Computer Architecture (Spring 2020)

Pipelining

Dr. Duo Liu (刘铎)
Office: Main Building 0626
Email: liuduo@cqu.edu.cn

Data Dependence Types

Flow dependence

$$\begin{tikzcd} r_3 & \leftarrow & r_1 \text{ op } r_2 \\ r_5 & \leftarrow & r_3 \text{ op } r_4 \arrow[ull] \end{tikzcd}$$

Read-after-Write (RAW)

Anti dependence

$$\begin{tikzcd} r_3 & \leftarrow & r_1 \text{ op } r_2 \\ r_1 \arrow[urr] & \leftarrow & r_4 \text{ op } r_5 \end{tikzcd}$$

Write-after-Read (WAR)

Output-dependence

$$\begin{tikzcd} r_3 & \leftarrow & r_1 \text{ op } r_2 \\ r_5 & \leftarrow & r_3 \text{ op } r_4 \\ r_3 \arrow[uu, bend left] & \leftarrow & r_6 \text{ op } r_7 \end{tikzcd}$$

Write-after-Write (WAW)

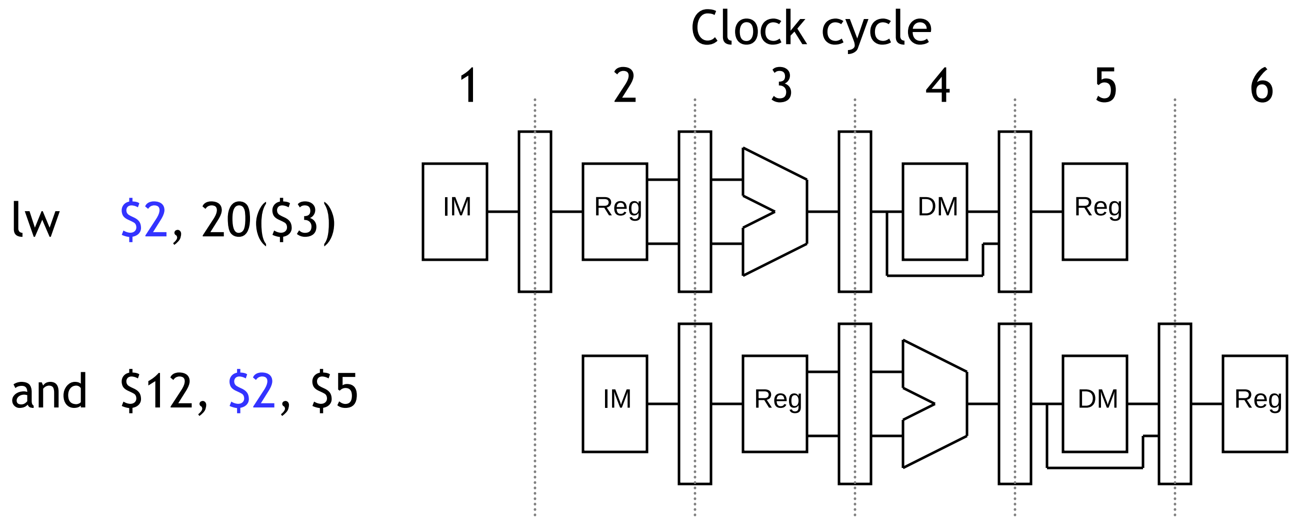
Data hazards in the pipeline diagram

		Clock cycle								
		1	2	3	4	5	6	7	8	9
sub	$2, $1, $3	IF	ID	EX	MEM	WB				
and	$12, $2, $5		IF	ID	EX	MEM	WB			
or	$13, $6, $2			IF	ID	EX	MEM	WB		
add	$14, $2, $2				IF	ID	EX	MEM	WB	
sw	$15, 100($2)					IF	ID	EX	MEM	WB

- The SUB instruction does not write to register $2 until clock cycle 5. This causes two data hazards in our current pipelined datapath.
  - The AND reads register $2 in cycle 3. Since SUB hasn’t modified the register yet, this will be the *old* value of $2, not the new one.
  - Similarly, the OR instruction uses register $2 in cycle 4, again before it’s actually updated by SUB.

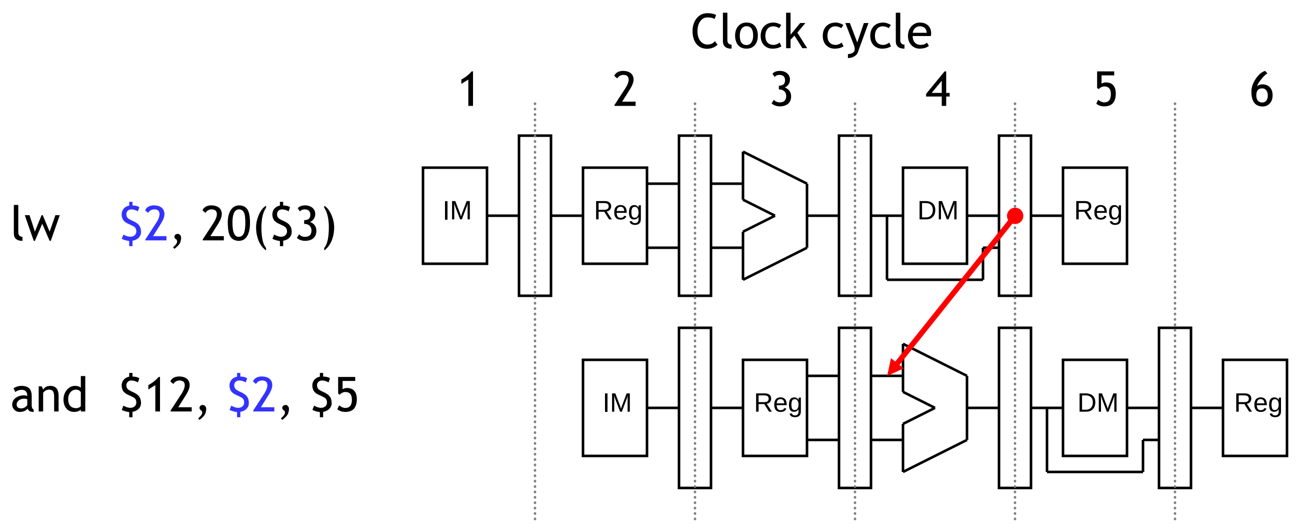
What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.
  - How does this change the data hazard?



