



# COMPUTER ARCHITECTURE

## 6 Central Processing Unit - CPU



## 6 Central Processing Unit – objectives and outcomes:

- A basic understanding of:
  - architecture (basic electronic circuits) and the operation of the CPU
  - synchronization of circuits with clock signal
  - Micro-programmed (SW) or Hard-wired (HW) implementation of the CPU
  
- Understanding of parallelism :
  - origins of existence
  - parallelisation on the instruction level
    - pipeline
  
- Understanding the execution of instructions in CPU



## 6 Central processing unit

- Basic structure and operation of the CPU
- ARM CPU – features summary
- Structure of CPU – ARM case
- Execution of instructions
- Parallel execution of instructions
- Pipelined CPU
- An example of a 5-stage pipelined CPU
- Multiple issue processors



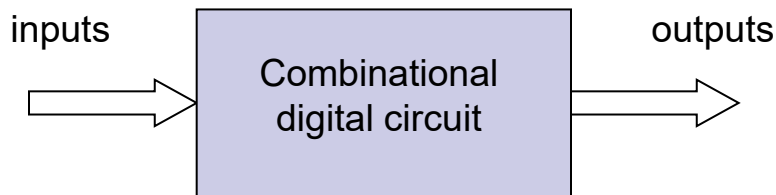
## 6.1 Basic structure and operation of the CPU

- CPU (Central Processing Unit or the CPU) is a unit that executes instructions, so its performance largely determines the performance of the whole computer.
- In addition to the CPU, most computers have also other processors, mainly in the input/output part of the computer.
- Basic principles of operation for all types of processors are identical.

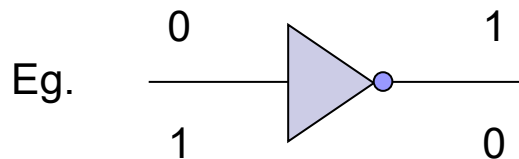


## Central processing unit

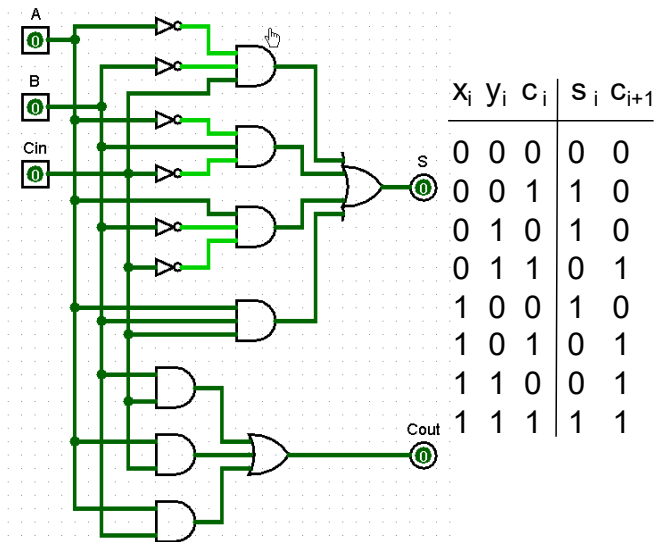
- CPU is a digital system (built from digital electronic circuits) specific types.
- Two groups of digital circuits:
  - Combinational digital circuits
    - Status output depends only on current state of the inputs



example: negator



Primer: 1-bitni seštevalnik

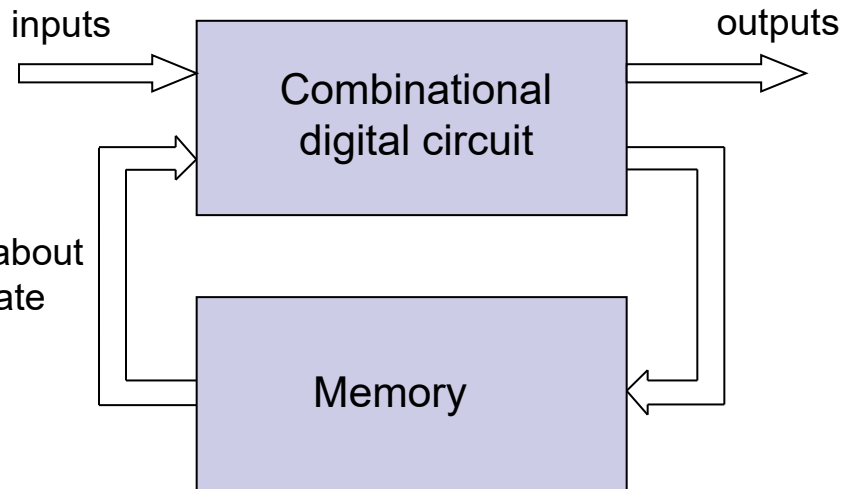




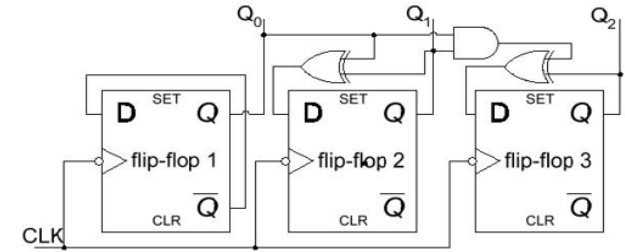
## Central processing unit

### □ Memory (sequential) digital circuits

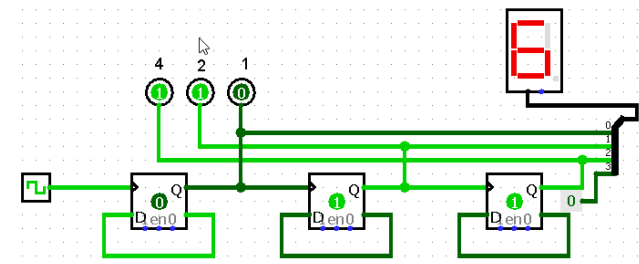
- The state of the outputs depends on the current state of inputs and the previous states of the inputs
- Memories remember the states
- Previous states are usually characterized as **internal states**, that reflect the previous states of inputs



Example: 3-bit counter



Example: 3-bit counter - Logisim

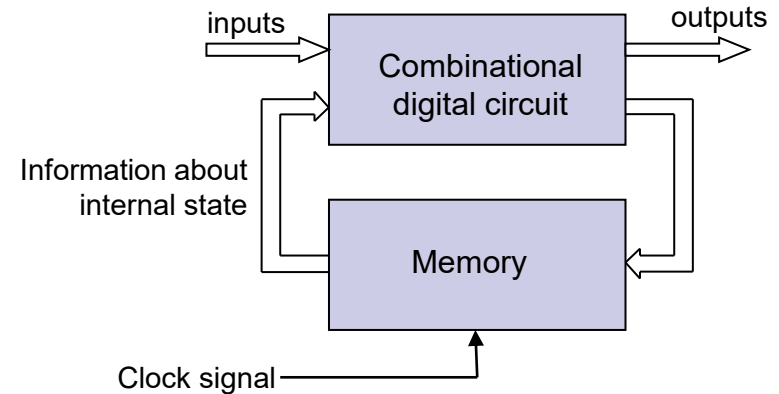




## Central processing unit

### ■ Memory (Sequential) circuit:

- Flip-flop - one-bit memory cell
- Register
- Counter
- Memory



### ■ Memory (sequential) digital circuits can be:

- **Asynchronous** - the state of the circuit is changed "Immediately" after the variation in input signals.
- **Synchronous** - the state of the circuit as a function of the input signals can only be changed at the edge of the clock signal.

### ■ CPU is built from

- Combinational and
- Memory (sequential) synchronous digital circuits.

### ■ The current state of the memory circuits presents **the state of the CPU.**



## Central processing unit

---

- The operation of the CPU at any time depends on the current state of the CPU inputs and the current internal state of the CPU.
- The number of possible internal states of the CPU depends on the size (capacity) of CPU.
- The number of bits, which represent the internal state of the CPU ranges from some 10 up to 10,000 or even more.
- Digital circuits that form a CPU today are usually on a single chip.





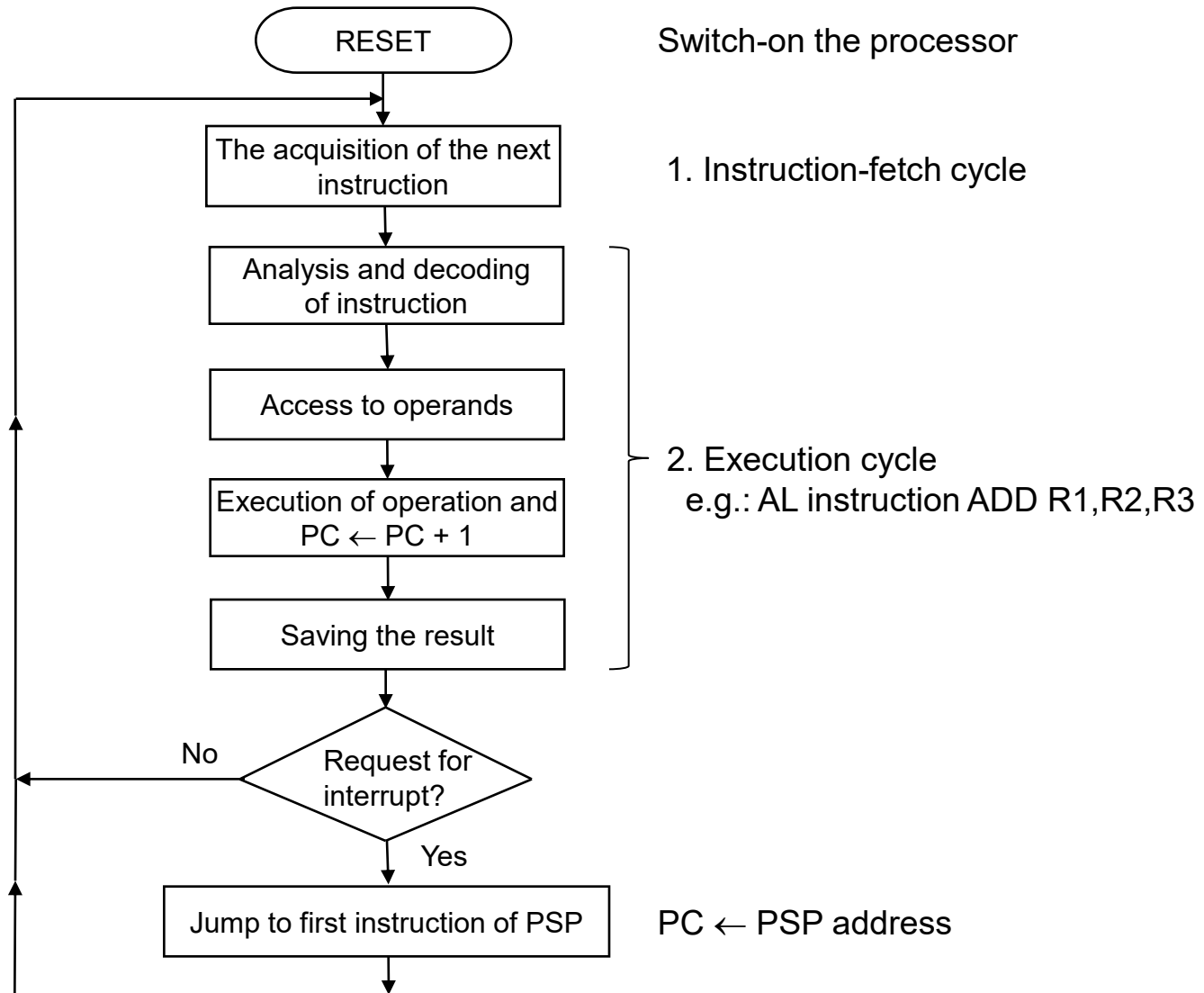
- The basic operation of the CPU in the Von Neumann computer was described using two steps:
  - 1. Taking instruction from memory (instruction-fetch cycle), the address of the instruction is in the program counter (PC)
  - 2. Execution of the fetched instruction (execution cycle),
  
- Each of these two main steps can be divided on even simpler sub-operations ( "Elementary" steps) ->



- The operation of the CPU in the Von Neumann computer was described using two steps:
  - 1. Taking instruction from memory (instruction-fetch cycle), the address of the instruction is in the program counter (PC)
  - 2. Execution of the fetched instruction (execution cycle), which can be divided to more sub-operations:
    - Analysis (decoding) the instruction
    - Transfer the operands in the CPU (if not already included in the CPU registers)
    - Execution of the instruction's specific operation
    - $PC \leftarrow PC + 1$  or  $PC \leftarrow \text{target address}$  in branch instructions
    - Saving the result (if necessary)

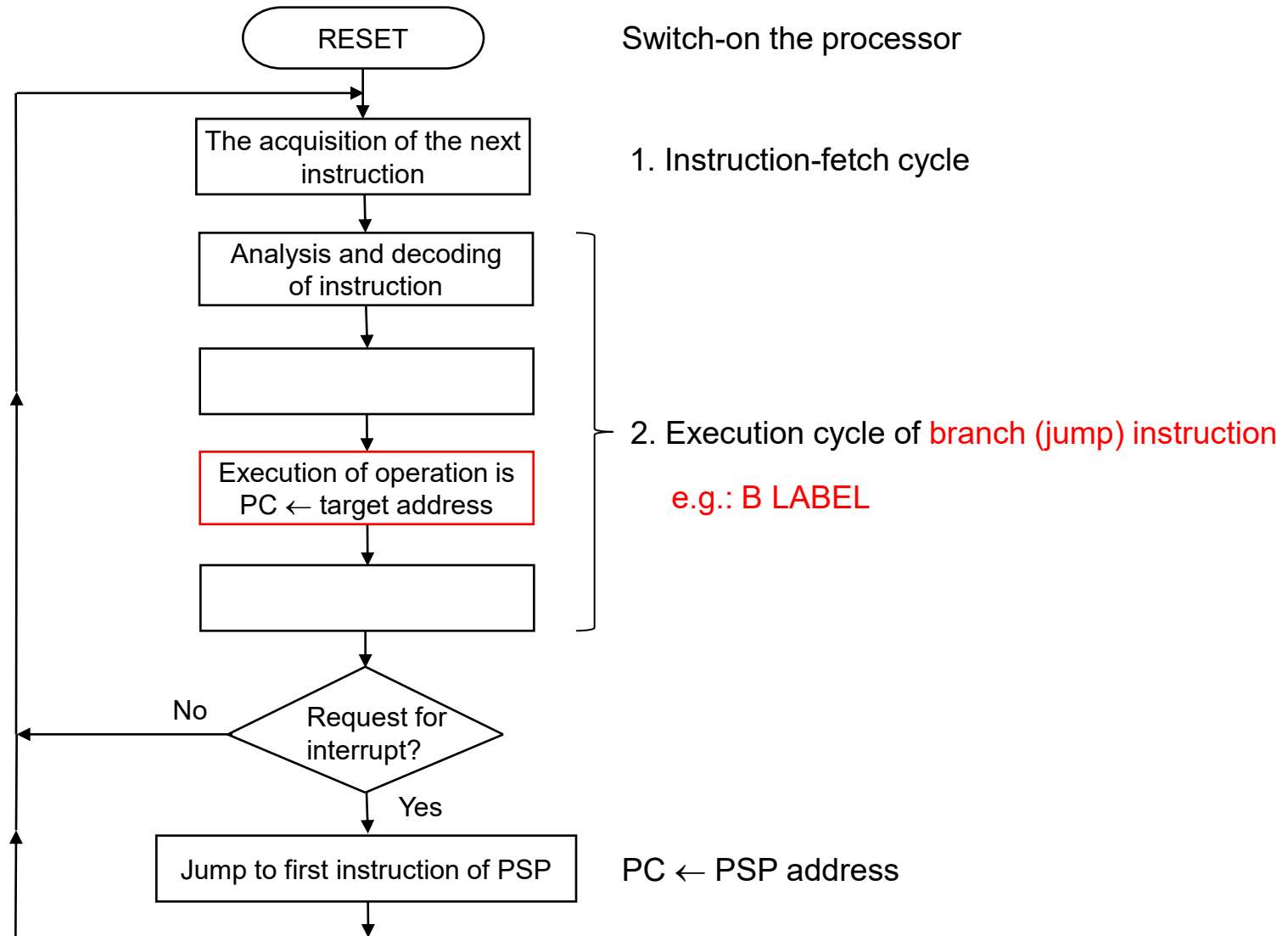


## Central processing unit





## Central processing unit

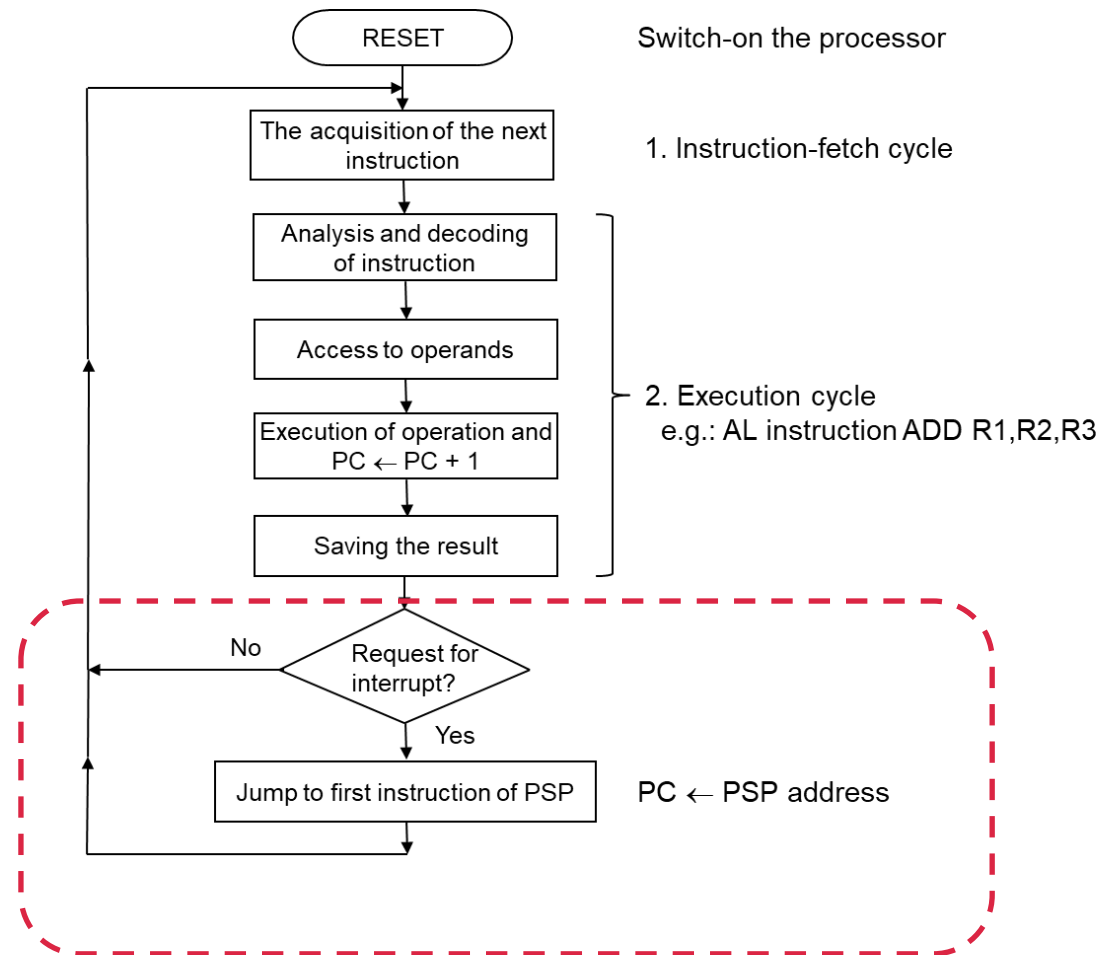




## Central processing unit

### Interrupts or traps:

- extra-ordinary events
- transparency important
  
- instead of next instruction, branch to first instruction of ISR (Interr. Service Routine) is executed.



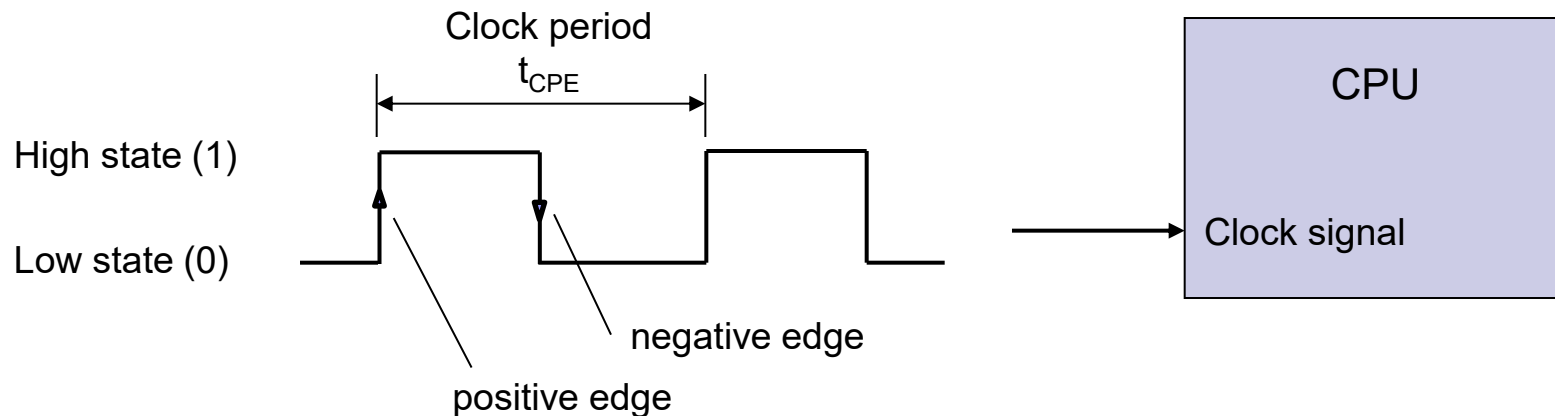


- The address of the first instruction after switching on (RESET) is determined by a certain rule.
- Upon completion of Step 2, the CPU starts again with the first step, which is repeated, as long as the CPU operates.
- The exception is when there is an interrupt or trap request.
- On such request, instead of fetching the next instruction, the jump instruction is executed to the address that is determined by the mode of interrupt or trap operation.



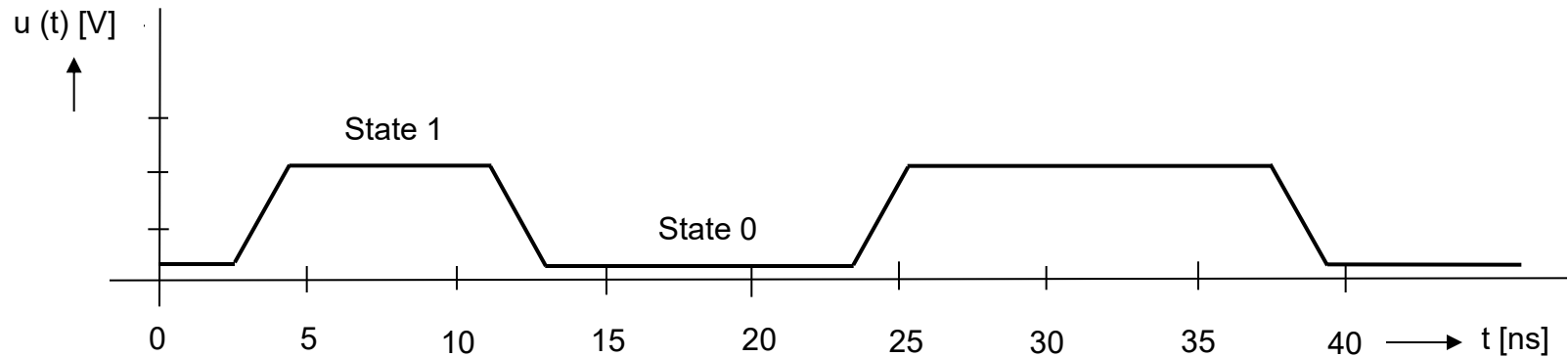
## Central processing unit

- Each of these steps is composed of more elementary steps and realization of CPU is basically the realization of these elementary steps.
- Each elementary step is carried out in one or more periods of clock signal - CPU clock.

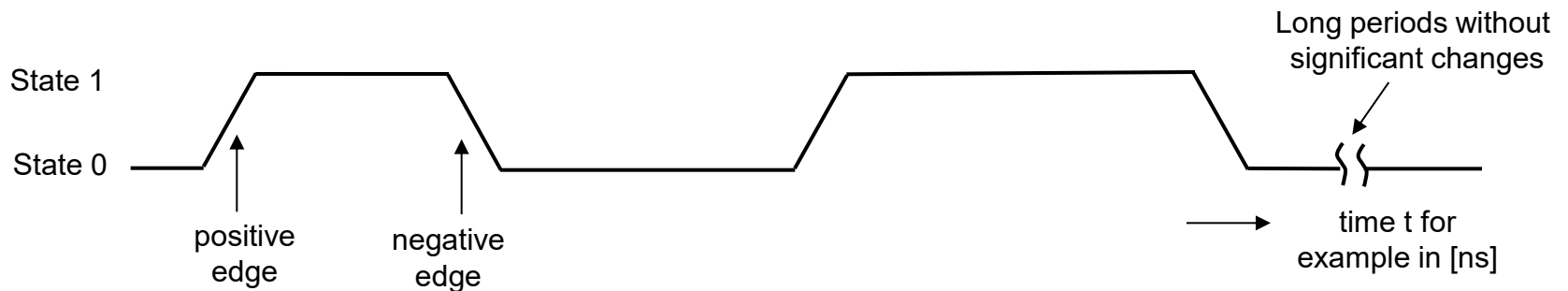




### Arbitrary (non-periodic) digital electrical signal



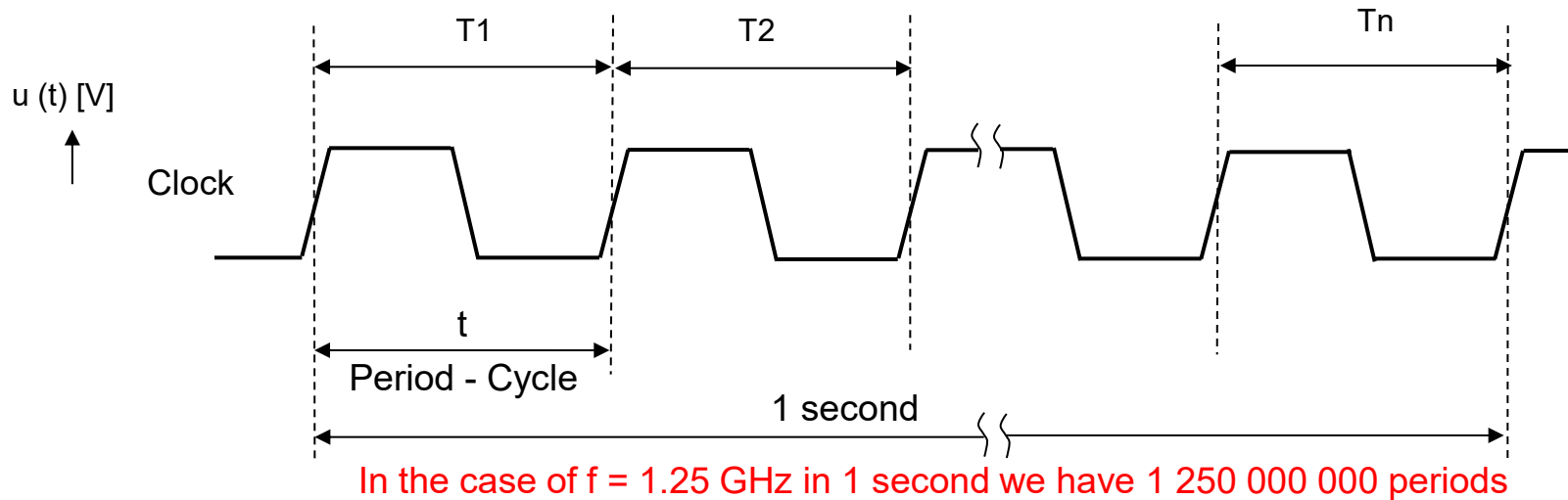
### Arbitrary (non-periodic) digital electrical signal - logical presentation







## Clock signal - periodic rectangular signal



The frequency of the periodic signal  $f$  = number of periods (cycles) in 1 second

The unit of frequency is Hertz [Hz]:  $1 \text{ Hz} = 1 [\text{Period/sec}] = 1 [1/\text{s}] = 1[\text{s}^{-1}]$

The duration of one period  $T = 1 / f$

$$f = 1,25[\text{GHz}] \Rightarrow t = \frac{1}{f} = \frac{1}{1,25 * 10^9 [1/\text{s}]} = \frac{1}{1,25} * 10^{-9} [\text{s}] = 0,8 * 10^{-9} [\text{s}] = 0,8[\text{ns}]$$



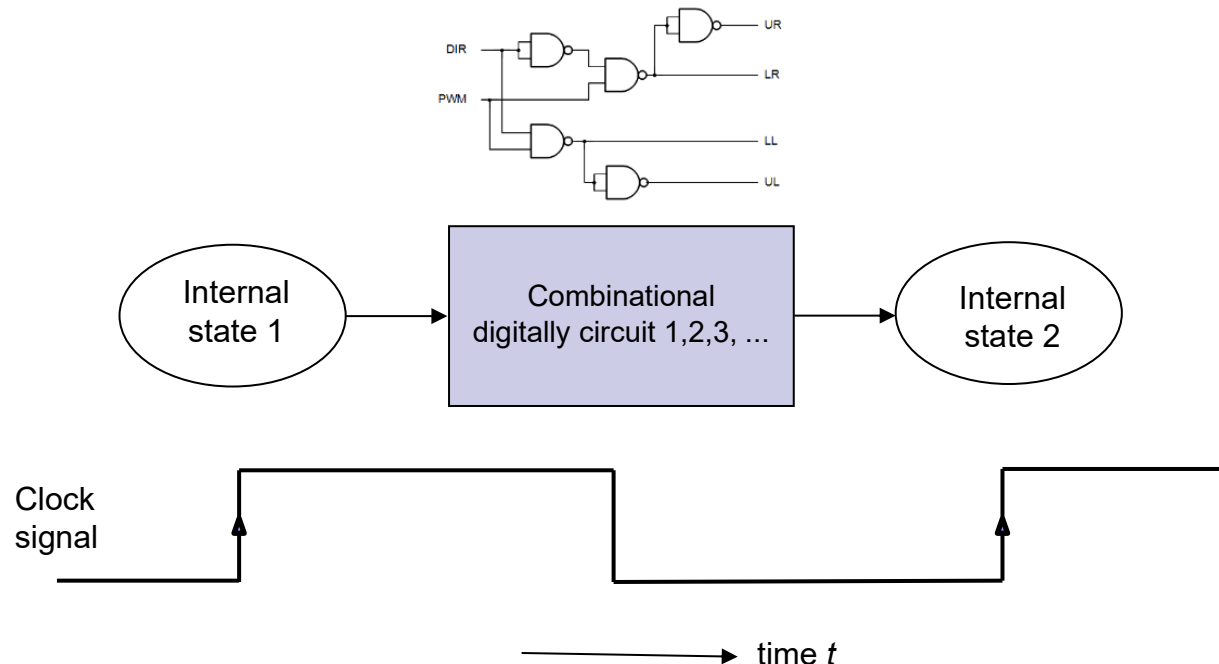
- The state of the CPU, such as the states of all synchronous digital circuits, changing only at the edge of the clock signal (clock signal transition from one state to another).
- Edge, at which the changes happen in the CPU, is called **active edge**.
- CPU can also change the state at the positive and negative edges, this means that both edges are active. In one clock cycle, two changes of the CPU state can be performed.

*Why is the clock signal needed at all? 2 points of view ->*



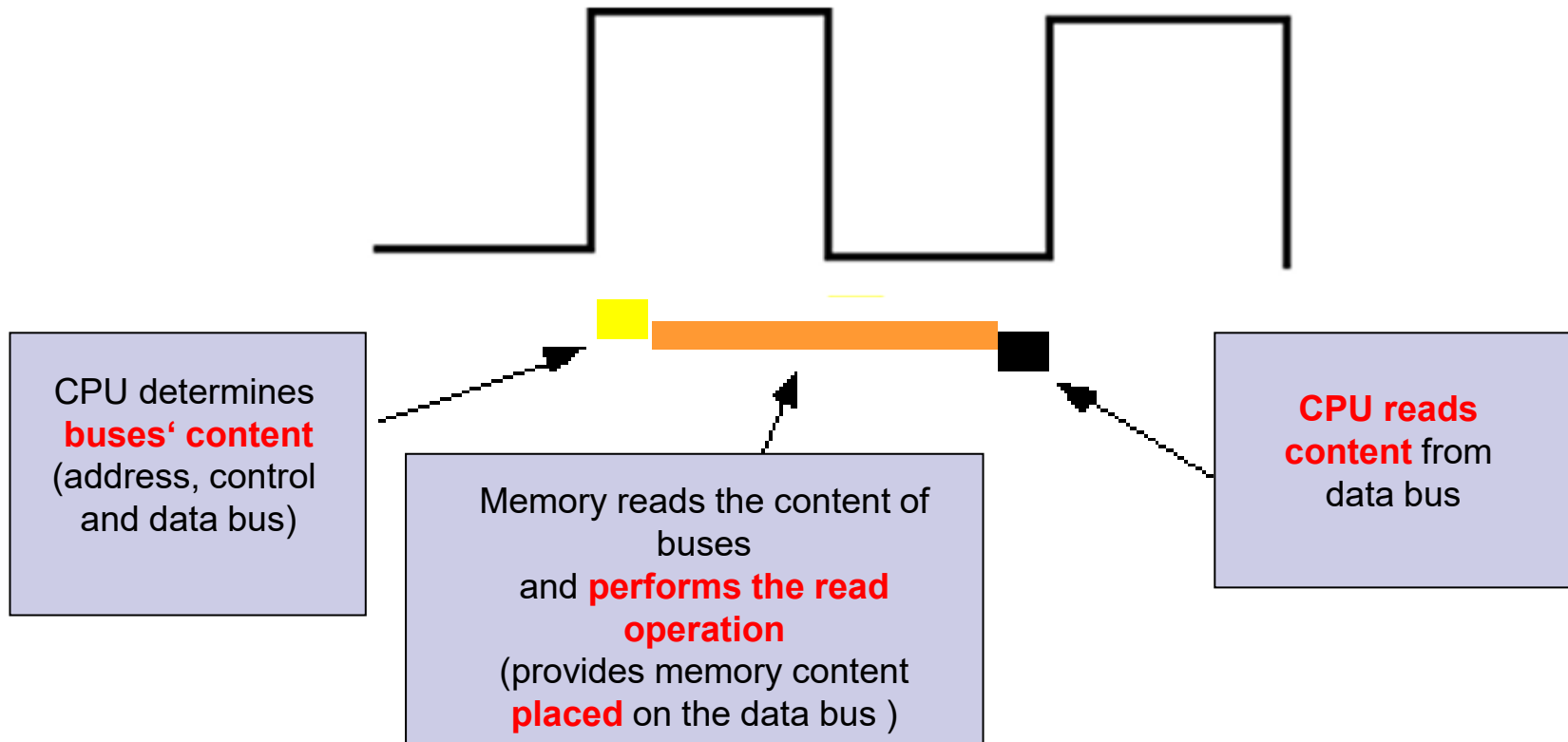
## Central processing unit

- Clock signal -> synchronization of combinational circuits with various speeds
  - In synchronous digital memory (sequential) system clock signal (usually edge) provides a moment of change to the internal state of the memory digital circuit.
  - When the input signals in the memory circuit becomes stable, at the active edge the change of the internal state of the memory circuit can occur.





- Clock signal -> synchronization of **multi-speed operations** in computer
  - For example, access to memory in one clock cycle (read operation):





- State of CPU changes on the edges of the internal clock. Shorter clock period (higher frequency) means faster performance of CPU.
- Shortening the clock period (increasing frequency) is determined by the speed of the digital circuits and the number of circuits (length of links) through which the signal propagates.
- The minimum duration of the elementary step in the CPU is one clock period (or even half-period, if both edges are active, but this requires more complex circuit).
- Fetch and execution cycles' duration is always an integer number of periods.
- Number of periods for the execution of the instruction can vary greatly.



## 6.2 ARM CPU – features summary

*Much more related details explained on LAB sessions*

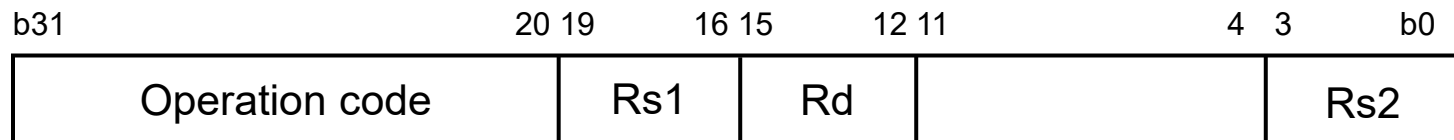
- RISC architecture (some exceptions)
- 3-operand register-register (load/store) computer
  - Access to the memory operands is only by using the LOAD and STORE
- 32-bit computer (FRI-SMS, ARM9, architecture ARMv5)
  - 32-bit memory address
  - 32-bit data bus,
  - 32-bit registers
  - 32-bit ALE
- 16 general purpose 32-bit registers
- Length of the memory operand 8, 16 and 32 bits
- Signed numbers are represented in two's complement
- Real numbers in accordance with standard IEEE-754 (in case of FP-unit)



- Composed memory operands are stored under the rule of little endian.
- The instructions and operands must be aligned in memory (stored on the natural addresses).
- All of the instructions are 32 bits long (4 bytes).
- ARM uses all three general addressing modes:
  - Immediate *ADD R1, R1, #1*
  - Direct (register) *ADD r1, r1, r2*
  - Indirect (register) - LOAD/STORE *LDR r1, [r0]*



- Instructions for conditional branches use PC-relative addressing.
- Example of format for ALU instruction:







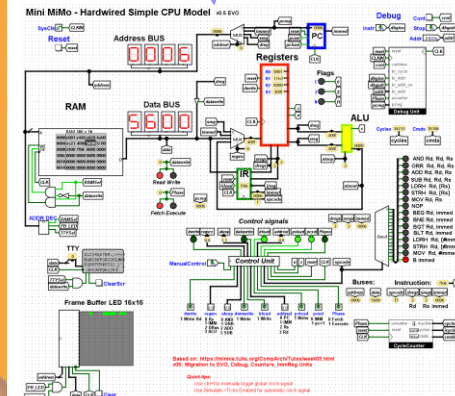
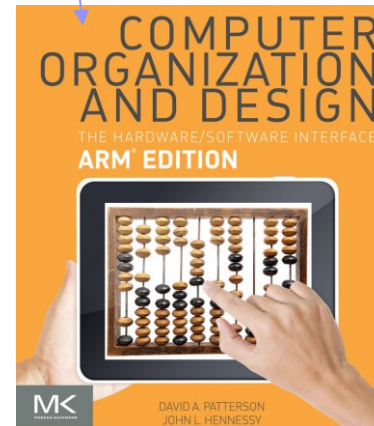
## 6.3 Structure of the CPU (example of ARM CPE LEGv8 & [Mini]MiMo)

### ■ 6.3.1 Data path (unit)

- ALU
- software accessible registers

### ■ 6.3.2 Control unit

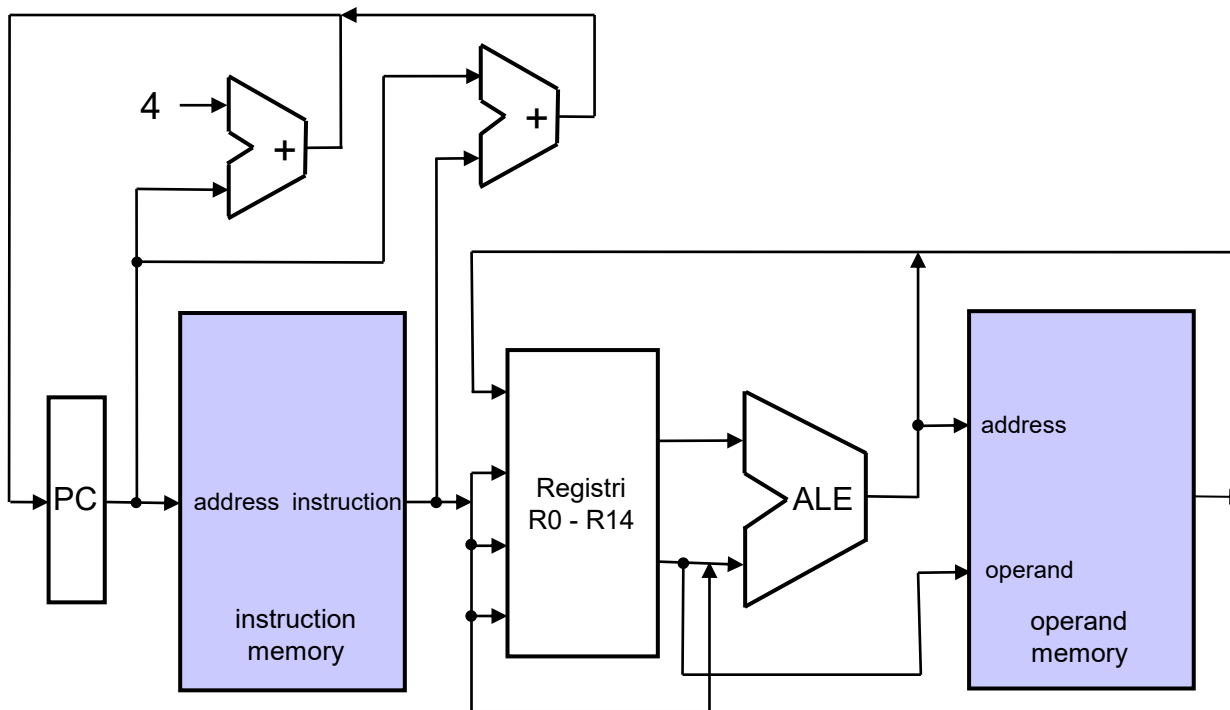
- Realization
  - Micro-programmed (SW) or
  - Hardwired (HW)





### 6.3.1 Data path (unit)

The simplified structure of the CPU data paths including instruction and operand memories

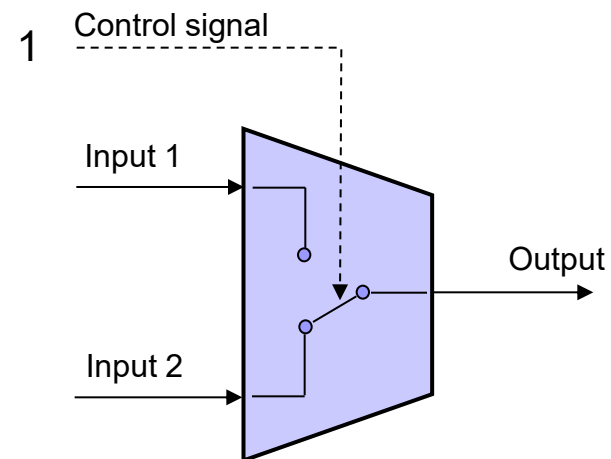
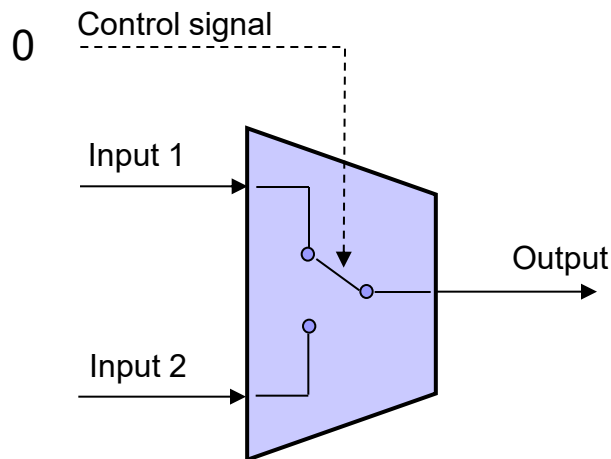


All data paths are M-bit, arrows indicate the direction of transfer



## Central processing unit - structure

- MUX - multiplexer: the digital circuit, that selects one from multiple input signals and connects it to the output.
- Selection of the input signal is determined by control signal.

