# Advanced Computer Architecture

—

## Part I: General Purpose Exploiting ILP Dynamically

paolo.ienne@epfl.ch

EPFL

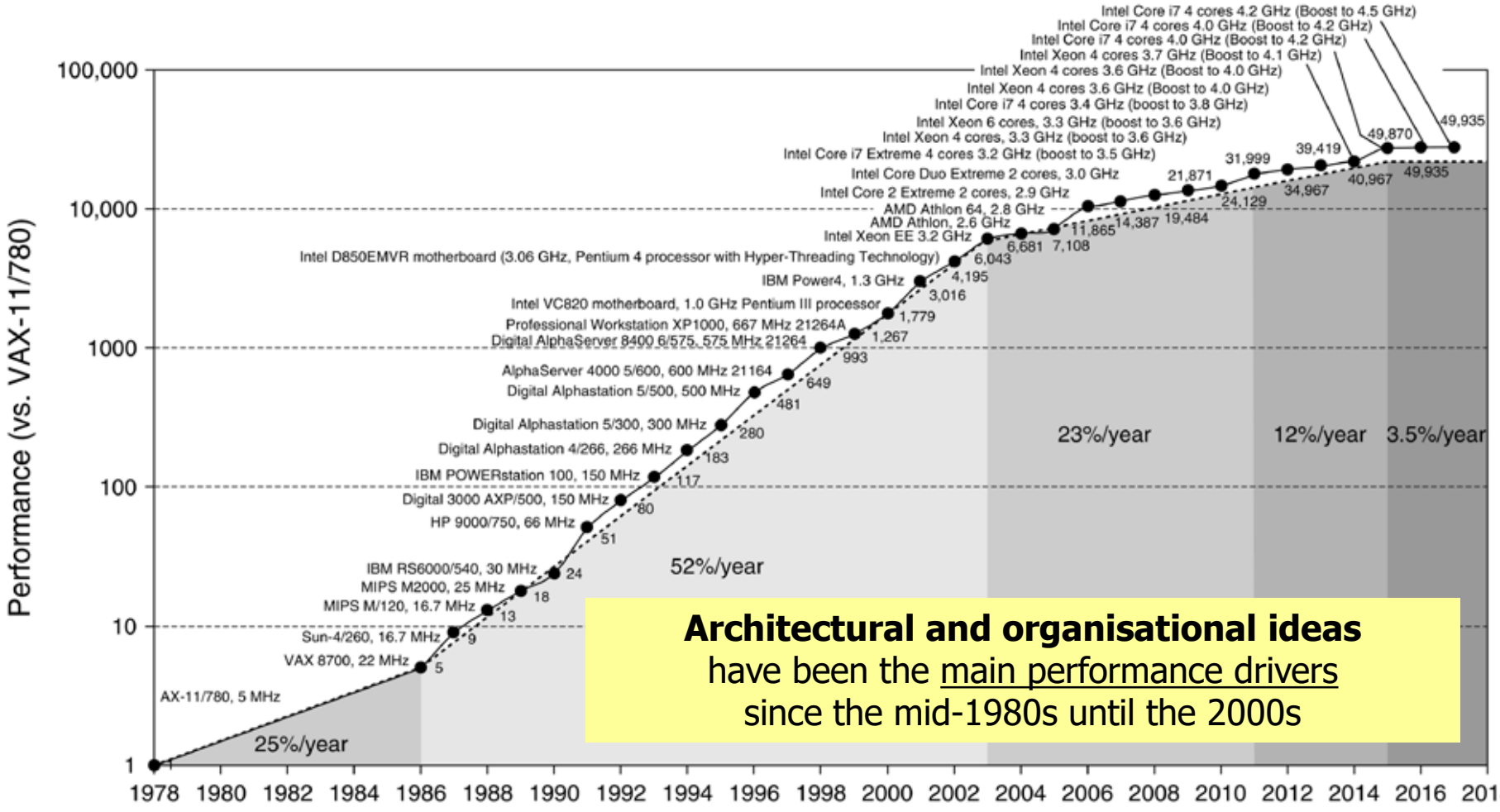
# 1

## ILP? The Traditional Way

## (Let's Make It Fast!)

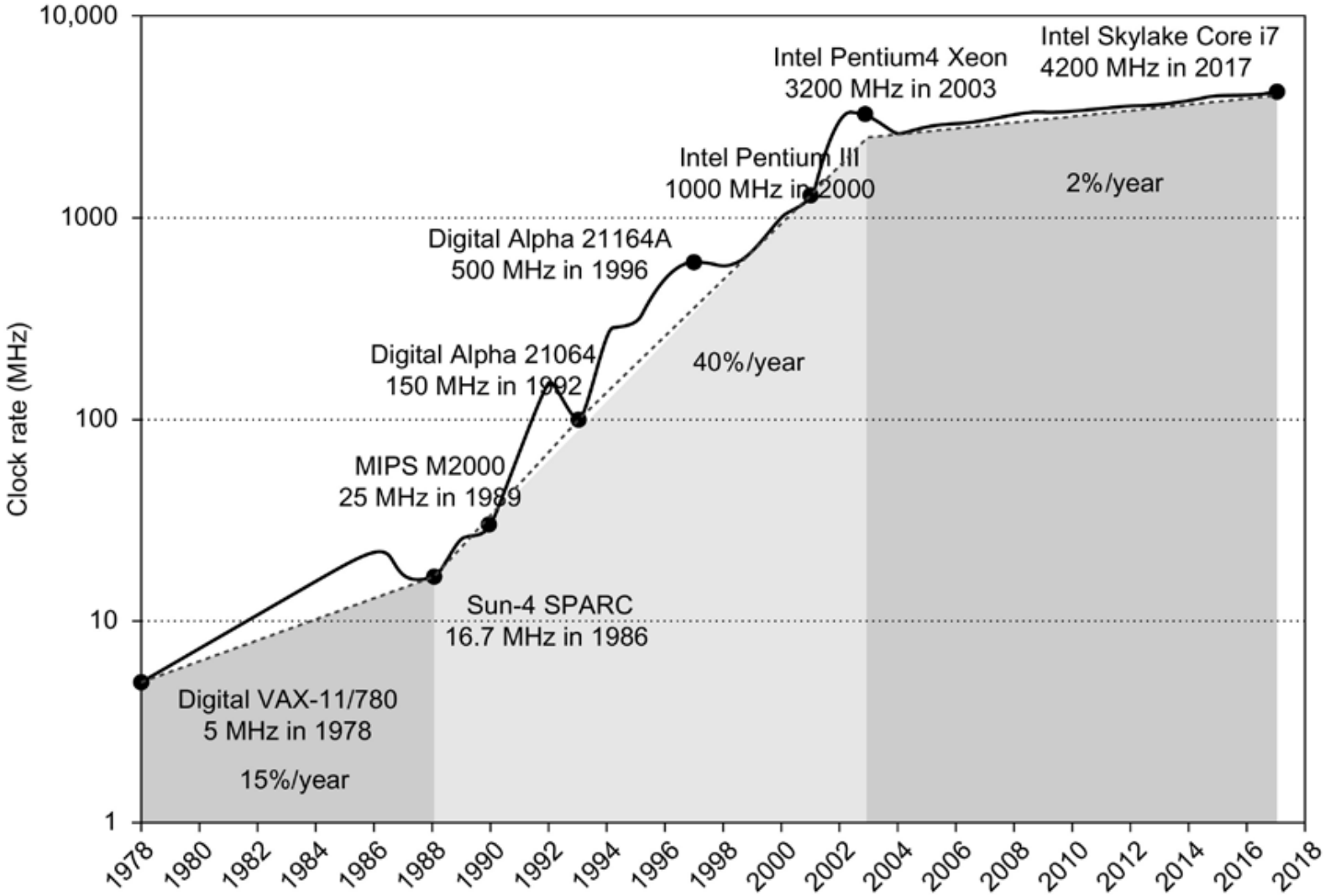
# Speed: Main Goal in General Purpose Computer Architecture

- Reduce delay per gate → **Technology** (~ x1.2/year)
- Improve architecture → **Parallelism** (~ x1.3/year)




Source: Hennessy & Patterson, © Elsevier 2019

# Clock Rate Does Not Grow Much (Anymore!)




Source: Hennessy & Patterson, © Elsevier 2019

# Sources of Parallelism

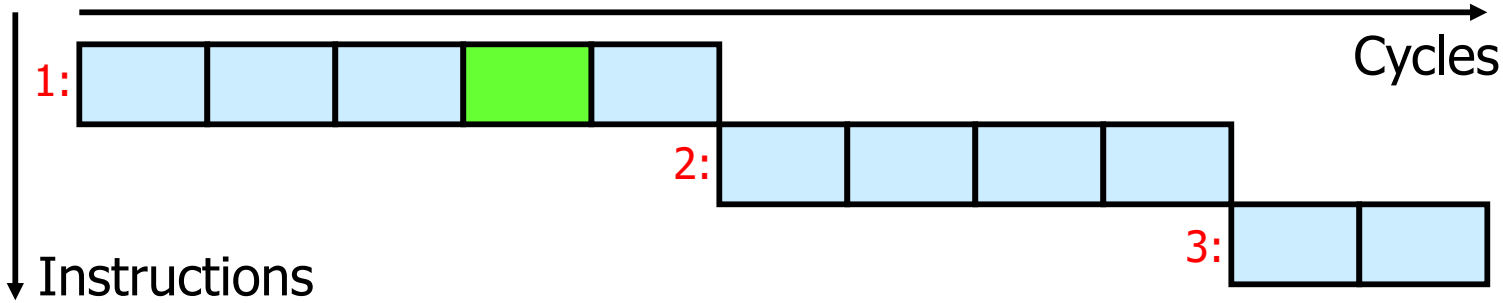
- Bit-level
  - Wider processor datapaths (8→16→32→64…)
- Word-level (SIMD)
  - Vector processors
  - Multimedia instruction sets (Intel's MMX and SSE, Sun's VIS, etc.)
- **Instruction-level**
  - **Pipelining**
  - **Superscalar**
  - **VLIW and EPIC**
- Task- and Application-levels…
  - Explicit parallel programming
  - Multiple threads
  - Multiple applications…

This lesson:

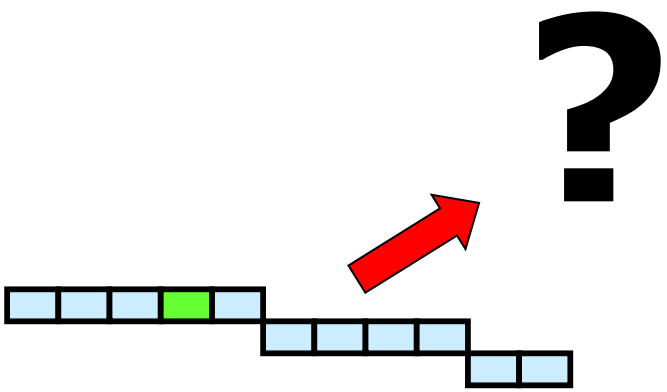
**ILP** = Instruction Level Parallelism

# Starting Point (Programmer Model)

- Sequential multicycle processor

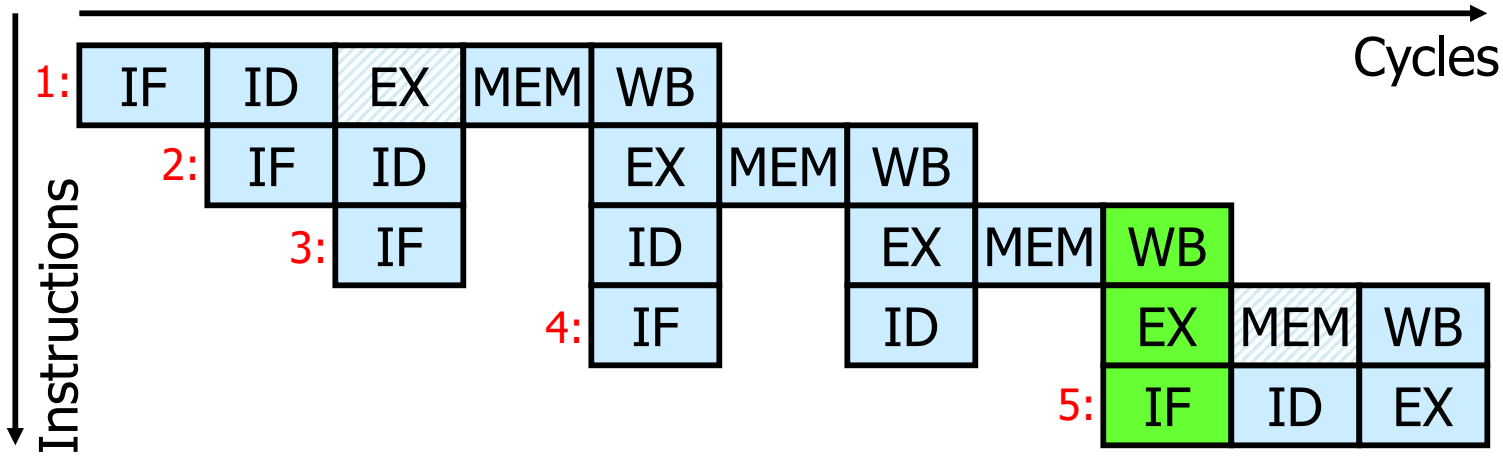


# ILP?

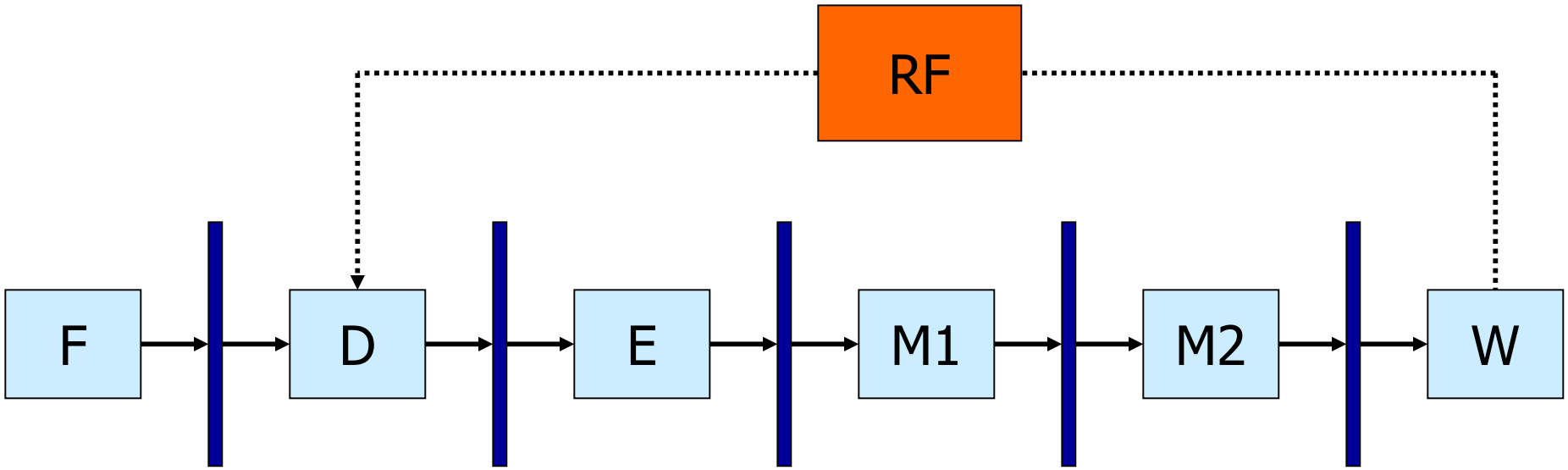

Standard

# First Step: Pipelining




- Simplest form of **Instruction Level Parallelism** (ILP): Several instructions are being executed at once

# Simple Pipeline

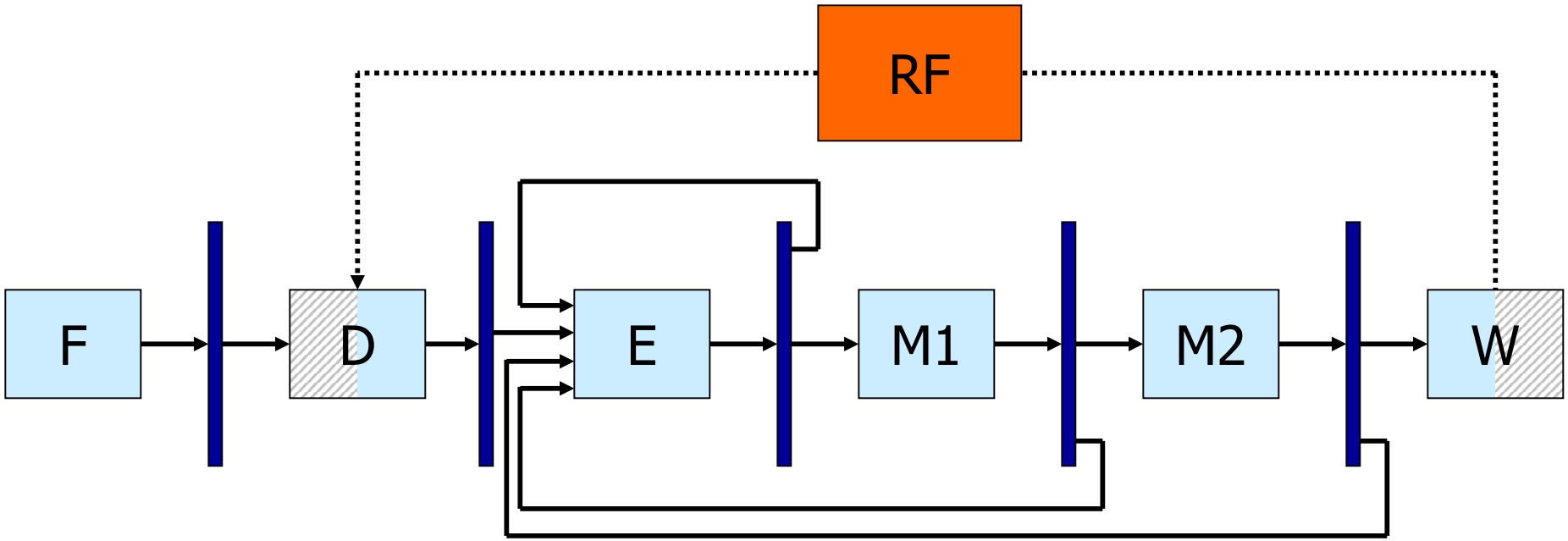


# Simple Pipelining

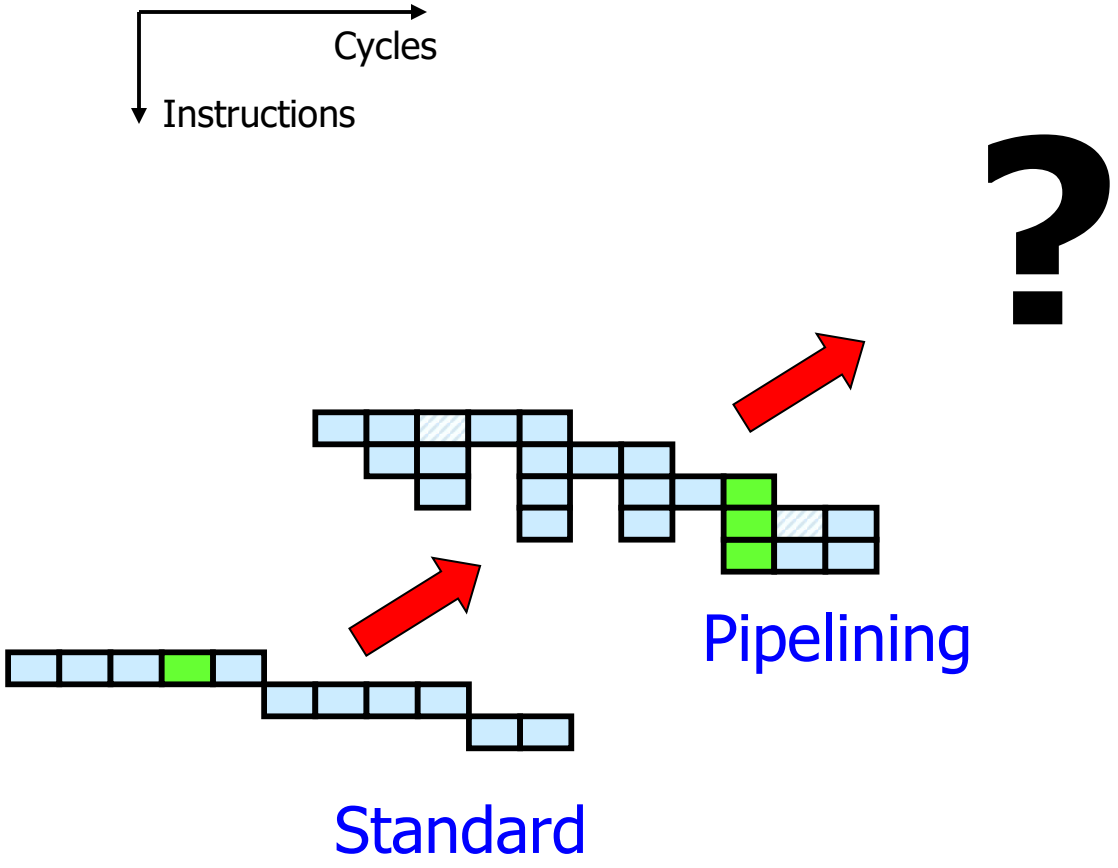
Scope for parallelism is limited:

- **Control hazards** limit the usability of the pipeline
  - Must squash fetched and decoded instruction following a branch
- **Data hazards** limit the usability of the pipeline
  - Whenever the next instruction cannot be executed, the pipeline is stalled and no new useful work is done until the "problem" is solved (e.g., cache miss)
- **Rigid sequencing**
  - Special "slots" for everything even if sometimes useless (e.g., MEM before WB)
  - Every instruction must be coerced to the same framework
  - Structural hazards avoided "by construction"

# Simple Pipeline with Forwarding



# ILP So Far…



# Dynamic Scheduling: The Idea

- Extend the scope to extract parallelism:

```
divd    $f0, $f2, $f4
addd    $f10, $f0, $f8
subd    $f12, $f8, $f14
```

- Why not to execute **`subd`** while **`addd`** waits for the result of **`divd`**?
- Relax a fundamental rule: instructions can be executed **out of program order**! (but the result must still be correct…)

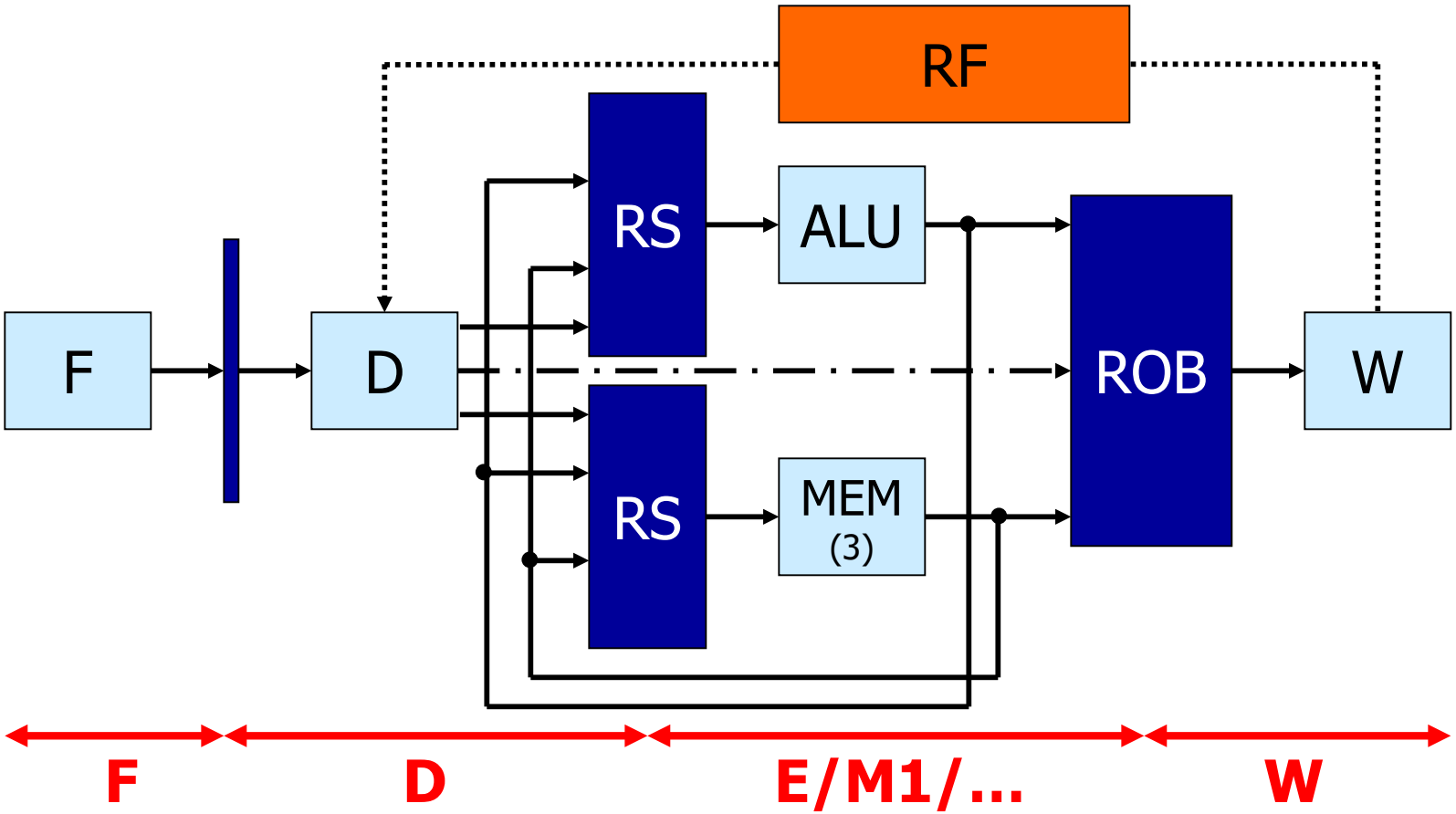
# Break the Rigidity of the Basic Pipelining

- **Continue fetching and decoding** even and especially if one cannot execute previous instructions
- **Keep writeback waiting** if there is a structural hazard, without slowing down execution

Solution:

- **Split the tasks** in independent units/pipelines
  - Fetch and decode
  - Execute
  - Writeback
- Clearly, instructions will now produce results **out-of-order (OOO)**

# Dynamically Scheduled Processor

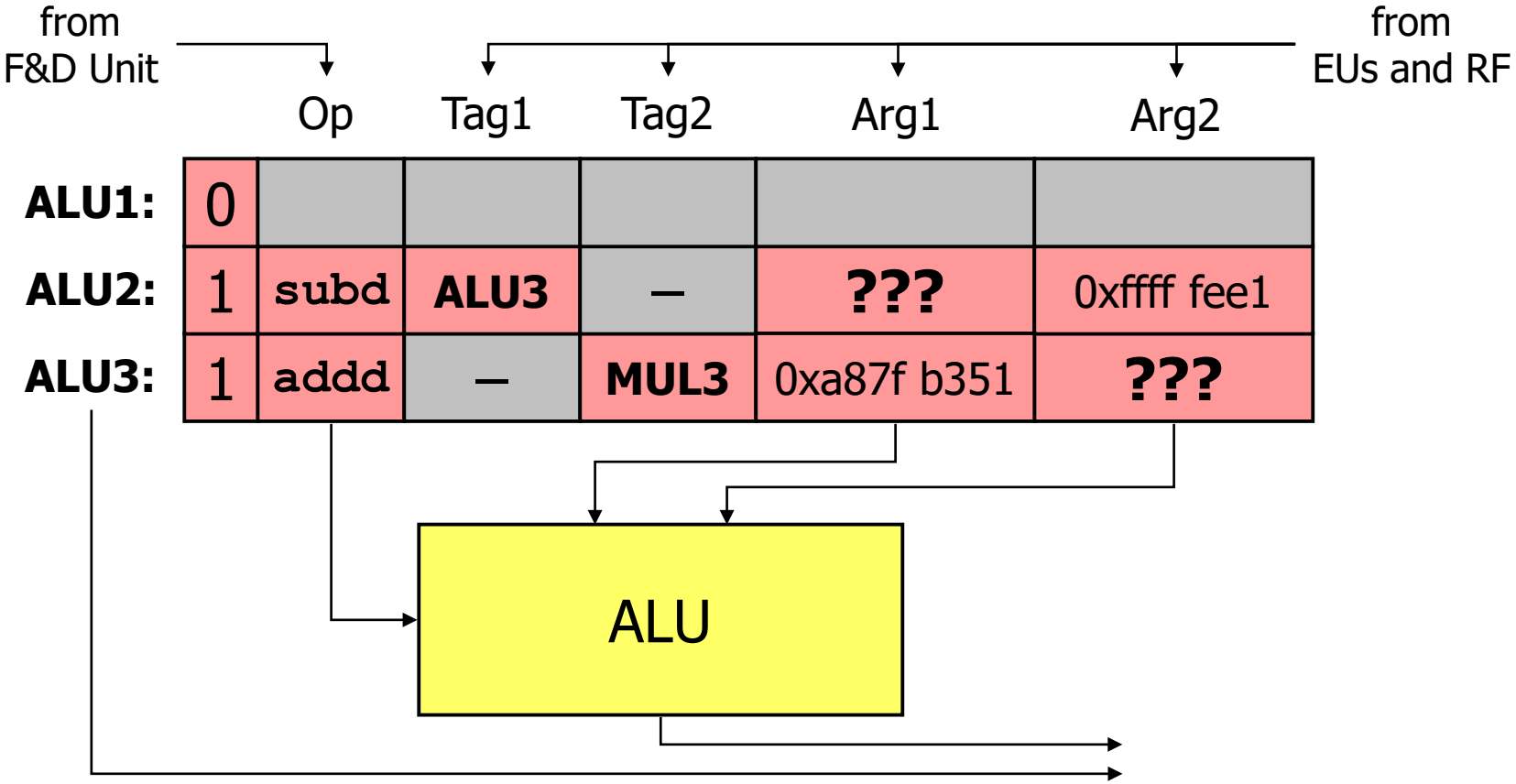


# Problems to Solve

- **Structural Hazards**
  - Are the required resources available?
  - New problem: previously handled by rigid pipeline
- **RAW Data Hazards**
  - Are the operands ready to start execution?
  - Old problem
- **WAR and WAW Data Hazards**
  - The new data overwrite something which is still required?
  - WAW is a completely new problem—impossible before; WAR often cannot occur

# Reservation Stations

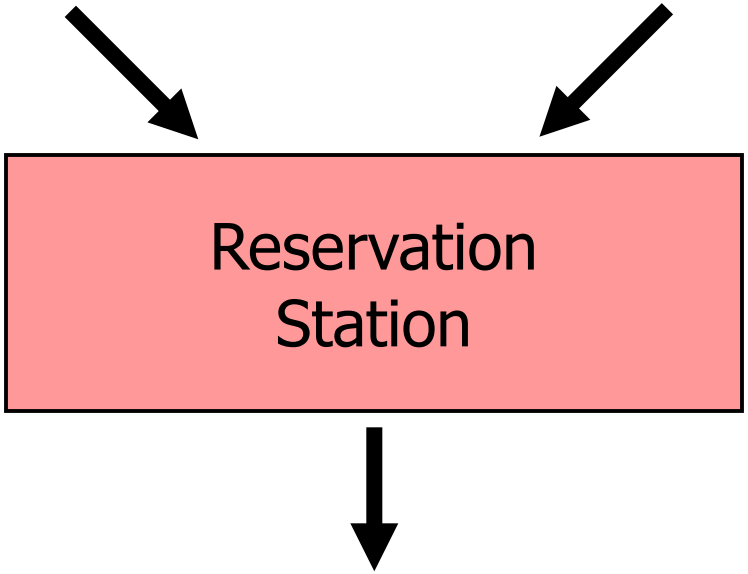
- A reservation station checks that the operands are available (RAW) and that the Execution Unit is free (Structural Hazard), then starts execution



# Reservation Stations

**Fetch&Decode Unit and Register File**
(1) Fetched operation descriptions and
(2a) known operands (from RF)
or (2b) source-operation tags

**All Execution Units**
(1) Tags of the executed operations
and (2) corresponding results



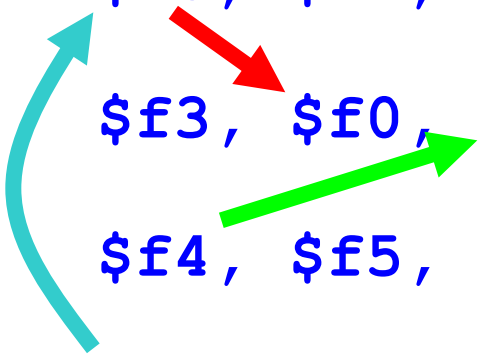

**Dependent Execution Unit**
(1) Description of operations ready to execute
with (2) corresponding tags and (3) operands

# Problems to Solve

- **Structural Hazards**
  - Are the required resources available?
  - New problem: previously handled by rigid pipeline
- **RAW Data Hazards**
  - Are the operands ready to start execution?
  - Old problem
- **WAR and WAW Data Hazards**
  - The new data overwrite something which is still required?
  - WAW is a completely new problem—impossible before; WAR often cannot occur

# WAR and WAW Data Dependences

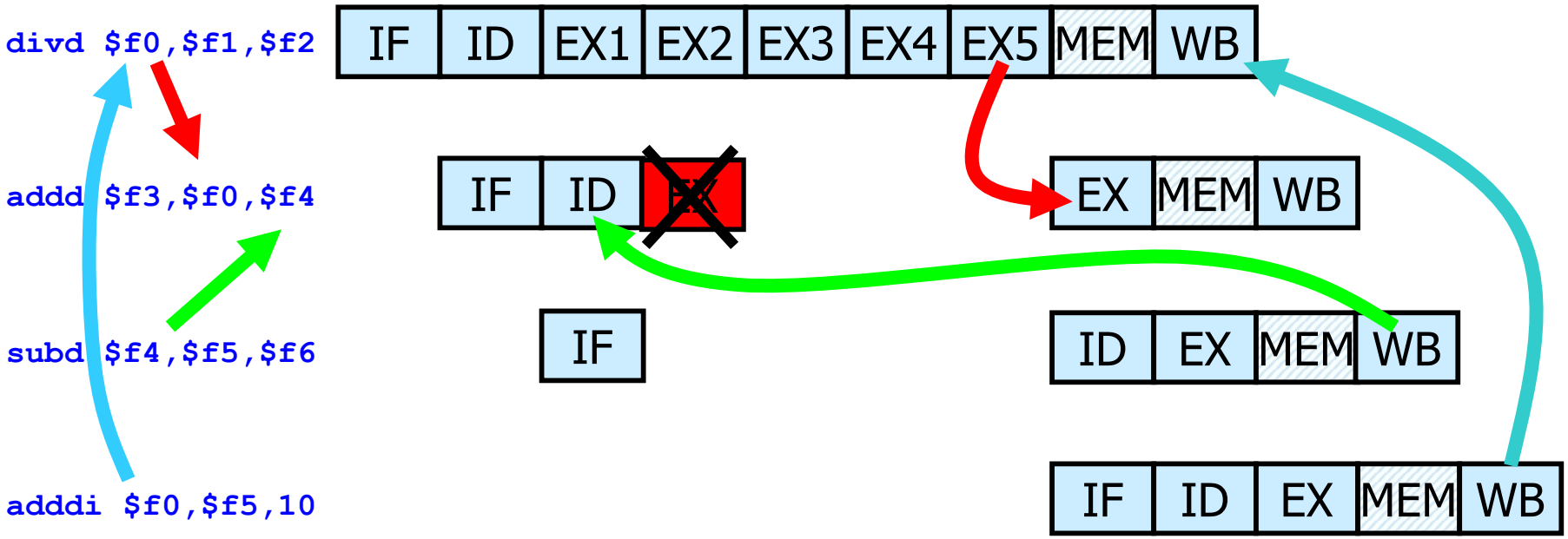
```
divd    $f0, $f1, $f2
addd    $f3, $f0, $f4
subd    $f4, $f5, $f6
adddi   $f0, $f5, 10
```



- **addd** has a RAW dependence on **divd**
- **subd** has a WAR dependence on **addd**
- **adddi** has a WAW dependence on **divd**

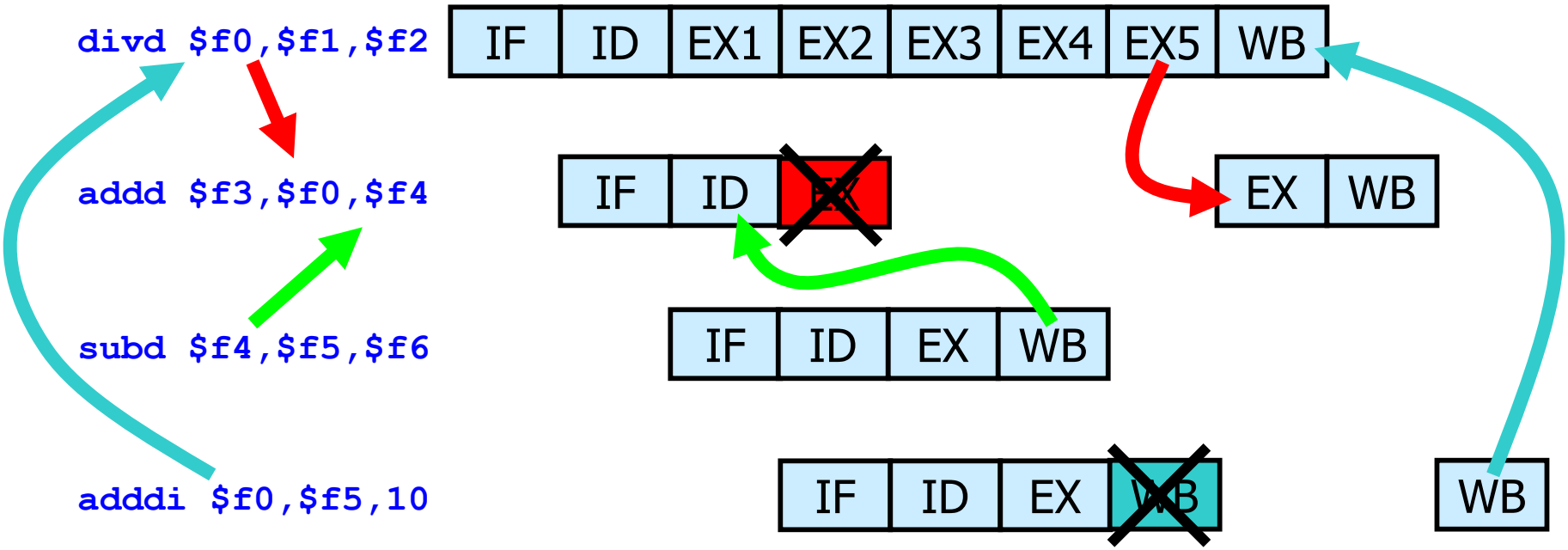
# In-order Completion

- Simple pipelines have no WAR and WAW hazards by construction



# Out-of-order Completion

- Dynamic pipelines may create WAW hazards

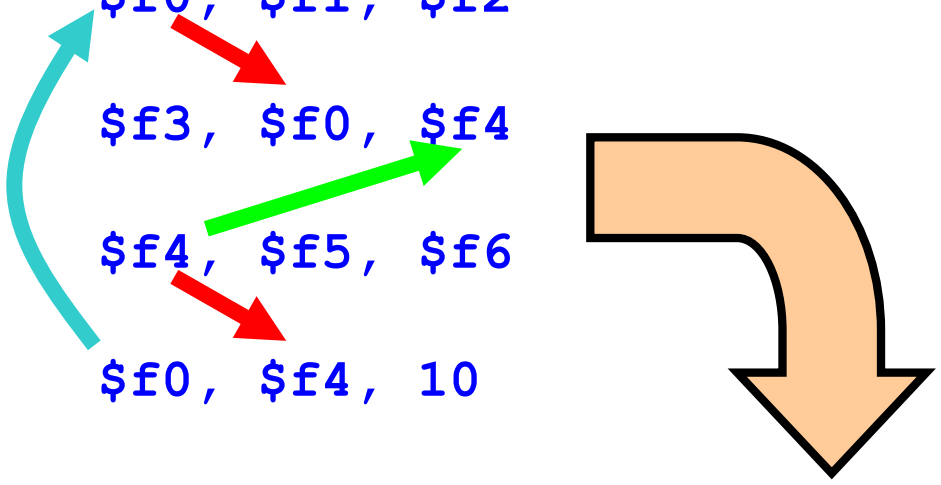


# Register Renaming

- WAW and WAR dependences are also called **name dependences**: they do not carry a value between to instructions
- Often created by compilers to reuse the same registers
- Can be removed by avoiding the use of the same "name" → **rename the destination register whenever a new value is created**
- Both the compiler (statically) and the processor (dynamically) can do that

# Register Renaming

```
divd    $f0, $f1, $f2
addd    $f3, $f0, $f4
subd    $f4, $f5, $f6
adddi   $f0, $f4, 10
```

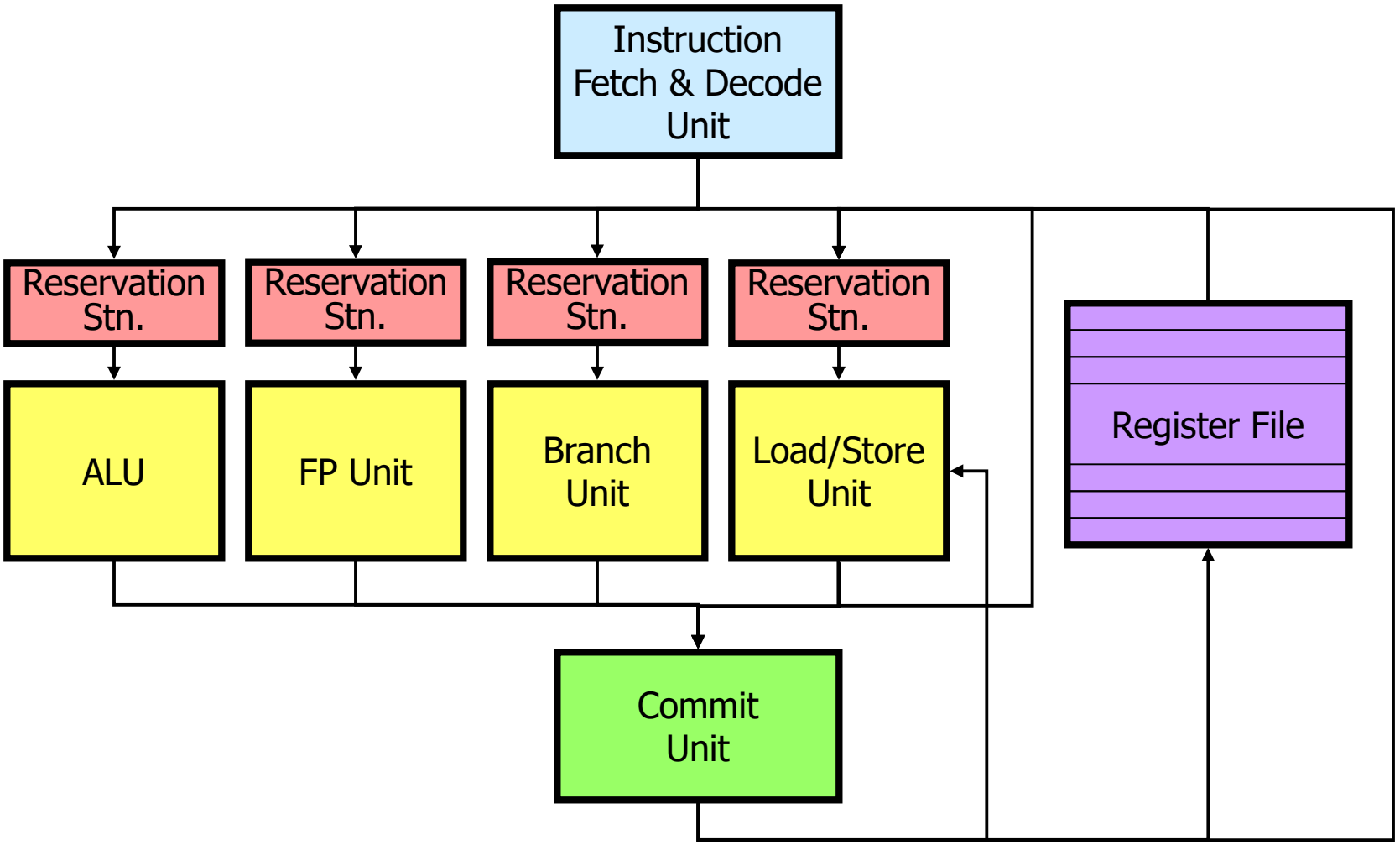


```
divd    $f0, $f1, $f2
addd    $f3, $f0, $f4
subd    $f4b, $f5, $f6
adddi   $f0b, $f4b, 10
```

# Reservation Stations

- Unavailable operands are identified by the name of the reservation station in charge of the originating instruction
- **Implicit register renaming**, thus removing WAR and WAW hazards
- New results are seen at their inputs through special result bus(es)
- Writeback into the registers can be done in-order or out-of-order

# Dynamically Scheduled Processor



# Out-of-order Commitment and Exceptions

- Exception handlers should know exactly where a problem has occurred, especially for **nonterminating exceptions** (e.g., page fault) so that they handle the event and resume exactly where the exception occurred
- Of course, one assumes that everything before the faulty instruction was executed and everything after was not
- With OOO dynamic execution it might no longer be true…

# A Problem with Exceptions…

- **Precise exceptions**
  - Reordering at commit; user view is that of a fully in-order processor
- **Imprecise exceptions**
  - No reordering; out-of-order completion visible to the user
  - The OS/programmer must be aware of the problem and take appropriate action (e.g., execute again the complete subroutine where the problem occurred)

**Precise**

```
    andi    $t4, $t2, 0xff
    andi    $t5, $t4, 0xff
    addi    $v0, $t5, 1
    srl     $t2, $t2, 8
→   lw      $t3, 8($t6)
    andi    $t4, $t3, 3
    addi    $t0, $t0, 4
    addi    $t1, $t1, 4
```

**Imprecise**

```
    andi    $t4, $t2, 0xff
    andi    $t5, $t4, 0xff
    addi    $v0, $t5, 1
    srl     $t2, $t2, 8
→   lw      $t3, 8($t6)
    andi    $t4, $t3, 3
    addi    $t0, $t0, 4
    addi    $t1, $t1, 4
```

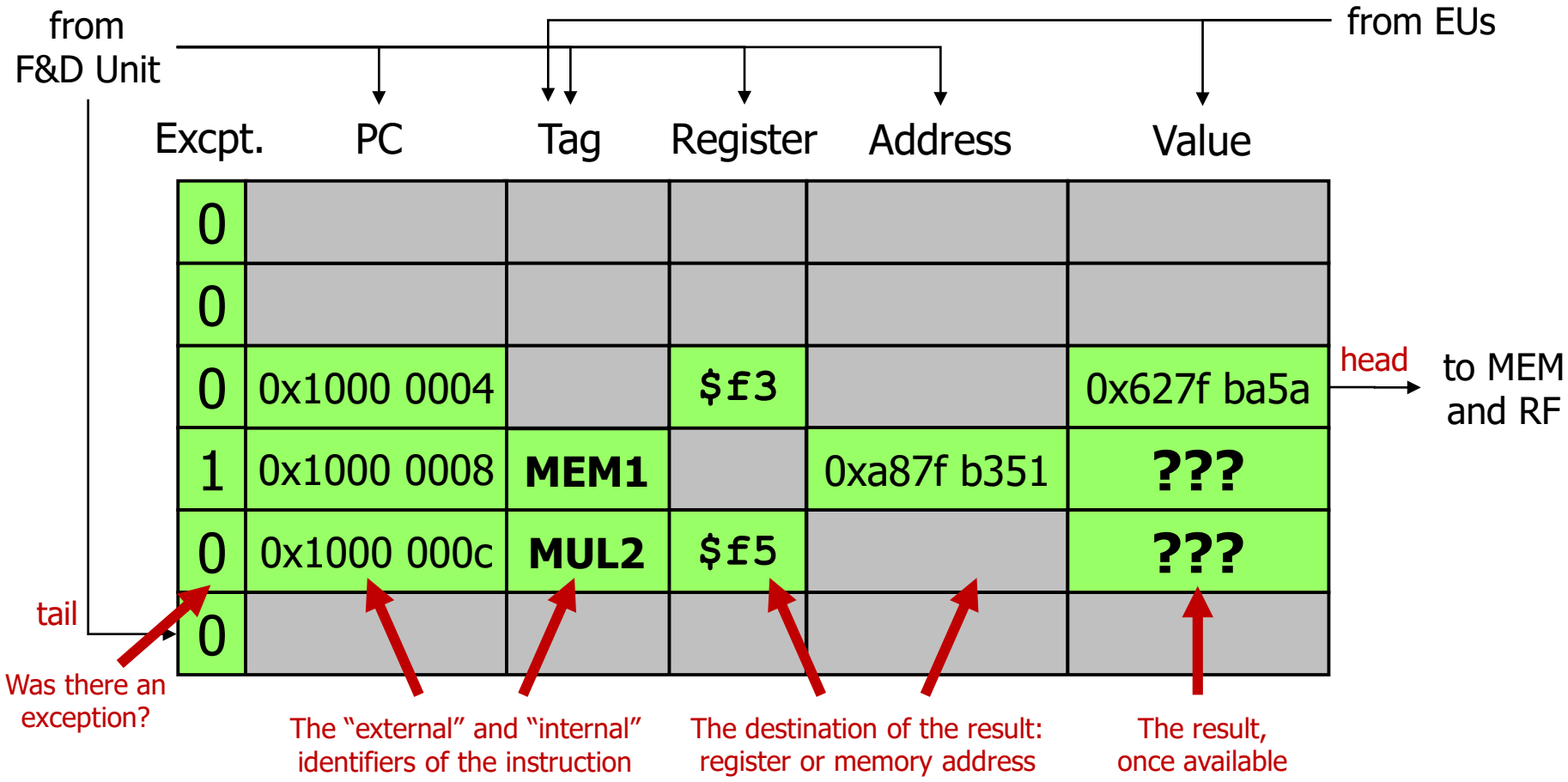
Generally unacceptable in contemporary systems (e.g., virtual memory, I/O interrupts, unsupported instructions)

# Reordering

- **Fundamental observation**: a processor can do whatever it wants provided that it gives the appearance of sequential execution (i.e., the architectural machine state is updated in program order)
- New phase: COMMIT or RETIRE or GRADUATE (besides the usual F, D, E, W)
- This observation is fundamental because it allows many techniques (precise interrupts, speculation, multithreading, etc.)

