# Chapter 14
# Instruction Level Parallelism and Superscalar Processors

**2020.6**

**Howon Kim**

- 정보보호 및 지능형 IoT연구실 - http://infosec.pusan.ac.kr
- 부산대 지능형융합보안대학원 - http://aisec.pusan.ac.kr

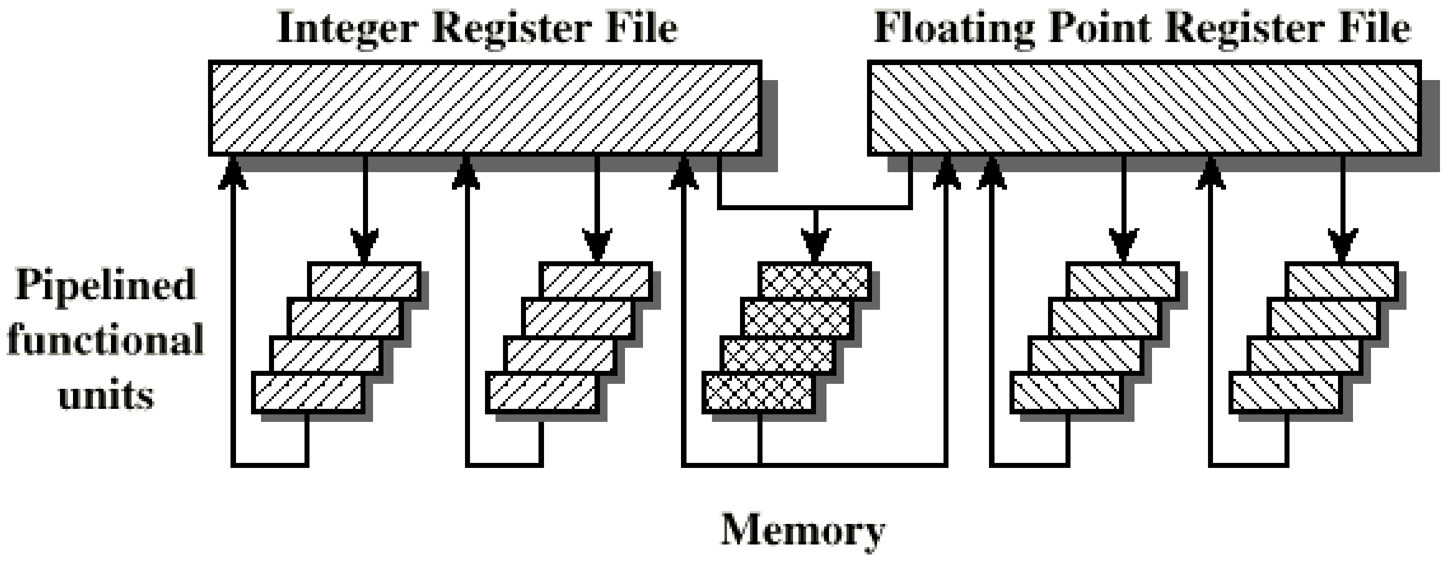
## What is Superscalar?

- Common instructions (arithmetic, load/store, conditional branch) can be initiated and executed independently
- Equally applicable to RISC & CISC
- In practice usually RISC

## Why Superscalar?

- Most operations are on scalar quantities (see RISC notes)
- Improve these operations to get an overall improvement