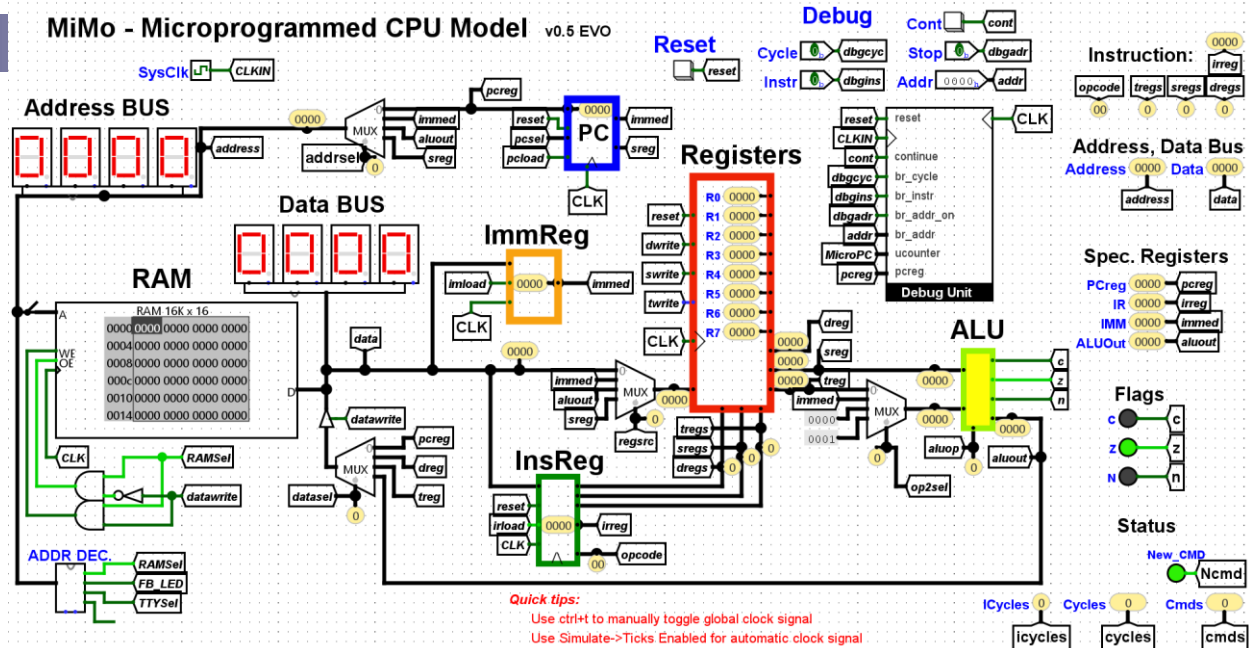


# Model of CPU: MiMo

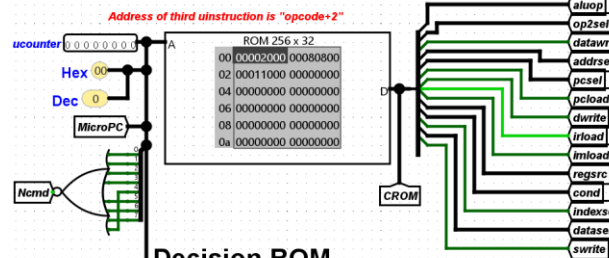
## Model of CPU implemented with logic gates in Logisim

## MiMo – Microprogrammed Model of CPU (course OR VSP)

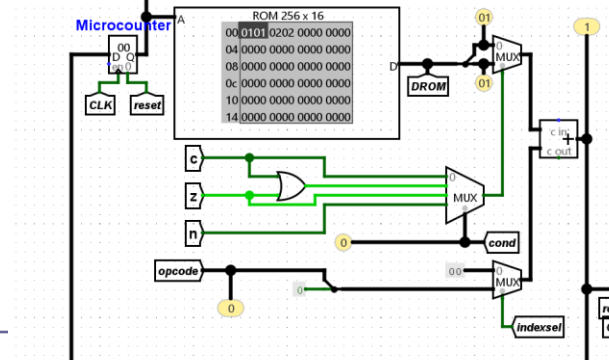
### MiMo - Microprogrammed CPU Model v0.5 EVO



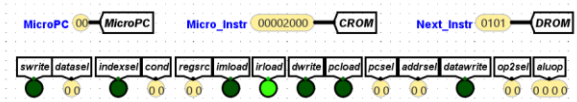
### Microcode Control Unit Control ROM



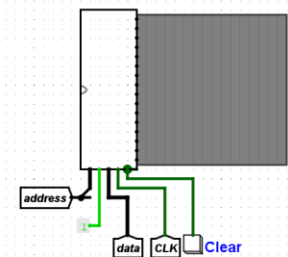
### Decision ROM



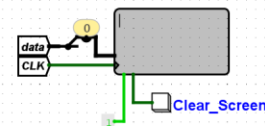
### Micro Instruction



### Frame Buffer LED 16x16



### TTY





# Model of CPU: MiMo

Model of CPU implemented with logic gates in Logisim

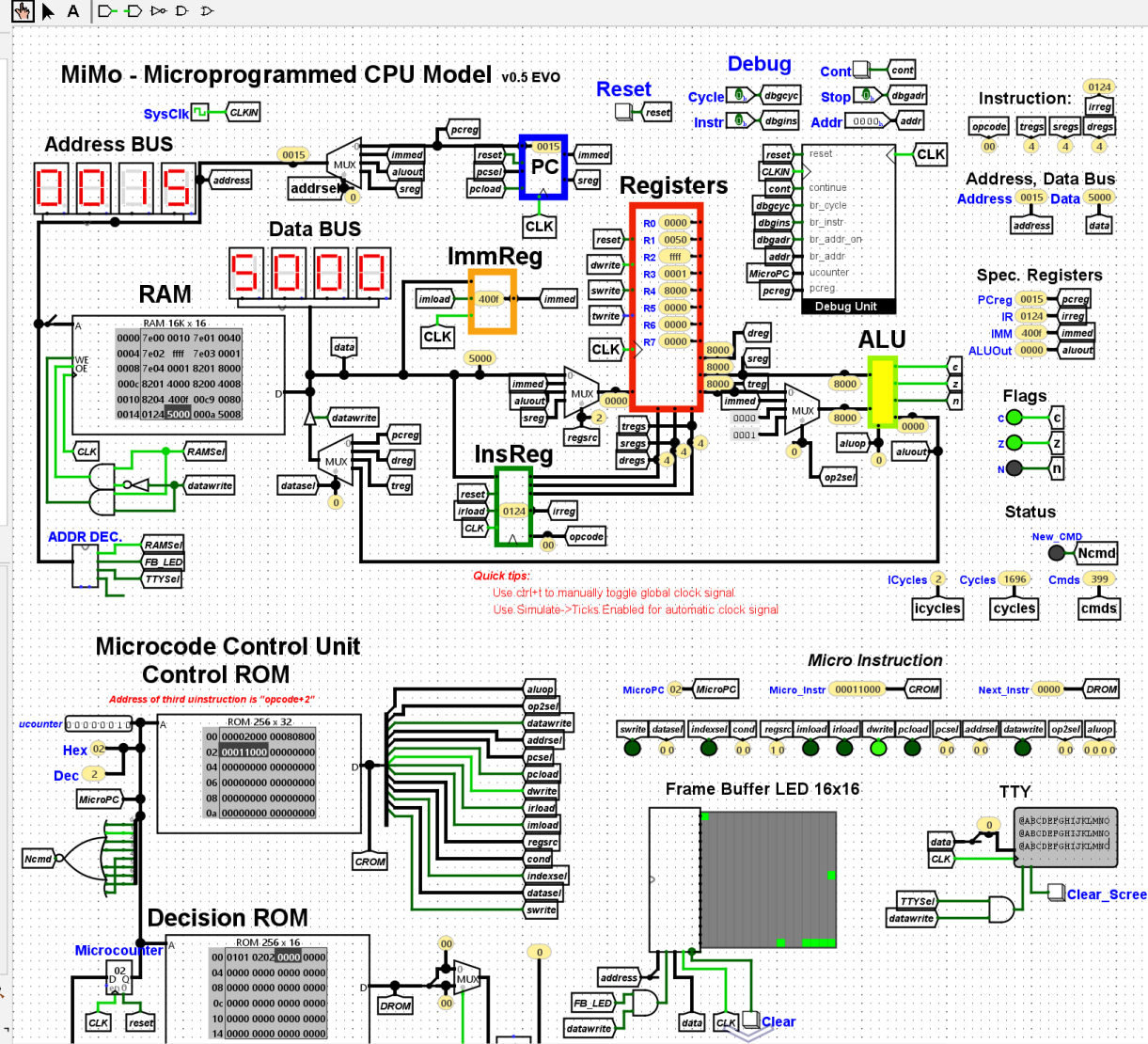
# MiMo – Microprogramm ed Model of CPU Video

Design Simulate

- \*mimo\_v05\_EVO
  - MiMo\_v05
  - InstructionReg
  - Registerfile
  - ALU
  - PC
  - ShowHexa
  - Frame\_Buffer\_16x16
  - Address\_Decoder
  - Counters
  - DebugUnit
  - ImmediateReg
  - Wiring
  - Gates
  - Plexers
  - Arithmetic
  - Memory
  - Input/Output

Properties State

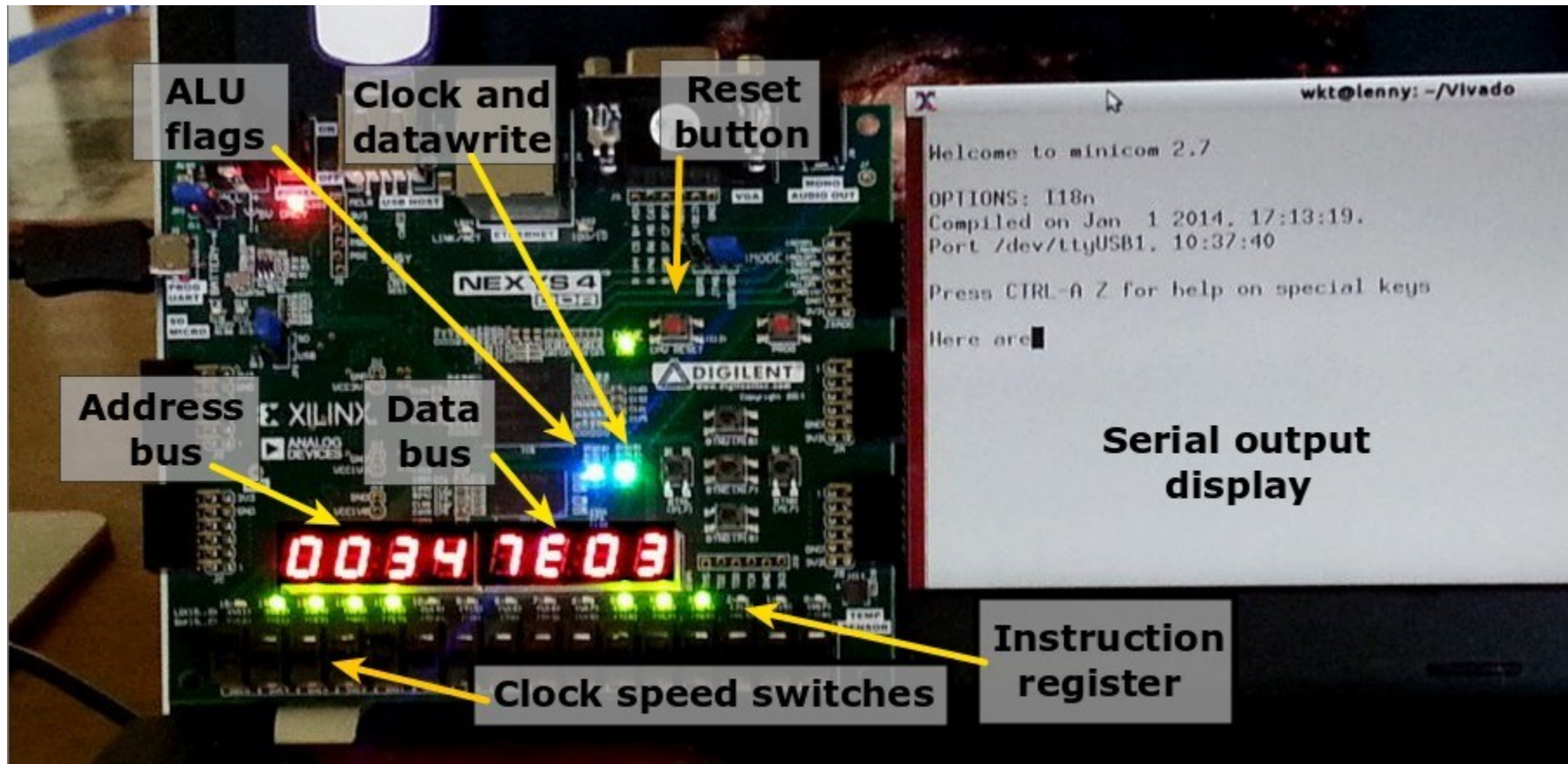
Auto x1/2 x1 x2





# MiMO – Microprogrammed Model of CPU

## FPGA implementation

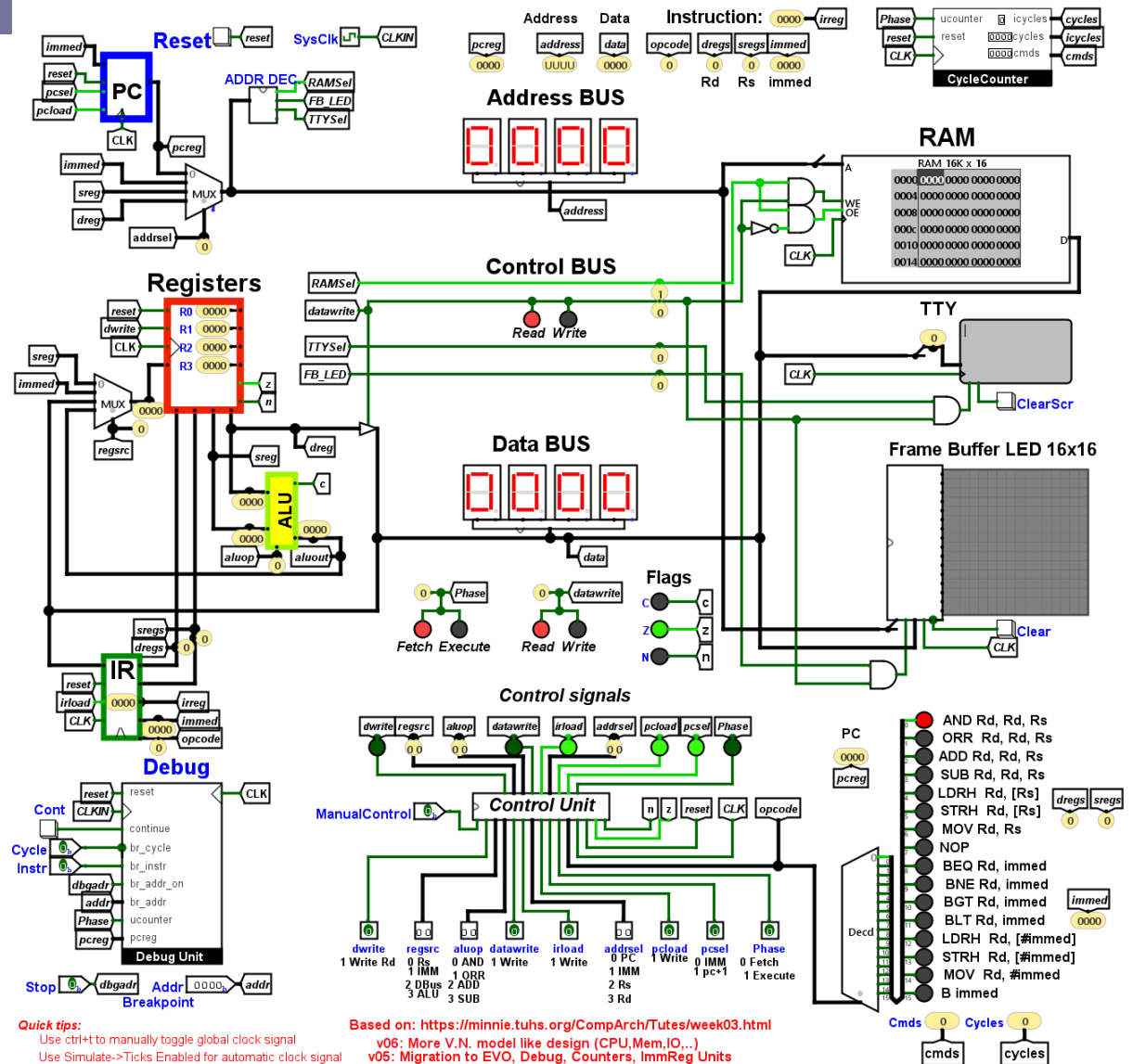


# Model of CPU: Mini MiMo (course RA VSP)

Model of CPU  
implemented with  
logic gates in  
Logisim

Mini MiMo –  
Simple Hardwired  
Model of CPU  
(16 instr., assembler  
in Excel, ...)

Mini MiMo - Hardwired Simple CPU Model v0.6 EVO



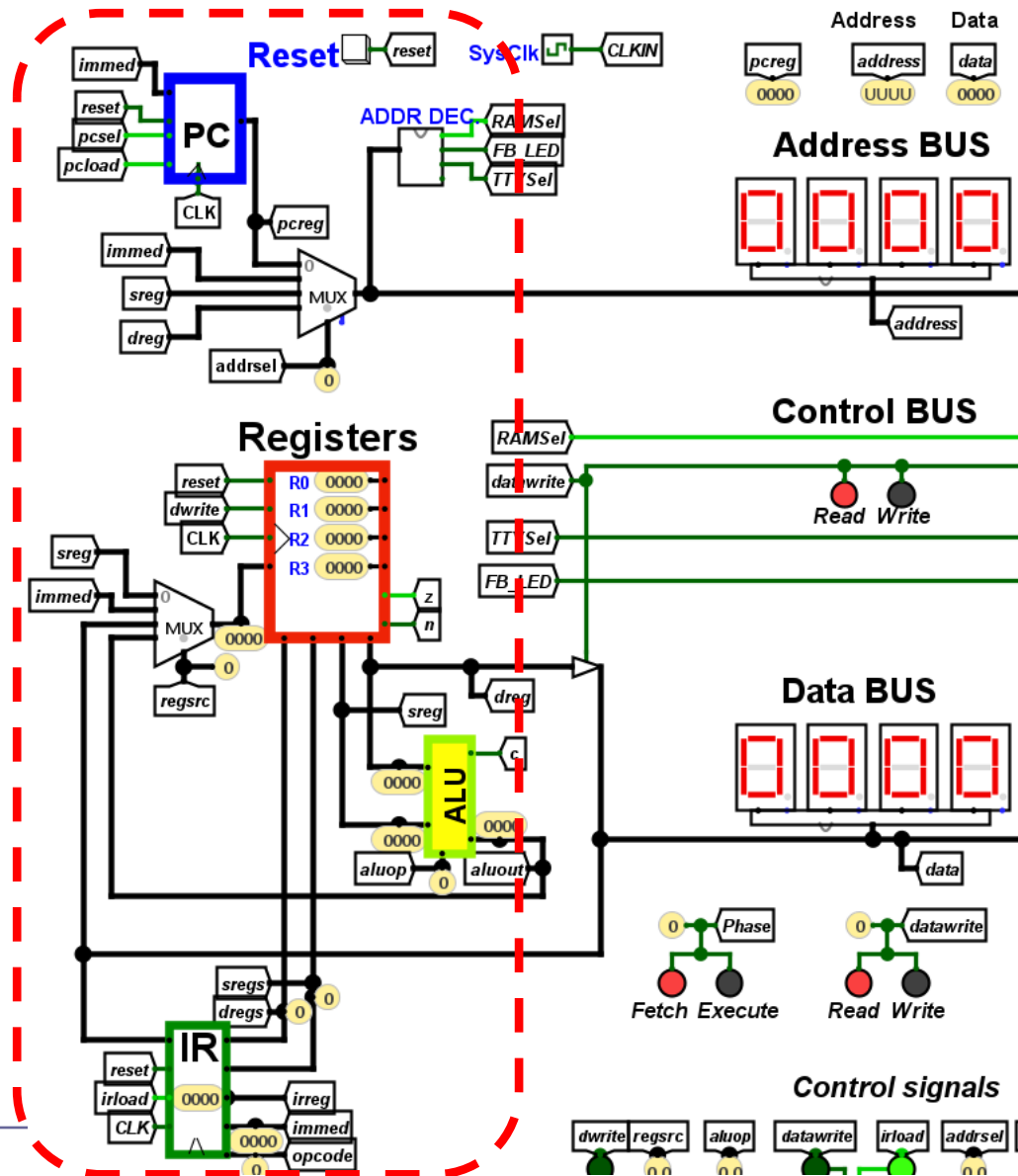
[https://github.com/LAPSyLAB/RALab-STM32H7/tree/main/MiniMiMo\\_HW\\_CPE\\_Model](https://github.com/LAPSyLAB/RALab-STM32H7/tree/main/MiniMiMo_HW_CPE_Model)



# Mini MiMo Datapath

## Central processing unit - structure

### 6.3.1 Data path (unit)

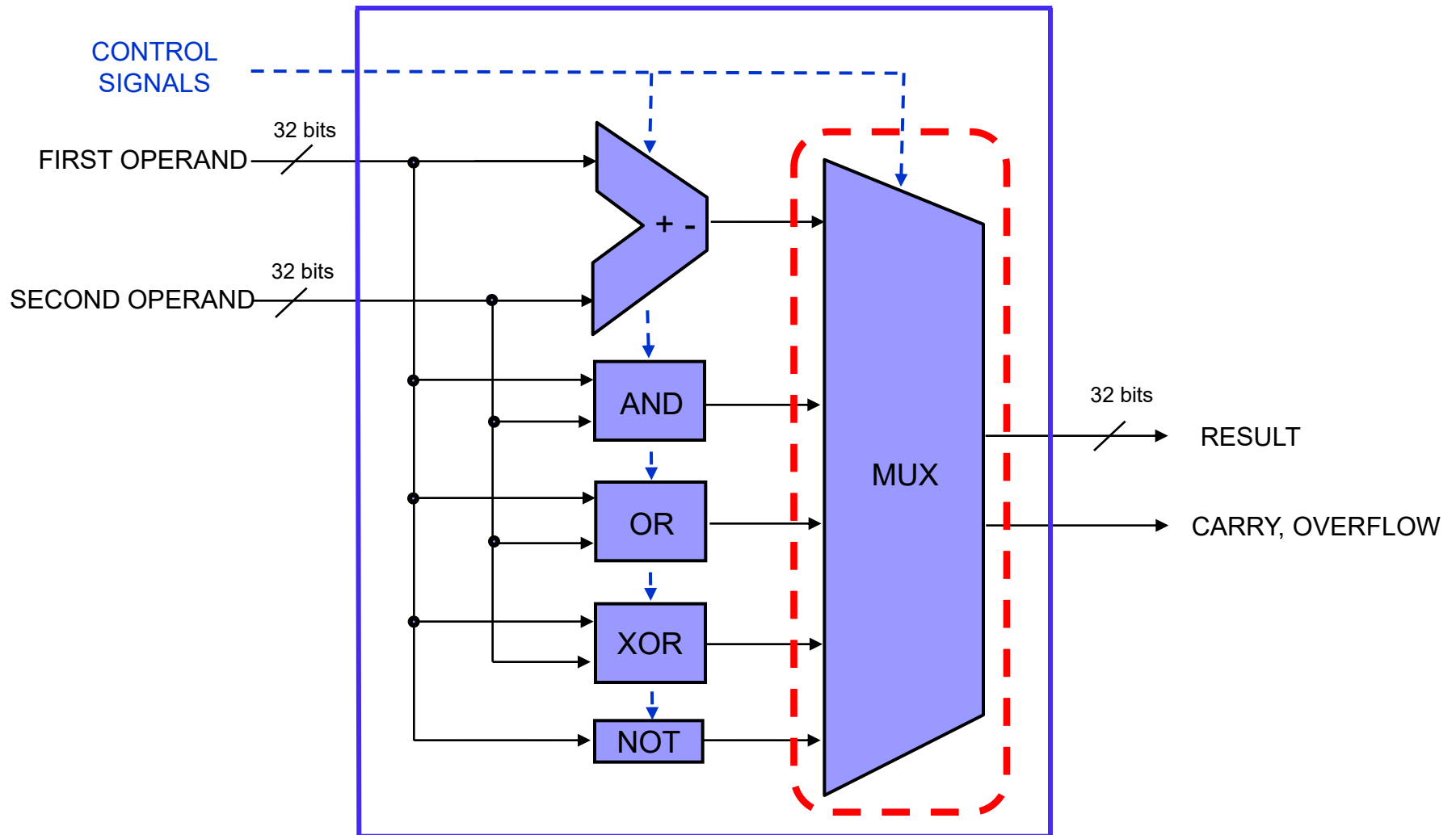


All thicker data paths are more than 1-bit wide





## ALU – datapath and control signals

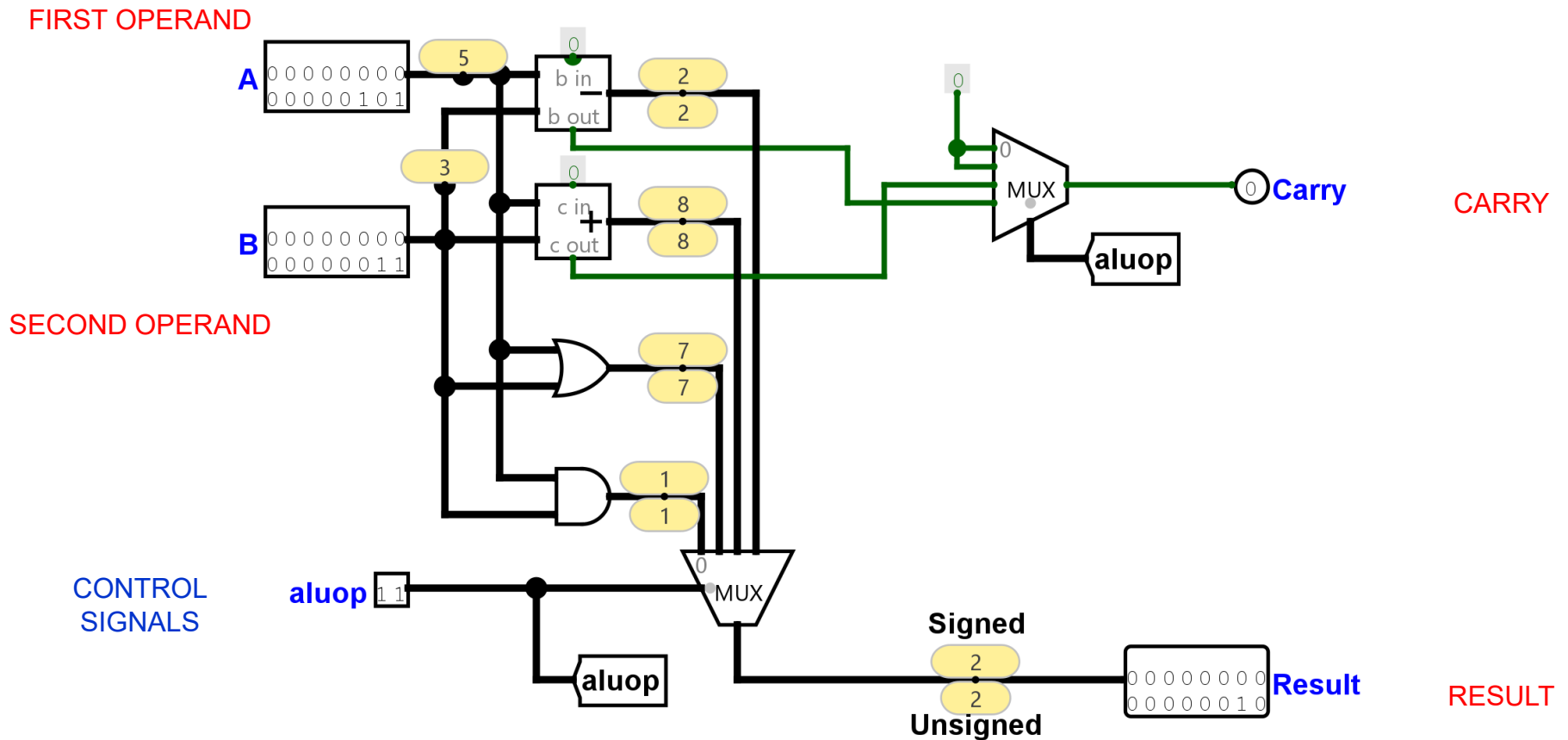




# ALU – datapath and control signals

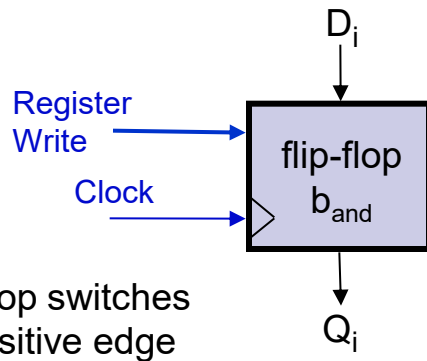
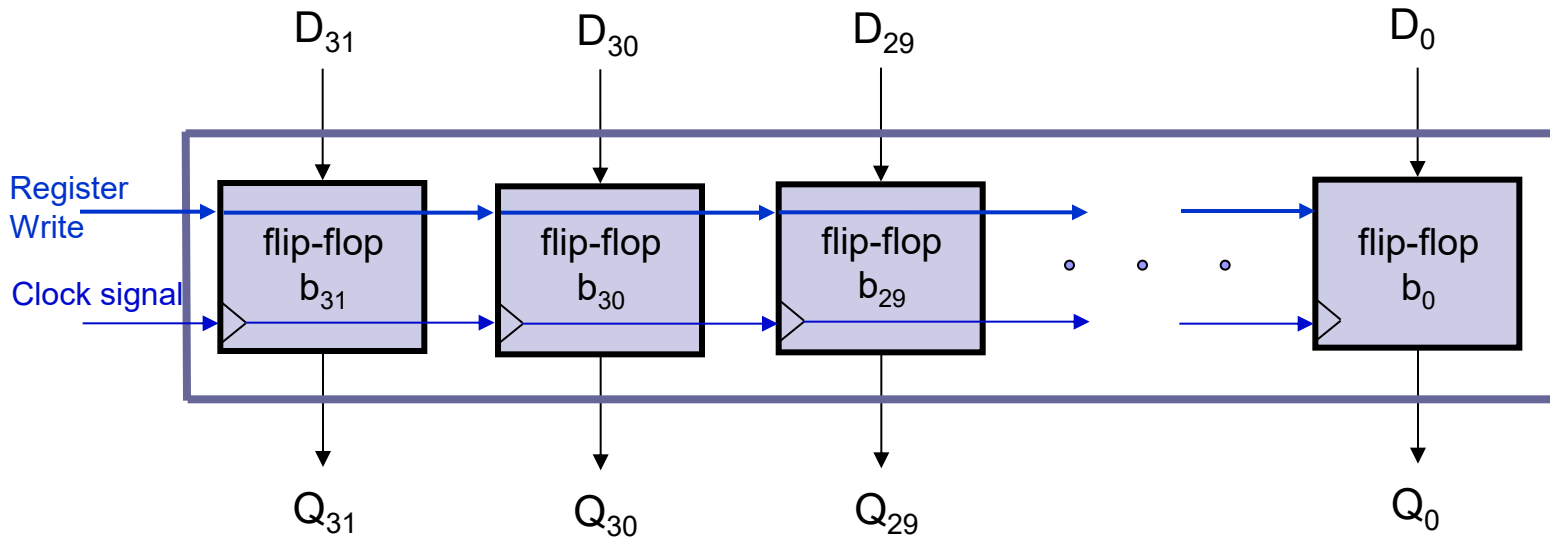
## Case of Mini MiMo CPU

### 16-bit ALU



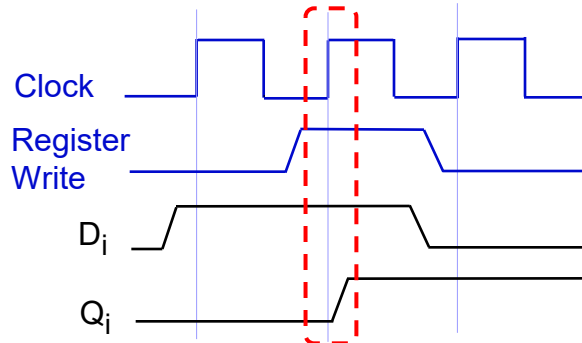


# 32-bit register



Flip-flop switches on positive edge

Timing diagram



Truth Table

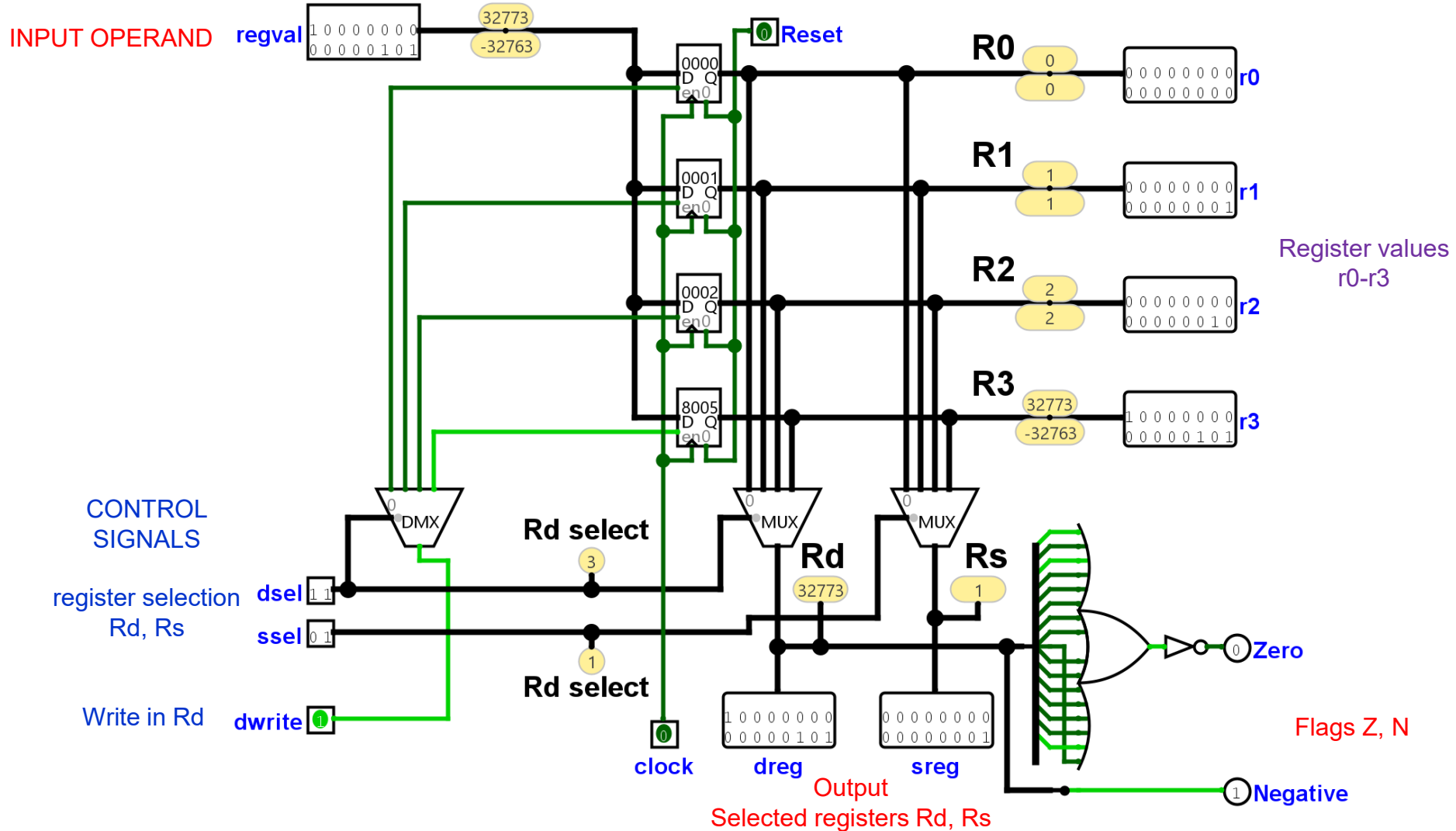
Clock	Reg.W	$D_i$	$Q_i$
↑	0	0	Q
↑	0	1	Q
↑	1	0	0
↑	1	1	1



