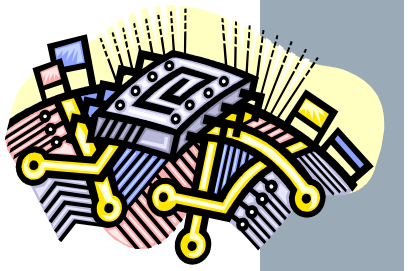
Chapter 7

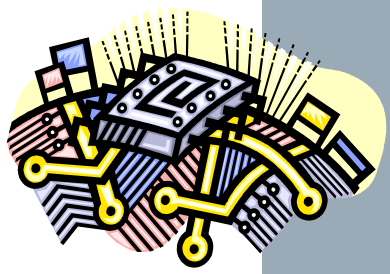
# Processor Structure and Function

# Contents



# 14.1- Processor Organization

## Processor Requirements:



- **Fetch instruction** (from memory (register, cache, main memory)
- **Interpret instruction** (what action is required)
- **Fetch data** (data from memory or an I/O module)
- **Process data** (performing some operations on data)
- **Write data** (writing result to memory or an I/O module)

➔ In order to do these things the processor needs to store some data temporarily and therefore needs a small internal memory

# CPU With the System Bus and CPU Internal Structure

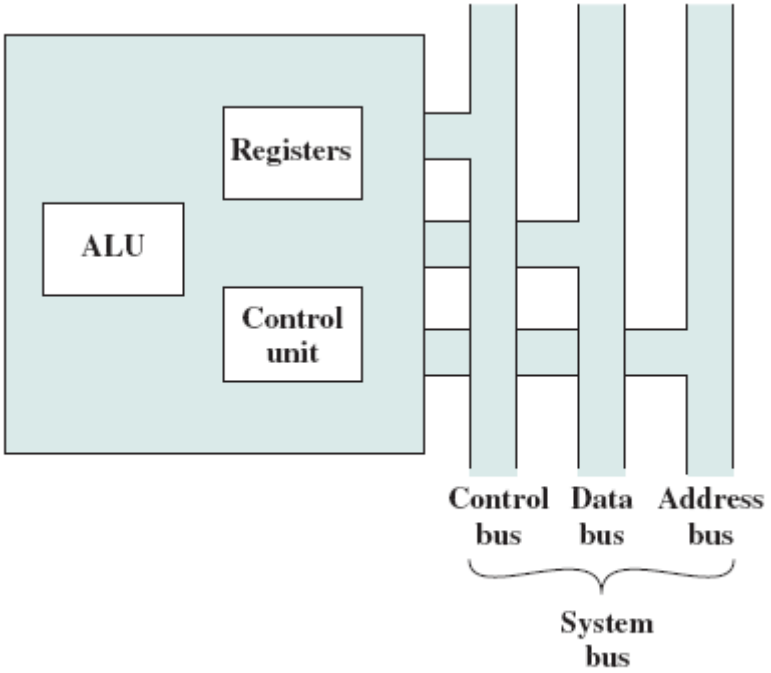



Figure 14.1 The CPU with the System Bus

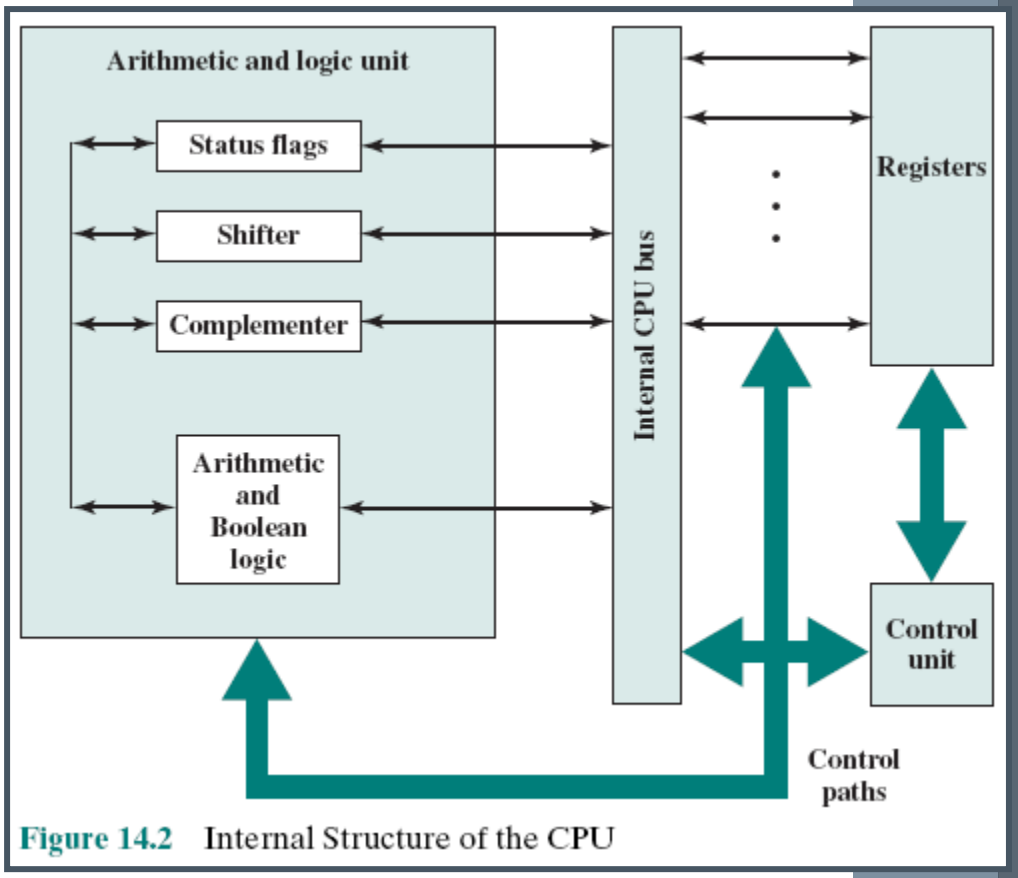



Figure 14.2 Internal Structure of the CPU

# 14.2- Register Organization

Within the processor there is a set of registers that function as a level of memory above main memory and cache in the hierarchy

The registers in the processor perform two roles:

USER-VISIBLE REGISTERS

- Enable the machine or assembly language programmer to minimize main memory references by optimizing use of registers

CONTROL AND STATUS REGISTERS

- Used by the control unit to control the operation of the processor and by privileged operating system programs to control the execution of programs

# User-Visible Registers

Referenced by means of the machine language that the processor executes

Categories:

- **General purpose**
  - Can be assigned to a variety of functions by the programmer
- **Data**
  - May be used only to hold data and cannot be employed in the calculation of an operand address
- **Address**
  - May be somewhat general purpose or may be devoted to a particular addressing mode
  - Examples:  segment pointers, index registers, stack pointer
- **Condition codes**
  - Also referred to as *flags*
  - Bits set by the processor hardware as the result of operations

# Table 14.1: Condition Codes

**Table 14.1** Condition Codes

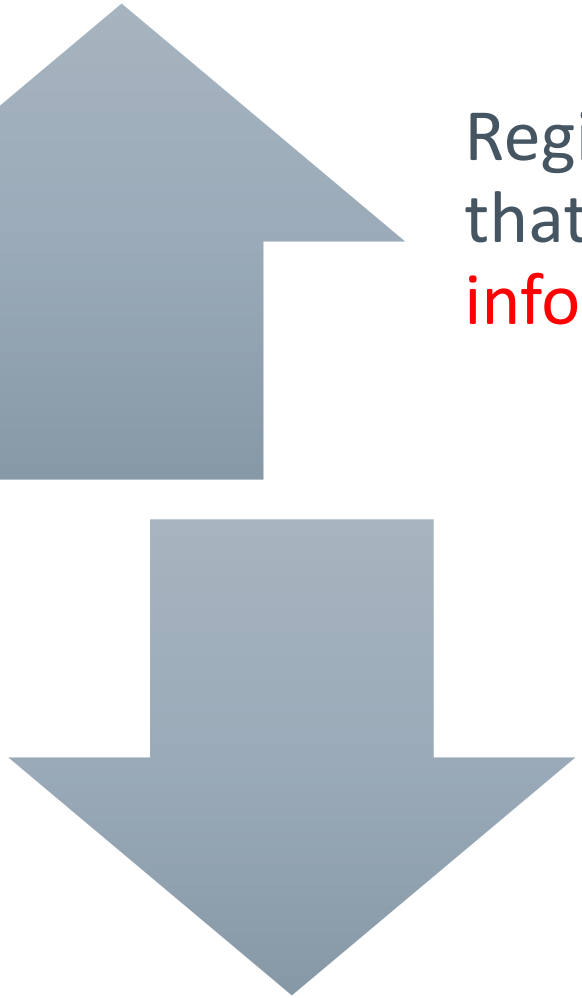
| Advantages | Disadvantages |
|---|---|
| 1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed.<br>2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST AND BRANCH.<br>3. Condition codes facilitate multiway branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero.<br>4. Condition codes can be saved on the stack during subroutine calls along with other register information. | 1. Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the microprogrammer and compiler writer.<br>2. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections.<br>3. Often condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control, and atomic semaphore operations.<br>4. In a pipelined implementation, condition codes require special synchronization to avoid conflicts. |

# Control and Status Registers

Four registers are essential to instruction execution:

- **Program counter (PC)**
  - Contains the address of an instruction to be fetched
- **Instruction register (IR)**
  - Contains the instruction most recently fetched
- **Memory address register (MAR)**
  - Contains the address of a location in memory
- **Memory buffer register (MBR)**
  - Contains a word of data to be written to memory or **the word most recently read**

