Data dependency handling

Seehwan Yoo

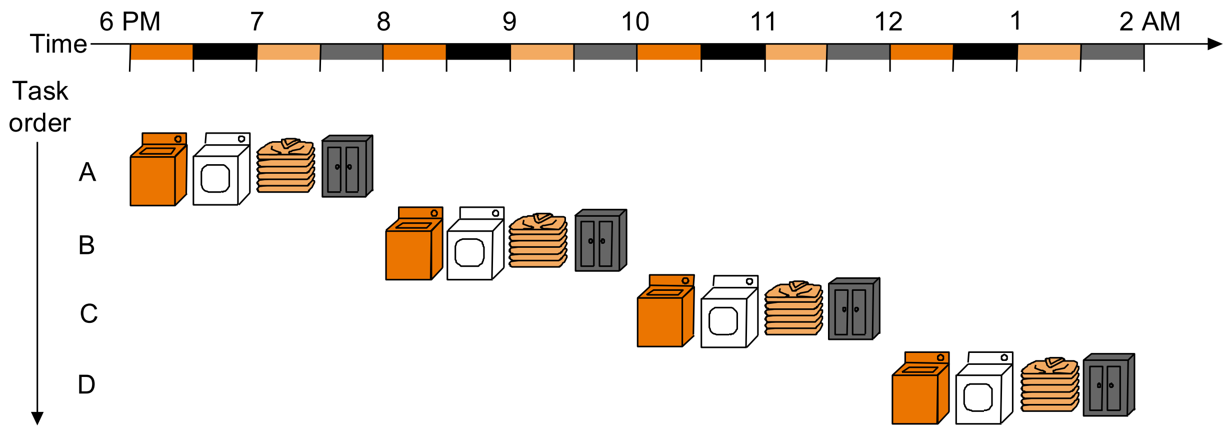
Dept. of Mobile Systems Engineering

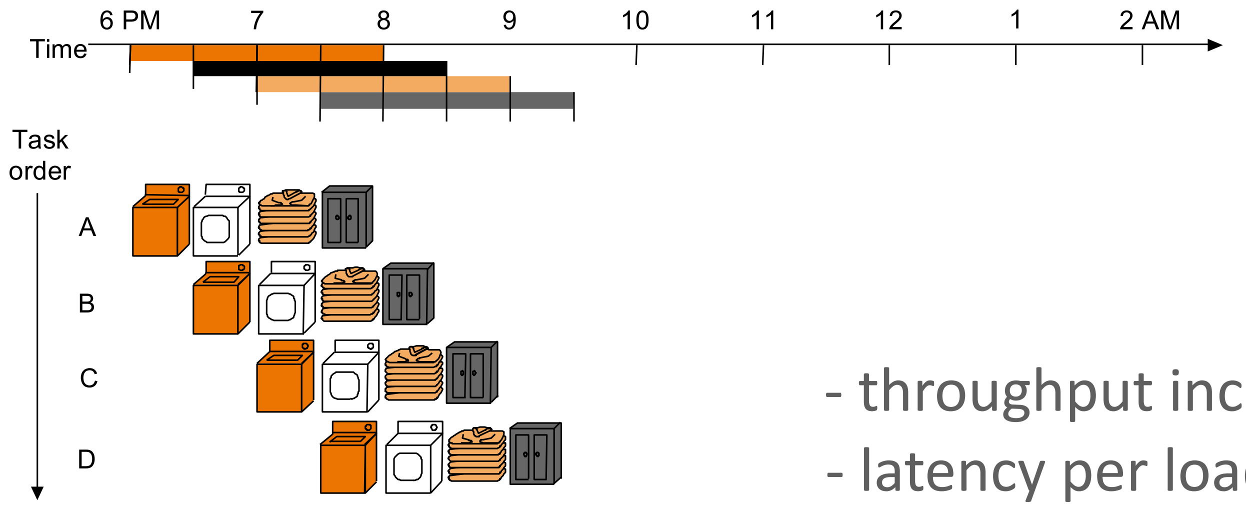
Dankook University

Before we move on…

- Launched programming assignment #3
  - we will have show time!
    - demos for some good works
  - Before show time, you may have hard time..
    - you have one month (actually more than a month)
    - start early, relax early – adjust your schedule
- Expectations from programming assignments
  - detailed understandings on course materials
    - I give you conceptual things, you take out all the details in it
  - improved programming skill
    - If you want to be a software programmer/engineer,
      you should at least be able to implement
      what you have in your mind, and
    - we are practicing some cases for that

