



NTNU

Det skapende universitet

TDT4255 Computer Design

Lecture 4: Out-of-order Execution

Magnus Jahre

Outline

- Appendix C.4 – C.5, C.7 - C.9
- Chapter 3 (except 3.2, 3.7, 3.11 and 3.12)

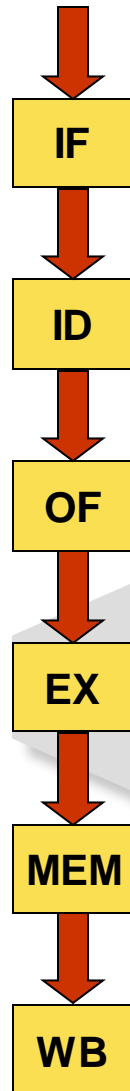
BEYOND SCALAR PIPELINES

*Slides in this section are by Lieven Eeckhout, Ghent University.
Reused with permission.*

Limitations of a scalar pipeline

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

Problem #1: Unification

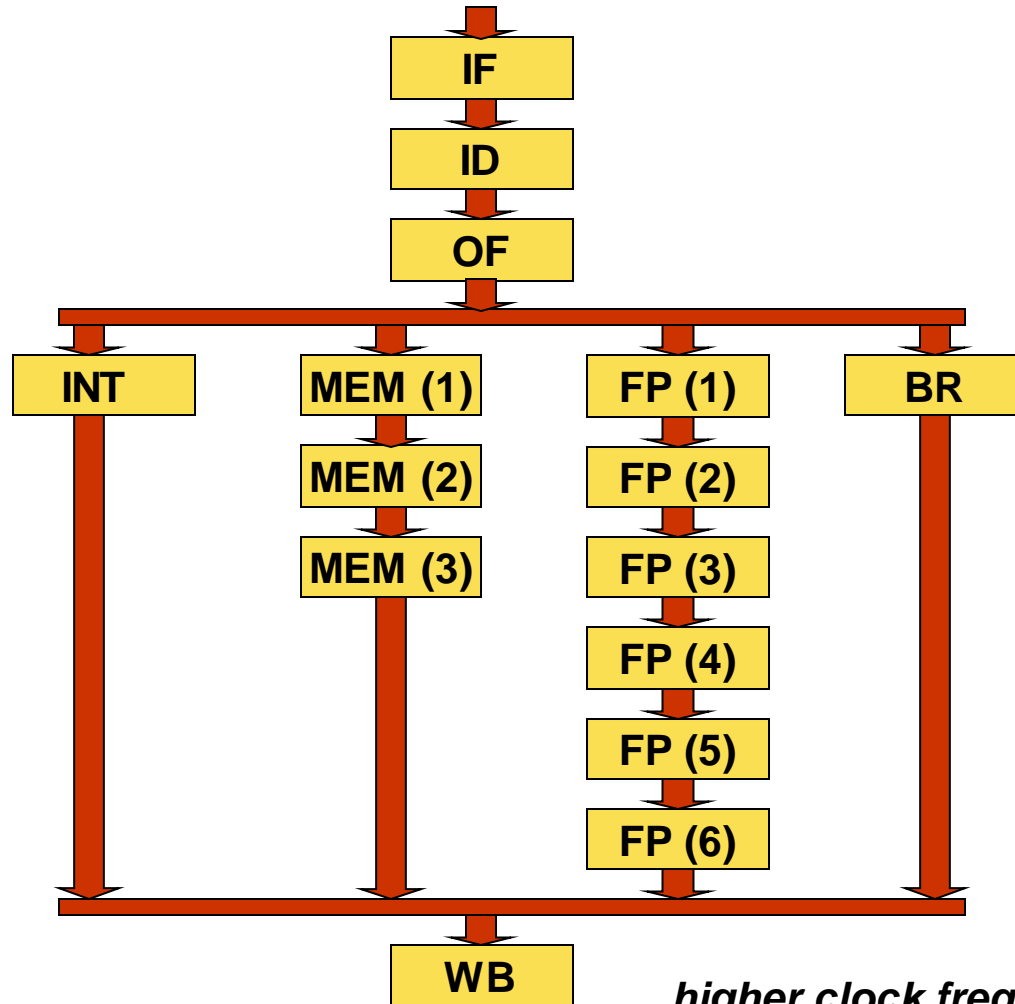


EX executes integer and floating-point operations in a single clock cycle.

Clock frequency is determined by the slowest pipeline stage.

Solution: diversification

Diversified pipeline

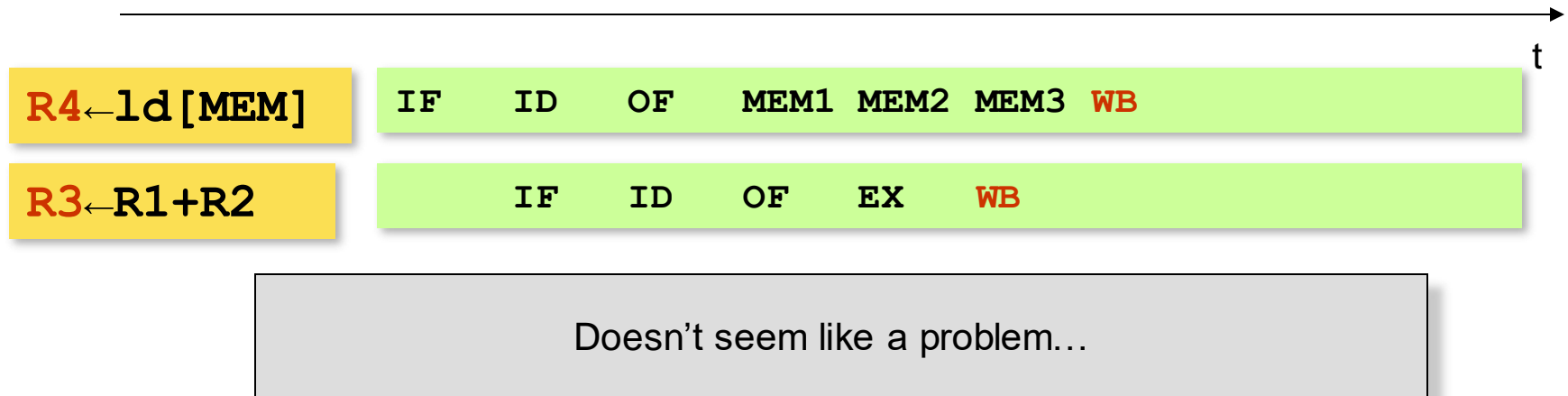


*higher clock frequency than
unified pipeline*

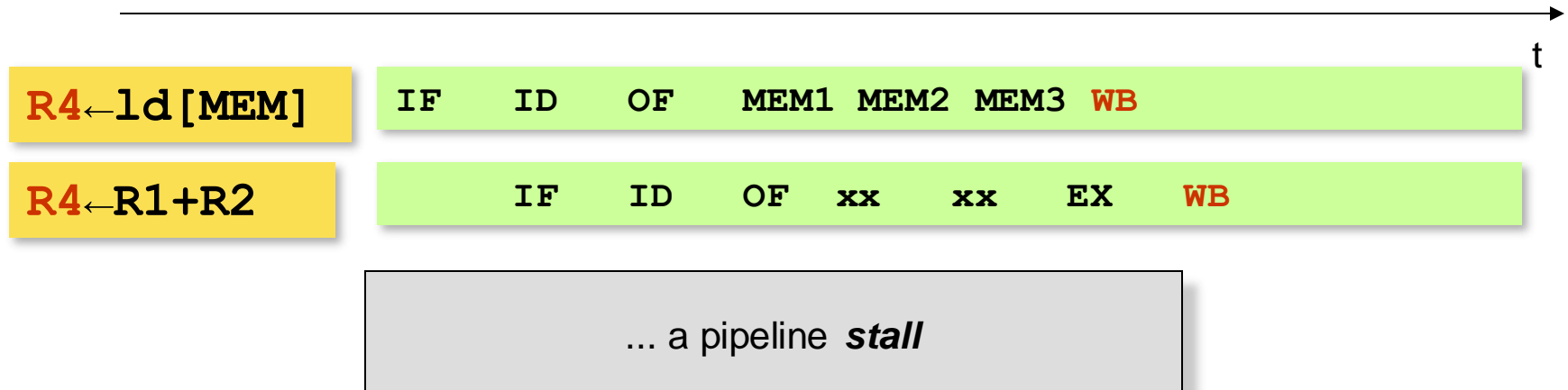
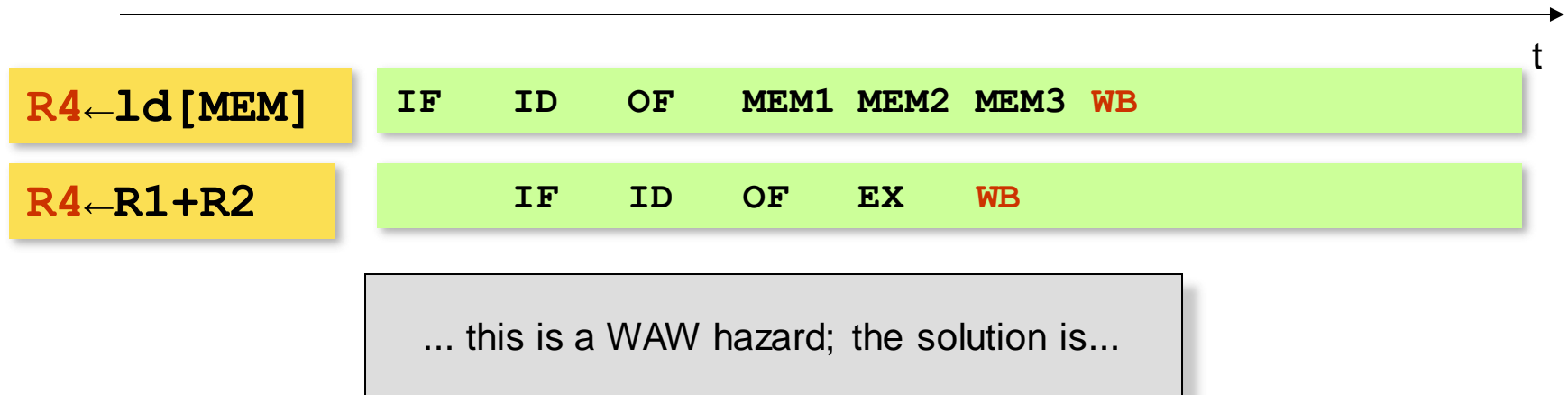
Three problems

- Out-of-order completion
 - Writing back results out of program order
- Multiple write operations to the register file (RF) in the same clock cycle
- Exceptions

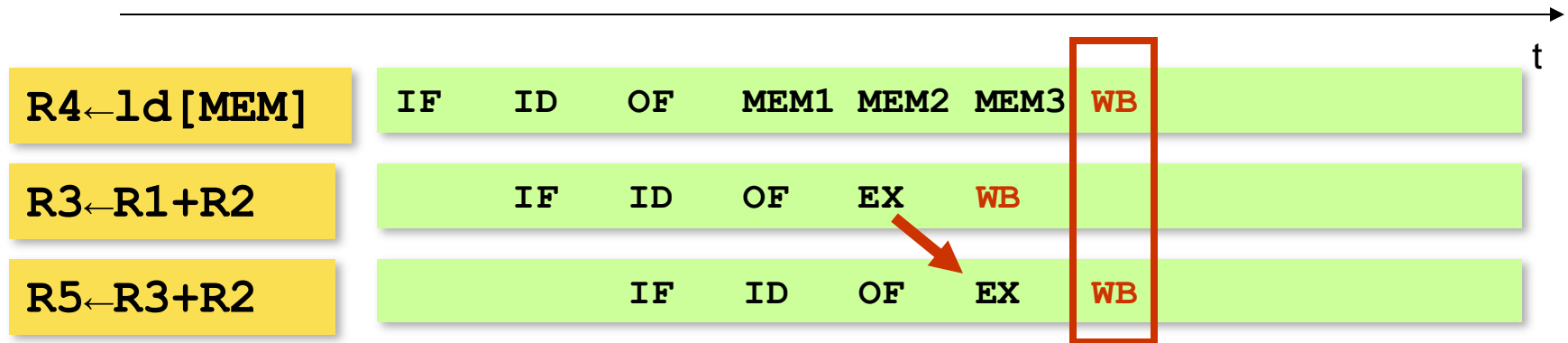
Out-of-order completion



BUT ...



Multiple WBs per cycle

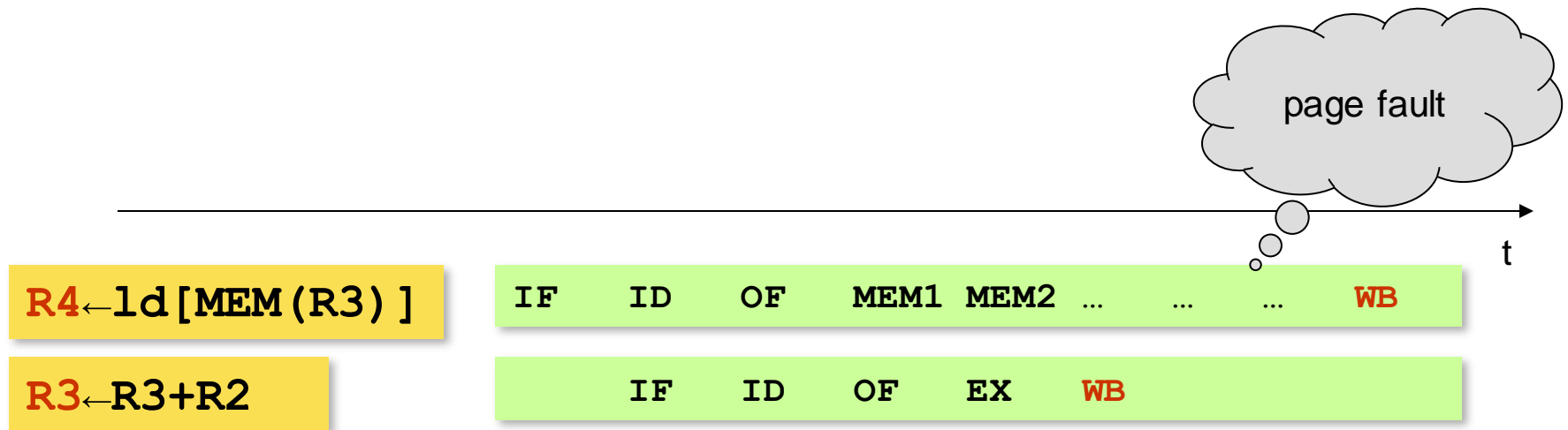


- Impossible in case there is only one write port to the RF
- Solution:
 - Add write port, or
 - Consider write port as a **structural hazard**



What about exceptions?

