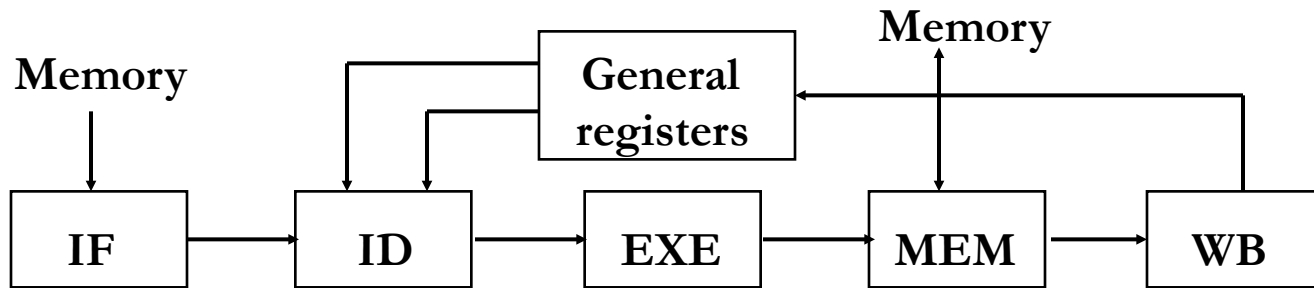


# Improving Performance: Pipelining

## Phases of instruction execution



- IF** Instruction Fetch (includes PC increment)
- ID** Instruction Decode + fetching values from general purpose registers
- EXE** Execute arithmetic/logic operations or address computation
- MEM** Memory access or branch completion
- WB** Write Back results to general purpose registers (a.k.a. Commit)

# Generalized Phases of Instruction Execution

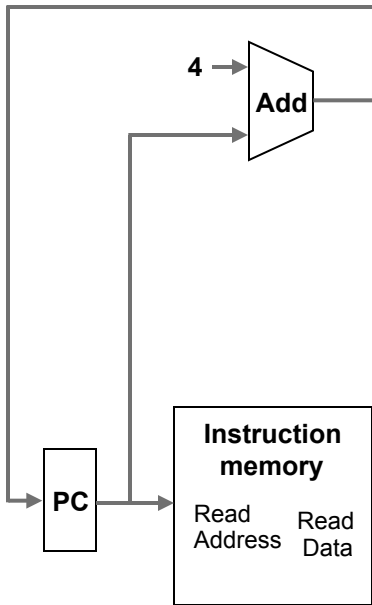
---

- Instruction Fetch
  - InstructionRegister = MemRead (INST\_MEM, PC)
- Decoding
  - Generate datapath control signals
  - Determine register operands
- Operand Assembly
  - Trivial for some ISAs, not for others
  - E.g. select between literal or register operand; operand pre-scaling
  - Sometimes considered to be part of the Decode phase
- Function Evaluation or Address Calculation (Execution)
  - Add, subtract, shift, logical, etc.
  - Address calculation is simply addition
- Memory Access (if required)
  - *Load*: ReadData = MemRead(DATA\_MEM, MemAddress, Size)
  - *Store*: MemWrite (DATA\_MEM, MemAddress, WriteData, Size)
- Completion
  - Update processor state modified by this instruction
  - Interrupts or exceptions may prevent state update from taking place

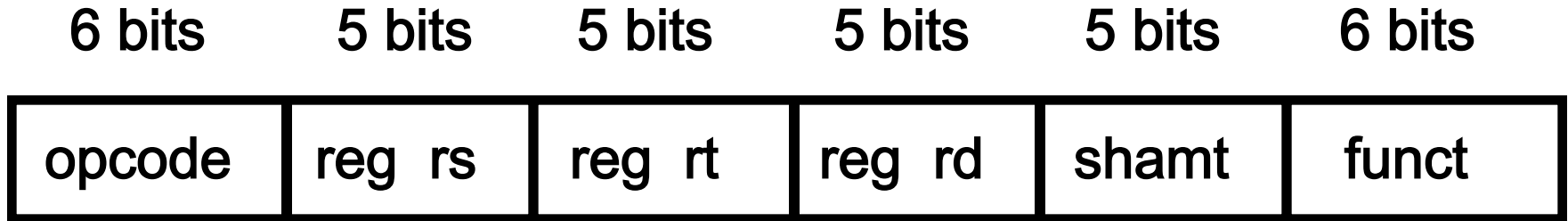
**Note: INST\_MEM and DATA\_MEM may be same or separate physical memories**

# Instruction fetch

- Read from memory (typically, Instruction Cache) at address given by PC
- Increment PC, i.e.  $PC = PC + \text{sizeof}(\text{instruction})$



# MIPS R-type instruction format

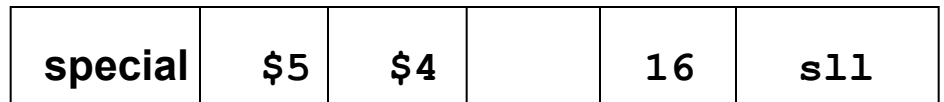


Destination register for R-type format

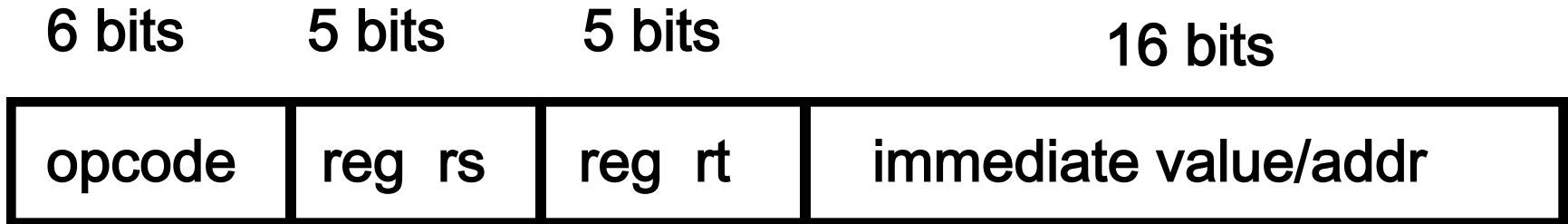
add      \$1, \$2, \$3



sll      \$4, \$5, 16



# MIPS I-type instruction format



Destination register for Load

`lw $1, offset($2)`

<code>lw</code>	<code>\$2</code>	<code>\$1</code>	<code>address offset</code>
-----------------	------------------	------------------	-----------------------------

`beq $4, $5, .Label1`

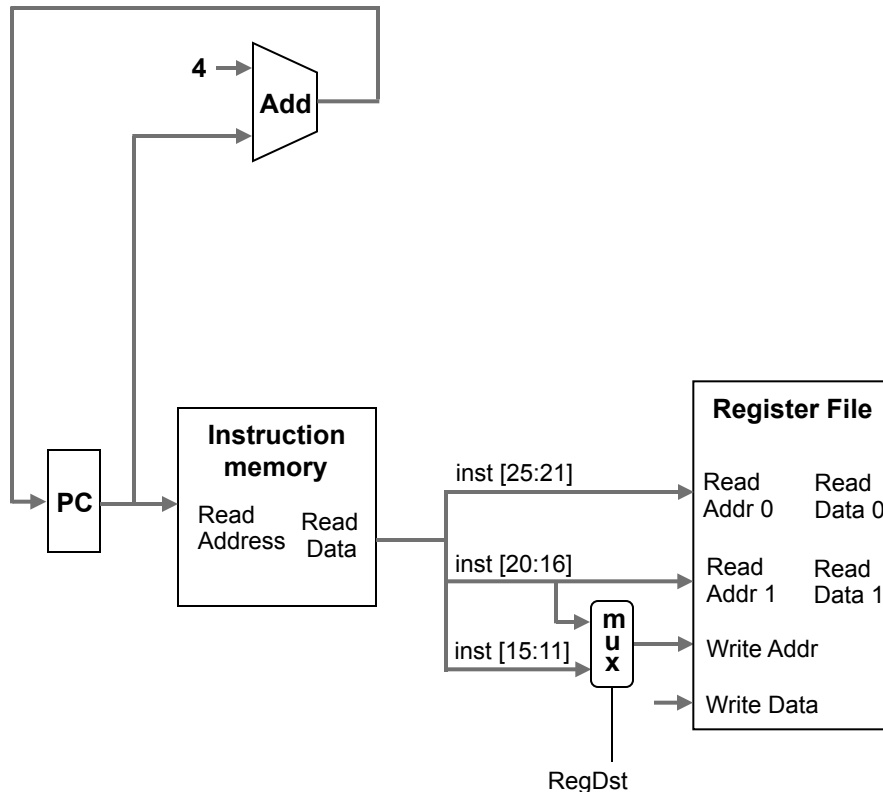
<code>beq</code>	<code>\$4</code>	<code>\$5</code>	<code>(PC - .Label1) &gt;&gt; 2</code>
------------------	------------------	------------------	--

`addi $1, $2, -10`

<code>addi</code>	<code>\$2</code>	<code>\$1</code>	<code>0xfff6</code>
-------------------	------------------	------------------	---------------------

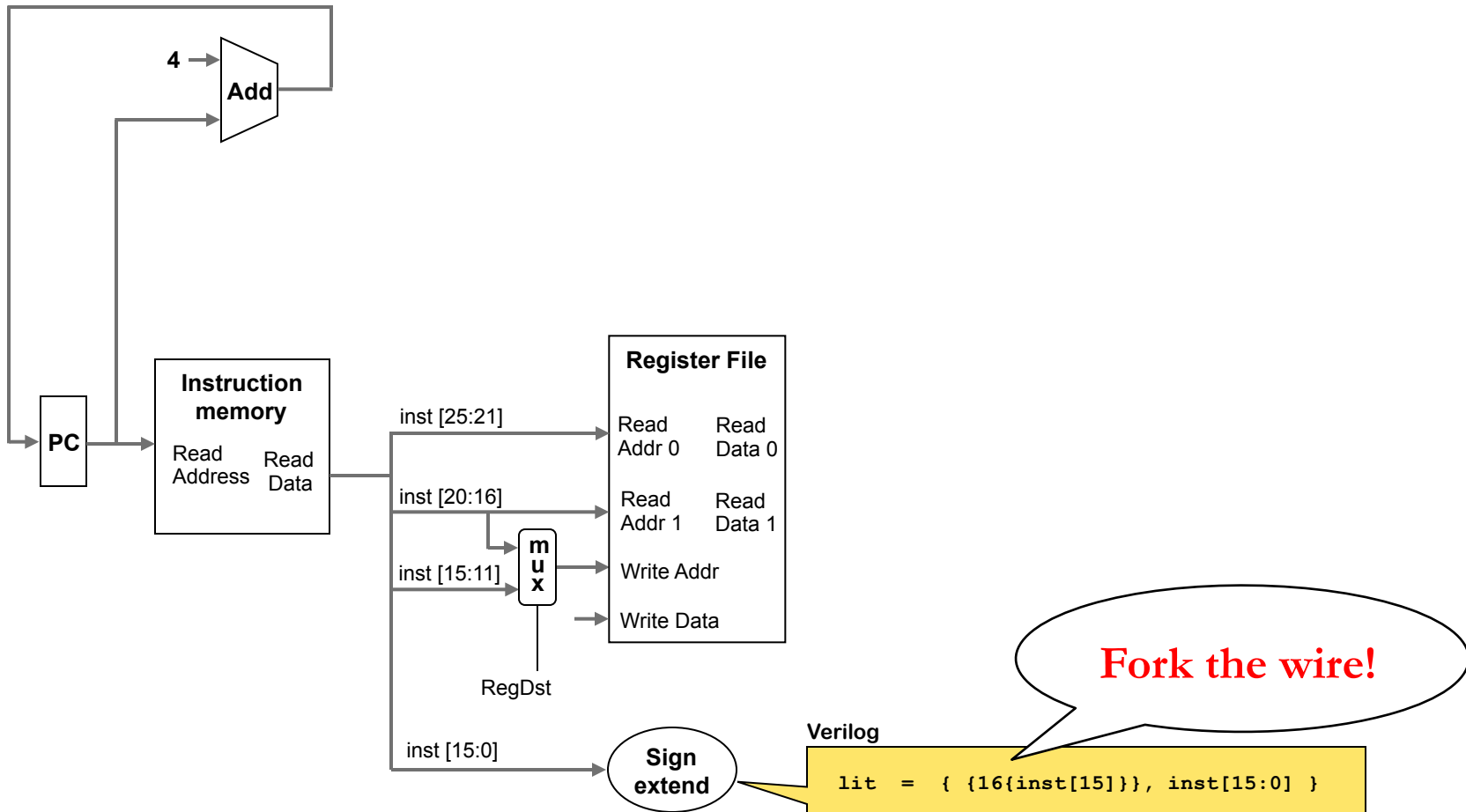
# Reading Registers

- Use source register fields to address the register file and read two registers
- Select the destination register address, according to the format



