_data;
11 memcpy(data_copy, shared_data, 1000);
```

- The use of **volatile** prevents the compiler from reordering or omitting instructions
- There is still a problem, that can occur through out-of-order memory access...



# Instruction scheduling - C example

## Core 1: (producer)

```
1 byte data[1000];
2 volatile byte shared_data[1000];
3 volatile bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 flag = true; //data ready
```

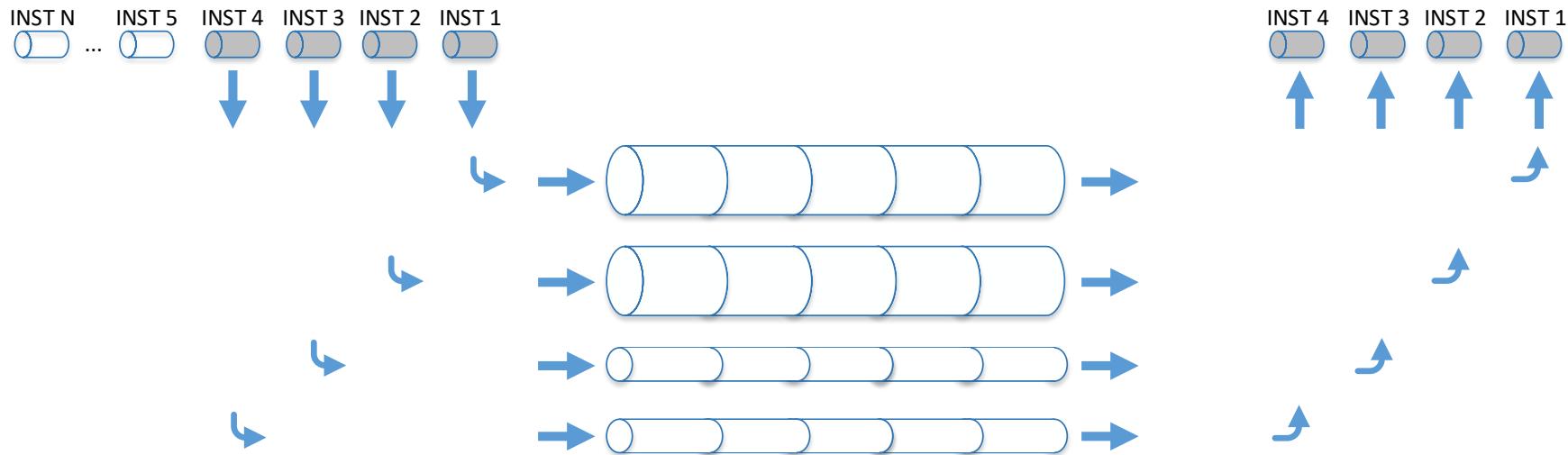
## Core 2: (consumer)

```
1 byte data_copy[1000];
2 volatile byte* shared_data; //attach
3 volatile bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 // data_copy = shared_data;
11 memcpy(data_copy, shared_data, 1000);
```

- The use of **volatile** prevents the compiler from reordering or omitting instructions
- There is still a problem, that can occur through **out-of-order memory** access...



# Superscalar architecture

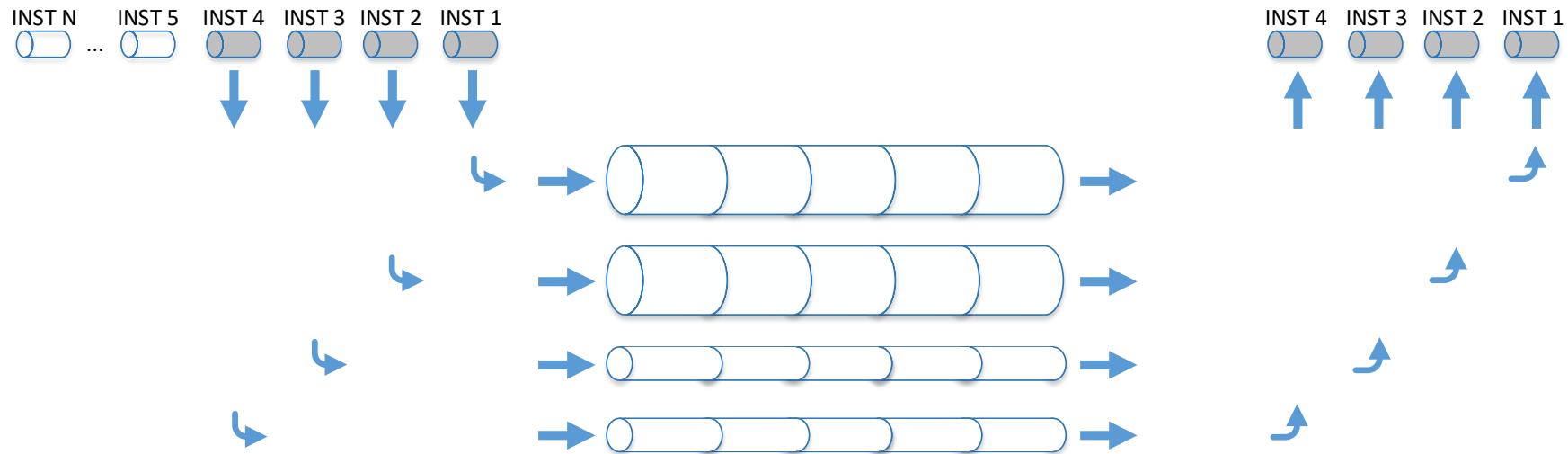


## Superscalar architecture:

- The CPU provides several pipelines