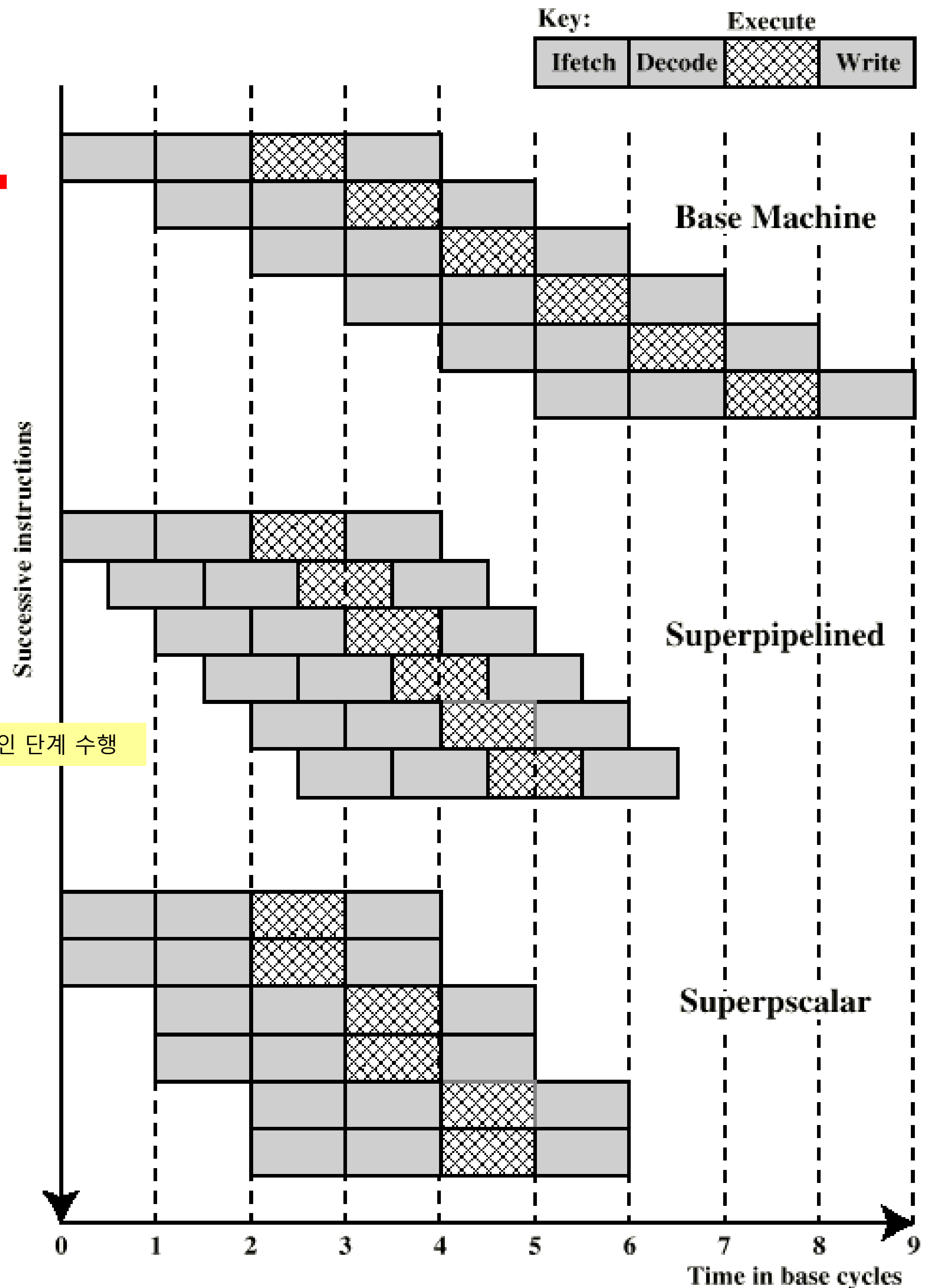
## General Superscalar Organization



## Superpipelined

- 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



Clock cycle당 두개의 파이프라인 단계 수행

## Limitations

- Instruction level parallelism (ILP)

프로그램 명령어를 병렬로 수행하는
정도(degree를 의미함)

  - ILP is supported by Compiler based optimisation + Hardware techniques
- ILP is Limited by
  - True data dependency
  - Procedural dependency
  - Resource conflicts
  - Output dependency
  - Antidependency

## True Data Dependency

- ADD r1, r2 (r1 := r1+r2;)
- MOVE r3,r1 (r3 := r1;)
- Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished

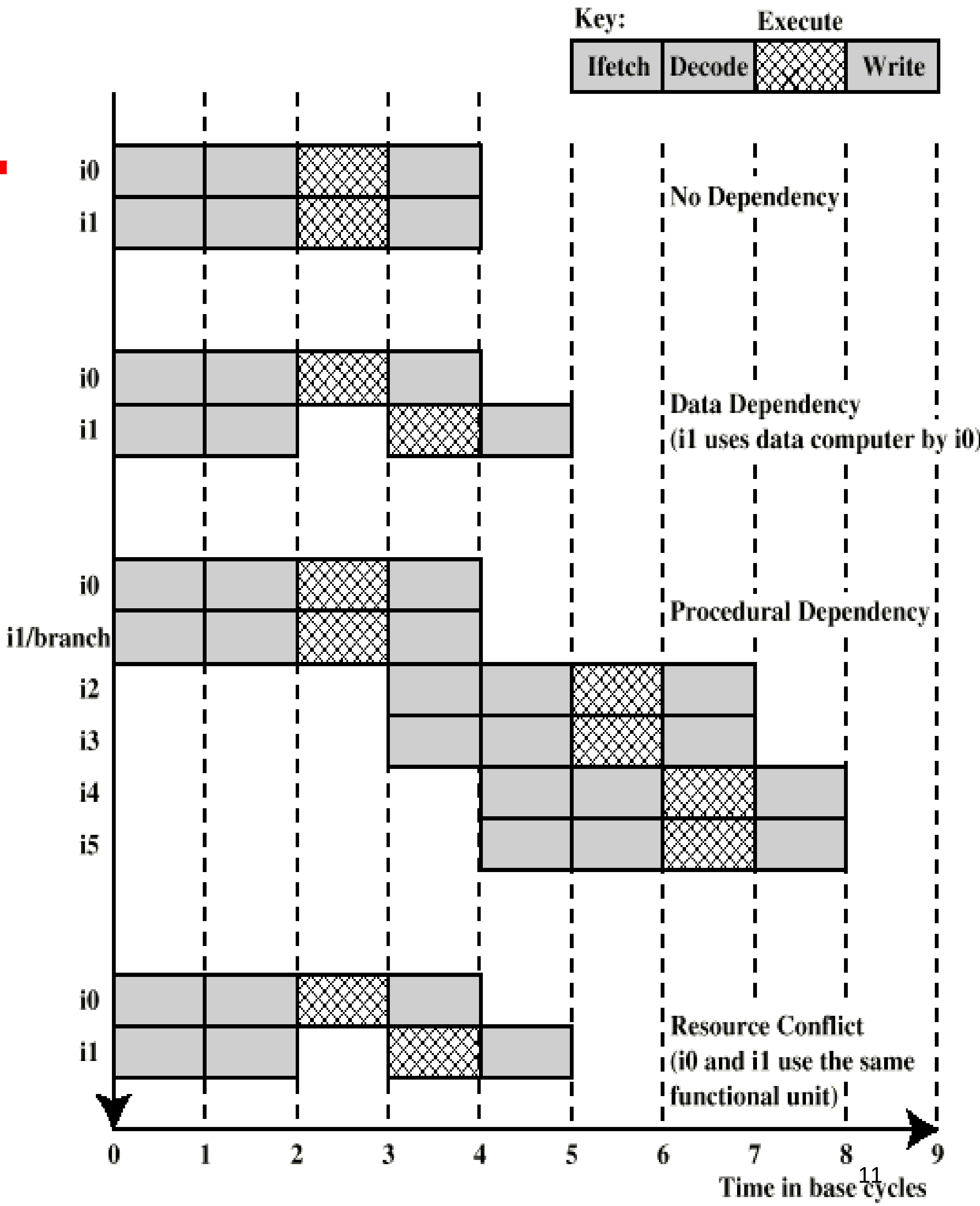
## Procedural Dependency

- Can not execute instructions after a branch in parallel with instructions before a branch
- Also, if instruction length is not fixed, instructions have to be decoded to find out how many fetches are needed
- This prevents simultaneous fetches

## Resource Conflict

- Two or more instructions requiring access to the same resource at the same time
  - —e.g. two arithmetic instructions
- Can duplicate resources
  - —e.g. have two arithmetic units

# Effect of Dependencies



## Design Issues

- Instruction level parallelism
  - If instructions in a sequence are independent, execution can be overlapped
    - Governed by data and procedural dependency
- Machine Parallelism
  - Ability to take advantage of instruction level parallelism
    - Governed by the number of instructions that can be fetched and executed at the same time (number of parallel pipelines)
    - Governed by the speed and sophistication of the mechanisms that the processor uses to find independent instructions

## Instruction Issue Policy

- Instruction issue:
  - refers to the process of initiating instruction execution in the processor's functional units
- Instruction issue policy :
  - refers to the protocol used to issue instructions

# Instruction Issue Policy

- The processor is trying to look ahead of the current point of execution 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 change registers and memory
- The more sophisticated the processor, the less it is bound by a strict relationship between these orderings
- To optimize utilization of the various pipeline elements, the processor will need to alter one or more of these orderings with respect to the ordering to be found in a strict sequential execution
- The one constraint on the processor is that the result must be correct
  - Thus, the processor must accommodate the various dependencies and conflicts discussed earlier

