Logic Design and Computer Organization

IS234AT

Unit - V

Dr. Premananda B.S.

Modules

I. Synchronous Sequential Logic

II. Registers, Counters, and Memory

III. A Top-Level View of Computer Function and Interconnection, Cache Memory

IV. Input/ Output and Computer Arithmetic

V. Instruction Sets Characteristics and Functions, Processor Structure and Function, and Parallel Processing

Books

- Computer Organization and Architecture Designing for Performance, William Stallings, 10th Edition, Pearson, ISBN: 978-0134101613

Instruction Sets Characteristics and Functions

- Machine Instruction Characteristics
- Types of Operands
- Types of Operations
- Intel x86 Data Types
- Addressing Modes
- Instruction Formats

Processor Structure and Function

- Processor Organization
- Register Organization
- Instruction Cycle
- Instruction Pipelining

Parallel Processing

- Multiple Processor Organizations
- Symmetric Multiprocessors
- Cache Coherence
- MESI Protocol

Machine Instruction Characteristics

- The operation of the processor is determined by the instructions it executes, referred to as *machine instructions* or *computer instructions.*
- The collection of different instructions that the processor can execute is referred to as the processor's *instruction set.*
- Each instruction must contain the information required by the processor for execution.
- Figure 12.1, shows the steps involved in instruction execution and, by implication, defines the elements of a machine instruction.

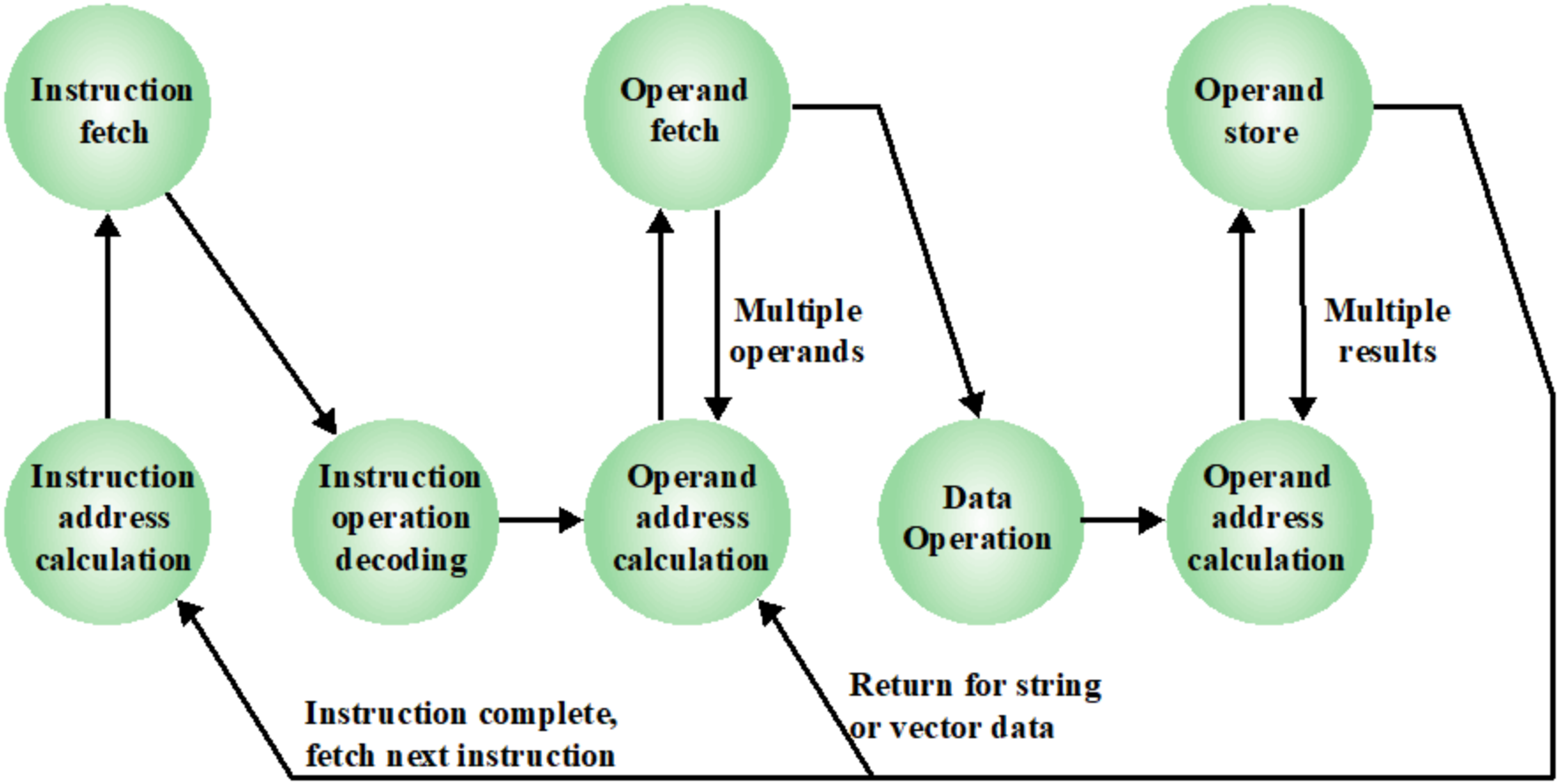


Figure 12.1 Instruction Cycle State Diagram

Elements of a Machine Instruction

Operation code (opcode)

- Specifies the operation to be performed.
- The operation is specified by a binary code, known as the operation code, or *opcode*

Source operand reference

- The operation may involve one or more source operands, that is, operands that are inputs for the operation

Result operand reference

- The operation may produce a result

Next instruction reference

- Tells the processor where to fetch the next instruction after the execution of this instruction is complete

Source and result operands can be in one of four areas:

1) Main or virtual memory

- As with next instruction references, the main or virtual memory address must be supplied

2) I/O device

- The instruction must specify the I/O module and device for the operation. If memory-mapped I/O is used, this is just another main or virtual memory address

3) Processor register

- A processor contains one or more registers that may be referenced by machine instructions.
- If more than one register exists each register is assigned a unique name or number and the instruction must contain the number of the desired register

4) Immediate

- The value of the operand is contained in a field in the instruction being executed

Instruction Representation

- Within the computer each instruction is represented by a sequence of bits
- The instruction is divided into fields, corresponding to the constituent elements of the instruction

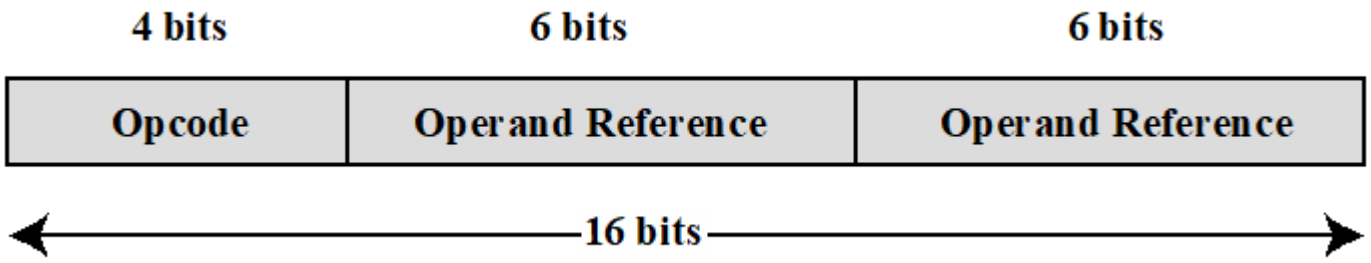


Figure 12.2 A Simple Instruction Format

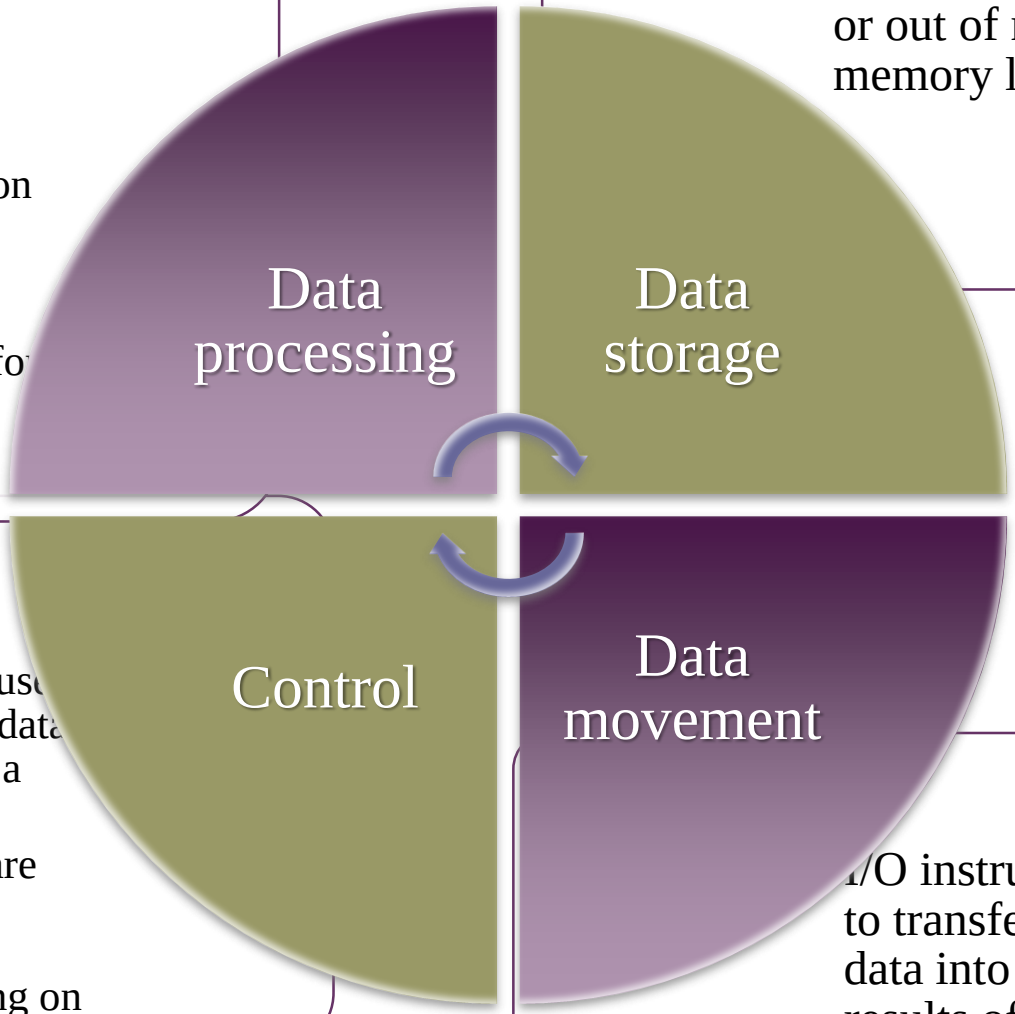
Instruction Representation

- Opcodes are represented by abbreviations called *mnemonics*
- Examples include:
  - ADD Add
  - SUB Subtract
  - MUL Multiply
  - DIV Divide
  - LOAD Load data from memory
  - STOR Store data to memory
- Operands are also represented symbolically
- Each symbolic opcode has a fixed binary representation
  - The programmer specifies the location of each symbolic operand

Instruction Types

- Arithmetic instructions provide computational capabilities for processing numeric data
- Logic (Boolean) instructions operate on the bits of a word as bits rather than as numbers, thus they provide capabilities fo processing any other type of data the user may wish to employ

- Movement of data into or out of register and or memory locations



- Test instructions are us to test the value of a dat word or the status of a computation
- Branch instructions are used to branch to a different set of instructions depending on the decision made

I/O instructions are needed to transfer programs and data into memory and the results of computations back out to the user

Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions

Instruction		Comment
LOAD	D	$AC \leftarrow D$
MPY	E	$AC \leftarrow AC \times E$
ADD	C	$AC \leftarrow AC + C$
STOR	Y	$Y \leftarrow AC$
LOAD	A	$AC \leftarrow A$
SUB	B	$AC \leftarrow AC - B$
DIV	Y	$AC \leftarrow AC \div Y$
STOR	Y	$Y \leftarrow AC$

(c) One-address instructions

Figure 12.3 Programs to Execute $\mathbf{Y = \frac{A - B}{C + (D \times E)}}$

Table 12.1: Utilization of Instruction Addresses (Nonbranching Instructions)

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	A ¬ B OP C
2	OP A, B	A ¬ A OP B
1	OP A	AC ¬ AC OP A
0	OP	T ¬ (T – 1) OP T

AC = accumulator
T = top of stack
(T – 1) = second element of stack
A, B, C = memory or register locations

Instruction Set Design

Very complex because it affects so many aspects of the computer system

Defines many of the functions performed by the processor

Programmer's means of controlling the processor

Fundamental design issues:

Operation repertoire	Data types	Instruction format	Registers	Addressing
• How many and which operations to provide and how complex operations should be	• The various types of data upon which operations are performed	• Instruction length in bits, number of addresses, size of various fields, etc.	• Number of processor registers that can be referenced by instructions and their use	• The mode or modes by which the address of an operand is specified

Instruction Sets Characteristics and Functions

- Machine Instruction Characteristics
- **Types of Operands**
- Types of Operations
- Intel x86 Data Types
- Addressing Modes
- Instruction Formats

Types of Operands

- Machine instructions operate on data.
- The general categories of data are:
  - Addresses
  - Numbers
  - Characters
  - Logical data

Numbers

- All machine languages include numeric data types
- Numbers stored in a computer are limited, true in two senses:
  1. Limit to the magnitude of numbers representable on a machine
  2. In the case of floating-point numbers, a limit to their precision
- Three types of numerical data are common in computers:
  - Binary integer or binary fixed point
  - Binary floating point
  - Decimal
- Packed decimal
  - Each decimal digit is represented by a 4-bit code with two digits stored per byte
  - To form numbers 4-bit codes are strung together, usually in multiples of 8 bits

Characters

- A common form of data is text or character strings
- Textual data in character form cannot be easily stored or transmitted by data processing and communications systems because they are designed for binary data
- Commonly used character code is the International Reference Alphabet (IRA)
  - Referred to in the United States as the American Standard Code for Information Interchange (ASCII)
- Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC)
  - EBCDIC is used on IBM mainframes

Logical Data

- An *n*-bit unit consisting of ***n*** 1-bit items of data, each item having the value 0 or 1

- Two advantages to bit-oriented view:
  1. Memory can be used efficiently for storing an array of Boolean or binary data items in which each item can take on only the values 1 (true) and 0 (false)
  2. To manipulate the bits of a data item
     - If floating-point operations are implemented in software, we need to be able to shift significant bits in some operations
     - To convert from IRA to packed decimal, we need to extract the rightmost 4 bits of each byte

Instruction Sets Characteristics and Functions

- Machine Instruction Characteristics
- Types of Operands

Types of Operations

- Intel x86 Data Types
- Addressing Modes
- Instruction Formats

Types of Operation

- The number of different opcodes varies from machine to machine.
- The same general types of operations are found on all machines.
- A typical categorization is the following:
  - Data transfer
  - Arithmetic
  - Logical
  - Conversion
  - I/O
  - System control
  - Transfer of control
- Table 12.3 lists common instruction types in each category (page no. 426).
- A survey of various types of operations, together with a brief discussion of the actions taken by the processor to execute a particular type of operation is summarized in Table 12.4.

Table 12.3: Common Instruction Set Operations (page 1 of 2)

Type	Operation Name	Description
Data Transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand
Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT (complement)	Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end

Table 12.3: Common Instruction Set Operations (page 2 of 2)

Type	Operation Name	Description
Transfer of Control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued
Input/Output	Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

Table 12.4: Processor Actions for Various Types of Operations

Data Transfer	Transfer data from one location to another
	If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags
Logical	Same as arithmetic
Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
Transfer of Control	Update program counter. For subroutine call/return, manage parameter passing and linkage
I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

Data Transfer

- The fundamental type of machine instruction is the data transfer instruction.
- The data transfer instruction must specify several things:
  - First, the location of the source and destination operands must be specified. Each location could be memory, a register, or the top of the stack.
  - Second, the length of data to be transferred must be indicated.
  - Third, as with all instructions with operands, the mode of addressing for each operand must be specified.
- The choice of data transfer instructions to include in an instruction set exemplifies the kinds of trade-offs the designer must make.
  - E.g., the general location (memory or register) of an operand can be indicated in either the specification of the opcode or the operand.

Data Transfer

- In terms of processor action, data transfer operations are perhaps the simplest type.
- If both source and destination are registers, then the processor causes data to be transferred from one register to another; this is an operation internal to the processor.
- If one or both operands are in memory, then the processor must perform some or all of the following actions:
  1. Calculate the memory address, based on the address mode.
  2. If the address refers to virtual memory, translate from virtual to real memory address.
  3. Determine whether the addressed item is in cache.
  4. If not, issue a command to the memory module.

Table 12.5
Examples of IBM EAS/390 Data Transfer Operations

Operation Mnemonic	Name	Number of Bits Transferred	Description
L	Load	32	Transfer from memory to register
LH	Load Halfword	16	Transfer from memory to register
LR	Load	32	Transfer from register to register
LER	Load (Short)	32	Transfer from floating-point register to floating-point register
LE	Load (Short)	32	Transfer from memory to floating-point register
LDR	Load (Long)	64	Transfer from floating-point register to floating-point register
LD	Load (Long)	64	Transfer from memory to floating-point register
ST	Store	32	Transfer from register to memory
STH	Store Halfword	16	Transfer from register to memory
STC	Store Character	8	Transfer from register to memory
STE	Store (Short)	32	Transfer from floating-point register to memory
STD	Store (Long)	64	Transfer from floating-point register to memory

Arithmetic

- Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide
- These are provided for signed integer (fixed-point) numbers
- They also provided for floating-point and packed decimal numbers
- Other possible operations include single-operand instructions:
  - Absolute: Take the absolute value of the operand
  - Negate: Negate the operand
  - Increment: Add 1 to the operand
  - Decrement: Subtract 1 from the operand

Logical

- Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable units, referred to as “bit twiddling”.
- A few basic logical operations that can be performed on Boolean or binary data are shown in Table 12.6.
- The NOT operation inverts a bit.
- AND, OR, and Exclusive-OR (XOR) are the common logical functions with two operands.
- EQUAL is a useful binary test.

Table 12.6: Basic Logical Operations

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P=Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

Logical

- The basic operations are illustrated in Figure 12.6.
- With a **logical shift,** the bits of a word are shifted left or right:
  - On one end, the bit shifted out is lost.
  - On the other end, a 0 is shifted in.
  - Logical shifts are useful for isolating fields within a word.
  - The 0s that are shifted into a word displace unwanted information that is shifted off the other end.
- The **arithmetic shift** operation treats the data as a signed integer and does not shift the sign bit:
  - On a right arithmetic shift, the sign bit is replicated into the bit position to its right.
  - On a left arithmetic shift, a logical left shift is performed on all bits but the sign bit, which is retained.
  - These operations can speed up certain arithmetic operations.

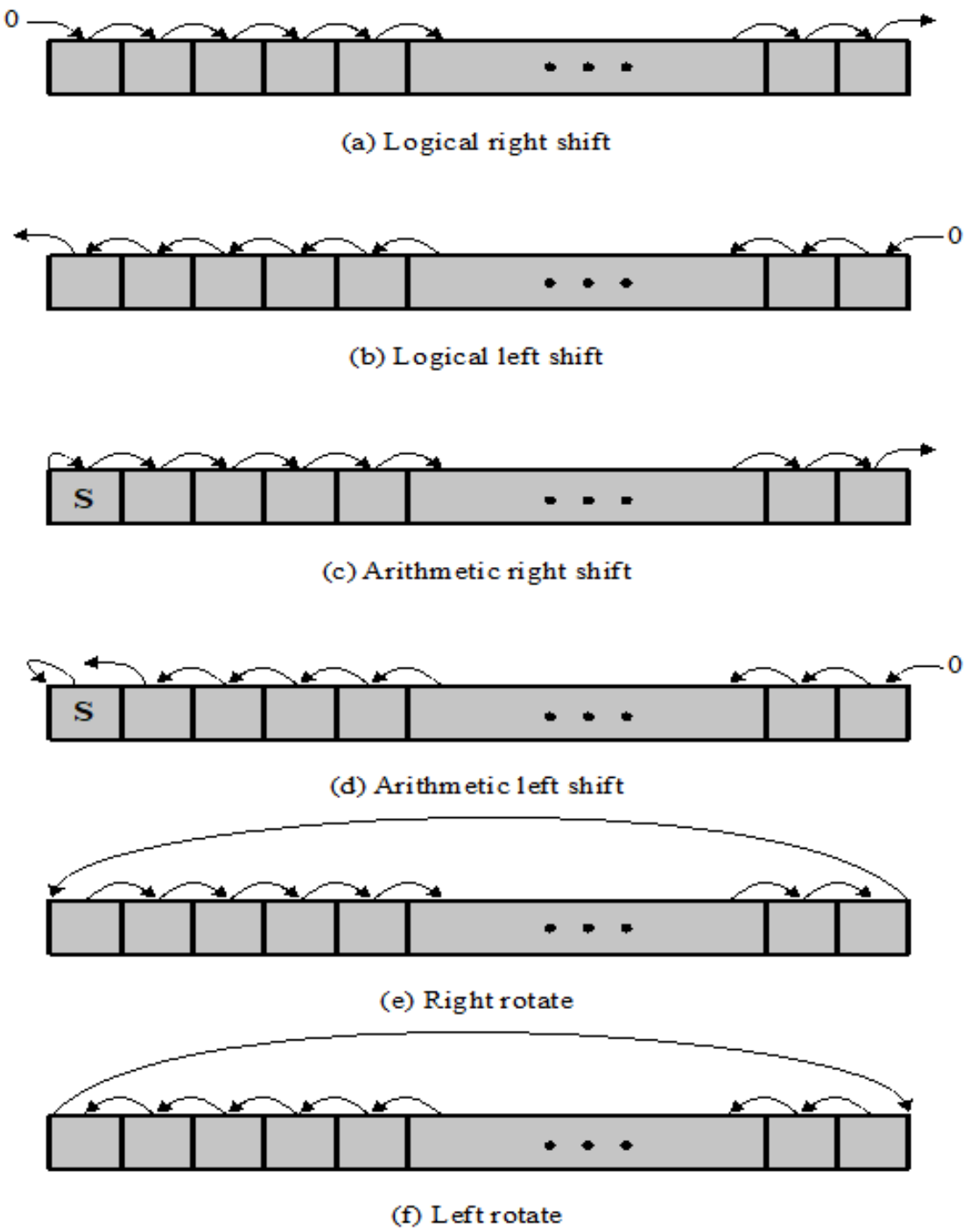


Figure 12.6 Shift and Rotate Operations

Logical

- Table 12.7 gives examples of all of the shift and rotate operations.