Review - pipeline



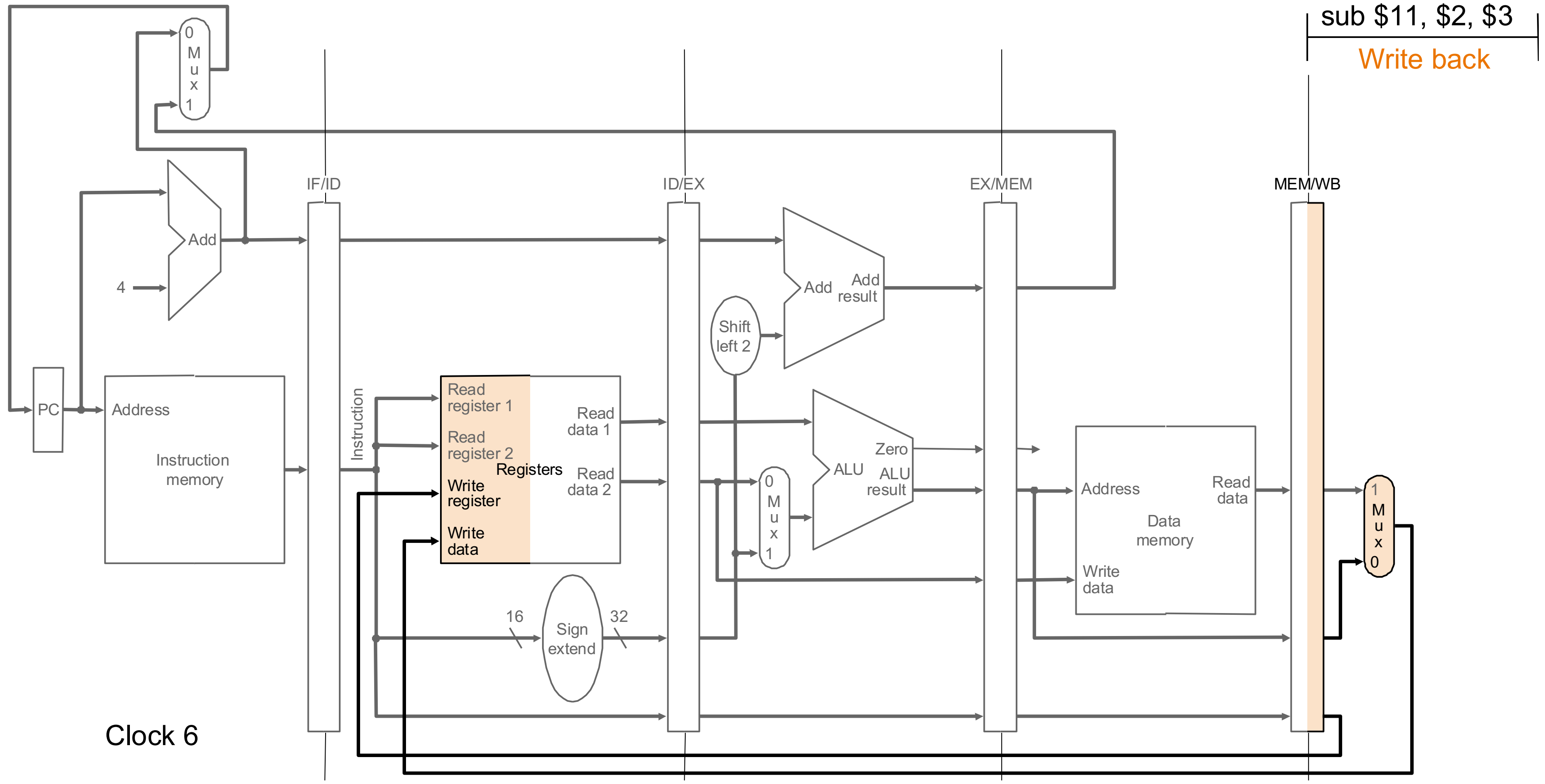


- throughput increased by 4
- latency per load is the same

Issues in pipelined U-arch

- ideal pipeline
  - same instruction execution
  - independent instructions execution
  - uniform execution time for each stage
- Reality
  - different instructions are coming
  - some instructions have dependency
  - non-uniform execution time
- Pipeline stall
  - pipeline is not fully filled in
  - reasons
    - resource contention
    - dependency
    - long latency operation

Review: Pipelined Operation Example



Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Data dependency

- What is ?
  - two instructions in the pipeline use the same register values
  - result should follow the original programming semantics
    - same result with single-cycle operation
  - Three 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

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

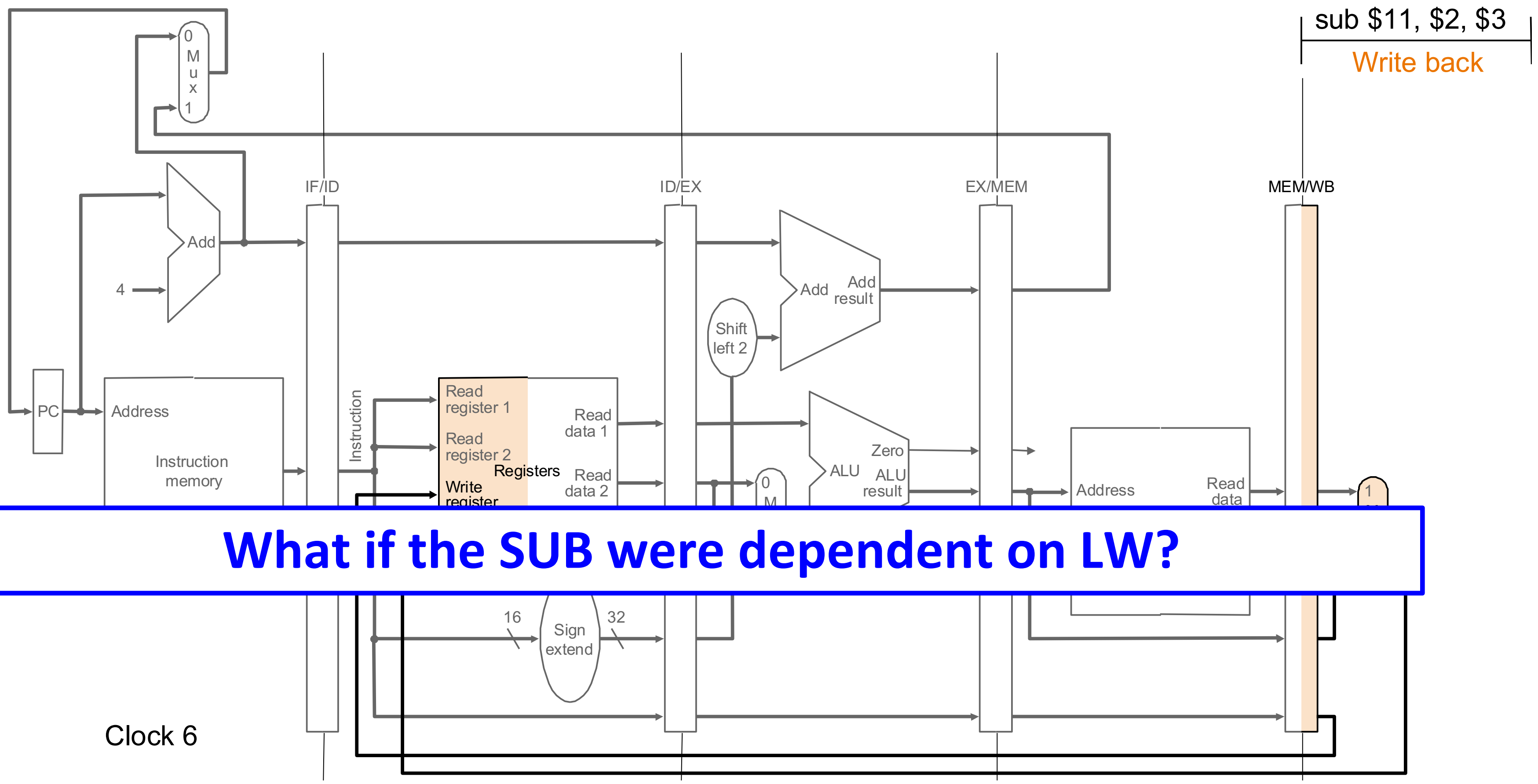
$r_5 \leftarrow r_3 \text{ op } r_4$

$r_3 \leftarrow r_6 \text{ op } r_7$

Write-after-Write (WAW)

