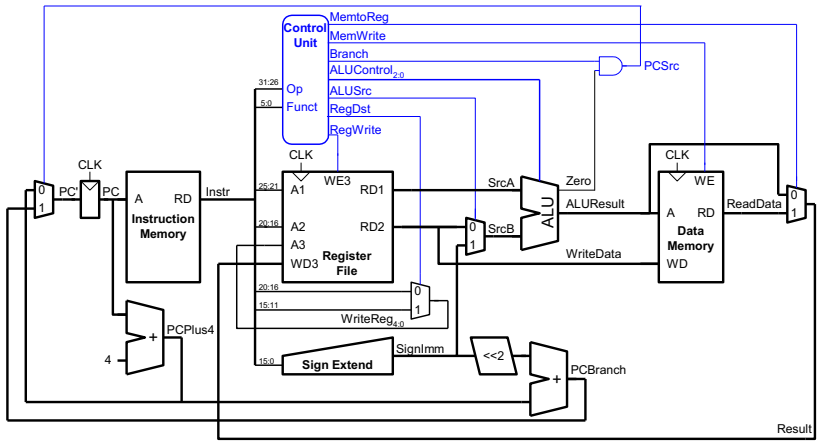
Performance: Pipelining

Jan Reineke
Universität des Saarlandes

Roadmap: Computer architecture



1. Combinatorial circuits: Boolean Algebra/Functions/Expressions/Synthesis
2. Number representations
3. Arithmetic Circuits: Addition, Multiplication, Division, ALU

4. Sequential circuits: Flip-Flops, Registers, SRAM, Moore and Mealy automata
5. Verilog

6. Instruction Set Architecture
7. Microarchitecture

8. **Performance:** RISC vs. CISC, **Pipelining**, Memory Hierarchy

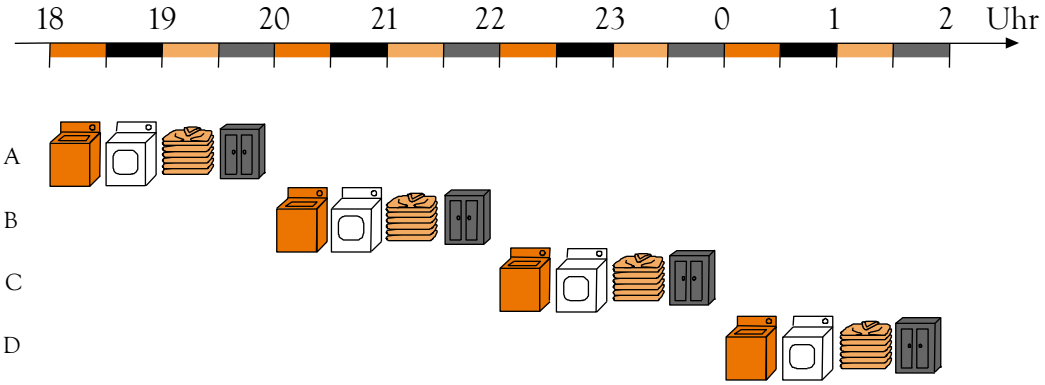
Pipelining: An example from daily life

- Persons A, B, C, D are returning from vacation; there is a lot of dirty laundry to wash!
- At our disposal:
  - A washing machine (1/2 hour runtime)
  - A dryer (1/2 hour runtime)
  - An ironing machine (1/2 hour of work to iron)
  - A linen closet (1/2 hour of work to put away)
- Every person washes their own laundry by themselves!

There are two options to perform the work!

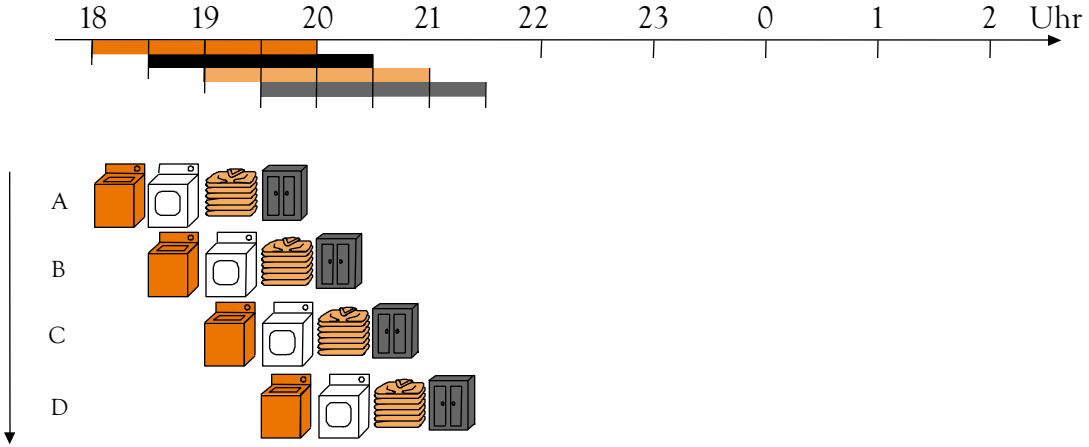
Pipelining: An example from daily life

Analogously to the single-cycle implementation:



Duration of work:
8 hours

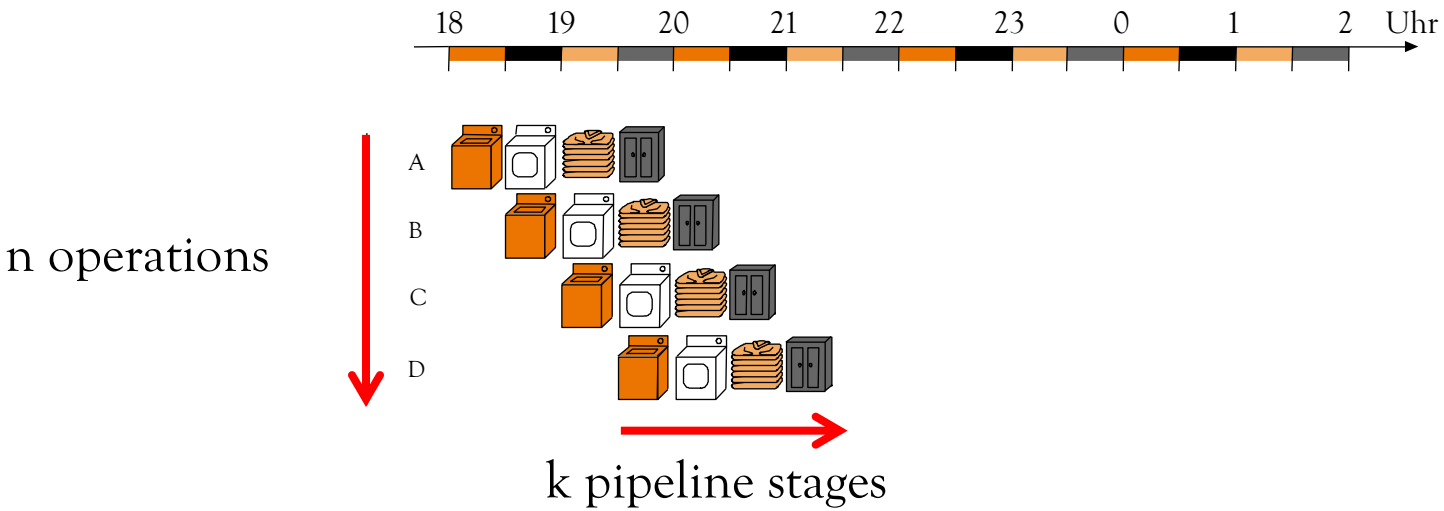
With pipelining:



Duration of work:
3 1/2 hours

Observations

- Latency of each individual operation is **unchanged**
- Number of completed operations per hour (throughput) **increased**



Speedup = Time without optimization/Time with optimization
= n*k / (k+n-1)

Efficiency = Used resources/Available resources
= k*n / k*(k+n-1) = n/(k+n-1)

Idealizing assumptions in the example

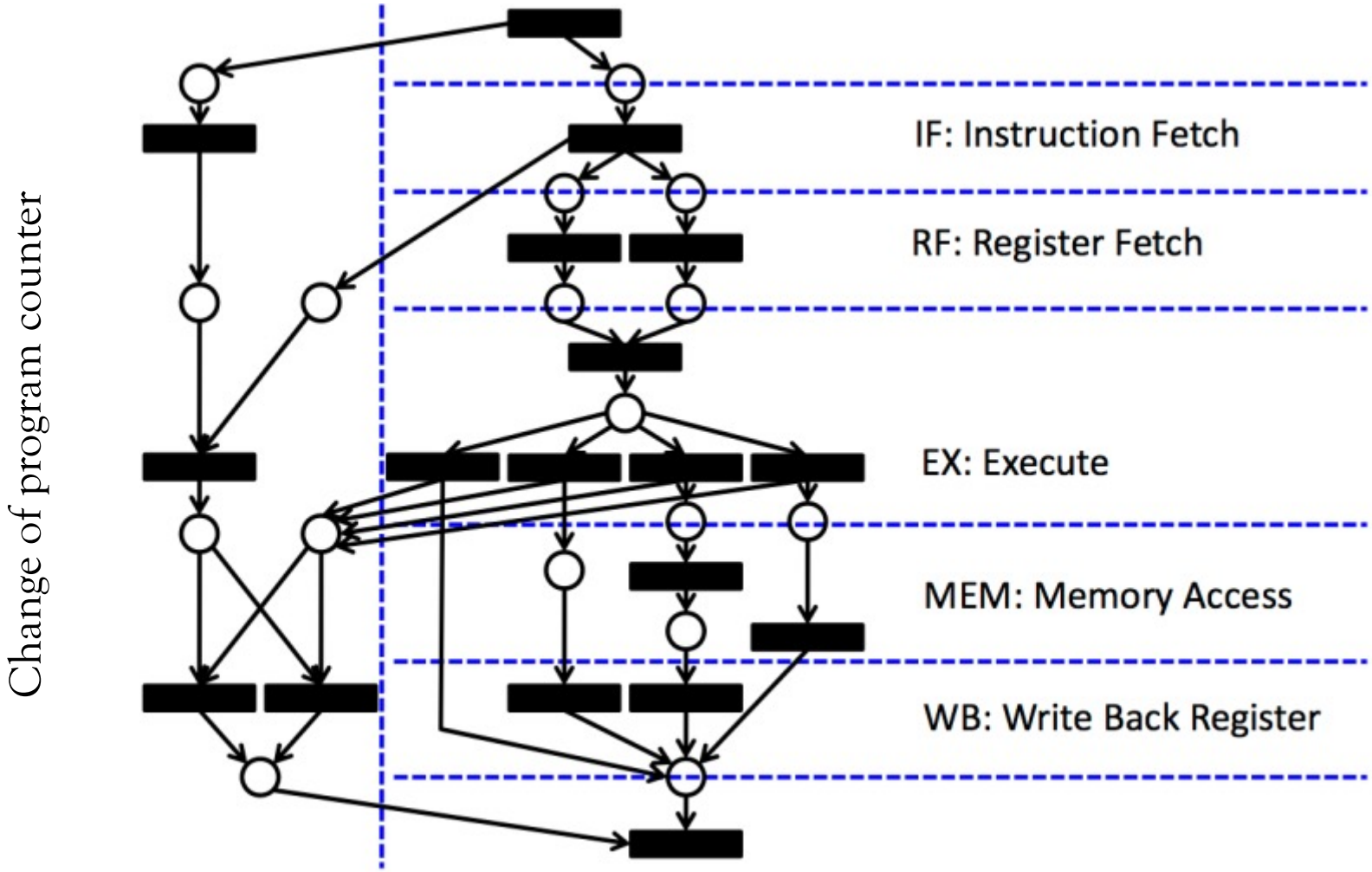
- Homogeneous partial operations
  - The operation can be split into partial operations of equal length.
- Repetition of identical operations
  - The same operation is repeatedly performed on different “inputs”.
- Repetition of independent operations
  - All operations are pairwise independent

