

CS-200

Computer Architecture

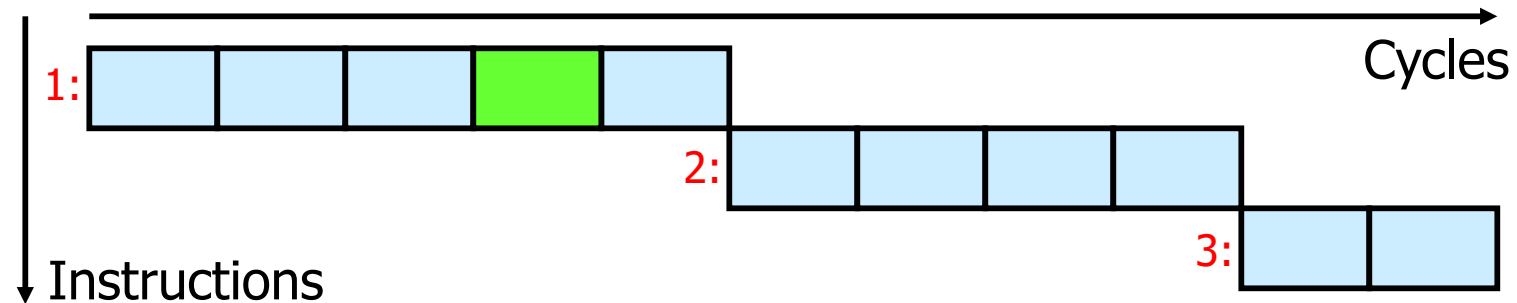
Part 4d. Instruction Level Parallelism

Dynamic Scheduling

Paolo Ienne
<paolo.ienne@epfl.ch>

Starting Point (Programmer Model)

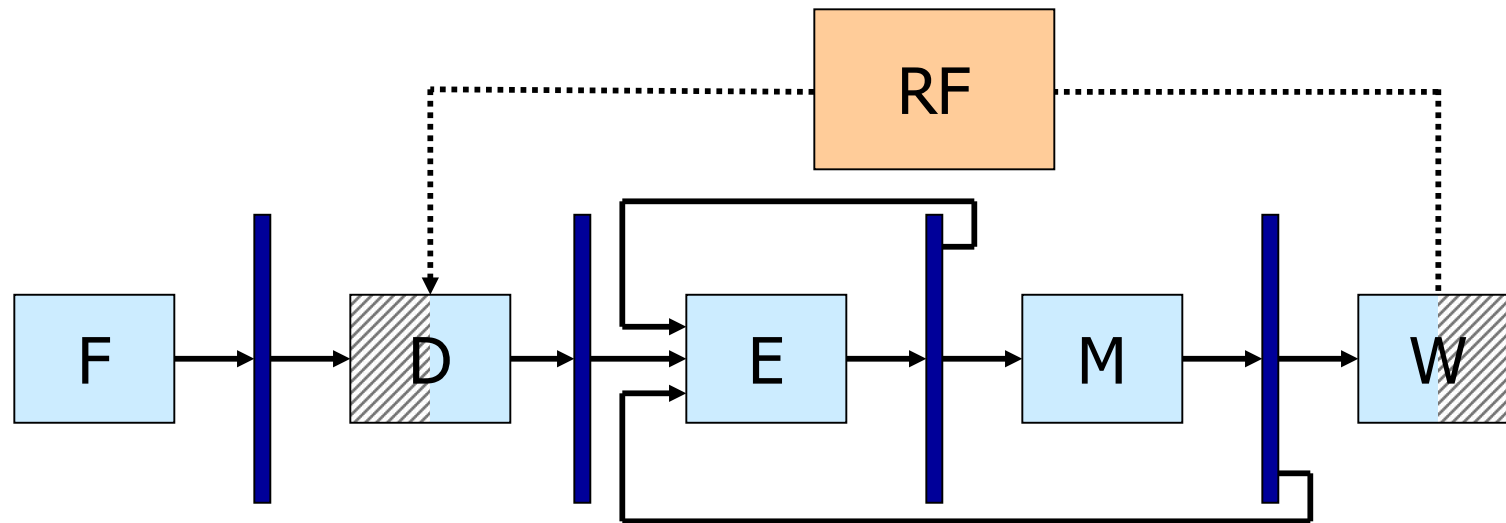
- Sequential multicycle processor



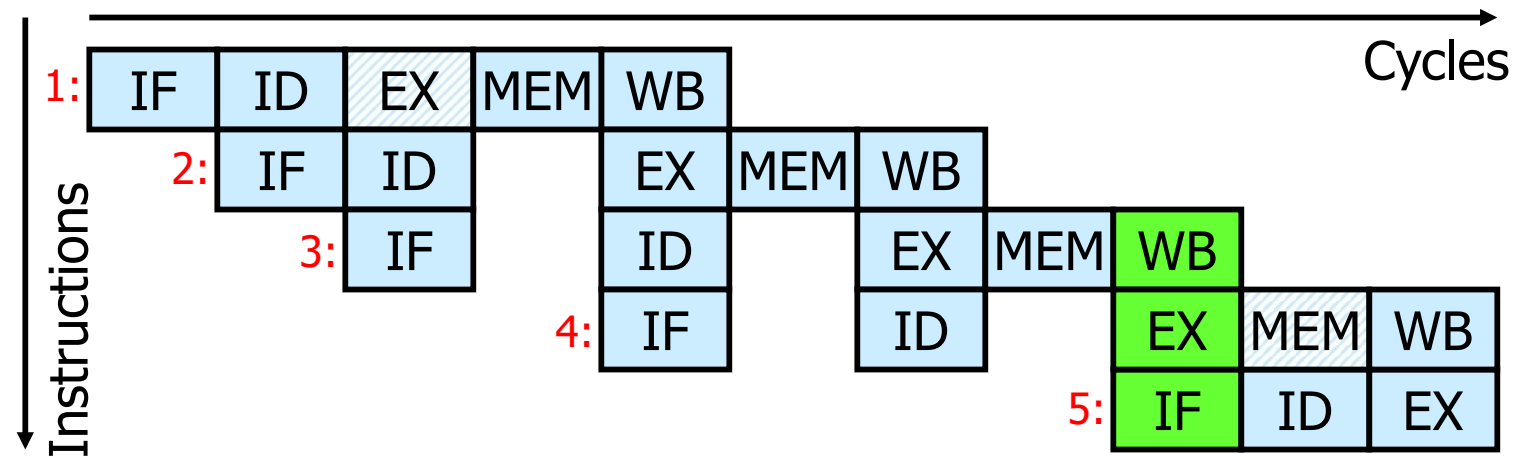
- The processing of each instruction individually as been squeezed as much as possible to make execution as fast as possible
- What's next?!

Pipelined Processor

- For instance, 5-stage pipeline with all forwarding paths ($E \rightarrow E$, $M \rightarrow E$, and $W \rightarrow D$) typical of **MIPS** and **RISC-V**