# Register unit

## Register File

### Case of Mini MIMO CPU



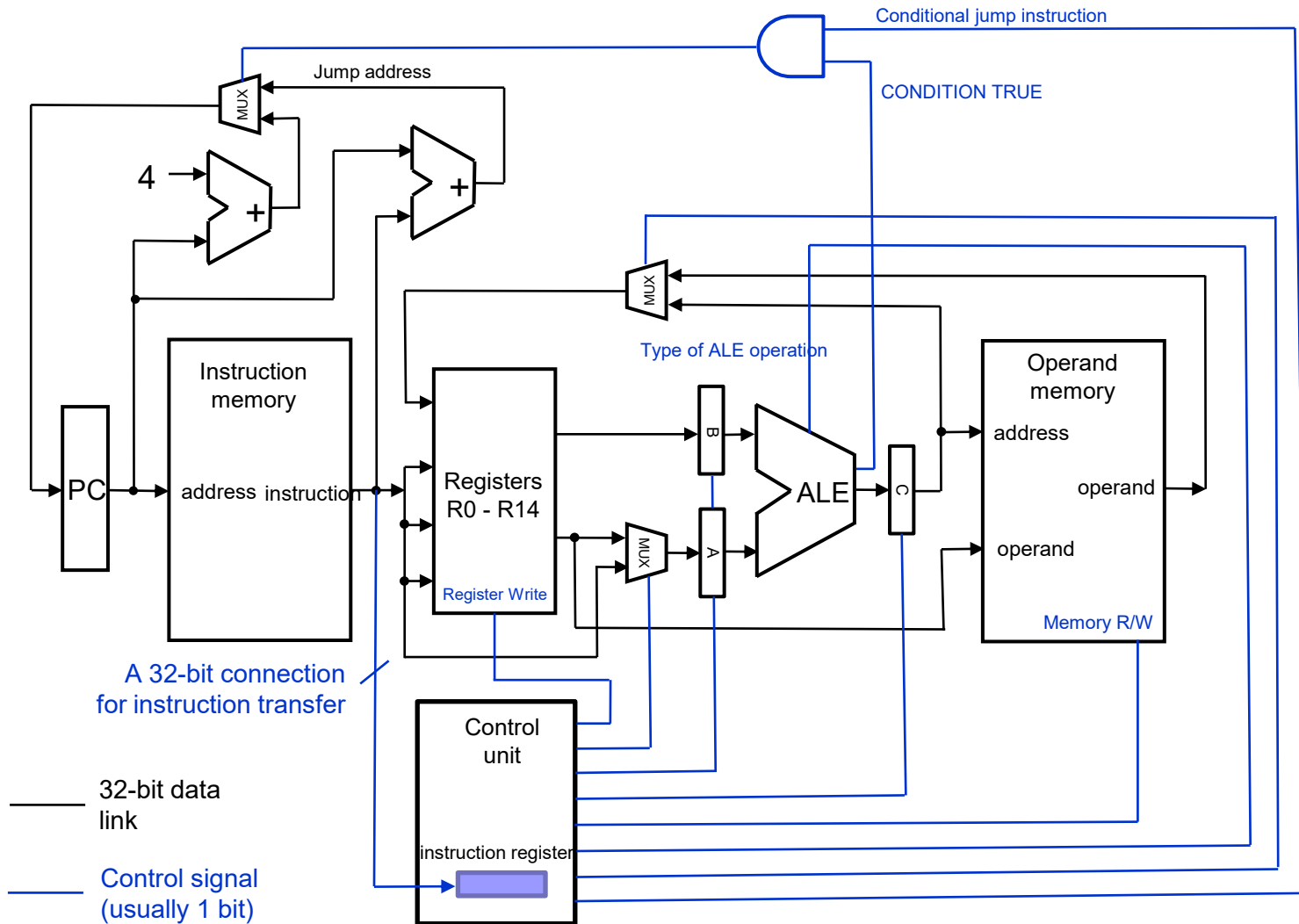


## 6.3.2 Control Unit (CU)

- Is digital circuit (memory + combinational), that on the basis of the content in the instruction (register) determines **control signals**.
- Control signals **trigger elementary steps** in the datapath and consequently the execution of this instruction.
- IR register = 32-bit instruction register in which the instruction is transferred during the instruction-fetch cycle: machine instruction is read from the memory.
  - IR ... "Instruction Register "
- 2 possible ways of CU implementation:
  - Micro programmed (SW: simple, slower)
  - Hard wired (HW: complex, faster)



# CPU: datapath, control unit, and control signals

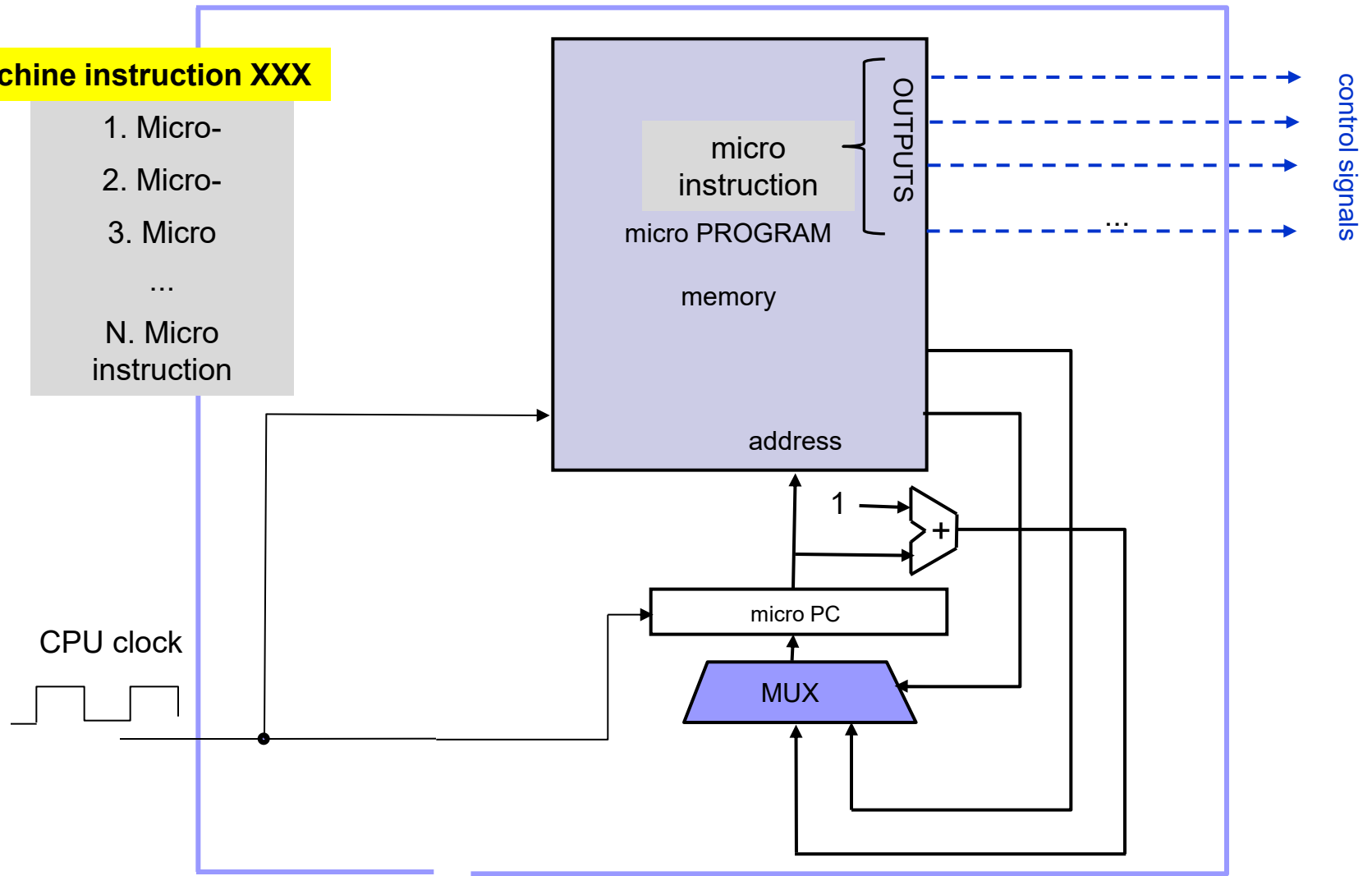




# Control unit (Micro-programmed implementation – e.g. MiMo model)

## Machine instruction XXX

- 1. Micro-
- 2. Micro-
- 3. Micro
- ...
- N. Micro instruction





# Control unit (Micro-programmed implementation –MiMo model)

Micro program for instruction :

JNEZ Rs,immed

Primeri stanj kontrolnih signalov

Machine instr. XXX

- 1. Micro-
- 2. Micro-
- 3. Micro
- ...

Machine instr. XXX

- 1. Micro instr.
- 2. Micro instr.
- 3. Micro instr.
- ...
- N. Micro instr.

JNEZ Rs,immed:

fetch:

addrsel=pc irload=1  
pclload=1 pcsel=pc, opcode jump

40:

addrsel=pc imload=1  
aluop=sub op2sel=const0, if z then pcincr else jump

pcincr:

pclload=1 pcsel=pc, goto fetch

jump:

pclload=1 pcsel=immed, goto fetch

Nasl. vodilo=PC

Vpis v ukazni reg.

PC=Tak. operand

Vpis v PC

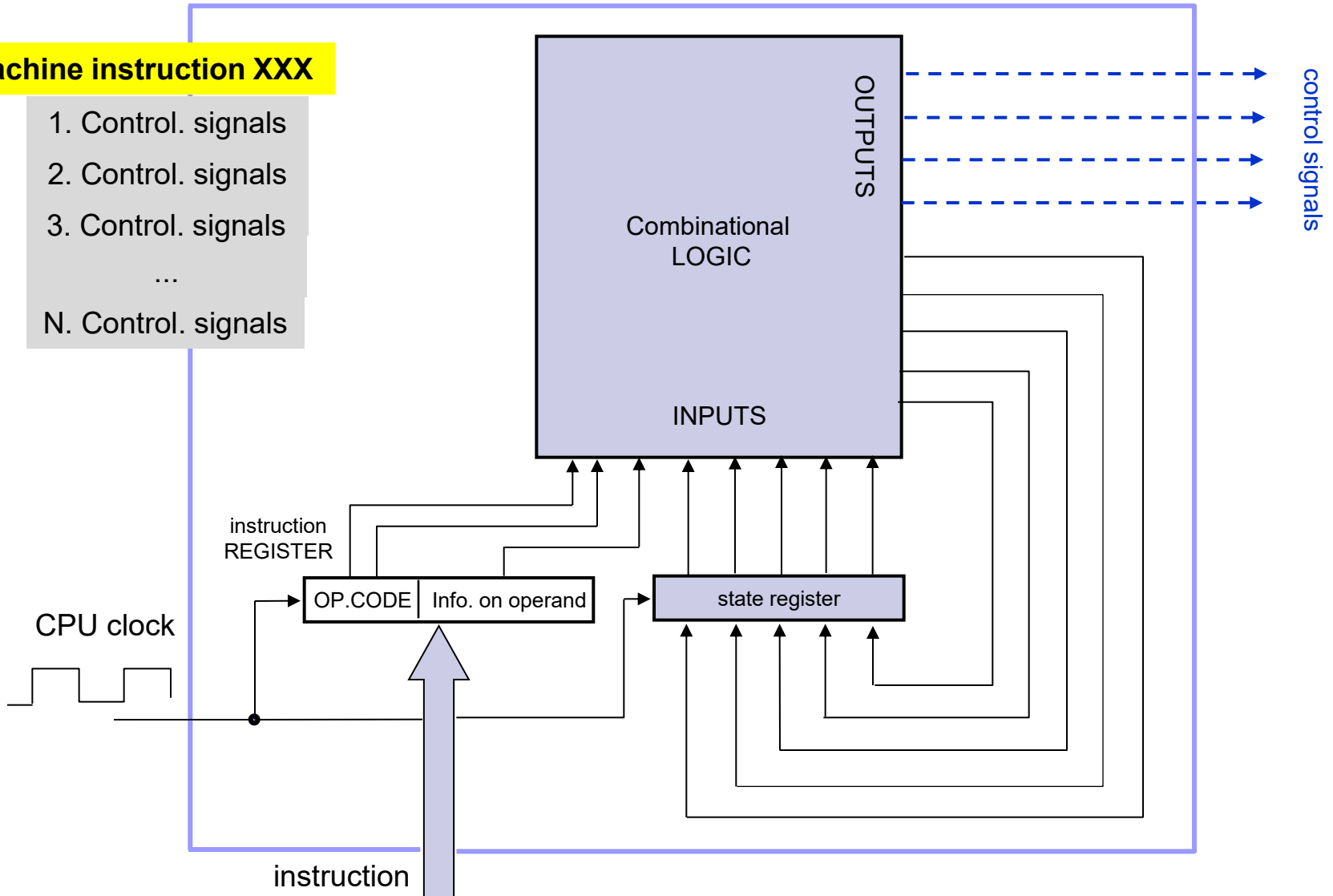




# Control unit (Hard-wired)

## Machine instruction XXX

- 1. Control. signals
- 2. Control. signals
- 3. Control. signals
- ...
- N. Control. signals

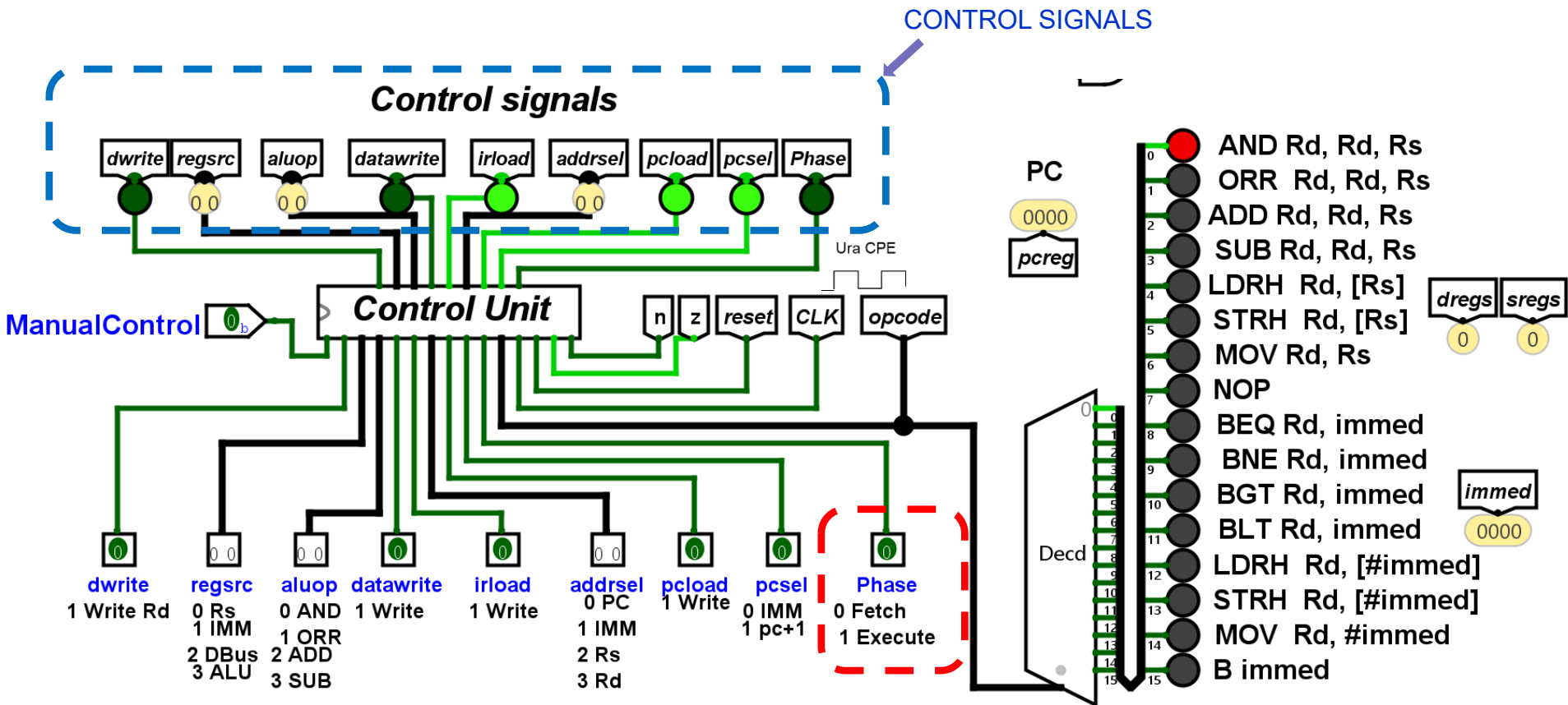




# Control unit (Hard-wired): case Mini MIMO

## Machine instruction XXX

1. FETCH - Control signals
2. EXECUTE - Control signals



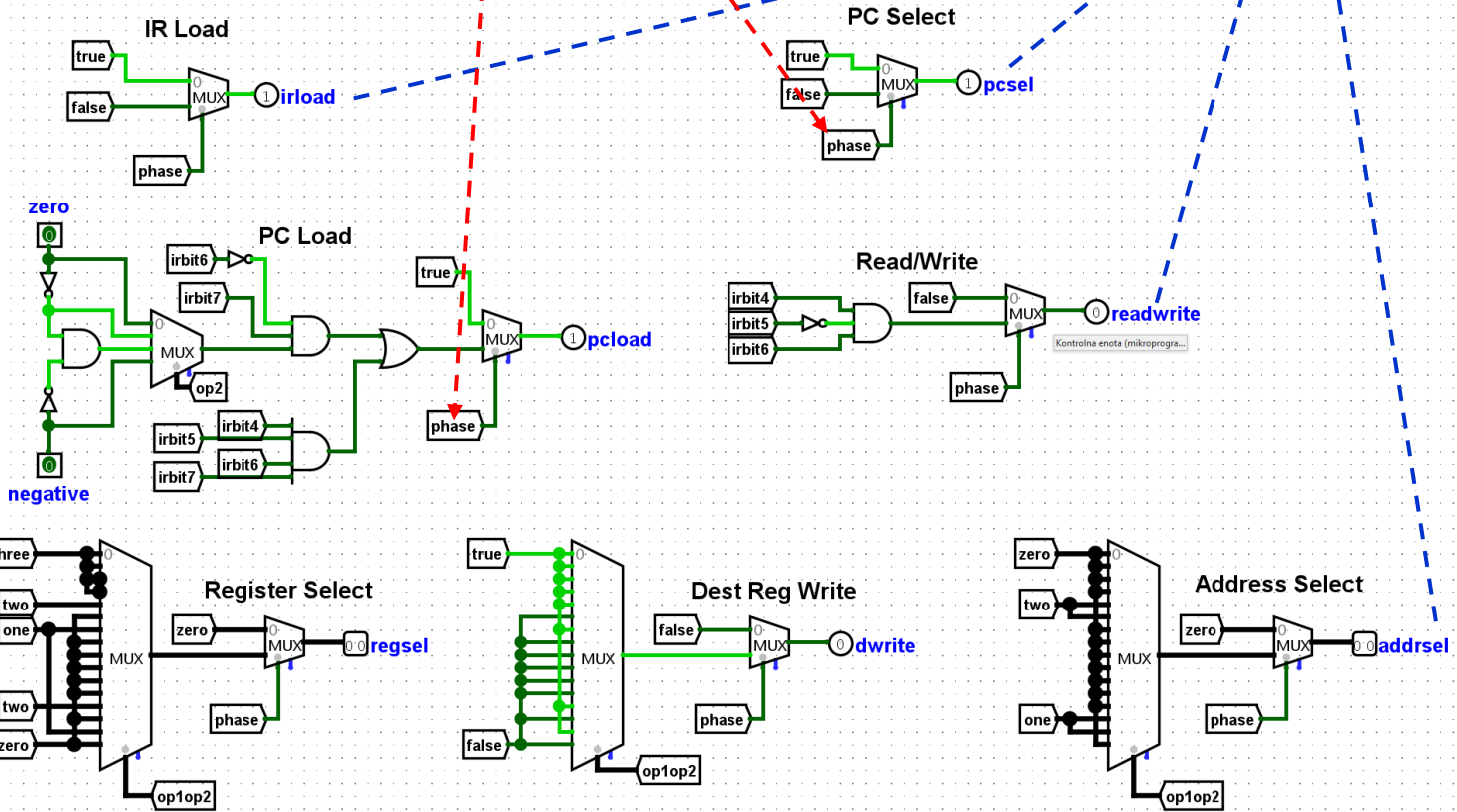
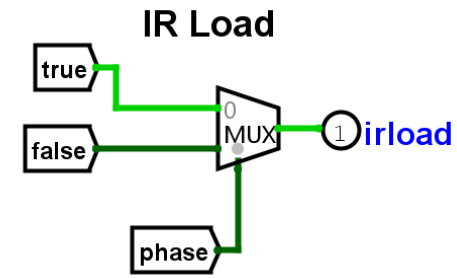
# Control unit (Hard-wired): case Mini Mimo

## Machine instruction XXX

1. FETCH - Control signals
2. EXECUTE - Control signals

INPUTS → COMBINATORIAL CURCUITS → OUTPUTS

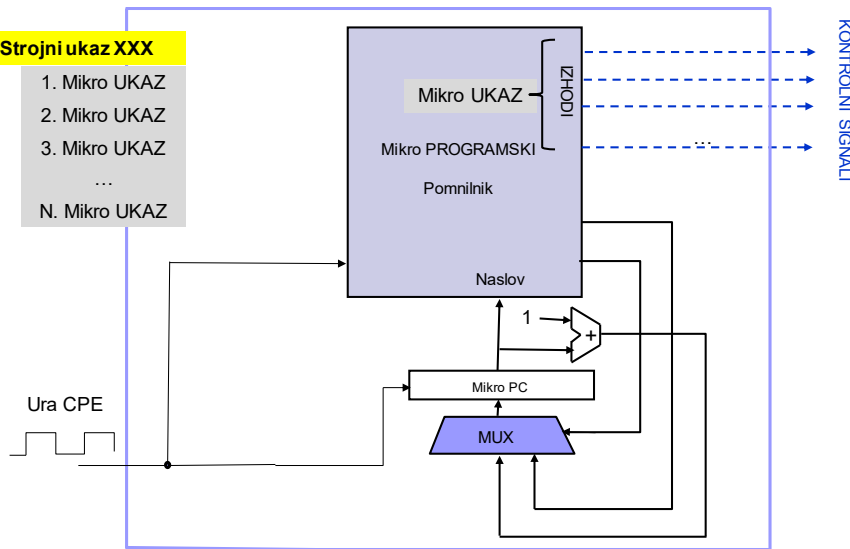
Phase = 0..FETCH,  
1..EXECUTE



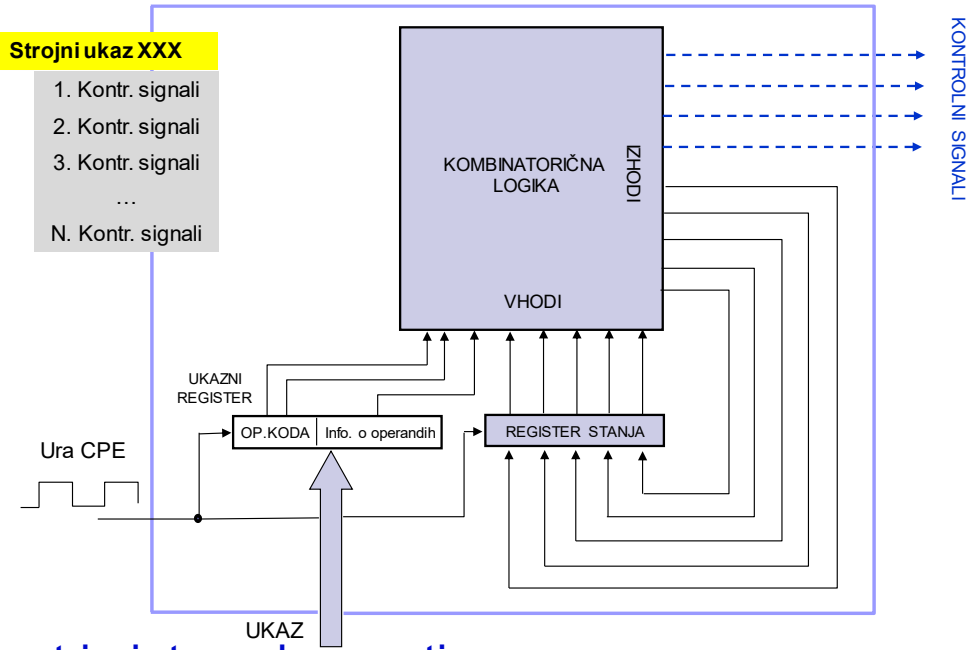


# CU Implementation approaches - Comparison

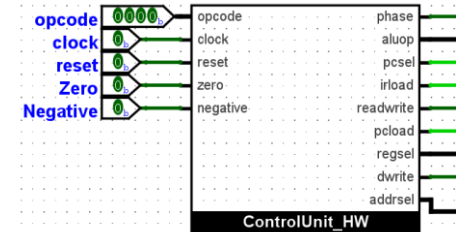
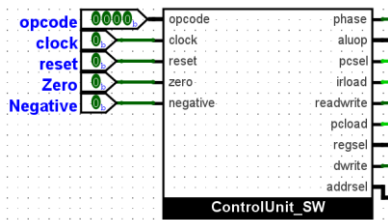
## Control unit (Micro-programmed)



## Control unit (Hard-wired)



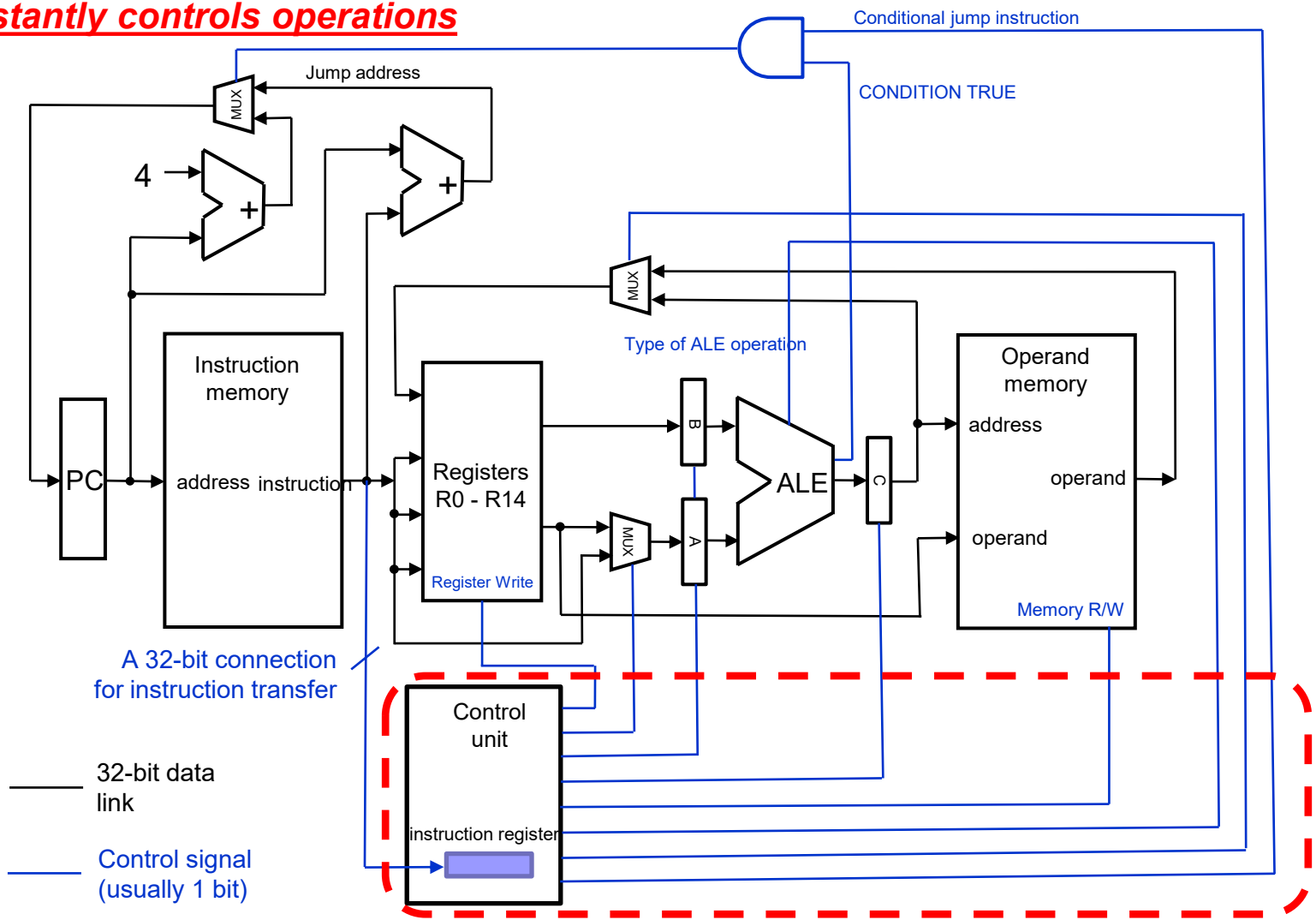
Externally same, different in internal operation





# CPU: datapath, control unit, and control signals

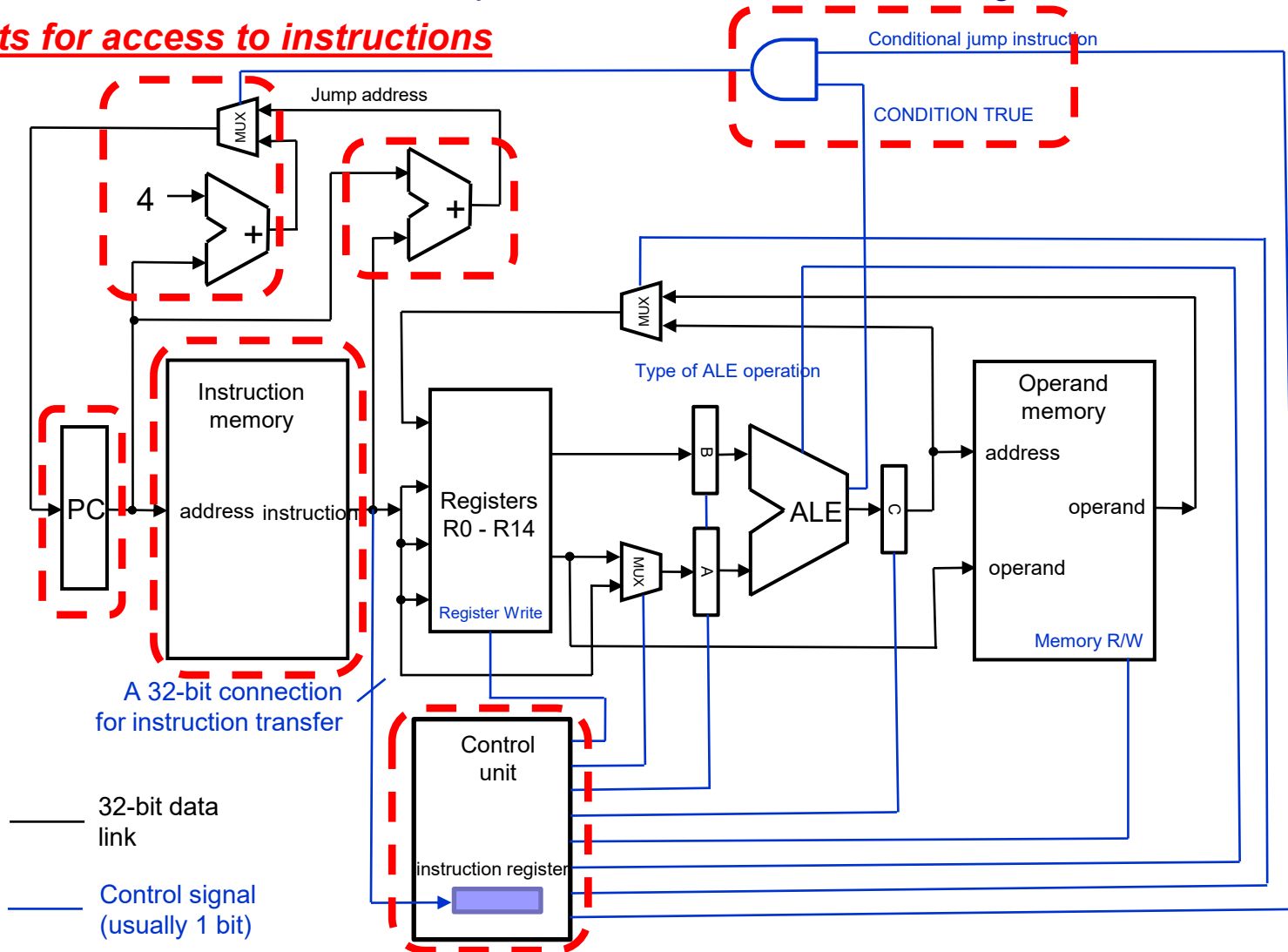
## CU constantly controls operations





# CPU: datapath, control unit, and control signals

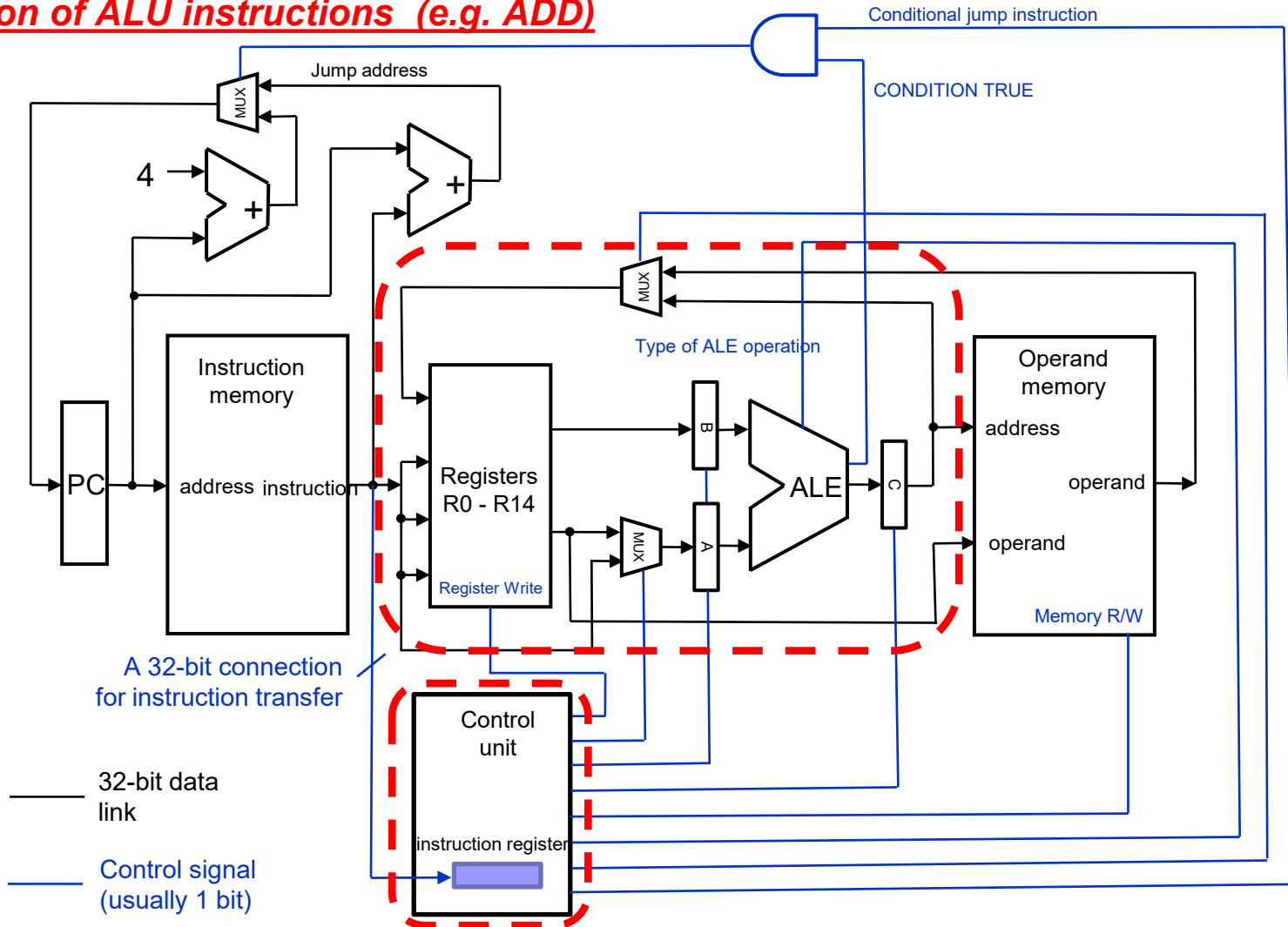
## Elements for access to instructions





## CPU: datapath, control unit, and control signals

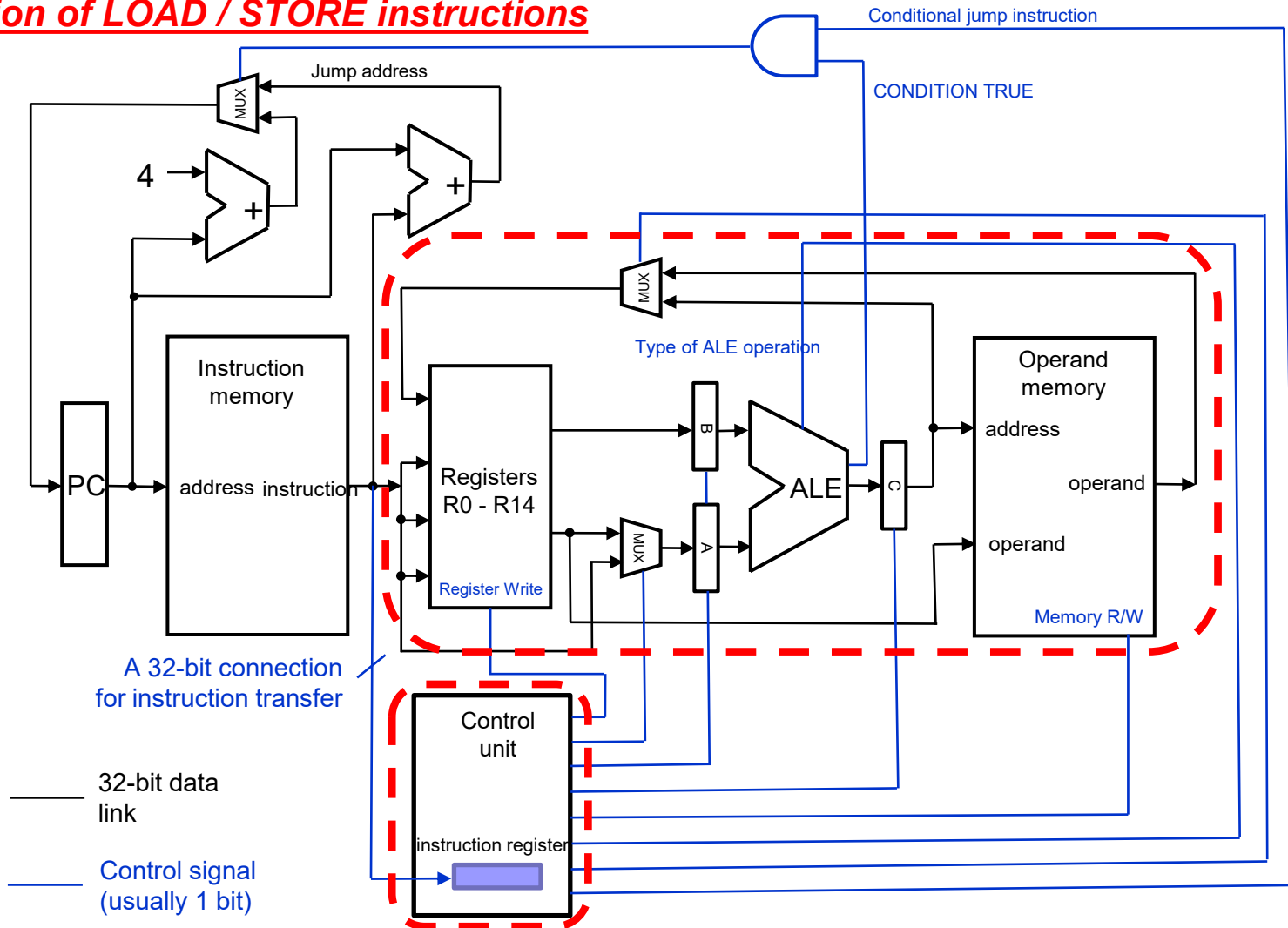
### Execution of ALU instructions (e.g. ADD)





