Instruction-Level Parallelism and Superscalar Processors

Chapter 16

Joannah Nanjekye

August 06, 2024

Pipeline Performance with Stalls

From Lecture 17 we can derive more performance equations:

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

Pipeline Performance with Stalls

Pipelining is decreasing the CPI or the clock cycle time. The ideal CPI on a pipelined processor is always 1

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

If we ignore the cycle time overhead of pipelining and assume the stages are perfectly balanced, then the cycle time of the two processors can be equal:

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

Pipeline Performance with Stalls

If all instructions take the same number of cycles, which must also equal the number of pipeline stages, then the unpipelined CPI is equal to the depth of the pipeline:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If pipelining improves the clock cycle time, then we can calculate the CPI of the unpipelined processor, as well as the pipelined processor:

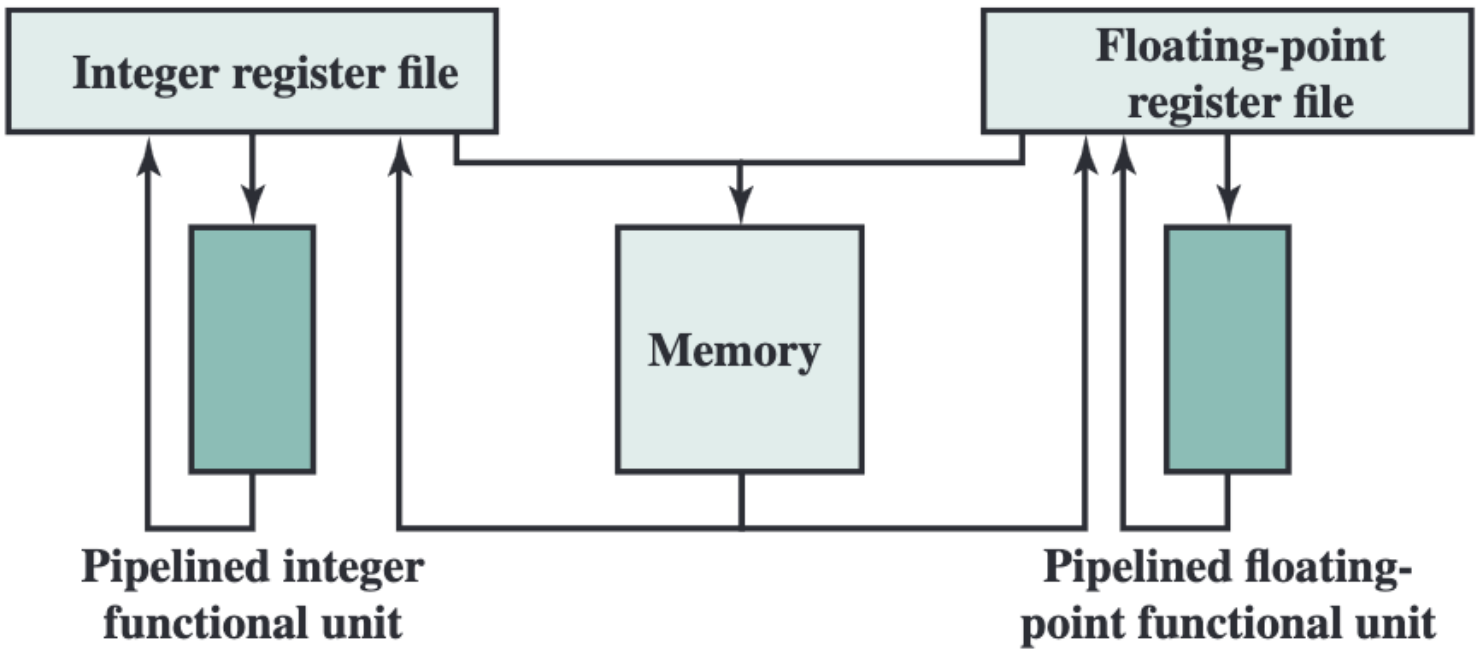
$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

- The instruction execution rate, CPI, will be less than 1, so instead we use IPC: instructions per clock cycle
- E.g., a 3 GHz, four-way multiple-issue processor can execute at a peak rate of 12 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4

If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?

Scalar processor

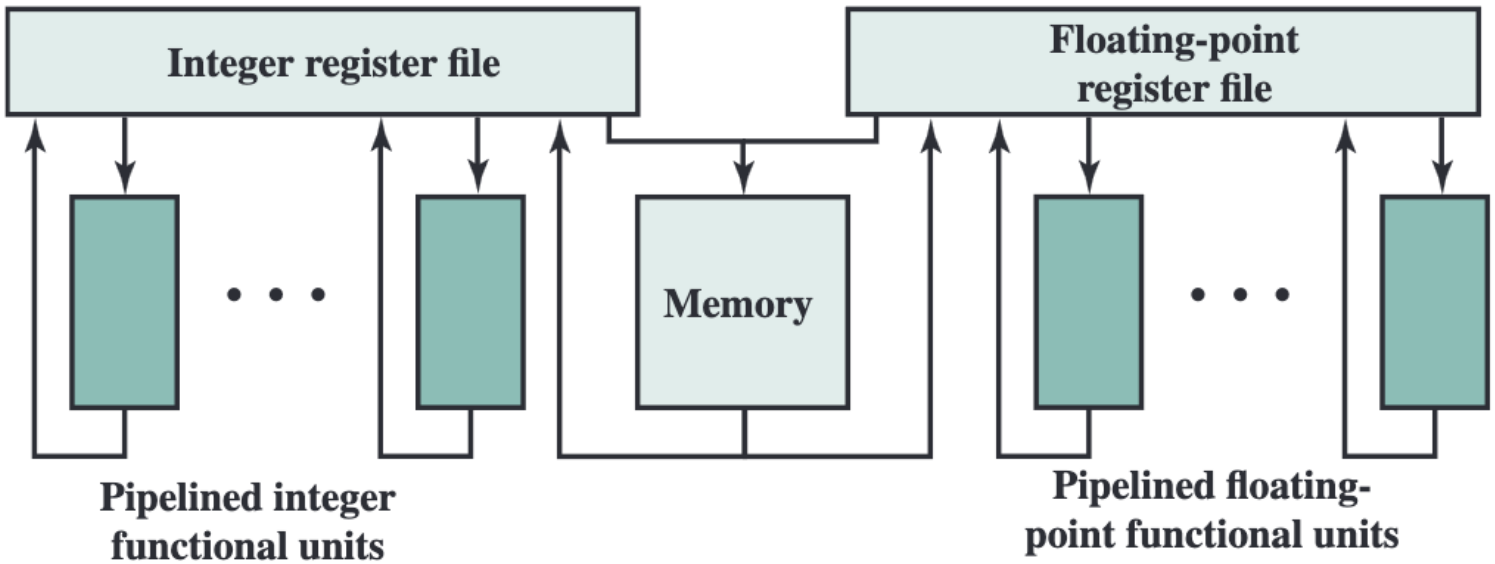
- A single pipelined functional unit exists for operations
- Pipelines allow for performance increases through parallelism
- Parallelism is by enabling multiple instructions to be at different stages of the pipeline



