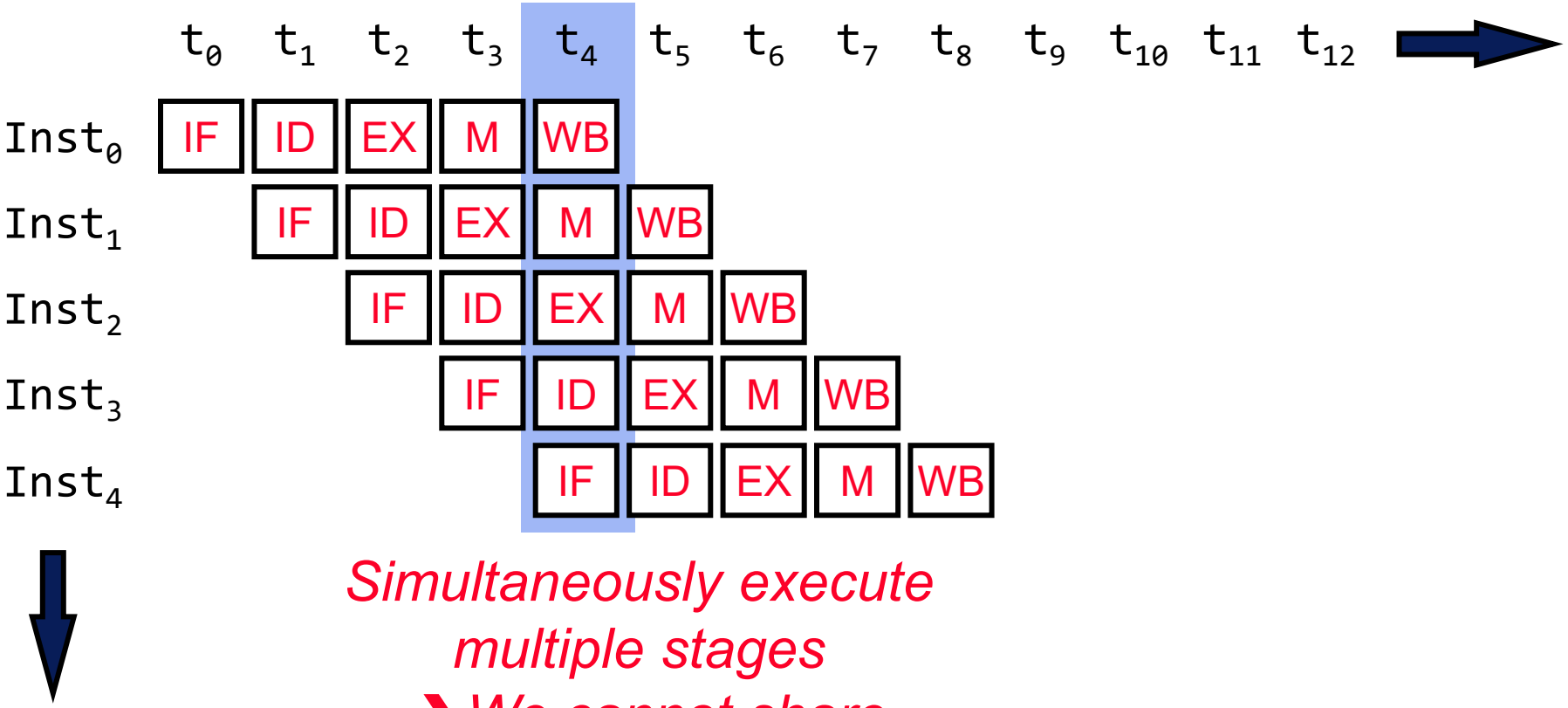
Lecture 8:
CPU – data hazard

Hunjun Lee
hunjunlee@hanyang.ac.kr

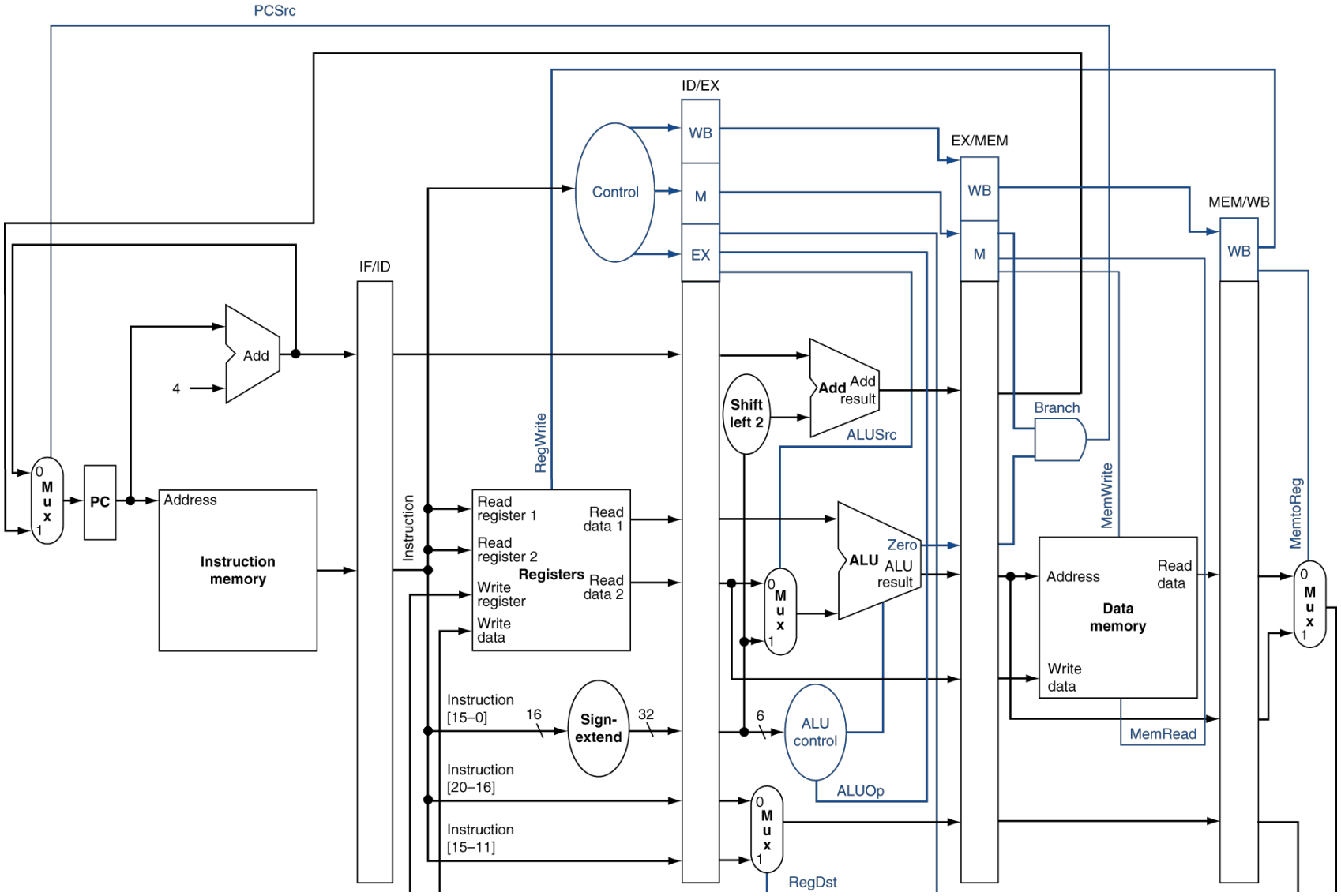
Illustrating 5-stage pipeline operation



Simultaneously execute multiple stages

➔ We cannot share resources

Pipelined Control



In Reality

- No identical operations **e.g., R-type, I-type & J-type**

  ⇒ Unify instruction types

  - Combine instruction types to flow through “multi-function” pipe

- No uniform sub-operations **e.g., RF read VS. Memory write?**

  ⇒ Balance pipeline stages

  - Stage-latency calculation to make balanced stages

- No independent operations **e.g., Waiting data to be produced?**

  ⇒ Remove dependency and/or busy resources

  - Inter-instruction dependency detection and resolution

There are three types of pipeline hazards

- Data Hazard
  - Data dependency (Read-after-write: RAW)
  - Anti dependency (Write-after-read: WAR)
  - Output dependency (Write-after-write: WAW)
- Control Hazard
  - Data dependency of program counter
- Structural Hazard
  - Due to the lack of resources
    - Ex1) We need to split the instruction and data memory (to simultaneously execute IF and MEM stage
    - Ex2) We may execute multiple instructions in the EX stage (#ALU < # instructions ready)

Data dependence

Today's topic!