- The pipelines may be specialised for integer or floating point operations
- The CPU loads multiple instructions at each clock cycle
- Everything is done automatically inside the CPU: programming model stays unchanged



# Superscalar architecture

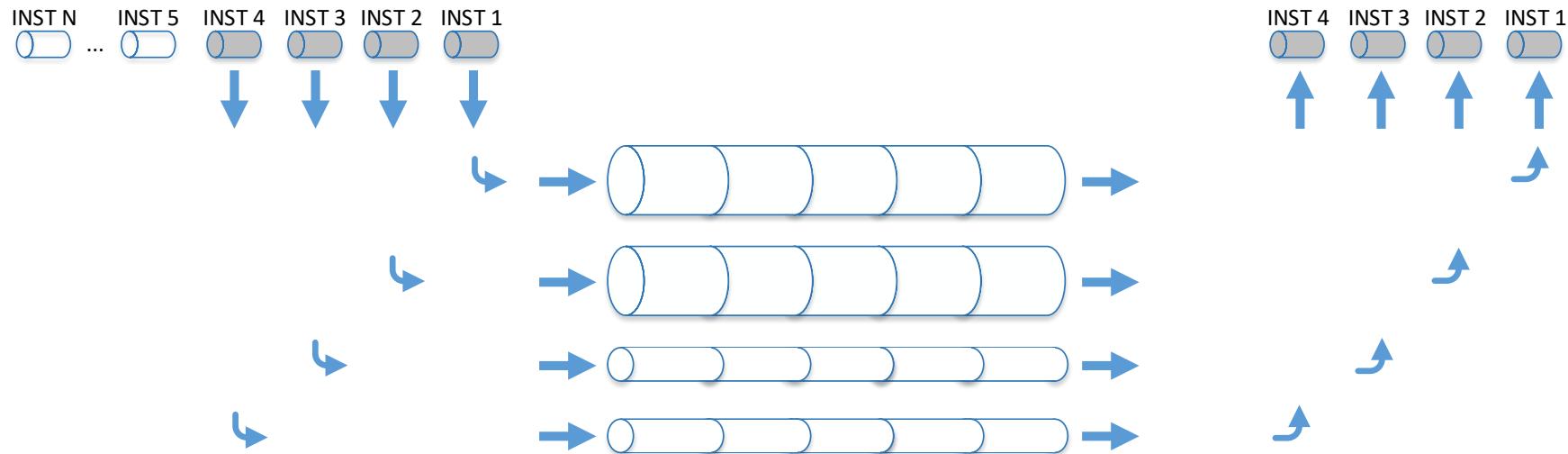


## Superscalar architecture:

- The CPU provides **several pipelines**
- The pipelines may be specialised for integer or floating point operations
- The CPU loads **multiple instructions** at each clock cycle
- Everything is done automatically inside the CPU: **programming model stays unchanged**



# Superscalar architecture

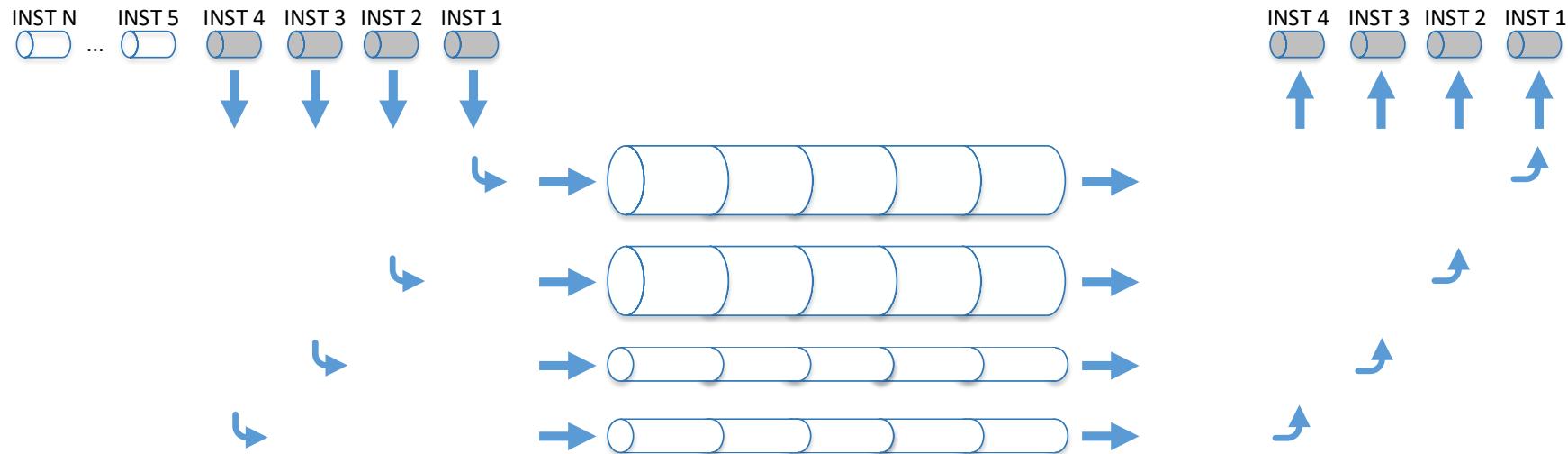


## Superscalar architecture:

- The CPU provides **several pipelines**
- The pipelines may be specialised for integer or floating point operations
- The CPU loads **multiple instructions** at each clock cycle
- Everything is done automatically inside the CPU: **programming model stays unchanged**

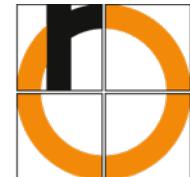