Transfer to the MIPS processor: Division of instruction execution into phases

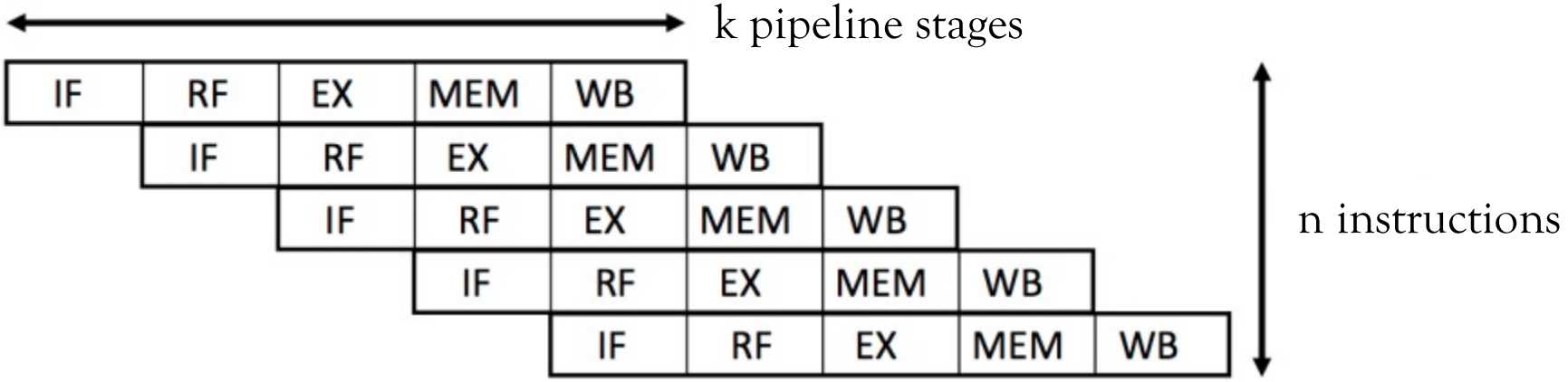
- Divide execution of machine instruction into several phases of ideally equal length.
- A sensible division depends on the instruction set and the employed technology.

- In the MIPS datapath the following division is sensible:
  - **IF (Instruction Fetch)**: Reading the instruction.
  - **ID (Instruction Decode)** or **RF (Register Fetch)**: Decoding the instruction and reading the register contents.
  - **EX (Execute)**: Execute the instruction or compute an address.
  - **MEM (Memory)**: Access to data memory.
  - **WB (Write Back)**: Storing the result in the register file.

Illustration of division into phases



Speedup and efficiency: Homogeneous partial operations

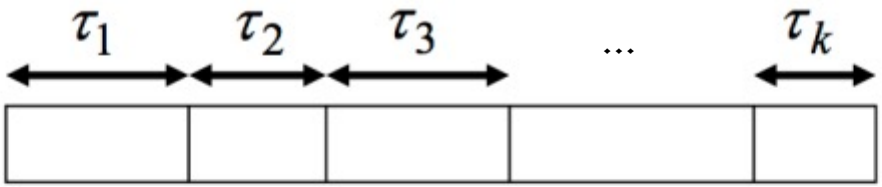


$$Speedup = \frac{n \cdot k}{k + n - 1} \xrightarrow{n \to \infty} k$$

$$\text{Efficiency} = \frac{n \cdot k}{(k + n - 1) \cdot k} = \frac{Speedup}{k} \xrightarrow{n \to \infty} 1$$

Speedup and efficiency: Inhomogeneous partial operations

Assumption: Phases may have different lengths:



$$Speedup = \frac{n \cdot \sum_{1 \le i \le k} \tau_i}{(k+n-1) \cdot \max_{1 \le i \le k} \tau_i} \xrightarrow{n \to \infty} \frac{\sum_{1 \le i \le k} \tau_i}{\max_{1 \le i \le k} \tau_i}$$

$$\text{Efficiency} = \frac{n \cdot \sum_{1 \le i \le k} \tau_i}{(k+n-1) \cdot k \cdot \max_{1 \le i \le k} \tau_i} = \frac{Speedup}{k} \xrightarrow{n \to \infty} \frac{\sum_{1 \le i \le k} \tau_i}{k \cdot \max_{1 \le i \le k} \tau_i}$$

Example: MIPS datapath

Assumption about the delays in the datapath:

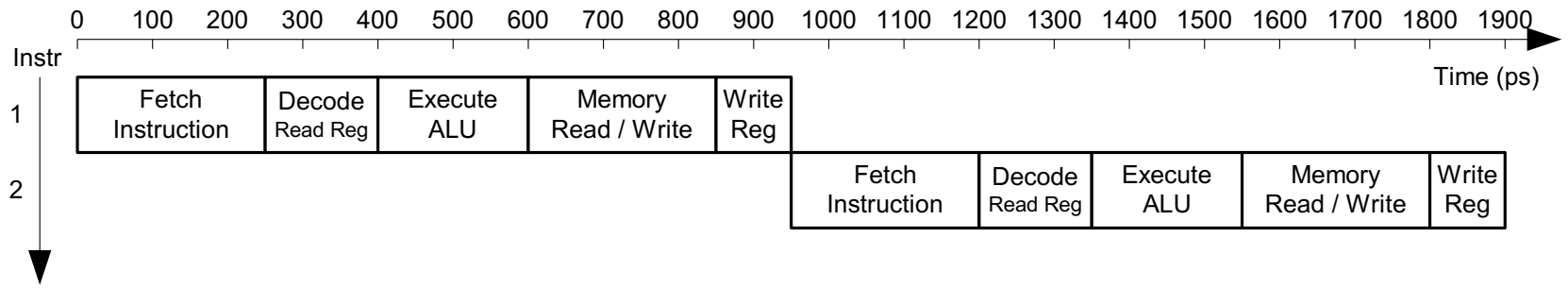
- 100 ps = 100 * 10^{-12} seconds for reading from and writing to registers
- 200 ps for all other components (ALU, main memory)

Instruction	IF	ID/RF	EXE	MEM	WB	Total time
`lw`	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
`sw`	200 ps	100 ps	200 ps	200 ps		700 ps
`R-typ`	200 ps	100 ps	200 ps		100 ps	600 ps
`beq`	200 ps	100 ps	200 ps			500 ps

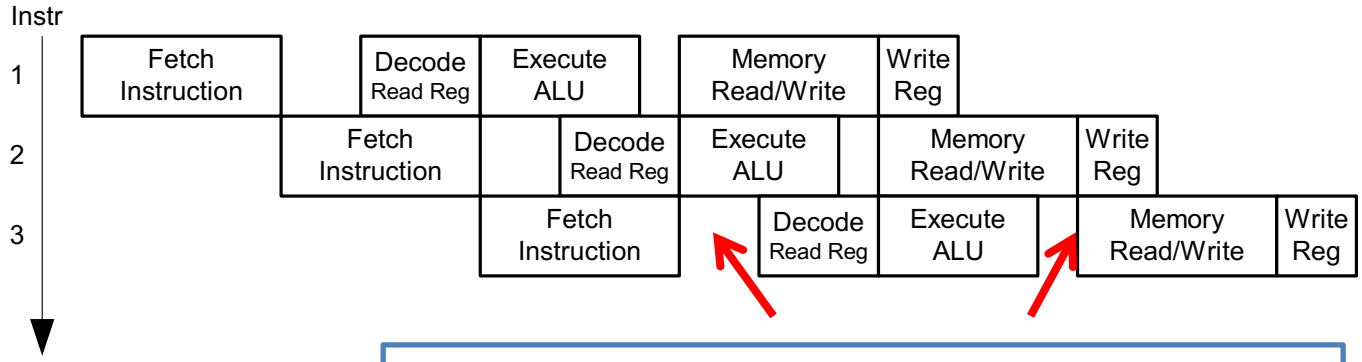
- *Efficiency* = 800 ps / (5 * 200 ps) = 80%
- Single cycle: 1 / 800 ps = 1,25 * 10^9 Instructions/s
  Pipelining: 1 / 200 ps = 5 * 10^9 Instructions/s
  → *Speedup* (for `lw`) = 4

Single-cycle vs Pipelined implementation

Single-Cycle

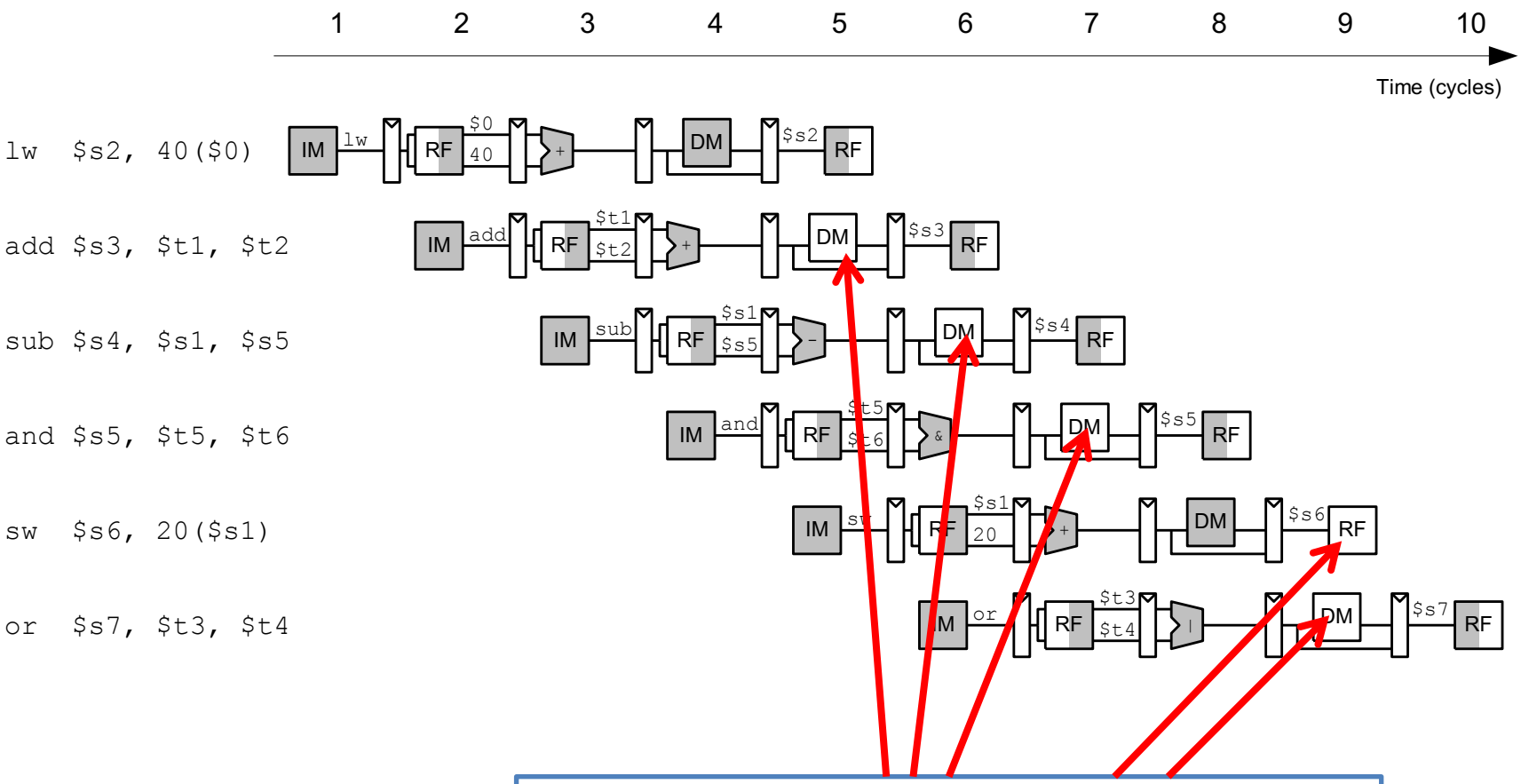


Pipelined



Internal fragmentation
(due to inhomogeneous latencies)