## Superscalar Instruction Issue Policies

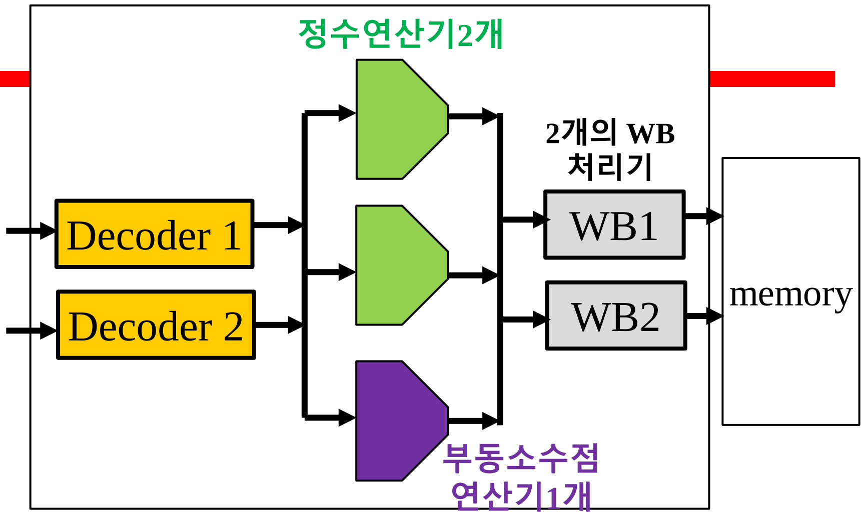
- In-order issue with in-order completion
- In-order issue with out-of-order completion
- Out-of-order issue with out-of-order completion

## In-Order Issue
## In-Order Completion

- Issue instructions in the order they occur
- Not very efficient
- May fetch >1 instruction
- Instructions must stall if necessary

# In-Order Issue In-Order Completion (Diagram)

**가정:**
- 두 개의 명령어를 동시에 fetch하여 decoding할 수 있음
- 실행 unit은 두 개의 정수 연산과 한 개의 부동 소수점 연산 동시 수행
- 두 개의 write-back 파이프라인 단계 가짐



본 예제를 위한 superscalar 프로세서 구조

제한:
-I1을 실행하는데 두 사이클이 필요.
-I3/I4 resource conflict. I5/I6도 resource conflict
-I5는 I4의 결과값을 사용함. ※ I1과 I2는 data dependency를 가지지 않음.

| 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 |

## In-Order Issue
## Out-of-Order Completion

- Output dependency

가정에선 **Data dependency** 없다고 했음

- R3:= R3 + R5; (I1)
- R4:= R3 + 1; (I2)
- R3:= R5 + 1; (I3)

