Chapter 16 - Instruction-Level Parallelism and Superscalar Processors

Luis Tarrataca
luis.tarrataca@gmail.com

CEFET-RJ

Table of Contents

Table of Contents

Scalar Processor

The first processors were known as scalar:

What is a scalar processor? Any ideas?

Scalar Processor

The first processors were known as scalar:

What is a scalar processor? Any ideas?

In a scalar organization, a single pipelined **functional unit** exists for:

- Integer operations;
- And one for floating-point operations;

Functional unit:

- Part of the CPU responsible for calculations;

Scalar Processor

In a scalar organization, a single pipelined **functional unit** exists for:

- Integer operations;
- And one for floating-point operations;

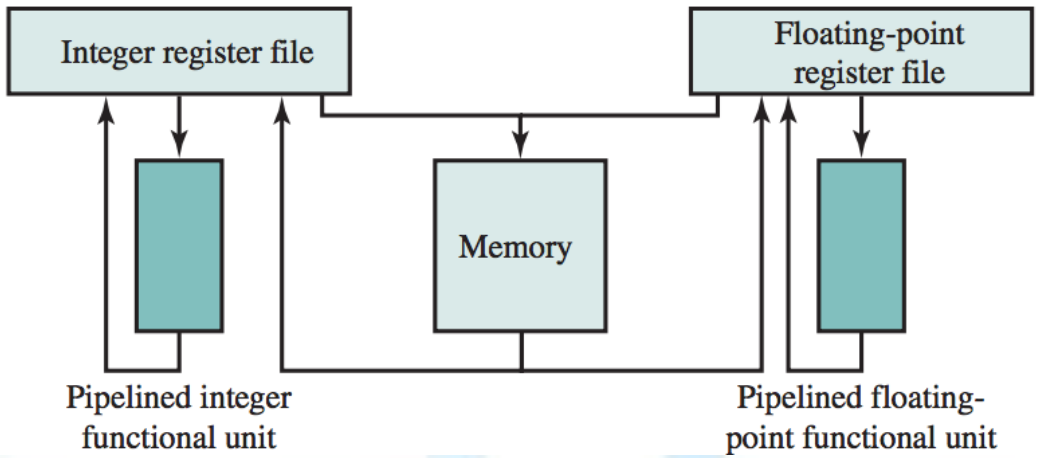


Figure: Scalar Organization (Source: (Stallings, 2015))

But why do we need to separate pipelines? Any ideas?

But why do we need to separate pipelines? Any ideas?

Integer and floating point processing is **different**:

- We studied this in Chapter 10 of the book;

Pipelines allow for performance increases through parallelism:

Remember how is parallelism achieved through a pipeline? Any ideas?

Pipelines allow for performance increases through parallelism:

Remember how is parallelism achieved through a pipeline? Any ideas?

Parallelism is achieved by:

- Enabling multiple instructions to be at different stages of the pipeline

Superscalar Processor

Term **superscalar** refers to a processor that is designed to:

- Improve the performance of the execution of scalar instructions;
- Represents the next evolution step;

Superscalar Processor

Term **superscalar** refers to a processor that is designed to:

- Improve the performance of the execution of scalar instructions;
- Represents the next evolution step;

How do you think this next evolution step is obtained? Any ideas?

Superscalar Processor

The term **superscalar** refers to a processor that is designed to:

- Improve the performance of the execution of scalar instructions;
- Represents the next evolution step;

How do you think this next evolution step is obtained? Any ideas?

- Simple idea: increase number of pipelines ;)

Superscalar processor

- Ability to execute instructions in different pipelines:
  - Independently and concurrently;

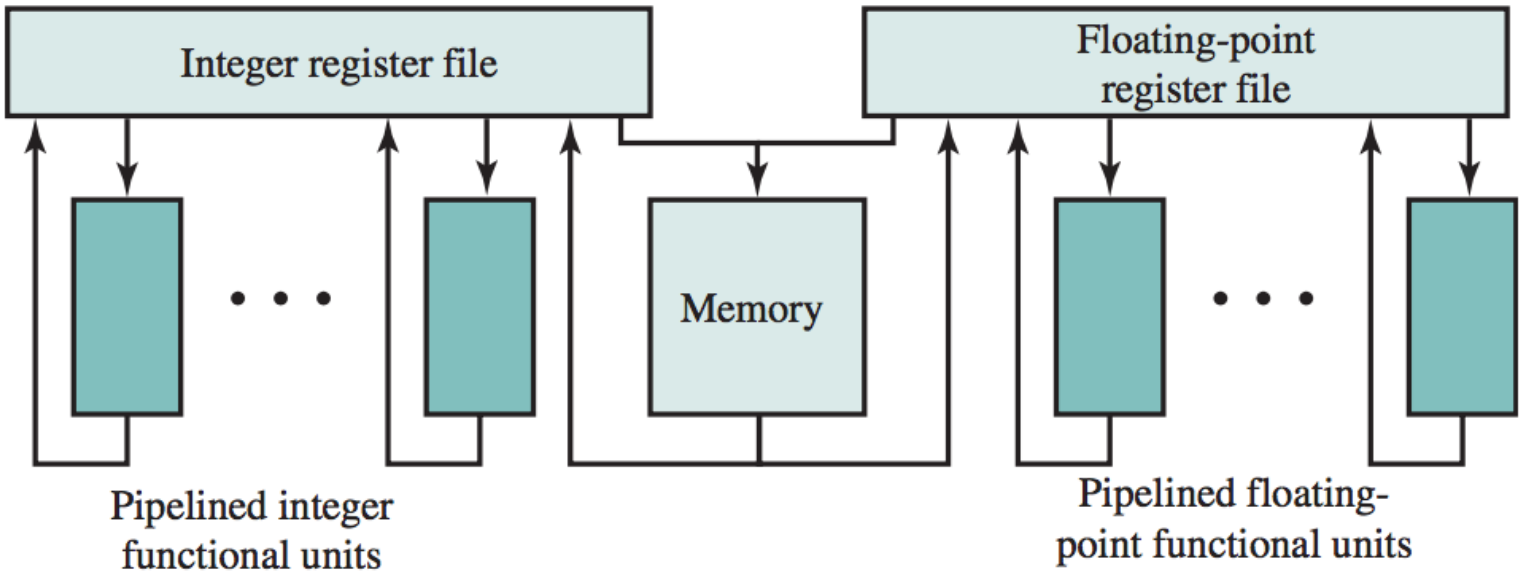


Figure: Superscalar Organization (Source: (Stallings, 2015))

However...

Pipeline concept already introduced some problems. Remember which?

However...

Pipeline concept already introduced some problems. Remember which?

- Resource Hazards;
- Data Hazards:
  - RAW
  - WAR
  - WAW
- Control Hazards;

Accordingly, how do we avoid some of the known pipeline issues?

But how do we avoid some of the known pipeline issues?