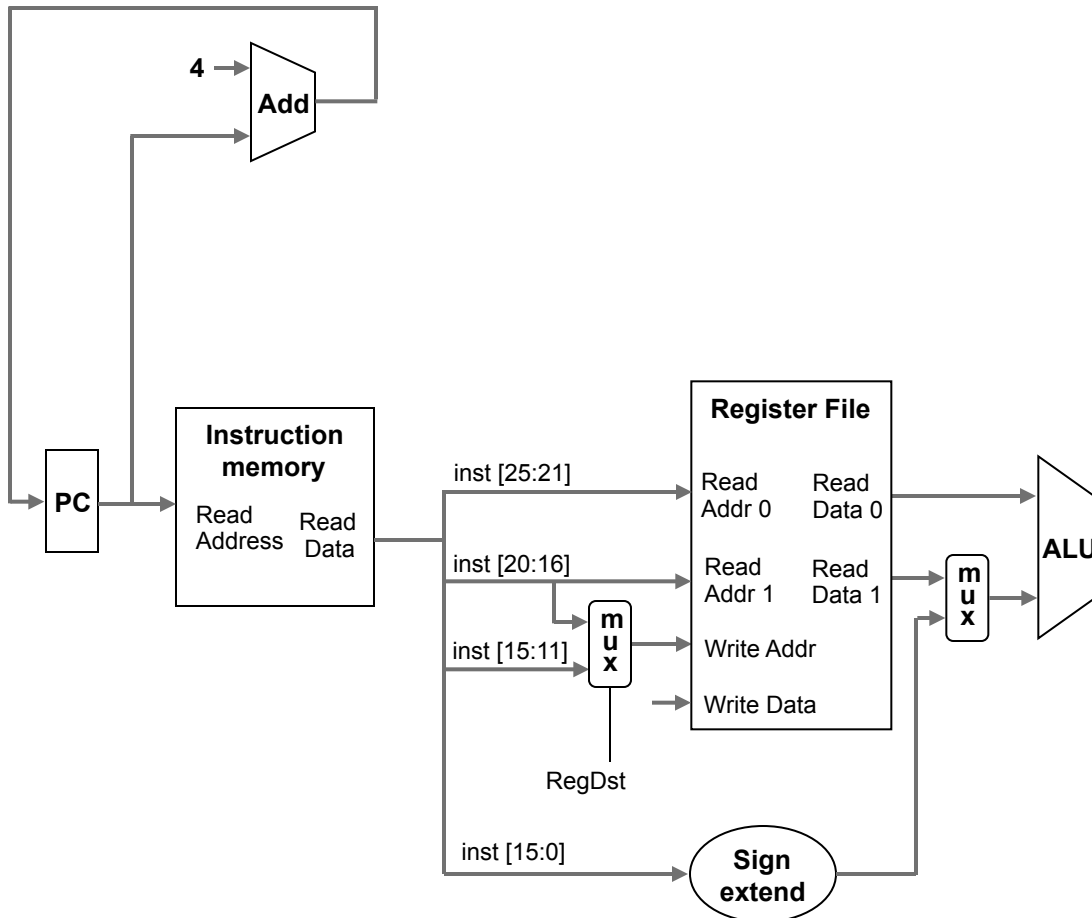
# Extracting the literal operand

- Sign-extend the 16-bit literal field, for those instructions that have a literal



# Performing the Arithmetic

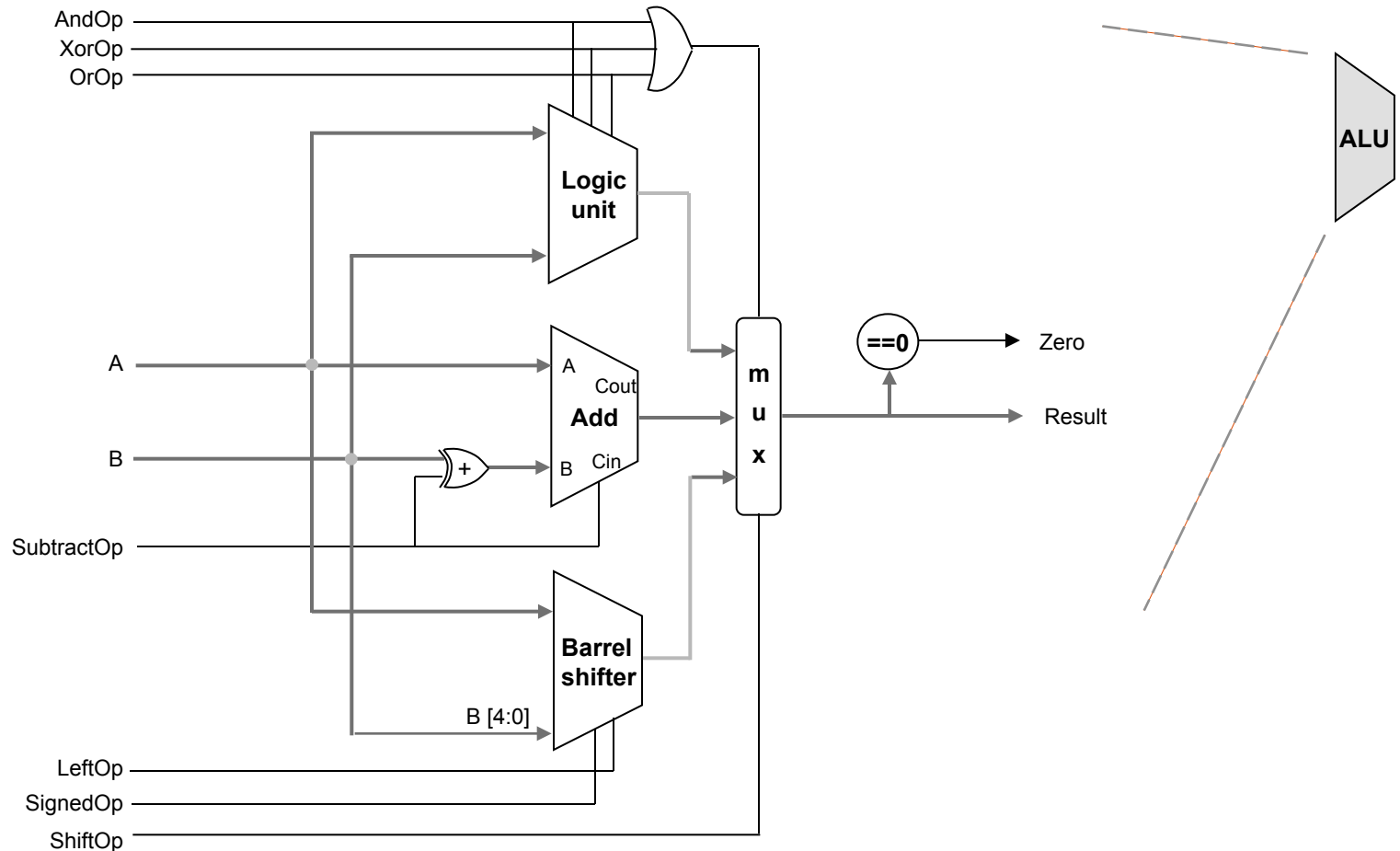
- Perform arithmetic or logical operation on Read Data 0 and either Read Data 1 or the sign-extended literal





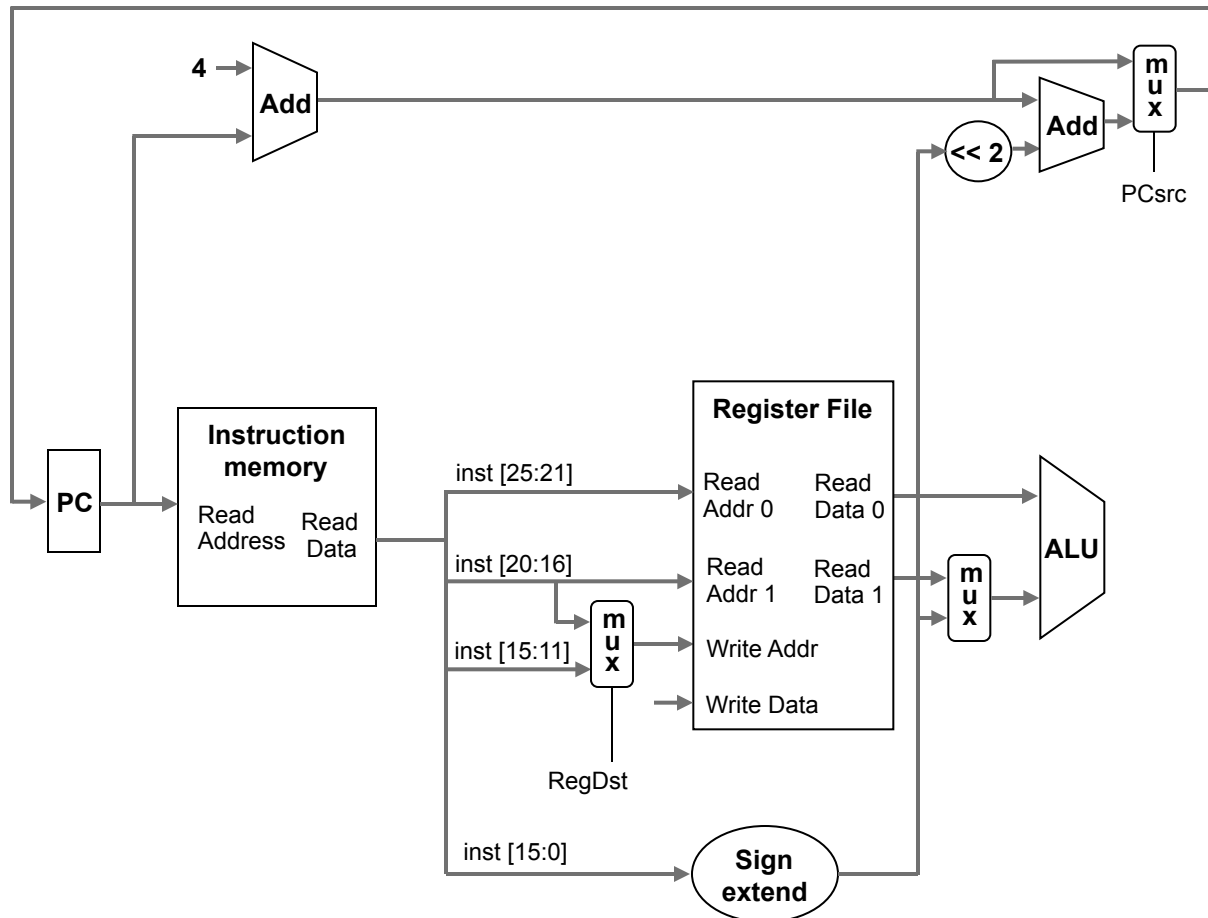
# Inside the ALU

- Adder, Logic Unit, and Barrel Shifter are separate combinational logic blocks



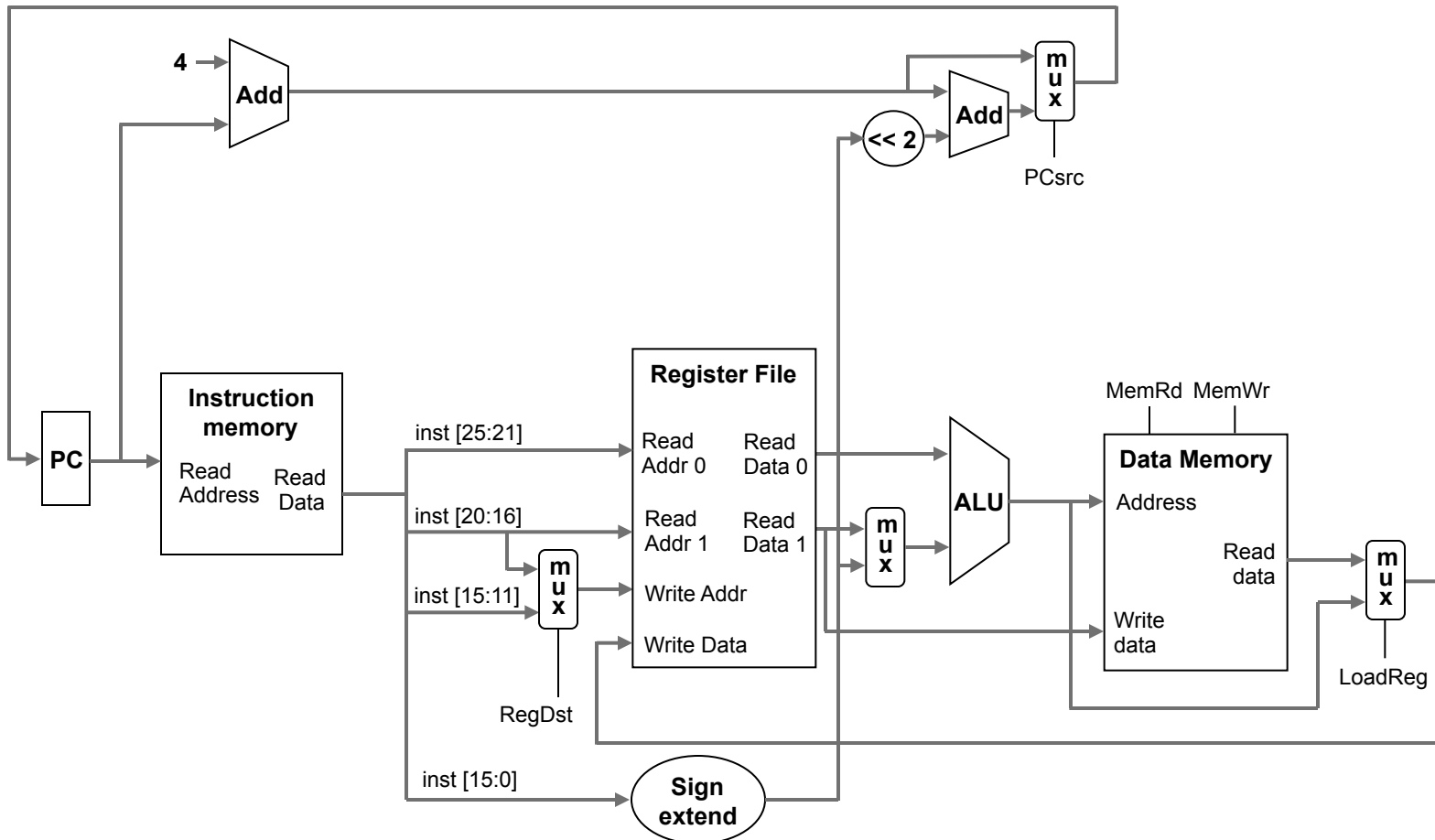
# Computing Branch Displacements

- Compute sum of PC and scaled, sign-extended literal displacement
- The main ALU is used for evaluating the branch condition (BEQ, BNE)



