

# Computer Architecture

## Course code: 0521292B

### 07. Dependence Handling

Jianhua Li

College of Computer and Information  
Hefei University of Technology

slides are adapted from CA course of wisc, princeton, mit, berkeley, etc.


**The uses of the slides of this course are for educational purposes only and should be used only in conjunction with the textbook. Derivatives of the slides must acknowledge the copyright notices of this and the originals.**



# Data Dependence Types

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write  
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read  
(WAR)

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$



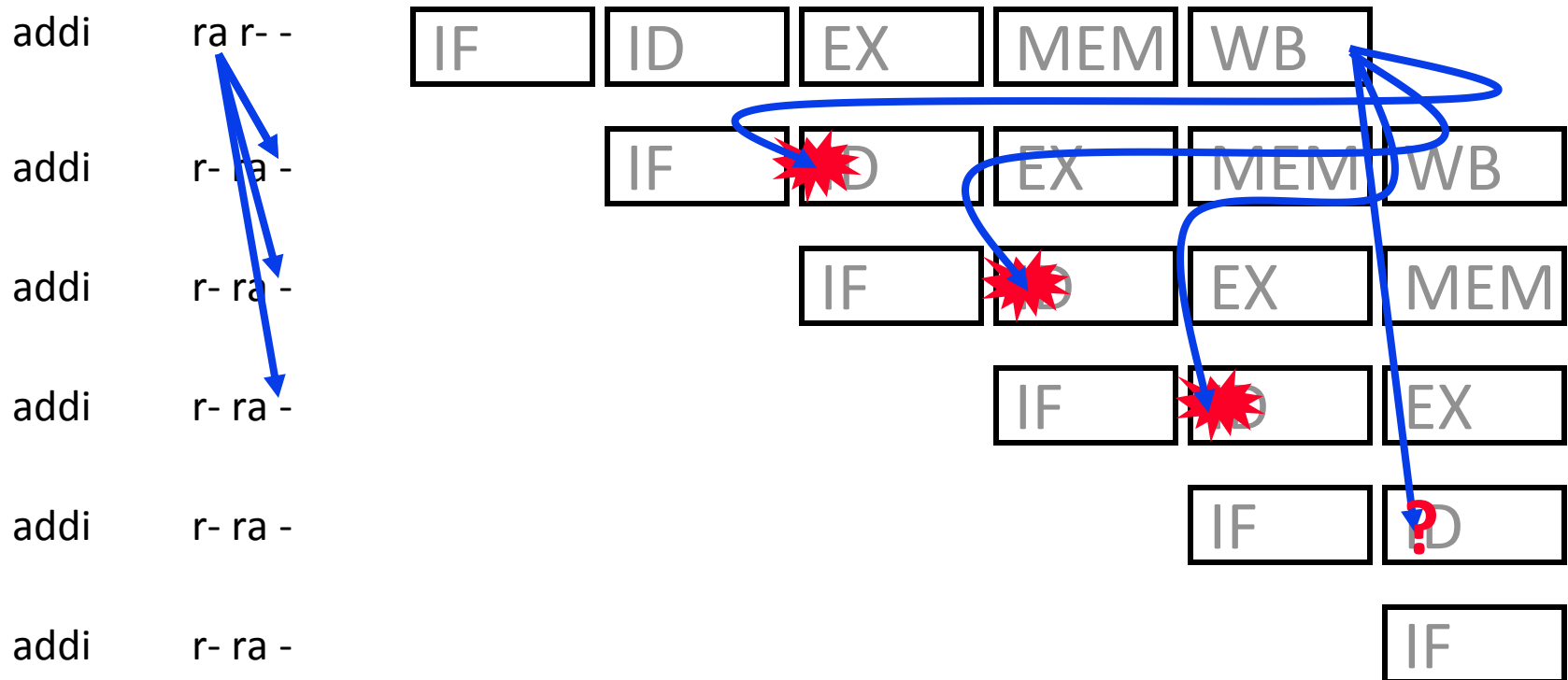
Write-after-Write  
(WAW)

# 记住如何处理数据相关

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Four typical ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute “speculatively”, and verify

# RAW相关处理

- Which one of the following flow dependences lead to conflicts in the 5-stage pipeline?

