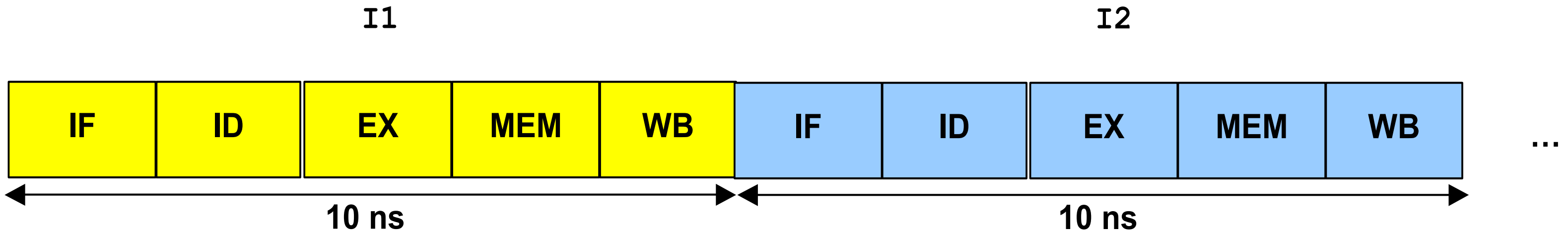
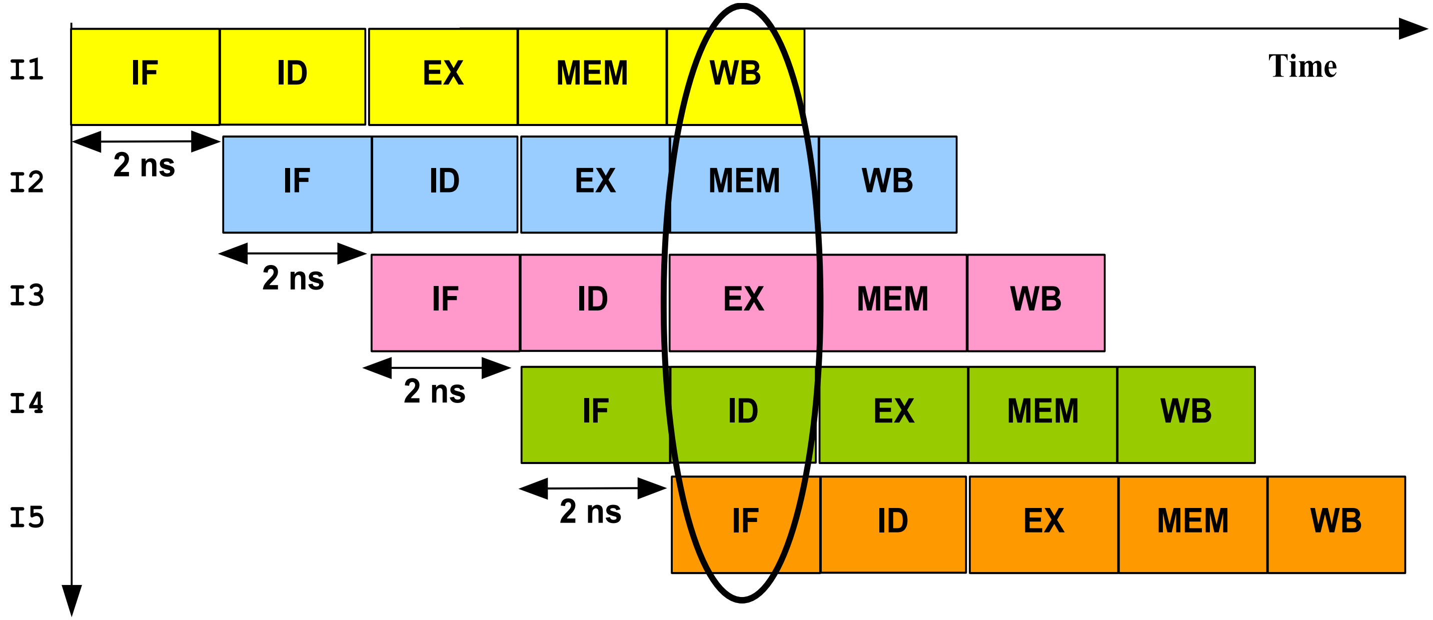
Course on: “Advanced Computer Architectures”

Introduction to Instruction Level Parallelism

Prof. Cristina Silvano
Politecnico di Milano
email: cristina.silvano@polimi.it

Sequential vs. Pipelining Execution





5-stage pipeline => 5 different instructions overlapped

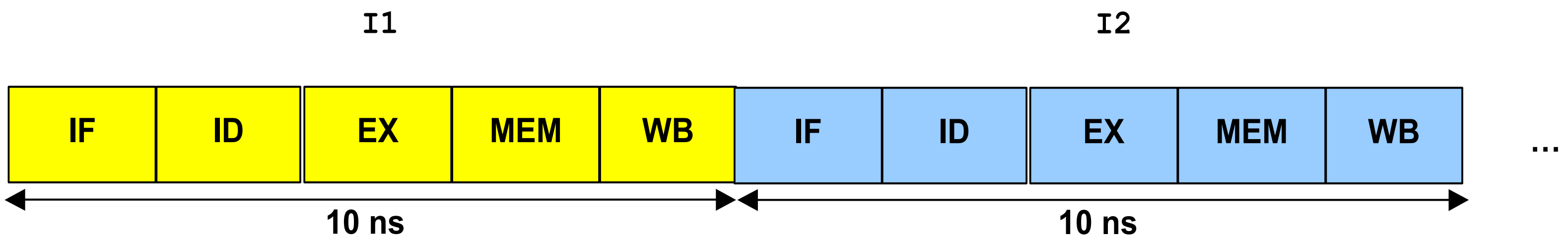
Pipelining Execution

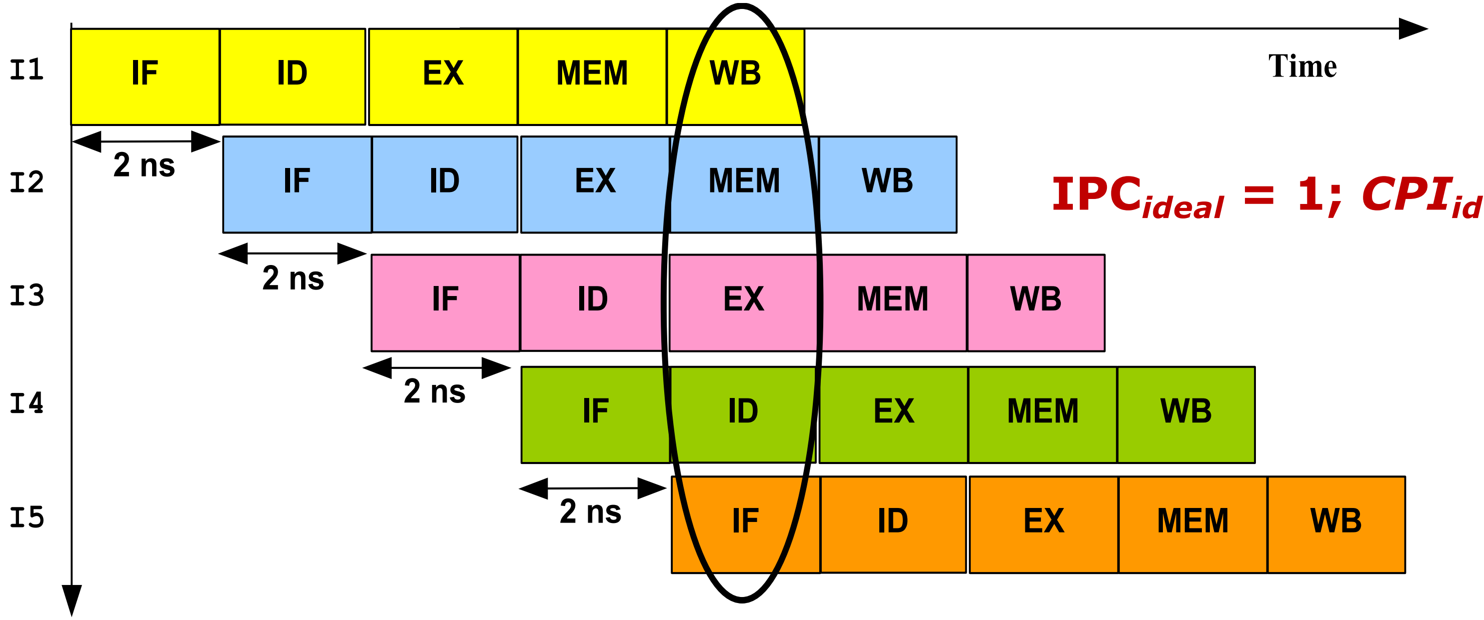
- Pipelining overlaps the execution of instructions
  => it exploits the ***"Instruction Level Parallelism"***

- When exploiting ILP, the goal is to maximize performance in terms of throughput: **Instructions per clock (IPC)**
  … and minimize the **CPI = 1 / IPC**

- *Remind that **pipelining** improves instruction throughput, but not the latency of the single instruction!*

Best case in pipelining execution





$IPC_{ideal} = 1$; $CPI_{ideal} = 1$

The problem of dependences

- Determining **dependences** among instructions is critical to define the amount of parallelism existing in a program.
- If two instructions are **dependent** to each other, they cannot be executed in parallel: they must be executed in order or only partially overlapped.