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



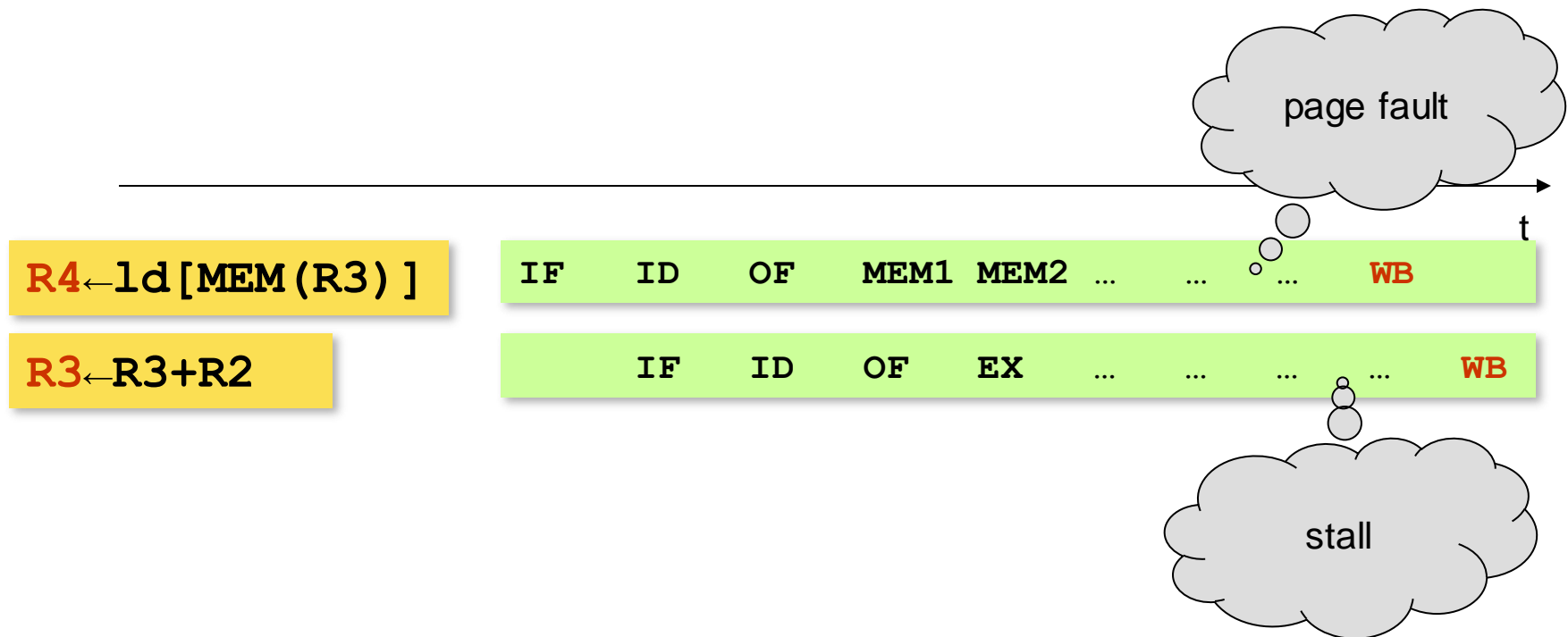
Interrupts vs. exceptions

- 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

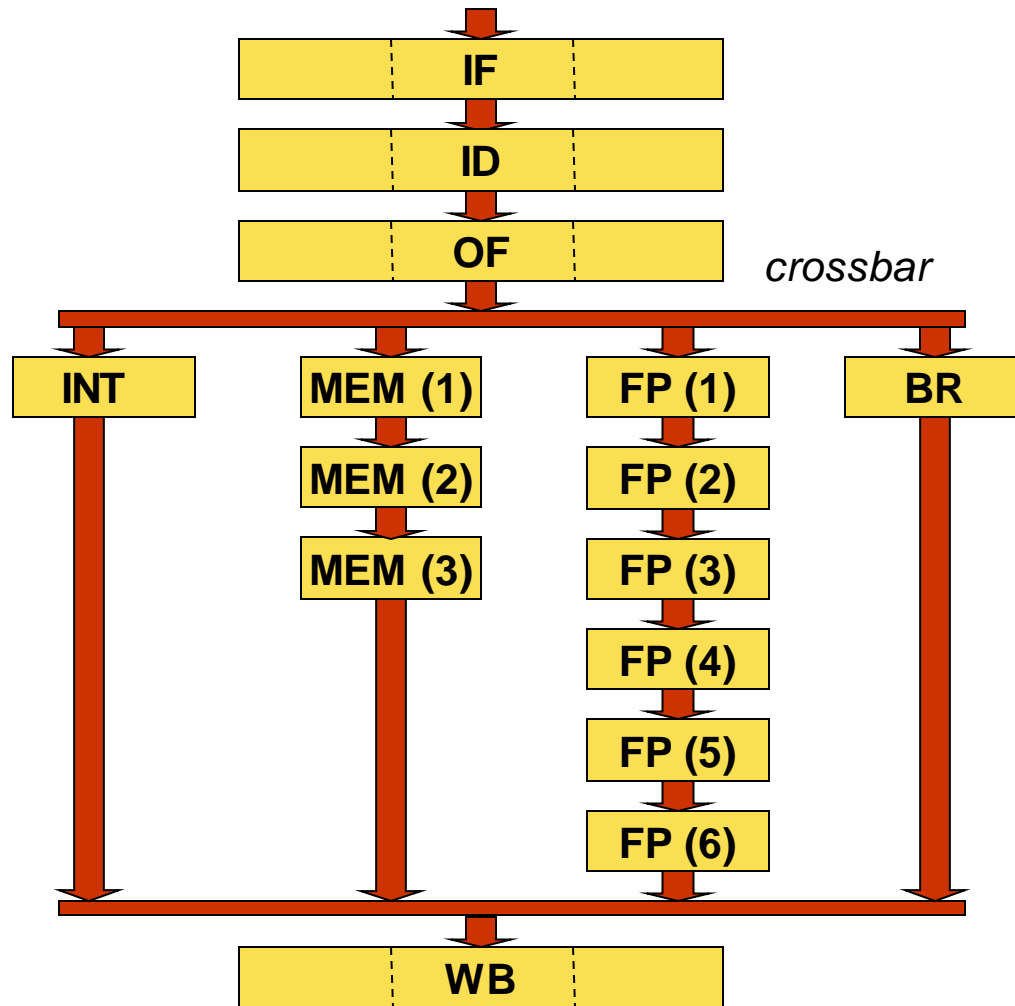
Guaranteeing precise exceptions



Limitations of a scalar pipeline

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

Superscalar pipeline



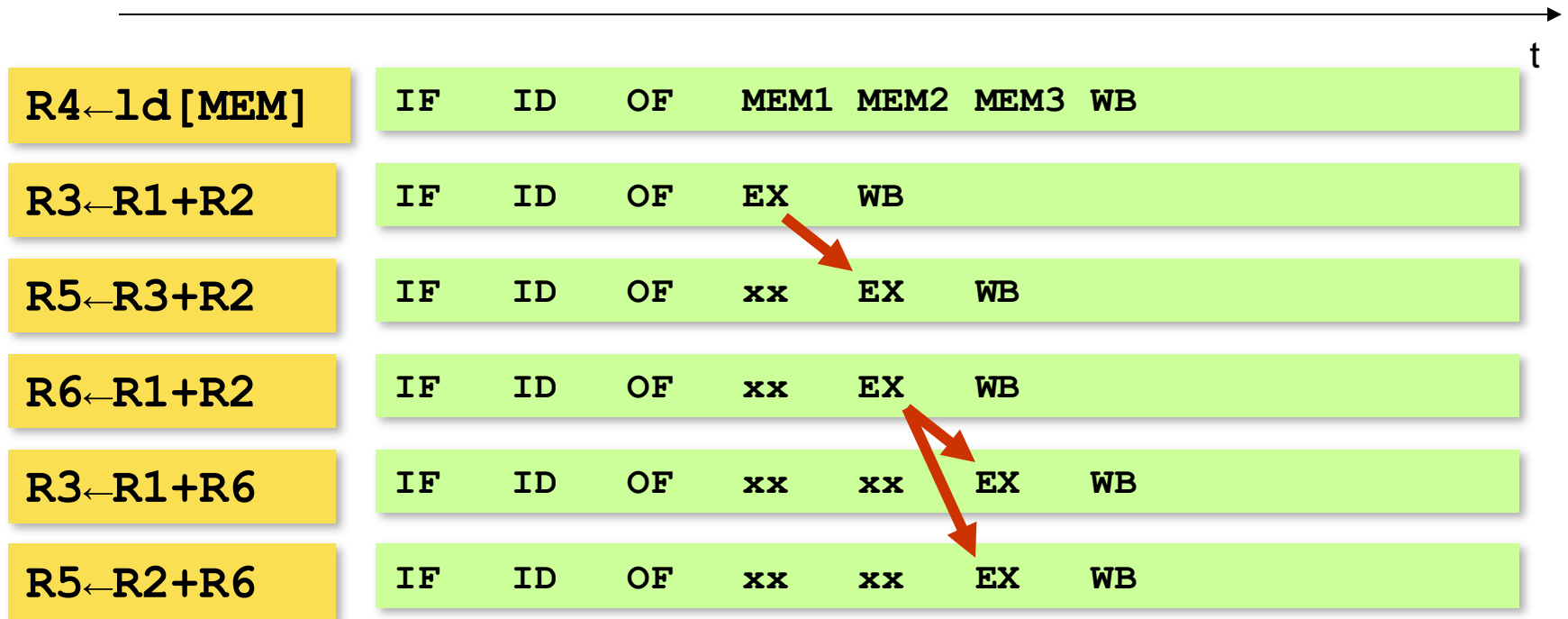
Superscalar pipeline

- 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 an FU

Superscalar execution

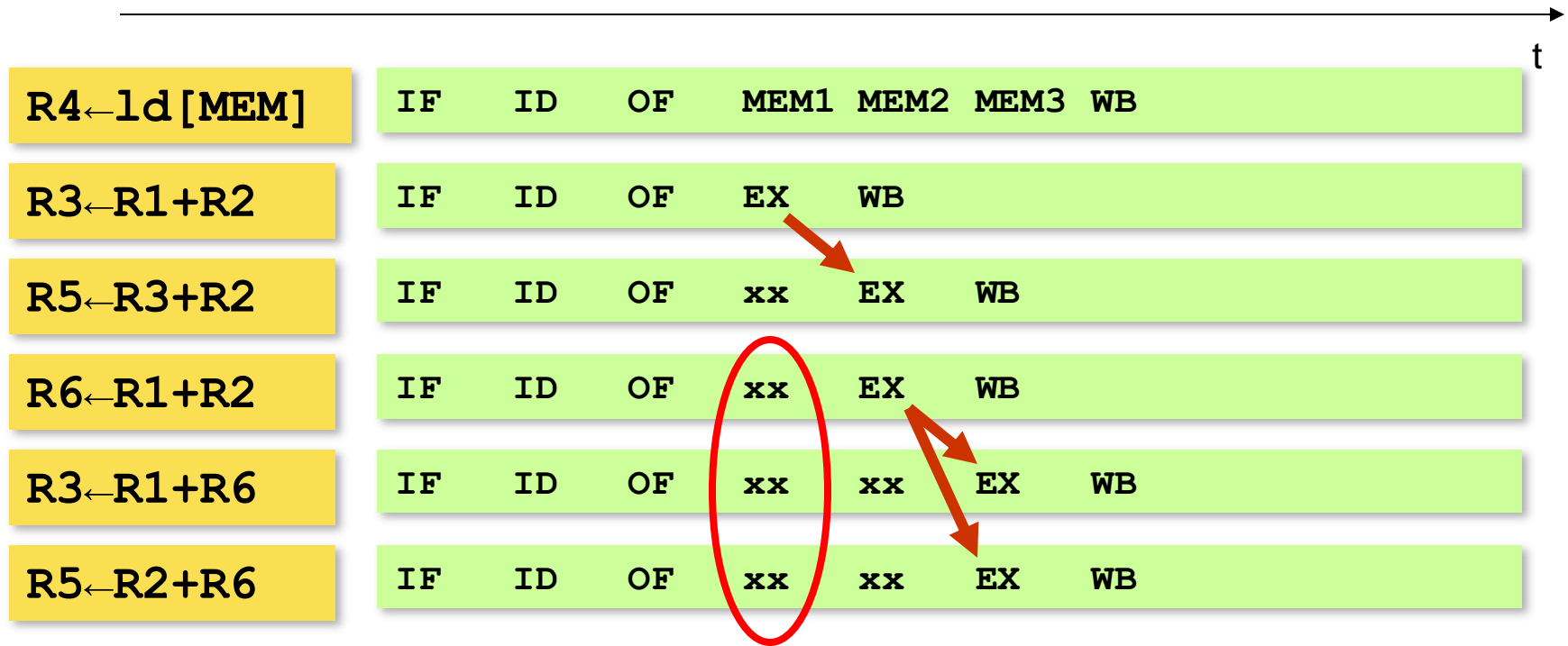


(assuming out-of-order completion)

Limitations of a scalar pipeline

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

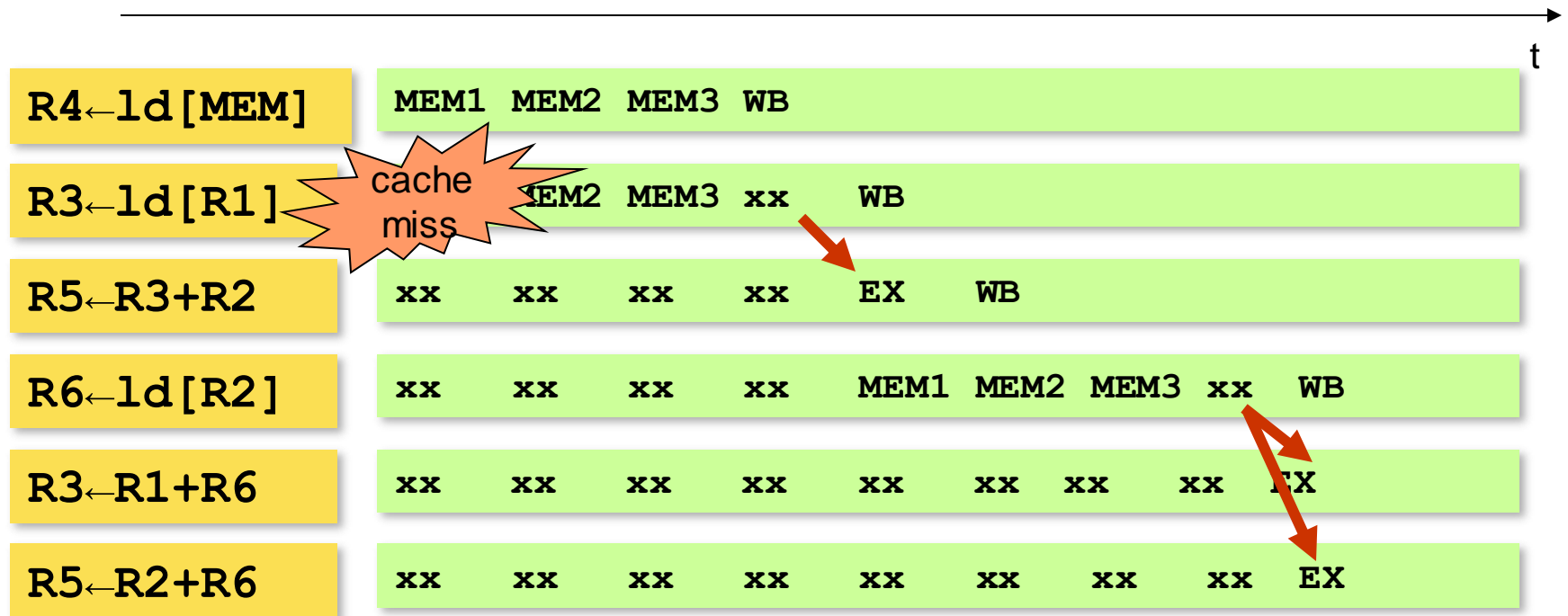
In-order issue



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 instructions
and miss events*



Fundamental limitation

- An instruction 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 OVERVIEW

*Slides in this section are by Lieven Eeckhout, Ghent University.
Reused with permission.*

