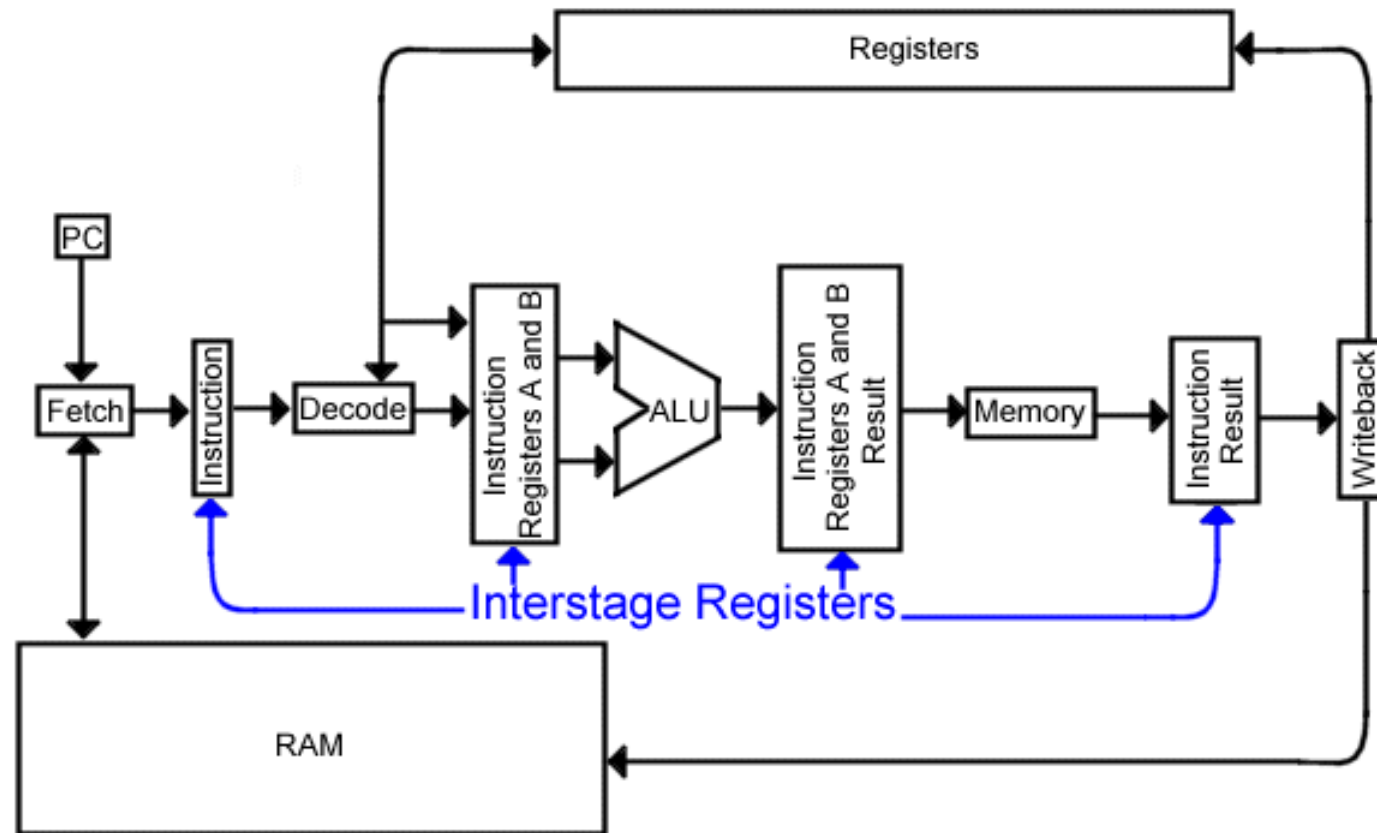
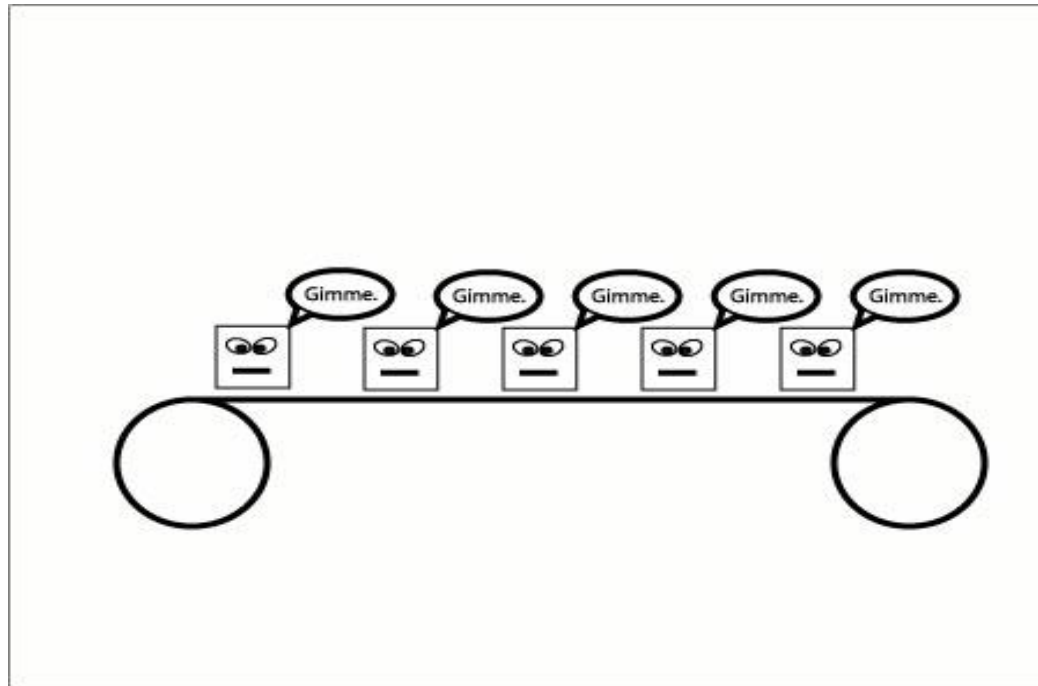


High Speed Design: Pipelining



Pipelining

- *Pipelining* is an implementation technique whereby multiple instructions are overlapped in execution. Pipelining is the **key** implementation technique that is currently used to make high-performance systems.



