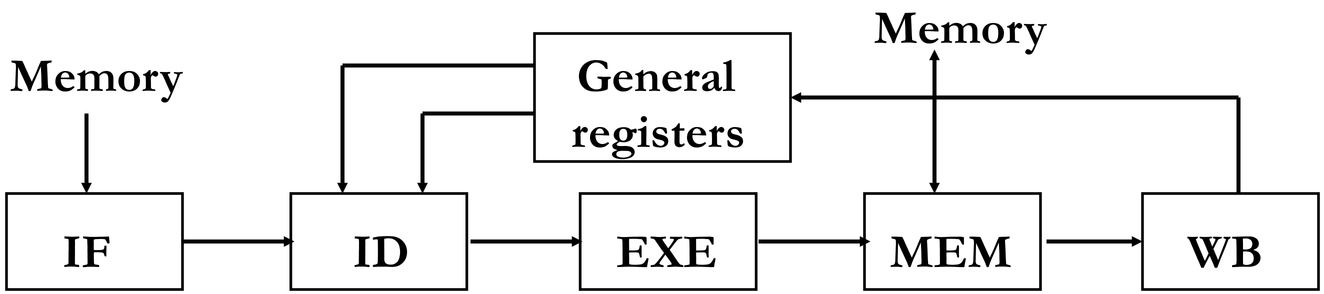
Improving Performance: Pipelining



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)

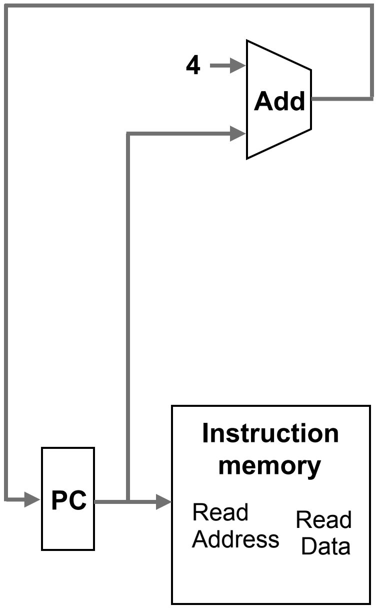
Phases of Instruction Execution

- 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
  - Add, subtract, shift, logical, etc.
  - Address calculation is simply unsigned 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 Instruction Cache at address given by PC
- Increment PC, i.e. PC = PC + sizeof(instruction)



MIPS R-type instruction format

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	reg rs	reg rt	reg rd	shamt	funct

↑ Destination register for R-type format (reg rd)

`add $1, $2, $3`

special	$2	$3	$1		add

`sll $4, $5, 16`

special	$5	$4		16	sll

MIPS I-type instruction format

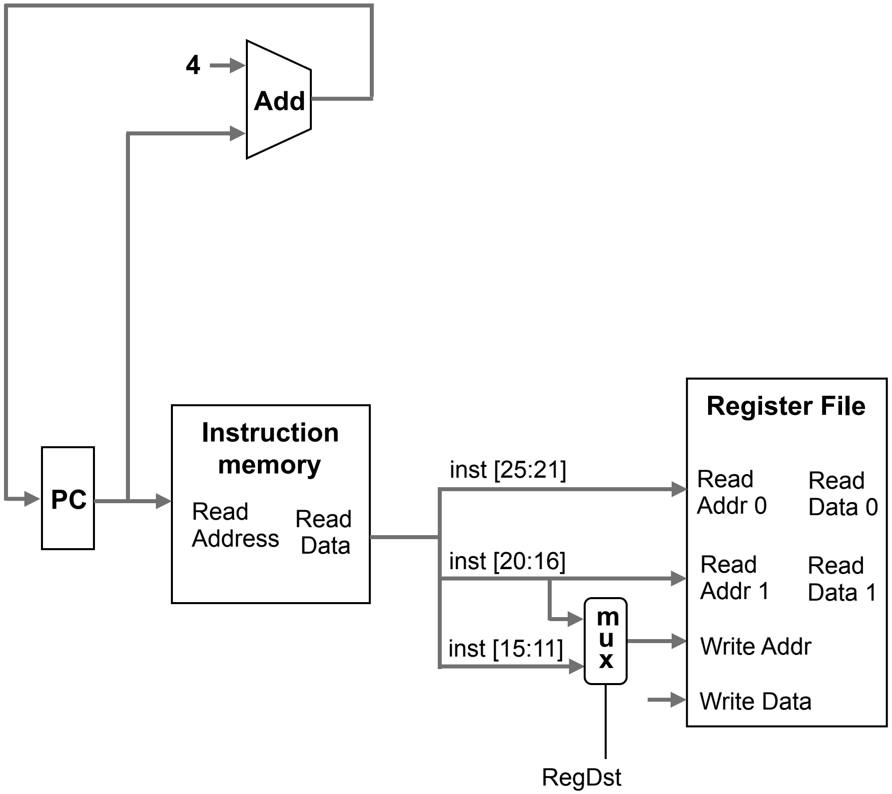
6 bits	5 bits	5 bits	16 bits
opcode	reg rs	reg rt	immediate value/addr

↑ (reg rt) Destination register for Load

Instruction				
`lw $1, offset($2)`	lw	$2	$1	address offset
`beq $4, $5, .Label1`	beq	$4	$5	(PC - .Label1) >> 2
`addi $1, $2, -10`	addi	$2	$1	0xfff6