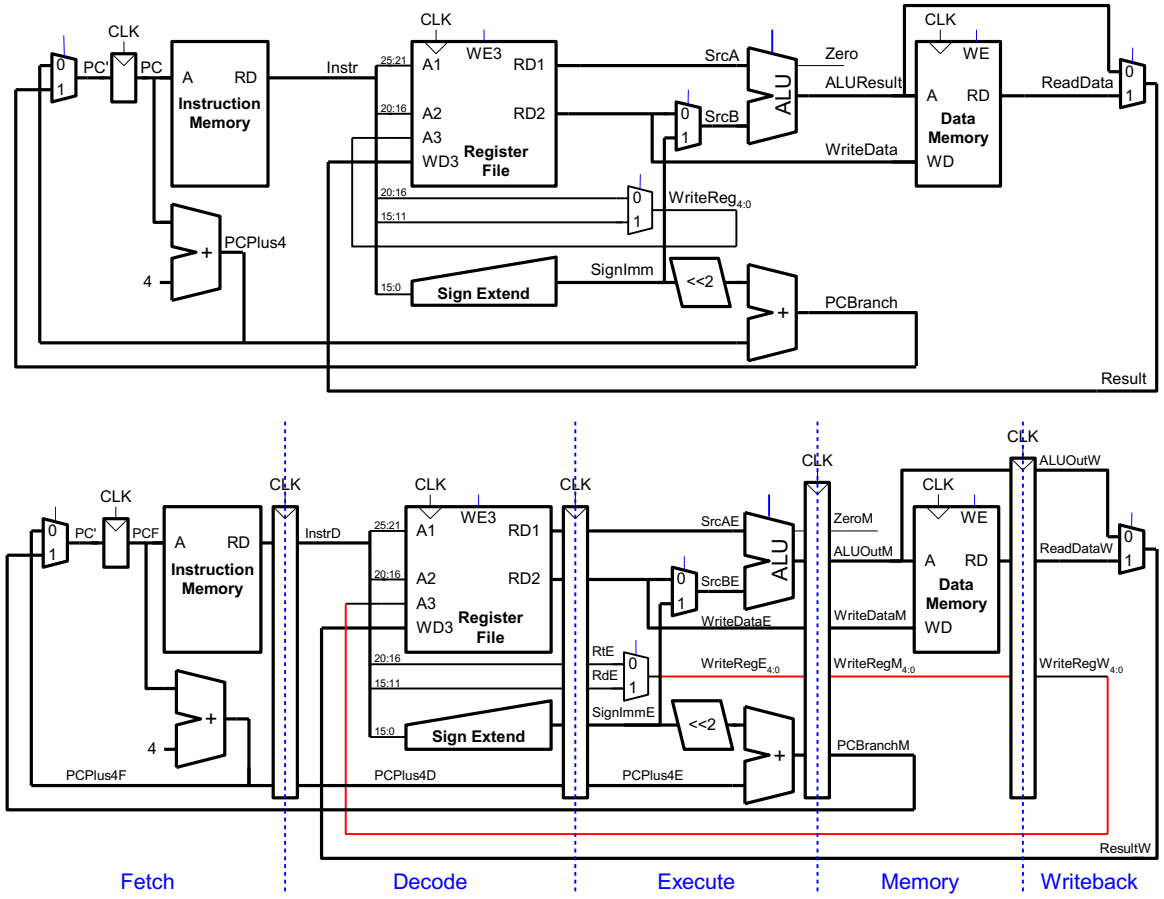
Pipelining


1
2
3
4
5
6
7
8
9
10
Time (cycles)
lw $s2, 40($0)
add $s3, $t1, $t2
sub $s4, $s1, $s5
and $s5, $t5, $t6
sw $s6, 20($s1)
or $s7, $t3, $t4
IM
RF
DM
$0
40
$t1
$t2
$s1
$s5
$t5
$t6
$t3
$t4
20
$s2
$s3
$s4
$s5
$s6
$s7
External fragmentation
(due to "superfluous" pipeline stages)

MIPS processor with pipelining

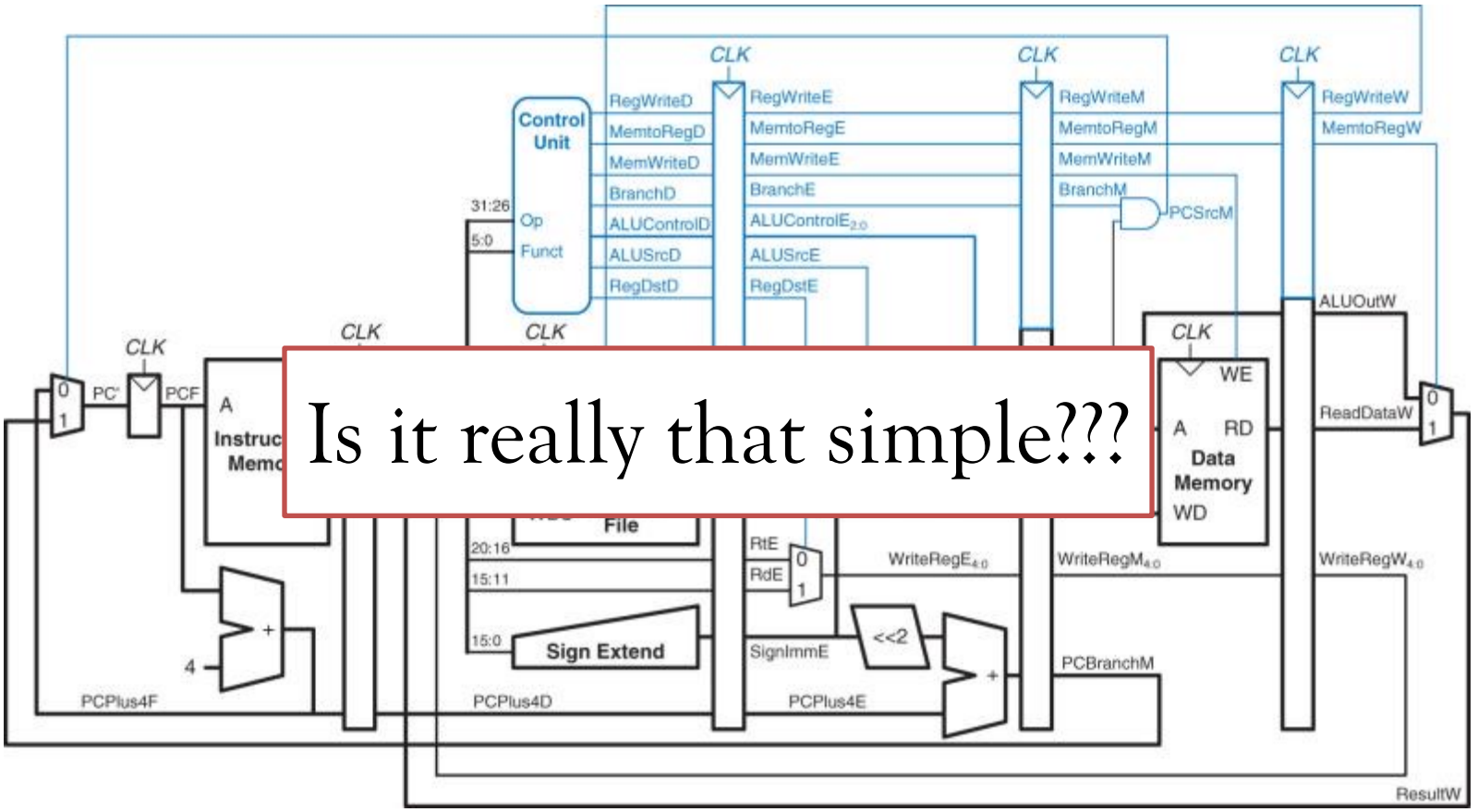
- Overlap the 5 phases (IF, DE, EX, MEM, WB) of the instruction execution
- Requires additional registers to remember multiple active instructions
- These registers conceptually lie between the pipeline stages

Single-cycle vs pipeline datapath



Pipelining: Control unit

- Decoder of the single-cycle implementation can be reused
- Control is delayed to the right pipeline stage



Remember: Idealizing assumptions in the pipelining example

- Homogeneous partial operations
  - The operation can be split into partial operations of equal length. → otherwise internal fragmentation
- Repetition of identical operations
  - The same operation is repeatedly performed on different "inputs". → otherwise external fragmentation

- Repetition of independent operations

  All operations are pairwise independent

  → Does **not always** hold in MIPS. When?

