CSE-323
Computer Architecture

Instruction Set and Processor Organization

Lecturer Afia Anjum
Military Institute of Science and technology

Pipeline Strategy

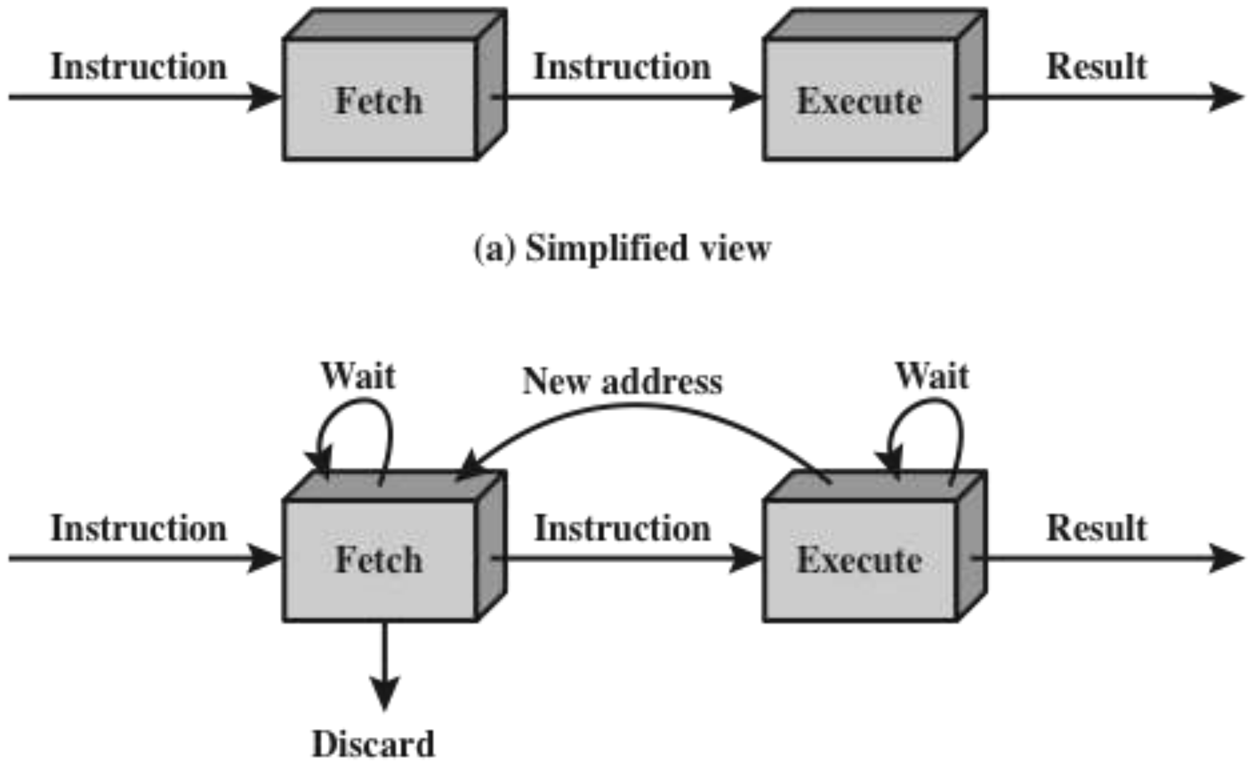
To improve the performance of a CPU we have two options:

1. Improve the hardware by introducing faster circuits.
2. Arrange the hardware such that more than one operation can be performed at the same time.

Since, the cost of faster circuits is quite high, we have to adopt the 2nd option.

- **Pipelining :** Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. Simultaneous execution of more than one instruction takes place in a pipelined processor.

Two-stage Instruction Pipeline



(b) Expanded view

Figure 14.9 Two-Stage Instruction Pipeline

Two-stage Instruction Pipeline

Disadvantages:

- The execution time will generally be longer than the fetch time. Thus the fetch stage may have to wait for some time before it can empty the buffer.

- When conditional branch occurs, then the address of next instruction to be fetched become unknown. Then the execution stage have to wait while the next instruction is fetched.
  - Simple Solution: Guessing can reduce the time loss.

