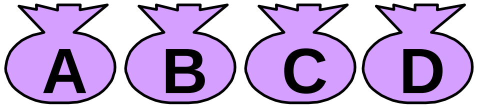
Pipelining

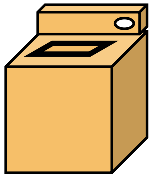
Example: Doing the laundry

Ann, Brian, Cathy, & Dave

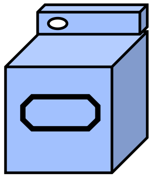


each have one load of clothes to wash, dry, and fold

Washer takes 30 minutes

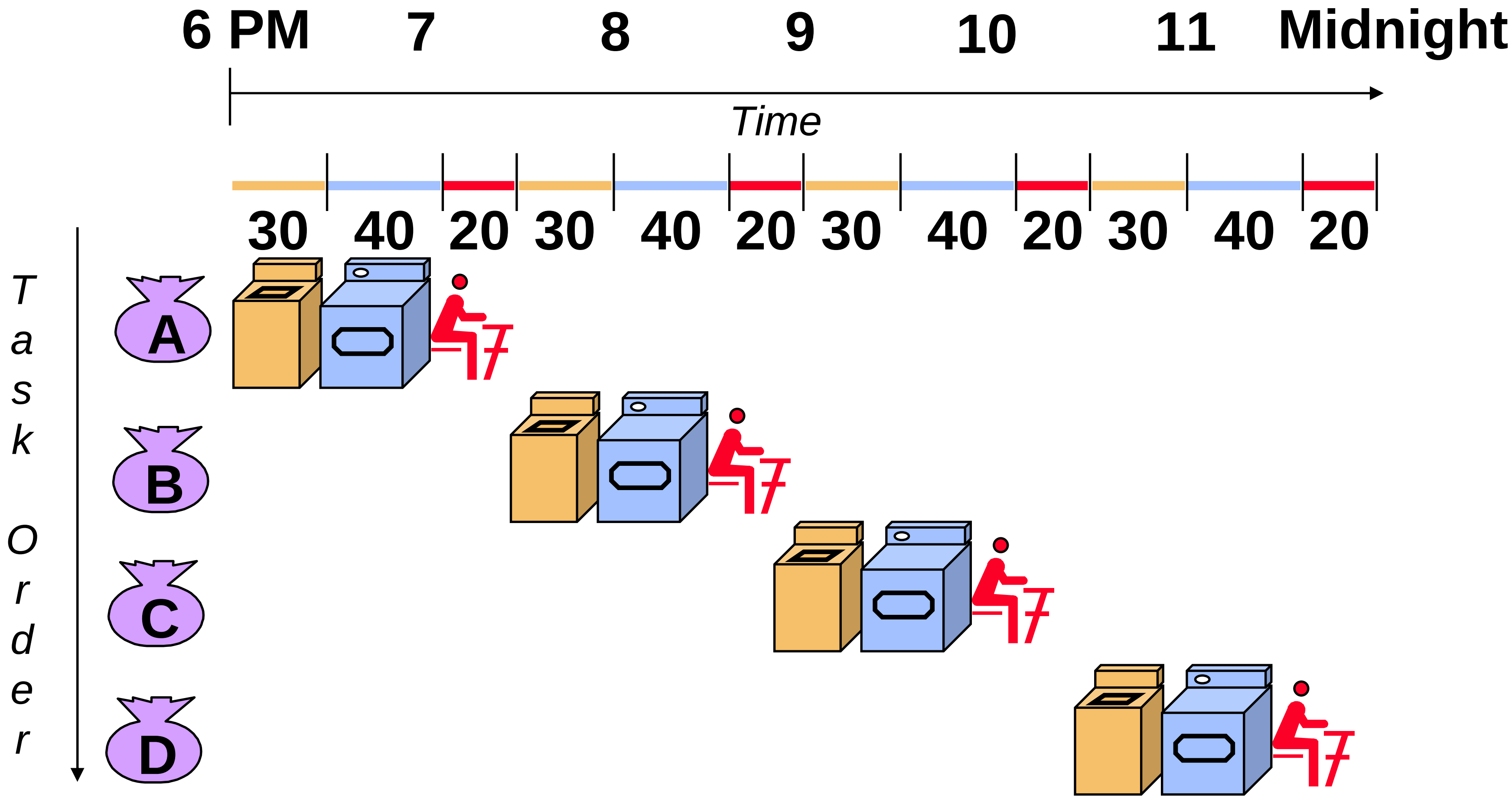


Dryer takes 40 minutes



“Folder” takes 20 minutes

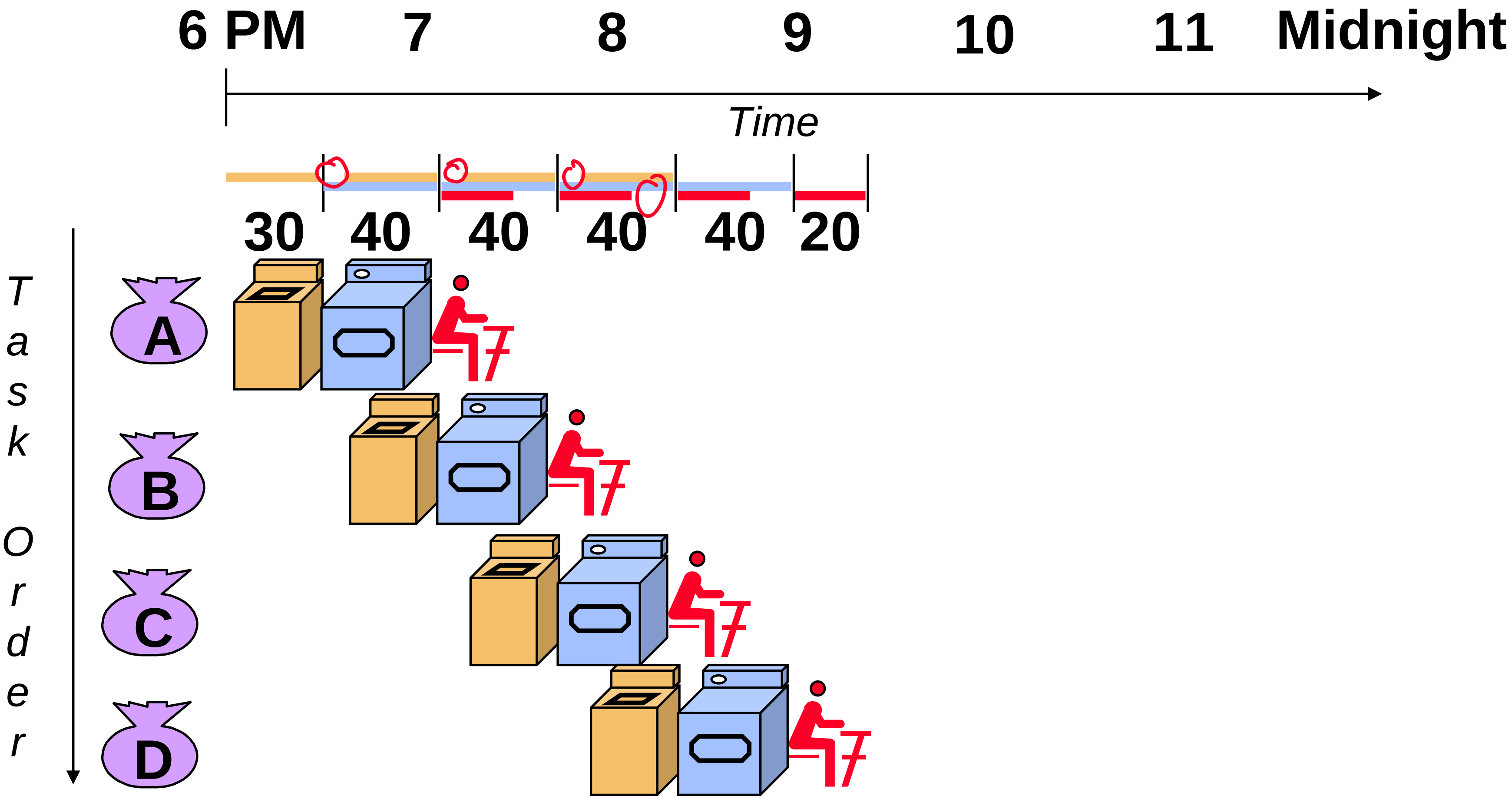
Sequential Laundry



Sequential laundry takes 6 hours for 4 loads

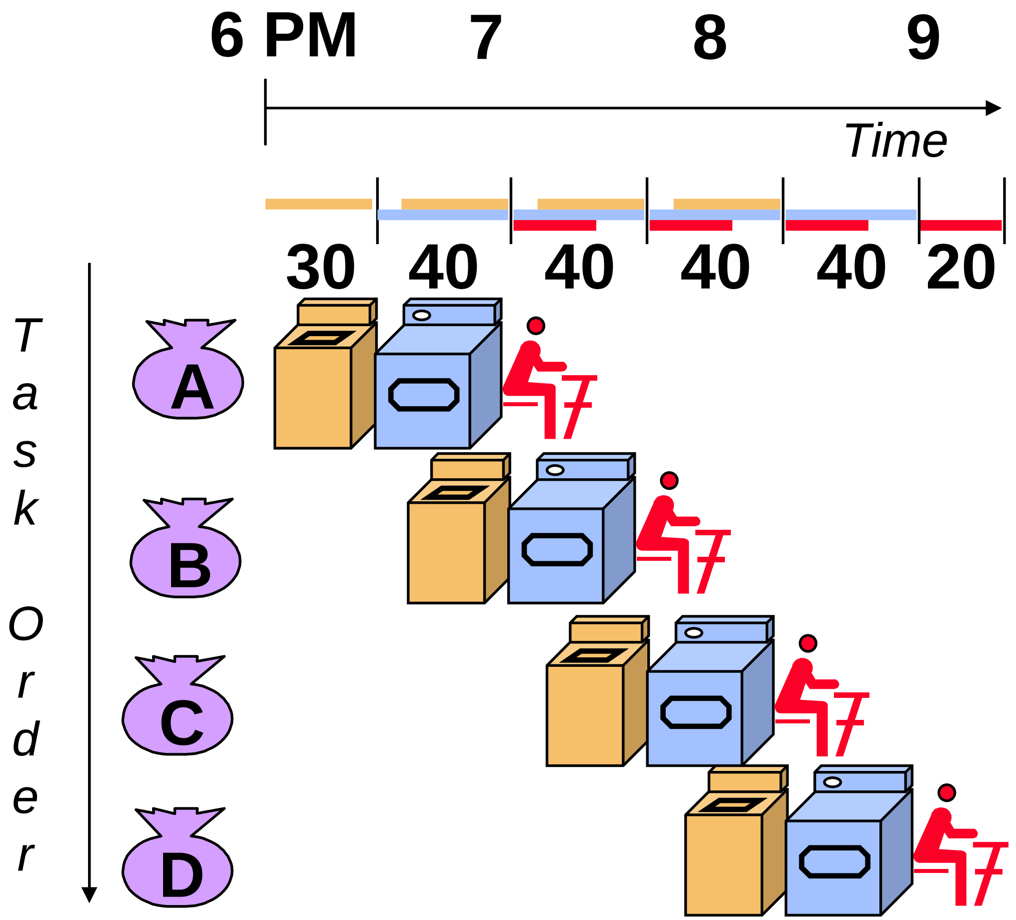
If they learned pipelining, how long would laundry take?

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

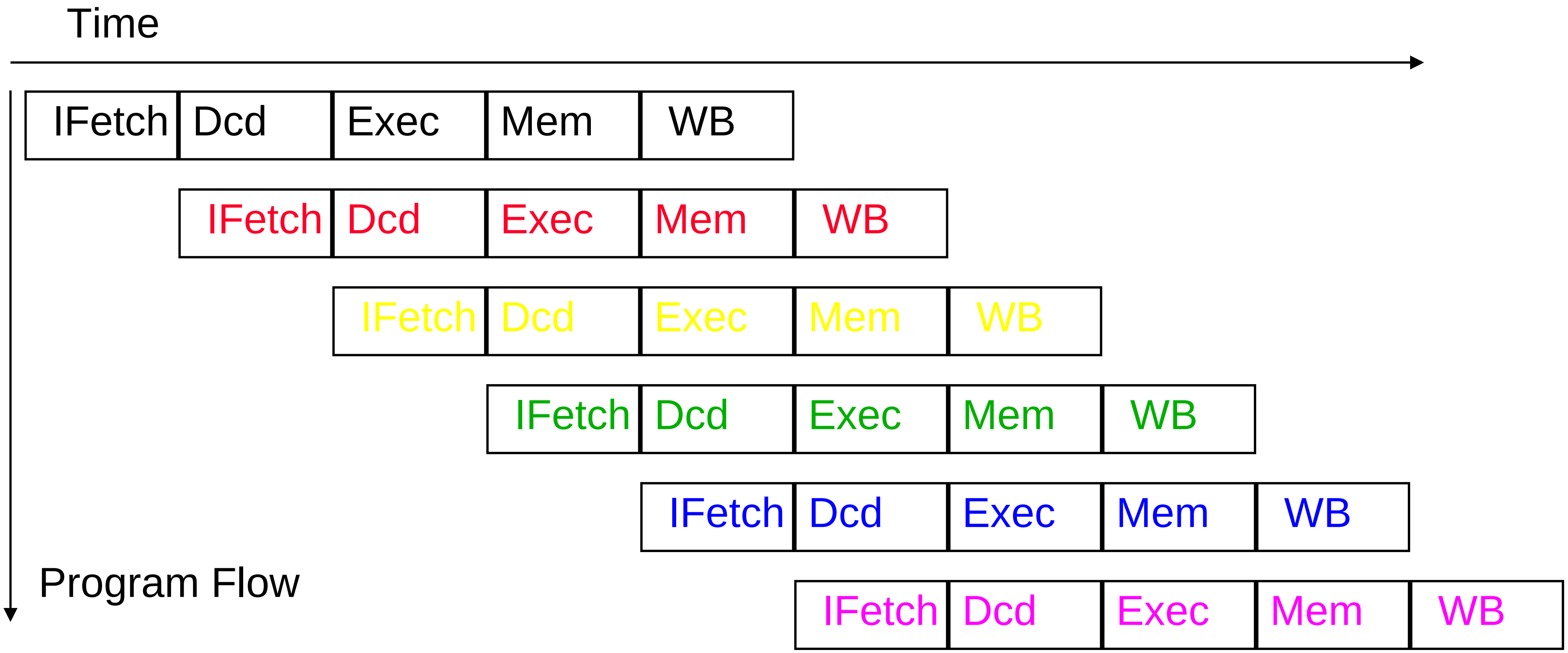
Potential speedup = Number pipe stages

Unbalanced lengths of pipe stages reduces speedup