# CPU: datapath, control unit, and control signals

## Execution of LOAD / STORE instructions

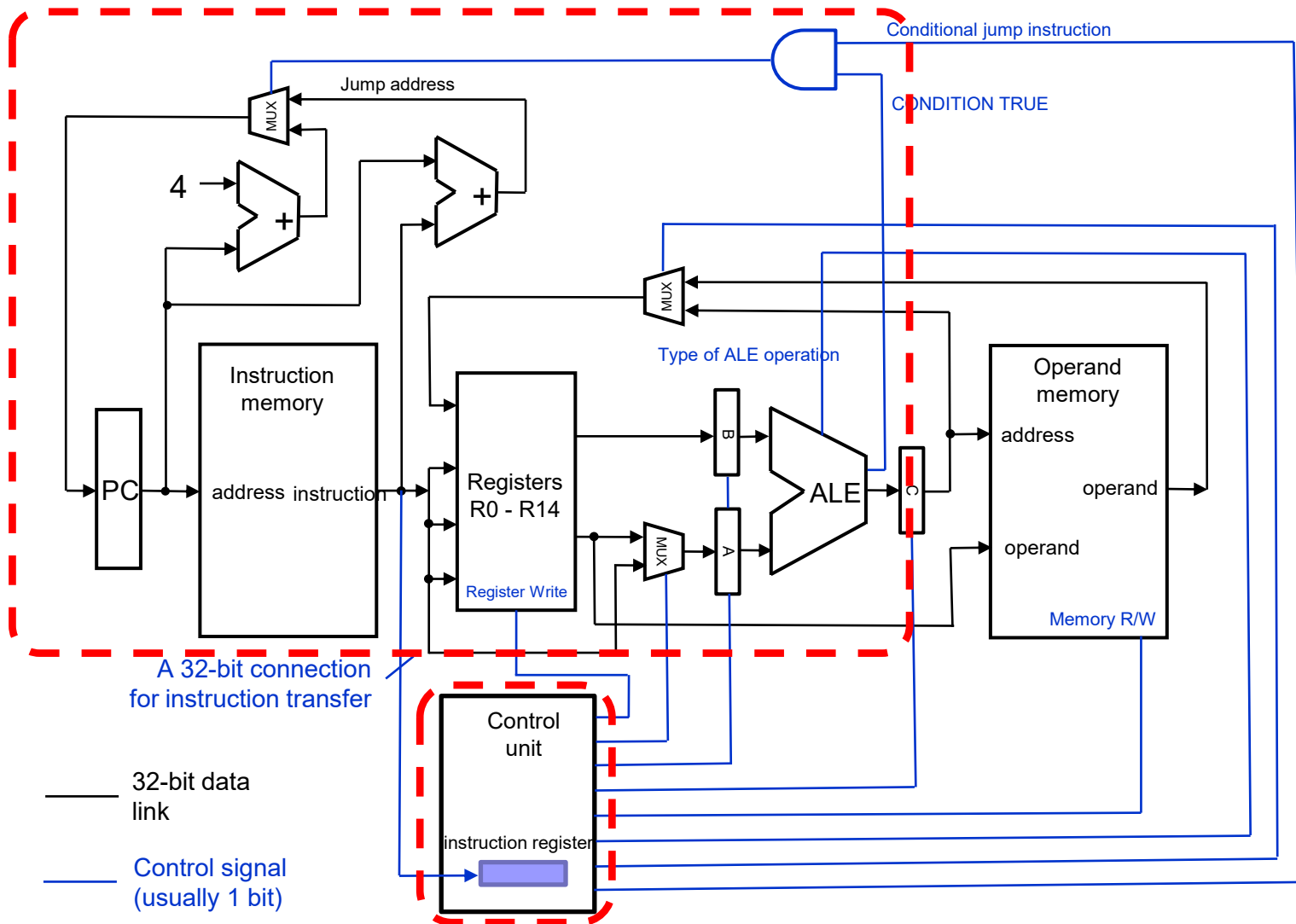






# CPU: datapath, control unit, and control signals

## Execution of branch instructions



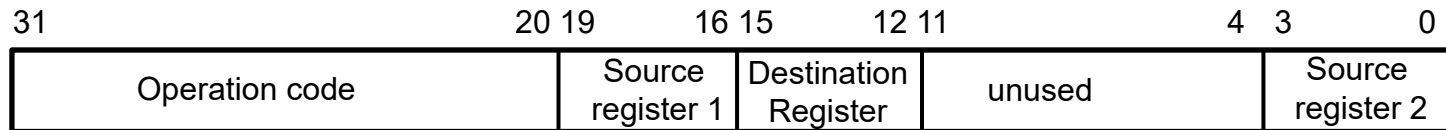


## 6.4 Execution of instructions

An example of execution of a typical instruction for ALU operation:

■ **ADD R10, R1, R3**                      @ R10 ← R1 + R3

Instruction Format:

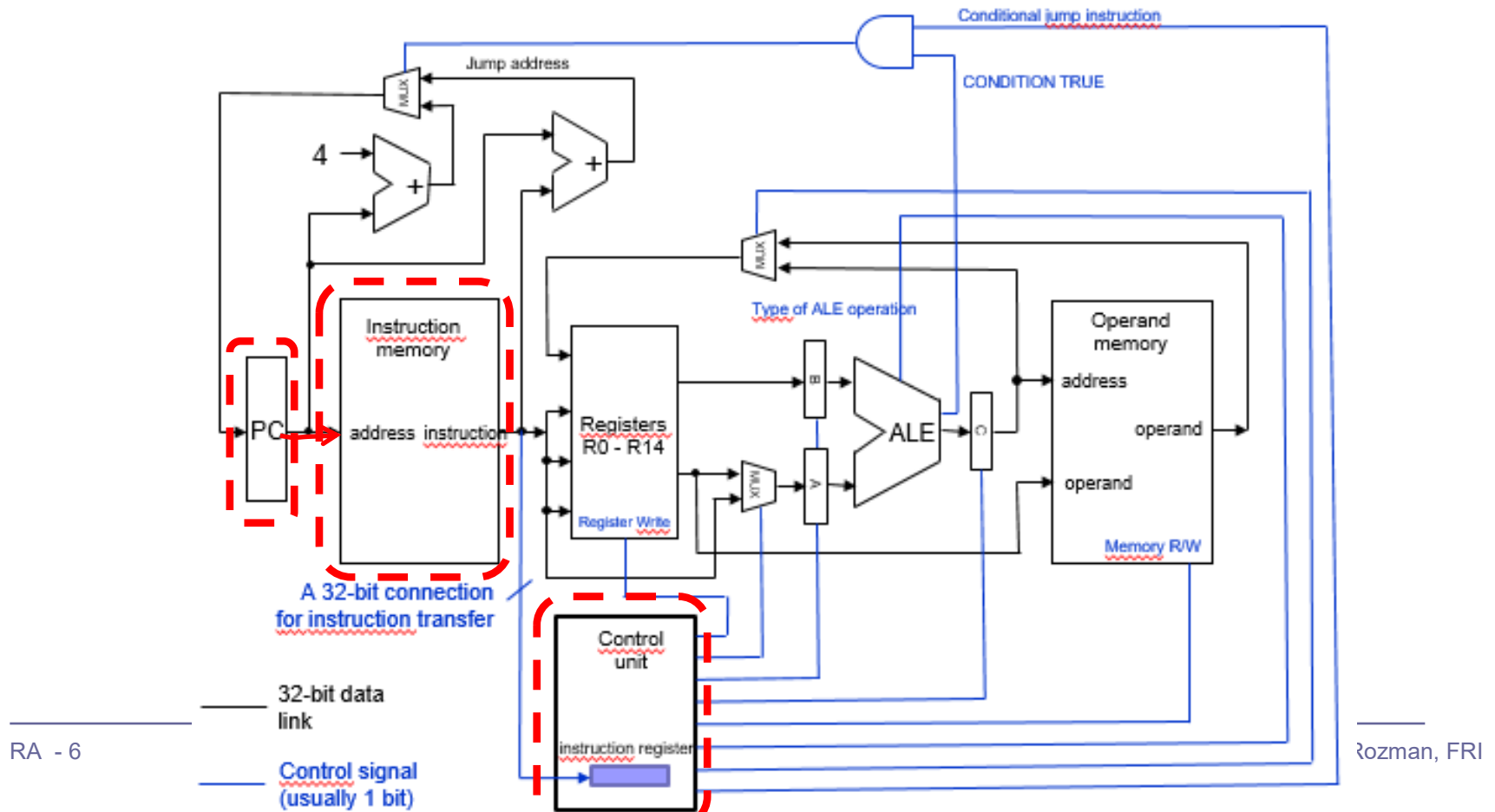
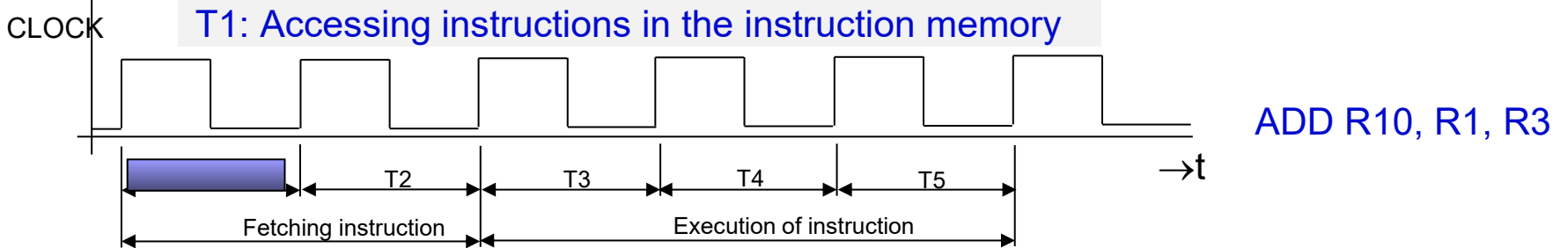


Machine instruction:





# Execution of the instruction ADD: 1. elementary step (T1) = 1 T<sub>clk</sub> (Clock period)



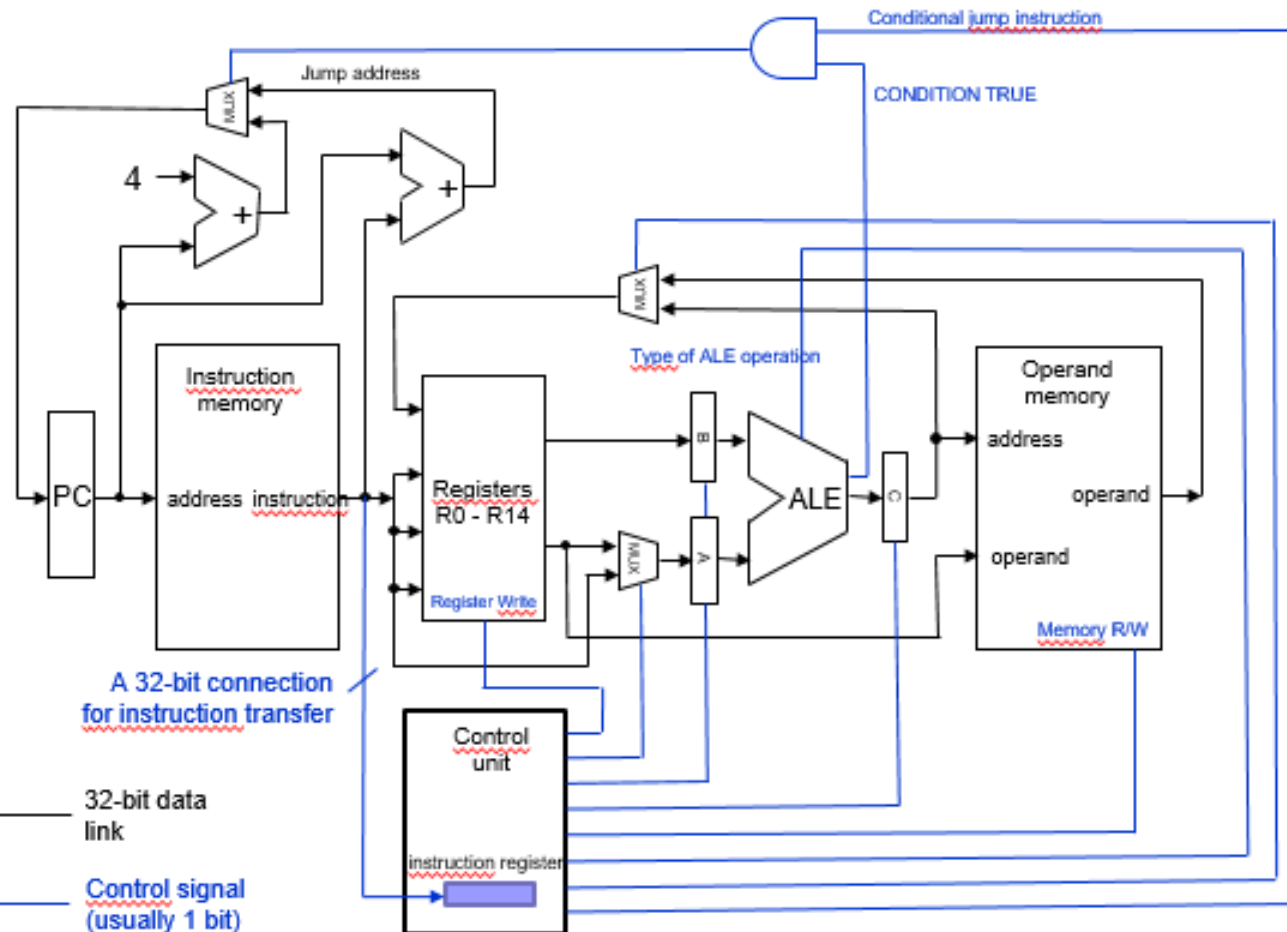
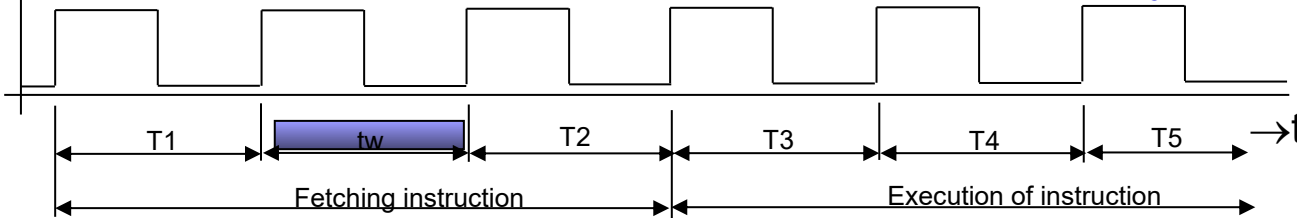


# Execution of the instruction ADD: 2. elementary step ( $T_w$ ) = $n T_{cpe}$ (Clock period)

CLOCK

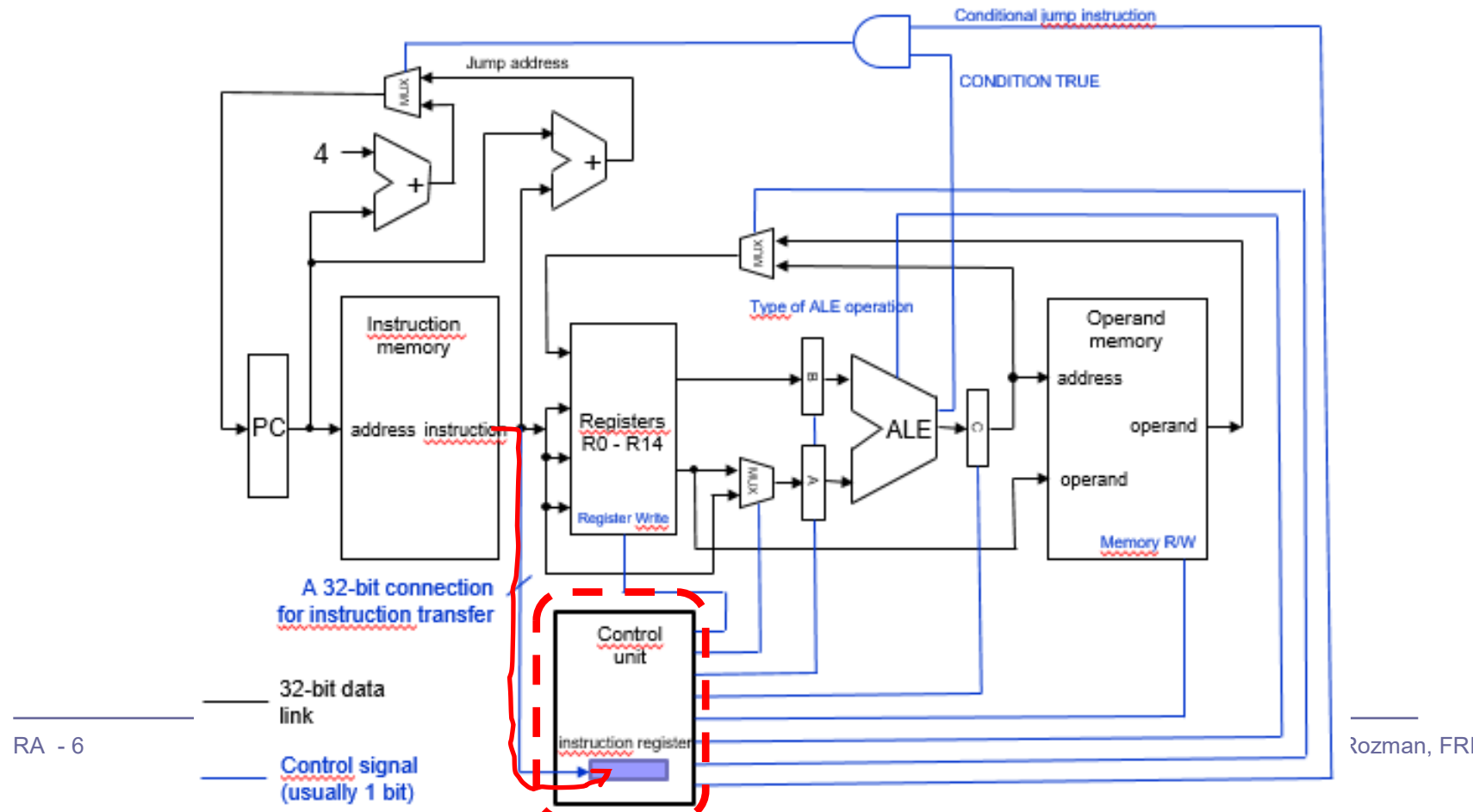
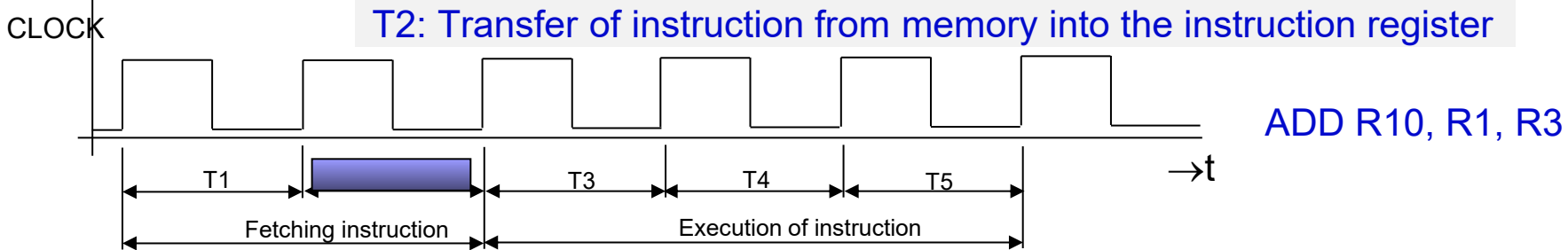
$n T_w$ : On instruction fetch maybe wait clock cycles are needed

ADD R10, R1, R3



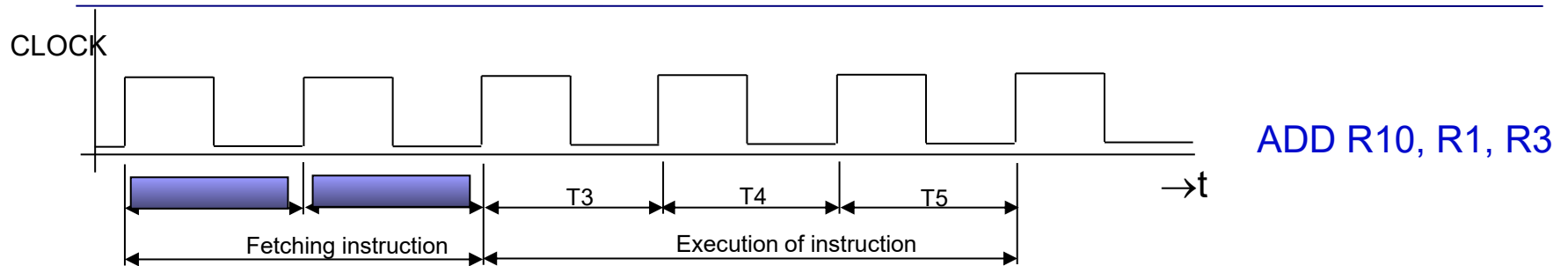


# Execution of the instruction ADD: 2. elementary step (T2) = 1 T<sub>cp</sub> (Clock period)





## Execution of the instruction ADD



- Execution of the instruction `ADD` lasts for example 5 periods ( $CPI_{ALU} = 5$ )
  - **T1**: Read instruction from memory
  - **T2**: Transfer of instruction from memory into the instruction register
  - **T3**: Decode the instruction and access to the operands in registers `R1`, `R3`
  - **T4**: Execution of the operation (addition)
  - **T5**: Saving the result in the register `R10` (writeback)

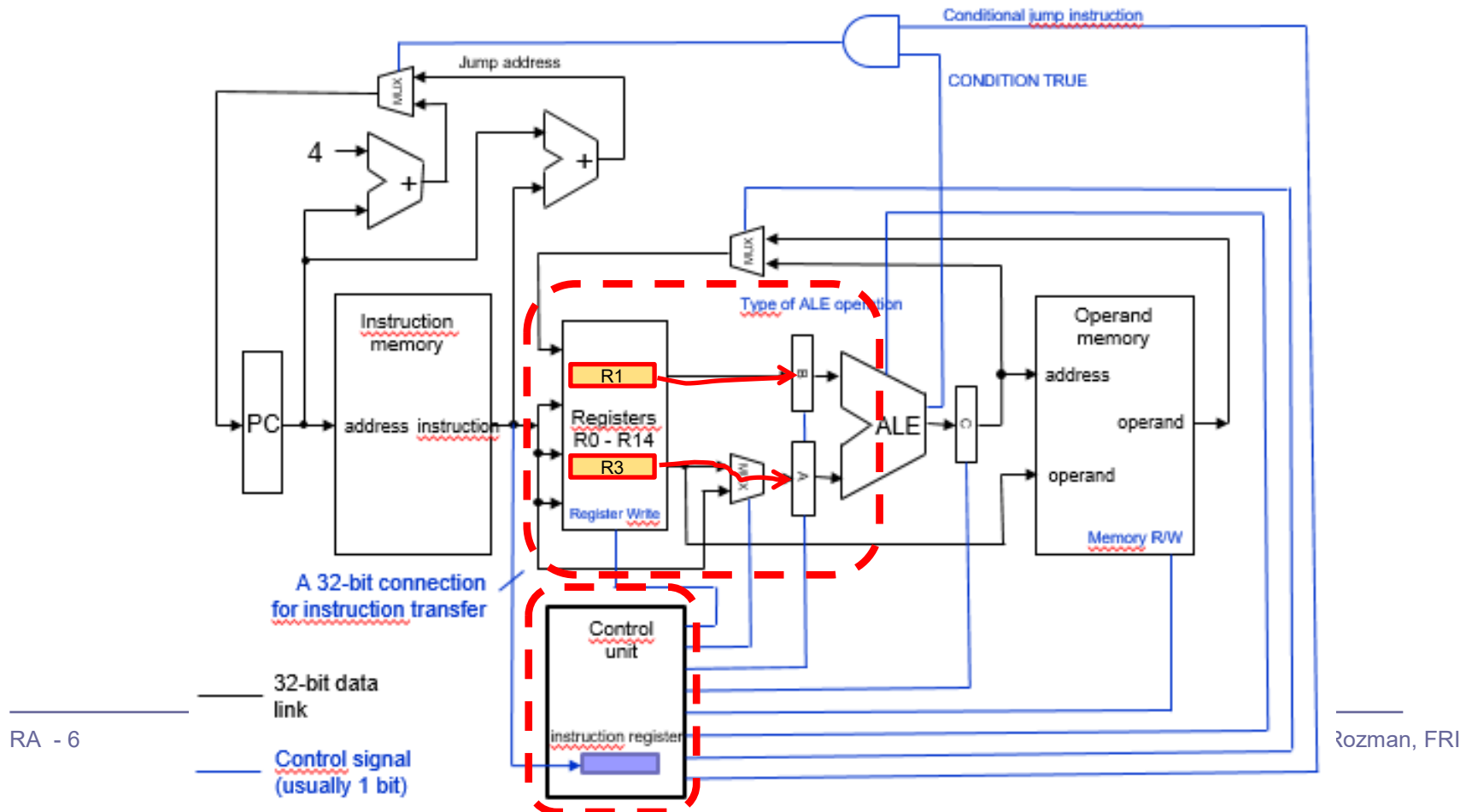
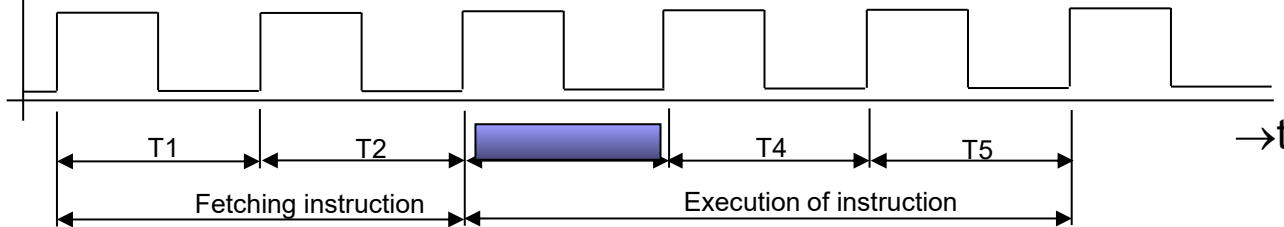


## Execution of the instruction ADD: 3. elementary step (T3) = 1 T<sub>clk</sub> (Clock period)

CLOCK

T3: Decode the instruction and access operands in reg. R1, R3

ADD R10, R1, R3



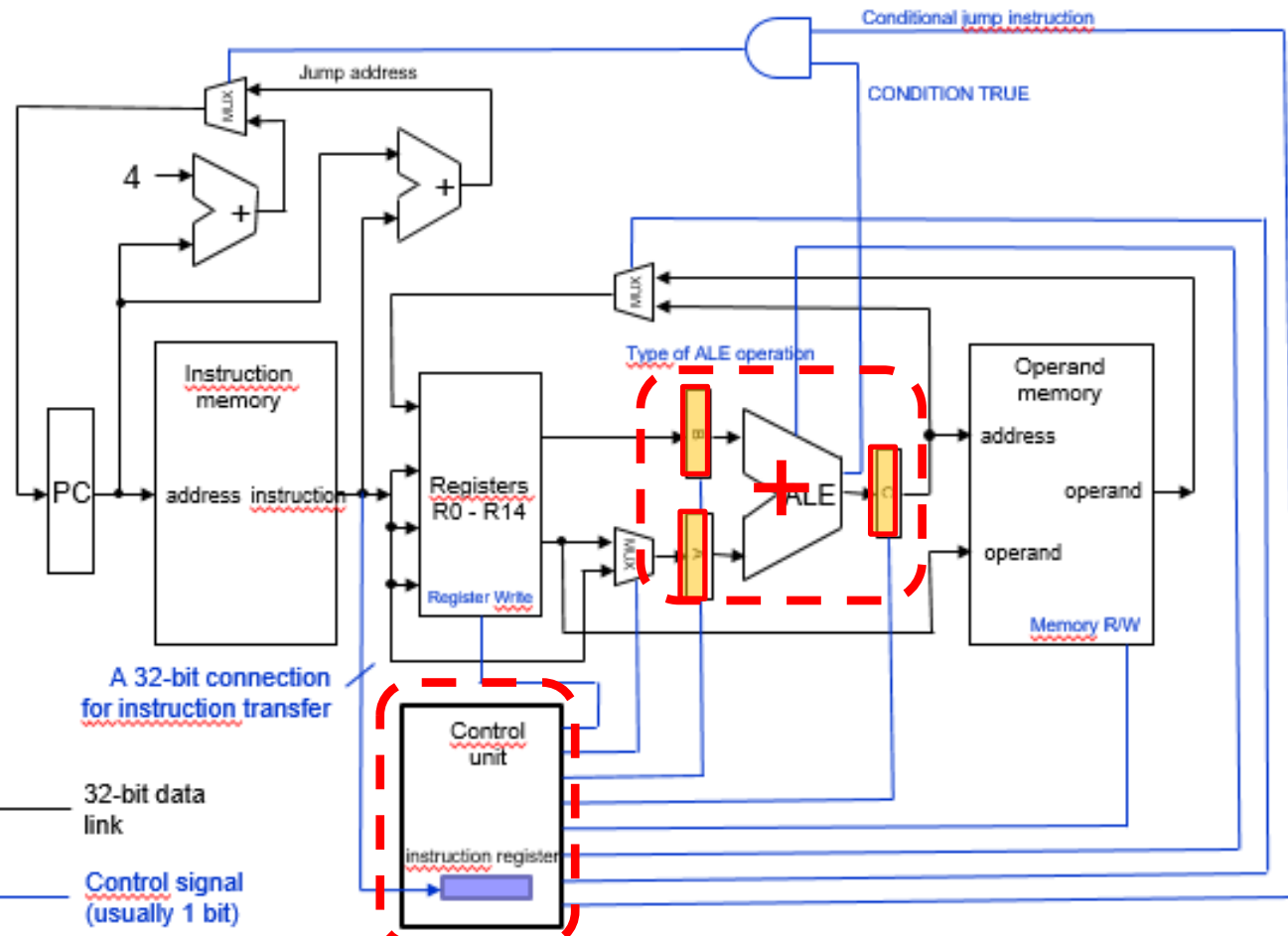
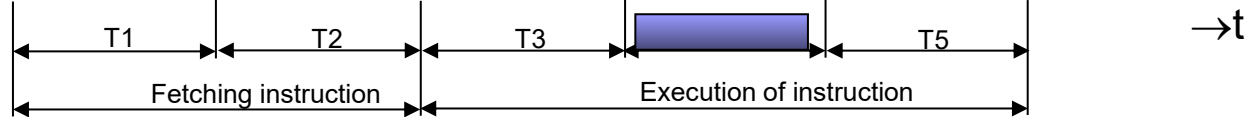


# Execution of the instruction ADD: 4. elementary step (T4) = 1 T<sub>cp</sub> (Clock period)

CLOCK

T4: Execution of the operation (addition)

ADD R10, R1, R3

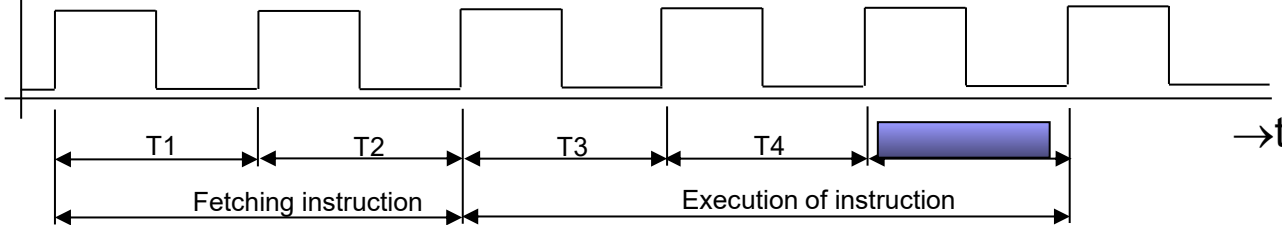




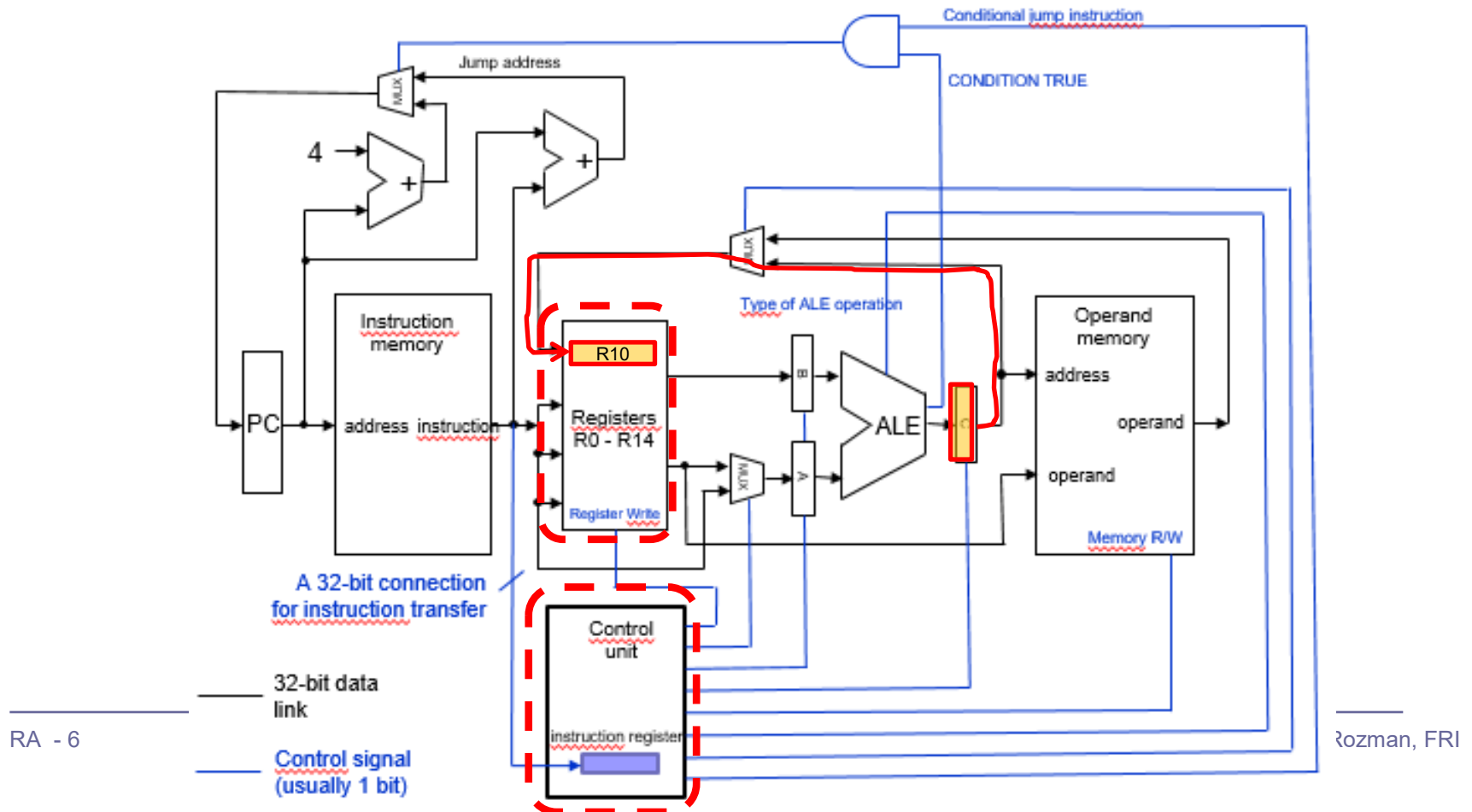


# Execution of the instruction ADD: 5. elementary step (T5) = 1 T<sub>cp</sub> (Clock period)

CLOCK

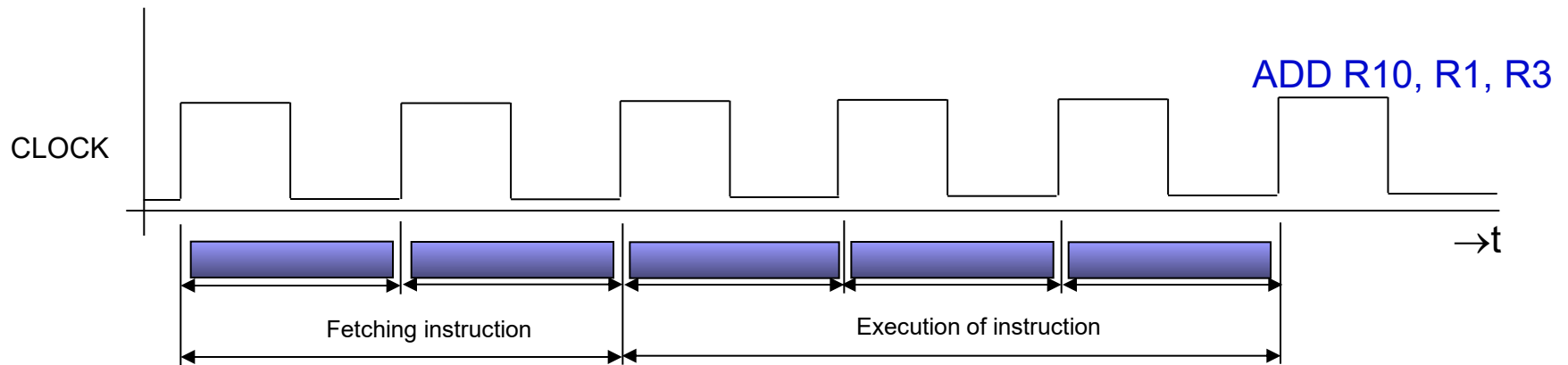


ADD R10, R1, R3





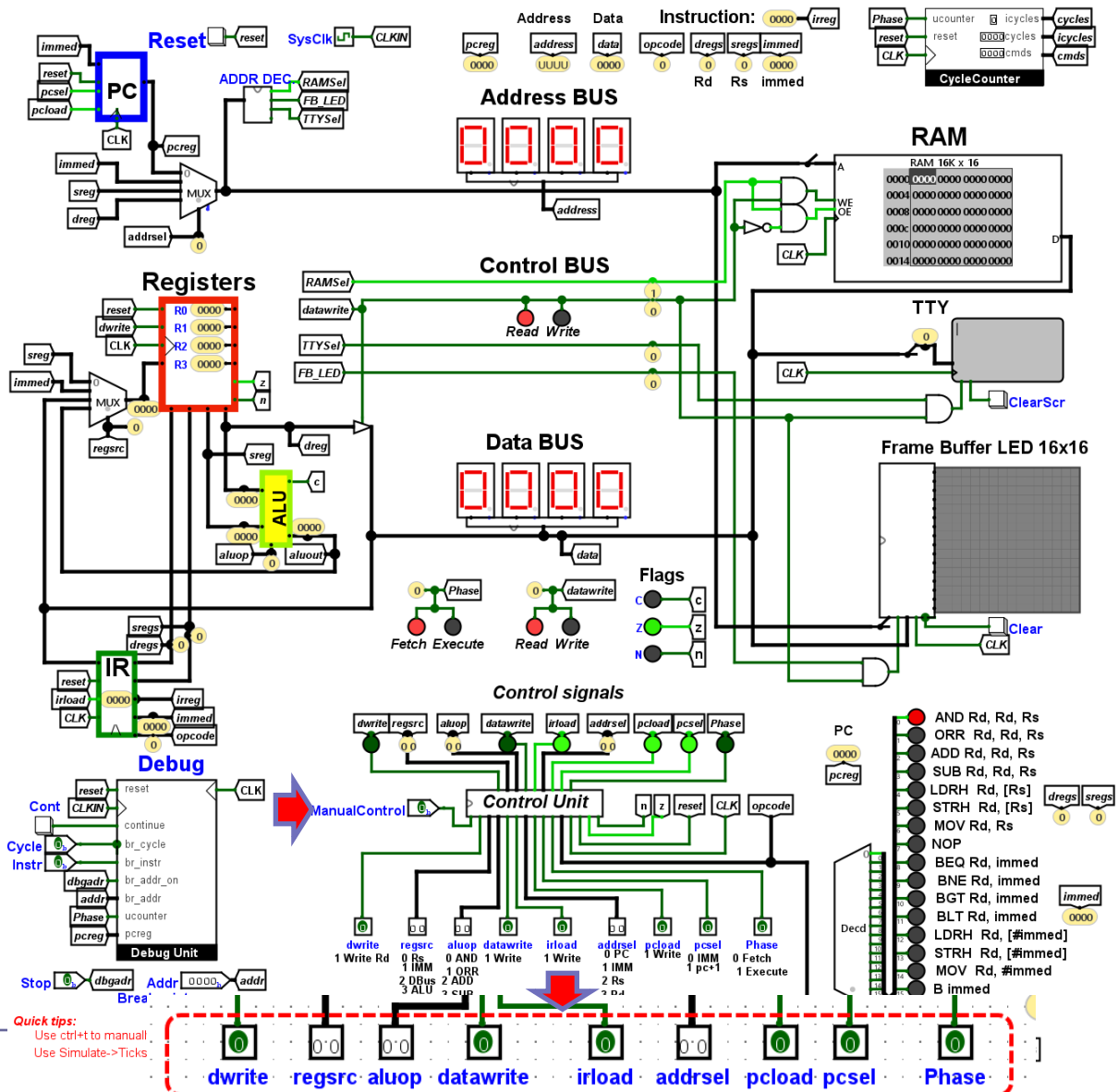
## Execution of the instruction ADD: Summary



- Execution of the instruction ADD lasts for example 5 periods ( $CPI_{ALU} = 5$ )
  - T1: Read instruction from memory
  - T2: Transfer of instruction from memory into the instruction register
  - T3: Decode the instruction and access to the operands in registers R1, R3
  - T4: Execution of the operation (addition)
  - T5: Saving the result in the register R10 (writeback)

# CPU – instr. execution: case Mini MiMo CPU

Mini MiMo - Hardwired Simple CPU Model v0.6 EVO



**Quick tips:**  
 Use ctrl+ to manual.  
 Use Simulate->Ticks.

# case Mini MiMo CPU: Sum of two numbers

16 bitni ukazi - format:

op1	op2	Rd	Rs	immediate
2b	2b	2b	2b	8b

## Program

Naslov	Ozna ka	Ukaz v zbirniku	Strojni ukaz
0x0000	main:	MOV R0, #0x20	e020
0x0001		LDRH R1, [R0]	4400
0x0002		MOV R0, #0x21	e021
0x0003		LDRH R2, [R0]	4800
0x0004		ADD R2, R2, R1	2900
0x0005		MOV R0, #0x22	e022
0x0006		STRH R2, [R0]	5800
0x0007	inf:	B inf	f007

## Assembler in Excel

	A	B	C	D	E	I
	Address	Instruction	Rd	Rs	Immed	Machine instr.
1	0	MOV Rd, #immed	R0	R0	32	E020
2	1	LDRH Rd, [Rs]	R1	R0		4400
3	2	MOV Rd, #immed	R0	R0	33	E021
4	3	LDRH Rd, [Rs]	R2	R0		4800
5	4	ADD Rd, Rd, Rs	R2	R1		2900
6	5	MOV Rd, #immed	R0	R0	34	E022
7	6	STRH Rd, [Rs]	R2	R0		5800
8	7	B immed	R0	R0	7	F007



```

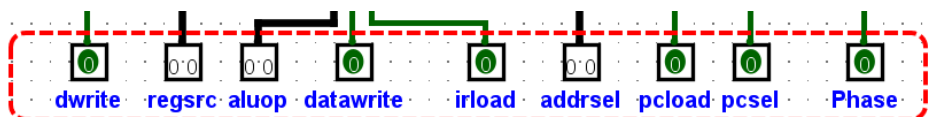
minimimo_vsota.ram  test17-tournament.s  minimimo_sestej.ram
1  v3.0 hex words addressed
2  0000: e020 4400 e021 4800 2900 e022 5800 f007
3  0010: 0000 0000 0000 0000 0000 0000 0000 0000
4  0020: 0010 0040 0000 0000 0000 0000 0000 0000
5
    
```



Mini MiMo CPU  
RAM memory

## Control Unit Control signals for execution of :

op1	op2	ARM9 zapis	pc sel	pc load	ir load	rw	dwrite	addr sel	reg sel	d reg	s reg	aluop
xx	xx	FETCH - vsi ukazi	1(pc+1)	1	1	0	0	0(pc)	x	x	x	x
00	10	ADD Rd, Rd, Rs	x	0	0	0	1	x	3(ALU)	Rd	Rs	op2=0b10
01	00	LDRH Rd, [Rs]	x	0	0	0	1	2(Rs)	2(Dbuss)	Rd	Rs	x
01	01	STRH Rd, [Rs]	x	0	0	1	0	2(Rs)	x	Rd	Rs	x
11	10	MOV Rd, #immed	x	0	0	0	1	x	1(IM)	Rd	x	x
11	11	B immed	0(IM)	1	0	0	0	x	x	x	x	x

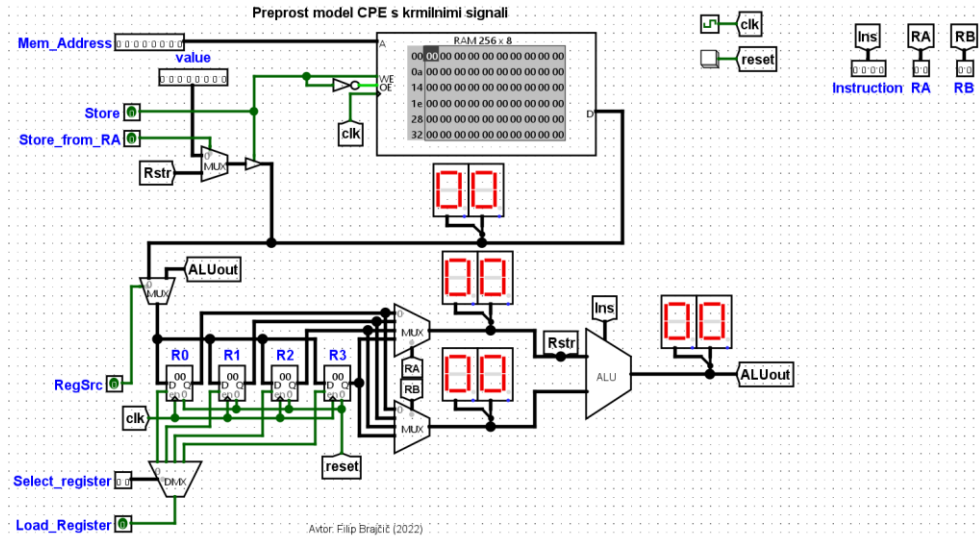
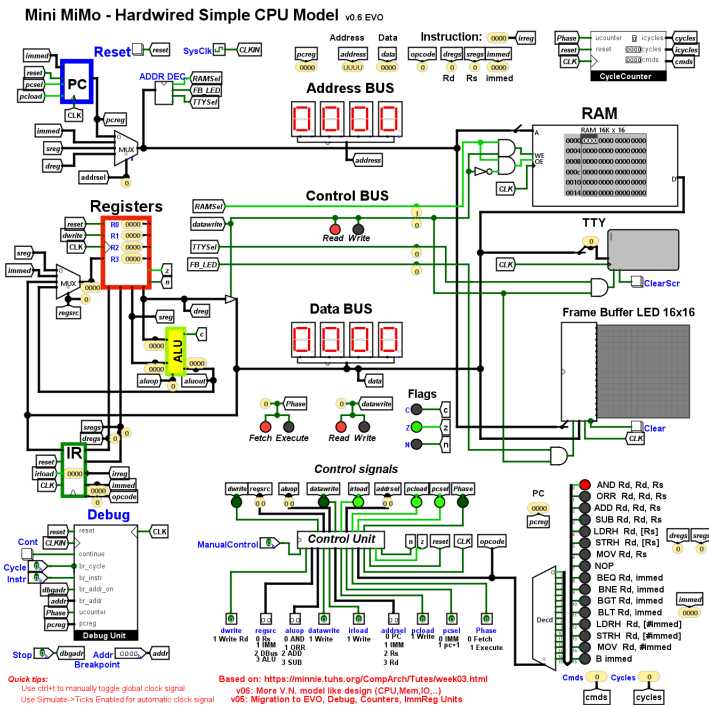




# Challenge (HW1 or optional extension to HW1)

Program/edit Mini  
MiMo model

Challenge: Dare to create your  
own CPU ?



