Computer Architecture
Improving performance

http://d3s.mff.cuni.cz/teaching/nswi143

Lubomír Bulej

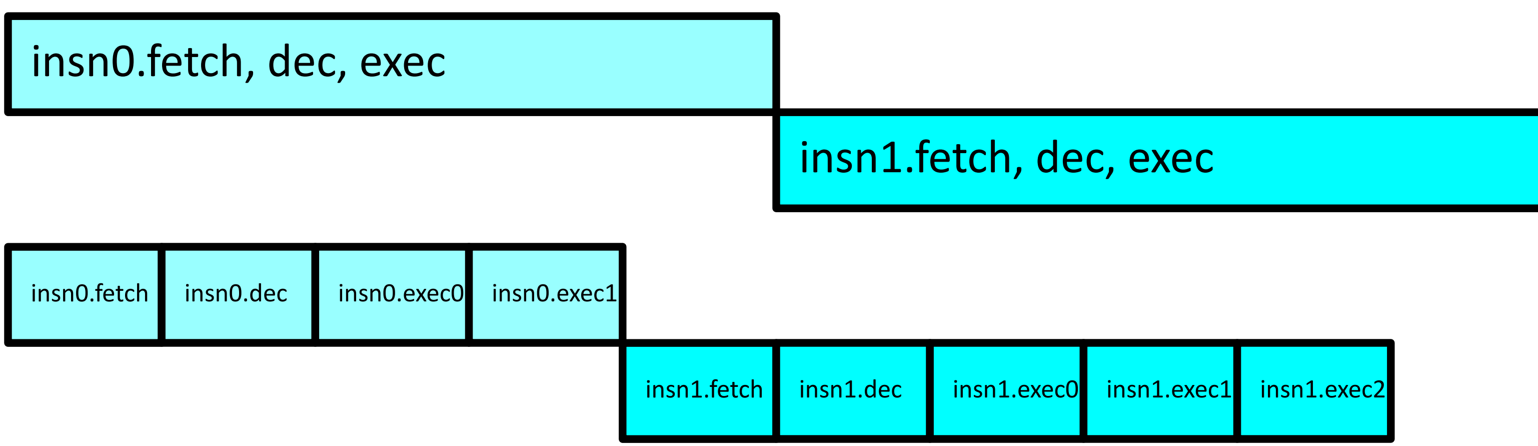
bulej@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Factors limiting CPU performance

- **Clock cycle length**
  - Limited by the most complex step of the most complex instruction
  - **Speedup:** moving from single-cycle to multi-cycle datapath
    - Simple instructions can be executed faster

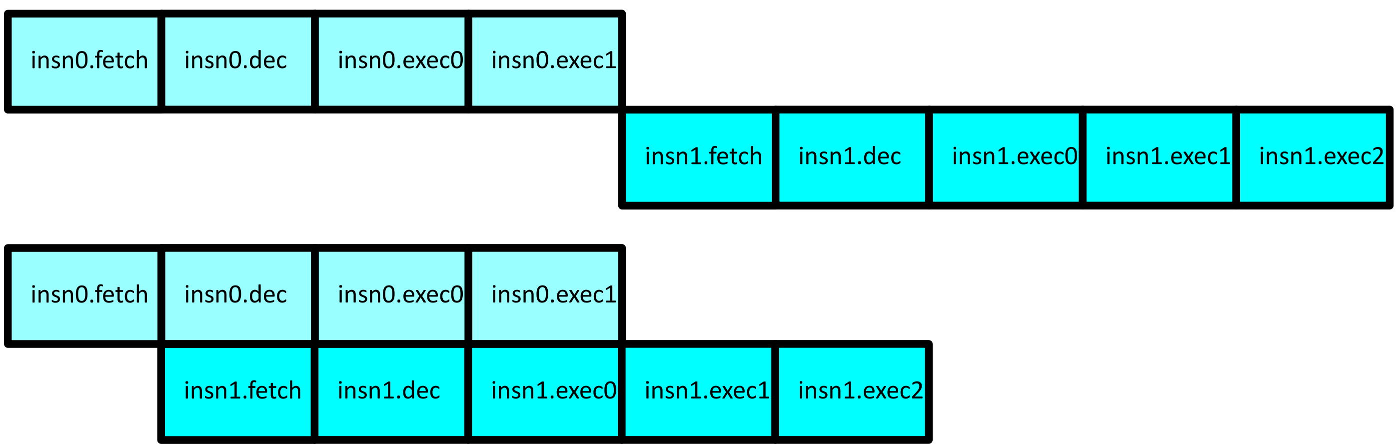


Factors limiting CPU performance (2)

- **Clocks per instruction (CPI)**
  - Limited by the number of instructions executed at the same time
    - Even a multi-cycle datapath executes only a single instruction at a time
  - Latency vs. throughput
    - **Latency** of a single instruction is determined by clock cycle length (we cannot keep shortening it forever)
    - **Throughput** of a sequence of instructions (whole program) can be improved by executing multiple instructions at the same time

Pipelined instruction execution

- **Hiding instruction latencies**
  - The datapath starts the 1st step of the next instruction while executing the 2nd step of the previous one
  - Instruction-level parallelism (preserves sequential execution model)
  - Latency (execution time) of individual instructions remains unchanged, but overall throughput increases



Pipelined processor performance

- **Rough estimate**
  - Executing *n* instructions, clock cycle *t*, *k* steps per instruction

    $$T = n \cdot (k \cdot t)$$

  - Pipelined execution in *k*-stage pipeline
    - The first instruction leaves the pipeline after *k* clocks, all other after 1 clock

    $$T_p = k \cdot t + (n-1) \cdot t$$

  - Speedup

    $$Speedup = \frac{T}{T_p} = \frac{n \cdot (k \cdot t)}{k \cdot t + (n-1) \cdot t} = \frac{n \cdot k}{k + (n-1)}$$

  - Speedup for *n* >> *k*

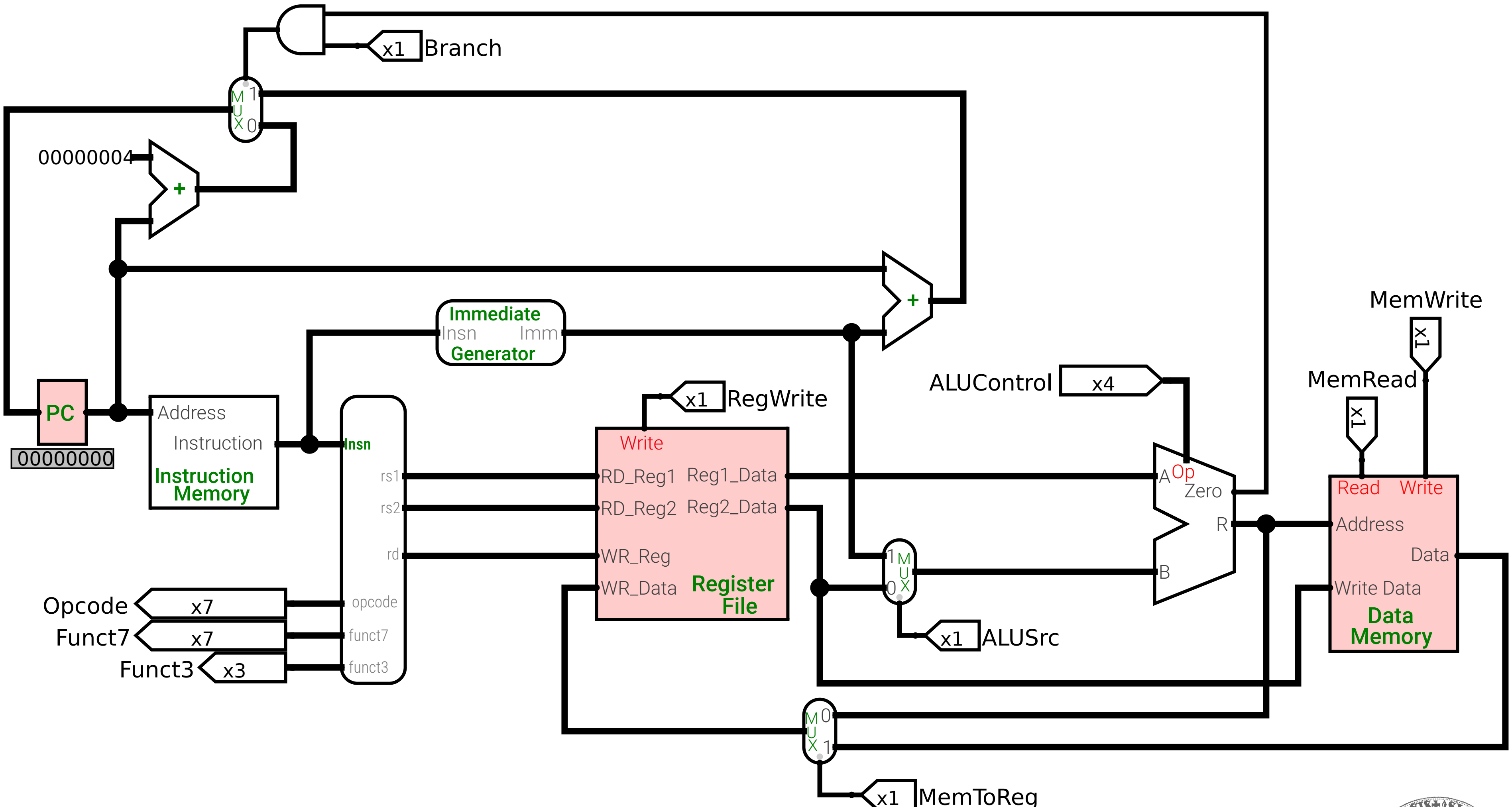
    $$k + (n-1) \approx n$$

    $$Speedup \rightarrow k$$

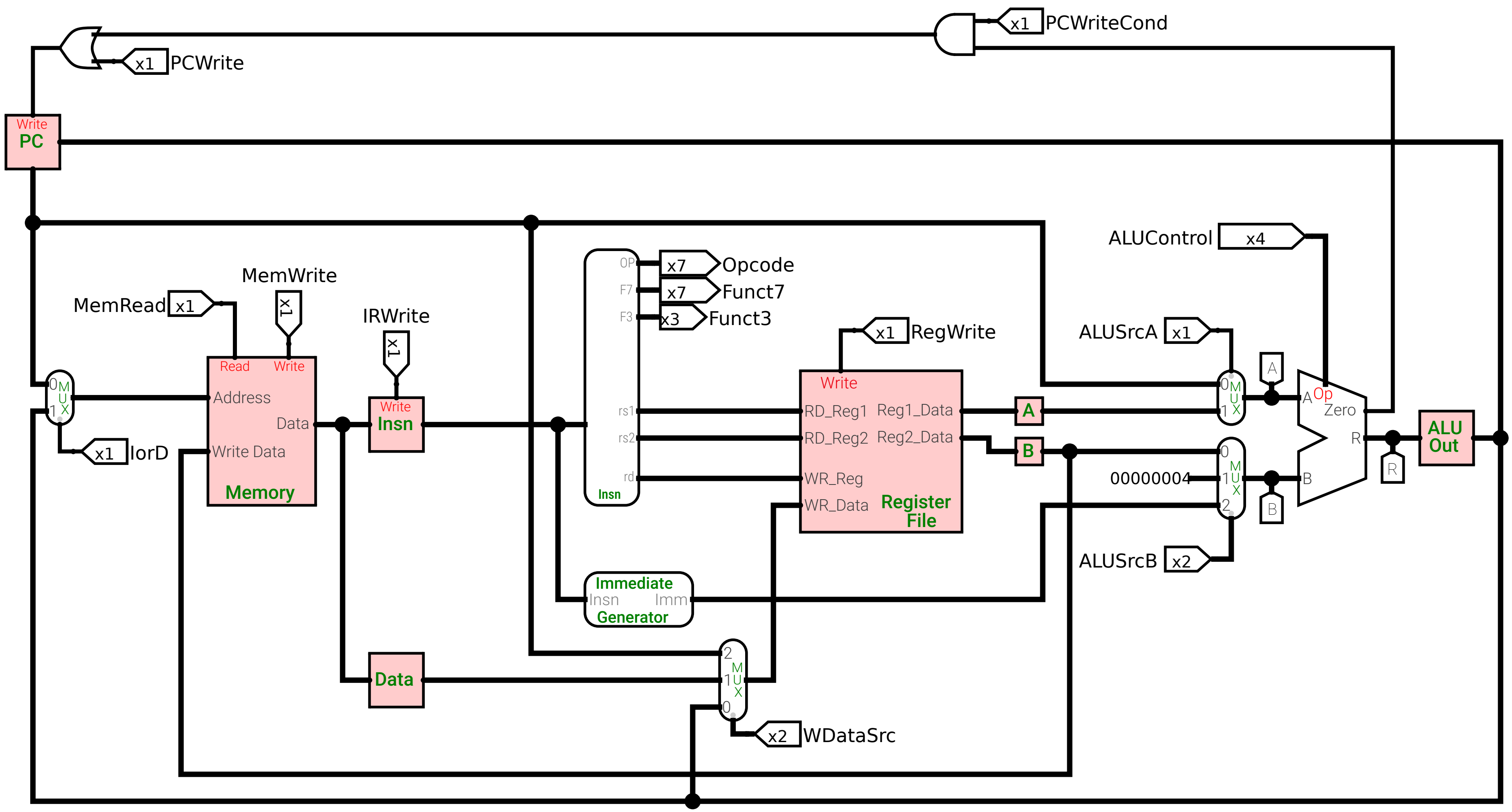
Datapath for pipelined execution

- **Basic idea**
  - Single-cycle datapath as a foundation
    - Separate instruction and data memories
    - Additional adders (ALU cannot be shared)
  - Elements of the multi-cycle datapath
    - Executing instructions in multiple steps
    - Latch registers to retain the results of the previous step (memory, register, and ALU outputs)

