

Lecture 11: Pipelining Hazards

- Today
 - Review pipeline hazards
 - Data hazards
 - Eliminating them with forwarding
 - Memory hazards
 - Load delay slot, stalling
 - Control hazards
 - Branch delay slot, branch prediction

Pipeline Hazards

- Data hazards

- an instruction uses the result of a previous instruction (RAW)

ADD R1, R2, R3 or SW R1, 3(R2)
ADD R4, R1, R5 LW R3, 3(R2)



- Control hazards

- the location of an instruction depends on a previous instruction

JMP LOOP

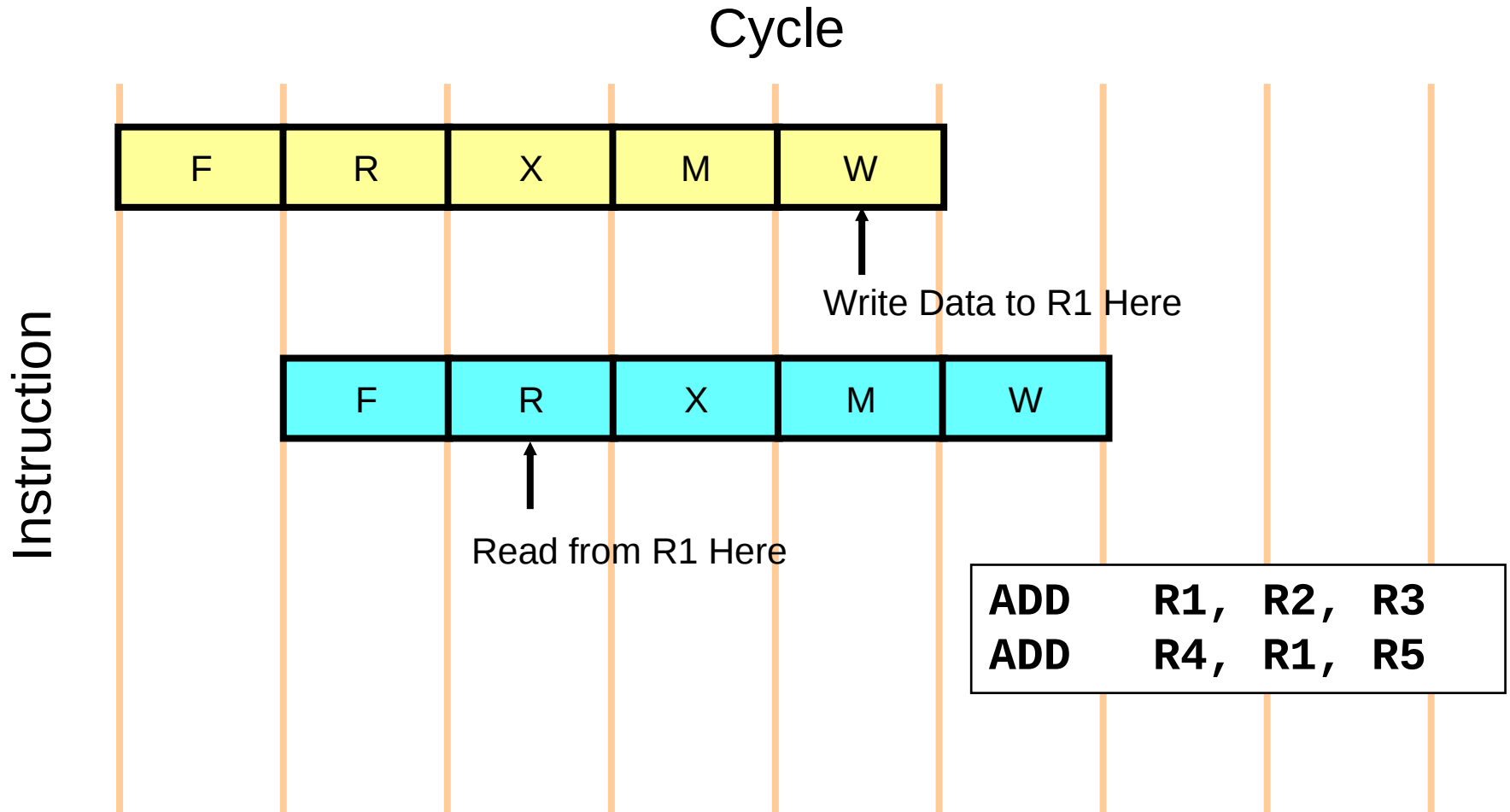
...

LOOP: ADD R1, R2, R3

- Structural hazards

- two instructions need access to the same resource
 - e.g., single memory shared for instruction fetch and load/store
 - collision in reservation table