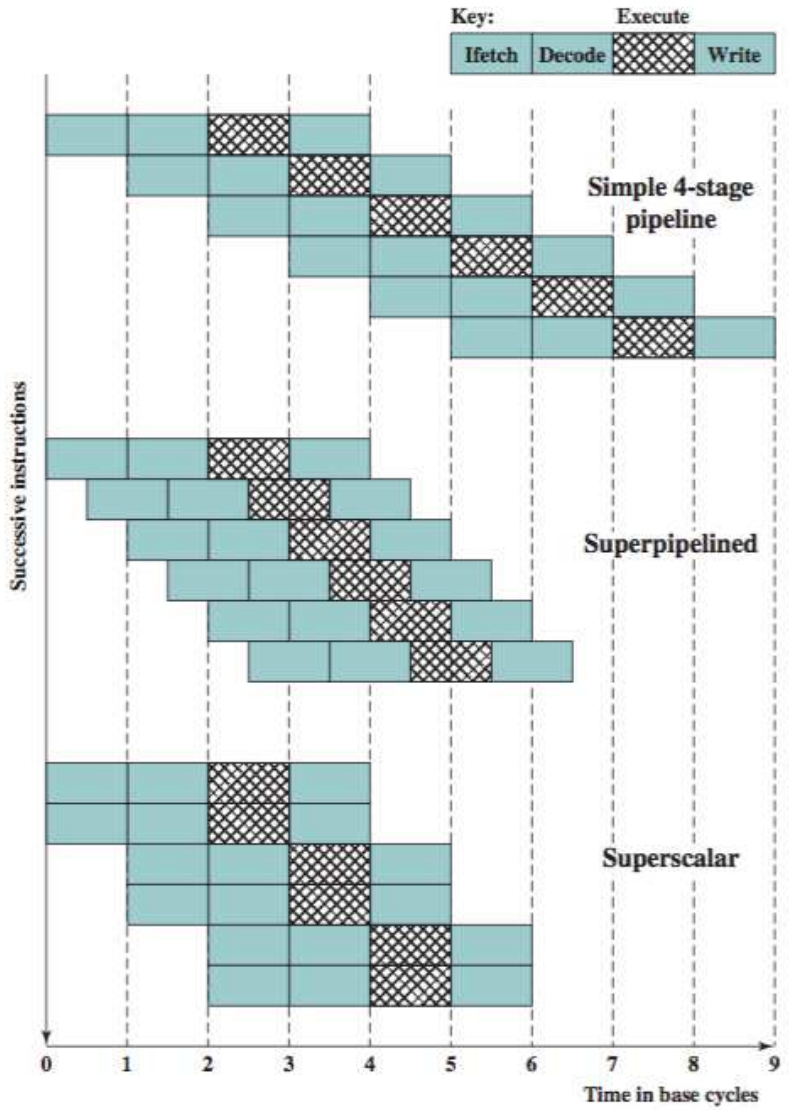
- Responsibility of the hardware and the compiler to:
  - Assure that parallel execution does not violate program intent;
  - Tradeoff between performance and complexity;

Superscalar vs. Superpipelined

Superpipelining is an alternative performance method to superscalar:

- Many pipeline stages require less than half a clock cycle;
- A pipeline clock is used instead of the overall system clock:
  - To advance between the different pipeline stages;

Consider the following execution scenario:



How much time is required for the normal pipeline approach? Why?

How much time is required for the superpipeline approach? Why?

How much time is required for the superscalar approach? Why?

Figure: Comparison of superscalar and superpipeline approaches (Source:

From the previous figure, **base pipeline**:

- Issues one instruction per clock cycle;
- Can perform one pipeline stage per clock cycle;
- Although several instructions are executing concurrently:
  - Only one instruction is in its execution stage at any one time.
- Total time to execute 6 instructions: 9 cycles.

From the previous figure, **superpipelined** implementation:

- Capable of performing two pipeline stages per clock cycle;
- Each stage can be split into two nonoverlapping parts:
  - With each executing in half a clock cycle;
- Total time to execute 6 instructions: 6.5 cycles.
  - Theoretical speedup: $1 - \frac{6.5}{9} \approx 28\%$

From the previous figure, **superscalar** implementation:

- Capable of executing two instances of each stage in parallel;
- Total time to execute 6 instructions: 6 cycles
  - Theoretical speedup: $1 - \frac{6}{9} \approx 33\%$

From the previous figure:

- Both the superpipeline and the superscalar implementations:
  - Have the same number of instructions executing at the same time;
  - However, superpipelined processor falls behind the superscalar processor:
    - Parallelism empowers greater performance;

Constraints

Superscalar approach depends on:

- Ability to execute multiple instructions in parallel;
- True **instruction-level parallelism**

However, parallelism creates additional issues:

- Fundamental limitations to parallelism

What are some of the limitations to parallelism? Any ideas?

What are some of the limitations to parallelism? Any ideas?

- Data dependency;
- Procedural dependency;
- Resource conflicts;

Lets have a look at these.

Data dependency

Consider the following sequence:

```
ADD EAX, ECX ;load register EAX with the con-
             ;tents of ECX plus the contents
             ;of EAX
MOV EBX, EAX ;load EBX with the contents of EAX
```

Figure: True Data Dependency (Source: (Stallings, 2015))

Can you see any problems with the code above?

Consider the following sequence:

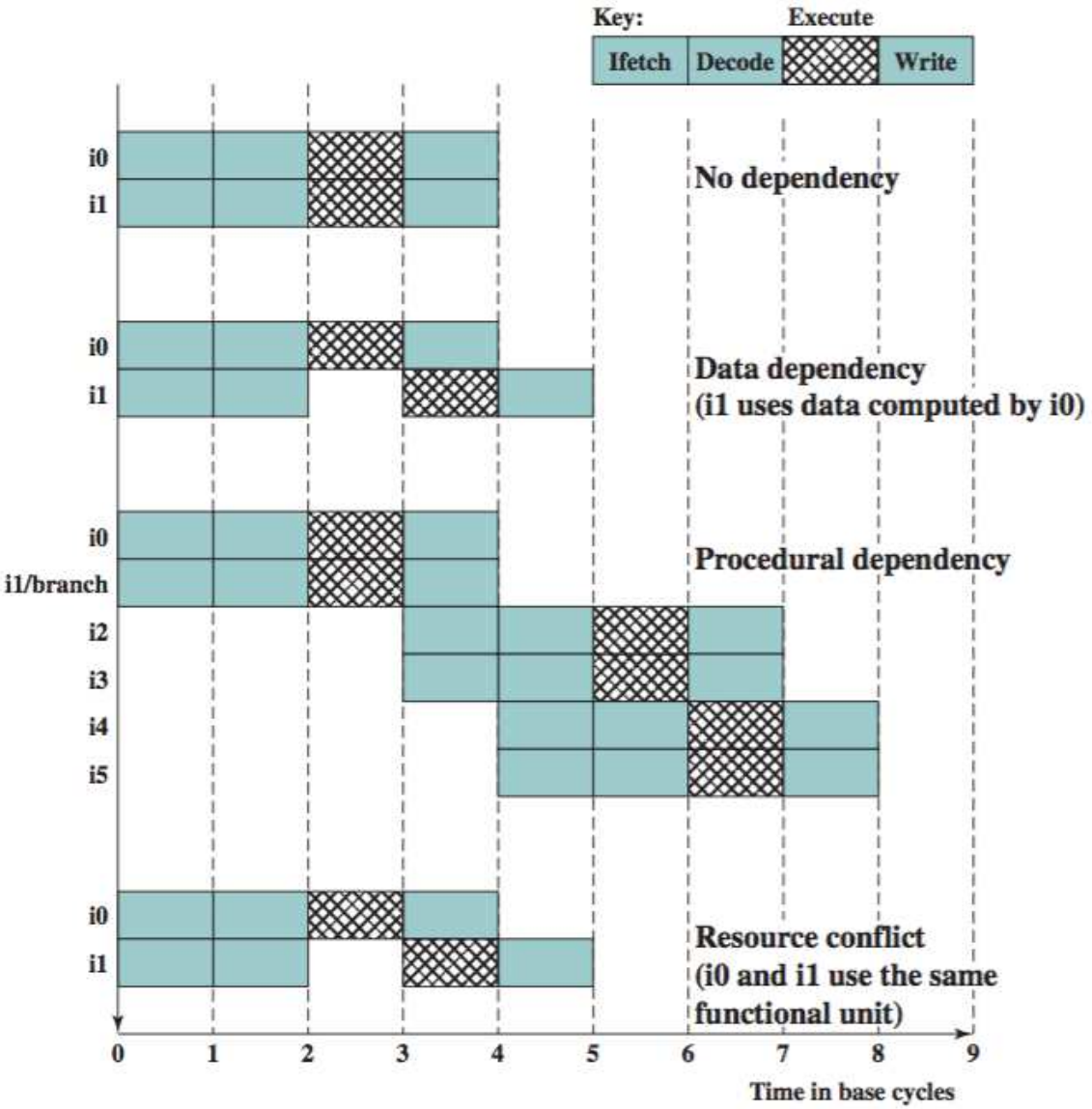
```
ADD EAX, ECX ;load register EAX with the con-
             ;tents of ECX plus the contents
             ;of EAX
MOV EBX, EAX ;load EBX with the contents of EAX
```

Figure: True Data Dependency (Source: (Stallings, 2015))

Can you see any problems with the code above?