Table 12.7: Examples of Shift and Rotate Operations

Input	Operation	Result
10100110	Logical right shift (3 bits)	00010100
10100110	Logical left shift (3 bits)	00110000
10100110	Arithmetic right shift (3 bits)	11110100
10100110	Arithmetic left shift (3 bits)	10110000
10100110	Right rotate (3 bits)	11010100
10100110	Left rotate (3 bits)	00110101

Conversions

- Conversion instructions are those that change the format or operate on the format of data.
- An example is converting from decimal to binary.
- An example of a more complex editing instruction is the EAS/390 Translate (TR) instruction.
- This instruction can be used to convert from one 8-bit code to another, and it takes three operands:

  TR R1 (L), R2
- Locations 10F0 through 10F9 will contain the values 30 through 39, F0 is the EBCDIC code for the digit 0, and 30 is the IRA code for the digit 0, and so on through digit 9.
- Assume the following:
  - Locations 2100–2103 contain F1 F9 F8 F4.
  - R1 contains 2100 and R2 contains 1000.
- Then, if we execute TR R1 (4), R2
  Locations 2100–2103 will contain 31 39 38 34.

Input/Output

- Variety of approaches taken:
  - Isolated programmed I/O
  - Memory-mapped programmed I/O
  - DMA
  - Use of an I/O processor
- Many implementations provide only a few I/O instructions, with the specific actions specified by parameters, codes, or command words

System Control

Instructions that can be executed only while the processor is in a certain privileged state or is executing a program in a special privileged area of memory

These instructions are reserved for the use of the operating system

Examples of system control operations:

- A system control instruction may read or alter a control register
- An instruction to read or modify a storage protection key
- Access to process control blocks in a multiprogramming system

Transfer of Control

- Reasons why transfer-of-control operations are required:
  - It is essential to be able to execute each instruction more than once
  - Virtually all programs involve some decision making
  - It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time
- Common transfer-of-control operations found in instruction sets:
  - Branch
  - Skip
  - Procedure call

Branch Instructions

- A branch or jump instruction, has as one of its operands the address of the next instruction to be executed, the instruction is a **conditional branch** instruction.
- The branch is made only if a certain condition is met, otherwise, the next instruction in sequence is executed (increment program counter).
- A branch instruction in which the branch is always taken is an **unconditional branch.**
- Ways of generating the condition to be tested in a conditional branch instruction.

1. First, most machines provide a 1-bit or multiple-bit condition code that is set as the result of some operations.
   - E.g.: an arithmetic operation (ADD, SUBTRACT, and so on) could set a 2-bit condition code with one of the following four values: 0, positive, negative, overflow.
   - On such a machine, there could be four different conditional branch instructions:
     - BRP X Branch to location X if result is positive.
     - BRN X Branch to location X if result is negative.
     - BRZ X Branch to location X if result is zero.
     - BRO X Branch to location X if overflow occurs.
2. Another approach that can be used with a three-address instruction format is to perform a comparison and specify a branch in the same instruction.
   - For example, BRE R1, R2, X Branch to X if contents of R1 = contents of R2.

- Figure 12.7 shows examples of these operations.

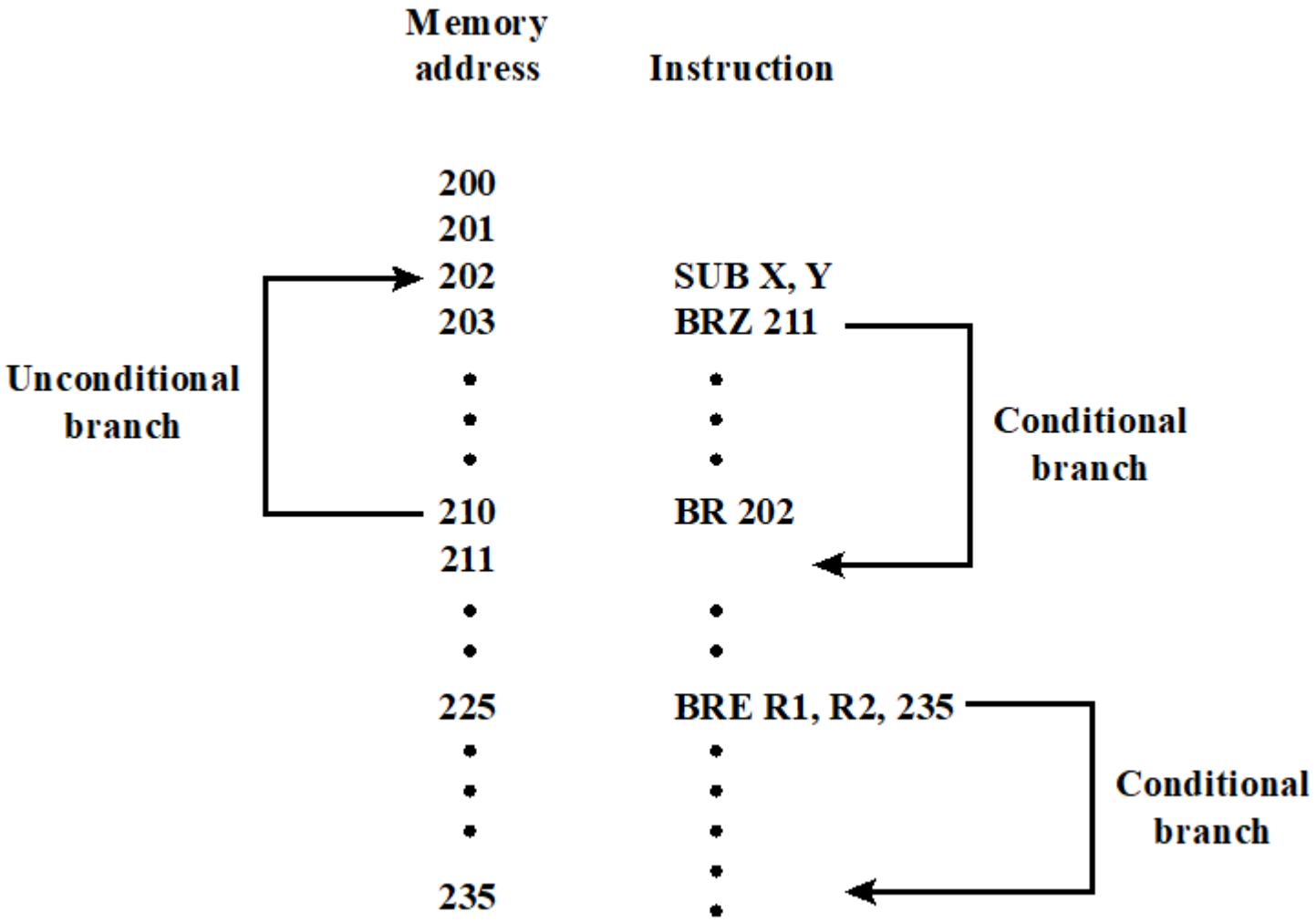
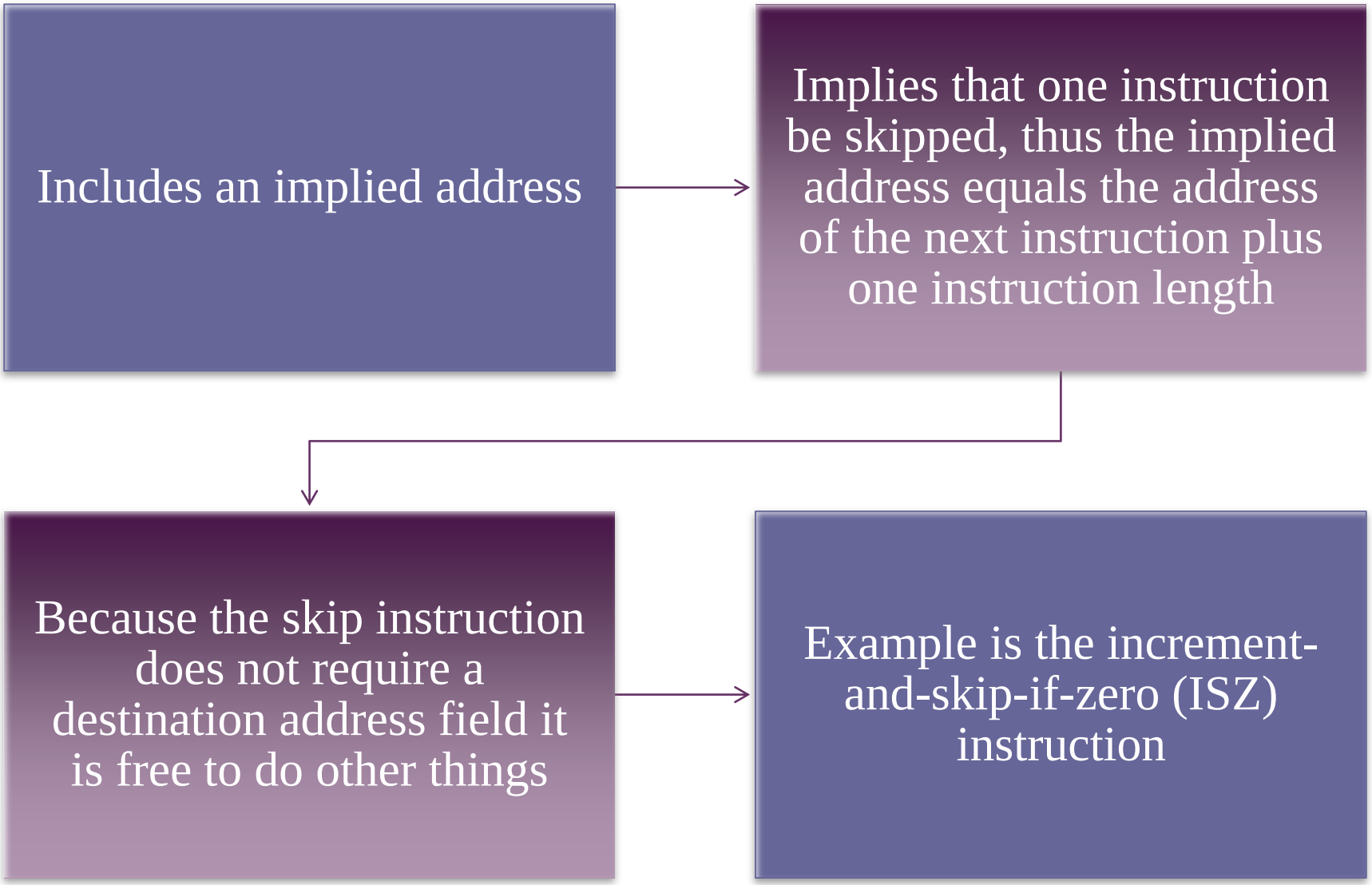


Figure 12.7 Branch Instructions

Skip Instructions



Procedure Call Instructions

- A procedure is a self-contained computer program that is incorporated into a larger program:
  - At any point in the program the procedure may be invoked, or *called*
  - Processor is instructed to go and execute the entire procedure and then return to the point from which the call took place
- Two principal reasons for use of procedures:
  - Economy: A procedure allows the same piece of code to be used many times, making the most efficient use of storage space in the system
  - Modularity: allows large programming tasks to be subdivided into smaller units, eases the programming task
- The procedure mechanism involves two basic instructions:
  - A call instruction that branches from the present location to procedure
  - Return instruction that returns from the procedure to the place from which it was called

- Figure 12.8a illustrates the use of procedures to construct a program.
- This behavior is illustrated in Figure 12.8b.

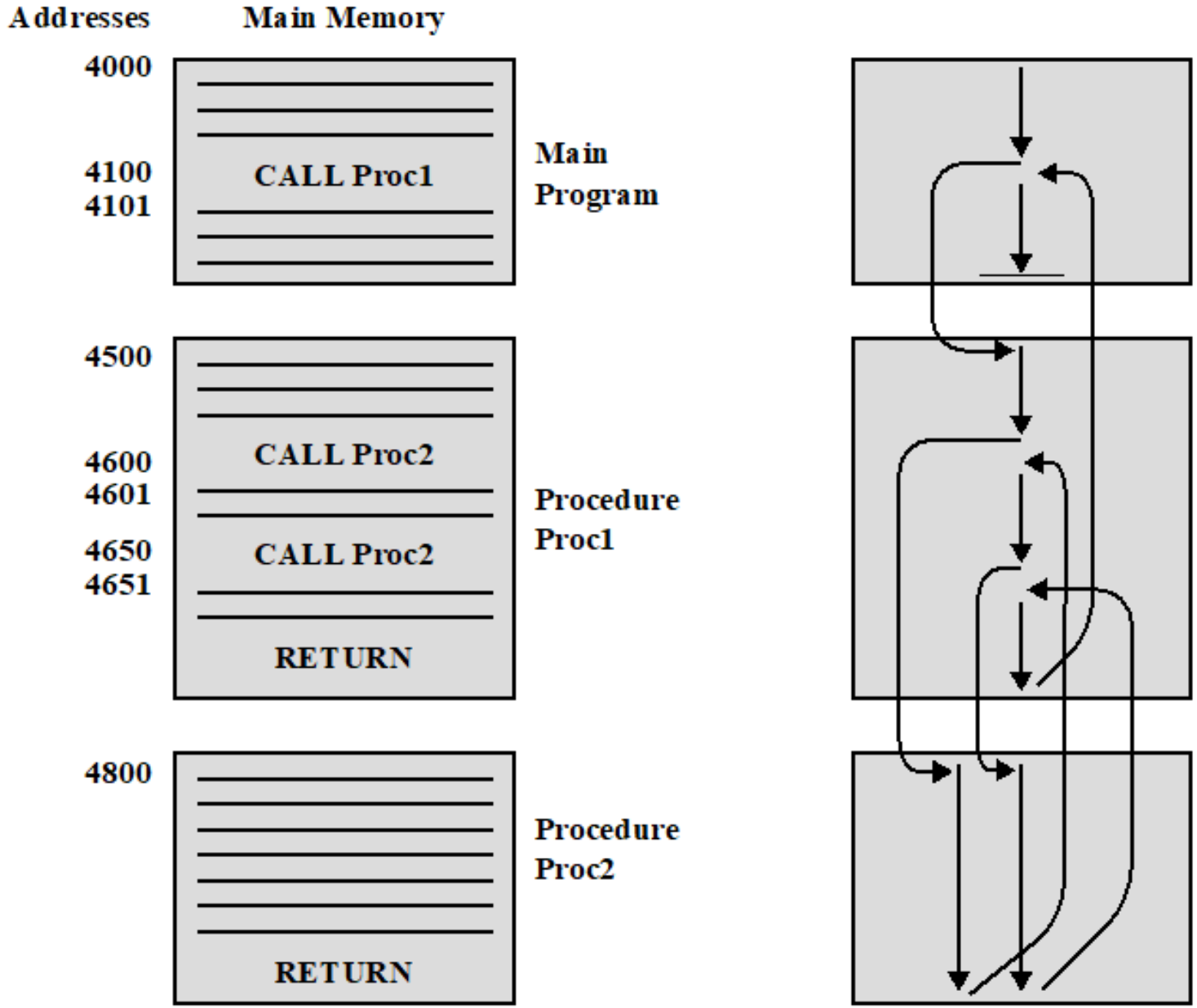


Figure 12.8 Nested Procedures

- A powerful approach is to use a stack, when the processor executes a call, it places the return address on the stack.
- When it executes a return, it uses the address on the stack.
- Figure 12.9 illustrates the use of the stack.
  - In addition to providing a return address, it is also necessary to pass parameters with a procedure call, these can be passed in registers.
  - Store the parameters in memory just after the CALL instruction.
- Both of these approaches have drawbacks.
  - If registers are used, the called program and the calling program must be written to assure that the registers are used properly.
  - The storing of parameters in memory makes it difficult to exchange a variable number of parameters.
- Both approaches prevent the use of reentrant procedures.

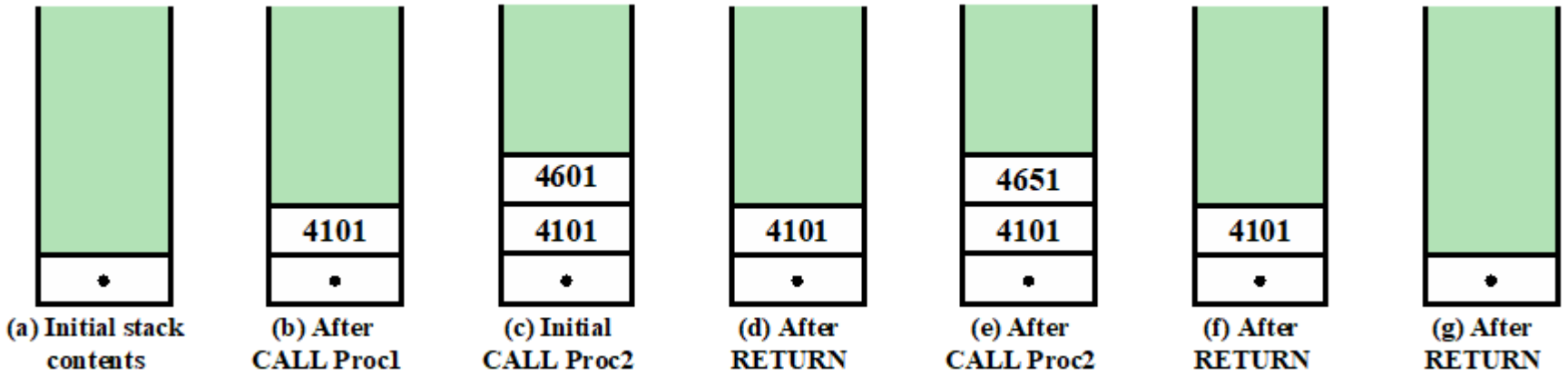


Figure 12.9 Use of Stack to Implement Nested Procedures of Figure 12.8

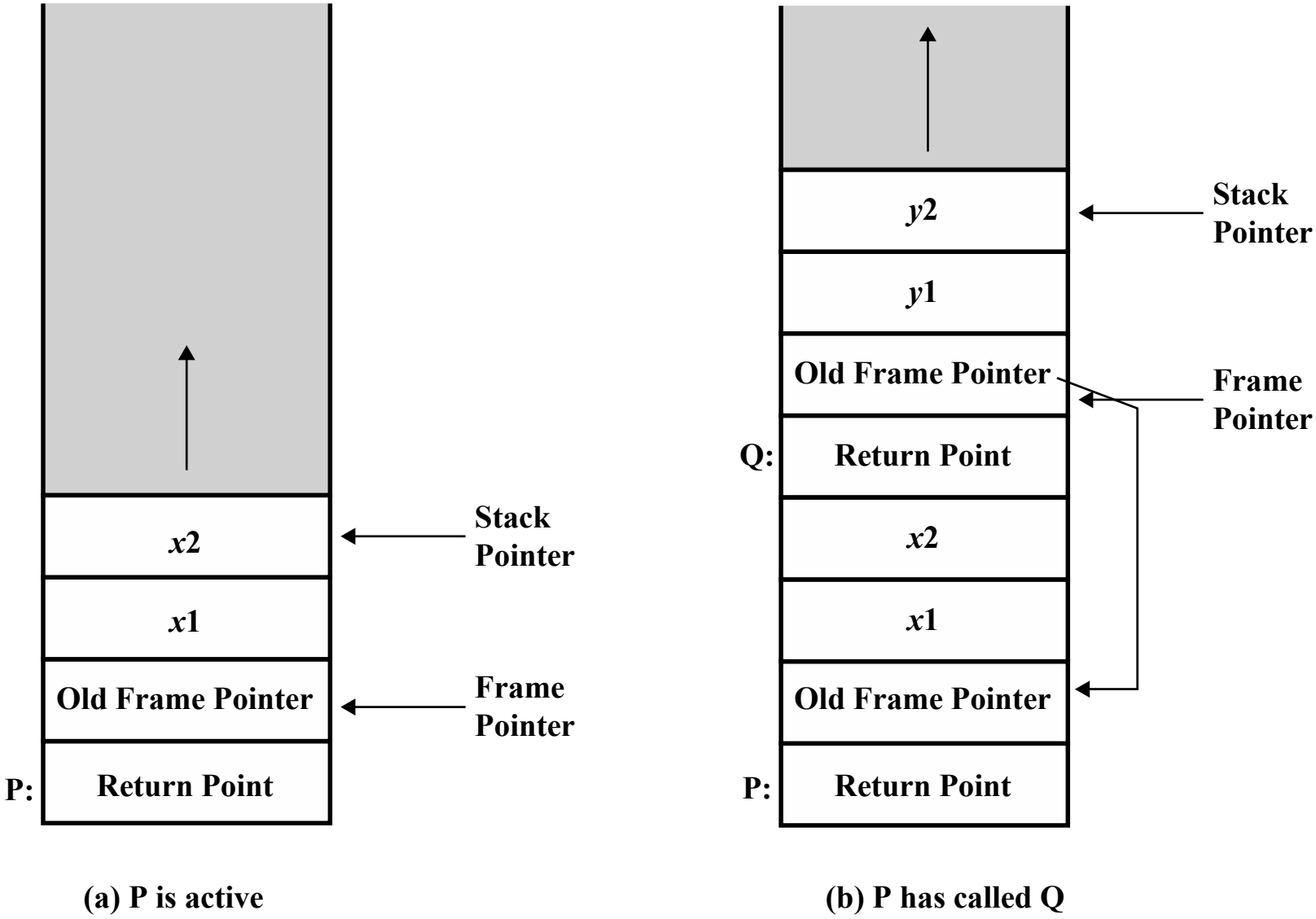


Figure 12.10 Stack Frame Growth Using Sample Procedures P and Q

Instruction Sets Characteristics and Functions

- Machine Instruction Characteristics
- Types of Operands
- Types of Operations
- **Intel x86 Data Types**
- Addressing Modes
- Instruction Formats

INTEL x86 Data Types

- The x86 can deal with data types of 8 (byte), 16 (word), 32 (doubleword), 64 (quad-word), and 128 (double quadword) bits in length.
- When data are accessed across a 32-bit bus, data transfers take place in units of doublewords, starting at addresses divisible by 4.
- The processor converts the request for misaligned values into a sequence of requests for the bus transfer.
- The x86 uses the little-endian style; the LSB is stored in the lowest.
- The byte, word, doubleword, quadword, and double quadword are referred to as general data types.
- In addition, the x86 supports an impressive array of specific data types that are recognized and operated on by particular instructions.
- Table 12.2 summarizes these types.