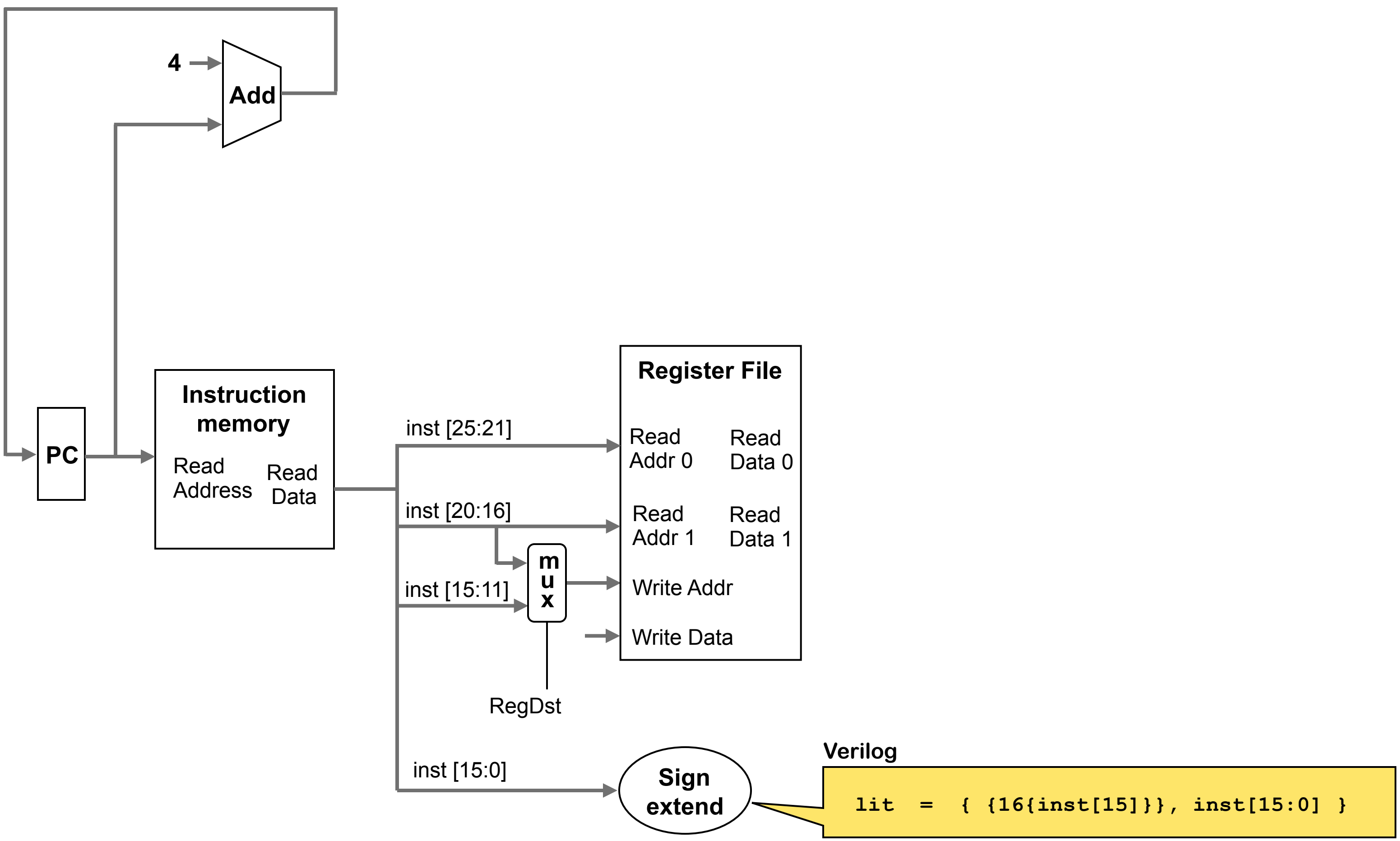
Reading Registers

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



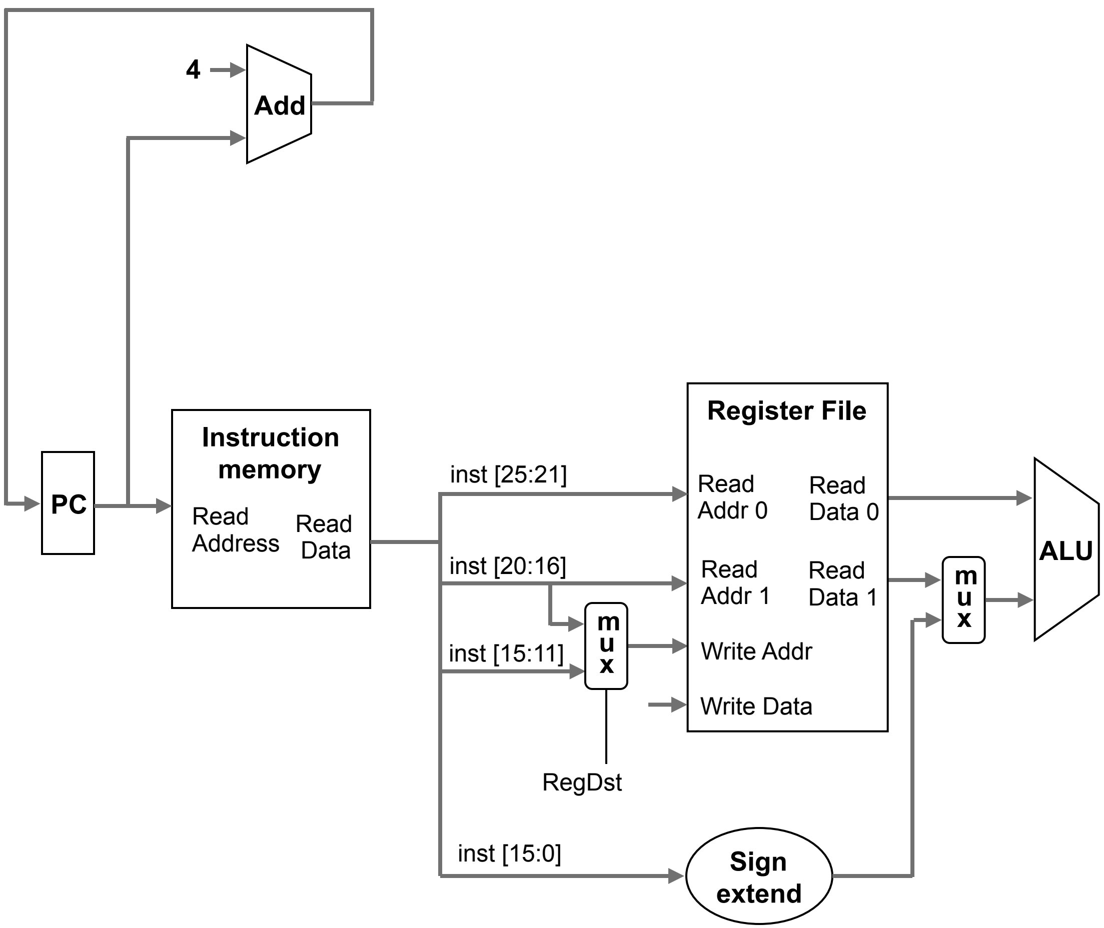
Extracting the literal operand

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



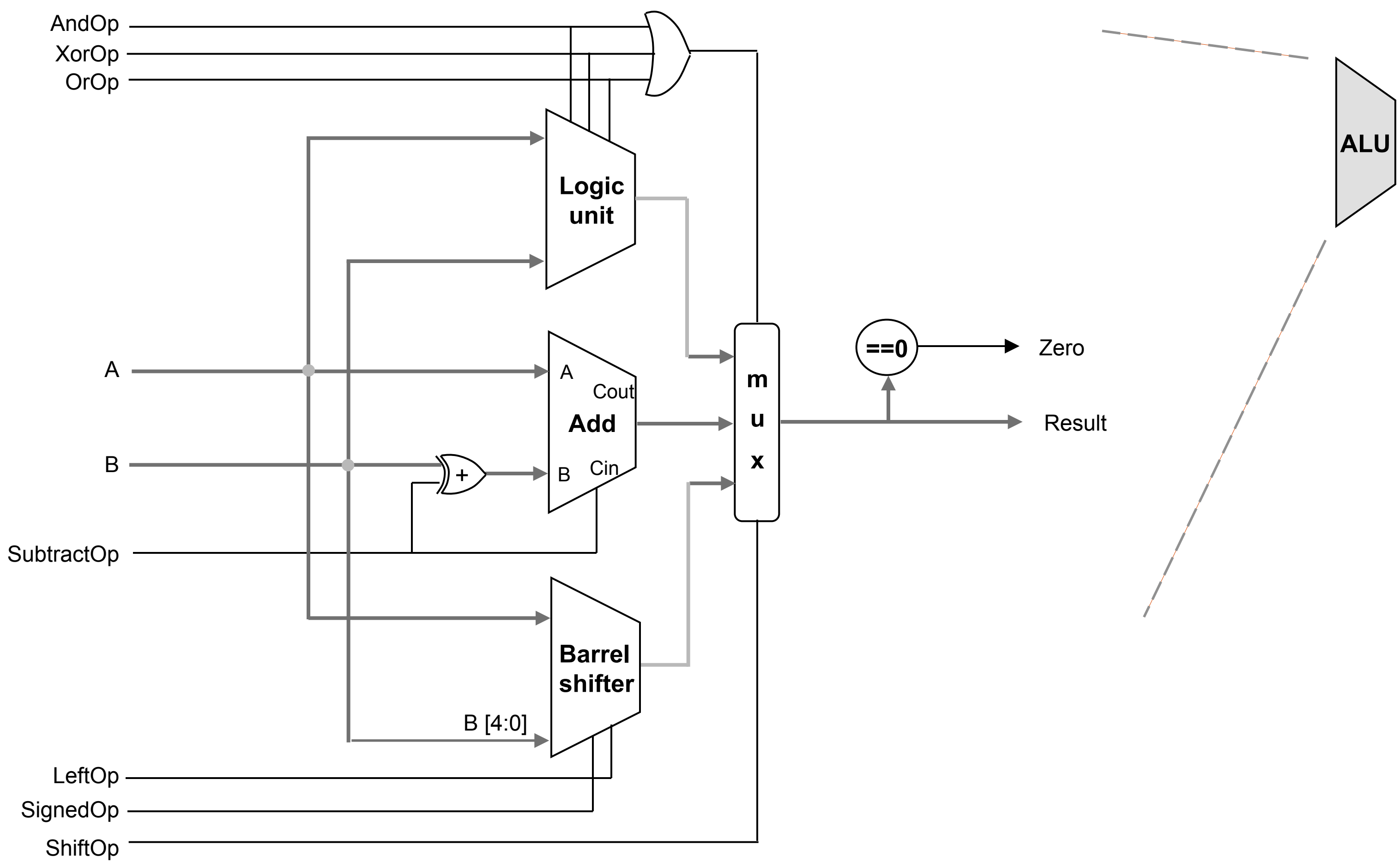
Performing the Arithmetic

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



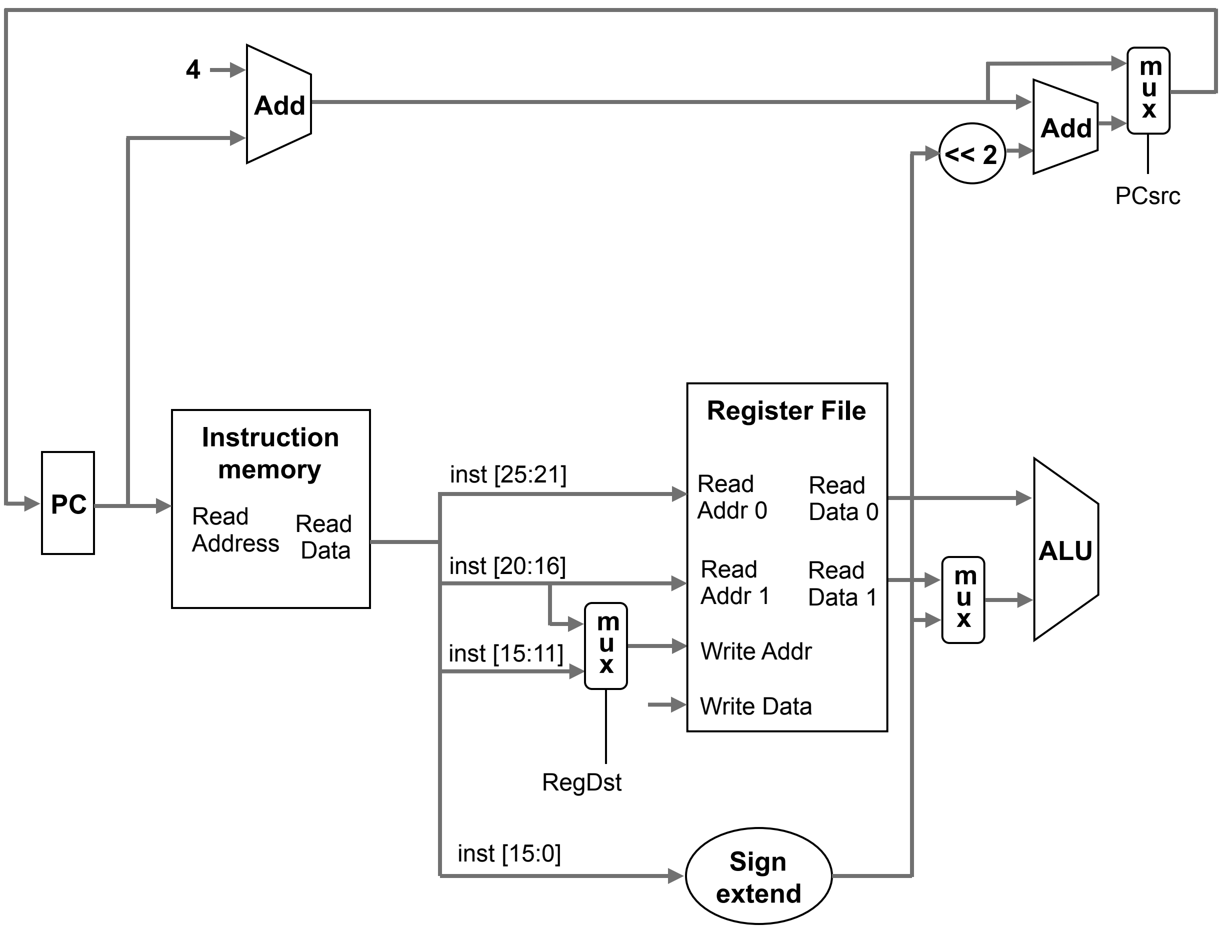
Inside the ALU

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



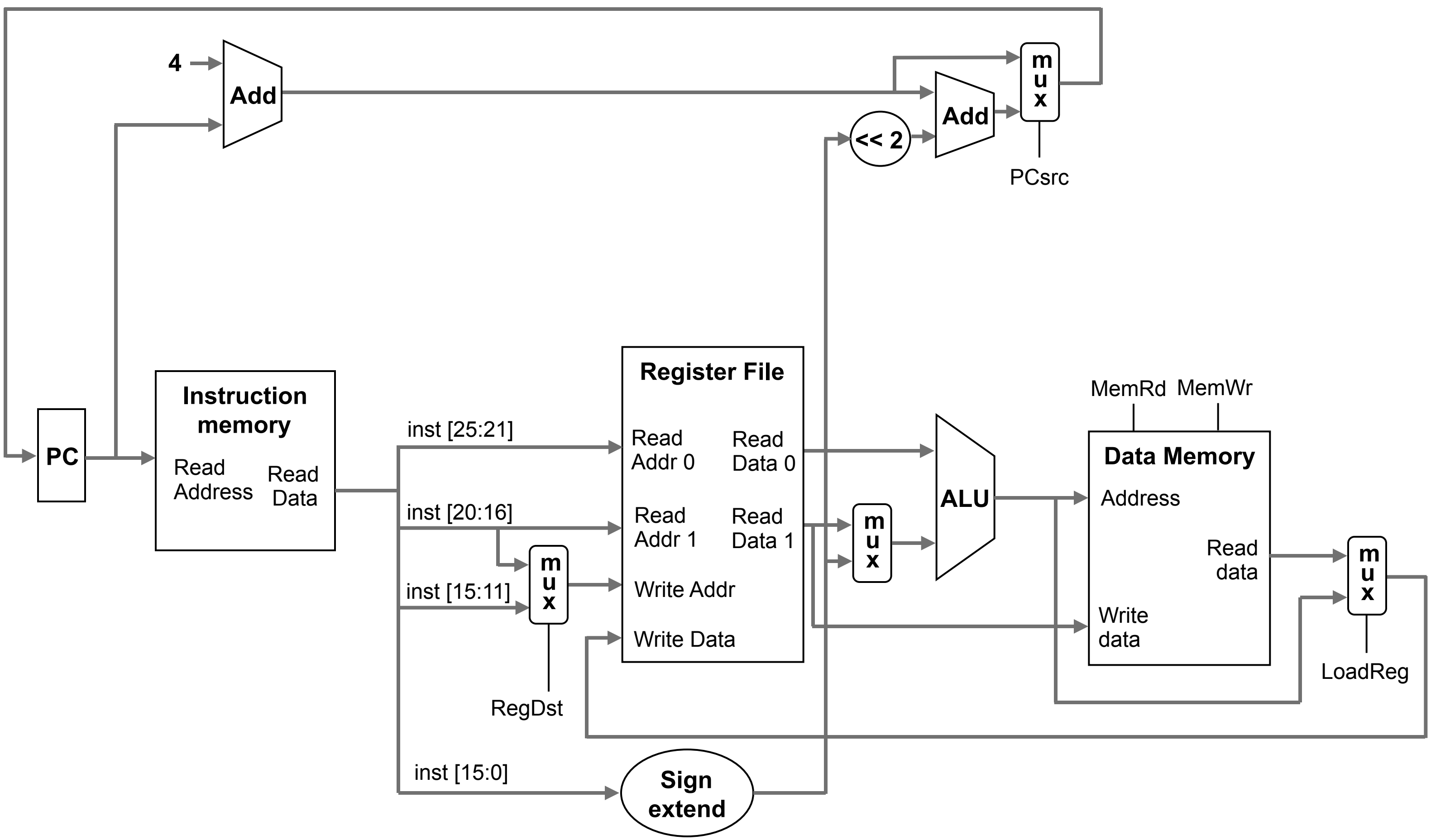
Computing Branch Displacements

- Compute sum of PC and scaled, sign-extended literal displacement
- Can’t share ALU, it might be needed for comparisons during branch operations