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

Stall for Dependences

Pipelined Execution



Time

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

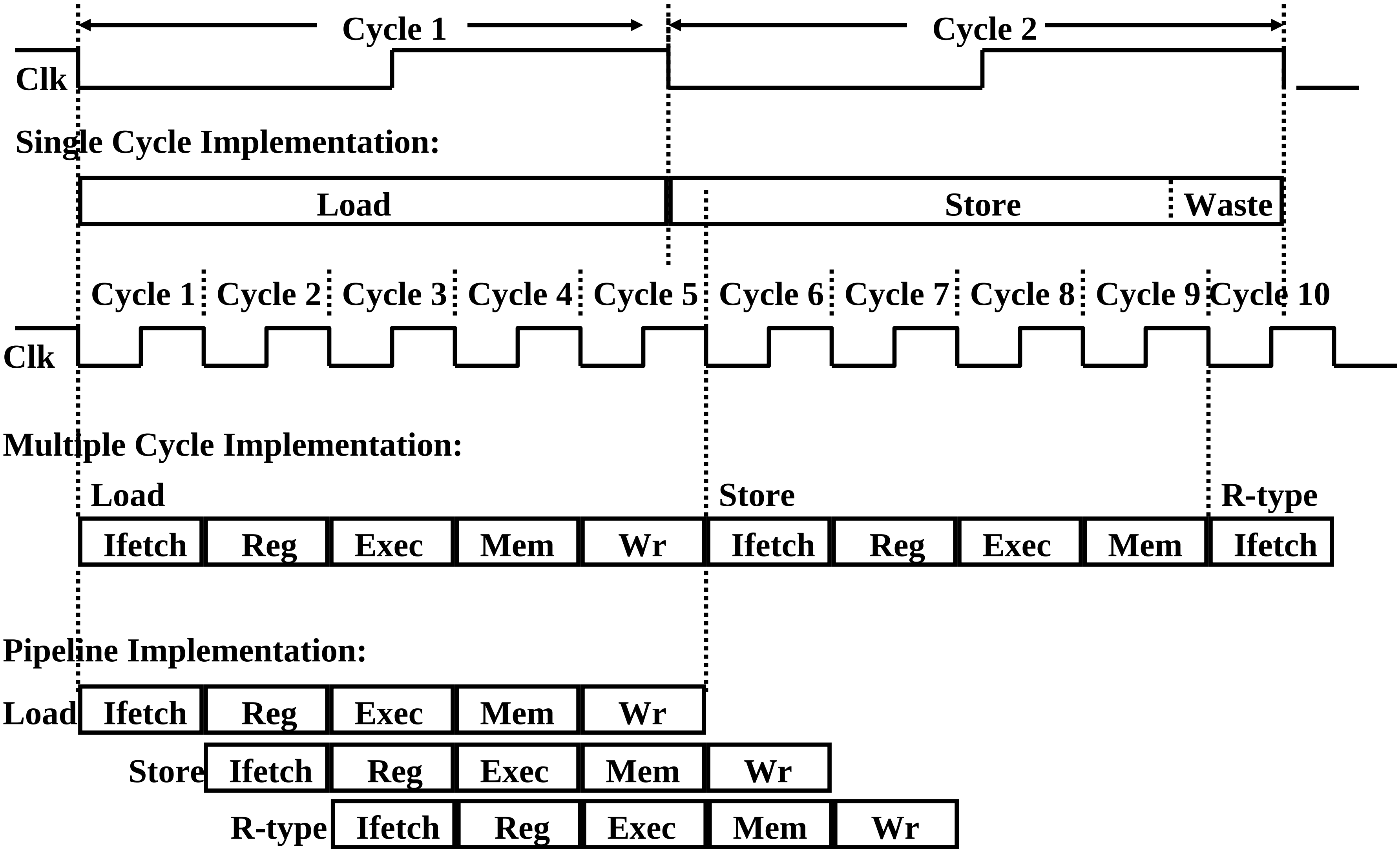
IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

Program Flow

Now we just have to make it work

Single Cycle, Multiple Cycle, vs. Pipeline



Cycle 1 Cycle 2

Clk

Single Cycle Implementation:

Load | Store | Waste

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9 Cycle 10

Clk

Multiple Cycle Implementation:

Load: Ifetch Reg Exec Mem Wr

Store: Ifetch Reg Exec Mem

R-type: Ifetch

Pipeline Implementation:

Load Ifetch Reg Exec Mem Wr

Store Ifetch Reg Exec Mem Wr

R-type Ifetch Reg Exec Mem Wr

Why Pipeline?

