Advanced Computer Architecture

LECTURE 1: PIPELINING

JAN LEMEIRE

SHEN & LIPASTI CHAPTERS 2 & 4

Historic Perspective

- First employed in early 1960s
- During the 1980s, it was the cornerstone of RISC approach
- Intel i486 was first pipelined CISC processor (1989)
- Today, almost all processors are pipelined

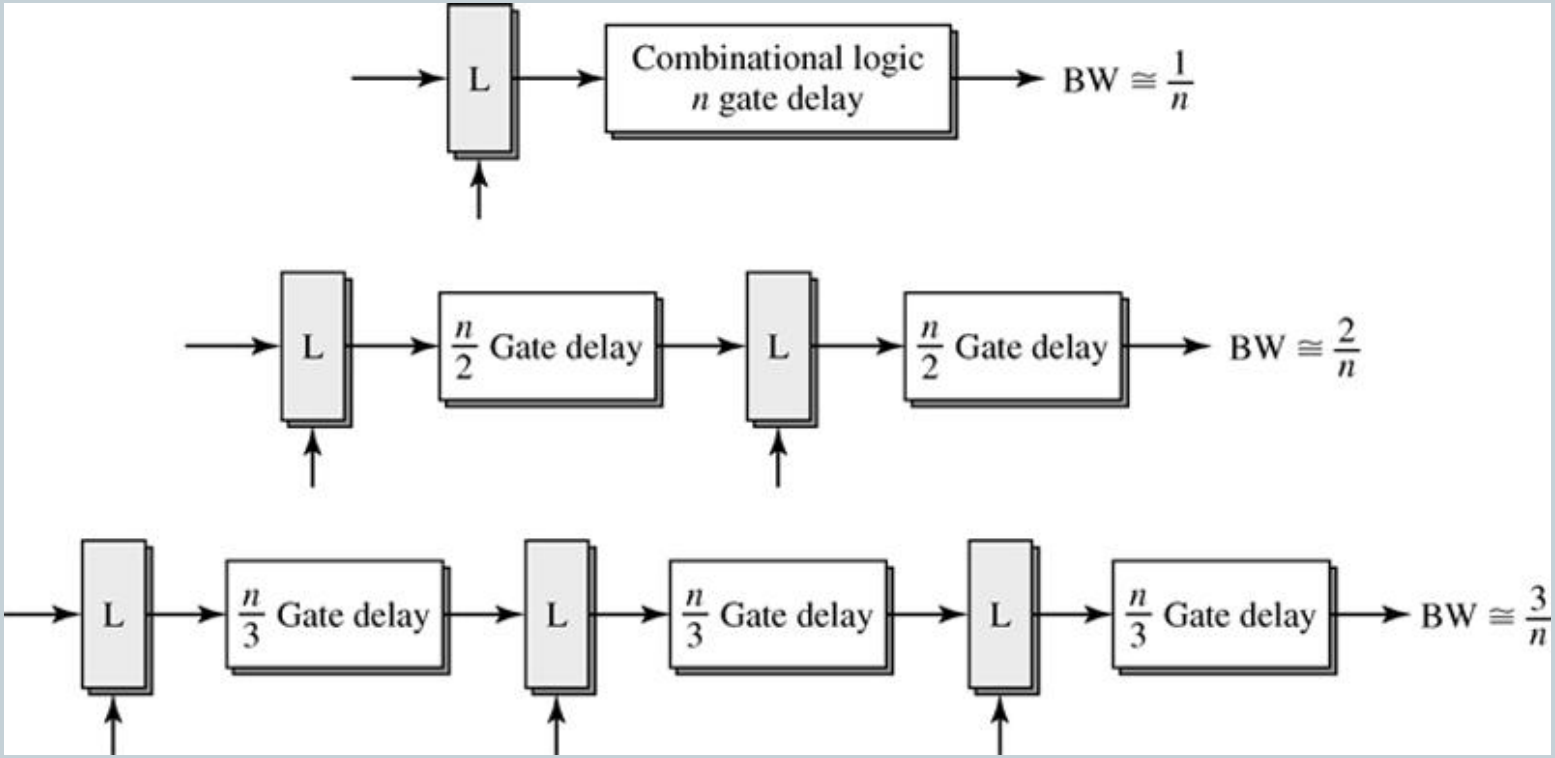
Throughput vs Latency

- **Latency**: time between start and finish of a job
- **Throughput**: number of jobs per second/hour/day/…

- Example: sending letters via airmail
  - More letters on a plane: more throughput, but same latency
  - More planes with same amount of letters: latency decreases and throughput increases
  - Less planes, with more letters: same throughput, higher latency

Pipelining Principle

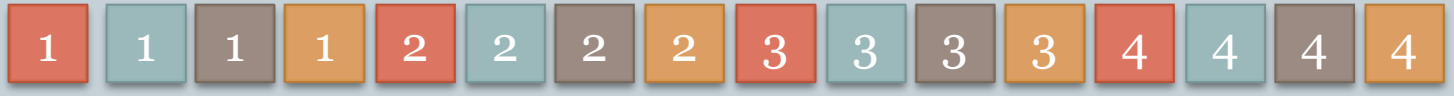
- Goal: increase throughput without much additional HW and without additional latency



Pipelining Principle

- Long operations

- Combination of short operations



- Pipelining

Pipelining Overhead

- Pipelining Idealisms
  - Uniform subcomputations
    - Can pipeline into stages with equal delay
    - Balance pipeline stages, otherwise have **internal fragmentation**
    - Also: no additional delay by the interstage buffers/clocking requirements
  - Identical computations
    - Can fill pipeline with identical work and no unused pipeline stages
    - Unify instruction types (example later) to avoid external fragmentation
  - Independent computations
    - No relationships between work units
    - Minimize pipeline stalls (dynamic external fragmentation)
- Are these practical?
  - No, but can get close enough to get significant speedup
  - *Deviations determine performance loss (inefficiencies)*

Pipeline Design Tasks

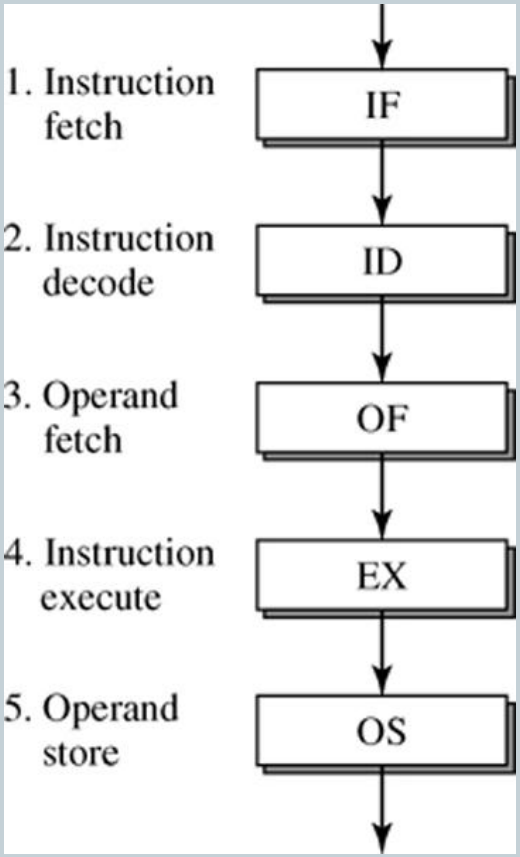
- Perform stage quantization
  - to create uniform, balanced pipeline stages
- Unify different resource requirements for different instruction types
  - to minimize the underuse of resources
  - to enable execution of all instruction types
- Deal with dependent operations
  - Since not all instructions will be independent
  - Must not punish execution of independent instructions

Impact on ISA

- Uniform subcomputations
  - longest latency undividable instruction must be found
  - is usually the memory access
  - this should not be slowed down by complex addressing modes
  - caches are required because of high latency to main memory
- Unifying the resource requirements
  - is easier for simple, less diverse RISC instructions
- Deal with dependencies
  - is very hard in HW with complex addressing modes
  - can be done in HW (RISC) or in SW (VLIW)

Pipeline Design: Balanced Stages (1)

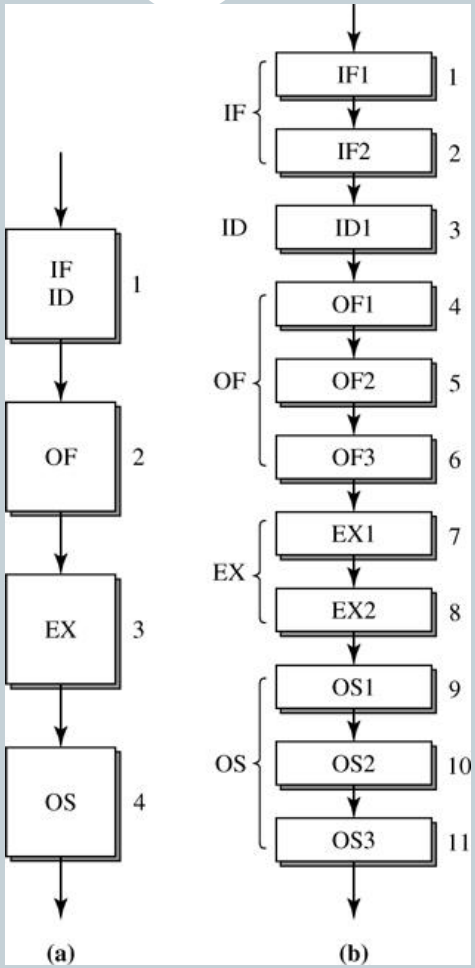
- Typically five tasks in instruction execution
  - IF: instruction fetch
  - ID: instruction decode
  - OF: operand fetch
  - EX: instruction execution
  - OS: operand store, often called write-back WB



Pipeline Design: Balanced Stages (2)

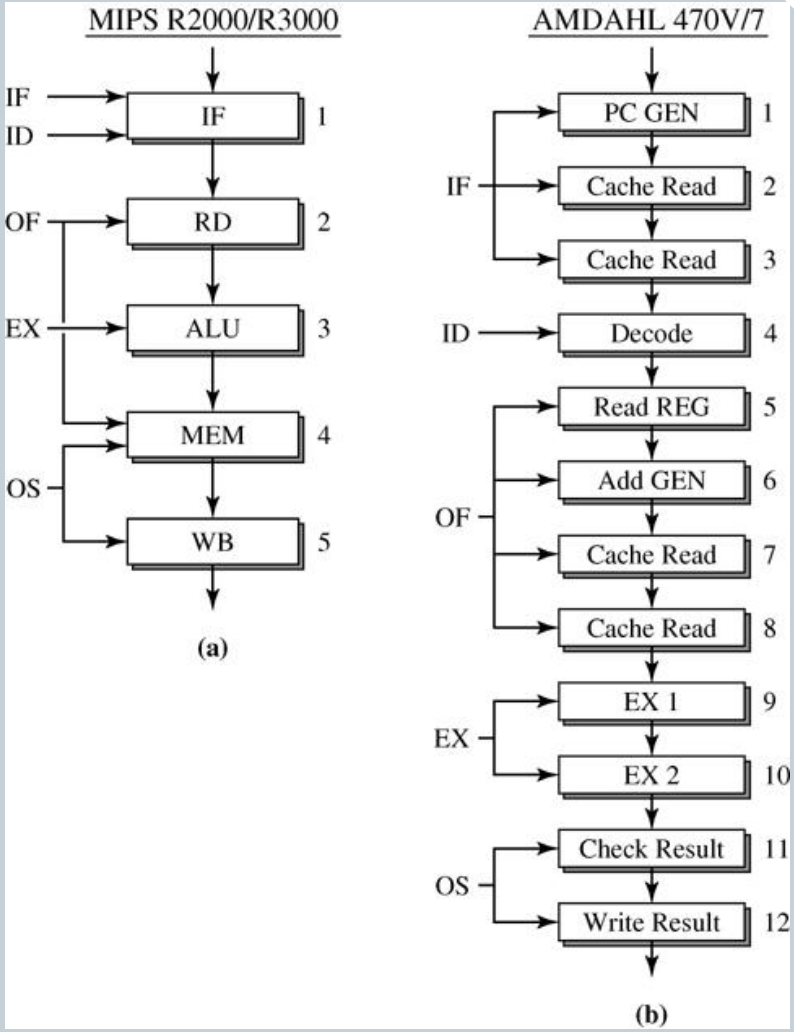
Two techniques

1) merge stages



2) subdivide stages

Pipeline Design: Balanced Stages (3)