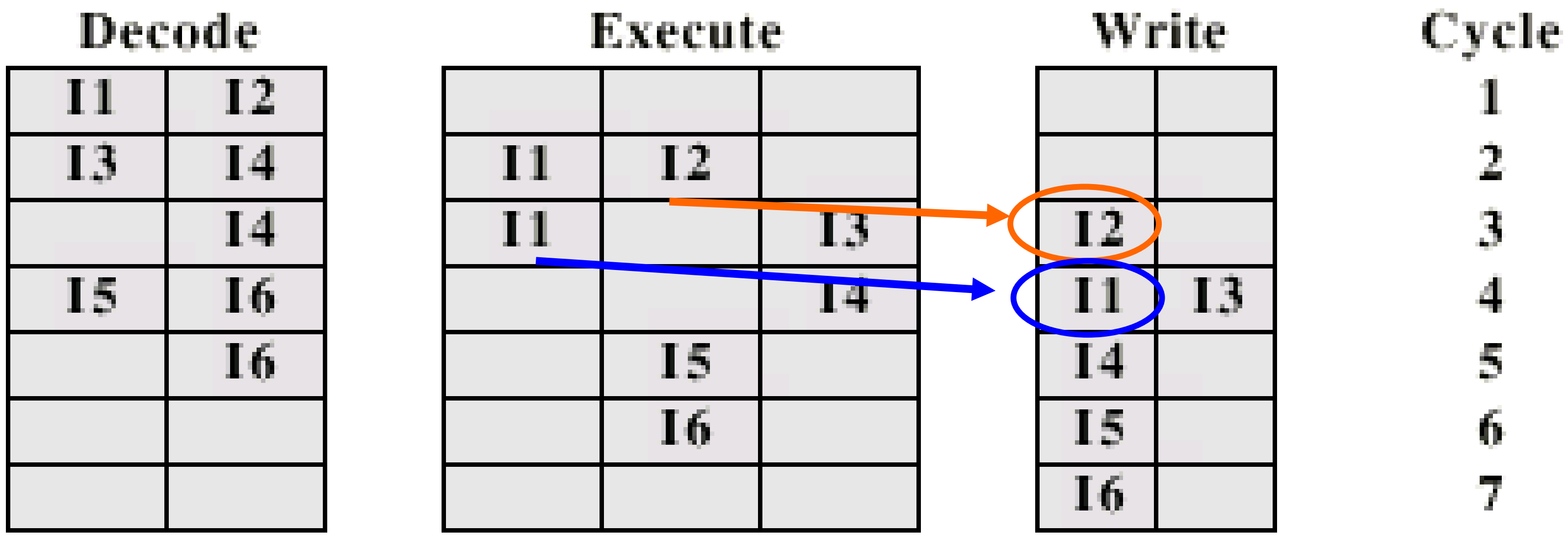
Output Dependency

- If I3 completes before I1, the result from I1 will be wrong - output (write-write) dependency
  - Cf. If I2 depends on result of I1, then we call it data dependency

A commonly used naming convention for data dependencies is the following: Read-after-Write or RAW (flow dependency), Write-after-Write or WAW (output dependency), and Write-After-Read or WAR (anti-dependency).

# In-Order Issue Out-of-Order Completion

제한:
-I1을 실행하는데 두 사이클이 필요.
-I3/I4 resource conflict. I5/I6도 resource conflict
-I5는 I4의 결과값을 사용함. ※ I1과 I2는 data dependency를 가지지 않음.



| 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 |

- **I1,I2를 동시에 issuing 했지만, I2를 I1보다 이전에 실행 종료(WB)했음(out-of-order completion,** 가정에서 data dependency가 없다고 했음).
- 이 때문에,I2의 결과를 Cycle #3에서 write-back함
- 이는 결과적으로 I3 실행을 이전 경우보다 더 빨리 할 수 있도록 하며, I3의 결과를 I1의 결과와 함께 Cycle #4에 write-back할 수 있게 됨

두 개의 write-back 파이프라인 단계 가짐. 즉, 최대 두개의 결과값을 동시에 write-back 할 수 있음.

R3:= R3 + R5; (I1)
R4:= R3 + 1; (I2)
R3:= R5 + 1; (I3)

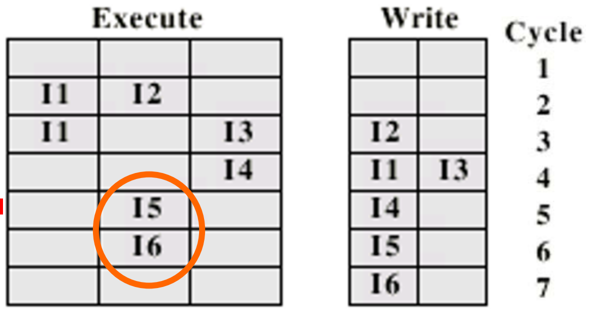
만일, 왼쪽과 같은 경우라면 I1과 I3의 실행 순서에 의해 최종 R3의 값이 바뀌게 되며, 이를 output dependency (write-after-write dependency라고 함)

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

- Decouple decode pipeline from execution pipeline
- Can continue to fetch and decode until this pipeline is full
- When a functional unit becomes available an instruction can be executed
- Since instructions have been decoded, processor can look ahead