Suppose we execute 100 instructions

Single Cycle Machine

4500 200

45 ns/cycle x 1 CPI x 100 inst = ____ ns

45

Multicycle Machine

4000 160

10 ns/cycle x 4.0 CPI x 100 inst = ____ ns

Ideal pipelined machine

1040 80

10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = ____ ns

CPI for Pipelined Processors

Ideal pipelined machine

10 ns/cycle x (**1 CPI** x 100 inst + 4 cycle drain) = ____ ns

CPI in pipelined processor is “issue rate”. Ignore fill/drain, ignore latency.

Example: A processor wastes 2 cycles after every branch, and 1 after every load, during which it cannot issue a new instruction. If a program has 10% branches and 30% loads, what is the CPI on this program?

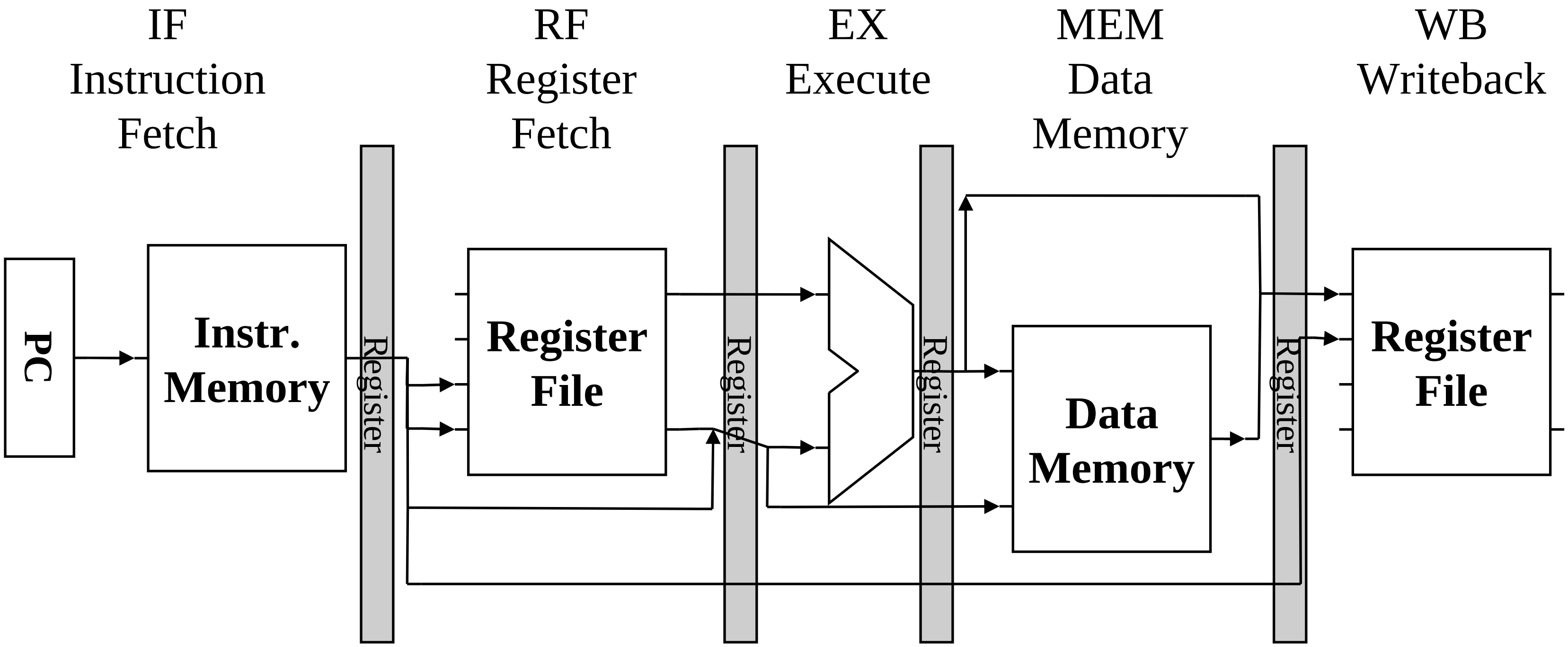
$$CPI = .6 \times 1 + (1+1) \times .3 + (1+2) \times .1 =$$

$$.6 + .6 + .3$$

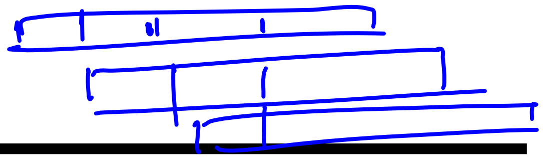
$$= 1.5$$

Pipelined Datapath

Divide datapath into multiple pipeline stages



Pipelined Control

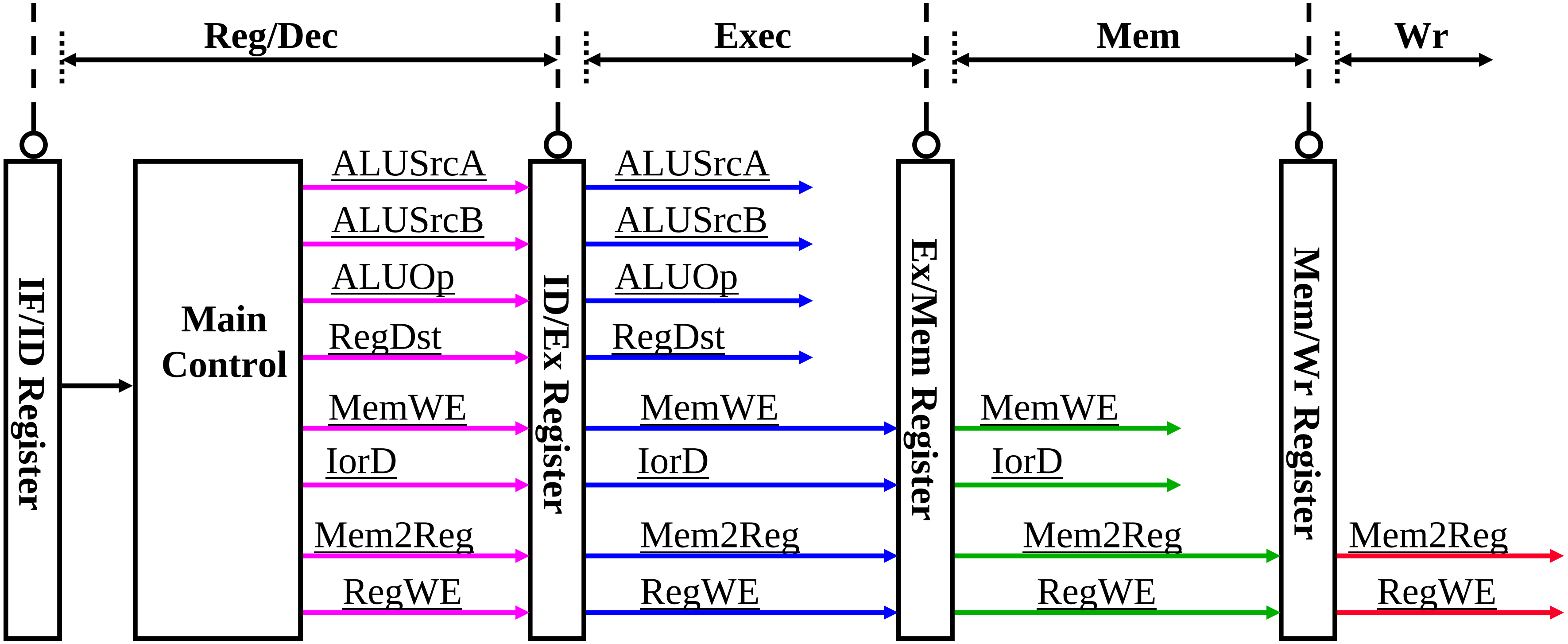


The Main Control generates the control signals during Reg/Dec

Control signals for Exec (ALUOp, ALUSrcA, …) are used 1 cycle later

Control signals for Mem (MemWE, IorD, …) are used 2 cycles later

Control signals for Wr (Mem2Reg, 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

