

18-447

Computer Architecture

Lecture 9: Data Dependence Handling

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2013, 2/6/2013

Reminder: Homework 2

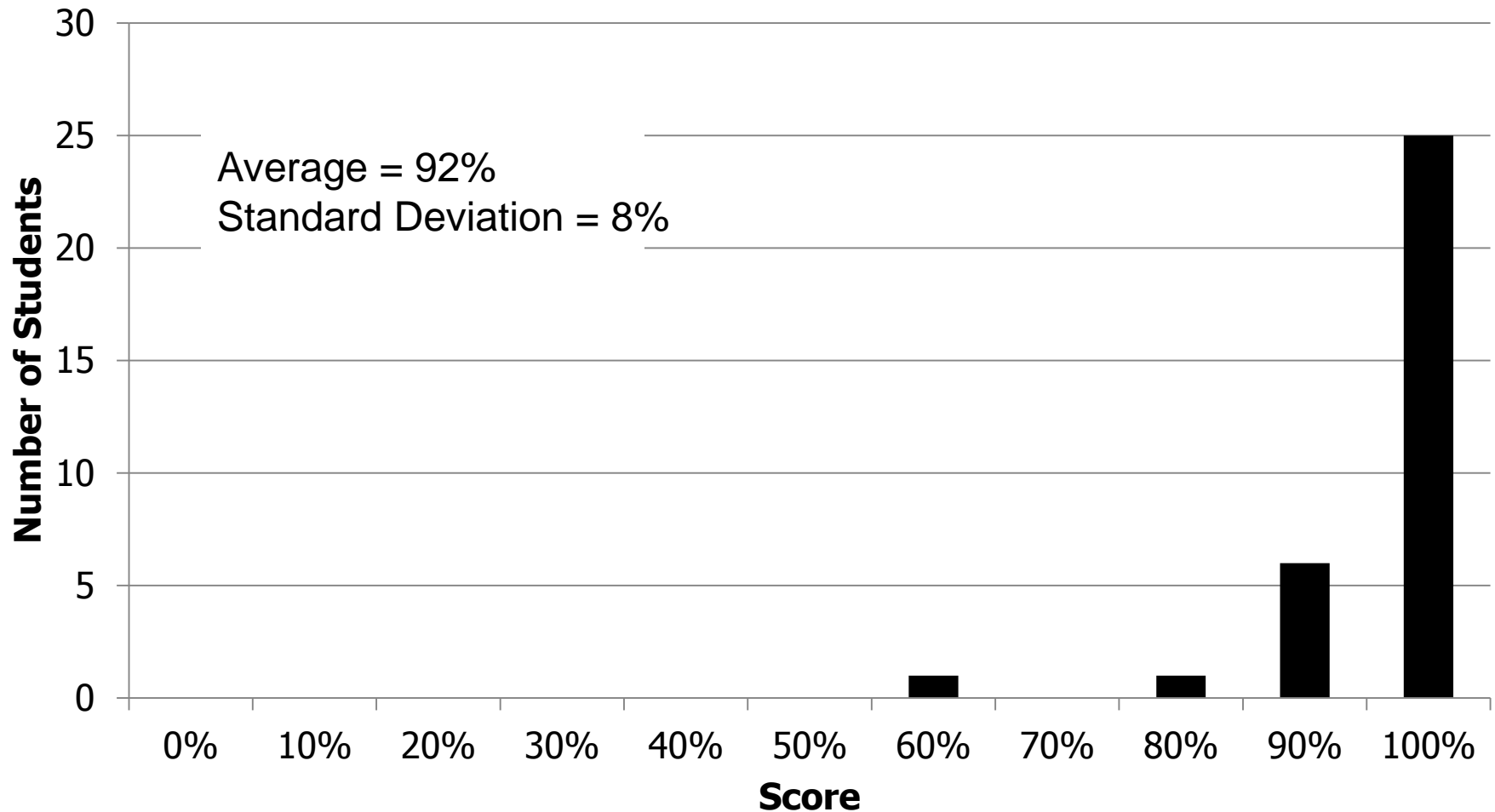
- Homework 2 out
 - Due February 11 (next Monday)
 - LC-3b microcode
 - ISA concepts, ISA vs. microarchitecture, microcoded machines
- Remember: Homework 1 solutions were out

Reminder: Lab Assignment 2

- Lab Assignment 1.5
 - Verilog practice
 - Not to be turned in
- Lab Assignment 2
 - Due Feb 15
 - Single-cycle MIPS implementation in Verilog
 - All labs are individual assignments
 - No collaboration; please respect the honor code
 - Do not forget the extra credit portion!

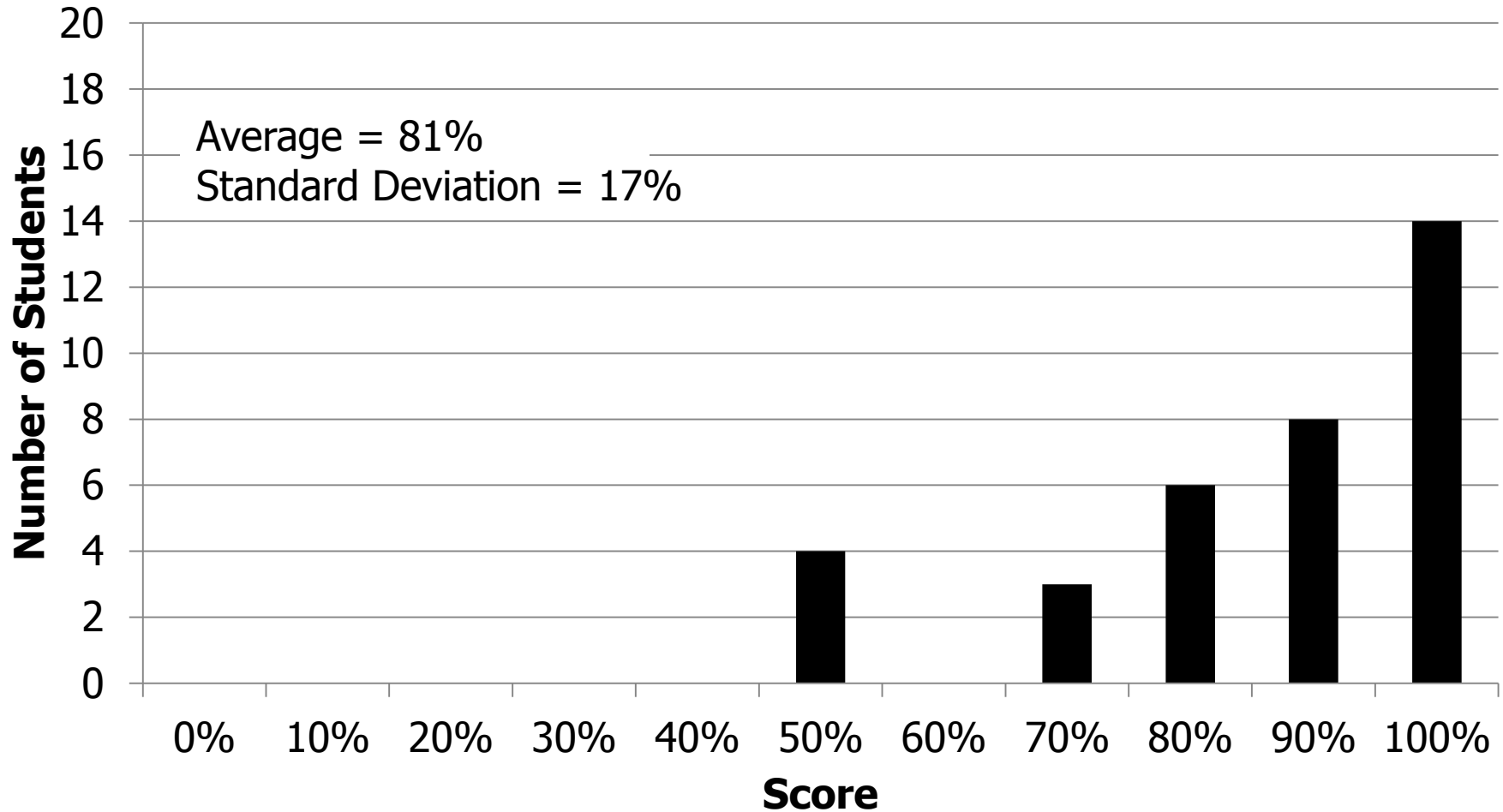
Homework 1 Grades

HW 1 Score Distribution



Lab 1 Grades

Lab 1 Score Distribution



Readings for Next Few Lectures

- P&H Chapter 4.9-4.11
- Smith and Sohi, “[The Microarchitecture of Superscalar Processors](#),” Proceedings of the IEEE, 1995
 - ❑ More advanced pipelining
 - ❑ Interrupt and exception handling
 - ❑ Out-of-order and superscalar execution concepts

Today's Agenda

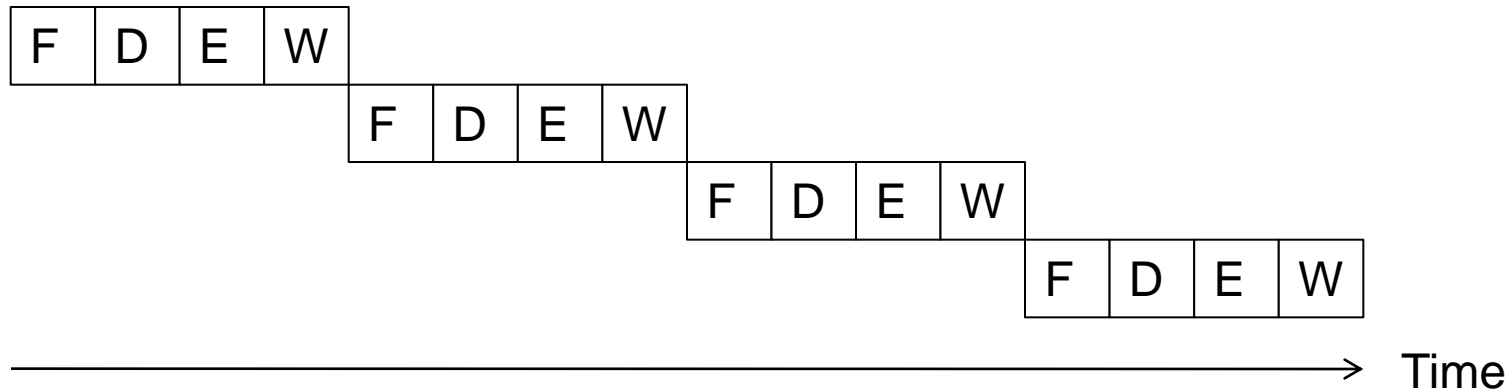
- Deep dive into pipelining
 - Dependence handling