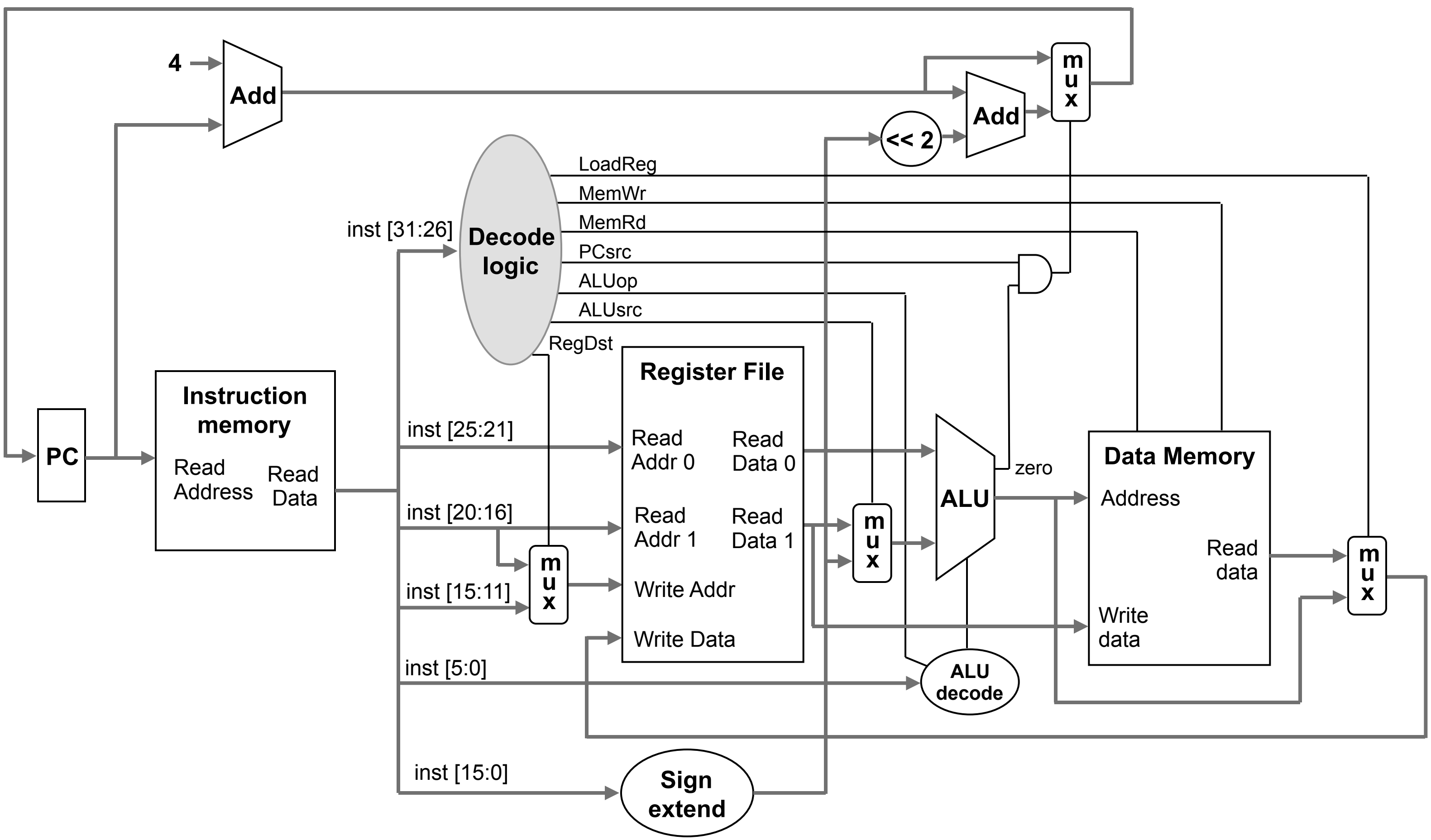
Accessing Memory – Loads & Stores

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

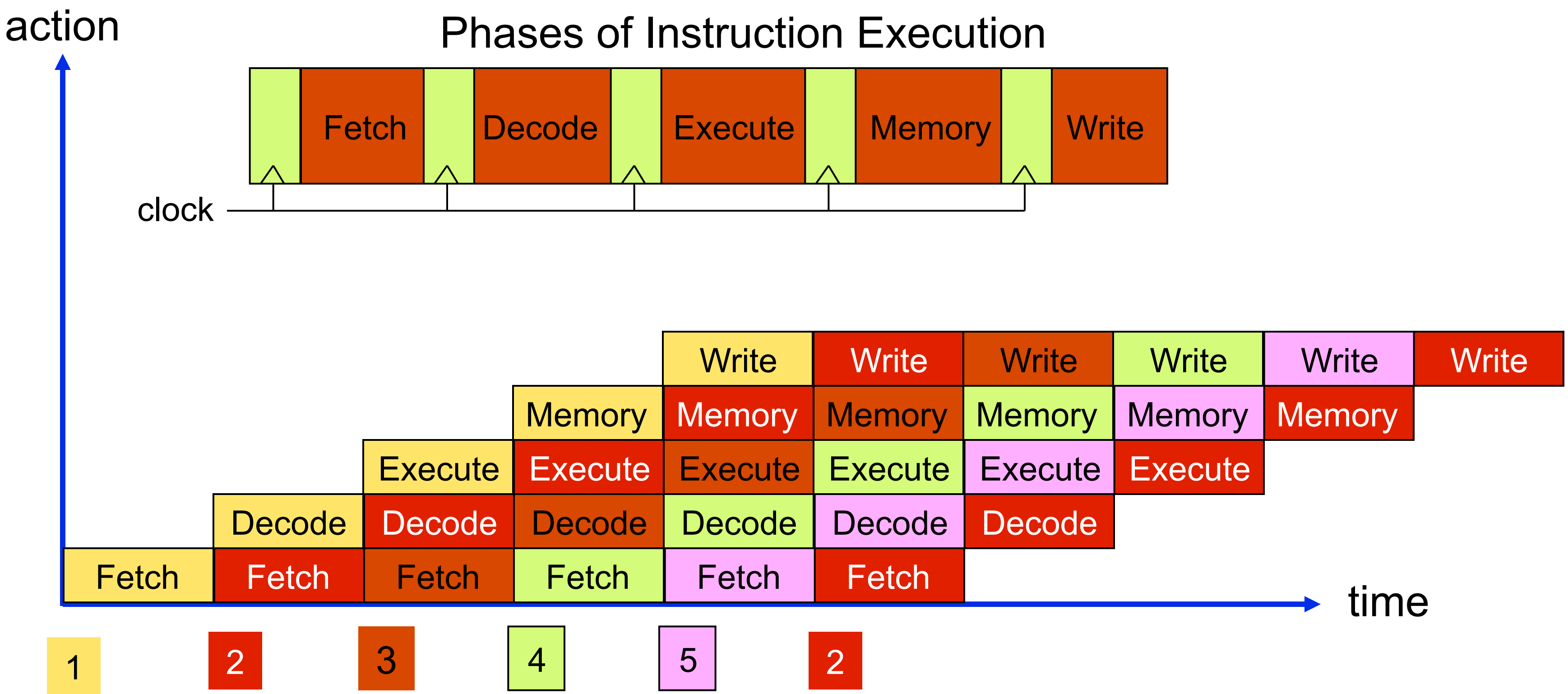


Decoding Instructions

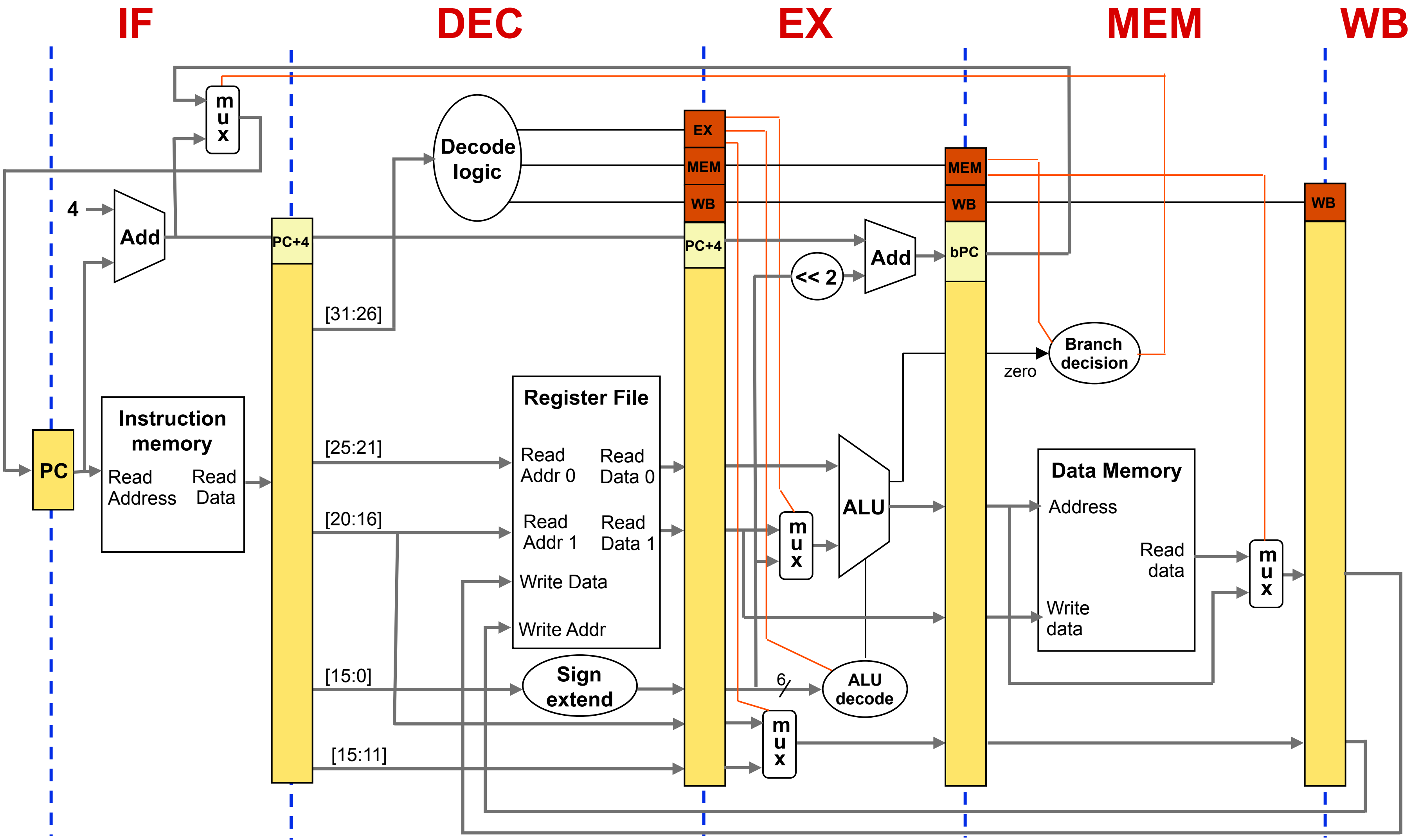
- Control signals driven by combinational logic, based on instruction opcode



