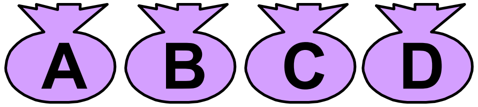
Pipelining

Readings: 4.5-4.8

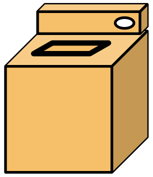
Example: Doing the laundry

Ann, Brian, Cathy, & Dave

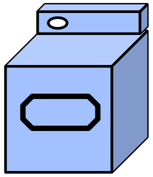


each have one load of clothes to wash, dry, and fold

Washer takes 30 minutes

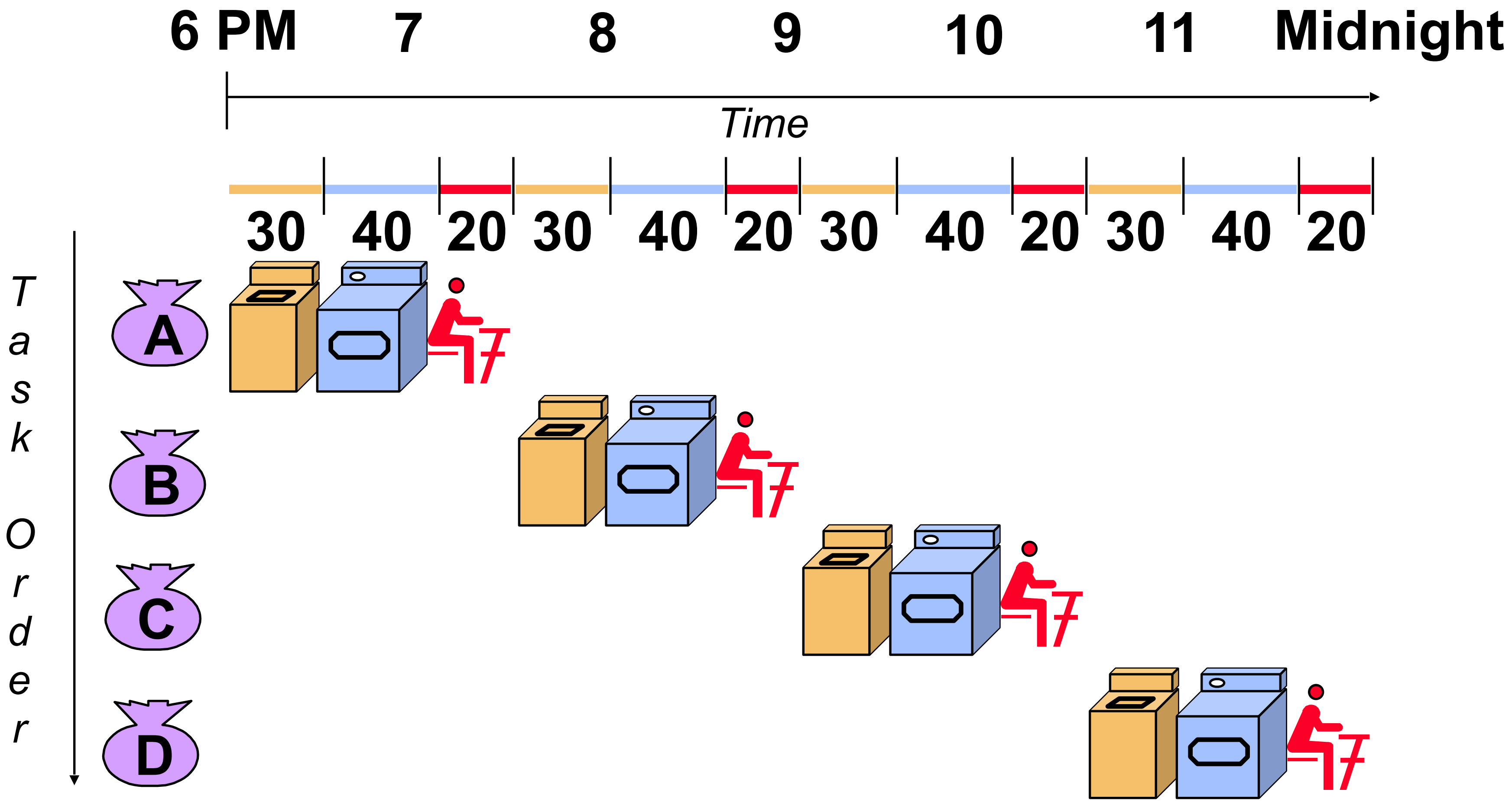


Dryer takes 40 minutes



“Folder” takes 20 minutes

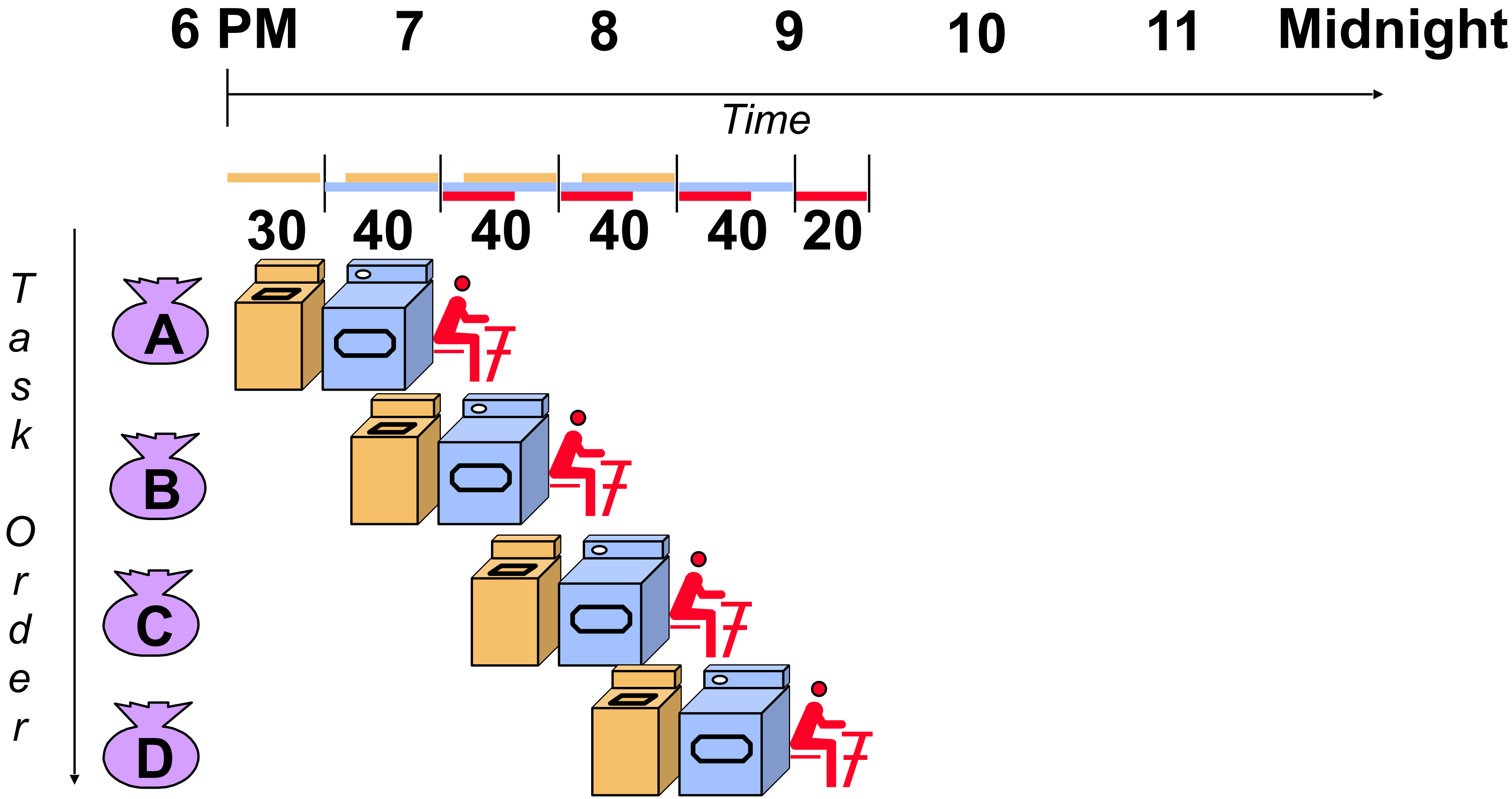
Sequential Laundry



Sequential laundry takes 6 hours for 4 loads

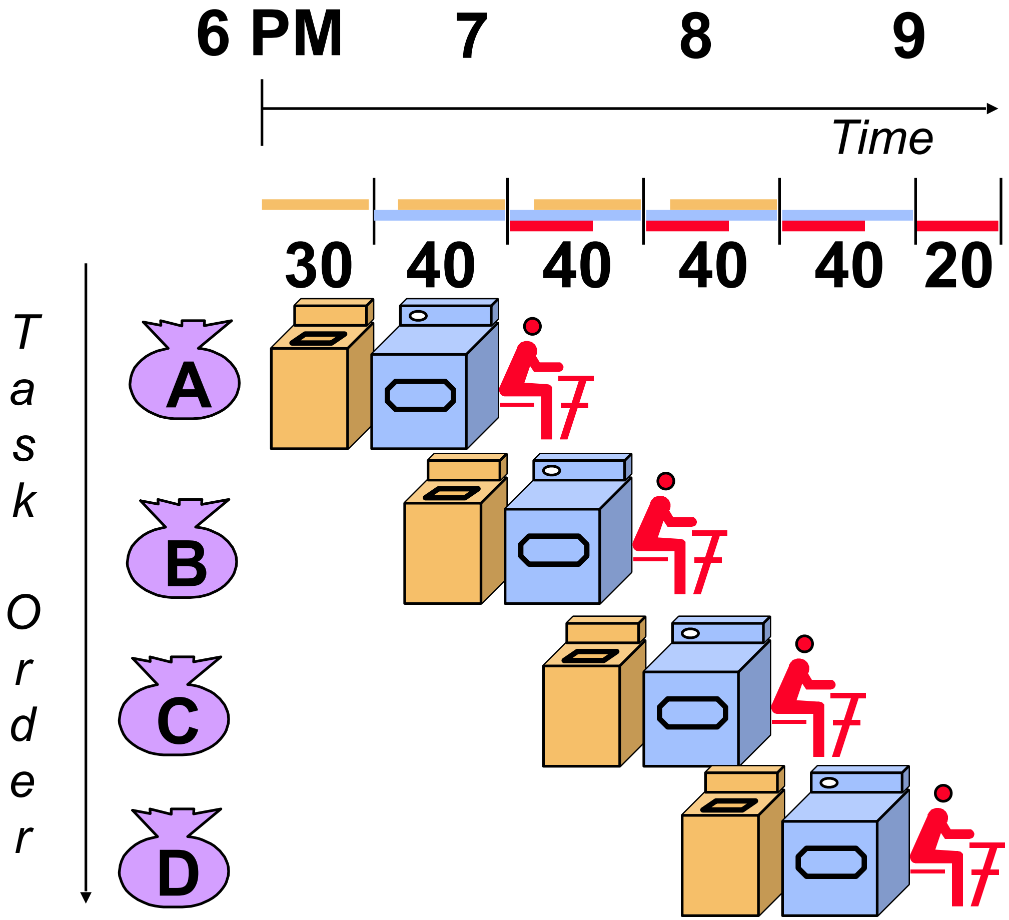
If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



