

Course on: “Advanced Computer Architectures”

Instruction Level Parallelism

Part I - Introduction



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

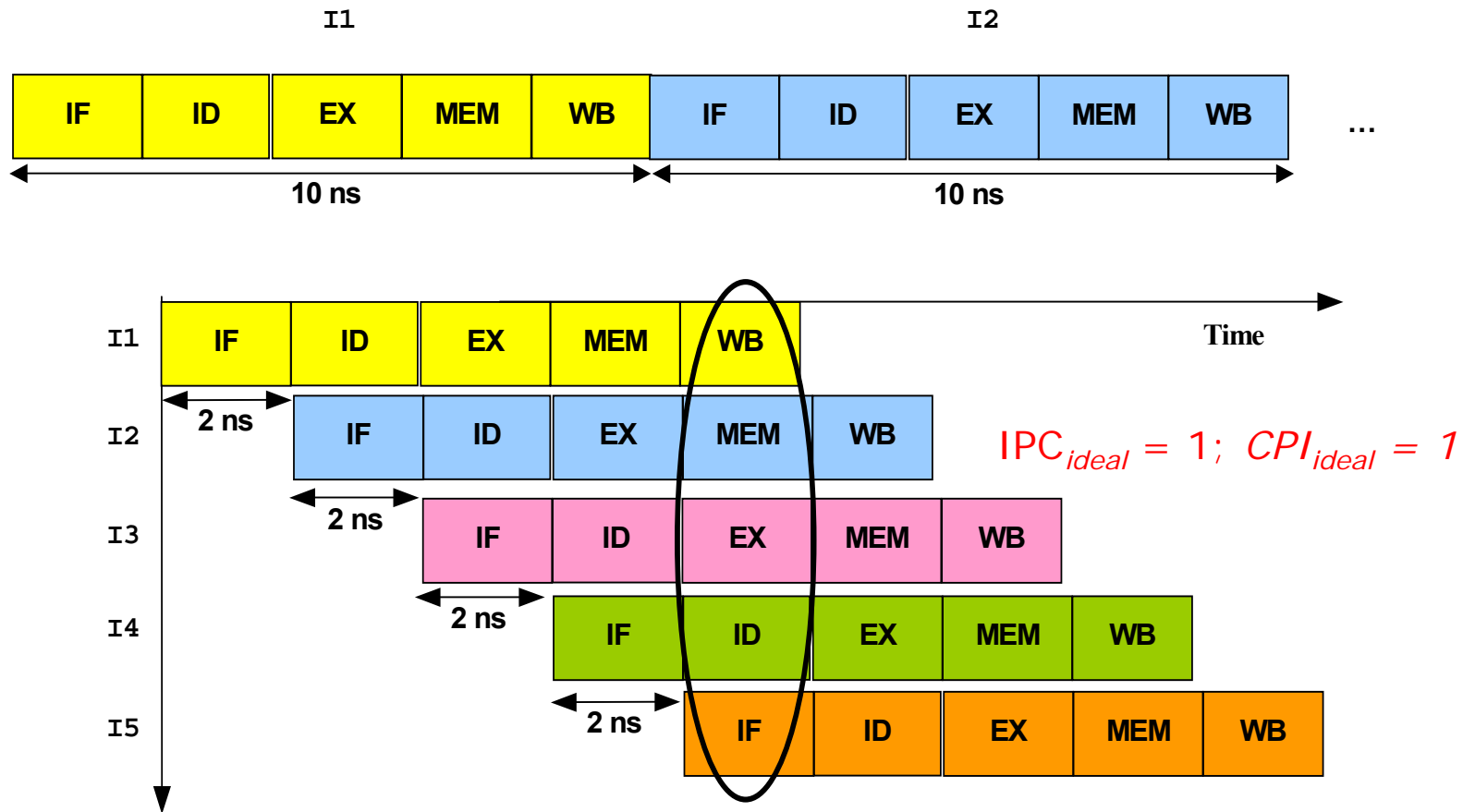
Outline of Part I

Introduction to ILP

Dynamic scheduling vs Static scheduling

Superscalar vs VLIW

Sequential vs. Pipelining Execution



Getting higher performance...

- In a pipelined machine, actual CPI is derived as:
$$CPI_{pipeline} = CPI_{ideal} + \text{Structural Stalls} + \text{Data Hazard Stalls} + \text{Control Stalls} + \text{Memory Stalls}$$
- Reduction of any right-hand term reduces $CPI_{pipeline}$ to CPI_{ideal} (and increases *Instructions Per Clock*:
 $IPC = 1 / CPI$)
- **Best case:** the max throughput would be to complete 1 Instruction Per Clock:
 $IPC_{ideal} = 1; CPI_{ideal} = 1$

Summary of Pipelining Basics

- Hazards limit performance:
 - **Structural:** Need more HW resources
 - **Data:** Need forwarding, Compiler scheduling
 - **Control:** Early evaluation, Branch Delay Slot, Static and Dynamic Branch Prediction
- Increasing length of pipe (***superpipelining***) increases impact of hazards
- Pipelining helps instruction throughput, not latency

Summary of Dependences

- Determining **dependences** among instructions is critical to defining 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.
- **Three different types of dependences:**
 - **Data Dependences (or True Data Dependences)**
 - **Name 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 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 (*it can generate a **WAR***). 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 (*it can generate a **WAW***). 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.

Summary: Types of Data Hazards

Consider executing a sequence of


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

Data-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_5 \leftarrow (r_3) \text{ op } (r_4) \end{array}$$



Read-after-Write
(RAW) hazard

Anti-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_1 \leftarrow (r_4) \text{ op } (r_5) \end{array}$$


Write-after-Read
(WAR) hazard

Output-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_3 \leftarrow (r_6) \text{ op } (r_7) \end{array}$$


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.

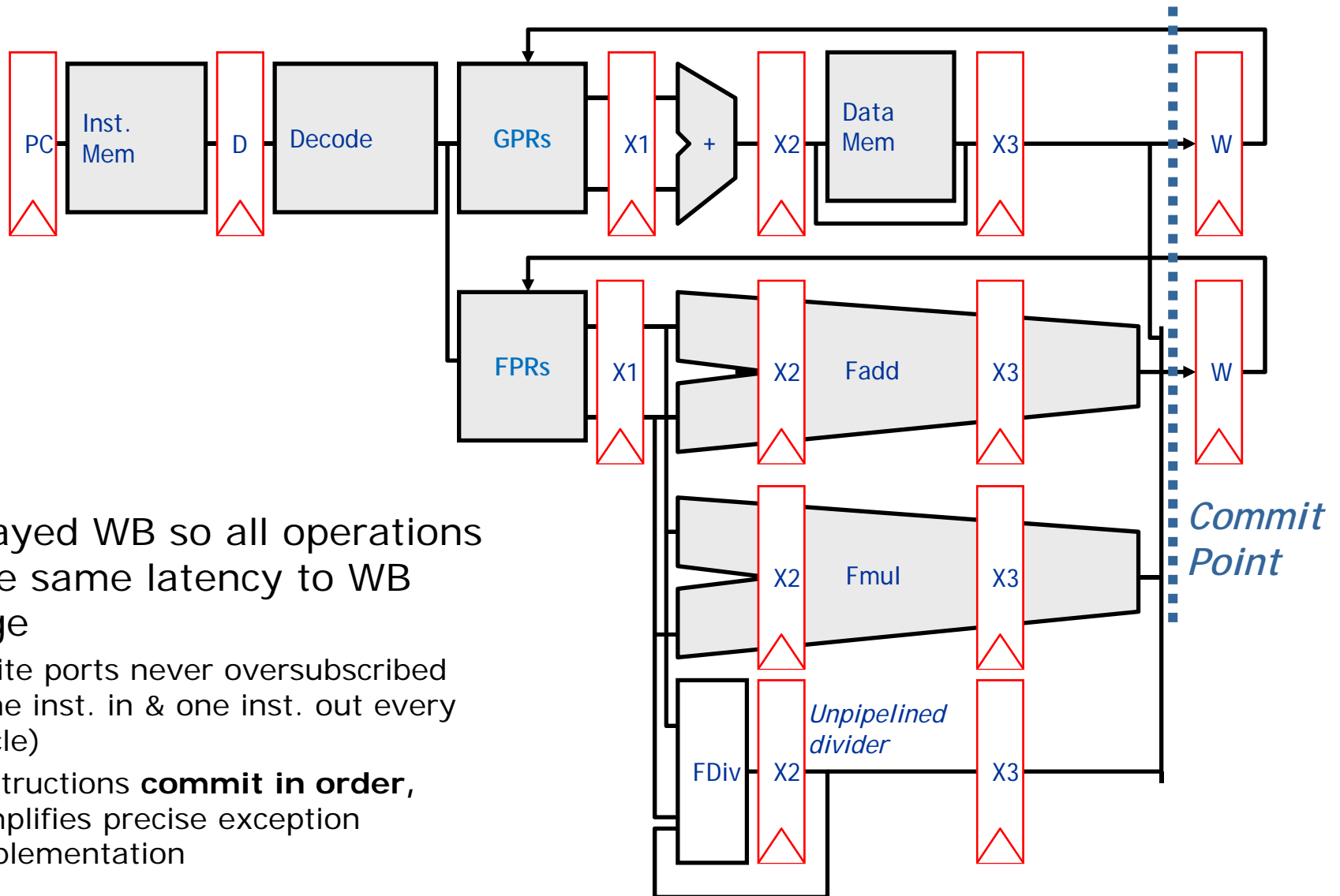
Program Properties

- **Two properties** are critical to program correctness (and normally preserved by maintaining both data and control dependences):
 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-cycles: Basic Assumptions

