Instruction-level Parallel Processing

ILP – An Overview

Tópico 2

Instruction-level Parallel Processing

Tópico 2 : *Exploiting Instruction-level Parallelism (ILP) – An Overview.*

Tópico 3 : Instruction-level Parallel Processors.

Tópico 4 : Processing Control Transfer Instructions.

Tópico 5 : Code Scheduling for Instruction-level Parallel Processors.

An Outline of Tópico 2

- The evolution of ILP processing.
- Limitations on ILP (in software) – Dependencies between instructions.
- Matching *available* (software) and *utilisable* (hardware) parallelism – Instruction scheduling.
- Preserving sequential consistency.
- Speed-up potential of ILP processing.

The Evolution of ILP Processing

The evolution of von Neumann processors can be attributed to three areas of development:

1. improvements in technology, marked by increasing clock rates;
2. the functional evolution of processors; and
3. improvements in compiler design.

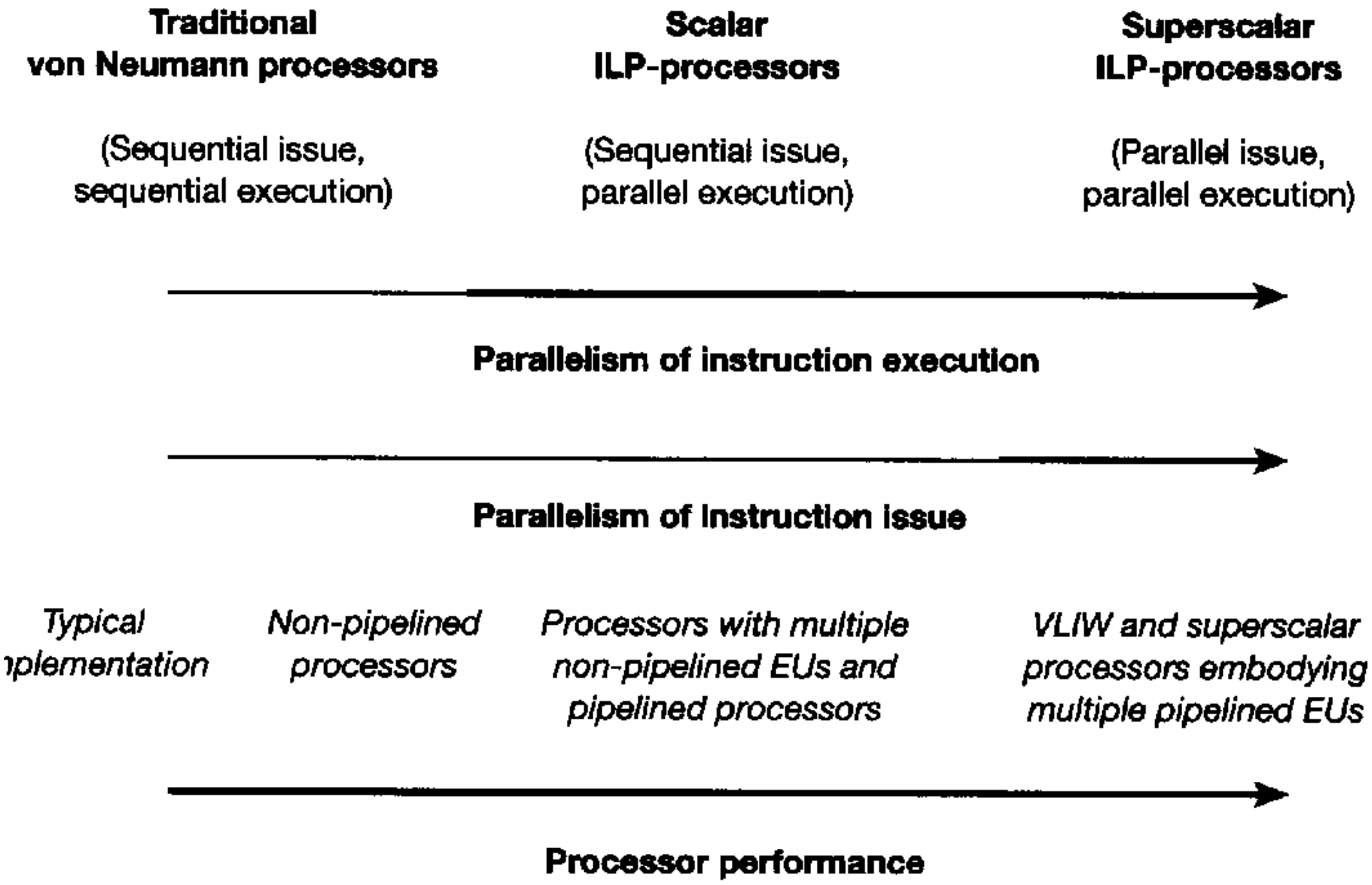
Functional Evolution of Processors

Functional evolution has been achieved primarily by increasing the degree of parallelism of the internal operations, the issue and execution of instructions.

This evolution took place in three phases and lead to three distinct designs:

1. *Traditional von Neumann processors* characterised by sequential issue and execution of instructions;
2. *Scalar ILP-processors* introduced parallel instruction execution (either through multiple functional units or by pipelining);
3. *Superscalar ILP-processors* replaced sequential instruction issue with parallel issue.

Evolution of von Neumann Processors



Scalar ILP-Processors

- Pipelining
- Instruction processing is subdivided into a number of sucessive subtasks (*stages*) each taking less than a clock cycle.
