



# Advanced Computer Architecture

## Instruction Pipelining

Instructor

**Dr. Dinh-Duc Anh-Vu**

<http://www.cse.hcmut.edu.vn/~anhvu>

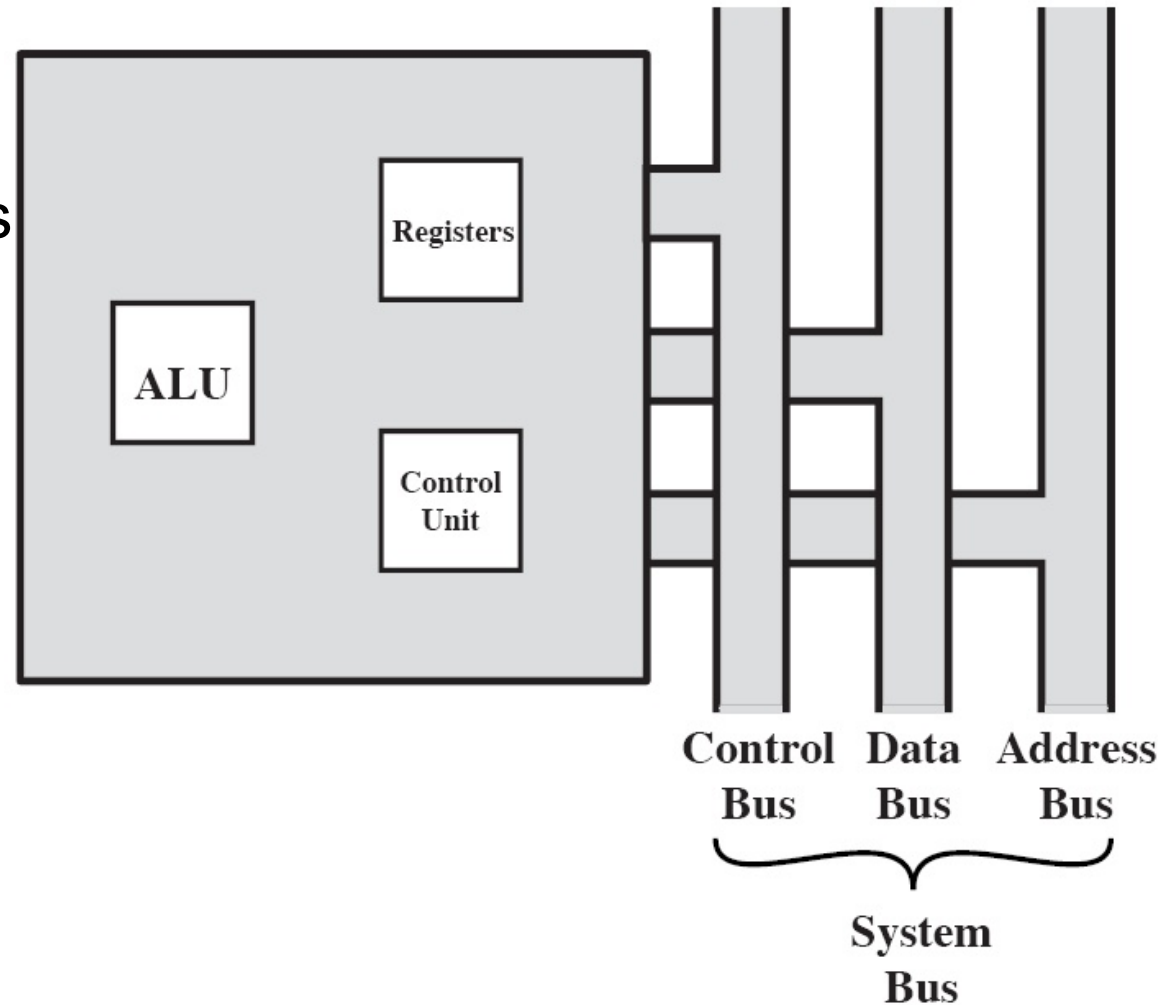
# Lecture 3

- Basic concepts
- Pipeline hazards
- Branch handling
- Branch prediction



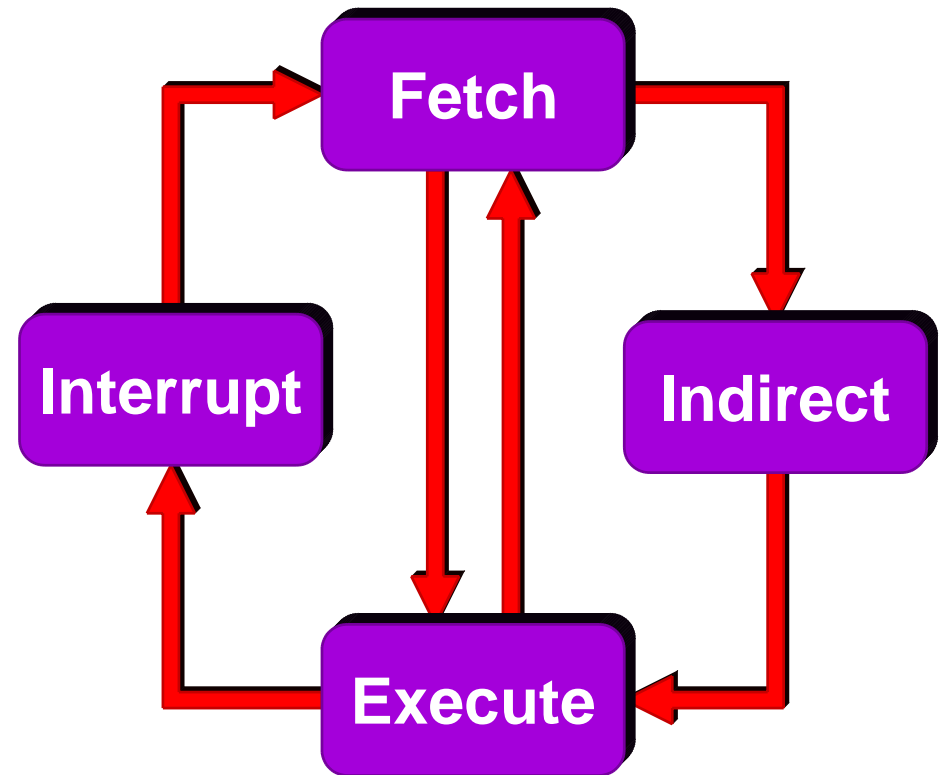
# CPU Structure

- CPU must:
  - Fetch instructions
  - Interpret instructions
  - Fetch data
  - Process data
  - Write data

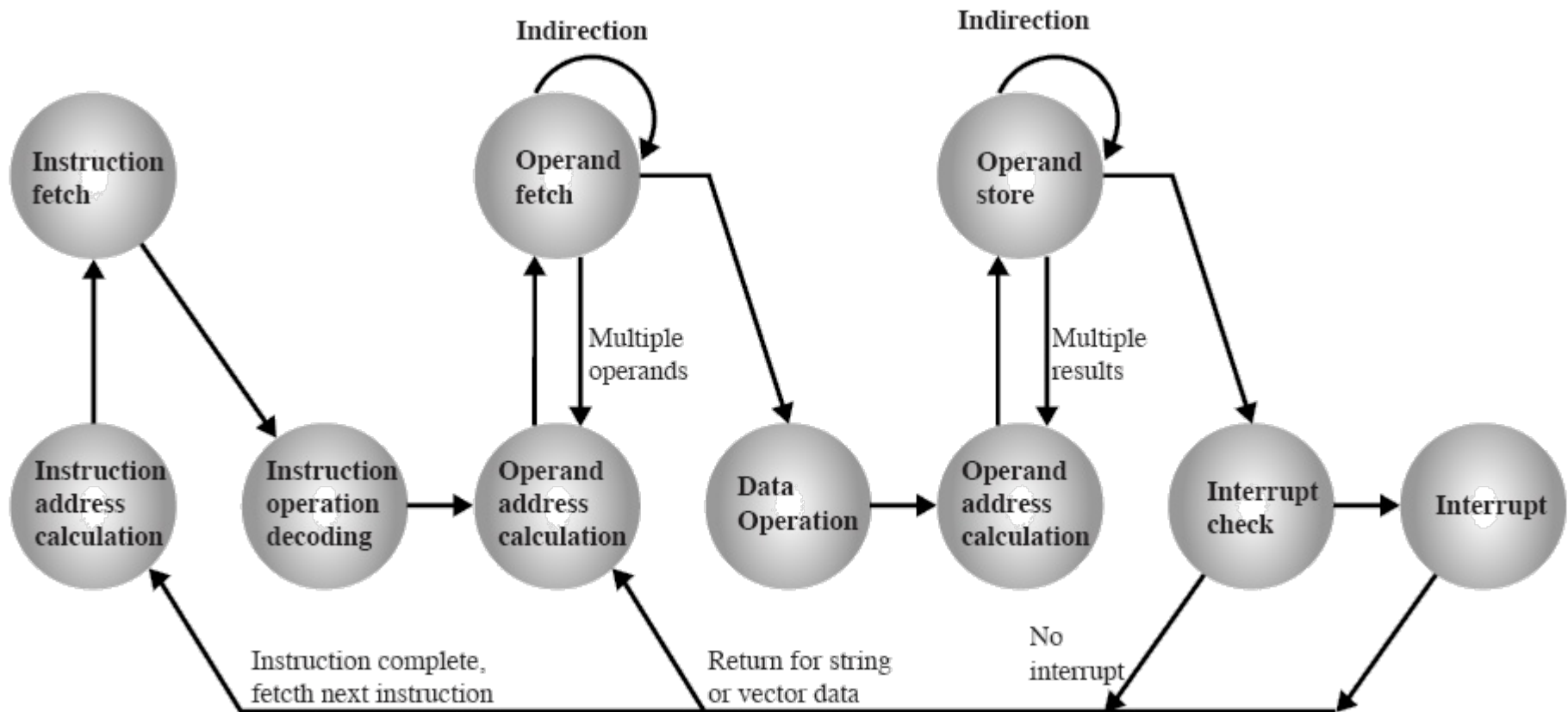


# Instruction Cycle

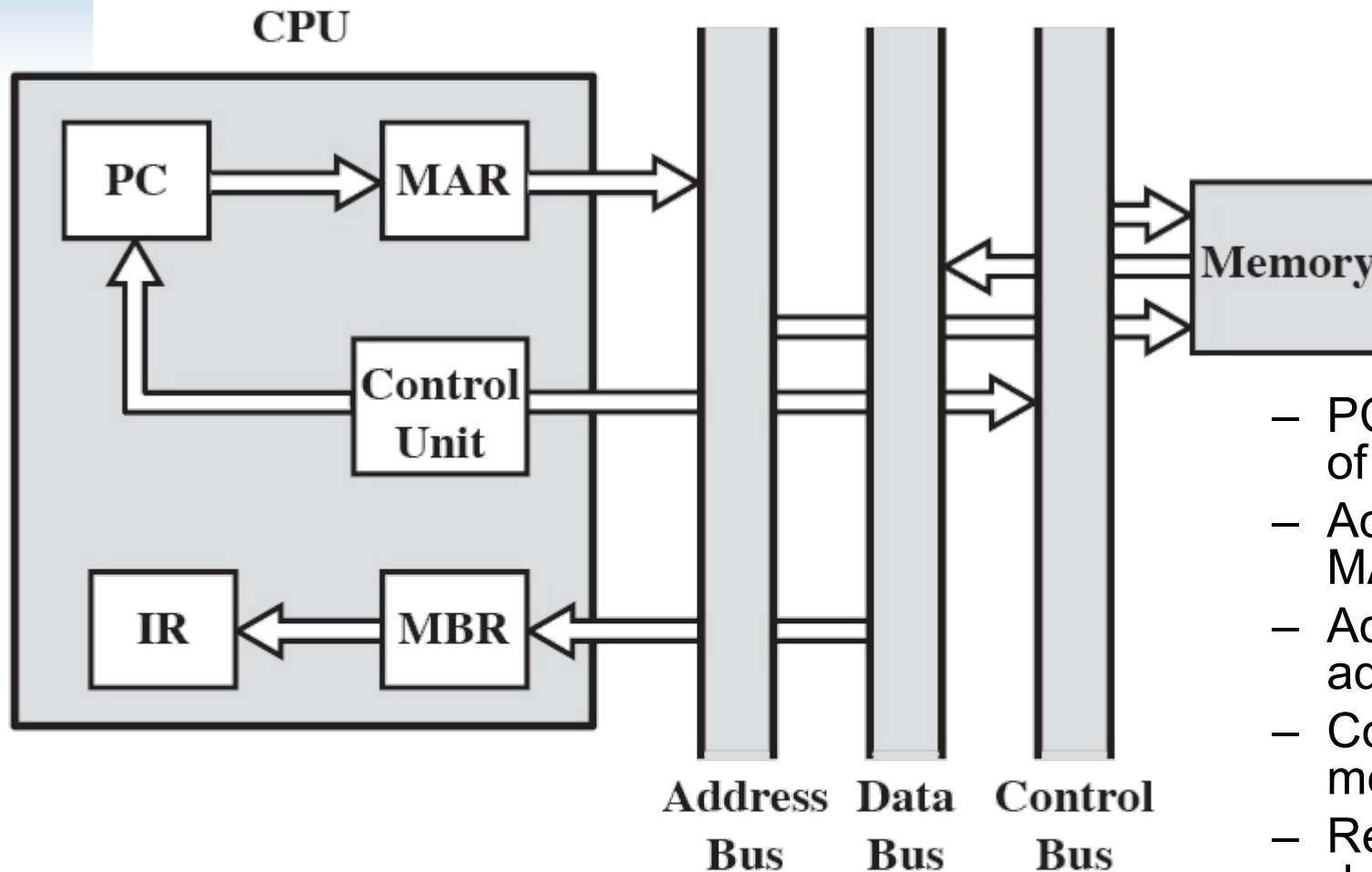
- Fetch cycle
- Indirect cycle
- Execute cycle
- Interrupt cycle