# Superscalar architecture

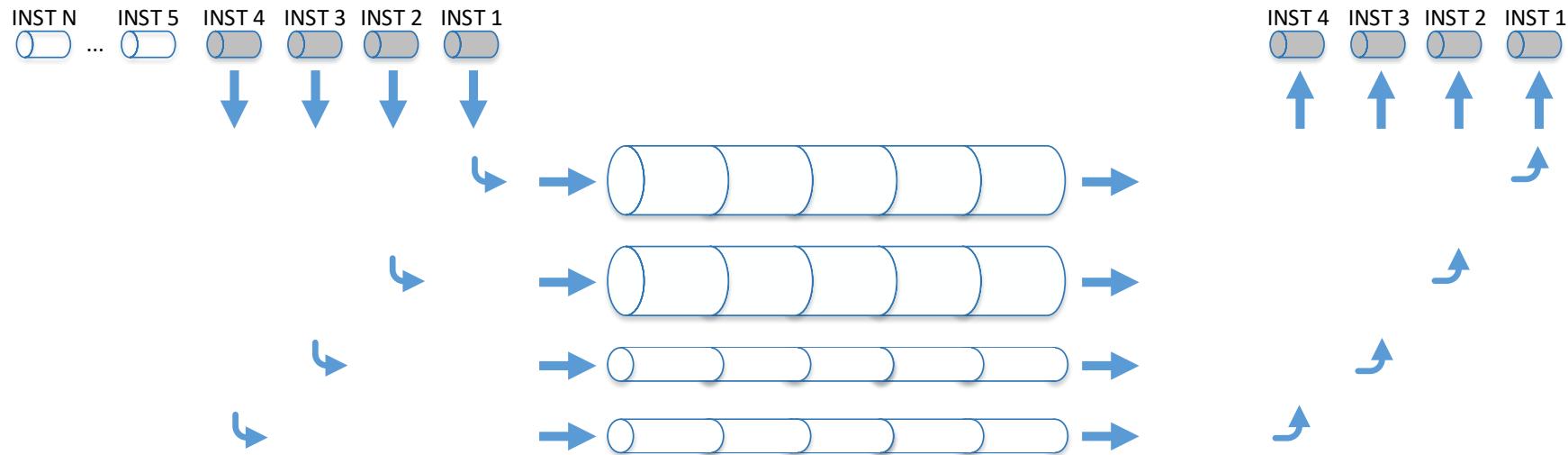


## Superscalar architecture:

- The CPU provides **several pipelines**
- The pipelines may be specialised for integer or floating point operations
- The CPU loads **multiple instructions** at each clock cycle
- Everything is done automatically inside the CPU: programming model stays unchanged



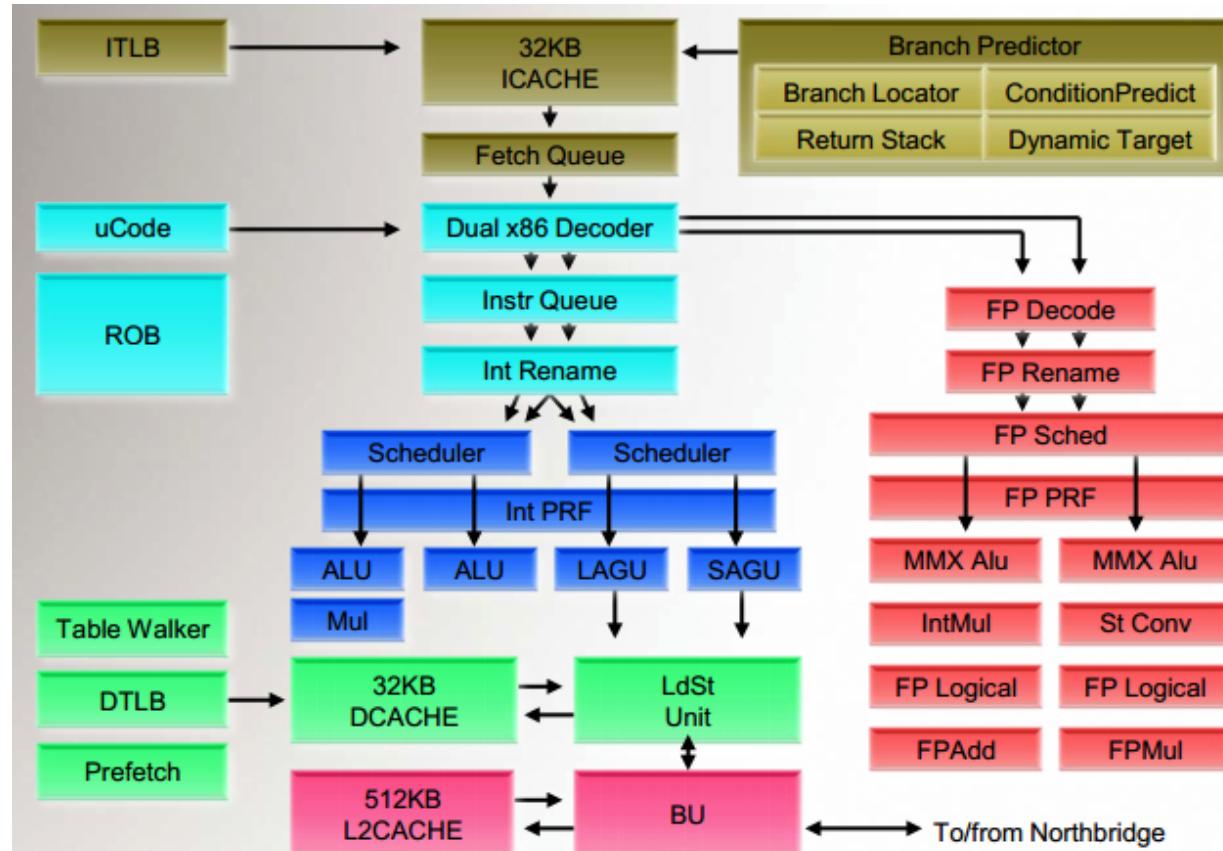
# Superscalar architecture



## Superscalar architecture:

- The CPU provides **several pipelines**
- The pipelines may be specialised for integer or floating point operations
- The CPU loads **multiple instructions** at each clock cycle
- Everything is done automatically inside the CPU: **programming model stays unchanged**

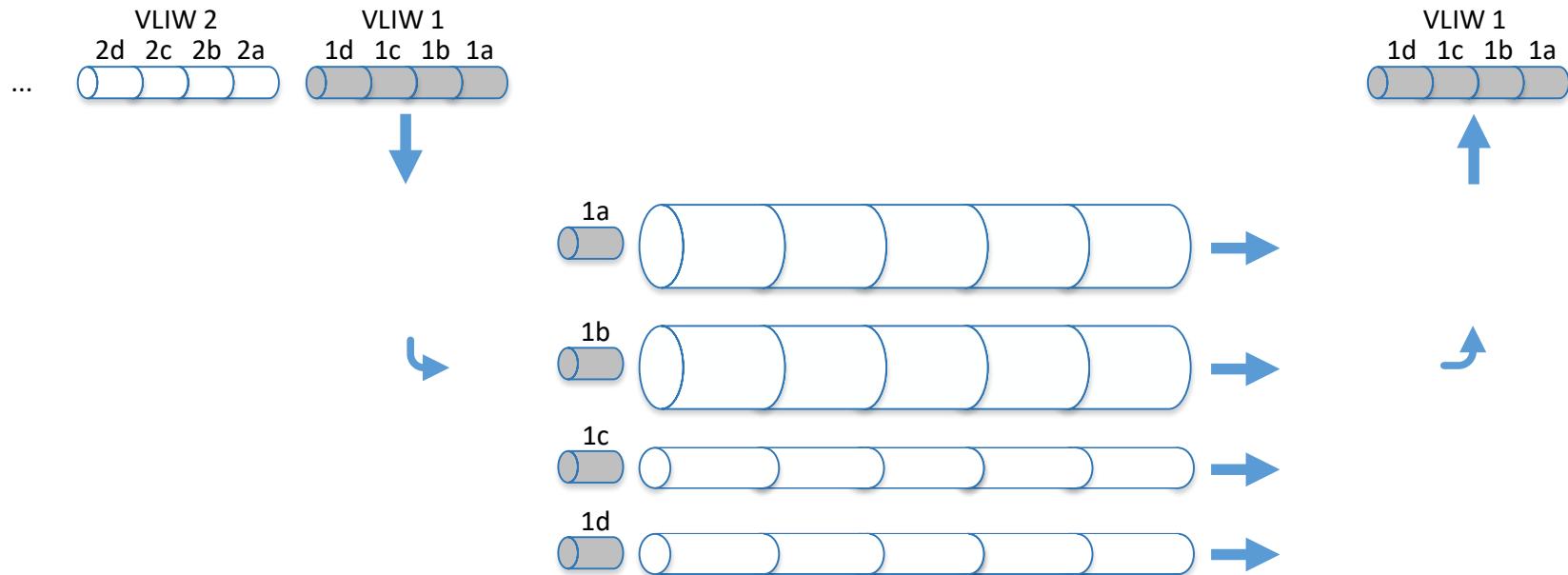
# Example: AMD Bobcat superscalarity



[Source: AMD Bobcat & Bulldozer Hot Chips Presentations 2011, available on youtube, last access 6.3.2015]

Notice: AGU = address generation units, LAGU = AGU for loading operations, SAGU = AGU for storage operations