Dependencies

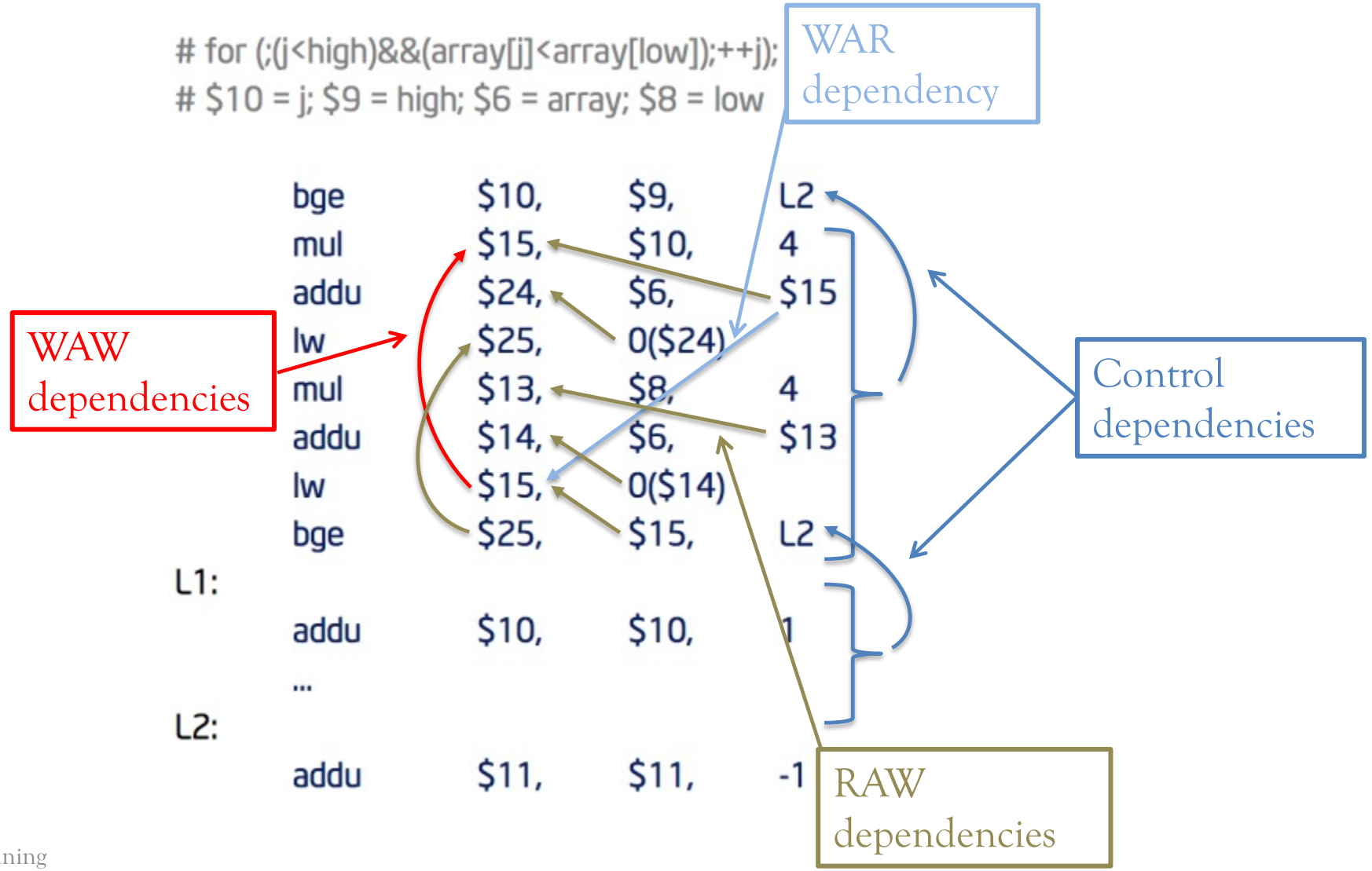
- When an instruction depends on the result of another instruction
- Two types of dependencies:
  - *Control dependence:*
    The result of an instruction is required to determine which instructions are executed next
  - *Data dependence, 3 types:*
    - Read-after-Write (RAW, "true dependency"):
      Instruction reads from register that is written to by prior instruction
    - Write-after-Read (WAR, "anti-dependency"):
      Instruction overwrites register that prior instruction reads
    - Write-after-Write (WAW, "output dependency"):
      Instruction overwrites register written to by prior instruction

Example: Dependencies

```
# for (;(j<high)&&(array[j]<array[low]);++j);
# $10 = j; $9 = high; $6 = array; $8 = low

        bge     $10,    $9,     L2
        mul     $15,    $10,    4
        addu    $24,    $6,     $15
        lw      $25,    0($24)
        mul     $13,    $8,     4
        addu    $14,    $6,     $13
        lw      $15,    0($14)
        bge     $25,    $15,    L2
L1:
        addu    $10,    $10,    1
        ...
L2:
        addu    $11,    $11,    -1
```

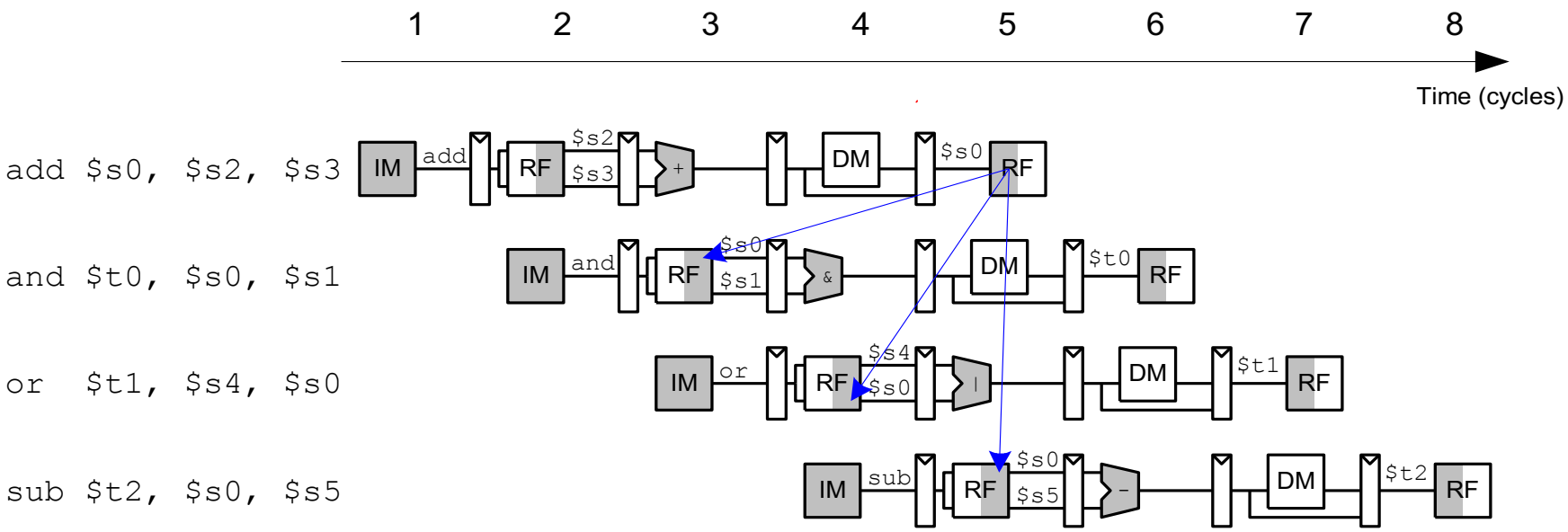
Example: Dependencies (Resolution)



Pipeline hazards

- A **hazard** is a situation in which an operation **cannot be executed immediately**, as this would violate a dependency
- Two types of hazards in in-order processor:
  - *Control hazard:*
    The result of a branch instruction is not yet available
  - *Data hazard:*
    - Read-after-Write (RAW):
      Instruction reads from register that is written to by other instruction and that has not yet been performed

Example: Data hazards

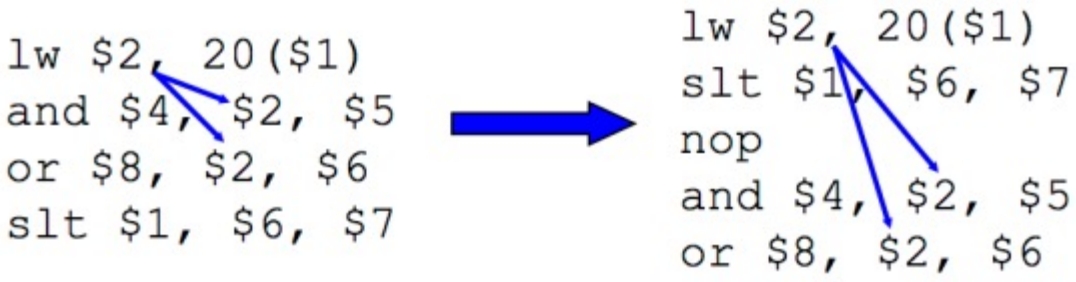


Avoiding data hazards: Compiler

Avoiding data hazards via the compiler:

Compiler may try to reorder instructions to avoid hazards.

- *Example*:



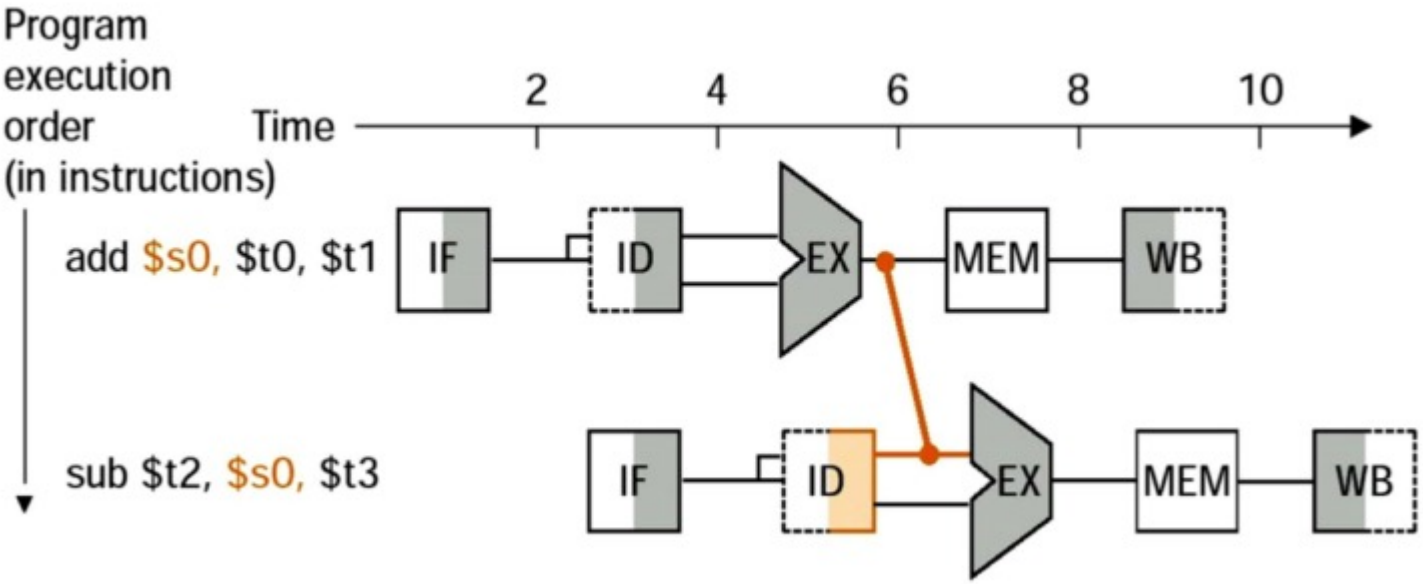
- If the compiler is unable to insert a sensible instruction, it may insert a `nop` ("no operation", "no op").
  It has no effect, but traverses the pipeline like all other instructions.

Avoiding data hazards: Forwarding

Avoiding data hazards via forwarding:

- The missing data is "forwarded" to the consuming instruction before storing the result in the register file.