# Instruction Cycle State Diagram



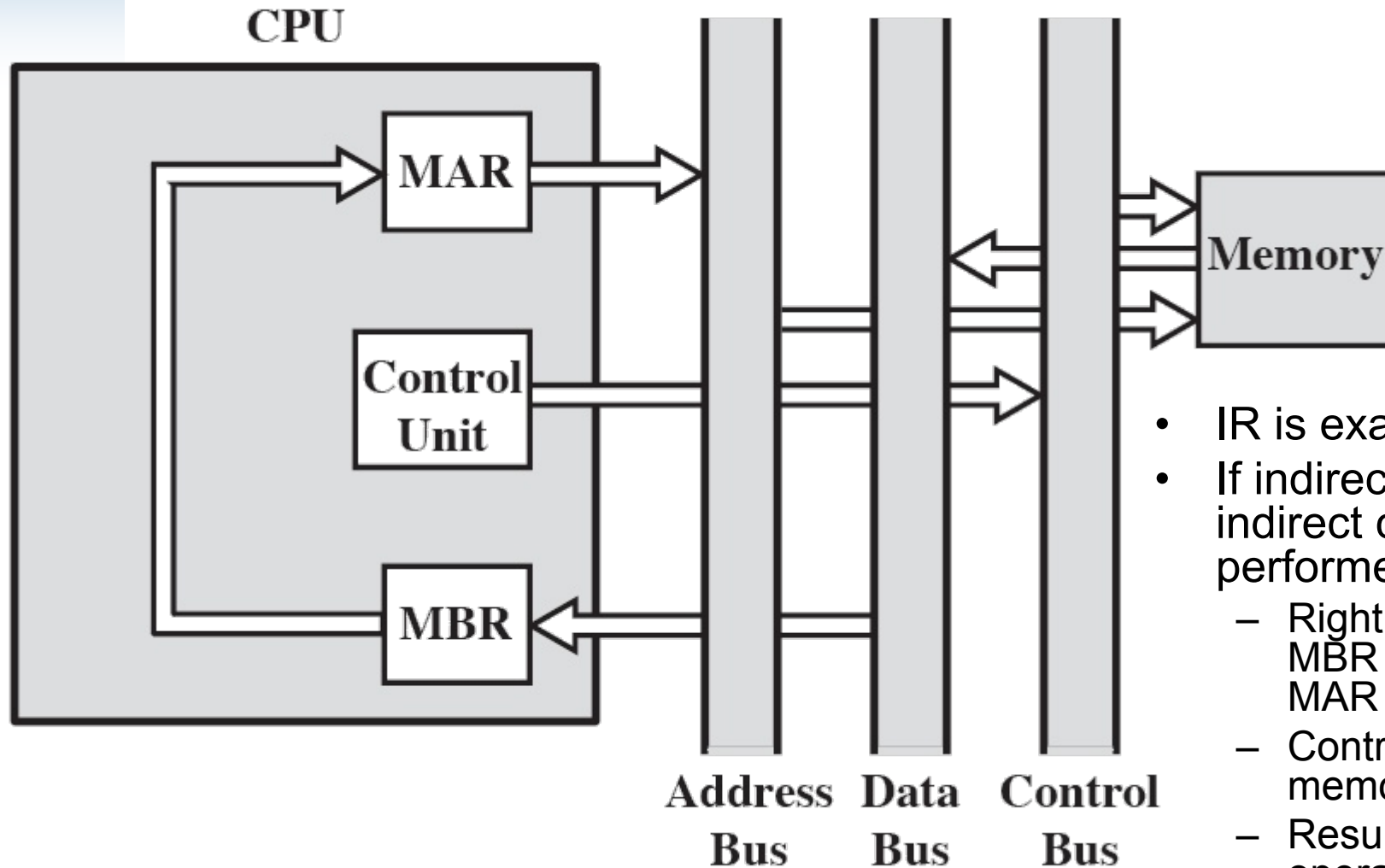
# Data Flow – Fetch Cycle



- PC contains address of next instruction
- Address moved to MAR
- Address placed on address bus
- Control unit requests memory read
- Result placed on data bus, copied to MBR, then to IR
- Meanwhile PC incremented by 1

MBR = Memory buffer register  
MAR = Memory address register  
IR = Instruction register  
PC = Program counter

# Data Flow – Indirect Cycle



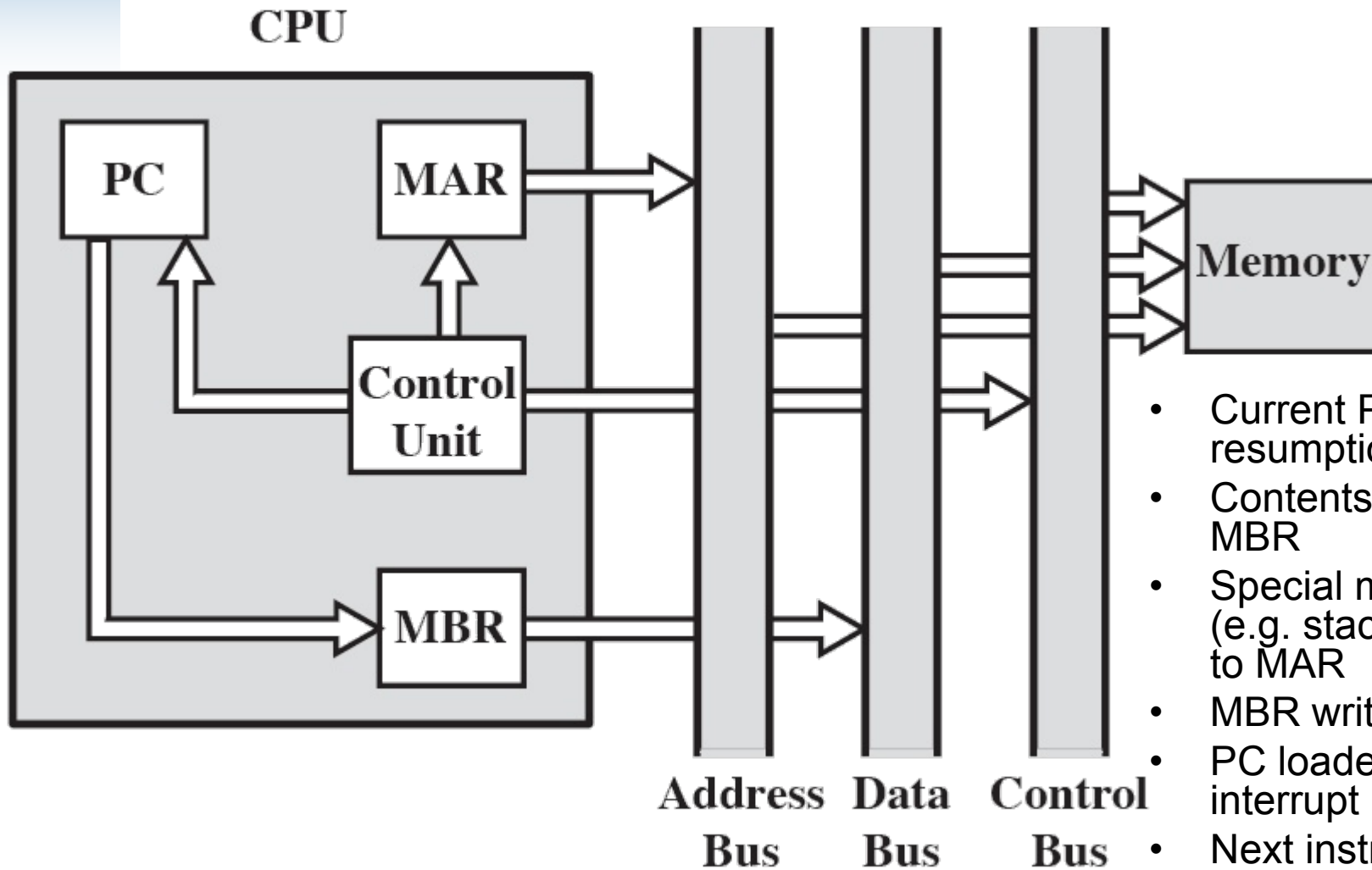
- IR is examined
- If indirect addressing, indirect cycle is performed
  - Right most N bits of MBR transferred to MAR
  - Control unit requests memory read
  - Result (address of operand) moved to MBR

# Data Flow – Execute Cycle

- May take many forms
- Depends on instruction being executed
- May include
  - Memory read/write
  - Input/Output
  - Register transfers
  - ALU operations



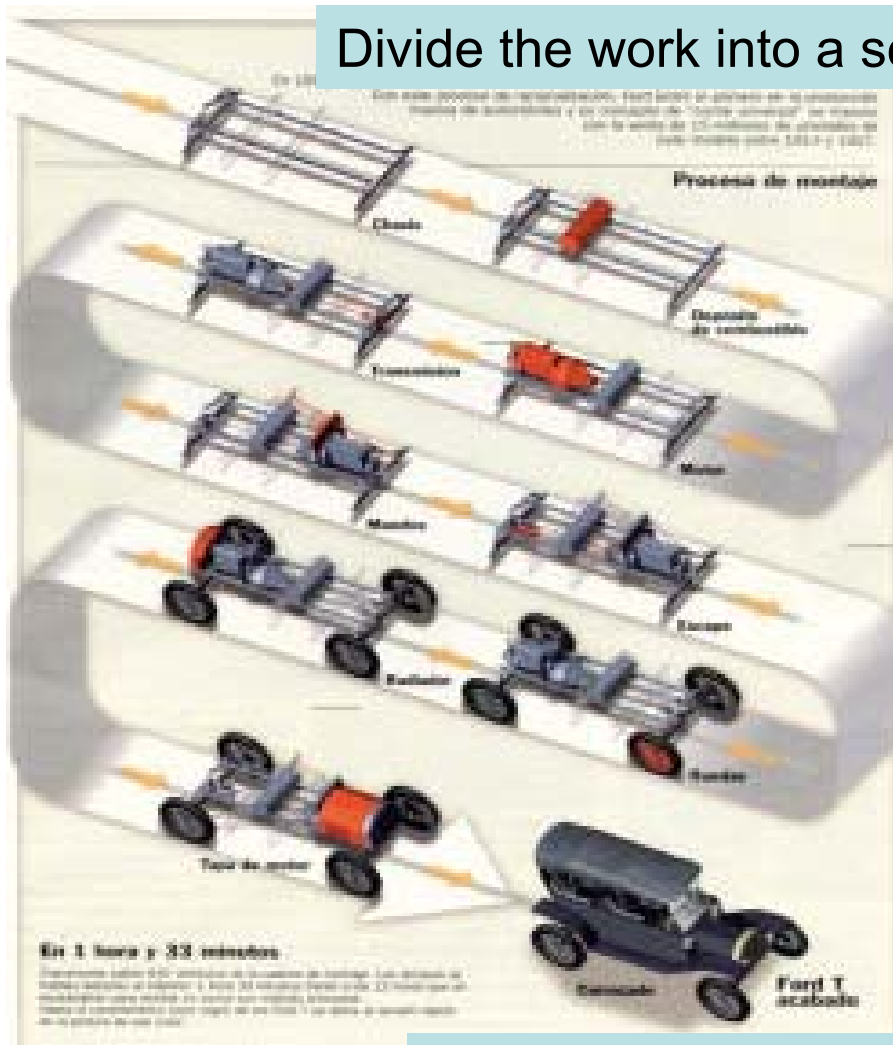
# Data Flow – Interrupt Cycle



- Current PC saved to allow resumption after interrupt
- Contents of PC copied to MBR
- Special memory location (e.g. stack pointer) loaded to MAR
- MBR written to memory
- PC loaded with address of interrupt handling routine
- Next instruction (first of interrupt handler) can be fetched