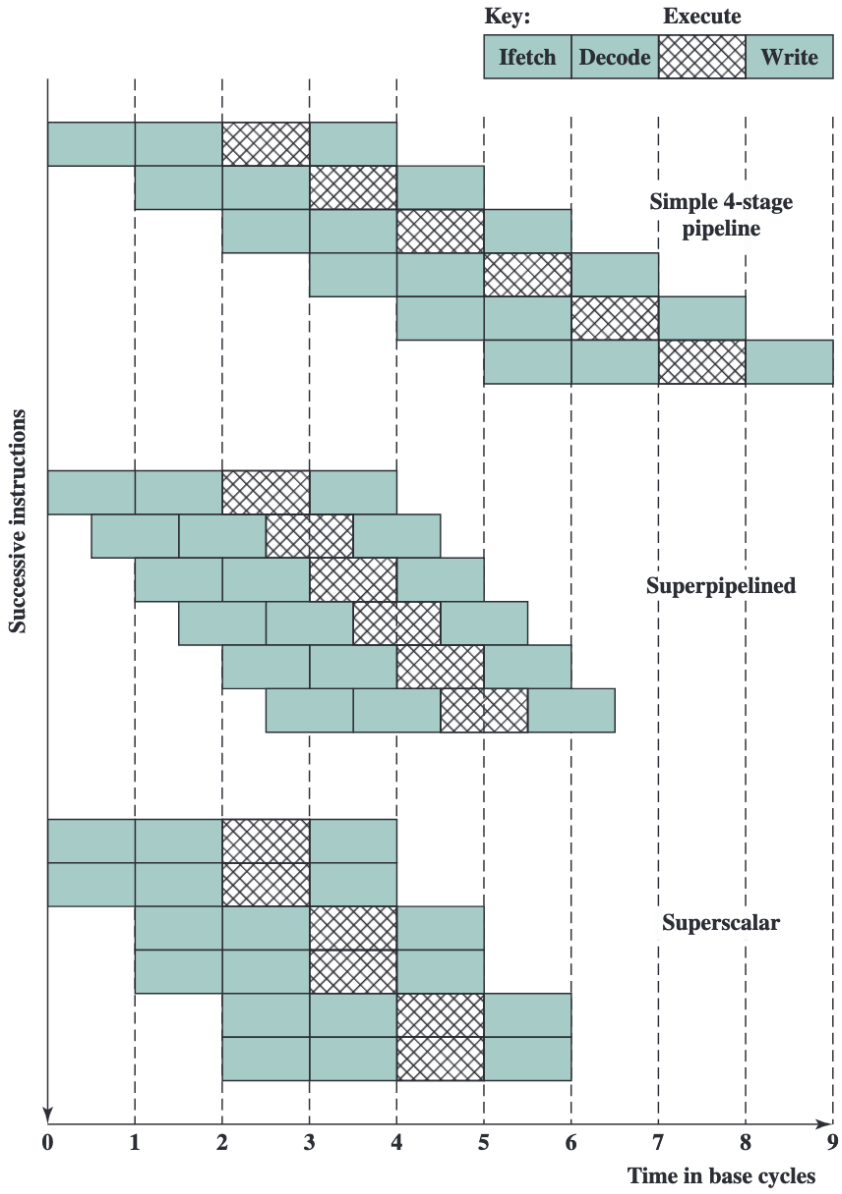
Superscalar processor

A processor that is designed to:

- Improve the performance of the execution of scalar instructions
- To have multiple functional units, implemented as a pipeline
- To execute instructions in different pipelines independently and concurrently



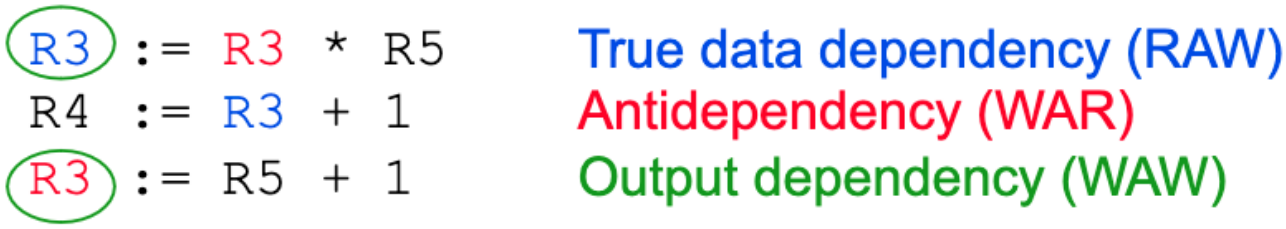
Superscalar versus Superpipelined



Constraints

- True data dependency
- Procedural dependency
- Resource conflicts
- Output dependency
- Antidependency

Data Hazards



True dependency (RAW)

Later instruction using a value (not yet) produced by an earlier instruction

Antidependencies (WAR)

Later instruction (that executes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that executes later)

Output dependency (WAW)