Principle:

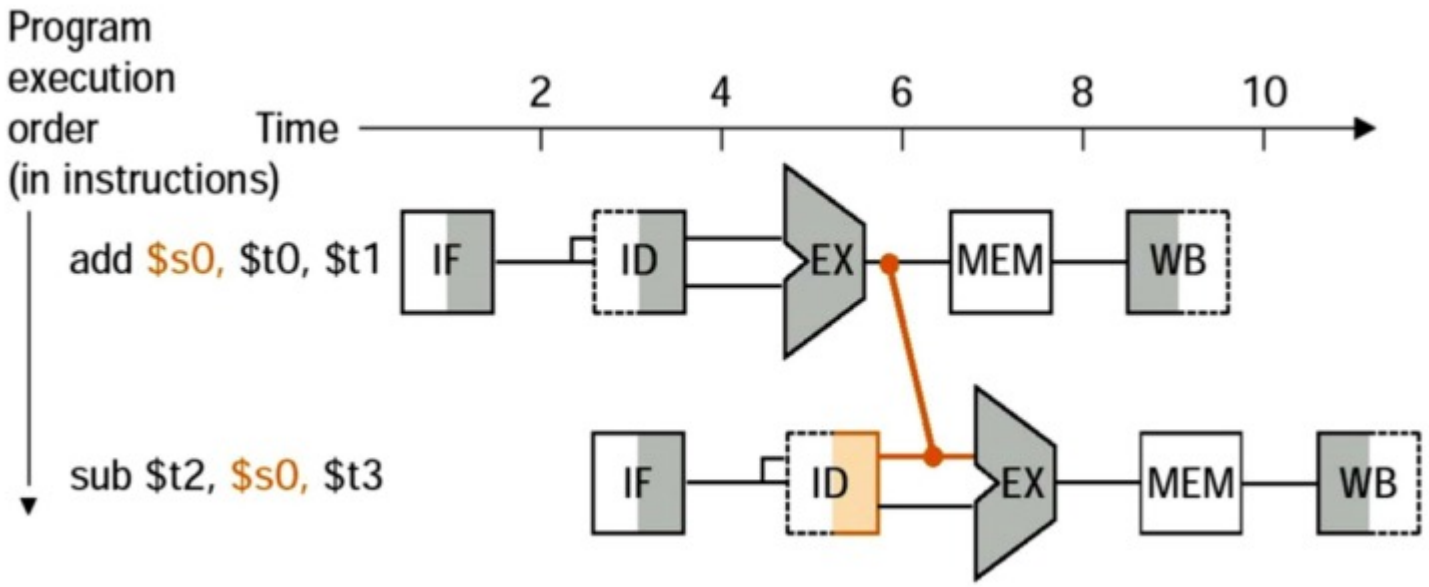


Avoiding data hazards: Forwarding

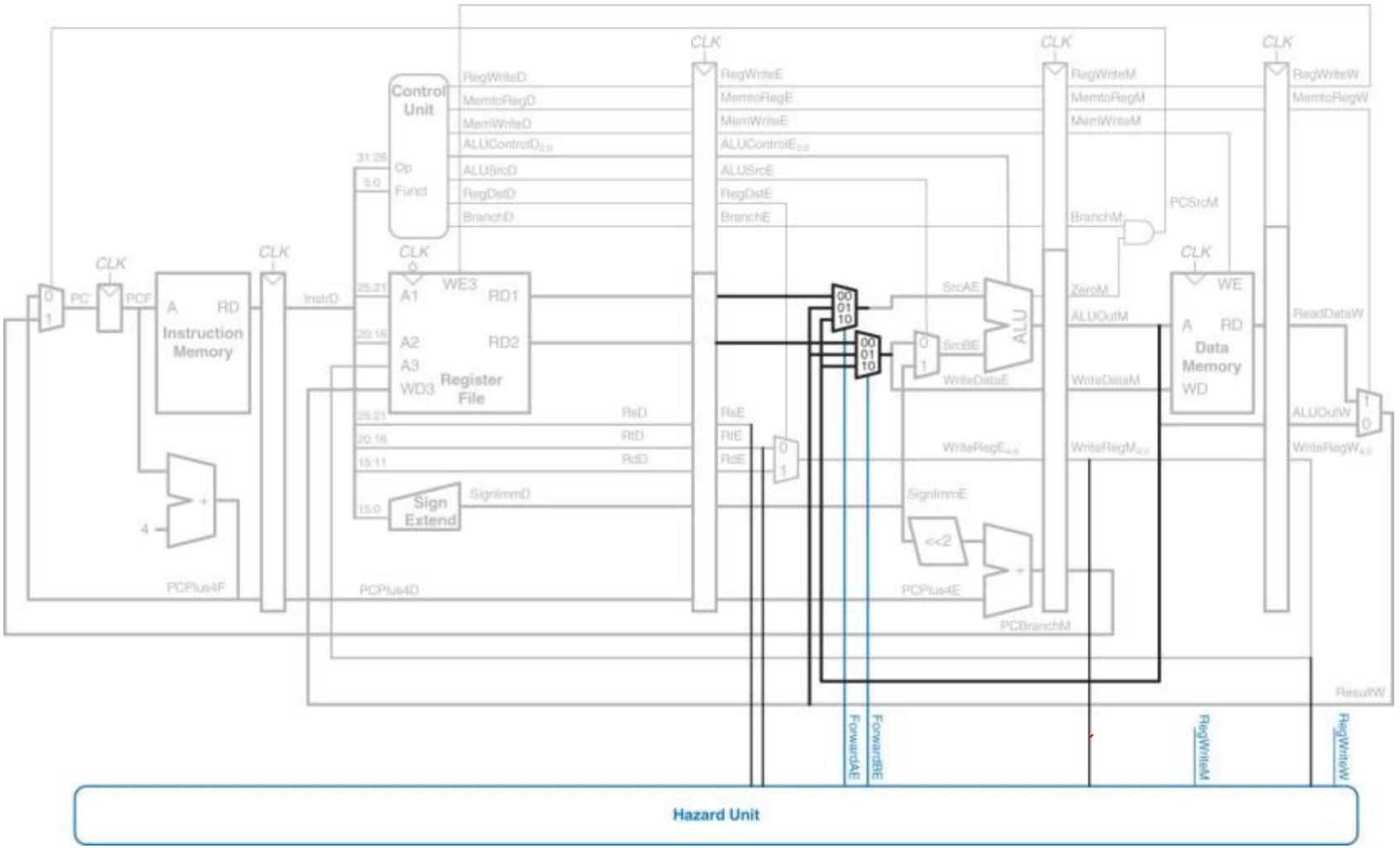
Why not write back to the register file right after the execute phase?

→ WAW dependency on loads (lw)
→ Register file cannot handle multiple writes simultaneously

Principle:



Forwarding: Datapath and control

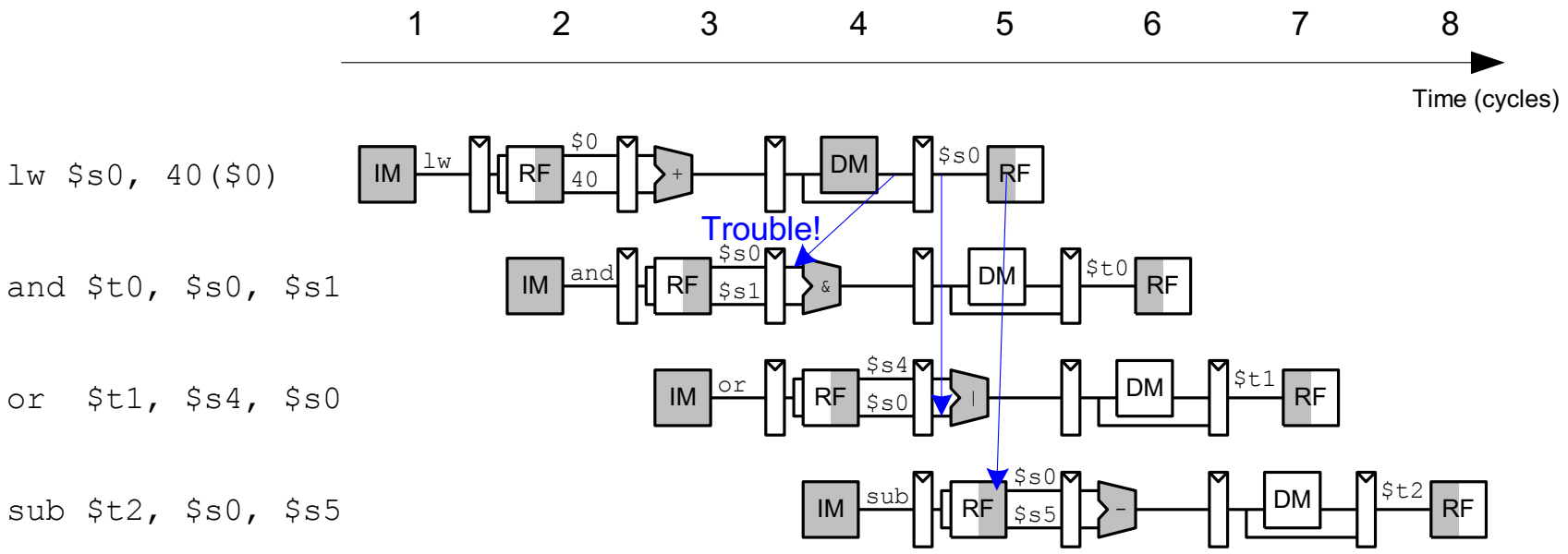


Avoiding data hazards: Stalls

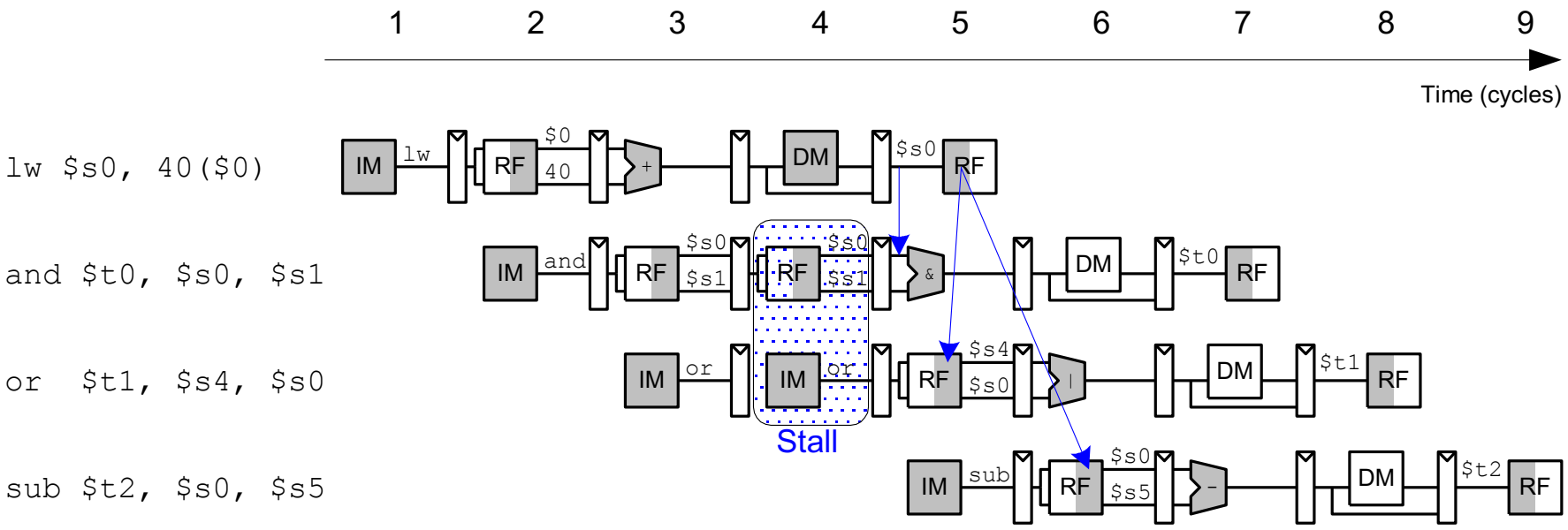
Avoiding data hazards via stalls

- "Forwarding" **cannot avoid all** hazards.
- Repair by inserting **"bubble"**. This corresponds to the insertion of a `nop` instruction.
- This "bubble" is not inserted in the IF phase, but only in a later phase.
- Instructions in *later* stages traverse the pipeline as usual. Instructions in *earlier* stages are stalled.