Pipelining doesn't help latency of single task, it helps throughput of entire workload

Pipeline rate limited by slowest pipeline stage

Multiple tasks operating simultaneously using different resources

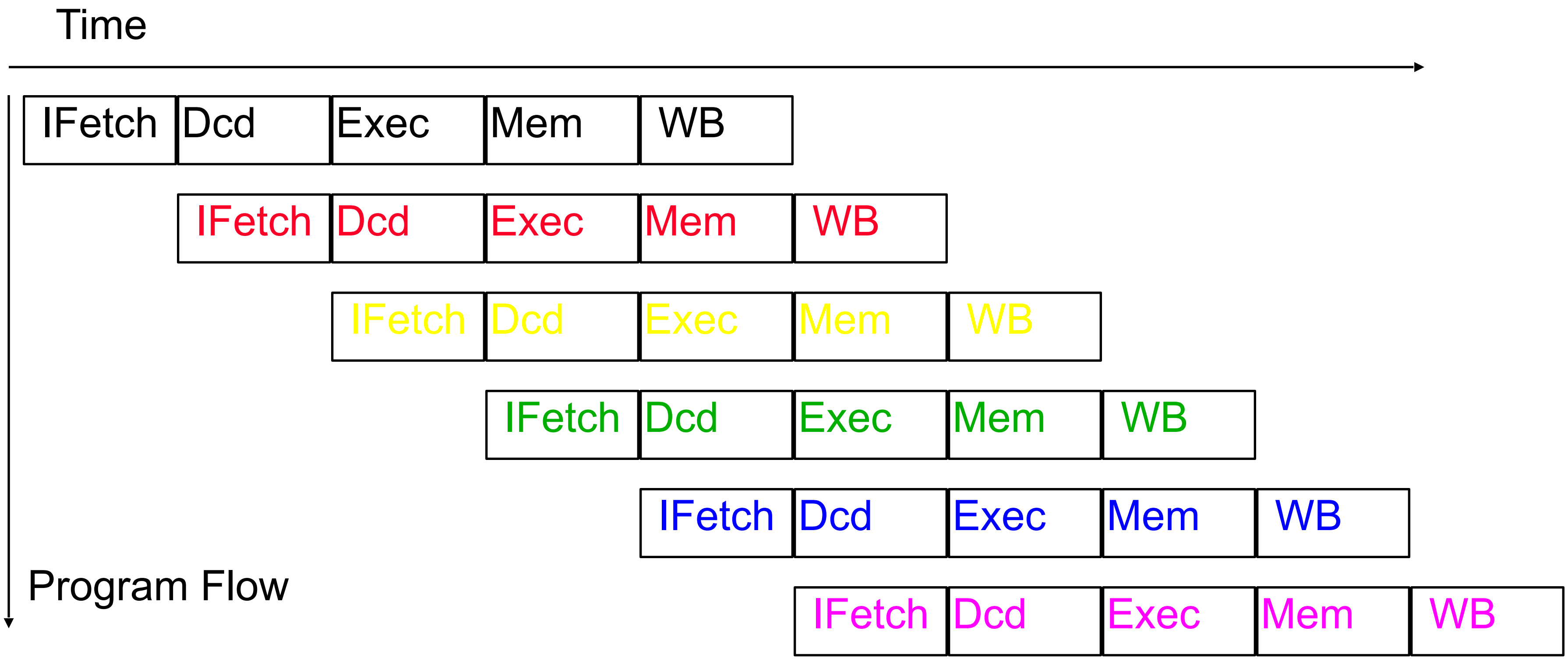
Potential speedup = Number pipe stages

Unbalanced lengths of pipe stages reduces speedup

Time to "fill" pipeline and time to "drain" it reduces speedup

Stall for Dependences

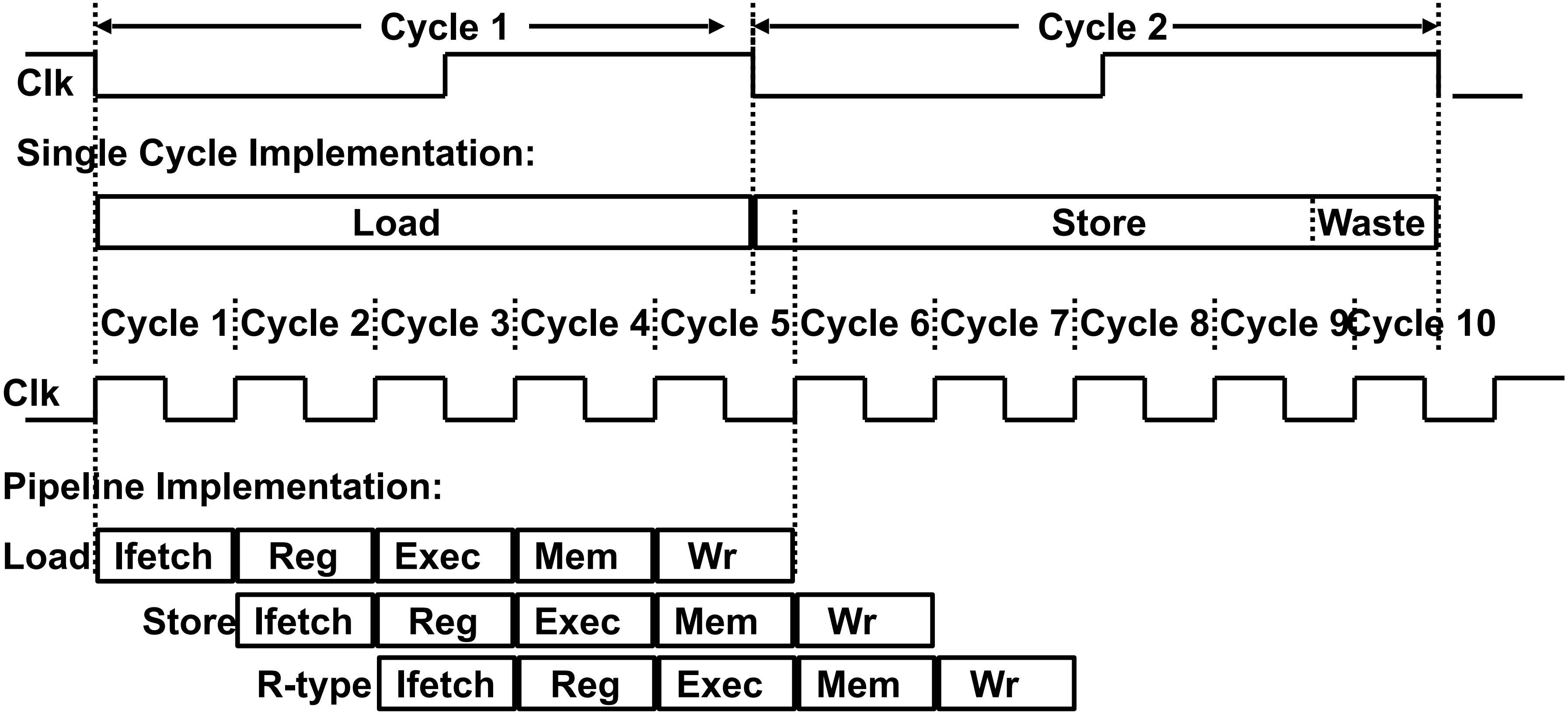
Pipelined Execution



IFetch	Dcd	Exec	Mem	WB					
	IFetch	Dcd	Exec	Mem	WB				
		IFetch	Dcd	Exec	Mem	WB			
			IFetch	Dcd	Exec	Mem	WB		
				IFetch	Dcd	Exec	Mem	WB	
					IFetch	Dcd	Exec	Mem	WB