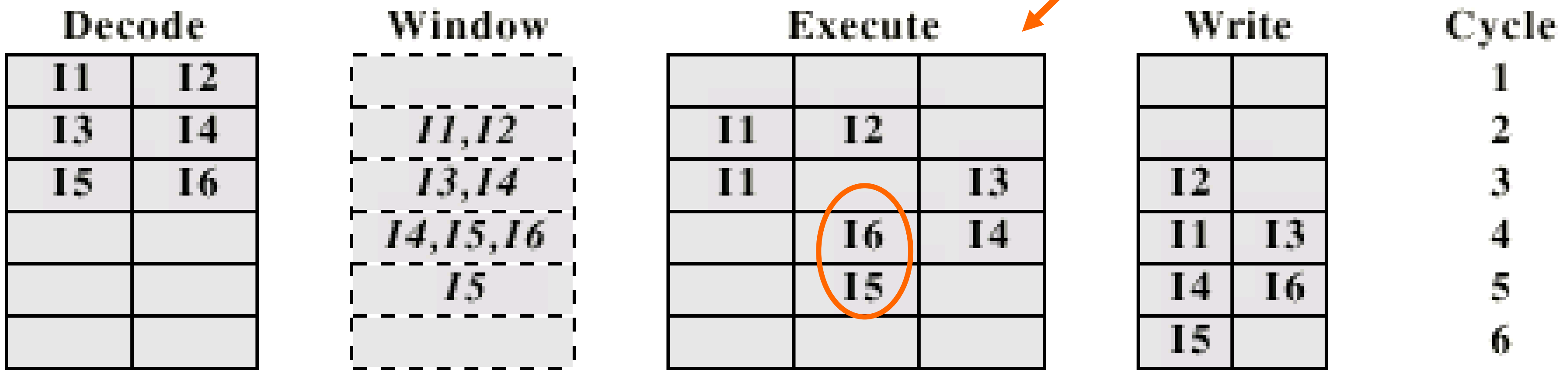
# Out-of-Order Issue Out-of-Order Completion

제한:
-I1을 실행하는데 두 사이클이 필요.
-I3/I4 resource conflict. I5/I6도 resource conflict
-I5는 I4의 결과값을 사용함.  ※ I1과 I2는 data dependency를 가지지 않음.



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

In-Order Issue Out-of-Order Completion



| 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 |

각 cycle에서 두 개의 명령어들이 인출되어 decoding 단계로 들어감

가정:

I5는 I4에 의존하지만, I6는 I4에 의존하지 않음

→ 이 때문에, **I6가 I5보다 먼저 issuing이 되어 실행될 수 있음.**

→ **I5와 I6의 순서를 바꿔서 수행함**

→ Execute 및 write 단계에서 1 cycle save가능

제한:
-I1을 실행하는데 두 사이클이 필요하다
-I3와 I4는 동일한 functional unit을 사용하고자 하며, 이 때문에 서로 conflict함
-I5는 I4의 결과값을 사용함
-I5와 I6는  동일한 functional unit을 사용하고자 하며, 이 때문에 서로 conflict함
※ I1과 I2는 data dependency를 가지지 않음.

## Antidependency

- Write-after-read dependency
  - R3:=R3 + R5; (I1)
  - R4:=R3 + 1; (I2)
  - R3:=R5 + 1; (I3)
  - R7:=R3 + R4; (I4)
  - I3 can not complete before I2 starts as I2 needs a value in R3 and I3 changes R3
  - Antidependency is used because the constraint is similar to that of a true data dependency, but reversed
  - Instead of the first instruction producing a value that the second instruction uses, the second instruction destroys a value that the first instruction uses.

Cf. Read-after-write dependency: true data dependency

## Register Renaming

- Out-of-order issue with out-of-order completion may give rise to the possibility of output dependencies and antidependencies.
- Output and antidependencies occur because register contents may not reflect the correct ordering from the program
- May result in a pipeline stall
- Registers allocated dynamically
  - i.e. registers are not specifically named

Register renaming : duplication of resource !

R3:=R3 + R5; (I1)
R4:=R3 + 1; (I2)
R3:=R5 + 1; (I3)
R7:=R3 + R4; (I4)

# Register Renaming example

- R3b:=R3a + R5a (I1)
- R4b:=R3b + 1 (I2)
- R3c:=R5a + 1 (I3)
- R7b:=R3c + R4b (I4)

참고) 이전 예에서는 I1과 I2가 R3로 인한 data dependency가 없다고 봤음. (즉, I1 실행 결과인 R3를 I2에서 읽는게 아니고 I2의 R3는 I1의 원래의 R3값임)

I1과 I2의 우측, R3 값을 각각 R3a와 R3b로 복사했으며, I1,I2가 동시에 실행될 수 있게 함 (레지스터 하나를 두개의 명령어가 동시에 읽을 수 없기 때문에)

하지만, 만약 I1과 I2가 동시에 실행되지 않고, out of order issue에 의해, 어떤 다른 명령어가 I1과 I2 사이에 들어와서, I1이 실행이 끝나서 R3값이 업데이트 된 후에, I2가 읽힌다면, I2의 실행 결과는 원래 의도한 것과 다른 값이 될 수도 있음(I1, I2 동시 실행 보장해야 함)

