William Stallings
Computer Organization and Architecture
7th Edition

Chapter 14
Instruction Level Parallelism and Superscalar Processors

Key points

- Superscalar is to use multiple *independent* instruction pipelines in a processor exploring instruction-level parallelism.

- A superscalar processor fetches multiple instructions at a time and attempts to find nearby instructions that are *independent* of one another.

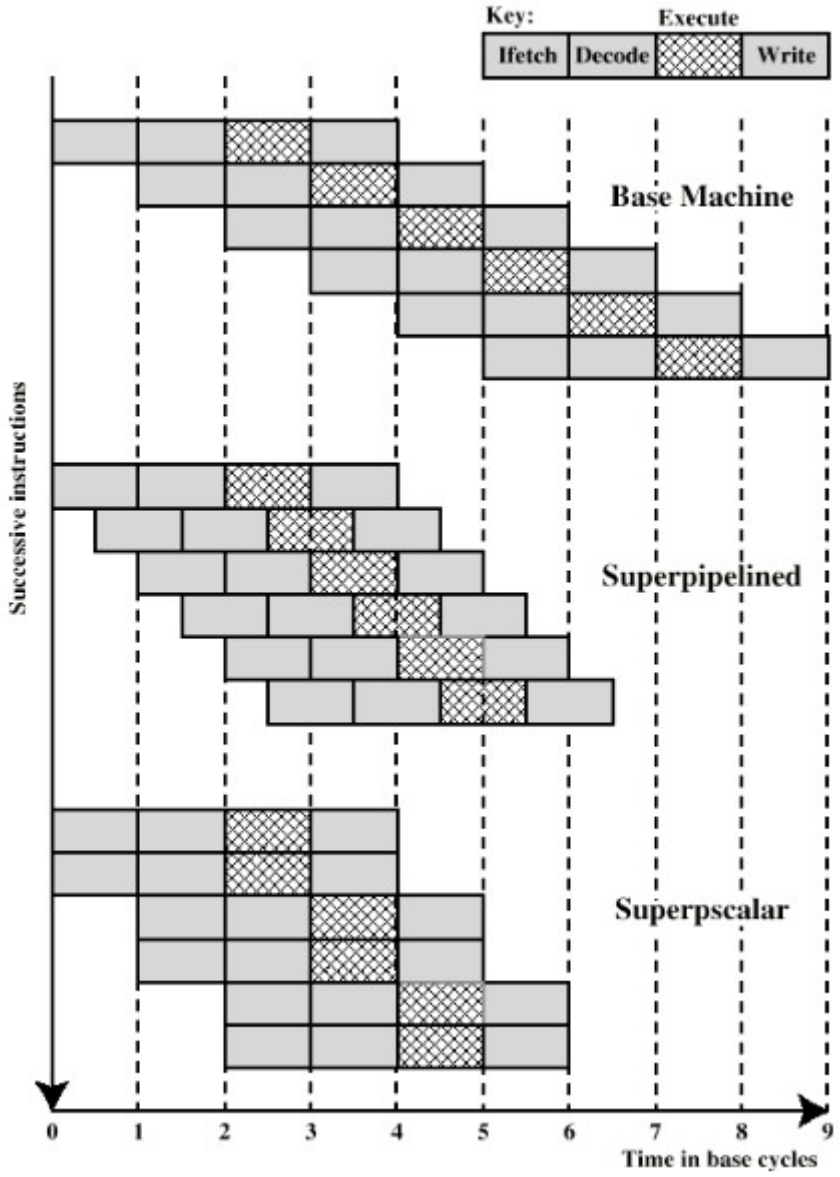
Key points

- The processor may *eliminate* some *unnecessary dependencies* by the use of additional registers and the renaming of register references in the original code.

- Most superscalar machines use branch prediction methods *rather than delayed branches* to improve efficiency.

What is Superscalar?