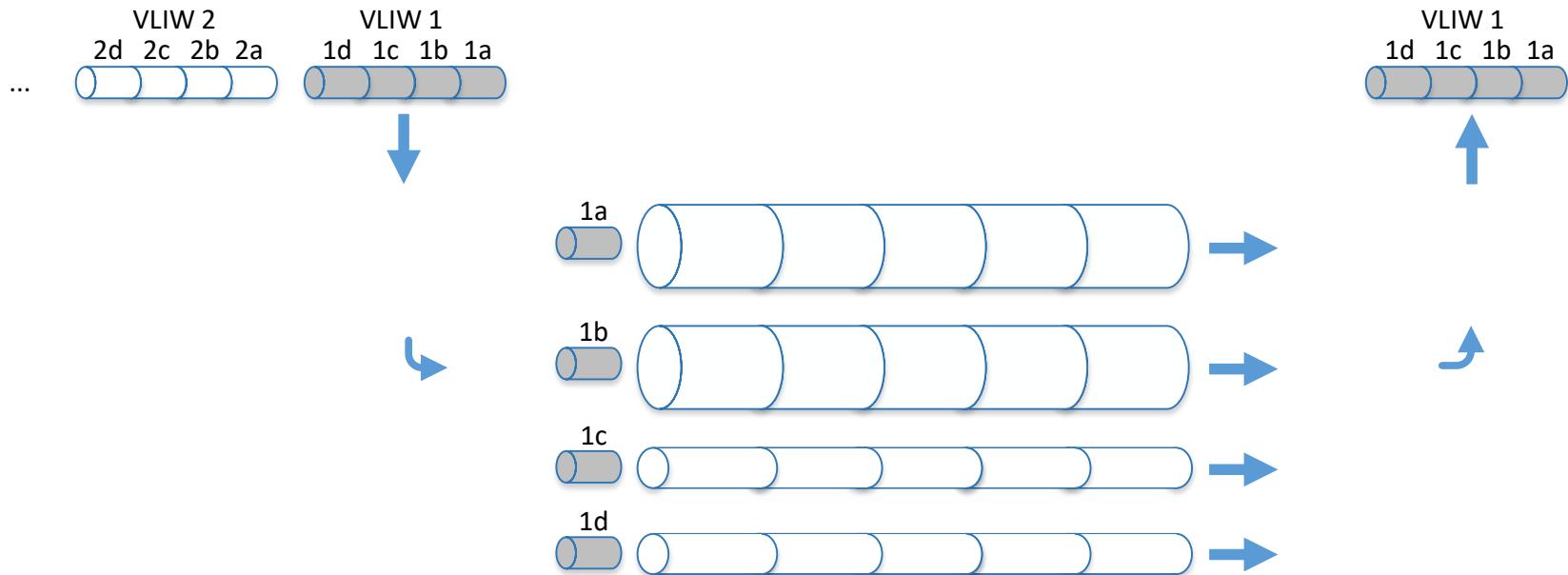
# VLIW - Very long instruction word



## VLIW architecture:

- Parallel operations are packed into a very long instruction
- Packing is done by the compiler at compile time: **programming model changed!**
- The partial instructions of a VLIW must correspond to the functional units of the hardware
- It's used by the Intel Itanium architecture (EPIC - Explicitly parallel instruction computer)

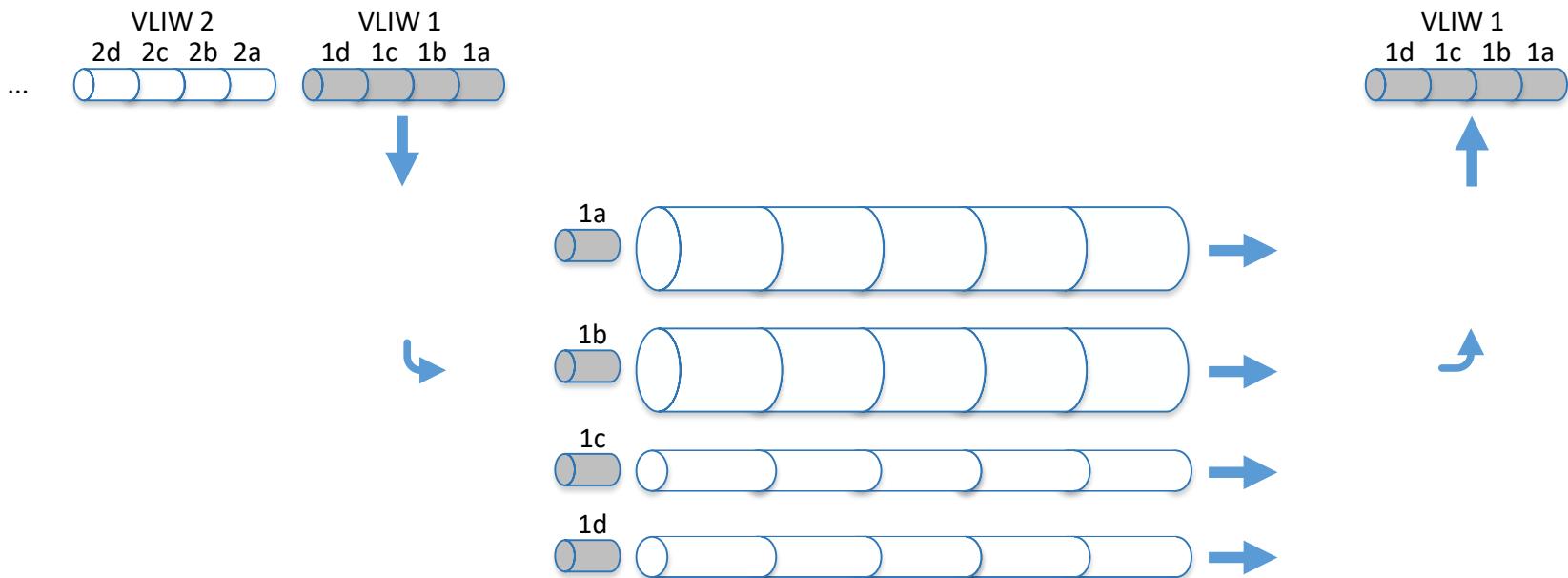
# VLIW - Very long instruction word



## VLIW architecture:

- Parallel operations are packed into a very long instruction
- Packing is done by the compiler at compile time: *programming model changed!*
- The partial instructions of a VLIW must correspond to the functional units of the hardware
- Its used by the Intel Itanium architecture (EPIC - Explicitly parallel instruction computer)

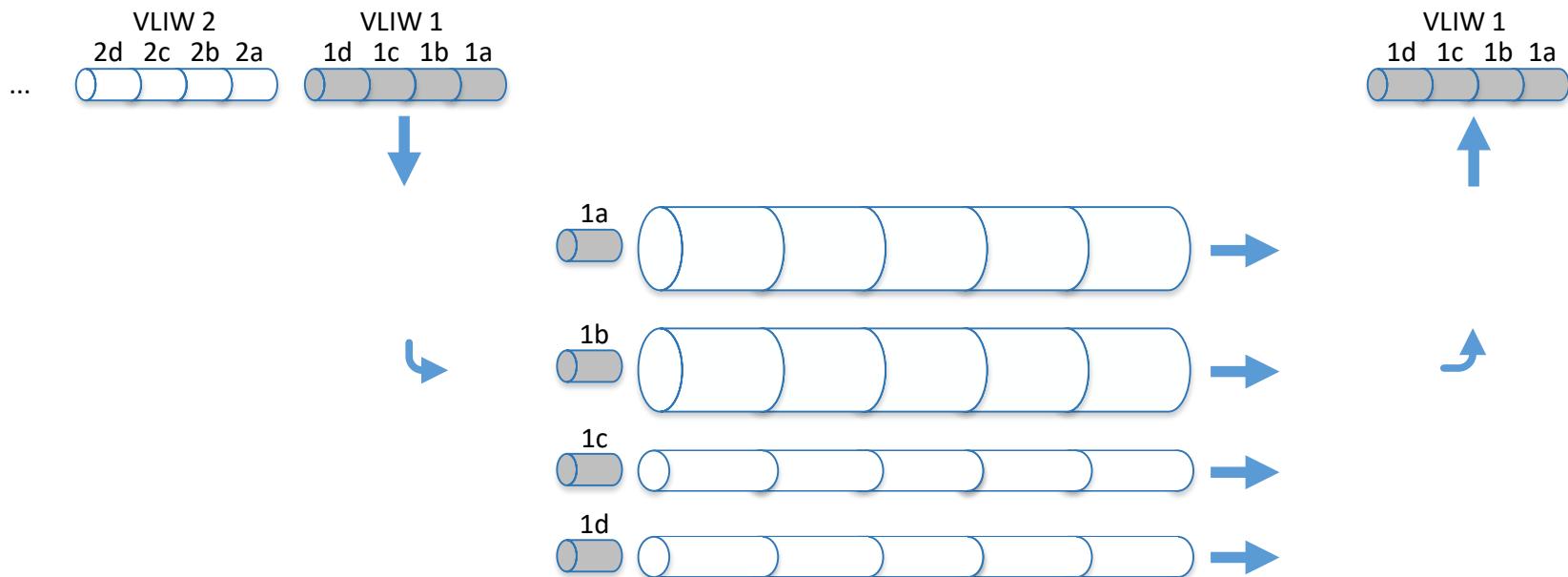