Data dependence

```
op r3 r1 r2
op r5 r3 r4
```

Read-after-Write (RAW)

Anti-dependence

```
op r3 r1 r2
op r1 r4 r5
```

Write-after-Read (WAR)

Later…

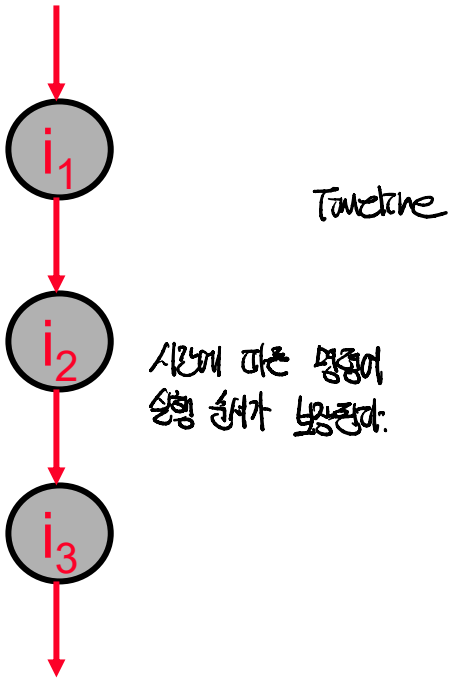
Output-dependence

```
op r3 r1 r2
op r5 r3 r4
op r3 r6 r7
```

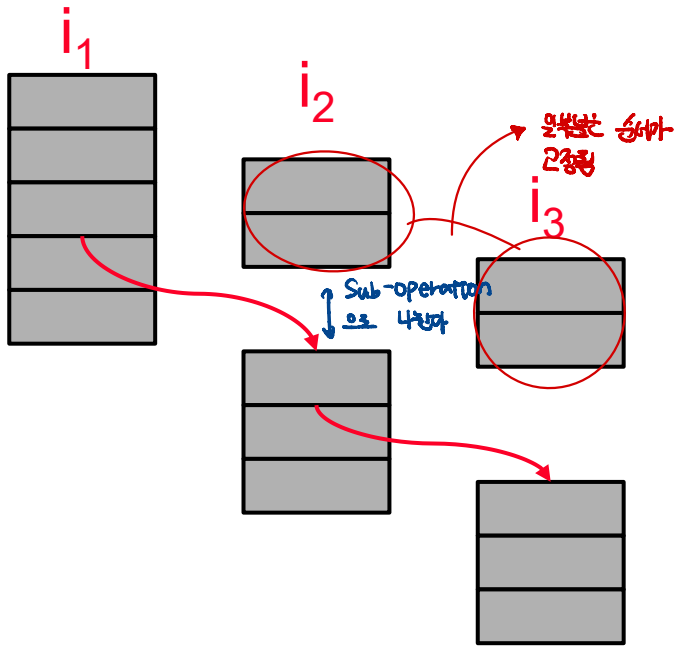
Write-after-Write (WAW)

Dependencies and Pipelined Execution

Sequential and **Atomic** instruction semantics



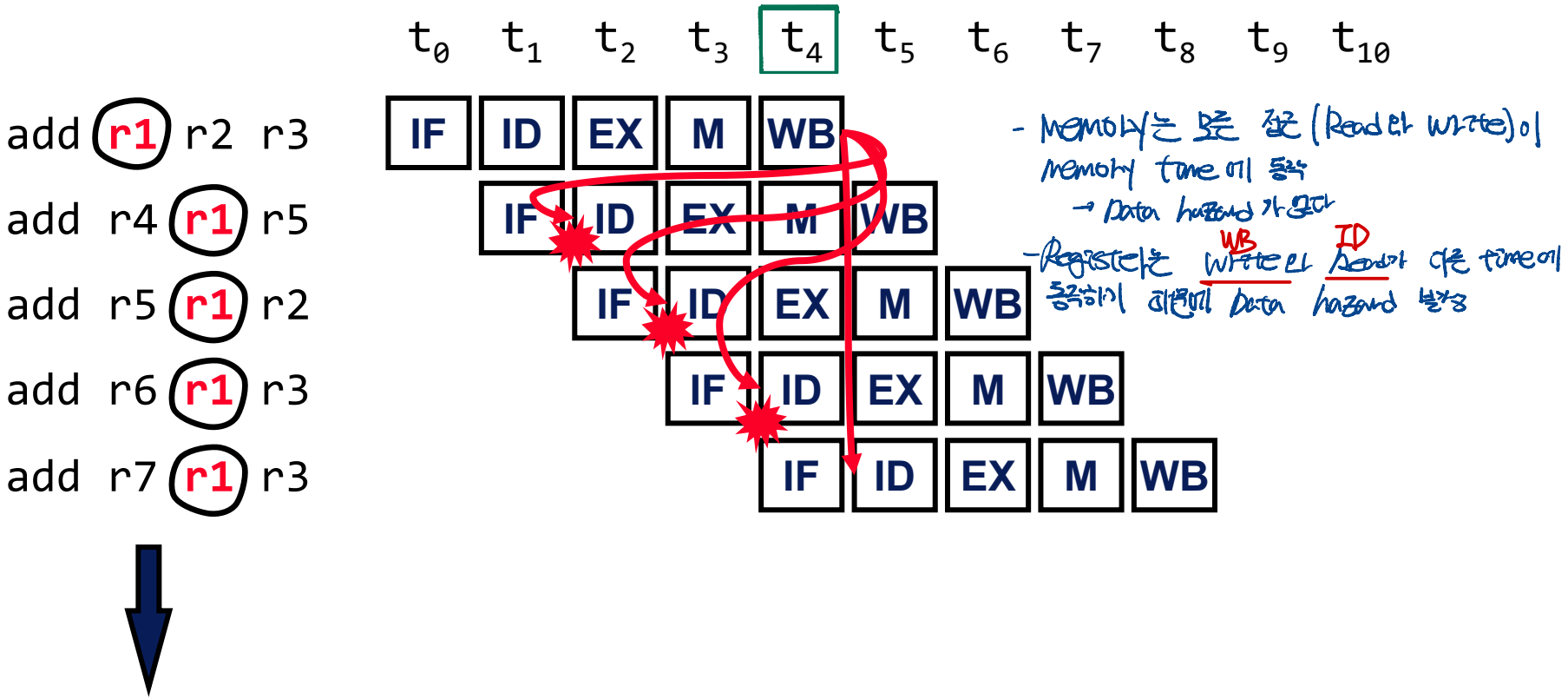
Two instructions may require ordering of (only) **sub-operations**



Defines what is correct
➔ Architecture

Defines what is implemented
➔ Architecture

RAW Dependency and Hazard



We cannot execute ID stage of the following instructions until the first instruction undergoes WB stage

Register data hazard analysis

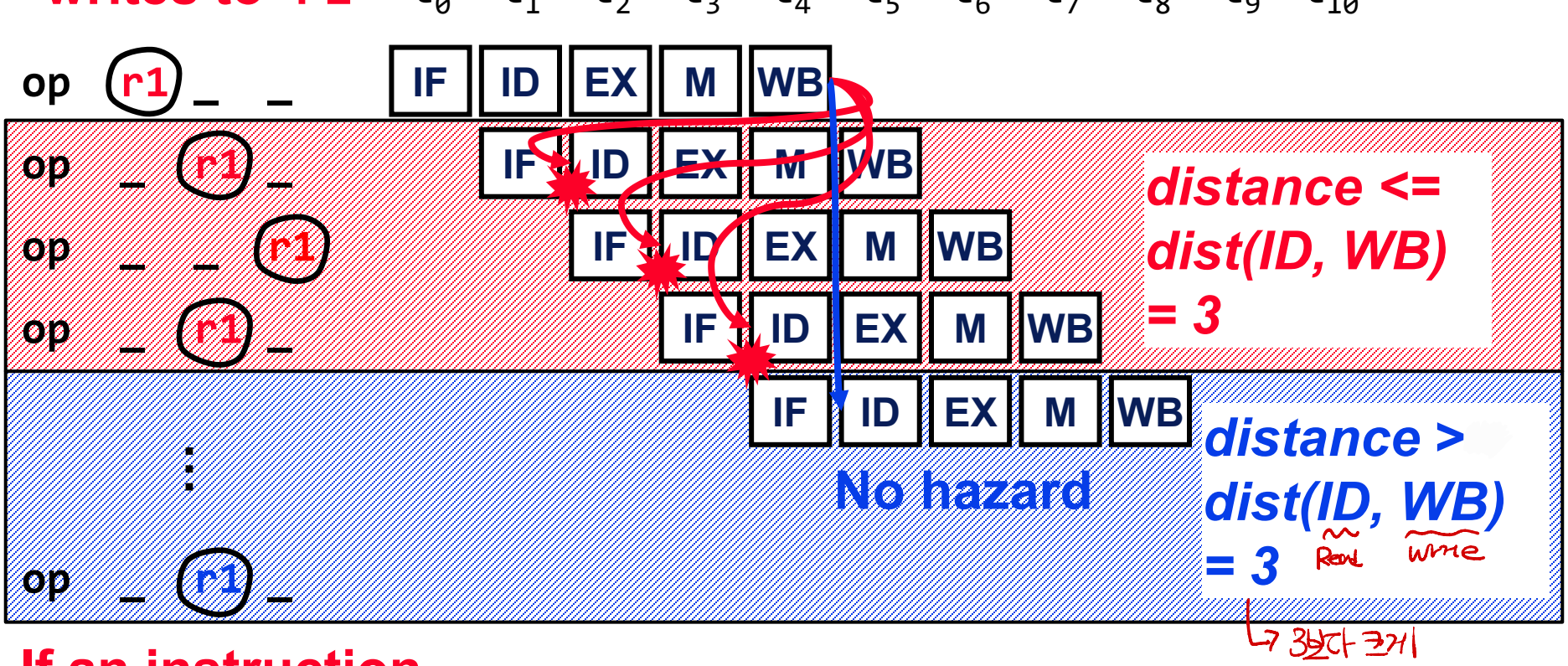
	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	use	use	use	use		use
EX						
MEM						
WB	produce	produce				

The problem is related to (1) the distance between two instructions and (2) the distance between **"use"** and **"produce"** stages

➔ Two dependent R/I-type instructions should be separated by at least dist(ID,WB)+1, which is "4"