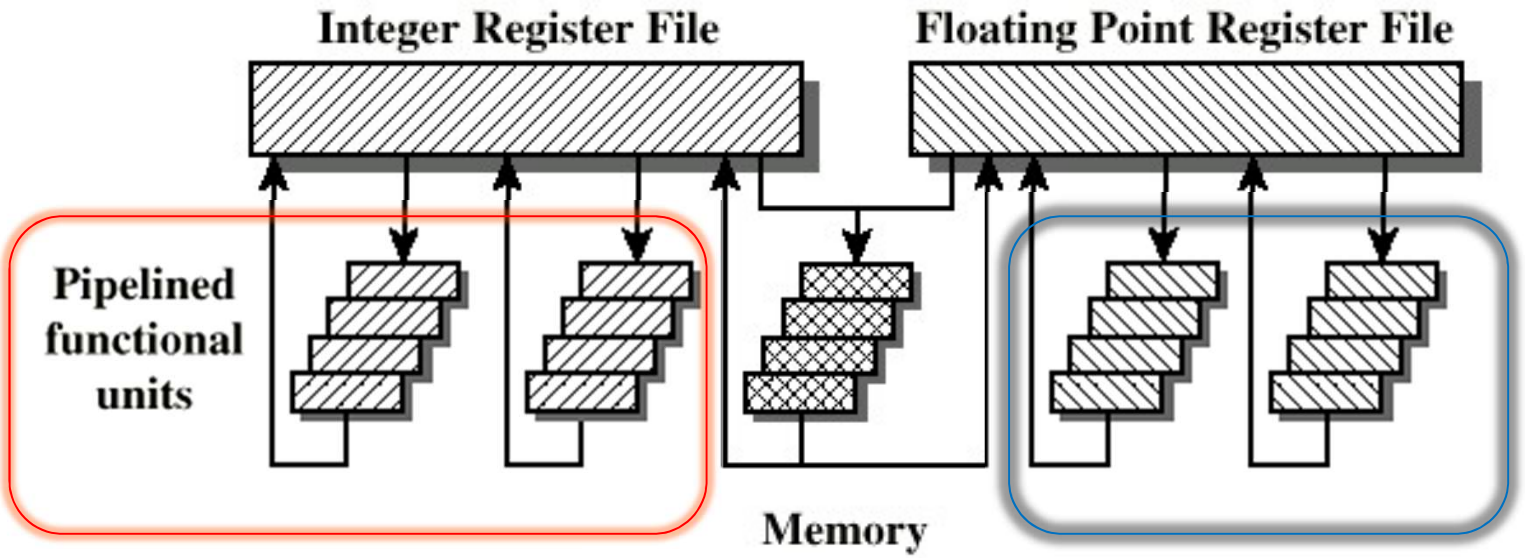
- Common instructions (arithmetic, load/store, conditional branch) can be initiated *simultaneously* and executed *independently*
- Equally applicable to *RISC & CISC*
- Standard method for high performance microprocessors



14.1 Overview

- The term superscalar first coined in 1987, refers to a machine that is designed to improve the performance of the execution of scale instructions
- Superscalar versus superpipelined
- Limitations

