ITMO

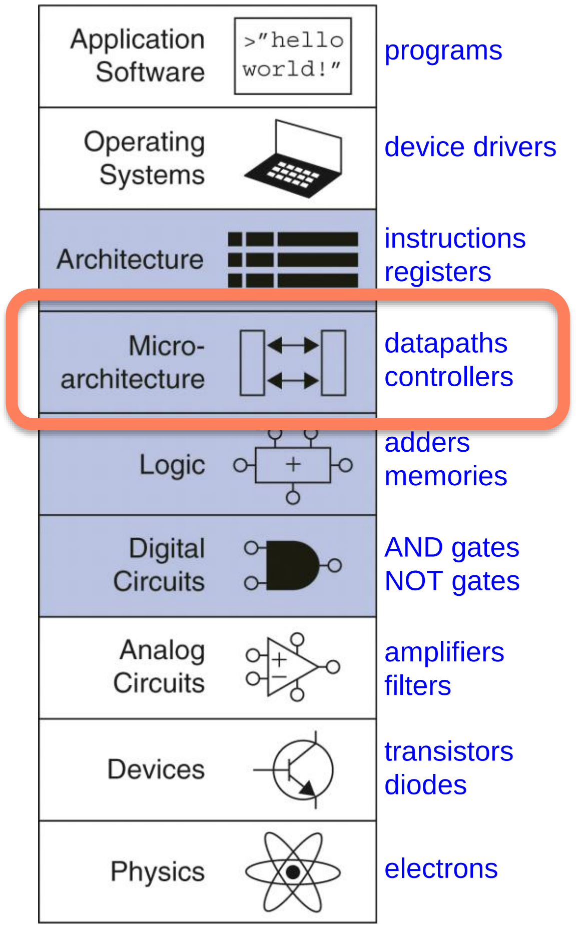
Computer Systems Design
Lesson 9
Basic microarchitectural templates.
Multi-cycle and pipelined processors.

Alexander Antonov, Assoc. Prof., ITMO University

Hangzhou, 2025

Outline the lesson

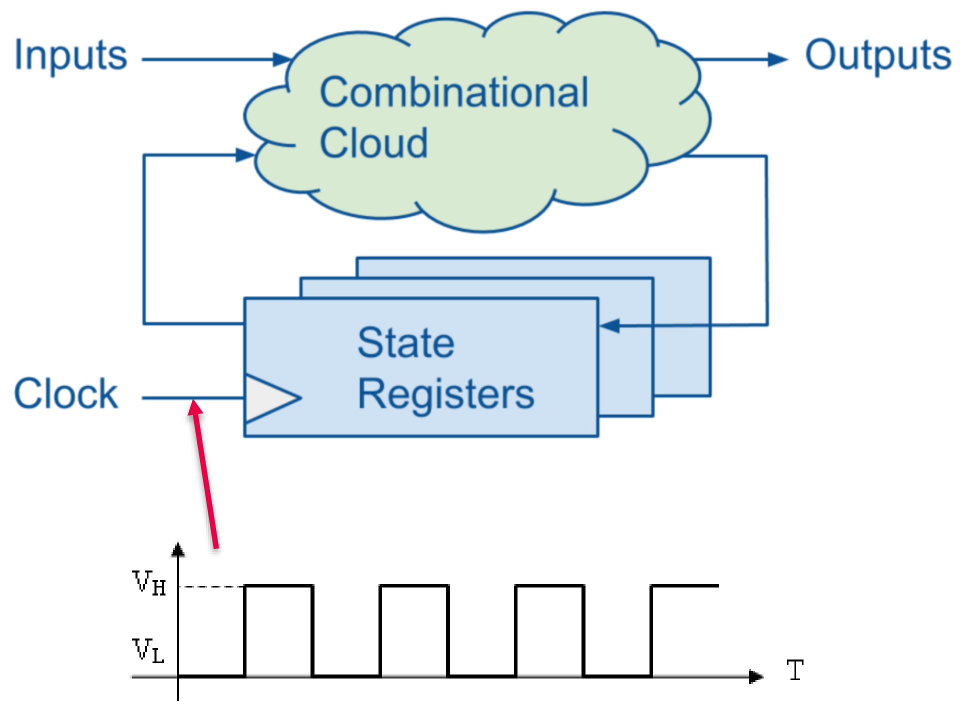
- Concept of single-cycle implementation of digital circuits
- Concept of multi-cycle implementation of digital circuits
- Concept of pipelined implementation of digital circuits
- PPA metrics: performance, power, area
- Analysis and comparison of implementations
- Achieving PPA trade-off



Synchronous design – Huffman model

Synchronous circuit – common template for digital circuits

Adds necessary predictability to timing correlations (trading potential efficiency)



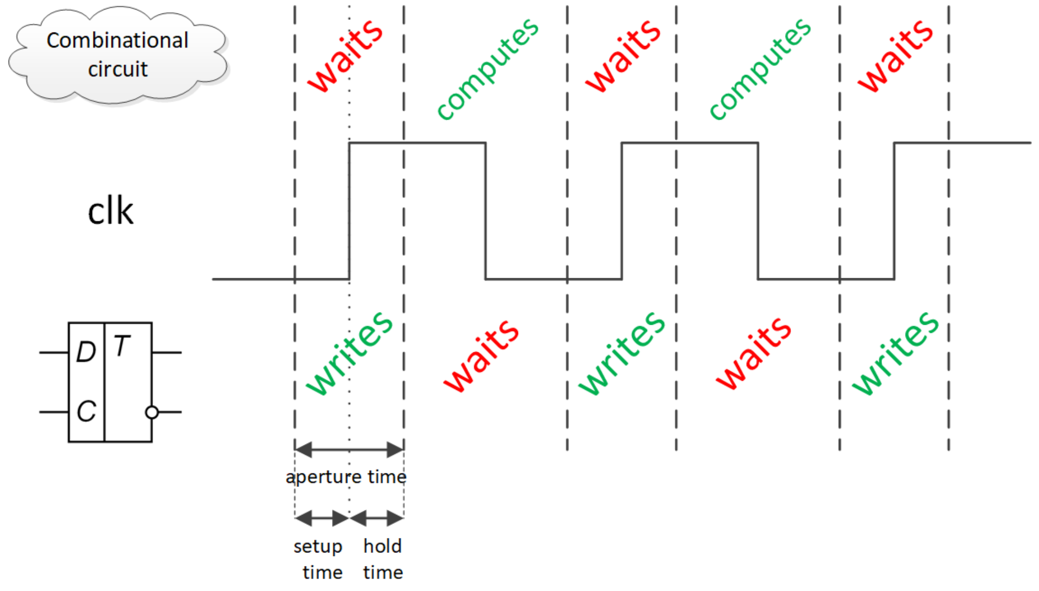
Consists of intermittent:

- ***combinational logic***
- ***flip-flops*** (clocked, edge-triggered registers and memories)

Synchronous design – operation in time

Operation of synchronous design in time is intermittent phases:

- ***computing***: combinational circuit operation, registers are insensitive to input
- ***fixating result***: data is written to registers, combinational circuits must output stable signal during the aperture time (setup time + clock edge + hold time)



Synchronous design statically fixes the <u>phase difference ranges</u> of data signals relative to clock signal