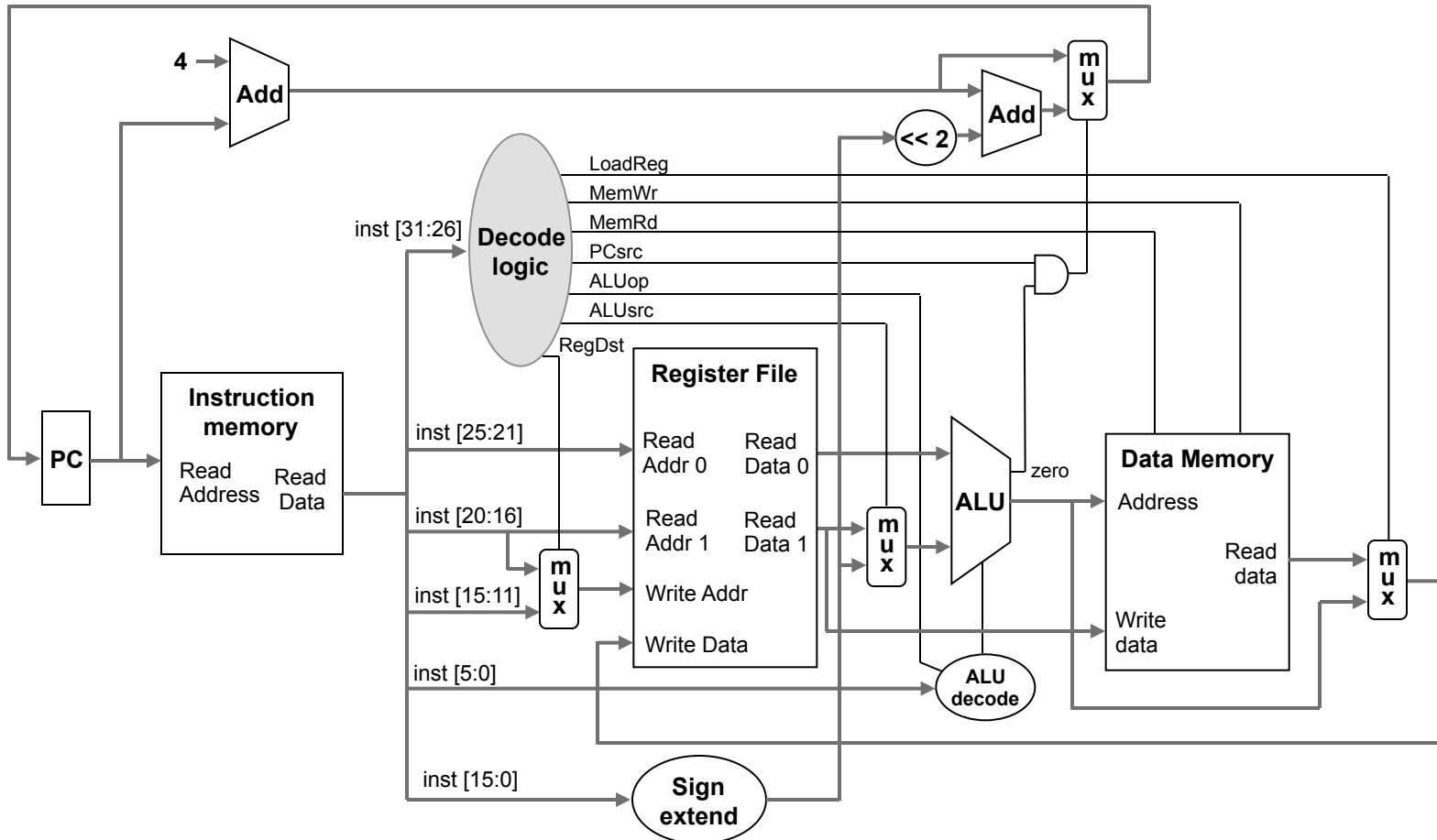
# Accessing Memory – Loads & Stores

- Load and Store instructions use the ALU result as the effective address
- Store instructions use Read Data 1 as the store data

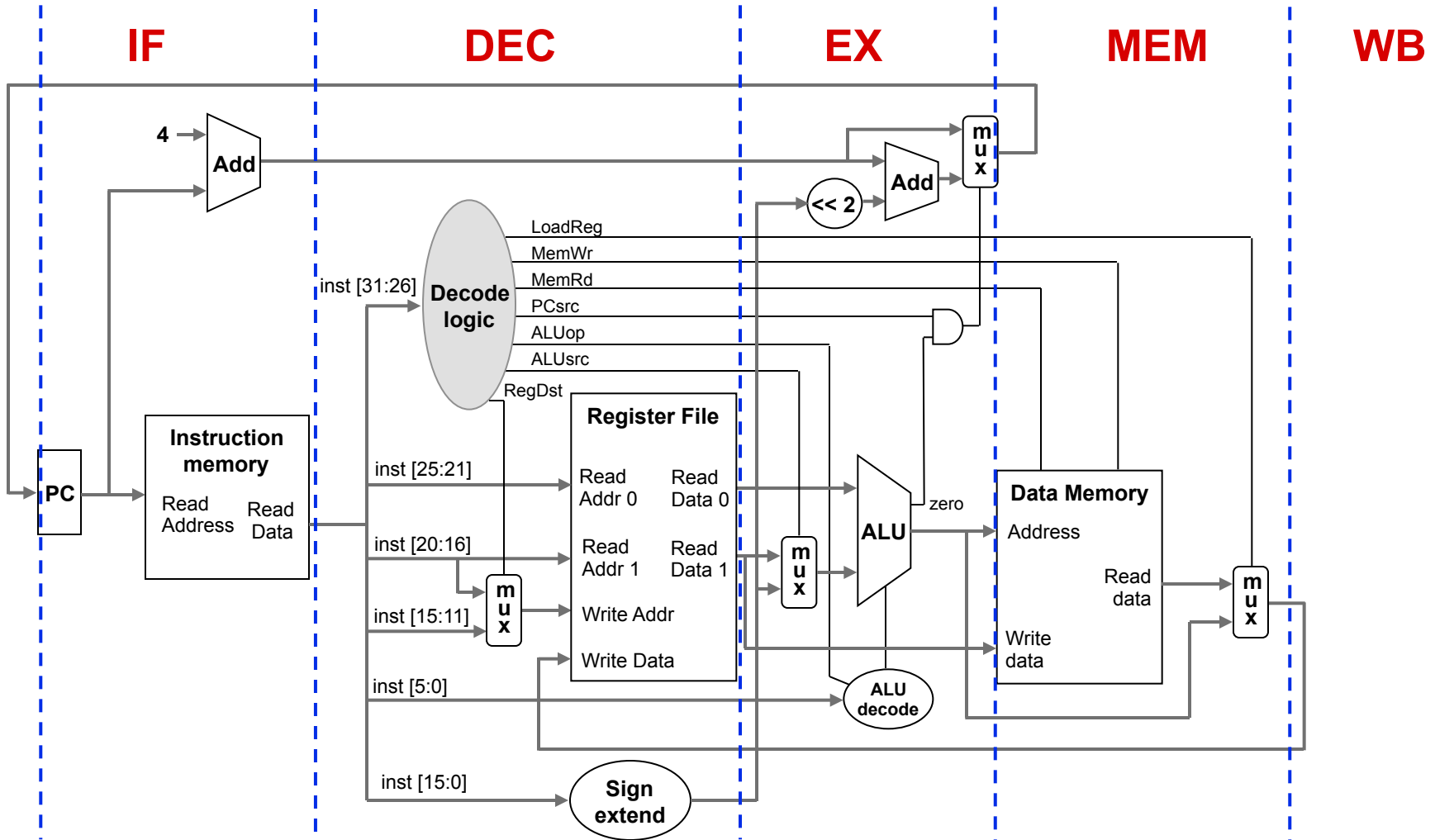


# Controlling Instruction Execution

- Control signals driven by combinational logic, based on instruction opcode



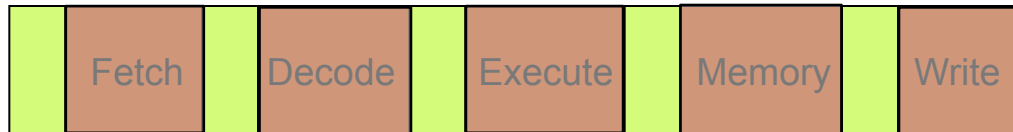
# Putting it all together



# Motivating Pipelined Instruction Execution

---

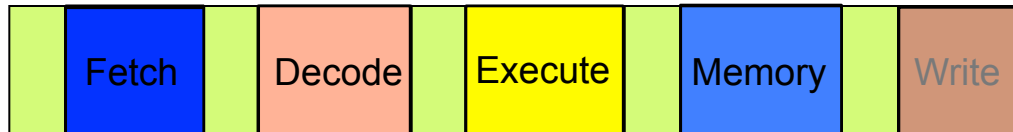
## Phases of Instruction Execution



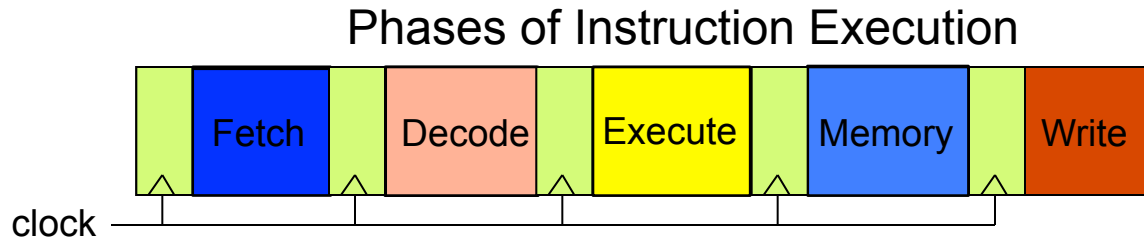
# Pipelined Instruction Execution

---

## Phases of Instruction Execution



# Pipelined Instruction Execution



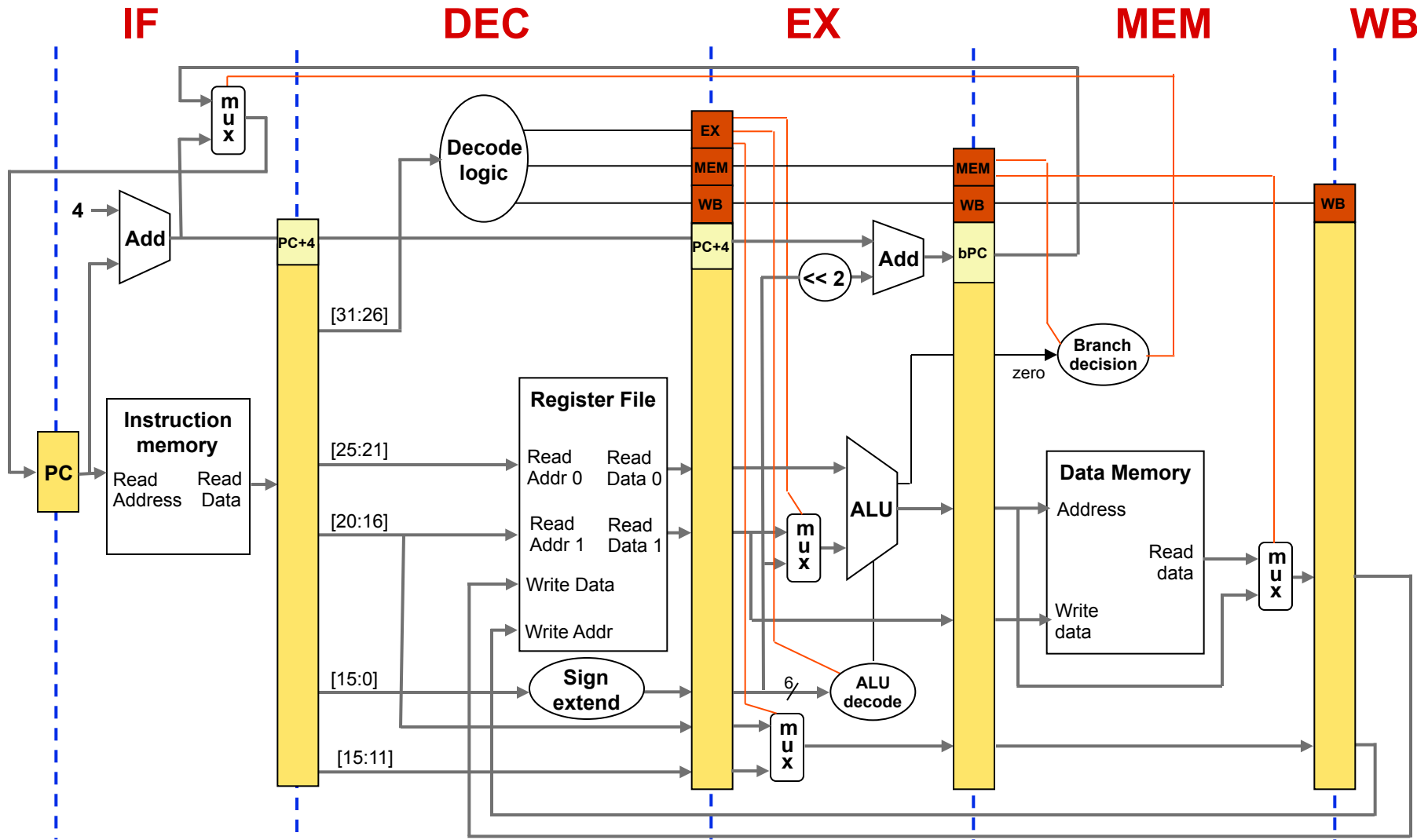
**Problem: need a way to separate instruction state between pipeline stages**