General Superscalar Organization



2 Integer Instructions

2 Floating Instructions

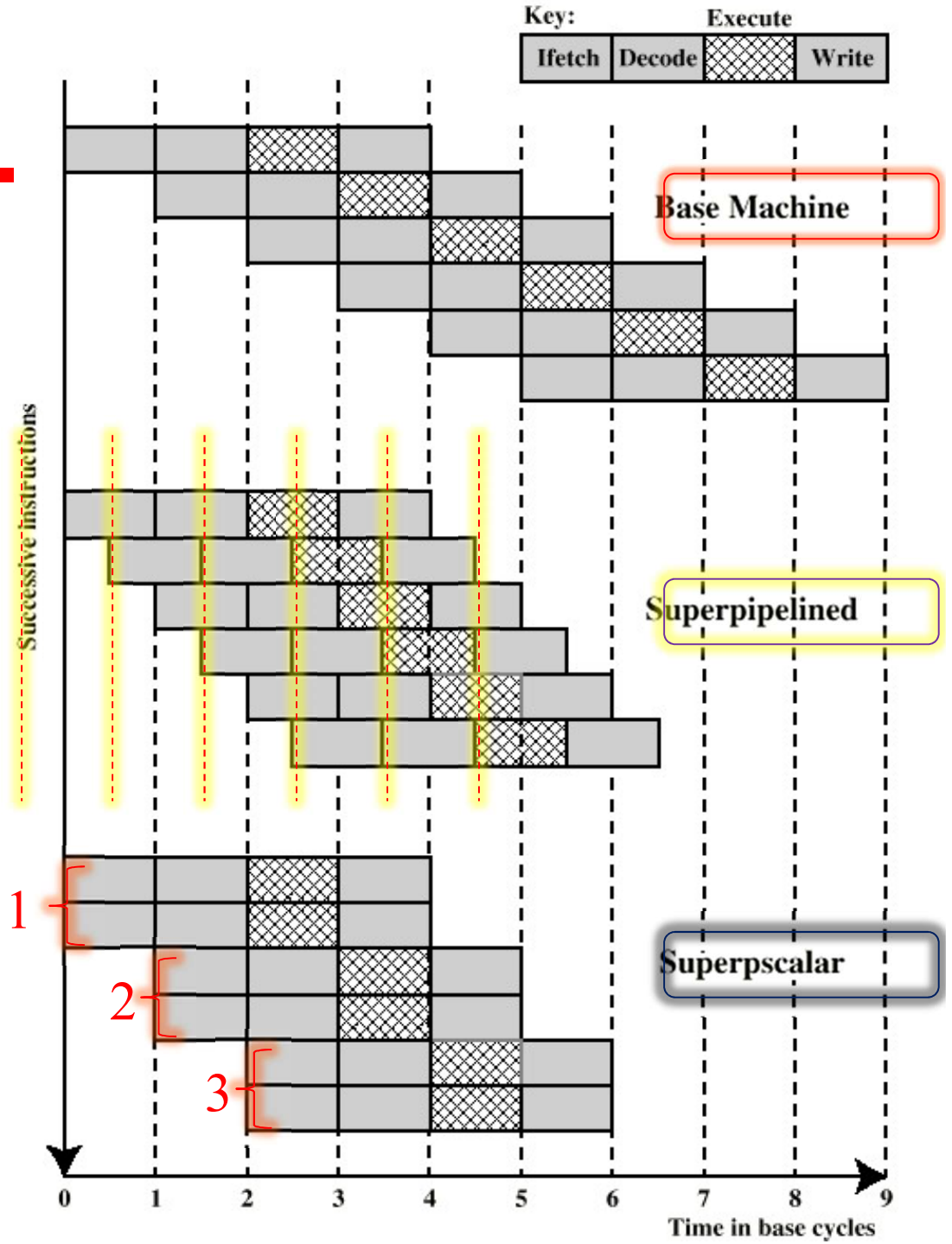
1 Memory Instruction

Superscalar VS Superpipeline

- Many pipeline stages need *less than half a clock cycle*
- ***Double*** internal clock speed gets two tasks per external clock cycle

- Superscalar allows ***parallel*** fetch execute

Superscalar v Superpipeline



Limitations

- Instruction level parallelism (*key*)
- *Compiler* based optimisation (*Cond.1*)
- *Hardware* techniques (*Cond.2*)

- Limited by
  - —True data *dependency*
  - —Procedural *dependency*
  - —Resource *conflicts*
  - —Output *dependency*
  - —Antidependency

True Data Dependency

- ADD r1, r2 (r1 := r1+r2; *Translation?*)
- MOVE r3,r1 (r3 := r1; *Translation?*)

- (for Superscalar) Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished

➢ ***Flow dependency*** or ***write-read dependency***

True Data Dependency

- *Memory execution* makes the delay longer
  - —Memory access may take 2 or more clock cycles to LOAD/STORE;

✓ For RISC, *reorder instructions* can speed up pipeline

X For Superscalar, reorder is *less effective* than in RISC pipeline

Procedural Dependency

- For ***conditional branch***: Can not execute instructions after a *branch* in parallel with instructions before a branch

- Also, if ***instruction length is not fixed*** (*i.e. for CISC*), instructions have to be decoded to find out how many fetches are needed
- This prevents simultaneous fetches

➢ One reason of superscalar being more readily ***applicable to RISC***

Resource Conflict

- Two or more instructions requiring access to the ***same resource*** at the same time
  - e.g. two arithmetic instructions, both need ALU device;

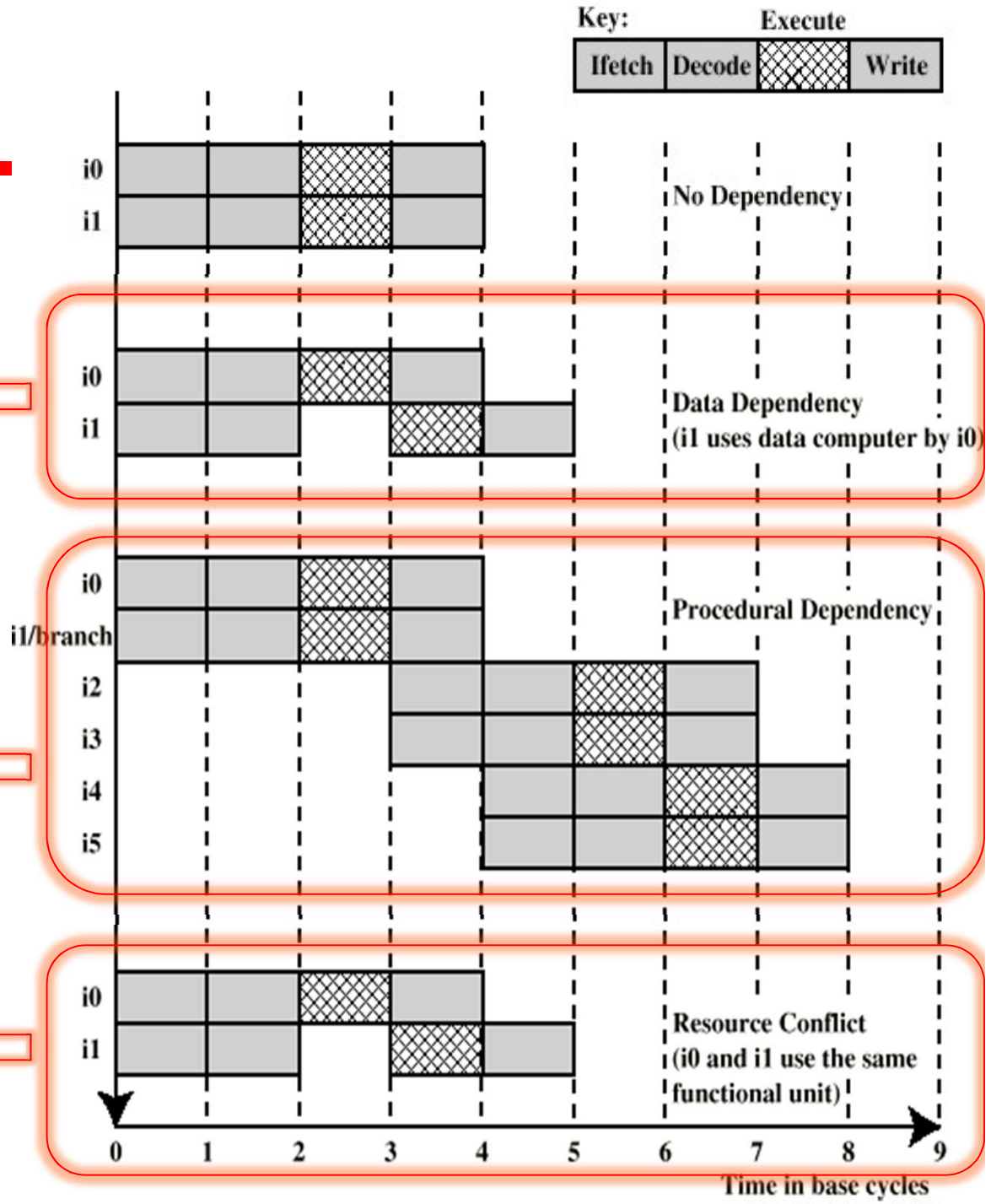
- Can duplicate resources
  - e.g. have two arithmetic units