- We consider ***single-issue*** processors
- Instructions are then issued ***in-order***
- Execution stage might require ***multiple cycles***, depending on the operation type.
- Memory stage might require ***multiple cycles*** access time due to data cache misses

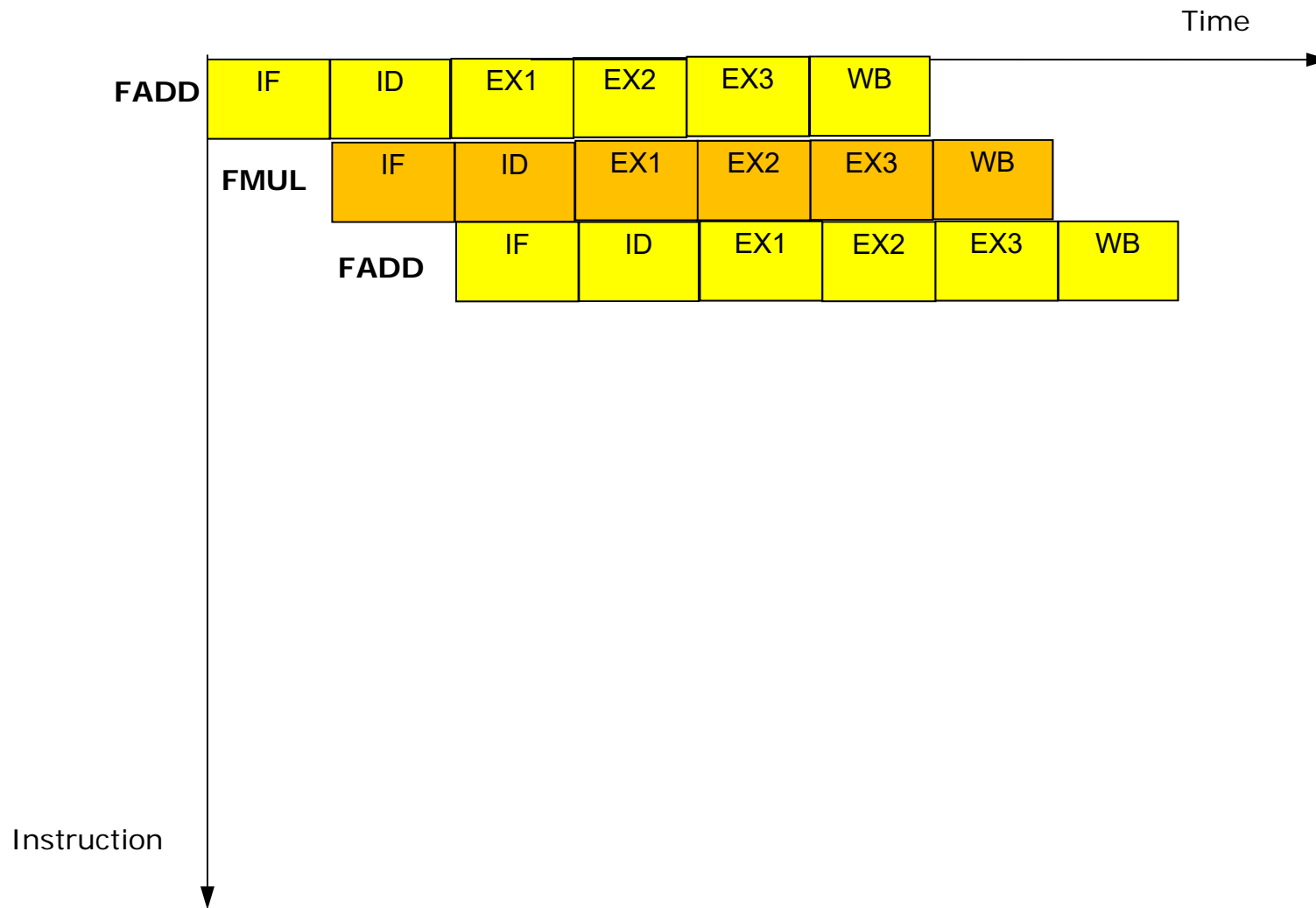
Complex Multi-cycle In-Order Pipeline



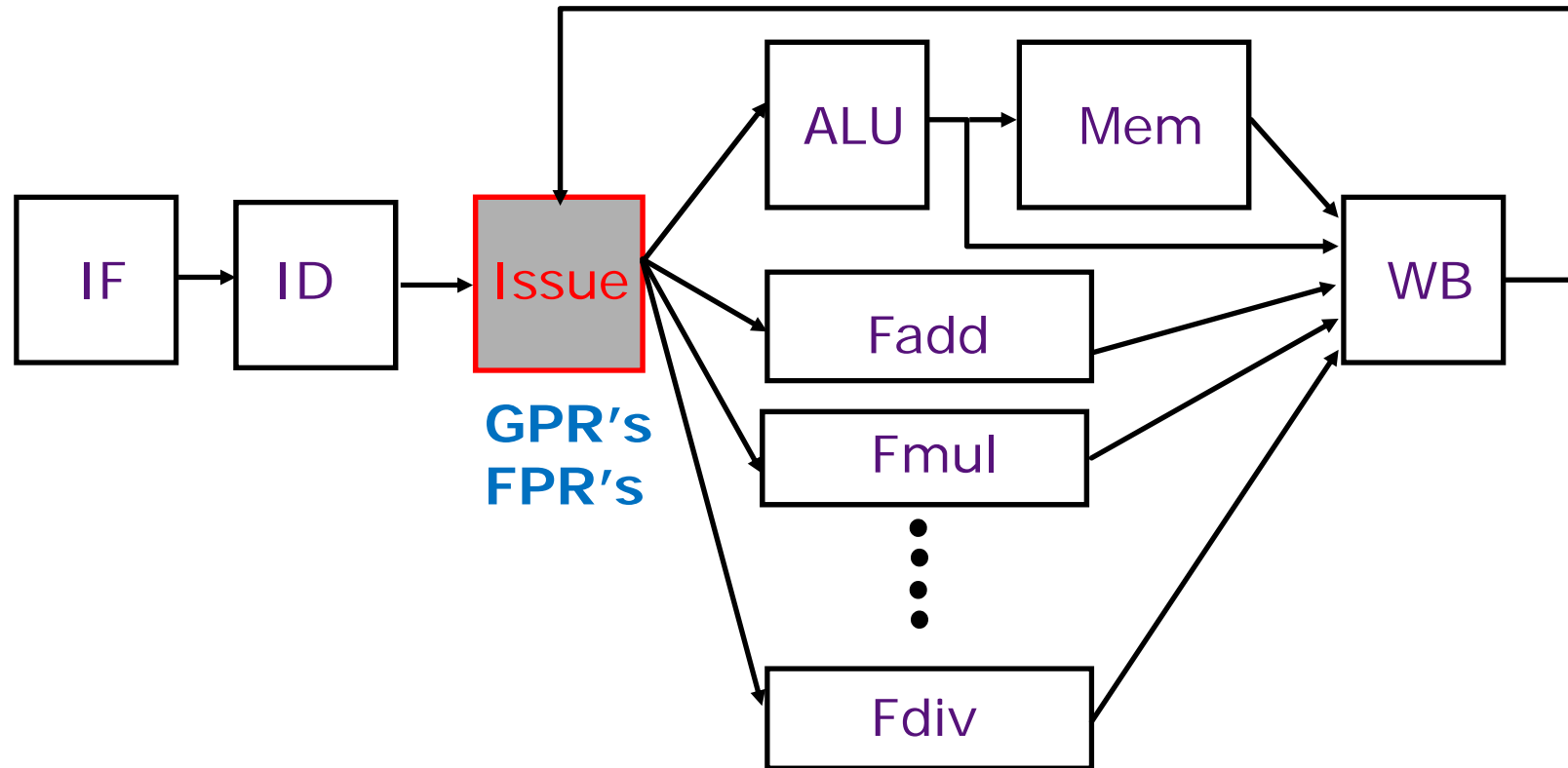
Delayed WB so all operations have same latency to WB stage

