



POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

Advanced Computer Architectures

Introduction to Instruction-Level Parallelism

A.Y. 2024/2025 | Christian Pilato (christian.pilato@polimi.it)

Outline

- ILP Definition
- Complex Pipelining
- Intro to Superscalar Architectures, Static and Dynamic Schedulers

Definition of ILP

ILP = Potential overlap of execution among unrelated instructions

Definition of ILP

ILP = Potential overlap of execution among unrelated instructions

Overlapping possible if:

- No Structural Hazards
- No RAW, WAR, or WAW Stalls
- No Control Stalls

Review: Pipeline performance

Pipeline CPI = Ideal pipeline CPI

Ideal pipeline CPI: measure of the maximum performance attainable by the implementation

Review: Pipeline performance

Pipeline CPI = **Ideal pipeline CPI** + **Structural Stalls**

Ideal pipeline CPI: measure of the maximum performance attainable by the implementation

Structural hazards: HW cannot support this combination of instructions

Review: Pipeline performance

Pipeline CPI = **Ideal pipeline CPI** + **Structural Stalls** + **Data Hazard Stalls**

Ideal pipeline CPI: measure of the maximum performance attainable by the implementation

Structural hazards: HW cannot support this combination of instructions

Data hazards: Instruction depends on the result of prior instruction still in the pipeline

Review: Pipeline performance

Pipeline CPI = **Ideal pipeline CPI** + **Structural Stalls** + **Data Hazard Stalls** + **Control Stalls**

Ideal pipeline CPI: measure of the maximum performance attainable by the implementation

Structural hazards: HW cannot support this combination of instructions

Data hazards: Instruction depends on the result of prior instruction still in the pipeline

Control hazards: Caused by the delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)

Review: Summary of Pipelining Basics

Hazards limit performance

Structural: need more HW resources

Data: need forwarding, compiler scheduling

Control: early evaluation & PC, delayed branch, predictors


Increasing the length of the pipeline increases the impact of hazards

Pipelining helps instruction bandwidth, not latency

Review: Types of Data Hazards

Data-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$
 $r_5 \leftarrow (r_3) \text{ op } (r_4)$




Read-after-Write
(RAW) hazard

Review: Types of Data Hazards

Data-dependence


$r_3 \leftarrow (r_1) \text{ op } (r_2)$
 $r_5 \leftarrow (r_3) \text{ op } (r_4)$



Read-after-Write
(RAW) hazard

Anti-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$
 $r_1 \leftarrow (r_4) \text{ op } (r_5)$




Write-after-Read
(WAR) hazard

Review: Types of Data Hazards

Data-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$
 $r_5 \leftarrow (r_3) \text{ op } (r_4)$ Read-after-Write (RAW) hazard




Anti-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$
 $r_1 \leftarrow (r_4) \text{ op } (r_5)$ Write-after-Read (WAR) hazard



Output-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$
 $r_3 \leftarrow (r_6) \text{ op } (r_7)$ Write-after-Write (WAW) hazard

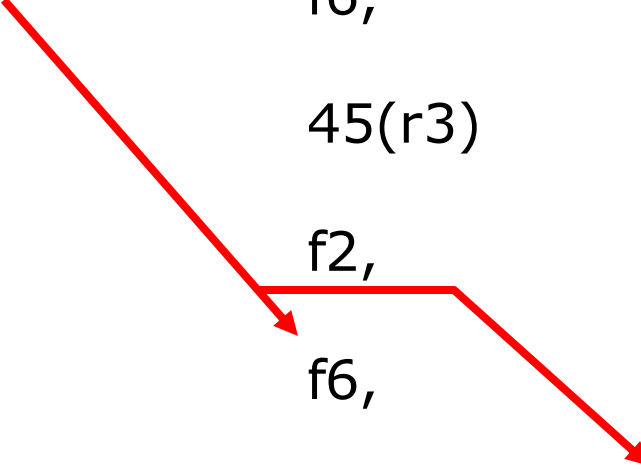


Data Hazards: An Example

		<i>dest</i>	<i>src1</i>	<i>src2</i>
I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

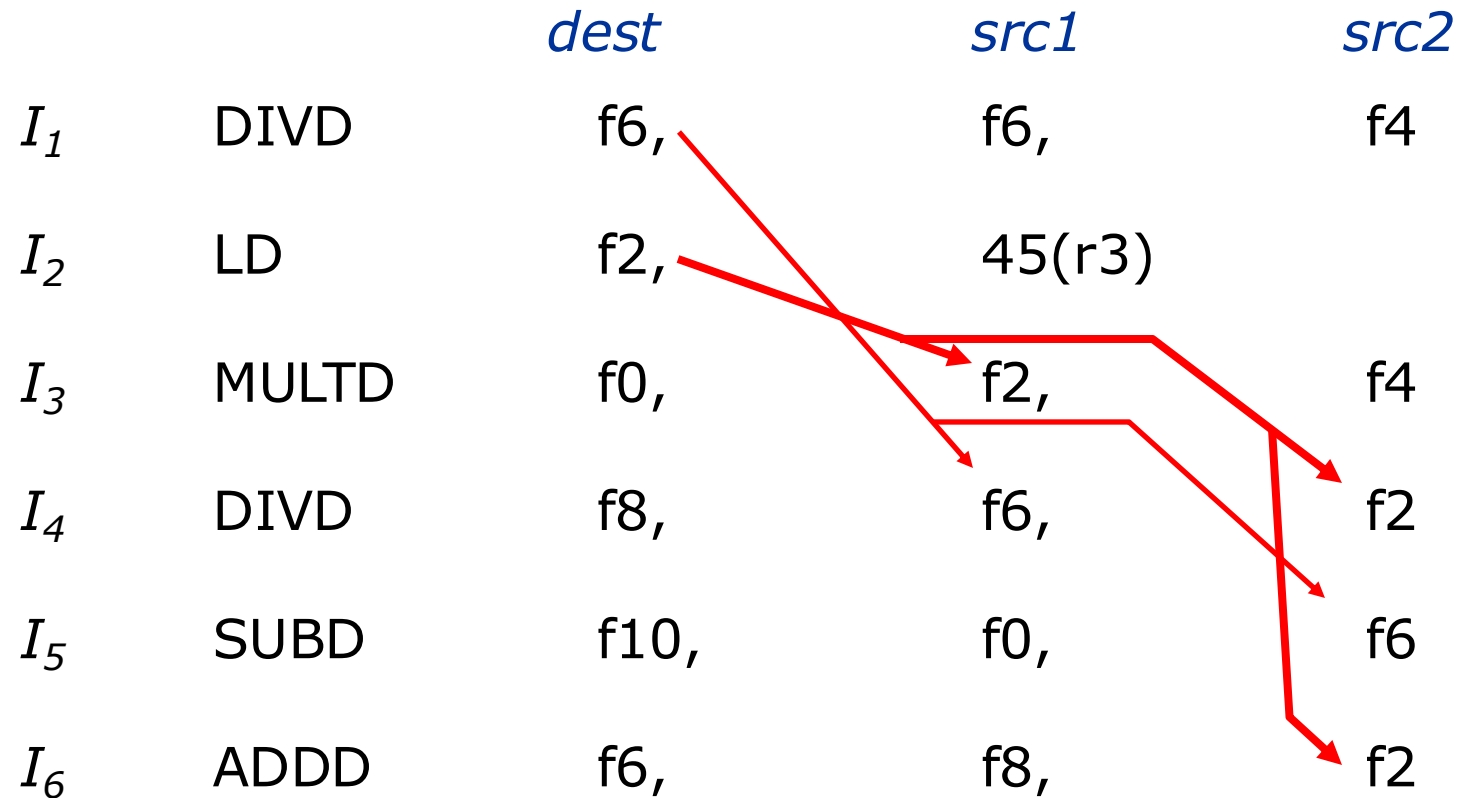
Data Hazards: An Example

		<i>dest</i>	<i>src1</i>	<i>src2</i>
I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2