Example of challenge solution  
from 22/23

[https://github.com/LAPSyLAB/RALab-STM32H7/tree/main/MiniMiMo\\_HW\\_CPE\\_Model](https://github.com/LAPSyLAB/RALab-STM32H7/tree/main/MiniMiMo_HW_CPE_Model)

[https://github.com/LAPSyLAB/RALab-STM32H7/tree/main/LogisimEVO\\_vezja/Prispevki](https://github.com/LAPSyLAB/RALab-STM32H7/tree/main/LogisimEVO_vezja/Prispevki)



## 6.5 Parallel execution of instructions

- Typical CPU arch. – execution of machine instructions takes at least 3 or 4 clock periods, usually even more.
- The average number of instructions executed by the CPU in one second (*IPS - Instructions Per Second*):

$$IPS = \frac{f_{CPE}}{CPI}$$

IPS is a very large number, so we divide it by  $10^6$  and get MIPS

$$MIPS = \frac{f_{CPE}}{CPI \cdot 10^6}$$

MIPS = Million Instructions Per Second

$f_{CPE}$  = Frequency of the CPU clock

CPI = Cycles Per Instruction  
(average number of clock periods  
for the execution of one instruction)



- MIPS - the number of instructions executed by the CPU in one second, can be increased in two ways: to increase  $f_{CPE}$  and/or reduce the CPI:

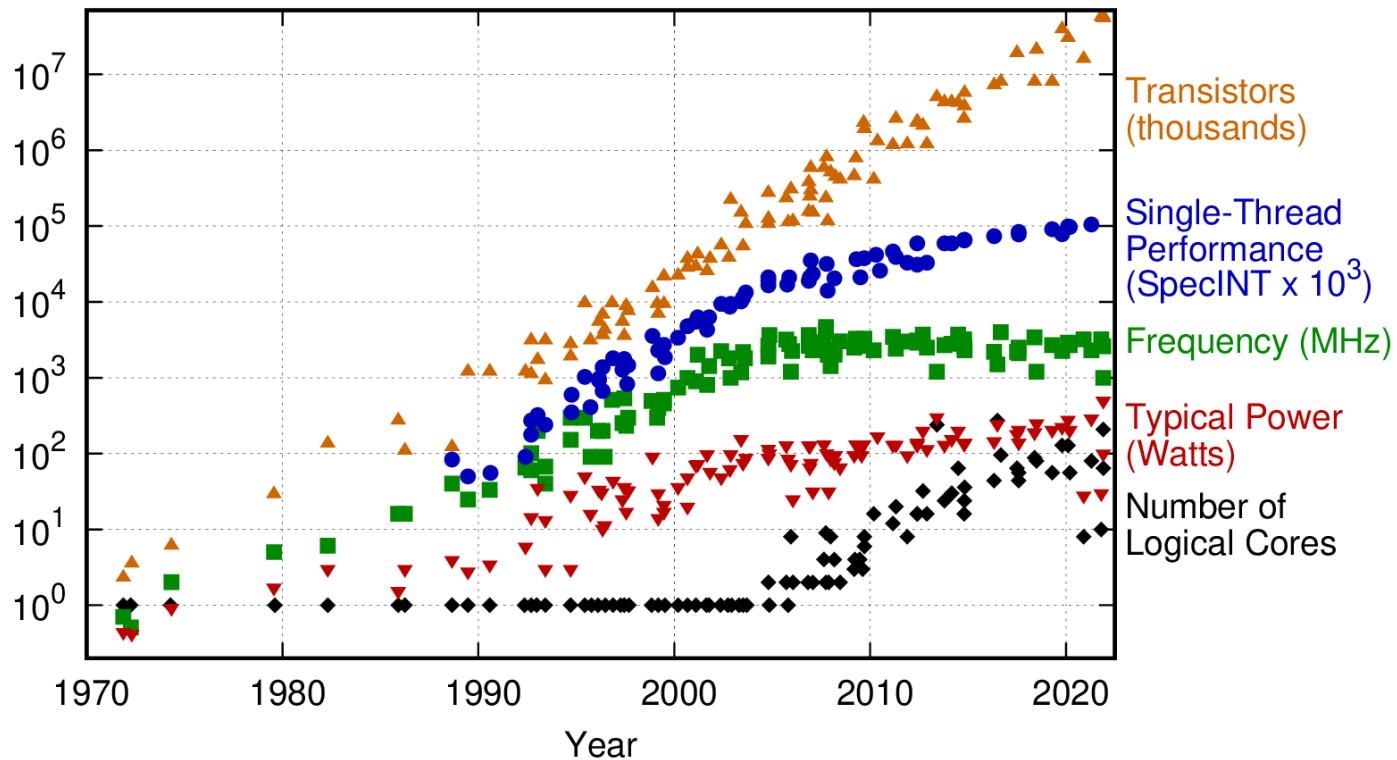
$$\uparrow MIPS = \frac{\uparrow f_{CPE}}{\downarrow CPI \cdot 10^6}$$

- Using faster electronic elements (increase  $f_{CPE}$  = more periods in one second)
- With the use of a larger number of elements we can reduce the CPI (less clock cycles per instruction) where more instructions are executed in one clock cycle
- Use of faster electronic components does not allow larger increase in speed; it also causes other problems.



## General trends in Computing Evolution

50 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2021 by K. Rupp

Vir: <https://raw.githubusercontent.com/karlrupp/microprocessor-trend-data/master/50yrs/50-years-processor-trend.png>





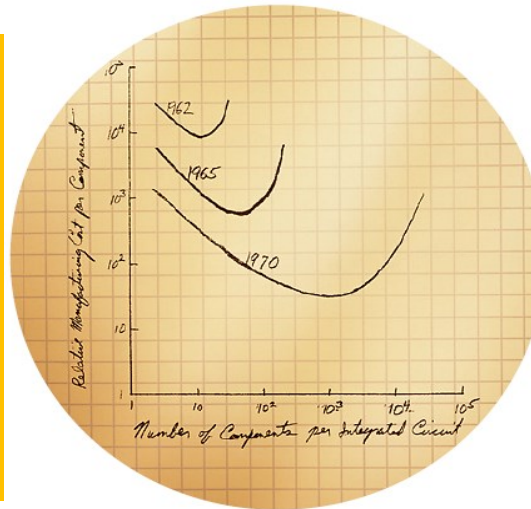
## Increasing the number of transistors - Moore's Law

- Electronic Magazine has published an article in 1965 by Gordon E. Moore in which he predicted that the number of transistors that producers are able to produce on a chip doubles every year.
- In 1975, the prediction was adjusted to the period of two years (number of transistors doubling every two years).
- As it was then intended as experimental rule should apply the next few years, it is still valid today and is known as Moore's Law.



# Moore's Law

relative Manufacturing cost per Component



In 1965, Gordon Moore predicted the pace of silicon technology. Decades later, Moore's Law remains true, driven largely by Intel's unparalleled silicon expertise.

According to Moore's Law, the number of transistors on a chip roughly doubles every two years. As a result the scale gets smaller and smaller. For decades, Intel has met this formidable challenge through investments in technology and manufacturing resulting in the unparalleled silicon expertise that has made Moore's Law a reality.

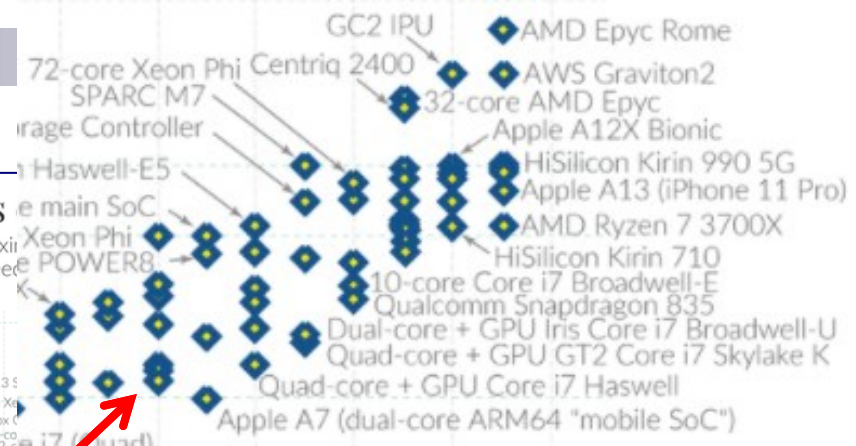
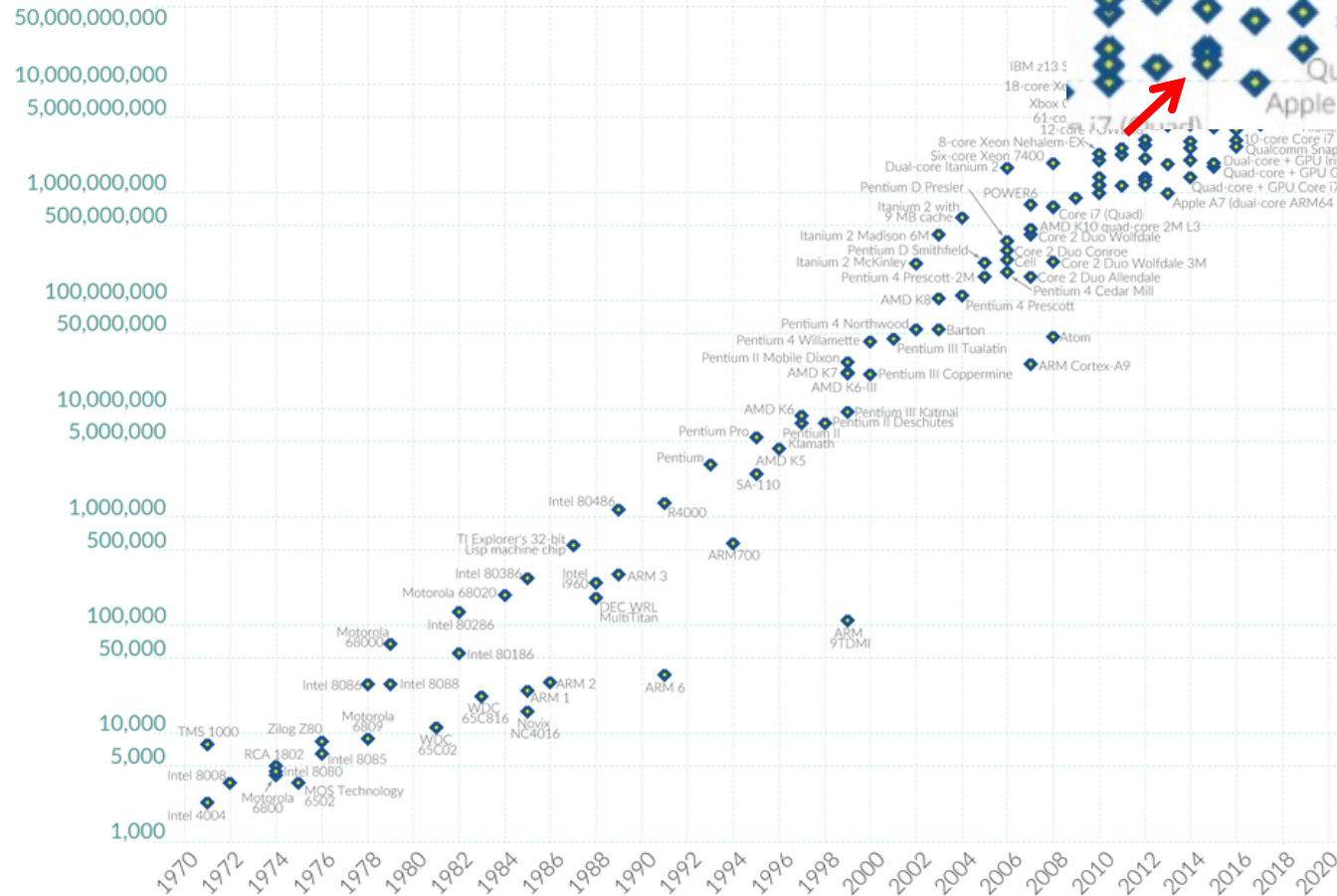


- Gordon E. Moore is now honorary president of Intel, in 1968 he was co-founder and executive vice president of Intel.
- With the same technology in the period of 20 years some time ago, the maximum speed of logic elements increased by about 10 times.
- At the same time, the maximum number of elements on a single chip increased by about 500 to as much as 5000-times in the memory chips.

# Moore's Law: The number of transistors on microchips doubles

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed.

## Transistor count



Data source: Wikipedia ([wikipedia.org/wiki/Transistor\\_count](https://wikipedia.org/wiki/Transistor_count))

OurWorldinData.org – Research and data to make progress against the world's largest problems.

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.



## Moore's law – Transistor count through time

Processor	Transistor count	Year	Designer	Process (nm)	Area (mm <sup>2</sup> )	Transistor density (tr./mm <sup>2</sup> )
MP944 (20-bit, 6-chip, 28 chips total)	74,442 (5,360 excl. ROM & RAM) <sup>[14][15]</sup>	1970 <sup>[12][a]</sup>	Garrett AiResearch	?	?	?
Intel 4004 (4-bit, 16-pin)	2,250	1971	Intel	10,000 nm	12 mm <sup>2</sup>	188
TMX 1795 (8-bit, 24-pin)	3,078 <sup>[16]</sup>	1971	Texas Instruments	?	30.64 mm <sup>2</sup>	100.5
Intel 8008 (8-bit, 18-pin)	3,500	1972	Intel	10,000 nm	14 mm <sup>2</sup>	250
NEC $\mu$ COM-4 (4-bit, 42-pin)	2,500 <sup>[17][18]</sup>	1973	NEC	7,500 nm <sup>[19]</sup>	?	?
Toshiba TLCS-12 (12-bit)	11,000+ <sup>[20]</sup>	1973	Toshiba	6,000 nm	32.45 mm <sup>2</sup>	340+
Intel 4040 (4-bit, 16-pin)	3,000	1974	Intel	10,000 nm	12 mm <sup>2</sup>	250
Motorola 6800 (8-bit, 40-pin)	4,100	1974	Motorola	6,000 nm	16 mm <sup>2</sup>	256
Intel 8080 (8-bit, 40-pin)	6,000	1974	Intel	6,000 nm	20 mm <sup>2</sup>	300
TMS 1000 (4-bit, 28-pin)	8,000 <sup>[b]</sup>	1974 <sup>[21]</sup>	Texas Instruments	8,000 nm	11 mm <sup>2</sup>	730
MOS Technology 6502 (8-bit, 40-pin)	4,528 <sup>[c][22]</sup>	1975	MOS Technology	8,000 nm	21 mm <sup>2</sup>	216
Intersil IM6100 (12-bit, 40-pin; clone of PDP-8)	4,000	1975	Intersil	?	?	?
CDP 1801 (8-bit, 2-chip, 40-pin)	5,000	1975	RCA	?	?	?
RCA 1802 (8-bit, 40-pin)	5,000	1976	RCA	5,000 nm	27 mm <sup>2</sup>	185
Zilog Z80 (8-bit, 4-bit ALU, 40-pin)	8,500 <sup>[d]</sup>	1976	Zilog	4,000 nm	18 mm <sup>2</sup>	470
Intel 8085 (8-bit, 40-pin)	6,500	1976	Intel	3,000 nm	20 mm <sup>2</sup>	325

■ ■ ■

Apple A17	19,000,000,000 <sup>[187]</sup>	2023	Apple	3 nm	103.8 mm <sup>2</sup>	183,044,315
Sapphire Rapids quad-chip module (up to 60 cores and 112.5 MB of cache) <sup>[188]</sup>	44,000,000,000–48,000,000,000 <sup>[189]</sup>	2023	Intel	10 nm ESF (Intel 7)	1,600 mm <sup>2</sup>	27,500,000–30,000,000
Apple M2 Pro (12-core 64-bit ARM64 SoC, SIMD, caches)	40,000,000,000 <sup>[190]</sup>	2023	Apple	5 nm	?	?
Apple M2 Max (12-core 64-bit ARM64 SoC, SIMD, caches)	67,000,000,000 <sup>[190]</sup>	2023	Apple	5 nm	?	?
Apple M2 Ultra (two M2 Max dies)	134,000,000,000 <sup>[6]</sup>	2023	Apple	5 nm	?	?
AMD Epyc Bergamo (4th gen/97X4 series) 9-chip module (up to 128 cores and 256 MB (L3) + 128 MB (L2) cache)	82,000,000,000 <sup>[191]</sup>	2023	AMD	5 nm (CCD) 6 nm (IOD)	?	?
AMD Instinct MI300A (multi-chip module, 24 cores, 128 GB GPU memory + 256 MB (LLC/L3) cache)	146,000,000,000 <sup>[192][193]</sup>	2023	AMD	5 nm (CCD, GCD) 6 nm (IOD)	1,017 mm <sup>2</sup>	144,000,000
Processor	Transistor count	Year	Designer	Process (nm)	Area (mm <sup>2</sup> )	Transistor density (tr./mm <sup>2</sup> )



## Moore's law – Transistor count regarding the type of device

Year	Component	Name	Number of MOSFETs (in trillions)	Remarks
2022	Flash memory	Micron's V-NAND module	5.3	stacked package of sixteen 232-layer 3D NAND dies
2020	any processor	Wafer Scale Engine 2	2.6	wafer-scale design of 84 exposed fields (dies)
2024	GPU	Nvidia B100	0.208	Uses two reticle limit dies, with 104 billion transistors each, joined together and acting as a single large monolithic piece of silicon
2023	microprocessor (commercial)	M2 Ultra	0.134	SoC using two dies joined together with a high-speed bridge
2020	DLP	Colossus Mk2 GC200	0.059	An IPU <sup>[clarification needed]</sup> in contrast to CPU and GPU

DLP... „Deep learning processor“



## How to effectively utilize multiple items?

- Efficient **increase in speed of CPU**:
  - CPU performs parallel **more operations**, which means an increase in the number of needed logic elements.

## Parallelism can be exploited on several levels:

- Parallelism at the level of instructions:
  - Some instructions in the program **can be carried out simultaneously** – in parallel
  - CPU in the form of **pipeline**:
    - Exploitation of **parallelism at the level of instructions**
    - ***An important advantage: the programs stay the same !!!***
    - ***Limited***, so we are looking for other options



- The first higher-level parallelism is called **parallelism at the level of threads.**
  - Multithreading
  - Multi-core processors
  
- **Parallelism at the level of CPU** (MIMD - multiprocessors, multicomputers)
  
- **Data-level Parallelism** (GPU, SIMD, Vector units)



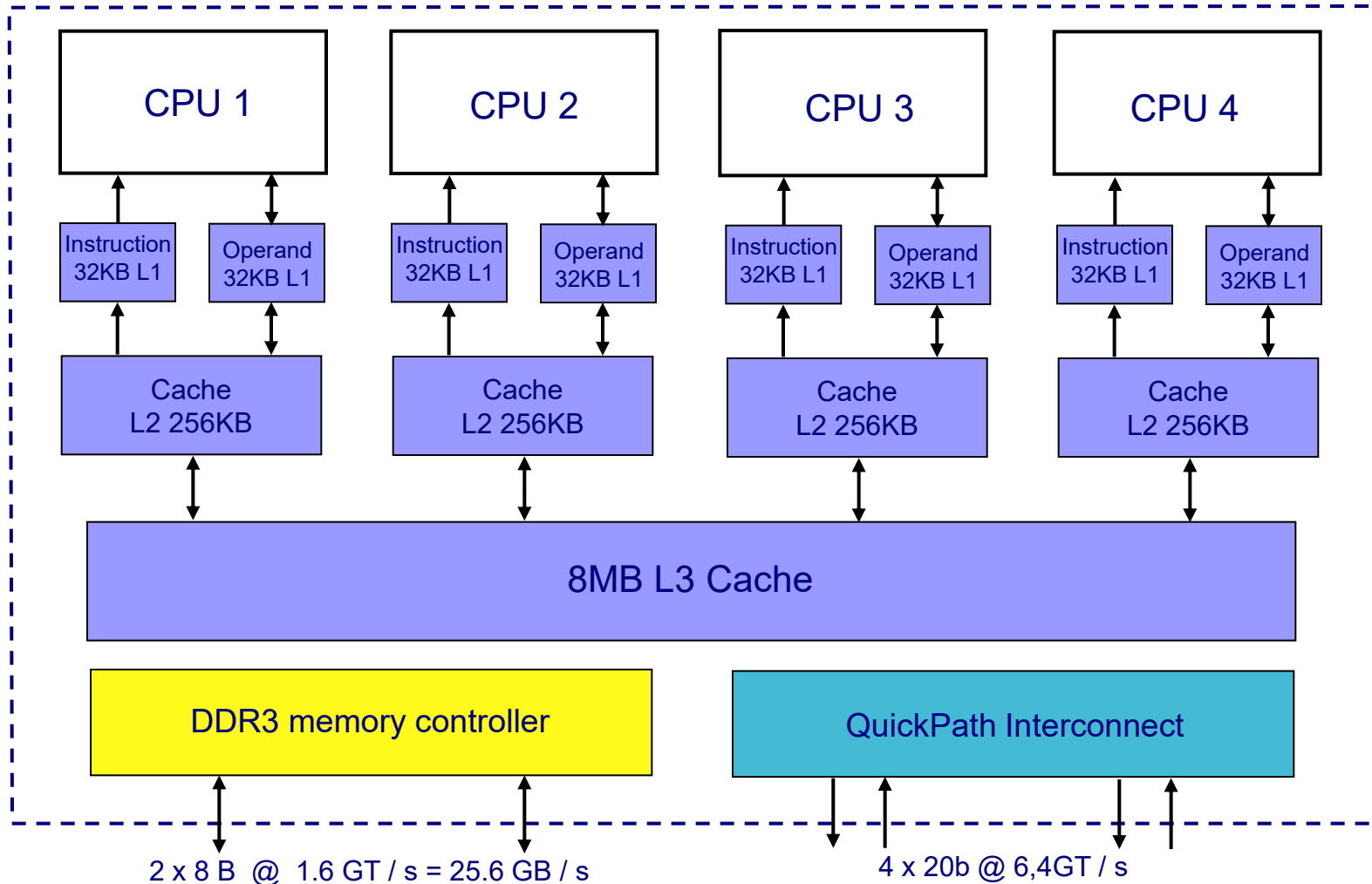


## ■ Intel Core i7 Haswell

- Feature size 22 nm ( $= 22 * 10^{-9}$  m)
- The number of transistors 1.6 billion ( $= 1600000000$ )
- The size of the chip 160 mm<sup>2</sup> (From 10x to 26x mm<sup>2</sup>)
- The clock frequency from 2.0 GHz to 4.4 GHz
- The number of cores (CPU) 4
- graphics processor
- Socket LGA 1150
- TDP (Thermal design Power) from 11.5 W to 84 W
- Price  $\approx$  300-400 \$



## Structure of 4-core processor Intel Core i7 (Haswell)

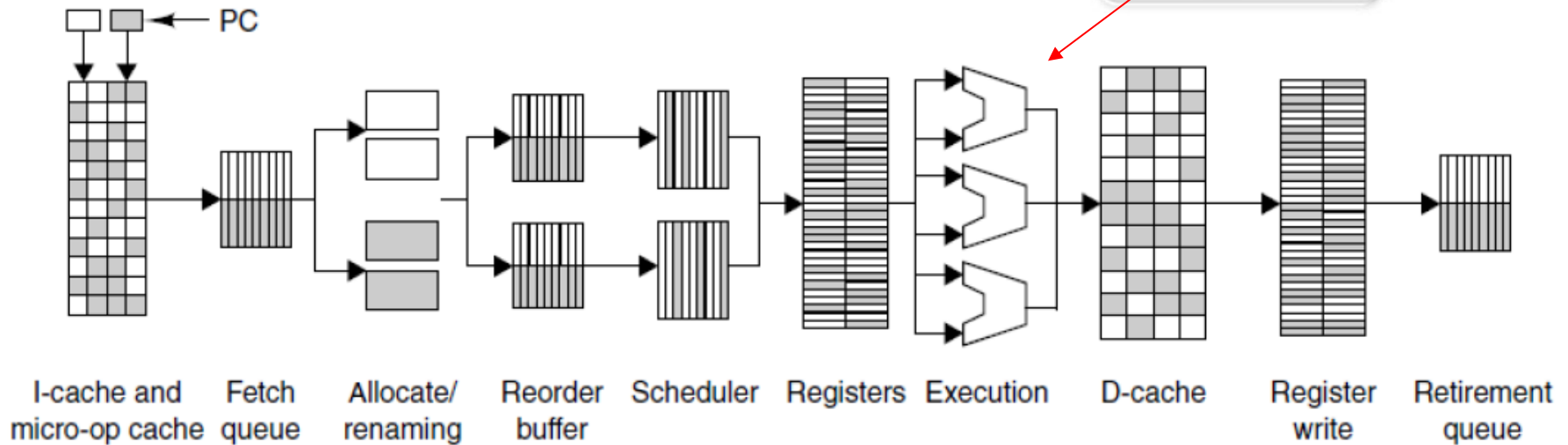
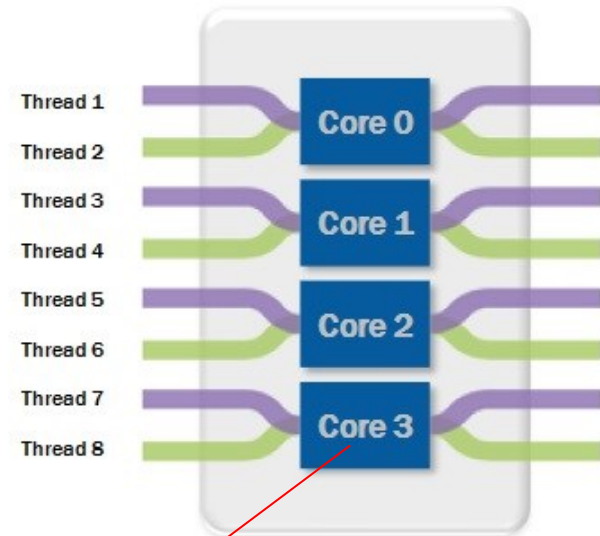




## Simultaneous Multi Threading (SMT)

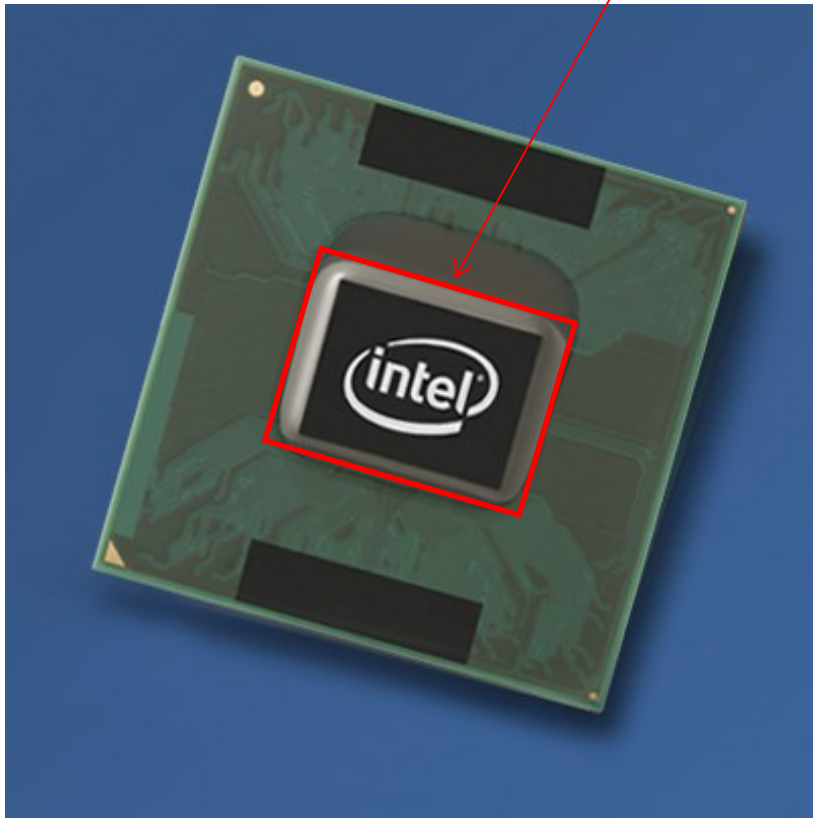
„Hyperthreading“ on Core i7

1 core supports 2 threads  
(two „virtual“ cores)

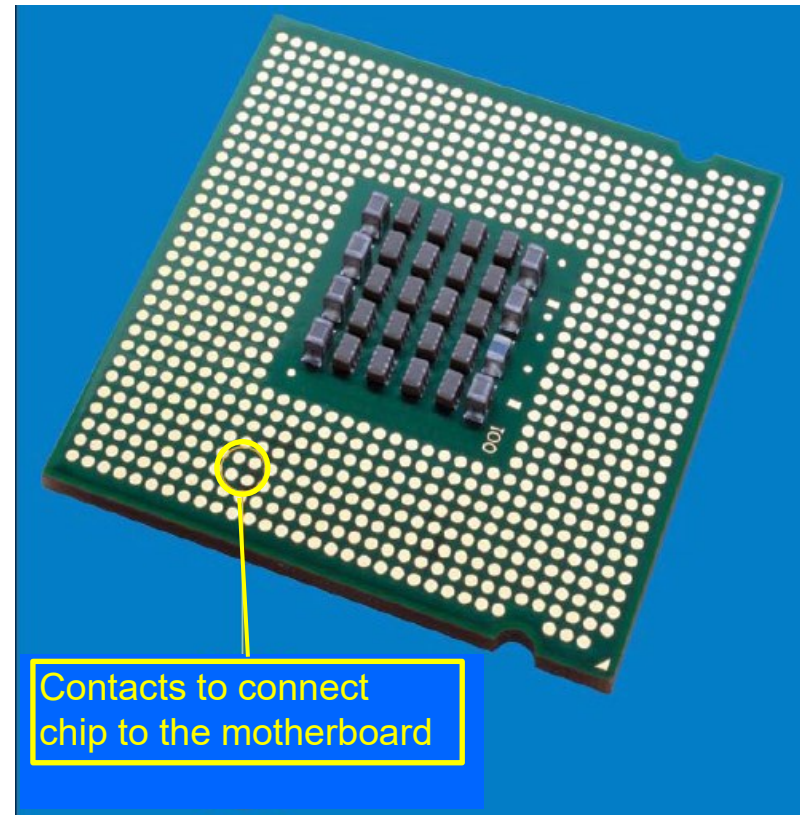




CPU chip on the socket with the contacts (LGA775)

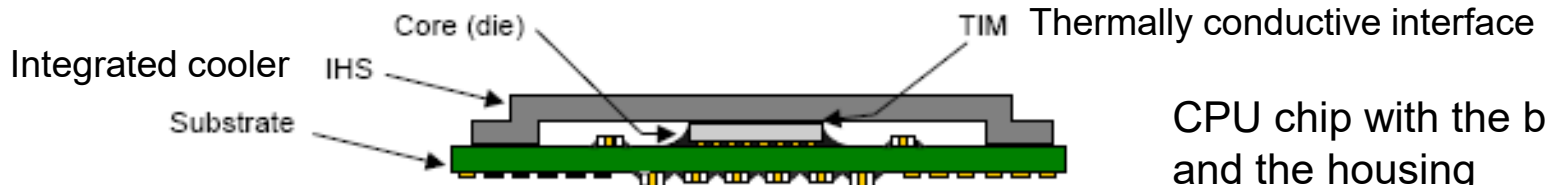


The upper side

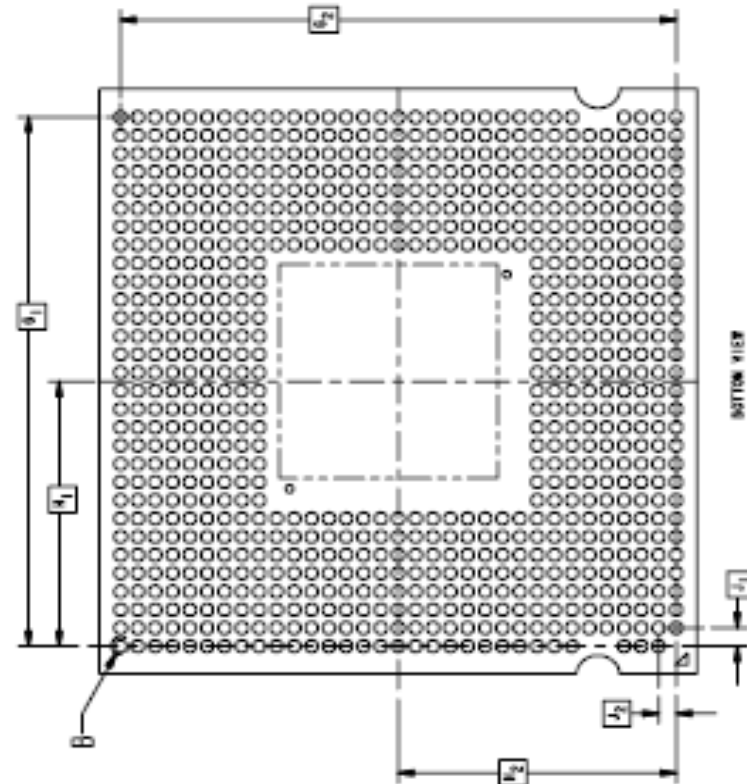


Contacts to connect chip to the motherboard

Lower side with the contacts and the capacitors



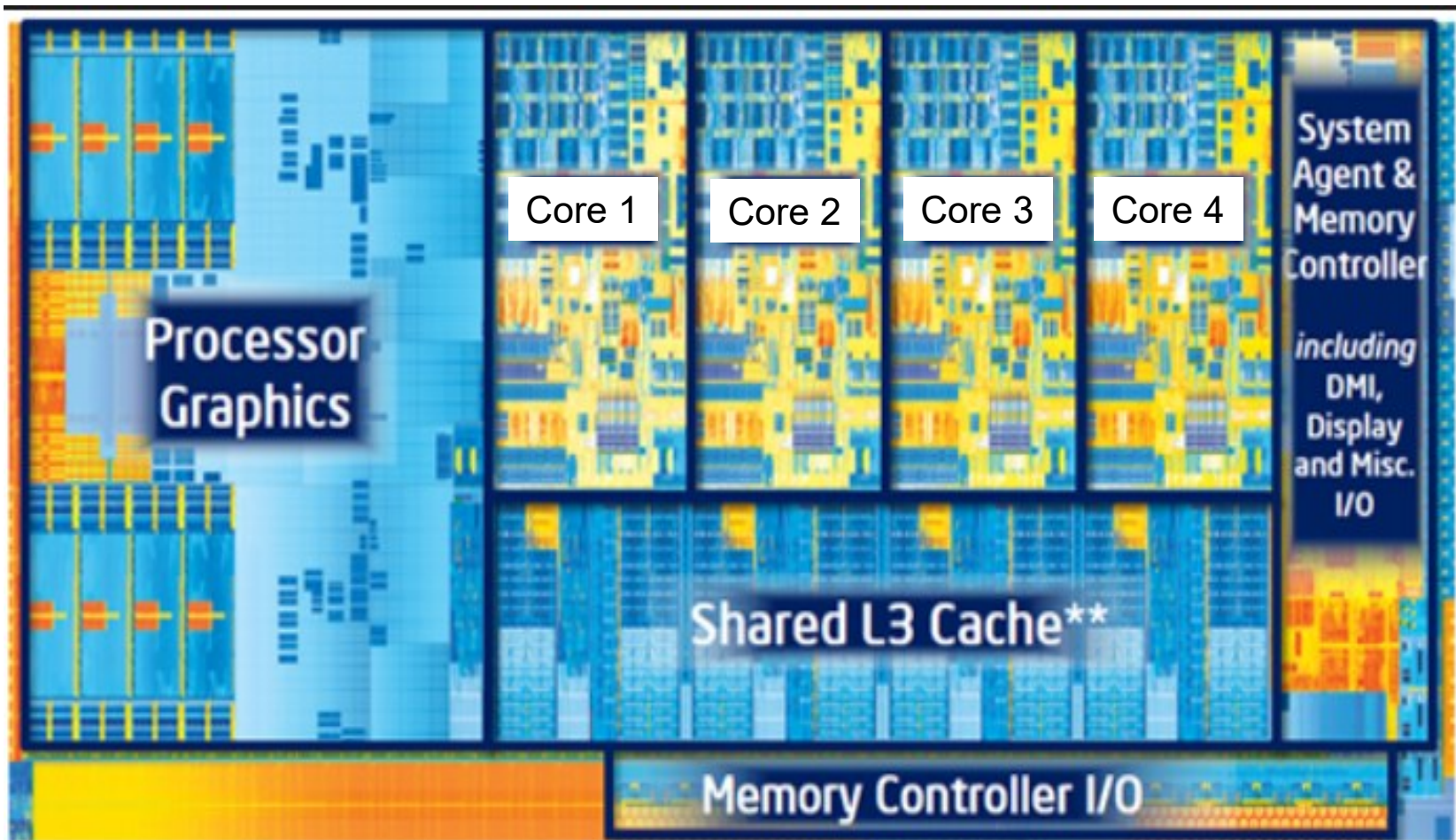
CPU chip with the base and the housing – cross section