Pipelined Instruction Execution

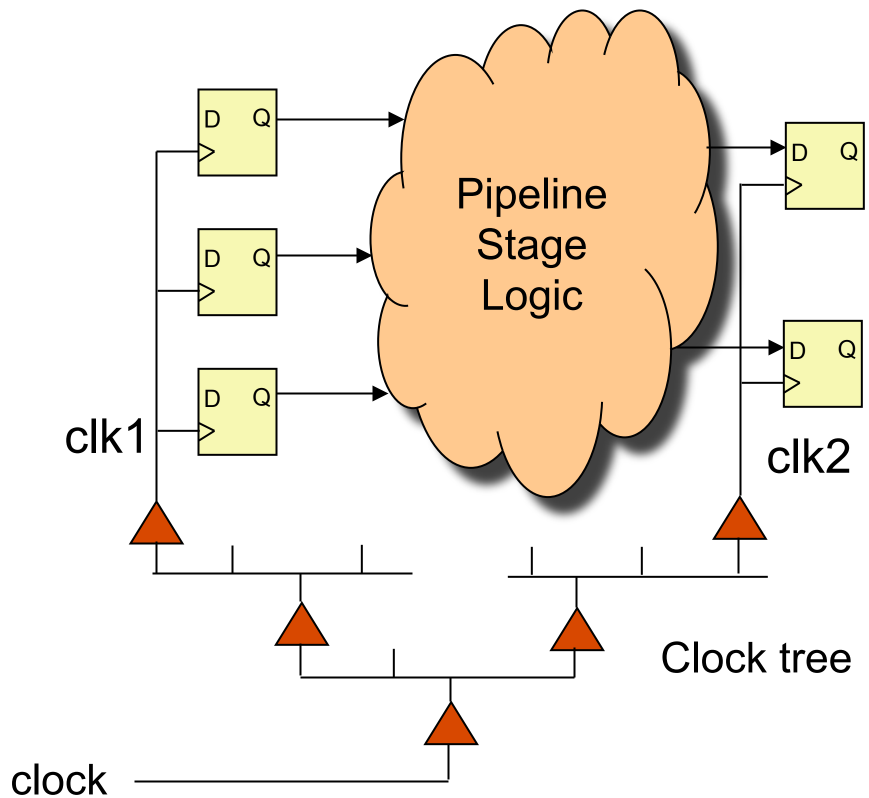


CPU Pipeline Structure



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

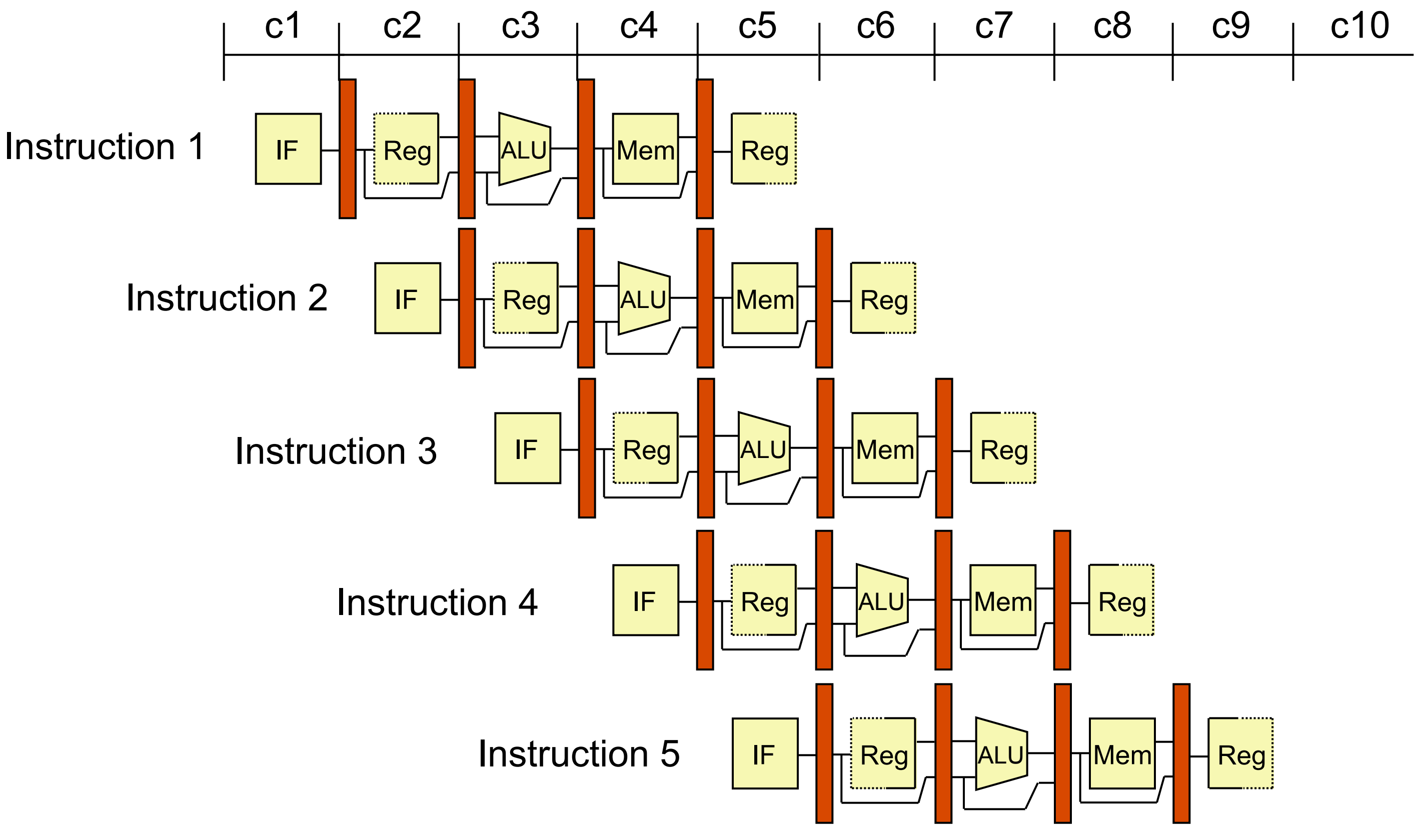


Ideally, all delays through every pipeline stage are identical

In practice this is hard to achieve

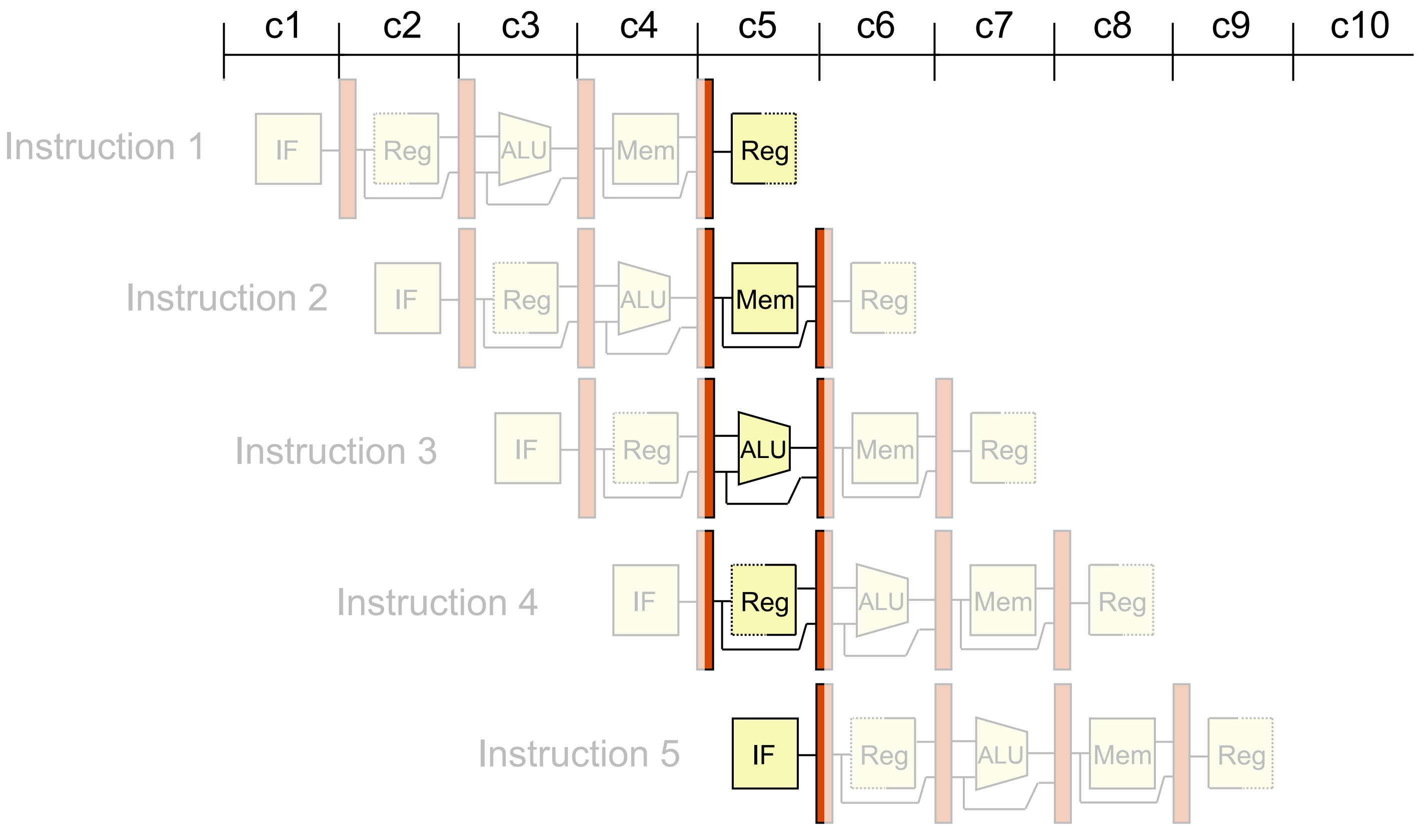
Representing a sequence of instructions

- 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
- This form is used in H&P, and throughout the remainder of these notes.

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

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

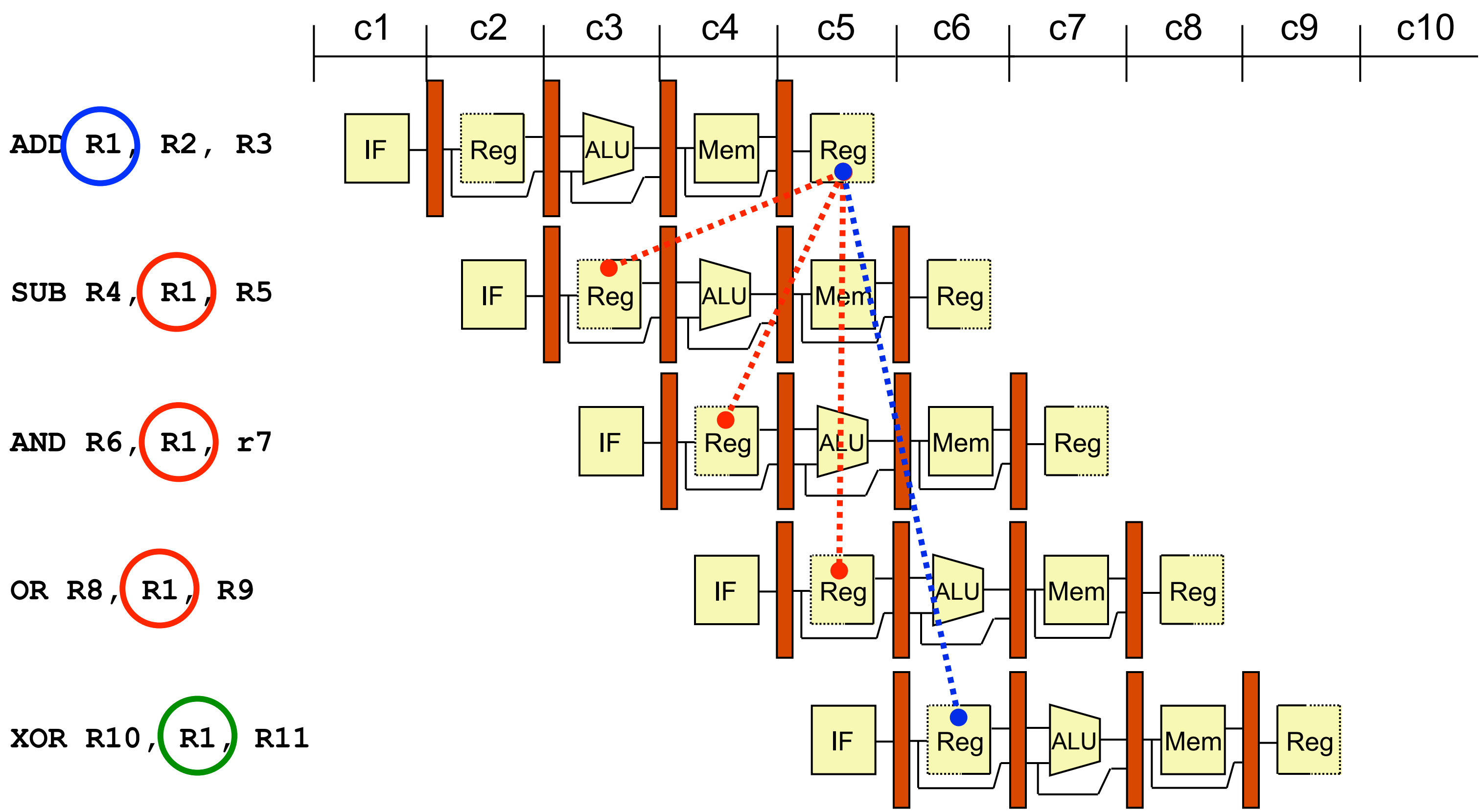
Instruction \ cycle	1	2	3	4	5	6	7	8	9	10
lw $1,($2)	IF	DEC	EX	MEM	WB					
instruction 2		IF	DEC	EX	MEM	WB				
instruction 3			IF	DEC	EX	MEM	WB			
instruction 4				IF	DEC	EX	MEM	WB		
instruction 5					IF	DEC	EX	MEM	WB	