**Solution: clocked pipeline latches (registers)**

- Operands (e.g., register values)
- Intermediate values (e.g., ld/st address)
- Control signals

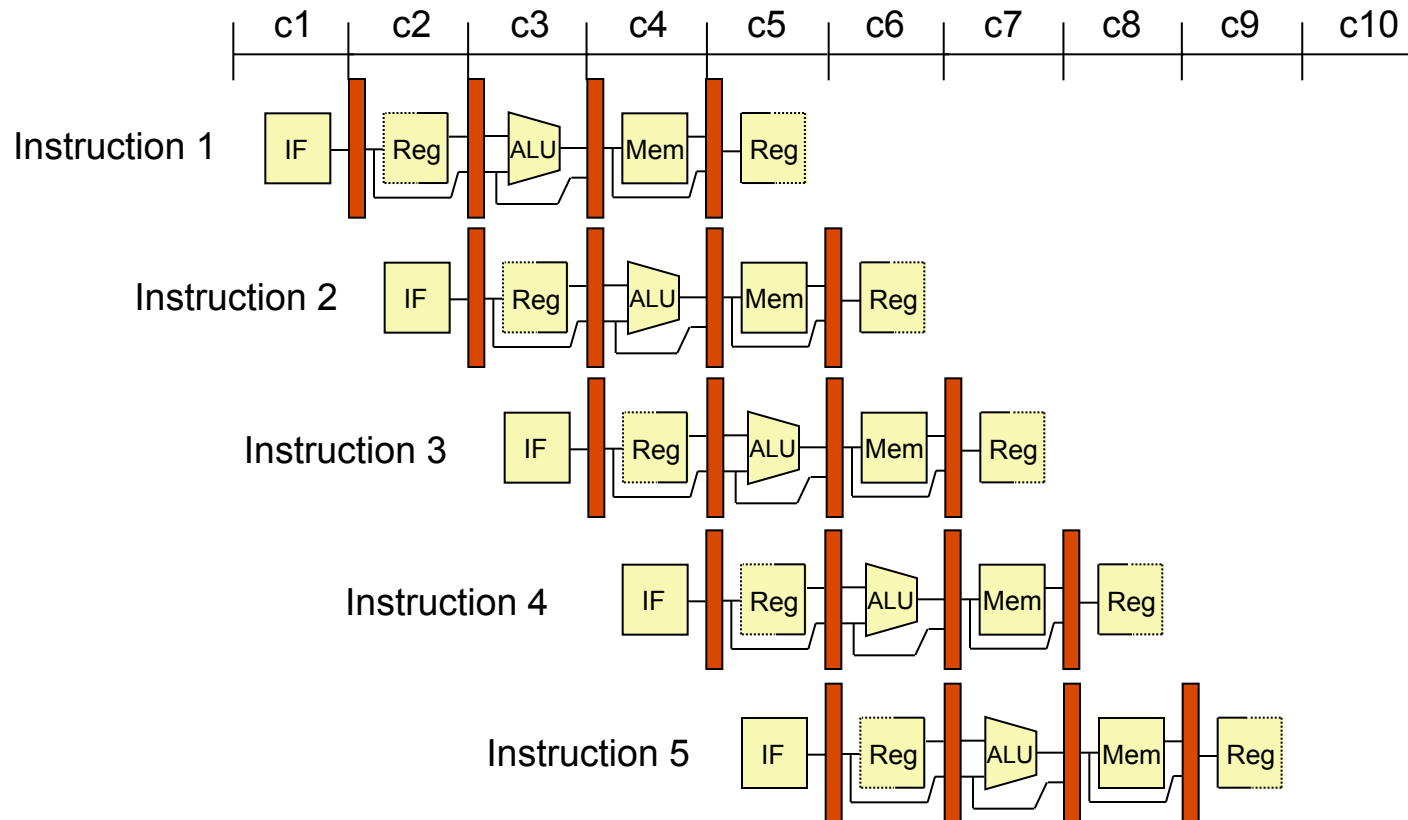


# CPU Pipeline Structure



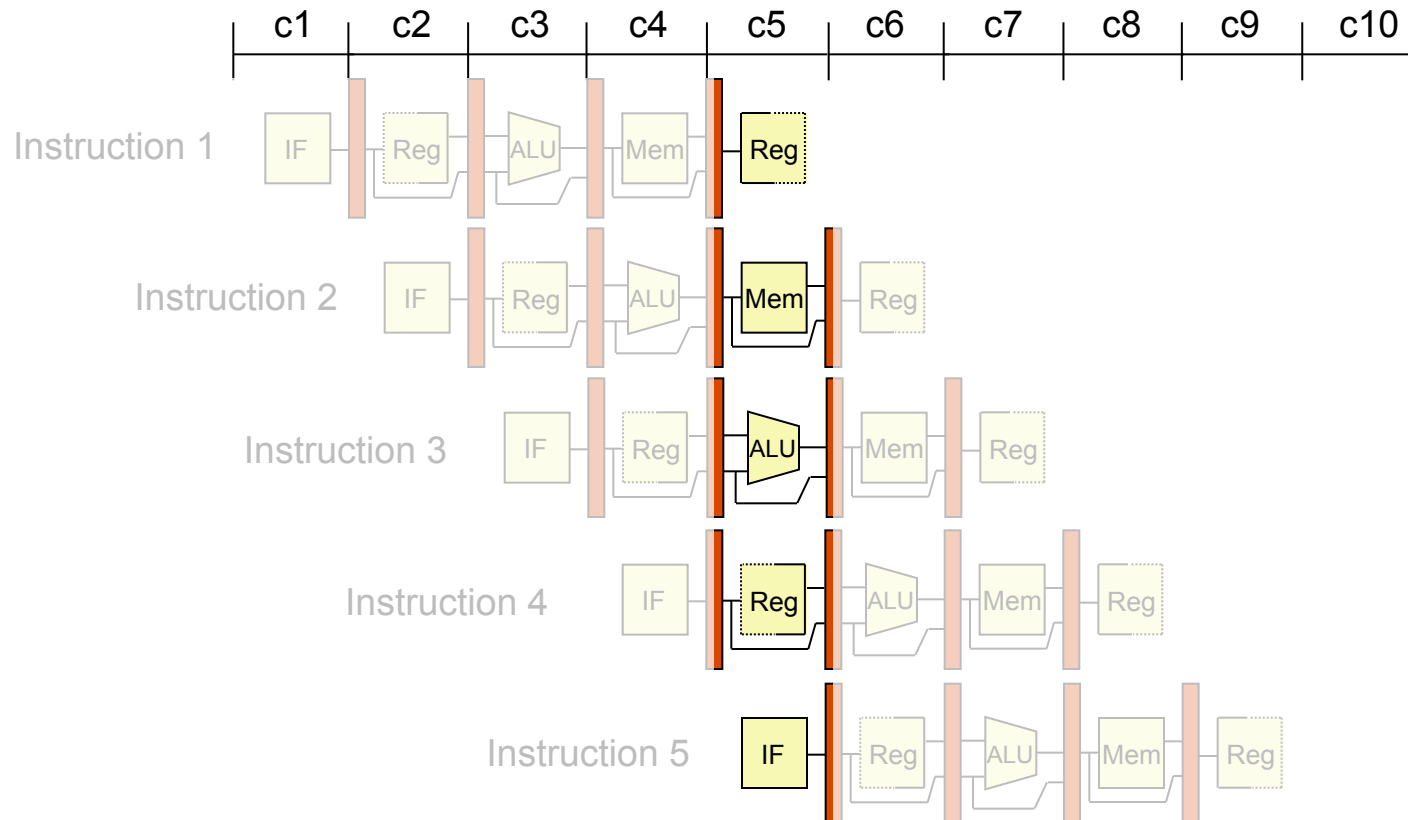
# Representing a sequence of instructions

- Space-time diagram of pipeline
- Think of each instruction as a time-shifted pipeline



# Information flow constraints

- Information from one instruction to any successor must always move from left to right



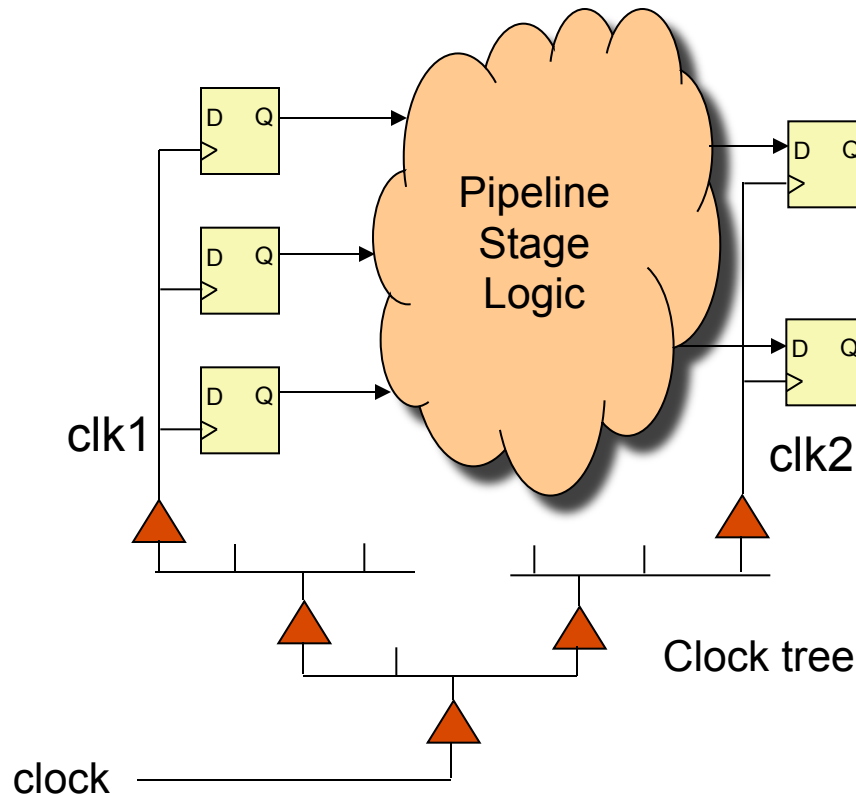
# Another way to represent pipeline timing

- A similar, and slightly simpler, way to represent pipeline timing:
  - Clock cycles progress left to right
  - Instructions progress top to bottom
  - Time at which each instruction is present in each pipeline stage is shown by labelling appropriate cell with pipeline name

Instruction \ cycle	1	2	3	4	5	6	7	8	9
instruction 1	IF	DEC	EX	MEM	WB				
instruction 2		IF	DEC	EX	MEM	WB			
instruction 3			IF	DEC	EX	MEM	WB		
instruction 4				IF	DEC	EX	MEM	WB	
instruction 5					IF	DEC	EX	MEM	WB

# Implementation Issues: Pipeline balance

- Each pipeline stage is a combinational logic network
  - Registered inputs and outputs
  - Longest circuit delay through all stages determines clock period



Ideally, all delays through every pipeline stage are identical

In practice this is hard to achieve

# Pipeline Hazards

---

- Hazards are pipeline events that restrict the pipeline flow
- They occur in circumstances where two or more activities cannot proceed in parallel
- There are three types of hazard:
  - **Structural Hazards**
    - Arise from resource conflicts, when a set of actions have to be performed sequentially because there is not sufficient resource to operate in parallel
  - **Data Hazards**
    - Occur when one instruction depends on the result of a previous instruction, and that result is not yet available. These hazards are exposed by the overlapped execution of instructions in a pipeline
  - **Control Hazards**
    - These arise from the pipelining of branch instructions, and other activities that change the PC.