- Second instruction can be fetched and decoded but cannot executed:
  - Until the first instruction executes;
- Second instruction needs data produced by the first instruction;
- A.k.a. read after write **RAW** dependency;

Example


Key:
Execute
Ifetch
Decode
Write
i0
i1
No dependency
i0
i1
Data dependency
(i1 uses data computed by i0)
i0
i1/branch
i2
i3
i4
i5
Procedural dependency
i0
i1
Resource conflict
(i0 and i1 use the same
functional unit)
0
1
2
3
4
5
6
7
8
9
Time in base cycles

From the previous figure:

- **With no dependency:**
  - two instructions can be fetched and executed in parallel;
- **Data dependency between the 1^{st} and 2^{nd} instructions:**
  - 2^{nd} instruction is delayed as many clock cycles as required to remove the dependency

In general:

> Instructions must be delayed until its input values have been produced.

Procedural Dependencies

Presence of branches complicates pipeline operation:

- Instructions following a branch:
  - 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:
    - More severe because a greater magnitude of opportunity is lost;

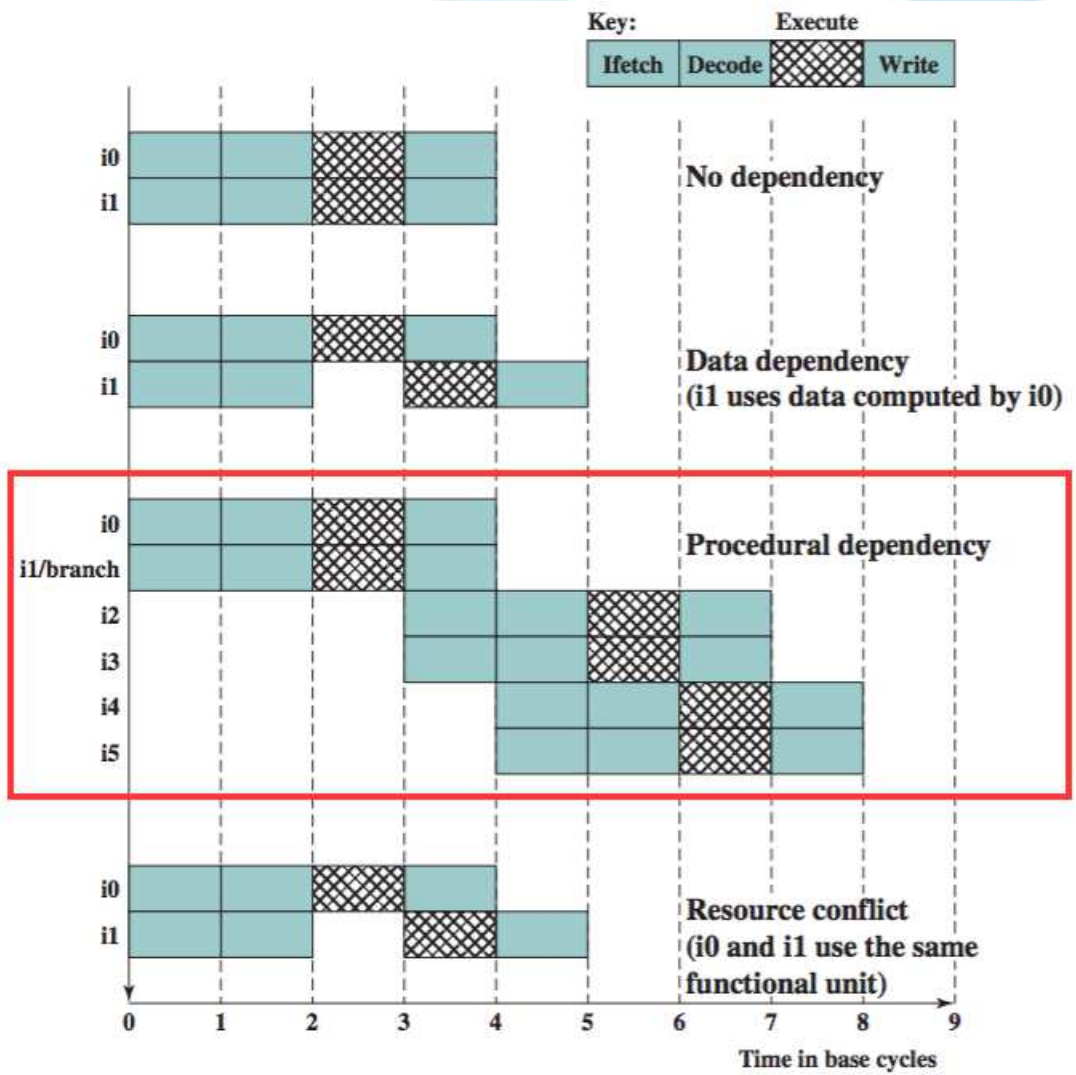


Figure: Effect of dependencies (Source: (Stallings, 2015))

Resource Conflict

Instruction competition for the same resource at the same time:

- Resource examples:
  - Bus;
  - Memory;
  - Registers;
  - ALU;
- Resource conflict exhibits similar behavior to a data dependency:
  - Resource conflicts can be overcome by duplication of resources:
    - whereas a true data dependency cannot be eliminated