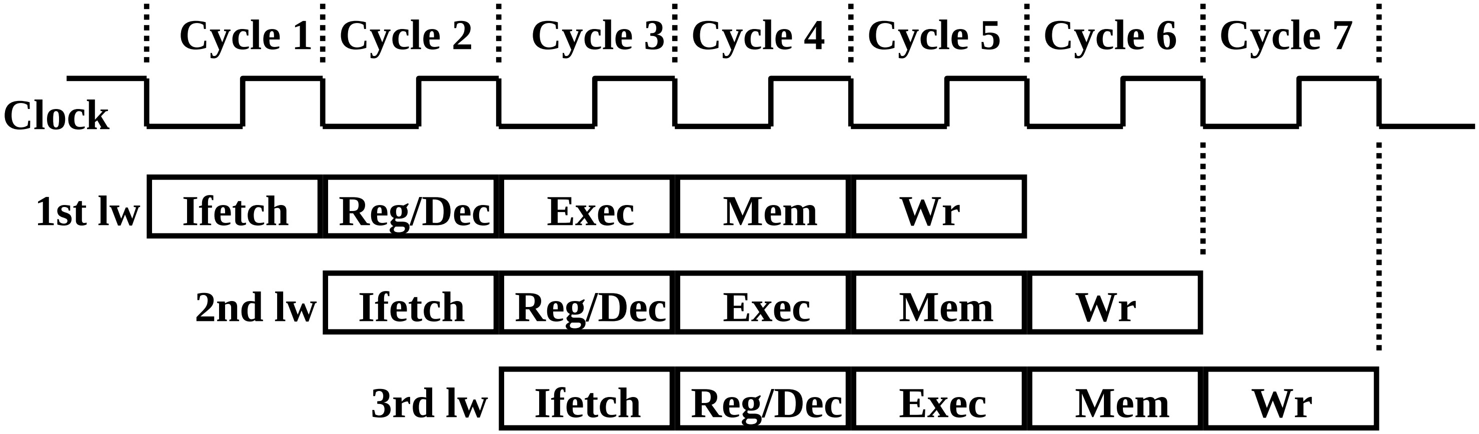
Can always resolve hazards by **waiting**

- 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
1st lw	Ifetch	Reg/Dec	Exec	Mem	Wr		
2nd lw		Ifetch	Reg/Dec	Exec	Mem	Wr	
3rd lw			Ifetch	Reg/Dec	Exec	Mem	Wr

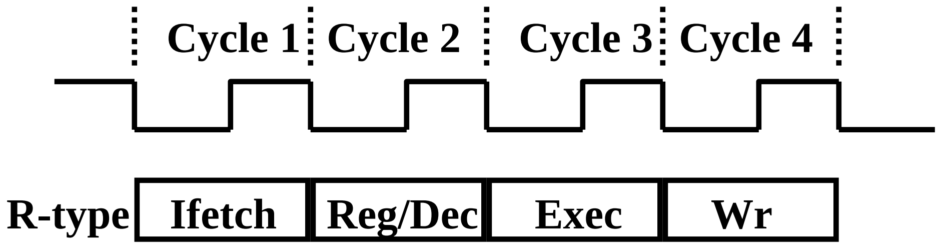
The Four Stages of R-type

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

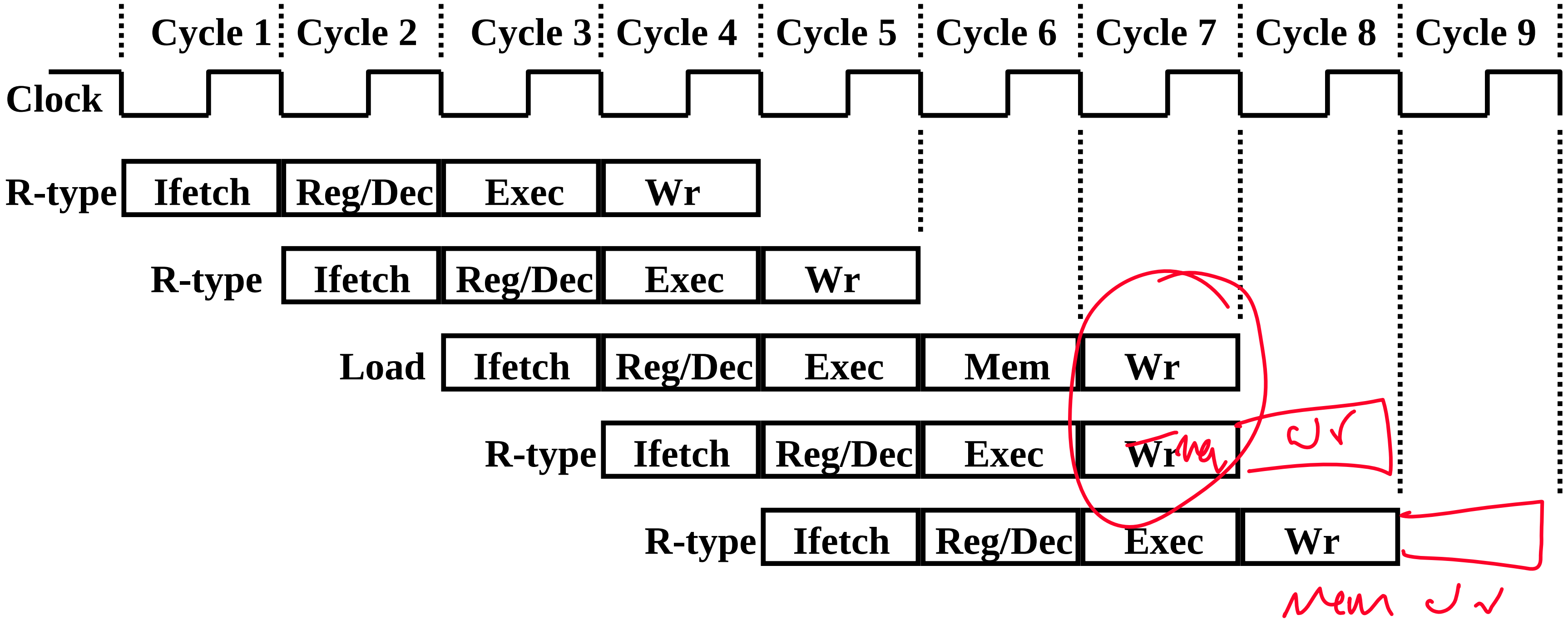


Cycle 1 Cycle 2 Cycle 3 Cycle 4

R-type Ifetch Reg/Dec Exec Wr

Structural Hazard

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



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 Wr

R-type Ifetch Reg/Dec Exec Wr

Load Ifetch Reg/Dec Exec Mem Wr

R-type Ifetch Reg/Dec Exec ~~Wr~~ mem wr

R-type Ifetch Reg/Dec Exec Wr

mem wr

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

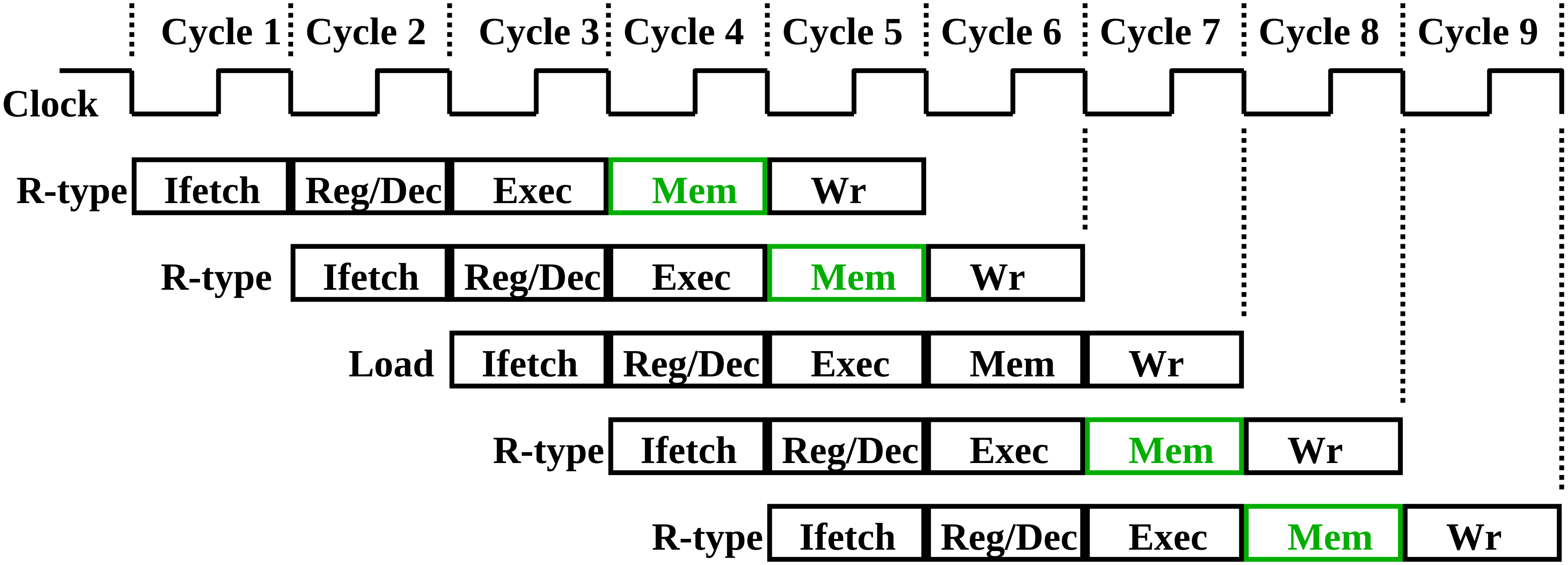
	1	2	3	4
R-type	**Ifetch**	**Reg/Dec**	**Exec**	**Wr**

