Instruction level Parallelism (Complements)

→ Efficiency of piplining

Degree success by running a program in appelined processor :

$$CPI_{pug} = CPI_{ideal} + \text{structural stalls} + \text{data stalls} + \text{control stalls}.$$

How to exploit ILP with two different approaches

- relying on software tech to elicit parallelism at compile time.
- relying on hardware to help discover and exploit dynamically the inherent parallelism during execution.

→ Data dependences and hazards

3 types of dependences :

- data dependences
- name dependences
- control dependences

↳ Data dependences

Two instructions have data dependencies with values produced being used by another on a chain of dependencies.

Whether a given dependence produces a hazard and whether that hazard causes a stal are properties of the pipeline organization.

Data dependence conveys three ideas:
- the possibility of a hazard
- the order the results must be computed
- how much parallelism can be exploited

They can be overcome in two ways:
- maintaining dependence but avoiding hazard
- eliminating dependence by transforming the code. (scheduling the code)

↳ Name dependences

Happens when two instructions use the same register or memory location, called name, without any flow of information taking place between them. Two types:

- an antidependence occurs when an instruction writes to a register or memory location that the other instruction reads. Instruction ordering must be preserved.
- an output dependence occurs when both instructions write to a register/memory location. The original order must be preserved.

Renaming can be used for register operands.

↳ Data hazards

Exist whenever there's a data dependence between instructions and they are close enough to generate a change in order of access to the operand involved in the dependence. They can be classified in three different categories:

- RAW (read after write): j tries to read the operand before i writes a value to it, so j gets the wrong value. Most common, true data dependence.
- WAW (write after write): two operations write and the wrong value is written. Output dependence.
- WAR (write after read): j tries to write a a value to an operand before i reads it, so i gets a wrong value. It cannot occur in most static issue pipelines.

→ Control dependences

A control dependence determines the ordering of an instruction i with respect to a branch. Two constraints are in general imposed by control dependences:

- an instruction that is controll dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
- an instruction that is not control dependent on a branch, cannot be moved after the branch so that its exception is controlled by the branch.

→ Tournament predictors

Type of branch prediction that uses a tournament between different prediction mechanisms. Typically uses two or more predictors, such as local or global history prediction and then selects the best prediction among them.

→ Dynamic scheduling

In-order instruction issue and execution can generate data dependences that canot be solved by forwarding, so the interlocking unit stalls the pipeline.

Dynamic sheduling is another way of addressing the problem where the hardware rearranges the order of execution while maintaining data flow.

Advantages:

- allows code to be run on different pipelines.
- enables handling of dependencies unknown at compile time.
- allows the processor to cope with unpredicted delays, such as cache misses.

A pipeline with these features performs out-of-order execution, which introduces the possibility of WAR and WAW hazards which did not exist in the classical 5-step pipeline. Also creates complications in handling exceptions

To allow out-of-order executions, the ID stage of the classical 5-stage pipeline is split into two steps

- issue → instruction decoding and checking for structural hazards.
- read operands → waiting until all data hazards are cleared before reading the operands.

→ Scoreboarding

The goal of a scoreboard is to try and maintain an execution rate of one instruction per clock cycle, provided there are not structural hazards.

When an instruction which has been issued is stalled, other instructions in the processing Table that do not depend on any active or stalled instruction, are looked up in the processing table and if one is found, it is executed.

When an instruction is fetched, its scoreboard entry is updated to indicate that its in the fetch stage. As the instruction moves through the pipeline, its scoreboard entry is updated to reflect its progress.

Steps an instruction goes through in a scoreboard control:

1. issue → instruction is fetched from memory and assigned to a scoreboard entry. The scoreboard updates the entry stage to the instn. fetch stage.

2. read operands → instn. reads operands. The scoreboard entry is updated to indicate the instruction is in the decode stage. If that instn. is dependent on a register/memory location that is not ready, the instruction is held on the pipeline until data is available.

3. execution → the instruction is executed. When the result is ready, it notifies the scoreboard that it has completed the operation.

4. write result → once the scoreboard is aware the functional unit has terminated its operation, it checks for WAR hazards and, if required, prevents the instruction from completing.

Scoreboard internal data structure:

. instruction status: table with as many entries as the number of instructions under processing. It specifies which of the four stages the instruction is in.

- **functional unit status:** table with as many entries as the number of functional units.

- **register result status:** single entry table with as many fields as registers in the bank. Indicates which functional unit will write the register if an active instruction has the register as its destination.