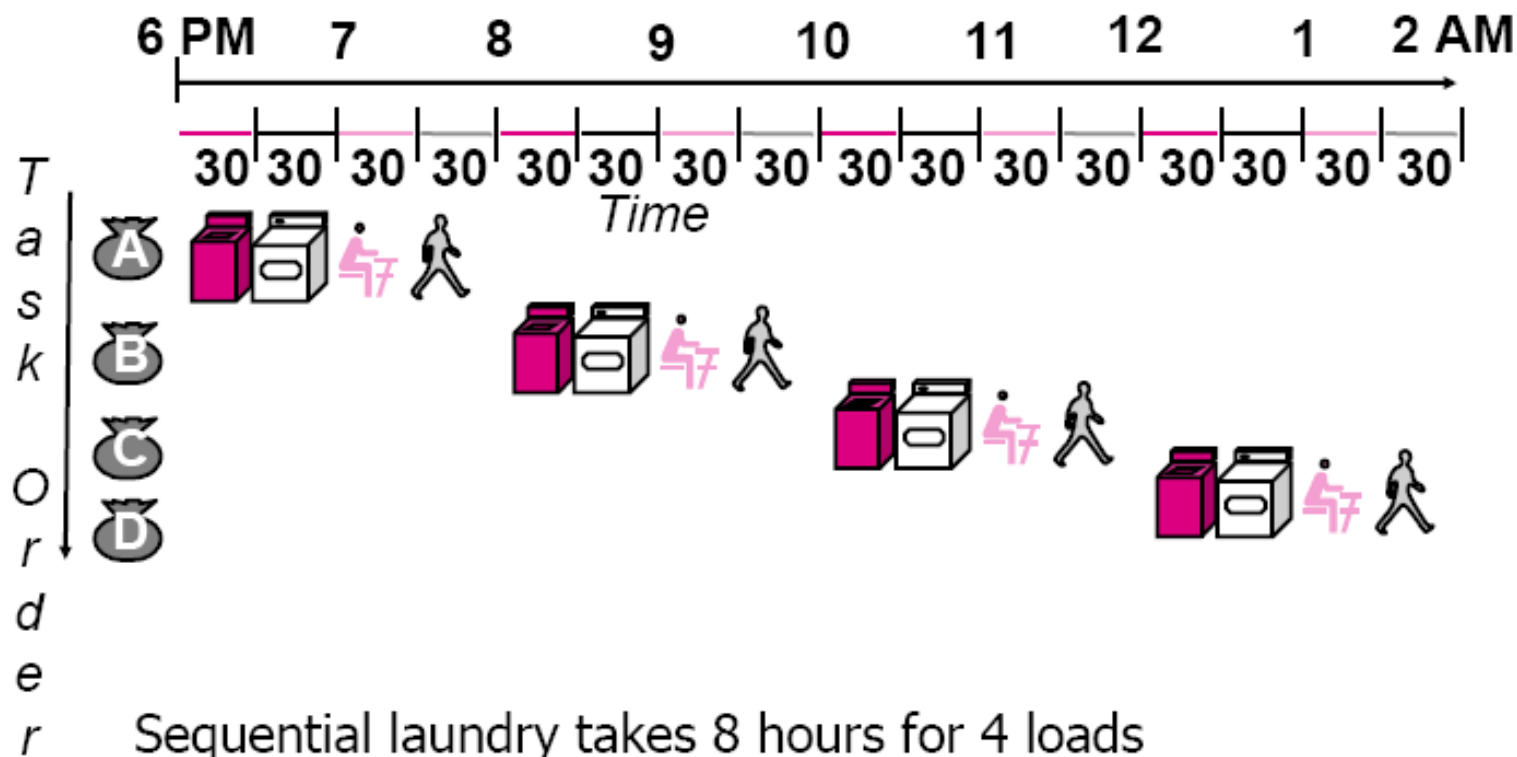
Pipelining in real life

Laundry Example

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers

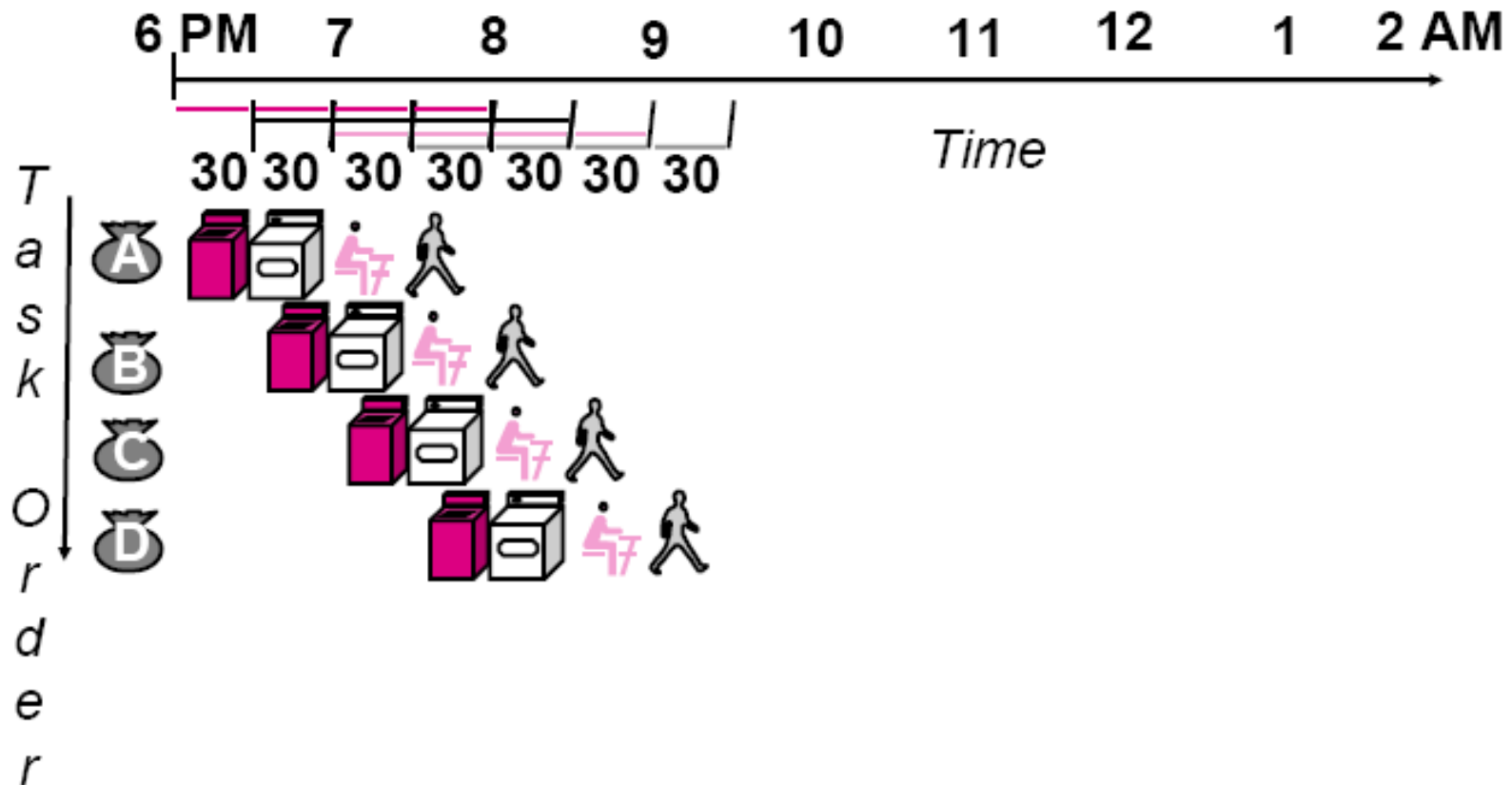


Sequential processing



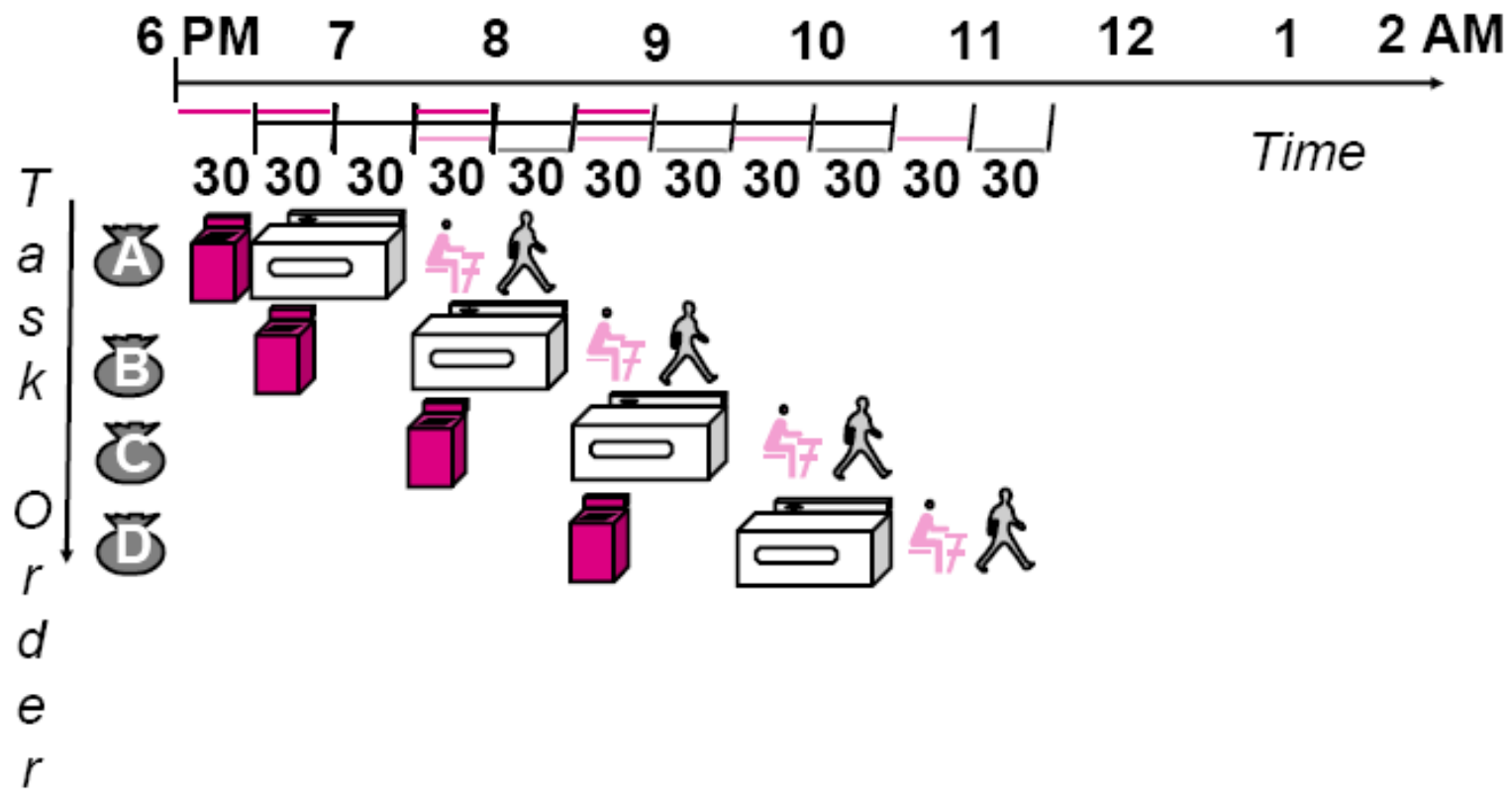
If they learned pipelining, how long would laundry take?

Pipelined processing



- Pipelined laundry takes 3.5 hours for 4 loads!

Unbalanced Pipeline



5.5 Hours. What is going on here?



Pipelining Principles

- Pipelining does not help the *latency* of a single task, it helps the *throughput* of the entire workload.
- The pipeline rate is limited by the *slowest* pipeline stage.
- *Multiple* tasks operating simultaneously.
- Potential speedup = *number of pipe stages*
- Unbalanced lengths of pipe stages reduces speedup.
- Time to “fill” pipeline and time to “drain” it reduces speedup.



Hardware Pipelining

- Pipelining is an implementation technique that exploits parallelism among instructions in a sequential instruction stream.
- A major advantage of pipelining over “parallel processing” is that it is not visible to the programmer.
- In a computer system, each pipeline stage completes a part of the instruction being executed.
- The time required between moving an instruction one step down the pipeline is a *machine (clock) cycle*.
- The length of a clock cycle is determined by the time required for the slowest stage to proceed.

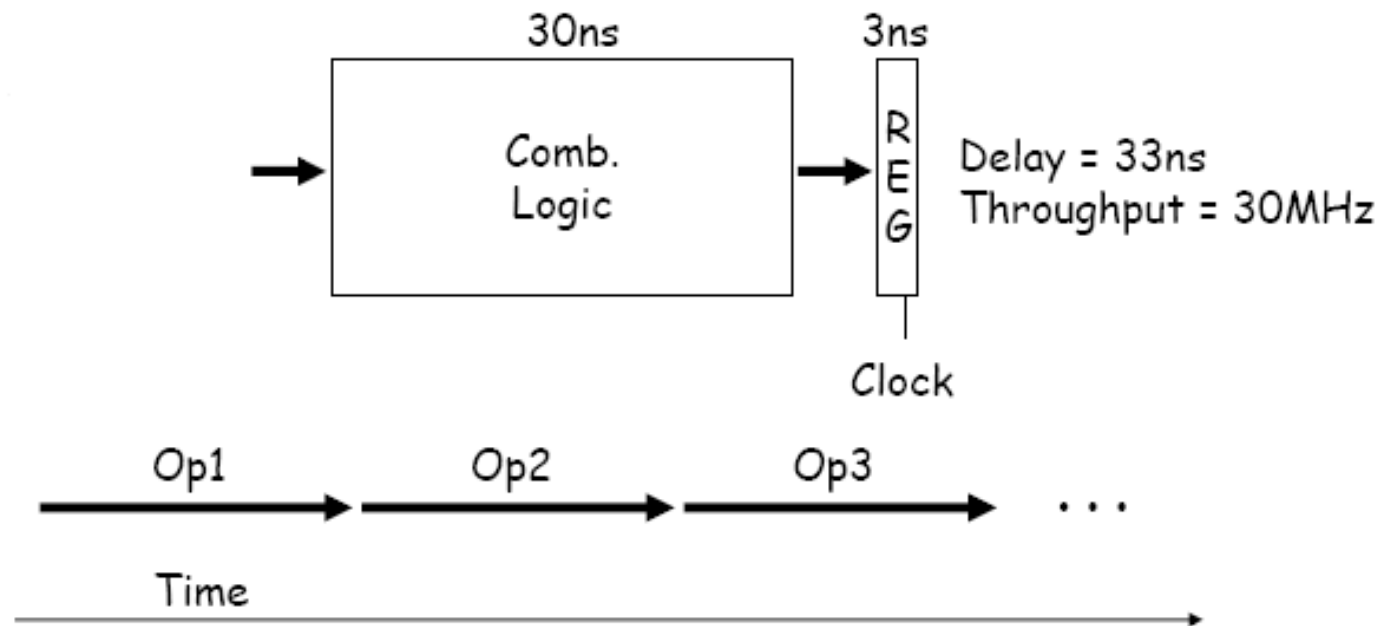
Hardware Pipelining

- The hardware architect should try to *balance* the length of each pipeline stage.
- In a perfect pipeline, the time per instruction on the pipeline computer is:

$$\frac{\text{Time per instruction on unpipelined computer}}{\text{Number of pipe stages}}$$

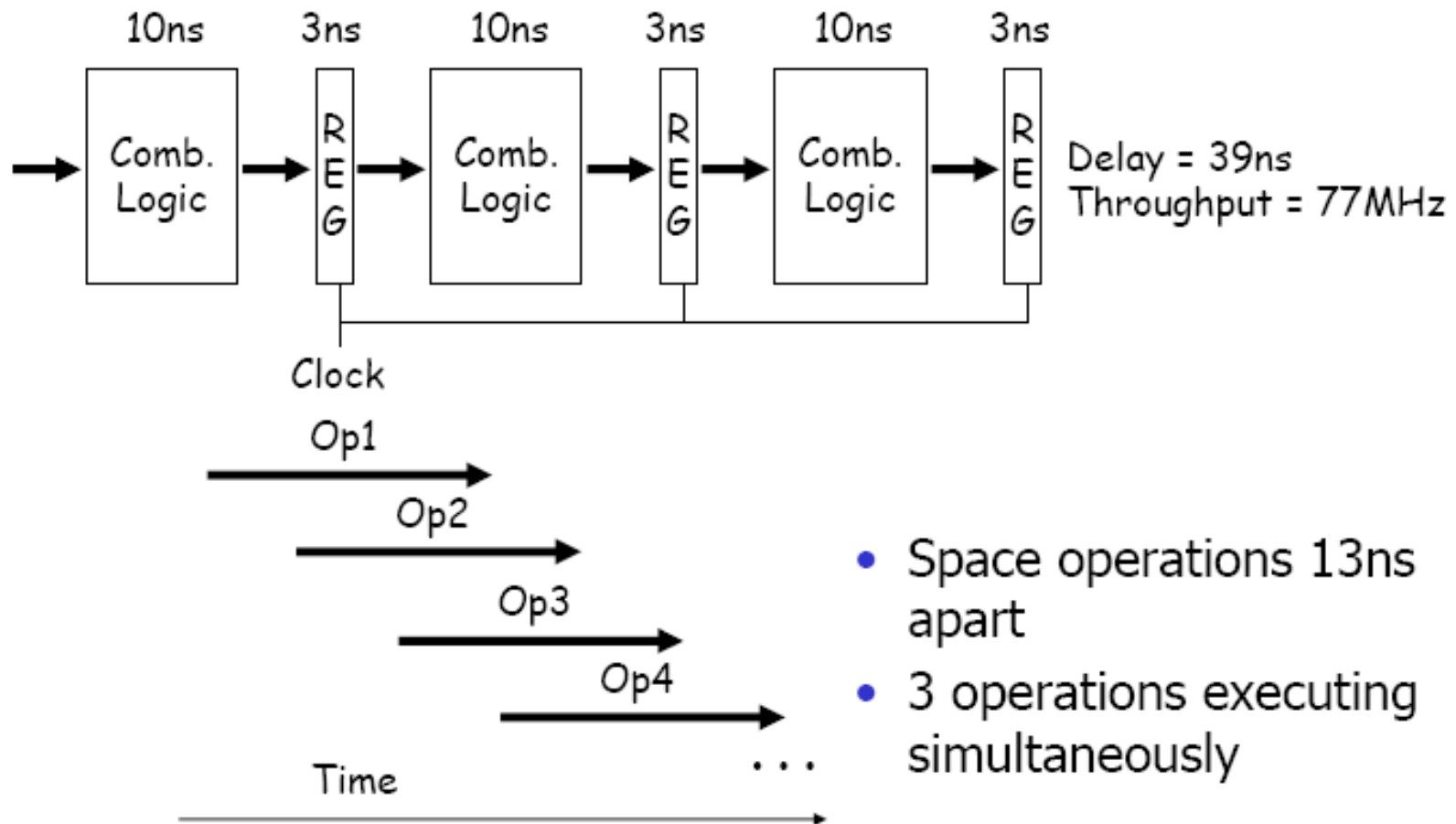
- If we have n stages, we could speedup the execution n times — speedup = n .
- In practice, the pipeline stages will not be perfectly balanced - and there are additional over-heads. But we can get *close* to the ideal case.

