

Advanced Computer Architectures

(High Performance Processors and Systems)

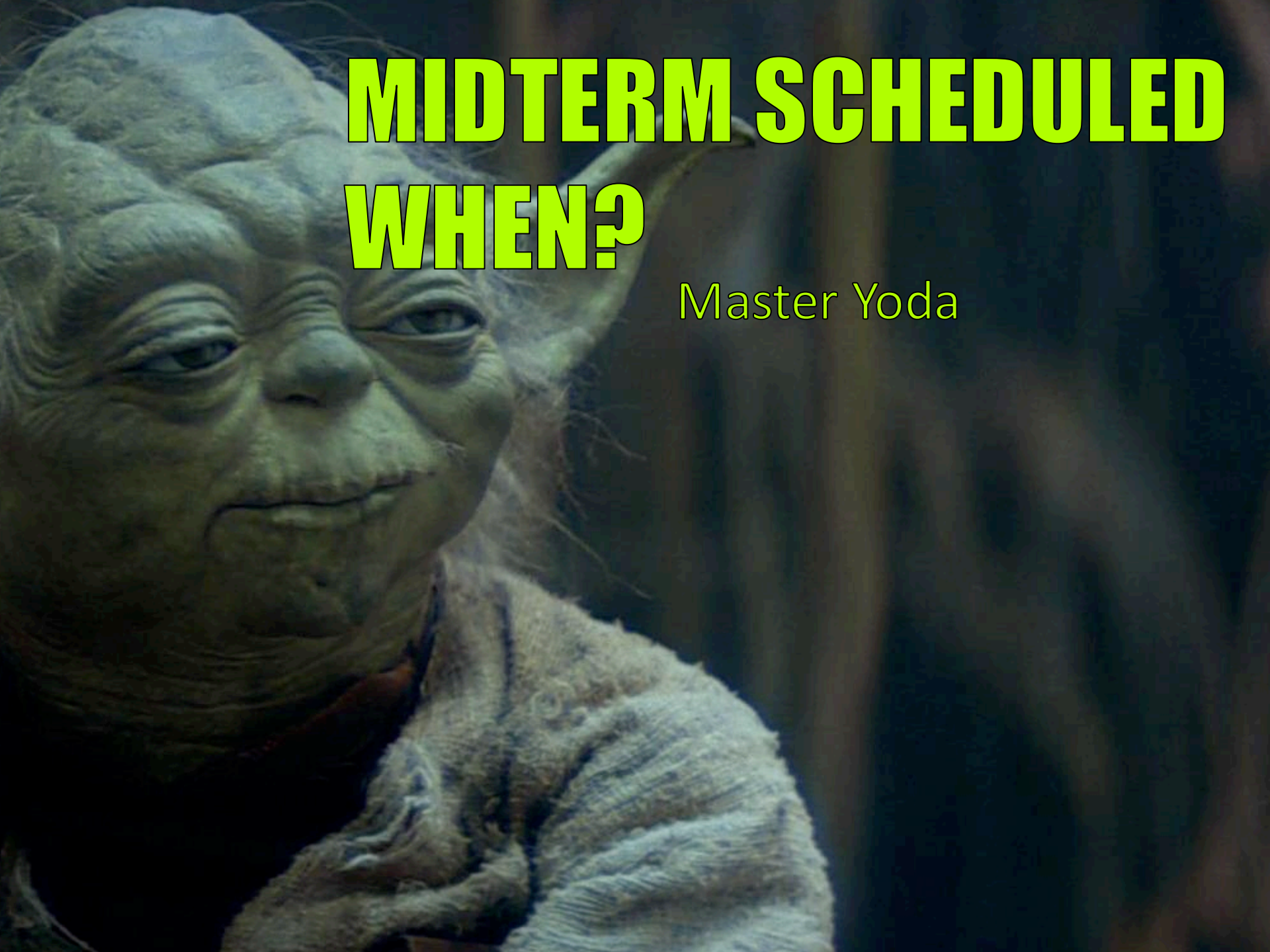
Introduction to Instruction Level Parallelism

Politecnico di Milano

v1

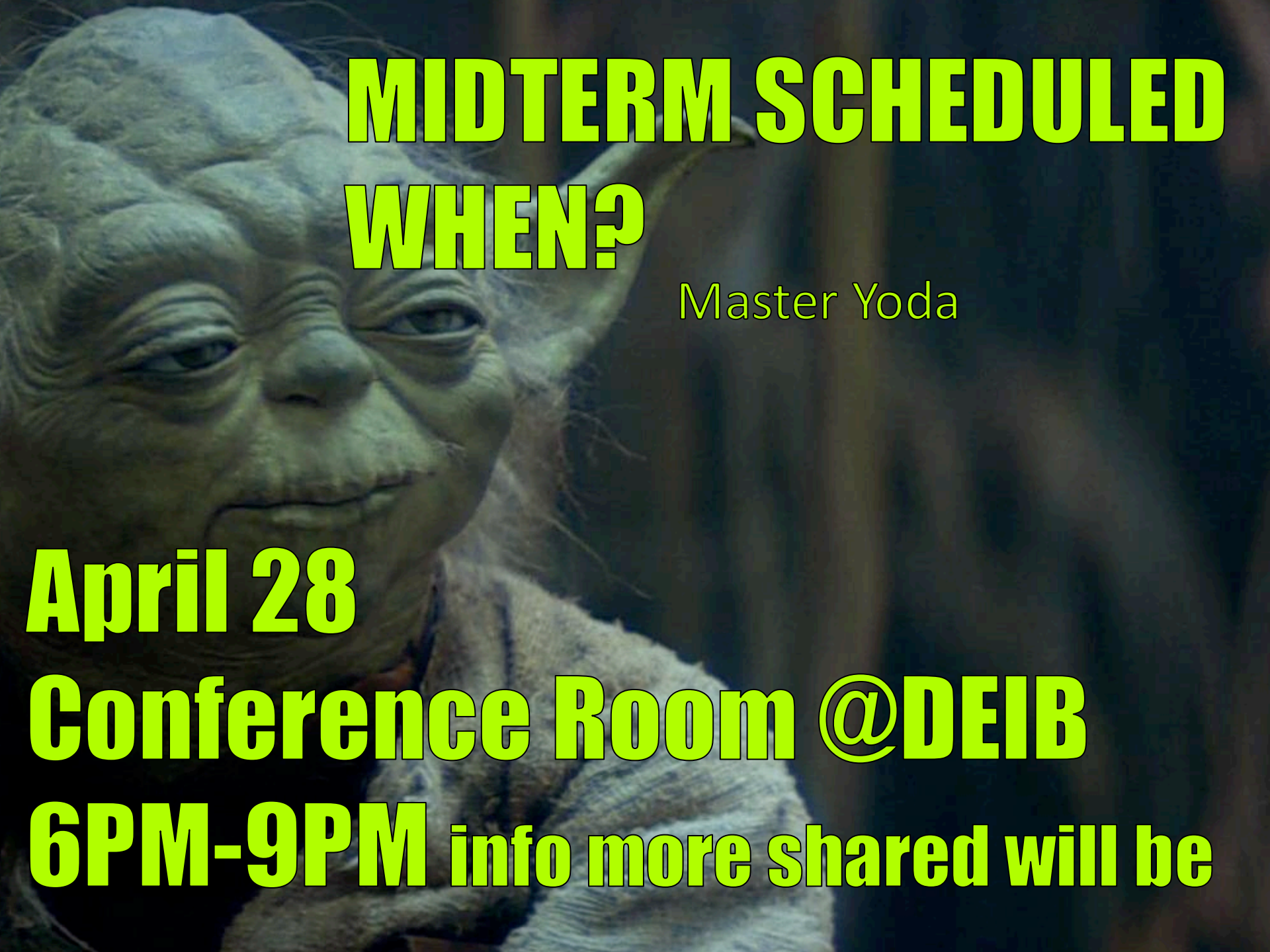
Alessandro Verosimile <alessandro.verosimile@polimi.it>

Marco D. Santambrogio <marco.santambrogio@polimi.it>



**MIDTERM SCHEDULED
WHEN?**

Master Yoda



MIDTERM SCHEDULED WHEN?