Single-cycle (combinational) implementation

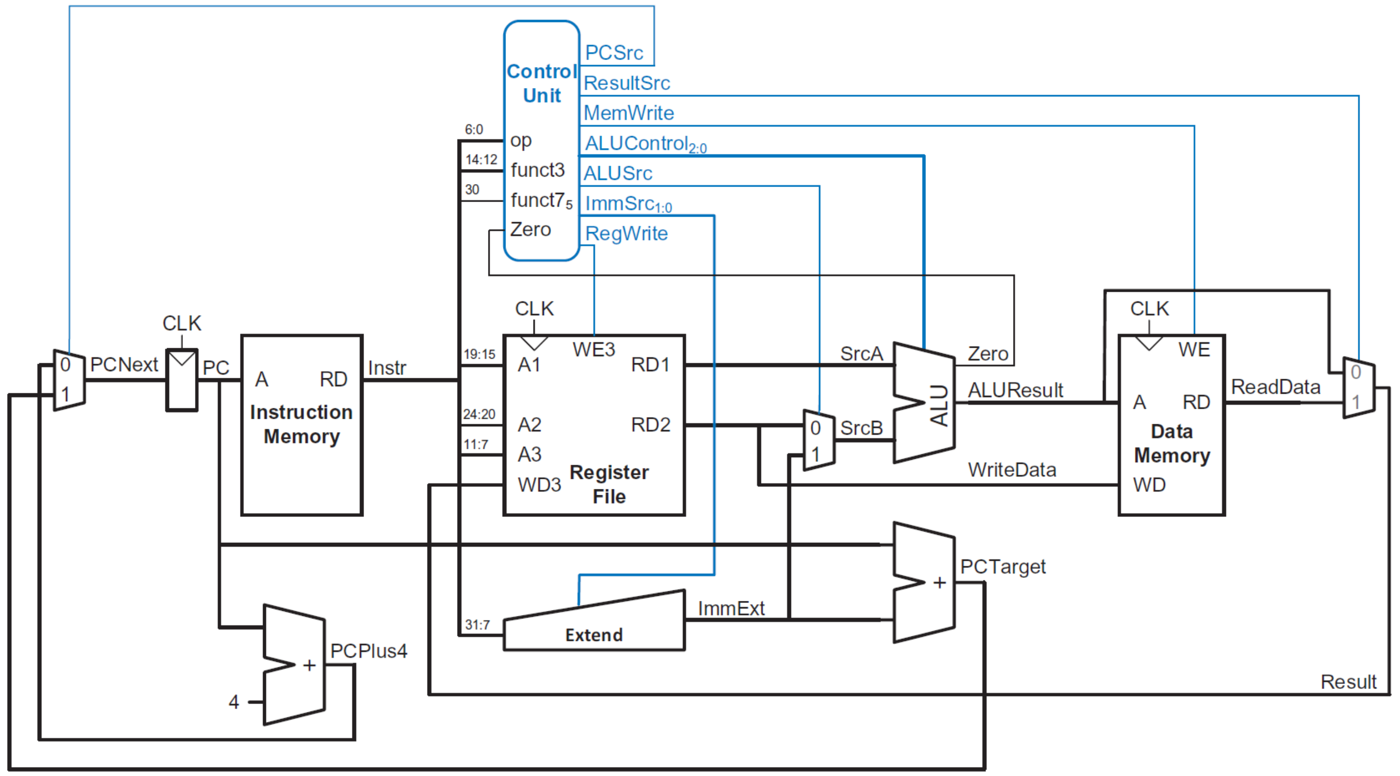
Single-cycle implementation – implements transactions (instructions, memory requests, I/O transactions) in a single clock cycle

Consists in following decisions:

- All states are synchronized by a special signal (synchronization signal)
- Execution of all transactions is done within the same time (single clock cycle)
- Execution of all transactions involves the same datapath

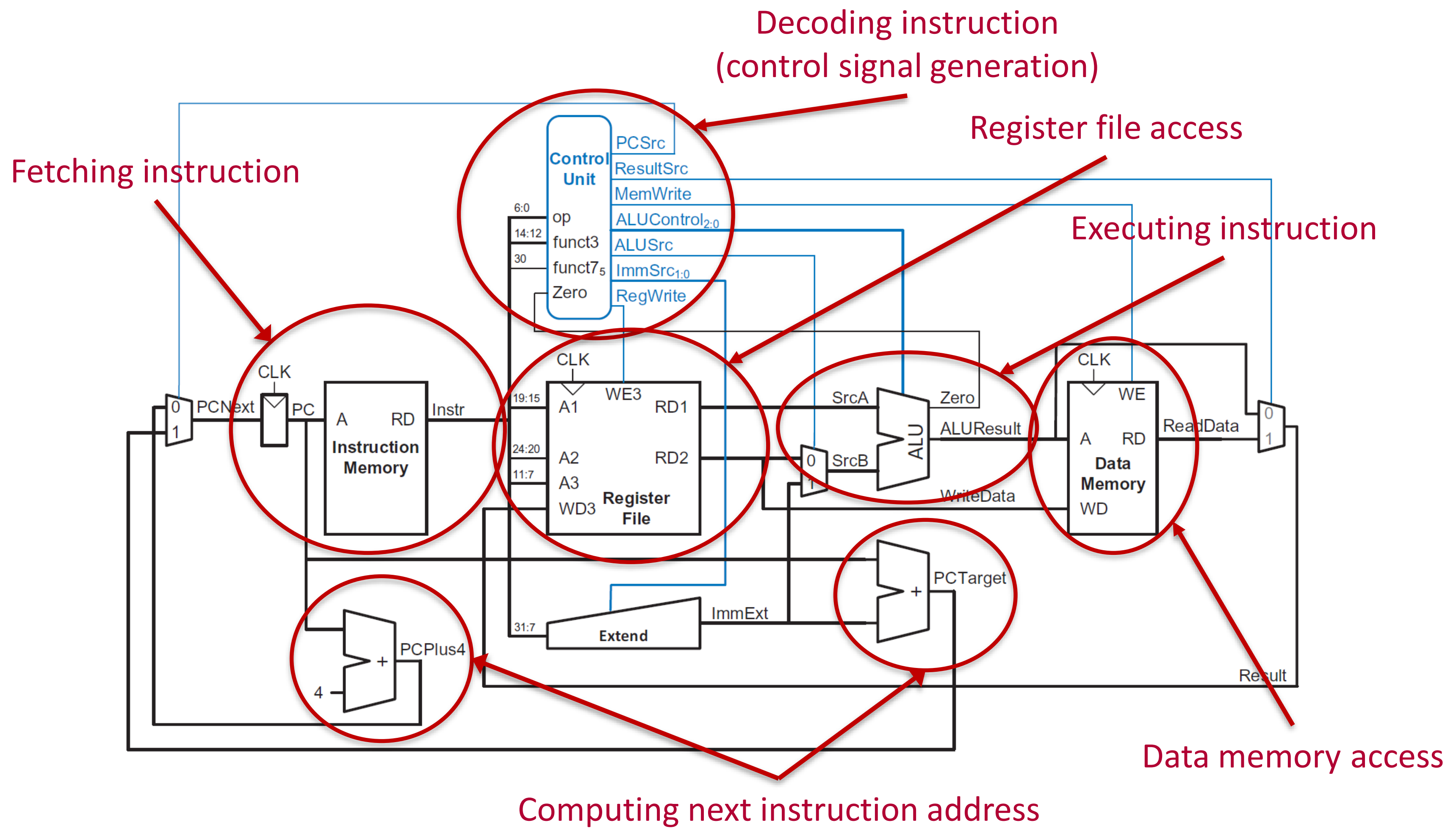
				single-cycle					
cycle	0			1			2		
trx	0			1			2		
ops	🟩 op0	🟩 op1	🟧 op2	🟧 op0	🟩 op1	🟧 op2	🟧 op0	🟩 op1	🟩 op2

Example: typical single-cycle RISC processor



Picture: Sarah Harris, David Harris. Digital Design and Computer Architecture, RISC-V Edition. Morgan Kaufmann, 2021