Now we just have to make it work

Single Cycle vs. Pipeline



Pipelined Datapath

Divide datapath into multiple pipeline stages

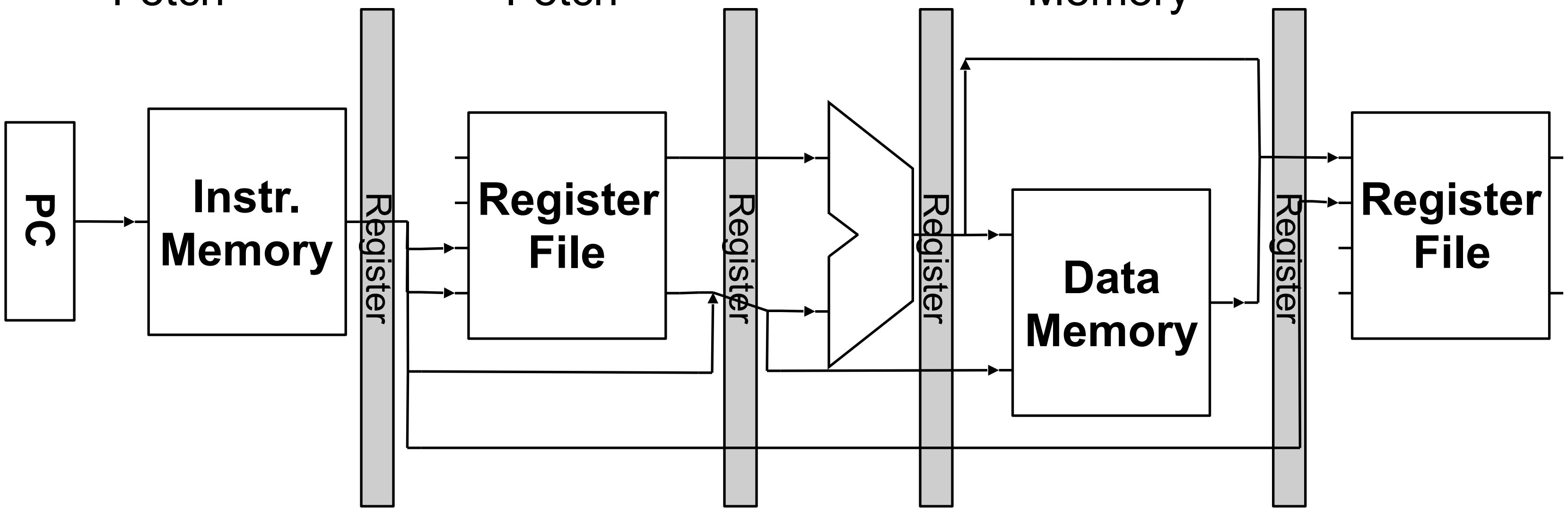
IF
Instruction
Fetch

RF
Register
Fetch

EX
Execute

MEM
Data
Memory

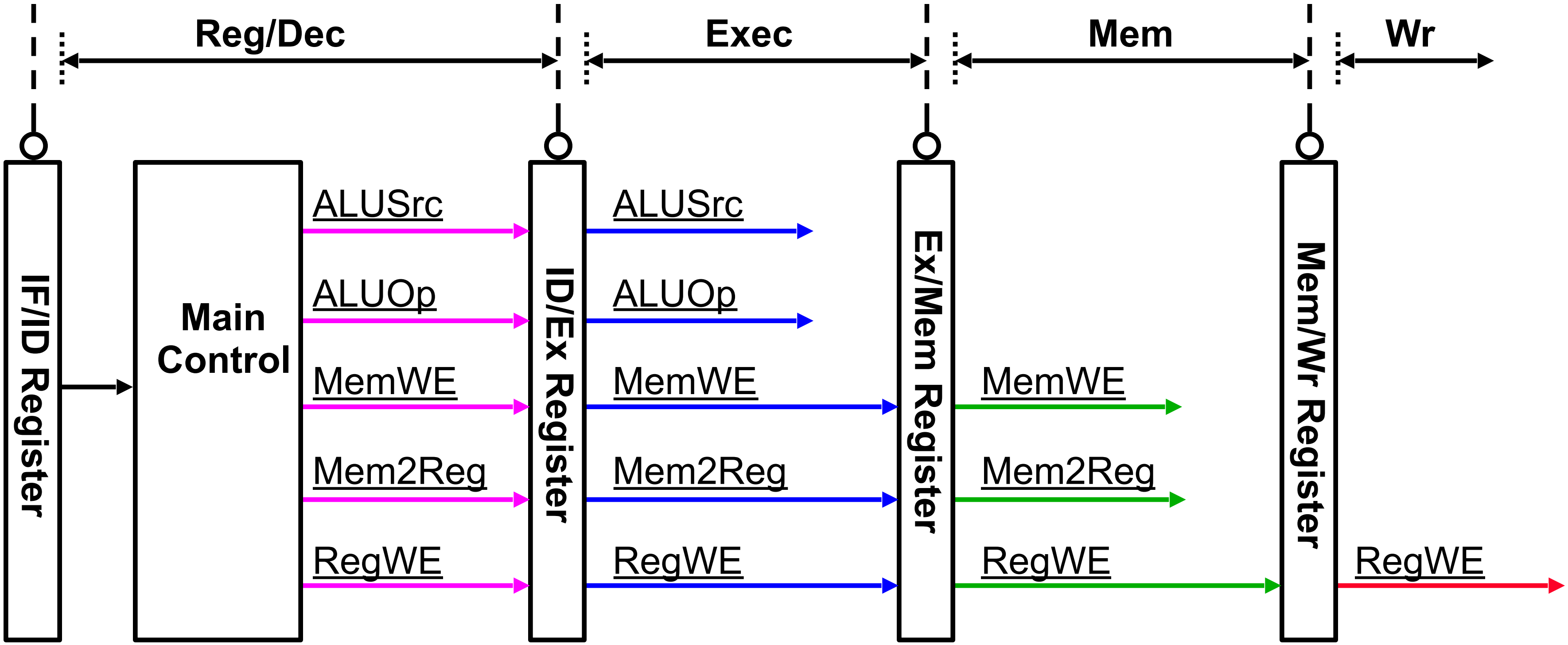
WB
Writeback



Pipelined Control

The Main Control generates the control signals during Reg/Dec

- Control signals for Exec (ALUOp, ALUSrc, …) are used 1 cycle later
- Control signals for Mem (MemWE, Mem2Reg, …) are used 2 cycles later
- Control signals for Wr (RegWE, …) are used 3 cycles later



Can pipelining get us into trouble?

Yes: **Pipeline Hazards**

- **structural hazards**: attempt to use the same resource two different ways at the same time
  - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
- **data hazards**: attempt to use item before it is ready
  - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
  - instruction depends on result of prior instruction still in the pipeline
- **control hazards**: attempt to make decision before condition evaluated
  - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
  - branch instructions

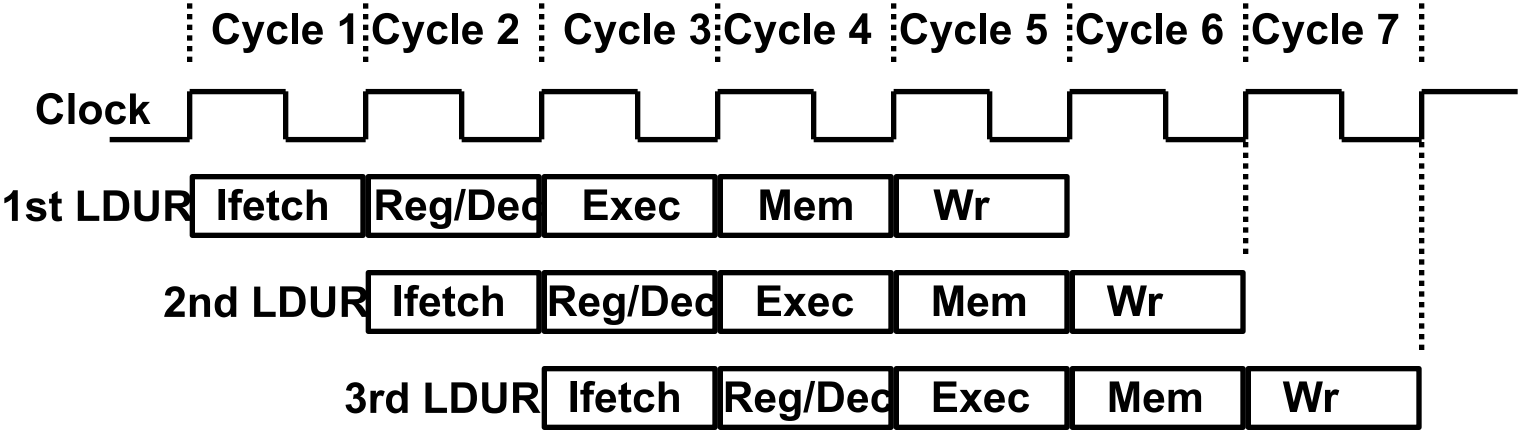
Can always resolve hazards by **waiting**

- pipeline control must detect the hazard
- take action (or delay action) to resolve hazards

Pipelining the Load Instruction

The five independent functional units in the pipeline datapath are:

- Instruction Memory for the Ifetch stage
- Register File’s Read ports (bus A and busB) for the Reg/Dec stage
- ALU for the Exec stage
- Data Memory for the Mem stage
- Register File’s Write port (bus W) for the Wr stage



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
Clock							
1st LDUR	Ifetch	Reg/Dec	Exec	Mem	Wr		
2nd LDUR		Ifetch	Reg/Dec	Exec	Mem	Wr	
3rd LDUR			Ifetch	Reg/Dec	Exec	Mem	Wr

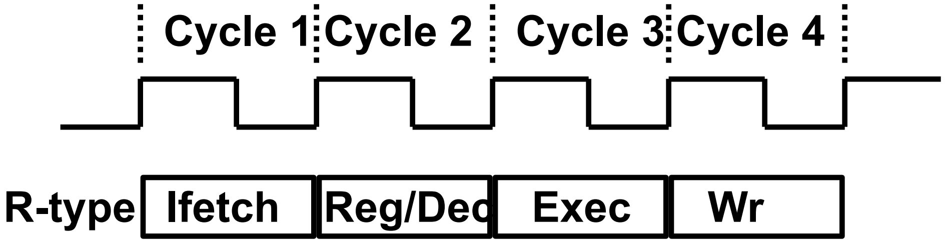
The Four Stages of R-type

Ifetch: Fetch the instruction from the Instruction Memory

Reg/Dec: Register Fetch and Instruction Decode

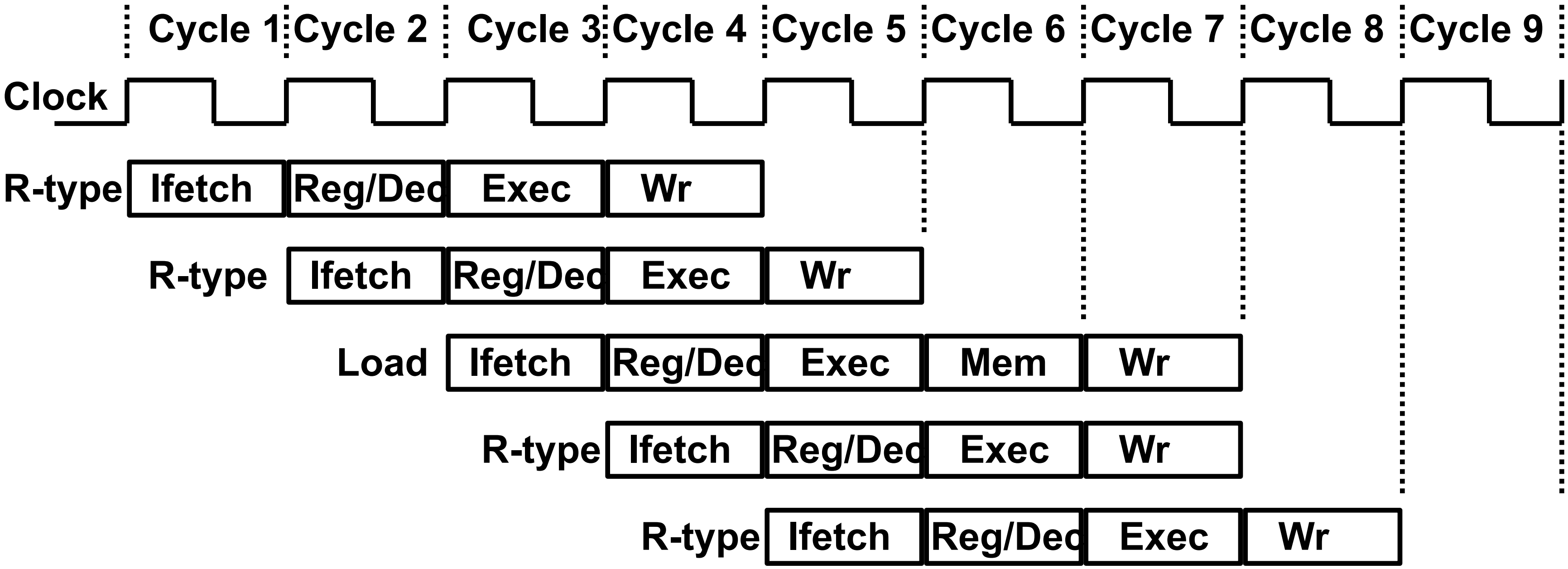
Exec: ALU operates on the two register operands

Wr: Write the ALU output back to the register file



Structural Hazard

Interaction between R-type and loads causes structural hazard on writeback



Important Observation

Each functional unit can only be used once per instruction

Each functional unit must be used at the same stage for all instructions:

Load uses Register File's Write Port during its 5th stage

	1	2	3	4	5
Load	**Ifetch**	**Reg/Dec**	**Exec**	**Mem**	**Wr**

R-type uses Register File's Write Port during its 4th stage

	1	2	3	4
R-type	**Ifetch**	**Reg/Dec**	**Exec**	**Wr**

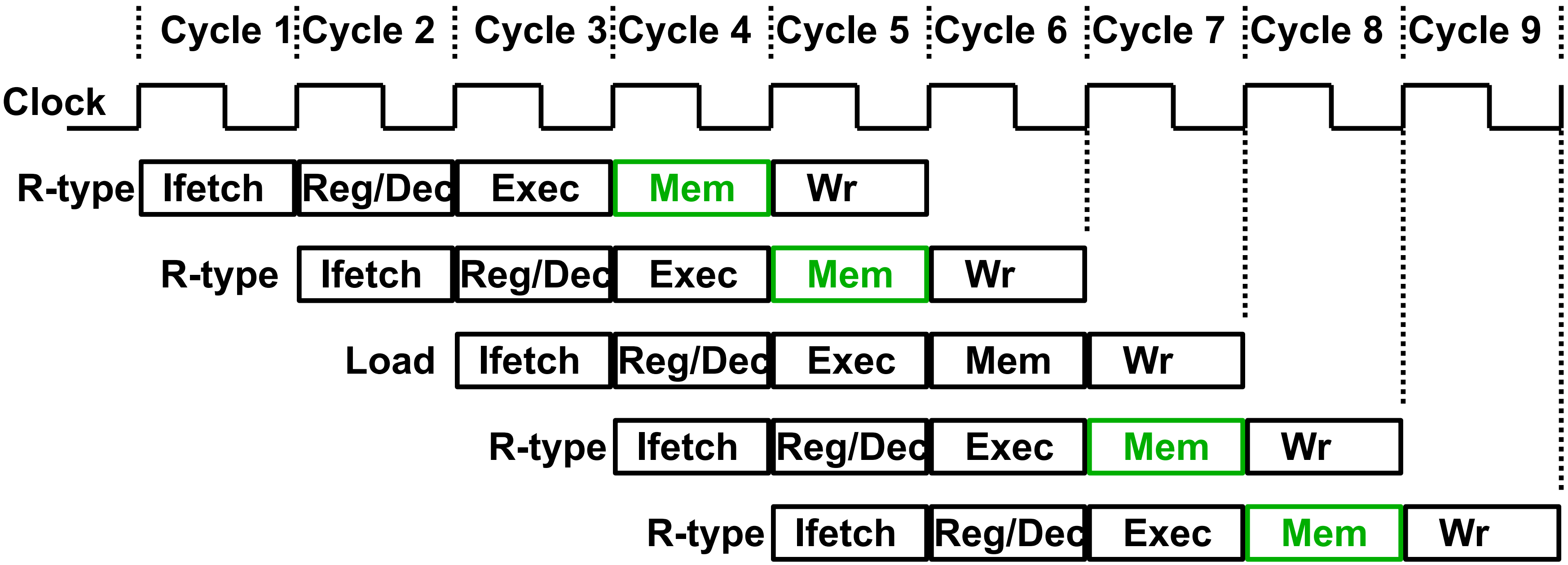
Solution: Delay R-type's register write by one cycle:

Now R-type instructions also use Reg File's write port at Stage 5

Mem stage is a NOOP stage: nothing is being done.

	1	2	3	4	5
R-type	**Ifetch**	**Reg/Dec**	**Exec**	**Mem**	**Wr**

Pipelining the R-type Instruction



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr				
R-type		Ifetch	Reg/Dec	Exec	Mem	Wr			
Load			Ifetch	Reg/Dec	Exec	Mem	Wr		
R-type				Ifetch	Reg/Dec	Exec	Mem	Wr	
R-type					Ifetch	Reg/Dec	Exec	Mem	Wr

The Four Stages of Store

Ifetch: Fetch the instruction from the Instruction Memory

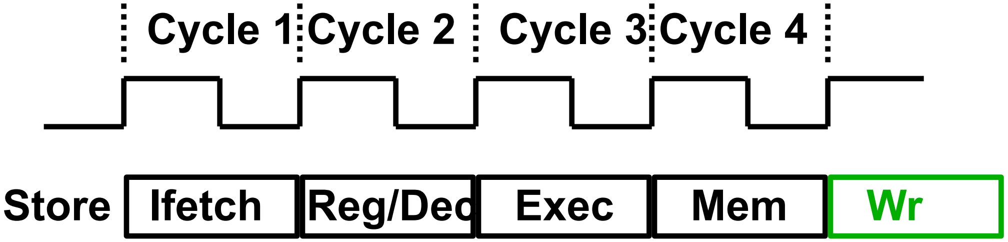
Reg/Dec: Register Fetch and Instruction Decode

Exec: Calculate the memory address

Mem: Write the data into the Data Memory

Wr: NOOP

Compatible with Load & R-type instructions



The Stages of Conditional Branch

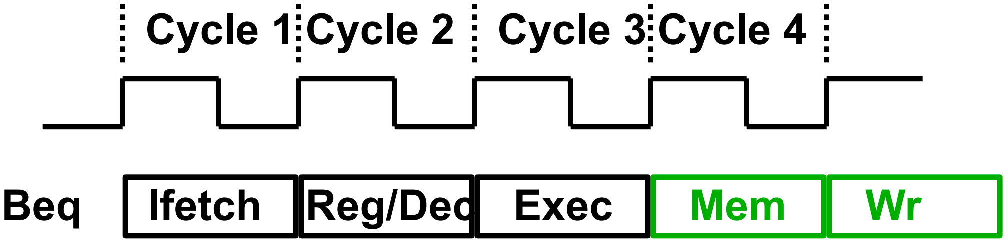
Ifetch: Fetch the instruction from the Instruction Memory

Reg/Dec: Register Fetch and Instruction Decode, compute branch target

Exec: Test condition & update the PC

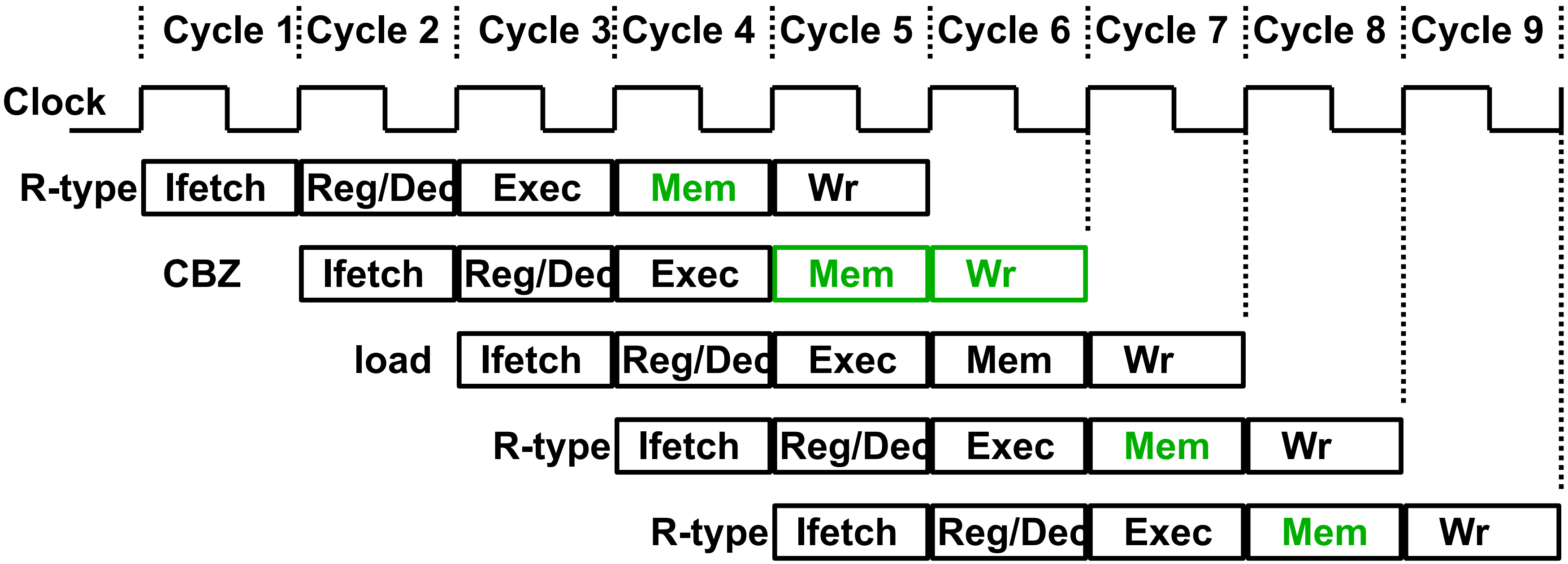
Mem: NOOP

Wr: NOOP



Control Hazard

Branch updates the PC at the end of the Exec stage.

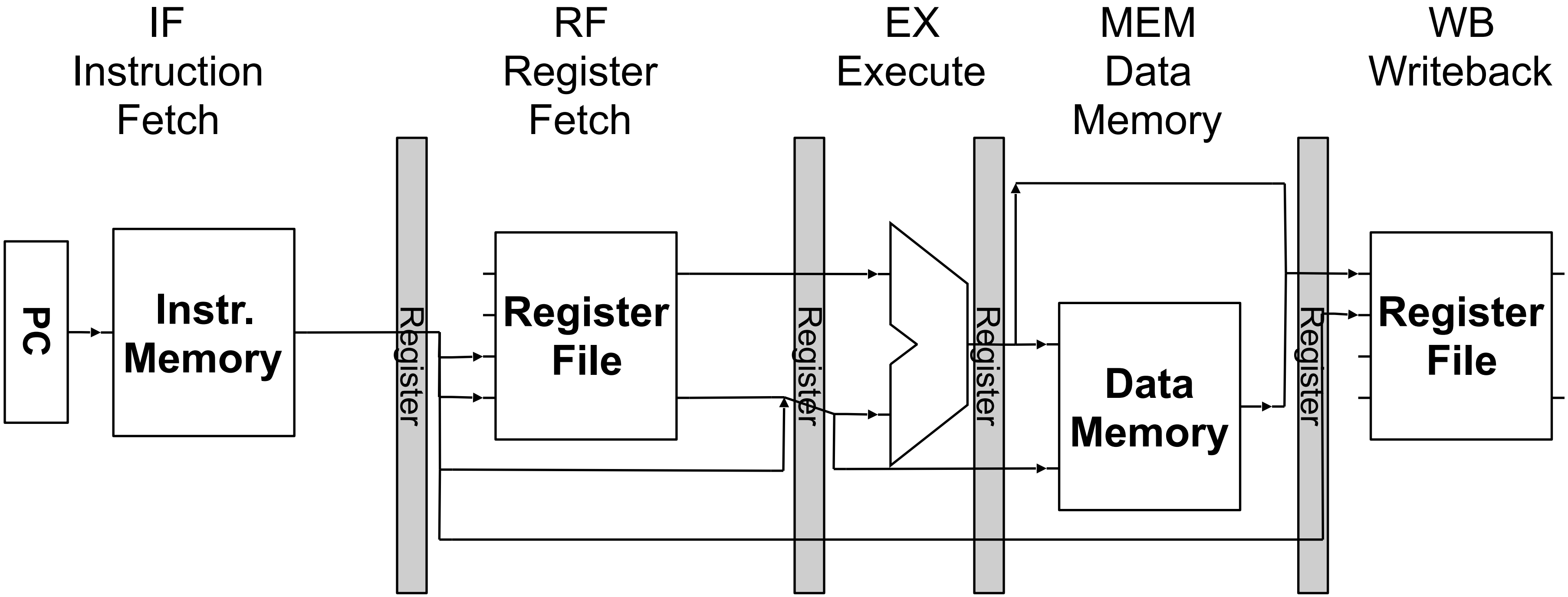


	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr				
CBZ		Ifetch	Reg/Dec	Exec	Mem	Wr			
load			Ifetch	Reg/Dec	Exec	Mem	Wr		
R-type				Ifetch	Reg/Dec	Exec	Mem	Wr	
R-type					Ifetch	Reg/Dec	Exec	Mem	Wr

Accelerate Branches

When can we compute branch target address?