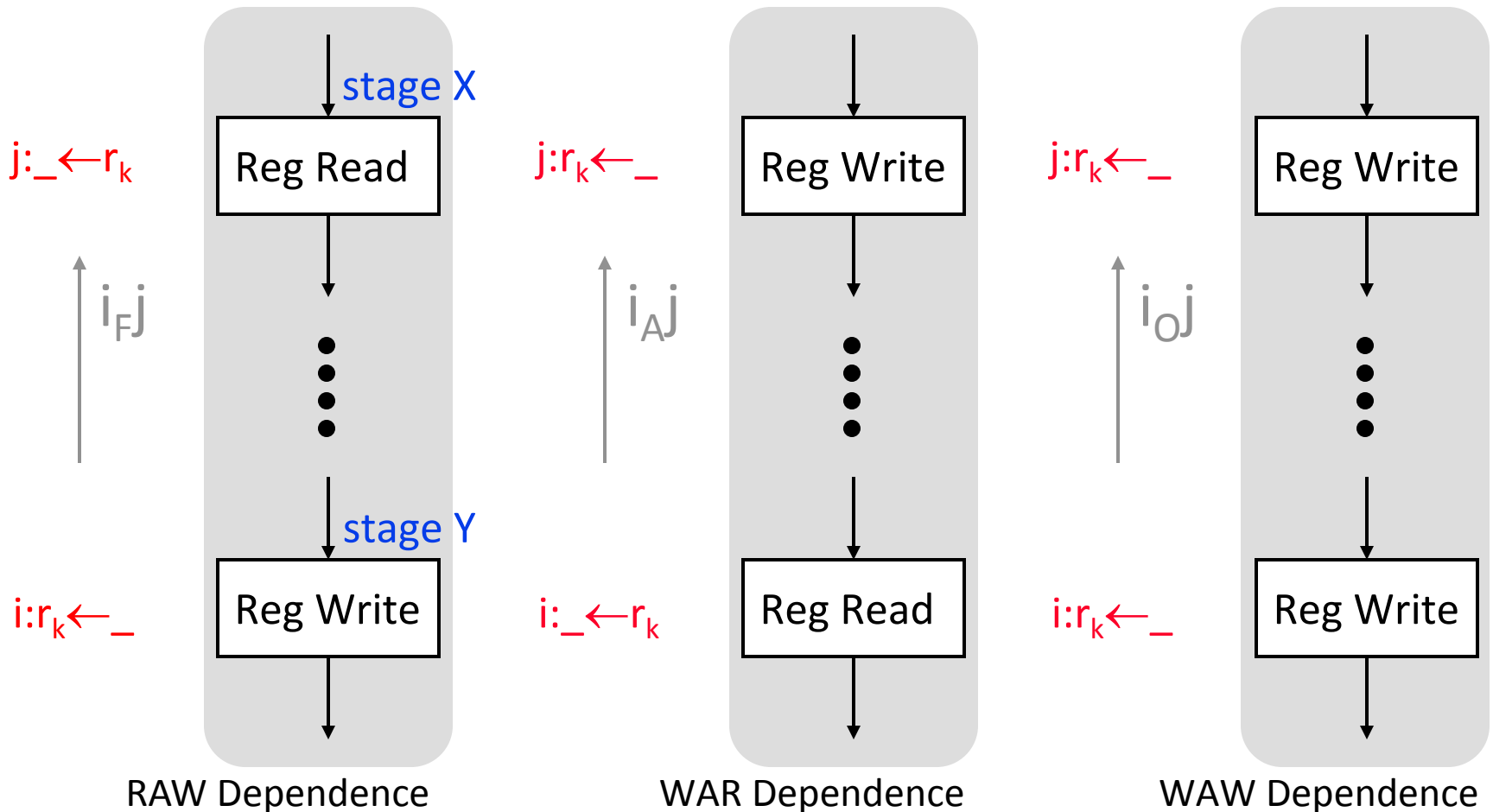


# 寄存器数据相关分析

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

- For a given pipeline, when is there a potential conflict between two data dependent instructions?
  - dependence type: RAW, WAR, WAW?
  - instruction types involved?
  - distance between the two instructions?

# Safe and Unsafe Movement of Pipeline



$\text{dist}(i,j) \leq \text{dist}(X,Y) \Rightarrow$  **Unsafe to keep  $j$  moving**  
 $\text{dist}(i,j) > \text{dist}(X,Y) \Rightarrow$  **Safe**