Example: typical single-cycle RISC processor



Multi-cycle implementation

Multi-cycle implementation – dividing one long cycle of single cycle (combinational) implementation with **several faster cycles**, such as each new transaction starts after completion of previous one

control steps, c-steps – execution steps for given operation synced to clock signal

Consists in following decisions:

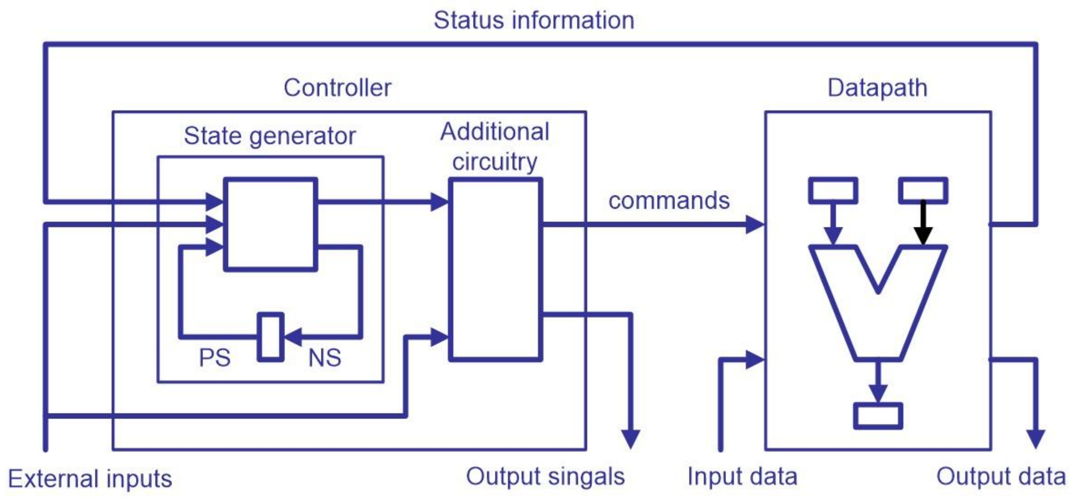
- Splitting computation into a series of c-steps
- Allocation of minimum needed number of functional units (FUs) for each c-step
- Implementation of control logic (FSM) managing data transfers and FU activation

single-cycle

transactions	trx0			trx1			trx2		
clock cycles	0			1			2		
ops	op0	op1	op2	op0	op1	op2	op0	op1	op2

multi-cycle

transactions	trx0			trx1			trx2		
clock cycles	0	1	2	3	4	5	6	7	8
ops	op0	op1	op2	op0	op1	op2	op0	op1	op2



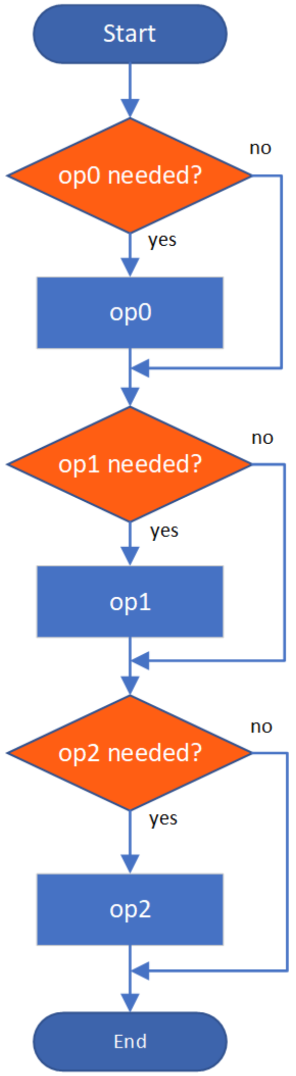
“Shortcuts” for non-uniform computations

Unused operations for certain transactions can be removed

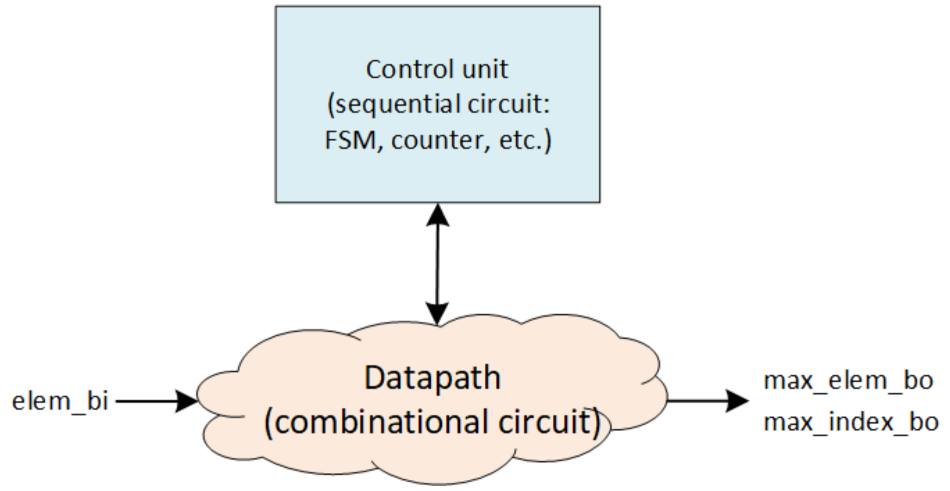
Can speed-up execution of ***non-uniform computations***
(e.g. complex CPU instructions)

single-cycle									
transactions	trx0			trx1			trx2		
clock cycles	0			1			2		
ops	op0	op1	op2	op0	op1	op2	op0	op1	op2

multi-cycle					
transactions	trx0		trx1	trx2	
clock cycles	0	1	2	3	4
ops	op0	op1	op1	op1	op2