# VLIW - Very long instruction word



## VLIW architecture:

- Parallel operations are packed into a very long instruction
- Packing is done by the compiler at compile time: **programming model changed!**
- The partial instructions of a VLIW must correspond to the functional units of the hardware
- Its used by the Intel Itanium architecture (EPIC - Explicitly parallel instruction computer)

# VLIW - Very long instruction word

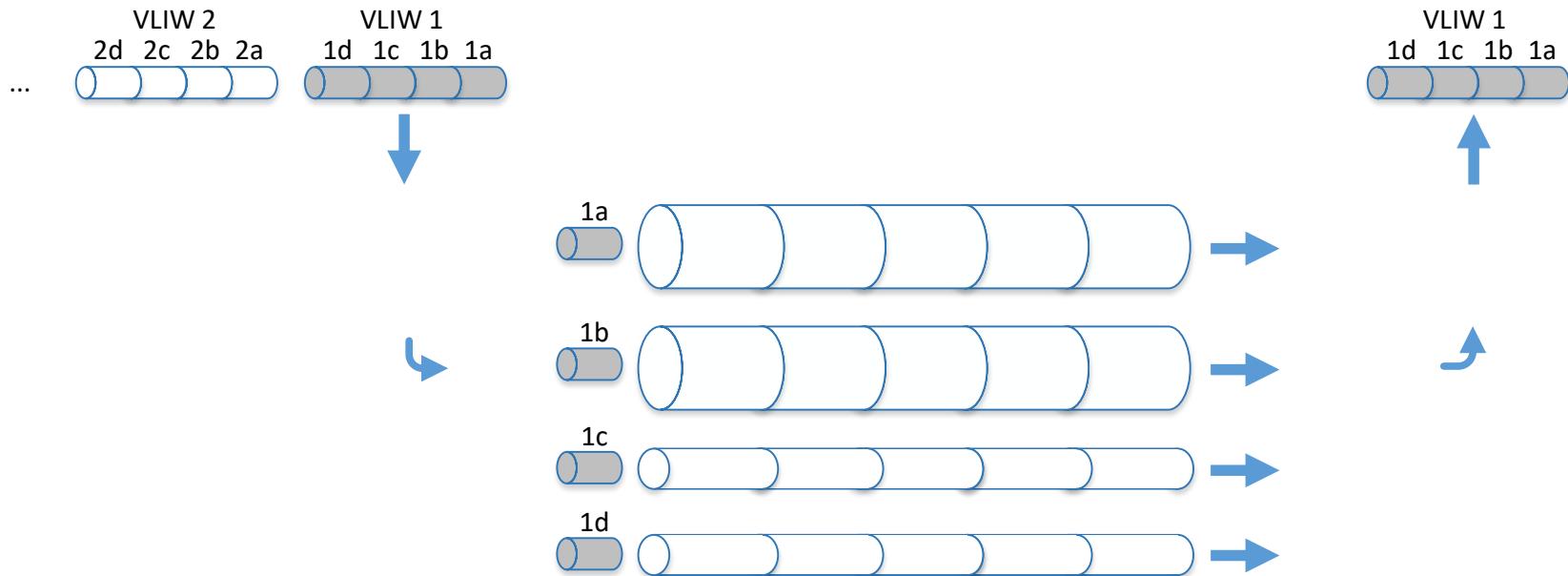


## VLIW architecture:

- Parallel operations are packed into a very long instruction
- Packing is done by the **compiler at compile time**: **programming model changed!**
- The partial instructions of a VLIW must correspond to the functional units of the hardware

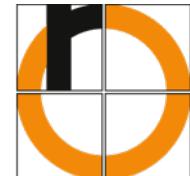
■ Its used by the Intel Itanium architecture (EPIC - Explicitly parallel instruction computer)

# VLIW - Very long instruction word



## VLIW architecture:

- Parallel operations are packed into a very long instruction
- Packing is done by the **compiler at compile time**: **programming model changed!**
- The partial instructions of a VLIW must correspond to the functional units of the hardware
- Its used by the **Intel Itanium** architecture (EPIC - Explicitly parallel instruction computer)



# Out-of-order memory access

**Idea:** The **hardware (CPU)** changes the order of the instructions: with the goal to better utilise the hardware

## Details:

- Memory access (instruction) is **not** done in the specified order
- Some accesses are “**too early**” (e.g. speculative load)
- Some accesses are “**too late**” (e.g. posted writes)
- Attention: The **semantics must be guaranteed!**



# Out-of-order memory access

**Idea:** The **hardware (CPU)** changes the order of the instructions: with the goal to better utilise the hardware

## Details:

- Memory access (instruction) is **not** done in the specified order
  - Some accesses are “too early” (e.g. speculative load)
  - Some accesses are “too late” (e.g. posted writes)
  - Attention: The semantics must be guaranteed!