Two instructions write the same register or memory location

Example

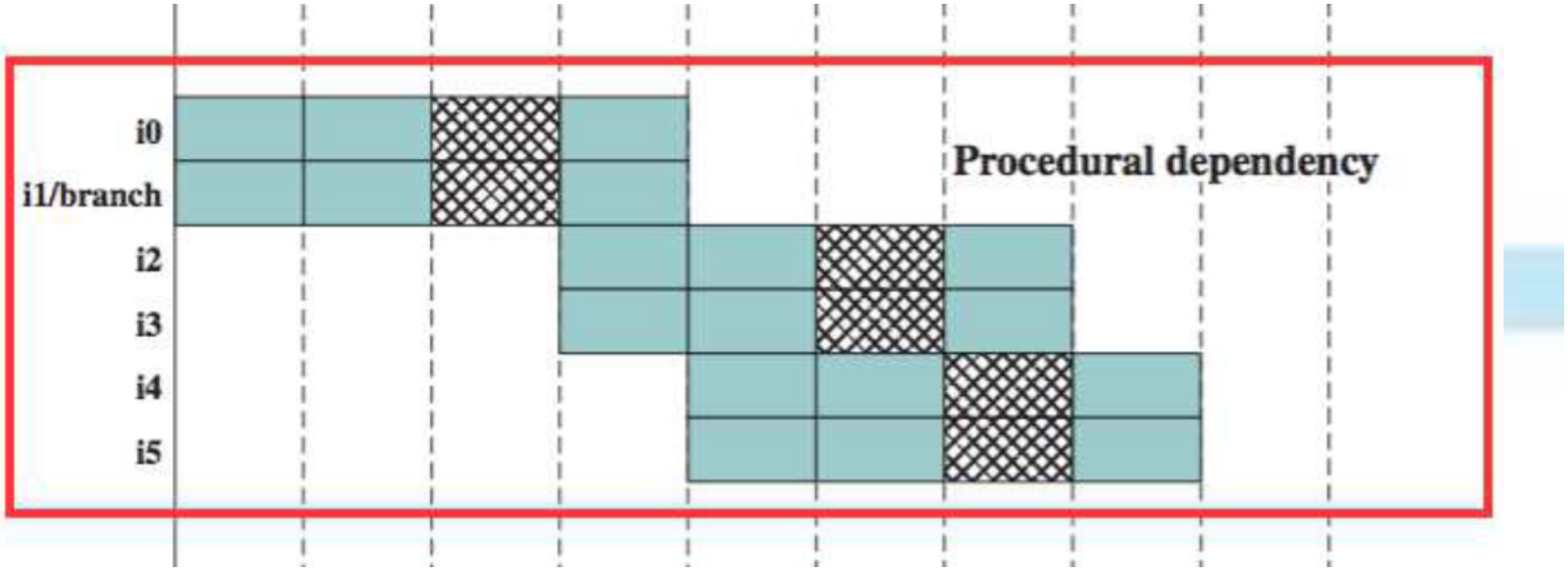
Find all data dependencies in this instruction sequence

```
I1: ADD R1, R2, R1
I2: LW  R2, 0(R1)
I3: LW  R1, 4(R1)
I4: OR  R3, R1, R2
```

Procedural Dependencies

Presence of branches complicates pipeline operation

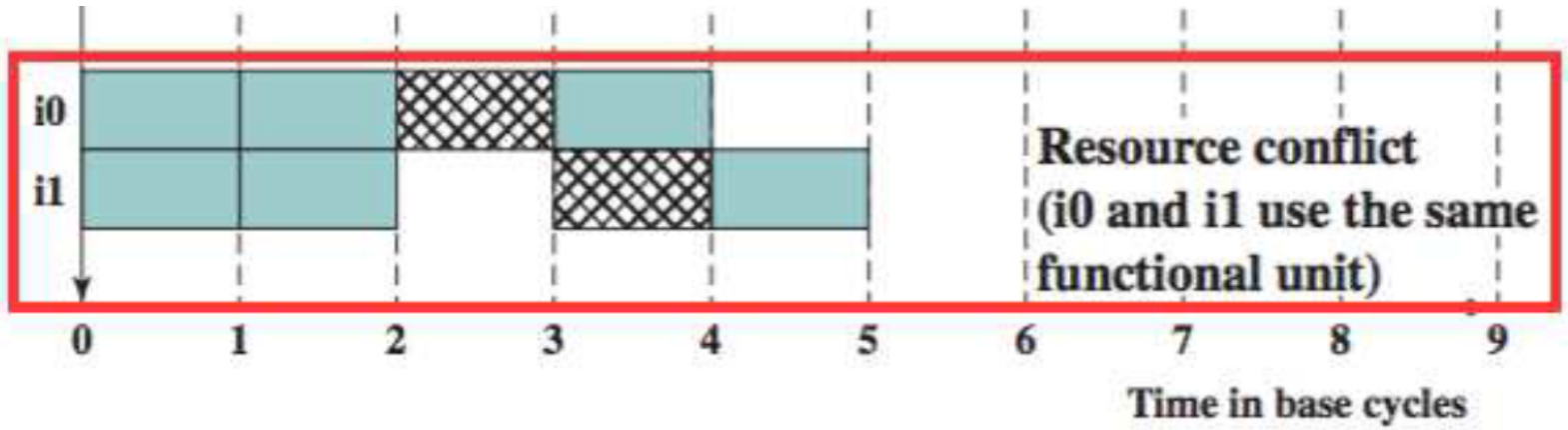
- Depend on whether the branch was taken or not taken
- This cannot be determined until the branch is executed
- This type of procedural dependency also affects a scalar pipeline



Resource Conflicts

Instruction competition for the same resource at the same time

- Resource examples include; bus, memory, registers, ALU
- Resource conflict exhibits similar behavior to a data dependency
- Resource conflicts can be overcome by duplication of resources
- While a true data dependency cannot be eliminated



Instruction-level Parallelism (ILP)

A measure of the average number of instructions in a program that a processor might be able to execute at the same time

- Mostly determined by the number of true (data) dependencies and procedural (control) dependencies
  - These are also dependent on the instruction set architecture and on the application
- And operation latency
  - The time until the result of an instruction is available for use as an operand in a subsequent instruction

```
Load R1 ← R2          Add R3 ← R3, "1"
Add R3 ← R3, "1"      Add R4 ← R3, R2
Add R4 ← R4, R2       Store [R4] ← R0
```

Machine Parallelism

A measure of the ability of the processor to take advantage of the ILP of the program. Determined by:

- The number of instructions that can be fetched and executed at the same time
- The speed and sophistication of the mechanisms that the processor uses to find independent instructions

Instruction Issue Policy

Look ahead of the current point of execution to locate instructions that can be brought into the pipeline and executed

- **Instruction issue:** process of initiating instruction execution in the processor's functional units
- **Instruction issue policy:** Mechanism to issue instructions

Three kinds of ordering are important:

- Instructions are fetched
- Instructions are executed
- Instructions update the contents of register and memory locations

The result must be correct

Multiple-Issue Processor Styles

- Static multiple-issue processors:
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - Fast runtime
  - Limited performance (variable values available when the program is running)
- Dynamic multiple-issue processors (superscalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8) are being made dynamically (at run time by the hardware)
  - Hardware penalty
  - Complete knowledge on the program

Techniques

- Hardware
  - Out-of-order issue
  - Register Renaming
- Software
  - Loop unrolling
  - Instruction scheduling

Superscalar Instruction Issue Policies

They can be grouped into the following categories:

- In-order issue with in-order completion
- In-order issue with out-of-order completion
- Out-of-order issue with out-of-order completion

In-order issue with in-order completion

- The simplest policy
- Issue instructions respecting original sequential execution (in-order issue)
- And write the results in the same order (in-order completion)
- This policy is used as baseline for comparing other robust approaches

In-order issue with in-order completion

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1					3
	I4			I3	I1	I2	4
I5	I6			I4			5
	I6		I5		I3	I4	6
			I6				7
					I5	I6	8

- Instructions fetched two at a time
- The next instructions must wait until the pair of decoder pipeline stages has cleared
- Instruction issuing is stalled by a functional unit conflict or delay

