

18-447

Computer Architecture
Lecture 8: Data and Control
Dependence Handling

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2014, 1/31/2014

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
- McFarling, “Combining Branch Predictors,” DEC WRL Technical Report, 1993.
- Kessler, “The Alpha 21264 Microprocessor,” IEEE Micro 1999.


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

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



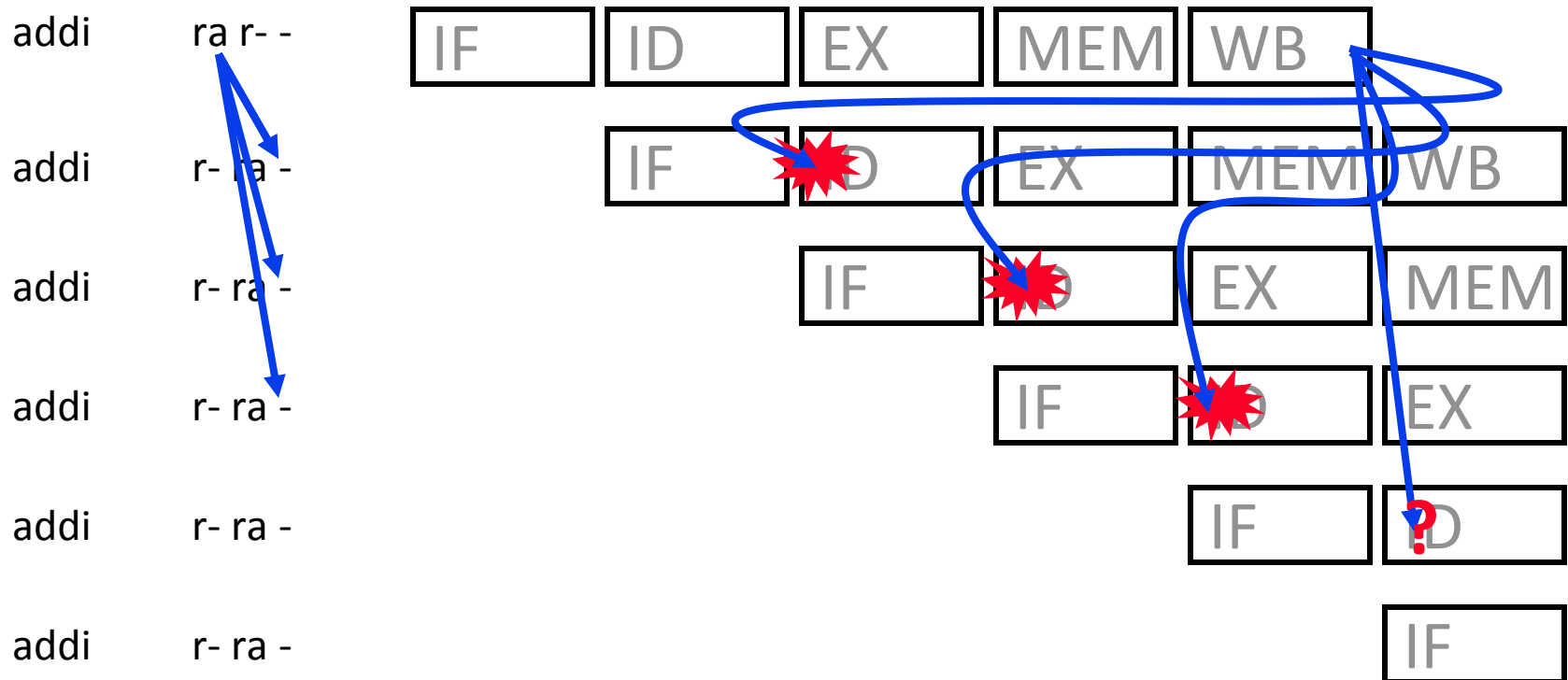
Write-after-Write
(WAW)