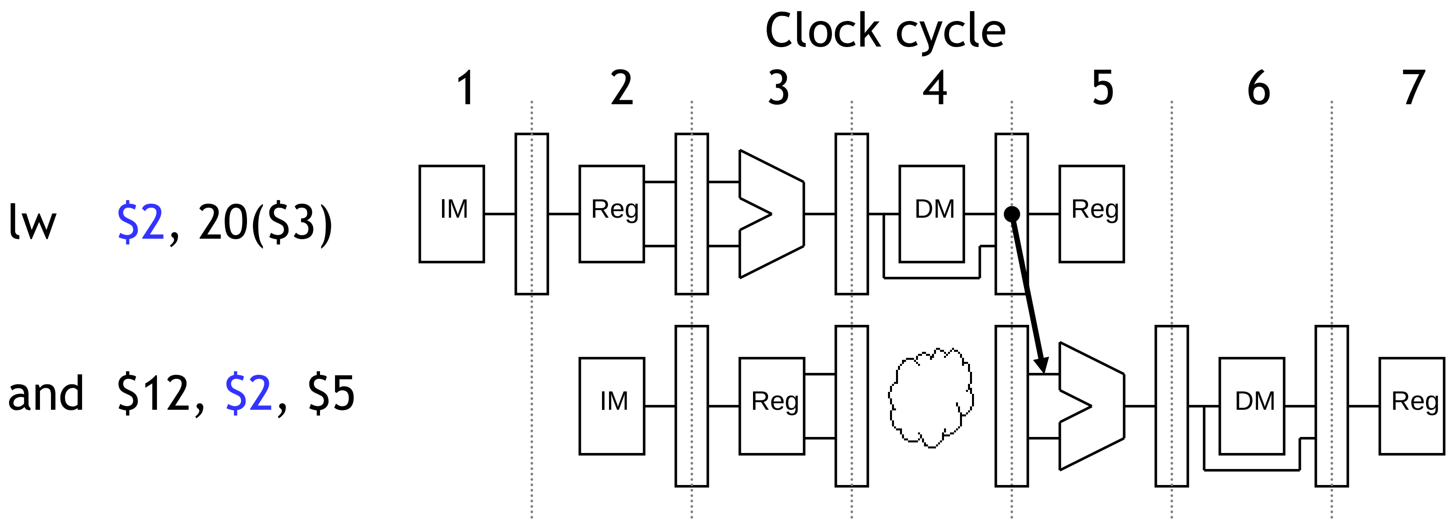
What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.
  - The load data doesn't come from memory until the *end* of cycle 4.
  - But the AND needs that value at the *beginning* of the same cycle!
- This is a "true" data hazard—the data is not available when we need it.



Stalling

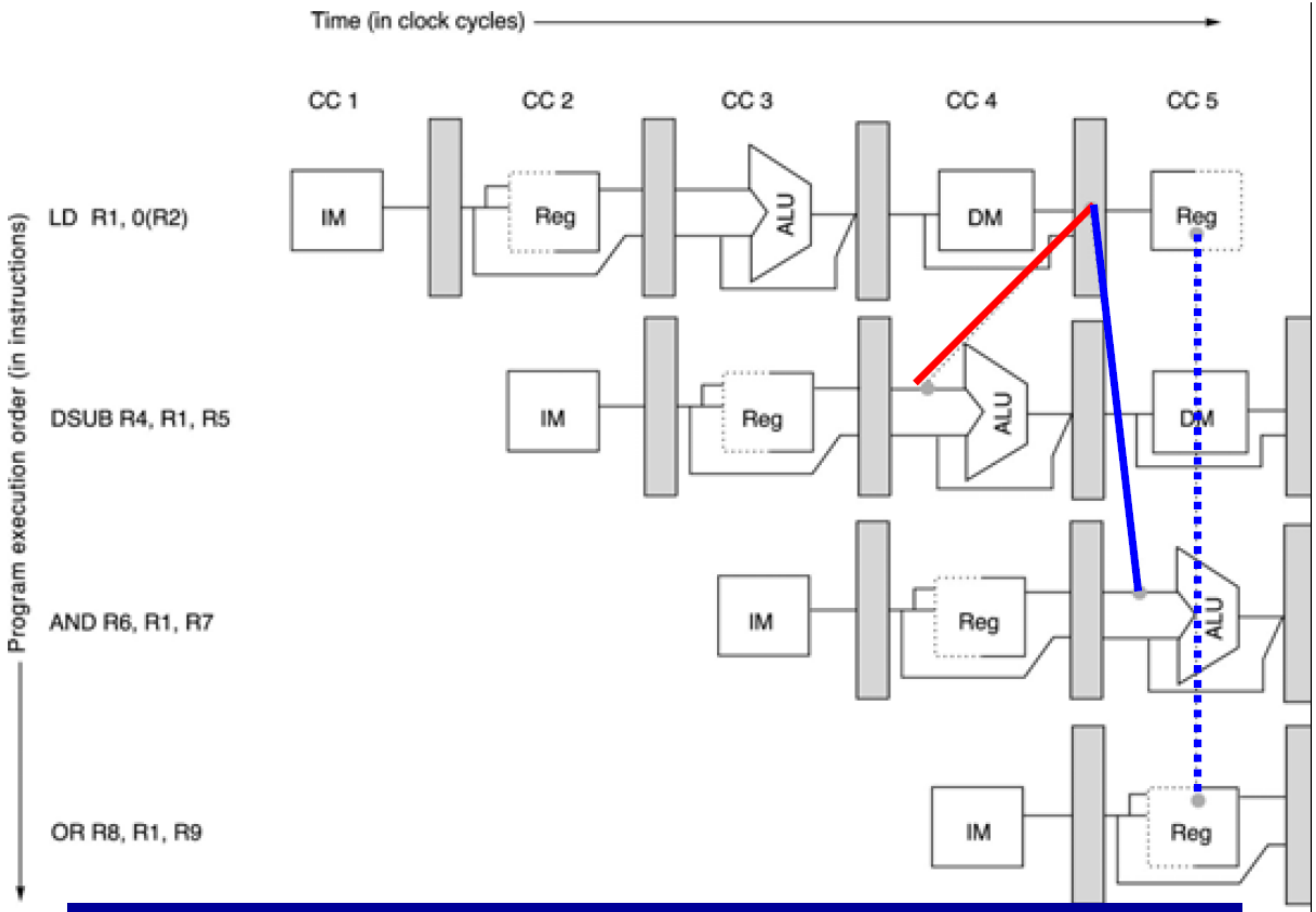
- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.



- Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.

Dealing with Data Hazards: Pipeline Interlocks

- …when forwarding is not possible…
  - because we can't operate backward in time!
- … we need a different HW solution
  - pipeline interlocks introduce stalls in the middle of an instruction

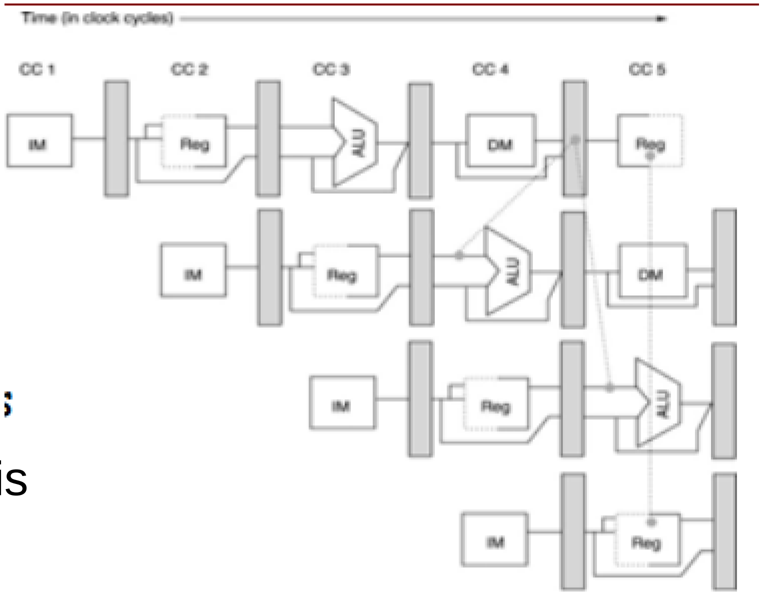


Interlocks are implemented using NOPs; a NOP is an instruction that does nothing. For instance `DADD R0, R0, R0`

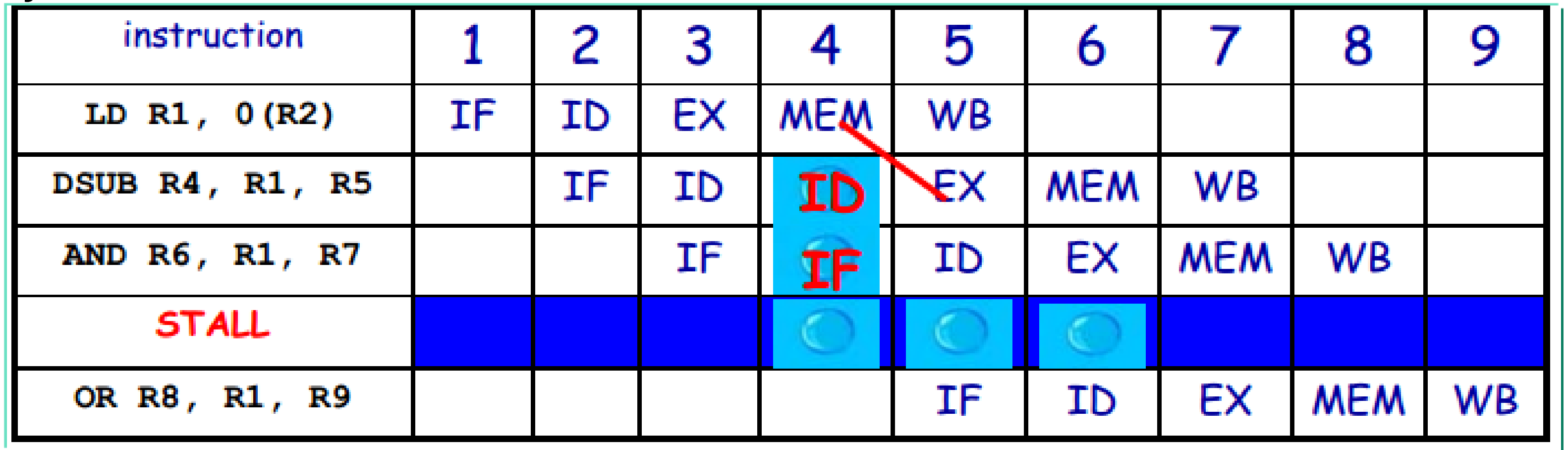
Pipeline Stalls due to Interlocks

At Cycle 4 the progression of **DSUB** (and all the following instructions) is halted

- no new instruction is fetched
- a NOP instruction is inserted into the EX stage and will exit the pipe at Cycle 6



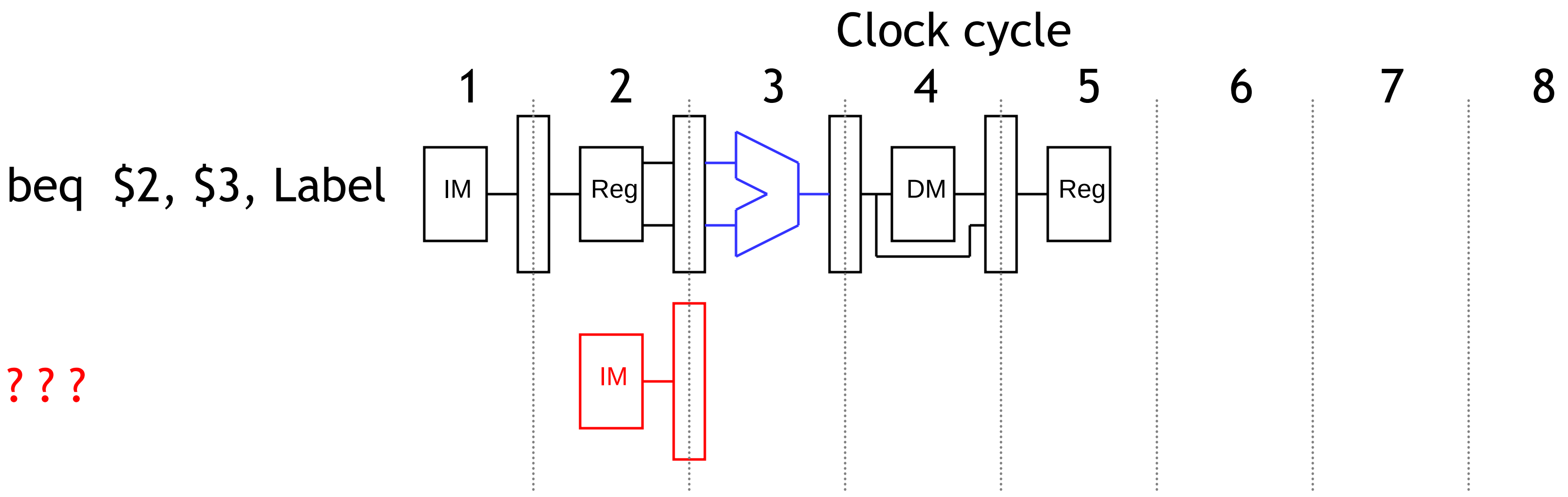
A NOP Instruction (a bubble) that is generated by one interlock activation increases the CPI of the stalled instruction by the length of one clock cycle