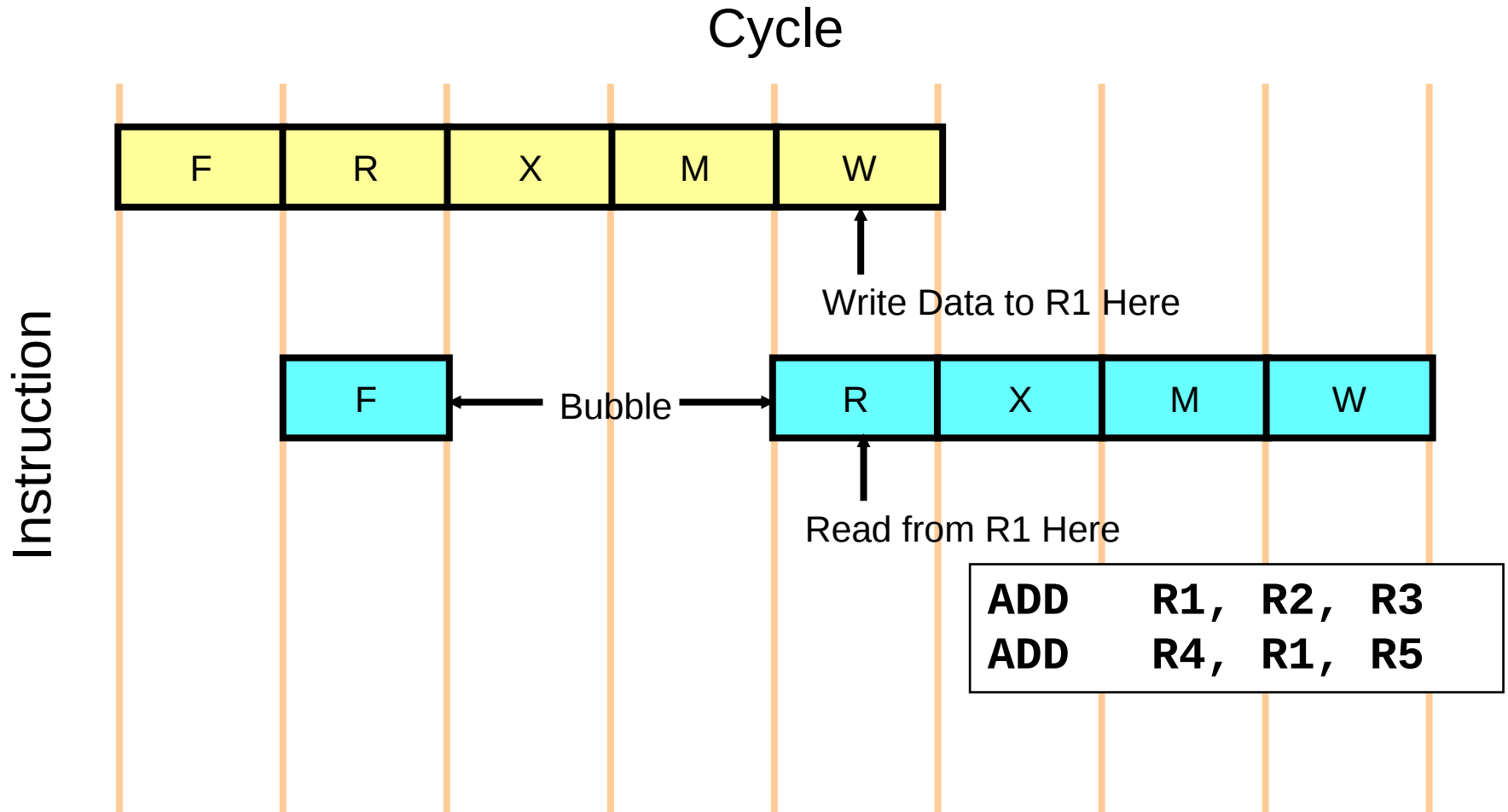
Data Hazards (RAW)



Resolving Hazards: Pipeline Stalls

- Can resolve any type of hazard
 - data, control, or structural
- Detect the hazard
- Freeze the pipeline up to the dependent stage until the hazard is resolved

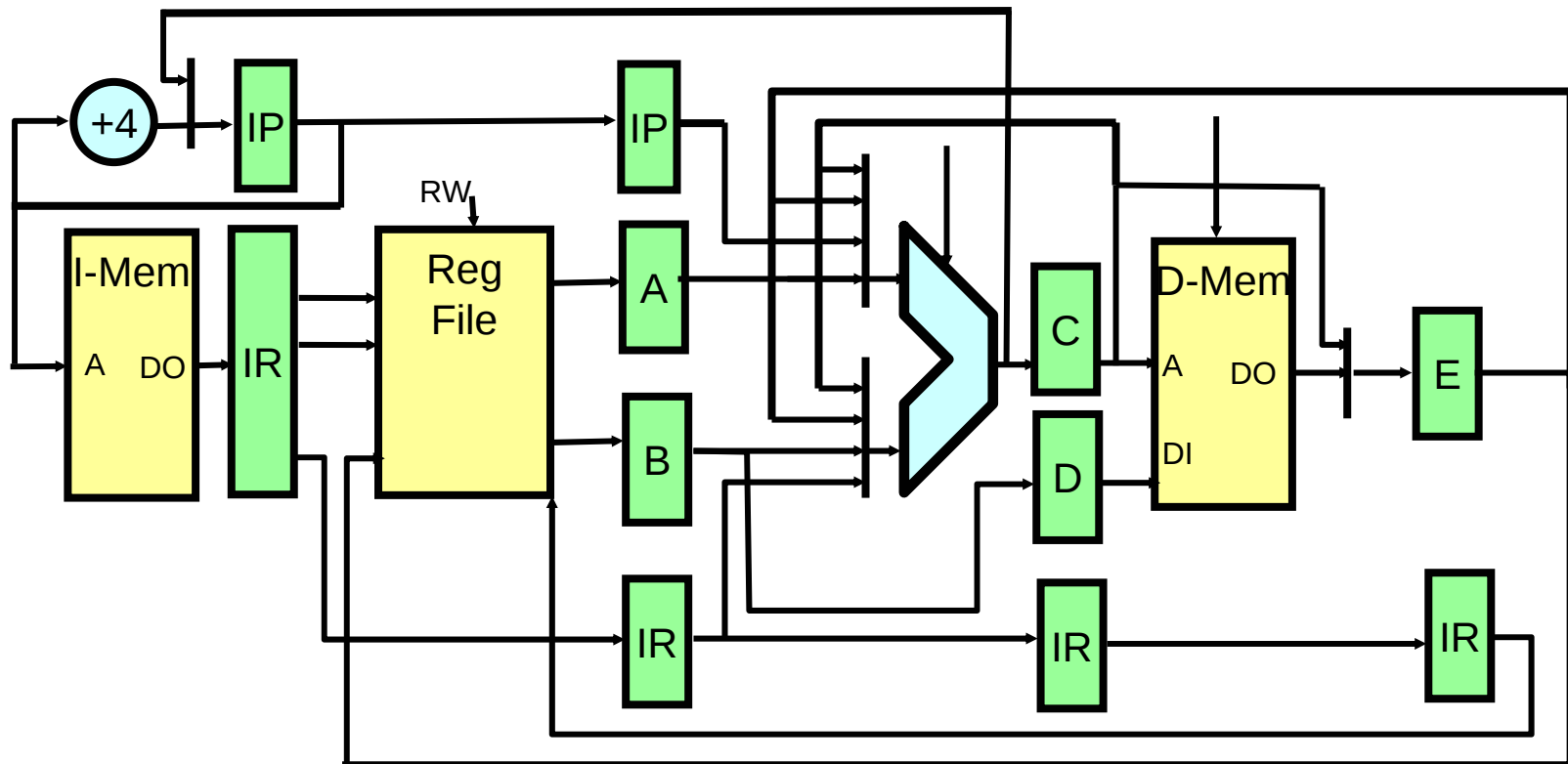
Example Pipeline Stall (Diagram)