Master Yoda

April 28

Conference Room @DEIB

6PM-9PM info more shared will be

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 =

Review: Pipeline performance

- Pipeline CPI = Ideal pipeline CPI +
 pipeline CPI: measure of the maximum
 performance attainable by the implementation
 - Structural hazards: HW cannot support this

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
 - Data hazards: Instruction depends on result of prior

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 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 result of prior instruction still in the pipeline
 - **Control hazards**: Caused by 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 length of pipe increases impact of hazards
- Pipelining helps instruction bandwidth, not latency

Review: Types of Data Hazards

Data-dependence


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



Review: Types of Data Hazards


Data-dependence

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



Anti-dependence


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



Review: Types of Data Hazards


Data-dependence

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




Anti-dependence

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



Output-dependence

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

		<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

		<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

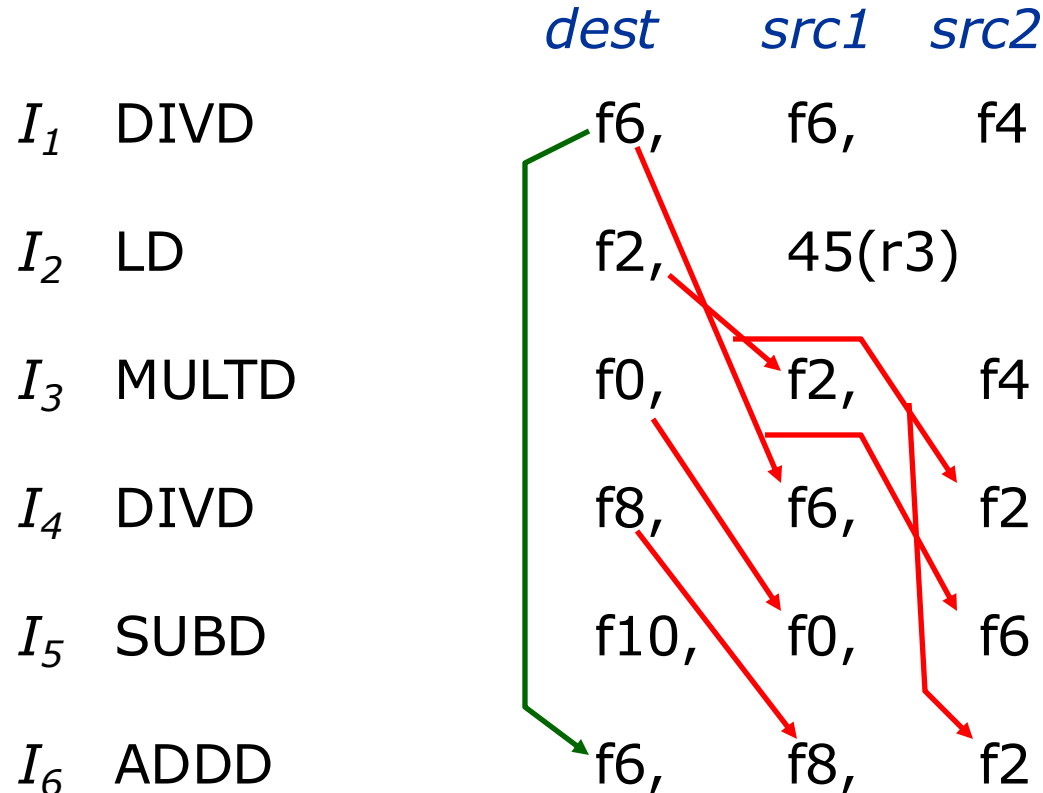
RAW Hazards

Data Hazards: An Example

		<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

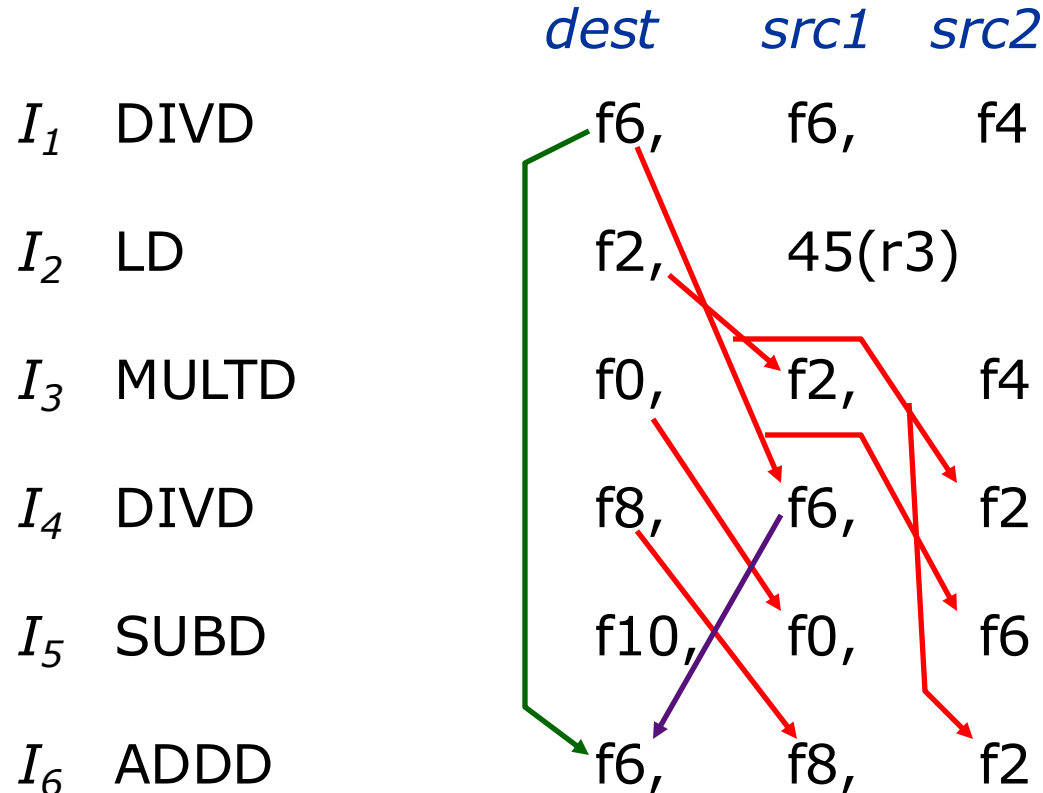
RAW Hazards

Data Hazards: An Example



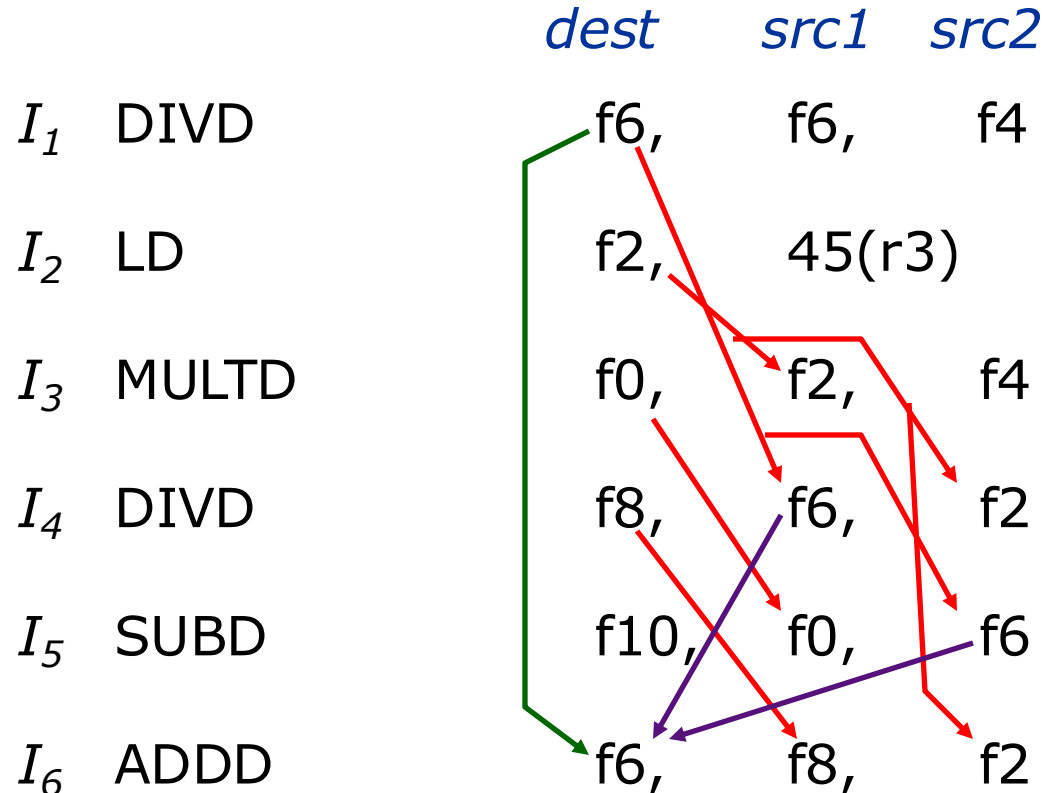
RAW Hazards *WAW Hazards*

Data Hazards: An Example



RAW Hazards *WAW Hazards* *WAR Hazards*

Data Hazards: An Example



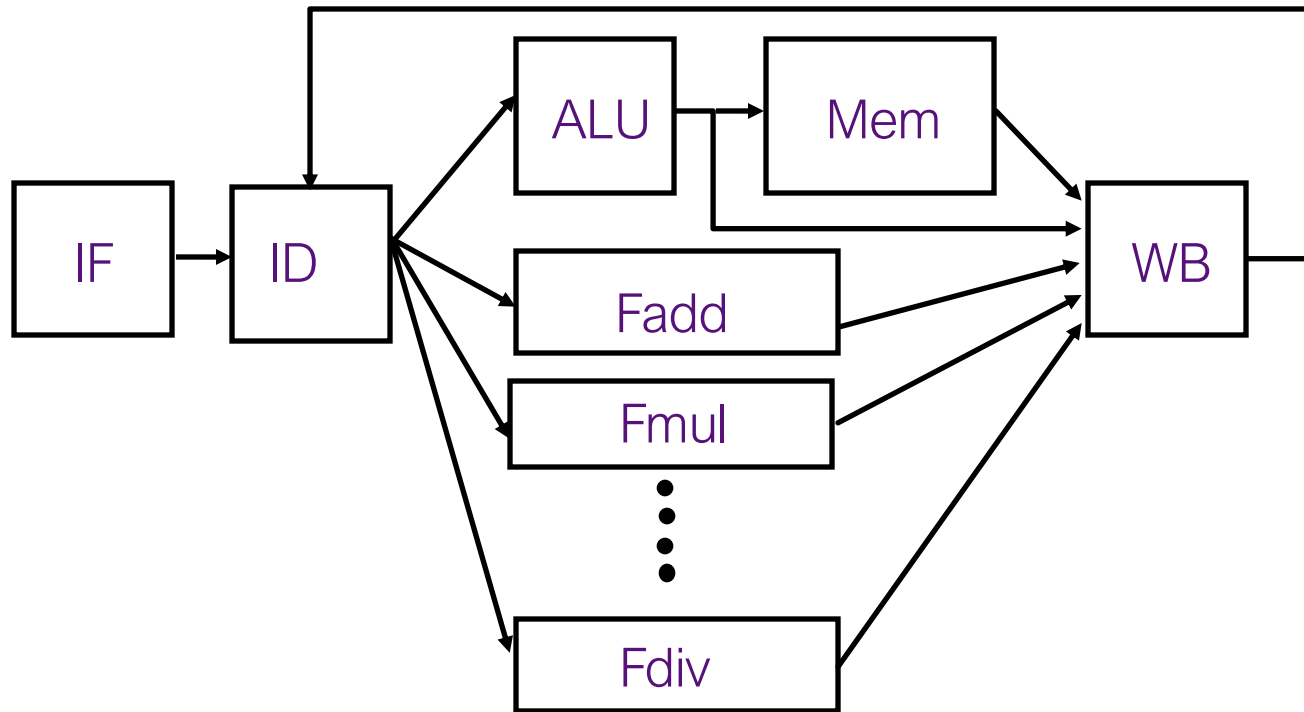
RAW Hazards *WAW Hazards* *WAR Hazards*

A new scenario

- What about mixing Integer and Floating point operations?

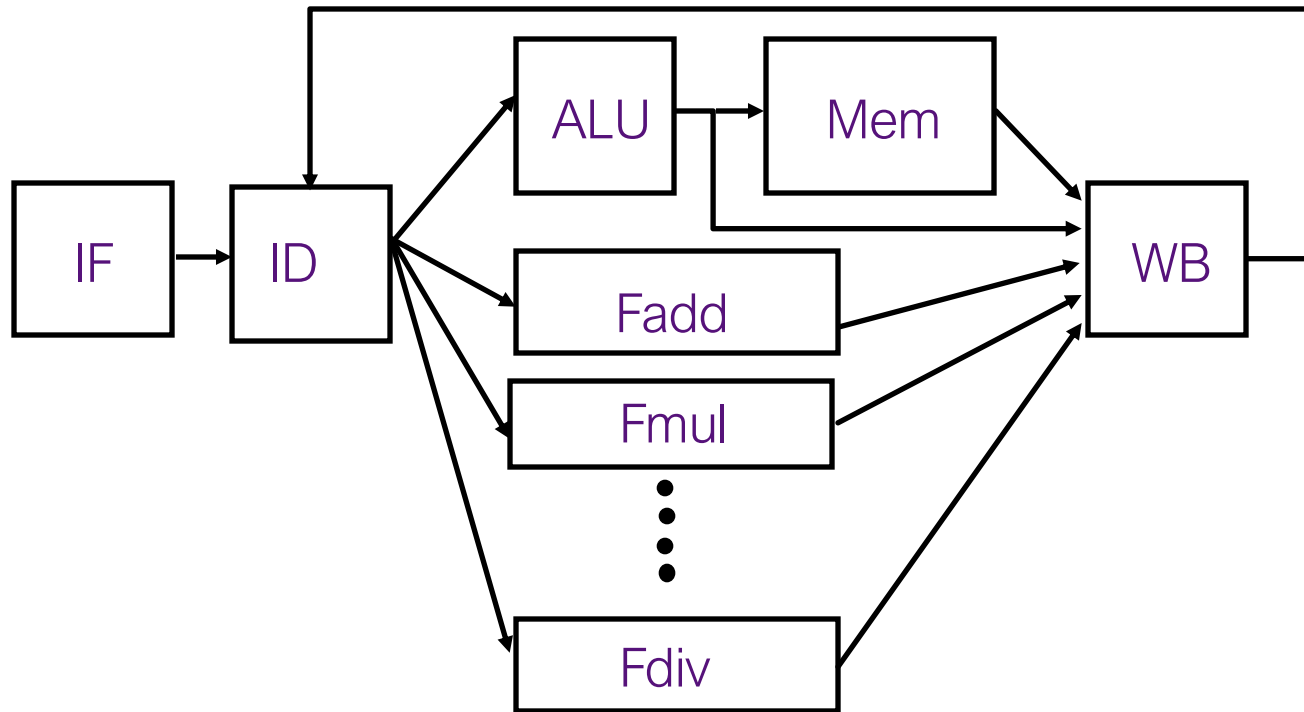
A new scenario

- What about mixing Integer and Floating point operations?