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

In-order Issue & In-order Commit

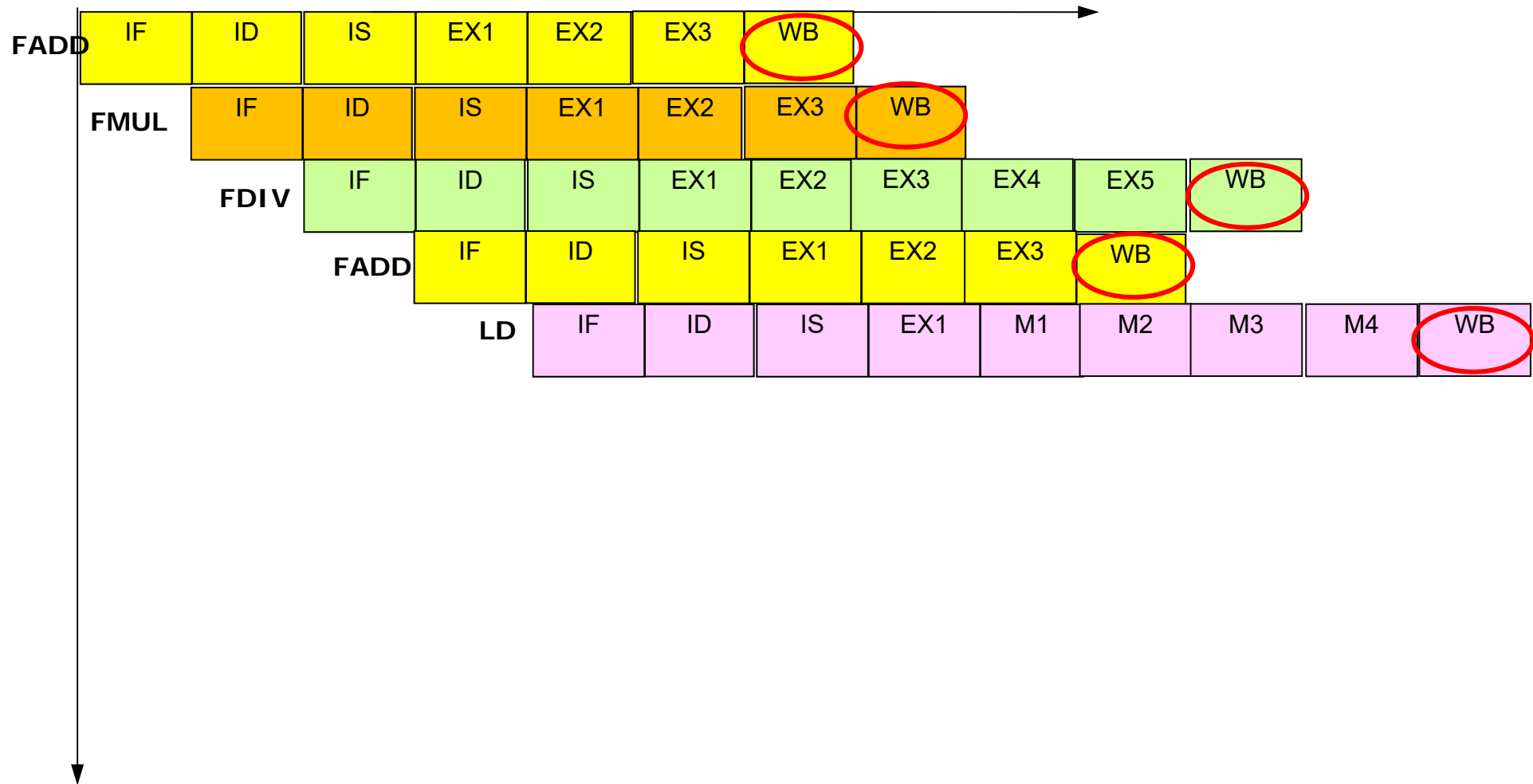


Complex Multi-cycle Out-of-order Pipeline



- Multiple functional units and memory units
- Long latency **multi-cycle floating-point operations**
- Memory systems with variable access time: **Multi-cycle memory accesses** due to data cache misses (statically unpredictable)

In-order Issue & Out-of-order Execution and Commit



Definition of Instruction Level Parallelism

- **ILP = Exploit potential overlap of execution among unrelated instructions**
- Overlapping possible whenever:
 - No Structural Hazards
 - No RAW, WAR or WAW Hazards
 - No Control Hazards

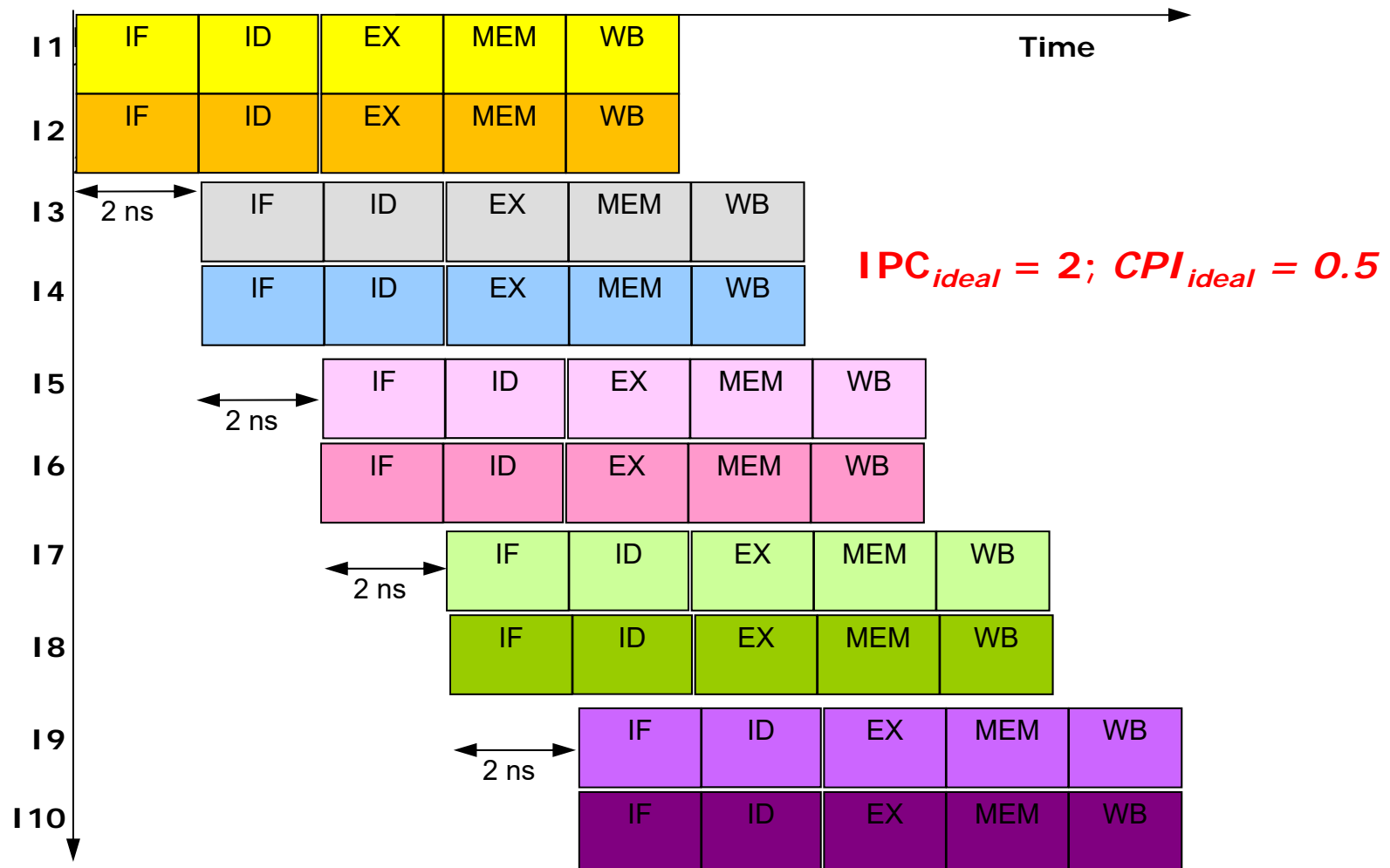
Getting higher performance...

- To reach higher performance (for a given technology) – more parallelism must be extracted from the program. In other words...**multiple-issue**
- Dependences must be detected and solved, and instructions must be *re-ordered* (**scheduled**) so as to achieve highest parallelism of instruction execution compatible with available resources.

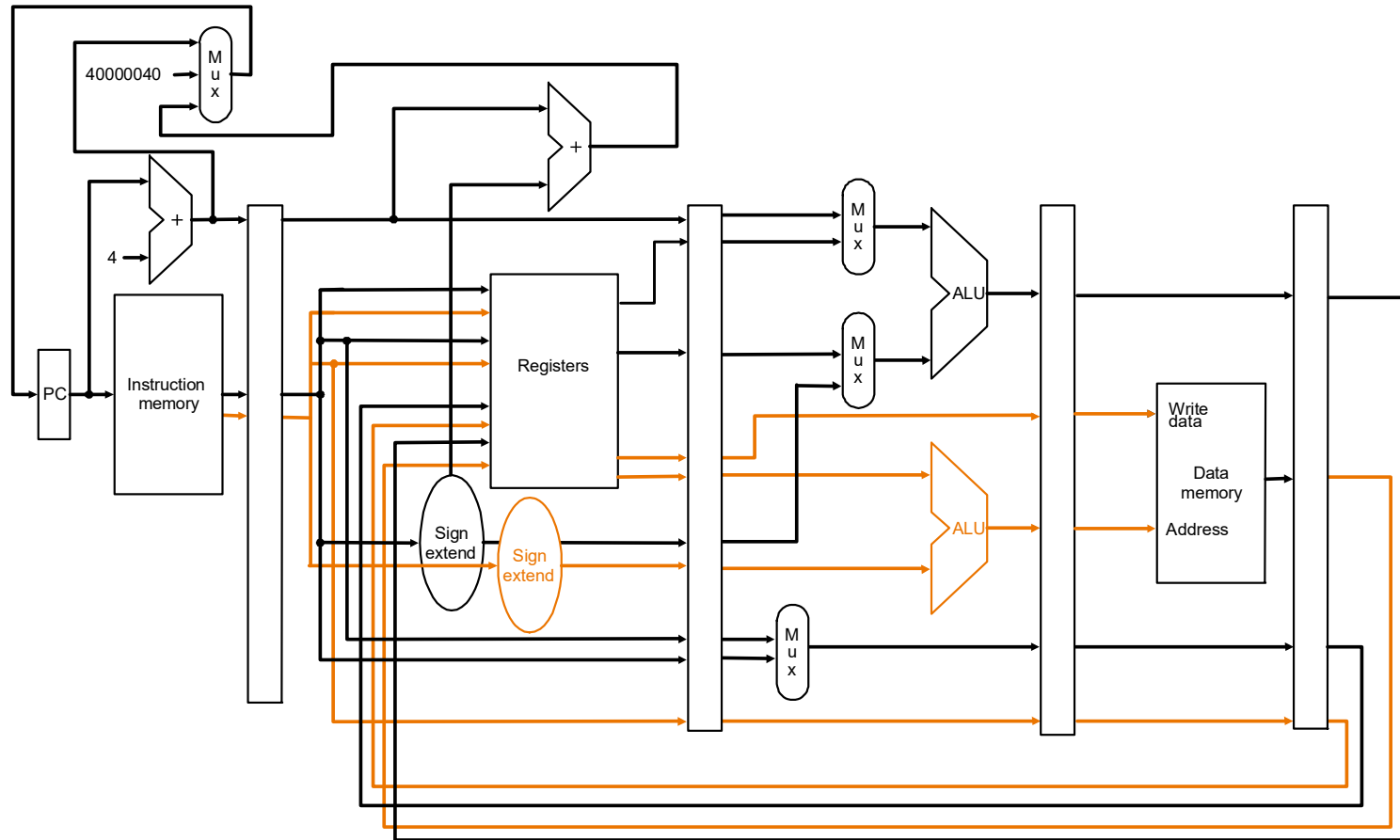
Getting higher performance...