- There are **three** different types of dependences in a code:
  1. **True Data Dependences:** an instruction j is dependent on a data produced by a previous instruction i;
  2. **Name Dependences:** two instructions use the same register or memory location;
  3. **Control Dependences:** they impose the ordering of instructions

Name Dependences

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

- Name dependences are ***not*** true data dependences, since there is no value (no data flow) being transmitted between the two instructions => this is just a ***register reuse!***

- There are **two types** of name dependences:
  1. **Anti-dependences**
  2. **Output Dependences**

Two types of name dependences

- Let’s consider **Ii** that precedes instruction **Ij** in program order:

  1. **Anti-dependence:**
     When **Ij** writes a register or memory location that instruction **Ii** reads => *it can generate a* **Write After Read – WAR hazard**.
     Original instructions ordering must be preserved to ensure that **Ii** reads the previous value:

     Ii: $r_3 \leftarrow (\mathbf{r_1})$ op (r_2)

     Ij: $\mathbf{r_1} \leftarrow (r_4)$ op (r_5)

  2. **Output Dependence:**
     When **Ii** and **Ij** write the same register or memory location => *it can generate a* **Write After Write – WAW hazard**.
     Original instructions ordering must be preserved to ensure that the value finally written corresponds to **Ij.**

     Ii: $\mathbf{r_3} \leftarrow (r_1)$ op (r_2)

     Ij: $\mathbf{r_3} \leftarrow (r_6)$ op (r_7)

How to solve name dependences?

- Name dependences are ***not*** true data dependences, since there is no value passed (no data flow) between instructions.
- Name dependences are due to **register reuse.**
- ***Register Renaming:*** If the register used could be changed, then the instructions do not conflict anymore.

- Examples:

Ii: $r_3 \leftarrow (\mathbf{r_1})$ op (r_2)

Ij: $\mathbf{r_1} \leftarrow (r_4)$ op (r_5) => $\mathbf{r_4} \leftarrow (r_4)$ op (r_5)

Ii: $\mathbf{r_3} \leftarrow (r_1)$ op (r_2)

Ij: $\mathbf{r_3} \leftarrow (r_6)$ op (r_7) => $\mathbf{r_4} \leftarrow (r_6)$ op (r_7)

Register Renaming

- **Register Renaming** can be more easily done, if there are enough registers available in the ISA.
- **Register Renaming** can be done either statically by the compiler or dynamically by the hardware.
- Dependences trough memory locations are more difficult to detect **("memory disambiguation"** problem), since two addresses may refer to the same location but can look different.

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 architecture.
    - **RAW hazards** correspond to **true data dependences**
    - **WAR hazards** correspond to **anti-dependences**
    - **WAW hazards** correspond to **output dependences**
- **Dependences** are a property of the program, while **hazards** are a property of the pipeline architecture.

Summary on Data Dependences & Hazards

Consider executing a sequence of

$$r_k \leftarrow (r_i)\ \text{op}\ (r_j)$$

True Data-dependence

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

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

Read-after-Write
(RAW) hazard

Anti-dependence

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

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

Write-after-Read
(WAR) hazard

Output-dependence

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

$\mathbf{r_3} \leftarrow (r_6)\ \text{op}\ (r_7)$

Write-after-Write
(WAW) hazard

Summary of Control Dependences

- A **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 (as seen when we've studied scheduling techniques to fill in the branch delay slot).

Program Properties

- **Two properties** are critical to preserve program correctness (and normally preserved by maintaining both data and control dependences during scheduling):

1. **Data flow:** Actual flow of data values among instructions that produces the correct results and consumes them.

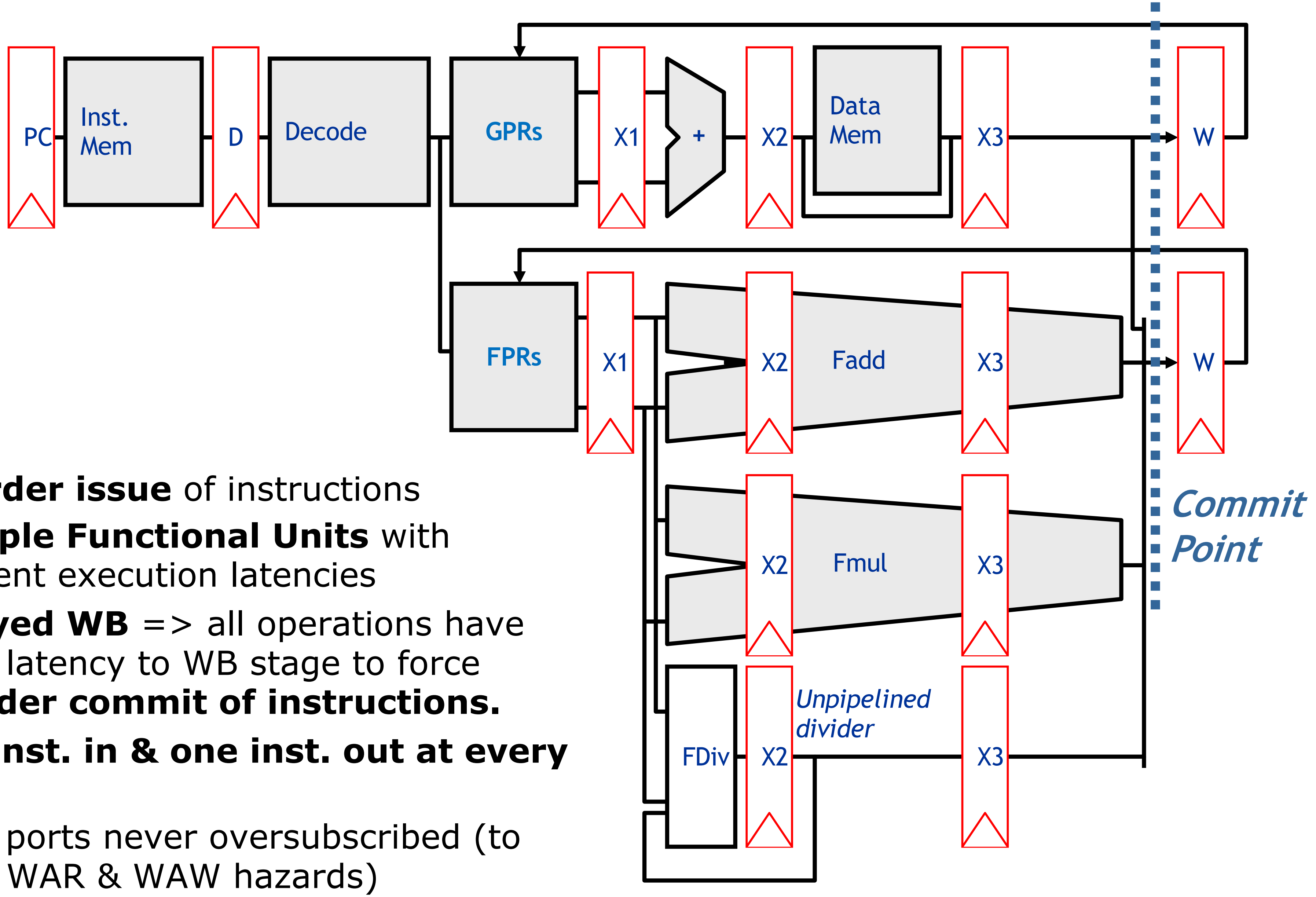
2. **Exception behavior:** Preserving exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.

MULTI-CYCLE PIPELINING & DYNAMIC SCHEDULING

Multi-cycle Pipeline: Basic Assumptions

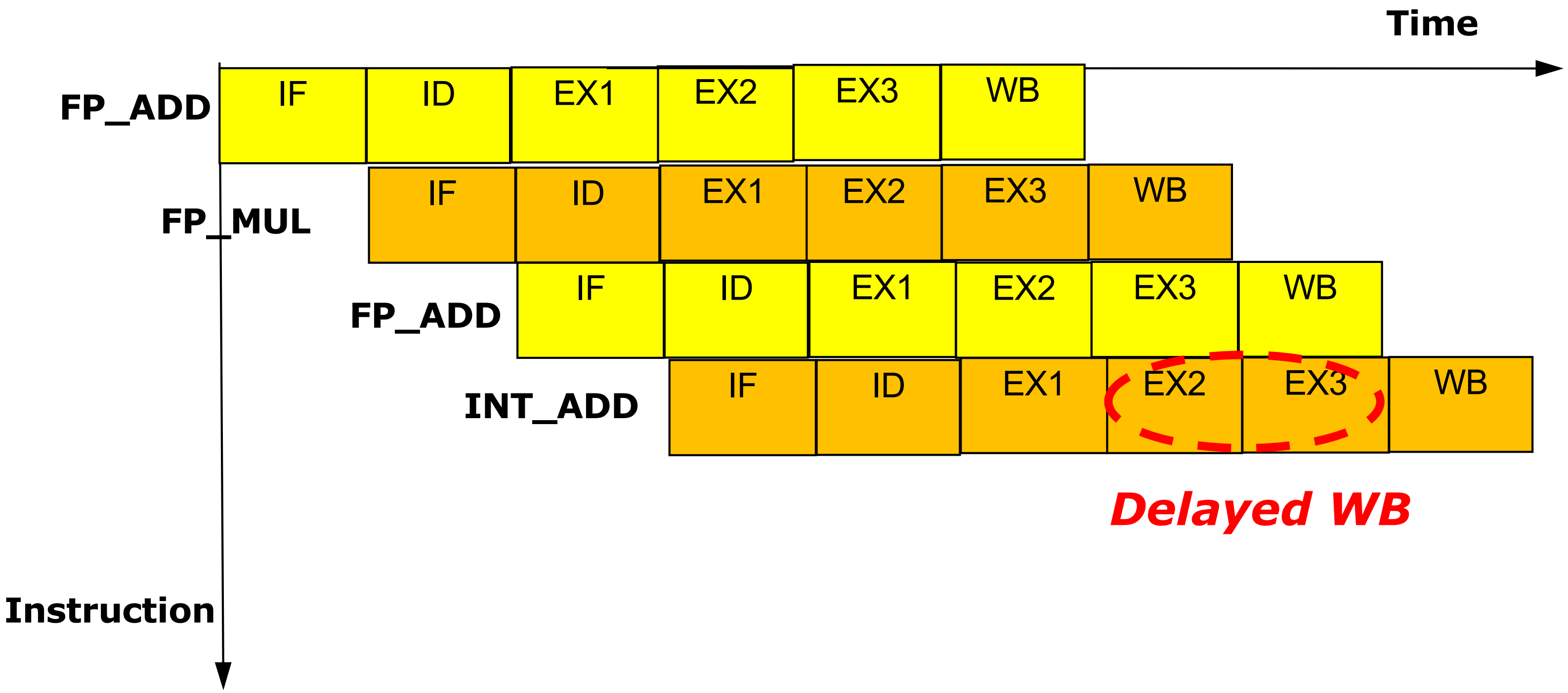
- We consider ***single-issue*** processors (one instruction issued per clock cycle).
- Instructions are then ***issued in-order.***
- **Execution stage** might require ***multiple cycles latency,*** depending on the operation type (i.e., multiply operations are typically longer than add/sub operations)
- **Memory stages** might require ***multiple cycles*** access time due to instruction and data cache misses.