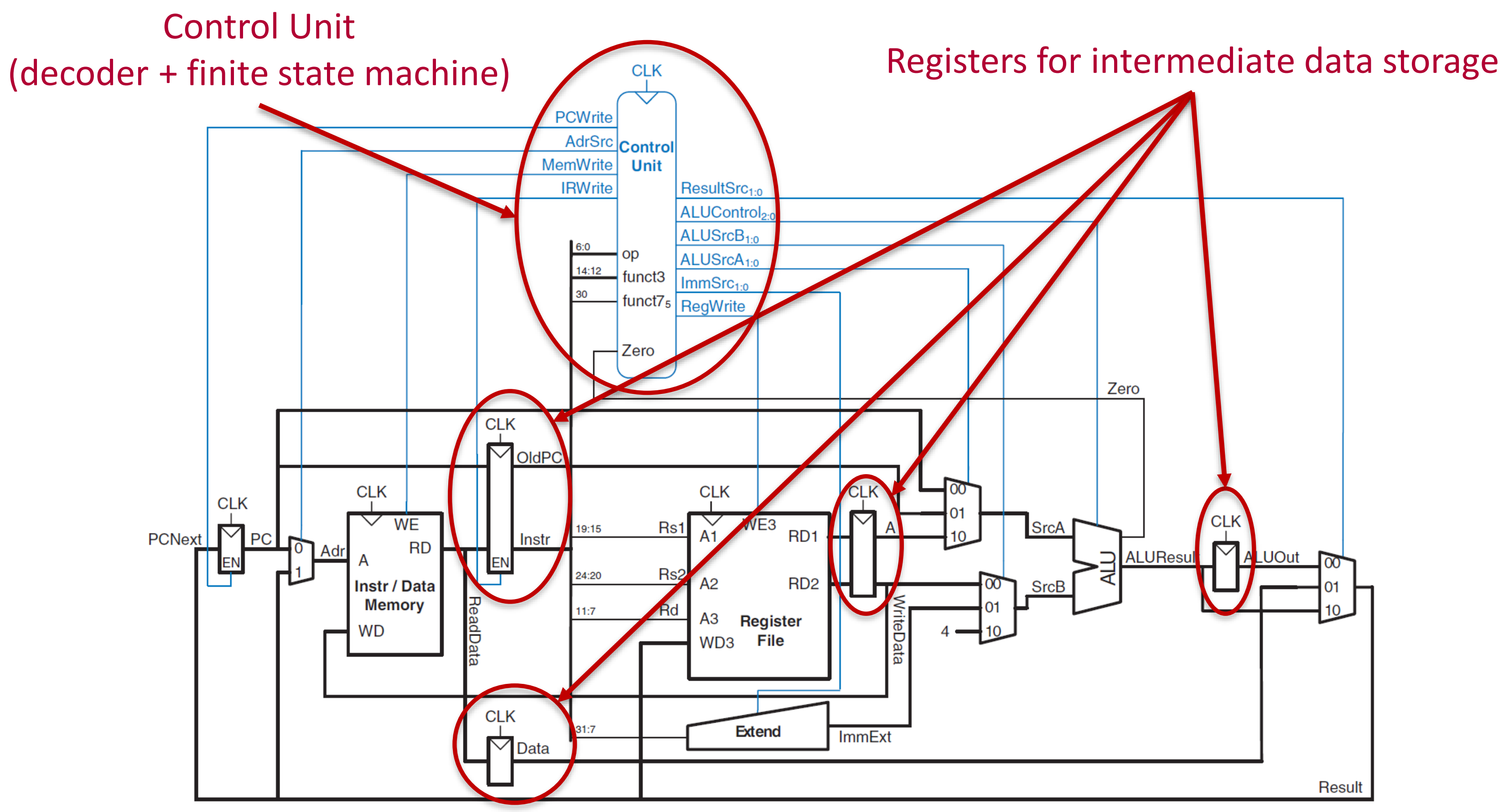


Example: finding maximum value in an array



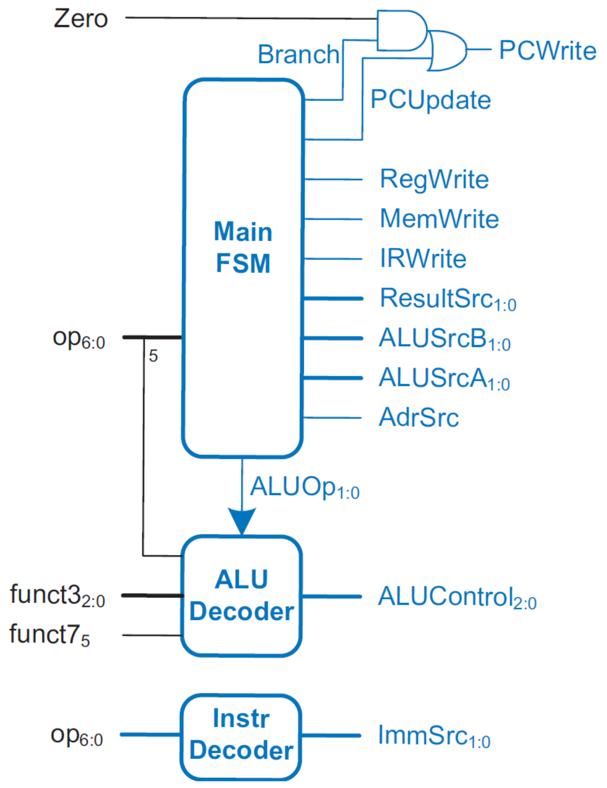
c-step number	operation
0	check 1st element
1	check 2nd element
2	check 3rd element
…	…
15	check 15th element

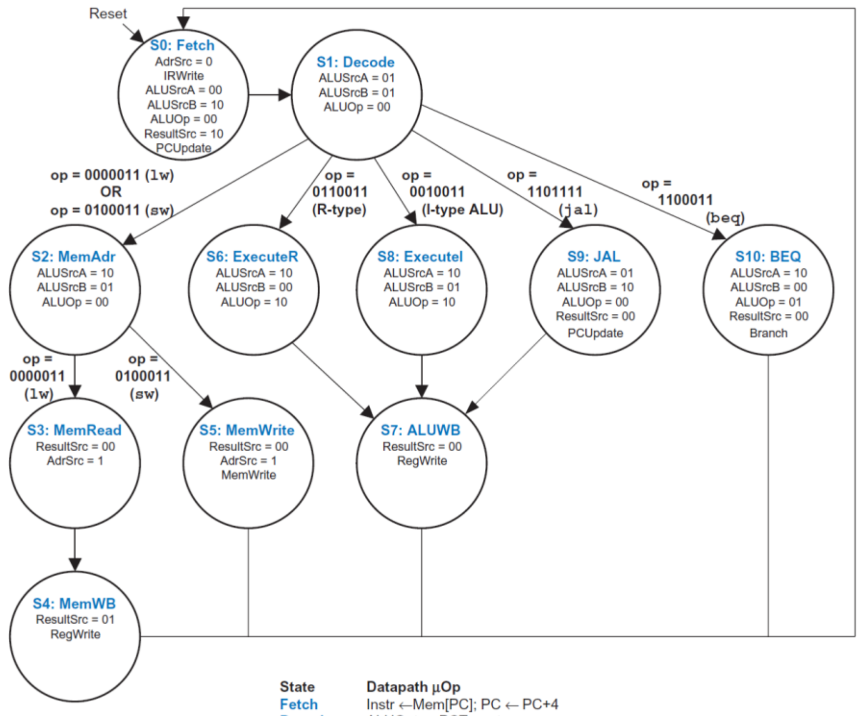
Example: typical multi-cycle RISC processor



Picture: Sarah Harris, David Harris. Digital Design and Computer Architecture, RISC-V Edition. Morgan Kaufmann, 2021

Multi-cycle RISC-V processor control unit





State	Datapath µOp
Fetch	Instr ←Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1oprs2
ExecuteI	ALUOut ← rs1opimm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4

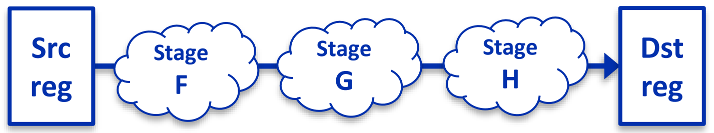
Picture: Sarah Harris, David Harris. Digital Design and Computer Architecture, RISC-V Edition. Morgan Kaufmann, 2021

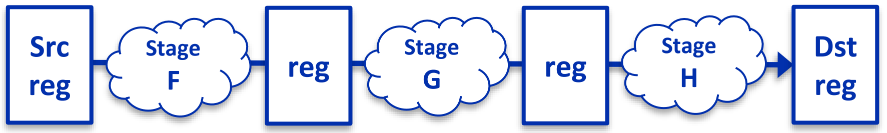
Pipelined implementation

Pipelined implementation - technique allowing fold improvement of a digital circuit throughput without fold replication of digital logic.