Recall: single-cycle datapath

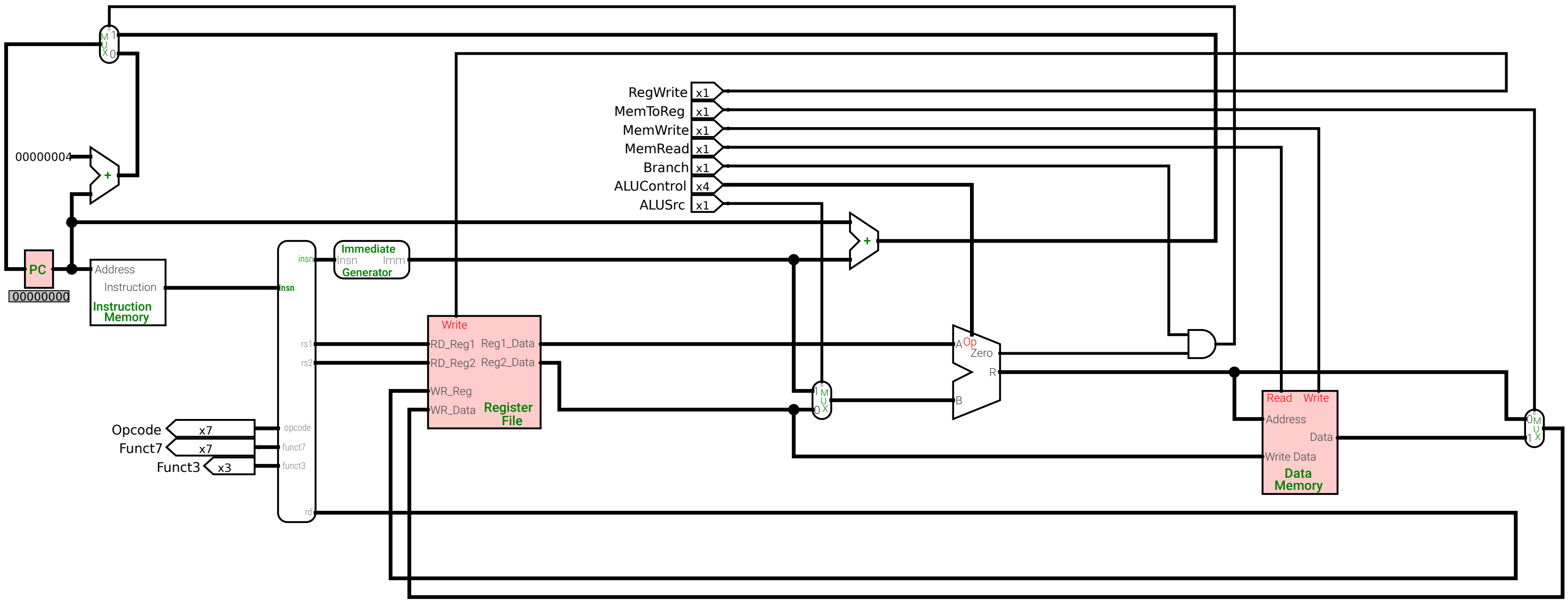


Recall: multi-cycle datapath



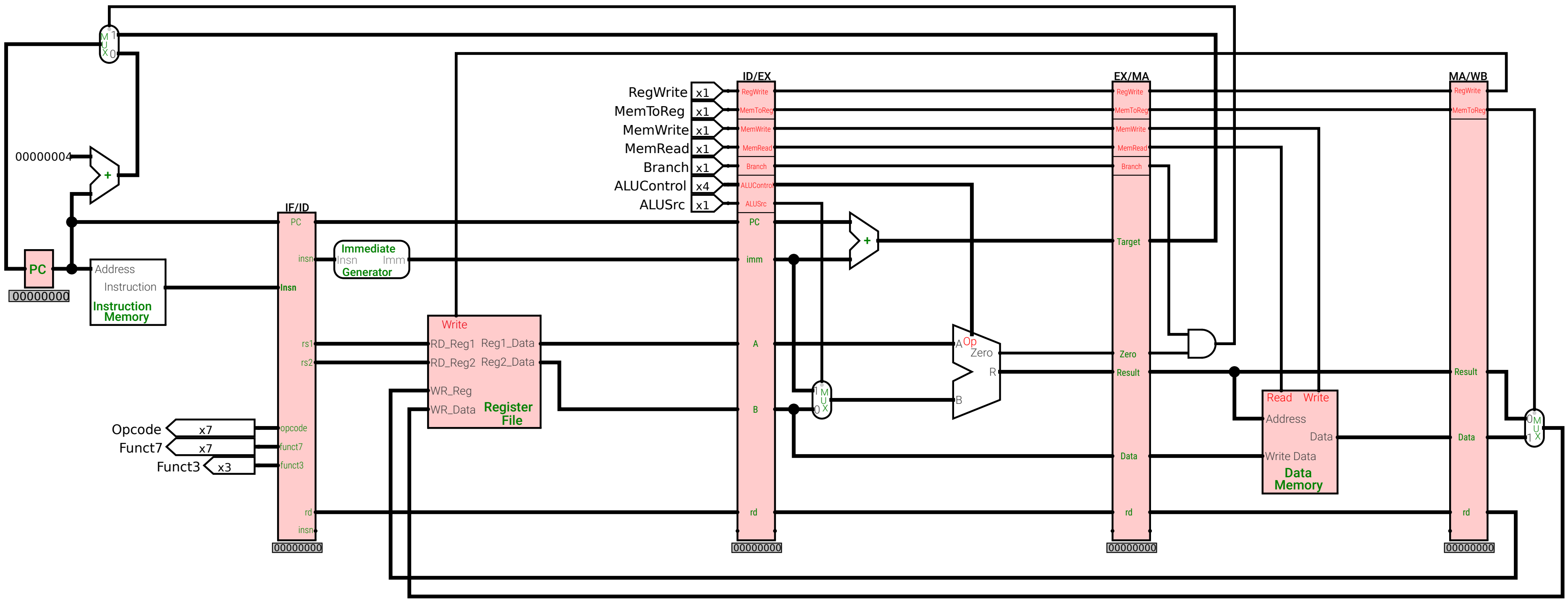
Datapath for pipelined execution (2)

- **Start with single-cycle...**



Datapath for pipelined execution (3)

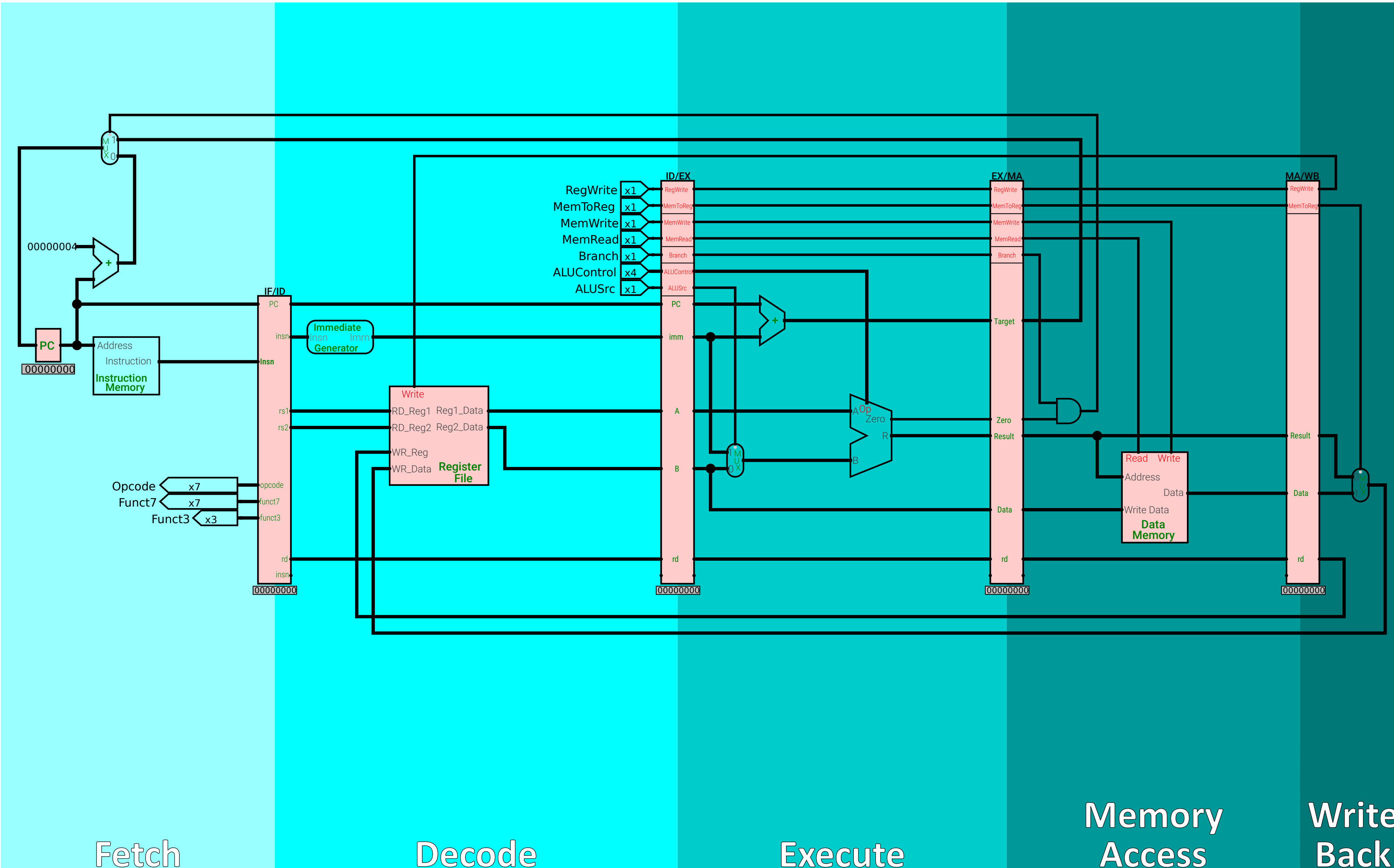
- … and just add pipeline registers.



Datapath for pipelined execution (4)

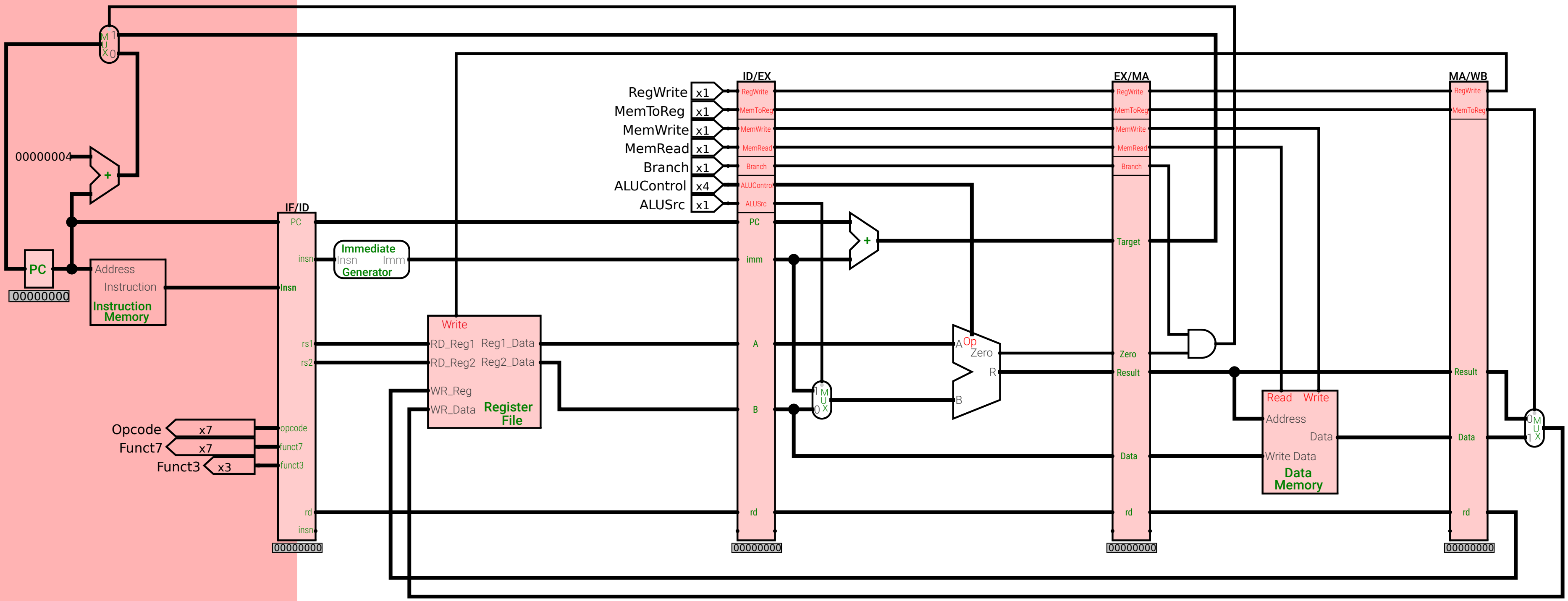
- **Datapath split into *k* stages**
  - Each stage is processing different instruction
    - The slowest stage determines the pipeline speed
    - Latches to hold results between successive stages
      - Instruction state, operands, results, control signals
      - Instructions in the datapath are in different state of execution
  - **Ideal case:** CPI = 1
    - The pipeline *completes* one instruction in each cycle
      - Instruction latency increases overhead, not throughput
  - **Realistic case**: CPI > 1
    - Pipeline delay and overhead

Datapath for pipelined execution (5)



