Lecture #1: Superscalar Execution

Parallel Computer Systems

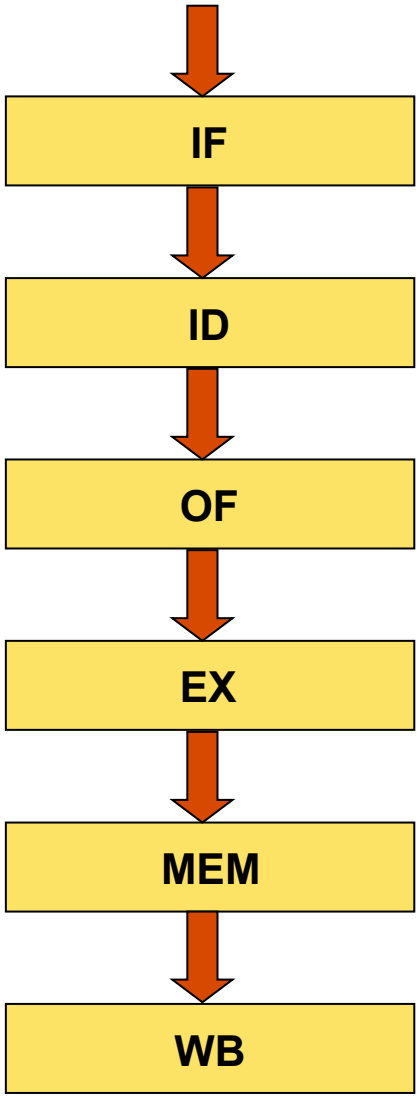
Lieven Eeckhout

Academiejaar 2014-2015
Ghent University

Overview

- Hazards in instruction pipelines
  - due to data dependences
  - due to control dependences
- From scalar, to superscalar, to out-of-order
  - Diversified pipeline
  - Superscalar pipeline
  - Out-of-order execution
- The out-of-order pipeline

Instruction Pipeline



Instruction Fetch

Instruction Decode

Operand Fetch

Execute

Memory Read/Write

Write Back

Why don’t we have perfect pipeline behavior?

Program dependences

- Three types
  - Data dependence through memory
  - Data dependence through registers
  - Control dependence

- A hazard is a result of not respecting a program dependence
- A hazard should therefore NOT happen!