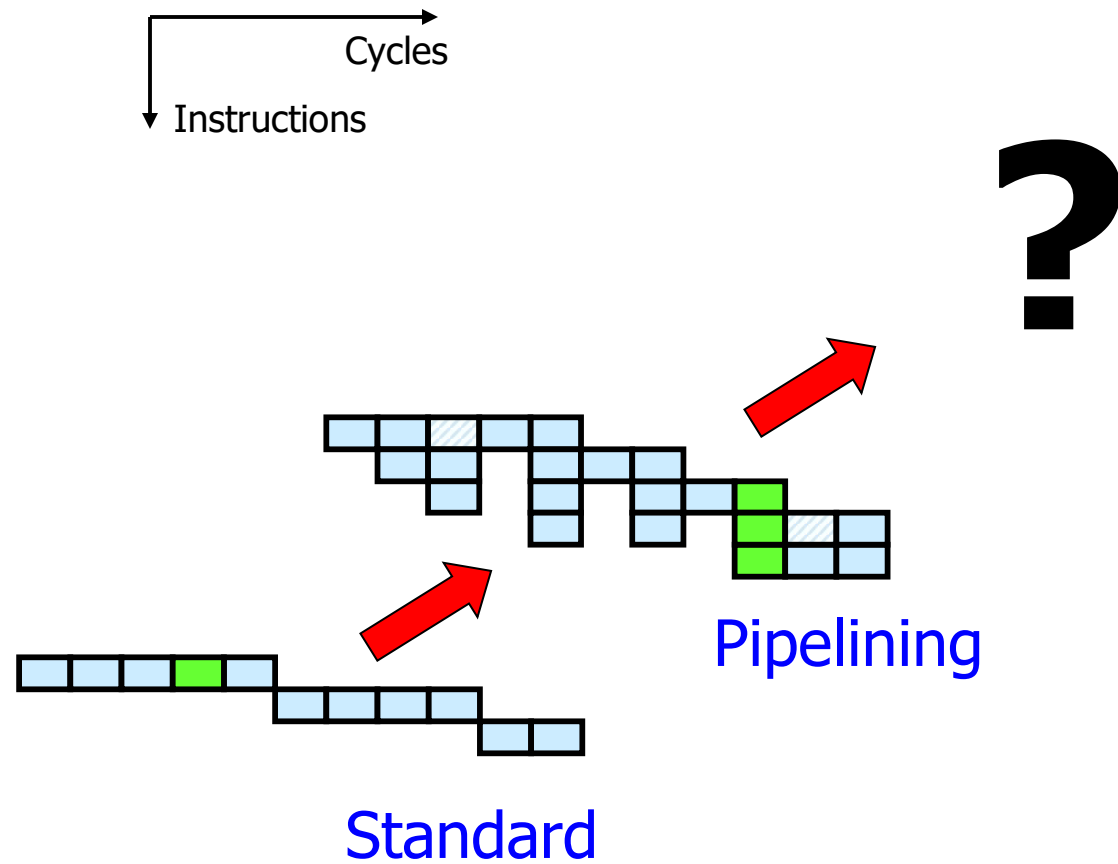


First Step: Pipelining



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

ILP So Far...

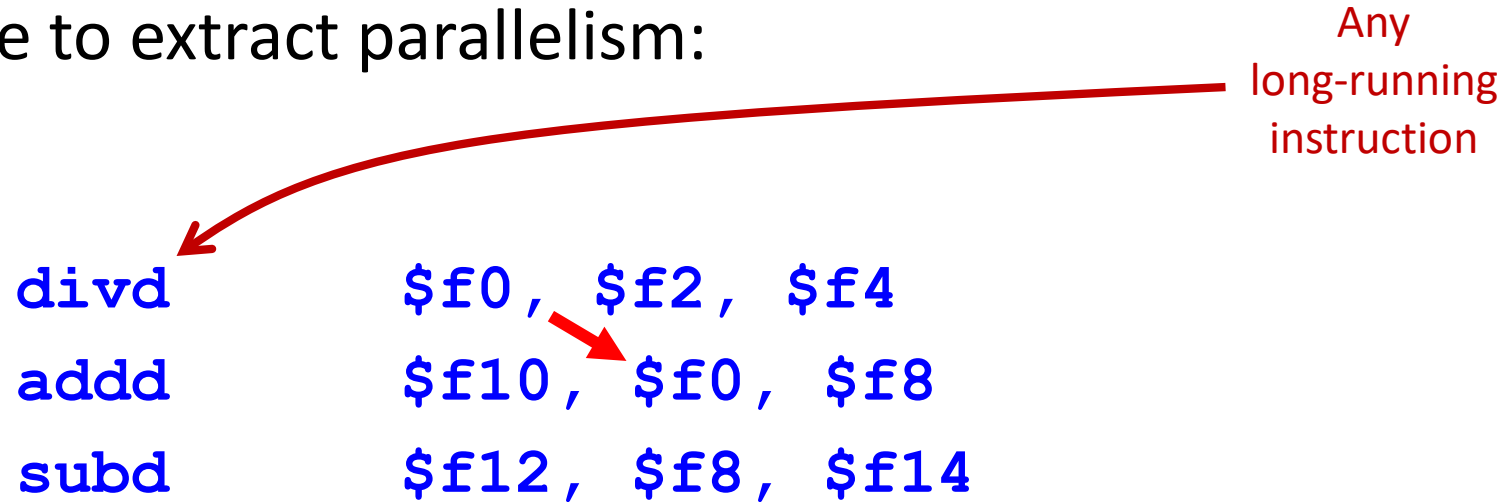


Simple Pipelining

- Scope for **parallelism is limited**:
 - **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)
 - **Control hazards** limit the usability of the pipeline
 - Must squash fetched and decoded instruction following a branch
- **Rigid sequencing**
 - Special “slots” for everything even if sometimes useless (e.g., **M**)
 - Every instruction must be **coerced to the same framework** (floating point vs. integer?)
 - Structural hazards avoided “by construction”

Dynamic Scheduling: The Idea

- Extend the scope to extract parallelism:



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

Any
long-running
instruction

- 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

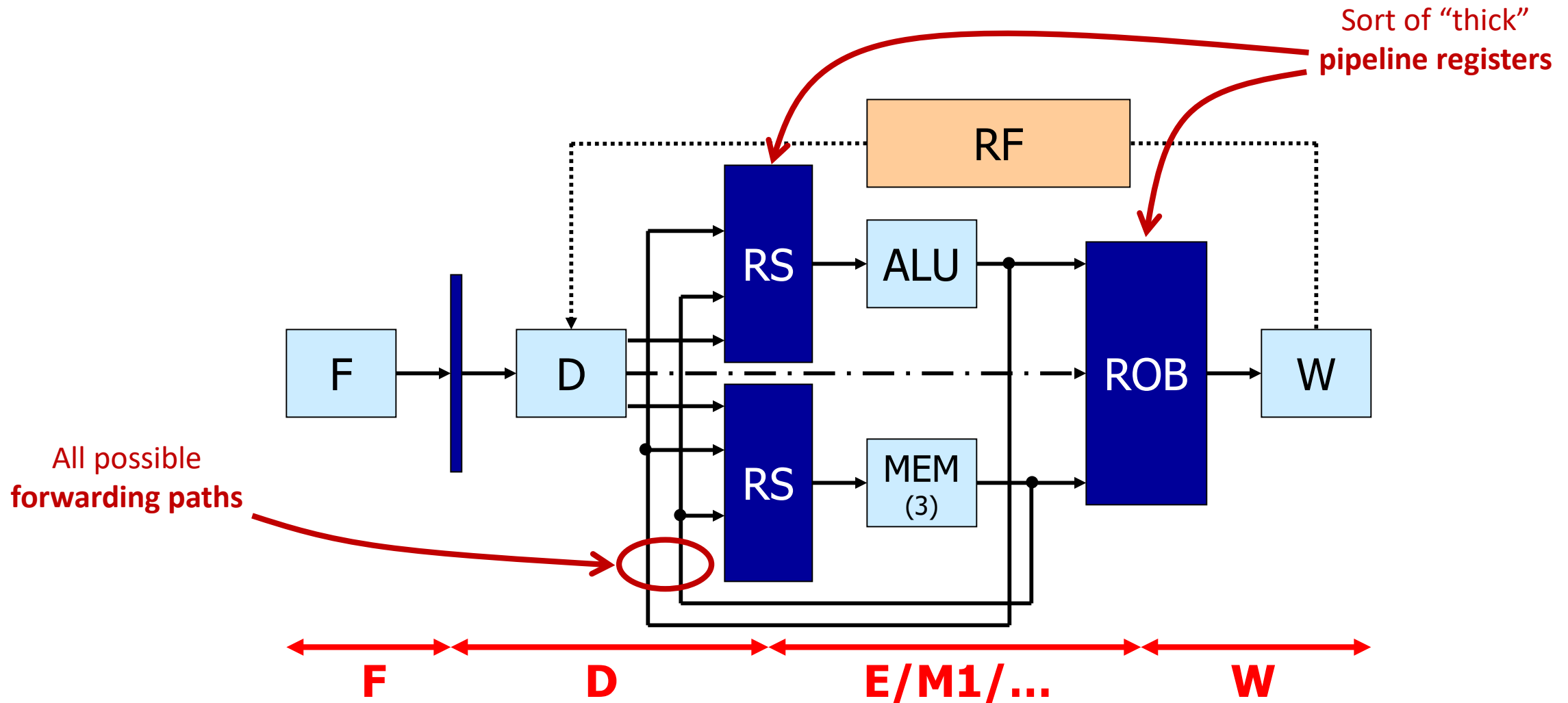
Fetch and Decode

Execute

Writeback

- Clearly, results will also **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**

