B35APO: Computer Architectures

Lecture 05. Pipelined Instruction Execution

Pavel Píša
pisa@fel.cvut.cz

Petr Štěpán
stepan@fel.cvut.cz

17. března, 2025

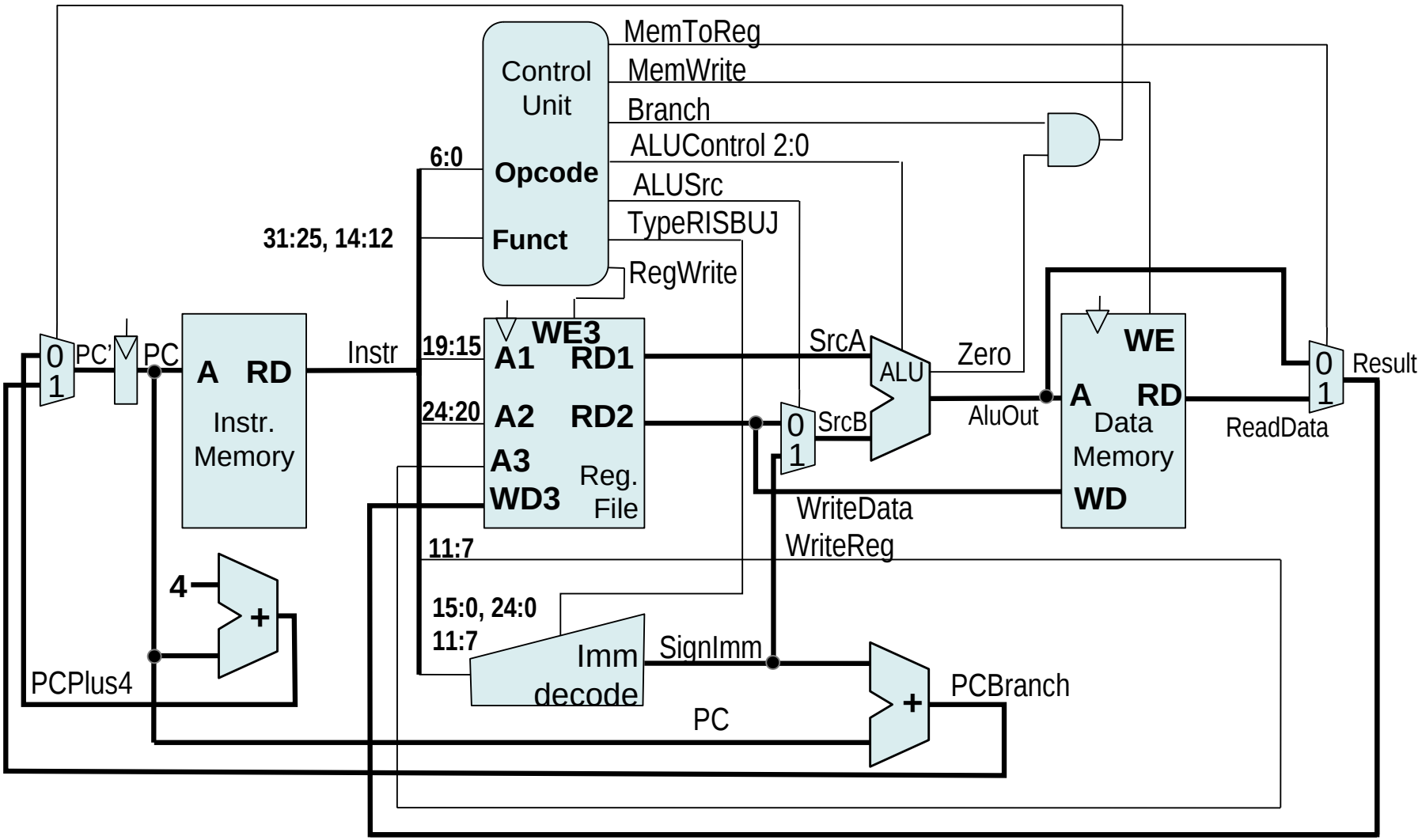
Outline

The Goal of Today's Lecture

- Convert/extend CPU presented in the lecture 3 to the pipelined CPU design to make it faster
- The following instructions are considered for our CPU design: `add`, `sub`, `and`, `or`, `slt`, `addi`, `lw`, `sw`, and `beq`
- Instruction coding stays the same as well

Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
R	fnct7		rs2		rs1	fnct3	rd		opcode
I	imm[11:0]				rs1	fnct3	rd		opcode
S	imm[11:5]		rs2		rs1	fnct3	imm[4:0]		opcode
B	imm [12]	imm [10:5]	rs2		rs1	fnct3	imm[4:1]	imm [11]	opcode
U	imm[31:12]						rd		opcode
J	imm [20]	imm[10:1]		imm [11]	imm[19:12]		rd		opcode

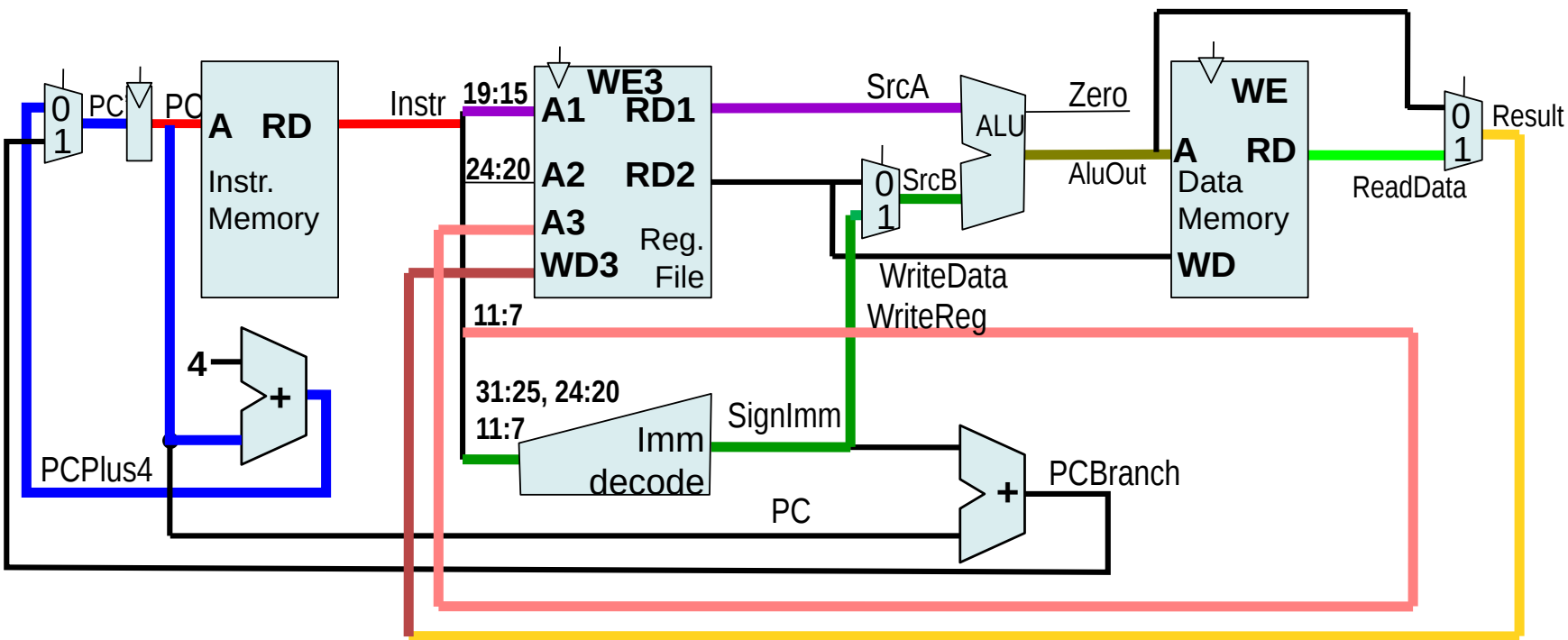
Single Cycle CPU with Memories (from Lecture 3)



The Throughput of a Single-cycle Processor Is Limited

- Maximum instructions per second $IPS = IC/T = IPC_{avk} \cdot f_{clk}$
- Limited by the longest signal path time from the previous registered value to the input of the register (critical path)
- In our design, the limitation is `lw` instruction execution

$$T_c = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$$



Critical Path Limiting Throughput

$f_{clk} = 1/T_C$ where T_C is the period of clock to process one cycle

$T_C = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$

We consider the following times required for signal propagation

$$
\begin{aligned}
t_{PC} &= 30 \text{ ns} \\
t_{Mem} &= 300 \text{ ns} \\
t_{RFread} &= 150 \text{ ns} \\
t_{ALU} &= 200 \text{ ns} \\
t_{Mux} &= 20 \text{ ns} \\
t_{RFsetup} &= 20 \text{ ns}
\end{aligned}
$$

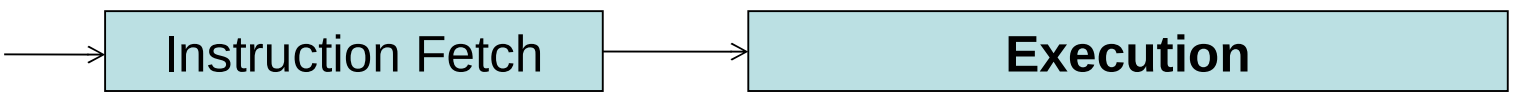
$T_C = 1020\,\text{ns} \rightarrow f_{clkmax} = 980\,\text{kHz}$

$IPS = IC/T = IPC_{avg} \cdot f_{clk}$

$IPS = 1 \cdots 980e3 = 980\,000$ instructions per second

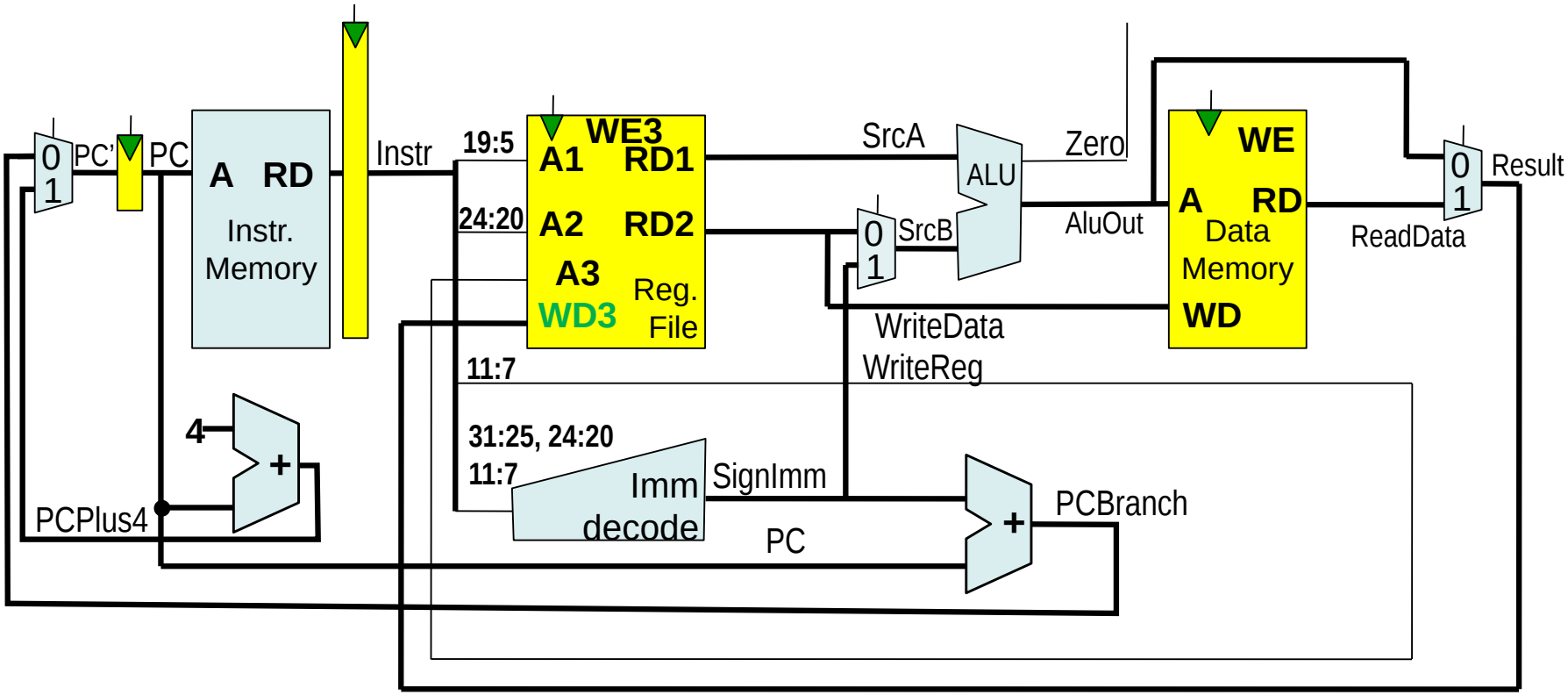
Separate Instruction Fetch and Execution

- Fetching an instruction and execution of data load from memory ($2 \times t_{Mem}$) usually accounts for a significant part of the total cycle time
- It helps to reduce the cycle time if an instruction is already loaded in a previous cycle



1. Instruction fetch and instruction counter increment $PC = PC + 4$
2. The actual execution of the instruction in the following cycle

Non-Pielined Execution with Instructions Prefetching



- ↓ in the figure represents the clock input active on the rising edge of the clock signal

Critical Path in Case of Prefetching

For the previously considered parameters

t_{PC}	$=$	30 ns	t_{Mem}	$=$	300 ns
t_{RFread}	$=$	150 ns	t_{ALU}	$=$	200 ns
t_{Mux}	$=$	20 ns	$t_{RFsetup}$	$=$	20 ns

- We consider $T_{C_{fetch}} = t_{PC} + t_{Mem}$ running in parallel with the execution of the preceding instruction
- $T_{C_{exec}} = t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$

after substituting the values into the expressions

- Consider $T_{C_{fetch}} = 30 + 300 = 330\,\text{ns}$
- $T_{C_{exec}} = 150 + 200 + 300 + 20 + 20 = 690\,\text{ns}$
- $T_{C_{fetch}} < T_{C_{exec}}$ so $T_C = T_{C_{exec}} = 690\,\text{ns}$
- $\rightarrow f_C = 1.45\,\text{MHz} \rightarrow IPS = 1\,450\,000$

Without modifications, the delay of the jump instructions (beq) will occur, see later in the lecture

Outline

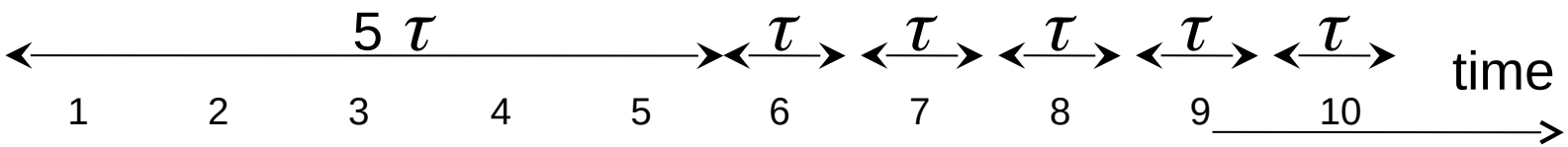
Pipelined Instruction Execution

We will consider splitting instruction execution into five stages.

1. **IF – Instruction Fetch** – setup **PC** value to the memory address input, fetch the instruction and prepare $PC = PC + 4$ in parallel
2. **ID – Instruction Decode** – decode opcode, immediate operand and fetch registers according to `rs1` and `rs2` fields
3. **EX – Instruction EXecution** – execute the requested operation, pass register values and immediate operands to ALU
4. **MEM – MEMory Accesses** – if requested, write data to memory (**sw**) or read (**lw**) them
5. **WB – WriteBack** – write the result to the register file for instructions of register-register and register-immediate class or instruction load (result source is ALU or memory)

Overlapping/Pipelined Sequential Execution of Instructions

IF	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
ID		I1	I2	I3	I4	I5	I6	I7	I8	I9
EX			I1	I2	I3	I4	I5	I6	I7	I8
MEM				I1	I2	I3	I4	I5	I6	I7
ST					I1	I2	I3	I4	I5	I6



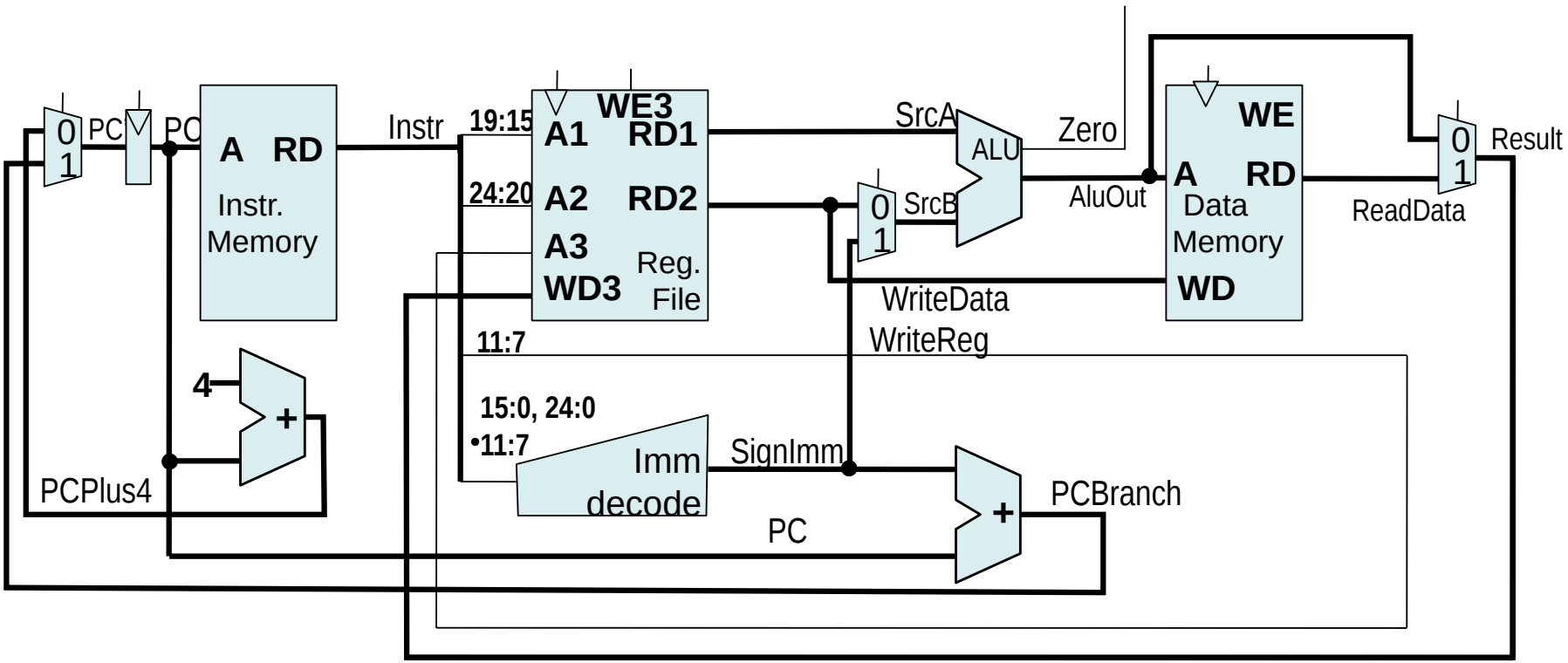
$\tau = max\{\tau_i\}_{i=1}^{k}$, where τ_i is the delay in each stage