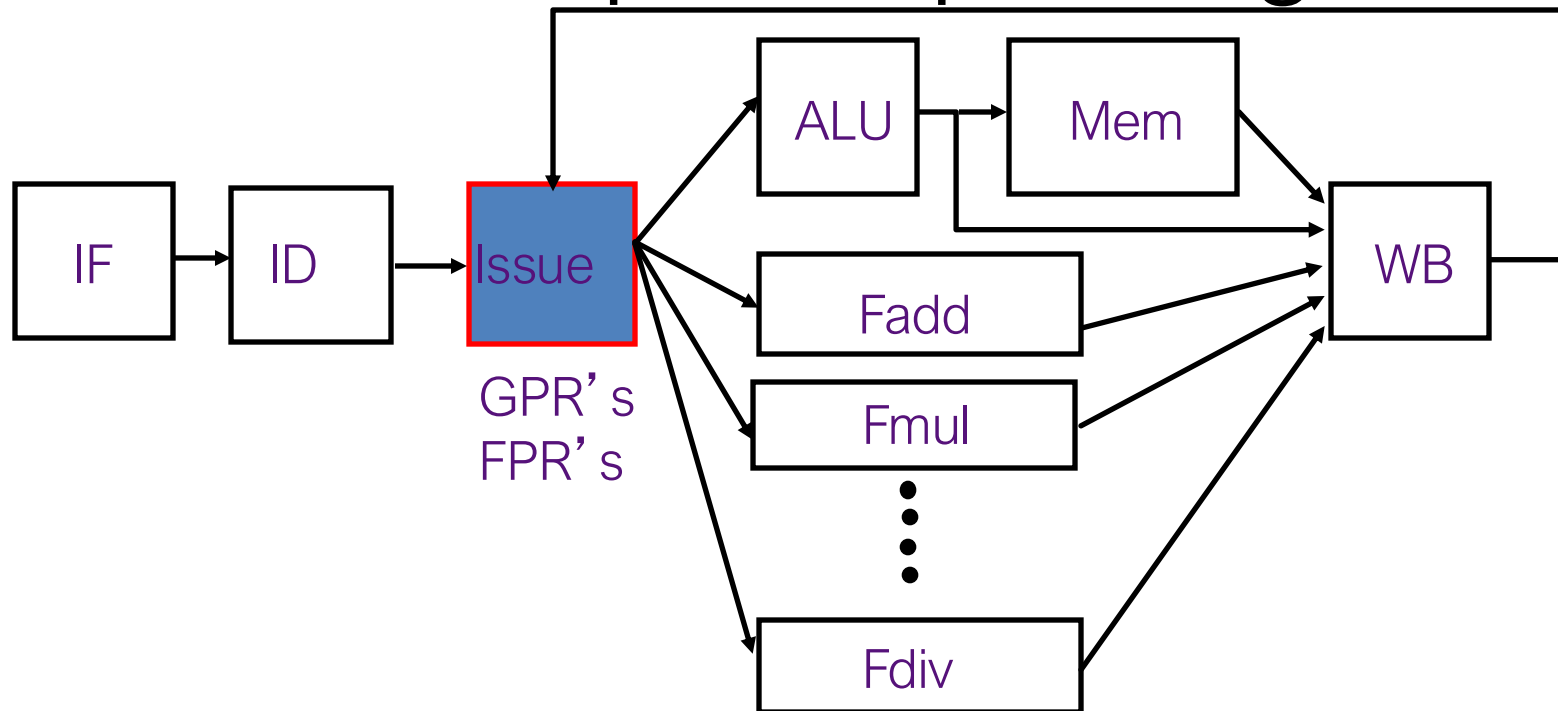
A new scenario

- What about mixing Integer and Floating point operations?



- How to handle different registers?

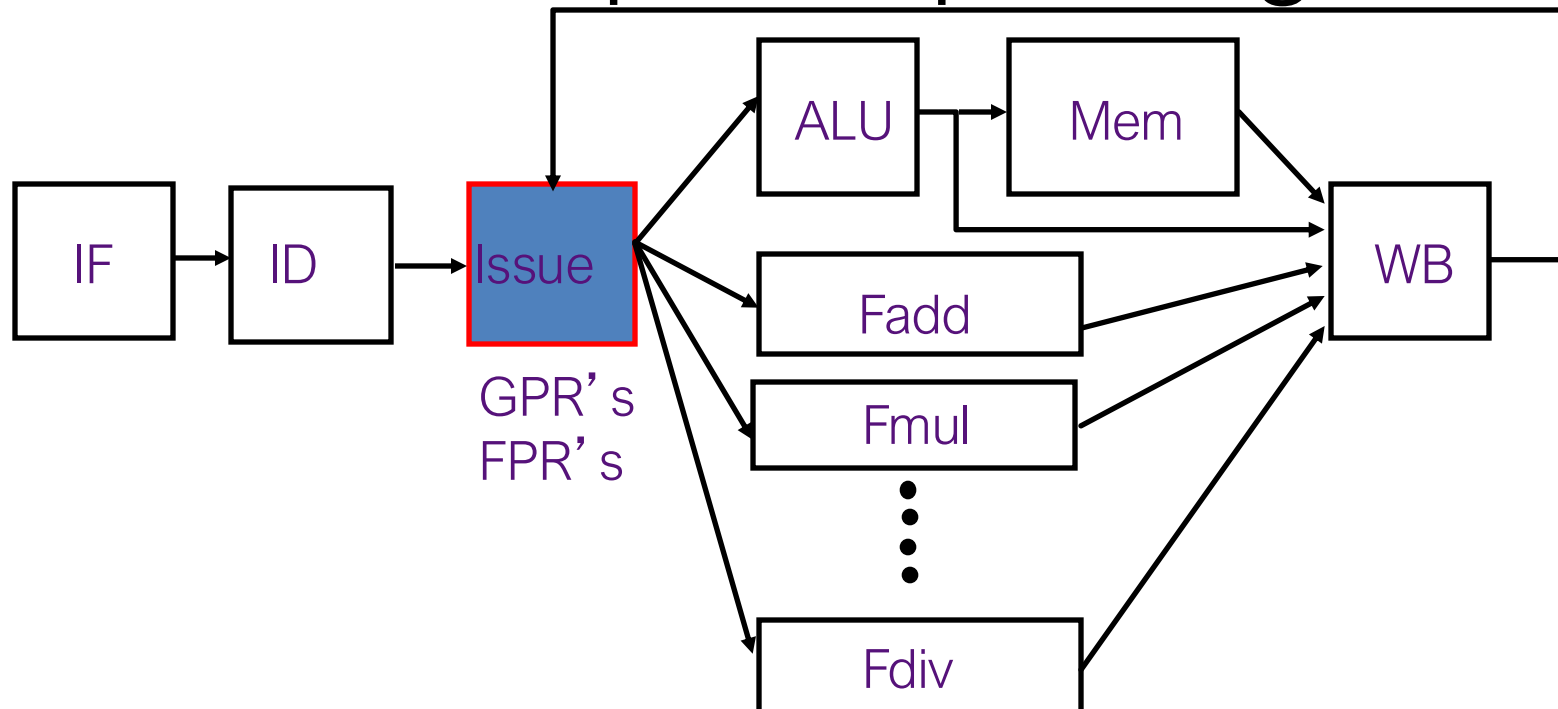
Complex Pipelining



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

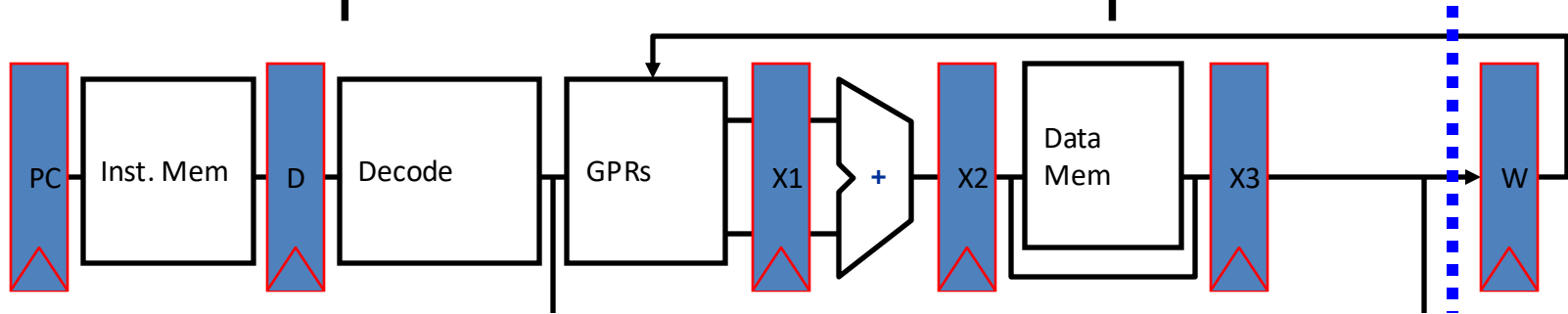
- Long latency or partially pipelined floating-point units
- Multiple function and memory units
- Memory systems with variable access time
- Precise exception