DANKOOK UNIVERSITY

Pipelined Operation Example

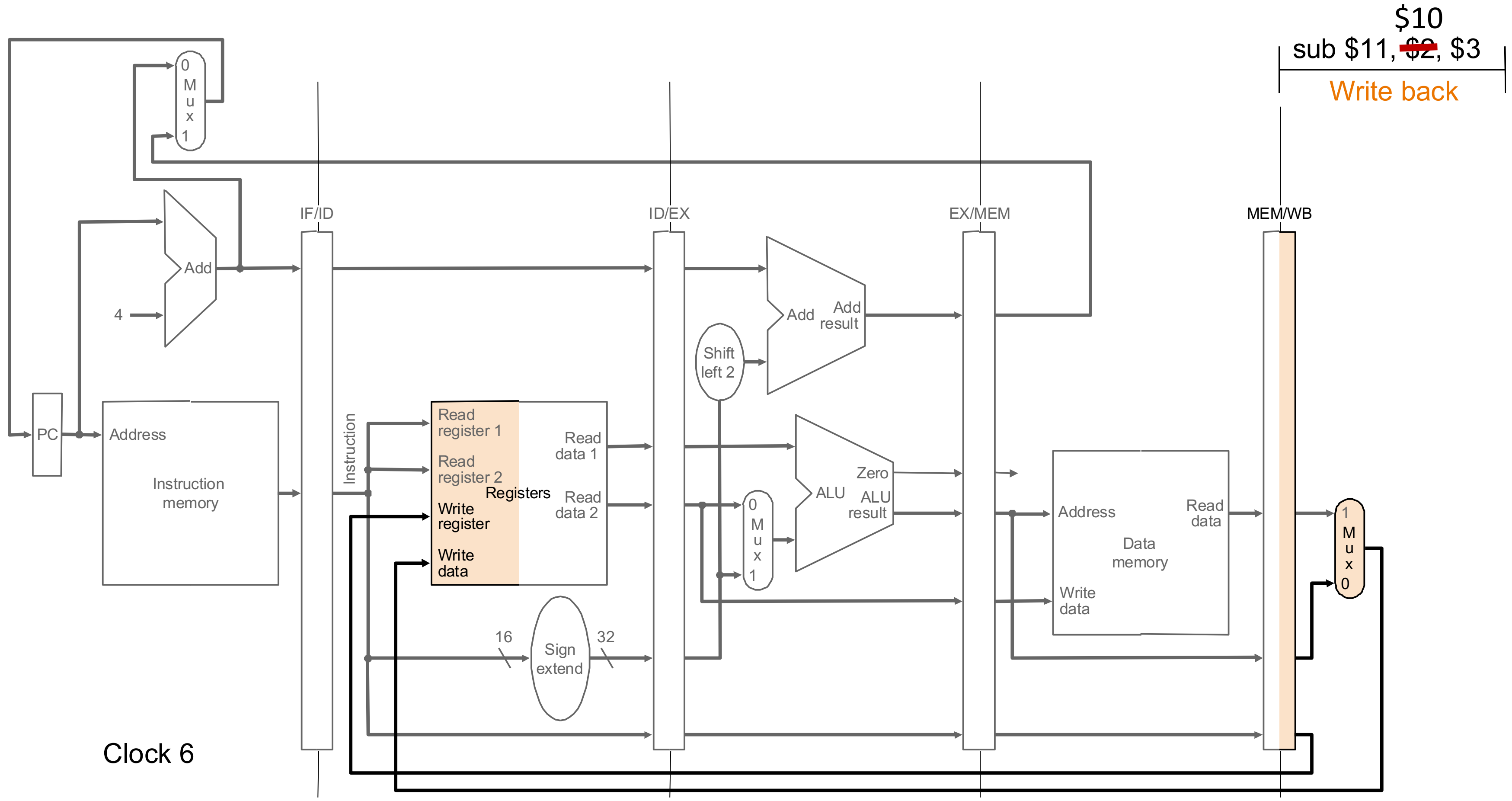


What if the SUB were dependent on LW?

Clock 6

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Pipelined Operation Example



Clock 6

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order

- Flow dependences are more interesting

- Five fundamental 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
  - Do something else (fine-grained multithreading)
    - No need to detect

Interlocking

- Detection of dependence between instructions in a pipelined processor to guarantee correct execution

- Software based interlocking

  vs.

- Hardware based interlocking

- MIPS acronym?

Approaches to Dependence Detection (I)

- Scoreboarding
  - Each register in register file has a Valid bit associated with it
  - An instruction that is writing to the register resets the Valid bit
  - An instruction in Decode stage checks if all its source and destination registers are Valid
    - Yes: No need to stall… No dependence
    - No: Stall the instruction

- Advantage:
  - Simple. 1 bit per register

- Disadvantage:
  - Need to stall for all types of dependences, not only flow dep.

- Not Stalling on Anti and Output Dependences
    - What changes would you make to the scoreboard to enable this?

Approaches to Dependence Detection (II)

- Combinational dependence check logic
  - Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
  - Yes: stall the instruction/pipeline
  - No: no need to stall… no flow dependence

- Advantage:
  - No need to stall on anti and output dependences

- Disadvantage:
  - Logic is more complex than a scoreboard
  - Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)

Once You Detect the Dependence in Hardware

- What do you do afterwards?

- Observation: Dependence between two instructions is detected before the communicated data value becomes available

- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: …

Data Forwarding/Bypassing

- Problem: A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file

- Goal: We do not want to stall the pipeline unnecessarily

- Observation: The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file)

- Idea: Add additional dependence check logic and data forwarding paths (buses) to supply the producer's value to the consumer right after the value is available

- Benefit: Consumer can move in the pipeline until the point the value can be supplied → less stalling

A Special Case of Data Dependence

- Control dependence
    - Data dependence on the Instruction Pointer / Program Counter

Control Dependence

- 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 know whether or not the fetched instruction is a control-flow instruction?