Unpipelined System

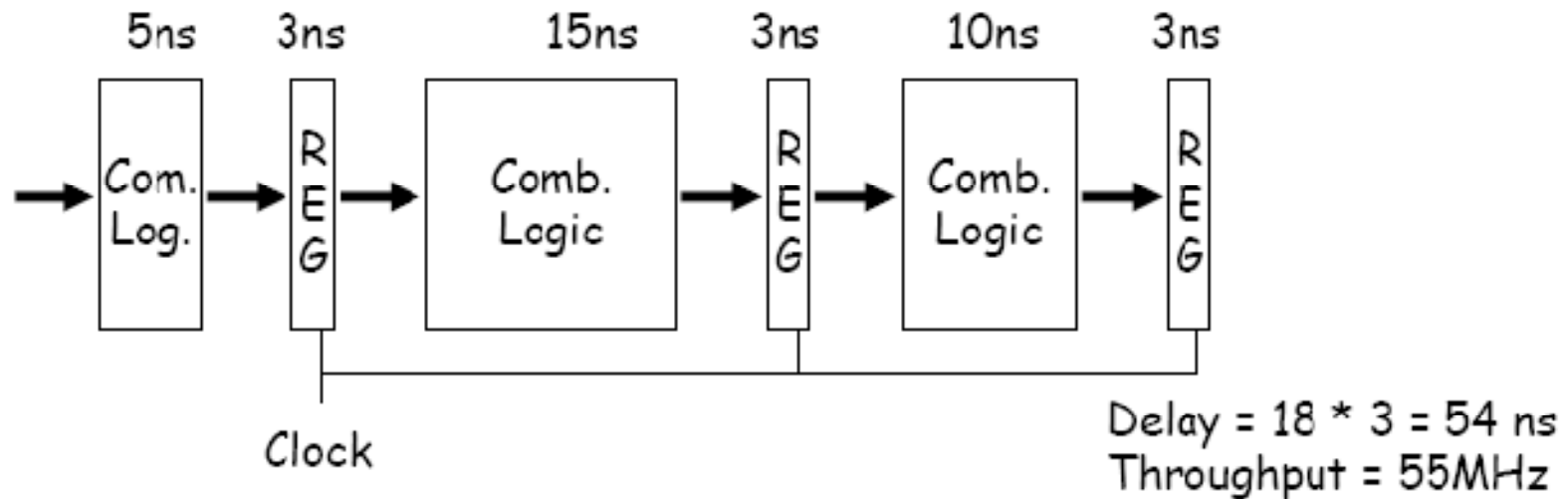


- One operation must complete before next can begin
- Operations spaced 33ns apart

3 Stage Pipelined System

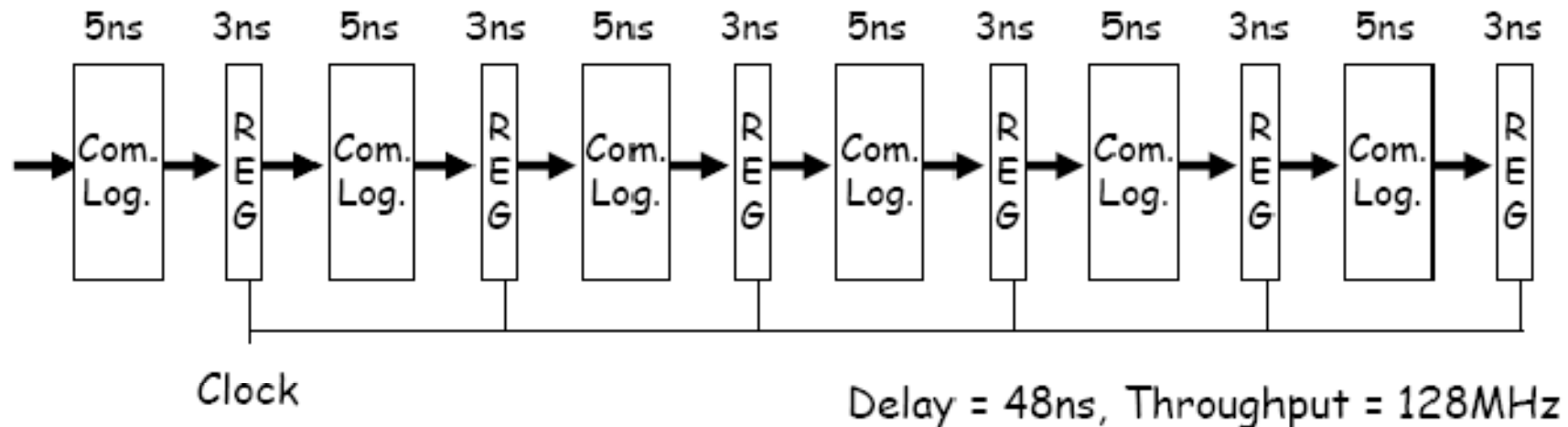


3 Stage Unbalanced Pipelined System



- Throughput limited by slowest stage
Delay determined by clock period * number of stages
- Must attempt to balance stages

Deep Pipelined System



- Diminishing returns as we add more pipeline stages
- Register delays become limiting factor
 - Increased latency
 - Small throughput gains

Pipeline in real system- CPU

CPU Life Cycle:

- ▶ At most one of the following can occur within one clock cycle:
 - One ALU operation
 - One register file access (1 write and 2 reads)
 - One memory access (Actually will take multiple clock cycles before we get a cache)
- ▶ Our execution stages will be separated into 5 cycles
 - Instruction fetch
 - Fetch new instruction from memory, compute next PC value
 - Performed for all instructions
 - Decode
 - Fetch register values from register file, compute branch address
 - Performed for all instructions



Pipeline in real system- CPU

- Execute
 - Perform A/L/S operation for A/L/S R and I-type instructions
 - Compute address for load and store instructions
 - Determine if branch is taken for branch instructions
 - Jump for jump instructions
 - Link for branch-and-link and jump-and-link instructions
- Memory
 - Access memory for load and store instructions (skip for all others)
- Write back
 - Write register result back to register file for A/L/S/load instructions (skip for all others)

Pipeline in real system – MIPS CPU

- Instructions in a MIPS processor are executed in at most five clock cycles as follows:

1. *Instruction fetch cycle (IF)*

`IR = Memory[PC]`

`PC = PC + 4`

2. *Instruction decode & register fetch cycle (ID)*

`A = Reg[IR[25-21]]`

`B = Reg[IR[20-16]]`

`Target = PC + (sign-extend(`

`IR[15-0]) << 2)`

Pipeline in real system – MIPS CPU

3. *Execution, memory address computation, or branch completion (EX)*

■ Memory reference

`ALUoutput = A + sign-
extend(IR[15-0])`

■ Arithmetic-logical instruction (R-type)

`ALUoutput = A op B
Branch If (A == B) PC = Target`

4. *Memory access or R-type instruction completion cycle (MEM)*

■ Memory reference

`memory-data =
Memory[ALUoutput]`

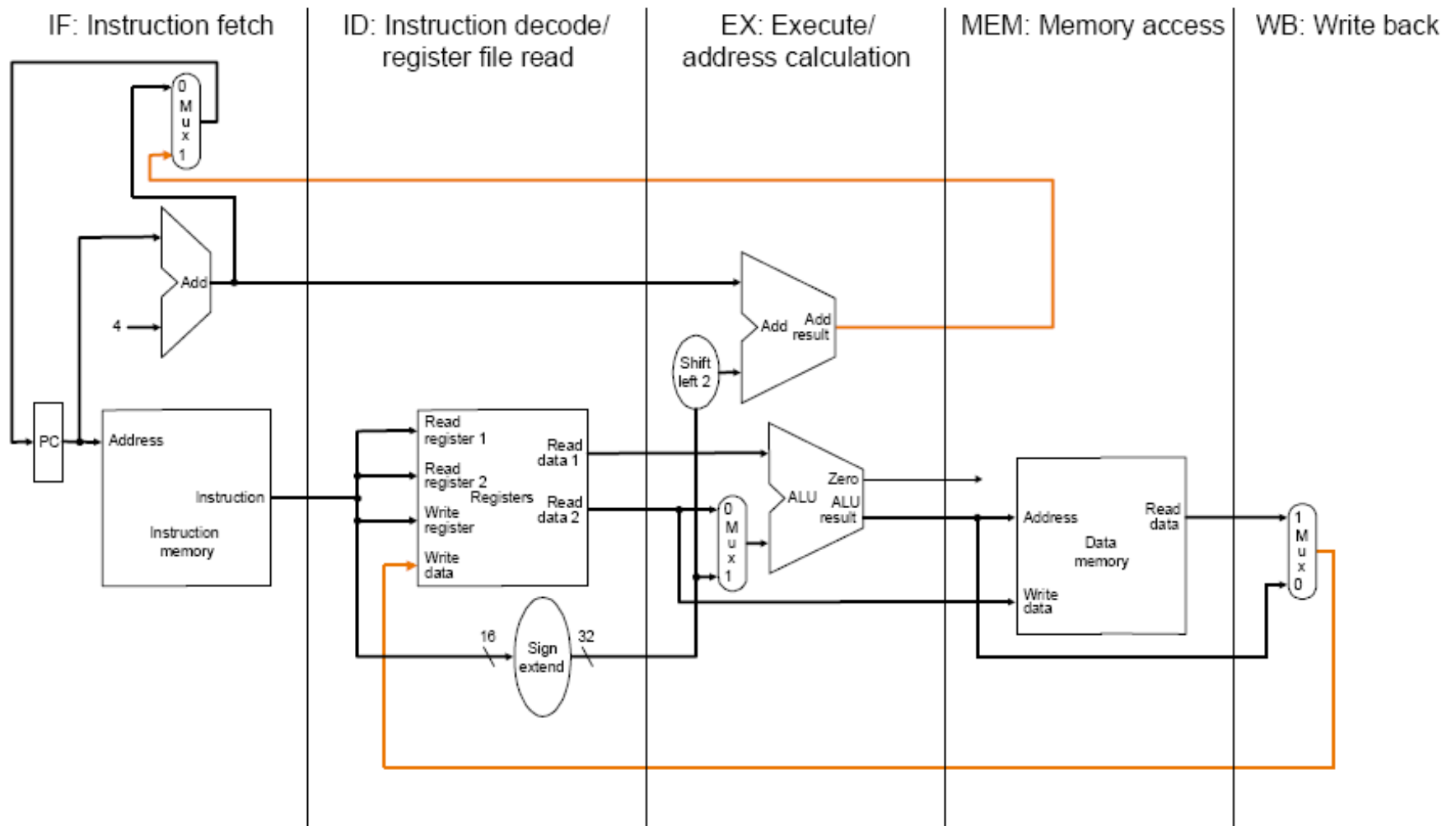
or

`Memory[ALUoutput] = B`

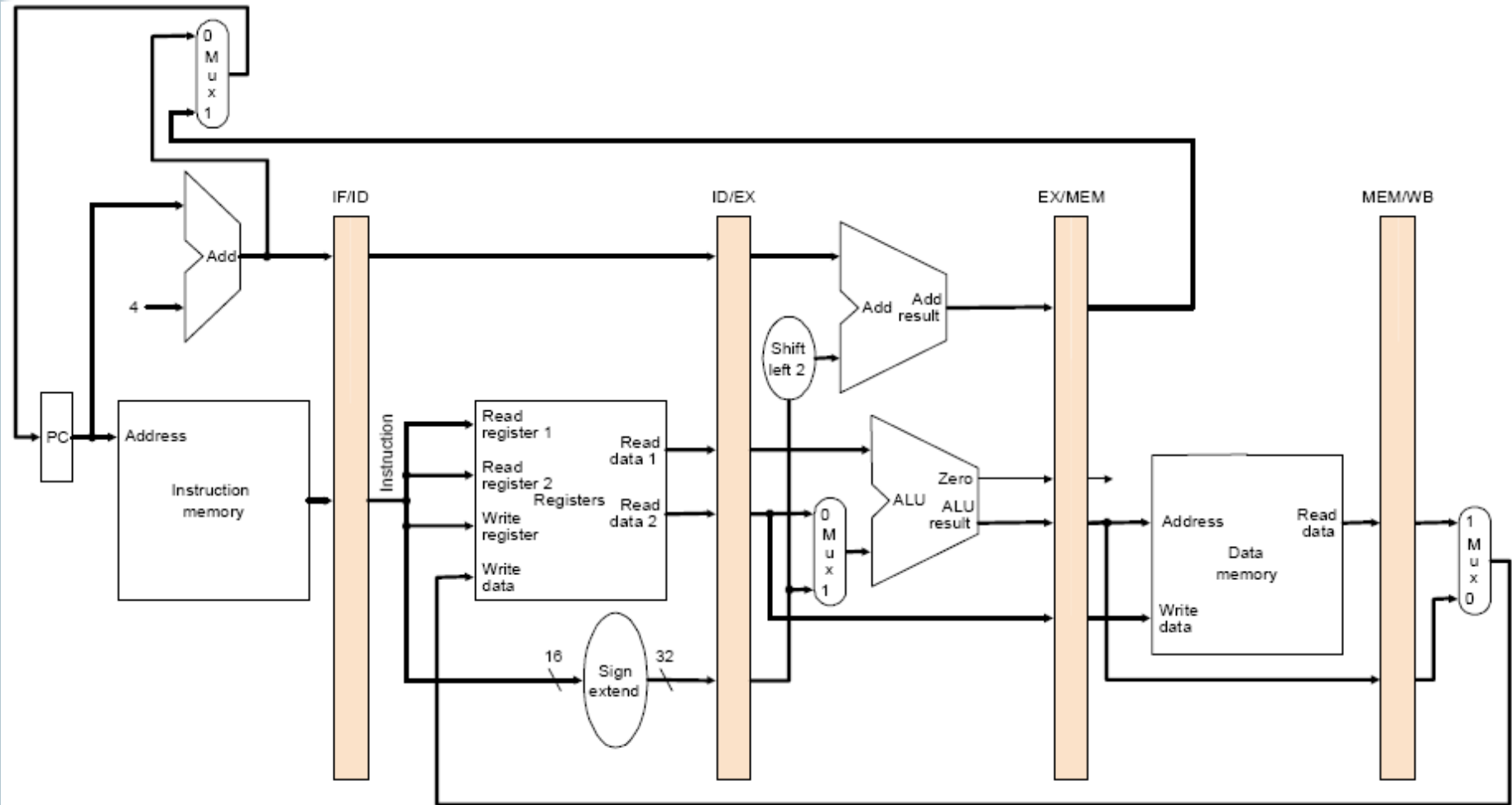
■ Arithmetic-logical instruction (R-type)