Little Endian and Big Indian

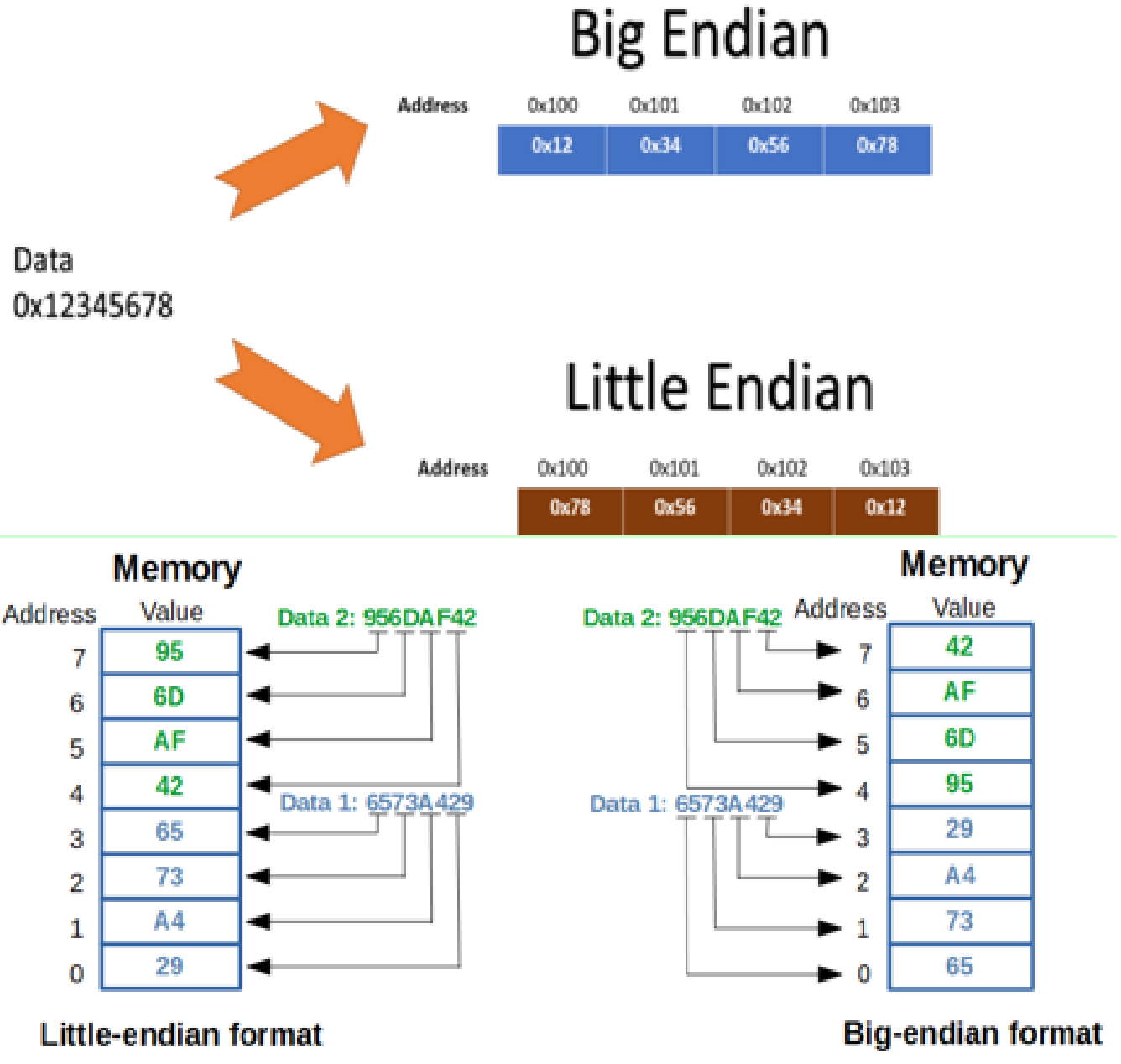


Table 12.2: x86 Data Types

Data Type	Description
General	Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.
Integer	A signed binary value co4ntained in a byte, word, or doubleword, using twos complement representation.
Ordinal	An unsigned integer contained in a byte, word, or doubleword.
Unpacked binary coded decimal (BCD)	A representation of a BCD digit in the range 0 through 9, with one digit in each byte.
Packed BCD	Packed byte representation of two BCD digits; value in the range 0 to 99.
Near pointer	A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.
Far pointer	A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.
Bit field	A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.
Bit string	A contiguous sequence of bits, containing from zero to $2_{32} - 1$ bits.
Byte string	A contiguous sequence of bytes, words, or doublewords, containing from zero to $2_{32} - 1$ bytes.
Floating point	See Figure 12.4.
Packed SIMD (single instruction, multiple data)	Packed 64-bit and 128-bit data types

INTEL x86 Data Types

- Figure 12.4 illustrates the x86 numerical data types.
- The signed integers are in twos complement representation and may be 16, 32, or 64 bits long.
- The floating- point type actually refers to a set of types that are used by the floating-point unit and operated on by floating-point instructions.
- The three floating-point representations conform to the IEEE 754 standard.
- The packed SIMD (single-instruction-multiple-data) data types were introduced to the x86 architecture as part of the extensions of the instruction set to optimize performance of multimedia applications
- These extensions include MMX (multimedia extensions) and SSE (streaming SIMD extensions)

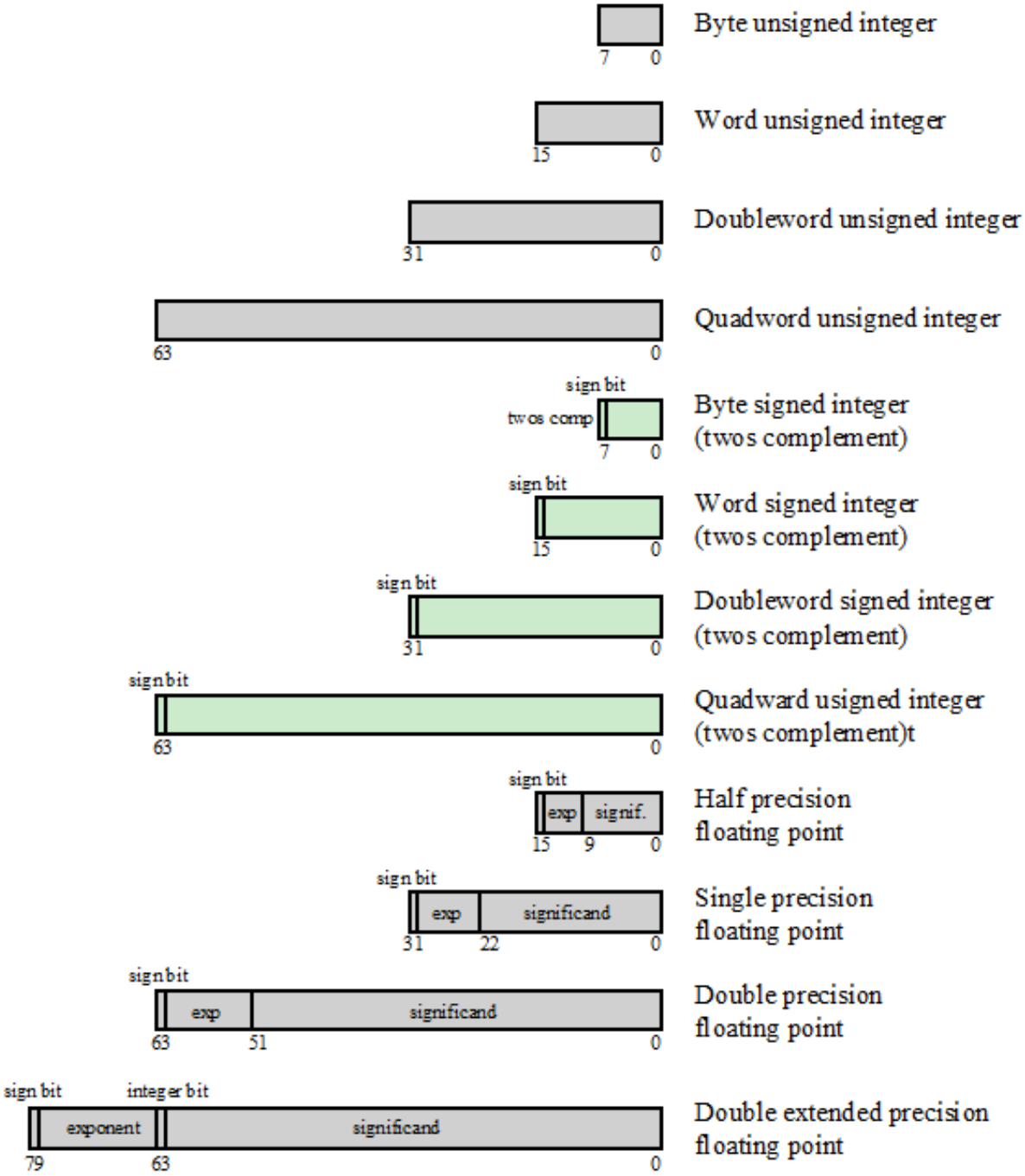


Figure 12.4 x86 Numeric Data Formats

Single-Instruction-Multiple-Data (SIMD) Data Types

- Data types:
  - **Packed byte and packed byte integer:** Bytes packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer
  - **Packed word and packed word integer:** 16-bit words packed into a 64-bit quad- word or 128-bit double quadword, interpreted as a bit field or integer
  - **Packed doubleword and packed doubleword integer:** 32-bit doublewords packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer
  - **Packed quadword and packed qaudword integer:** Two 64-bit quadwords packed into a 128-bit double quadword, interpreted as a bit field or as an integer
  - **Packed single-precision floating-point and packed double-precision floating- point:** Four 32-bit floating-point or two 64-bit floating-point values packed into a 128-bit double quadword.

Summary

- Machine instruction characteristics
  - Elements of a machine instruction
  - Instruction representation
  - Instruction types
  - Number of addresses
  - Instruction set design
- Types of operands
  - Numbers
  - Characters
  - Logical data

Instruction Sets: Characteristics and Functions

- Intel x86 data types
- Types of operations
  - Data transfer
  - Arithmetic
  - Logical
  - Conversion
  - Input/output
  - System control
  - Transfer of control
- Intel x86 operation types

Instruction Sets Characteristics and Functions

- Machine Instruction Characteristics
- Types of Operands
- Types of Operations
- Intel x86 Data Types
- **Addressing Modes**
- Instruction Formats

Addressing Modes

- The most common addressing techniques, or modes are:
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register indirect
  - Displacement
  - Stack
- These modes are illustrated in Figure 13.1.
- Table 13.1 indicates the address calculation performed for each addressing mode.

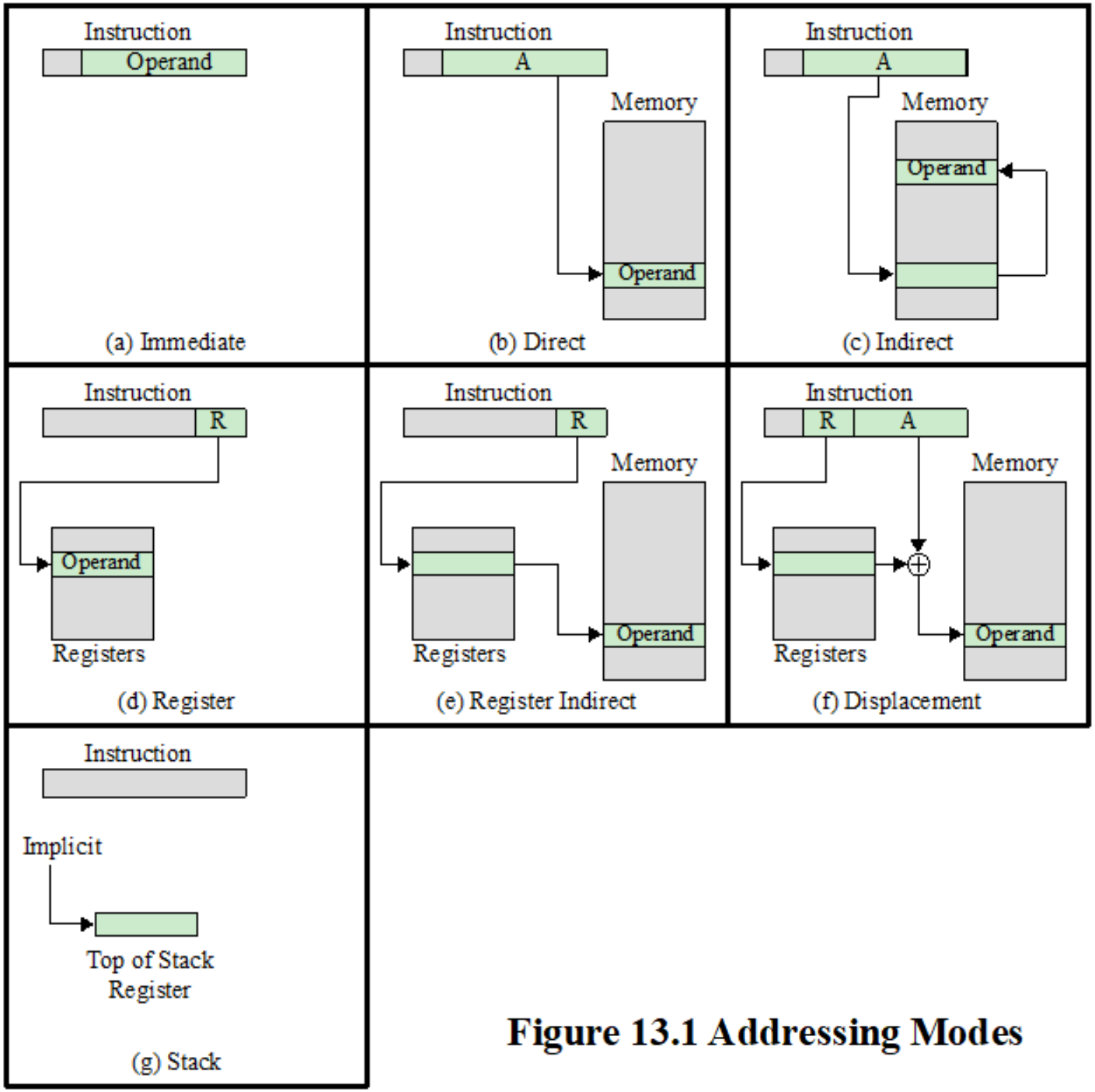


Figure 13.1 Addressing Modes

Table 13.1: Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

A = contents of an address field in the instruction

R = contents of an address field in the instruction that refers to a register

EA = actual (effective) address of the location containing the referenced operand

(X) = contents of memory location X or register X

Immediate Addressing

- Simplest form of addressing

Operand = A

- Used to define and use constants or set initial values of variables:
  - The number will be stored in twos complement form
  - The leftmost bit of the operand field is used as a sign bit
- Advantage:
  - No memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle
- Disadvantage:
  - The size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length

Direct Addressing

- A simple form of addressing is direct addressing, in which the address field contains the effective address of the operand

$$EA = A$$

- The technique was common in earlier generations of computers but is not common on contemporary architectures
- It requires only one memory reference and no special calculation
- The limitation is that it provides only a limited address space

Indirect Addressing

- Reference to the address of a word in memory which contains a full-length address of the operand

$$EA = (A)$$

  - Parentheses are to be interpreted as meaning *contents of*
- Advantage:
  - For a word length of N an address space of 2^N is now available
- Disadvantage:
  - Instruction execution requires 02 memory references to fetch the operand
  - One to get its address and a second to get its value
- Other variant of indirect addressing is multilevel or cascaded

$$EA = (\ldots (A) \ldots)$$

  - Disadvantage is that three or more memory references could be required to fetch an operand

Register Addressing

- **Register addressing** is similar to direct addressing.
- The address field refers to a register rather than a main memory address:

$$EA = R$$

- If the contents of a register address field in an instruction is 5, then register R5 is the intended address, and the operand value is contained in R5.
- An address field that references registers will have from 3 to 5 bits, so that a total of from 8 to 32 general-purpose registers can be referenced.
- Advantages:
  - only a small address field is needed in the instruction
  - no time-consuming memory references are required
- If used in an instruction set, the processor registers will be heavily used.
- Because of the limited number of registers their use in this fashion makes sense only if they are employed efficiently.
- Disadvantage: the address space is very limited

Register Indirect Addressing

- Analogous to indirect addressing
- The only difference is whether the address field refers to a memory location or a register

$$EA = (R)$$

- Address space limitation of the address field is overcome by having that field refer to a word-length location containing an address
- Uses one less memory reference than indirect addressing

Displacement Addressing

- Combines the capabilities of direct addressing and register indirect addressing

$$EA = A + (R)$$

- Requires that the instruction have two address fields, at least one of which is explicit:
  - The value contained in one address field (value = A) is used directly
  - The other address field refers to a register whose contents are added to A to produce the effective address
- Most common uses:
  - Relative addressing
  - Base-register addressing
  - Indexing

Relative Addressing

The implicitly referenced register is the program counter (PC)

- The next instruction address is added to the address field to produce the EA
- The address field is treated as a twos complement number for this operation
- The effective address is a displacement relative to the address of the instruction

Exploits the concept of locality

Saves address bits in the instruction if most memory references are relatively near to the instruction being executed

Base-Register Addressing

- The referenced register contains a main memory address and the address field contains a displacement from that address
- The register reference may be explicit or implicit
- Exploits the locality of memory references
- Convenient means of implementing segmentation
- In some implementations a single segment base register is employed and is used implicitly
- In others the programmer may choose a register to hold the base address of a segment and the instruction must reference it explicitly

Indexing

- The address field references a main memory address and referenced register contains a positive displacement from that address
- The method of calculating the EA is the same as for base-register addressing
- An important use is to provide an efficient mechanism for performing iterative operations
- Autoindexing: Automatically increment or decrement a index register after each reference to it

$$EA = A + (R)$$

$$(R) \leftarrow (R) + 1$$

- Postindexing: Indexing is performed after the indirection

$$EA = (A) + (R)$$

- Preindexing: Indexing is performed before the indirection

$$EA = (A + (R))$$

Stack Addressing

- A stack is a linear array of locations, sometimes referred to as a *pushdown list* or *last-in-first-out queue*
- A stack is a reserved block of locations
  - Items are appended to the top of the stack so that the block is partially filled
- Associated with the stack is a pointer whose value is the address of the top of the stack:
  - The stack pointer is maintained in a register
  - Thus references to stack locations in memory are in fact register indirect addresses
- The stack mode of addressing is a form of implied addressing
- The machine instructions need not include a memory reference but implicitly operate on the top of the stack

X86 Addressing Modes

- The x86 has variety of addressing modes intended to allow the efficient execution of high-level languages.
- Figure 13.2 indicates the logic involved.
- The segment register determines the segment that is the subject of the reference.
- There are six segment registers; the one being used for a particular reference depends on the context of execution and the instruction.
- Each segment register holds an index into the segment descriptor table, which holds starting address of the corresponding segments.
- Associated with each user-visible segment register is a segment descriptor register, which records the access rights for the segment as well as the starting address and limit (length) of the segment.
- In addition, there are two registers that may be used in constructing an address: the base register and the index register.
- Table 13.2 lists the x86 addressing modes.

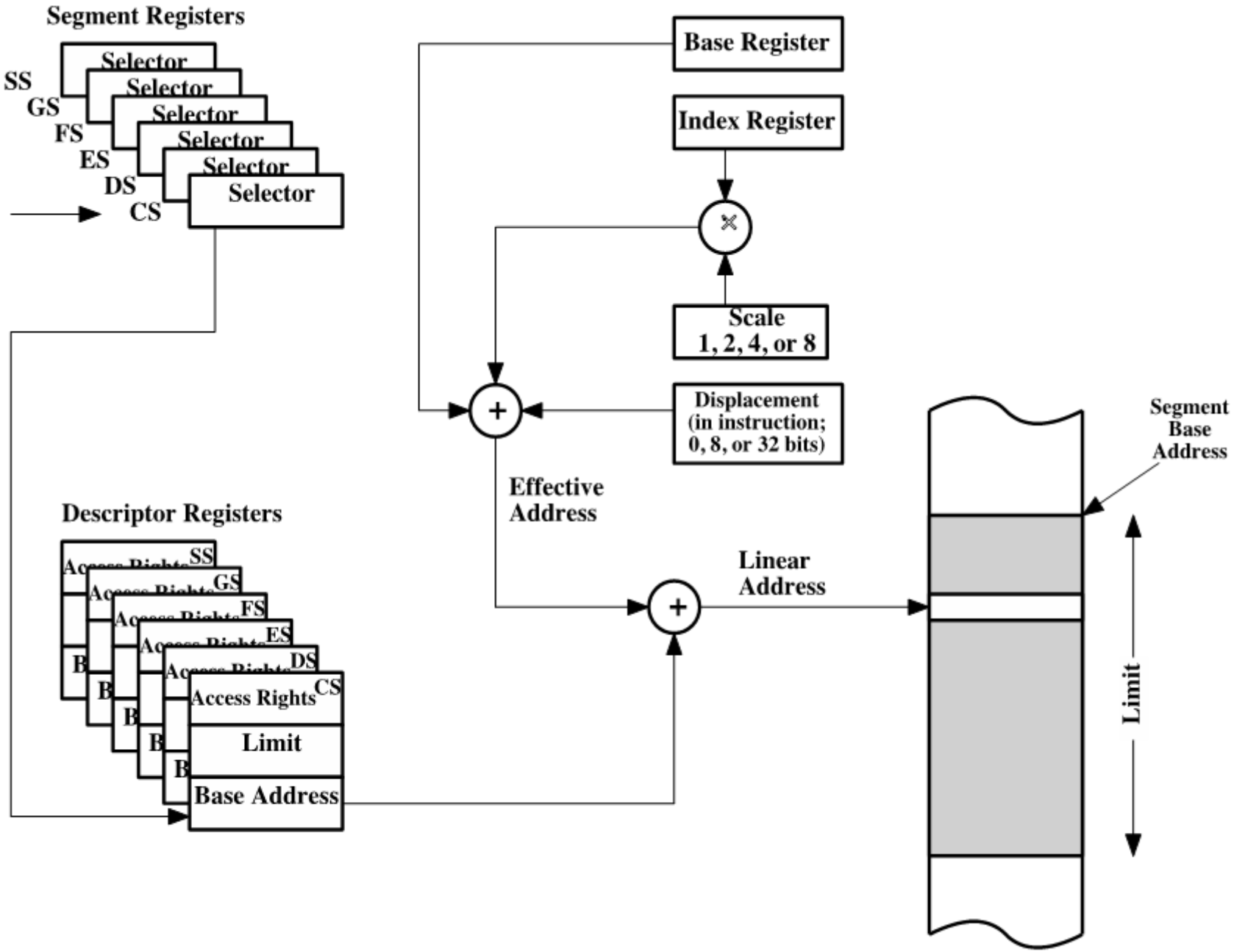


Figure 13.2 x86 Addressing Mode Calculation

Table 13.2: x86 Addressing Modes

Mode	Algorithm
Immediate	Operand = A
Register Operand	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) ´ S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) ´ S + (B) + A
Relative	LA = (PC) + A

LA = linear address
(X) = contents of X
SR = segment register
PC = program counter
A = contents of an address field in the instruction
R = register
B = base register
I = index register
S = scaling factor

Instruction Sets Characteristics and Functions

- Machine Instruction Characteristics
- Types of Operands
- Types of Operations
- Intel x86 Data Types
- Addressing Modes

- **Instruction Formats**

Instruction Formats

- An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields.
- An instruction format must include an opcode and, implicitly or explicitly, zero or more operands.
- Each explicit operand is referenced using one of the addressing modes.
- The format must, implicitly or explicitly, indicate the addressing mode for each operand.
- For most instruction sets, more than one instruction format is used.
- The design of an instruction format is a complex art, and an amazing variety of designs have been implemented.