Data Dependence Handling: *More Depth & Implementation*

How to Handle Data Dependences

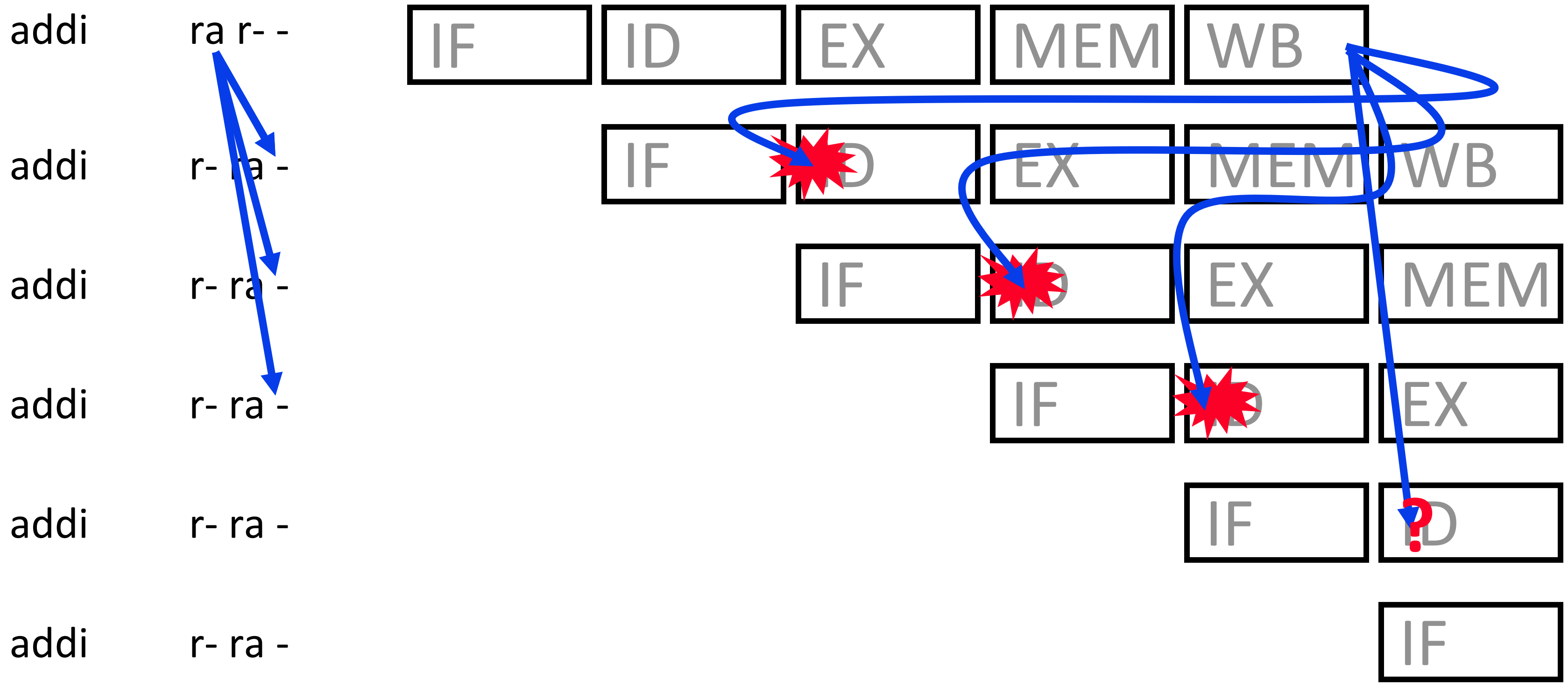
- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order

- Flow dependences are more interesting

- Five fundamental 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
  - Do something else (fine-grained multithreading)
    - No need to detect

RAW Dependence Handling

- Following flow dependences lead to conflicts in the 5-stage pipeline

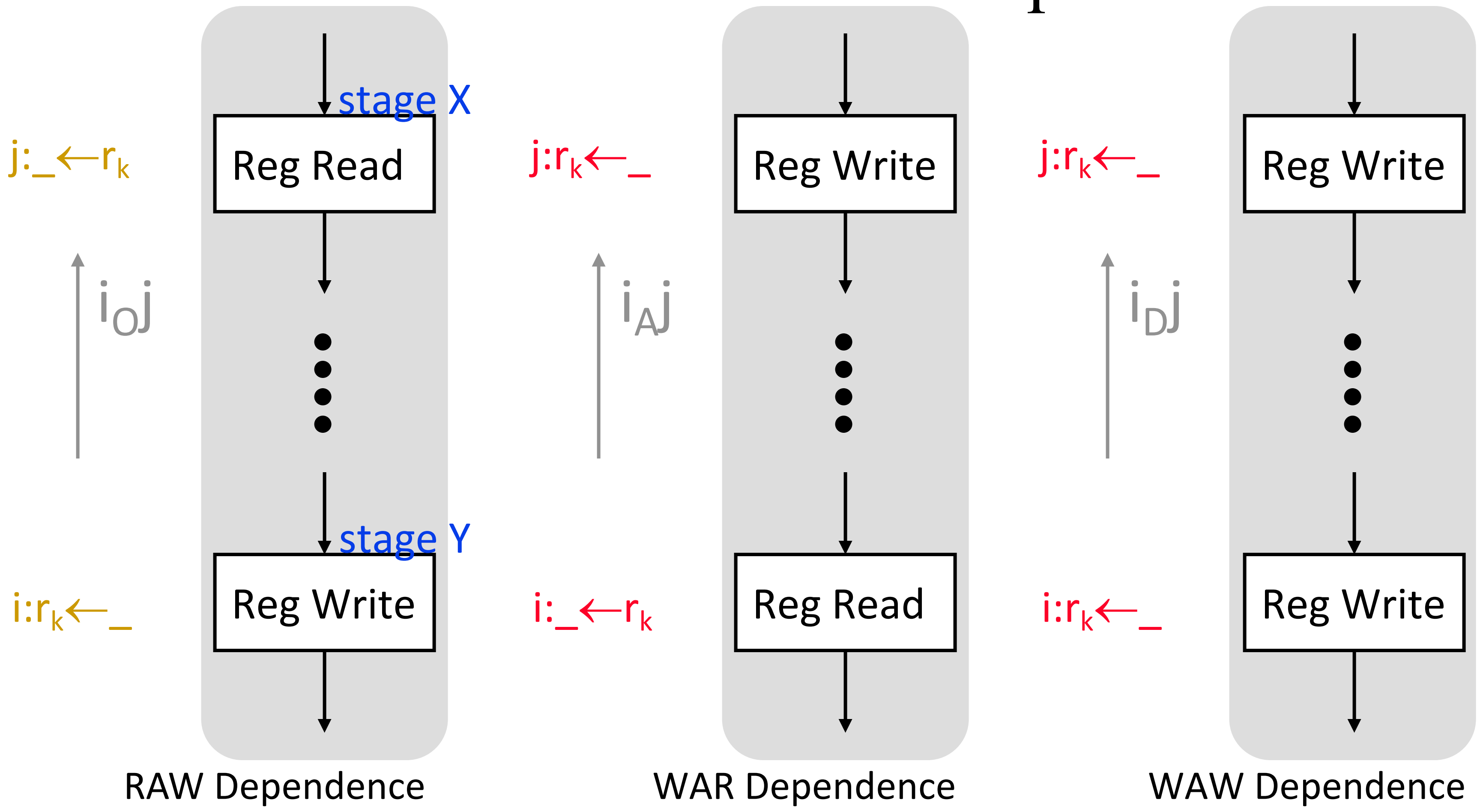


Register Data Dependence Analysis

	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 2 data dependent instructions?
    - dependence type: RAW, WAR, WAW?
    - instruction types involved?
    - distance between the two instructions?