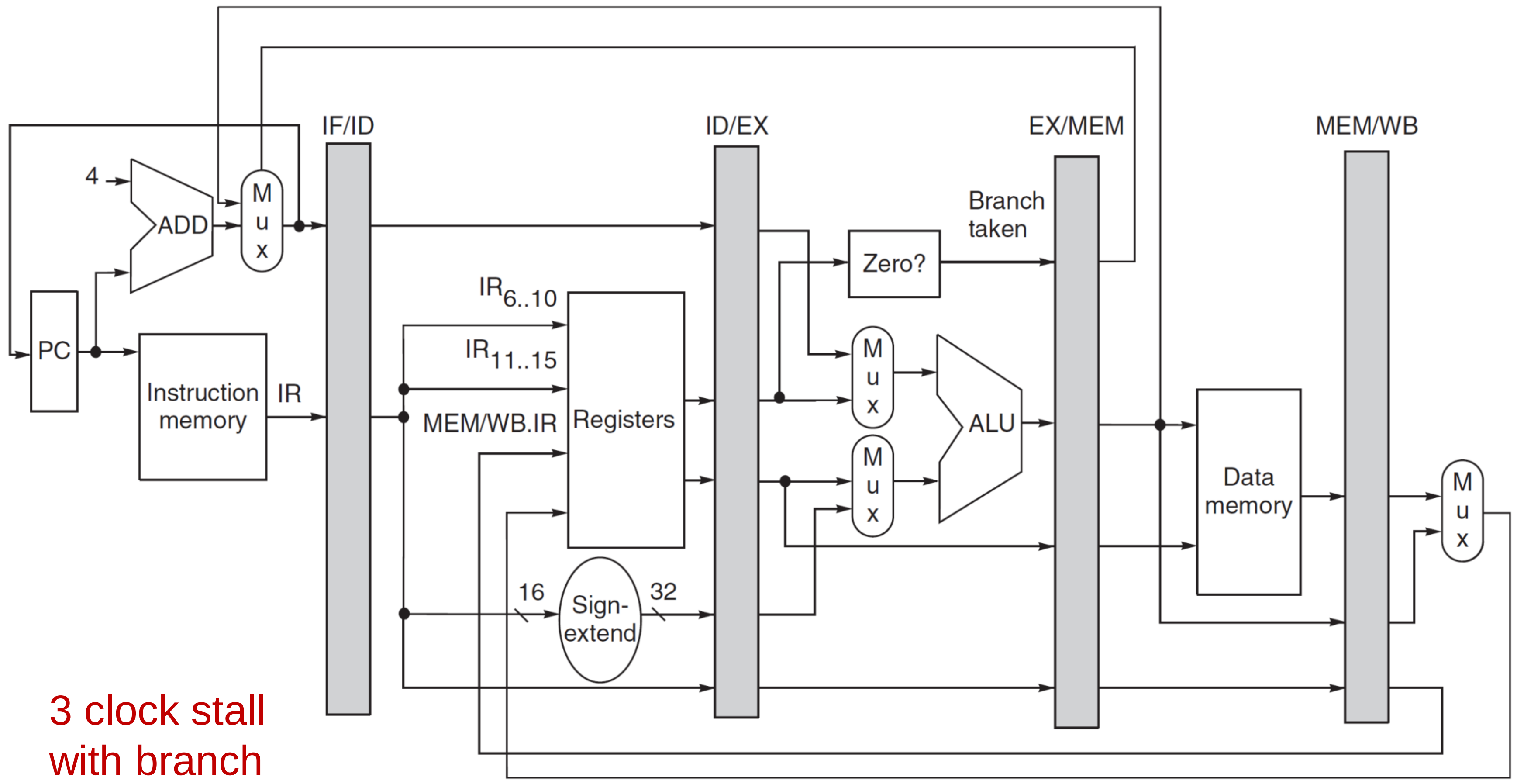
instruction	1	2	3	4	5	6	7	8	9
LD R1, 0(R2)	IF	ID	EX	MEM	WB				
DSUB R4, R1, R5		IF	ID	ID	EX	MEM	WB		
AND R6, R1, R7			IF	IF	ID	EX	MEM	WB	
STALL				○	○	○			
OR R8, R1, R9					IF	ID	EX	MEM	WB

Branches

- Most of the work for a branch computation is done in the EX stage.
  - The branch target address is computed.
  - The source registers are compared by the ALU, and the Zero flag is set or cleared accordingly.
- Thus, the branch decision cannot be made until the end of the EX stage.
  - But we need to know which instruction to fetch next, in order to keep the pipeline running!
  - This leads to what's called a control hazard.