# Out-of-order memory access

**Idea:** The **hardware (CPU) changes the order of the instructions**: with the goal to better utilise the hardware

## Details:

- Memory access (instruction) is **not** done in the specified order
- Some accesses are “**too early**” (e.g. speculative load)
- Some accesses are “**too late**” (e.g. posted writes)
- Attention: The **semantics must be guaranteed!**



# Out-of-order memory access

**Idea:** The **hardware (CPU)** changes the order of the instructions: with the goal to better utilise the hardware

## Details:

- Memory access (instruction) is **not** done in the specified order
- Some accesses are “**too early**” (e.g. speculative load)
- Some accesses are “**too late**” (e.g. posted writes)
- Attention: The semantics must be guaranteed!



# Out-of-order memory access

**Idea:** The **hardware (CPU) changes the order of the instructions**: with the goal to better utilise the hardware

## Details:

- Memory access (instruction) is **not** done in the specified order
- Some accesses are “**too early**” (e.g. speculative load)
- Some accesses are “**too late**” (e.g. posted writes)
- Attention: The **semantics must be guaranteed!**



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
  - 2 Load operands (may take some cycles)
  - 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution

- 1 Fetch instruction
- 2 Queue instruction to an instruction queue (instruction buffer)
- 3 Load operands: Instructions wait in the queue until its input operands are available (a instruction can leave the queue before an older instructions)
- 4 Execute instruction by the appropriate functional unit
- 5 Queue results
- 6 Save the results into the register—only after all older instructions have their results saved to the register



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution

- 1 Fetch instruction
- 2 Queue instruction to an instruction queue (instruction buffer)
- 3 Load operands: Instructions wait in the queue until its input operands are available (a instruction can leave the queue before an older instructions)
- 4 Execute instruction by the appropriate functional unit
- 5 Queue results
- 6 Save the results into the register—only after all older instructions have their results saved to the register



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution

- 1 Fetch instruction
- 2 Queue instruction to an instruction queue (instruction buffer)
- 3 Load operands: Instructions wait in the queue until its input operands are available (a instruction can leave the queue before an older instructions)
- 4 Execute instruction by the appropriate functional unit
- 5 Queue results
- 6 Save the results into the register—only after all older instructions have their results saved to the register



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution

- 1 Fetch instruction
- 2 Queue **instruction** to an instruction queue (instruction buffer)
- 3 Load operands: **Instructions wait** in the queue until its input operands are available (a instruction can leave the queue before an older instructions)
- 4 Execute instruction by the appropriate functional unit
- 5 Queue results
- 6 Save the results into the register—only after all older instructions have their results saved to the register



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution

- 1 Fetch instruction
- 2 Queue **instruction** to an instruction queue (instruction buffer)
- 3 Load operands: **Instructions wait** in the queue until its input operands are available (a instruction can leave the queue before an older instructions)
- 4 Execute instruction by the appropriate functional unit
- 5 Queue results
- 6 Save the results into the register—only after all older instructions have their results saved to the register



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution

- 1 Fetch instruction
- 2 Queue **instruction** to an instruction queue (instruction buffer)
- 3 Load operands: **Instructions wait** in the queue until its input operands are available (a instruction can leave the queue before an older instructions)
- 4 Execute instruction by the appropriate functional unit
- 5 Queue **results**
- 6 Save the results into the register—only after all older instructions have their results saved to the register



# Out-of-order memory access

## In-order execution

- 1 Fetch instruction
- 2 Load operands (may take some cycles)
- 3 Execute instruction by the appropriate functional unit
- 4 Save the results into the register

## Out-of-order execution

- 1 Fetch instruction
- 2 Queue **instruction** to an instruction queue (instruction buffer)
- 3 Load operands: **Instructions wait** in the queue until its input operands are available (a instruction can leave the queue before an older instructions)
- 4 Execute instruction by the appropriate functional unit
- 5 Queue **results**
- 6 Save the results into the register—**only after all older instructions have their results saved** to the register



# Out-of-order memory access - C example

## Core 1: (producer)

```
1 byte data[1000];
2 volatile byte shared_data[1000];
3 volatile bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 flag = true; //data ready
```

## Core 2: (consumer)

```
1 byte data_copy[1000];
2 volatile byte* shared_data; //attach
3 volatile bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 // data_copy = shared_data;
11 memcpy(data_copy, shared_data, 1000);
```

Any problems with that code?



# Out-of-order memory access - C example

## Core 1: (producer)

```
1 byte data[1000];
2 volatile byte shared_data[1000];
3 volatile bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 flag = true; //data ready
```

## Core 2: (consumer)

```
1 byte data_copy[1000];
2 volatile byte* shared_data; //attach
3 volatile bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 // data_copy = shared_data;
11 memcpy(data_copy, shared_data, 1000);
```

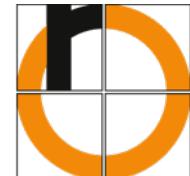
Any problems with that code?



# Out-of-order memory access - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the **out-of-order memory access** can cause **unpredictable behaviour** in (concurrent programs, device drivers) when **accessing the same data**.

Prevent hardware from reordering



# Out-of-order memory access - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the **out-of-order memory access** can cause **unpredictable behaviour** in (concurrent programs, device drivers) when **accessing the same data**.