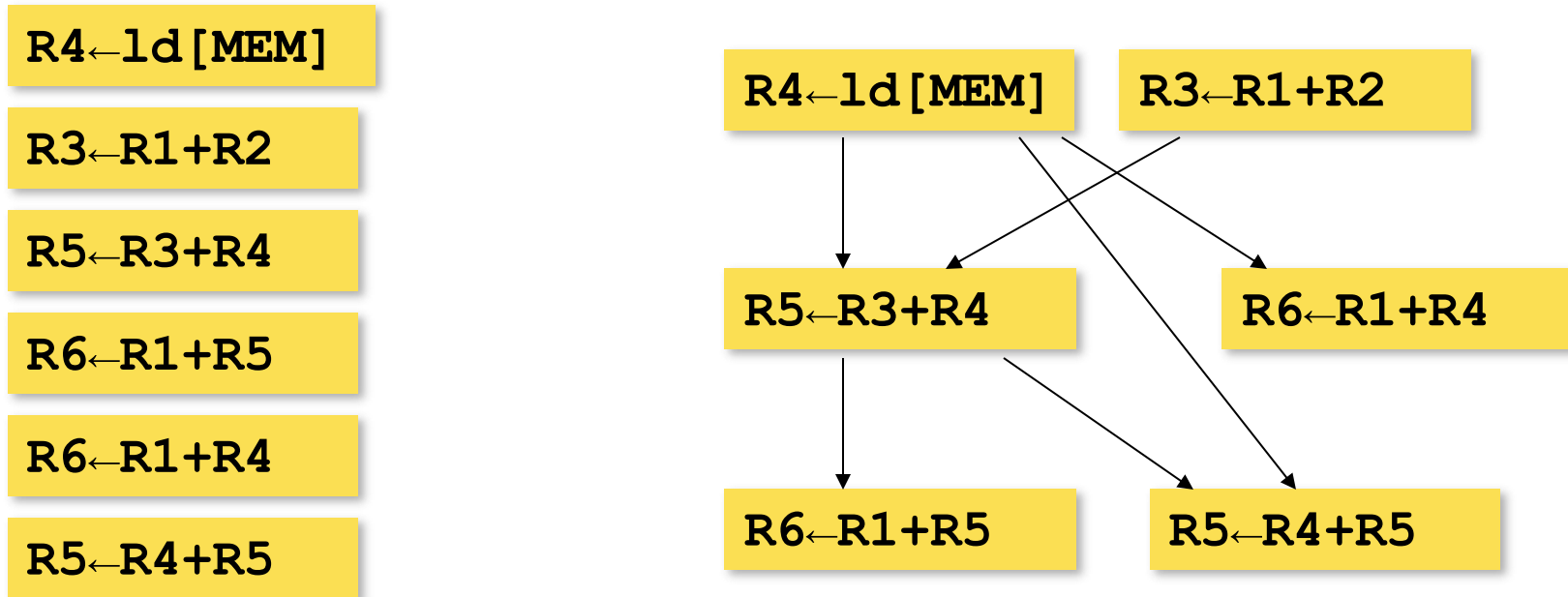
Out-of-order execution

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

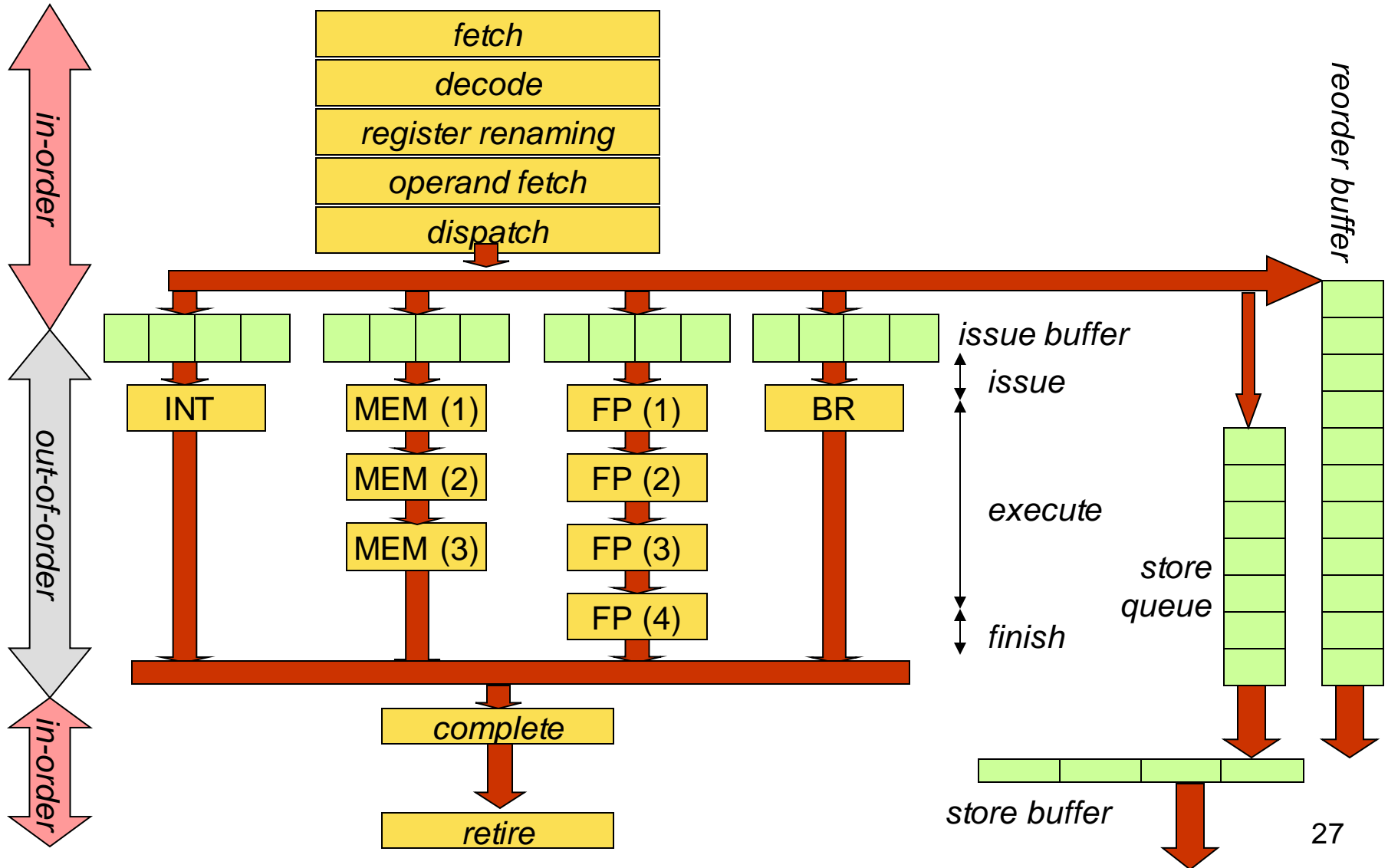
Only RAW dependences remain!



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.

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 instructions in reservation stations (issue buffers) 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 \neq 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

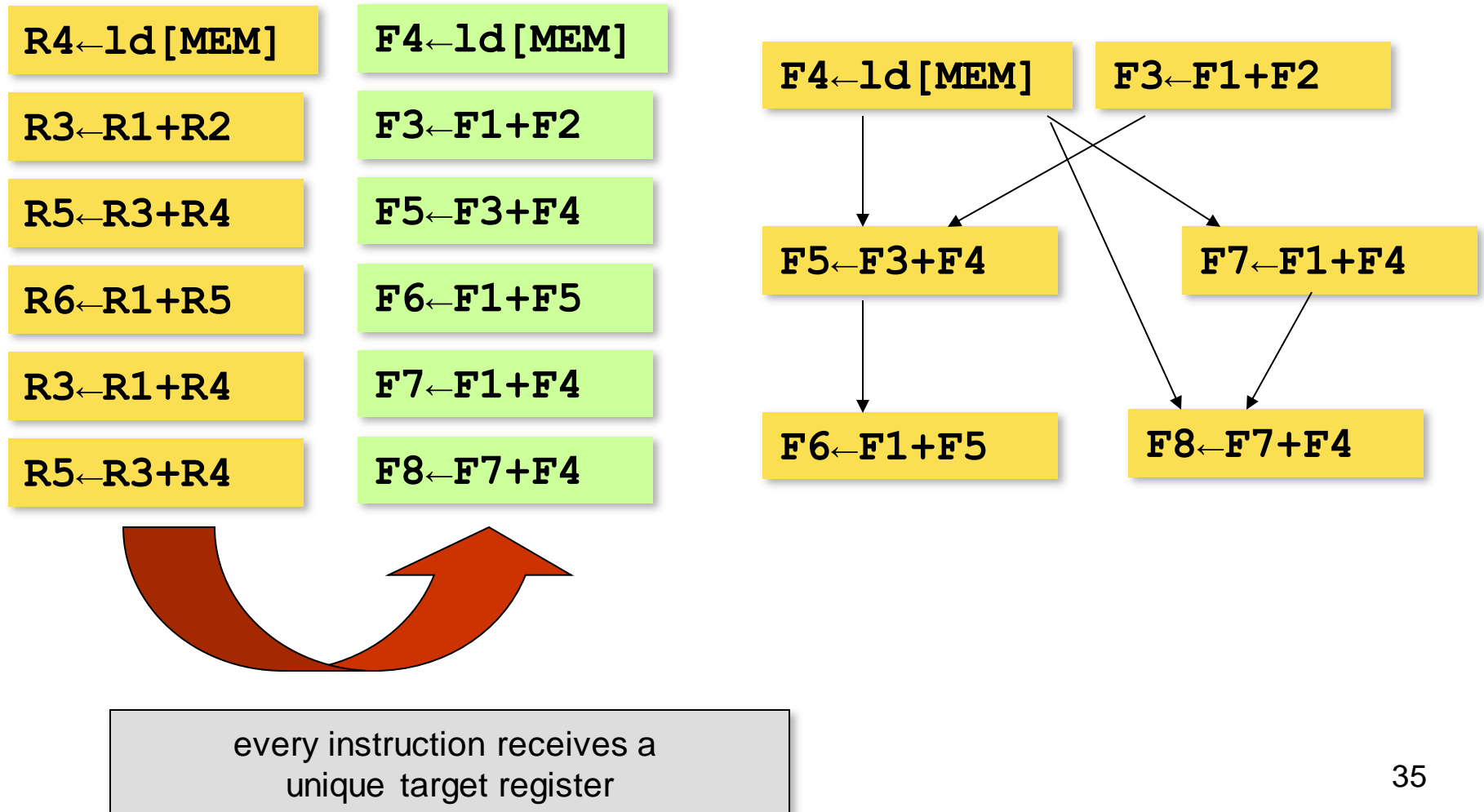
REGISTER RENAMING

*Slides in this section are by Lieven Eeckhout, Ghent University.
Reused with permission.*

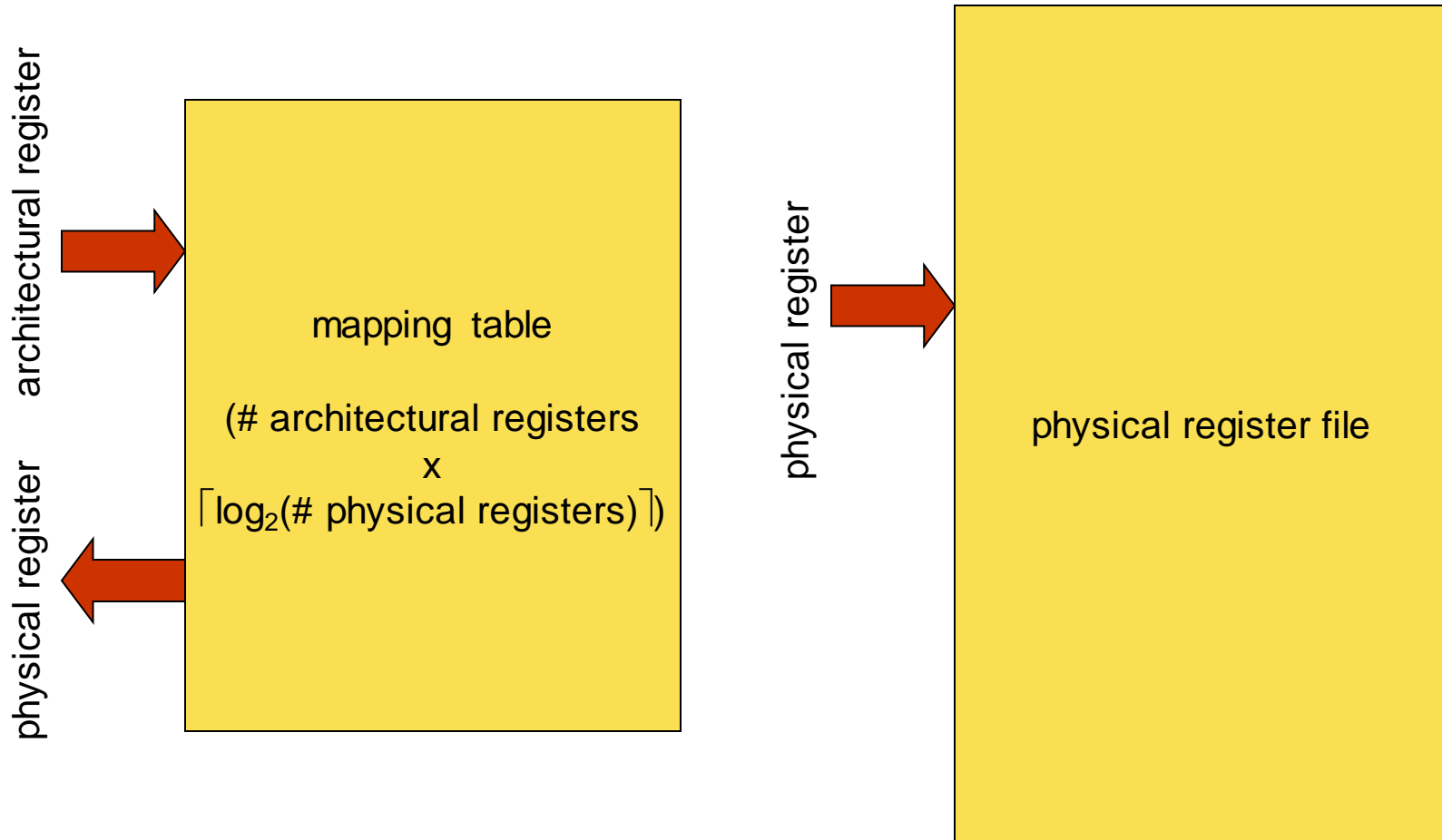
Register renaming

- Goal: remove anti- and output dependences through registers; only real data dependences through registers remain
- By doing so, we achieve the ‘data flow limit’
- Architectural registers are renamed to physical/rename registers
- **A physical register is written at most once by an instruction in execution**
 - this enables data flow execution through only real data dependences
- Done in HW

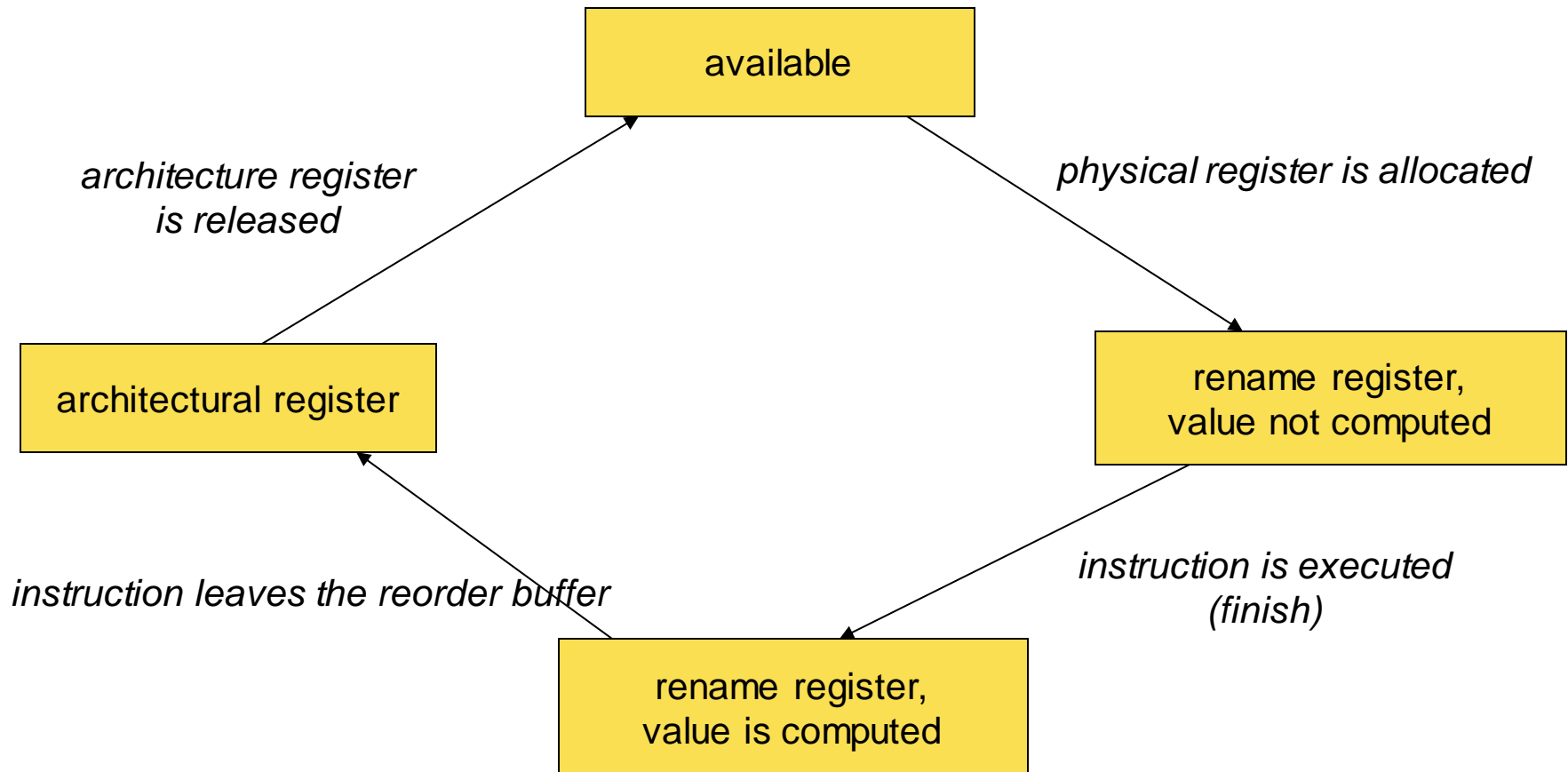
Register renaming: idea



Architectural to physical register mapping



FSM per physical register



Implementation

- Initialization
 - All physical registers that map to architectural registers are in the ‘architectural register’ state
 - All other physical registers are ‘available’
- An instruction’s input operand(s)
 - Read the physical register that corresponds with the architectural register from the mapping table
- Output operand
 - Select an ‘available’ physical register, and change the state to ‘rename register, value not computed’; update the mapping table
 - In case there are no more ‘available’ physical registers, stall the pipeline until physical registers become available

Implementation (cont.)

- When an instruction finishes its execution on a functional unit -- *finish*
 - Change state to ‘rename register, value computed’
- When an instruction leaves the ROB -- *complete*
 - (i) change state of the physical target register to ‘architectural register’ AND
 - (ii) the physical register previously associated with that same architecture register changes its state to ‘available’
 - Old value is no longer needed (for sure!)
 - (This previous mapping is kept track of in the ROB.)