Data Dependences

- Real 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
```

Hazards due to memory dependences?

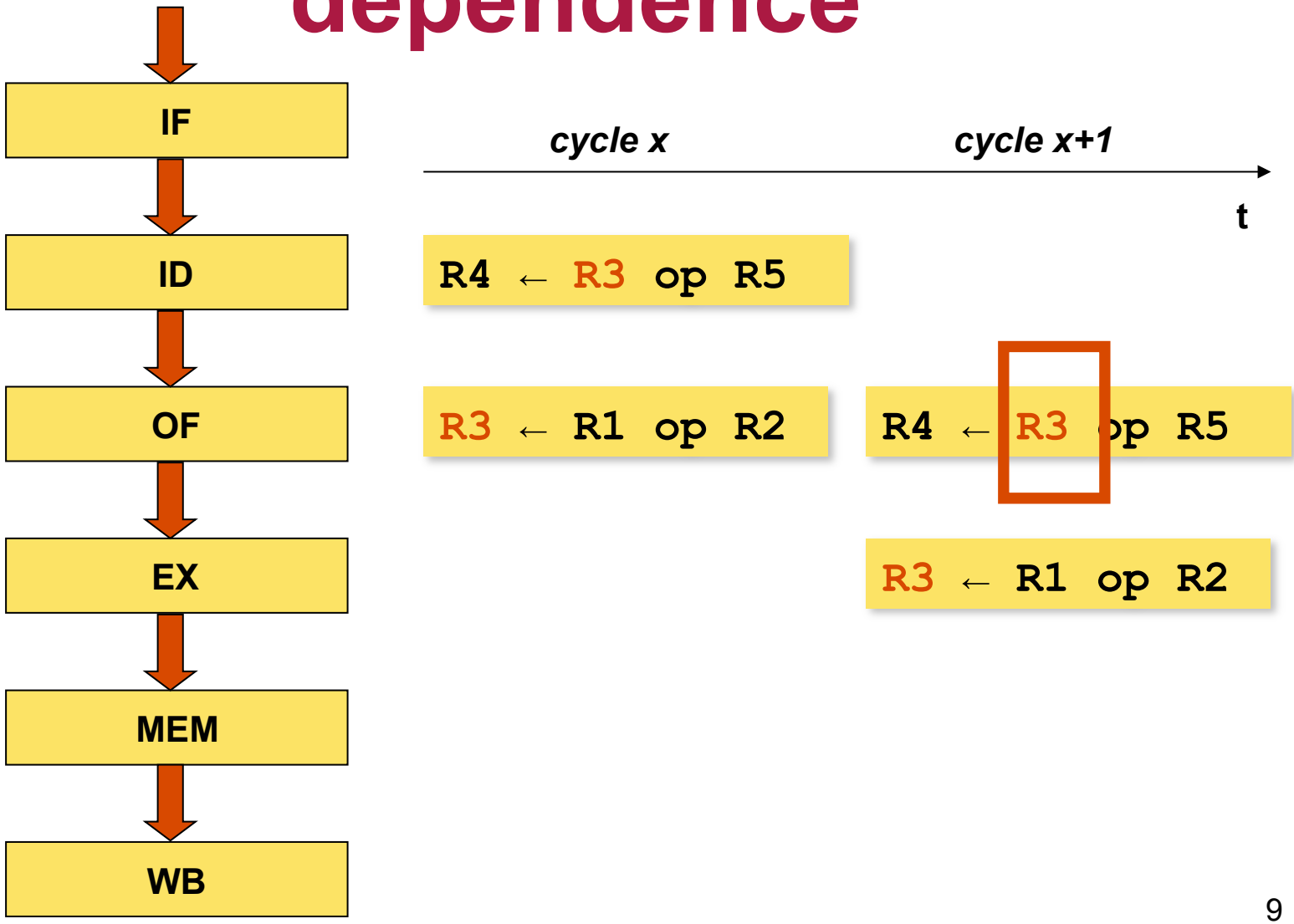
- Only the MEM trap reads/writes to/from memory
- All accessess to memory execute sequentially and in program order

- Answer: NO

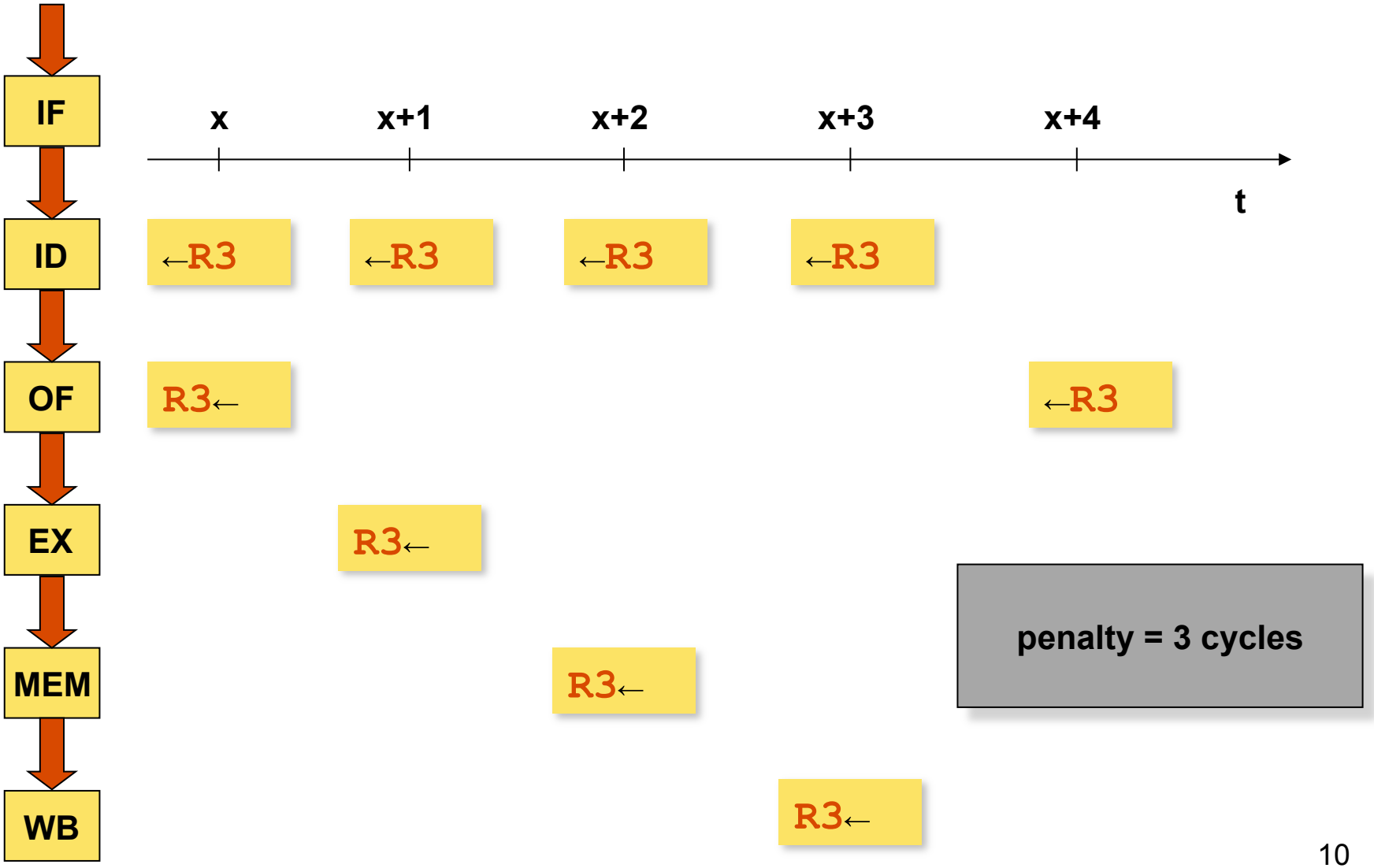
Hazards due to register data dependences?

- Hazard due to WAW register dependence?
  - NO: only WB trap writes, and does so sequentially and in program order
- Hazard due to WAR register dependence?
  - NO: reading is done in OF trap; writing is done in WB trap
- Hazard due to RAW register dependence?
  - YES: may happen if an instruction reads an old value from the register file

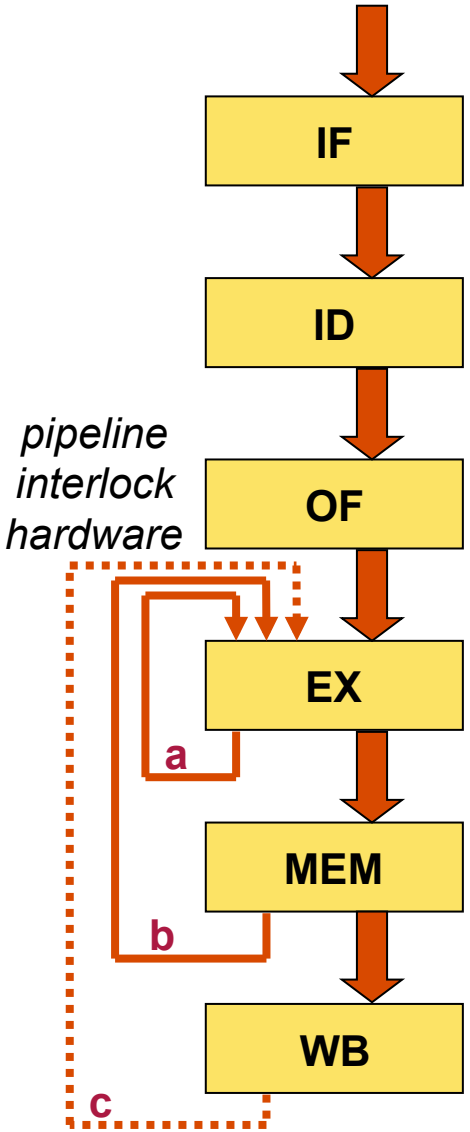
Hazard due to RAW register dependence



Naive solution: Pipeline Stall



Better solution: Forwarding



pipeline interlock hardware

←R3

←R3 ←R3

←R3 ←R3 ←R3

R3← ←R3 ←R3 ←R3

via a *via b* *via c*

R3← ←R3 ←R3

R3← ←R3

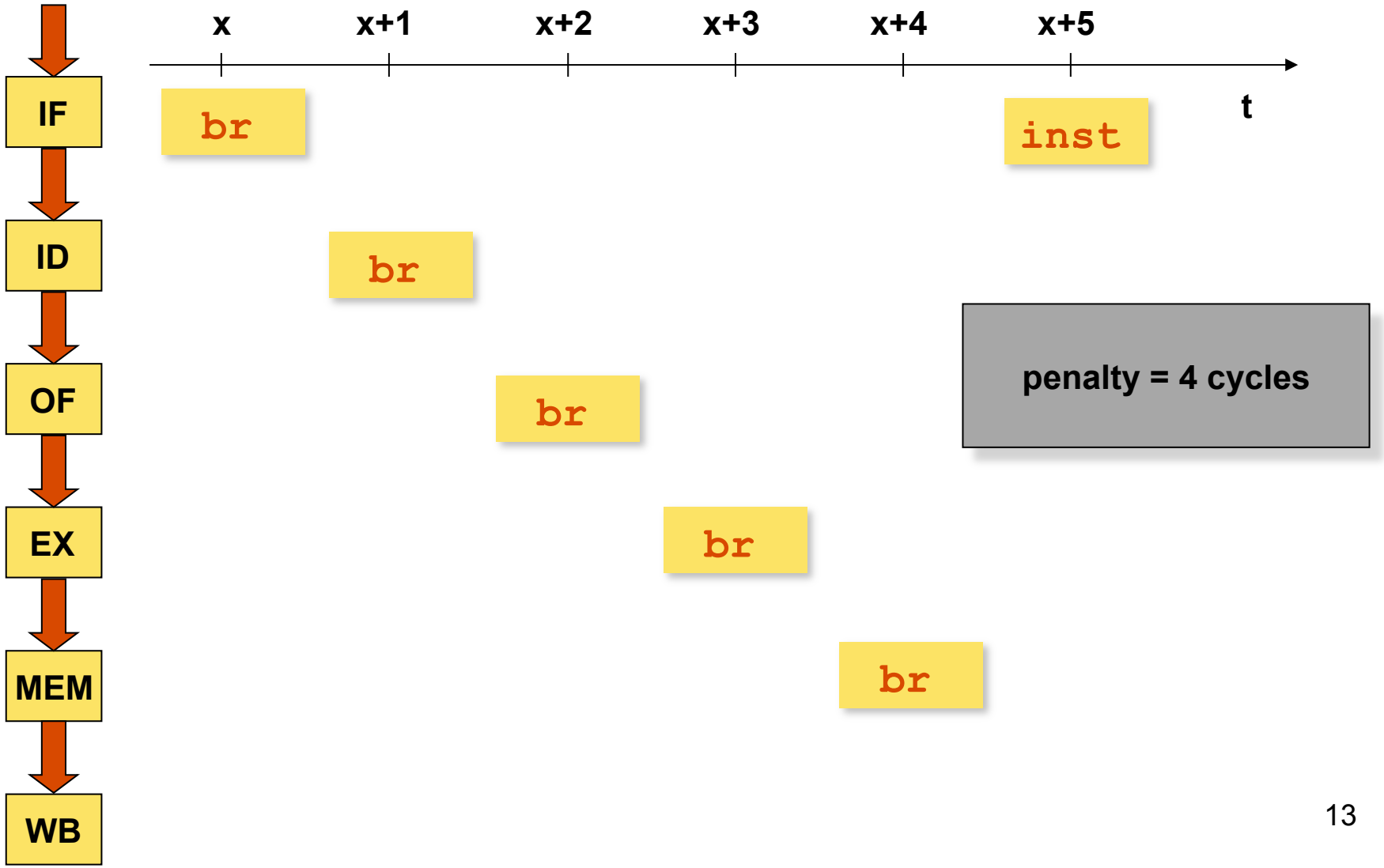
forwarding path c could be implemented through the register file

data-dependent insns execute in subsequent cycles

Overview

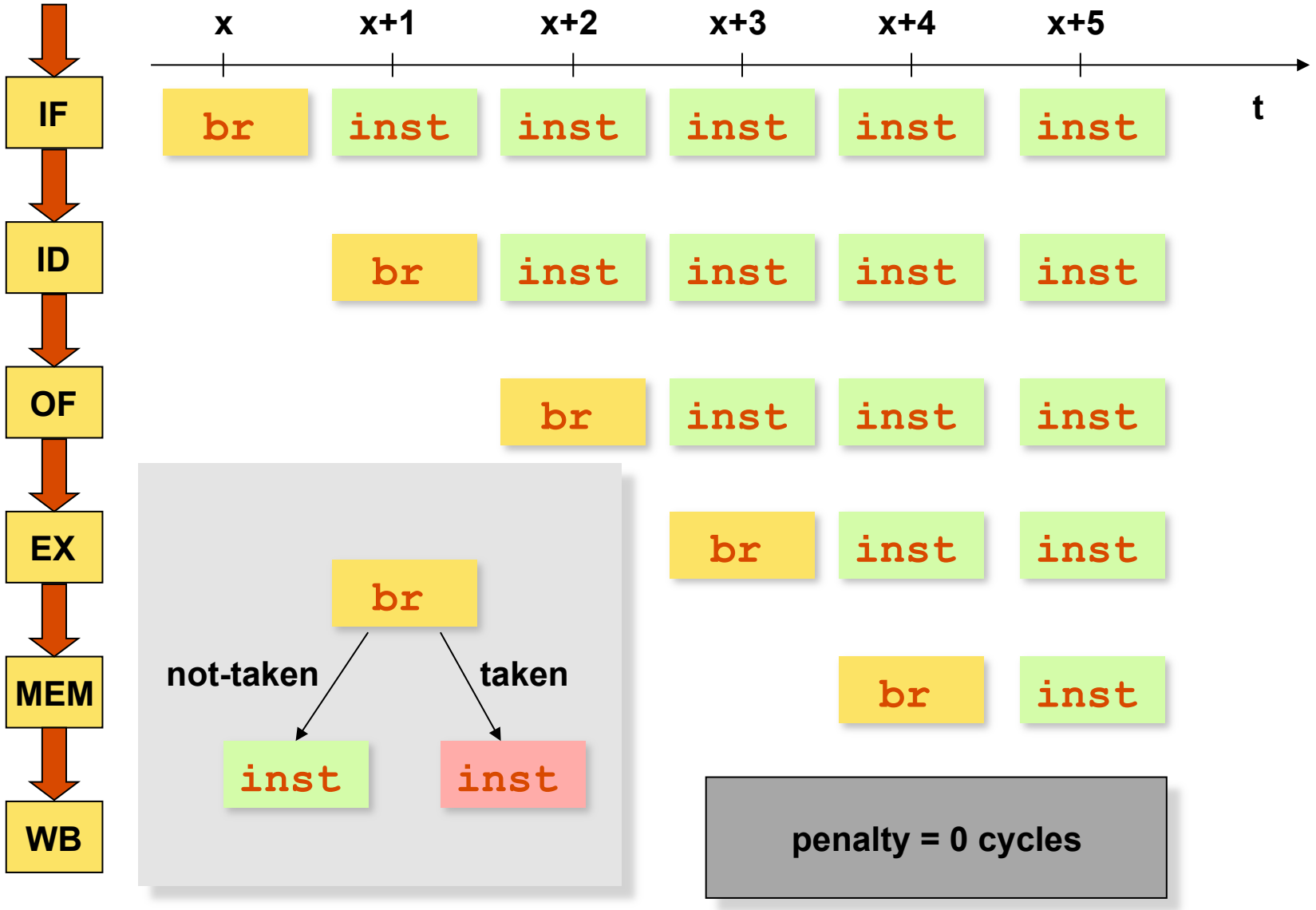
- Hazards in instruction pipelines
  - due to data dependences