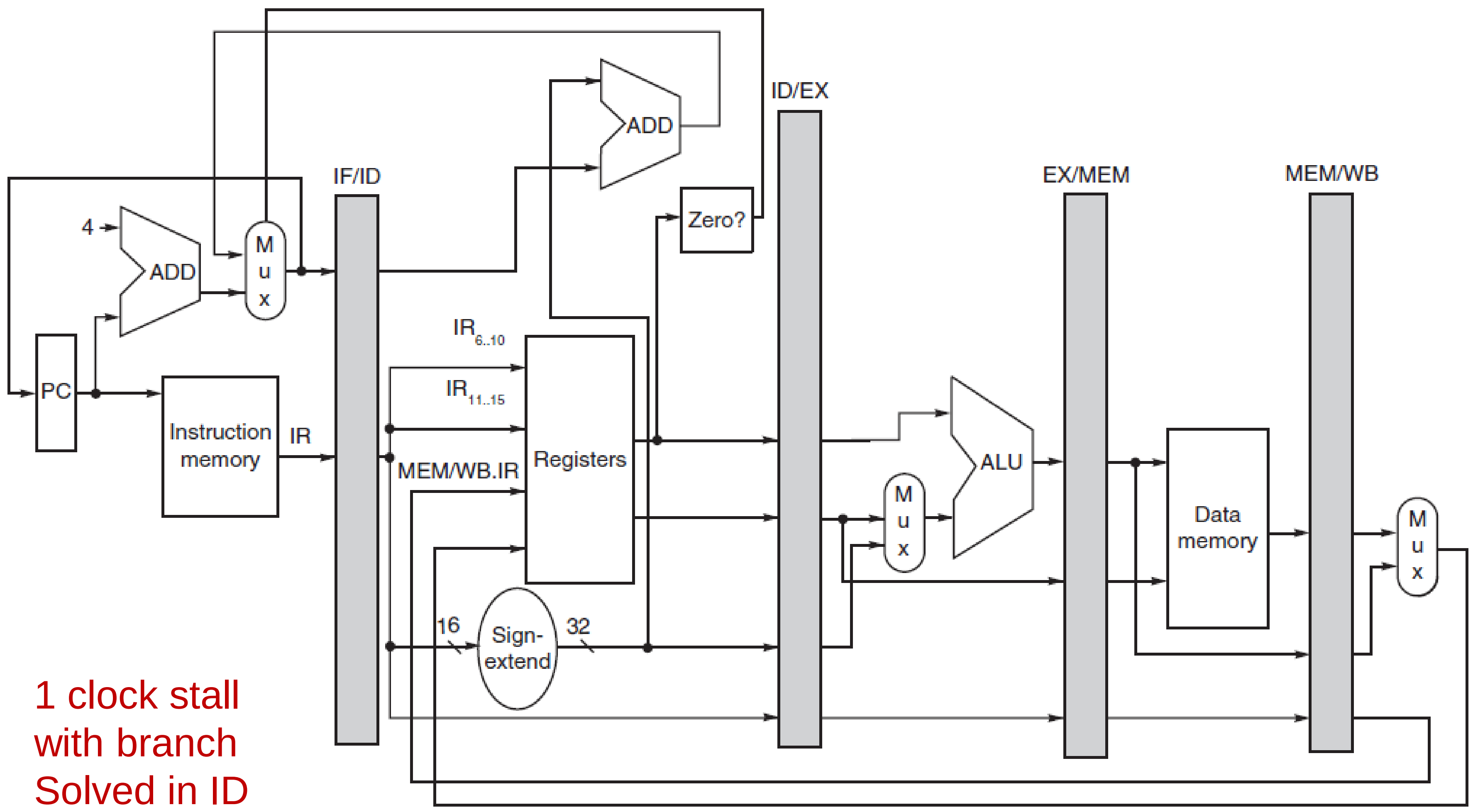


MIPS Pipeline Implementation



3 clock stall
with branch

MIPS Pipeline Implementation (revised)



1 clock stall
with branch
Solved in ID

Branch (Control) Hazards and Their Impact on the Pipeline Performance

Control hazards can cause a greater performance loss for a pipelined implementation than data hazards. In first approximation:

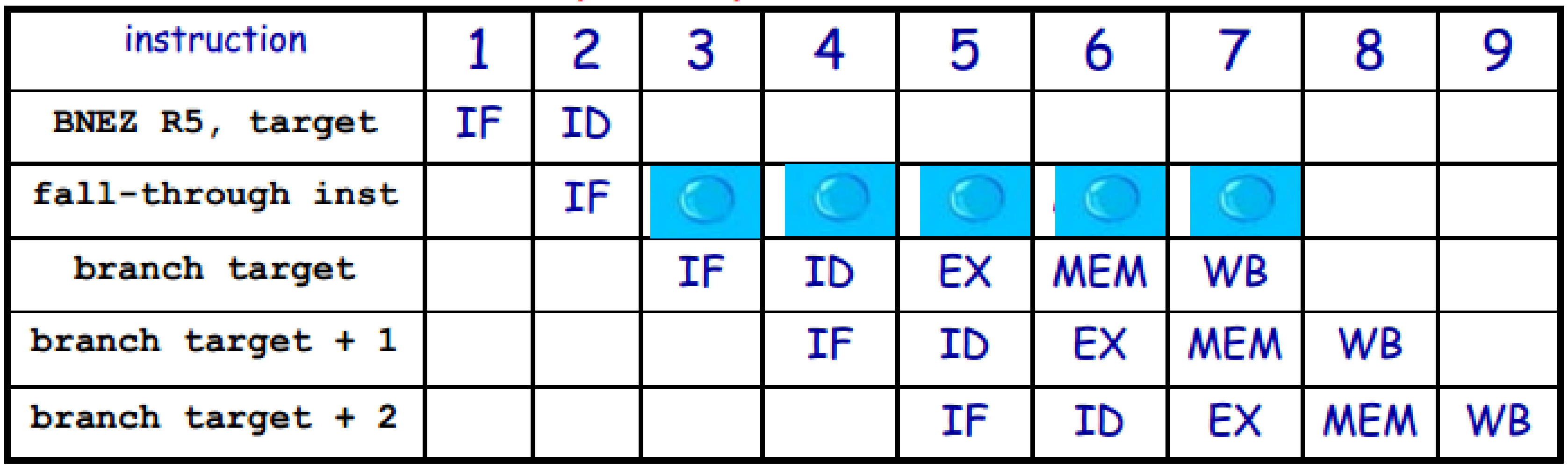
$$\text{speedupFromPipelining} = \frac{\text{pipelineDepth}}{1 + [\,\text{branchFrequency} \times \text{branchPenalty}\,]}$$

- performance loss can vary between 10% and 30% depending on the branch frequency
- generally, the deeper the pipeline, the worse the branch penalty (measured in clock cycles)