Multi-cycle In-Order Pipeline



- **In-order issue** of instructions
- **Multiple Functional Units** with different execution latencies
- **Delayed WB** => all operations have same latency to WB stage to force **in-order commit of instructions.**
- **One inst. in & one inst. out at every cycle**
- Write ports never oversubscribed (to avoid WAR & WAW hazards)

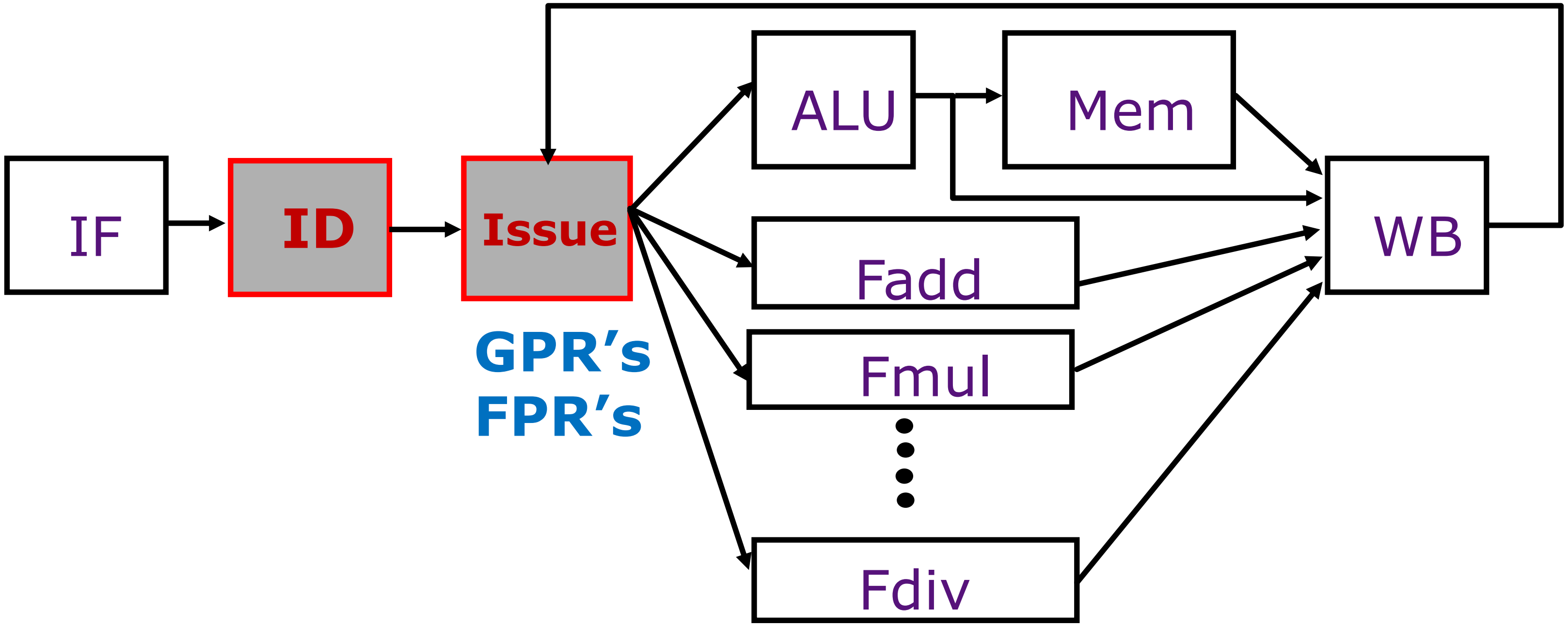
Multi-cycle In-order Pipeline



FP_ADD	IF	ID	EX1	EX2	EX3	WB			
FP_MUL		IF	ID	EX1	EX2	EX3	WB		
FP_ADD			IF	ID	EX1	EX2	EX3	WB	
INT_ADD				IF	ID	EX1	EX2	EX3	WB

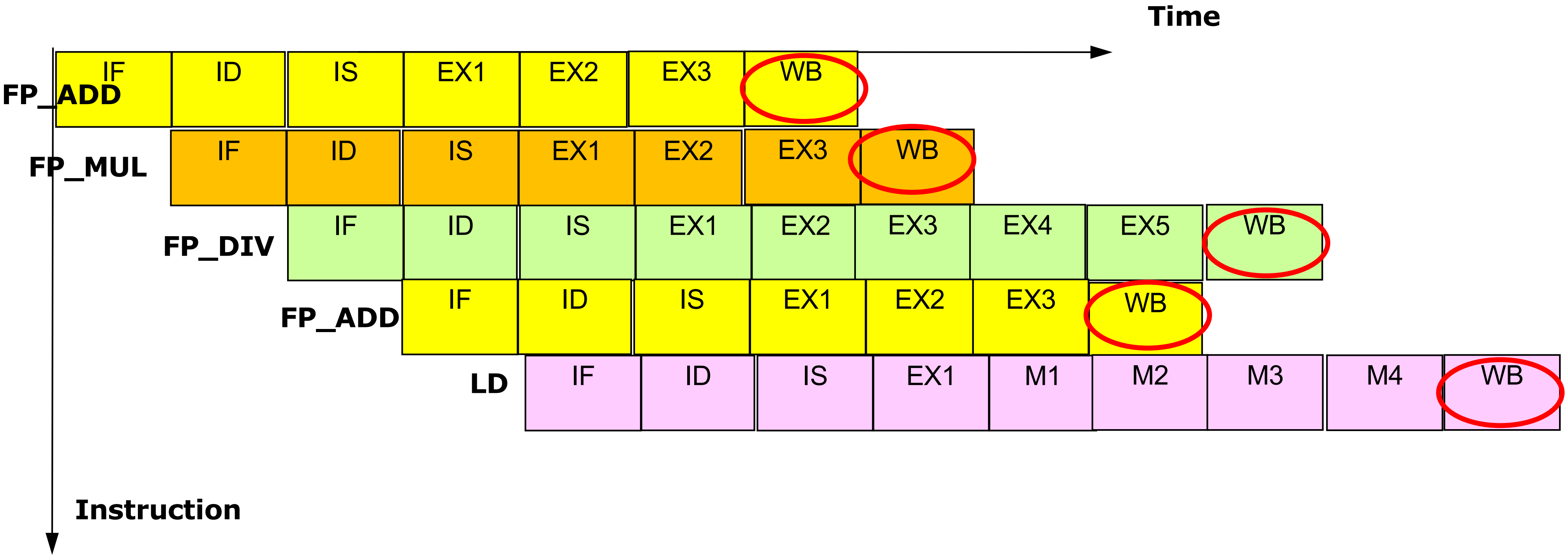
- **In-order issue & In-order commit** of instructions.
- This avoids the generation of **WAR** & **WAW hazards** and preserves the **precise exception model.**

Multi-cycle Out-of-order Pipeline



- ID stage split in **2 stages**: Instr. Decode (**ID**) & Register Read (**Issue**);
- **Multiple** functional units with variable latency;
- Long latency **multi-cycle floating-point instructions;**
- Memory systems with variable access time: **Multi-cycle memory accesses** due to data cache misses (unpredictable statically);
- **No more commit point** => **Out-of-order commit:** Need to check for WAR & WAW hazards and imprecise exception

Multi-cycle Out-of-order Pipeline



FP_ADD	IF	ID	IS	EX1	EX2	EX3	WB						
FP_MUL		IF	ID	IS	EX1	EX2	EX3	WB					
FP_DIV			IF	ID	IS	EX1	EX2	EX3	EX4	EX5	WB		
FP_ADD				IF	ID	IS	EX1	EX2	EX3	WB			
LD					IF	ID	IS	EX1	M1	M2	M3	M4	WB

- **In-order issue** of instructions
- **Out-of-order execution & out-of-order commit** of instructions
- Need to check the generation of **WAR** & **WAW** hazards and **imprecise exceptions.**

Another Key Idea: Dynamic Scheduling

- **Problem:** Hazards due to true data dependences that cannot be solved by forwarding cause **stalls** of the pipeline => no new instructions are fetched nor issued even if they are not data dependent;

- **Solution: Allow data independent instructions behind a stall to proceed**
  - HW manages dynamically the instruction execution to reduce stalls: an instruction execution begins as soon as their operands are available.
- **This generates out-of-order execution and completion (commit)**

- *Dynamic Scheduling can be applied to a single-issue scalar pipeline, but we'll see that it can be applied also to multiple-issue pipelines…*

Example of dynamic scheduling

```
DIVD  F0,F2,F4        # exec. takes many cycles
ADDD  F10,F0,F8       # RAW F0
SUBD  F12,F8,F14
```

- **ADDD** stalls for RAW hazard on **F0** (waiting many clock cycles for DIVD commit).
- **SUBD** would stall even if not data dependent on anything in the pipeline.