  - due to control dependences
- From scalar, to superscalar, to out-of-order
  - Diversified pipeline
  - Superscalar pipeline
  - Out-of-order execution
- The out-of-order pipeline

Hazard due to control dependence

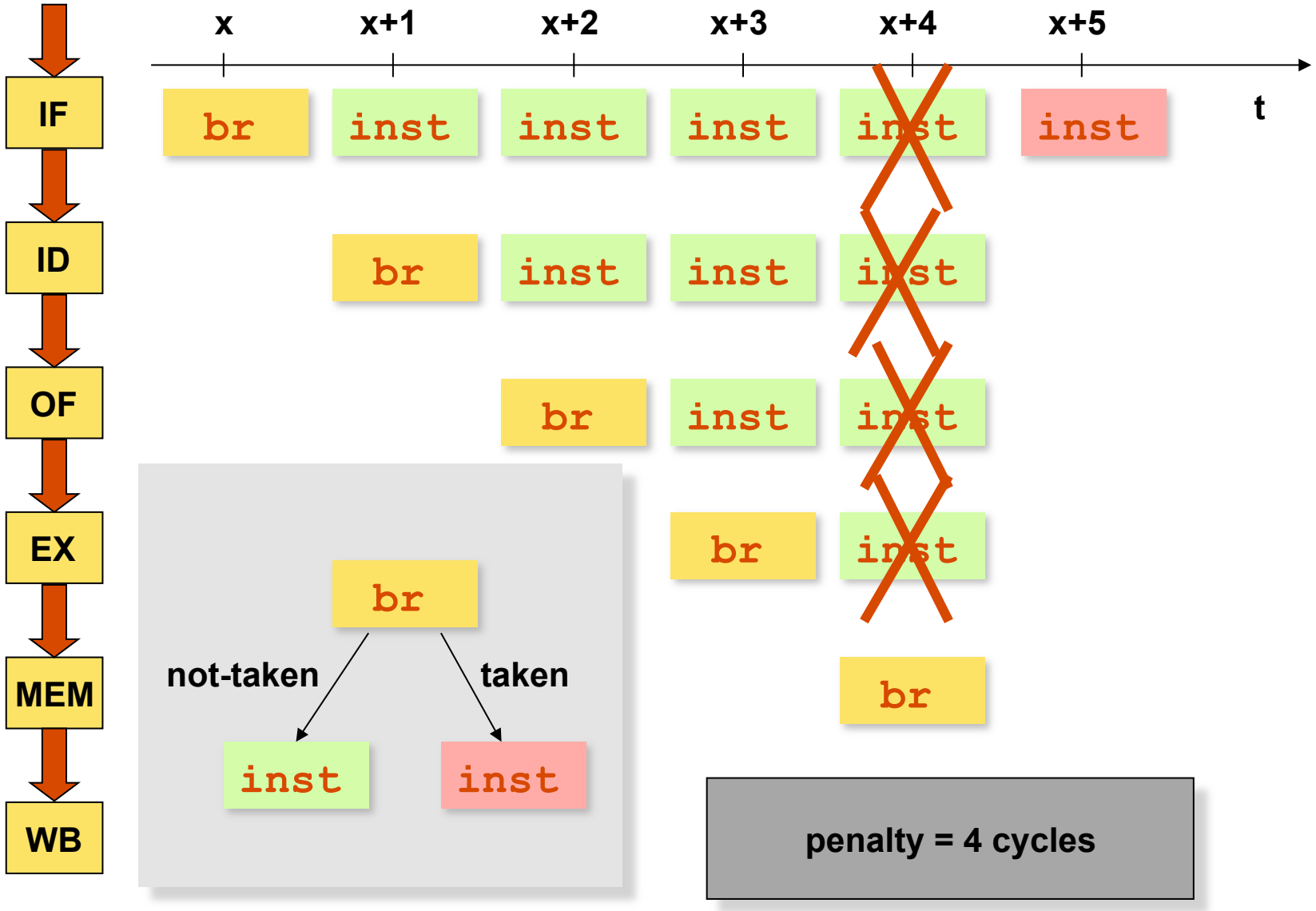


Solution: Branch Prediction & Speculative Execution

Upon a correct prediction



Upon a misprediction



Cost per mispredicted branch

- Cost per mispredicted branch is proportional to pipeline depth (no. pipeline stages)

- Early pipelines had 4-6 stages
- Modern pipelines have 12-15 stages
  - Example: Intel Pentium 4 had 20 stages (Extreme Edition had 31 stages)
  - Reason: Higher clock frequenties

Iron Law of Performance

- Execution time T (seconds) =

  N (no. insns to execute) x

  CPI (avg no. cycles per insn) x

  (1/f) (clock frequency f)

- Performance = 1 / Execution time

Optimal pipeline depth

- Deeper pipelines:
  - higher clock frequency f
  - higher cost due to mispredictions → higher CPI
- Hence: there is an optimum

- Also, dynamic power consumption increases with clock frequency! ($P \sim f^3$)
  - see later

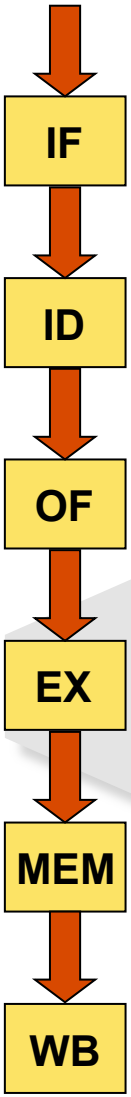
Overview

- Hazards in instruction pipelines
  - due to data dependences
  - due to control dependences