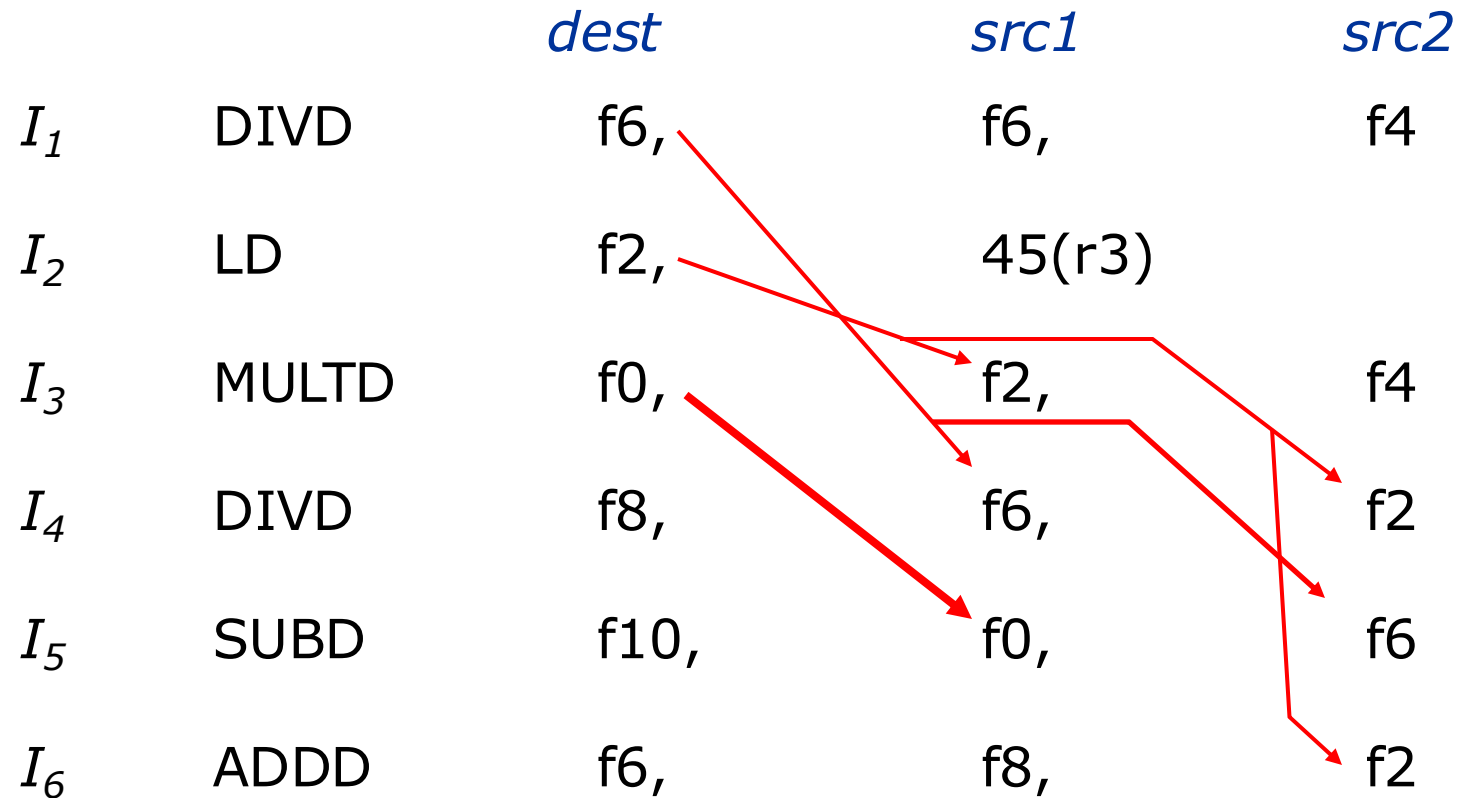
RAW Hazards

Data Hazards: An Example



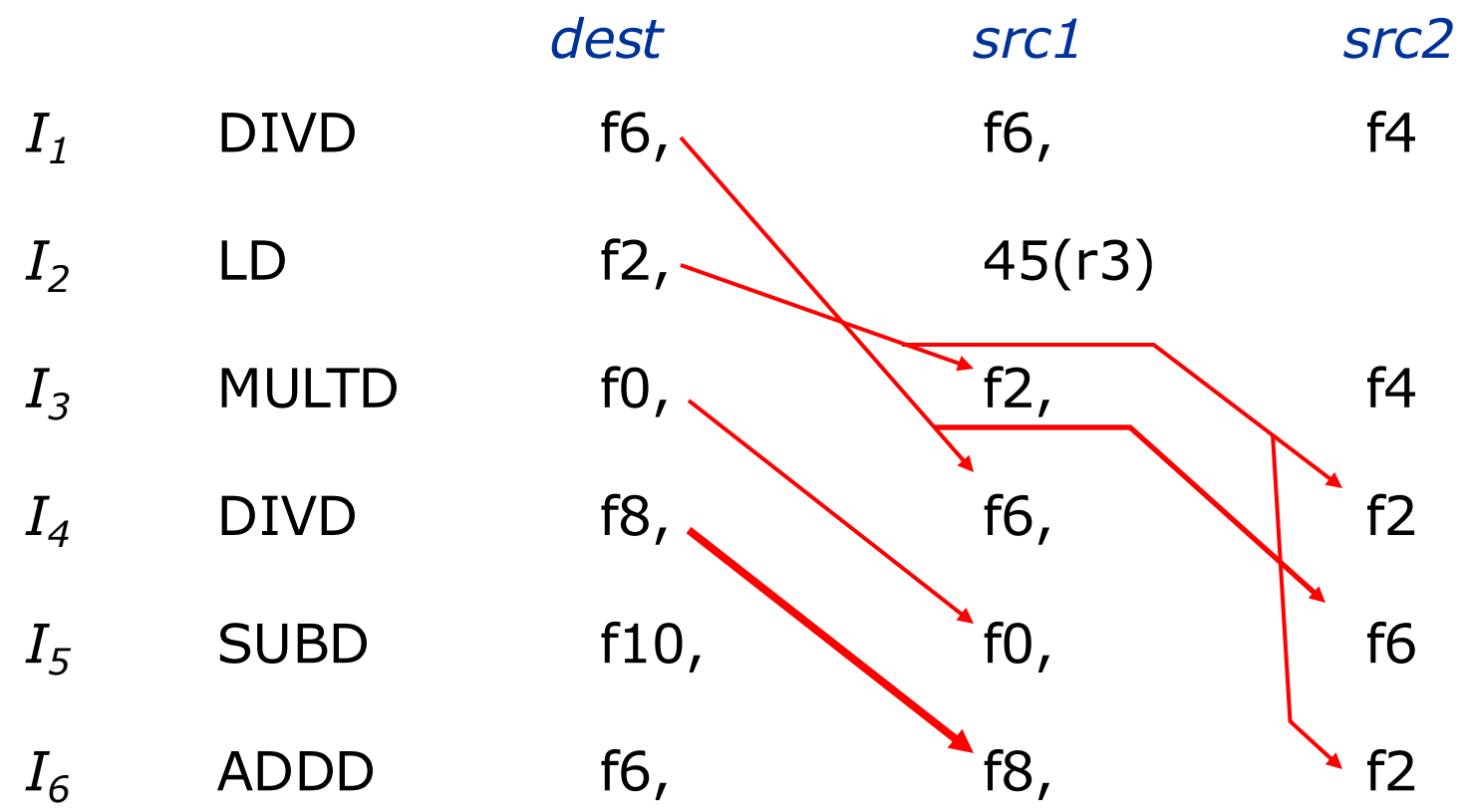
RAW Hazards

Data Hazards: An Example



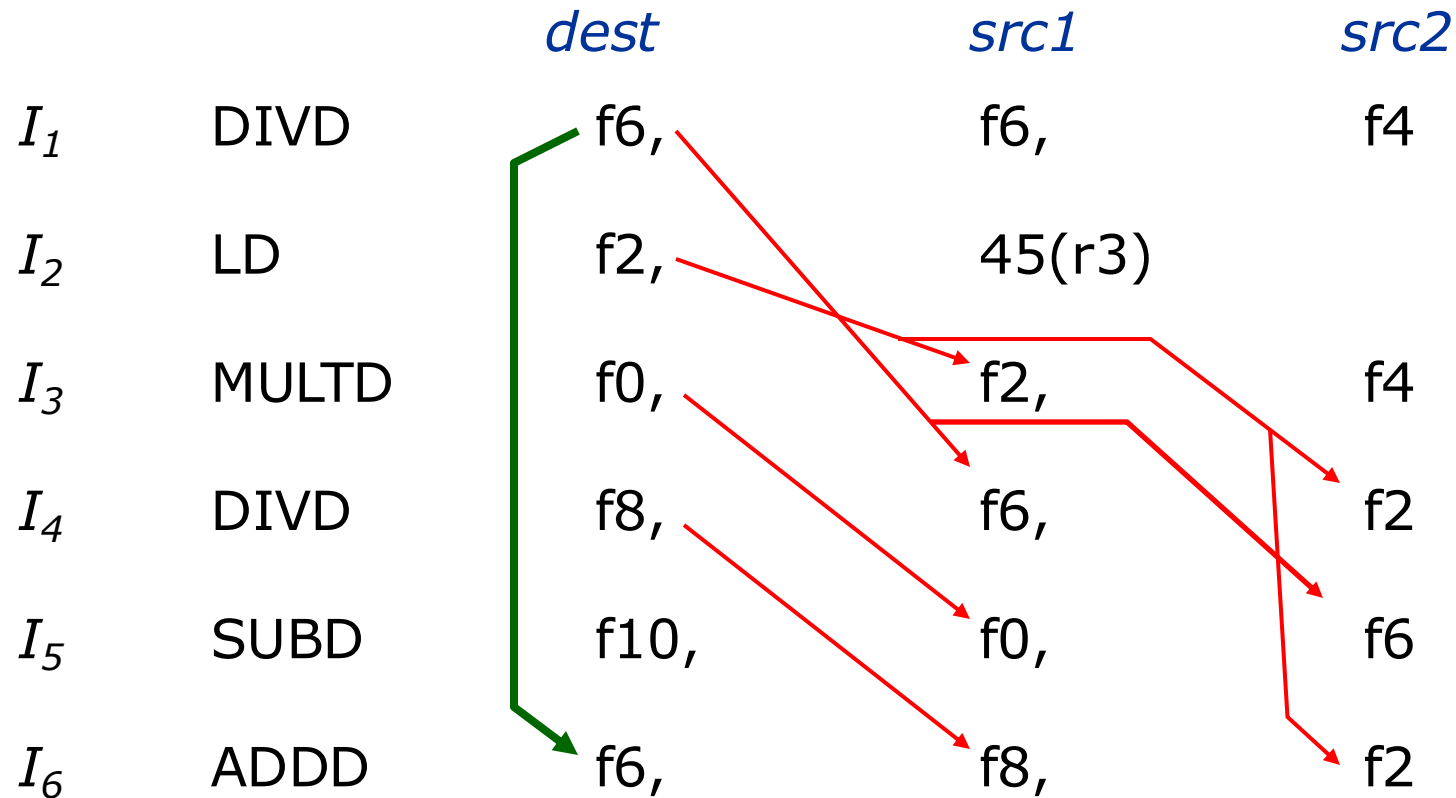
RAW Hazards

Data Hazards: An Example

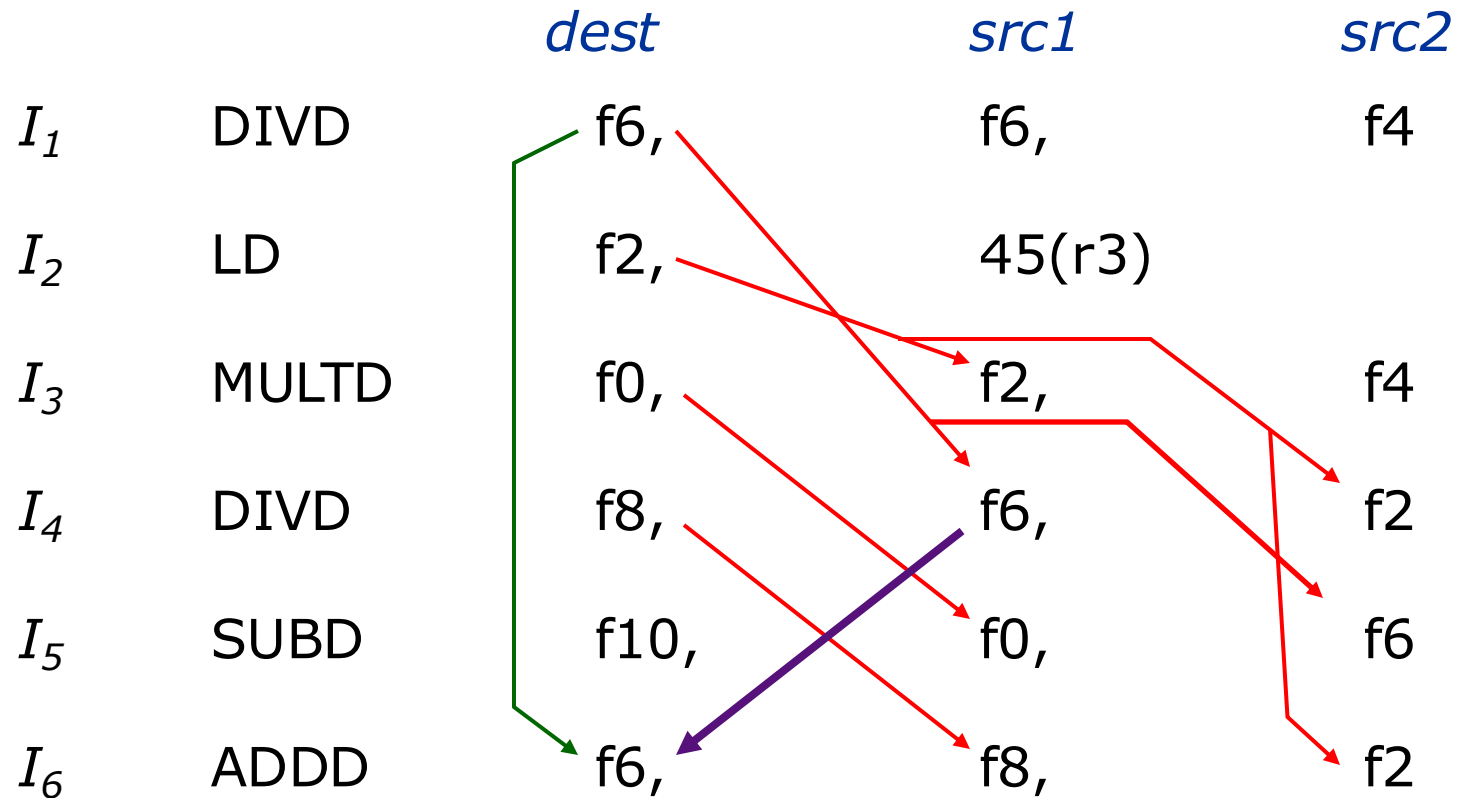


RAW Hazards

Data Hazards: An Example



Data Hazards: An Example

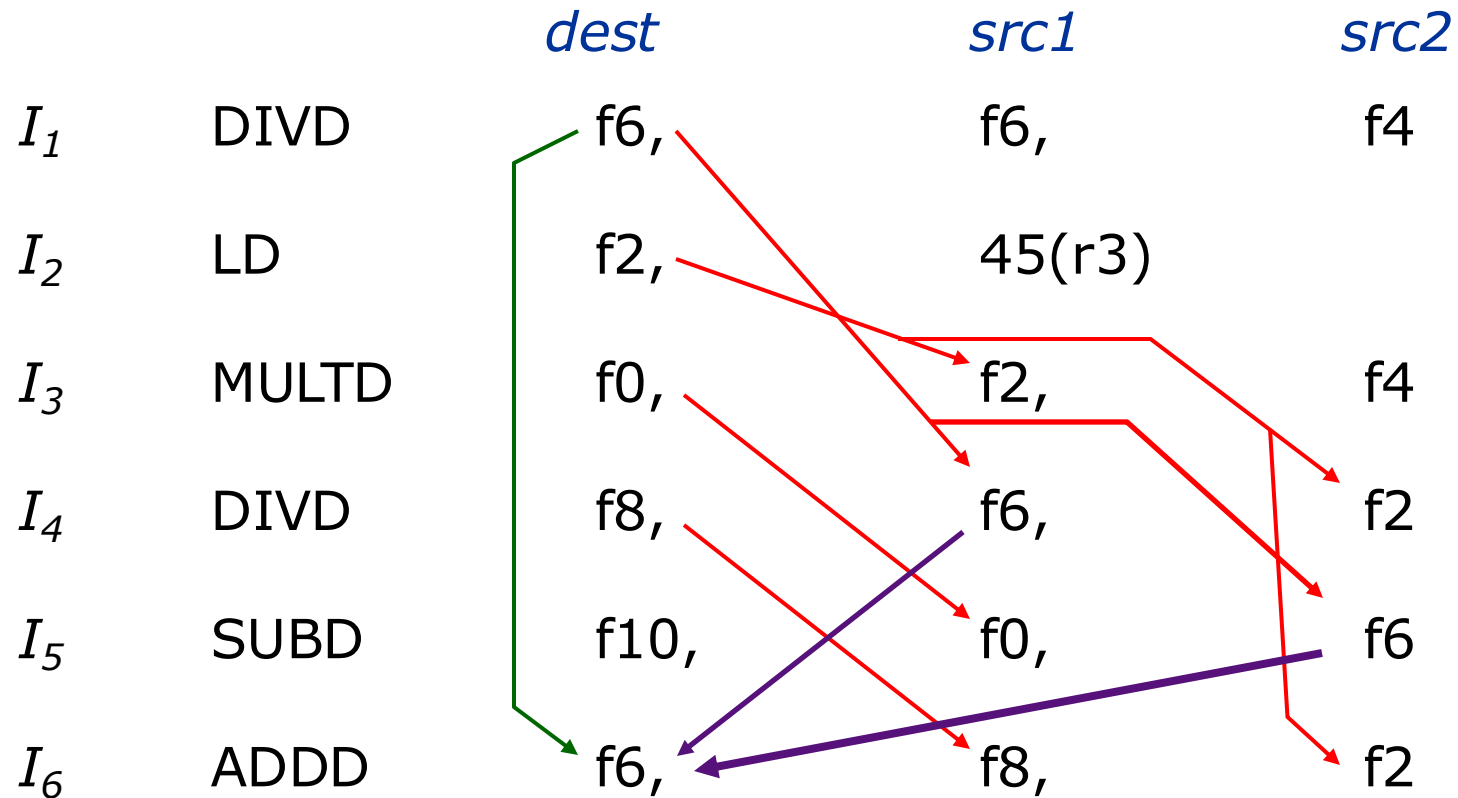


RAW Hazards

WAW Hazards

WAR Hazards