`Reg[IR[15-11]] = ALUoutput`

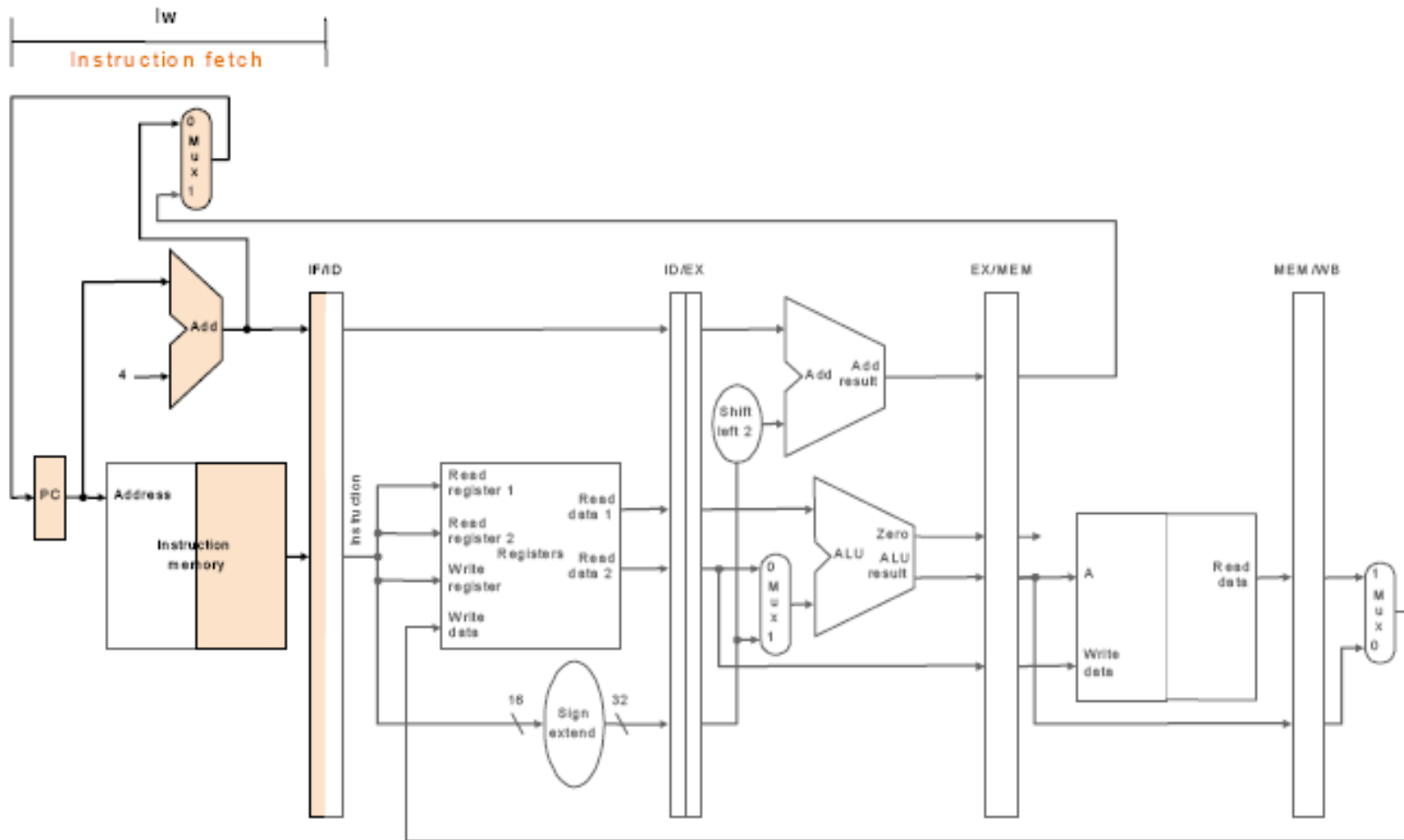
Pipelining first step: Splitting to phases



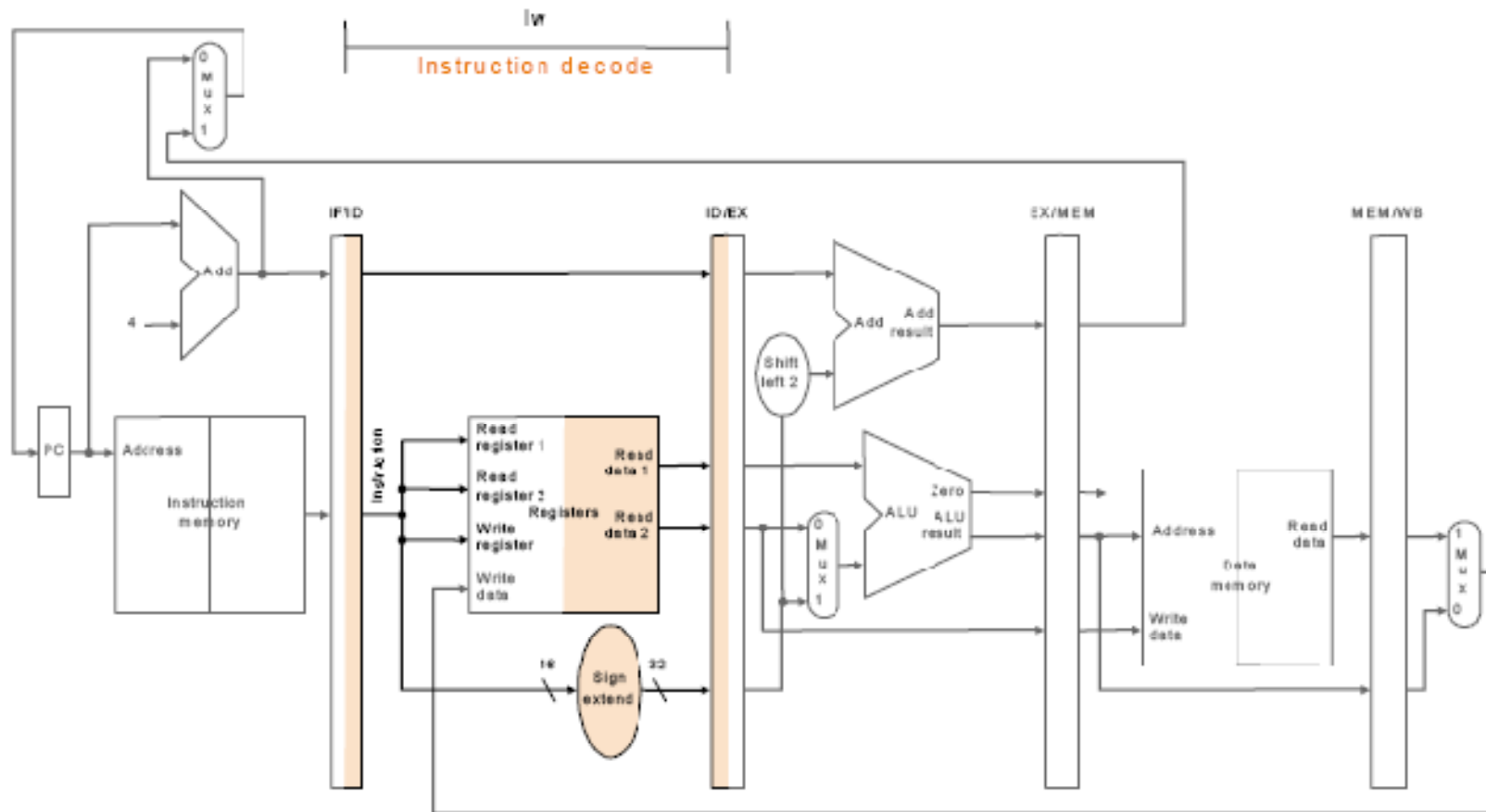
Pipelining second step: Adding registers