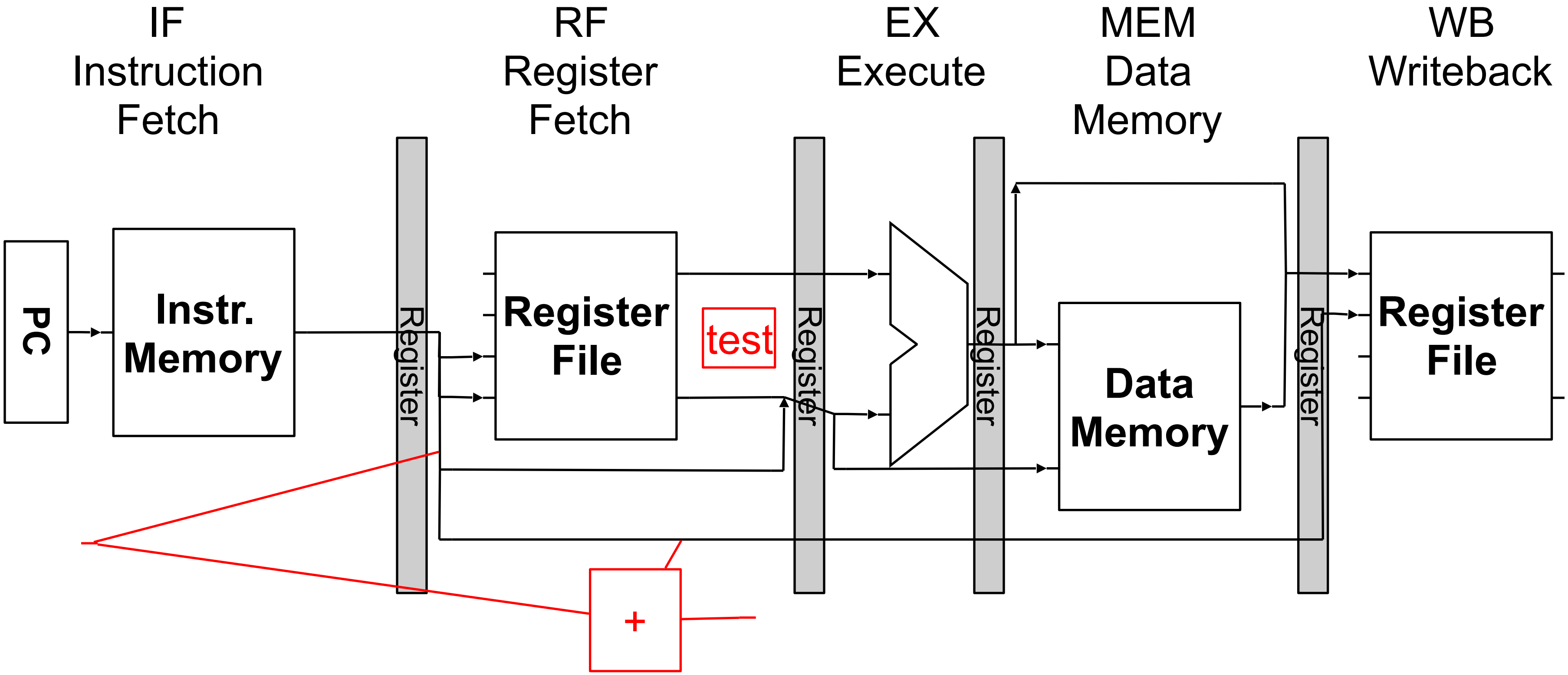
When can we compute the CBZ condition?



Accelerate Branches

When can we compute branch target address?

When can we compute beq condition?



Solution #3: Branch Delay Slot

Redefine branches: Instruction directly after branch always executed

Instruction after branch is the **delay slot**

Compiler/assembler **fills** the delay slot

```
~~ADD X1, X0, X4~~
CBZ X2, FOO
ADD X1, X0, X4
```

No wasted cycles

```
~~SUB X2, X0, X3~~
ADD X1, X0, X4
CBZ X1, FOO
SUB X2, X0, X3
```

No wasted cycles

```
  ADD X1, X0, X4
  CBZ X1, FOO
  ADD X1, X2, X0
  ADD X1, X3, X3
  …
FOO:
  ~~ADD X1, X2, X0~~
```

Assume 50% branch,
Wastes ½ cycle per branch

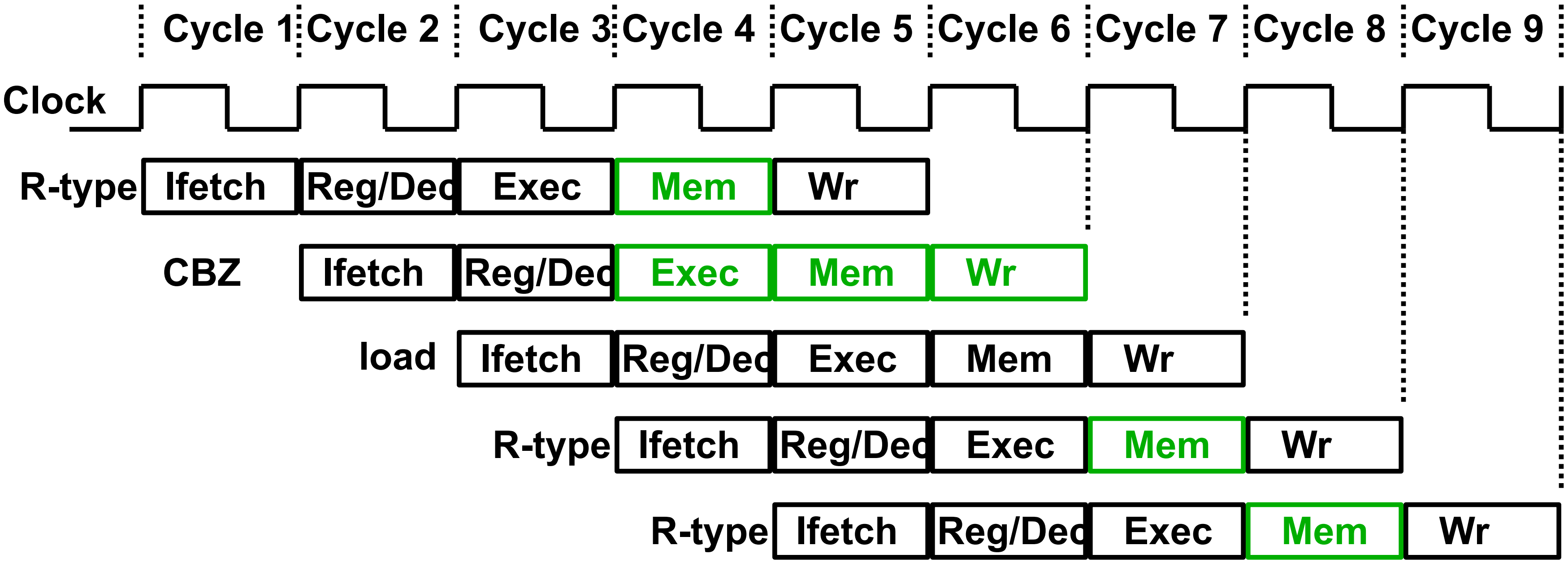
Compare vs. stall

```
ADD X1, X0, X4
CBZ X1, FOO
ADD X31, X31, X31
```

Insert noop
Wastes 1 cycle per branch

Control Hazard 2

Branch updates the PC at the end of the Reg/Dec stage.



Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9

Clock

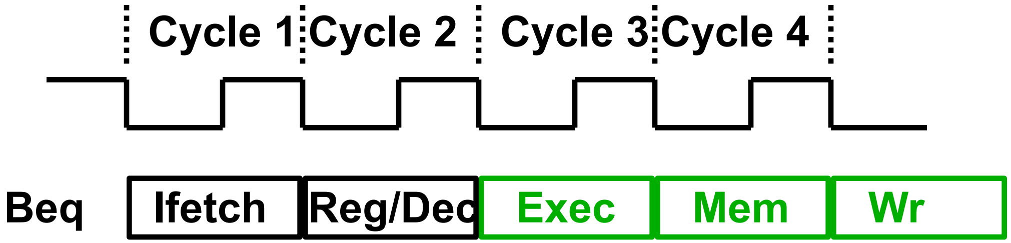
R-type Ifetch Reg/Dec Exec Mem Wr

CBZ Ifetch Reg/Dec Exec Mem Wr

load Ifetch Reg/Dec Exec Mem Wr

R-type Ifetch Reg/Dec Exec Mem Wr

R-type Ifetch Reg/Dec Exec Mem Wr

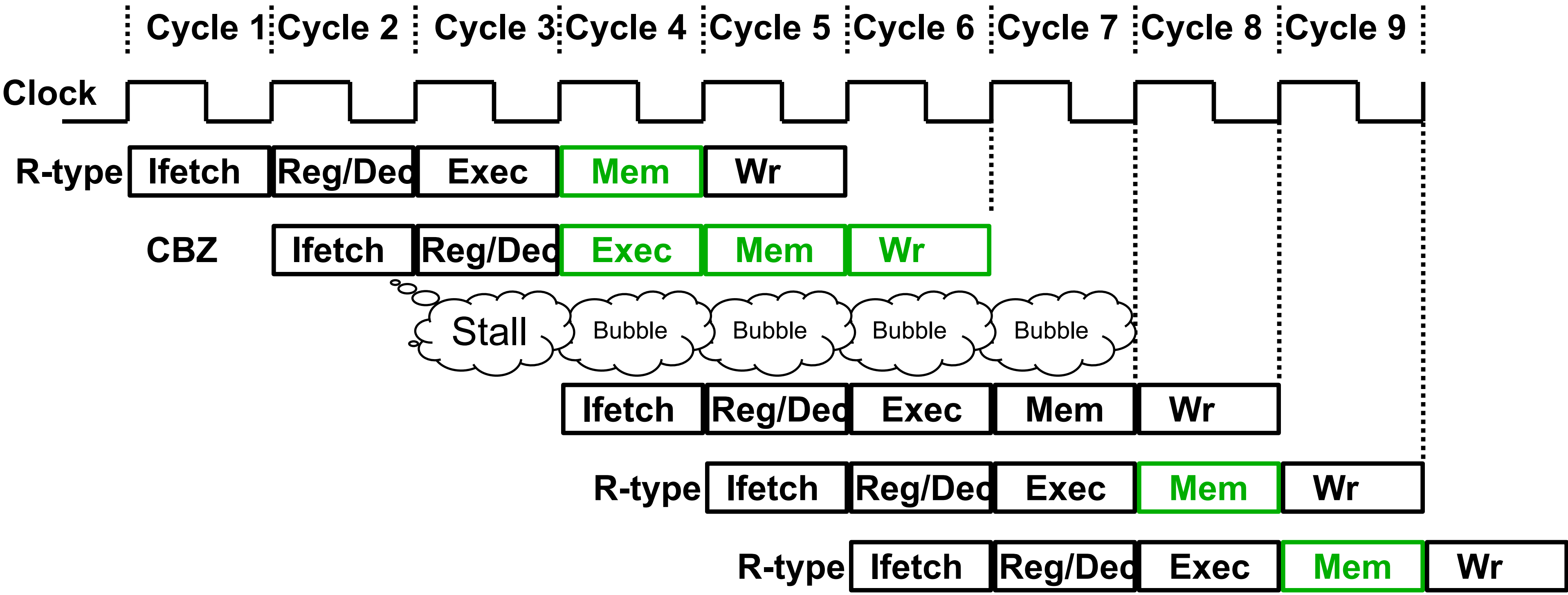


Cycle 1 Cycle 2 Cycle 3 Cycle 4

Beq Ifetch Reg/Dec Exec Mem Wr

Solution #1: Stall

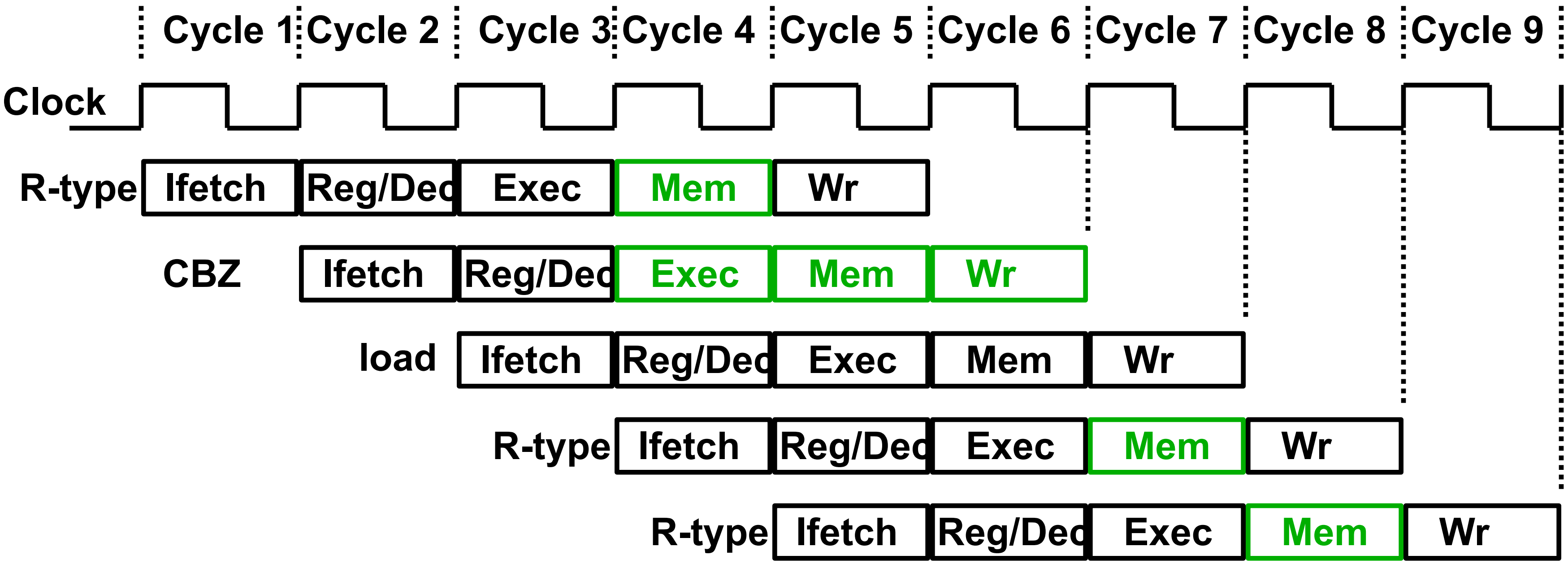
Delay loading next instruction, load no-op instead



CPI if all other instructions take 1 cycle, and branches are 20% of instructions?

Solution #2: Branch Prediction

Guess all branches not taken, squash if wrong



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr				
CBZ		Ifetch	Reg/Dec	Exec	Mem	Wr			
load			Ifetch	Reg/Dec	Exec	Mem	Wr		
R-type				Ifetch	Reg/Dec	Exec	Mem	Wr	
R-type					Ifetch	Reg/Dec	Exec	Mem	Wr

CPI if 50% of branches actually not taken, and branch frequency 20%?

Solution #3: Branch Delay Slot

Redefine branches: Instruction directly after branch always executed

Instruction after branch is the **delay slot**

Compiler/assembler **fills** the delay slot

```
ADD X1, X0, X4
CBZ X2, FOO
```

```
SUB X2, X0, X3
ADD X1, X0, X4
CBZ X1, FOO
```

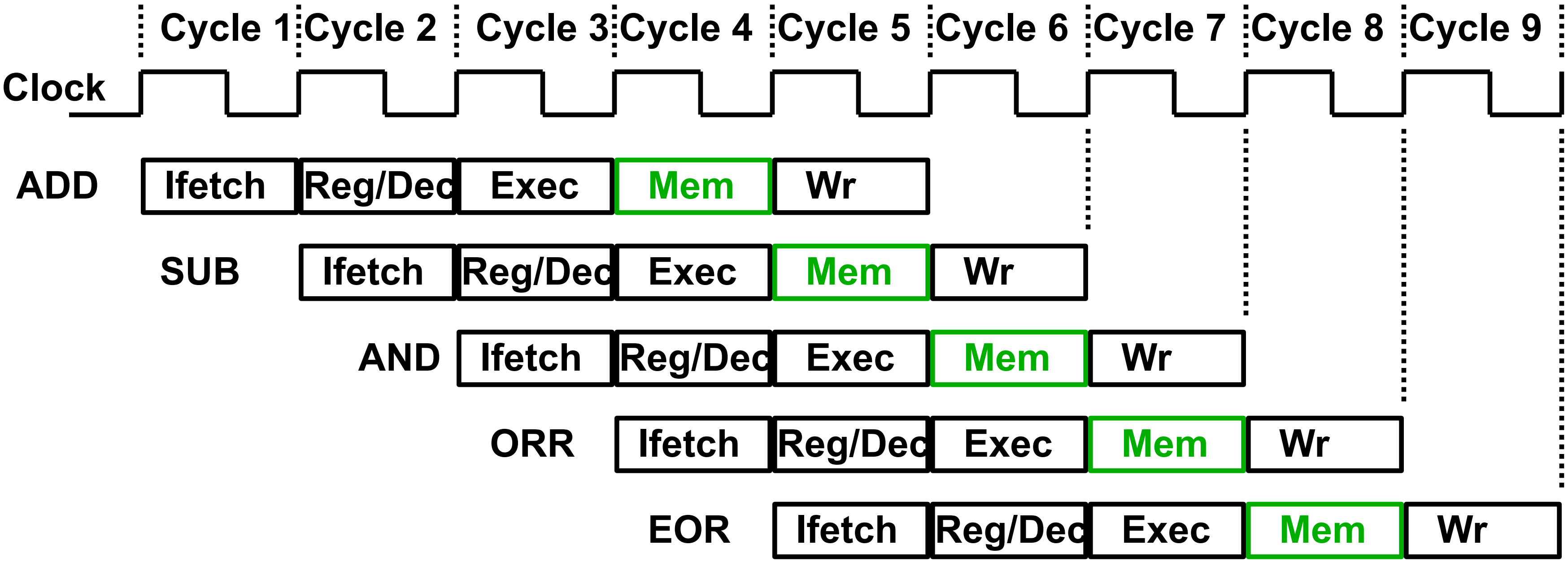
```
  ADD X1, X0, X4
  CBZ X1, FOO

  ADD X1, X3, X3
  …
FOO:
  ADD X1, X2, X0
```

```
ADD X1, X0, X4
CBZ X1, FOO
```

Data Hazards

Consider the following code:

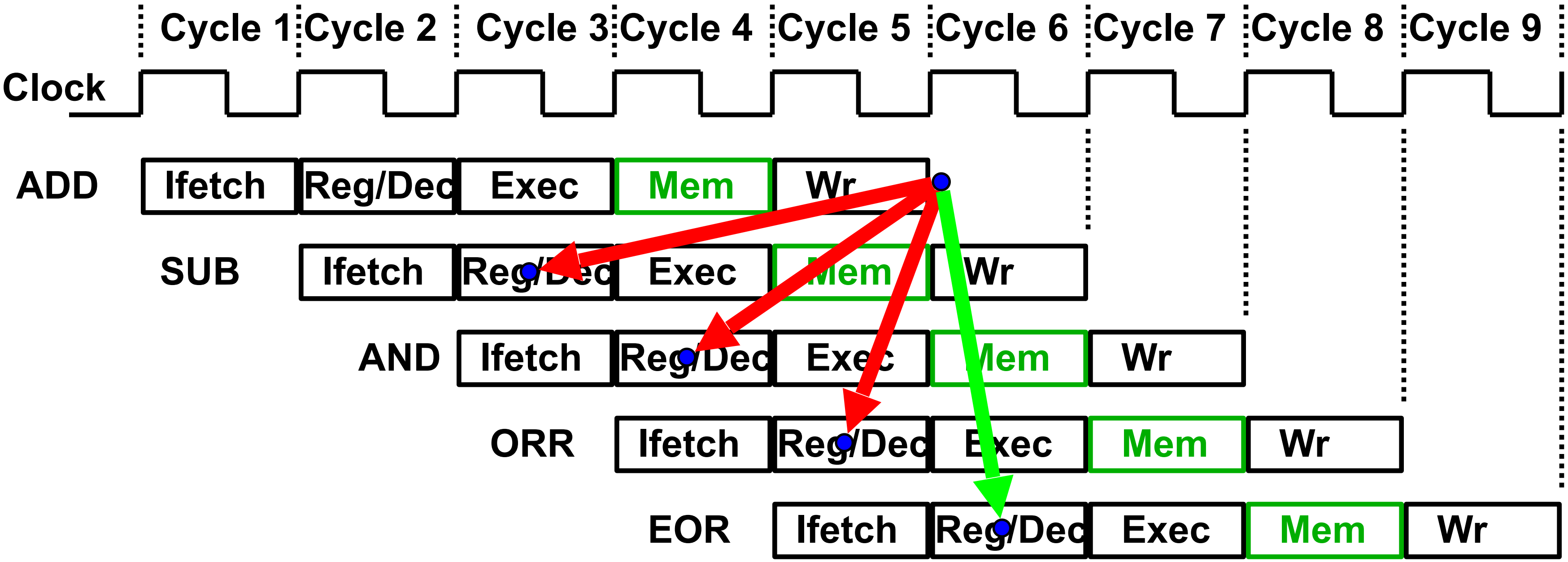
```
ADD X0, X1, X2
SUB X3, X0, X4
AND X5, X0, X6
ORR X7, X0, X8
EOR X9, X0, X10
```