- Together these stages operate like an assembly line (*pipeline*).
- A new instruction enters the pipeline every cycle.
- Pipelining offers higher performance at a cost of increased hardware complexity (over non-pipelined architectures).

Superscalar ILP-Processors

- Replication (i.e. multiple functional units).
  - Parallel datapath operation.
  - An orthogonal alternative to parallel execution.
- Requires *highly multiported* register files.
  - Current technology restricts the number of ports to less than 20. Thus limiting the scalarity of superscalar and VLIW processors.

Very Long Instruction Word and Superscalar Processors

- *Very Long Instruction Word* (VLIW) processors issue multi-operation instructions to specify what each functional unit should do.

- Superscalar processors can issue more than one instruction from a traditional sequential stream of instructions to available functional units.

Very Long Instruction Word and Superscalar Processors$_{(cont)}$

Constraints arise from the fact that ILP-processors allow instructions to execute in parallel – dependencies among instructions and the need to preserve the *sequential execution semantics* must be taken into account.

- Each VLIW instruction must be *dependency-free.*

- Superscalar processors cope with *dependencies* using hardware.

Dependencies Between Instructions

There are three kinds of dependencies:

Data dependencies – subsequent instructions are dependent on each other because of data.

Control dependencies – subsequent instructions are dependent on the outcome of a conditional transfer instruction.

Resource dependencies – subsequent instructions require the same resource(s).