Necessary condition for RAW hazards

If an instruction writes to r1

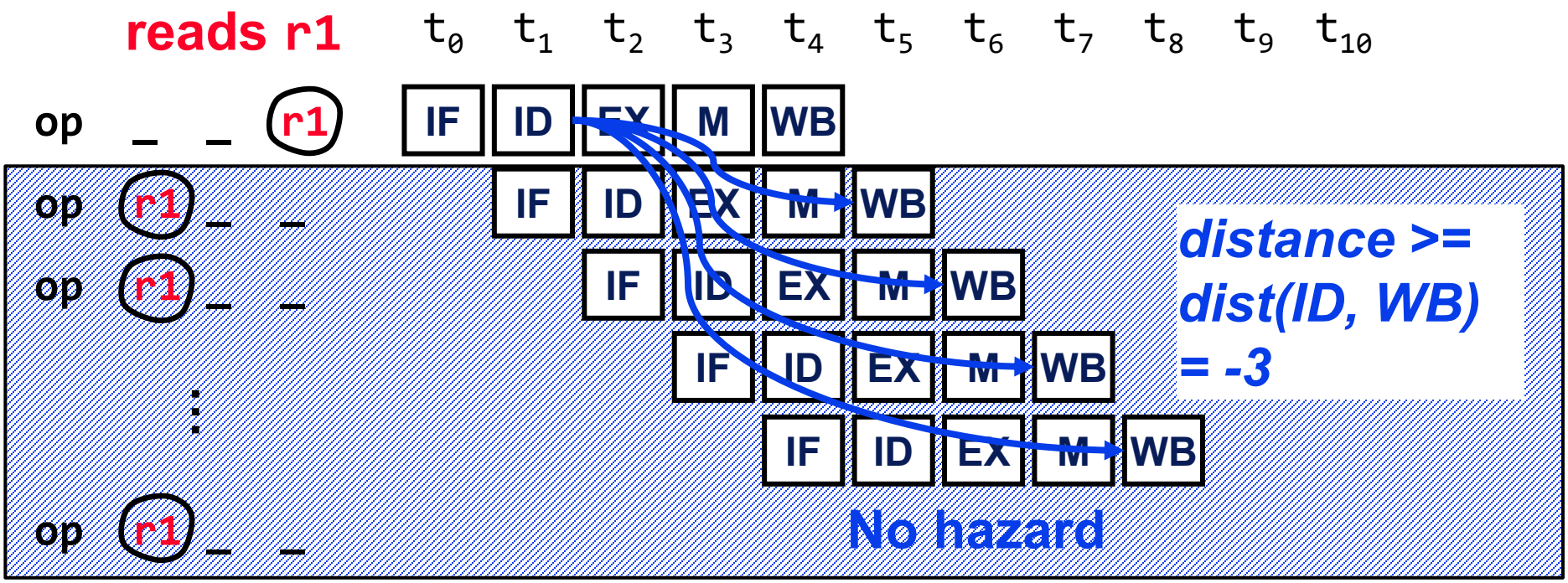


If an instruction utilizes r1 as an operand

What about WAW and WAR hazards?

WAR / WAW hazard

If an instruction reads r1



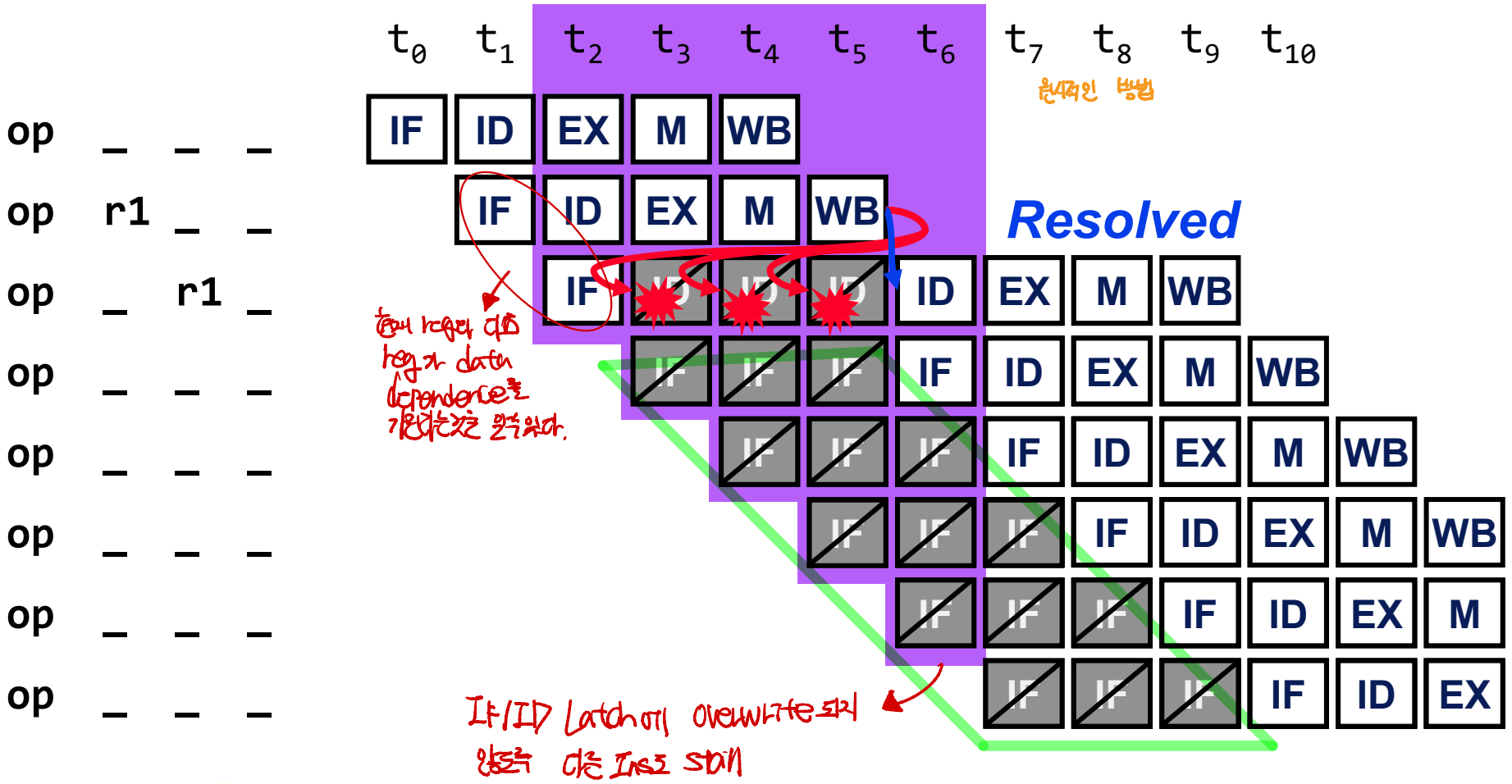
If an instruction writes to r1

Actually, there is no such thing as an WAR / WAW hazard if the CPU processes instructions in-order

→ 명령어가 순서대로 실행되는 경우, WAR과 WAW는 전혀 문제 없음 등등 etc.

➔ ***You will learn out-of-order CPU later …***

A universal hazard resolution: "Pipeline Stall" (= let's wait) resolve

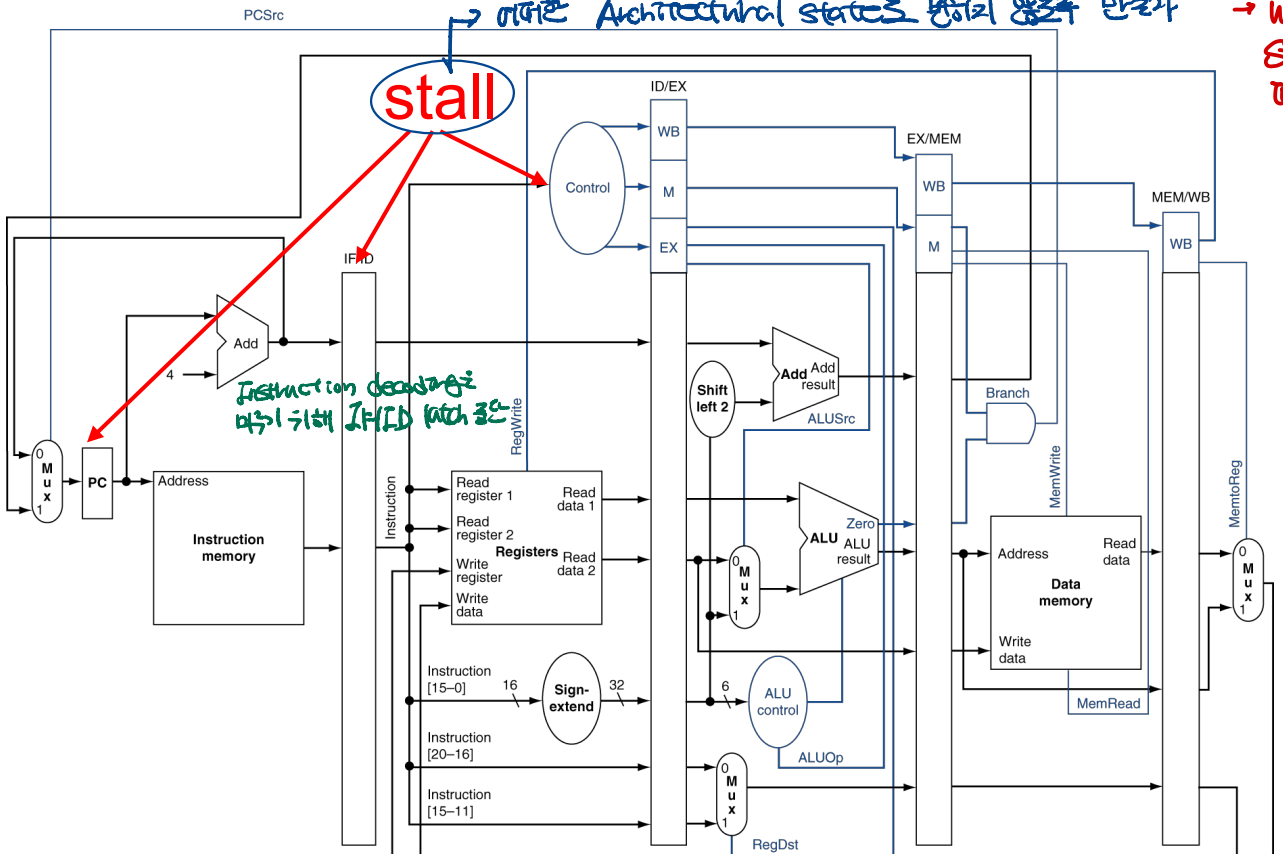


Stall: make younger instructions wait until hazard is resolved

Stop all up-stream stages & Drain all down-stream stages

What is stall?