Review: Pipelining: Basic Idea

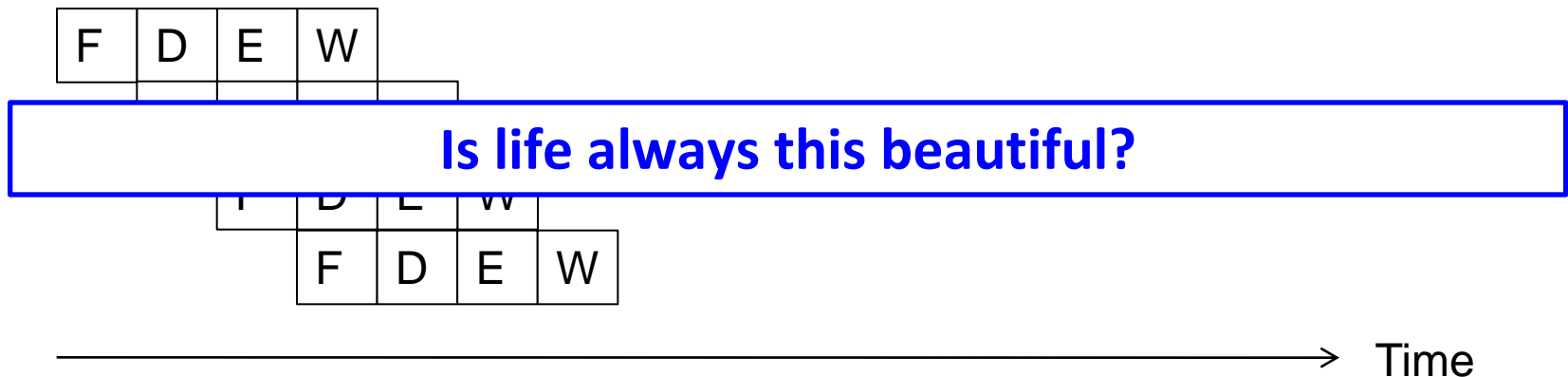
- Idea:
 - Divide the instruction processing cycle into distinct “stages” of processing
 - Ensure there are enough hardware resources to process one instruction in each stage
 - Process a different instruction in each stage
 - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: ???

Review: Execution of Four Independent ADDs

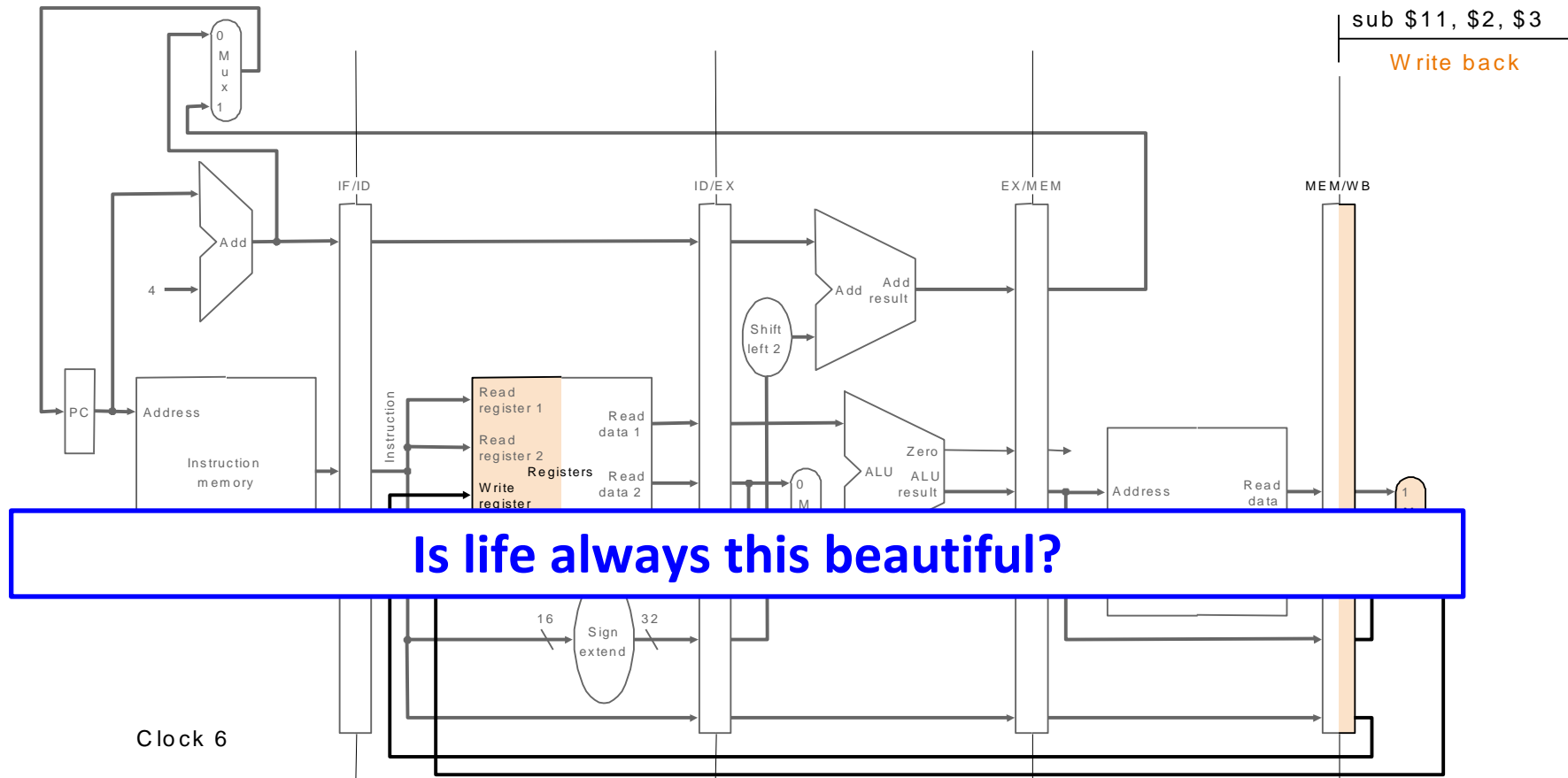
- Multi-cycle: 4 cycles per instruction



- Pipelined: 4 cycles per 4 instructions (steady state)



Review: Pipelined Operation Example



Review: Instruction Pipeline: Not An Ideal Pipeline

■ Identical operations ... NOT!

⇒ different instructions do not need all stages

- Forcing different instructions to go through the same multi-function pipe
- external fragmentation (some pipe stages idle for some instructions)

■ Uniform suboperations ... NOT!

⇒ difficult to balance the different pipeline stages

- Not all pipeline stages do the same amount of work
- internal fragmentation (some pipe stages are too-fast but take the same clock cycle time)

■ Independent operations ... NOT!

⇒ instructions are not independent of each other

- Need to detect and resolve inter-instruction dependencies to ensure the pipeline operates correctly
- Pipeline is not always moving (it stalls)

Review: Fundamental Issues in Pipeline Design

- Balancing work in pipeline stages
 - How many stages and what is done in each stage
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
 - Handling dependences
 - Data
 - Control
 - Handling resource contention
 - Handling long-latency (multi-cycle) operations
- Handling exceptions, interrupts
- Advanced: Improving pipeline throughput
 - Minimizing stalls


Review: Data Dependences

- Types of data dependences
 - Flow dependence (true data dependence – read after write)
 - Output dependence (write after write)
 - Anti dependence (write after read)
- Which ones cause stalls in a pipelined machine?
 - For all of them, we need to ensure semantics of the program is correct
 - Flow dependences always need to be obeyed because they constitute true dependence on a value
 - Anti and output dependences exist due to limited number of architectural registers
 - They are dependence on a name, not a value
 - We will later see what we can do about them

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)