• Various strategies to reduce the branch penalty
  - static (compile time) schemes
    • they are fixed for each branch during the entire execution
  - dynamic
    • hardware and software techniques
    • more sophisticated branch prediction schemes

Reducing Pipeline Branch Penalties - I: Freezing (Flushing) the Pipeline

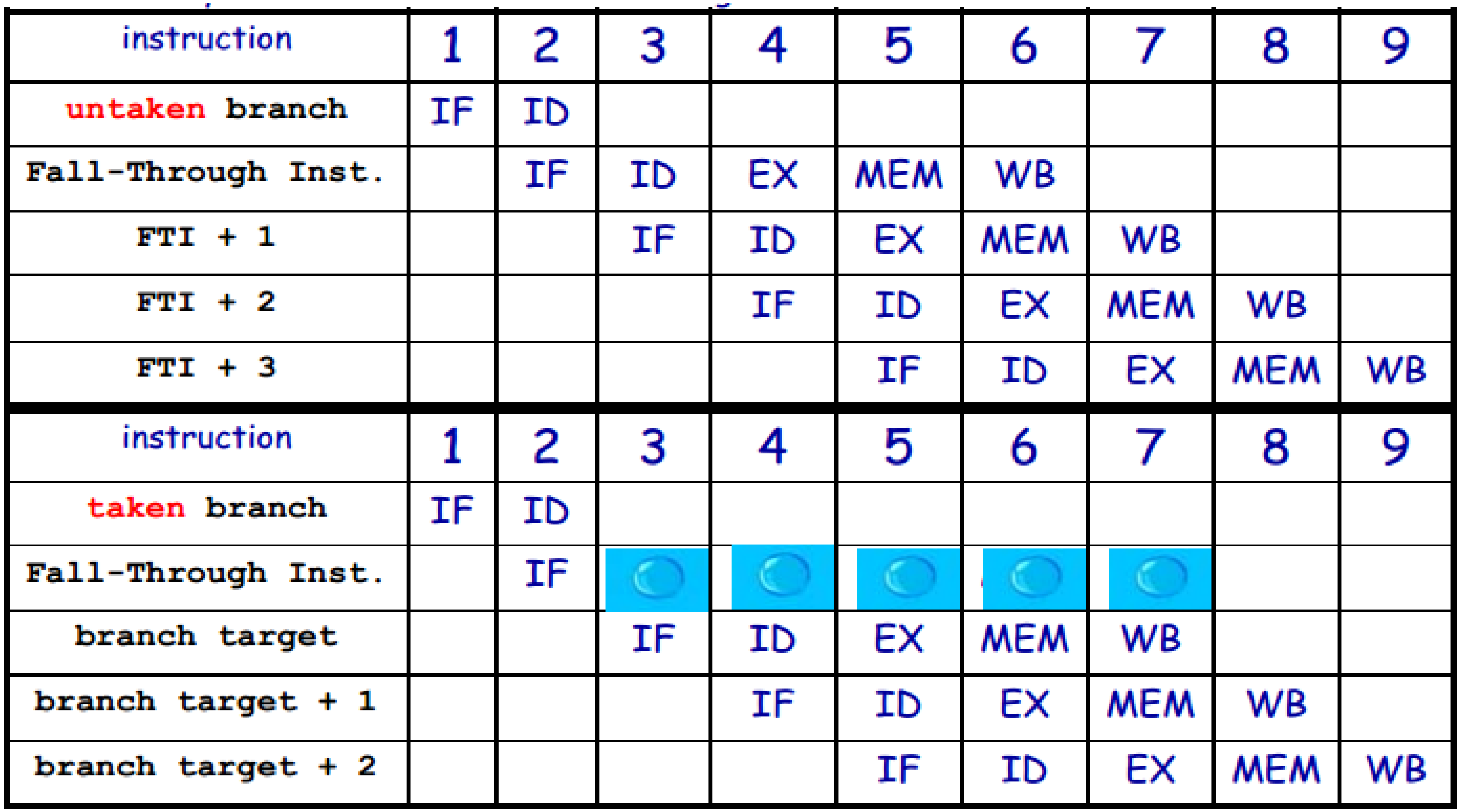
- Assuming now that the target address calculation and the comparison are performed during the ID stage, then, in case of “branch taken”, the PC is changed at the end of the ID stage
  - simplest strategy: freezing the pipeline as soon as the instruction is detected as a branch and regardless of its outcome (branch penalty is fixed, SW cannot reduce it)



instruction	1	2	3	4	5	6	7	8	9
BNEZ R5, target	IF	ID							
fall-through inst		IF	●	●	●	●	●		
branch target			IF	ID	EX	MEM	WB		
branch target + 1				IF	ID	EX	MEM	WB	
branch target + 2					IF	ID	EX	MEM	WB

Reducing Pipeline Branch Penalties - II: Predicting “Non-Taken”

If the branch is taken, all the fetched instruction must be turned into NOPs and it is necessary to restart the fetch at the target address. The state must be unaffected!!



instruction	1	2	3	4	5	6	7	8	9
untaken branch	IF	ID							
Fall-Through Inst.		IF	ID	EX	MEM	WB			
FTI + 1			IF	ID	EX	MEM	WB		
FTI + 2				IF	ID	EX	MEM	WB	
FTI + 3					IF	ID	EX	MEM	WB

instruction	1	2	3	4	5	6	7	8	9
taken branch	IF	ID							
Fall-Through Inst.		IF	●	●	●	●	●		
branch target			IF	ID	EX	MEM	WB		
branch target + 1				IF	ID	EX	MEM	WB	
branch target + 2					IF	ID	EX	MEM	WB