- From scalar, to superscalar, to out-of-order
  - Diversified pipeline
  - Superscalar pipeline
  - Out-of-order execution
- The out-of-order pipeline

Limitations of a scalar pipeline

- Unifying instruction types is a problem
- Maximum IPC = 1
- In-order execution

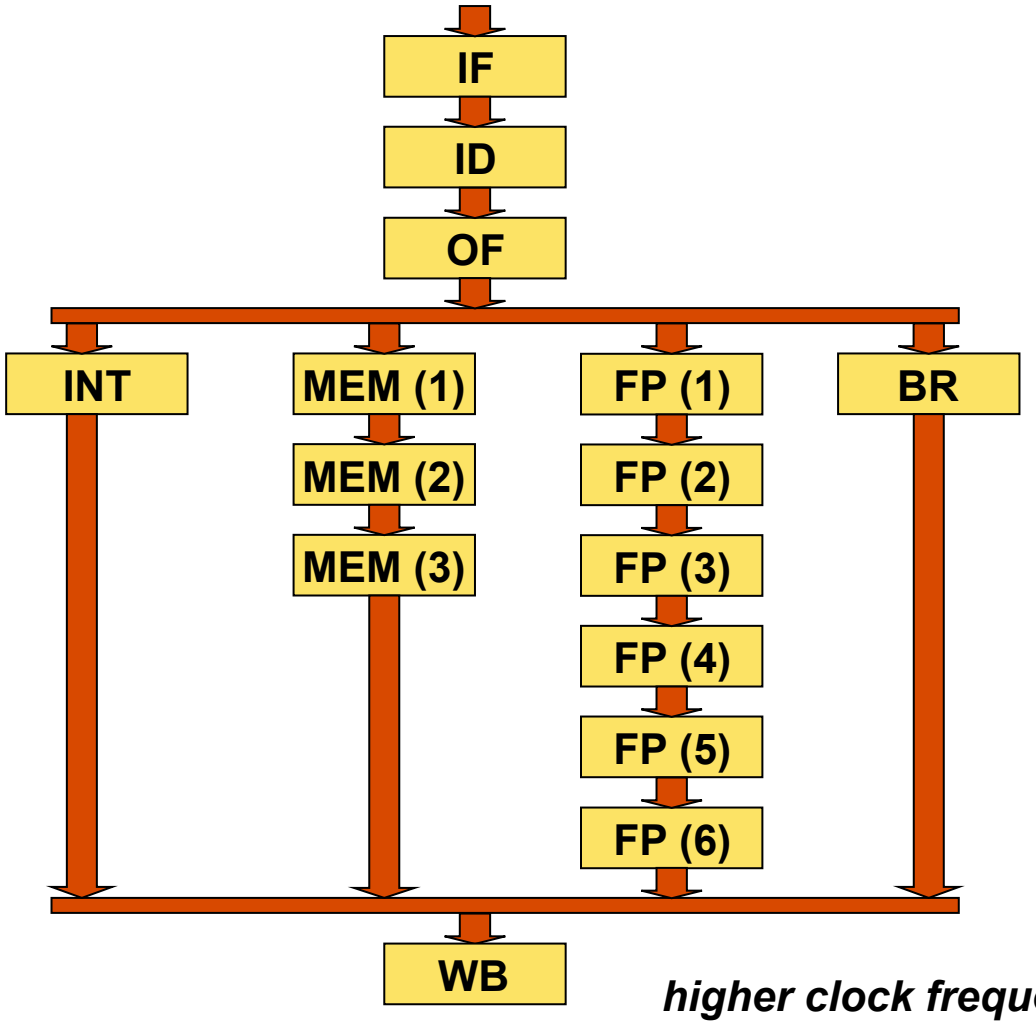
Problem #1: Unification



EX executes integer and floating-point operations in a single clock cycle.
Clock frequency is determined by the slowest pipeline stage.

Solution: diversification

Diversified pipeline

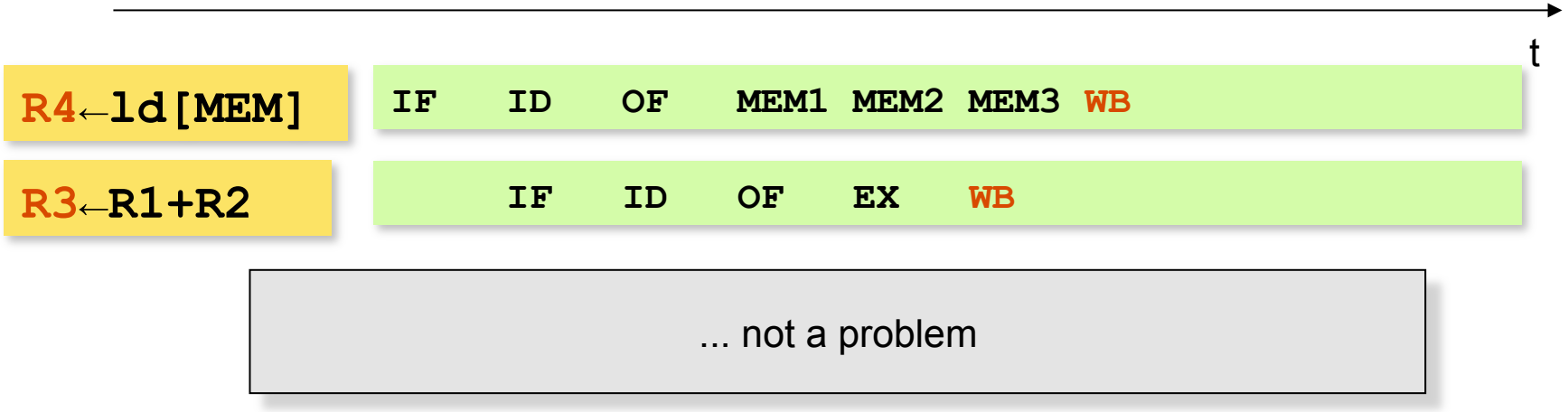


higher clock frequency than unified pipeline

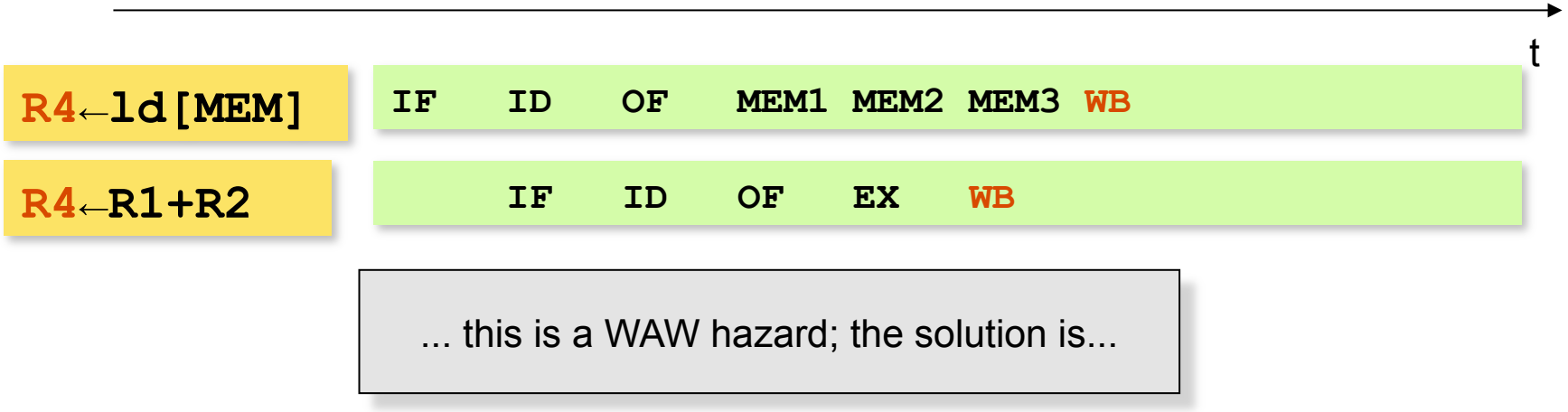
Three problems

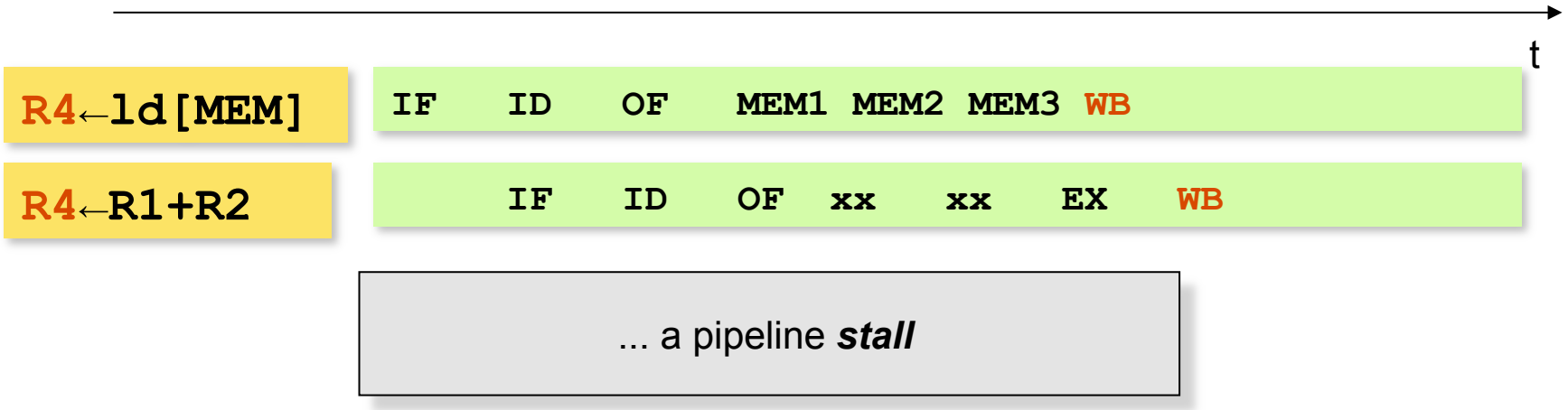
- Out-of-order completion
  - Writing back results out of program order
- Multiple write operations to the RF in the same clock cycles
- Exceptions

Out-of-order completion

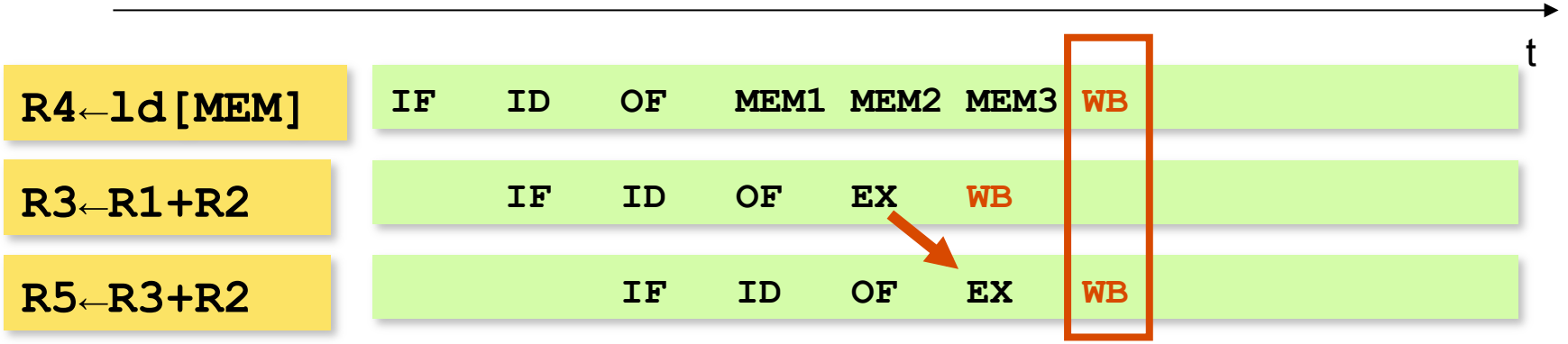


BUT ...

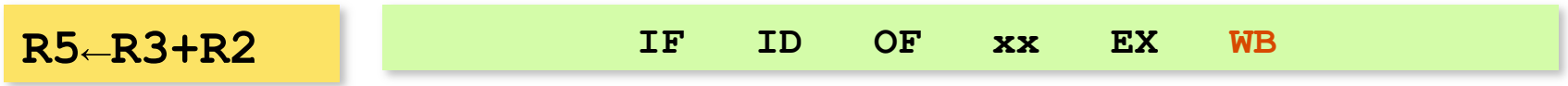

t
R4←ld[MEM]
IF ID OF MEM1 MEM2 MEM3 WB
R4←R1+R2
IF ID OF EX WB
... this is a WAW hazard; the solution is...


t
R4←ld[MEM]
IF ID OF MEM1 MEM2 MEM3 WB
R4←R1+R2
IF ID OF xx xx EX WB
... a pipeline *stall*

Multiple WBs per cyclus

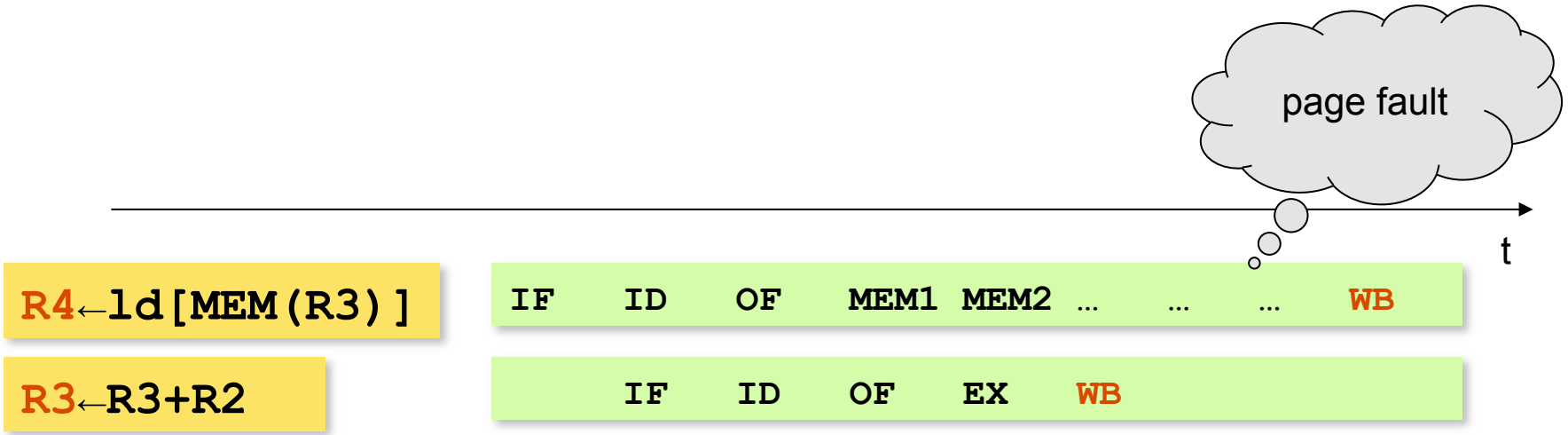