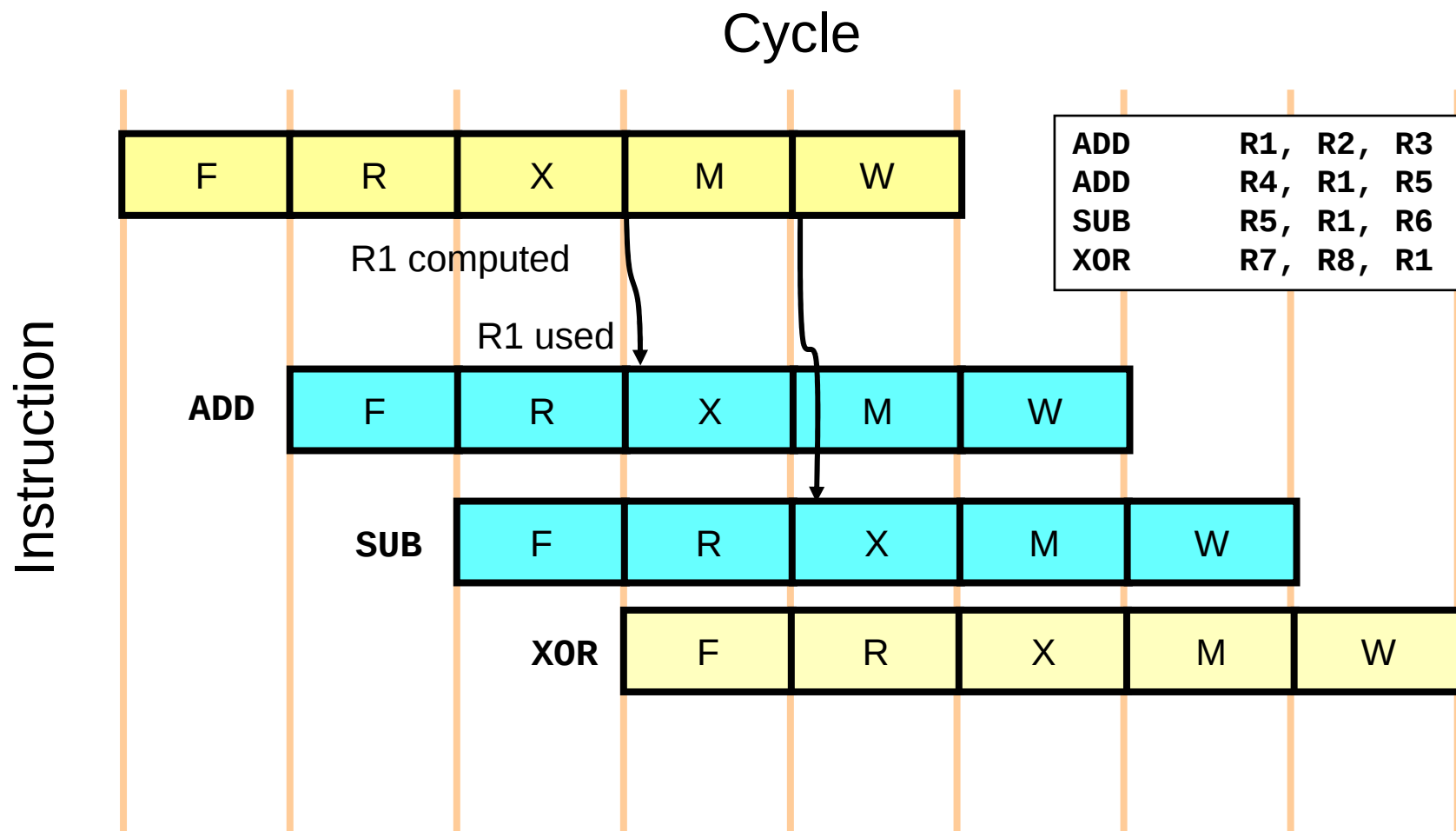
Resolving Hazards: Bypass (Forwarding)

- If data is available elsewhere in the pipeline, there is no need to stall
- Detect condition
- Bypass (or forward) data directly to the consuming pipeline stage
- Bypass eliminates stalls for *single-cycle* operations
 - reduces longest stall to N-1 cycles for N-cycle operations