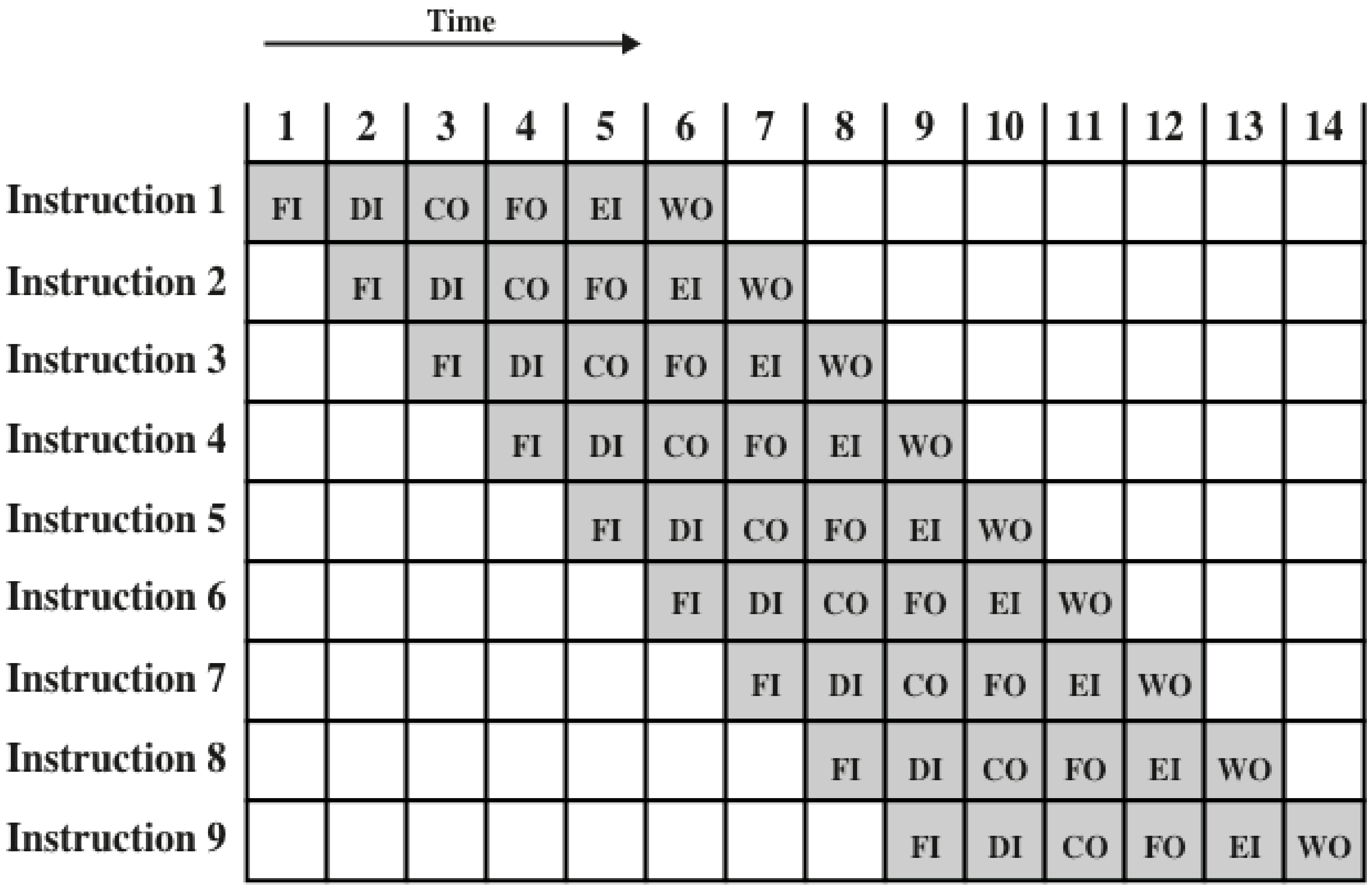
Decomposition of Instruction Stages

To gain further speedup, the pipeline can be divided into more stages(6 stages) :

- **Fetch instruction(FI):** Read the next expected instruction into a buffer.
- **Decode instruction(DI):** Determine the opcode and the operand specifiers.
- **Calculate operands(CO):** Calculate the effective address of each source operand.
- **Fetch operands(FO):** Fetch each operand from memory. Operands in registers need not be fetched.
- **Execute instructions(EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
- **Write operand(WO):** Store the result in memory.

With this decomposition, the various stages will be of more nearly equal duration.

Timing Diagram for Instruction Pipeline Operation



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

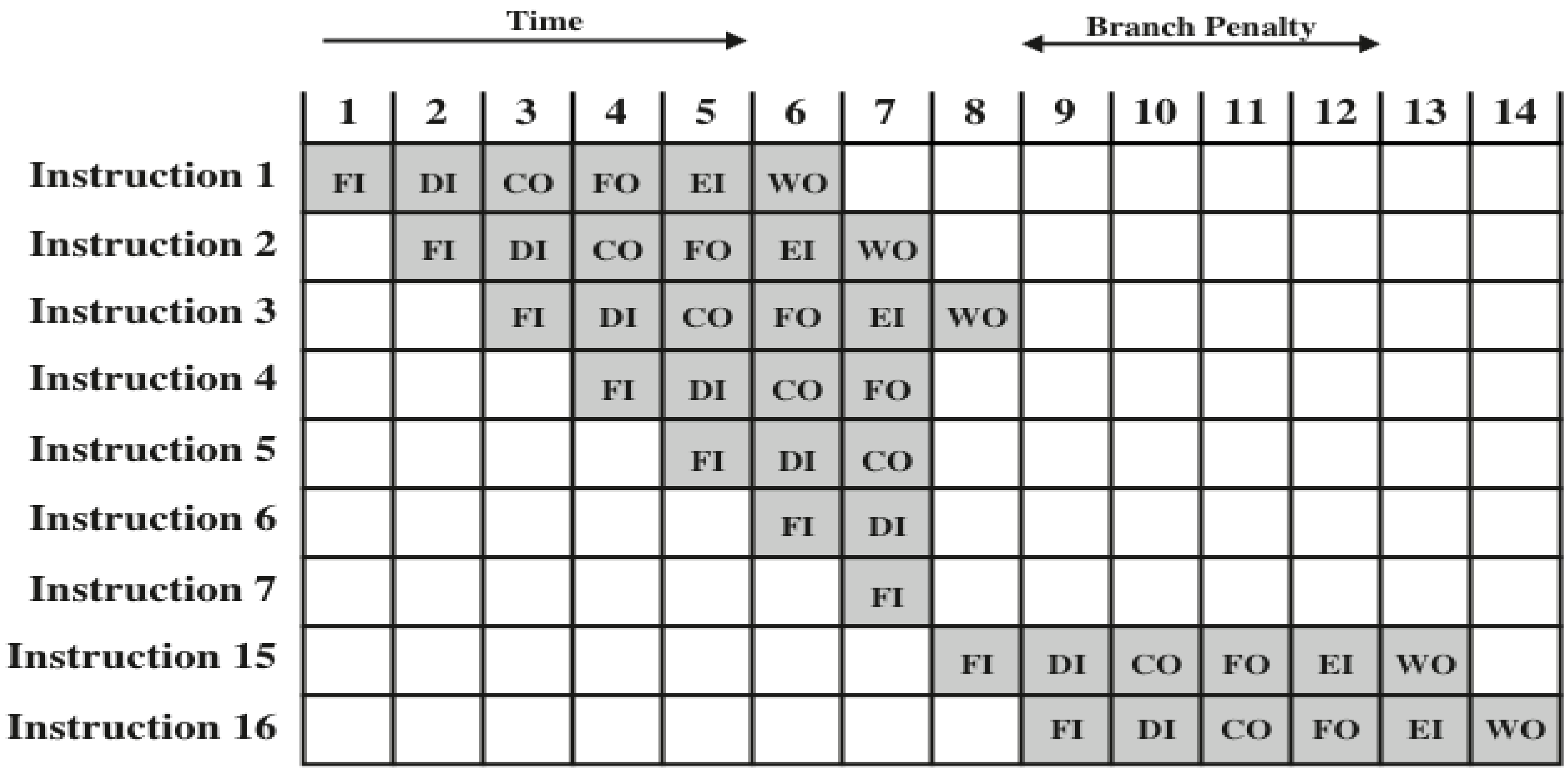
Figure 14.10 Timing Diagram for Instruction Pipeline Operation