In-order issue with out-of-order completion

- Used in scalar RISC processors to improve the performance
- For instructions that require multiple cycles
- Any number of instructions may be in the execution stage
- Instruction issuing is stalled by a resource conflict, a data dependency, or a procedural dependency

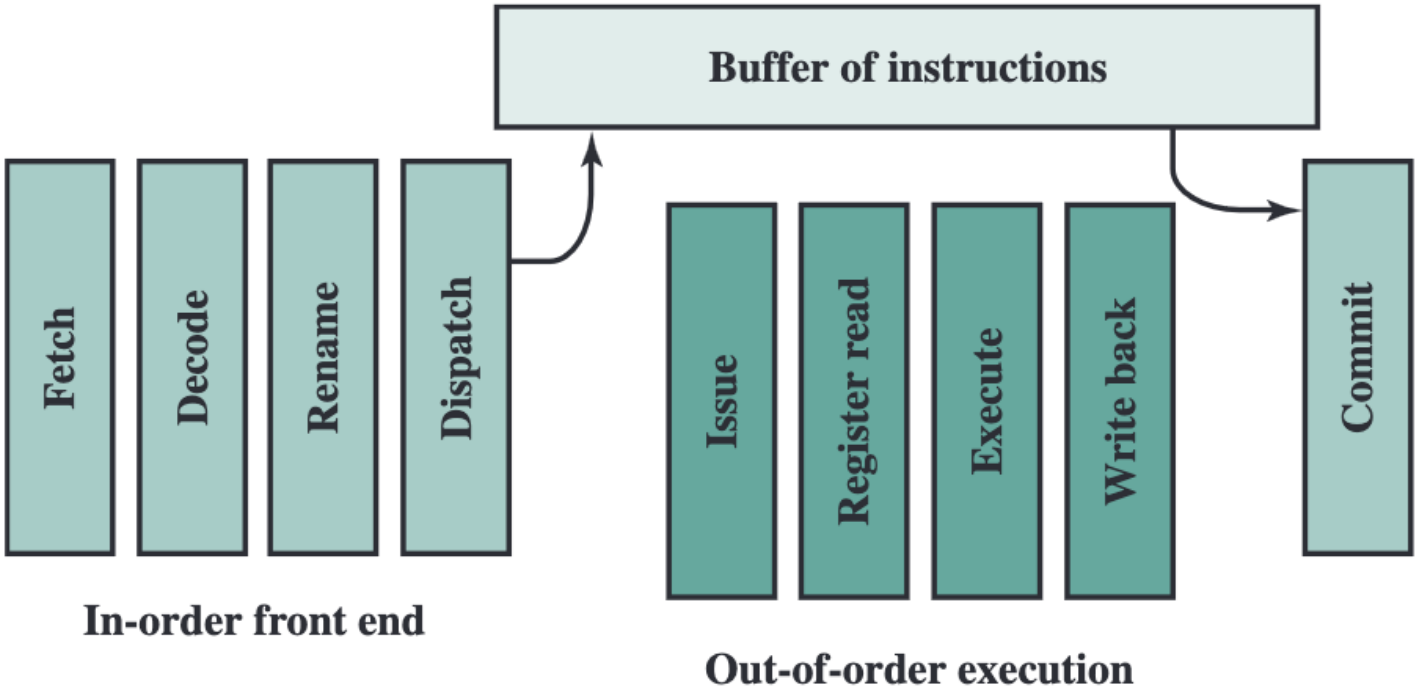
In-order issue with out-of-order completion

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
	I4	I1		I3	I2		3
I5	I6			I4	I1	I3	4
	I6		I5		I4		5
			I6		I5		6
					I6		7

- Instruction I2 is allowed to run to completion prior to I1
- This allows I3 to be completed earlier, with the net result of a savings of one cycle
- Requires more complex instruction issue logic than in-order completion
- Difficult to deal with interrupts and exceptions

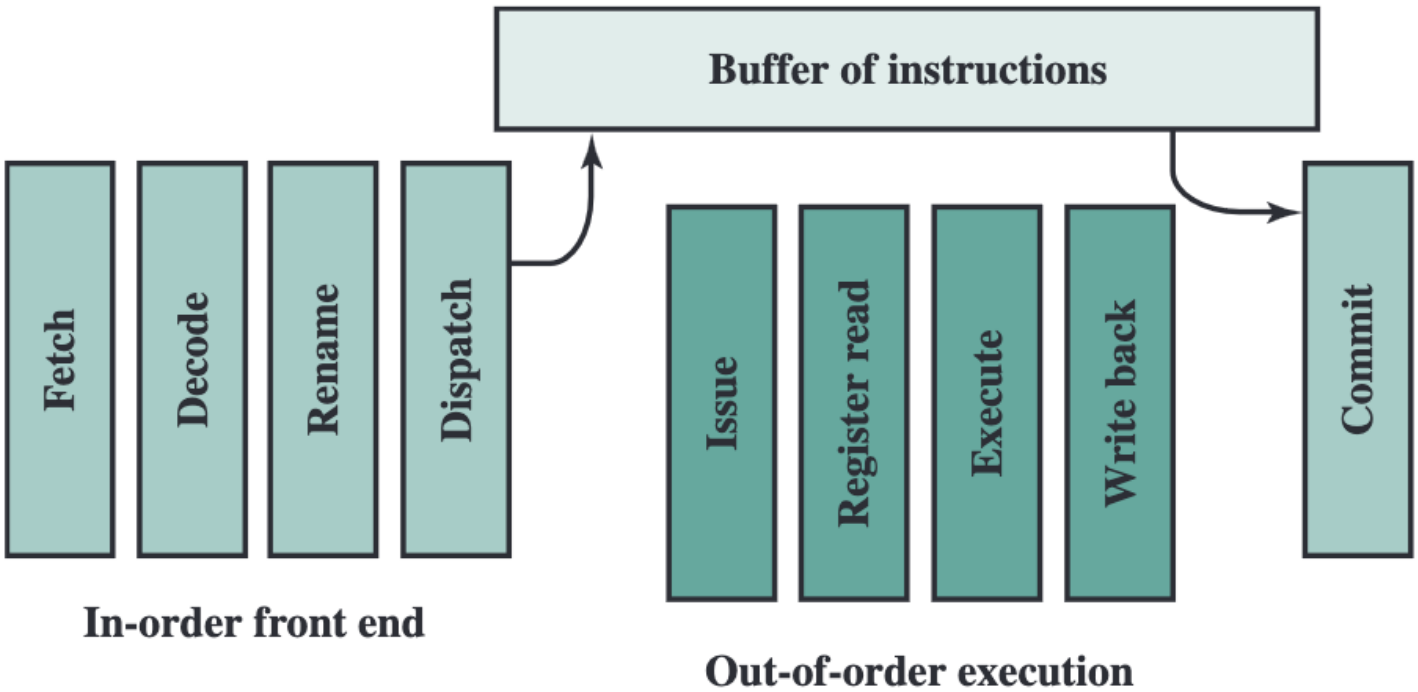
Out-of-order issue with out-of-order completion