# RAW 相关分析示例

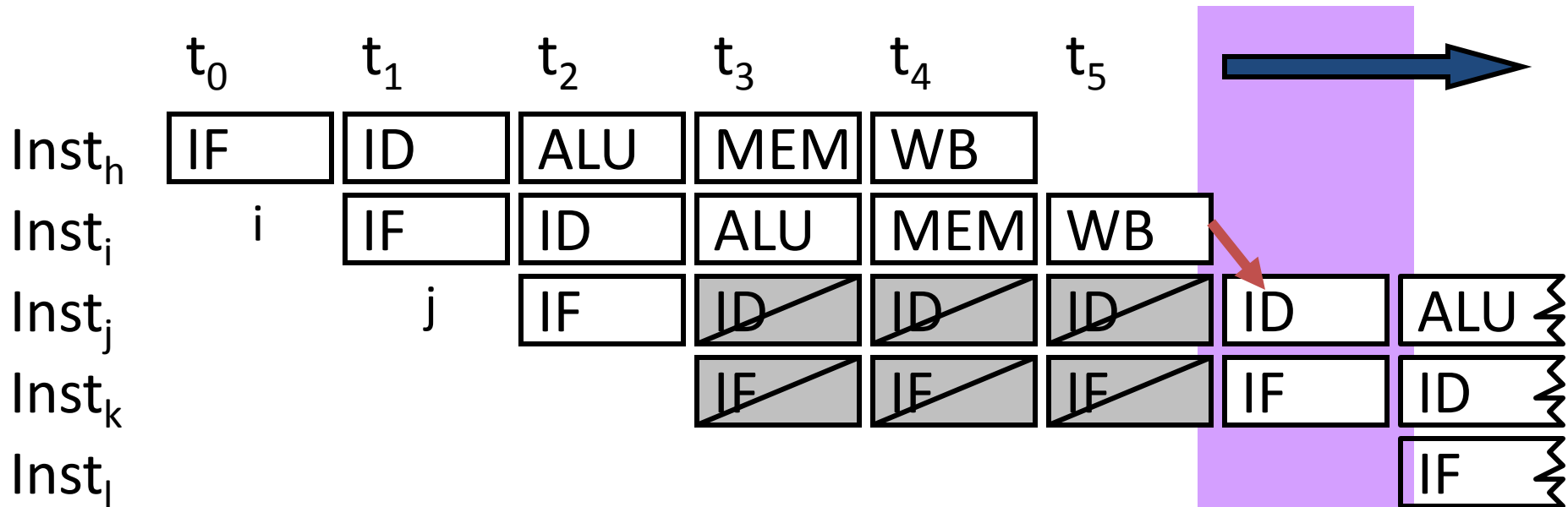
	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

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

What about WAW and WAR dependence?

What about memory data dependence?