Data Hazards

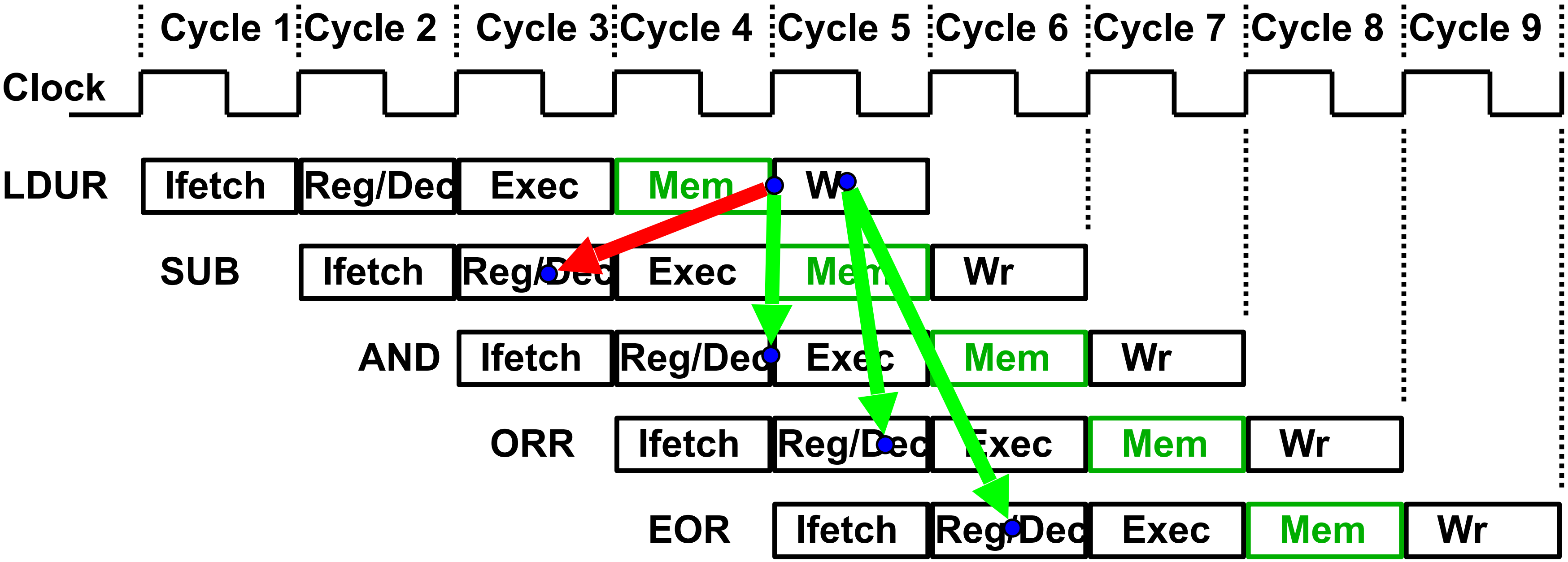
Consider the following code:

```
ADD X0, X1, X2
SUB X3, X0, X4
AND X5, X0, X6
ORR X7, X0, X8
EOR X9, X0, X10
```



Data Hazards on Loads

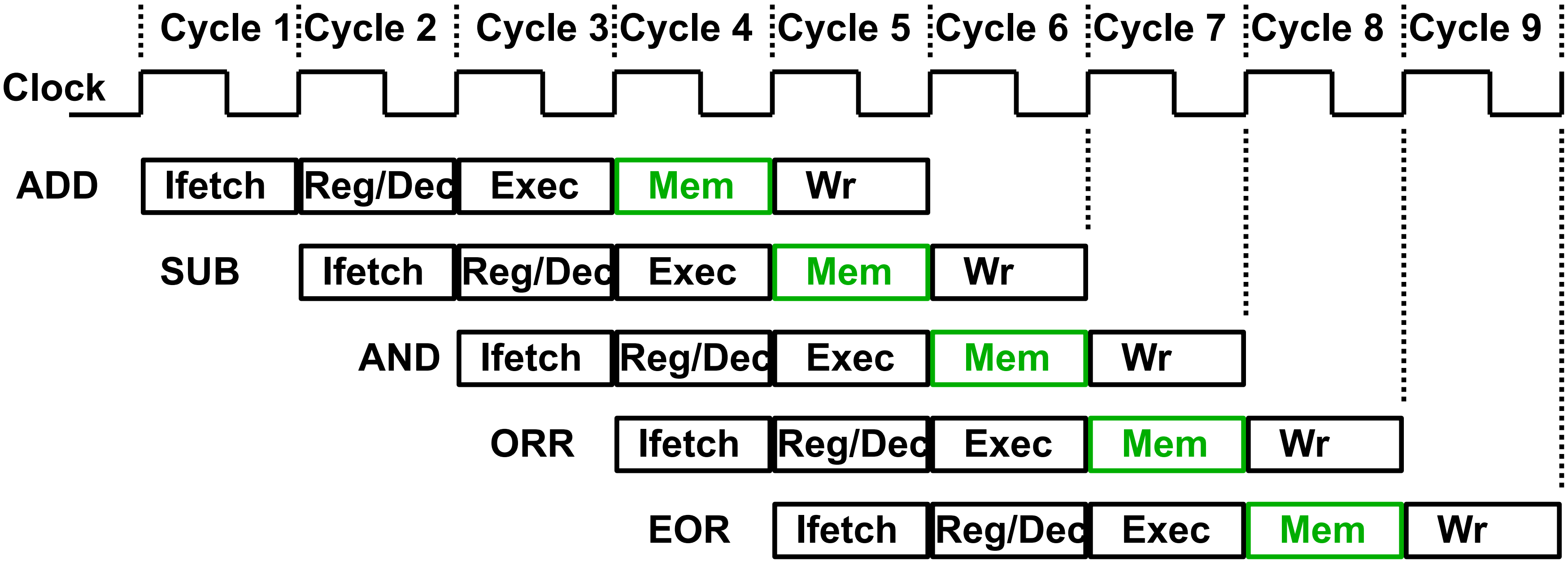
LDUR X0, [X31, 0]
SUB X3, X0, X4 – Cannot be solved – data not available when needed.
AND X5, X0, X6 – Handled by forwarding logic
ORR X7, X0, X8 – Fixed by register file bypass
EOR X9, X0, X10 – Not a problem



Design Register File Carefully

What if reads see value after write during the same cycle?