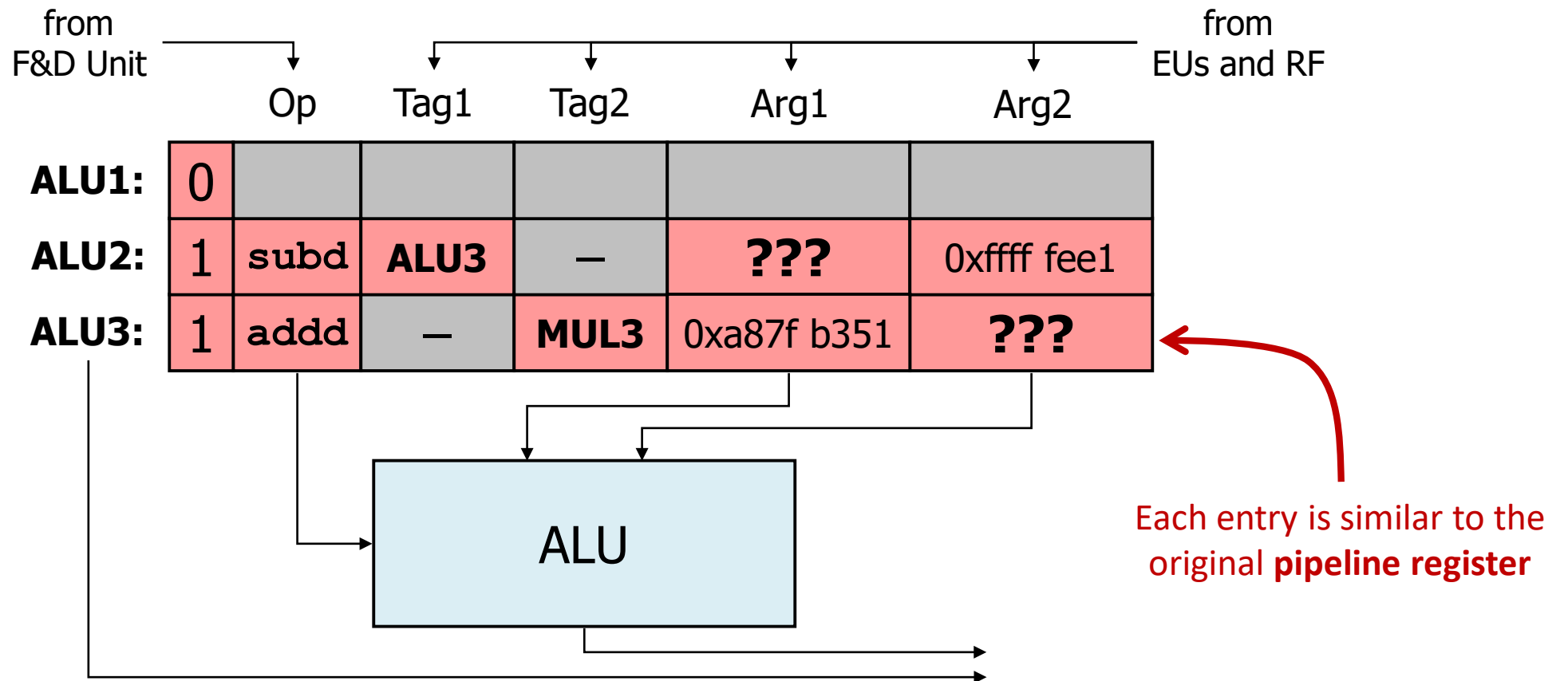
- Are the operands ready to start execution?
- Old problem

- **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



Reservation
Station

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 Dependencies

divd \$f0, \$f1, \$f2
add \$f3, \$f0, \$f4
subd \$f4, \$f5, \$f6
adddi \$f0, \$f5, 10



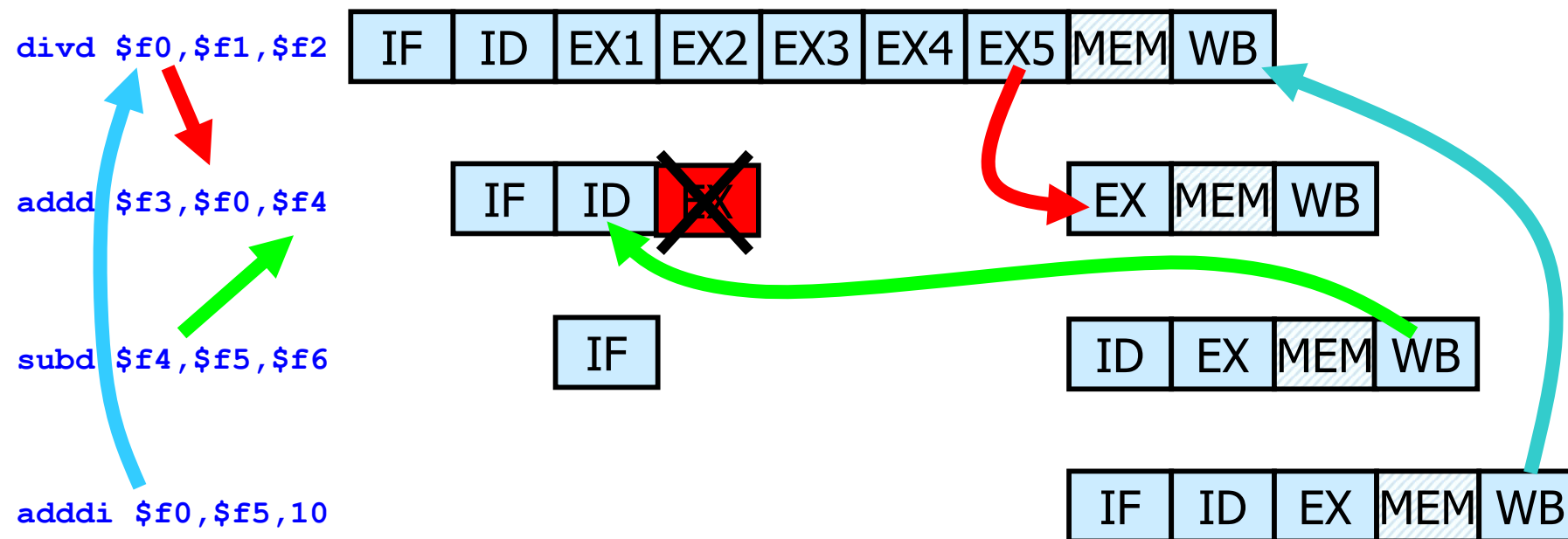
- **add** has a **RAW** dependence on **divd**
- **subd** has a **WAR** dependence on **add**
- **adddi** has a **WAW** dependence on **divd**