- Impossible in case there is only one write port to the RF
- Solution:
  - Add write port, or
  - Consider write port as a ***structural hazard***



What about exceptions?

- OoO completion doesn't enable precise exceptions
  - *Imprecise exceptions*



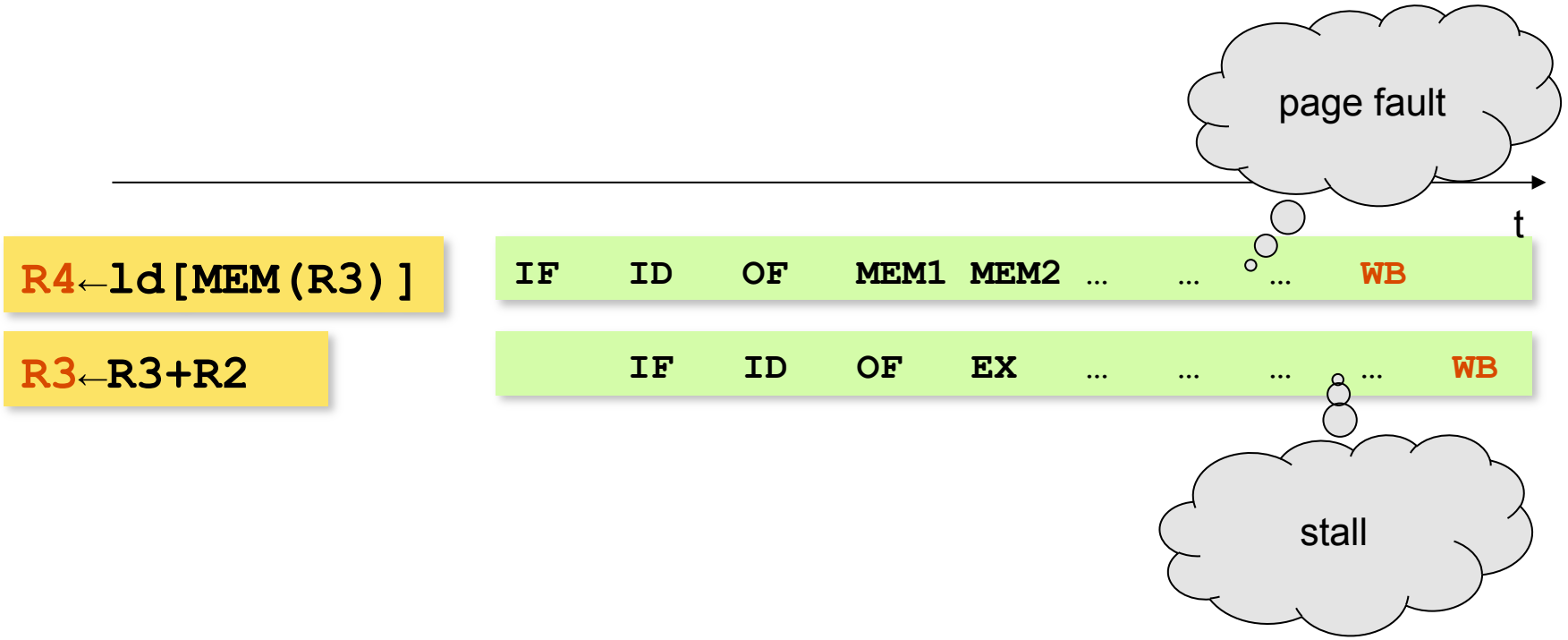
Interrupts vs. exceptions

- Interrupts
  - Typically due to external factors
  - Asynchronous to program execution
  - Processing:
    - Stop fetching new instructions
    - Drain the pipeline (execute instructions in the pipeline)
    - Store architecture state (registers, PC, etc.)
    - Handle the interrupt
    - Restore program's architecture state, and resume program execution
  - OoO completion is not a problem

Interrupts vs. exceptions

- Exceptions/faults
  - Synchronous: as a result of program execution
  - e.g., division by zero, page fault, overflow, etc.
  - *Precise exception*
    - Store the architecture state from just before the instruction that caused the exception
    - Handle the exception
    - Restore architecture state and resume execution from the instruction that caused the exception

Guaranteeing precise exceptions



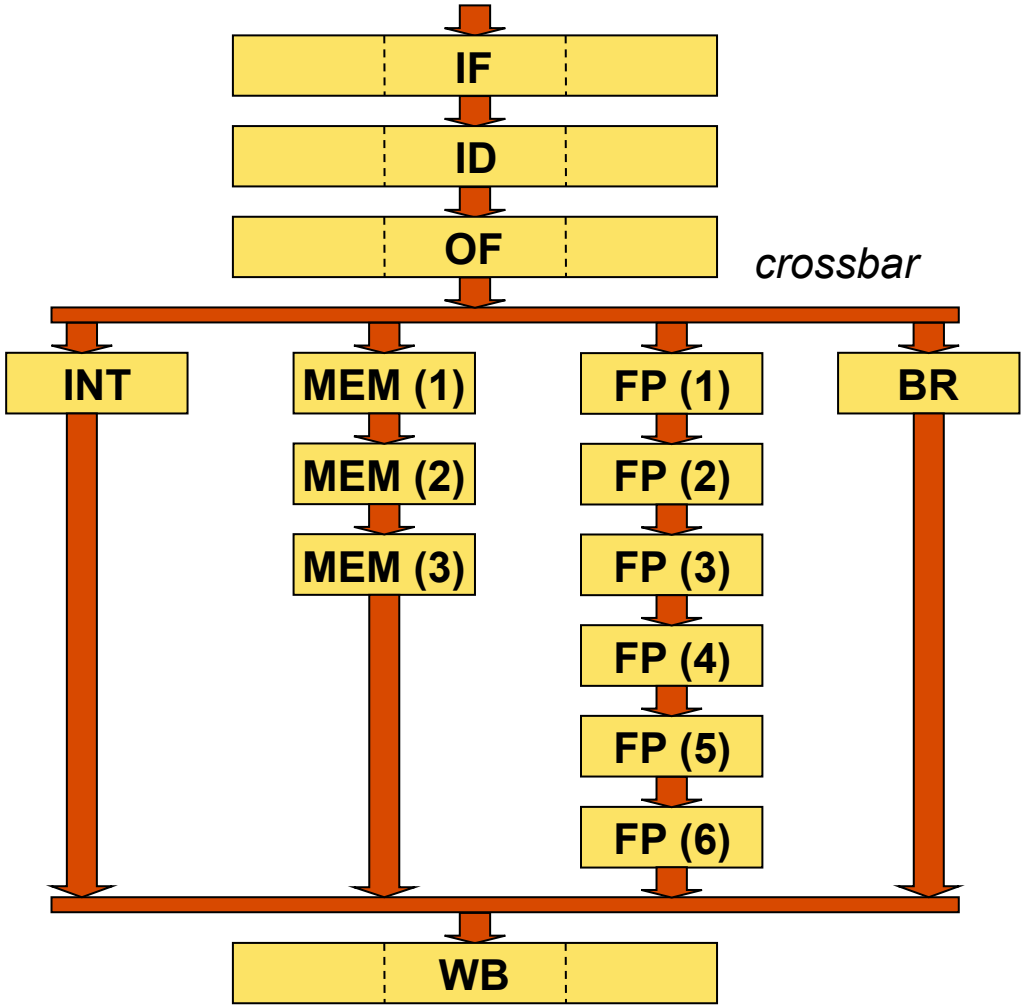
Limitations of a scalar pipeline

- Unifying instruction types is a problem

- Maximum IPC = 1