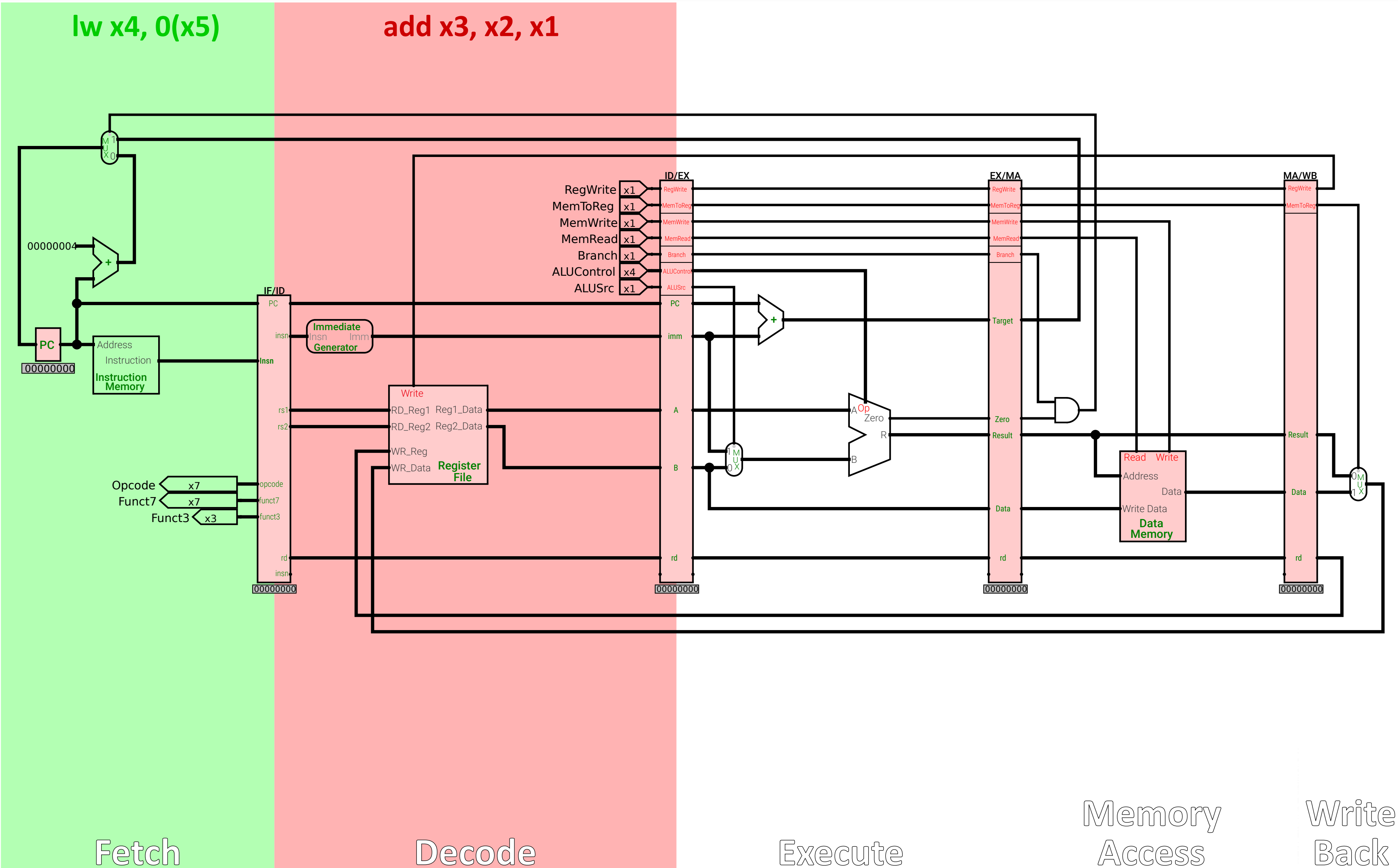
Executing 3 instructions, cycle 1

add x3, x2, x1

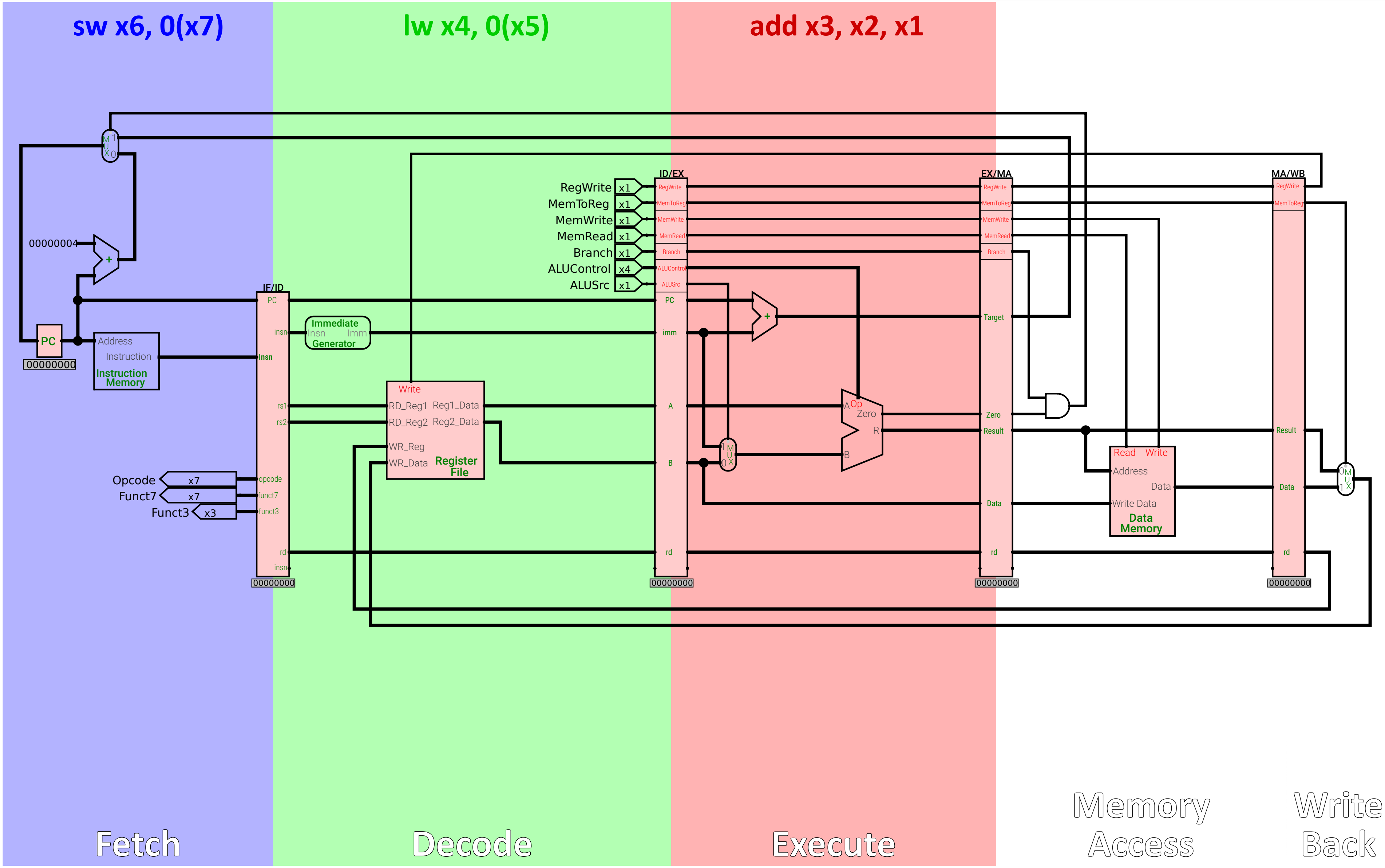


Fetch Decode Execute Memory Access Write Back

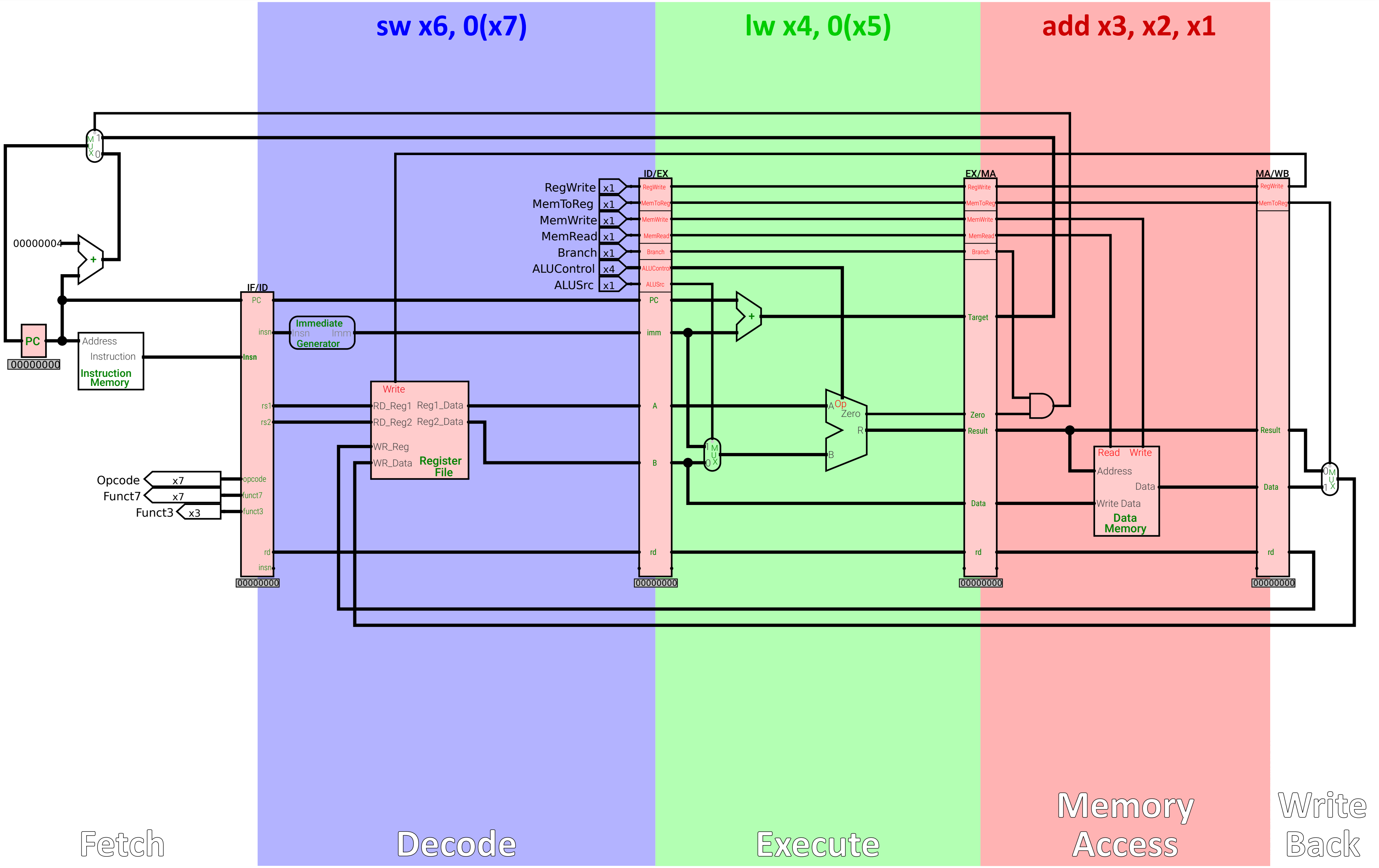
Executing 3 instructions, cycle 2



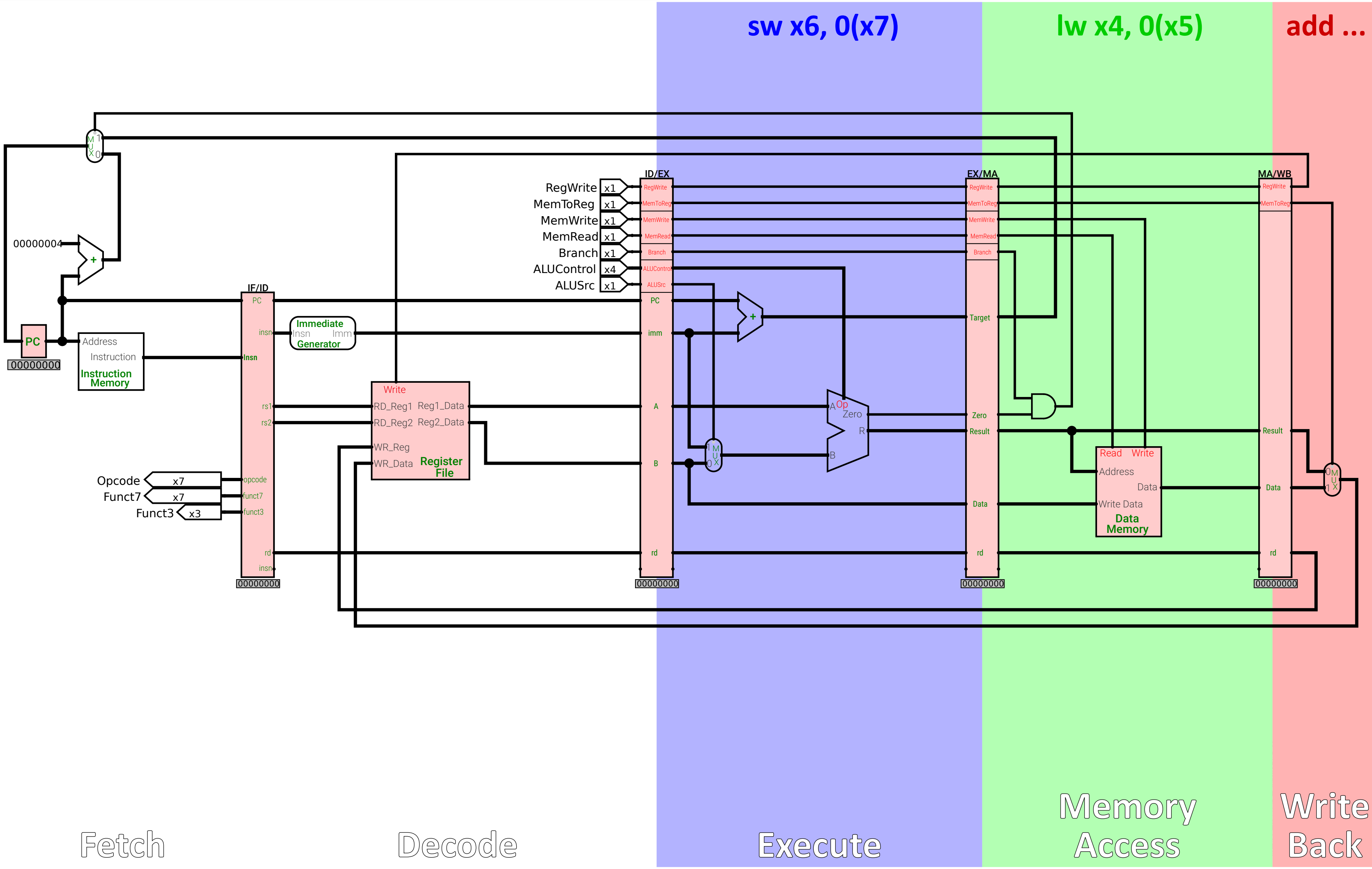
Executing 3 instructions, cycle 3



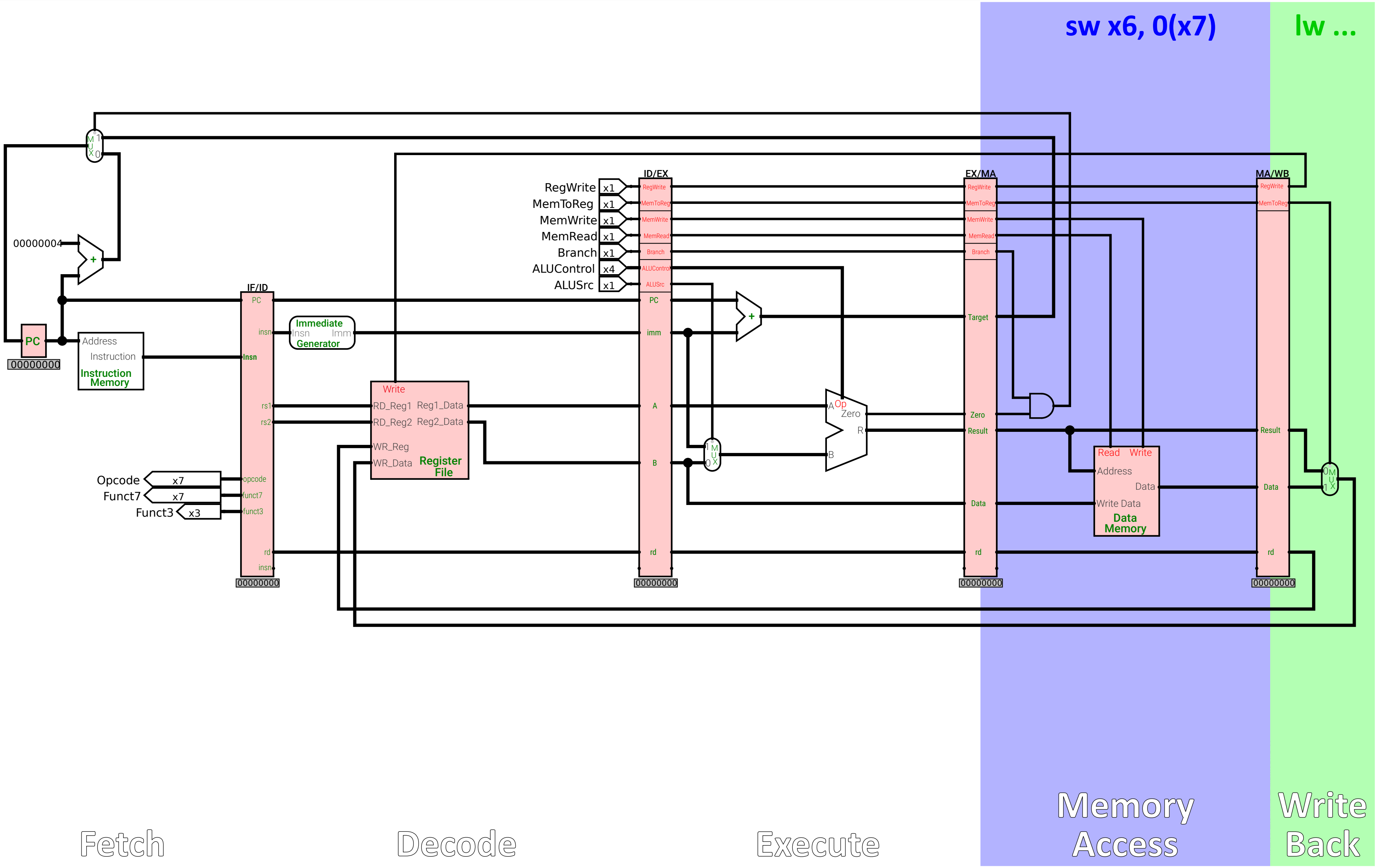
Executing 3 instructions, cycle 4



Executing 3 instructions, cycle 5

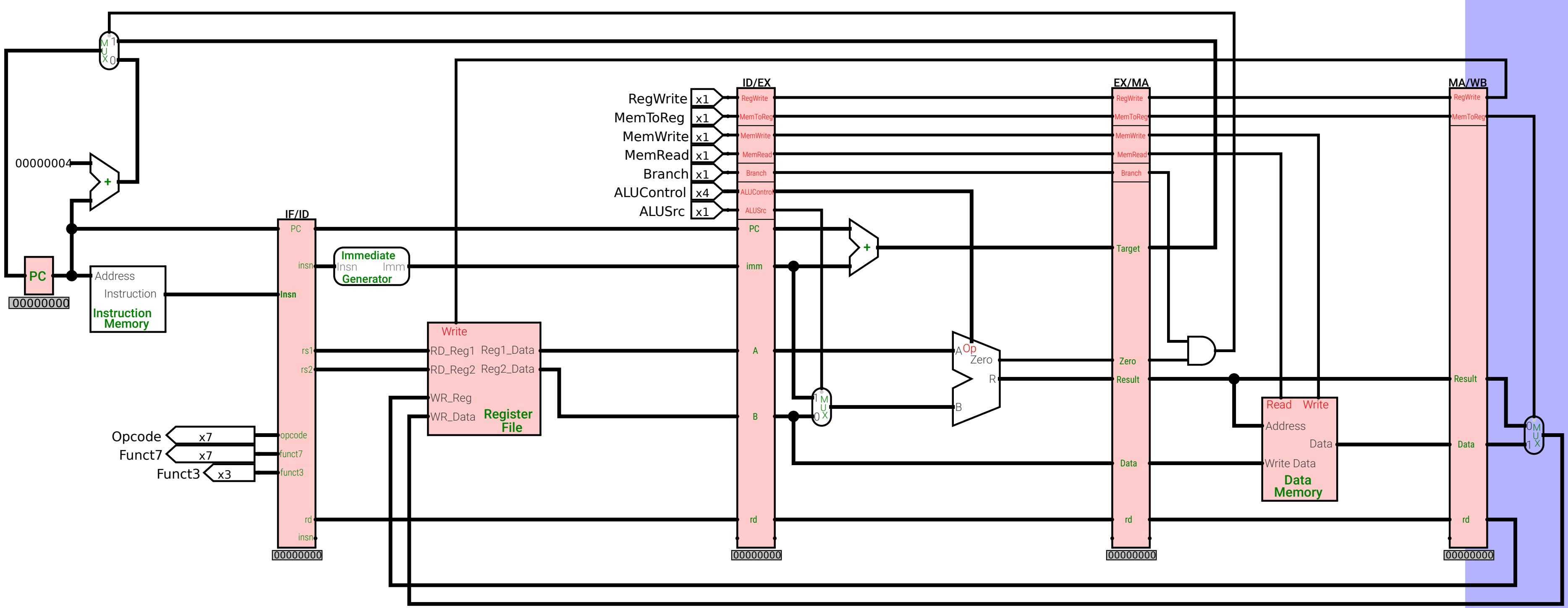


Executing 3 instructions, cycle 6



Executing 3 instructions, cycle 7

sw …

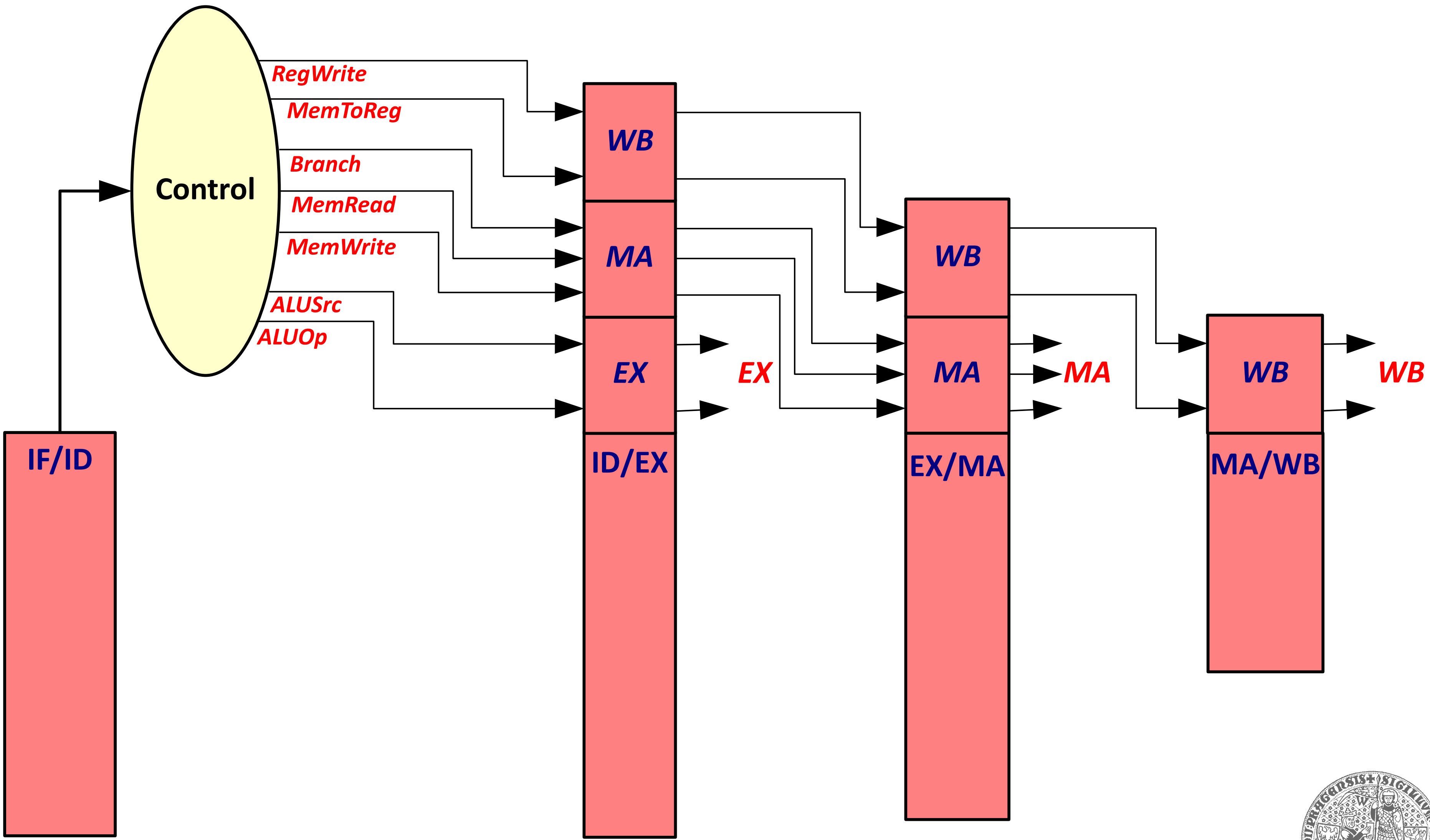


Fetch Decode Execute Memory Access Write Back

Pipeline control

- **Based on single-cycle control**
  - Control signals need to be activated in stages
  - Combinational logic or ROM decodes opcode
  - Signal path for control signals is pipelined, with latch registers between stages
    - Each instructions "carries" its own control signals with it after it has been decoded
- **Based on multi-cycle control**
  - Mostly complex solutions
    - A single finite-state automaton
    - Hierarchy of automatons, one for each stage

Pipeline control (2)



Pipelined datapath performance

- **Single-cycle datapath**
  - Clock = ***50ns***, ***CPI=1*** $\Rightarrow$ **50*ns*** per instruction
- **Multi-cycle datapath**
  - 20% branch (3T), 20% load (5T), 60% ALU (4T)
  - Clock = ***11ns, CPI≈ (20% × 3) + (20% × 5) + (60% × 4) = 4***
  - ***44ns*** per instruction
- **Pipelined datapath**
  - Clock = ***12ns*** (approx. 50ns/5 stages + latch overhead)
  - ***CPI = 1*** (one instruction retired in each cycle)
    - But in reality ***CPI = 1 + stall penalty > 1***
  - ***CPI = 1.5*** $\Rightarrow$ ***18ns*** per instruction

Designing ISA for pipelining

- **Equal-length instructions**
  - Easy to fetch instructions in stage 1 and decode them in stage 2
    - Multi-byte instructions considerably more complex to fetch/decode
- **Few instruction formats, fixed position of source register fields**
  - Stage 2 can start reading register file while the instruction is being decoded
    - Asymmetric instruction format would require splitting stage 2 to first decode an instruction and then to read the registers