Consists in following decisions:

- The datapath is split into a sequence of stages
  - *the delay of each stage is smaller than the delay of the entire initial datapath*
  - *registers inserted between stages ("pipeline registers")*
- Processing of a new request starts as soon as the first stage is ready to accept it
  - *processing of consecutive requests* ***overlap*** *in time (temporal parallelism)*





$$BW_{peak(a)} = Fmax_{a} = \frac{1}{t_{FGH}},$$

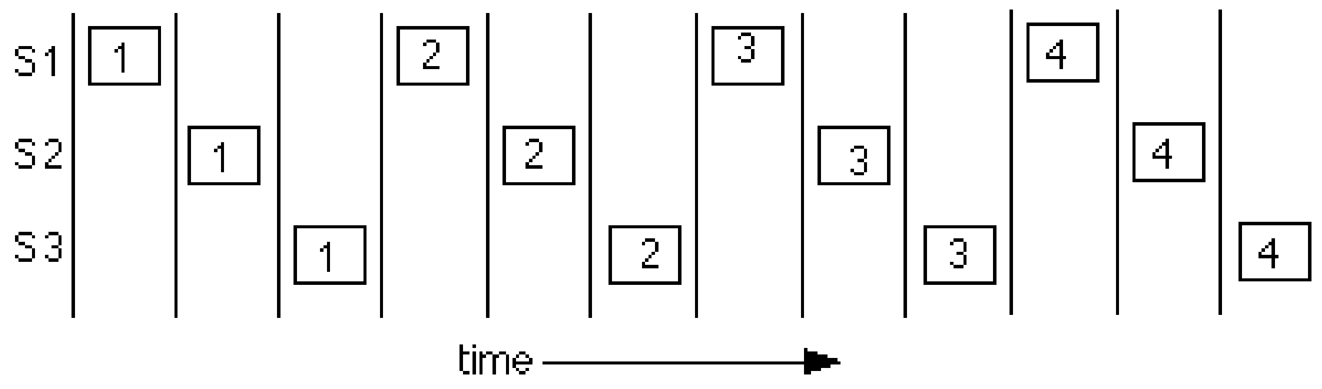
$$BW_{peak(б)} = Fmax_{б} = \frac{1}{\max(t_F, t_G, t_H)},$$

$$\max(t_F, t_G, t_H) < t_{FGH} \Rightarrow BW_{peak(б)} > BW_{peak(a)}$$

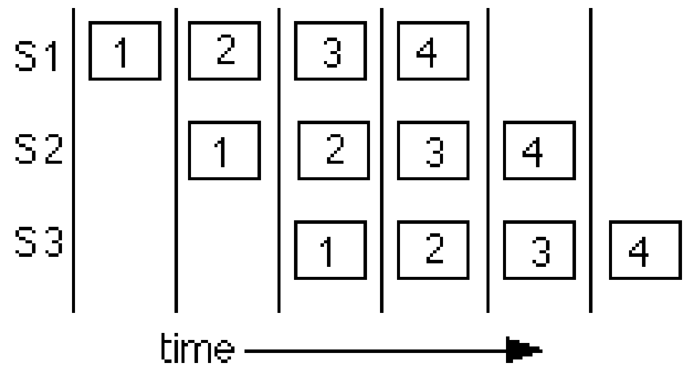
All modern high-performance designs (processors, interconnects, memories, etc.) are pipelined

Pipeline operation in time

- Multi-cycle implementation



- Pipelined implementation



Pipeline idealisms

Pipeline idealisms* – properties of computational traffic and design that maximize gain from pipelining

- Uniform-delay subcomputations

  *otherwise - **internal fragmentation**, "slow" stages slowdown the entire pipeline,*
  non-fold increase of throughput

- Uniform types of requests

  *otherwise - **external fragmentation**, under-utilization of stages, decrease of potential performance*

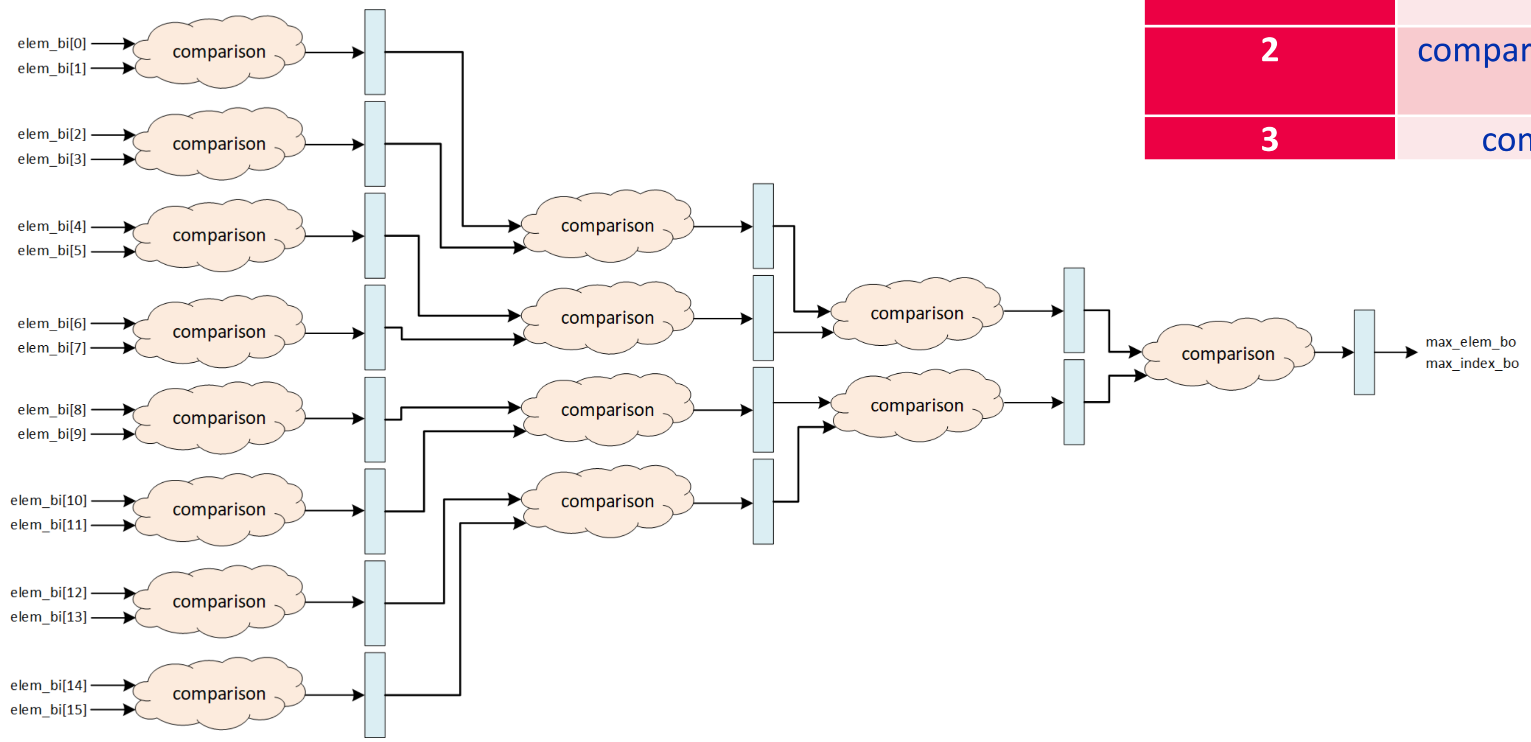
- Independency of requests (no feedback dependencies in processing algorithm)