- In-order execution

Superscalar pipeline



Superscalar pipeline

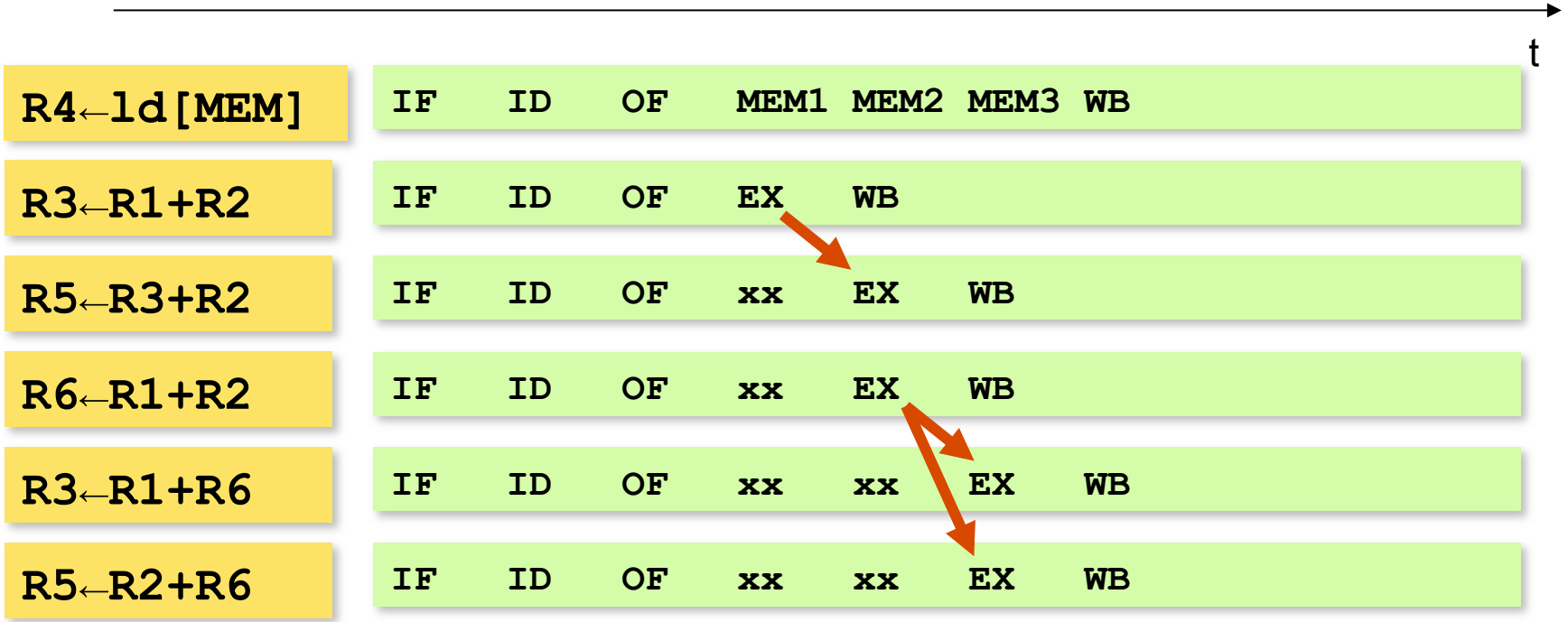
- Parallelism in time
  - Pipelining
  - Relatively cheap
- Parallelism in space
  - Superscalar execution
  - Relatively expensive (more HW is needed)

- Superscalar pipeline
  - Parallelism in time ***and*** space

Preventing hazards

- Is done *at issue time*, before sending an instruction to a FU

Superscalar execution

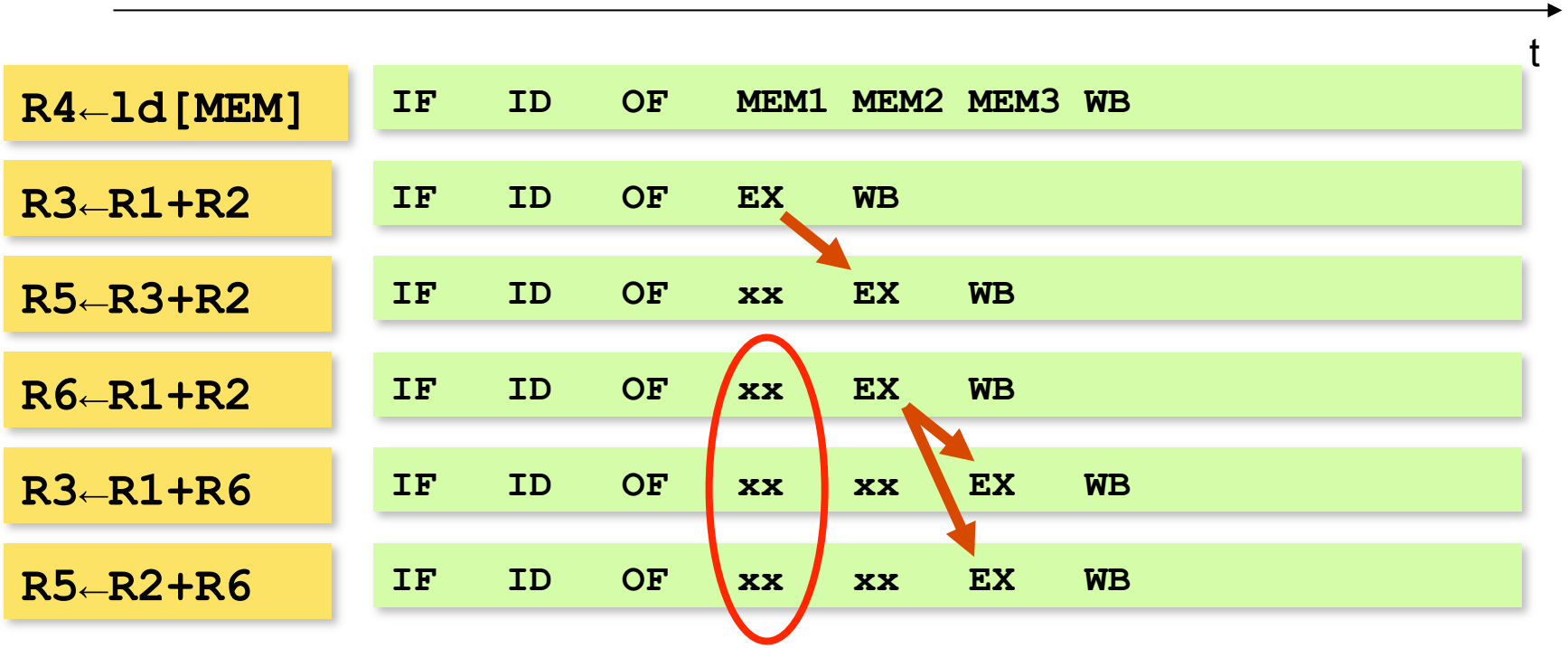


(assuming out-of-order completion)

Limitations of a scalar pipeline

- Unifying instruction types is a problem
- Maximum IPC = 1
- In-order execution

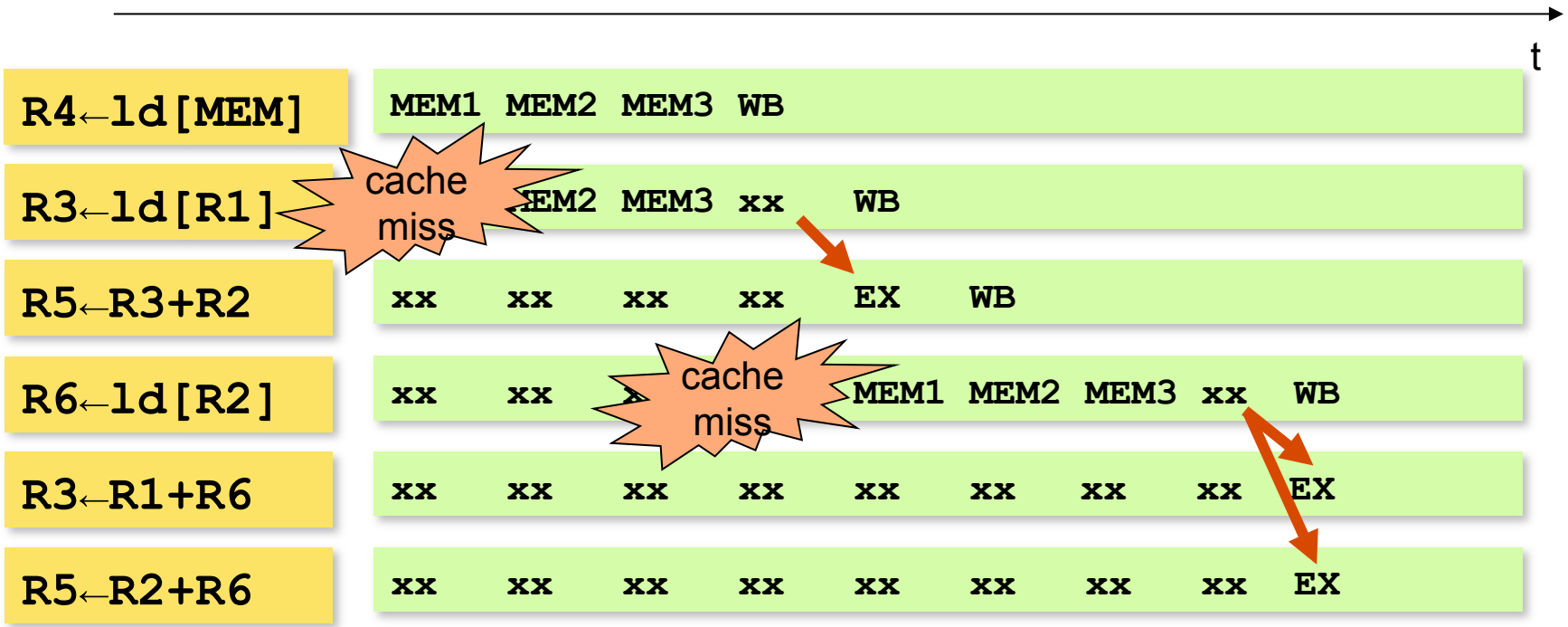
In-order issue



Insns 4 thru 6 stall although they are independent of prior insns in the dynamic instruction stream.
This is a ***fundamental limitation*** of in-order issue.

Cost increases with increased instruction latencies

unnecessary serialization of insns and miss events



Fundamental limitation

- An insn that stalls, stalls the upstream part of the pipeline
  - ***Stall-on-use***
  - **This includes insns that are independent of the stalling insn!**
  - **Hence, instruction latencies (incl. cache misses) are unnecessarily serialized**

Out-of-order execution