The time to execute n instructions in the k-stage pipeline

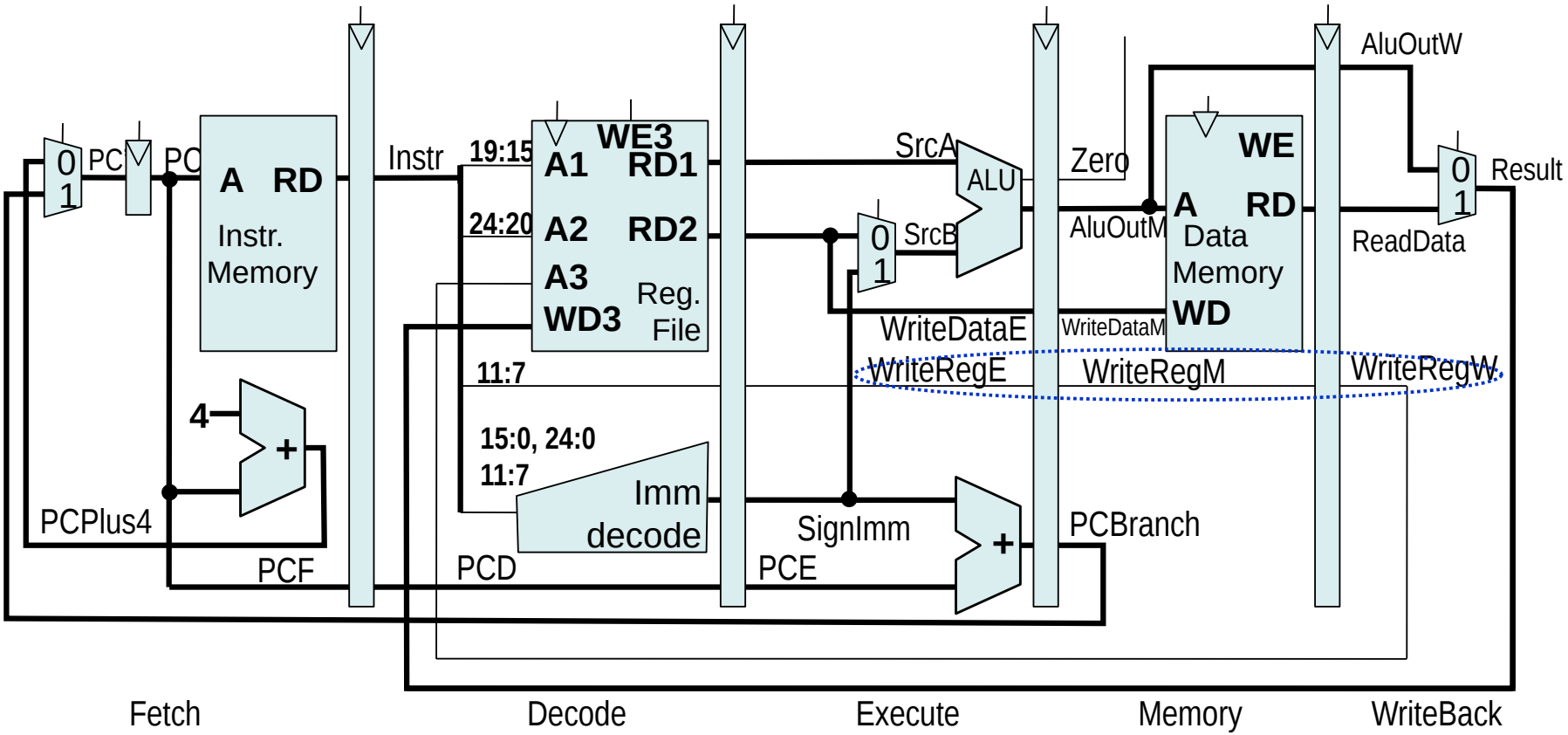
$$T_{k,n} = k \cdot \tau + (n-1)\tau$$

Speedup $S_{k,n} = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{k\tau+(n-1)\tau}$ $\lim_{n\to\infty} S_k = k$

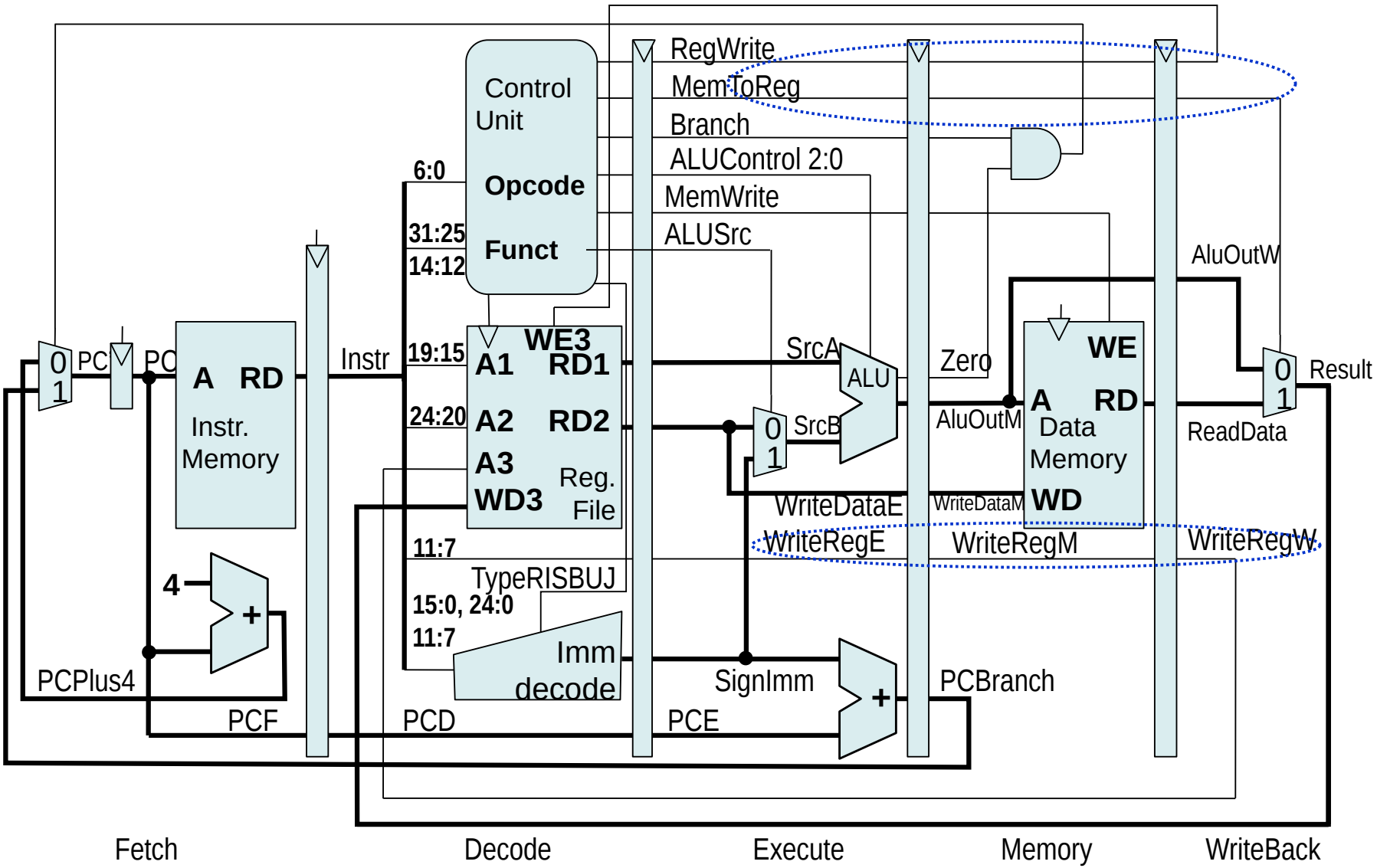
Single Cycle Processor (from Lecture 3) – Datapath



Pipelined Five-Stage Design – Datapath



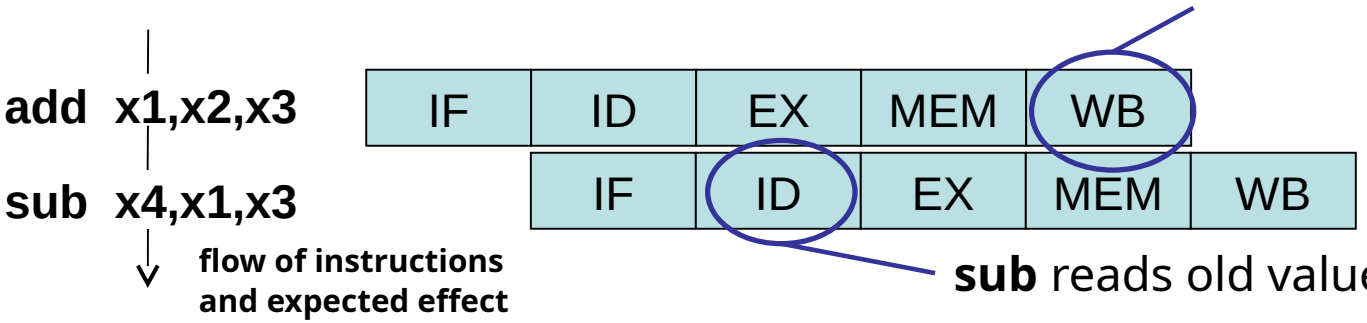
Pipelined Five-Stage Design Including Control Unit



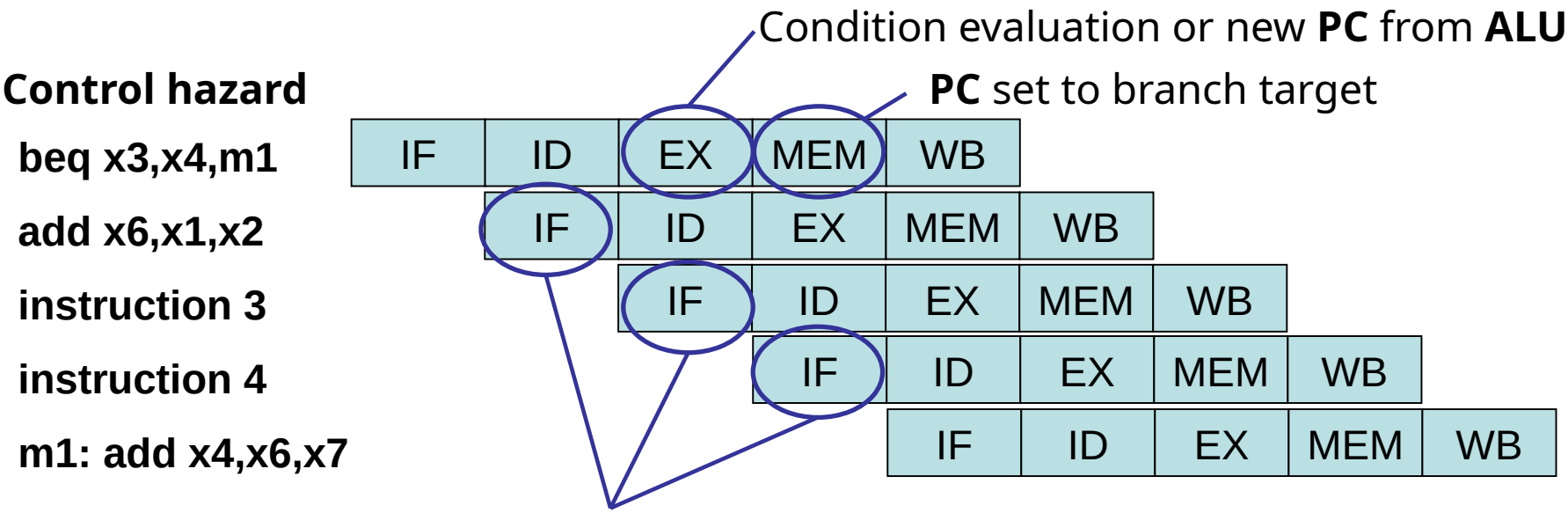
Outline

Semantics Violations – Data and Control Hazards

Data hazard

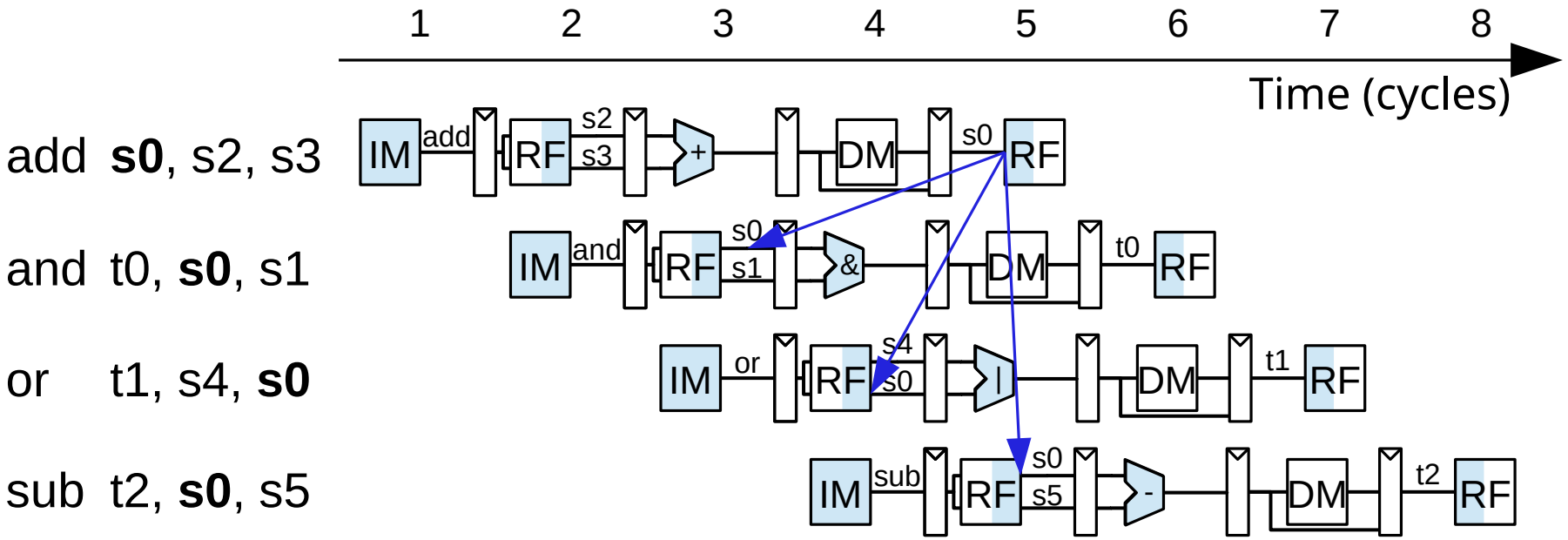


Control hazard



Should be these instructions fetched (and executed then)?