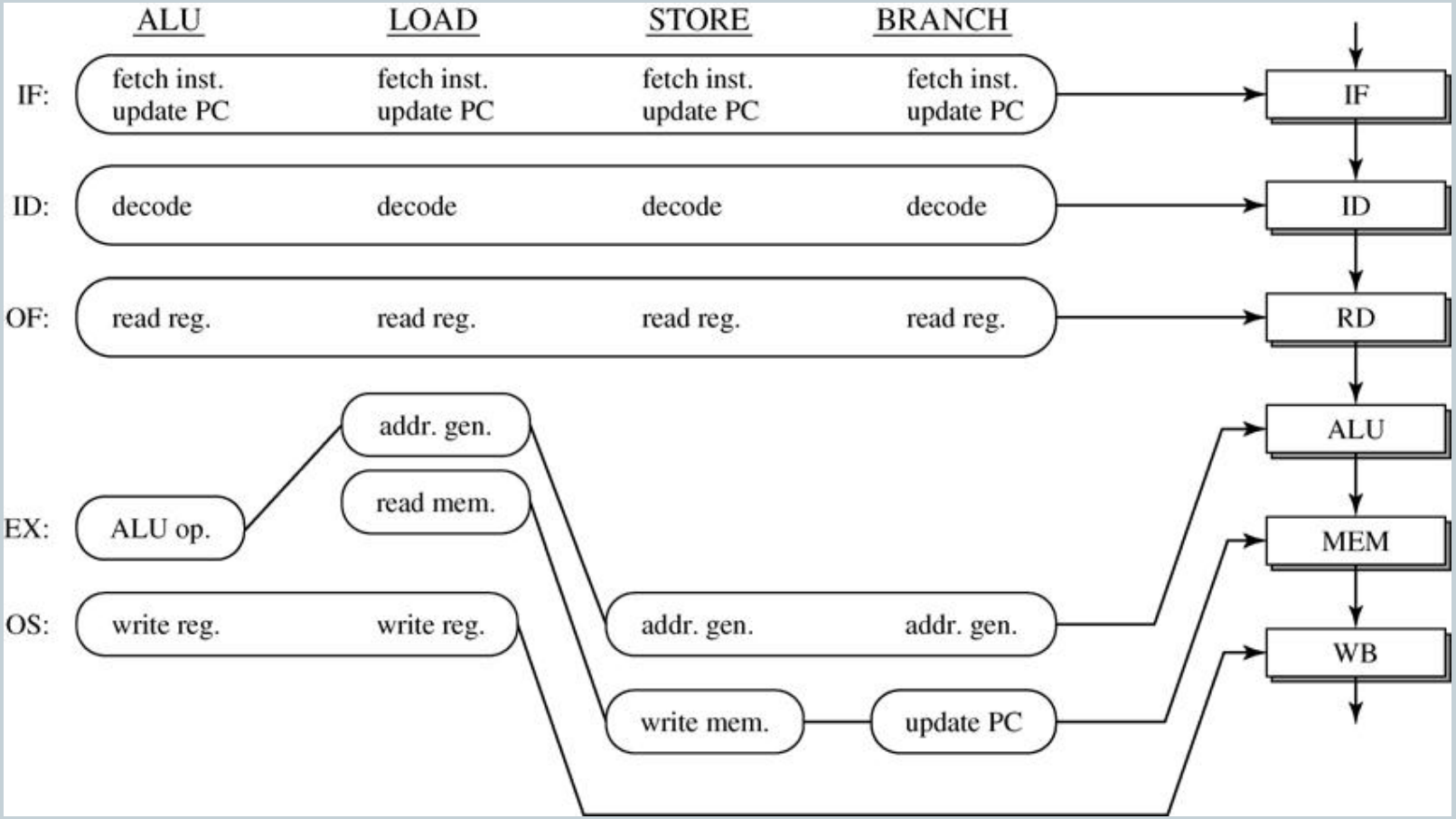


- More stages is more complex
  - more concurrent register file accesses
  - more concurrent memory accesses
  - pipelined memory accesses are complex

Pipeline Design: Unified Instruction Types (1)

- Different types of instructions
  - ALU instructions
  - Memory accesses
  - Branch Instructions
- Coalescing of requirements
  - Analyze subcomputation sequences and resource requirements
  - Find commonalities and merge them
  - In case of flexibility, shift or reorder subcomputations to ease merging

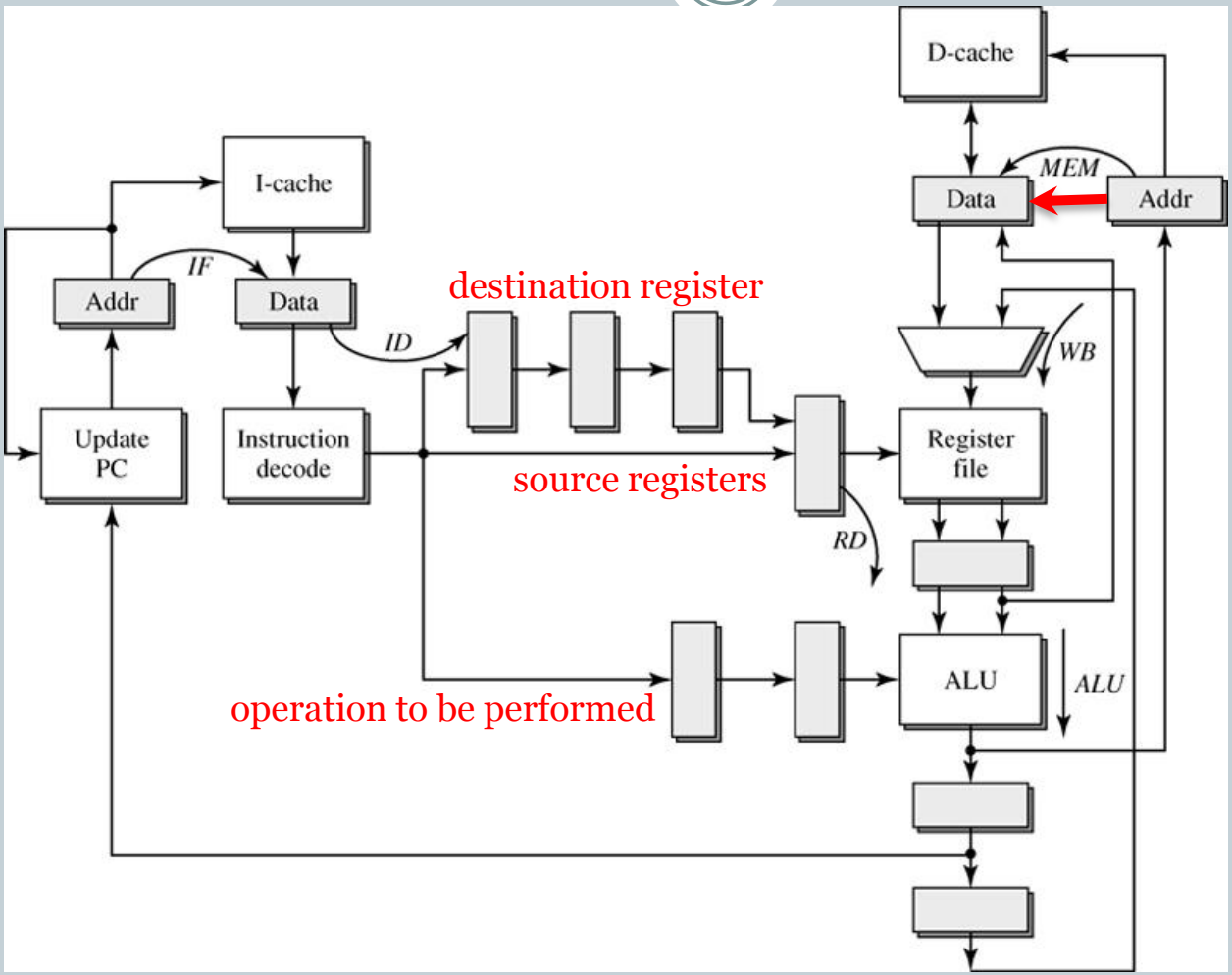
Pipeline Design: Unified Instruction Types (2)



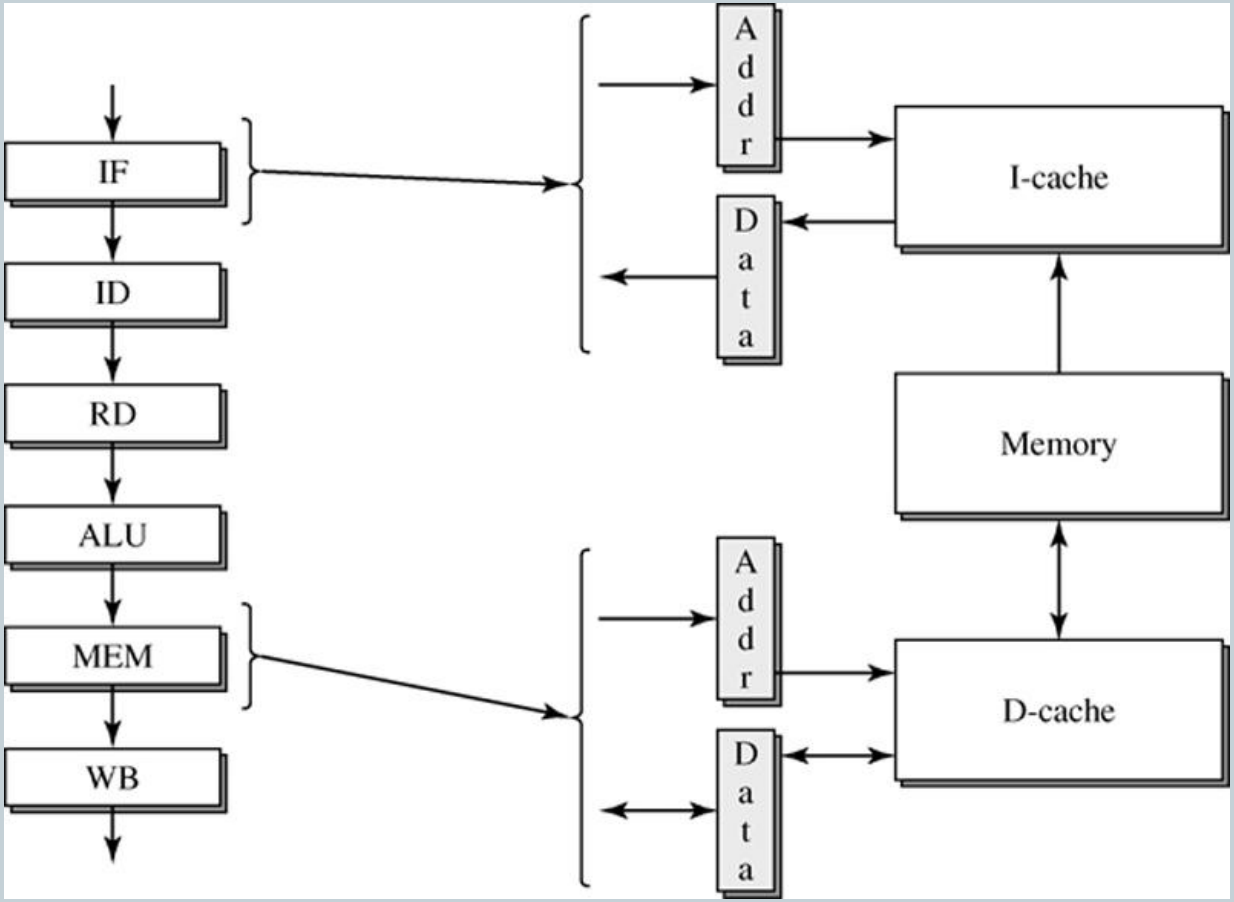
Pipeline Design: Unified Instruction Types (3)

- Objectives
  - minimize the total number of resources
  - maximize utilization, minimize idling stages
  - limit instruction latency
  - put idle cycles at the end (to minimize dependencies)

Pipeline Design: Unified Instruction Types (4)



Pipeline Design: Unified Instruction Types (5)



Pipeline Design: Minimize Pipeline Stalls (1)

- Multiple instructions in pipeline, in different stages
- Be carefull with data dependencies

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}
ID		I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
RD			I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
ALU				I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM					I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB						I_1	I_2	I_3	I_4	I_5	I_6

Suppose I4 consumes value produced by I2.

Need to avoid that I4 reads operands before I2 writes them.

Solution: detect dependence and delay instruction I4.