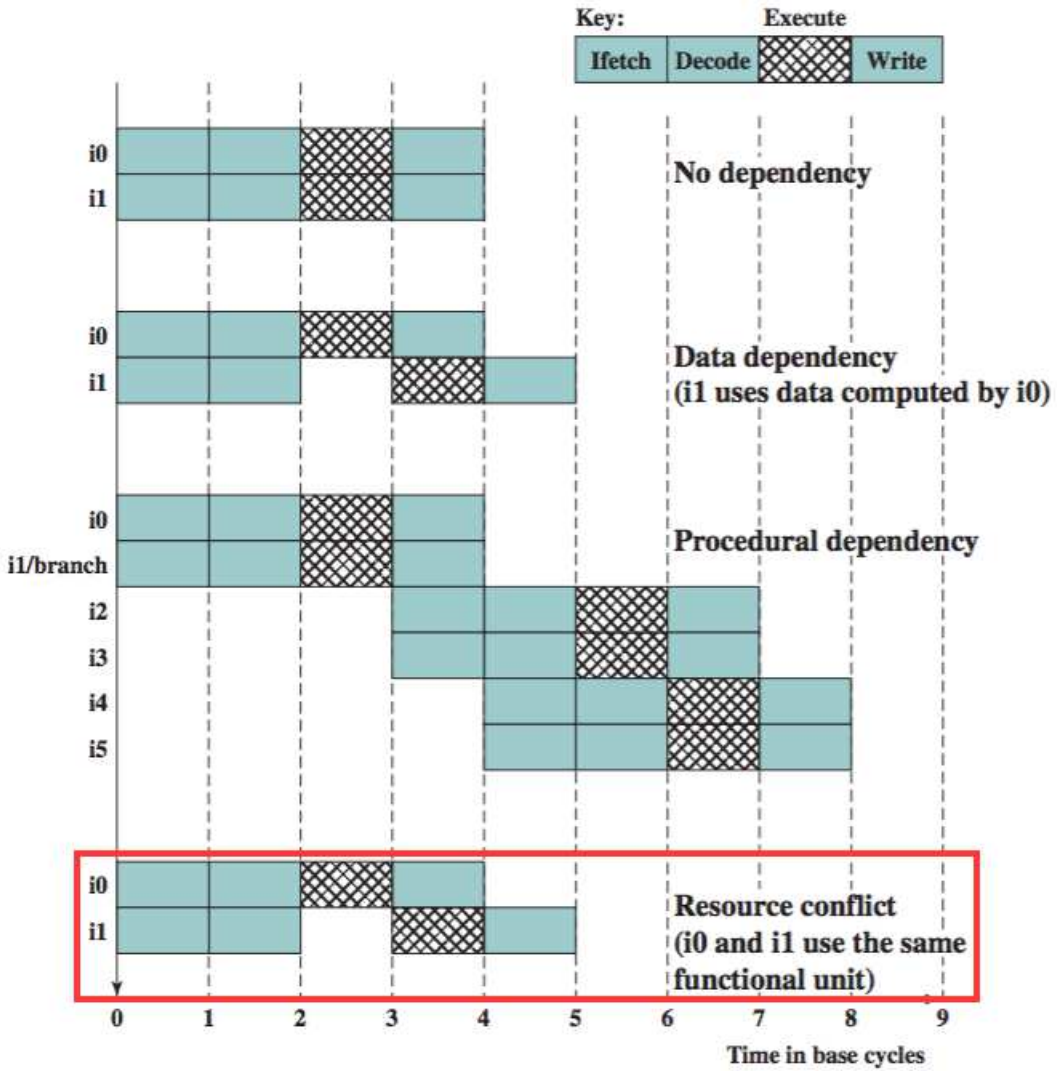


Figure: Effect of dependencies (Source: (Stallings, 2015))

Design Issues

Next, lets have a look at the different design issues to consider:

- Instruction-Level Parallelism and Machine Parallelism;
- Instruction Issue Policy;
- Register Renaming;
- Branch Prediction
- Superscalar Execution
- Superscalar Implementation

Important distinction:

What do you think is the difference between:

- Instruction-level parallelism?
- Machine-level parallelism?

Instruction-level parallelism

Instruction-level parallelism exists when instructions in a sequence:

- are independent and thus can be executed in parallel;

As an example consider the following two code fragments:

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

Figure: Instruction level parallelism (Source: (Stallings, 2015))

Instructions on the:

- Left are independent: can be executed in parallel;
- Right are dependent: cannot be executed in parallel;

Degree of instruction-level parallelism is determined by the

- Frequency of true data dependencies;
- Procedural dependencies (JMPs) in the code;

These depend on the instruction set and the application.

Machine Parallelism

Machine parallelism is a measure of the ability of the processor to:

- Take advantage of instruction-level parallelism:
  - Number of parallel pipelines;
- Determined by:
  - Number of instructions that can be fetched at the same time;
  - Number of instructions that can be executed at the same time;
  - Ability to find independent instructions.

Instruction Issue Policy

Processor must also be able to identify instruction-level parallelism:

- This is required in order to **orchestrate**:
  - Fetching, decoding, and execution of instructions in parallel;

In essence:

- Processor needs to locate instructions that can be pipelined and executed
- Goal: optimize pipeline usage;

In essence:

- Processor needs to locate instructions that can be pipelined and executed
- Goal: optimize pipeline usage;

What factors influence this ability to locate these instructions? Any ideas?

In essence:

- Processor needs to locate instructions that can be pipelined and executed
- Goal: optimize pipeline usage;

What factors influence this ability to locate these instructions? Any ideas?

Hint: Do we always need to execute instructions in the original sequential order?

In essence:

- Processor needs to locate instructions that can be pipelined and executed
- Goal: optimize pipeline usage;

What factors influence this ability to locate these instructions? Any ideas?

Hint: Do we always need to execute instructions in the original sequential order?

- No! As long the final result is correct!

Three types of orderings are important in this regard:

- Order in which instructions are fetched;
- Order in which instructions are executed;
- Order in which instructions update register/memory contents;

To optimize utilization of the various pipeline elements:

- Processor may need to alter one or more of these orderings:
  - Regarding the original sequential execution.
- This can be done: **as long as the final result is correct**;
- Therefore: we need to look at how instructions are issued:
  - This is known as: **instruction issue policies**;

Instruction issue policies fall 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

Without getting into much details for now:

> What do you think each one of these policies does? Any ideas?

Lets have a look at these

In-order issue with in-order completion

Simplest instruction issue policy:

- Issue instructions respecting original sequential execution:
  - A.k.a. **in-order issue**
- And write the results in the same order:
  - A.k.a. **in-order completion**

This instruction policy can be used as a **baseline**:

- for comparing more sophisticated approaches.

Consider the following example:

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

Figure: In-order issue with in-order completion (Source: (Stallings, 2015))

Assume a superscalar pipeline capable of:

- Fetching and decoding two instructions at a time;
- Having three separate functional units:
  - *E.g.:* two integer arithmetic and one floating-point arithmetic;
- Having two instances of the write-back pipeline stage;

Example assumes the following constraints on a six-instruction code:

- I1 requires two cycles to execute.
- I3 and I4 conflict for a functional unit.
- I5 depends on the value produced by I4.
- I5 and I6 conflict for a functional unit.

From the previous example:

- Instructions are fetched two at a time and passed to the decode unit;
- Because instructions are fetched in pairs:
  - Next two instructions waits until the pair of decode stages has cleared.
- To guarantee in-order completion:
  - when there is a conflict for a functional unit:
    - issuing of instructions temporarily stalls.
- Total time required is eight cycles.

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

Figure: In-order issue with out-of-order completion (Source: (Stallings, 2015))

- Instruction I2 is allowed to run to completion prior to I1;
- Allows I3 to be completed earlier, saving one cycle.
- Total time required: 7 cycles:
  - Speedup: $1 - \frac{7}{8} \approx 12,5\%$

With out-of-order completion:

- Any number of instructions may be in the execution stage at any one time:
  - Up to the maximum degree of machine parallelism across all functional units.
- **Instruction issuing is stalled by:**
  - Resource conflict;
  - Data dependency;
  - Procedural dependency.

Out-of-Order issue with Out-Of-Order Completion

With in-order issue:

- Processor will only decode instructions up to a dependency or conflict;
- No additional instructions are decoded until the conflict is resolved;
- As a result:
  - Processor cannot look ahead of the point of conflict;
  - Subsequent independent instructions that:
    - Could be useful will not be introduced into the pipeline.

To allow **out-of-order issue**:

- Necessary to decouple the decode and execute stages of the pipeline;
- This is done with a buffer referred to as an **instruction window**;

With this organization (1/2):

- Processor places instruction in window after decoding it;
- As long as the window is not full:
  - Processor will continue to fetch and decode new instructions;

With this organization (2/2):

- When a functional unit becomes available in the execute stage:
  - Instruction from the window may be issued to the execute stage;
  - **Any instruction may be issued, provided that:**
    - It needs the particular functional unit that is available;
    - No conflicts or dependencies block this instruction;

The result of this organization is that:

- Processor has a **lookahead** capability:

What do you think the term **lookahead** means? Any ideas?

The result of this organization is that:

- Processor has a **lookahead** capability:

> What do you think the term **lookahead** means? Any ideas?