- In a **multiple-issue pipelined** processor, the ideal **CPI** would be $CPI_{ideal} < 1$
- If we consider for example **2-issue** processor, *best case*: max throughput would be to complete 2 Instructions Per Clock:
 $IPC_{ideal} = 2; CPI_{ideal} = 0.5$

ILP: Dual-Issue Pipelining Execution



2-issue MIPS Pipeline Architecture



2-instructions issued per clock:

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

Key Idea: Dynamic Scheduling

- **Problem:** Hazards due to 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 rearranges dynamically the instruction execution to reduce stalls
- **Enables out-of-order execution and completion (commit)**
- First implemented in CDC 6600 (1963).

Example 1

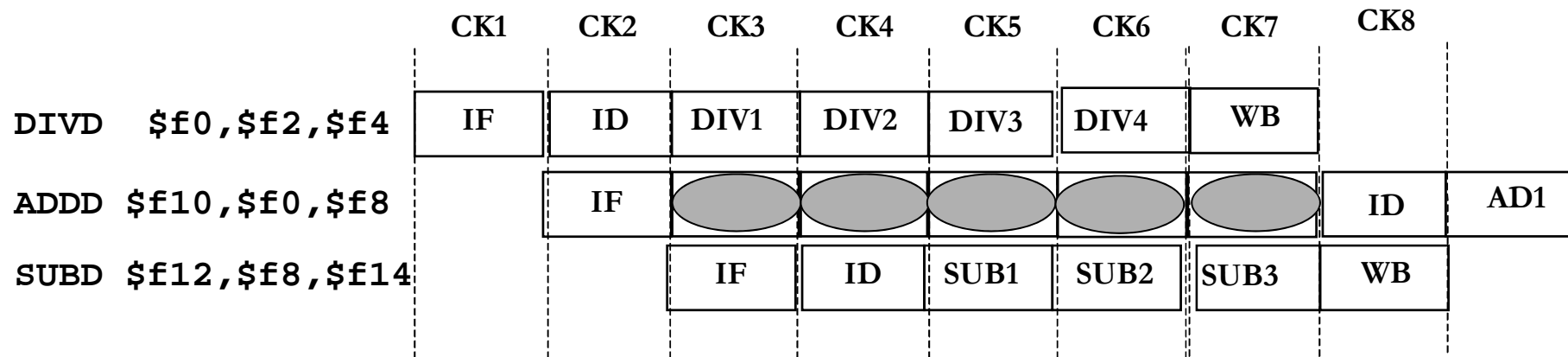
DIVD **F0**, F2, F4

ADDD F10, **F0**, F8 # RAW F0

SUBD **F12**, **F8**, **F14**

- RAW Hazard: ADDD stalls for F0 (waiting that DIVD commits).
- SUBD would stall even if not data dependent on anything in the pipeline without dynamic scheduling.
- **BASIC IDEA: to enable SUBD to proceed
=> out-of-order execution**

Example 1



Exception handling

- Problem with **out-of order completion**
 - Must preserve exception behavior as in-order execution
- Solution: ensure that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed

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 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

Instruction Level Parallelism

- Two strategies to support ILP:
 - **Dynamic Scheduling:** Depend on the hardware to locate parallelism
 - **Static Scheduling:** Rely on compiler for identifying potential parallelism
- Hardware intensive approaches dominate desktop and server markets

Dynamic Scheduling

- The hardware **reorder** dynamically the instruction execution to reduce pipeline stalls while maintaining data flow and exception behavior.
- Main advantages **(PROs)**:
 - 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 (code portability).
- Those advantages are gained at a cost of **(CONS)**:
 - A significant increase in hardware complexity
 - Increased power consumption
 - Could generate *imprecise* exceptions

Dynamic Scheduling

- **Simple pipeline:** hazards due to data dependences that cannot be hidden by forwarding ***stall*** the pipeline – no new instructions are fetched nor issued.
- ***Dynamic scheduling:*** Hardware reorder instructions execution so as to reduce stalls, maintaining data flow and exception behaviour.
- *Typical example: **Superscalar Processor***

Dynamic Scheduling (2)

- Basically: Instructions are *fetch*ed and *issue*d 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 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

Static Scheduling

- Static detection and resolution of dependences
 - ⇒ **static scheduling**: accomplished by the **compiler**
 - ⇒ dependences 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** generated by the compiler

Static Scheduling

- **Compilers** can use sophisticated algorithms for code scheduling to exploit **ILP (Instruction Level Parallelism)**.
 - The size of 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 usually quite **small** and the amount of parallelism available within a basic block is quite **small**.
 - Example: For typical MIPS programs the average branch frequency is between 15% and 25% \Rightarrow from 4 to 7 instructions execute between a pair of branches.

Static Scheduling

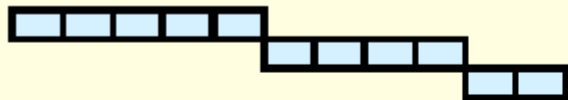
- Data dependence can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size.
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks (i.e. across branches such as in trace scheduling).

Main Limits of Static Scheduling

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

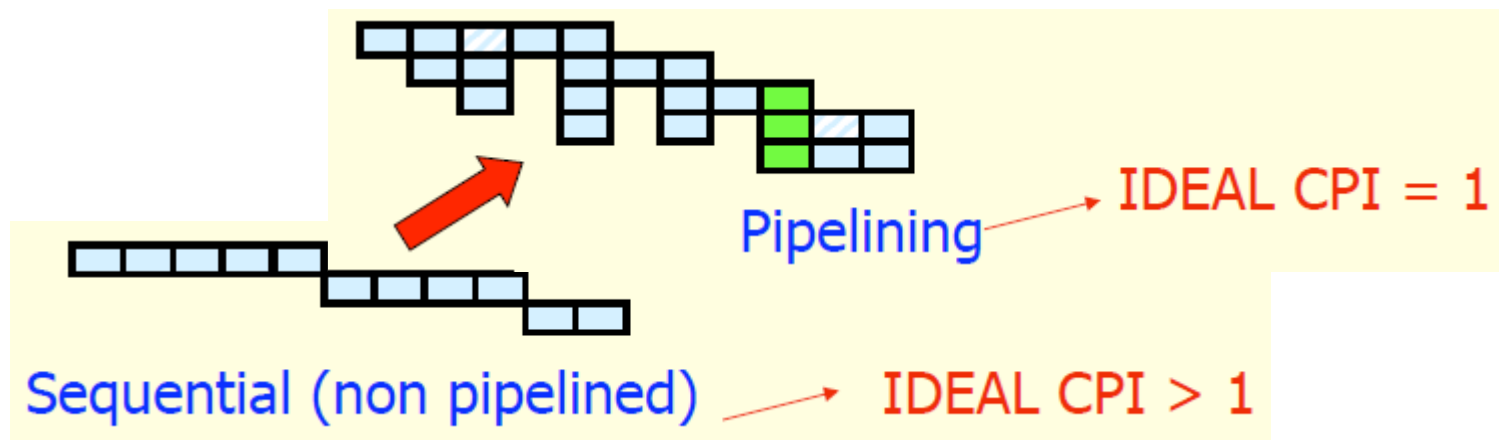
Several steps towards exploiting more ILP