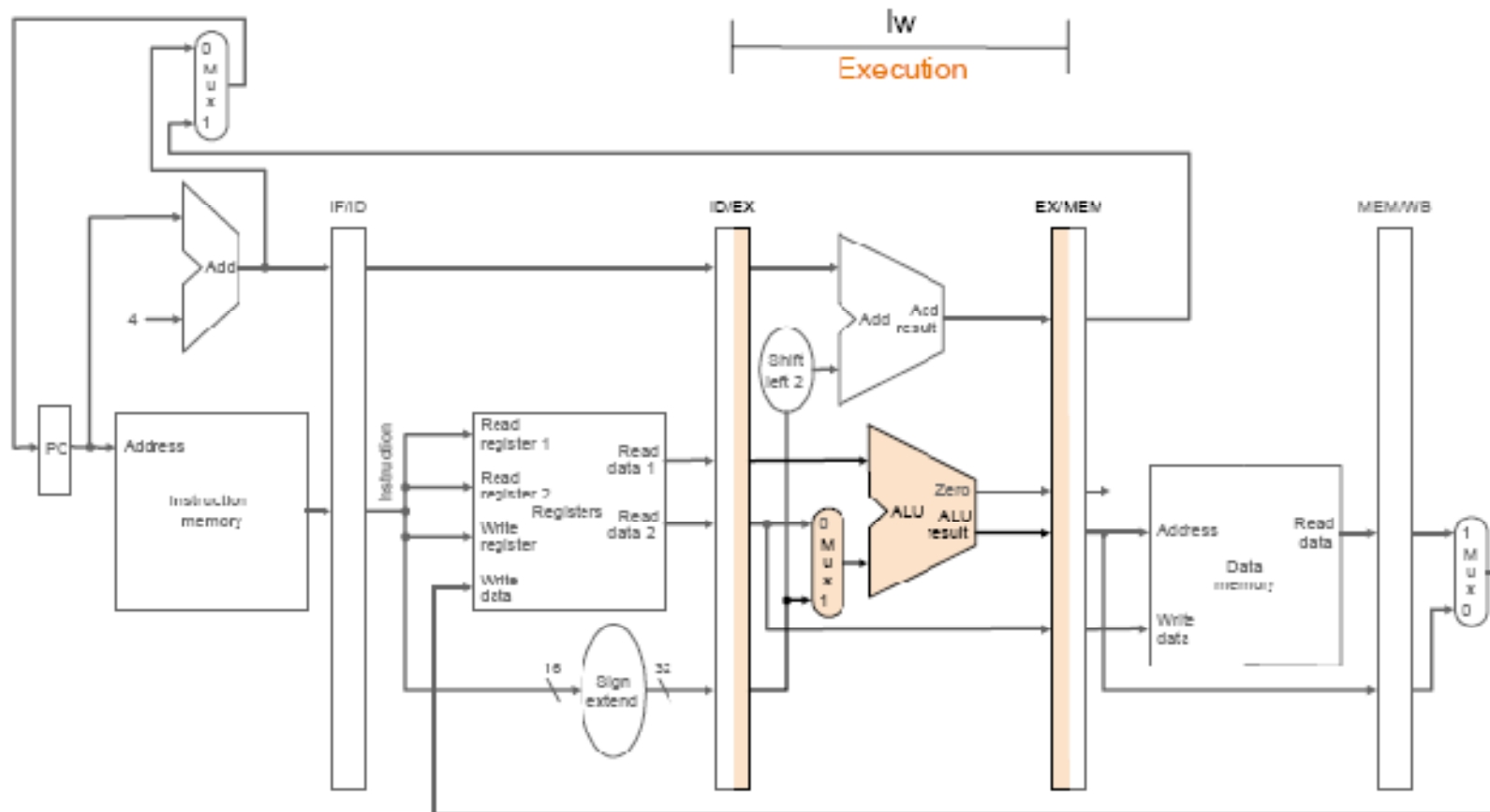
Hardware adaptation for pipelining



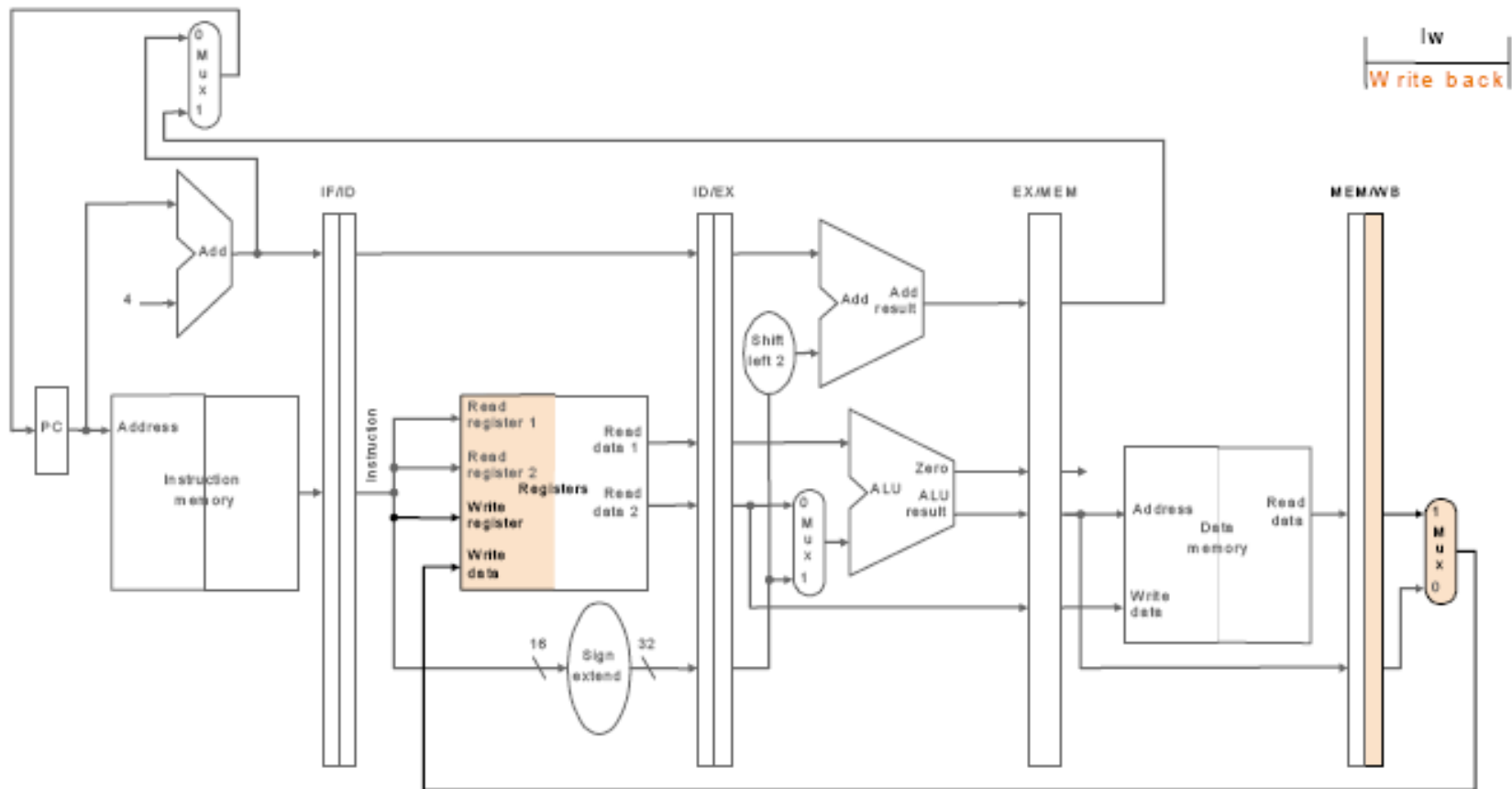
Hardware adaptation for pipelining



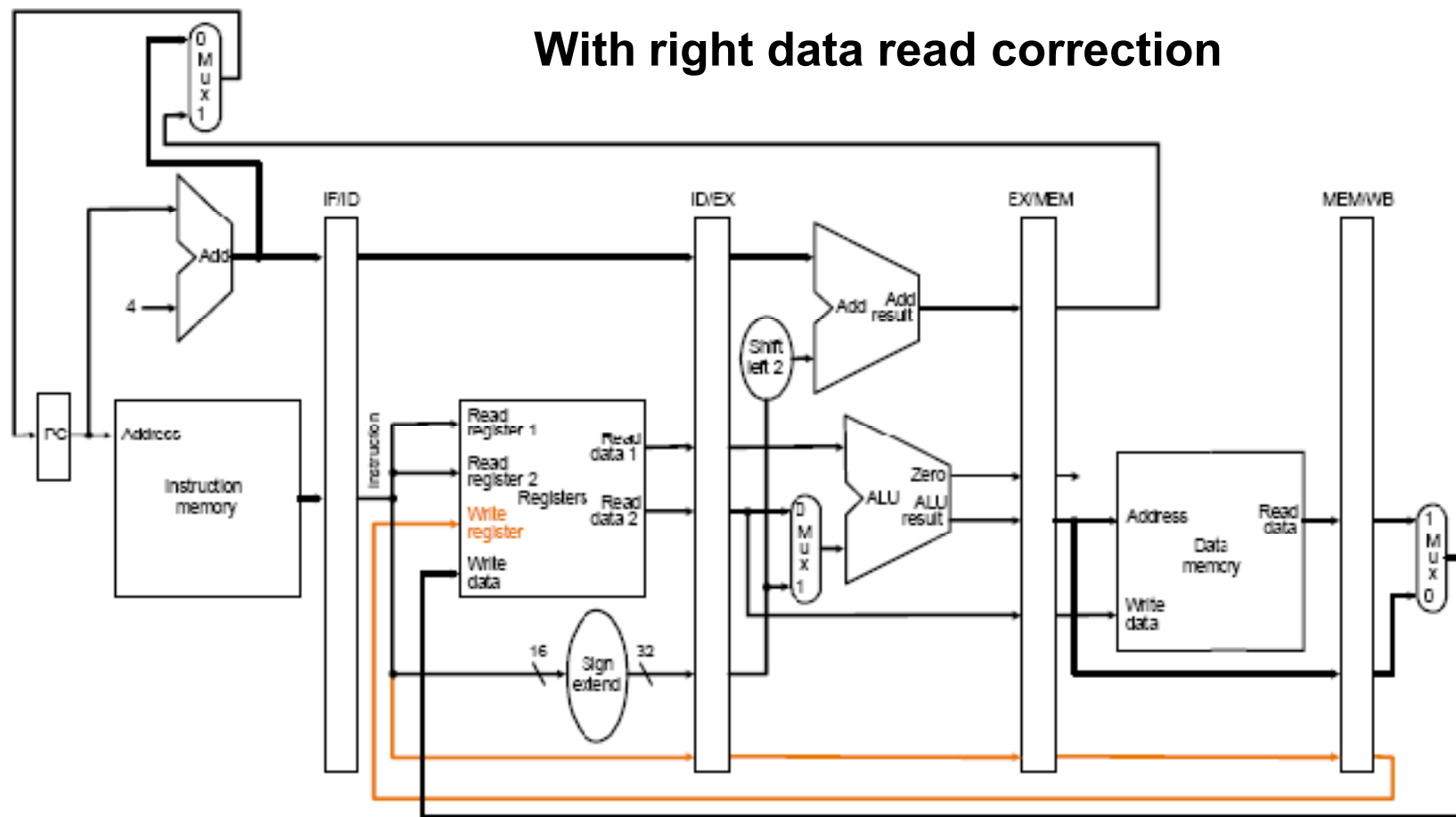
Hardware adaptation for pipelining



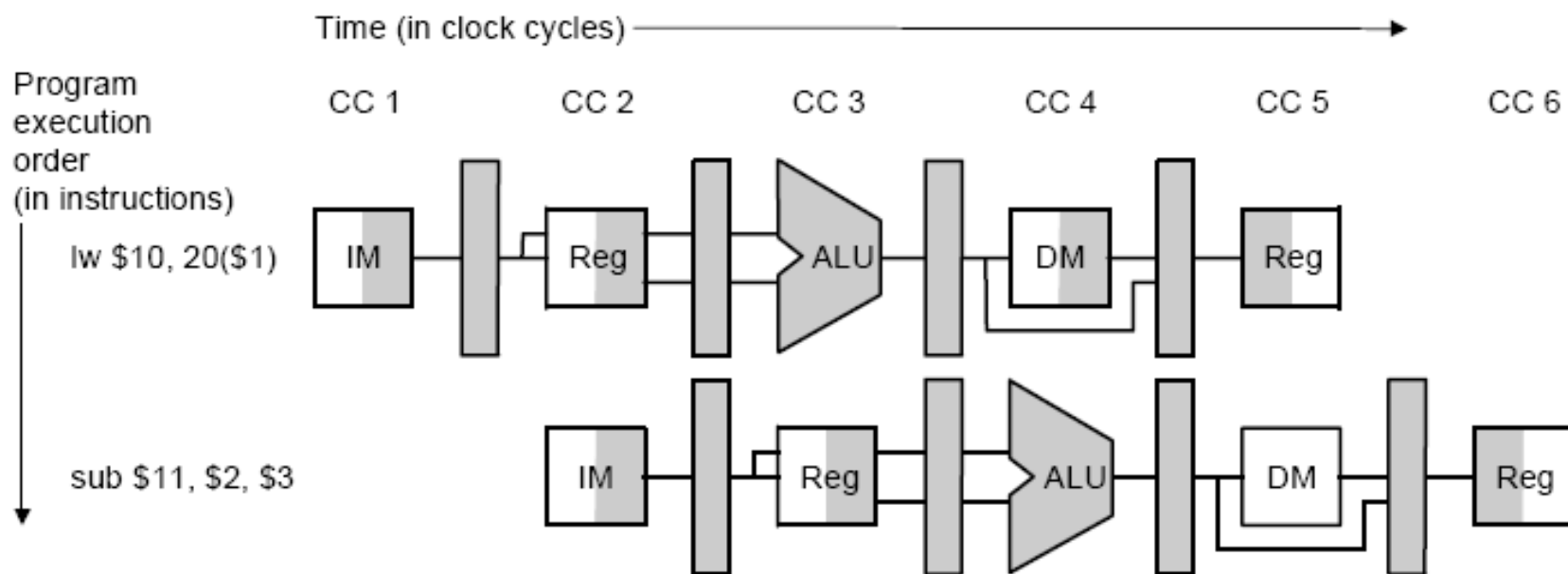
Hardware adaptation for pipelining



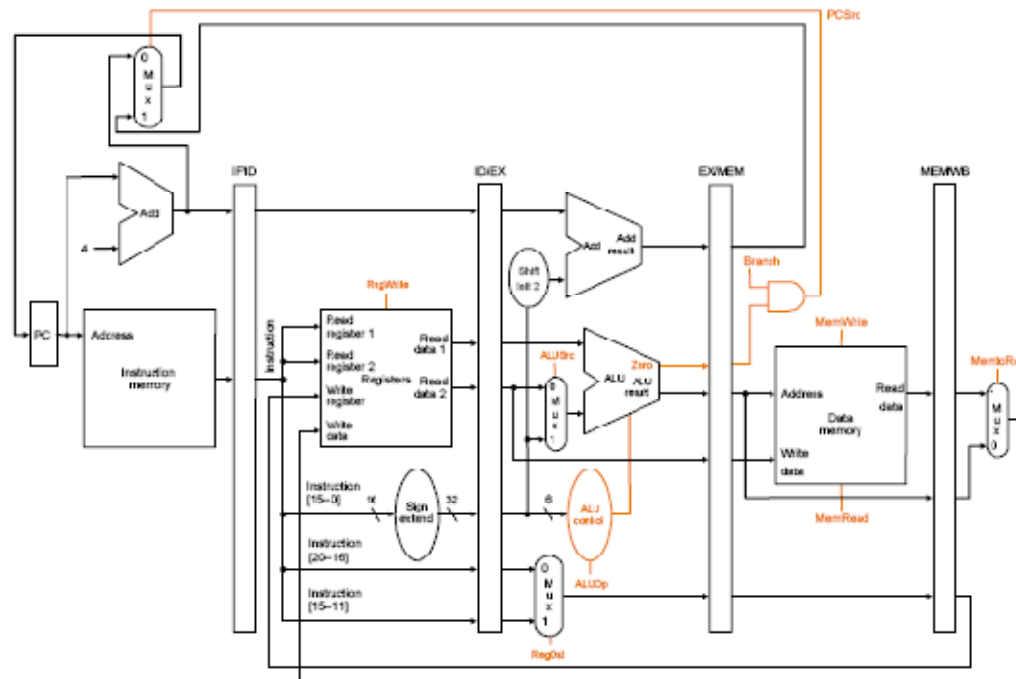
Hardware adaptation for pipelining



Pipeline time sequence



Pipeline Control



Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X



Pipeline Hazards

- There are three types of hazards in a pipeline, they are as follows:
 - Structural Hazards: are created when the data path hardware in the pipeline cannot support all of the overlapped instructions in the pipeline.
 - Data Hazards: When there is an instruction in the pipeline that affects the result of another instruction in the pipeline.
 - Control Hazards: The PC causes these due to the pipelining of branches and other instructions that change the PC.



Dealing With Structural Hazards

- Structural hazards result from the CPU data path not having resources to service all the required overlapping resources.
- Suppose a processor can only read and write from the registers in one clock cycle. This would cause a problem during the ID and WB stages.
- Assume that there are not separate instruction and data caches, and only one memory access can occur during one clock cycle. A hazard would be caused during the IF and MEM cycles.