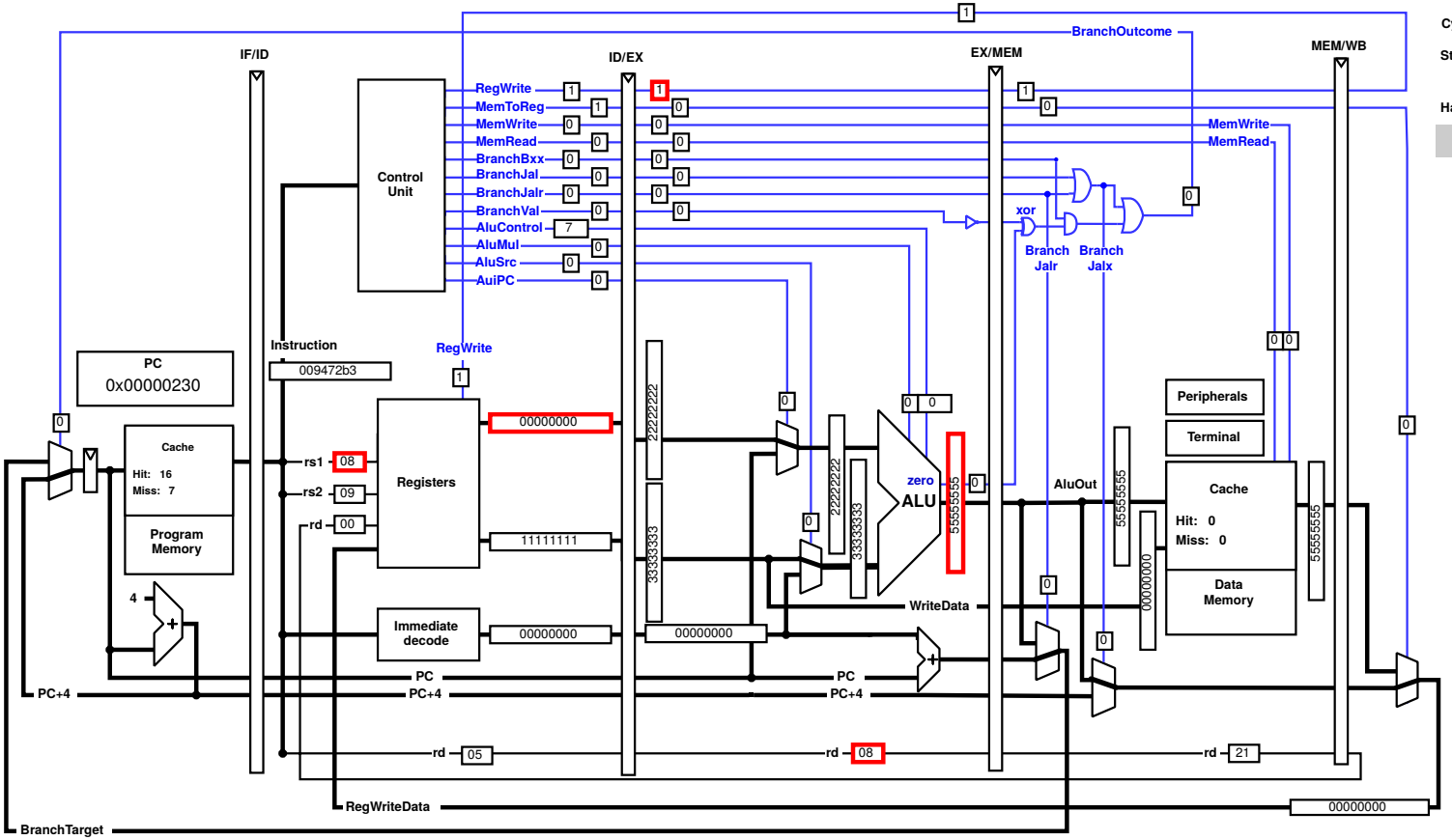
Cause of Data Hazards



- The register file is accessed from two stages (Decode, WriteBack) – writing is done in the first half of the cycle, reading in the second half
  $\Rightarrow$ there is no hazard for sub **s0** input operand
- Read-After-Write (RAW) hazard occurs in instructions **and** and **or** when reading **s0** in cycle 3 and 4
- How can such hazard be prevented without pipeline throughput degradation?

Data Hazard in the "and" Instruction in the QtRVSim

or x6, x20, **x8**	and x5, **x8**, x9	add **x8**, x18, x19	addi x21, x21, 1365	nop
or t1, s4, **s0**	and t1, **s0**, s1	add **s0**, s2, s3	addi s5, s5, 1365	nop



Cycles: **23**

Stalls: **11**

Hazard Unit:

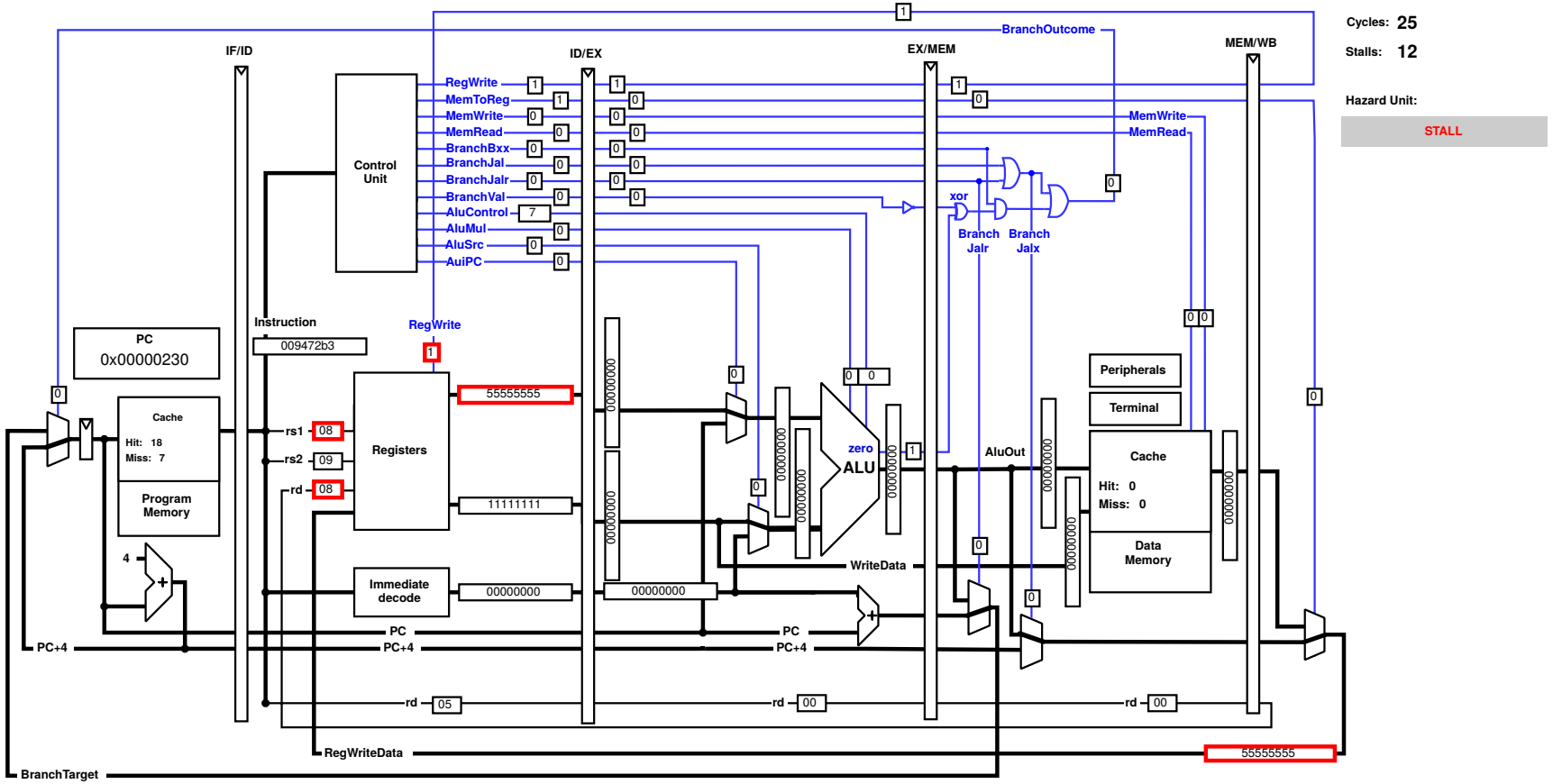
NORMAL

QtRvSim `https://github.com/cvut/qtrvsim`

`https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S`

QtRVSim – Resolve Data Hazard by Stall

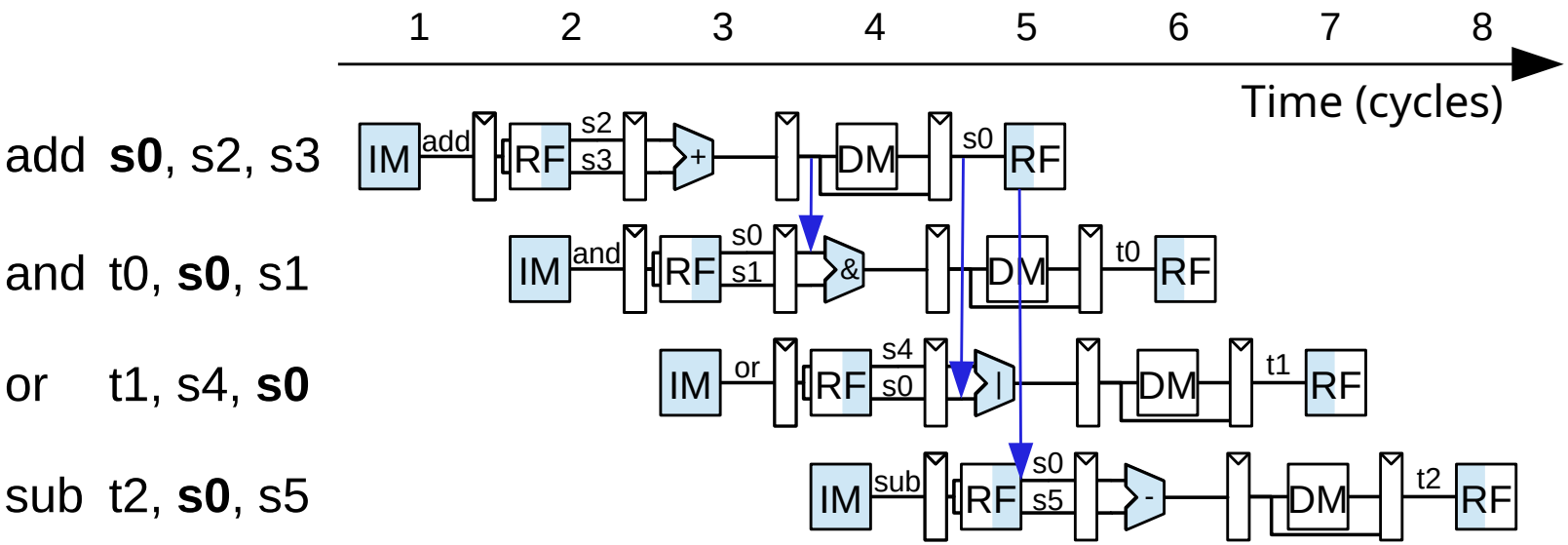
or x6, x20, **x8**	and x5, **x8** , x9	nop	nop	add **x8** , x18, x19
or t1, s4, **s0**	and t1, **s0** , s1	stall	stall	add **s0** , s2, s3



QtRvSim `https://github.com/cvut/qtrvsim`

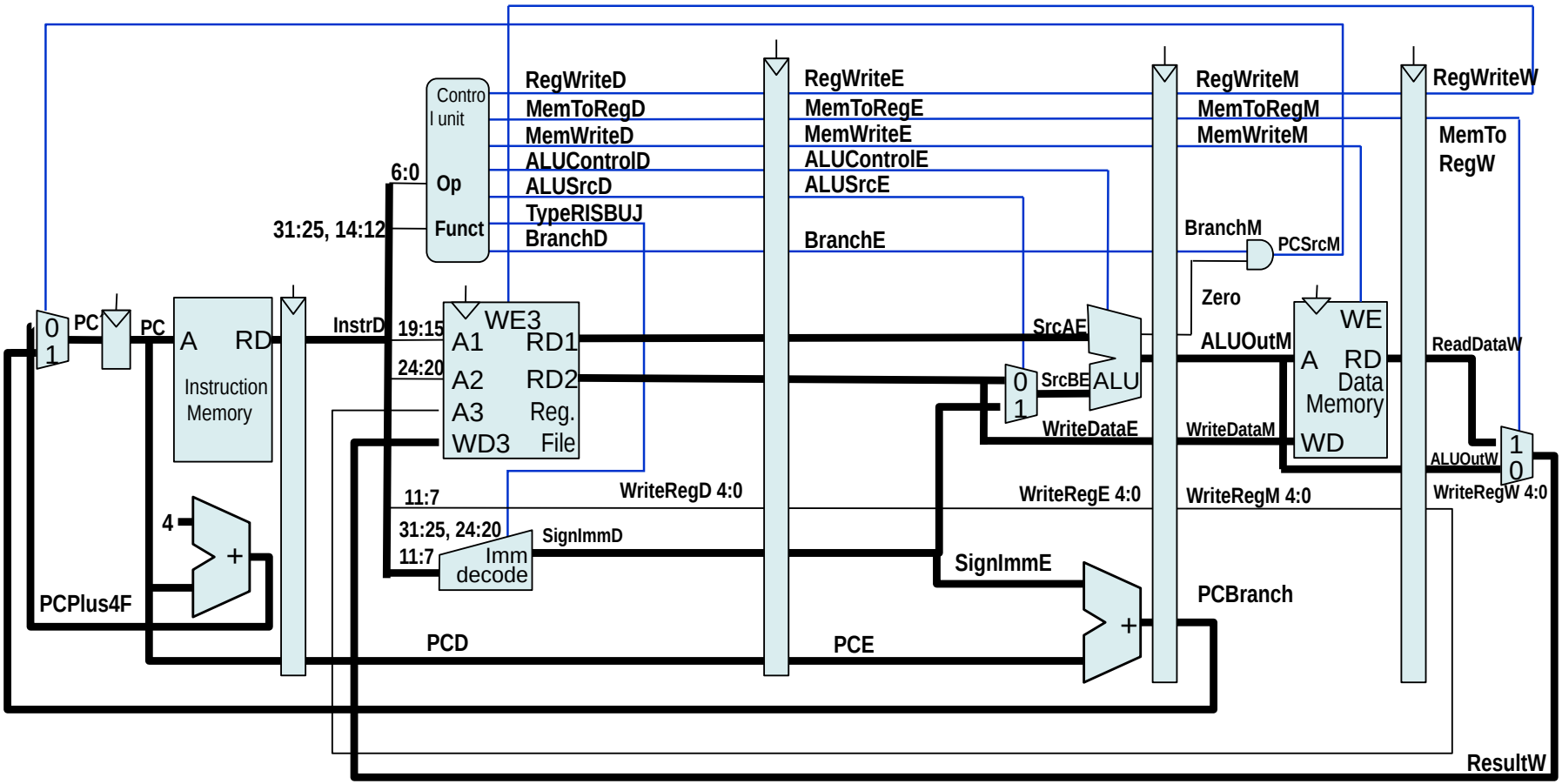
`https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S`

Data Hazard Resolution by Forwarding



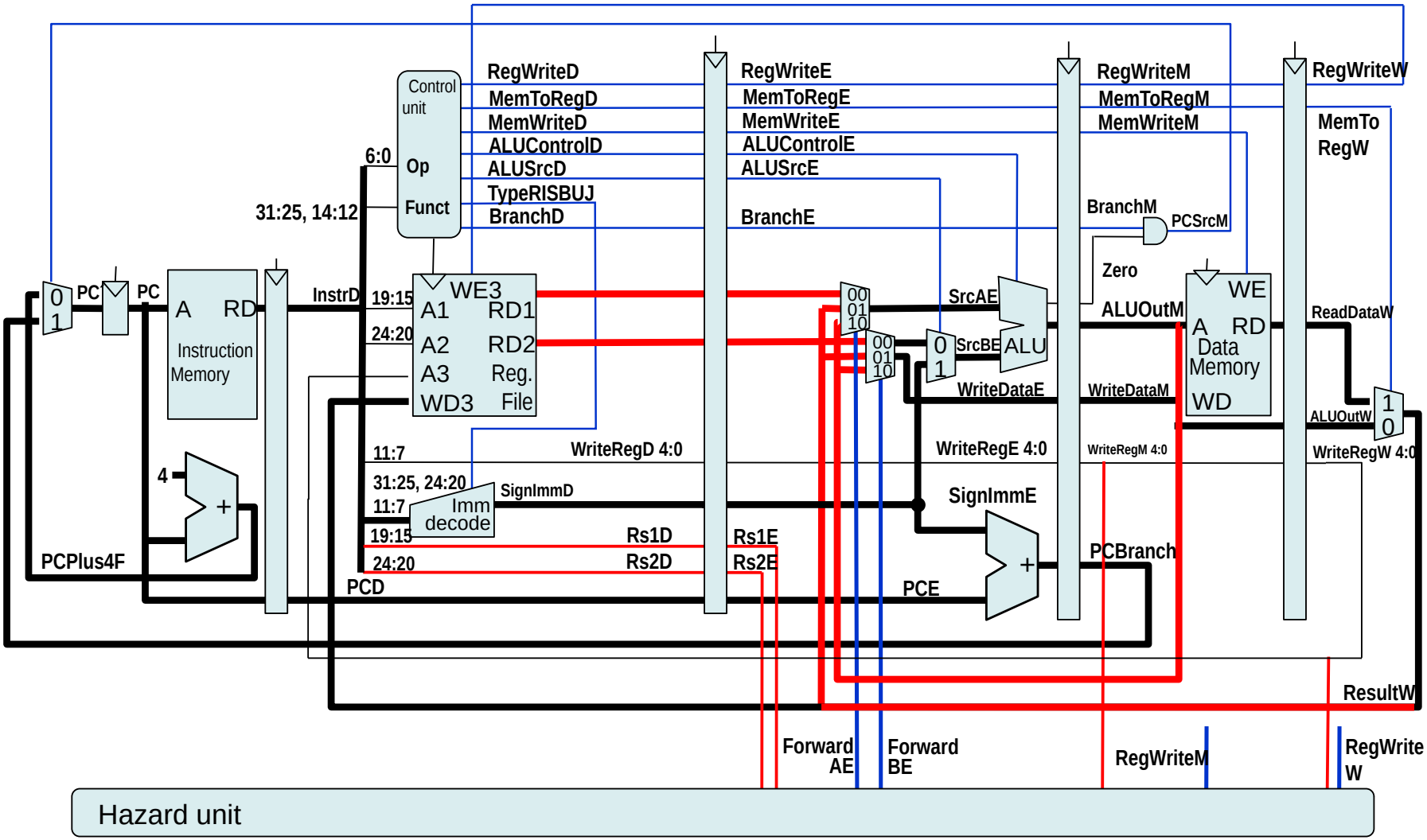
- If a result is available (computed) before subsequent instruction(s) requires the value then data hazard can be avoided by forwarding
- Data hazard occurs in the considered pipeline design if a source register (**rs1**, **rs2**) in the **EX** stage corresponds to the target register in **MEM** or **WB** (except X0/zero)
- The register numbers are fed to the Hazard Unit (HU)