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

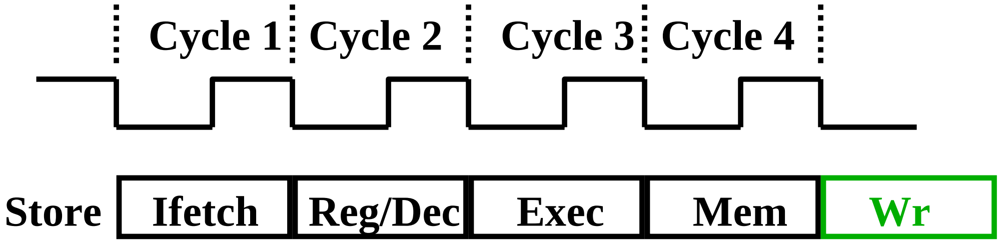
Reg/Dec: Register Fetch and Instruction Decode

Exec: Calculate the memory address

Mem: Write the data into the Data Memory

Wr: NOOP

Compatible with Load & R-type instructions



The Stages of Branch

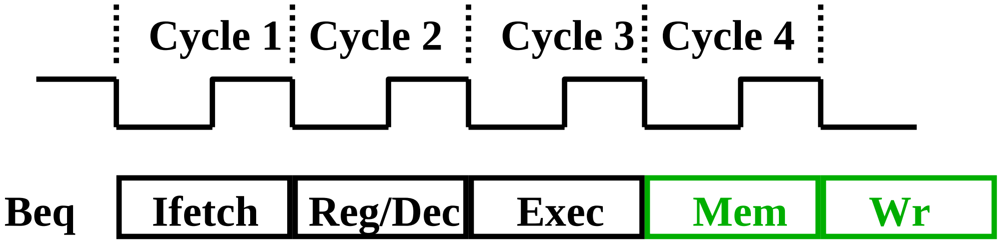
Ifetch: Fetch the instruction from the Instruction Memory

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

Exec: Test condition & update the PC

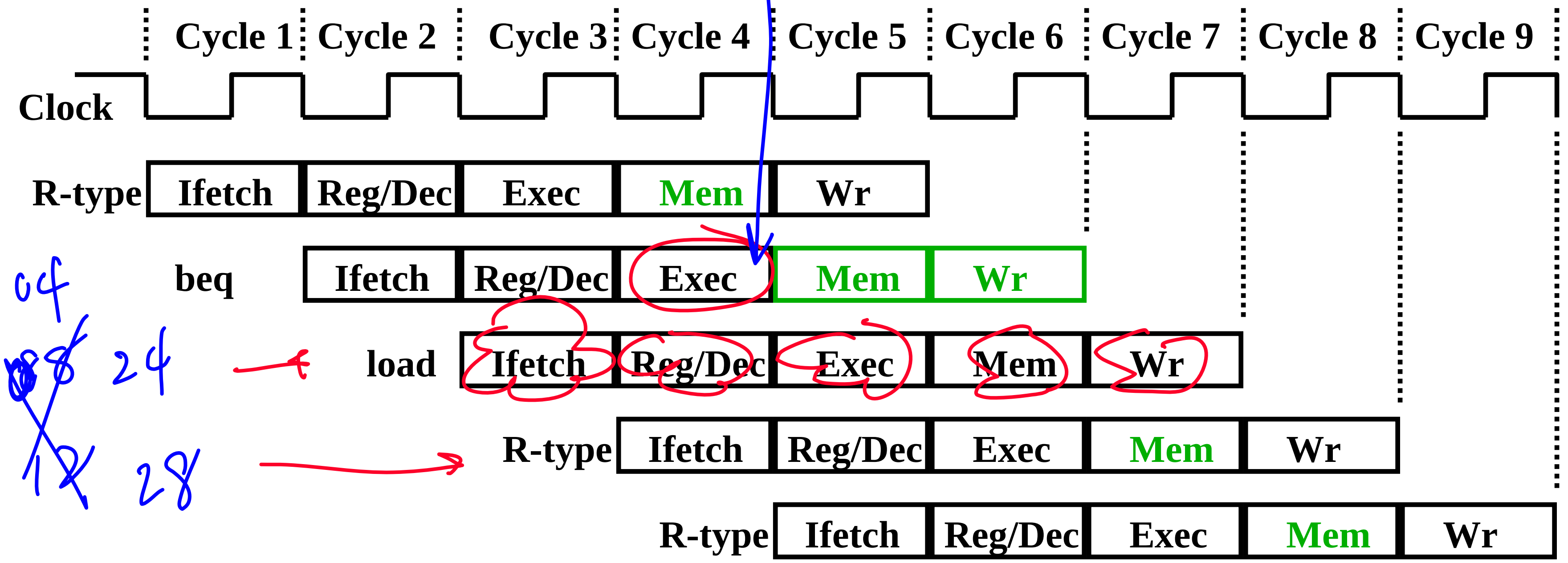
Mem: NOOP

Wr: NOOP



Control Hazard

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



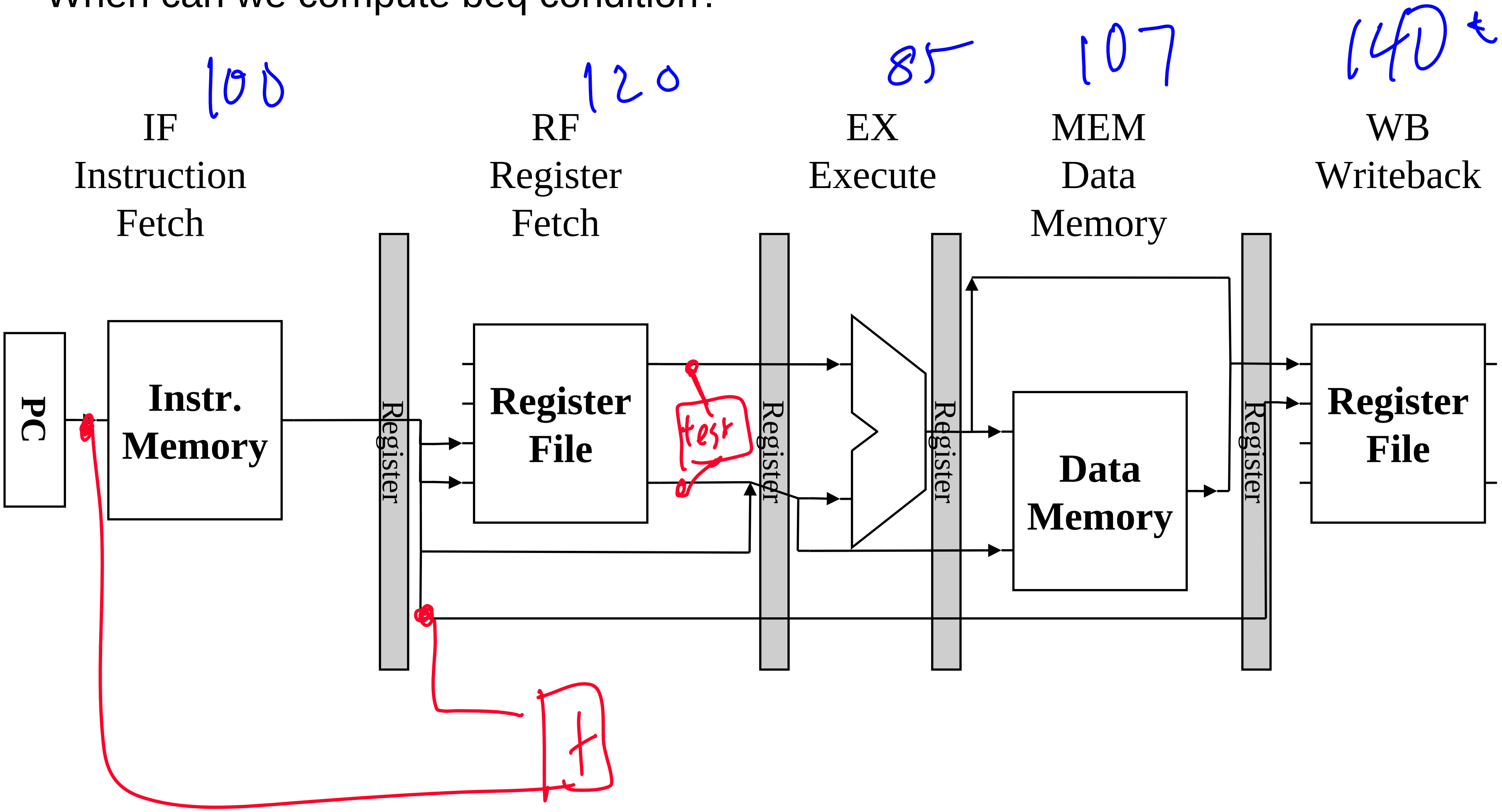
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr				
beq		Ifetch	Reg/Dec	Exec	Mem	Wr			
load			Ifetch	Reg/Dec	Exec	Mem	Wr		
R-type				Ifetch	Reg/Dec	Exec	Mem	Wr	
R-type					Ifetch	Reg/Dec	Exec	Mem	Wr