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

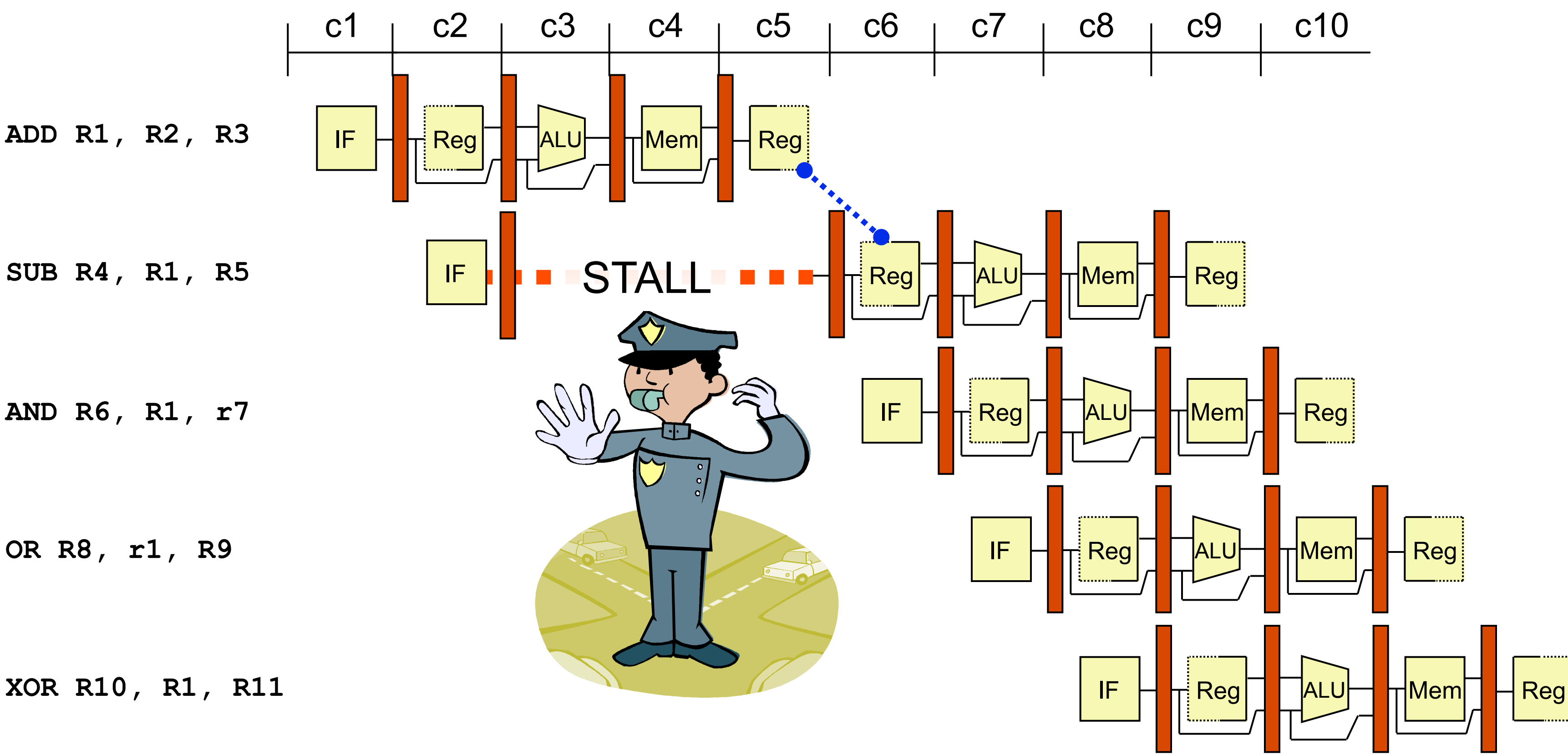
Data Hazards

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



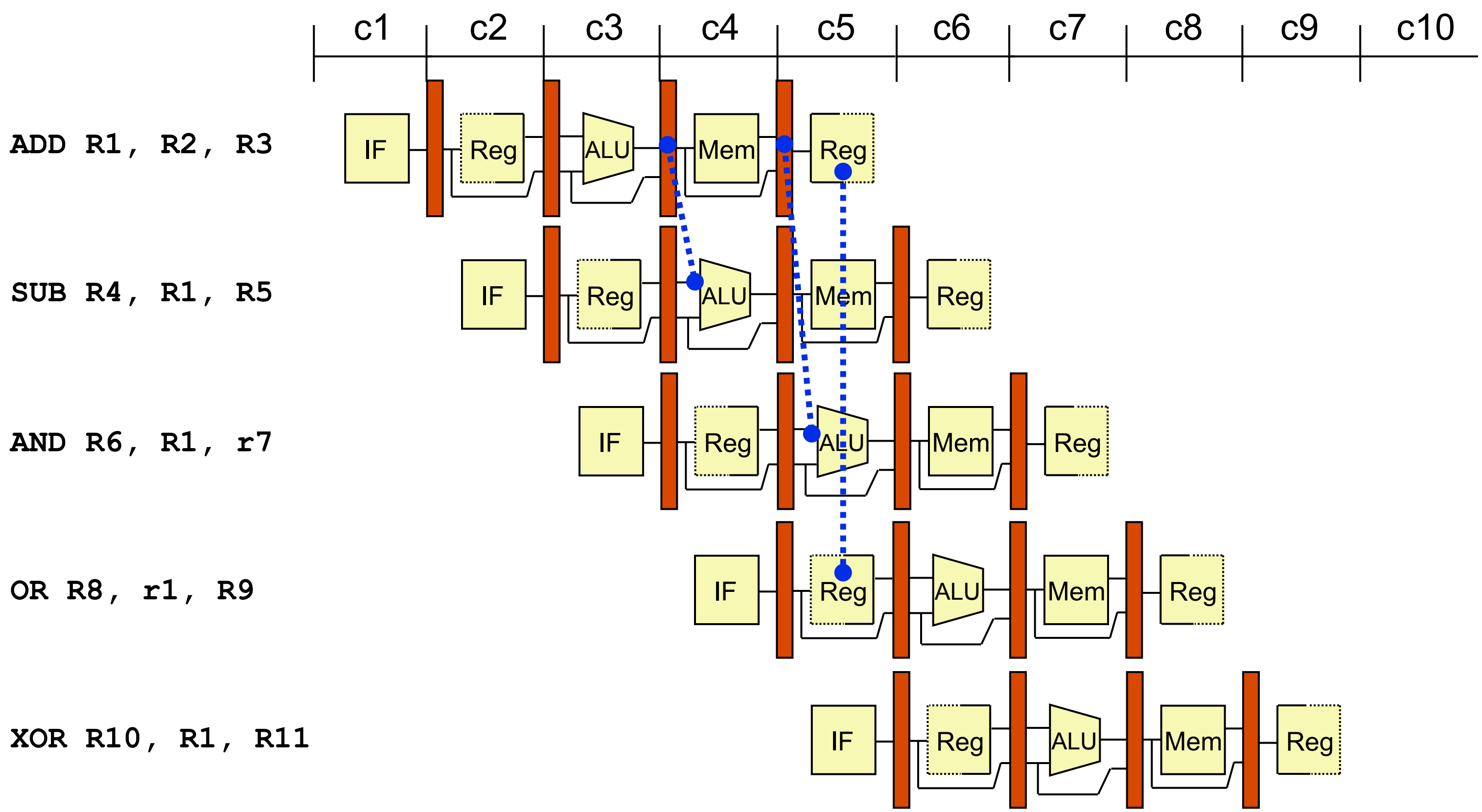
Data hazards lead to pipeline stalls

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

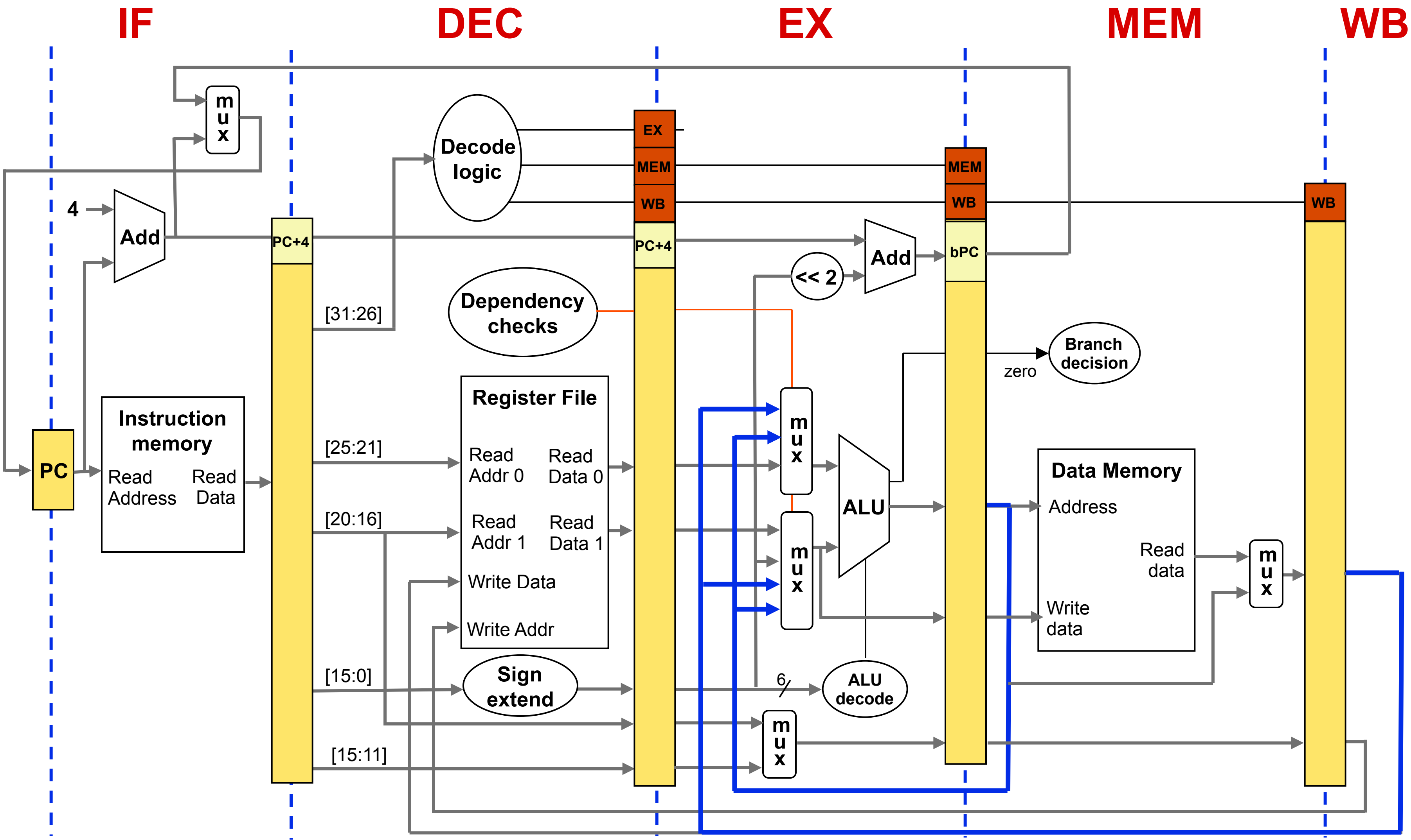


Minimising data hazards by data-forwarding

- 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.