Complex Pipelining



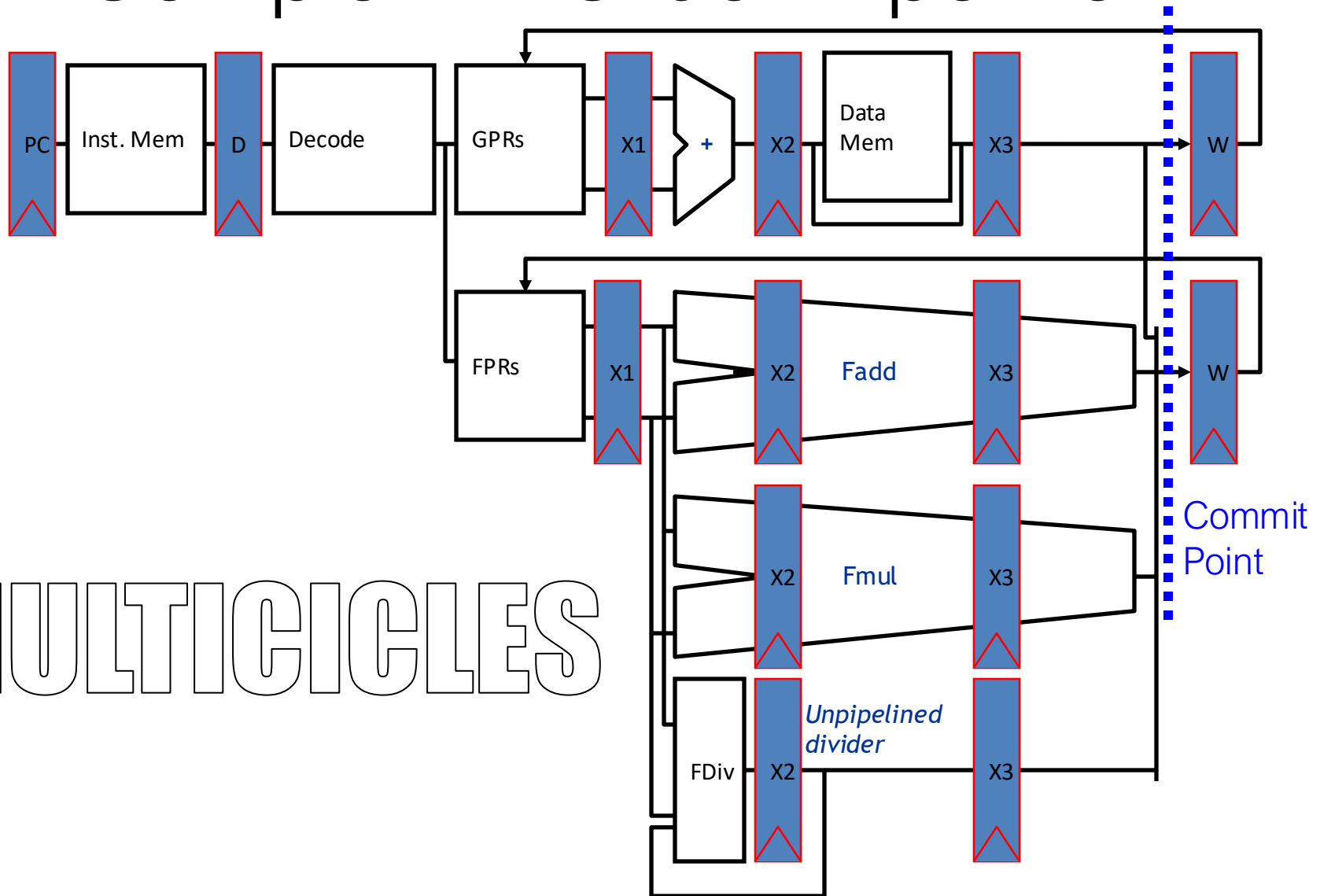
BALANCED STAGES

Complex In-Order Pipeline



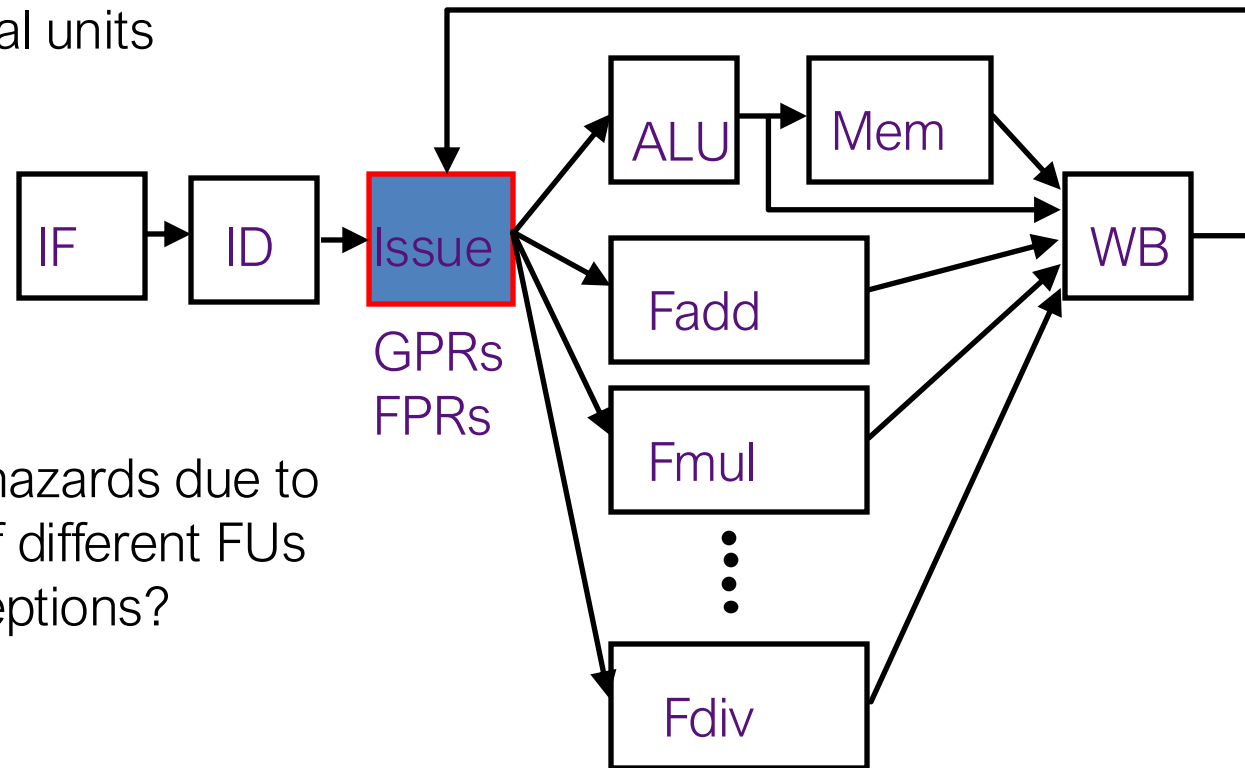
HOW TO BALANCE
DIFFERENT EXECUTION

Complex In-Order Pipeline



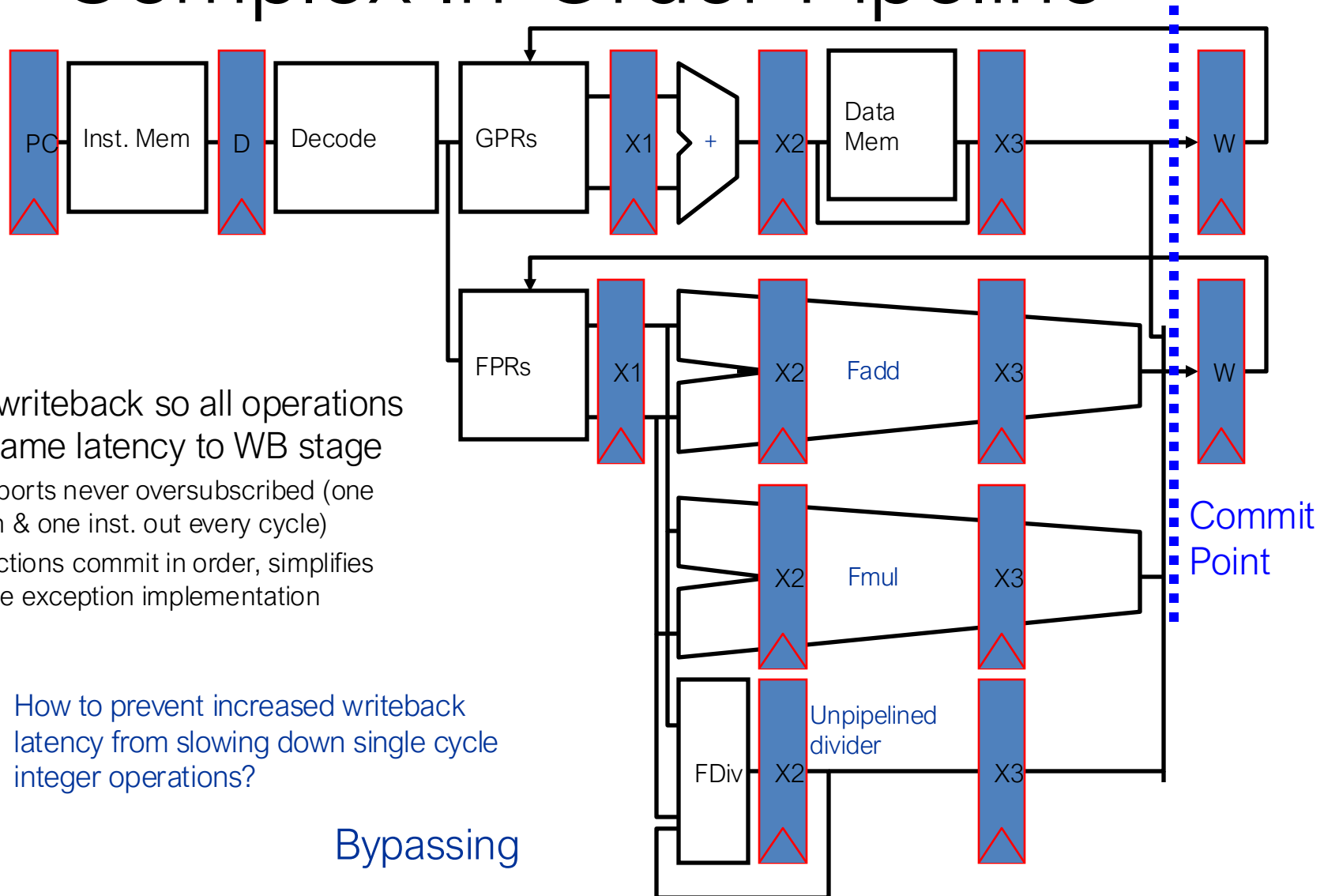
Issues in Complex Pipeline Control

- 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?