Instruction Length

- The basic design issue faced is the instruction format length.
- Affected by:
  - Memory size
  - Memory organization
  - Bus structure
  - Processor complexity
  - Processor speed
- Should be equal to the memory-transfer length or one should be a multiple of the other
- Should be a multiple of the character length, which is usually 8 bits, and of the length of fixed-point numbers

Allocation of Bits

- Number of addressing modes
- Number of operands
- Register versus memory
- Number of register sets
- Address range
- Address granularity

Variable-Length Instructions

- Variations can be provided efficiently and compactly
- Increases the complexity of the processor
- Does not remove the desirability of making all of the instruction lengths integrally related to word length:
  - Because the processor does not know the length of the next instruction to be fetched a typical strategy is to fetch a number of bytes or words equal to at least the longest possible instruction
  - Sometimes multiple instructions are fetched

x86 Instructions Formats

- The x86 is equipped with a variety of instruction formats.
- Figure 13.9 illustrates the general x86 instruction format.
- Instructions are made up of from zero to four optional instruction prefixes, a 1- or 2-byte opcode, an optional address specifier an optional displacement, and an optional immediate field.
- The prefix bytes:
  - Instruction prefixes
  - Segment override
  - Operand size
  - Address size
- The instruction itself includes the following fields:
  - Opcode
  - ModR/M (Operand is in a Register or in Memory)
  - SIB (Scale Index Base)
  - Displacement
  - Immediate

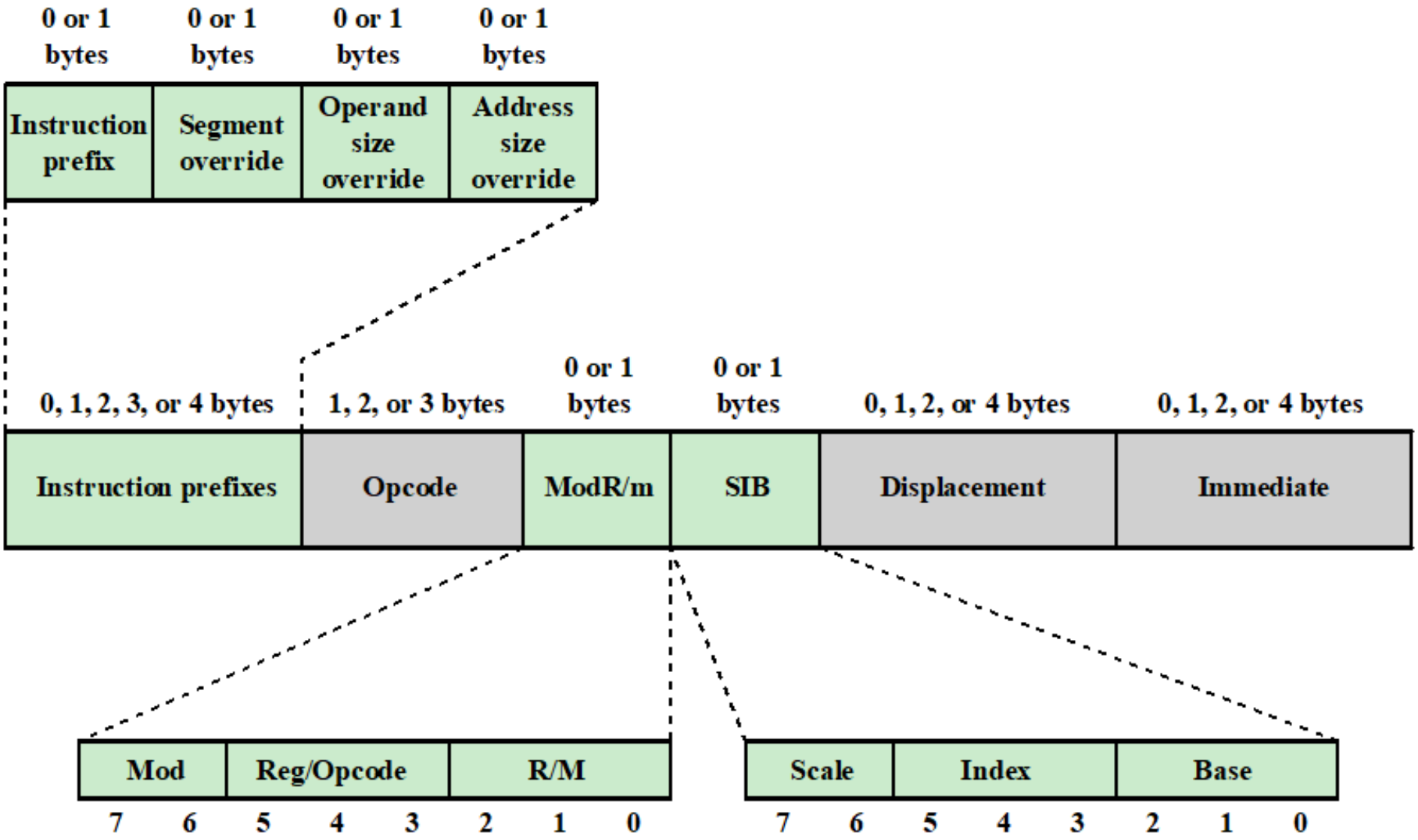


Figure 13.9 x86 Instruction Format

Summary

- Addressing modes
  - Immediate addressing
  - Direct addressing
  - Indirect addressing
  - Register addressing
  - Register indirect addressing
  - Displacement addressing
  - Stack addressing

Instruction Sets: Addressing Modes and Formats

- x86 addressing modes
- Instruction formats
  - Instruction length
  - Allocation of bits
  - Variable-length instructions
- X86 instruction formats

x86 Operation Types

- The x86 provides a complex array of operation types including a number of specialized instructions
- Provide tools for the compiler writer to produce optimized machine language translation of high-level language programs
- Provides four instructions to support procedure call/return:
  - CALL
  - ENTER
  - LEAVE
  - RETURN
- When a new procedure is called the following must be performed upon entry to the new procedure:
  - Push the return point on the stack
  - Push the current frame pointer on the stack
  - Copy the stack pointer as the new value of the frame pointer
  - Adjust the stack pointer to allocate a frame

Table 12.8: x86 Status Flags

Status Bit	Name	Description
CF	Carry	Indicates carrying or borrowing out of the left-most bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.
PF	Parity	Parity of the least-significant byte of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.
AF	Auxiliary Carry	Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation. Used in binary-coded decimal arithmetic.
ZF	Zero	Indicates that the result of an arithmetic or logic operation is 0.
SF	Sign	Indicates the sign of the result of an arithmetic or logic operation.
OF	Overflow	Indicates an arithmetic overflow after an addition or subtraction for twos complement arithmetic.

Table 12.9: x86 Condition Codes for Conditional Jump and SETcc Instructions

Symbol	Condition Tested	Comment
A, NBE	CF=0 AND ZF=0	Above; Not below or equal (greater than, unsigned)
AE, NB, NC	CF=0	Above or equal; Not below (greater than or equal, unsigned); Not carry
B, NAE, C	CF=1	Below; Not above or equal (less than, unsigned); Carry set
BE, NA	CF=1 OR ZF=1	Below or equal; Not above (less than or equal, unsigned)
E, Z	ZF=1	Equal; Zero (signed or unsigned)
G, NLE	[(SF=1 AND OF=1) OR (SF=0 and OF=0)] AND [ZF=0]	Greater than; Not less than or equal (signed)
GE, NL	(SF=1 AND OF=1) OR (SF=0 AND OF=0)	Greater than or equal; Not less than (signed)
L, NGE	(SF=1 AND OF=0) OR (SF=0 AND OF=1)	Less than; Not greater than or equal (signed)
LE, NG	(SF=1 AND OF=0) OR (SF=0 AND OF=1) OR (ZF=1)	Less than or equal; Not greater than (signed)
NE, NZ	ZF=0	Not equal; Not zero (signed or unsigned)
NO	OF=0	No overflow
NS	SF=0	Not sign (not negative)
NP, PO	PF=0	Not parity; Parity odd
O	OF=1	Overflow
P	PF=1	Parity; Parity even
S	SF=1	Sign (negative)

Table 12.10: MMX Instruction Set

Category	Instruction	Description
Arithmetic	PADD [B, W, D]	Parallel add of packed eight bytes, four 16-bit words, or two 32-bit doublewords, with wraparound.
	PADDS [B, W]	Add with saturation.
	PADDUS [B, W]	Add unsigned with saturation
	PSUB [B, W, D]	Subtract with wraparound.
	PSUBS [B, W]	Subtract with saturation.
	PSUBUS [B, W]	Subtract unsigned with saturation
	PMULHW	Parallel multiply of four signed 16-bit words, with high-order 16 bits of 32-bit result chosen.
	PMULLW	Parallel multiply of four signed 16-bit words, with low-order 16 bits of 32-bit result chosen.
	PMADDWD	Parallel multiply of four signed 16-bit words; add together adjacent pairs of 32-bit results.
Comparison	PCMPEQ [B, W, D]	Parallel compare for equality; result is mask of 1s if true or 0s if false.
	PCMPGT [B, W, D]	Parallel compare for greater than; result is mask of 1s if true or 0s if false.
Conversion	PACKUSWB	Pack words into bytes with unsigned saturation.
	PACKSS [WB, DW]	Pack words into bytes, or doublewords into words, with signed saturation.
	PUNPCKH [BW, WD, DQ]	Parallel unpack (interleaved merge) high-order bytes, words, or doublewords from MMX register.
	PUNPCKL [BW, WD, DQ]	Parallel unpack (interleaved merge) low-order bytes, words, or doublewords from MMX register.
Logical	PAND	64-bit bitwise logical AND
	PNDN	64-bit bitwise logical AND NOT
	POR	64-bit bitwise logical OR
	PXOR	64-bit bitwise logical XOR
Shift	PSLL [W, D, Q]	Parallel logical left shift of packed words, doublewords, or quadword by amount specified in MMX register or immediate value.
	PSRL [W, D, Q]	Parallel logical right shift of packed words, doublewords, or quadword.
	PSRA [W, D]	Parallel arithmetic right shift of packed words, doublewords, or quadword.
Data Transfer	MOV [D, Q]	Move doubleword or quadword to/from MMX register.
State Mgt	EMMS	Empty MMX state (empty FP registers tag bits).

x86 SIMD Instructions

- 1996 Intel introduced MMX technology into its Pentium product line
  - MMX is a set of highly optimized instructions for multimedia tasks
- Video and audio data are typically composed of large arrays of small data types
- Three new data types are defined in MMX
  - Packed byte
  - Packed word
  - Packed doubleword
- Each data type is 64 bits in length and consists of multiple smaller data fields, each of which holds a fixed-point integer

Memory Reference Instructions

Opcode			D/I	Z/C	Displacement	
0		2	3	4	5	11

Input/Output Instructions

1	1	0	Device		Opcode	
0		2	3	8	9	11

Register Reference Instructions

Group 1 Microinstructions

1	1	1	0	CLA	CLL	CMA	CML	RAR	RAL	BSW	IAC
0	1	2	3	4	5	6	7	8	9	10	11

Group 2 Microinstructions

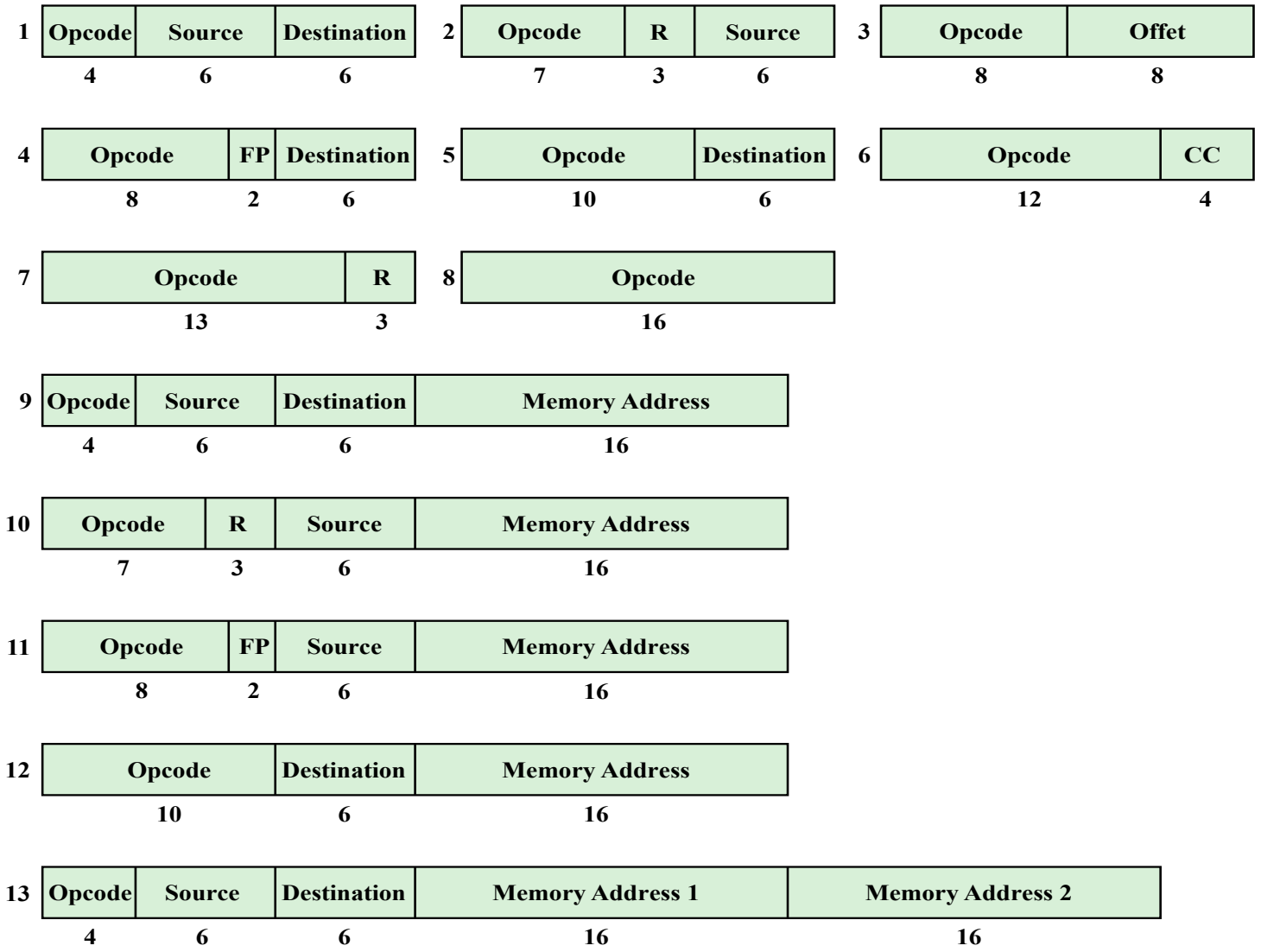
1	1	1	1	CLA	SMA	SZA	SNL	RSS	OSR	HLT	0
0	1	2	3	4	5	6	7	8	9	10	11

Group 3 Microinstructions

1	1	1	1	CLA	MQA	0	MQL	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11

D/I = Direct/Indirect address
Z/C = Page 0 or Current page
CLA = Clear Accumulator
CLL = Clear Link
CMA = CoMplement Accumulator
CML = CoMplement Link
RAR = Rotate Accumultator Right
RAL = Rotate Accumulator Left
BSW = Byte SWap
IAC = Increment ACcumulator
SMA = Skip on Minus Accumulator
SZA = Skip on Zero Accumulator
SNL = Skip on Nonzero Link
RSS = Reverse Skip Sense
OSR = Or with Switch Register
HLT = HaLT
MQA = Multiplier Quotient into Accumulator
MQL = Multiplier Quotient Load

Figure 13.5 PDP-8 Instruction Formats



Numbers below fields indicate bit length
Source and Destination each contain a 3-bit addressing mode field and a 3-bit register number
FP indicates one of four floating-point registers
R indicates one of the general-purpose registers
CC is the condition code field

Figure 13.7 Instruction Formats for the PDP-11

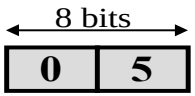
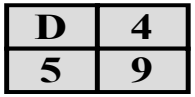
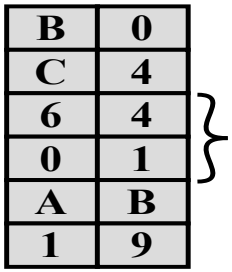
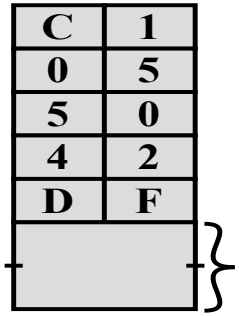
Hexadecimal Format	Explanation	Assembler Notation and Description
	Opcode for RSB	RSB Return from subroutine
	Opcode for CLRL Register R9	CLRL R9 Clear register R9
	Opcode for MOVW Word displacement mode, Register R4 356 in hexadecimal Byte displacement mode, Register R11 25 in hexadecimal	MOVW 356(R4), 25(R11) Move a word from address that is 356 plus contents of R4 to address that is 25 plus contents of R11
	Opcode for ADDL3 Short literal 5 Register mode R0 Index prefix R2 Indirect word relative (displacement from PC) Amount of displacement from PC relative to location A	ADDL3 #5, R0, @A[R2] Add 5 to a 32-bit integer in R0 and store the result in location whose address is sum of A and 4 times the contents of R2

Figure 13.8 Examples of VAX Instructions

Processor Structure and Function

- **Processor Organization**
- **Register Organization**
  - Instruction Cycle
  - Instruction Pipelining

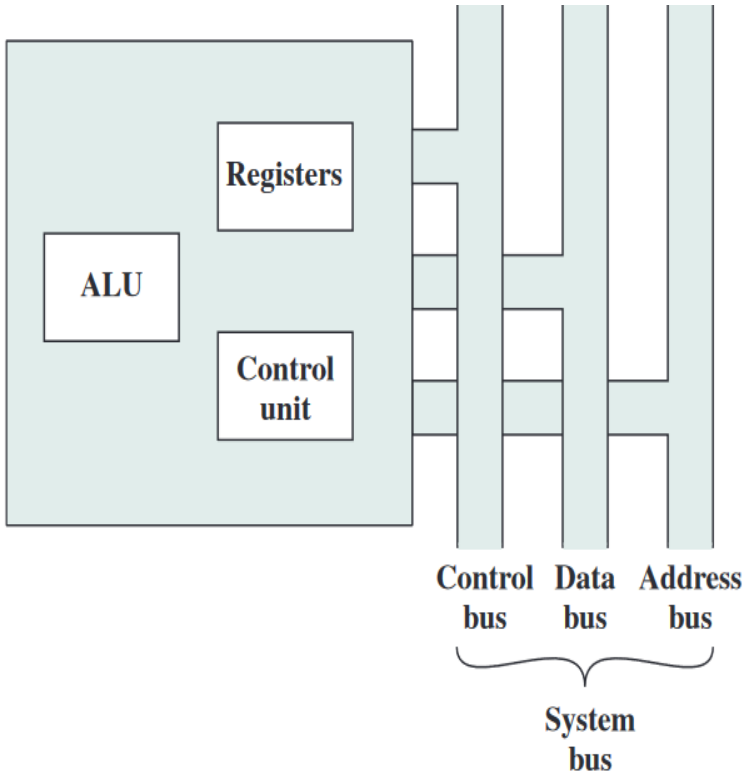
Processor Organization

Processor Requirements:

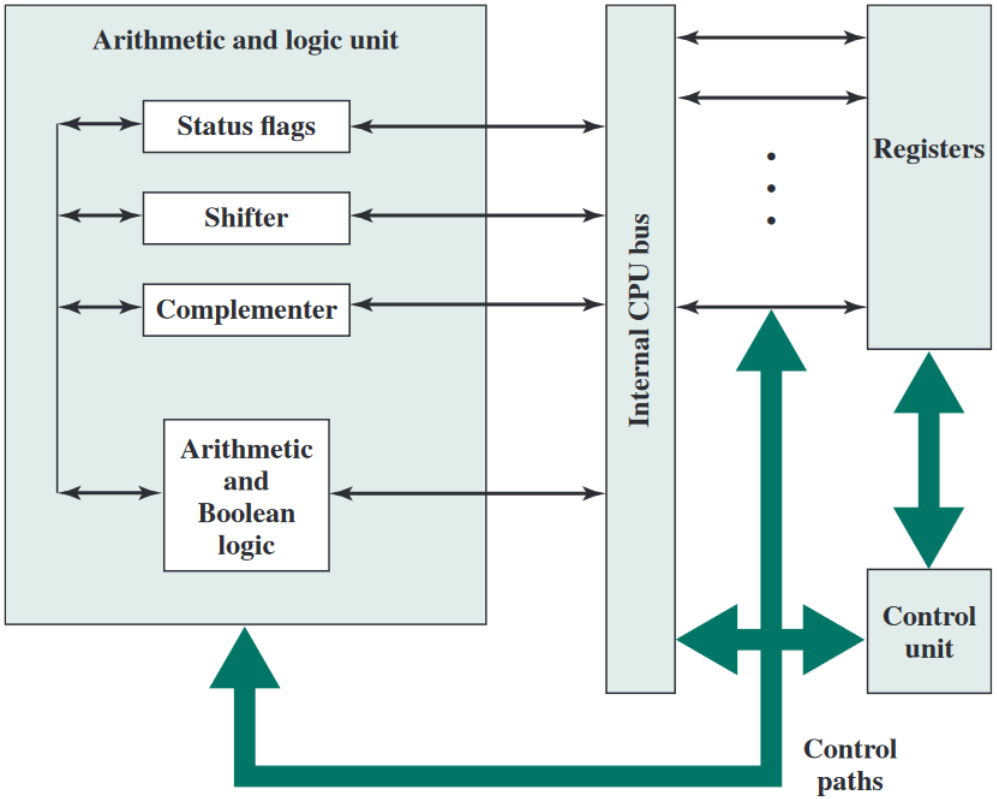
- **Fetch instruction:** The processor reads an instruction from memory (register, cache, main memory).
- **Interpret instruction:** The instruction is decoded to determine what action is required.
- **Fetch data:** The execution of an instruction may require reading data from memory or an I/O module.
- **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.