Decomposition of Instruction Stages

Several other factors serve to limit the performance enhancement. For example:

- If the six stages are not of equal duration, there will be some waiting involved at various pipe- line stages, as discussed before for the two-stage pipeline.
- Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches.
- A similar unpredictable event is an interrupt.

The Effect of a Conditional Branch on Instruction Pipeline Operation


Time
Branch Penalty

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

The Effect of a Conditional Branch on Instruction Pipeline Operation

- Assume that the instruction 3 is a conditional branch to instruction 15.
- Until the instruction is executed there is no way of knowing which instruction will come next
- The pipeline will simply loads the next instruction in the sequence and execute
- Branch is not determined until the end of time unit 7
- During time unit 8,instruction 15 enters into the pipeline
- No instruction complete during time units 9 through 12
- This is the performance penalty incurred because of conditional branch

Six Stage Instruction Pipeline Flowchart

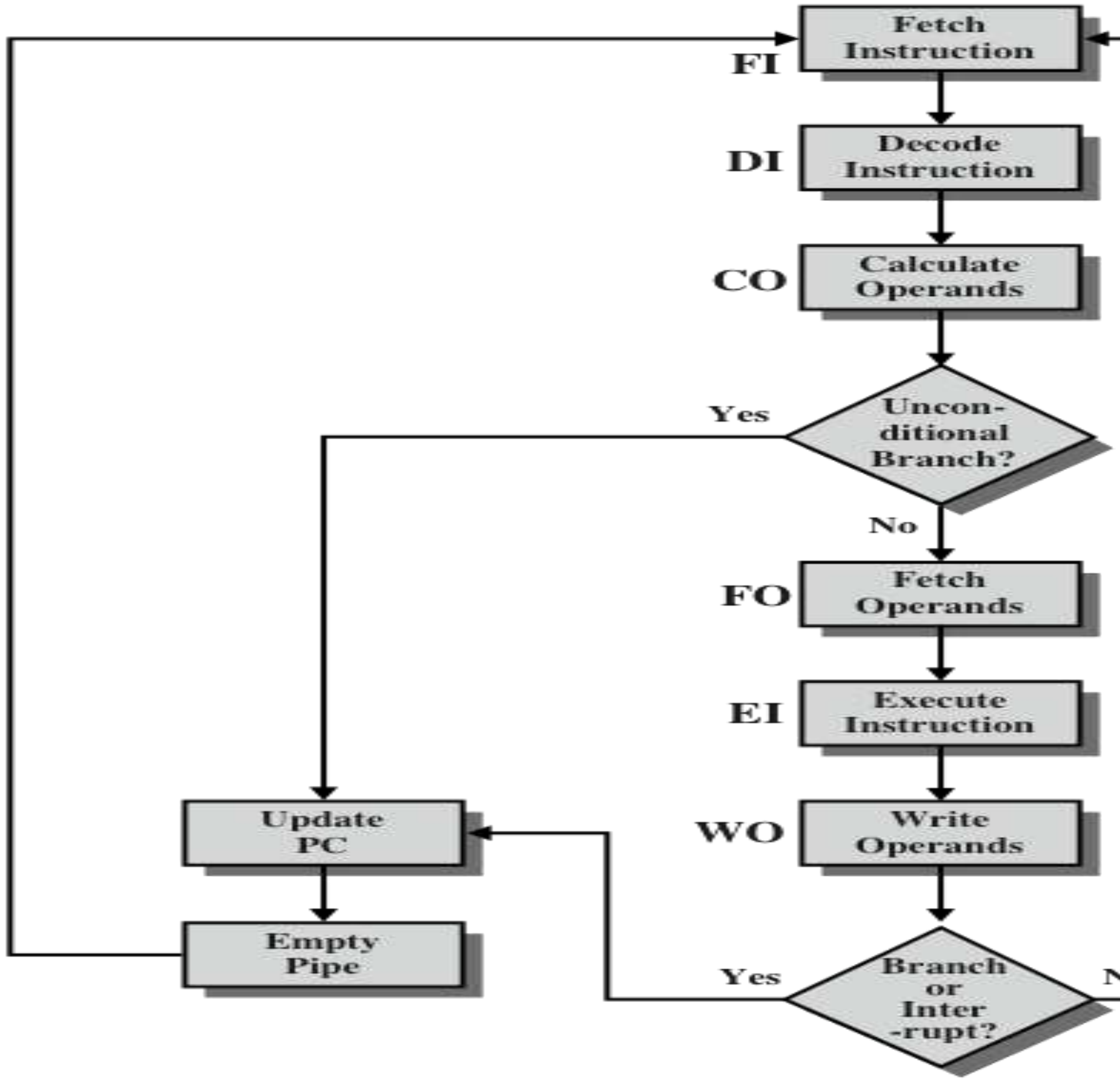
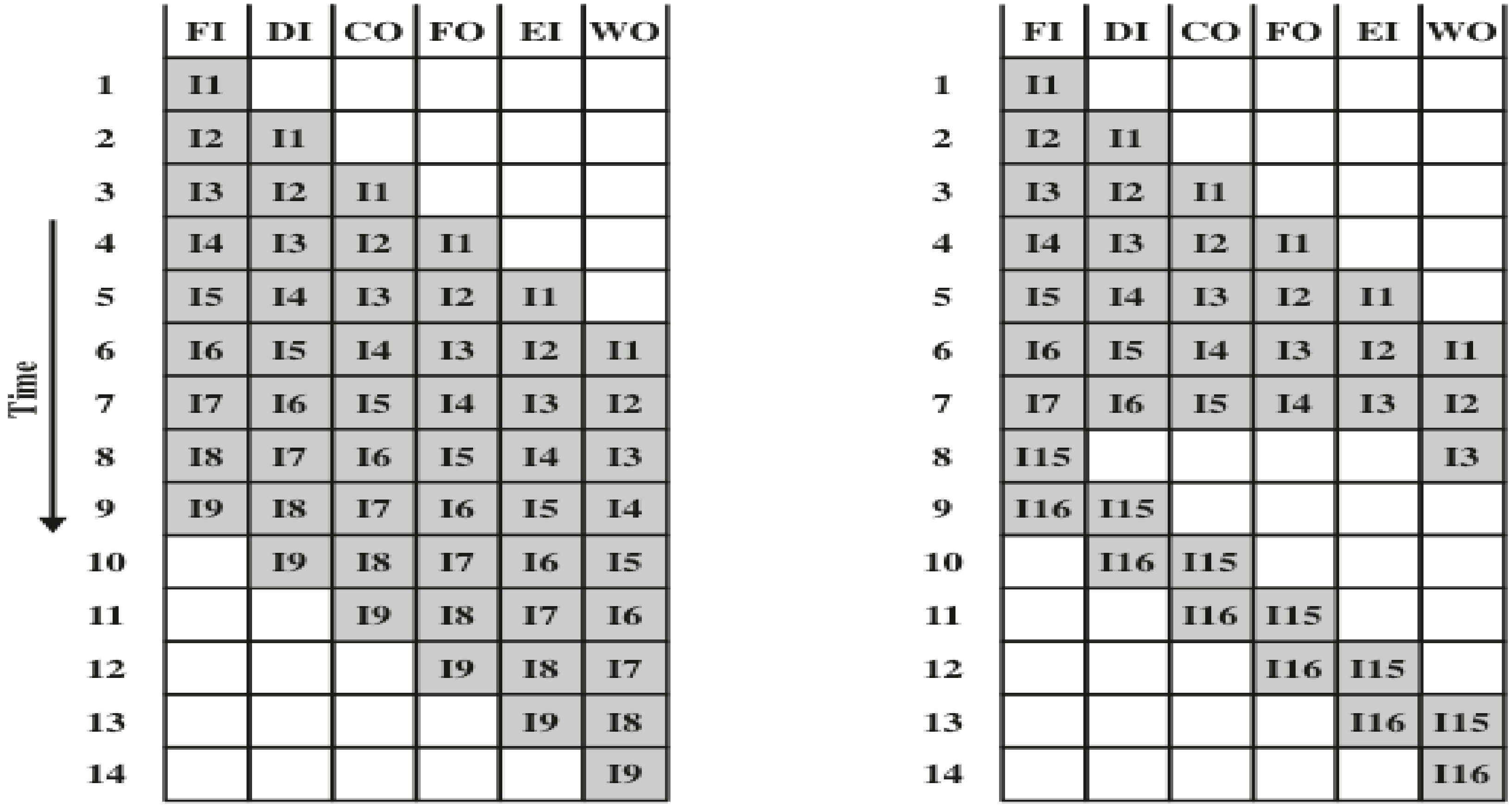


Figure 14.12 Six-Stage Instruction Pipeline

Alternate way to present a pipeline



	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I8	I7	I6	I5	I4	I3
9	I9	I8	I7	I6	I5	I4
10		I9	I8	I7	I6	I5
11			I9	I8	I7	I6
12				I9	I8	I7
13					I9	I8
14						I9

(a) No branches

	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I15					I3
9	I16	I15				
10		I16	I15			
11			I16	I15		
12				I16	I15	
13					I16	I15
14						I16

(b) With conditional branch

Figure 14.13 An Alternative Pipeline Depiction

Pipeline Hazard

- There are a lot of factors that halt the performance of a pipeline.
- In the previous slides, we saw some of them like conditional and unconditional branches, interrupts etc. In the later parts, we will examine this issue in a more systematic way
- A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution.
- Such a pipeline stall is also referred to as a **pipeline bubble**.
- There are three types of hazards:
  - Resource
  - Data
  - Control

Resource Hazard

- A resource hazard occurs when two or more instructions that are already in the pipeline need the same resource.
- The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline.
- A resource hazard is sometimes referred to as a **structural hazard.**

Resource Hazard - Example

Let us consider 5 stages of instruction pipeline:

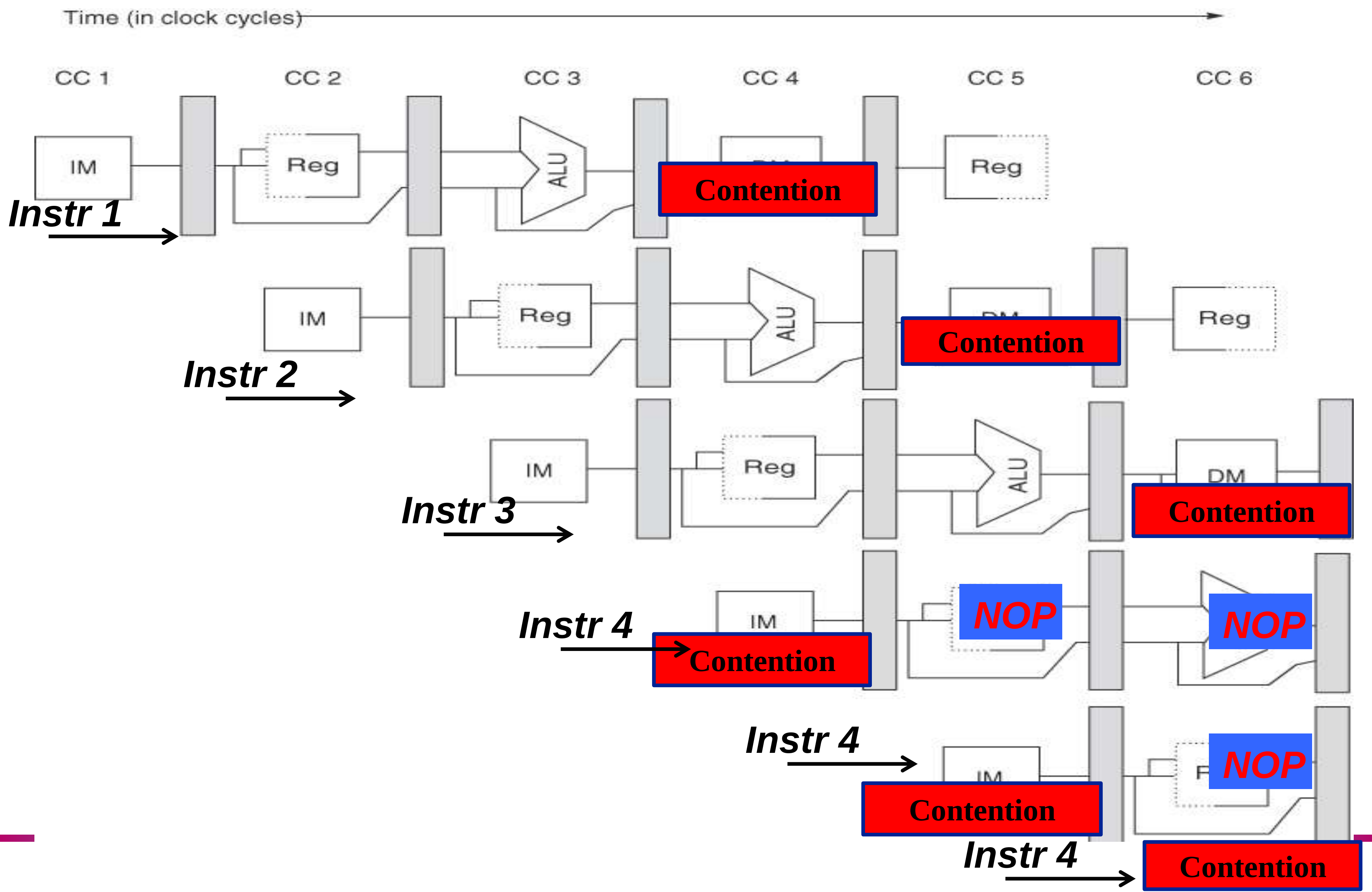
- IM :
  - Fetch instruction and pass to next stage
  - Increment the PC value to point at the next instruction
- Reg :
  - First half of cycle : Decode op code
  - Second half of cycle : Read operands from register file
- ALU :
  - ALU Computation
  - Calculate effective address for Load / Store instruction
- DM :
  - Data memory access for read / write
  - Used only in case of Load / Store instruction
- Reg :
  - Write to register file
  - Assumption being all writes to register files are done in 1st half of cycle.

Resource Hazard - Example

Now assume that main memory has a single port and that all instruction fetches and data read, write to memory must be performed using that port only. So these can not be performed in parallel.

- Contention for resource - In cc4 the **Instr 1**, is doing DM and **Instr 4**, is doing IM. So Instr 4 is put on hold.
- In cc5 when **Instr 4** is again attempting IM , there is again a contention as **Instr 2** is doing DM
- Similarly in cc6 also there will be a contention with Instr 3
- So three cycles would be wasted out of six cycles.
- In 6 cycles, 3 instructions got executed. Therefore IPC becomes 0.5