Several steps towards exploiting more ILP

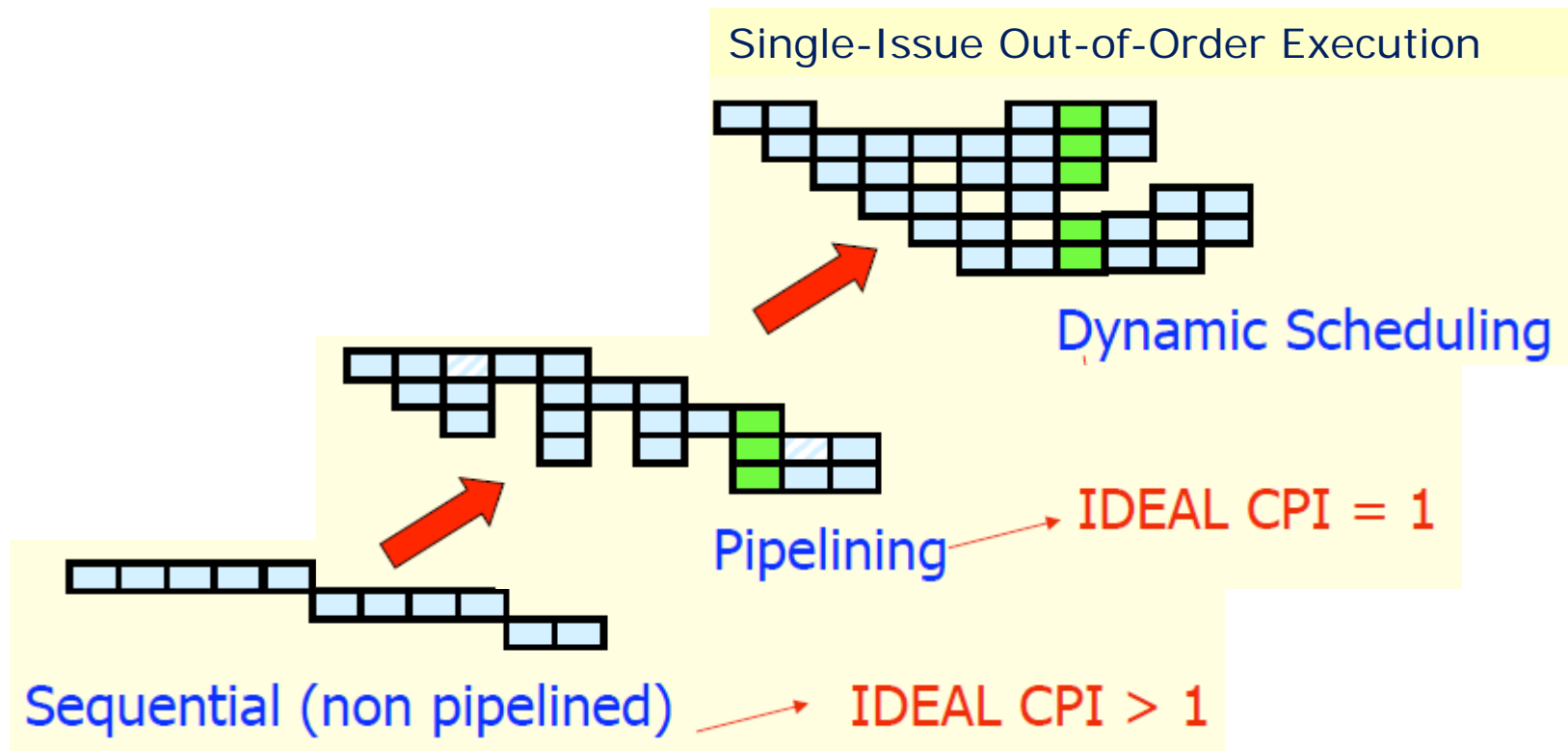


Sequential (non pipelined) \rightarrow IDEAL CPI > 1

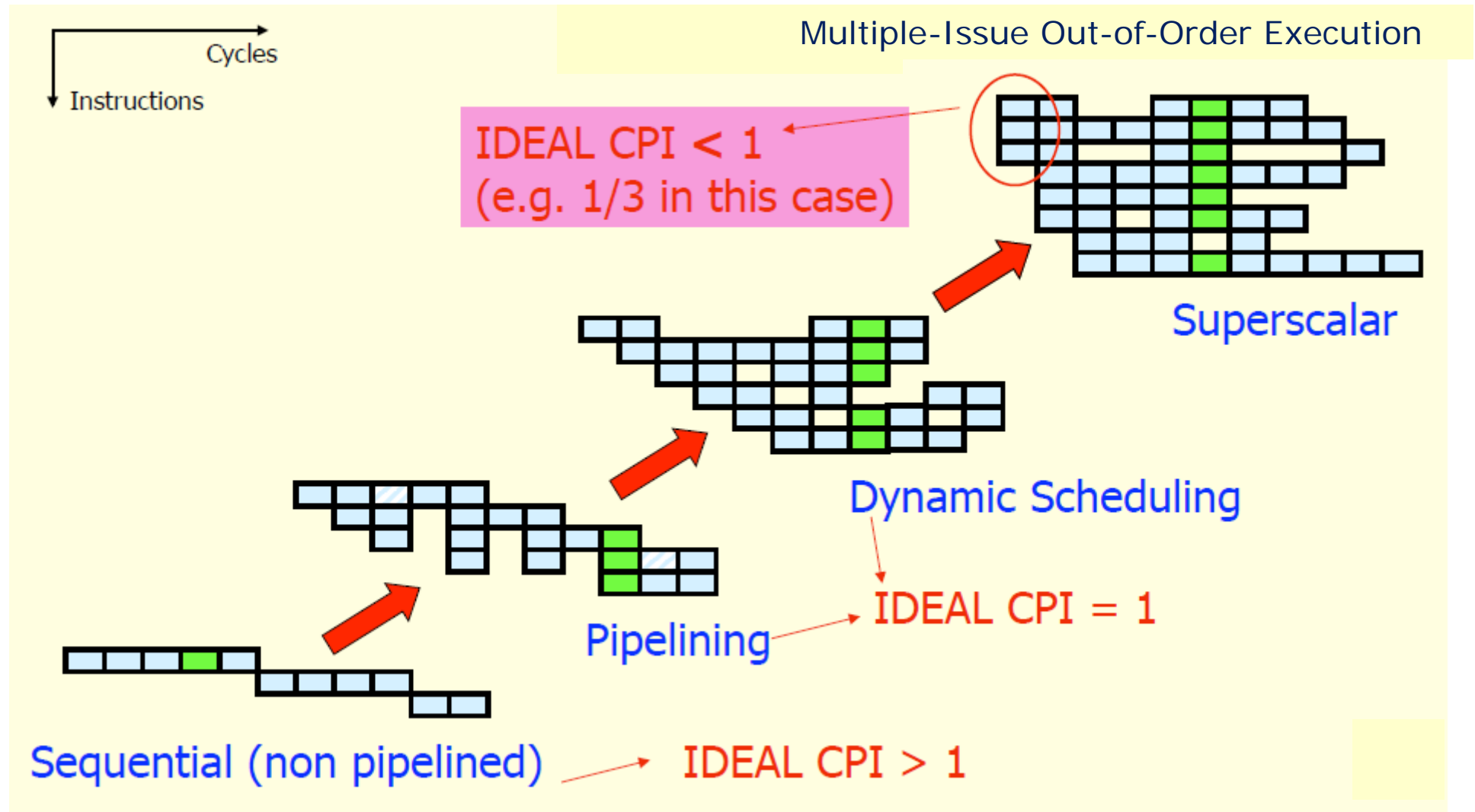
Several steps towards exploiting more ILP



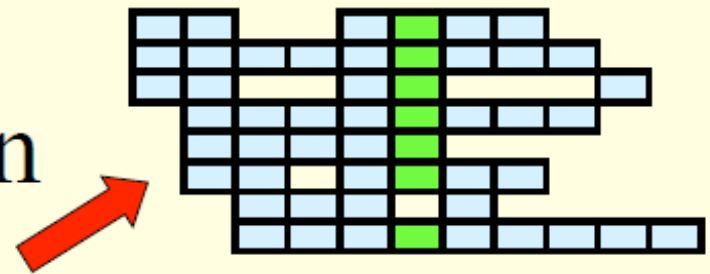
Several steps towards exploiting more ILP