- **Memory operands only appear in loads or stores**
  - Stage 3 (execute) can be used to calculate memory address for accessing memory in the subsequent stage
    - Operating directly on memory operands would require expanding stages 3 and 4 into address stage, memory stage, and execute stage
- **Operands must be aligned in memory**
  - Single data transfer instruction requires only one memory access
    - Data can be transferred in a single pipeline stage

Why is CPI = 1 unachievable?

- **Realistic pipeline**
  - CPI = 1 + *stall penalty*
    - Penalty corresponds to frequency and duration of pipeline stalls
      - Big penalties not an issue, if they are very rare
      - Penalties impact the optimal number of pipeline stages
  - **Stall** is a cycle in which pipeline does not retire an instruction
    - One stage must wait for another to complete
    - Inserted to prevent a pipeline hazard
  - **Hazard**
    - A situation when the next instruction cannot execute in the following clock cycle

Pipeline hazards

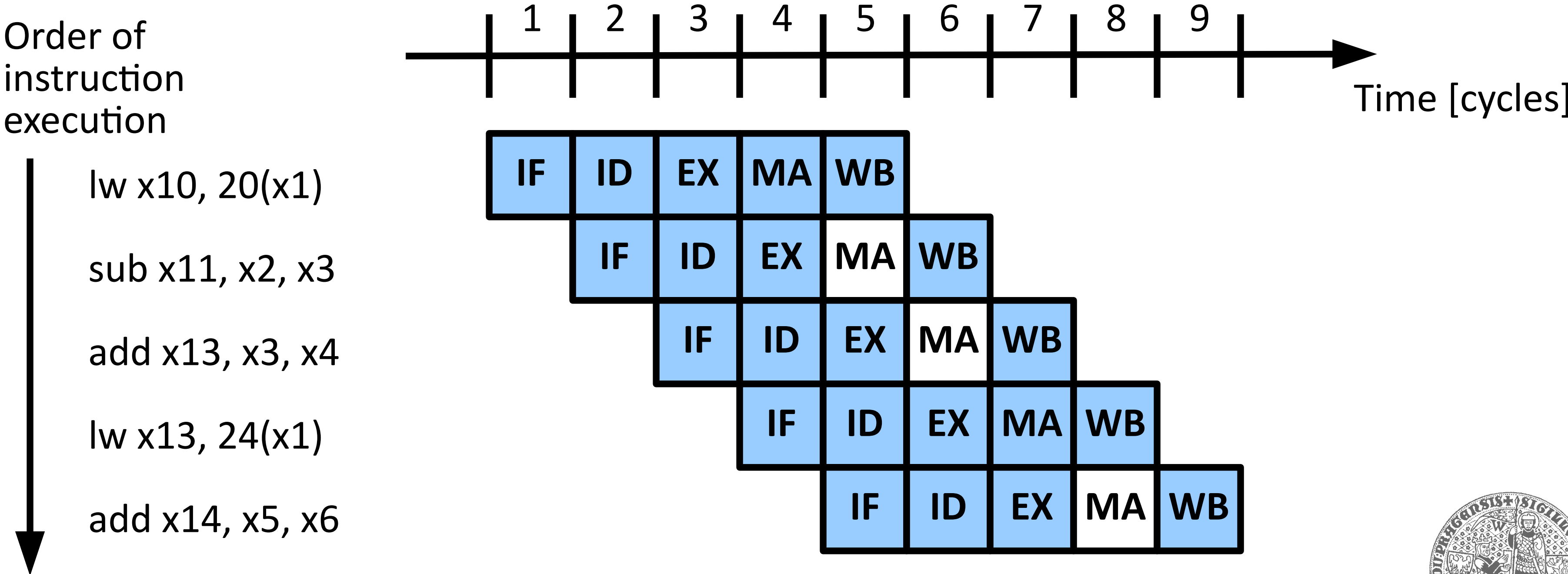
- **Structural hazard**
  - A datapath does not support a specific combination of instructions
  - Concurrent use of a shared resource from multiple pipeline stages
  - Example: shared instruction and data memory
    - Load instructions in 4th stage of execution would interfere with instruction fetch
    - Solution: separate instruction and data memories
      - Real CPU: separate instruction and data cache

Pipeline hazards (2)

- **Data hazard**
  - Instruction does not have data for execution
    - Operand values are the results of an instruction that is still in the pipeline
    - Needs to wait for the preceding instructions to finish
- **Control hazard**
  - Pipeline needs to make a decision before executing an instruction
  - Branch instruction finished in the 4th stage (MA)
    - By that time, the pipeline will have fetched 3 other instructions

Pipeline diagrams

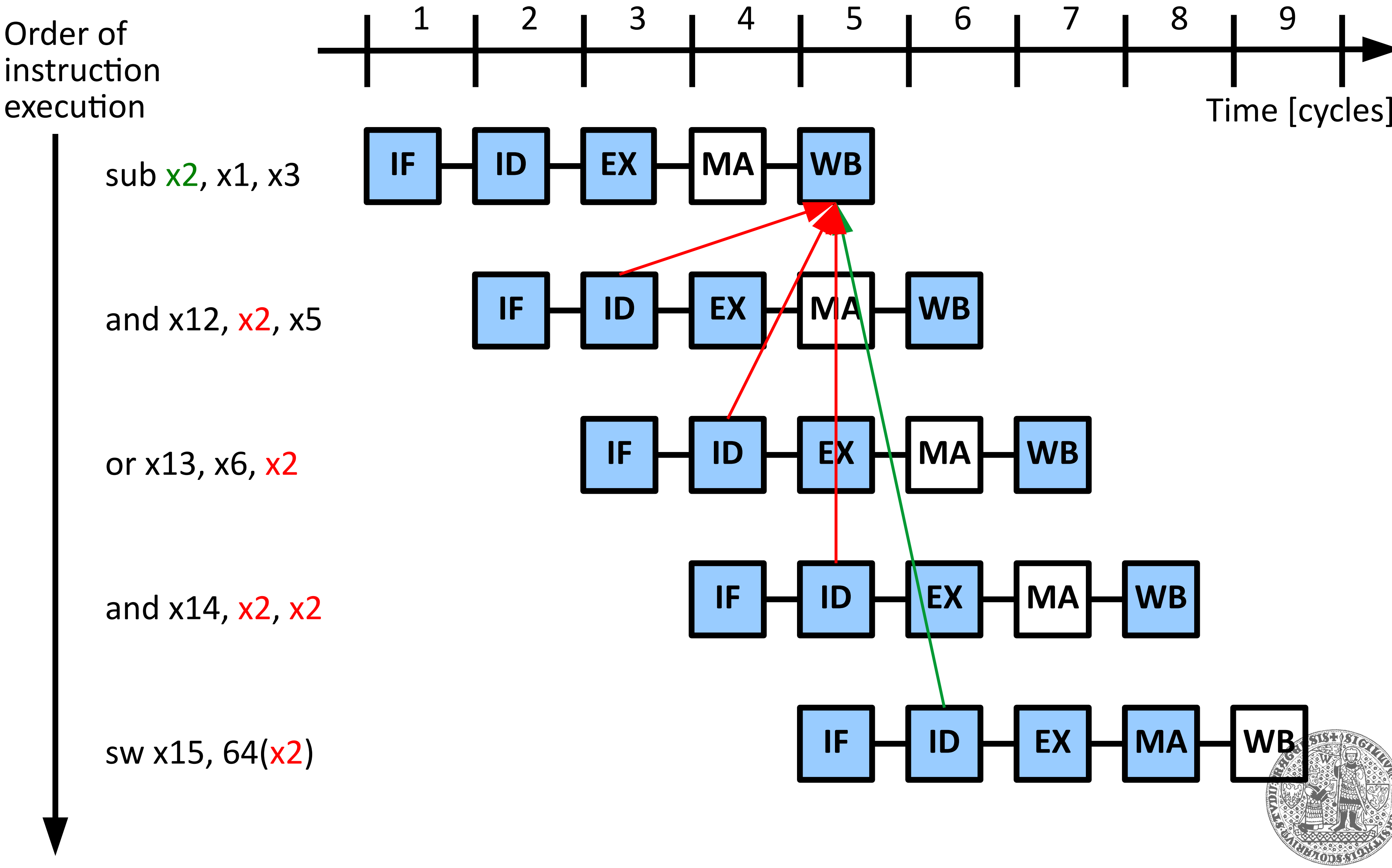
- **Simplified pipeline representation**
  - Each stage takes 1 cycle to execute
  - Discrete time in clock cycles