True or **data**
dependencies

Name
dependencies

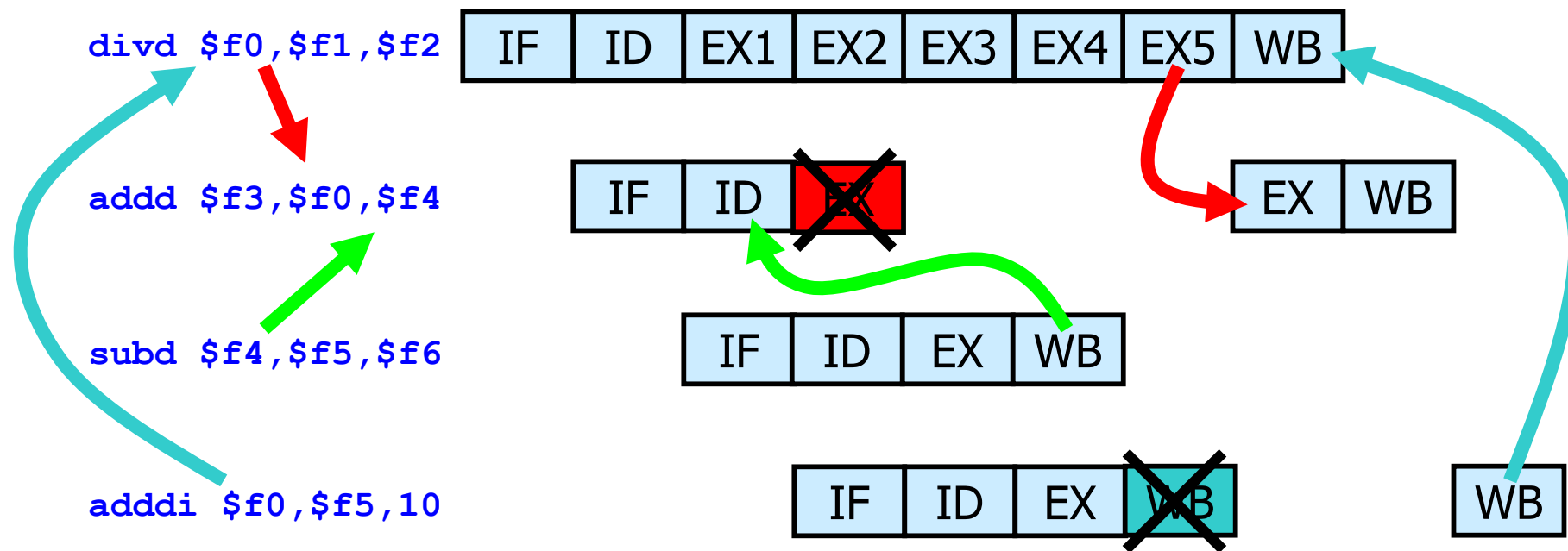
In-order Completion

- Simple pipelines have no WAR and WAW hazards by construction



Out-of-order Completion

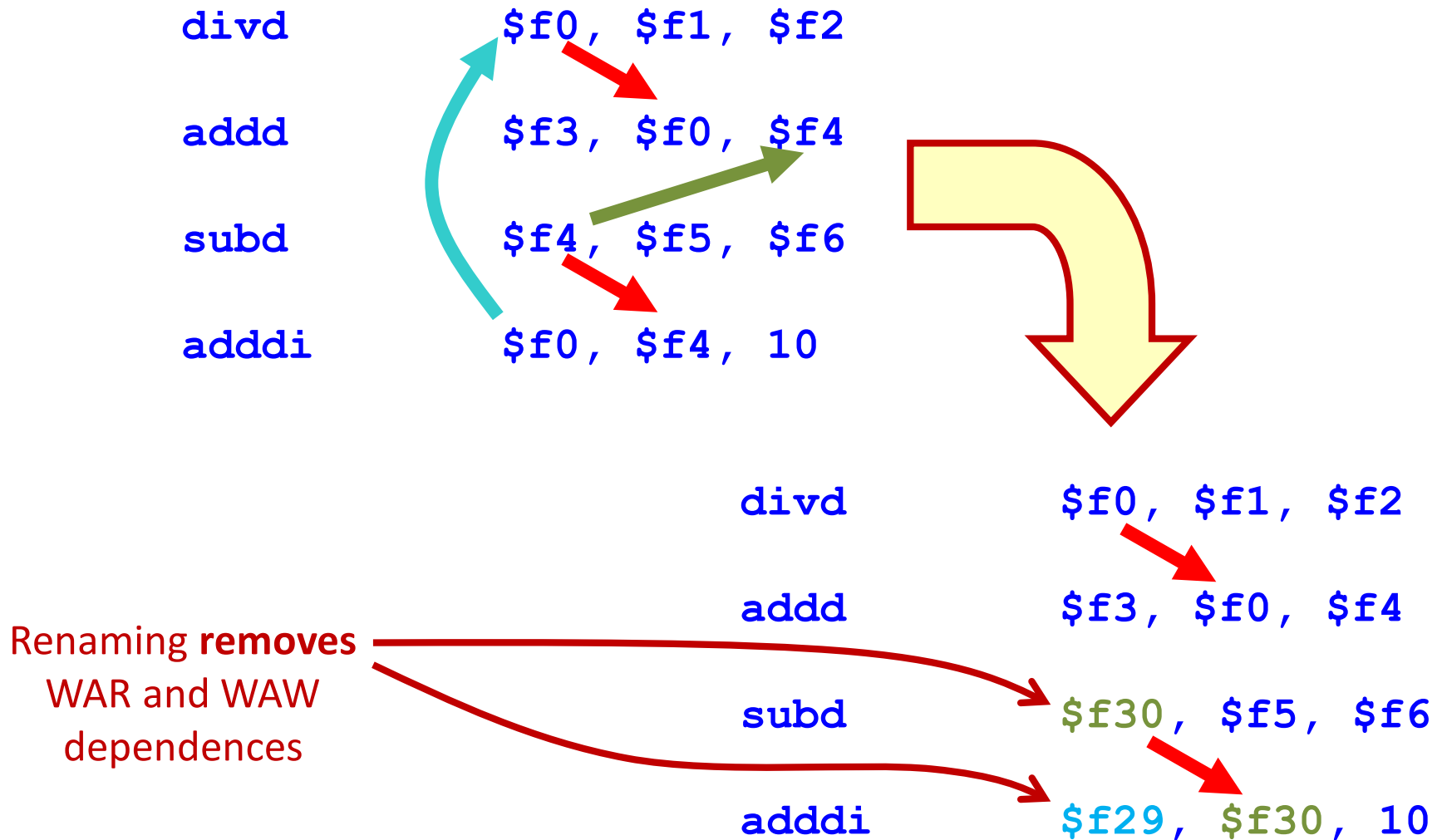
- Dynamic pipelines may create WAW hazards



Register Renaming

- WAW and WAR dependences are also called **name dependences**: they do not carry a value between 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

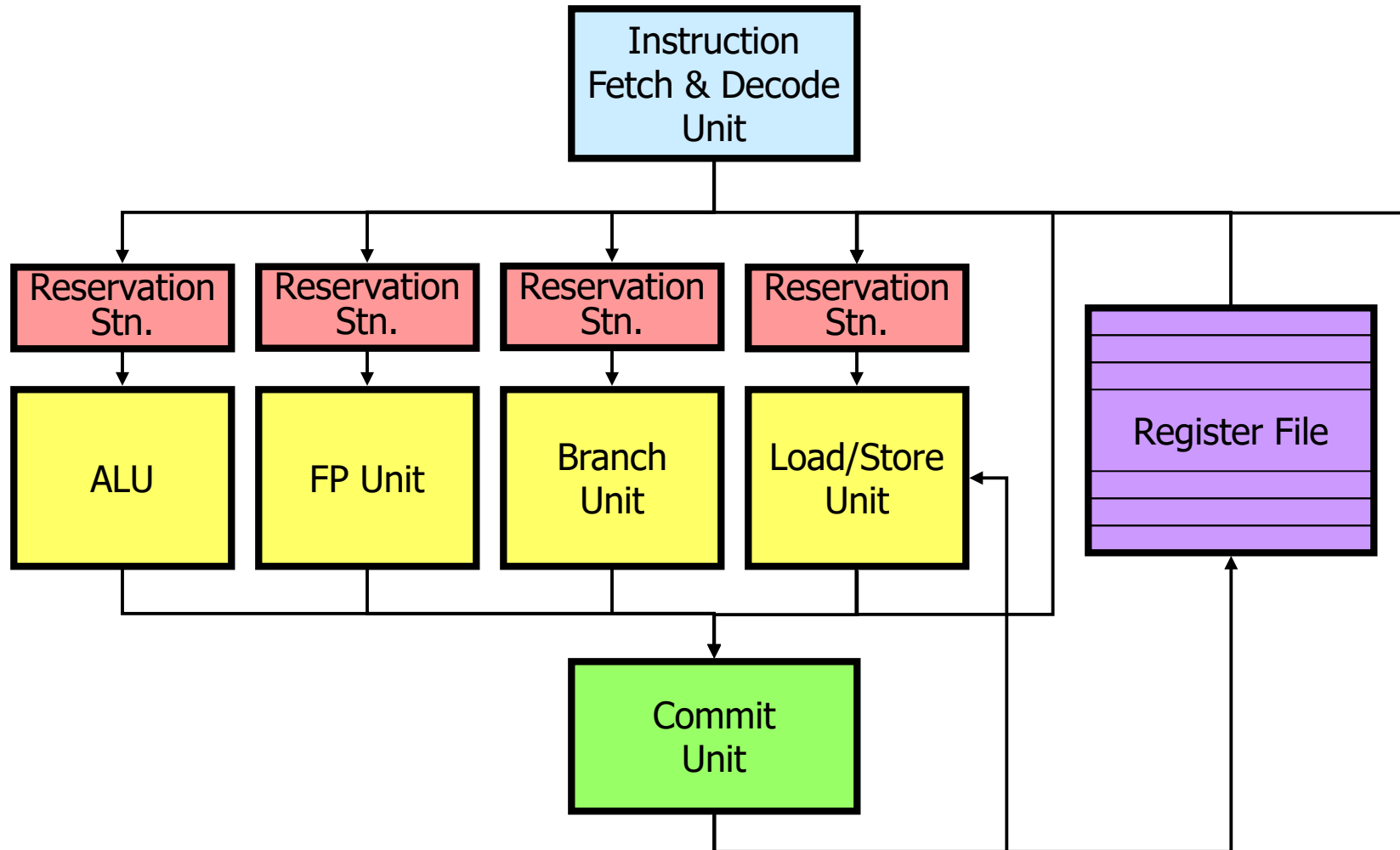
Register Renaming



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, to some extent, 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)
- Of course, one assumes that everything before the faulty instruction was executed and everything after was not
- With 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)