Several steps towards exploiting more ILP

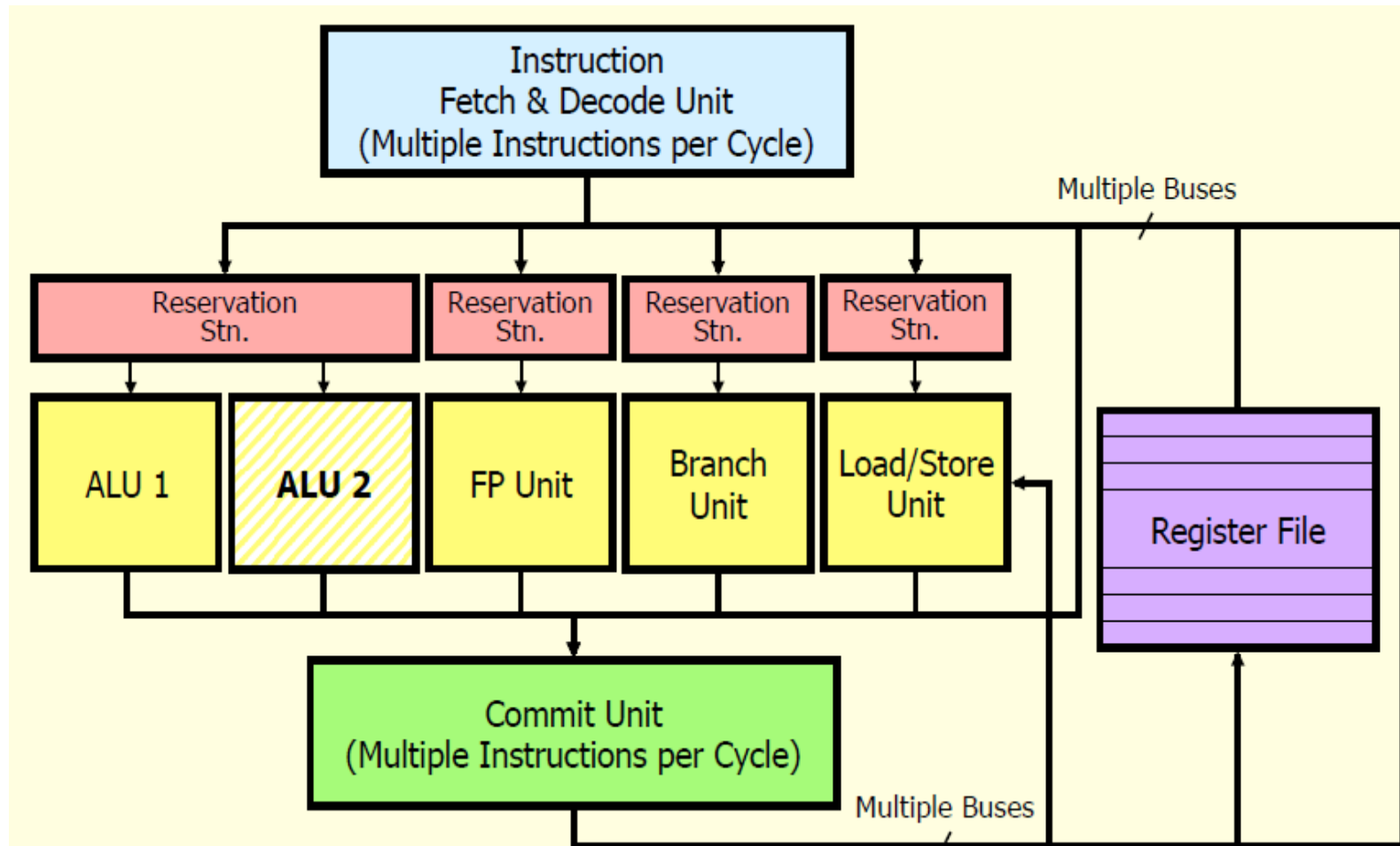


Superscalar Execution

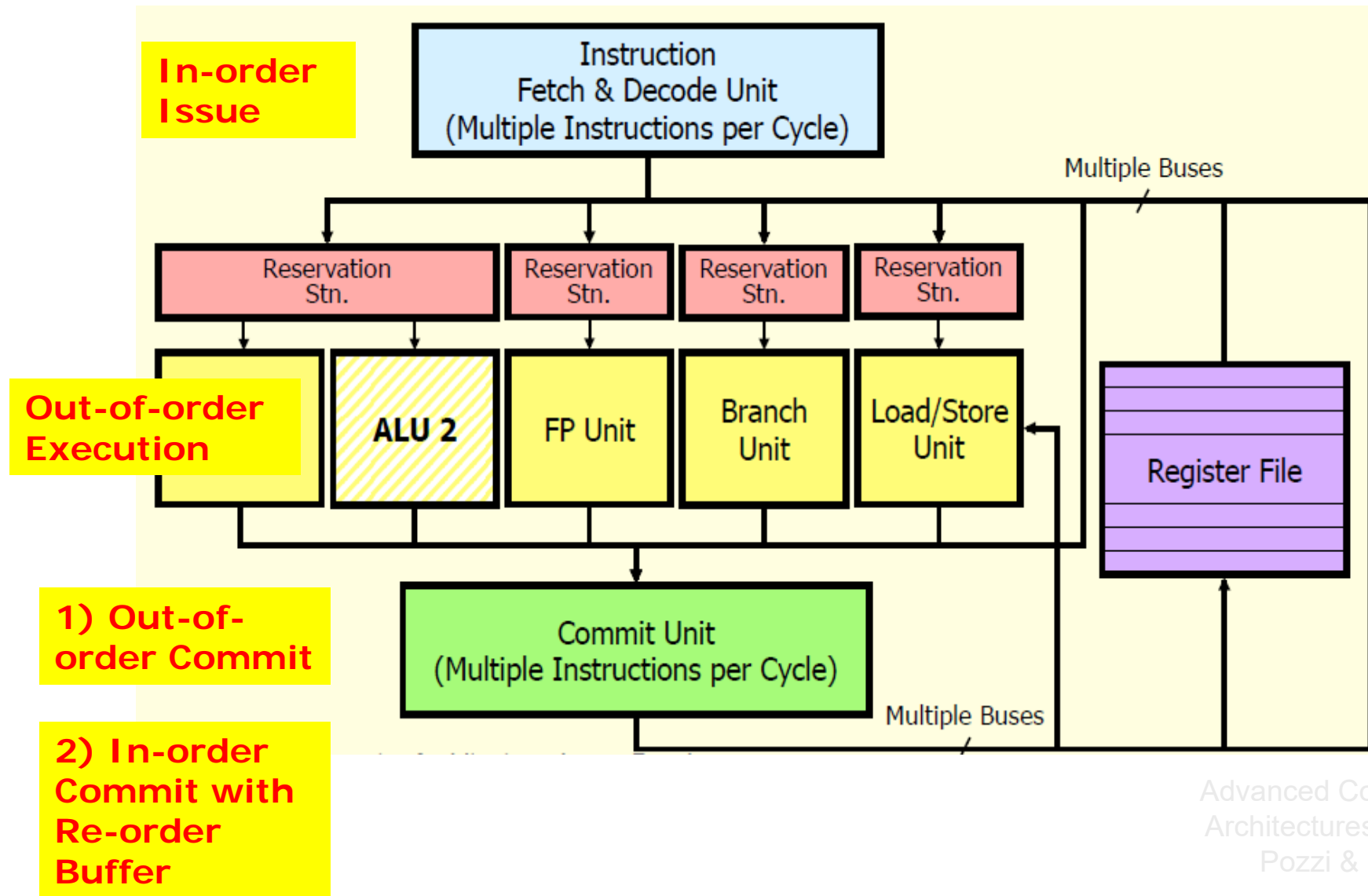


- This is what **all high-end computers now do**
 - (PowerPC, Pentium, Sparc, ...)
- **Main idea: why not more than one instruction beginning execution (issued) per cycle?**
- Key requirements are
 - **Fetching more instructions in a cycle:** no big difficulty provided that the instruction cache can sustain the bandwidth
 - **Decide on data and control dependencies:** dynamic scheduling already takes care of this

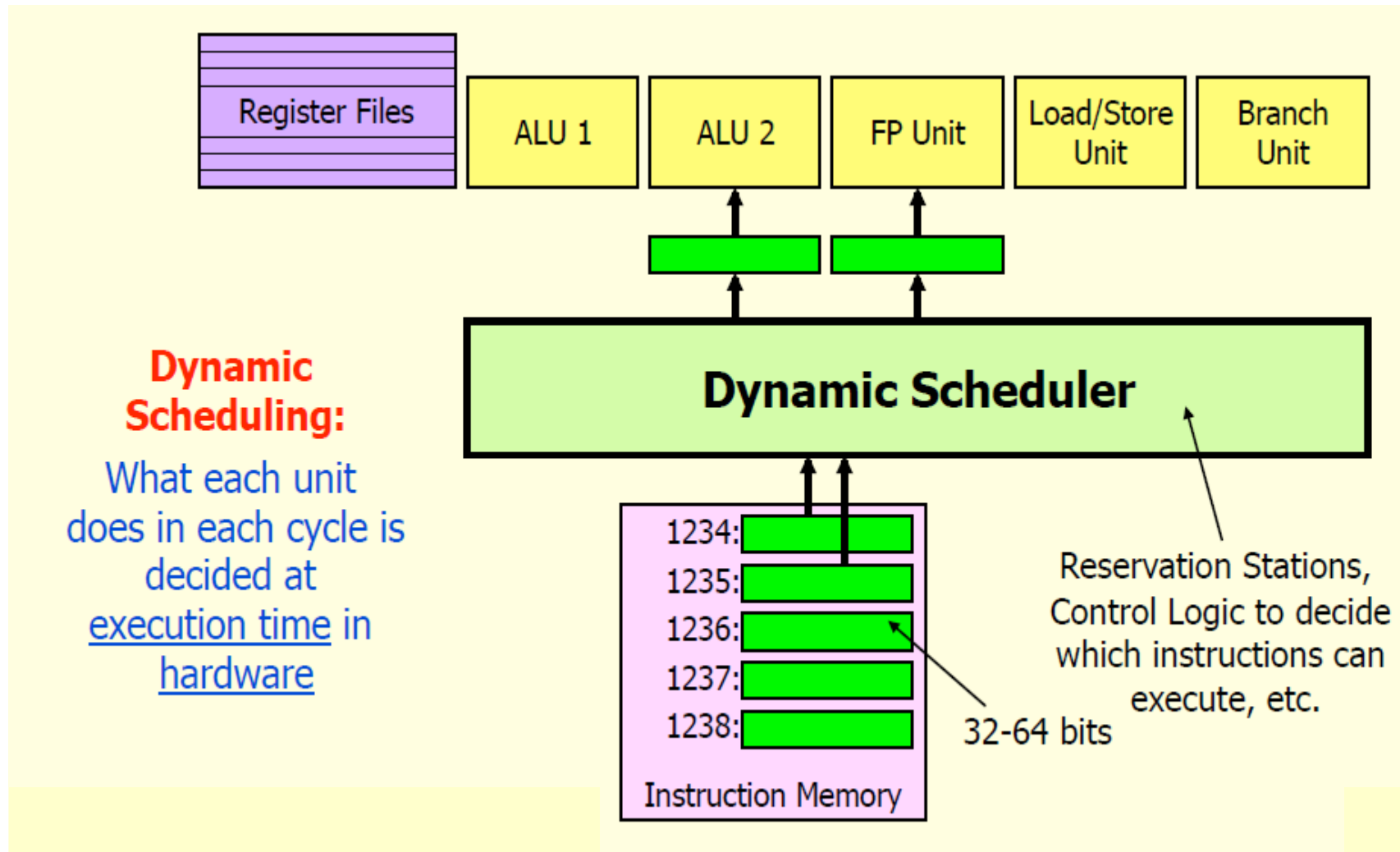
Superscalar Processor: Multiple-issue + Dynamic Scheduling