# What is a Pipeline?

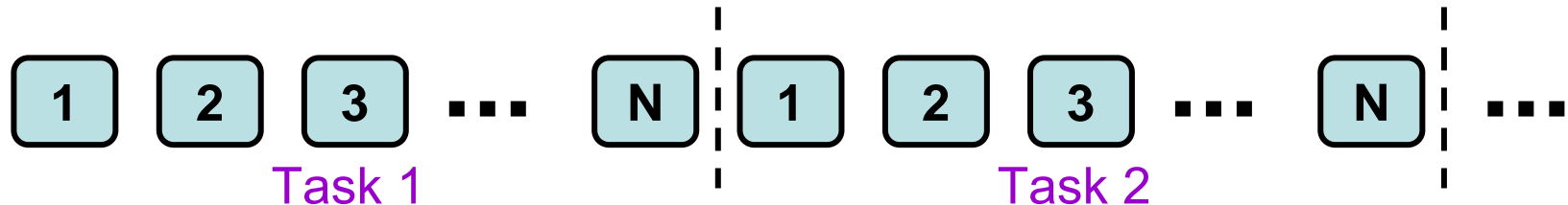
Divide the work into a sequence of simpler tasks.



Employ one worker for each simpler task.

# Basic Concepts

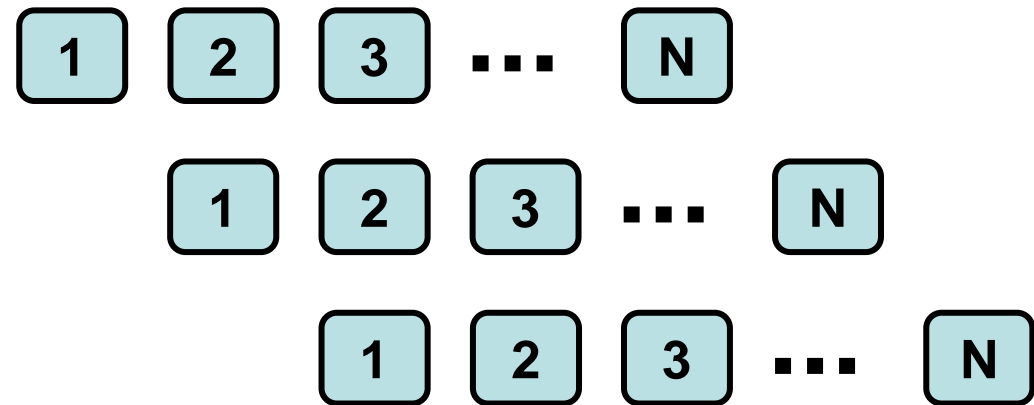
- Sequential execution of an N-stage task:



- Production time: N time units.
- Resource needed: one general-purpose machine.
- Productivity: **one product per N time units.**

- Pipelined execution of an N-stage task:

- Production time: N time units.
- Resource needed: N special-purpose machines.
- Productivity: about **one product per time unit.**



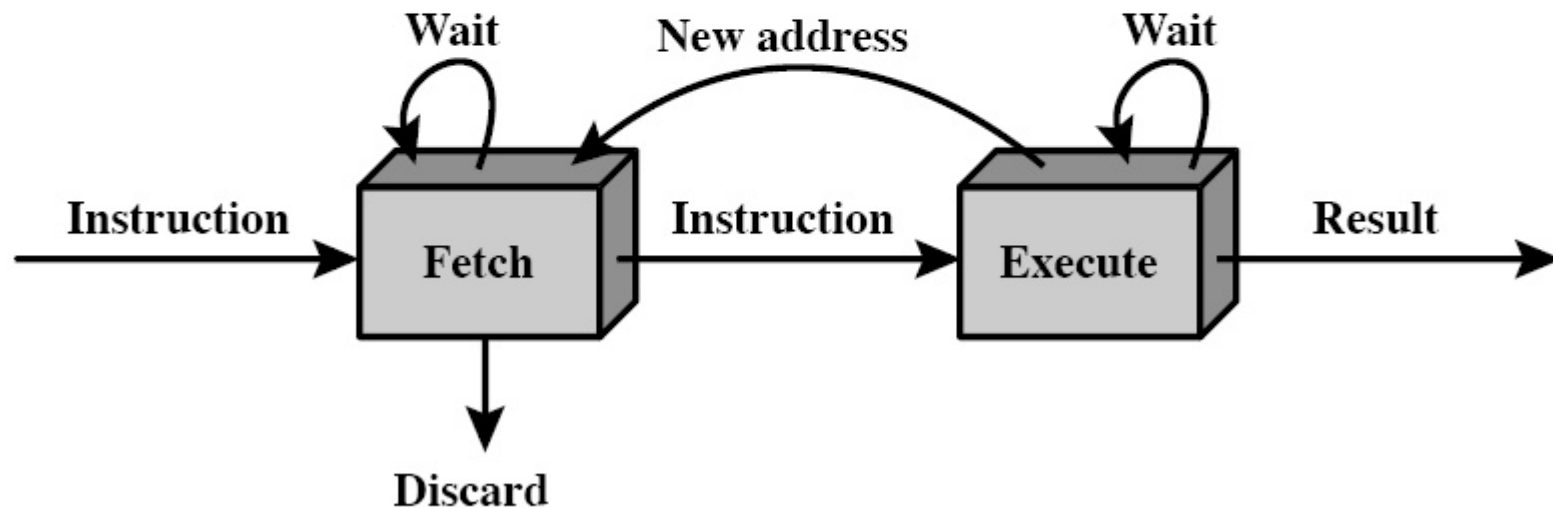
# Pipelining Strategy

- Prefetch
  - Fetch accessing main memory
  - Execution usually does not access main memory
  - Can fetch next instruction during execution of current instruction
  - Called instruction prefetch
- Improved Performance
  - But not doubled:
    - Fetch usually shorter than execution
      - Prefetch more than one instruction?
    - Any jump or branch means that prefetched instructions are not the required instructions
  - Add more stages to improve performance

# 2-stage Instruction Pipeline



(a) Simplified view

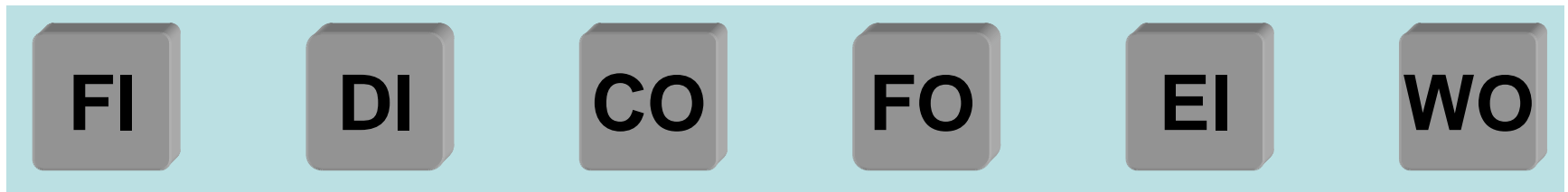


(b) Expanded view

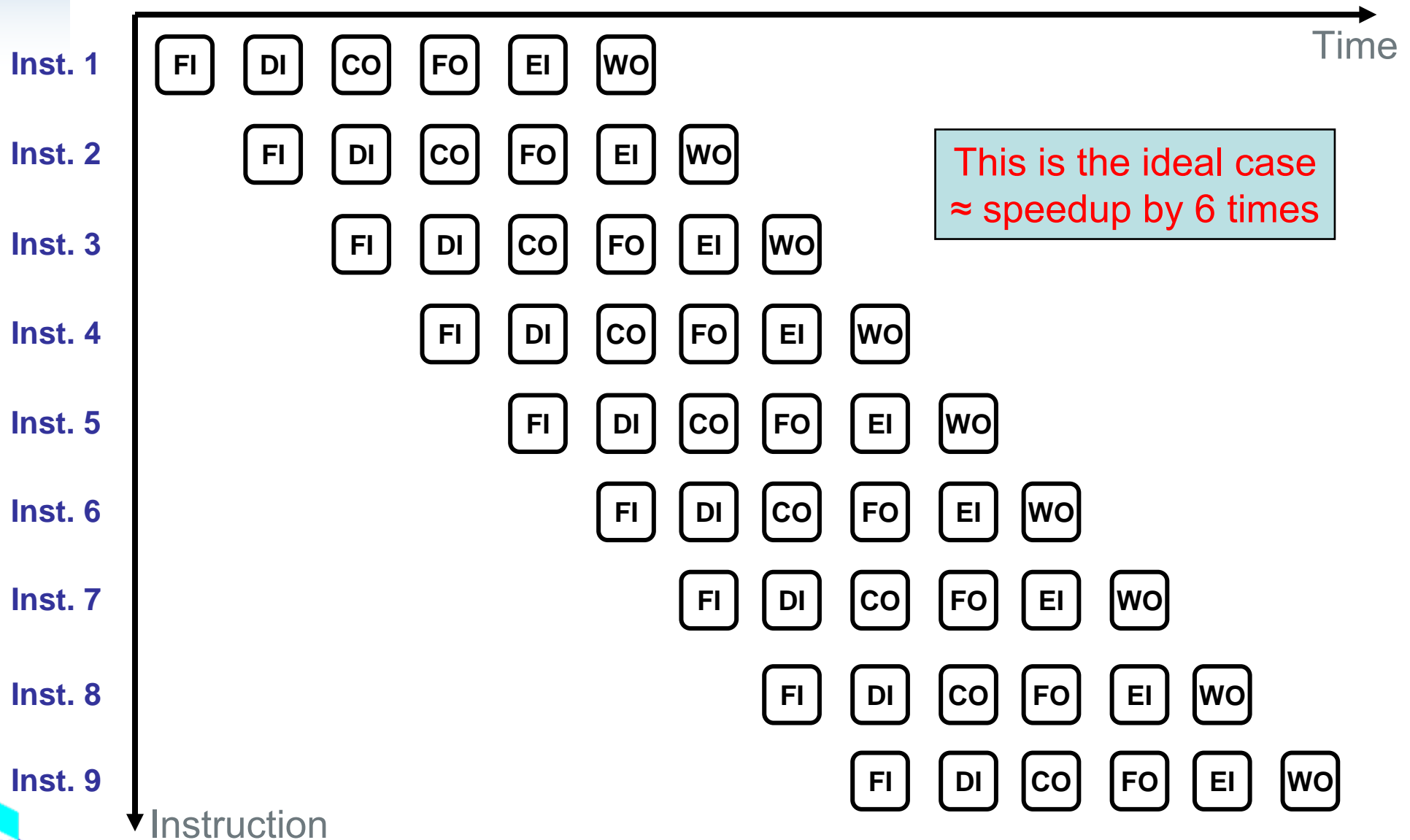
# Instruction Execution Stages

A typical instruction execution sequence:

1. Fetch Instruction (FI): Fetch the instruction.
2. Decode Instruction (DI): Determine the op-code and the operand specifiers.
3. Calculate Operands (CO): Calculate the effective addresses.
4. Fetch Operands (FO): Fetch the operands.
5. Execute Instruction (EI): Perform the operation.
6. Write Operand (WO): Store the result in memory.