# Pipeline Stall: 解决数据相关



$i: r_x \leftarrow \_$   
 bubble  
 bubble  
 bubble  
 $j: \_ \leftarrow r_x$

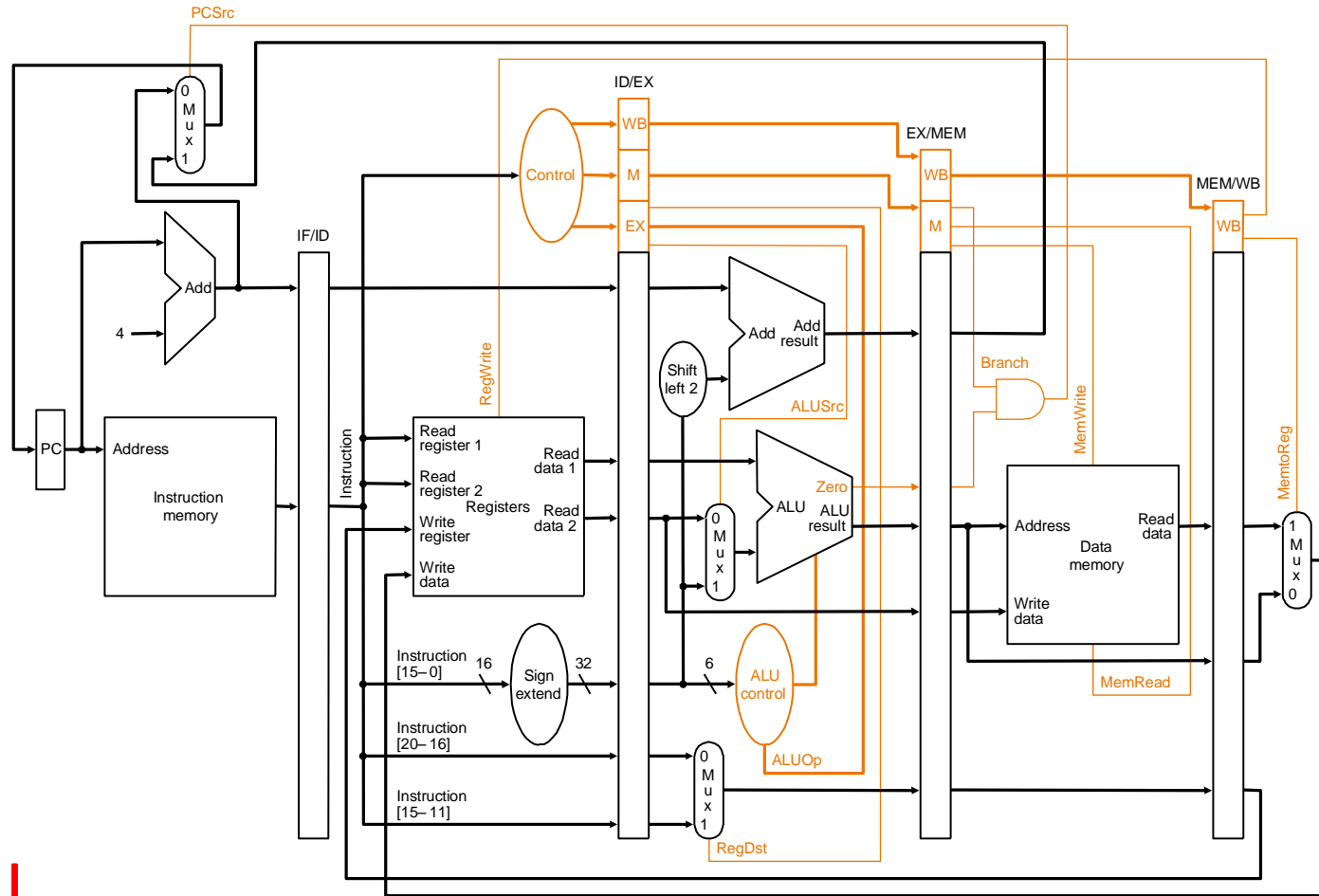
$\text{dist}(i,j)=4$

Stall = make the dependent instruction wait until its source data value is available

1. stop all up-stream stages
2. drain all down-stream stages



# 如何实现Pipeline Stall



- Stall

- disable **PC** and **IR** latching; ensure stalled instruction stays in its stage
- Insert “invalid” **instructions/nops** into the stage following the stalled one (called “bubbles”)

# Stall Conditions

- Instructions  $I_A$  and  $I_B$  (where  $I_A$  comes before  $I_B$ ) have RAW dependence iff
  - $I_B$  (R/I, LW, SW, Br or JR) reads a register written by  $I_A$  (R/I or LW)
  - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$
- Must stall the ID stage when  $I_B$  in ID stage wants to read a register to be written by  $I_A$  in EX, MEM or WB stage

# Stall Condition Evaluation Logic

- Helper functions
  - $rs(l)$  returns the  $rs$  field of  $l$
  - $use\_rs(l)$  returns true if  $l$  requires  $RF[rs]$  and  $rs \neq r0$
- Stall when: (满足下列条件之一)
  - $(rs(IR_{ID}) == dest_{EX}) \ \&\& \ use\_rs(IR_{ID}) \ \&\& \ RegWrite_{EX}$
  - $(rs(IR_{ID}) == dest_{MEM}) \ \&\& \ use\_rs(IR_{ID}) \ \&\& \ RegWrite_{MEM}$
  - $(rs(IR_{ID}) == dest_{WB}) \ \&\& \ use\_rs(IR_{ID}) \ \&\& \ RegWrite_{WB}$
  - $(rt(IR_{ID}) == dest_{EX}) \ \&\& \ use\_rt(IR_{ID}) \ \&\& \ RegWrite_{EX}$
  - $(rt(IR_{ID}) == dest_{MEM}) \ \&\& \ use\_rt(IR_{ID}) \ \&\& \ RegWrite_{MEM}$
  - $(rt(IR_{ID}) == dest_{WB}) \ \&\& \ use\_rt(IR_{ID}) \ \&\& \ RegWrite_{WB}$
- It is crucial that the EX, MEM and WB stages continue to advance normally during stall cycles

# Pipeline Stall对性能的影响

- Each stall cycle corresponds to **one lost cycle** in which no instruction can be completed
  - For a program with N instructions and S stall cycles,  
Average CPI =  $(N+S)/N$
  - S depends on
    - frequency of RAW dependences
    - exact distance between the dependent instructions
    - distance between dependences
- suppose  $i_1, i_2$  and  $i_3$  all depend on  $i_0$ , once  $i_1$ 's dependence is resolved,  $i_2$  and  $i_3$  must be okay too**

# Sample Assembly (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }

```
for2tst:  addi    $s1, $s0, -1
          slti    $t0, $s1, 0
          bne     $t0, $zero, exit2
          sll     $t1, $s1, 2
          add     $t2, $a0, $t1
          lw      $t3, 0($t2)
          lw      $t4, 4($t2)
          slt     $t0, $t4, $t3
          beq     $t0, $zero, exit2
          .....
          addi    $s1, $s1, -1
          j       for2tst
```

3 stalls

3 stalls

3 stalls

3 stalls

3 stalls

3 stalls

How many stalls?