- We should not modify programmer visible states during stall (bubble)
  - Disable PC and IR latching
  - Control should set RegWrite=0 and MemWrite=0



When and how long to stall?

- Instructions I_A and I_B (where I_A comes before I_B) have RAW hazard iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $Dist(I_A, I_B) \leq dist(ID, WB) = 3$ ← Hazard 가능성↑
  - Stall cycles: $dist(ID, WB) - Dist(I_A, I_B) + 1$

$= 4 - Dist(I_A, I_B)$ I_A와 I_B가 떨어진 거리

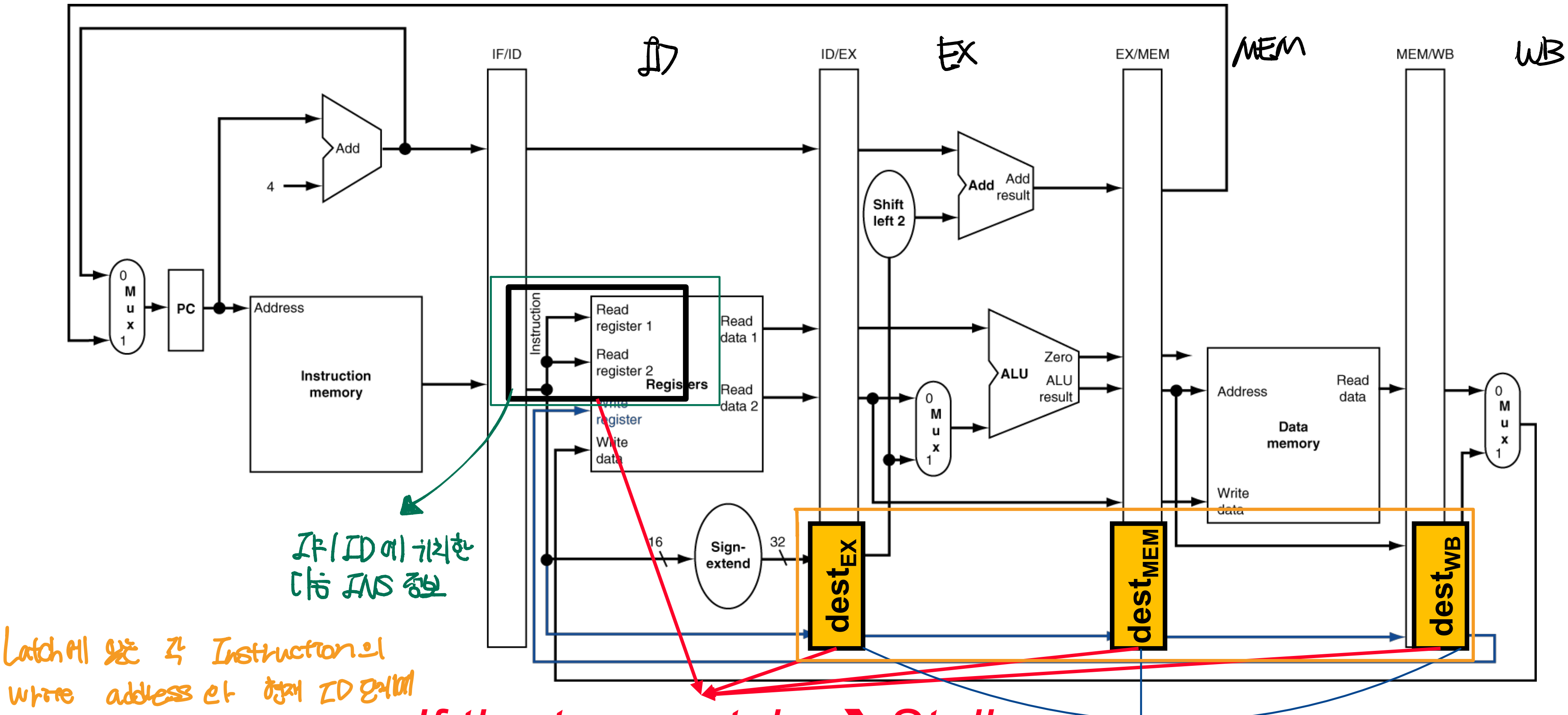
- In other words, we must stall I_B in ID stage if it wants to read a register **to be written** by **ALL** I_A that might exist in EX, MEM or WB stage

I_A가 끝나기 전까지 반드시 Stall 해줘야 한다.

Stall condition

ID 단계에서 Read register는 확인하기 전에 Data Hazard를 확인해야한다.
→ Hazard Detection Unit는 IF/ID latch 내부에 있는 $rs, $rt 값을 보고 Hazard Detection을 확인한다.

Latch에 있는 Write back address 확인



If the two match ➔ Stall

→ EX, MEM, WB 에서 사용하는 dest_reg

멀리 떨어져있는 두 단계가 Data dependency가 있다면 n cycle stall을 걸어준다.
↳ 종류에 따라 다르다.

Stall condition

- Helper functions
  - rs(I) returns the rs field of I (instruction register)
  - use_rs(I) returns true if I requires **RF**[rs]

- Stall when
  - (rs($\mathbf{IR_{ID}}$)==$dest_{EX}$) && use_rs($\mathbf{IR_{ID}}$) && $RegWrite_{EX}$ // 3 Cycles
  - (rs($\mathbf{IR_{ID}}$)==$dest_{MEM}$) && use_rs($\mathbf{IR_{ID}}$) && $RegWrite_{MEM}$ // 2 Cycles
  - (rs($\mathbf{IR_{ID}}$)==$dest_{WB}$) && use_rs($\mathbf{IR_{ID}}$) && $RegWrite_{WB}$ // 1 Cycle

  - (rt($\mathbf{IR_{ID}}$)==$dest_{EX}$) && use_rt($\mathbf{IR_{ID}}$) && $RegWrite_{EX}$ // 3 Cycles
  - (rt($\mathbf{IR_{ID}}$)==$dest_{MEM}$) && use_rt($\mathbf{IR_{ID}}$) && $RegWrite_{MEM}$ // 2 Cycles
  - (rt($\mathbf{IR_{ID}}$)==$dest_{WB}$) && use_rt($\mathbf{IR_{ID}}$) && $RegWrite_{WB}$ // 1 Cycle

It is crucial that the EX, MEM and WB stages continue to advance as normal during these stall cycles

Impact of Stall on Performance

- Without stall

  Ideal IPC = 1 ← Because of pipelining

- With stall
  - Each stall cycle corresponds to 1 lost cycle
  - For a program with N instructions and S stall cycles,

    Average IPC (with stall) =N/(N+S) S cycle stall

- S depends on

  Stall ↑ → IPC ↓

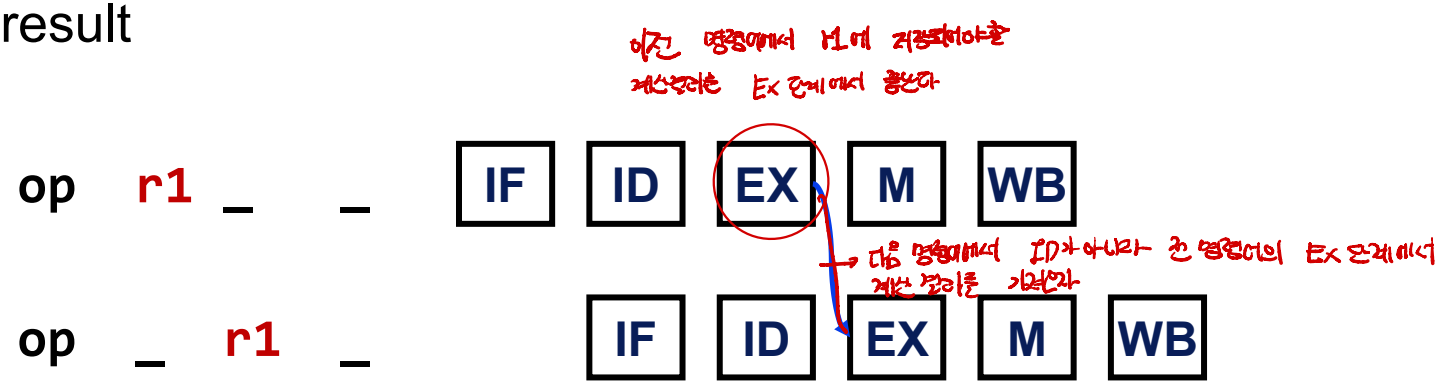
  Stall의 크기는 줄여 해결
  - Frequency of RAW hazards
  - Exact distance between the hazard-causing instructions
  - Hazard stall ↓

Data forwarding (= register bypassing)

- It is intuitive to think of RF as 'flow of right data'
  - "add rx ry rz" literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]
  - So, "add rx ry rz" really means

(1) Get "the computation results" of the previous instructions to define the values of RF[ry] and RF[rz], respectively, and

(2) Until another instruction redefines RF[rx], younger instructions that refers to RF[rx] can use this instruction's result



Why not by directly use the computation results that are not "yet" written to the register file?