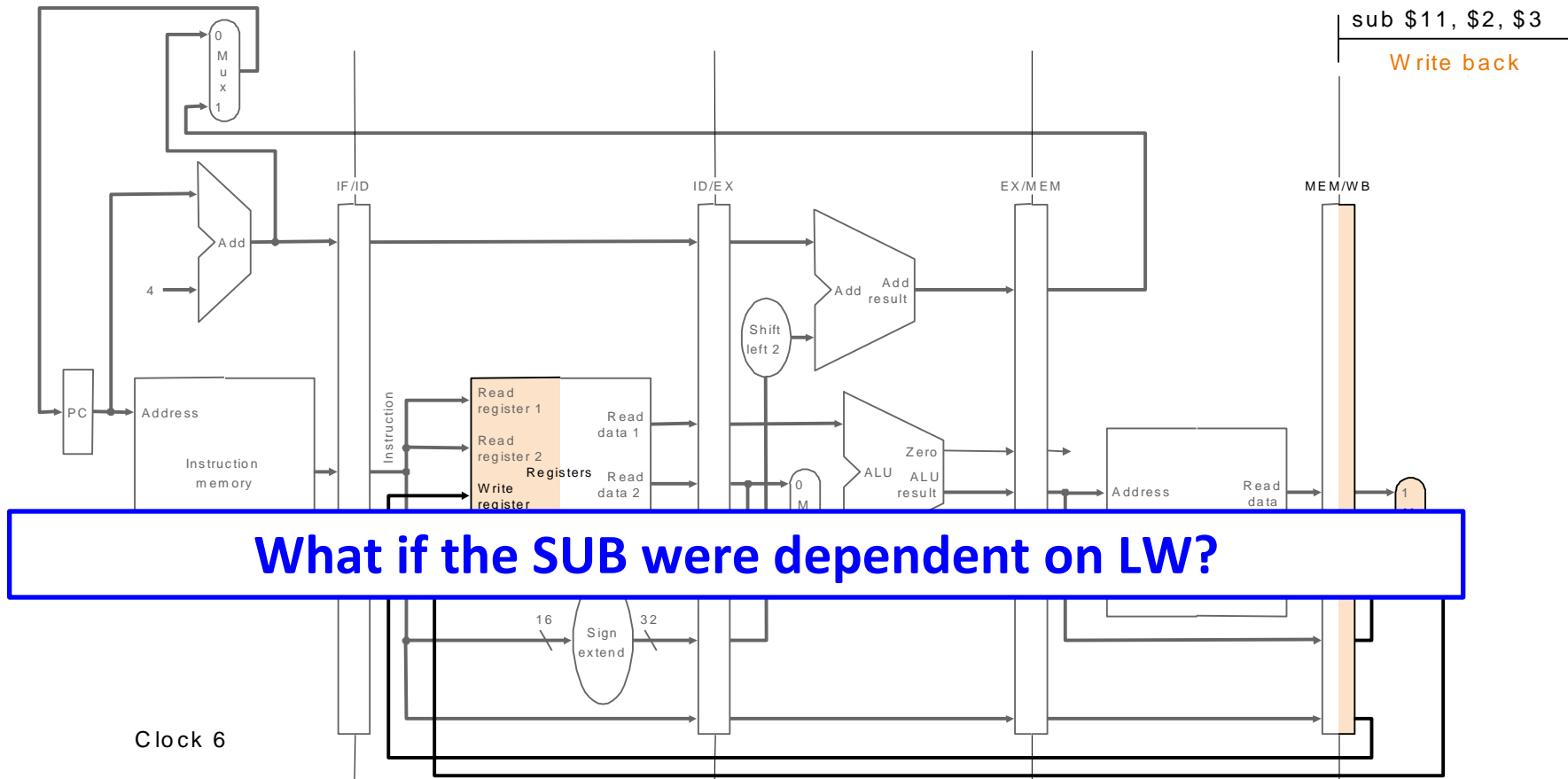
Output-dependence

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



Write-after-Write
(WAW)

Pipelined Operation Example



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

Remember: 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

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



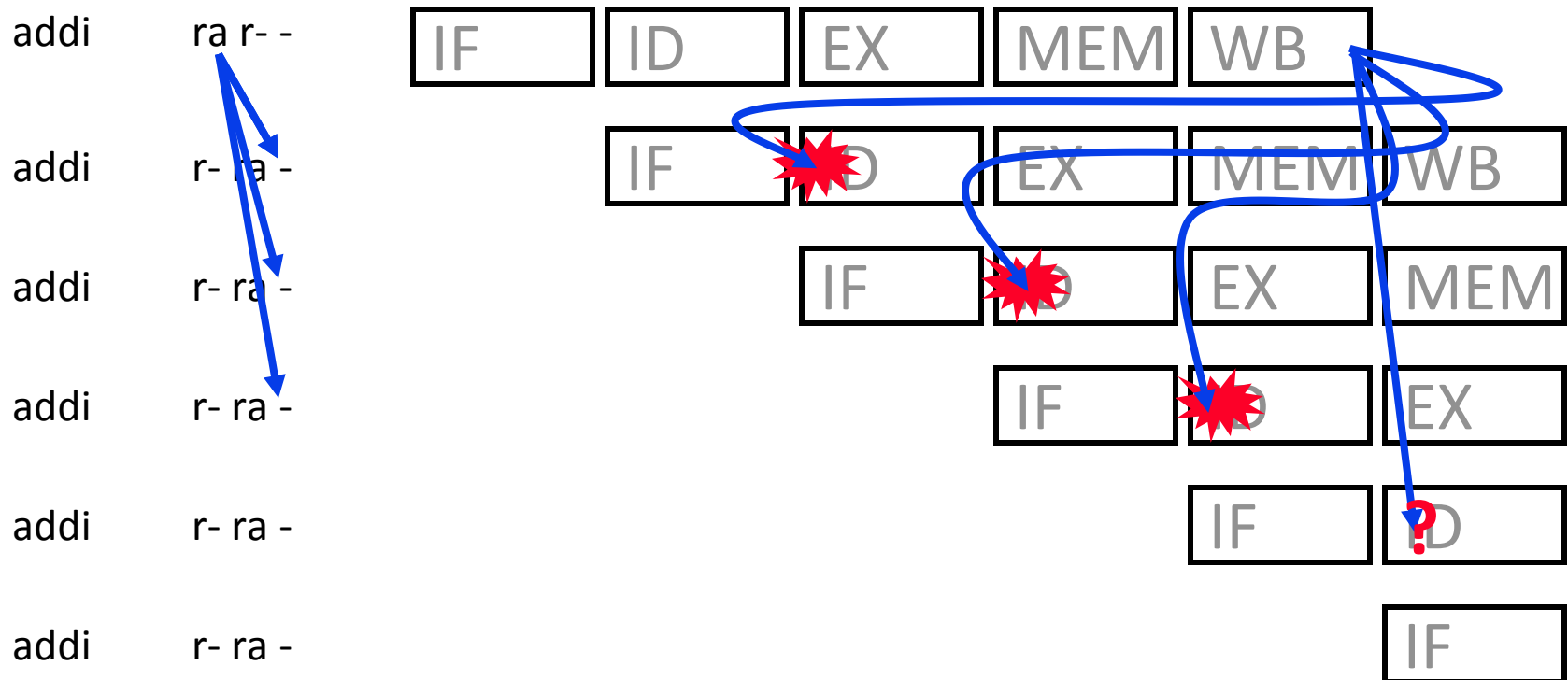
Write-after-Write
(WAW)