Accelerate Branches

beq $a0, $b0, Foo

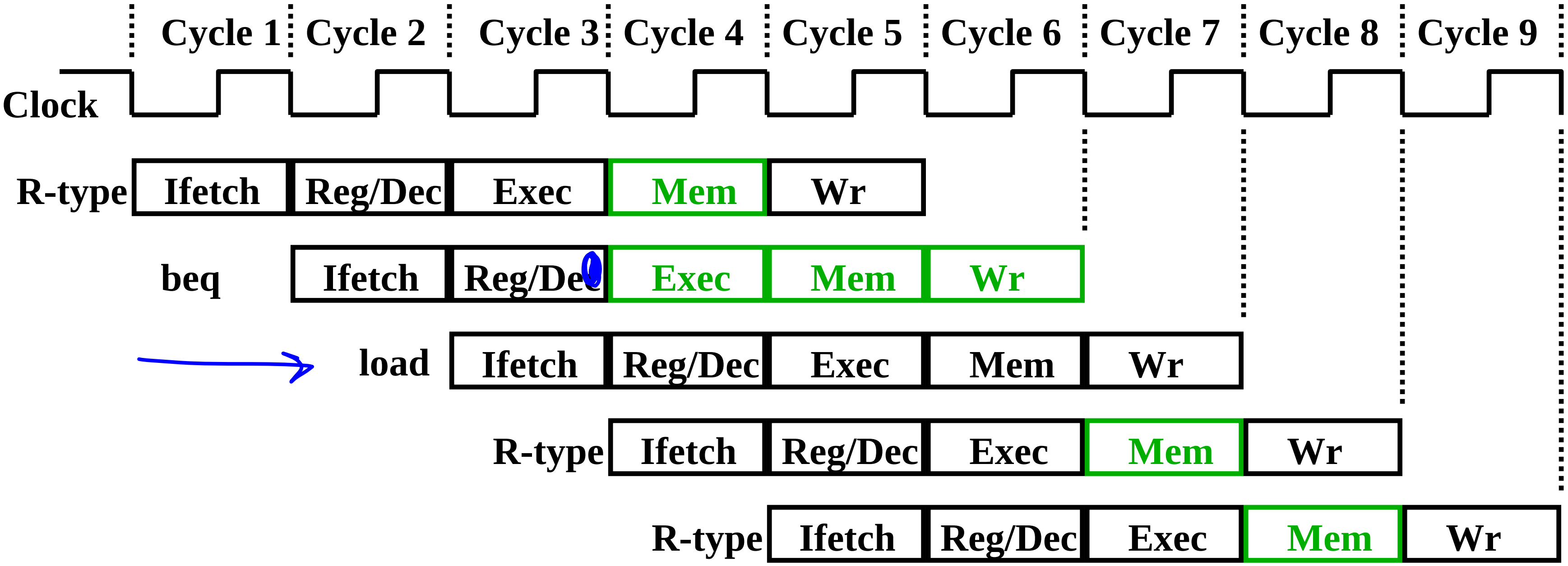
When can we compute branch target address?

When can we compute beq condition?



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

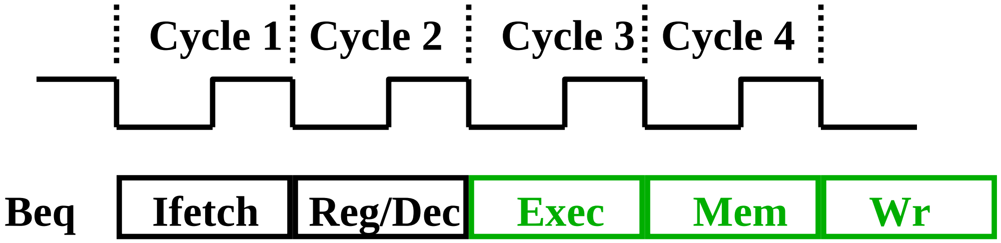
R-type Ifetch Reg/Dec Exec Mem Wr

beq 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

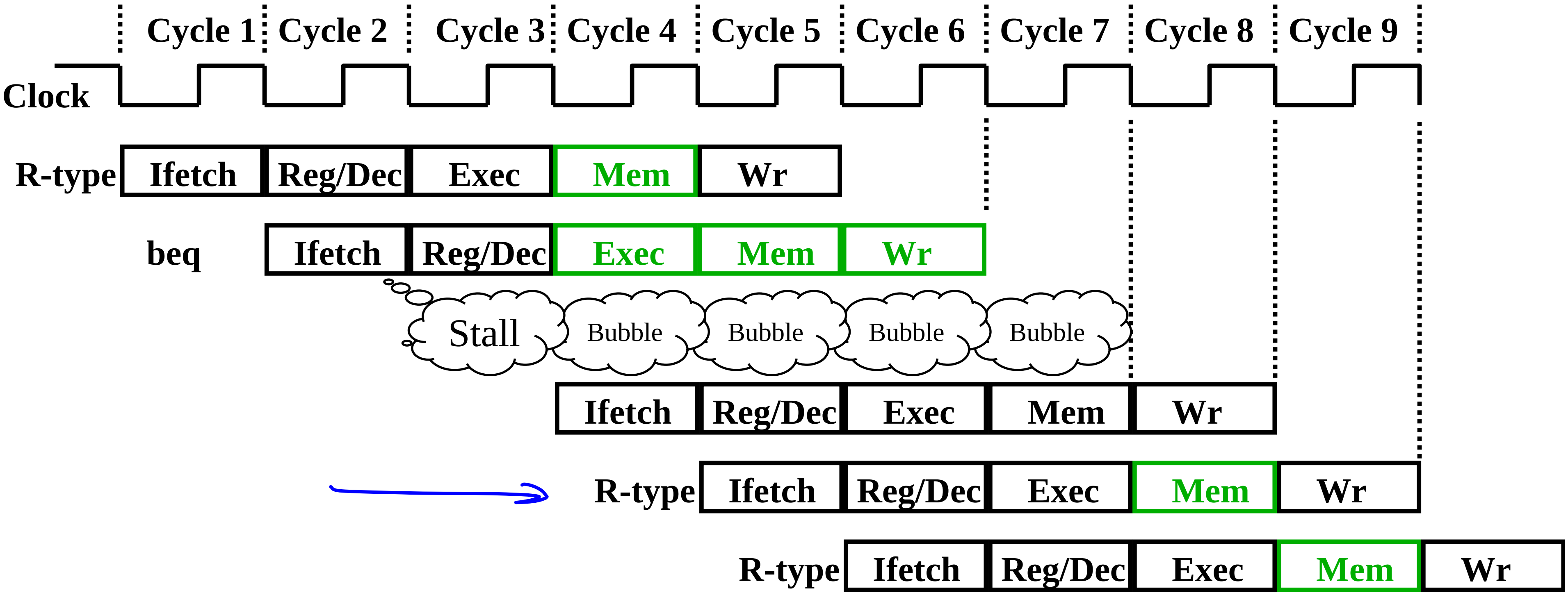


Cycle 1 Cycle 2 Cycle 3 Cycle 4

Beq Ifetch Reg/Dec Exec Mem Wr

Solution #1: Stall

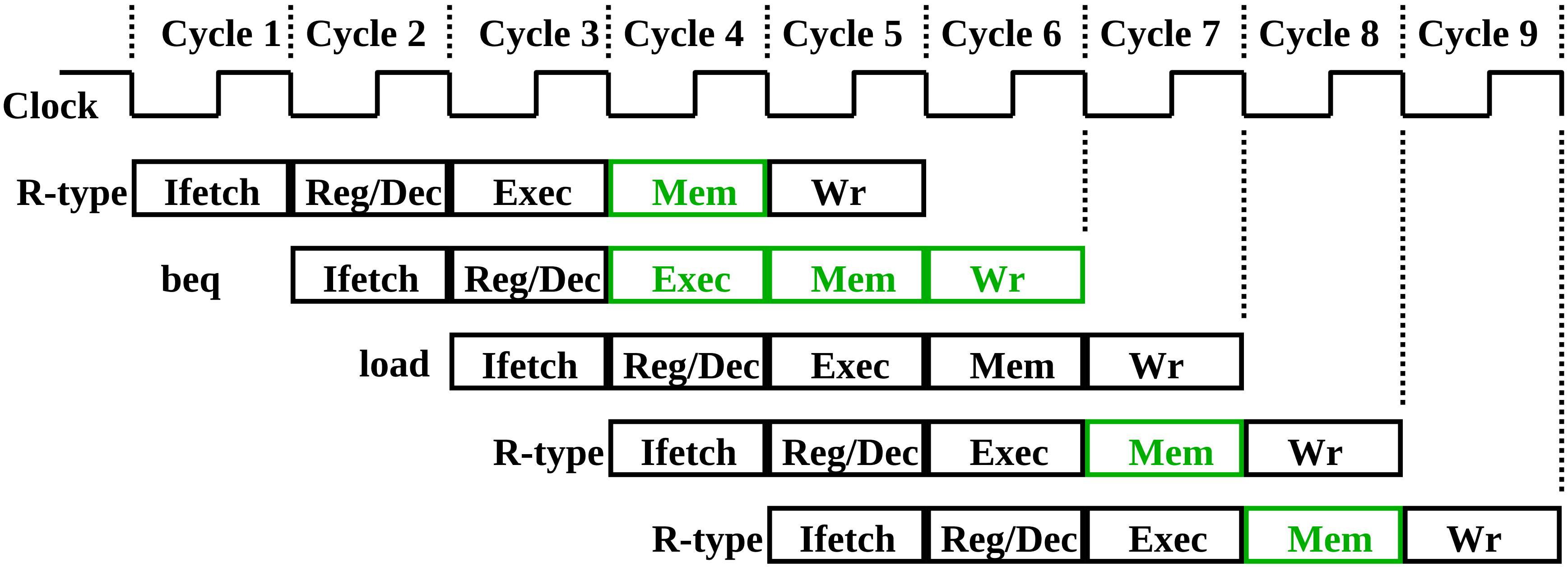
Delay loading next instruction, load no-op instead