Resolving RAW Hazard by Forwarding

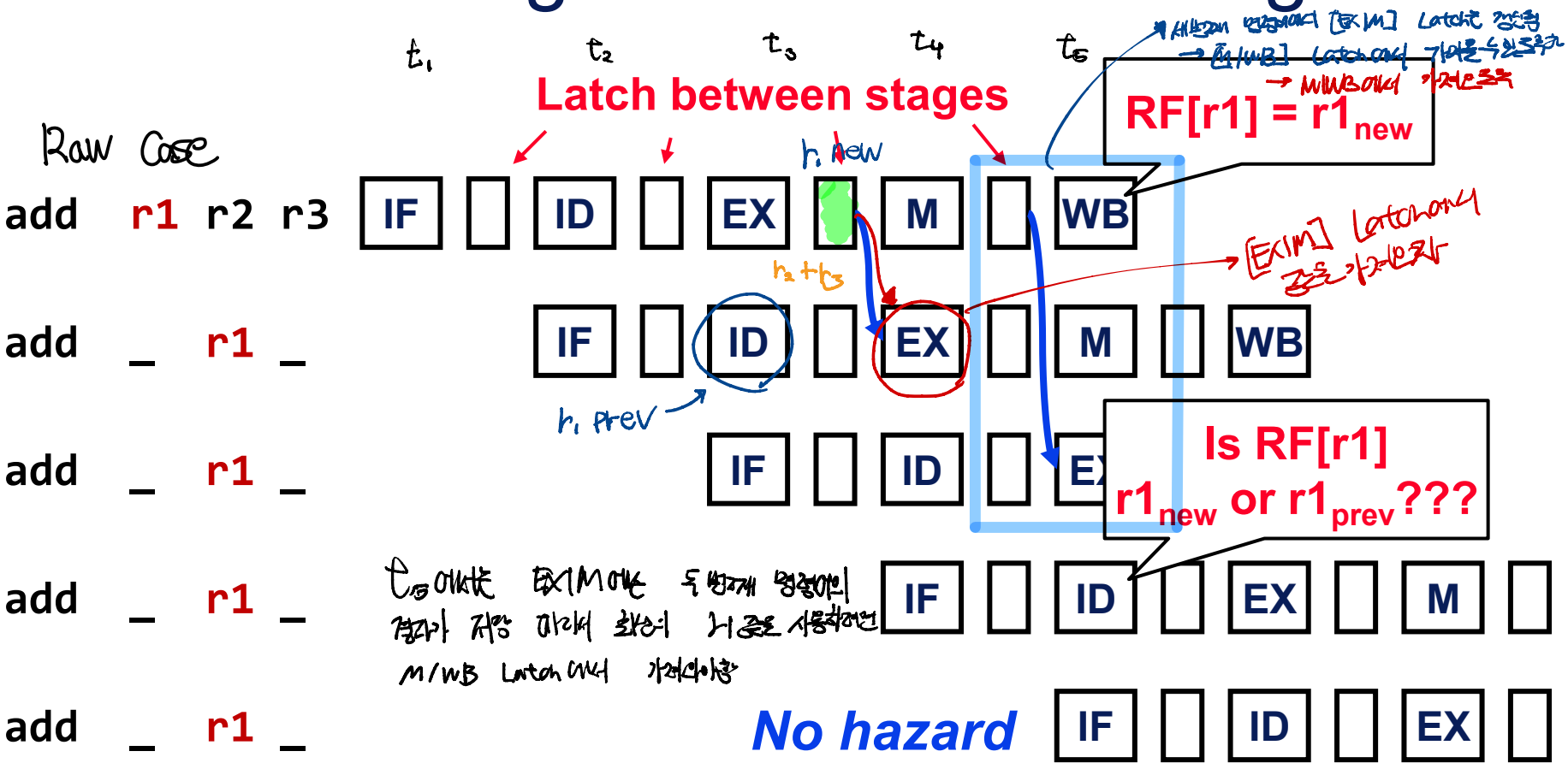
- Instructions I_A and I_B (where I_A comes before I_B) have RAW hazard iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $dist(I_A, I_B) \leq dist(ID, WB) = 3$

- In other words, if I_B in ID stage reads a register written by I_A in EX, MEM or WB stage, then the operand required by I_B is not yet in RF

  → Retrieve operand **"from datapath"** instead of waiting until the RF is updated!

  → Retrieve operand **"from the youngest definition"** if multiple definitions are outstanding!

Looking closer at forwarding



We need to transfer the data from the latches to the execution stages

Internal RF forwarding

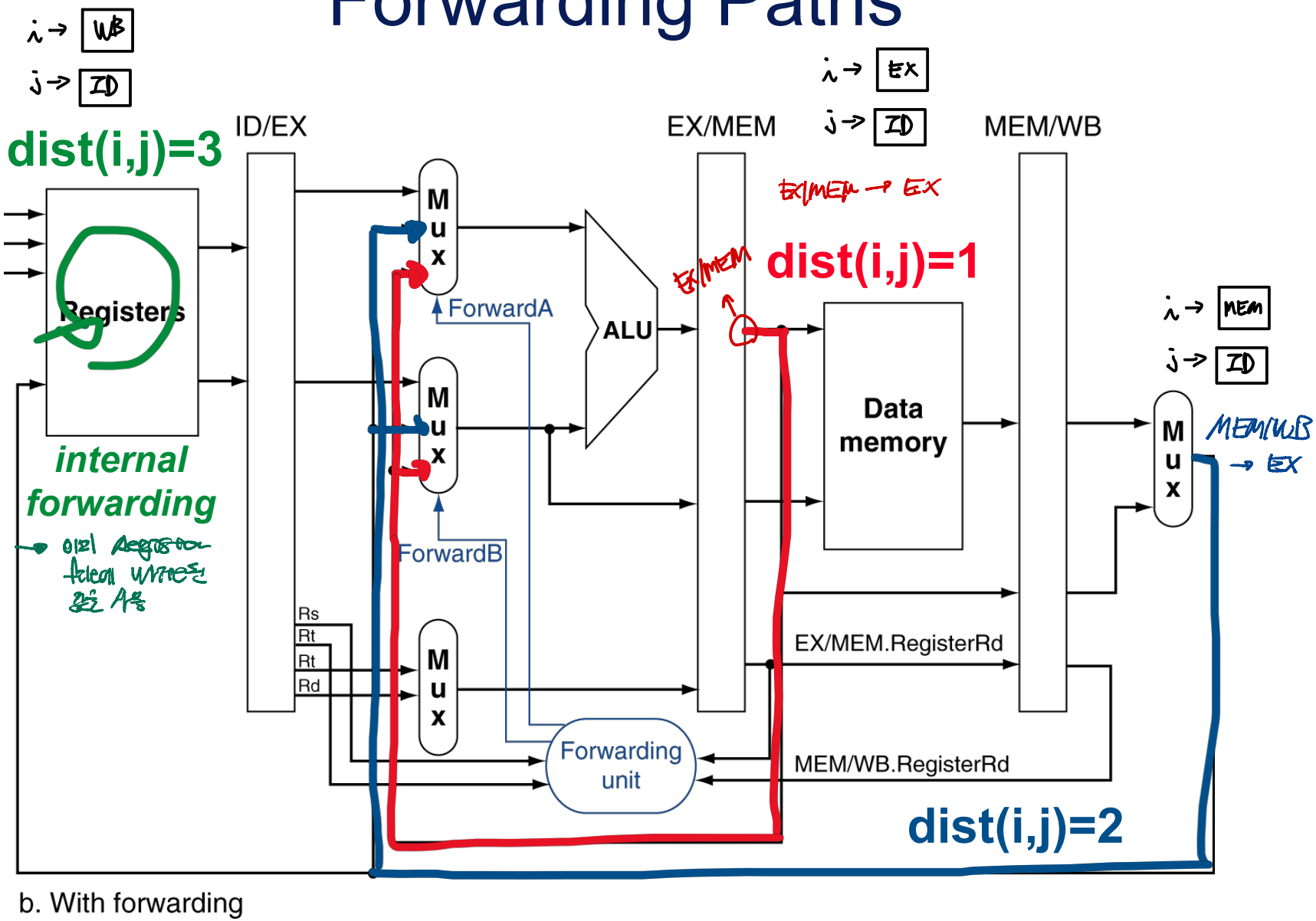
- We may design an RF that internally forwards data

```
// Verilog-style pseudocode
module RF (clk, raddr, rdata, waddr, wdata);
        // port declaration
        …
        // storage
        reg [31:0] RF_data [0:31];

        always @(*) begin
                if(waddr == raddr1)   rdata = wdata;
                else                  rdata = RF_data[raddr];
        end

        always @(posedge clk) begin
                // write RF
                RF_data[waddr] <= wdata;
        end
endmodule
```

Forwarding Paths



b. With forwarding

Forwarding Logic

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file. Register file에서 그대로 가져다 사용 : RAW 발생 X
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