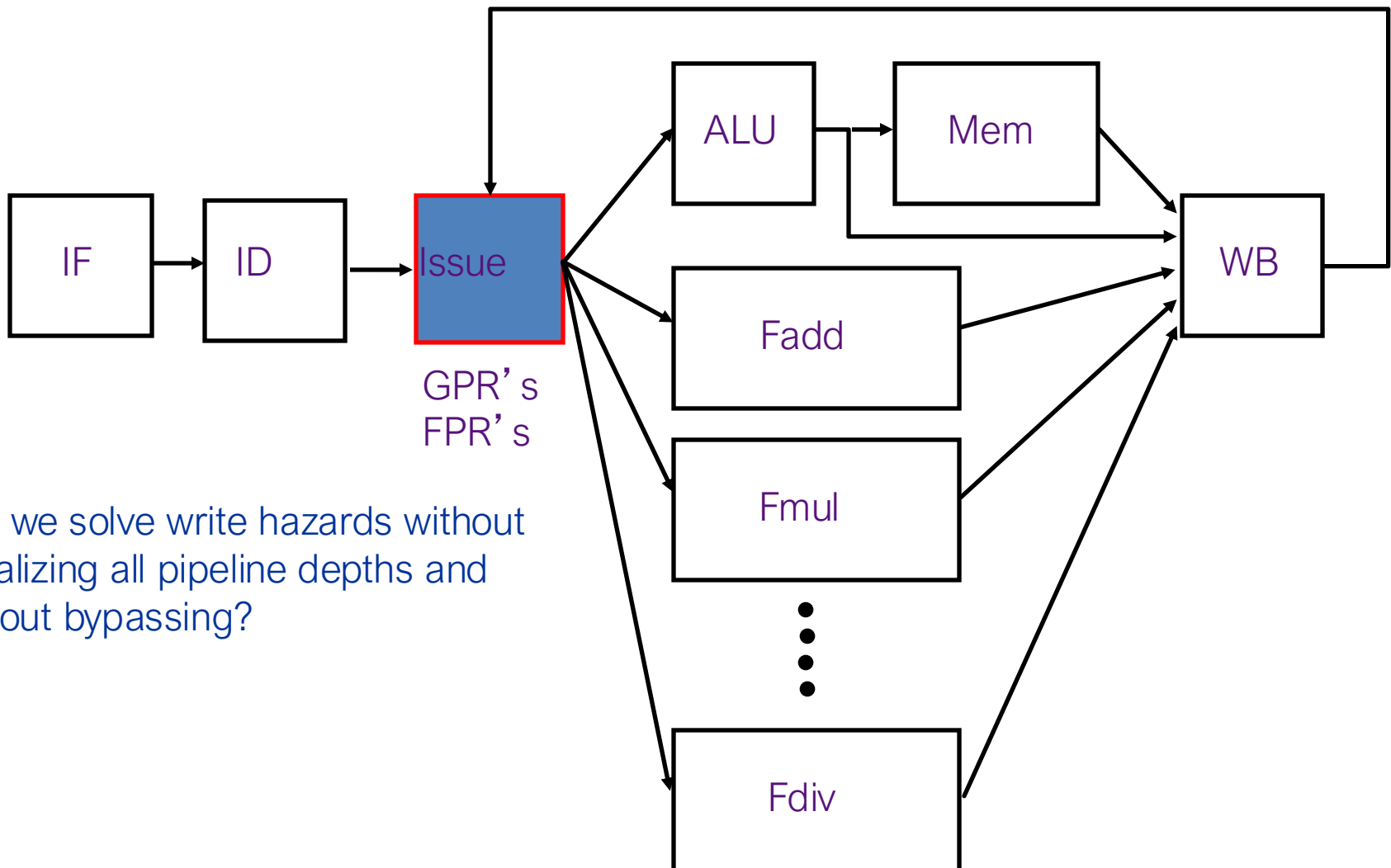
Complex In-Order Pipeline



Delay writeback so all operations have same latency to WB stage

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

Complex Pipeline



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 keeps 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

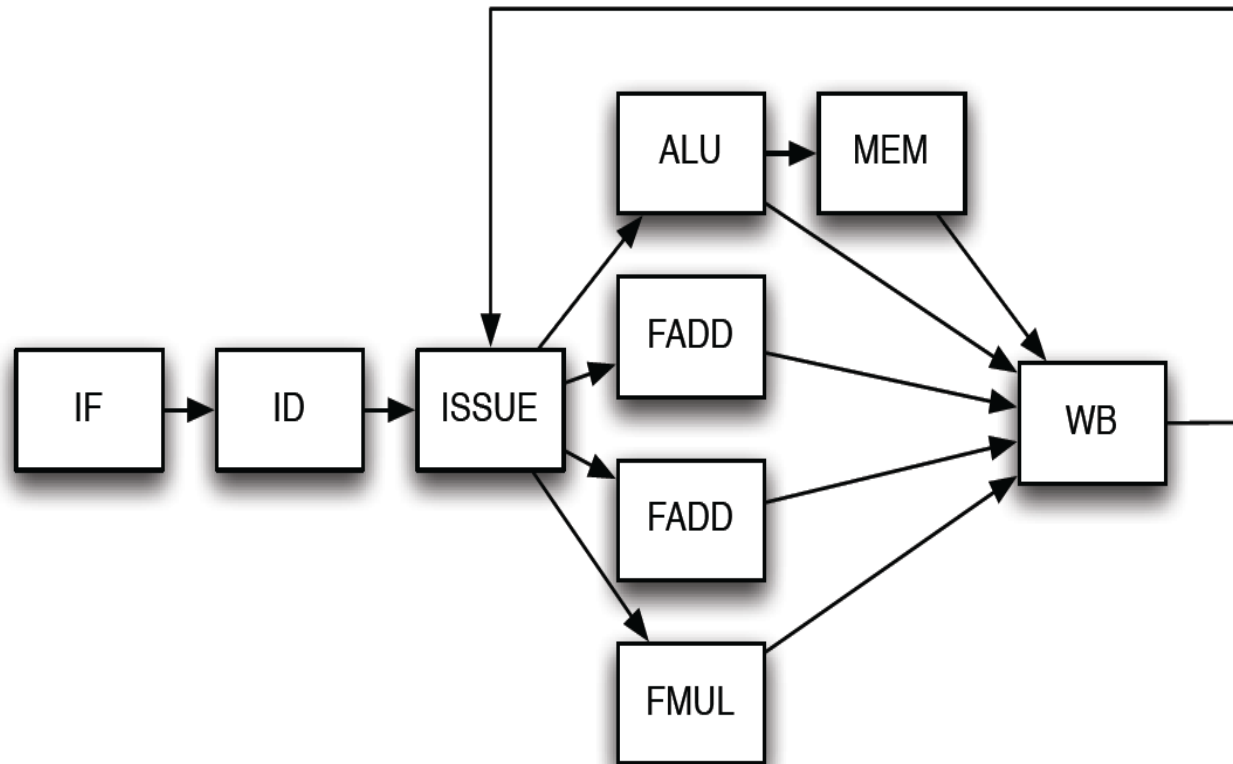
Is the required function unit available?