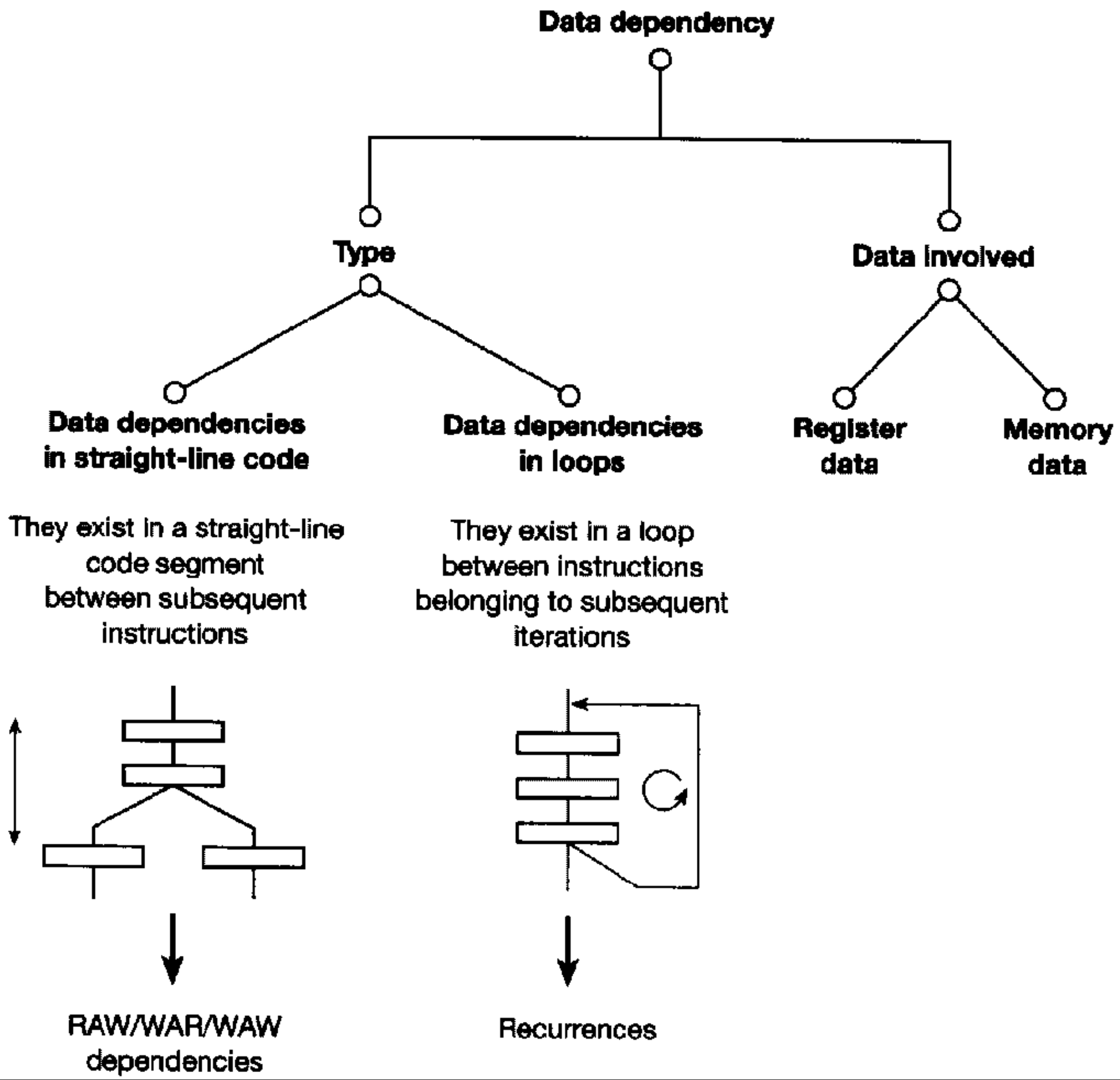
Data Dependencies

Data dependencies can occur in:

- *straight-line code* as read-after-write (RAW), write-after-read (WAR) or write-after-write (WAW) dependencies; or in

- *loops*, these are known as *recurrences*;