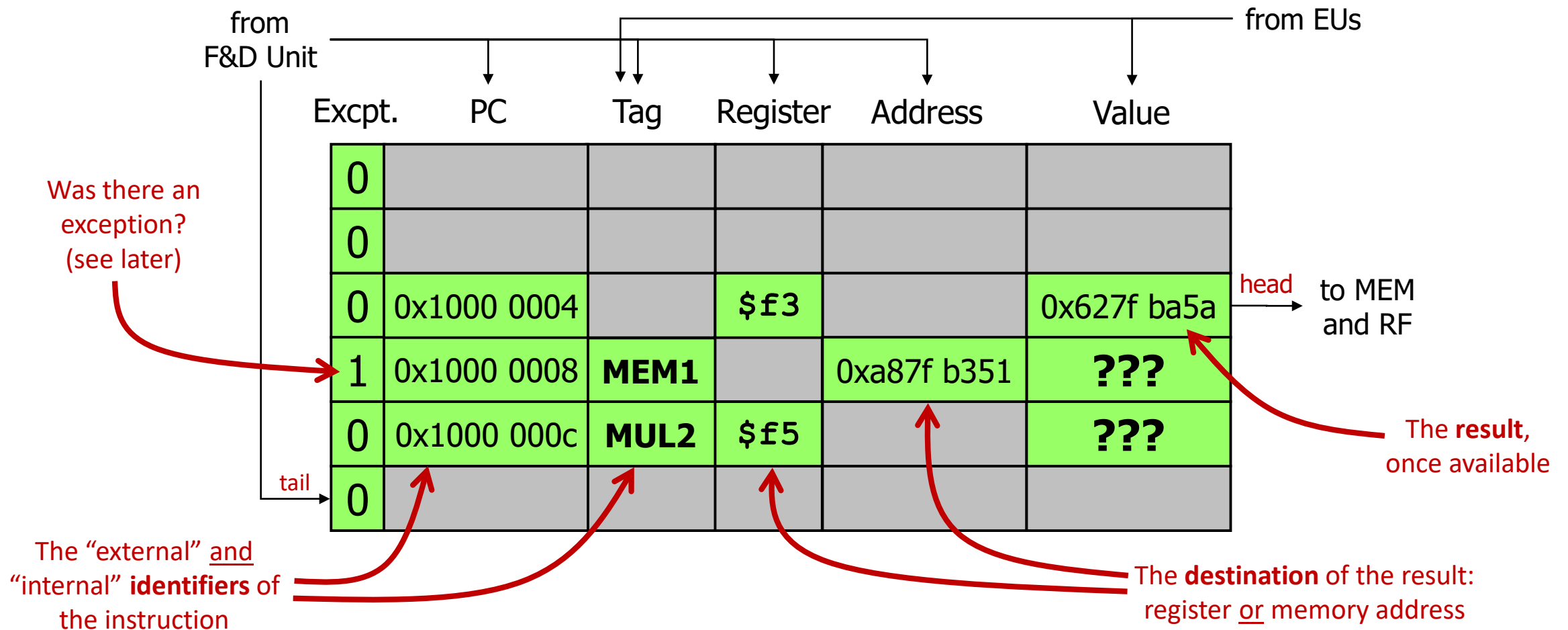
Generally **unacceptable** in contemporary systems (e.g., virtual memory, I/O interrupts, unsupported instructions)

	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

Reordering Instructions at Writeback

- Needs a reorder buffer in the Commit Unit



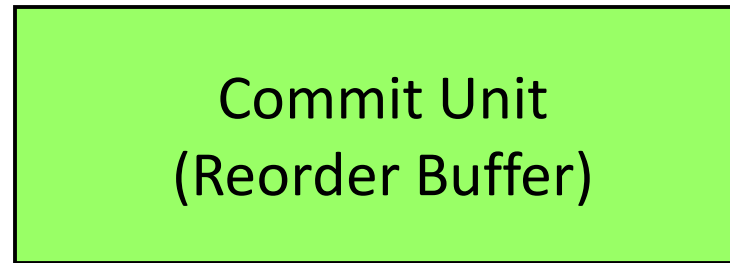
Reorder Buffer

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

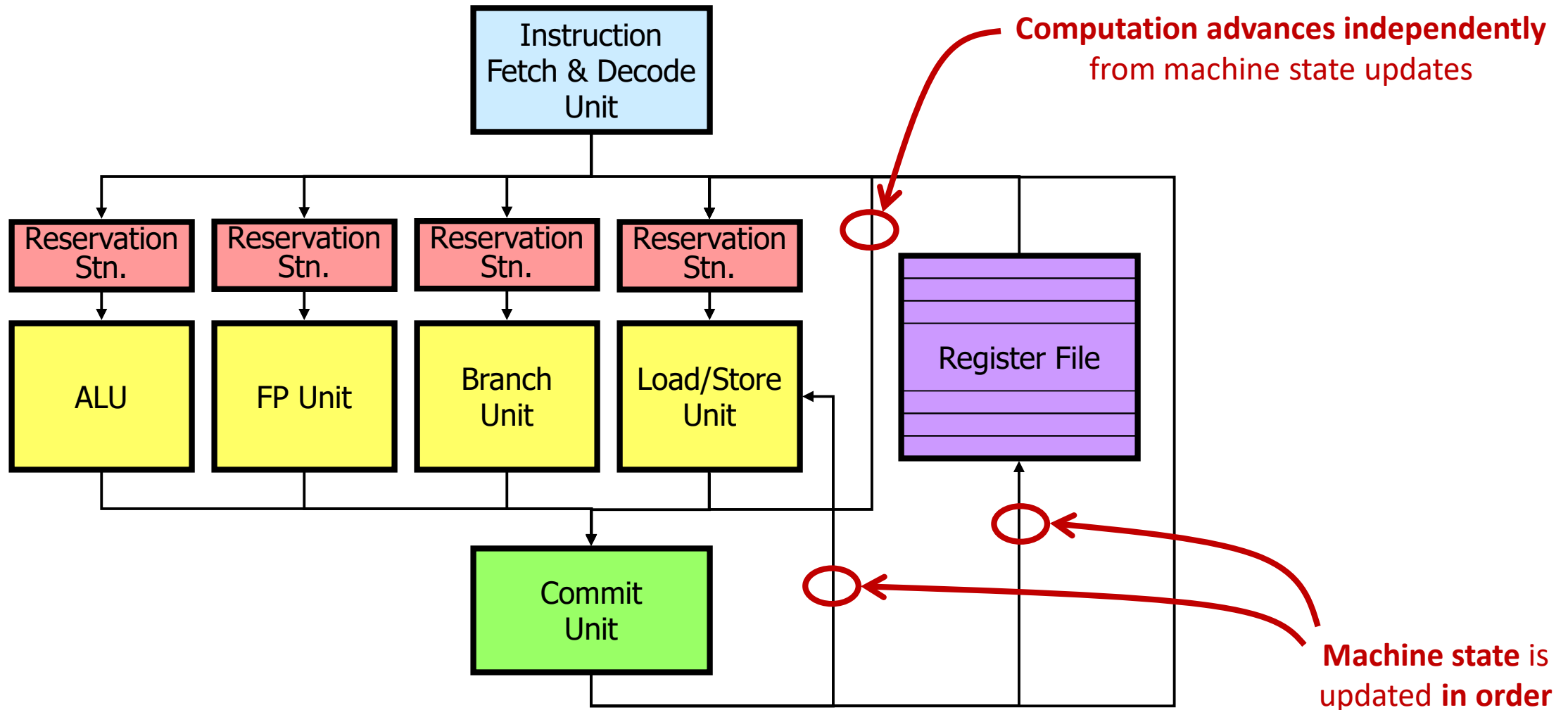


Commit Unit
(Reorder Buffer)