Safe and Unsafe Movement of Pipeline



dist(i,j) $\leq$ dist(X,Y) $\Rightarrow$ Unsafe to keep j moving

dist(i,j) $>$ dist(X,Y) $\Rightarrow$ Safe

RAW Dependence Analysis Example

	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				

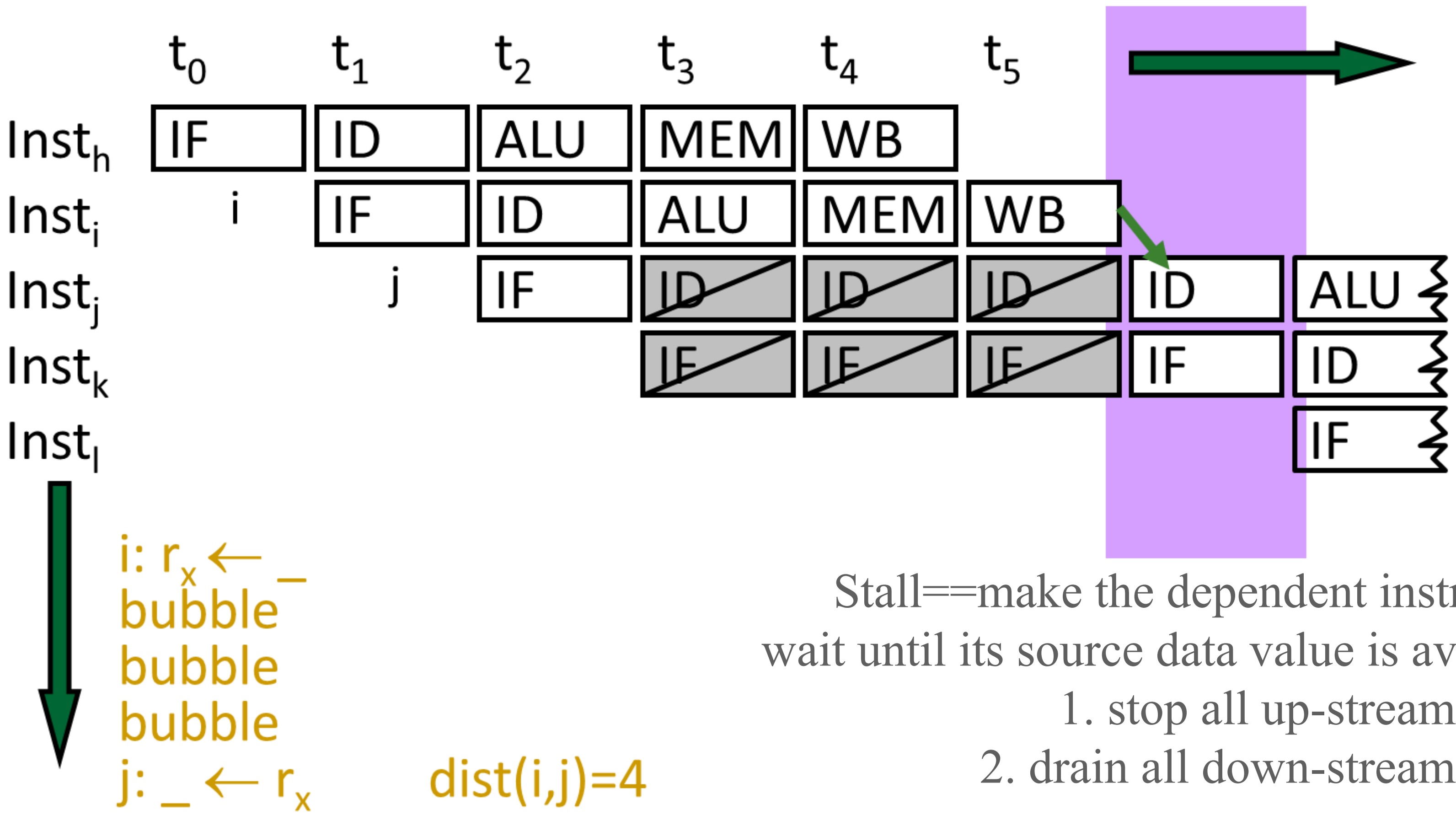
- Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $dist(I_A, I_B) <= dist(ID, WB) = 3$

What about WAW and WAR dependence?

What about memory data dependence?

DANKOOK UNIVERSITY

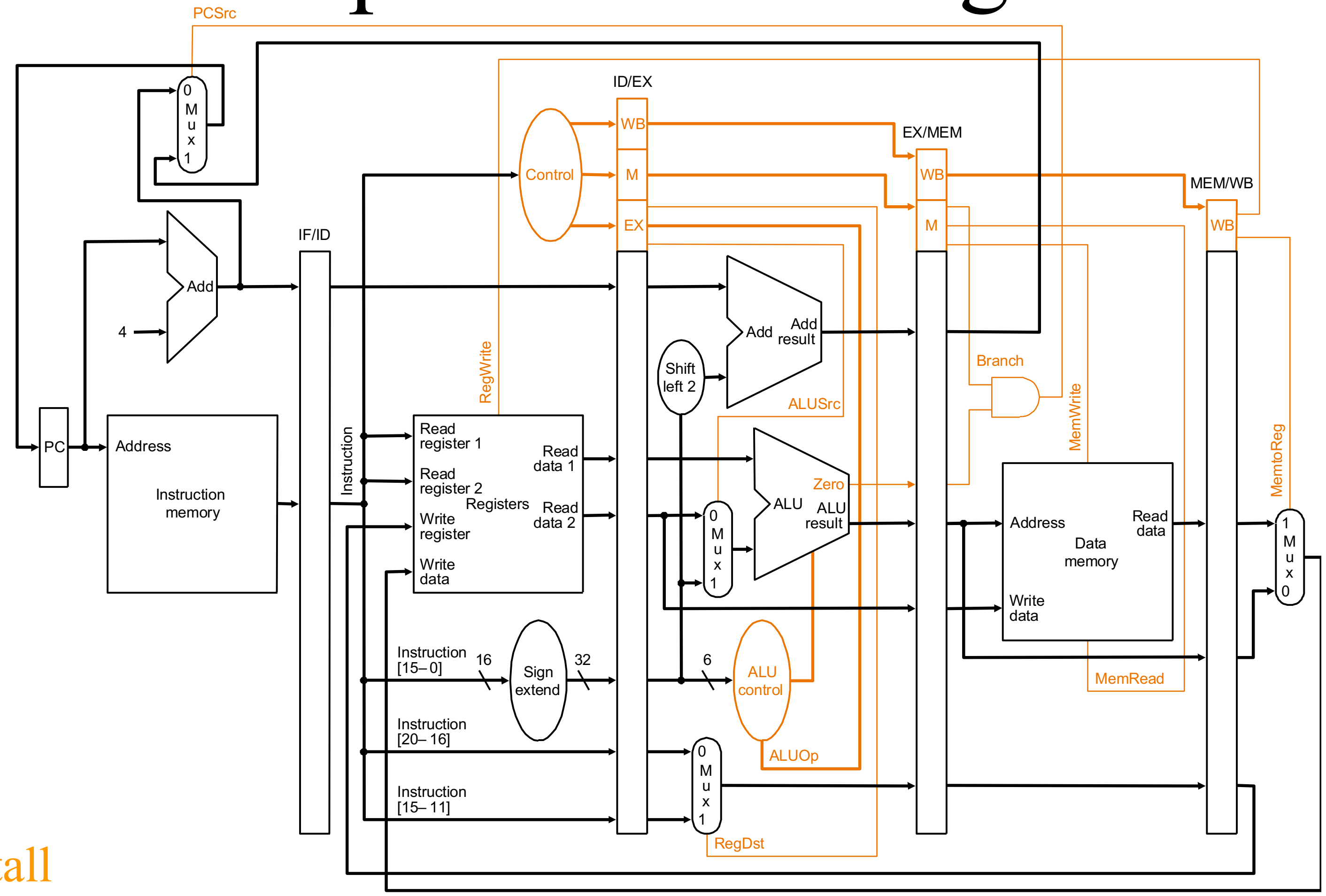
Pipeline Stall: Resolving Data Dependence