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

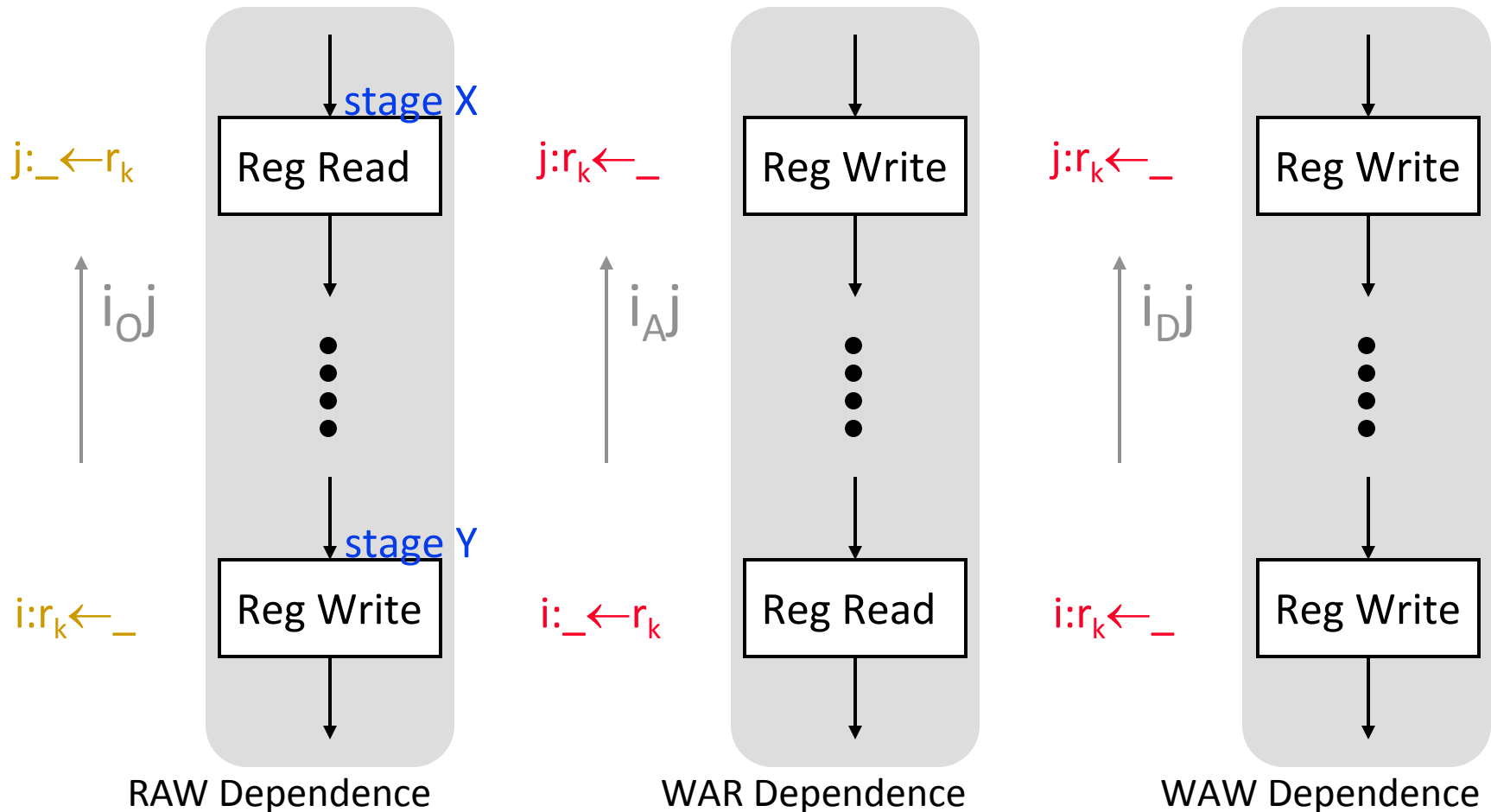


Register Data Dependence Analysis

	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



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

RAW Dependence Analysis Example

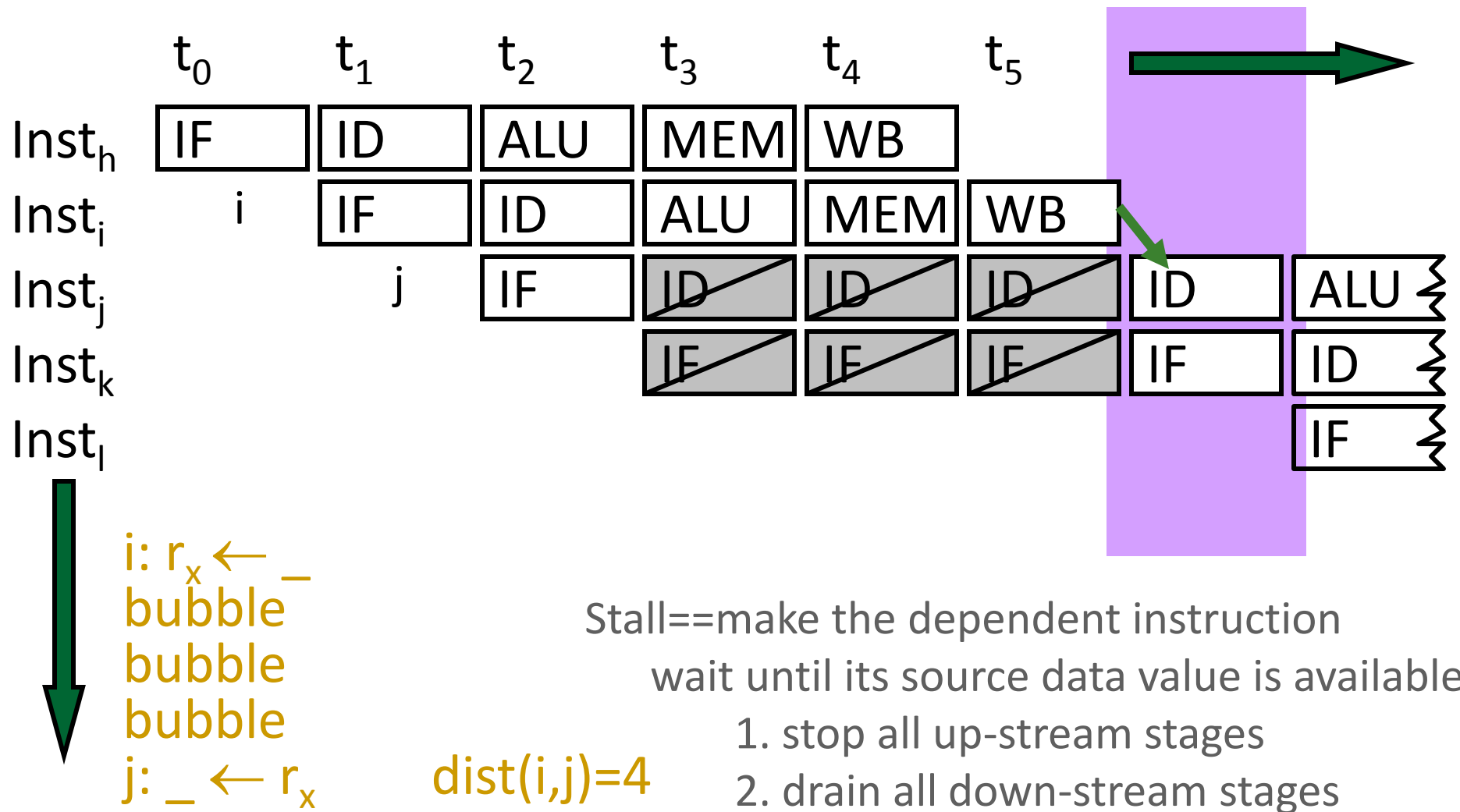
	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

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

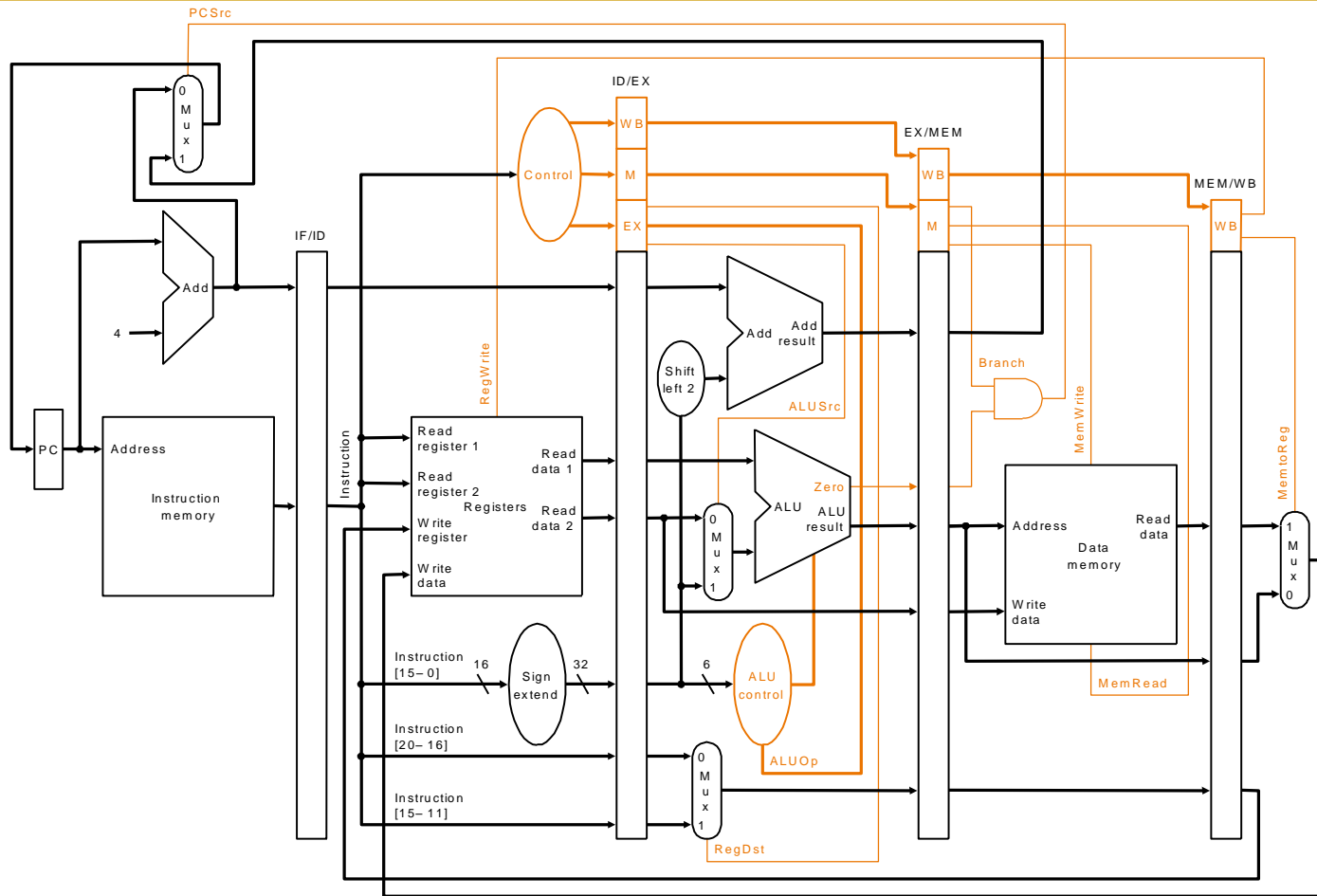
What about WAW and WAR dependence?

What about memory data dependence?

Pipeline Stall: Resolving Data Dependence



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

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)
 - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$
- 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 $rs \neq r0$
- 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

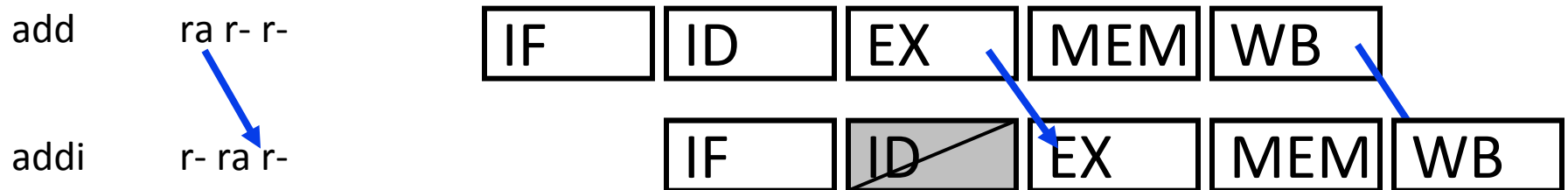
Sample Assembly (P&H)

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

```
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:
```

Data Forwarding (or Data Bypassing)