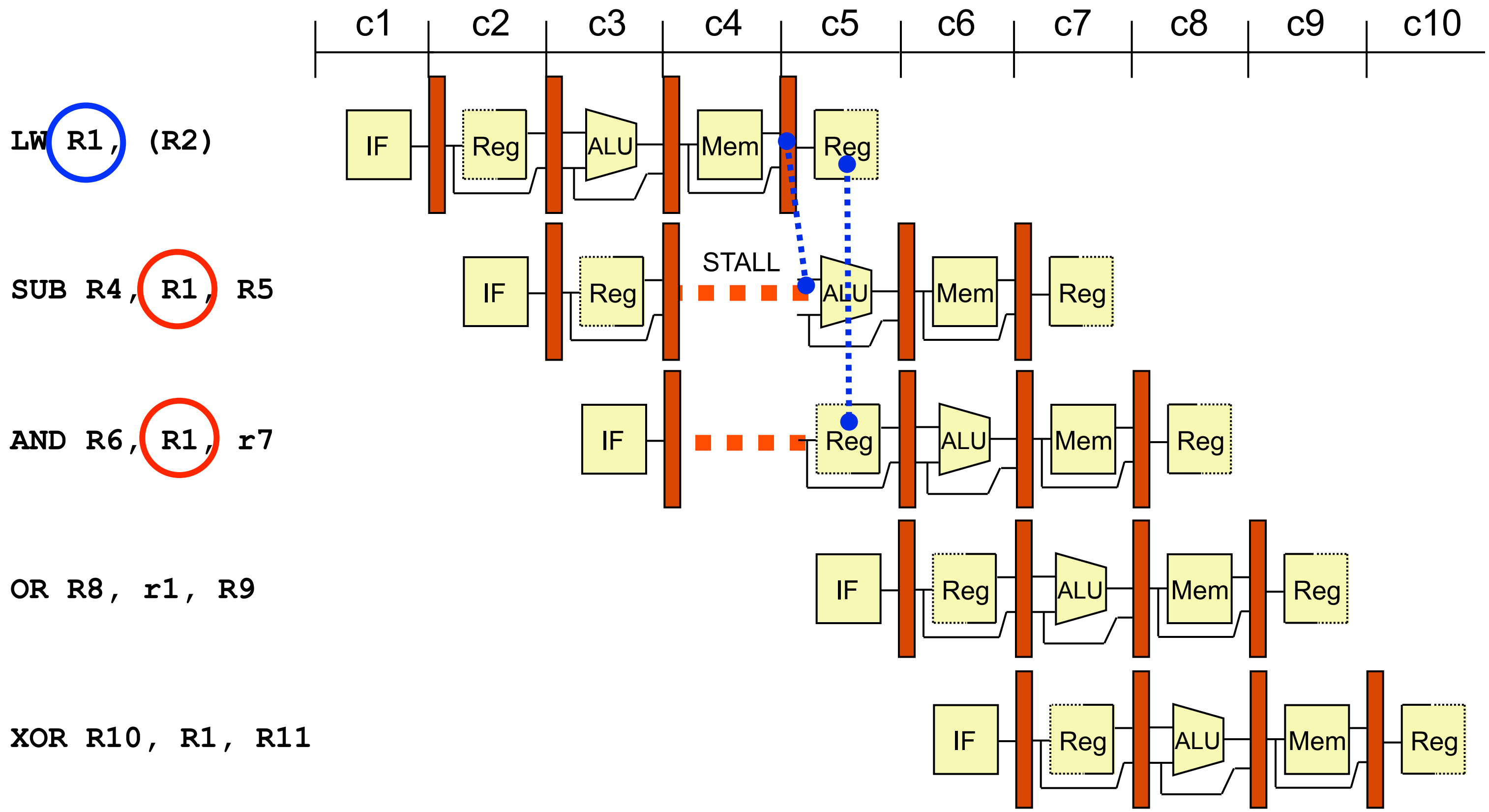


CPU pipeline showing forwarding paths



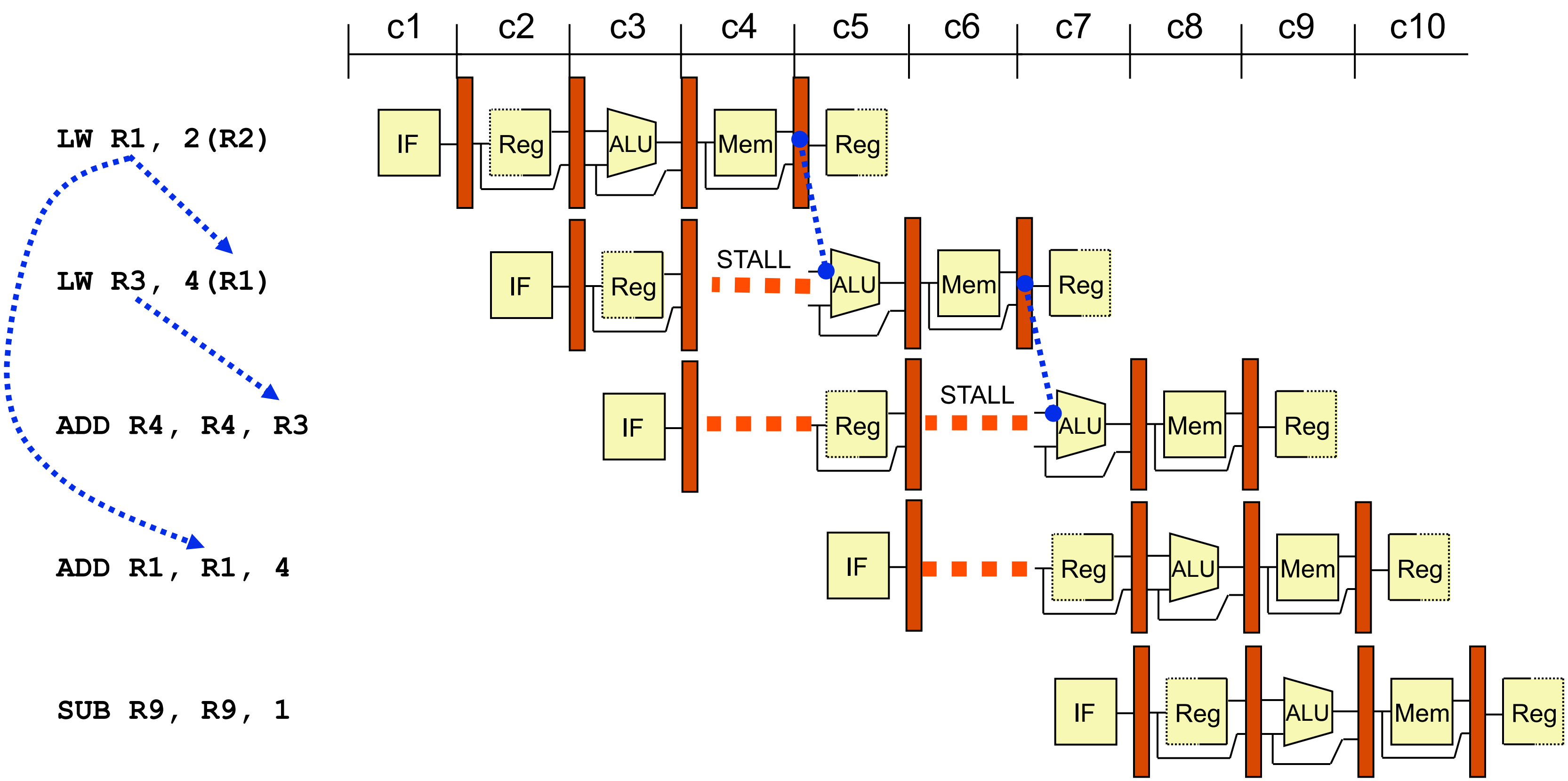
Data hazards requiring a stall

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



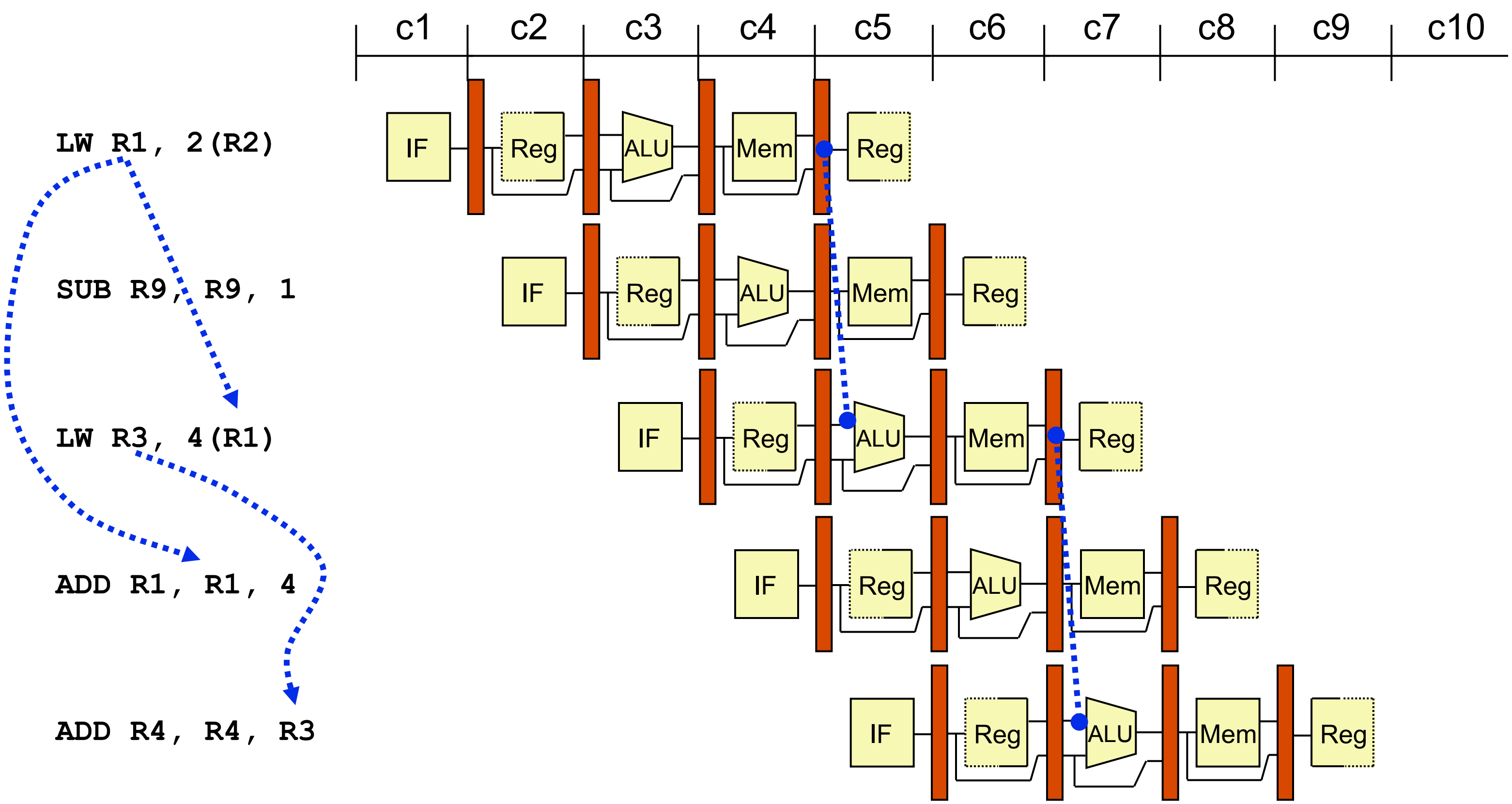
Code scheduling to avoid stalls (before)

- 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 1st load
- ADD to R1 can be placed after LW to R3 to hide the load delay on R3



General Performance Impact of Hazards

$$\text{Speedup from pipelining: } S = \frac{CPI_{unpipelined}}{CPI_{pipelined}} \times \frac{clock_{unpipelined}}{clock_{pipelined}}$$

$CPI_{pipelined}$ = ideal CPI + stall cycles per instruction = 1 + stall cycles per instruction