# Instruction Pipelining



# Alternative Pipeline Depiction

Time ↓

	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



# Lecture 3

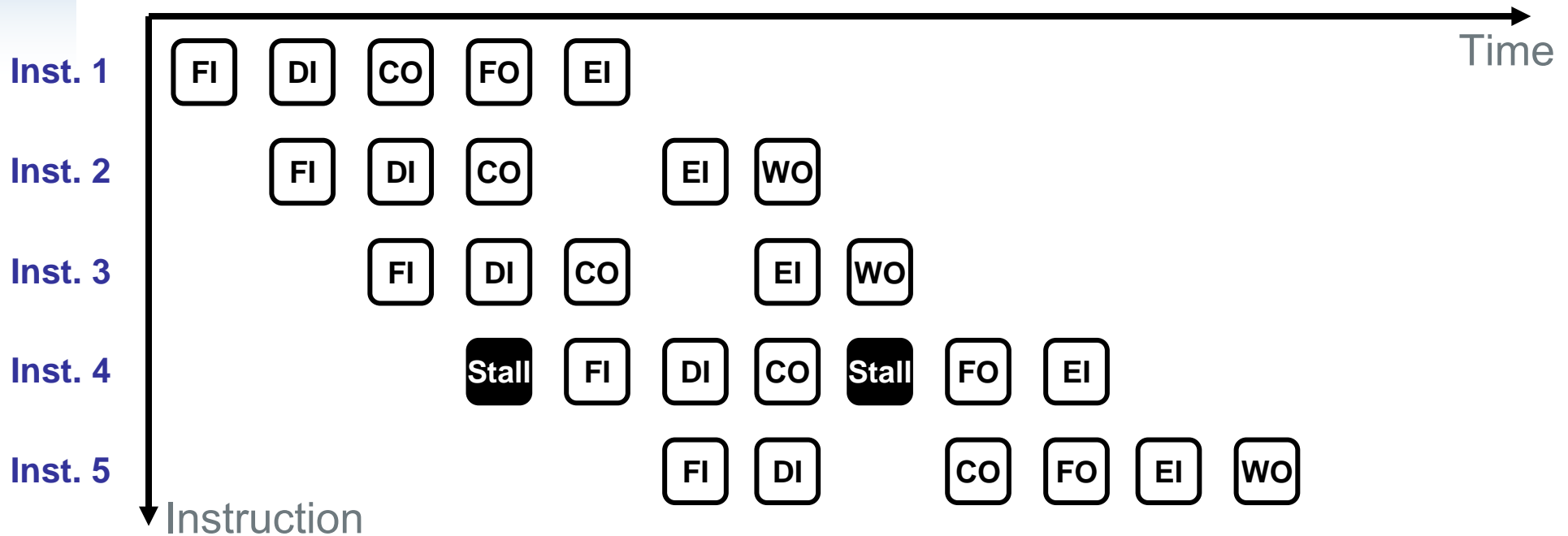
- Basic concepts
- Pipeline hazards
- Branch handling
- Branch prediction



# Pipeline Hazards

- There are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle. The instruction is said to be **stalled**.
- When an instruction is stalled:
  - All instructions later in the pipeline than the stalled instruction are also stalled;
  - No new instructions are fetched during the stall;
  - Instructions earlier than the stalled one continue as usual.
- Types of hazards (conflicts):
  - Structural hazards
  - Data hazards
  - Control hazards

# Structural Hazards



- Hardware conflicts – caused by the use of the same hardware resource at the same time (e.g., memory conflicts).
- Penalty: 1 cycle.

# Structural Hazard Solutions

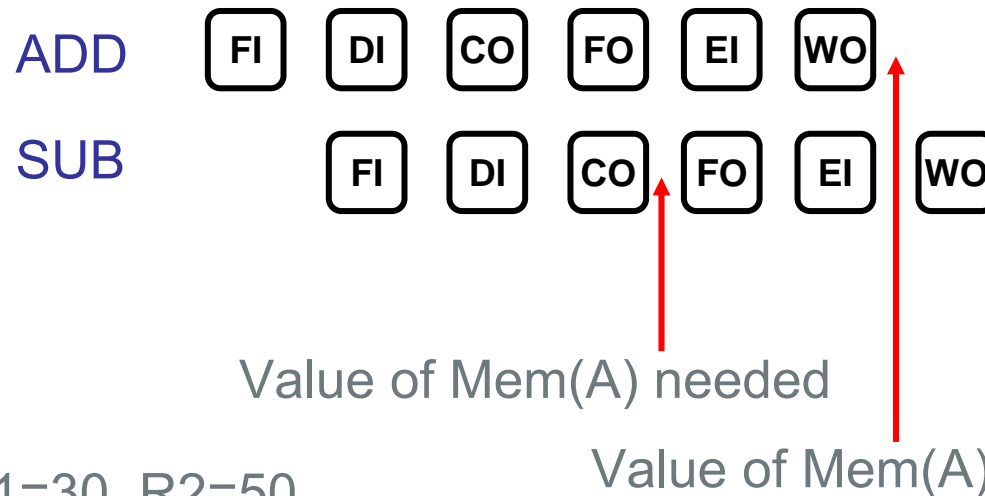
- In general, the hardware resources in conflict are duplicated in order to avoid structural hazards.
- Functional units (ALU, FP unit) can also be pipelined themselves in order to support several instructions at the same time.
- Memory conflicts can be solved by
  - having two separate caches, one for instructions and the other for operands (Harvard architecture);
  - Using multiple banks of the main memory; or
  - keeping as many intermediate results as possible in the registers.

# Data Hazards

- Caused by reversing the order of data-dependent operations due to the pipeline (e.g., WRITE/READ conflicts).

ADD A, R1;  
SUB A, R2;

$\text{Mem}(A) \leftarrow \text{Mem}(A) + R1$   
 $\text{Mem}(A) \leftarrow \text{Mem}(A) - R2$



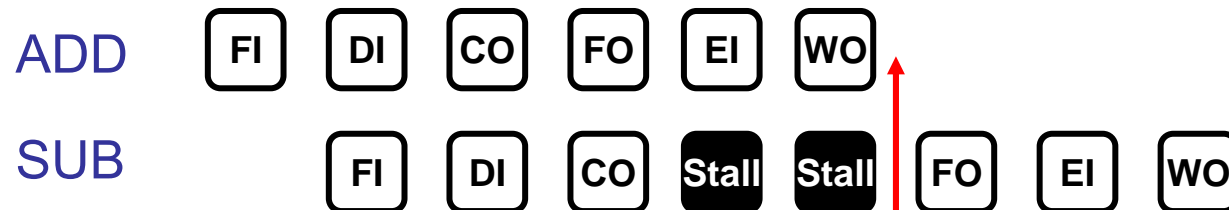
$\text{Mem}(200)=100, R1=30, R2=50$

# Data Hazards (Cont'd)

- Penalty: 2 cycles

ADD A, R1;  
SUB A, R2;

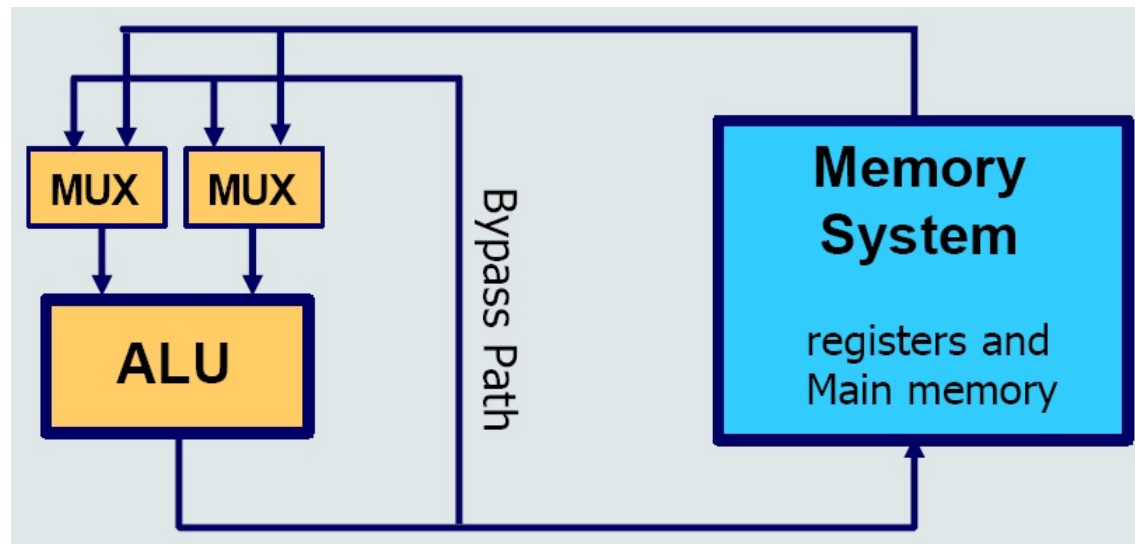
$\text{Mem}(A) \leftarrow \text{Mem}(A) + R1$   
 $\text{Mem}(A) \leftarrow \text{Mem}(A) - R2$



Value of Mem(A) available

# Data Hazard Solutions

- The penalty due to data hazards can be reduced by a technique called forwarding (bypassing).



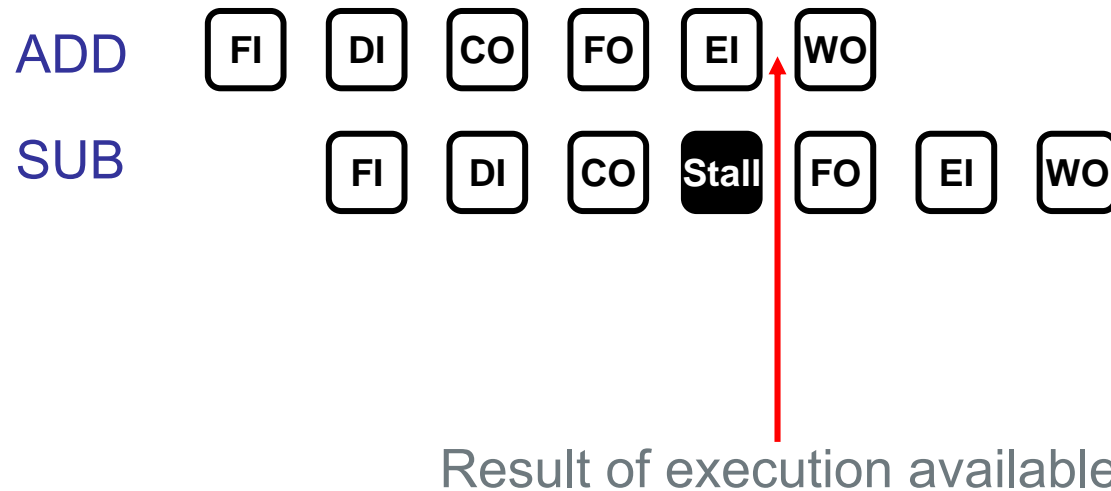
- The ALU result is fed back to the ALU input. If the hardware detects that the value needed for the current operation is the one produced by the previous operation (but which has not yet been written back) it selects the forwarded result, instead of the value from register or memory.