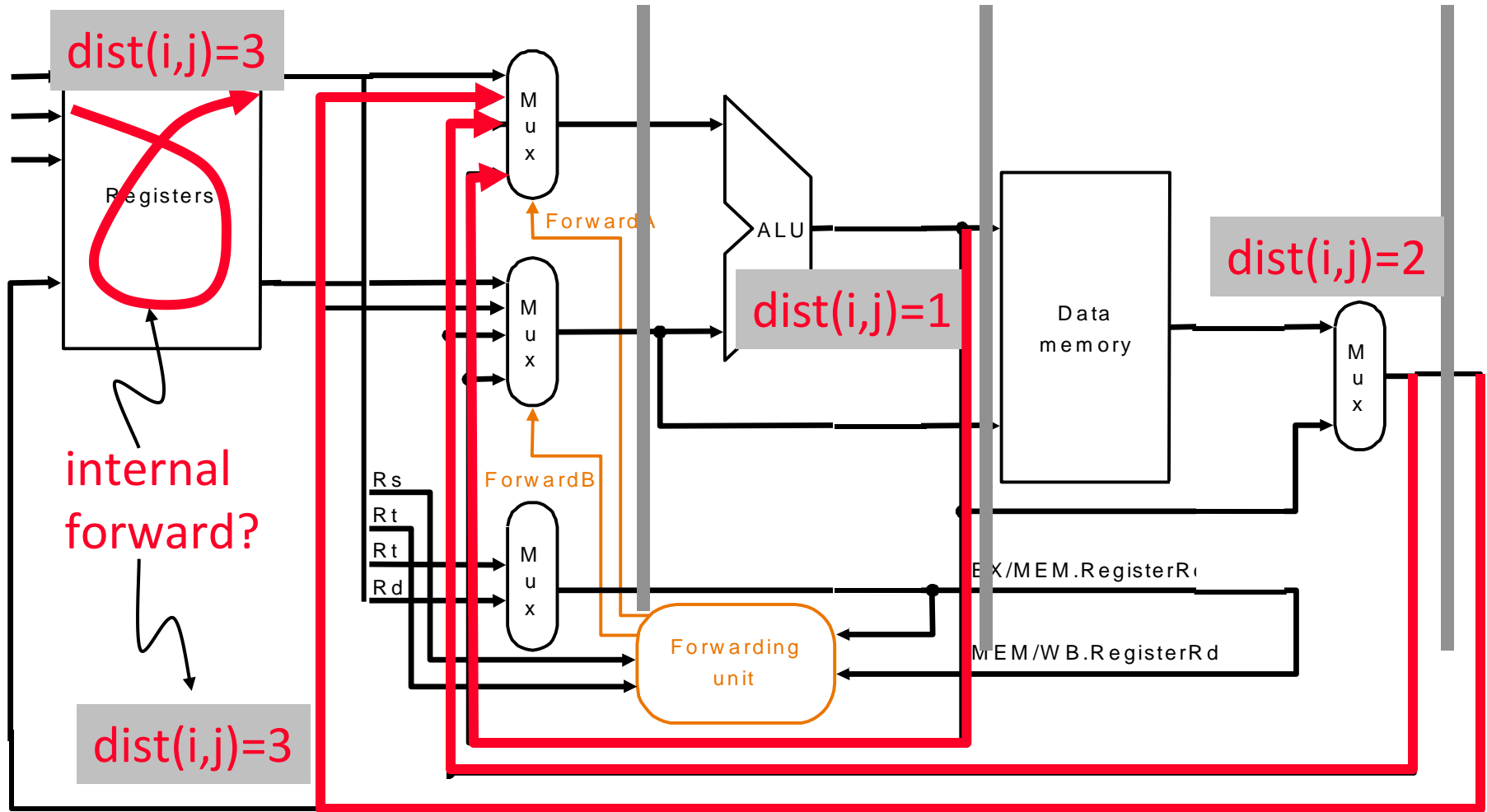
- It is intuitive to think of RF as state
 - “add rx ry rz” literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]
- But, RF is just a part of a computing abstraction
 - “add rx ry rz” means 1. get the results of the last instructions to define the values of RF[ry] and 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, thus



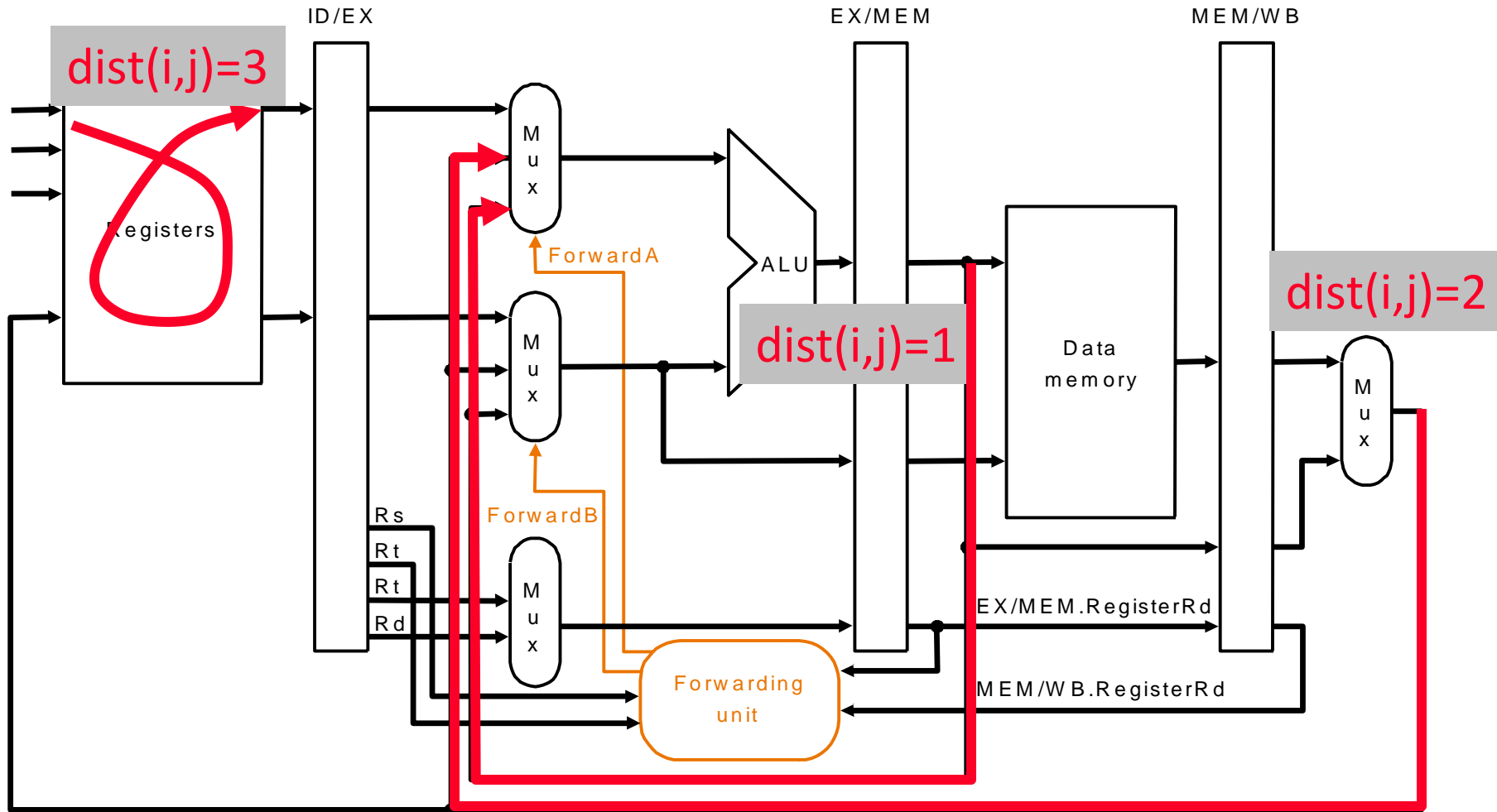
Resolving RAW Dependence with Forwarding

- 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)
 - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$
- 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
 - ⇒ retrieve operand from datapath instead of the RF
 - ⇒ 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 ( $rs_{EX} \neq 0$ ) && ( $rs_{EX} == dest_{MEM}$ ) &&  $RegWrite_{MEM}$  then  
    forward operand from MEM stage          // dist=1  
else if ( $rs_{EX} \neq 0$ ) && ( $rs_{EX} == dest_{WB}$ ) &&  $RegWrite_{WB}$  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

Sample Assembly, Revisited (P&H)

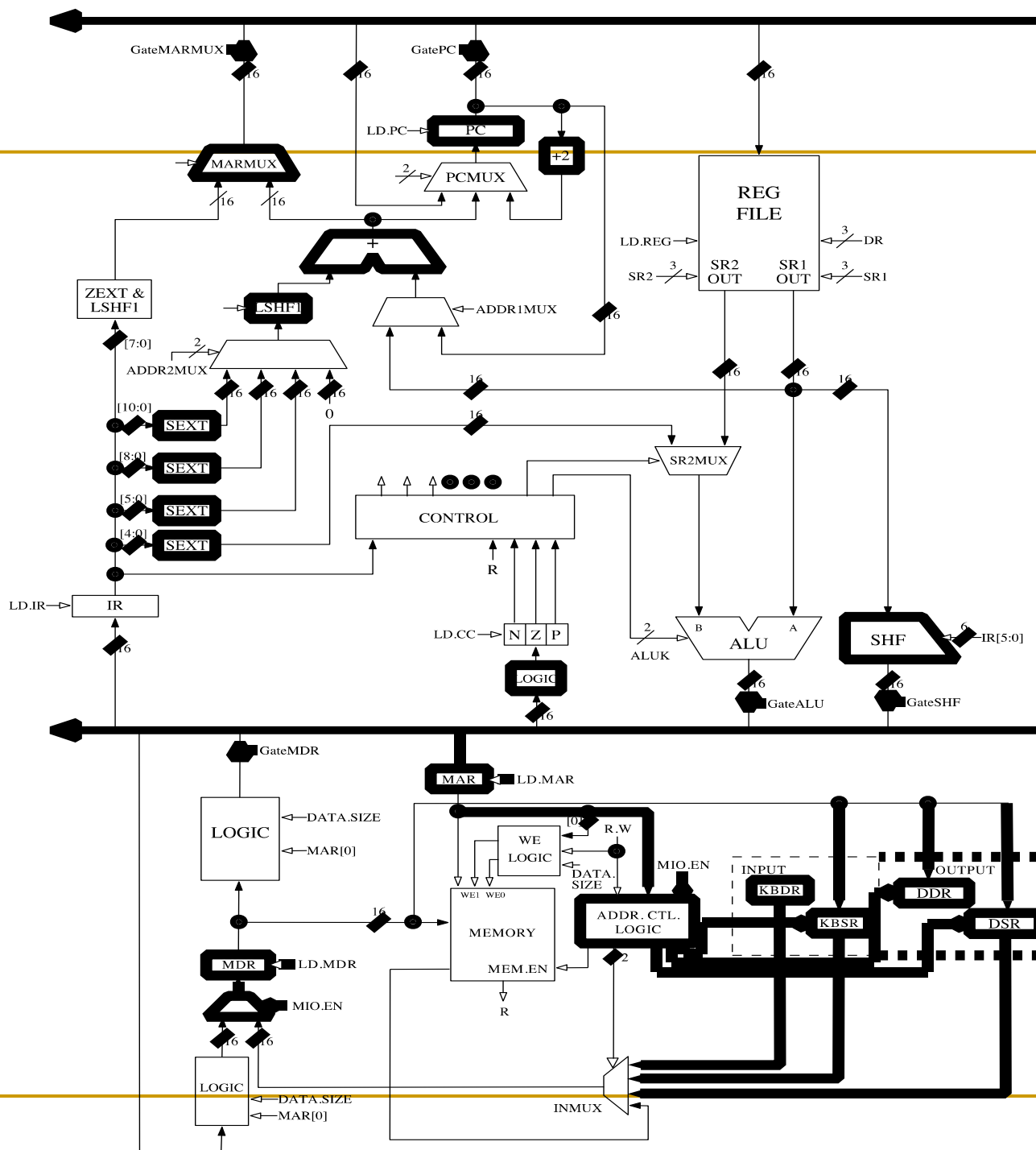
■ for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { }

```
                                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:
```