Pipeline Design: Minimize Pipeline Stalls (2)

- Data dependencies
  - True dependence = read-after-write (RAW)

```
V3 ← V1 op V2
V4 ← V3 op V5
```

  - Anti dependence = write-after-read (WAR)

```
V3 ← V1 op V2
V1 ← V4 op V5
```

  - Output dependence = write-after-write (WAW)

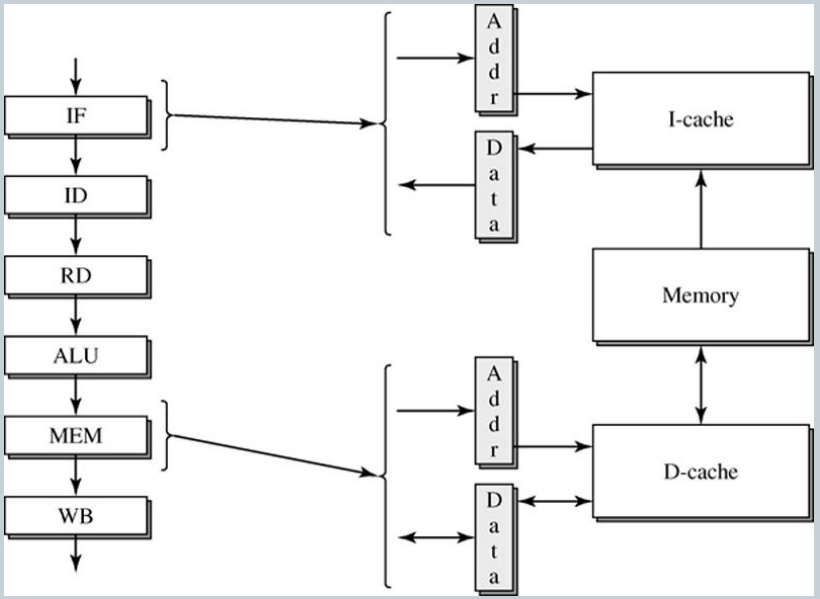
```
V3 ← V1 op V2
V3 ← V4 op V5
```

- A hazard: messing up the program by not respecting a dependency

Pipeline Design: Minimize Pipeline Stalls (3)

- Dependencies Through Memory?

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}
ID		I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
RD			I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
ALU				I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM					I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB						I_1	I_2	I_3	I_4	I_5	I_6

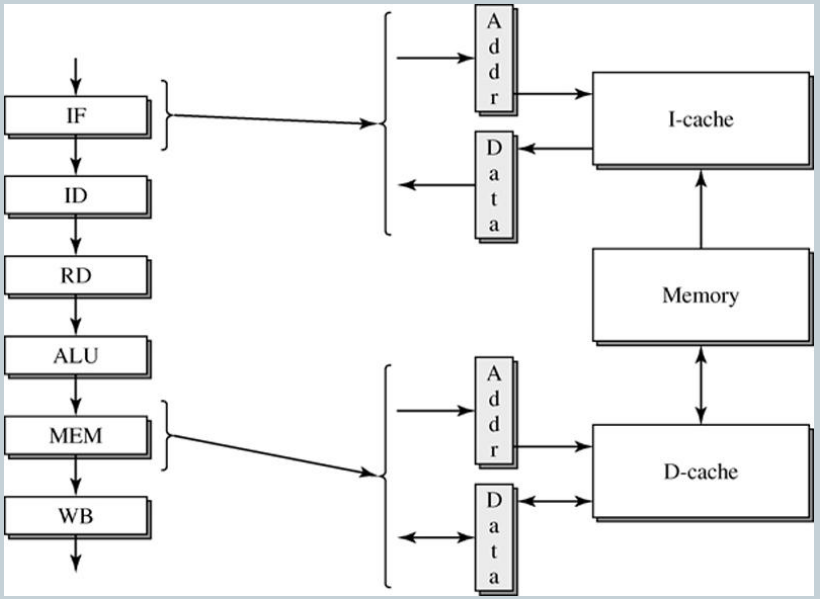


Not possible since (1) only one stage accesses memory, and
(2) all instructions pass through mem stage in program order.

Pipeline Design: Minimize Pipeline Stalls (4)

- What about WAW hazards through registers?

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}
ID		I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
RD			I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
ALU				I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM					I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB						I_1	I_2	I_3	I_4	I_5	I_6

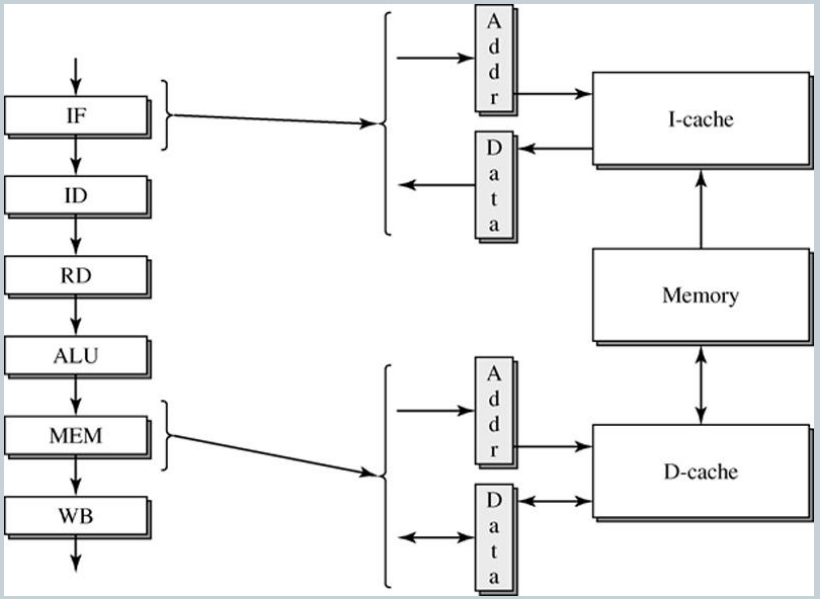


Not possible since (1) all writes to register happen in single WB stage, and
(2) all instructions pass through WB stage in program order.

Pipeline Design: Minimize Pipeline Stalls (5)

- What about WAR hazards through registers?

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}
ID		I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
RD			I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
ALU				I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM					I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB						I_1	I_2	I_3	I_4	I_5	I_6



Not possible since (1) writes occur in WB stage after reads in RD stage, and
(2) all instructions pass through these stages in program order

Pipeline Design: Minimize Pipeline Stalls (6)

- RAW hazards

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_1	I_2	I_3	I_4	I_5	I_6	I_6	I_6	I_7	I_8	I_9
ID		I_1	I_2	I_3	I_4	I_5	I_5	I_5	I_6	I_7	I_8
RD			I_1	I_2	I_3	I_4	I_4	I_4	I_5	I_6	I_7
ALU				I_1	I_2	I_3			I_4	I_5	I_6
MEM					I_1	I_2	I_3			I_4	I_5
WB						I_1	I_2	I_3			I_4

Suppose I4 consumes value produced by I2.

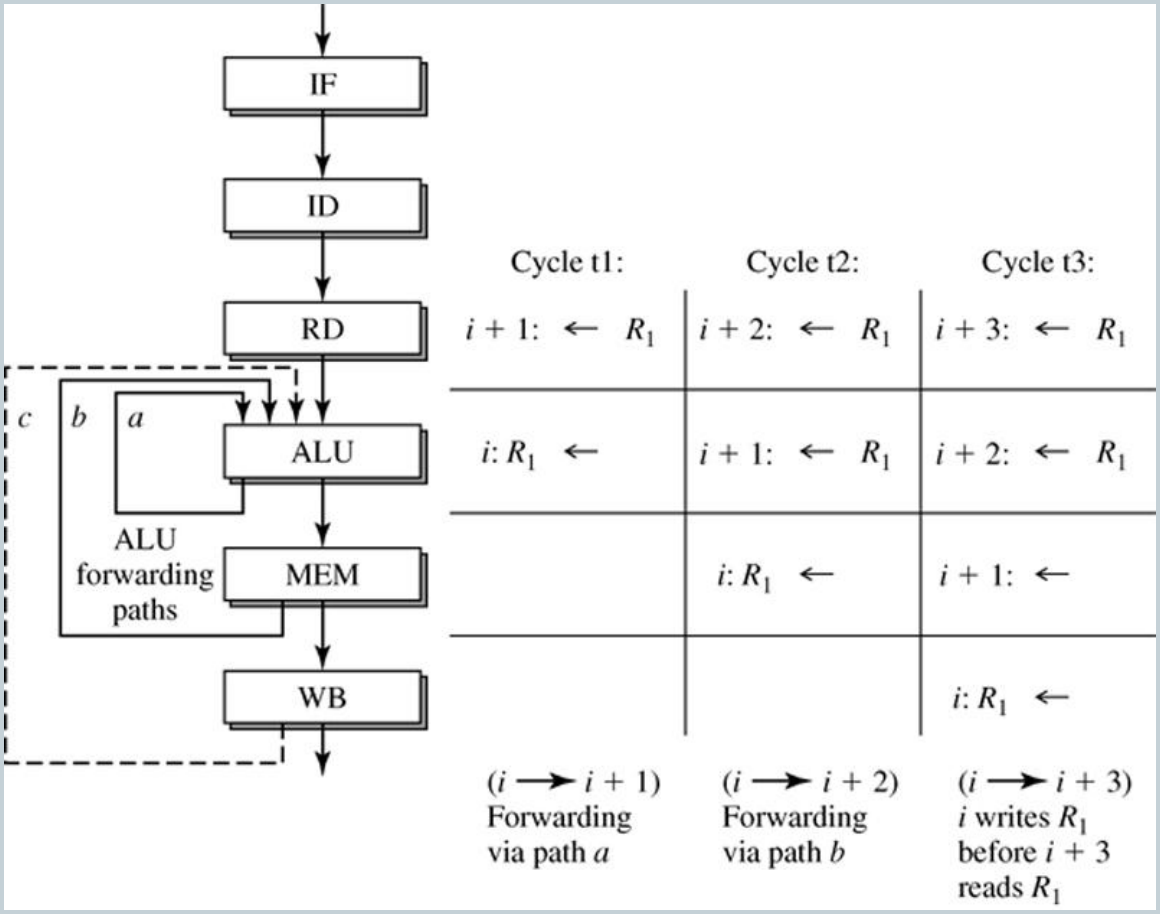
Need to avoid that I4 reads operands before I2 writes them.

Solution: detect dependence and delay instruction I4.

Result: pipeline bubble or stall, performance drop

Pipeline Design: Minimize Pipeline Stalls (7)

- RAW hazards: forwarding



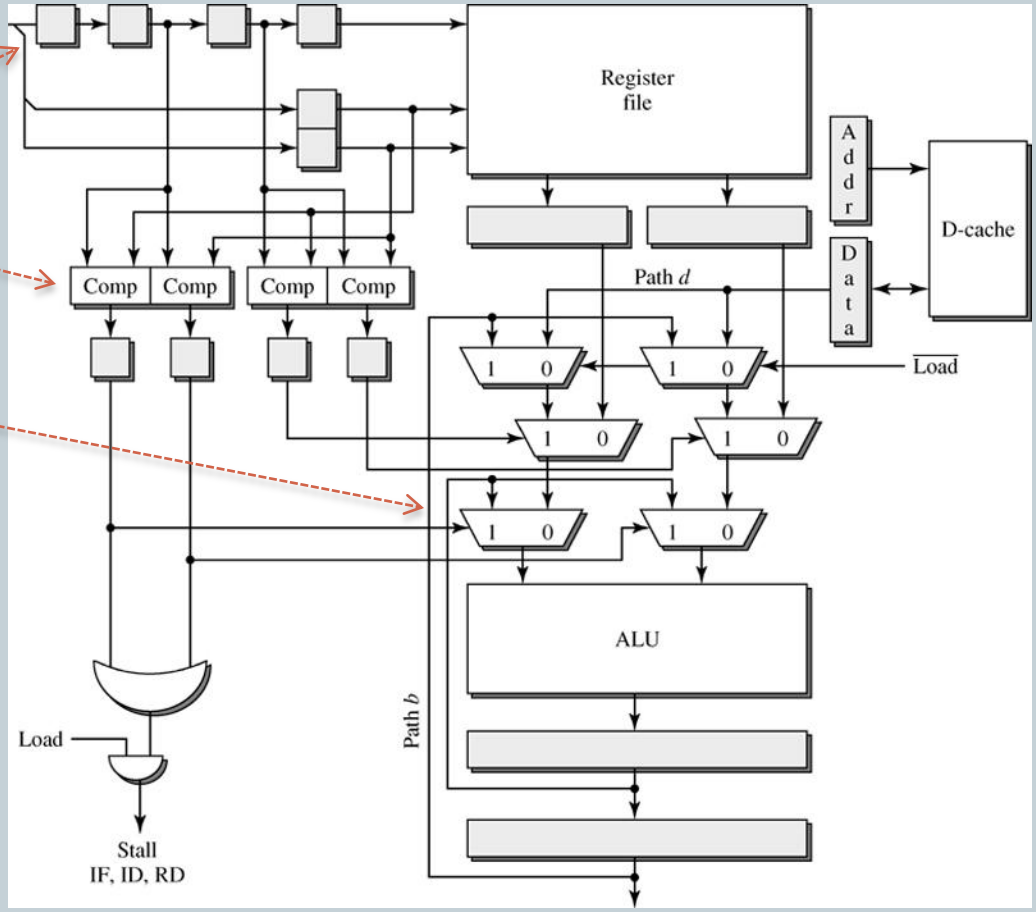
Pipeline Design: Minimize Pipeline Stalls (8)