# Reordering Instructions at Writeback

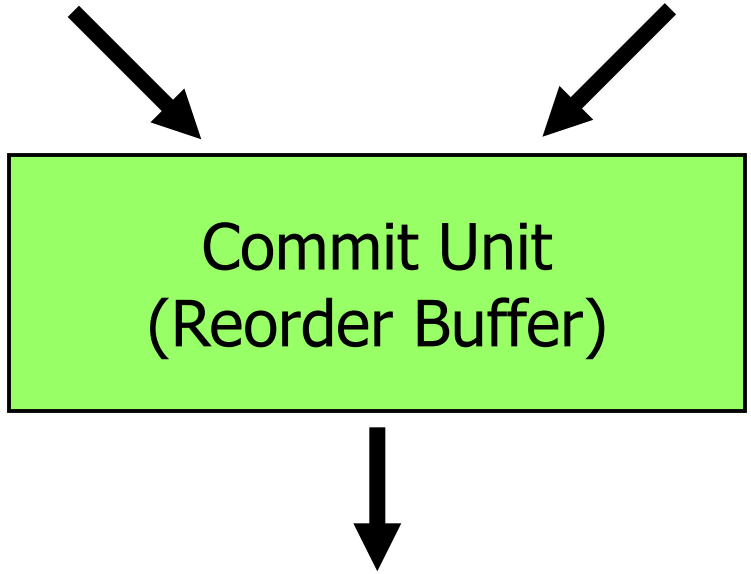
- Needs a reorder buffer in the Commit Unit



# Reorder Buffer

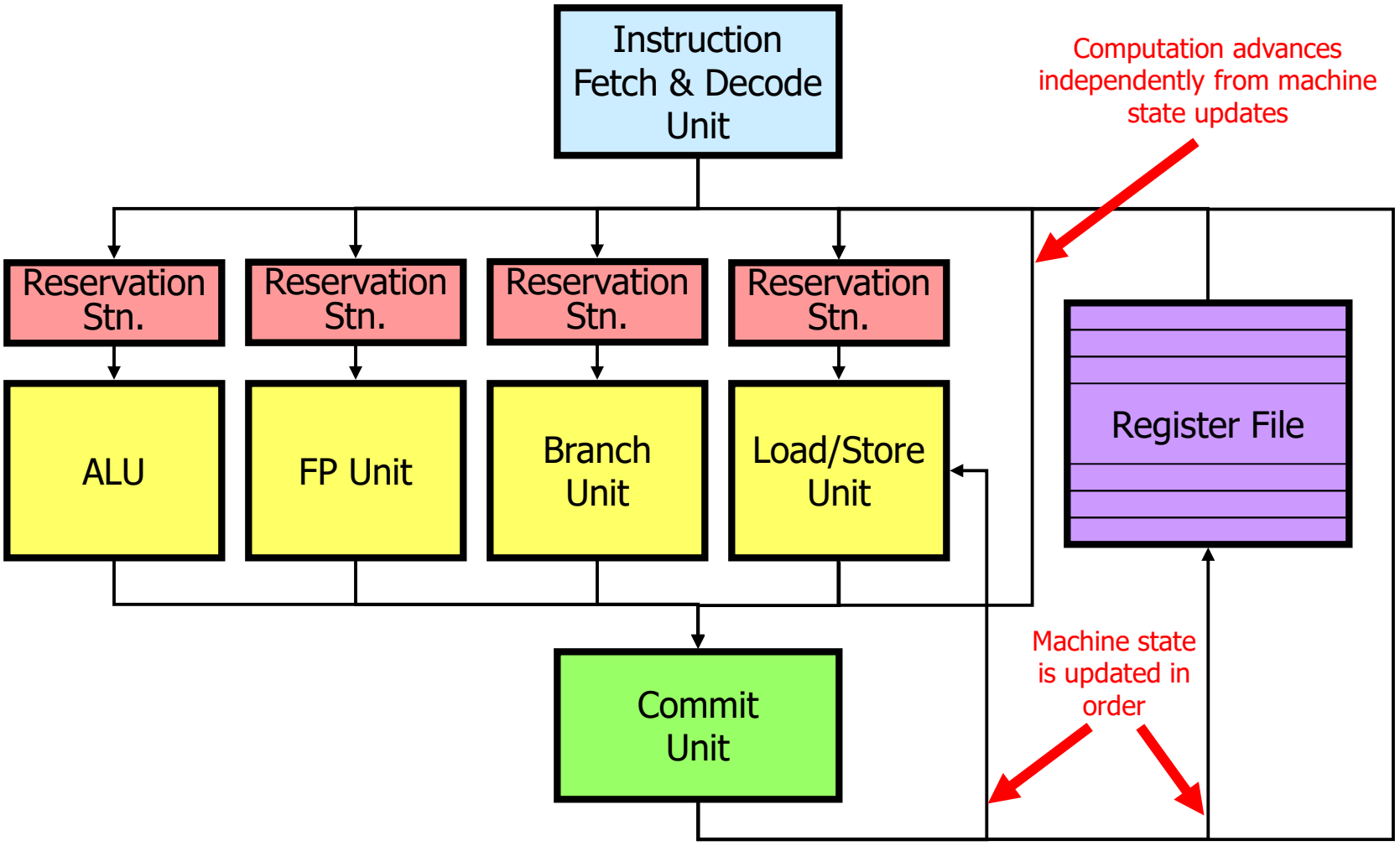
**Fetch&Decode Unit**
(1) Fetched-operation tags in original order, (2) destination register or address, and (3) PC

**All Execution Units**
(1) Tags of the executed operations and (2) corresponding results




**Register File and Memory**
For each instruction, in the original fetch order,
(1) destination register or address and (2) value to write

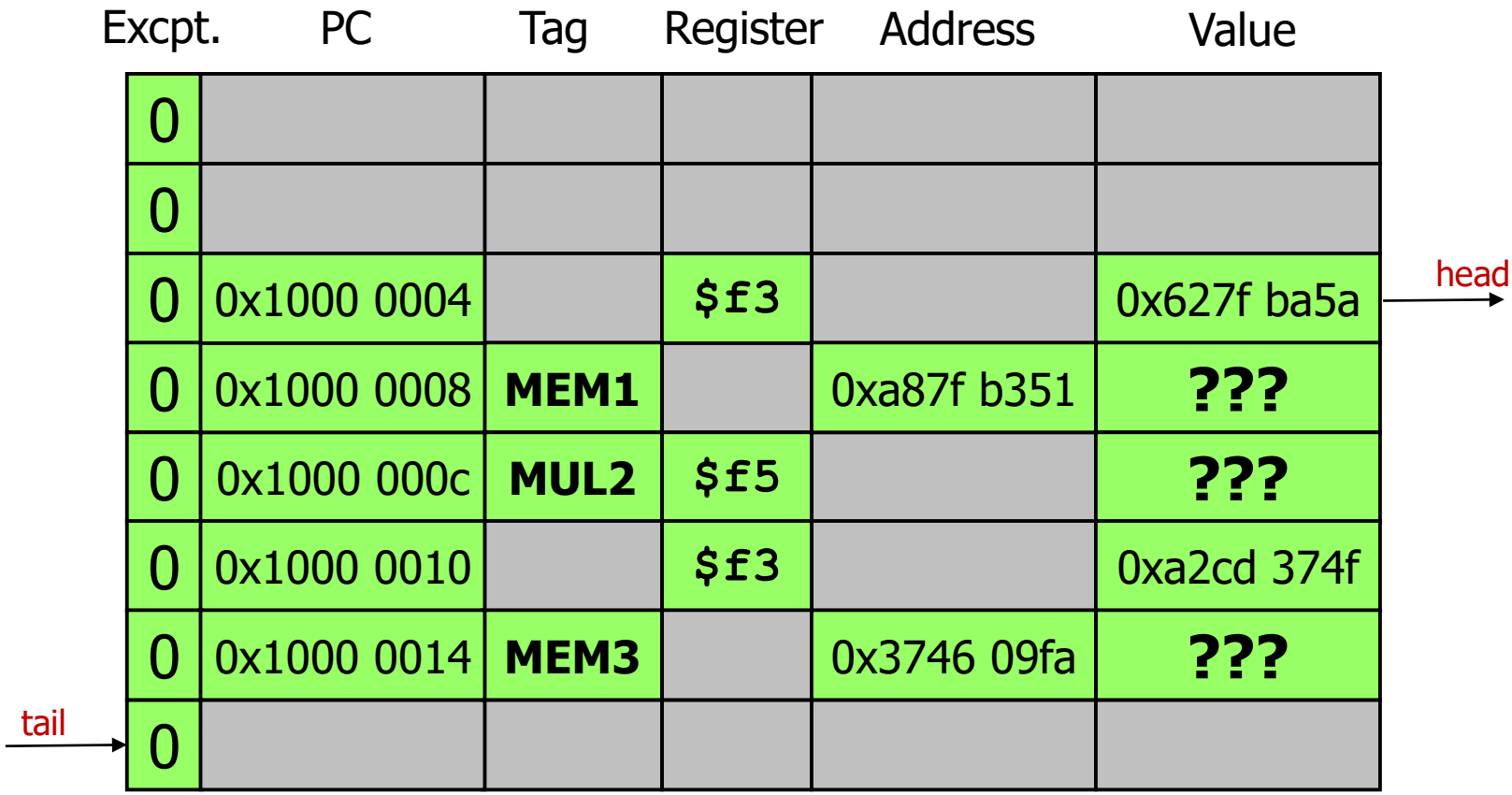
# Dynamically Scheduled Processor



# Problems to Solve

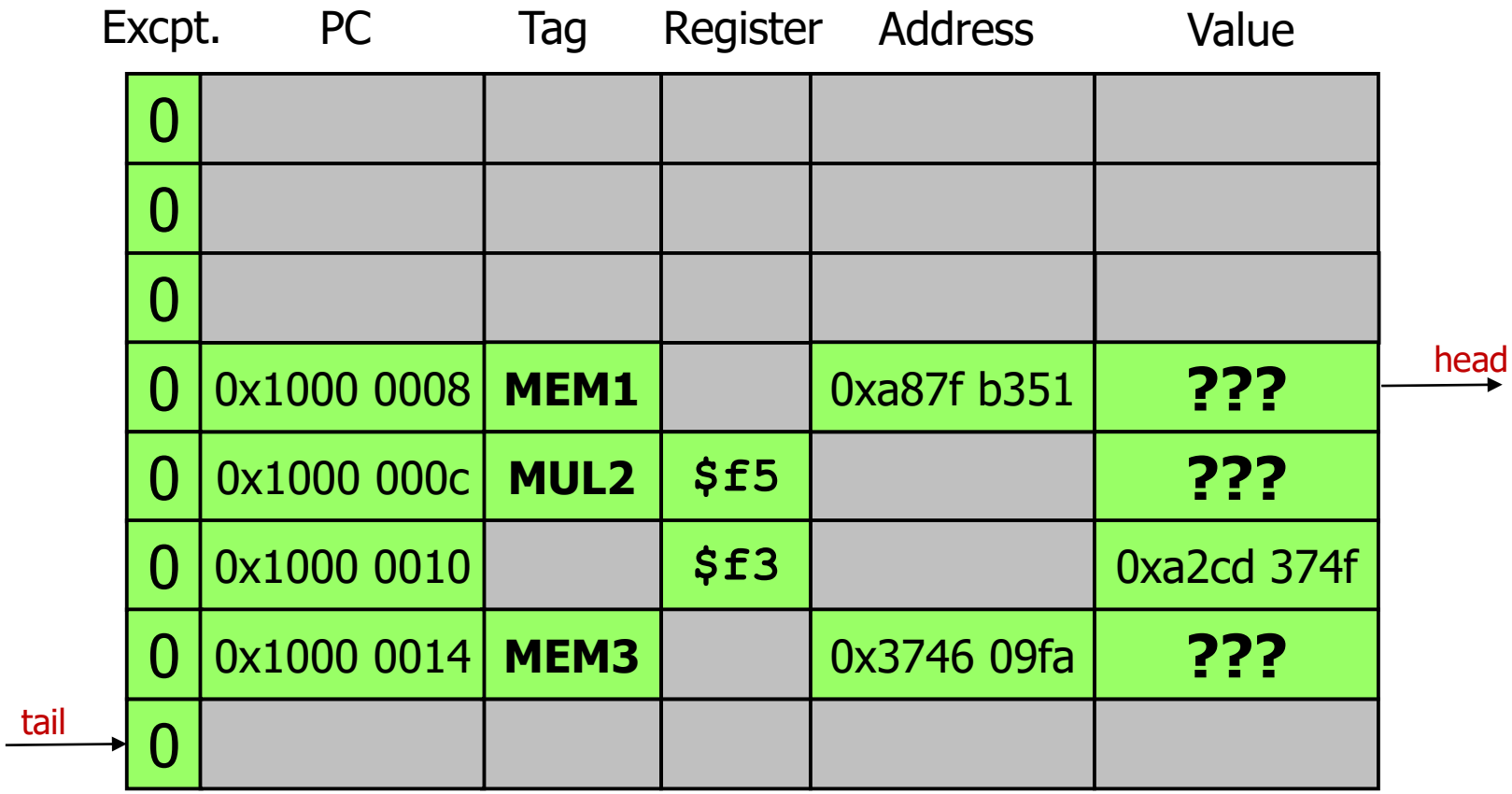
- **Structural Hazards**
  - Are the required resources available?
  - New problem: previously handled by rigid pipeline
- **RAW Data Hazards**
  - Are the operands ready to start execution?
  - Old problem
- **WAR and WAW Data Hazards**
  - The new data overwrite something which is still required?
  - WAW is a completely new problem—impossible before; WAR often cannot occur

# Committing Instructions (1/4)



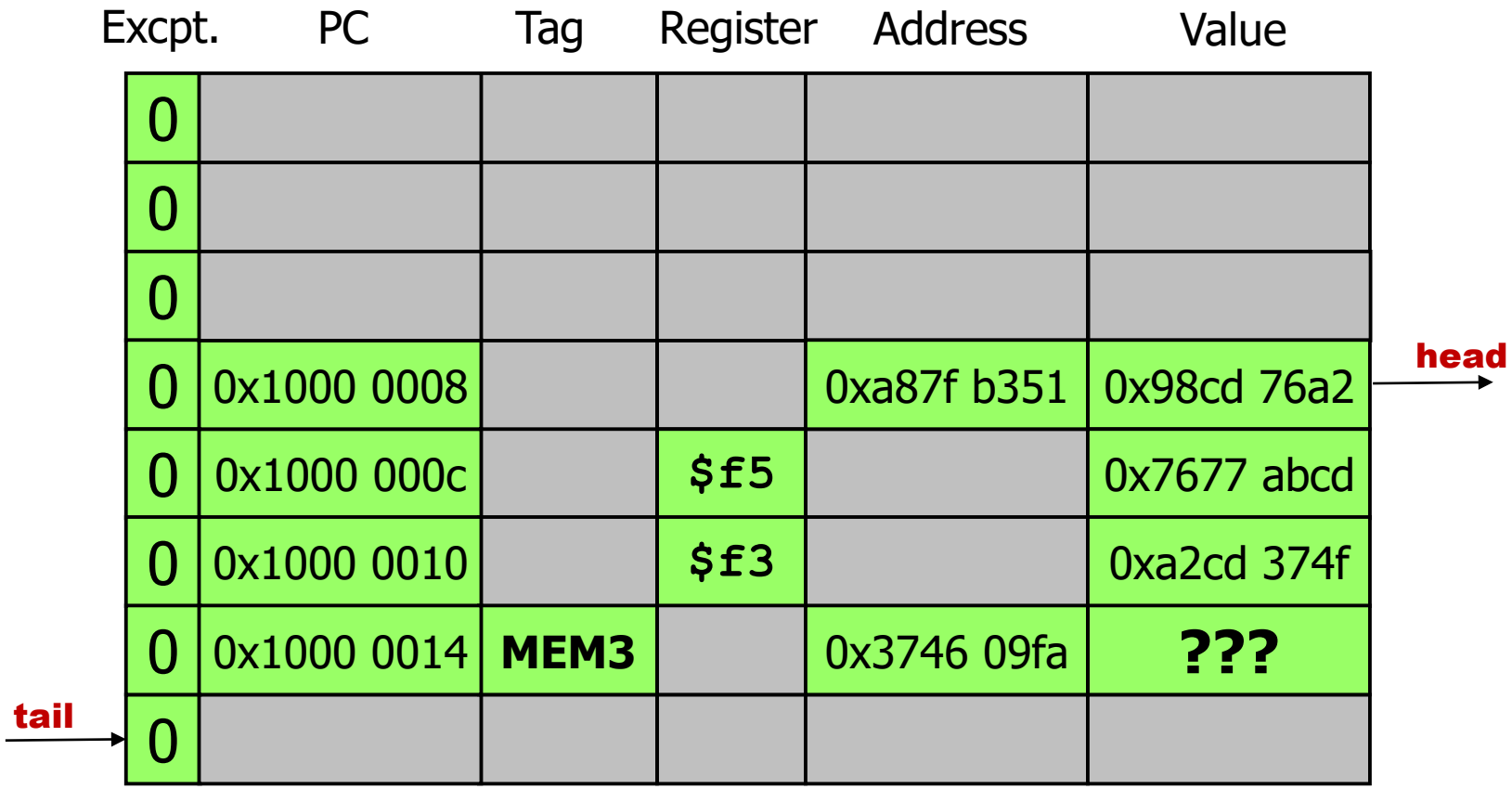

Write `0x627fba5a` to register `$f3`

# Committing Instructions (2/4)



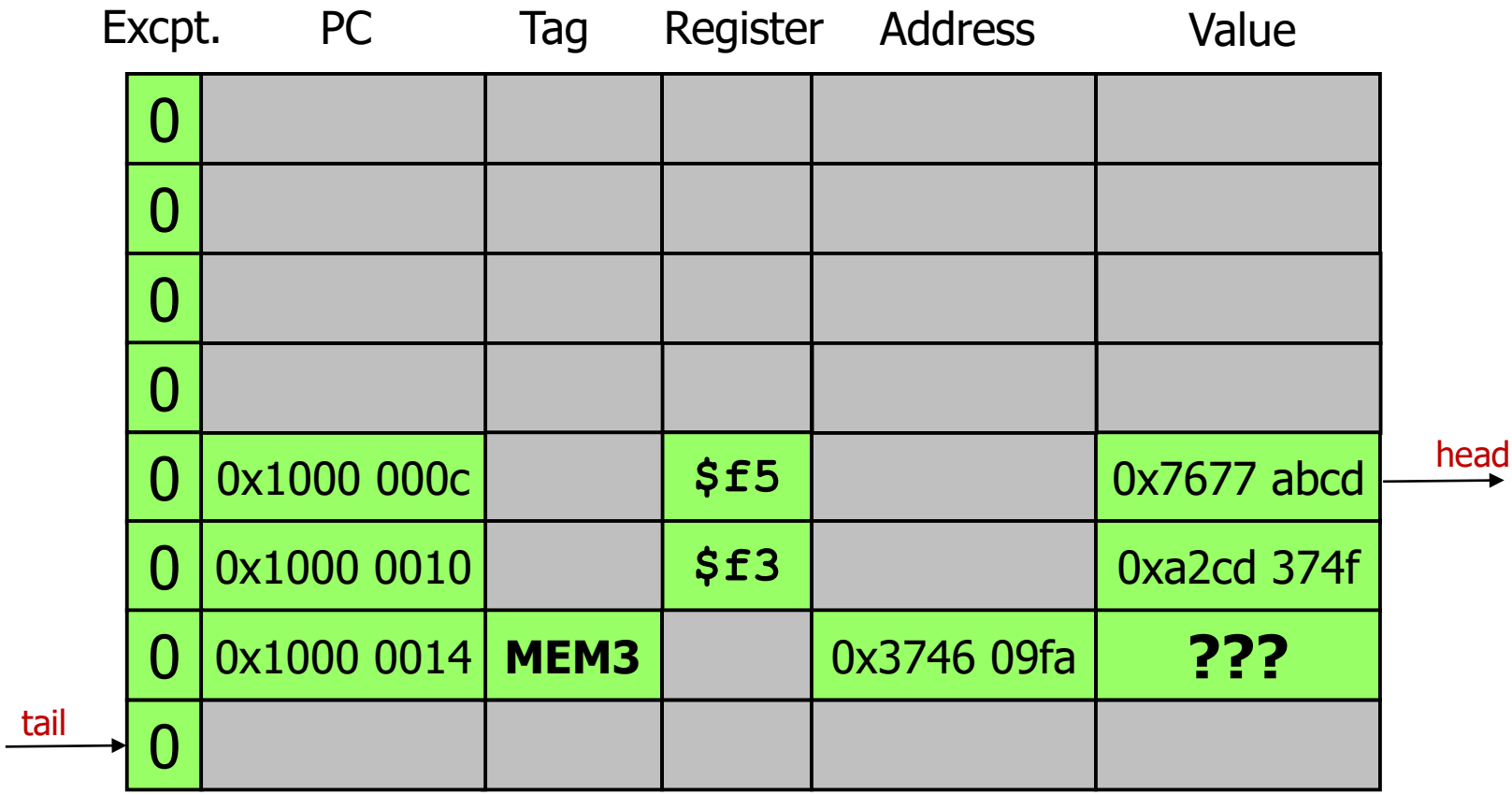

**Wait** until the oldest instruction has its result

# Committing Instructions (3/4)




Write `0x98cd76a2` to memory location `0xa87fb351`

# Committing Instructions (4/4)



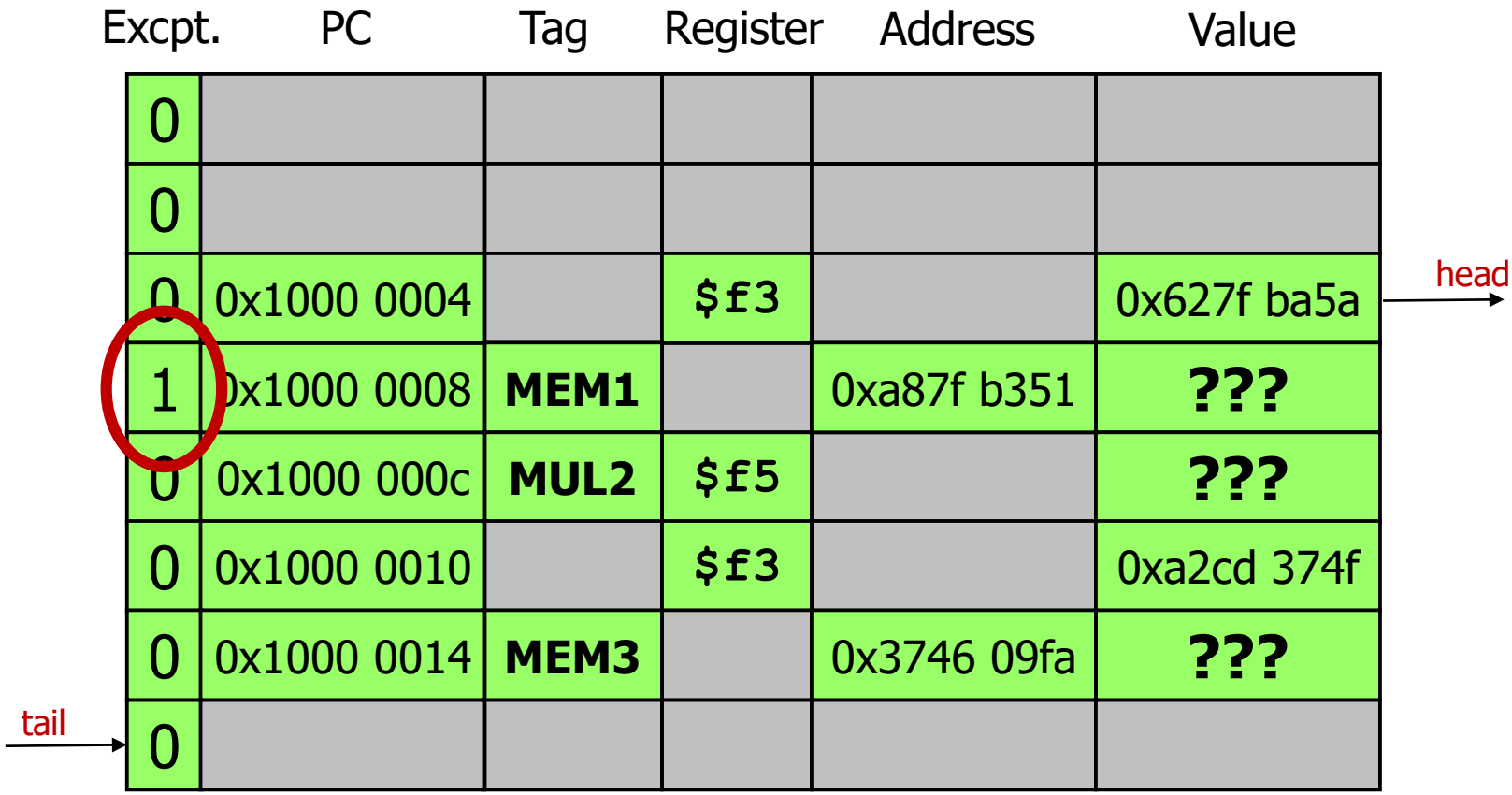

Write `0xa2cd374f` to register `$f5`

# Reordering and Precise Exceptions

How does this help with exceptions?

- When a synchronous exception happens, we do not report it but we **mark the entry** corresponding to the instruction which caused the exception in the ROB
- When we would be ready to **commit** the instruction, we **raise the exception** instead
- We also **trash** the content of the ROB and of all RSs

# Reporting Exceptions (1/4)



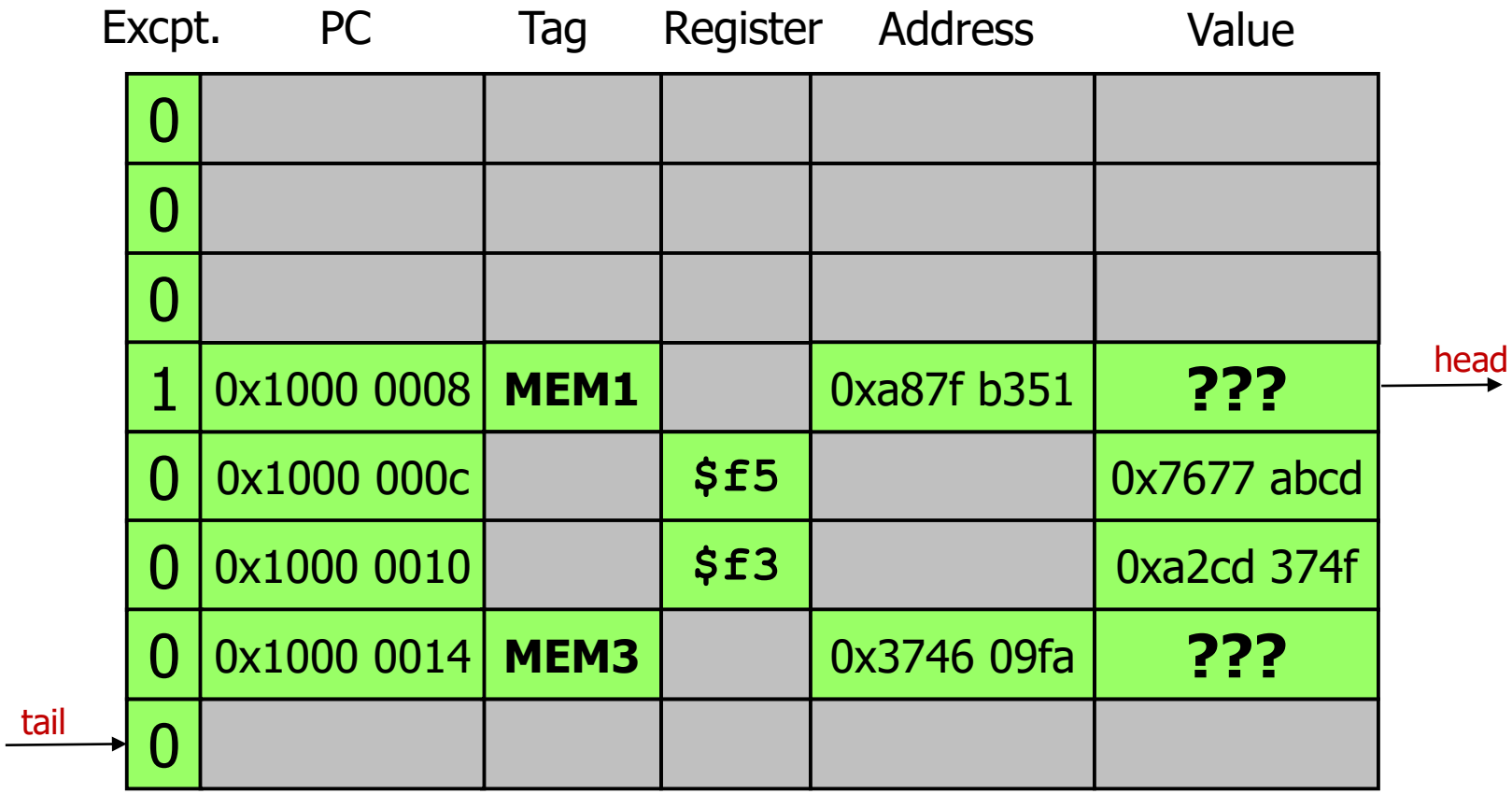

The store **MEM1** results in a *TLB Miss* → We simply **record it**

# Reporting Exceptions (2/4)

| | Excpt. | PC | Tag | Register | Address | Value |
|---|---|---|---|---|---|---|
| | 0 | | | | | |
| | 0 | | | | | |
| head → | 0 | 0x1000 0004 | | `$f3` | | 0x627f ba5a |
| | 1 | 0x1000 0008 | **MEM1** | | 0xa87f b351 | **???** |
| | 0 | 0x1000 000c | **MUL2** | `$f5` | | **???** |
| | 0 | 0x1000 0010 | | `$f3` | | 0xa2cd 374f |
| | 0 | 0x1000 0014 | **MEM3** | | 0x3746 09fa | **???** |
| tail → | 0 | | | | | |

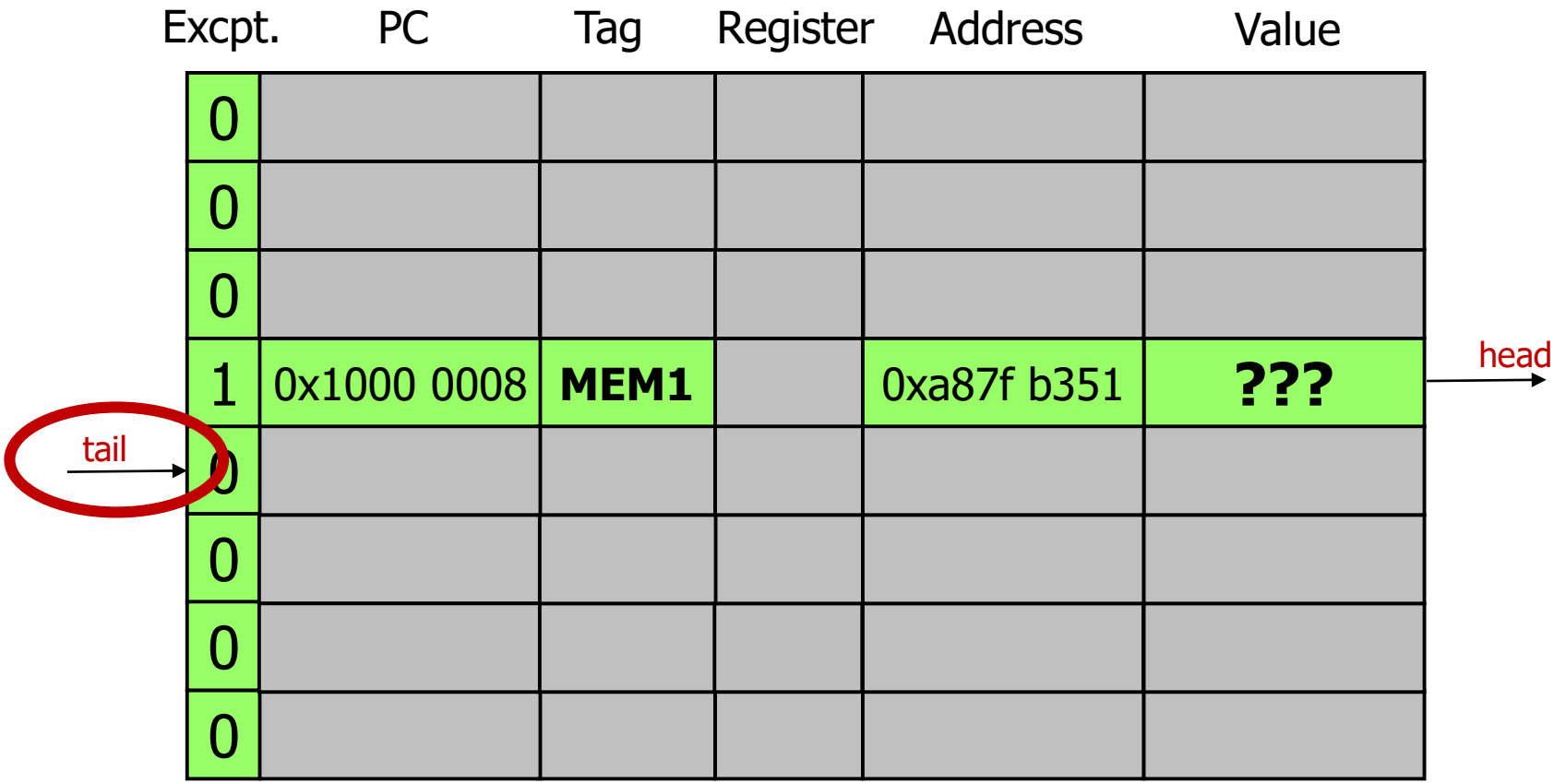
Write `0x627fba5a` to register `$f3` as if nothing happened

# Reporting Exceptions (3/4)




**Now** raise the *TLB Miss* exception at location `0x10000008`

# Reporting Exceptions (4/4)

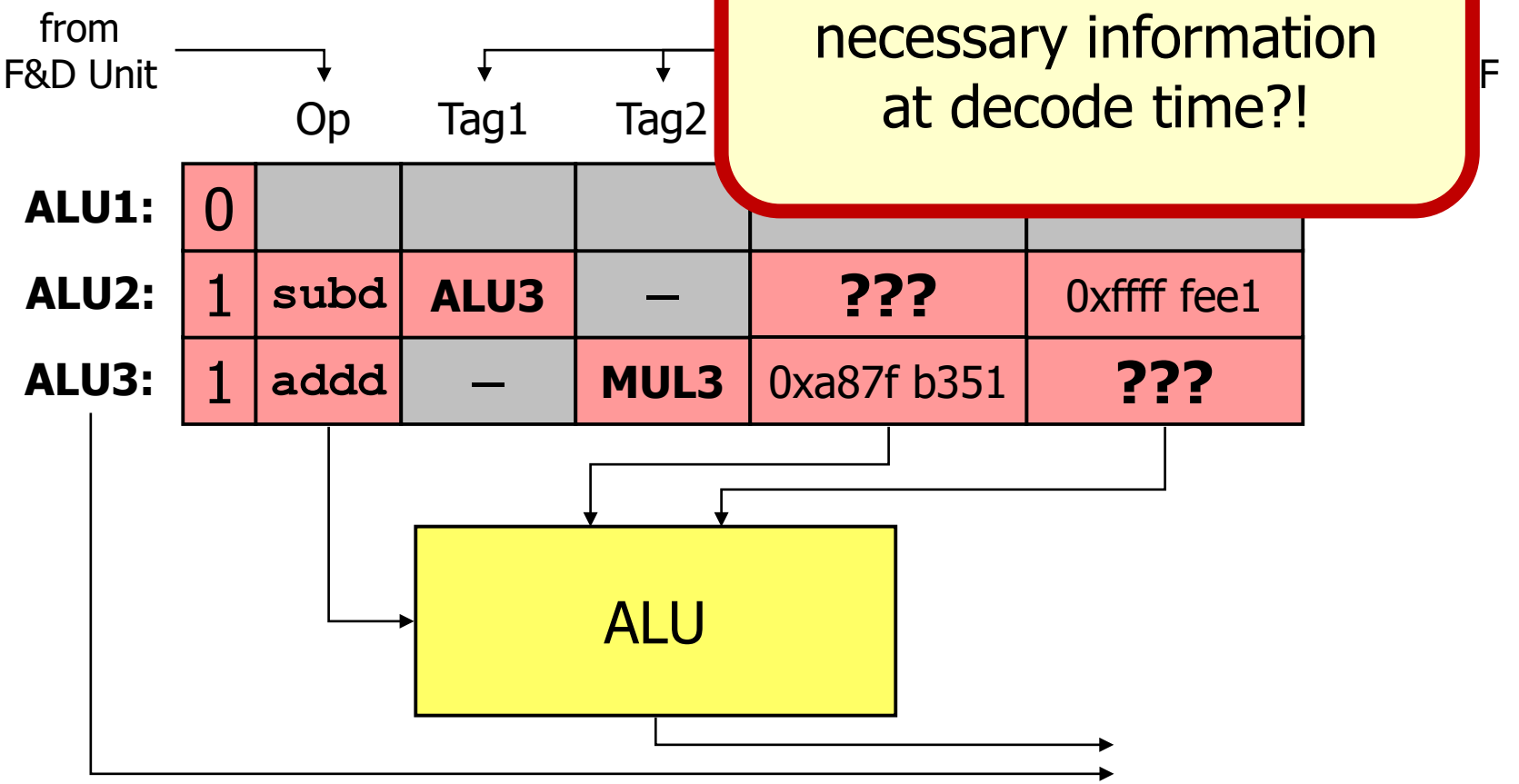



But also **squash** all instructions which followed the exception

# Reservation Stations

- A reservation station checks that the operands are available (RAW) and that the Execution Unit is free (Structural

Where do we get the necessary information at decode time?!



# Decoding and Dependences

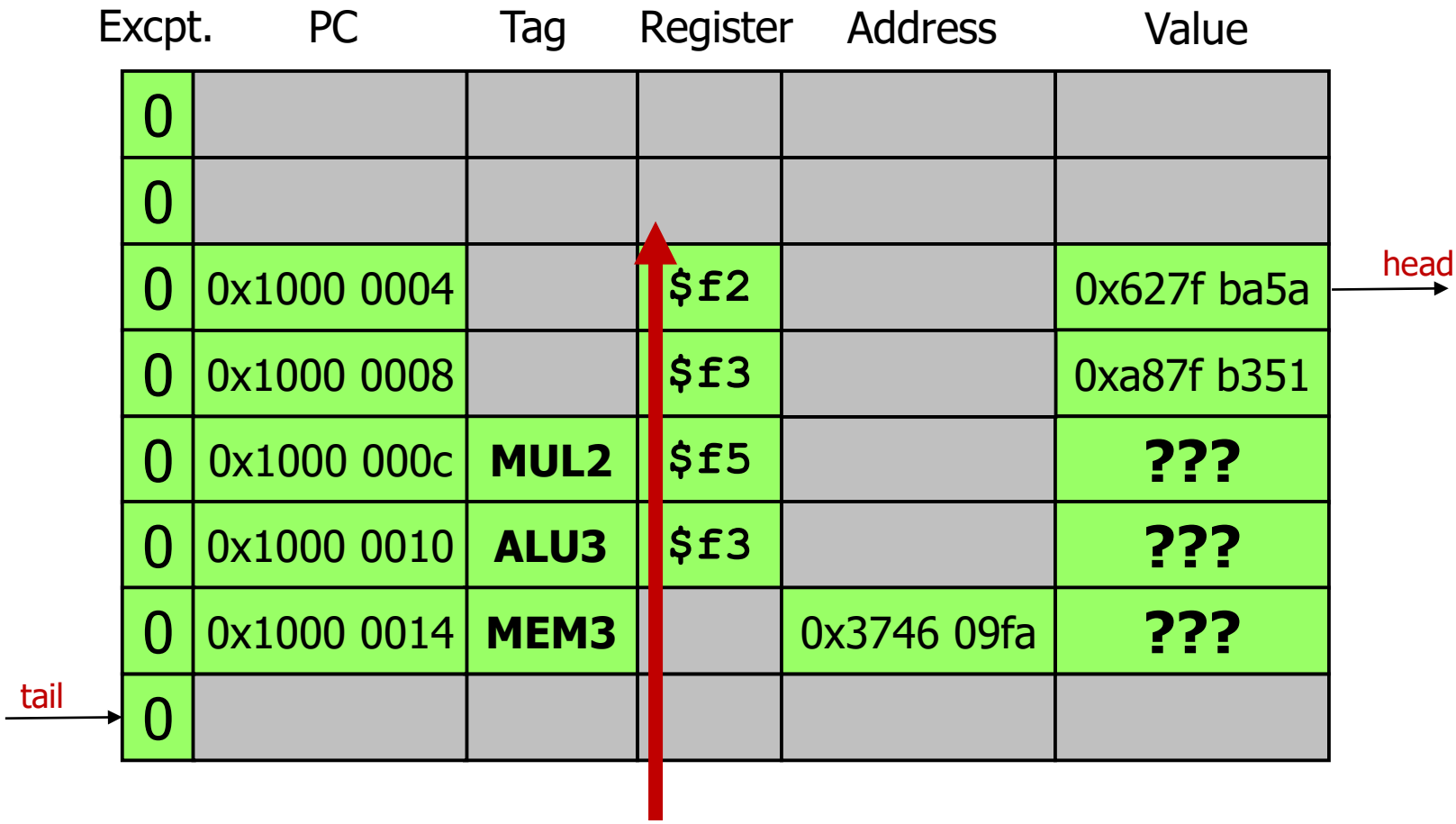
When decoding an instruction, we are supposed to put, for each operand, either a tag or a value in the corresponding reservation station—but how do we know if we can read the register file, for instance?!

The **Reorder Buffer** knows of all instructions not yet committed and of their destination registers

Possible situations:

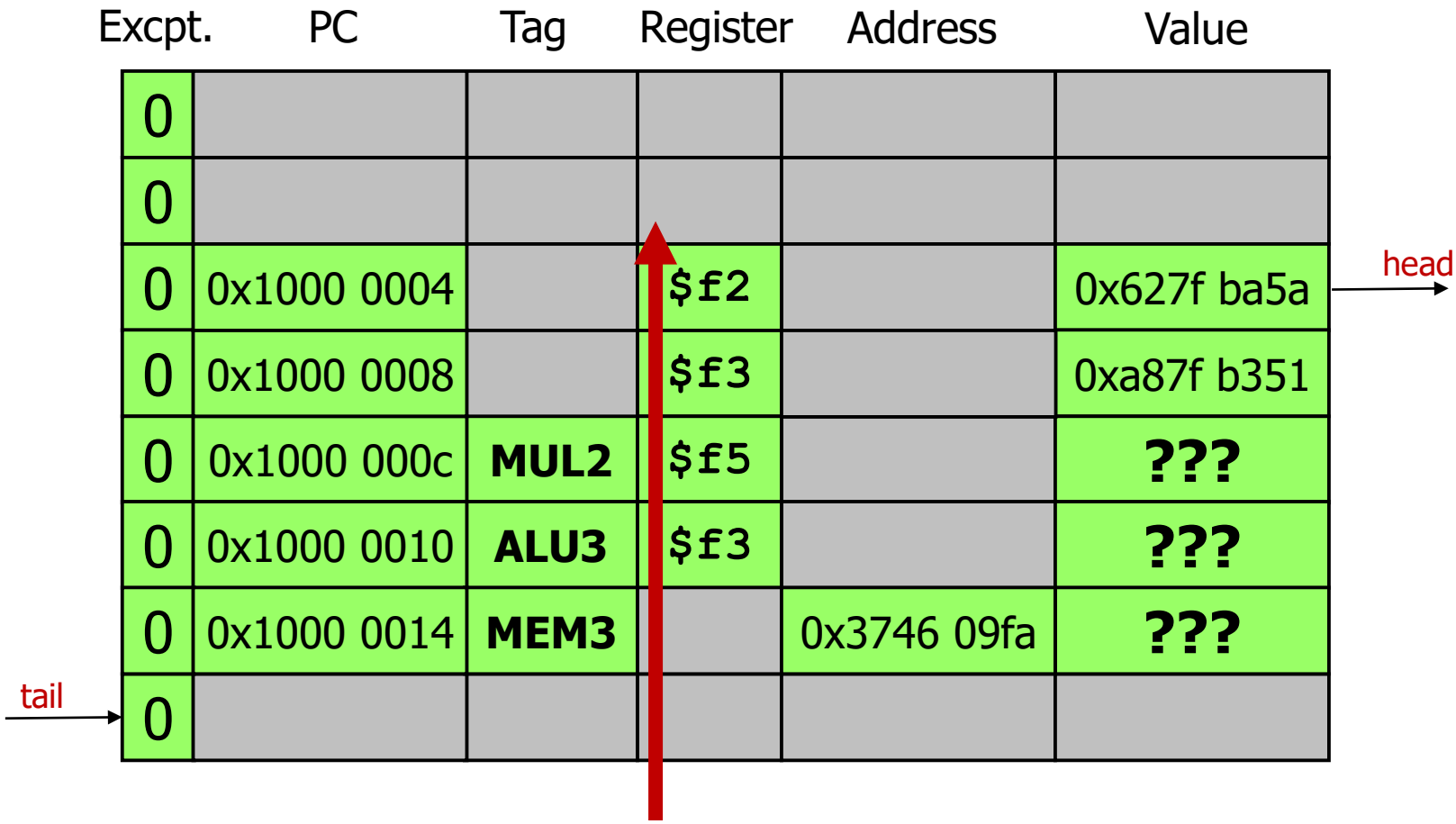
- **No dependence** → Read the **value** from the **RF**
- **Dependence** from an ongoing instruction
  - If the value is computed → Get the **value** from the **ROB**
  - If the value is not yet computed → Get the **tag** from the **ROB**

# No Dependence



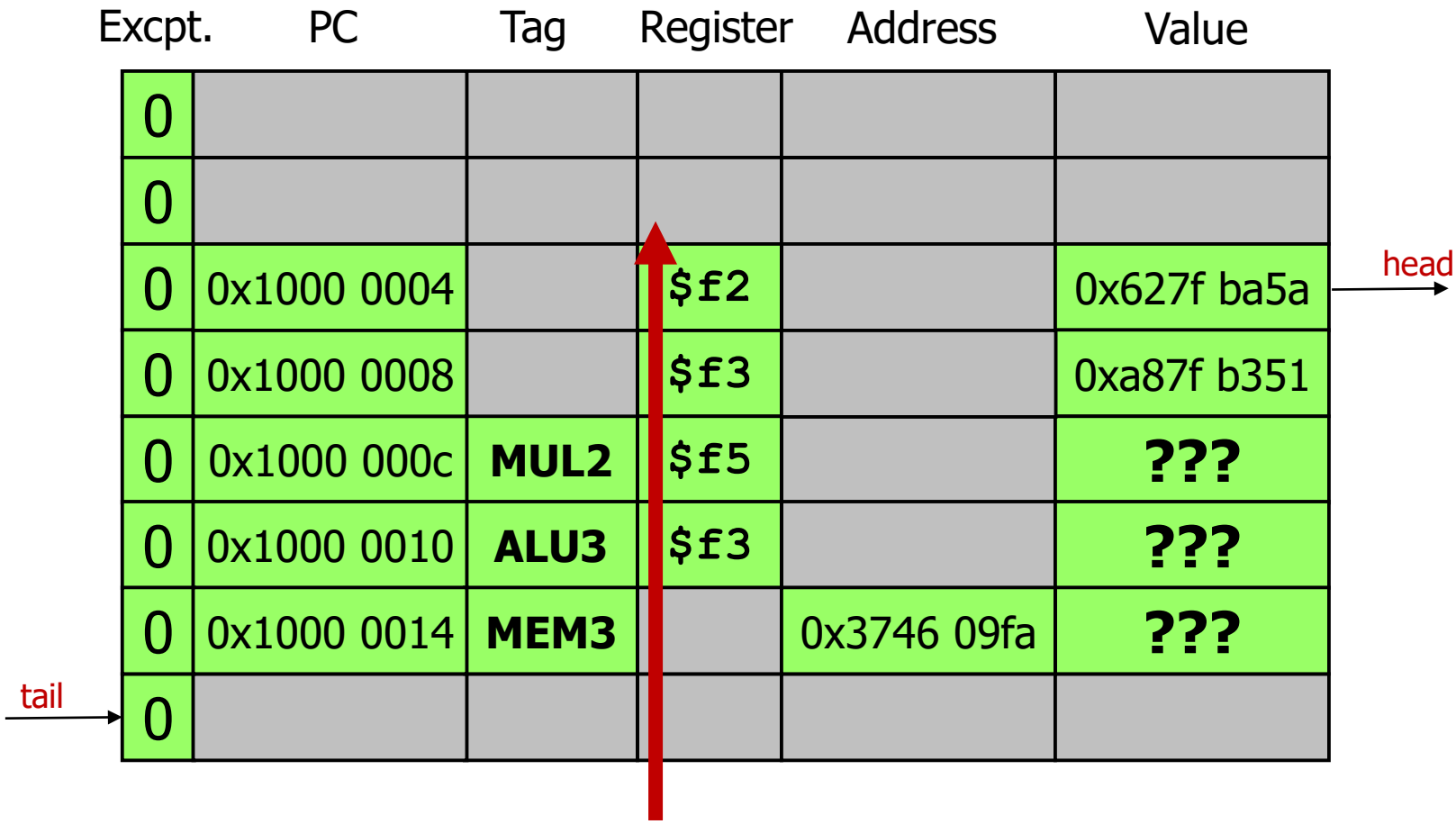

Looking for `$f1`? No ongoing instruction will produce it, hence it is safe to read it from the Register File

# Dependence and Value in the ROB



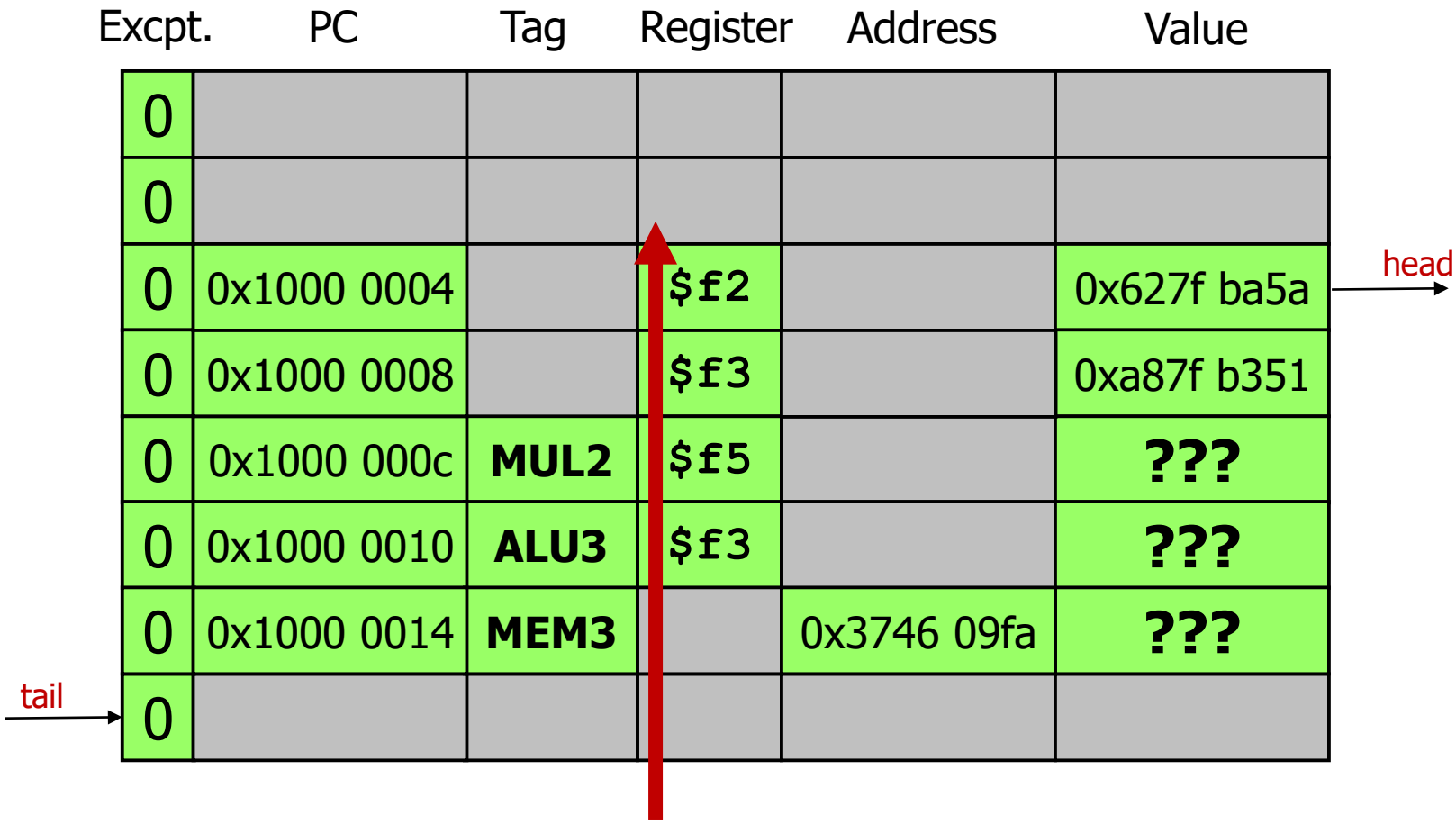

Looking for `$f2`? An ongoing instruction has produced it, hence we should read `0x627fba5a` from the ROB

# Dependence and Tag in the ROB




Looking for `$f5`? An ongoing instruction will produce it, hence we need to use tag **MUL2** as found in the ROB

# Multiple Dependences?




Looking for `$f3`? Two ongoing instructions produce it and it is the **most recent** one which matters (tag **ALU3** here)

# Dependences through Memory

The way we detect and resolve dependences through memory (a store at some address and a subsequent load from the same address) is the same as for registers

For every **load**, check the ROB:

a) If there is **no store to the same address** in the ROB, get the value from memory (i.e., from the cache)

b) If there is a **store to the same address** in the ROB, either get the value (if ready) or the tag

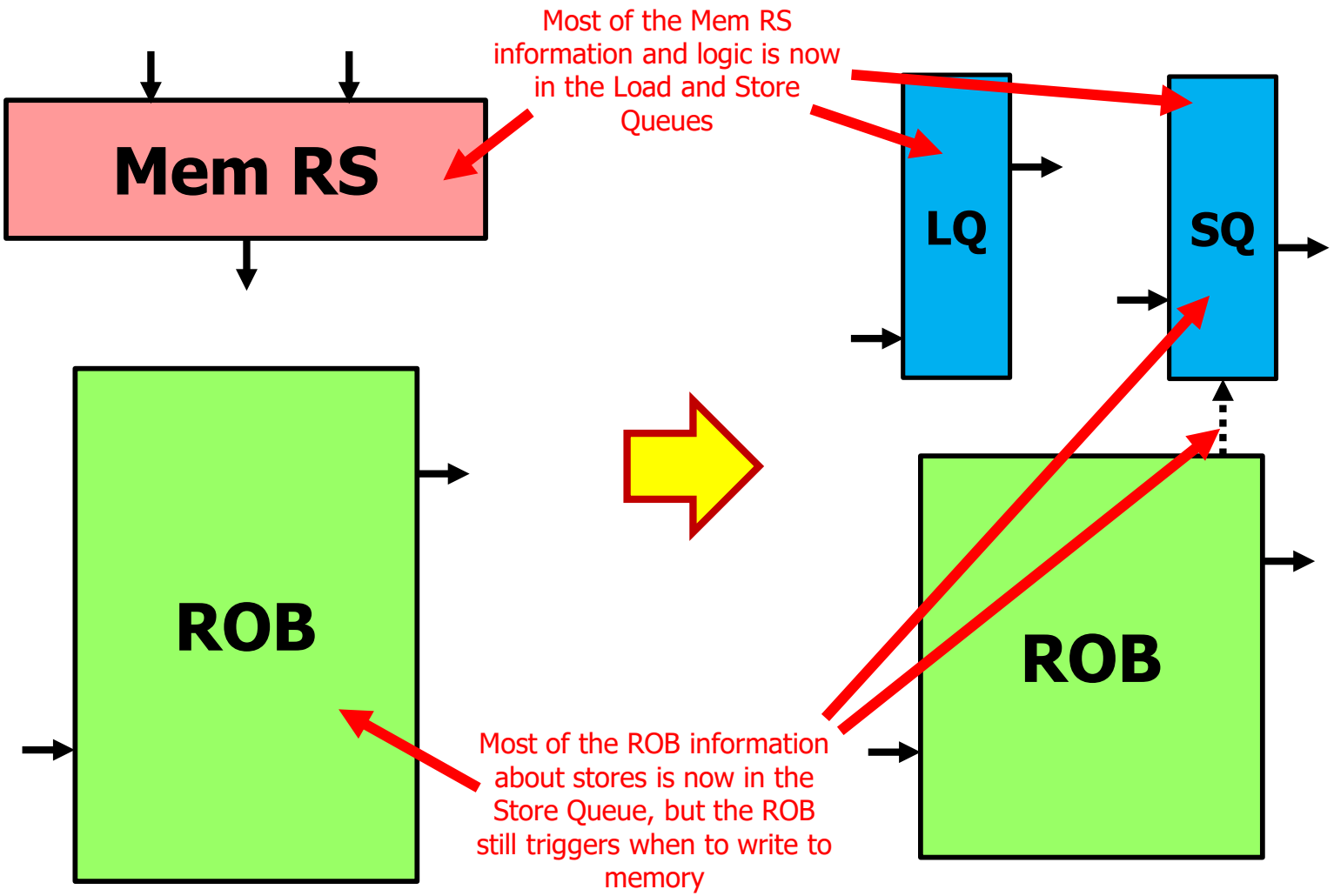
*but there is an additional situation now*

c) If there is a store to an **unknown address** in the ROB or if the address of the load is unknown, **wait!**