- **BASIC IDEA: to enable SUBD to proceed**
  => this generates *out-of-order execution*

Example of dynamic scheduling (cont.)

	CK1	CK2	CK3	CK4	CK5	CK6	CK7	CK8	
`DIVD $f0,$f2,$f4`	IF	ID	DIV1	DIV2	DIV3	DIV4	WB		
`ADDD $f10,$f0,$f8`		IF	●	●	●	●	●	ID	AD1
`SUBD $f12,$f8,$f14`			IF	ID	SUB1	SUB2	SUB3	WB	

Imprecise exceptions

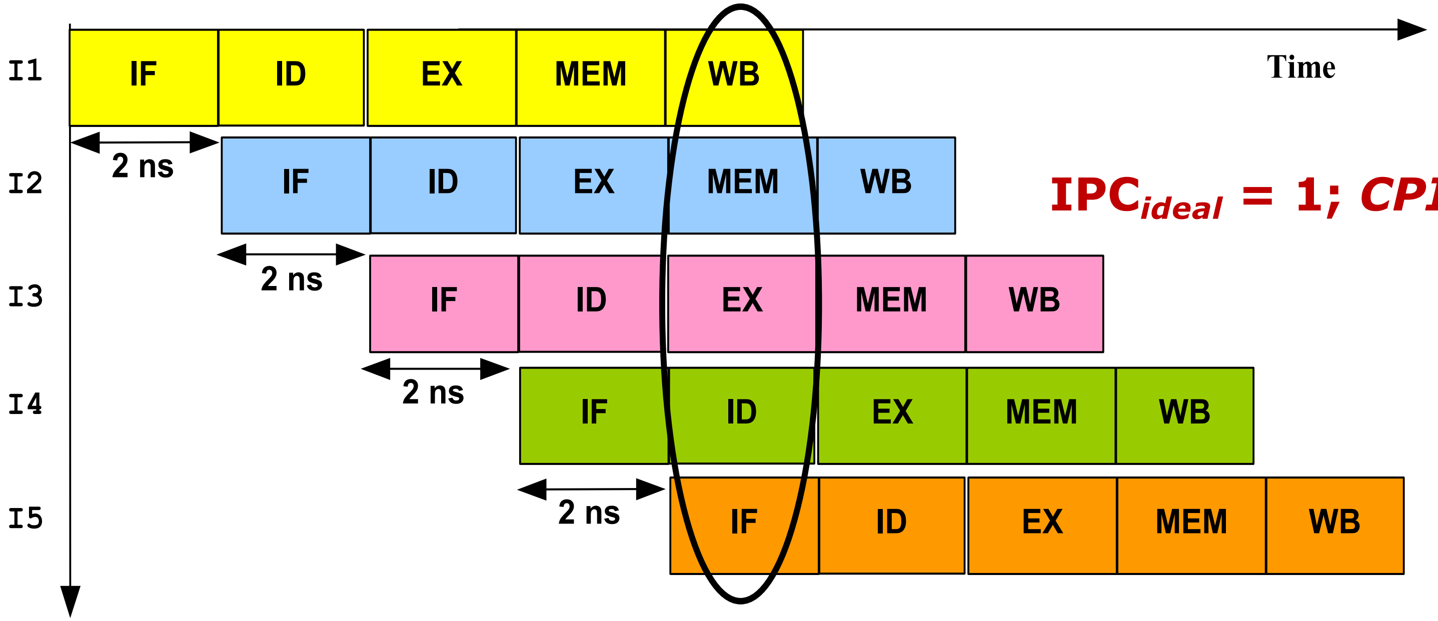
- An exception is ***imprecise*** if the processor state when an exception is raised does not look exactly as if the instructions were executed in-order.

- Imprecise exceptions can occur in **out-of-order** because:
  - The pipeline may have ***already*** completed instructions that are ***later*** in program order than the instruction causing the exception
  - The pipeline may have ***not yet*** completed some instructions that are ***earlier*** in program order than the instruction causing the exception

- Imprecise exception make it difficult to restart execution after handling

MULTIPLE-ISSUE PROCESSORS

ILP: Scalar Pipeline (baseline)



IPC_{ideal} = 1; *CPI_{ideal} = 1*

5-stage pipeline => 5 different instructions overlapped

Getting higher performance...

- Scalar pipeline limited to $\boldsymbol{CPI_{ideal} = 1}$
  - *It can never fetch and execute more than **one** instruction per clock: **single-issue***
  - *It can be even worst due **stalls** added to solve hazards*
- Goal: To reach higher performance => ***more parallelism*** must be extracted from the program. In other words...***multiple-issue***
  - *It means fetching and executing more than **one** instruction per clock*
- Instruction dependences must be detected and solved: instructions must be *re-ordered (**scheduled**)* to achieve the highest ILP given the available resources.

Getting higher performance...

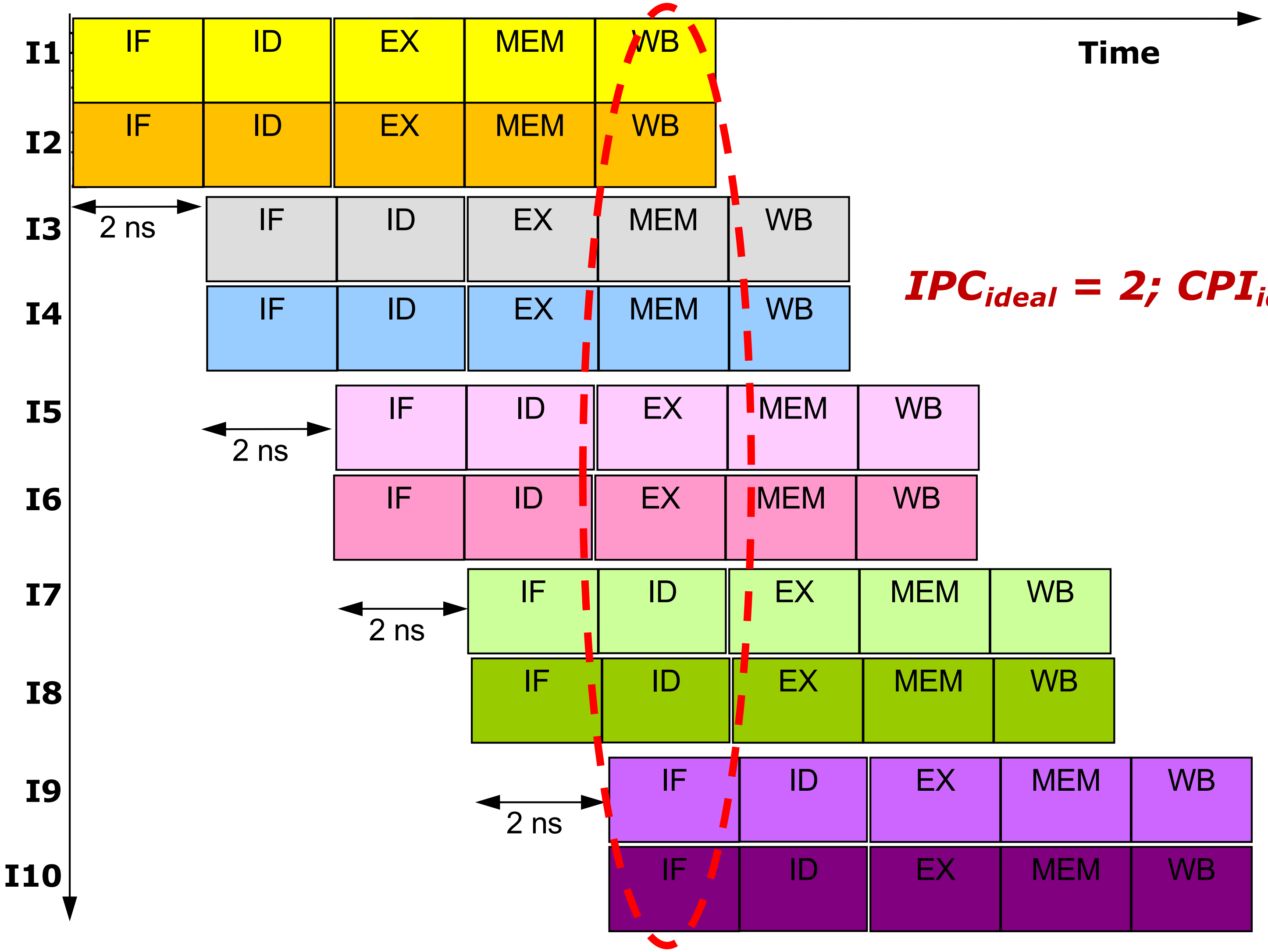
- In a **multiple-issue pipelined** processor, we want to fetch and execute more than 1 instruction per clock