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

How to Implement Stalling



- Stall
  - disable **PC** and **IR** latching; ensure stalled instruction stays in its stage
  - Insert "invalid" instructions/nops into the stage following the stalled one

DANKOOK UNIVERSITY

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)
  - $dist(I_A, I_B) \leq dist(ID, WB) = 3$

- In other words, must stall when I_B in ID stage wants to read a register to be written by I_A in EX, MEM or WB stage

Stall Conditions

- Helper functions
  - rs(I) returns the rs field of I
  - use_rs(I) returns true if I requires **RF**[rs] and rs!=r0
- Stall when
  - (rs(**IR$_{ID}$**)==dest$_{EX}$) && use_rs(**IR$_{ID}$**) && RegWrite$_{EX}$ or
  - (rs(**IR$_{ID}$**)==dest$_{MEM}$) && use_rs(**IR$_{ID}$**) && RegWrite$_{MEM}$ or
  - (rs(**IR$_{ID}$**)==dest$_{WB}$) && use_rs(**IR$_{ID}$**) && RegWrite$_{WB}$ or
  - (rt(**IR$_{ID}$**)==dest$_{EX}$) && use_rt(**IR$_{ID}$**) && RegWrite$_{EX}$ or
  - (rt(**IR$_{ID}$**)==dest$_{MEM}$) && use_rt(**IR$_{ID}$**) && RegWrite$_{MEM}$ or
  - (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

Impact of Stall on Performance

- Each stall cycle corresponds to 1 lost ALU cycle
- 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

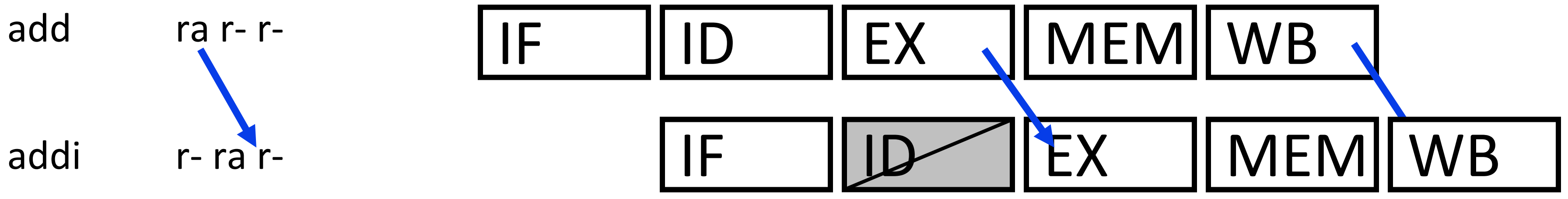
Sample Assembly (P&H)

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

```
          addi   $s1, $s0, -1            3 stalls
for2tst:  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:
```

Data Forwarding (or Data Bypassing)

- It is intuitive to think of RF as state
  - "add rx ry rz" literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]
- But, RF is just a part of a computing abstraction
  - "add rx ry rz" means 1. get the results of the last 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] should use this instruction's result
- What matters is to maintain the correct "dataflow" between operations, th us