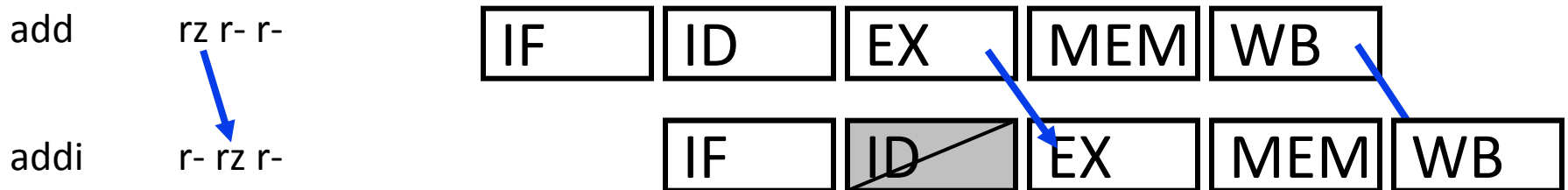
exit2:

# 通过Data Forwarding减少Stall

- Also called Data Bypassing
- Forward the value to the dependent instruction as soon as it is available
- Dataflow model
  - Data value supplied to dependent instruction as soon as it is available
  - Instruction executes when all its operands are available
- Data forwarding brings a pipeline closer to data flow execution principles

# Data Forwarding (Bypassing)

- 将RF看做状态(**state**)更加直观
  - “add rx ry rz” literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]
- 但是，RF仅仅是通信抽象的一部分
  - “add rx ry rz” means
    1. get the results of the last instructions to define the values of RF[ry] and RF[rz], respectively,
    2. until another instruction redefines RF[rx], younger instructions that refer to RF[rx] should use this instruction's result
- What matters is to maintain the correct “data flow” between operations, thus