- Forwarding implementation

Delay lines

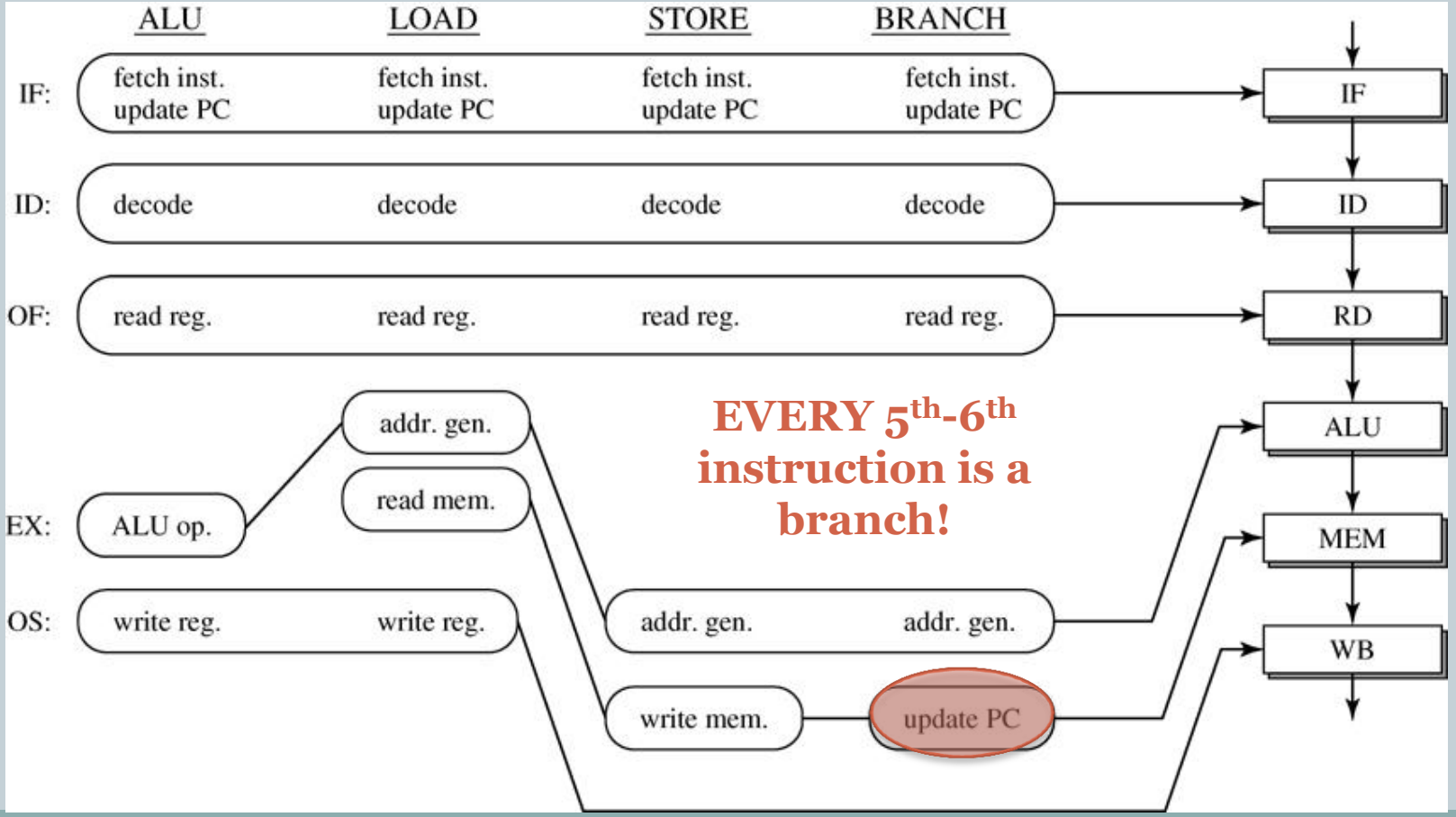
Comparators

Multiplexors



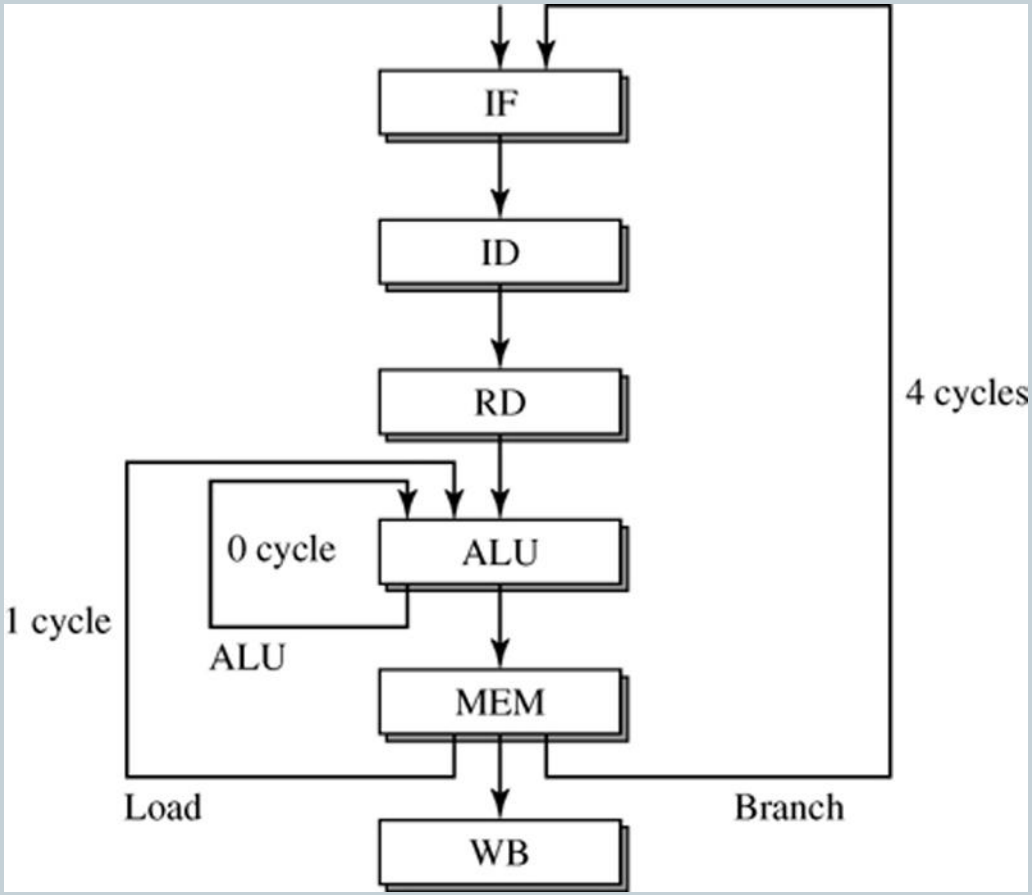
Pipeline Design: Minimize Pipeline Stalls (9)

- What about control dependencies and branches



Pipeline Design: Minimize Pipeline Stalls (A)

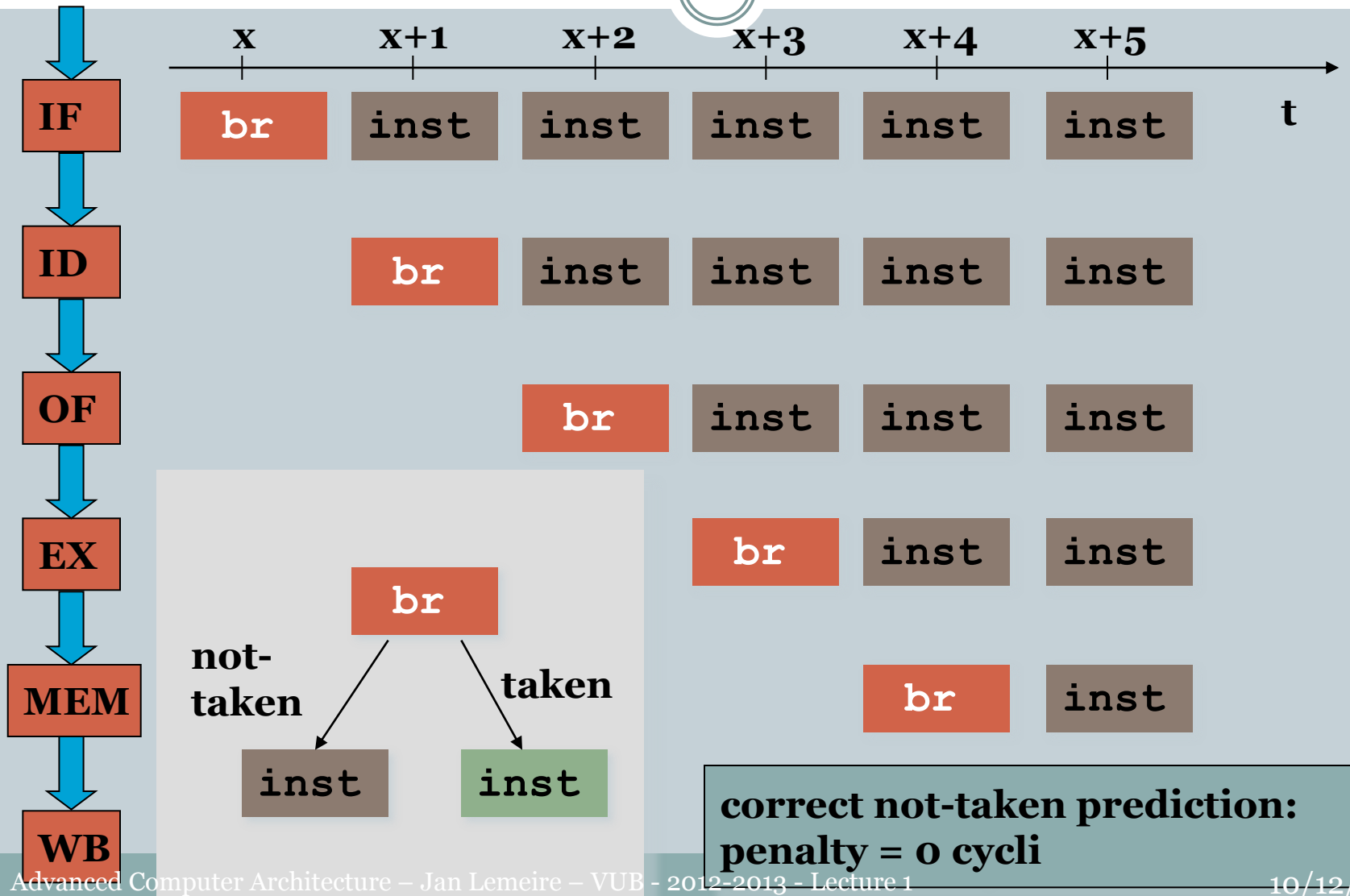
- Similar solution: branch forwarding to save a cycle



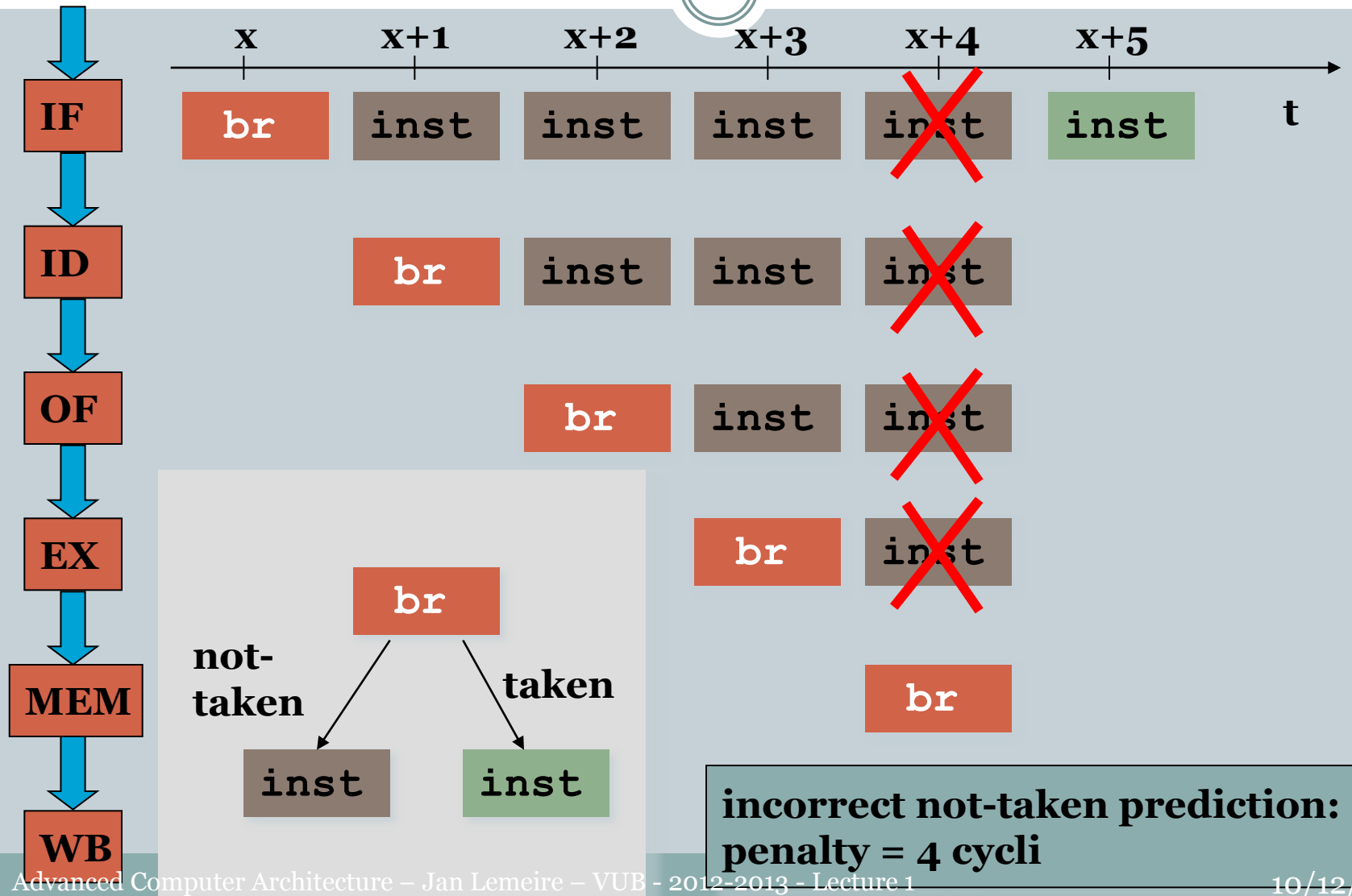
Pipeline Design: Minimize Pipeline Stalls (B)