Data Hazards: An Example



RAW Hazards

WAW Hazards

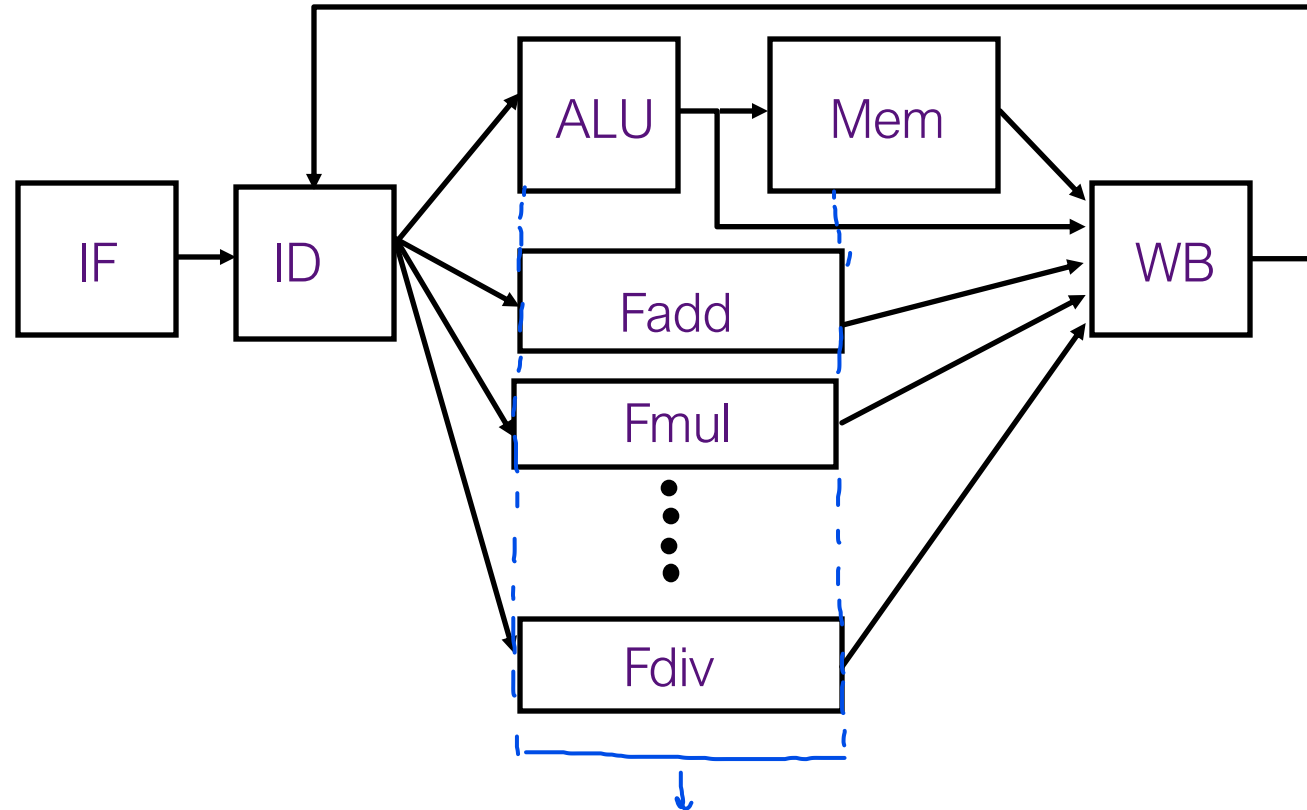
WAR Hazards

A new scenario

What about mixing **Integer** and **Floating-Point** operations?

A new scenario

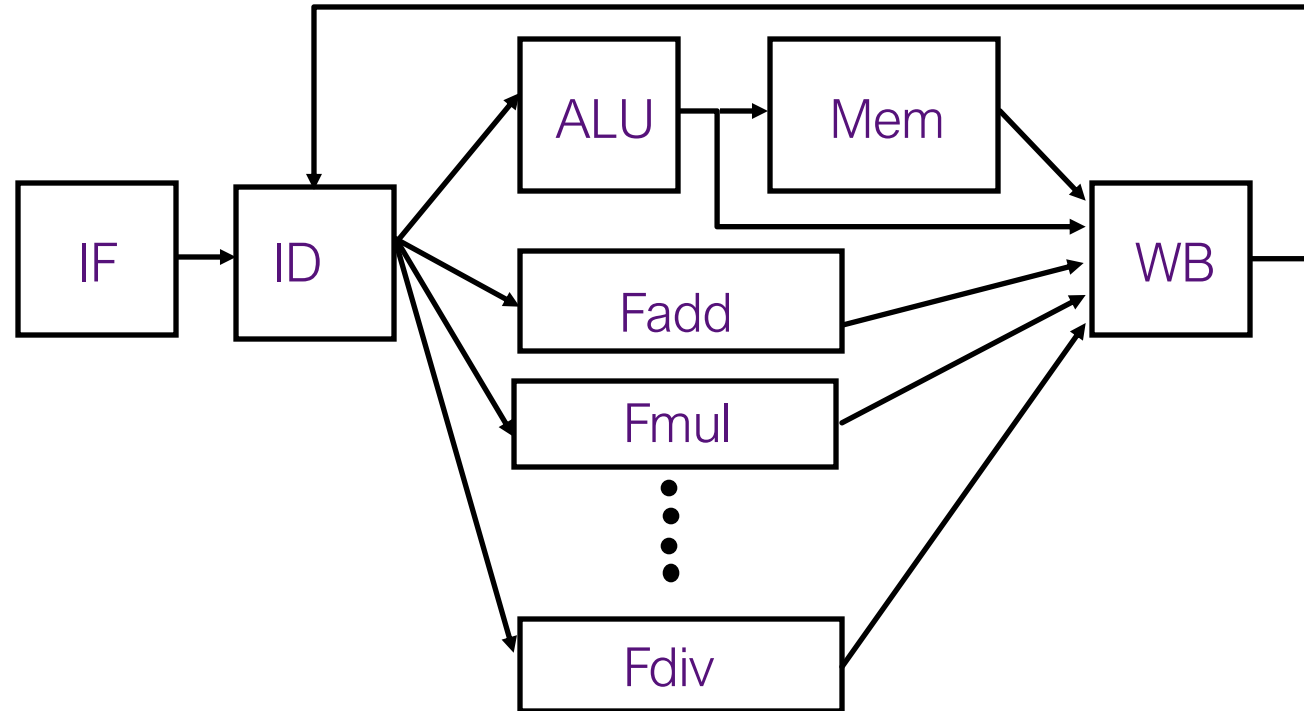
What about mixing **Integer** and **Floating-Point** operations?



we see operation on floating points (Foperation) take longer time wrt operations performed by ALU on fixed point values.