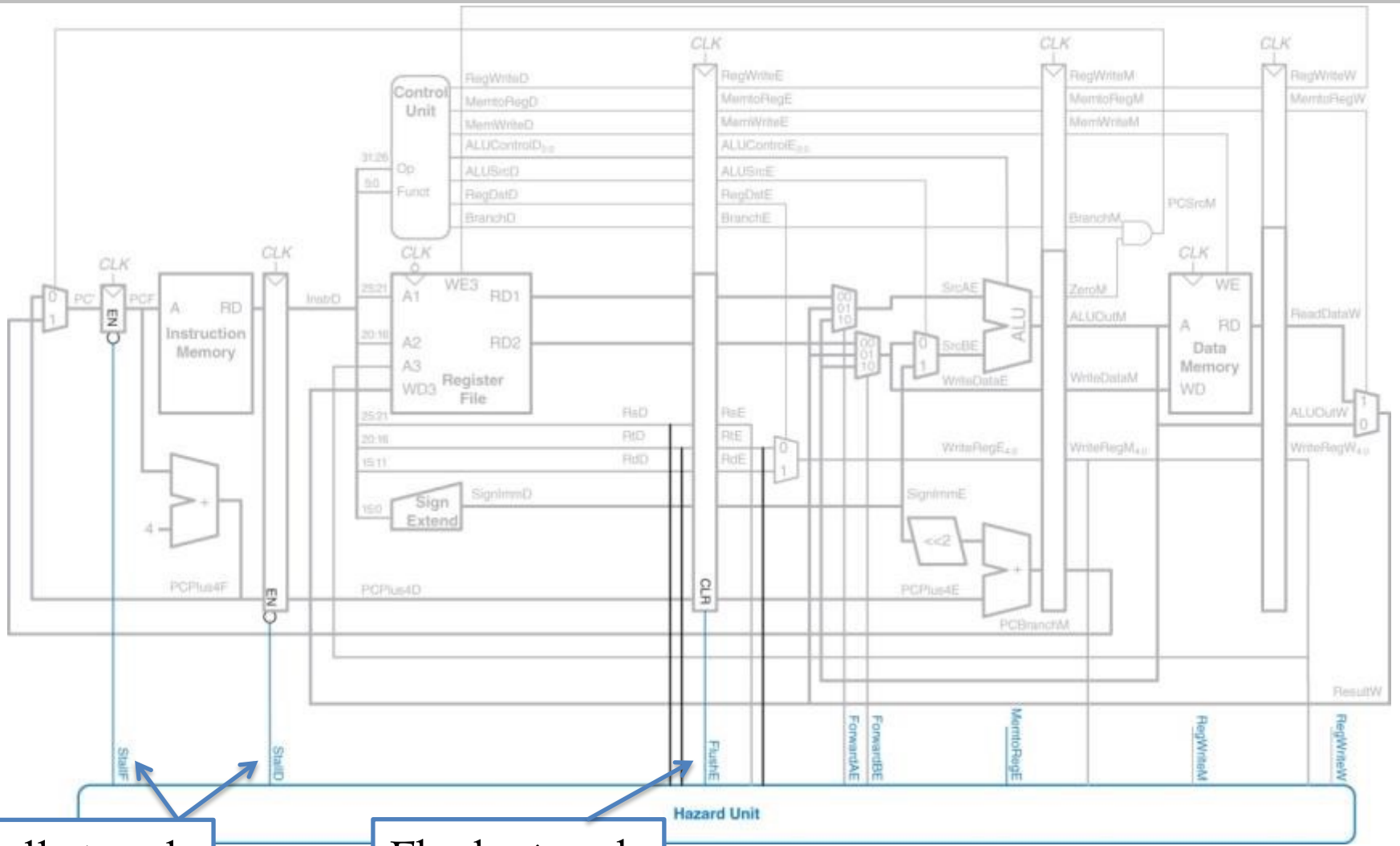
Avoiding data hazards: Stalls



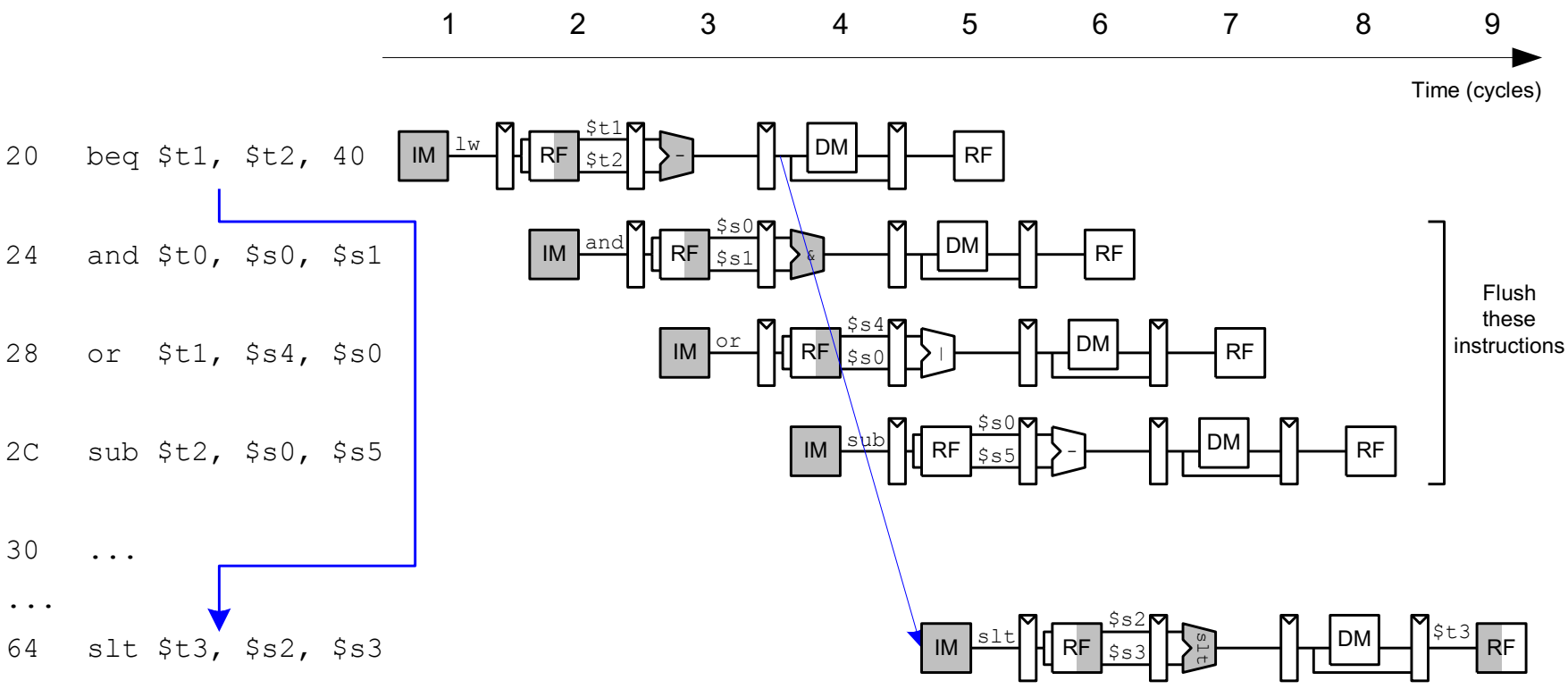
Avoiding data hazards: Stalls



Stalling and forwarding unit



Example: Control hazards



Avoiding control hazards: Stalls

- Insertion of "bubbles" as with data hazards
  - *Example*: Branching decision known after the MEM phase. Every branch leads to 3 stall cycles:

Branch	IF	ID	EX	MEM	WB		
Instr. i+1/k		IF	0	0	IF	ID	EX
Instr. i+2/k+1						IF	ID
Instr. i+3/k+2							IF

Computation of the average cycles per instruction (CPI):

- Assumptions: five-stages pipeline, 1 cycle per stage, **only stalls due to branches, 30% of all instructions are branches**.
- CPI = 0,7 + 0,3*4 = 1,9
  → slowed down almost by a factor of 2!

Avoiding control hazards: Static prediction

- Processor predicts that branches are not taken.
  - If this prediction **comes true**, execution continues.
  - **Otherwise** the instructions are aborted and the execution continues with the correct instructions.
- Example of an incorrect prediction, branching decision is known after the MEM phase:

Branch	IF	ID	EX	MEM	WB		
Instr. i+1/k		IF	ID	EX	IF	ID	EX
Instr. i+2/k+1			IF	ID	O	IF	ID
Instr. k+2				IF	O	O	IF

Avoiding control hazards: Static prediction

Computation of the average cycles per instruction (CPI):

- Assumptions: five-stages pipeline, 1 cycle per stage, only stalls due to branches, 30% of all instructions are branches. **One half of all branches is executed**.
- CPI = 0,7 + 0,3*0,5*4 + 0,3*0,5 = 1,45

Avoiding control hazards: Dynamic prediction

The processor makes the prediction **based on the history of prior branching decisions**.