- Additional solution: branch not taken prediction
  - keep fetching & decoding & …
  - If branch is not taken, oke, keep executing
  - If branch is taken: flush all the instructions past the branch from the pipeline, and start fetching again
- Another solution: delay slot
  - keep fetching & decoding & executing delay slot instructions
  - compiler places only instructions in the delay slot that are executed on taken and non-taken path
  - instructions in delay slot need not be flushed!

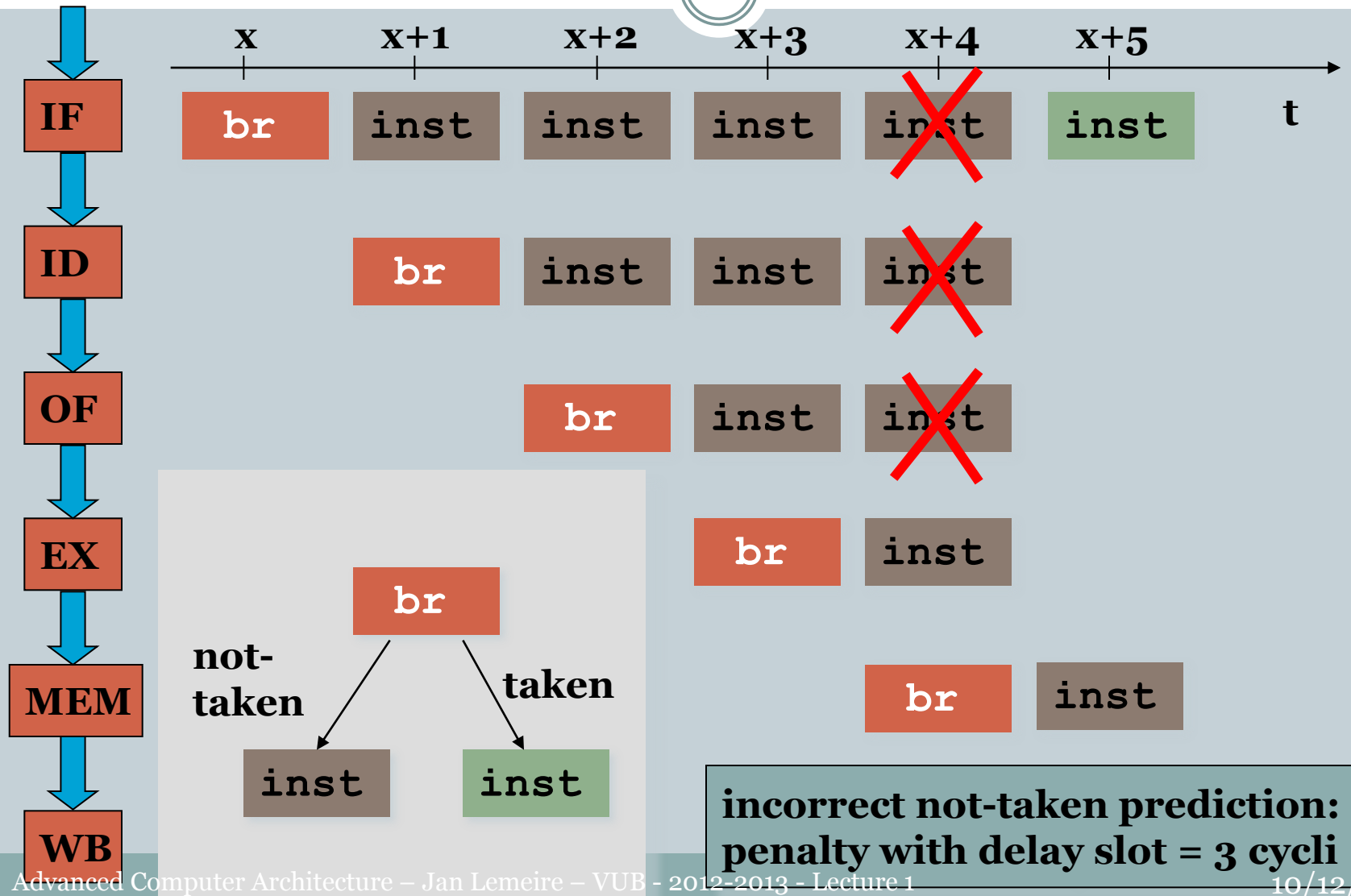
Pipeline Design: Minimize Pipeline Stalls (C)



Pipeline Design: Minimize Pipeline Stalls (D)



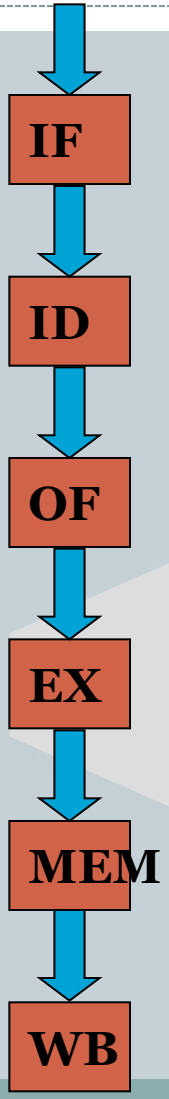
Pipeline Design: Minimize Pipeline Stalls (E)



Limitations to Scalar Pipelines

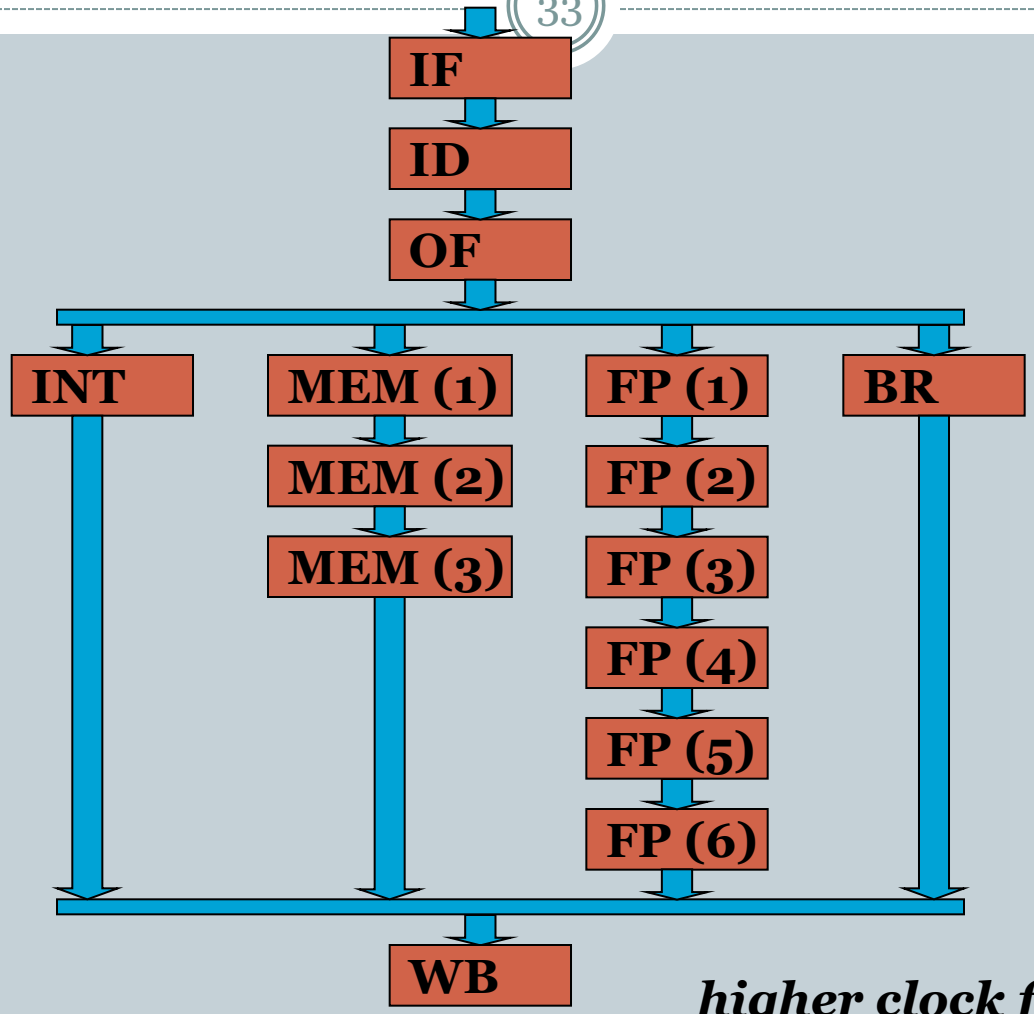
- Instruction type unification is a problem
  - e.g.: floating-point addition vs. integer addition
- Fundamental limit: IPC ≤ 1
- In-order execution: IPC < 1
  - stalls caused by dependencies

Problem 1: Unification



EX has to execute integer addition as well as floating-point addition in one cycle …
Solution: diversification