Recap of Previous Design and Prepare It for Forwarding



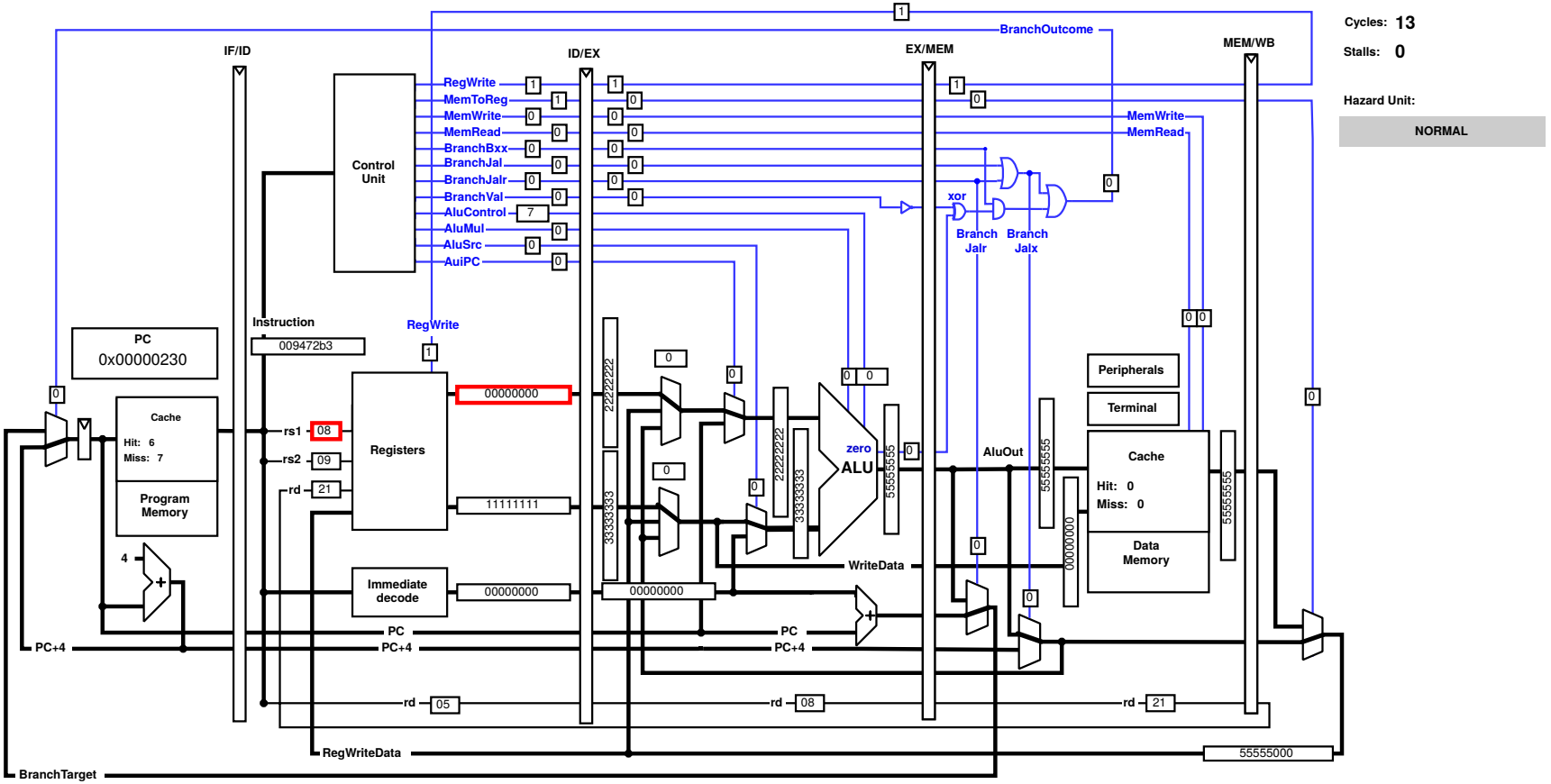
You need to know the previous **A1** (**rs1**) and **A2** (**rs2**) in **EX**. The **RegWrite** signals from **MEM** and **WB** has to be monitored as well to check that the register specified by **WriteReg** in **MEM** and **WB** is indeed being written.

Processor Design with MEM and WB to EX Forwarding



QtRvSim – Data Hazards Resolved by Forwarding

or x6, x20, **x8**	and x5, **x8** , x9	add **x8** , x18, x19	addi x21, x21, 1365	lui x21, 0x55555
or t1, s4, **s0**	and t1, **s0** , s1	add **s0** , s2, s3	addi s5, s5, 1365	lui s5, 0x55555

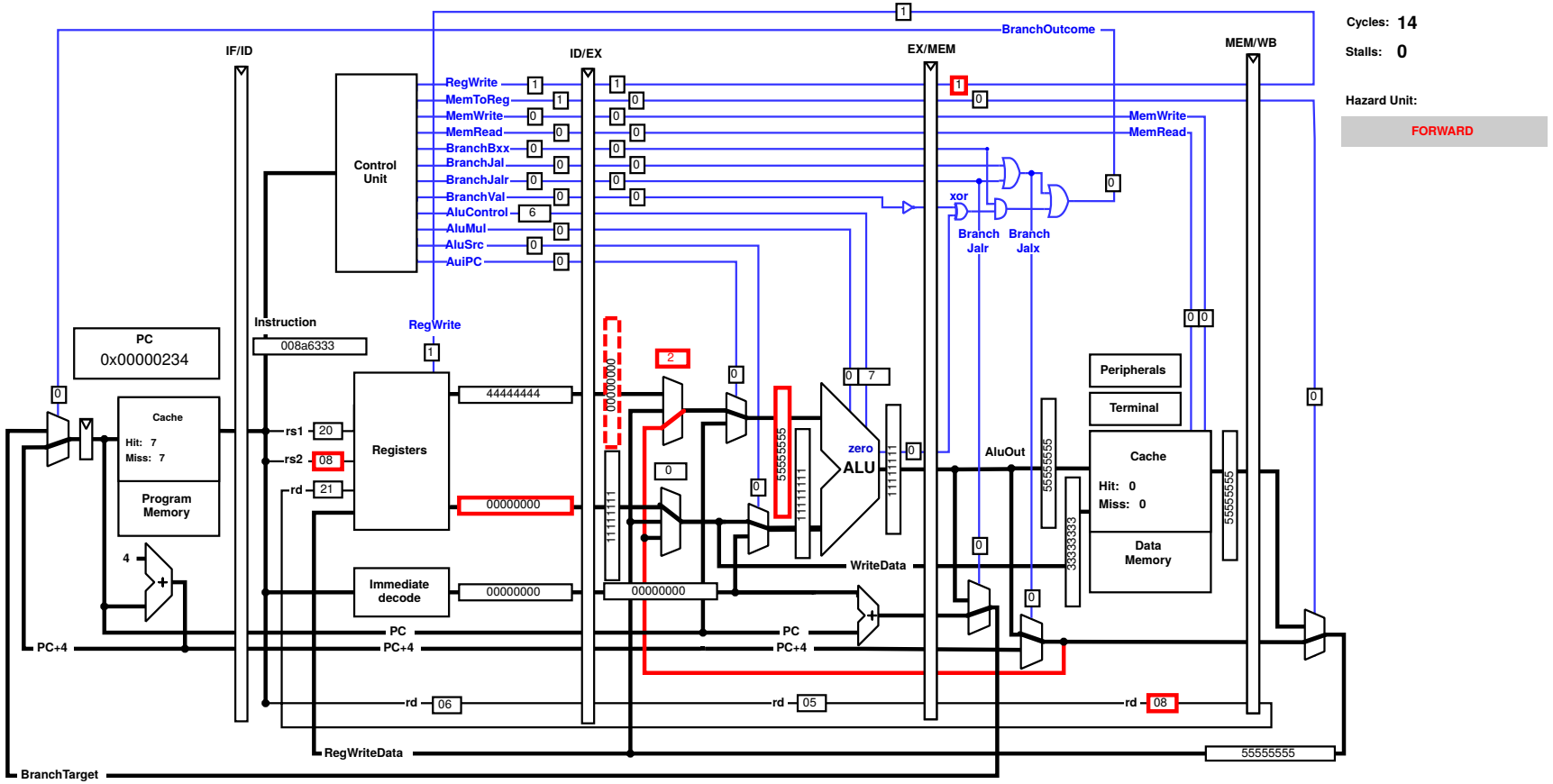


QtRvSim `https://github.com/cvut/qtrvsim`

`https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S`

Forwarding the "and" Instruction rs1 Input from MEM

sub x7, **x8**, x21	or x6, x20, **x8**	and x5, **x8** , x9	add **x8** , x18, x19	addi x21, x21, 1365
sub t2, **s0**, s5	or t1, s4, **s0**	and t1, **s0** , s1	add **s0** , s2, s3	addi s5, s5, 1365

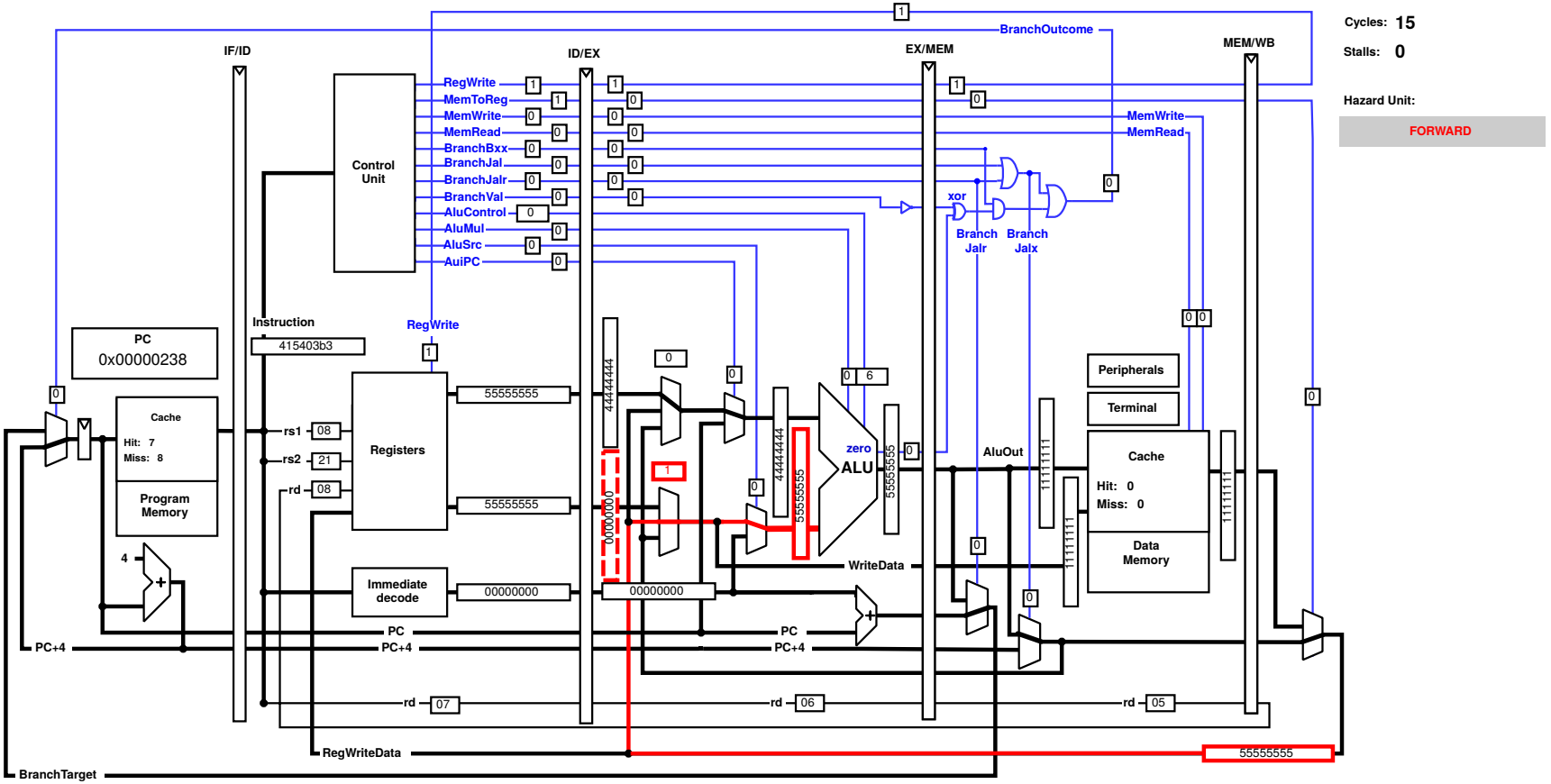


QtRvSim `https://github.com/cvut/qtrvsim`

`https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S`

Forwarding the ”or” Instruction rs2 Input from WB

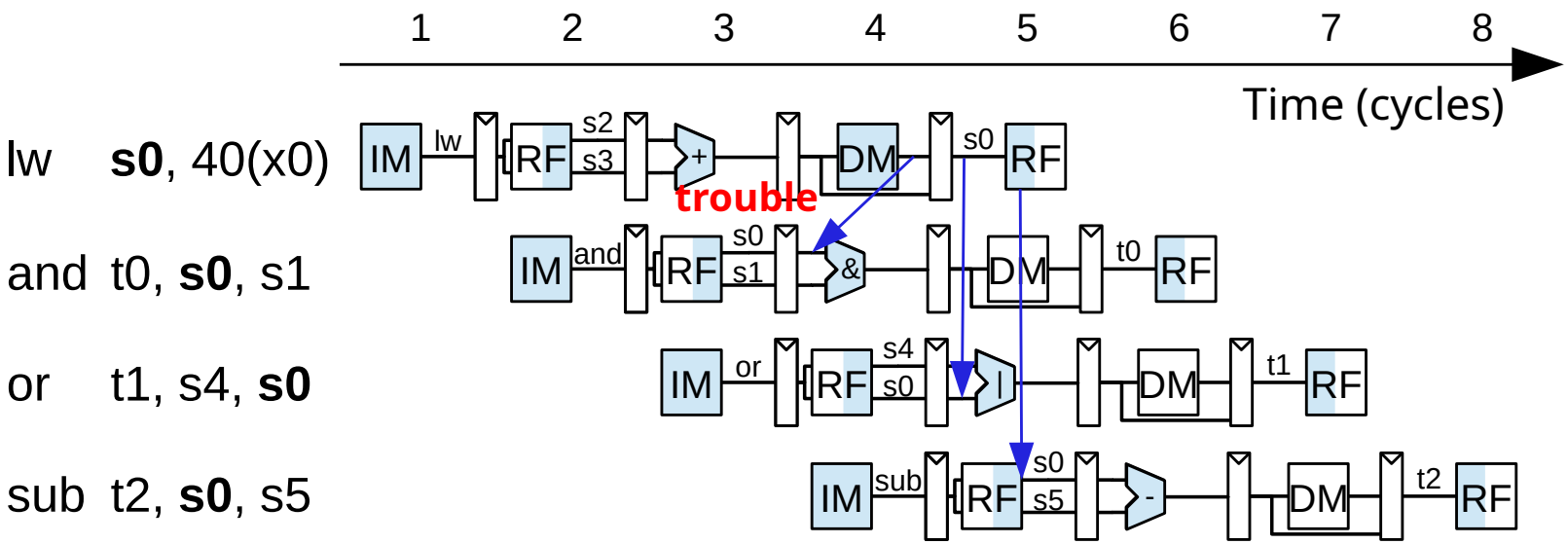
unknown	sub x7, **x8** , x21	or x6, x20, **x8**	and x5, **x8** , x9	add **x8** , x18, x19
unknown	sub t2, **s0** , s5	or t1, s4, **s0**	and t1, **s0** , s1	add **s0** , s2, s3



QtRvSim `https://github.com/cvut/qtrvsim`

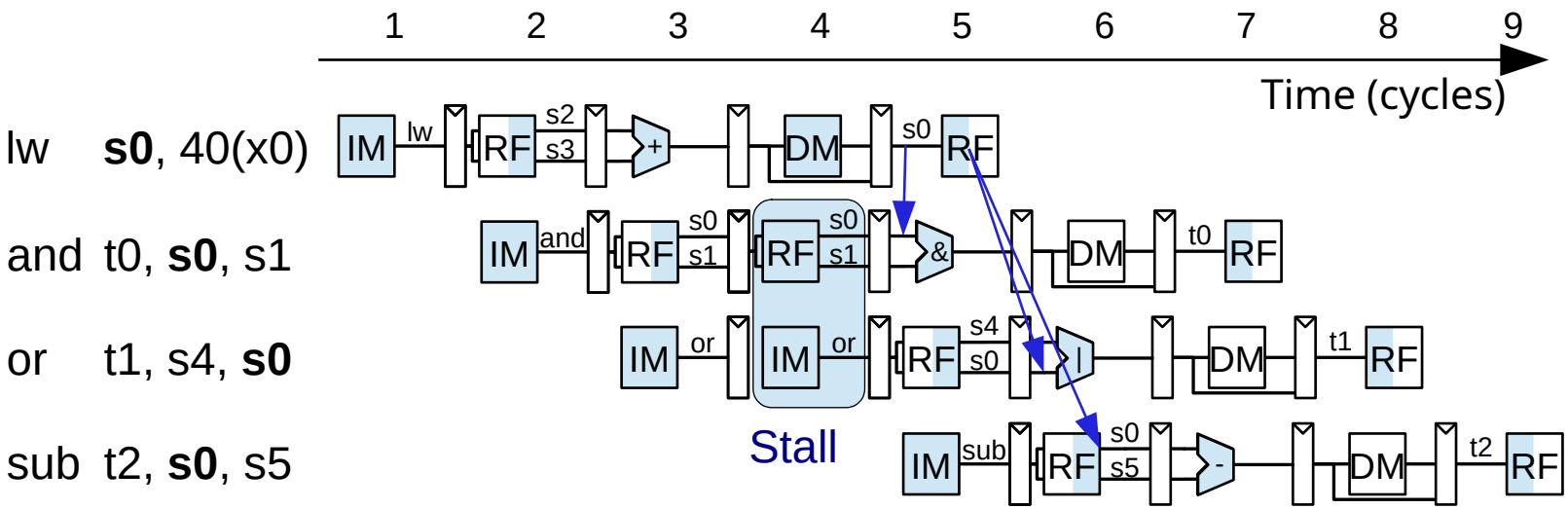
`https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S`