# Program Status Word (PSW)

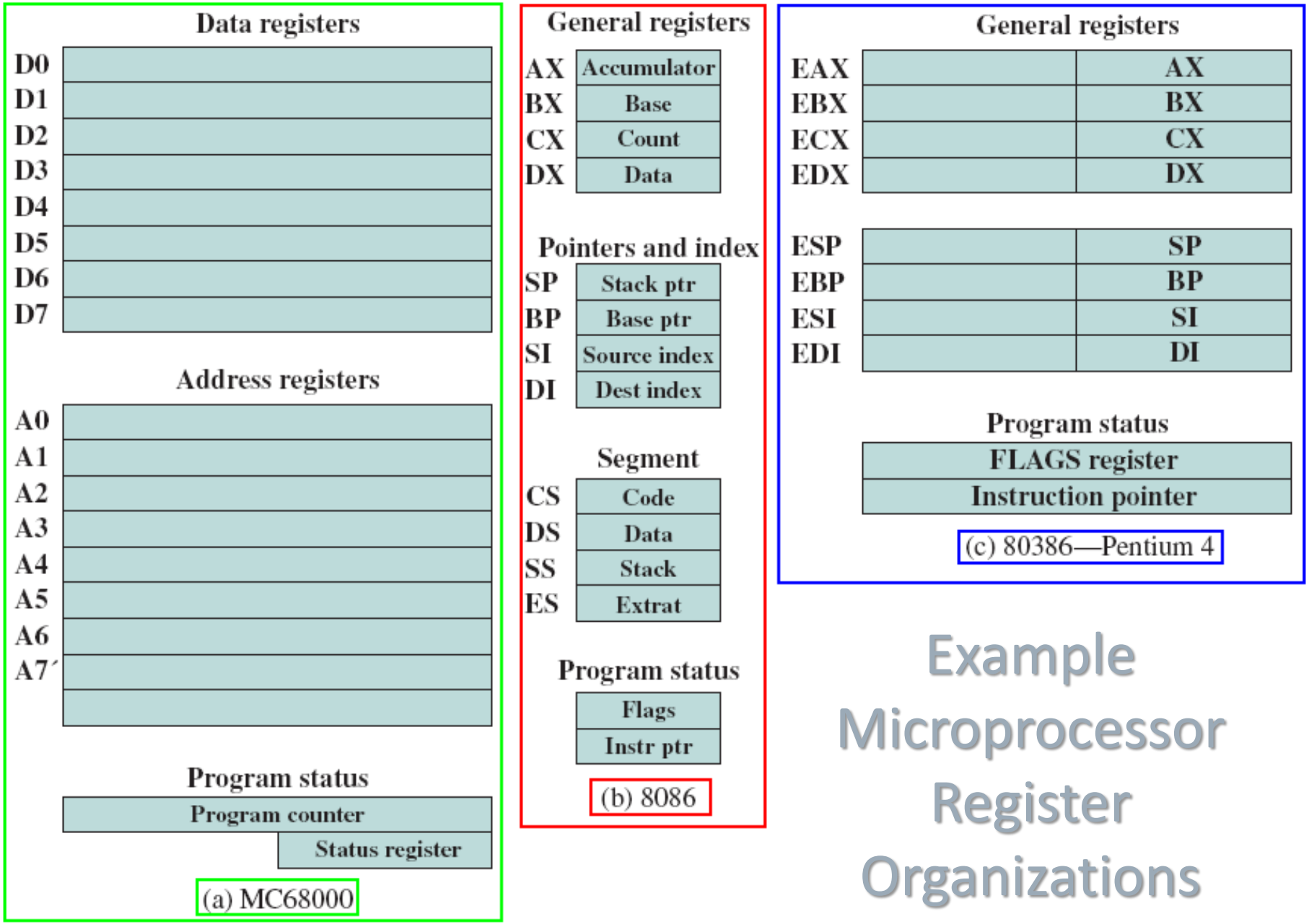


Register or set of registers that contain status information

Common fields or flags include:

- Sign
- Zero
- Carry
- Equal
- Overflow
- Interrupt Enable/Disable
- Supervisor

Status information are used to give a decision for branching




Example Microprocessor Register Organizations

**Figure 14.3** Example Microprocessor Register Organizations

# 14.3- Instruction Cycle

Includes the following stages:

**Fetch**

Read the **next instruction** from memory into the processor

**Execute**

Interpret the **opcode** and **perform** the indicated operation

**Interrupt**

If interrupts are enabled and an interrupt has occurred, **save the current process state and service the interrupt**