- Do these things the processor needs to store some data temporarily and therefore needs a small internal memory.

Internal Structure of a Processor



CPU with the System Bus



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: 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. 2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST and BRANCH. 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. 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. 2. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections. 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. 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.

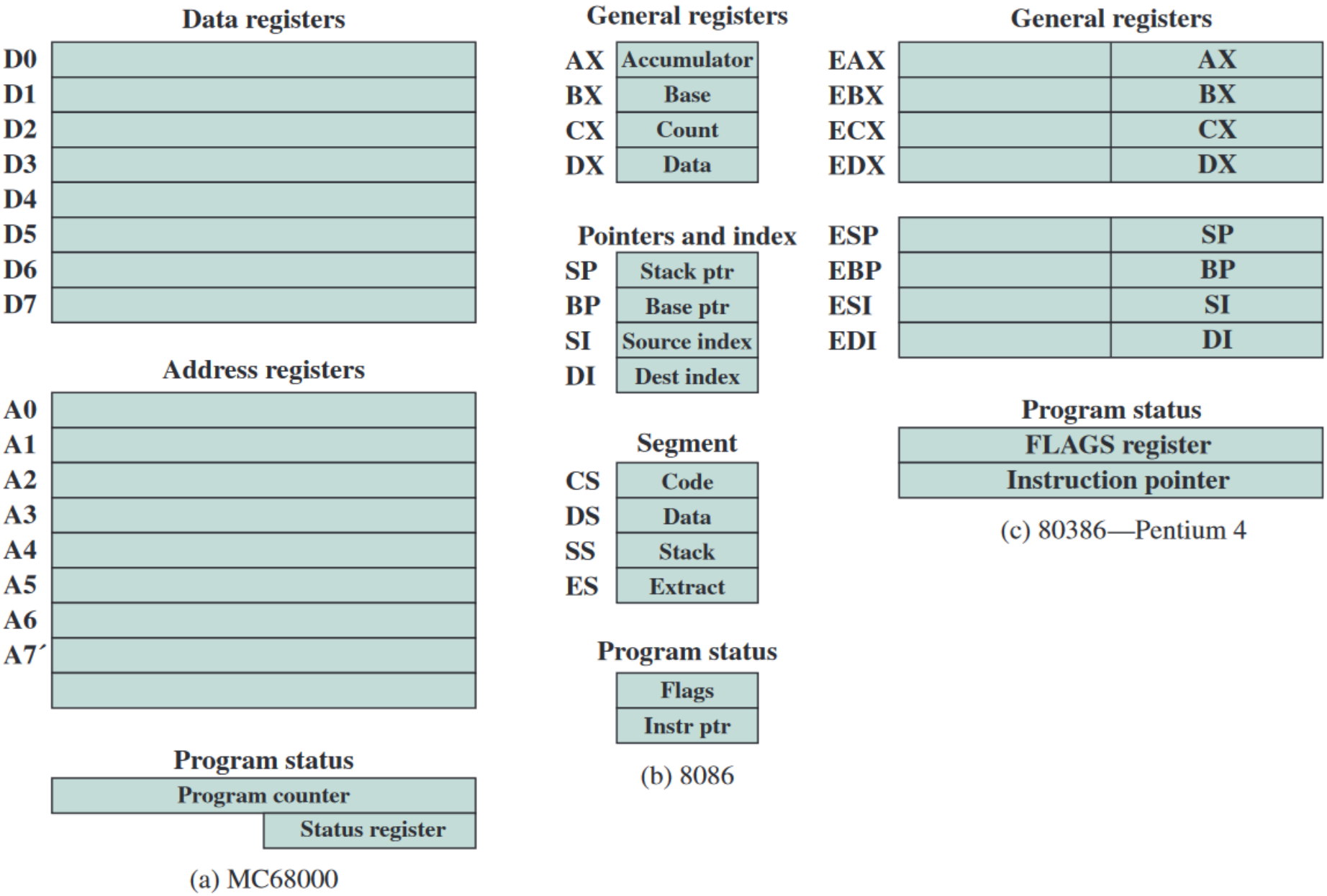
Program Status Word (PSW)

- Processor designs include a register or set of registers, often known as the *program status word* (PSW), that contain status information.
- The PSW contains condition codes plus other status information.
- Common fields or flags include the following:
  - **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.

Microprocessor Register Organizations

- Consider two 16-bit microprocessors: the Motorola MC68000 and the Intel 8086.
- Figures (a) and (b) depict the register organization of each; internal registers, such as a memory address register, are not shown.
- The MC68000 partitions its 32-bit registers into eight data registers and nine address registers.
- The eight data registers are used primarily for data manipulation and are also used in addressing as index registers.
- The registers allows 8-, 16-, and 32-bit data operations, determined by opcode.
- The address registers contain 32-bit (no segmentation) addresses; two of these registers are also used as stack pointers, one for users and one for the operating system, depending on the current execution mode.
- Both registers are numbered 7, because only one can be used at a time.
- The MC68000 also includes a 32-bit program counter and a 16-bit status register.
- Motorola team wanted a regular instruction set, with no special-purpose registers.
- A concern for code efficiency led them to divide the registers into wo functional components, saving one bit on each register specifier.
- This is a compromise between complete generality and code compaction.

Microprocessor Register Organizations



Microprocessor Register Organizations

- The Intel 8086 takes a different approach to register organization.
- Every register is special purpose, although some registers are also usable as general purpose.
- The 8086 contains four 16-bit data registers that are addressable on a byte or 16-bit basis, and four 16-bit pointer and index registers.
- The data registers can be used as general purpose in some instructions.
- The registers are used implicitly.
- The four pointer registers are also used implicitly in a number of operations; each contains a segment offset.
- There are four 16-bit segment registers:
  - Three of the four segment registers are used in a dedicated, implicit fashion, to point to the segment of the current instruction, a segment containing data, and a segment containing a stack, respectively.
  - The dedicated and implicit uses provide for compact encoding at the cost of reduced flexibility.
- 8086 also includes an instruction pointer and a set of 1-bit status and control flags.

Microprocessor Register Organizations

- A register organization design is illustrated in Figure (c).
- Figure shows the user-visible register organization for the Intel 80386, which is a 32-bit microprocessor designed as an extension of the 8086.
- The 80386 uses 32-bit registers.
- To provide upward compatibility for programs written on the earlier machine, the 80386 retains the original register organization embedded in the new organization.
- The architects of the 32-bit processors had limited flexibility in designing the register organization.

Processor Structure and Function

- Processor Organization
- Register Organization
- **Instruction Cycle**
- Instruction Pipelining

Instruction Cycle

An 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.

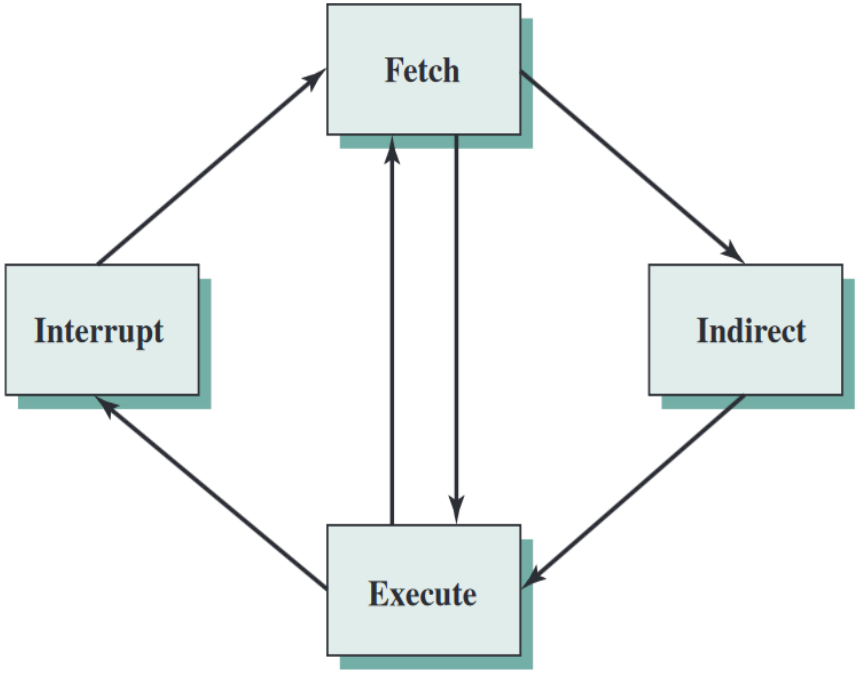


Figure The Instruction Cycle

- The fetching of indirect addresses as one more instruction stages, as shown in Figure.

Instruction Cycle State Diagram

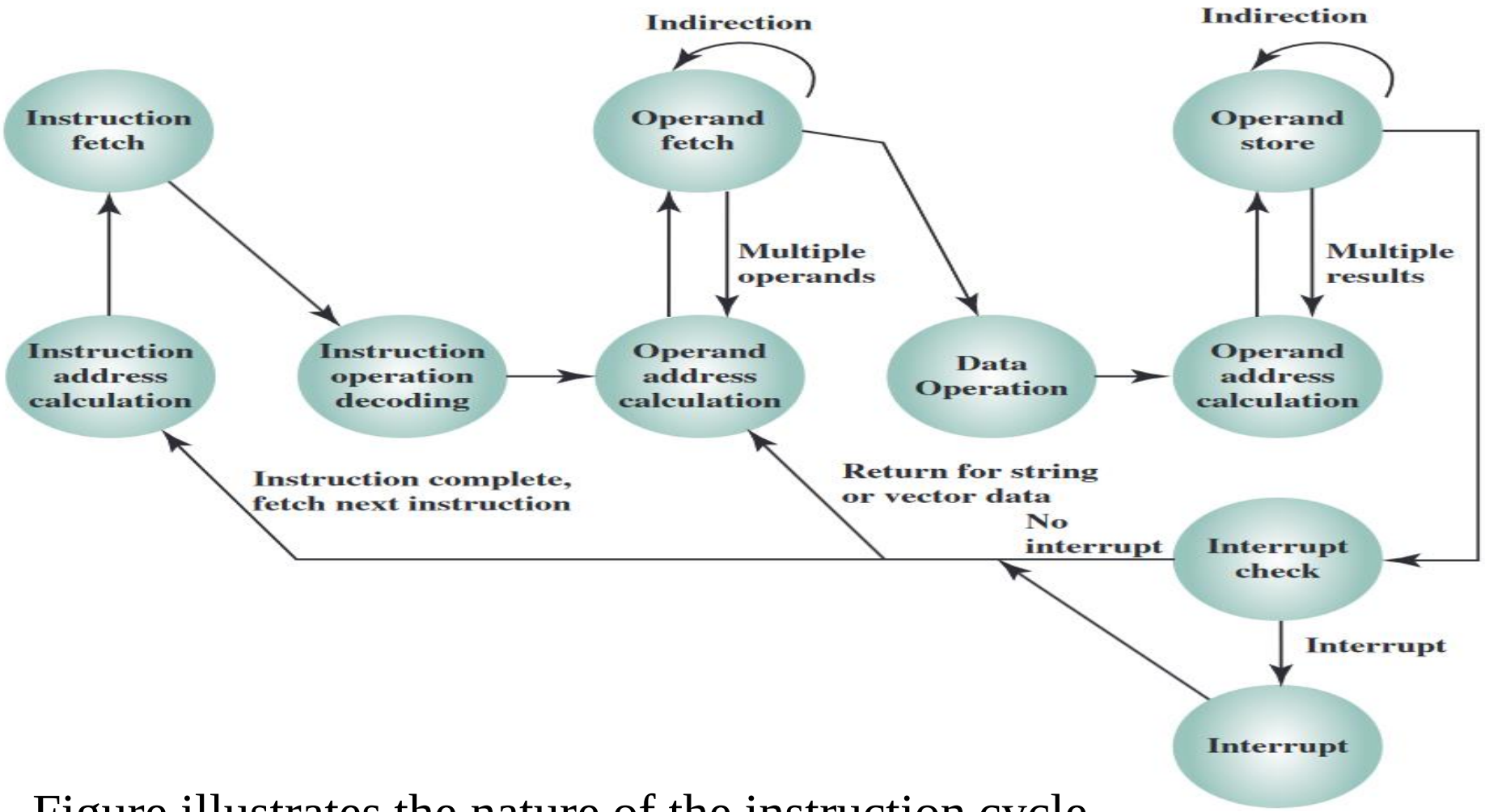
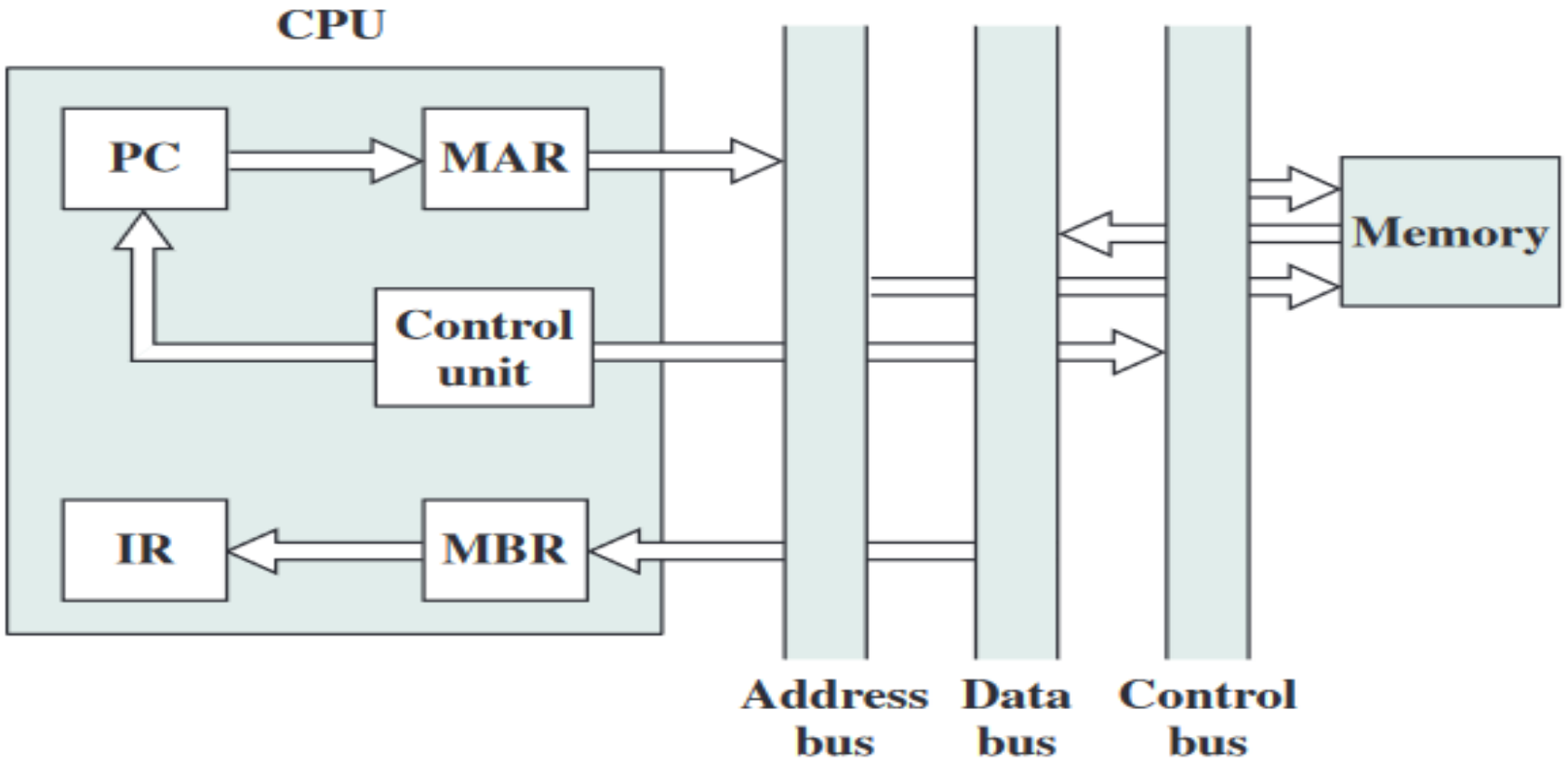


Figure illustrates the nature of the instruction cycle

Data Flow: Fetch, Indirect, and Interrupt Cycle

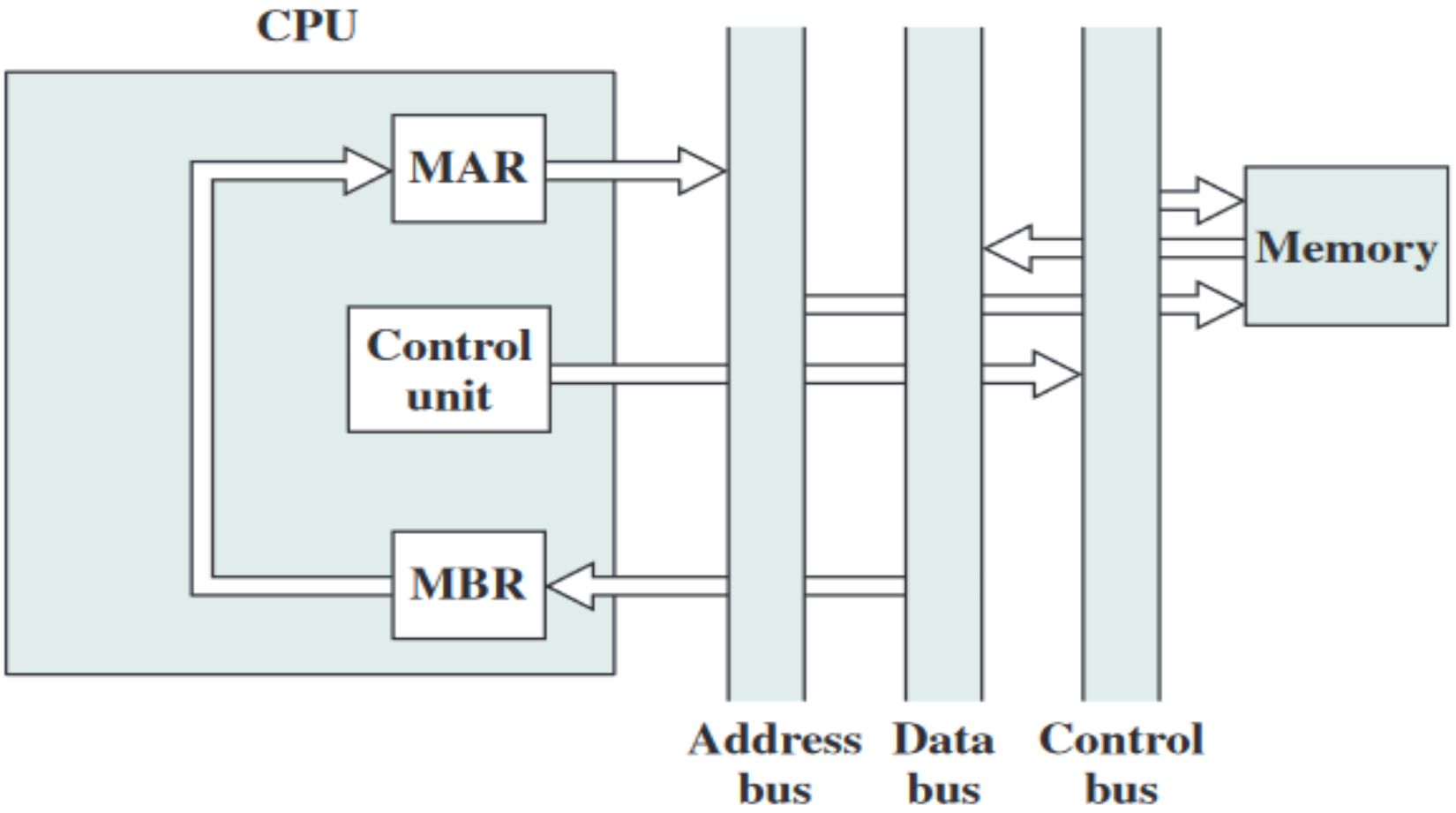
- During the ***fetch cycle***, an instruction is read from memory. PC contains the address of the next instruction to be fetched, this address is moved to the MAR and placed on the address bus. The control unit requests a memory read, and the result is placed on the data bus and copied into the MBR and then moved to the IR. The PC is incremented by 1, preparatory for the next fetch.
- After fetch cycle, 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. 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.
- The *execute cycle* takes many forms; the form depends on which of the various machine instructions is in the IR. This cycle may involve transferring data among registers, read or write from memory or I/O, and/or the invocation of the ALU.
- The ***interrupt cycle*** is simple and predictable. The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt. Thus, the contents of the PC are transferred to the MBR to be written into memory. The special memory location reserved for this purpose is loaded into the MAR from the control unit. The PC is loaded with the address of the interrupt routine. The next instruction cycle will begin by fetching the appropriate instruction.