$$\boldsymbol{IPC_{ideal} > 1 \Rightarrow CPI_{ideal} < 1}$$

- Example: ***Dual-issue pipeline processor***
  Ideal case: max throughput would be to complete 2 Instructions Per Clock:

$$\boldsymbol{IPC_{ideal} = 2 \Rightarrow CPI_{ideal} = 0.5}$$

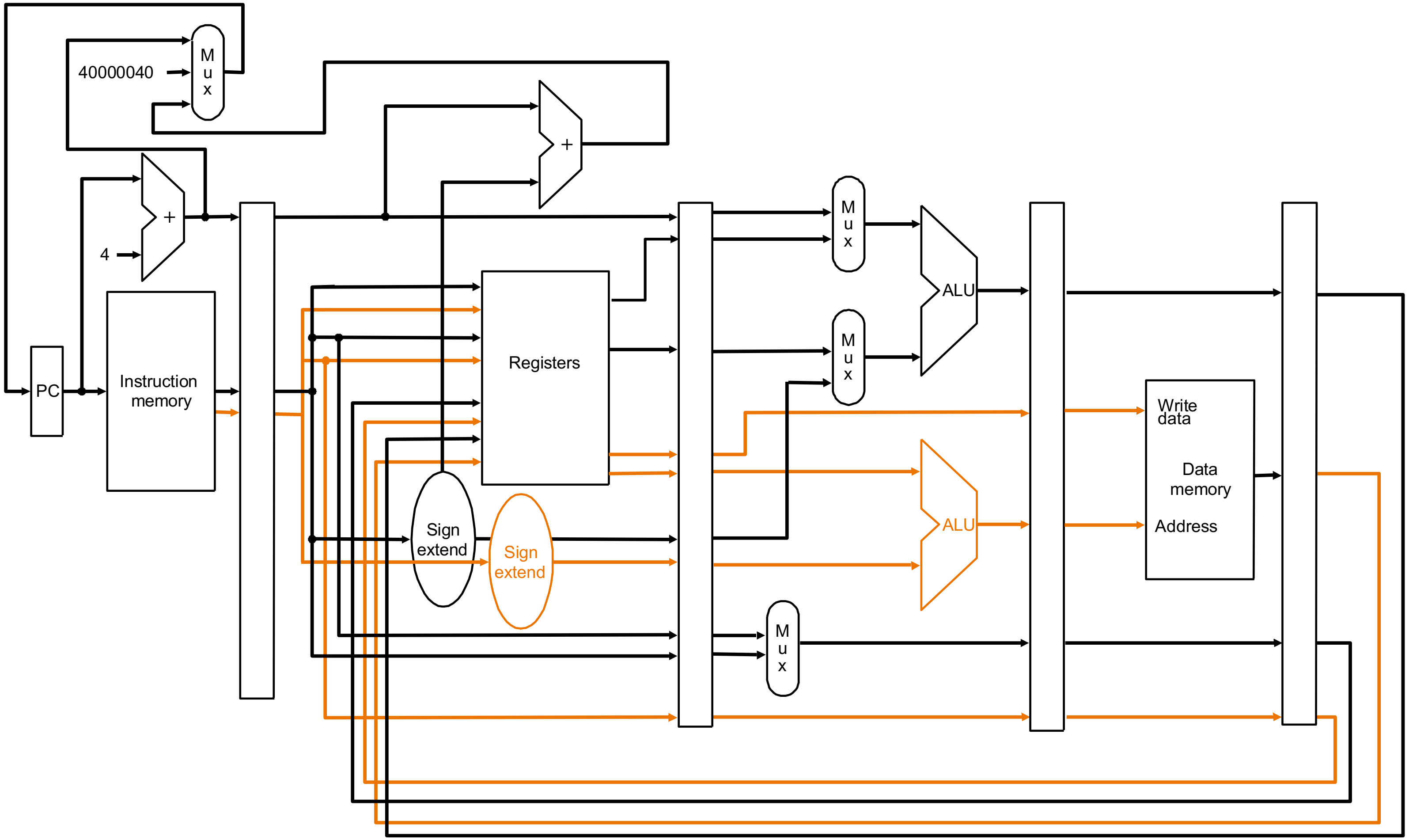
ILP: Dual-Issue Pipeline



IPC_{ideal} = 2; CPI_{ideal} = 0.5

2-issue 5-stage pipeline => 2 x 5 different instructions overlapped

Dual-issue MIPS Pipeline Architecture



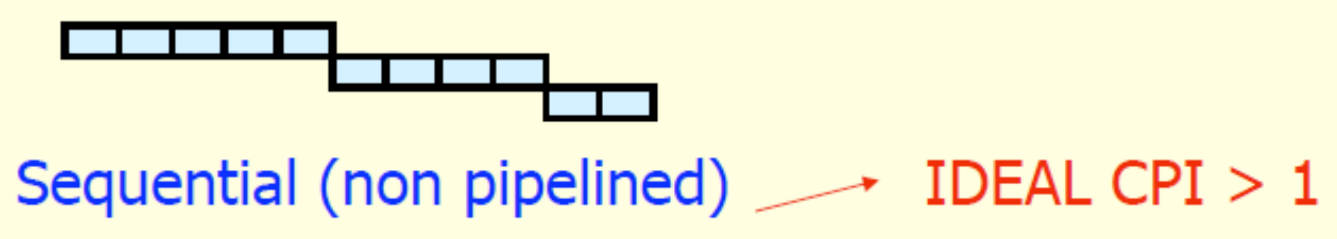
2-instructions issued per clock:

- 1 ALU or BR instruction
- 1 load/store instruction

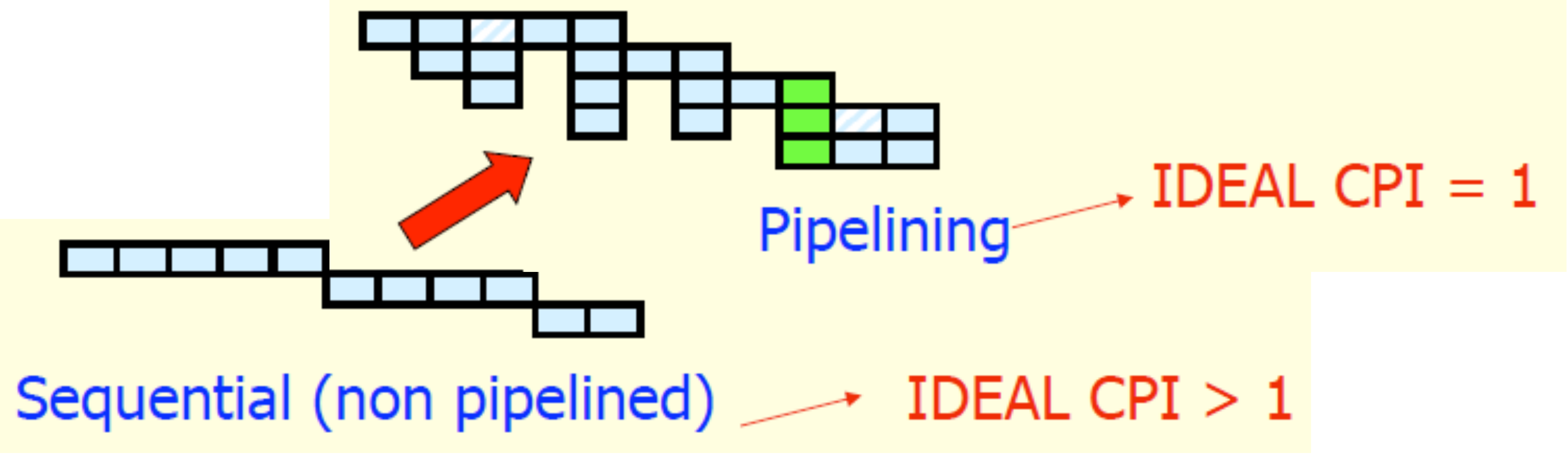
Multi-port RF: 4 read ports & 2 write ports

Recap

- *How to exploit the potential parallelism of the execution among independent instructions?*