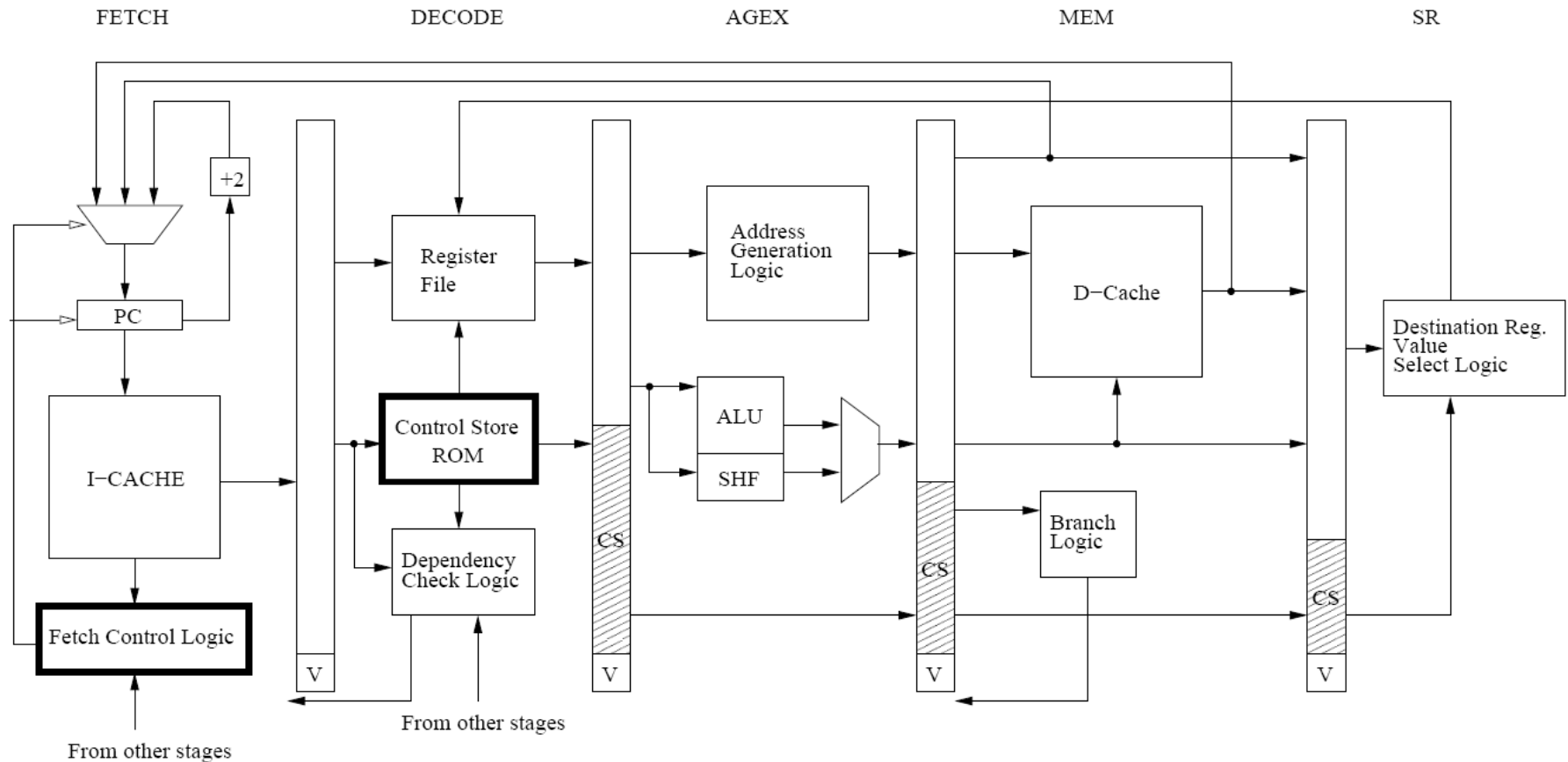
Pipelining the LC-3b

Pipelining the LC-3b

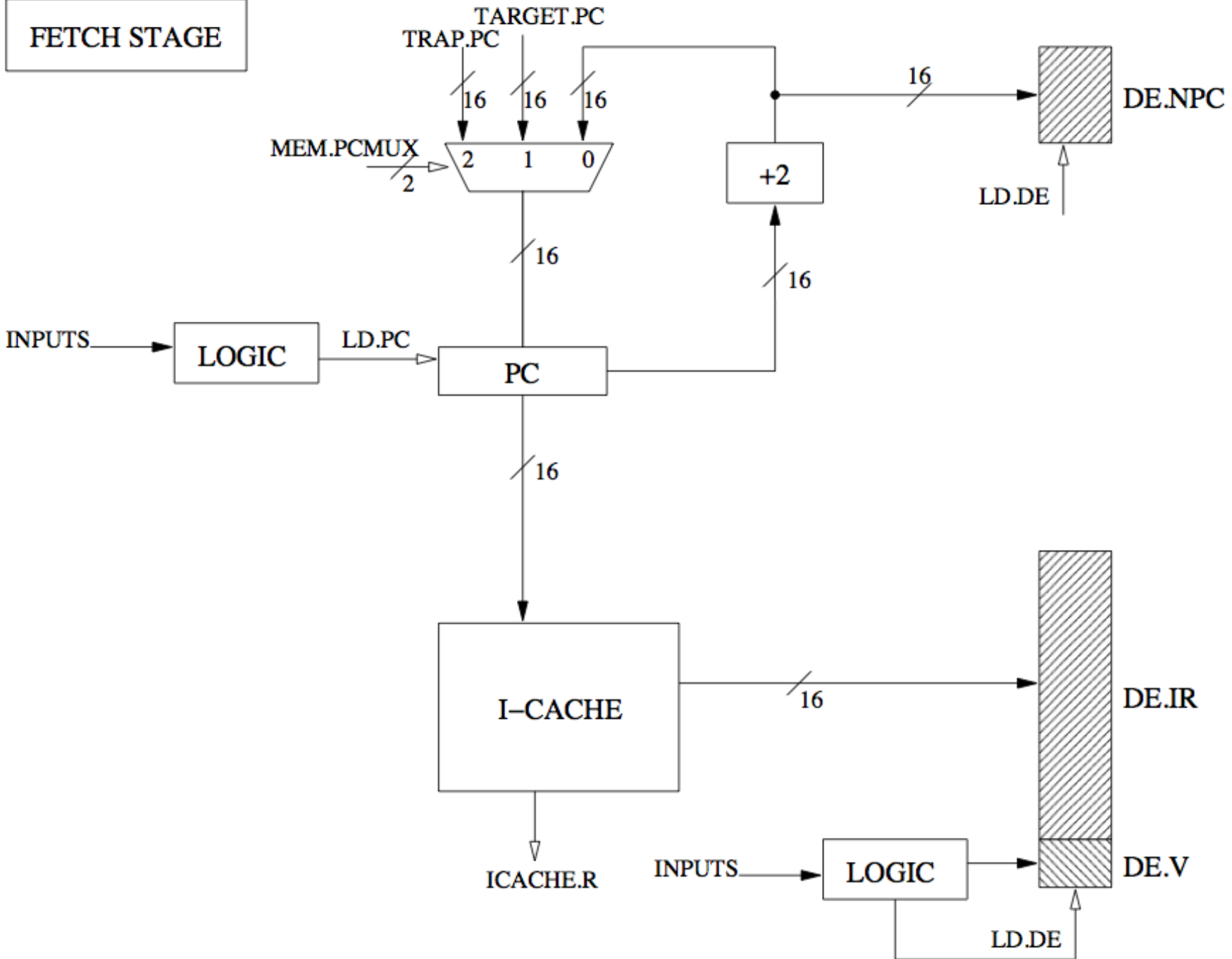
- Let's remember the single-bus datapath
- We'll divide it into 5 stages
 - Fetch
 - Decode/RF Access
 - Address Generation/Execute
 - Memory
 - Store Result
- Conservative handling of data and control dependences
 - Stall on branch
 - Stall on flow dependence