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

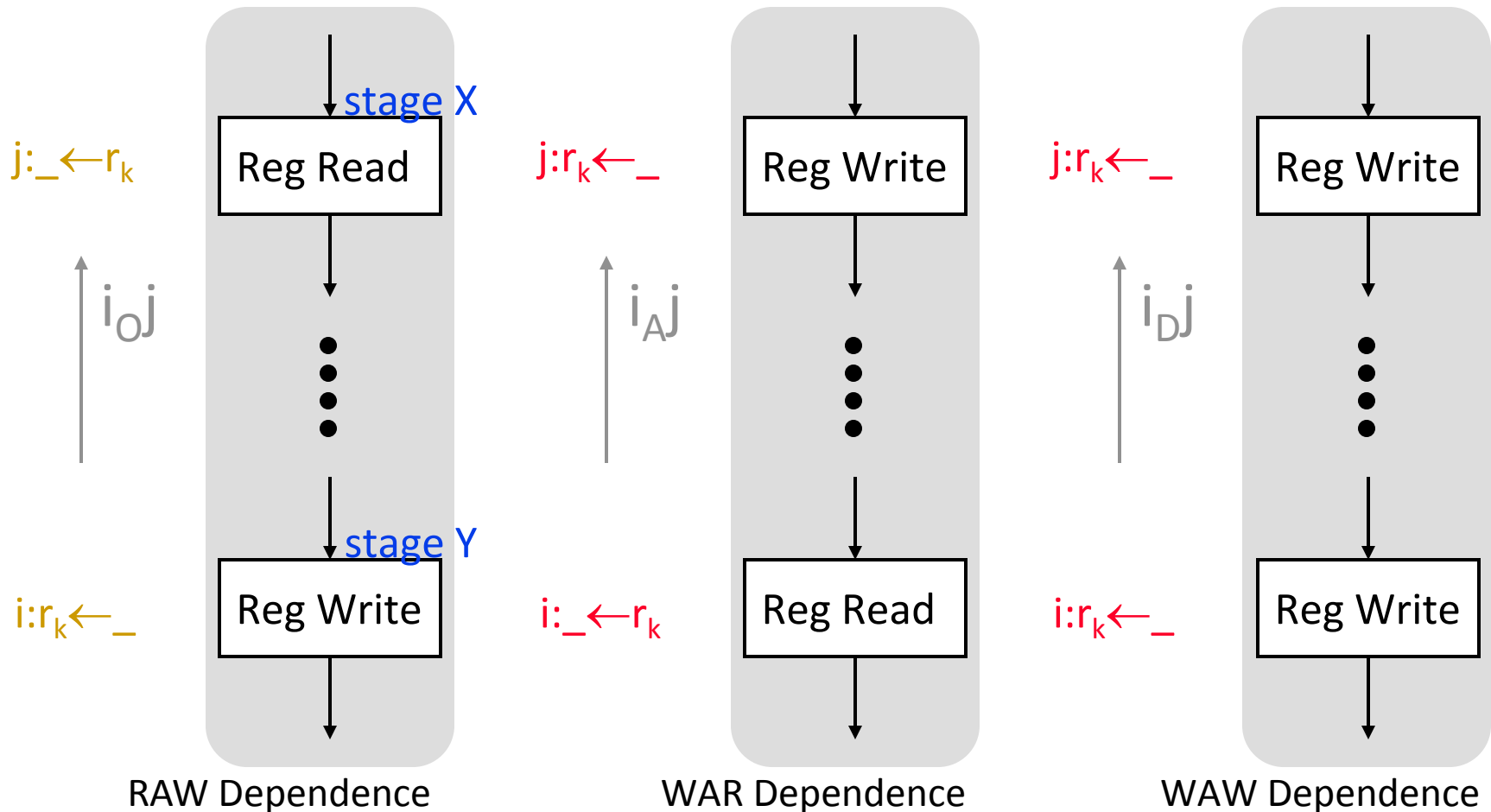


Register Data Dependence Analysis

	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

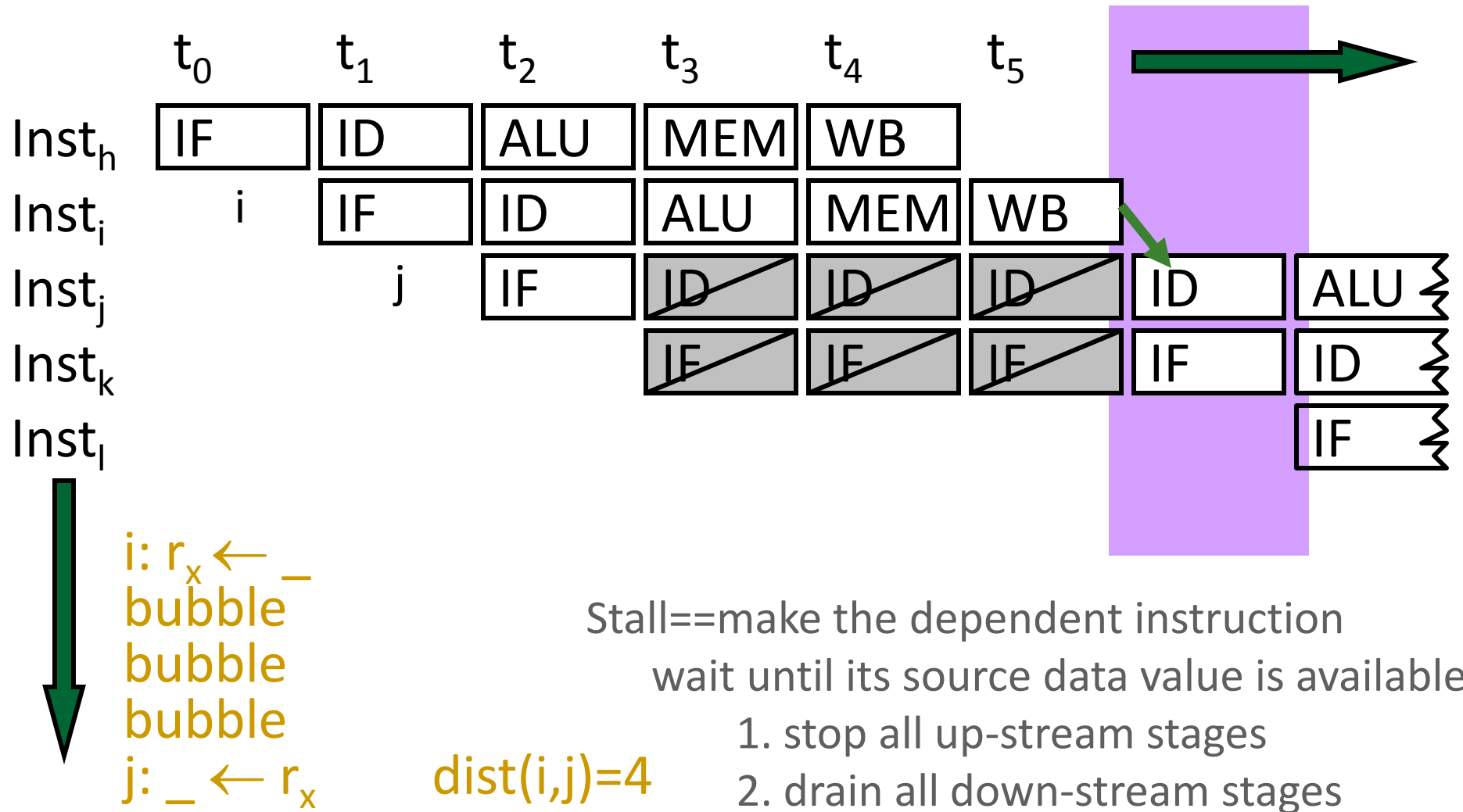
	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

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

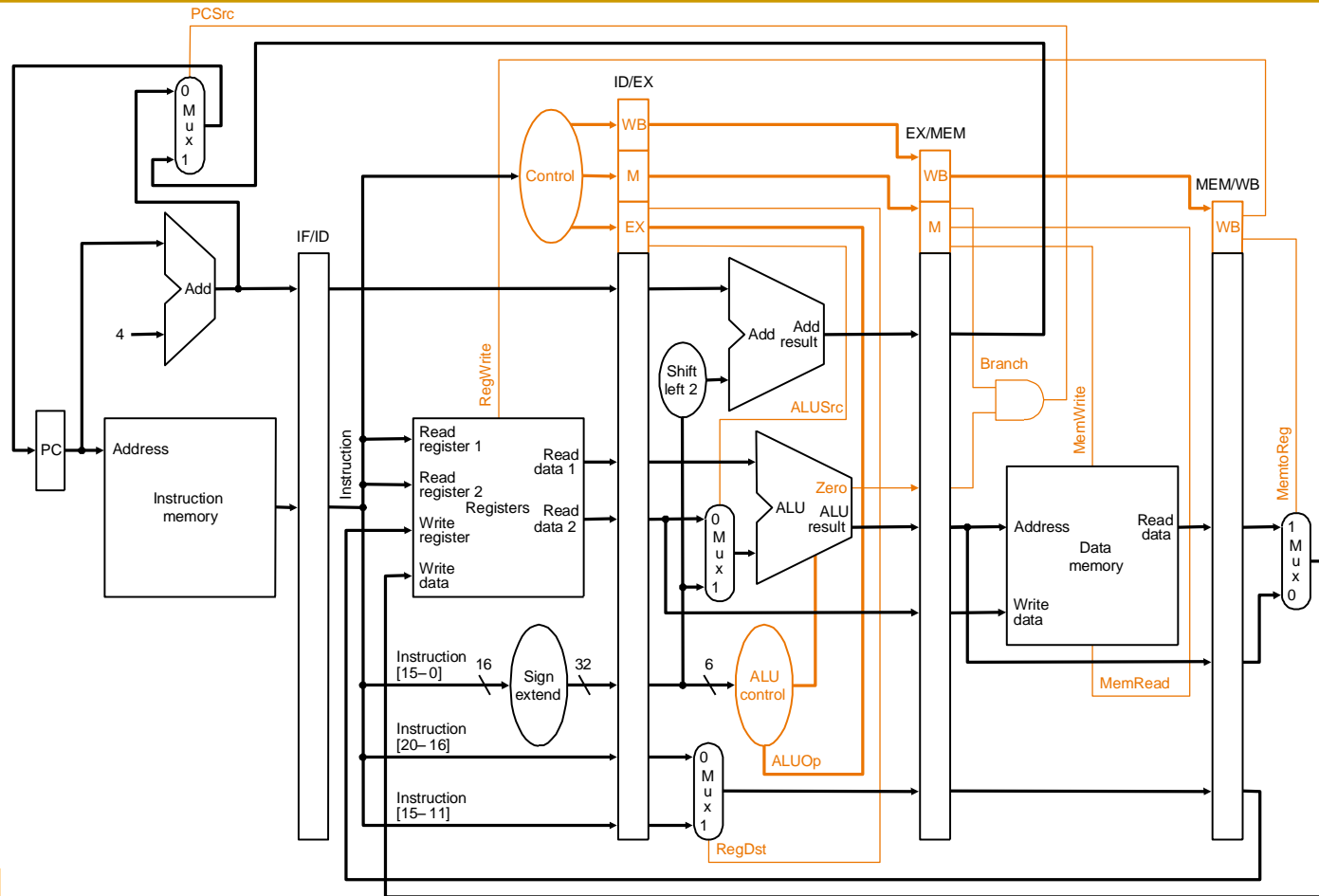
What about WAW and WAR dependence?