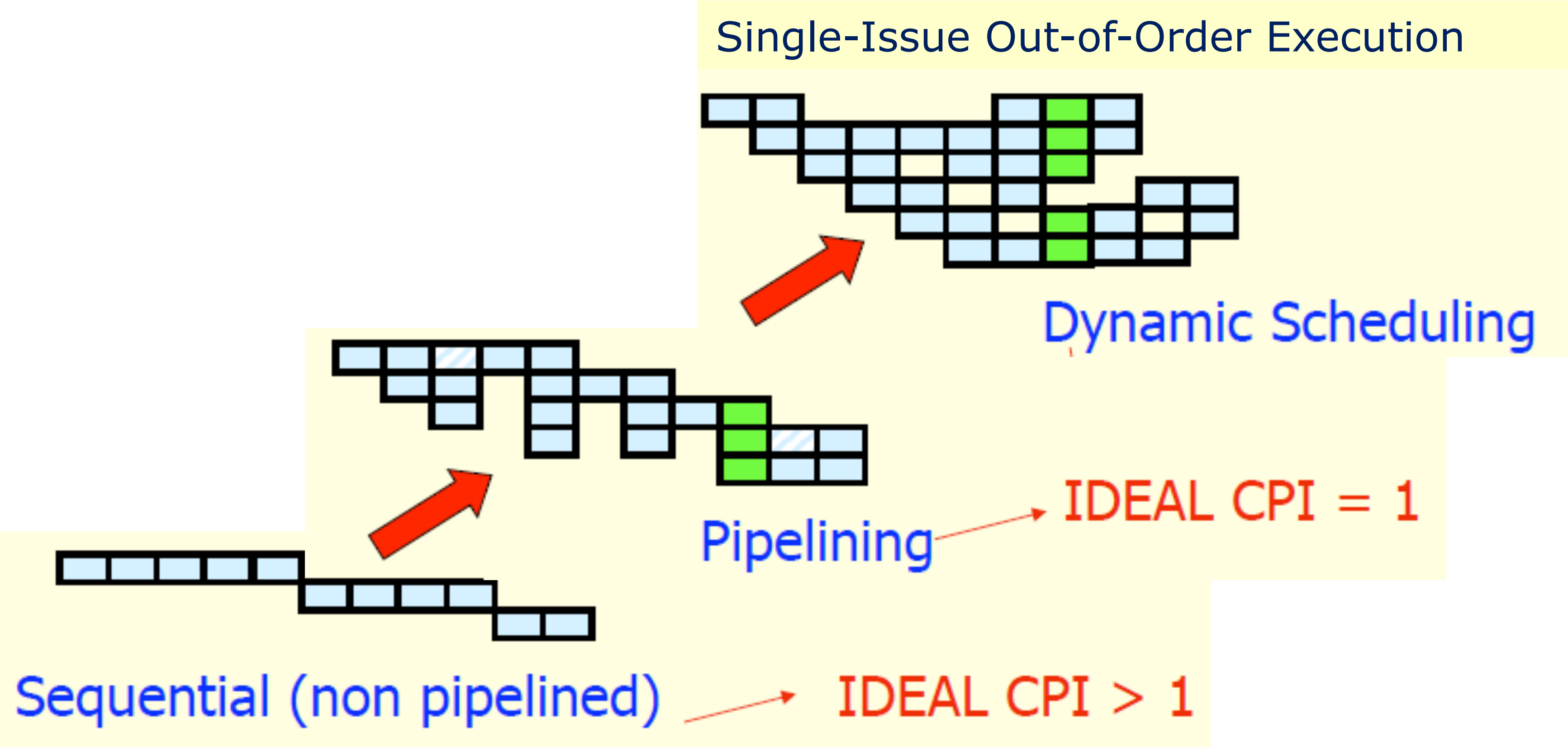
Several steps towards exploiting more ILP



Pipelining → IDEAL CPI = 1

Sequential (non pipelined) → IDEAL CPI > 1

Several steps towards exploiting more ILP

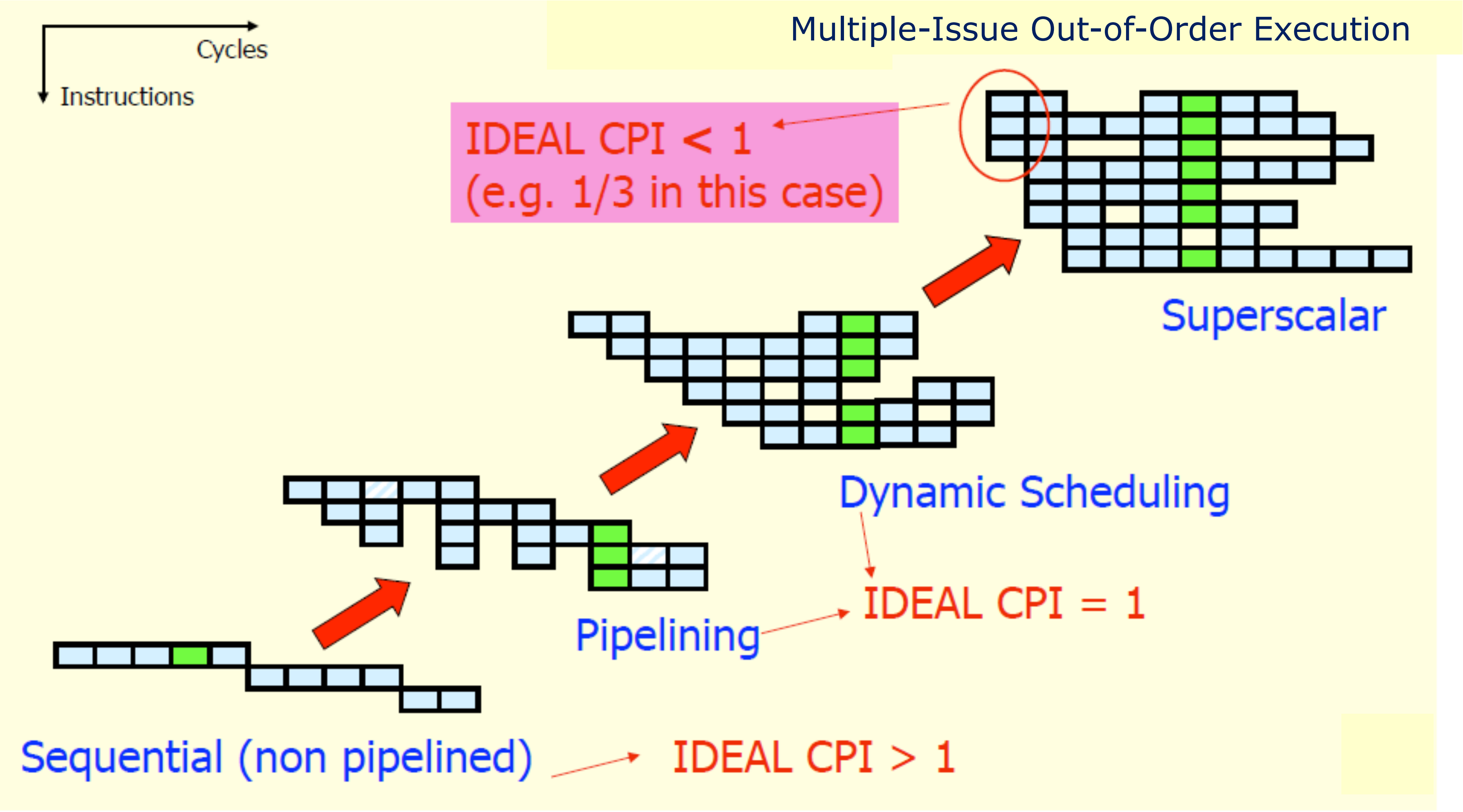

Single-Issue Out-of-Order Execution
Dynamic Scheduling
Pipelining
IDEAL CPI = 1
Sequential (non pipelined)
IDEAL CPI > 1

Several steps towards exploiting more ILP

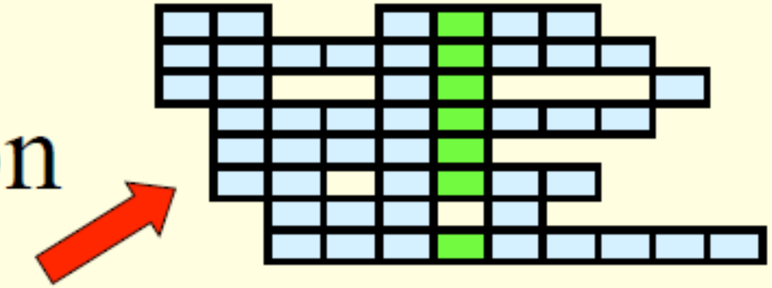
- **Dynamic Scheduling** can be applied to single-issue scalar processors but also to ***multi-issue processors***

  => *Dual-issue MIPS processor and more in general* ***superscalar*** *processors*

Several steps towards exploiting more ILP



Superscalar Execution



- Combining two main ideas:
  1. Why not fetching and beginning execution of more than one instruction per cycles?