Resolving RAW Dependence with Forwarding

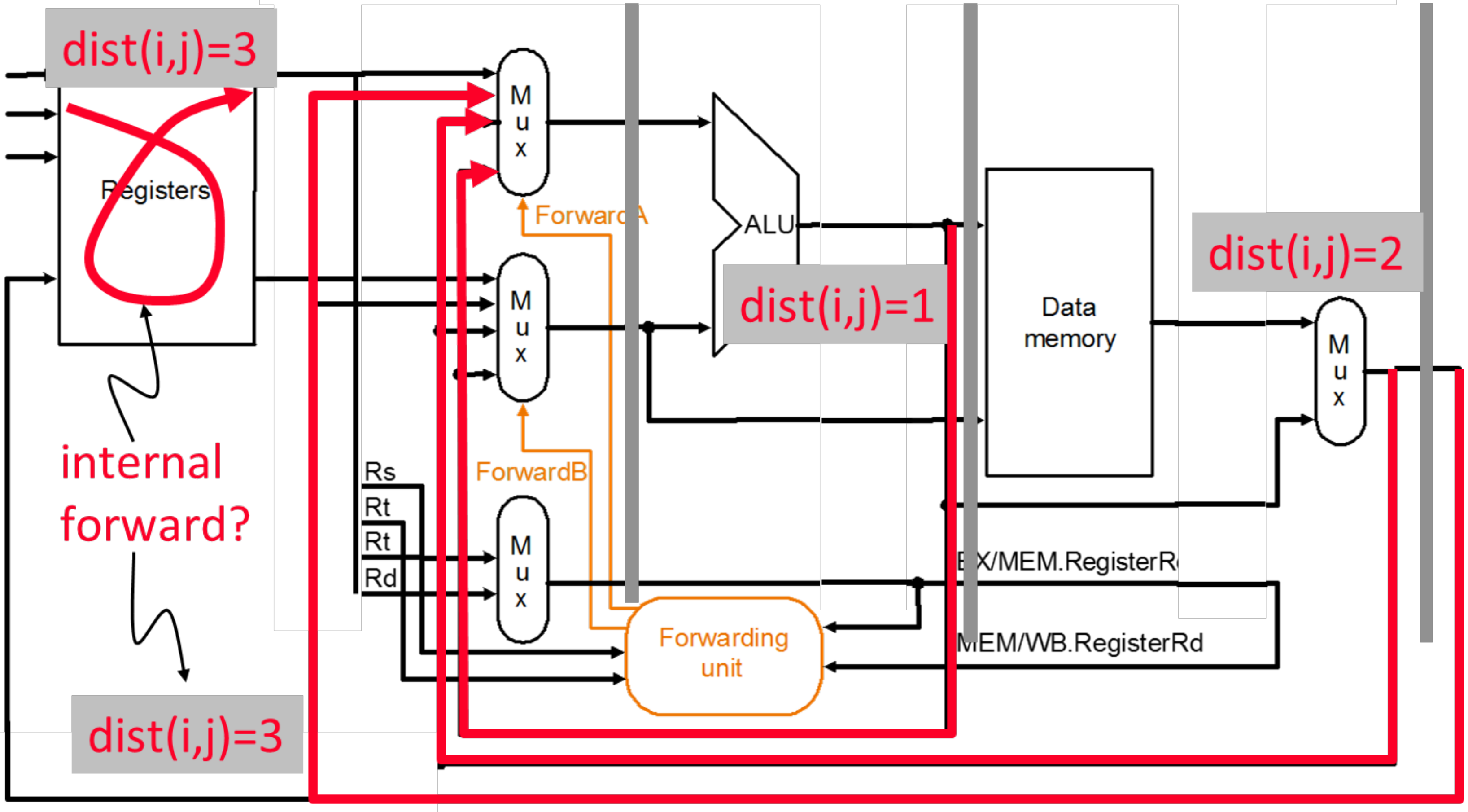
- Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $\text{dist}(I_A, I_B) \le \text{dist}(ID, WB) = 3$

- 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

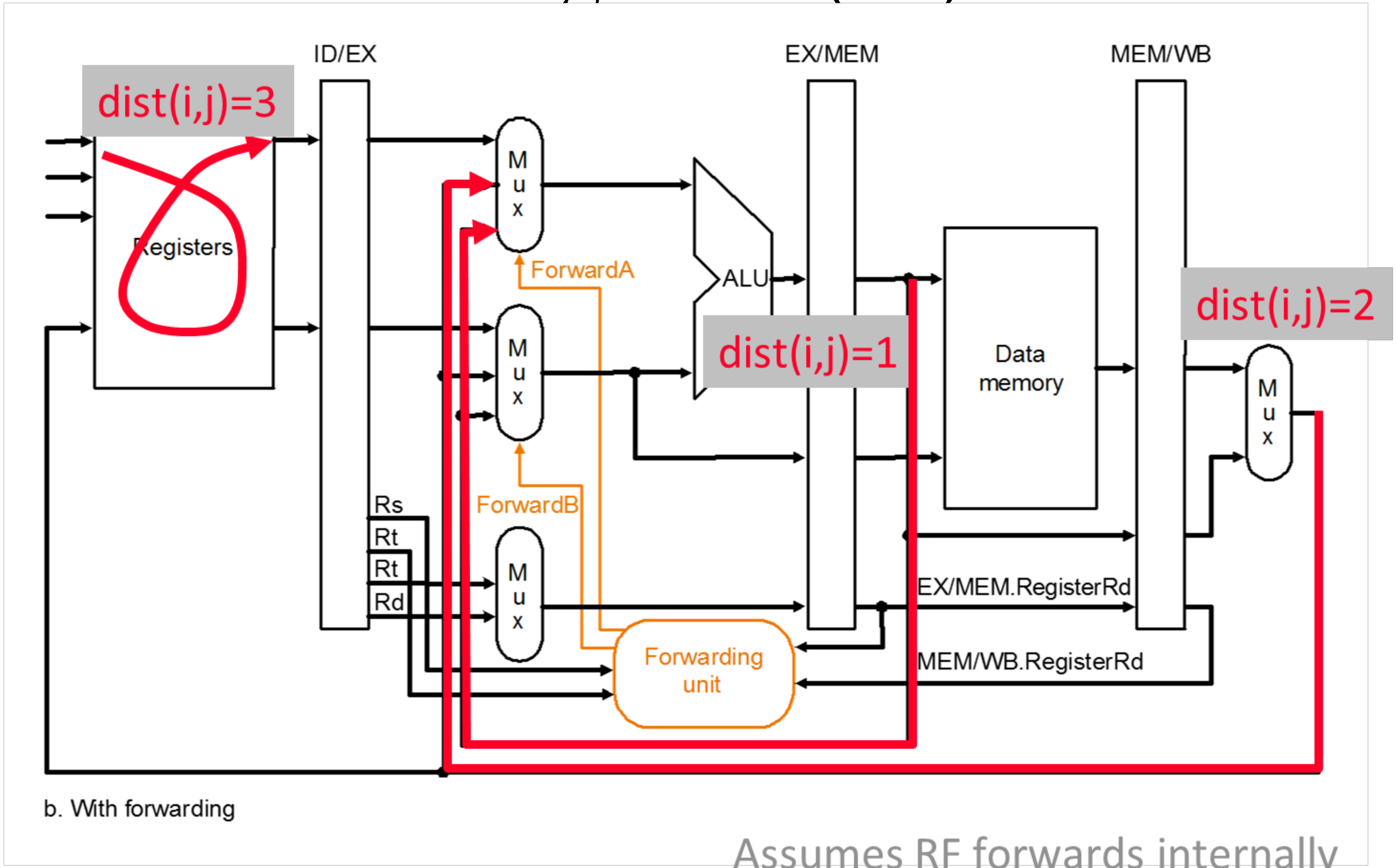
  $\Rightarrow$ retrieve operand from datapath instead of the RF

  $\Rightarrow$ retrieve operand from the youngest definition if multiple definitions are outstanding

Data Forwarding Paths (v1)



Data Forwarding Paths (v2)



Assumes RF forwards internally

Data Forwarding Logic (for v2)

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

Ordering matters!! Must check youngest match first

Why doesn't use_rs( ) appear in the forwarding logic?

What does the above not take into account?