Solution 1: Diversified Pipeline

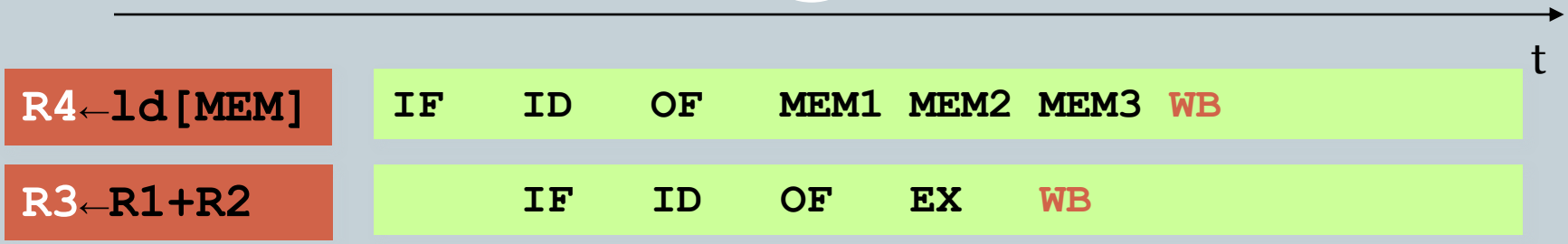


higher clock frequency than unified pipeline

Problems of Diversification

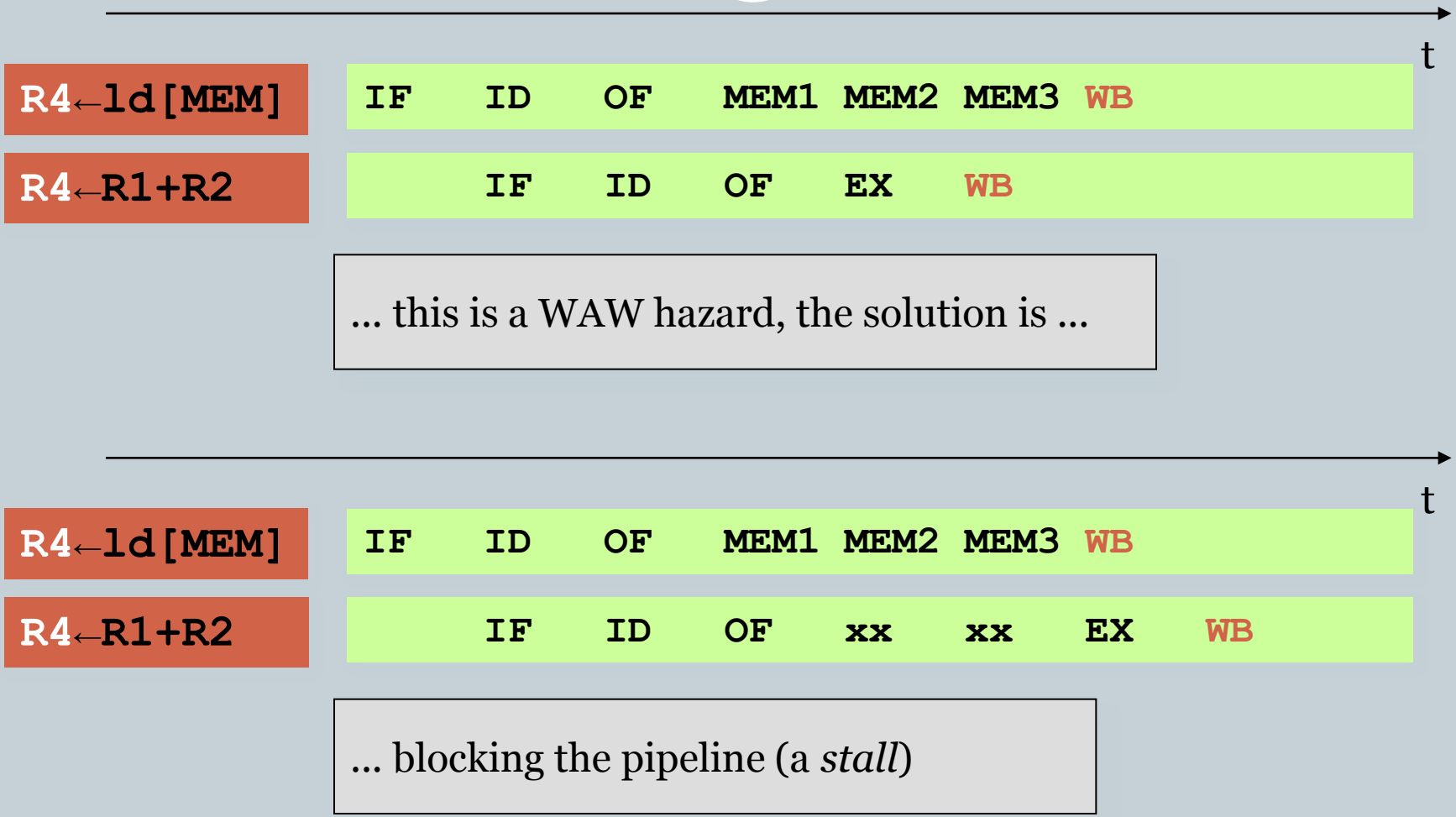
- Out-of-order completion
  - Writing back the results can happen out-of-order (= WAW hazard)
- Potentially more write operations to register file in WB stage per klok cycle (= structural hazard)
- Exceptions

Out-of-order completion

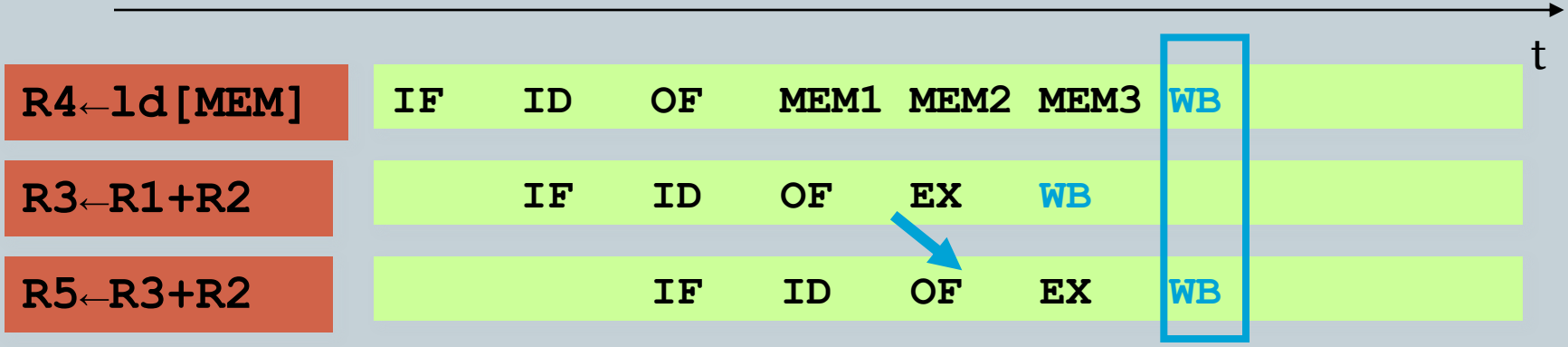


... is not really a problem (except for exceptions—see later)...

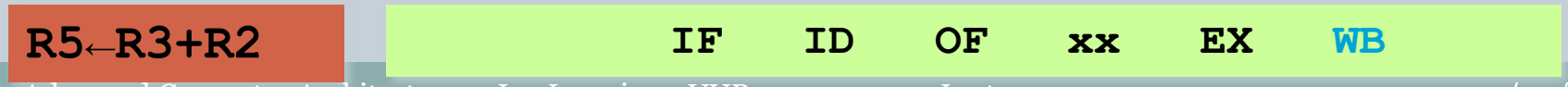
but …


t
R4←ld[MEM]
IF ID OF MEM1 MEM2 MEM3 WB
R4←R1+R2
IF ID OF EX WB
… this is a WAW hazard, the solution is …
t
R4←ld[MEM]
IF ID OF MEM1 MEM2 MEM3 WB
R4←R1+R2
IF ID OF xx xx EX WB
… blocking the pipeline (a *stall*)

Multiple WBs per cycle



- Is impossible because of only 1 write port to register file
- Solution:
  - Add a write port, or
  - Treat the write port as *structural hazard* (i.e. an additional stall)



Interrupts vs. exceptions (1)

- Interrupts
  - Typically because of external factors
  - Asynchronuous with respect to program executing
  - Interrupt handling:
    - Stop fetching new instructions
    - Finish executing instructions in the pipeline
    - Save architectural state
    - Handle the interrupt
    - Restore state and continue executing the program

Interrupts vs. exceptions (2)

- Exceptions/faults
  - Caused by something in the execution of the program
  - division by zero, page fault, overflow, etc.
  - *Precise exception*
    - Store state from just before instruction
    - Handle the exception
    - Continue execution from instruction that caused the exception
  - What happens with out-of-order completion?

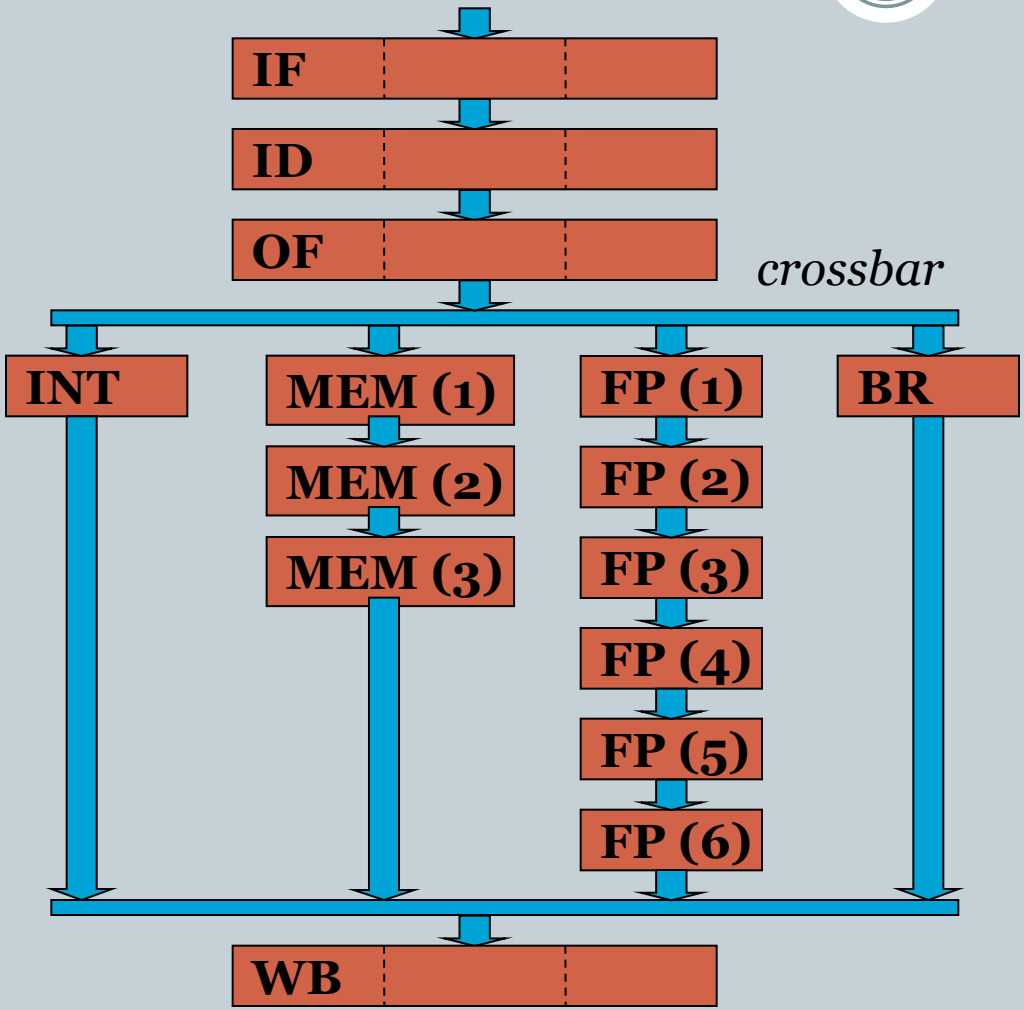
Exceptions en OoO completion

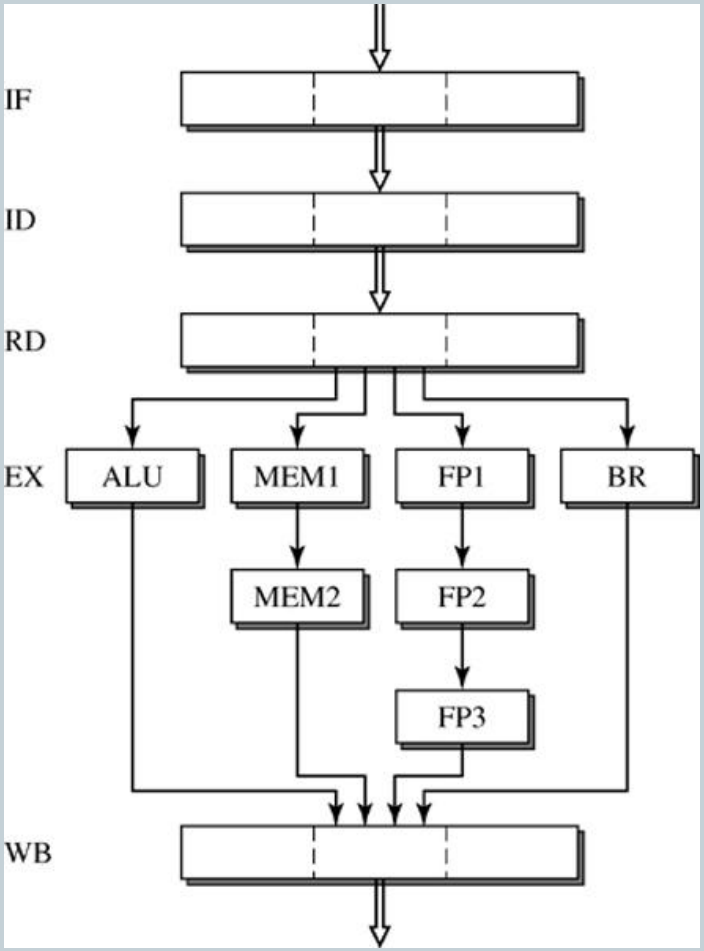
- Because of OoO completion, precise exceptions cannot be guaranteed
  - *Imprecise exceptions*
  - Hard to support in modern processors, complicates design

Limitations to Scalar Pipelines

- Instruction type unification is a problem
  - e.g.: floating-point addition vs. integer addition
  - solved with pipeline diversification
- Fundamental limit: IPC ≤ 1
- In-order execution: IPC < 1
  - stalls caused by dependencies

Solution 2: Superscalar pipeline (1)





Solution 2: Superscalar pipeline (2)