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 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 1 clock cycle
- Memory operations take 2 clock cycles (includes time in ALU)
- FP ALU operations take 2 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)

Assumptions

- All functional units are pipelined
- **Registers are read in 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 1 clock cycle
- Memory operations take 2 clock cycles (includes time in ALU)
- FP ALU operations take 2 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)

Assumptions

- All functional units are pipelined
- Registers are read in 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 1 clock cycle
- Memory operations take 2 clock cycles (includes time in ALU)
- FP ALU operations take 2 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)

Assumptions

- All functional units are pipelined
- Registers are read in 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 1 clock cycle
- Memory operations take 2 clock cycles (includes time in ALU)
- FP ALU operations take 2 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)

Assumptions

- All functional units are pipelined
- Registers are read in 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 1 clock cycle
- Memory operations take 2 clock cycles (includes time in ALU)
- FP ALU operations take 2 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)

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)

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

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

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

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

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

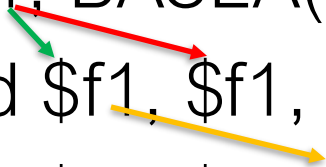
sw.d \$f1, BASEA(\$r1)

RAW f1, WAW f1

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)
```



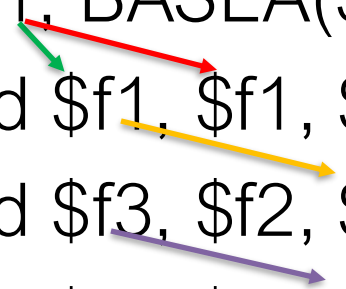
RAW f1, WAW f1

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

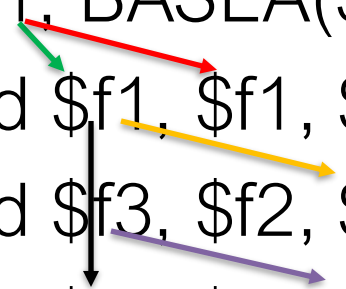
RAW f1

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)
```