Effect of Dependencies

RISC is better than Superscalar!
Reorder is effective.

RISC is more readily than CISC !

Duplicate resource!



14.2 Design Issues

- Instruction level parallelism and Machine Parallelism
- Instruction issue policy
- Register renaming
- Machine parallelism
- Branch
- Superscalar execution

Instruction level parallelism

- Instructions in a sequence are ***independent*** (key)

➢ Execution can be ***overlapped***

X Governed by data and procedural *dependency* and operation *latency*

Instruction level parallelism

- E.g1 *Independent* instructions

  Load R1 ← R2

  Add R3 ← R3, "1"

  Add R4 ← R4, R2

- E.g2 *Dependent* instructions (?)

  Add R3 ← R3, "1"

  Add R4 ← R3, R2

  Store [R4] ← R0

Machine Parallelism

- Machine (CPU's) ***ability*** to take advantage of instruction level parallelism

- limited by
  - the ***speed*** and sophistication of the mechanisms to find independent instructions
  - ***number*** of parallel pipelines to be fetched and executed

Instruction Issue Policy

- *Instruction issue*: process of initiation instruction execution in the processor's functions
- *Instruction issue policy*: protocol used to issue instructions

➢ ***Important orders***:

- Order in which instructions are *fetched*
- Order in which instructions are *executed*
- Order in which instructions *change registers and memory*

In-Order Issue
In-Order Completion

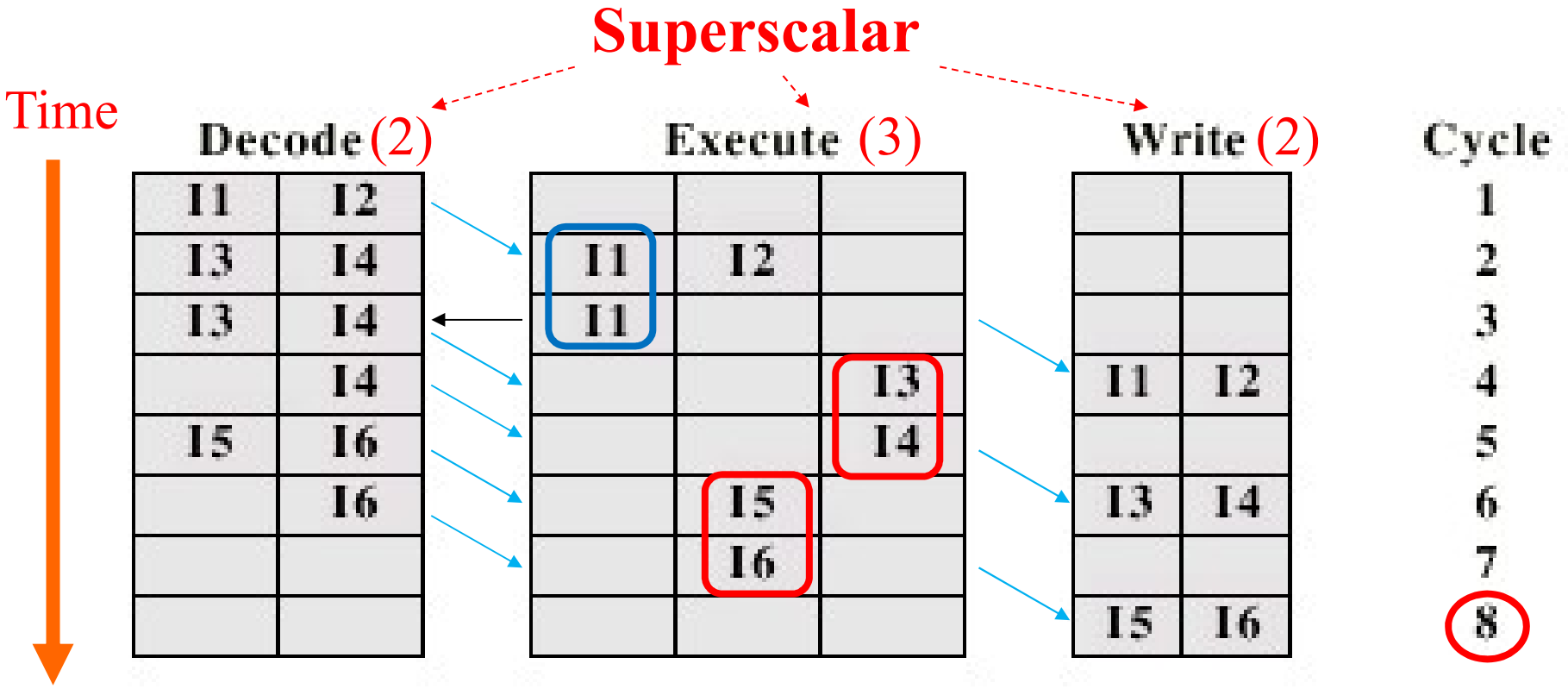
- Issue instructions in the exact order they occur
- Write results in that same order