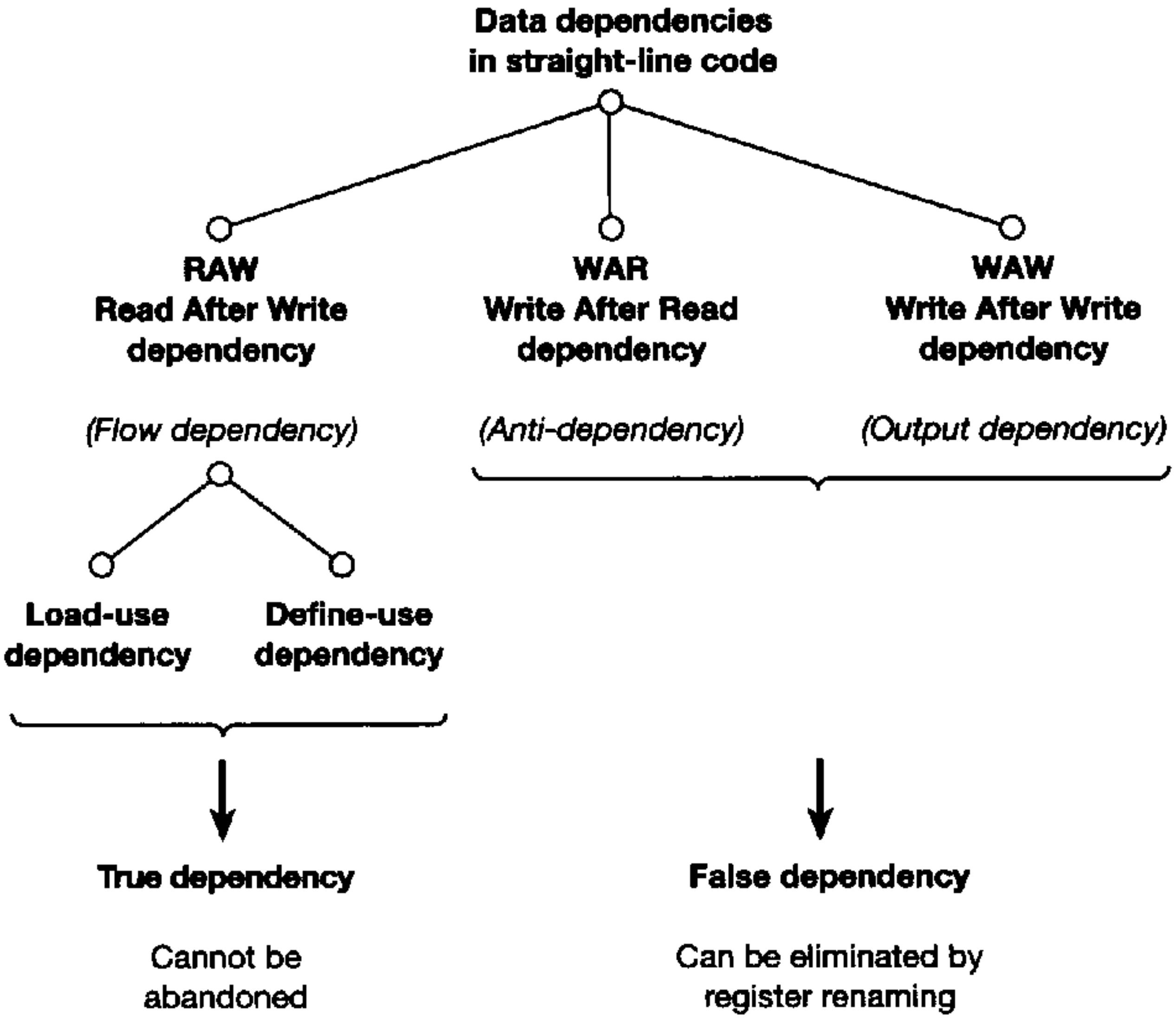
and may involve register data or memory data.


Data dependency
Type
Data involved
Data dependencies in straight-line code
Data dependencies in loops
Register data
Memory data
They exist in a straight-line code segment between subsequent instructions
They exist in a loop between instructions belonging to subsequent iterations
RAW/WAR/WAW dependencies
Recurrences

Data Dependencies in Straight-line Code

1. Read-after-Write (RAW) dependencies, also known as *flow dependencies*, are true dependencies since they cannot be circumvented. They can be either a
   - *load-use dependency*
   - *define-use dependency*
2. Write-after-Read (WAR) dependencies are also termed *anti-dependencies*.
3. Write-after-Write (WAW) dependencies are also termed *output dependencies*.

Both WAR and WAW dependencies are false dependencies and can be eliminated by renaming the destination register of the affected instruction to an unused one.


Data dependencies
in straight-line code
RAW
Read After Write
dependency
WAR
Write After Read
dependency
WAW
Write After Write
dependency
(Flow dependency)
(Anti-dependency)
(Output dependency)
Load-use
dependency
Define-use
dependency
True dependency
Cannot be
abandoned
False dependency
Can be eliminated by
register renaming

Data Dependencies in Loops

Recurrence is where an instruction belonging to a particular iteration is data dependent on an instruction of a previous loop iteration.