# Data Hazard Solutions (Cont'd)

- Penalty: Reduced to 1 cycle

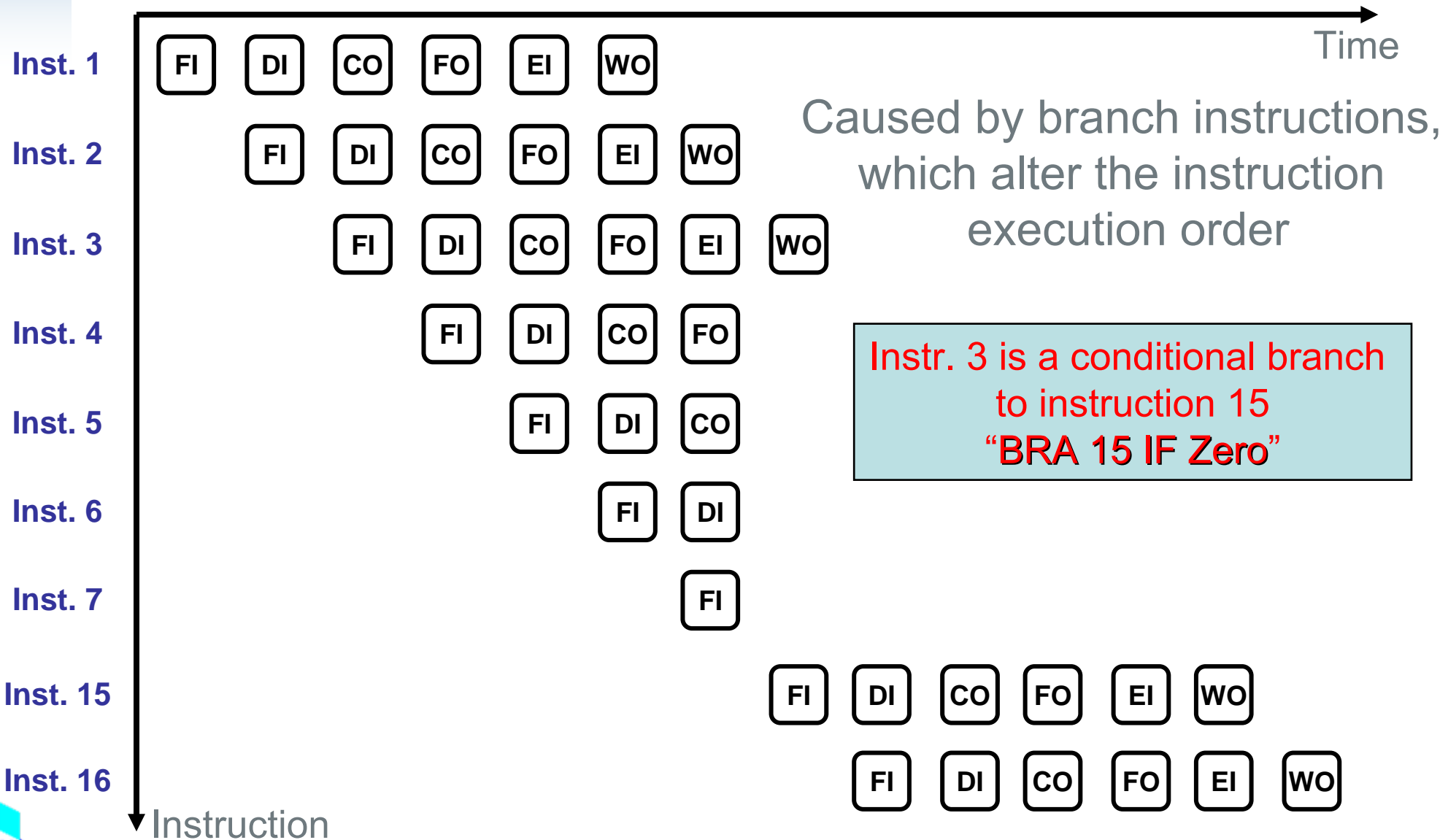
ADD A, R1;  
SUB A, R2;

$\text{Mem}(A) \leftarrow \text{Mem}(A) + R1$   
 $\text{Mem}(A) \leftarrow \text{Mem}(A) - R2$

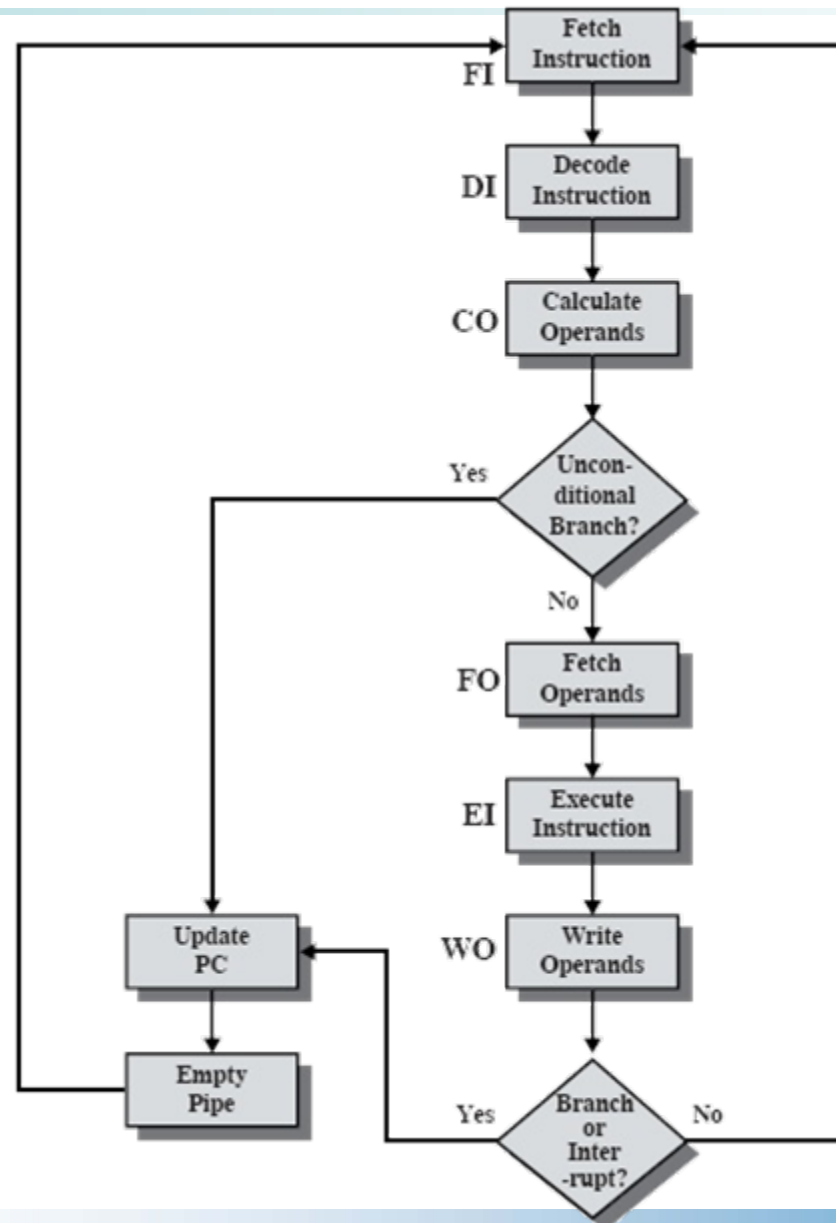




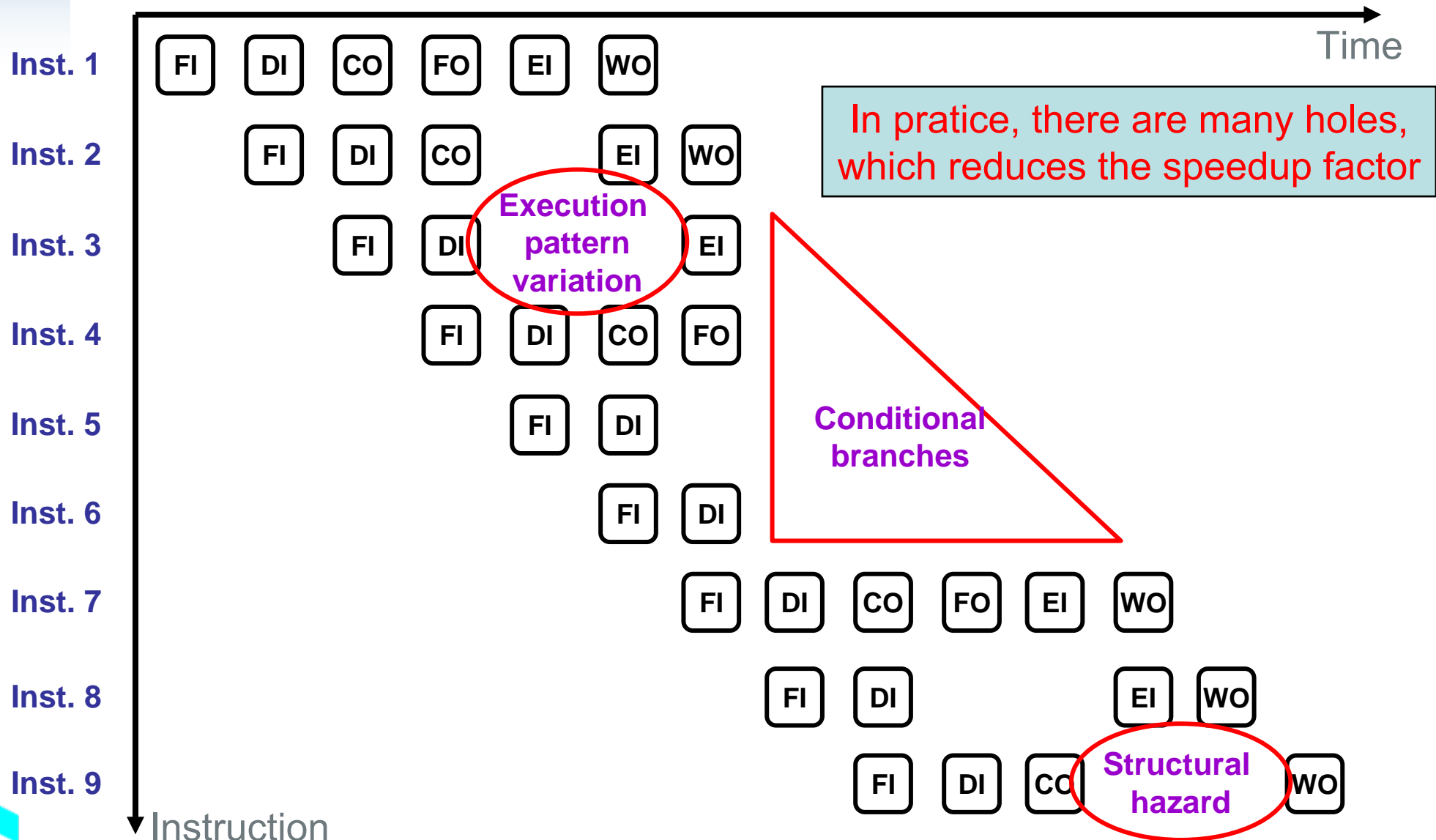
# Control Hazards



# 6-stage CPU Instruction Pipeline



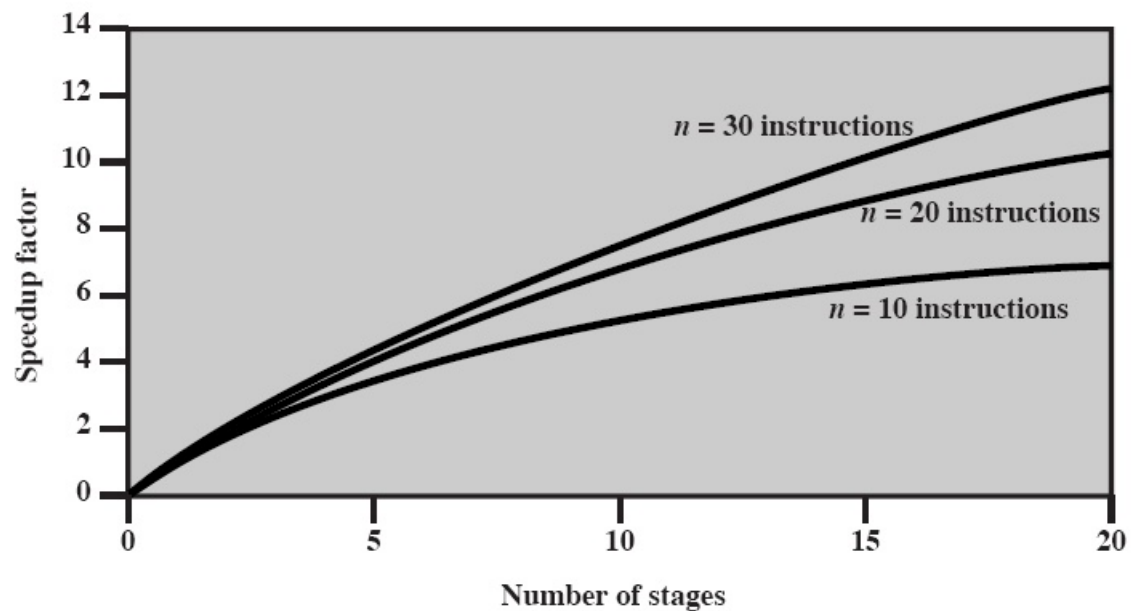
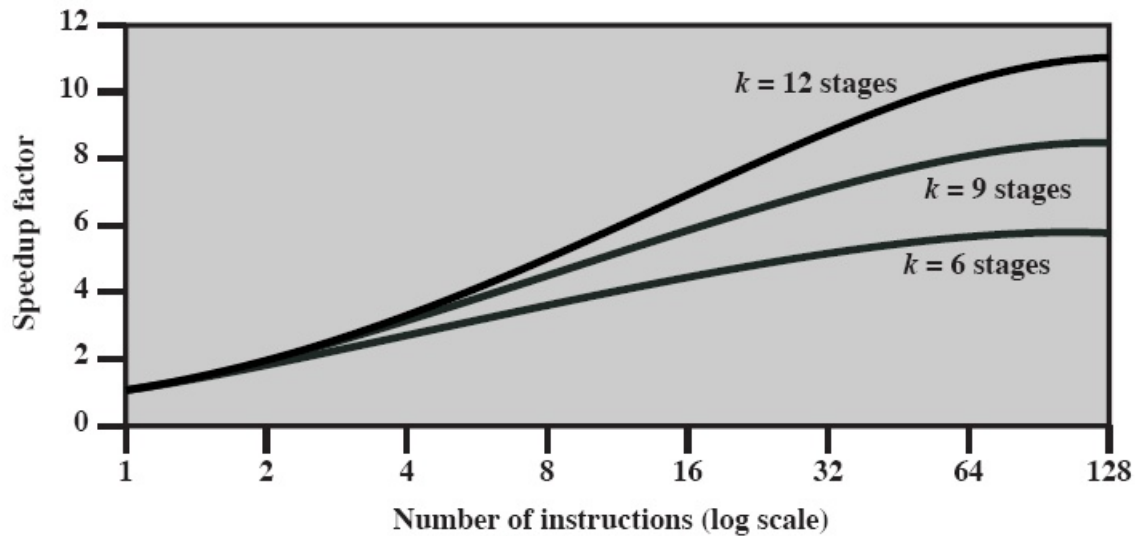
# Instruction Pipelining in Practice