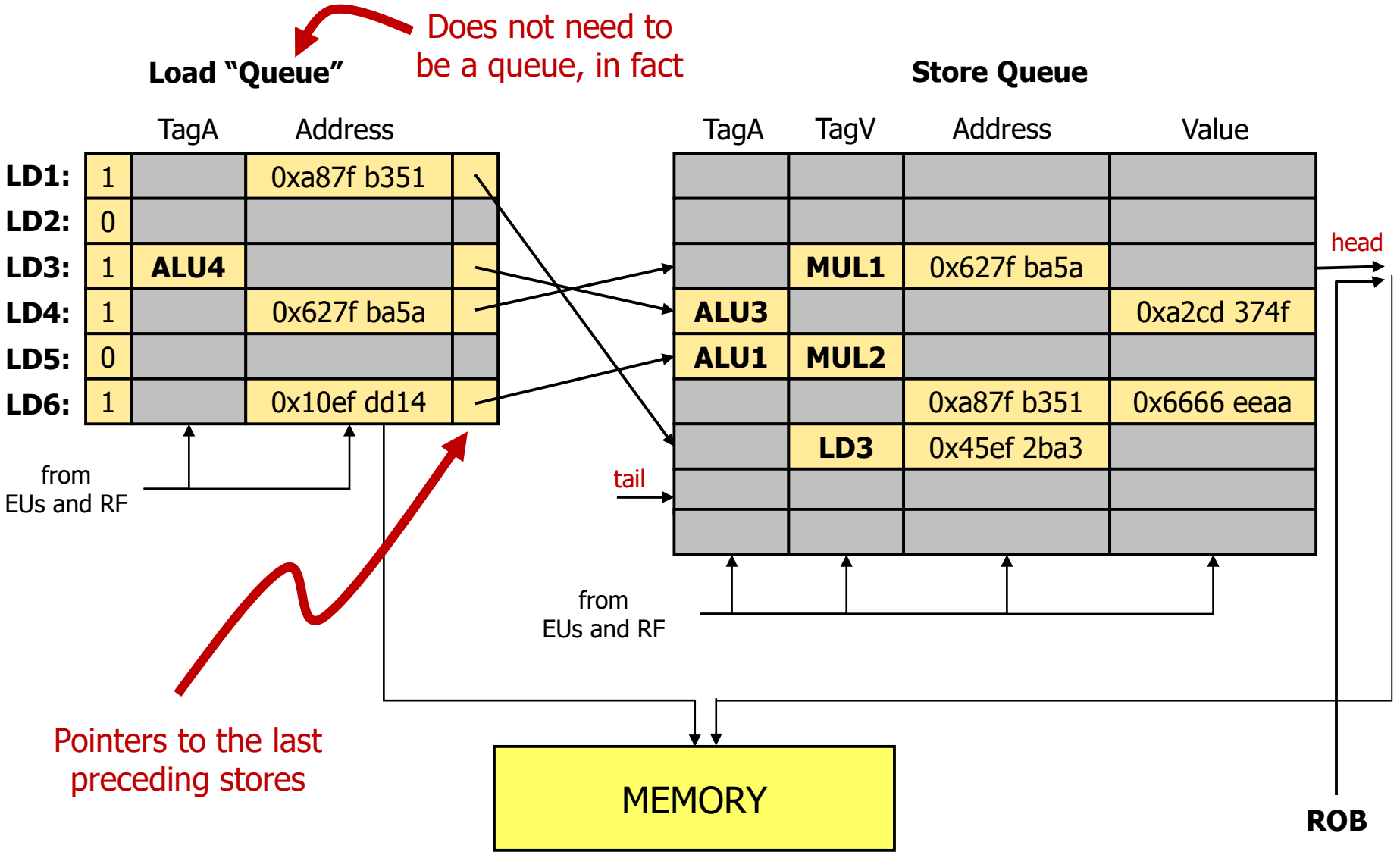
# Load-Store Queues

- The fact that there could be a **store to a yet-unknown address in ROB** the makes things harder:
  - Not only loads need to wait in the memory RS for their addresses (= waiting for their operands, which is normal RS business)
  - Ready loads (= with known addresses) need to **keep checking the ROB** until the address of all preceding stores is known
- In practice, this implies a strong coordination between the **memory RS** and the **memory (=store) part of the ROB** → All this is thus typically implemented in a **Load-Store Queue** (in turn, in fact, better implemented as individual load and store queues)
- The **load queue** may not be a queue, after all (see later)

# Load-Store Queues



# Example of Load Store Queue



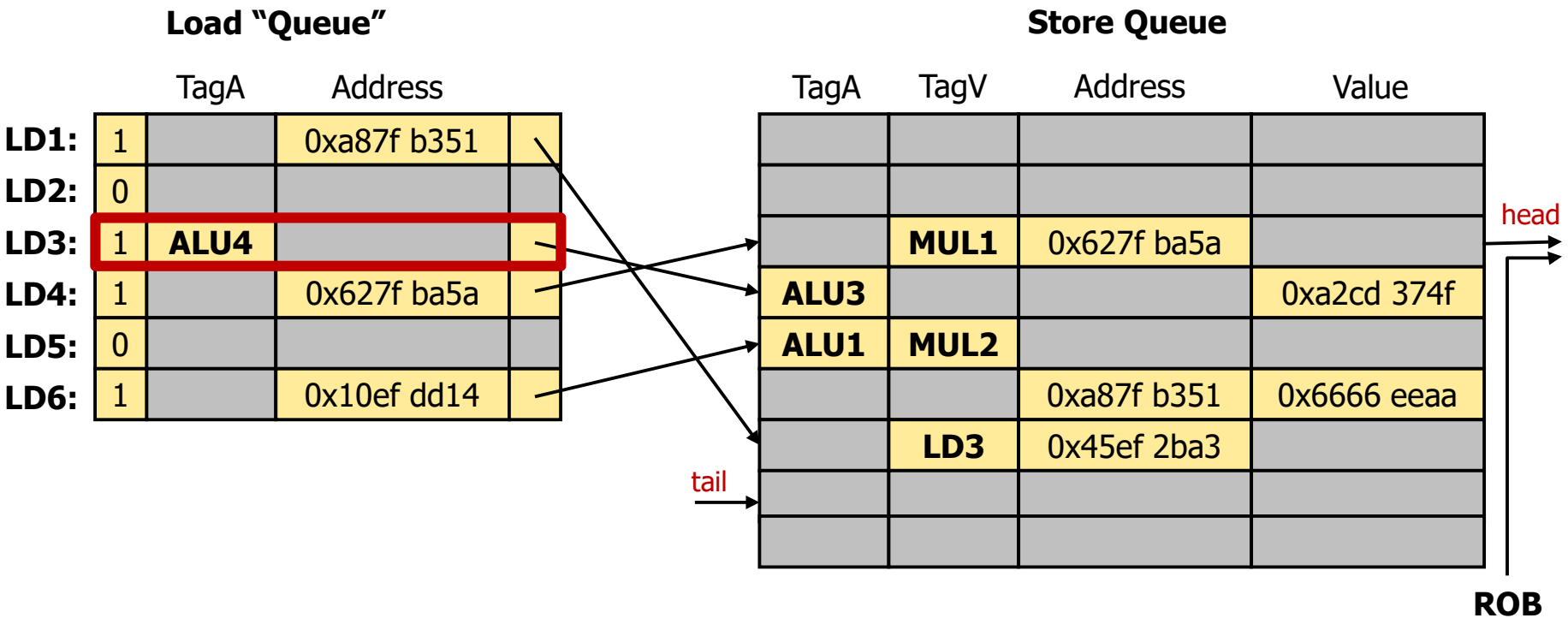
# Load Queue Functionality

- All ready loads (= those at known addresses) are checked **concurrently**
- Each **load compares** its address **with all preceding store** addresses and does (approximately) the following:
  - If any of the preceding **stores misses the address → do nothing**
  - If all preceding stores have an address and there is **no collision → issue the load** if there are available memory ports
  - If the load address equals one or more of the store addresses and if the last of the **colliding stores has the value → memory bypass** = load is executed and the returned value comes from the store queue
  - If the load address equals one or more of the store addresses and if the last of the **colliding stores has no value yet → do nothing** (will be a memory bypass later)
- This behaviour is essentially that of an RS but with the additional issue of checking for emerging collisions in the store queue

# Store Queue Functionality

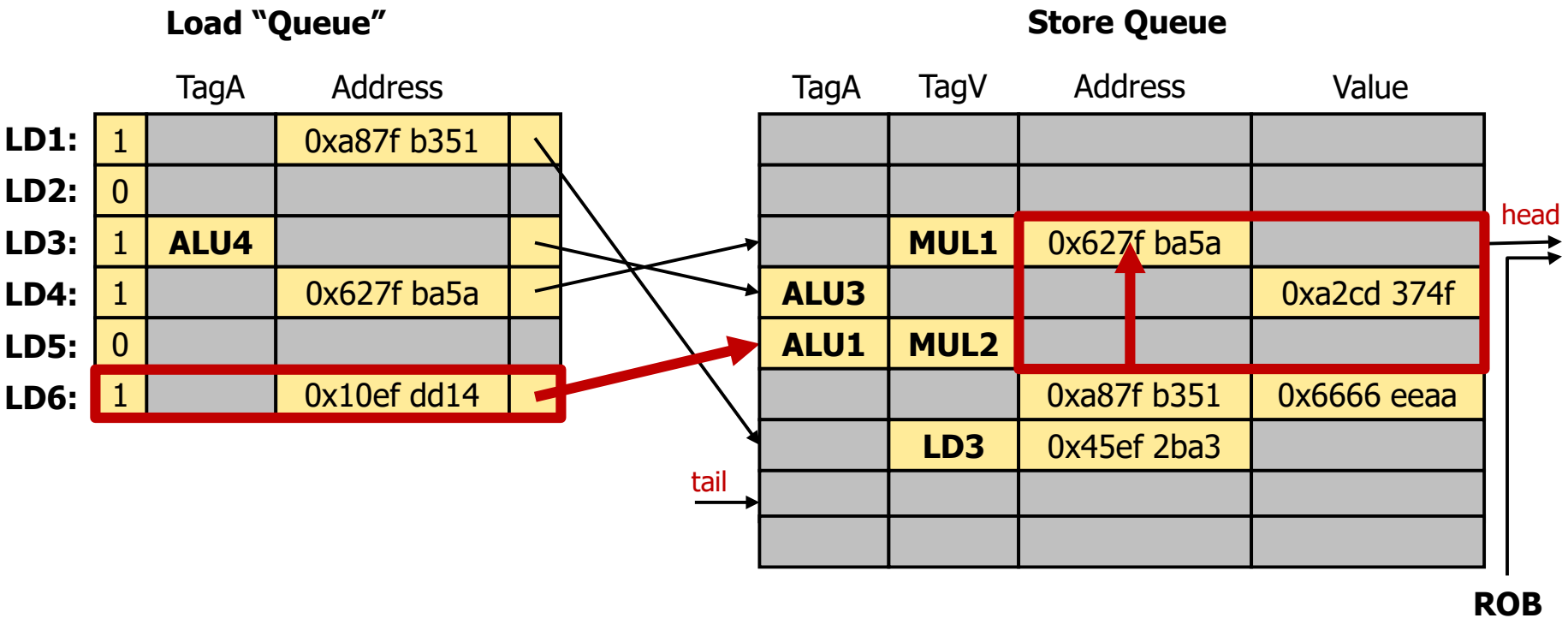
- Stores are executed only if
  1. The address and the data for the store are known (= standard RS functionality)
  2. All preceding stores executed (= in-order commit as ROB)
  3. The store is enabled from the ROB (= in-order commit w.r.t. other instructions in the ROB)
- If any of the tests fail, the store is kept waiting
- This behaviour is essentially that of an RS combined with the reordering of a ROB

# Example of Load Store Queue: Normal RS Behaviour in Load Queue



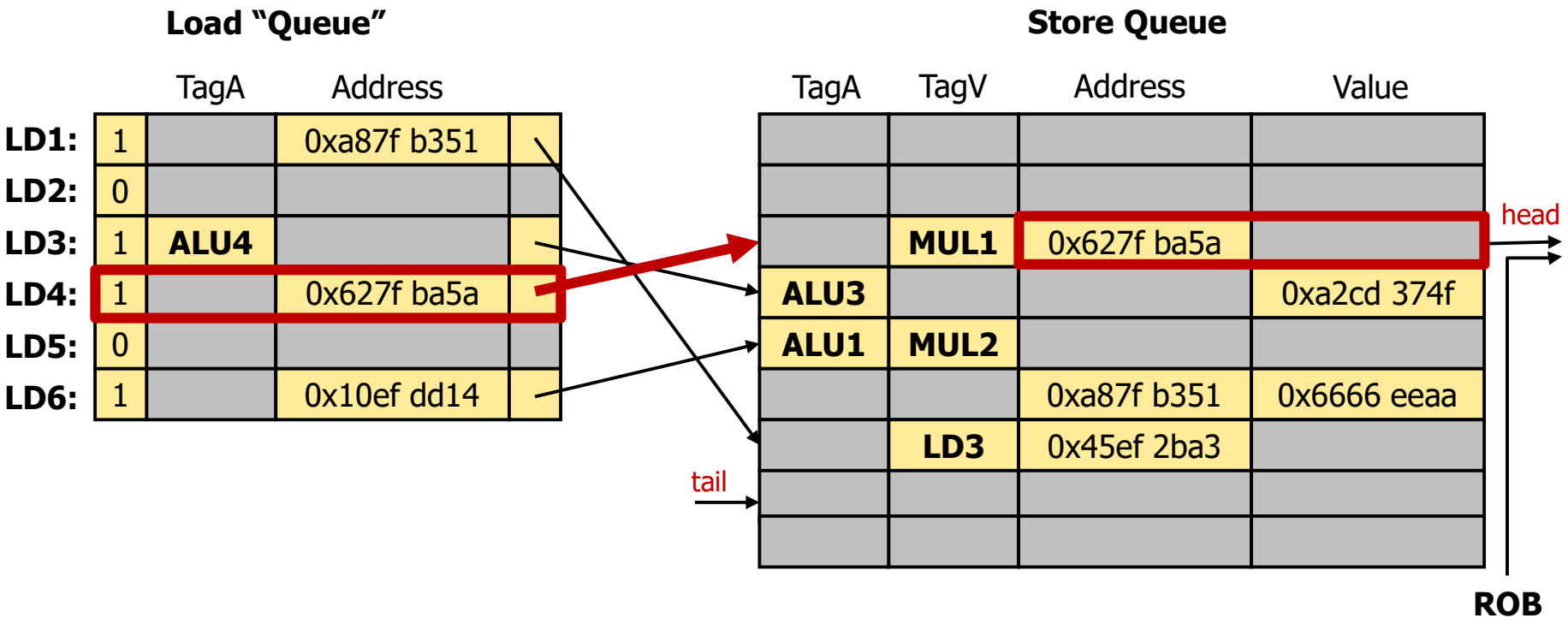

**LD3:** Unknown read address → wait

# Example of Load Store Queue Potential RAW through Memory



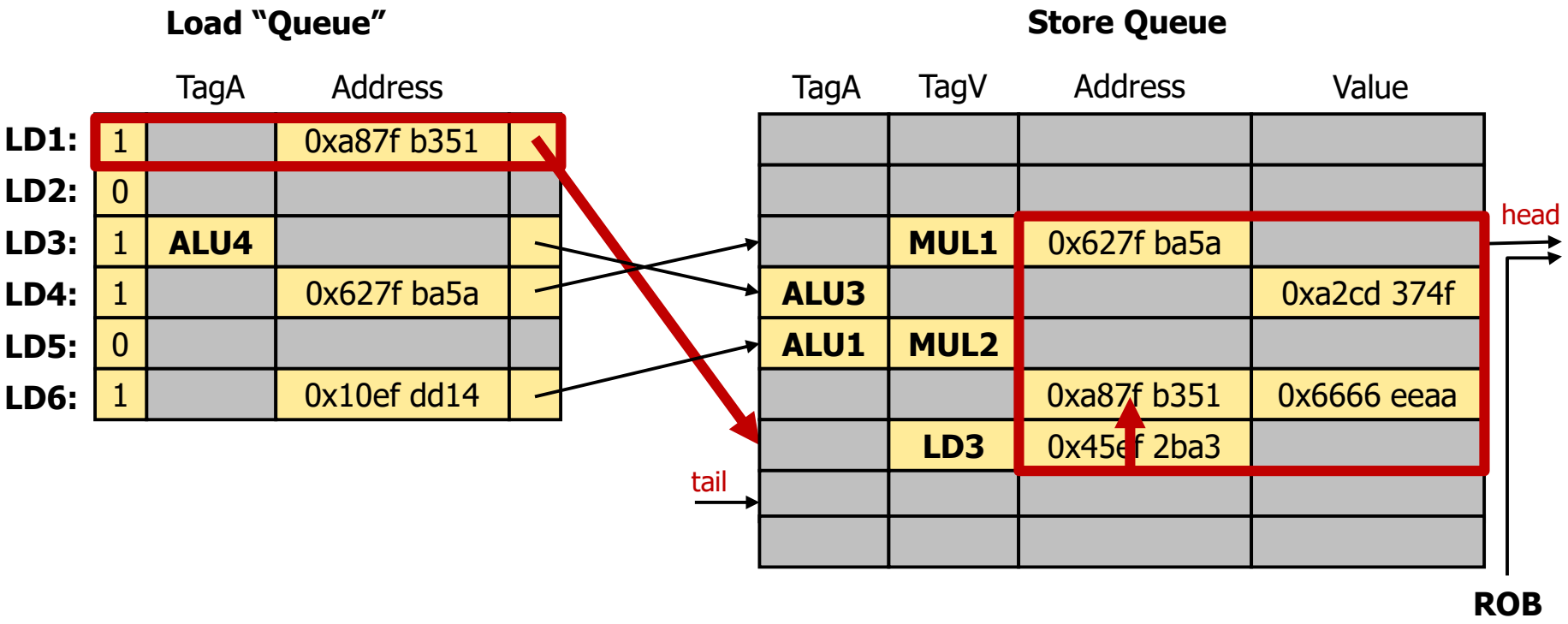

**LD6:** Potential RAW with stores whose address is unknown → wait

# Example of Load Store Queue: RAW through Memory



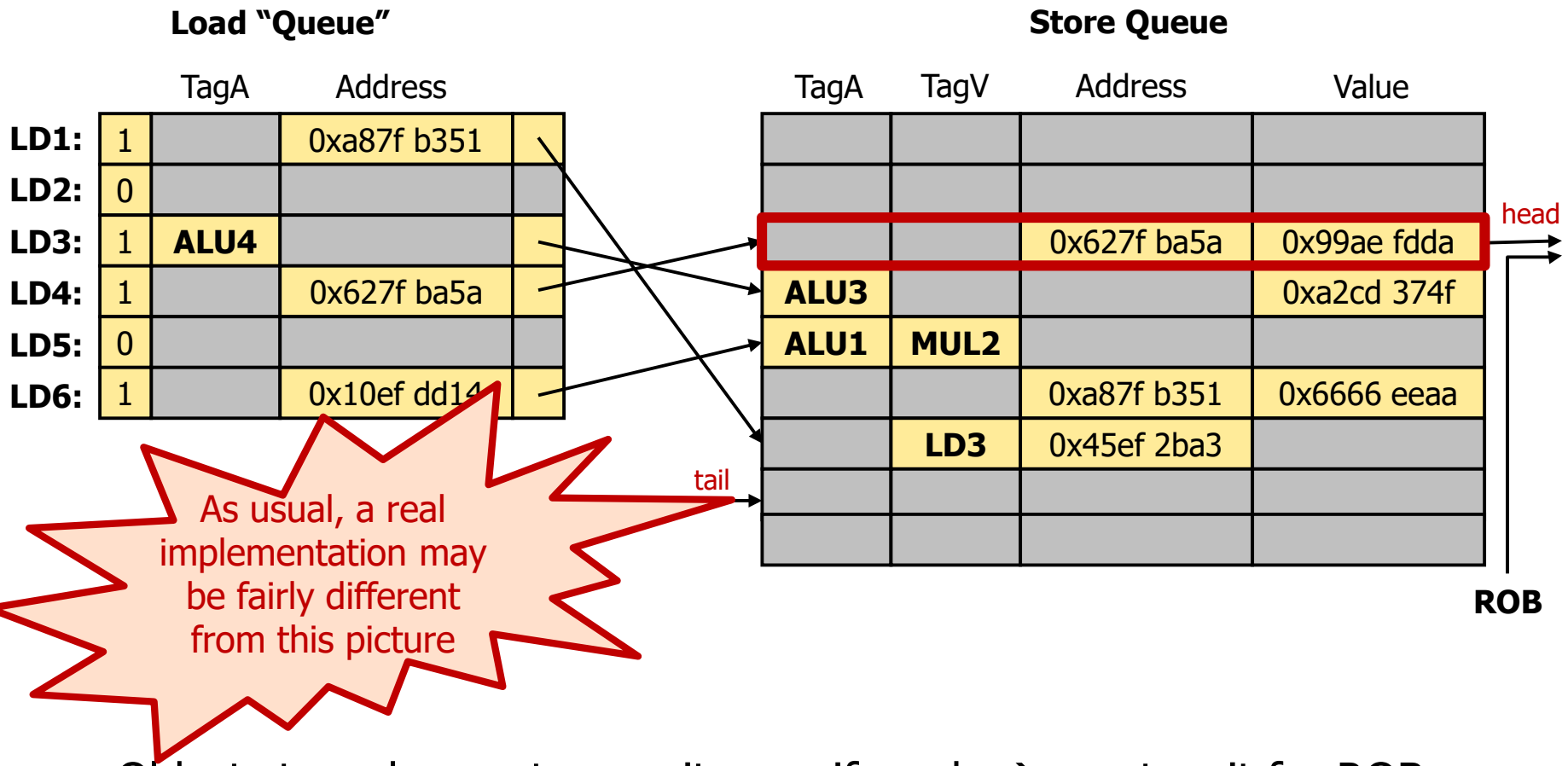

**LD4:** Known RAW at address `0x627f ba5a` → wait

# Example of Load Store Queue: RAW through Memory (bypass)




**LD1:** Known RAW at address **`0xa87f b351`** → return **`0x6666 eeaa`** without accessing memory

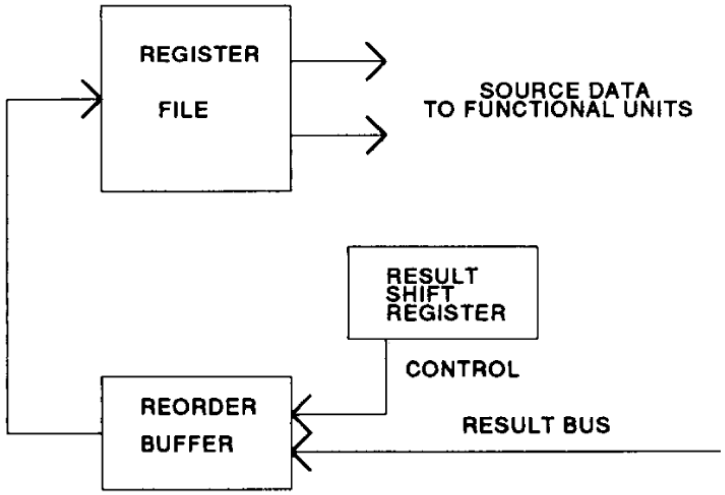
# Example of Load Store Queue: Stores Released by ROB

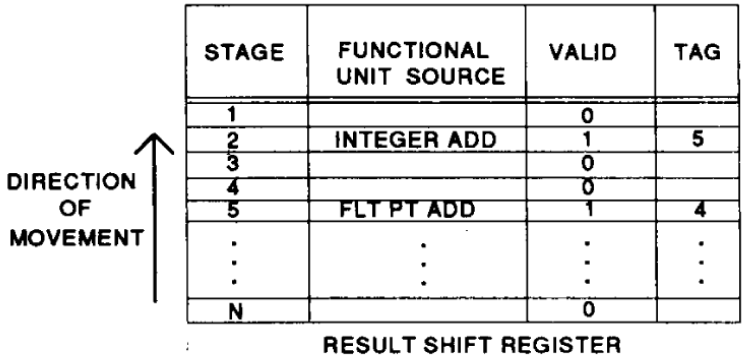


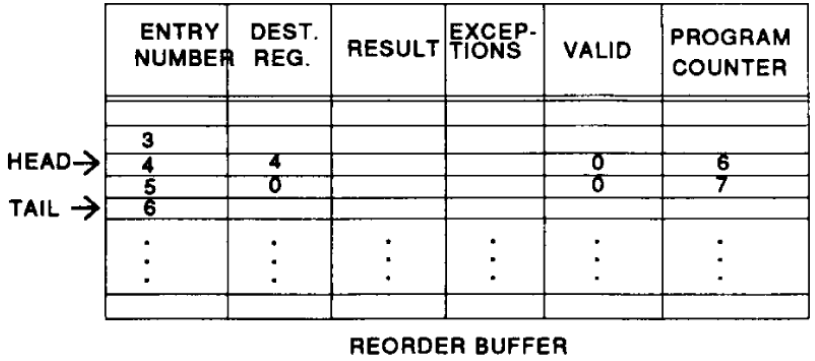

Oldest store does not commit even if ready → must wait for ROB

# Origins of Reordering

- Robert **Tomasulo** in 1967 for the IBM System/360 Model 91's floating point unit, but no support for precise interrupts
- Smith & Pleszkun on precise interrupts, 1988

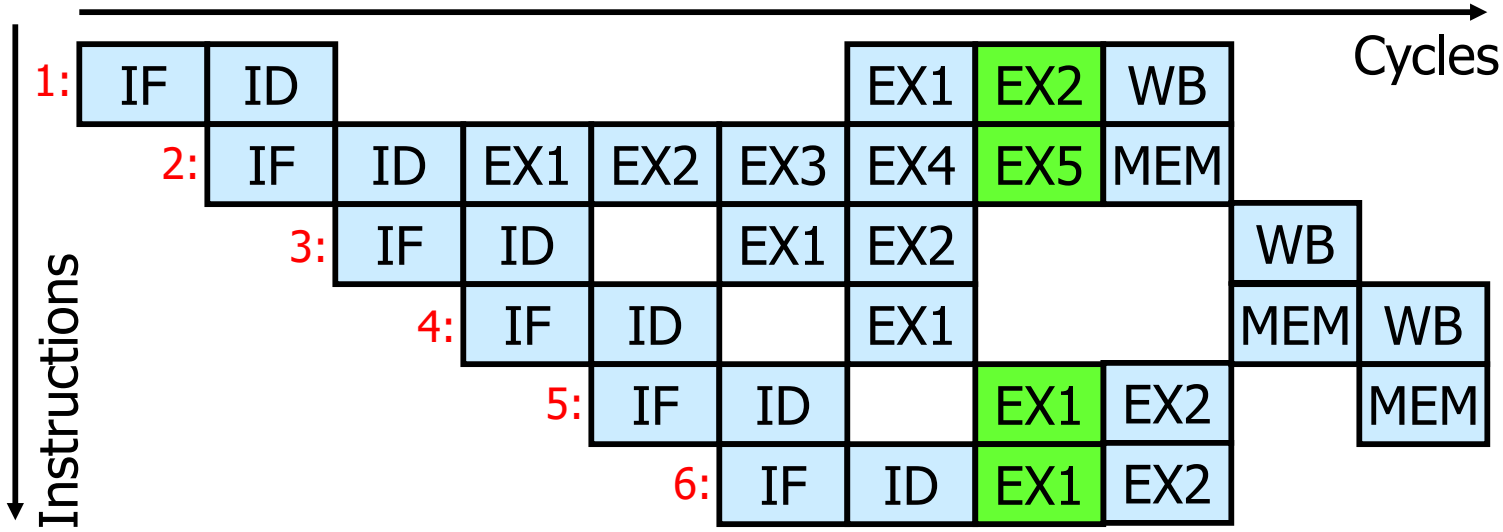


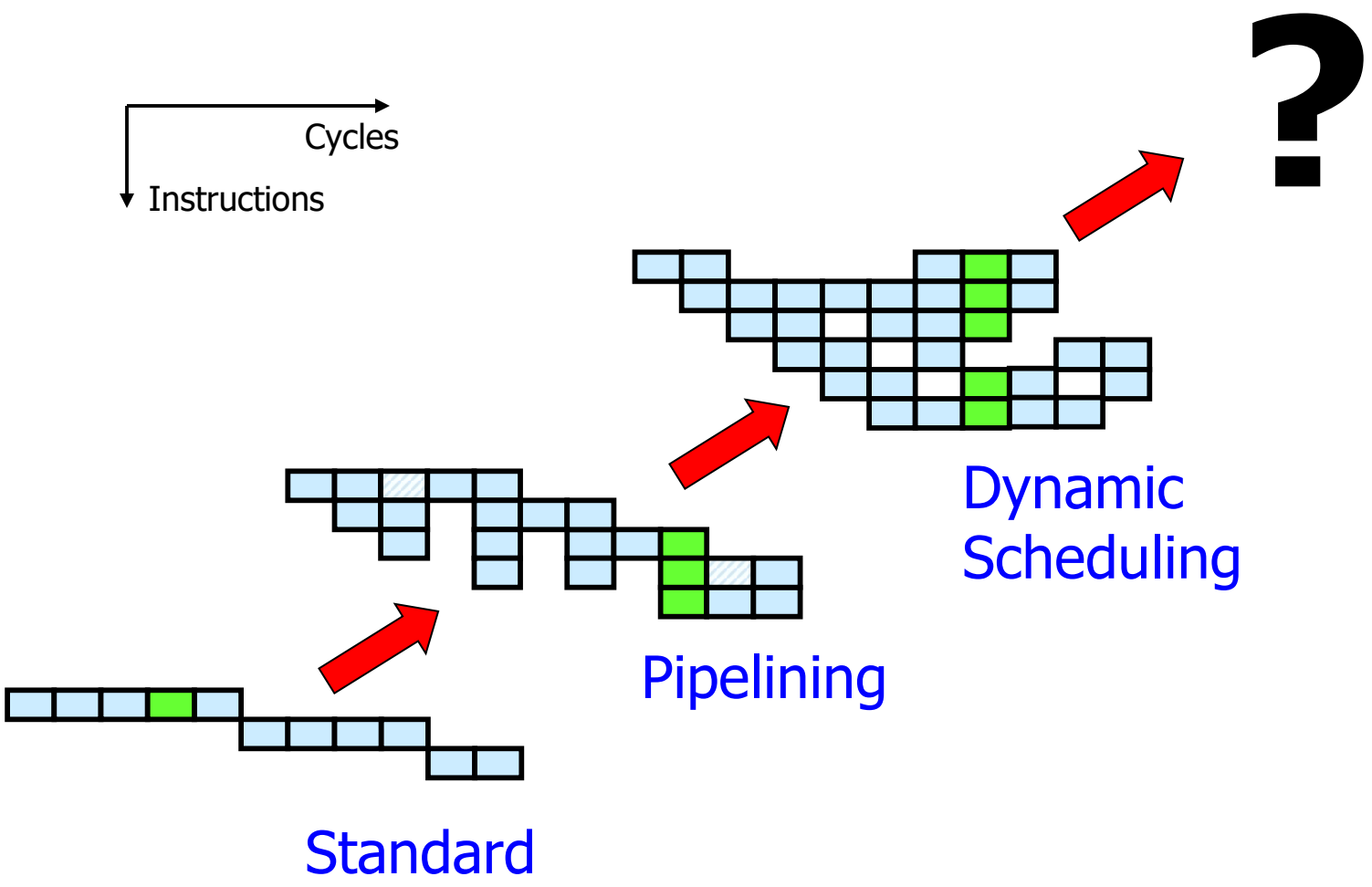

Source: Smith & Pleszkun, © IEEE 1988

# Second Step: Dynamic Scheduling




- Tangible amount of ILP now possible
- What's next?!

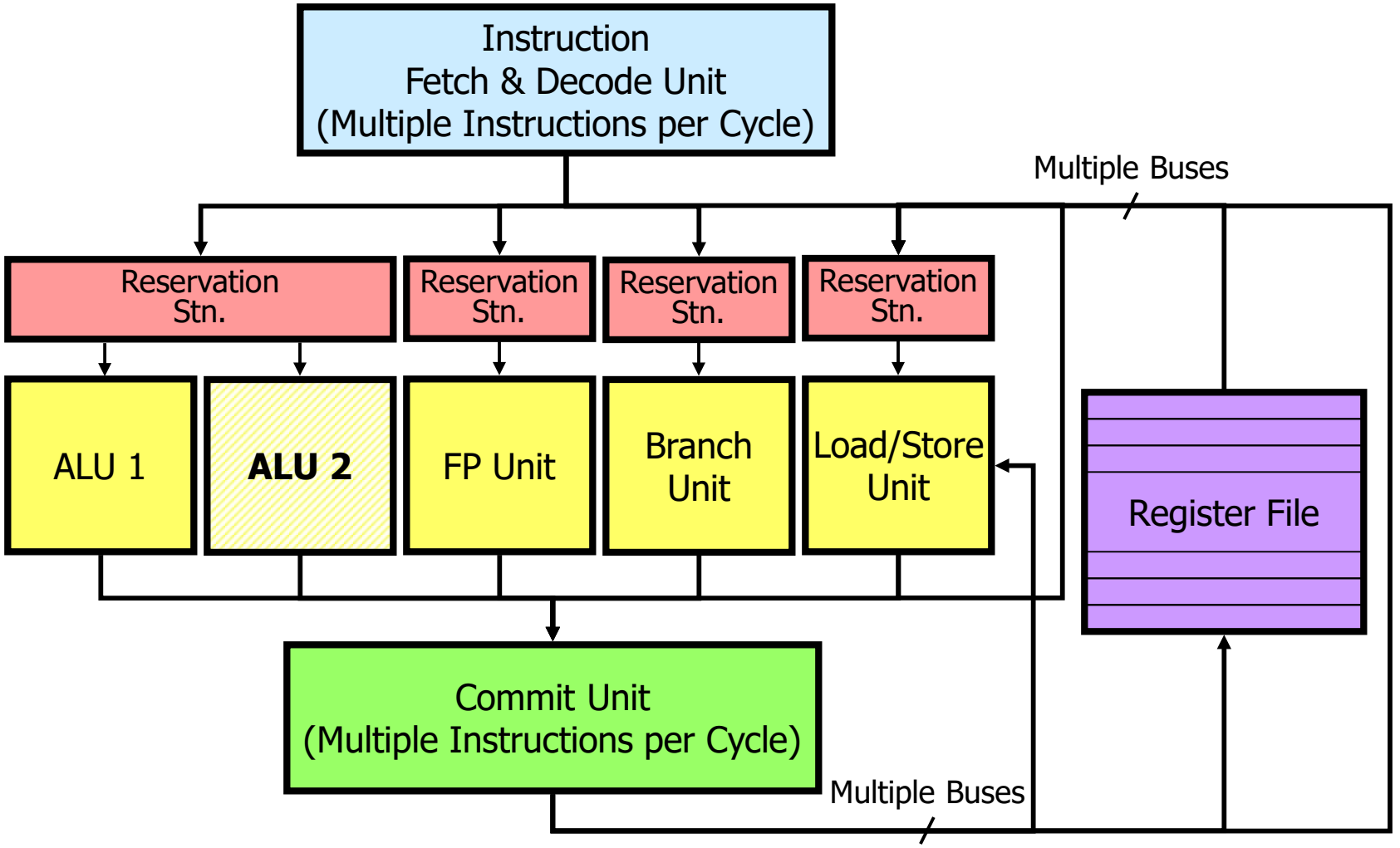
# ILP So Far...


Cycles
Instructions
?
Dynamic Scheduling
Pipelining
Standard

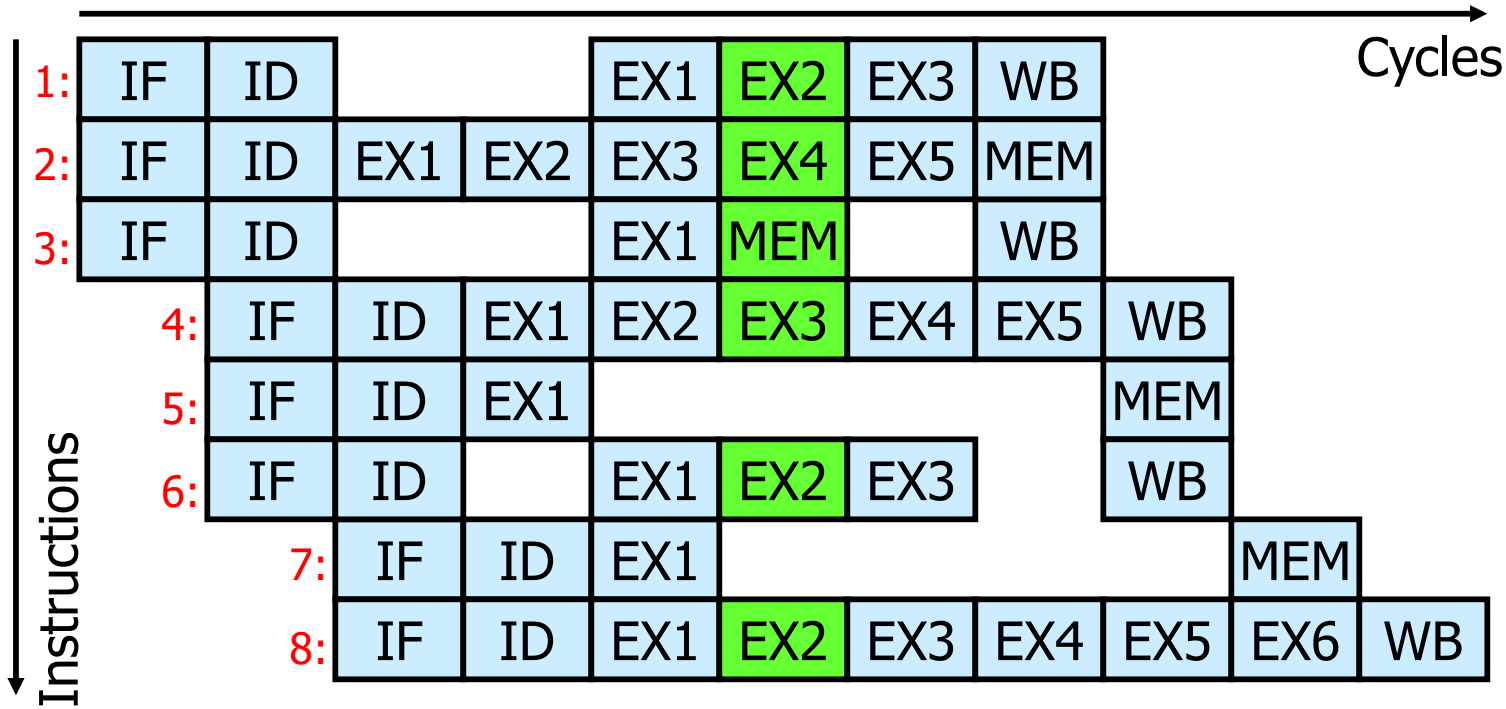
# Superscalar Execution

- Why not more than one instruction beginning execution (issued) per cycle?
- Key requirements are
  - Fetching more instruction in a cycle: no big difficulty provided that the instruction cache can sustain the bandwidth
  - Decide on data and control dependencies: dynamic scheduling already takes care of this

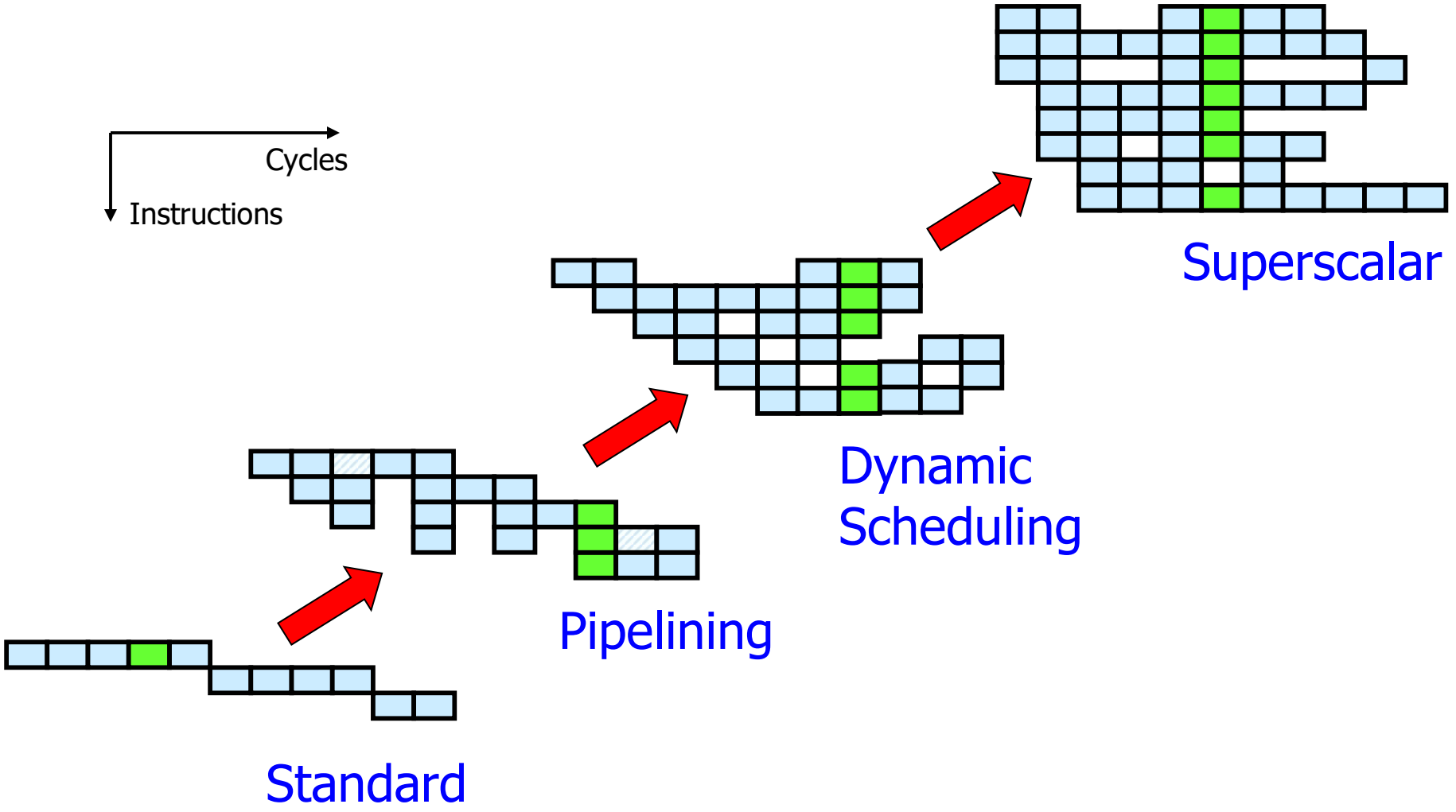
# Superscalar Processor



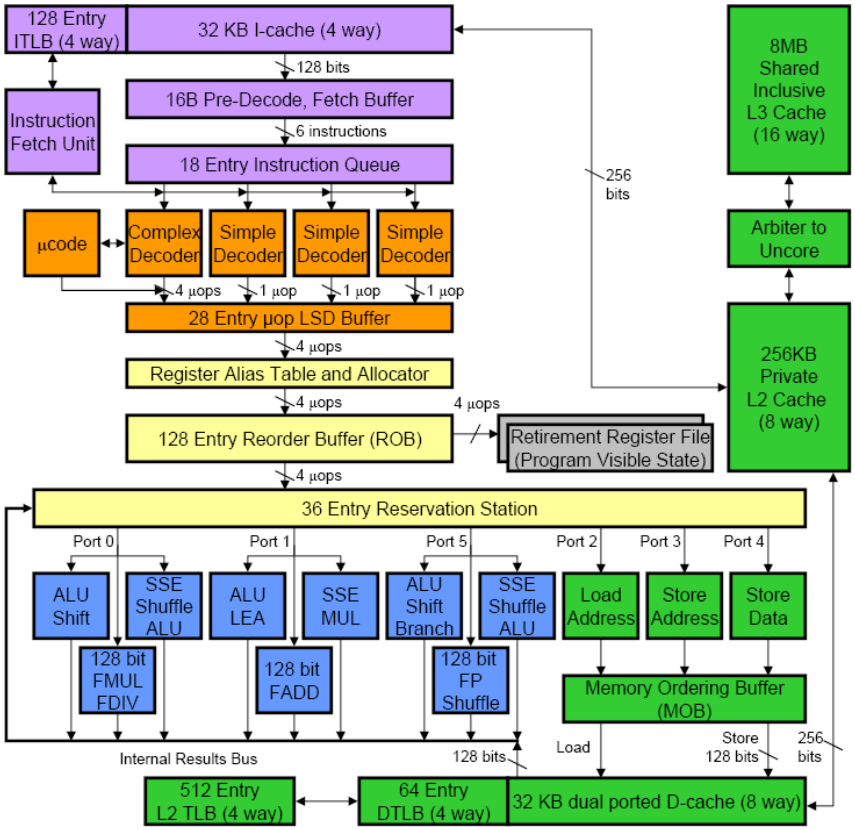
# Third Step: Superscalar Execution



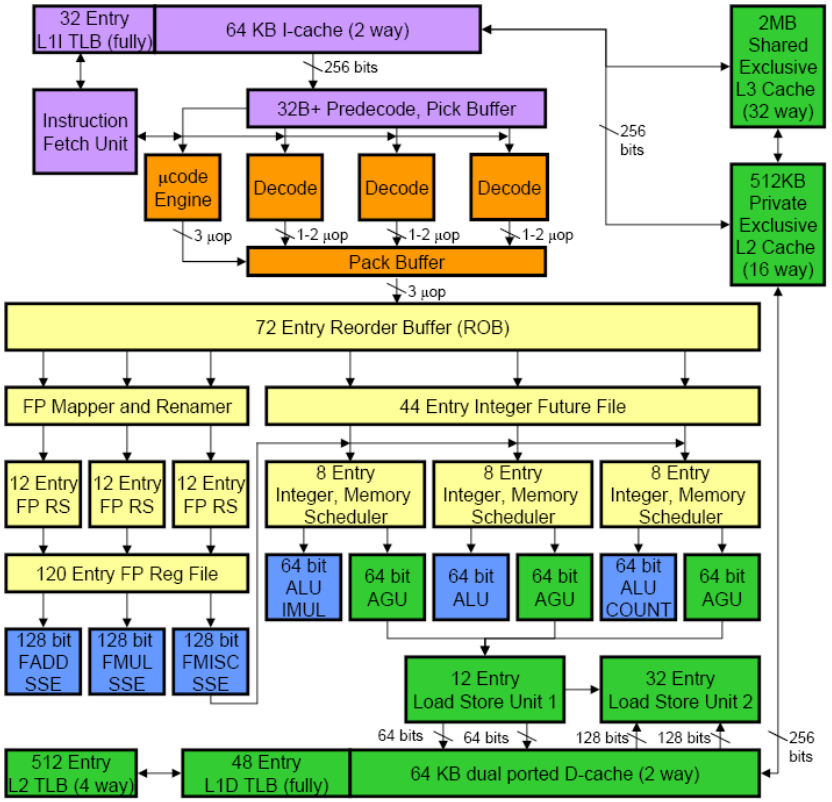
# Several Steps in Exploiting ILP



# Intel Nehalem and AMD Barcelona: Now Oldish Microarchitectures



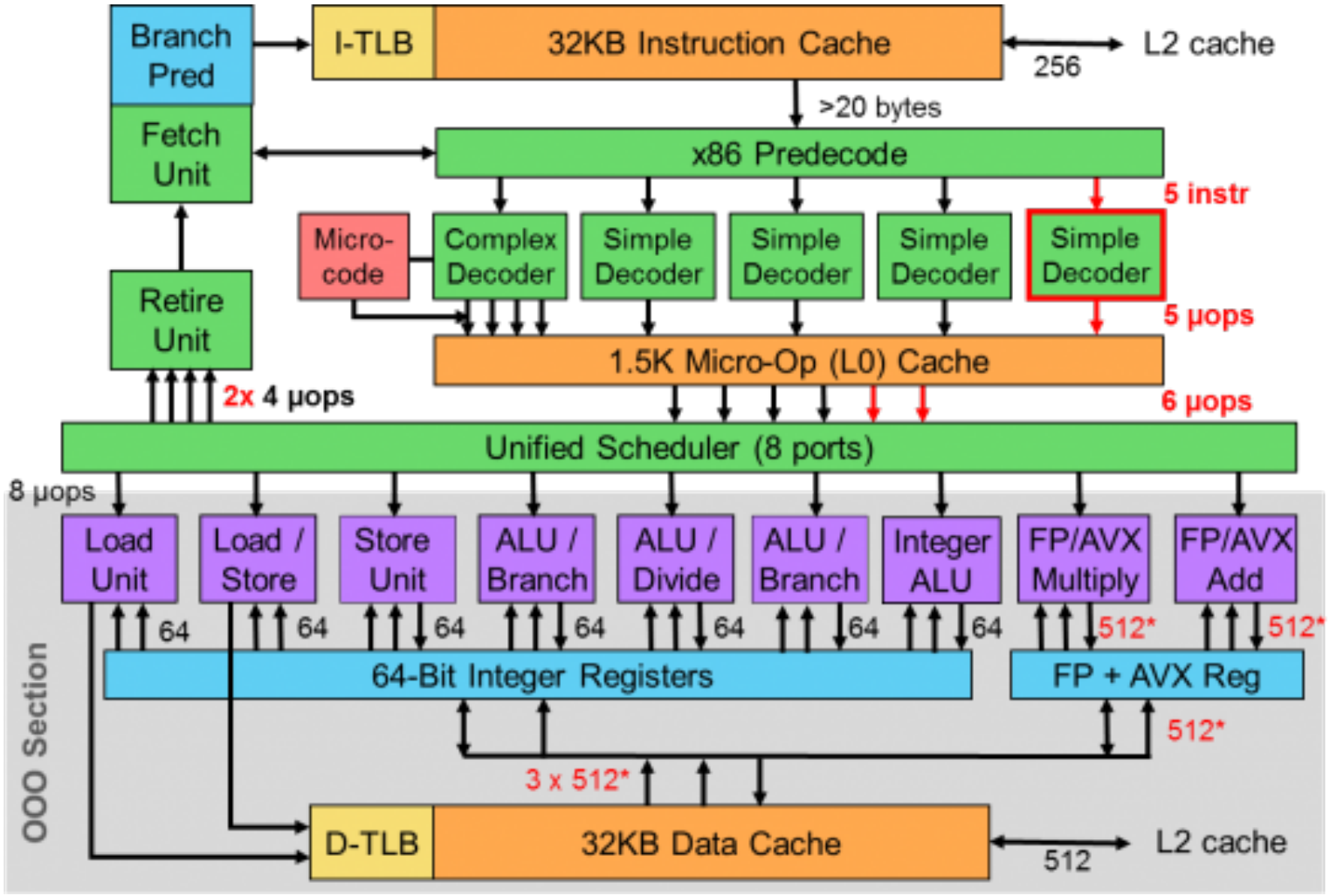

Intel Nehalem



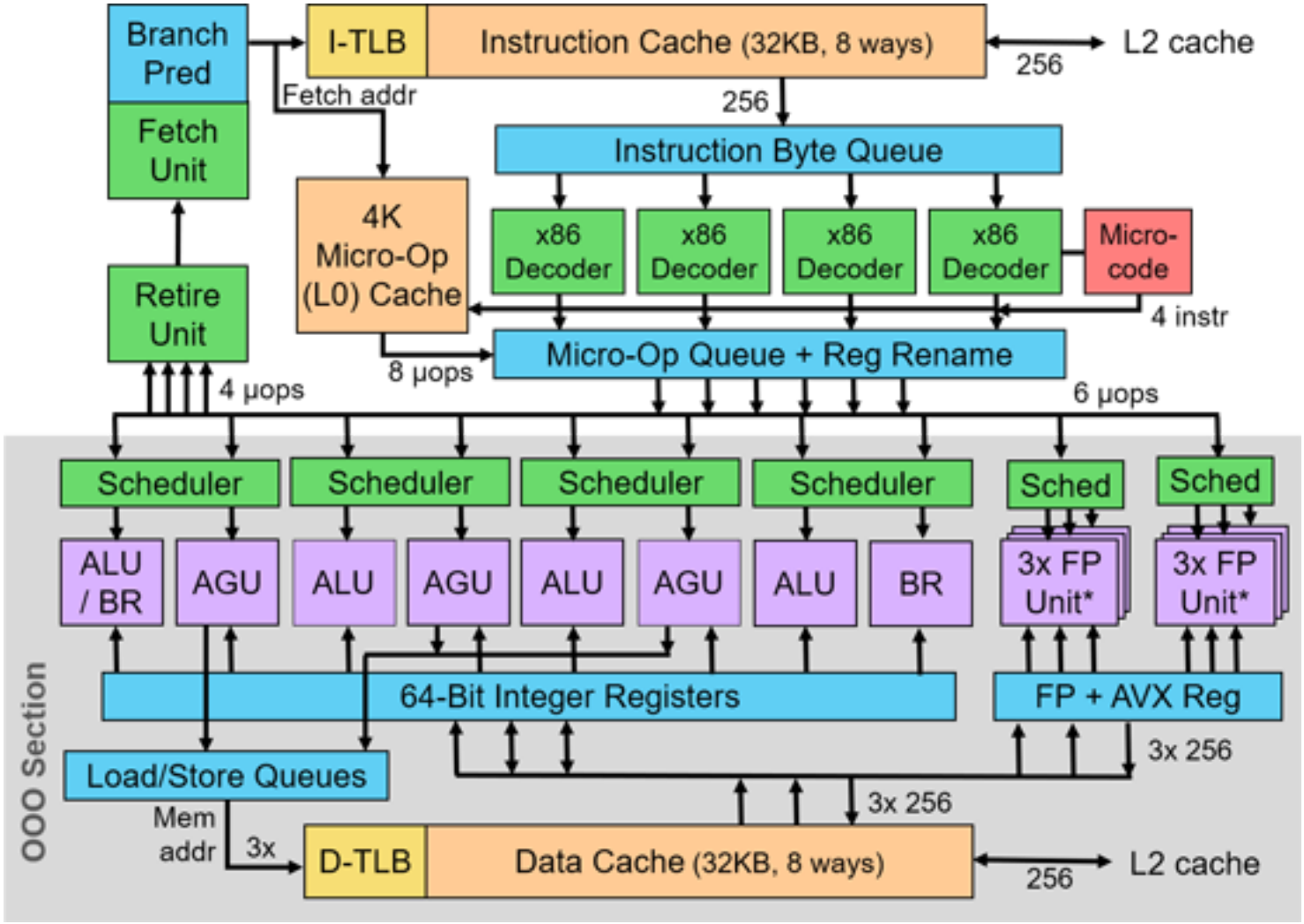

AMD Barcelona

Source: http://realworldtech.com/, © RWT 2008

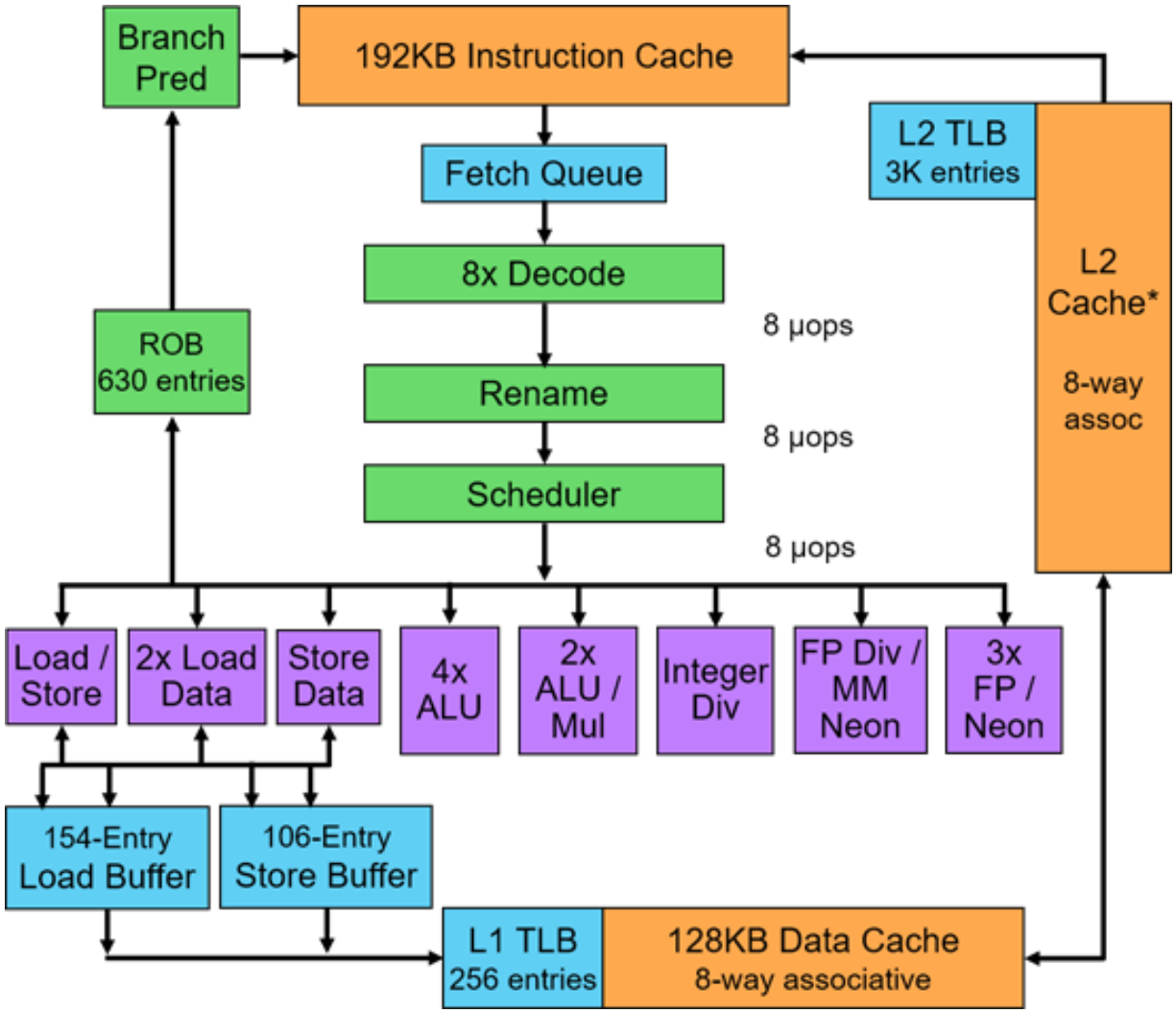
# Intel Skylake Microarchitecture



# AMD Zen 3 Microarchitecture



# Apple Firestorm Microarchitecture



# References on ILP

- AQA 5th ed., Appendix C
- CAR, Chapter 4—Introduction
- J. E. Smith and A. R. Pleszkun, *Implementation of Precise Interrupts in Pipelined Processors*, IEEE Transactions on Computers, 37(5):562-73, May 1988