Simple Pipeline with Bypass Multiplexers

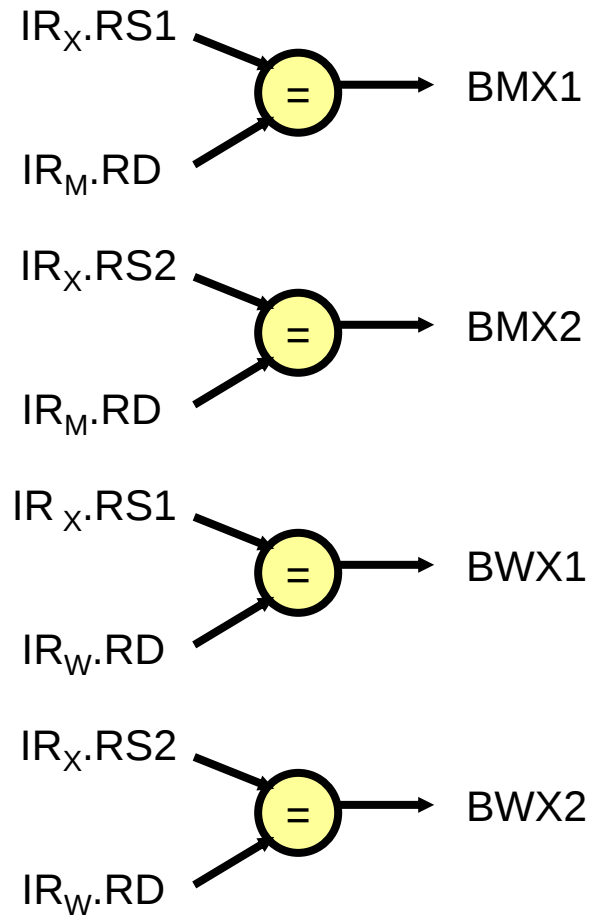


Data Hazards With Bypassing



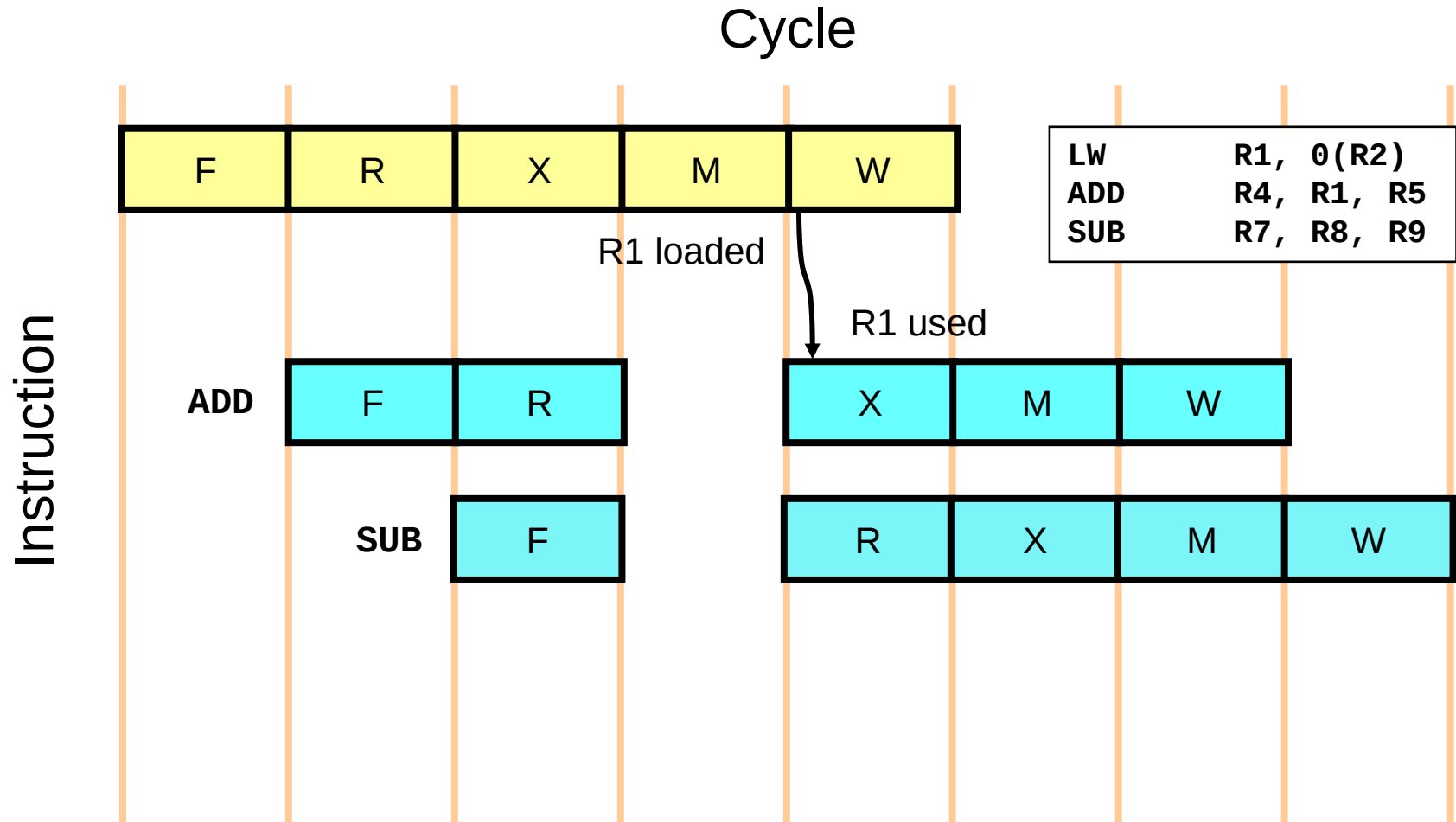
Control of Bypass (show with next figure)

- Compare source register fields of IR_X to destination register fields of IR_M and IR_W .
- If match and fields *active*, enable appropriate bypass path