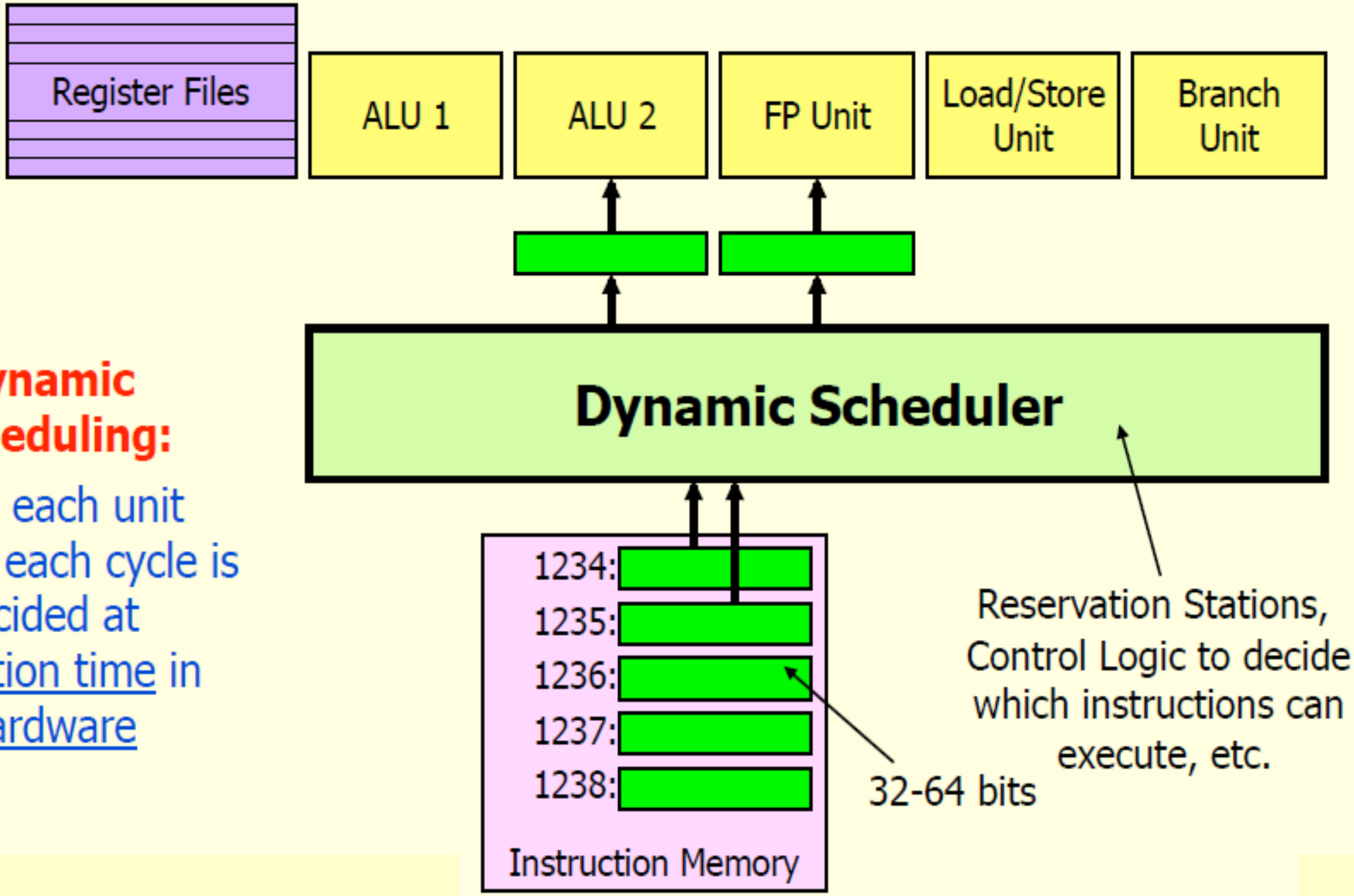
     ✓***Multiple-issues***
  2. Why not to check and manage at runtime data and control dependencies?

     ✓***Dynamic scheduling:*** Hardware reorder instructions execution so as to reduce stalls, maintaining data flow and exception behaviour.

Dynamic Scheduler

Dynamic Scheduling:

What each unit does in each cycle is decided at execution time in hardware



Recap on Dynamic Scheduling

- Instructions are *fetched* and *issued in program order* **(in-order-issue)**
- Execution begins as soon as **RAW hazards** are solved and operands are available

  => *possible generation of* ***out of order execution*** – note: it is possible even with pipelined scalar architectures due to multi-cycle latencies.
- Out-of order execution introduces possible **WAR and WAW data hazards.**
- Out-of order execution generates ***out of order commit.***

Recap on superscalar processors: Pros and Cons

- Main advantage:
  - Very high performance: ***CPI_{ideal} = 1 / issue-width***

- Disadvantages
  - Very complex logic and area cost to check and manage dependencies at runtime, i.e. to decide which instructions can be issued at every clock cycle;
  - Cycle time limited by scheduling logic (dispatcher and associated dependency checking logic)
  - It does not scale well: almost impractical to make issue-width greater than 4 …

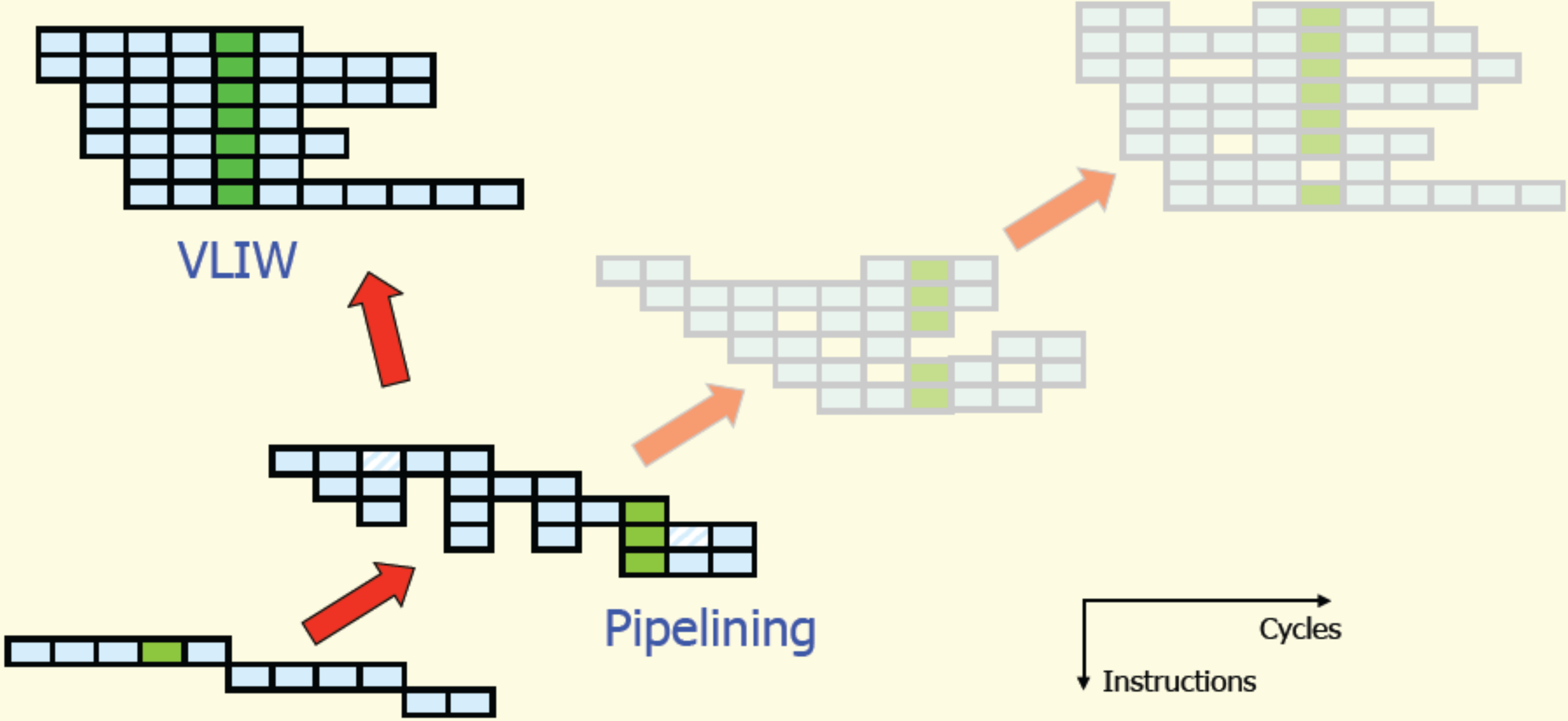
Why not using static scheduling?

- Static detection and resolution of dependences
  ⇨ **static scheduling** done by the **compiler**
  ⇨ instruction dependences are avoided by code reordering at compile time.

- Output of the compiler: instructions are reordered into dependency-free parallel instructions, otherwise NOPs are introduced in the code.

- Typical architecture: **VLIW (Very Long Instruction Word)** processors expect **dependency-free code** generated at static time by the compiler.

Very Long Instruction Word: An Alternative Way of Extracting ILP



VLIW

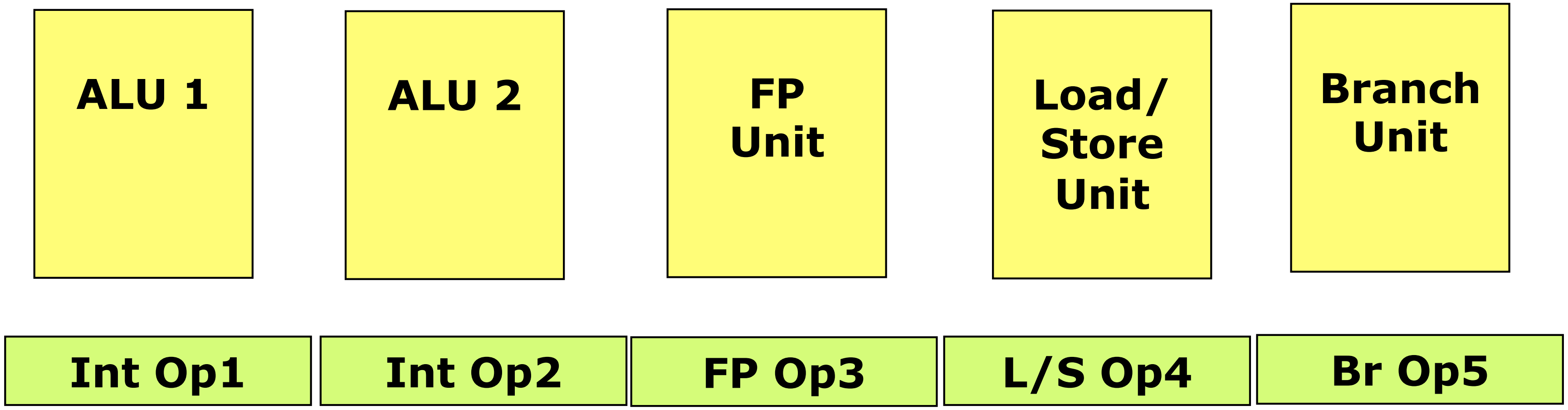
Pipelining

Cycles

Instructions

Sequential (no pipelining)

