AAiT

# Computer Architecture & Organization

## Chapter 6 & 7

- Processor Structure and Function
- Pipelining

# Processor Organization

## Processor Requirements:

- Fetch instruction
  - The processor reads an instruction from memory (register, cache, main memory)
- Interpret instruction
  - The instruction is decoded to determine what action is required
- Fetch data
  - The execution of an instruction may require reading data from memory or an I/O module
- Process data
  - The execution of an instruction may require performing some arithmetic or logical operation on data
- Write data
  - The results of an execution may require writing data 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

Figure 14.1 is a simplified view of a processor, indicating its connection to the rest of the system via the system bus.

- Recall that the major components of the processor are an *arithmetic and logic unit* (ALU) and a *control unit* (CU).
- The ALU does the actual computation or processing of data.
- The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU.
- In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called *registers.*

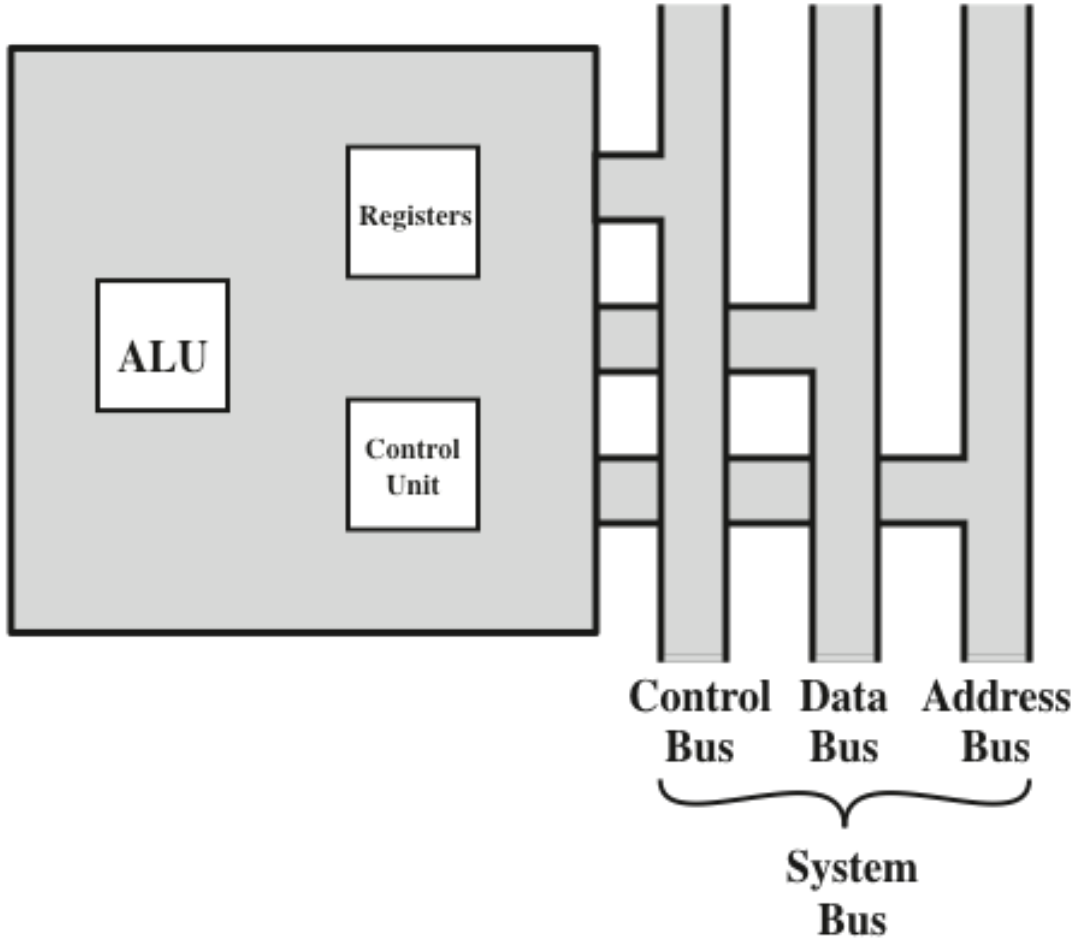



**Figure 14.1 The CPU with the System Bus**

# CPU Internal Structure

Figure 14.2 is a slightly more detailed view of the processor.

- The data transfer and logic control paths are indicated, including an element labeled *internal processor bus.* This element is needed to transfer data between the various registers and the ALU because the
- ALU operates only on data in the internal processor memory.
- Note the similarity between the internal structure of the computer as a whole and the internal structure of the processor. In both cases, there is a small collection of major elements (computer: processor, I/O, memory; processor: control unit, ALU, registers) connected by data paths.

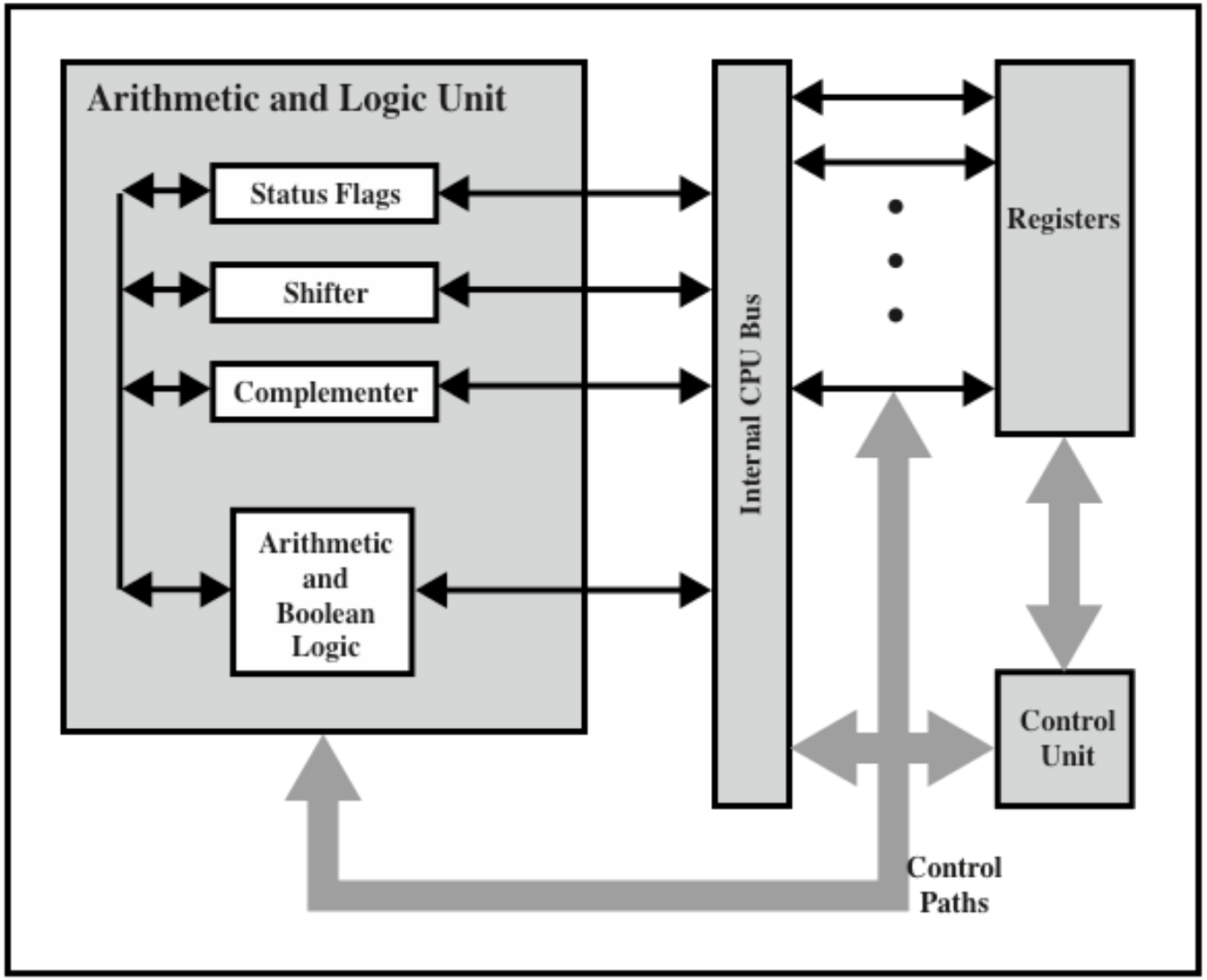



**Figure 14.2 Internal Structure of the CPU**

# 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

- Many processors, including those based on the IA-64 architecture and the MIPS processors, do not use condition codes at all. Rather, conditional branch instructions specify a comparison to be made and act on the result of the comparison.

| 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)

The *program status word* (PSW) is a register or set of registers that contain status information

Common fields or flags include:

- **Sign:** Contains the sign bit of the result of the last arithmetic operation.
- **Zero:** Set when the result is 0.
- **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multiword arithmetic operations.
- **Equal:** Set if a logical compare result is equality.
- **Overflow:** Used to indicate arithmetic overflow.
- **Interrupt Enable/Disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicates whether the processor is executing in supervisor or user mode. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

# Example Microprocessor Register Organizations

- Register organization of two 16-bit microprocessors that were designed at about the same time: the Motorola MC68000 and the Intel 8086 (Figures 14.3a and b).
- The MC68000 partitions its 32-bit registers into eight data registers and nine address registers allowing 8-, 16-, and 32-bit data operations, determined by opcode
- The MC68000 also includes a 32-bit program counter and a 16-bit status register.
- The 8086 contains four 16-bit data registers that are addressable on a byte or 16-bit basis, four 16-bit pointer and index registers, four 16-bit segment registers and also includes an instruction pointer and a set of 1-bit status and control flags
- The 80386 uses 32-bit registers and provide upward compatibility by retaining the original register organization embedded in the new organization