# Number of Pipeline Stages

- In general, a larger number of stages gives better performance.
- However:
  - A larger number of stages increases the overhead in moving information between stages and synchronization between stages.
  - The complexity of the CPU grows with the number of stages.
  - It is difficult to keep a large pipeline at maximum rate because of pipeline hazards.
- Intel 80486 and Pentium:
  - Five-stage pipeline for integer instructions.
  - Eight-stage pipeline for FP instructions.
- IBM PowerPC:
  - Four-stage pipeline for integer instructions.
  - Six-stage pipeline for FP instructions.

# Speedup Factors



# Lecture 3

- Basic concepts
- Pipeline hazards
- Branch handling
- Branch prediction

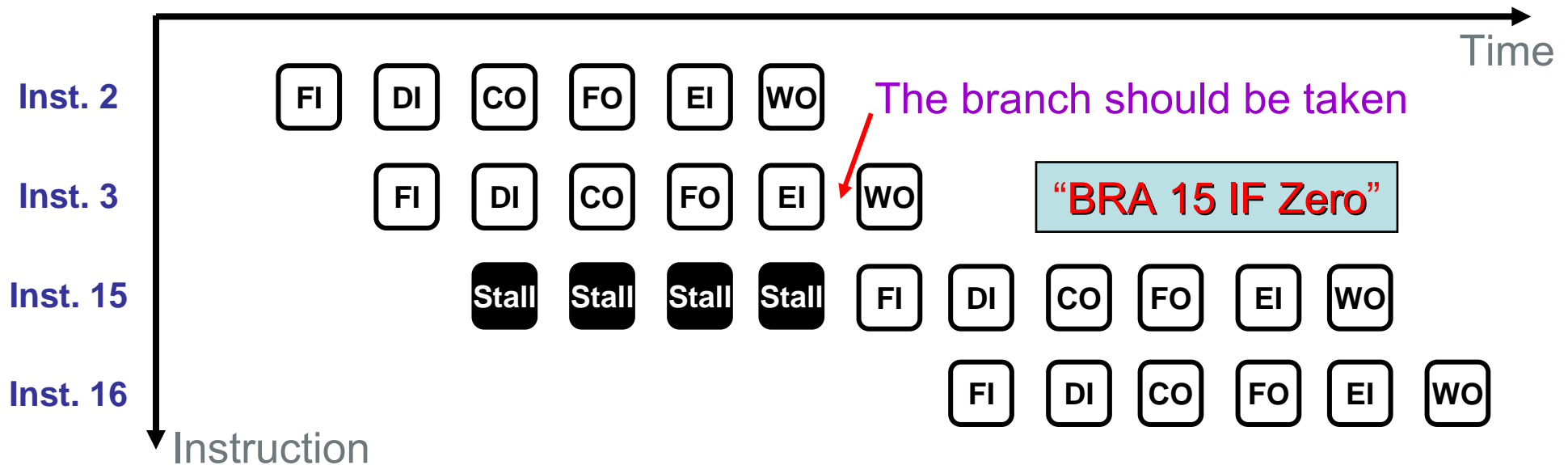


# Dealing with Branches

- One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline
- Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not
- A variety of approaches have been taken for dealing with conditional branches
  - Wait
  - Multiple streams
  - Prefetch branch target
  - Loop buffer
  - Delayed branch
  - Branch prediction

# Branch Handling – Wait

- Stop the pipeline until the branch instruction reaches the last stage.

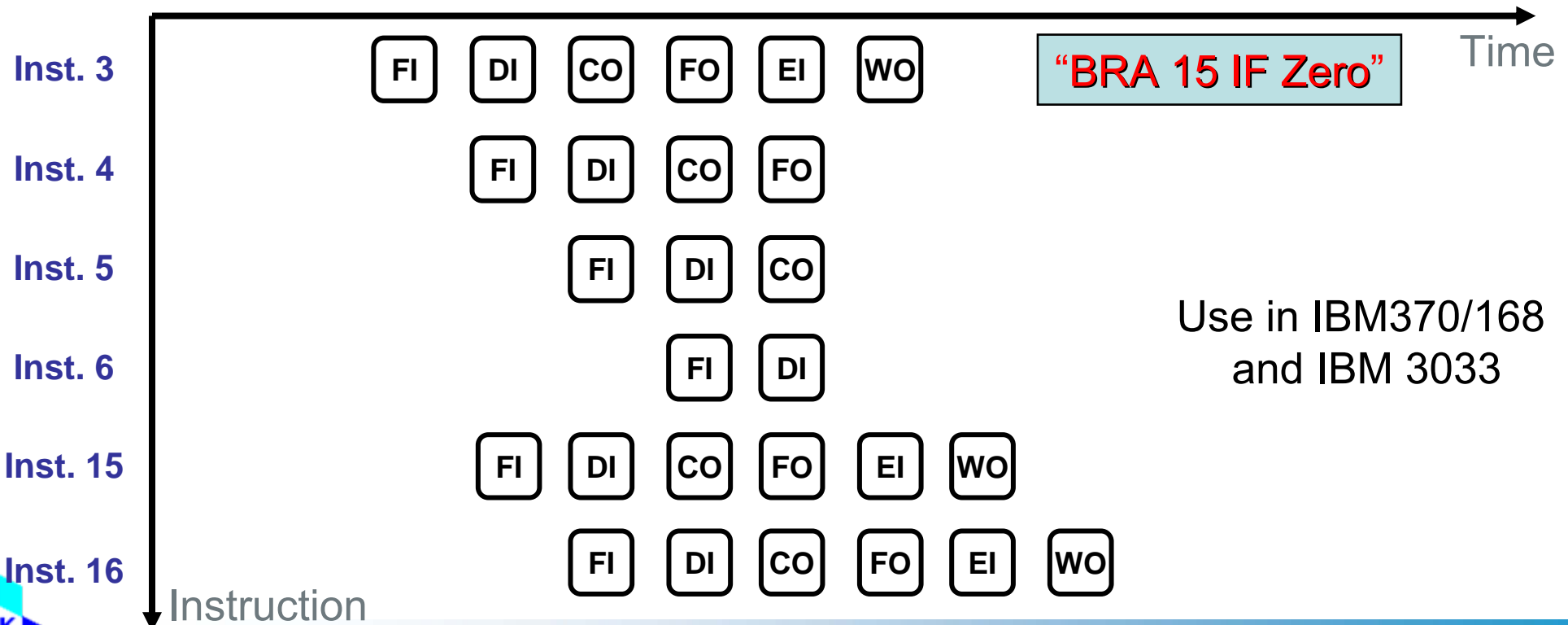


- Large loss of performance, since 20% - 35% of the instructions executed are branches (conditional and unconditional).



# Branch Handling – Multiple Streams

- Multiple streams — implement hardware to deal with all possibilities
  - Have two pipelines
  - Prefetch each branch into a separate pipeline
  - Use appropriate pipeline
  - Leads to bus & register contention
  - Multiple branches lead to further pipelines being needed

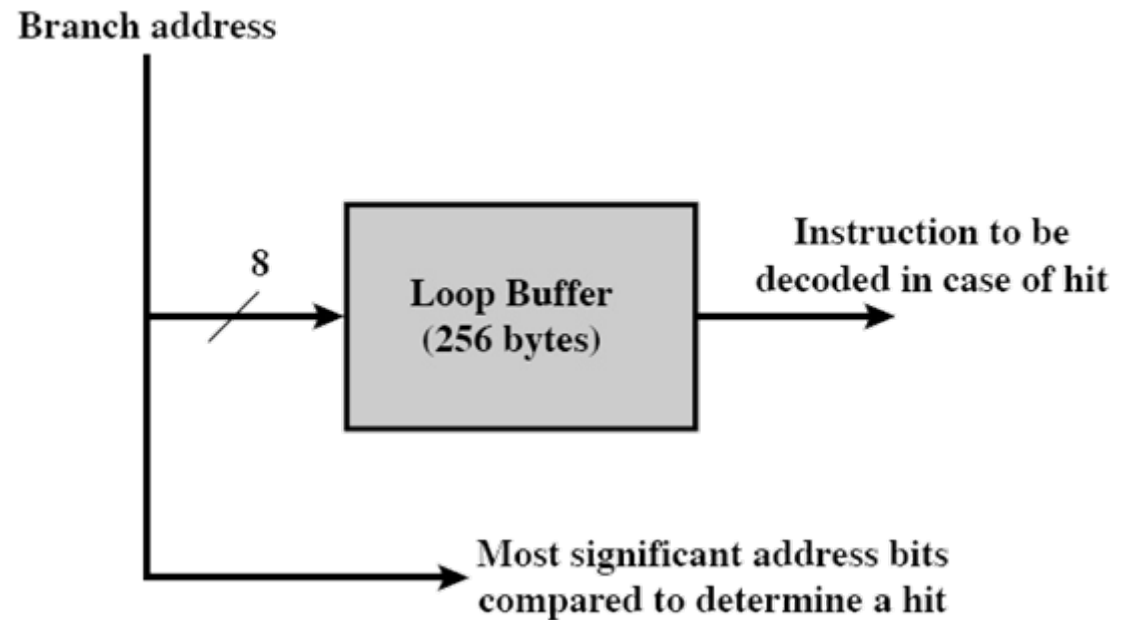


# Branch Handling – Pre-fetch br. target

- When a conditional branch is recognized, the target of the branch is pre-fetched, in addition to the instruction following the branch.
  - Keep target until branch is executed
  - Used by IBM 360/91

# Branch Handling – Loop Buffer (1)

- Use a small, very high-speed memory to keep the  $n$  most recently fetched instructions in sequence.
- If a branch is to be taken, the buffer is first checked to see if the branch target is in it. If so, the next instruction is fetched from the buffer
  - Used by CRAY-1



# Branch Handling – Loop Buffer (2)

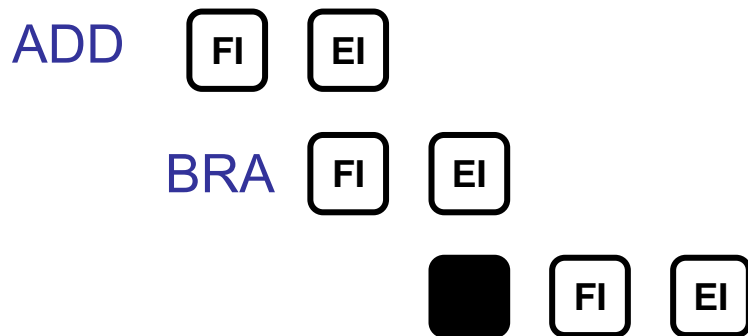
- Loop buffer's benefits
  - With the use of pre-fetching, the loop buffer will contain some instruction sequentially ahead of the current instruction. Thus, instructions fetched in sequence will be available w/o the usual memory access time.
  - If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer.
    - Useful for IF-THEN and IF-THEN-ELSE sequences
  - This technique is particularly well suited to dealing with loops or iterations.
    - Instructions in a loop are fetched from memory only once for the first iteration

# Branch Handling – Delayed Branch

- Re-arrange the instructions so that branching occur later than specified.