Data hazard

- **Dependencies between instruction operands**
  - Operand is a result of a preceding instruction
  - Operand is the content of memory read by preceding instruction
- **Finding dependencies during design**
  - Graph of dependencies
    - Nodes = pipeline elements active at given time
    - Edges = control or data signals
    - Dependencies = edges pointing to “future time”
- **Detecting dependencies in hardware**
  - Compare source and destination register numbers in all instructions present in the pipeline

Data hazard (2)



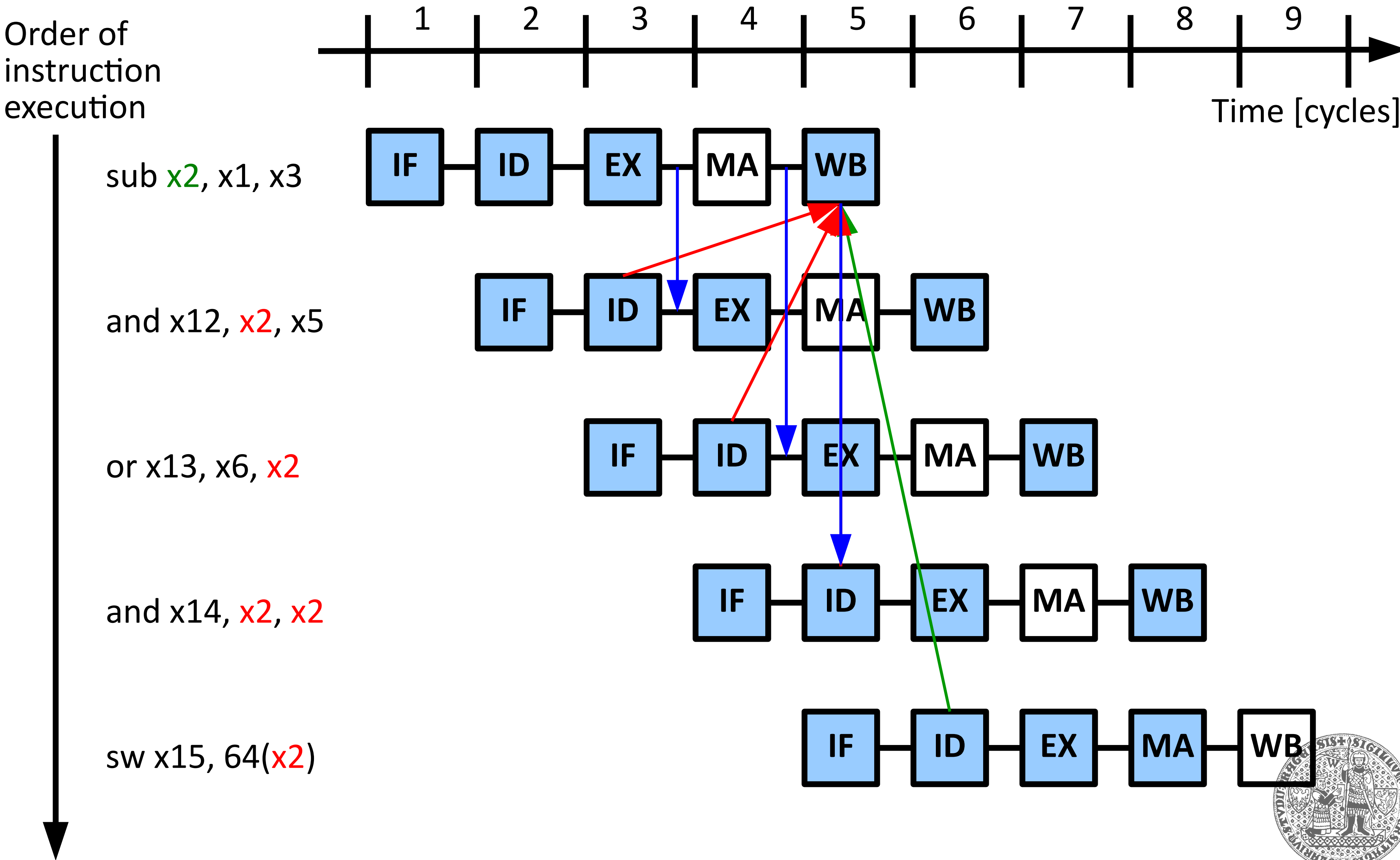
Dealing with data hazards

- **Compiler level (*software interlock*)**
  - Ordering instructions so that they reach pipeline only when all the operands are available
    - Need to insert other (independent) instructions between mutually dependent instructions
    - Using a no-operation (*nop*) instruction in the worst case
  - Theoretically possible, practically infeasible
    - Leaks CPU implementation details across the hardware-software interface (ISA)
    - ***MIPS*** *=* ***M****icroprocessor without* ***I****nterlocked* ***P****ipeline* ***S****tages*

Dealing with data hazards (2)

- *Forwarding/bypassing*
  - Use the intermediate values (not yet written to registers) as operands for dependent instructions
    - Fetch operand from pipeline registers of the preceding instructions.
  - *Forwarding unit*
    - Control circuitry to detect dependencies and enable forwarding of values
    - Checks if source operand of an instruction is a destination operand of any of the preceding instructions
      - EX/MA.RD := ID/EX.RS1
      - EX/MA.RD := ID/EX.RS2
      - MA/WB.RD := ID/EX.RS1
      - MA/WB.RD := ID/EX.RS2

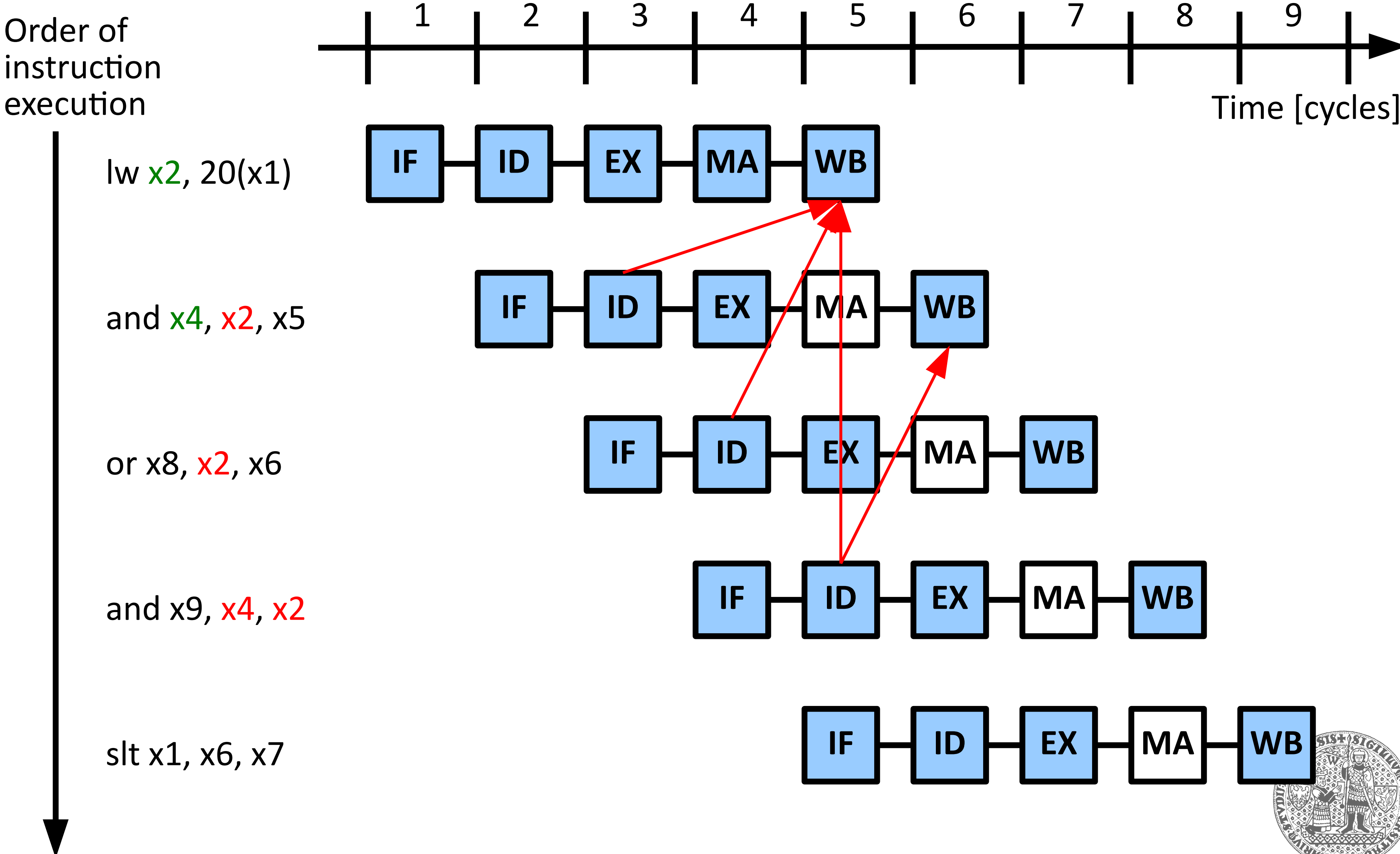
Data hazard – forwarding/bypassing



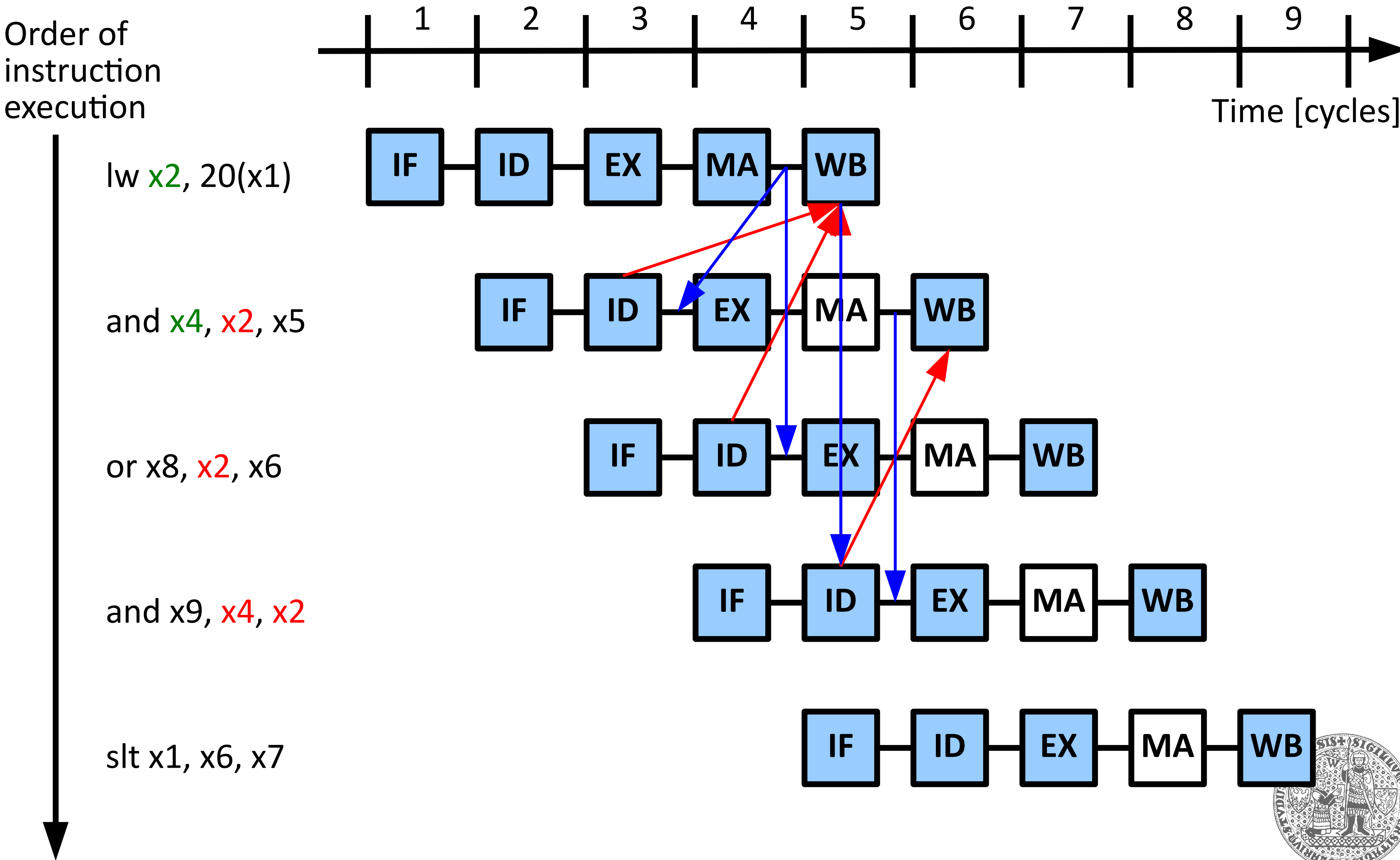
Dealing with data hazards (3)

- **Delay instruction execution (pipeline stall)**
  - Pipeline executes an “empty” operation
  - Necessary in case of ***load/use dependency***
    - An instruction immediately following a load instruction uses the result of the load
  - *Hazard detection unit*
    - Control circuitry to detect dependency and cause pipeline stall
    - Checks if the source operand of an instruction is the target operand of the earlier memory load instruction
      - ID/EX.MemRead && ID/EX.RD != 0 &&
        (IF/ID.RS1 = ID/EX.RD || IF/ID.RS2 == ID/EX.RD)