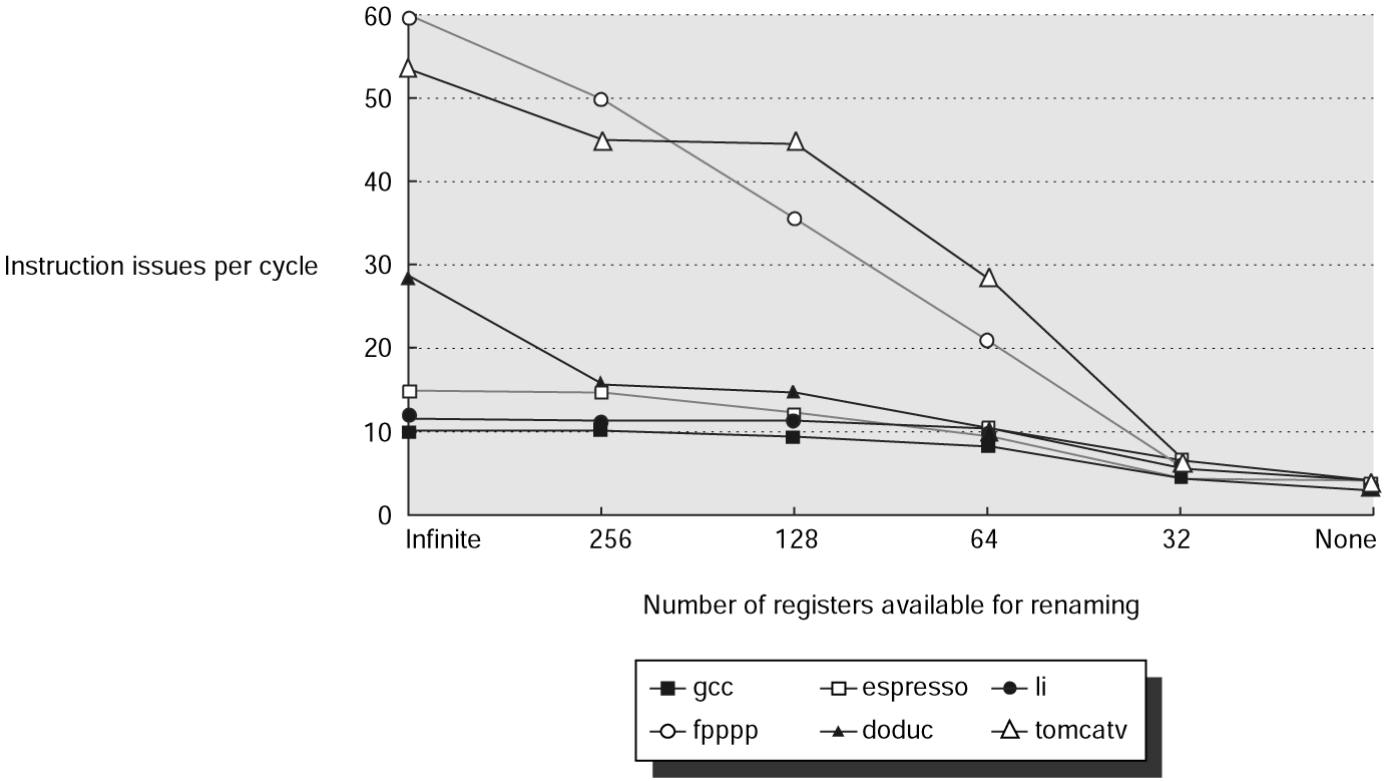
# 2

## Register Renaming

## (How Do I Get Rid of WAR and WAW?!...)

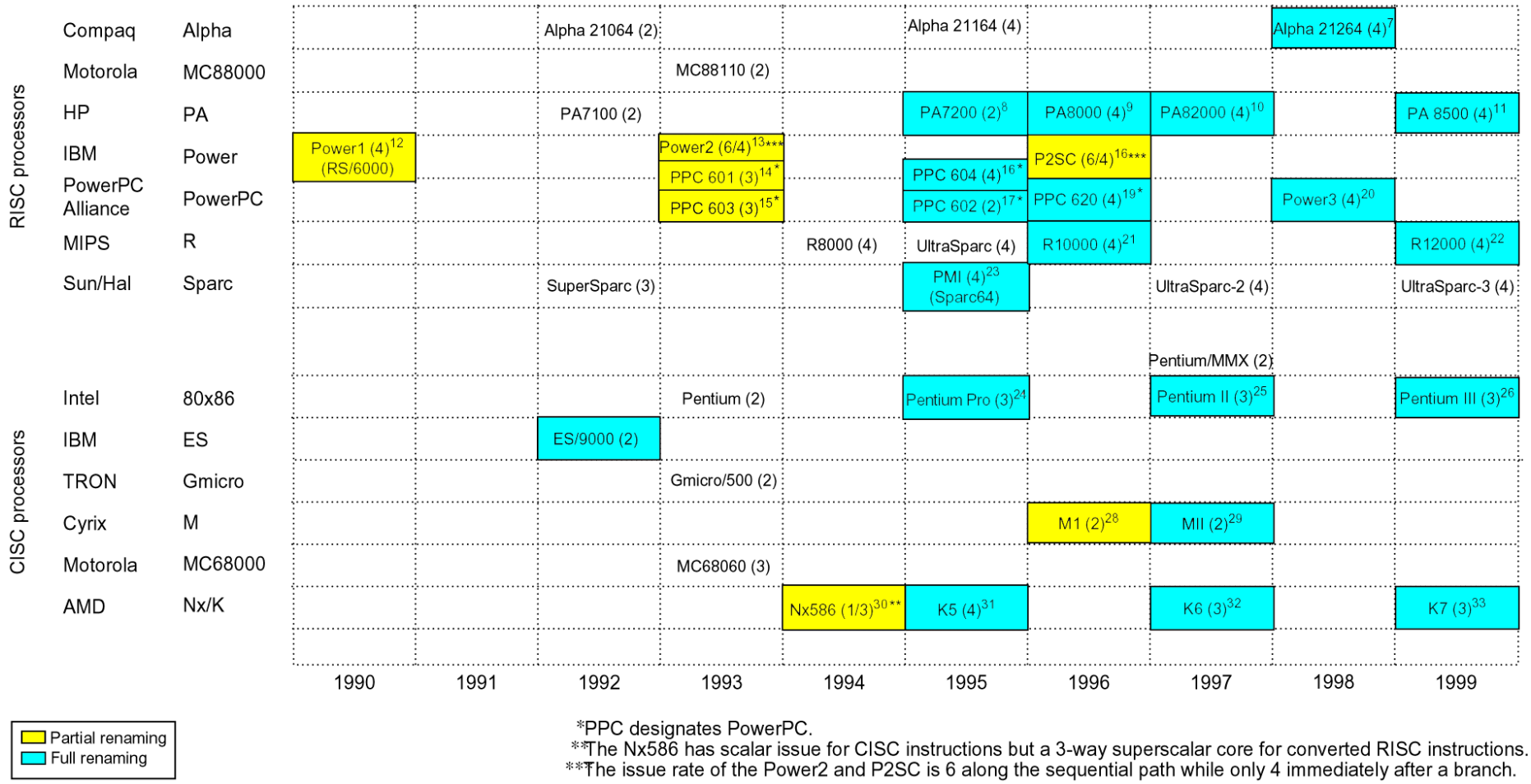
# Register Renaming

- Importance of removing WAR and WAW dependences with "close-to-ideal" instruction windows (2K entries) and maximum issue rate (64 per cycle)




Source: AQA, © Morgan Kauffman 1996

# A Little History of (Modern) Renaming




*PPC designates PowerPC.
**The Nx586 has scalar issue for CISC instructions but a 3-way superscalar core for converted RISC instructions.
**The issue rate of the Power2 and P2SC is 6 along the sequential path while only 4 immediately after a branch.

Source: Sima, © IEEE 2000

First: **IBM 360/91** (1967, FP partial renaming)

# Main Dimensions in Renaming Policies

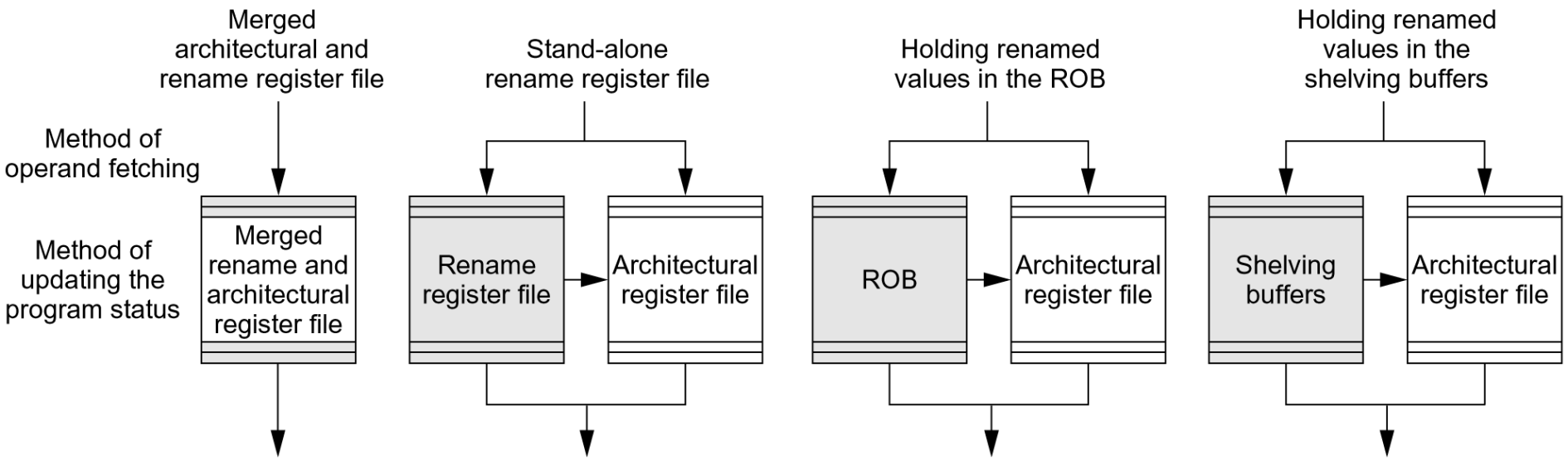
1. Scope of register renaming
   - Simple: only some classes of registers are renamed (e.g., integer or FP only)
2. Layout of the renamed registers
   - Where are they?
3. Method of register mapping
   - Allocation, tracking, and deallocation
4. Rename rate
   - How many instructions can be renamed at once?

# Where Are the Rename Registers?

Four possibilities:

1. Merged rename and architectural RF
2. Split rename and architectural RFs
3. Renamed values in the reorder buffer
4. Renamed values in the reservation stations (a.k.a. shelving buffers)

# Four Possible Locations for Rename Registers




Power1 (1990)
Power2 (1993)
ES/9000 (1992)
Nx586 (1994)
PMI (Sparc64, 1995)
R10000 (1996)
R12000 (1999)
Alpha 21264 (1998)

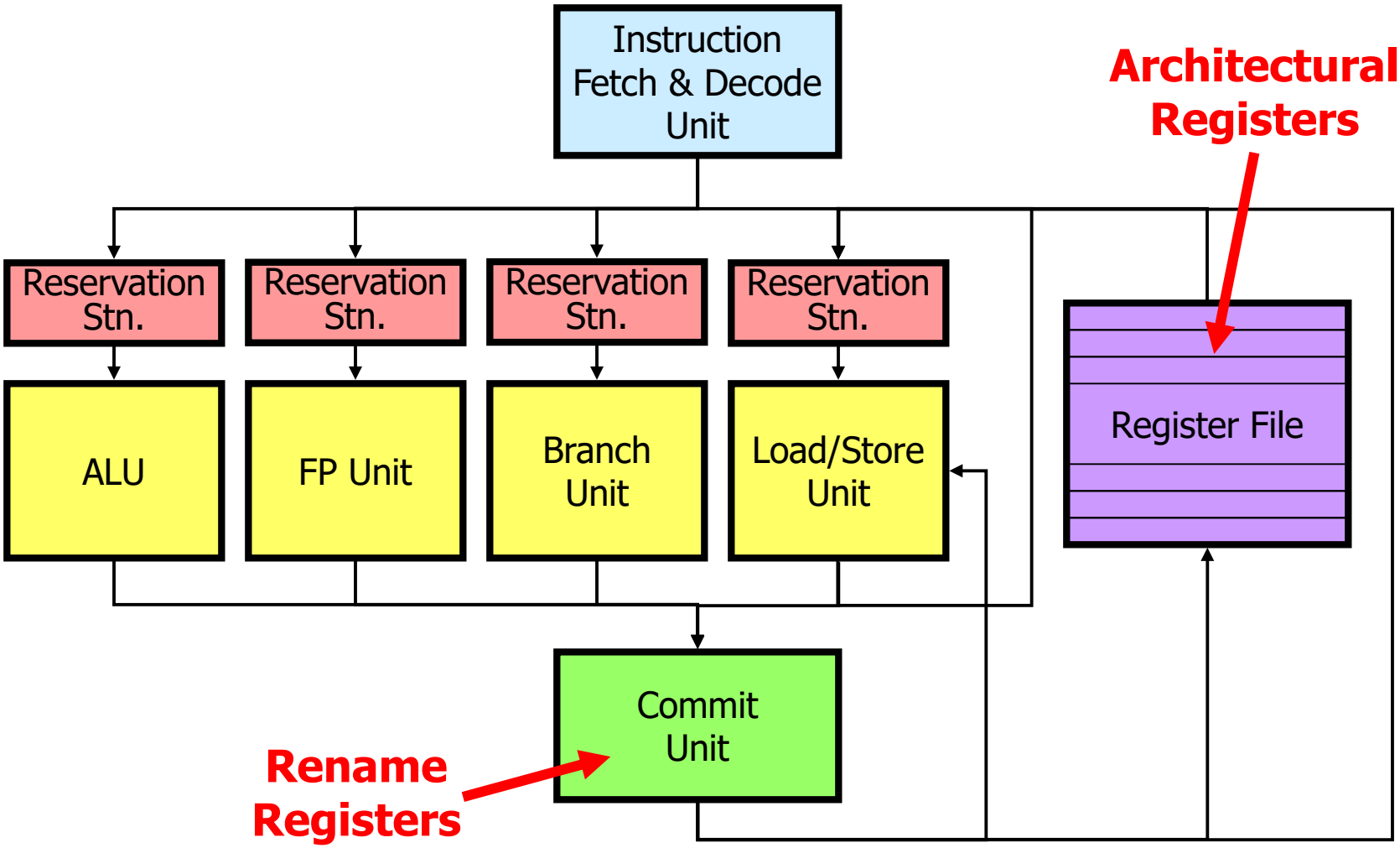
PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
Power3 (1998)
PA 8000 (1996)
PA 8200 (1997)
PA 8500 (1999)

Am290000 superscalar (1995)
K5 (1995)
M1 (1995)
K6 (1997)
Pentium Pro (1995)
Pentium II (1997)
Pentium III (1999)

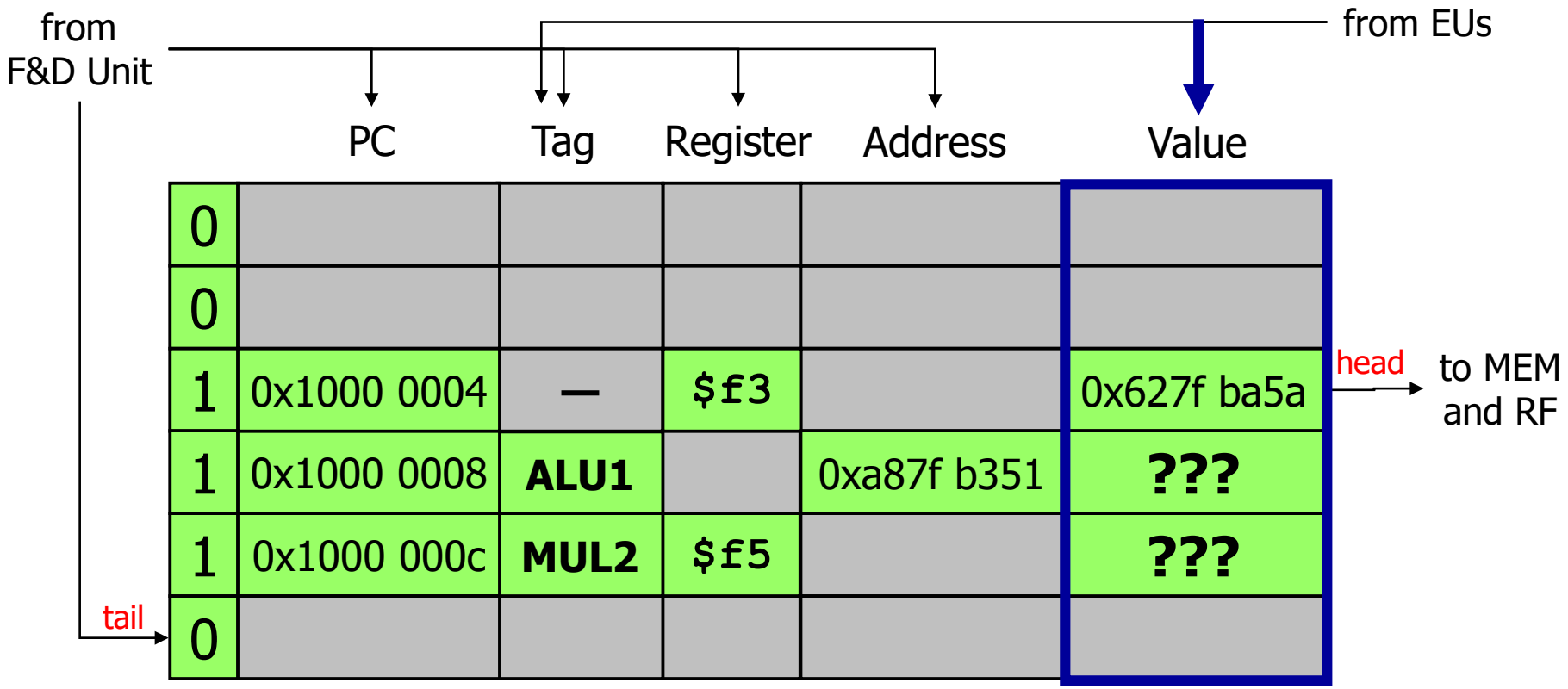
—

Source: Sima, © IEEE 2000

# Dynamically Scheduled Processor

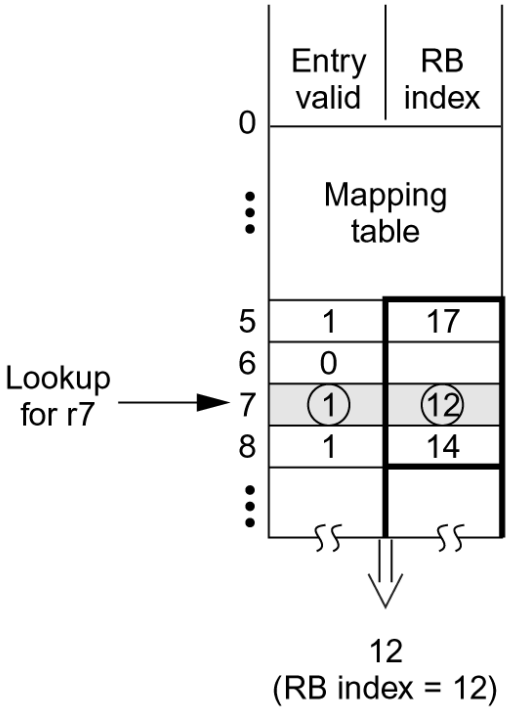


# Typical ROB

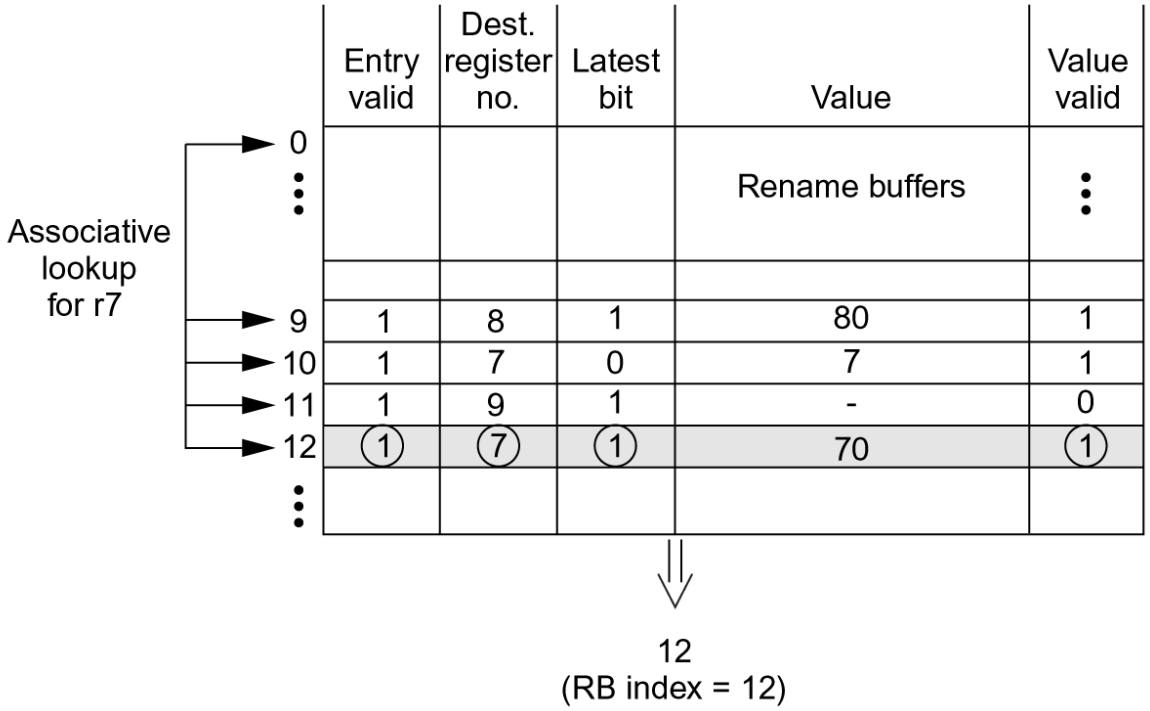

from
F&D Unit
from EUs
PC
Tag
Register
Address
Value
0
0
1
0x1000 0004
—
$f3
0x627f ba5a
1
0x1000 0008
ALU1
0xa87f b351
???
1
0x1000 000c
MUL2
$f5
???
0
head
to MEM
and RF
tail

# Tracking the Mapping: Where is Physically an Architectural Register?

## Mapping in a **Mapping Table**

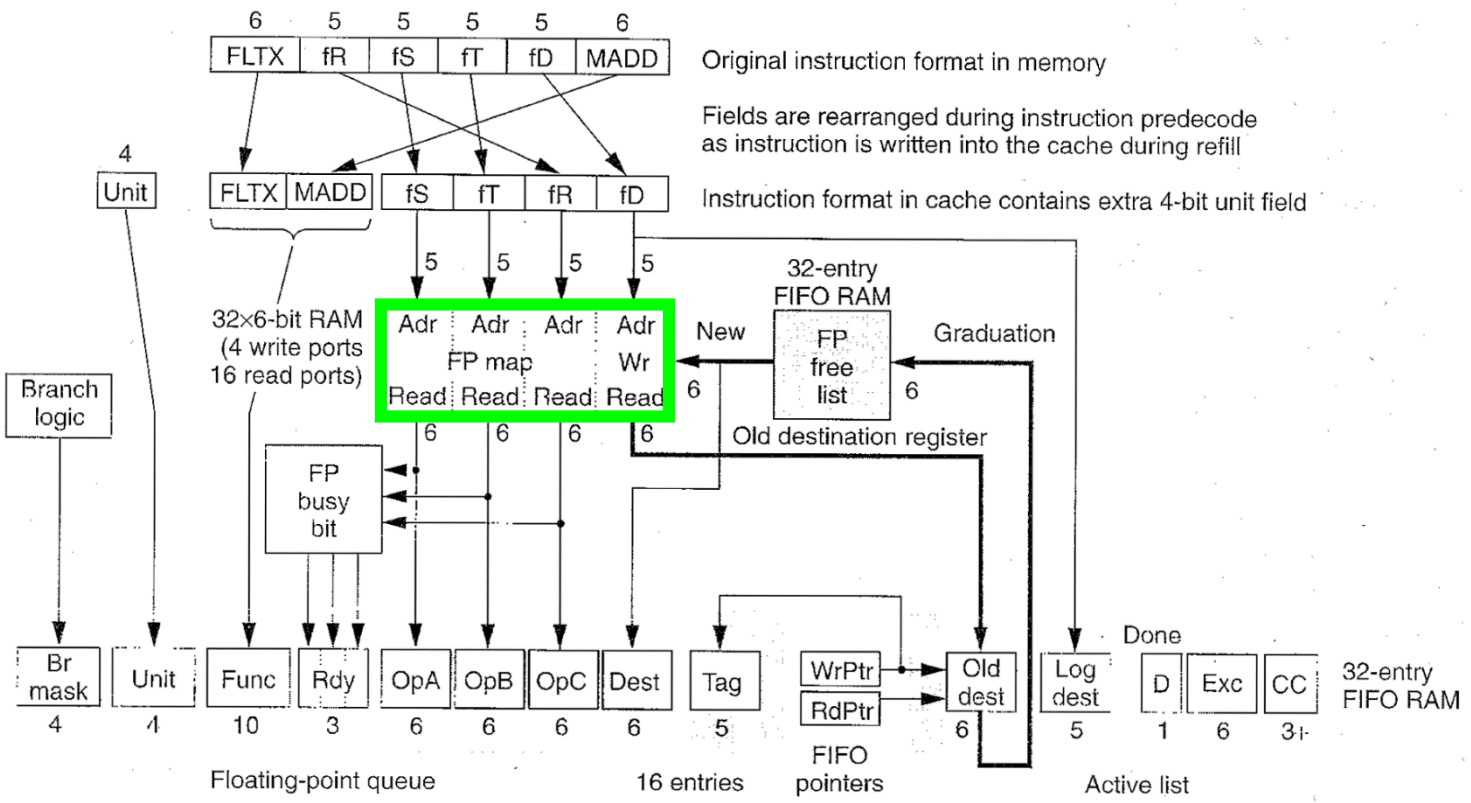



## Mapping in the **Rename Buffer**




Source: Sima, © IEEE 2000

# MIPS R10000: Merged RF with Mapping Table

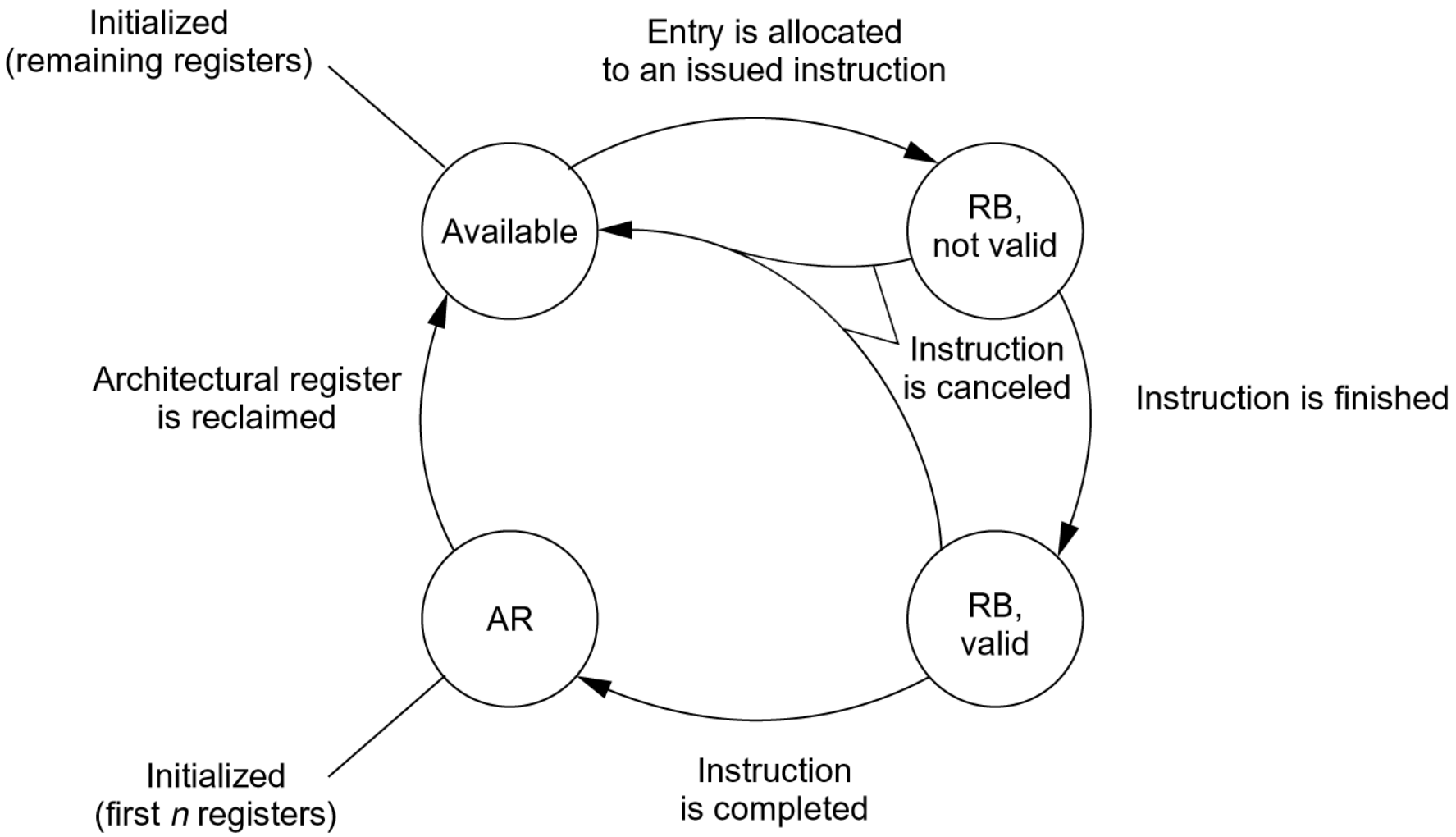



Source: Yeager, © IEEE 1996

Remark the complexity of the **Mapping Table**:

- 4-issue processor (→ 4x above scheme in parallel)
- 16 parallel accesses: 16 read ports and 4 write ports!

# Possible States of each Register in a Merged File



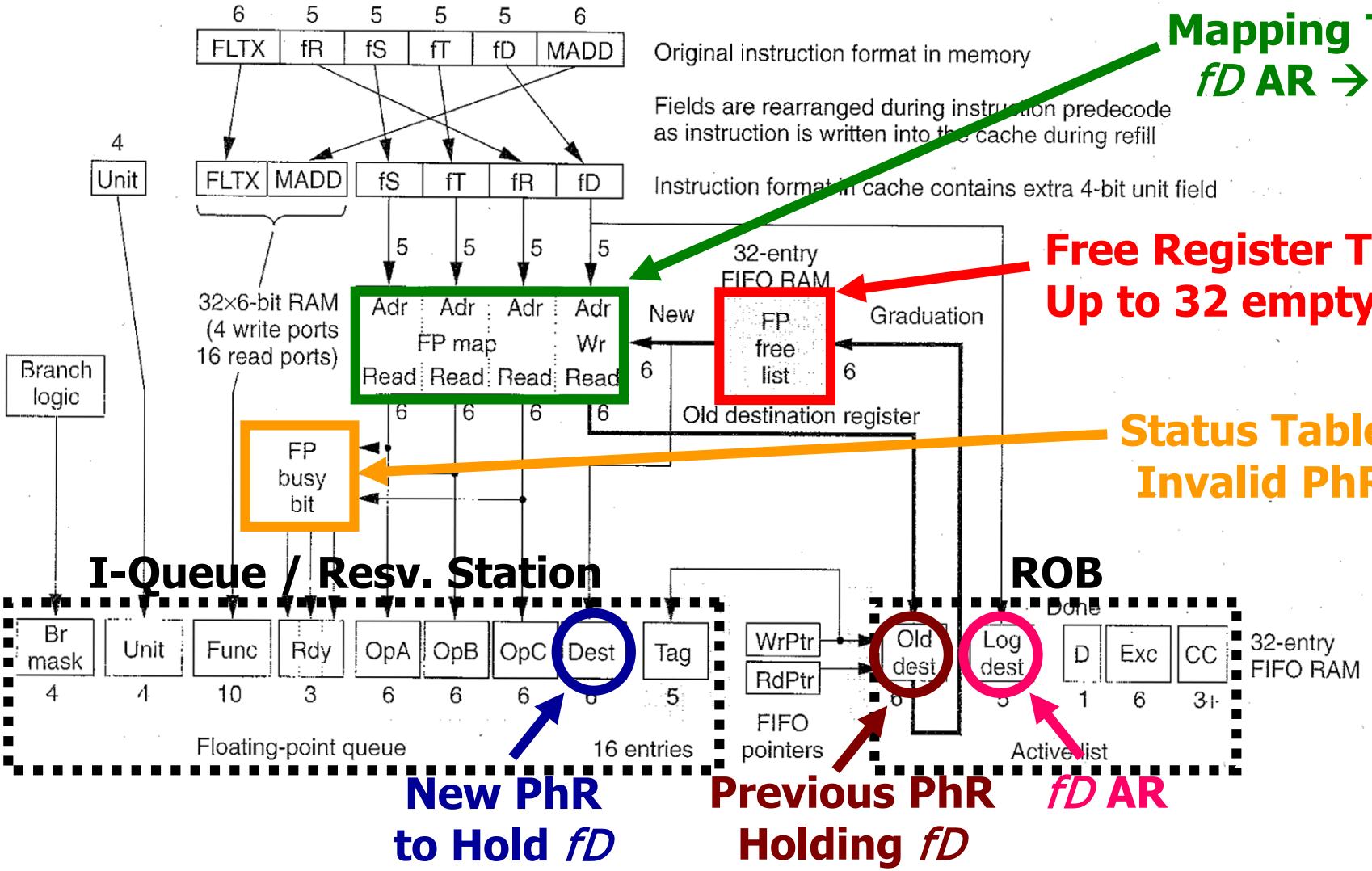

Source: Sima, © IEEE 2000

# State Transitions in a Merged File

- Initialisation:
  - First N registers are "AR", others "Available"

1. *Available* → *Renamed Invalid*
   - Instruction enters the Reservation Stations and/or the ROB: register allocated for the result (i.e., register uninitialised)
2. *Renamed Invalid* → *Renamed Valid*
   - Instruction completes (i.e., register initialised)
3. *Renamed Valid* → *Architectural Register*
   - Instruction commits (i.e., register "exists")
4. *Architectural Register* → *Available*
   - Another instruction commits to the same AR (i.e., register is dead)
5. *Renamed Invalid* and *Renamed Valid* → *Available*
   - Squashing

# MIPS R10000: 32 AR, 64 PhR, Merged Register File




Source: Yeager, © IEEE 1996

# MIPS R10000: Information Flow

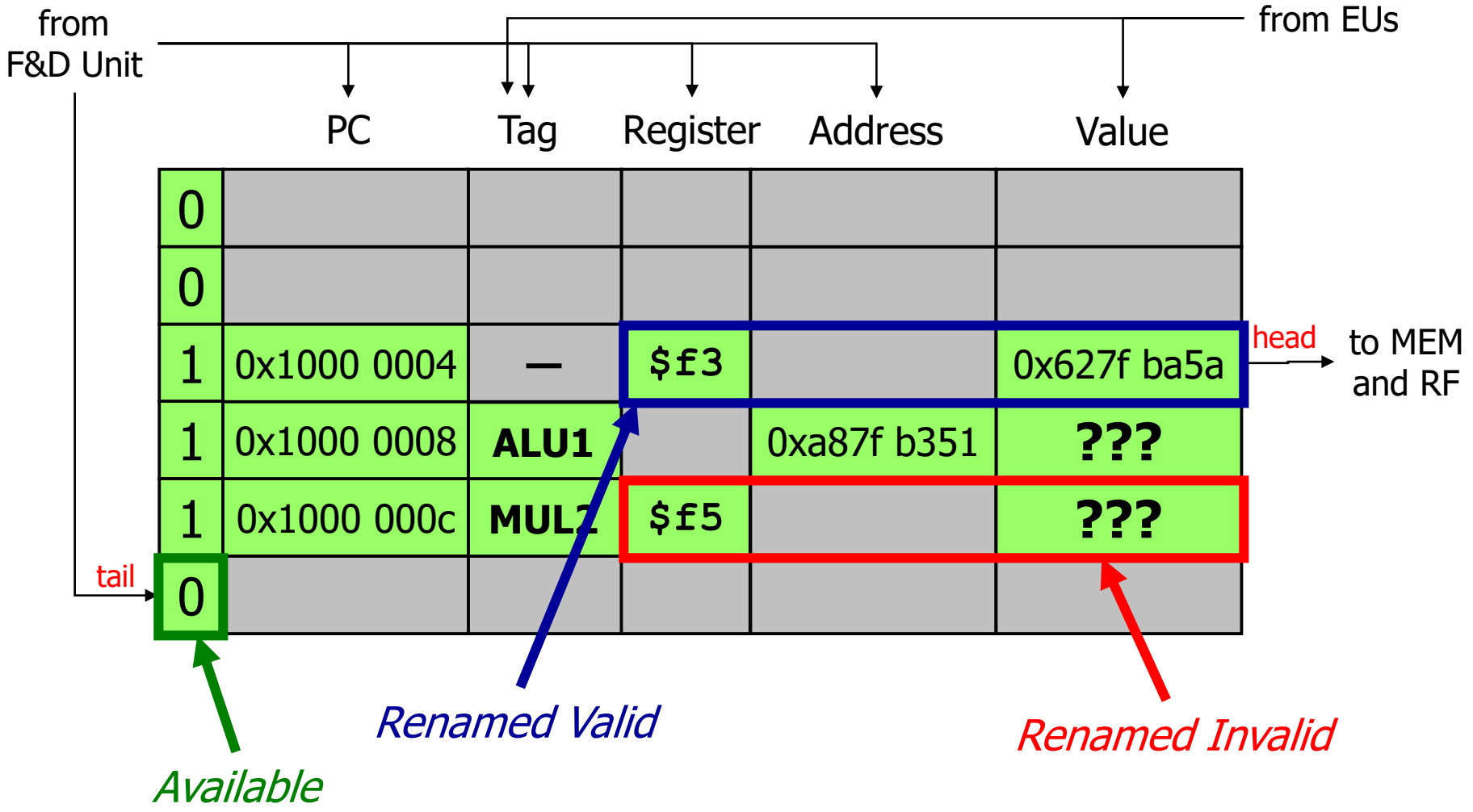
1. *Available* → *Renamed Invalid*
   - Read new PhR from top of Free Register Table
   - Create new mapping *LogDest* → *Dest* in the Mapping Table
   - Set corresponding *Busy-Bit* (=invalid) in the Status Table
2. *Renamed Invalid* → *Renamed Valid*
   - Write PhR *Dest* indicated in the I-Queue
   - Reset corresponding *Busy-Bit* (=valid) in the Status Table
   - Mark as *Done* in the corresponding entry in the ROB
3. *Renamed Valid* → *Architectural Register*
   - Implicit (removal of historical mapping *LogDest* → *Dest*)
4. *Architectural Register* → *Available*
   - Free PhR indicated by *OldDest* in the entry removed from the ROB
5. *Renamed Invalid* and *Renamed Valid* → *Available*
   - Restore mapping from all squashed ROB entries (from tail to head) as *LogDest* → *Dest*
   - Reset corresponding *Busy-Bit* (=valid) in the Status Table

# State Transitions Replaced by Copying in Stand-alone RRF

- Initialisation:
  - All Rename Registers are "Available"

1. *Available* → *Renamed Invalid*
   - Instruction enters the Reservation Stations and/or the ROB: register allocated for the result (i.e., register uninitialised)
2. *Renamed Invalid* → *Renamed Valid*
   - Instruction completes (i.e., register initialised)
3. *Renamed Valid* → *Available*
   - Instruction commits (i.e., register "exists")
   - → Value **is copied** in the Architectural RF
4. *Renamed Invalid* and *Renamed Valid* → *Available*
   - Squashing (no copy to the Architectural RF)

# State of the Rename Registers in the Commit Unit (ROB)



# How Many Rename Registers?

- In-Flight instructions:

$$N_{in-flight} = N_{RS} + N_{EU} + N_{LD} + N_{ST}$$

- Rename Registers:

$$N_{rename} \leq N_{RS} + N_{EU} + N_{LD}$$

- ROB size:

$$N_{ROB} \leq N_{in-flight}$$

Note: if strictly < then structural stalls can occur

# Number of Rename Registers

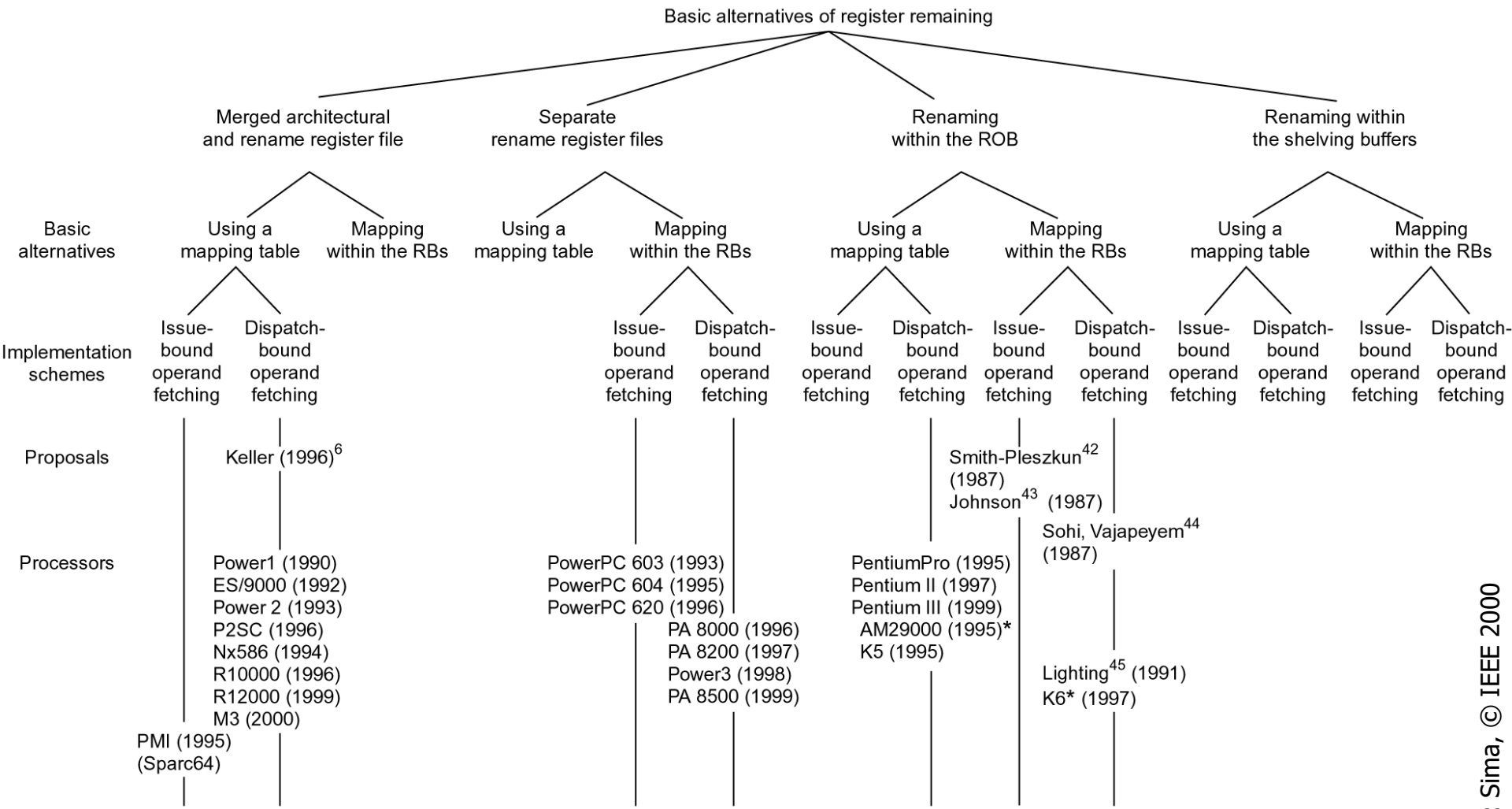
**Type and available number of rename buffers in recent superscalars as well as four related parameters of the enlisted processors.**

| Processor type (year of volume shipment) | Type of rename buffer | No. of rename buffers FX | No. of rename buffers FP | Issue rate | Width of dispatch window (*wdw*) | Total no. of rename buffers (*nr*) | Reorder width (*nROB*) |
|---|---|---|---|---|---|---|---|
| **RISC processors** | | | | | | | |
| PowerPC 603 (1993) | Ren. reg. file | N/A | 4 | 3 | 3 | N/A | 5 |
| PowerPC 604 (1995) | Ren. reg. file | 12 | 8 | 4 | 12 | 20 | 16 |
| PowerPC 620 (1996) | Ren. reg. file | 8 | 8 | 4 | 15 | 16 | 16 |
| Power3 (1998) | Ren. reg. file | 16 | 24 | 4 | 20 (?) | 40 | 32 |
| R10000 (1996) | Merged | 32 | 32 | 4 | 48 | 64 | 32 |
| R12000 (1998) | Merged | 32 | 32 | 4 | 48 | 64 | 48 |
| Alpha 21264 (1998) | Merged | 48 | 41 | 4 | 35 | 89 | 80 |
| PA 8000 (1986) | Ren. reg. file | 56 | 56 | 4 | 56 | 112 | 56 |
| PM1 (1996) | Merged | 38 | 24 | 4 | 36 | 62 | 62 |
| **x86 (CISC) processors** | | | | | | | |
| Pentium Pro (1995) | In the ROB | 40 | | 3[2] | 20[1] | 40 | 40[1] |
| Pentium II (1997) | In the ROB | 40 | | 3[2] | 20[1] | 40 | 40[1] |
| K5 (1995) | In the ROB | 16 | | 4[2] | 11[1] (?) | 16 | 16[1] |
| K6 (1996) | In the ROB | 24 | | 3[2] | 24[1] | 24 | 24[1] |
| M3 (2000 expected) | Merged | 32 | N/A | 3[2] | 56[1] | N/A | 32[2] |

[1] RISC operations
[2] x86 instructions (on average, produce 1.3 to 1.9 RISC operations[34])
? Questionable data
N/A Not available

Source: Sima, © IEEE 2000

# Actual Choices in Commercial Implementations




*The shelving buffers are also implemented in the ROB. The resulting unit is occasionally called the DRIS.