Data Flow: Fetch Cycle

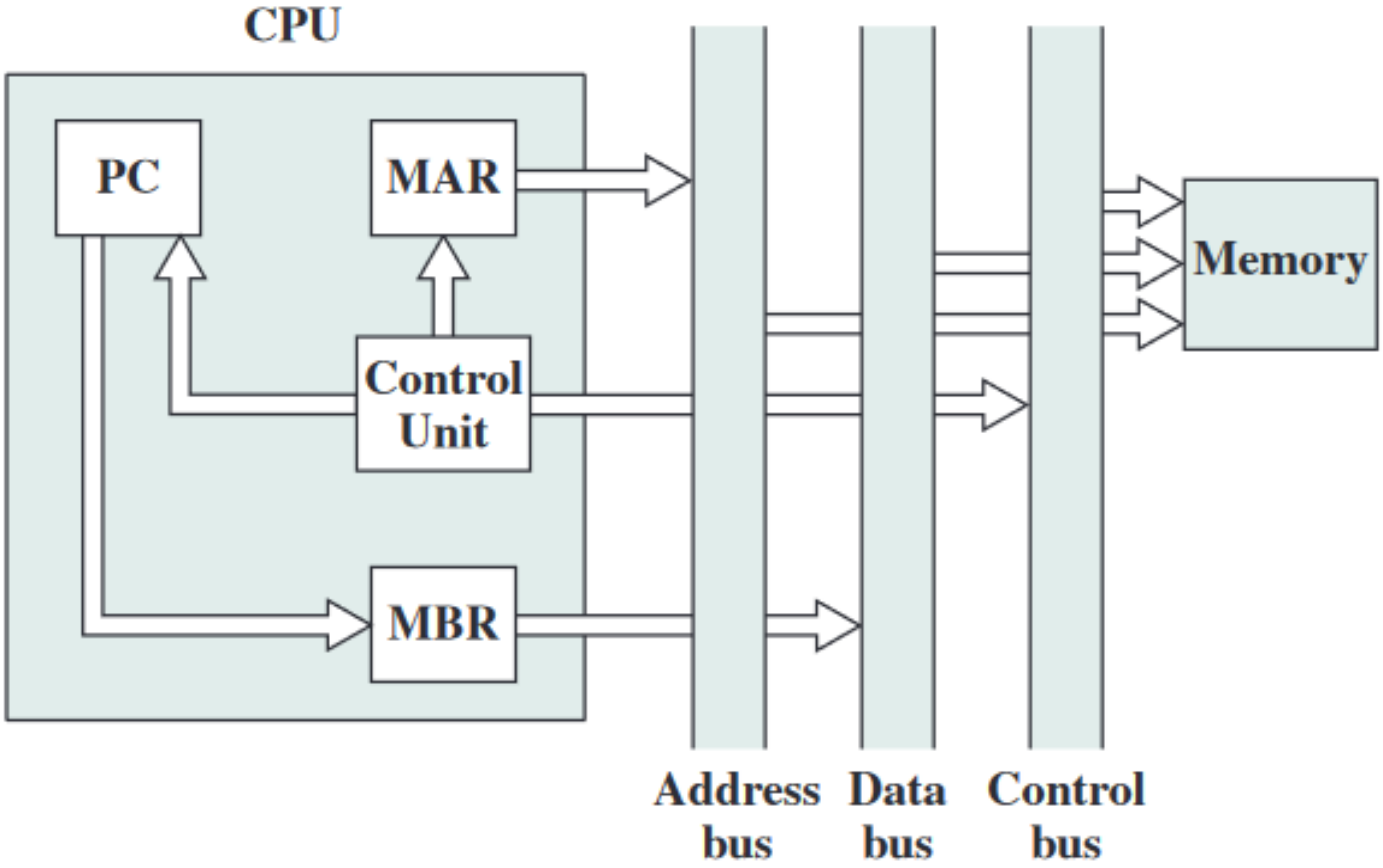


MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Data Flow: Indirect Cycle



Data Flow: Interrupt Cycle



Processor Structure and Function

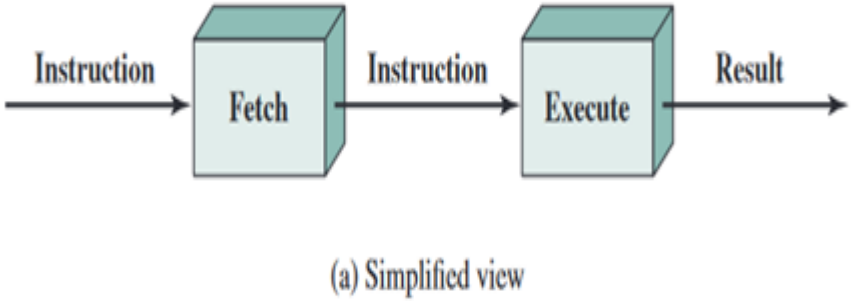
- Processor Organization
- Register Organization
- Instruction Cycle
- **Instruction Pipelining**

Pipelining Strategy

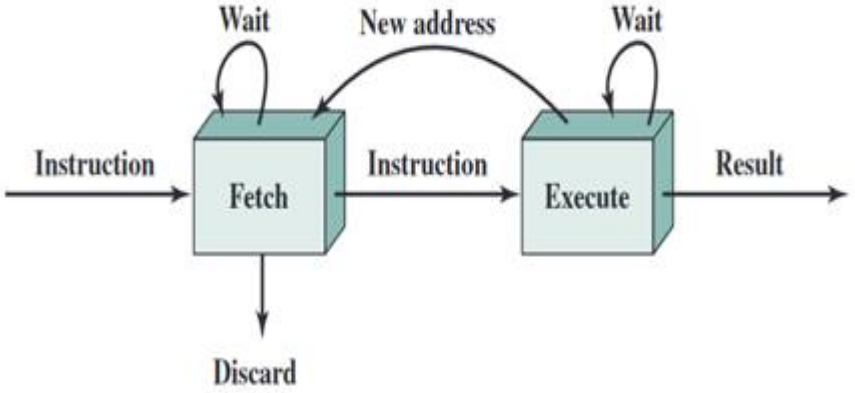
- Instruction pipelining is similar to the use of an assembly line in a manufacturing plant.
- An assembly line takes advantage of the fact that a product goes through various stages of production.
- By laying the production process out in an assembly line, products at various stages can be worked on simultaneously.
- This process is also referred to as *pipelining*, because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.
- To apply this concept to instruction execution, we must recognize that, an instruction has a number of stages.
- Consider subdividing instruction processing into two stages:
  - fetch instruction
  - execute instruction

Instruction Pipelining

- The pipeline has two independent stages:
  - The first stage fetches an instruction and buffers it.
  - When the second stage is free, first stage passes it the buffered instruction.
  - While the second stage is executing the instruction, the first stage takes unused memory cycles to fetch and buffer the next instruction, called instruction prefetch or *fetch overlap*.
- Pipelining requires registers to store data between stages.



(a) Simplified view



(b) Expanded view

Six-stage Pipeline

- **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, in the specified destination operand location.
- **Write operand (WO):** Store the result in memory.

Timing Diagram for Instruction Pipeline Operation

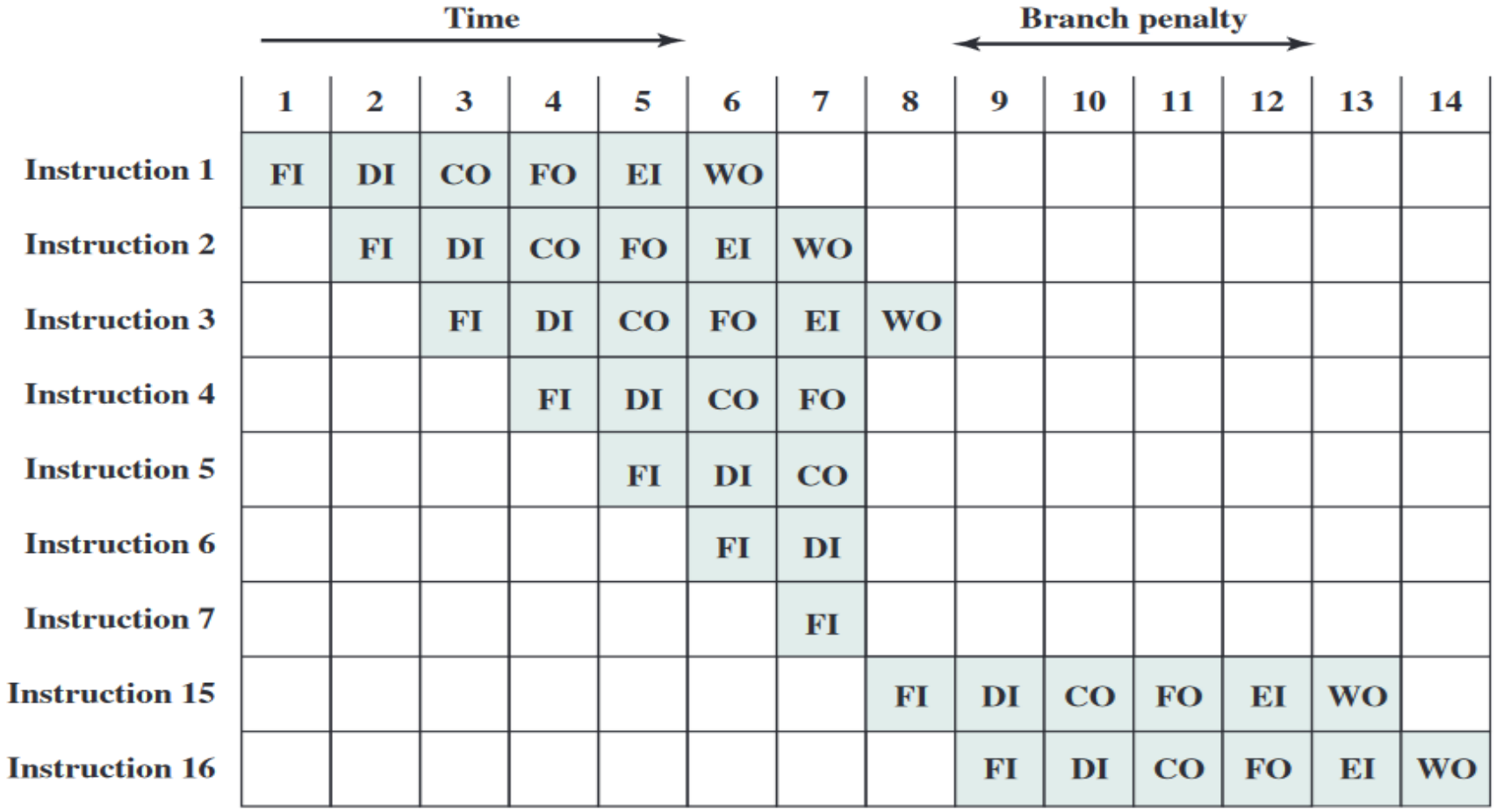
- Figure shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

Time →

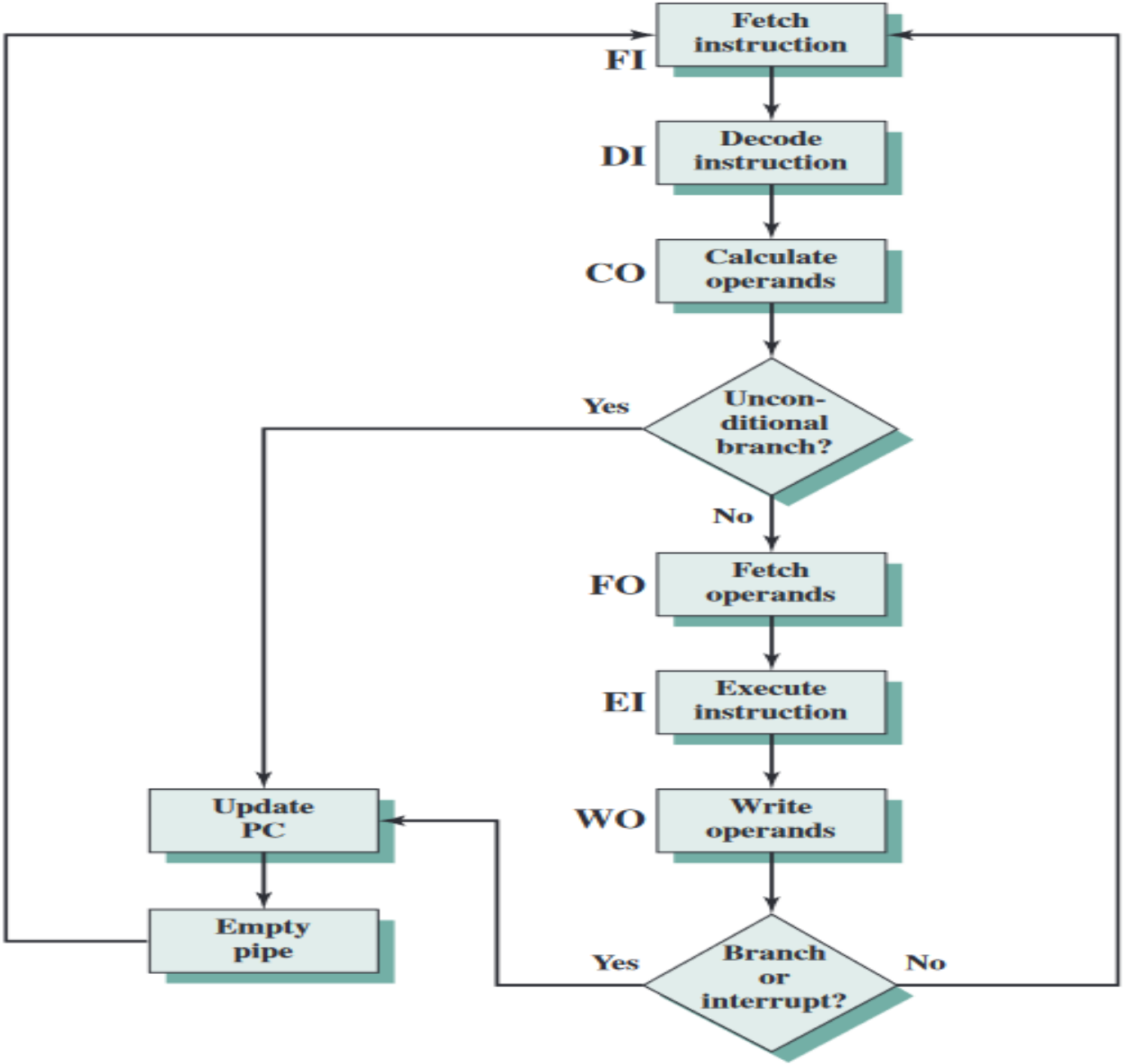
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO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

Effect of a Conditional Branch Instruction Pipeline Operation

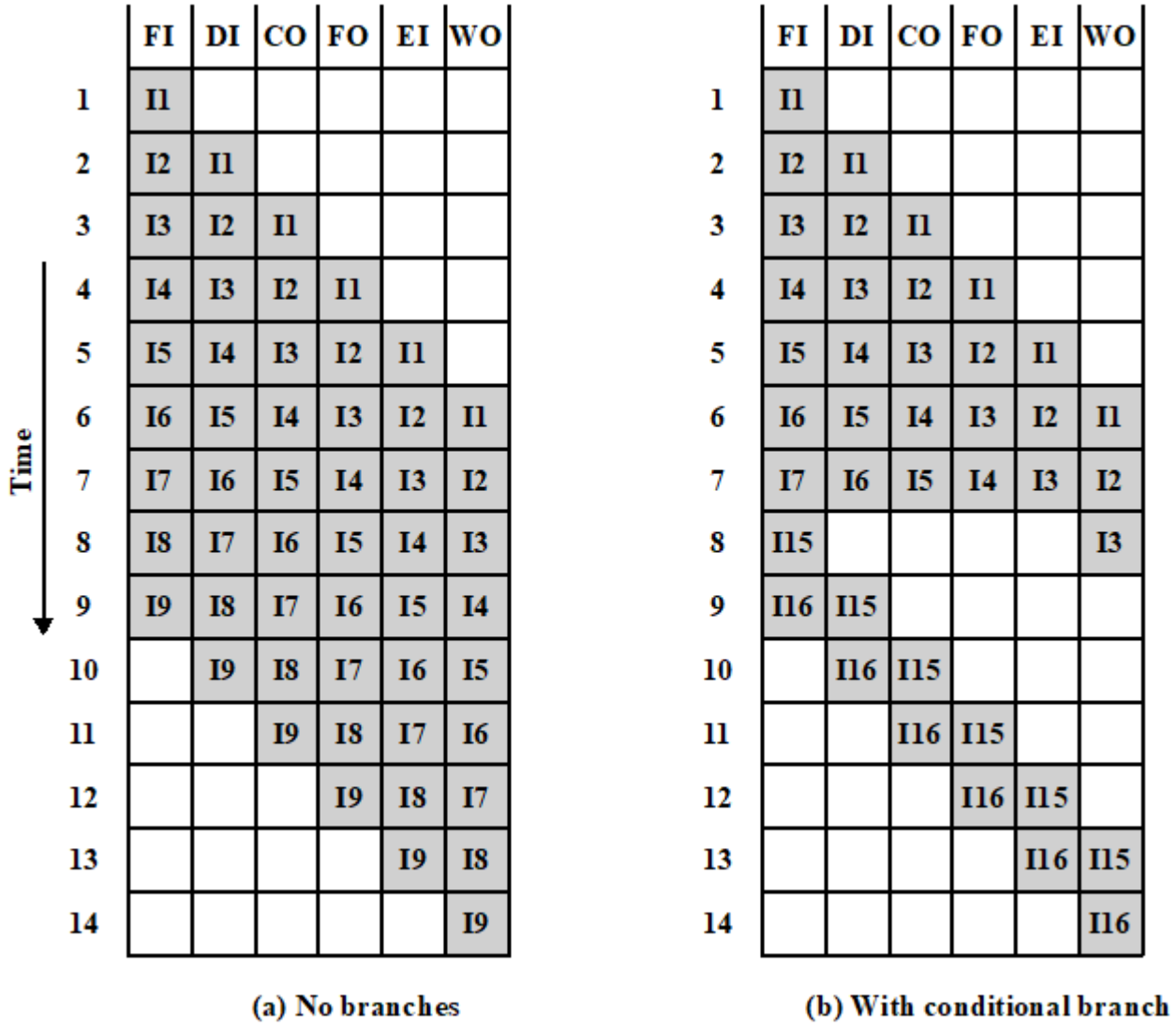
- Figure illustrates the effects of the conditional branch on instruction pipeline operation



Six-stage CPU Instruction Pipeline



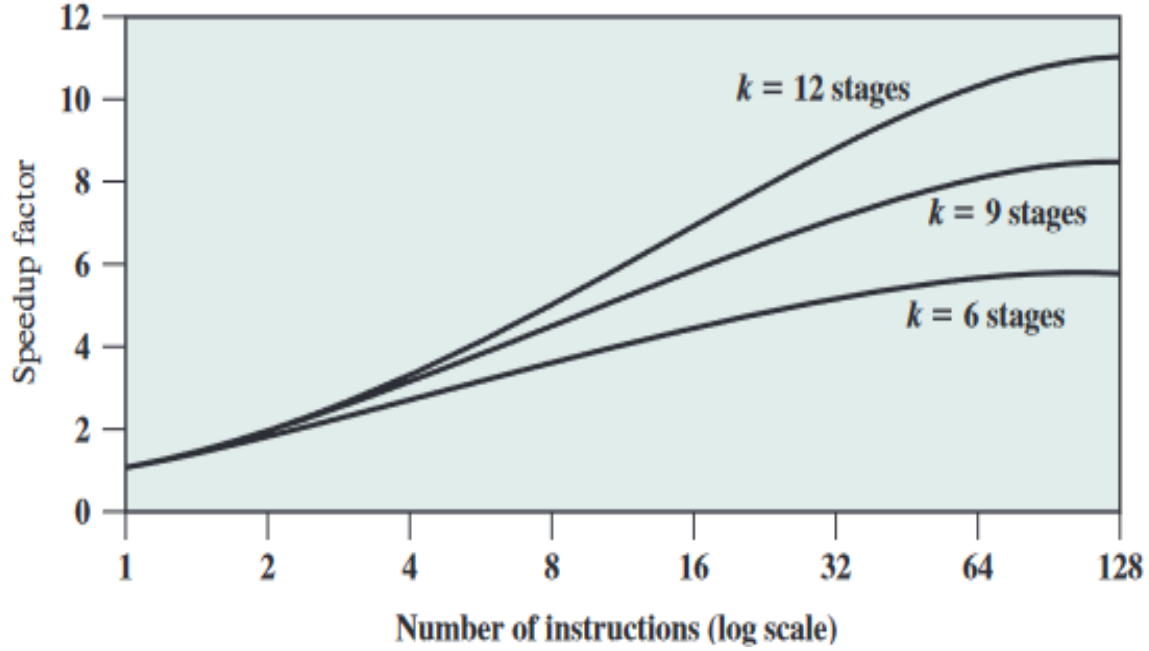
- Figure shows sequence of events, with time progressing vertically down the figure.
- Each row showing the state of the pipeline at a given point in time.