Data Forwarding (Dependence Analysis)

	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 (delayed load slot)

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:
```

Register Forwarding Summary

- Reduce stall cycles
  - Do not wait until dependent instruction completes the write back stage
- Fetch ALU input data from later stages
  - in the EX stage, check whether ALU input has been produced already
  - get the required value from later stages (WB, MEM)
- Two further considerations
  - Continuous forwarding
    - forwarded result could be forwarded again
    - find the youngest value
  - Load RAW dependency
    - need at least one stall (cannot produce writeback result after ex stage)
    - MIPS inserts nop after 'load word' instruction

DANKOOK UNIVERSITY

Appendix: Score boarding

- Scoreboard for register access
- Simple score board

Reg #	data	valid
0	0	1
1	aaa	1
2	bbb	1
…	…	1
31	XXX	1

2) when decoding, disable writeback register

Reg #	data	valid
0	0	1
1	aaa	1
2	bbb	~~1~~ → 0
…	…	1
31	XXX	1

3) enable back after write back

Reg #	data	valid
0	0	1
1	aaa	1
2	bbb	~~1~~ → ~~0~~ → 1
…	…	1
31	XXX	1

- valid bit check when decoding stage
  - disable the target register until the corresponding instruction completes the writeback stage
  - e.g.) add r2, r3, r2: disable r2 until it is written back
- re-enable after written back

Simple score boarding does not work!

- A counter example case
  - inst1: add r2, r3, r2 (r2 is disabled for the next 3 cycles)
  - inst2: add r2, r3, r1 (does not stall, no flow dependence)
  - inst3: add r7, r3, r4 (does not stall, no flow dependence)
  - inst4: add r8, r3, r1 (does not stall, no flow dependence)
    - ---After this, r2 is re-enabled!
  - inst5: add r2, r3, r2
    - inst5 will get value from inst1, which is wrong value
    - inst5's r2 is dependent upon inst2, not inst1
- single bit counter cannot detect output dependency!

Check the youngest register value

- Find correct flow dependent instruction
  - remember the corresponding instruction that produces result
  - inst1: add r2, r3, r2 (r2 is disabled for the next 3 cycles)
  - inst2: add r2, r3, r1
  - inst3: add r7, r3, r4
  - inst4: add r8, r3, r1
  - inst5: add r2, r3, (~~r2~~ → inst2's r2)
  - inst6: add r2, r4, (~~r2~~ → inst5's r2)
  - inst7: add r4, (~~r2~~ → inst6's r2), r3

Proper way of handling dependency using score boarding

- score boarding with instruction sequence tag
- sequence tag: put a tag for every fetched instruction
  - tag value: sequence number of fetched instruction

Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb	~~1~~→0	1
…	…	1	
31	XXX	1	

meaning: if you want to read r2, wait until inst1 writes back

- When decoding inst1,
- inst1: add r2, r3, r2 (r2 is disabled for the next 3 cycles)

DANKOOK UNIVERSITY

Proper way of handling dependency using score boarding

- score boarding with instruction sequence tag
- sequence tag: put a tag for every fetched instruction
  - tag value: sequence number of fetched instruction

Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb	~~1~~$\rightarrow$0	~~1~~$\rightarrow$2
…	…	1	
31	XXX	1	

meaning: if you want to read r2, use inst2's r2

- When decoding inst2,
- inst2: inst2: add r2, r3, r1

Proper way of handling dependency using score boarding

- score boarding with instruction sequence tag
- sequence tag: put a tag for every fetched instruction
  - tag value: sequence number of fetched instruction

Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb’	~~1~~→0	~~1~~→2
…	…	1	
31	XXX	1	

meaning: if you want to read r2, use inst2’s r2

- At cycle 5,
  - write back r2 from inst1 (valid unchanged, tag unchanged)
  - Note that inst2 has not write back, yet!

DANKOOK UNIVERSITY

Proper way of handling dependency using score boarding

- score boarding with instruction sequence tag
- sequence tag: put a tag for every fetched instruction
    - tag value: sequence number of fetched instruction

Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb''	~~1~~→~~0~~→1	~~1~~→2
…	…	1	
31	XXX	1	

meaning: if you want to read r2, use inst2's r2

- At cycle 6,
    - write back r2 from inst2 (valid changed, tag unchanged)
    - bbb''

Proper way of handling dependency using score boarding

- score boarding with instruction sequence tag
- sequence tag: put a tag for every fetched instruction
  - tag value: sequence number of fetched instruction

Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb’’	~~1~~→~~0~~→1	~~1~~→2
…	…	1	
31	XXX	1	

meaning: if you want to read r2, use inst2’s r2

- Decoding inst5,
  - inst5: add r2, r3, r2 (use inst2’s r2 because tag is 2)
  - read r2 (bbb”) from inst2, just updated!

DANKOOK UNIVERSITY

Proper way of handling dependency using score boarding

- score boarding with instruction sequence tag
- sequence tag: put a tag for every fetched instruction
  - tag value: sequence number of fetched instruction

Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb''	…~~1~~→0	~~1~~→~~2~~→5
…	…	1	
31	XXX	1	

meaning: if you want to read r2, use inst5's r2

- Decoding inst5,
  - inst5: add r2, r3, r2 (use inst2's r2 because tag is 2)
  - update score board

DANKOOK UNIVERSITY

Proper way of handling dependency using score boarding

- score boarding with instruction sequence tag
- sequence tag: put a tag for every fetched instruction
  - tag value: sequence number of fetched instruction

Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb''	…~~1~~→0	~~1~~→~~2~~→5
…	…	1	
31	XXX	1	

meaning: if you want to read r2, use inst5's r2

- Decoding inst6,
  - inst6: add r2, r4, r2 (use inst5's r2 because tag is 5)
  - wait(stall) until inst5 writes back result to r2

DANKOOK UNIVERSITY

Proper way of handling dependency using score boarding

- score boarding with instruction sequence tag
- sequence tag: put a tag for every fetched instruction
  - tag value: sequence number of fetched instruction

Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb’’	…~~1~~→0	…5→6
…	…	1	
31	XXX	1	

meaning: if you want to read r2, use inst6’s r2

- Decoding inst6,
  - inst6: add r2, r4, r2 (use inst5’s r2 because tag is 5)
  - update score board

DANKOOK UNIVERSITY

Scoreboarding summary

- Decoding stage
  - update score board (for output dependence)
  - read register value from correct instruction (not stage)
- Write back stage
  - update register that is visible to user
- Stalling instruction
  - has to remember the instruction, which is waiting for
- In fact, there is reservation station (RS)
  - stalling instructions are waiting in RS
  - waiting for meeting the dependency conditions
  - we will be back in some time