  - Independent instructions that can be brought into the execute stage.
  - Instructions are issued from the window with little regard for original order:
    - No conflicts or dependencies must exist!
    - Then the program execution will behave correctly;

Lets consider the following example:

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

Figure: Out-of-Order issue with Out-Of-Order Completion (Source: (Stallings, 2015))

From the previous figure:

- During each of the first three cycles:
  - Two instructions are fetched into the decode stage;
  - Subject to the constraint of the buffer size:
    - Two instructions move from the decode stage to the instruction window.
- Note that in this example:
  - Possible to issue instruction I6 ahead of I5:
    - Recall that I5 depends on I4, but I6 does not;
  - Total execution time: 6 cycles
    - Speedup: $1 - \frac{6}{8} = 25\%$

Out-of-order issue out-of-order completion conclusions

In conclusion:

What do you think are the main differences between?

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

Out-of-order issue out-of-order completion conclusions

In conclusion (1/2):

What do you think are the main differences between?

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

Out-of-order issue out-of-order completion **still** needs to respect constrains:

- Instruction cannot be issued if it violates a dependency or conflict;
- Just like the In-order issue out-of-order completion policy;

So what is the difference then? Any ideas?

Out-of-order issue out-of-order completion conclusions

In conclusion (2/2):

So what is the difference then? Any ideas?

Difference is that more instructions are available for issuing:

- With In-order issue out-of-order completion the pipeline:
  - Stops issuing any more instructions as soon as a conflict is found!
- With Out-of-order issue out-of-order completion the pipeline:
  - Is still able to issue **othe**r instructions that don't have dependencies;
  - Reducing the probability that a pipeline stage will have to idle;

What are the main conclusions you can draw from instruction issue policies? Any ideas?

What are the main conclusions you can draw from instruction issue policies? Any ideas?

When instructions are issued in sequence and complete in sequence:

- Contents of each register are known at each point in the execution;

When out-of-order techniques are used:

- Values in registers cannot be fully known at each point in time;
- This causes WAR, WAR, RAW problems...
- Big confusion =’(

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

What do you think is the cause of this problem? Any ideas?

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

What do you think is the cause of this problem? Any ideas?

Cause: Multiple instructions competing for the use of the same registers:

- Generating pipeline constraints that slow performance.

Problem: Multiple instructions competing for the use of the same registers:

- Generating pipeline constraints that slow performance.

What would be a method for dealing with this problem?

Problem: Multiple instructions competing for the use of the same registers:

- Generating pipeline constraints that slow performance.

What would be a method for dealing with this problem?

- We could try to rename the registers ;)
- Essentially we are trying to resolve the issue by duplicating the resource;

Register Renaming

Processor registers need to be:

- Allocated dynamically by the processor hardware;
- Associated with the values needed by instructions at points in time;

When an instruction executes that has a register as a destination:

- New register is allocated for that value;
- Subsequent instructions that access the value in the register:
  - Need to refer to the allocated register;

Example

Consider the following code:

```
I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4
```

Figure: Register Renaming (Source: (Stallings, 2015))

How could the problems present be solved? Any ideas?

Example

$$
\begin{aligned}
&I1: \quad R3_b \leftarrow R3_a \ op \ R5_a \\
&I2: \quad R4_a \leftarrow R3_b + 1 \\
&I3: \quad R3_c \leftarrow R5_a + 1 \\
&I4: \quad R7_a \leftarrow R3_c \ op \ R4_a
\end{aligned}
$$

Figure: Register Renaming (Source: (Stallings, 2015))

- Register reference without the subscript refers to the original register;
- Register reference with the subscript refers to an allocated register;
- Subsequent instructions reference the most recently allocated register;
- **Important:** your book chooses some weird subscripts...

Example

$$
\begin{aligned}
&I1: \quad R3_b \leftarrow R3_a \ op \ R5_a \\
&I2: \quad R4_a \leftarrow R3_b + 1 \\
&I3: \quad R3_c \leftarrow R5_a + 1 \\
&I4: \quad R7_a \leftarrow R3_c \ op \ R4_a
\end{aligned}
$$

Figure: Register Renaming (Source: (Stallings, 2015))

- Creation of register $R3_c$ in instruction I3 avoids:
  - WAR dependency on I2;
  - WAW dependency on I1;
  - Interfering the correct value being accessed by I4;
- As a result I3 can be issued immediately;
  - Without renaming I3 cannot be issued until I1 is complete and I2 is issued.

But how can we gain a sense of how much performance is gained with such strategies?

But how can we gain a sense of how much performance is gained with such strategies?

- Use one scalar processor devoid of these strategies as a base system;
- Start adding various superscalar features;
- Comparison need to be performed against different programs.

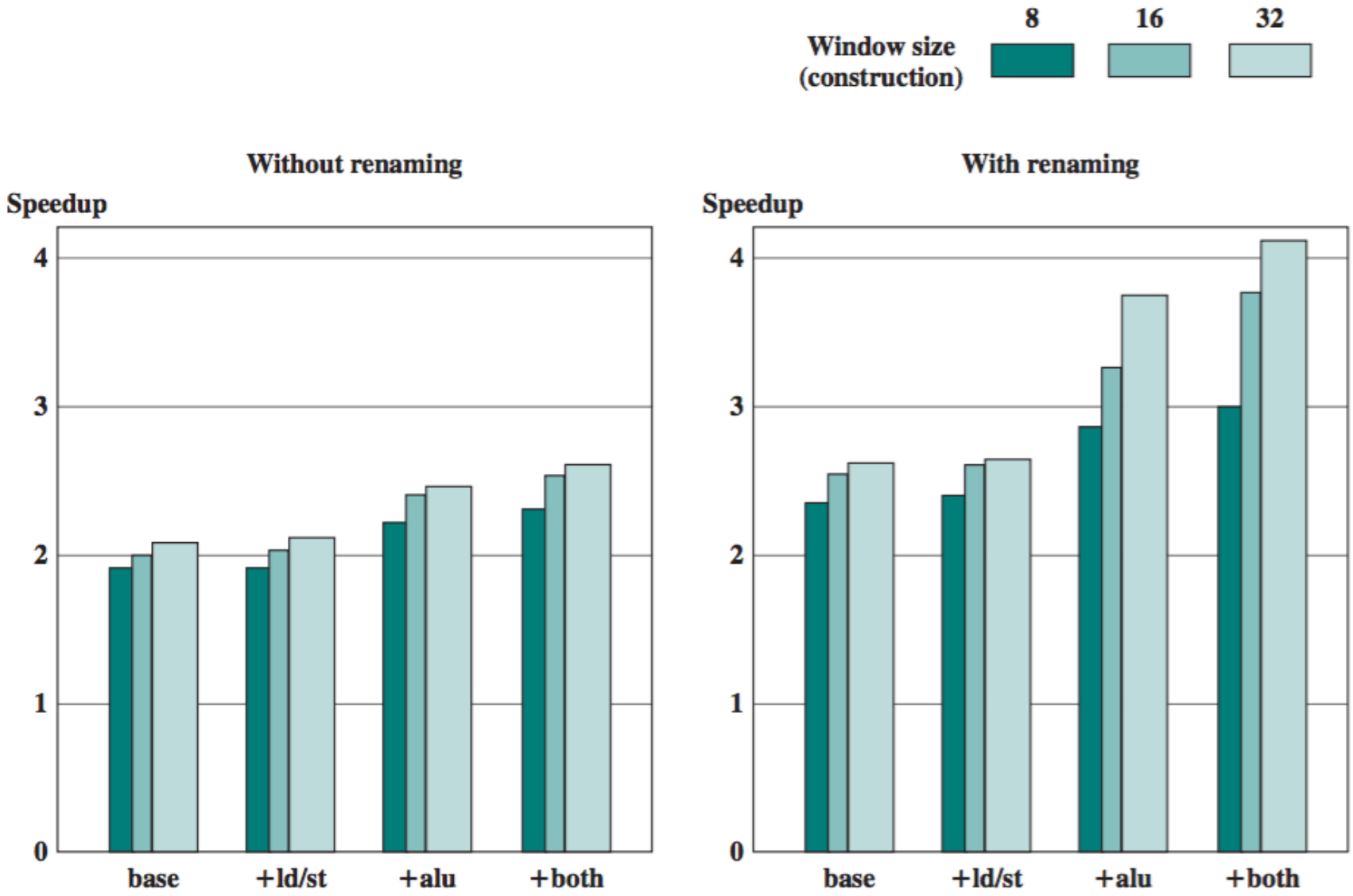


Figure: Speedups of various machine organizations without procedural dependencies (Source: (Stallings, 2015))