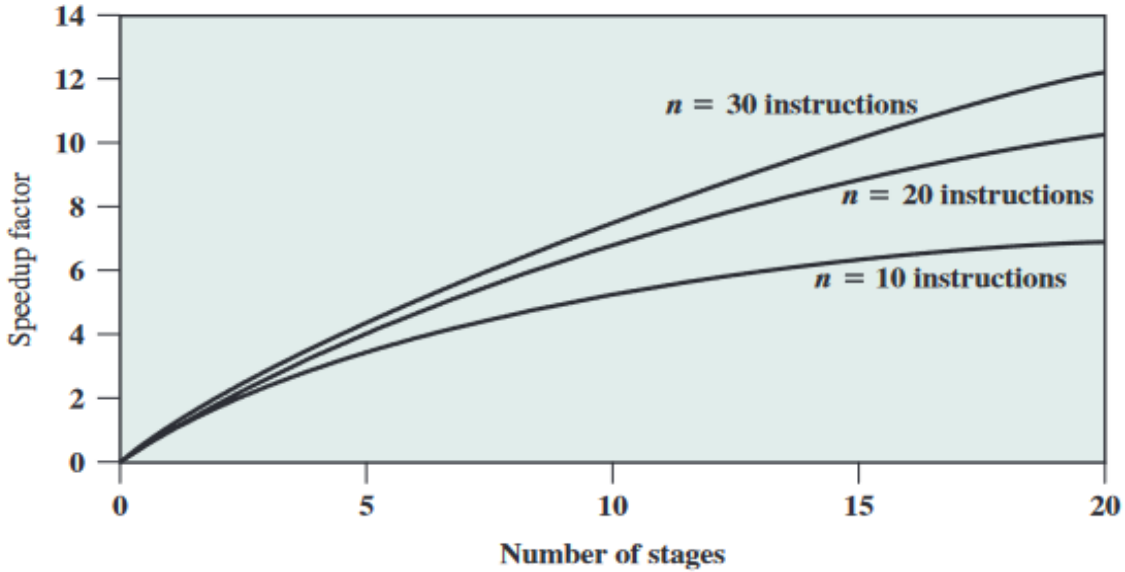


(a) No branches

(b) With conditional branch

Speedup Factors with Instruction Pipelining



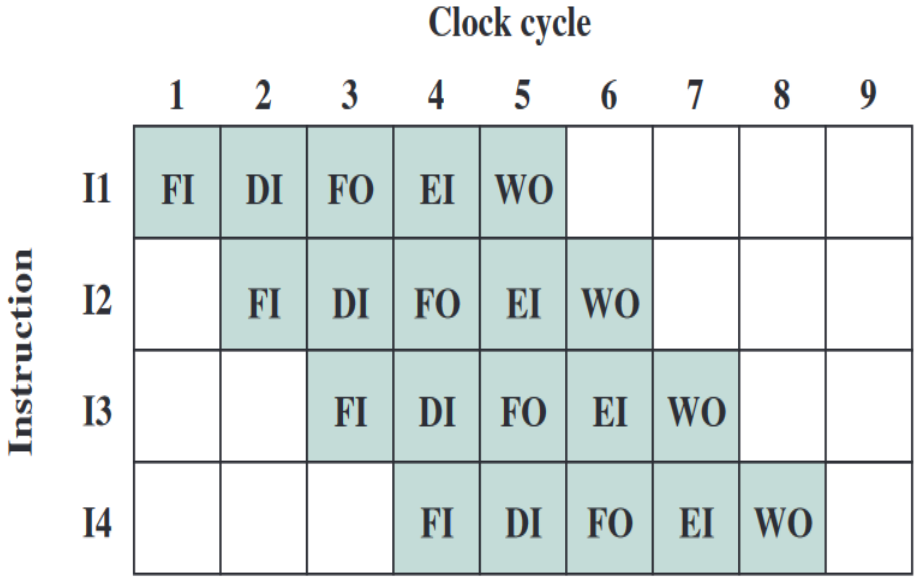


Pipeline Hazards

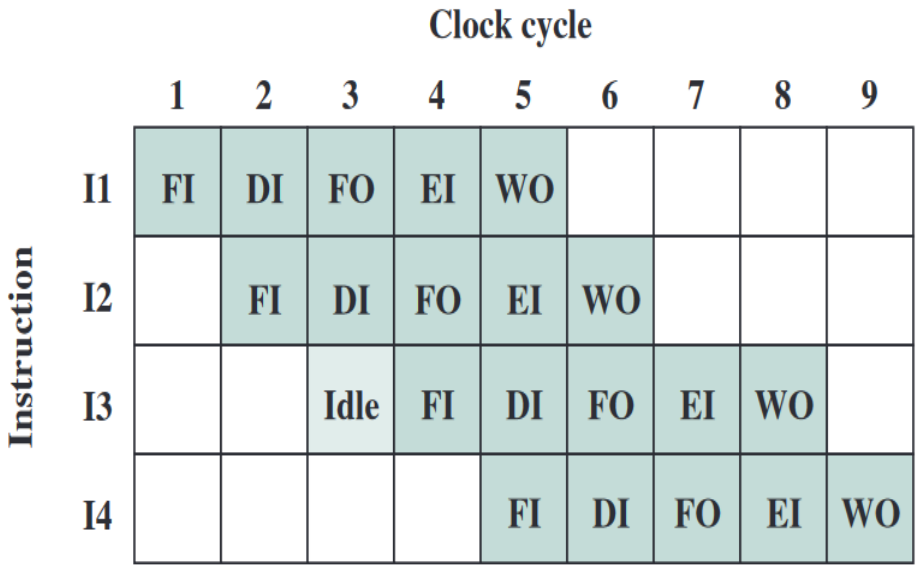
- A pipeline hazard occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution.
- Such a pipeline stall is also referred to as a *pipeline bubble*.
- There are three types of hazards:
  - Resource
  - Data
  - Control

Resource 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 sometime referred to as a *structural hazard*.



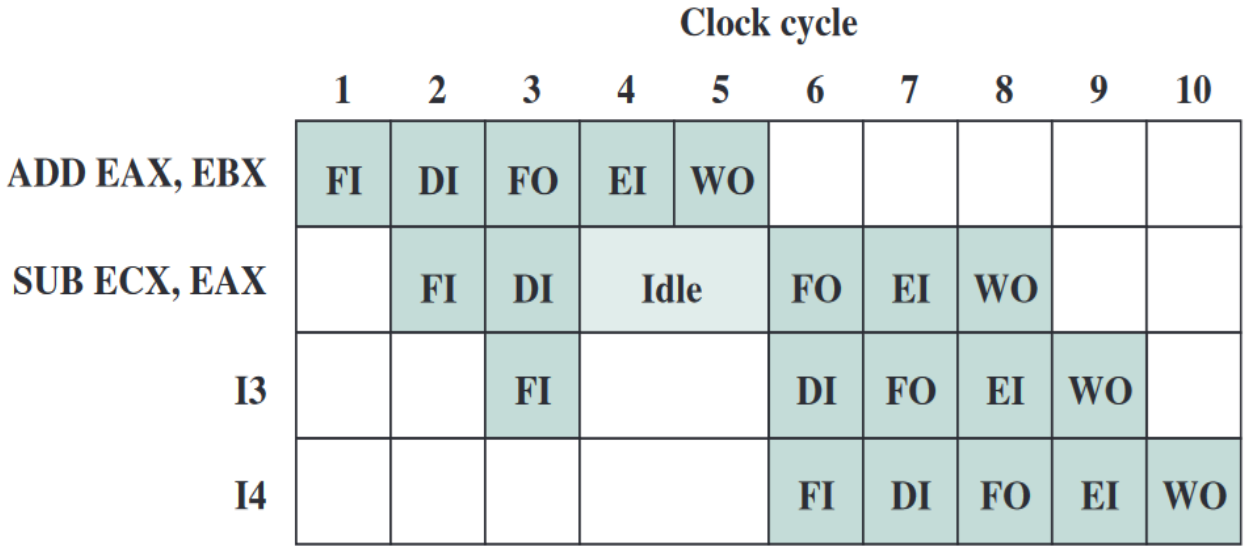
(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

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 instructions are executed in a pipeline, then it is possible for the operand value to be updated to produce a different result than would occur with strict sequential execution.




```
ADD EAX, EBX /* EAX = EAX + EBX
SUB ECX, EAX /* ECX = ECX - EAX
```

Data Hazards: Types

- **Read after write (RAW), or true dependency:**
  - An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location.
  - A hazard occurs if the read takes place before the write operation is complete.
- **Write after read (WAR), or antidependency:**
  - An instruction reads a register or memory location and a succeeding instruction writes to the location.
  - A 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.
  - A hazard occurs if the write operations take place in the reverse order of the intended sequence.

Control Hazards

- A 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

- The problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline.
- Approaches have been taken for dealing with conditional 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 (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.
- Despite these drawbacks, this strategy can improve performance.

Prefetch Branch Target

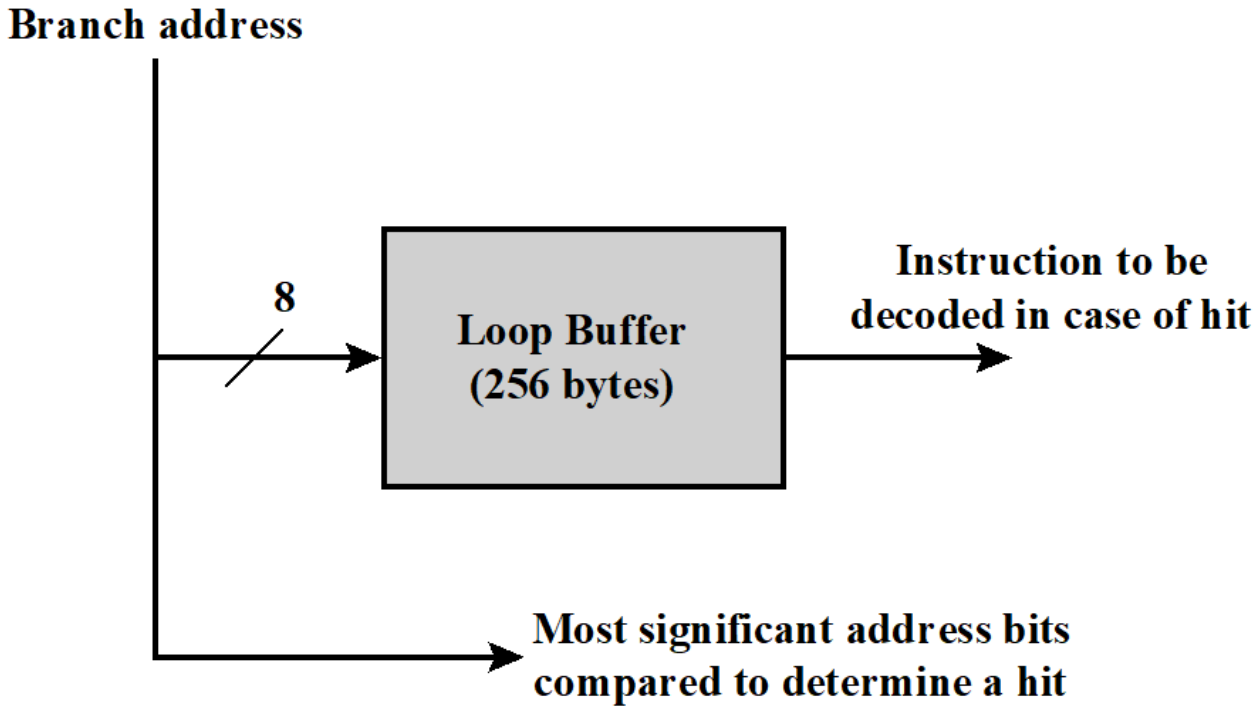
- When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch
- Target is then saved until the branch instruction is executed
- If the branch is taken, the target has already been prefetched
- IBM 360/91 uses this approach

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

Loop Buffer

- Figure gives an example of a loop buffer.
- If the buffer contains 256 bytes, and byte addressing is used, then the least significant 8 bits are used to index the buffer.
- The remaining most significant bits are checked to determine if the branch target lies within the environment captured by the buffer.



Branch Prediction

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

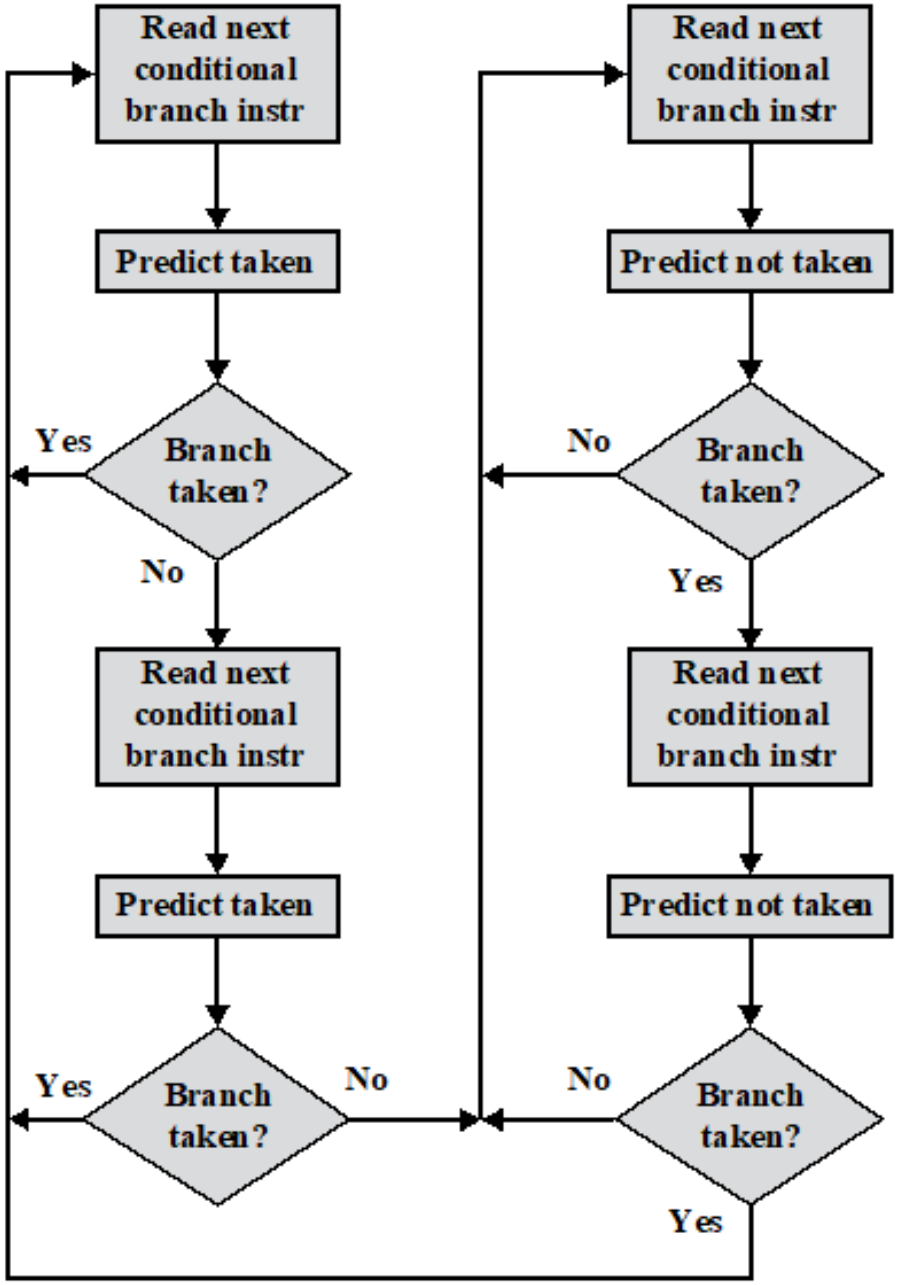
1. Predict never taken
2. Predict always taken
3. Predict by opcode

- 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

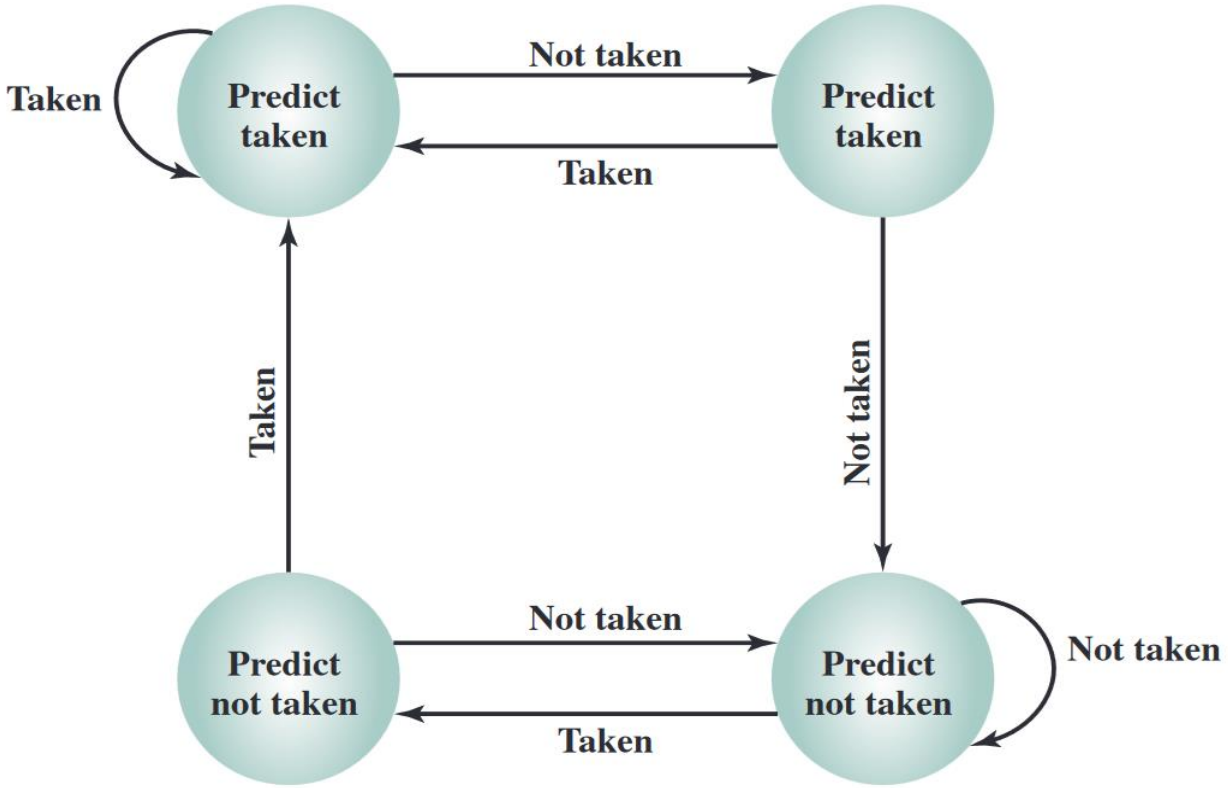
- These approaches are dynamic
- They depend on the execution history

Branch Prediction Flowchart



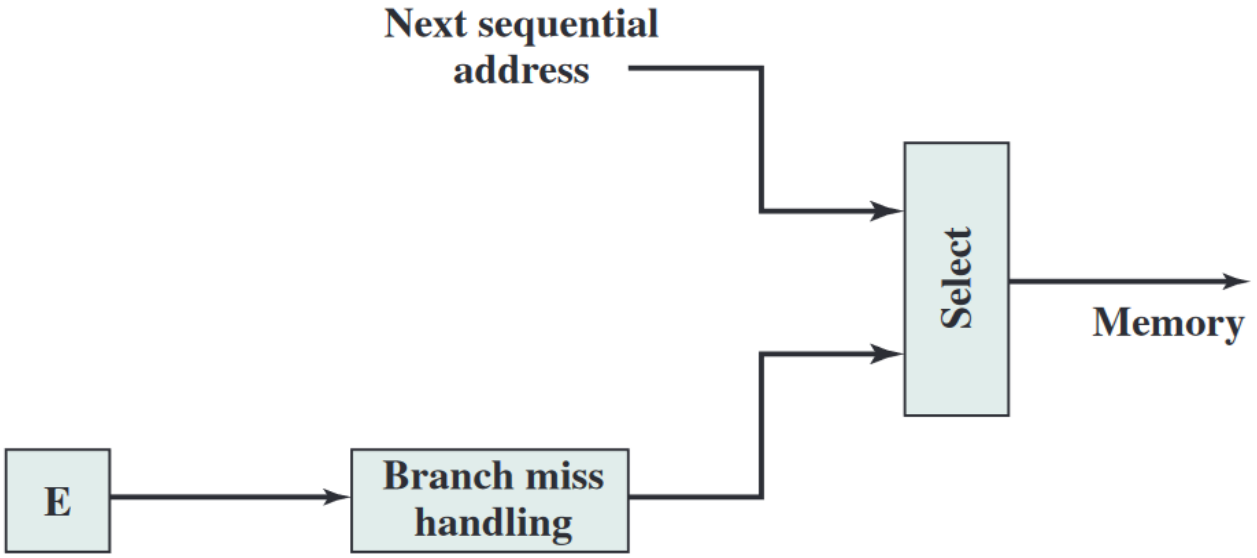
Branch Prediction State Diagram

- The decision process can be represented more compactly by a finite-state machine, as shown in Figure.



Dealing with Branches

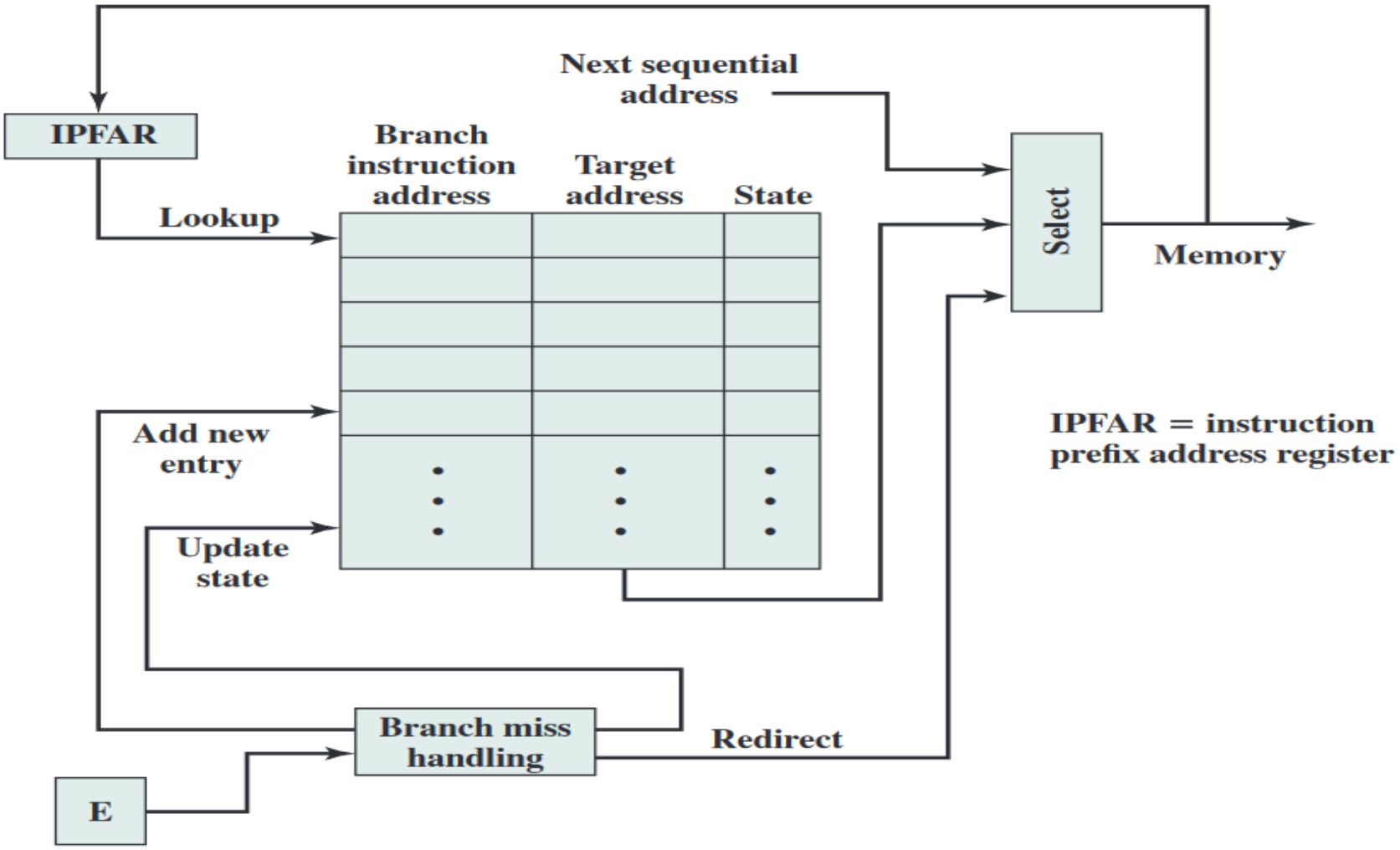
- Figure (a) illustrates a predict-never-taken strategy.