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				
beq		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 $t1, $t0, $t0
beq $t2, $t3, FOO
```

~~sub $t2, $t0, $t3~~

```
add $t1, $t0, $t0
beq $t1, $t3, FOO
```

sub $t2, $t0, $t3

```
  add $t1, $t0, $t0
  beq $t1, $t3, FOO

  add $t1, $t3, $t3
  …
FOO:
  add $t1, $t2, $t0
```

```
add $t1, $t0, $t0
beq $t1, $t3, FOO
```

add $a, $t0, $t0

add

beq

FOO

beq

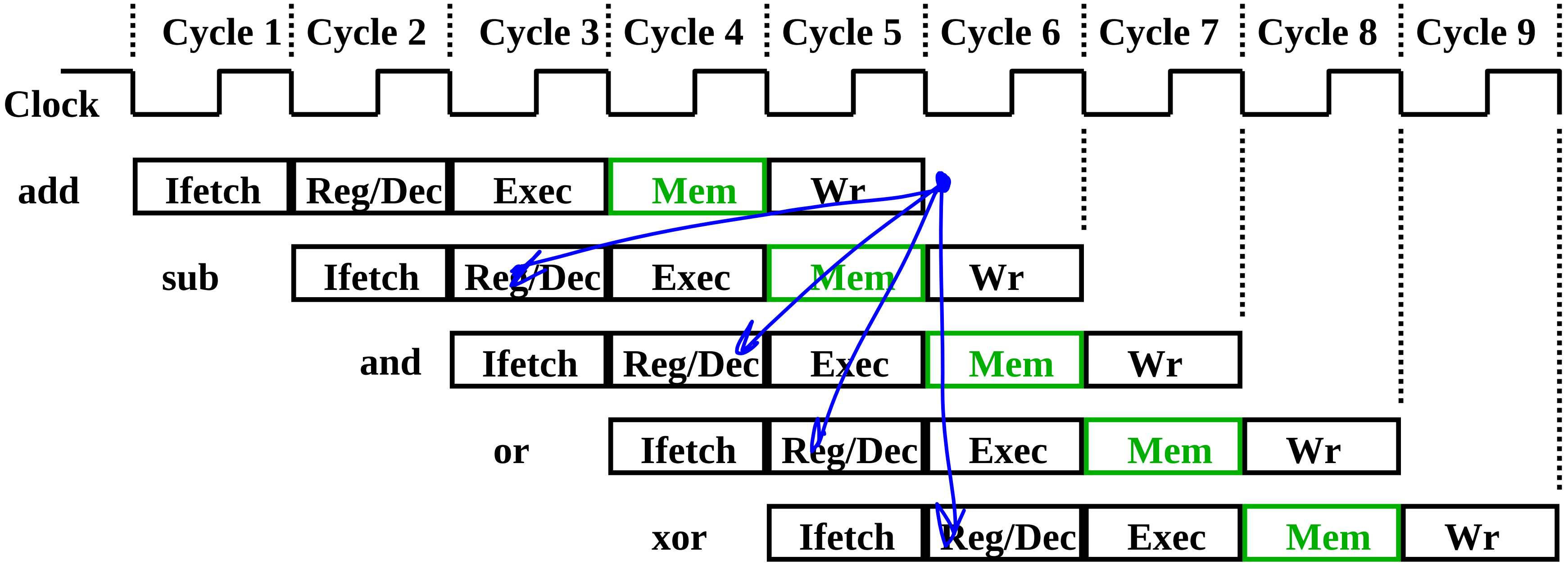
TOP:

beq

Data Hazards

Consider the following code:

```
add $t0, $t1, $t2
sub $t3, $t0, $t4
and $t5, $t0, $t7
or $t8, $t0, $s0
xor $s1, $t0, $s2
```



Design Register File Carefully

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

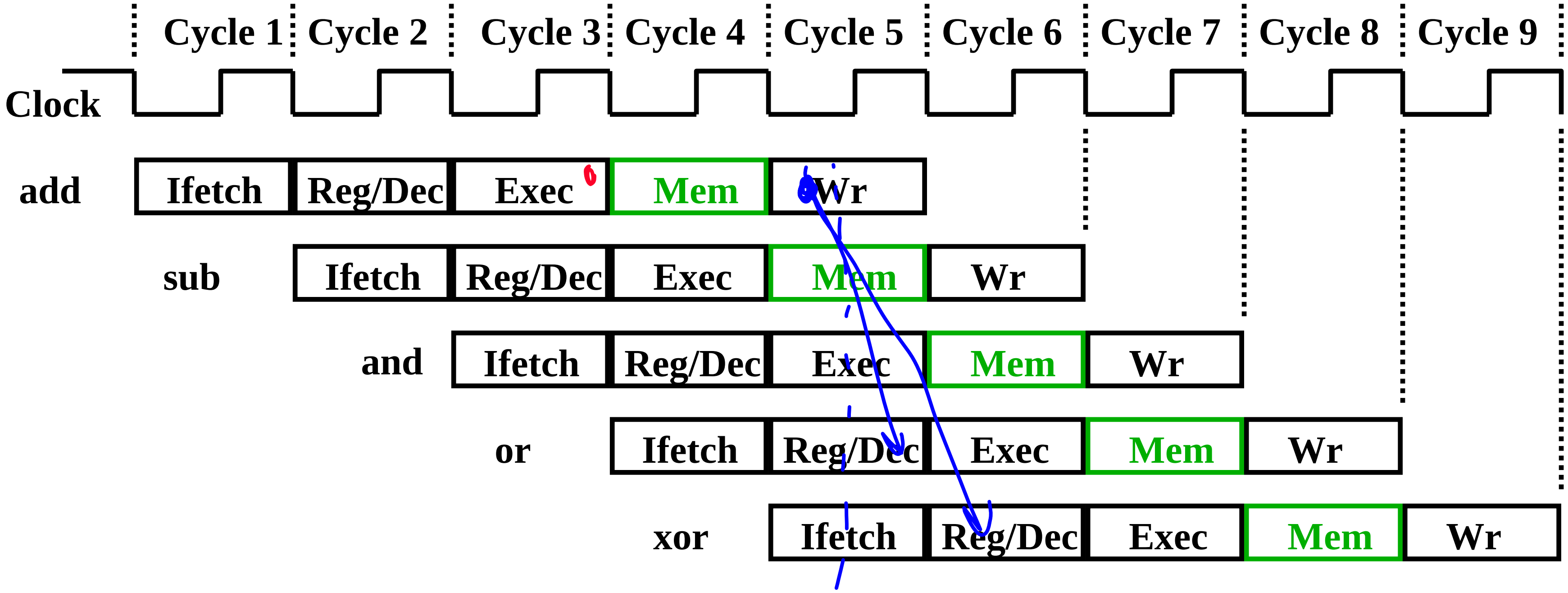
add $t0, $t1, $t2

sub $t3, $t0, $t4

and $t5, $t0, $t7

or $t8, $t0, $s0

xor $s1, $t0, $s2



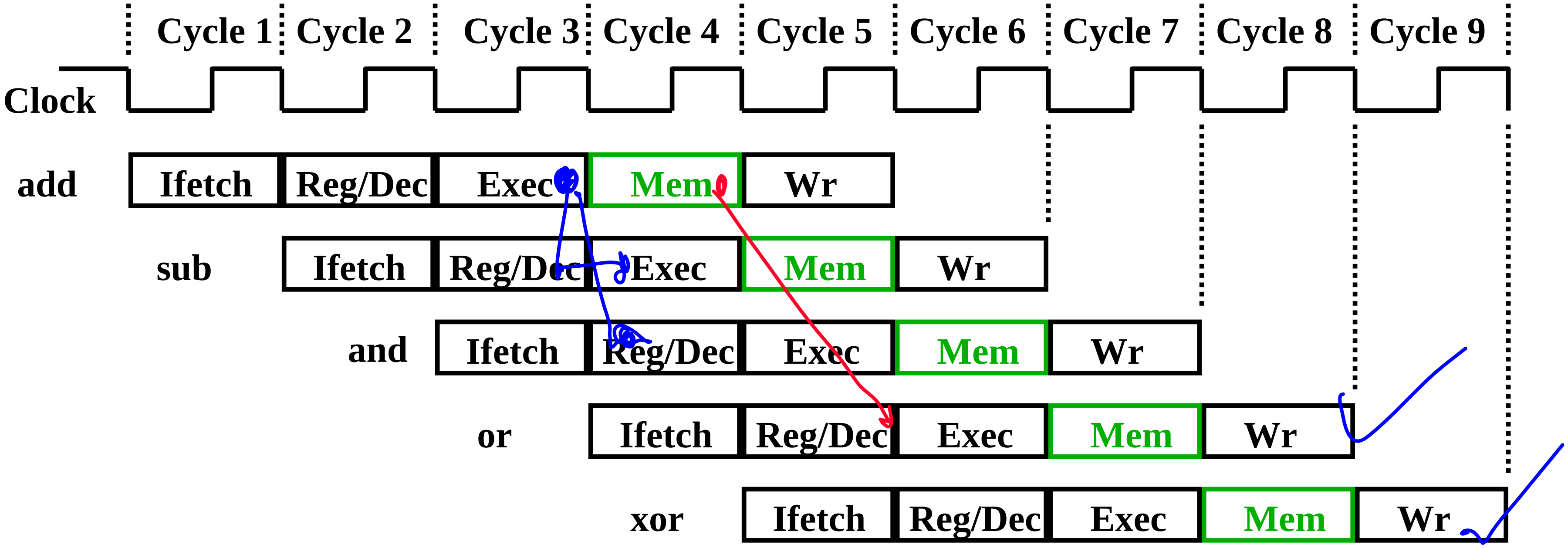
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
add	Ifetch	Reg/Dec	Exec	Mem	Wr				
sub		Ifetch	Reg/Dec	Exec	Mem	Wr			
and			Ifetch	Reg/Dec	Exec	Mem	Wr		
or				Ifetch	Reg/Dec	Exec	Mem	Wr	
xor					Ifetch	Reg/Dec	Exec	Mem	Wr

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 $t0, $t1, $t2
sub $t3, $t0, $t4
and $t5, $t0, $t7
or $t8, $t0, $s0
xor $s1, $t0, $s2
```