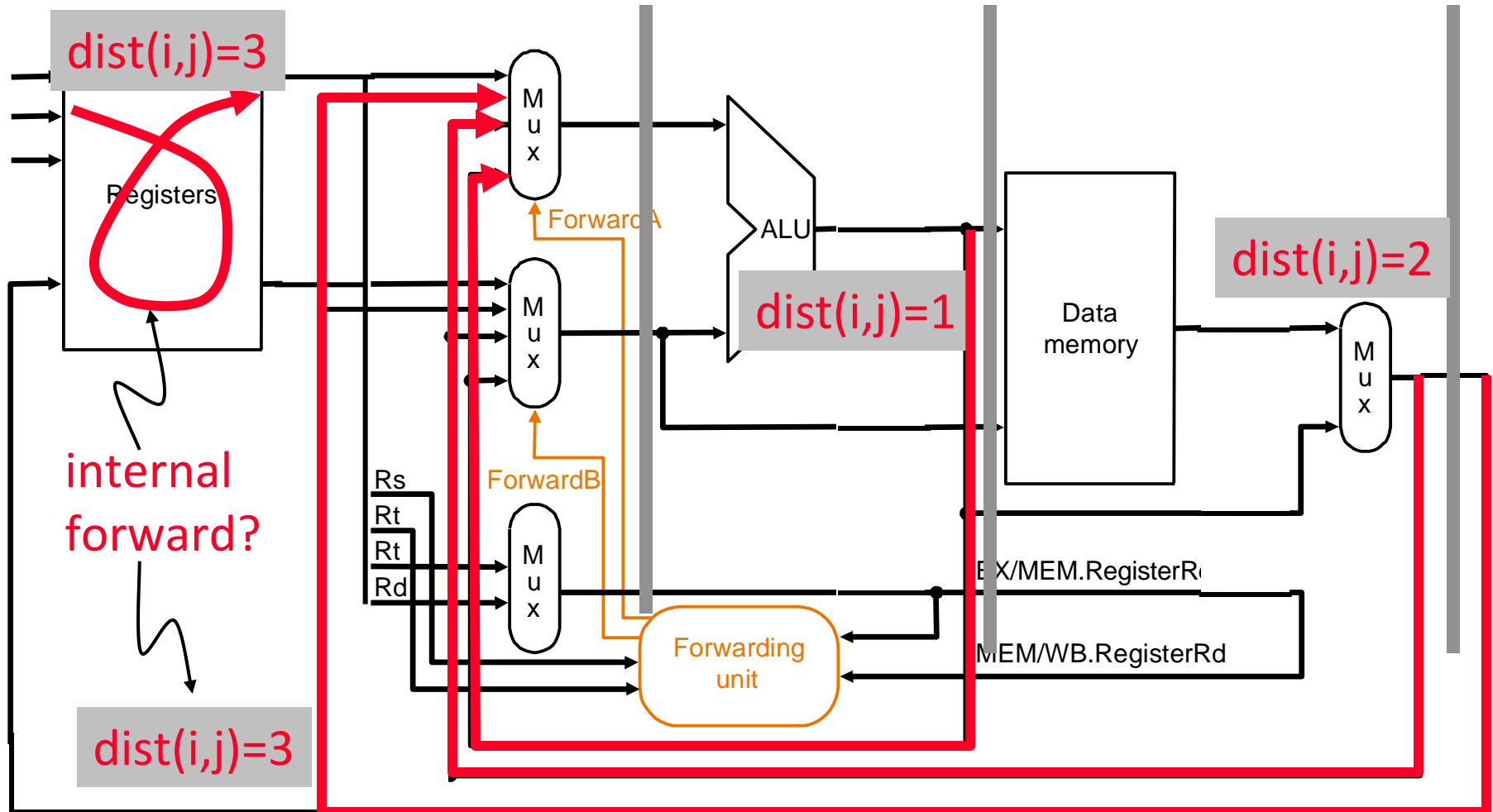


# 利用Forwarding解决RAW相关

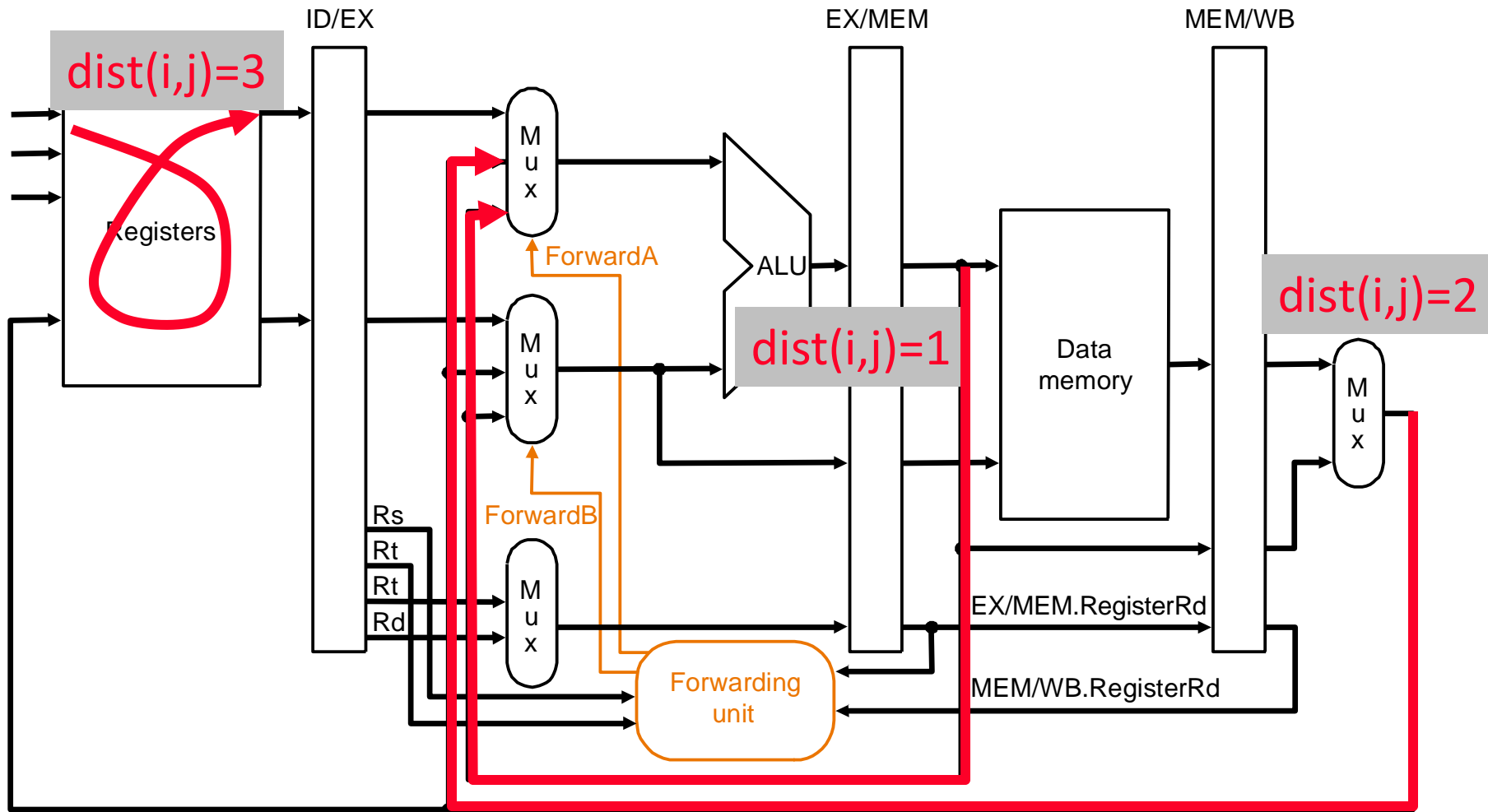
- Instructions  $I_A$  and  $I_B$  (where  $I_A$  comes before  $I_B$ ) have RAW dependence iff
  - $I_B$  (R/I, LW, SW, Br or JR) reads a register written by  $I_A$  (R/I or LW)
  - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$
- In other words, if  $I_B$  in ID stage reads a register written by  $I_A$  in EX, MEM or WB stage, then the operand required by  $I_B$  is not yet in RF
- How forwarding works?
  - $\Rightarrow$  retrieve operand from datapath instead of the RF
  - $\Rightarrow$  retrieve operand from the youngest definition if multiple definitions are outstanding