# Structural Hazards

- Multi-cycle operations
- Memory or register file port restrictions

Example structural hazard caused by having only one memory port

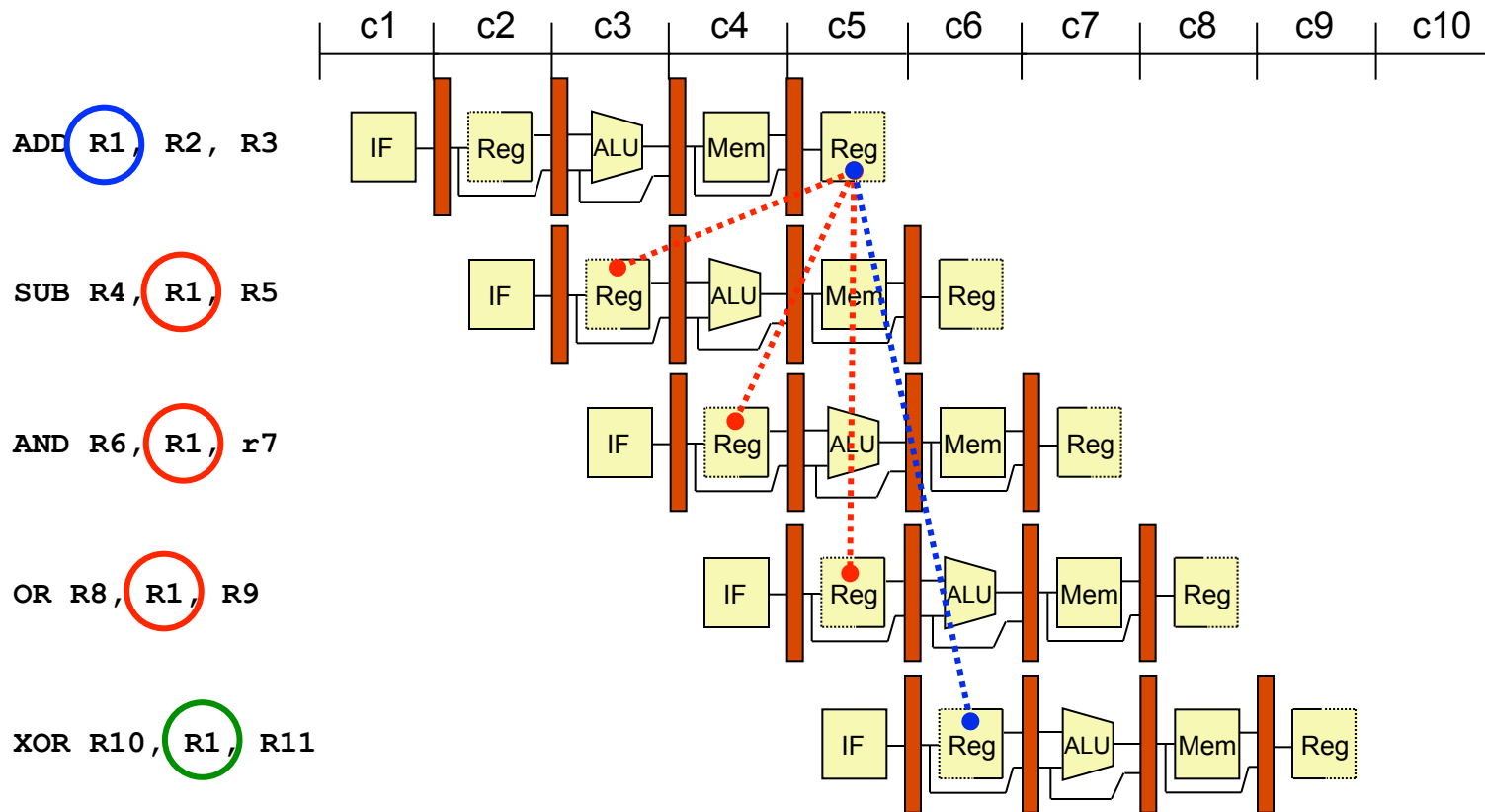
Instruction \ cycle	1	2	3	4	5	6	7	8	9	10
lw \$1,(\$2)	IF	DEC	EX	MEM	WB					
instruction 2		IF	DEC	EX	MEM	WB				
instruction 3			IF	DEC	EX	MEM	WB			
instruction 4				IF	DEC	EX	MEM	WB		
instruction 5					IF	DEC	EX	MEM	WB	

Effect is to STALL instruction 4, delaying its entry to IF by one cycle

Instruction \ cycle	1	2	3	4	5	6	7	8	9	10
lw \$1,(\$2)	IF	DEC	EX	MEM	WB					
instruction 2		IF	DEC	EX	MEM	WB				
instruction 3			IF	DEC	EX	MEM	WB			
instruction 4				IF	IF	DEC	EX	MEM	WB	
instruction 5						IF	DEC	EX	MEM	WB

# Data Hazards

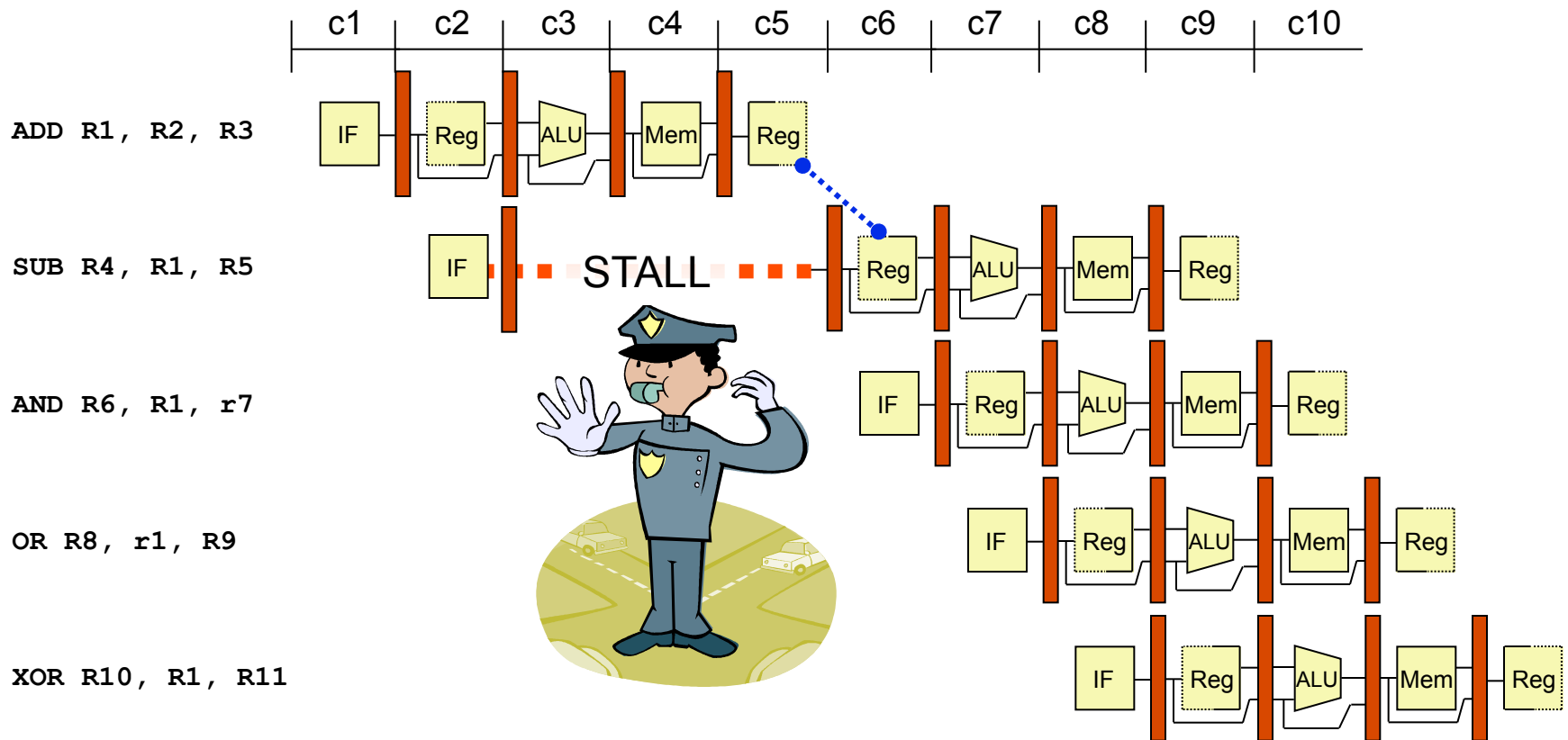
- Overlapped execution of instructions means information may be required before it is available.





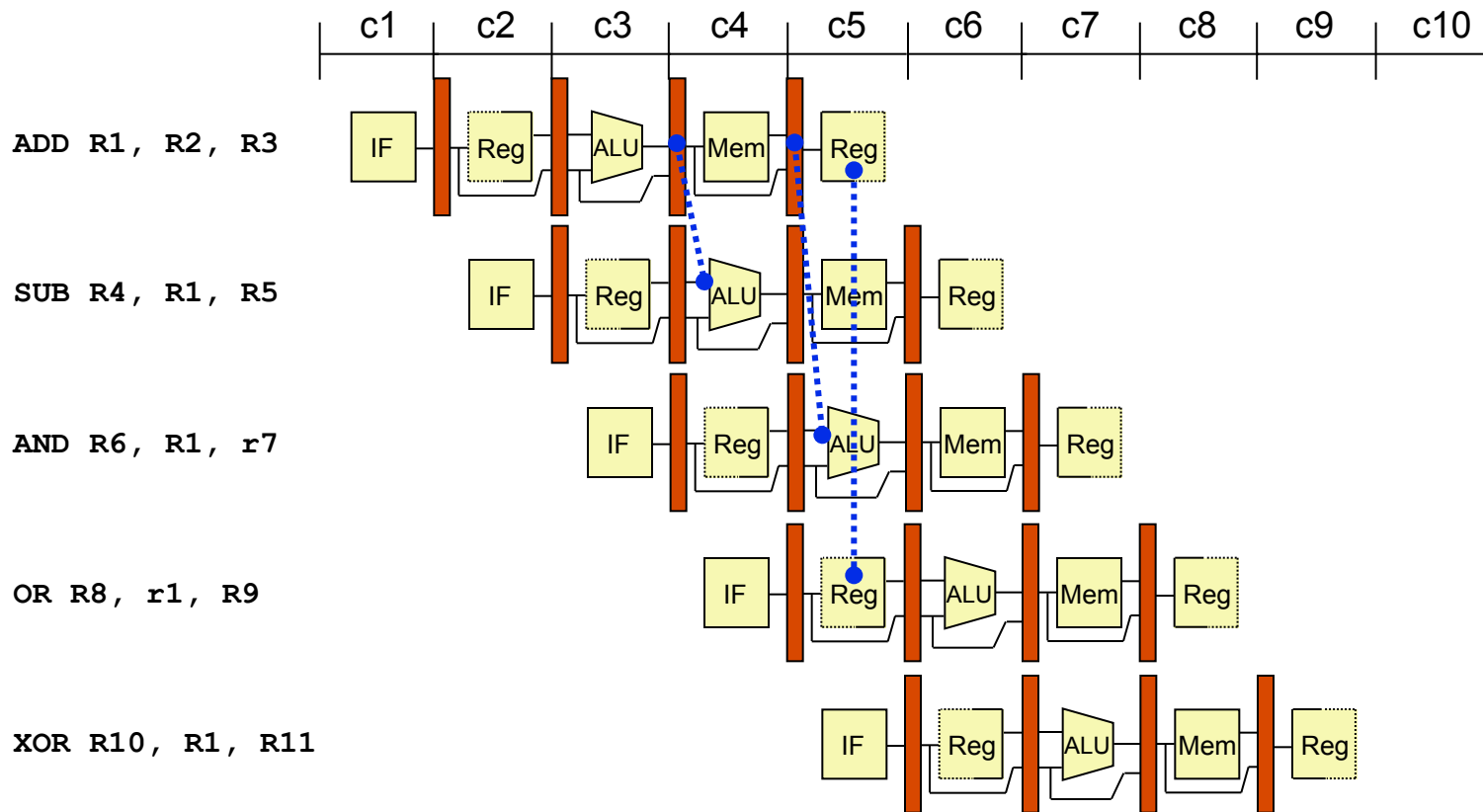
# Data hazards lead to pipeline stalls

- SUB instruction must wait until R1 has been written to register file
- All subsequent instructions are similarly delayed