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 *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

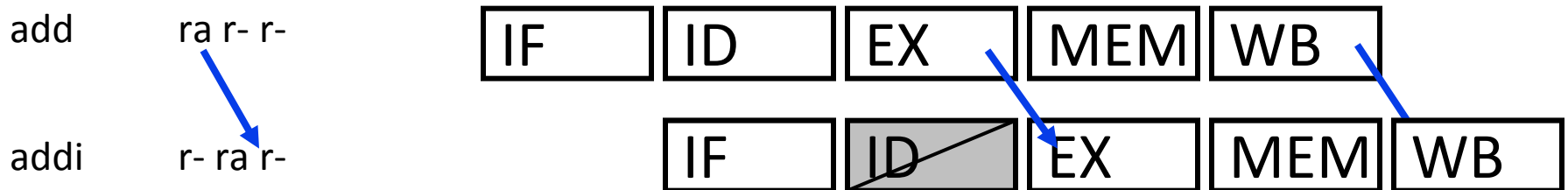
Sample Assembly (P&H)

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

```
for2tst:  addi    $s1, $s0, -1           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)

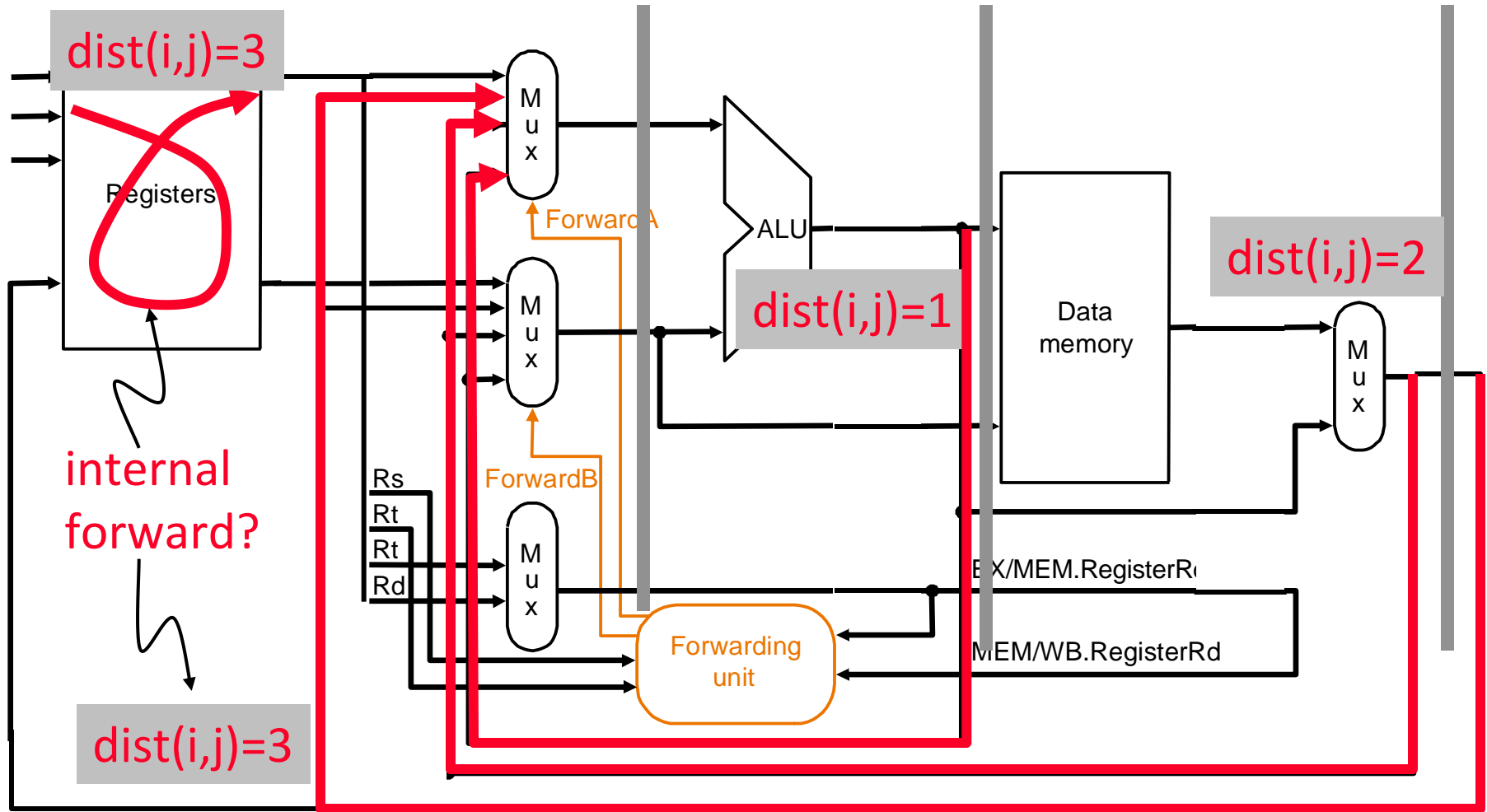
- It is intuitive to think of RF as state
 - “add rx ry rz” literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]
- But, RF is just a part of a communication abstraction
 - “add rx ry rz” means 1. get the results of the last instructions to define the values of RF[ry] and