From the previous Figure:

- *Y*-axis is the mean speedup of the superscalar over the scalar machine;
- *X*-axis shows the results for four alternative processor organizations:
  - 1^{st} : no duplication of functional units, can issue instructions out of order;
  - 2^{nd} : duplicates the load/store functional unit that accesses a data cache;
  - 3^{rd} : duplicates the ALU;
  - 4^{th} : duplicates both load/store and ALU
- Window sizes of 8, 16, 32 instructions are shown.
- 1^{st} graph, no register naming is allowed, whilst in the 2^{nd} graph it is;

What conclusions can you derive from the previous picture?

Some conclusions (1/2):

- Probably not worthwhile to add functional units without register renaming.
- Performance improvement at the cost of hardware complexity.
- Register renaming gains are achieved by adding more functional units.

Some conclusions (2/2):

- Significant difference in performance gain regarding instruction window:
  - Small window prevents effective utilization of extra functional units;
  - Processor needs to look far ahead to:
    - Find independent instructions capable of using the hardware more fully.

Superscalar Execution Overview (1/8)

Lets review how all these concepts work together:

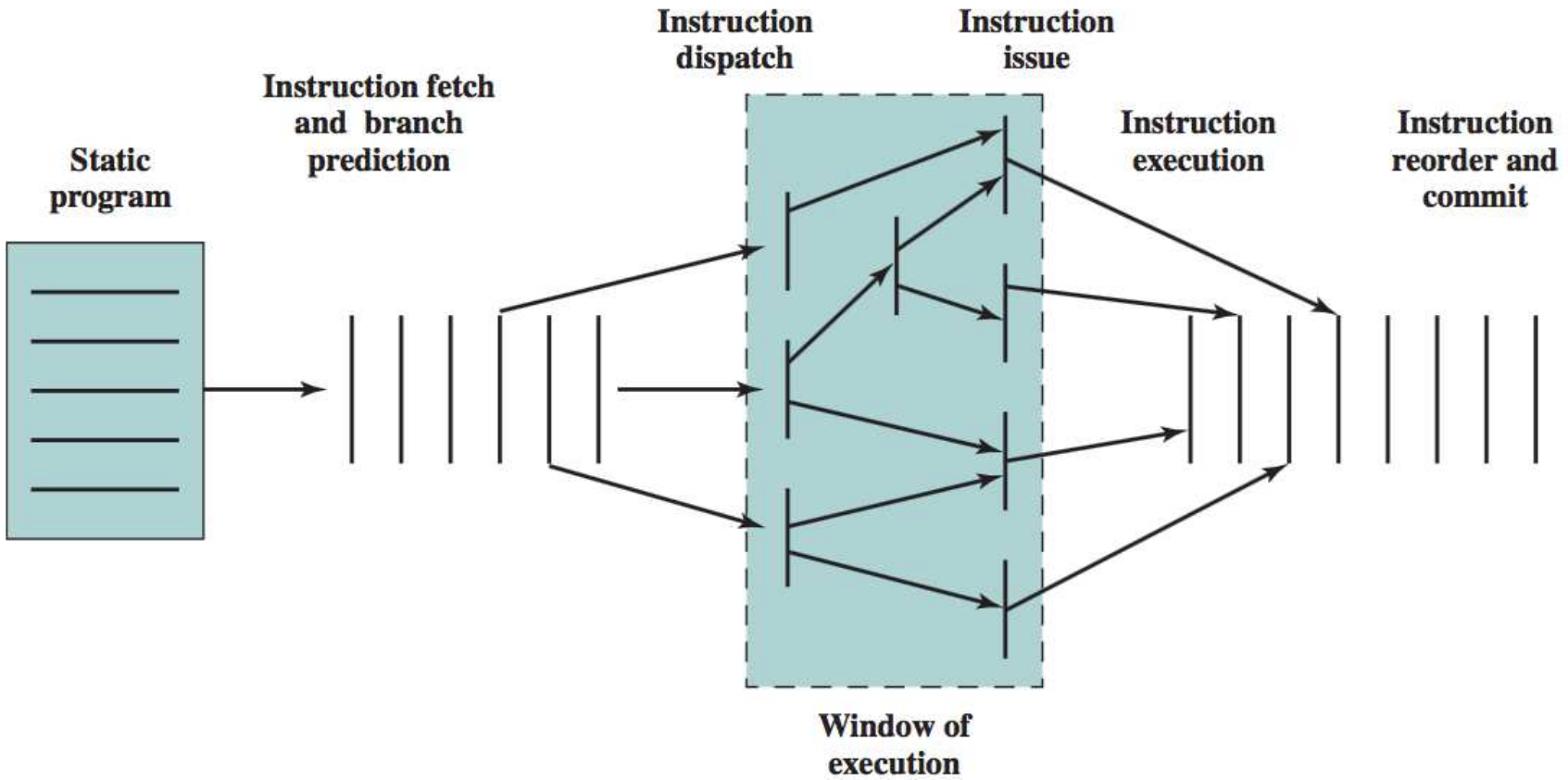


Figure: Conceptual depiction of superscalar processing (Source: (Stallings, 2015))

Superscalar Execution Overview (2/8)