Dealing With Data Hazards

An example for data hazards:

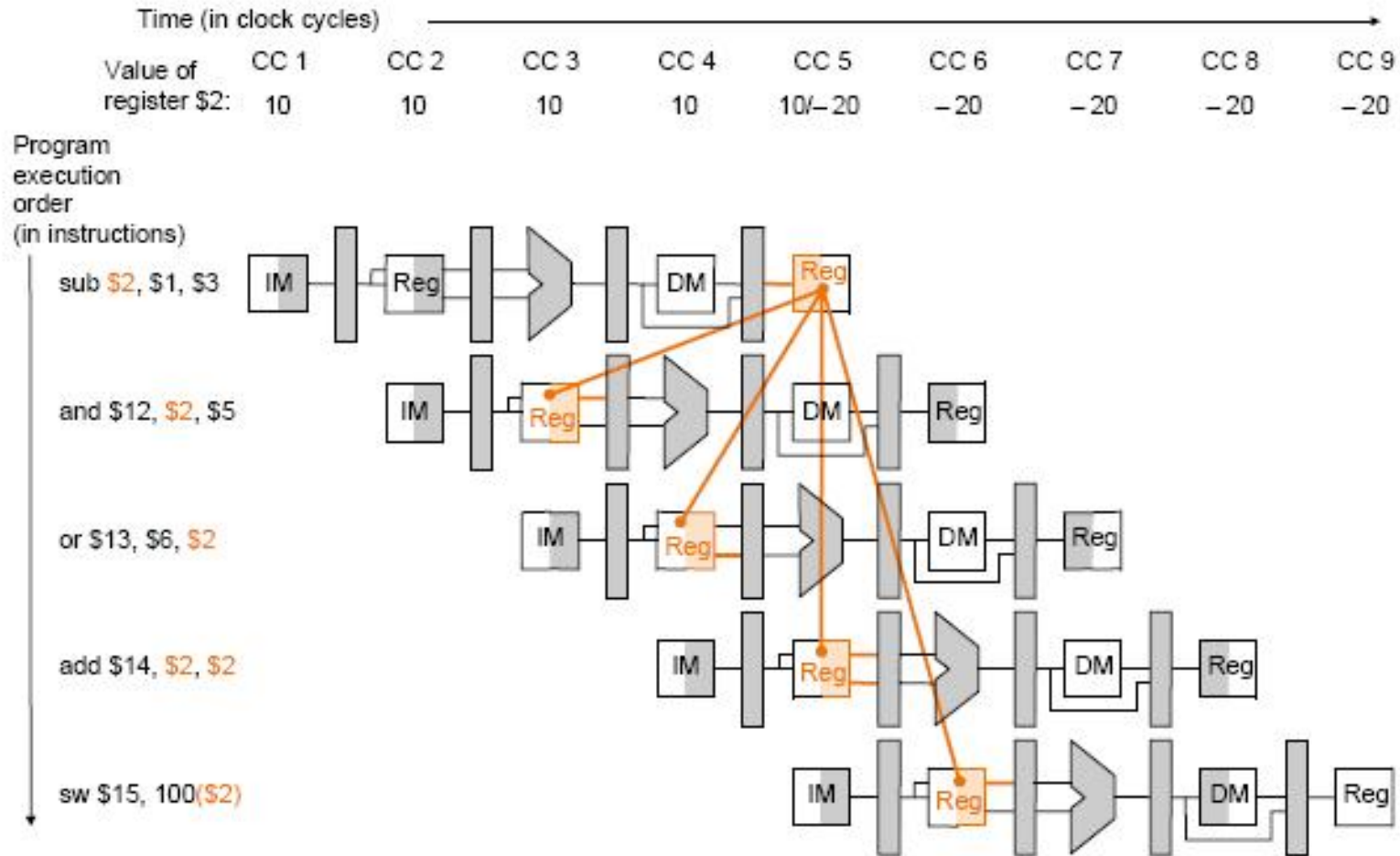
```
sub $2, $1, $3  
and $12, $2, $5  
or $13, $6, $2  
add $14, $2, $2  
sw $15, 100($2)
```

Dealing With Data Hazards

```
sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

An example for data hazards:
Register \$2 is updated only at the
WB phase, i.e., the 5th clock
cycle (actually at the end of the
5th clock cycle). However, we try
to use it at the 3rd clock cycle
when we read \$2 at the decode
phase of the and instruction

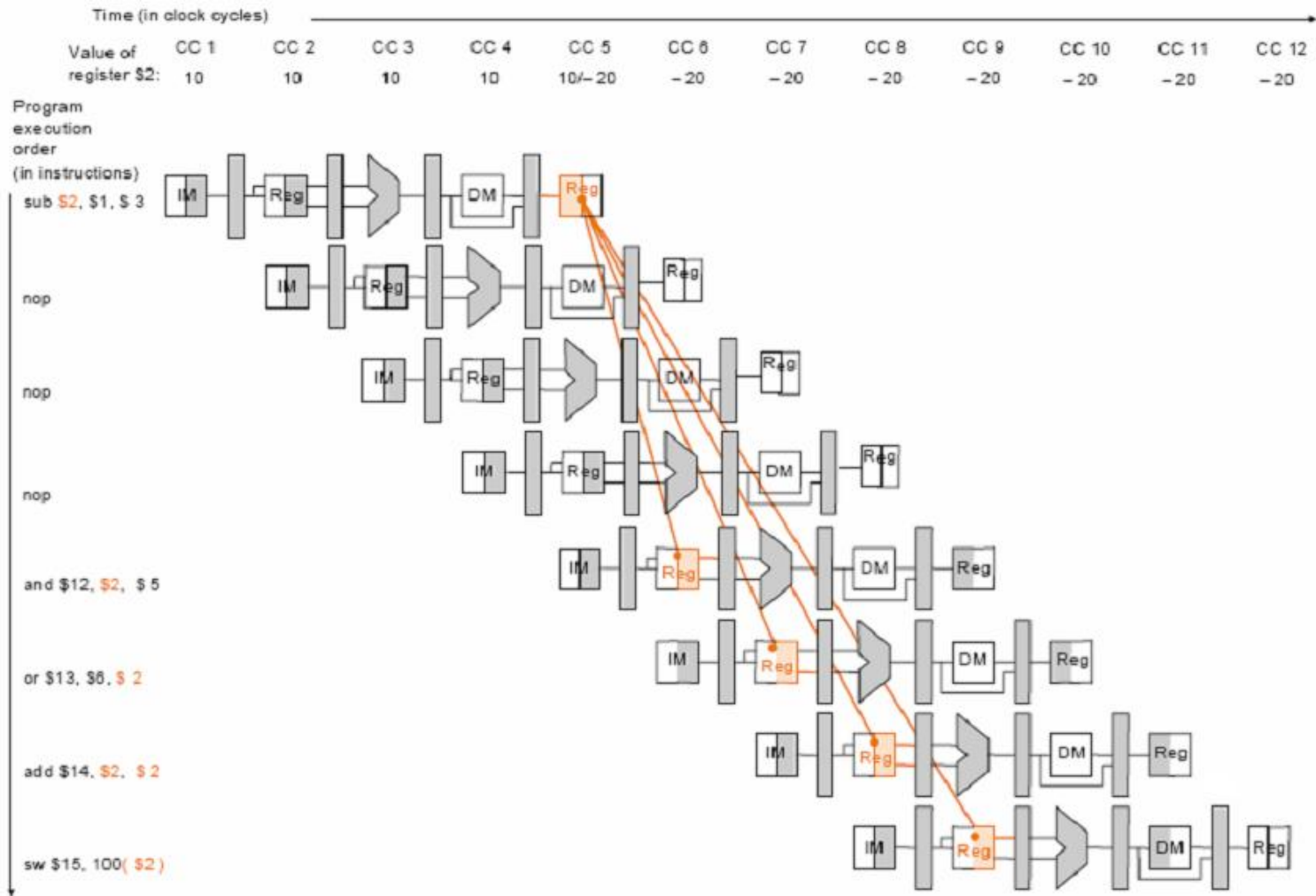
Dealing With Data Hazards



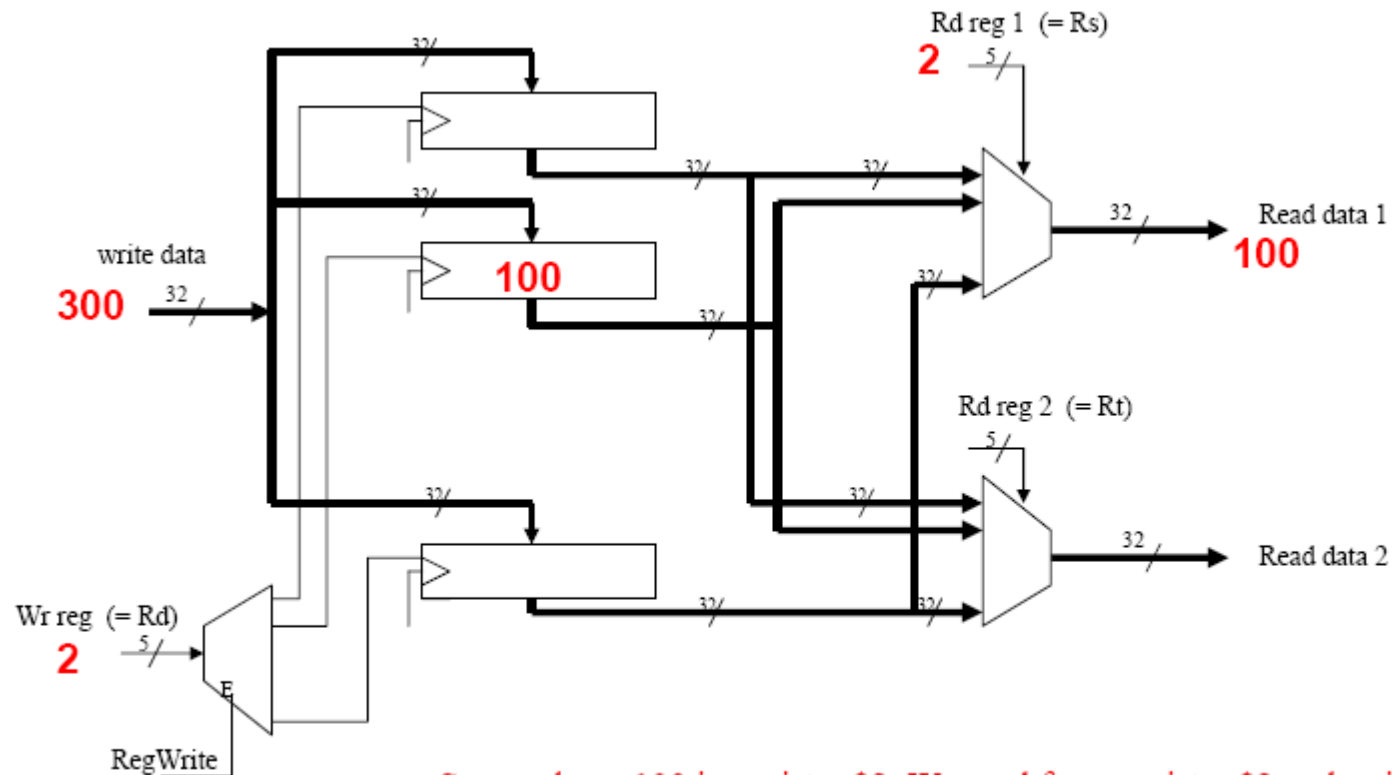
Dealing With Data Hazards- NOP (delay)

```
sub $2, $1, $3
nop
nop
nop
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

Dealing With Data Hazards- NOP (delay)