## Prevent hardware from reordering

- Use of a **memory barrier**: a special hardware instruction (`memory_barrier`, `FENCE`)
- Forces the completion of all outstanding (instructions) memory accesses



# Out-of-order memory access - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the **out-of-order memory access** can cause **unpredictable behaviour** in (concurrent programs, device drivers) when **accessing the same data**.

## Prevent hardware from reordering

- Use of a **memory barrier**: a special hardware instruction (`memory_barrier`, `FENCE`)
  - Forces the completion of all outstanding (instructions) memory accesses



# Out-of-order memory access - problem

**Problem:** Through **asynchronous execution** units (threads, processes, interrupts, ...), the **out-of-order memory access** can cause **unpredictable behaviour** in (concurrent programs, device drivers) when **accessing the same data**.

## Prevent hardware from reordering

- Use of a **memory barrier**: a special hardware instruction (`memory_barrier`, `FENCE`)
- Forces the completion of all outstanding (instructions) memory accesses



# Out-of-order memory access - C example

## Core 1: (producer)

```
1 byte data[1000];
2 volatile byte shared_data[1000];
3 volatile bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 __asm { memory_barrier };
8 flag = true; //data ready
```

## Core 2: (consumer)

```
1 byte data_copy[1000];
2 volatile byte* shared_data; //attach
3 volatile bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 __asm { memory_barrier };
11 // data_copy = shared_data;
12 memcpy(data_copy, shared_data, 1000);
```

- The use of `__asm { memory_barrier }`; and `volatile` should fix the problem
- Best solution: use of a `semaphore` or a monitor, then all this is already solved for you (by OS software)



# Out-of-order memory access - C example

## Core 1: (producer)

```
1 byte data[1000];
2 volatile byte shared_data[1000];
3 volatile bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 __asm { memory_barrier };
8 flag = true; //data ready
```

## Core 2: (consumer)

```
1 byte data_copy[1000];
2 volatile byte* shared_data; //attach
3 volatile bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 __asm { memory_barrier };
11 // data_copy = shared_data;
12 memcpy(data_copy, shared_data, 1000);
```

- The use of `__asm { memory_barrier };` and `volatile` should fix the problem
- Best solution: use of a `semaphore` or a monitor, then all this is already solved for you (by OS software)



# Out-of-order memory access - C example

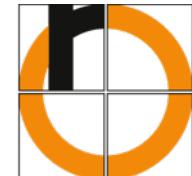
## Core 1: (producer)

```
1 byte data[1000];
2 volatile byte shared_data[1000];
3 volatile bool flag = false;
4
5 // shared_data = data;
6 memcpy(shared_data, data, 1000);
7 __asm { memory_barrier };
8 flag = true; //data ready
```

## Core 2: (consumer)

```
1 byte data_copy[1000];
2 volatile byte* shared_data; //attach
3 volatile bool* flag; //attach
4
5
6
7
8
9 while (flag != true) {} //busy wait
10 __asm { memory_barrier };
11 // data_copy = shared_data;
12 memcpy(data_copy, shared_data, 1000);
```

- The use of `__asm { memory_barrier };` and `volatile` should fix the problem
- **Best solution:** use of a **semaphore** or a monitor, then all this is already solved for you (by OS software)



# Overview

Technique

How

Unit

Time

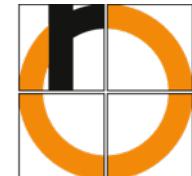


# Overview

**Technique**  
Pipelining

**How**  
parallelisation of parts of an instruction

**Unit** hardware  
**Time** runtime



# Overview

## Technique

Pipelining

Instruction scheduling

## How

parallelisation of parts of an instruction

reordering of instructions

## Unit

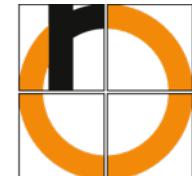
hardware

compiler

## Time

runtime

compile time



# Overview

## Technique

Pipelining

Instruction scheduling

Superscalarity

## How

parallelisation of parts of an instruction

reordering of instructions

parallelisation on instruction level

## Unit

hardware

compiler

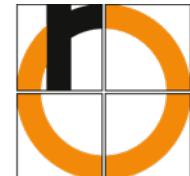
hardware

## Time

runtime

compile time

runtime



# Overview

## Technique

Pipelining

Instruction scheduling

Superscalarity

VLIW

## How

parallelisation of parts of an instruction

reordering of instructions

parallelisation on instruction level

parallelisation on instruction level

## Unit

hardware

compiler

hardware

compiler

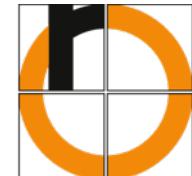
## Time

runtime

compile time

runtime

compile time



# Overview

## Technique

Pipelining

Instruction scheduling

Superscalarity

VLIW

Out-of-order memory access

## How

parallelisation of parts of an instruction

reordering of instructions

parallelisation on instruction level

parallelisation on instruction level

reordering of instructions

## Unit

hardware

compiler

hardware

compiler

hardware

## Time

runtime

compile time

runtime

compile time

runtime



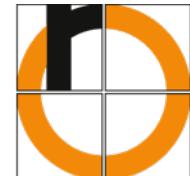
# Summary and outlook

## Summary

- Pipelining
- Instruction scheduling
- Superscalar architecture
- VLIW
- Out-of-order memory access

## Outlook

- Memory



# Summary and outlook

## Summary

- Pipelining
- Instruction scheduling
- Superscalar architecture
- VLIW
- Out-of-order memory access

## Outlook

- Memory