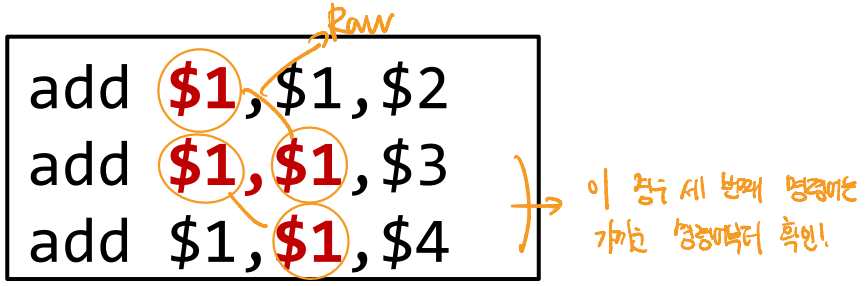
rs != 0 ?
→ rs == 0 : Zero reg

Forwarding Logic

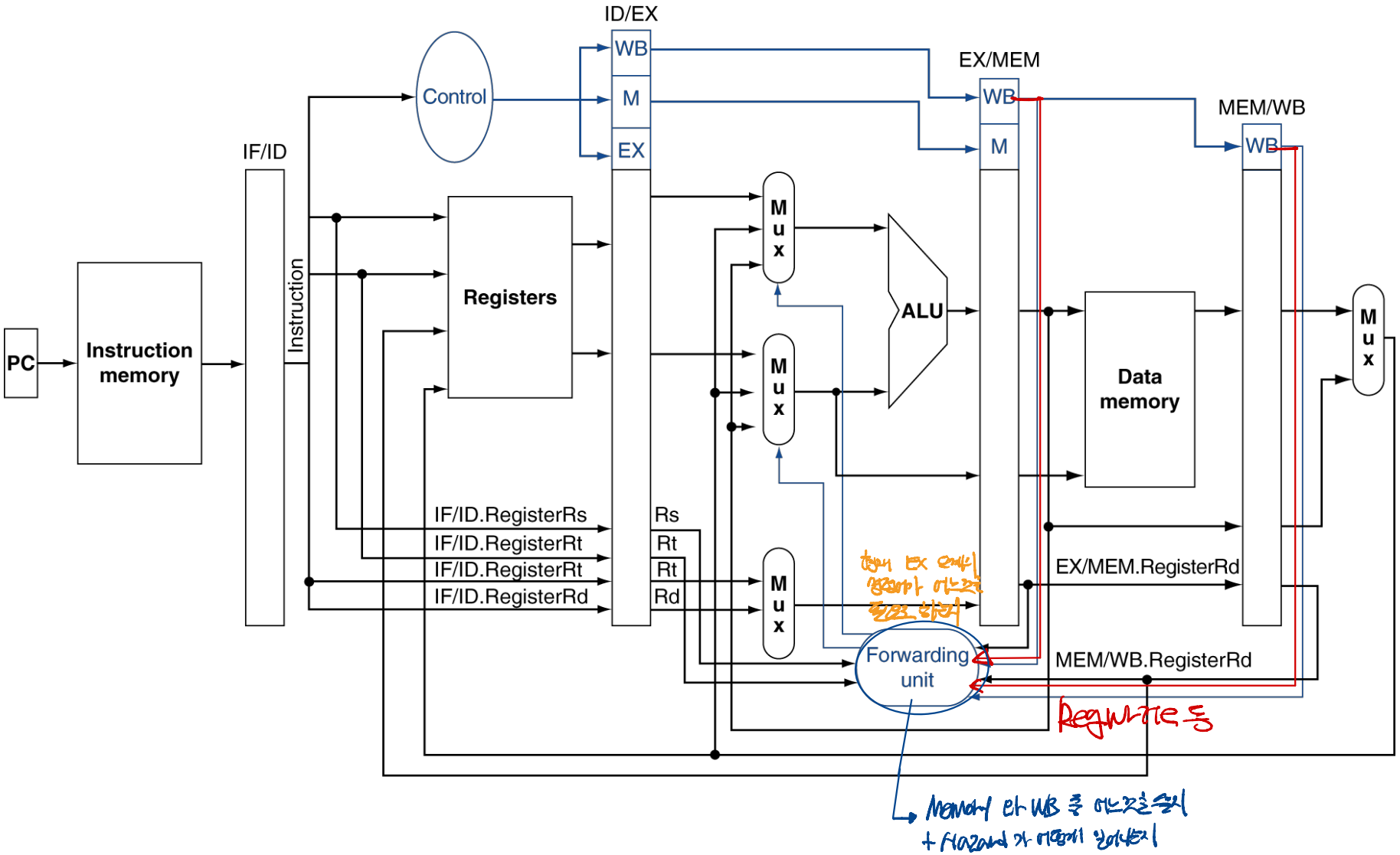
항상 0으로 고정된 Register
→ 굳이 forwarding할 필요없다

if (rs_{EX}!=0) && (rs_{EX}==$dest_{MEM}$) && $RegWrite_{MEM}$ then
    forward operand <u>from MEM stage</u> // dist=1
else if (rs_{EX}!=0) && (rs_{EX}==$dest_{WB}$) && $RegWrite_{WB}$ then
    forward operand <u>from WB stage</u> // dist=2
else
    use the operand from register file // dist >= 3

Ordering matters!! Consider the following …

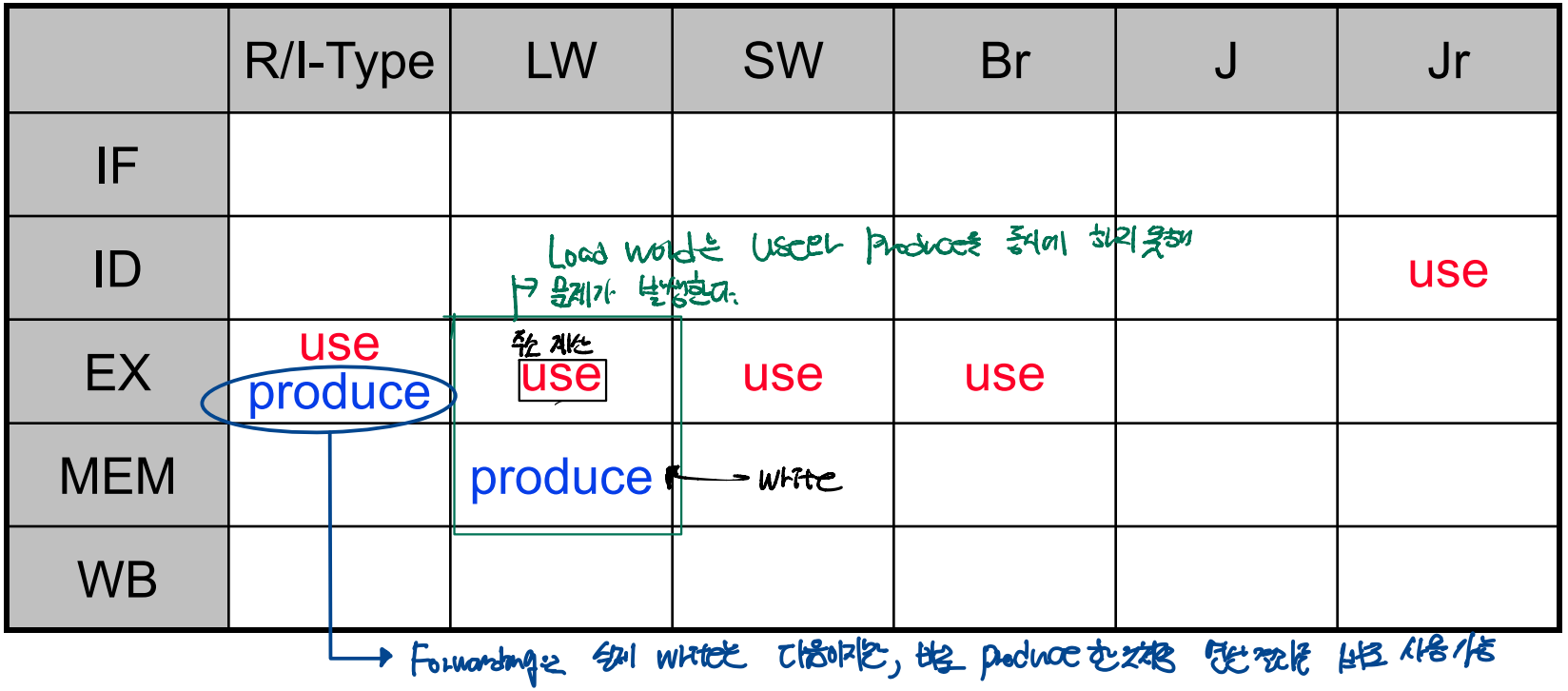


Datapath with forwarding



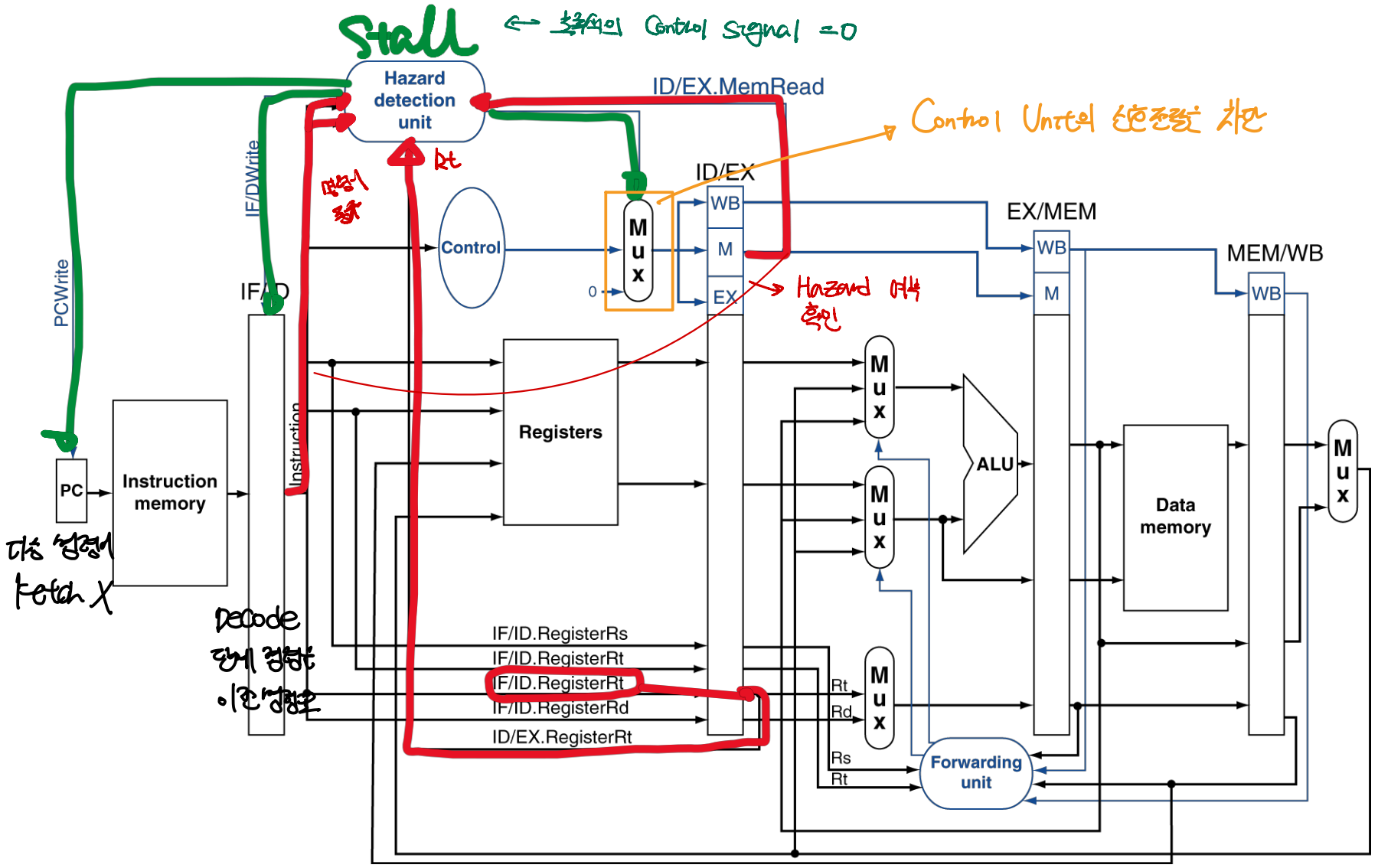
Data Hazard Analysis (with Forwarding)