즉, 쉽게 생각하면, 이를 해결하는방법은 I1과 I2의 우측 R3를 모두 R3a로 하면 되는 것 같은데, 안되는 이유는 레지스터 R3a를 동시에 읽을 수 없기때문에 이렇게 함

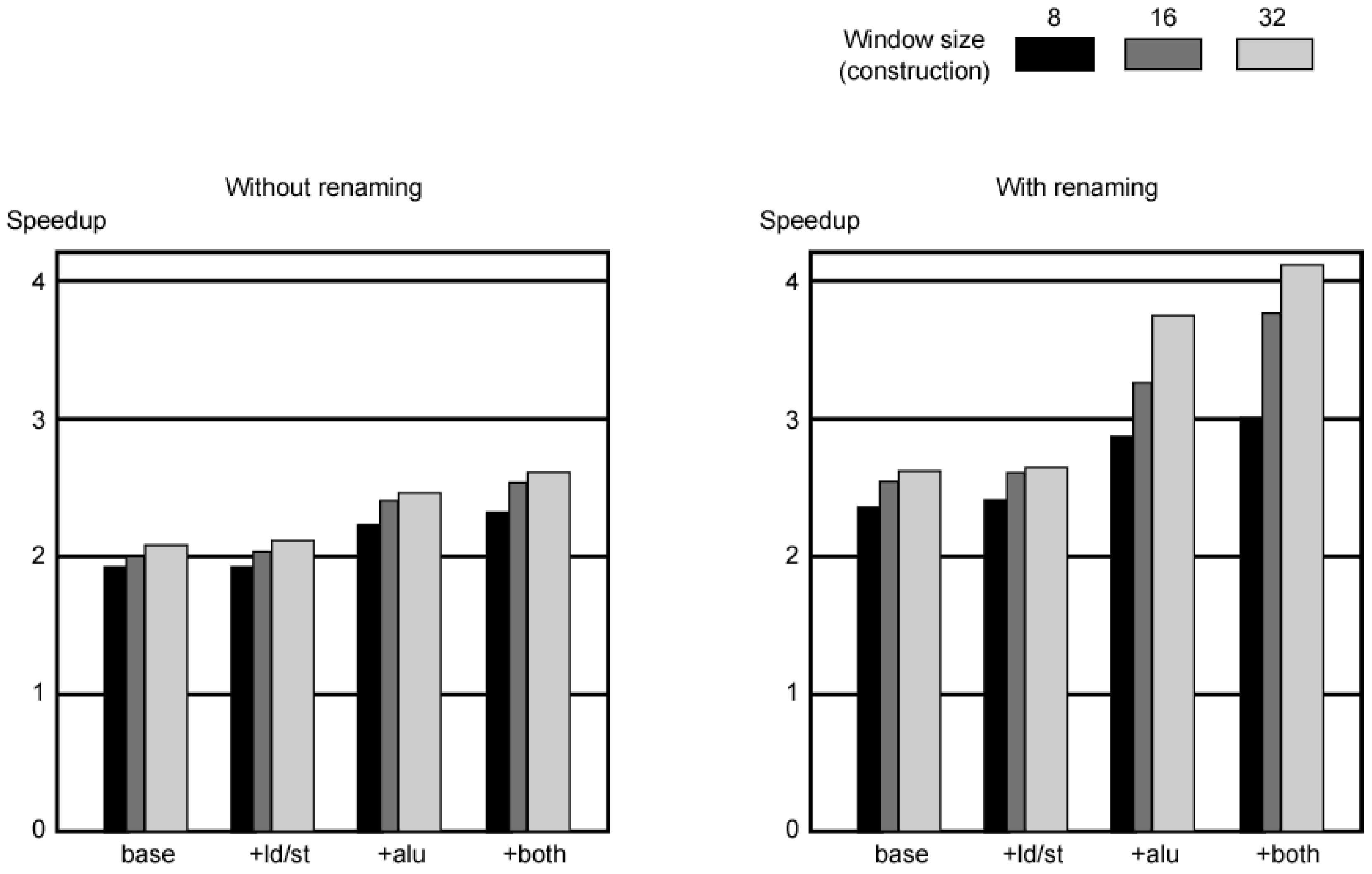
- Without subscript refers to logical register in instruction
- With subscript is hardware register allocated
- Note R3a R3b R3c

- 그리고 R3c를 I3에서 사용함으로서 I2와의 antidependency 문제가 없어졌으며,
- I1과 I3의 output dependency도 없어짐.