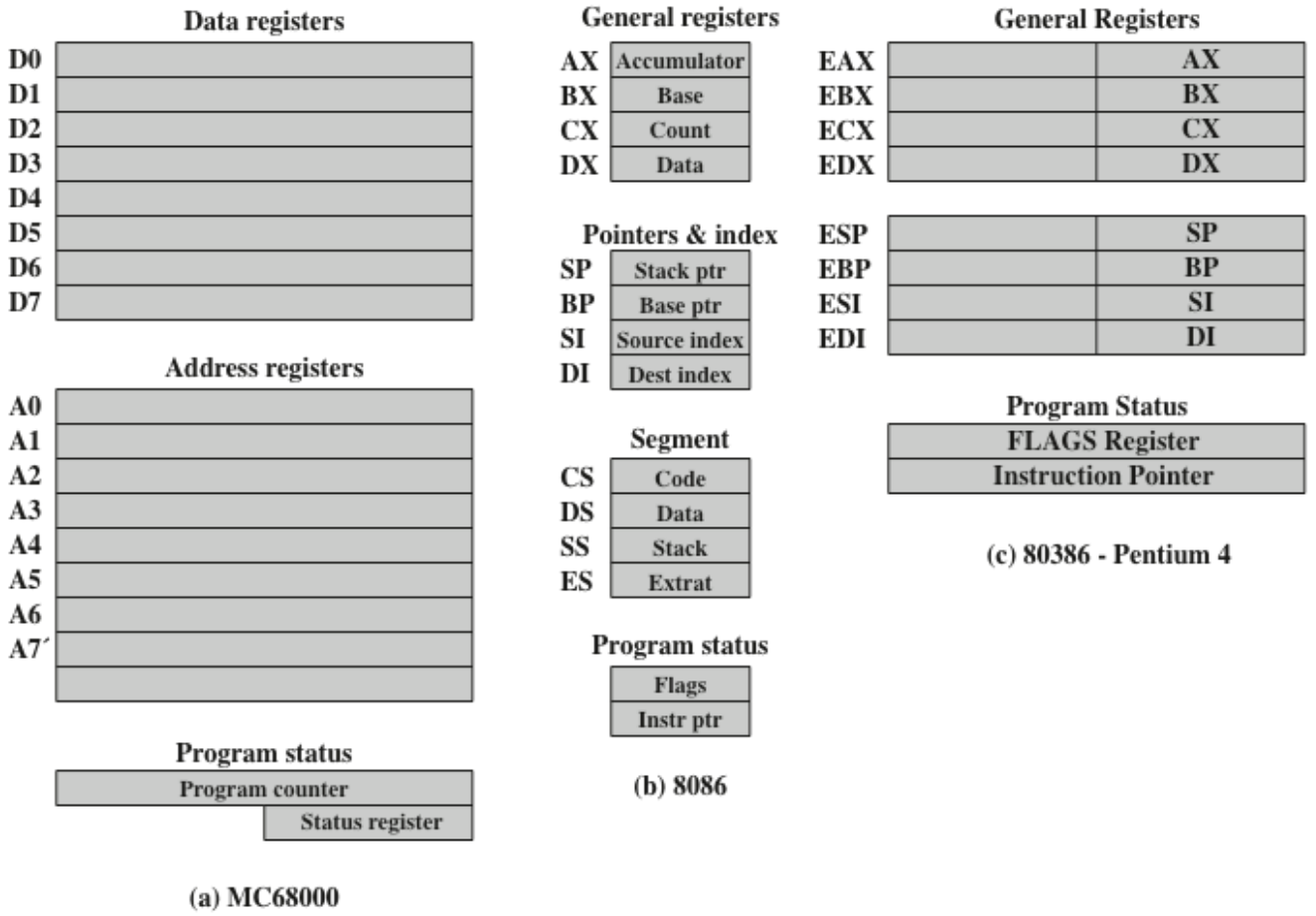



**Figure 14.3 Example Microprocessor Register Organizations**

Q How could 80386 be compatible with 8086 w/o segment registers? Why do you think 80386 doesn't need CS, DS, SS & ES?

# Instruction Cycle

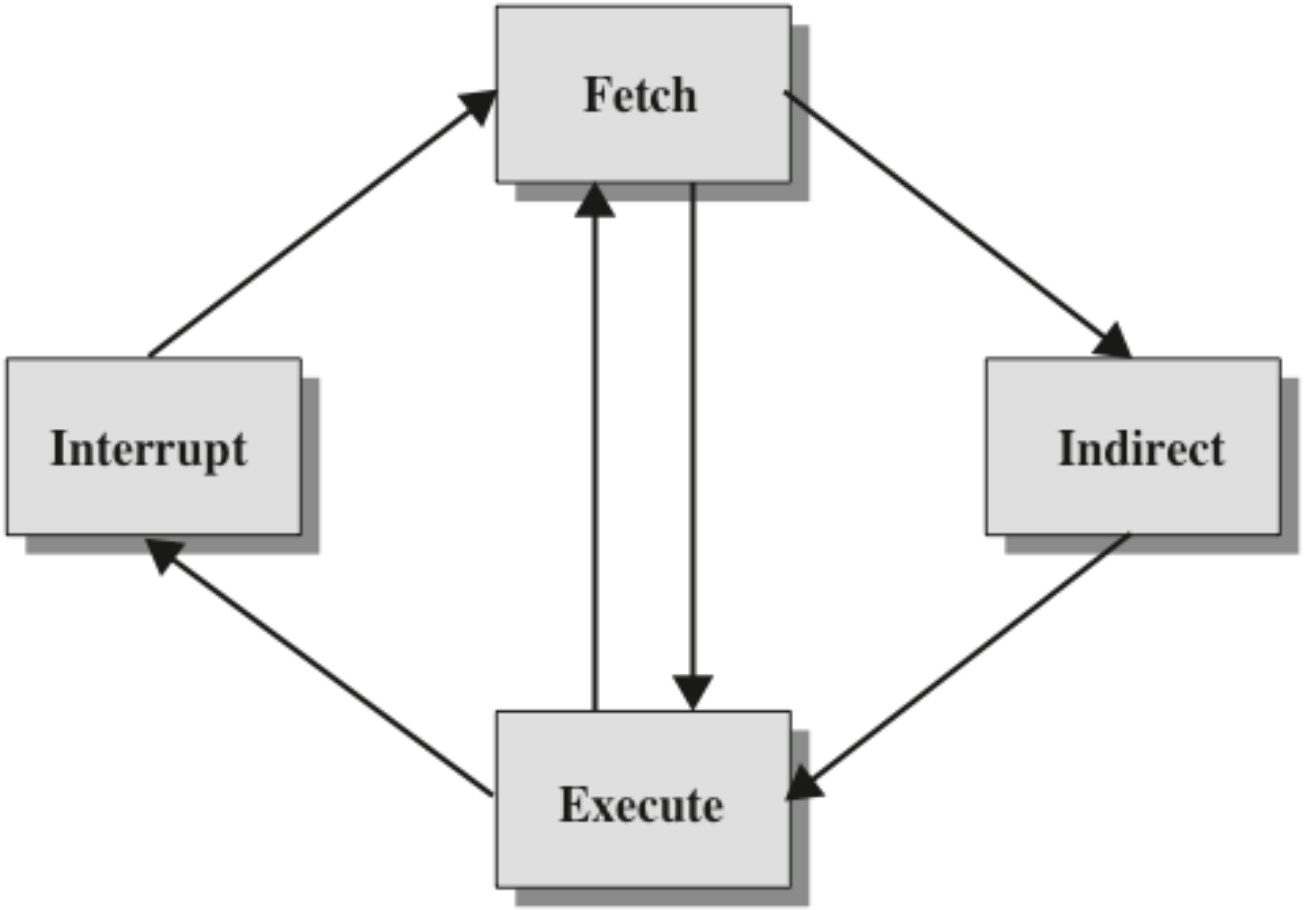
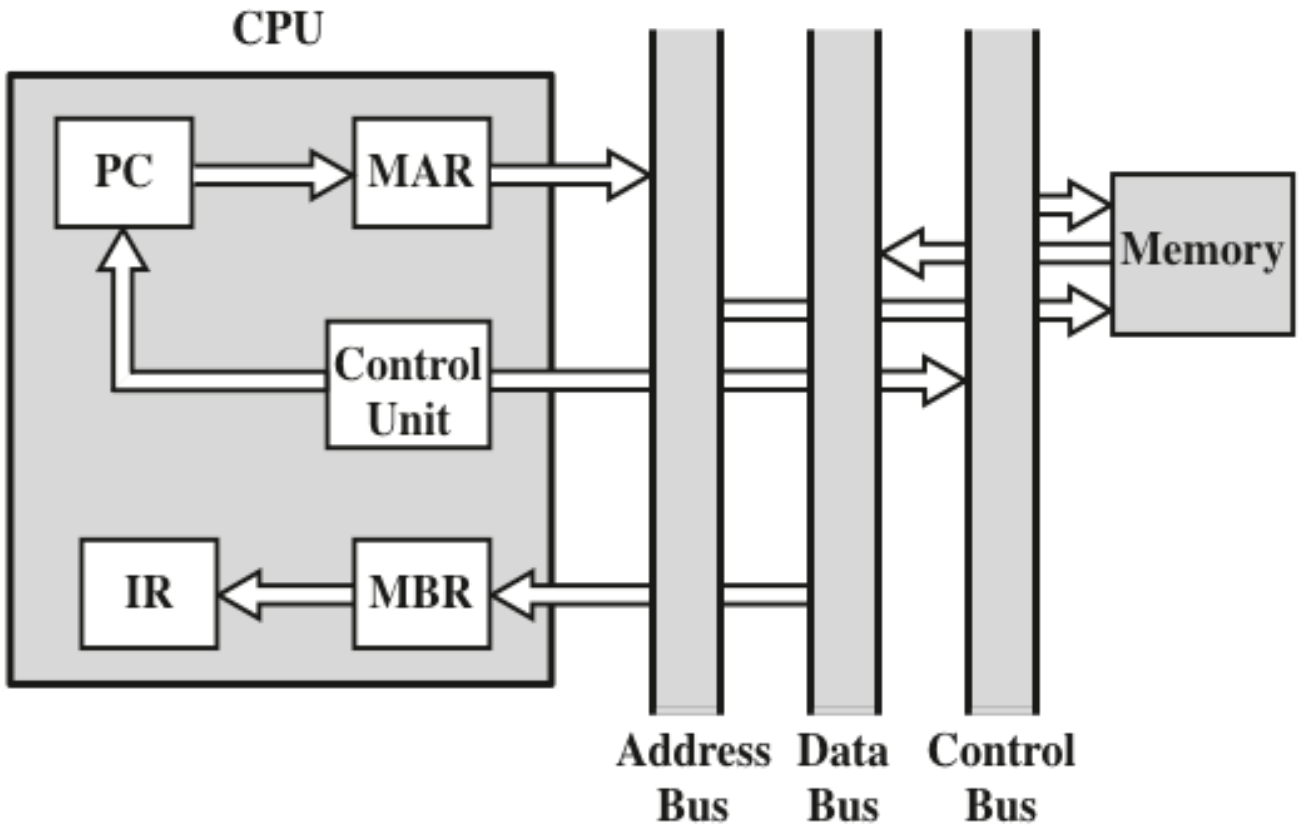



Figure 14.4 The Instruction Cycle

- After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing.
- Following execution, an interrupt may be processed before the next instruction fetch.

# Data Flow, Fetch Cycle




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

**Figure 14.6 Data Flow, Fetch Cycle**

# Data Flow, Indirect Cycle

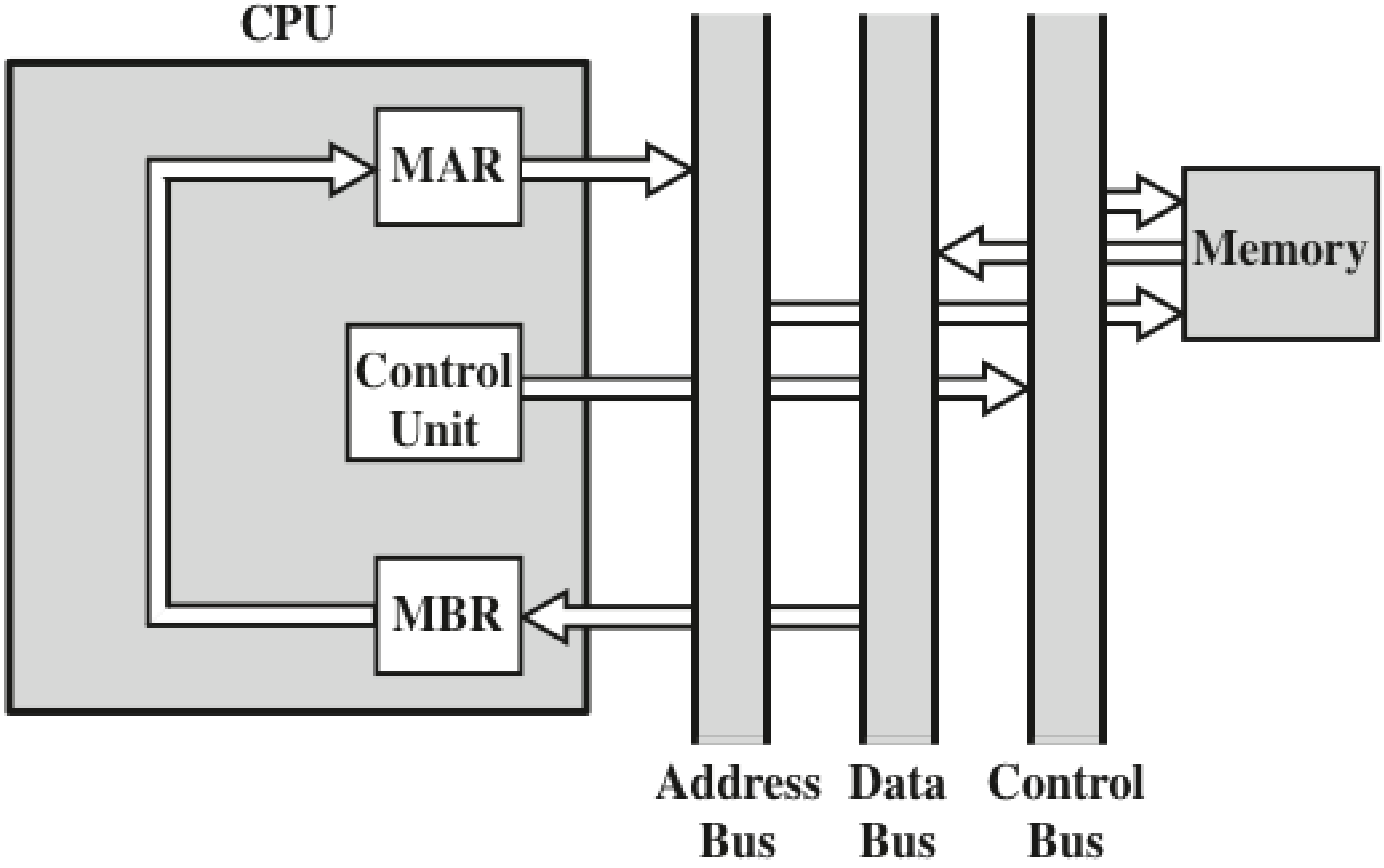



**Figure 14.7 Data Flow, Indirect Cycle**

- Once the fetch cycle is over, the control unit examines the contents of the IR to determine if it contains an operand specifier using indirect addressing.
- If so, an *indirect cycle* is performed. This is a simple cycle. The right- most *N* bits of the MBR, which contain the address reference, are transferred to the MAR.
- Then the control unit requests a memory read, to get the desired address of the operand into the MBR