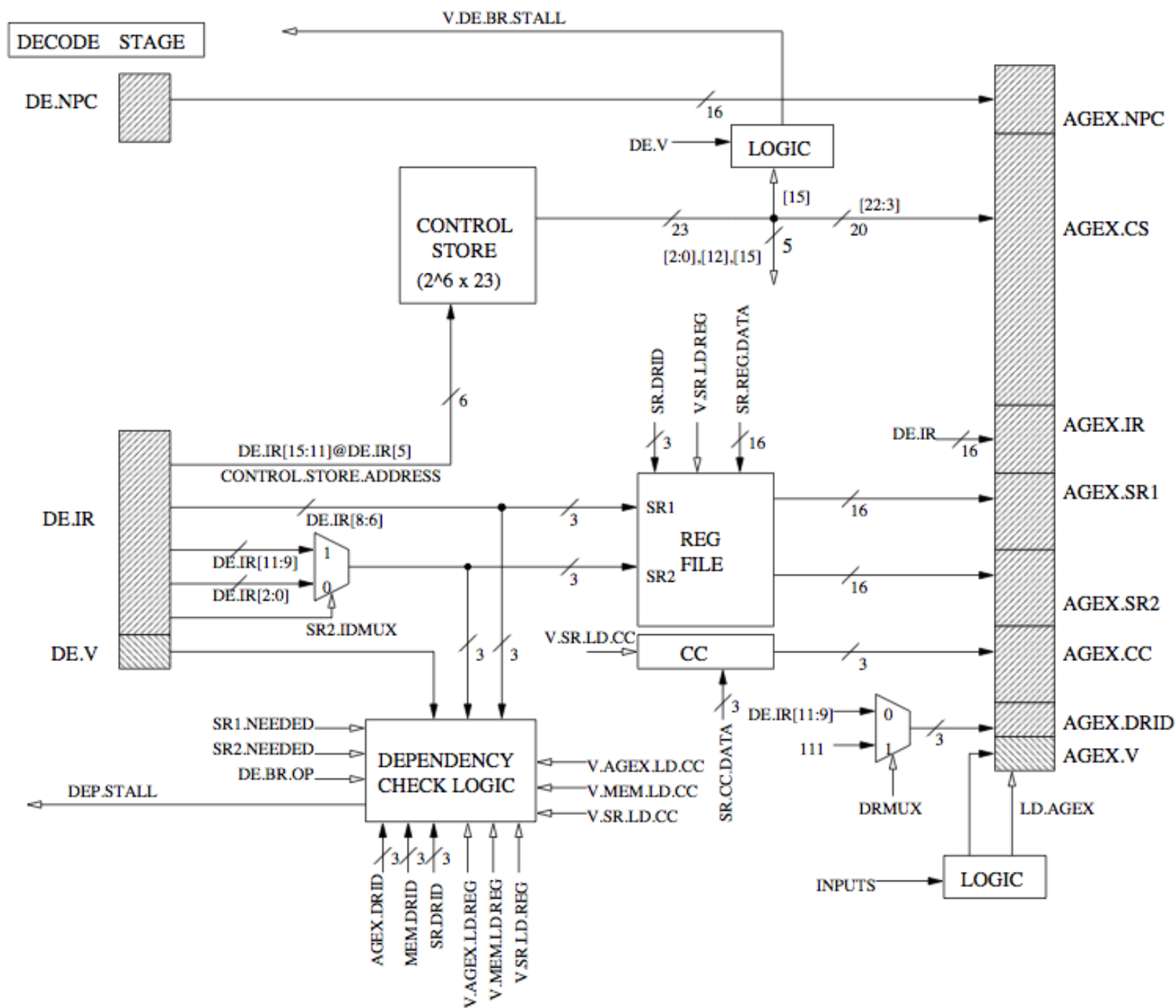


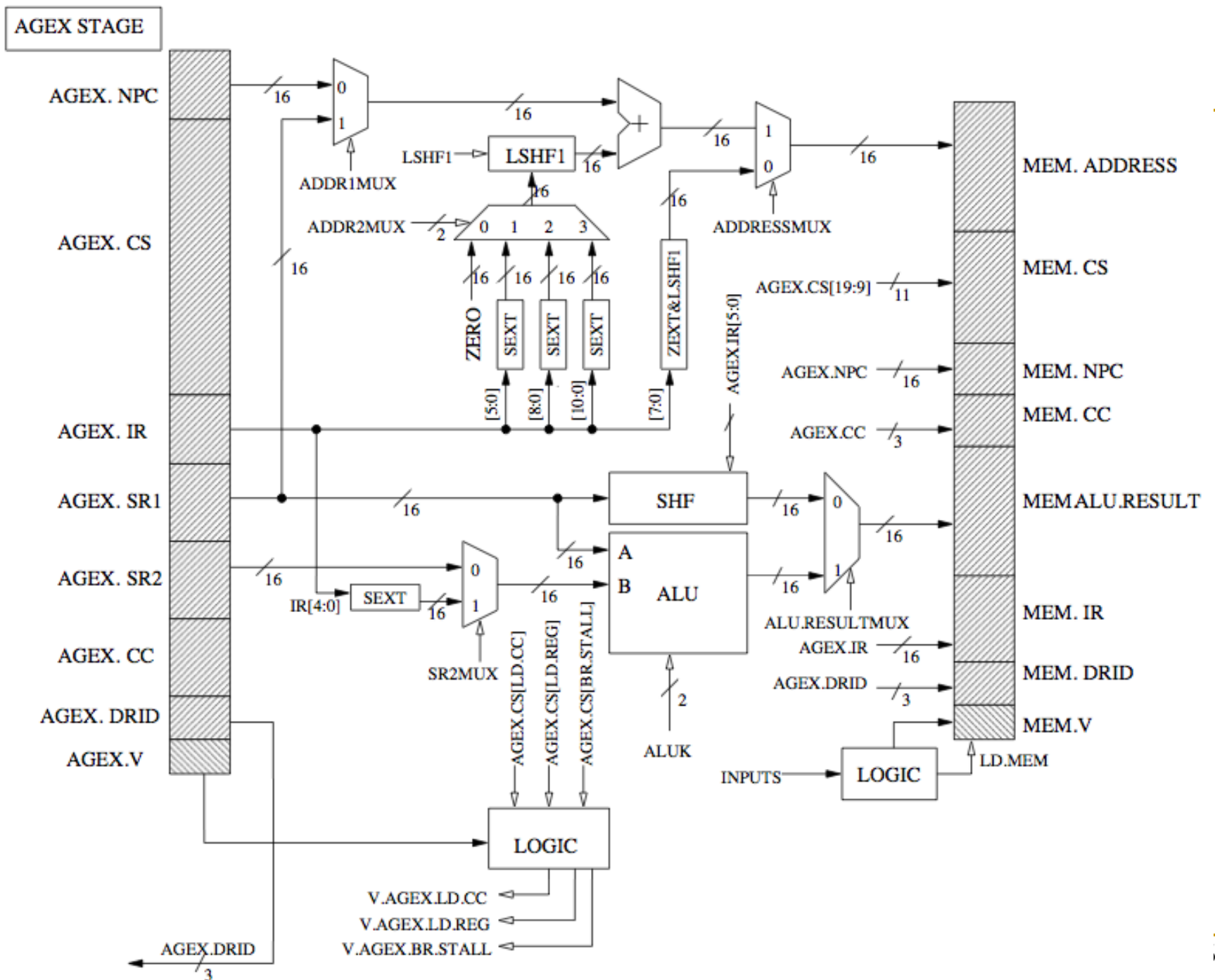
An Example LC-3b Pipeline

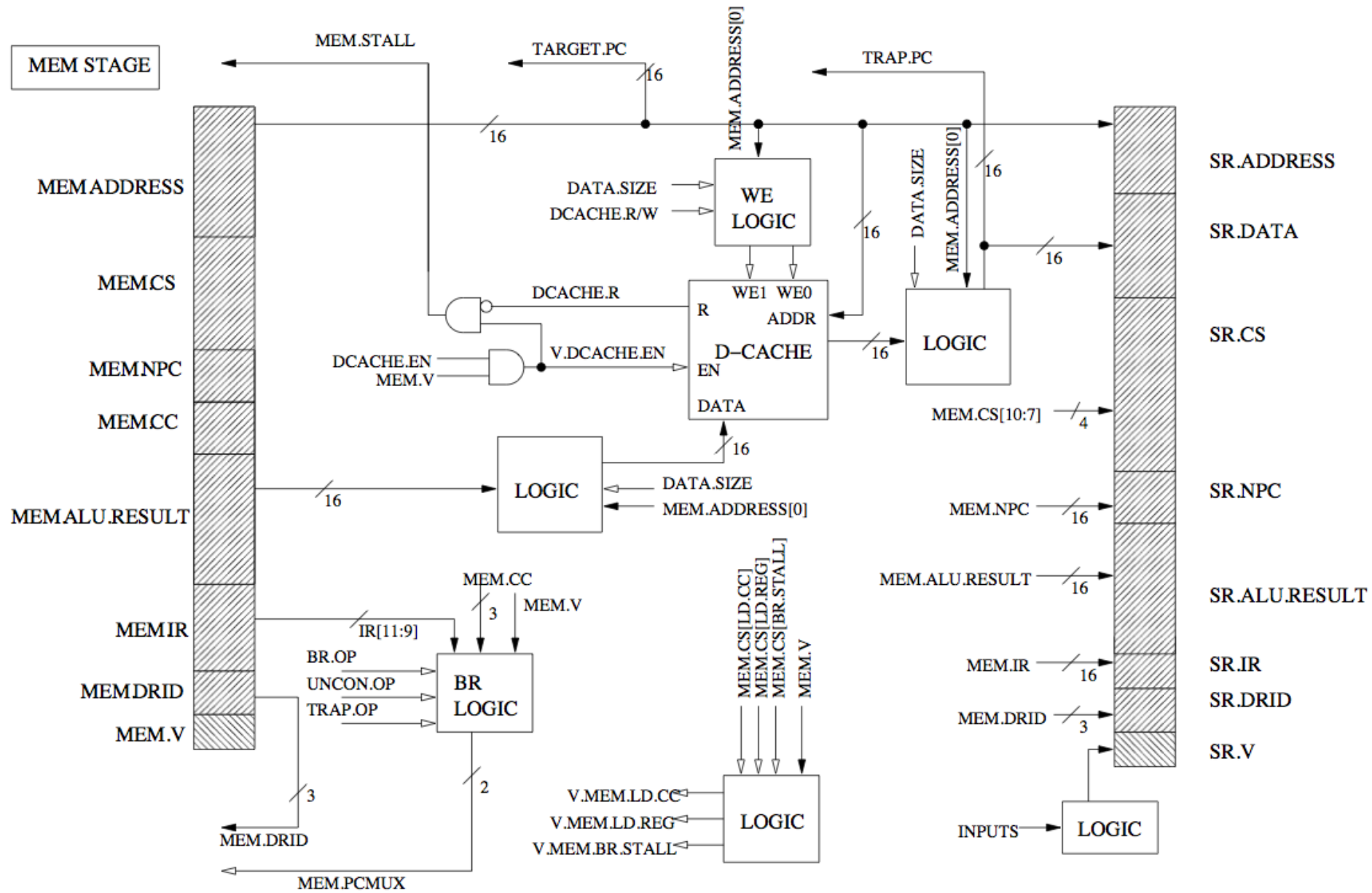


FETCH STAGE

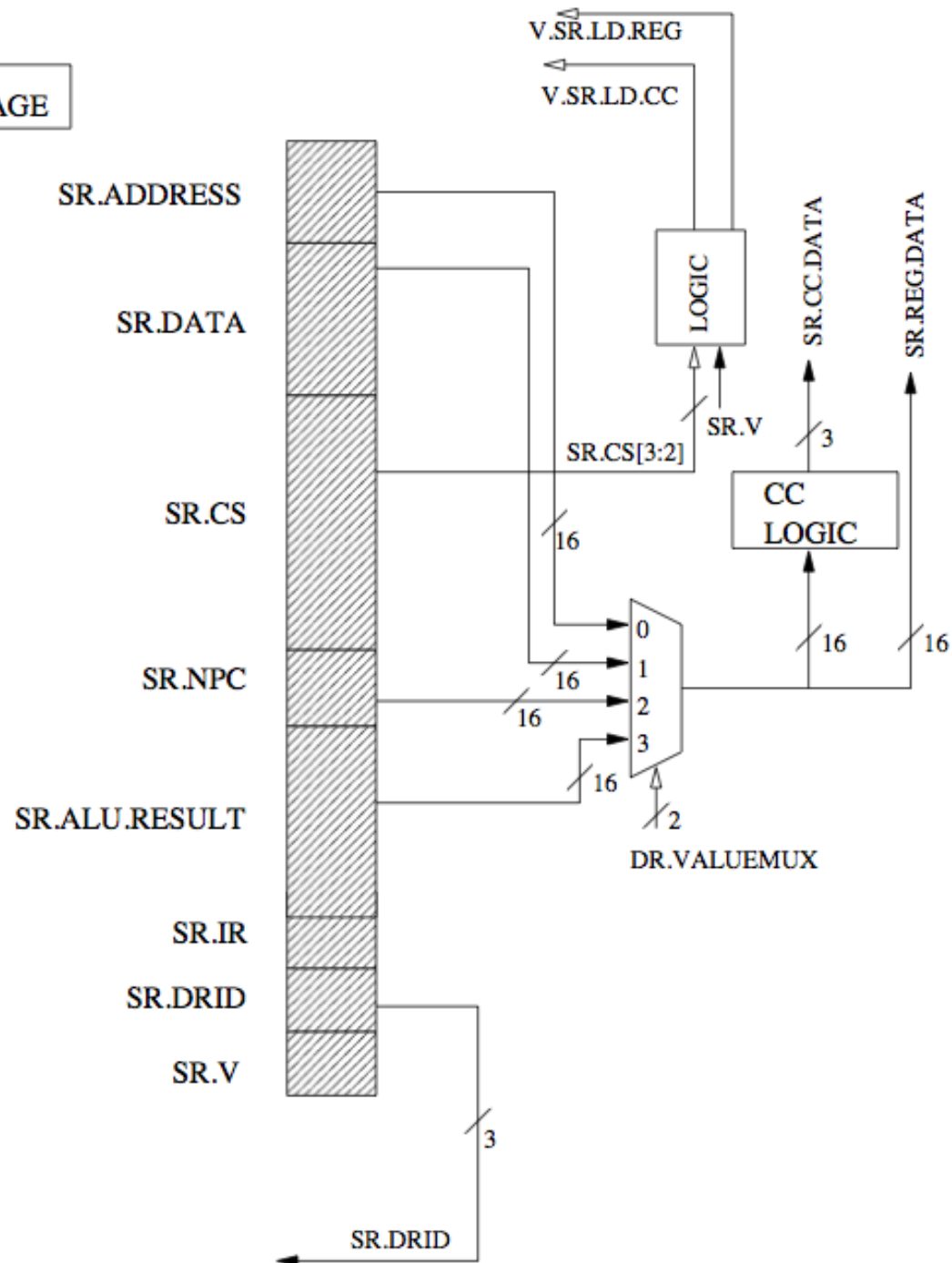








SR STAGE



Control of the LC-3b Pipeline

- Three types of control signals
- Datapath Control Signals
 - Control signals that control the operation of the datapath
- Control Store Signals
 - Control signals (microinstructions) stored in control store to be used in pipelined datapath (can be propagated to stages later than decode)
- Stall Signals
 - Ensure the pipeline operates correctly in the presence of dependencies

Stage	Signal Name	Signal Values	
FETCH	MEM.PCMUX/2:††	PC+2	;select pc+2
		TARGET.PC	;select MEM.TARGET.PC (branch target)
		TRAP.PC	;select MEM.TRAP.PC
		NO(0), LOAD(1)	
DECODE	LD.PC/1:†	NO(0), LOAD(1)	
		NO(0), LOAD(1)	
	DRMUX/1:	11.9	;destination IR[11:9]
		R7	;destination R7
	SR1.NEEDED/1:	NO(0), YES(1)	;asserted if instruction needs SR1
	SR2.NEEDED/1:	NO(0), YES(1)	;asserted if instruction needs SR2
	DE.BR.OP/1:	NO(0), BR(1)	;BR Opcode
	SR2.IDMUX/1:†	2.0	;source IR[2:0]
		11.9	;source IR[11:9]
	LD.AGEX/1:†	NO(0), LOAD(1)	
	V.AGEX.LD.CC/1:††	NO(0), LOAD(1)	
	V.MEM.LD.CC/1:††	NO(0), LOAD(1)	
	V.SR.LD.CC/1:††	NO(0), LOAD(1)	
	V.AGEX.LD.REG/1:††	NO(0), LOAD(1)	
	V.MEM.LD.REG/1:††	NO(0), LOAD(1)	
	V.SR.LD.REG/1:††	NO(0), LOAD(1)	
AGEX	ADDR1MUX/1:	NPC	;select value from AGEX.NPC
		BaseR	;select value from AGEX.SR1(BaseR)
	ADDR2MUX/2:	ZERO	;select the value zero
		offset6	;select SEXT[IR[5:0]]
		PCoffset9	;select SEXT[IR[8:0]]
		PCoffset11	;select SEXT[IR[10:0]]
	LSHF1/1:	NO(0), 1bit Left shift(1)	
	ADDRESSMUX/1:	7.0	;select LSHF(ZEXT[IR[7:0]],1)
		ADDER	;select output of address adder
	SR2MUX/1:	SR2	;select from AGEX.SR2