# Instruction Cycle

Loop due to additional memory accesses

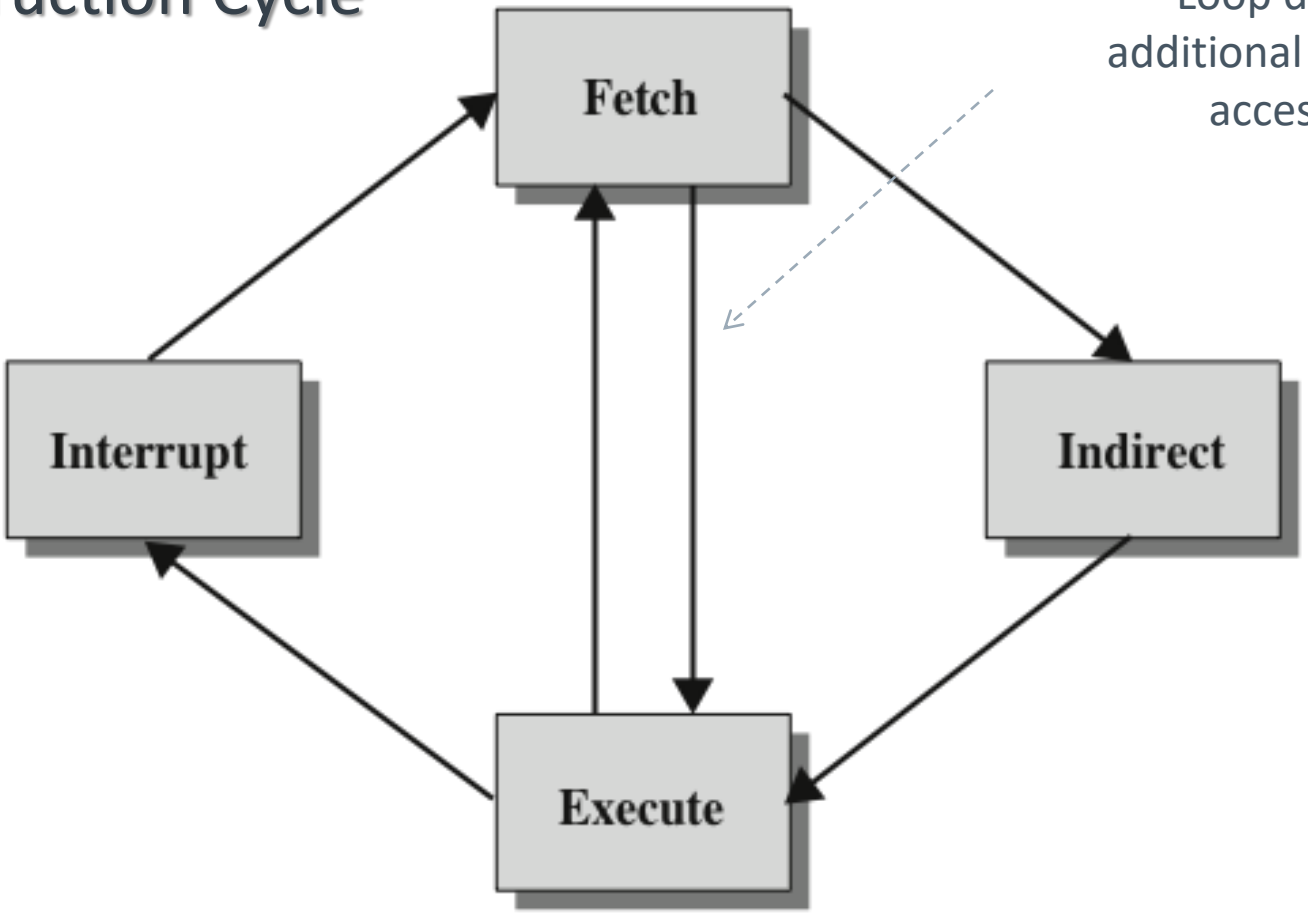



Figure 14.4 The Instruction Cycle

# Instruction Cycle State Diagram

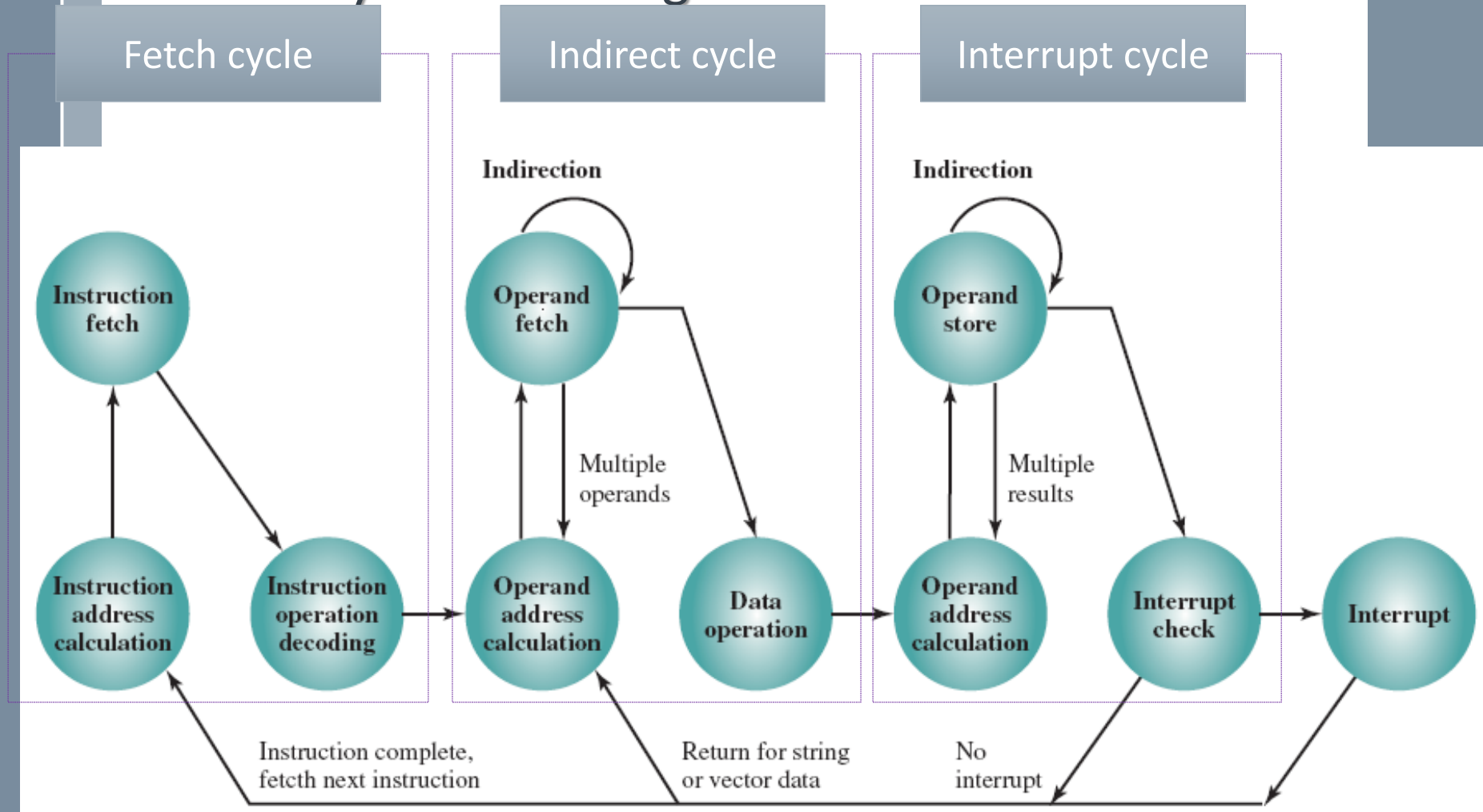



**Figure 14.5** Instruction Cycle State Diagram

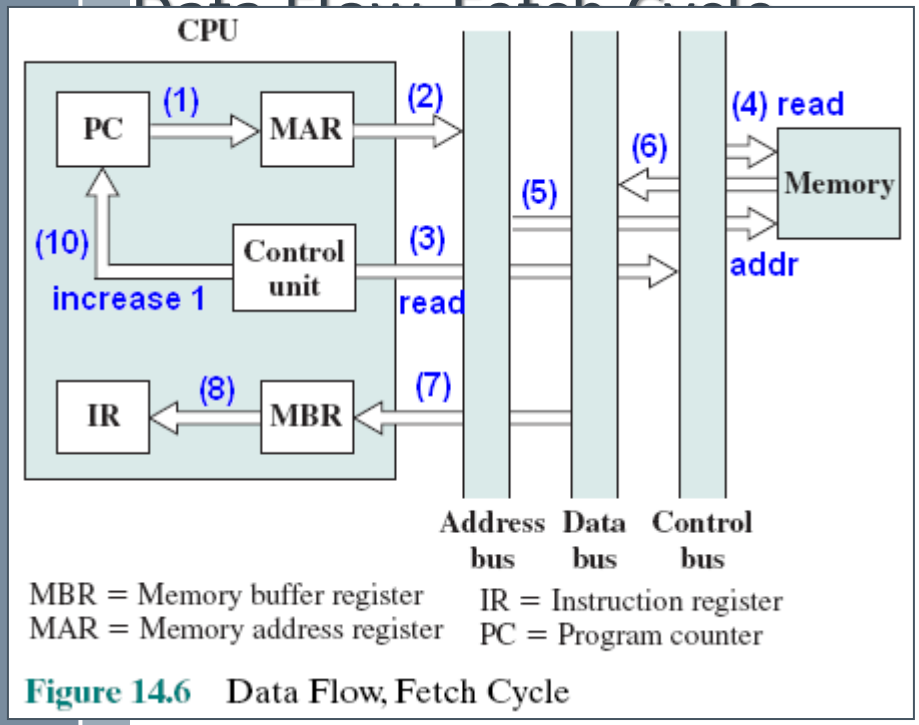



Figure 14.6 Data Flow, Fetch Cycle

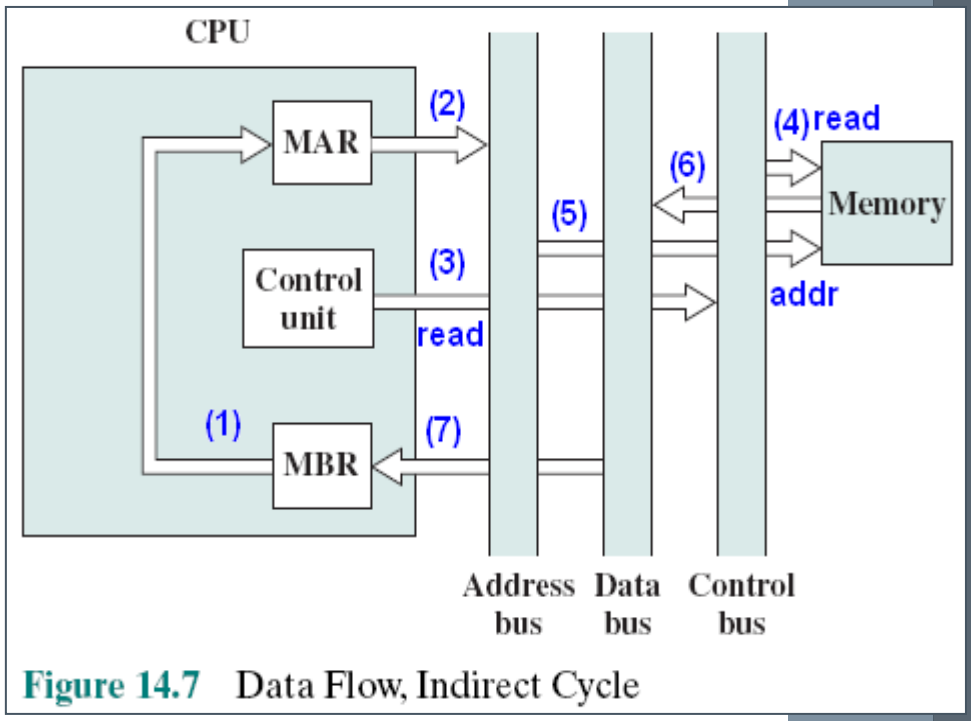



Figure 14.7 Data Flow, Indirect Cycle

Fetch cycle for the next instruction
(Instruction index is in PC)

MAR: Memory Address Register

MBR: Memory buffer Register

The CU examines the contents of the IR to determine if it contains an operand specified by indirect addressing→ Use indirect cycle(data address is in MBR)

# Data Flow: Interrupt Cycle

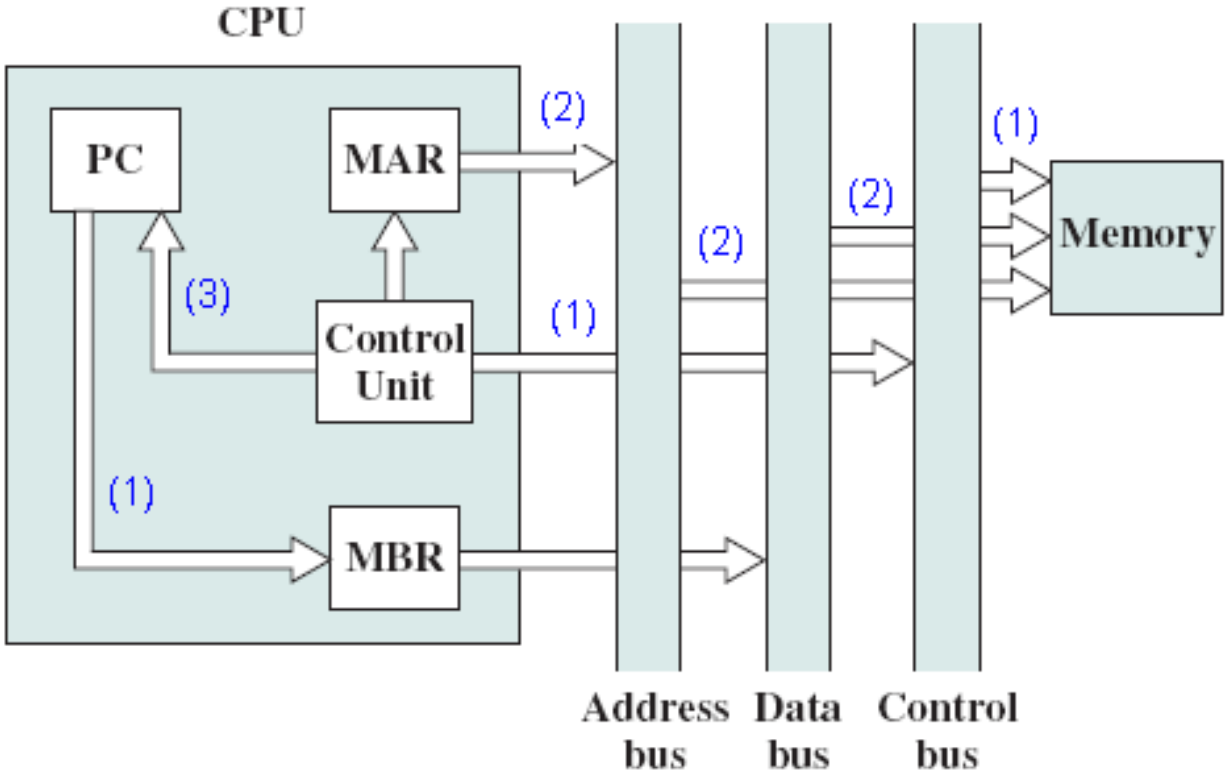



**Figure 14.8** Data Flow, Interrupt Cycle

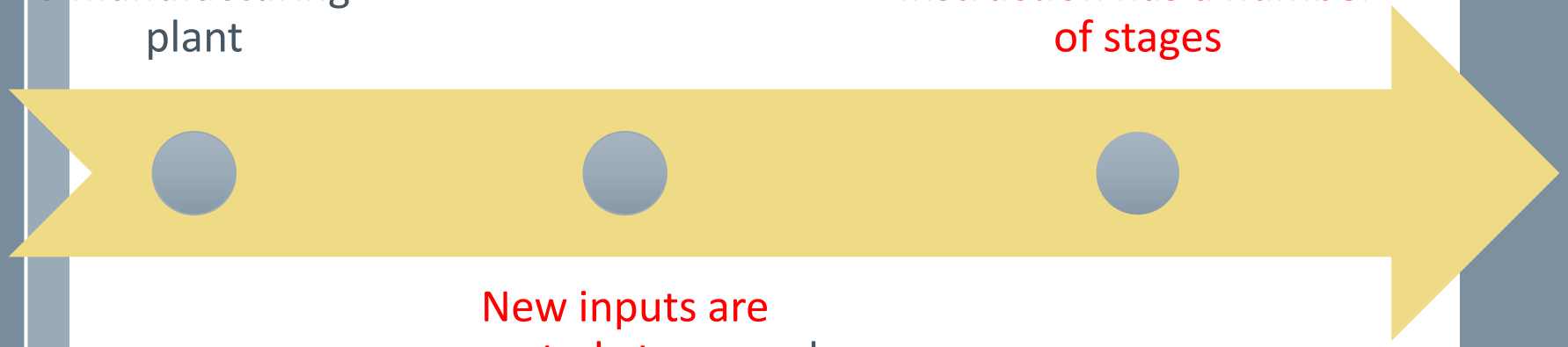
(1) Store PC (return point after executing interrupt routine)
(2) Store current state (values in registers before running interrupt routine)
(3) Fetch cycle is used to load interrupt routine