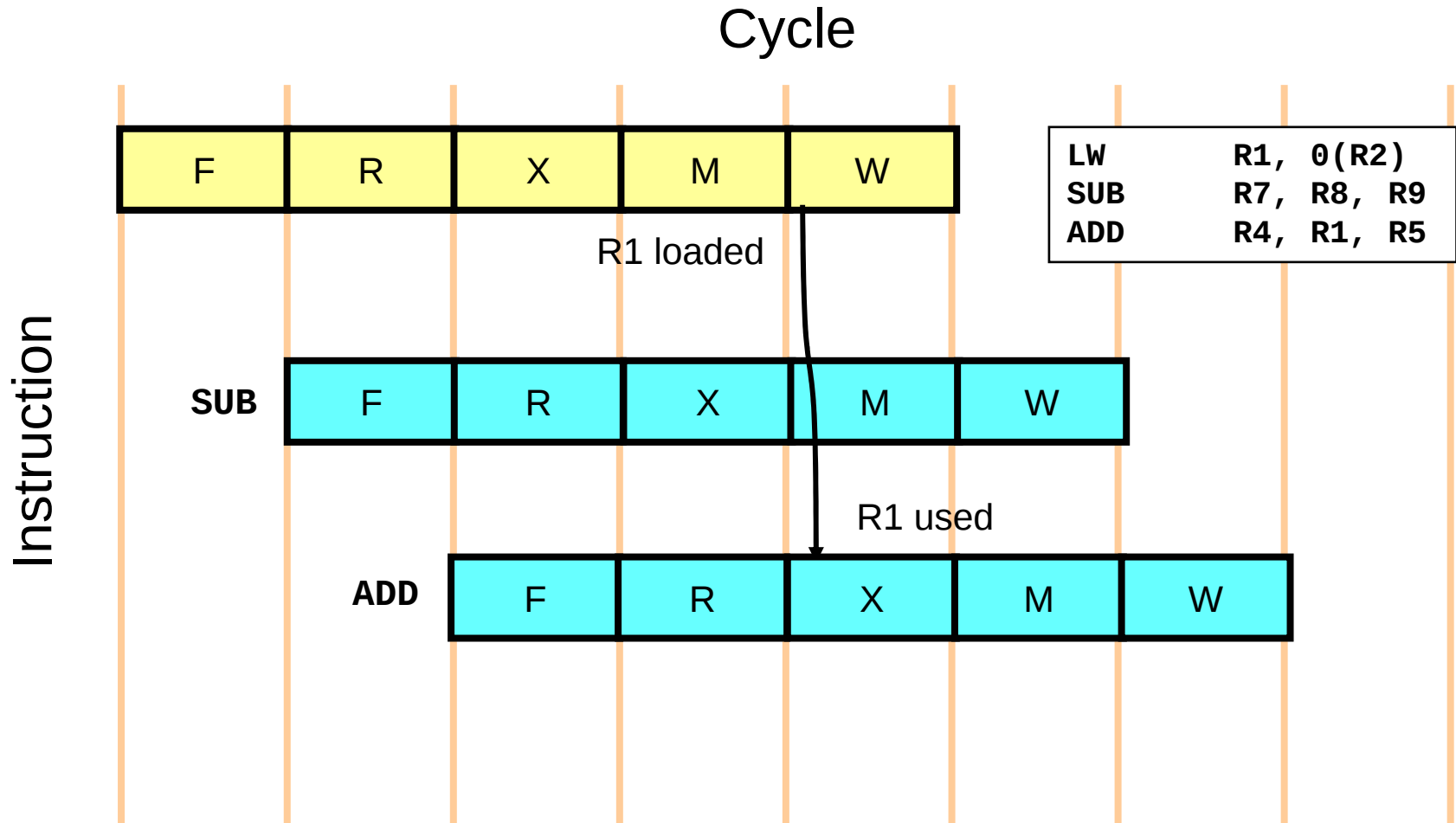


Figures 6.40 and 6.41 from text

Memory Data Hazards



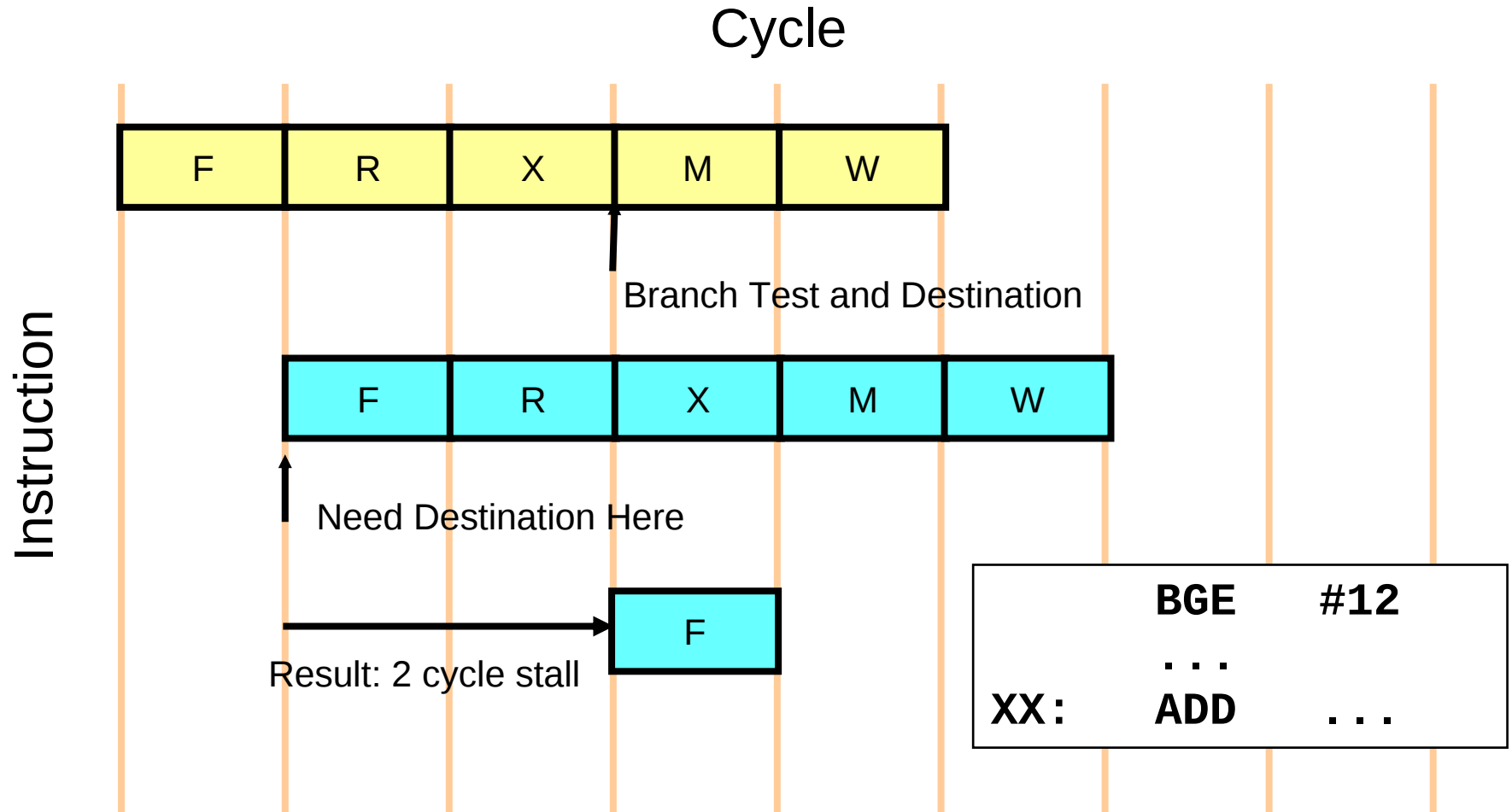
Instruction Scheduling (Load Delay Slots)