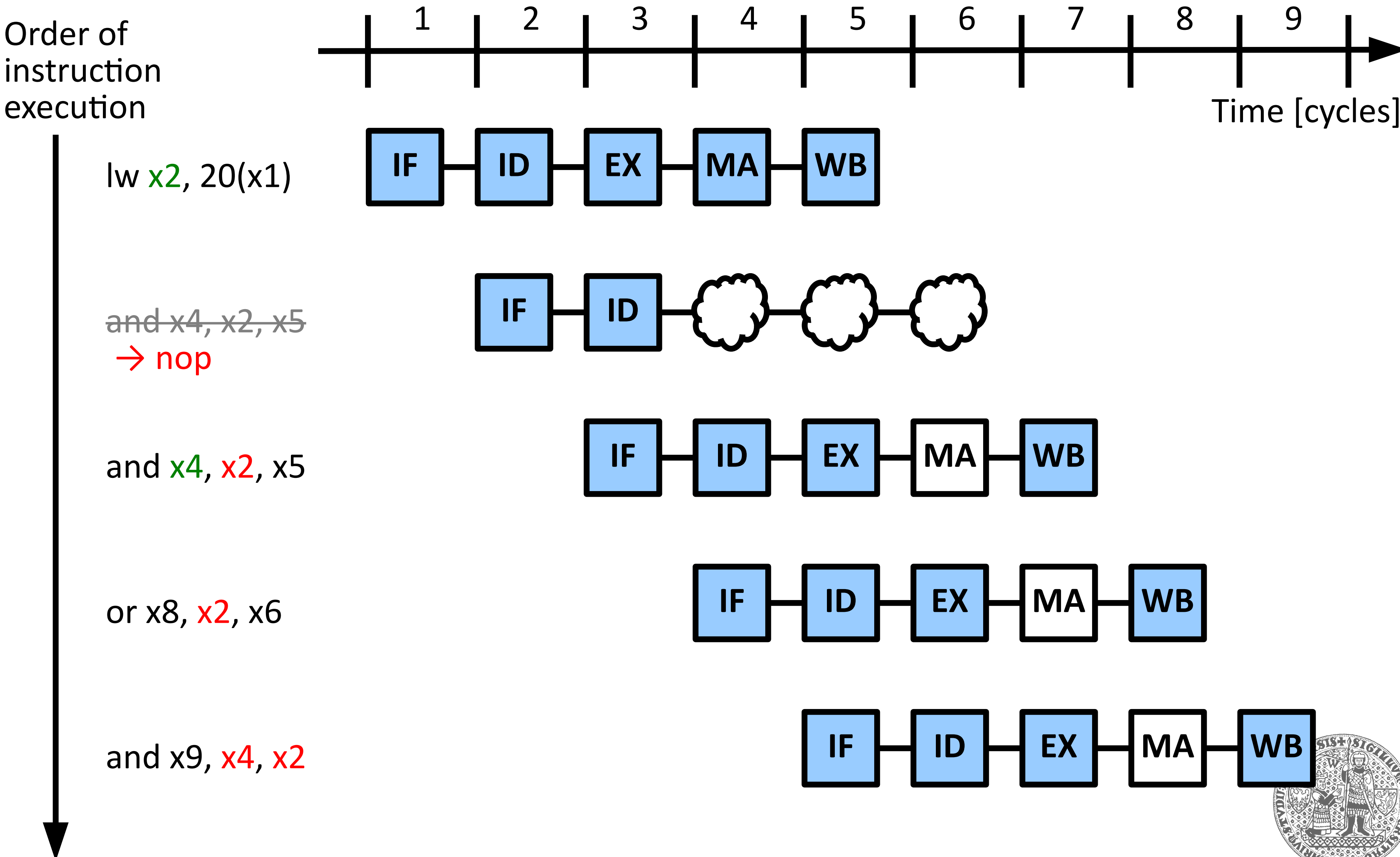
Data hazard – load/use dependency



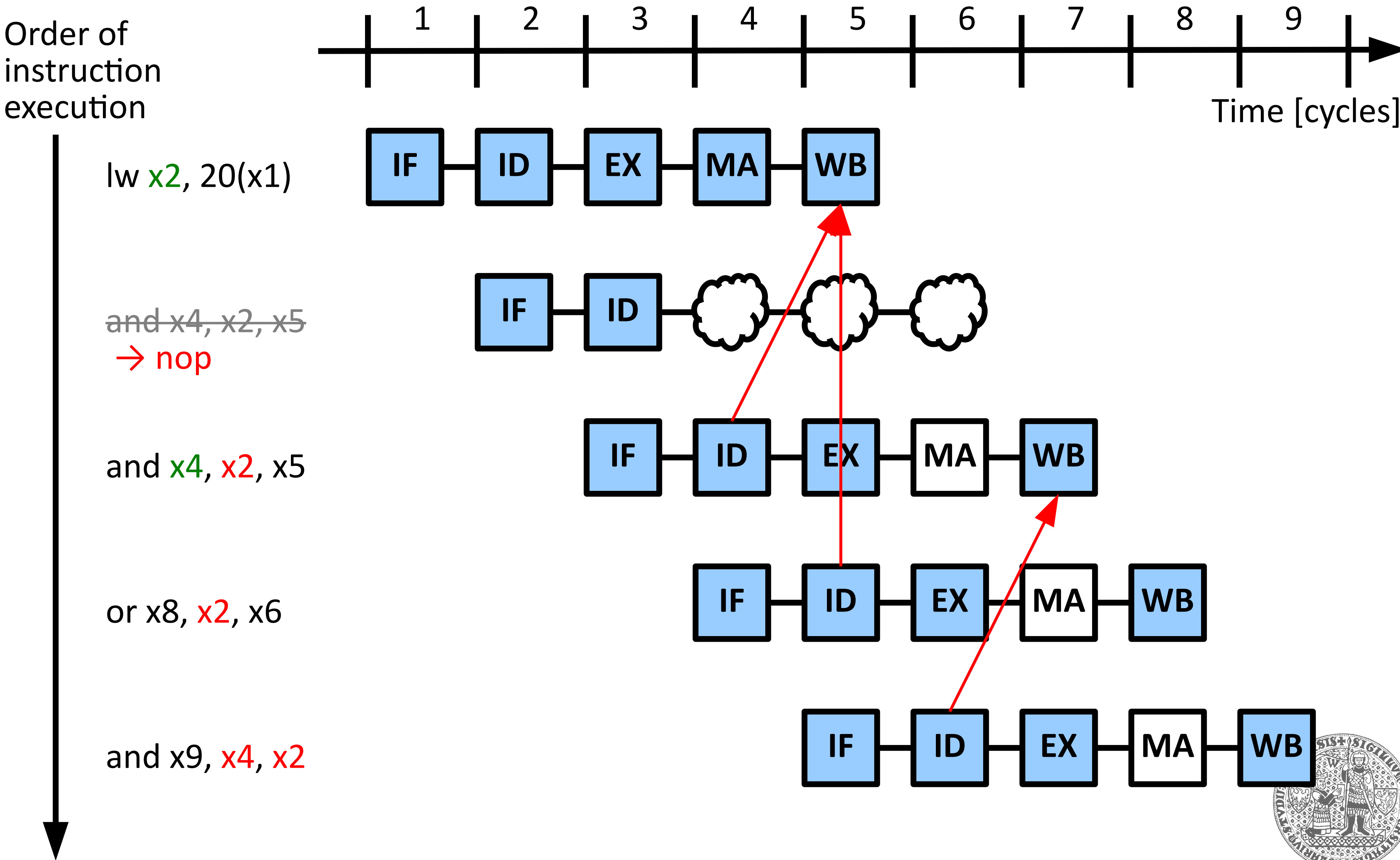
Data hazard – load/use & forwarding



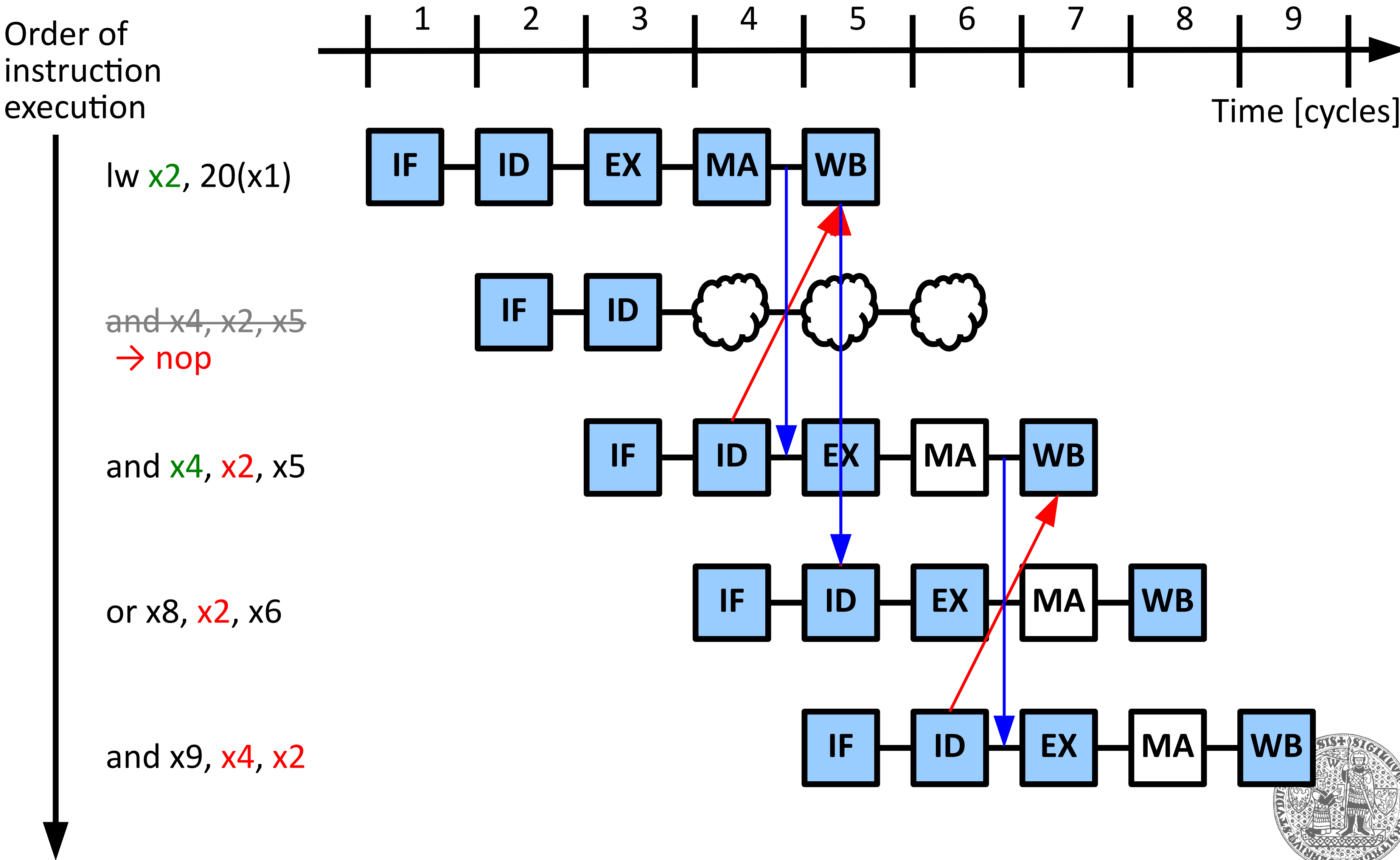
Data hazard – pipeline stall



Data hazard – pipeline stall (2)



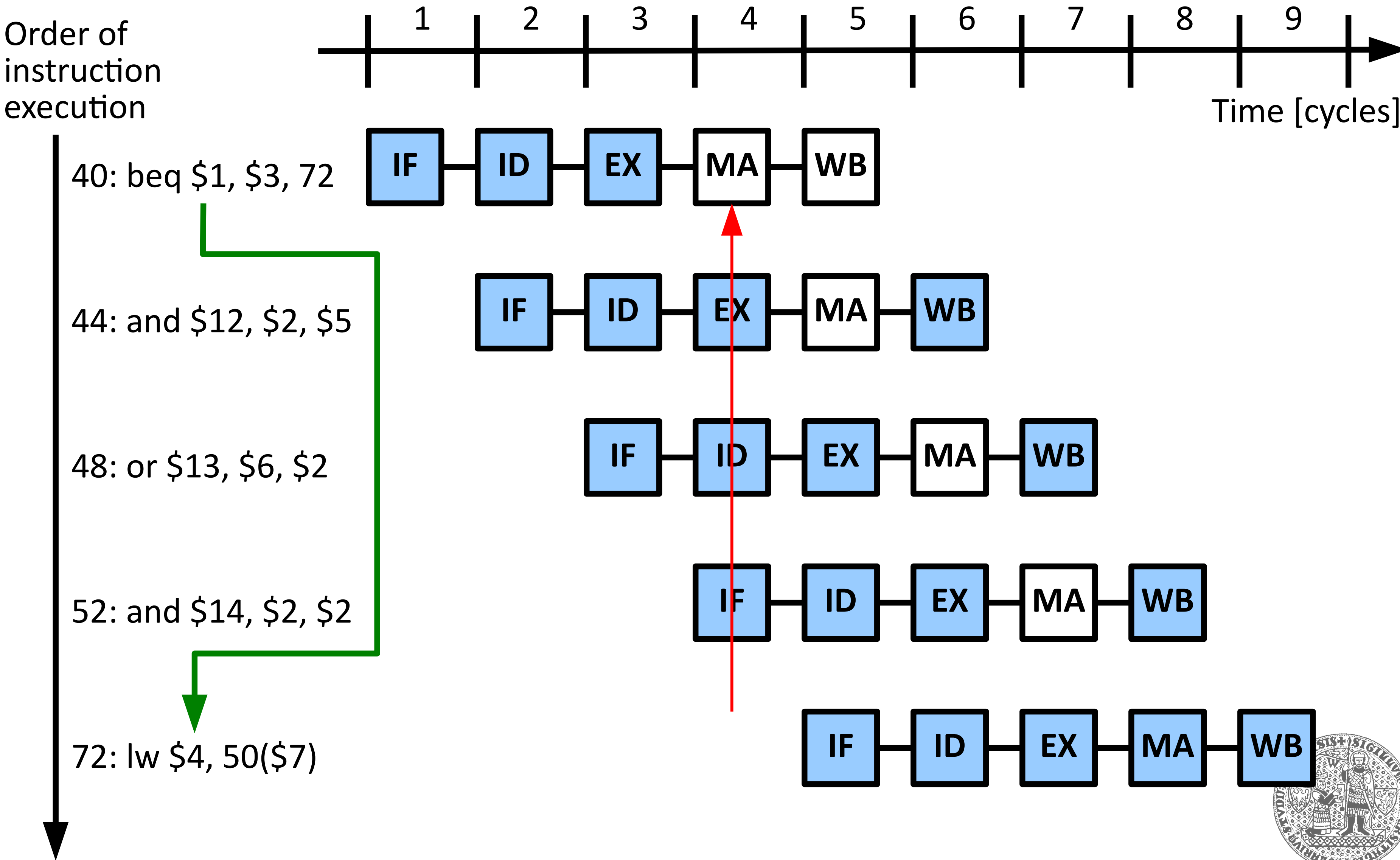
Data hazard – pipeline stall (3)



Control hazard

- **Which address to read the next instruction from?**
  - PC value influenced by jump and branch instructions
    - Depends on the result of an instruction executed several cycles later than required: we need to read an instruction in every cycle
  - Exceptions and interrupts
- **Handling control hazard**
  - Forwarding not possible
    - Target address may be known, but the branch condition is evaluated later
  - **Goal:** minimize pipeline stalls

Control hazard – branching



Dealing with control hazards

- **Stall until branch outcome is known**
- **Try to keep the pipeline full**
  - Assume branch not taken (until proven otherwise)
  - Reduce the delay of branches
    - So far PC for next cycle selected in MA stage
    - Execute branch earlier → less instructions to flush
      - Branch target: PC and immediate value already in IF/ID pipeline register → move branch adder from EX to ID stage
      - Branch condition: compare registers during ID stage, requires extra circuitry and forwarding/hazard detection logic
      - Requires simple test condition
      - Reduces branch penalty to 1 cycle if branch is taken
    - *MIPS (but not RISC-V): Branch delay slot*
      - Always execute 1 more instruction after branch