	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		
or				Ifetch	Reg/Dec	Exec	Mem	Wr	
xor					Ifetch	Reg/Dec	Exec	Mem	Wr

Forwarding (cont.)

add 1,2,3

sub 1,3 4

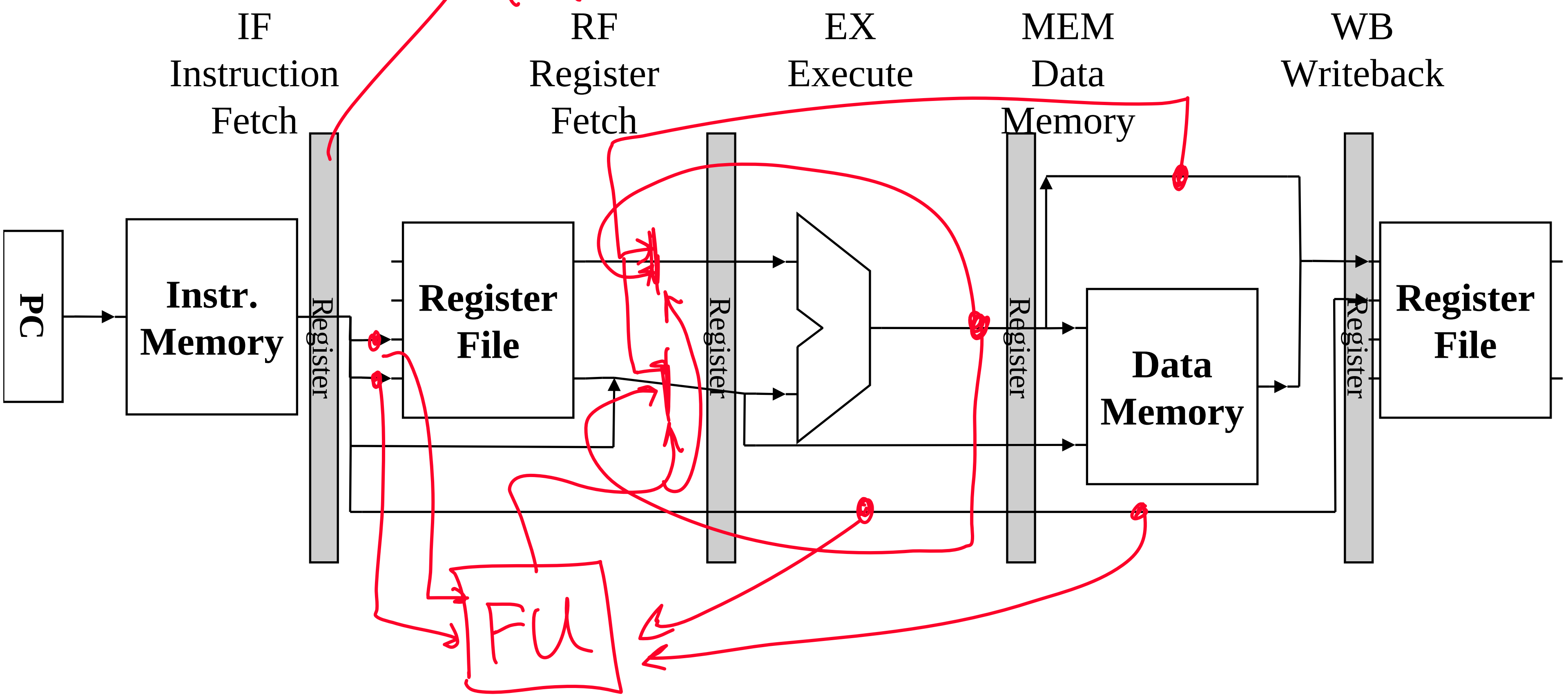
xor 2,1,6

Requires values from last two ALU operations.

Remember destination register for operation.

Compare sources of current instruction to destinations of previous 2.

IF/RF



Data Hazards on Loads

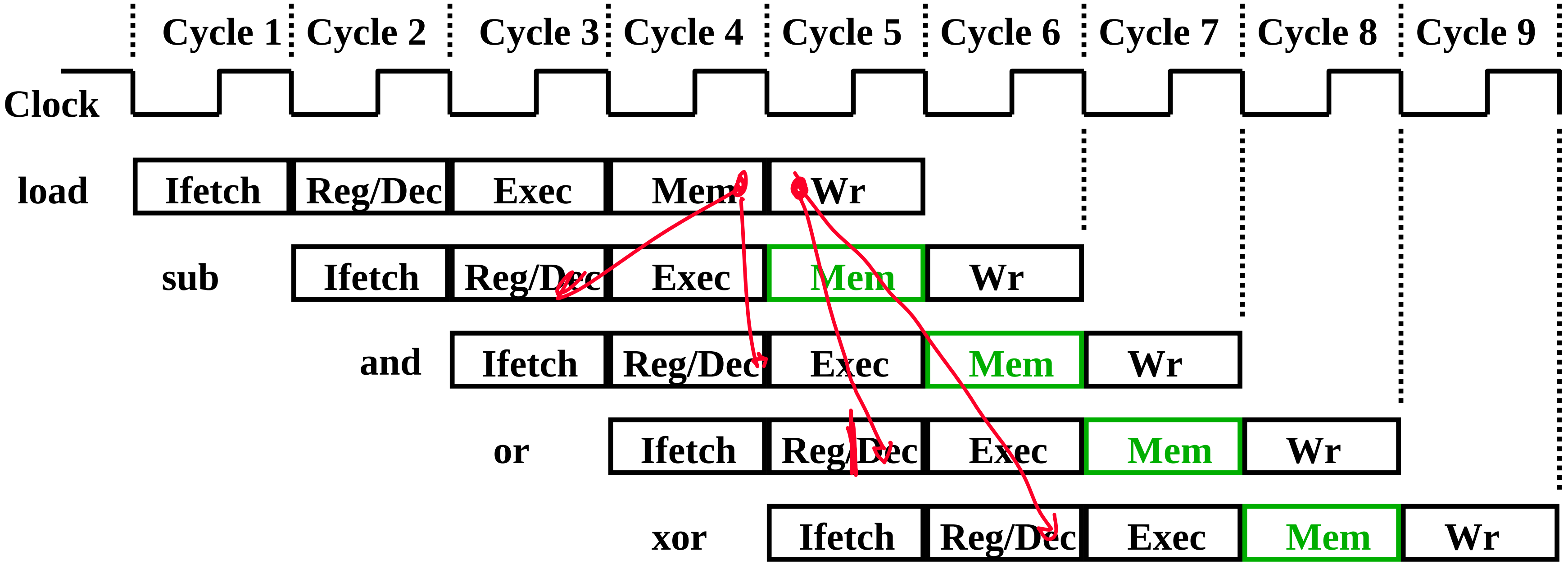
load $t0, 0($t1)

sub $t3, $t0, $t4 BAD.

and $t5, $t0, $t7 fixed

or $t8, $t0, $s0 fixed with new reg file.

xor $s1, $t0, $s2 OK


Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
Clock
load Ifetch Reg/Dec Exec Mem Wr
sub Ifetch Reg/Dec Exec Mem Wr
and Ifetch Reg/Dec Exec Mem Wr
or Ifetch Reg/Dec Exec Mem Wr
xor Ifetch Reg/Dec Exec Mem Wr

Data Hazards on Loads

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.

Pipelined CPI, cycle time

CPI, assuming compiler can fill 50% of delay slots

Instruction Type	Type Cycles	Type Frequency	Cycles * Freq
ALU	1.0	50%	0.5
Load	1.5	20%	0.3
Store	1.0	10%	0.1
Branch	1.5	20%	0.3
		CPI:	1.2

Pipelined: cycle time = 1ns. Delay for 1M instr:

Multicycle: CPI = 4.0, cycle time = 1ns. Delay for 1M instr:

Single cycle: CPI = 1.0, cycle time = 4.5ns. Delay for 1M instr:

Pipelined CPU Summary

Stops waste (of hardware) = better utilization.

Can do multiple pipelines.

Lots of control = AREA

dealing w/ hazards.

Faster clock rates