- In-order issue stalls decoding until a conflict is resolved
- Requires to decouple the decode and execute stages of the pipeline
- Achieved with a buffer, **instruction window**



Buffer Operation

- Instructions are fetched and decoded if buffer is not full
- Instructions are moved to the execute stage if a functional unit is available
- Instruction is issued if:
  - It needs the available functional unit
  - There are no conflicts or dependencies



Out-of-order issue with out-of-order completion

- It is possible to issue instruction I6 ahead of I5 (recall that I5 depends on I4, but I6 does not)
- The pipeline also stalls if there is a dependency or conflict
- Chances of stalling are less since more instructions are available for issuing
- Out-of-order buffer can be supported with a reorder buffer

Decode		Window	Execute			Write		Cycle
I1	I2							1
I3	I4	*I1,I2*	I1	I2				2
I5	I6	*I3,I4*	I1		I3	I2		3
		I4,I5,I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

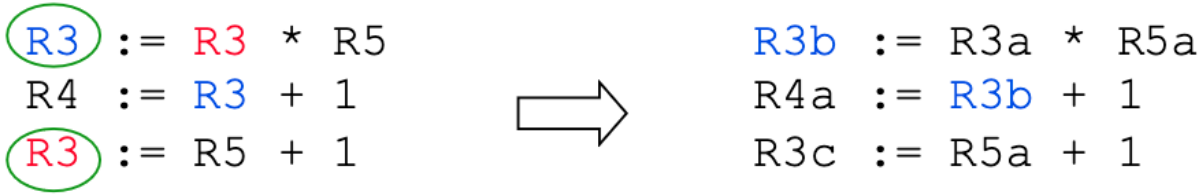
Register Renaming

Problem: Values in registers cannot be fully known at each point in time

- Main issues are WAW dependencies and WAR dependencies
- Differ from RAW data dependencies and resource conflicts, which reflect the flow of data
- WAW dependencies and WAR dependencies are due to values in registers nolonger reflecting the sequence of values dictated by the program flow

Resolve Storage Conflicts

Register Renaming: The processor renames the original register identifier in the instruction to a new register (one not in the visible register set)



- The hardware that does renaming assigns a ”replacement” register from a pool of free registers
- Releases it back to the pool when its value is superseded and there are no outstanding references to it

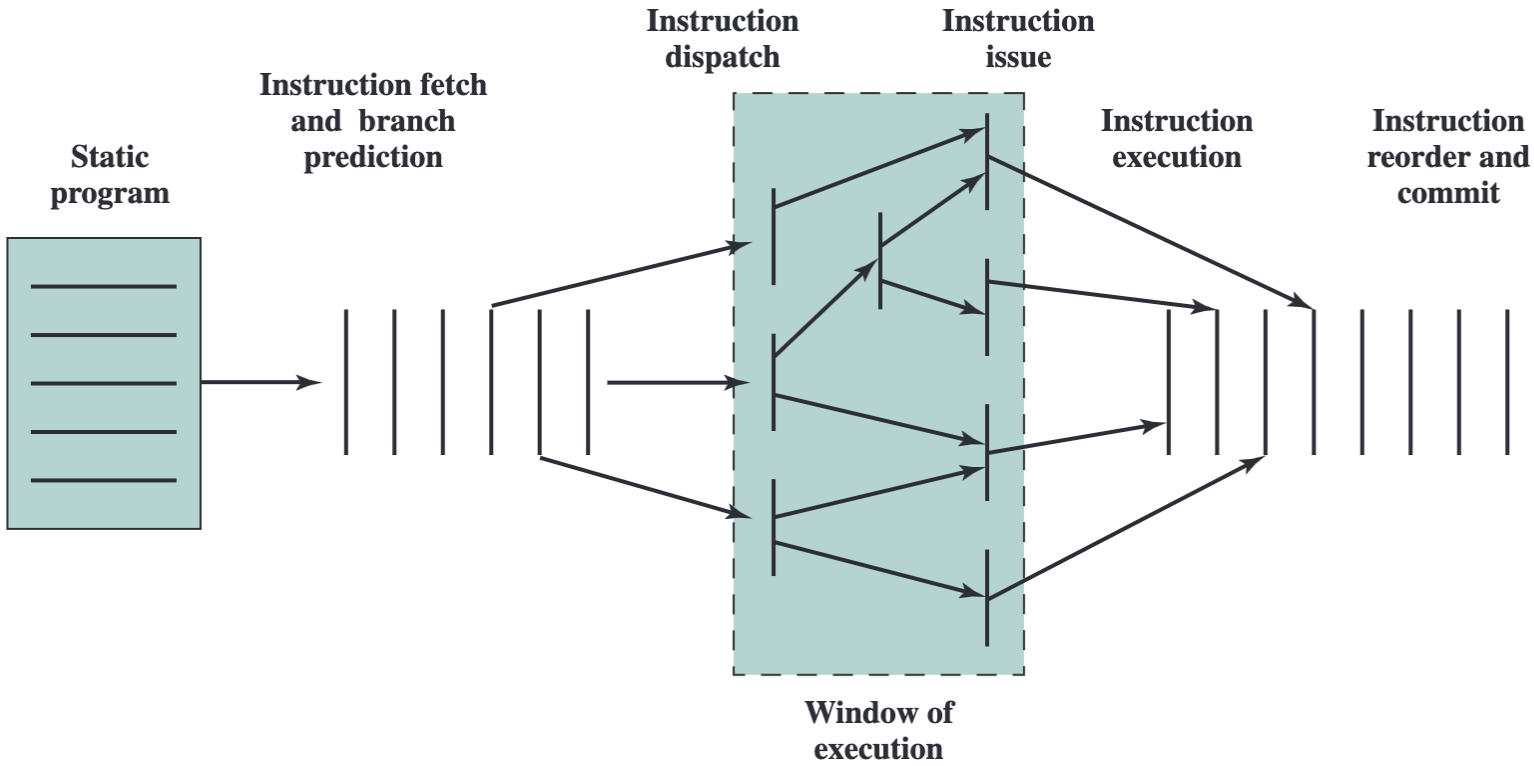
Resolve Control Dependency

Speculation: Allow execution of future instrâs that (may) depend on the speculated instruction