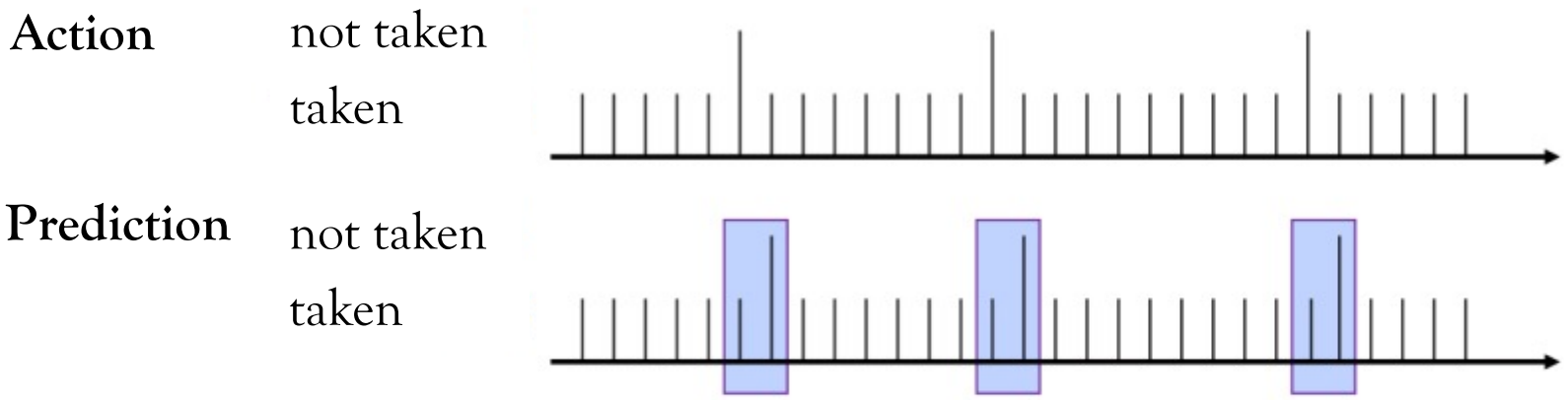
Simplest approach:

- If this branch was taken last time, predict it is also taken this time. Otherwise predict the branch is not taken.
- This scheme can be enhanced to take into account "older" branching decisions.
  E.g. some AMD processors employ neural networks for branch prediction.

Avoiding control hazards: Dynamic prediction

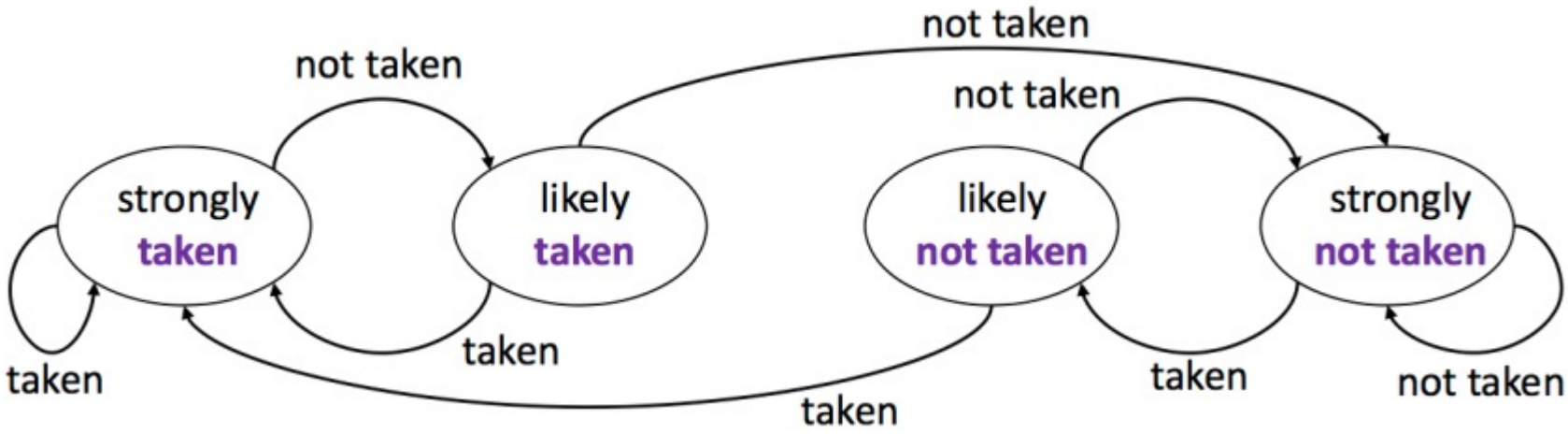
Problem with simple dynamic prediction:

In case of loops, branching probabilities are very non-uniform (e.g. 99% taken, 1% not taken). At an incorrect prediction, the next prediction is also wrong:



Avoiding control hazards: Dynamic prediction

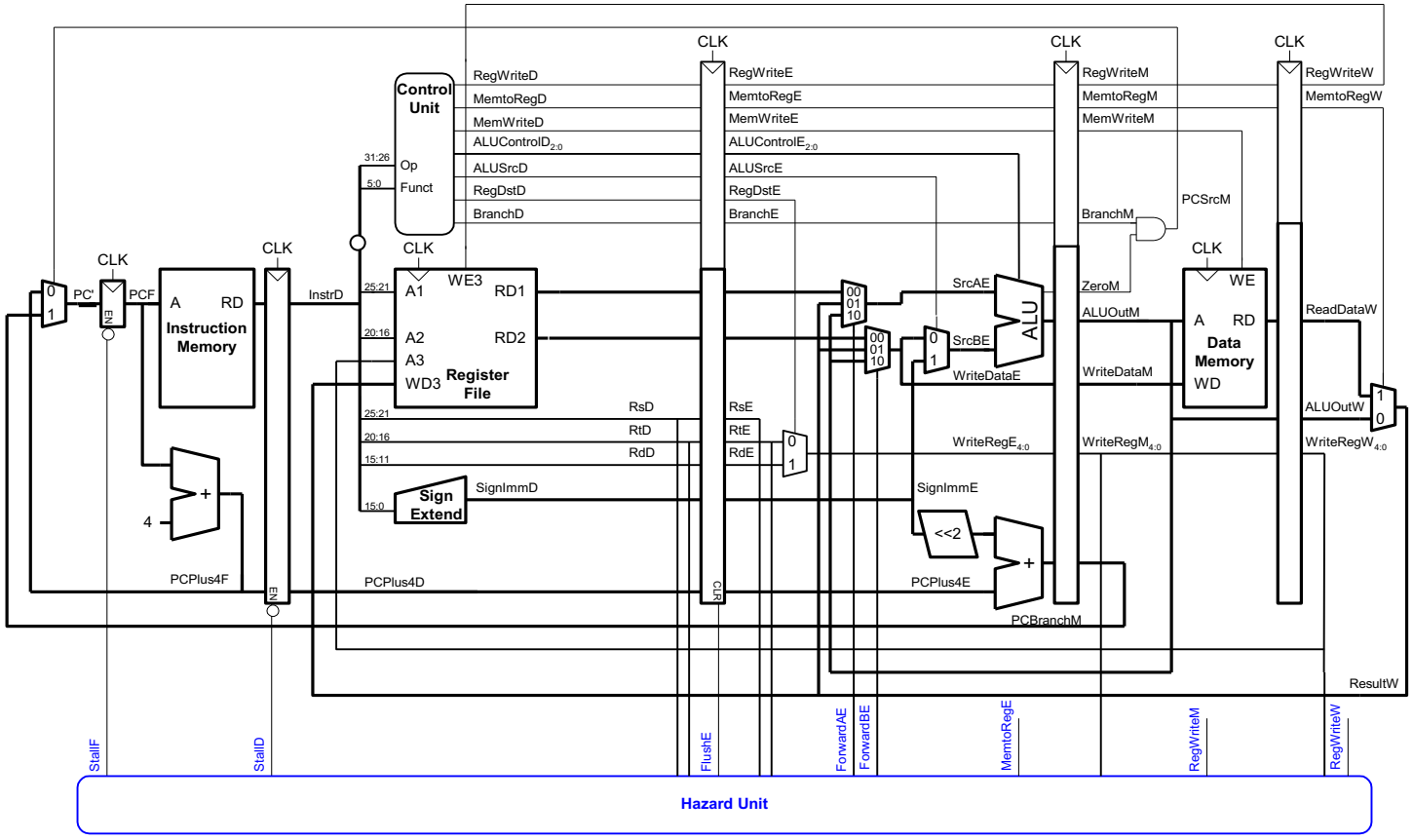
- *Solution via 2-Bit prediction:*
  The prediction is only changed upon two successive incorrect predictions.
- Modeled via *Moore automaton*:



Avoiding control hazards: Early branch resolution

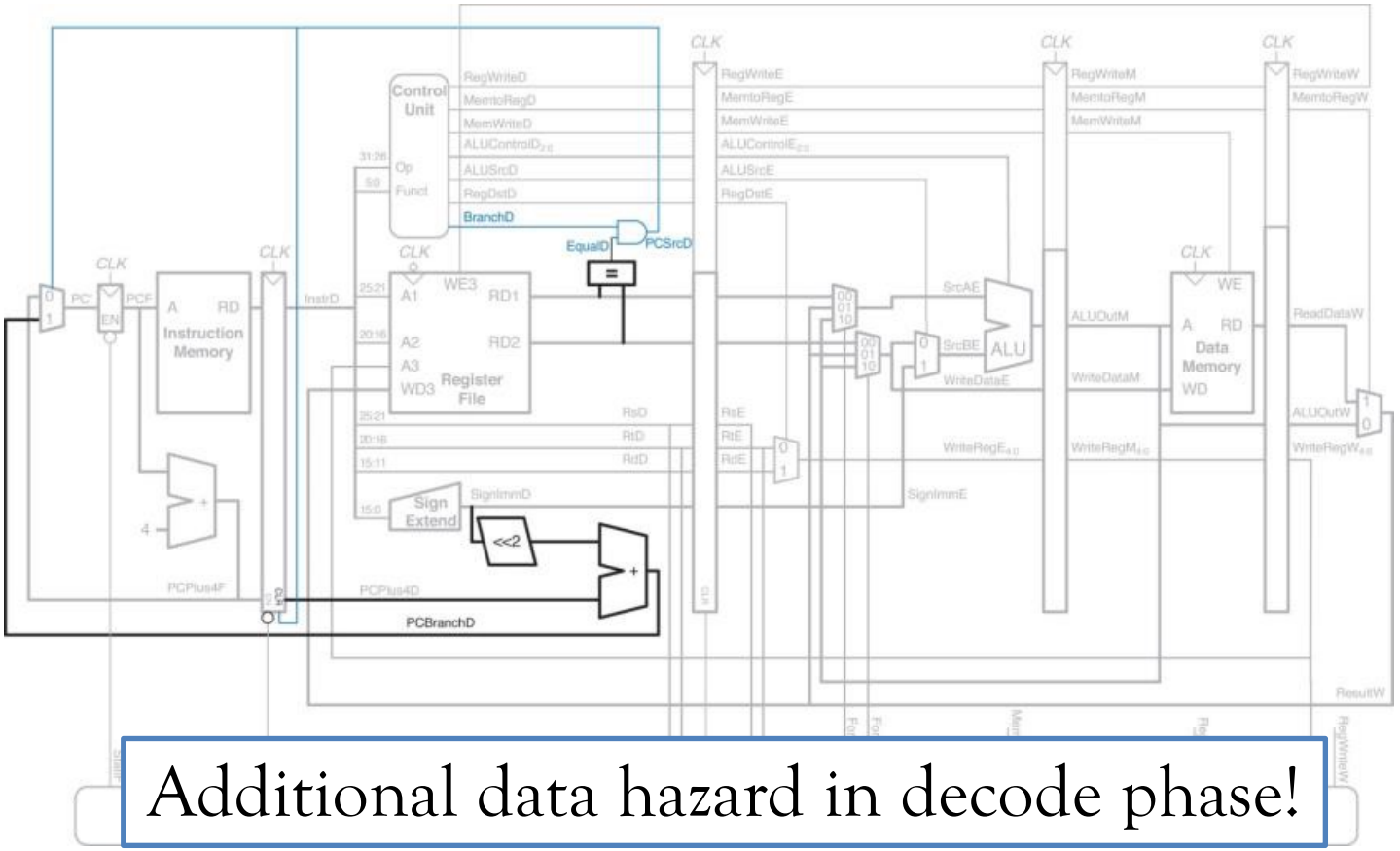
Reducing the stall cycles by resolving the branching decision earlier.