Persisting Problem in the "lw" Case – Solved by Stalling



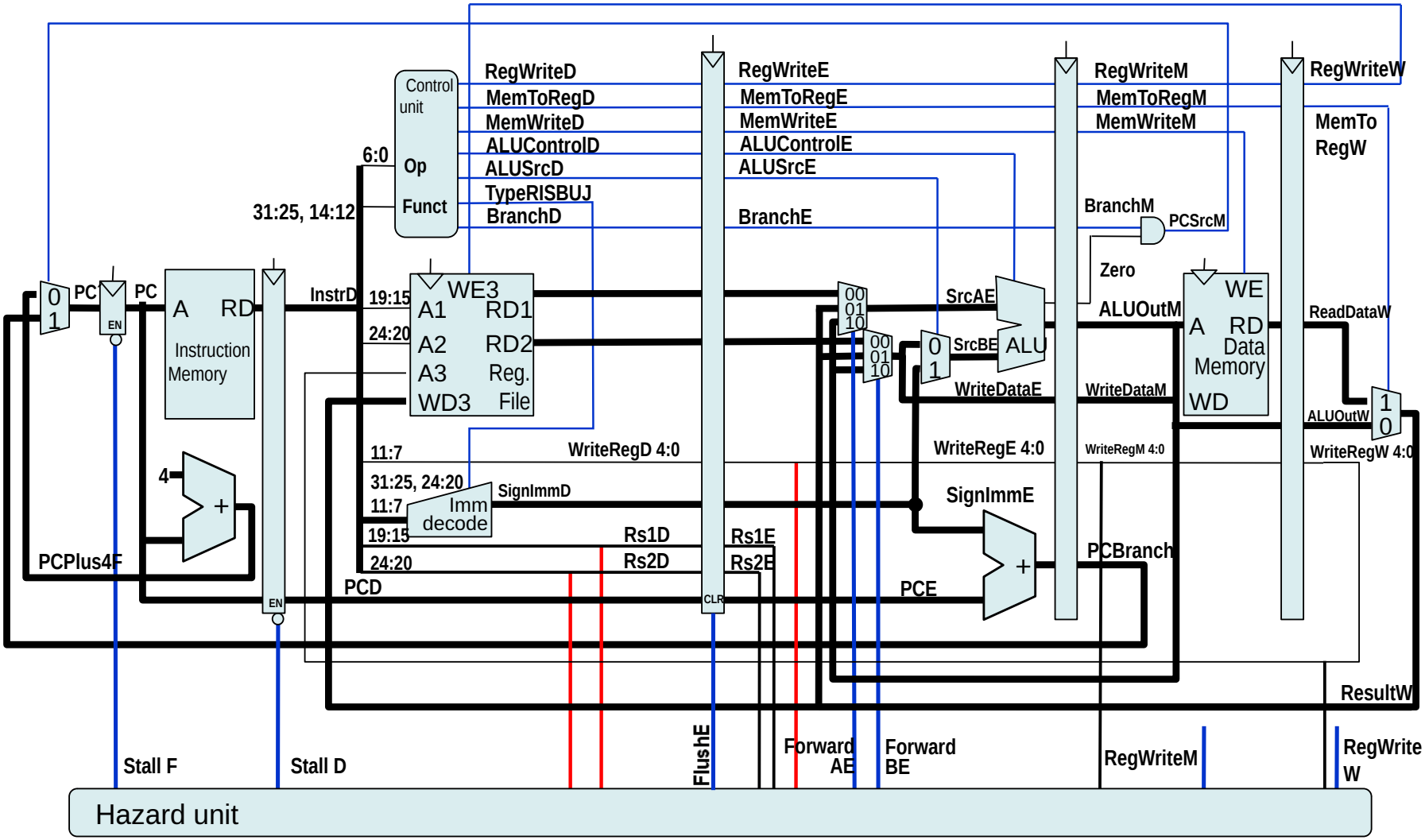
- If subsequent instructions require result before it is available in CPU then the pipeline has to be stalled (**stall** and **nop** inserted/interstage flush)
- Stalling resolves hazards but degrades throughput
- Stages preceding the stage affected by the hazard are suspended until all results required by subsequent instructions are available then results are forwarded to all locations which required their values

Data Hazard Resolution by Stalling



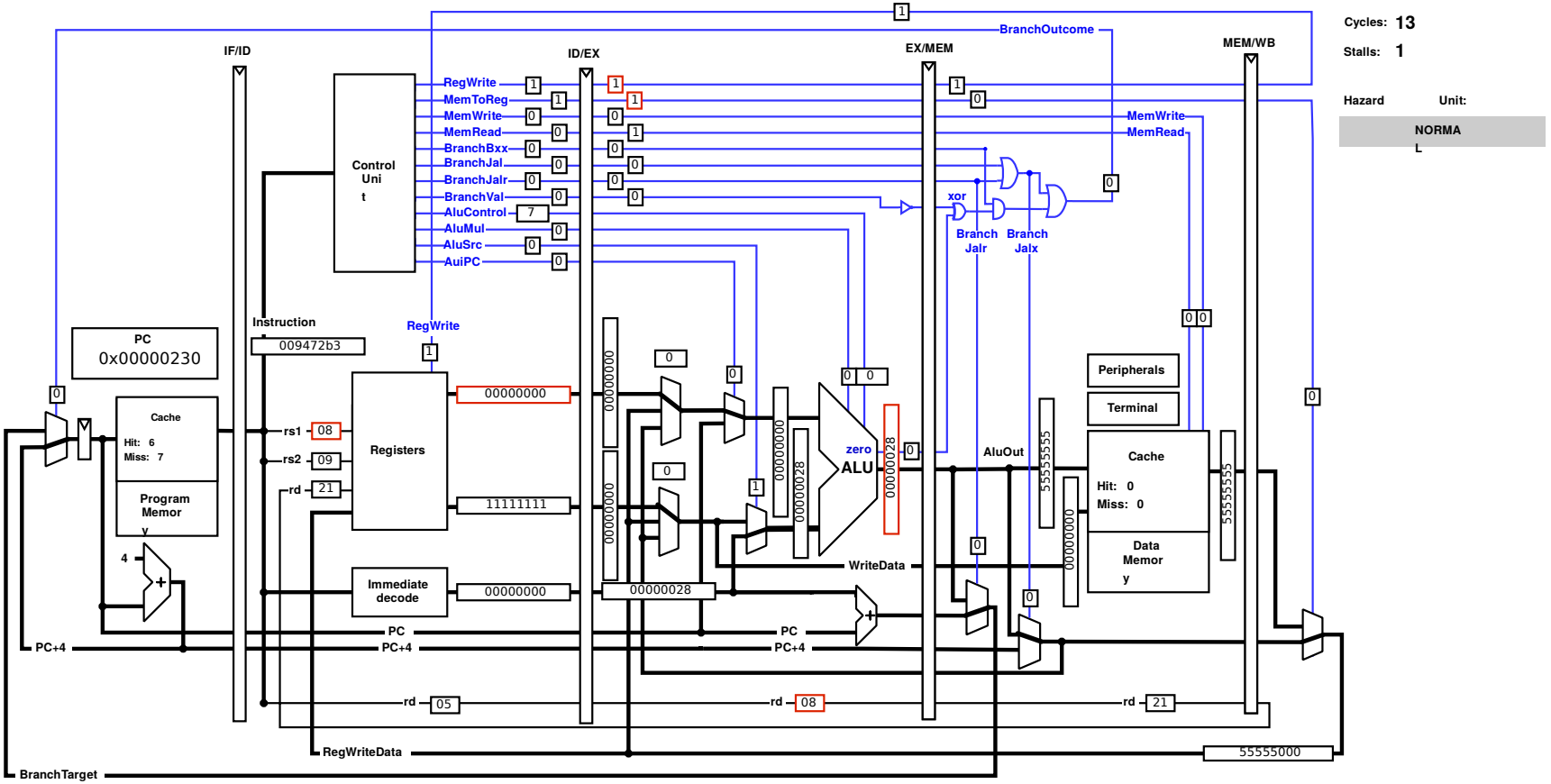
- Stalling is realized by holding the contents of the interstage registers (by gating their clock or blocking their latch enable signals)
- Results from stage(s) with missing data/dependencies are discarded, coresponding instruction(s) memory and register write enable as well as other signals are reset (held inactive)
- Both is achieved by introduction of control signals to hold and/or reset inter-stages registers

Avoid Remaining Data Hazards by Stalling – CPU Diagram



QtRvSim – Data Hazard in "lw" Avoided by Stalling

or x6, x20, **x8**	and x5, **x8**, x9	lw **x8**, 40(x0)	addi x21, x21, 1365	lui x21, 0x55555
or t1, s4, **s0**	and t0, **s0**, s1	lw **s0**, 40(zero)	addi s5, s5, 1365	lui s5, 0x55555

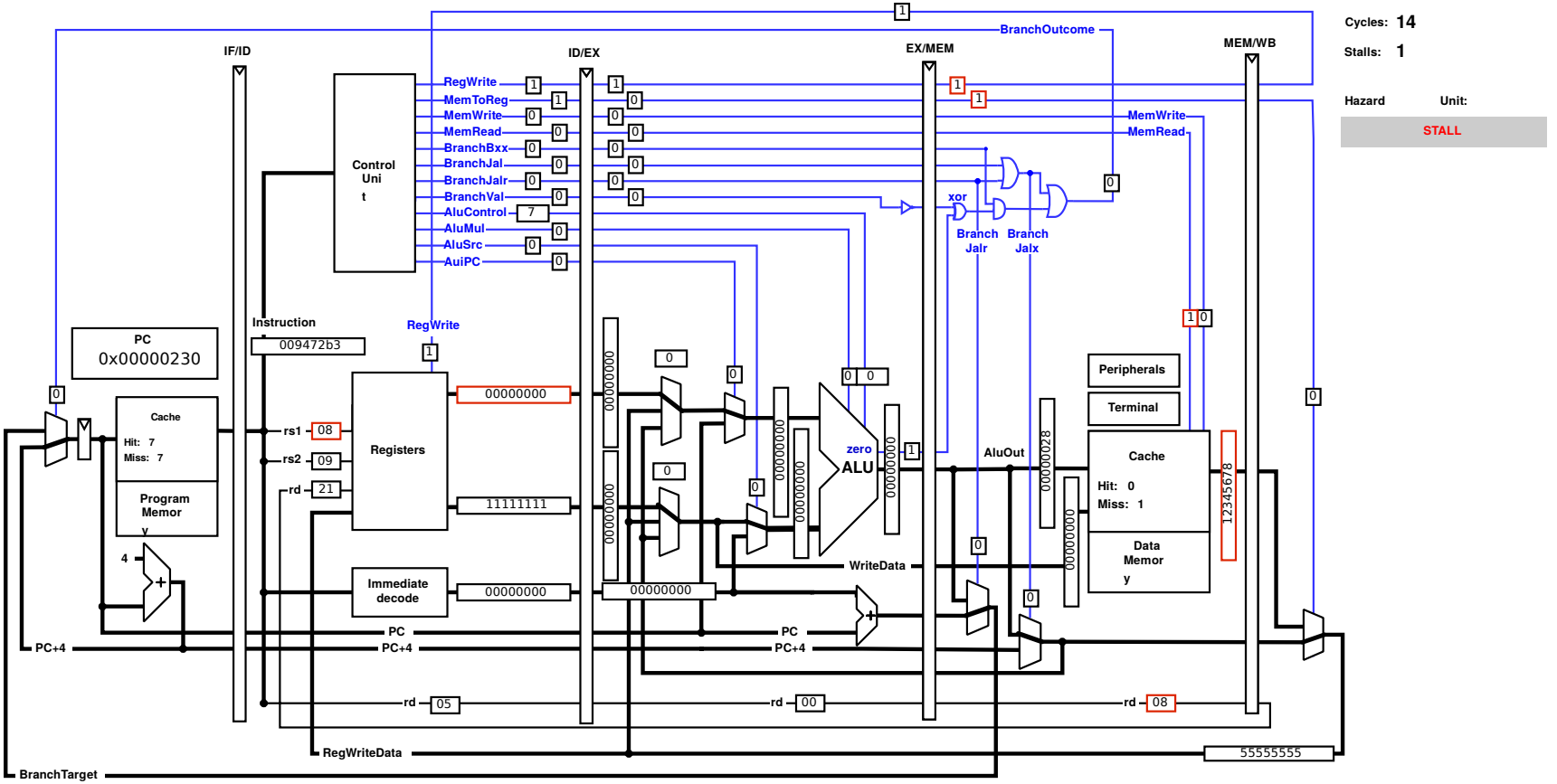


QtRvSim `https://github.com/cvut/qtrvsim`

`https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/lw-hazards.S`

QtRvSim – Data Hazard in "lw" Avoided by Stalling

or x6, x20, **x8**	and x5, **x8**, x9	nop	lw **x8**, 40(x0)	addi x21, x21, 1365
or t1, s4, **s0**	and t0, **s0**, s1	stall	lw **s0**, 40(zero)	addi s5, s5, 1365

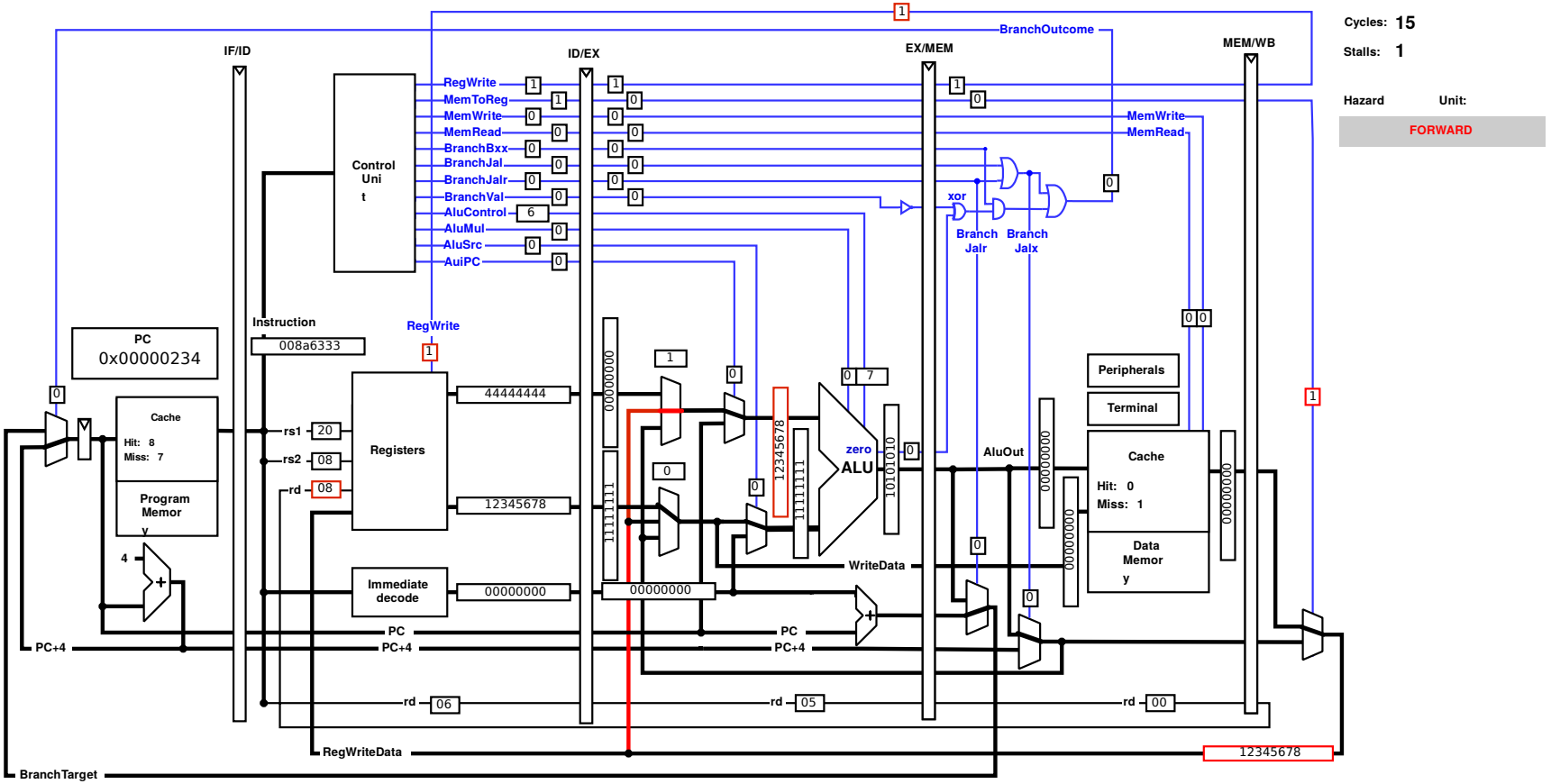


QtRvSim `https://github.com/cvut/qtrvsim`

`https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/lw-hazards.S`

QtRvSim – Data Hazard in "lw" Avoided by Stalling

sub x7, **x8**, x21	or x6, x20, **x8**	and x5, **x8**, x9	nop	lw **x8**, 40(x0)
sub t2, **s0**, s5	or t1, s4, **s0**	and t0, **s0**, s1	stall	lw **s0**, 40(zero)