Socket LGA775 - view from below



## Intel chip Core i7 (Haswell)

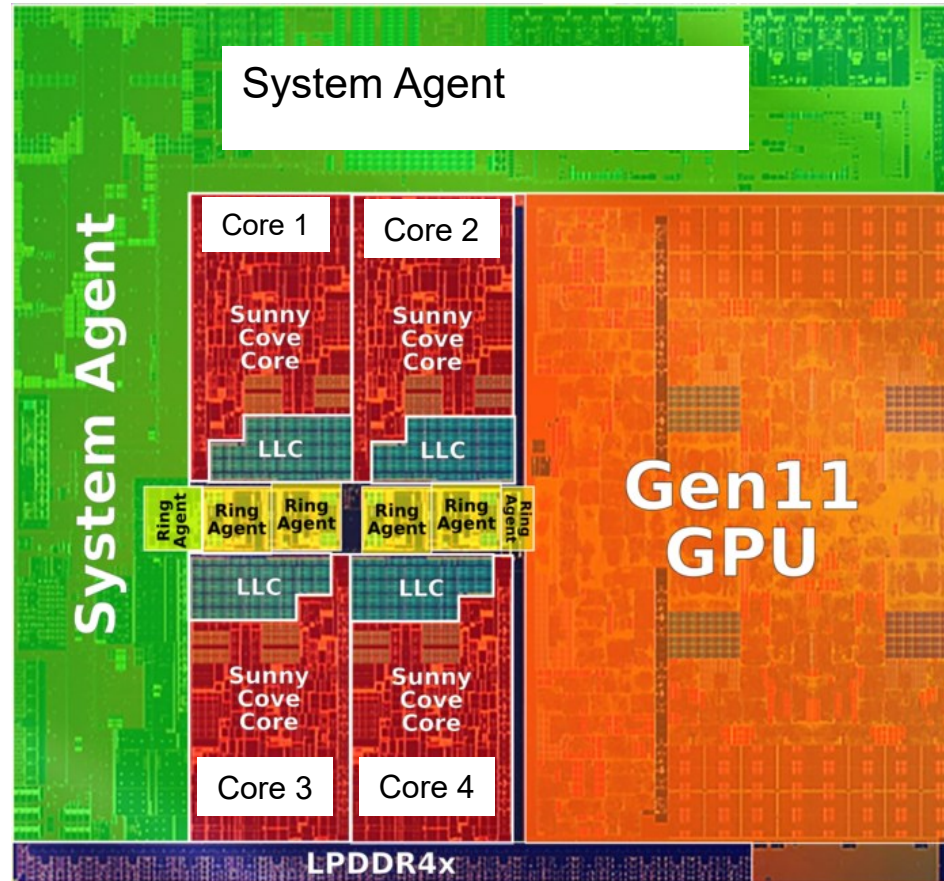




## Example parallelism level instructions, threads and cores Intel 80x86

Intel Core i7

(Ice Lake I.2019) Chip

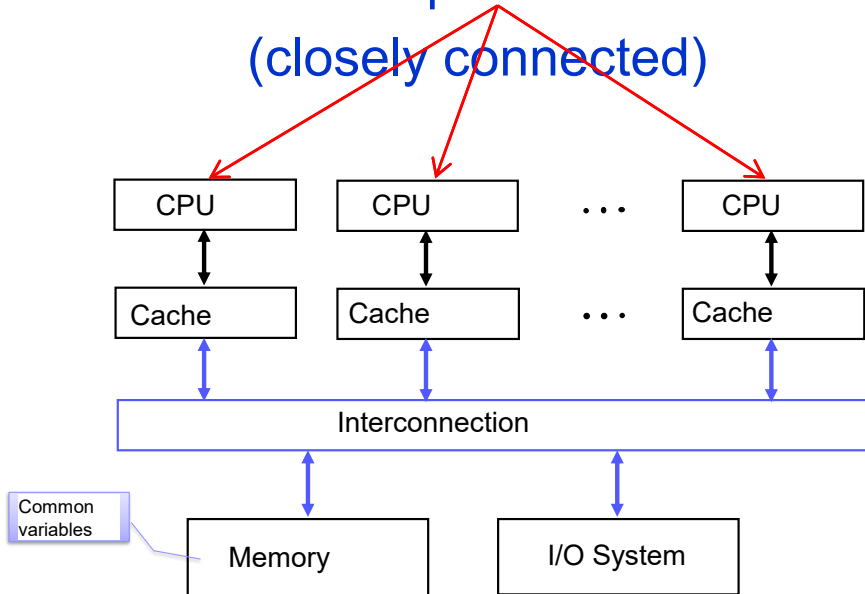




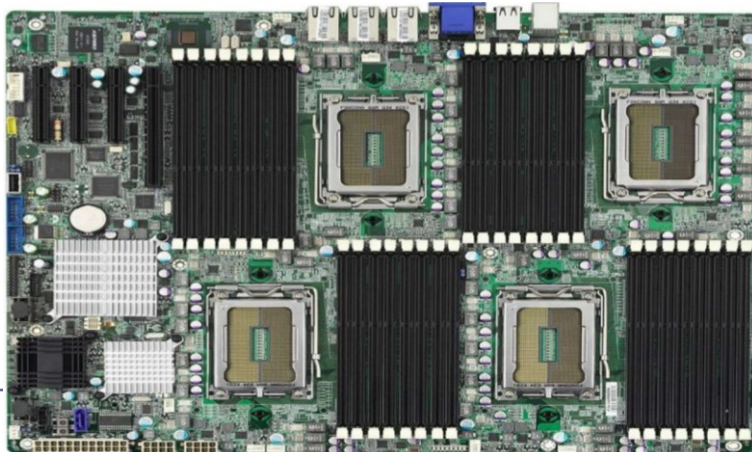
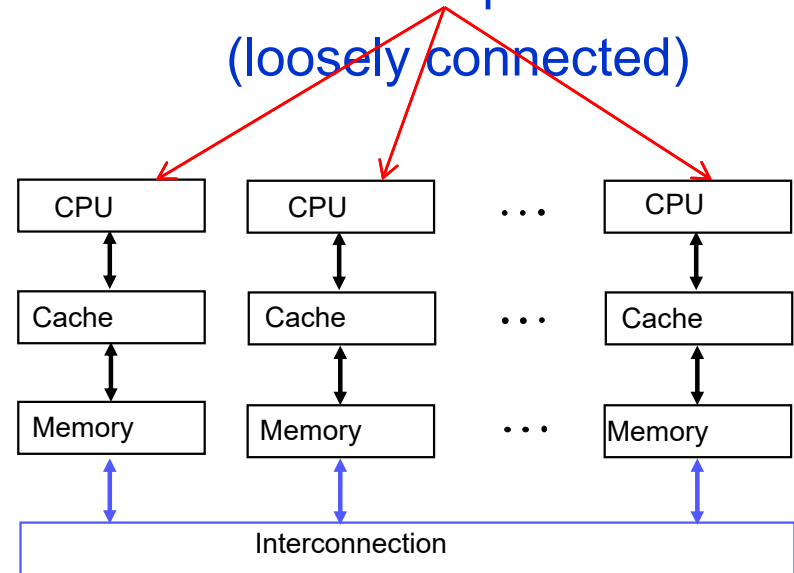
## Case: CPU-level parallelism: MIMD Computers

### Examples: MIMD (Multiple Instruction multiple Data)

#### Multiprocessor (closely connected)



#### Multicomputers (loosely connected)



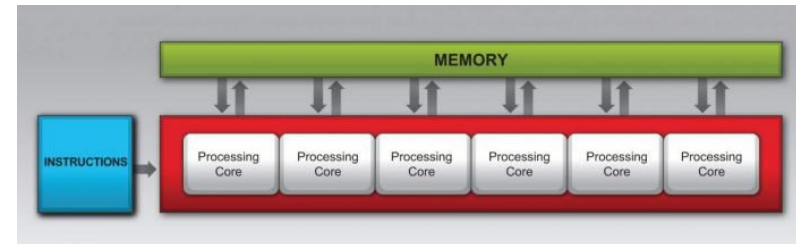
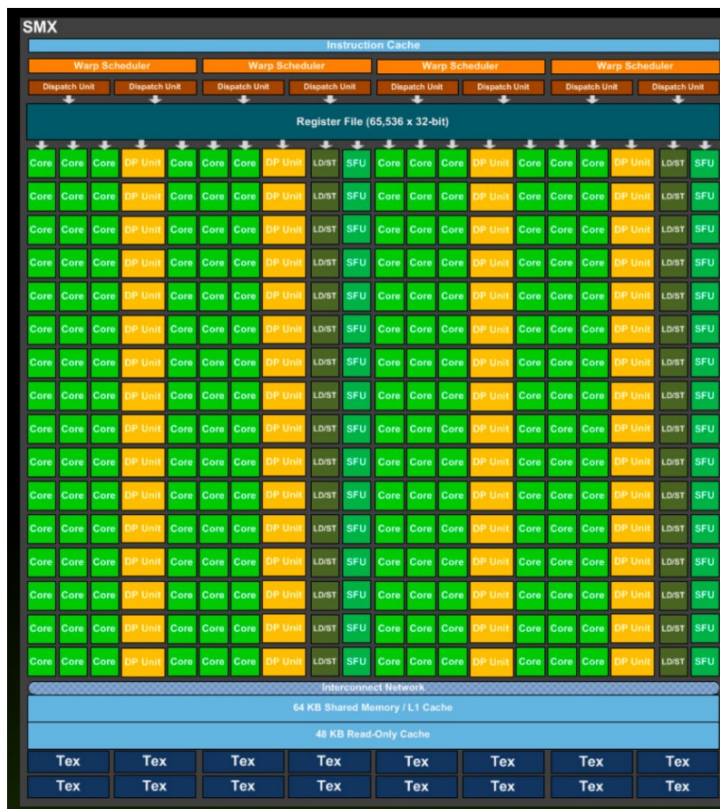




## Case: CPU-level parallelism: GPU, SIMD, Vector units

### ■ Parallel processing of data

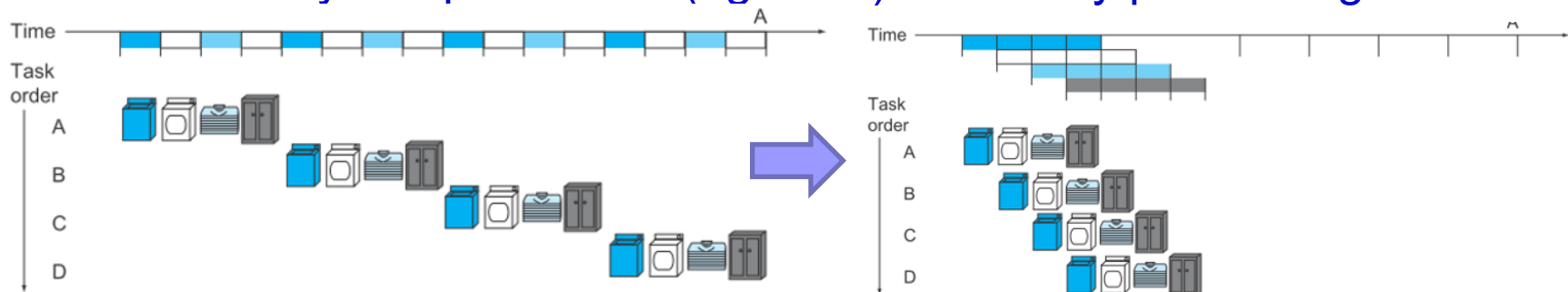
Tesla K40 ima vsega skupaj 1920 procesnih elementov (15 CU \* 128 PE v CU).



<https://doc.sling.si/workshops/programming-gpu/GPE/teslak40/>

## 6.6 Pipelined CPU (data unit)

- It is the realization of the CPU, where several instructions are executed simultaneously, so that the elementary steps of the instructions overlap.
- In a pipelined CPU, instructions are executed similar to industrial assembly line production (eg. cars) or laundry processing facilities:



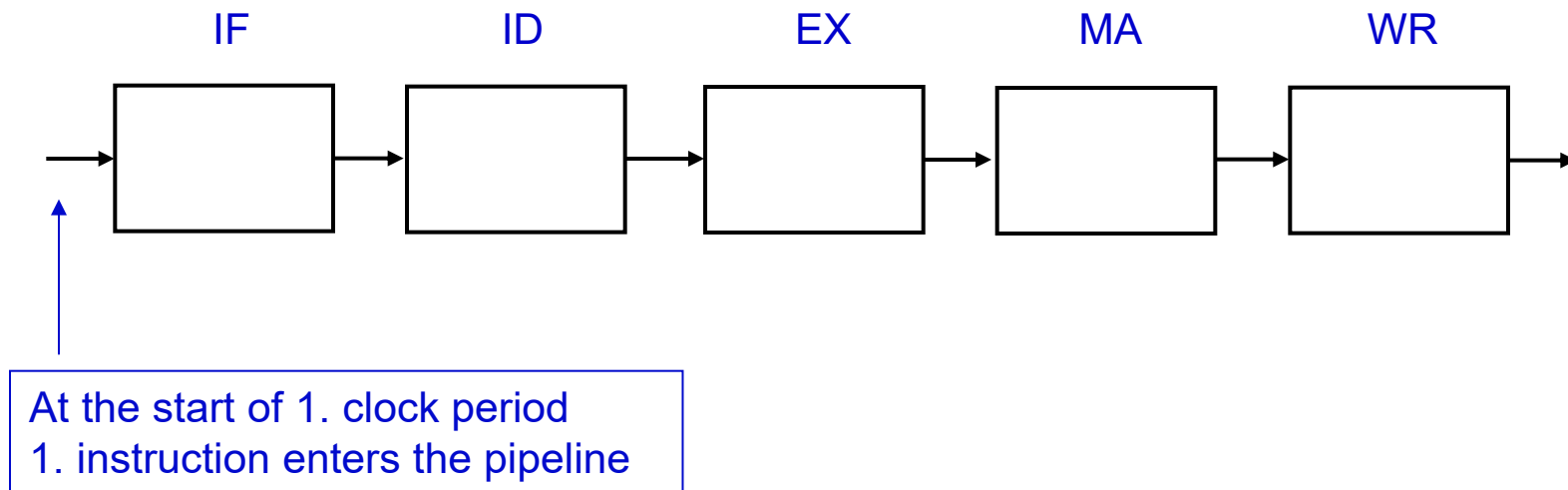
- Execution of the instruction can be divided into smaller elementary steps, **sub-operations**. Each sub-operation takes only fraction of the total time required to execute a instruction.



- CPU is divided into **stages** or **pipeline segments**, that correspond to sub-operations of instruction.
- each sub-operation is executed by a certain stage or segment of the pipeline.
- The stages are interconnected, on the one side instructions enter, then they travel through the stages, where sub-operations are executed, and they exit on on the other side of the pipeline.
- At the same time, there are as many instructions executed in parallel as many stages is there in the pipeline.



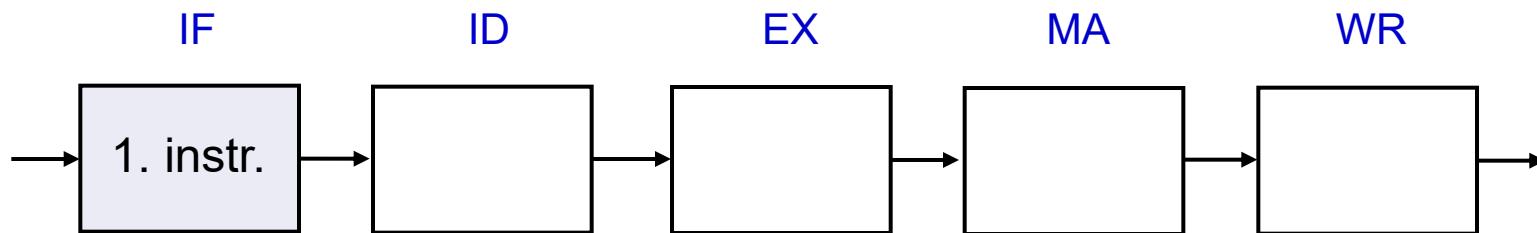
## Case: operation of 5-stage pipelined CPU





## Case: operation of 5-stage pipelined CPU

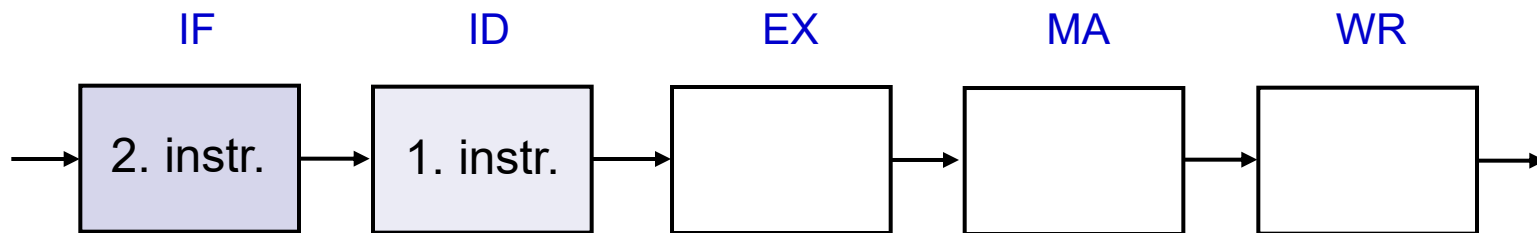
1. clock period





## Case: operation of 5-stage pipelined CPU

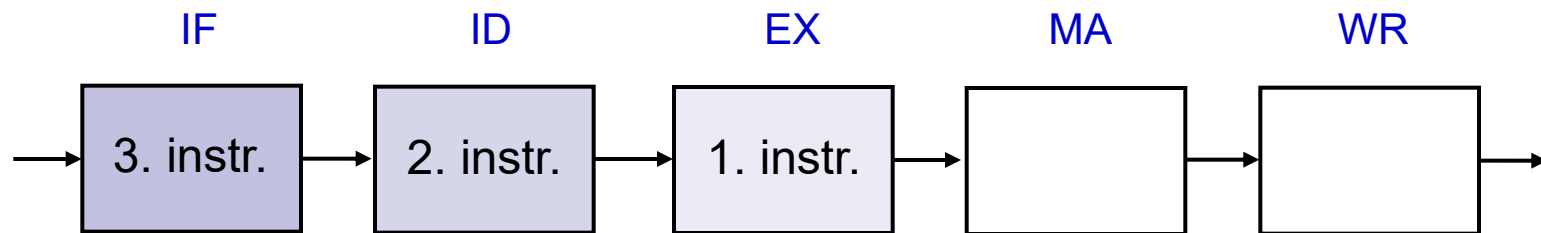
2. clock period





## Case: operation of 5-stage pipelined CPU

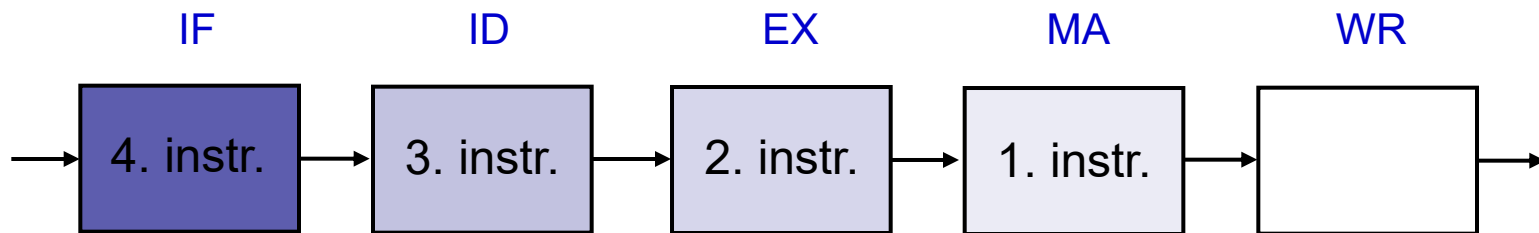
### 3. clock period





## Case: operation of 5-stage pipelined CPU

### 4. clock period

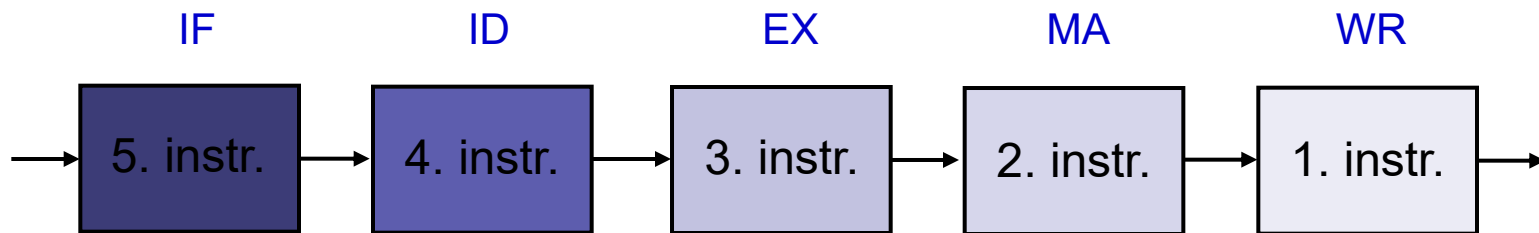






## Case: operation of 5-stage pipelined CPU

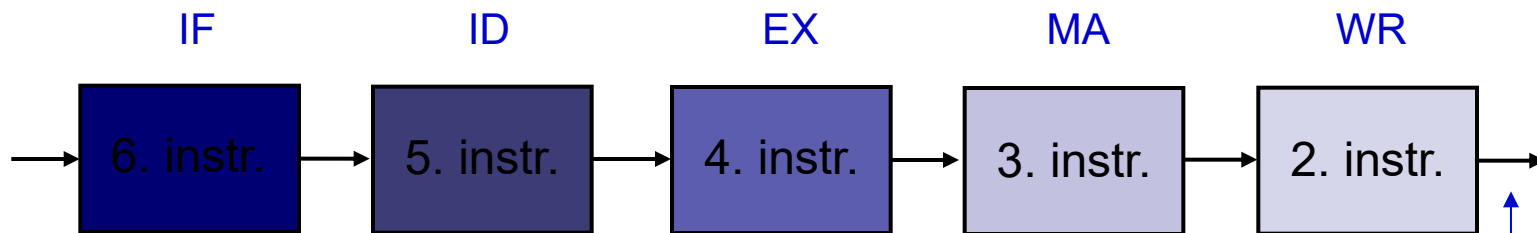
5. clock period





## Case: operation of 5-stage pipelined CPU

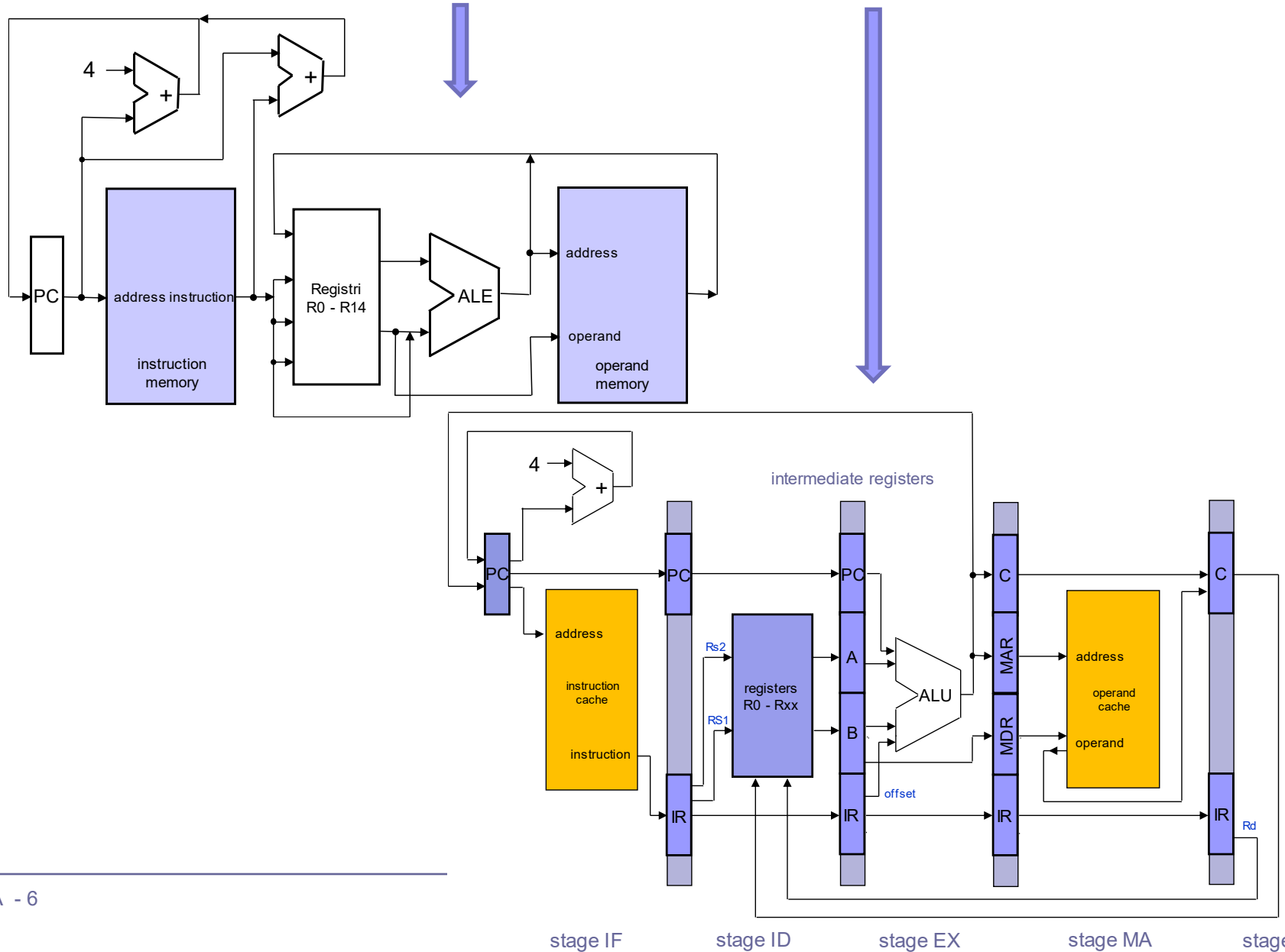
6. clock period



After the end of the 5th clock period, the first instruction completes execution (leaves the pipeline)



# Comparison of non-pipelined and 5-stage pipelined CPU



# Comparison of operation of non-pipelined and pipelined CPU

T1: Read instruction from memory

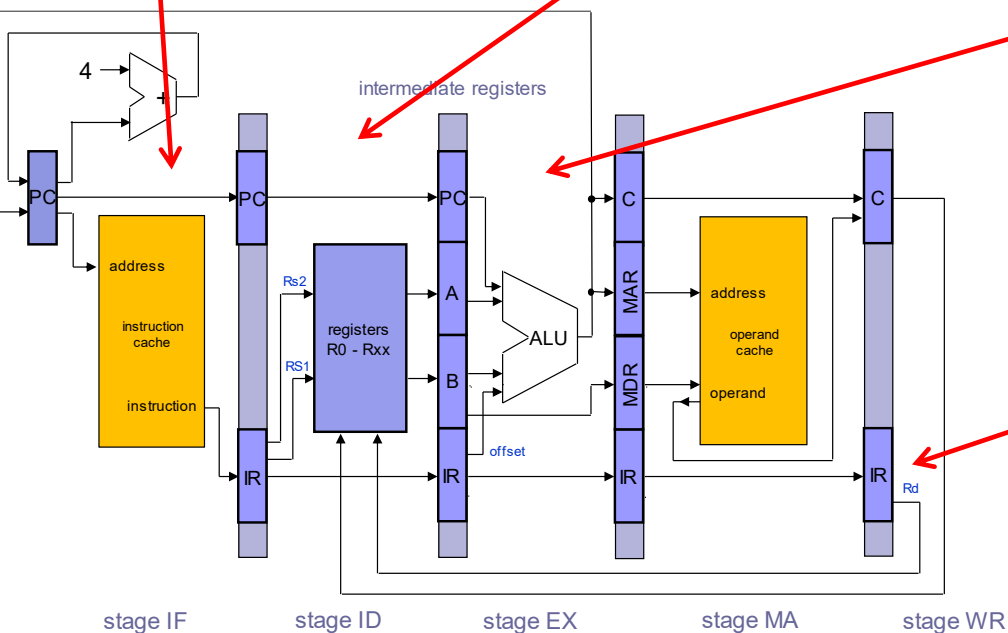
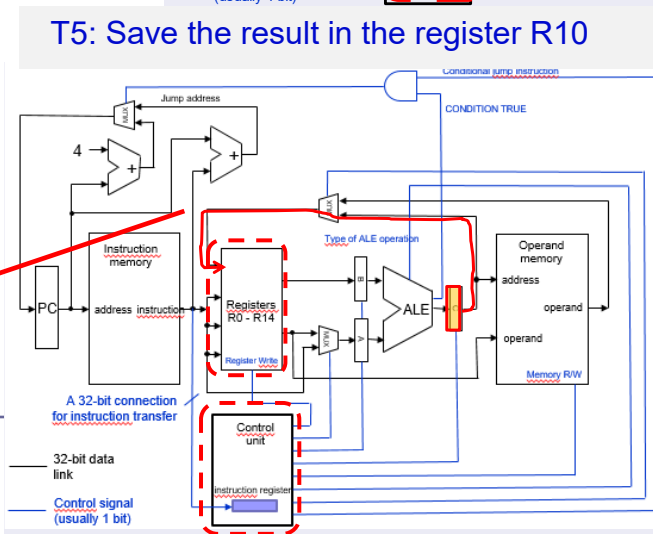
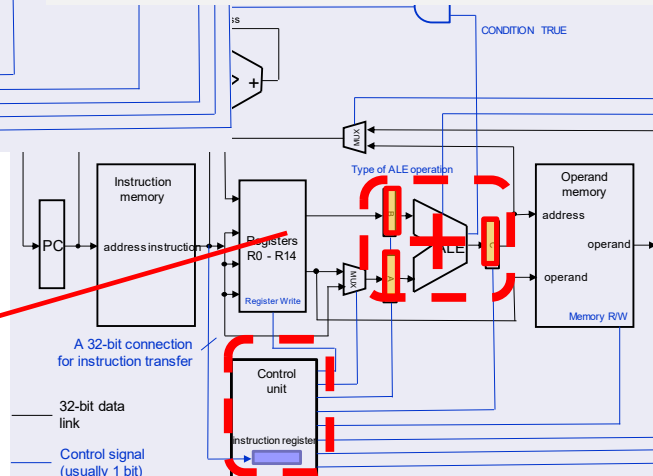
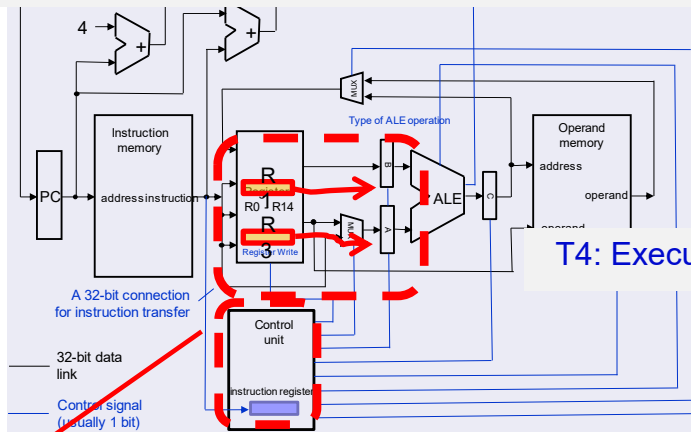
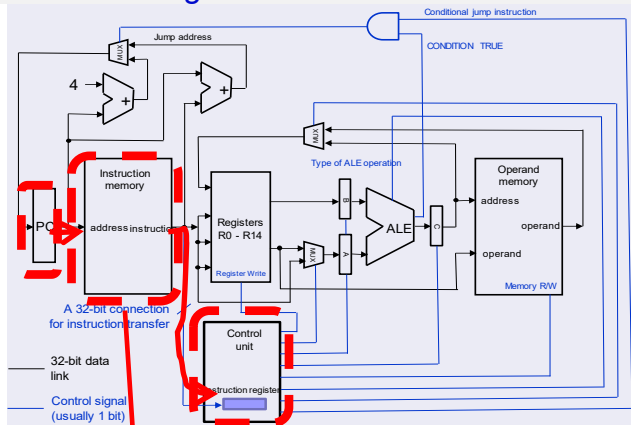
T2: Transfer of instruction from memory into the instruction register

T3: Decode the instruction and access to the operands in R1 and R3

T4: Execute operation (addition)

T5: Save the result in the register R10

ADD R10, R1, R3





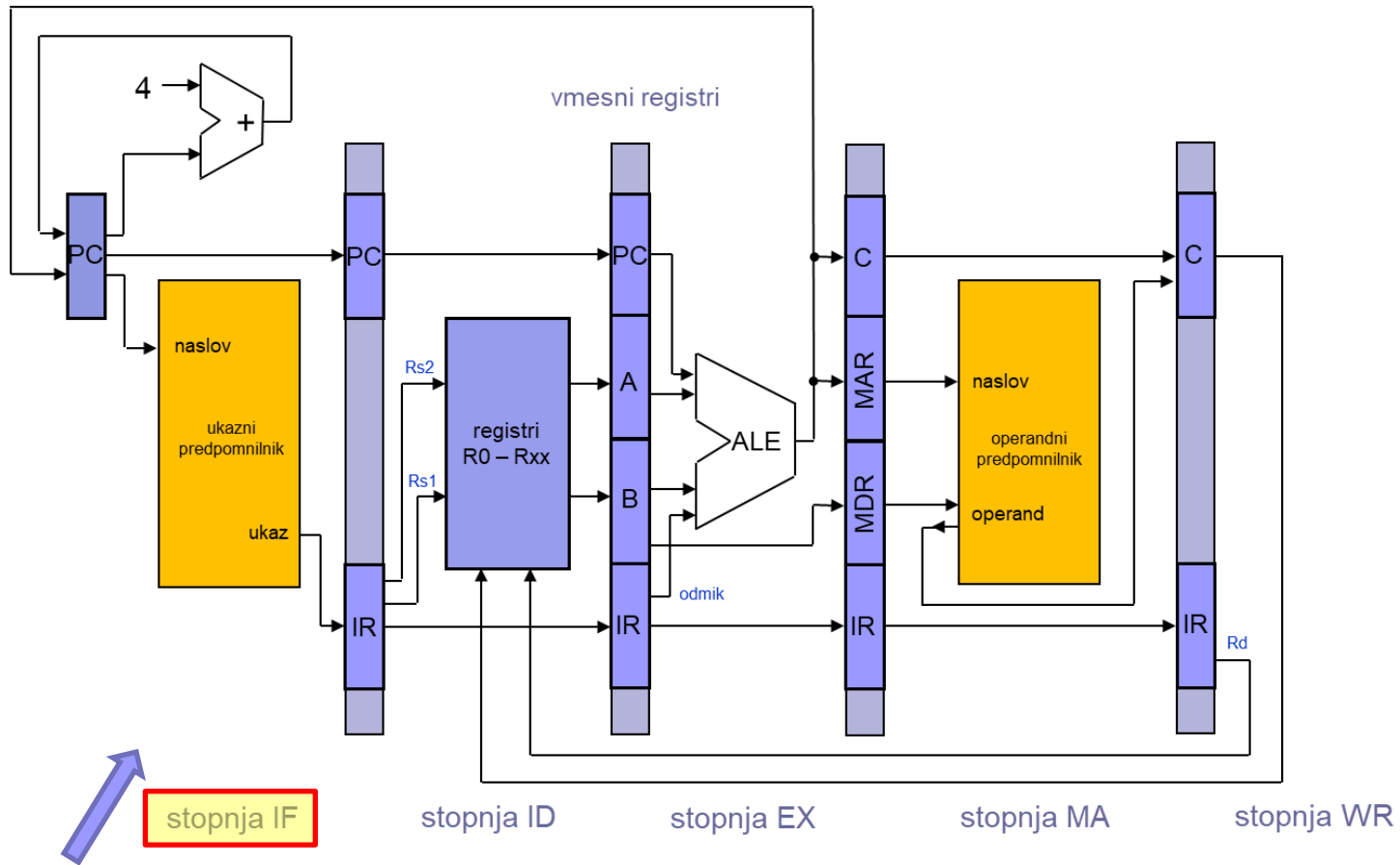
## Central processing unit - execute instructions

---

- The execution of the instructions can be divided into for example to 5 general elementary steps (5-stage pipeline):
  - Reading instruction (IF - Instruction Fetch )
  - Decoding instruction and access to registers (ID - Instruction decode )
  - Execution of instruction (EX – Execute )
  - Memory access (MA - Memory Access )
    - (Only for the LOAD instruction and STORE)
  - Saving the result in the register (WR - Write Register )
- If we can unify all the instructions to these common elementary steps, we can also speed up the execution of the instructions:
  - more instructions can be executed at the same time (each in its own elementary step) -> pipeline



- Performance of the pipelined CPU is determined by the rate of exit from the instruction pipeline.
- Since stages are linked together, the shifts of instructions from one stage to another has to be executed at the same time.
- The shifts typically occur each clock cycle.
- Duration of one clock period  $t_{CPE}$  can not be shorter than the time required to execute the slowest sub-operation in the pipeline.