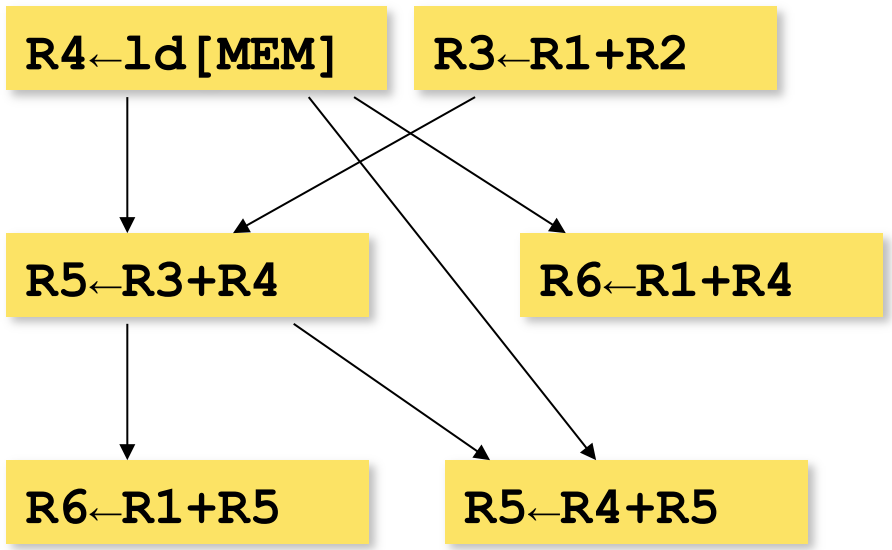
Key Idea:

- Remove all output and anti dependences from the dynamic instruction stream through register renaming
- Only real data dependences remain
- Enables data flow execution of instructions
  - insns execute as soon as their inputs are available

Data flow limit

Only RAW dependences remain!

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



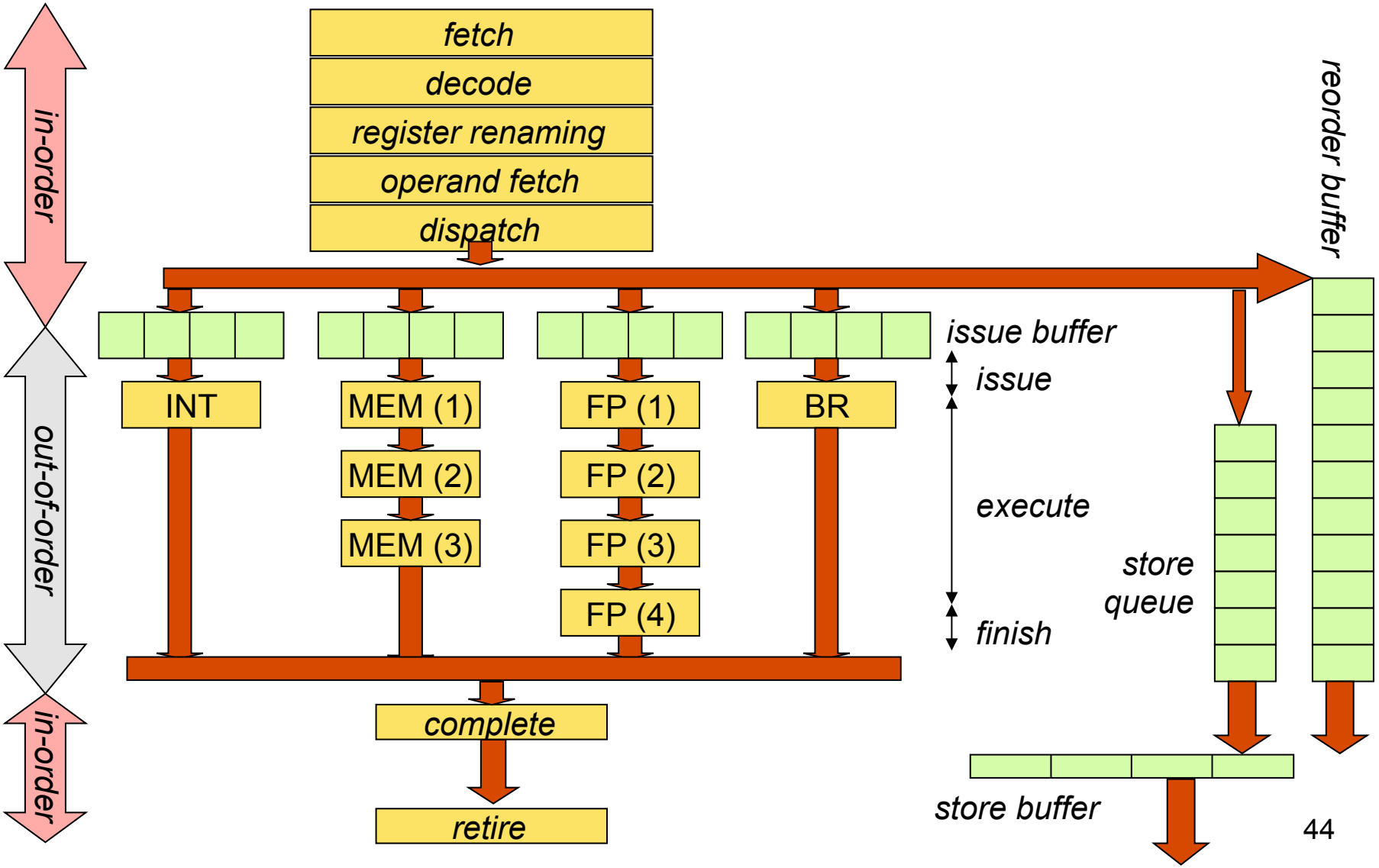
Data Flow Graph (DFG) shows the instructions as nodes, and real (RAW) data dependences as edges.

The height of the DFG shows the minimum number of cycles needed to execute the insns in data flow manner.

Overview

- Hazards in instruction pipelines
  - due to data dependences
  - due to control dependences
- From scalar, to superscalar, to out-of-order
  - Diversified pipeline
  - Superscalar pipeline
  - Out-of-order execution
- The out-of-order pipeline

Superscalar OoO processor



New pipeline stage: Register renaming

- Eliminates WAR and WAW register dependences
- Only RAW dependences remain
  - enables data flow / OOO execution

New pipeline stage: Dispatch

- Inserting instructies in reservation station and reorder buffer

New pipeline stages: Completion vs retirement

- Completion
  - Architecture state is updated
  - For software, the instruction appears to be fully executed
- Retirement
  - For stores: data is written into the memory hierarchy

- For non-store insns: completion = retirement
- For stores: completion ≠ retirement
  - completion: store moves from store queue to store buffer
  - retirement: store is written to memory hierarchy and leaves the store buffer

Two new structures

- Issue buffer = reservation station
  - Keeps track of yet to execute instructions
  - Instructions are inserted in program order
  - Instructions execute on FU and leave the issue buffer (possibly) out of program order
- Reorder buffer = completion buffer
  - Keeps track of ‘window’ of instructions currently under execution; maintains program order
  - Instructions are inserted in program order
  - Instructions leave the reorder buffer in program order
    - enables precise exceptions

“Sequentiality is an illusion”

- Software assumes that instructions execute in program order
- Hardware exploits instruction-level parallelism
  - parallelism in time: pipelining
  - parallelism in space: superscalar execution
  - even *out-of-order* execution
- In-order completion provides the illusion of sequential execution to software and enables precise exceptions