Source: Microprocessor Report, © Cahners 2009

# High-End Processors in 2009

| Processor | Intel 1-core Xeon | AMD 1-core Opteron 854 | Intel 2-core Xeon X5270 [1] | AMD 2-core Opteron 8224SE | Intel 4-core Xeon X7350 [2] | AMD 4-core Opteron 8360SE [3] | Intel 6-core Xeon X7460 [4] |
|---|---|---|---|---|---|---|---|
| Bit-width | 32/64-bit | 32/64-bit | 32/64-bit | 32/64-bit | 32/64-bit | 32/64-bit | 32/64-bit |
| Cores/chip x Threads/core | 1 x 2 | 1 x 1 | 2 x 1 | 2 x 1 | 4 x 1 | 4 x 1 | 6 x 1 |
| Clock Rate | 3.80GHz | 2.80GHz | 3.50GHz | 3.20GHz | 2.93GHz | 2.50GHz | 2.67GHz |
| Cache: L1-L2-L3 - I/D or Unified | 12K/16K - 2M - N/A | 64K/64K - 1M - N/A | 2 x 32K/32K - 6M - NA | 2 x 64K/64K - 2 x 1M - N/A | 4 x 32K/32K - 2 x 4M - N/A | 4 x 64K/64K - 4 x 512K - 2M | 6 x 32K/32K - 3 x 3M - 16M |
| Execution Rate/Core | 3 Instructions | 3 Instructions | 1 Complex + 3 Simple | 3 Instructions | 1 Complex + 3 Simple | 3 Instructions | 1 Complex + 3 Simple |
| Pipeline Stages | 31 | 12 int / 17 fp | 14 | 12 int / 17 fp | 14 | 12 int / 17 fp | 14 |
| Out of Order | 126 | 72 | 96 | 72 | 96 | 72 | 96 |
| Memory Bus | 800MHz | 6.4GB/s | 1333MHz | 10.6GB/s | 1066MHz | 10.6GB/s | 1064MHz |
| Package | LGA-775 | uPGA 940 | LGA-771 | LGA-1207 | LGA-771 | LGA-1207 | LGA-771 |
| IC Process | 90nm 7M | 90nm 9M | 45nm | 90nm 9M | 65nm 8M | 65nm 11M | 45nm |
| Die Size | 109mm$^2$ | 106 mm$^2$ | 107mm$^2$ | 227mm$^2$ | 2 x 143mm$^2$ | 283mm$^2$ | 503mm$^2$ |
| Transistors | 169M | 120M | 410M | 233M | 2 x 291M | 463M | 1900M |
| List Price (Intro) | $903 | $1,514 | $1,172 | $2,149 | $2,301 | $2,149 | $2,729 |
| Power (Max) | 110W | 93W | 80W | 120W | 130W | 105W | 130W |
| Availability | 3Q05 | 3Q05 | 3Q08 | 3Q07 | 3Q07 | 2Q08 | 4Q08 |
| Scalability | 1–2 Chips | 2–4 Chips | 1–2 Chips | 1–4 Chips | 1–4 Chips | 2–4 Chips | 1–4 Chips |
| SPECint/fp2006 [Cores] | 11.4/11.7 [2] | 11.2/12.1 [2] | 26.5$^x$/25.5$^x$ [4] | 14.1/14.2 [8] | 21.7$^y$/18.9$^y$ [16] | 14.4$^x$/18.5$^x$ [8] | 22.0$^x$/22.3$^x$ [24] |
| SPECint/fp2006_rate [Cores] | 20.9/18.8 [2] | 41.4/45.6 [4] | 84.9$^x$/57.7$^x$ [4] | 105/96.7 [8] | 184$^y$/108 [16] | 170$^x$/156$^x$ [16] | 274$^x$/142$^x$ [24] |
| x86 Codename | Irwindale | Athens | Wolfdale | Santa Rosa | Tigerton | Barcelona | Dunnington |
| Microarchitecture | Netburst | K8 | Core | K8 | Core | K10 | Core |

| Processor | Intel Itanium 2 9050 | Intel Itanium 9150M | IBM POWER5+ | IBM POWER6 | Fujitsu SPARC64 VI | Fujitsu SPARC64 VII | Sun UltraSPARC T2+ |
|---|---|---|---|---|---|---|---|
| Bit-width | 64-bit | 64-bit | 64-bit | 64-bit | 64-bit | 64-bit | 64-bit |
| Cores/Chip x Threads/Core | 2 x 2 | 2 x 2 | 2 x 2 | 2 x 2 | 2 x 2 | 4 x 2 | 8 x 8 |
| Clock Rate | 1.60GHz | 1.67GHz | 2.20GHz | 5.00GHz | 2.40GHz | 2.52GHz | 1.40GHz |
| Cache: L1-L2-L3 - I/D or Unified | 2 x 16K/16K - 1M/256K - 12M(on) | 2 x 16K/16K - 1M/256K - 12M(on) | 2 x 64K/32K 1.92M - 36M(off) | 2 x 64K/64K - 2 x 4M - 32M(off) | 2 x 128K/128K - 6M - N/A | 4 x 64K/64K - 6M - N/A | 8 x 8K/16K - 4M - NA |
| Execution Rate/Core | 6 Issue | 6 Issue | 5 Issue | 7 Issue | 4 Issue | 4 Issue | 16 Issue |
| Pipeline Stages | 8 | 8 | 15 | 13 | 15 | 15 | 8 int / 12 fp |
| Out of Order | None | None | 200 | Limited | 64 | 64 | None |
| Memory Bus | 8.5GB/s | 10.6GB/s | 12.8GB/s | 75GB/s | 8GB/s | 8GB/s | 42.7GB/s |
| Package | mPGA-700 | mPGA-700 | MCM-5370 Pins | N/A | 412 I/O Pins | 412 I/O Pins | 1831 Pins |
| IC Process | 90nm 7M | 90nm 7M | 90nm 10M | 65nm 10M | 90nm 10M | 65nm 11M | 65nm |
| Die Size | 596mm$^2$ | 596mm$^2$ | 245mm$^2$ | 341mm$^2$ | 421mm$^2$ | 400mm$^2$ | 342mm$^2$ |
| Transistors | 1.72B | 1.72B | 276M | 790M | 540M | 600M | 503M |
| List Price (Intro) | $3,692 | $3,692 | N/A | N/A | N/A | N/A | N/A |
| Power (Max) | 104W | 104W | 100W | >100W | 120W | 135W | 95W |
| Availability | 3Q06 | 4Q07 | 4Q05 | 2Q08 | 2Q07 | 3Q08 | 2Q08 |
| Scalability | 1–64 Chips | 8–128 Chips | 1–32 Chips | 2–32 Chips | 4–64 Chips | 4–64 Chips | 2 Chips |
| SPECint/fp2006 [Cores] | 14.5/17.3 [2] | N/A | 10.5/12.9 [1] | 15.8*/20.1 [1] | 9.7/21.7$^y$ [32] | 10.5$^x$/25.0$^x$ [64] | N/A |
| SPECint/fp2006_rate [Cores] | 1534/1671 [128] | 2893/N/A [256] | 197/229 [16] | 1837$^x$/1822 [64] | 1111/1160 [128] | 2088$^x$/1861$^x$ [256] | 142/111 [16] |
| Architecture Status | Inactive | Active | Inactive | Active | Inactive | Active | Active |

All SPEC scores are base. * Score measured at 4.20GHz (not 5.00GHz).

**No renaming only in UltraSparc:** Use of register windows made it very difficult to implement renaming (but Fujitsu eventually managed)

**Nor in Itanium, of course…**

# References on Register Renaming

- AQA 5th ed., Appendix C and Chapter 3
- PA, Sections 6.3, 6.4, and 6.5
- CAR, Chapter 5—Introduction
- D. Sima, *The Design Space of Register Renaming Techniques*, IEEE Micro, (20):5, Sept.-Oct. 2000
- K. C. Yeager, *The MIPS R10000 Superscalar Microprocessor*, IEEE Micro, 16(2):28-40, April 1996

# 3

## Prediction and Speculation

## (Don’t Know It? Don’t Wait but Guess…)

# Prediction & Speculation: The Idea

- Some operation takes awfully long?
- The processor needs the result to proceed?
  - To fetch the next instruction, one needs to know which one must be fetched
  - To perform a computation, one needs the operands

**Don't wait!!!**

1. **Make a guess** (➔ *Predict*) and
2. **Proceed tentatively** (➔ *Speculate*)

# General Problems

1. How do I make a good guess?
   - Either one outcome is typical and far more frequent
     - *Static* prediction
   - Or I need to remember some history
     - *Dynamic* prediction

2. What do I do if the guess was wrong?
   - Undo speculatively-executed instructions ("squash")
     - May cost nothing—e.g.,
       - Squash some results
     - May cost something—e.g.,
       - Empty pipelines
       - Restore saved state
       - Execute compensation code

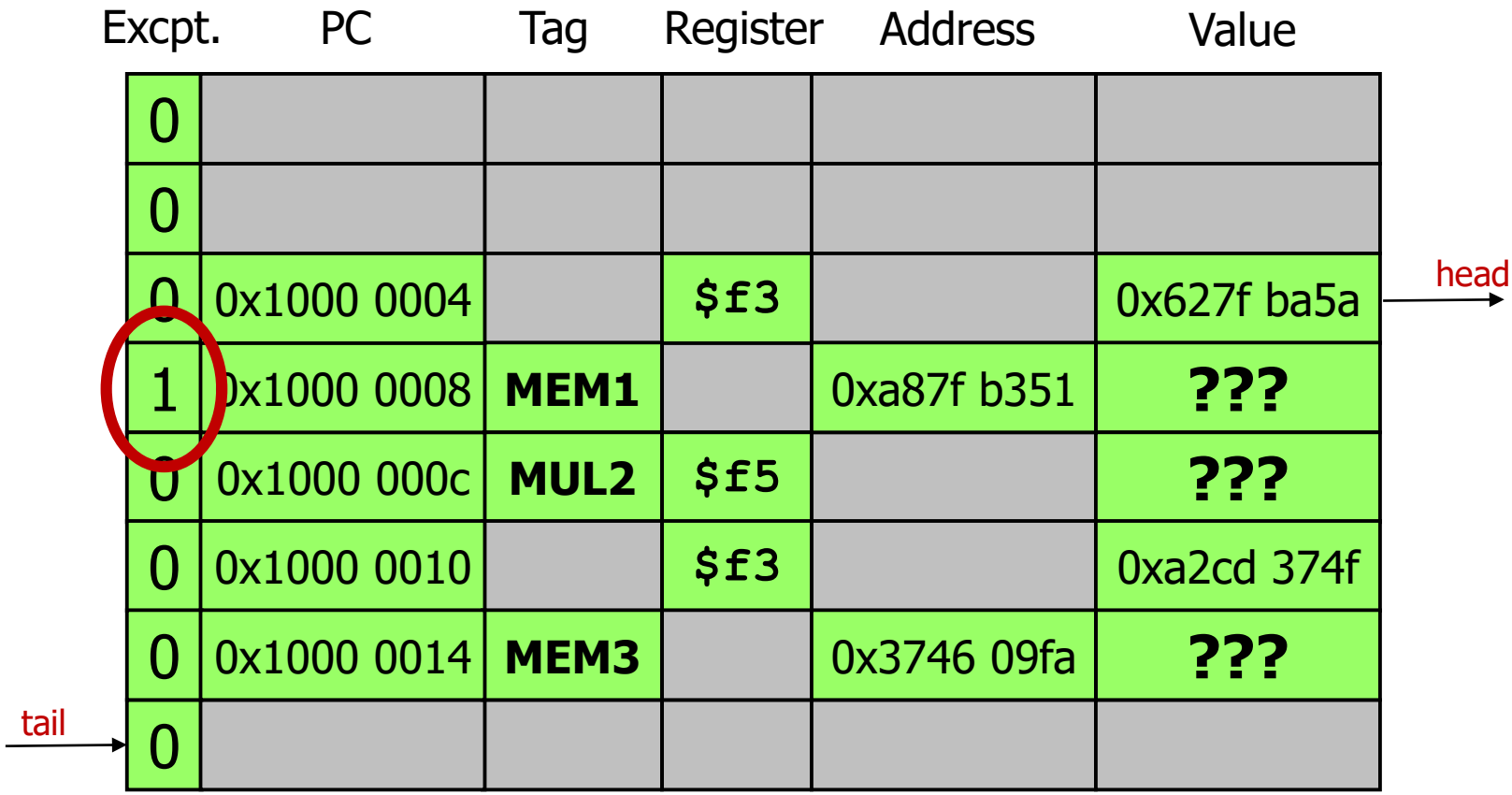
# Prediction & Speculation

- Have we already seen a form of prediction and speculation in this course?

**Precise exceptions**

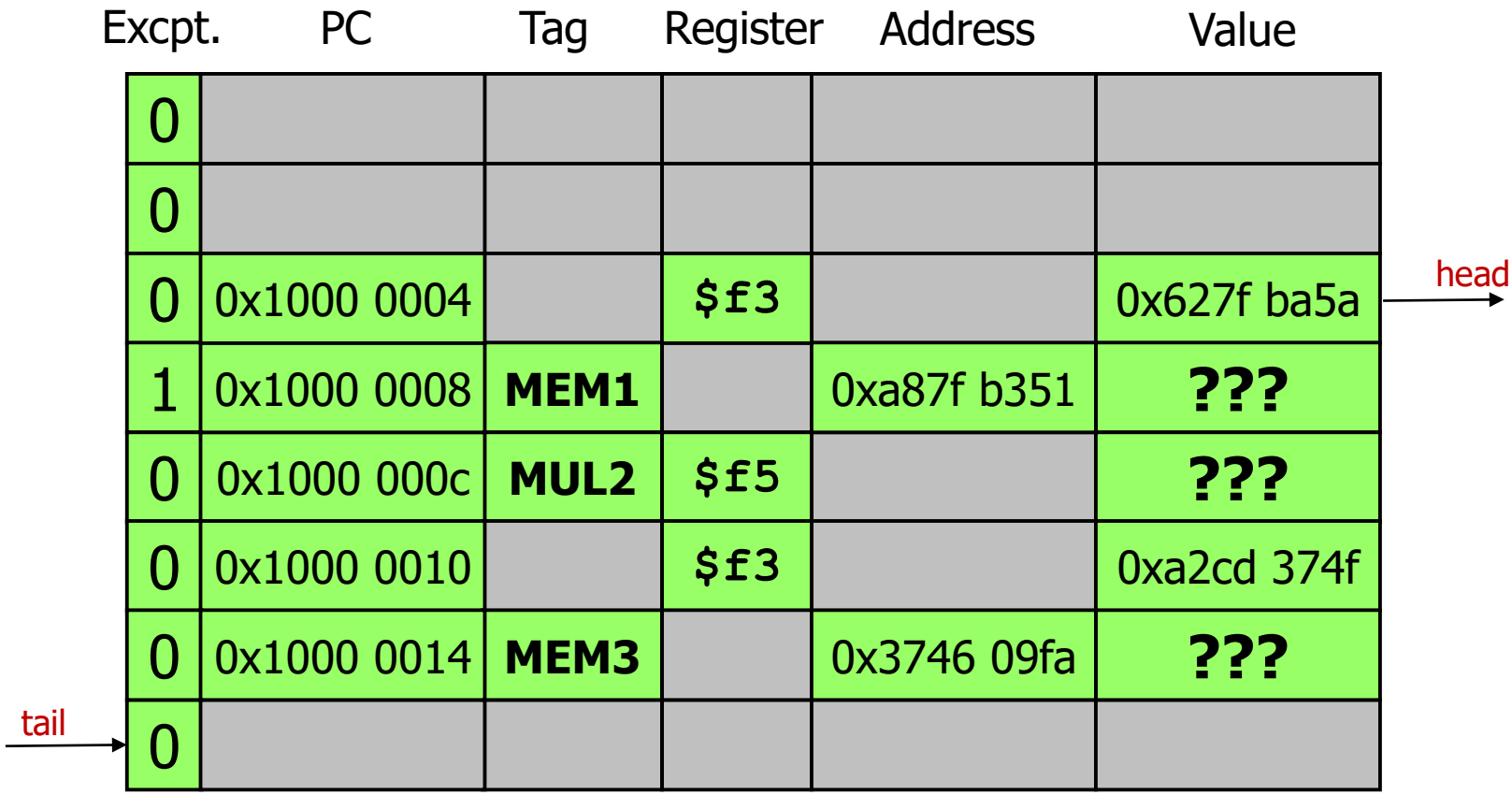
- **Prediction:** For every instruction, we have guessed that there will be no exception (static prediction)
- **Speculation:** In case of exception we have used the ROB to squash all instructions after the faulty one raising the exception

# Precise Exceptions (1/4)



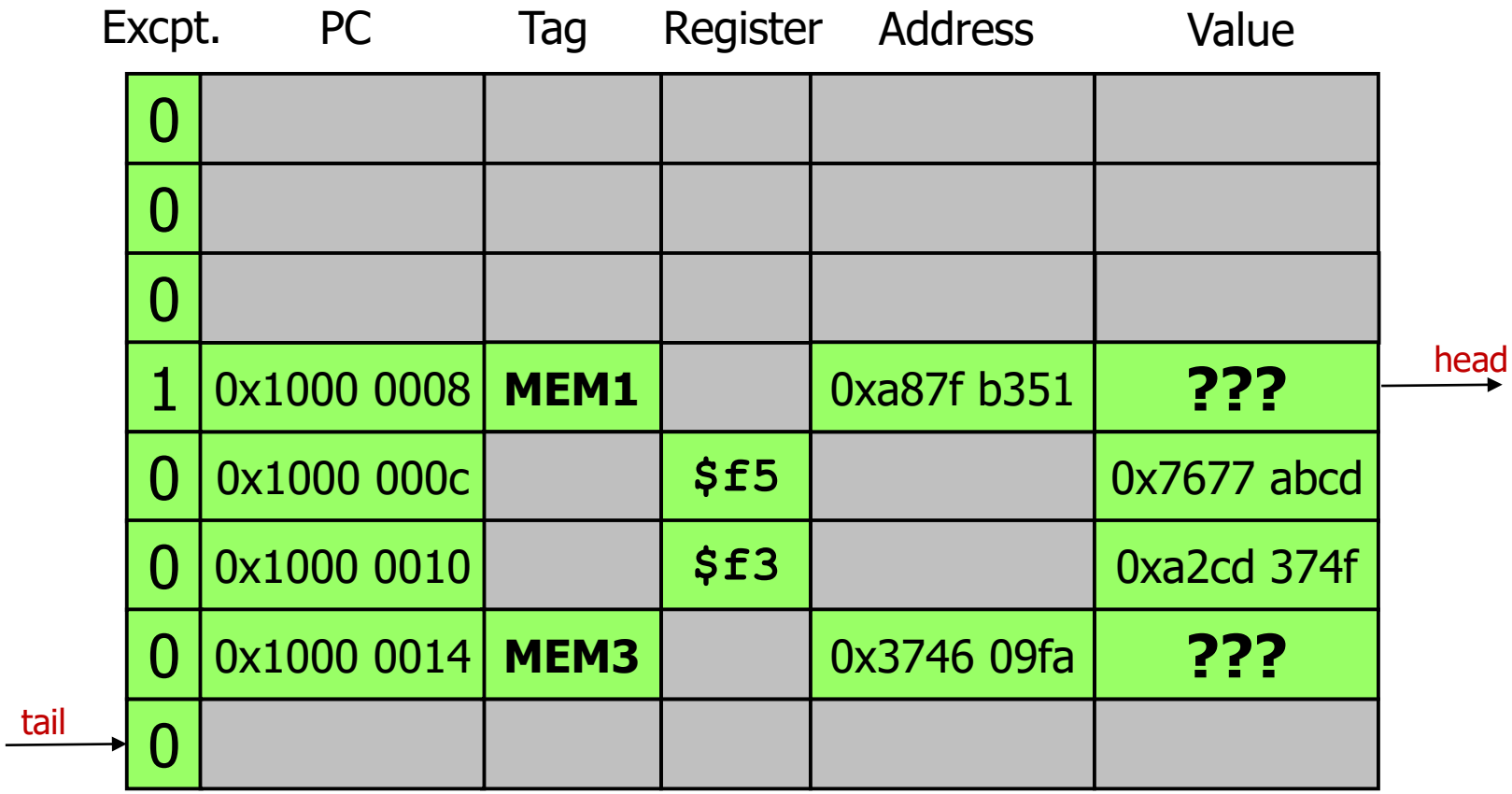

The store **MEM1** results in a *TLB Miss* → We simply **record it**

# Precise Exceptions (2/4)



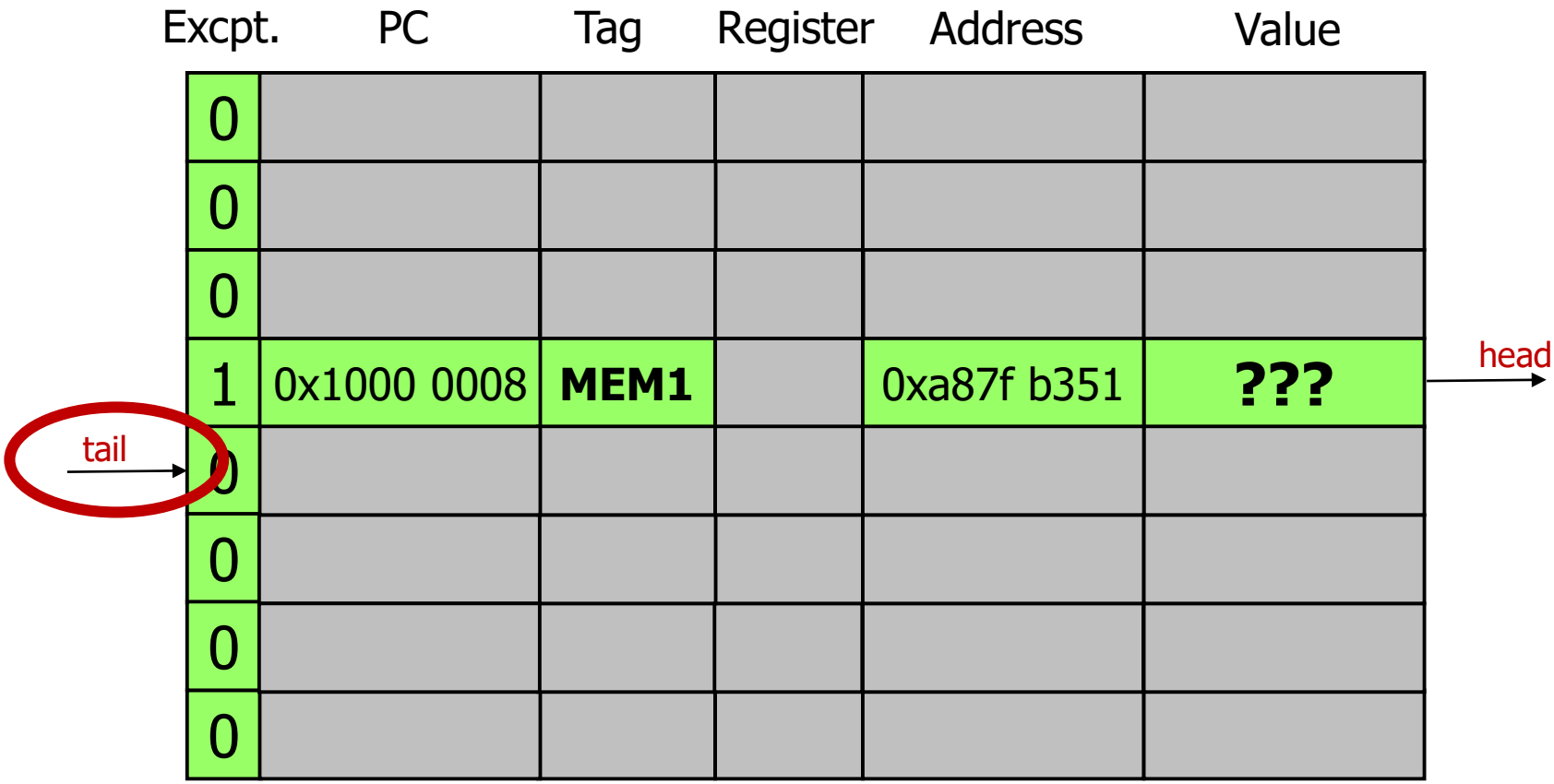

Write `0x627fba5a` to register `$f3` as if nothing happened

# Precise Exceptions (3/4)




**Now** raise the *TLB Miss* exception at location `0x10000008`

# Precise Exceptions (4/4)




But also **squash** all instructions which followed the exception

# A General Idea

- After a prediction, hold every potential change in state of the processor (e.g., register values, memory writes) in a buffer
- If the prediction turns out to be correct, let the content of the buffer affect the state (= COMMIT)
- If the prediction turns out to be wrong, simply trash the content of the buffer (step 4 above)

**Our ROB does just that!**

…and once we have it, we can do much more with it!

# Prediction & Speculation

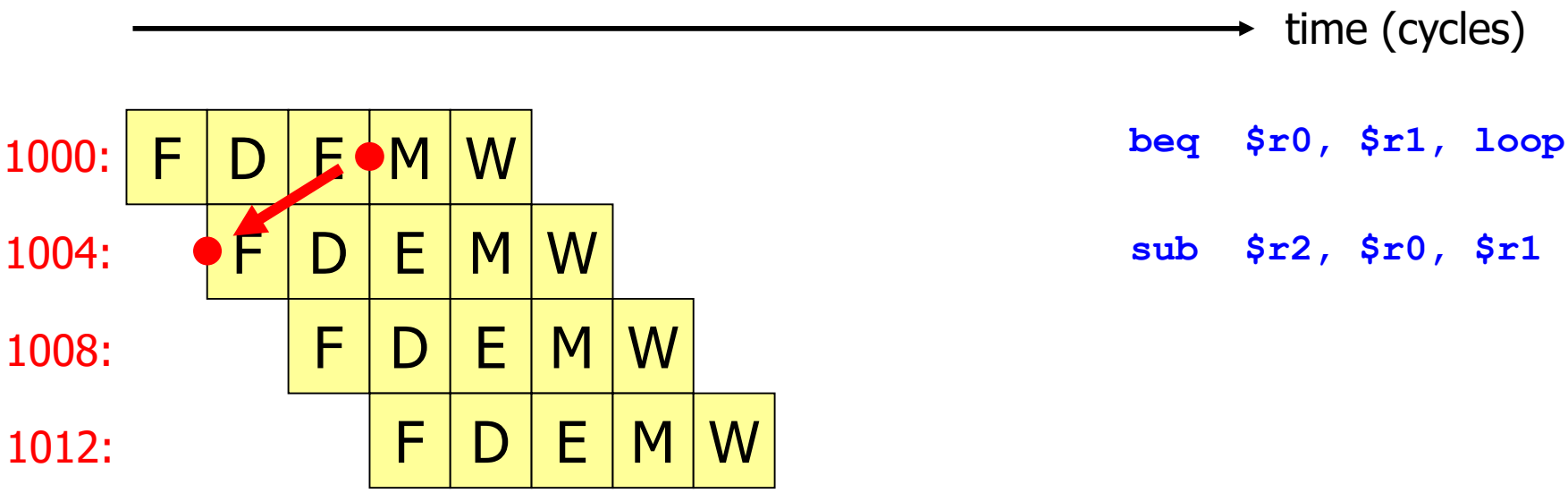
- So far:
  - Precise exceptions

What's next?

# Branch Prediction and Speculation

- **Prediction**
  - *Static:* Maybe we can assume that every backward branch is part of a loop and thus usually taken
  - *Dynamic:* Maybe we can observe what happens during execution and learn
- **Speculation**
  - In a simple pipeline we may simply fetch and decode instructions ➔ easy, no state changes
  - In a complex OOO superscalar we may really execute instructions speculatively ➔ ROB

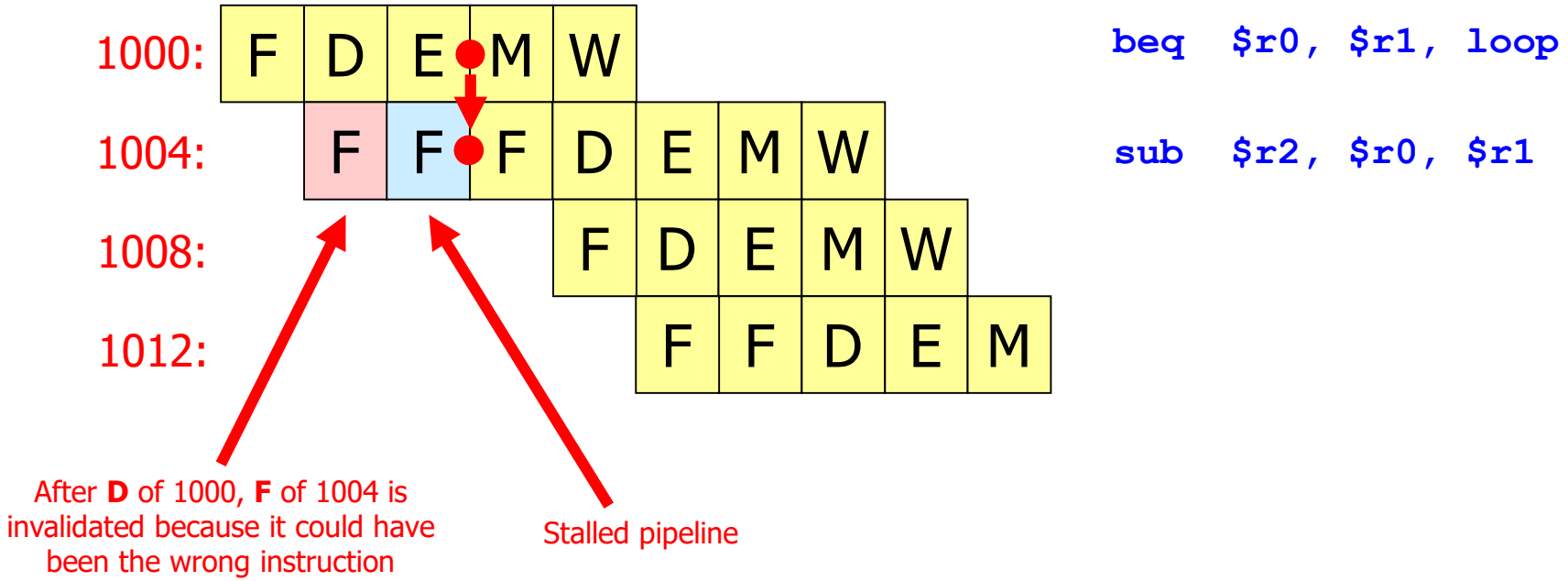
# Control Hazards




**Causality violation!**

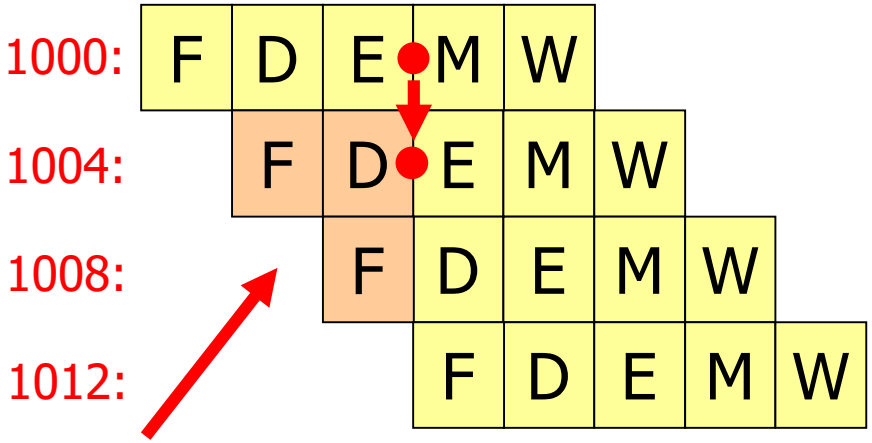
We fetch an instruction before we know which one!

# Control Hazards Solved by Stalling the Pipeline



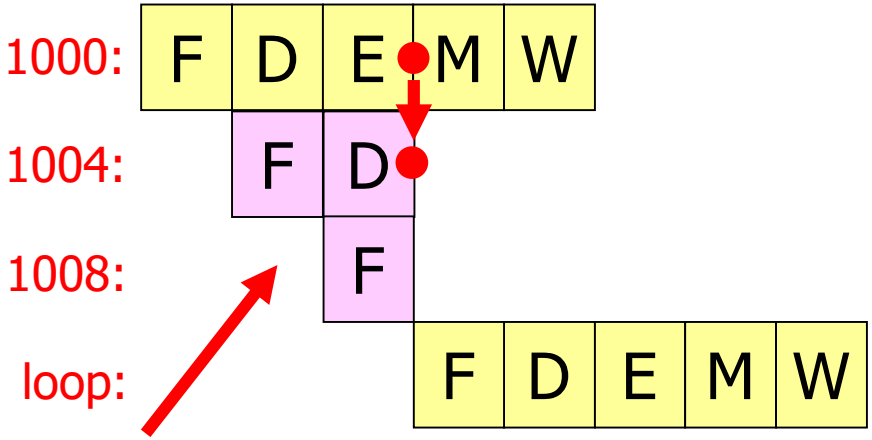

- We can stall the pipeline once it is discovered, after **D**, that an instruction was a branch
- If, for instance, the correct address of the next instruction is known at the end of the **E** stage, 2 cycles are lost **every branch**

# Speculative Fetch and Decode




```
beq  $r0, $r1, loop
sub  $r2, $r0, $r1
```

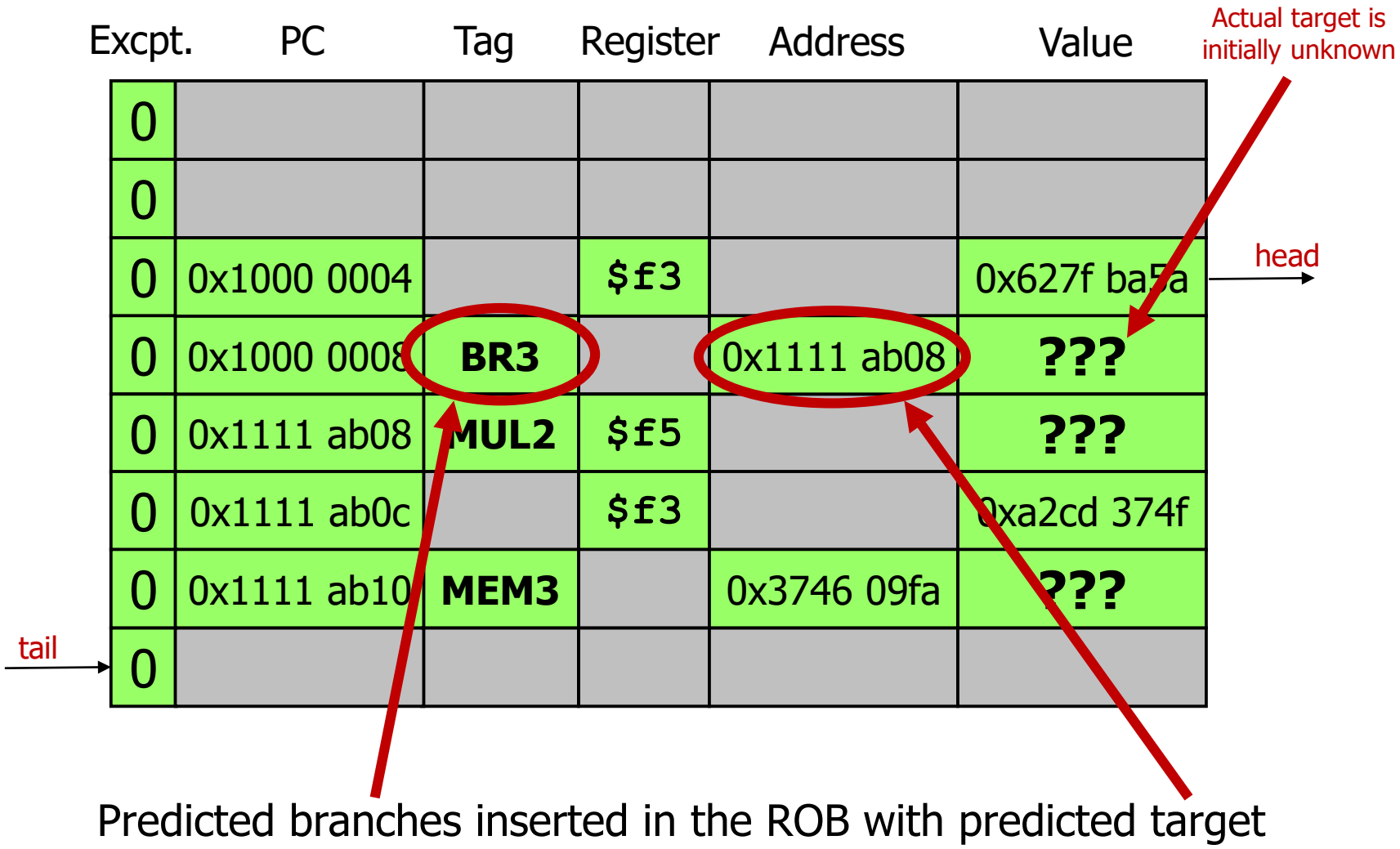
Speculative **F** and **D**, correctly predicted



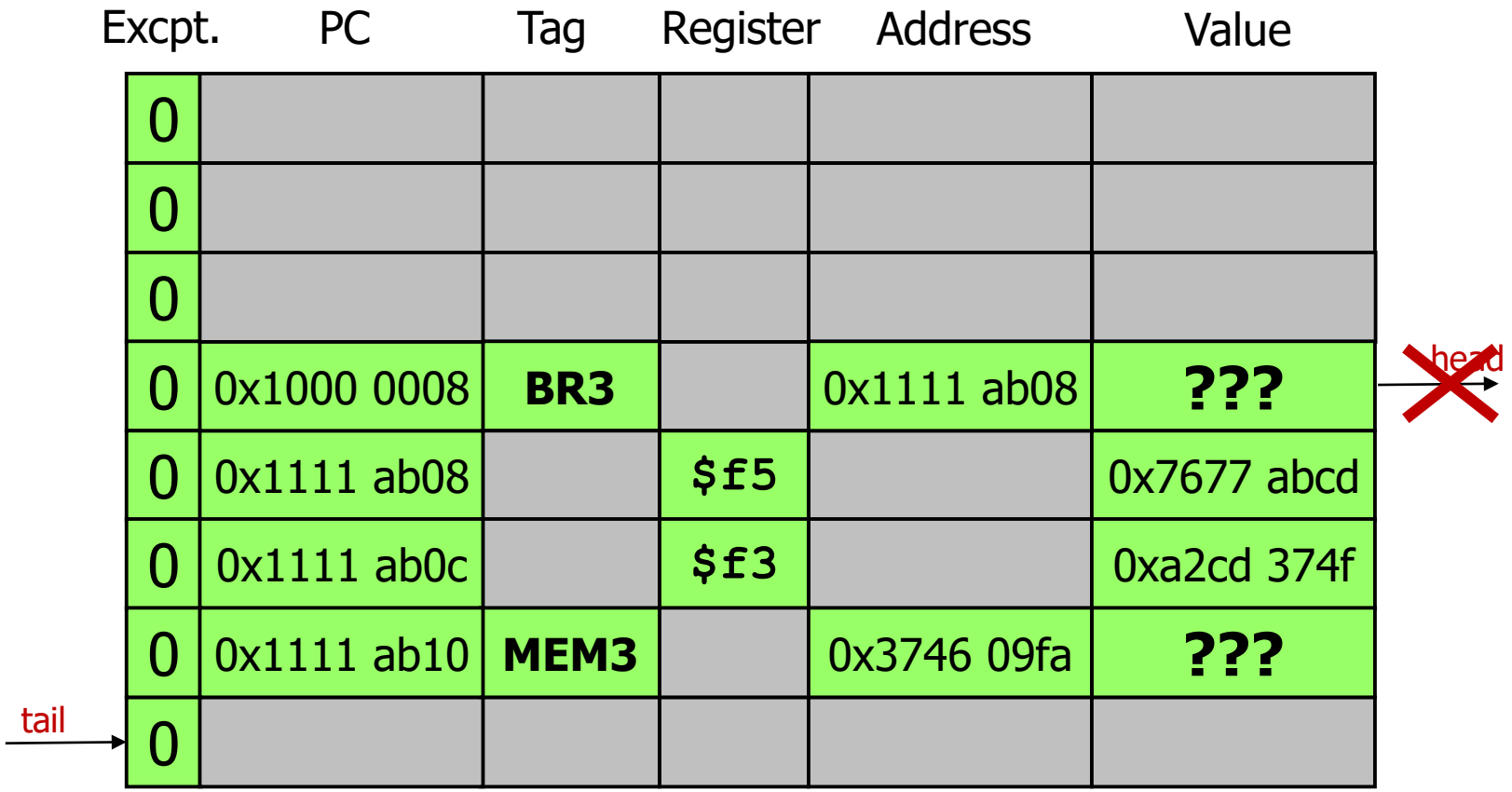

```
beq  $r0, $r1, loop
sub  $r2, $r0, $r1
```

Speculative **F** and **D**, wrongly predicted and thus squashed (simple invalidation in the pipeline)

# Branches in the ROB

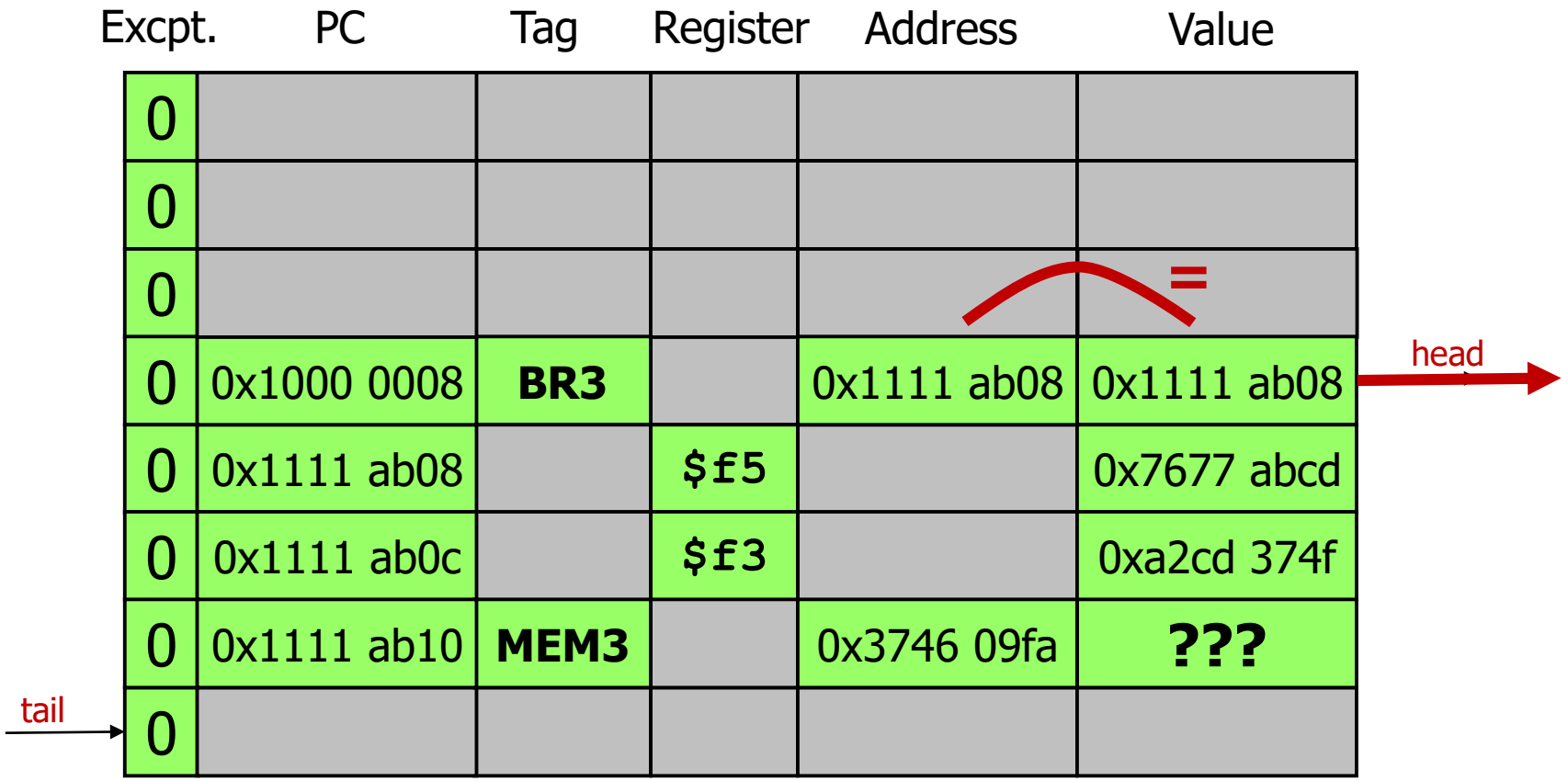


# Branches without Outcome Block the ROB



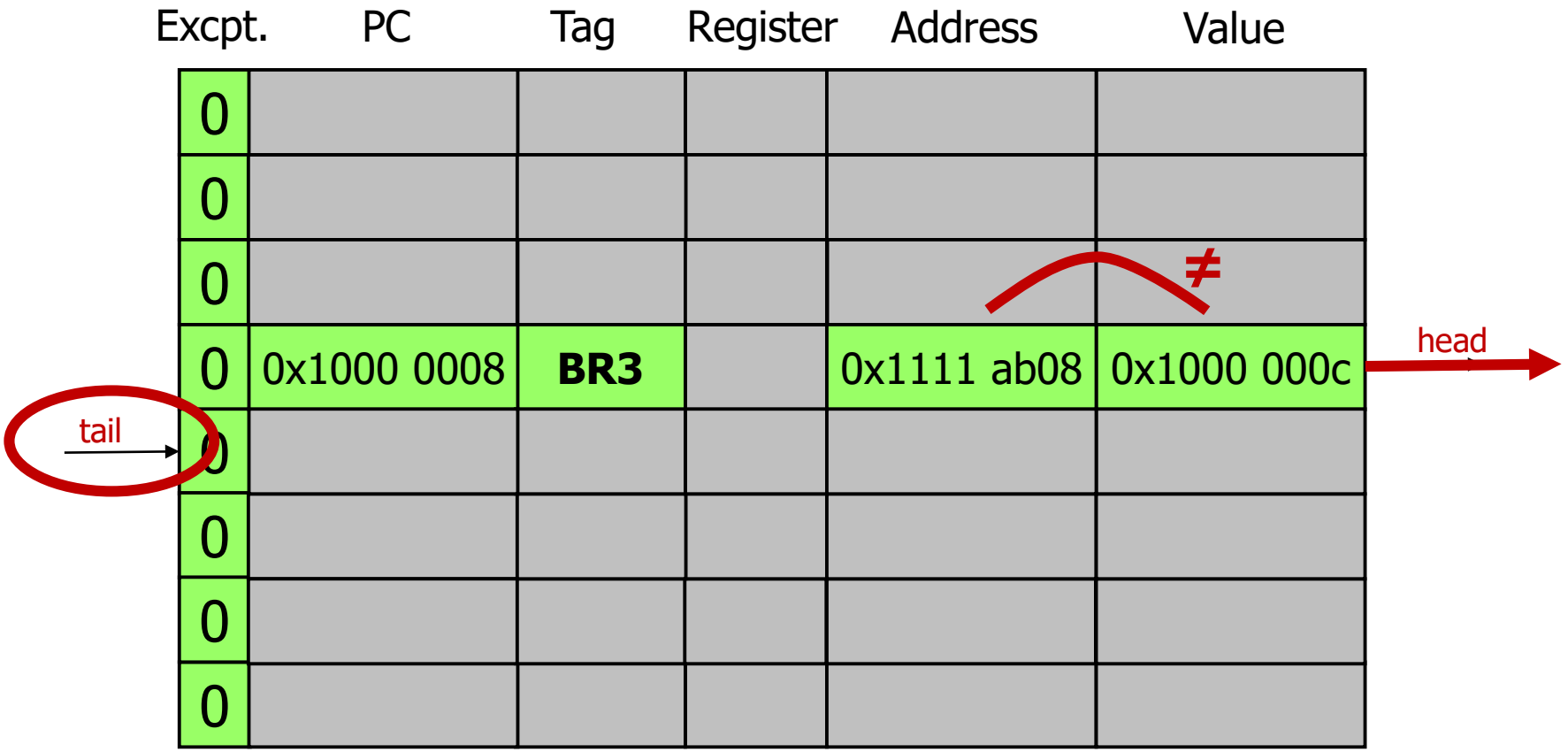

A predicted branch whose outcome is **unknown** cannot be committed

# Correctly Predicted Branches Are Ignored




**BR3** can commit (= do nothing and remove from ROB)

# Mispredicted Branches Trigger a Squash

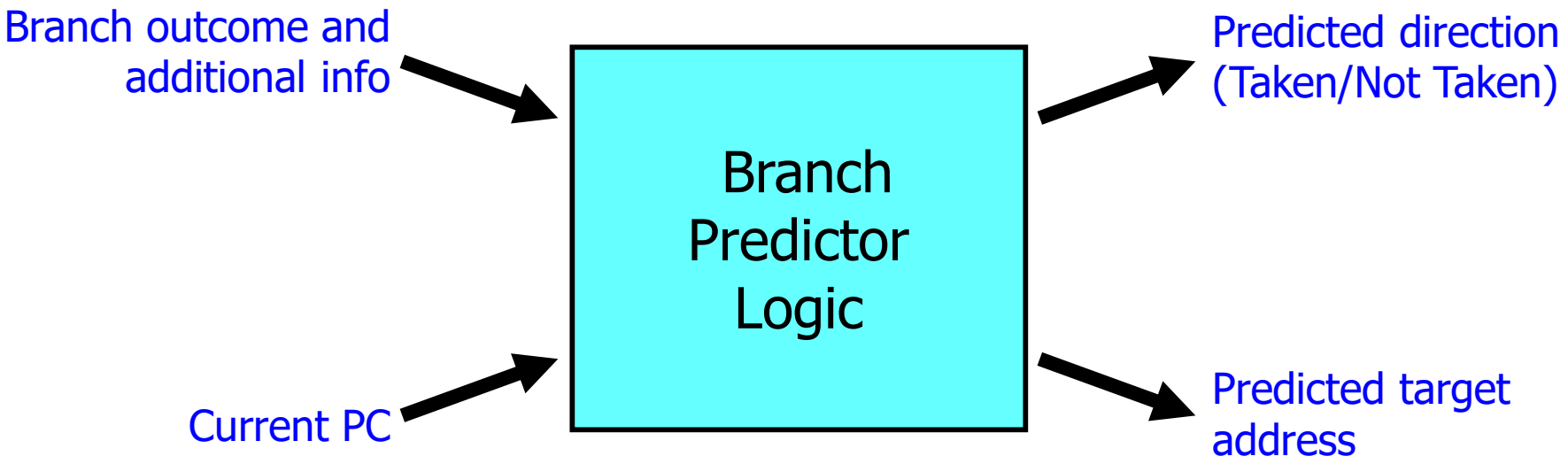



**BR3** triggers a **squash** and causes **fetch** to restart at `0x1000000c`

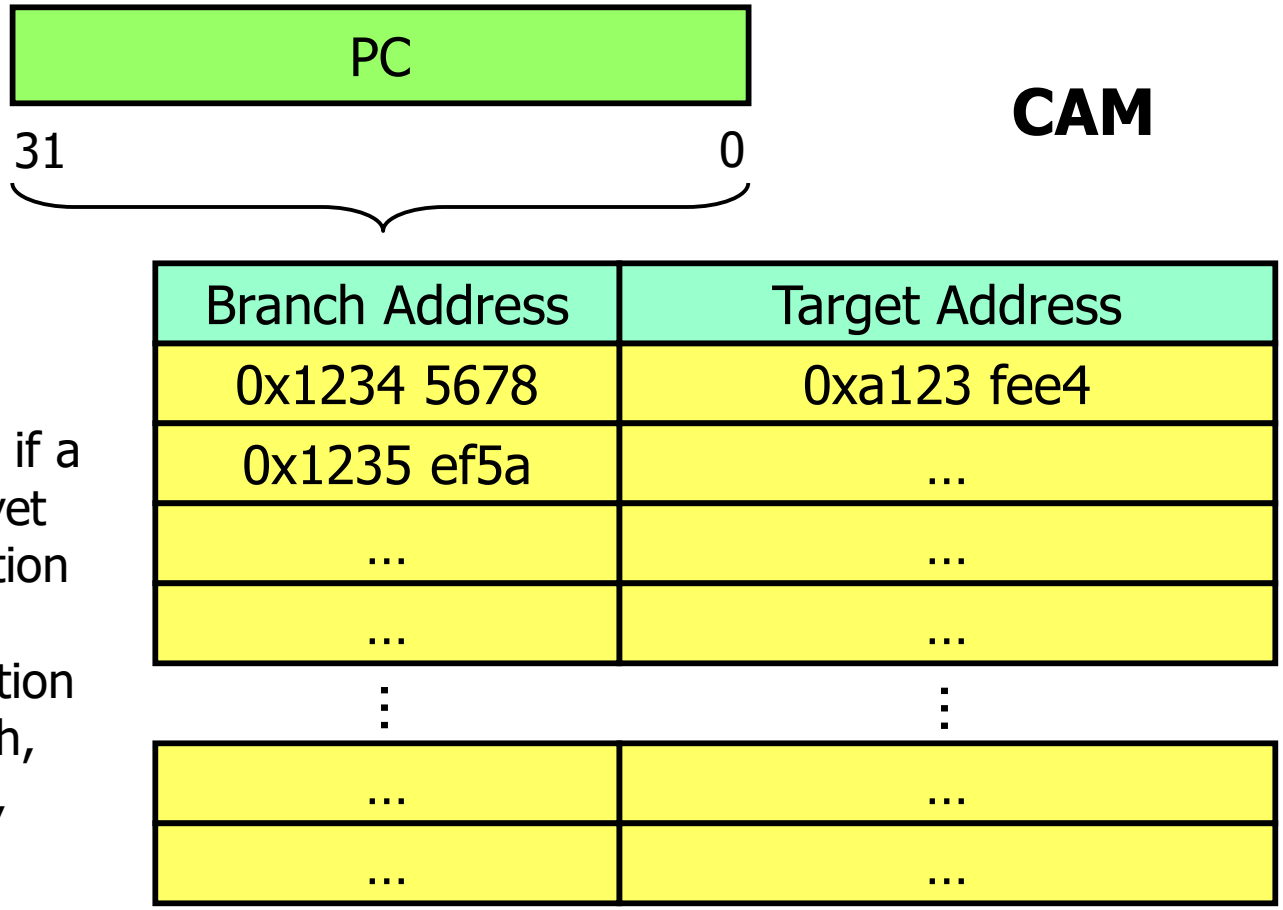
# Branch Prediction and Speculation

- **Prediction**
  - *Static:* Maybe we can assume that every backward branch is part of a loop and thus usually taken
  - *Dynamic:* Maybe we can observe what happens during execution and learn
- **Speculation**
  - In a simple pipeline we may simply fetch and decode instructions ➔ easy, no state changes
  - In a complex OOO superscalar we may really execute instructions speculatively ➔ ROB

# Branch Prediction

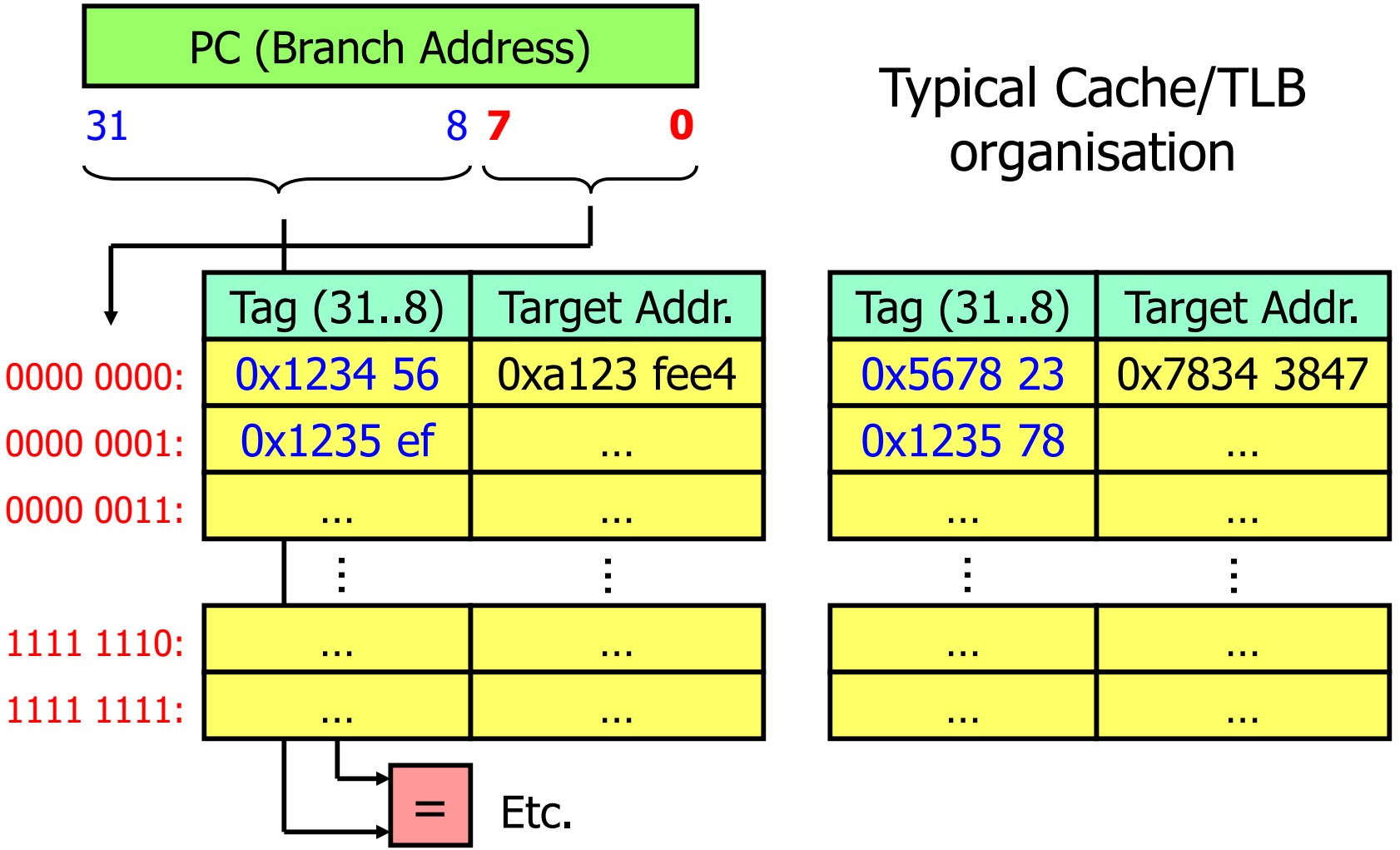


# Branch Target Buffers




One needs to know if a just fetched and yet undecoded instruction is a branch and what is the destination (computed branch, relative address, return, etc.)