  *otherwise - **pipeline interlocks**, **stalls** while waiting for data*

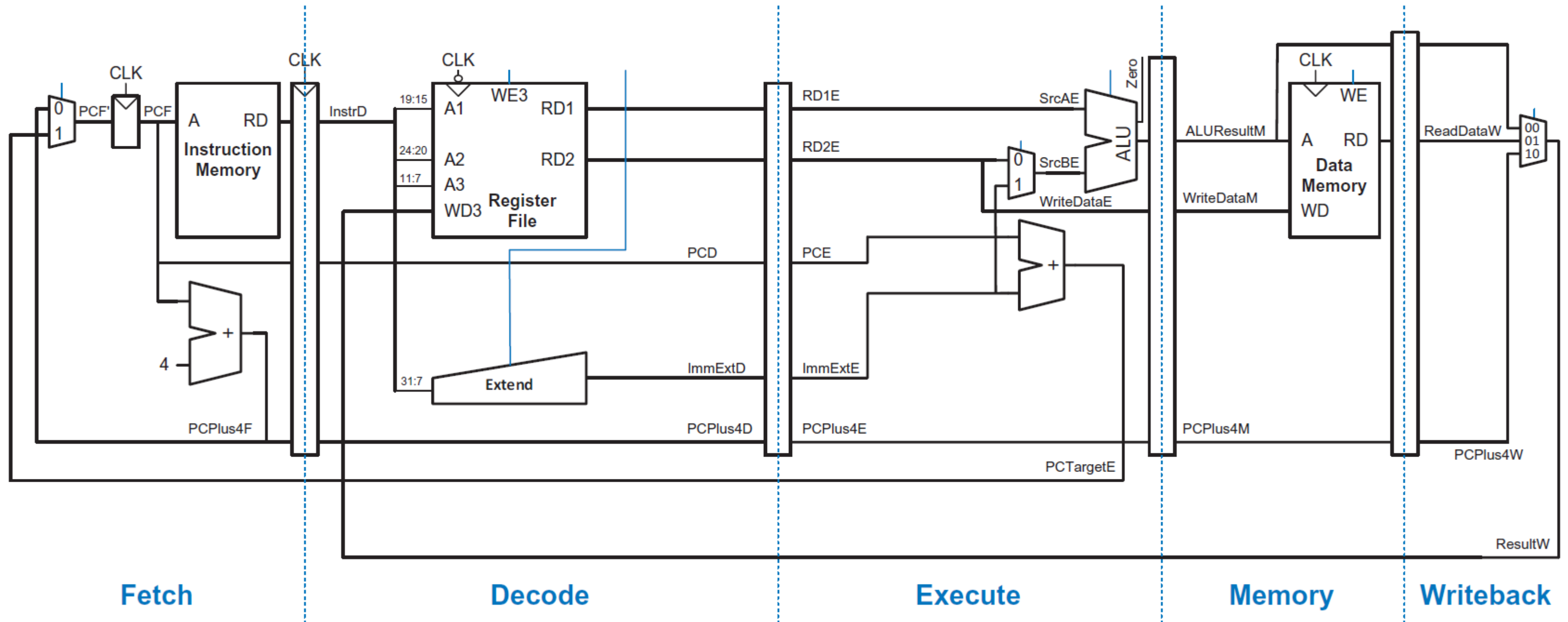
Shen, J. P., & Lipasti, M. H. Modern processor design: fundamentals of superscalar processors. Waveland Press, 2013

Example: finding maximum value in an array

c-step number	operation
0	compare elements in pairs: 0-1; 2-3; 4-5; 6-7; 8-9; 10-11; 12-13; 14-15
1	compare pairing results from stage 0 in pairs: 0-1; 2-3; 4-5; 6-7
2	compare pairing results from stage 1 in pairs: 0-1; 2-3
3	compare pairing results from stage 2



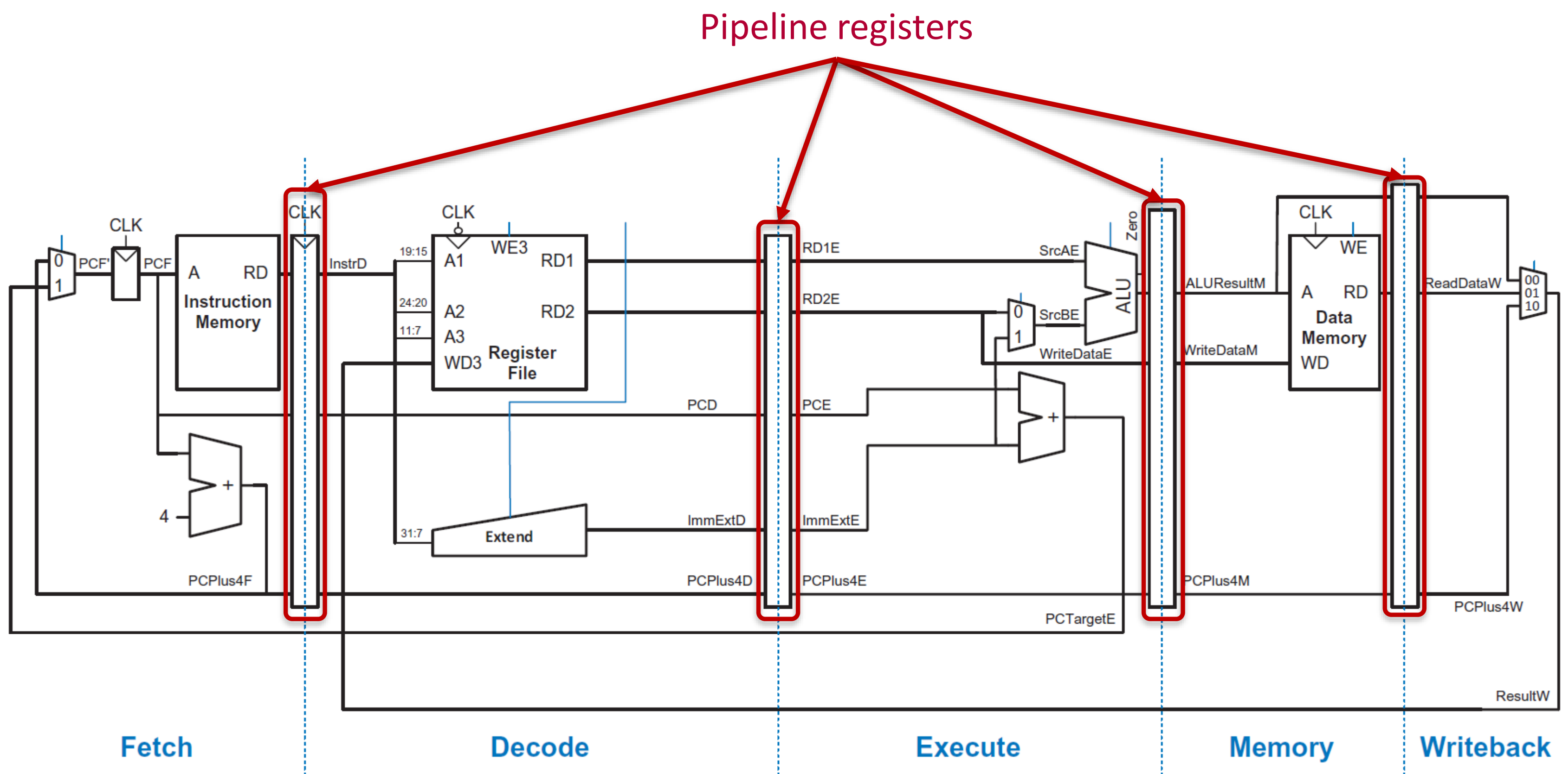
Example: classic RISC pipeline



Picture: Sarah Harris, David Harris. Digital Design and Computer Architecture, RISC-V Edition. Morgan Kaufmann, 2021