Example

mapping table

physical register file

$R4 \leftarrow ld[MEM]$

$R3 \leftarrow R1 + R2$

$R5 \leftarrow R3 + R4$

$R5 \leftarrow R1 + R5$

$R3 \leftarrow R3 + R5$

$R5 \leftarrow R3 + R5$

R1 : F1

R2 : F2

R3 : F3

R4 : F4

R5 : F5

F1 (R1) : AR

F2 (R2) : AR

F3 (R3) : AR

F4 (R4) : AR

F5 (R5) : AR

F6 (--) : AV

F7 (--) : AV

F8 (--) : AV

F9 (--) : AV

F10 (--) : AV

AR = architectural register

AV = available

RR = rename register

Example

$R4 \leftarrow ld[MEM]$

$F6 \leftarrow ld[MEM]$

$R3 \leftarrow R1 + R2$

$R5 \leftarrow R3 + R4$

$R5 \leftarrow R1 + R5$

$R3 \leftarrow R3 + R5$

$R5 \leftarrow R3 + R5$

R1 : F1

R2 : F2

R3 : F3

R4 : F6

R5 : F5

F1 (R1) : AR

F2 (R2) : AR

F3 (R3) : AR

F4 (R4) : AR

F5 (R5) : AR

F6 (R4) : RR

F7 (--) : AV

F8 (--) : AV

F9 (--) : AV

F10 (--) : AV

AR = architectural register

AV = available

RR = rename register

Example

$R4 \leftarrow ld[MEM]$

$F6 \leftarrow ld[MEM]$

$R3 \leftarrow R1 + R2$

$F7 \leftarrow F1 + F2$

$R5 \leftarrow R3 + R4$

$R5 \leftarrow R1 + R5$

$R3 \leftarrow R3 + R5$

$R5 \leftarrow R3 + R5$

R1 : F1

R2 : F2

R3 : F7

R4 : F6

R5 : F5

F1 (R1) : AR

F2 (R2) : AR

F3 (R3) : AR

F4 (R4) : AR

F5 (R5) : AR

F6 (R4) : RR

F7 (R3) : RR

F8 (--) : AV

F9 (--) : AV

F10 (--) : AV

AR = architectural register

AV = available

RR = rename register

Example

$R4 \leftarrow ld[MEM]$

$F6 \leftarrow ld[MEM]$

$R3 \leftarrow R1 + R2$

$F7 \leftarrow F1 + F2$

$R5 \leftarrow R3 + R4$

$F8 \leftarrow F7 + F6$

$R5 \leftarrow R1 + R5$

$R3 \leftarrow R3 + R5$

$R5 \leftarrow R3 + R5$

R1 : F1

R2 : F2

R3 : F7

R4 : F6

R5 : F8

F1 (R1) : AR

F2 (R2) : AR

F3 (R3) : AR

F4 (R4) : AR

F5 (R5) : AR

F6 (R4) : RR

F7 (R3) : RR

F8 (R5) : RR

F9 (--) : AV

F10 (--) : AV

AR = architectural register

AV = available

RR = rename register

Example

$R4 \leftarrow ld[MEM]$

$F6 \leftarrow ld[MEM]$

$R3 \leftarrow R1 + R2$

$F7 \leftarrow F1 + F2$

$R5 \leftarrow R3 + R4$

$F8 \leftarrow F7 + F6$

$R5 \leftarrow R1 + R5$

$F9 \leftarrow F1 + F8$

$R3 \leftarrow R3 + R5$

$R5 \leftarrow R3 + R5$

R1 : F1

R2 : F2

R3 : F7

R4 : F6

R5 : F9

F1 (R1) : AR

F2 (R2) : AR

F3 (R3) : AR

F4 (R4) : AR

F5 (R5) : AR

F6 (R4) : RR

F7 (R3) : RR

F8 (R5) : RR

F9 (R5) : RR

F10 (--) : AV

AR = architectural register

AV = available

RR = rename register

Example

$R4 \leftarrow ld[MEM]$

$F6 \leftarrow ld[MEM]$

$R3 \leftarrow R1 + R2$

$F7 \leftarrow F1 + F2$

$R5 \leftarrow R3 + R4$

$F8 \leftarrow F7 + F6$

$R5 \leftarrow R1 + R5$

$F9 \leftarrow F1 + F8$

$R3 \leftarrow R3 + R5$

$F10 \leftarrow F7 + F9$

$R5 \leftarrow R3 + R5$

R1 : F1
R2 : F2
R3 : F10
R4 : F6
R5 : F9

F1 (R1) : AR
F2 (R2) : AR
F3 (R3) : AR
F4 (R4) : AR
F5 (R5) : AR
F6 (R4) : RR
F7 (R3) : RR
F8 (R5) : RR
F9 (R5) : RR
F10 (R3) : RR

AR = architectural register
AV = available
RR = rename register

Example

$R4 \leftarrow ld[MEM]$

$F6 \leftarrow ld[MEM]$

$R3 \leftarrow R1 + R2$

$F7 \leftarrow F1 + F2$

$R5 \leftarrow R3 + R4$

$F8 \leftarrow F7 + F6$

$R5 \leftarrow R1 + R5$

$F9 \leftarrow F1 + F8$

$R3 \leftarrow R3 + R5$

$F10 \leftarrow F7 + F9$

$R5 \leftarrow R3 + R5$

R1 : F1
R2 : F2
R3 : F10
R4 : F6
R5 : F9

F1 (R1) : AR
F2 (R2) : AR
F3 (R3) : AR
F4 (R4) : AR
F5 (R5) : AR
F6 (R4) : RR
F7 (R3) : RR
F8 (R5) : RR
F9 (R5) : RR
F10 (R3) : RR

Register renaming now blocks because there no more physical registers available

Example

$R4 \leftarrow ld[MEM]$

$R3 \leftarrow R1 + R2$

$R5 \leftarrow R3 + R4$

$R5 \leftarrow R1 + R5$

$R3 \leftarrow R3 + R5$

$R5 \leftarrow R3 + R5$

$F8 \leftarrow F7 + F6$

$F9 \leftarrow F1 + F8$

$F10 \leftarrow F7 + F9$

**R1 : F1
R2 : F2
R3 : F10
R4 : F6
R5 : F9**

**F1 (R1) : AR
F2 (R2) : AR
F3 (--) : AV
F4 (--) : AV
F5 (R5) : AR
F6 (R4) : AR
F7 (R3) : AR
F8 (R5) : RR
F9 (R5) : RR
F10 (R3) : RR**

Assume, the first two insns leave the ROB...

Example

$R4 \leftarrow ld[MEM]$

$R3 \leftarrow R1 + R2$

$R5 \leftarrow R3 + R4$

$R5 \leftarrow R1 + R5$

$R3 \leftarrow R3 + R5$

$R5 \leftarrow R3 + R5$

$F8 \leftarrow F7 + F6$

$F9 \leftarrow F1 + F8$

$F10 \leftarrow F7 + F9$

$F3 \leftarrow F10 + F9$

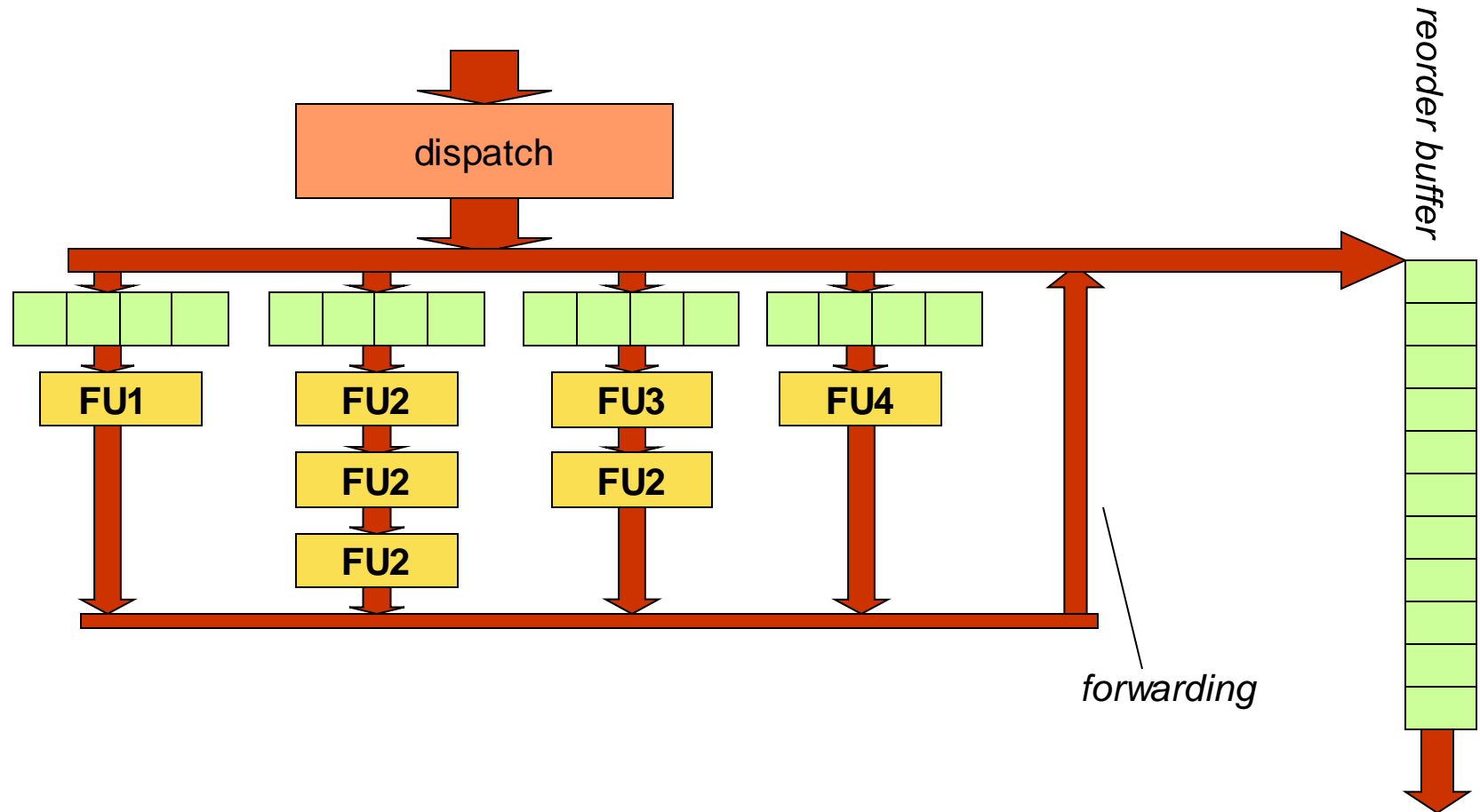
R1 : F1
R2 : F2
R3 : F10
R4 : F6
R5 : F3

F1 (R1) : AR
F2 (R2) : AR
F3 (R5) : RR
F4 (--) : AV
F5 (R5) : AR
F6 (R4) : AR
F7 (R3) : AR
F8 (R5) : RR
F9 (R5) : RR
F10 (R3) : RR

DATAFLOW EXECUTION

*Slides in this section are by Lieven Eeckhout, Ghent University.
Reused with permission.*

Out-of-order execution



Renaming and dispatch

- Register renaming is done in front-end of the pipeline
 - Only RAW dependences through registers remain
 - Happens before dispatch
- Dispatch
 - Allocate space and insert insns in reservation station (issue buffer)
 - Allocate space in reorder buffer (ROB)
 - In case no space left in issue buffer OR reorder buffer, stall the front-end pipeline (all stages up until dispatch)

Reservation station = issue buffer

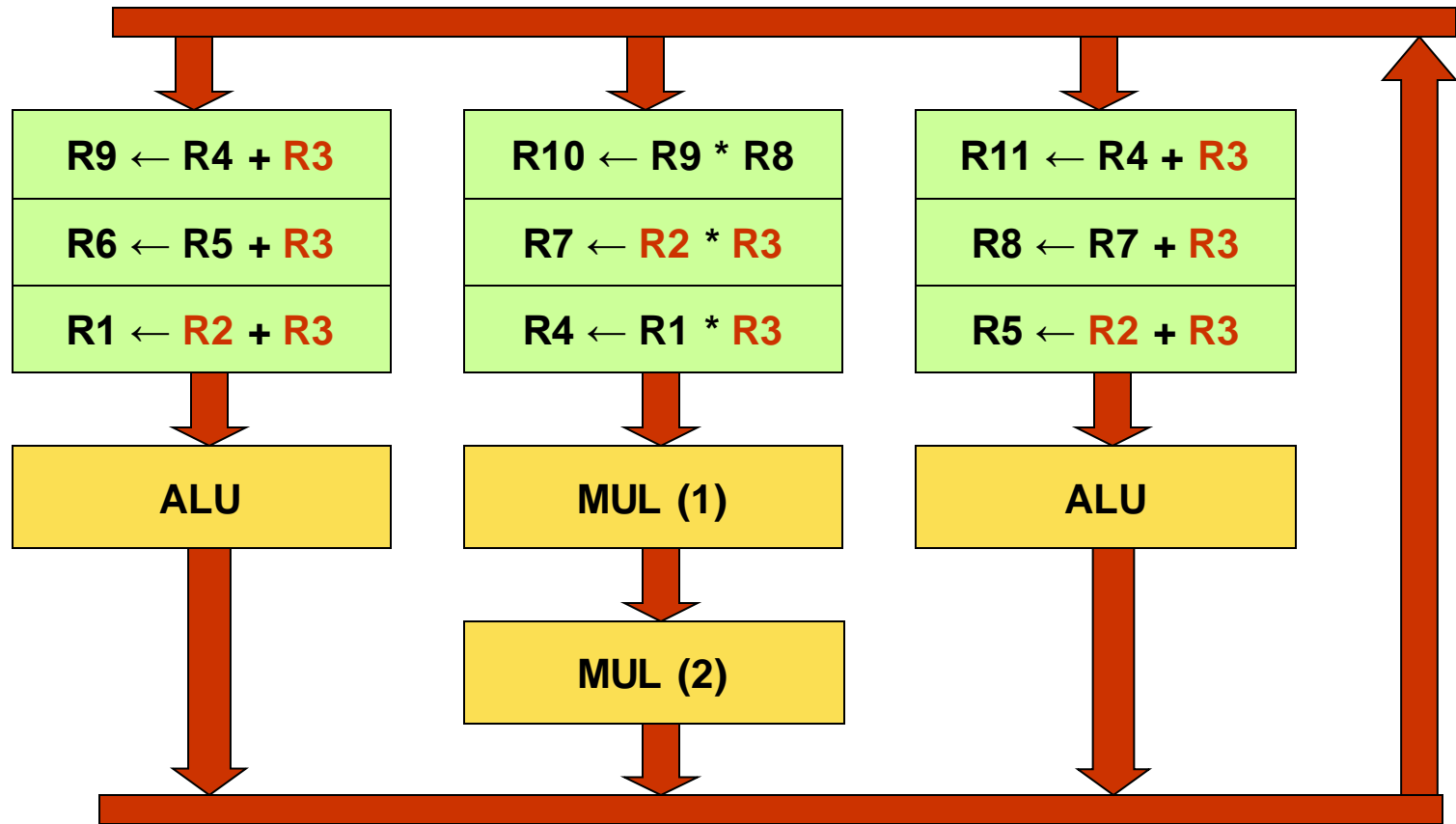
- Buffering of insns that wait for their input operands; once the operands are available, insns are issued to a functional unit
- Instruction waits for its input operands
 - (note: only RAW dependences!)
 - Either available from the OF stage
 - Or, the value still needs to be produced
- An instruction leaves the issue buffer when issued or when executed
- This principle was introduced by Tomasulo in the IBM 360/91 in 1967!
- Issue buffer decouples front-end from back-end of the pipeline

Reservation station:

How it works

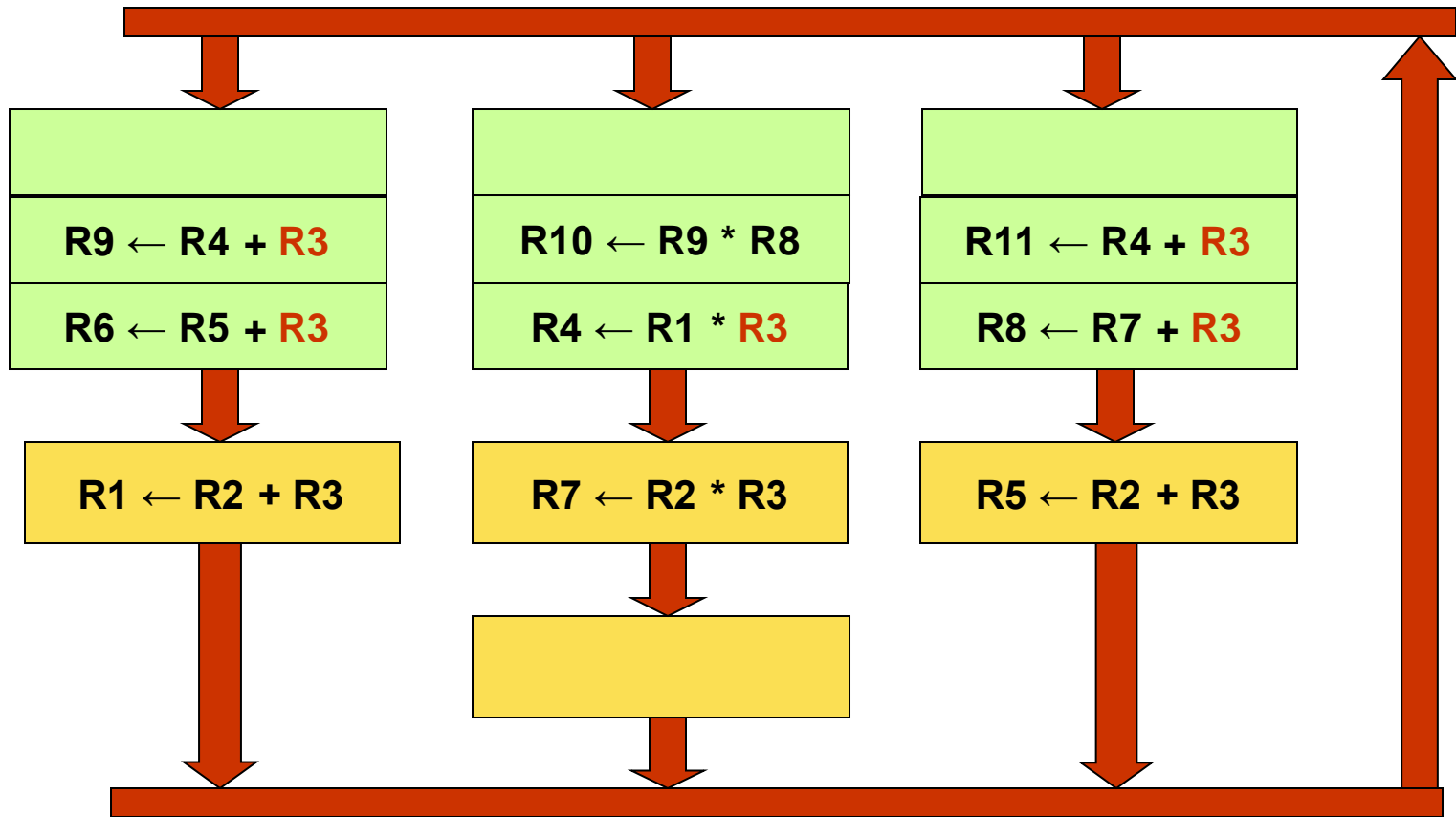
- Insns wait until all input operands are available (*ready* = 1)
- If ready ***and*** FU is available, start instruction execution -- *issue*
- When execution is done – *finish* – put the target register ID and the result on the (forwarding) bus
- Instructions ‘watch’ the result bus for available registers