- The terms *inter-iteration data dependency* or *loop-carried dependency* are also used.
- A recurrence is said to be **of kth order**, if the associated computation needs results which have been computed in the preceeding **k** iterations.
- *First order linear recurrences* are the most common with the form $X(i) = A(i) * X(i-1) + B(i)$

[A directed graph expressing data dependencies is usually called a Data Dependency Graph.]

Control Dependencies

- Can seriously impede the performance of parallel instruction execution.
- Typically, 20% of instructions in general purpose programs (gpps) and 5-10% in scientific ones (sps) are conditional branches (based on published data).
- Increased parallel execution requires the processor to raise the issue rate, thus the higher the probability of encountering a conditional control dependency in each cycle.
- Techniques to reduce the effects of control dependencies has become a research field of it own.

[A directed graph expressing control dependencies is usually called a Control Dependency Graph.]

Resource Dependencies

An instruction is *resource-dependent* on a previously issued instruction if it requires a hardware resource which is still being used by that instruction.

Thus, resource dependencies are constraints caused by limited resources and which reduce the degree of parallelism that can be achieved.

With the evolution of integrated circuit technology, more and more hardware resources are becoming available in processors and related constraints will impede performance to a lesser extent than at present.

Dependency Detection and Resolution

Dependency Detection and Resolution is a means to extract *parallelism*. The two basic approaches are termed static and dynamic:

- *Static* detection and resolution is accomplished by the *compiler*, which avoids dependencies by reordering the code and, if necessary, inserting null instructions to produce dependency-free code.

- Dynamic detection and resolution of dependencies is performed by the processor.

This is how instruction-level functional parallelism, which is implicit, is made explicit.

Code Reordering

The performance of an ILP-processor can be significantly increased by parallel code optimisation.

Parallel optimisation goes beyond traditional sequential code optimisation. It is achieved by *reordering* the sequence of instructions by appropriate code transformations for parallel execution.

Thus, *ILP-instruction scheduling* encompasses both the *detection and resolution of dependencies* and *parallel code optimisation.*