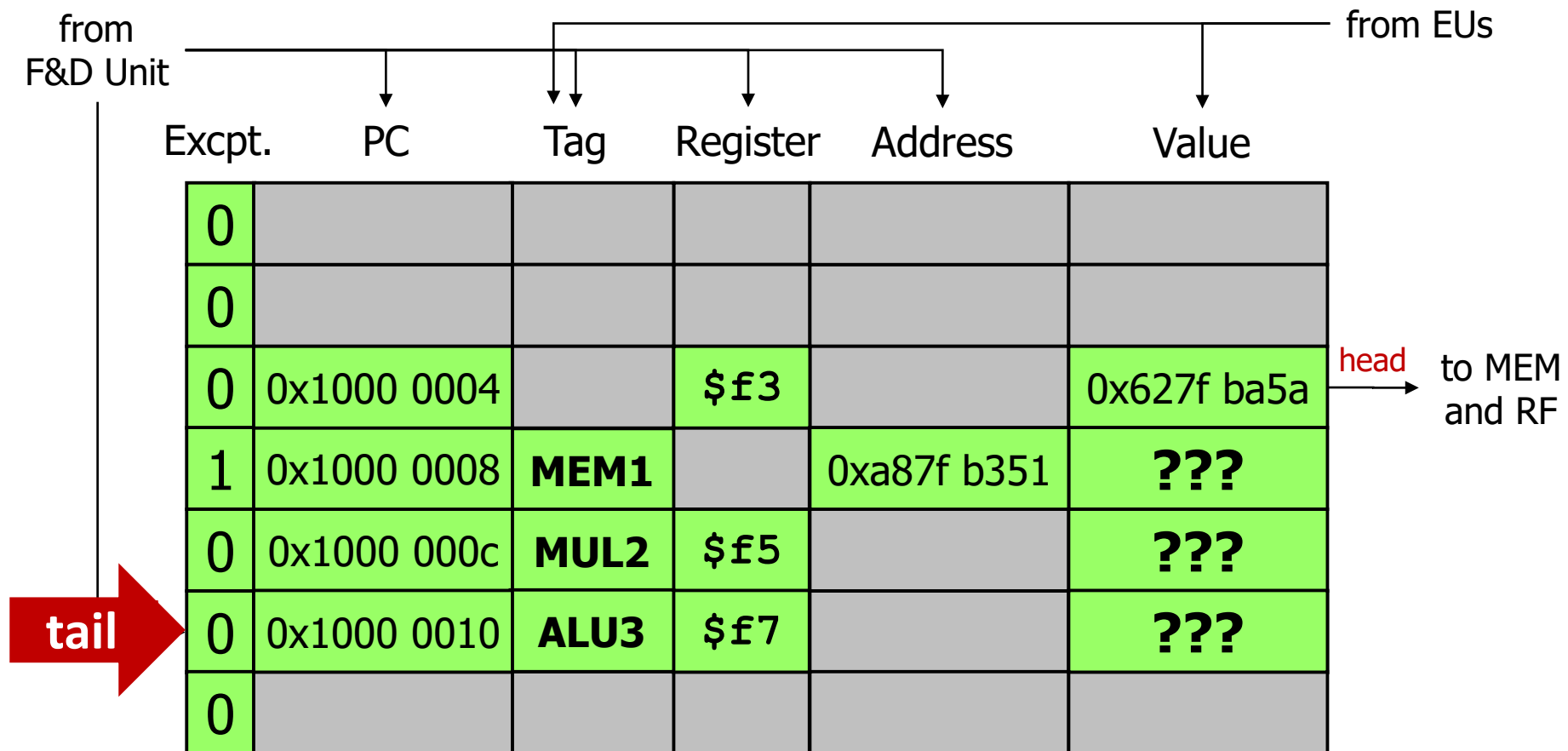
Register File and Memory

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

Dynamically Scheduled Processor



Fetching and Decoding Instructions



Committing Instructions (1/4)

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0	0x1000 0004		\$f3		0x627f ba5a
0	0x1000 0008	MEM1		0xa87f b351	???
0	0x1000 000c	MUL2	\$f5		???
0	0x1000 0010		\$f3		0xa2cd 374f
0	0x1000 0014	MEM3		0x3746 09fa	???
0					

Diagram illustrating the state of a commit table during instruction committing. The table has columns: Excpt., PC, Tag, Register, Address, and Value. The 'head' pointer points to the entry with PC 0x1000 0004 (Register \$f3, Value 0x627f ba5a). The 'tail' pointer points to the entry with PC 0x1000 0014 (Tag MEM3, Address 0x3746 09fa).

Write **0x627fba5a** to register **\$f3**

Committing Instructions (2/4)

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0					
0	0x1000 0008	MEM1		0xa87f b351	???
0	0x1000 000c	MUL2	\$f5		???
0	0x1000 0010		\$f3		0xa2cd 374f
0	0x1000 0014	MEM3		0x3746 09fa	???
0					

Diagram illustrating the state of instructions in a commit buffer. The buffer is a table with columns: Excpt., PC, Tag, Register, Address, and Value. The instructions are ordered by their PC (Program Counter) values. The oldest instruction (PC 0x1000 0008) is highlighted in green and has its result (???) stored in the Value field. A red arrow labeled "head" points to this instruction. The newest instruction (PC 0x1000 0014) is also highlighted in green and has its result (???) stored in the Value field. A red arrow labeled "tail" points to the instruction with PC 0x1000 0014.

Wait until the oldest instruction has its result

Committing Instructions (3/4)

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0					
0	0x1000 0008			0xa87f b351	0x98cd 76a2
0	0x1000 000c		\$f5		0x7677 abcd
0	0x1000 0010		\$f3		0xa2cd 374f
0	0x1000 0014	MEM3		0x3746 09fa	???
0					

Diagram illustrating the state of a memory buffer during instruction committing. The buffer is a table with 8 rows and 6 columns: Excpt., PC, Tag, Register, Address, and Value. The first four rows are empty. The fifth row (index 4) has PC 0x1000 0008, Address 0xa87f b351, and Value 0x98cd 76a2. The sixth row (index 5) has PC 0x1000 000c, Register \$f5, and Value 0x7677 abcd. The seventh row (index 6) has PC 0x1000 0010, Register \$f3, and Value 0xa2cd 374f. The eighth row (index 7) has PC 0x1000 0014, Tag MEM3, Address 0x3746 09fa, and Value ???.

A red arrow labeled "head" points to the Value field of the fifth row (0x98cd 76a2). A red arrow labeled "tail" points to the Excpt. field of the eighth row (0).

Write **0x98cd76a2** to memory location **0xa87fb351**

Committing Instructions (4/4)

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0					
0					
0	0x1000 000c		\$f5		0x7677 abcd
0	0x1000 0010		\$f3		0xa2cd 374f
0	0x1000 0014	MEM3		0x3746 09fa	???
0					

Diagram illustrating the state of a commit table during instruction committing. The table has columns: Excpt., PC, Tag, Register, Address, and Value. The current instruction being committed is highlighted in green (row 6). The 'head' pointer points to the current instruction, and the 'tail' pointer points to the next free slot (row 8).

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

32

32

32

Reporting Exceptions (2/3)

Excpt.	PC	Tag	Register	Address	Value
0					
0					
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	???
0					

Diagram illustrating the state of an exception table. The table has columns: Excpt., PC, Tag, Register, Address, and Value. The rows are indexed 0 to 7. The 'head' pointer points to the entry at index 3 (PC: 0x1000 0004, Register: \$f3, Value: 0x627f ba5a). The 'tail' pointer points to the entry at index 7 (Excpt.: 0).

Write **0x627fba5a** to register **\$f3** as if nothing happened

Reporting Exceptions (3/3)

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0					
1	0x1000 0008	MEM1		0xa87f b351	???
0	0x1000 000c		\$f5		0x7677 abcd
0	0x1000 0010		\$f3		0xa2cd 371f
0	0x1000 0014	MEM3		0x3716 09fa	???
tail → 0					

Exception PC →

head →

Now raise the *TLB Miss* exception in the instruction at location **0x10000008**

Problems to Solve

- **Structural Hazards**

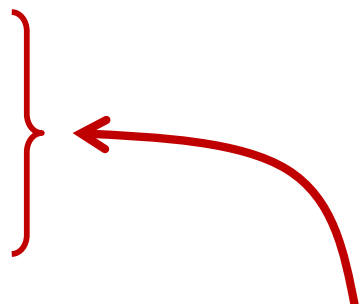
- Are the required resources available?
- New problem: previously handled by rigid pipeline