Dealing with control hazards (2)

- **Trying to keep the pipeline full**
  - Where to read next instruction from?
    - *Branch target buffer*
      - Cache target addresses of branch instructions
    - Execute instructions speculatively
      - Keep executing instructions regardless of branch condition
      - If we later find that we should execute instructions on another path, just flush the pipeline and start over
      - May require partial virtualization of register file and store buffers

Branch prediction

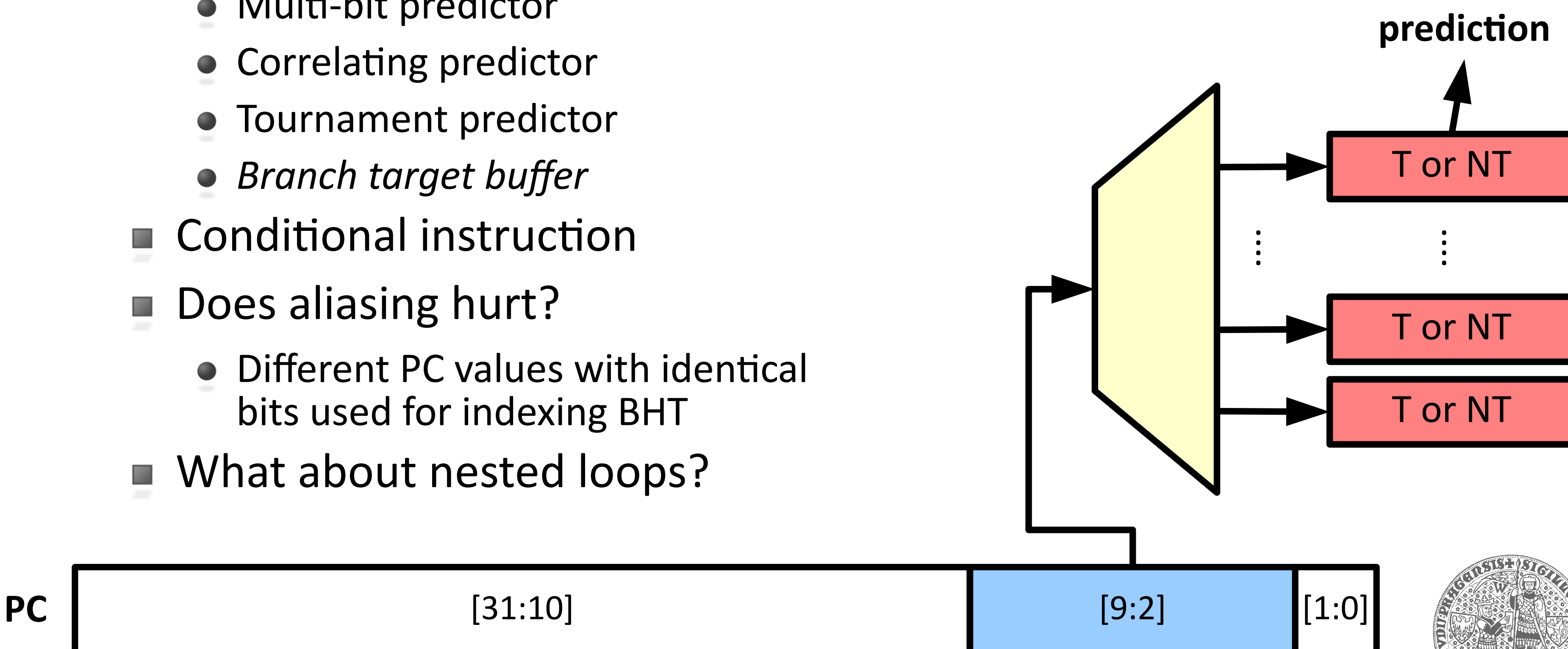
- **Static prediction**
  - Ignores history of branch outcomes
  - Without hints
    - Heuristics determined by hardware
    - Generally assume branch not taken
    - Complex heuristics (e.g., branch distance) uncommon
  - With hint
    - The more likely outcome determined by the instruction opcode

Branch prediction (2)

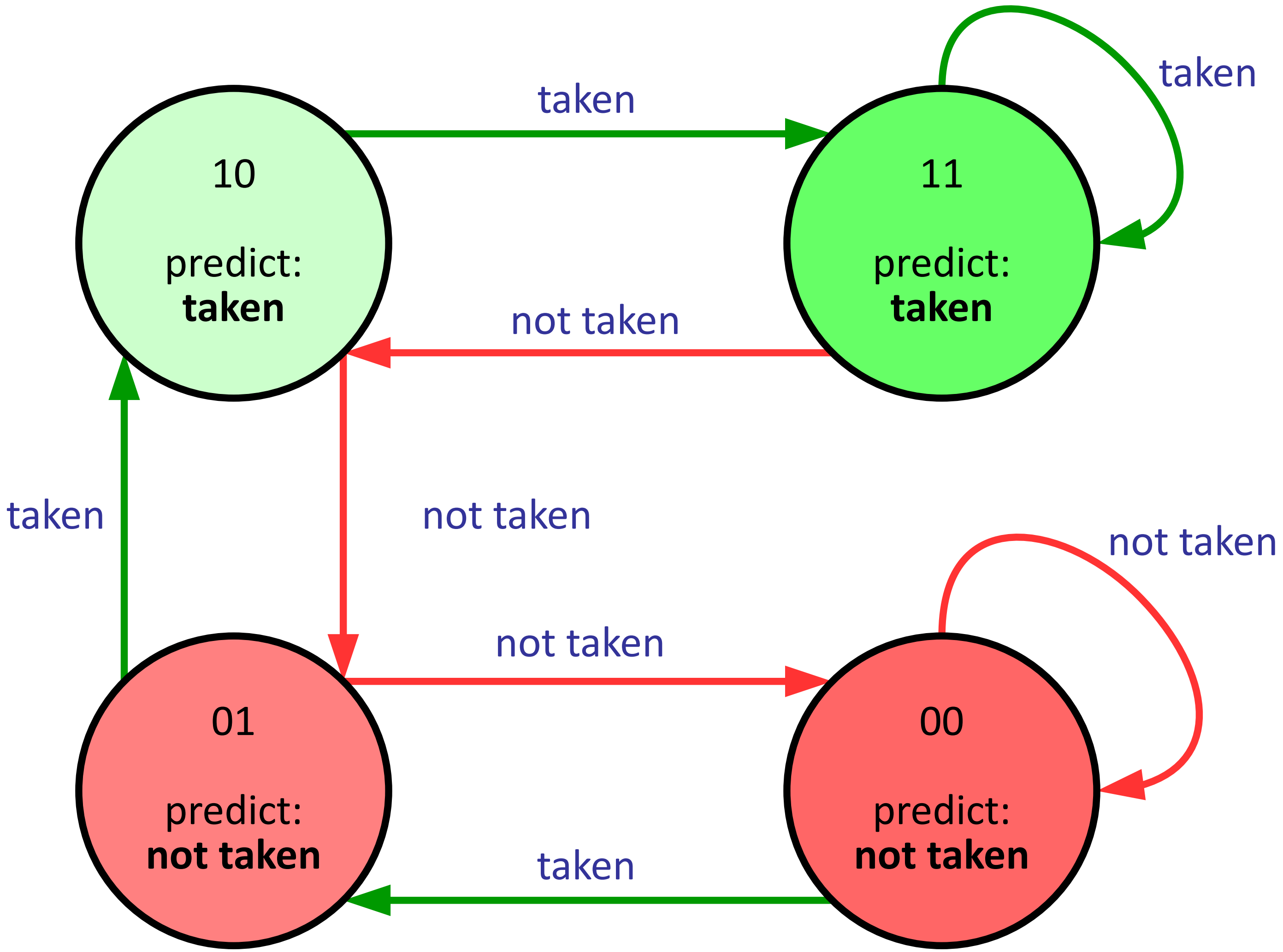
- **Dynamic prediction**
  - Takes past branch outcomes into account
  - *Branch prediction buffer (history table)*
    - Keeps the state of a predictor for a particular instruction
  - 1-bit predictor (2 states)
    - State reflects the previous outcome
    - Predicts the same behavior as in the past
  - Problem with loops: branch back except on last iteration
    - 2 mispredictions for simple loops
    - Multiplied in nested loops
  - 2-bit predictor (4 states)
    - General approach: count prediction success/failure, middle of range break point between predictions
    - Reduces mispredictions for cases strongly favoring certain outcome (typical for many branches)

Branch history table

- **Basic (1-bit) predictor**
  - Table of prediction bits indexed by (part of) PC
  - Extensions
    - Multi-bit predictor
    - Correlating predictor
    - Tournament predictor
    - *Branch target buffer*
  - Conditional instruction
  - Does aliasing hurt?
    - Different PC values with identical bits used for indexing BHT
  - What about nested loops?



2-bit branch predictor (saturated counter)



Pipelined datapath and exceptions

- **Pipeline contains *k* instructions**
  - Which instruction caused an exception?
    - Needs to be propagated through pipeline registers
  - On multiple exceptions, which one to handle first?
    - The one that is the earliest
  - Exception handling
    - Keep the processor state consistent
      - Data from pipeline registers are not written back (register file and memory contain values before the exception occurred)
    - Flush the pipeline before handling the exception
      - Similar logic to speculative handling of branch instructions

Increasing pipeline length

- **Trend: pipelines getting longer**
  - 486 (5 stages), Pentium (7 stages)
  - Pentium III (12 stages), Pentium 4 (20 – 31 stages)
  - Core (14 stages)
  - Consequences
    - Higher clock rate
      - Not linear with pipeline length, causes performance drop starting at certain pipeline lengths
        - Pentium 4 at 1 GHz slower than Pentium III at 800 MHz
    - Generally higher CPI
      - More costly penalties for mispredicted branches
      - Delays due to hazards that cannot be handled using forwarding/bypassing

Increasing the number of pipelines

- ***Flynn bottleneck***
  - Theoretical limitation of a scalar pipeline
    - 1 instruction in each stage → CPI = IPC = 1
    - Impossible to reach in practice (hazards)
    - Diminishing returns from increasing pipeline length
- **Superscalar (multiple issue) pipeline**
  - Instructions scheduled to multiple pipelines
    - 4 or more pipelines in modern processors
  - Exploiting instruction-level parallelism
    - Independent instructions can be executed in parallel

Instruction-level parallelism

- **Compiler schedules instructions**
  - Necessary even for scalar pipeline (reduce potential hazards)
  - More complex for superscalar pipeline
    - How many independent instructions streams can we find in a program?
      - **Ideal case:** copying a block of memory (unrolling the loop creates many independent instructions)
      - Normal programs contain significantly less opportunities
    - **An alternative:** *Simultaneous multi-threading* (SMT)

Simultaneous multi-threading

- **Execute instructions from more threads**
  - At the level of superscalar pipeline
    - Instructions from independent threads are independent by definition → more efficient use of superscalar pipeline
    - More energy efficient than implementing multiple cores
      - Additional register file and instruction reading logic
      - The rest of the CPU remains unchanged
    - The operating system "sees" multiple logical CPUs
    - **Problem:** Shared resources (cache, memory bandwidth)
    - *Intel Hyper-Threading Technology*

Temporal multi-threading

- **SMT adapted to a single pipeline**
  - Technically: thread switching on the CPU
  - **Fine-grained**
    - Switch thread with each instruction
    - Niagara (Sun UltraSPARC T1)
  - **Coarse-grained**
    - Switch when an instruction causes a delay (pipeline stall, cache miss, page fault)
    - Montecito (Intel Itanium 2)

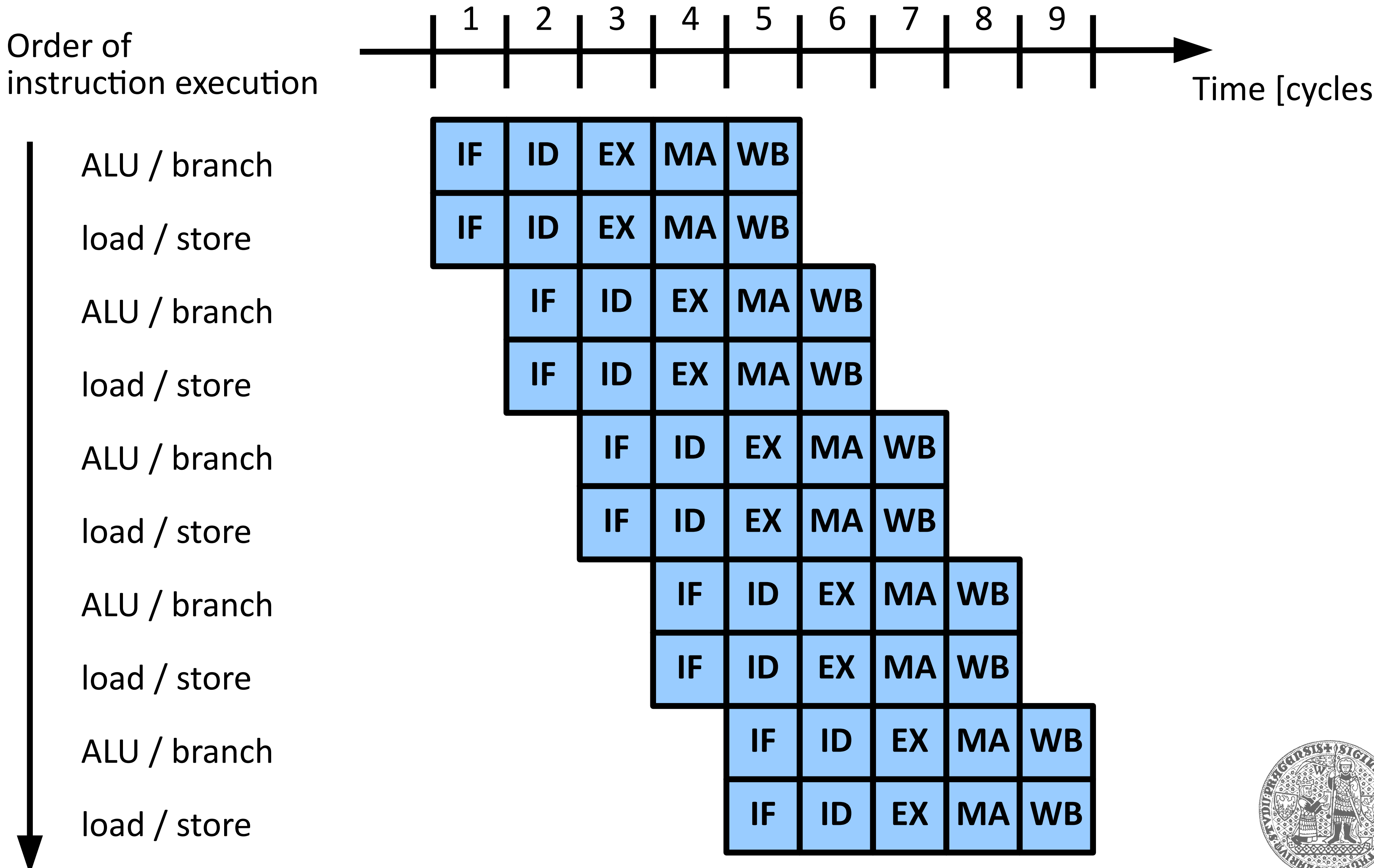
Common superscalar pipeline

- **Reading instructions**
  - A block of memory (16, 32 or 64 bytes), 4 – 16 instructions
  - Predicting one conditional branch in each cycle
- **Parallel instruction decoding**
  - Detecting dependencies and hazards
- **Multi-port register array with additional registers**
- **Multiple execution units**
  - Different ALUs, forwarding/bypassing logic
- **Access to memory**