Resource Hazard - Example



Resource Hazard - Solution

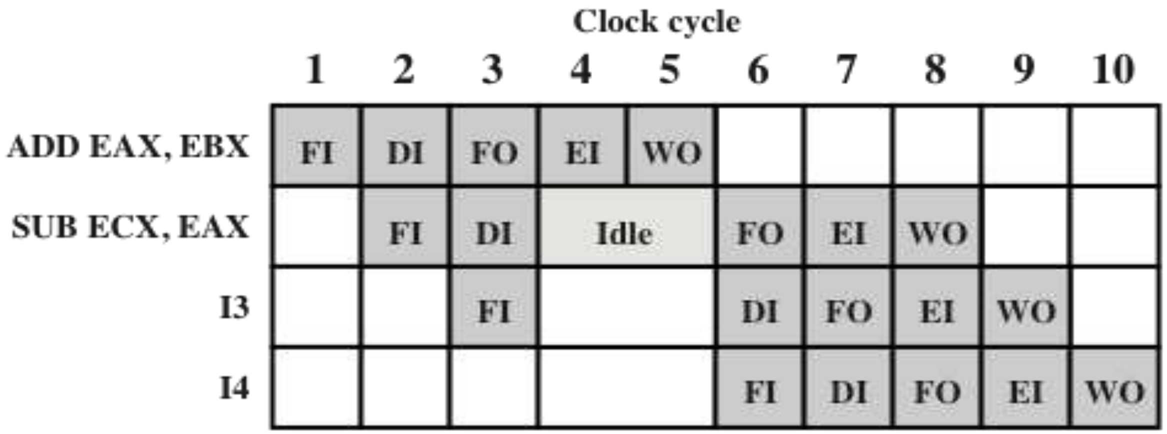
When contention occurs, later instructions and all the successors are delayed until a cycle is found when the resource is free

Solution:

- Eliminate contention by providing more resources
  - Separate memory space for instruction and data
  - Separate input output port

Data Hazard

- A data hazard occurs when there is a conflict in the access of an operand location
- In other words, a data hazard occurs when an instruction depends on result of prior instruction still in the pipeline



	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX	FI	DI	FO	EI	WO					
SUB ECX, EAX		FI	DI	Idle		FO	EI	WO		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

Figure 14.16 Example of Data Hazard

Data Hazard

Types of Data Hazard:

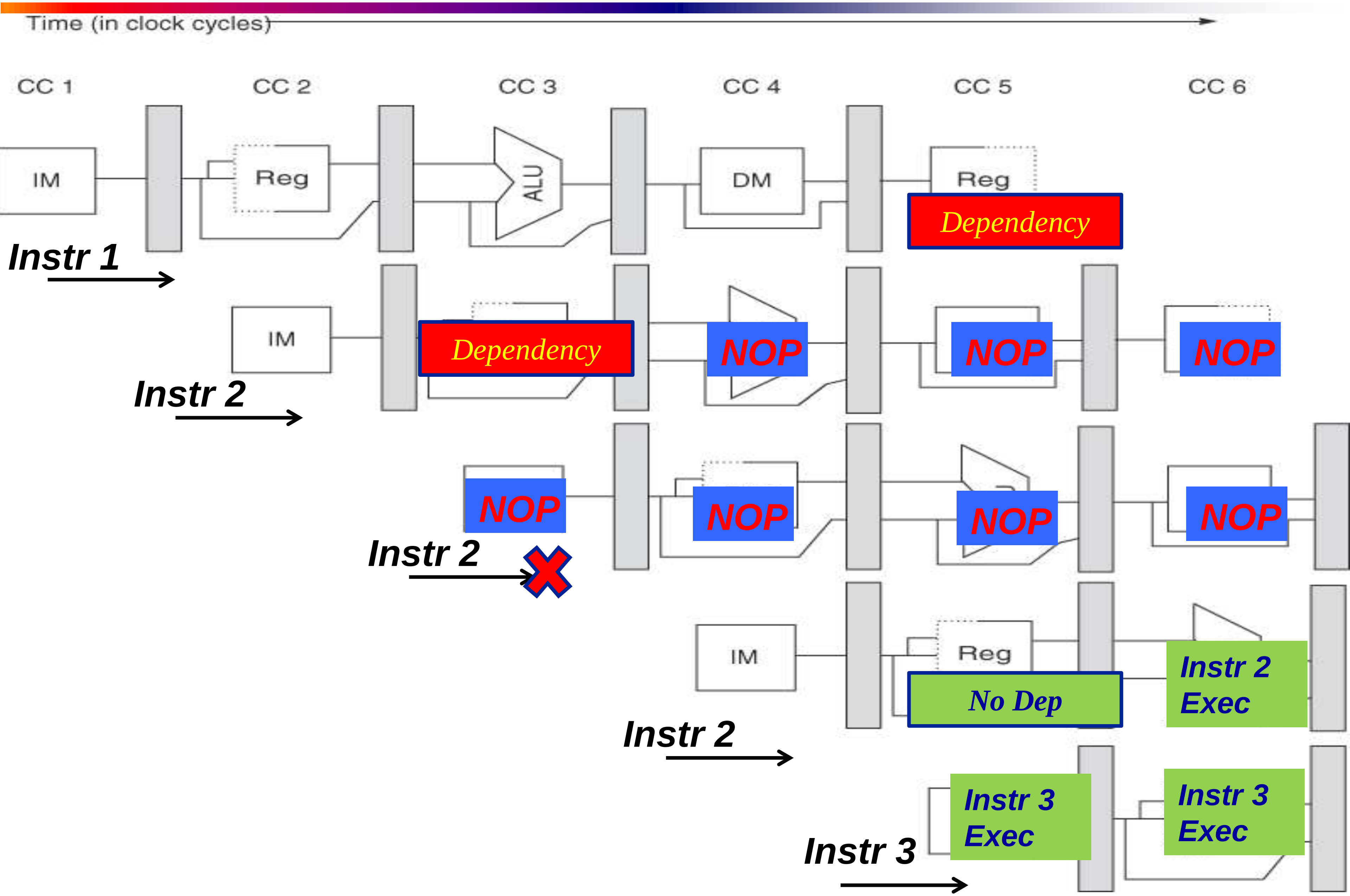
- Read after write (RAW), or true dependency
  - An instruction modifies a register or memory location
  - Succeeding instruction reads data in memory or register location
  - Hazard occurs if the read takes place before write operation is complete
- Write after read (WAR), or antidependency
  - An instruction reads a register or memory location
  - Succeeding instruction writes to the location
  - Hazard occurs if the write operation completes before the read operation takes place
- Write after write (WAW), or output dependency
  - Two instructions both write to the same location
  - Hazard occurs if the write operations take place in the reverse order of the intended sequence

Data Hazard - Example

Assume following sequence of instructions in a pipeline:-

- Instr 1 : Add r1,r2 -> r3
- Instr 2 : Add r3,r4 -> r5

Data Hazard - Example



Data Hazard - Solution

- **Software / Compiler :** Execute some other instructions in cc2 and cc3
  - Detect the problem at the time of compiling the program.
  - Schedule another instruction which is independent of the outcome of Instr 1, to execute in the next clock cycles after Instr 1 till the value of r3 is available in cc5.