		4.0	;IR[4:0]
	ALUK/2:	ADD(00), AND(01)	
ALU.RESULTMUX/1:		XOR(10), PASSB(11)	
		SHIFTER	;select output of the shifter
		ALU	;select tput out the ALU
	LD.MEM/1:†	NO(0), LOAD(1)	
MEM	DCACHE.EN/1:	NO(0), YES(1)	;asserted if the instruction accesses memory
	DCACHE.RW/1:	RD(0), WR(1)	
	DATA.SIZE/1:	BYTE(0), WORD(1)	
	BR.OP/1:	NO(0), BR(1)	;BR
	UNCON.OP/1:	NO(0), Uncond.BR(1)	;JMP,RET,JSR,JSRR
SR	TRAP.OP/1:	NO(0), Trap(1)	;TRAP
	DR.VALUEMUX/2:	ADDRESS	;select value from SR.ADDRESS
		DATA	;select value from SR.DATA
		NPC	;select value from SR.NPC
		ALU	;select value from SR.ALU.RESULT
	LD.REG/1:	NO(0), LOAD(1)	
	LD.CC/1:	NO(0), LOAD(1)	

Table 1: Data Path Control Signals
†: The control signal is generated by logic in that stage
††: The control signal is generated by logic in another stage

Control Store in a Pipelined Machine

Number	Signal Name	Stages
0	SR1.NEEDED	DECODE
1	SR2.NEEDED	DECODE
2	DRMUX	DECODE
3	ADDR1MUX	AGEX
4	ADDR2MUX1	AGEX
5	ADDR2MUX0	AGEX
6	LSHF1	AGEX
7	ADDRESSMUX	AGEX
8	SR2MUX	AGEX
9	ALUK1	AGEX
10	ALUK0	AGEX
11	ALU.RESULTMUX	AGEX
12	BR.OP	DECODE, MEM
13	UNCON.OP	MEM
14	TRAP.OP	MEM
15	BR.STALL	DECODE, AGEX, MEM
16	DCACHE.EN	MEM
17	DCACHE.RW	MEM
18	DATA.SIZE	MEM
19	DR.VALUEMUX1	SR
20	DR.VALUEMUX0	SR
21	LD.REG	AGEX, MEM, SR
22	LD.CC	AGEX, MEM, SR

Table 2: Control Store ROM Signals

Stall Signals

- Pipeline stall: Pipeline does not move because an operation in a stage cannot complete
- Stall Signals: Ensure the pipeline operates correctly in the presence of such an operation
- Why could an operation in a stage not complete?

Signal Name	Generated in	
ICACHE.R/1:	FETCH	NO, READY
DEP.STALL/1:	DEC	NO, STALL
V.DE.BR.STALL/1:	DEC	NO, STALL
V.AGEX.BR.STALL/1:	AGEX	NO, STALL
MEM.STALL/1:	MEM	NO, STALL
V.MEM.BR.STALL/1:	MEM	NO, STALL

Table 3: STALL Signals