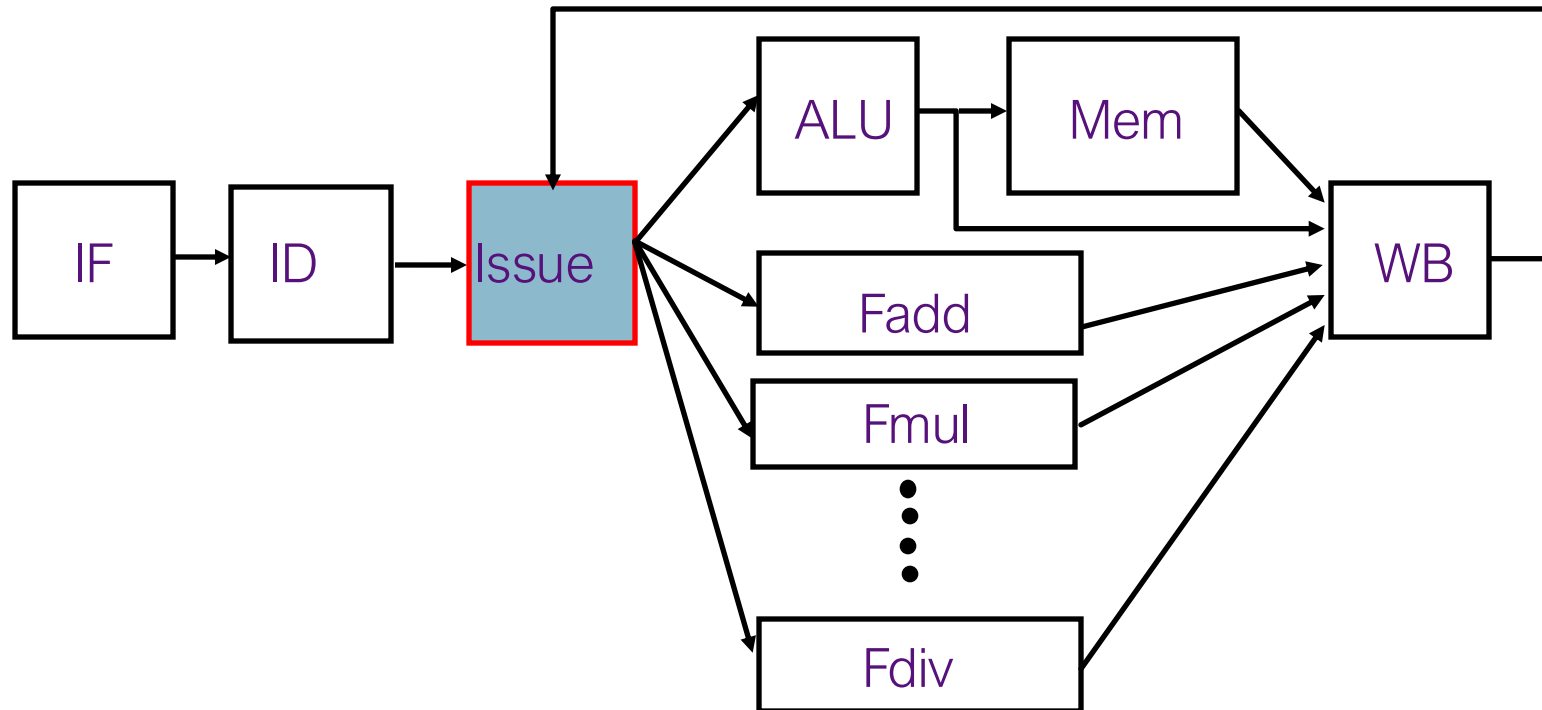
A new scenario

What about mixing **Integer** and **Floating-Point** operations?



How to handle different registers?

Complex Pipelining

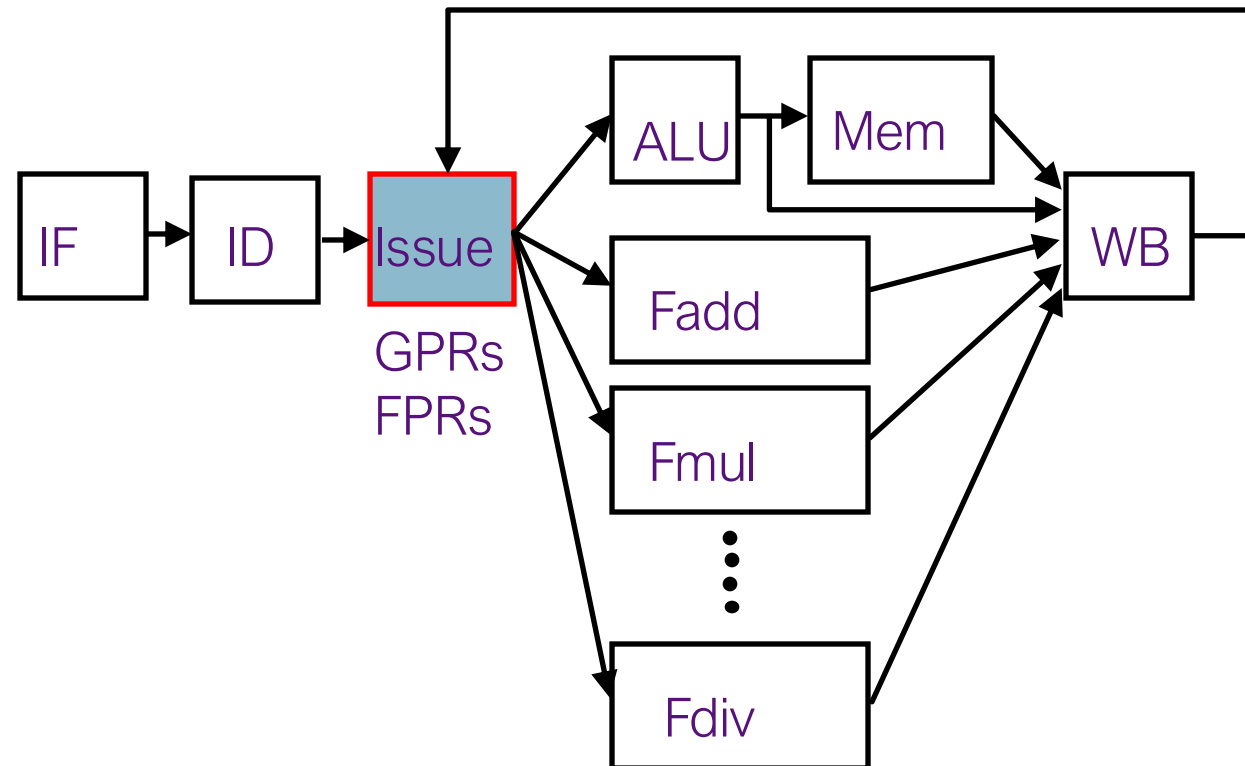


Pipelining becomes complex when we want high performance in the presence of:

- Long latency or partially pipelined floating-point units
- Multiple function and memory units
- Memory systems with variable access time
- Exceptions

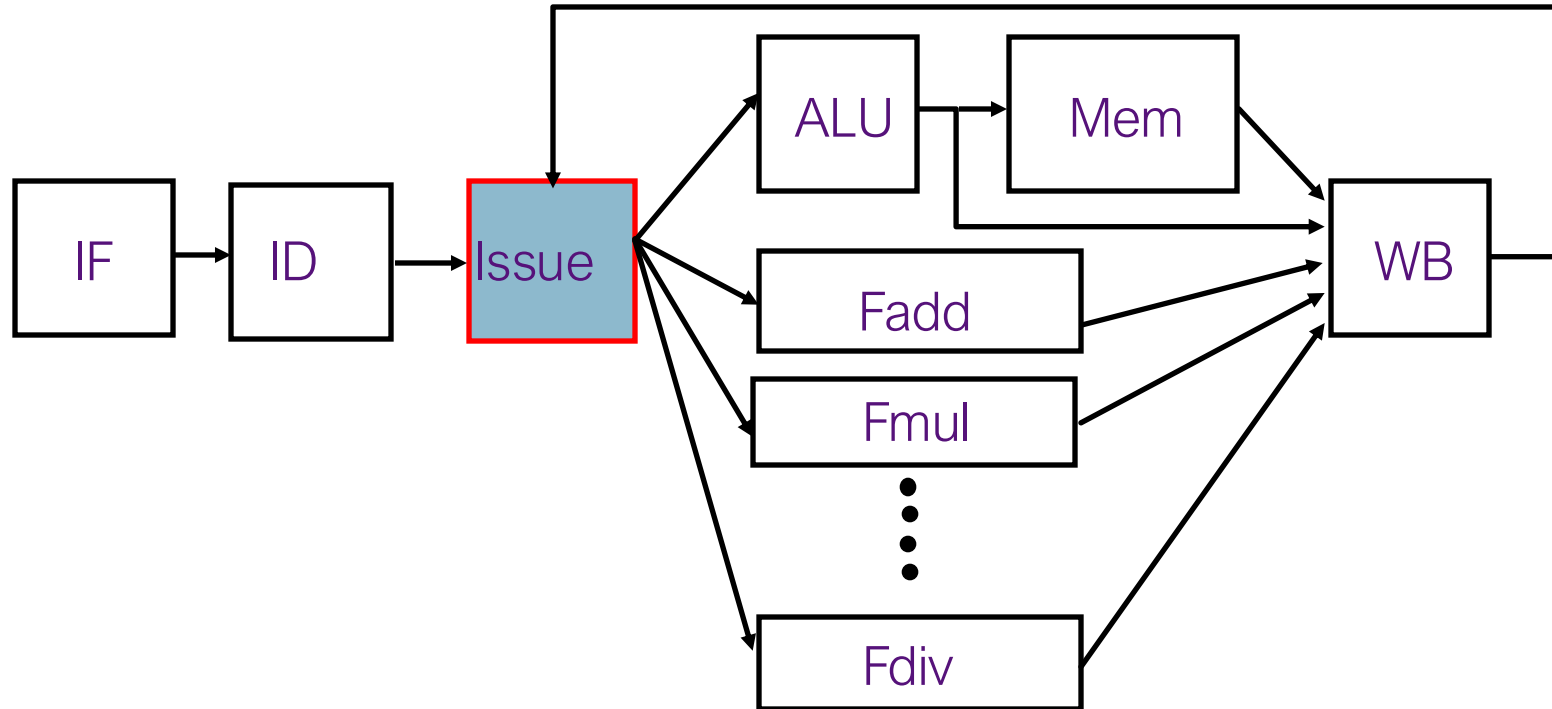
Complex Pipelining

- Structural conflicts at the **execution stage** if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the **write-back stage** due to variable latencies of different functional units



- Out-of-order **write hazards** due to variable latencies of different FUs
- How to handle exceptions?