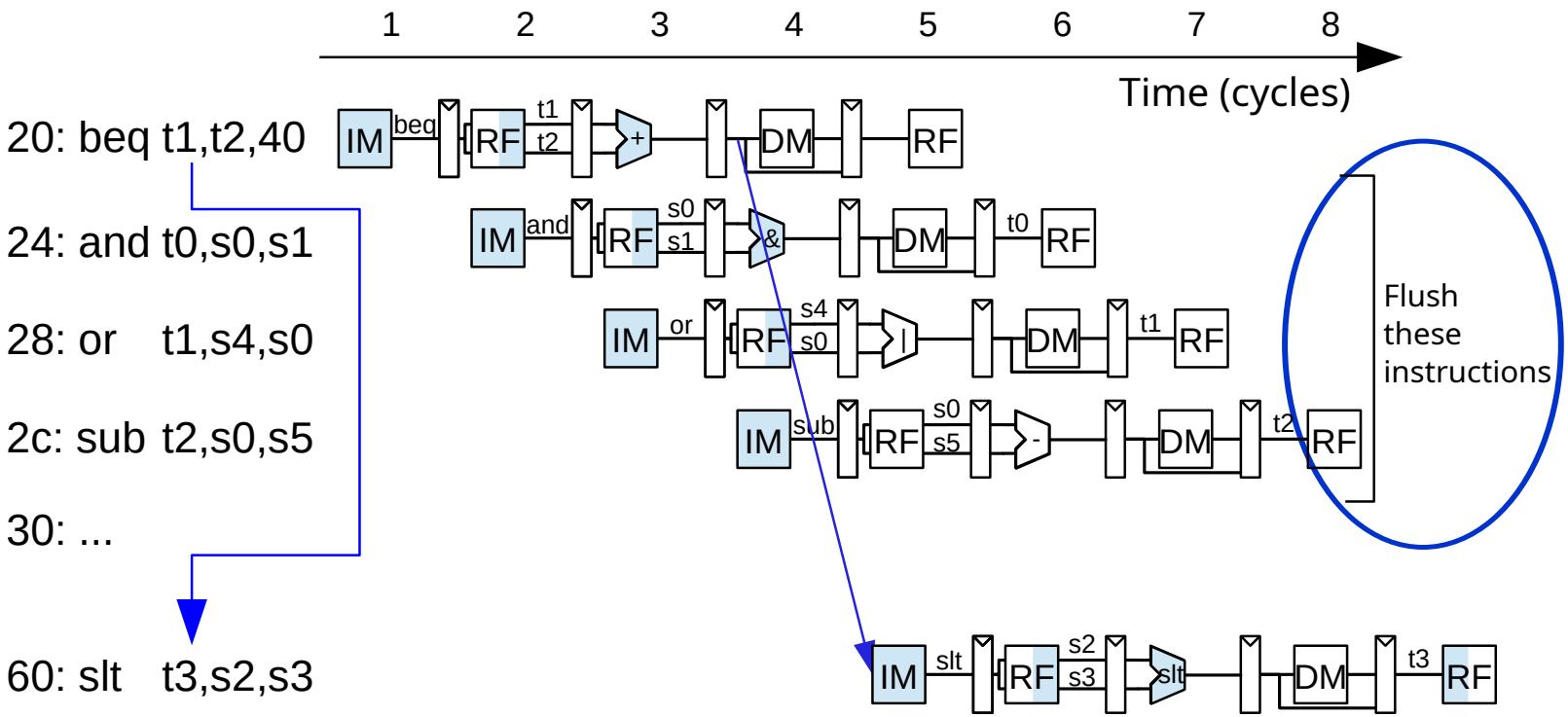


QtRvSim `https://github.com/cvut/qtrvsim`

`https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/lw-hazards.S`

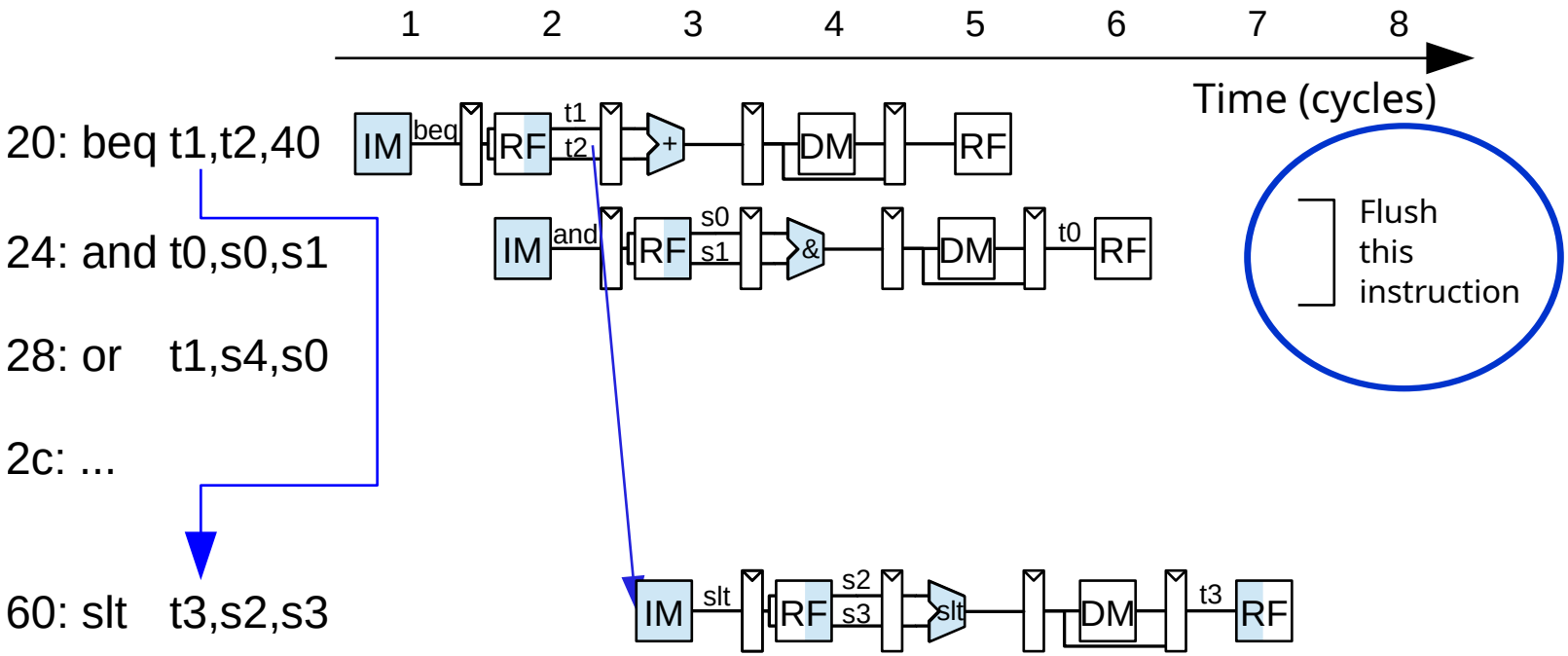
Outline

Control Hazards (Branch and Jump Instructions)



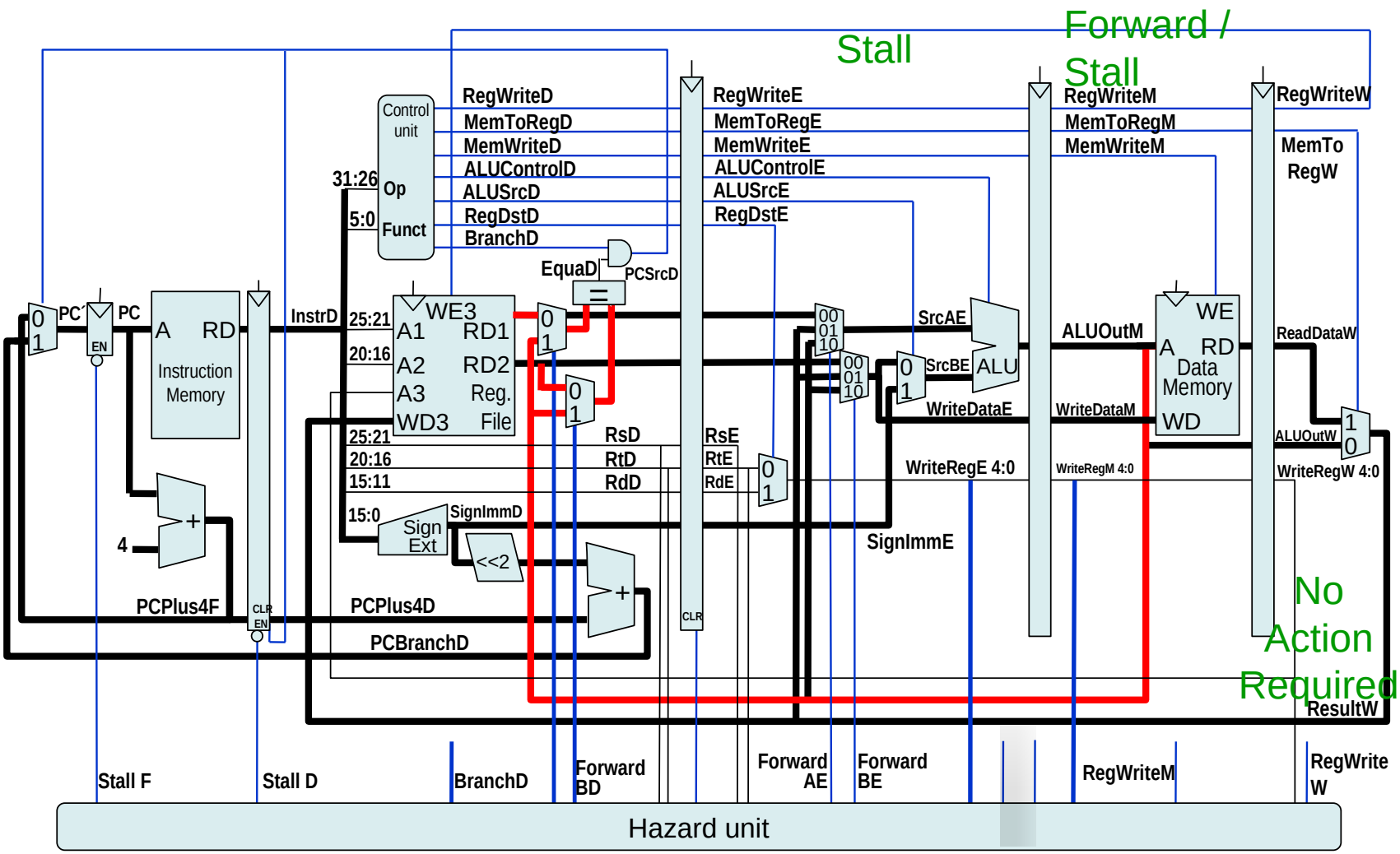
- The result is not known until the fourth cycle, then it is loaded to the PC and so the first instruction at the jump target address may not be loaded until cycle five.
- Jumps in this way represent a major limitation on the time usage of the execution units in the pipeline

Alternative, Move Branch Decision Earlier

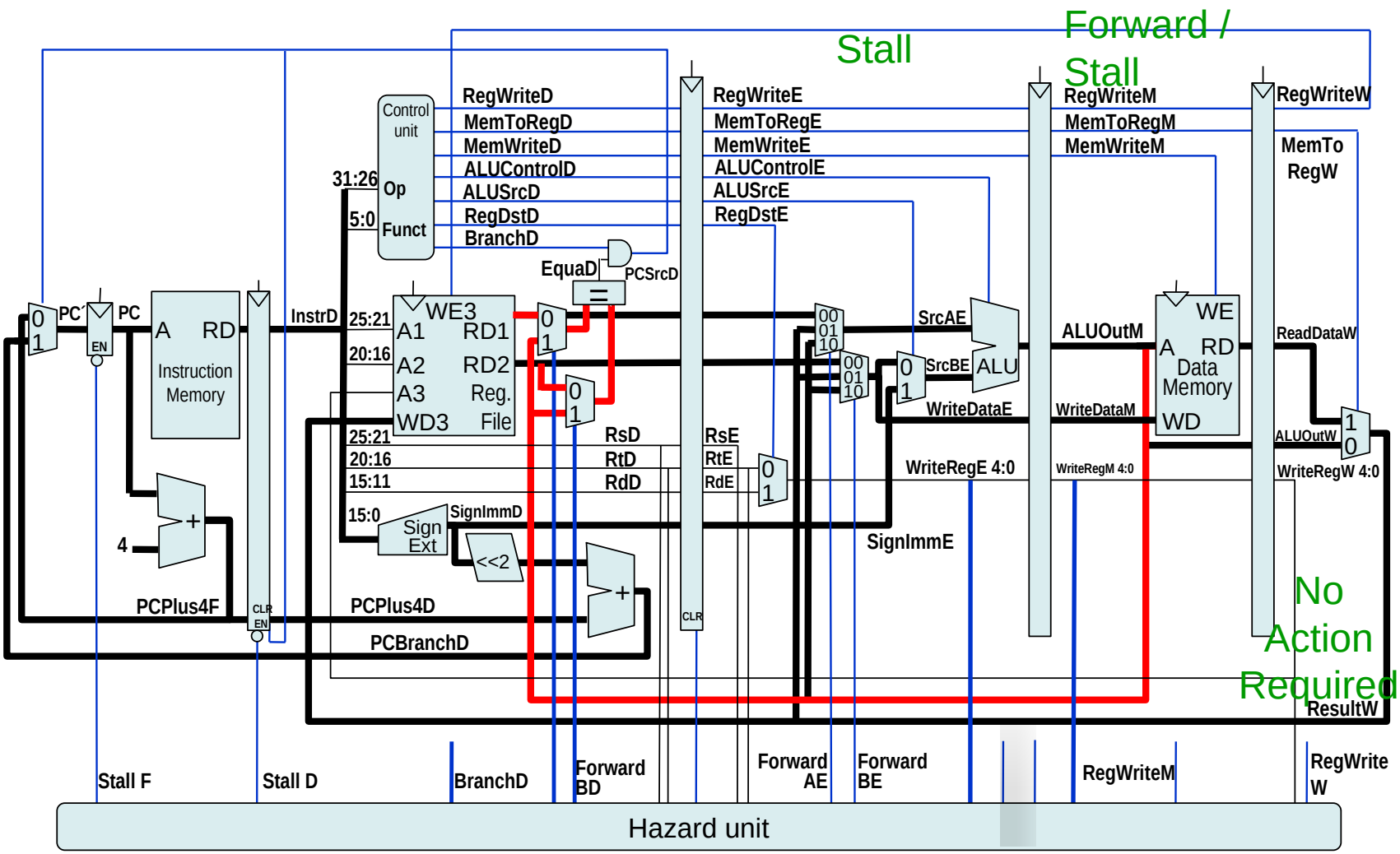


- This causes structural hazard, **ALU** is needed in both **EX** and **ID**.
- Structural hazards can be solved by duplicating given component (ALU in the respective case). A complete adder, subtractor or comparison takes a long time, a solution may be to restrict the branch conditions to equality/non-equality only (just xor between operands led to bitwise nor)

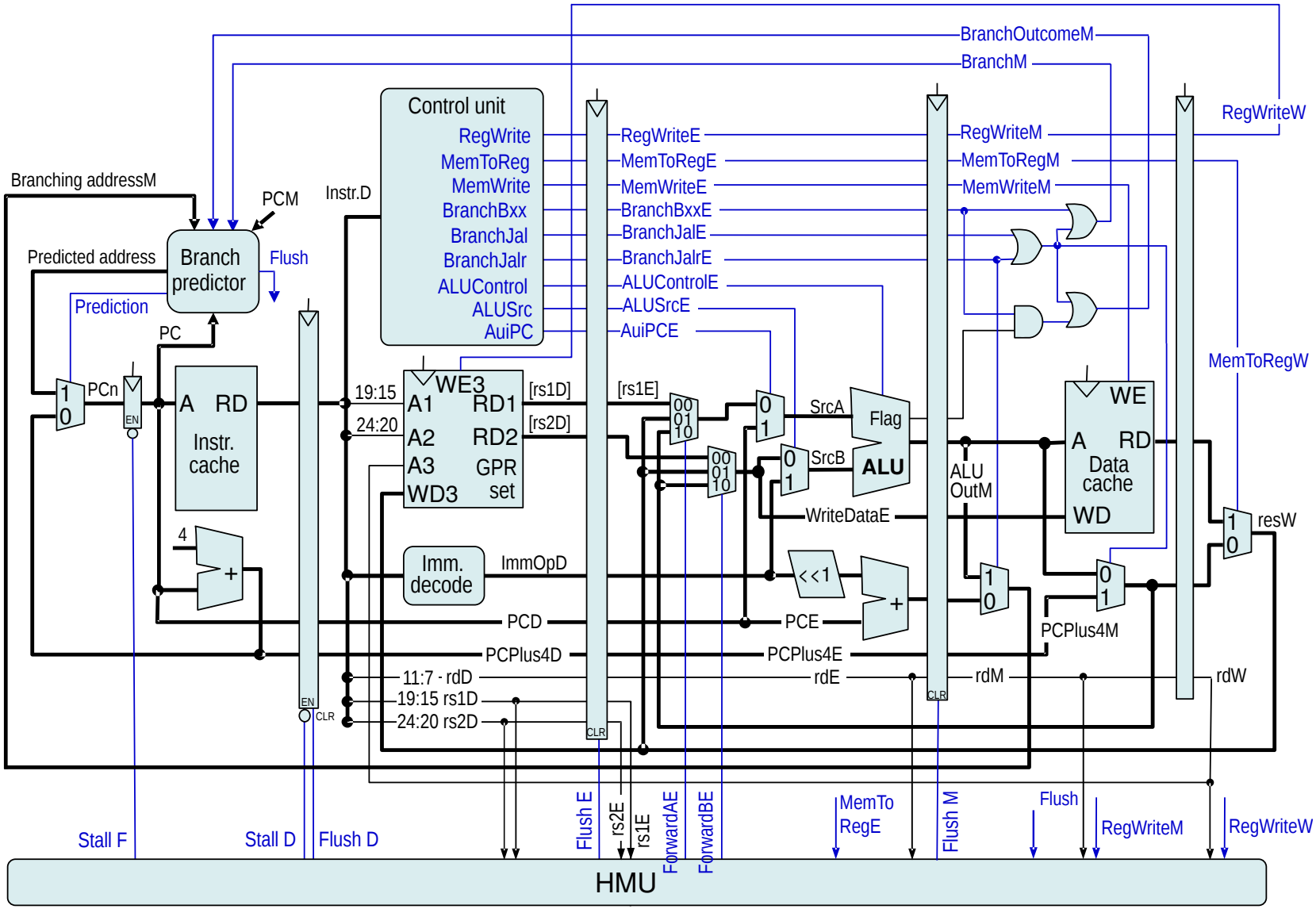
MIPS Resolve Control Hazards by Early Evaluate and Flush