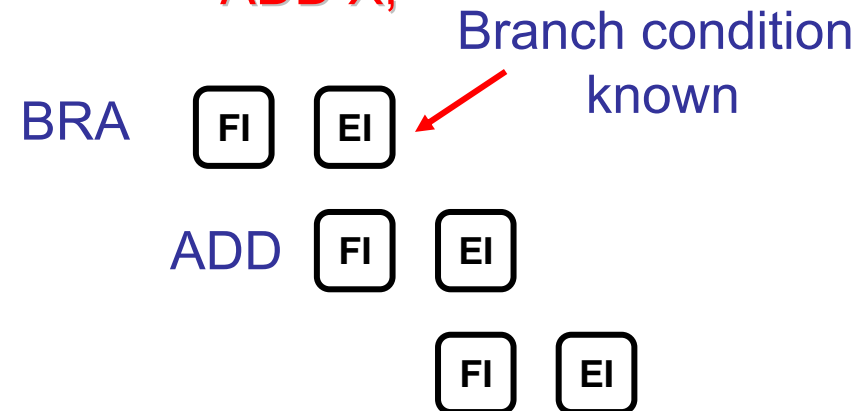
Normal instruction sequence:

ADD X;  
BRA L;



Delayed branch:

BRA L;  
ADD X;



- The compiler** has to find an instruction which can be moved from its original place to the branch delay slot, and will be executed regardless of the outcome of the branch (60% to 85% success rate).

# Lecture 3

- Basic concepts
- Pipeline hazards
- Branch handling
- Branch prediction

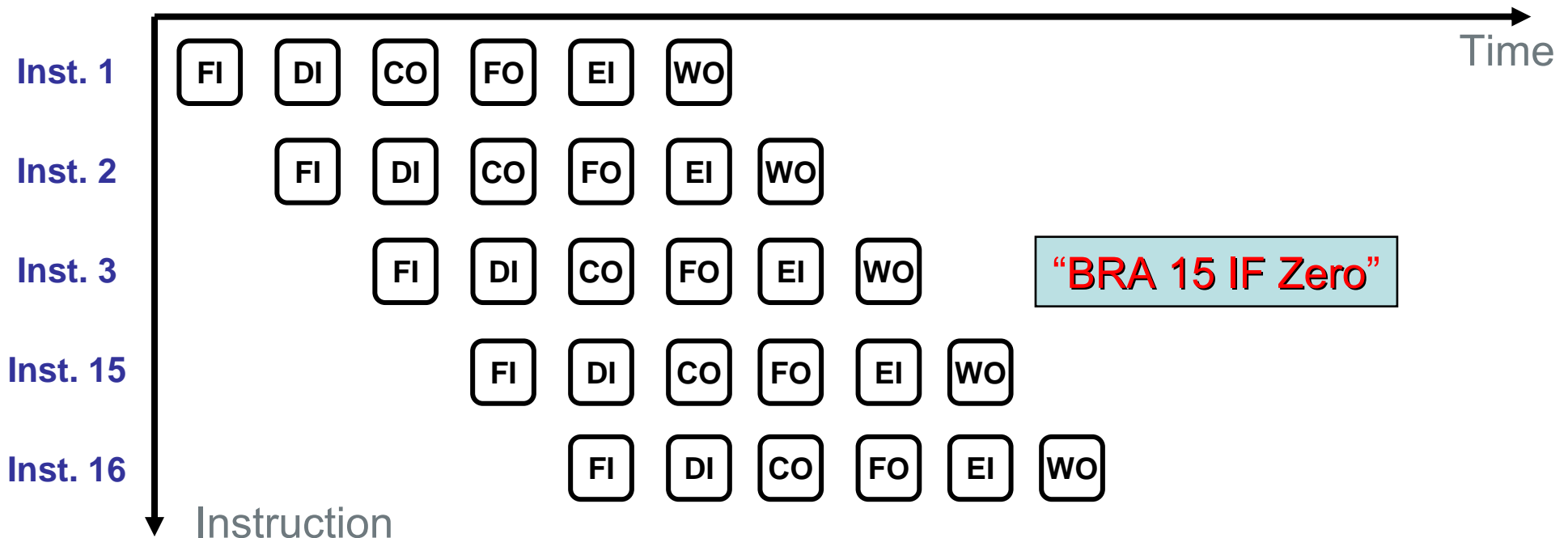


# Branch Prediction

- When a branch is encountered, a prediction is made and the predicted path is followed.
- The instructions on the predicted path are fetched.
- The fetched instruction can also be executed – Speculative execution. The results produced of these executions should be marked as tentative.
- When the branch outcome is decided, if the prediction is correct, the special tags on tentative results are removed.
- If not, the tentative results are removed. And the execution goes to the other path.
- Branch prediction can base on **static** information or **dynamic** information.

# Static Branch Prediction

- Predict always taken
  - Assume that jump will happen.
  - Always fetch target instruction.

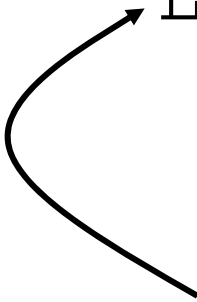




# Static Branch Prediction (Cont'd)

```
sum = 0; //Java code  
for (i = 0; i < 1000; i++)  
    sum += a[i];
```

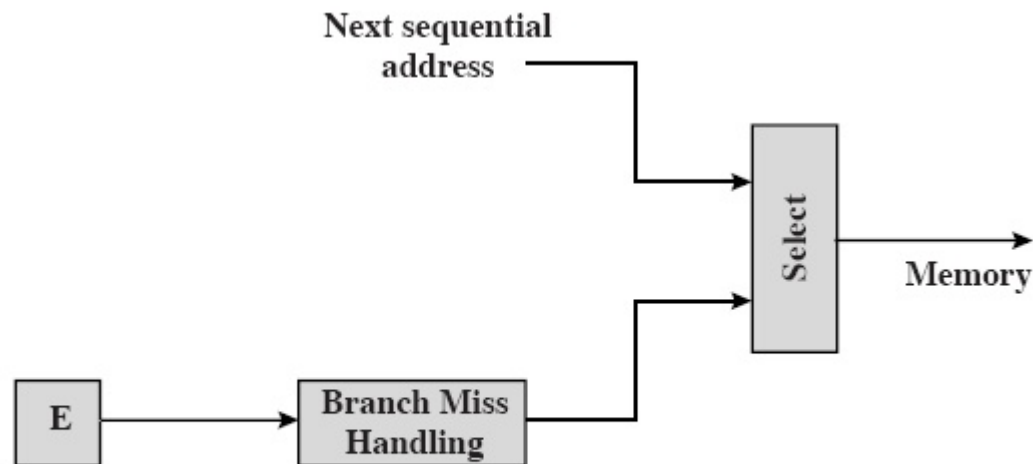
```
        MOVE #0, R0           ; sum  
        MOVE #1000, R1        ; index  
L1:     ADD R0, (R1)  
        ADD R1, #1  
        COMP R1, #'1000'  
        BNZ L1  
        MOVE R0, sum
```



The prediction will  
be correct 99.9% of  
the time with this  
example!!!

# Static Branch Prediction (Cont'd)

- Predict never taken
  - Assume that jump will not happen.
  - Always fetch next instruction.
  - 68020 & VAX 11/780
  - VAX will not prefetch instructions after branch if a page fault would result (O/S vs CPU design)



(a) Predict never taken strategy

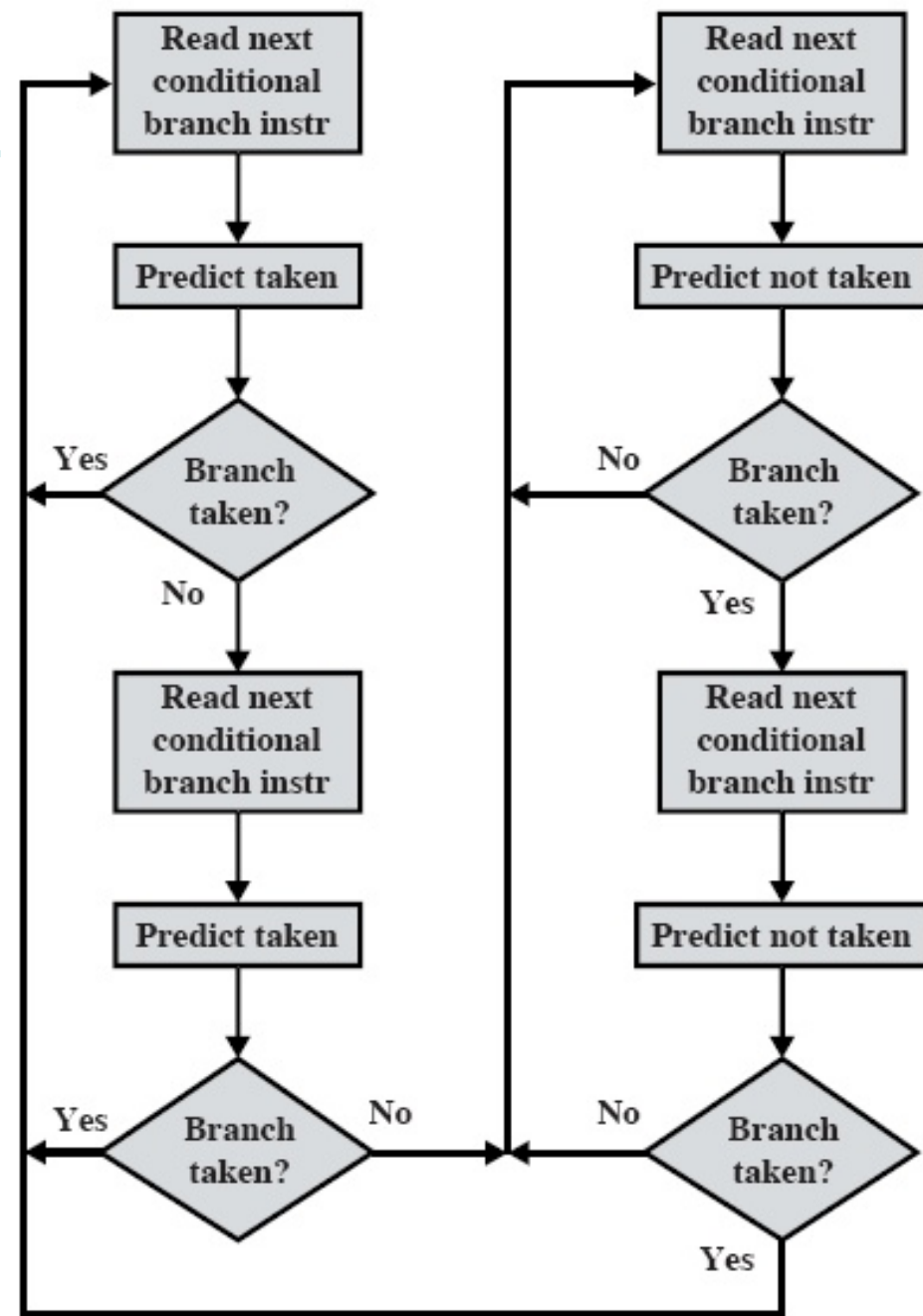
# Static Branch Prediction (Cont'd)

- Predict by Operation Codes
  - Some instructions are more likely to result in a jump than others.
  - Can get up to 75% success.