# 14.4- Instruction Pipeling
# Pipelining Strategy

A way to improve performance is performing jobs in parallel manner

Similar to the use of an assembly line in a manufacturing plant

To apply this concept to instruction execution we must recognize that an instruction has a number of stages



New inputs are accepted at one end before previously accepted inputs appear as outputs at the other end

An assembly line (dây chuyền xử lý) in which some operations are performed concurrently

# Two-Stage Instruction Pipeline

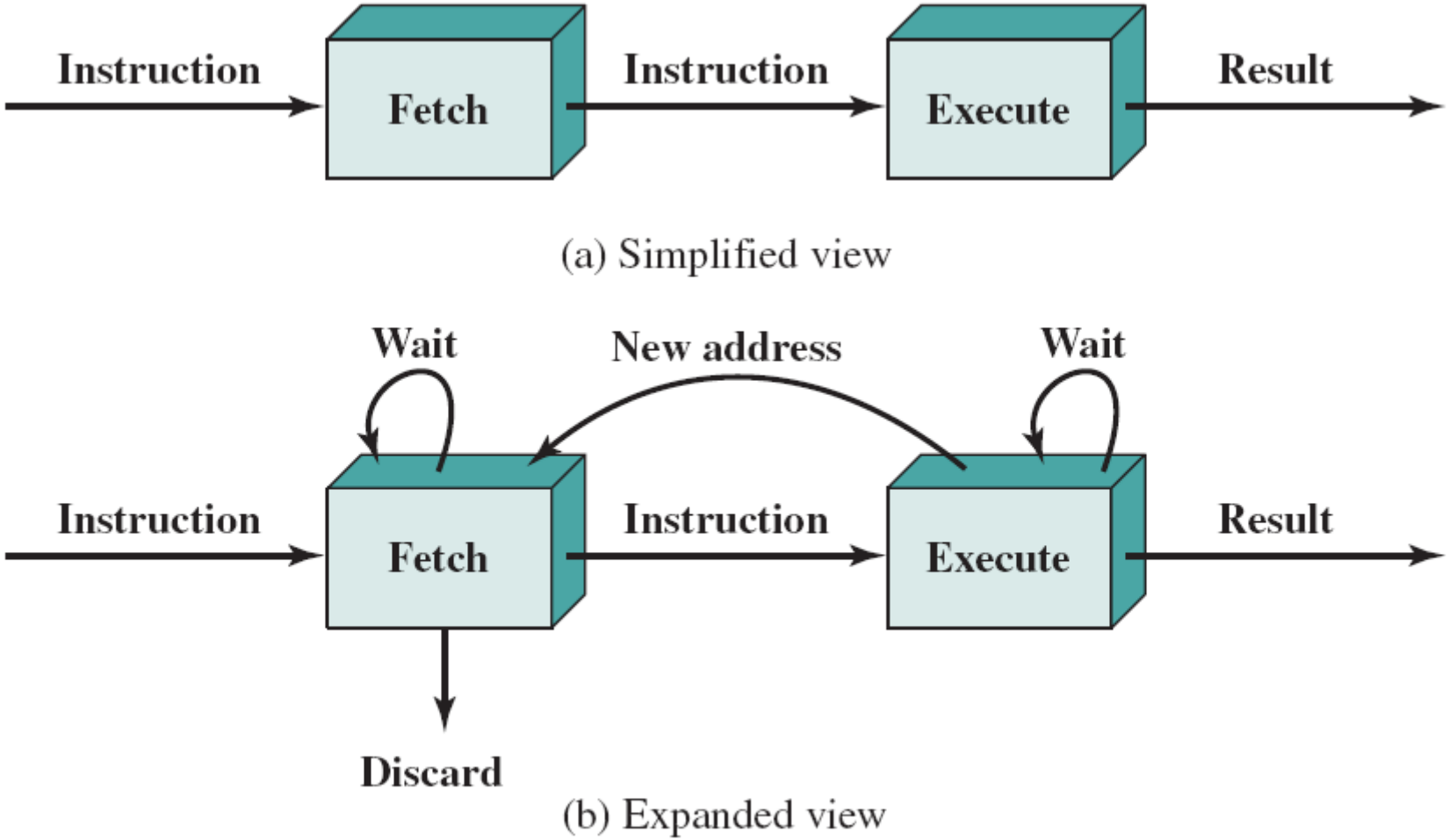
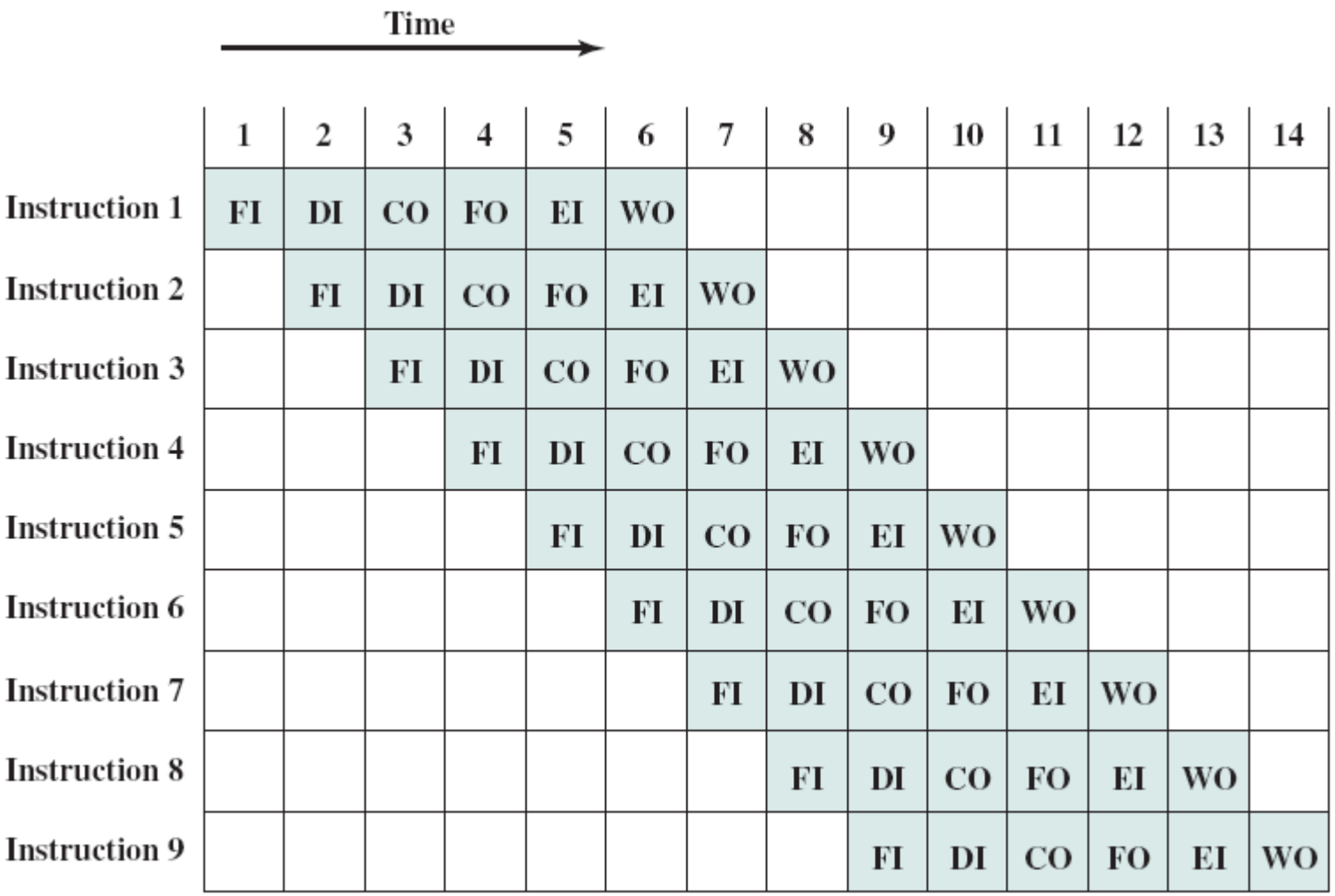



**Figure 14.9** Two-Stage Instruction Pipeline

# Additional 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
  - This may involve displacement, register indirect, indirect, or other forms of address calculation
- **Fetch operands (FO)**
  - Fetch each operand from memory
  - Operands in registers need not be fetched
- **Execute instruction (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