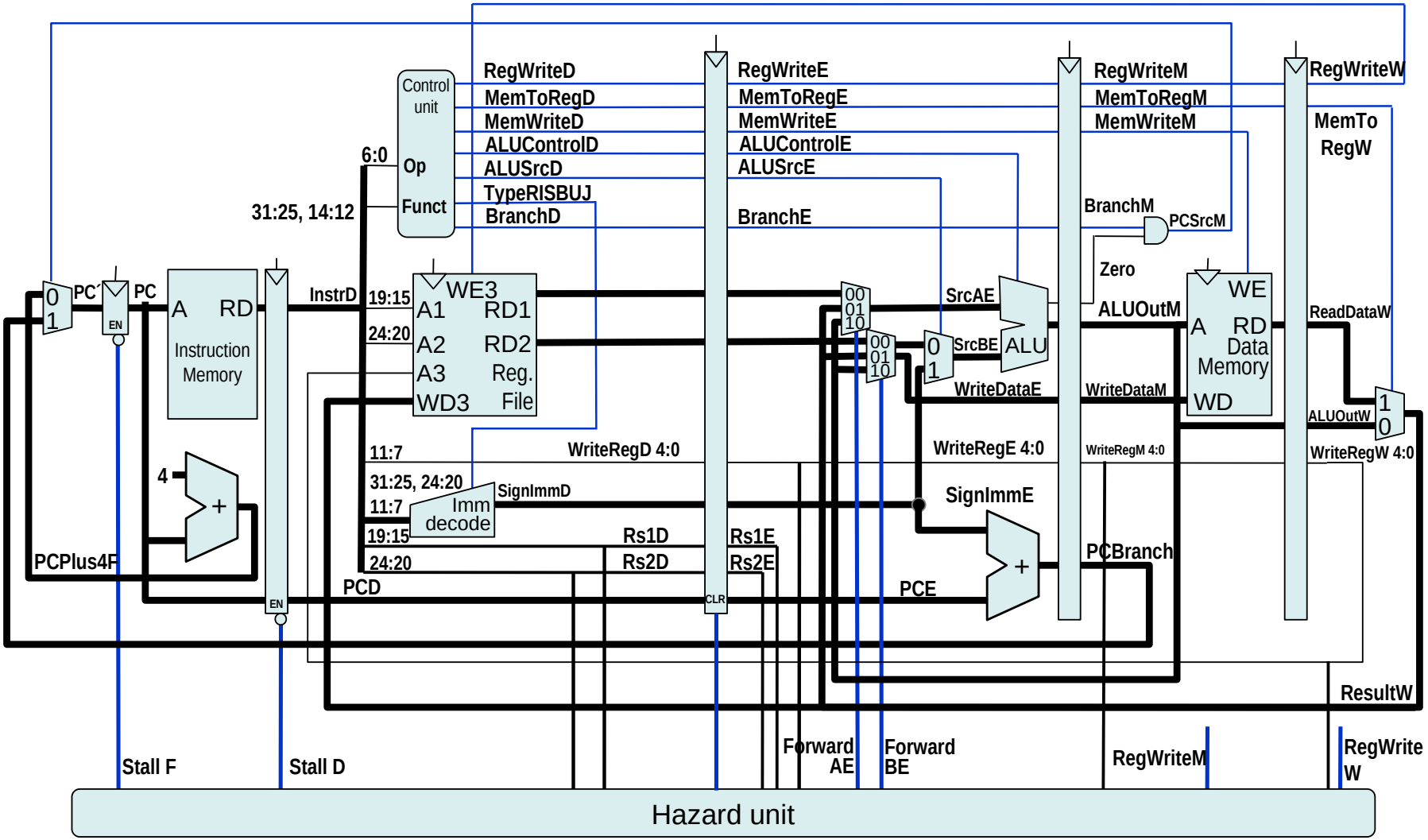
Resolution of Hazards on Comparator Inputs – MIPS



Branch Prediction and Speculative Execution (Next Lect.)



Result of Pipelined RISC-V Processor Design

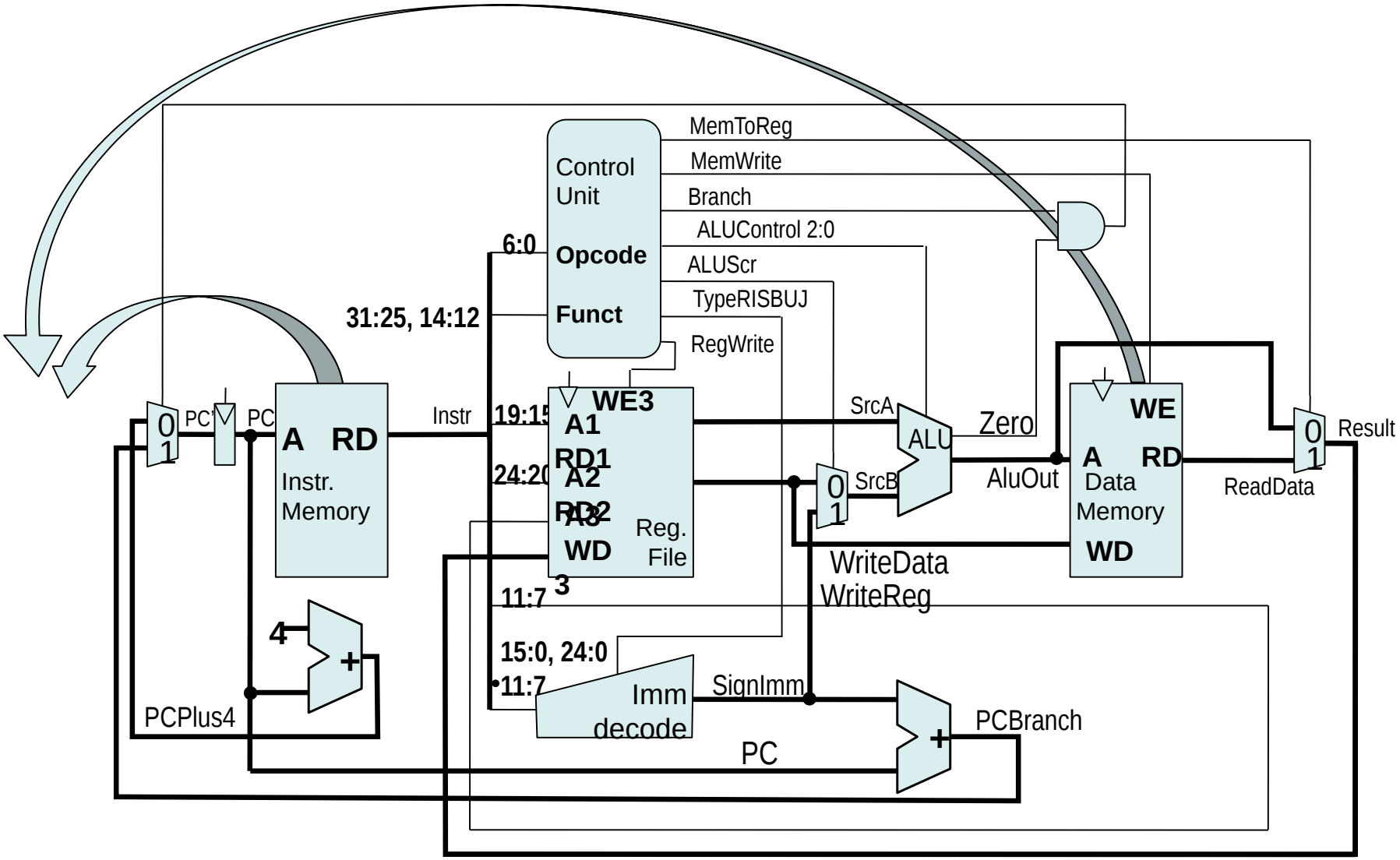


Performance of the Designed Pipelined Processor

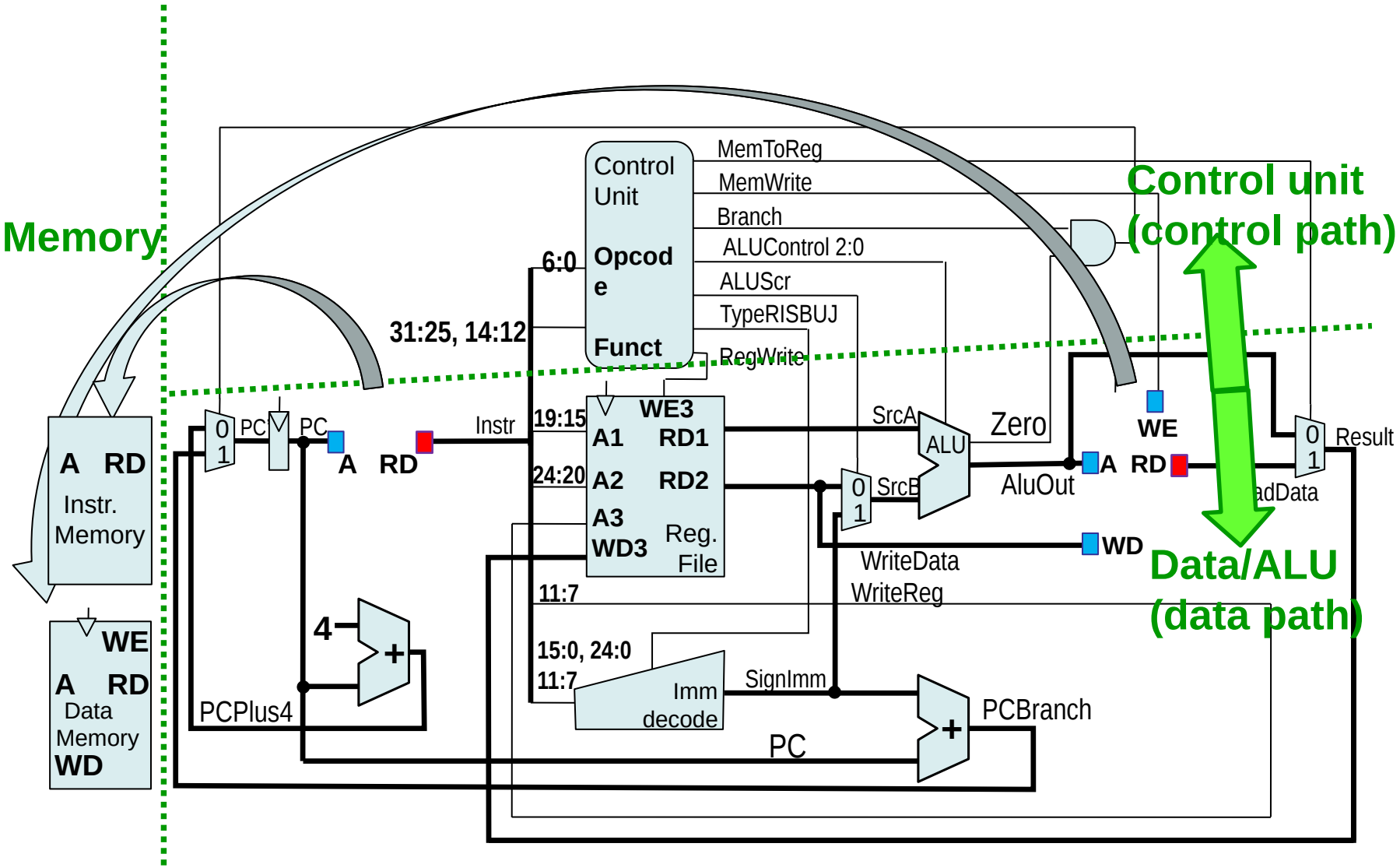
- What will be the maximum achievable frequency of the processor?
- Which stage is the slowest one?
- The minimum feasible cycle time is given by the slowest stage
- In our case, we are dealing with memory
  $T_C = 300\,\text{ns} \rightarrow f_{clk\,max} = 3\,333\,\text{kHz}$
  if we don't consider extra cycles to fill the pipeline (no pauses and emptying during branches, for example) then for an ideal $IPC = 1$
  $IPS = 1 \cdot 3\,333e3 = 3\,333\,000$ instructions per second
- The introduction of five-stage pipelined processing has led to an increase in throughput of $3\,333\,000/980\,000 = 3.4\times$ assuming $IPC = 1$

Outline

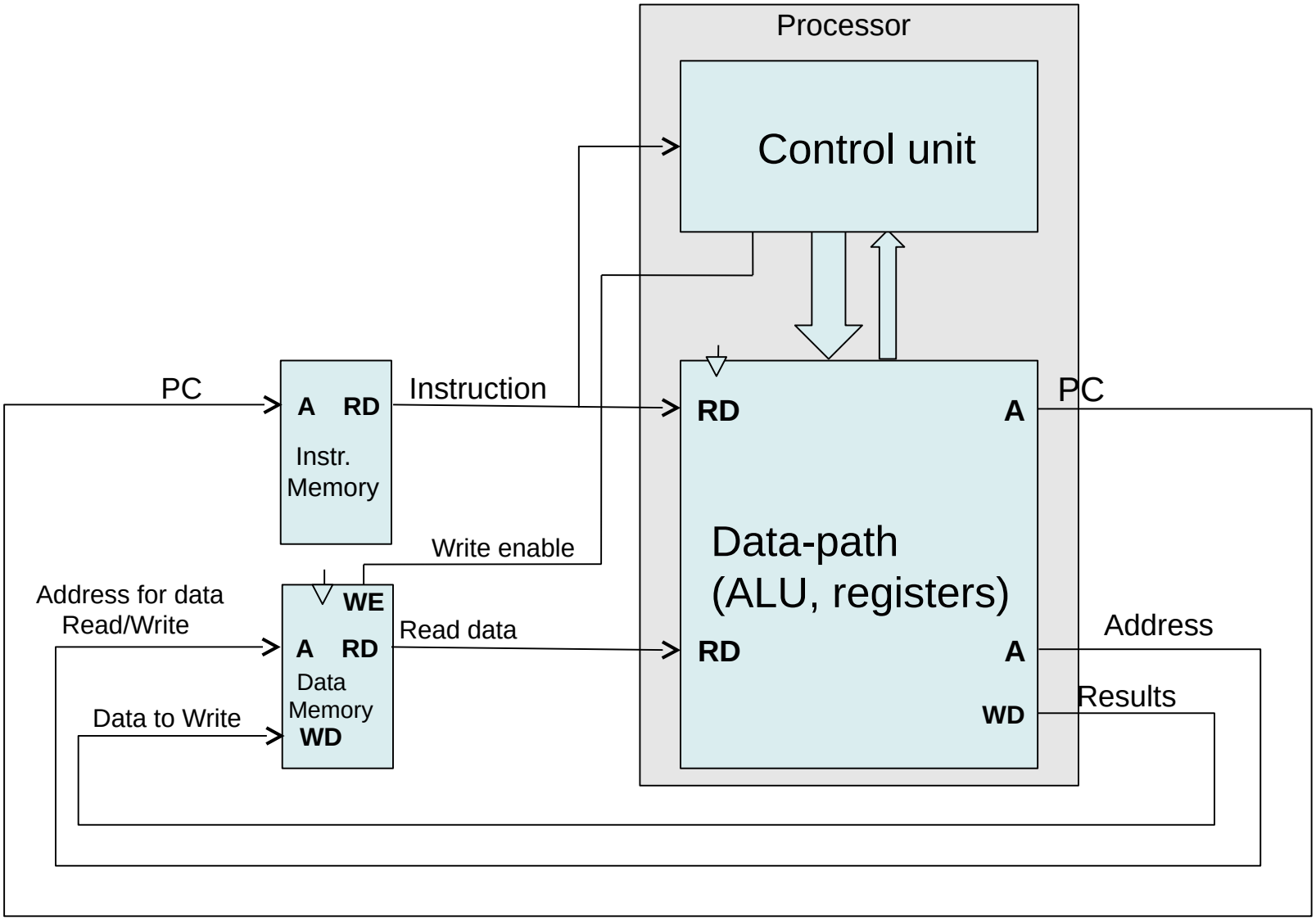
Design Correspondence to a Real Processor with External Memory (for simplicity we don't consider the pipeline)