- Temporal Parallelism
  - Pipeline
  - Relatively Cheap
- Spatial Parallelism
  - Superscalar
  - Relatively expensive (more hardware)
- Superscalar pipeline
  - Both temporal and spatial parallelism
- Potential speedup of the pipeline: **depth * width**

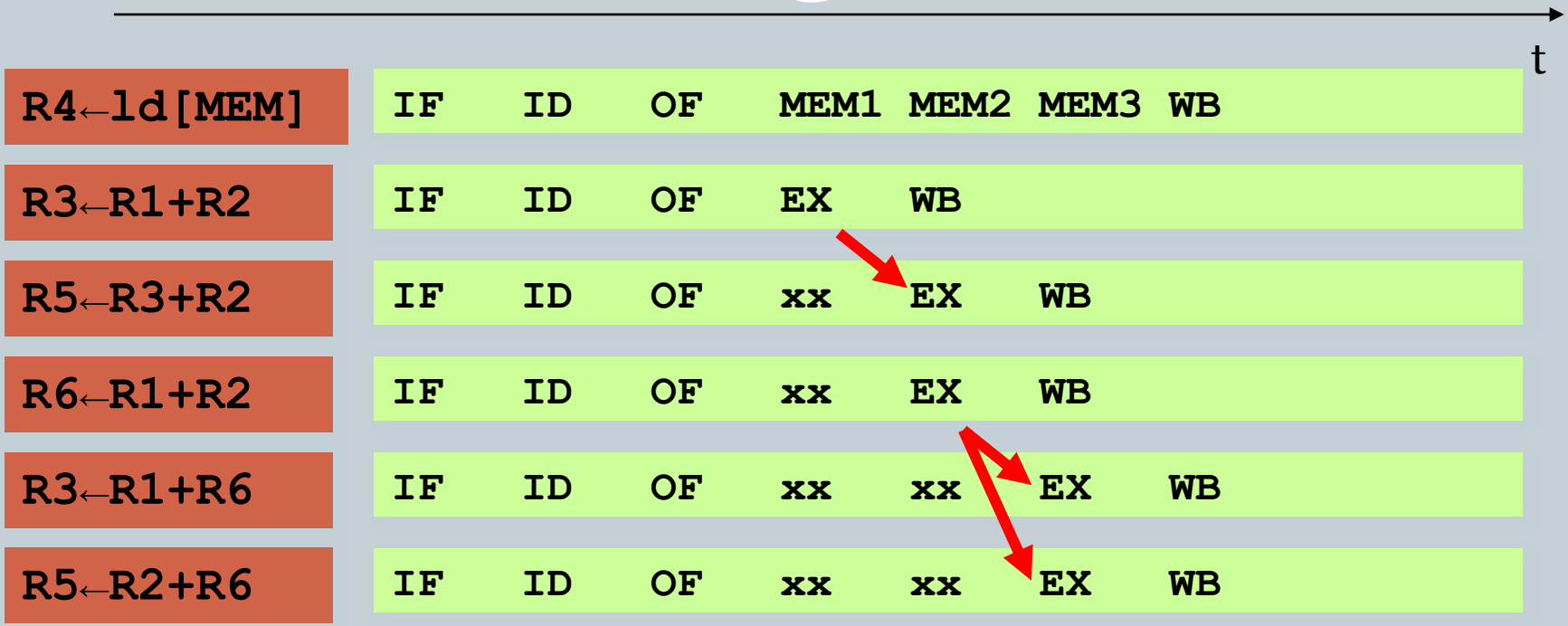
Solution 2: Superscalar pipeline (3)

- Additional HW cost:
  - More register ports
    - 2 x w read ports
    - w write ports
    - can do with less, at the expense of structural hazards
  - More bandwidth to I$ and D$ (caches)
  - Interconnections
    - To distribute instructions over pipelines
    - Complexity w^2
  - Hazard detection is more complex

Solution 2: Superscalar pipeline (4)

- Dealing with hazards
  - Typically before instruction is executed
  - i.e. *at operand fetch time*
  - Once in the function unit (execution stages of pipeline), the instruction is no longer blocked

Solution 2: Superscalar pipeline (5)

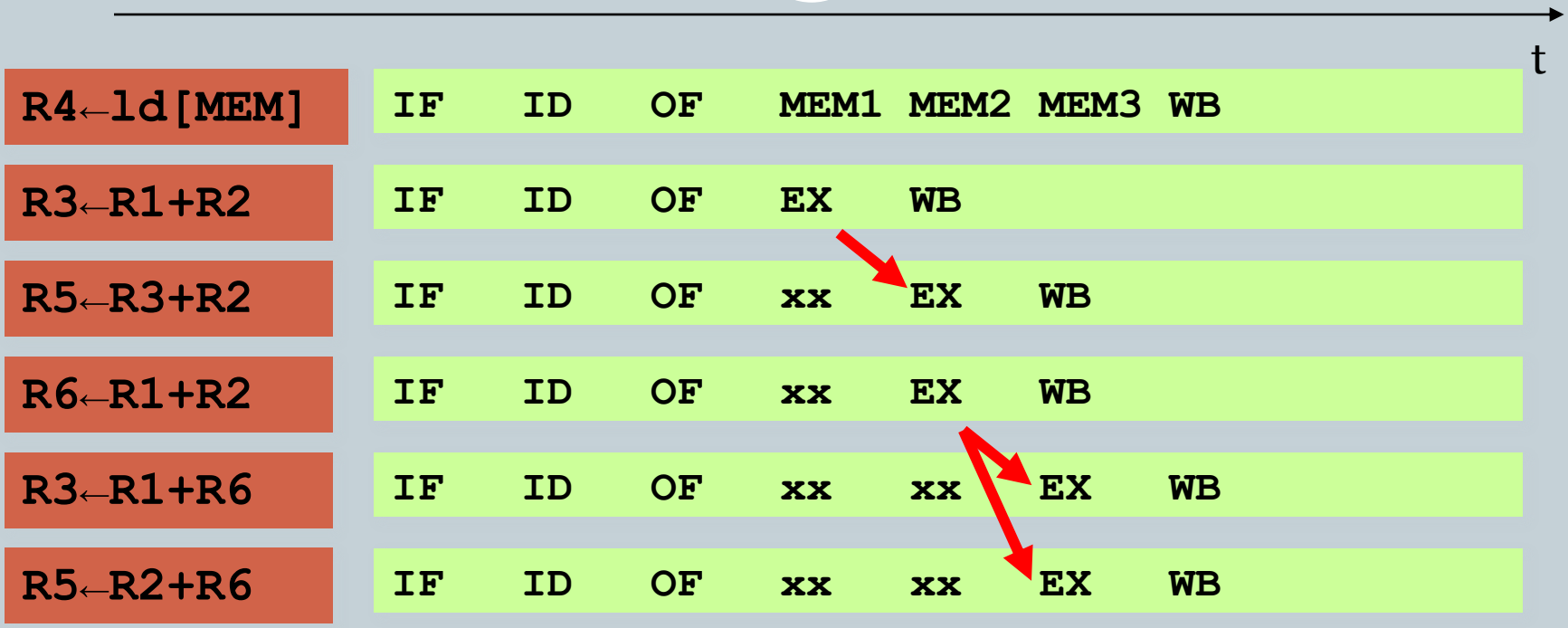


First and second instruction executed together as they are independent. Third instruction is blocked because of RAW hazard with second instruction. Blocking happens in OF stage.

Limitations to Scalar Pipelines

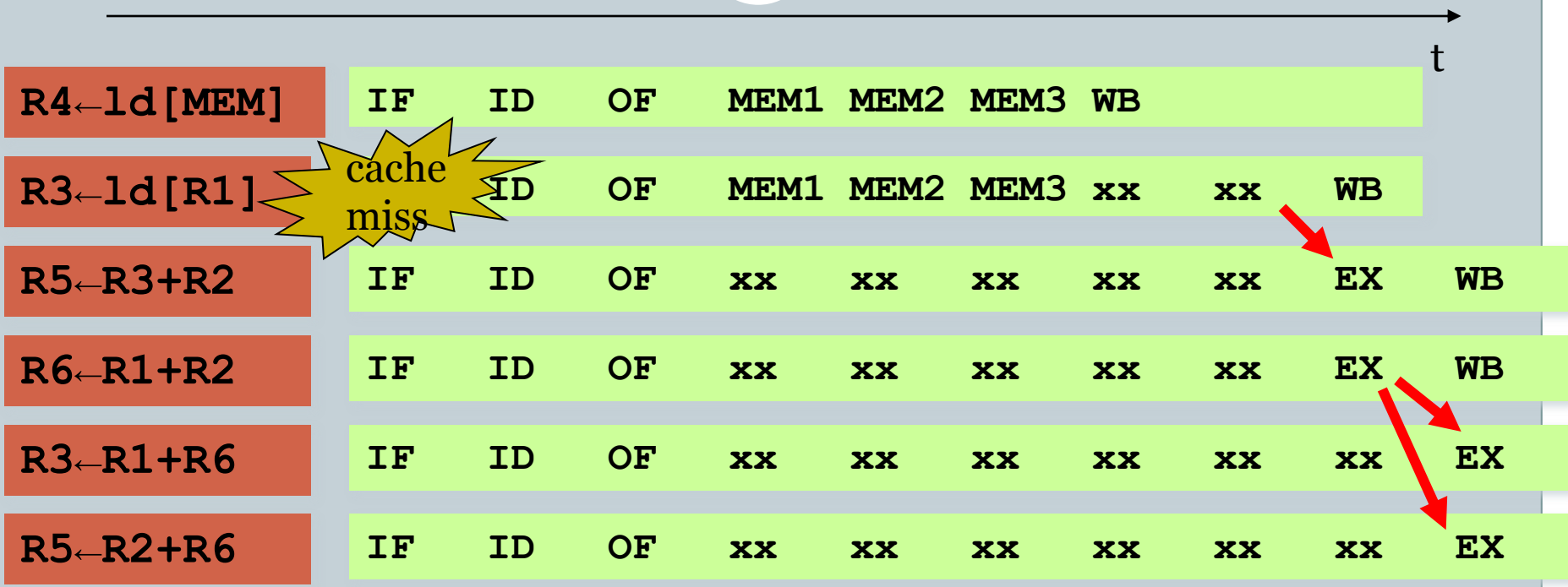
- Instruction type unification is a problem
  - e.g.: floating-point addition vs. integer addition
  - solved with pipeline diversification
- Fundamental limit: IPC ≤ 1
  - solved with superscalar pipeline
- In-order execution: IPC < 1
  - stalls caused by dependencies

Problem with in-order execution (1)



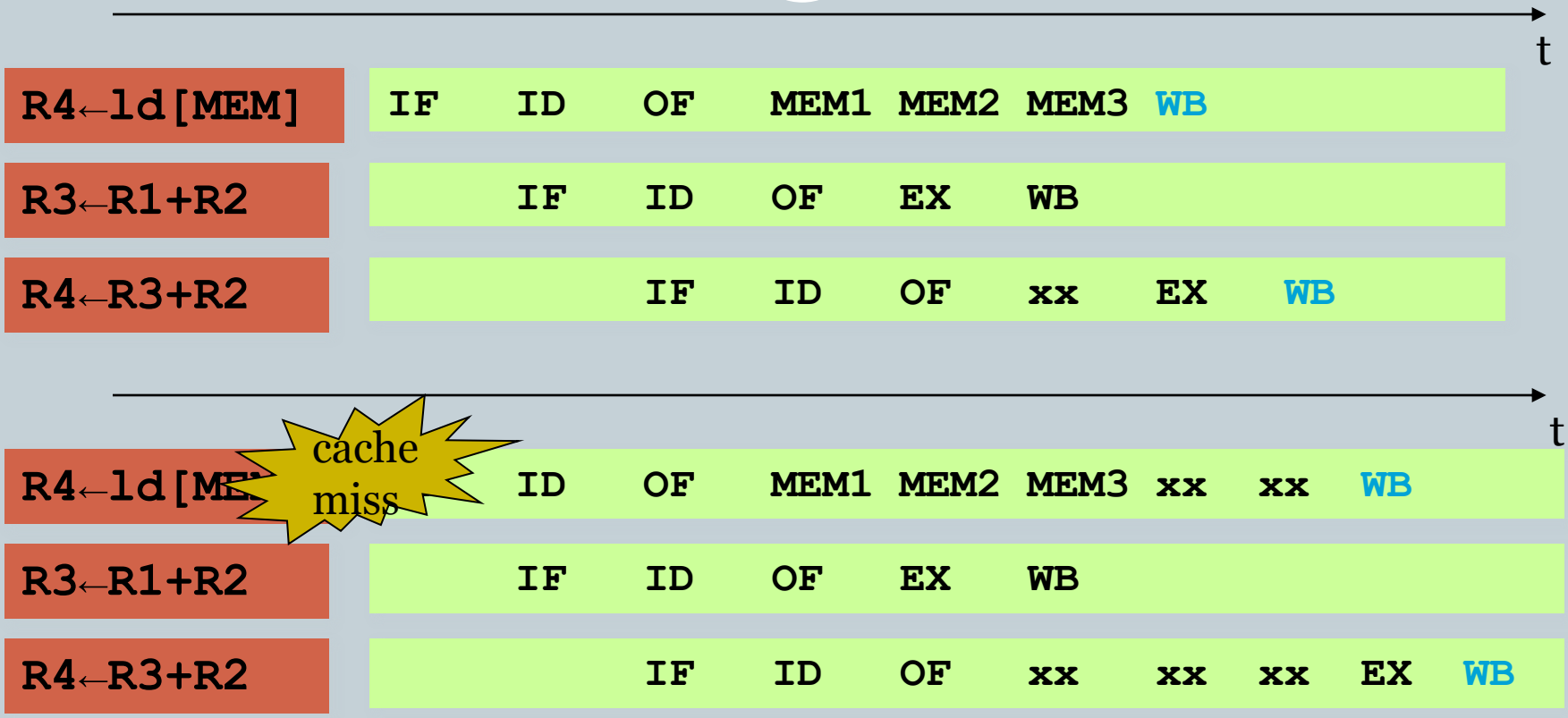
Fourth to sixth instruction are blocked when second instruction is blocked, even though they were not dependent on first two instructions. This is a fundamental problem of in-order issue machines.