➢ Not very efficient

✘ May fetch >1 instruction

✘ Instructions must stall if necessary (dependency, branch…)

In-Order Issue
In-Order Completion (Diagram)



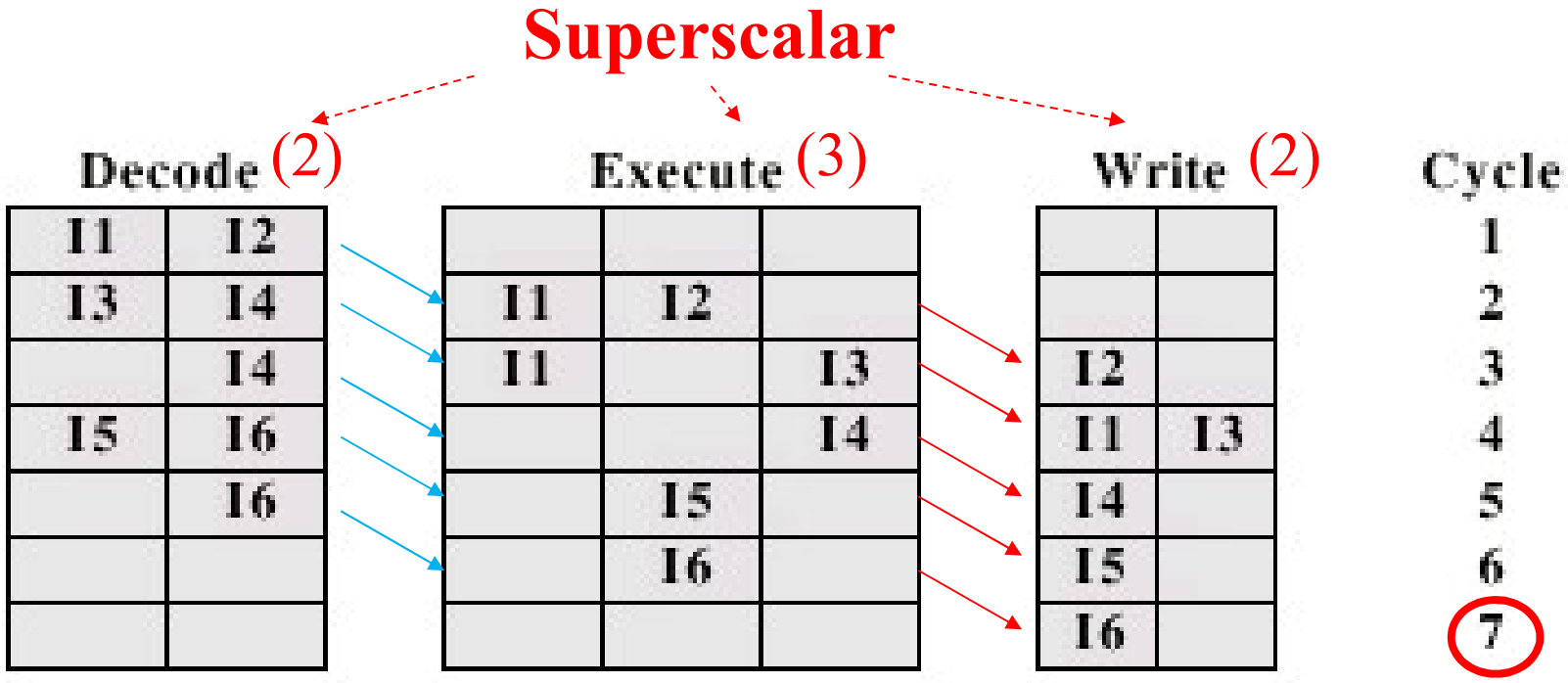
- I1 needs 2 time cycles;
- I3, I4 need the same function unit;
- I5 need the outputs of I4;
- I5, I6 need the same function unit;

In-Order Issue
Out-of-Order Completion

- Any number of instructions may be in the execution stage at any one time
- *Stalled* by a **resource conflict** , a **data dependency,** or a **procedural dependency**