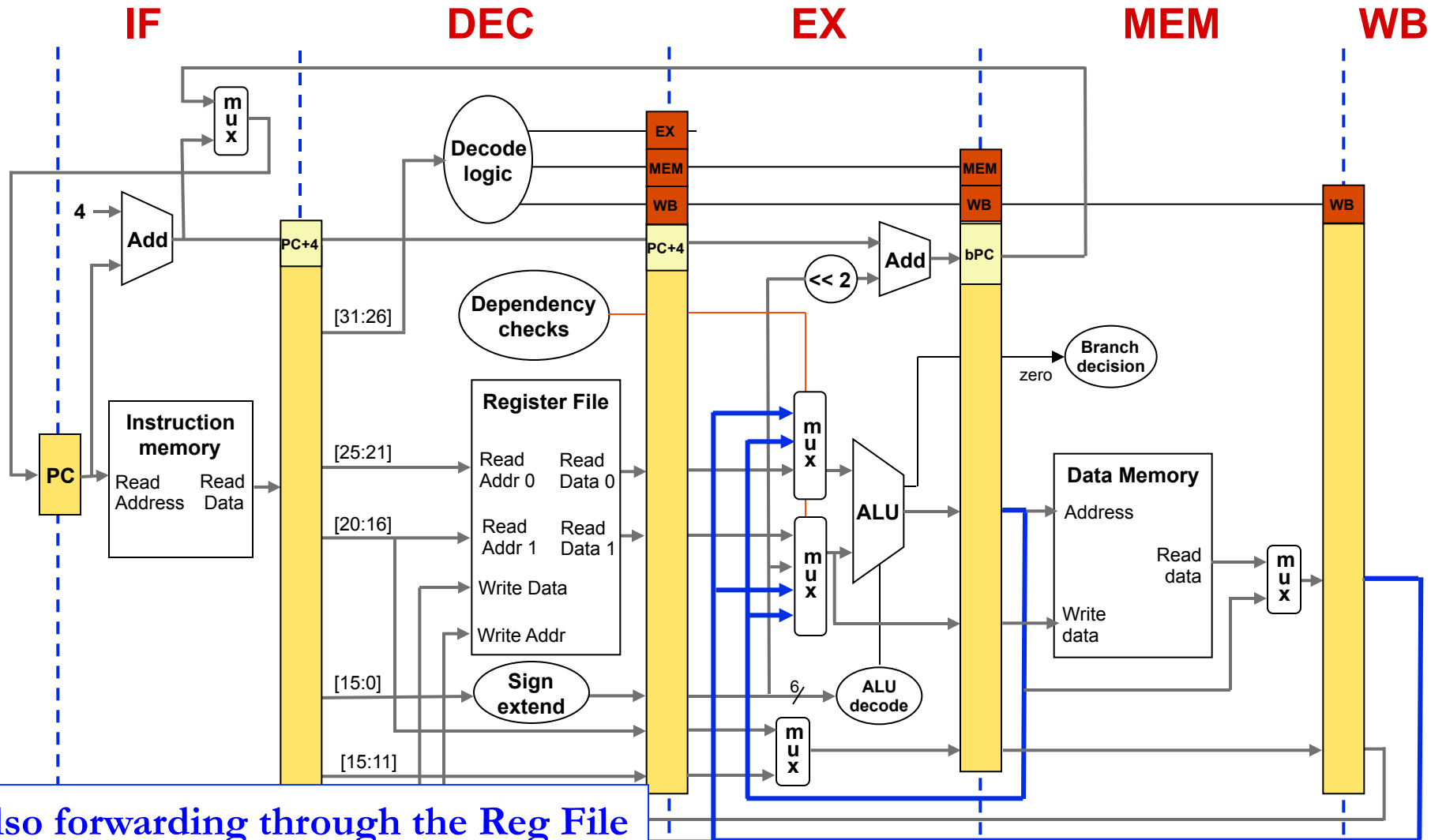


# Minimising data hazards by data-forwarding

- Key idea is to bypass the register file and forward information, as soon as it becomes available within the pipeline, to the place it is needed.



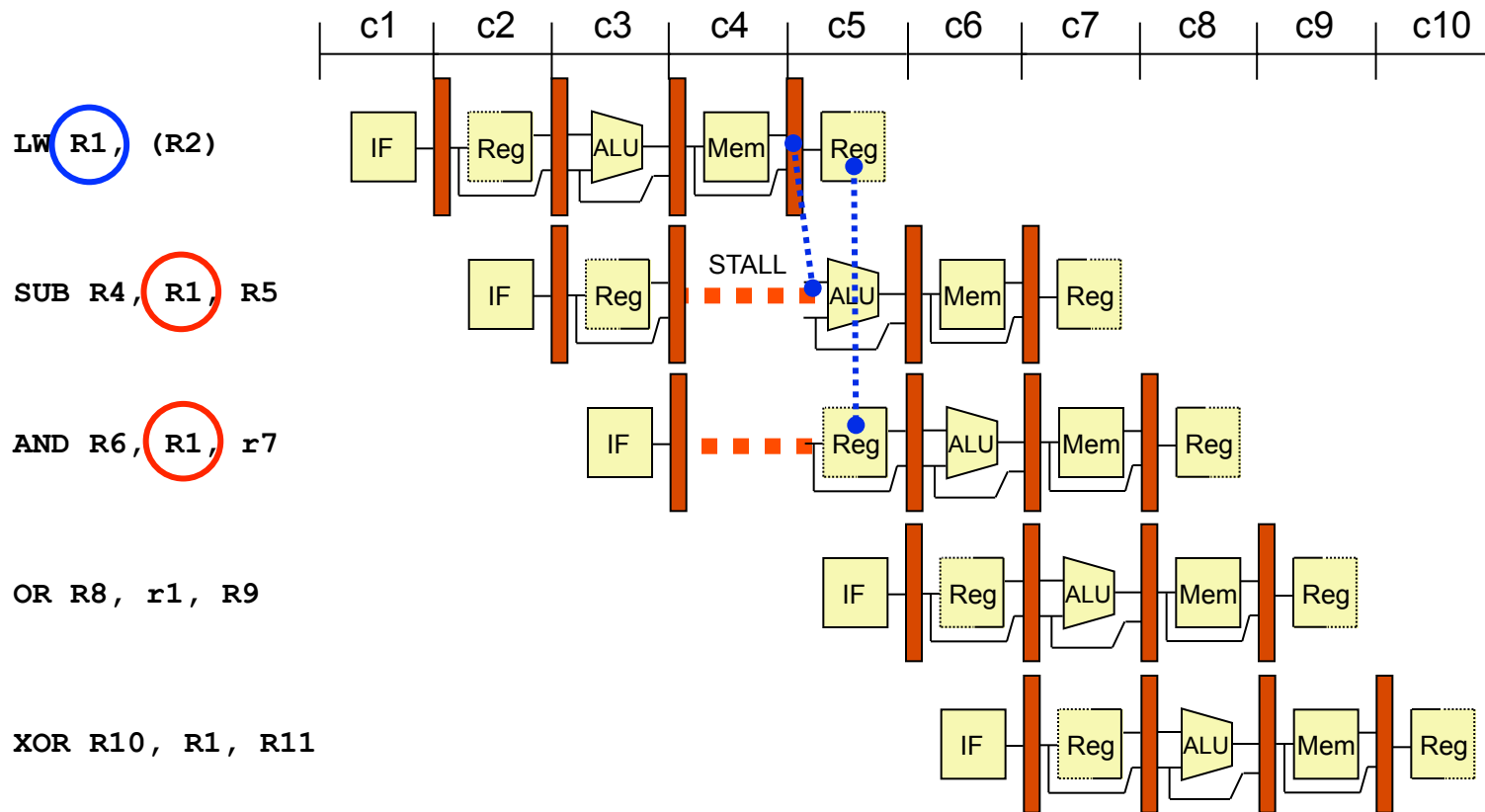
# CPU pipeline showing forwarding paths



Also forwarding through the Reg File  
(no new datapath needed)

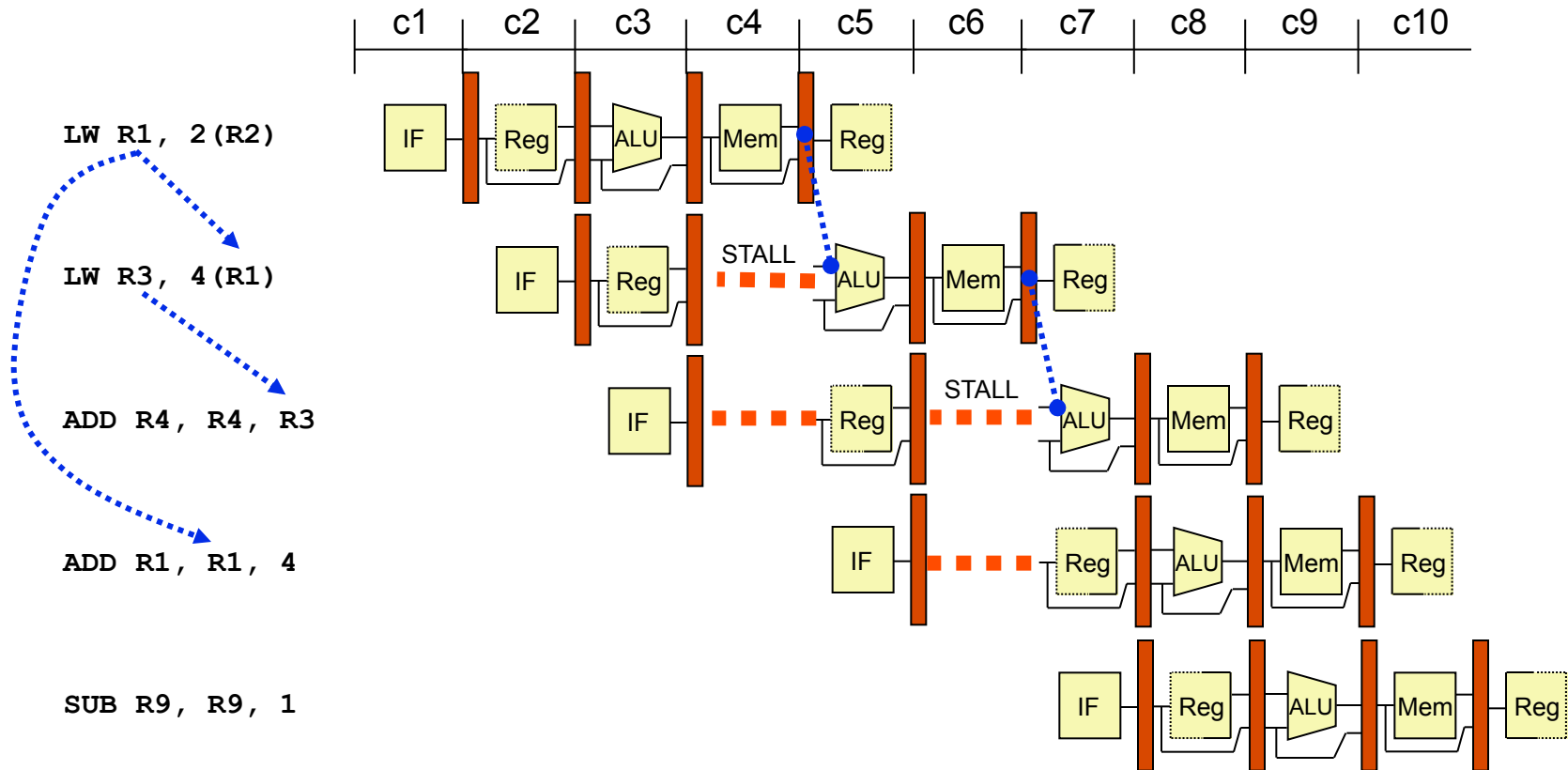
# Data hazards requiring a stall

- Hazards involving the use of a Load result usually require a stall, even if forwarding is implemented