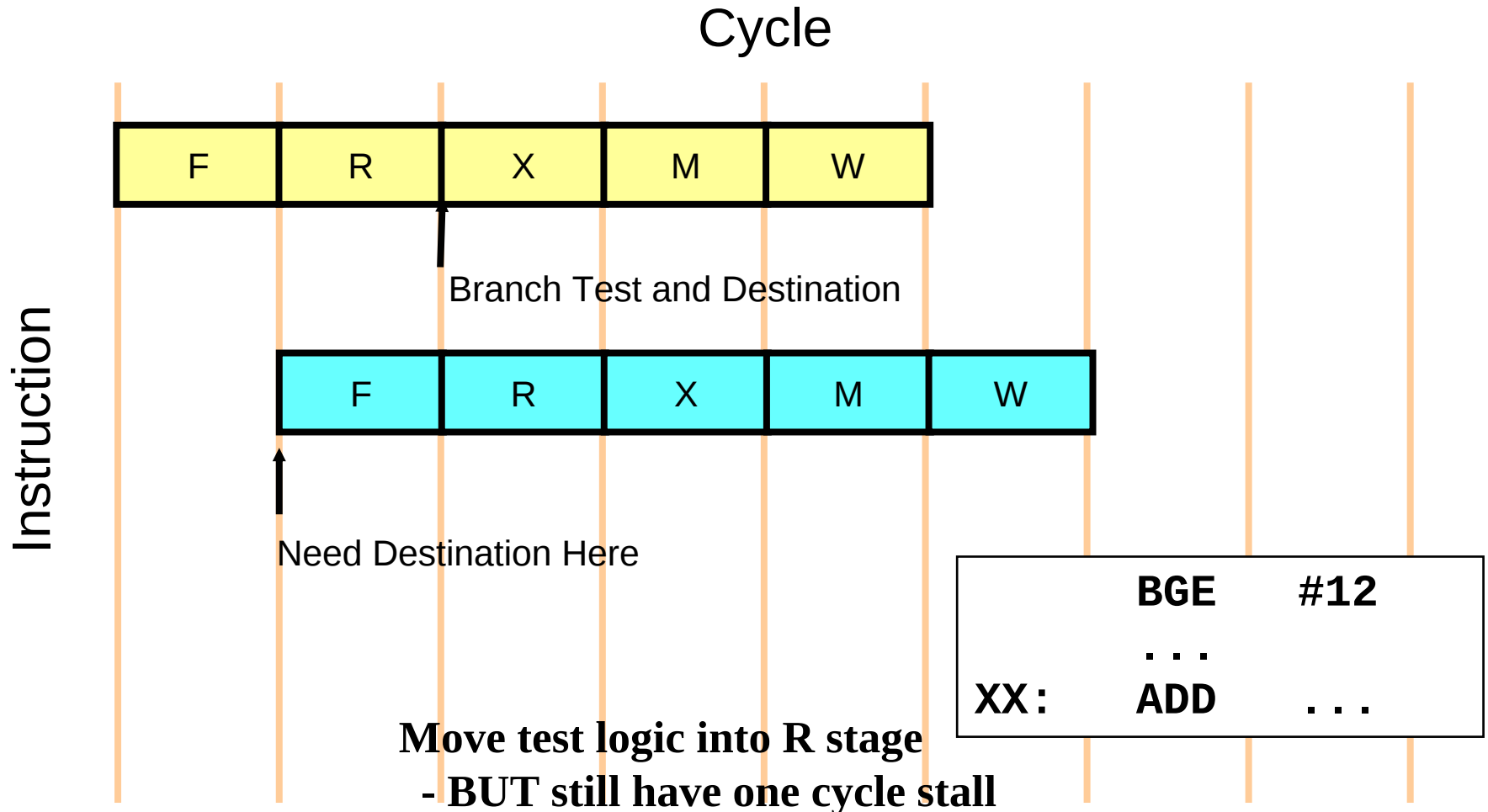
Figures 6.46 to 6.48 from text

(Control for load stall; and an example)

Control Hazards (Branch on condition)



Reducing Control Hazards



Figures 6.51, 6.52

Branch Delay Slots

- Since we need to have a dead cycle anyway, let's put a useful instruction there
- Advantage:
 - Do more useful work
 - Potentially get rid of all stalls
- Disadvantage:
 - Exposes microarchitecture to ISA
 - Deeper pipelines require more delay slots

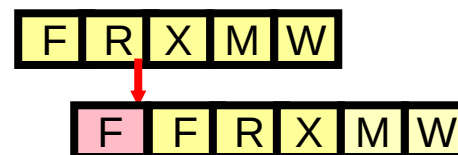
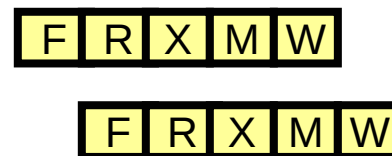
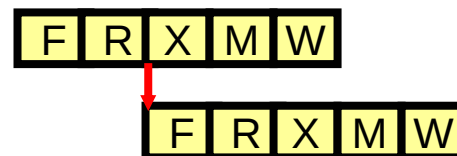
```
ADD R2, R3, R4  
BNEZ R5, _loop  
NOP
```



```
BNEZ R5, _loop  
ADD R2, R3, R4
```