# Data Flow, Interrupt Cycle

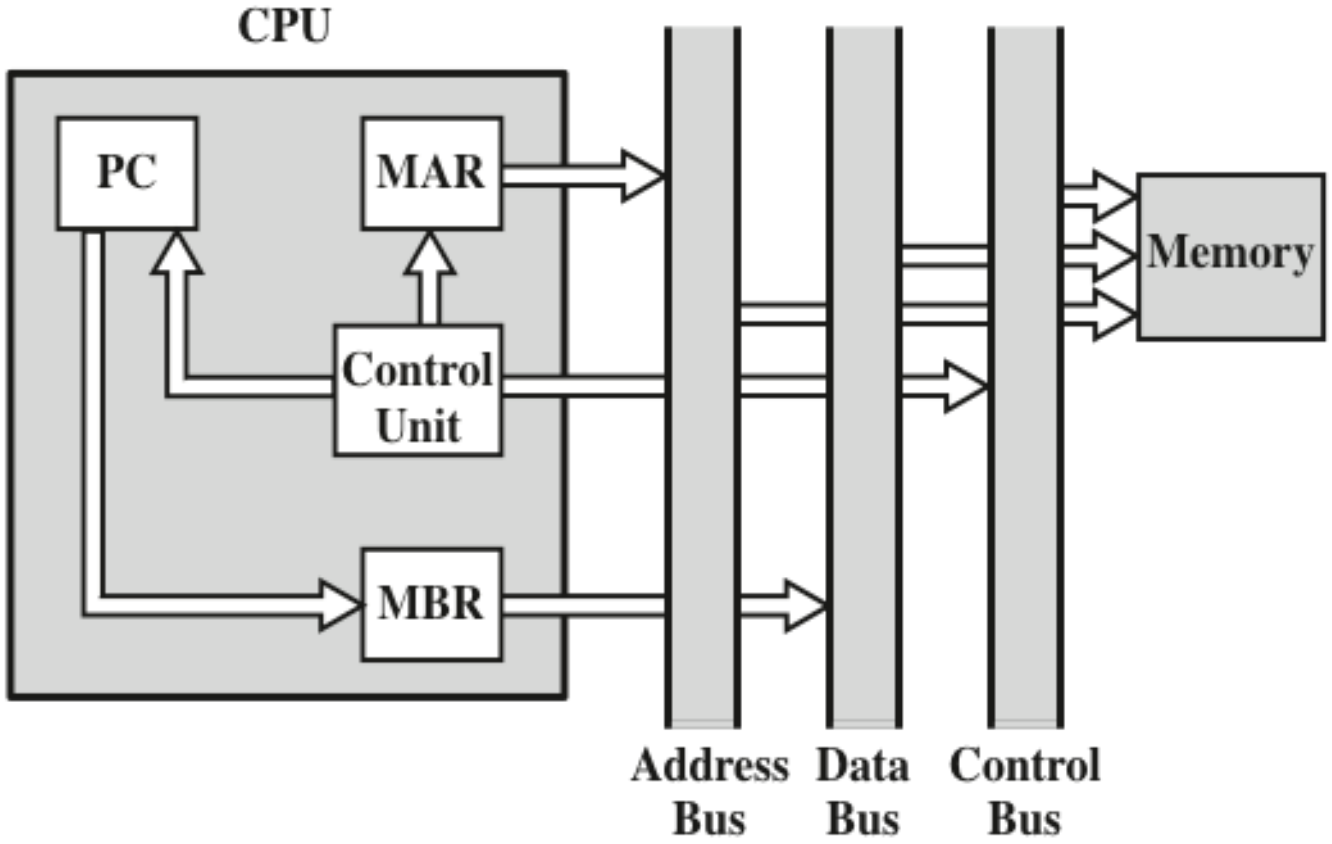



**Figure 14.8 Data Flow, Interrupt Cycle**

# Two-Stage Instruction Pipeline

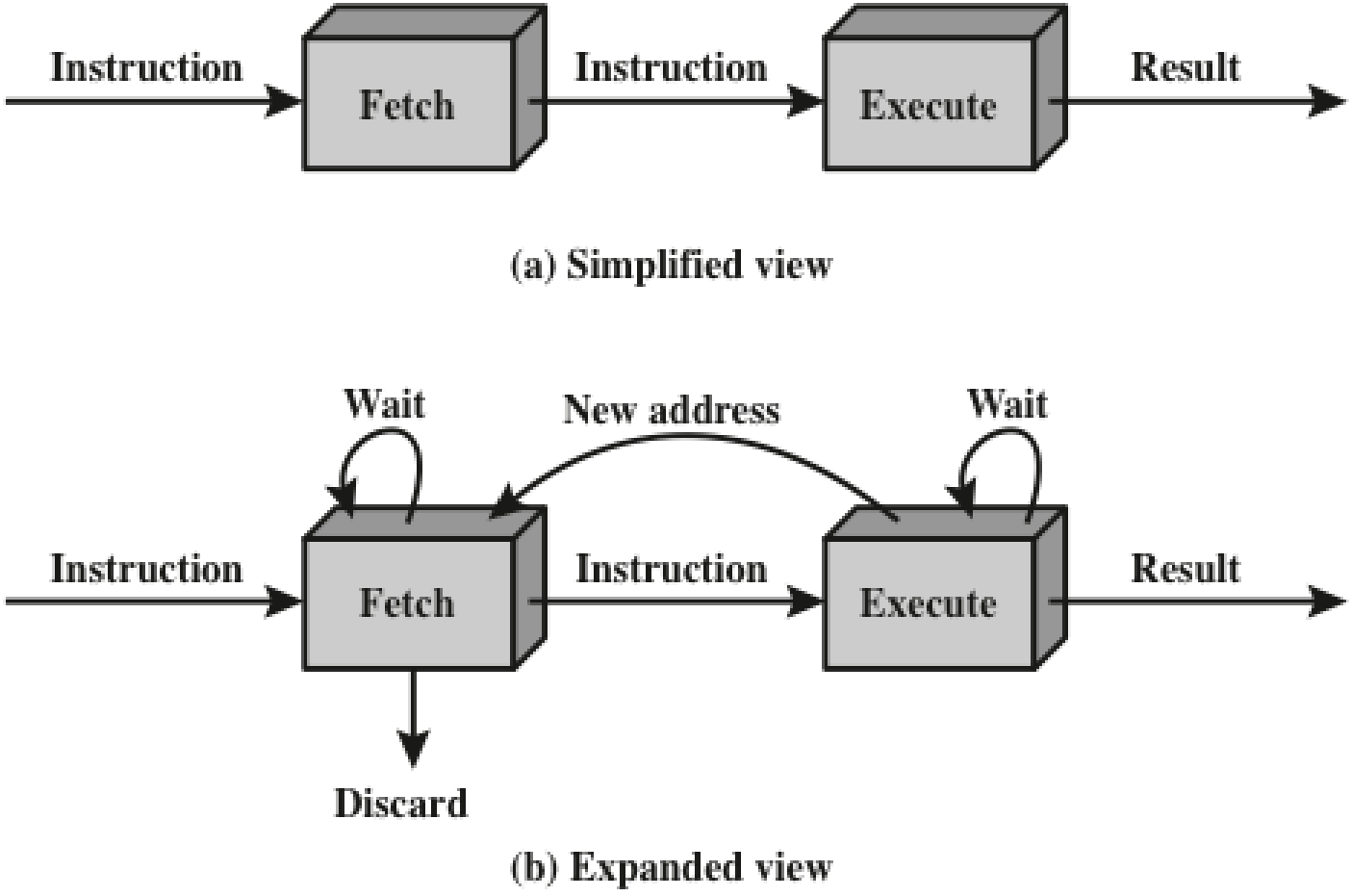



**Figure 14.9 Two-Stage Instruction Pipeline**

- A conditional branch instruction makes the address of the next instruction to be fetched unknown
- Guessing can reduce the time loss. A simple rule is the following:
  - When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction.
  - Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