- **Hardware Solution :**
  - Bypassing / Forwarding

Data Hazard – Solution- Bypassing

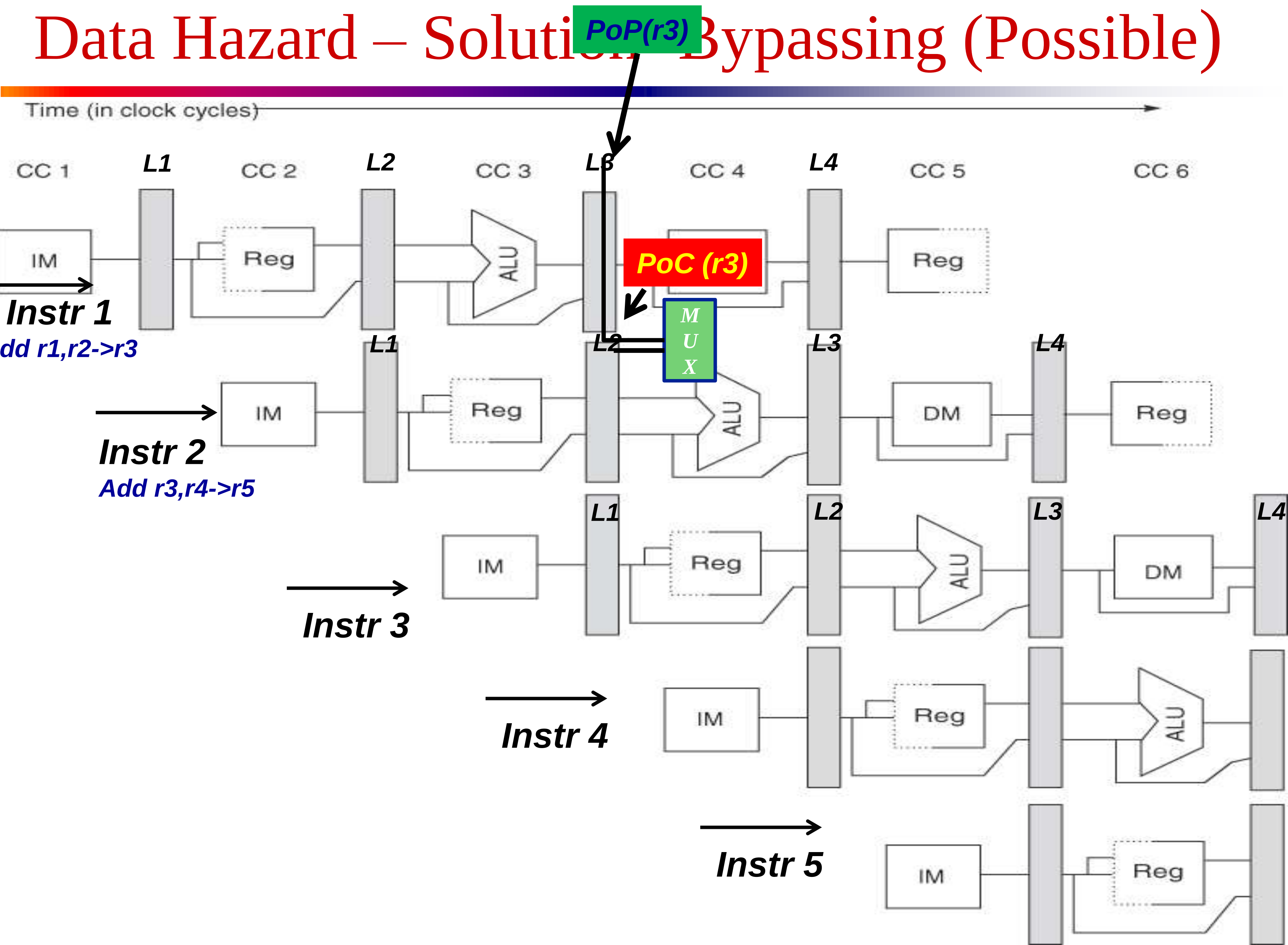
- Sequence of instrs in the pipeline :-

  Instr 1 : Add r1,r2 -> r3

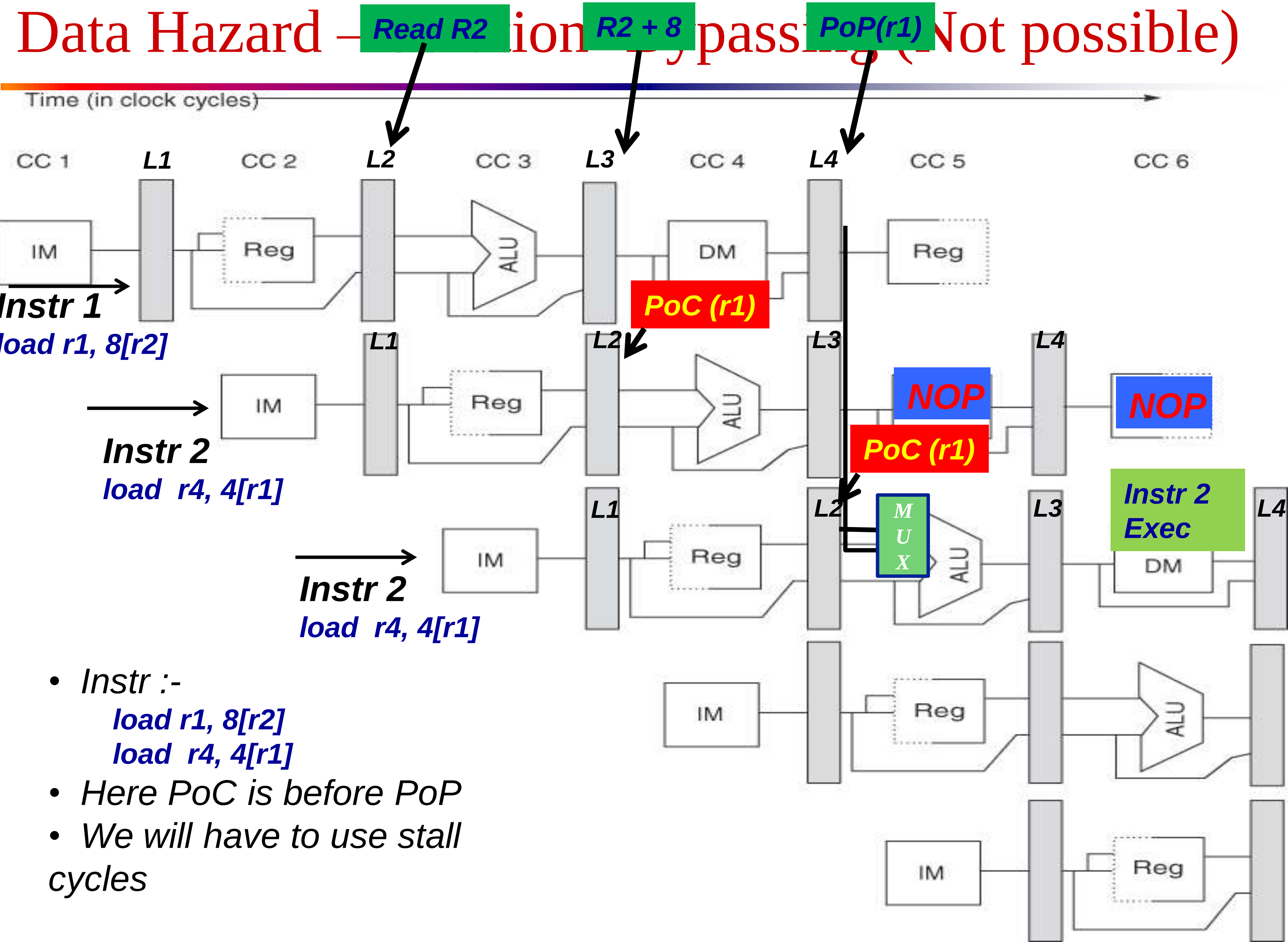
  Instr 2 : Add r3,r4 -> r5

- **General Rules:**
  - Data values are computed in ALU stage but "publicized" or written in Reg stage. So we should use it from the moment it has been computed!
  - To do so, we need to identify "Point of Production (PoP)" and "Point of Consumption (PoC)"
  - If PoP is after PoC then nothing can be done and stall cycles will be needed
  - Otherwise it is possible to use bypassing / forwarding and issue the instructions back to back

Data Hazard – Solution Bypassing (Possible)



Data Hazard – [illegible]tion By[illegible]passing (Not possible)



- *Instr :-*
  - ***load r1, 8[r2]***
  - ***load r4, 4[r1]***
- *Here PoC is before PoP*
- *We will have to use stall cycles*

Control Hazard

- Control Hazard, also known as branch hazard, occurs when we don’t know what the next instruction is.
- Mostly caused by branches.
- When the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded.
- Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.
- Dealing with Branches to avoid control hazard:
  - Multiple streams
  - Prefetch branch target
  - Loop buffer
  - Branch prediction

Solution: Multiple Streams

- Have 2 pipelines
- Prefetch each branch into a separate pipeline
- Use the appropriate pipeline

Drawbacks:

- With multiple pipelines there are contention delays for access to the registers and to memory.
- Multiple branches lead to further pipelines being needed

Solution: Prefetch Branch Target

- When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch
- Target is then saved until the branch instruction is executed
- If the branch is taken, the target has already been prefetched
- IBM 360/91 uses this approach

Solution: Loop Buffer

- A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the *n* most recently fetched instructions, in sequence.
- If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer.
- This strategy is particularly well suited to dealing with loops, or iterations; hence the name *loop buffer*. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

Solution: Branch Prediction

Various techniques can be used to predict whether a branch will be taken:

- Static approach:
  - **Predict never taken:** Always fetch next instruction
  - **Predict always taken:** Always fetch target instruction
  - **Predict by opcode**: The processor assumes that the branch will be taken for certain branch opcodes and not for others. The success rates is sometimes greater than 75% with this strategy.
- Dynamic approach:
  - **Taken/not taken switch:** One or more bits can be associated with each conditional branch instruction that reflect the recent history of the instruction. These bits are referred to as a taken/ not taken switch that directs the processor to make a particular decision the next time the instruction is encountered. A cache can be maintained
  - **Branch history table:** Another possibility is to maintain a small table for recently executed branch instructions with one or more history bits in each entry. The processor could access the table associatively, like a cache, or by using the low-order bits of the branch instruction’s address.

"If you're always trying to be normal,
You'll never know how amazing you could be!"

-Maya Angelou