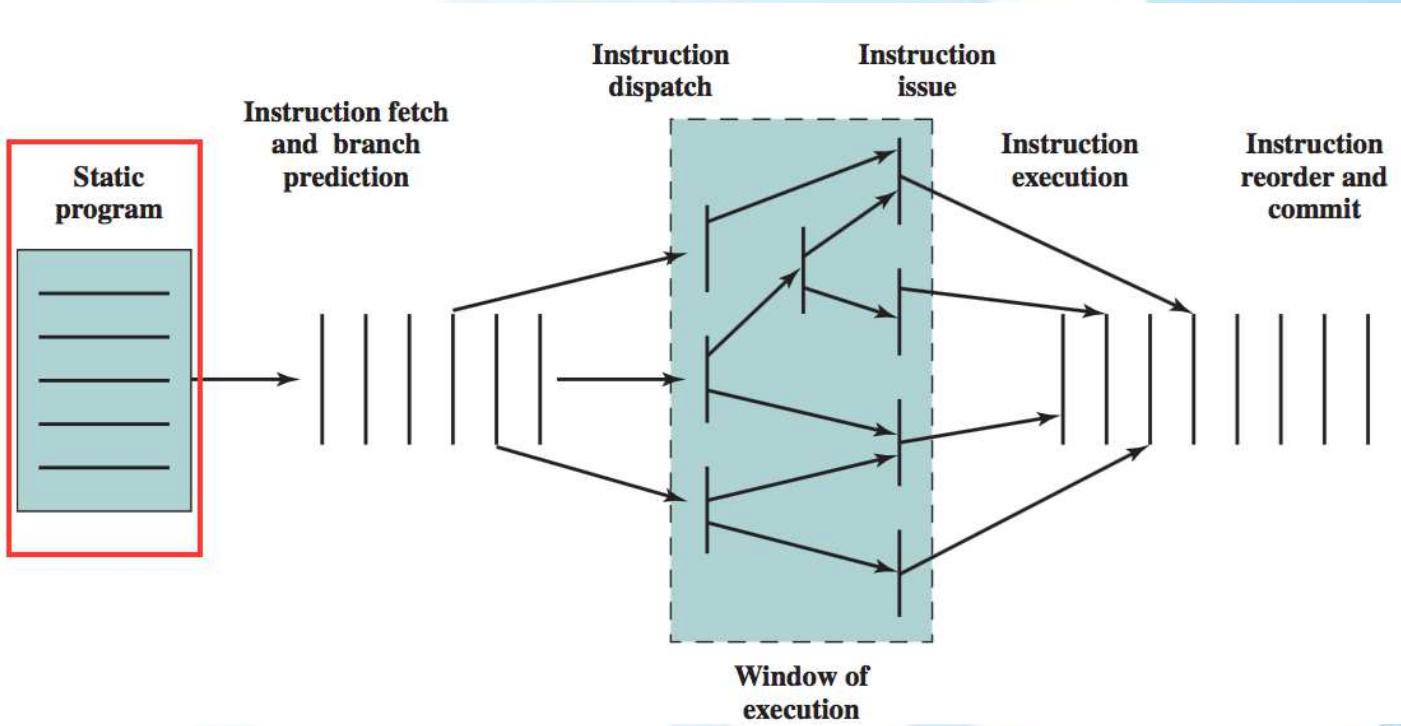


Figure: Conceptual depiction of superscalar processing (Source: (Stallings, 2015))

1. Program to be executed consists of a linear sequence of instructions;
2. This is the original sequential program generated by the compiler;

Superscalar Execution Overview (3/8)

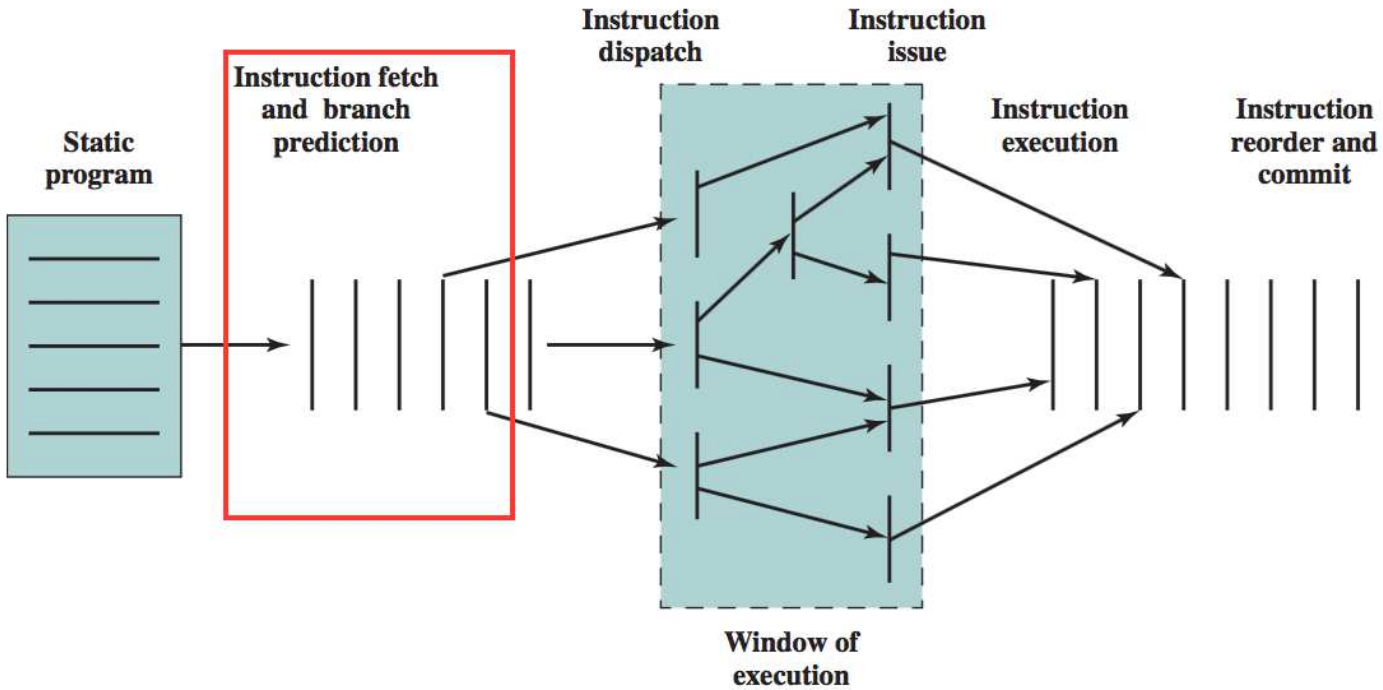


Figure: Conceptual depiction of superscalar processing (Source: (Stallings, 2015))

3. Instruction fetch stage generates a dynamic stream of instructions;
4. Processor attempts to remove dependencies from stream, *e.g.:*
   - Register renaming;

Superscalar Execution Overview (4/8)

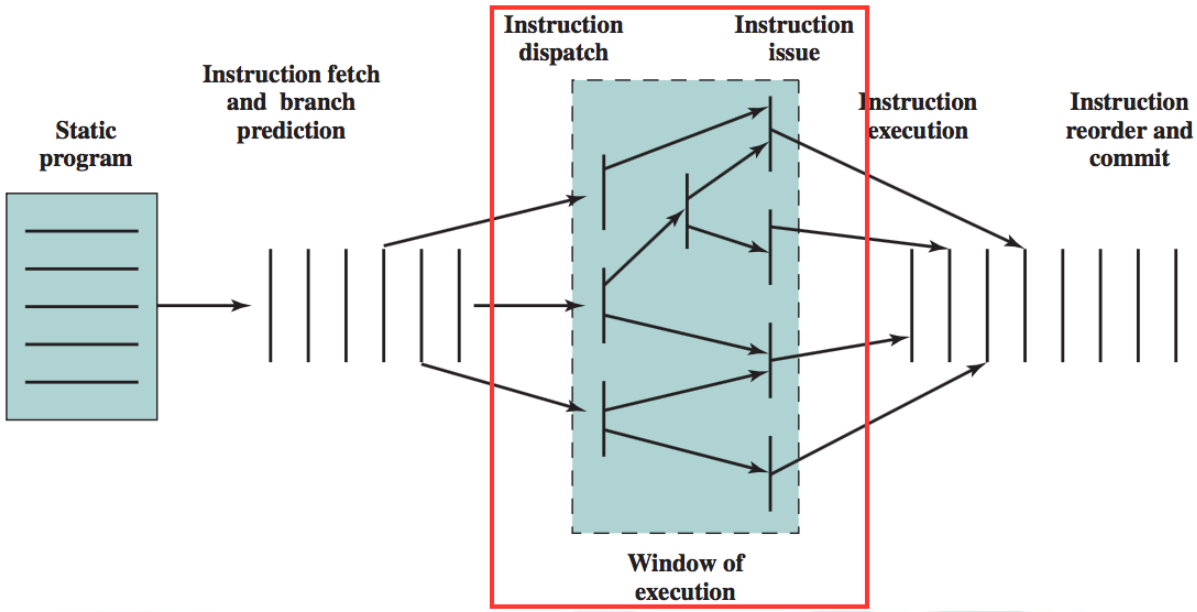


Figure: Conceptual depiction of superscalar processing (Source: (Stallings, 2015))