Superscalar Processor: Multiple-issue + Dynamic Scheduling

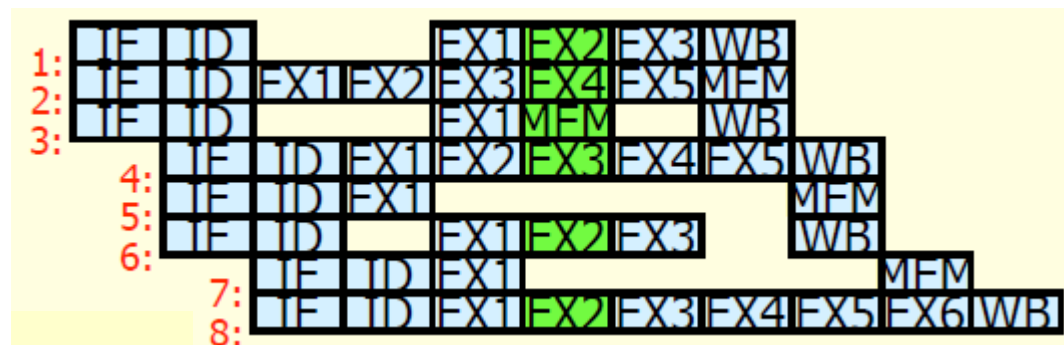
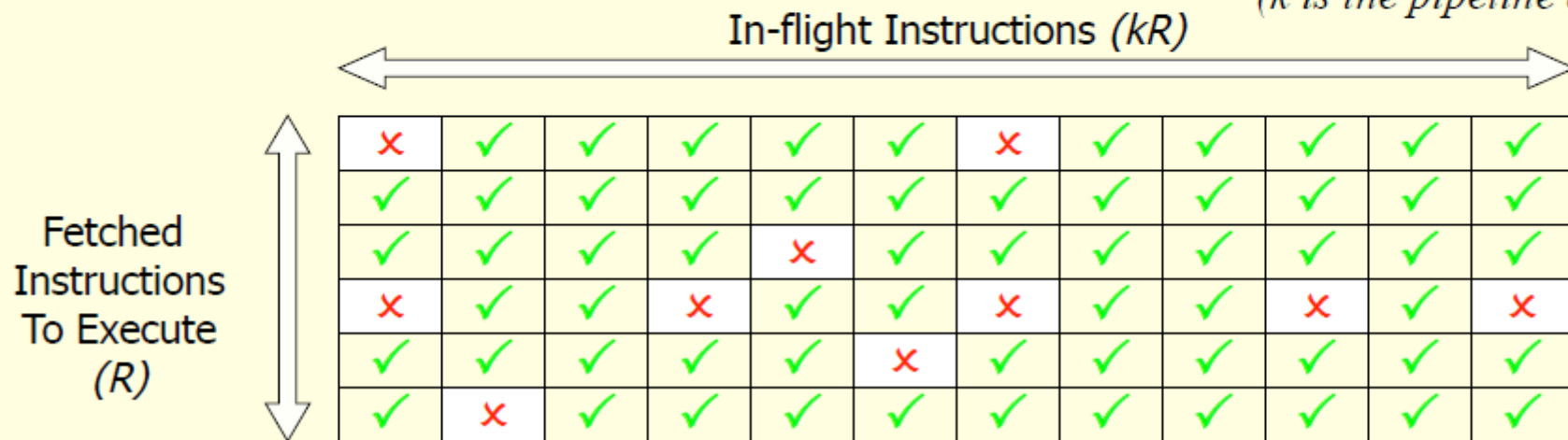


Dynamic Scheduler



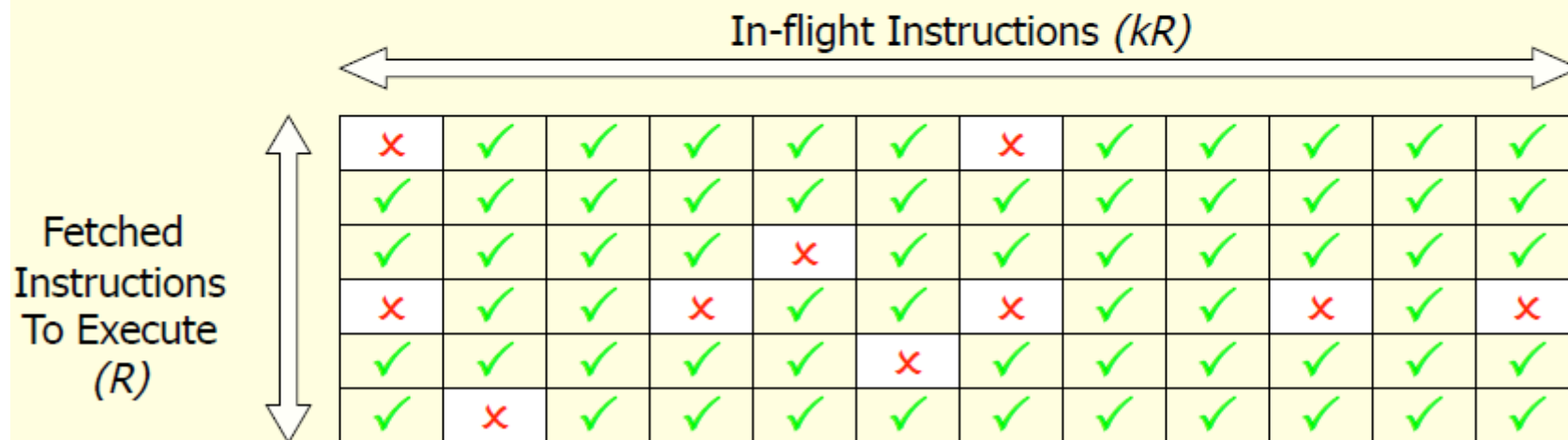
Dynamic Scheduler

- Scheduling complexity (e.g., checking dependences) is typically of the order of the square in the issue rate (R) *(k is the pipeline depth)*



Dynamic Scheduler

- Every cycle, the processor needs to decide which instructions can begin execution
- It needs to check all fetched instructions with all in-flight instructions to see which are *independent* and therefore can start execution



- There is a limit to how many instructions can be checked during a clock cycle

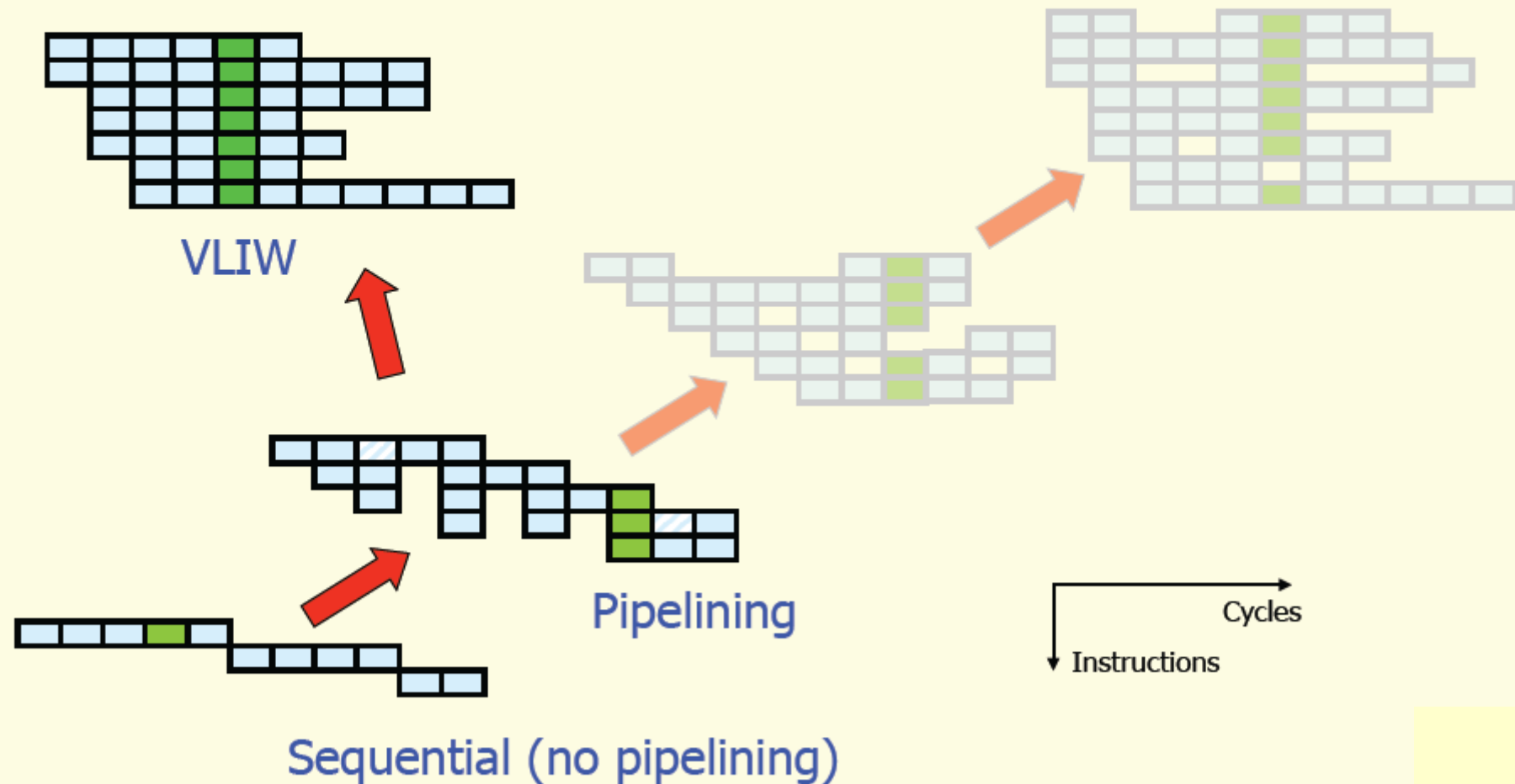
Dynamic Scheduling is expensive!