Speculating for control hazards

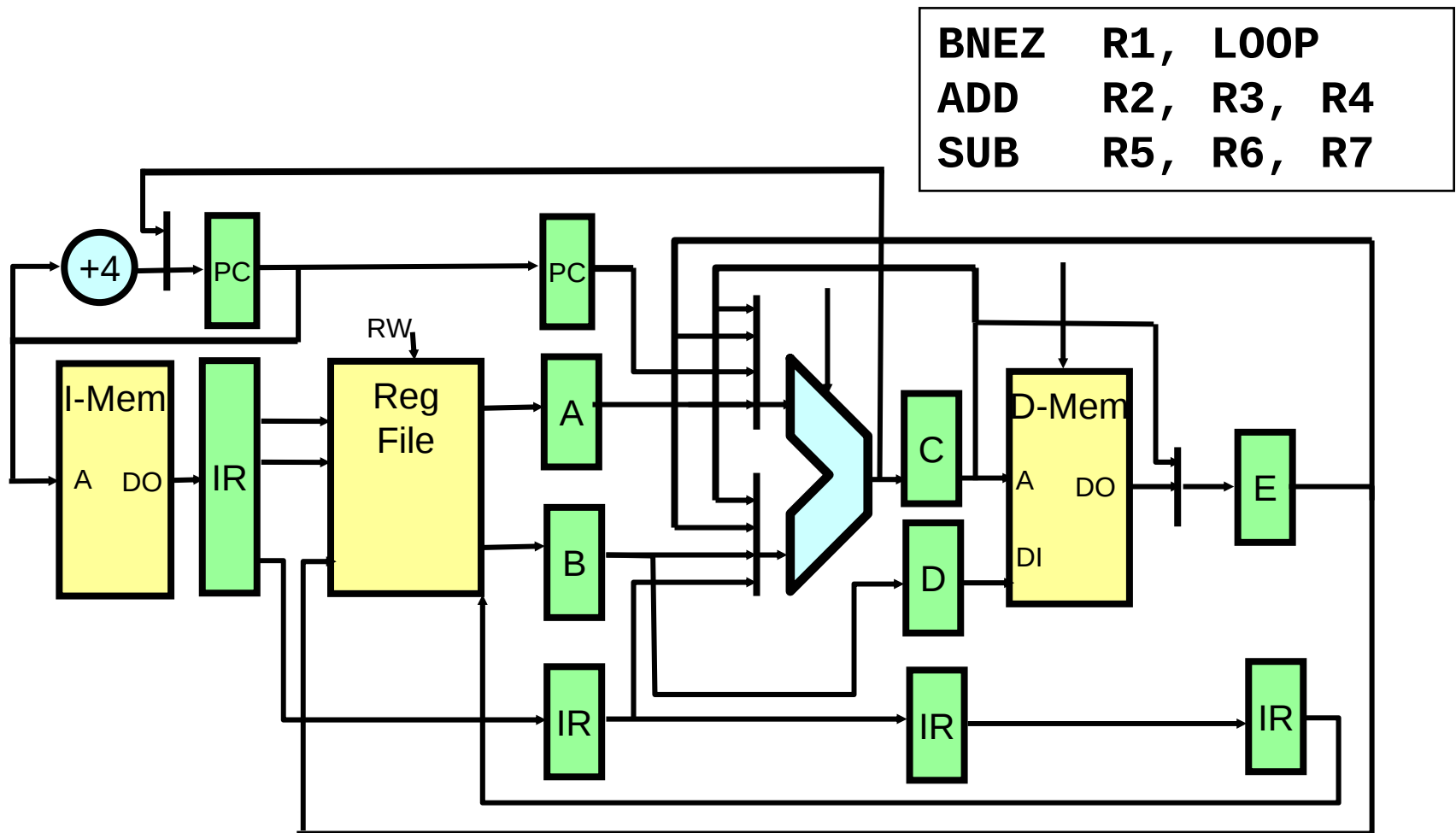
- Conservatively, the pipeline waits until the branch target is computed before fetching the next instruction.
- Alternatively, we can speculate which direction and to what address the branch will go.
- Need to confirm speculation and back up later.