- **Output dependency**
  - I1: R3 ← R3 op R5
  - I2: R4 ← R3 + 1
  - I3: R3 ← R5 + 1
  - I4: R7 ← R3 op R4
  - I2 depends on result of I1 - data dependency
  - I4 depends on result of I3 - data dependency
  - If I3 completes before I1, the result from I1 will be wrong - output (**write-write**) dependency

In-Order Issue
Out-of-Order Completion (Diagram)

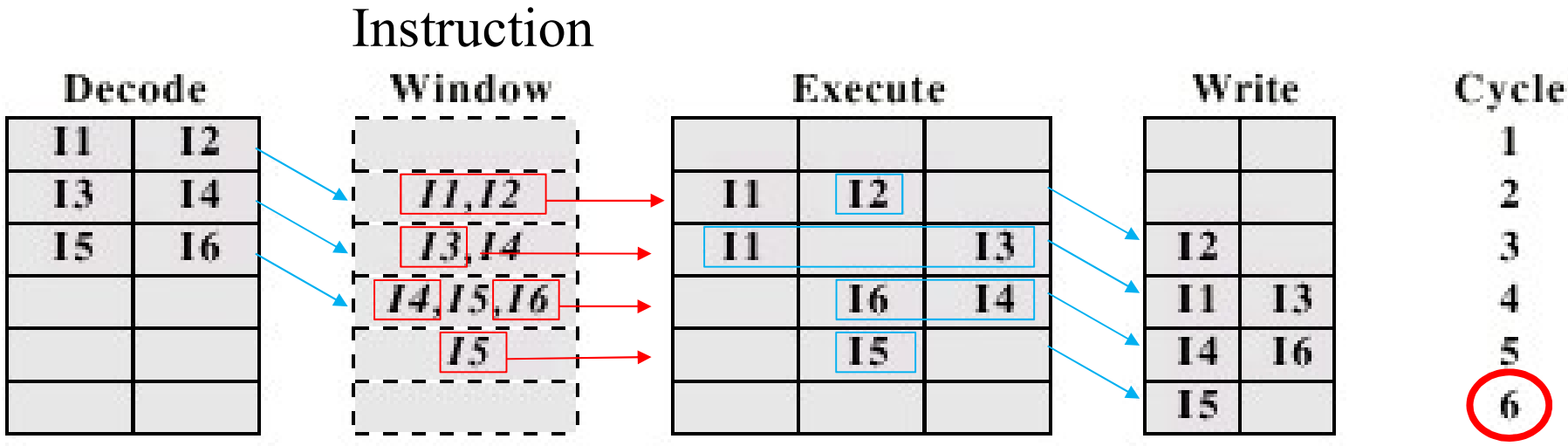


- I1 costs 2 time cycles;
- I3, I4 need the same function unit;
- I5 need the outputs of I4;
- I5, I6 need the same function unit;
- *I2 can be completed before I1;*

Out-of-Order Issue
Out-of-Order Completion

- Decouple *decode* pipeline from *execution* pipeline
- ***Instruction window*** is used
- Can continue to *fetch and decode* until this pipeline is full
- When a functional unit becomes available an instruction (no conflict or dependency) can be executed
- Since instructions have been decoded, processor can look ahead
- *The outcome must be right!*

Out-of-Order Issue Out-of-Order Completion (Diagram)



- I1 costs 2 time cycles;
- I3, I4 need the same function unit;
- I5 need the outputs of I4;
- I5, I6 need the same function unit;

Antidependency

- ***Read-write*** *dependency*
  - I1: R3 ← R3 op R5
  - I2: R4 ← R3 + 1
  - I3: R3 ← R5 + 1
  - I4: R7 ← R3 op R4
  - I3 can not complete before I2 starts as I2 needs a value in R3 and I3 changes R3

Register Renaming

- *Output and antidependencies* occur because *register contents* may not reflect the correct ordering from the program
- Replace with another Register would solve the problem
- May result in a pipeline stall
- *Dynamical allocations* of registers take the most of registers ->making conflict more acute

➢ ***Duplication of resources*** is one method to cope with conflict

Register Renaming example

- $R3_b := R3_a + R5_a$ (I1)
- $R4_b := R3_b + 1$ (I2)
- $R3_c := R5_a + 1$ (I3)
- $R7_b := R3_c + R4_b$ (I4)
- Without subscript refers to *logical register* in instruction
- With subscript is *hardware register* allocated
- *Reduce the unnecessary output/anti-dependencies*