VLIW: Very Long Instruction Word



ALU 1

ALU 2

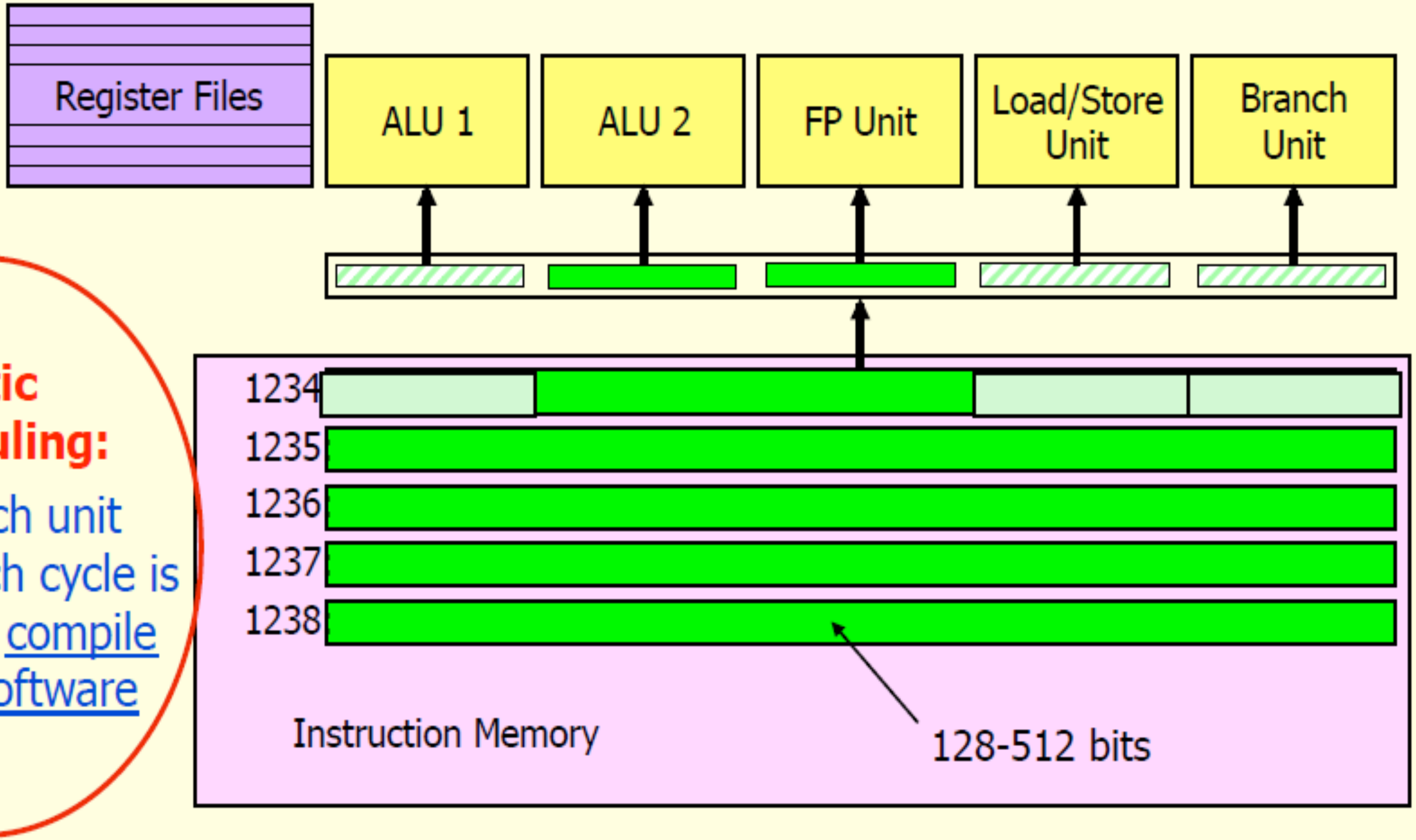
FP Unit

Load/ Store Unit

Branch Unit

Int Op1	Int Op2	FP Op3	L/S Op4	Br Op5

(Statically Scheduled) Very Long Instruction Word Processor (VLIW)



Static Scheduling:

What each unit does in each cycle is decided at compile time in software

Main Limits of Static Scheduling

- Unpredictable branch behavior: the code parallelization is limited to basic blocks.
- Unpredictable cache behavior: Variable memory latency for hits/misses.
- Complexity of compiler technology: The compiler needs to find a lot of parallelism in order to keep the multiple functional units of the processors busy
- Code size explosion due to insertion of NOPs
- Low code portability and binary code incompatibility: Consequence of the larger exposure of the microarchitecture (= implementation details) at the compiler in the generated code
- Low performance portability

Recap on Static vs Dynamic Scheduling

- Two strategies to support ILP:
  - **Static Scheduling:** Rely on the compiler for identifying potential parallelism
  - **Dynamic Scheduling:** Depend on the hardware to locate parallelism

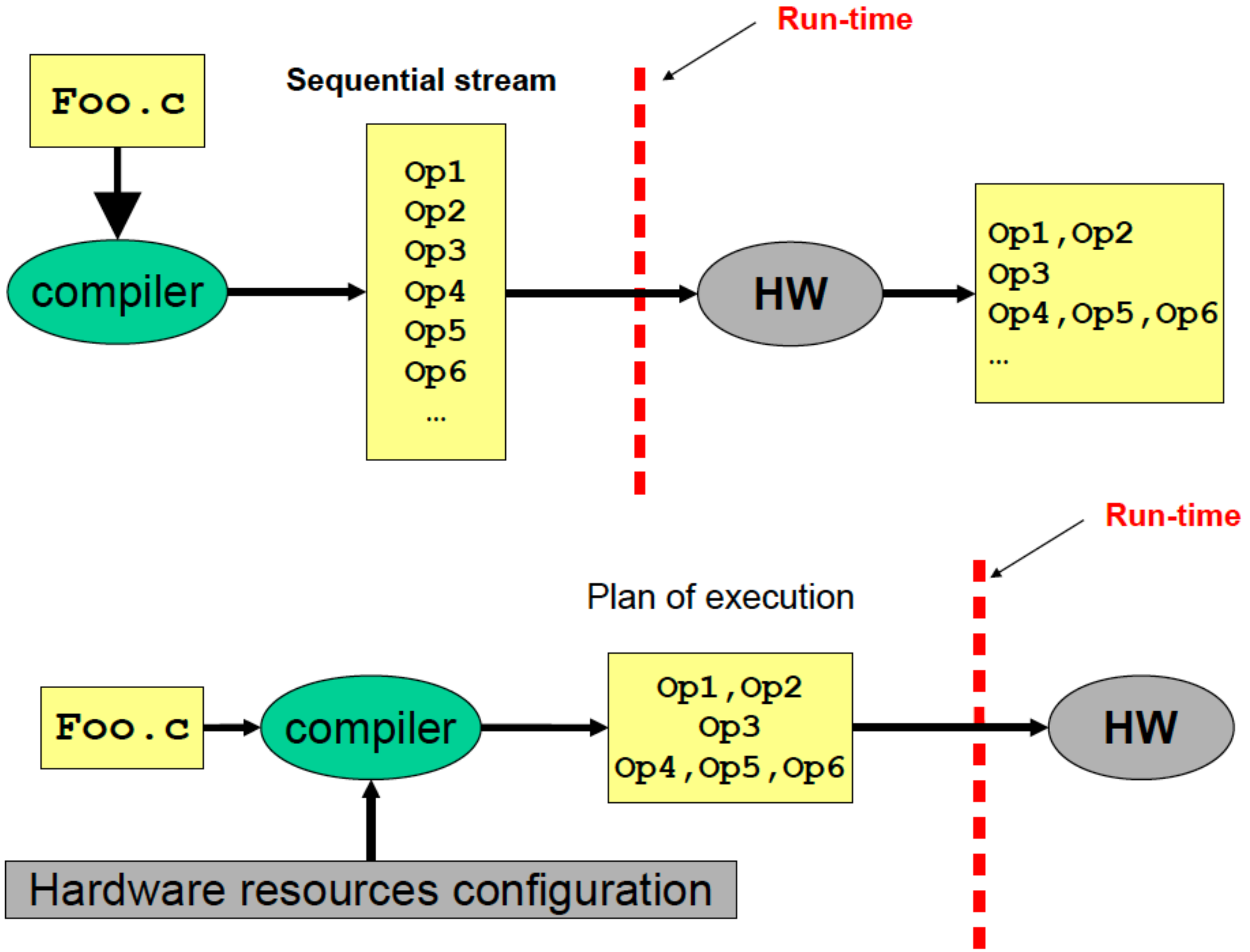
ILP: Static vs Dynamic Scheduling

- Instruction Scheduling: Deciding ***when*** and ***where*** to execute an instruction
  - *i.e.* in which cycle and in which functional unit

  1. For a superscalar processor it is decided **at runtime,** by the hardware logic.
  2. For a VLIW processor it is decided **at compile time,** by the compiler, and therefore by a SW program
     - Suitable for embedded processors: Simpler HW design (no dynamic scheduler), smaller area and power consumption … and cheap

ILP: Static vs Dynamic Scheduling

- *Who decides about the ILP?*



Issue-width limited in practice

- The **issue width** is the number of instructions that can be issued in a single cycle by a **multiple issue processor**
- When superscalars were invented, 2- and rapidly 4-issue width processors were created.
- Due to the intrinsic level of parallelism of a program, it is hard to decide which 8, or 16, instructions can execute every cycle!

How to overcome the ILP limitations?

Solution: Introduce more levels of parallelism

- Multi-threading (Hypethreading)
- Multi-processing and multi-cores
- Data-level Parallelism: Vector Processors and GPUs
- Heterogeneity: Host processor and co-processor