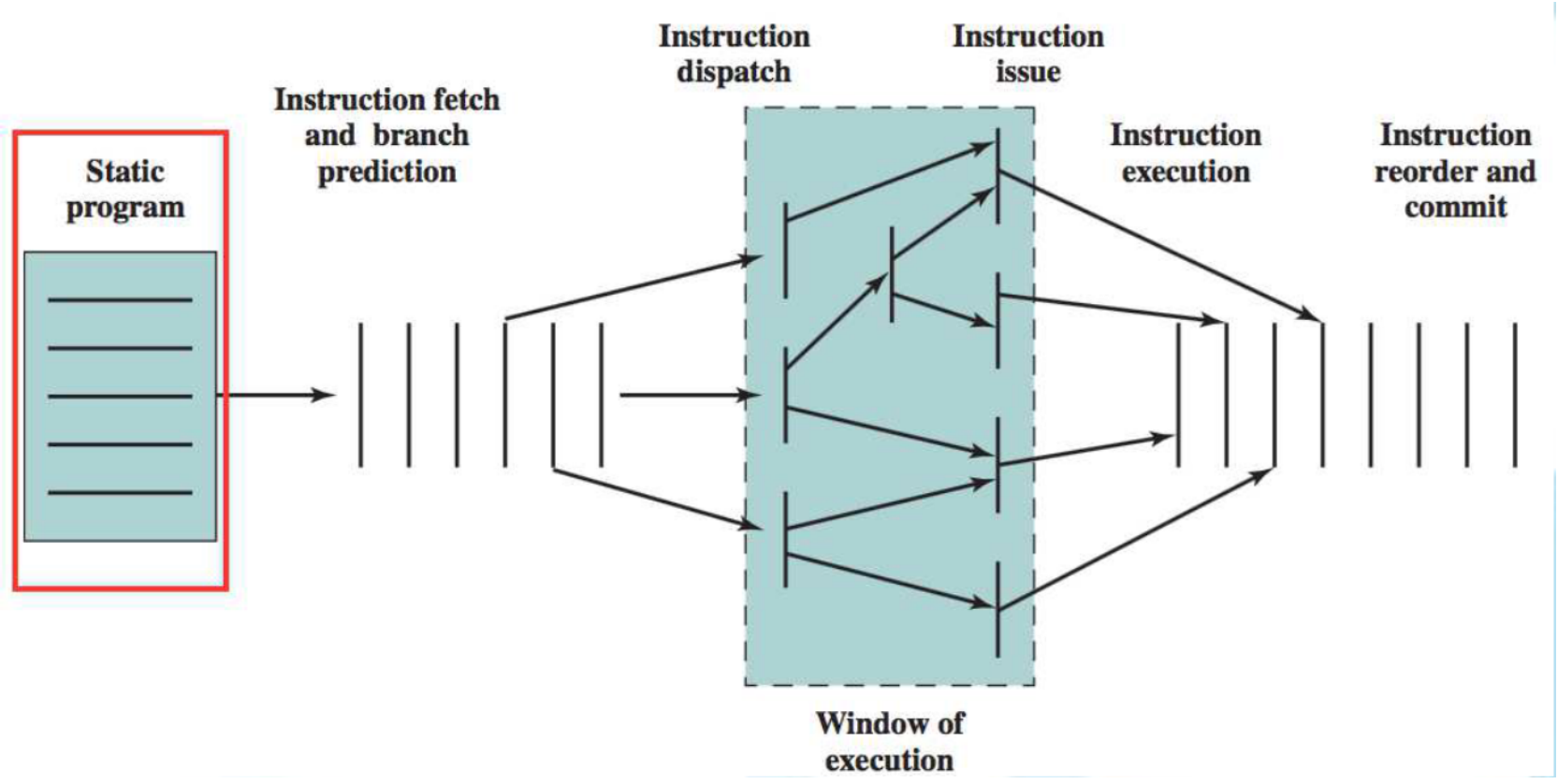
- Speculate on the outcome of a conditional branch **branch prediction**
- Speculate that a store (for which we donât yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store **load speculation**

  Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur

Superscalar Execution Overview

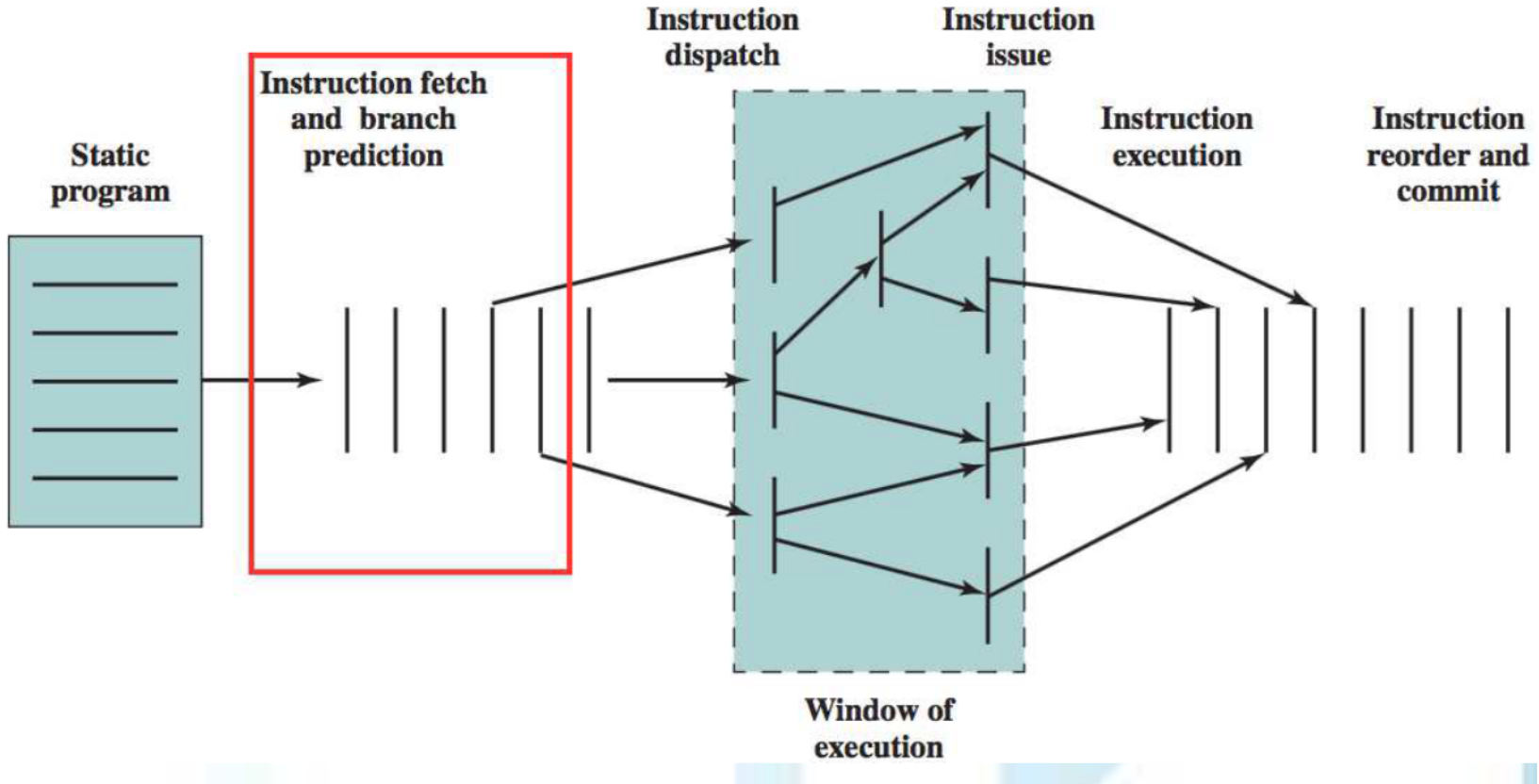


Superscalar Execution Overview



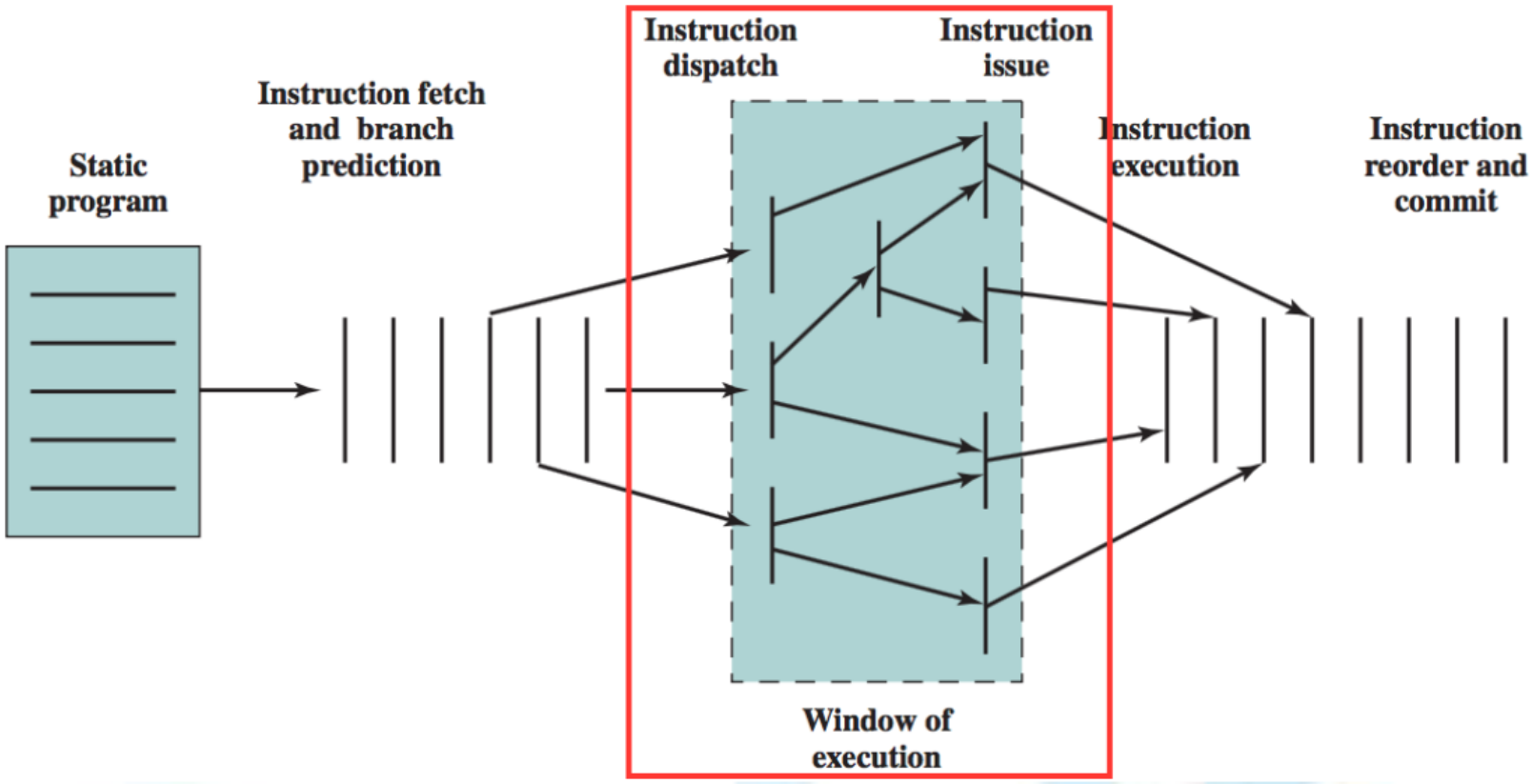
- The program to be executed consists of a linear sequence of instructions
- This is the static program as written by the programmer or generated by the compiler