Complex Pipelining



BALANCED STAGES

Complex In-Order Pipeline

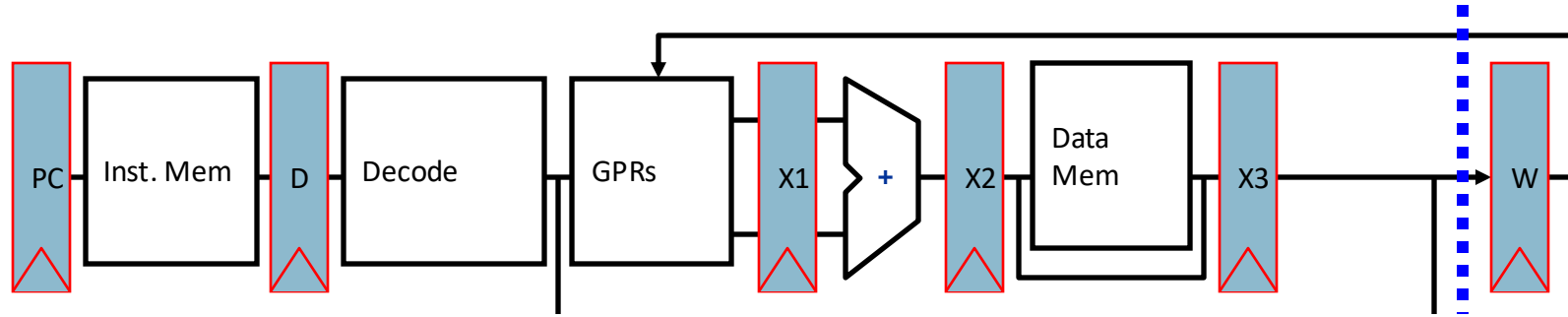
Fetch two instructions per cycle;

- issue both simultaneously if **one is integer/memory** and the **other is floating-point**

Inexpensive way of increasing throughput

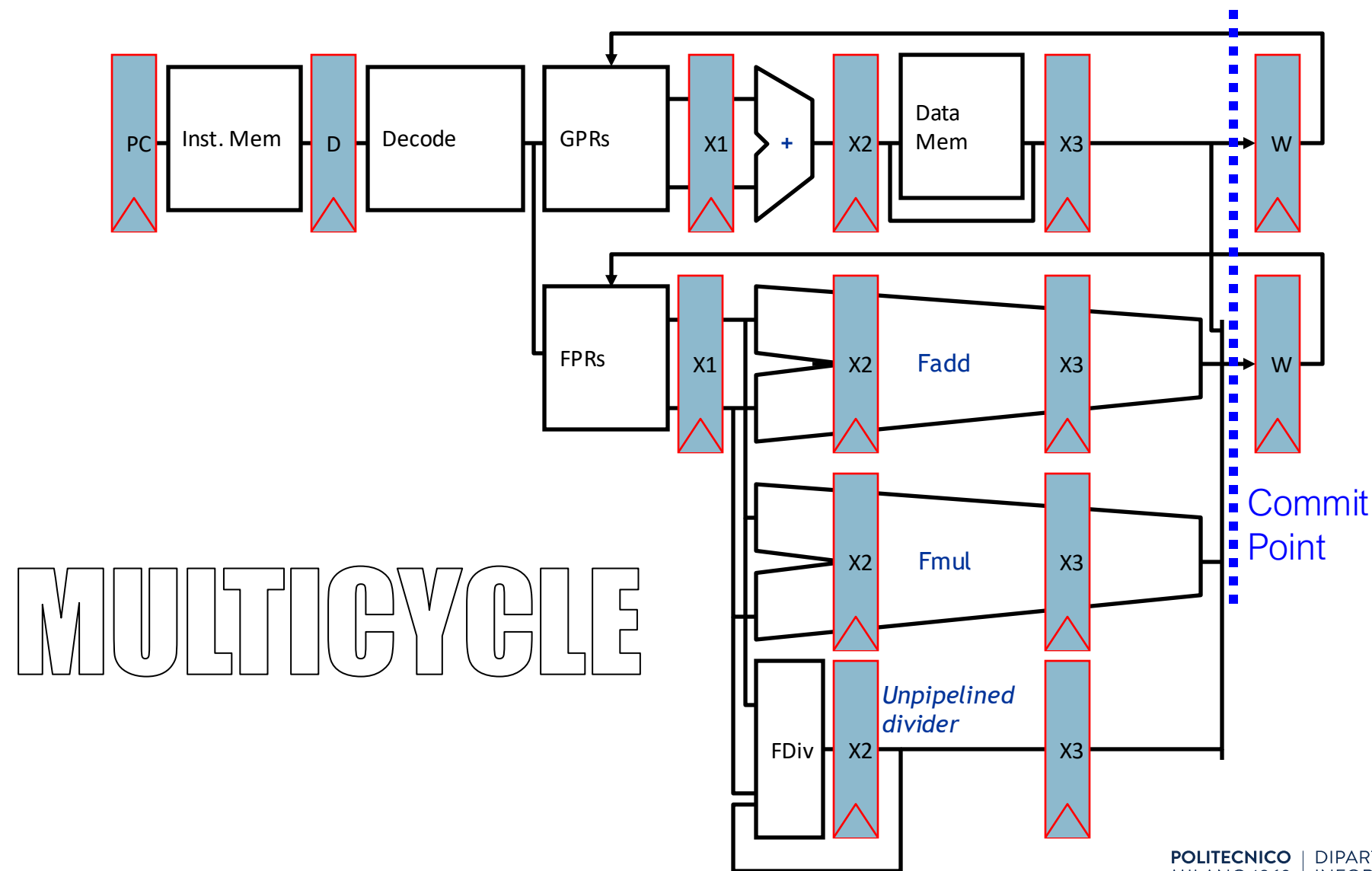
- examples include Alpha 21064 (1992) & MIPS R5000 series (1996)

Complex In-Order Pipeline



HOW TO BALANCE
DIFFERENT EXECUTION

Complex In-Order Pipeline



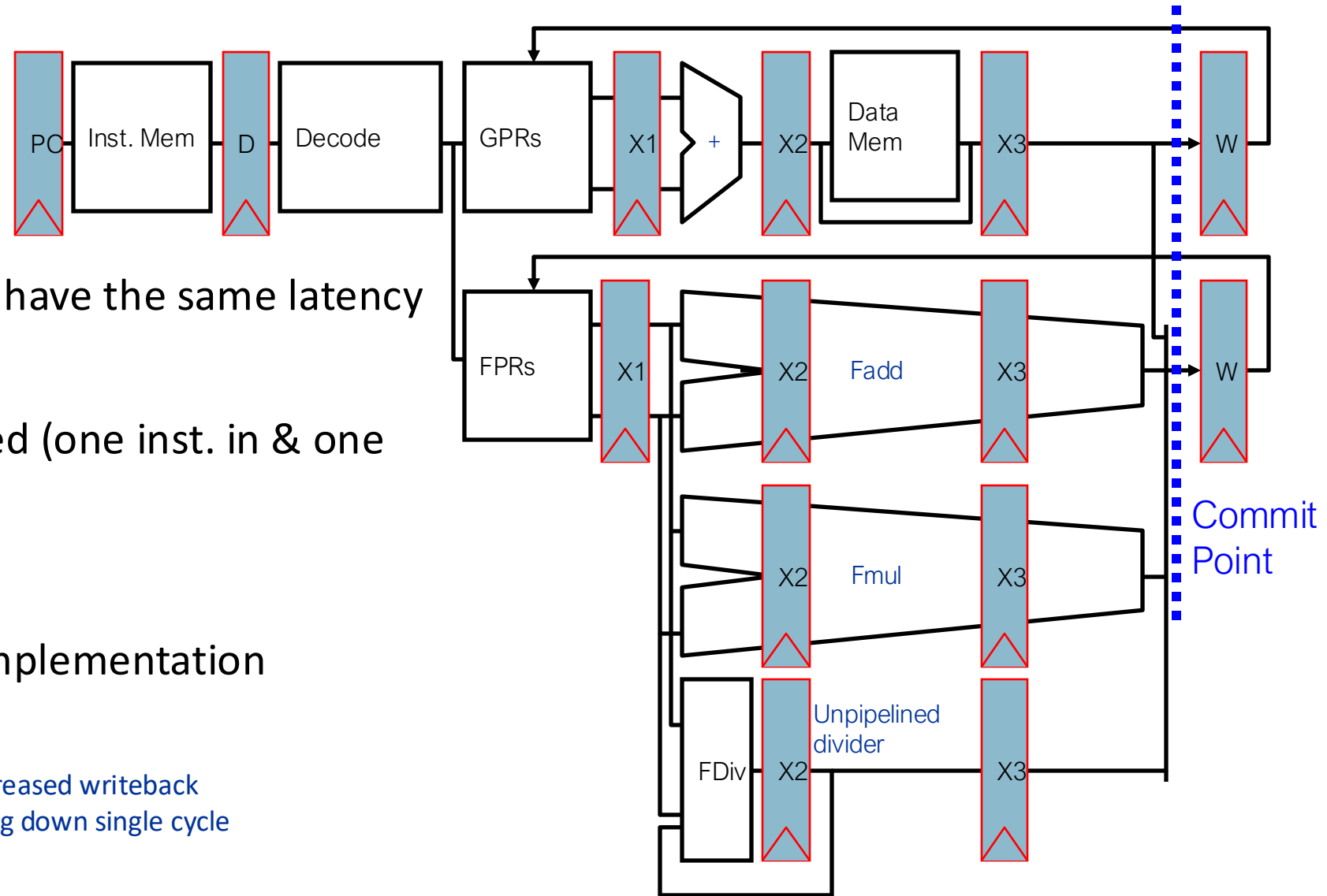
Complex In-Order Pipeline

Delay writeback so all operations have the same latency to the WB stage

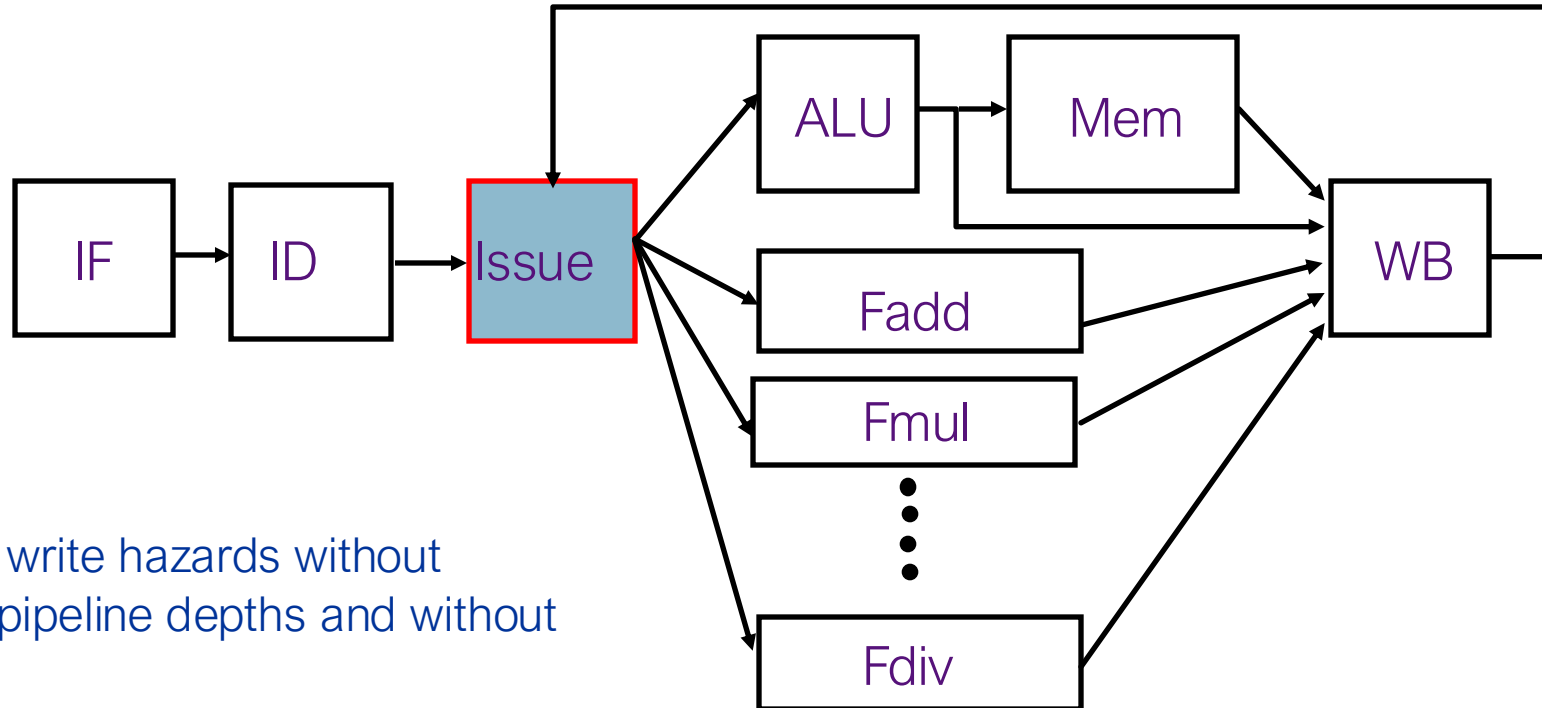
- Write ports never oversubscribed (one inst. in & one inst. out every cycle)
- Instructions commit in order
- It simplifies precise exception implementation

How to prevent increased writeback latency from slowing down single cycle integer operations?