# 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

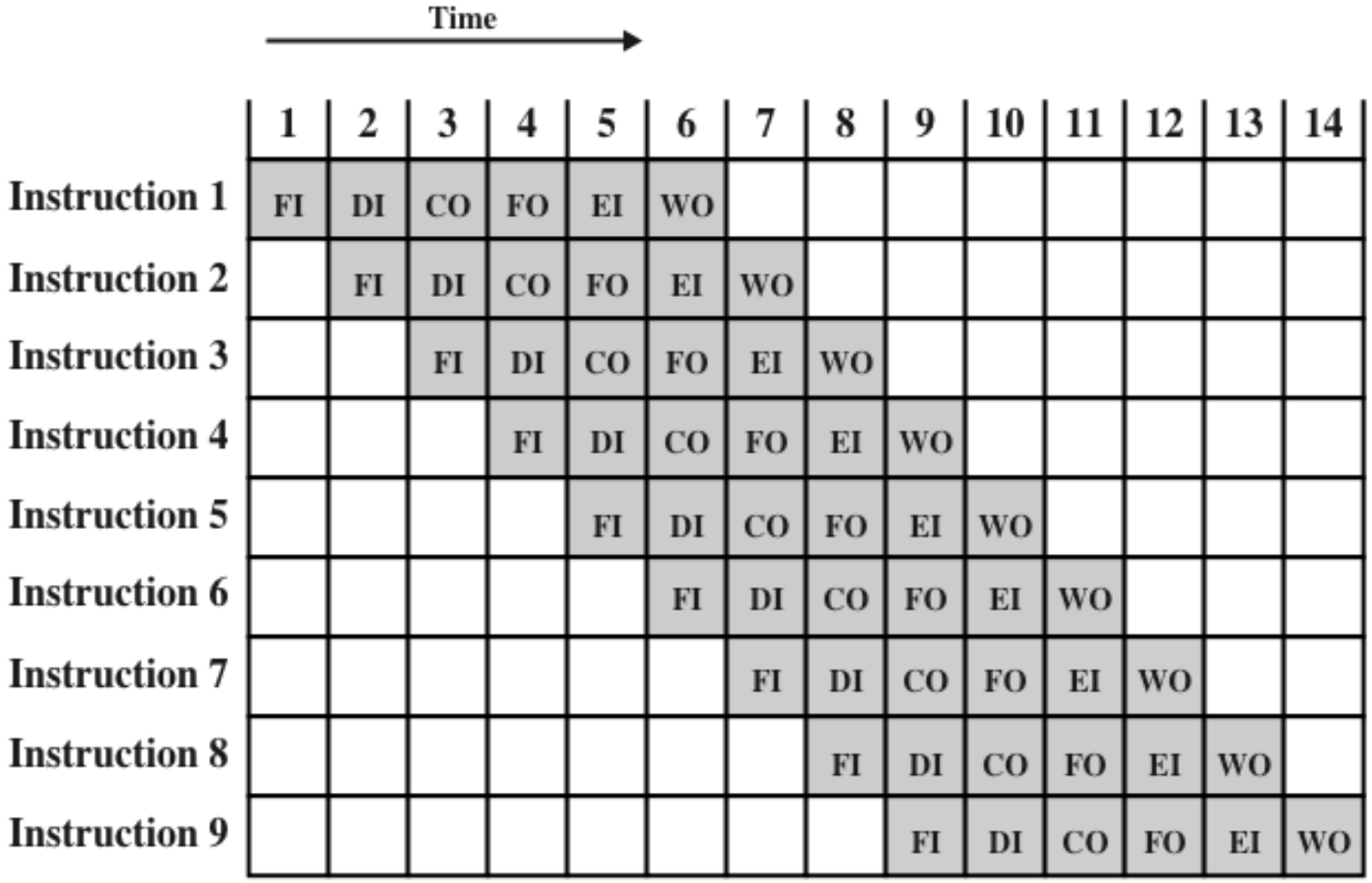



**Figure 14.10 Timing Diagram for Instruction Pipeline Operation**

# The Effect of a Conditional Branch on Instruction Pipeline Operation

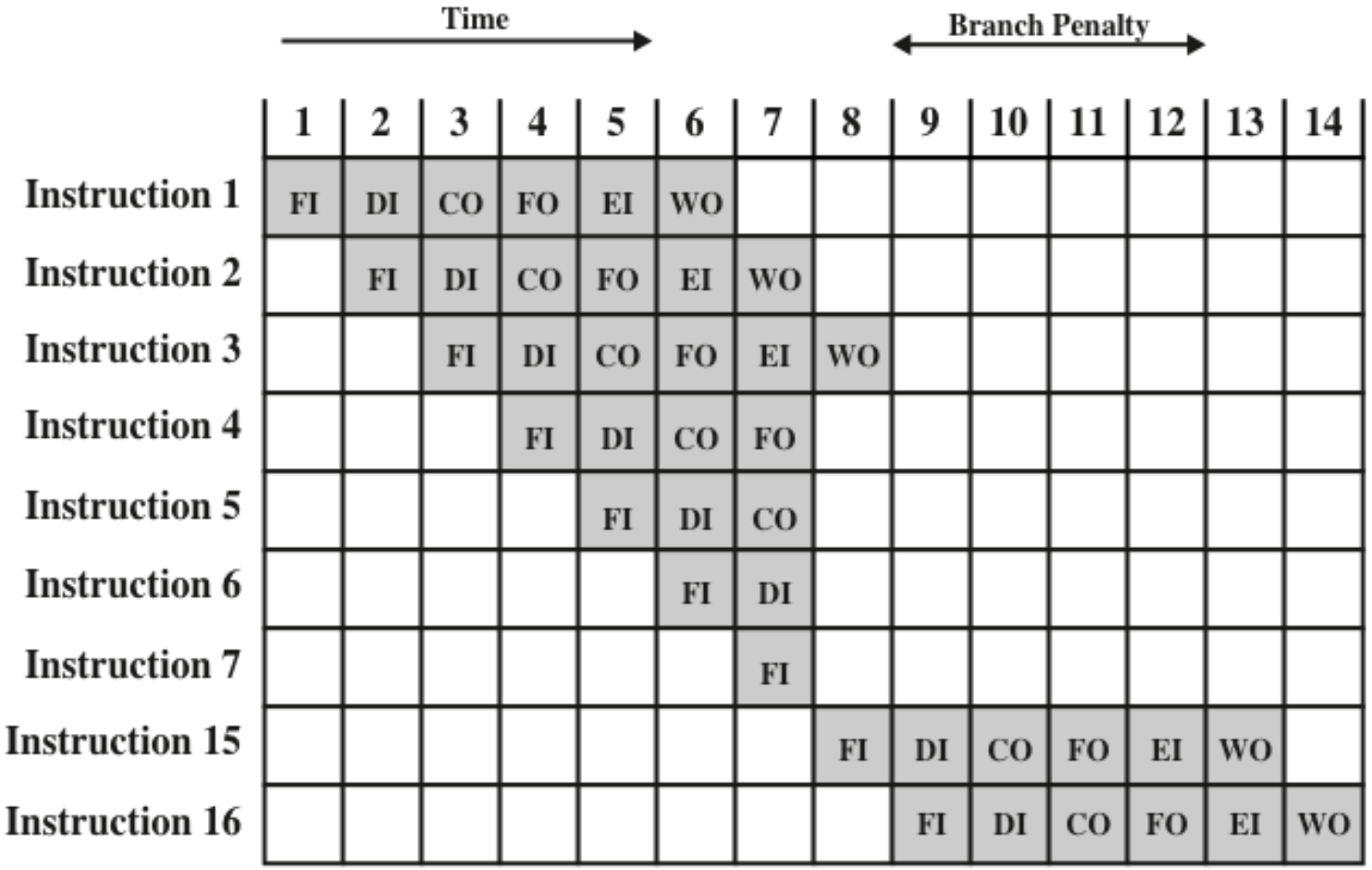



**Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation**

- Several other factors serve to limit the performance enhancement of pipeline:
  - stages are not of equal duration
  - conditional branch instruction, which can invalidate several instruction fetches
  - an interrupt.
- Figure 14.11 illustrates the effects of the conditional branch, using the same program as Figure 14.10.
- Assume that 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.

# Six Stage Instruction Pipeline

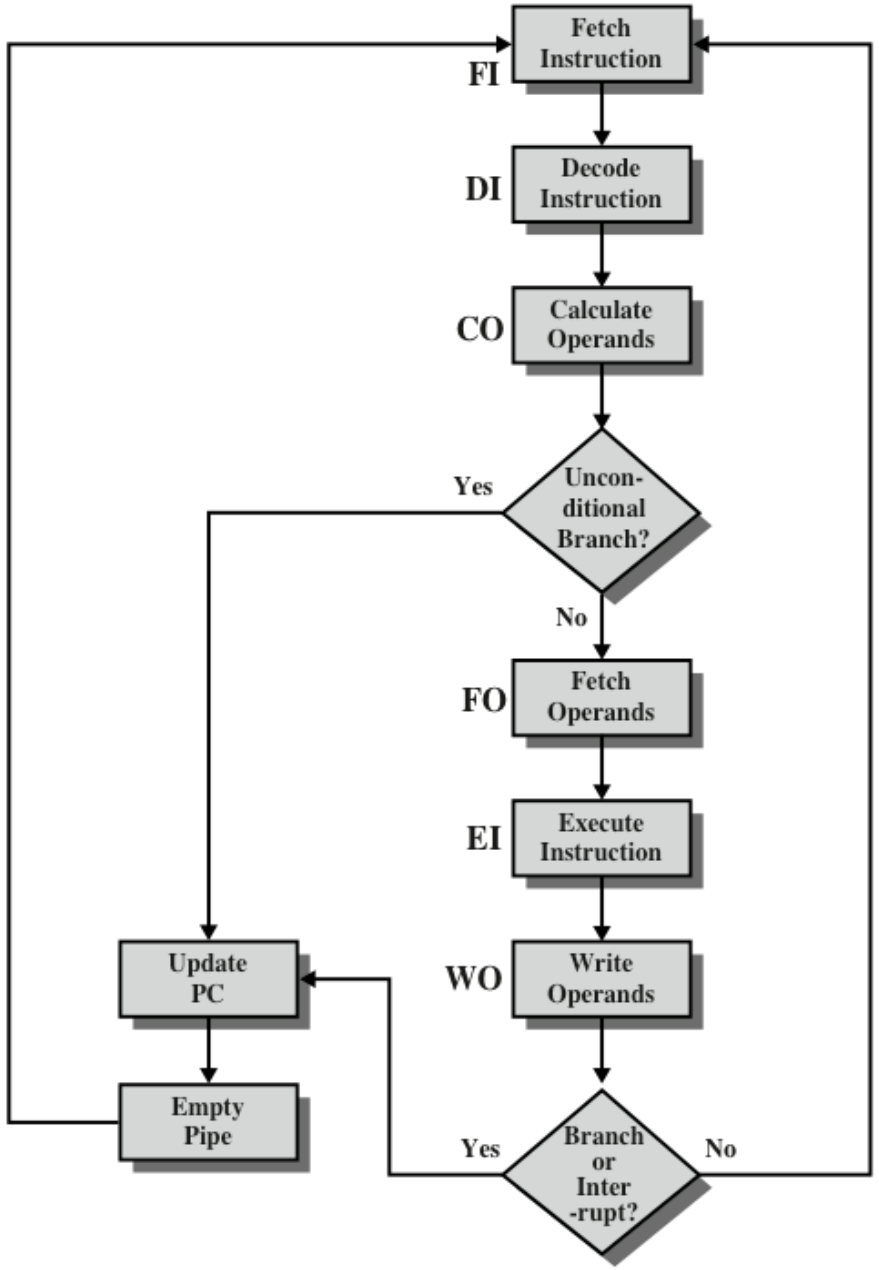



Figure 14.12 Six-Stage Instruction Pipeline

- The CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict.

# Alternative Pipeline Depiction

- Figure 14.13b, (which corresponds to Figure 14.11), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15.
- At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