RAW f1, WAW f1

RAW f1

RAW f3, 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

RAW f3, 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)
```

RAW f1, WAW f1

RAW f1

RAW f3, WAW f1, WAR f1

RAW f1

Just a reminder... Assumptions

- All functional units are pipelined
- Registers are read in 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 1 clock cycle
- Memory operations take 2 clock cycles (includes time in ALU)
- FP ALU operations take 2 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

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												

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

RAW f3, WAW f1, WAR f1

RAW f1

Assumptions

- All functional units are pipelined
- **Registers are read in 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 1 clock cycle
- Memory operations take 2 clock cycles (includes time in ALU)
- FP ALU operations take 2 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

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									

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)

RAW f1, **WAW f1**

RAW f1

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

RAW f1

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
 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)

RAW f1, **WAW** f1

RAW f1

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

RAW f1

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)

RAW f1, **WAW f1**

RAW f1

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

RAW f1

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			
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 let
the ISSUE stage after cc15

RAW f1, **WAW** f1

RAW f1

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

RAW f1

Getting higher performance...

- In a pipelined machine, actual CPI is derived as:

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

- 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](#)) so as to achieve highest parallelism of execution compatible with available resources.

Dependences

- Determining dependences 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 dependences:
 - Name Dependences
 - Data Dependences (or True Data Dependences)
 - Control Dependences

Name Dependences

- Name dependence occurs when 2 instructions use the same register or memory location (called name), but there is no flow of data between the instructions associated with that name.
- Two type 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.

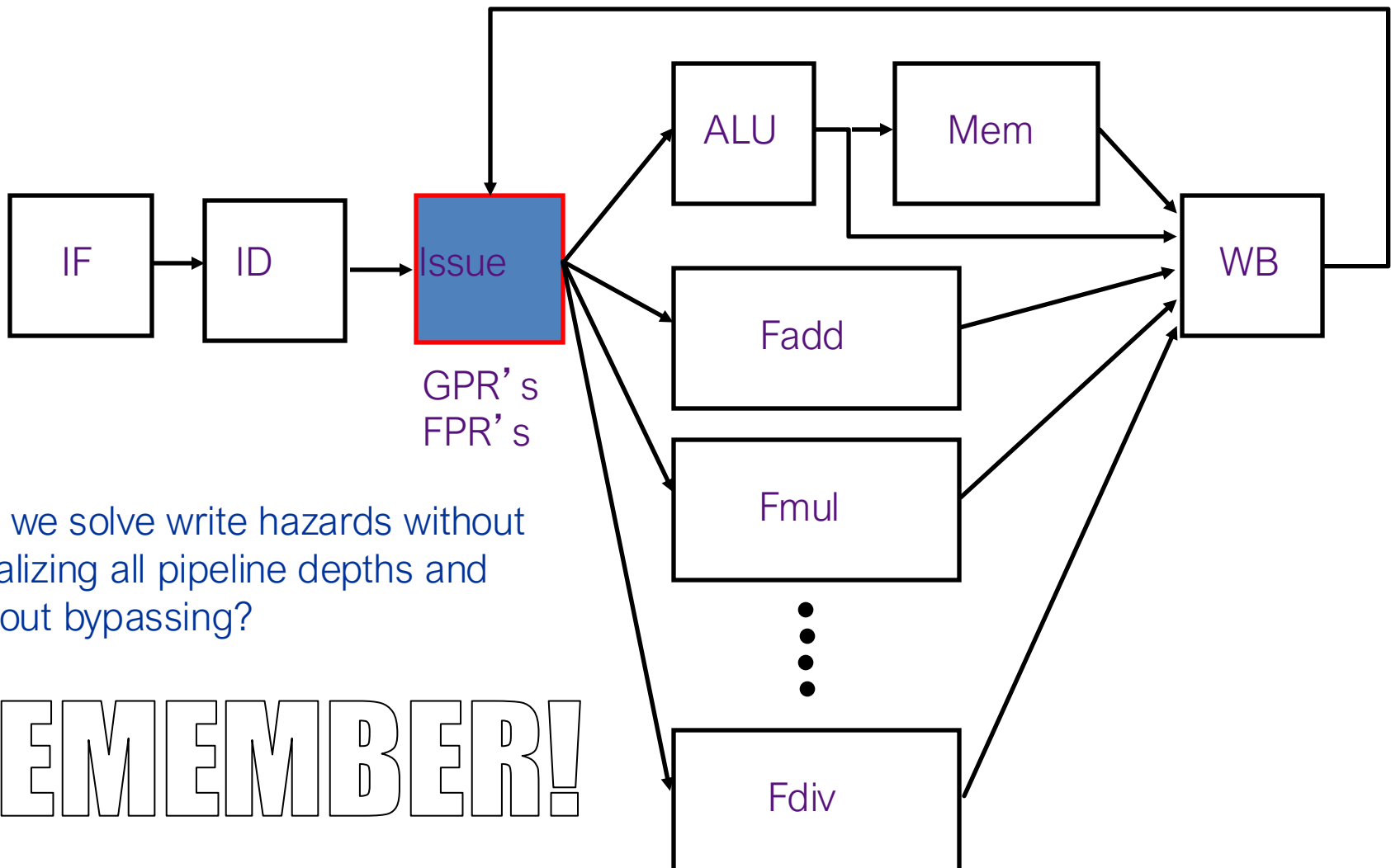
Control Dependences

- 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

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

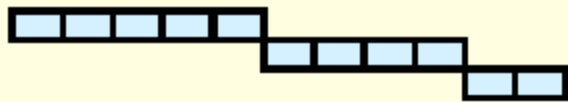
Dynamic Scheduling

- 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 dependences 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 imprecise exception

Dynamic Scheduling

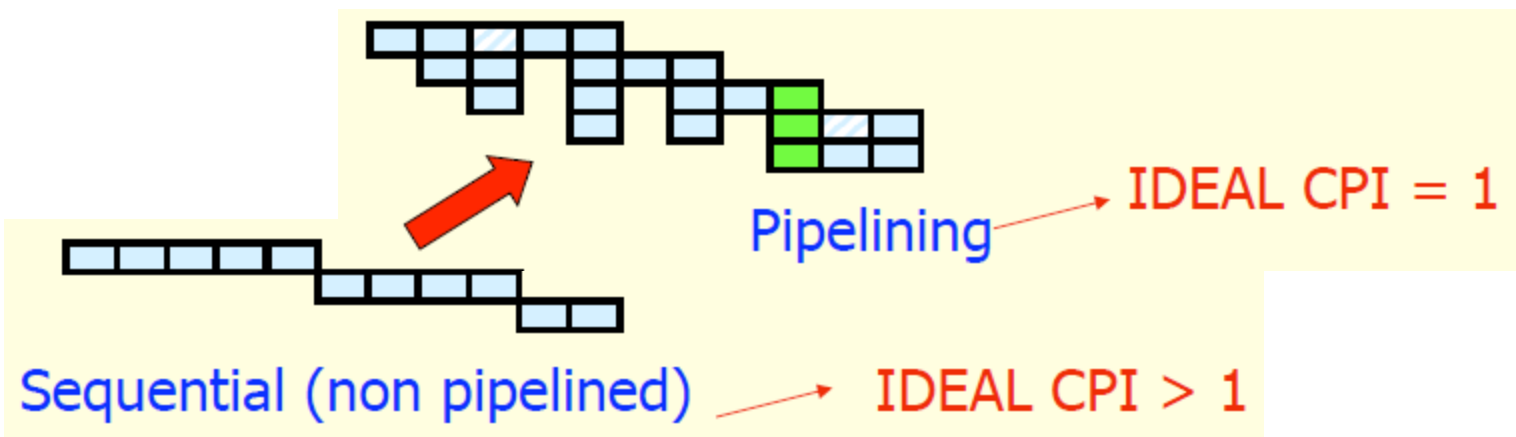
- Basically: Instructions are fetched and issued in program order (in-order-issue)
- Execution begins as soon as operands are available
 - possibly, out of order execution – note: possible even with pipelined scalar architectures.
- Out-of order execution introduces possibility of WAR, WAW data hazards.
- 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

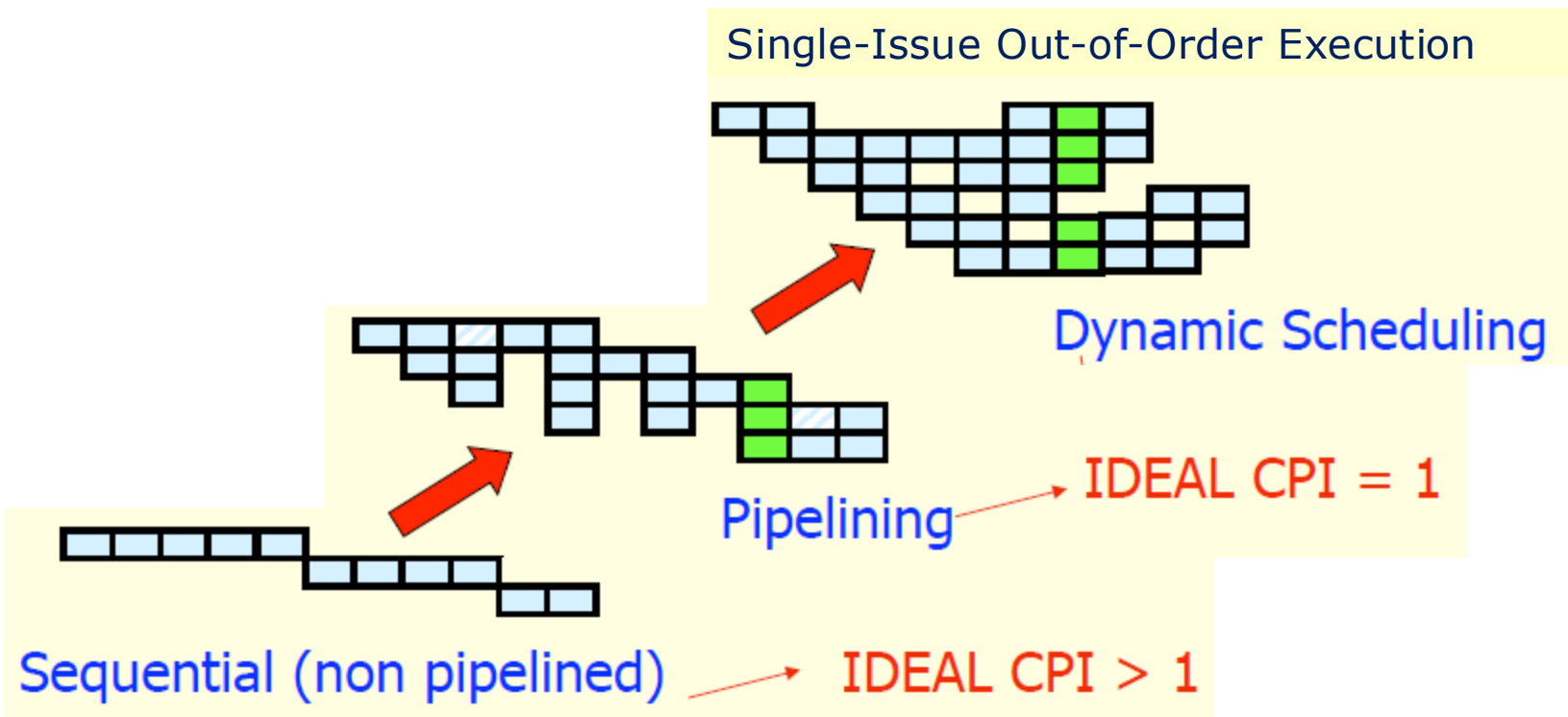


Sequential (non pipelined) \rightarrow IDEAL CPI > 1

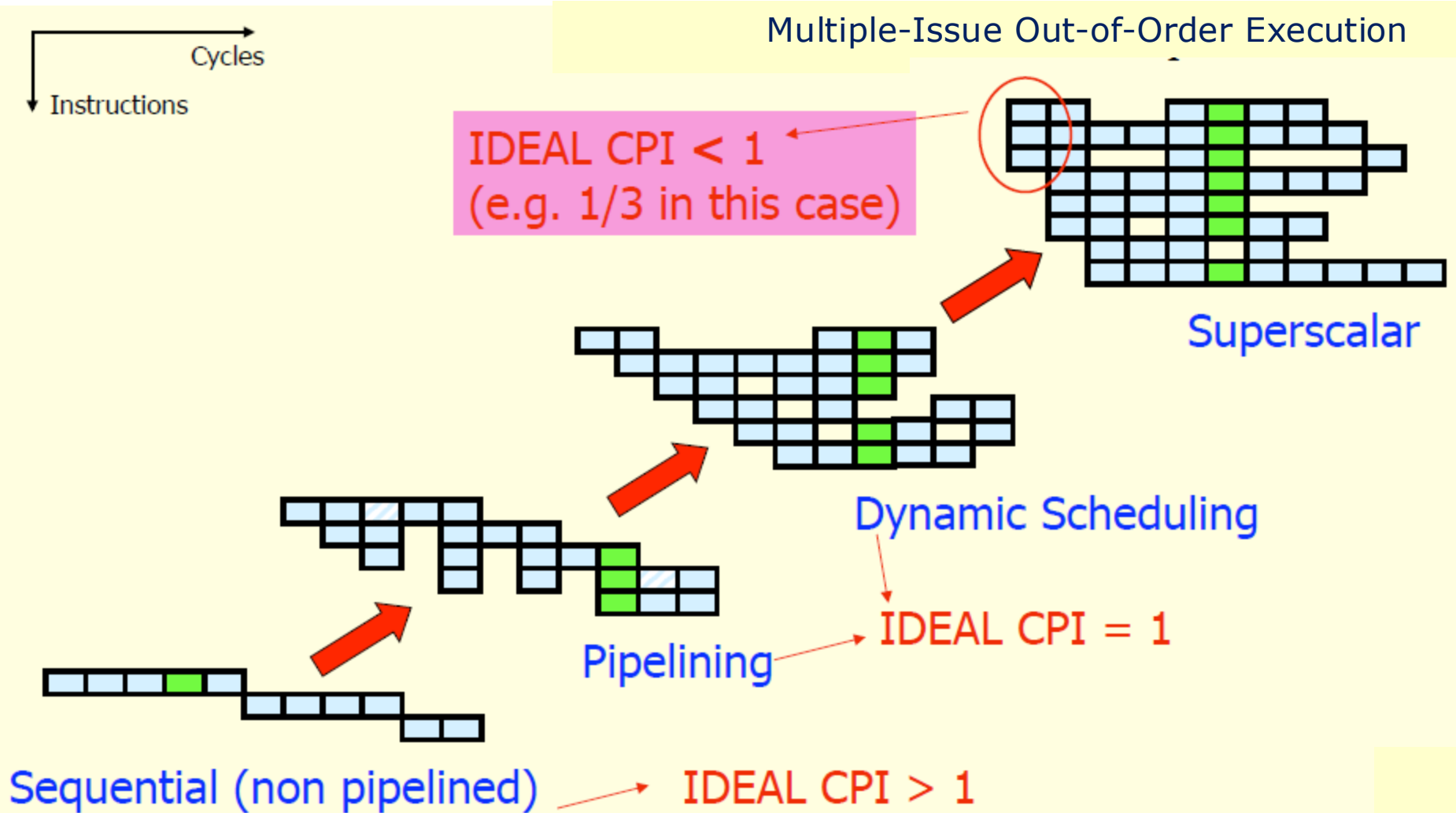
Several steps towards exploiting more ILP



Several steps towards exploiting more ILP



Several steps towards exploiting more ILP



THANK YOU
FOR YOUR ATTENTION