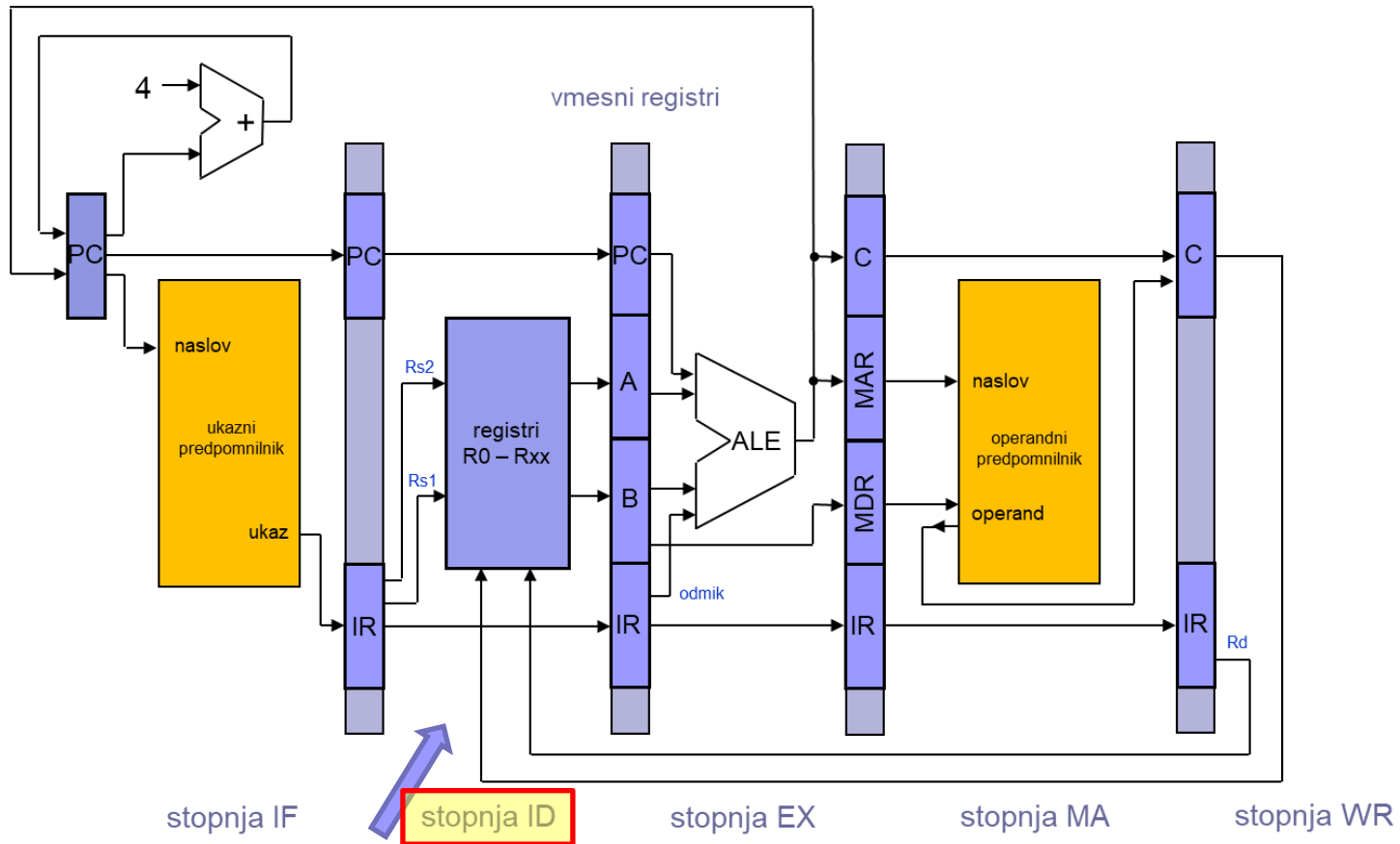
↑  
**stopnja IF**

Reading instruction  
IF = Instruction Fetch

1. Clock period

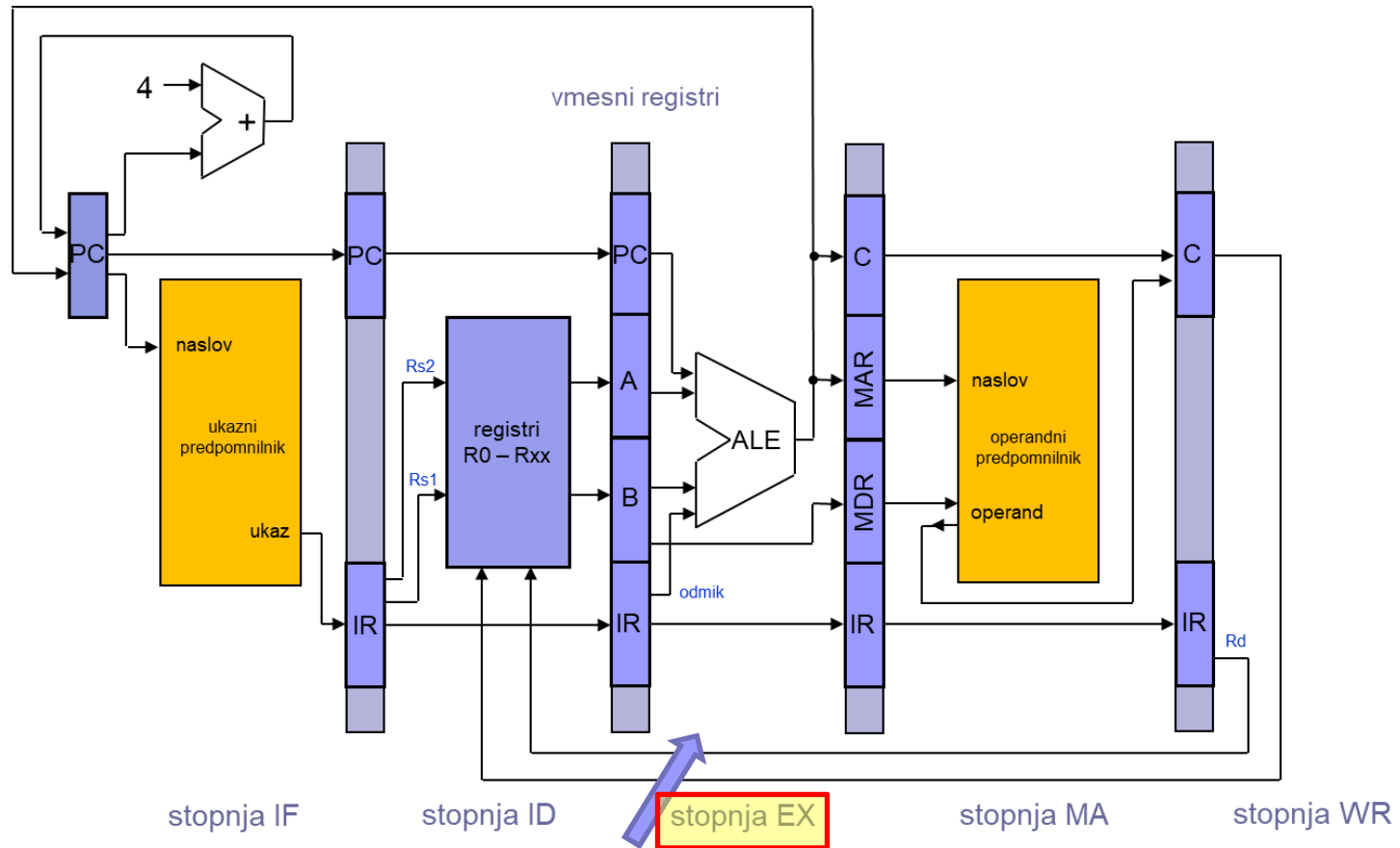


# Case: 5-stage pipelined CPU



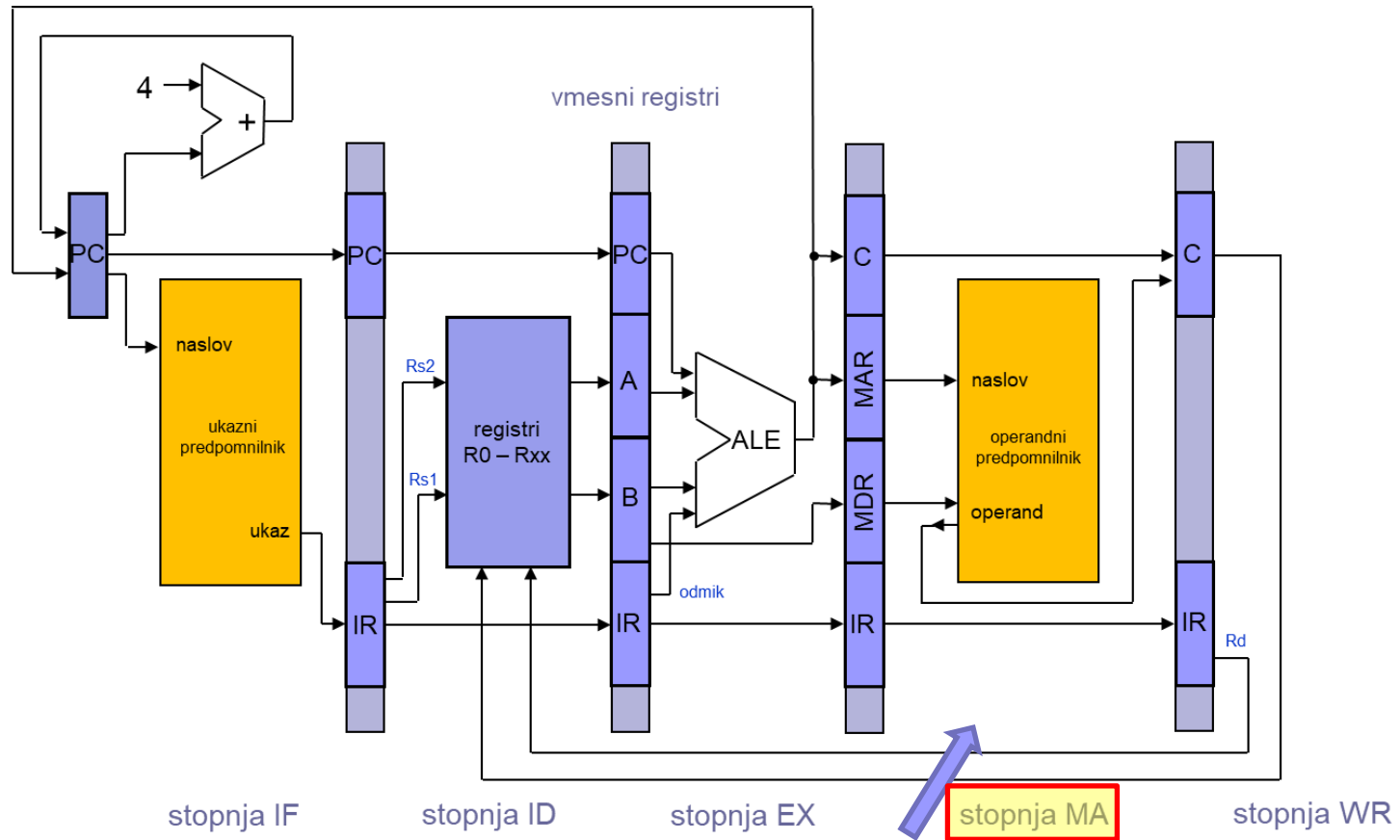
Decode instruction and  
access operands in  
the registers  
ID = Instruction Decode





Execution of operation  
EX = Execute

3. Clock period

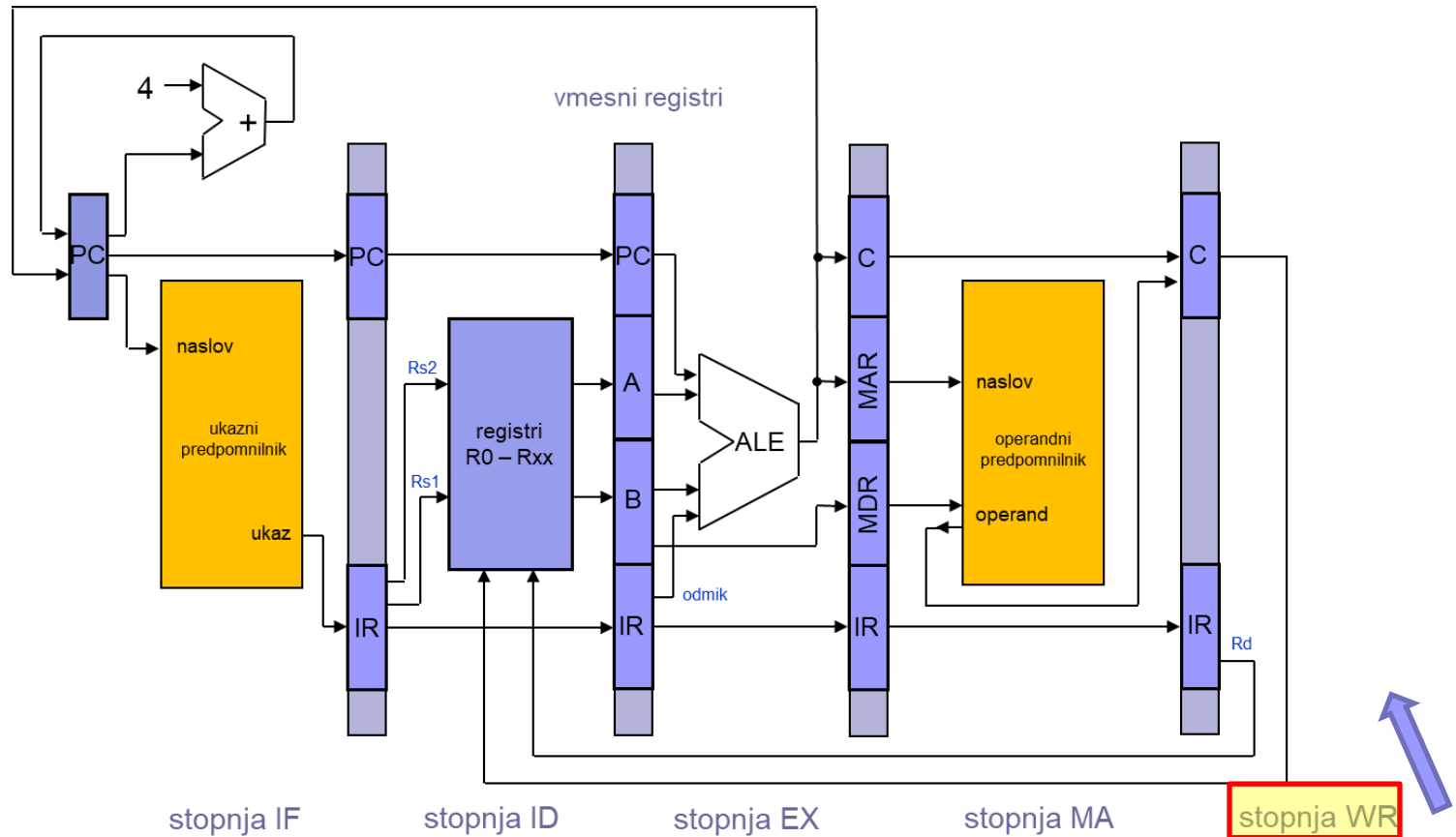


Access to operands in memory (LOAD / STORE)  
MA = Memory Access

## 4. Clock period



# Case: 5-stage pipelined CPU



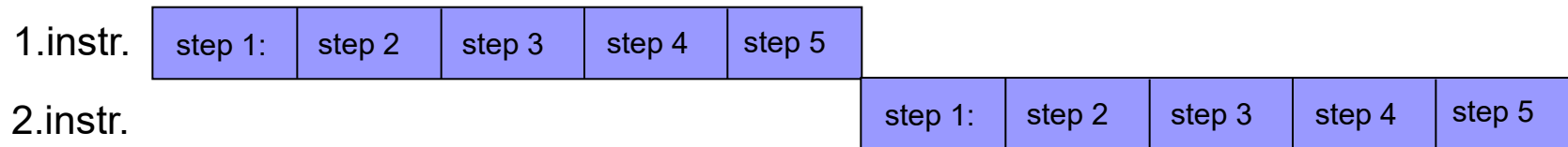
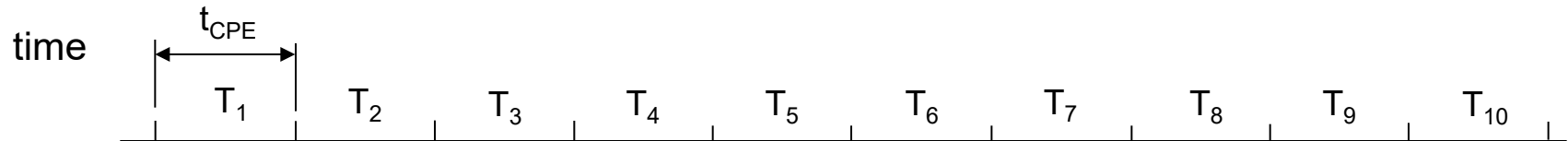
Saving result to register  
WR = Write Register

5. Clock period

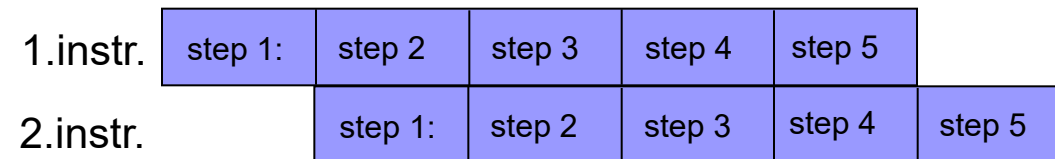
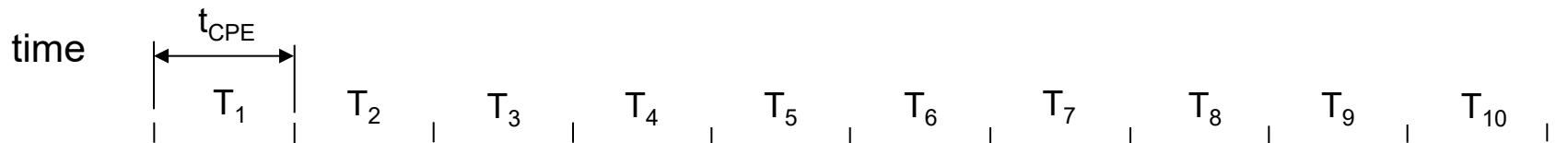


# Execution of instructions in non-pipelined and pipelined CPU

## Non-pipelined CPE



## Pipelined CPU





- Today, all more powerful processors are designed as a pipelined processors.
- In developing the pipelined CPU, it is important that executions of all sub-operations take about the same time - balanced pipeline.
- With an ideally balanced CPU with  $N$  stages or segments, the performance is  $N$  times greater than non-pipelined CPU.
- Each individual instruction is not executed any faster, but there are  $N$  instructions in the pipeline executed at the same time.



- At the output of the pipeline, we get  $N$  times more executed instructions than in non-pipelined CPU.
- The average number of clock cycles for the instruction ( $CPI$ ) is ideally  $N$  times lower than at the non-pipelined CPU.
- The duration of the execution of each instruction (latency) is equal to  $N \times t_{CPE}$ , that is, at the same clock period, the same in the non-pipelined CPU.



- Can we at a sufficiently large number of stages  $N$  make CPU much faster ( $N$  times faster)?
  - No. Instructions, that are in the pipeline at the same time (each in its stage), can depend on each other in some way dependent and therefore a certain instruction can not be always executed in next clock period.
  
- These events are called **pipeline hazards**.



- There are three types of pipeline hazards:
  - **structural hazards** – when several stages of the pipeline in the same clock period requires the same unit,
  - **data hazards** - where some instruction needs the result of the previous instruction, but is not yet available
  - **control hazards** – at the instructions that change the value of the PC (control instructions: jumps, branches, calls, ...)

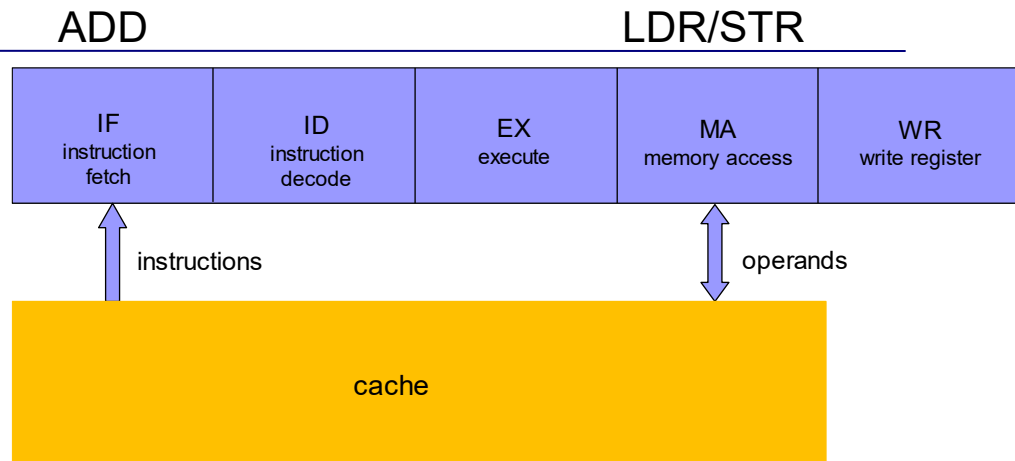




## Pipelined CPU - types of pipeline hazards:

### ■ structural hazards

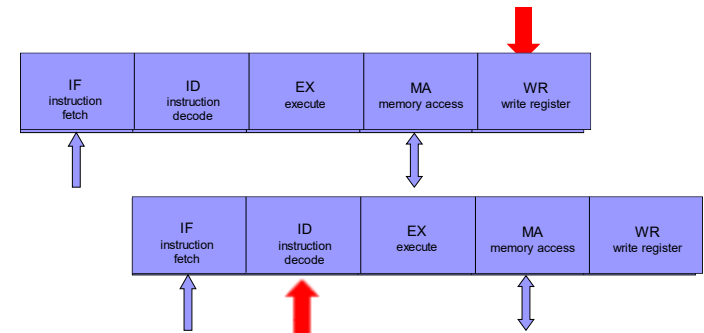
- access to the same unit (eg. cache)



### ■ data hazards

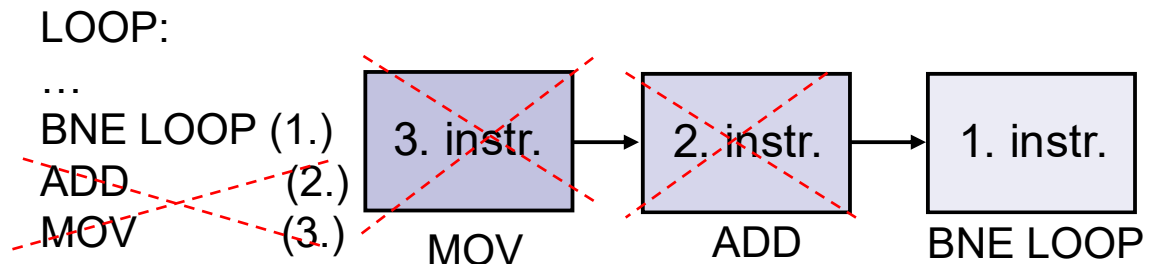
- operand dependence between instructions

ADD r1, r2, r3  
ADD r5, r3, r1



### ■ hazard control

- branch instructions (filling the pipeline)

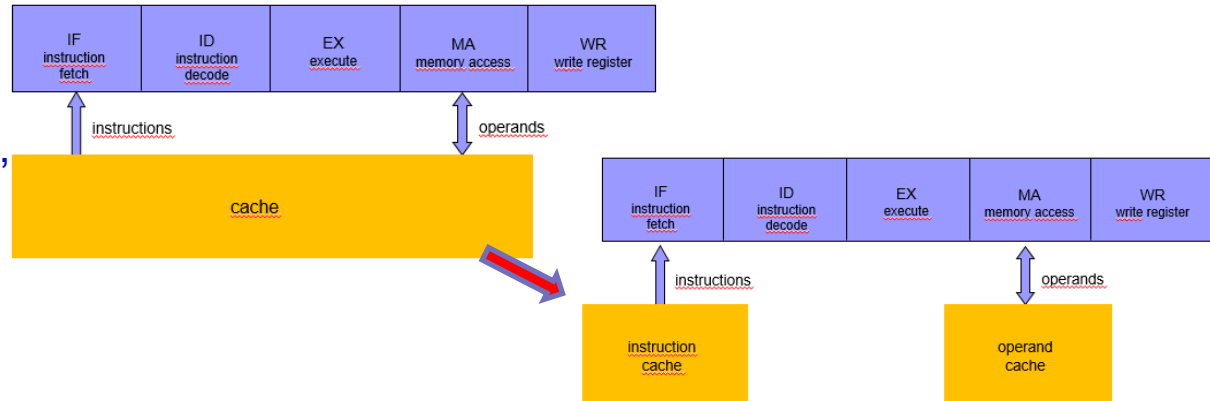




# Pipelined CPU - pipeline hazards: common solutions

## structural hazards

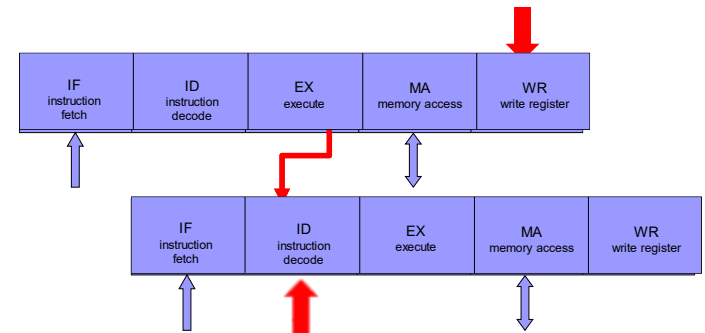
- Solution -> separation of caches (instructions, operands - Harvard Arch.



## data hazards

- Instruction reordering can also help (programmer, compiler)
- Solutions -> stall, operand forwarding between the stages

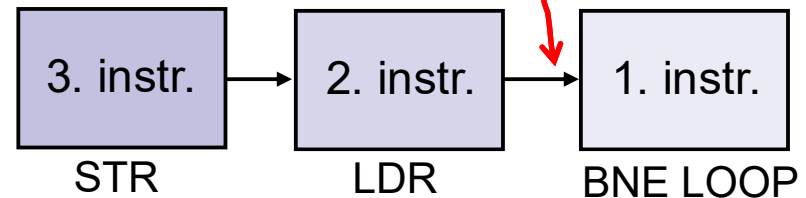
```
ADD r1, r2, r3
ADD r5, r3, r1
```



## control hazards

- Solution -> predict the condition and branch address

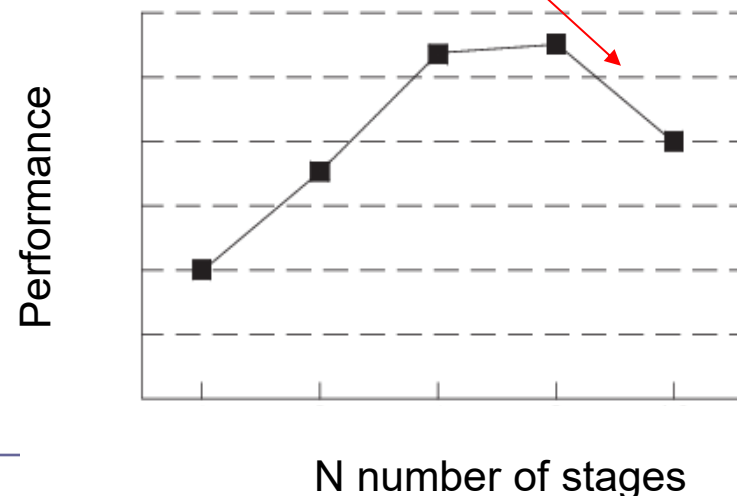
```
LOOP:
LDR      (2.)
STR      (3.)
BNE LOOP (1.)
ADD
MOV
```





## Pipelined CPU

- Due to the risk of pipeline hazards, part of the pipeline at least has to stop until hazard is resolved (the pipeline at that time does not accept new instructions).
- The increase in speed, therefore, is not ***N - times***.
- By increasing the number of stages  $N$ , the pipeline hazards occur more frequently and the pipeline is no longer as effective as with lower number of stages.





## 6.7 Cases of 5-stage pipelined CPU

- General 5-stage pipeline
- FRI SMS Atmel 9260 ARMv5



## General 5-stage pipeline

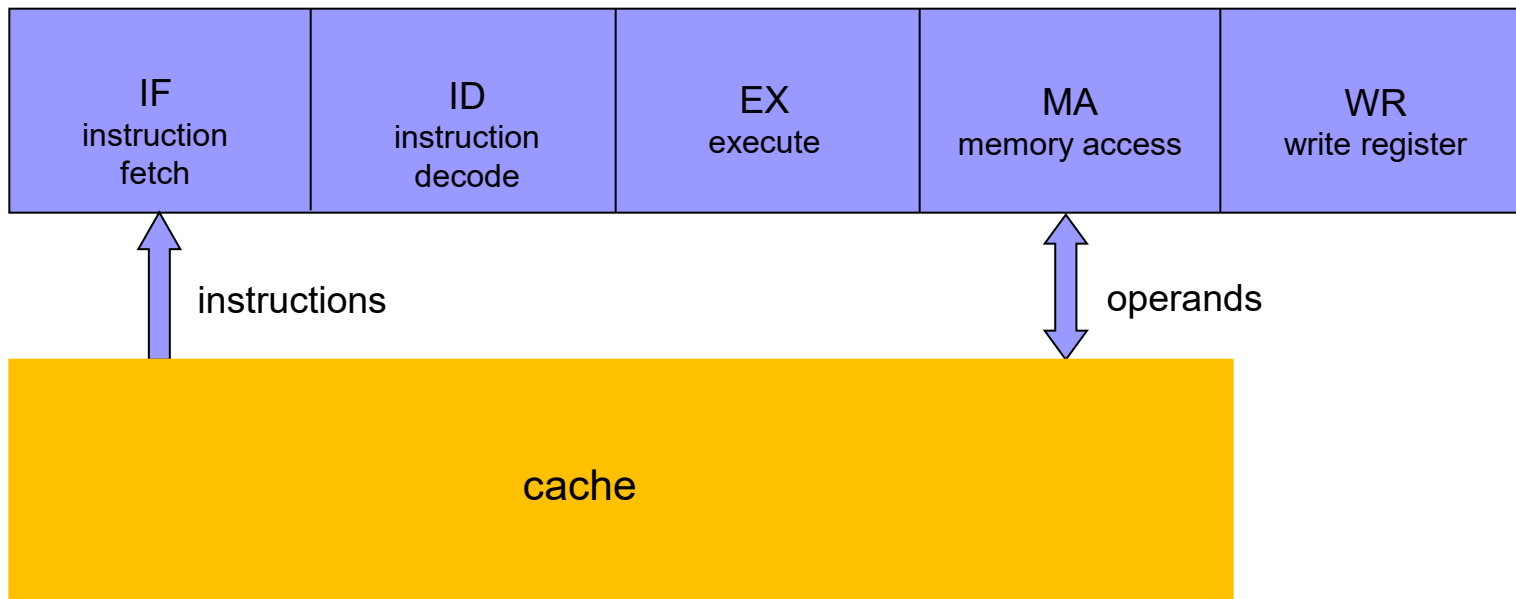
- The base should be the execution of instructions in five steps, as we described in the previous section.
- Execution of the instruction is divided into 5 sub-operations in accordance with the steps from the previous section, and CPU divided in five stages or segments:
  - Stage IF (Instruction Fetch) - read instruction
  - Stage ID (Instruction decode) – decode the instruction and access to registers
  - Stage EX (Execute) - the execution of the operation
  - Stage MA (Memory Access) - access memory
  - Stage WR (Write Register) - save the result



- Each stage of the pipeline must execute its sub-operation in single clock cycle (period).
- The IF and MA stages can simultaneously access memory (in same clock period) - a structural hazard happens.
- To eliminate this kind of structural hazards, we must divide the cache into separate instruction and operand caches (Harvard architecture principle).



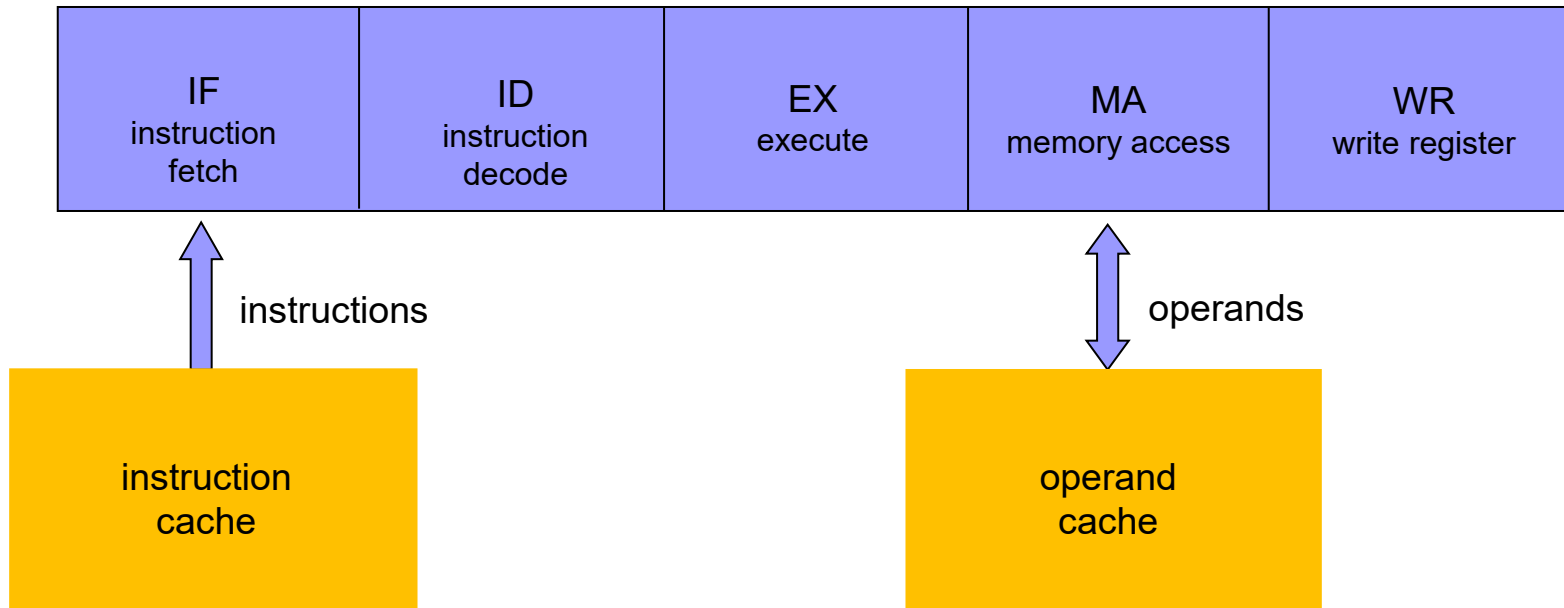
## Pipelined CPU



For the simultaneous access to instruction (stage IF) and operand in cache (stage MA), the structural hazard occurs in the pipeline



## Pipelined CPU



Structural hazard, that would occur due to simultaneous access of stages IF and MA to memory, is eliminated by using Harvard architecture on caches



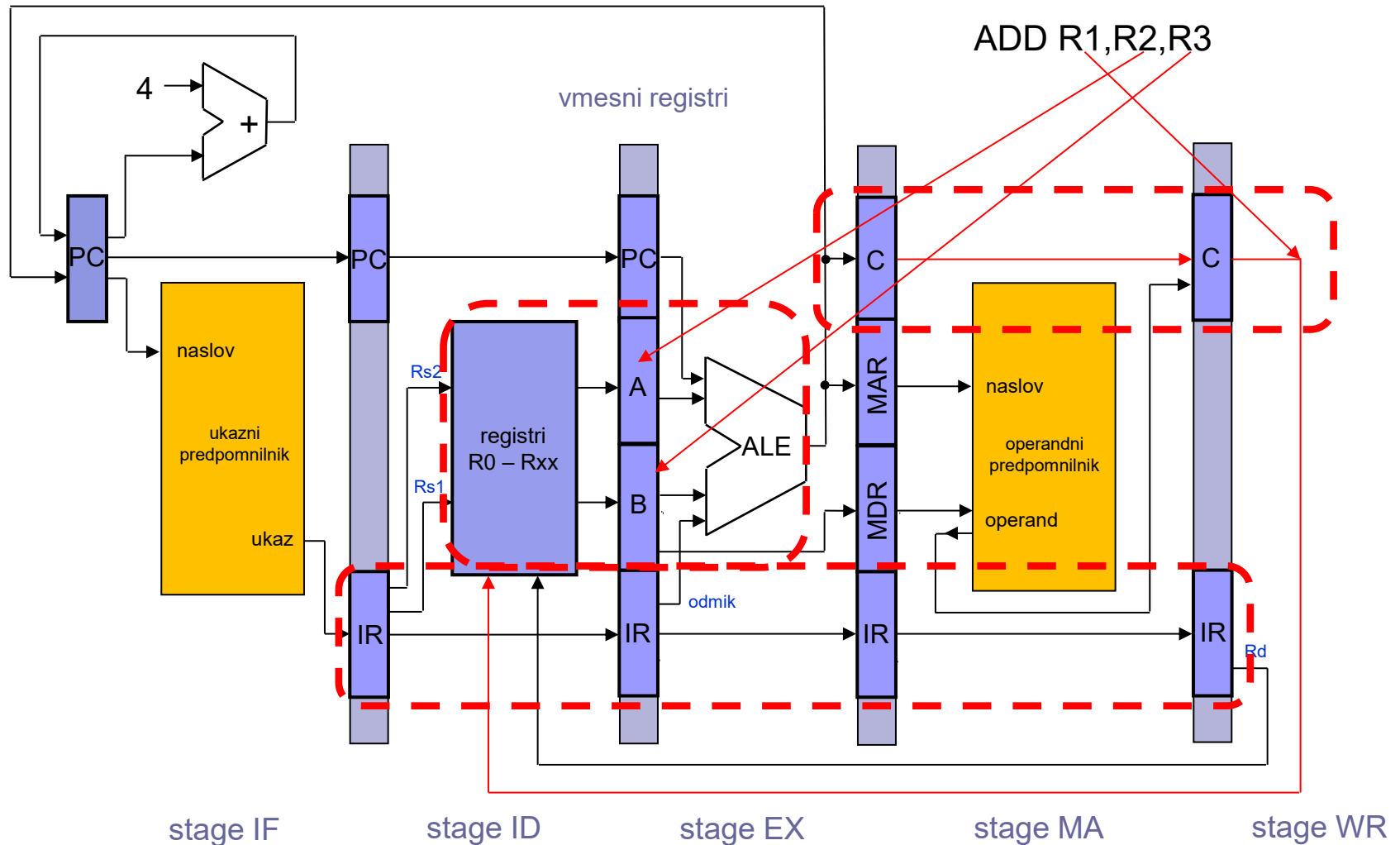


- In the IF stage of pipelined CPU, the access to the instruction cache happens each clock period, however, in the non-pipelined CPU access happens only every five clock periods (in case of 5 clock periods instructions).
- The speed of information transfer between the cache and the CPU must be in case of pipelined CPU, five times higher than in non-pipelined CPU.
- When designing the pipelined CPU, it is important to ensure that CPU units (registers, ALU, ...) are not required to do two different operations.