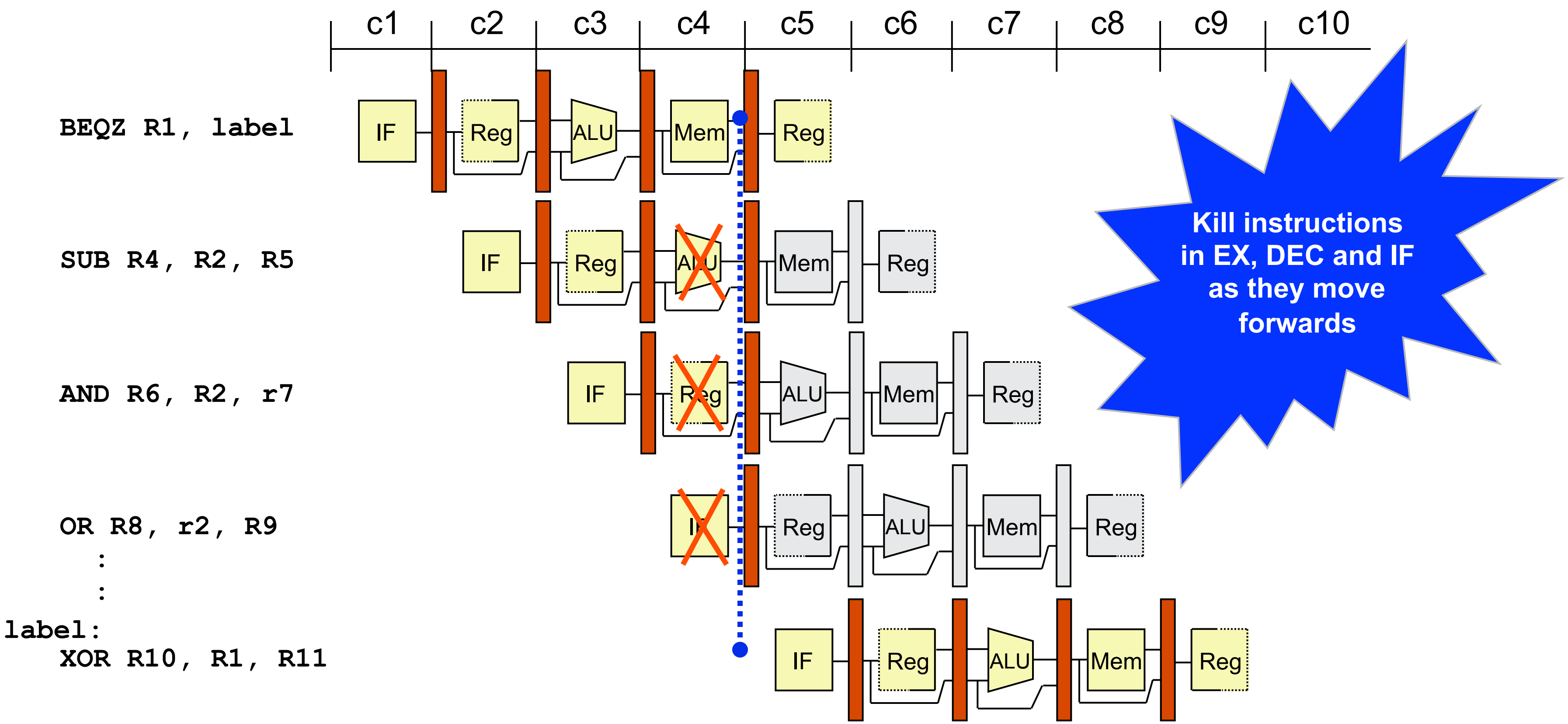
$CPI_{unpipelined}$ ~ pipeline depth

$$\frac{clock_{unpipelined}}{clock_{pipelined}} \sim 1$$

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

Control Hazards

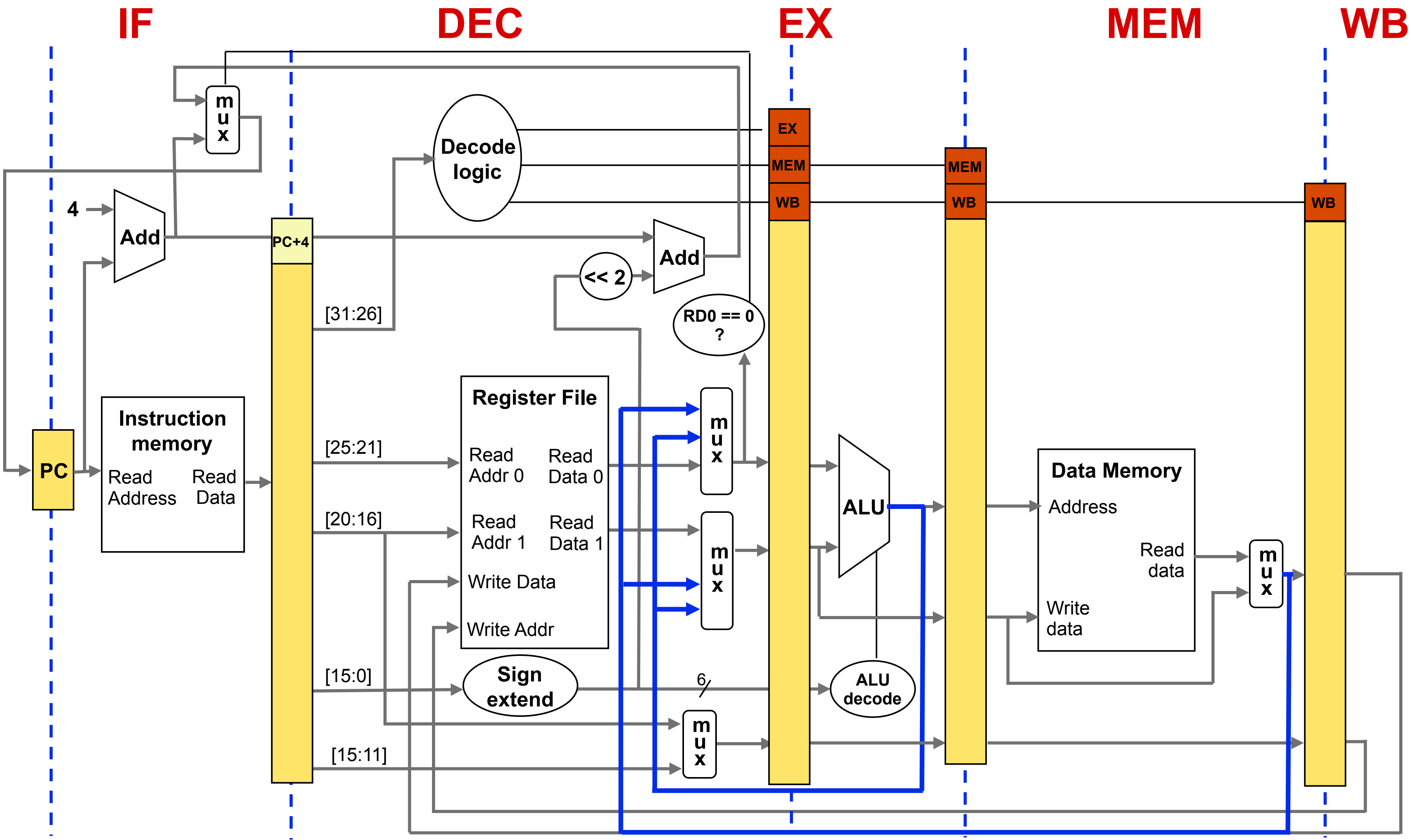
- When a branch is executed, PC is not affected until the branch instruction reaches the MEM stage.
- By this time 3 instructions have been fetched from the fall-through path.



Effect of branch penalty on CPI

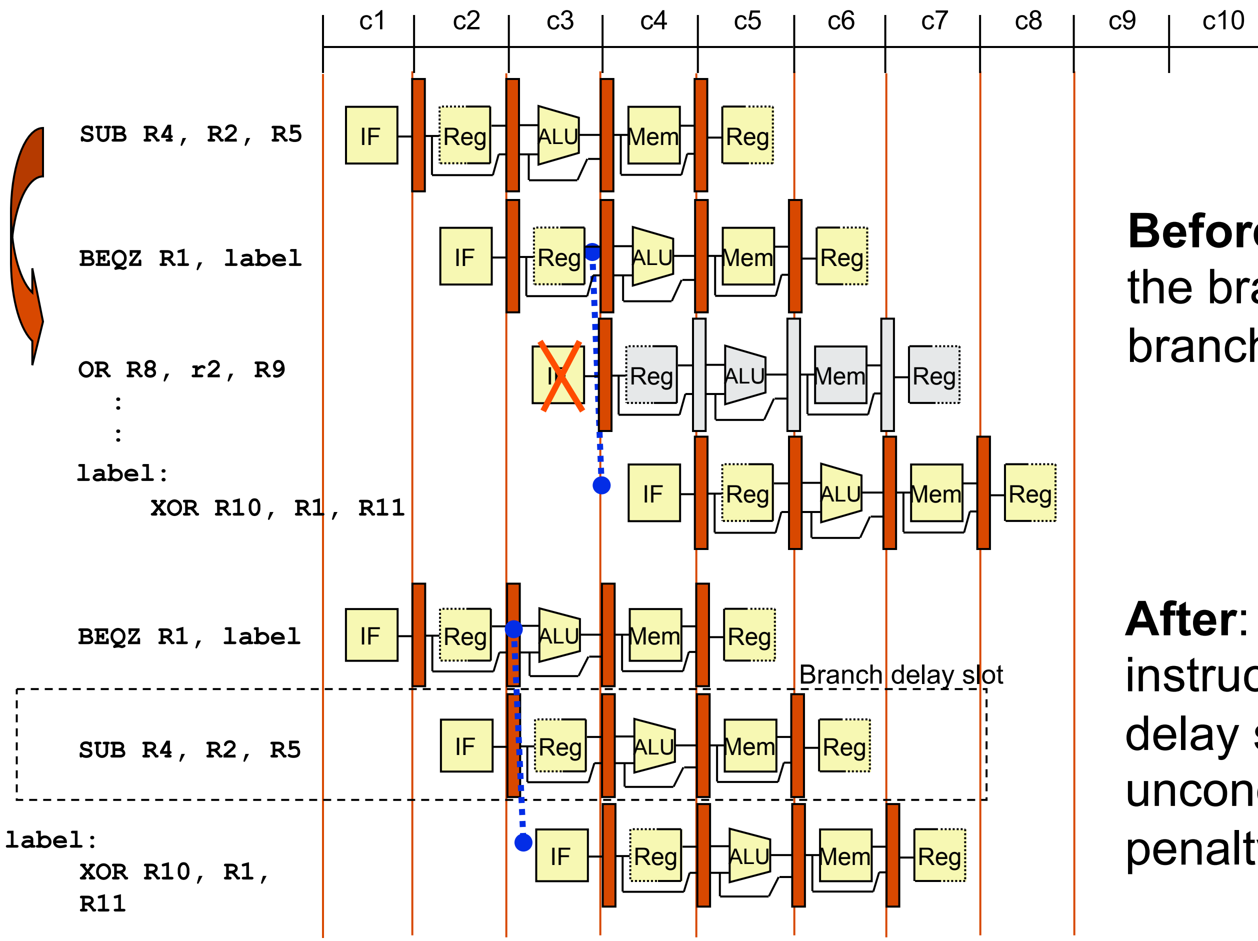
- In this example pipeline the cost of each branch is:
  - 1 cycle,   if the branch is not taken (due to load-delay slot)
  - 4 cycles, if the branch is taken
- If an equal number of branches are taken and not taken, and if 20% of all instructions are branches (a reasonable assumption), then
  - CPI = 0.8 + 0.2*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
  - CPI = 0.8 + 0.2*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
  - Branch prediction (static and dynamic)

Early branch outcome calculation - BEQZ, BNEZ



Delayed branch execution

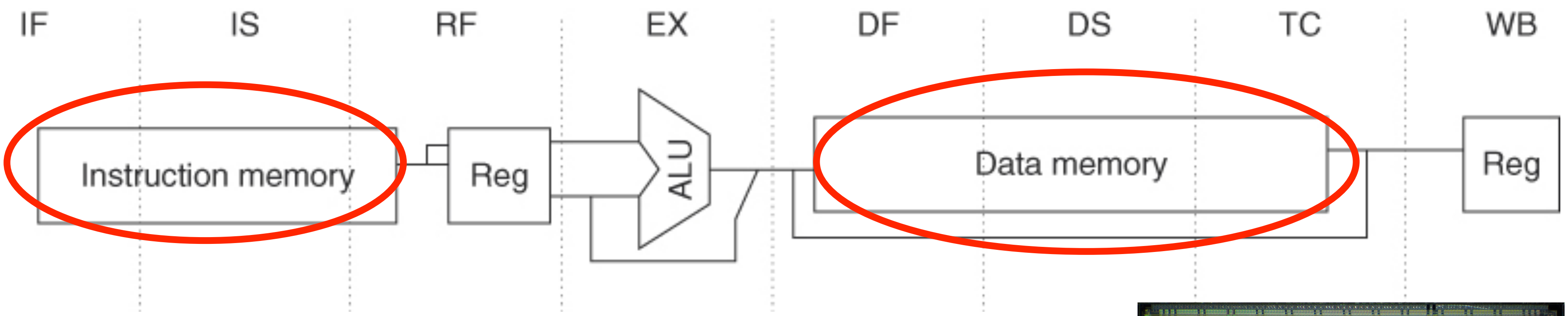
- Always execute the instruction immediately after the branch, regardless of branch outcome.