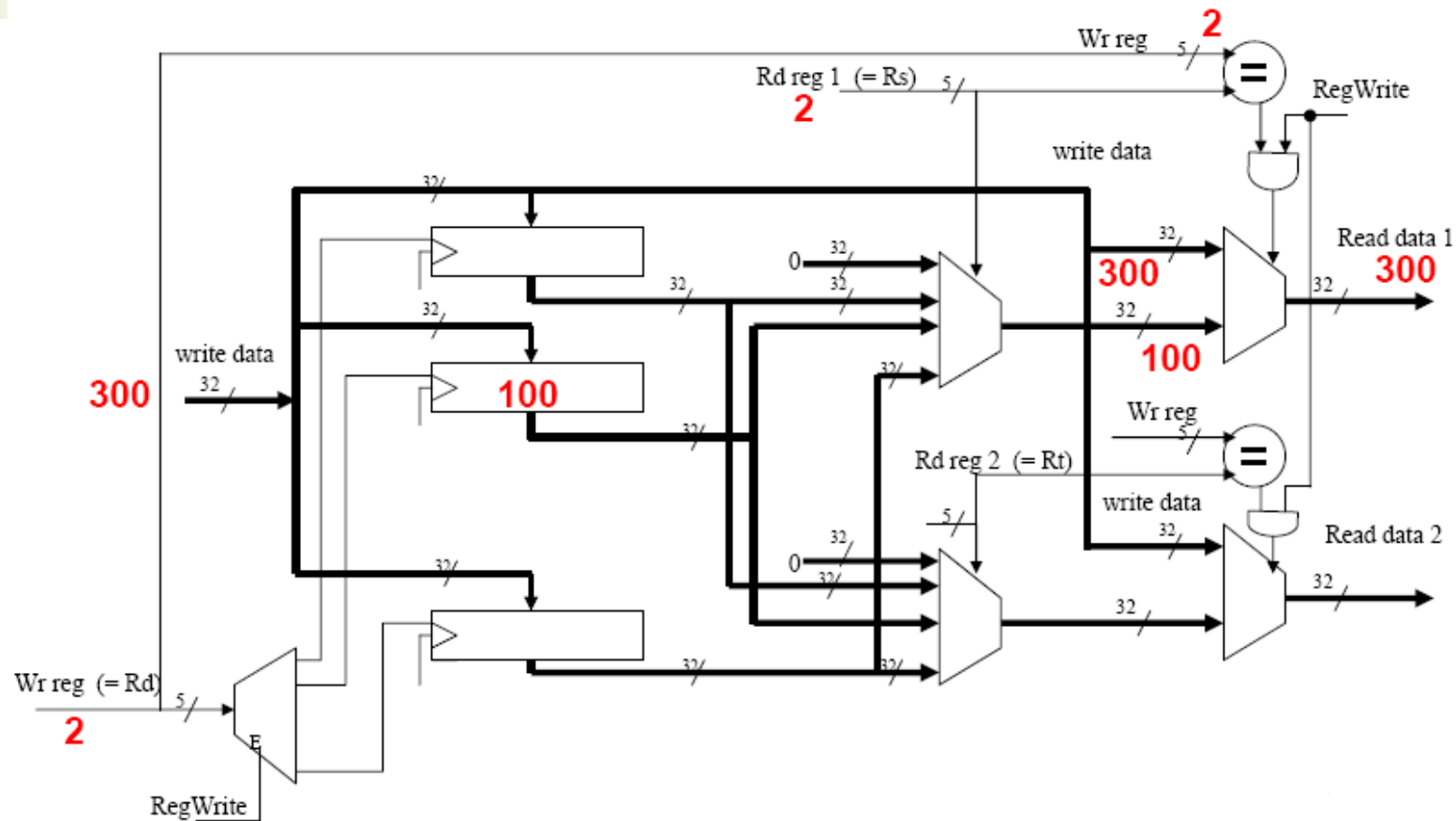


Data Hazards- Hardware Bottleneck



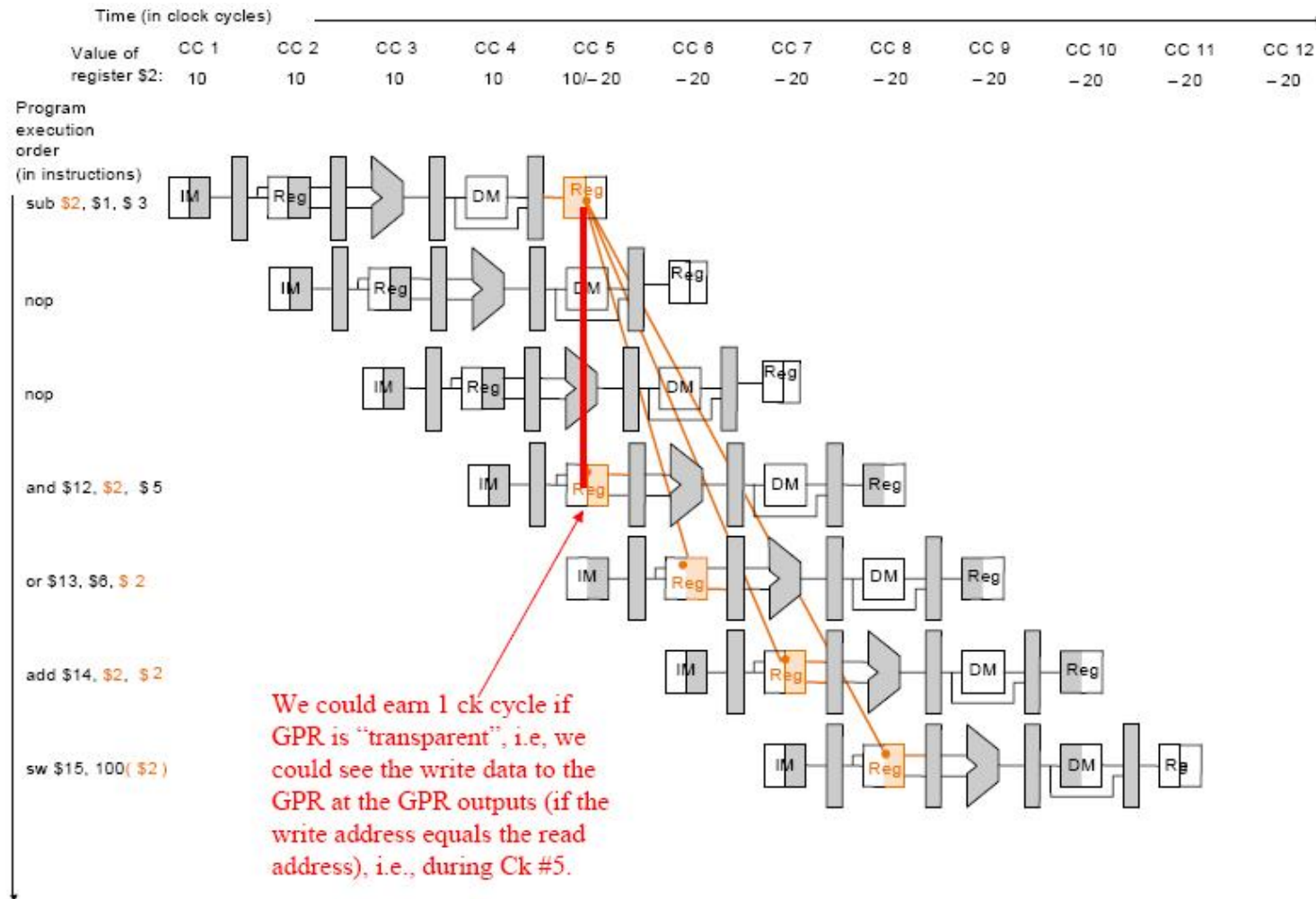
Say we have 100 in register \$2. We read from register \$2 and write 300 into register2. We still get 100 at the output!!! (until the next CK rising edge).

Data Hazards- Hardware Adaptation



When we read and write from/to the same register (e.g., \$2) simultaneously, we bypass the register, which is updated only at the next rising edge of the CK).

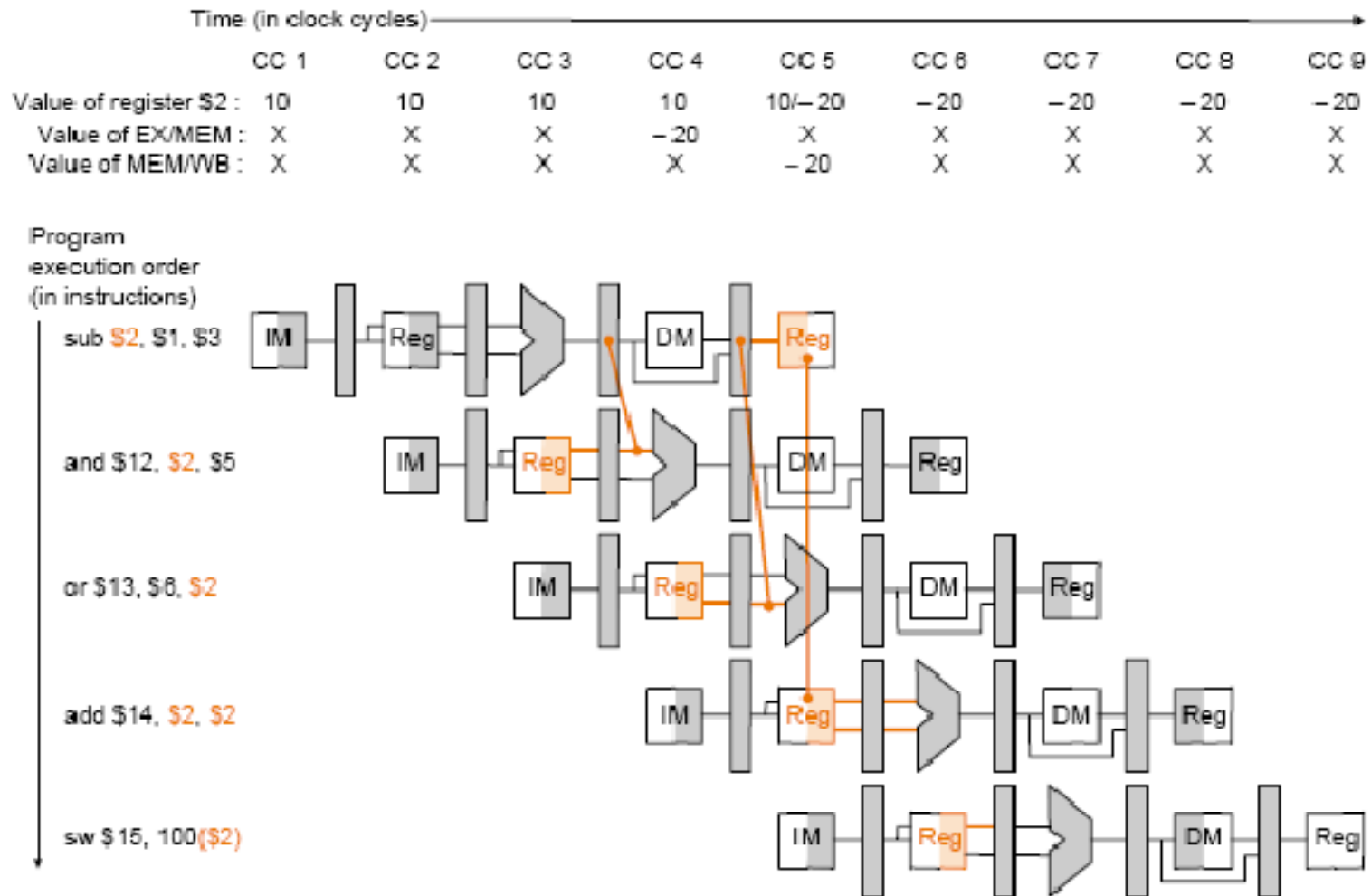
Data Hazards- Hardware Adaptation



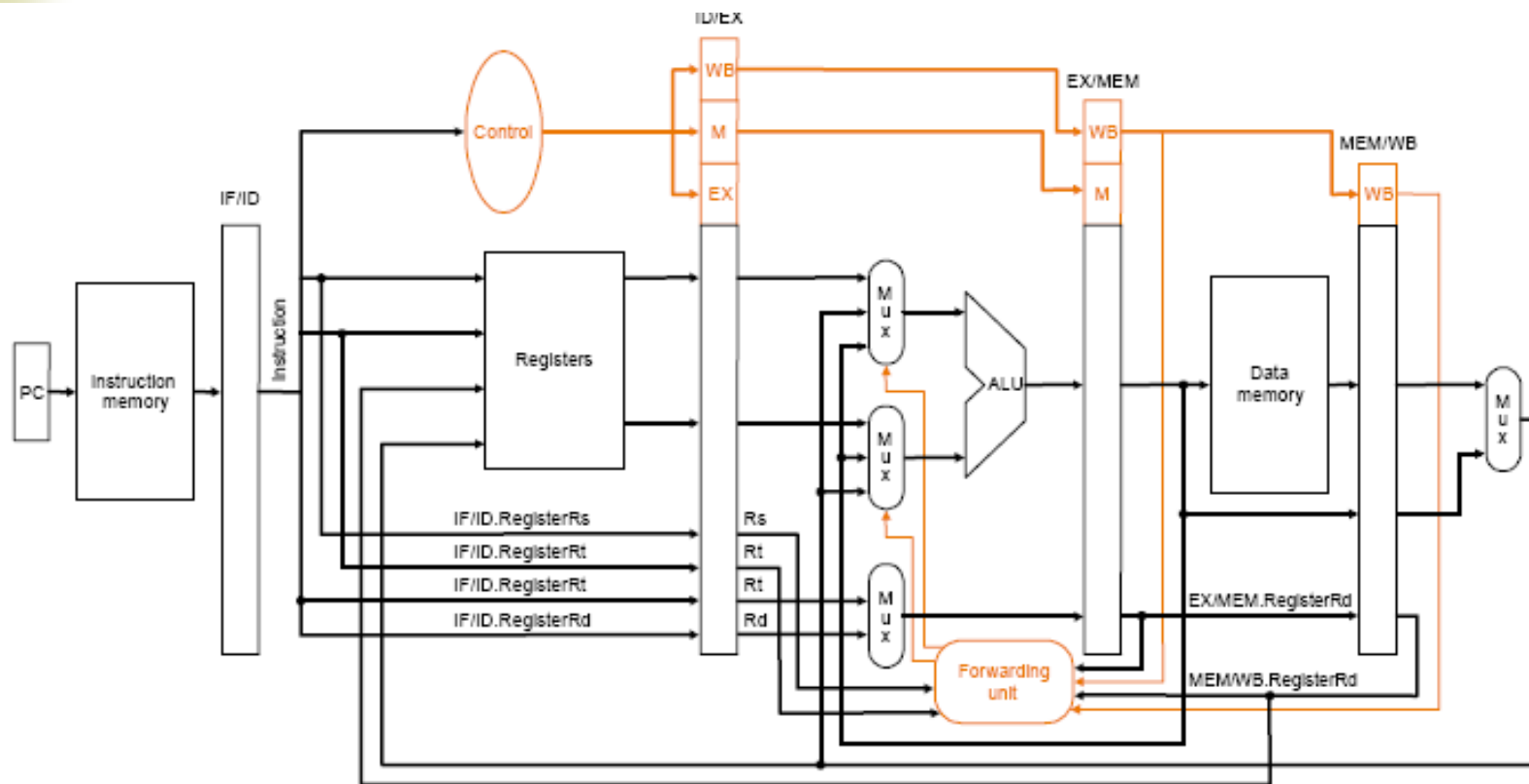
Data Hazards- Hardware Adaptation

```
sub $2, $1, $3  
nop  
nop  
and $12, $2, $5  
or  $13, $6, $2  
add $14, $2, $2  
sw  $15, 100($2)
```