# Data Forwarding Paths (v1)



# Data Forwarding Paths (v2)



b. With forwarding

Assumes RF forwards internally

## Data Forwarding Logic (for v2)

```
if (rsEX!=0) && (rsEX==destMEM) && RegWriteMEM then
    forward operand from MEM stage // dist=1
else if (rsEX!=0) && (rsEX==destWB) && RegWriteWB then
    forward operand from WB stage // dist=2
else
    use operand from register file // dist >= 3
```

Ordering matters!! Must check youngest match first

Why doesn't use<sub>rs</sub>( ) appear in the forwarding logic?

# Data Forwarding (相关分析)

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

- Even with data-forwarding, RAW dependence on an immediately preceding LW instruction requires a stall

# Sample Assembly, No Forwarding (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }

```
for2tst:    addi    $s1, $s0, -1           3 stalls
            slti    $t0, $s1, 0           3 stalls
            bne     $t0, $zero, exit2
            sll     $t1, $s1, 2           3 stalls
            add     $t2, $a0, $t1         3 stalls
            lw      $t3, 0($t2)
            lw      $t4, 4($t2)           3 stalls
            slt     $t0, $t4, $t3         3 stalls
            beq     $t0, $zero, exit2
            .....
            addi    $s1, $s1, -1
            j       for2tst
exit2:
```

# Sample Assembly, Revisited (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }

```
                addi    $s1, $s0, -1
for2tst:        slti    $t0, $s1, 0
                bne     $t0, $zero, exit2
                sll     $t1, $s1, 2
                add     $t2, $a0, $t1
                lw      $t3, 0($t2)
                lw      $t4, 4($t2)
                nop
                slt     $t0, $t4, $t3
                beq     $t0, $zero, exit2
                .....
                addi    $s1, $s1, -1
                j        for2tst
exit2:
```

# 控制相关处理

## 回顾：控制相关

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?
- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction
- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?
- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?