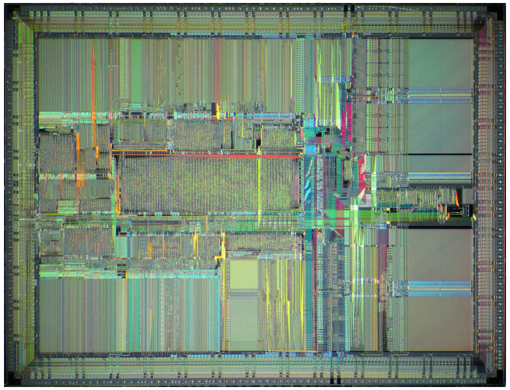


Before: instruction after the branch gets killed if the branch is taken

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

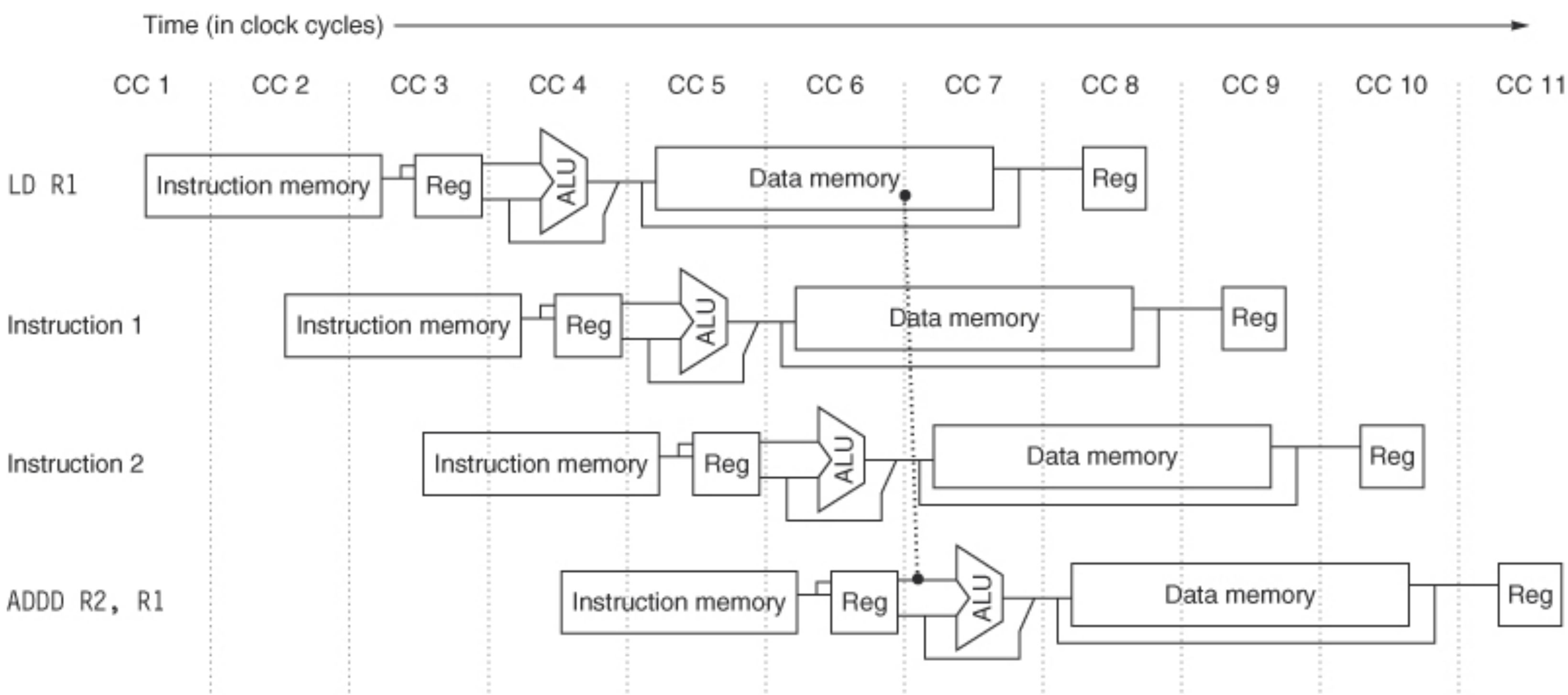
Case Study: Pipelining in MIPS R4000





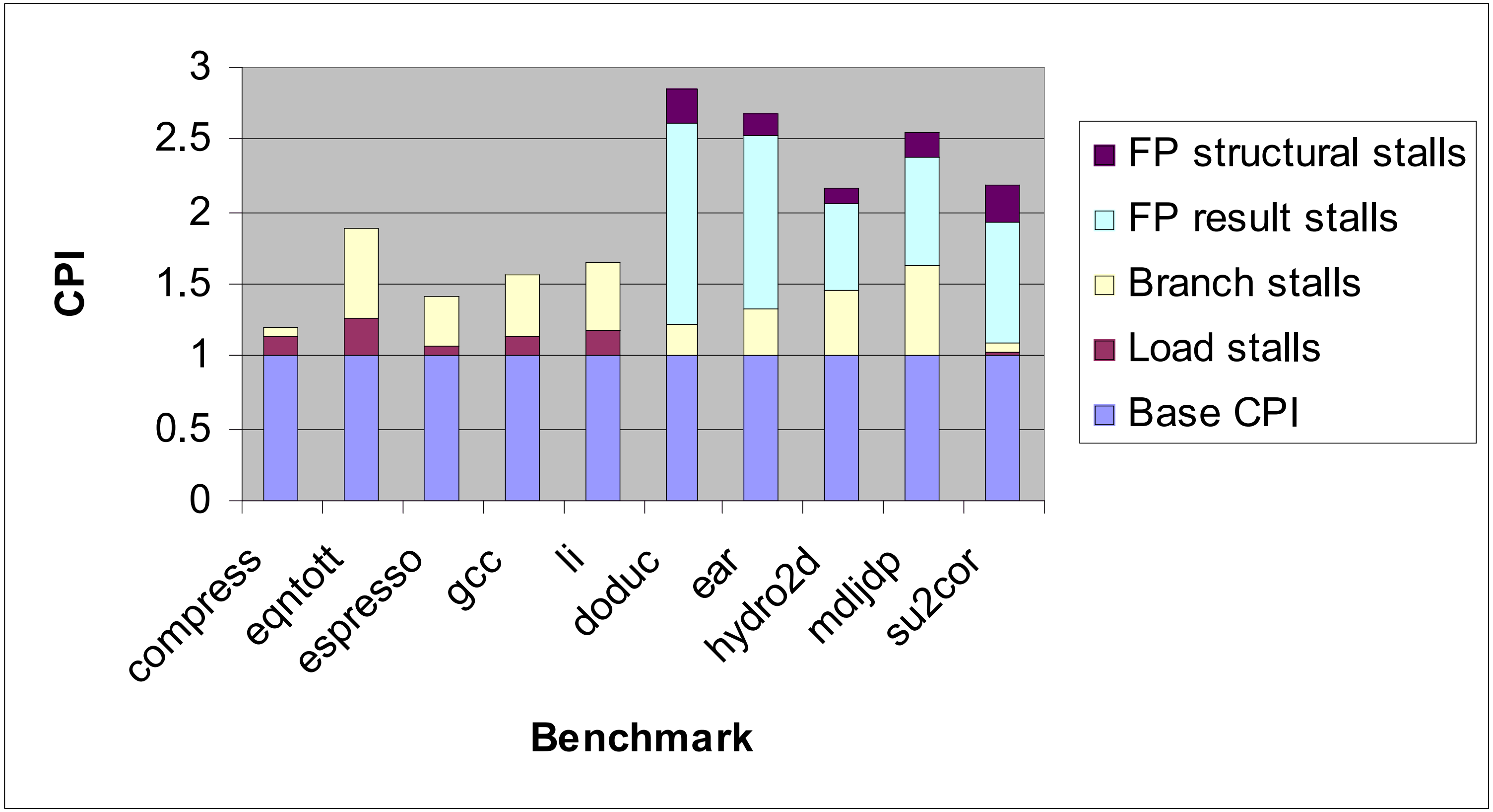
- Introduced in early 90s
  - 1.2 million transistors, 250 Mhz peak frequency
  - 64-bit CPU – one of the first!
- Notable feature: pipelined memory accesses

Load-to-use latency in the MIPS R4000



2-cycle load delay slot

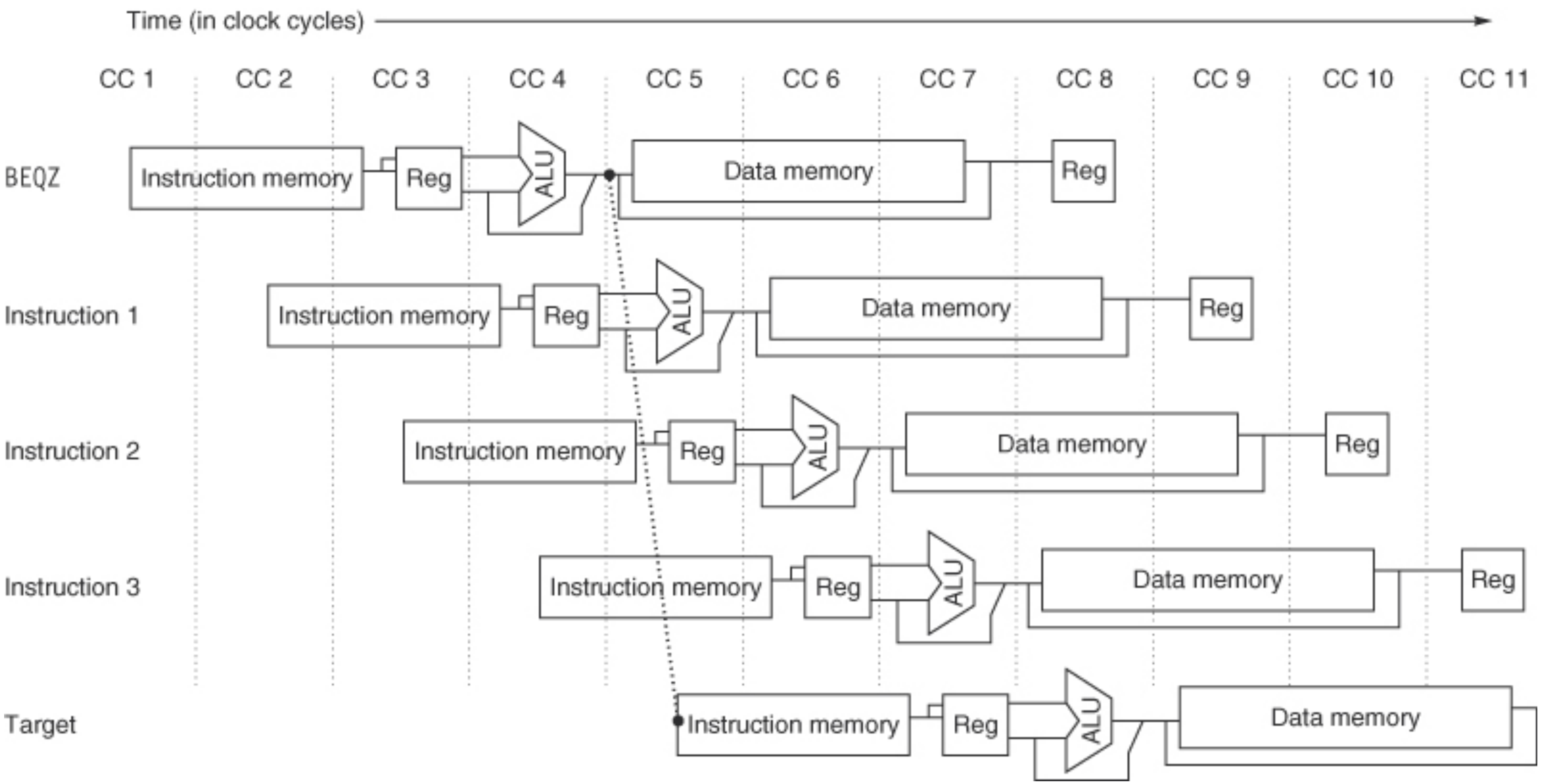
Impact of Empty Load-delay Slots on CPI



H&P 5/e Fig. C.52

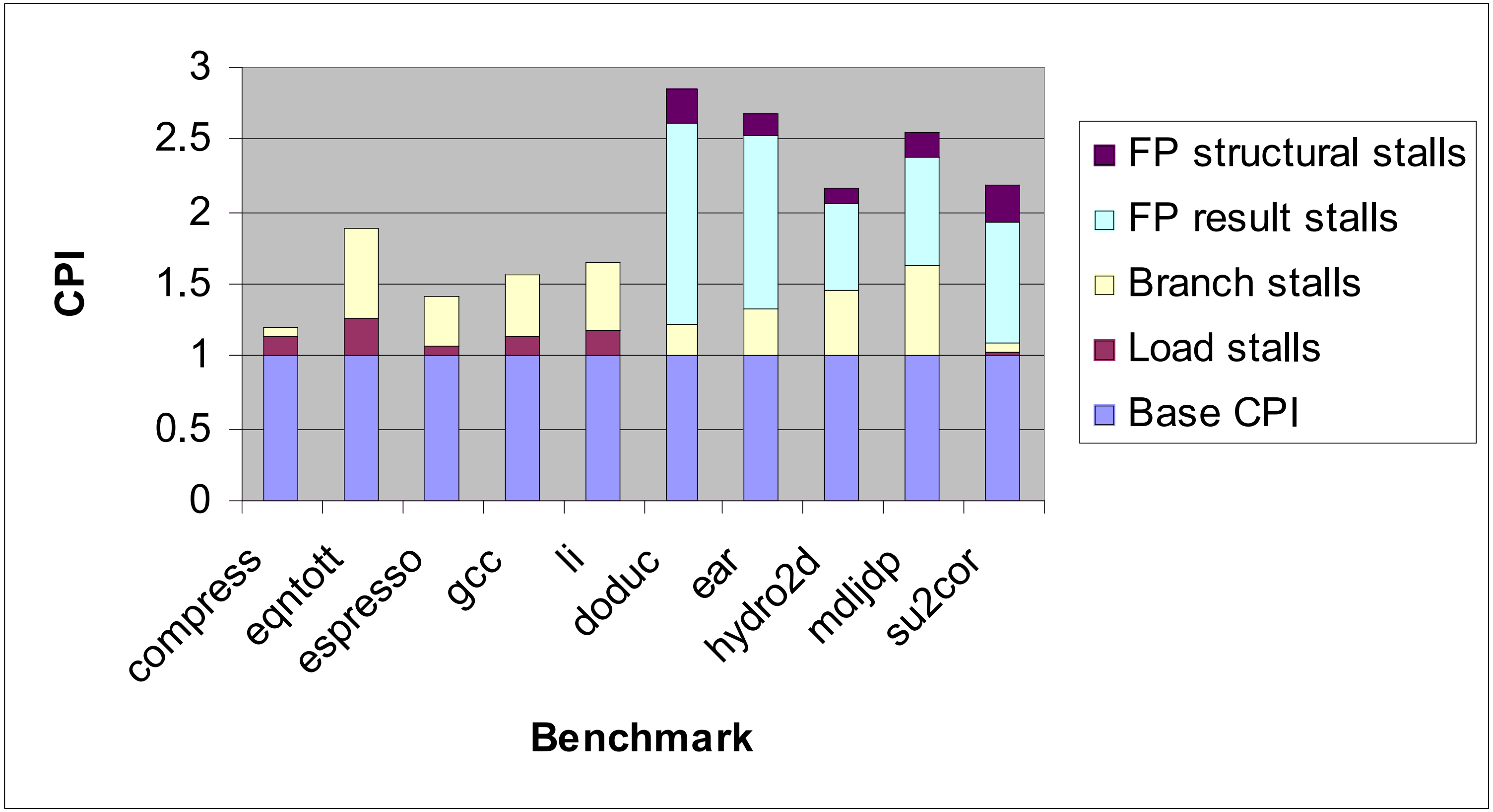
Bottom-line: CPI increase of 0.01 – 0.27 cycles

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