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID						use
EX	use	use	use	use		
MEM	produce	produce				
WB						



- Even with data-forwarding, RAW dependence on an immediate preceding **LW instruction produces a hazard**

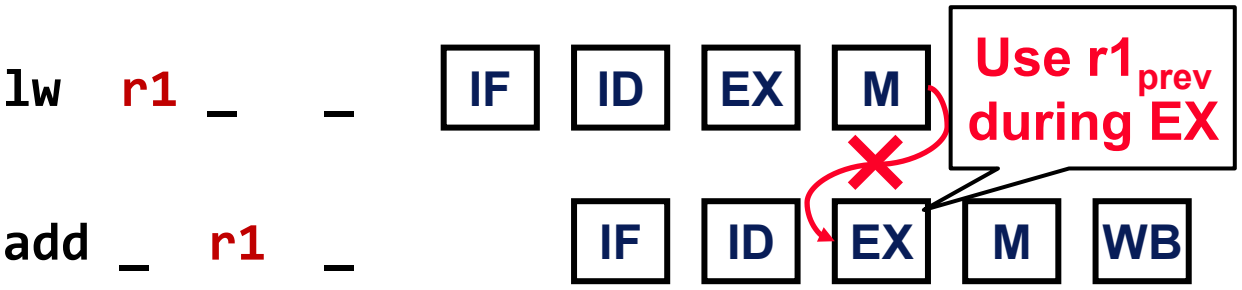
Datapath with hazard detection



Two ways to handle hazards

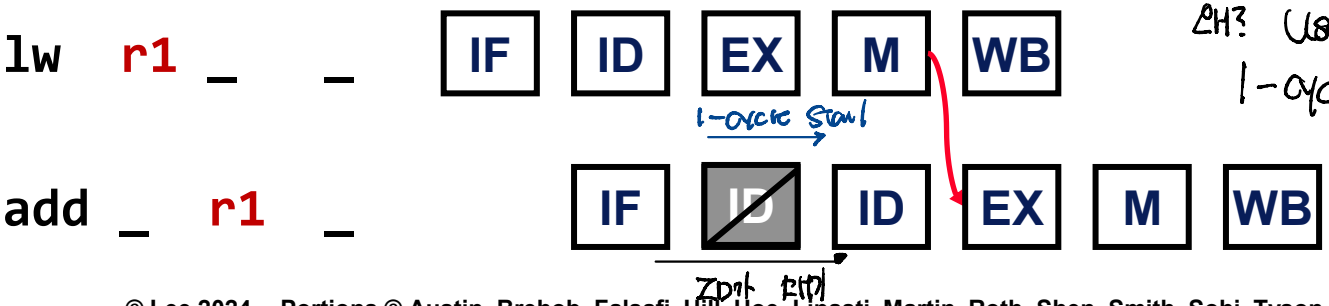
- Option 1) Let the software handle the hazards
  - The instruction immediately following a load sees **the old value**
  - The software (e.g., compiler) should handle the hazard (e.g., by inserting nop or by scheduling instructions)

하드웨어는 아무런 대응을 하지 않고, 컴파일러가 NOP을 삽입하든 등의 방식으로 처리하도록 한다.



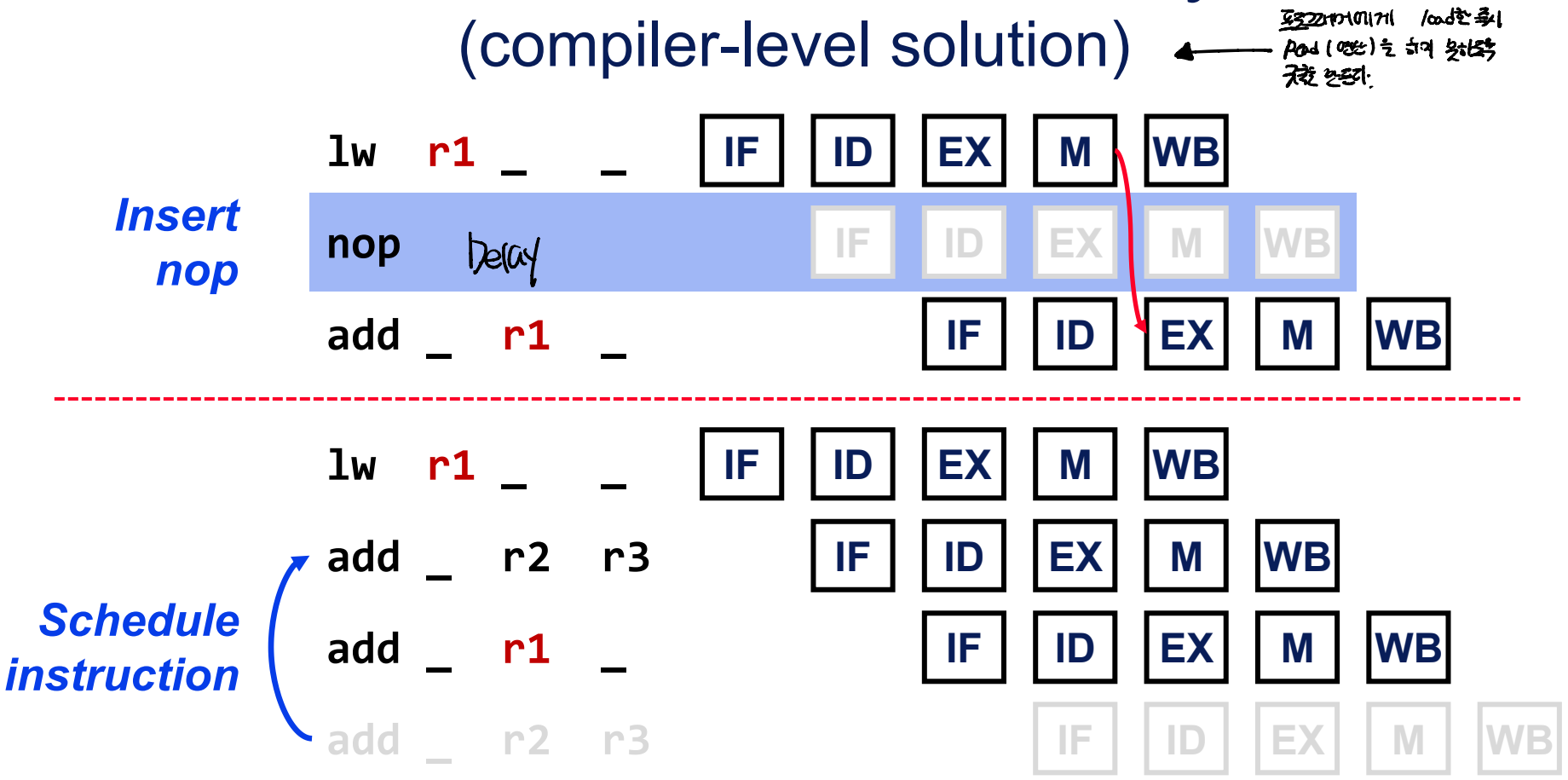
- Option 2) Use stall on load operation
  - A dependent immediate successor to LW **must stall 1 cycle in ID**
  - Stall = (rs(IR_{ID})==$dest_{EX}$) && use_rs(IR_{ID}) && $MemRead_{EX}$

1-cycle stall을 추가한다. 왜? User와 producer 1-cycle 차이



Another solution: load delay slot

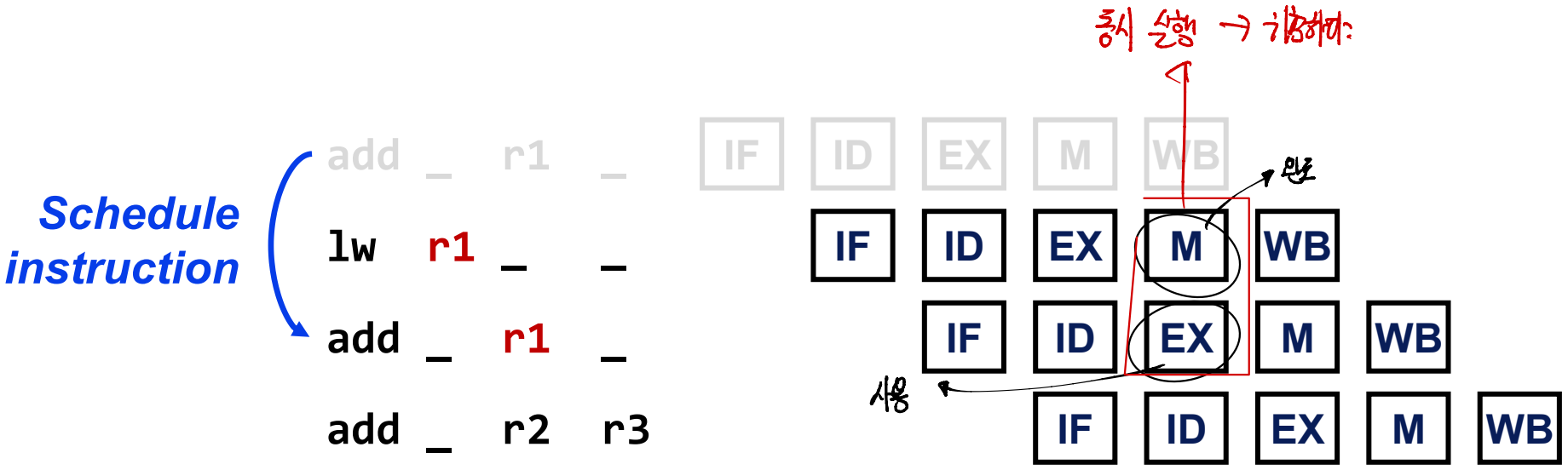
(compiler-level solution)



- Some ISA exposes the pipeline stages, so that the compiler may modify the code to remove data hazards
  - Insert nop or schedule independent instruction in between

Load delay slot!

