Chapter 15 - Instruction-Level Parallelism and Superscalar Processors

Luis Tarrataca
luis.tarrataca@gmail.com

CEFET-RJ

Table of Contents I

Table of Contents I

Scalar Processor

- A pipelined functional unit for integer operations;
- A pipelined functional unit for floating-point operations;
- Parallelism is achieved by:
  - enabling multiple instructions to be at different stages of the pipeline

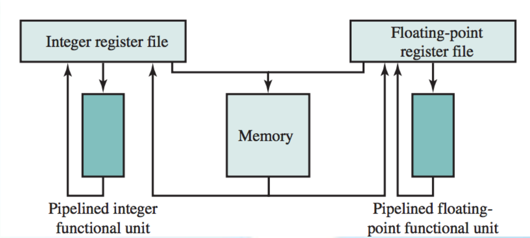

Figure: Scalar Organization (Source: (Stallings, 2015))

Superscalar processor

- ability to execute instructions in different pipelines:
  - independently and concurrently;

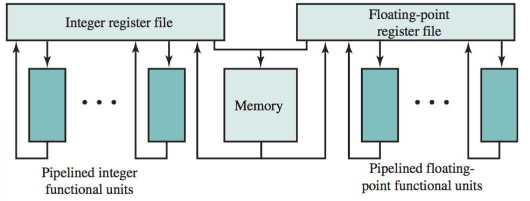

Figure: Superscalar Organization (Source: (Stallings, 2015))

- Multiple **functional units** exist:
  - Each of which is implemented as a **pipeline.**
  - Processor executes streams of instructions in parallel:
    - One stream for each pipeline.

But 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 the parallel execution does not violate the intent of the program;
    - tradeoff between performance and complexity;

Superscalar vs. Superpipelined

Superpipelining is an alternative performance method to superscalar:

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

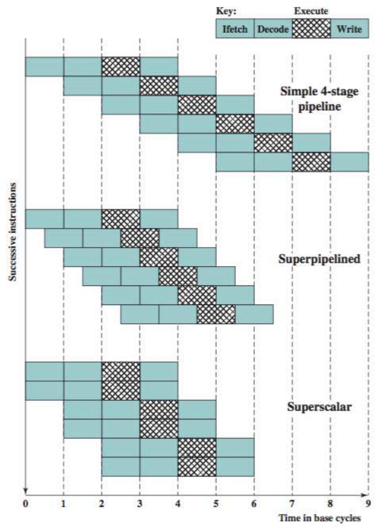

Figure: Comparison of superscalar and superpipeline approaches (Source: (Stallings, 2015))

From the previous figure, the **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.

From the previous figure, the **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;

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

- capable of executing two instances of each stage in parallel;

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

Do you have any idea of what are the fundamental limitation to paralllelism?

Do you have any idea of what are the fundamental limitation to paralllelism?

- True data dependency;
- Procedural dependency;
- Resource conflicts;

Lets have a look at these.

True 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;
- The second instruction needs data produced by the first instruction;
- A.k.a. read after write **RAW** dependency;

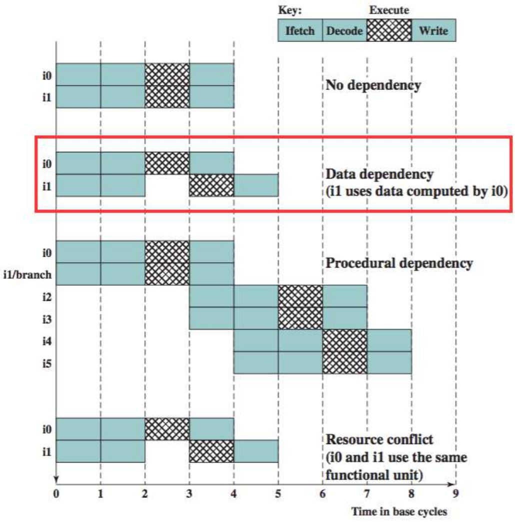

Figure: Effect of dependencies (Source: (Stallings, 2015))

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;

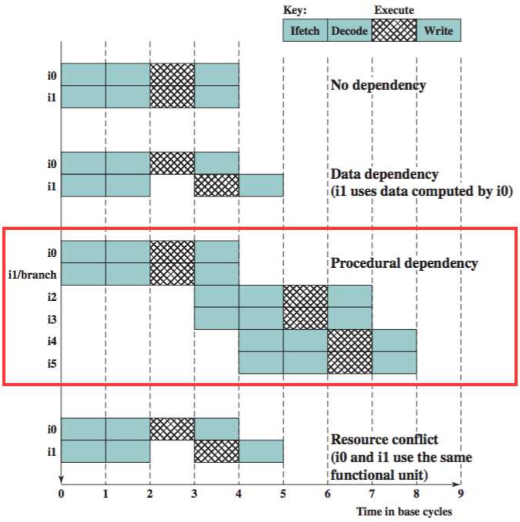

Figure: Effect of dependencies (Source: (Stallings, 2015))

Resource Conflict

Competition of two or more instructions 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

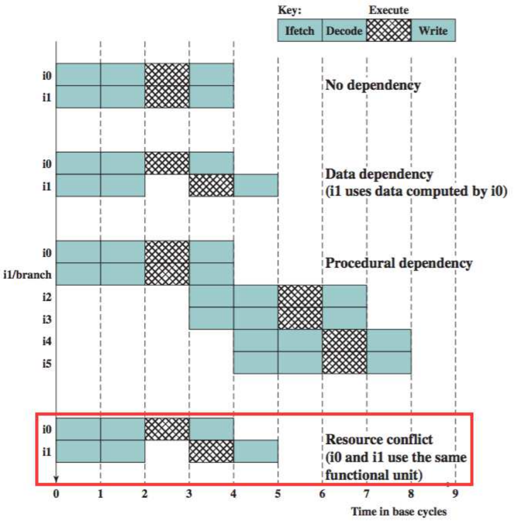

Figure: Effect of dependencies (Source: (Stallings, 2015))

There are several design issues to consider:

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

Guess what we will be seeing next? =)

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, and could be executed in parallel.
- right cannot be executed in parallel due to data dependency;

Degree of instruction-level parallelism is determined by the

- frequency of true data dependencies;
- procedural dependencies in the code;

These are dependent on the instruction set architecture and on the application.

Machine Parallelism

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

- take advantage of instruction-level parallelism;
- Determined by:
  - number of instructions that can be fetched at the same time;
  - number of instructions that can be executed at the same time;
  - speed and sophistication of the mechanisms that the processor uses 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;

Processor looks ahead to locate instructions that can be brought into the pipeline and executed:

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 the contents of register/memory locations

To optimize utilization of the various pipeline elements:

- processor may need to alter one or more of these orderings:
  - with respect to the ordering to be found in a strict sequential execution.
- This can be done as long as the final result is correct;

In general terms, instruction issue policies 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

Lets have a look at these;

In-order issue with in-order completion

Simplest instruction issue policy:

- Issue instructions in the exact order that would be achieved by 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 is seven cycles.

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:

- 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;
- When a functional unit becomes available in the execute stage:
  - An instruction from the instruction 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:
  - 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.
- 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!

References I

Stallings, W. (2015).

Computer Organization and Architecture.

Pearson Education.