ADD X0, X1, X2
SUB X3, X0, X4
AND X5, X0, X6
ORR X7, X0, X8
EOR X9, X0, X10

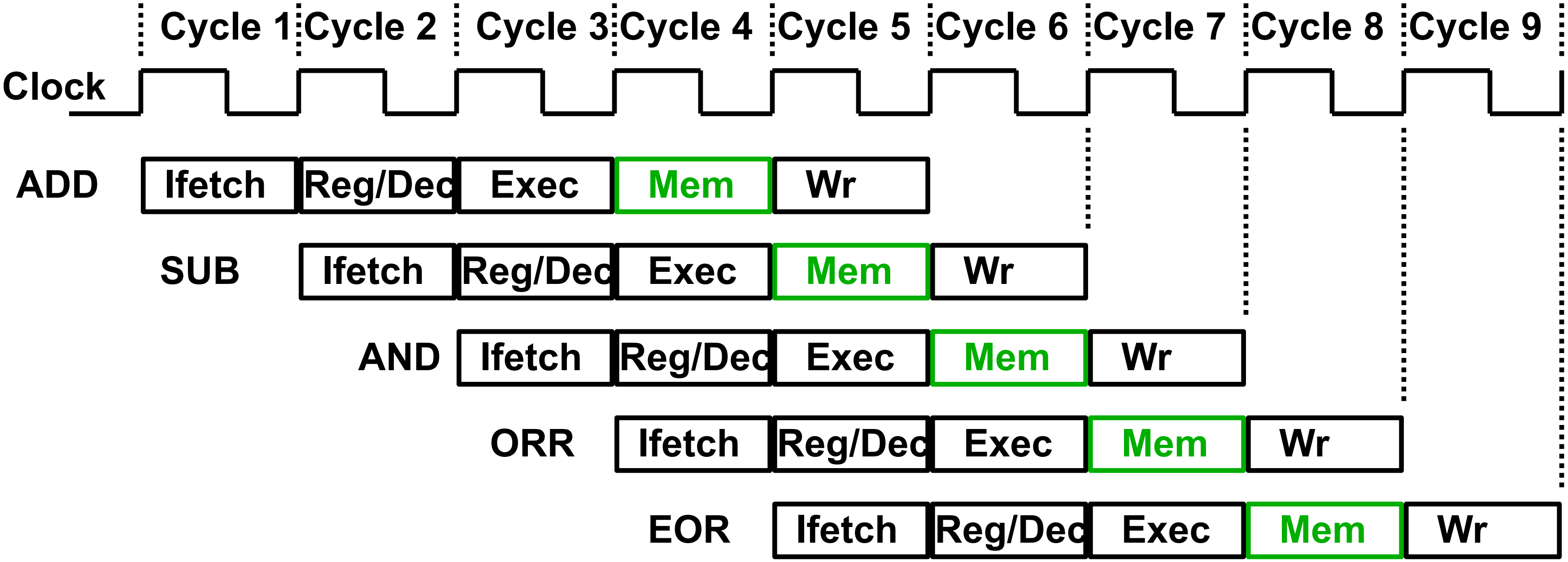


Forwarding

Add logic to pass last two values from ALU output to ALU input(s) as needed

Forward the ALU output to later instructions

ADD X0, X1, X2
SUB X3, X0, X4
AND X5, X0, X6
ORR X7, X0, X8
EOR X9, X0, X10



Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9

Clock

ADD Ifetch Reg/Dec Exec Mem Wr

SUB Ifetch Reg/Dec Exec Mem Wr

AND Ifetch Reg/Dec Exec Mem Wr

ORR Ifetch Reg/Dec Exec Mem Wr

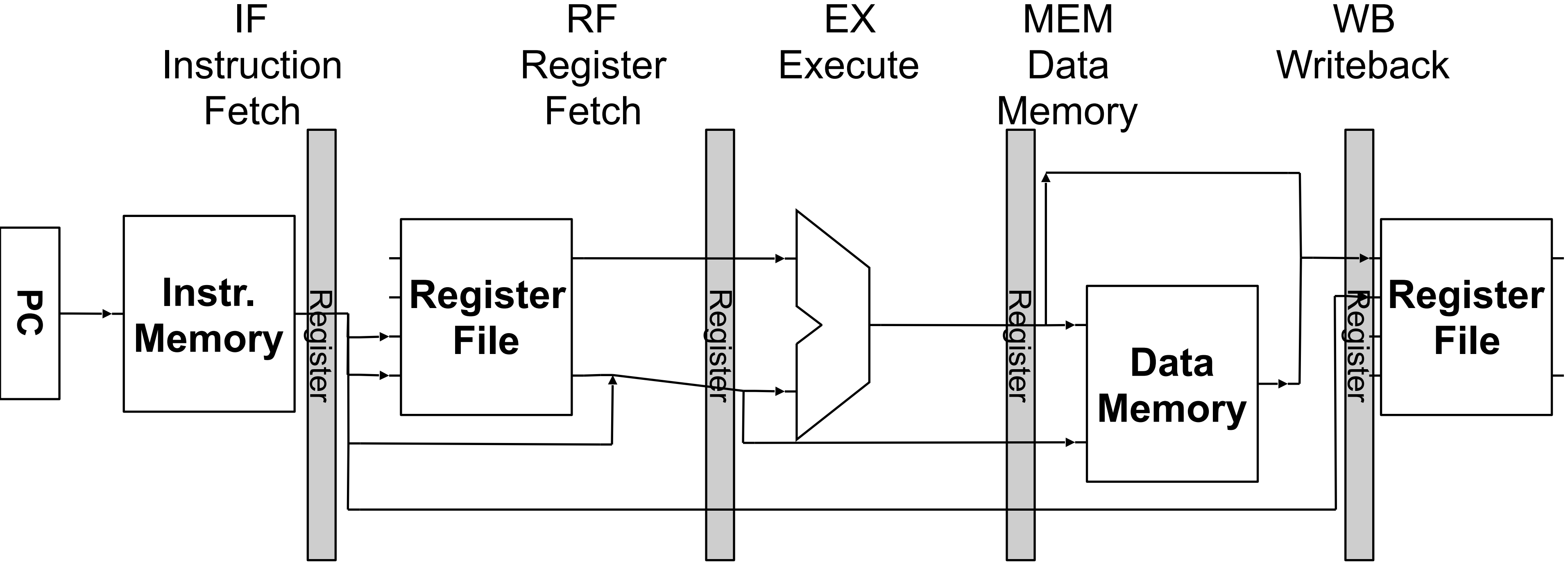
EOR Ifetch Reg/Dec Exec Mem Wr

Forwarding (cont.)

Requires values from last two ALU operations.

Remember destination register for operation.

Compare sources of current instruction to destinations of previous 2.

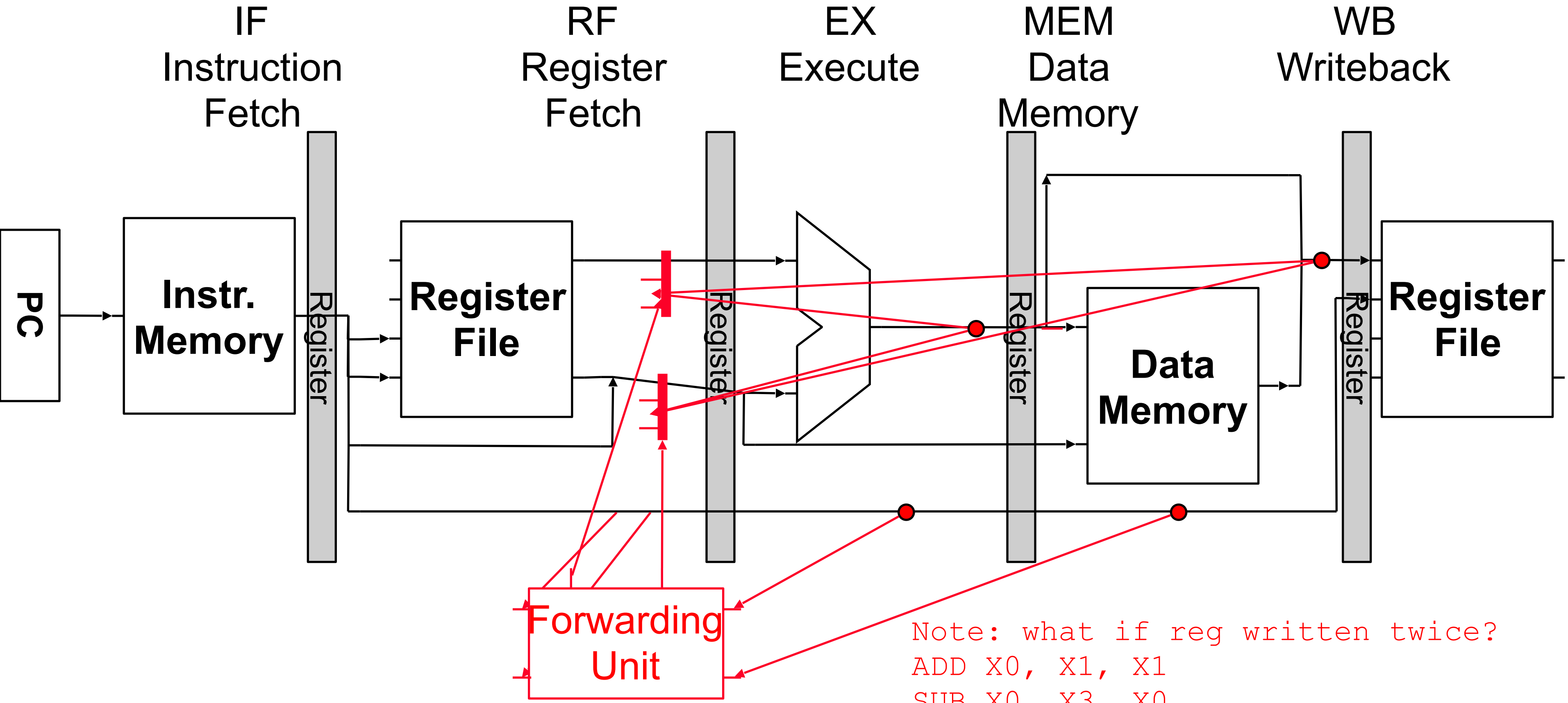


Forwarding (cont.)

Requires values from last two ALU operations.

Remember destination register for operation.

Compare sources of current instruction to destinations of previous 2.



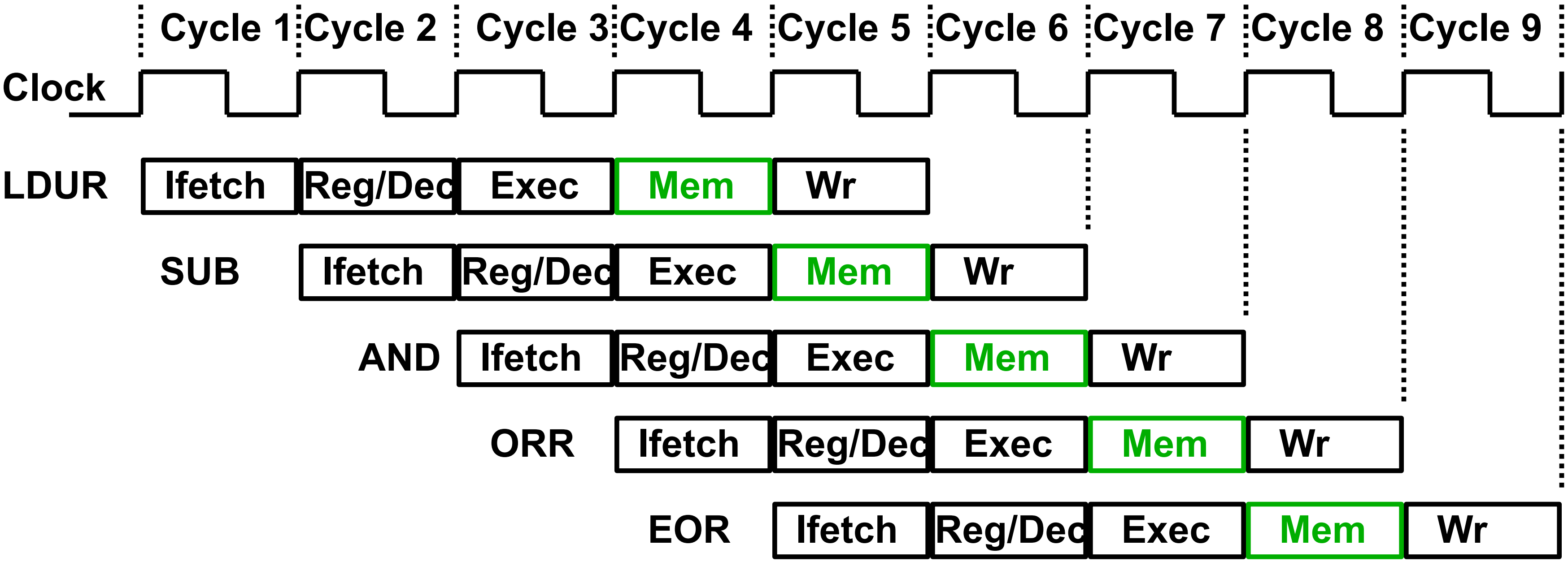
Note: what if reg written twice?

```
ADD X0, X1, X1
SUB X0, X3, X0
ORR X2, X0, X6
```

Write to X31?   STUR?

Data Hazards on Loads

```
LDUR X0, [X31, 0]
SUB X3, X0, X4
AND X5, X0, X6
ORR X7, X0, X8
EOR X9, X0, X10
```



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
LDUR	Ifetch	Reg/Dec	Exec	Mem	Wr				
SUB		Ifetch	Reg/Dec	Exec	Mem	Wr			
AND			Ifetch	Reg/Dec	Exec	Mem	Wr		
ORR				Ifetch	Reg/Dec	Exec	Mem	Wr	
EOR					Ifetch	Reg/Dec	Exec	Mem	Wr

Data Hazards on Loads (cont.)

Solution:

- Use same forwarding hardware & register file for hazards 2+ cycles later
- Force compiler to not allow register reads within a cycle of load
  - Fill delay slot, or insert no-op.