Example

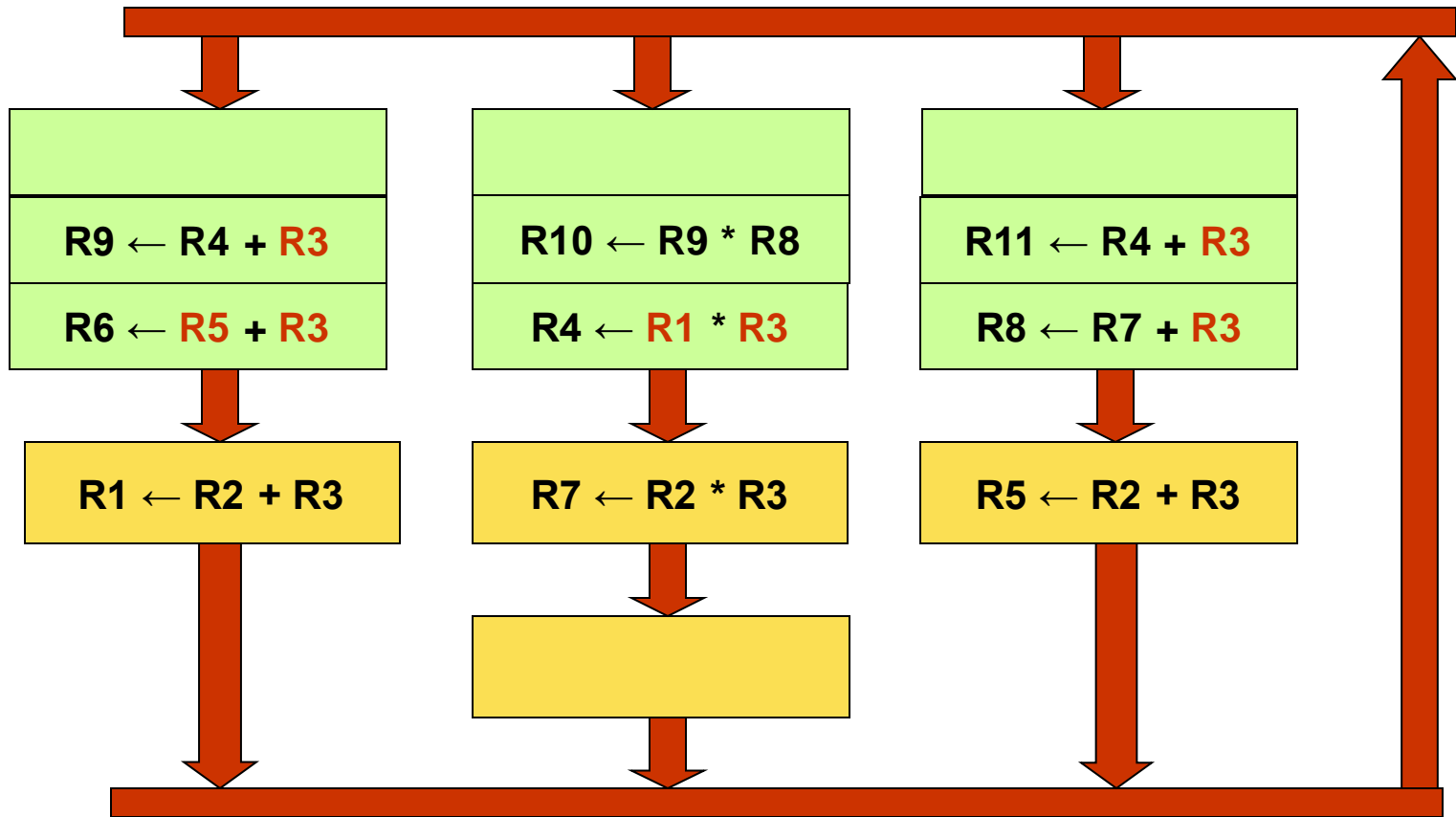


target operand and result bus

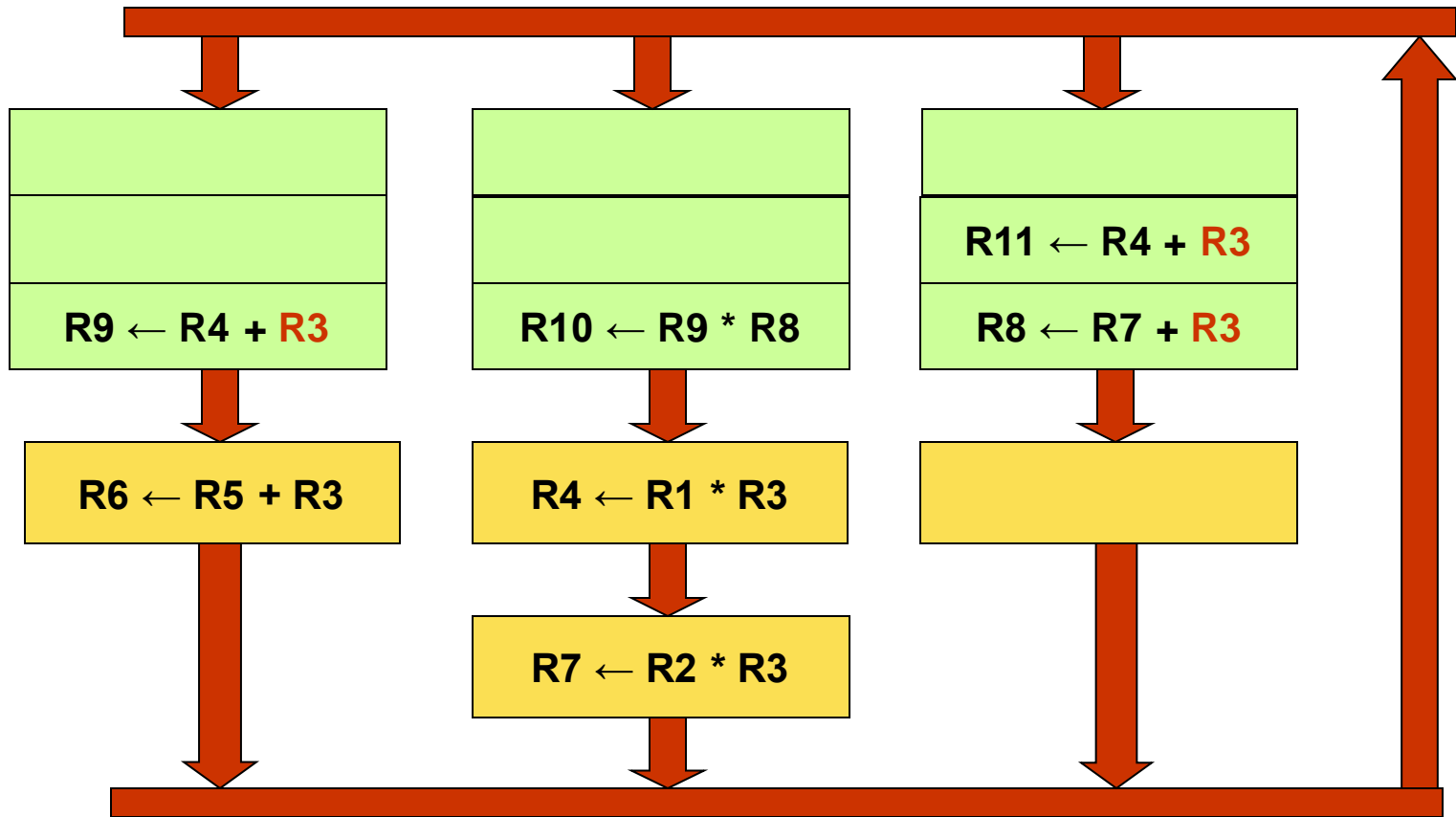
Example – cycle 1 (issue)



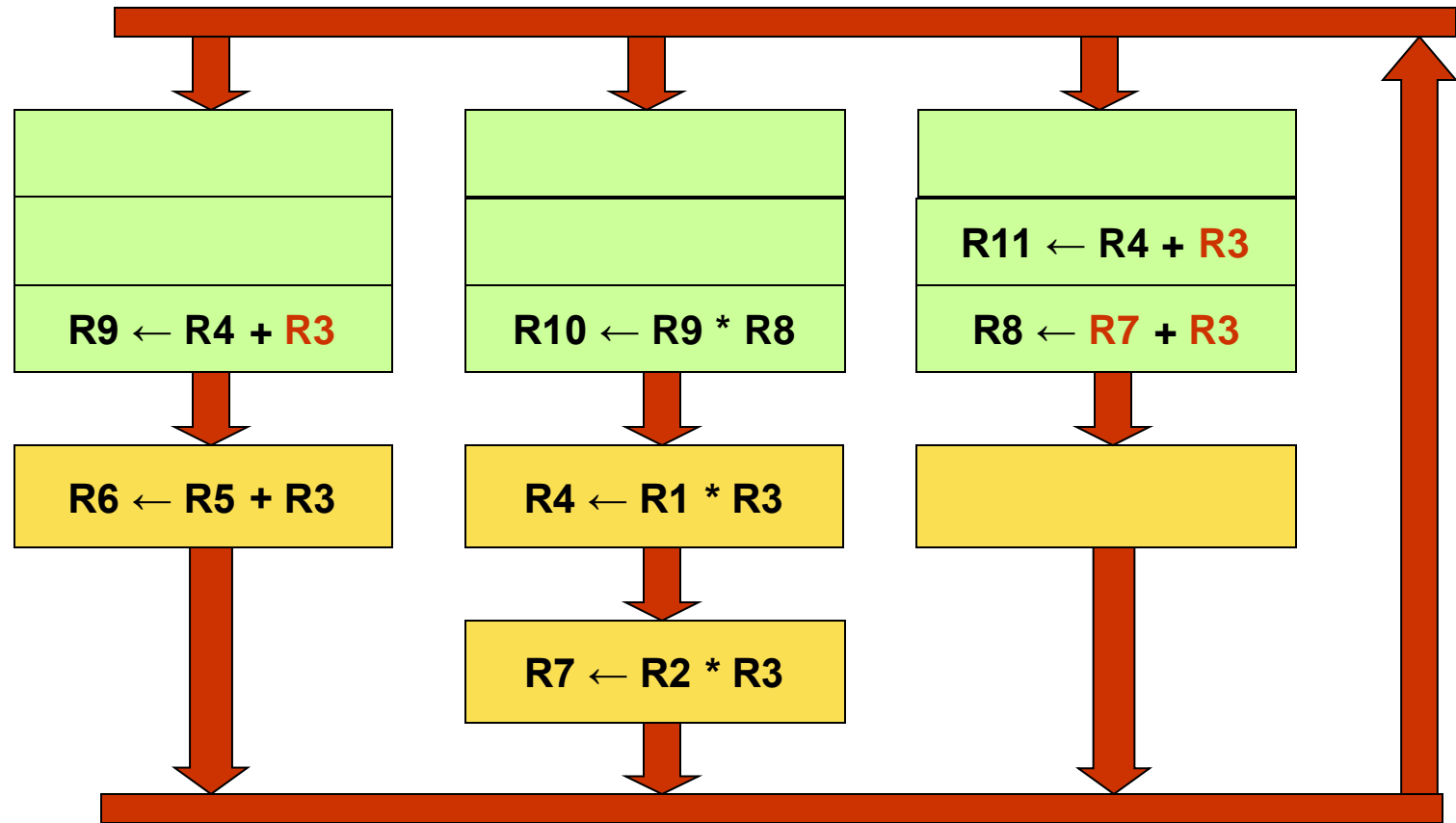
Example – cycle 1 (finish)



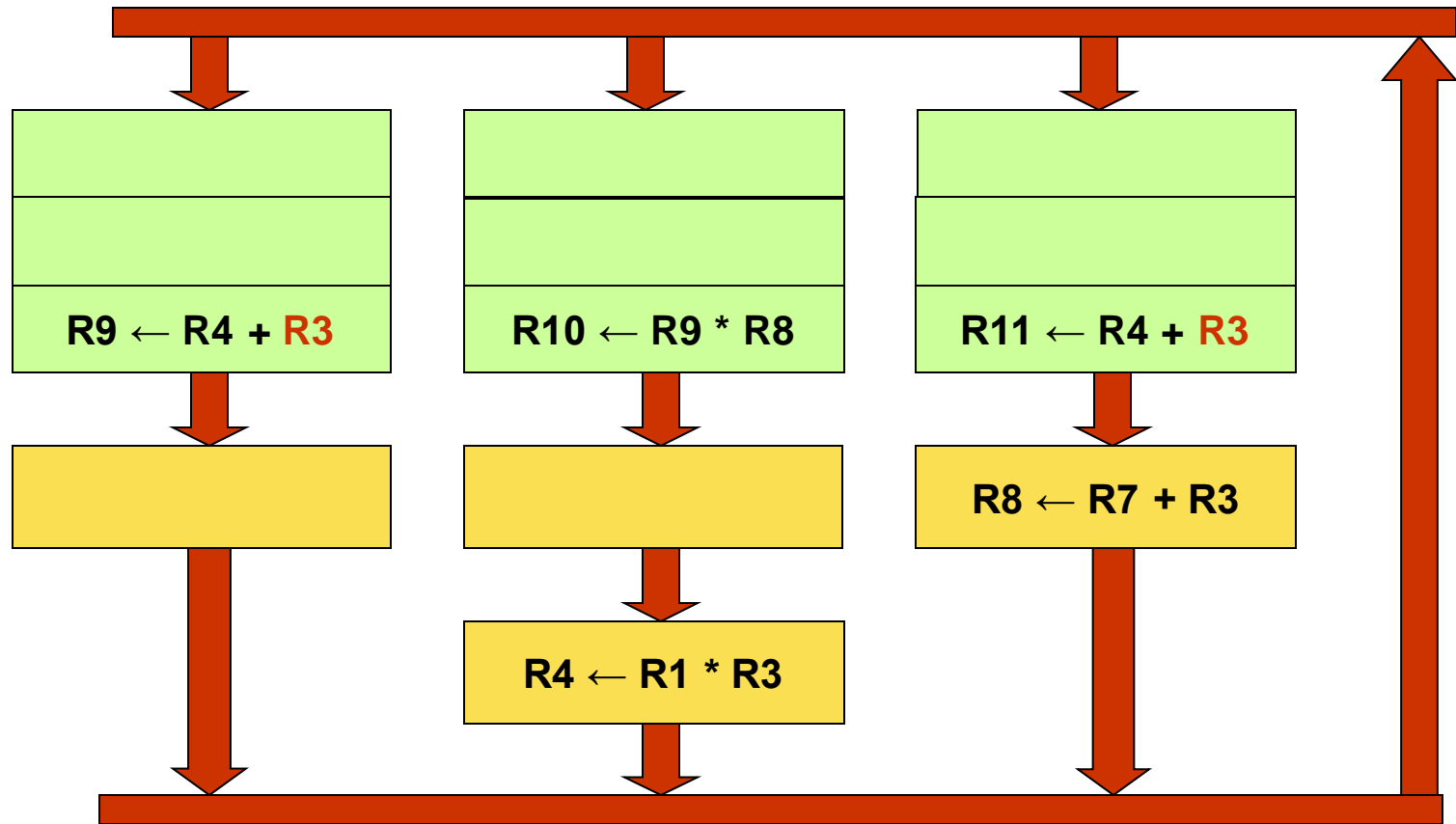
Example – cycle 2 (issue)