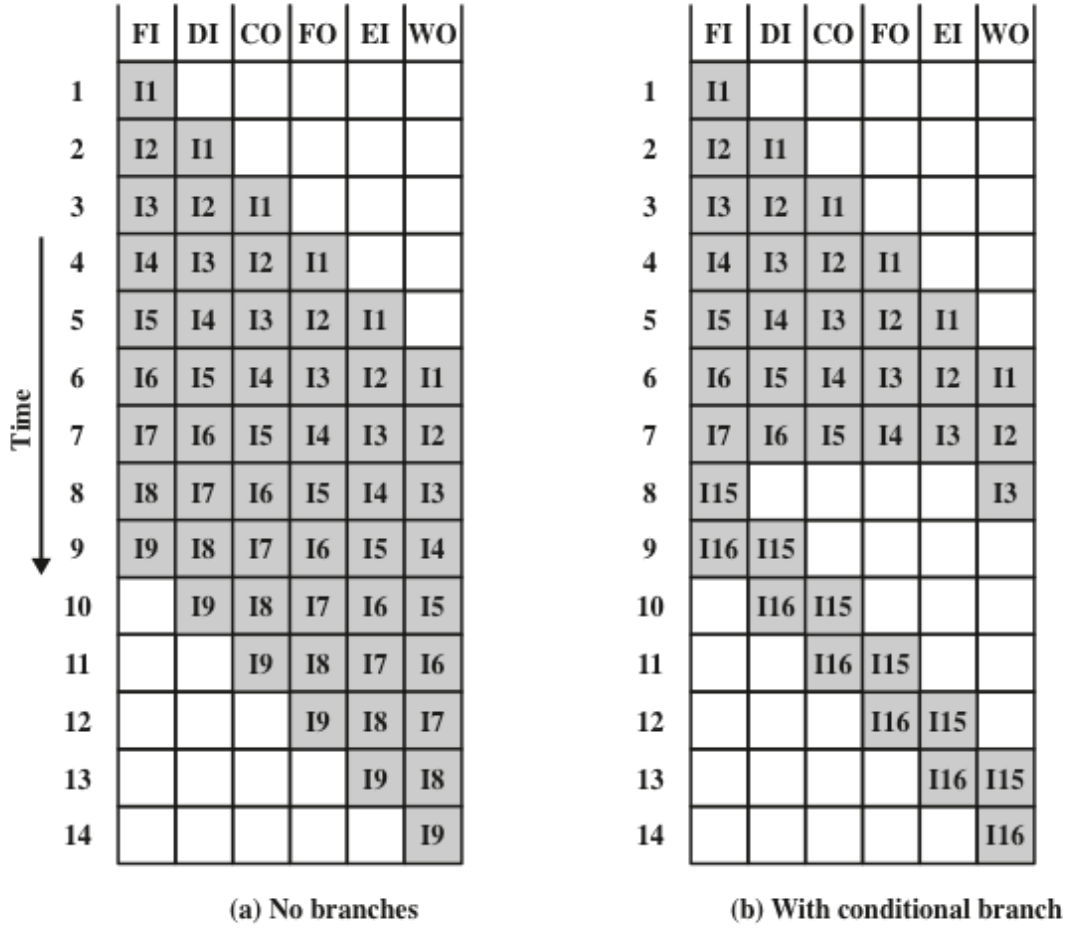
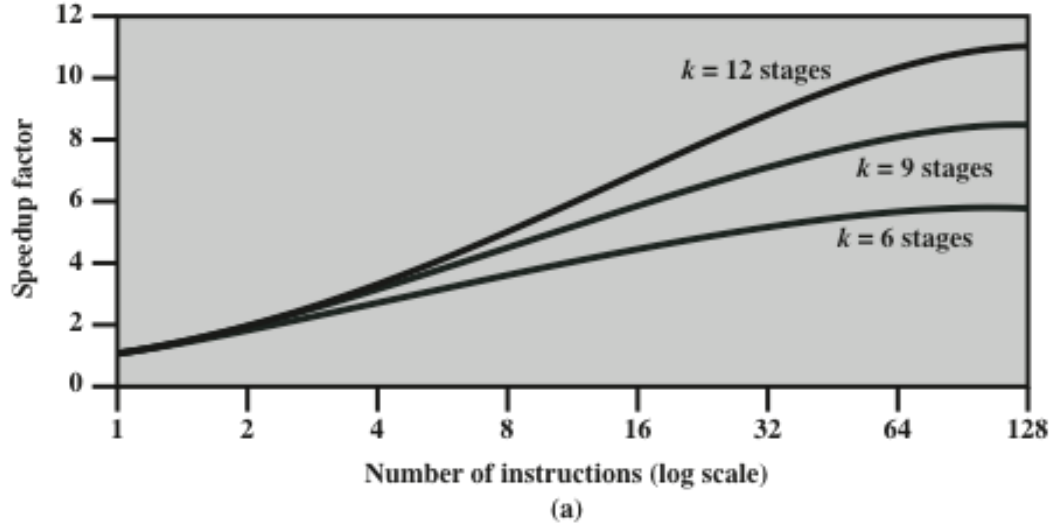



**Figure 14.13 An Alternative Pipeline Depiction**

# Speedup Factors with Instruction Pipelining



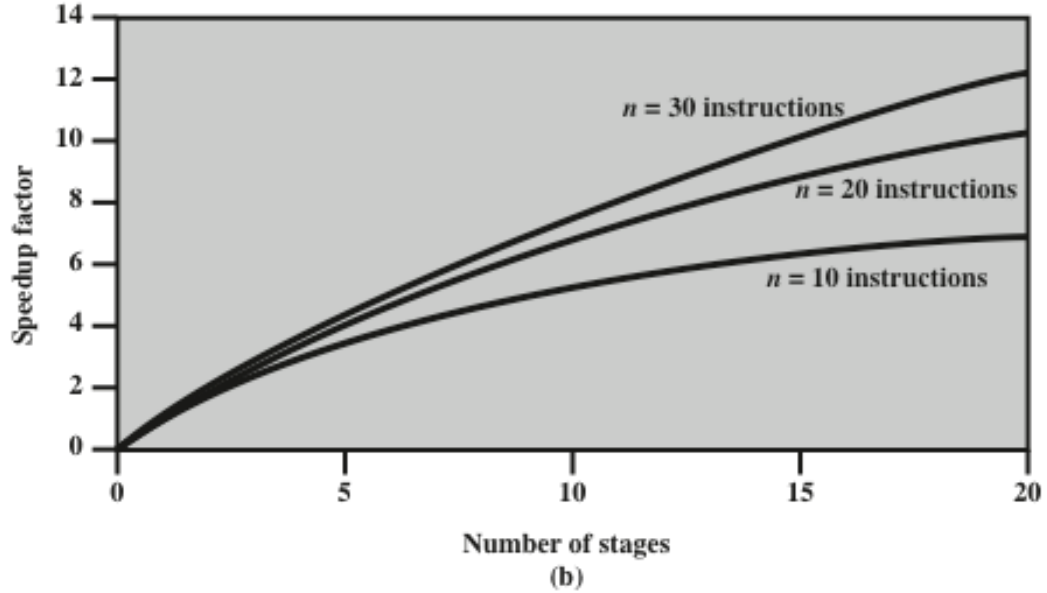




**Figure 14.14 Speedup Factors with Instruction Pipelining**

# Exercise

- Assuming a hypothetical processor with no branching, if:
  - τ (cycle time) is the average total time required for each stage to execute an instruction
  - k is number of stages

1. Derive the formula to calculate the total time required to execute n instructions.

$$T_{k,n} \approx \tau k + (n-1)\tau$$

2. Calculate for $T_{6,10} \approx 15\tau$
3. How long would it take for a similar processor without pipelining?

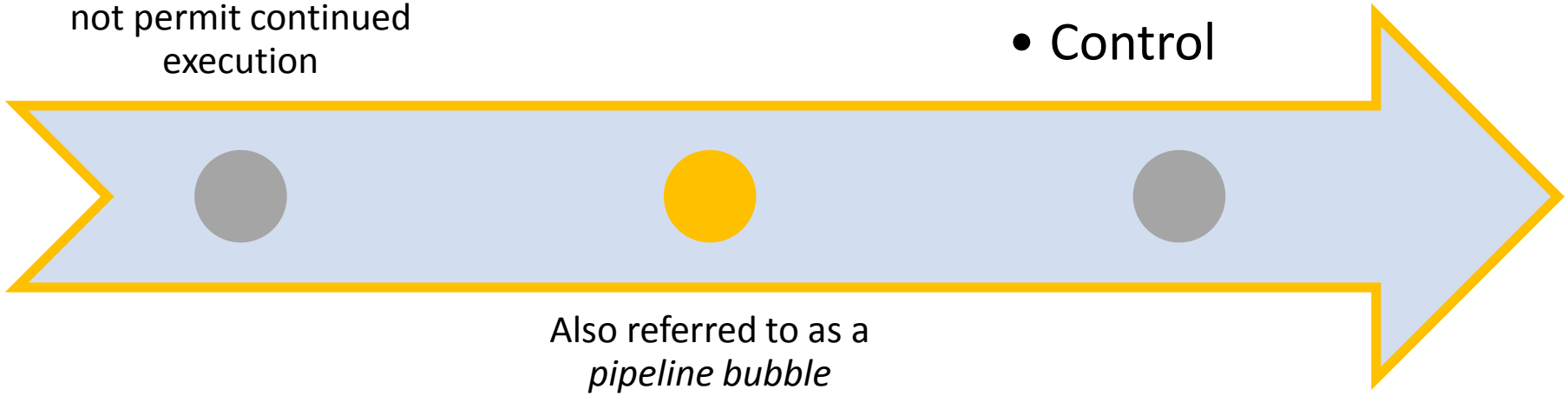
$$T_n \approx \tau k n$$

# Pipeline Hazards

Occur when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution

There are three types of hazards:

- Resource
- Data
- Control



Also referred to as a *pipeline bubble*

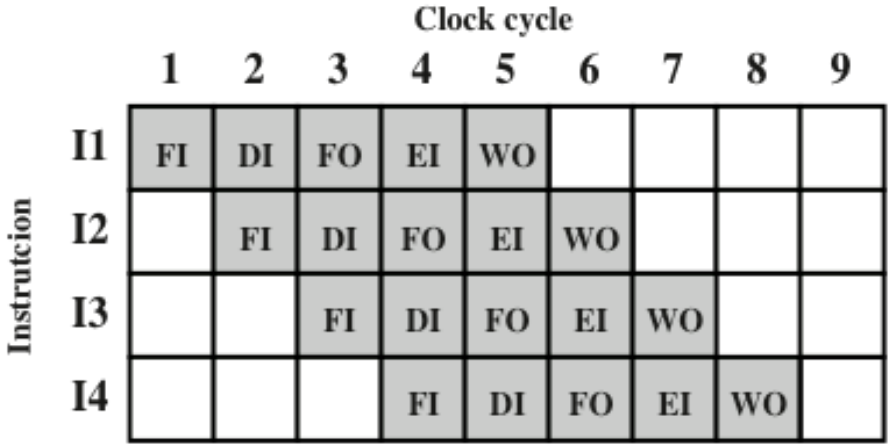
# Resource Hazards

A resource hazard occurs when two or more instructions that are already in the pipeline need the same resource

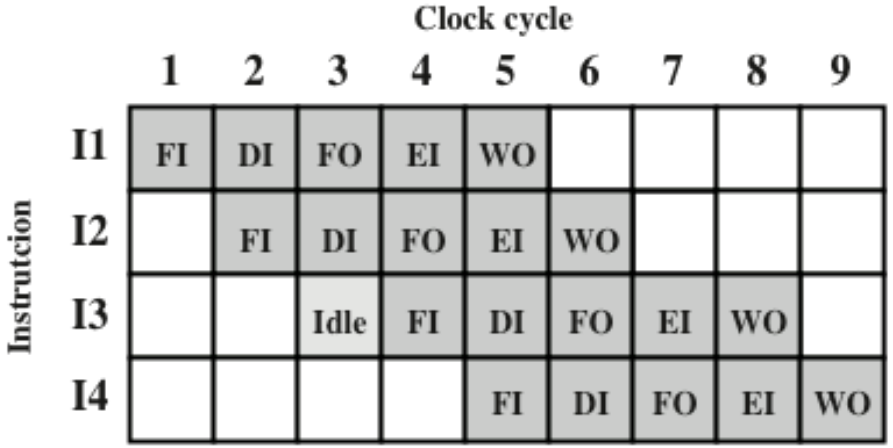
The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline

A resource hazard is sometimes referred to as a *structural hazard*

*E.g. When* main memory has a single port and that FI and FO must be performed one at a time from main memory.




(a) Five-stage pipeline, ideal case




(b) I1 source operand in memory

**Figure 14.15 Example of Resource Hazard**

# Data Hazards

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

Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution.

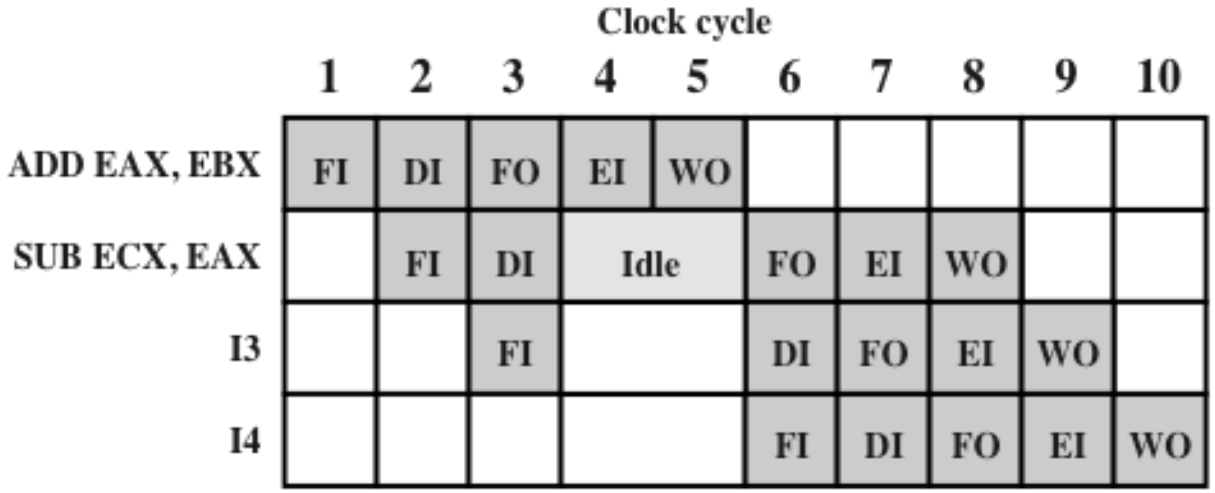



Figure 14.16 Example of Data Hazard

# 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

Q:

1. what type of data hazard could this code produce? Ans. RAW

```
ADD EAX, EBX
SUB ECX, EAX
```

2. How can you modify the code it produce WAR and WAW data hazards

# 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. There are two problems with this approach:

- 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
- Despite these drawbacks, this strategy can improve performance. Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.