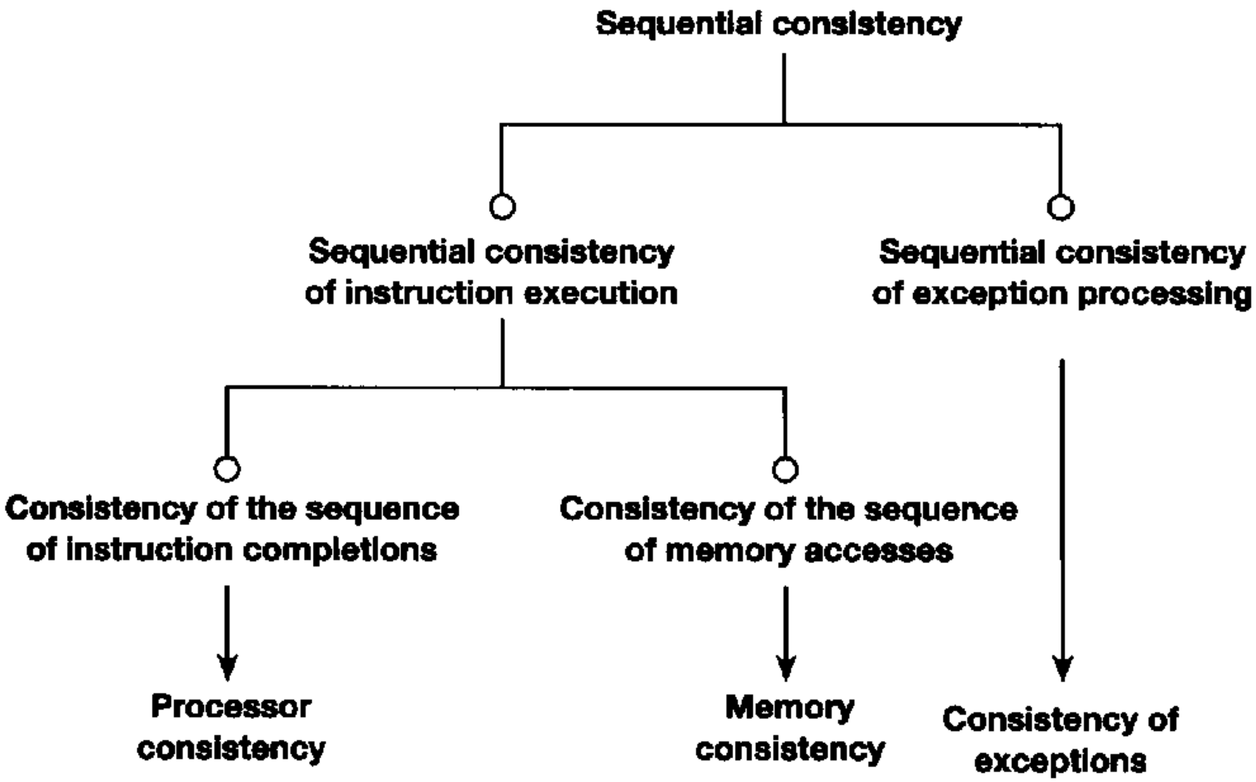
Basic Approaches Scheduling

ILP-instruction scheduling

Static scheduling boosted by parallel code optimization	**Dynamic scheduling without static parallel code optimization**	**Dynamic scheduling boosted by static parallel code optimization**
• Performed entirely by the compiler	• Performed entirely by the processor	• Performed by the processor in conjunction with the parallel optimizing compiler
• The processor receives dependency-free and optimized code for parallel execution	• The code is not optimized for parallel execution. The processor detects and resolves dependencies on its own	• The processor receives optimized code for parallel execution, but it detects and resolves dependencies on its own
Typical for VLIWs and a few pipelined processors (e.g. MIPS)	*Early ILP processors (e.g. CDC 6600, IBM 360/91 etc.)*	*Usual practice for pipelined and superscalar processors (e.g. RS/6000 and PL.8/XL, i960CA and QTC SF960)*

Preserving Sequential Consistency

The logical integrity of the program execution must be maintained when instructions are executed in parallel.



Speed-up Potential of ILP-processing

It is important to determine how far traditional sequential computation can be sped up by means of ILP-compilers and ILP-processors, i.e. to determine the extent to which instruction-level parallelism is available in programs.

When considering potential speedup, assume that:

- sufficient hardware resource are available so that resource constraints can be ignored; and
- through register renaming, false dependencies are eliminated.

[Numerous papers have been published on this topic since the early 1970s.]

Speed-up Potential$_{(cont)}$

- Early results revealed very low potential speedups – Averaging 2 for gps and 4 for sps.
- The reason for such disappointing results was that parallelism was only extracted from individual *basic blocks*.
- Typical average block sizes are around 6 (although somewhat higher for scientific programs).
- However, two possibilities remain. Utilising the potential speedup embodied in loops and the efficient handling of control dependencies.

Speed-up Potential$_{(cont)}$

- Loops are a promising source of parallelism when exploited by proper techniques (such as *Software pipelining*).

- For example, potential speed-up for 14 Livermore Loops averages 50. But the resource costs are about 50 processors and 400 registers.

- Using a perfect oracle (i.e. a processor that predicts branches perfectly, removes false dependencies and has sufficient hardware resources) speed-ups of the order of 10 to 100 (and higher in some cases) can be achieved for various benchmark programs.

Speed-up Potential$_{(cont)}$

- A crucial question is how close can *real systems* come to the upper limits of speed-up.
- Wall suggests ambitious systems (e.g. those with conditional branch prediction, 512 registers, register renaming, memory disambiguation and a 64 entry instruction window) can expect to achieve speed-up figures of about 4 for gps programs and around 10-20 for sps.
- Wall also showed that false dependency removal has a significant impact on increasing the speed-up.

D.W. Wall. Limits of instruction level parallelism. In *Proceedings of APLOS IV*, pages 176–188.

Summary

- Evolution of Instruction-Level Parallel (ILP) processors.
- ILP is limited by dependencies between instructions.
- In order to decide what can be executed in parallel, one must detect what cannot.
- It is important to determine the amount of ILP in typical applications.
- There seems to be enough potential parallelism to make it worthwhile designing ILP architectures.

A pergunta é "Como?" .....