Reducing Pipeline Branch Penalties – III: Branch Delaying

The instruction in the delay slot is executed whether or not the branch is taken

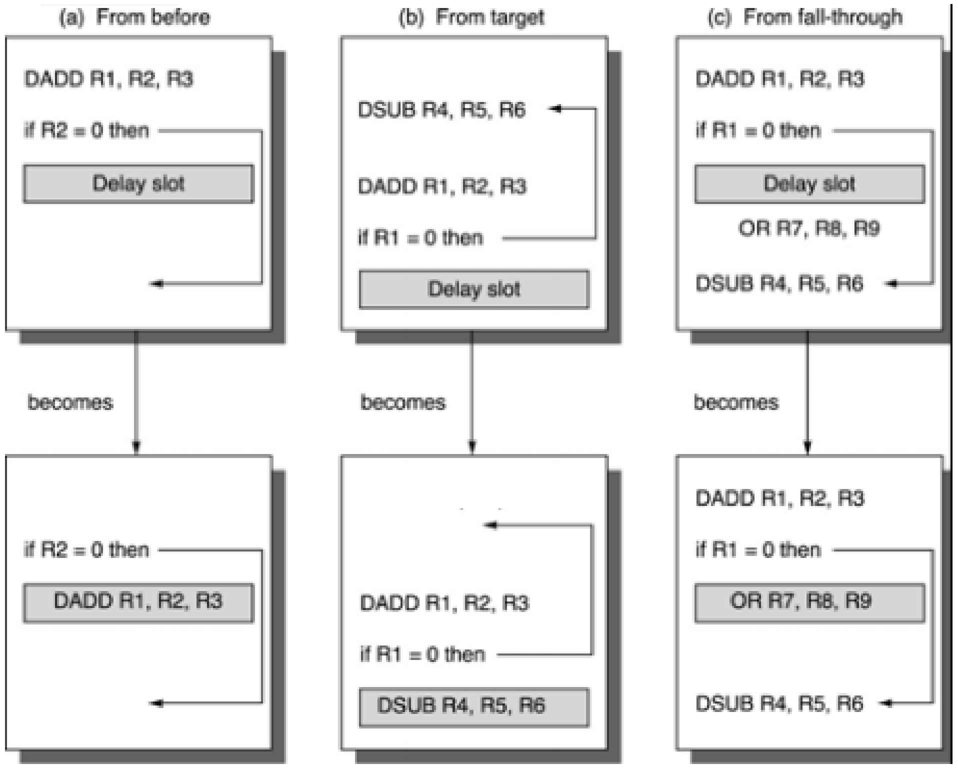
Scheduling problem for the compiler

Three strategies to fill the delay slot

a) always best choice if possible

b) preferred when branch is taken with high probability
   – e.g., loop branches

c) dual case as (b)



Strategies (b) and (c) can be applied only if it is ok to execute the moved instruction if the prediction is wrong (i.e. the processor state is not affected)

Summary

- Three kinds of hazards conspire to make pipelining difficult.
- Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- Data hazards can occur when instructions need to access registers that haven’t been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a “true” hazard, which must stall the pipeline.
- Control hazards arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.

Example

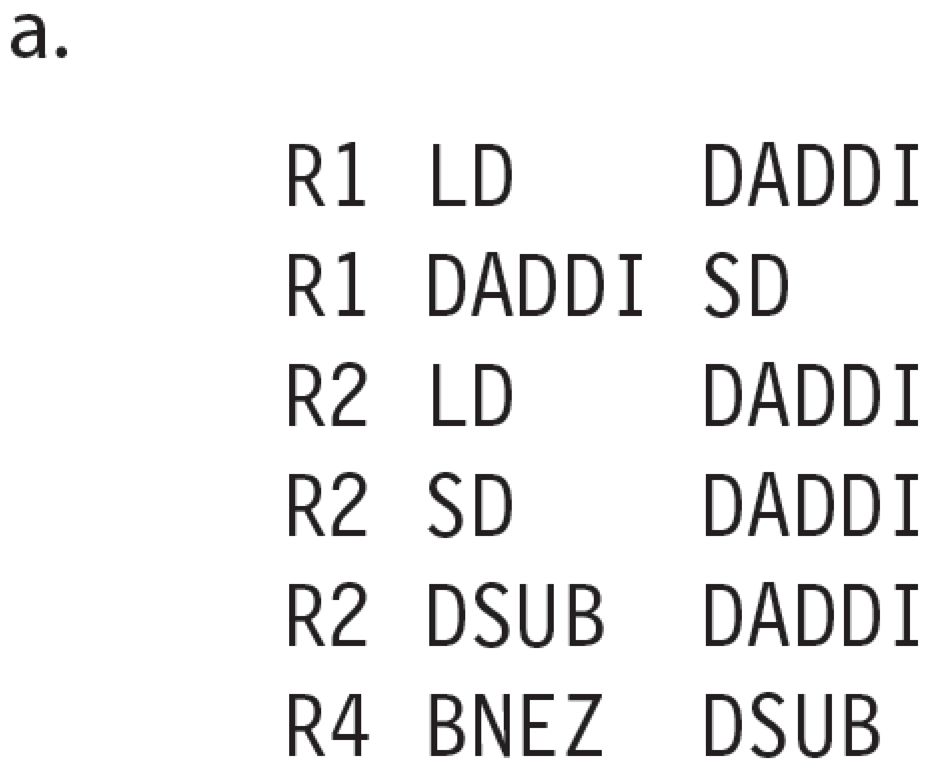
C.1 [15/15/15/15/25/10/15] <A.2> Use the following code fragment:

```
Loop:    LD       R1,0(R2)       ;load R1 from address 0+R2
         DADDI    R1,R1,#1       ;R1=R1+1
         SD       R1,0,(R2)      ;store R1 at address 0+R2
         DADDI    R2,R2,#4       ;R2=R2+4
         DSUB     R4,R3,R2       ;R4=R3-R2
         BNEZ     R4,Loop        ;branch to Loop if R4!=0
```

Assume that the initial value of R3 is R2 + 396.

a. [15] <C.2> Data hazards are caused by data dependences in the code. Whether a dependency causes a hazard depends on the machine implementation (i.e., number of pipeline stages). List all of the data dependences in the code above. Record the register, source instruction, and destination instruction; for example, there is a data dependency for register R1 from the LD to the DADDI.