## Machine Parallelism

- Duplication of Resources
- Out of order issue
- Renaming
- Not worth duplication functions without register renaming
- Need instruction window large enough (more than 8)

# Speedups of Machine Organizations Without Procedural Dependencies



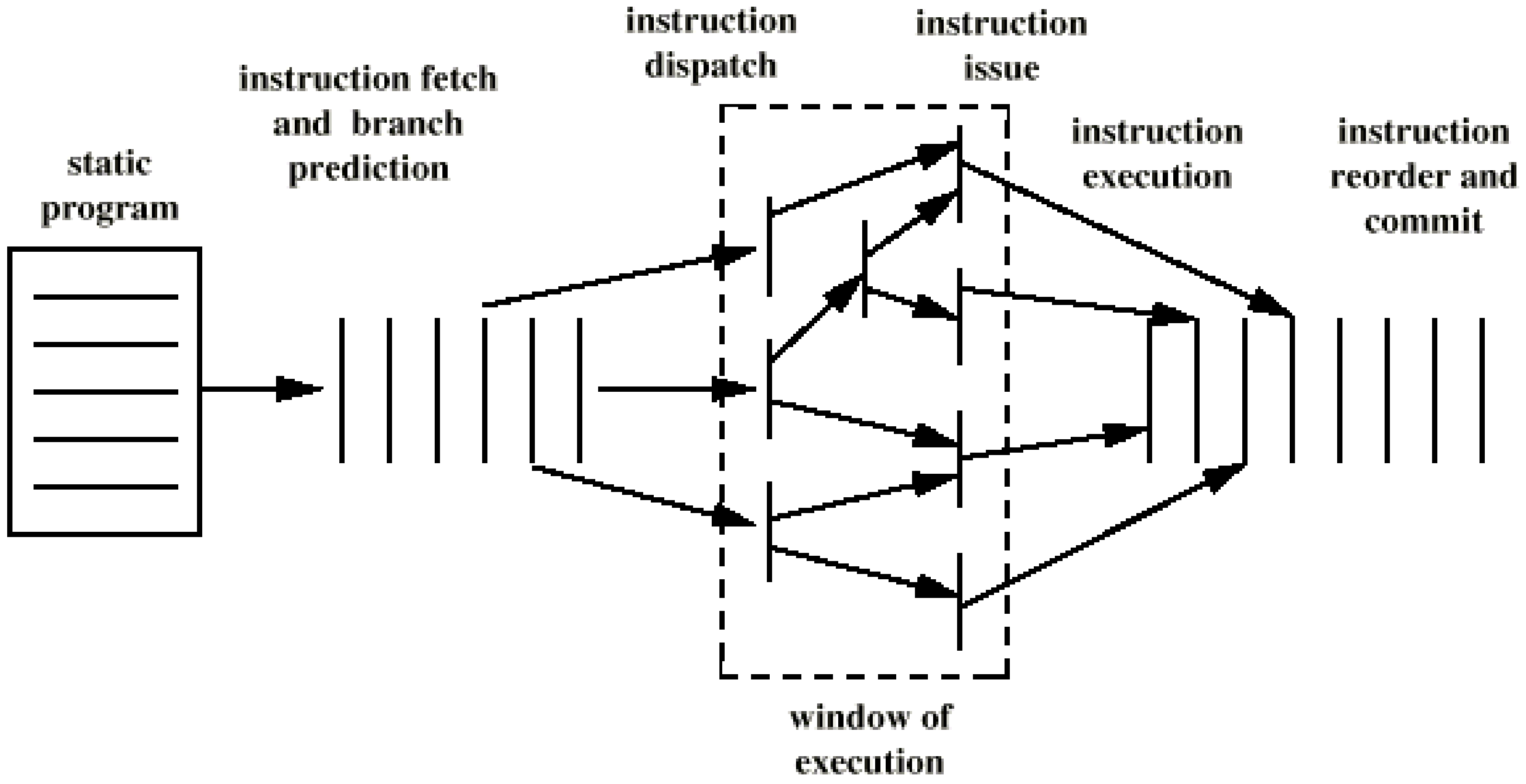
## Branch Prediction

- 80486 fetches both next sequential instruction after branch and branch target instruction
- Gives two cycle delay if branch taken

## 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



## 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