RF[rz], respectively, and 2. until another instruction redefines RF[rx], younger instructions that refer to RF[rx] should use this instruction’s result
- What matters is to maintain the correct “dataflow” between operations, thus



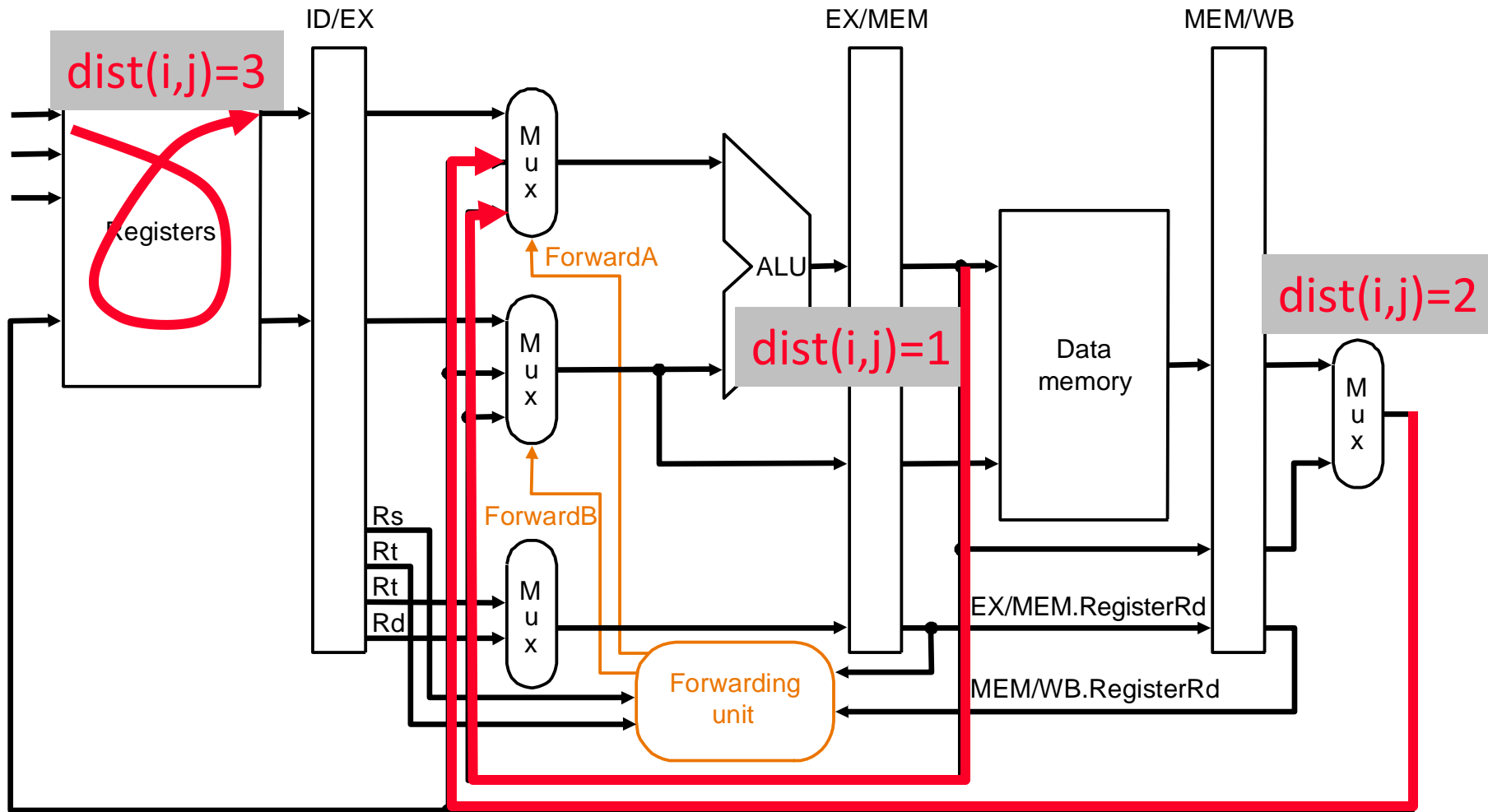
Resolving RAW Dependence with Forwarding

- 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 (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 AEX (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, 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)

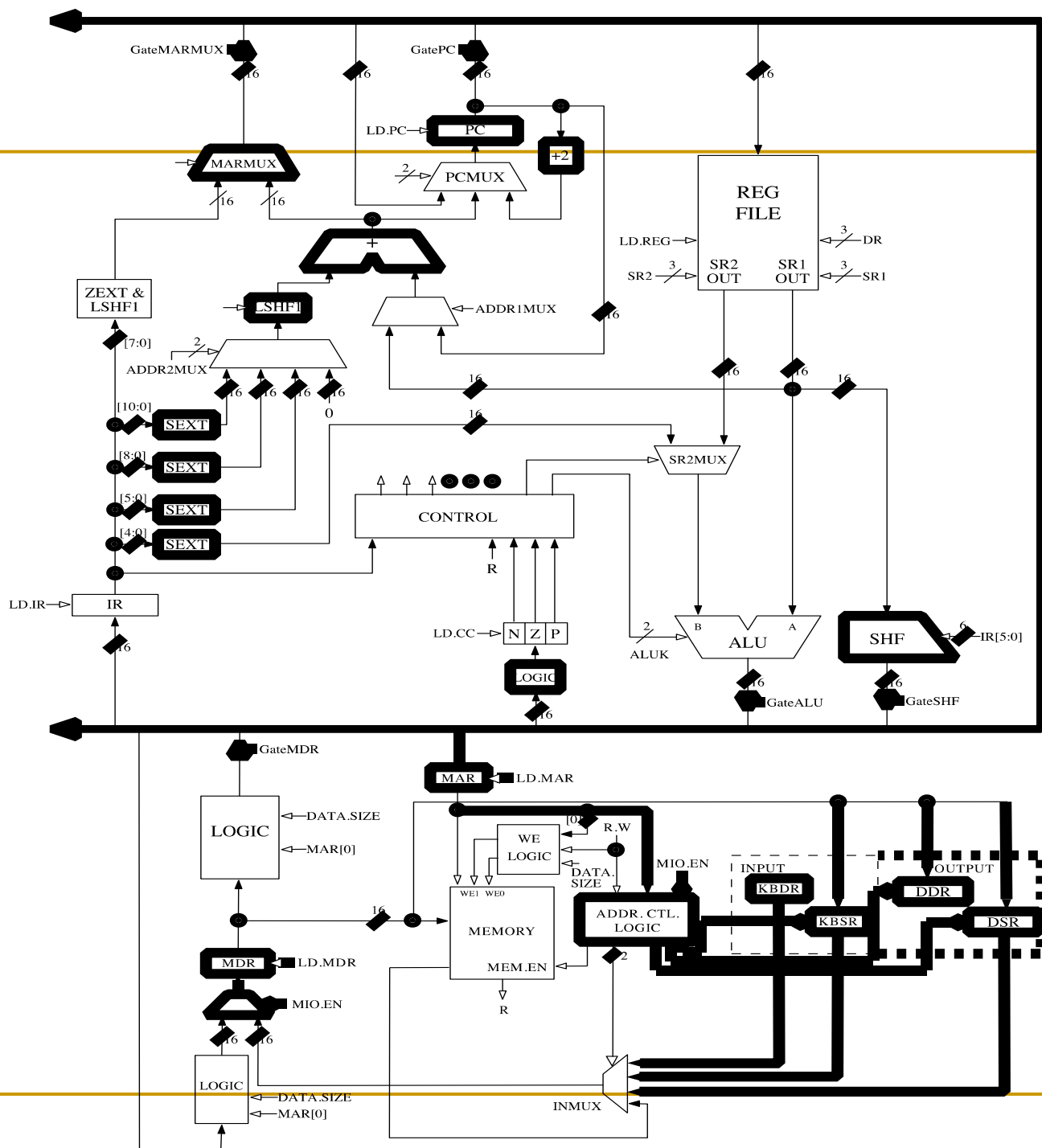
■ for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { }

```
                                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:
```

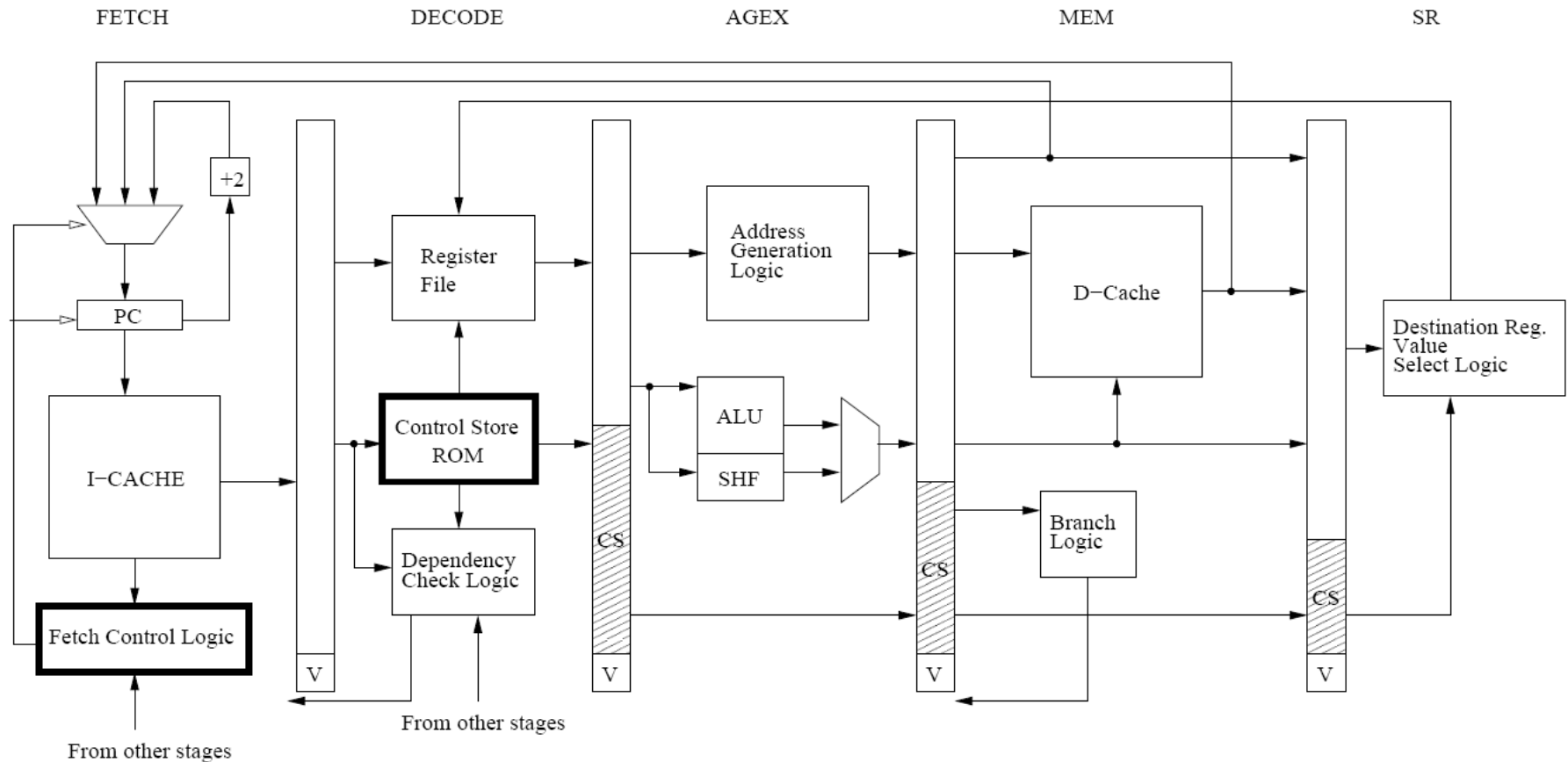
Pipelining the LC-3b

Pipelining the LC-3b

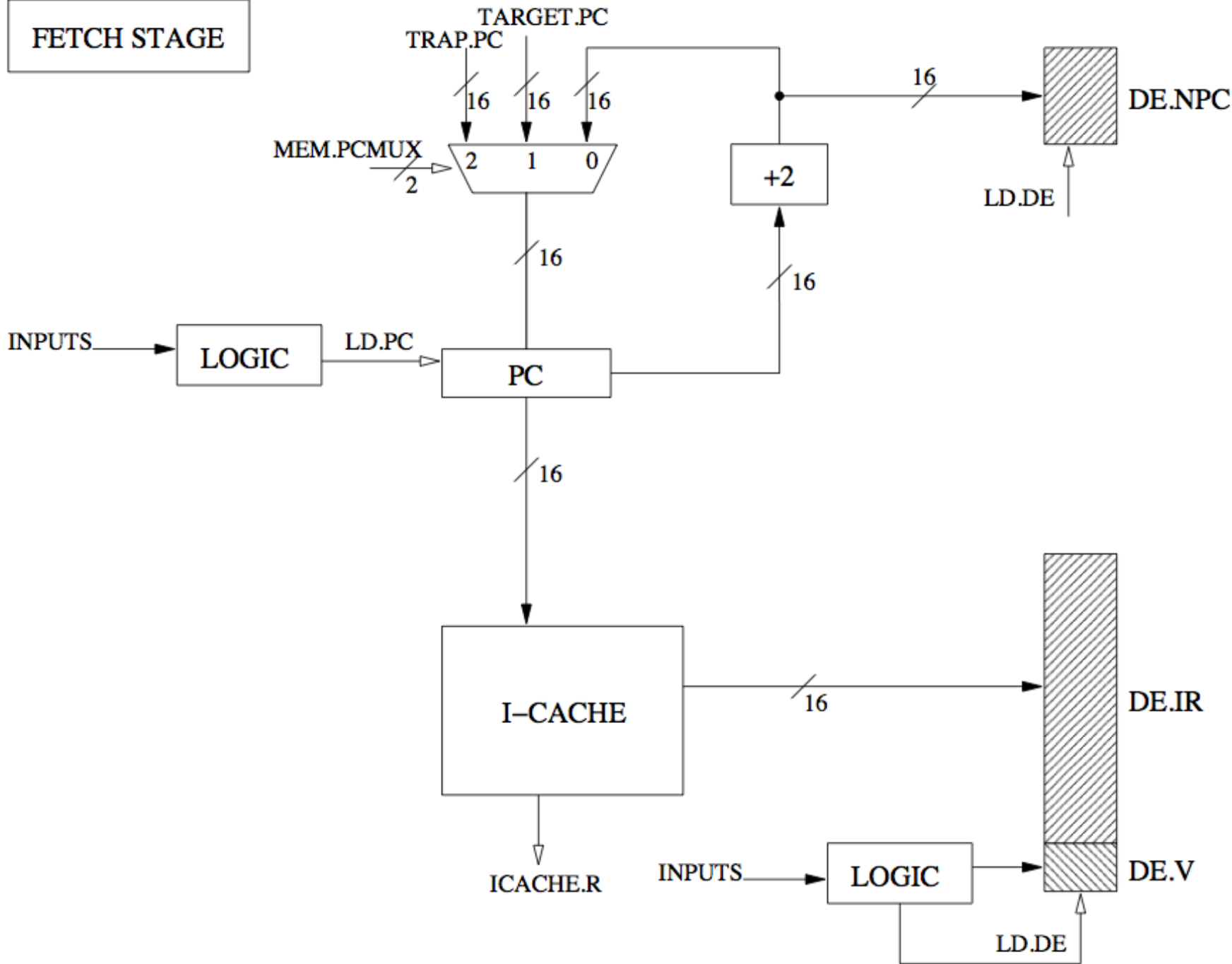
- Let's remember the single-bus datapath
- We'll divide it into 5 stages
 - Fetch
 - Decode/RF Access
 - Address Generation/Execute
 - Memory
 - Store Result
- Conservative handling of data and control dependences
 - Stall on branch
 - Stall on flow dependence

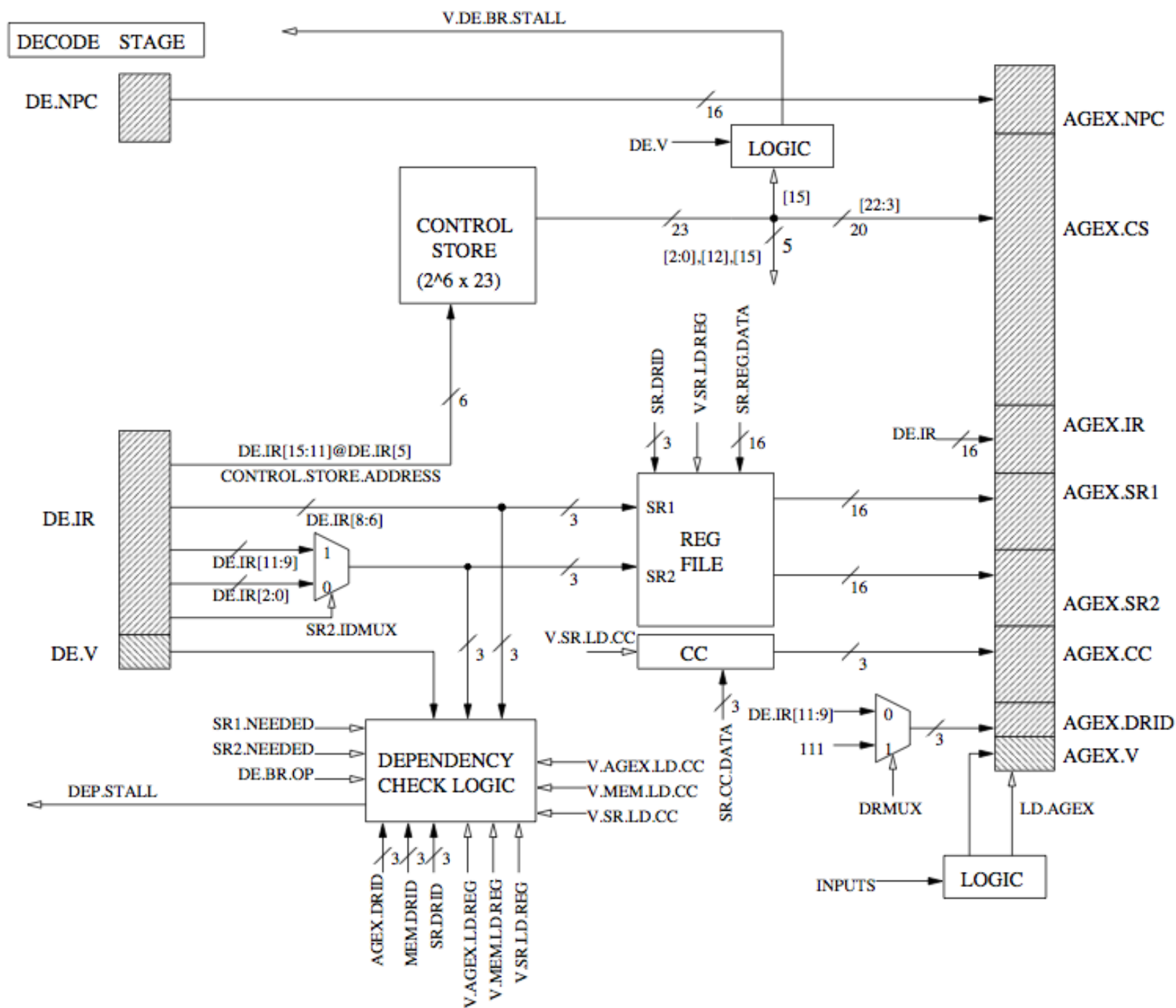


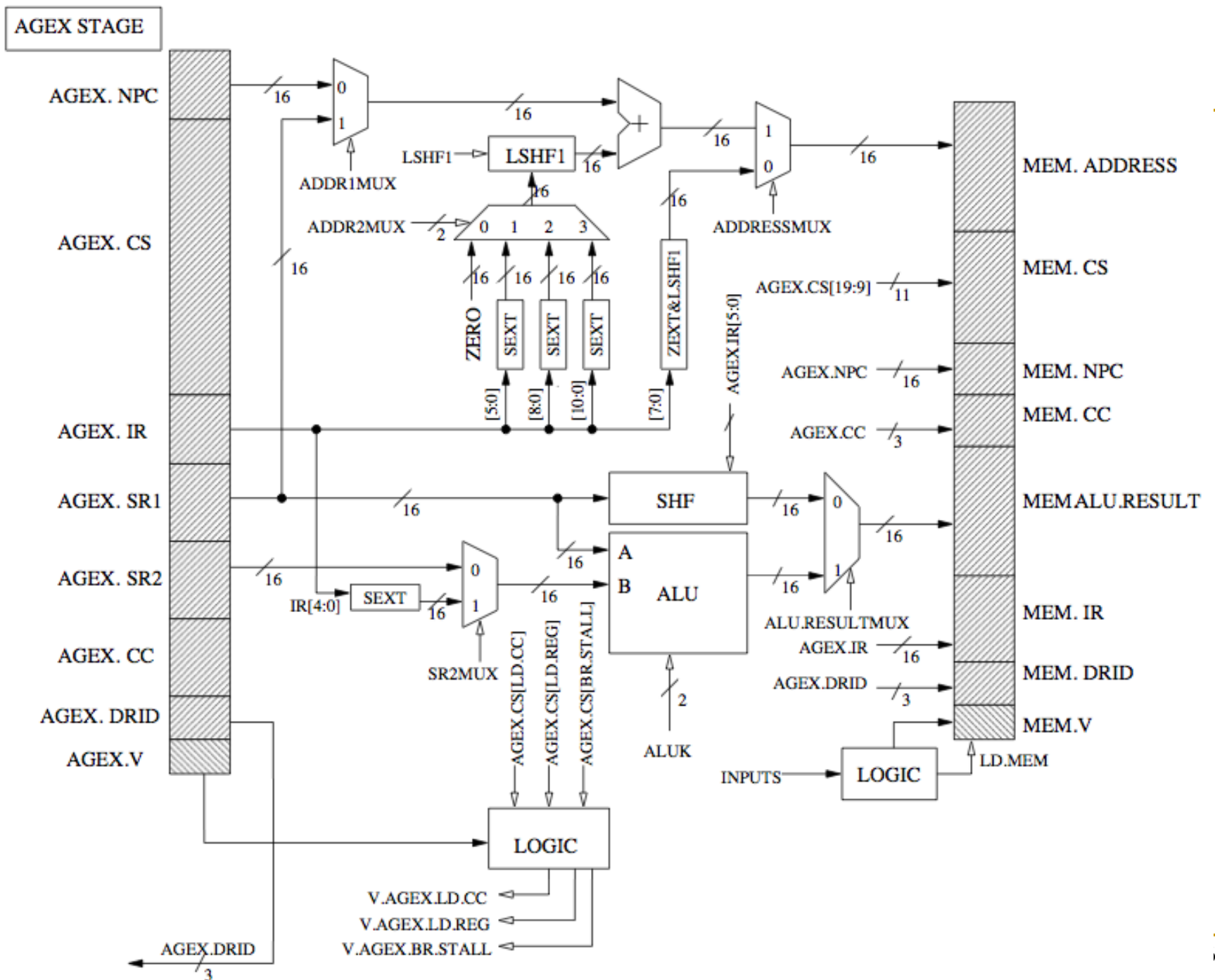
An Example LC-3b Pipeline



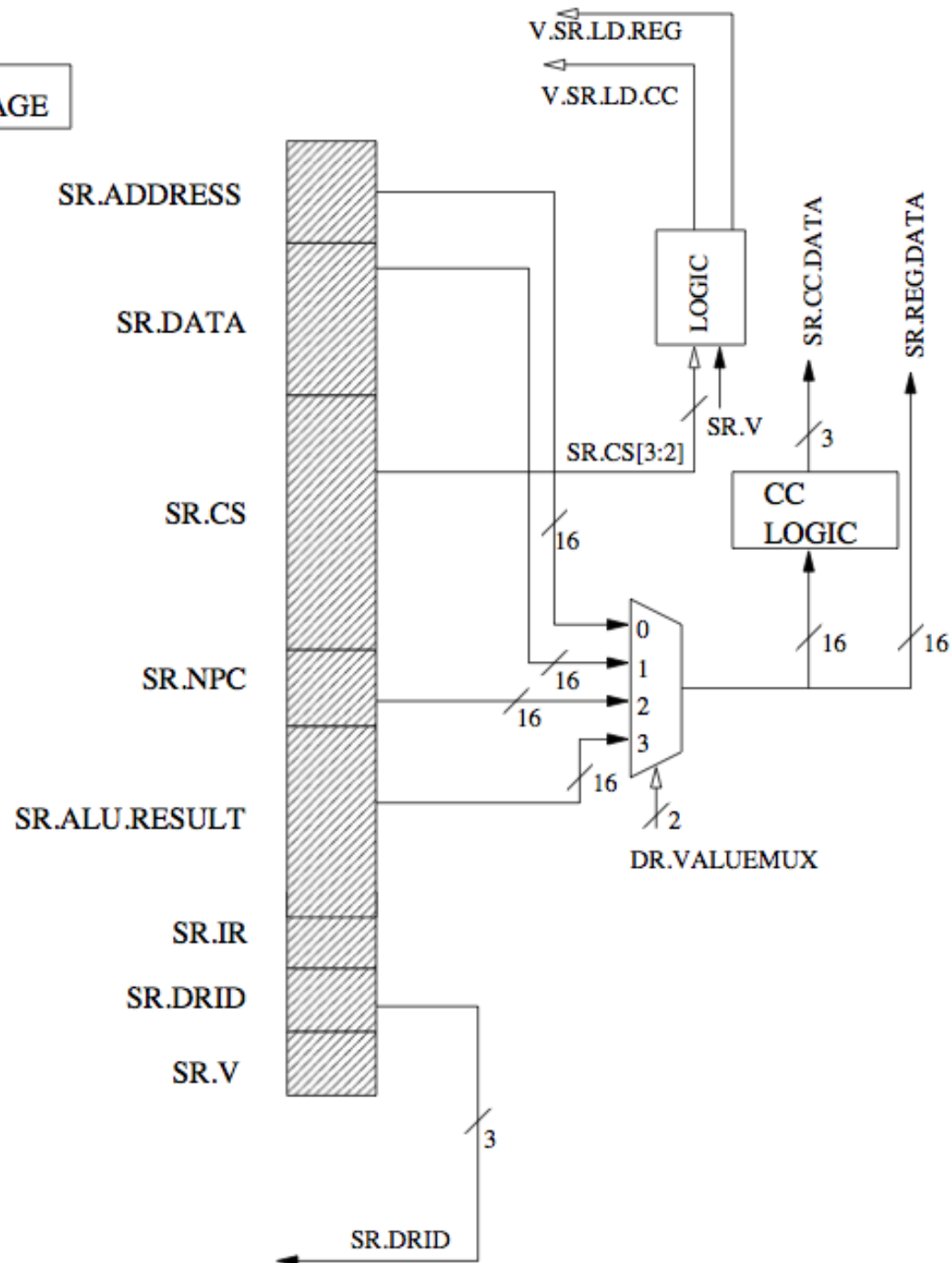
FETCH STAGE







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

Pipelined LC-3b

- <http://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=18447-lc3b-pipelining.pdf>

End of Pipelining the LC-3b

Questions to Ponder

- What is the role of the hardware vs. the software in data dependence handling?
 - ❑ Software based interlocking
 - ❑ Hardware based interlocking
 - ❑ Who inserts/manages the pipeline bubbles?
 - ❑ Who finds the independent instructions to fill “empty” pipeline slots?
 - ❑ What are the advantages/disadvantages of each?

Questions to Ponder

- What is the role of the hardware vs. the software in the order in which instructions are executed in the pipeline?