5. Processor dispatches instructions into an execution window;
6. In this window:
   - Instructions no longer form a sequential stream;
   - Instead instructions are structured according to data dependencies;

Superscalar Execution Overview (5/8)

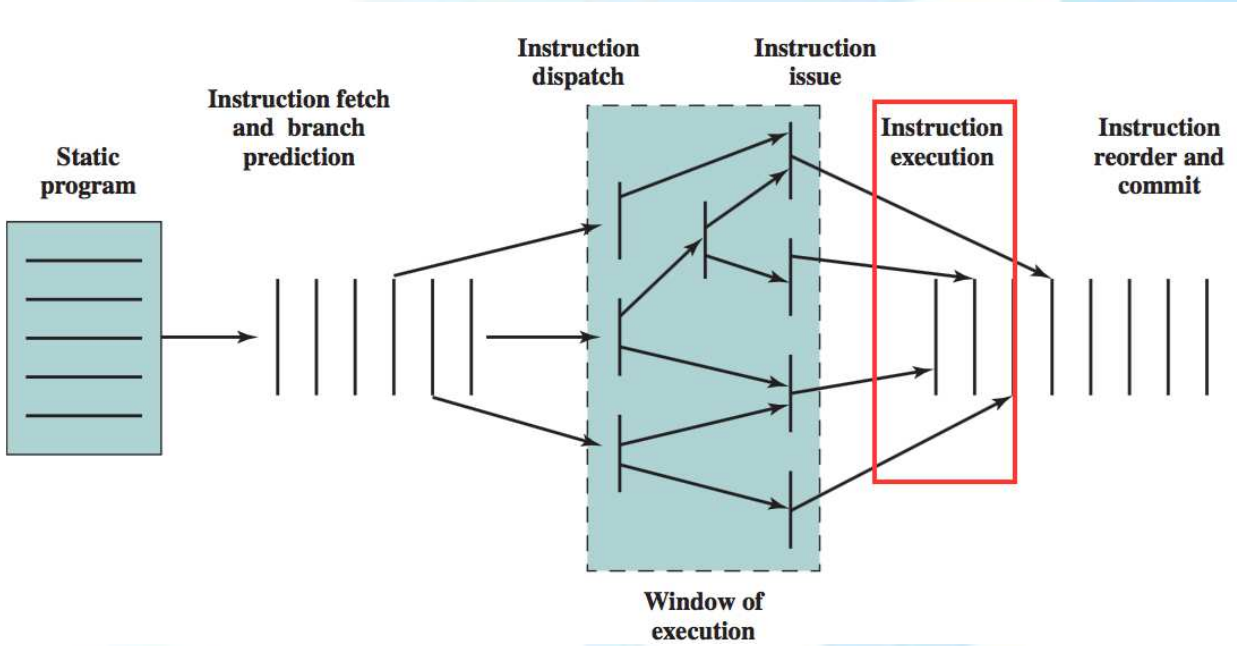


Figure: Conceptual depiction of superscalar processing (Source: (Stallings, 2015))

7. Processor executes each instruction in an order determined by:
   - Data dependencies;
   - Hardware resource availability;

Superscalar Execution Overview (6/8)

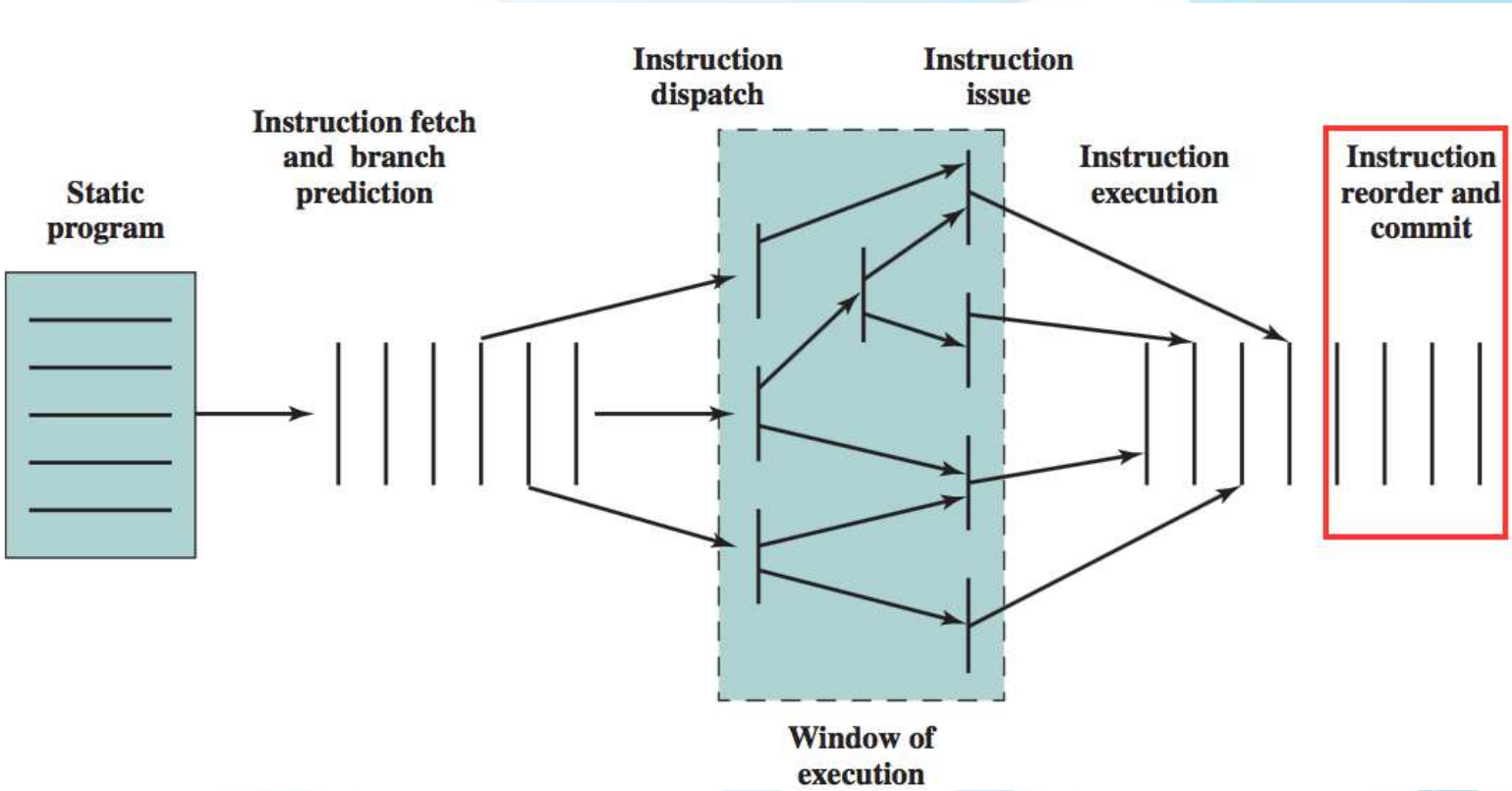


Figure: Conceptual depiction of superscalar processing (Source: (Stallings, 2015))

8. Instructions are put back into sequential order and their results recorded.

Superscalar Execution Overview (7/8)

With superscalar architecture (1/2):

- Instructions may complete in $\neq$ order from the one specified in program.
- Branch prediction and speculative execution means that:
  - Some instructions may complete execution and then must be abandoned;

Superscalar Execution Overview (8/8)

With superscalar architecture (2/2):

- Therefore memory and registers:
  - Cannot be updated immediately when instructions complete execution;
  - Results must be held in temporary storage that is made permanent when:
    - it is determined that the instruction executed in the sequential model;

References I

Stallings, W. (2015).

Computer Organization and Architecture.

Pearson Education.