# 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
- 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

# Branch Prediction Flow Chart

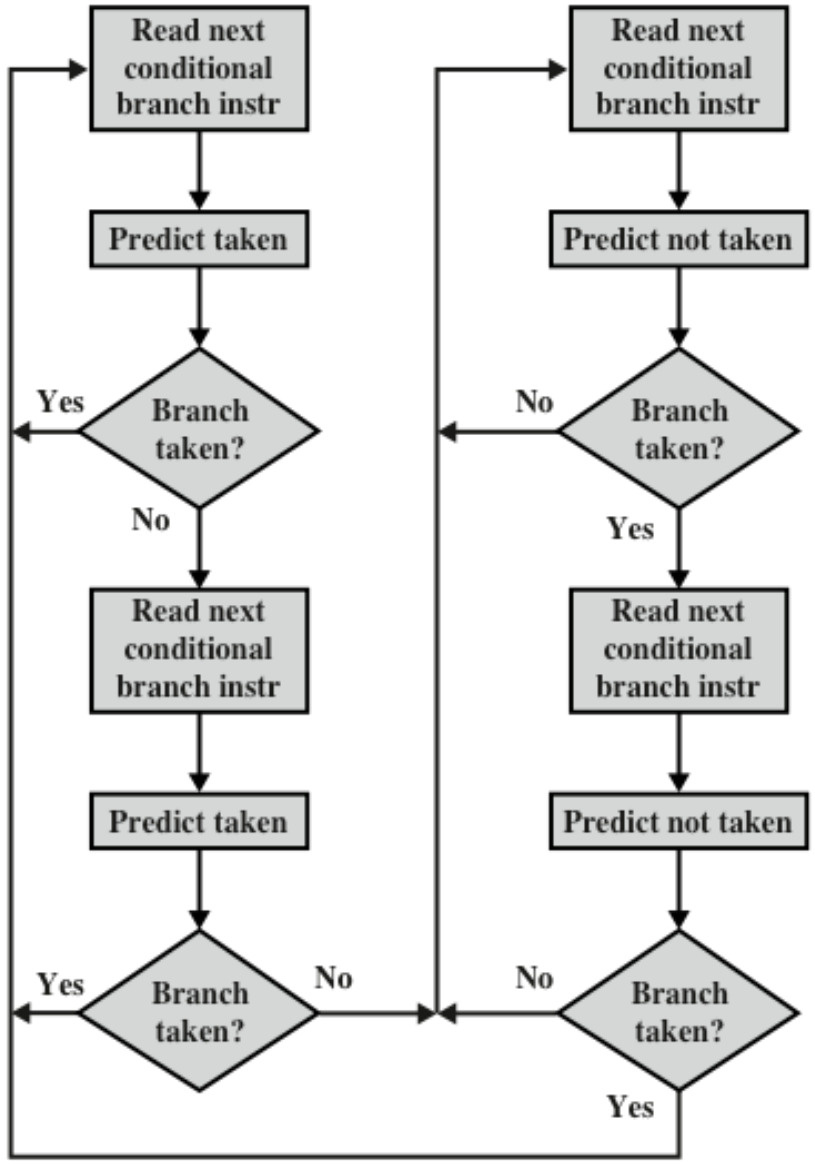



Figure 14.18 Branch Prediction Flow Chart

AAiT

# Branch Prediction State Diagram

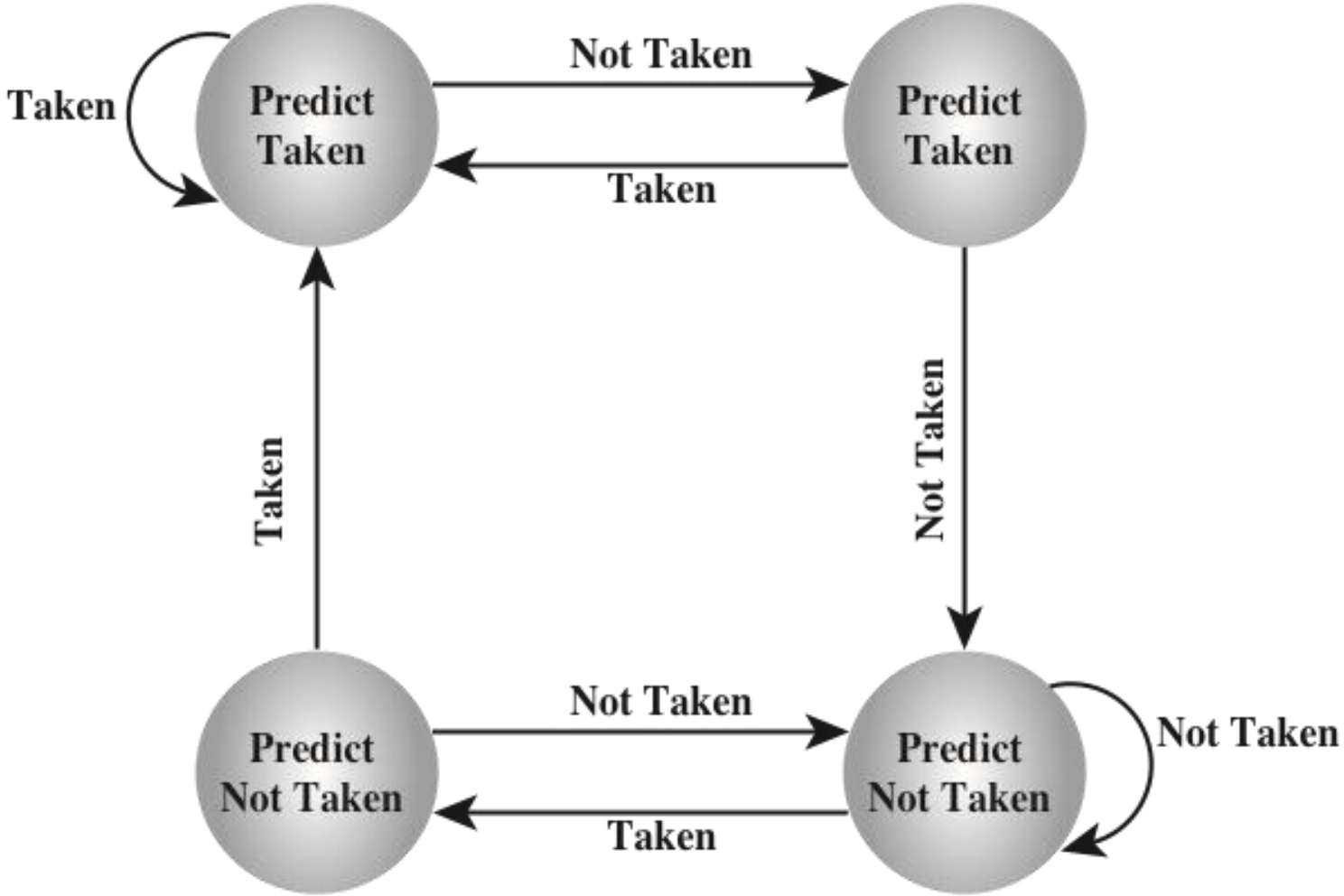



Figure 14.19 Branch Prediction State Diagram

# 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 14.21a shows that there is no delay introduced into the pipeline when a memory access is required.
- However, as Figure 14.21b shows, there can be a delay for values used to compute memory addresses. That is, if a value is loaded from memory into a register and that register is then used as a base register in the next instruction, the processor will stall for one cycle.
- Figure 14.21c illustrates the timing of a branch instruction, assuming that the branch is taken. The compare instruction updates condition codes in the WB stage, and bypass paths make this available to the EX stage of the jump instruction at the same time. In parallel, the processor runs a speculative fetch cycle to the target of the jump during the EX stage of the jump instruction. If the processor determines a false branch condition, it discards this prefetch and continues execution with the next sequential instruction (already fetched and decoded).


(a) No Data Load Delay in the Pipeline


(b) Pointer Load Delay


(c) Branch Instruction Timing

**Figure 14.21 80486 Instruction Pipeline Examples**

# Table 14.2
# x86 Processor Registers

AAiT

| Type | Number | Length (bits) | Purpose |
|---|---|---|---|
| General | 8 | 32 | General-purpose user registers |
| Segment | 6 | 16 | Contain segment selectors |
| EFLAGS | 1 | 32 | Status and control bits |
| Instruction Pointer | 1 | 32 | Instruction pointer |

**(a) Integer Unit in 32-bit Mode**

| Type | Number | Length (bits) | Purpose |
|---|---|---|---|
| General | 16 | 32 | General-purpose user registers |
| Segment | 6 | 16 | Contain segment selectors |
| RFLAGS | 1 | 64 | Status and control bits |
| Instruction Pointer | 1 | 64 | Instruction pointer |

**(b) Integer Unit in 64-bit Mode**

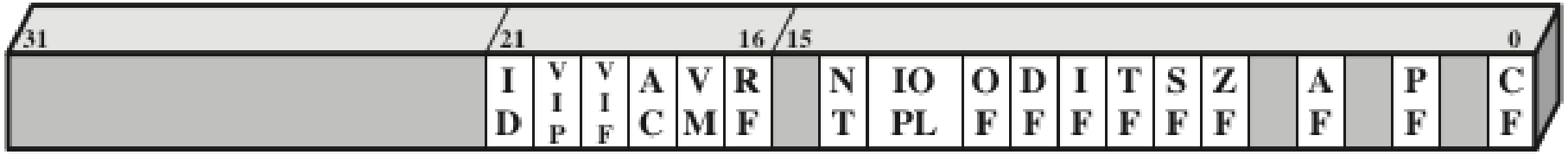
# Table 14.2 x86 Processor Registers

| Type | Number | Length (bits) | Purpose |
|---|---|---|---|
| Numeric | 8 | 80 | Hold floating-point numbers |
| Control | 1 | 16 | Control bits |
| Status | 1 | 16 | Status bits |
| Tag Word | 1 | 16 | Specifies contents of numeric registers |
| Instruction Pointer | 1 | 48 | Points to instruction interrupted by exception |
| Data Pointer | 1 | 48 | Points to operand interrupted by exception |

(c) Floating-Point Unit

# x86 EFLAGS Register

- The EFLAGS register indicates the condition of the processor and helps to control its operation.