# Timing Diagram for Instruction Pipeline Operation




**I**: Instruction
**O**: operand
**F**: Fetch
**C**: Calculate
**F**: Fetch
**E**: Execute
**W**: Write

**Figure 14.10** Timing Diagram for Instruction Pipeline Operation

# The Effect of a Conditional Branch on Instruction Pipeline Operation

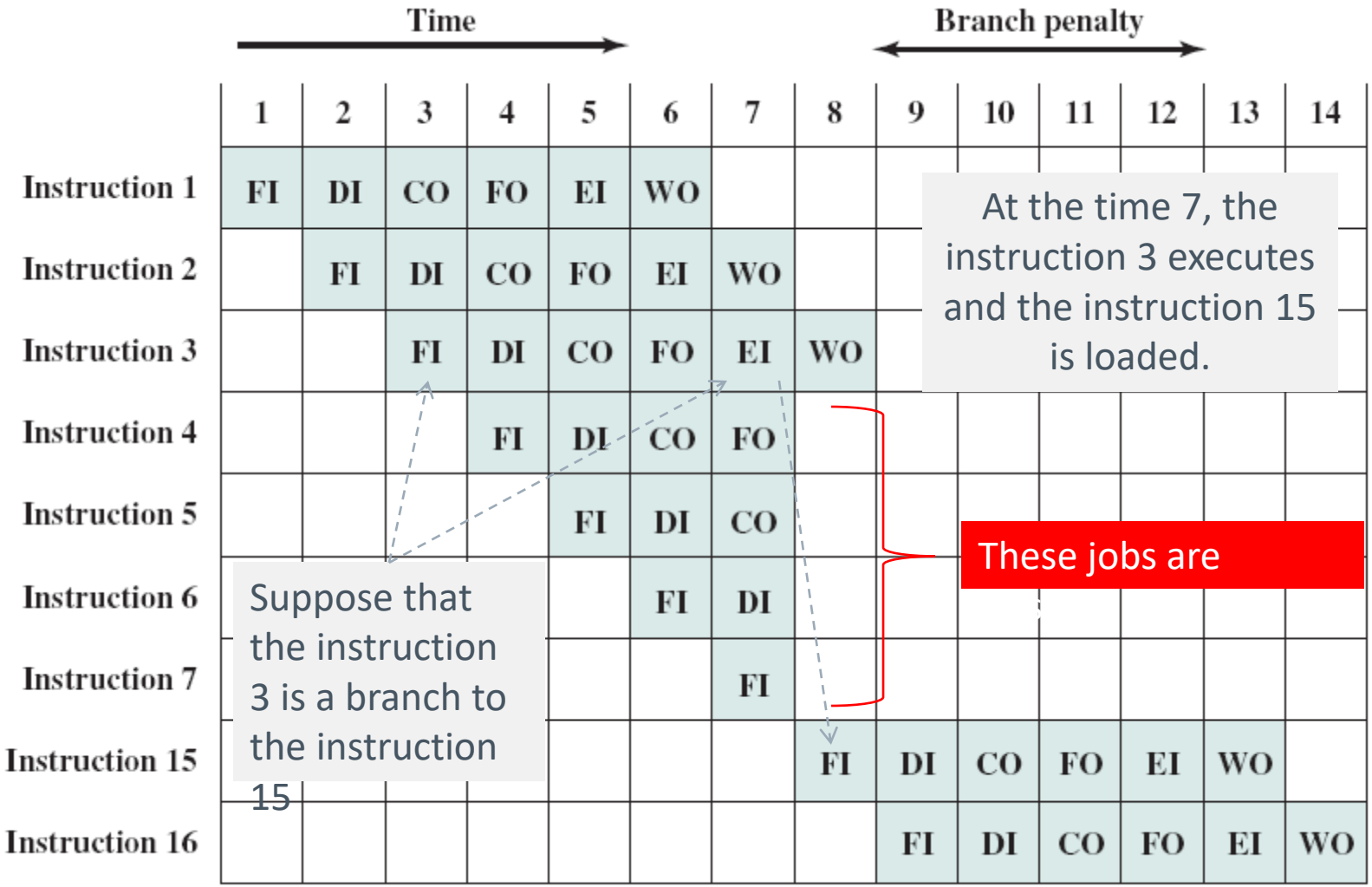



**Figure 14.11** The Effect of a Conditional Branch on Instruction Pipeline Operation

# SIX STAGE INSTRUCTION PIPELINE

Figure 14.12 indicates the logic needed for pipelining to account for branches and interrupts

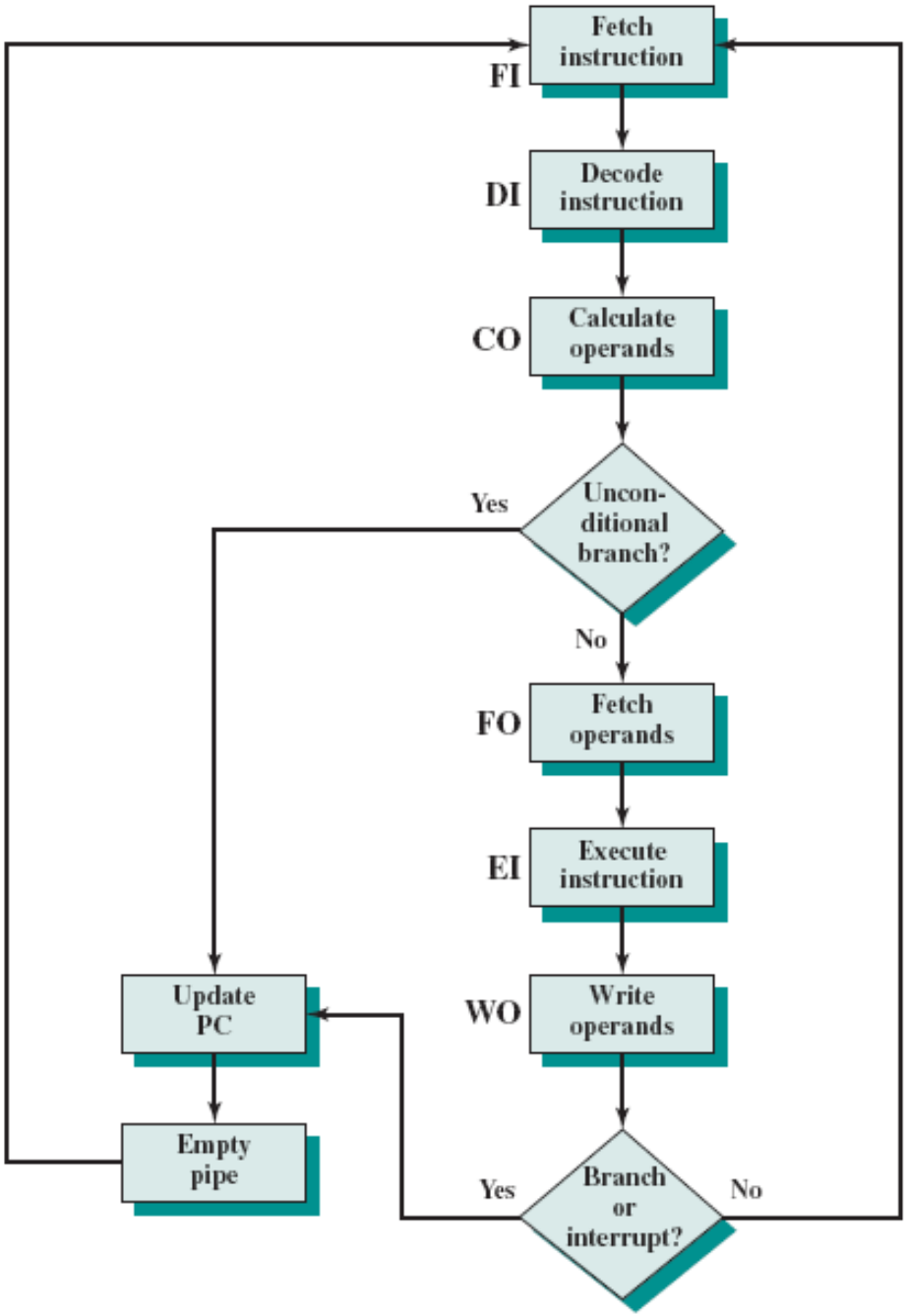
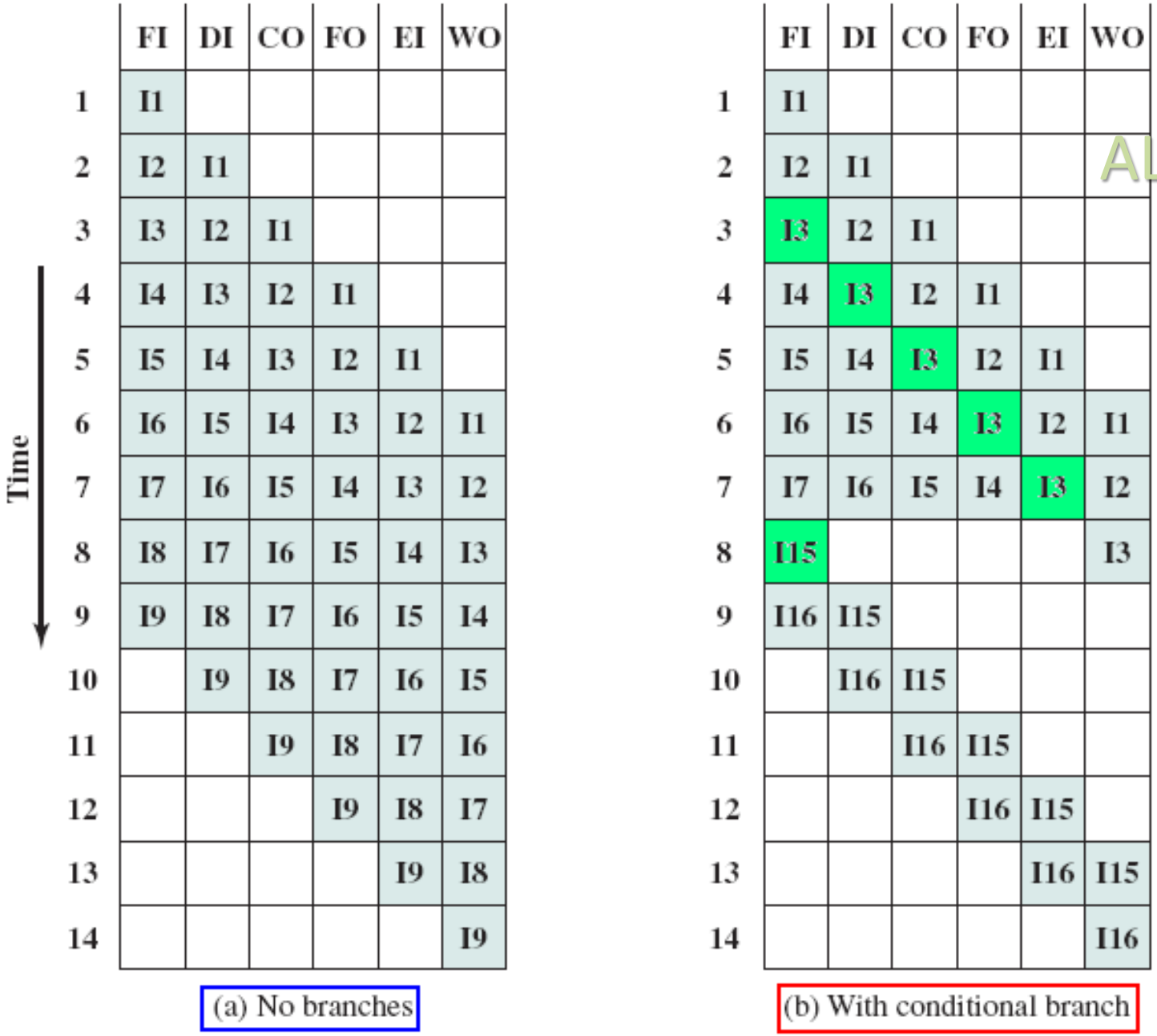