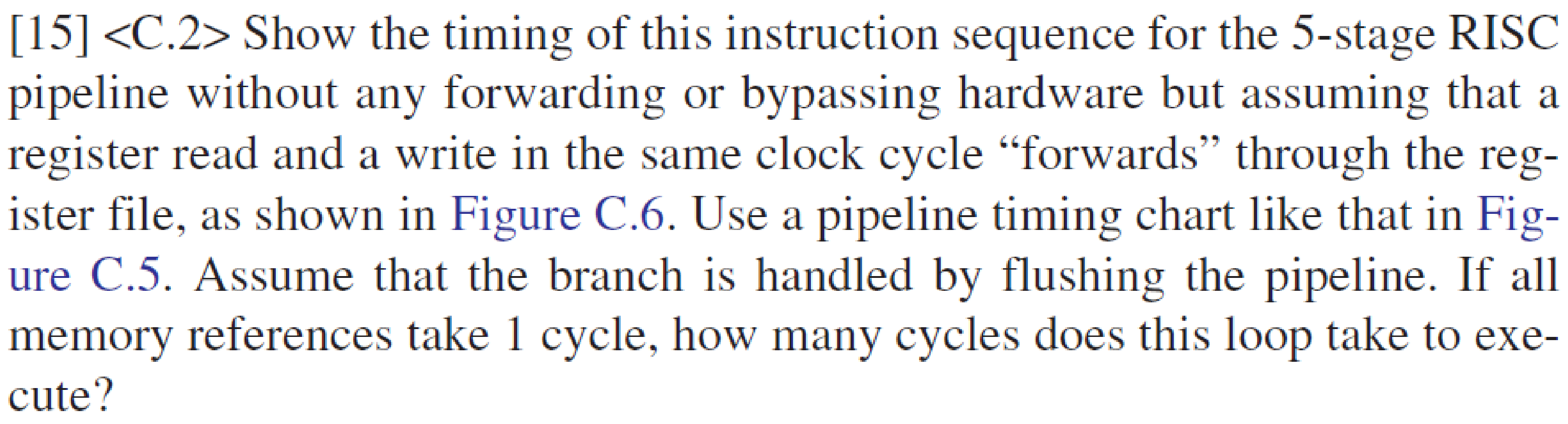
Example



a.

R1	LD	DADDI
R1	DADDI	SD
R2	LD	DADDI
R2	SD	DADDI
R2	DSUB	DADDI
R4	BNEZ	DSUB

Example



b. [15] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline without any forwarding or bypassing hardware but assuming that a register read and a write in the same clock cycle "forwards" through the register file, as shown in Figure C.6. Use a pipeline timing chart like that in Figure C.5. Assume that the branch is handled by flushing the pipeline. If all memory references take 1 cycle, how many cycles does this loop take to execute?

Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`LD R1, 0(R2)`	F	D	X	M	W													
`DADDI R1, R1, #1`		F	s	s	D	X	M	W										
SD R1, 0(R2)					F	s	s	D	X	M	W							
`DADDI R2, R2, #4`								F	D	X	M	W						
`DSUB R4, R3, R2`									F	s	s	D	X	M	W			
`BNEZ R4, Loop`												F	s	s	D	X	M	W
`LD R1, 0(R2)`																	F	D

Since the initial value of R3 is R2 + 396 and equal instance of the loop adds 4 to R2, the total number of iterations is 99. Notice that there are 8 cycles lost to RAW hazards including the branch instruction. Two cycles are lost after the branch because of the instruction flushing. It takes 16 cycles between loop instances; the total number of cycles is $98 \times 16 + 18 = 1584$. The last loop takes two addition cycles since this latency cannot be overlapped with additional loop instances.

Example

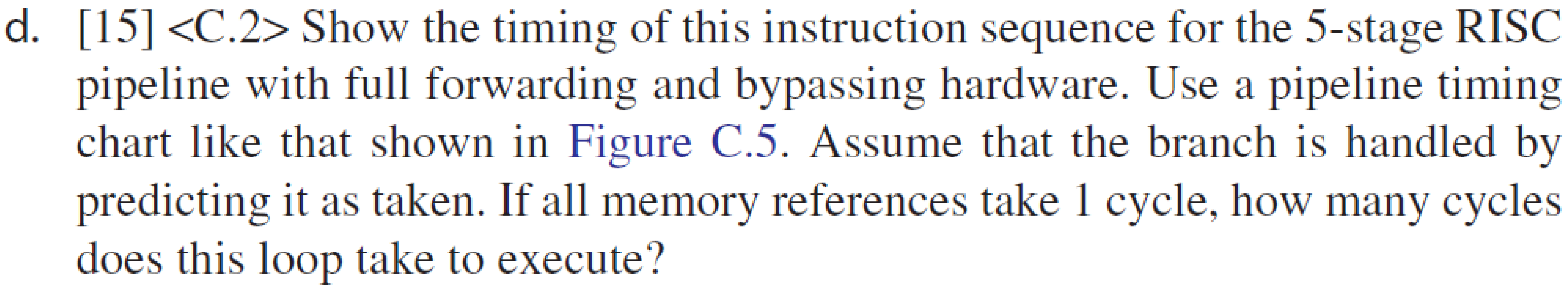
c. [15] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline with full forwarding and bypassing hardware. Use a pipeline timing chart like that shown in Figure C.5. Assume that the branch is handled by predicting it as not taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?

Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`LD R1, 0(R2)`	F	D	X	M	W													
`DADDI R1, R1, #1`		F	D	s	X	M	W											
`SD R1, 0(R2)`			F	s	D	X	M	W										
`DADDI R2, R2, #4`					F	D	X	M	W									
`DSUB R4, R3, R2`						F	D	X	M	W								
`BNEZ R4, Loop`							F	s	D	X	M	W						
(incorrect instruction)									F	s	s	s	s					
`LD R1, 0(R2)`										F	D	X	M	W				

Again we have 99 iterations. There are two RAW stalls and a flush after the branch since the branch is taken. The total number of cycles is $9 \times 98 + 12 = 894$. The last loop takes three addition cycles since this latency cannot be overlapped with additional loop instances.

Example



d. [15] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline with full forwarding and bypassing hardware. Use a pipeline timing chart like that shown in Figure C.5. Assume that the branch is handled by predicting it as taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?

Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`LD R1, 0(R2)`	F	D	X	M	W													
`DADDI R1, R1, #1`		F	D	s	X	M	W											
`SD R1, 0(R2)`			F	s	D	X	M	W										
`DADDI R2, R2, #4`					F	D	X	M	W									
`DSUB R4, R3, R2`						F	D	X	M	W								
`BNEZ R4, Loop`							F	s	D	X	M	W						
`LD R1, 0(R2)`									F	D	X	M	W					

Again we have 99 iterations. We still experience two RAW stalls, but since we correctly predict the branch, we do not need to flush after the branch. Thus, we have only $8 \times 98 + 12 = 796$.