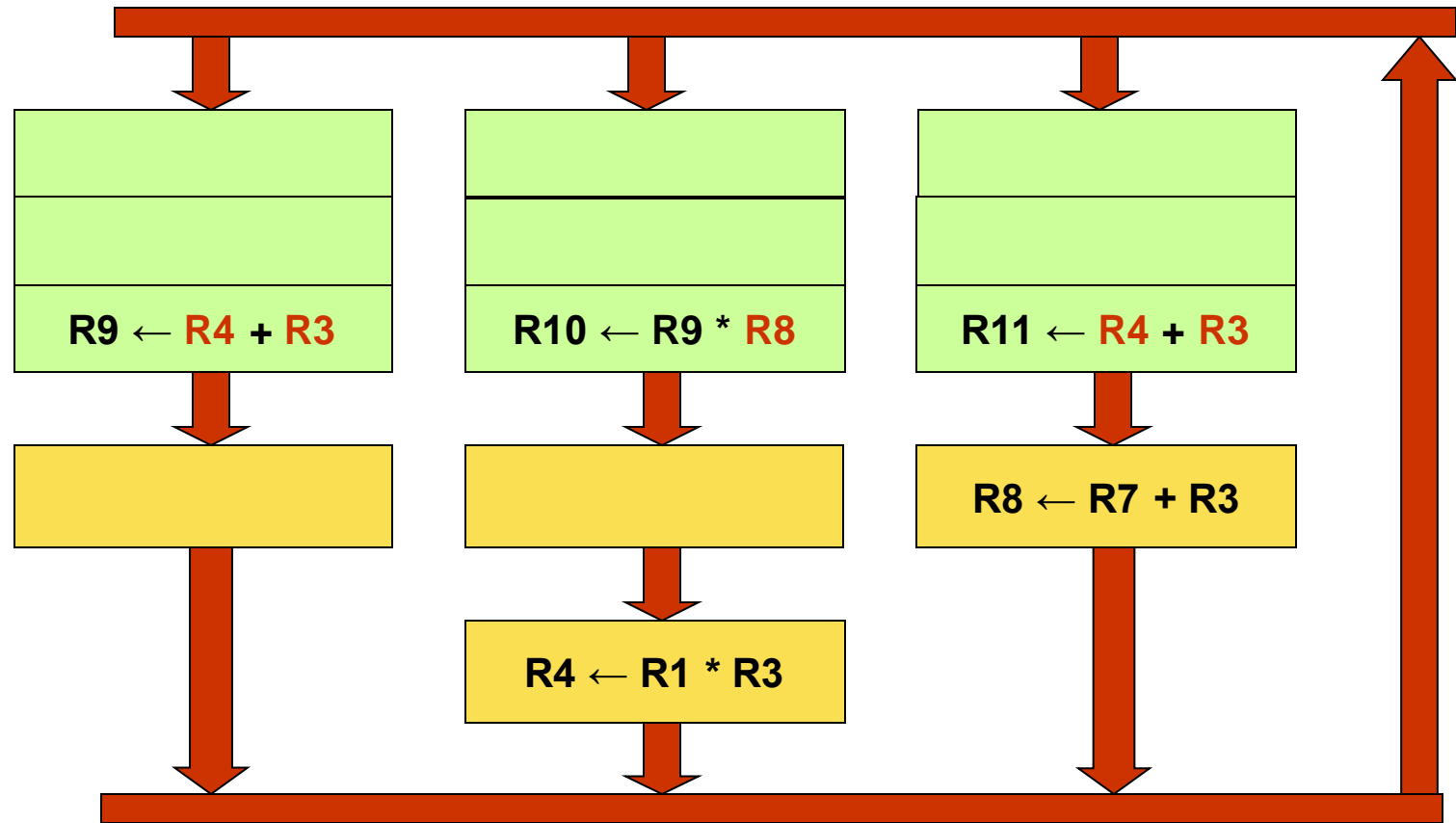
Example – cycle 2 (finish)



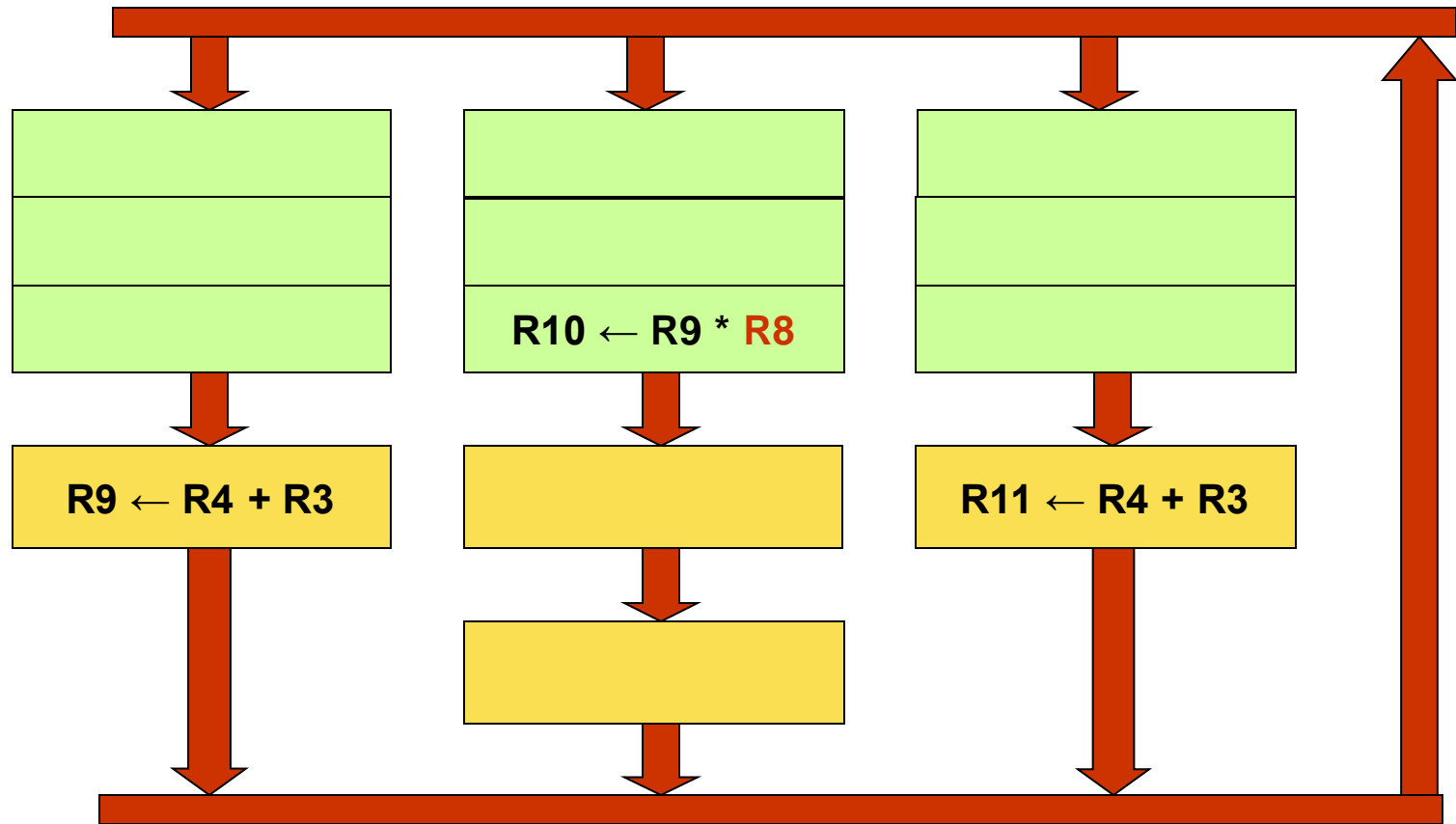
Example – cycle 3 (issue)



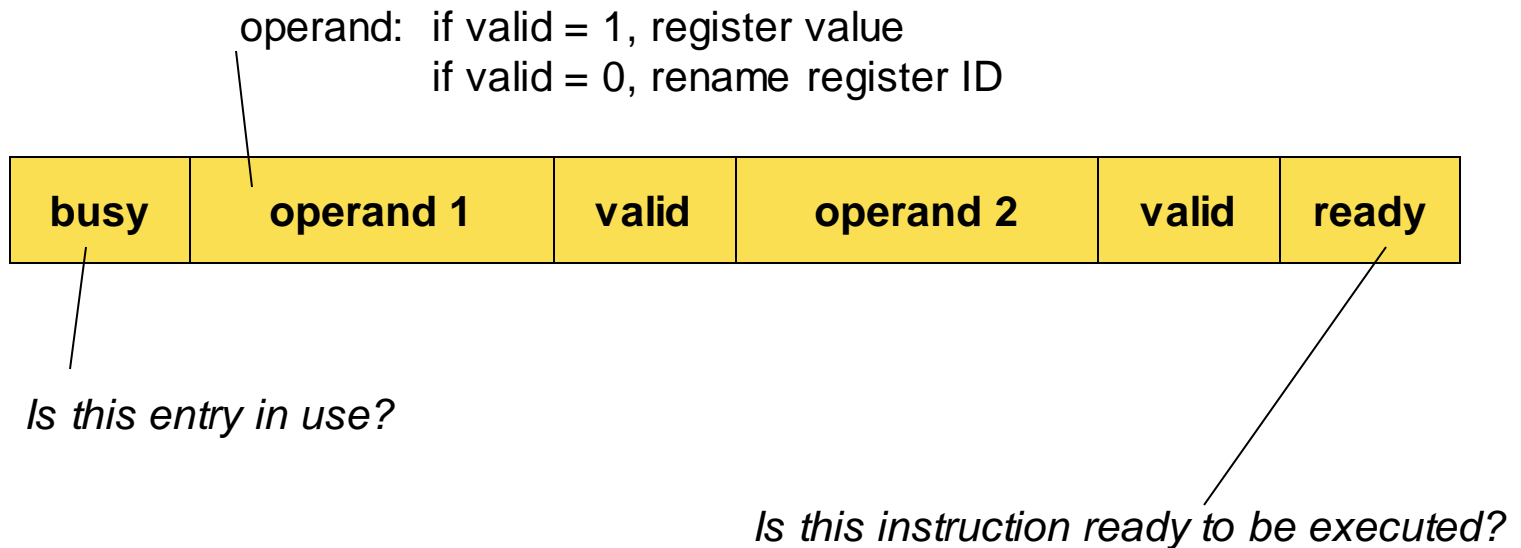
Example – cycle 3 (finish)



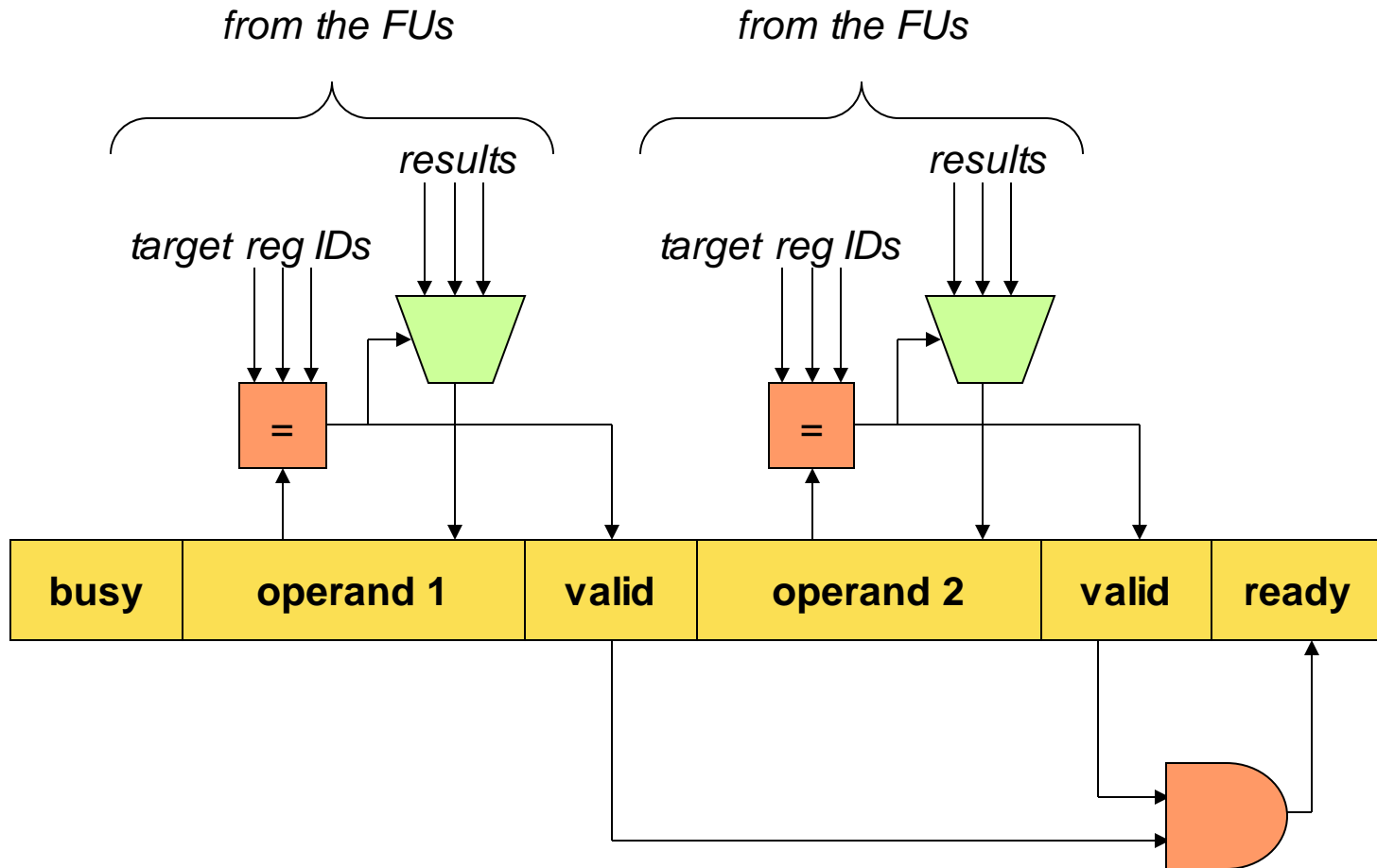
Example – cycle 4 (issue)



Reservation station entry



Wakeup



Instruction selection

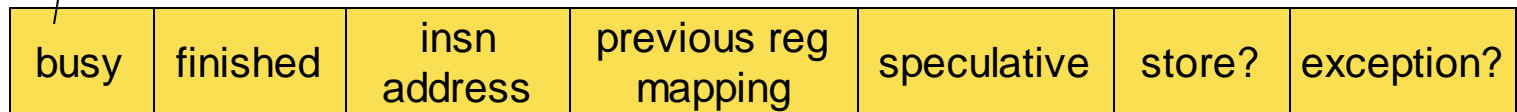
- Instruction can be selected for execution if all input operands are available (*ready = 1*)
- In case more than one instruction is ready to execute
 - Select oldest-first
 - Other insns wait till next cycle
- *Dynamic instruction scheduling*

Reorder buffer

- Contains all *in-flight* instructions
 - This includes all instructions after dispatch and before completion
 - Insns in issue buffer(s) is a subset of the insns in the ROB

ROB entry

element is in use



The diagram shows a horizontal row of seven yellow rectangular boxes, each representing a field in a ROB entry. A thin black line originates from the text 'element is in use' and points to the first box, which is labeled 'busy'.

busy	finished	insn address	previous reg mapping	speculative	store?	exception?
------	----------	-----------------	-------------------------	-------------	--------	------------

Reorder buffer

- Circular buffer w/ head and tail pointer
- Number of new insns per cycle is limited by the *dispatch width*
 - Dispatch happens at the tail
 - In-order dispatch
- Number of insns that leave the ROB is limited by the *completion width*
 - Completion happens at the head
 - In-order completion

Reorder buffer vs. reservation station

- Reorder buffer
 - Contains all in-flight insns
 - From dispatch till completion
- Reservation station
 - Contains insns that haven't been issued yet (or are being executed)
 - Is subset of insns in the ROB