**Figure 14.12** Six-Stage CPU Instruction Pipeline

# ALTERNATIVE PIPELINE DEPICTION




**I3** is a conditional branch to **I15**

**Figure 14.13** An Alternative Pipeline Depiction

# SPEEDUP FACTORS WITH INSTRUCTION PIPELINING

The larger the number of pipeline stages, the greater the potential for speedup ➔ **higher COST**

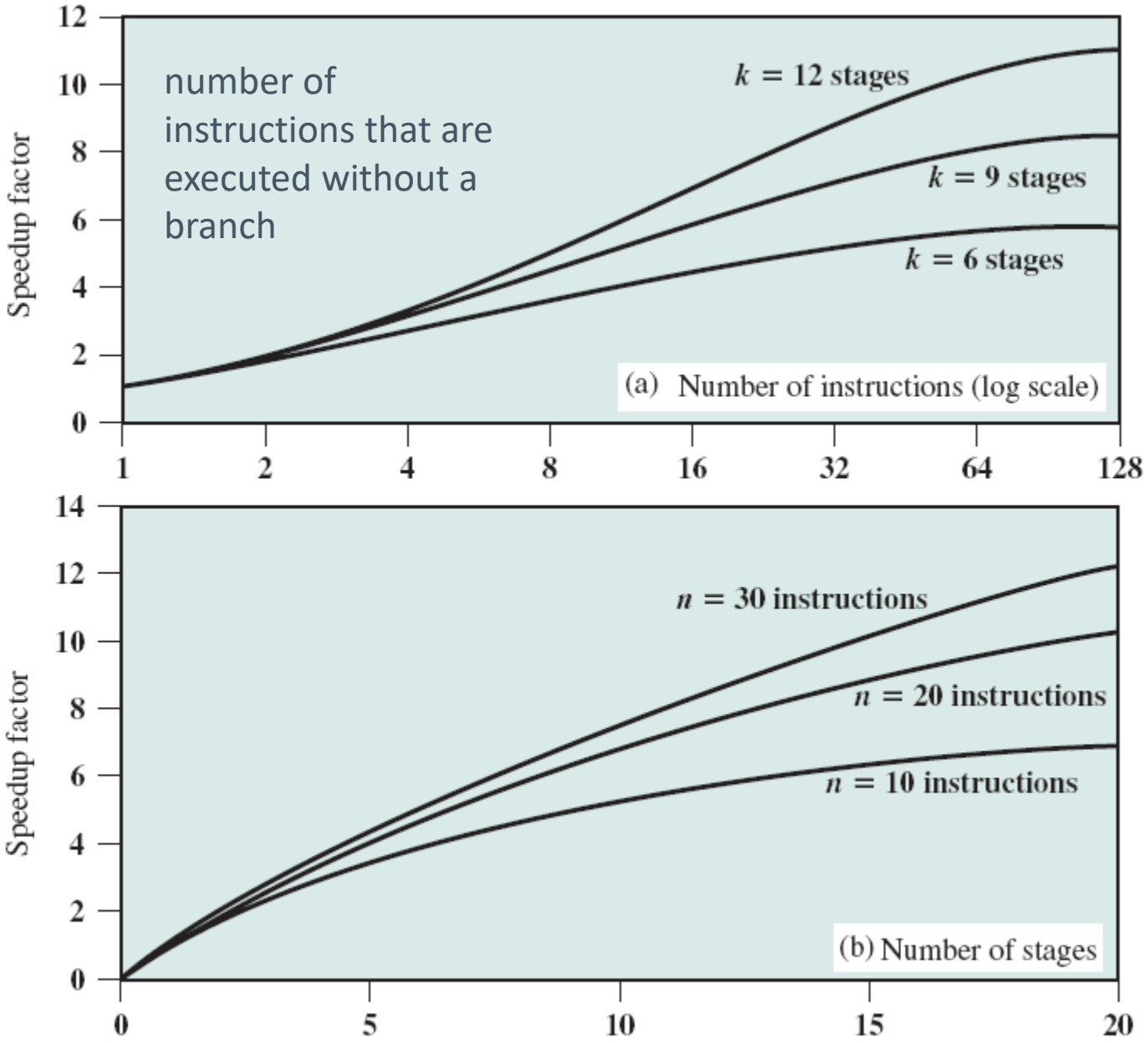



**Figure 14.14** Speedup Factors with Instruction Pipelining

# Pipeline Hazards (rủi ro)

Occur when the pipeline, or **some portion** of the pipeline, must **stall** (trì hoãn) because conditions do not permit continued execution

There are three **types** of hazards:

- **Resource**
- **Data**
- **Control**

Also referred to as a ***pipeline bubble***

# RESOURCE HAZARDS

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***

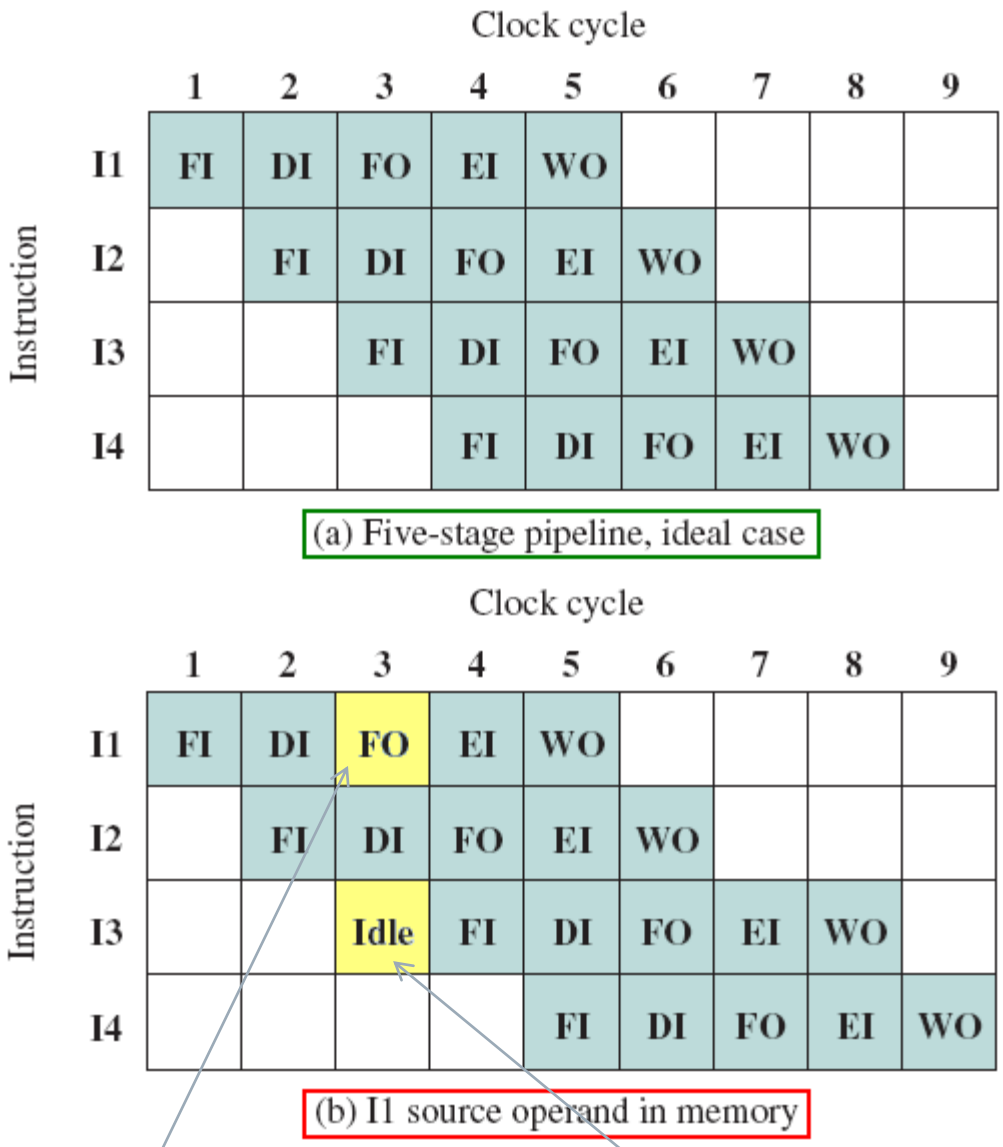



**Figure 14.15** Example of Resource Hazard

**FO is accessing memory. So, this step is idle**

# DATA HAZARDS

A data hazard occurs when there is a conflict in the access of an operand location

**RAW**

Instruction is executing and the register EAX is writing to. So, it can not be read.