- **RAW Data Hazards**

- Are the operands ready to start execution?
- Old problem

- **WAR and WAW Data Hazards**

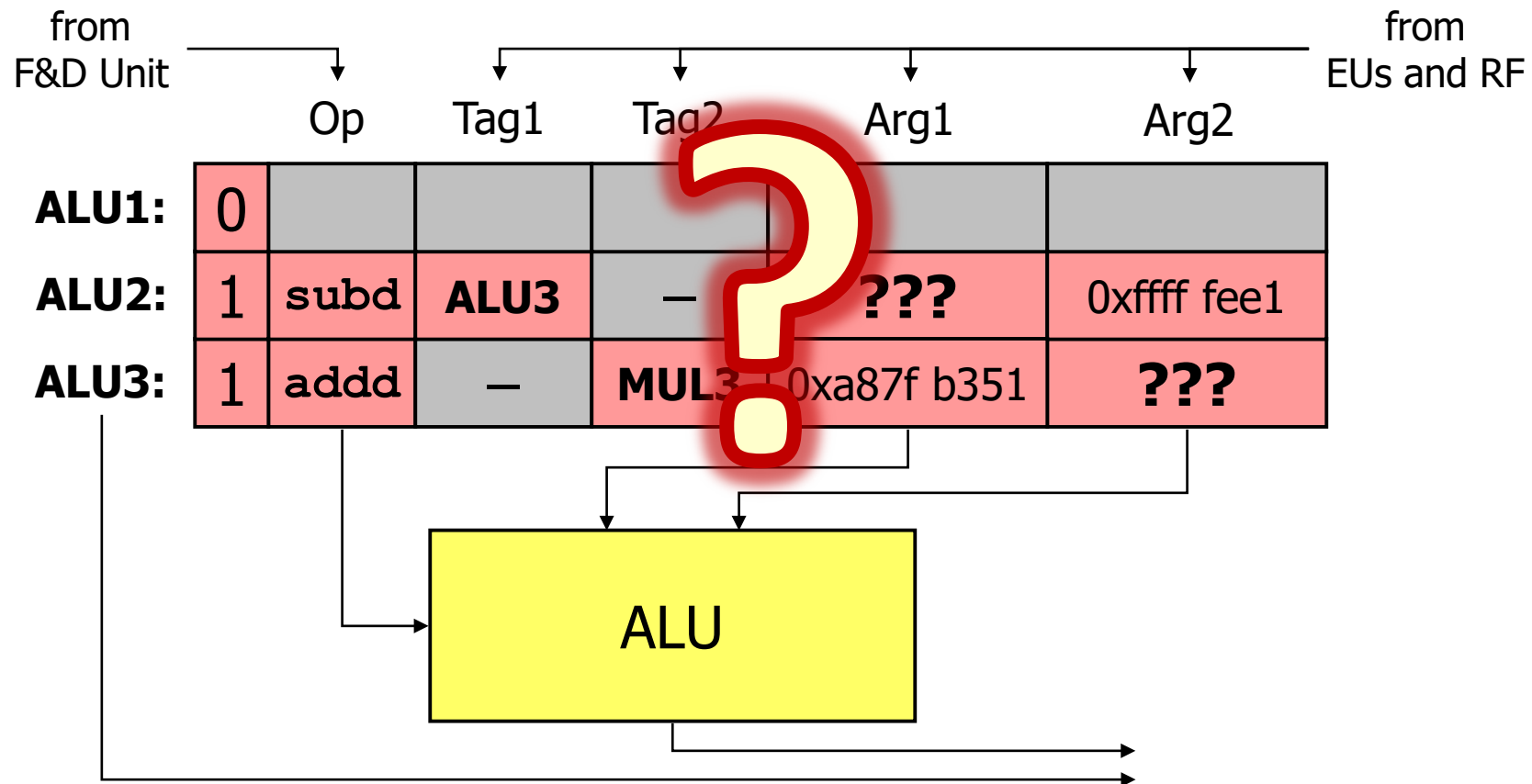
- The new data overwrite something which is still required?
- WAW is a completely new problem—impossible before; WAR often cannot occur



Reservation stations take care of
contention for execution units
The **commit unit** writes back one
instruction at a time

Reservation Stations

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



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 (ROB)** 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 already computed → Get the **value** from the **ROB**
 - If the value is not yet computed → Get the **tag** from the **ROB**

No Dependence

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0	0x1000 0004		\$f2		0x627f ba5a
0	0x1000 0008		\$f3		0xa87f b351
0	0x1000 000c	MUL2	\$f5		???
0	0x1000 0010	ALU3	\$f3		???
0	0x1000 0014	MEM3		0x3746 09fa	???
0					

Diagram illustrating a memory access scenario. A vertical red arrow points upwards from the bottom, indicating a search for register \$f1. The table shows instructions with their PC, Tag, Register, Address, and Value. The 'head' points to the top of the table, and the 'tail' points to the bottom. The current instruction being executed is the one with PC 0x1000 0014 (MEM3), which uses register \$f3. The value of \$f3 is 0xa87f b351. The instruction at PC 0x1000 000c (MUL2) uses register \$f5, which has a value of ???.

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

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0	0x1000 0004		\$f2		0x627f ba5a
0	0x1000 0008		\$f3		0xa87f b351
0	0x1000 000c	MUL2	\$f5		???
0	0x1000 0010	ALU3	\$f3		???
0	0x1000 0014	MEM3		0x3746 09fa	???
0					

Diagram illustrating the ROB (Reorder Buffer) structure. The table shows entries with columns: Excpt., PC, Tag, Register, Address, and Value. A red arrow points to the Register column, indicating a search for \$f2. The 'head' pointer is at the entry with PC 0x1000 0004 and Register \$f2. The 'tail' pointer is at the entry with PC 0x1000 0014.

Looking for **\$f2**?

An ongoing instruction has produced it, hence we should read **0x627fba5a** from the ROB

Dependence and Tag in the ROB

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0	0x1000 0004		\$f2		0x627f ba5a
0	0x1000 0008		\$f3		0xa87f b351
0	0x1000 000c	MUL2	\$f5		???
0	0x1000 0010	ALU3	\$f3		???
0	0x1000 0014	MEM3		0x3746 09fa	???
0					

Diagram illustrating the Register Order Buffer (ROB) structure. The table shows entries with Exception (Excpt.), Program Counter (PC), Tag, Register, Address, and Value. A red arrow points to the Register column, indicating a search for \$f5. The 'head' pointer is at the entry with PC 0x1000 0004, and the 'tail' pointer is at the entry with PC 0x1000 0014.

Looking for **\$f5**?

An ongoing instruction will produce it, hence we need to use tag **MUL2** as found in the ROB

Multiple Dependences?

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0	0x1000 0004		\$f2		0x627f ba5a
0	0x1000 0008		\$f3		0xa87f b351
0	0x1000 000c	MUL2	\$f5		???
0	0x1000 0010	ALU3	\$f3		???
0	0x1000 0014	MEM3		0x3746 09fa	???
0					

Diagram illustrating a memory structure (likely a cache or buffer) with columns: Excpt., PC, Tag, Register, Address, and Value. The structure is divided into two sections by a vertical red line. The left section (labeled 'tail' with an arrow) contains entries 0 through 7. The right section (labeled 'head' with an arrow) contains entries 0 through 7. A red arrow points upwards from the bottom of the 'tail' section to the 'Register' column of the entry at index 3 (PC: 0x1000 0008, Tag: \$f3, Value: 0xa87f b351). Another red arrow points from the 'Register' column of the entry at index 4 (PC: 0x1000 000c, Tag: MUL2, Value: ???) to the 'Register' column of the entry at index 5 (PC: 0x1000 0010, Tag: ALU3, Value: ???).

Looking for **\$f3**?

Two ongoing instructions produce it and it is the **most recent** one which matters (**ALU3**)

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:

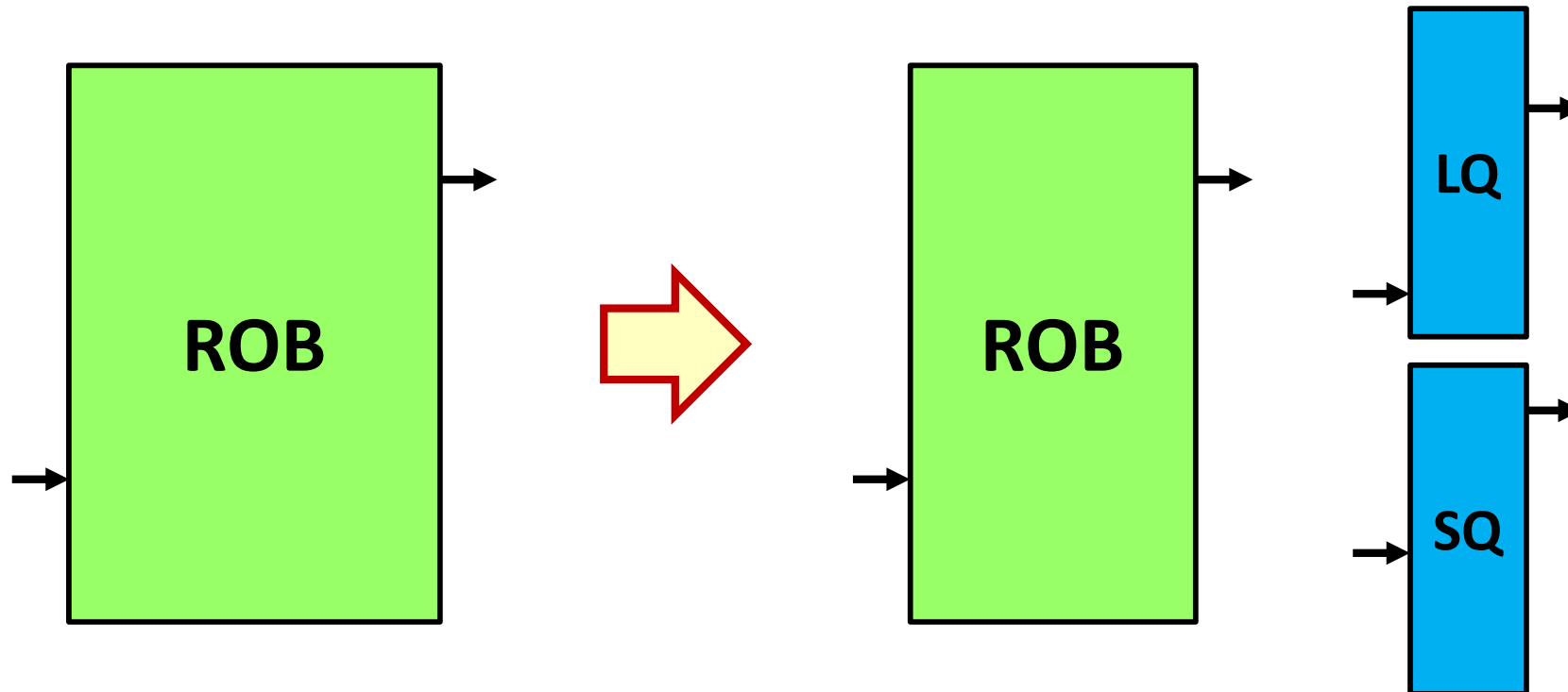
- a) If there is **no store to the same address** in the ROB, get the value from memory (i.e., from the cache)
- b) If there is a **store to the same address** in the ROB, either get the value (if ready) or the tag