Data Hazards- Hardware Forwarding

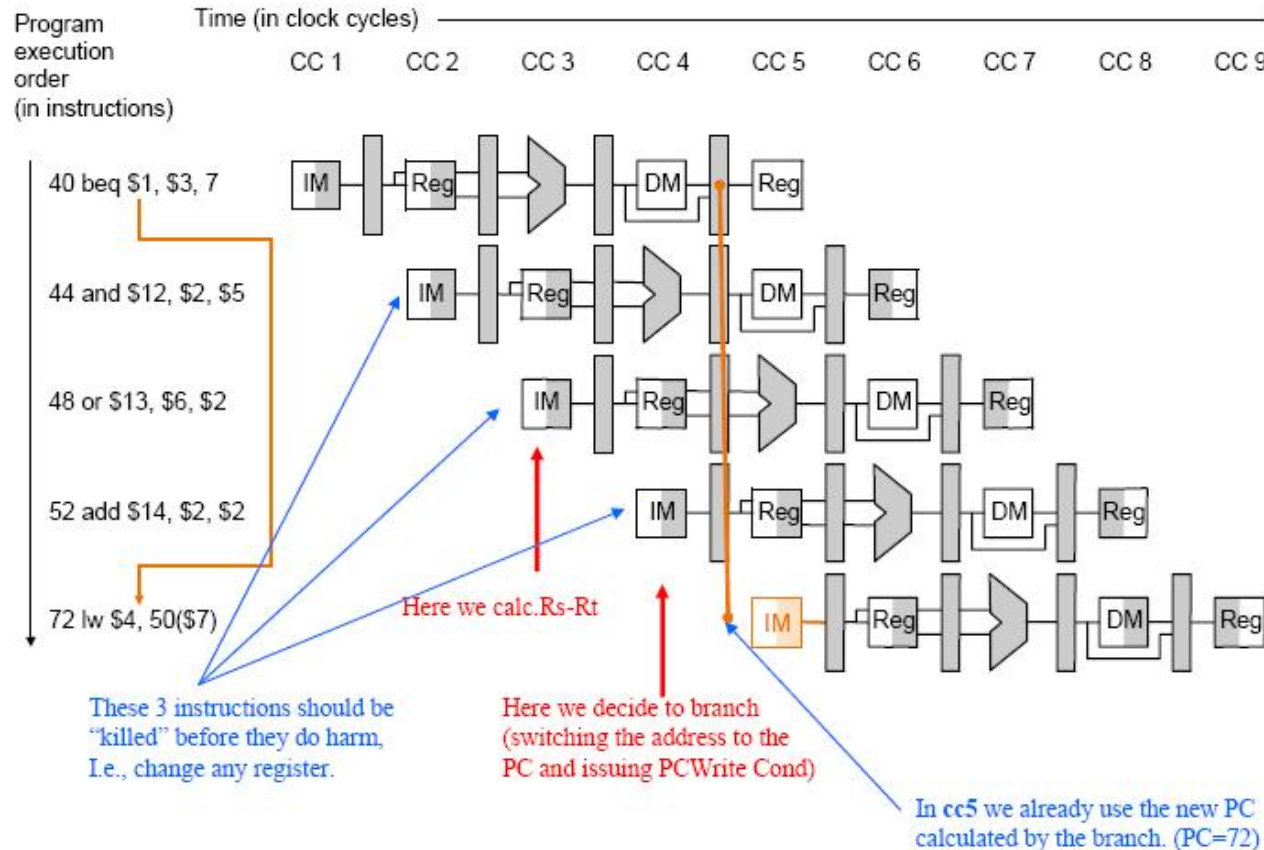


Data Hazards- Hardware Forwarding



If $ID/EX.Rs = EX/MEM.Rd$, i.e., the Rd of the previous instruction equals the Rs of the current instruction (which is in the “decode” phase), then we use the “ALUout” of the previous instruction instead of the output of the GPR.

Dealing With Control Hazards

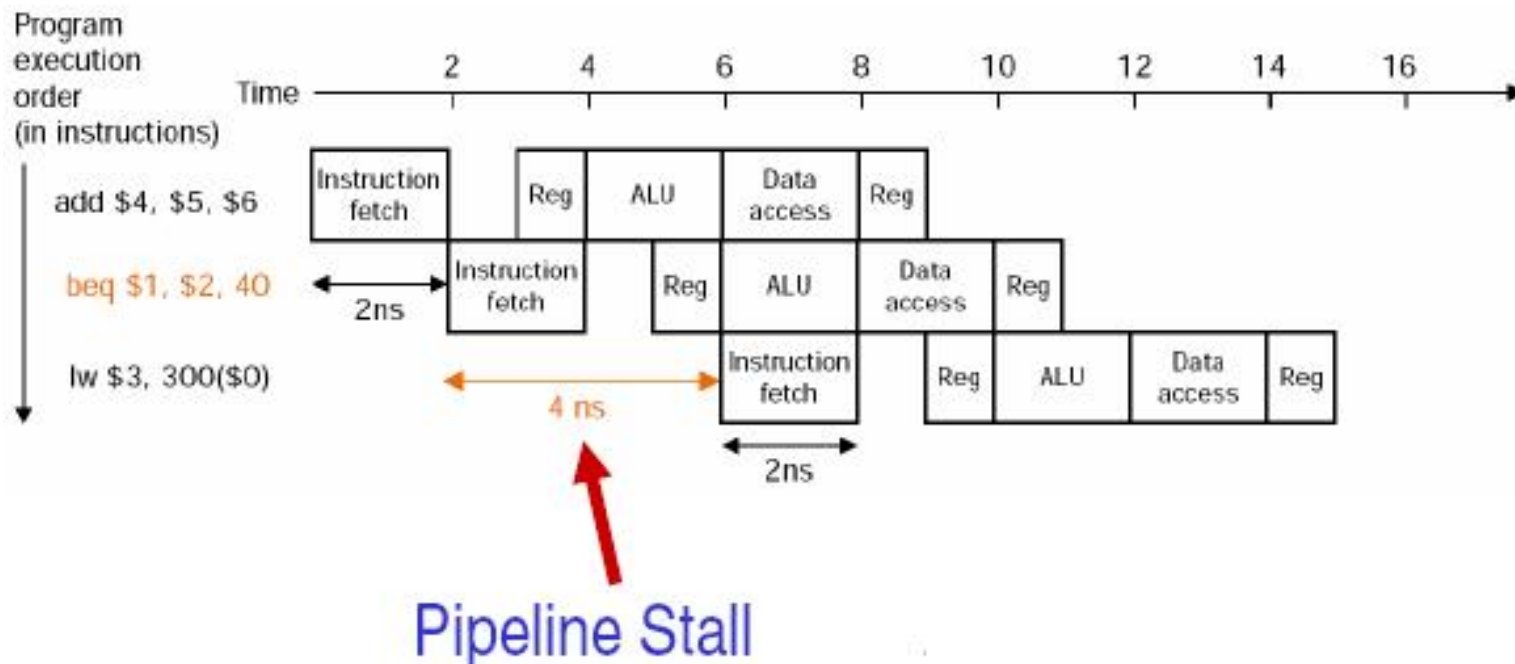


Killing an instruction also called “flushing” the pipeline, is easily done by clearing the IF/ID register of the instruction following the branch (if the branch is successful)

Dealing With Control Hazards

- What is the next instruction?
- Branch instructions take time to compute this.

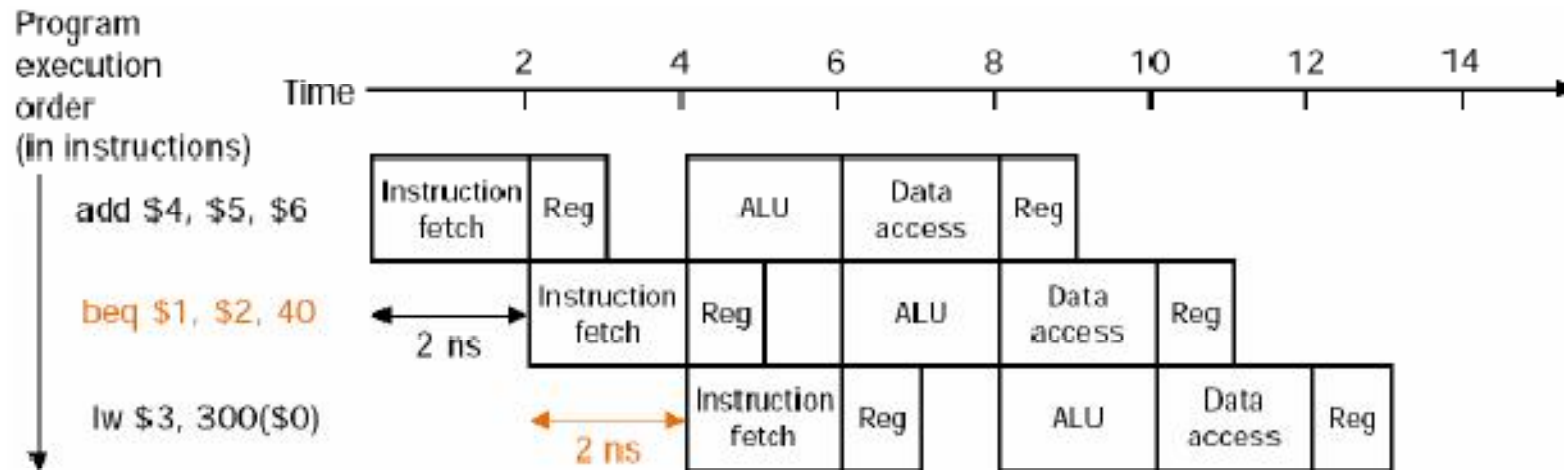
Solution 1: Stall



Dealing With Control Hazards

- What is the next instruction?
- Branch instructions take time to compute this.

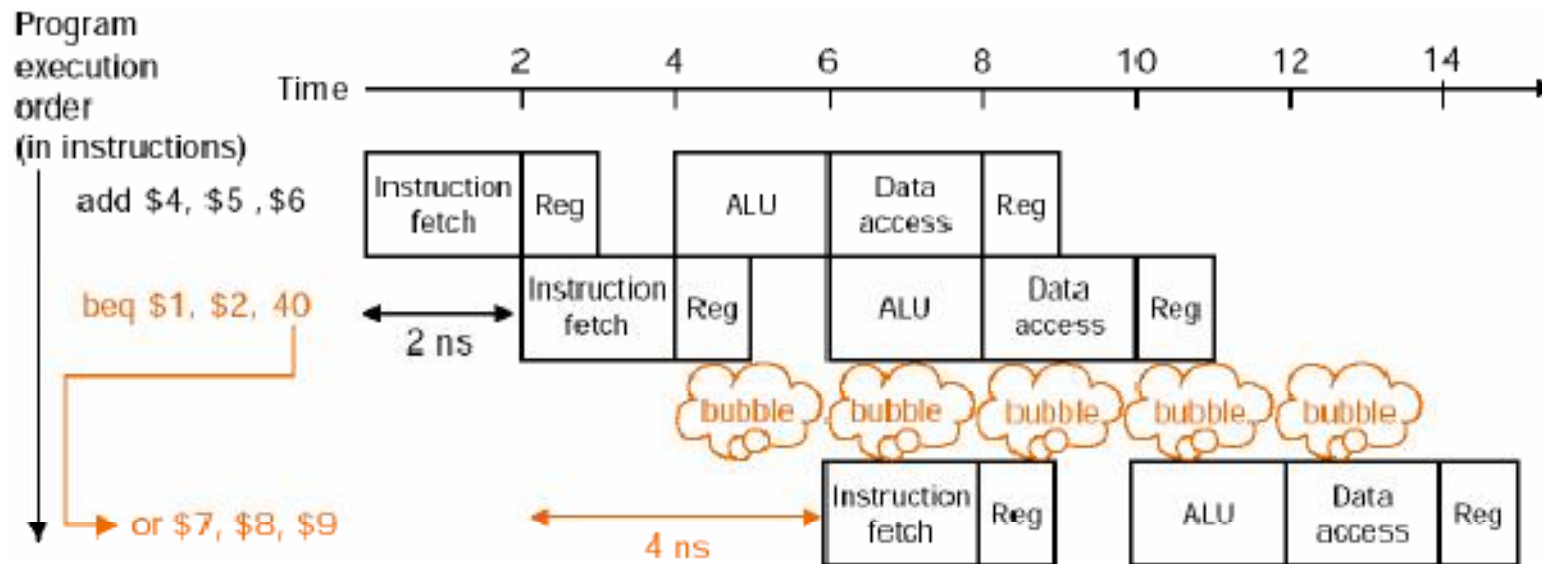
Solution 2: Predict the Branch Target



Dealing With Control Hazards

- What is the next instruction?
- Branch instructions take time to compute this.

Solution 2: (Mis)Predict the Branch Target



Pipeline Speedup

- Some performance expressions involving a realistic pipeline in terms of CPI. It is assumed that the clock period is the same for pipelined and unpipelined implementations.

$$\text{Speedup} = \text{CPI Unpipelined} / \text{CPI pipelined}$$

- We can look at pipeline performance in terms of a faster clock cycle time as well:

$$\text{Speedup} = \frac{\text{Clock cycle time unpipelined}}{\text{Clock cycle time pipelined}}$$

$$\text{Clock cycle pipelined} = \frac{\text{Clock cycle time unpipelined}}{\text{Pipeline Depth}}$$

$$\text{Speedup} = \frac{1}{1 + \text{Pipeline stalls per Ins}} \times \text{Pipeline Depth}$$



Summary

Summary

- Pipelining is a fundamental concept in computers/nature
 - Multiple instructions in flight
 - Limited by length of longest stage, Latency vs. Throughput
- Hazards gum up the works

Real Stuff

- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism to a point
- Pentium 4 has 22 pipe stages!