Hazard

X86 instruction

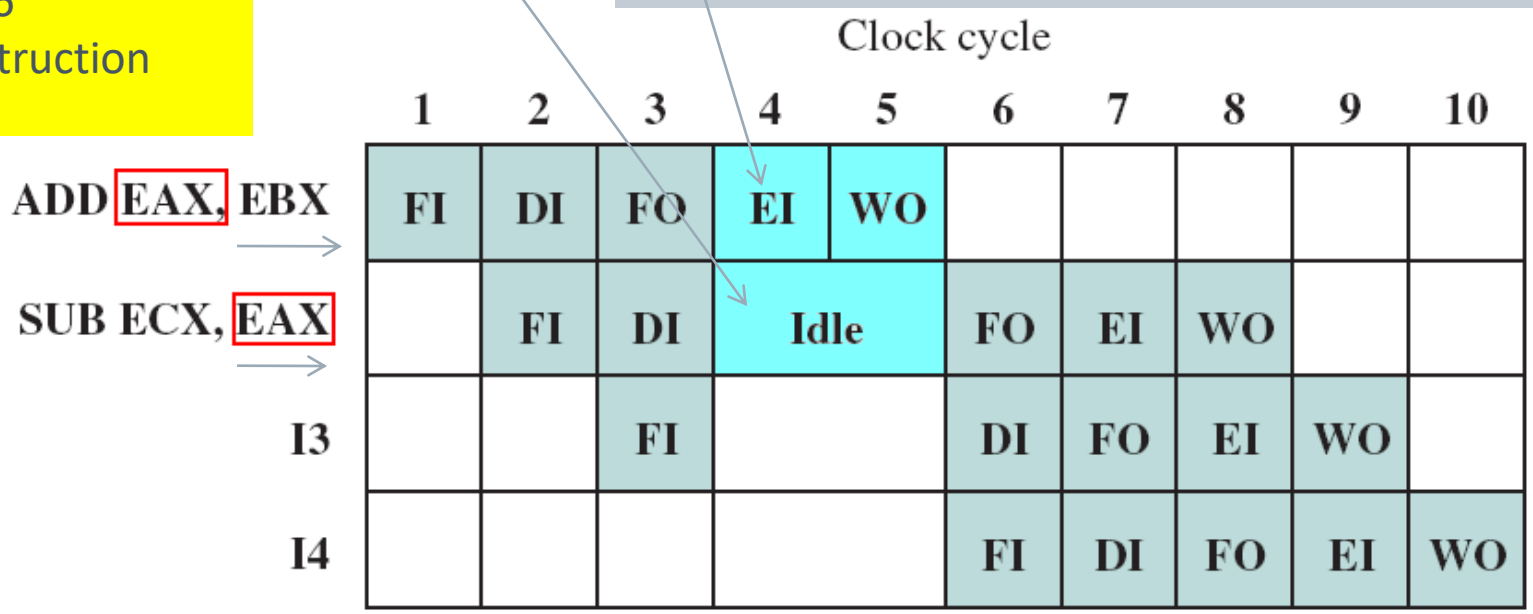



**Figure 14.16** Example of Data Hazard

# 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

# Control Hazard

- Also known as a ***branch hazard***
- Occurs when the pipeline makes the wrong decision on a branch prediction
- Brings instructions into the pipeline that must subsequently be discarded
- **Dealing with Branches**:
  - **Multiple streams**
  - **Prefetch branch target**
  - **Loop buffer**
  - **Branch prediction**
  - **Delayed branch**

# Multiple Streams

A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice

A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams

brute-force search or exhaustive search (vét cạn)

Drawbacks:

- With multiple pipelines there are contention delays for access to the registers and to memory
- Additional branch instructions may enter the pipeline before the original branch decision is resolved

# 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

# Loop Buffer

Small, very-high speed memory maintained by the instruction fetch stage of the pipeline and containing the *n* most recently fetched instructions, in sequence

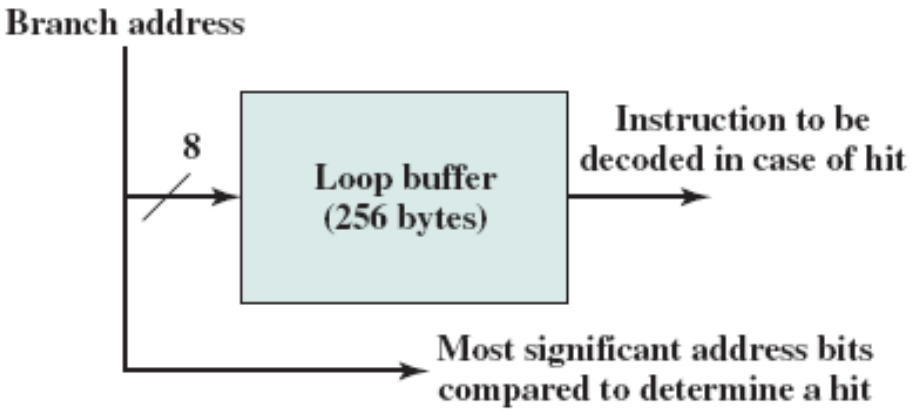



**Figure 14.17** Loop Buffer

- **Benefits**:
  - Instructions fetched in sequence will be available without 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
  - This strategy is particularly well suited to dealing with loops

**Similar** in principle **to a cache** dedicated to instructions.

Differences:

- The loop buffer only retains instructions in sequence
- Is much smaller in size and hence lower in cost

# Branch Prediction

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

1. Predict never taken
2. Predict always taken
3. Predict by opcode

- **These approaches are static**
- They do not depend on the execution history up to the time of the conditional branch instruction

1. Taken/not taken switch
2. Branch history table

- **These approaches are dynamic**
- They depend on the **execution history**

**How are predictions carried out?**

**Next slide**

➔ States of some last instructions (some bits) must be stores in cache

# BRANCH PREDICTION FLOW CHART

**If only one bit is stored, a loop may cause 2 errors in prediction: once on entering and once on exiting.**

If 2 bits are stored, a prediction algorithm is carried out using 2 branches (fig. 14.18)

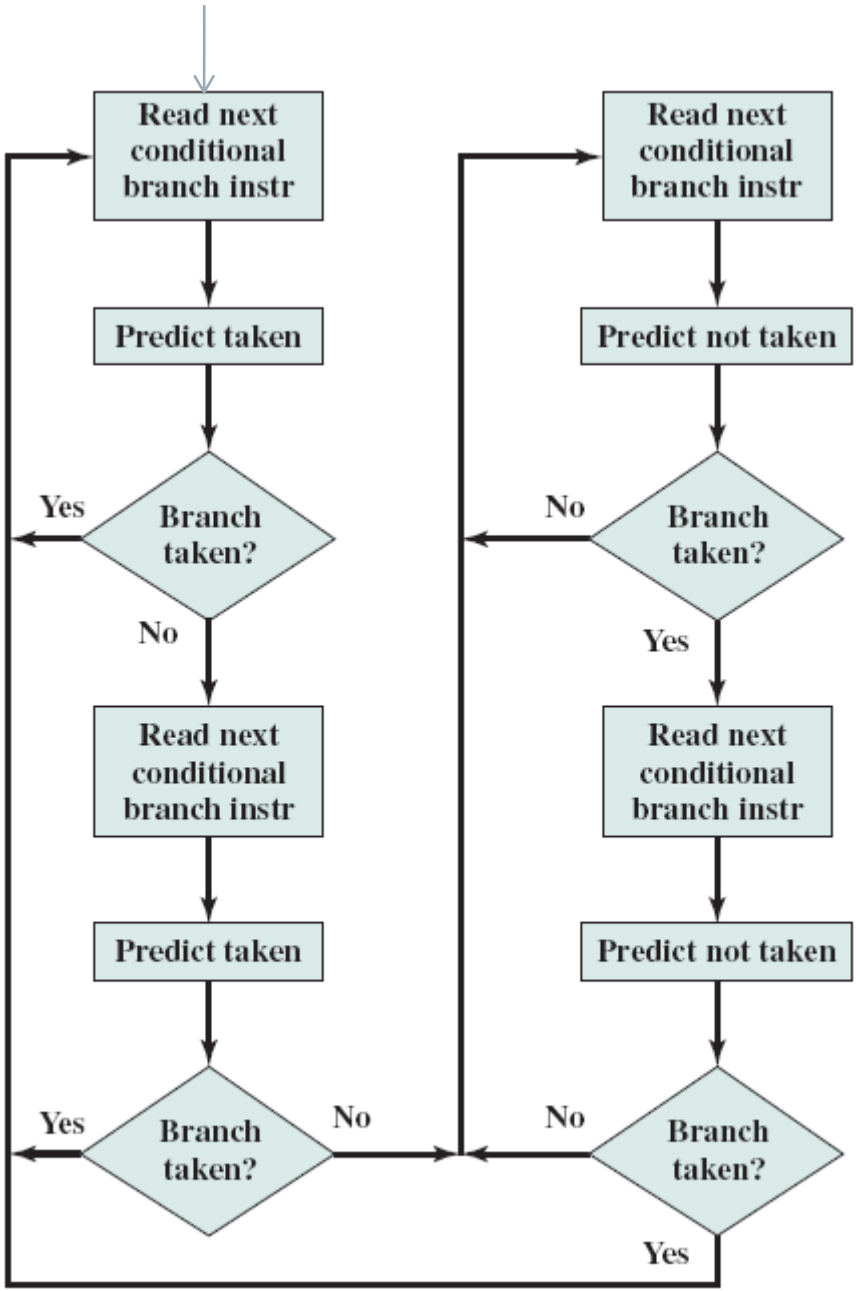



**Figure 14.18** Branch Prediction Flowchart

# Branch Prediction State Diagram

The decision process can be represented more compactly by a finite-state machine

Finite-state machine is a way to express a processing mechanism in which each part of input will determine a step of the process.