Bypassing



Complex Pipelining

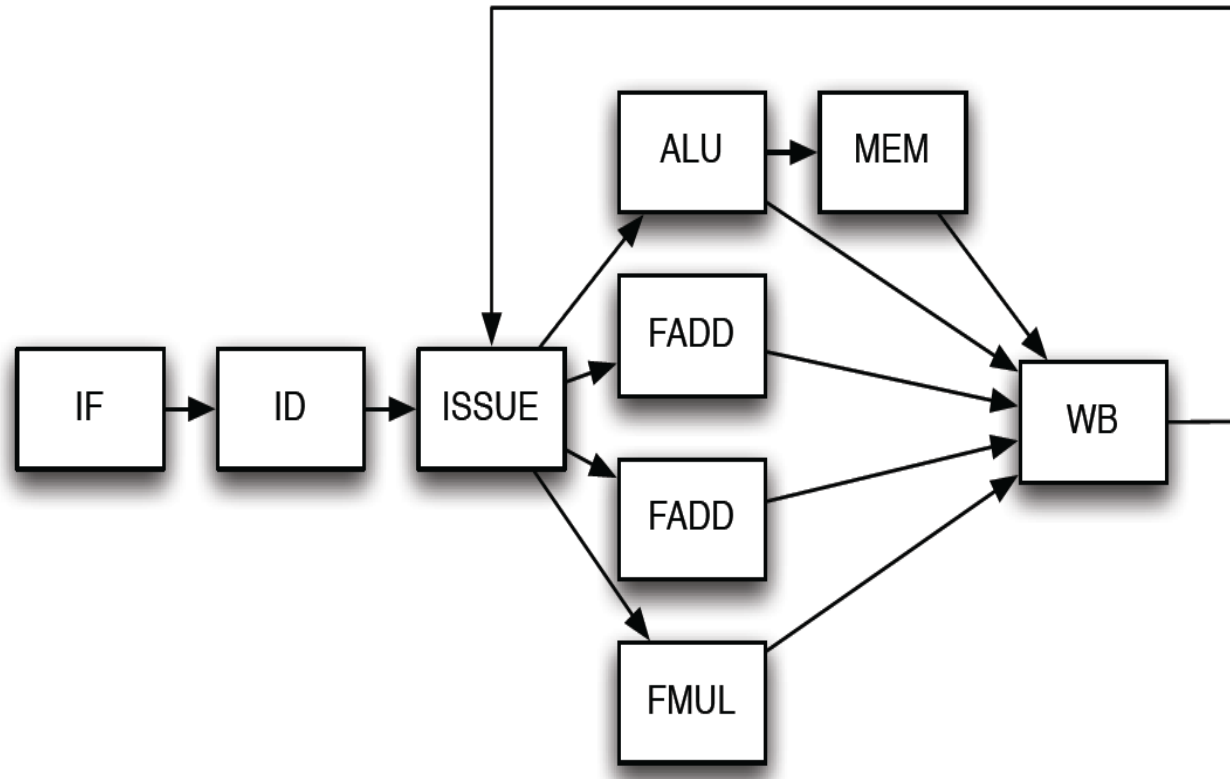


Can we solve write hazards without equalizing all pipeline depths and without bypassing?

When is it Safe to Issue an Instruction?

- Suppose a data structure to keep track of all the instructions in all the functional units
- The following checks need to be made before the **Issue stage** can dispatch an instruction
 - Is the required function unit available?
 - Is the input data available? ---> RAW?
 - Is it safe to write the destination? ---> WAR? WAW?
 - Is there a structural conflict at the WB stage?

Example



Assumptions

- All functional units are pipelined
 - Registers are read in the **Issue stage**: we consider that a register is written in the first half of the clock cycle and read in the second half
 - No forwarding
 - ALU operations take one clock cycle
- Memory operations take two clock cycles (including time in ALU) : one to compute the address and one to access memory
- FP ALU operations take two clock cycles
- The Write Back unit has a single write port -> thus no operation committing at the same time
- Instructions are fetched, decoded, and issued in order
- An instruction will only enter the **Issue stage** if it does not cause a WAR or WAW hazard
- Only one instruction can be issued at a time, and in the case multiple instructions are ready, the oldest one will go first (aka ISSUE seen as an infinite buffer, RAW can enter and will be solved here)

Example

Consider the following code

```
lw      $f1,    BASEA($r1)
```

```
add.d   $f1,    $f1,    $f5
```

```
add.d   $f3,    $f2,    $f1
```

```
add.d   $f1,    $f2,    $f3
```

```
sw.d    $f1,    BASEA($r1)
```

Example

Consider the following code

lw \$f1, BASEA(\$r1)

RAW f1

add.d \$f1, \$f1, \$f5

add.d \$f3, \$f2, \$f1

add.d \$f1, \$f2, \$f3

sw.d \$f1, BASEA(\$r1)

Example

Consider the following code

lw \$f1, BASEA(\$r1)

add.d \$f1, \$f1, \$f5

add.d \$f3, \$f2, \$f1

add.d \$f1, \$f2, \$f3

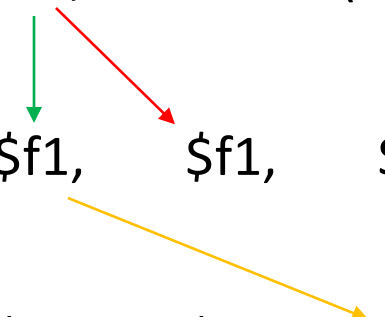
sw.d \$f1, BASEA(\$r1)

RAW f1, WAW f1

Example

Consider the following code

```
lw      $f1,    BASEA($r1)
add.d   $f1,    $f1,    $f5
add.d   $f3,    $f2,    $f1
add.d   $f1,    $f2,    $f3
sw.d    $f1,    BASEA($r1)
```

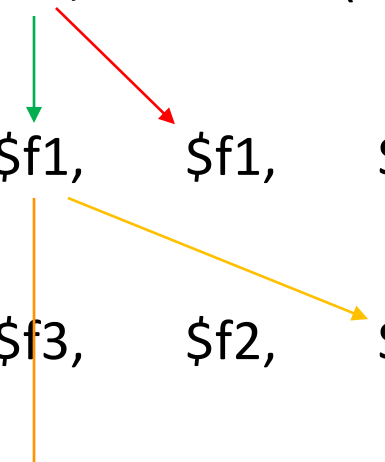


RAW f1, WAW f1
RAW f1

Example

Consider the following code

```
lw      $f1,    BASEA($r1)
add.d   $f1,    $f1,    $f5
add.d   $f3,    $f2,    $f1
add.d   $f1,    $f2,    $f3
sw.d    $f1,    BASEA($r1)
```



RAW f1, WAW f1

RAW f1

WAW f1

Example

Consider the following code

```
lw      $f1,    BASEA($r1)
add.d   $f1,    $f1,    $f5
add.d   $f3,    $f2,    $f1
add.d   $f1,    $f2,    $f3
sw.d    $f1,    BASEA($r1)
```

RAW f1, WAW f1
RAW f1
WAW f1
WAR f1

Example

Consider the following code

```
lw      $f1,    BASEA($r1)
add.d   $f1,    $f1,    $f5
add.d   $f3,    $f2,    $f1
add.d   $f1,    $f2,    $f3
sw.d    $f1,    BASEA($r1)
```

The diagram illustrates data dependencies between instructions in a control flow graph. The instructions are arranged vertically, with arrows indicating the flow of control. Colored arrows indicate specific types of dependencies:

- RAW f1, WAW f1:** A red arrow from the `$f1` in the first instruction to the `$f1` in the second instruction.
- RAW f1:** A green arrow from the `$f1` in the first instruction to the `$f1` in the second instruction.
- WAW f1:** An orange arrow from the `$f1` in the second instruction to the `$f1` in the third instruction.
- WAR f1:** A blue arrow from the `$f1` in the third instruction to the `$f1` in the fourth instruction.
- RAW f3:** A purple arrow from the `$f3` in the third instruction to the `$f3` in the fourth instruction.

RAW f1, WAW f1
RAW f1
WAW f1
WAR f1
RAW f3

Example

Consider the following code

```
lw      $f1,    BASEA($r1)
add.d   $f1,    $f1,    $f5
add.d   $f3,    $f2,    $f1
add.d   $f1,    $f2,    $f3
sw.d    $f1,    BASEA($r1)
```

The control flow graph consists of five nodes representing instructions. Arrows indicate data flow between registers in different instructions:

- From the `$f1` in the first instruction to the `$f1` in the second instruction (green arrow).
- From the `$f1` in the first instruction to the `$f1` in the third instruction (red arrow).
- From the `$f1` in the second instruction to the `$f1` in the third instruction (yellow arrow).
- From the `$f1` in the second instruction to the `$f1` in the fourth instruction (orange arrow).
- From the `$f3` in the third instruction to the `$f3` in the fourth instruction (purple arrow).
- From the `$f2` in the third instruction to the `$f2` in the fourth instruction (light blue arrow).
- From the `$f1` in the fourth instruction to the `$f1` in the fifth instruction (black arrow).