# Case: structure of 5-stage pipelined CPU

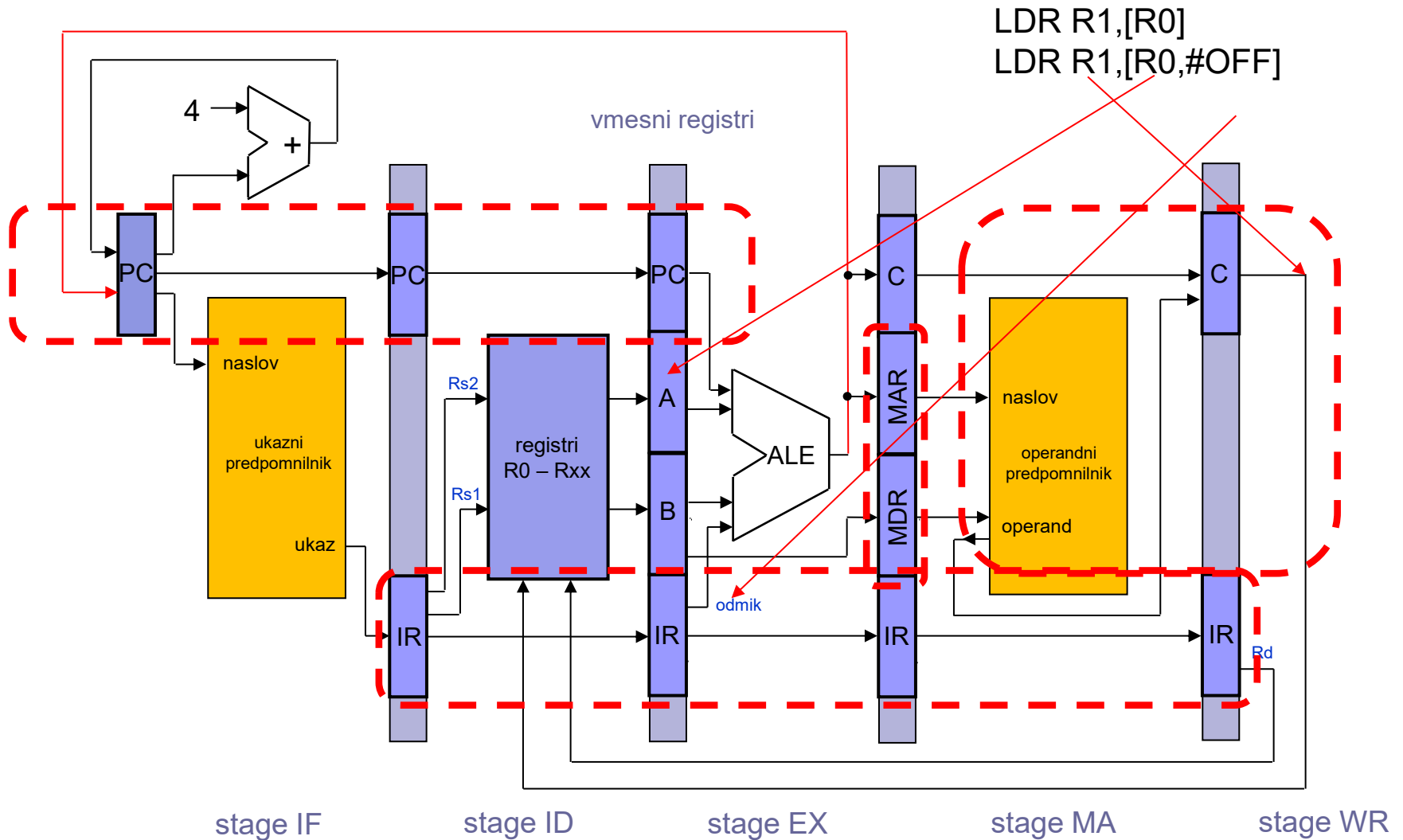
(ALU instruction: e.g. ADD R1,R2,R3)





# Case: structure of 5-stage pipelined CPU

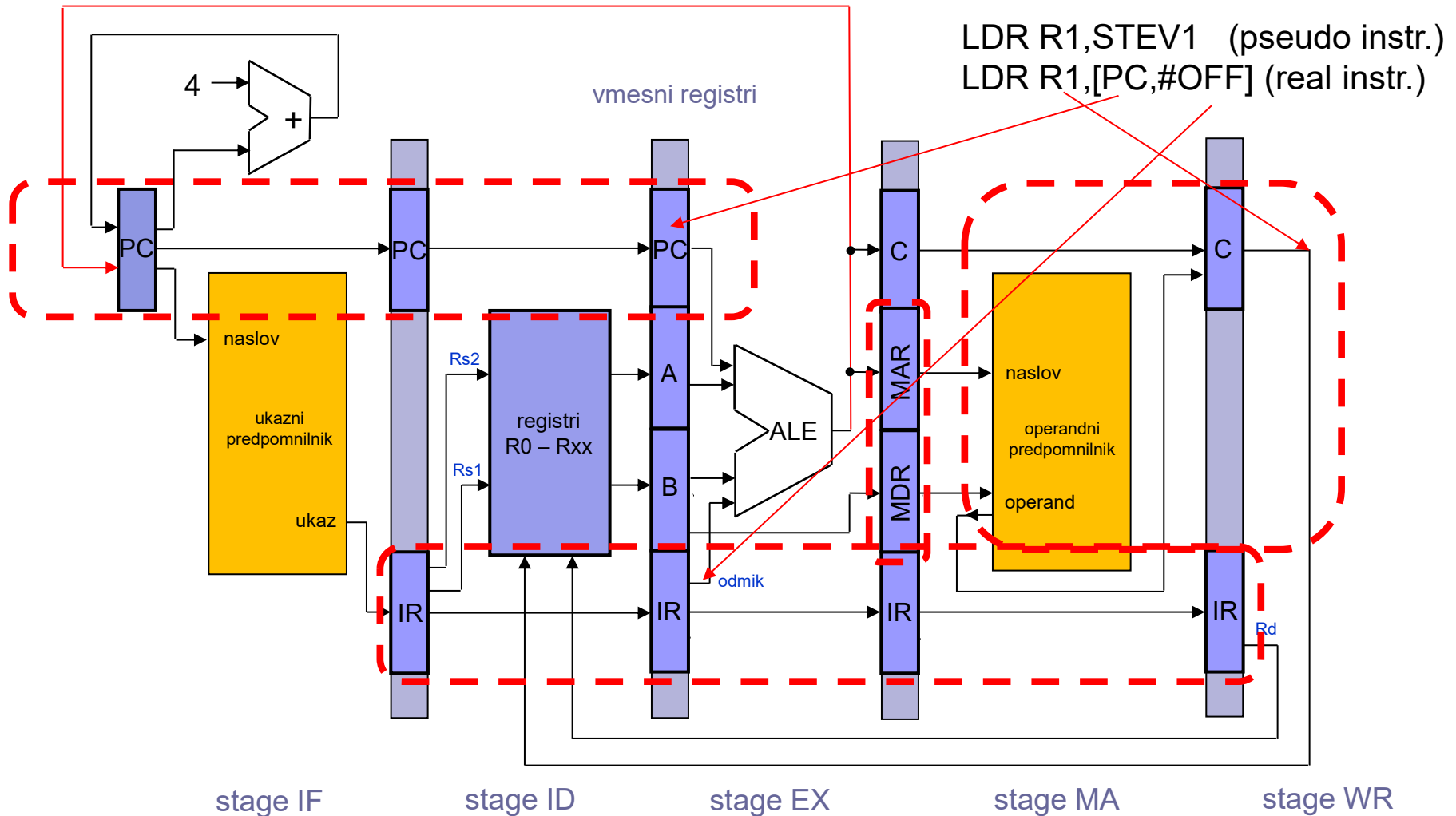
(LOAD/STORE instruction: Calculation of address in EX, access in MA)





# Case: structure of 5-stage pipelined CPU

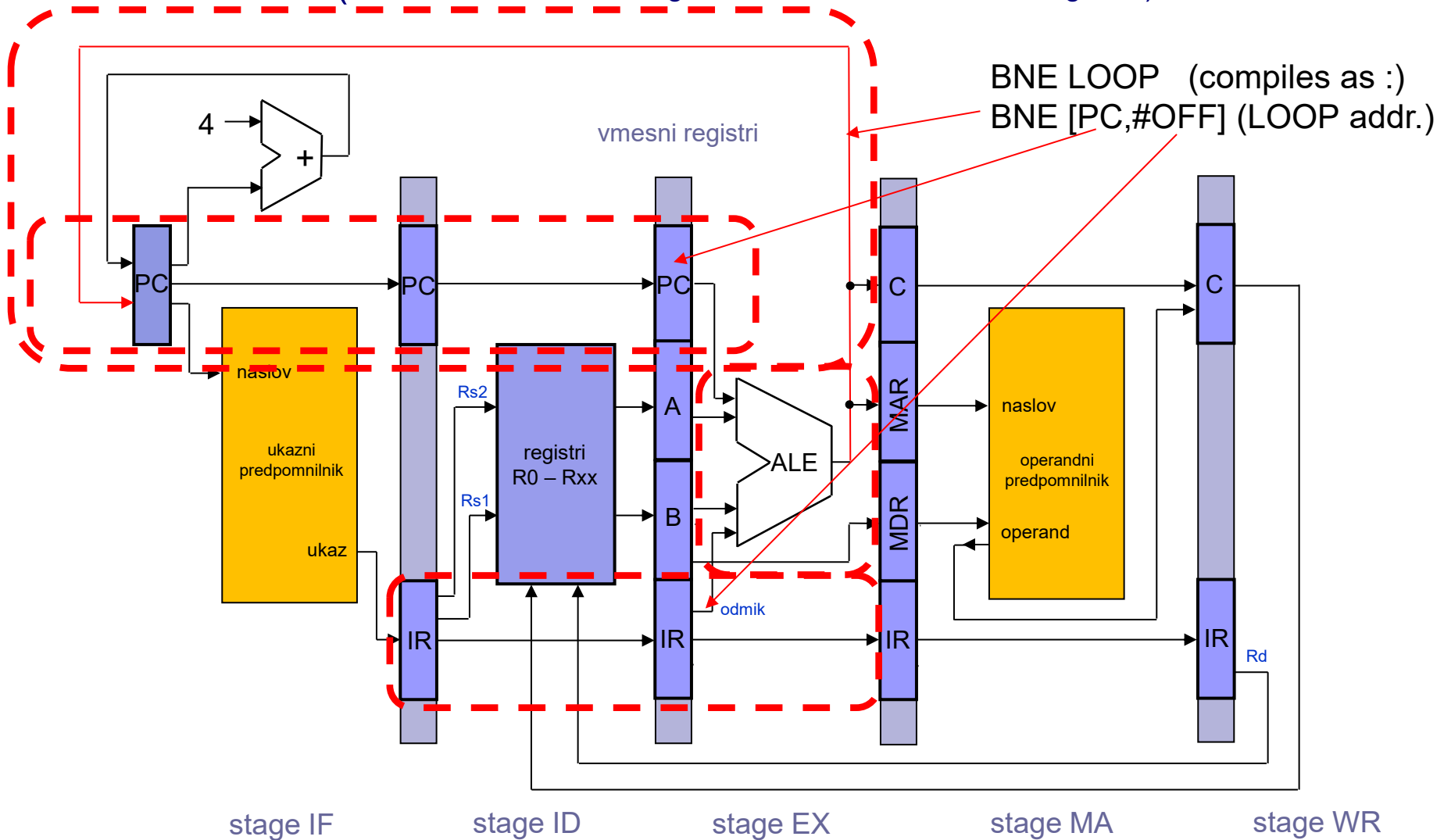
(LOAD/STORE instruction: Calculation of address in EX, access in MA)









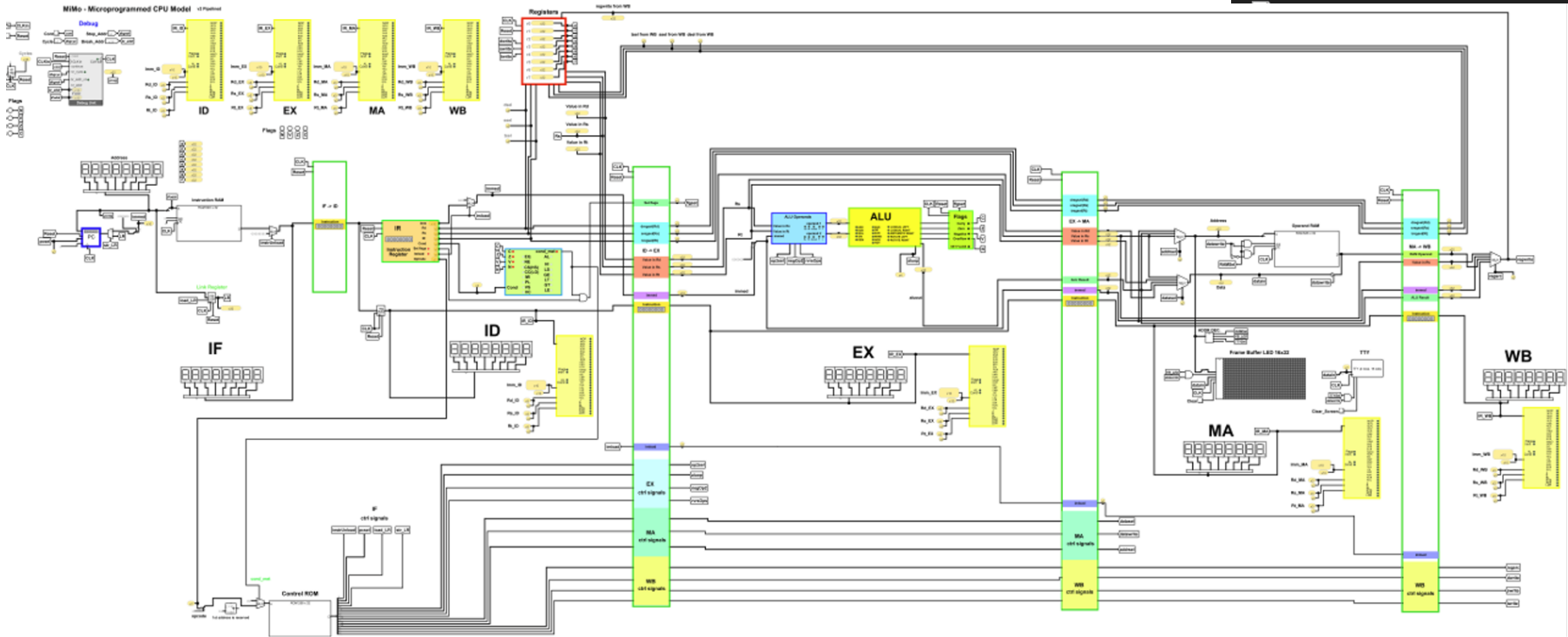
# Case: structure of 5-stage pipelined CPU

(BRANCH instructions: e.g. B, BNE LABEL in ALU in stage EX)








-  mimo\_32bit\_v2.1 - Zaklenitev.circ
-  mimo\_32bit\_v2.2 - Premoscanje.circ
-  mimo\_32bit\_v2.3 - Predikcije.circ
-  mimo\_32bit\_v2.circ

# MiMo v2 - 5. st. pipeline in Logisim



loop: @ stall	forwarding
mov r3, #3 @ 5, 22	5, 17
ldr r1, [r2] @ 6, 23	6, 18
add r1, r1, #1 @ 10, 27	8, 20 (here
add r7, r7, #1 @ 11, 28	9, 21
str r2, r1 @ 14 (written to operand memory on cycle 13, but left pip	
subs r4, r3, r1 @ 15, 32	11, 23
add r5, r5, #1 @ 17, 34	12, 24
add r7, r7, #1 @ 18, 35	13, 25
add r6, r1, r4 @ 19, 36	14, 26
jne loop @ 20, 37	15, 27

Comparison of stall and forwarding:  
37 (stall) and 27 (forwarding) clock periods for program execution

-  test1-nops\_needed.txt
-  test2-zaklenitev\_with\_no\_nops.txt
-  test3-operand\_forwarding.txt
-  test4-jumps\_in\_op\_forwarding.txt
-  test5-stall\_vs\_forwarding.txt

[https://github.com/LAPSyLAB/MiMo\\_Student\\_Release/tree/main/MiMo\\_v2\\_Pipelined\\_versions](https://github.com/LAPSyLAB/MiMo_Student_Release/tree/main/MiMo_v2_Pipelined_versions)



## Case: structure of 5-stage pipelined CPU

- The pipeline has 5 stages; between them there are intermediate registers in which the results of sub-operations in each level are stored and all data that is needed in following stages.
- In stage IF, the instruction is read and transferred to the instruction register, and the content of the program counter PC is increased by 4 (instructions are 4 bytes long).
- Program Counter is necessary to be increased in stage IF because usually in each clock period, one instruction is fetched from instruction cache.

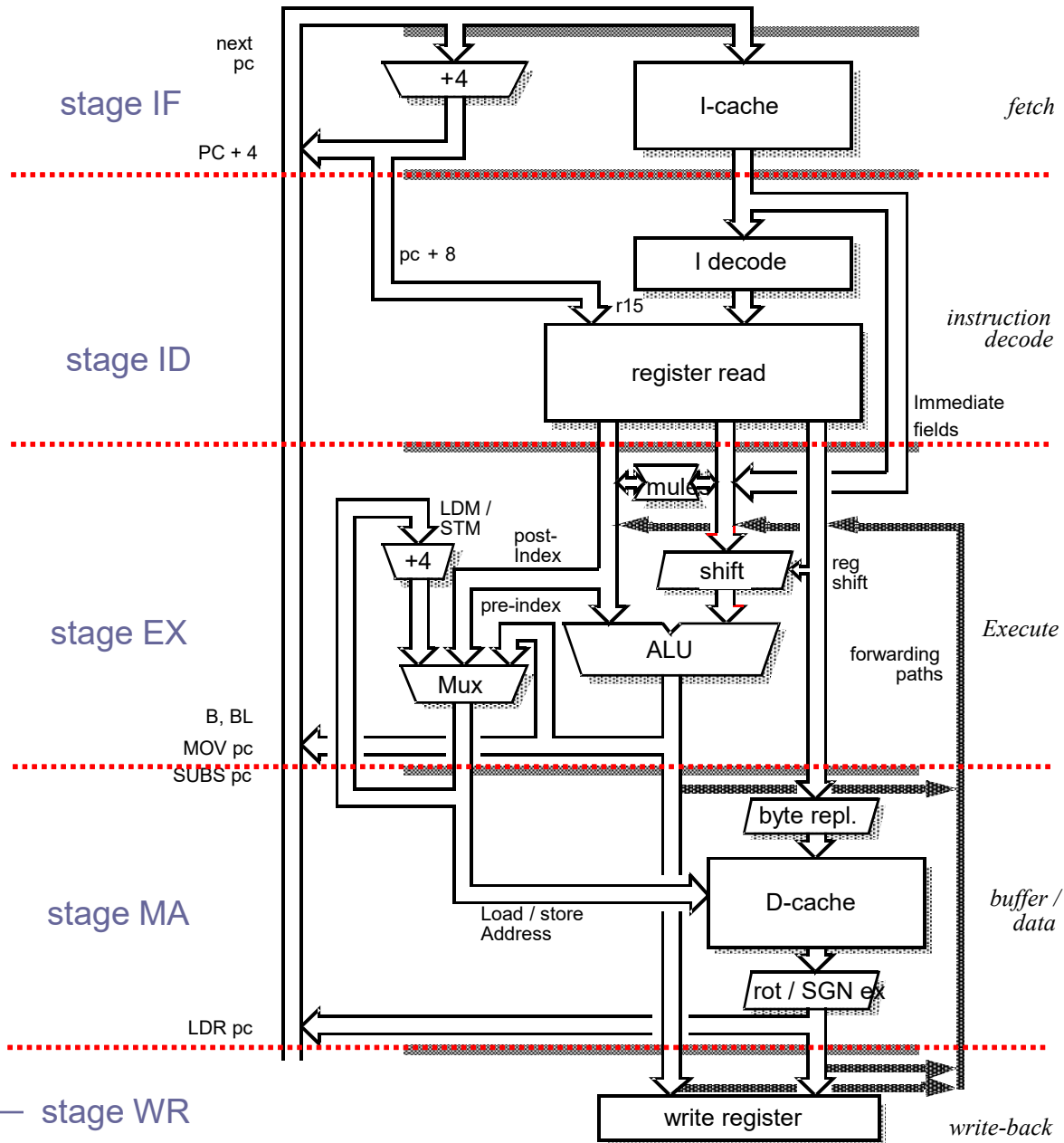


- The instruction currently executed (pointed by PC content) is stored in the intermediate registers (IR) because it is needed for branch instructions in the EX stage.
- Branch instructions usually write new address into PC (branch or target address), which is calculated by ALU in stage EX.
- Address for operands in instructions LOAD/STORE (indirect addressing) is also calculated by ALU in stage EX.
- Each stage executes its own instructions, therefore the intermediate registers IR in all stages always store the instructions that are read from instruction cache every clock period.



# Case: Structure of 5-stage pipelined CPU:

FRI SMS - Atmel 9260,  
ARMv5 architecture





## 6.8 Multiple issue processors

- With pipelined CPU and solving the pipeline hazards, we can achieve CPI values close to 1.
- If we want to reduce the CPI below 1, we must fetch and issue several instructions in in each clock period (and also executed them).
- Such processors are denoted as multiple-issue processors and can be divided into two groups:
  - superscalar processors – instructions, that are executed in parallel, are determined by a logic in a processor – dynamic decision
  - VLIW processors - instructions, that are executed in parallel, are determined by a program (compiler) – static decision

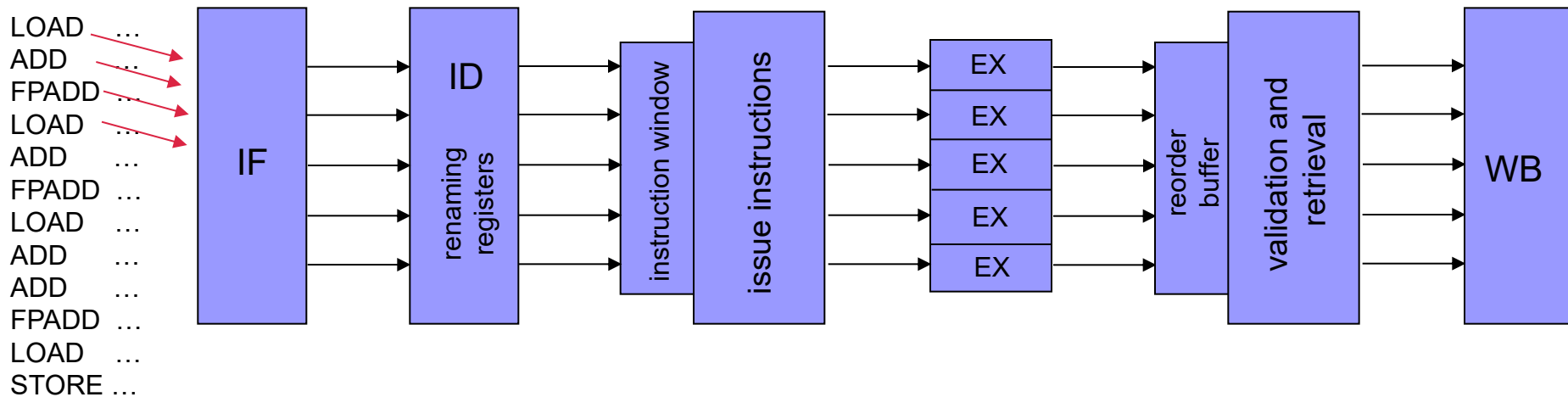


**Superscalar processor** is a pipelined processor which is capable of simultaneous fetching, decoding and executing several instructions.

- The number of fetched and issued instructions in one clock period is dynamically adjusted during the program execution and determined by processor's logic.
- Processor, that can issue a maximum of  $n$  instructions is denoted as *n-issue* superscalar processor.
- Parallel (superscalar) performance requires additional interfaces and additional stages for determining interdependencies, validation and eventual retrieval of results ->



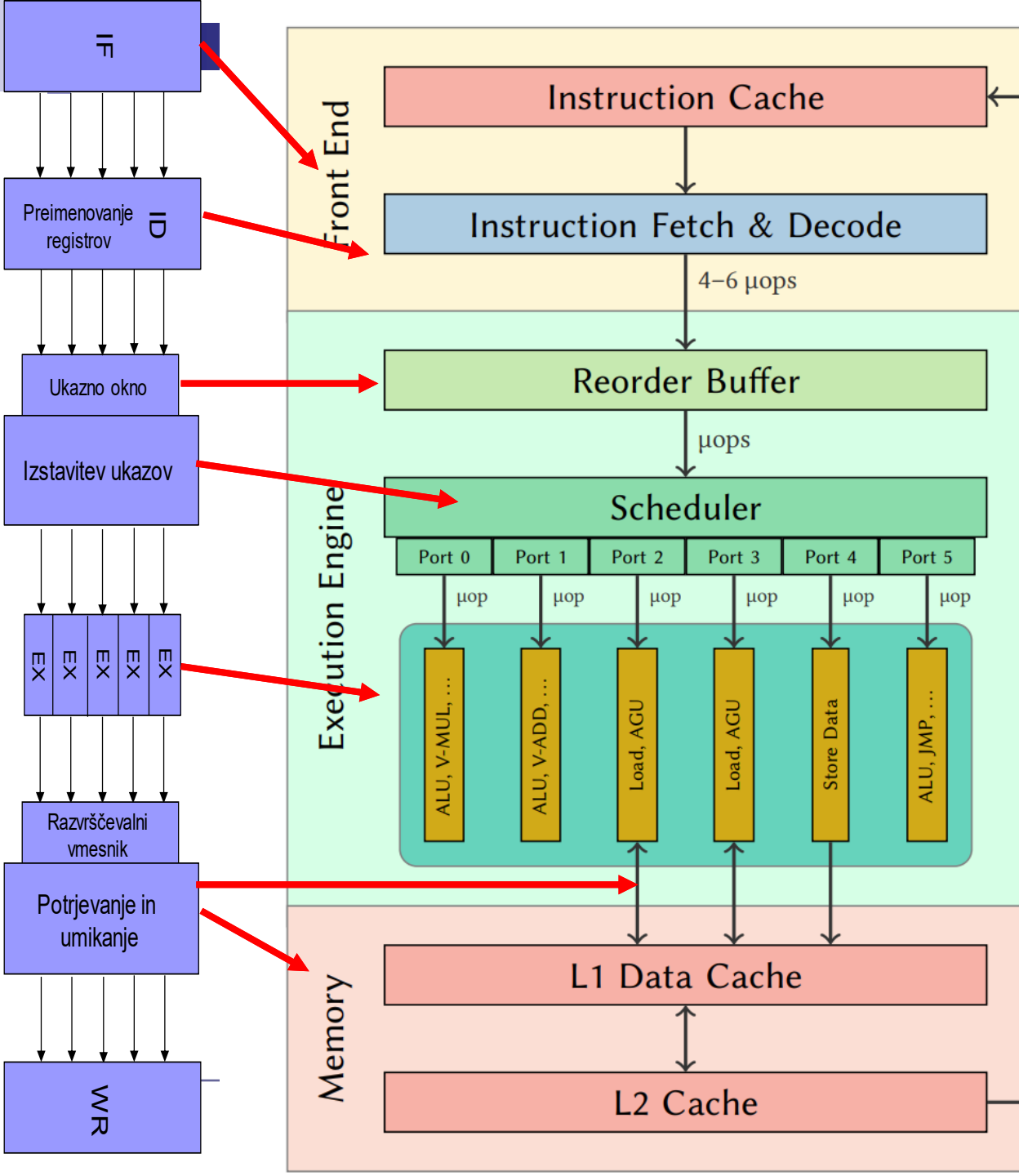
## Superscalar processor



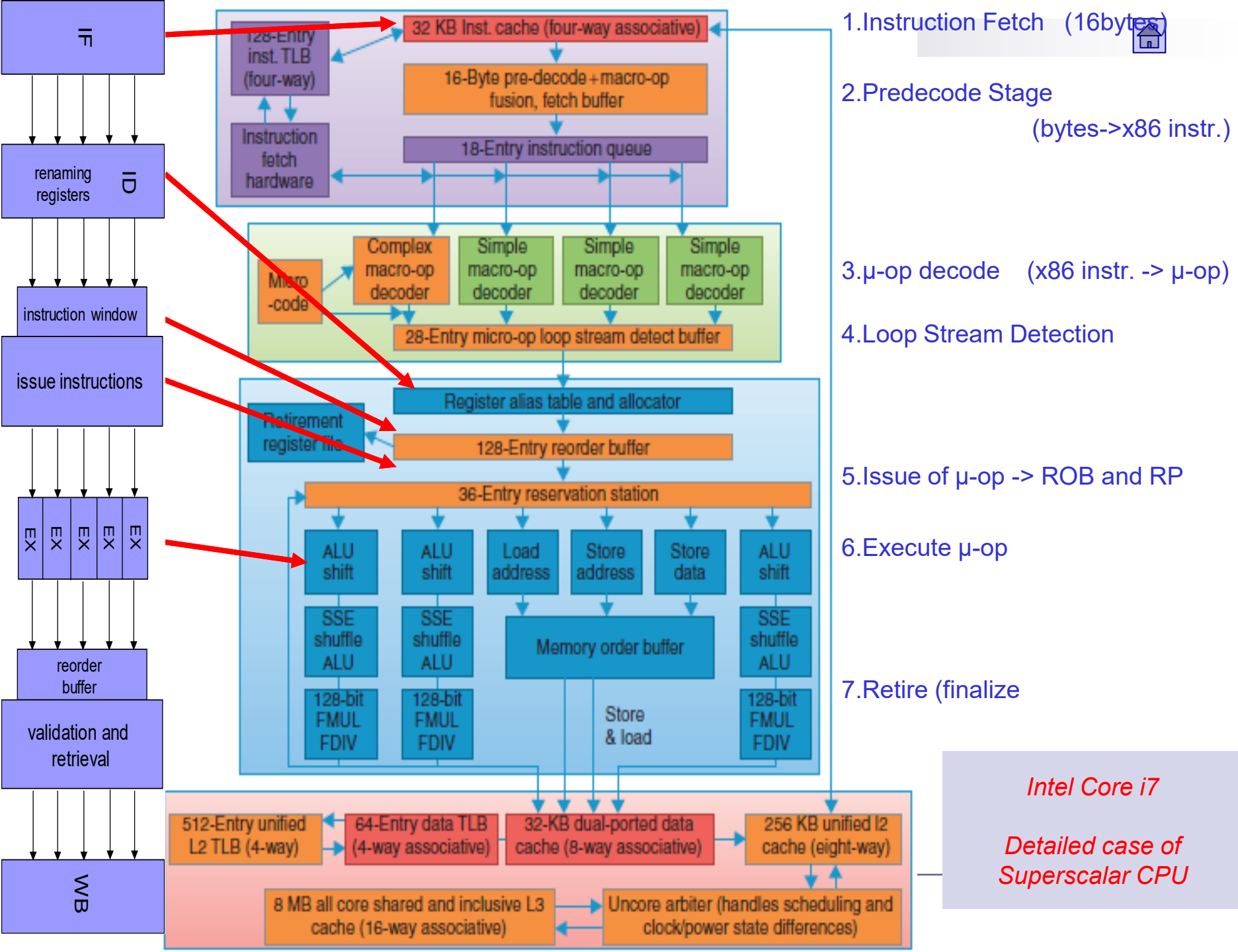
simplified scheme of superscalar processor  
based on 5-stage pipeline

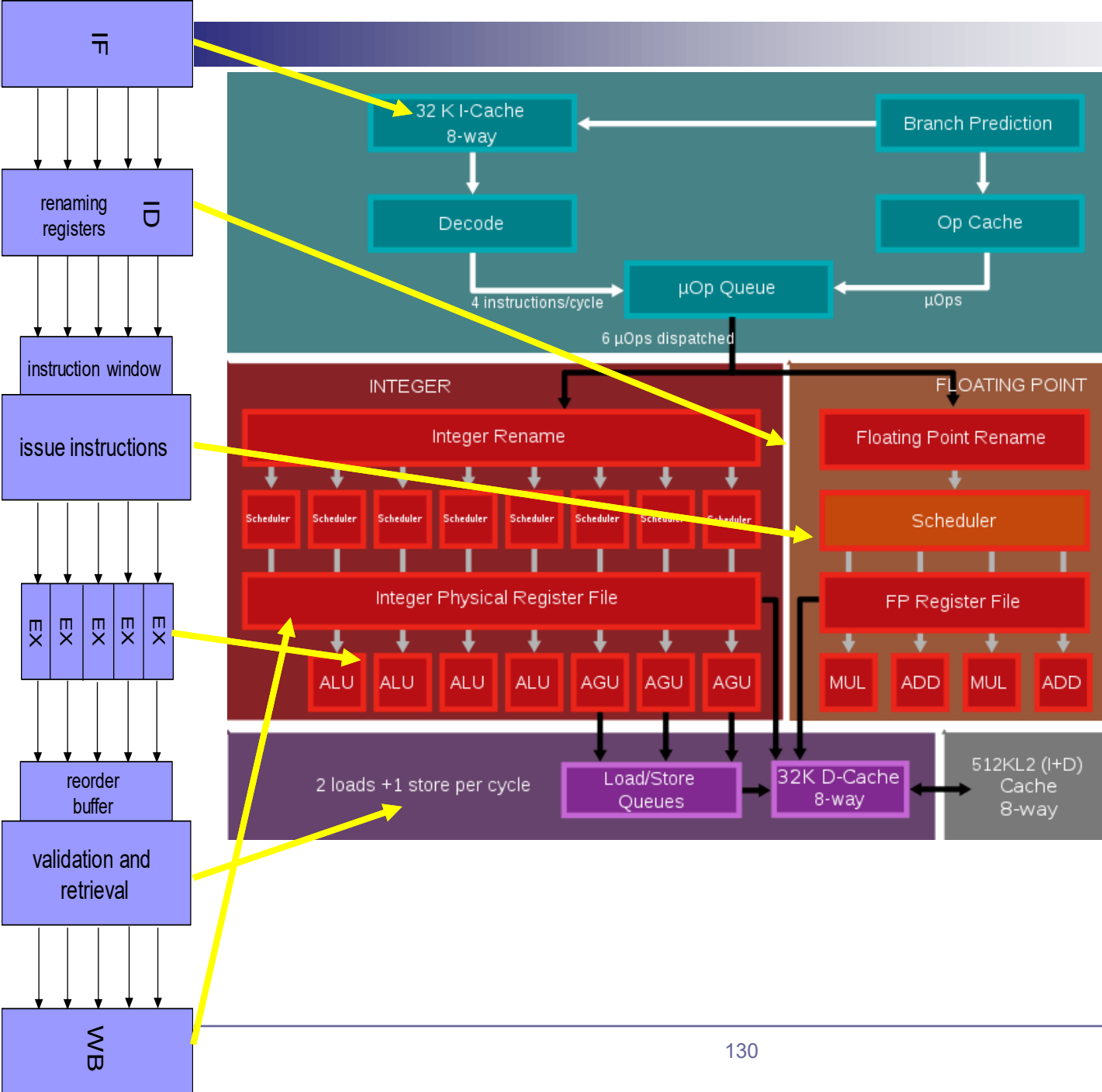
- One of the functional units in the EX stage is also stage MA (combined functional unit LOAD/STORE or separate functional units for LOAD and STORE).

*Simplified case of Superscalar CPU: Intel Core i7*



1. Instruction Fetch (16bytes)
2. Predecode Stage (bytes->x86 instr.)
3.  $\mu$ -op decode (x86 instr. ->  $\mu$ -op)
4. Loop Stream Detection
5. Issue  $\mu$ -op -> ROB in RP
6. Execute  $\mu$ -op
7. Retire (finalize)





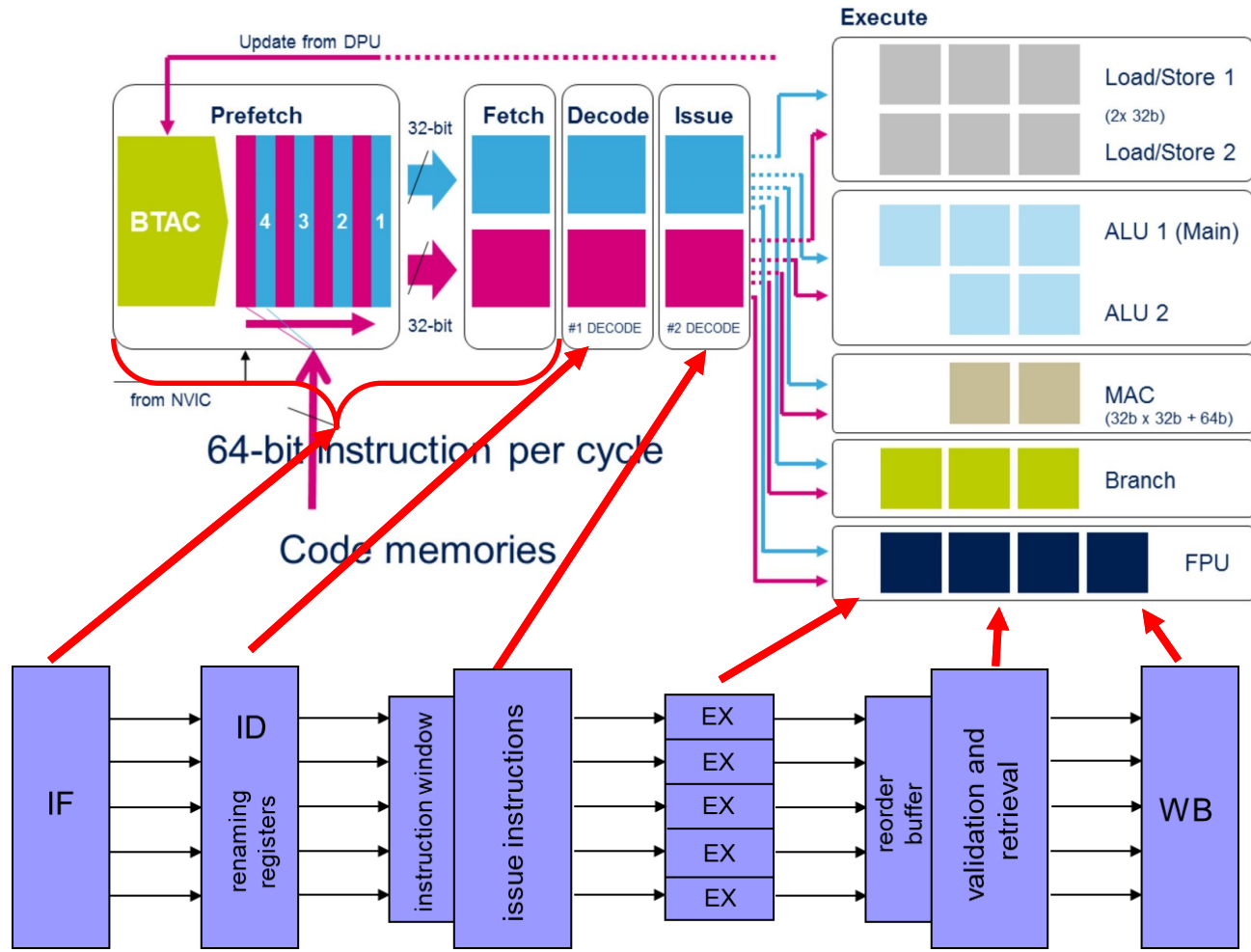
**AMD Zen 2**  
*Detailed case of  
superscalar CPU*



# ARM Cortex-M7 → Dual-issue



*ARM Cortex M7*  
*Case of dual-issue simpler pipeline*



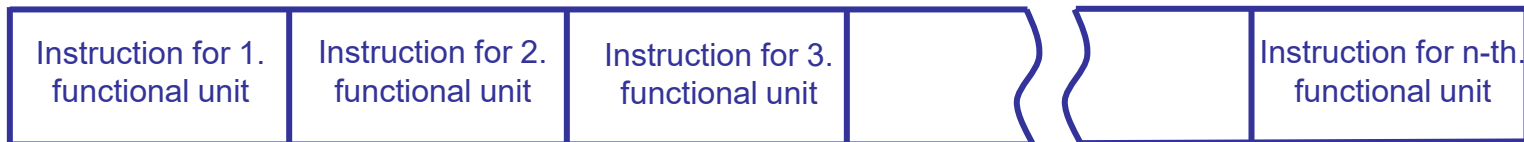




**VLIW (Very Long Instruction Word)** Processors are executing long instructions, which consist of several ordinary machine instructions that are executed in parallel by a processor using variety of functional units.

- In the long instruction, each unit executes its own instruction.

VLIW instruction consists of instructions for each functional unit



Case of VLIW instruction composition:





- Compiler is looking in program for mutually independent instructions, that can be executed in parallel in functional units, and merges them in long instructions.
- Number of instructions, which are fetched and issued in one clock period is determined by the compiler and is not changed during the execution (static decision).
- If the compiler can not find enough instructions for all functional units in long instruction, missing instructions are replaced by the instruction NOP (No OPeration).



# VLIW processor

Compiler finds independent instructions corresponding to functional units and creates „long instructions words“.

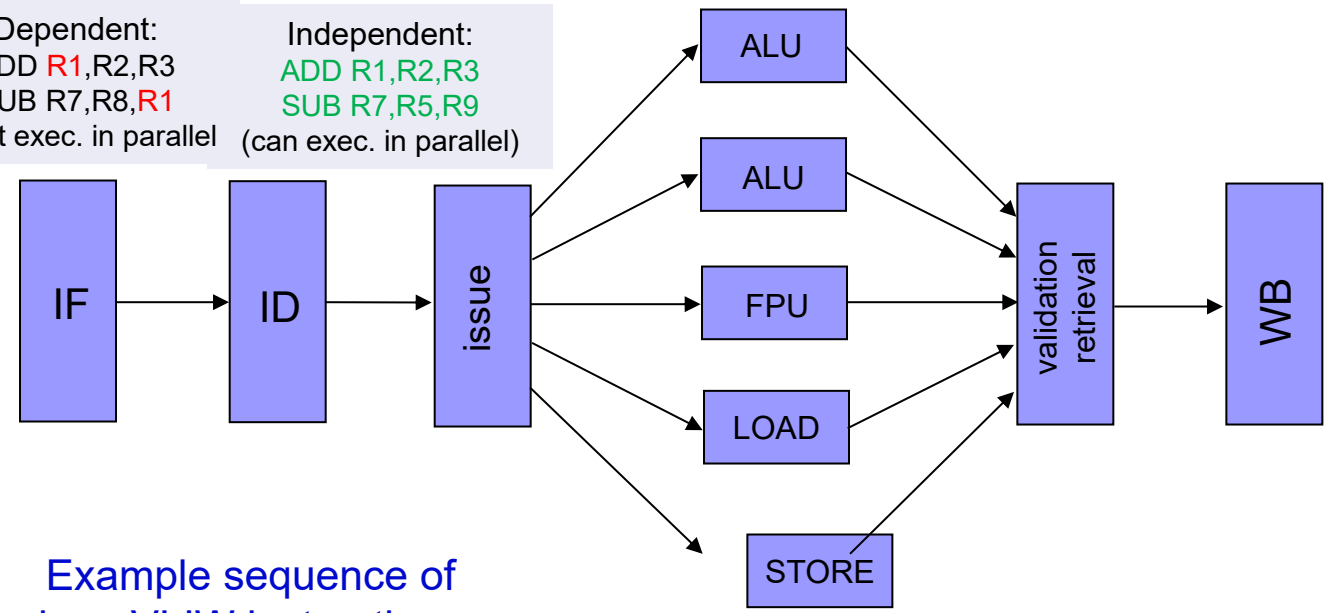
## Program

If corresponding and independent instruction is not found, NOP is inserted („-“ in VLIW instructions below).

LOAD ...  
 ---  
 ADD ...  
 FPADD ...  
 LOAD ...  
 ---  
 ADD ...  
 FPADD ...  
 LOAD ...  
 ---  
 ADD ...  
 ADD ...  
 FPADD ...  
 LOAD ...  
 STORE ...

Dependent:  
 ADD R1,R2,R3  
 SUB R7,R8,R1  
 (can't exec. in parallel)

Independent:  
 ADD R1,R2,R3  
 SUB R7,R5,R9  
 (can exec. in parallel)



Example sequence of long VLIW instructions



VLIW instruction

- NOP instruction

A = ALU instruction  
 F = FPU instruction  
 L = LOAD instruction  
 S = STORE instruction



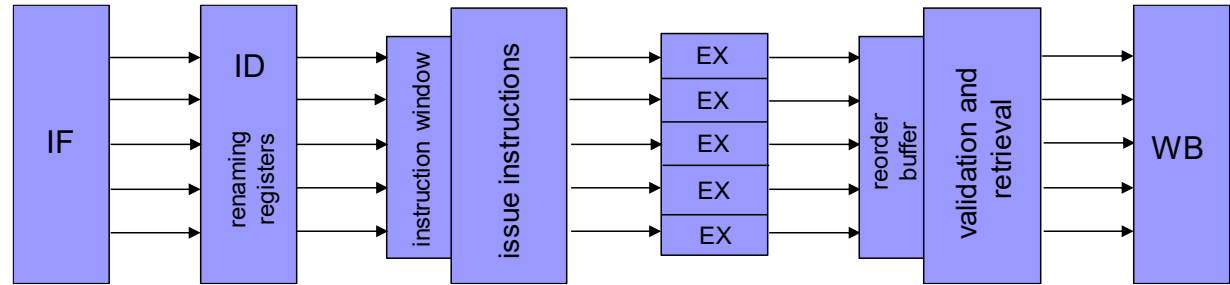
# Comparison: Superscalar vs. VLIW processor

## Superscalar processor

- Dynamic acquisition of several instructions (CPU decides during the execution)
- Complex realization

*more instructions at once*

LOAD ...  
 ADD ...  
 FPADD ...  
 LOAD ...  
 ADD ...  
 FPADD ...  
 LOAD ...  
 ADD ...  
 ADD ...  
 FPADD ...  
 LOAD ...  
 STORE ...



## VLIW processor

CPU – dynamical decisions

- Static schedule in long instructions (compiler decides before the execution)
- Simpler realization

*VLong Instr. Word (several shorter instr.)*

LOAD ...  
 ADD ...  
 FPADD ...  
 LOAD ...  
 ADD ...  
 FPADD ...  
 LOAD ...  
 ADD ...  
 ADD ...  
 FPADD ...  
 LOAD ...  
 STORE ...