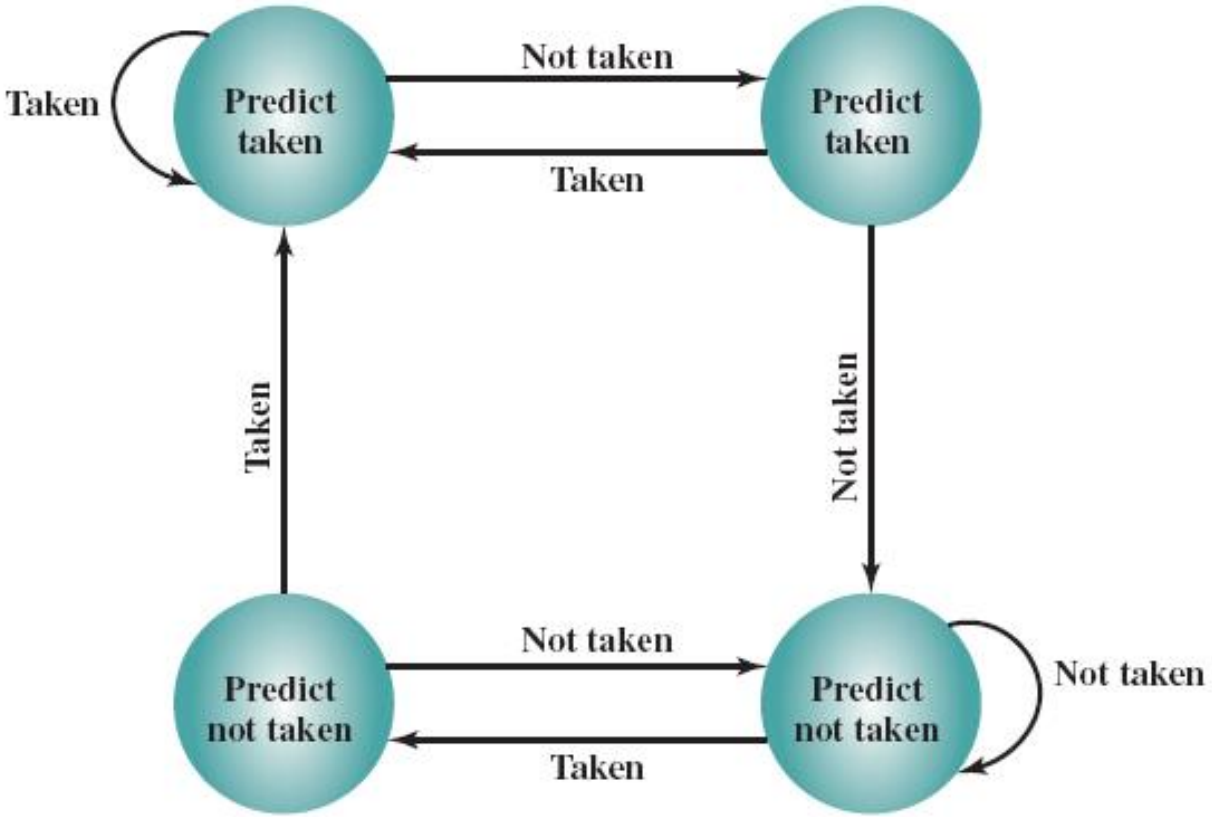



**Figure 14.19** Branch Prediction State Diagram

Some bits are stored: 0: Not taken, 1: Taken. A history can be as 01110

# DEALING WITH BRANCHES

Each prefetch triggers a lookup in the table.
**No match**: Fetch next sequential address.
**Match**: a prediction is made based on the state of the instruction: Either the next sequential address or the branch target address is fed to the select logic.

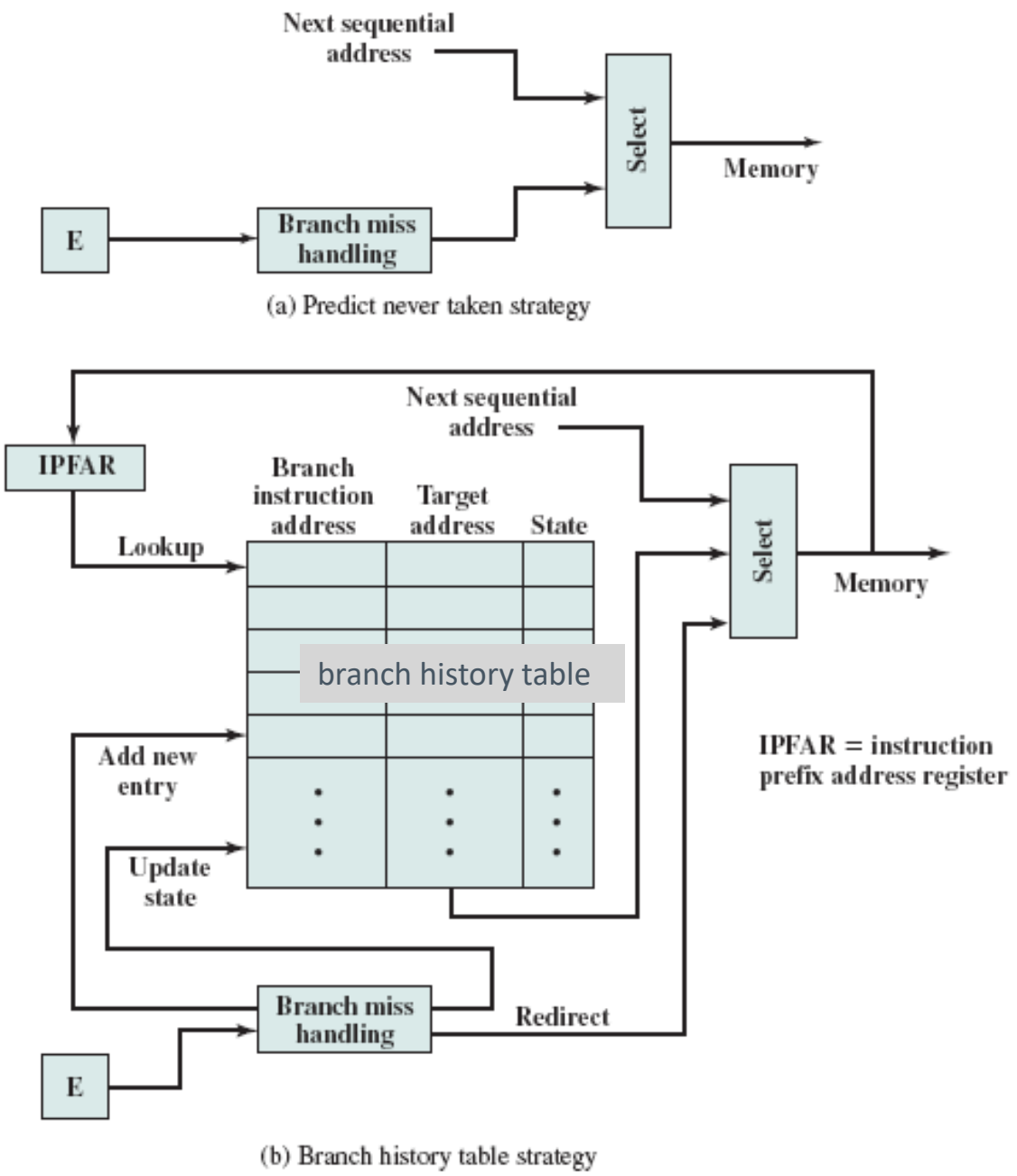



Figure 14.20 Dealing with Branches

# Delayed Branch

- It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired. This intriguing approach is examined in Chapter 15.

# Intel 80486 Pipelining

- **Fetch**
  - Objective is to **fill the prefetch buffers** with new data as soon as the old data have been consumed by the instruction decoder
  - **Operates independently** of the other stages to keep the prefetch buffers full
- **Decode stage 1**
  - **All opcode** and addressing-mode information is decoded in the D1 stage
  - **3 bytes** of instruction are passed to the D1 stage from **the prefetch buffers**
  - D1 decoder can then **direct the D2 stage** to capture the rest of the instruction
- **Decode stage 2**
  - Expands each opcode into control signals for the ALU
  - Also controls the computation of the more complex addressing modes
- **Execute**
  - Stage includes ALU operations, cache access, and register update
- **Write back**
  - **Updates registers** and **status flags** modified during the preceding execute stage

# 80486 INSTRUCTION PIPELINE EXAMPLES

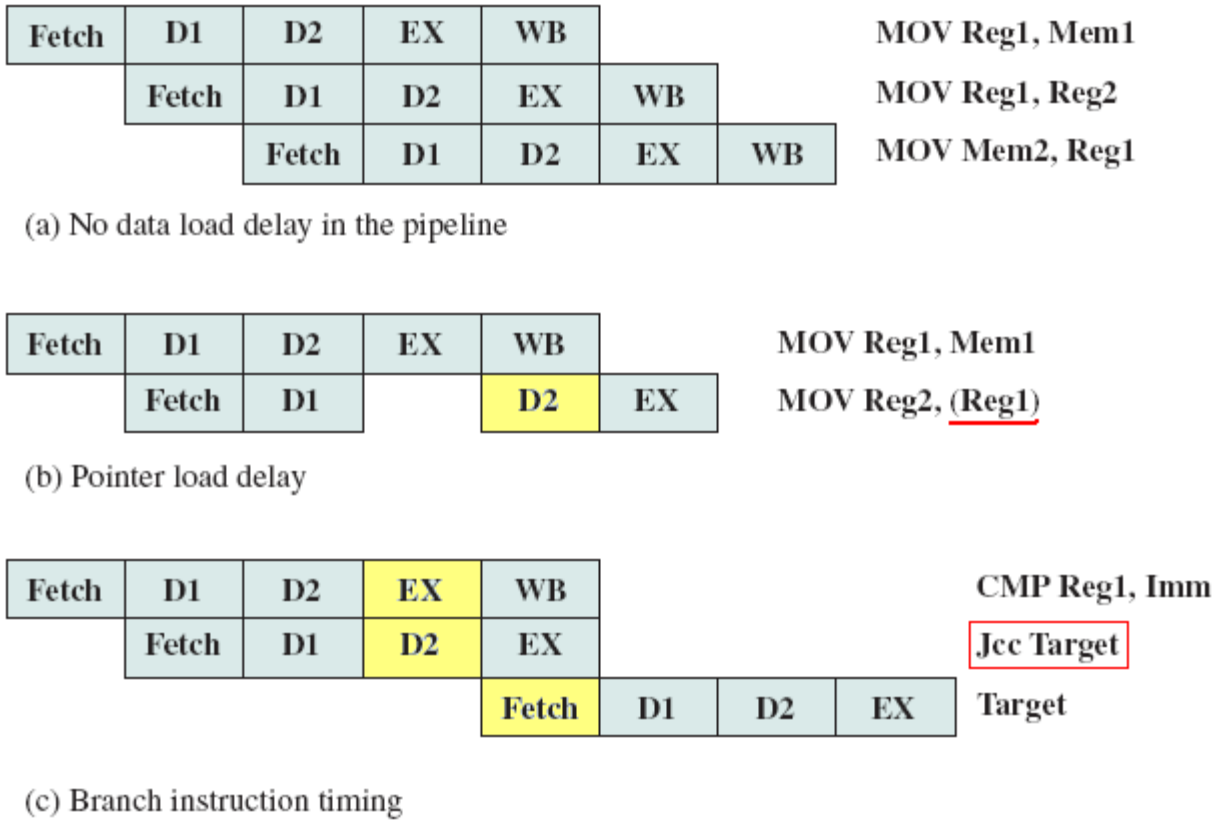



**Figure 14.21** 80486 Instruction Pipeline Examples