 - Software based instruction scheduling → static scheduling
 - Hardware based instruction scheduling → dynamic scheduling

More on Software vs. Hardware

- Software based scheduling of instructions → static scheduling
 - ❑ Compiler orders the instructions, hardware executes them in that order
 - ❑ Contrast this with dynamic scheduling (in which hardware will execute instructions out of the compiler-specified order)
 - ❑ How does the compiler know the latency of each instruction?
- What information does the compiler not know that makes static scheduling difficult?
 - ❑ Answer: Anything that is determined at run time
 - Variable-length operation latency, memory addr, branch direction
- How can the compiler alleviate this (i.e., estimate the unknown)?
 - ❑ Answer: Profiling

Control Dependence Handling

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

Branch Types

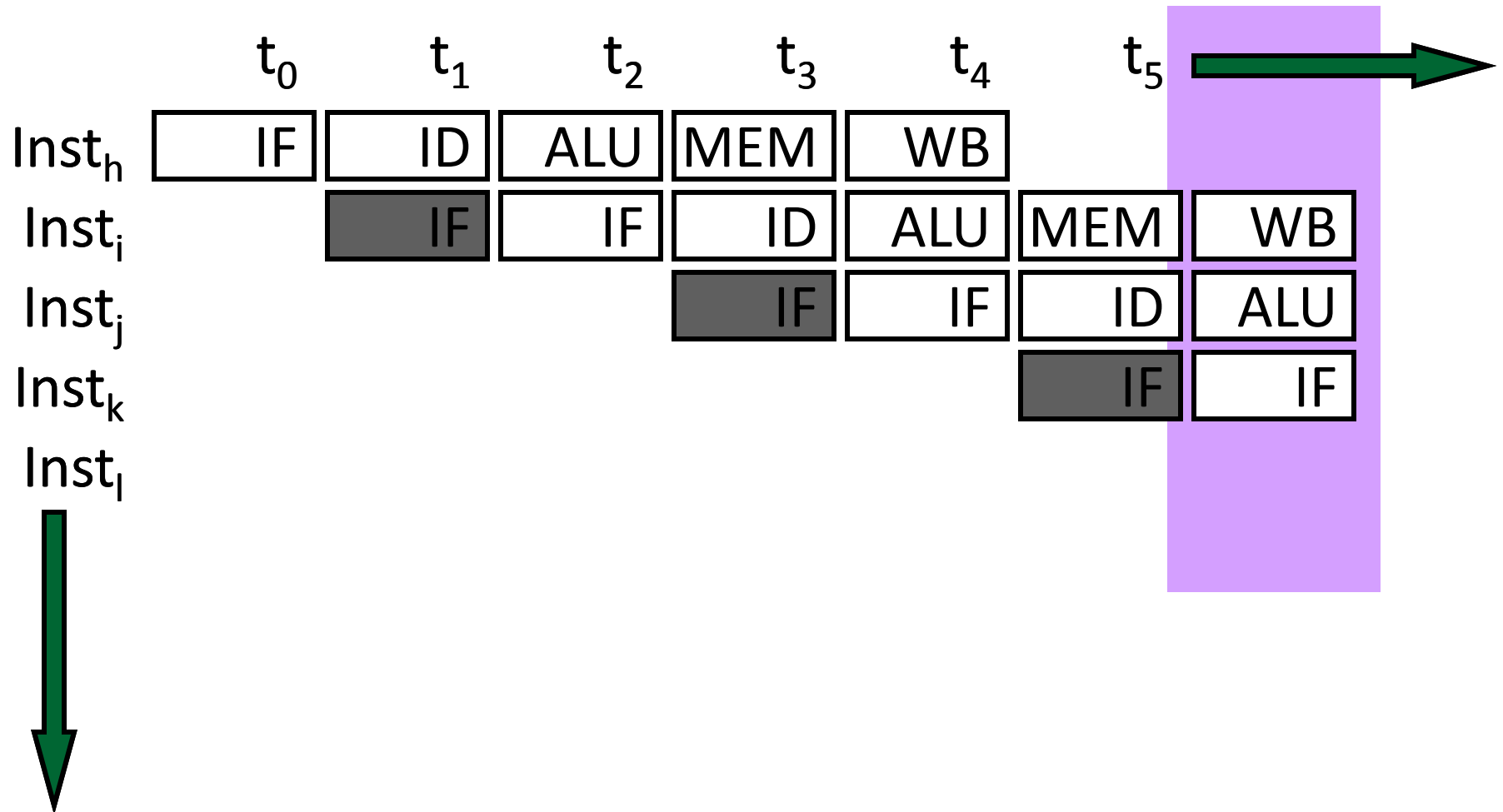
Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)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

Different branch types can be handled differently

How to Handle Control Dependences

- 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**)
 - Do something else (**fine-grained multithreading**)