# More Complex But Cheaper Branch Target Buffers



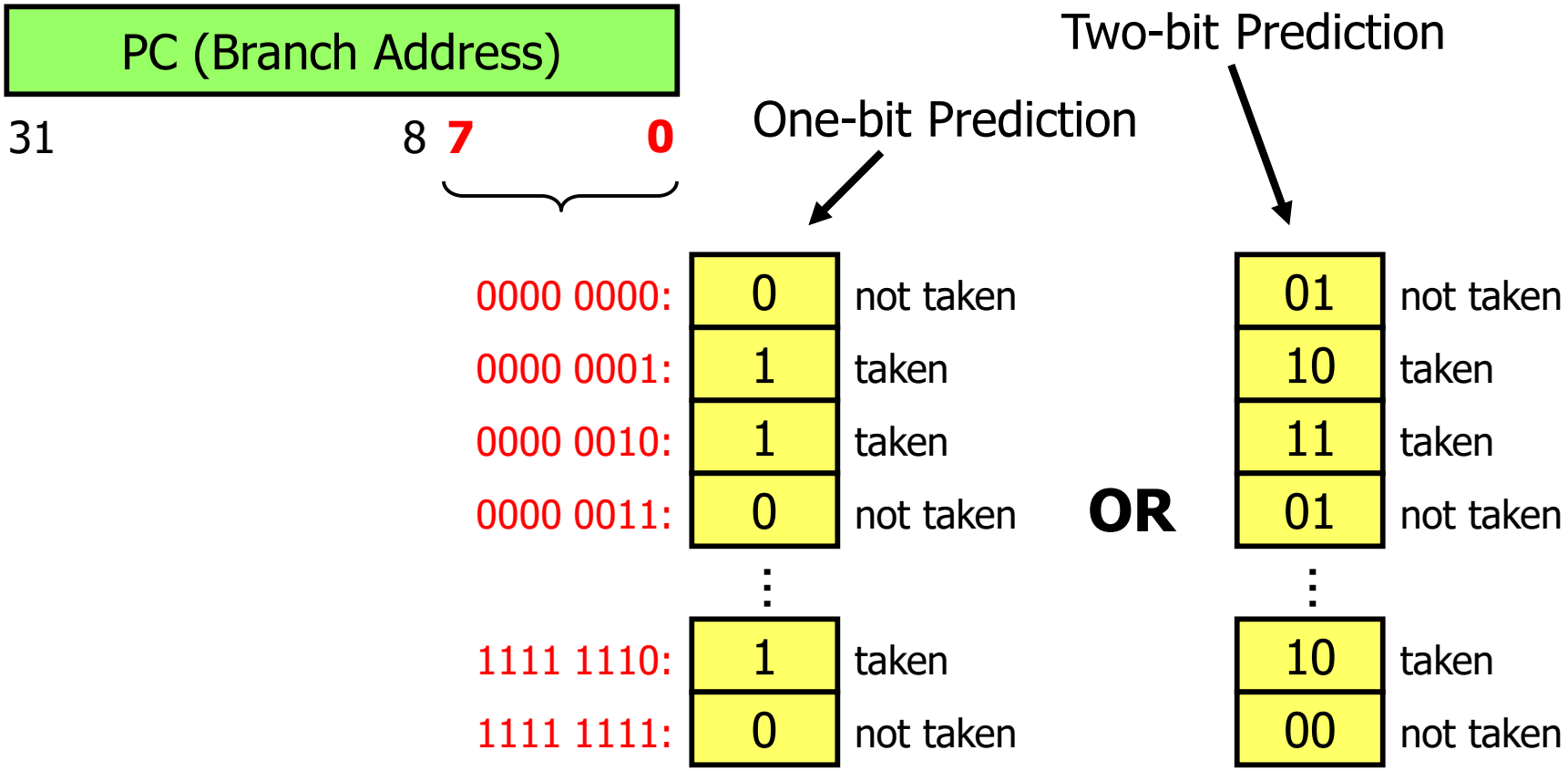
# Which Strategy to Predict?

- **Static** predictions: ignore history
  1. Never-taken or always-taken
  2. Always-taken-backward (e.g., loops)
  3. Compiler-specified, etc.
  - Still a form of **dynamic** control speculation, because the squashing process is done in hardware
- **Dynamic** prediction: learn from history
  - Record how often a branch was taken in the past

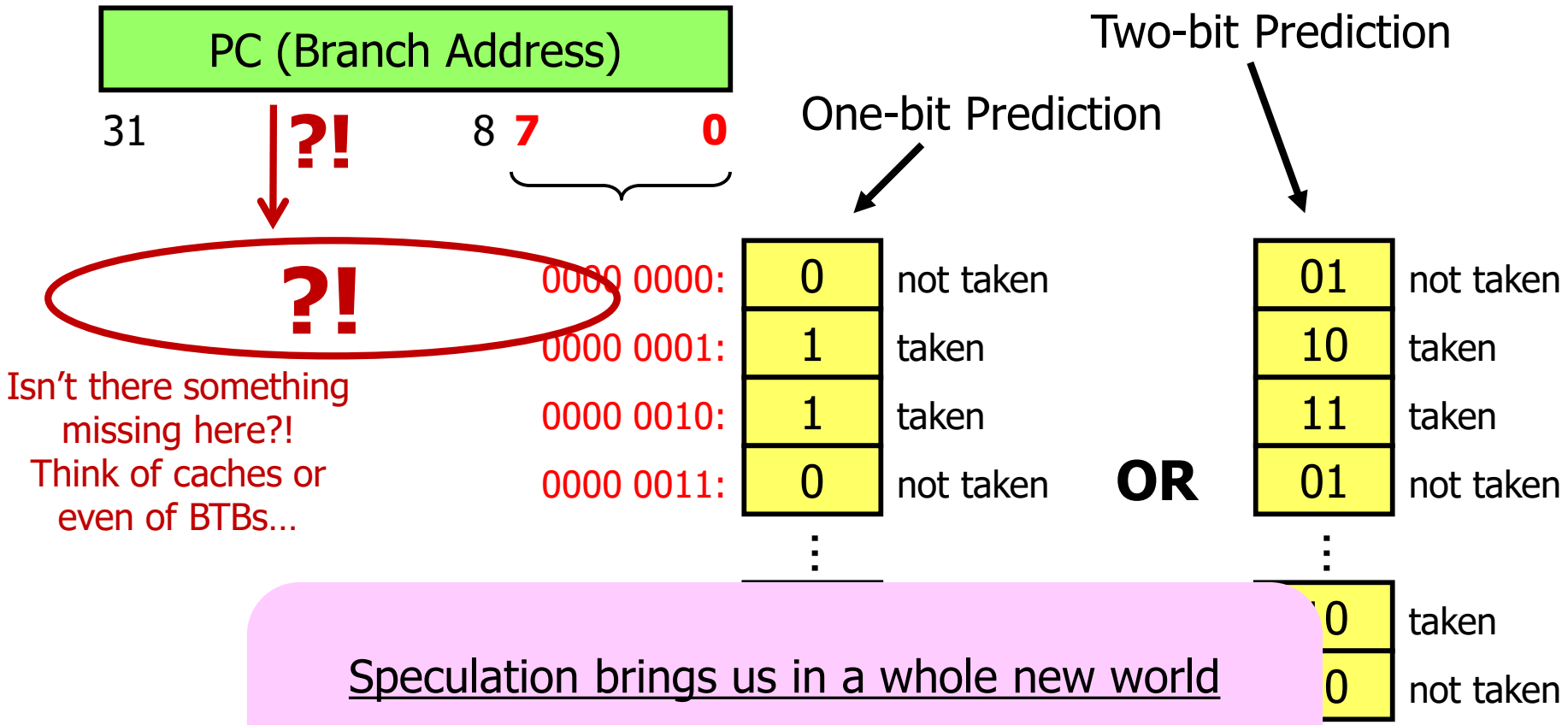
# Which Strategy to Predict Dynamically?

1. Same outcome as last time
   - Keep **one bit of history** per recently visited branches
     - Needs an associative memory ➔ expensive
   - Keep **one bit of history** per hashed address
     - Needs only a RAM ➔ inexpensive
     - Different branches alias ➔ mistakes, but we are only guessing, anyway…
2. Same outcome as last few times (inertia)
   - Keep a **two-bit saturating history counter** per hashed address; use sign as a predictor
     - Tuned to `for` loops: one misprediction is normal (last iteration) and should not modify the successive prediction (first iteration of a new execution)

# Branch History Table



# Branch History Table

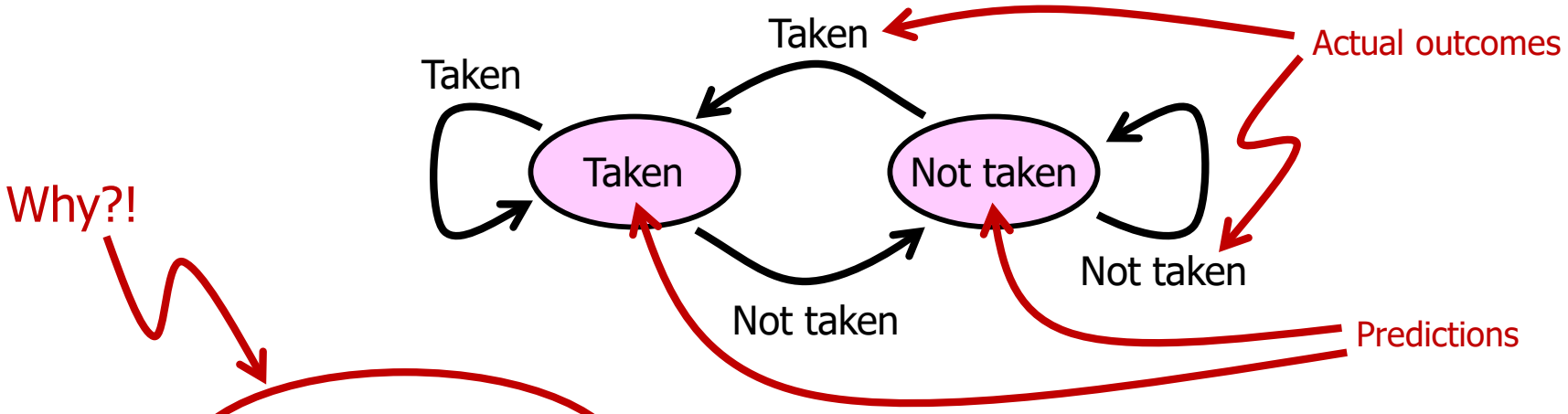



Speculation brings us in a whole new world

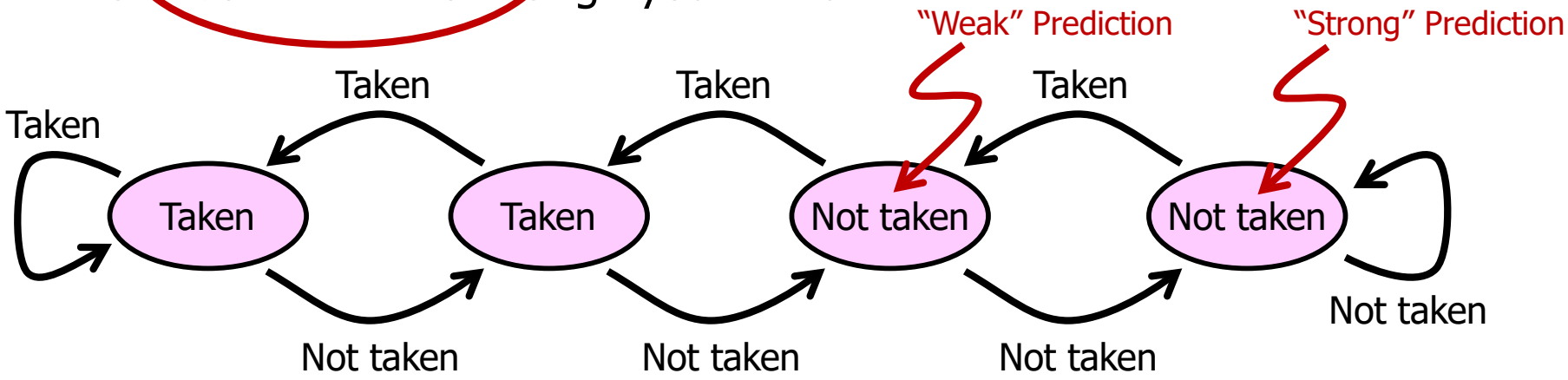
We do not care to be **always right**
but only to be **right most of the time**!

# One- vs. Two-Bit Prediction Schemes

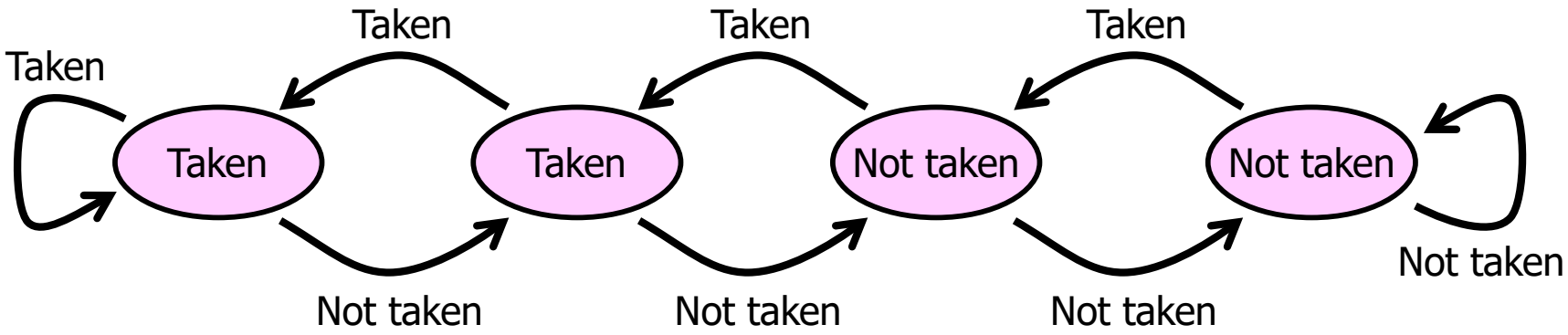
- Simplest one-bit predictor: "do the same as last time"




- Two-bit predictor (saturating counter): adding some "inertia" or "take some time to change your mind"



# One- vs. Two-bit Prediction Schemes and Loops

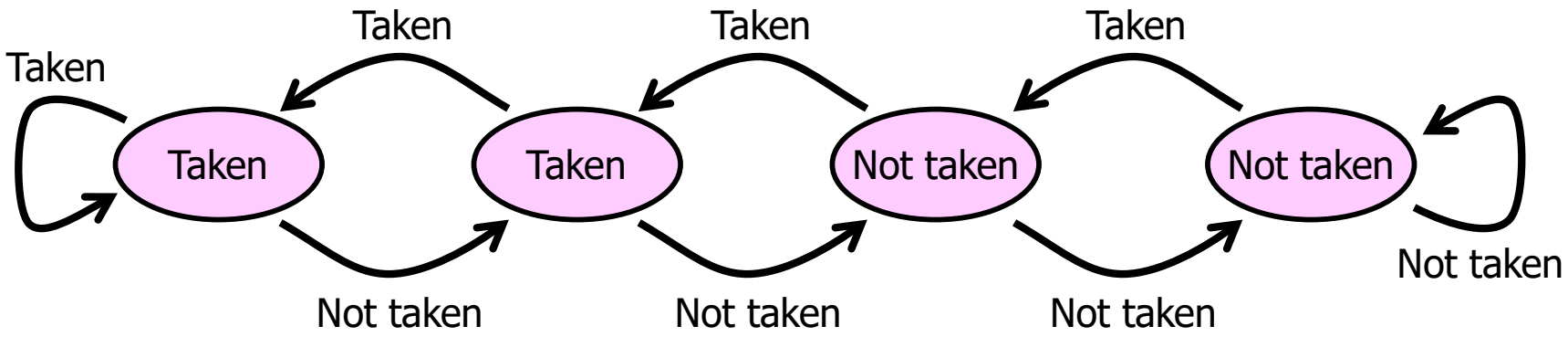



- How many mispredictions for `loop2` every iteration of `loop1`?

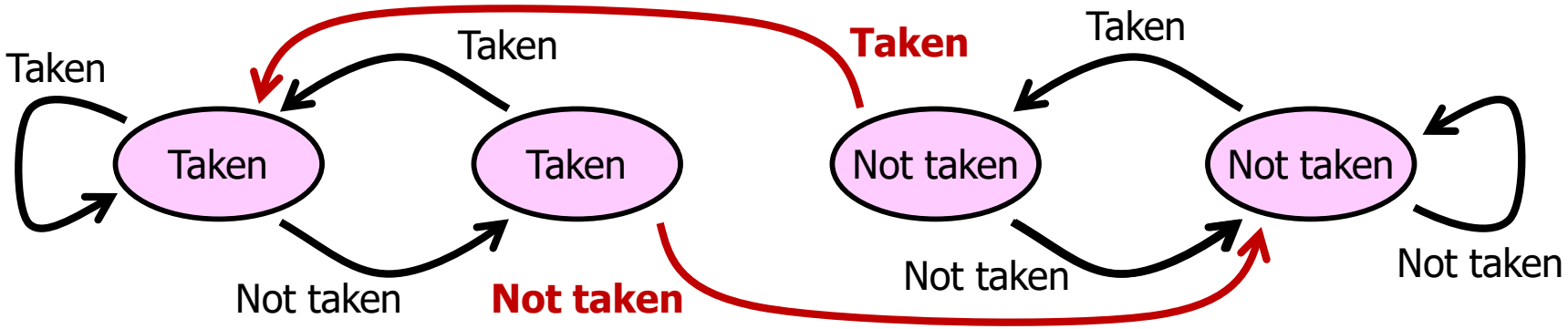
```
loop1: for (i = 0; i < ROW; i++) {
  …do something…
  loop2: for (j = 0; j < COL; j++) {
    …do something…
  }
}
```

# Exotic Prediction Schemes

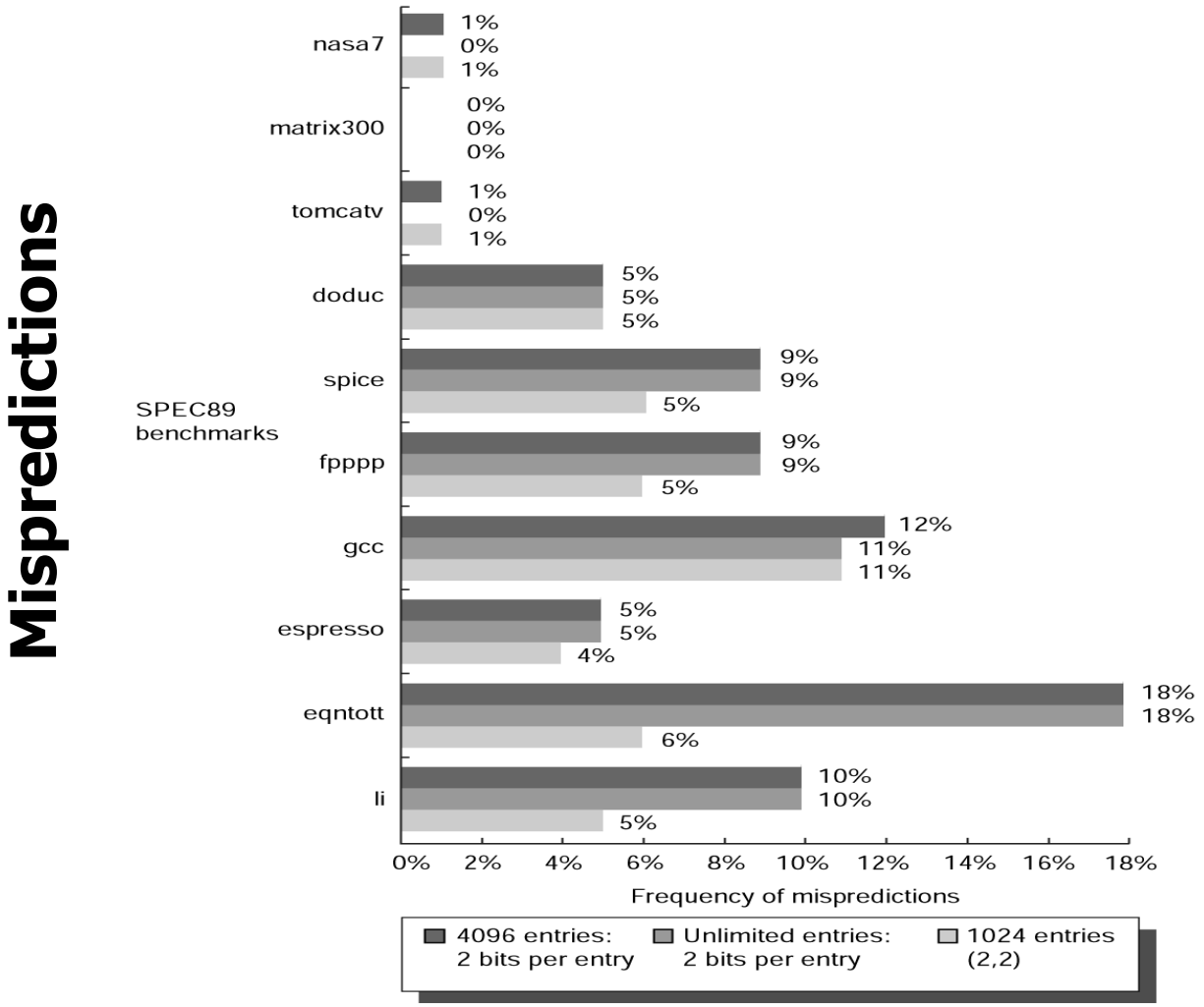
- Simple two-bit saturating counter




- Modified two bit saturating counter
  Two mispredictions → Strong reversal (e.g., UltraSPARC-I)

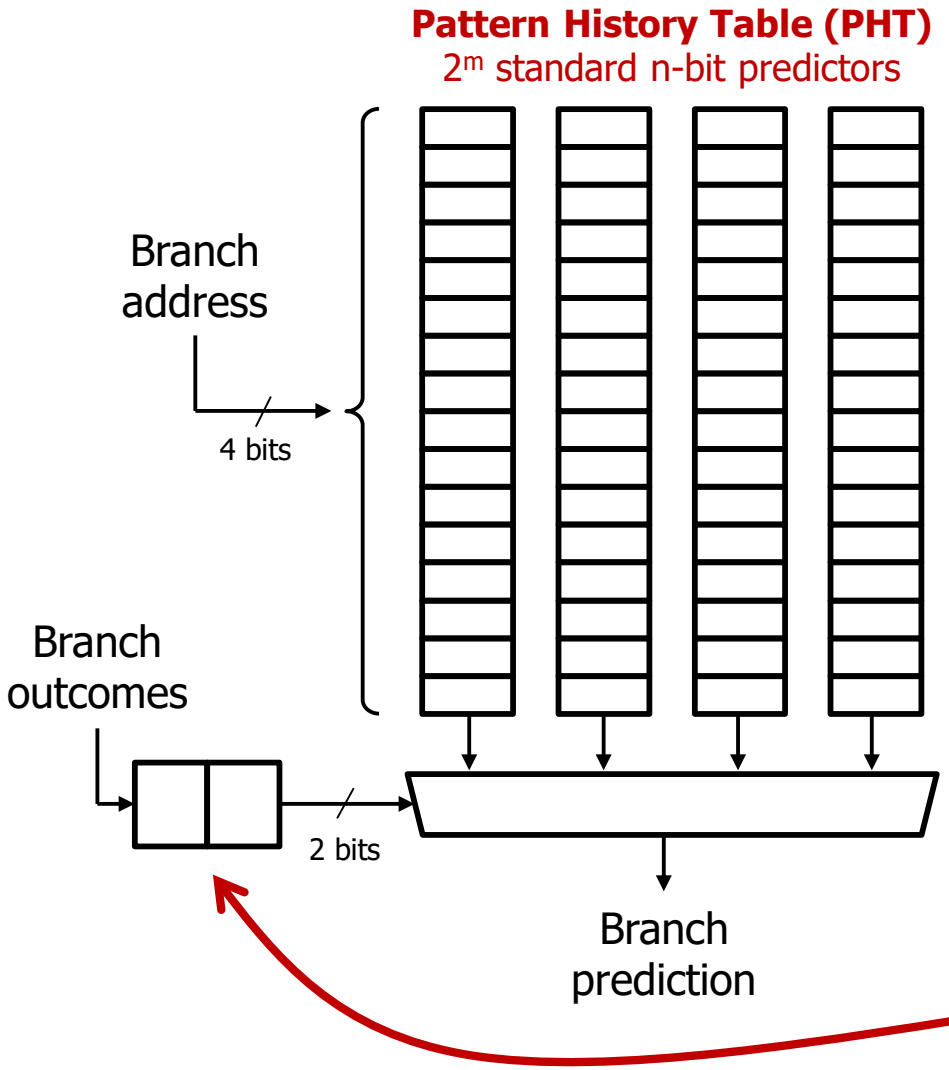


# Prediction Accuracy



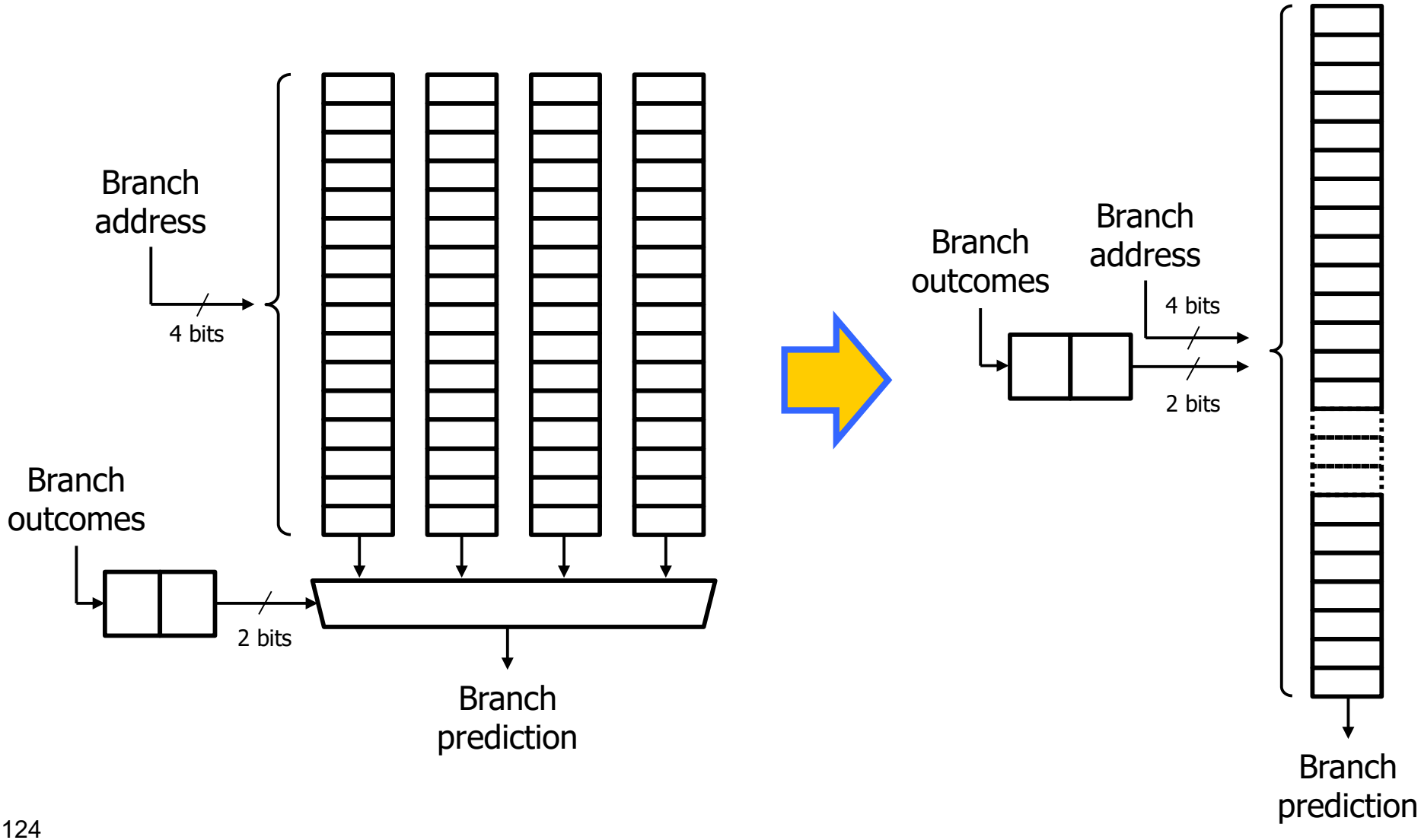

Source: AQA, © Morgan Kauffman 1996

# Exploiting Correlation (*two-level* or *gselect* Predictors)

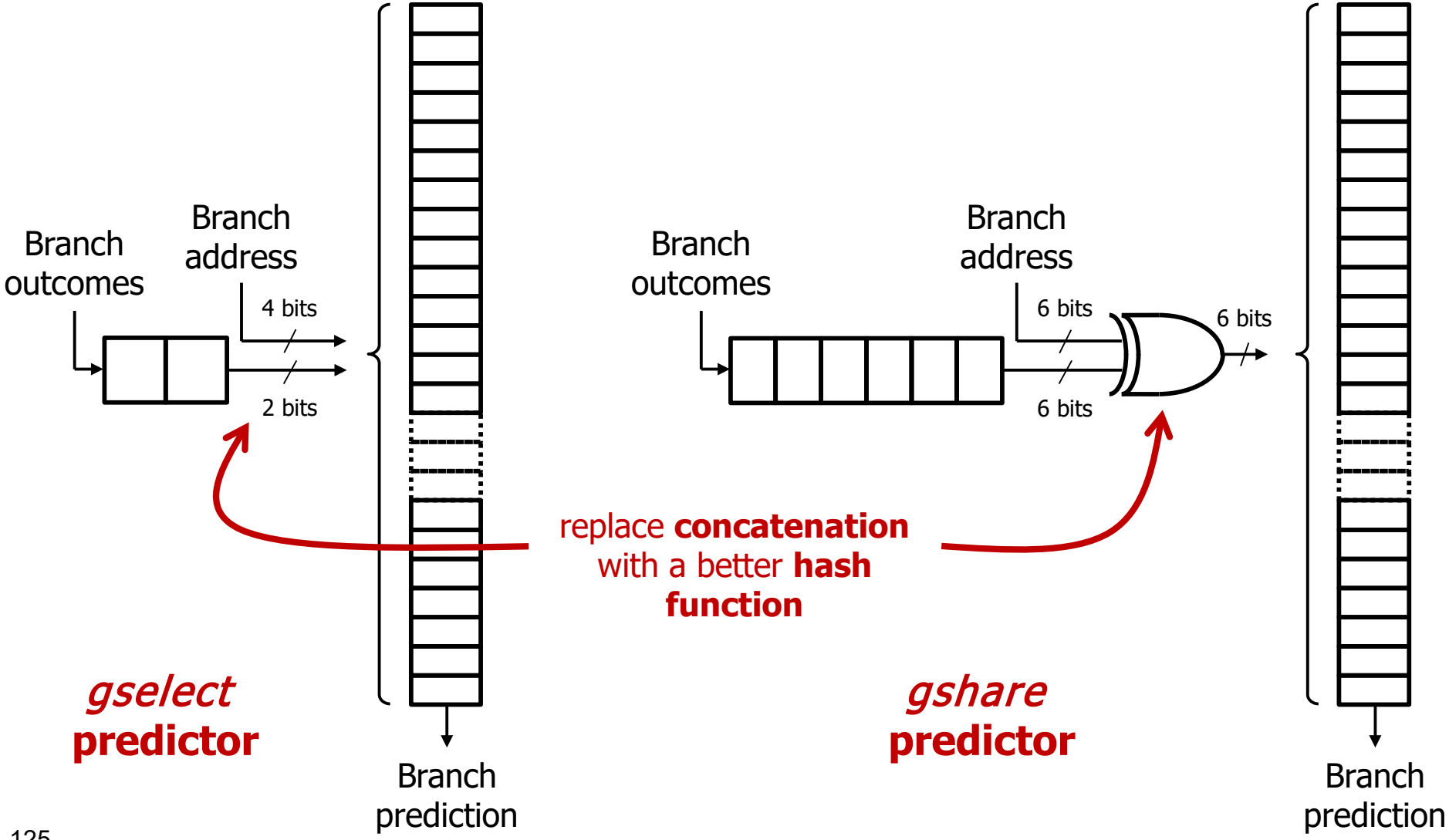



Exploit correlation:
**(m,n) Branch Predictor Buffer**
A global m-bit predictor uses the outcome of the last two branches to select one among four different predictors

# Exploiting Correlation (*two-level* or *gselect* Predictors)



# McFarling *gshare* Predictor



# The Sky (or the Architects' Ingenuity) Is the Limit…

- Tournament predictors
  - Combine several predictors (typically **local**, such as a simple 2-bit predictor, with **global**, such as a gshare predictor)
  - Use a **selector** to guess which would be the best predictor across the set
  - In case of misprediction it is not self-evident how to update the whole (update the predictors and/or update the selector?!)…
- Tagged hybrid predictors
- …

# Return Address Stack

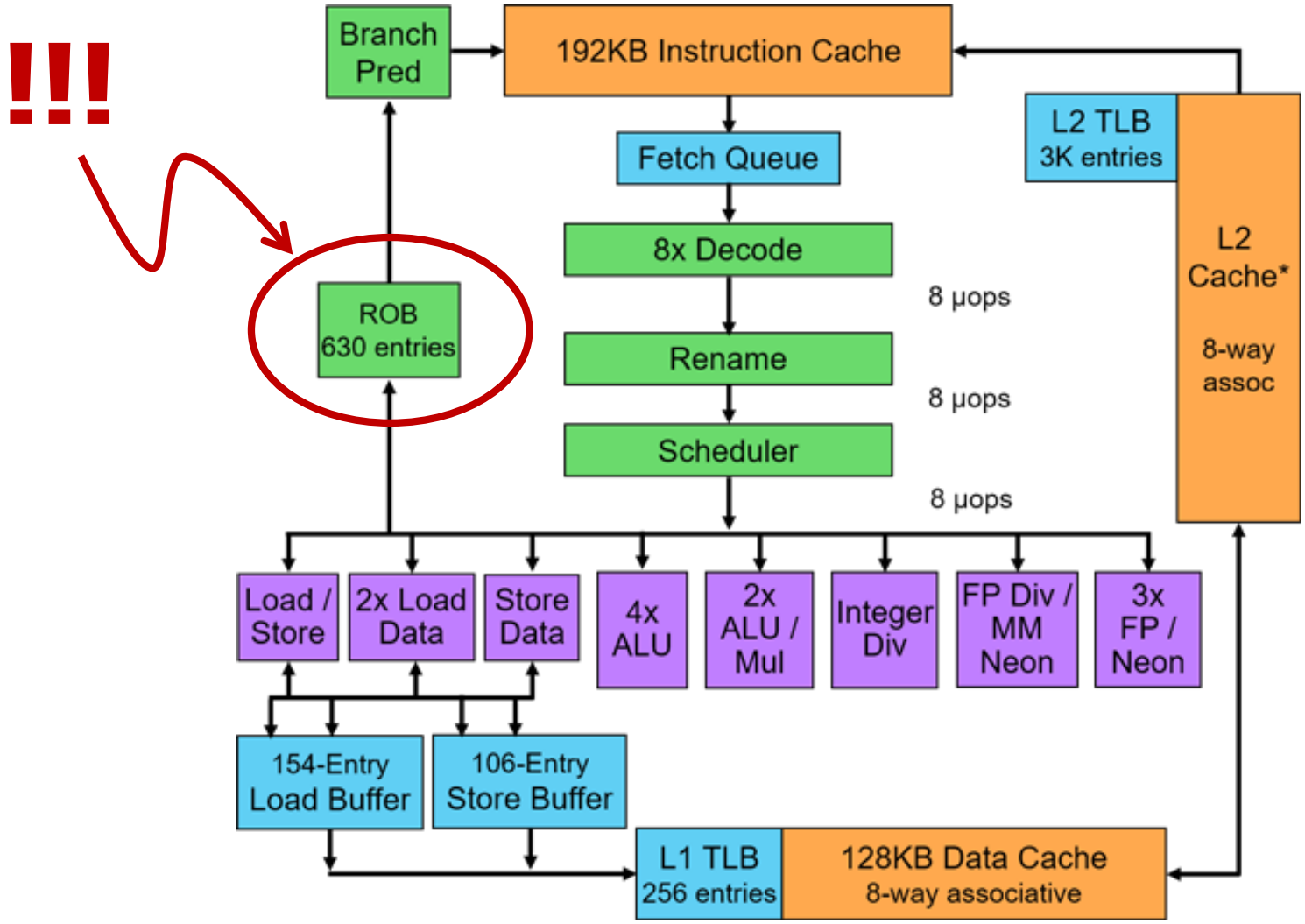
- Special elementary case of branch prediction:
  - Small stack (e.g., 8-16 values)
  - Each call (CALL, JAL, etc.) pushes a value
  - Each return (RET, JP $ra, etc.) pops a predicted return address
- Functionally identical to the "real" stack but avoids any SP manipulation, memory accesses, argument bypassing, etc.

# Misprediction Has High Cost → Lots of Efforts in Improving Accuracy

- Pipelines become more and more deep (e.g., up to 22-24 cycles in Pentium 4)
- Issue width grows (typically 3-8)
- Large number of in-flight instructions (hundreds)
- Many predicted branches in-flight at once
- Probability of executing speculatively something useful reduces quickly

$$p_{tot} = \prod_{i}^{all\_pred} p_i$$

# Apple Firestorm Microarchitecture

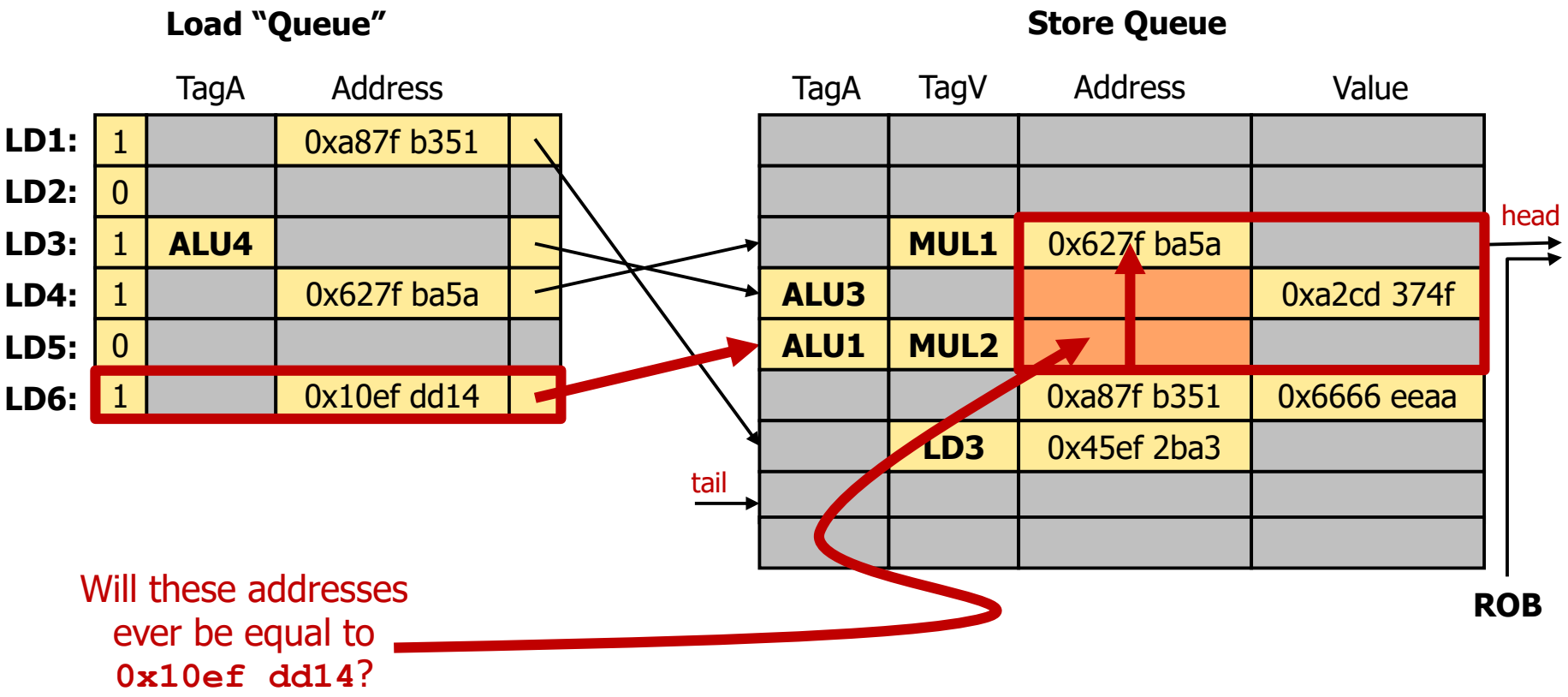


# Prediction & Speculation

- So far:
  - Precise exceptions
  - Branches

What's next?

# Reminder: Dependences through Memory




**LD6:** **Potential RAW** with stores whose address is unknown → **wait**

# Memory Dependence Prediction and Speculation

- **Prediction**
  - We can optimistically assume that there is **no dependence** (it is the only assumption that makes us gain time and the opposite assumption never leads to a functional mistake…)
- **Speculation**
  - If there was a dependence, every data dependent instruction should be squashed; independent instructions were actually correctly executed
  - If we accept to squash all following instructions, this situation is not qualitatively different from what we have seen for other cases ➔ ROB

# Alias Prediction

- One could certainly do better than simply assuming that any potential RAW through memory is not a RAW (= simple **static dependence prediction**)
- The goal is to reduce the probability of squashing and replaying (if squashing costed nothing, the static prediction would be ok, but squashing almost invariably has a cost—and definitely in terms of energy)
- Essentially one could build **dynamical predictors** similar in spirit to branch predictors (the intuition is that dependences are program specific but often **stable** during program execution) → learn from history and remember what happens on previous visits of a load
- In fact, one could even predict a specific dependence (**alias prediction**—that is, on which store a given load depends) and use it to bypass memory before addresses are known

# Prediction & Speculation

- So far:
  - Precise exceptions
  - Branches
  - Dependences in memory

What's next?

# Predicting the Next Miss? Prefetching I and D into the Cache

- Caches partially reduce memory latency, OOO execution partially hides memory latency
- What to do if there is a significant # of misses?
  - Misses in all cache levels need hundred(s) of cycles
  - OOO may not be enough to hide this

Nonbinding prefetch

**Idea**

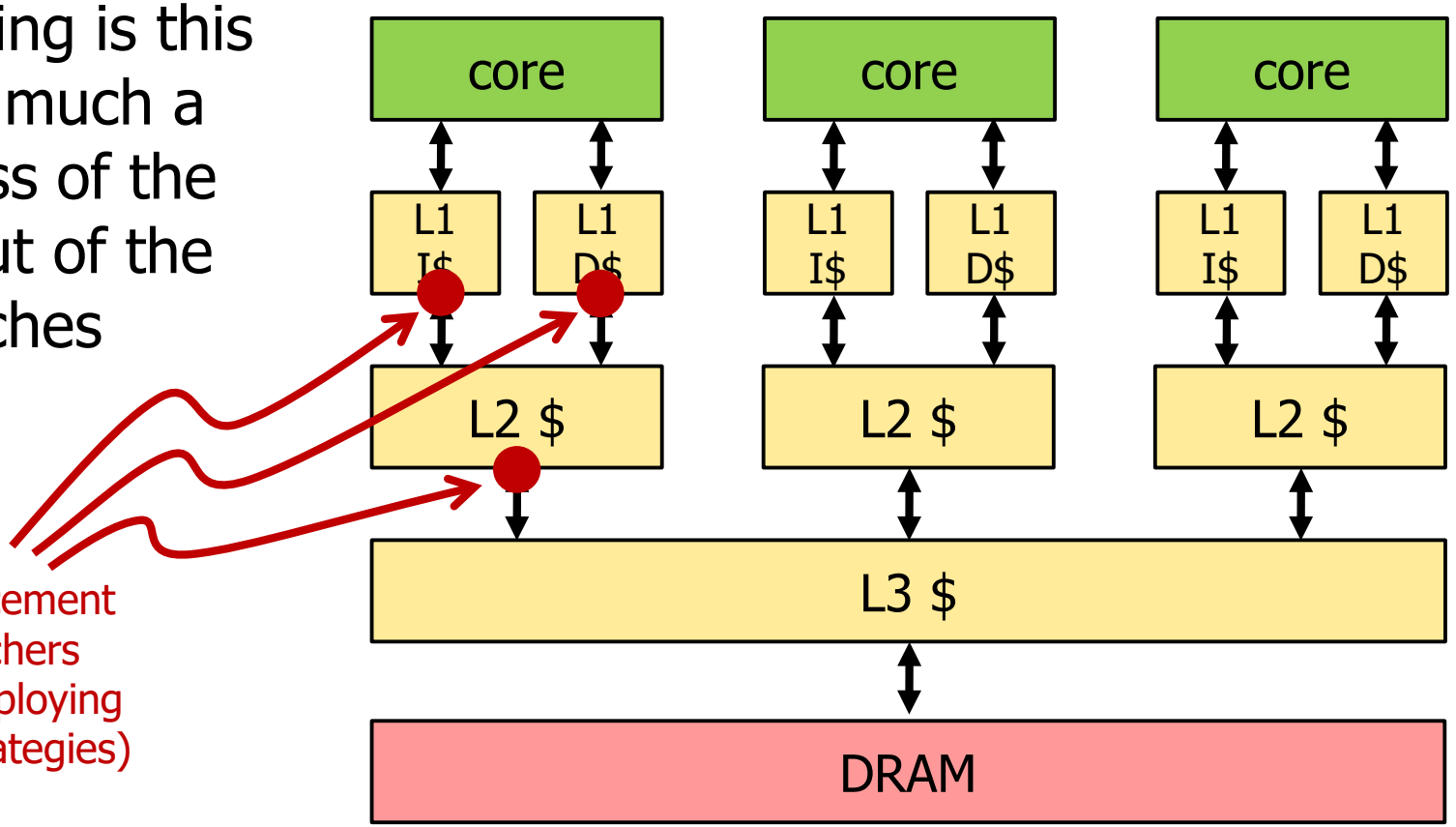
Fetch data into the cache
ahead of processor demanding it

# Prefetching
# Prediction and Speculation

- **Prediction**
  - **What** and **when** to get from memory
  - As usual, exploit **typical behavior** (e.g., programs are sequences of instructions) and learn from **execution history** (e.g., discover access stride)
- **Speculation**
  - Since we are putting data in the cache (which is not architecturally visible), **nothing to do to rollback**
  - Still, prefetching has a **cost** (besides energy, it consumes memory bandwidth) and could be damaging (leads to eviction of useful stuff)

# Prefetchers and Memory Hierarchy

No influence on the processor state → prefetching is this not so much a business of the core but of the caches




Typical placement of prefetchers (usually employing different strategies)

# Prefetching
# Prediction and Speculation

- **Prediction**
  - **What** and **when** to get from memory
  - As usual, exploit **typical behavior** (e.g., programs are sequences of instructions) and learn from **execution history** (e.g., discover access stride)
- **Speculation**
  - Since we are putting data in the cache (which is not architecturally visible), **nothing to do to rollback**
  - Still, prefetching has a **cost** (besides energy, it consumes memory bandwidth) and could be damaging (leads to eviction of useful stuff)

# Prefetching Coverage and Accuracy

- **Coverage**: How many misses prefetching removes?
- **Accuracy**: How many prefetched cache lines are useful over all prefetched lines?