RAW f1, WAW f1

RAW f1

WAW f1

WAR f1

RAW f3

WAW f1

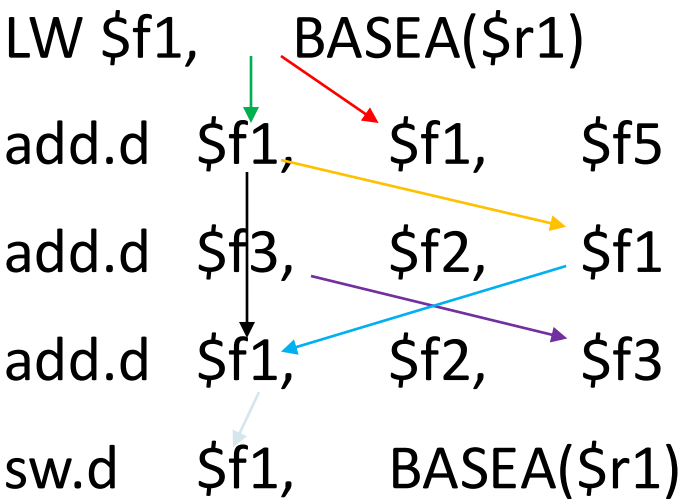
Just a reminder... Assumptions

- All functional units are pipelined
 - Registers are read in the **Issue stage**: we consider that a register is written in the first half of the clock cycle and read in the second half
 - No forwarding
 - ALU operations take one clock cycle
- Memory operations take two clock cycles (including time in ALU)
- FP ALU operations take two clock cycles
- The Write Back unit has a single write port
- Instructions are fetched, decoded, and issued in order
- An instruction will only enter the **Issue stage** if it does not cause a WAR or WAW hazard
- Only one instruction can be issued at a time, and in the case multiple instructions are ready, the oldest one will go first (aka ISSUE seen as an infinite buffer, RAW can enter and will be solved here)

Schedule

First operation is scheduled:

I	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18
1	F	D	I	ALU	M	W												



Schedule

I	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18
1	F	D	I	ALU	M	W												
2		F	D	S(ID)	S(ID)	I	FA	FA	W									

Floating point operation

stall in the decode stage until the first instruction is in the write stage (remember W and I can be in the same clock cycle since the Write is in the first half and I reads in the second half)

Because of the **WAW** we cannot enter the ISSUE before cc6

```

LW $f1, BASEA($r1)
add.d $f1, $f1, $f5
add.d $f3, $f2, $f1
add.d $f1, $f2, $f3
sw.d $f1, BASEA($r1)
  
```

Schedule

I	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18
1	F	D	I	ALU	M	W												
2		F	D	S(ID)	S(ID)	I	FA	FA	W									
3			F	S(IF)	S(IF)	D	I	S(IS)	S(IS)	FA	FA	W						

Remember: registers are read in ISSUE stage and it is an “infinite” buffer -> assumption
 That means that we can enter the ISSUE stage at cc7, RAW solved at cc9

LW \$f1, BASEA(\$r1)
 add.d \$f1, \$f1, \$f5
 add.d \$f3, \$f2, \$f1
 add.d \$f1, \$f2, \$f3
 sw.d \$f1, BASEA(\$r1)

Schedule

I	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18
1	F	D	I	ALU	M	W												
2		F	D	S(ID)	S(ID)	I	FA	FA	W									
3			F	S(IF)	S(IF)	D	I	S(IS)	S(IS)	FA	FA	W						
4						F	D	S(ID)	S(ID)	S(ID)	S(ID)	I	FA	FA	W			

Because of the **WAR** we can enter the ISSUE stage at cc12

LW \$f1, BASEA(\$r1)
 add.d \$f1, \$f1, \$f5
 add.d \$f3, \$f2, \$f1
 add.d \$f1, \$f2, \$f3
 sw.d \$f1, BASEA(\$r1)

Schedule

In case of RAW you enter the Issue Stage (unlike WAR or WAW where you stall in the Decode Stage before entering the IS) and then you stall here until you're ready. This is in the assumptions

I	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18
1	F	D	I	ALU	M	W												
2		F	D	S(ID)	S(ID)	I	FA	FA	W									
3			F	S(IF)	S(IF)	D	I	S(IS)	S(IS)	FA	FA	W						
4						F	D	S(ID)	S(ID)	S(ID)	S(ID)	I	FA	FA	W			
5							F	S(IF)	S(IF)	S(IF)	S(IF)	D	I	S(IS)	S(IS)	ALU	M	W

LW \$f1, BASEA(\$r1)
 add.d \$f1, \$f1, \$f5
 add.d \$f3, \$f2, \$f1
 add.d \$f1, \$f2, \$f3
 sw.d \$f1, BASEA(\$r1)

Because of the RAW we can enter the ISSUE stage at cc15

Getting higher performance...

In a pipelined machine, actual CPI is derived as:

$$CPI_{pipe} = CPI_{ideal} + \text{Structural_stalls} + \text{Data_hazard_stalls} + \text{Control_stalls}$$

Reduction of any right-hand term reduces CPI_{pipe} (increase Instructions Per Clock – IPC)

Technique to increase CPI_{pipe} could create further problems with hazards.

Some basic concepts and definitions

To reach higher performance (for a given technology) – more parallelism must be extracted from the program. In other words...

Dependences must be detected and solved, and instructions must be ordered (**scheduled**) to achieve the highest parallelism of execution compatible with available resources.

Dependences

Determining dependencies among instructions is critical to defining the amount of parallelism existing in a program.

If two instructions are dependent, they cannot execute in parallel; they must be executed in order or only partially overlapped.

Three different types of dependencies:

- Name Dependences

- Data Dependences (or True Data Dependences)

- Control Dependences

Name Dependences

Name dependence occurs when two instructions use the same register or memory location (called name), but there is no data flow between the instructions associated with that name

Two types of name dependences between an instruction i that precedes instruction j in program order:

Antidependence: when j writes a register or memory location that instruction i reads. The original instructions ordering must be preserved to ensure that i reads the correct value.

Output Dependence: when i and j write the same register or memory location. The original instructions ordering must be preserved to ensure that the value finally written corresponds to j .

Name Dependences

Name dependences are not true data dependences, since there is no value (no data flow) being transmitted between instructions

- If the name (register number or memory location) used in the instructions could be changed, the instructions do not conflict

Dependences through memory locations are more difficult to detect (“[memory disambiguation](#)” problem), since two addresses may refer to the same location but can look different.

- Register renaming can be more easily done
- Renaming can be done either statically by the compiler or dynamically by the hardware

Data Dependences and Hazards

A data/name dependence can potentially generate a data hazard (RAW, WAW, or WAR), but the actual hazard and the number of stalls to eliminate the hazards are a property of the pipeline

RAW hazards correspond to true data dependences

WAW hazards correspond to output dependences

WAR hazards correspond to antidependences

Dependences are a property of the program, while hazards are a property of the pipeline

Control Dependences

Control dependence determines the ordering of instructions and it is preserved by two properties:

- Instructions execution in program order to ensure that an instruction that occurs before a branch is executed before the branch
- Detection of control hazards to ensure that an instruction (that is control dependent on a branch) is not executed until the branch direction is known

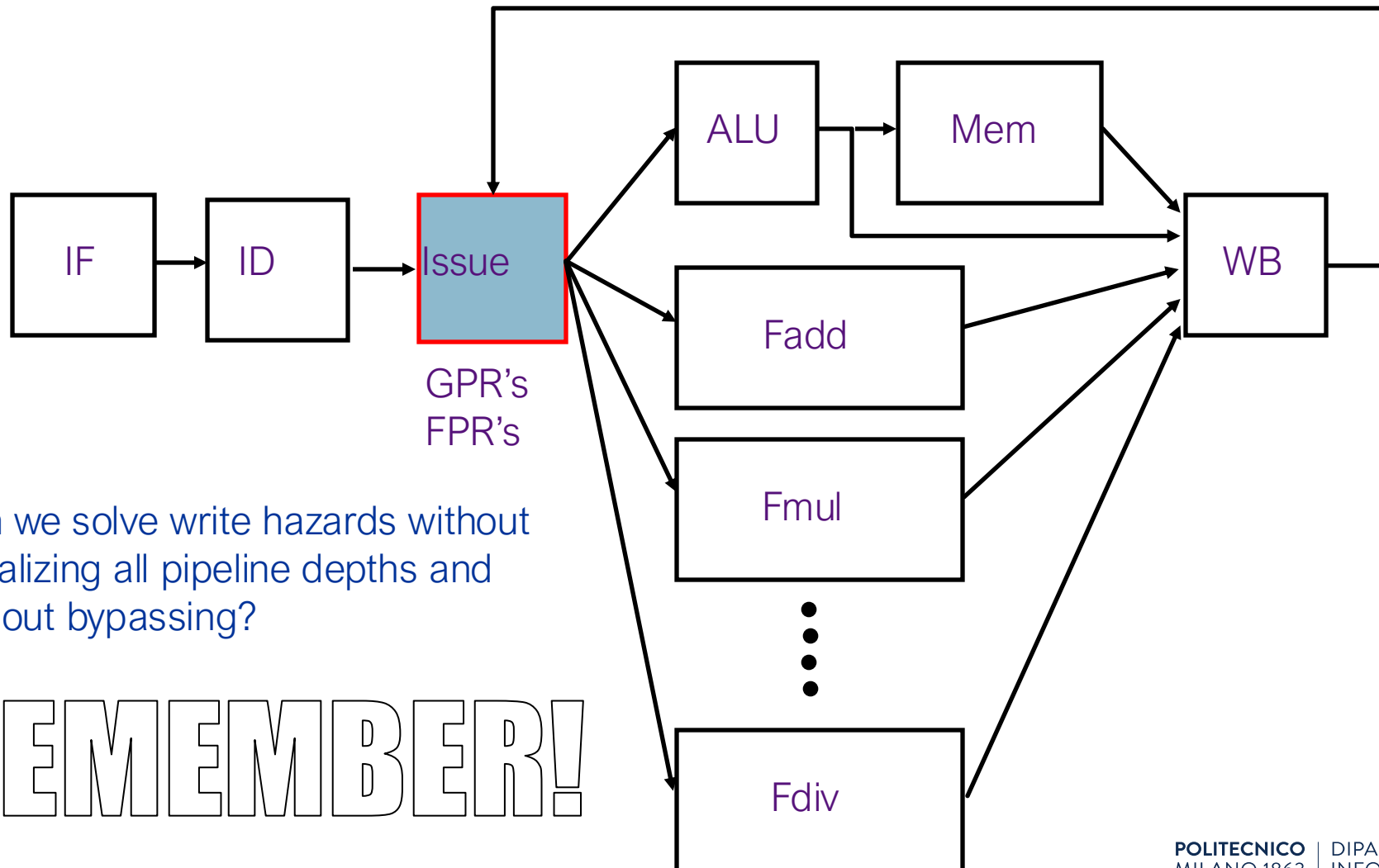
Although preserving control dependence is a simple way to preserve program order, control dependence is not the critical property that must be preserved

Program Properties

Two properties are critical to program correctness (and normally preserved by maintaining both data and control dependences):

- **Exception behavior:** Preserving exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.
- **Data flow:** Actual flow of data values among instructions that produces the correct results and consumes them.

Complex Pipeline



Can we solve write hazards without equalizing all pipeline depths and without bypassing?

REMEMBER!

Instruction Level Parallelism

Two strategies to support ILP:

Dynamic Scheduling: Depend on the hardware to locate parallelism

Static Scheduling: Rely on software for identifying potential parallelism

Hardware-intensive approaches dominate desktop and server markets

Dynamic Scheduling

The hardware reorders the instruction execution to reduce pipeline stalls while maintaining data flow and exception behavior

Main advantages:

- It enables handling some cases where dependencies are unknown at compile time
- It simplifies the compiler complexity
- It allows compiled code to run efficiently on a different pipeline.