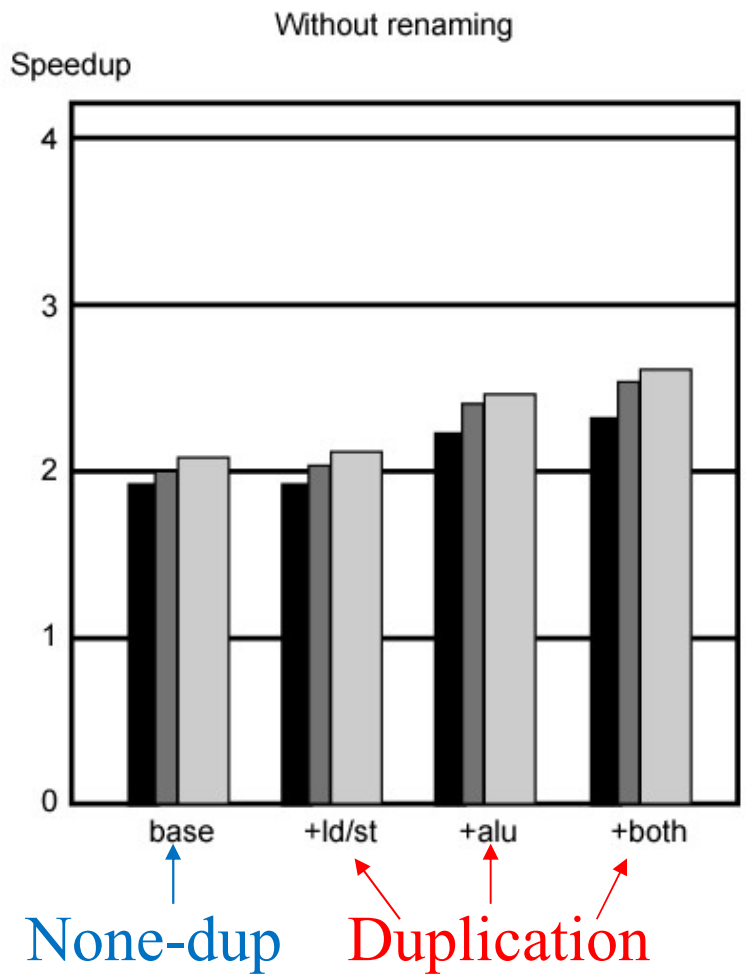
Machine Parallelism evaluation

- Techniques for performance enhancement:
  - —*Duplication* of Resources (***ld/st/alu…***)
  - —*Out of order* issue and completion (***inst. windows***)
  - —*Renaming* Registers (***duplicate reg.s***)

- Simulation Conclusions:
  - —Not worthwhile to add function units without register renaming
  - —Need instruction window large enough (more than 8)

Speedups of Machine Organizations Without Procedural Dependencies

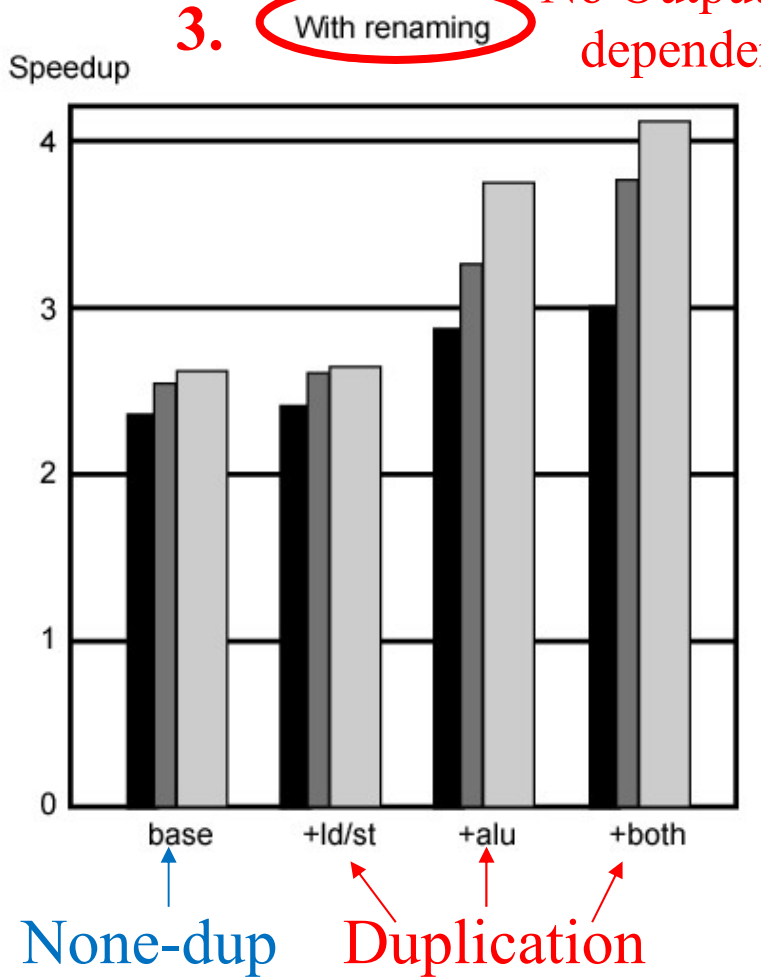
2. Window size (construction): 8, 16, 32



1. None-dup Duplication

3. With renaming

No Output/Anti-dependency.