Superscalar Execution Overview



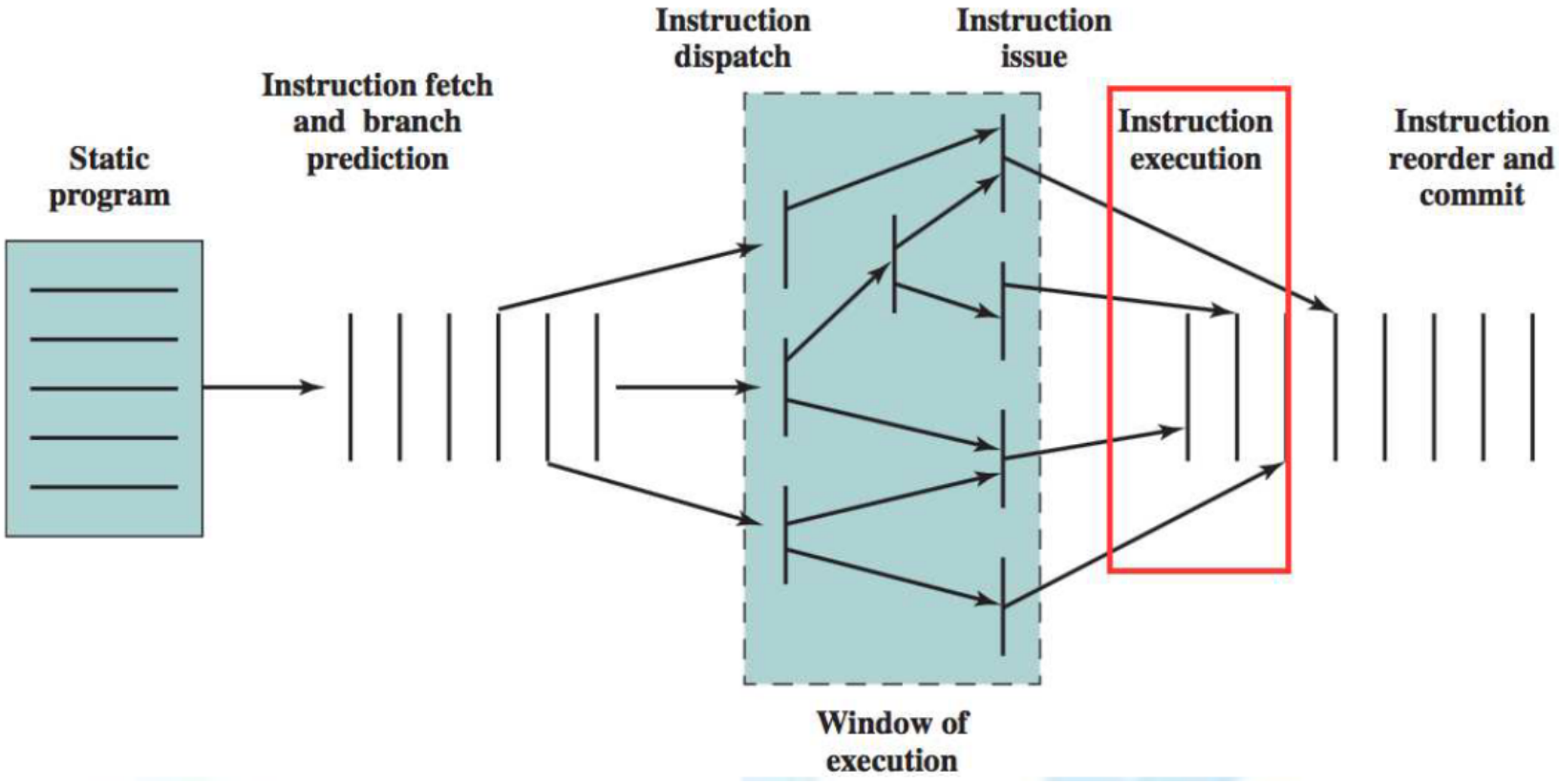
- Form a dynamic stream of instructions
- This stream is examined for dependencies, and the processor may remove artificial dependencies

Superscalar Execution Overview



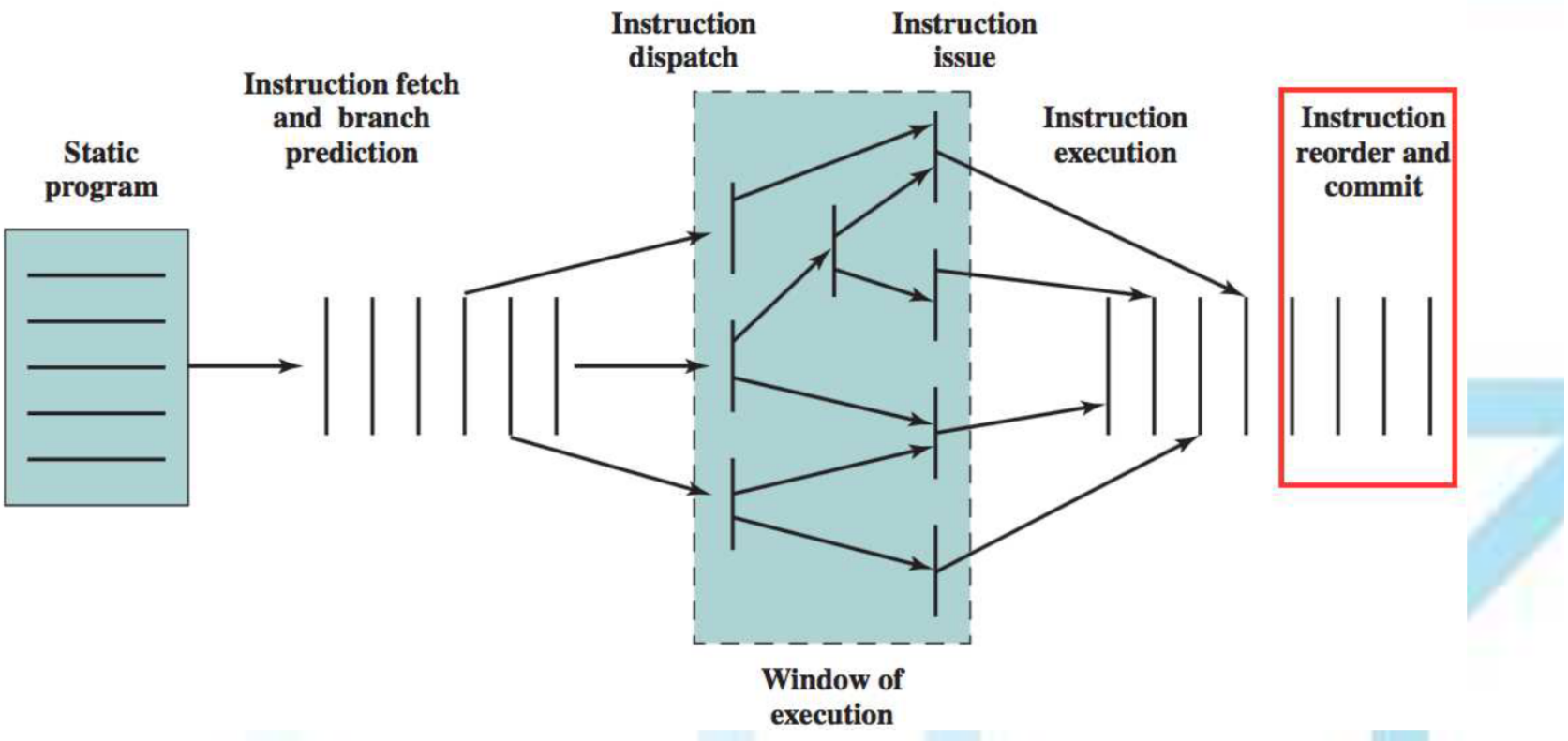
- Instructions are dispatched into a window of execution
- Instructions are structured according to their dependencies

Superscalar Execution Overview



- Instructions are executed in an order determined by:
  - True data dependencies
  - Hardware resource availability

Superscalar Execution Overview



- Instructions are put back into sequential order
- Their results are recorded

Sources Acknowledgement

- Course Textbook
- http://web.ist.utl.pt/luis.tarrataca/classes/computer_architecture/Chapter16-InstructionLevelParallelismAndSuperscalarProcessors.pdf
- http://www.cse.cuhk.edu.hk/ byu/CENG3420/2023Spring/slides/Lec16-ILP.pdf
- https://www.slideshare.net/slideshow/pipelining-and-ilp-instruction-level-parallelism/72975241