Avoiding control hazards: Original processor



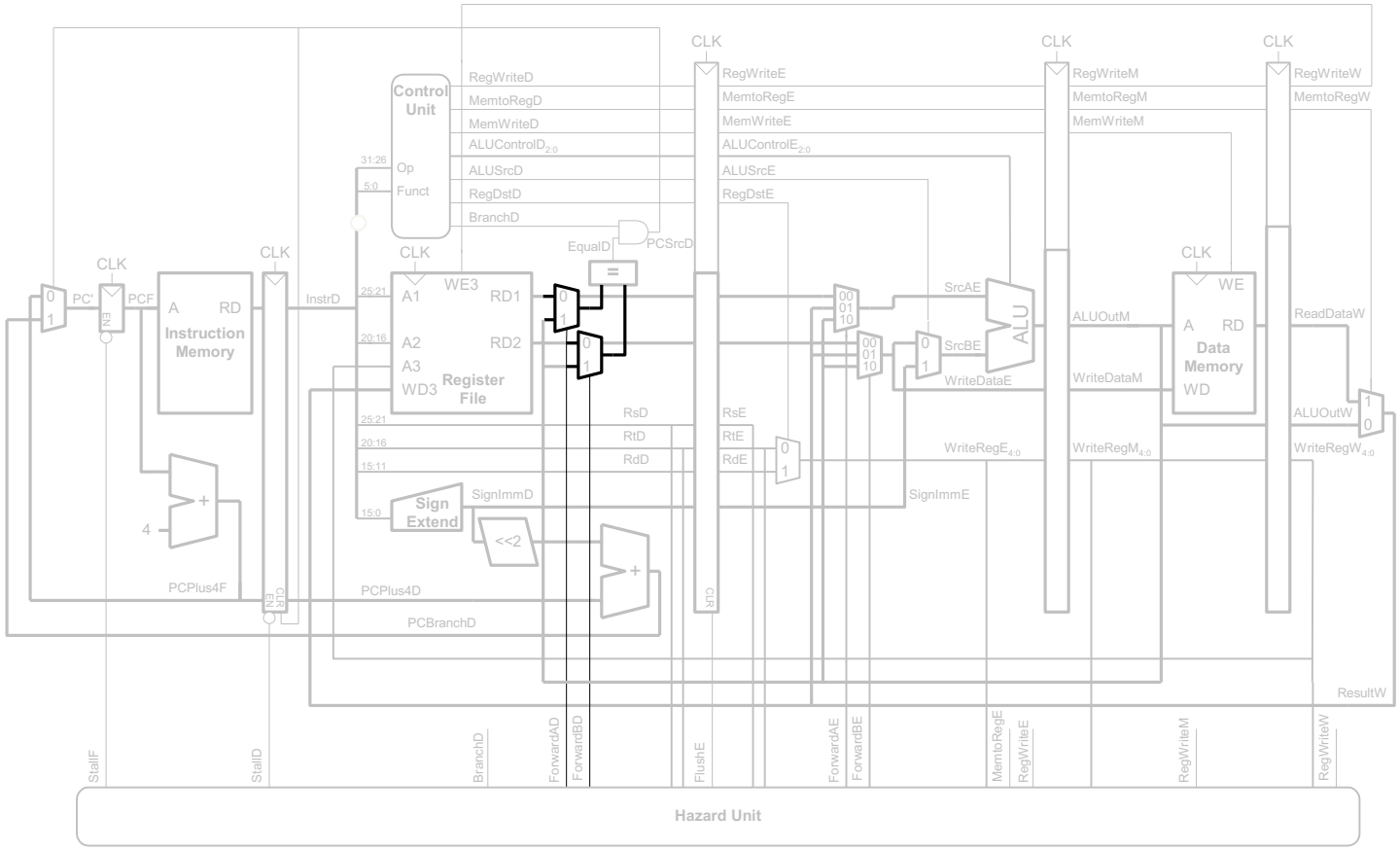
What can be changed?

Avoiding control hazards: Modification for early branch resolution

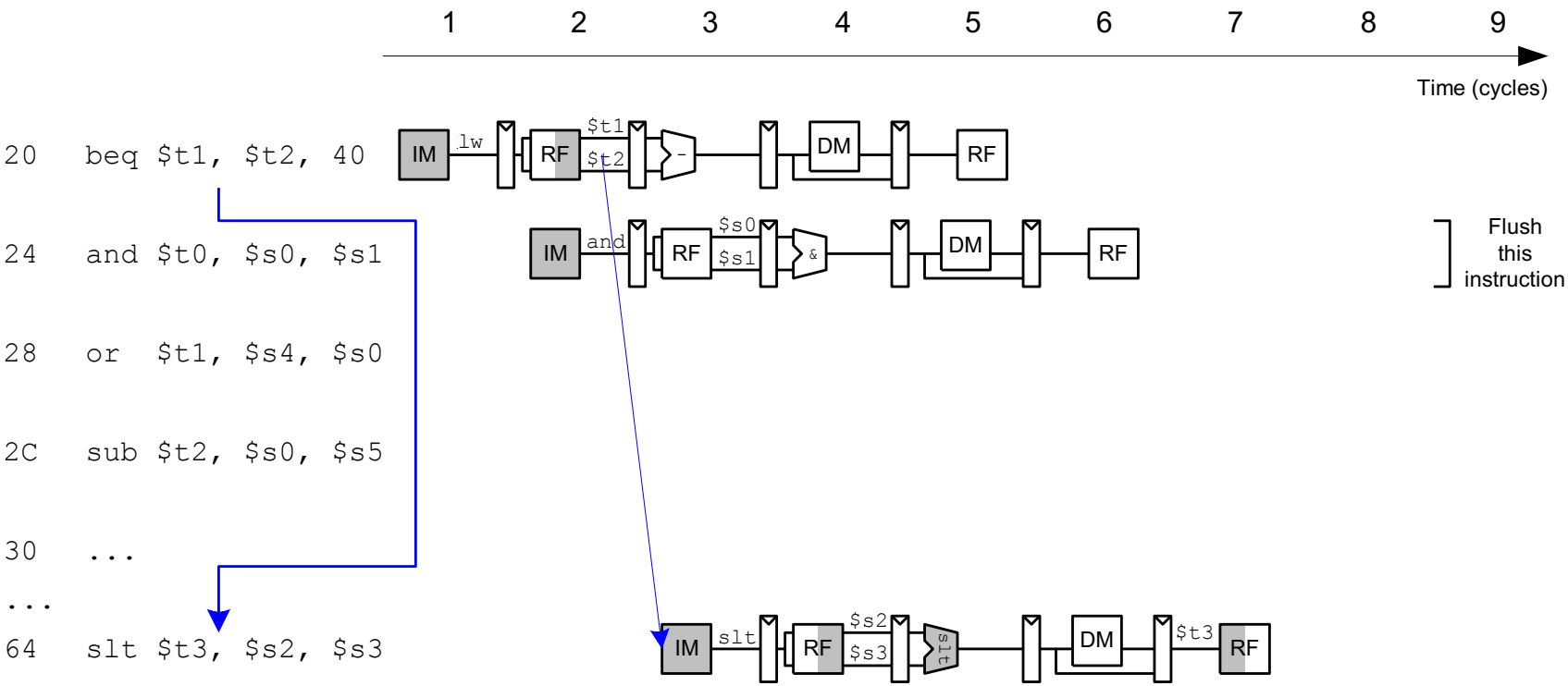


Additional data hazard in decode phase!

Avoiding control hazards: Modification for early branch resolution

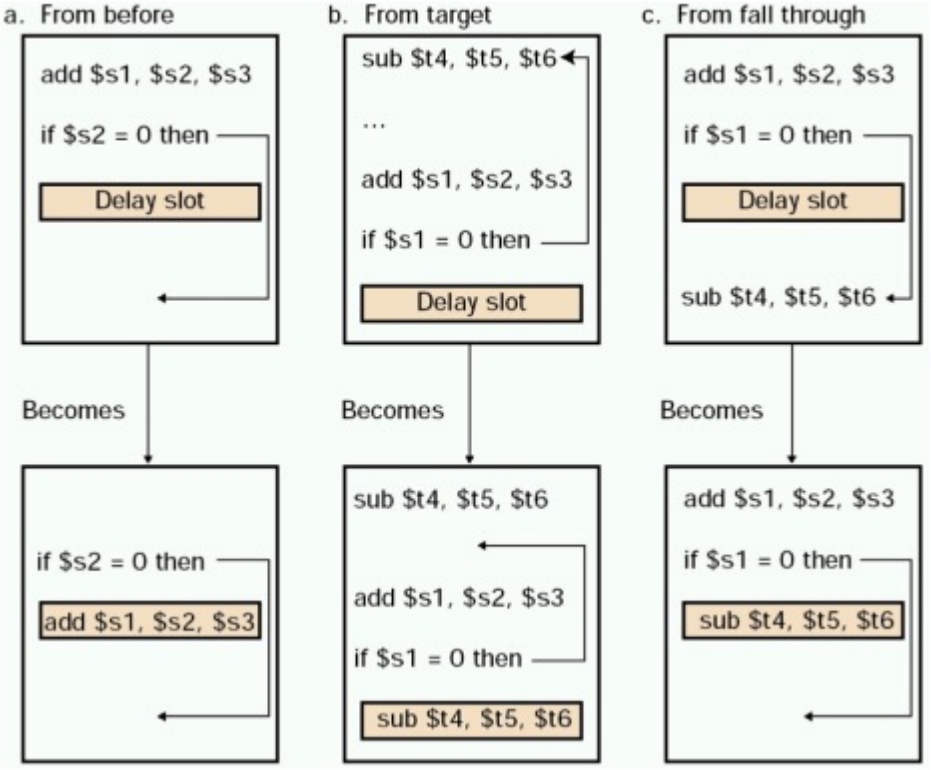


Effect of early branch resolution



Avoiding control hazards: Branch delay slots

- The semantics of the branch instruction is modified:
  The instruction following the branch instruction is executed **in any case**.
- The compiler needs to find instructions to fill this "branch delay slot" without violating the program's semantics.



Avoiding control hazards

Computation of the average cycles per instruction (CPI):

- Assumptions: five-stages pipeline, 1 cycle per stage, only stalls due to branches, 30% of all instructions are branches.
  The compiler can fill the branch delay slot in 60% of cases.
- CPI = 0,7 + 0,3*2 – 0,3*0,6 = 1,12

Summary: Pipelining

- Overlap instructions of consecutive instructions
- Three idealizing assumptions:
  - **Homogeneous partial operations**
    → *otherwise*: internal fragmentation
  - **Repetition of identical operations**
    → *otherwise*: external fragmentation
  - **Repetition of independent operations**
    → *otherwise*: hazards
    - Stalling
    - Forwarding
    - Branch Delay Slot