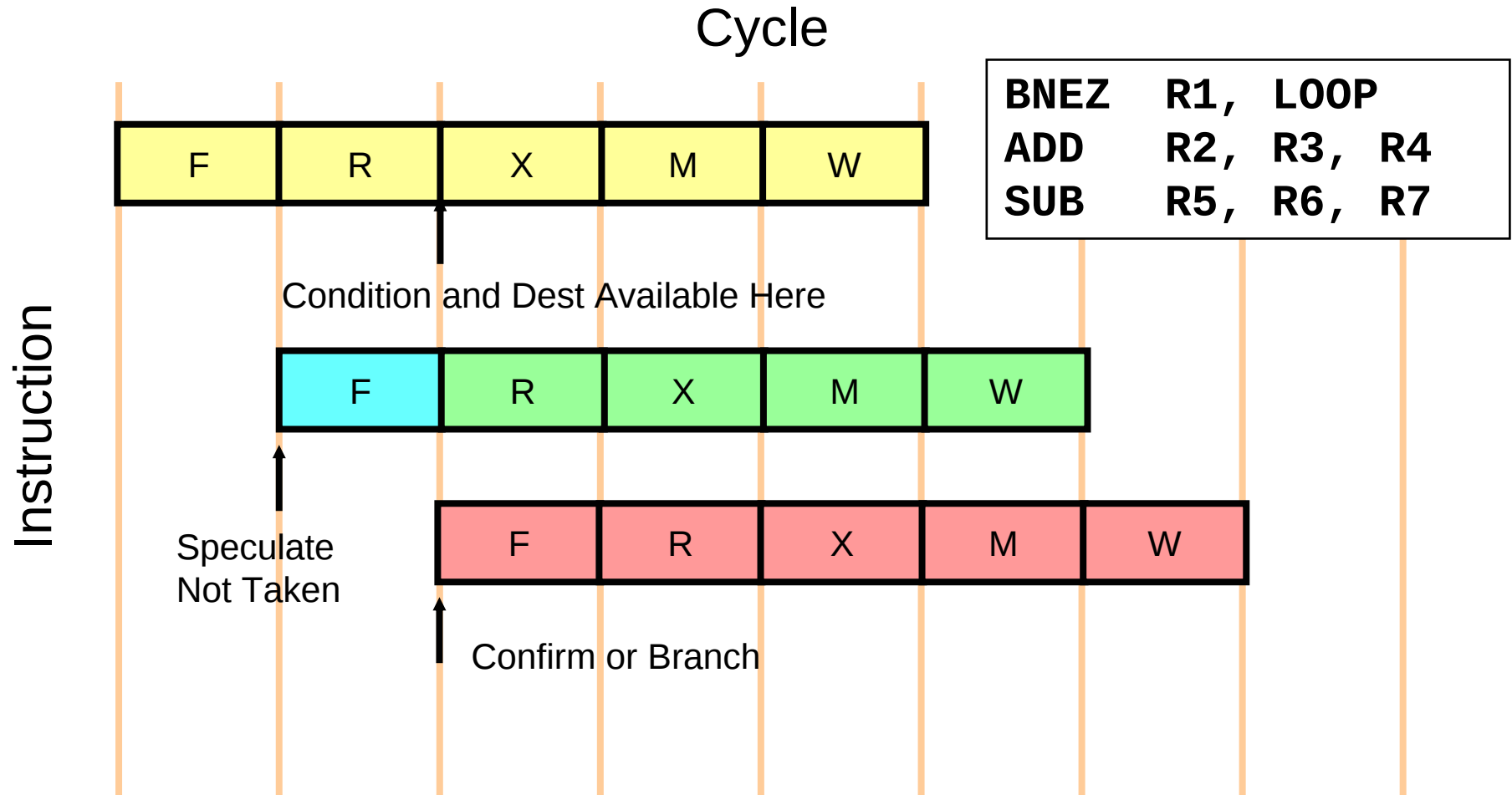
Predict Not Taken

Untaken Branch	F	R	X	M	W				
i+1		F	R	X	M	W			
i+2			F	R	X	M	W		
i+3				F	R	X	M	W	
i+4					F	R	X	M	W
Taken Branch	F	R	X	M	W				
i+1		F	?	?	?	?			
Branch target			F	R	X	M	W		
b+1				F	R	X	M	W	
b+2					F	R	X	M	W

Example Speculative Conditional Branch



Speculative Conditional Branch (Diagram)



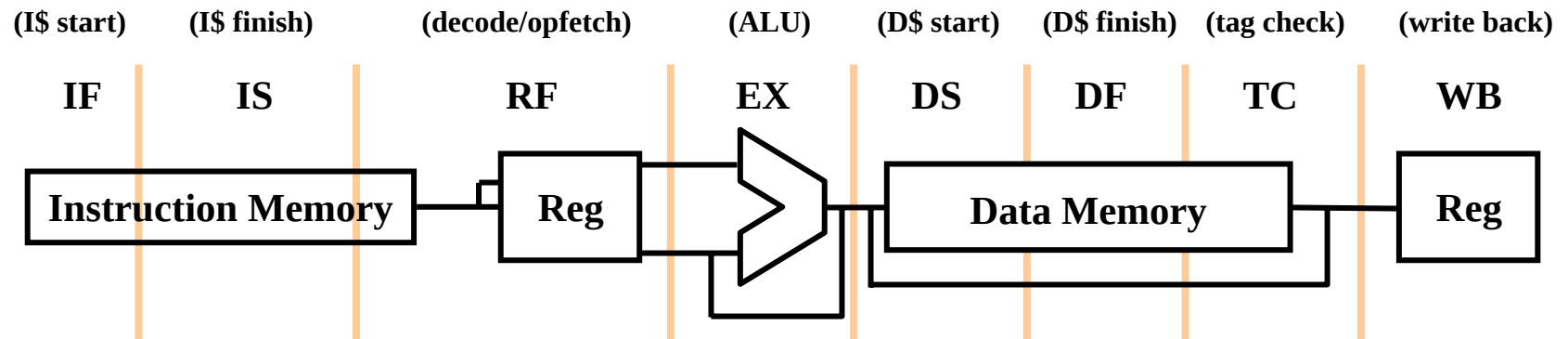
Control Hazards Summary

- Three approaches
 - Stall until new PC is known
 - Speculate that branch goes a particular way
 - If guess is right, great!
 - If guess is wrong, kill off speculated work
 - Delay slot
- Delay slot is only approach visible to programmer!
 - Unfortunately, MIPS picked this approach!

Exceptions – implicit conditional branches

- Examples of exceptions
 - Overflow of result
 - Page fault on load
- On an exception, branch to some address
- But – no explicit branch instruction!
- Architectural issues:
 - What exceptions are supported?
 - Are they “precise”?
 - i.e. behavior is as-if there was no pipelining
 - Adds significant complexity to implementation!

R4000 Pipeline



- How long is load delay?
- How long is branch delay?
- How many comparators are needed to implement the forwarding decisions?
- What instruction sequences will still cause stalls?

How Do We Speed up the Pipeline?

- Pipeline too long $\Rightarrow$ more ALUs (exploit ILP)
- WAR/WAW hazards $\Rightarrow$ register renaming

ADD R1, R2, R3
SUB R1, R4, R5 $\Rightarrow$ **ADD R1, R2, R3**
SUB R1', R4, R5

- Undetermined dependencies at compile time $\Rightarrow$ dynamic scheduling
 - Object code compatibility
 - Simplify compiler
- Too many branches $\Rightarrow$ better branch prediction
 - Or use predication to eliminate branches
- Unknown dependencies (control/data) $\Rightarrow$ speculate
- Explicitly parallel architectures (EPIC)

Summary

- Hazard detection and avoidance
- Improving Pipeline performance
- Next Time
 - Homework #3 is due!
 - Reading assignment: P&H 7.1 – 7.3