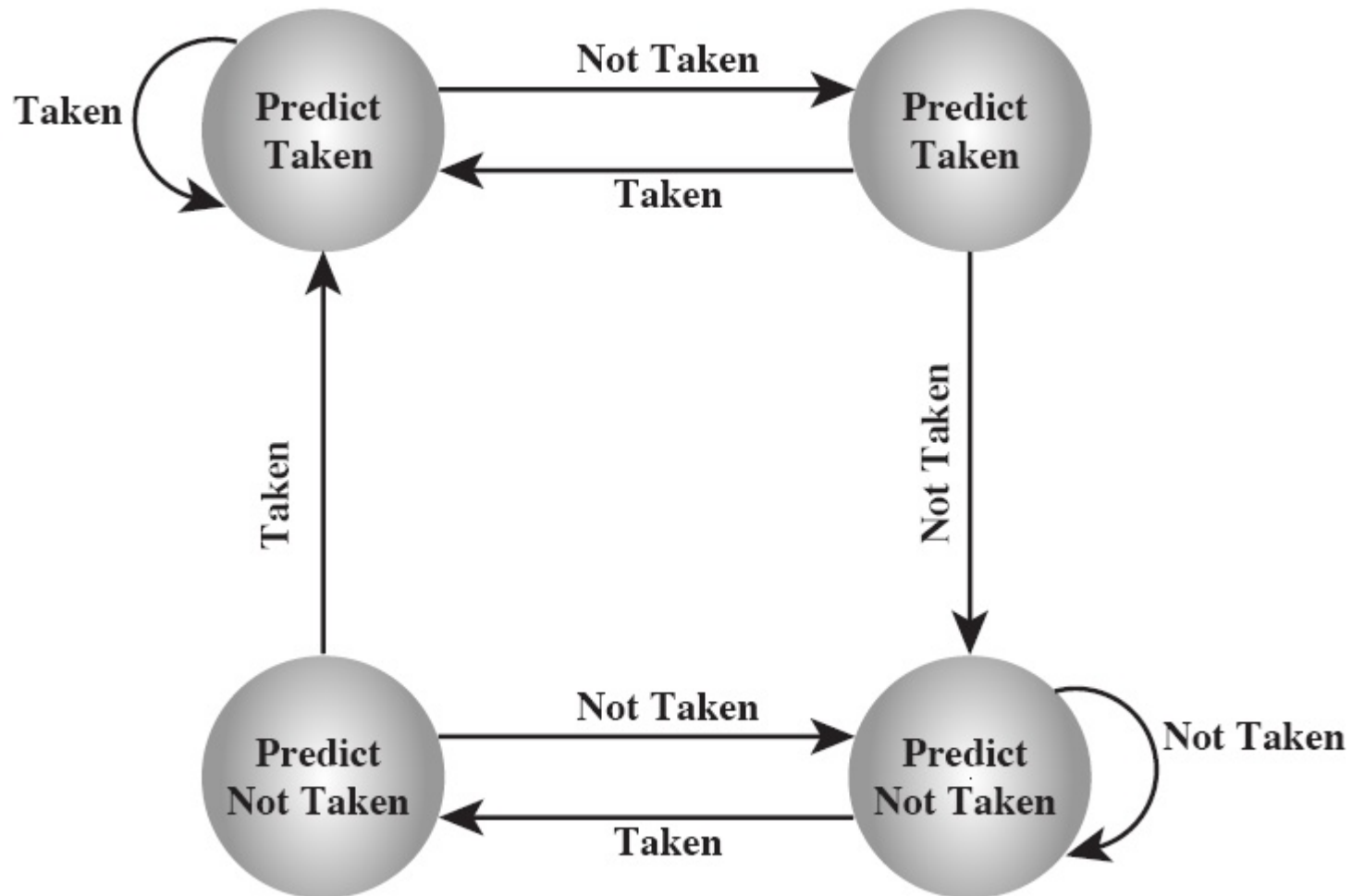
# Dynamic Branch Prediction

- Taken/Not taken switch
  - Based on previous history
  - Store information regarding branches in a branch-history table so as to more accurately predict the branch outcome.
    - Assume that the branch will do what it did last time.
  - One or more bits (**history bits**) can be associated with each conditional branch instruction that reflect the recent history of the instruction
  - History bits are kept in temporary high-speed storage
    - Associate history bits with any conditional branch instruction that is in a cache
    - Maintain a small table for recently executed branch instructions with history bits in each entry
  - Good for loops

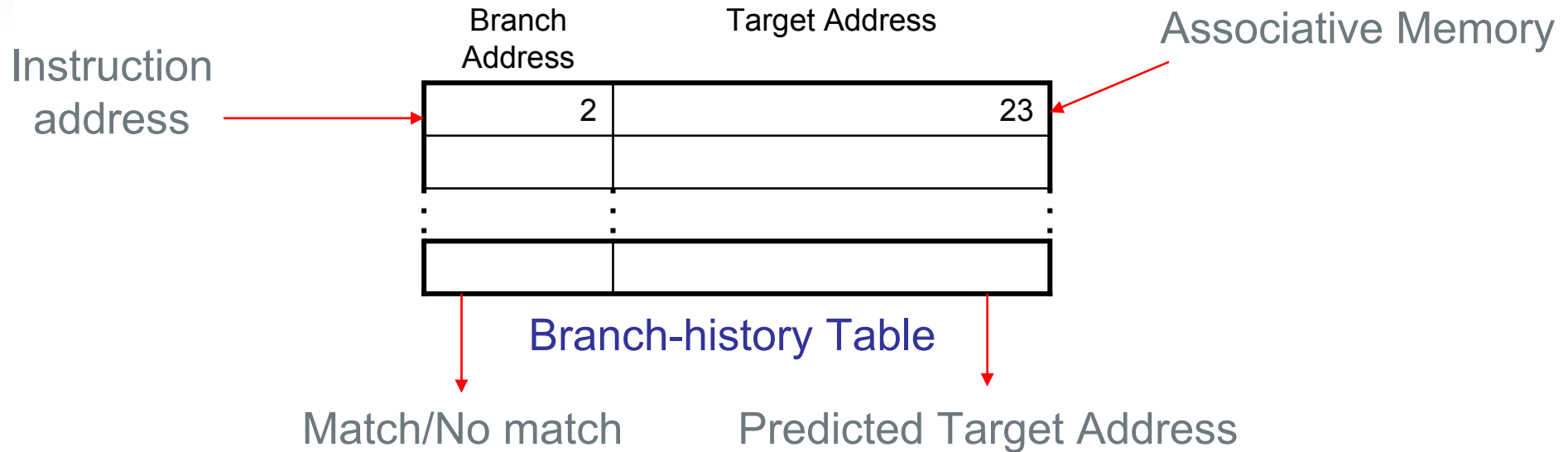
# Branch Prediction Flowchart



# Branch Prediction State Diagram

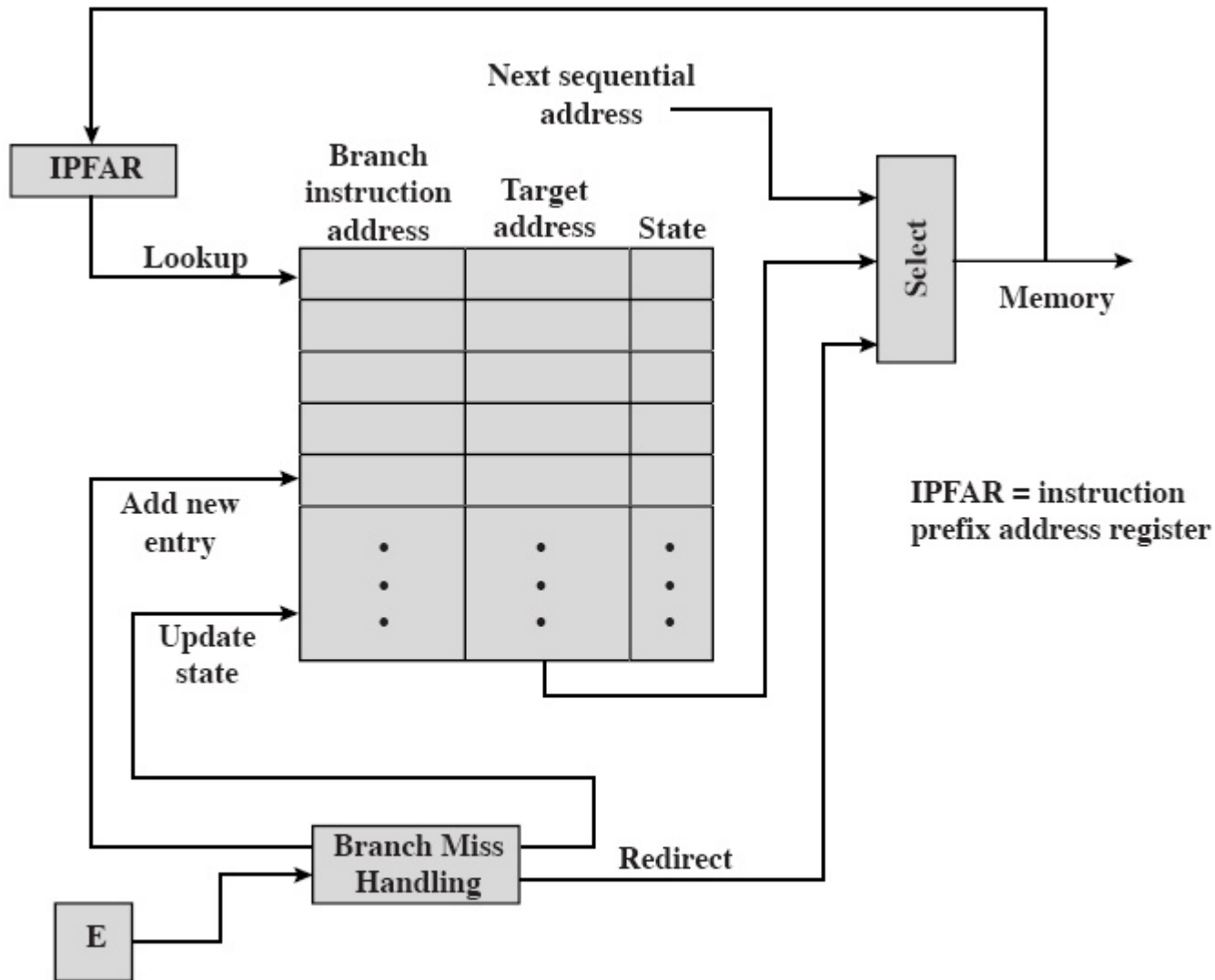


# Taken/Not taken switch



- Drawback
  - Because the target instruction address is an operand in the conditional branch instruction, target instruction cannot be fetched until the target address is decoded
- More information saved could achieve greater efficiency

# Branch History Table Strategy



(b) Branch history table strategy



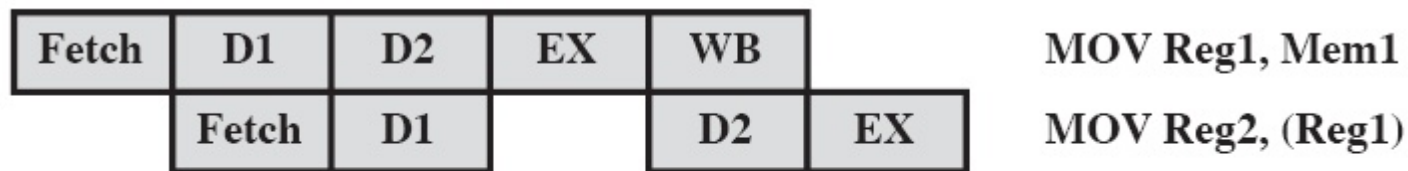
# Intel 80486 Pipelining

- Fetch
  - From cache or external memory
  - Put in one of two 16-byte prefetch buffers
  - Fill buffer with new data as soon as old data consumed
  - Average 5 instructions fetched per load
  - Independent of other stages to keep buffers full
- Decode stage 1
  - Opcode & address-mode info
  - At most first 3 bytes of instruction
  - Can direct D2 stage to get rest of instruction
- Decode stage 2
  - Expand opcode into control signals
  - Computation of complex address modes
- Execute
  - ALU operations, cache access, register update
- Writeback
  - Update registers & flags
  - Results sent to cache & bus interface write buffers

# 80486 Instruction Pipeline Examples



### (a) No Data Load Delay in the Pipeline



### (b) Pointer Load Delay



### (c) Branch Instruction Timing

# Summary

- Instruction execution can be substantially accelerated by instruction pipelining.
- A pipeline is organized as a succession of  $N$  stages. Ideally  $N$  instructions can be active inside a pipeline.
- Keeping a pipeline at its maximal rate is prevented by pipeline hazards.
  - Structural hazards are due to resource conflicts.
  - Data hazards are produced by data dependencies between instructions.
  - Control hazards are produced as consequence of branch instructions.
- Branch instructions can dramatically affect pipeline performance. It is very important to reduce penalties produced by branches.