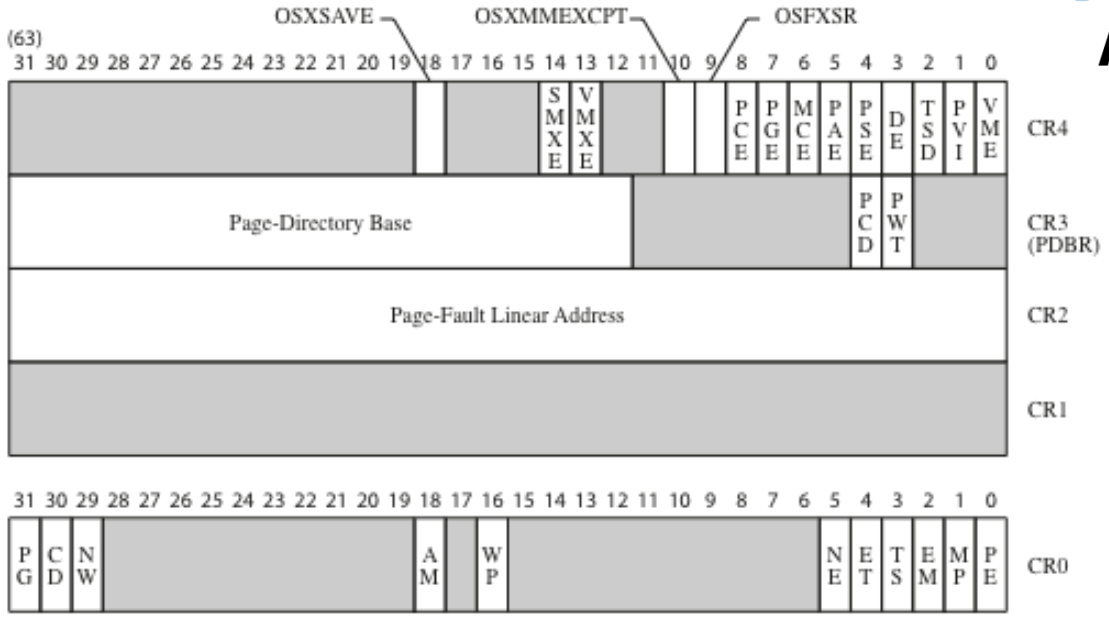

ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag

DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

**Figure 14.22 x86 EFLAGS Register**

# Control Registers

- The x86 employs four control registers (register CR1 is unused) to control various aspects of processor operation
- All of the registers except CR0 are either 32 bits or 64 bits long, depending on whether the implementation supports the x86 64-bit architecture.
- The CR0 register contains system control flags, which control modes or indicate states that apply generally to the processor rather than to the execution of an individual task.
- When paging is enabled, the CR2 and CR3 registers are valid




shaded area indicates reserved bits

| | | | | | |
|---|---|---|---|---|---|
| OSXSAVE | = | XSAVE enable bit | PCD | = | Page-level Cache Disable |
| SMXE | = | Enable Safer mode extensions | PWT | = | Page-level Writes Transparent |
| VMXE | = | Enable virtual machine extensions | PG | = | Paging |
| OSXMMEXCPT | = | Support unmasked SIMD FP exceptions | CD | = | Cache Disable |
| OSFXSR | = | Support FXSAVE, FXSTOR | NW | = | Not Write Through |
| PCE | = | Performance Counter Enable | AM | = | Alignment Mask |
| PGE | = | Page Global Enable | WP | = | Write Protect |
| MCE | = | Machine Check Enable | NE | = | Numeric Error |
| PAE | = | Physical Address Extension | ET | = | Extension Type |
| PSE | = | Page Size Extensions | TS | = | Task Switched |
| DE | = | Debug Extensions | EM | = | Emulation |
| TSD | = | Time Stamp Disable | MP | = | Monitor Coprocessor |
| PVI | = | Protected Mode Virtual Interrupt | PE | = | Protection Enable |
| VME | = | Virtual 8086 Mode Extensions | | | |

**Figure 14.23 x86 Control Registers**

# Interrupt Processing

## Interrupts and Exceptions

- Interrupts
  - Generated by a signal from hardware and it may occur at random times during the execution of a program
  - **Maskable**: Received on the processor's INTR pin. The processor does not recognize a maskable interrupt unless the interrupt enable flag (IF) is set.
  - **Nonmaskable**: Received on the processor's NMI pin. Recognition of such interrupts cannot be prevented.
- Exceptions
  - Generated from software and is provoked by the execution of an instruction
  - Processor detected
  - Programmed
- Interrupt vector table
  - Every type of interrupt is assigned a number
  - Number is used to index into the interrupt vector table

# The ARM Processor

ARM is primarily a RISC system with the following attributes:

- Moderate array of uniform registers
- A load/store model of data processing in which operations only perform on operands in registers and not directly in memory
- A uniform fixed-length instruction of 32 bits for the standard set and 16 bits for the Thumb instruction set
- Separate arithmetic logic unit (ALU) and shifter units
- A small number of addressing modes with all load/store addresses determined from registers and instruction fields
- Auto-increment and auto-decrement addressing modes are used to improve the operation of program loops
- Conditional execution of instructions minimizes the need for conditional branch instructions, thereby improving pipeline efficiency, because pipeline flushing is reduced

# Simplified ARM Organization

- The ARM processor organization varies substantially from one implementation to the next, particularly when based on different versions of the ARM architecture.
- Figure 14.25 is a generic ARM organization, arrows indicate the flow of data. Each box represents a functional hardware unit or a storage unit.
- ARM data processing instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*.
- The results of an operation are fed back to the destination register. Load/store instructions may also use the output of the arithmetic units to generate the memory address for a load or store.

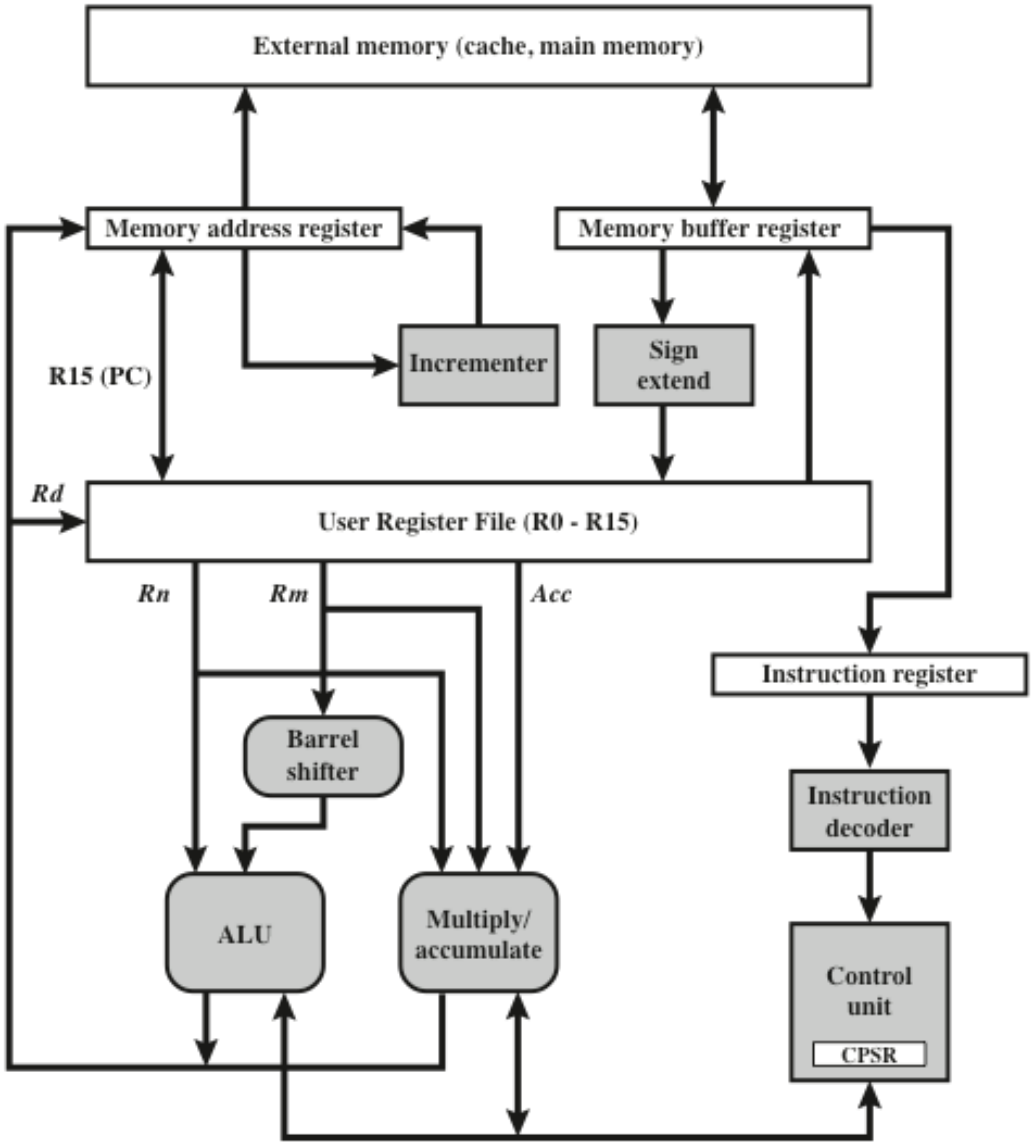
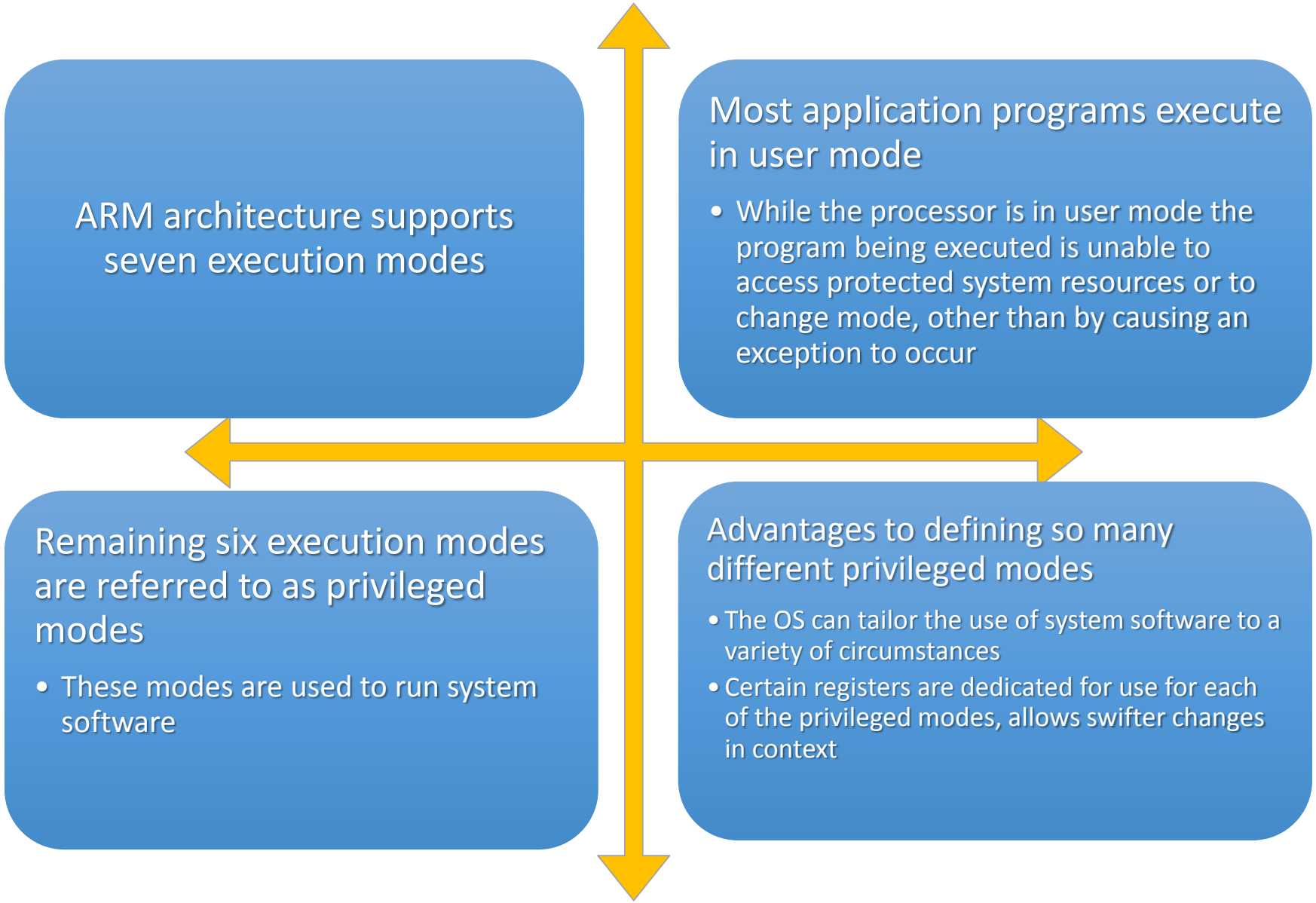
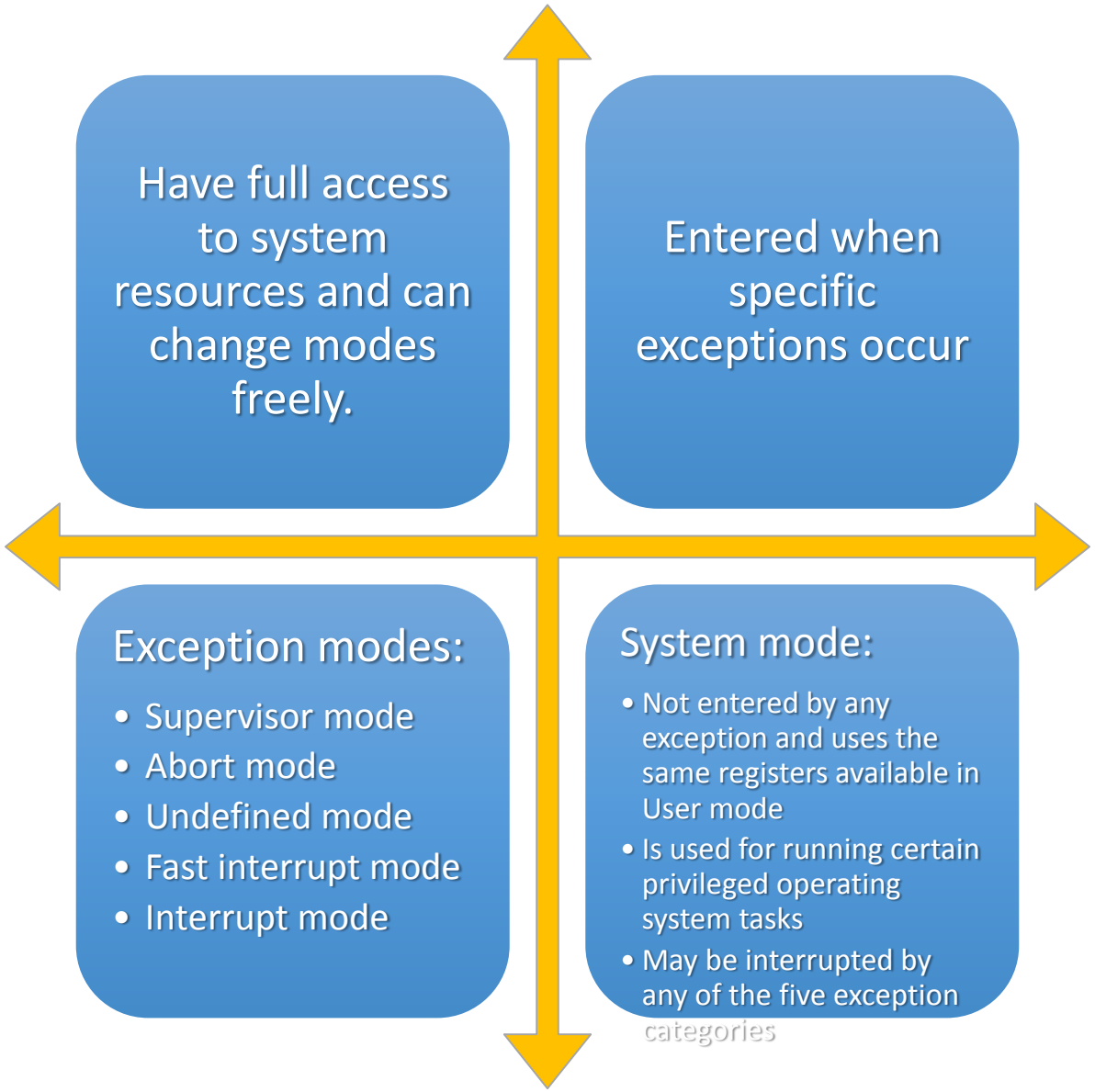