ITMO

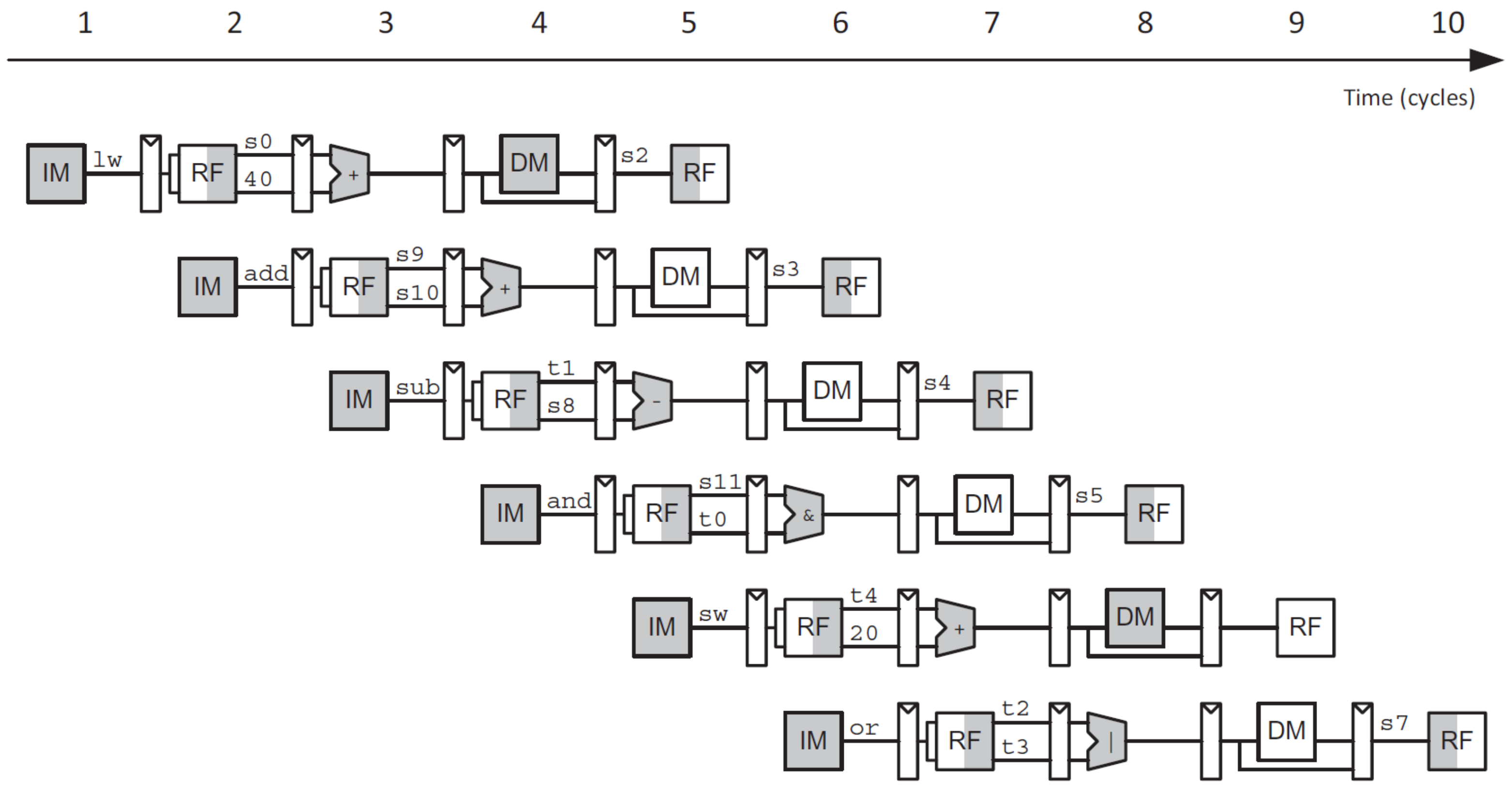
Example: classic RISC pipeline



Picture: Sarah Harris, David Harris. Digital Design and Computer Architecture, RISC-V Edition. Morgan Kaufmann, 2021

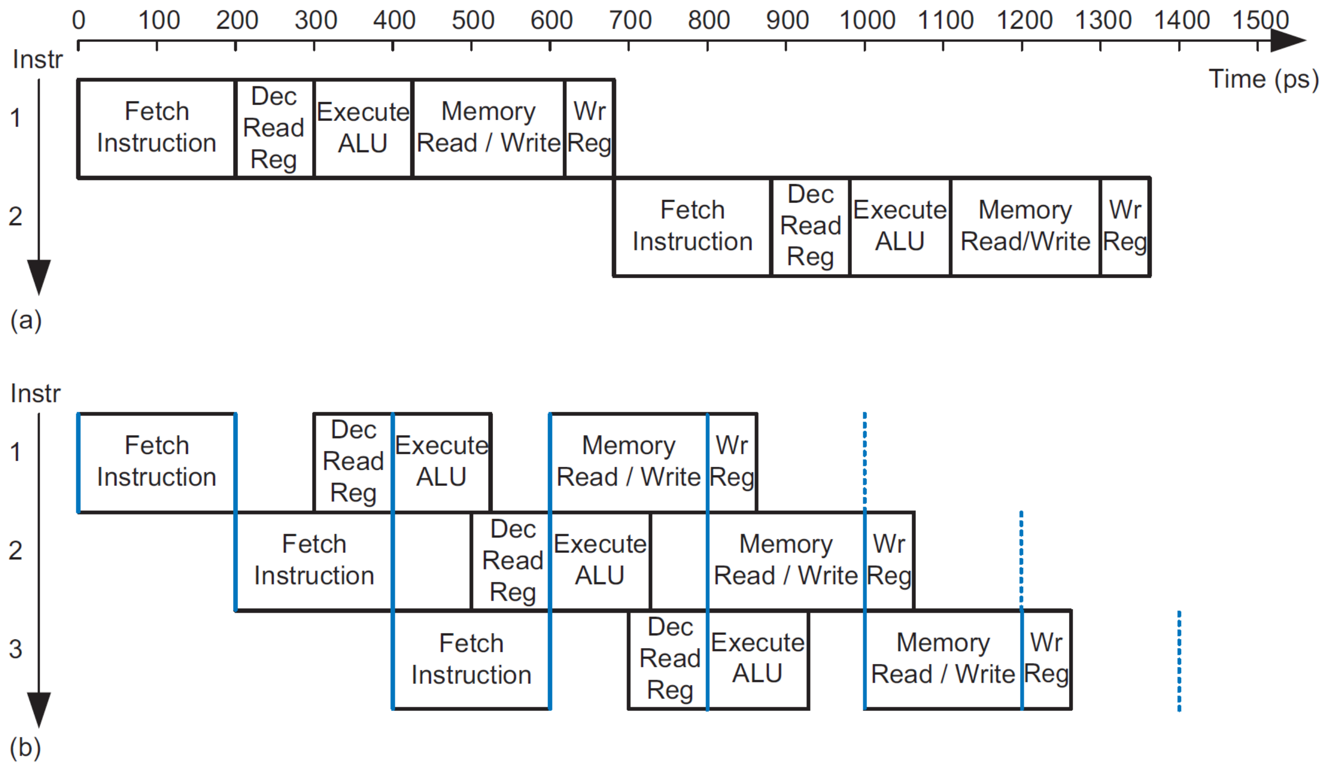
Example: classic RISC pipeline executing program

```
lw   s2, 40(s0)
add  s3, s9, s10
sub  s4, t1, s8
and  s5, s11, t0
sw   s6, 20(t4)
or   s7, t2, t3
```



ITMO

Example: single-cycle vs. pipelined RISC in time



Performance is better in pipelined implementation due to parallelism
However: clock frequency is limited by the longest stage

Pipeline hazards: concept

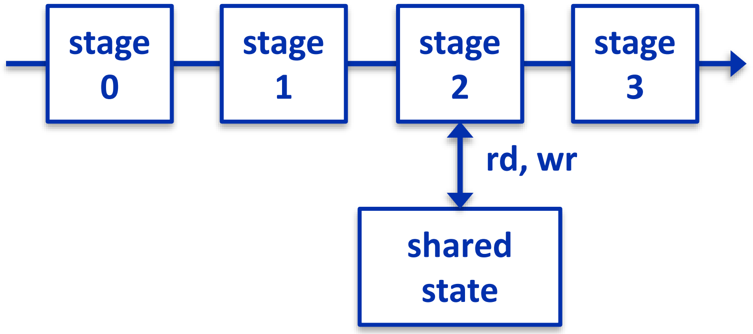
Problem of sequential → parallel instructions execution:

Changed ordering of working with shared states

Example: instruction flow

```
trx0: shared_state_v1 ← read (shared_state_v0) + 4;
trx1: shared_state_v2 ← read (shared_state_v1) + 8;
```

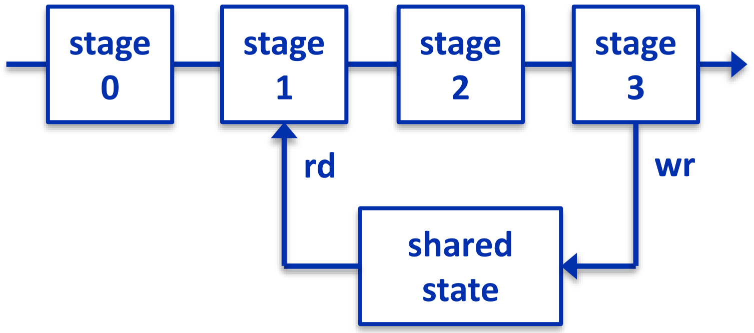
Expected result: `shared_state_v2 ← shared_state_v0 + 12;`



Operations sequence in hardware:

```
trx0: read (shared_state_v0);
trx0: shared_state_v1 ← read (shared_state_v0) + 4;
trx1: read (shared_state_v1);
trx1: shared_state_v2 ← read (shared_state_v1) + 8;
```

Result: shared_state_v0 + 12 (OK)



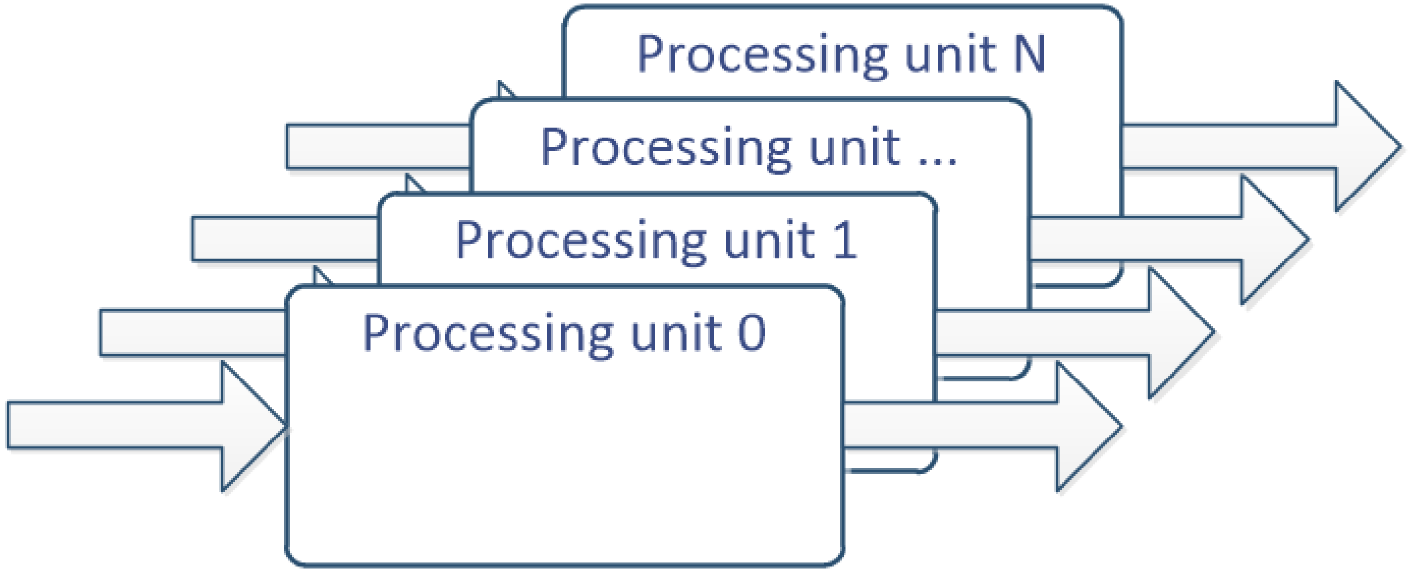
Operations sequence in hardware:

```
trx0: read (shared_state_v0);
trx1: read (shared_state_v0); // outdated, trx0 hasn't completed
trx0: shared_state_v1 ← read (shared_state_v0) + 4;
trx1: shared_state_v2 ← read (shared_state_v0) + 8;
```

Result: shared_state_v0 + 8 (not OK)

Spatial parallelism

- Allows to increase throughput with linear increase of HW resources and power
- Latency remains unchanged



Basic performance metrics

Basic metrics: PPA (Performance, Power, Area). Basic performance metrics: throughput and latency

- ***Throughput (bandwidth)*** – *number of requests* processed within time period
  - *II (Initiation Interval) – inverse characteristic*
- ***Latency*** – *time passed* from request processing initiation to completion
  - *Parallelism is not taken into account!*

Time is evaluated in two variants:

- ***Clock cycles***: c-steps
  - *Fully defined by RTL design*
  - *Will not change for various FPGA/ASIC implementations*
- ***Real time***: seconds
  - *Depends on achieved clock frequency*
  - *Specific to certain FPGA/ASIC implementation*

Combinational implementation: $\boldsymbol{II = 1;\ BW = \frac{1}{Lat}}$

Multi-cycle implementation: $\boldsymbol{II > 1;\ BW \sim \frac{1}{II} \sim \frac{1}{Lat}}$

Pipelined implementation: $\boldsymbol{II \to 1; BW_{peak} = \frac{1}{Lat} * N_{stages}}$

Performance of implementations

Maximum frequency:

- ***Combinational*** – worst (all computations – within one clock cycle)
- ***Multi-cycle*** – slightly worst than pipelined (auxiliary control logic)
- ***Pipelined*** – best (only a single stage within one clock cycle)

Throughput:

- ***Combinational*** – worst (low frequency, no parallelism)
- ***Multi-cycle*** – better that combinational for non-uniform traffic, slightly worse for uniform (extra control logic and buffers, lower frequency)
- ***Pipelined*** – best (high frequency, initiation interval = 1)

Latency:

- ***Combinational*** – best for uniform traffic (no buffers and control logic)
- ***Multi-cycle*** – better than combinational for non-uniform traffic, slightly worse for uniform (extra control logic and buffers, lower frequency)
- ***Pipelined*** – slightly better than multi-cycle (less control logic), but can suffer from external fragmentation

Resources consumption of implementations

Hardware resources:

- ***Combinational*** – not good (each subcomputation requires dedicated resource)
- ***Multi-cycle*** – best (HW can be reused for subcomputations, but requires extra buffers)
- ***Pipelined*** – worst (each subcomputation requires dedicated resource, extra buffers)

Power:

$$P_{total} = P_{static} + P_{dynamic}; \quad P_{static} = V * I_{leakage}; \quad P_{dynamic} = Q * f * C * V^2,$$

- ***Combinational*** – (typically) high static, low dynamic
- ***Multi-cycle*** – (typically) low static, high dynamic
- ***Pipelined*** – (typically) high static, high dynamic

Choosing implementation is a non-trivial optimization problem!

Exhaustive design space exploration can be needed

ITMO

Thank you for the lesson!

Alexander Antonov, Assoc. Prof., antonov@itmo.ru

Hangzhou, 2025