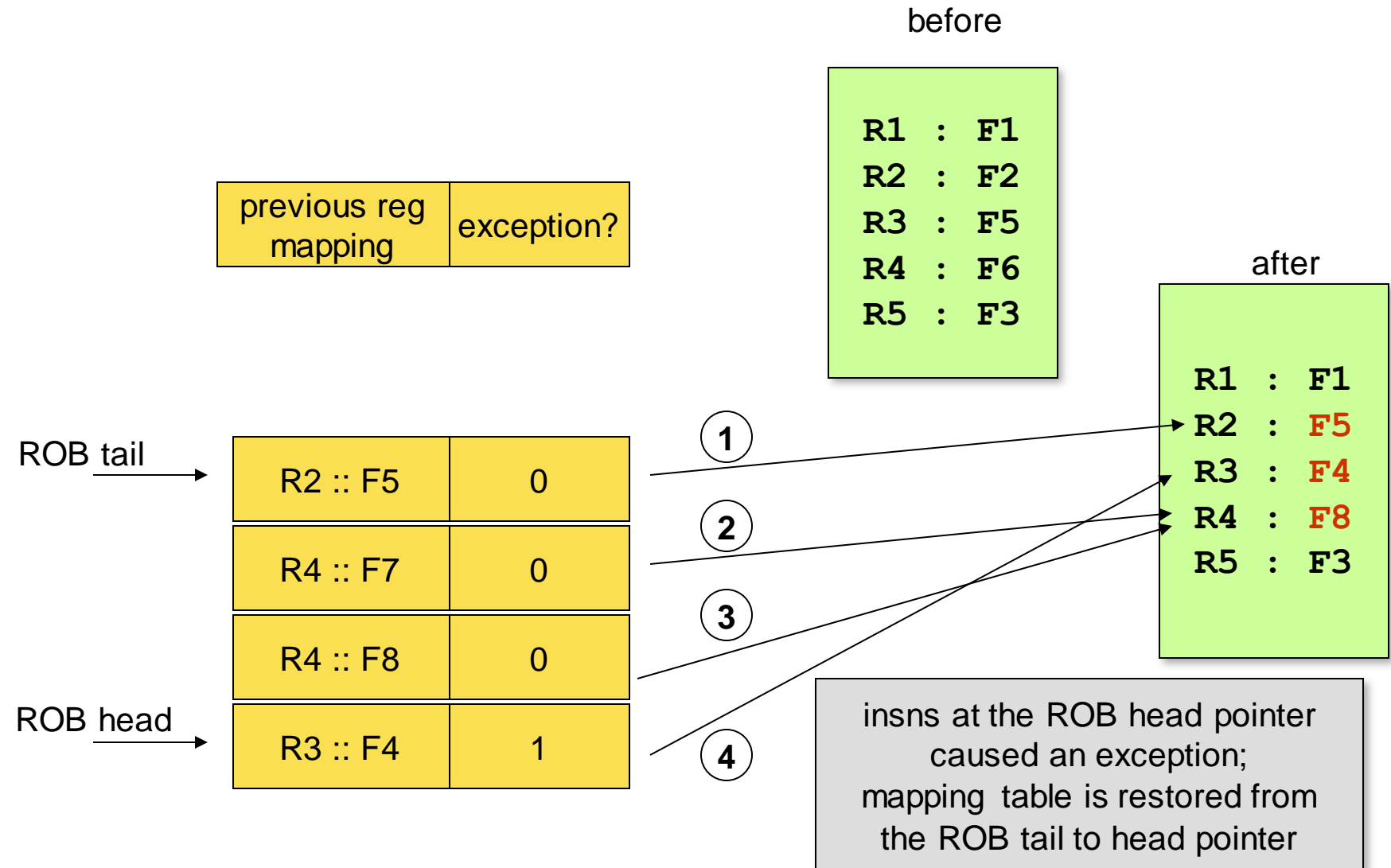
Instruction window

- ROB and reservation station can be combined into one structure, called the *instruction window*
 - Was done in the first out-of-order processors
 - e.g., HP PA-8000
- Disadvantage is the large structure and HW complexity (long wires)
- Most modern out-of-order processors use an ROB and reservation stations

Exceptions

- Precise exceptions
 - Recall: save architecture state prior to the exception; handle the exception; restart execution from the instruction that caused the exception
- How is this done in OOO processors?
 - Recall: ROB w/ in-order completion
 - Instructions that generate an exception set a flag in their ROB entry; insns prior to the excepting insn can complete; exception is handled when the excepting insn is about to be completed

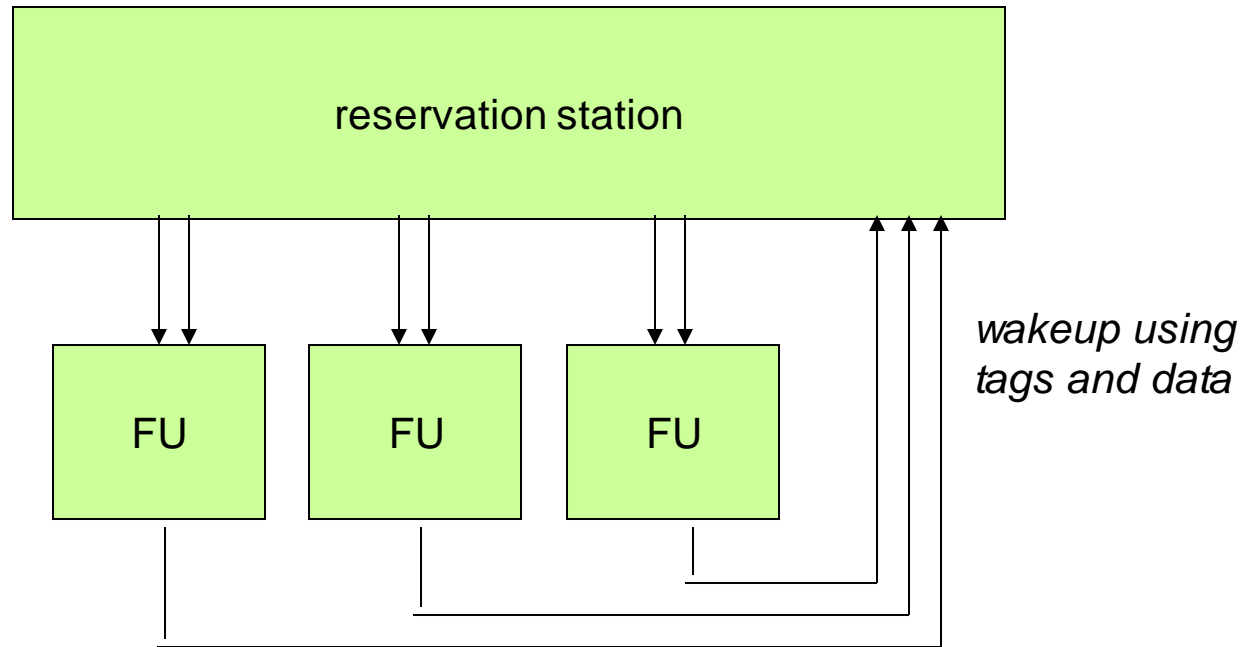
What about register renaming upon an exception?



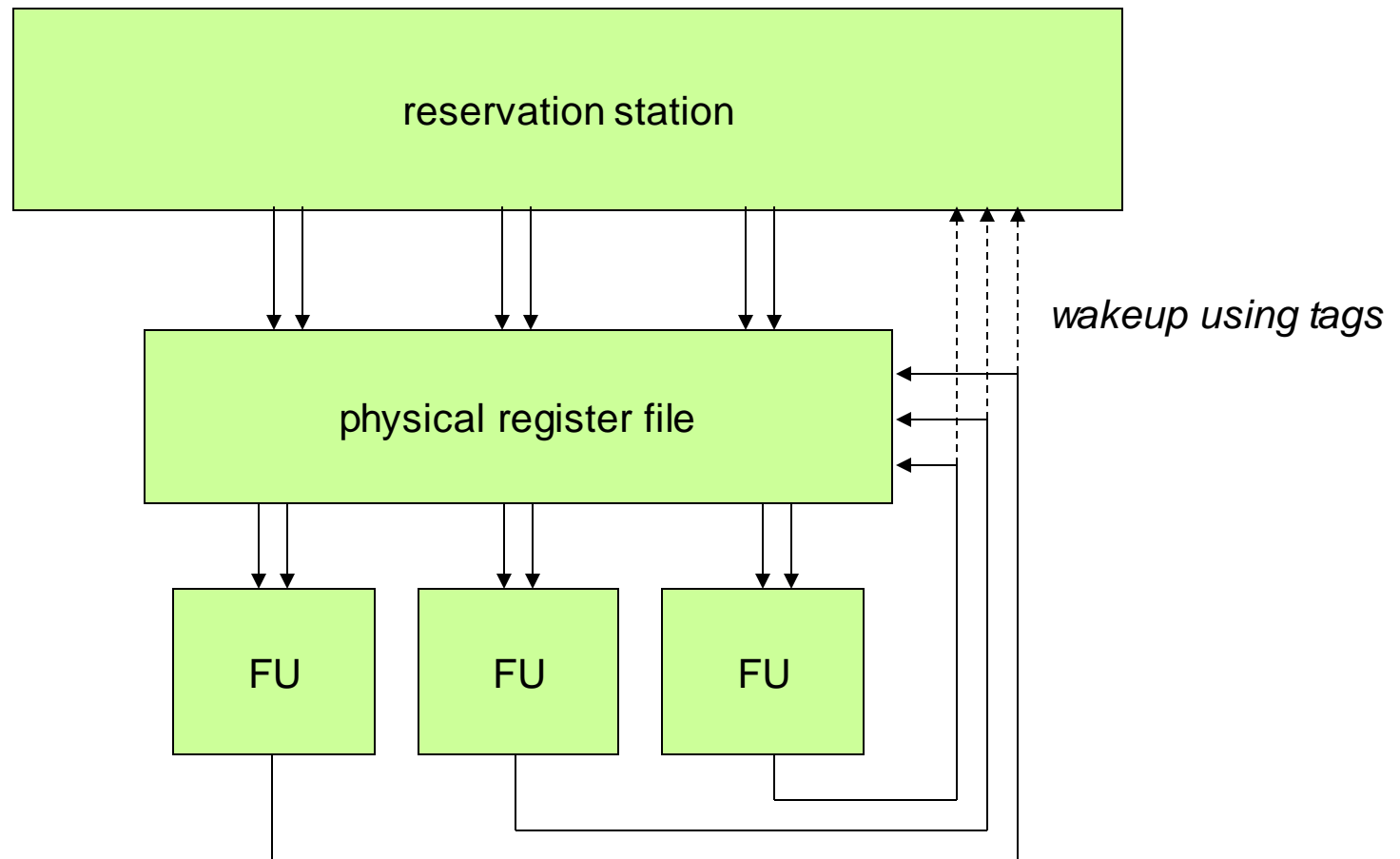
What about mispredicted branches?

- Same principle
 - Mapping between architectural and physical registers needs to be restored to the state just after the branch
- Can be implemented the same way as for exceptions
 - Mispredicted branches are more frequent than exceptions though, hence...
- A more efficient solution: *checkpointing*
 - Take a snapshot of the mapping table after each branch
 - Restore the mapping table from the snapshot upon a mispredicted branch

Data captured scheduling



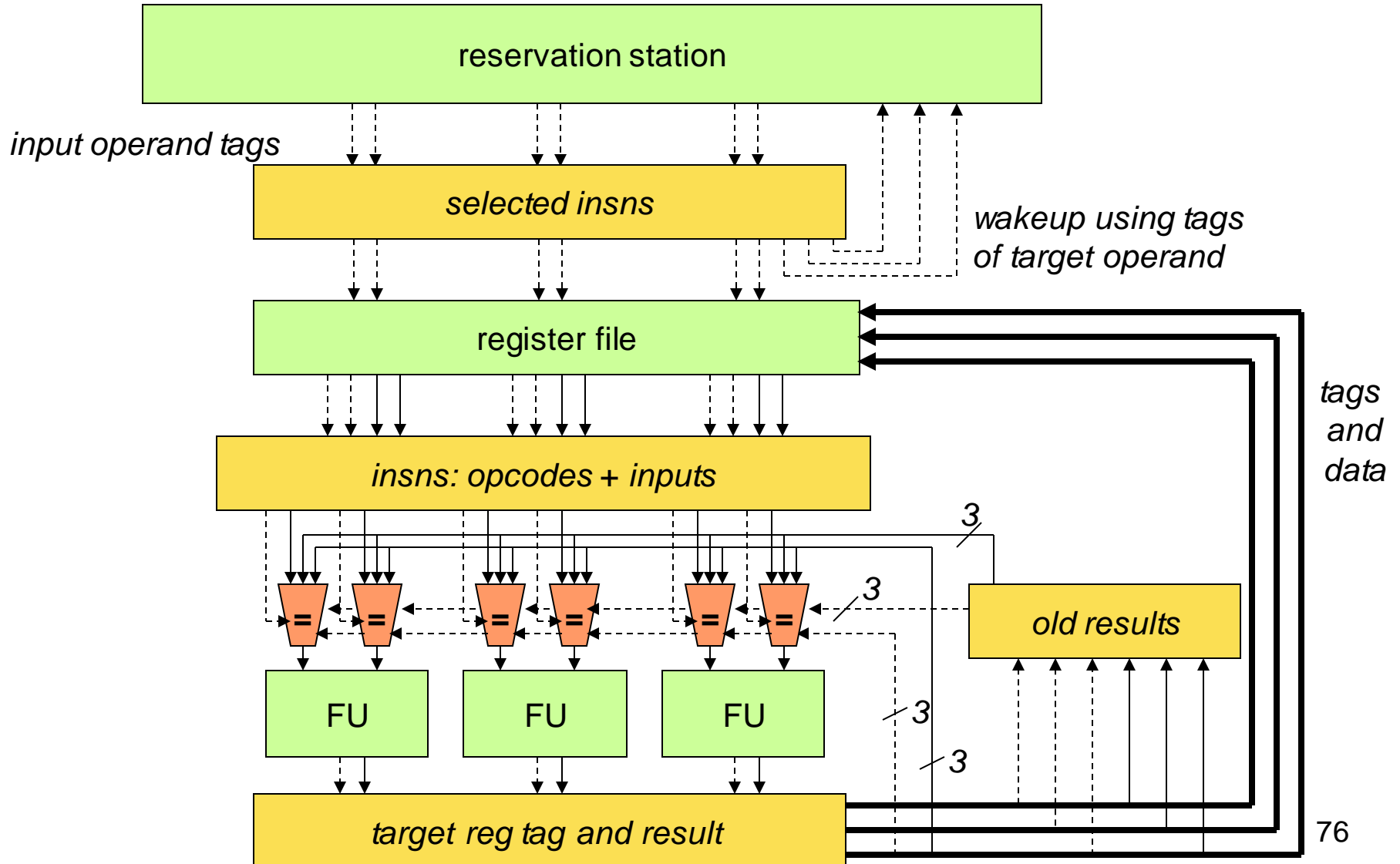
Non-data captured scheduling



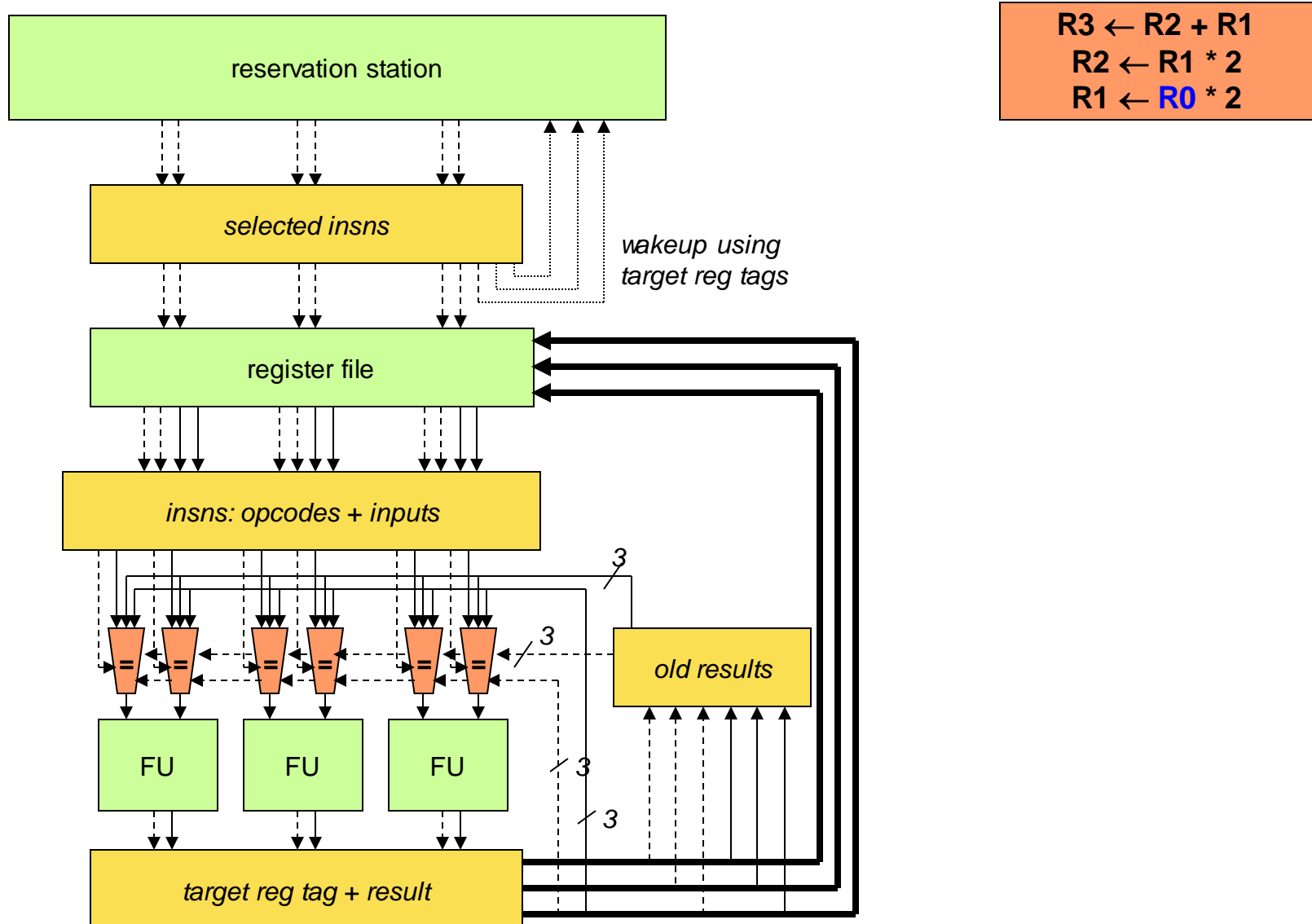
Advantage non-data captured organization

- Reservation station is less complex
 - No register values
 - Only register tags
 - Much less wiring between the FUs and the reservation station entries!