Sort of a trade-off:

- prefetching very **aggressively** improves coverage but reduces accuracy → **pollutes** the cache
- prefetching **conservatively** may improve accuracy but reduces coverage → little benefit

# Next-Line Prefetching

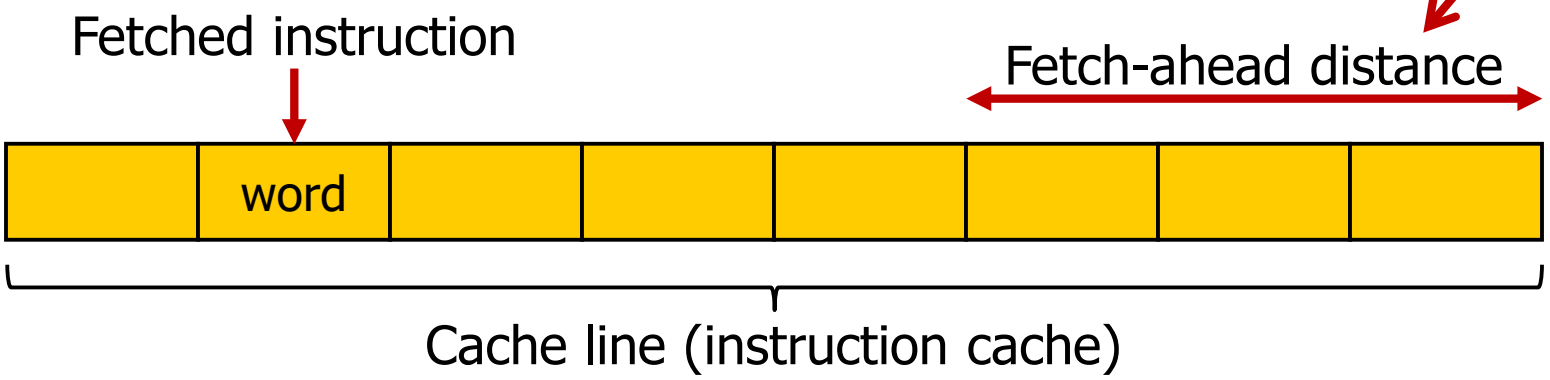
Similar to huge cache lines?

- Simplest intuition:
  - If cache line X is a miss, load X but also X+1

What?

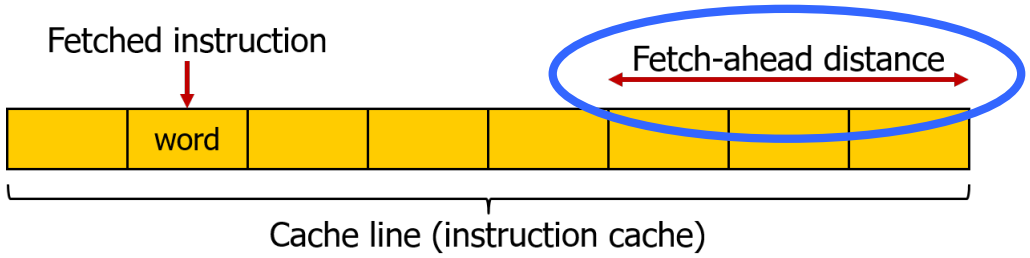
- Easiest scheme, no "intelligence"
- How to implement it? Lookahead?
  - Do not load X+1 immediately but wait until the processor asks for an instruction some "fetch-ahead distance" from the end of the line

When?



# Next-N-Line (or Stream) Prefetching

- Is the lookahead inside a cache line enough to hide the latency of a miss?

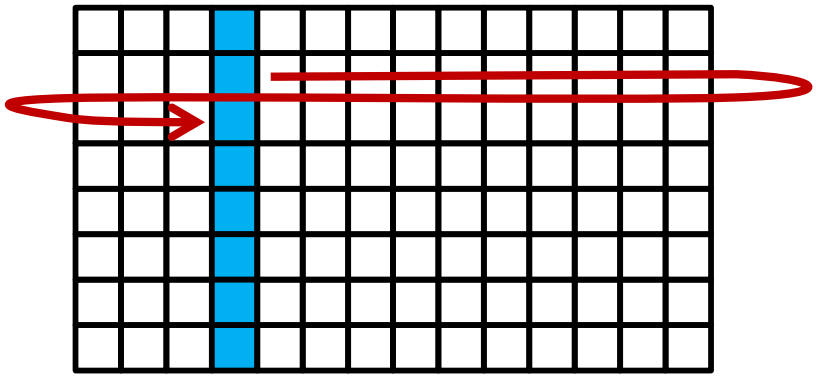



- The natural extension is, on a request for cache line X, to prefetch not only X+1 but also X+2, X+3,… X+N
- N is a critical parameter:
  - Too small → poor coverage
  - Too large → poor accuracy

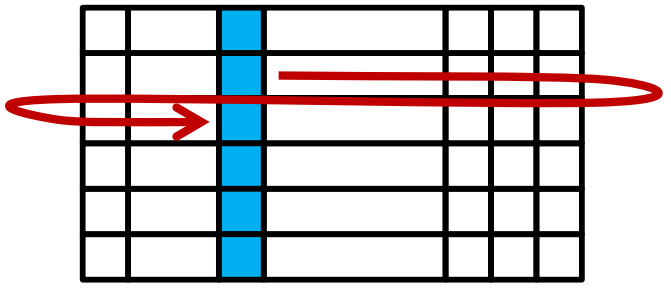
# Stride Prefetching

- For instructions or elements of matrix accessed row-wise, the sequence X, X+1, X+2, etc. is appropriate
- But what about other typical cases?

accessing a matrix column-wise



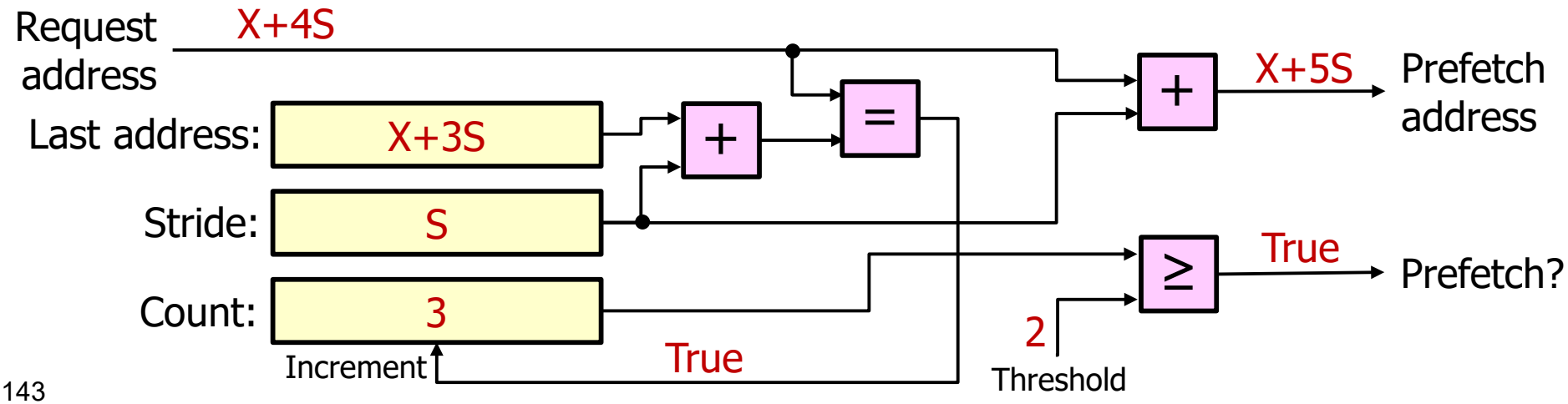
accessing a field in an array of `struct`



- Natural extension: distance (**stride**) should be >1
  - On a request for X, prefetch X+S, X+2S,... X+N·S

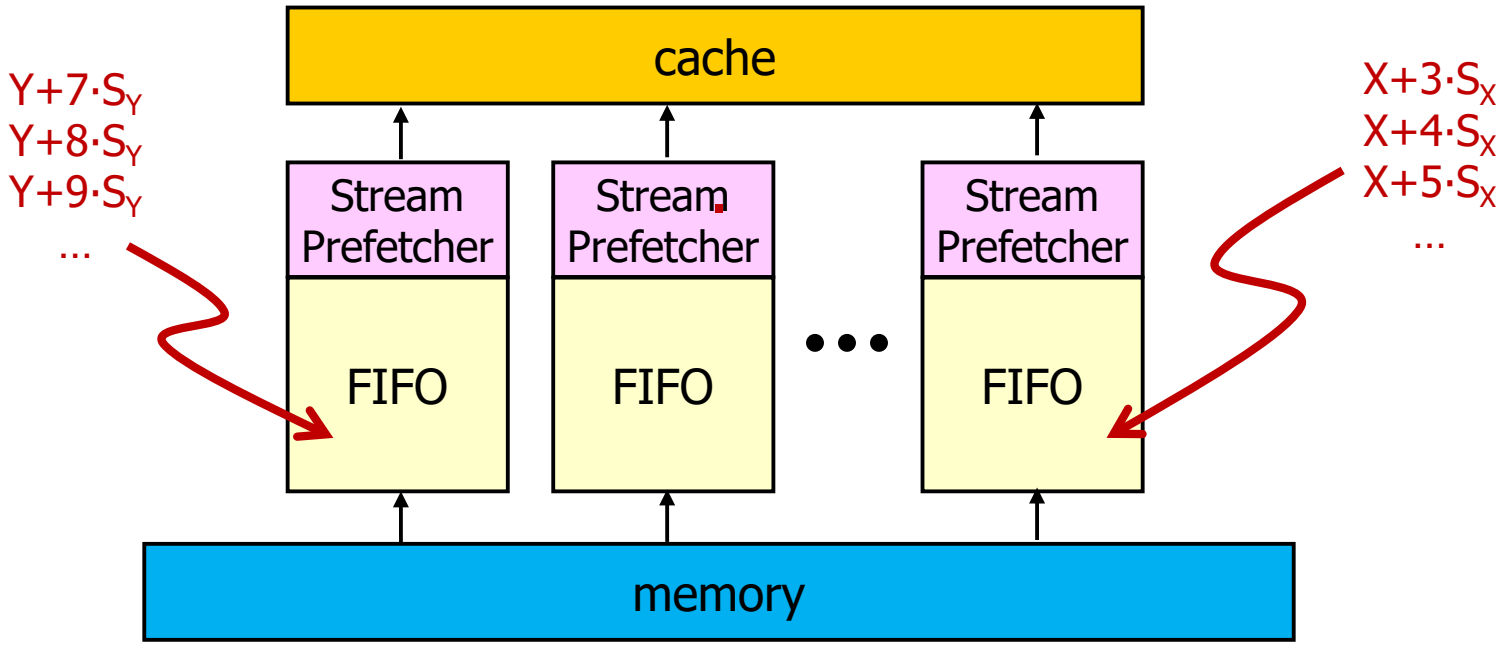
# Stride Prefetching

- Usually takes a few misses to detect and build confidence in a constant stride:
  - X → Compulsory miss
  - X+S → Compulsory miss, S hypothesis
  - X+2S → Compulsory miss, S confirmed
  - X+3S → Compulsory miss, S confirmed again, prefetch
  - X+4S → Hit, S confirmed again, prefetch



# Stream Buffers

- There may be various streams mixing (e.g., multiple arrays, etc.)
- Aggressive prefetching of multiple streams leads to cache pollution




- Implement **multiple** Next-N-Line or Stream prefetchers
- Place the prefetched lines in **FIFO buffers** instead of the cache

# Stream Buffers

- On a miss, check the head of all stream buffers:
  - **If match** (i.e., found the desired cache line), pop the desired entry from the buffer head, prefetch the $N^{th}$ cache line of the series, and place it at the buffer tail
  - **If no match**, evict one of the stream prefetchers (e.g., least successful or least recently used) and try to build a new stream

# Prediction & Speculation

- So far:
  - Precise exceptions
  - Branches
  - Dependences in memory
  - Prefetching

What's next?

# Dynamic Data Value Prediction

- Examples:
  - *Source Operand Value Prediction*: predict quasi-constant input operands
    - Many constant values during program execution
    - History table recording last value
  - *Value Stride Prediction*: predict constant increments across input operands
    - History table recording stride between last two values
  - *Load Addresses* and *Load Values*

# Speculation Is Not Necessarily a Run-Time Concept

- **Dynamic**: in hardware, no interaction whatsoever from the compiler
  - Binary code is unmodified
- **Static**: in software, planned beforehand by the compiler
  - Binary code is written in such a way as to do speculation (with or without some hardware support in the ISA)

# Static Control Speculation Example

- We need to compute:

```
if (A==0) A=B; else A=A+4;
```

- In assembly:

```
    LW    R1, 0(R3)     ; load A
    BNEZ  R1, L1        ; test A, possibly skip then
    LW    R1, 0(R2)     ; 'then' clause: load B
    J     L2            ; skip else
L1: ADD   R1, R1, 4     ; 'else' clause: compute A+4
L2: SW    0(R3), R1     ; store new A
```

- If we know that the 'then' clause is almost always executed, can we optimise this code?

# Static Control Speculation Example

- We could speculatively start earlier to load B into another register and, if needed, squash the value with the right one
- In assembly:

```
      LW     R1, 0(R3)      ; load A
      LW     R14, 0(R2)     ; speculative load B
      BEQZ   R1, L3         ; test A, possibly skip else
      ADD    R14, R1, 4     ; 'else' clause: compute A+4
L3:   SW     0(R3), R14     ; store new A
```

- Advantages: now we load B while the test is performed (→ in parallel)
- Any problem? As usual: exceptions…

# Exceptions and Static Speculation

- Some ways to handle exceptions in (software) speculative execution:
  - *Static renaming:* Hardware and operating systems cooperatively ignore exceptions
  - *Poison bits:* Mark results as speculative and delay exception at first use
  - *Speculative instructions:* Mark instruction as speculative and do not commit the result until speculation is solved

# Static Renaming and Hardware-Software Cooperation

- Back to our example:

```
      LW     R1, 0(R3)     ; load A
      LW     R14, 0(R2)    ; speculative load B
      BEQZ   R1, L3        ; test A, possibly skip else
      ADD    R14, R1, 4    ; 'else' clause: compute A+4
L3:   SW     0(R3), R14    ; store new A
```

- OS "helps" with two policies:
  - Nonterminating exceptions (e.g., Page Fault): resume independently from speculativeness → performance penalty, but execution ok
  - Terminating exceptions (e.g., Divide by Zero): ignore and return an undefined value → if it was speculated, it will be unused
- Problem: nonspeculative terminating exceptions?

# Static Renaming and Poison Bits

- Special marker for speculative instructions:

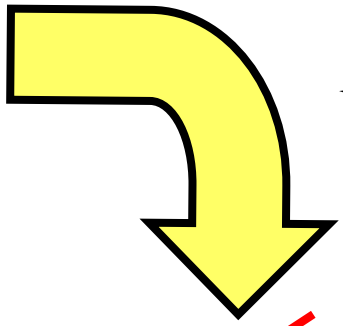
```
      LW     R1, 0(R3)      ; load A
      LW*    R14, 0(R2)     ; speculative load B
      BEQZ   R1, L3         ; test A, possibly skip else
      ADD    R14, R1, 4     ; 'else' clause: compute A+4
L3:   SW     0(R3), R14     ; store new A; report exceptions
```

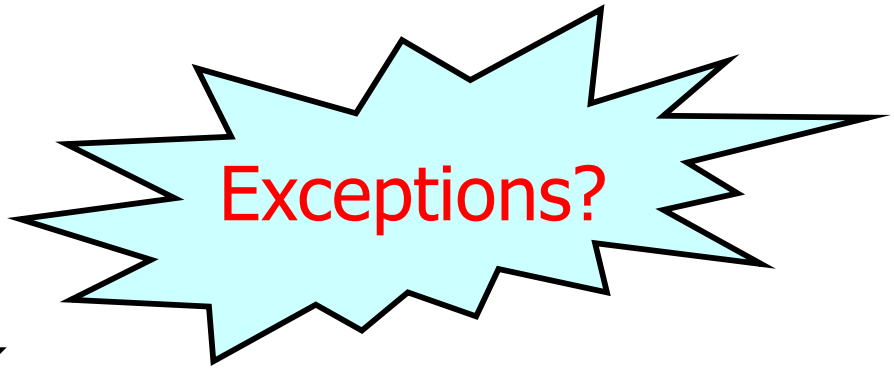
- The processor knows the load is speculative and turns on R14's Poison Bit if it raises a terminating exception, and suppresses the exception
- The add, if executed, resets the R14's Poison Bit
- When R14 is used, a deferred terminal exception is raised if its Poison Bit is set

# Example: Speculative Loads in Itanium (I)

- **Goal**: move loads as early as possible, even **speculatively** before preceding branches (i.e., without being sure they are really needed)

```
<some code>
(p1) br.cond somewhere
// ------ barrier
ld r1 = [r2]
<some code using r1>
```






```
ld r1 = [r2]                    // load could be speculated
                                // if old value r1 not needed
<some code>                     // <- neither here nor
(p1) br.cond somewhere          //    in "somewhere"
// ------ barrier
<some code using r1>            // but…
```

# Example: Speculative Loads in Itanium (II)

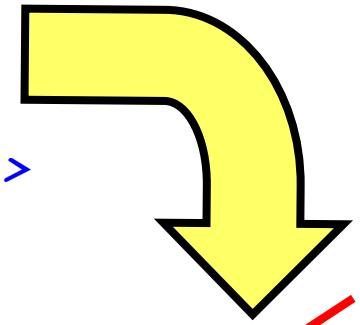
- Speculative loads and deferred exceptions to explicit compiler-generated fix-up code

```
ld.s r1 = [r2]              // speculative loads do not raise
                            // exceptions but mark the register
                            // with the additional NaT bit
<some code>
<some code using r1>        // NaT is propagated in further
                            // calculations, which also
                            // defer exceptions
(p1) br.cond somewhere
// ------ barrier
<some more code using r1>
chk.s r1, fix_code_r1       // call exception handler if needed
                            // to fix-up execution
```

# Static Data Dependence Speculation (I)

- Potential RAW dependencies through memory are to be conservatively assumed as real dependencies → Loss of useful reordering possibilities
- **Goal**: move loads as early as possible, even **speculatively** before preceding stores (i.e., without being sure that the value is right)

```
<some code>
st [r3] = r4
// ------ barrier
ld r1 = [r2]
<some code using r1>
```




```
ld r1 = [r2]                    // load could be speculated…
<some code>
st [r3] = r4                    // …but if r2==r3, r1 is WRONG!
// ------ barrier
<some code using r1>
```

# Static Data Dependence Speculation (II)

- Speculative Loads get executed but mark the destination register as "speculatively" loaded and track subsequent stores for a conflict

```
ld.a r1 = [r2]              // speculative loads are normal
                            // but mark always the register
                            // with the additional NaT bit
<some code>
<some code using r1>        // NaT is propagated in further
                            // calculations

st [r3] = r4                // successive stores are checked
                            // to see if they rewrite locations
                            // which were object of speculative
                            // loads
// ------ barrier
<some more code using r1>
chk.a r1, fix_code_r1       // if violated RAW dependence, call
                            // special fix-up routine
```

- Important advantage because loads (slow operations) can now be started earlier

# Predicated (= Guarded) Execution

- A special form of static control speculation?

  *"I cannot make a good prediction? I will avoid gambling and will do both"*

- A bit more than that: removes control flow change altogether (see lectures on statically scheduled processors)

- Not always a good idea: compiler trade-off
  - (Almost) free if one uses execution units which where not used otherwise (e.g., because of limited ILP)
  - Not free at all in the general case: more than needed is always executed

# Types of Prediction & Speculation

| | **Dynamic** (by the hardware) | **Static** (by the compiler) |
|---|---|---|
| **Exceptions** | ❑OOO execution and reordering<br>❑Imprecise exceptions in DBT (e.g., Transmeta Crusoe) | — |
| **Control** | ❑Branch Prediction | ❑Trace Scheduling<br>❑Hyperblocks<br>❑Predication<br>❑Speculative Loads (e.g., Itanium) |
| **Data Availability** | ❑Virtual memory | — |
| **Data Dependence** | ❑Load/Store Queues | ❑Advanced Loads (e.g., Itanium) |
| **Data Value** | — | ❑Dynamic compilers (e.g., DyC, Calpa) |

Not all of these are traditionally called "speculation"!

# References on Prediction & Speculation

- AQA 5th ed., Chapter 3 and Appendix H
- PA, Sections 4.3 and 5.3
- J. E. Smith, *A Study of Branch Prediction Strategies*, Eight International Symposium on Computer Architecture (ISCA), 135-48, May 1981

# 4

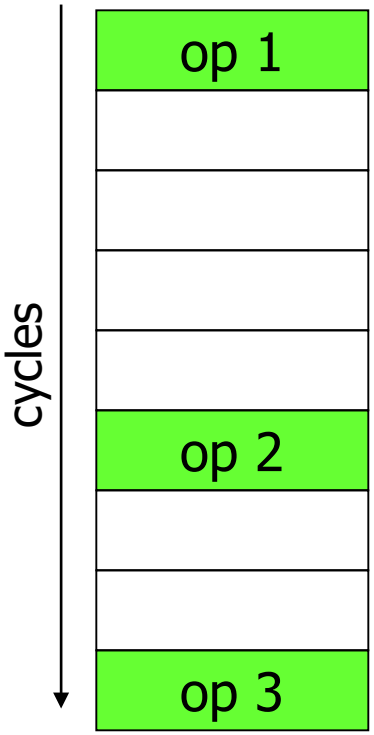
## Simultaneous Multithreading

## (How do I fill my issue slots?!...)

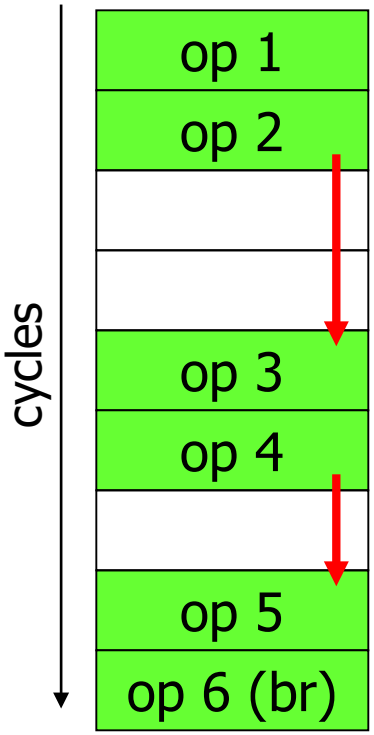
# Sources of Parallelism

- Bit-level
  - Wider processor datapaths (8→16→32→64…)
- Word-level (SIMD)
  - Vector processors
  - Multimedia instruction sets (Intel's MMX and SSE, Sun's VIS, etc.)
- Instruction-level
  - Pipelining
  - Superscalar
  - VLIW and EPIC
- **Task- and Application-levels…**
  - Explicit parallel programming
  - **Multiple threads**
  - Multiple applications…

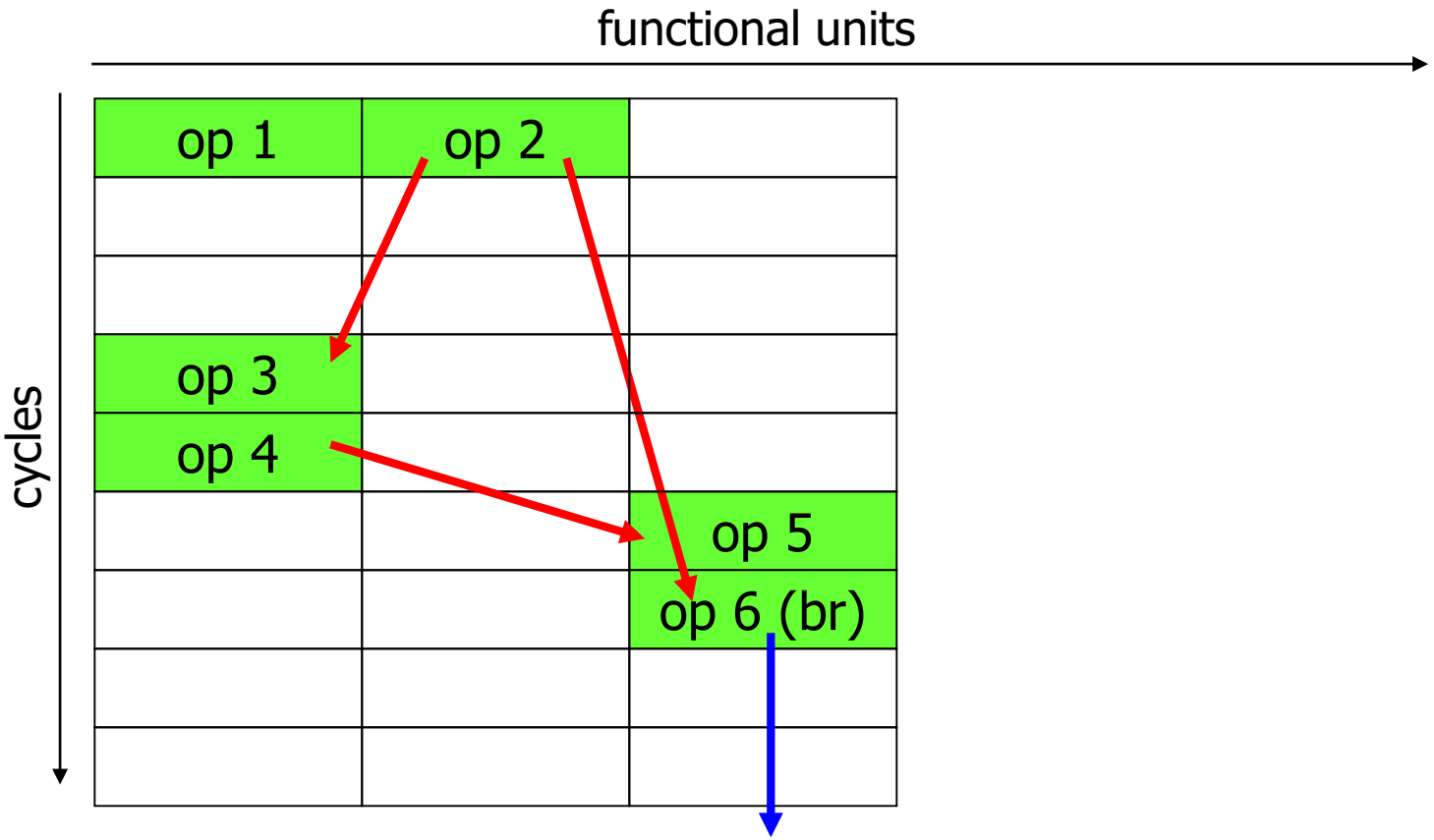
# Simple Sequential Processor



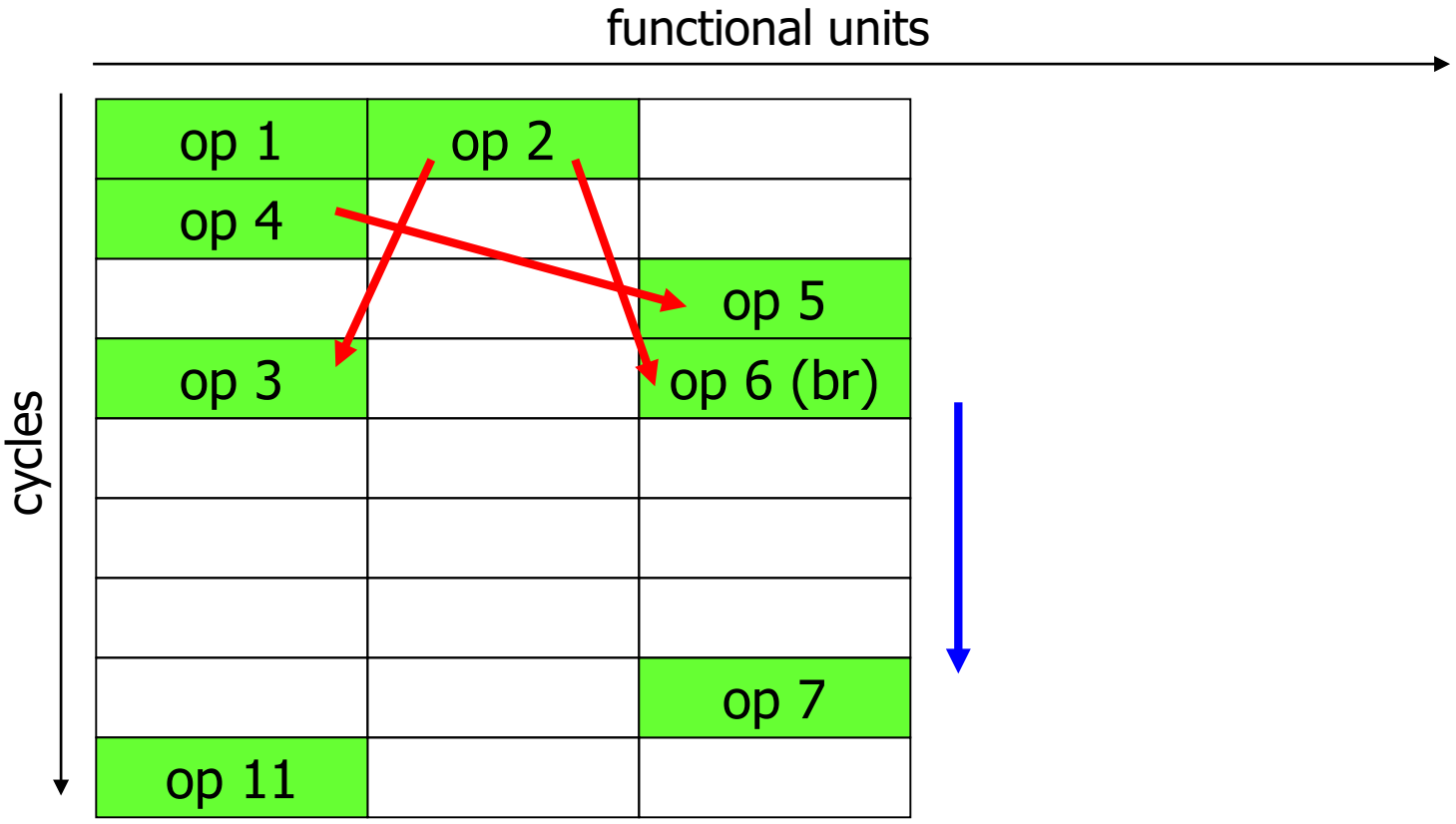
# Pipelined Processor



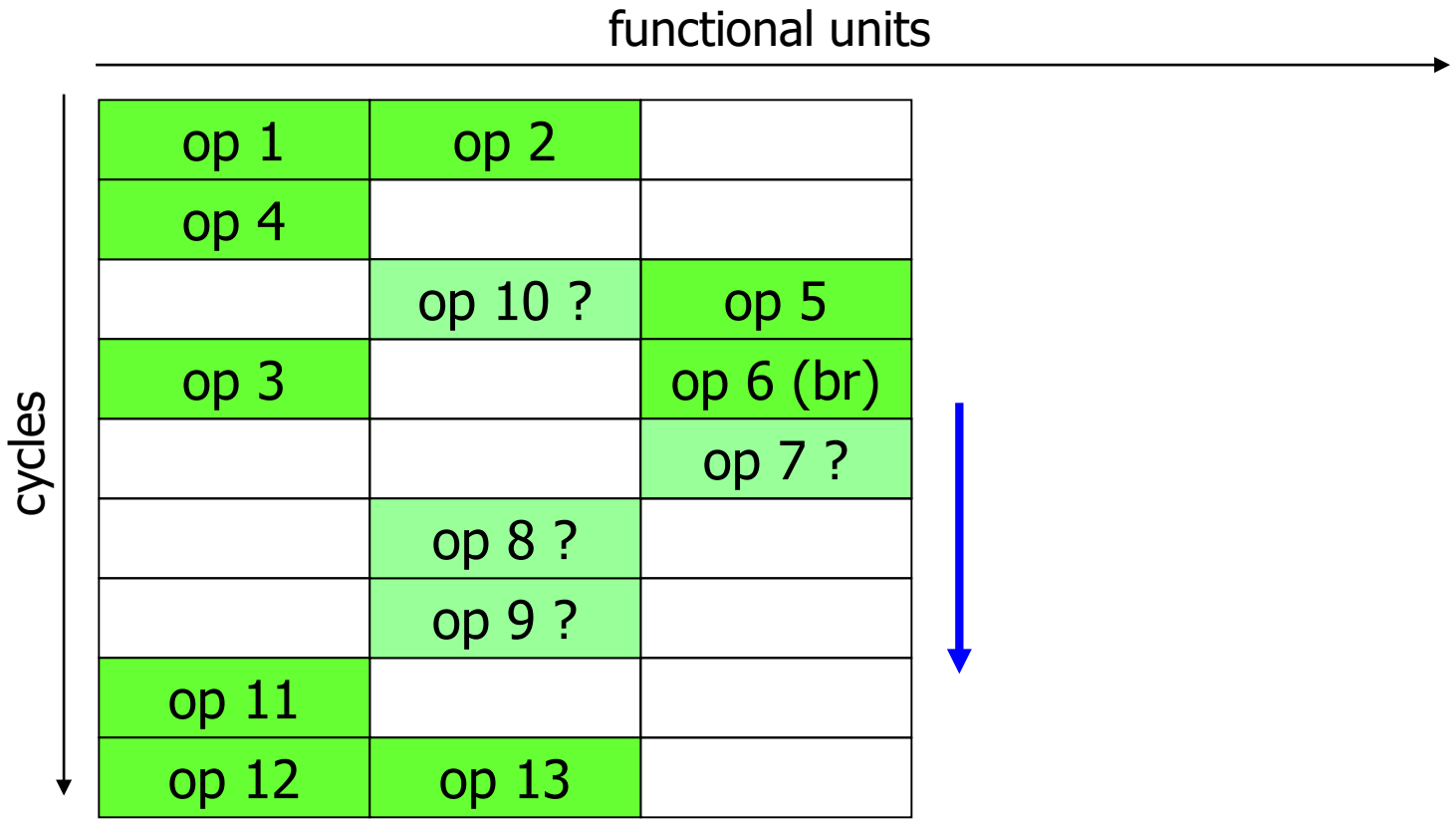
# Superscalar Processor



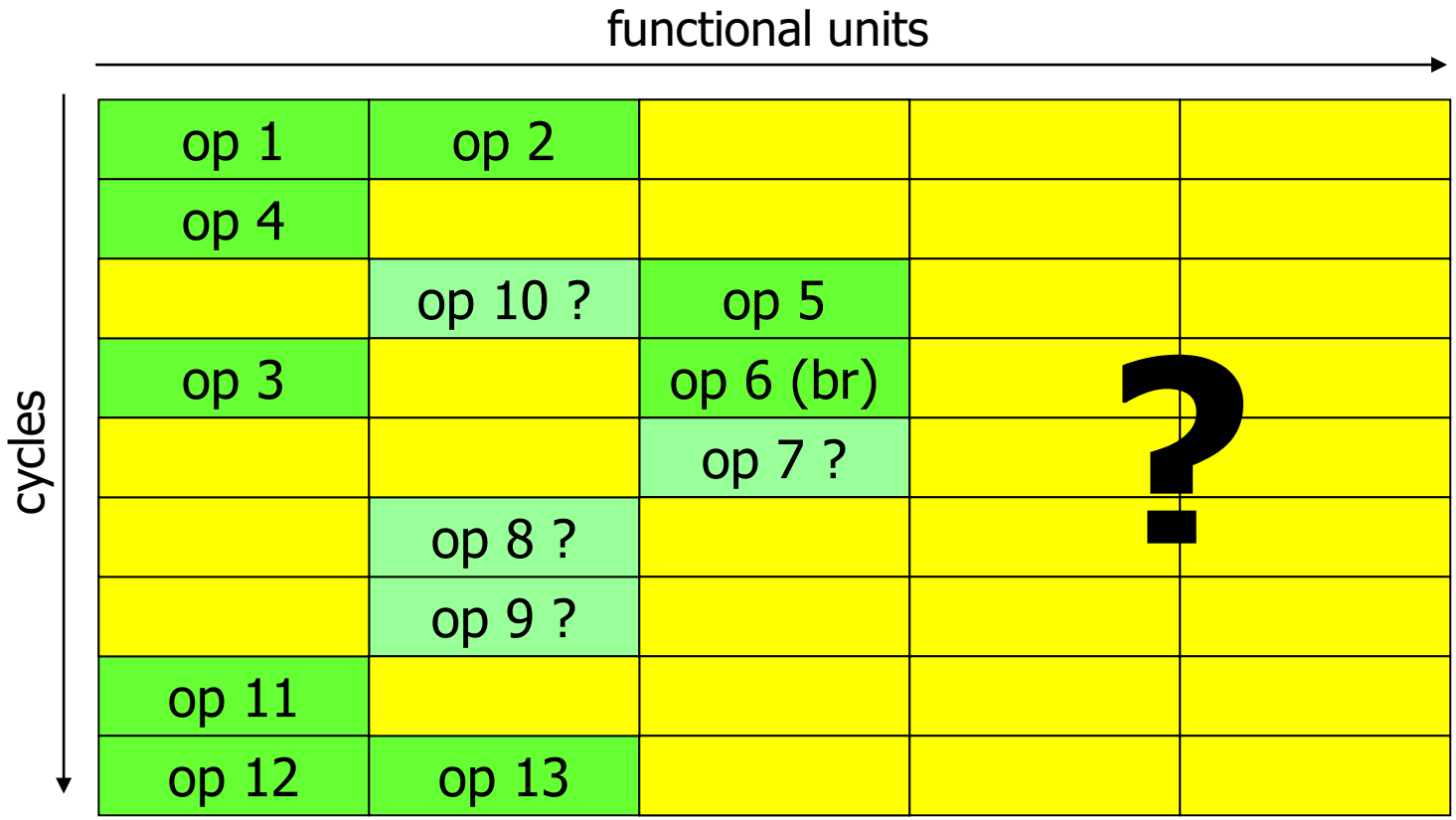
# OOO Superscalar Processor



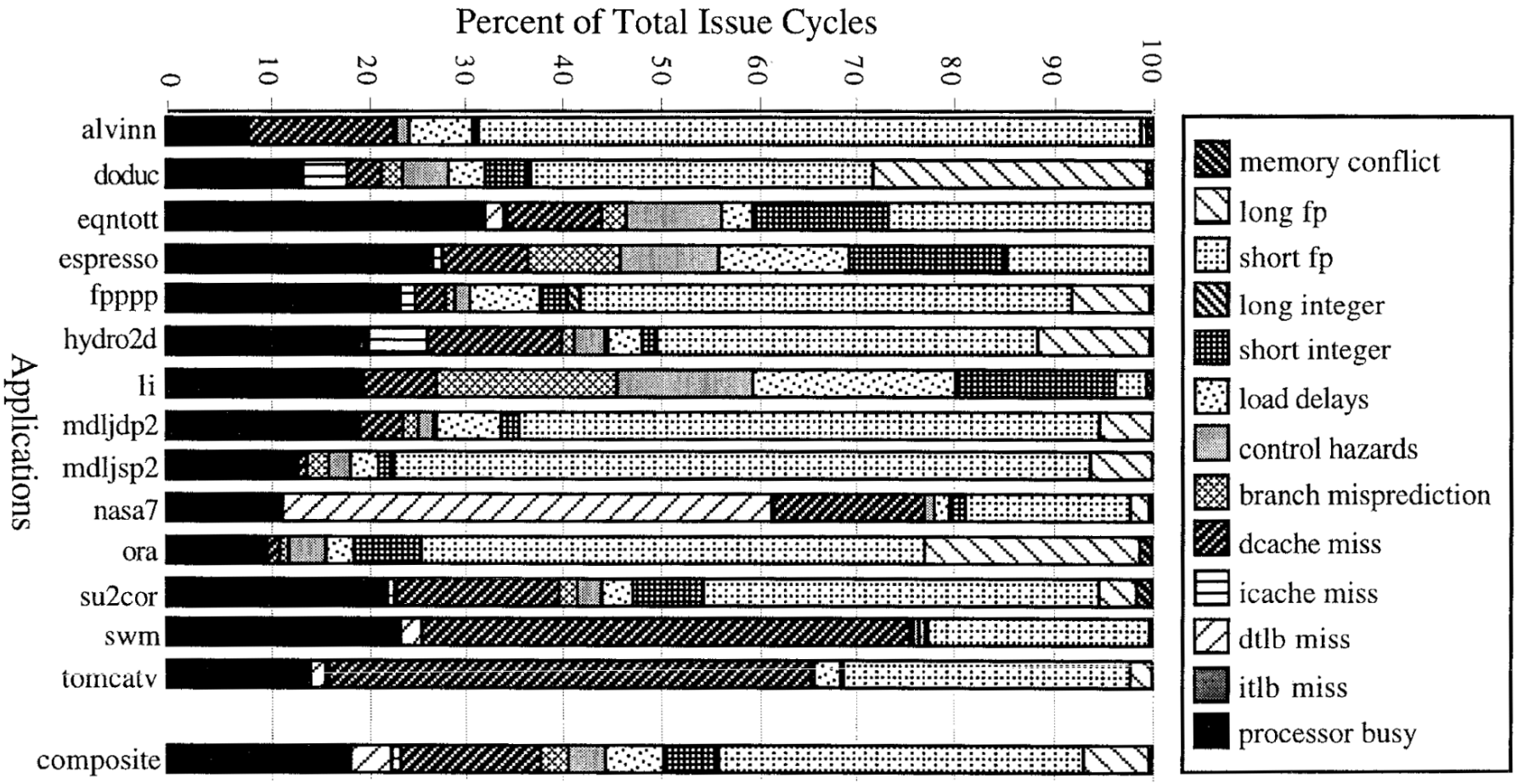
# Speculative Execution



# Limits of ILP

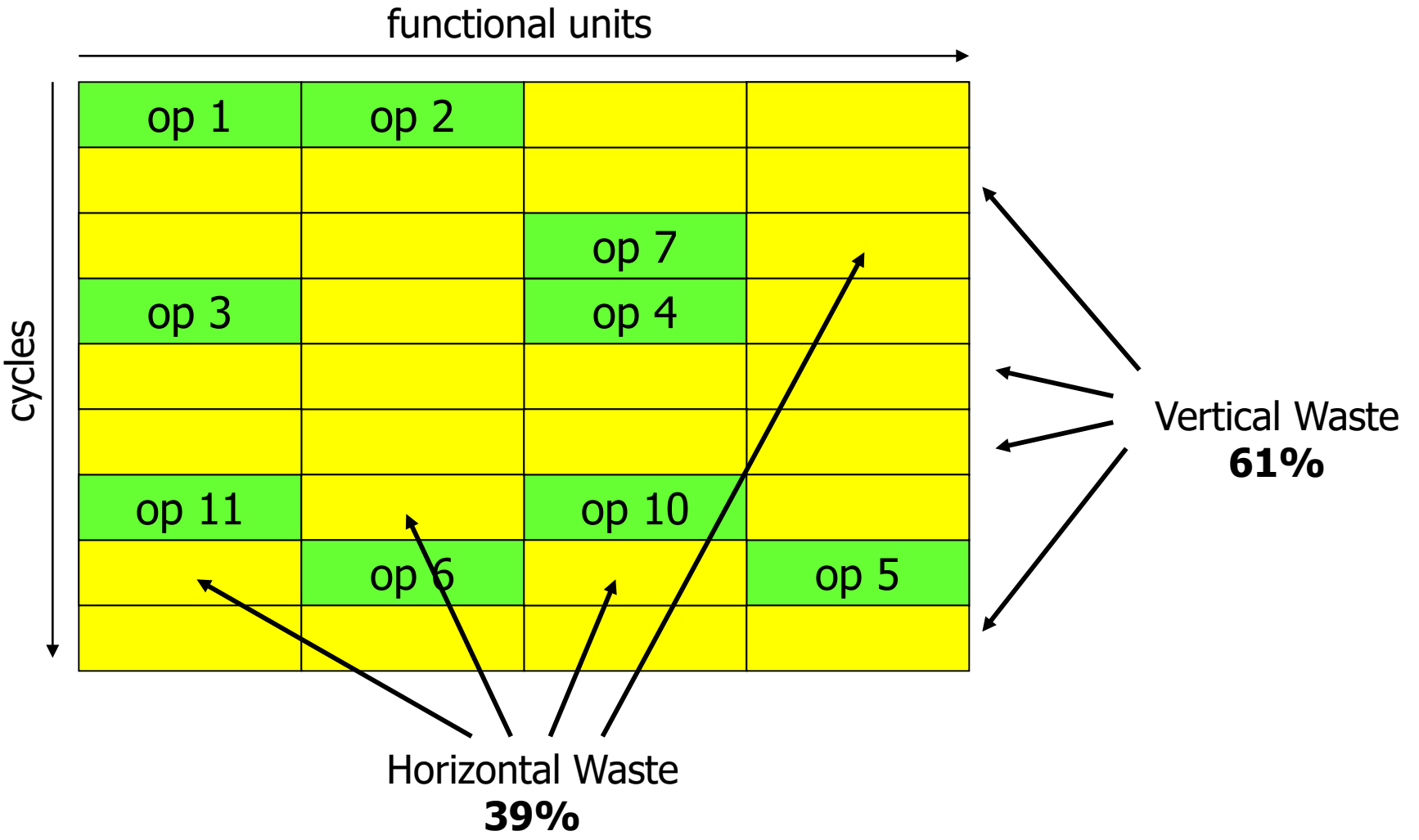


# Sources of Unused Issue Slots



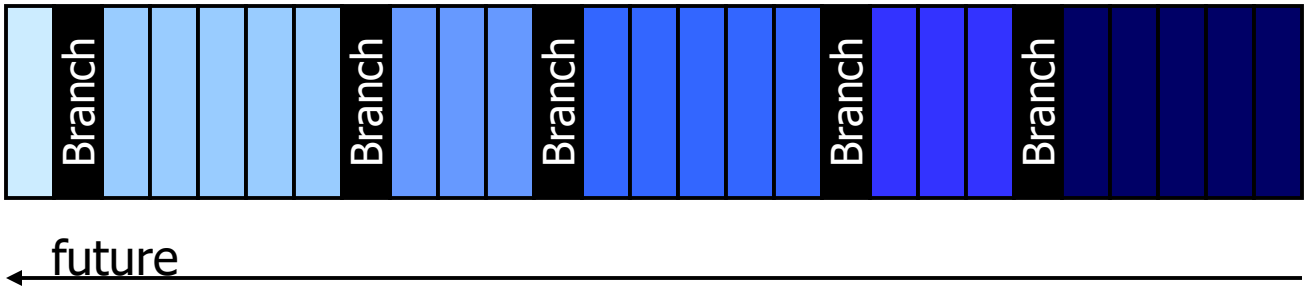

Source: Tullsen et al., © IEEE 1995

# Horizontal and Vertical Waste




Source: Tullsen et al., © IEEE 1995

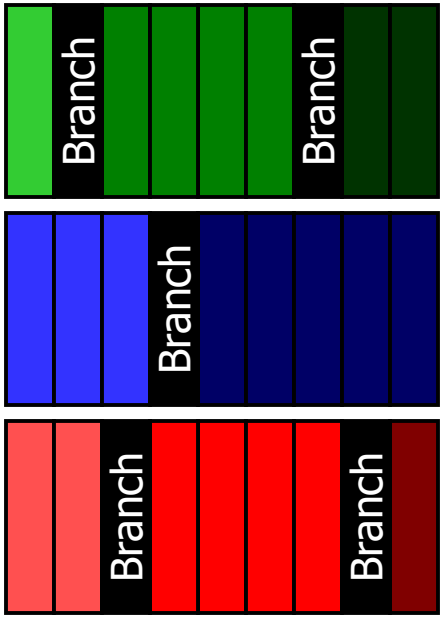
# Multithreading: The Idea




Issue

Rather than enlarging the **depth** of the instruction window (more speculation with lowering confidence!), enlarge its **"width"**

➔ **fetch from multiple threads!**




Issue

# Basic Needs of a Multithreaded Processor

- Processor must be aware of several independent states, one per each thread:
  - Program Counter
  - Register File (and Flags)
  - (Memory)
- Either multiple resources in the processor or a fast way to switch across states

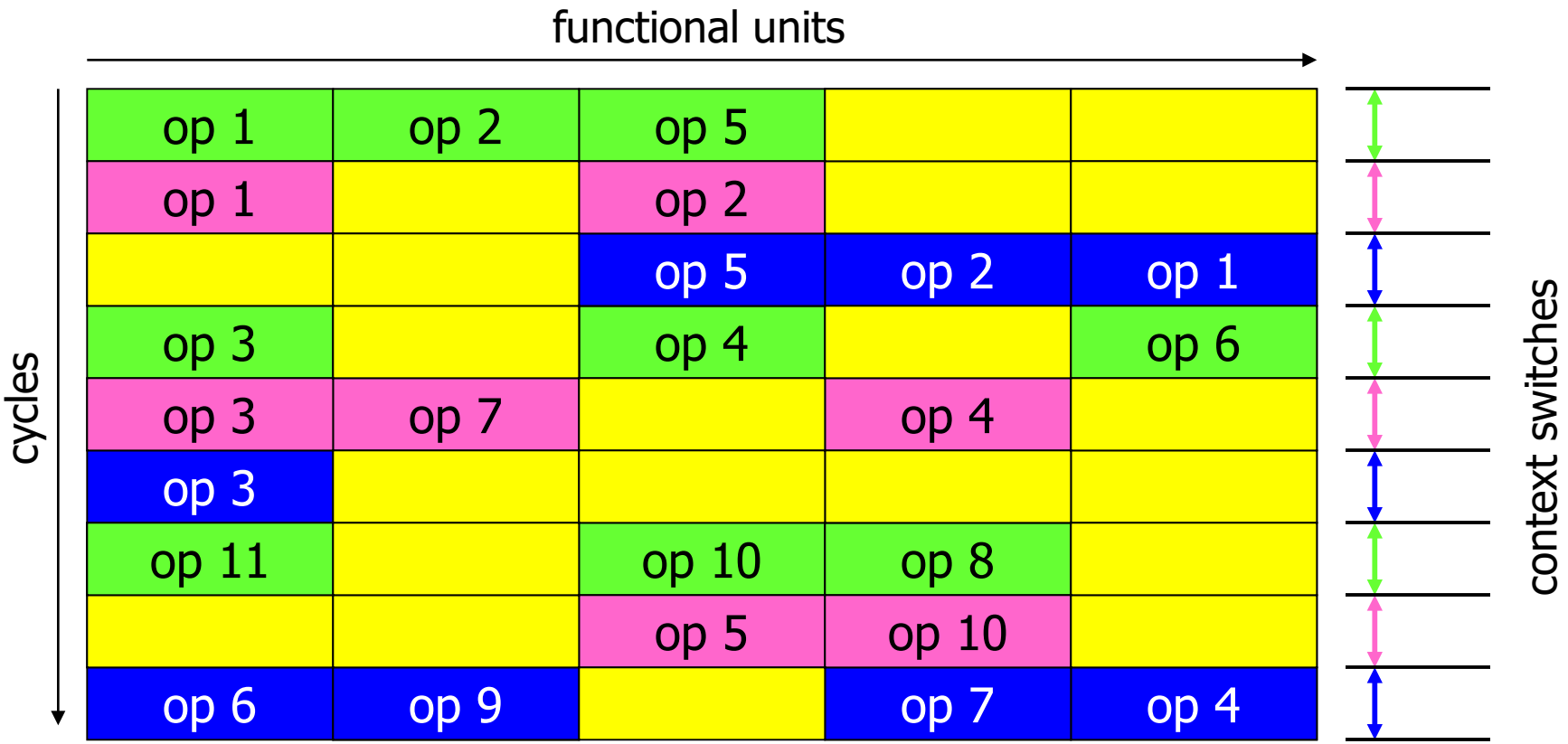
# Thread Scheduling

**When one switches thread?**

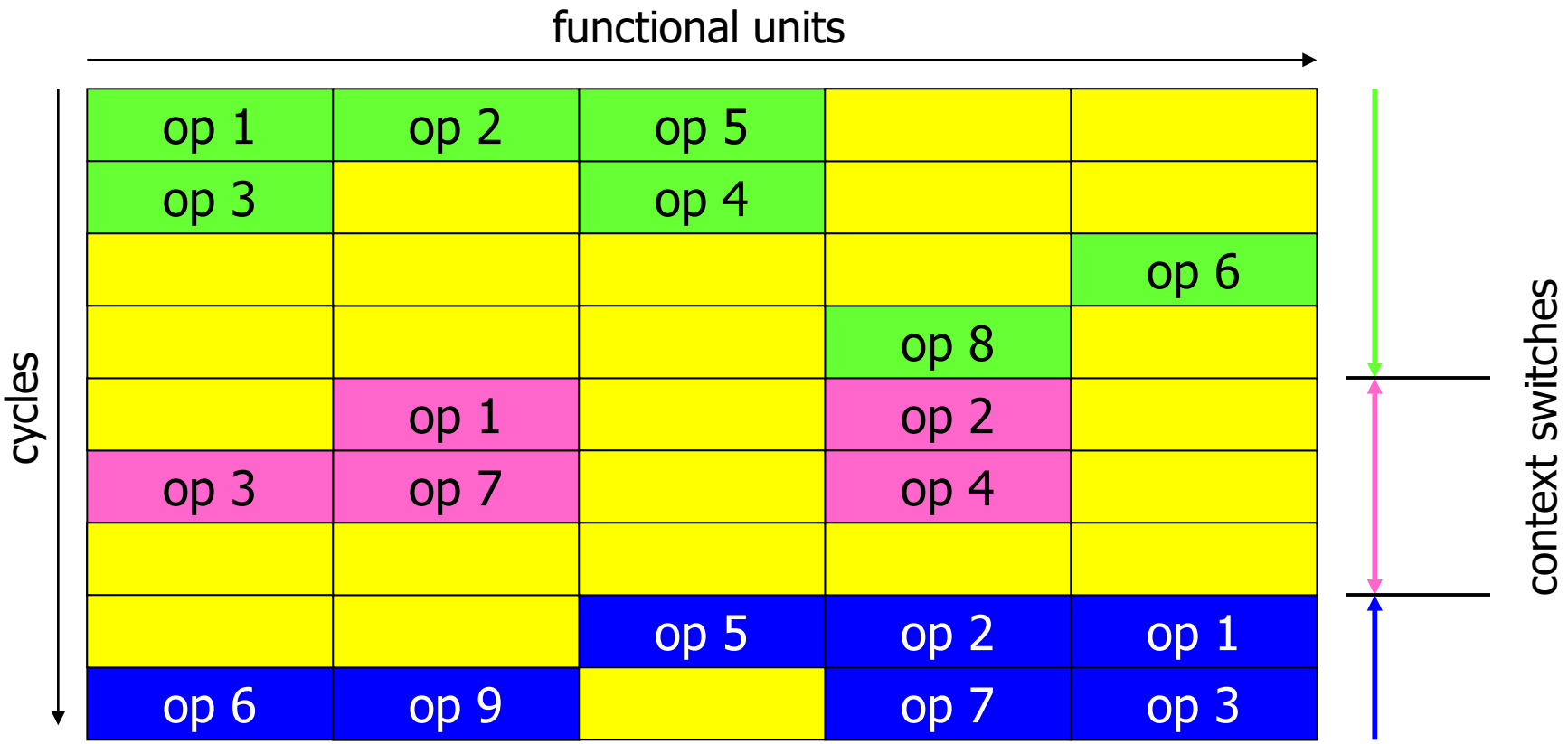
**Which thread will be run next?**

- Simple interleaving options:
  - Cycle-by-cycle multithreading
    - Round-robin selection between a set of threads
  - Block multithreading
    - Keep executing a thread until something happens
      - Long latency instruction found
      - Some indication of scheduling difficulties
      - Maximum number of cycles per thread executed