Static multiple issue

- **Instruction schedule determined by compiler**
  - Pipeline executes instruction packets in-order
  - ***Issue packet***
    - A group of instructions to execute in parallel
    - Slots in the issue packet not necessarily orthogonal
      - *Very Long Instruction Word* (VLIW)
      - *Explicit Parallel Instruction Computer* (EPIC)
  - Performance strongly depends on compiler
    - Identify instruction-level parallelism in code
    - Instruction scheduling (issuing instructions to slots)
    - Some data and control hazards handled by compiler
    - Static branch prediction

Example: static multiple issue RISC-V



Example: static multiple issue RISC-V (2)

- **Changes wrt. single issue**
  - Reading 64bit instructions → 8-byte alignment
    - Unused slot can contain NOP instruction
  - Register file: support access from both slots
  - Additional adder to compute memory addresses
- **Problems**
  - Longer latency to use results
    - Register operations 1 instruction, load 2 instructions
    - More complex instruction scheduling for compiler
  - Penalties due to hazards are more costly

Example: static multiple issue RISC-V (3)

- **How to schedule this code?**

```
Loop:   lw     t0, 0(s1)
        addu   t0, t0, s2
        sw     t0, 0(s1)
        addi   s1, s1, -4
        bne    s1, zero, Loop
```

	ALU or branch insn	Data transfer insn	Clock cycle
`Loop:`		`lw t0, 0(s1)`	1
	`addi s1, s1, -4`		2
	`addu t0, t0, s2`		3
	`bne s1, zero, Loop`	`sw t0, 4(s1)`	4

- **Performance?**
  - 4 cycles, 5 instructions → CPI = 0.8 (instead of 0.5)

Example: static multiple issue RISC-V (4)

- **Unrolling 4 loop iterations...**

	ALU or branch insn	Data transfer insn	Clock cycle
`Loop:`	`addi s1, s1, -16`	`lw t0, 0(s1)`	1
		`lw t1, 12(s1)`	2
	`addu t0, t0, s2`	`lw t2, 8(s1)`	3
	`addu t1, t1, s2`	`lw t3, 4(s1)`	4
	`addu t2, t2, s2`	`sw t0, 16(s1)`	5
	`addu t3, t3, s2`	`sw t1, 12(s1)`	6
		`sw t2, 8(s1)`	7
	`bne s1, zero, Loop`	`sw t3, 4(s1)`	8

- **Register renaming (here done by the compiler)**
  - Use a different register (instead of `t0`) for each iteration
  - Necessary to eliminate false dependencies due to loop unrolling

Example: Itanium (IA-64)

- **Key features**
  - Many registers
    - 128 general purpose, 128 floating point, 8 branch, 64 condition
    - Register windows with support for spilling into memory
  - EPIC *instruction bundle*
    - Bundle of instructions executed in parallel
    - Fixed format, explicit dependencies
      - **Stop bit:** Indicates if the next bundle depends on the actual bundle
  - Support for speculation and branch elimination
    - Instructions executed, but whether their effects will be permanent is decided later (if not, software needs to rollback)

Example: Itanium (IA-64) (2)

- **Other notable features**
  - *Instruction group*
    - Group of instructions without data dependencies
    - Separated by an instruction with a stop-bit
      - For forward compatibility (increasing the number of pipelines)
  - Instruction bundle structure
    - 5 bits template (execution units used)
    - 3 × 41 bits instructions
    - Most instructions can be conditional, depending on a chosen bit in a predicate register

Dynamic multiple issue

- **Instructions scheduled by pipeline**
  - Exploit instruction-level parallelism, eliminate hazards and stalls
  - Instructions executed **out-of-order**
    - Results committed in-order to maintain programming model
  - Compiler can try to make scheduling easier for the CPU
- **Speculative execution**
  - Execute operation with potentially wrong operands or without guaranteed that the result will be used
  - Rollback mechanism similar to branch prediction

Example: dynamic instruction scheduling

```
01          LOAD R2,A
02          ADD  R1,R2,R3
03          BPOS R1,LAB1   (Taken)
04          LOAD R4,B
05          BNEG R4,LAB2
06 LAB1:    LOAD R4,C
07          ADD  R5,R4,R3
08 LAB2:    SUB  R5,R7,R0
09          BPOS R5,LAB3  (NOT Taken)
10          ADD  R5,R0,R3
```

```
01          LOAD R2,A
06 LAB1:    LOAD R4,C
08 LAB2:    SUB  R5,R7,R0
10          ADD  R5,R0,R3

02          ADD  R1,R2,R3
07          ADD  R5,R4,R3
09          BPOS R5,LAB3  (NOT Taken)

03          BPOS R1,LAB1   (Taken)
```

Out-of-order execution

- **Execution driven by data dependencies**
  - Colliding register names in independent instructions
    - RAW (*Read After Write, true data dependency*)
      - Instruction result used as operand in subsequent instruction
    - WAW (*Write After Write, output dependency*)
      - Two instructions writing in the same register
      - Result correspond to that caused by the instruction executed later
    - WAR (*Write After Read, anti-dependency*)
      - Instruction is changing a register while another instruction is reading it
  - WAW and WAR can be dealt with using register renaming
    - Processor has more physical registers than what is mandated by ISA

Example: WAW elimination

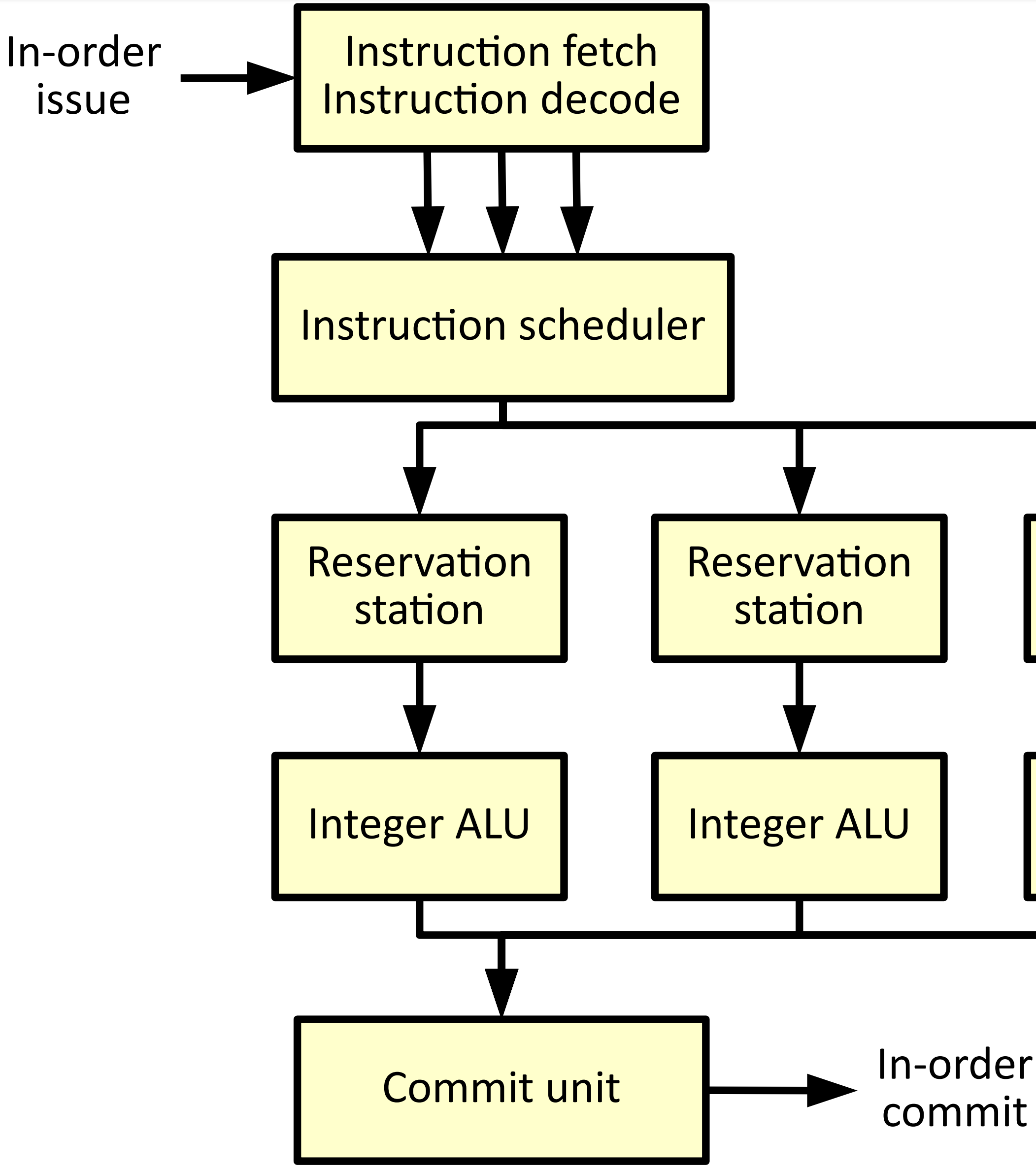
- **Code after reordering**

```
move r3, r7
add r3, r4, r5
move r1, r3
```

- **Code after register renaming**

```
move r3, r7
add fr8, r4, r5
move r1, fr8
```

Dynamic multiple issue (2)



Exceptions in out-of-order pipeline

- More complicated compared to scalar pipeline
  - More difficult to pinpoint the exact place where to interrupt program execution
    - Instructions following the instruction that caused an exception must not change machine state
      - Some of those could have been already executed
    - There must be no earlier unfinished instructions
    - All exceptions caused by earlier instructions must have been handled
  - Precise vs. imprecise exceptions
    - OOE + register renaming first implemented in IBM 360/91 (1969), widespread use in 1990s
    - Cause: imprecise exceptions + higher efficiency only for a small class of programs

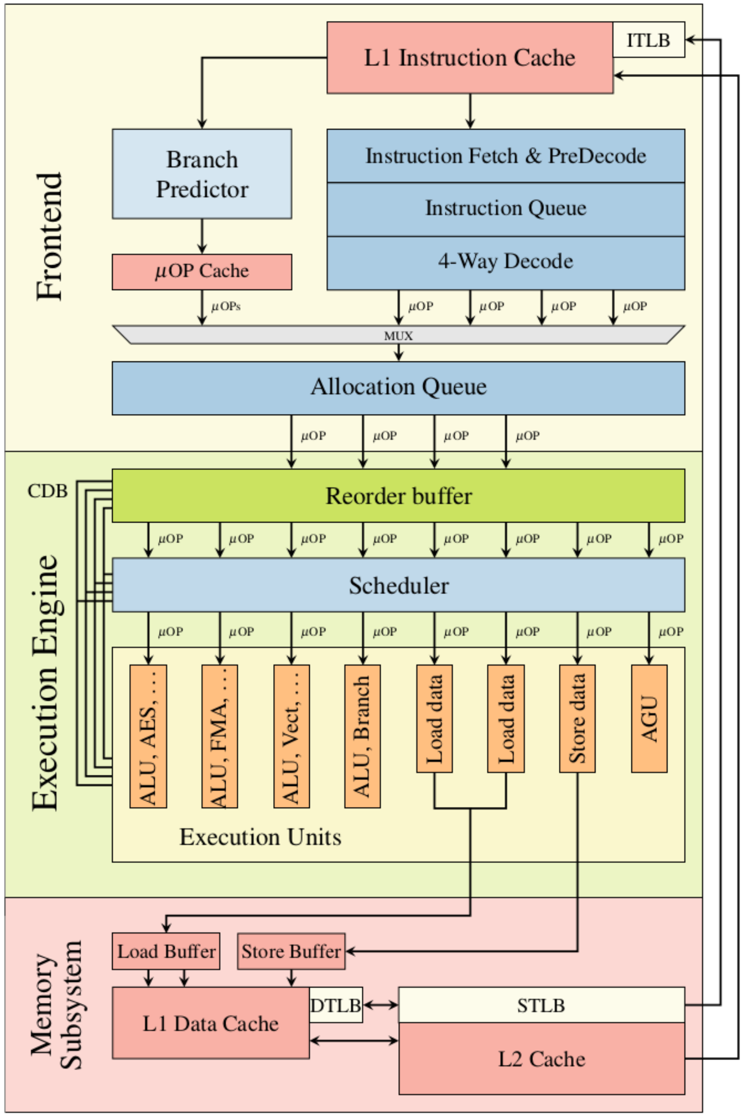
Speculative execution

- **Predicting properties/outcome of instruction**
  - Allows to start executing dependent instructions
  - Extra logic to handle bad speculation
    - In the compiler
      - Extra code generated to “repair” wrong speculations
    - In the processor
      - Speculative results not written back until confirmed
      - Speculatively executed instructions either don’t raise exceptions, or raise special kinds of exceptions

Example: IA-32

- **Intel Pentium Pro … Pentium 4**
  - CISC instruction set implemented using micro-ops on a post-RISC core
    - Instructions split into micro-ops
    - Pipeline executes micro-ops
  - Superscalar, out-of-order, speculative execution (including branch/jump prediction and register renaming)
- **Pentium 4**
  - *Trace cache* to speed up instruction decoding

Example: Skylake



- **Simplified view of the Skylake family microarchitecture**
  - Instructions decoded into micro-ops (µOPs)
  - µOPs executed out-of-order by execution units in the Execution Engine
  - Reorder Buffer responsible for register allocation, register renaming, and instruction retirement
    - Also eliminates register moves and zeroing idioms
  - Scheduler forwards µOPs to execution units depending on availability of data
  - Source: M. Lipp et al. Meltdown.

Core architecture in numbers

	Conroe	Nehalem	Sandy/Ivy Bridge	Haswell (Broadwell)	Skylake/ Kabylake
Allocation queue (decoded insn queue)	?	56 (2x 28)	56 (2x 28)	56	128 (2x 64)
Out-of-order window (reorder buffer)	96	128	168	192	224
Scheduler entries (reservation station)	32	36	54	60 (64)	97
Execution ports	?	6	6	8	8
Integer register file	N/A	N/A	160	168	180
FP register file	N/A	N/A	144	168	168
In-flight loads	32	48	64	72	72
In-flight stores	20	32	36	42	56

Designing an optimal ISA

- **Relative frequency of instructions (IBM 360)**

Group	Fraction
data movement	45,28 %
control	28,73 %
arithmetics	10,75 %
comparisons	5,92 %
logic operations	3,91 %
shifts, rotations	2,93 %
bit operations	2,05 %
I/O operations	0,43 %

Designing an optimal ISA (2)

- **Additional observations (IBM 360)**
  - 56 % immediates in the ±15 range (5 bits)
  - 98 % immediates in the ±511 range (10 bits)
  - 95 % subroutines can be passed arguments in less than 24 bytes
- **Additional observations (DEC Alpha)**
  - Typical program uses only 58 % of the available the instruction set
  - 98 % of instructions implemented in 15 % of firmware (PAL)

Designing an optimal ISA (3)

- **Historical focus**
  - Large instruction set, complex instructions
  - Trying to bridge the gap between assembler and higher-level programming language
- **Current focus**
  - Small instruction set, simple instructions
  - Faster instruction execution, easier to optimize (both at compile time and at runtime)
  - Special-purpose hardware (GPU, FPGA accelerators)

Post-RISC processor

- **CISC and RISC architectures converging**
  - Useful, complex (CISC-like) instructions added to RISC instruction set
  - Superscalar execution
  - Aggressive instruction reordering
    - Out-of-order speculative execution
    - Avoid relying on compiler optimizations
  - New specialized execution units
  - Trying to exploit as much as possible ILP