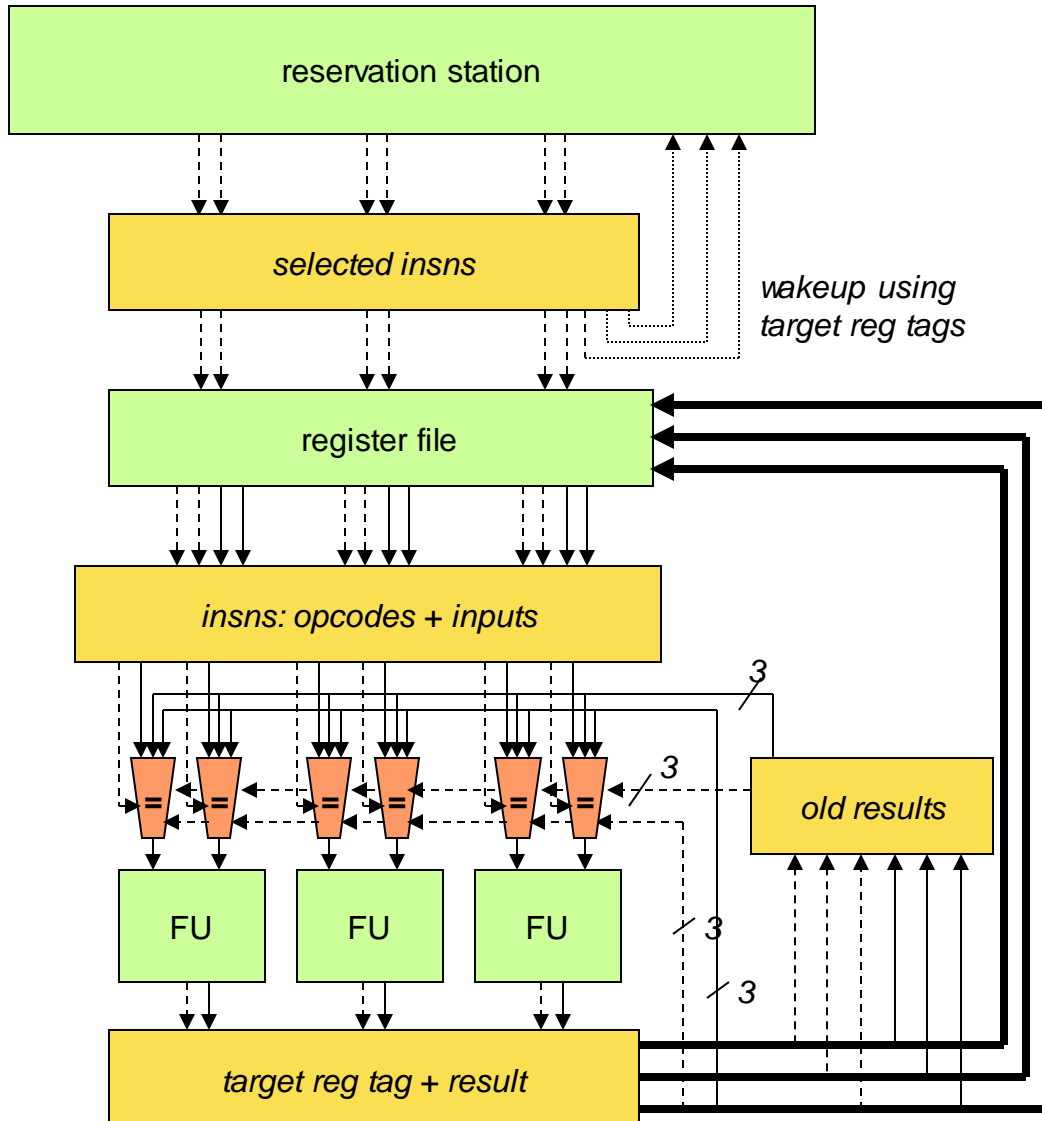
Pipelined organization



Example: cycle 0



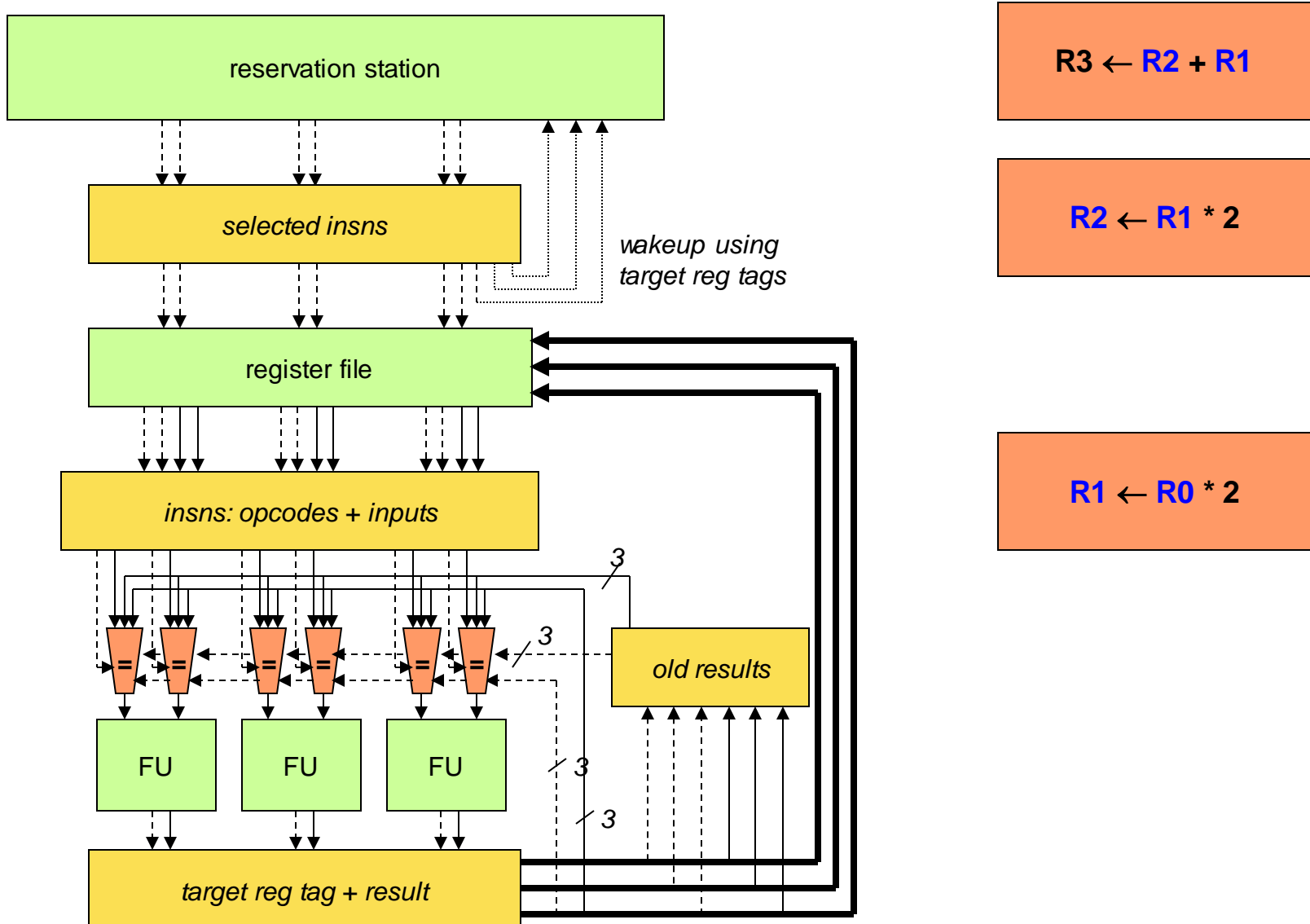
Example: cycle 1



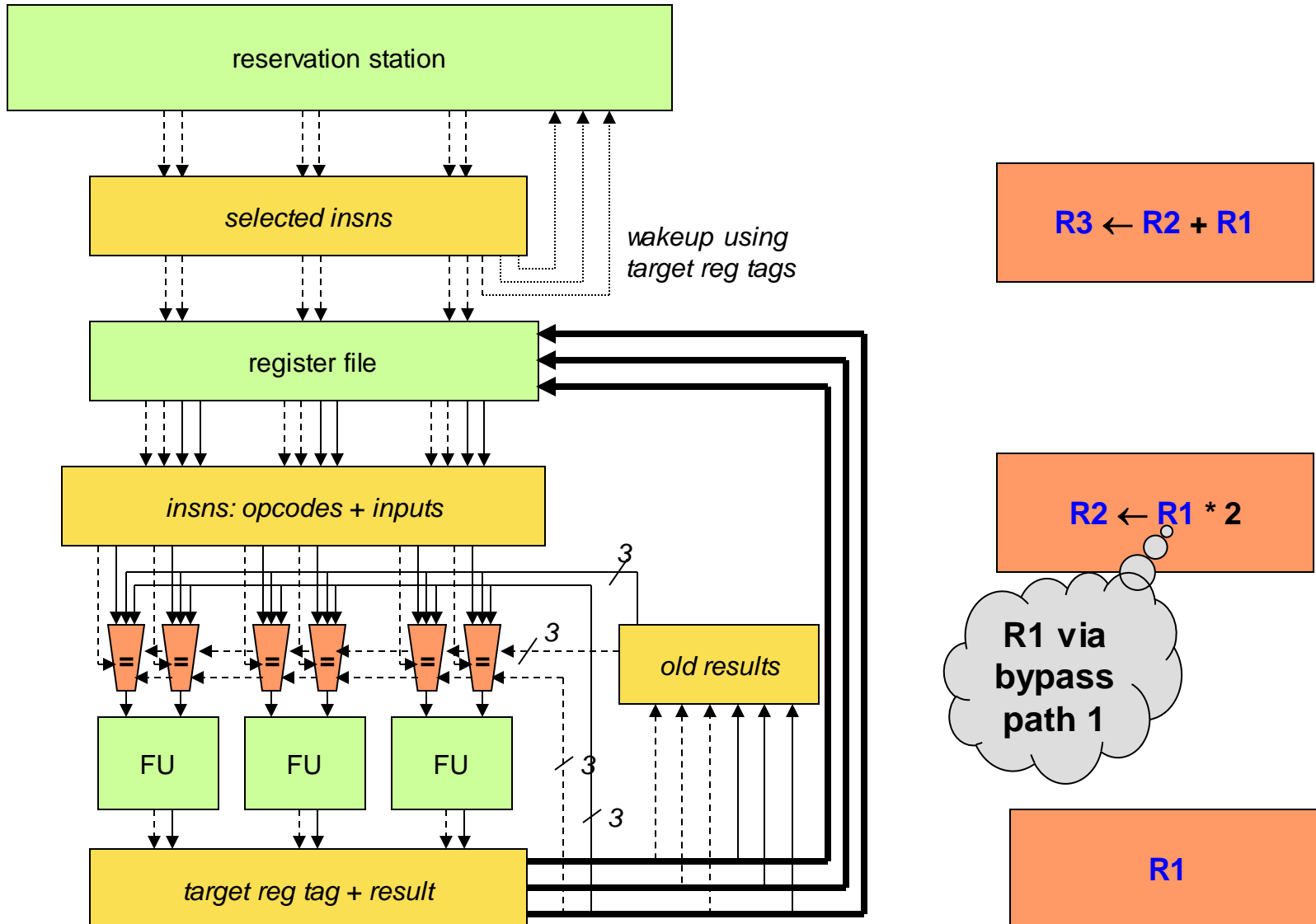
$R3 \leftarrow R2 + R1$
 $R2 \leftarrow R1 * 2$

$R1 \leftarrow R0 * 2$

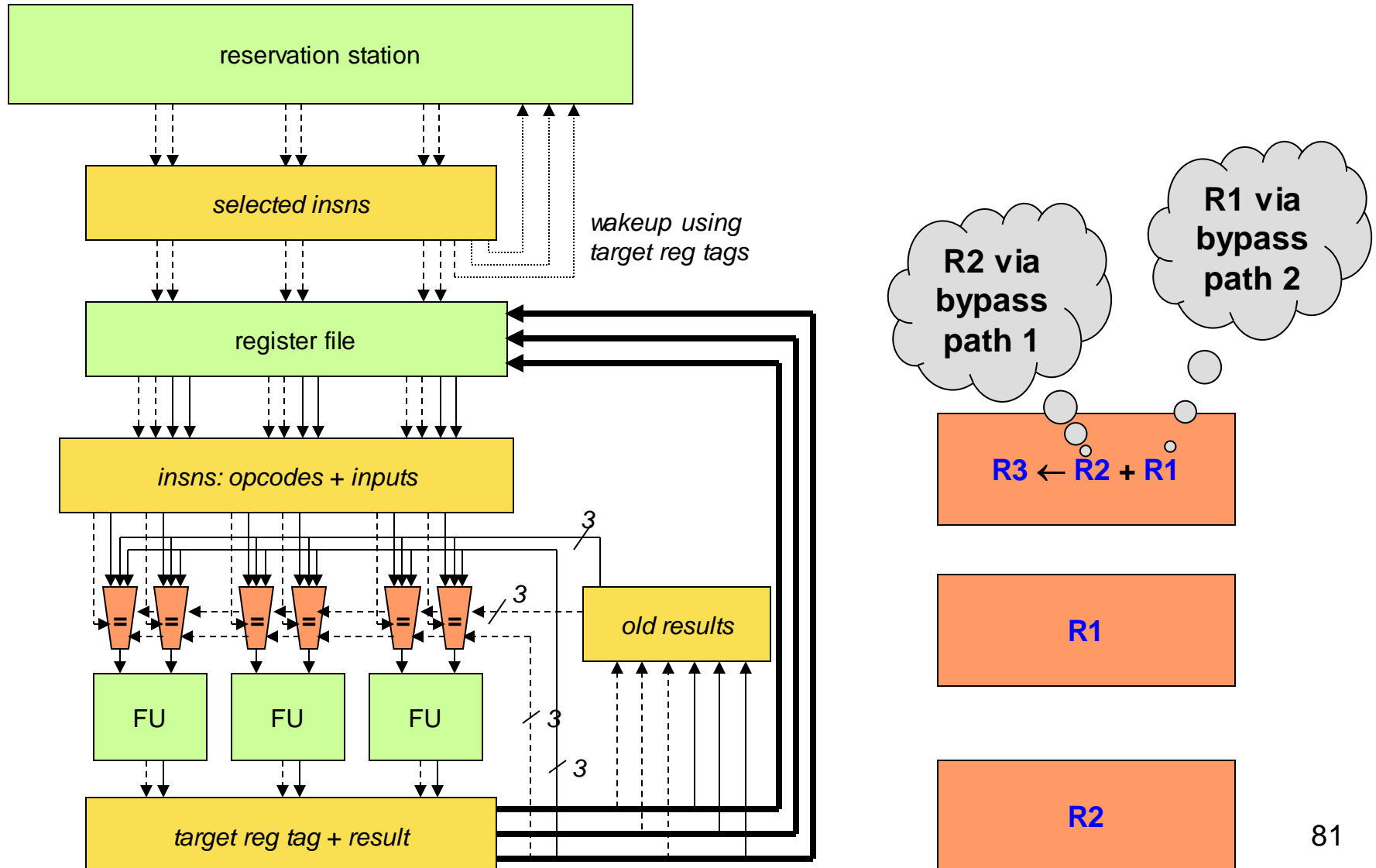
Example: cycle 2



Example: cycle 3



Example: cycle 4



LIMITS OF DATAFLOW EXECUTION

*Slides in this section are by Lieven Eeckhout, Ghent University.
Reused with permission.*

Is the data flow limit the real limit?

- Recall: register renaming removes all non-real data register dependences
- ***Data flow limit***
- “It’s impossible to break this limit”
- Research was done to break this limit
 - Value prediction
 - Around 1995

Value prediction

- Predict
 - the outcome of operations
 - loads
 - ALU operations
 - at the beginning of the pipeline
- Benefit: no need to wait for the result to be computed, hence dependent insns can execute sooner
- Speculative technique: prediction needs to be verified
 - if incorrect: restore state and re-execute

Why does this work?

- Value locality
 - Some insns repeatedly generate the same value
 - Other insns generate easy-to-predict patterns
 - Examples: loop iterator, reading a constant value from memory, initialization, strided accesses, etc.
- Implementation: keep track of old value (and stride) produced by an insn, to predict future values

SUMMARY

Summary

- Scalar pipelines have fundamental limitations
 - Treat all instructions equally although they require a different amount of computation (e.g., integer add vs. floating point div)
 - Maximum IPC is 1
 - All instructions are executed in order – even if they are independent
- Out-of-order execution
 - Removes the limitations of scalar pipelines at the cost of increased complexity
 - Register renaming to remove WAR and WAW hazards
 - Issue buffers enable issuing instructions out-of-order
 - The Reorder Buffer (ROB) makes sure that instructions commit in order (and thereby support precise exceptions)