but there is an additional situation now

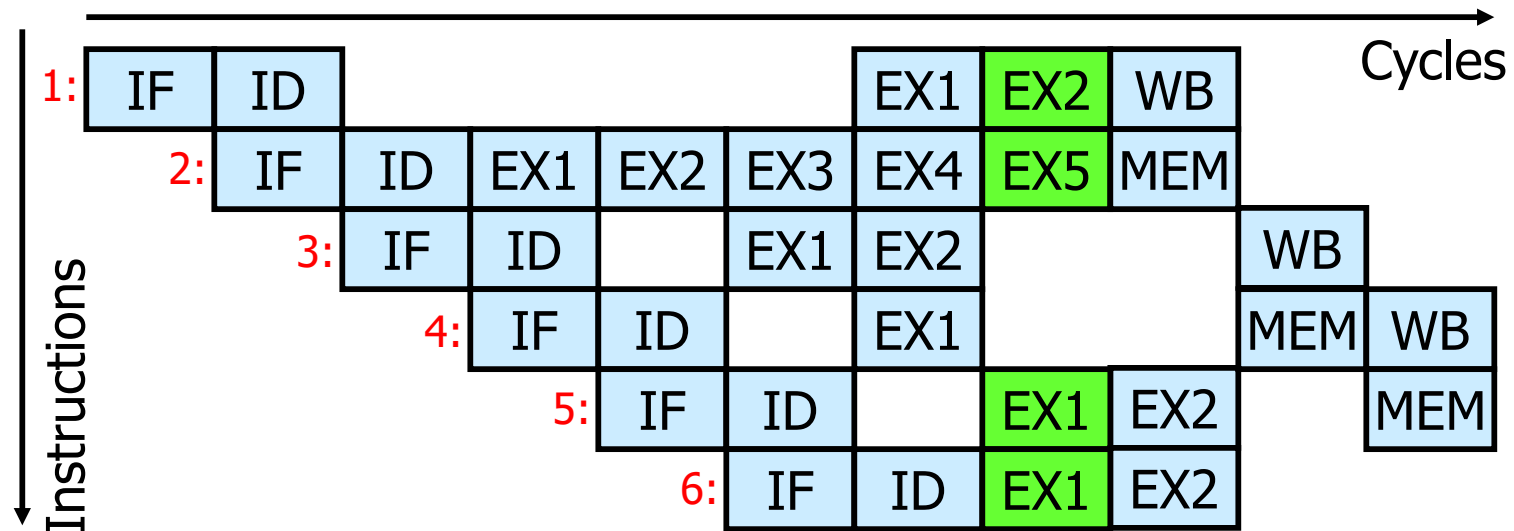
- c) If there is a store to an **unknown address** in the ROB or if the address of the load is unknown, **wait!**

Load-Store Queues

In practice, the **memory part of the ROB** is implemented separately and is called a **Load-Store Queue** (in turn, usually implemented as a Load and a Store queues)

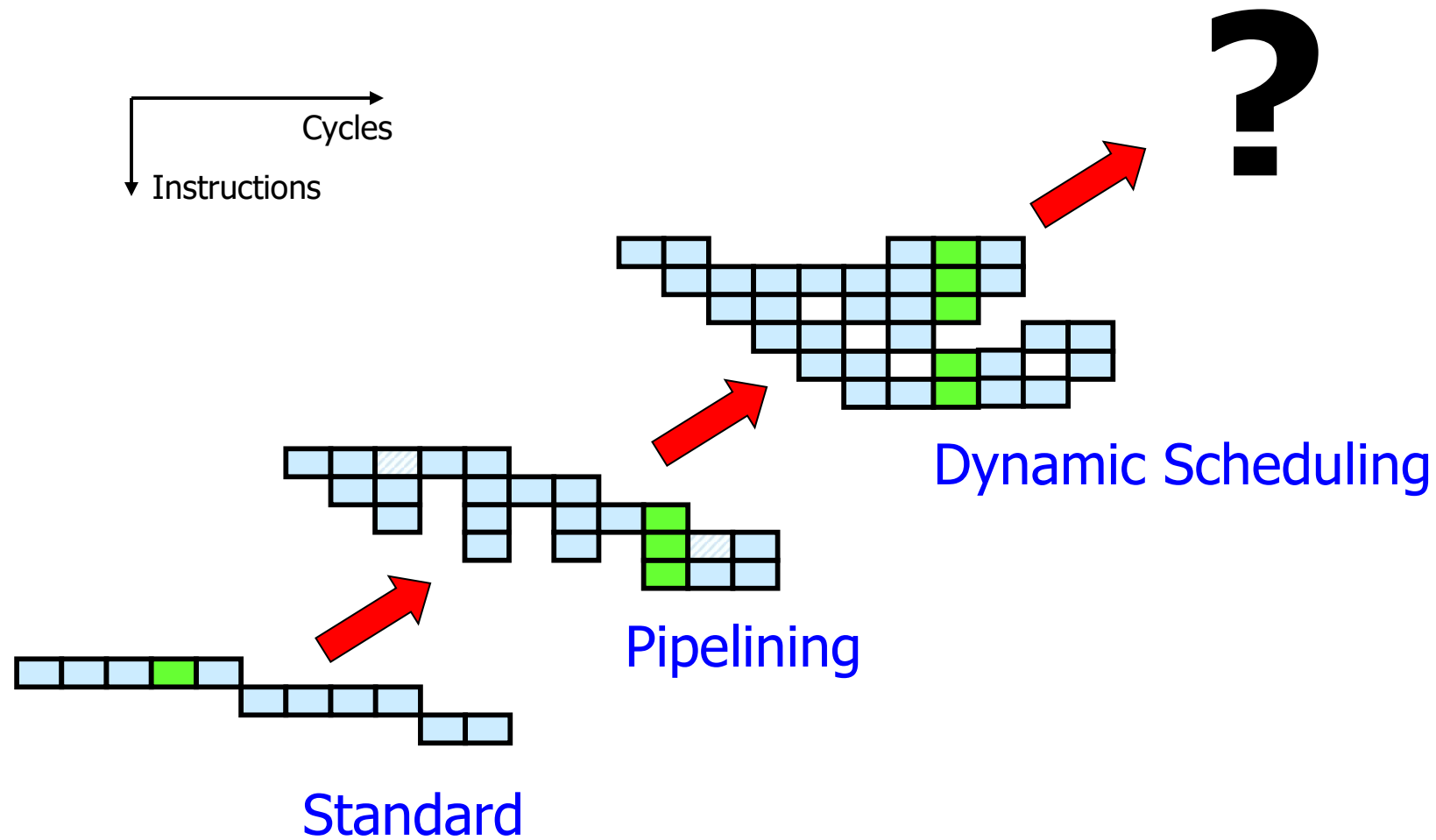


Second Step: Dynamic Scheduling



- The ability to reorder instructions **unlocks a tangible amount of ILP**

Summary: ILP So Far...



References

- Patterson & Hennessy, COD – RISC-V Edition
 - **Section 4.11** (in particular “Dynamic Multiple-Issue Processors”)