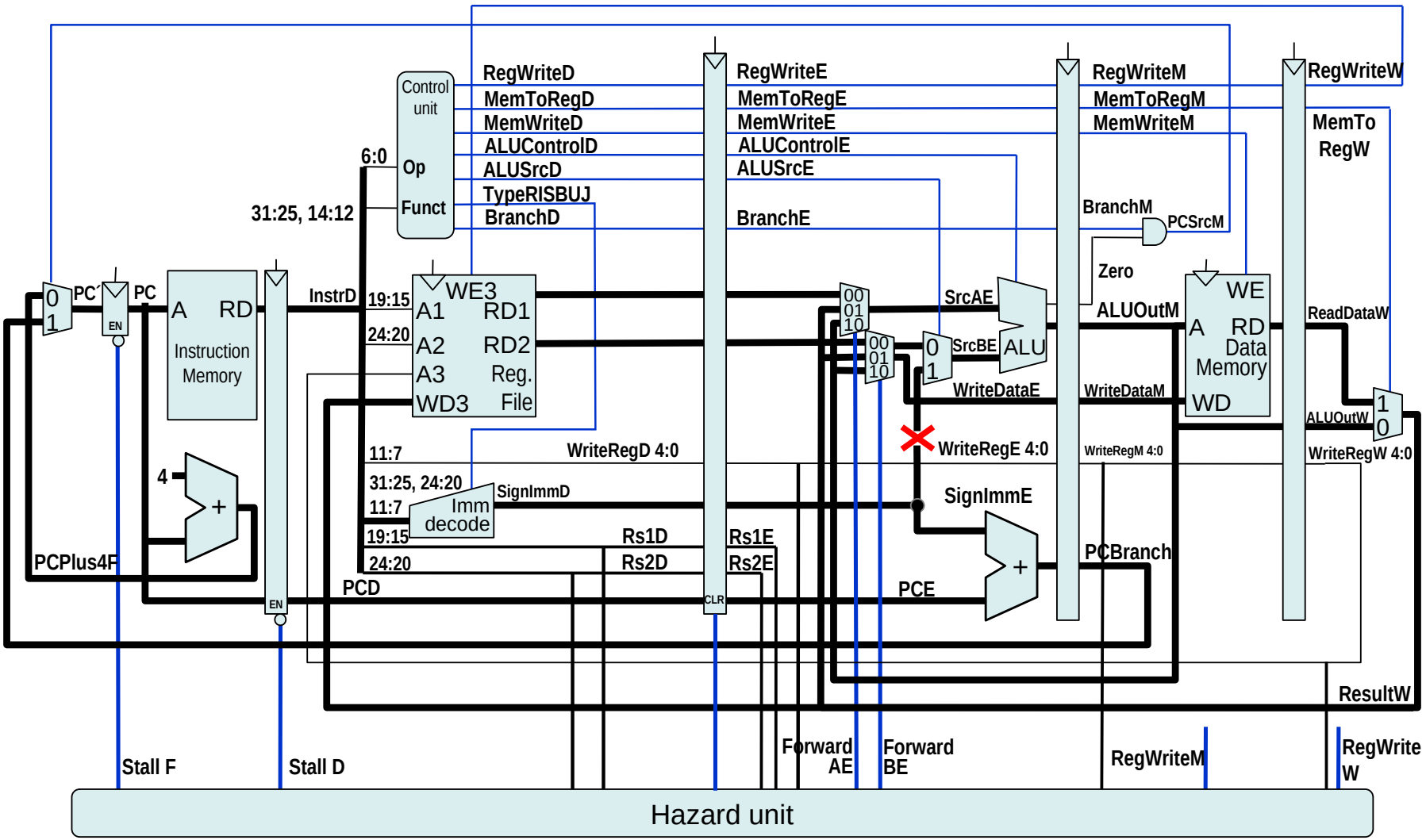
Memory Is no Longer Placed Inside the Processor



getting Back to the Original Separated CPU and Memory



Quiz – Which Instruction Misbehaves When Signal is Cut

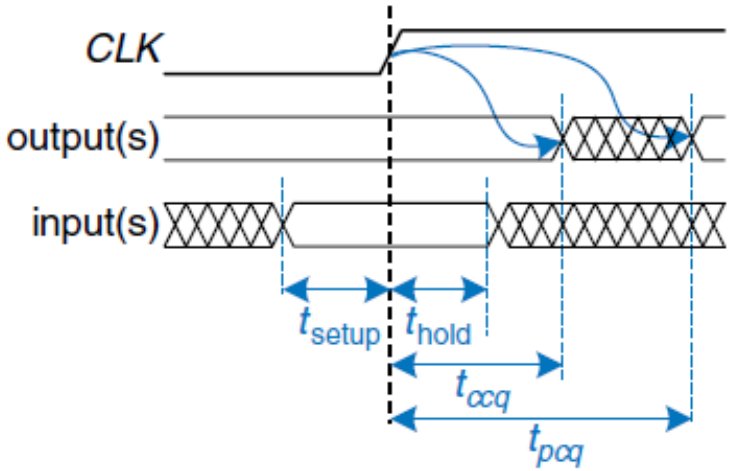


A) add **B)** addi **C)** beq **D)** or ?

Outline

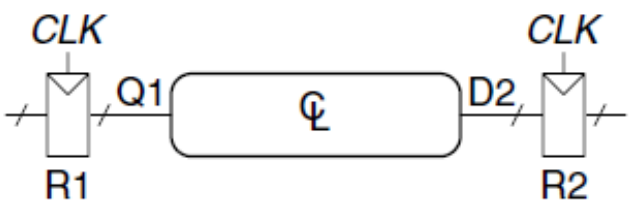
Requirements for Pipeline Timing

- The timing/AC characteristics of synchronous sequential circuit :

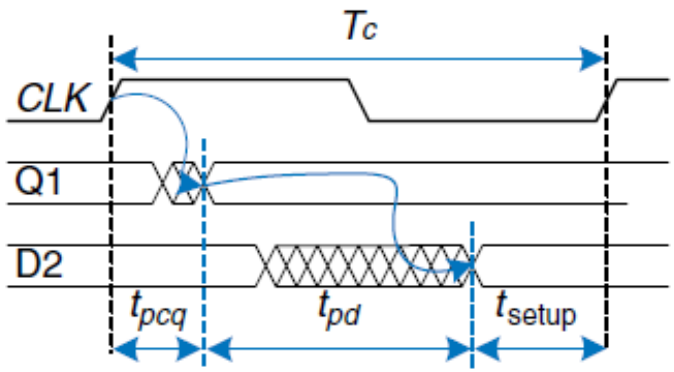


t_{setup} – inputs setup time
t_{hold} – inputs hold time

- Signal integrity constrain for the setup time before the clock:



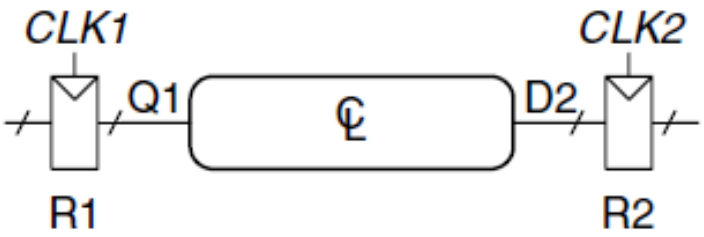
$$Tc >= t_{pcq} + t_{pd} + t_{setup}$$

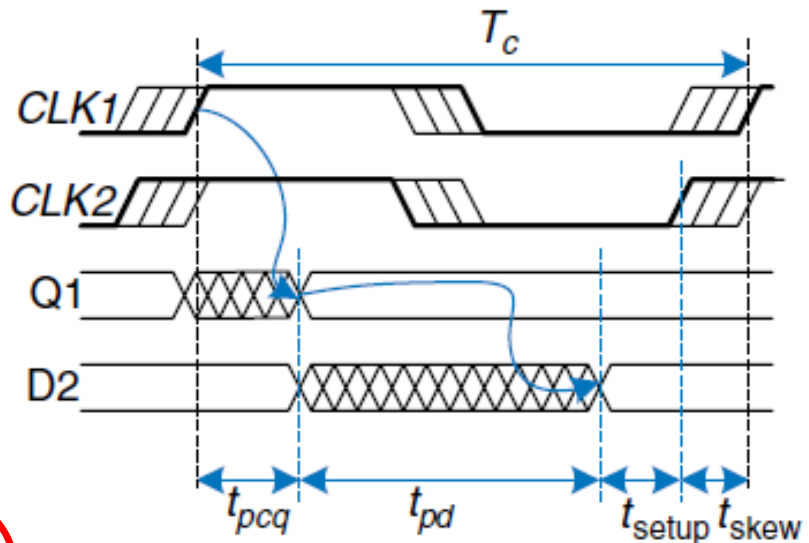


t_{pd} – combinatorial logic propagation delay

Requirements for Pipeline Timing

Constraint for the setup time (consider the clock distribution jitter):



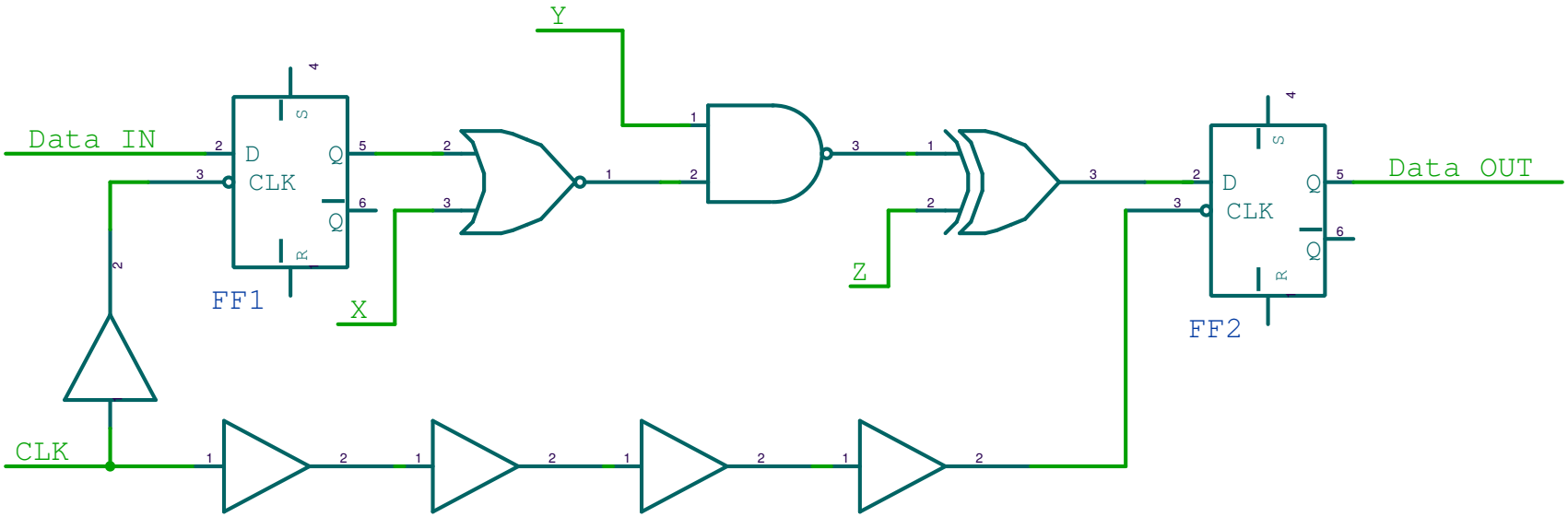


$$Tc >= t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$

Clock distribution jitter is limiting factor,
if it reaches or exceeds value of t_{pd}

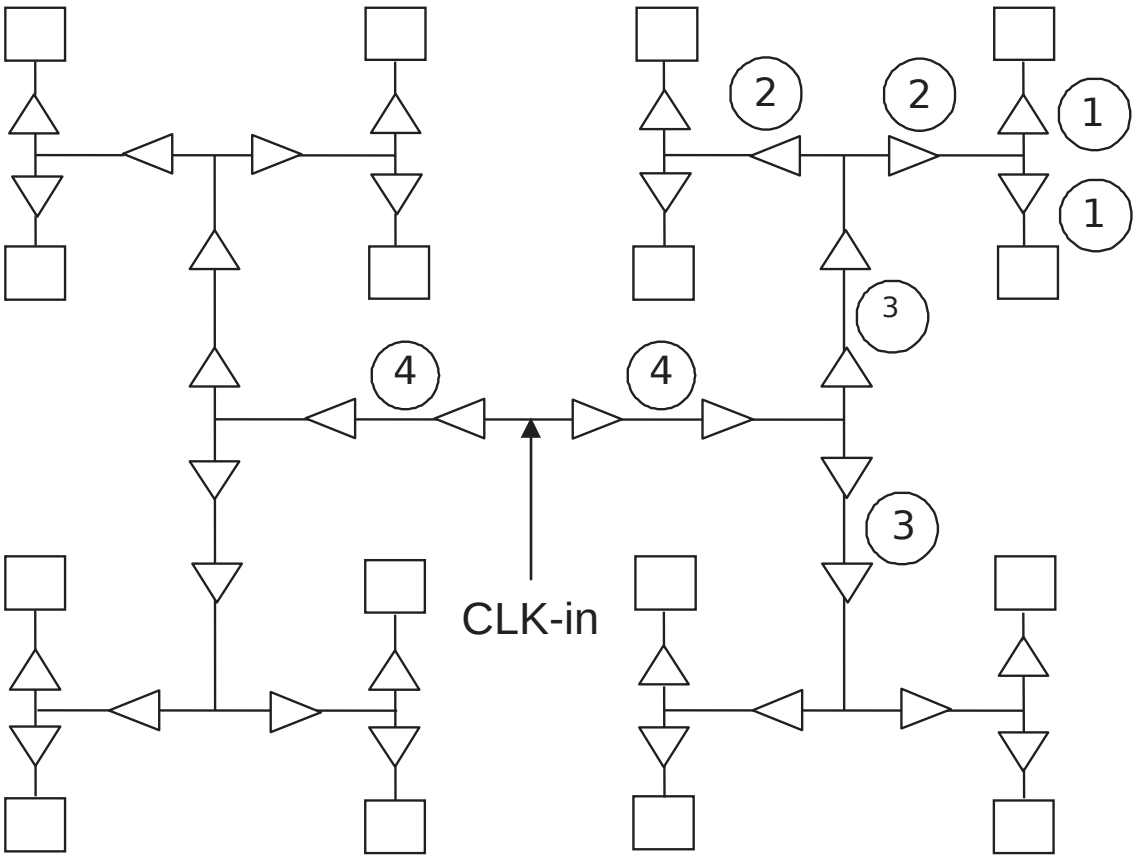
(too deep pipeline / too many stages…)

Clock Distribution Network Skew



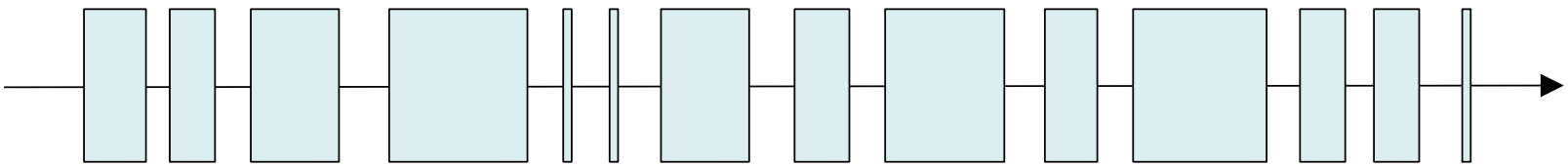
- **Positive Clock Skew** – clock arrives at the capturing sequential later than it arrives at the launching sequential
- **Negative Clock Skew** – clock arrives at the launching sequential later than it arrives at the capturing sequential
- **Local Clock Skew** – skew between any two sequentials with a valid timing path between them.
- **Global Clock Skew** – clock skew between any two sequentials in the design regardless of whether a timing paths exists between them

Clock Distribution Network – H-tree



source: Tawfik, S., Kursun, V.: Clock Distribution Networks with Gradual Signal Transition Time Relaxation for Reduced Power Consumption.

Pipeline Stages Balancing

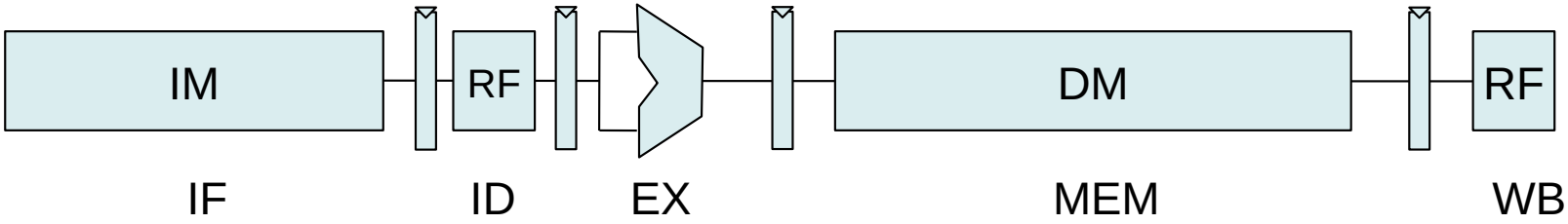


(applies to tree based adder, multiplier, (unrolled) iterative divider..)

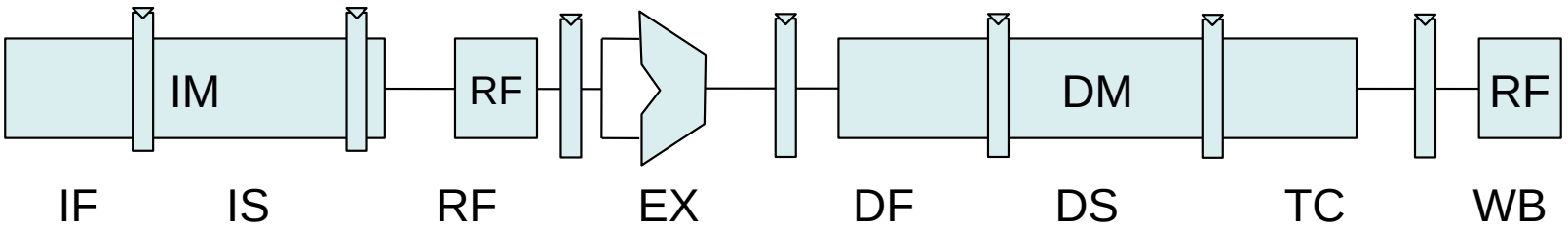
- **Balancing**: the goal is to divide the processing into N stages in such a way, that stage propagation delays are roughly the same...
- The number of stages reflects preference of performance (throughput) versus latency.

Superpipeline and Beyond

- Not well balanced 5-stage pipeline:



- Deeper pipeline is result of decomposing stages into more stages



- It allows CPU to work at higher frequencies but introduces many problems as well...
- Complex forwarding, more pipeline stalls, hazards need to be solved by complex logic

Superpipeline and Beyond

P5 (Pentium) : **5**
P6 (Pentium 3): **10**
P6 (Pentium Pro): **14**
NetBurst (Willamette, 180 nm) - Celeron, Pentium 4: **20**
NetBurst (Northwood, 130 nm) - Celeron, Pentium 4, Pentium 4 HT: **20**
NetBurst (Prescott, 90 nm) - Celeron D, Pentium 4, Pentium 4 HT, Pentium 4 ExEd: **31**
NetBurst (Cedar Mill, 65 nm): **31**
NetBurst (Presler 65 nm) - Pentium D: **31**
Core : **14**
Bonnell: **16**

Haswell **14**-**19**
Cooper Lake **14**-19

AMD Zen **19**
AMD Zen2 **19**

K7 Architecture - Athlon : **10**-**15**
K8 - Athlon 64, Sempron, Opteron, Turion 64: **12**-**17**

ARM 8-9: **5**
ARM 11: **8**
Cortex A7 2-wide **8**-**10**
Cortex A8 2-wide **13**
Cortex A15 3-wide **15**-**25**

Cortex-A35 2-wide **8**
Cortex-A53 2-wide **8**
Cortex-A57 3-wide **15**
Cortex-A77 4-wide **11**–**13**
Denver 2/7-wide **13**
M4 6-wide **15**
Lightning 7-wide **16**

SiFive FE310-G000 **5**
SiFive FU540-C000 **5**

The Optimum Pipeline Depth for a Microprocessor:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.4333&rep=rep1&type=pdf