Those advantages are gained at a cost:

- A significant increase in hardware complexity,
- Increased power consumption
- Could generate an imprecise exception

Dynamic Scheduling

Basically: Instructions are fetched and issued in program order (in-order-issue)

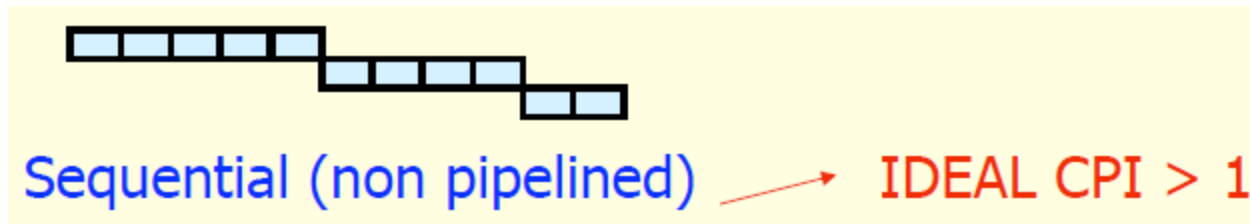
Execution begins as soon as operands are available

- possibly, out-of-order execution – note: possible even with pipelined scalar architectures

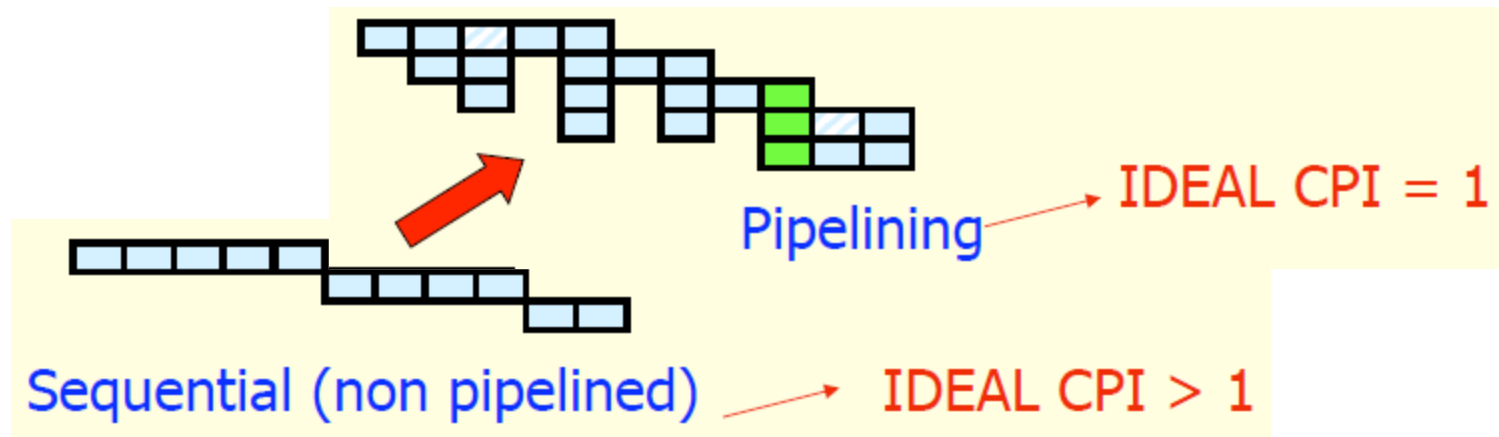
Out-of-order execution introduces the possibility of WAR and WAW data hazards

Out-of-order execution implies **out-of-order completion** unless there is a re-order buffer to get in-order completion

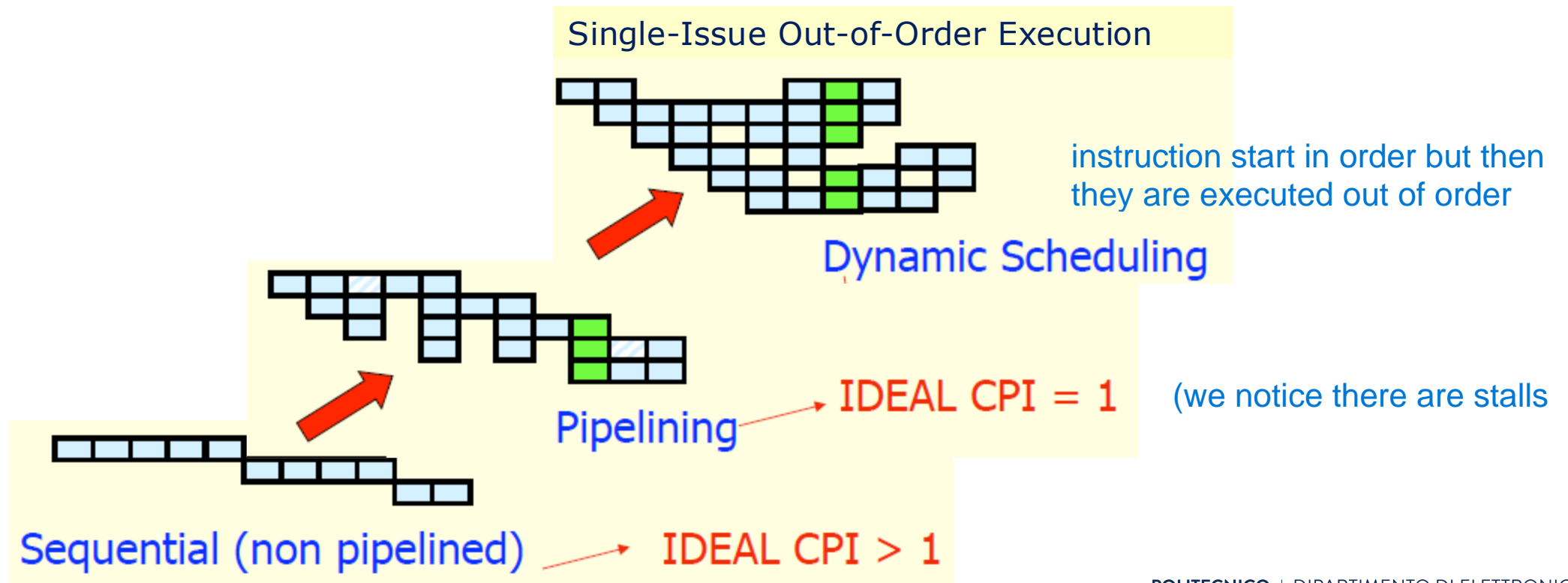
Several steps towards exploiting more ILP



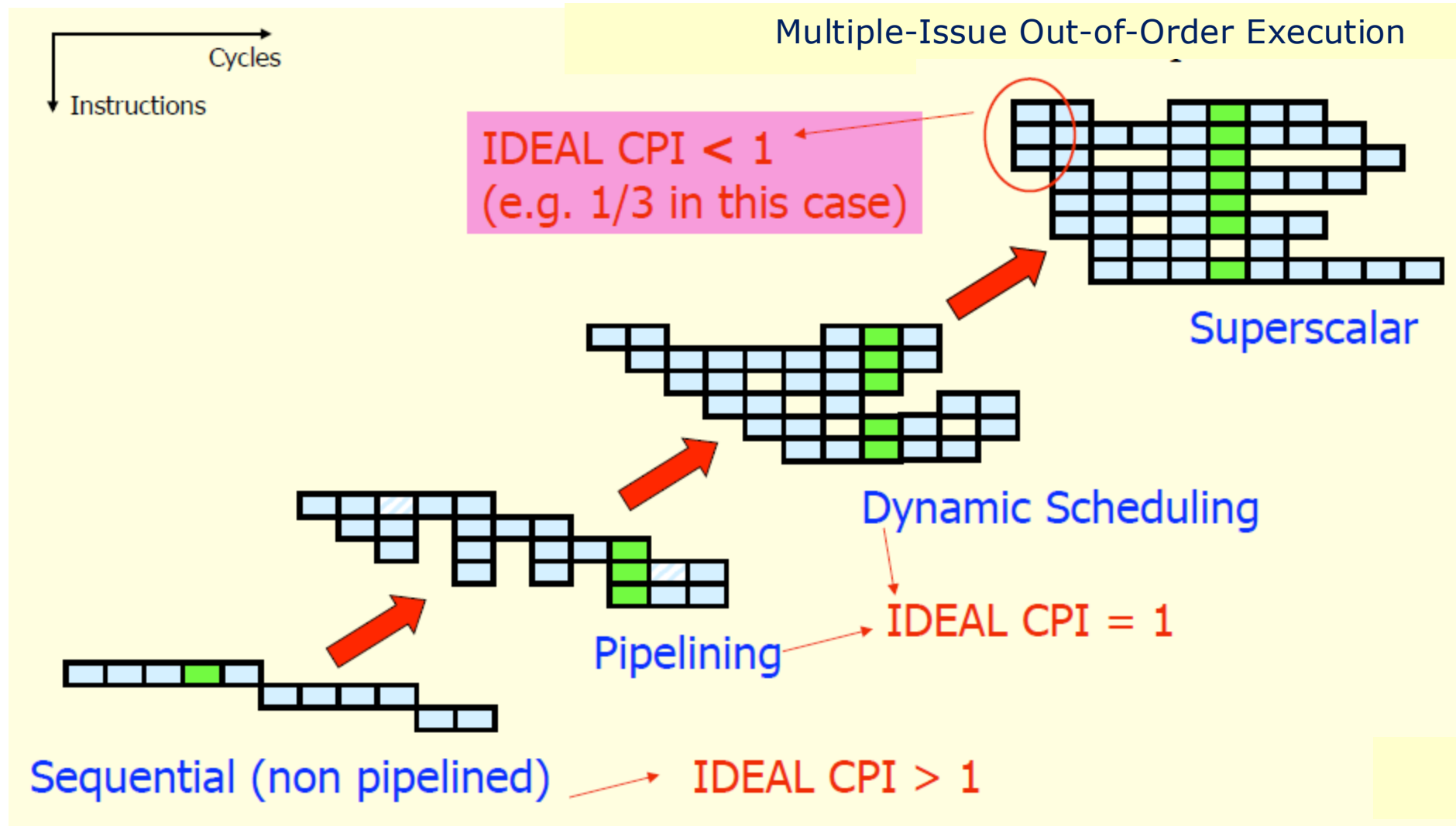
Several steps towards exploiting more ILP



Several steps towards exploiting more ILP



Several steps towards exploiting more ILP



Static Scheduling

Compilers can use sophisticated algorithms for code scheduling to exploit ILP (Instruction Level Parallelism)

The amount of parallelism available within a basic block – *a straight-line code sequence with no branches in except to the entry and no branches out except at the exit* – is quite small

Example: For typical MIPS programs, the average branch frequency is between 15% and 25% \Rightarrow from 4 to 7 instructions executed between a pair of branches.

Static Scheduling

Data dependencies can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size

To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks (i.e. across branches)

Detection and resolution of dependences

Static detection and resolution of dependences ([static scheduling](#)): accomplished by the compiler \Rightarrow dependencies are avoided by code reordering

Output of the compiler: reordered into dependency-free code

Typical example: VLIW (Very Long Instruction Word) processors expect [dependency-free code](#)

Limits of Static Scheduling

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity



Questions?