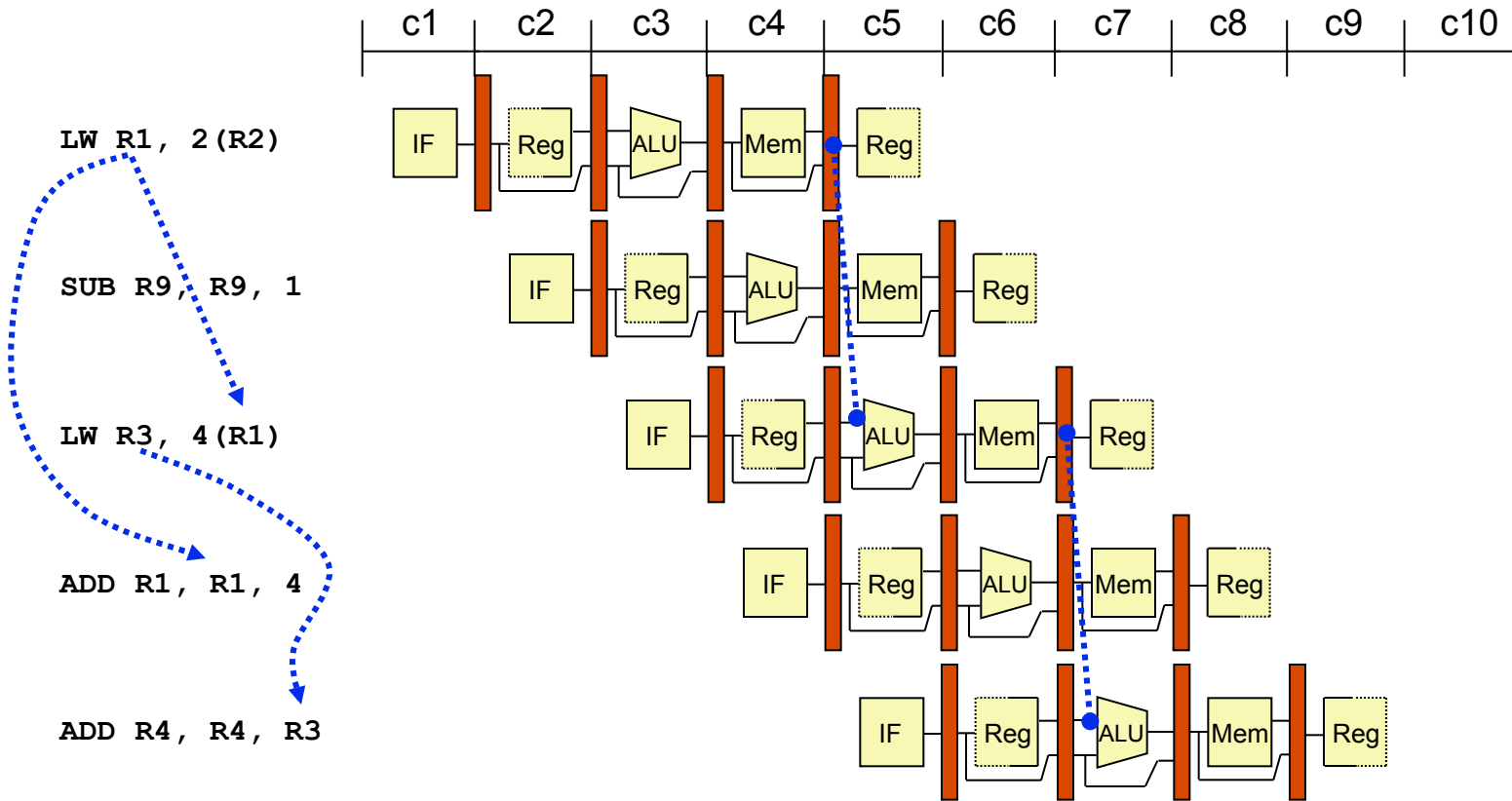
# Code scheduling to avoid stalls (before)

- Hazards involving the use of a Load may be avoided by reordering the code



# Code scheduling to avoid stalls (after)

- SUB is entirely independent of other instructions – place after 1<sup>st</sup> load
- ADD to R1 can be placed after LW to R3 to hide the load delay on R3



# General Performance Impact of Hazards

---

$$\text{Speedup from pipelining: } S = \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{clock}_{\text{unpipelined}}}{\text{clock}_{\text{pipelined}}}$$

$$\text{CPI}_{\text{pipelined}} = \text{ideal CPI} + \text{stall cycles per instruction} = 1 + \text{stall cycles per instruction}$$

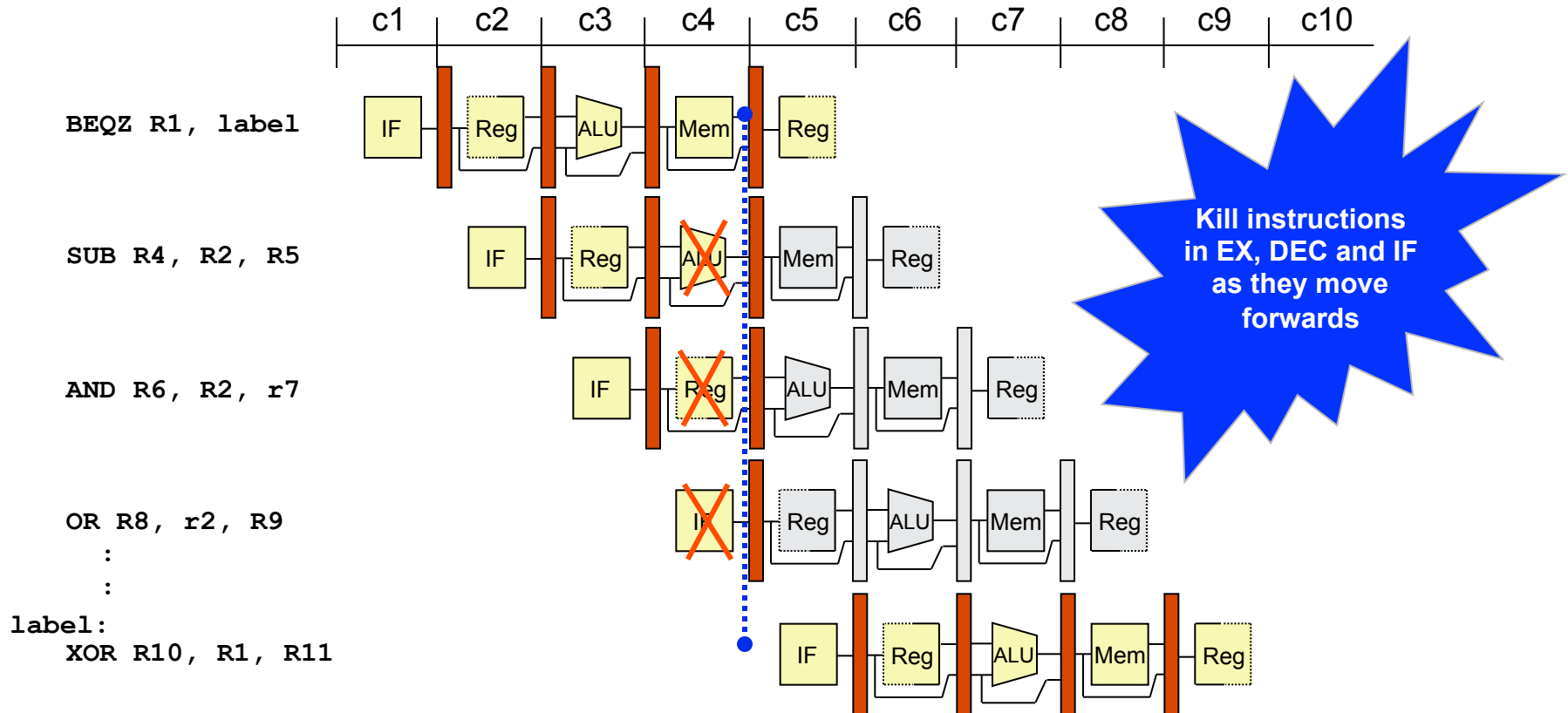
$$\text{CPI}_{\text{unpipelined}} \sim \text{pipeline depth}$$

$$\frac{\text{clock}_{\text{unpipelined}}}{\text{clock}_{\text{pipelined}}} \sim 1$$

$$S = \frac{\text{pipeline depth}}{1 + \text{stall cycles per instruction}}$$

# Control Hazards

- When a branch is executed, PC is not affected until the branch instruction reaches the MEM stage.
- By this time 3 instructions have been fetched from the fall-through path.



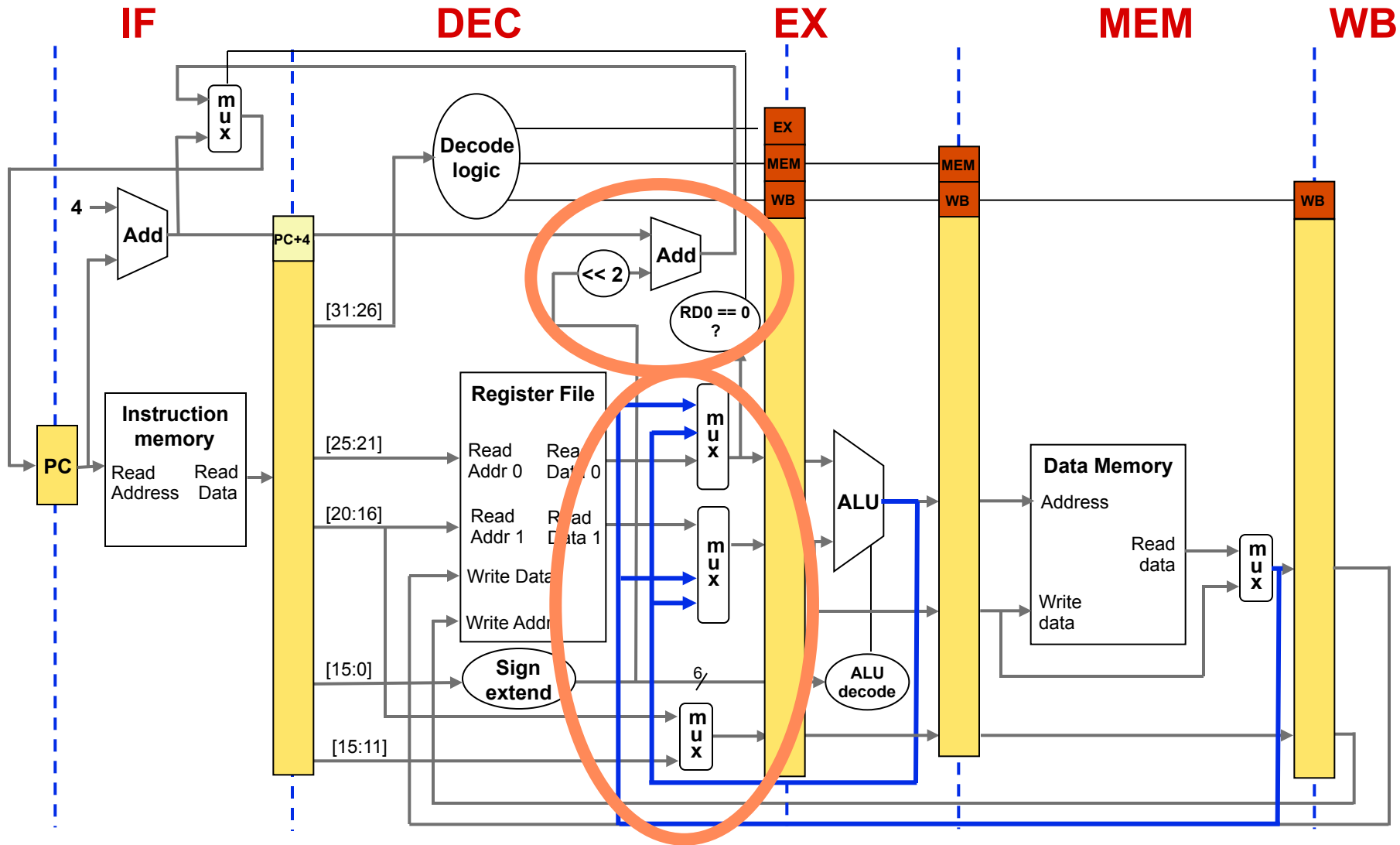


# Effect of branch penalty on CPI

---

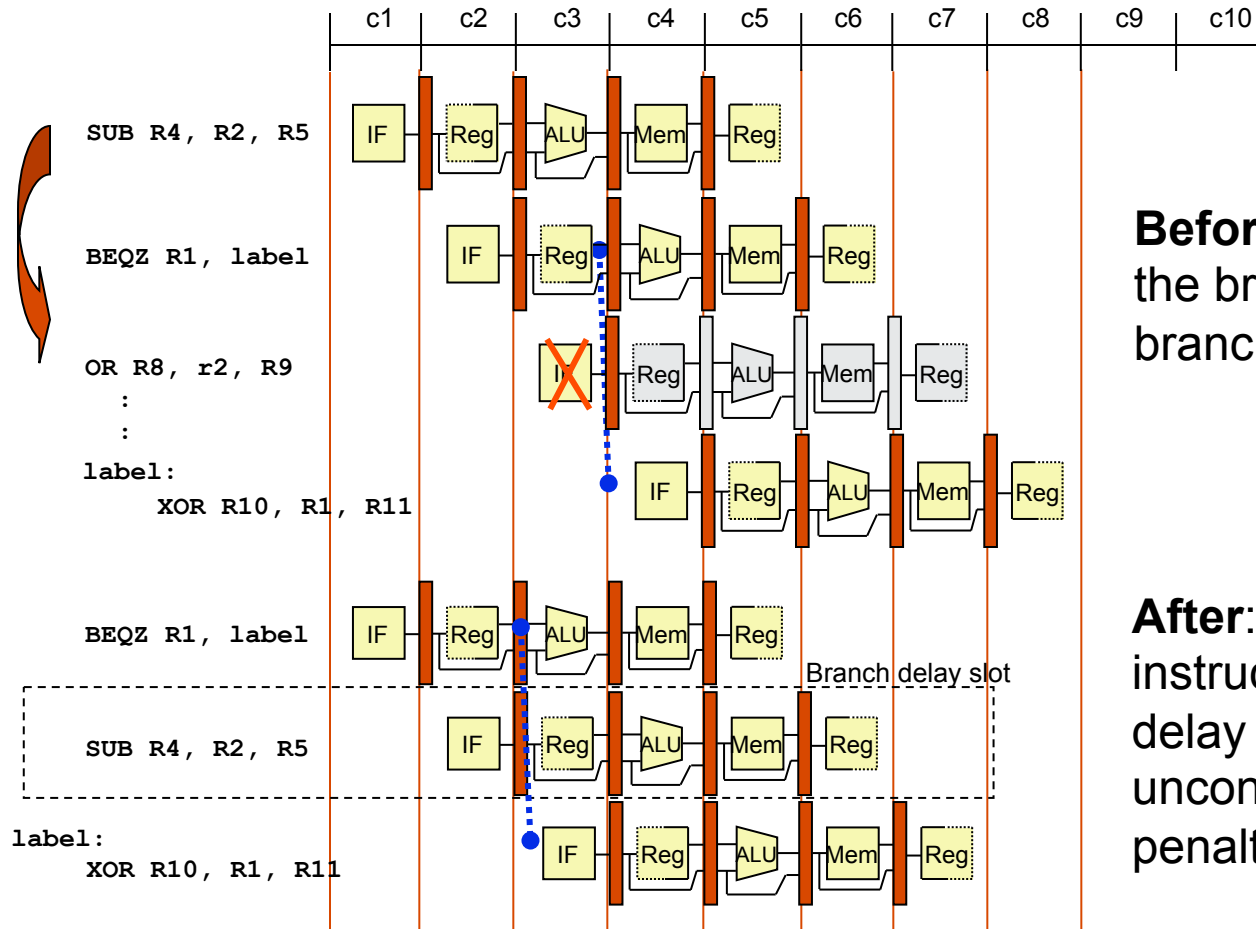
- In this example pipeline the cost of each branch is:
  - 1 cycle, if the branch is not taken (due to load-delay slot)
  - 4 cycles, if the branch is taken
- If an equal number of branches are taken and not taken, and if 20% of all instructions are branches (a reasonable assumption), then
  - $\text{CPI} = 0.8 + 0.2 \cdot 2.5 = 1.3$
  - This is a significant reduction in performance
- If the pipeline was deeper, with 2 stages for ALU and 2 stages for Decode, then:
  - Cost of taken branch would be 6 cycles
  - $\text{CPI} = 0.8 + 0.2 \cdot 3.5 = 1.5$
- Deeper pipelines have greater branch penalties, and potentially higher CPI
- Pentium 4 (Prescott) had 31 pipeline stages! (this was too deep)
- Several important techniques have been developed to reduce branch penalties
  - Early branch outcome
  - Delayed branches with branch delay slot(s)
  - Branch prediction (static and dynamic)