# Cycle-by-cycle Interleaving (or Fine-Grain) Multithreading



# Block Interleaving (or Coarse-Grain) Multithreading



# Fundamental Requirement

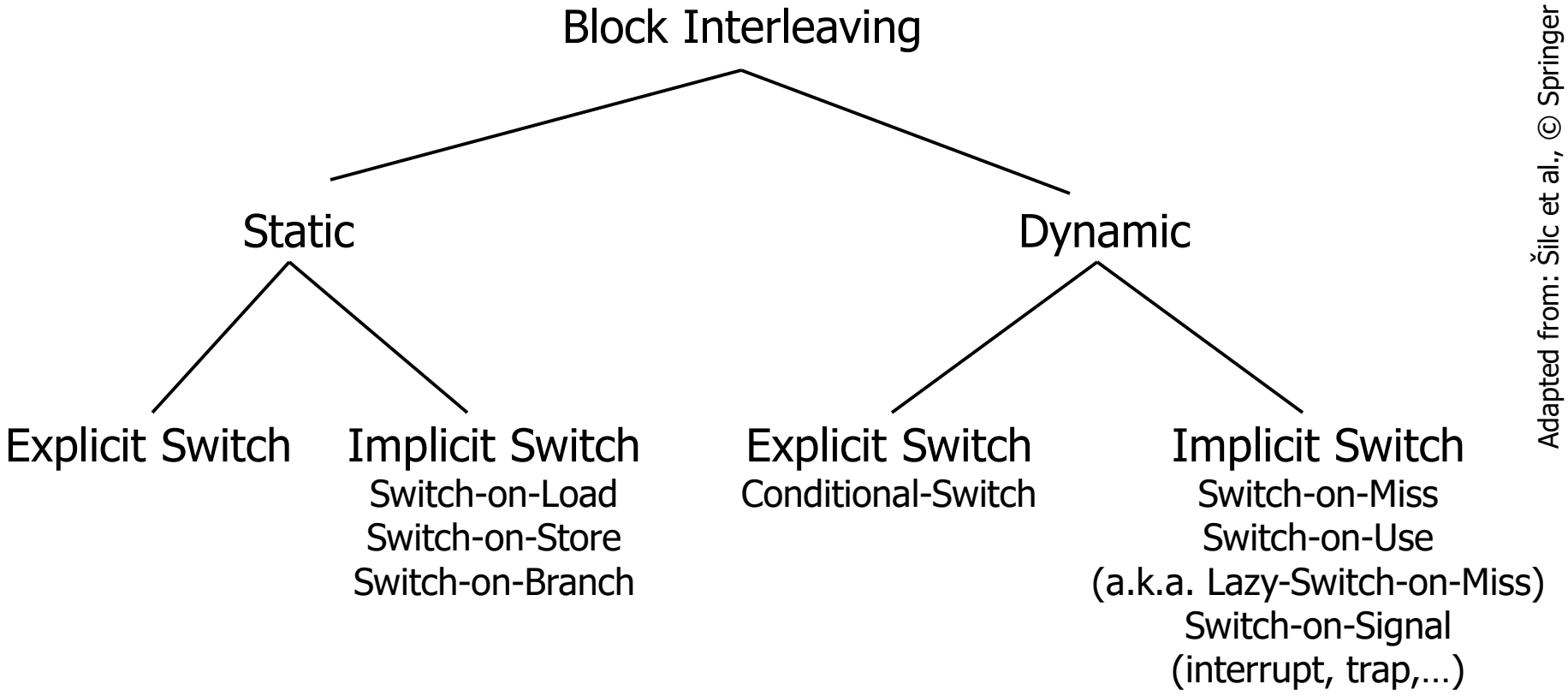
- Key issue in general-purpose processors which has prevented for many years multithreaded techniques to become commercially relevant

**It is not acceptable that single-thread performance goes significantly down or at all**

# Problems of Cycle-by-Cycle Multithreading

- Null time to switch context
  - ➔ Multiple Register Files
- ↗ No need for forwarding paths if threads supported are more than pipeline depth!
  - ➔ Simple(r) hardware
- ↗ Fills well short vertical waste (other threads hide latencies ~ no. of threads)
- ↘ Fills much less well long vertical waste (the thread is rescheduled no matter what)
- ↘ Does not reduce significantly horizontal waste (per thread, the instruction window is not much different…)
- ↘ **Significant deterioration of single thread job**

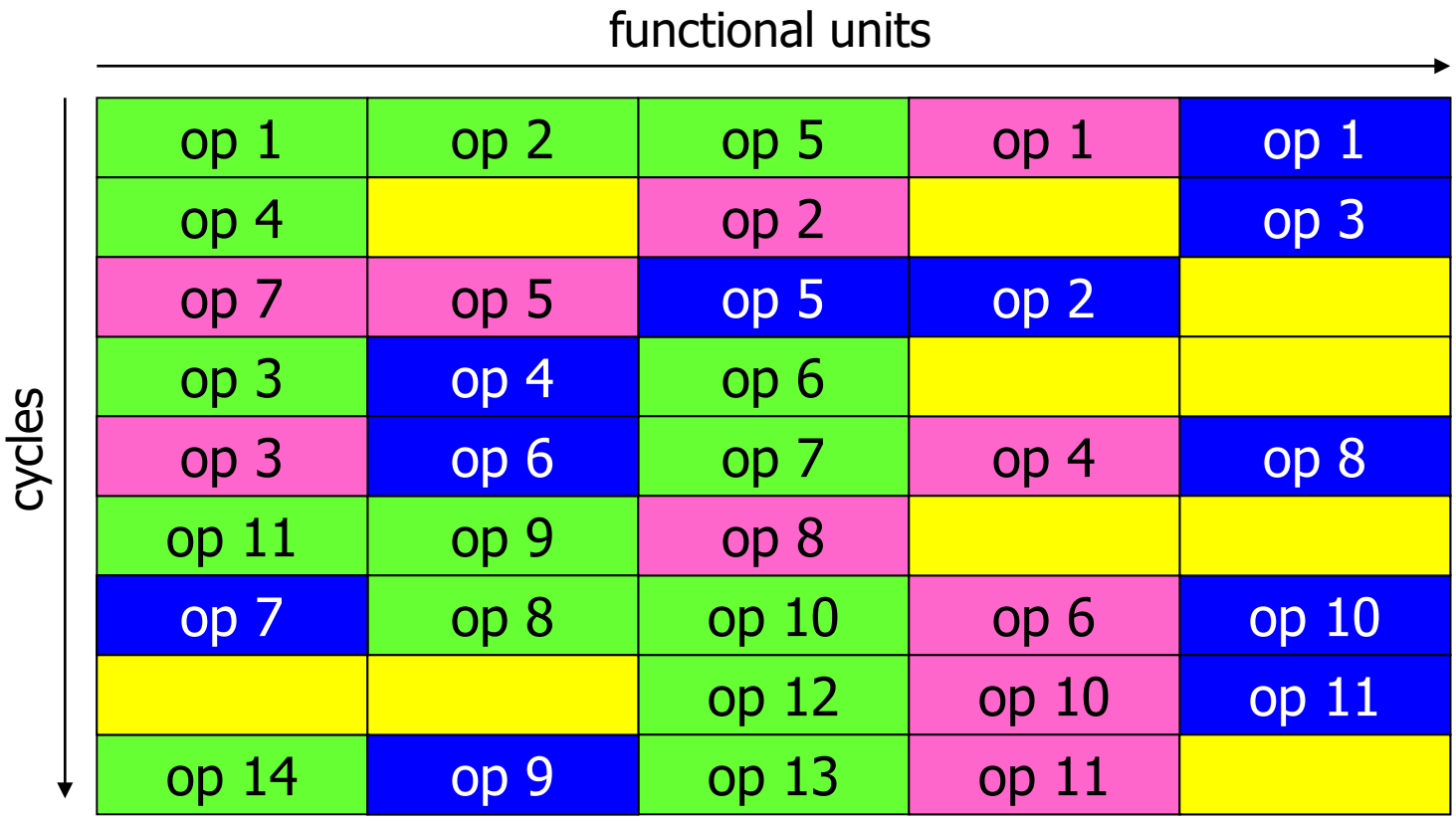
# Block Interleaving Techniques



# Problems of Block Multithreading

- Scheduling of threads not self-evident:
  - What happens of thread #2 if thread #1 executes perfectly well and leaves no gap?
  - Explicit techniques require ISA modifications ➔ Bad…
- ↗ More time allowable for context switch
- ↗ Fills very well long vertical waste (other threads come in)
- ↘ Fills poorly short vertical waste (if not sufficient to switch context)
- ↘ Does not reduce almost at all horizontal waste

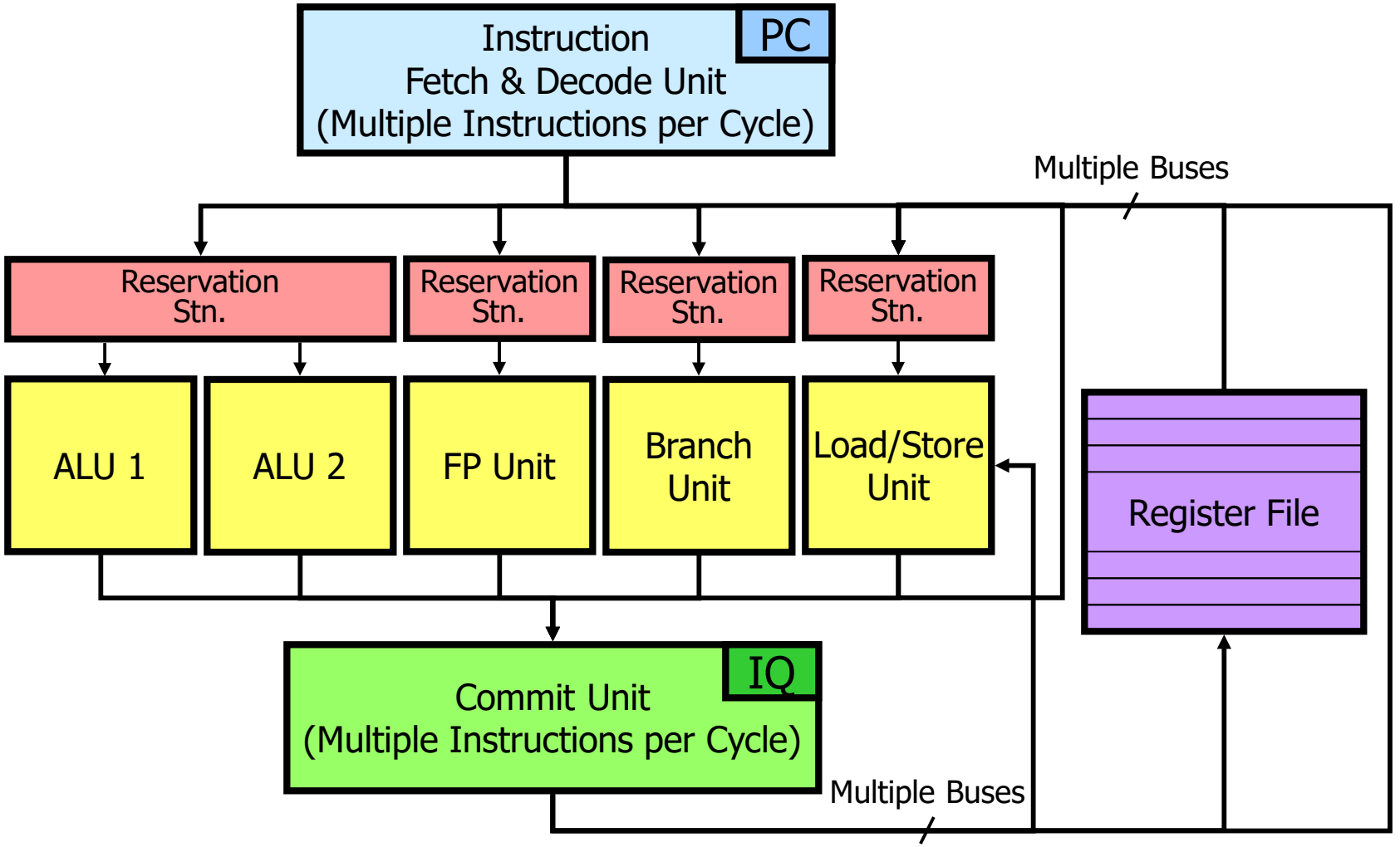
# Simultaneous Multithreading (SMT): The Idea



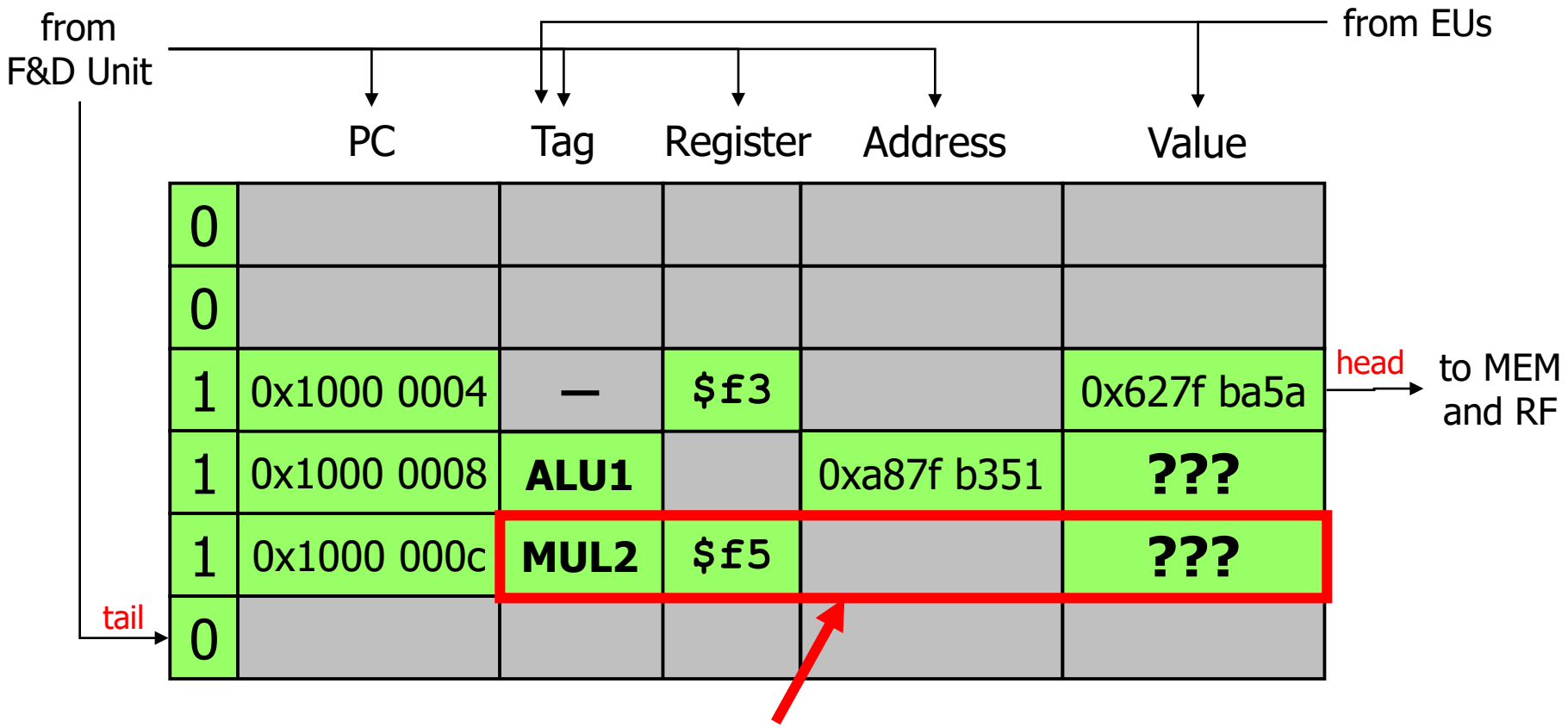
# Several Simple Scheduling Possibilities

- Prioritised scheduling?
  - Thread #0 schedules freely
  - Thread #1 is allowed to use #0 empty slots
  - Thread #2 is allowed to use #0 and #1 empty slots, etc.
- Fair scheduling?
  - All threads compete for resources
  - If several threads want the same resource, round-robin assignment

# Superscalar Processor



# Reorder Buffer

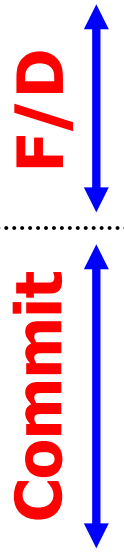



**Register Renaming**

(between fetch/decode and commit)

# What Must Be Added to a Superscalar to Achieve SMT?

- Multiple program counters (= threads) and a policy for the instruction fetch unit(s) to decide which thread(s) to fetch
- Multiple or larger register file(s) with at least as many registers as logical registers for all threads
- Multiple instruction retirement (e.g., per thread squashing)

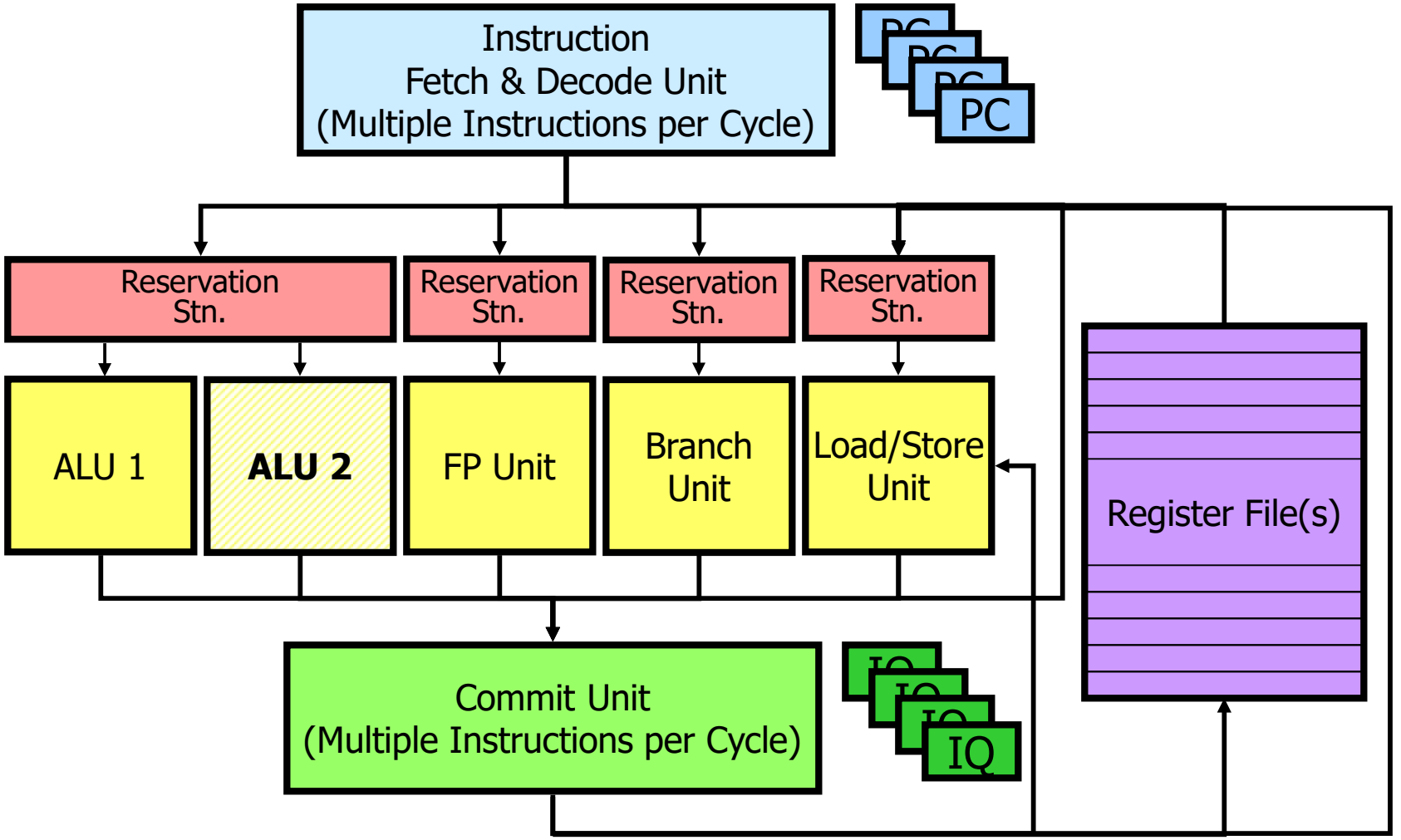



➔ No changes needed in the execution path

And also:

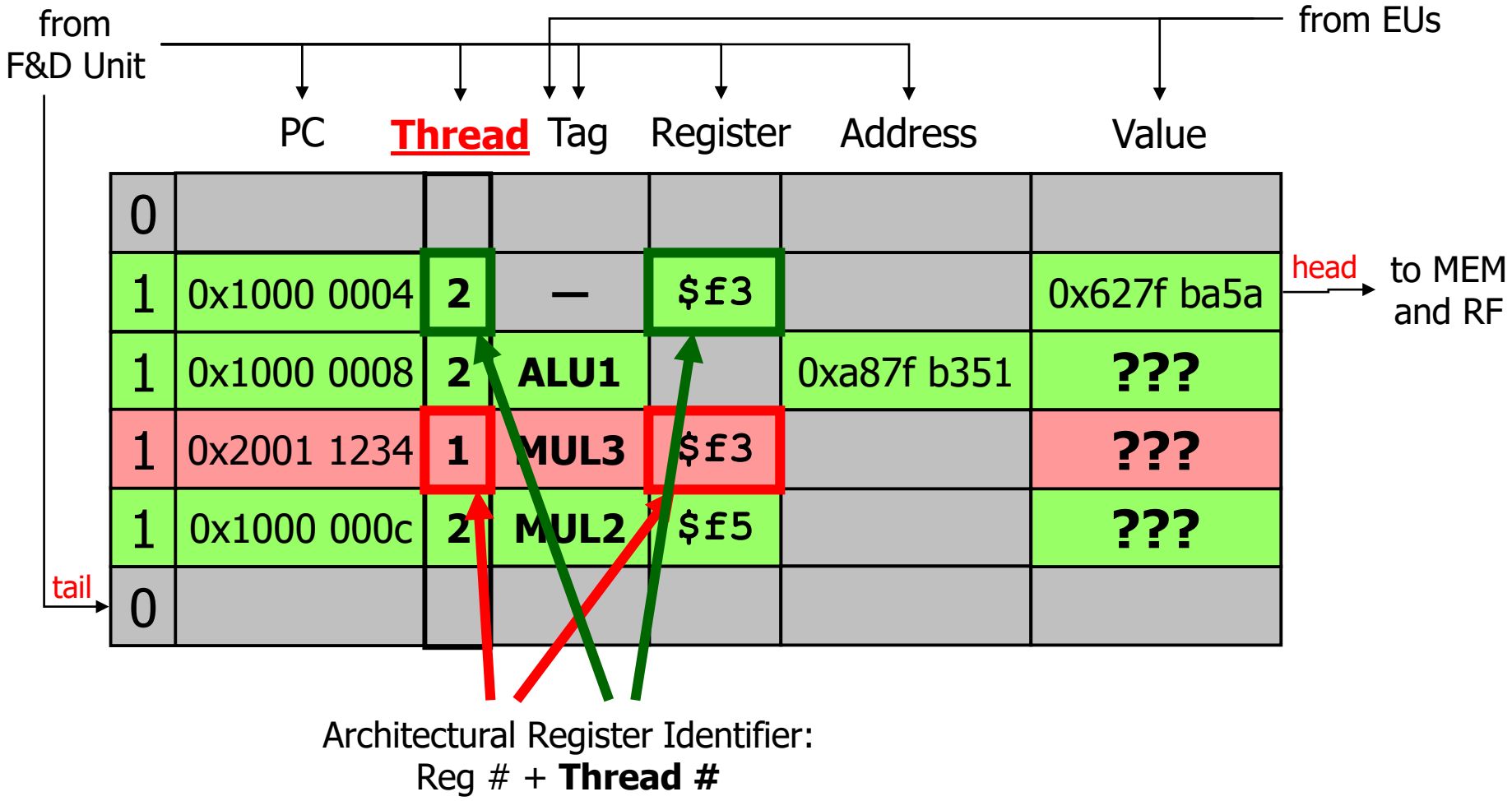
- Thread-aware branch predictors (BTBs, etc.)
- Per-thread Return Address Stacks

# SMT Processor as a Natural Extension of a Superscalar



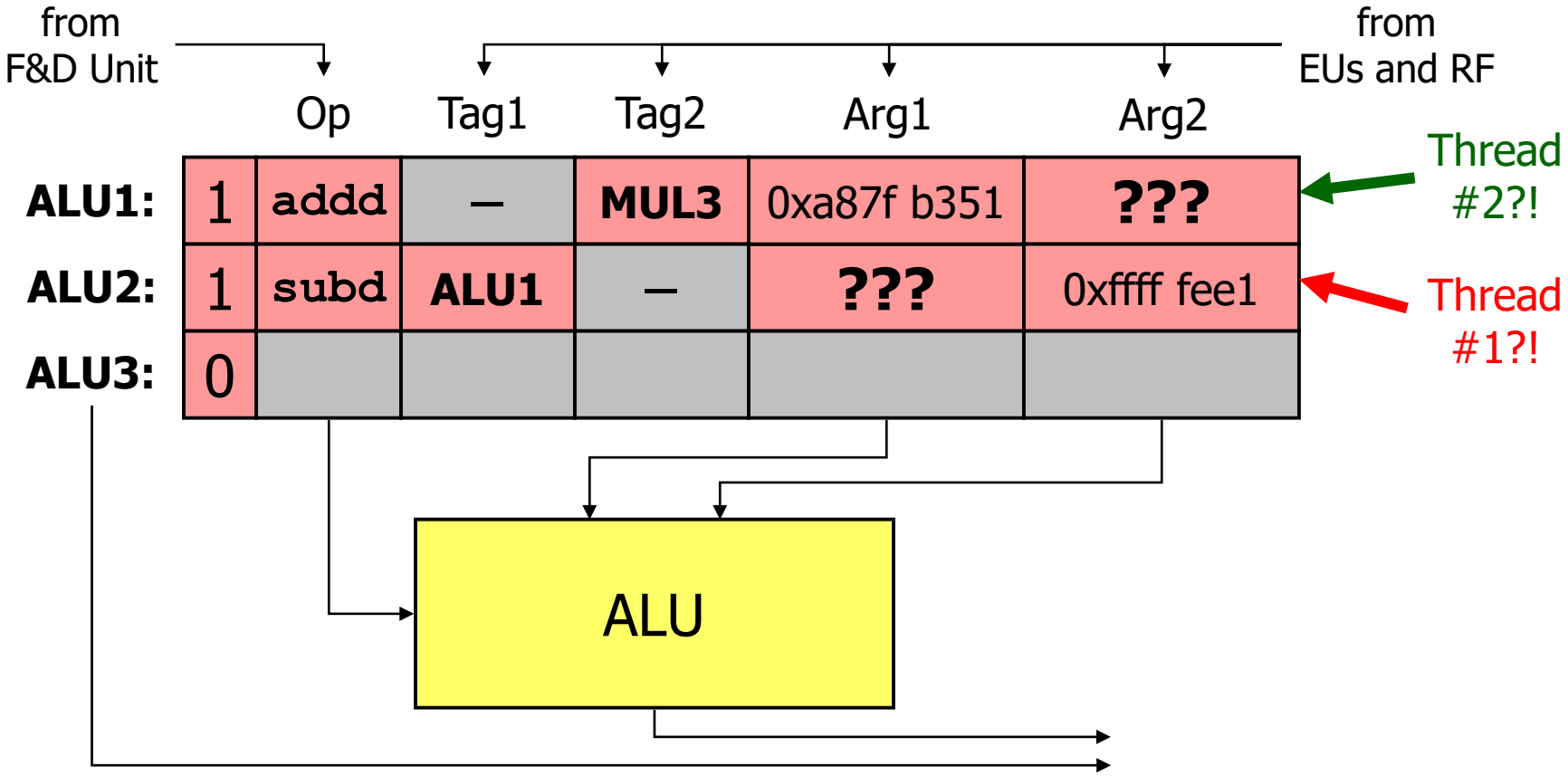
# Reorder Buffer Remembers the Thread of Origin

- Some changes to the reorder buffer in the Commit Unit—e.g.:



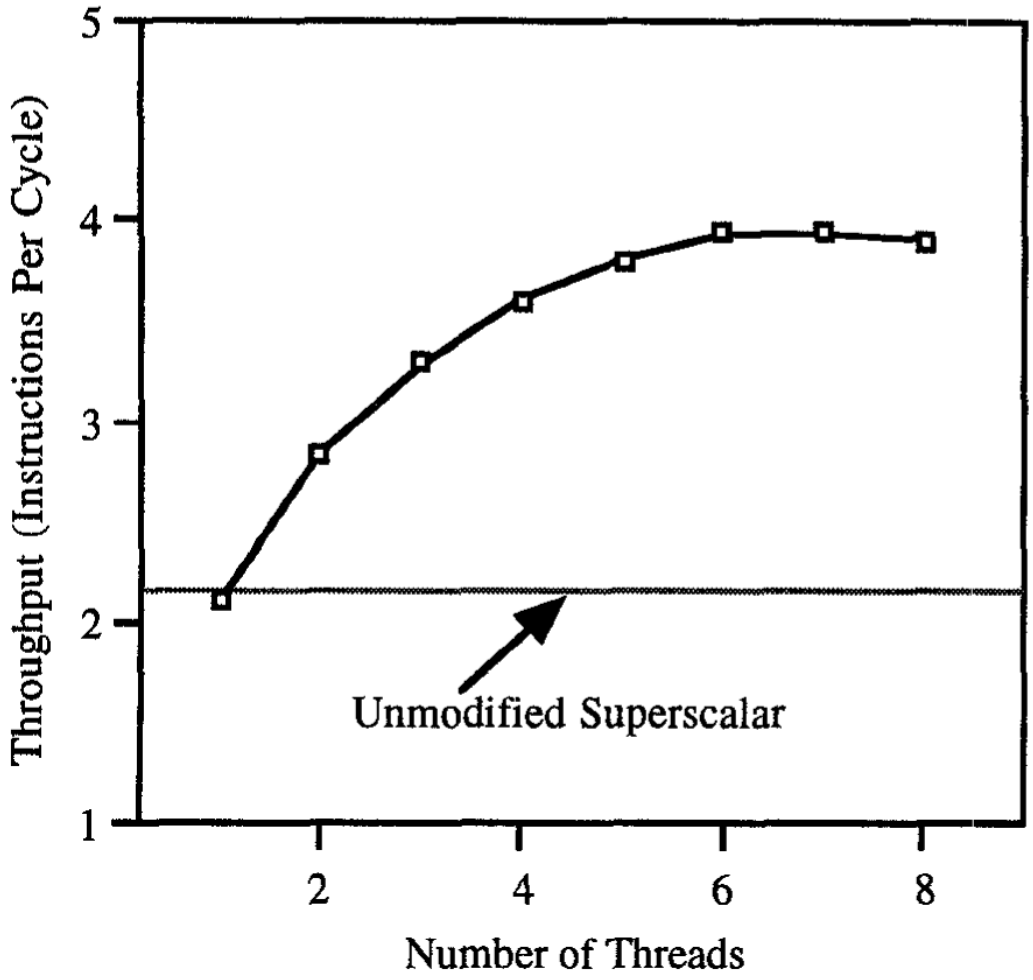
# Reservation Stations

- Reservation stations **do not** need to know which thread an instruction belongs to
- Remember: operand sources are renamed—physical regs, tags, etc.



# Does It Work?!

Fetch+Decode throughput = 8 IPC




Source: Tullsen et al., © IEEE 1996

# Main Results on Implementability SMT vs. Superscalar

From [TullsenJun96]:

- Instruction scheduling not more complex
- Register File datapaths not more complex (but much larger register file!)
- Instruction Fetch Throughput is attainable even without more fetch bandwidth
- Unmodified cache and branch predictors are appropropriate also for SMT
- SMT achieves better results than aggressive superscalar

# Where to Fetch?

- **Static** solutions: Round-robin
  - Each cycle 8 instructions from 1 thread
  - Each cycle 4 instructions from 2 threads, 2 from 4,…
  - Each cycle 8 instructions from 2 threads, and forward as many as possible from #1 then when long latency instruction in #1 pick rest from #2
- **Dynamic** solutions: Check execution queues!
  - Favour threads with minimal # of in-flight branches
  - Favour threads with minimal # of outstanding misses
  - **Favour threads with minimal # of in-flight instructions**
  - Favour threads with instructions far from queue head

# What to Issue?

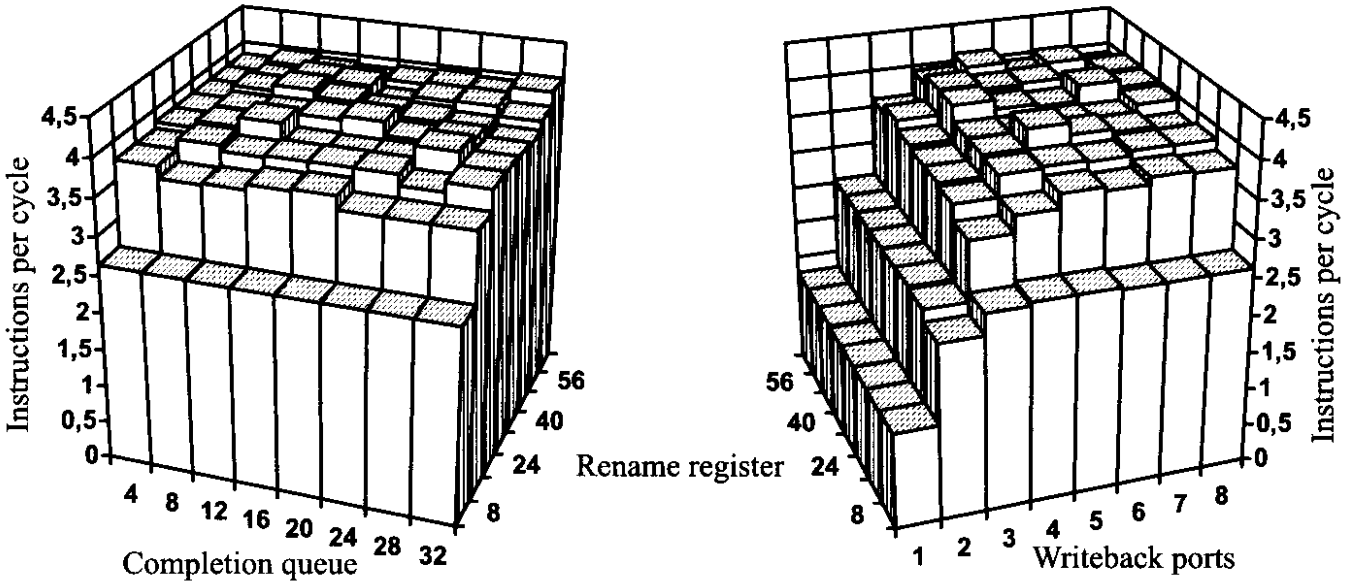
- Not **exactly** the same as in superscalars…
  - In superscalar: oldest is the best (least speculation, more dependent ones waiting, etc.)
  - In SMT not so clear: branch-speculation level and optimism (cache-hit speculation) vary across threads
- One can think of many selection strategies:
  - Oldest first
  - Cache-hit speculated last
  - Branch speculated last
  - Branches first…
- Important result: **doesn't matter too much!**

➔ **Issue Logic (critical in superscalars) can be left alone**

# Importance of Accurate Branch Prediction in SMT vs. Superscalar

- Reduce the impact of Branch Prediction was one of the qualitative initial motivations
- Results from [TullsenJun96]:
  - Perfect branch prediction advantage
    - **25%** at 1 thread
    - 15% at 4 threads
    - **9%** at 8 threads
  - Losses due to suppression of speculative execution
    - **-7%** at 8 threads
    - **-38%** at 1 thread (➔ speculation was a good idea…)

# Bottlenecks
# Sources of Unused Issue Slots



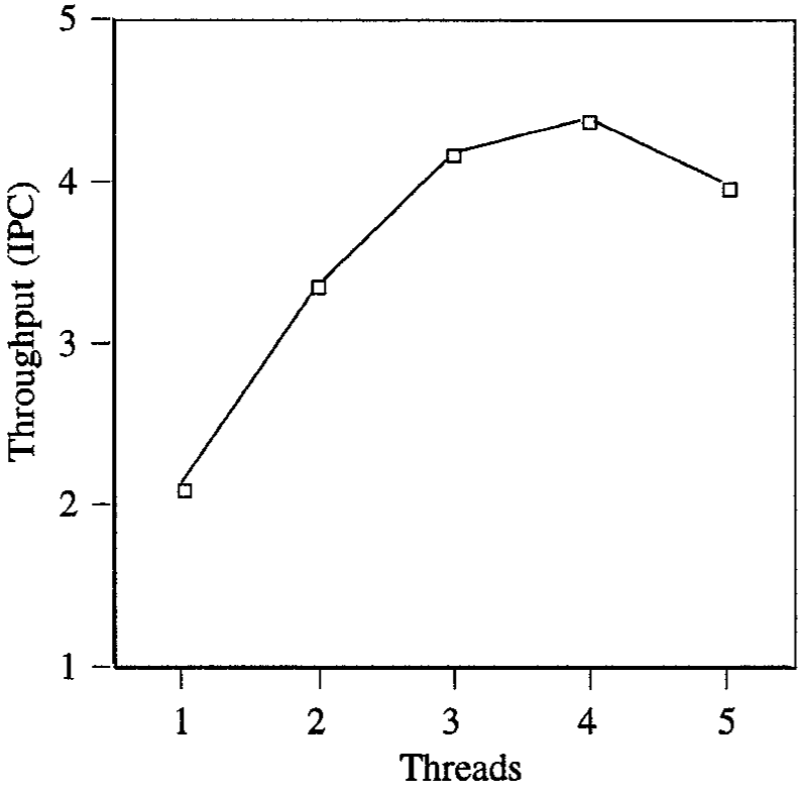

Source: Šilc et al., © Springer 1999

- Completion queue not very relevant (remember: this is out of the execution path…)
- **Rename register count** important
- Most critical: number of register **writeback ports**

SMT for **utilisation rate (EUs)** not **bandwidth!**…

# Bottlenecks

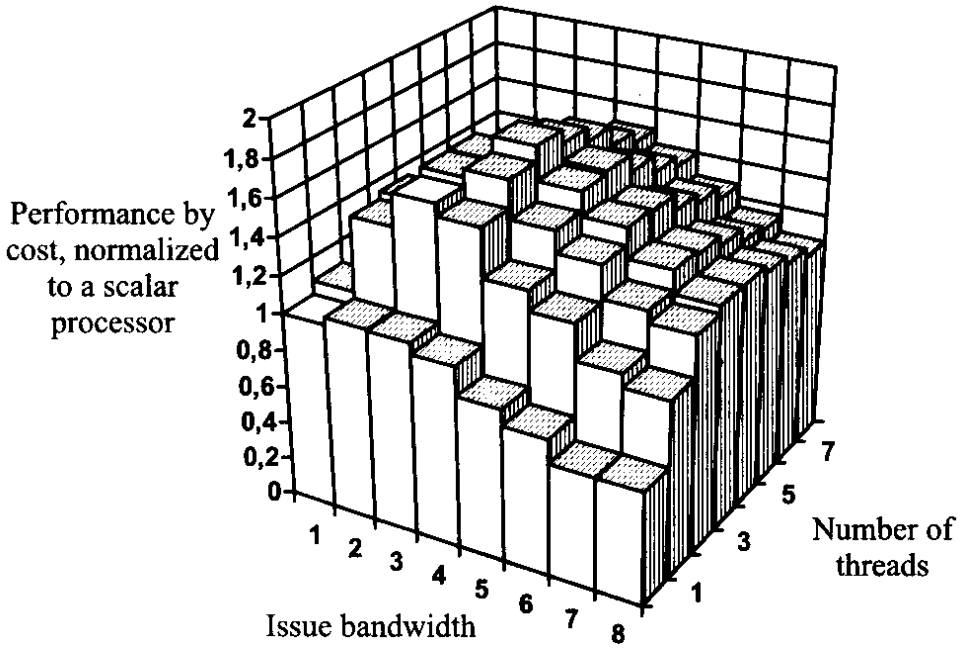
- Fetch and memory throughput are still bottlenecks
  - Fetch: branches, etc.
  - Memory not addressed
- Performance vs. # of rename registers (8T) in addition to the architectural ones
  - Infinite: +2%
  - 100: ref.
  - 90: -1%
  - 80 -3%
  - 70 -6%
- Register file access time likely limit to # of threads




**IPC vs. # threads**
**200 physical registers**

Source: Tullsen et al., © IEEE 1996

# Performance (IPC) per Unit Cost



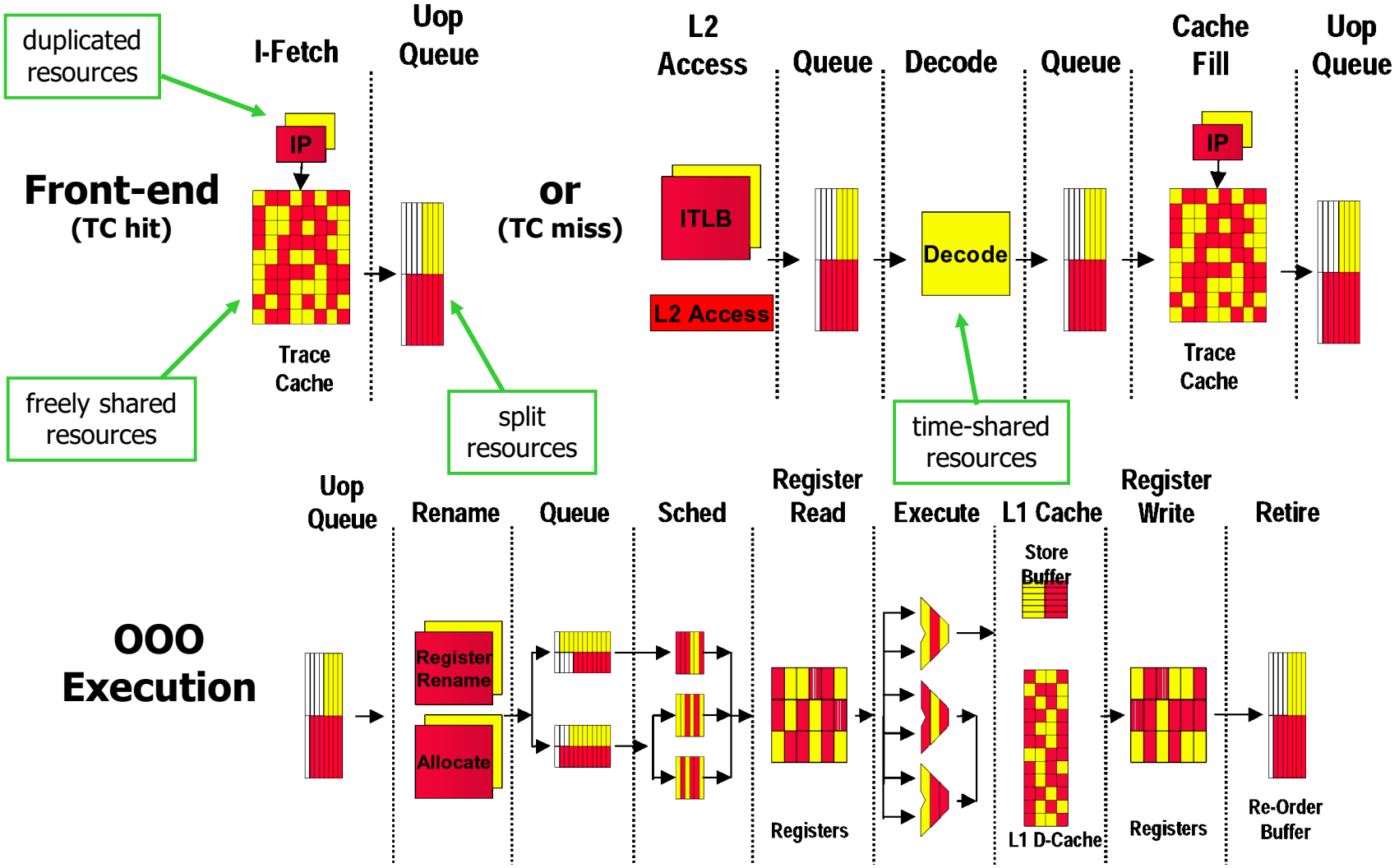

Source: Šilc et al., © Springer 1999

- Superscalars are cheap only for relatively small issue bandwidth, then quickly down
- SMT improves significantly the picture already with 2 threads and maximum moves to larger issue bandwidths with more threads

# Introduction of SMT in Commercial Processors

- Compaq Alpha 21464 (EV8)
  - 4T SMT
  - Project killed June 2001
- Intel Pentium IV (Xeon)
  - 2T SMT
  - Availability since 2002 (already there before, but not enabled)
  - 10-30% gains expected
- SUN Ultra III
  - 2-core CMP, 4T SMT

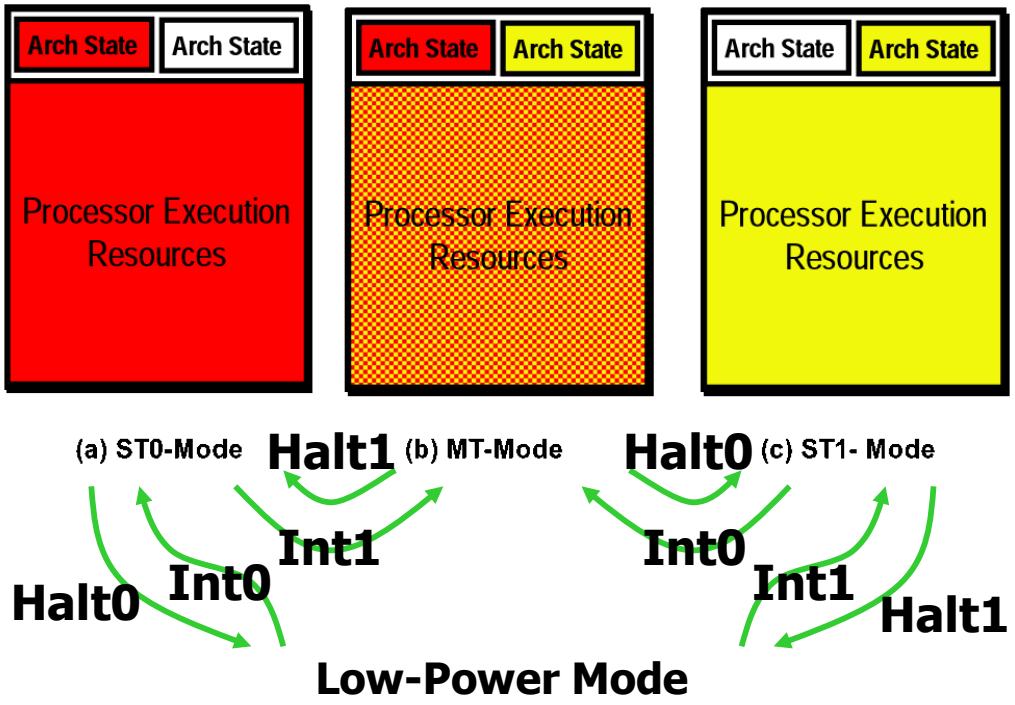
# Intel SMT: Xeon Hyper-Threading Pipeline




Source: Marr et al., © Intel 2002

# Intel SMT: Xeon Hyper-Threading Switching Off Threads

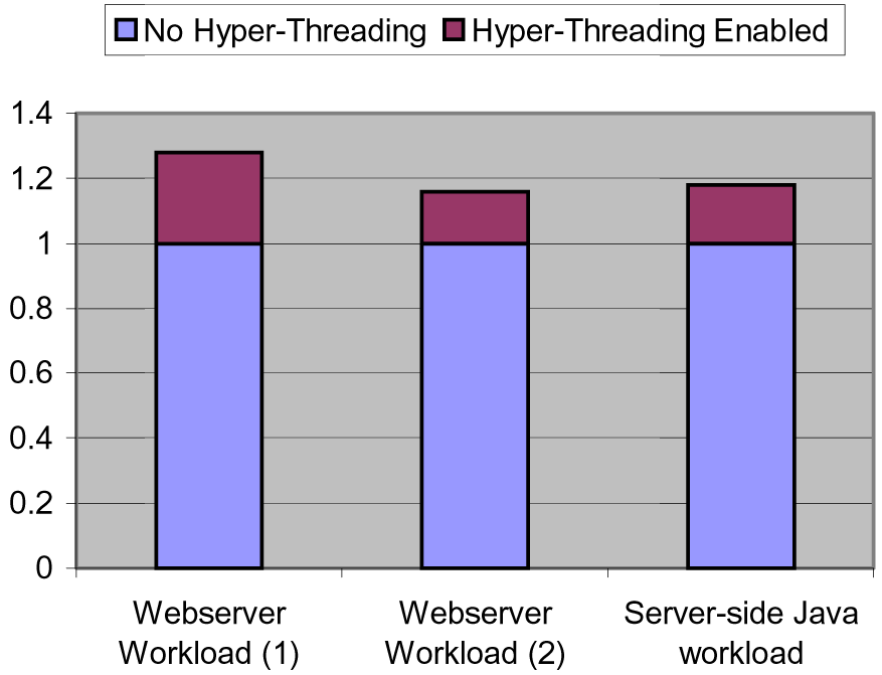
- What happens when there is only one thread? What does the OS when there is nothing to do? Ahem…
  - ➔ Four modes: Low-power, ST0, ST1, and MT




Source: Marr et al., © Intel 2002

# Intel SMT: Xeon Hyper-Threading Goals and Results

- Minimum additional cost: SMT = approx. 5% area
- No impact on single-thread performance
  - Recombine partitioned resources
- Fair behaviour with 2 threads




Source: Marr et al., © Intel 2002

# And Now, What's Next?

- Key ingredients for success so far:
  - Maximise compatibility, no info from programmers beyond straight sequential code and coarse threads
  - Aggressive prediction and speculation of anything predictable
  - Use irregular, fine-grained parallelism (ILP): it is "easier" to extract, can be done at runtime,…
- Problems:
  - Branch prediction accuracy hard to improve
  - Hard to exploit ILP any further within a thread

# References on Simultaneous Multithreading

- AQA 5th ed., Chapter 3
- PA, Sections 6.3, 6.4, and 6.5
- CAR, Chapter 5—Introduction
- D. M. Tullsen et al., *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor*, ISCA, 1996
- H. Marr et al., *Hyper-Threading Technology Architecture and Microarchitecture*, Intel Technology Journal, Q1, 2002