None-dup Duplication

Branch Prediction

High performance pipeline machines must deal with branches

- 80486 (CISC) ***fetches*** both *next sequential instruction* after branch and *branch target instruction*
- RISC machines use ***delayed branch*** strategy
- Delayed branch strategy has *less appeal* for superscalar machines <- *dependencies*

➢ ***Branch prediction*** techniques return

  —*Static* branch prediction

  —*Dynamic* branch prediction

RISC - Delayed Branch

- Calculate result of branch ***before*** unusable instructions pre-fetched
- Always execute single instruction ***immediately following*** branch

➢ Keeps pipeline full while fetching new instruction stream

- Not as good for superscalar
  - —*Multiple* instructions need to execute in delay slot
  - —*Instruction dependence* problems

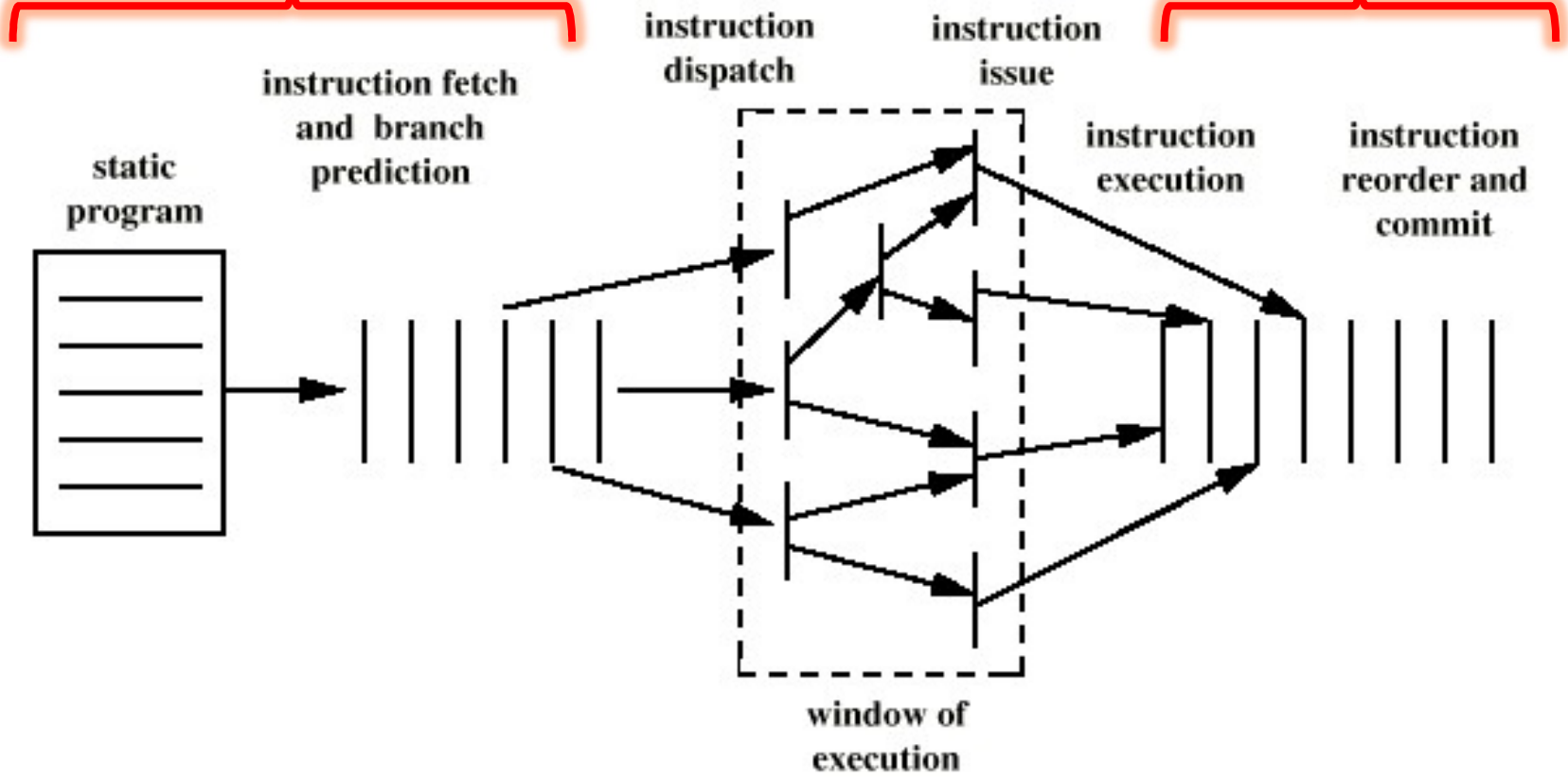
➢ Revert to ***branch prediction***

Superscalar Execution

In-order Sequential Fetch

Out-of-order Inst.

Reorder Sequential Inst.



Based on Data-dependency, Resource conflicts

Superscalar Implementation

- Simultaneously *fetch* multiple instructions
- Logic to *determine* true dependencies involving register values
- Mechanisms to *communicate* these values
- Mechanisms to *initiate* multiple instructions in parallel
- Resources for *parallel execution* of multiple instructions
- Mechanisms for *committing* process state in correct order

Homework

- Reading : Stallings ch14
- Translate Key terms
- Review questions: 4, 7
- Problem: 5