Problem with in-order execution (2)



Problem becomes even worse in case of long latency blocks, such as multiplication instructions or cache misses.

Problem with in-order execution (3)



Suppose there is only one register write port ...
The cost increases very quickly with increasing instruction latencies

Problem with in-order execution (4)

Scalar pipeline: advance in lockstep fashion (in order)

Problems:

- A blocking instruction blocks all following instructions
  - even if those are independent
  - is particularly problematic for long-latency instructions
- WAW dependencies limit parallelism even though they are not real dependencies
- Why are WAR dependencies no limitation?

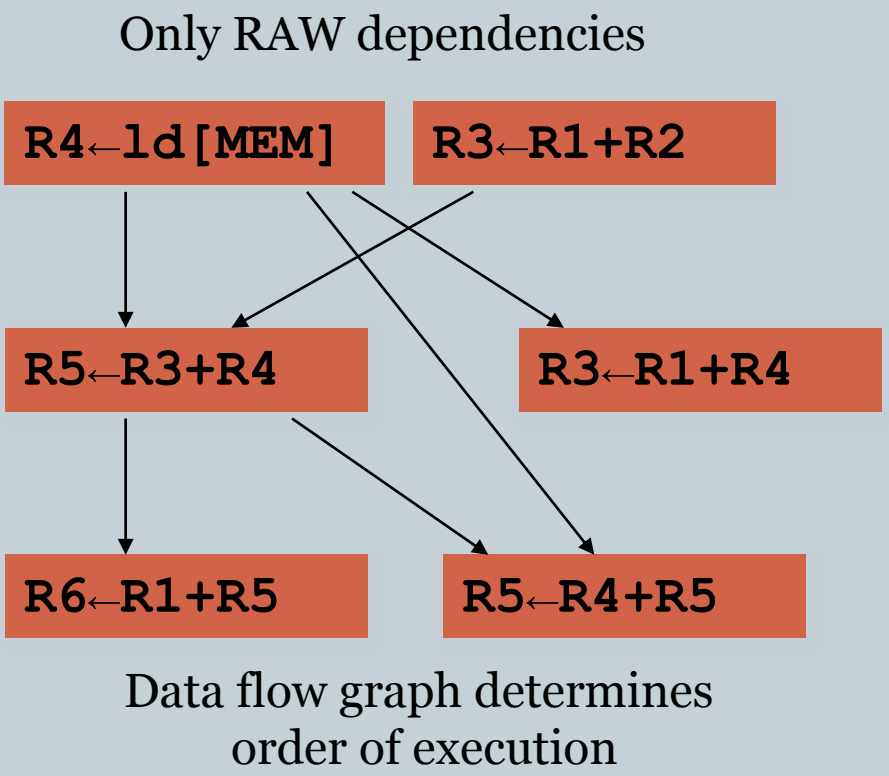
Solution 3: Out-of-order Execution (1)

- Idea:
  - True dependencies (RAW) determine execution
  - False (output and anti, WAR and WAW) dependences do not block instructions
    - Except when storing values becomes a resource constraint, see later
  - In other words, the data flow limit is reached, i.e. instructions are executed as soon as there operands are available.
- Out-of-order execution implies a *dynamic* pipeline

Solution 3: Out-of-order Execution (2)

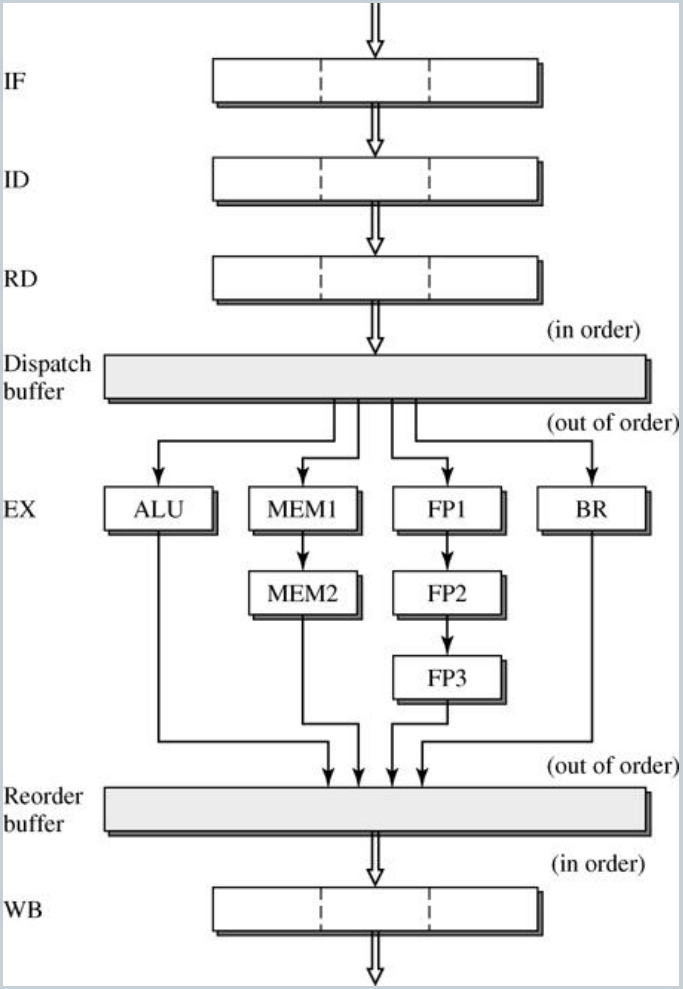
- Data flow limit

```
R4←ld[MEM]
R3←R1+R2
R5←R3+R4
R6←R1+R5
R3←R1+R4
R5←R4+R5
```

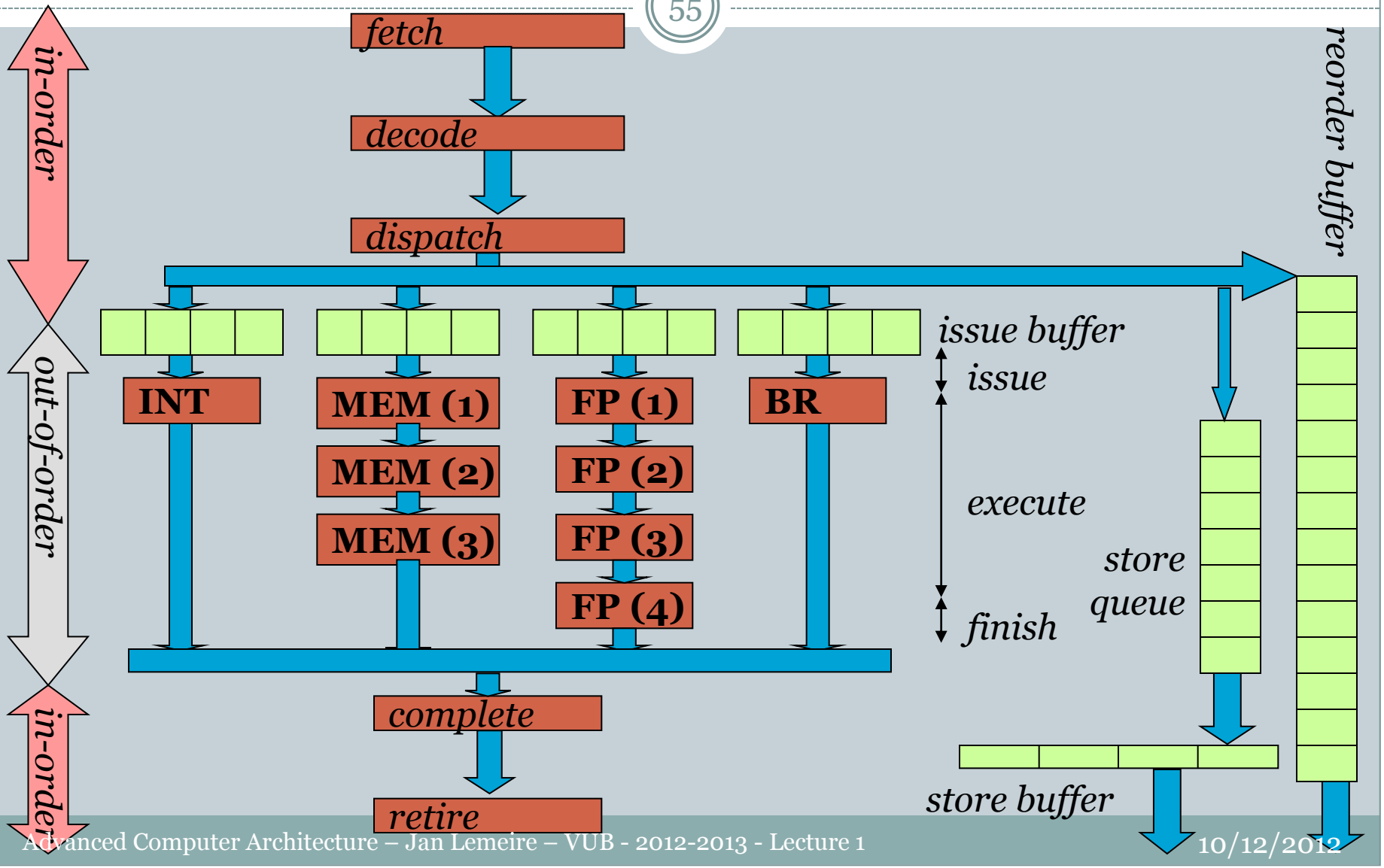


- Approximates upper limit on ILP

Solution 3: Out-of-order Execution (3)



Solution 3: Out-of-order Execution (4)



Solution 3: Out-of-order Execution (5)

- Issue buffer = reservation station
  - Instructions are inserted in issue buffer and reorder buffer in program order
  - Instructions are executed on FUs and might leave them in another order
- Reorder buffer = completion buffer
  - Out-of-order finish because of out-of-order issue en non-uniform execution latencies
  - In-order retirement = writing back the results in the registers
  - Enables precise exceptions
- Store queue en store buffer (see later)

Solution 3: Out-of-order Execution (6)

- "Sequentiality is an illusion"
  - Programmer sees instructions executing in program order, i.e., sequentially
  - In hardware many things happen in parallel
    - Temporally by pipelining
    - Spatially by exploiting ILP
      - In out-of-order processors this even involves instructions that are not executed in program order
  - In-order retirement guarantees sequential appearance
  - Parallelism at the instruction level: *instruction-level parallelism (ILP)*

Limitations to Scalar Pipelines

- Instruction type unification is a problem
  - e.g.: floating-point addition vs. integer addition
  - solved with pipeline diversification
- Fundamental limit: IPC ≤ 1
  - solved with superscalar pipeline
- In-order execution: IPC < 1
  - stalls caused by dependencies
  - solved with out-of-order execution

Summary

- Improve performance with
  - Pipelining: temporal parallelism
  - Superscalar design: spatial parallelism
- Optimal pipeline depth
  - Limitations on deeper pipelines
    - Technology aspects, data dependencies, control dependencies in code
- Superscalar processor
  - Execute more than one operation per cycle
  - In-order versus out-of-order

Important Streams

- Instruction stream
  - Front-end of the pipeline
  - Fetch as many as possible instructions into the pipeline per cycle
  - Lecture 3
- Data stream through registers
  - Detect and deal with dependencies between instructions
  - OoO execution of instructions
  - Lecture 3
- Data stream through memory
  - Get as much as possible data in and out of memory
  - Lecture 2

Acknowledgement

- Thanks for (parts of) slides
  - Bjorn De Sutter
  - Lieven Eeckhout
  - Mikko H. Lipasti
  - James C. Hoe
  - John P. Shen