- Large amount of logic, significant area cost
 - PowerPC 750 Instruction Sequencer is approx. 70% of the area of all execution units!
(Integer units + Load/Store units + FP unit)
- Cycle time limited by scheduling logic (dispatcher and associated dependency checking logic)
- Design verification extremely complex
 - Very complex irregular logic

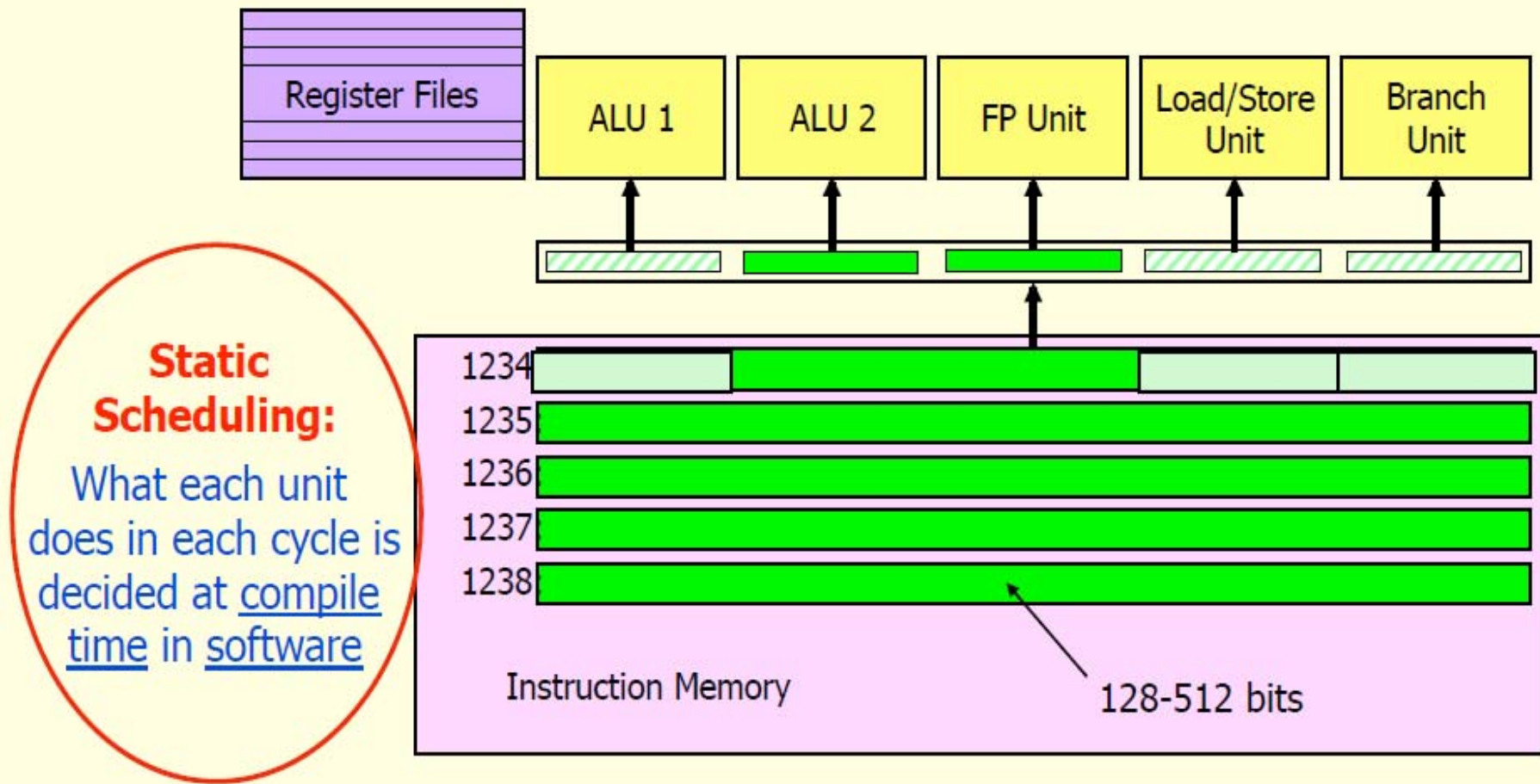
Summary of superscalar and dynamic scheduling

- Main advantage:
 - Very high performance: Ideal CPI very low:
 $\text{CPI}_{\text{ideal}} = 1 / \text{issue-width}$
- Disadvantages
 - Very expensive logic to decide dependencies and independencies, i.e. to decide which instructions can be issued every clock cycle
 - It does not scale: almost impractical to make issue-width greater than 4 (we would have to slow down the clock)

Very Long Instruction Word: An Alternative Way of Extracting ILP



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



Superscalar vs VLIW Scheduling

- Deciding ***when*** and ***where*** to execute an instruction
 - *i.e.* in which cycle and in which functional unit
- For a superscalar processor it is decided **at run time**, by custom logic in HW
- For a VLIW processor it is decided **at compile time**, by the compiler, and therefore by a SW program
 - Good for embedded processors: Simpler HW design (no dynamic scheduler), smaller area and power consumption ... and cheap

Challenges for VLIW

- Compiler technology
 - The compiler needs to find a lot of parallelism in order to keep the multiple functional units of the processors busy
- Binary incompatibility
 - Consequence of the larger exposure of the microarchitecture (= implementation choices) at the compiler in the generated code

Advantages of SW vs HW Scheduling

SW

(= Static = Compiler)

- 1) Source code available
(higher level information)
- 2) Global analysis possible
(inter-procedural analysis,
etc.)
- 3) More time available (not
bound by cycle-time)

HW

(= Dynamic = Instruction Scheduler)

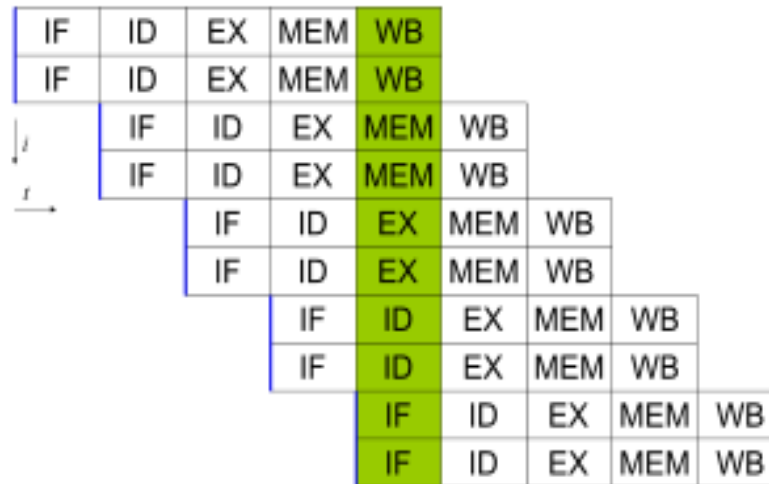
- 1) Run-time information
available (actual data,
addresses, pointers,
etc.)

Current Superscalar & VLIW processors

- Dynamically-scheduled superscalar processors are the commercial state-of-the-art for general purpose: current implementations of **Intel Core i**, **Alpha**, **PowerPC**, **MIPS** etc. are all **superscalar**
- **VLIW** processors are primarily successful as embedded media processors for consumer electronic devices (embedded):
 - TriMedia media processors by NXP (formerly Philips Semiconductors)
 - The C6000 DSP family by Texas Instruments
 - The STMicroelectronics ST200 family
 - The SHARC DSP by Analog Devices
 - Itanium 2 is the only general purpose VLIW, a 'hybrid' VLIW (EPIC, Explicitly Parallel Instructions Computing)

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 superscalar was invented, 2- and rapidly 4-issue width processors were created (i.e. 4 instructions executed in a single cycle, ideal CPI = 1/4)



Issue-Width limited in practice

- Now, the maximum (rare) is **6**, but no more exists.
- Issue width of current processors ranges from:
 - **single-issue** (ARM11, UltraSPARC-T1)
 - **2-issue** (UltraSPARC-T2/T3, Cortex-A8 & A9, Atom, Bobcat)
 - **3-issue** (Pentium-Pro/II/III/M, Athlon, Pentium-4, Athlon 64/Phenom, Cortex-A15)
 - **4-issue** (UltraSPARC-III/IV, PowerPC G4e, Core 2, Core i, Core i*2, Bulldozer)
 - **5-issue** (PowerPC G5)
 - or even **6-issue** (Itanium, but it's a VLIW).
- Because it is too hard to decide which 8, or 16, instructions can execute every cycle (too many!)
 - It takes too long to compute, so the frequency of the processor would have to be decreased
 - Limitation due to intrinsic level of parallelism

Issue-Width limited in practice

More levels of parallelism:

- Multi-threading
- Multi-processing and Multi-cores
- Vector Processors and GPUs