 - Eliminate control-flow instructions (**predicated execution**)
 - Fetch from both possible paths (if you know the addresses of both possible paths) (**multipath execution**)

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
 - Hardware: ??? (how can you do this in hardware...)

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

- How else can you make this more effective?
- Idea: Get rid of control flow instructions (or minimize their occurrence)
- How?
 1. Get rid of unnecessary control flow instructions → combine predicates (predicate combining)
 2. Convert control dependences into data dependences → predicated execution

Predicate Combining (*not* Predicated Execution)

- Complex predicates are converted into multiple branches
 - `if ((a == b) && (c < d) && (a > 5000)) { ... }`
 - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction instead of having one branch for each
 - Predicates stored and operated on using condition registers
 - A single branch checks the value of the combined predicate
- + Fewer branches in code → fewer mipredictions/stalls
- Possibly unnecessary work
 - If the first predicate is false, no need to compute other predicates
- Condition registers exist in IBM RS6000 and the POWER architecture

Predicated Execution

- Idea: Convert control dependence to data dependence
- Suppose we had a Conditional Move instruction...
 - CMOV condition, $R1 \leftarrow R2$
 - $R1 = (\text{condition} == \text{true}) ? R2 : R1$
 - Employed in most modern ISAs (x86, Alpha)
- Code example with branches vs. CMOVs
if (a == 5) {b = 4;} else {b = 3;}

CMPEQ condition, a, 5;

CMOV condition, b \leftarrow 4;

CMOV !condition, b \leftarrow 3;

Conditional Execution in ARM

- Same as predicated execution
- Every instruction is conditionally executed

Predicated Execution

- Eliminates branches → enables straight line code (i.e., larger basic blocks in code)
- Advantages
 - Always-not-taken prediction works better (no branches)
 - Compiler has more freedom to optimize code (no branches)
 - control flow does not hinder inst. reordering optimizations
 - code optimizations hindered only by data dependencies
- Disadvantages
 - Useless work: some instructions fetched/executed but discarded (especially bad for easy-to-predict branches)
 - Requires additional ISA support
- Can we eliminate all branches this way?

Predicated Execution

- We will get back to this...
- Some readings (optional):
 - Allen et al., “Conversion of control dependence to data dependence,” POPL 1983.
 - Kim et al., “Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution,” MICRO 2005.