- Is this possible?



Terminologies

- Dependencies
  - Ordering requirement between instructions
- Pipeline Hazards:
  - (Potential) violations of dependencies due to hazards
- Hazard Resolution:
  - **Static** → Schedule instructions at compile time to avoid hazards
  - **Dynamic** → Detect hazard and adjust pipeline operation

Stall, Flush or Forward

- Pipeline Interlock: Hazard 해결의 위한 Pipeline mechanism
  - Hardware mechanisms for dynamic hazard resolution
  - Detect and enforce dependences at run time

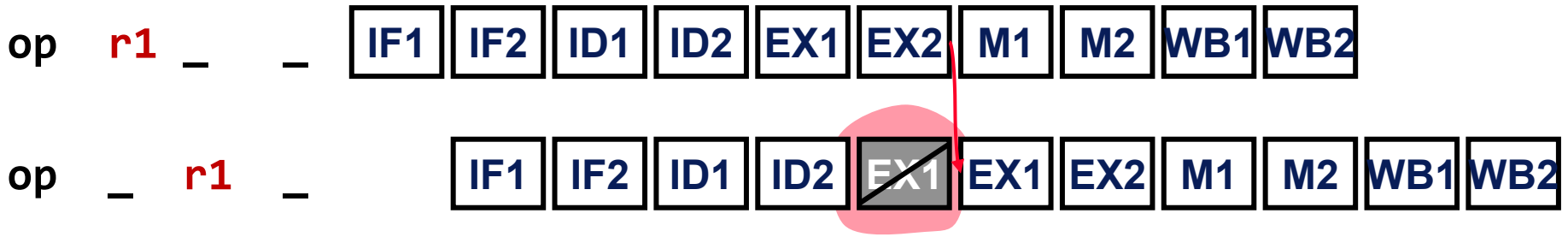
Is 5-stage pipeline sufficient?

5 stage보다 stage 개수를 늘리면? Hazard 가능성, 복잡

→ Pipelining은 몇이 할수 있는가

- There are still plenty of combinational delay between regiters
- "**Superpipelining**" increases pipelining degree such that even intrinsic operations (e.g., ALU, RF read/write, memory access) require multiple stages

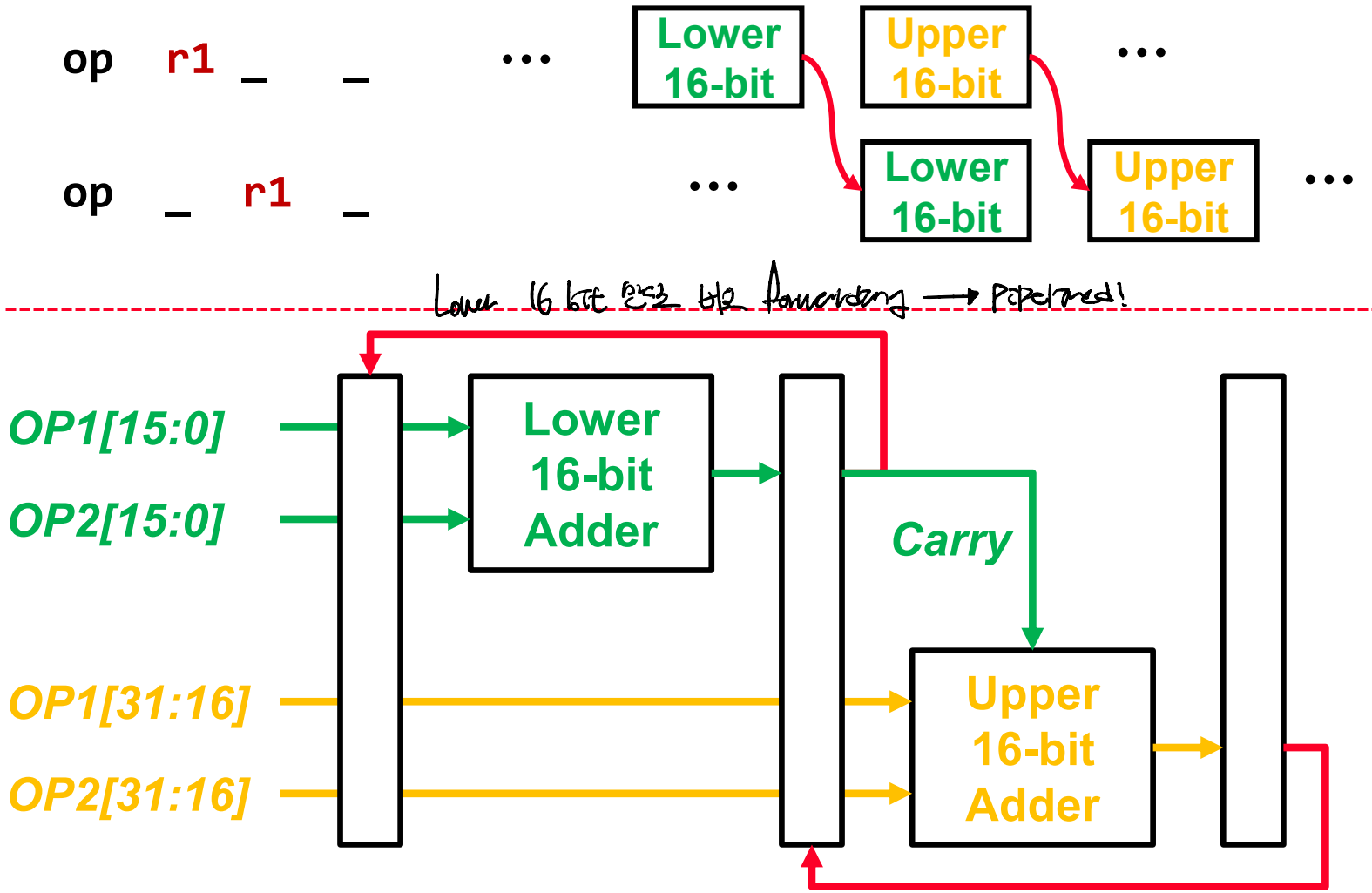
- Potential problem: "more data hazards"



Pipeline stall even w/ forwarding

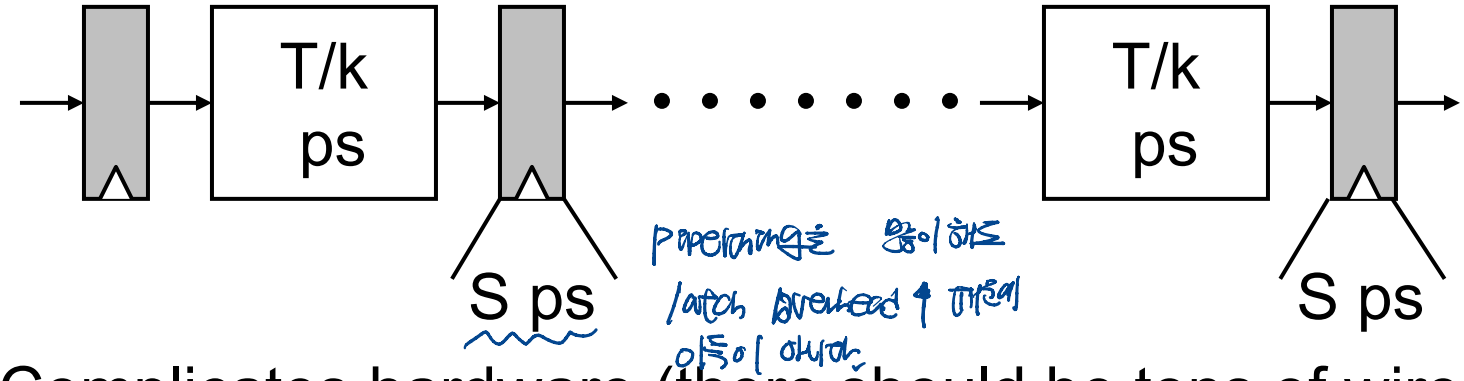
Intel P4's superpipelined Integer ALU

- Intel P4's ALU mitigates the dependency issues



But above all, you cannot pipeline till infinity!

- There are diminishing returns on clock speed



- Complicates hardware (there should be tons of wires for data forwarding)

- You cannot always solve the dependency problem w/ forwarding!

We need to use superscalar!!! (not now …)

Question?

Announcements:

Reading: *finish reading P&H Ch.4*

Handouts: *none*