**Figure 14.25 Simplified ARM Organization**

# Processor Modes



# Exception Modes




*** Supervisor mode:** Usually what the OS runs in. It is entered when the processor encounters a software interrupt instruction. Software interrupts are a standard way to invoke operating system services on ARM.

*** Abort mode:** Entered in response to memory faults.

*** Undefined mode:** Entered when the processor attempts to execute an instruction
that is supported neither by the main integer core nor by one of the coprocessors.

*** Fast interrupt mode:** Entered whenever the processor receives an interrupt signal from the designated fast interrupt source. A fast interrupt cannot be interrupted, but a fast interrupt may interrupt a normal interrupt.

• **Interrupt mode:** Entered whenever the processor receives an interrupt signal from any other interrupt source (other than fast interrupt). An interrupt may only be interrupted by a fast interrupt.

# ARM
# Register Organization

- Figure 14.26 depicts the user-visible registers for the ARM. The ARM processor has a total of 37 32-bit registers (16 (shared General *R0-R15)* + 1 shared CPSR + 20 specific to every mode, shaded ) , classified as follows:
  - Thirty-one registers referred to in the ARM manual as general-purpose registers. In fact, some of these, such as the program counters, have special purposes.
  - Six program status registers.

| Modes | | | | | | |
|---|---|---|---|---|---|---|
| | Privileged modes | | | | | |
| | | Exception modes | | | | |
| **User** | **System** | **Supervisor** | **Abort** | **Undefined** | **Interrupt** | **Fast Interrupt** |
| R0 | R0 | R0 | R0 | R0 | R0 | R0 |
| R1 | R1 | R1 | R1 | R1 | R1 | R1 |
| R2 | R2 | R2 | R2 | R2 | R2 | R2 |
| R3 | R3 | R3 | R3 | R3 | R3 | R3 |
| R4 | R4 | R4 | R4 | R4 | R4 | R4 |
| R5 | R5 | R5 | R5 | R5 | R5 | R5 |
| R6 | R6 | R6 | R6 | R6 | R6 | R6 |
| R7 | R7 | R7 | R7 | R7 | R7 | R7 |
| R8 | R8 | R8 | R8 | R8 | R8 | R8_fiq |
| R9 | R9 | R9 | R9 | R9 | R9 | R9_fiq |
| R10 | R10 | R10 | R10 | R10 | R10 | R10_fiq |
| R11 | R11 | R11 | R11 | R11 | R11 | R11_fiq |
| R12 | R12 | R12 | R12 | R12 | R12 | R12_fiq |
| R13 (SP) | R13 (SP) | R13_svc | R13_abt | R13_und | R13_irq | R13_fiq |
| R14 (LR) | R14 (LR) | R14_svc | R14_abt | R14_und | R14_irq | R14_fiq |
| R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) |
| CPSR | CPSR | CPSR | CPSR | CPSR | CPSR | CPSR |
| | | SPSR_svc | SPSR_abt | SPSR_und | SPSR_irq | SPSR_fiq |

Shading indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode.

SP = stack pointer
LR = link register
PC = program counter

CPSR = current program status register
SPSR = saved program status register

# Format of ARM CPSR and SPSR

- The CPSR is accessible in all processor modes. Each exception mode also has a dedicated SPSR that is used to preserve the value of the CPSR when the associated exception occurs.

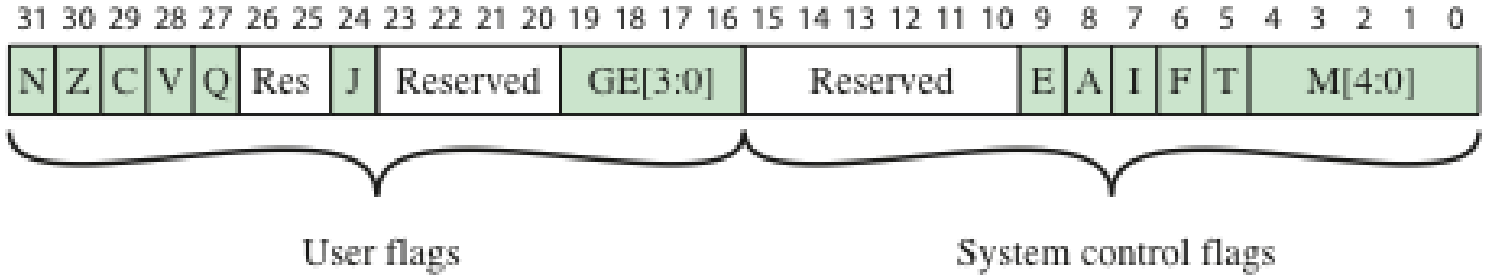



**Figure 14.27 Format of ARM CPSR AND SPSR**

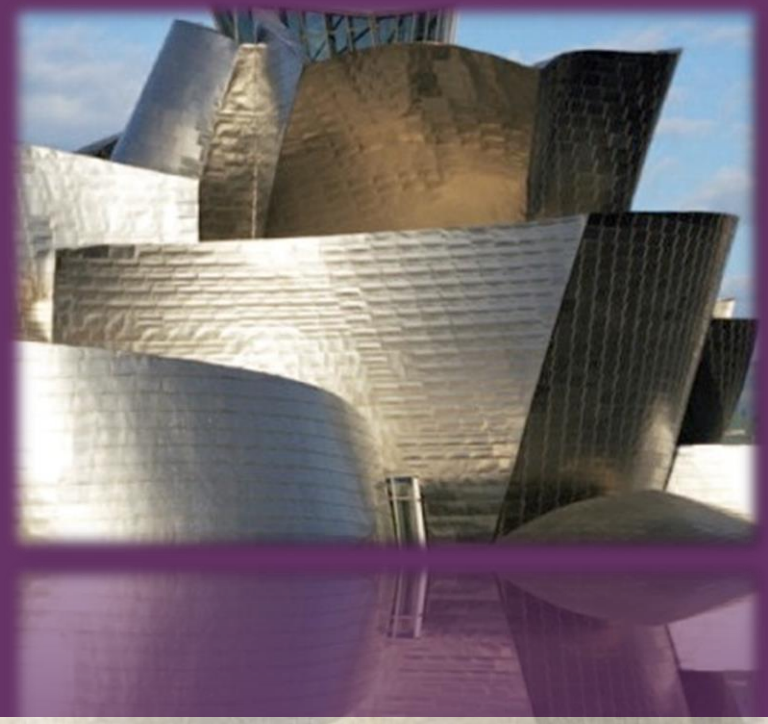
- **Condition code flags:** The N, Z, C, and V flags, which are discussed in Chapter 12.
- **Q flag:** used to indicate whether overflow and/or saturation has occurred in some SIMD-oriented instructions.
- **J bit:** indicates the use of special 8-bit instructions, known as Jazelle instructions, which are beyond the scope of our discussion.
- **GE[3:0] bits:** SIMD instructions use bits [19:16] as Greater than or Equal (GE) flags for individual bytes or halfwords of the result.
- **Ebit:** Controls load and store endianness for data; ignored for instruction fetches.
- **Interrupt disable bits:** The A bit disables imprecise data aborts when set; the I bit disables IRQ interrupts when set; and the F bit disables FIQ interrupts when set.
- **T bit:** Indicates whether instructions should be interpreted as normal ARM instructions or Thumb instructions.
- **Mode bits:** Indicates the processor mode.

# Summary

- Processor organization
- Register organization
  - User-visible registers
  - Control and status registers
- Instruction cycle
  - The indirect cycle
  - Data flow
- The x86 processor family
  - Register organization
  - Interrupt processing

## Processor Structure and Function

- Instruction pipelining
  - Pipelining strategy
  - Pipeline performance
  - Pipeline hazards
  - Dealing with branches
  - Intel 80486 pipelining
- The Arm processor
  - Processor organization
  - Processor modes
  - Register organization
  - Interrupt processing



# William Stallings
# Computer Organization and Architecture
# 9th Edition