# 分支类型

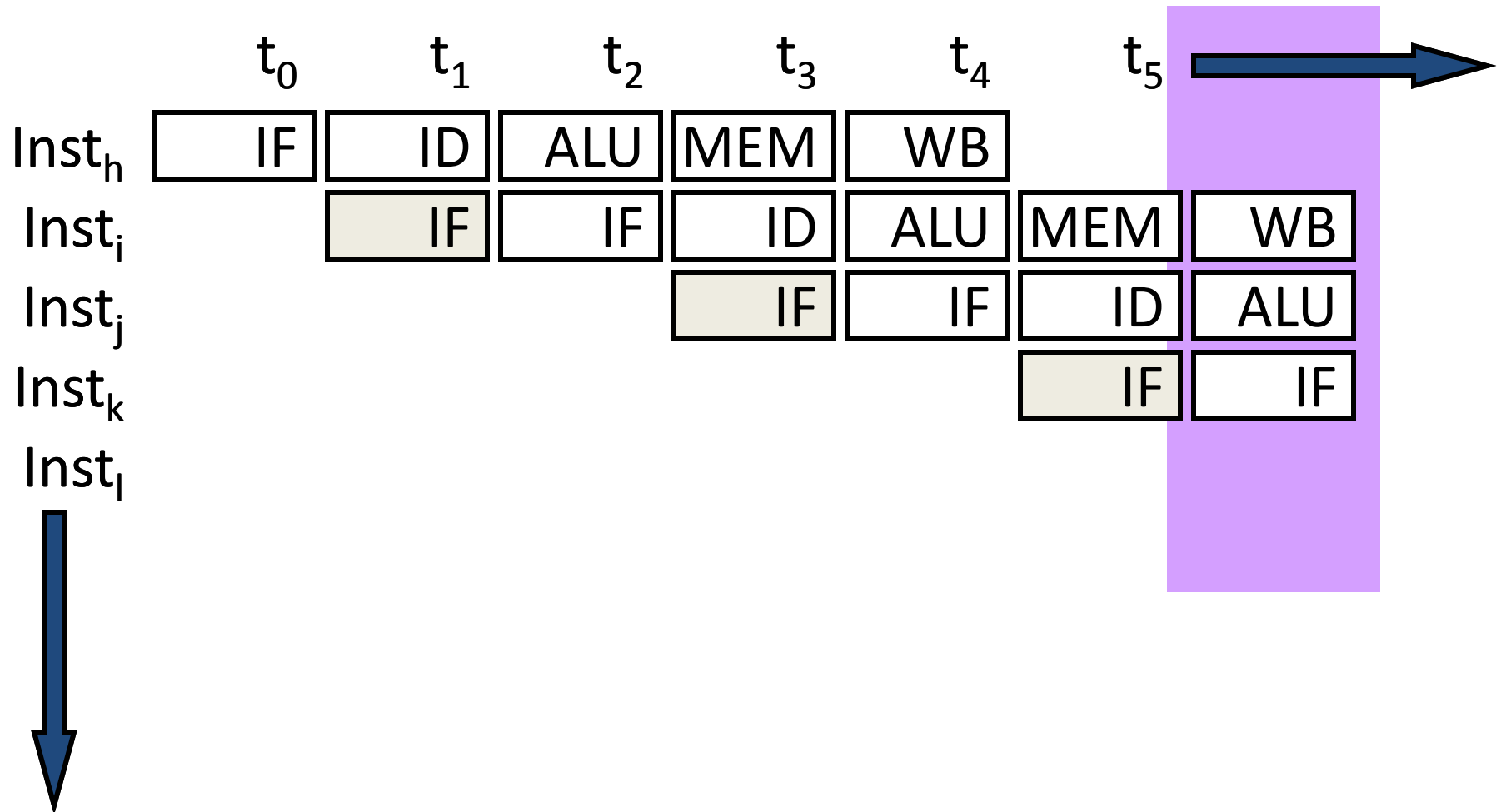
Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

# 如何处理控制相关

- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch delay slot)
  - .....

# Stall Fetch Until Next PC is Available: Good Idea?



This is the case with non-control-flow and unconditional br instructions!

# Doing Better than Stalling Fetch ...

- Rather than waiting for true-dependence on PC to resolve, just guess  $\text{nextPC} = \text{PC} + 4$  to keep fetching every cycle

Is this a good guess?

What do you lose if you guessed incorrectly?

- ~20% of the instruction mix is control flow
  - ~50 % of “forward” control flow (i.e., if-then-else) is taken
  - ~90% of “backward” control flow (i.e., loop back) is taken

Overall, typically ~70% taken and ~30% not taken

[Lee and Smith, 1984]

- Expect “ $\text{nextPC} = \text{PC} + 4$ ” ~86% of the time, but what about the remaining 14%?

# Guessing $\text{NextPC} = \text{PC} + 4$

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of **next fetch address prediction** (and branch prediction)
- How can you make this more effective?
- Idea: **Maximize the chances that the next sequential instruction is the next instruction to be executed**
  - Software: Lay out the control flow graph such that the “likely next instruction” is on the not-taken path of a branch
    - Profile guided code positioning → Pettis & Hansen, PLDI 1990.
  - Hardware: ??? (how can you do this in hardware...)
    - Cache traces of executed instructions → Trace cache

**Next Topic**

**Instruction Level Parallelism**