# Early branch outcome calculation - BEQZ, BNEZ



# Delayed branch execution (branch delay slot)

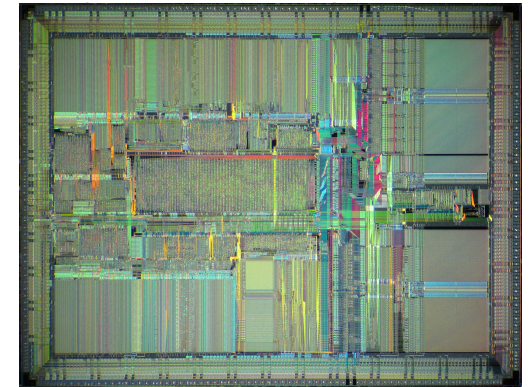
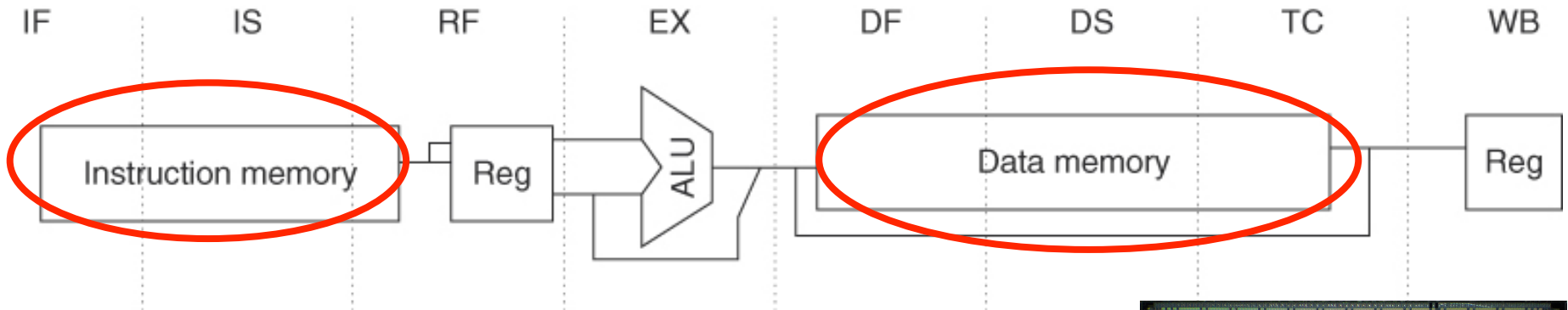
- Always execute the instruction immediately after the branch, regardless of branch outcome



**Before:** instruction after the branch gets killed if the branch is taken

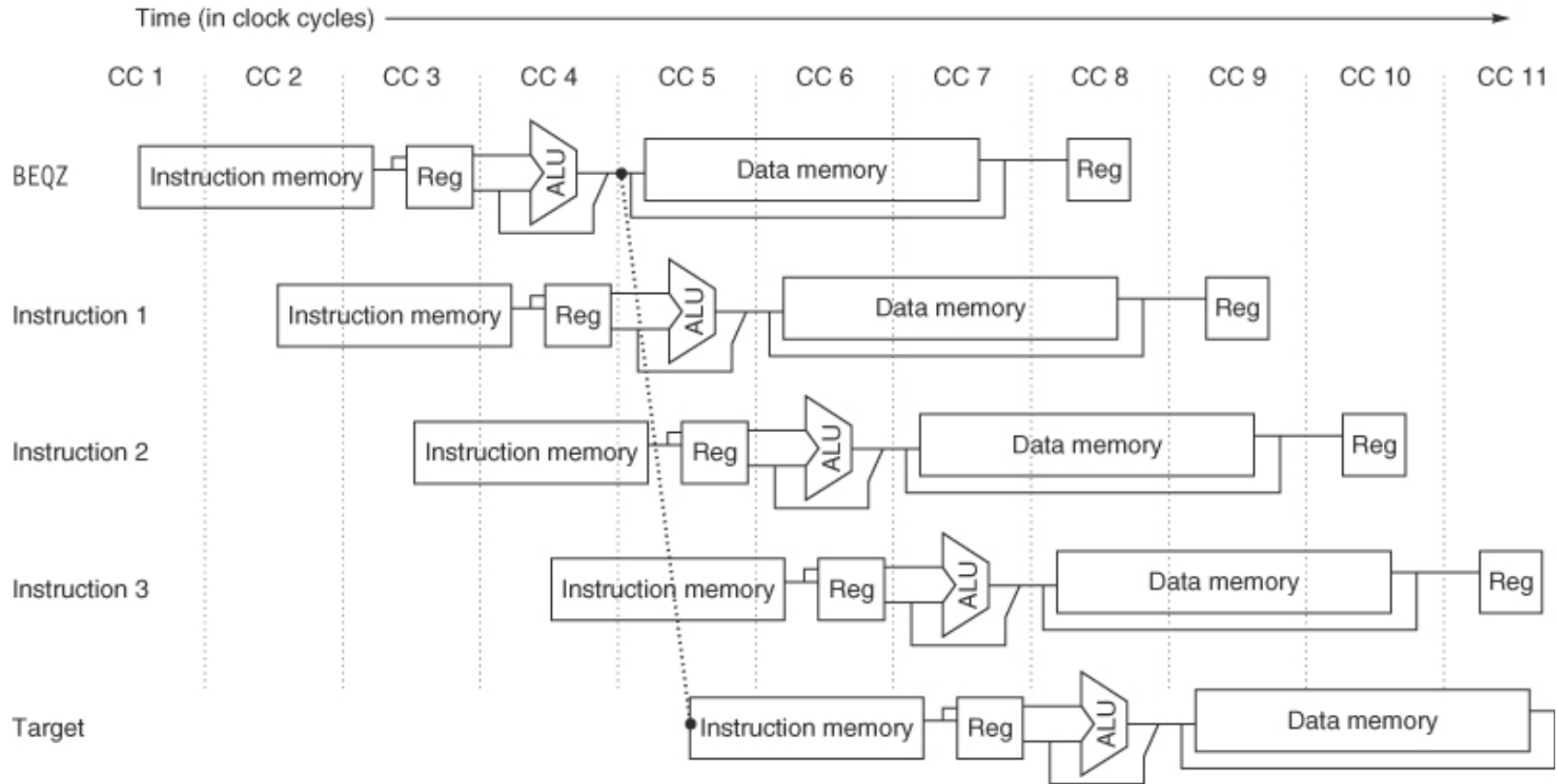
**After:** by moving the SUB instruction into the branch delay slot, and executing it unconditionally, the 1-cycle penalty is eliminated

# Case Study: Pipelining in MIPS R4000



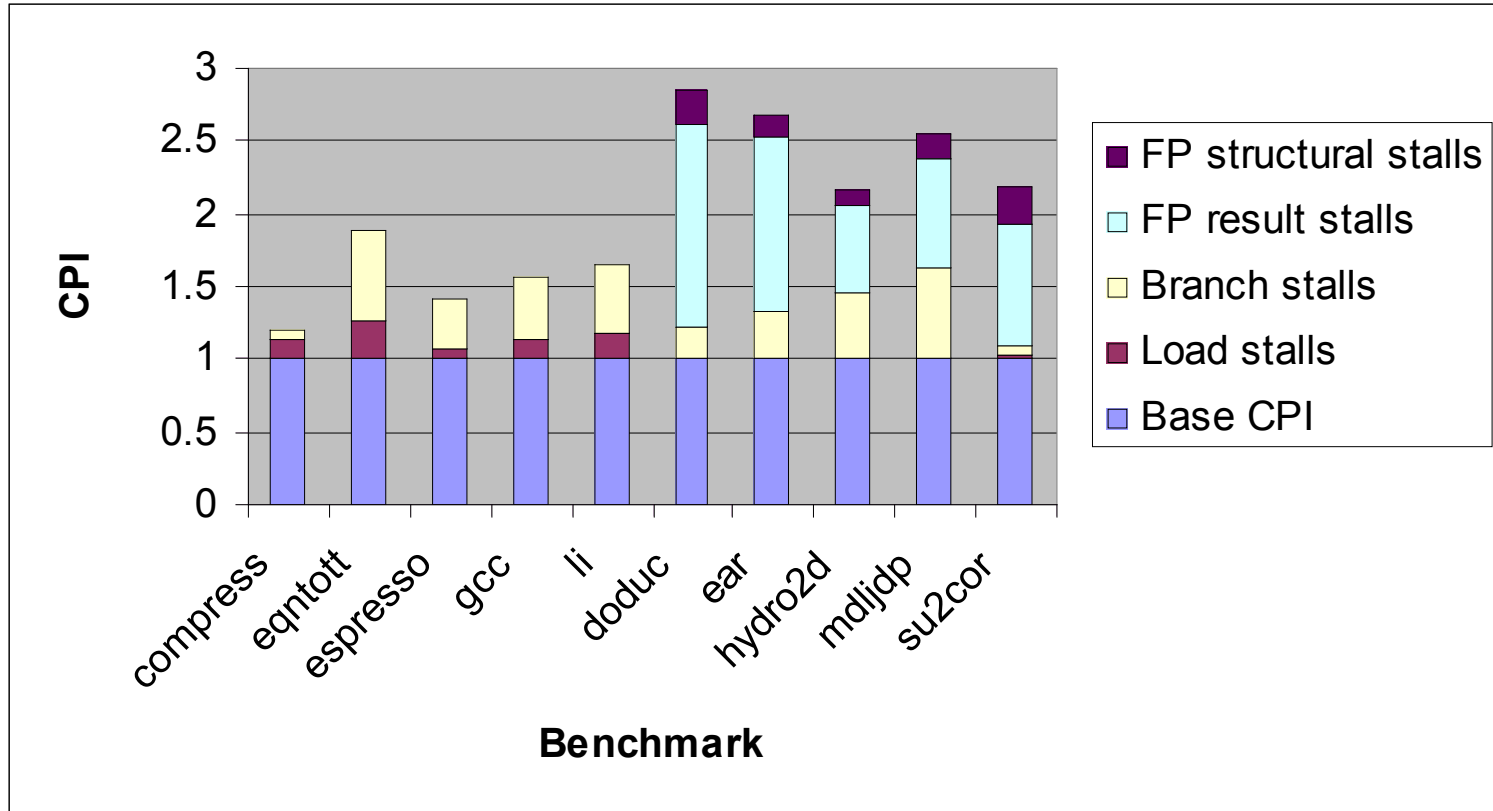
- Introduced in early 90s
  - 1.2 million transistors, 250 Mhz peak frequency
  - 64-bit CPU – one of the first!
- Notable feature: pipelined memory accesses

# Branch delay in MIPS R4000



**3-cycle branch taken delay**

# Impact of Branch Hazards on CPI



H&P 5/e  
Fig. C.52

**Bottom-line: CPI increase of 0.06 – 0.62 cycles**

# Assignments

---

- Assignment 1: out Mon, Feb 6  
due Mon, Feb 20
- Assignment 2: out late Feb, due in 2 weeks