(a) Predict never taken strategy

- The branch history table is a small cache memory associated with the instruction fetch stage of the pipeline, shown in Figure (b) next.
- Each entry in the table consists of three elements:
  - the address of a branch instruction
  - number of history bits that record the state of use of that instruction
  - information about the target instruction

Dealing with Branches

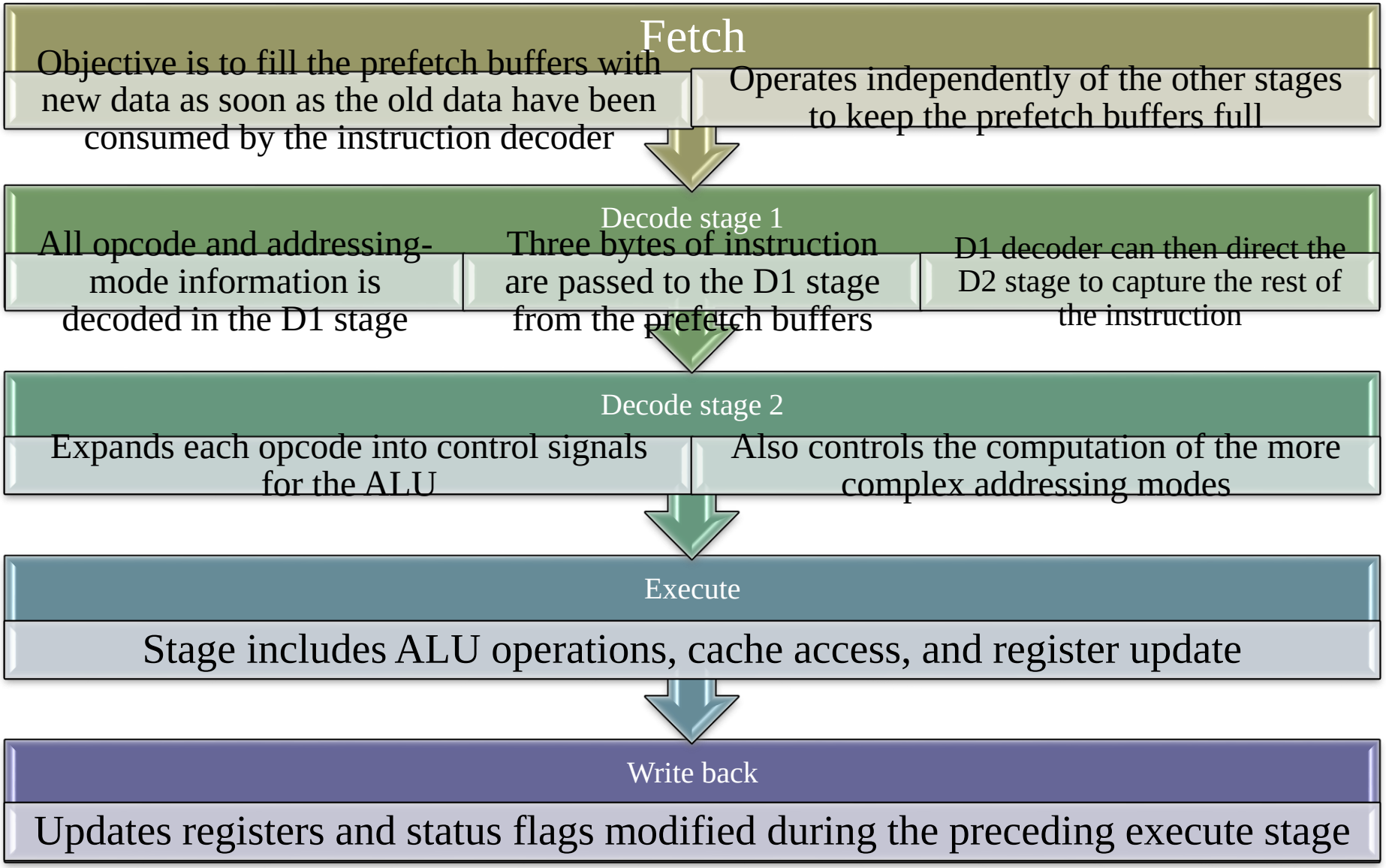


(b) Branch history table strategy

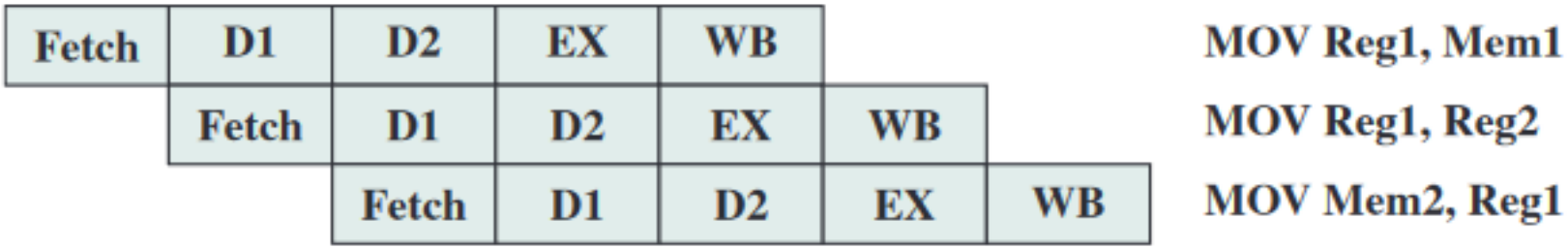
Delayed Branch

- It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.

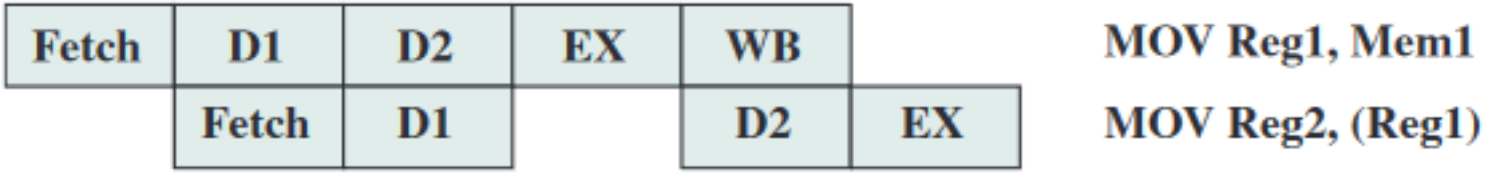
Intel 80486 Pipelining



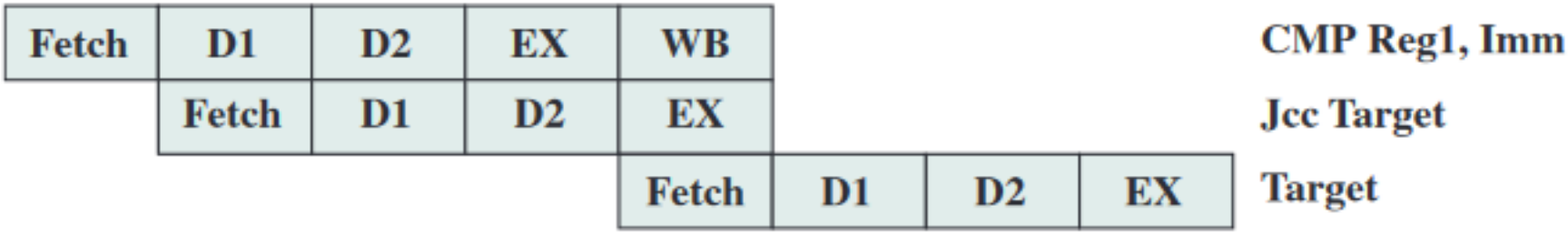
80486 Instruction Pipeline Examples



(a) No data load delay in the pipeline



(b) Pointer load delay



(c) Branch instruction timing

Parallel Processing

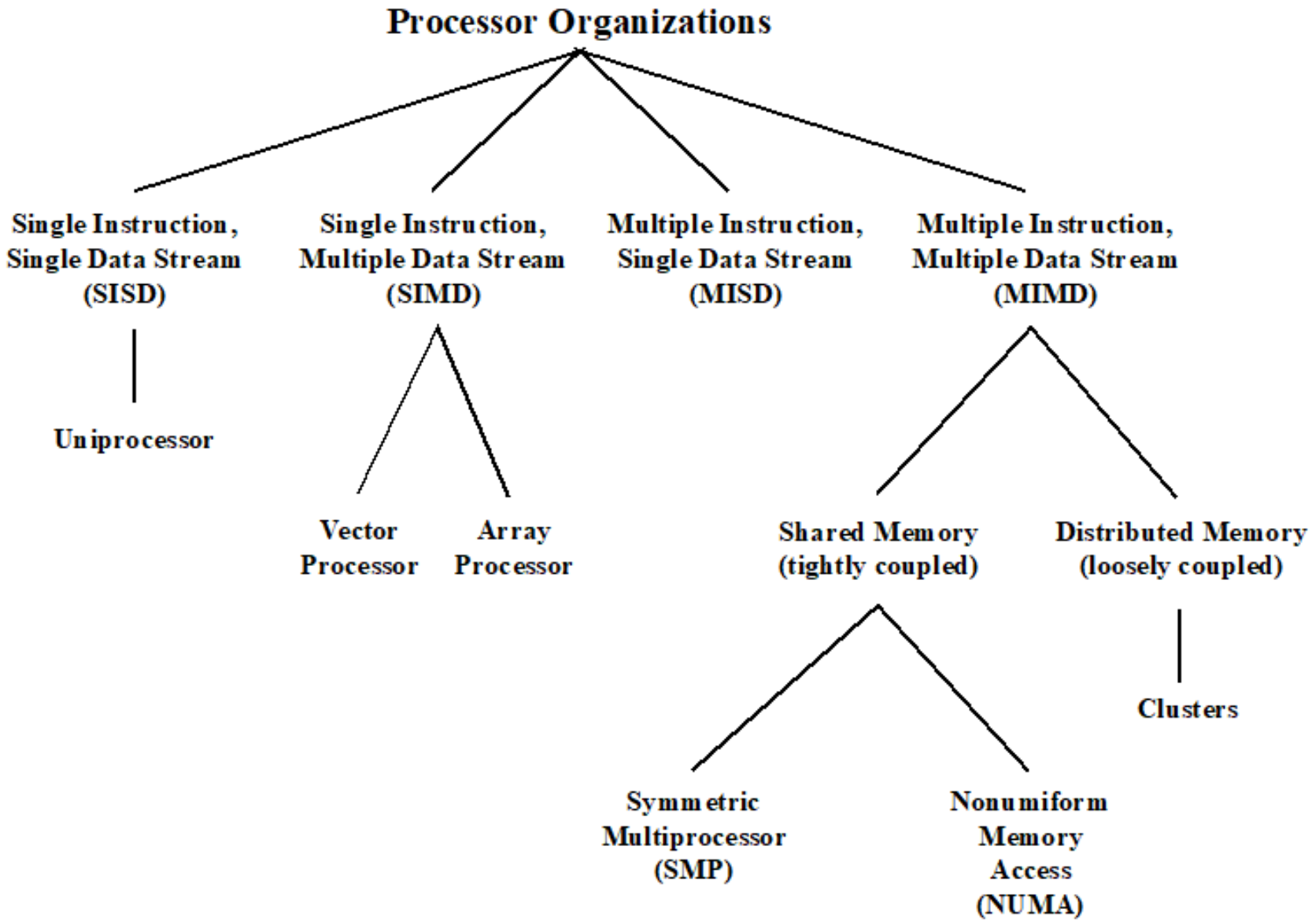
- **Multiple Processor Organizations**
- **Symmetric Multiprocessors**
  - Cache Coherence
  - MESI Protocol

Multiple Processor Organizations

Flynn proposed the following categories of computer systems:

- Single instruction, single data **(SISD)** stream
  - Single processor executes a single instruction stream to operate on data stored in a single memory
  - Uniprocessors fall into this category
- Single instruction, multiple data **(SIMD)** stream
  - A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis
  - Vector and array processors fall into this category
- Multiple instruction, single data **(MISD)** stream
  - A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence
  - Not commercially implemented
- Multiple instruction, multiple data **(MIMD)** stream
  - A set of processors simultaneously execute different instruction sequences on different data sets
  - SMPs, clusters and NUMA systems fit this category

A Taxonomy of Parallel Processor Architectures

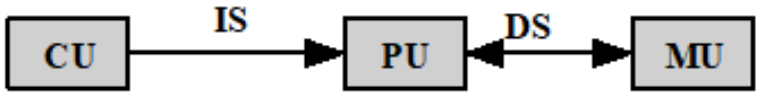


General Organization of the Taxonomy

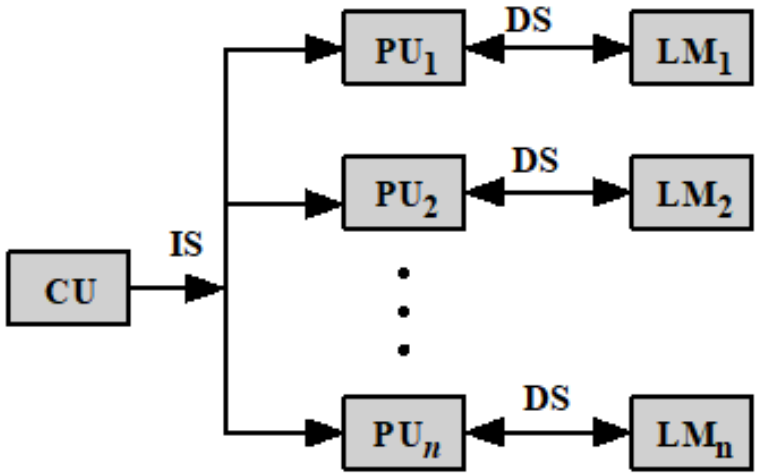
- Figure a shows the structure of an SISD.
- There is some sort of control unit (CU) that provides an instruction stream (IS) to a processing unit (PU).
- The processing unit operates on a single data stream (DS) from a memory unit (MU).
- With an SIMD, there is still a single control unit, now feeding a single instruction stream to multiple PUs.
- Each PU may have its own dedicated memory (illustrated in Figure b), or there may be a shared memory.
- With the MIMD, there are multiple control units, each feeding a separate instruction stream to its own PU.
- The MIMD may be a shared-memory multiprocessor (Figure c) or a distributed-memory multicomputer (Figure d).

General Organization of the Taxonomy:

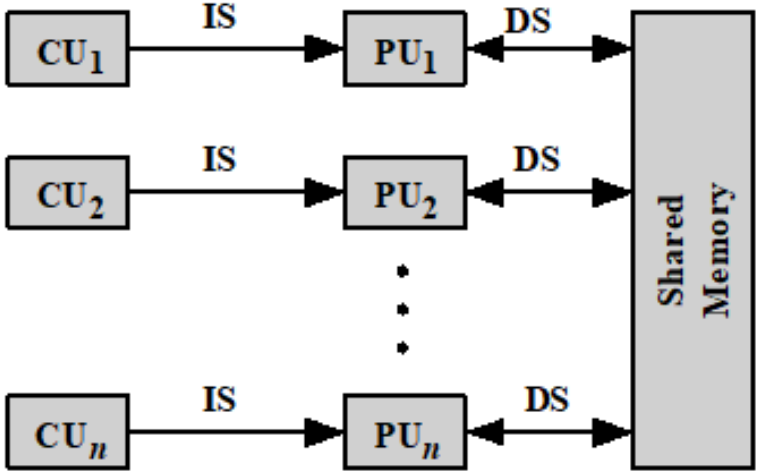
Alternative Computer Organizations



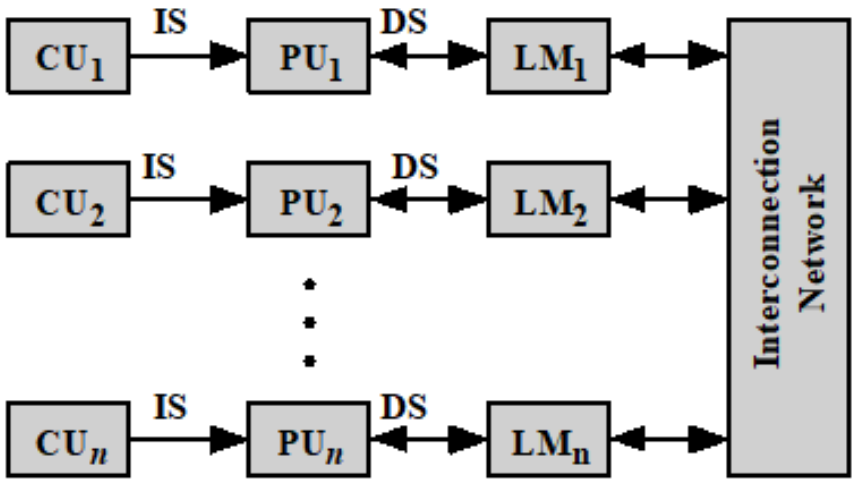
(a) SISD



(b) SIMD (with distributed memory)



(c) MIMD (with shared memory)



(d) MIMD (with distributed memory)

CU = control unit
IS = instruction stream
PU = processing unit
DS = data stream
MU = memory unit
LM = local memory

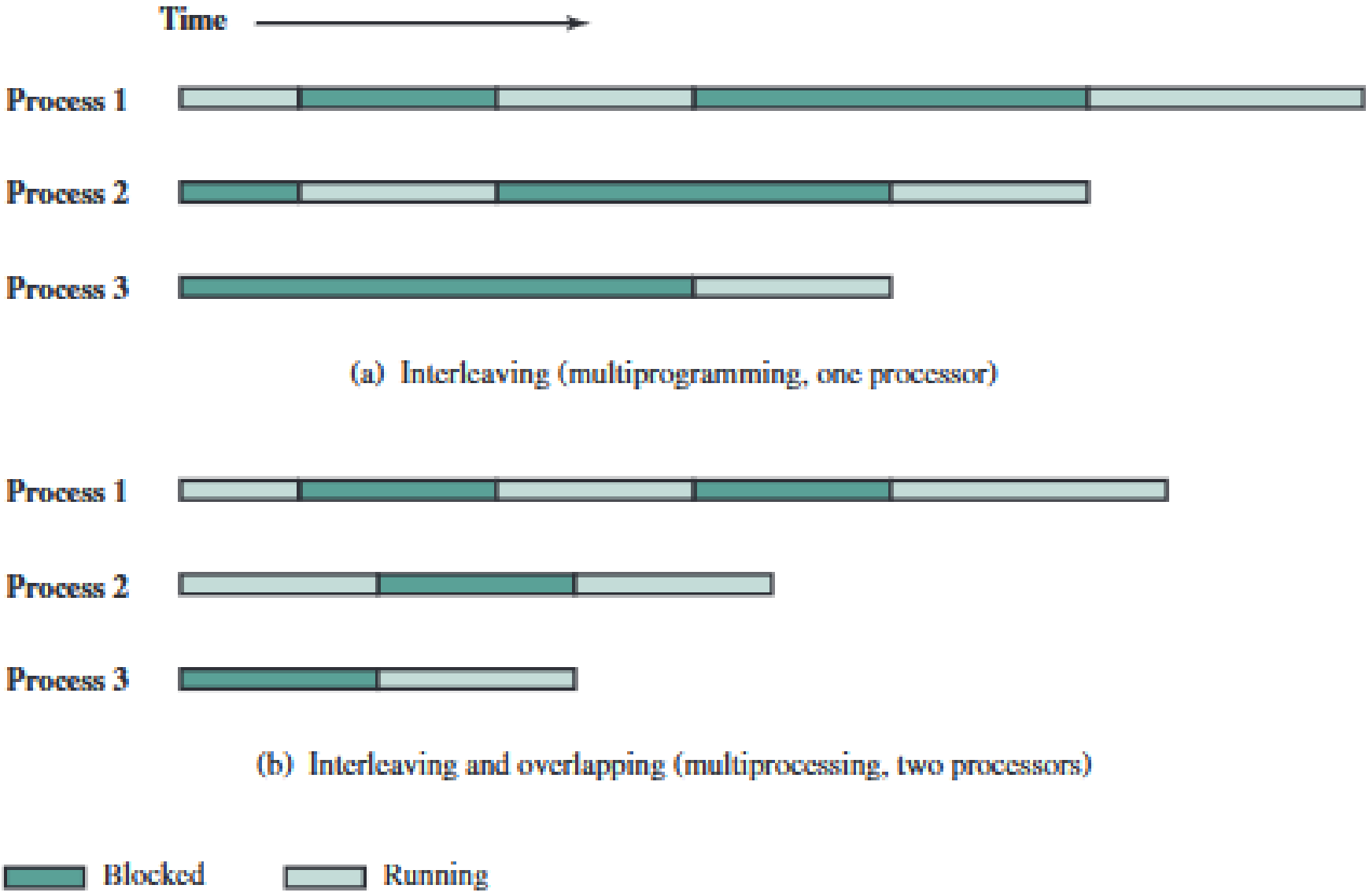
SISD = single instruction, single data stream
SIMD = single instruction, multiple data stream
MIMD = multiple instruction, multiple data stream

Symmetric Multiprocessors (SMP)

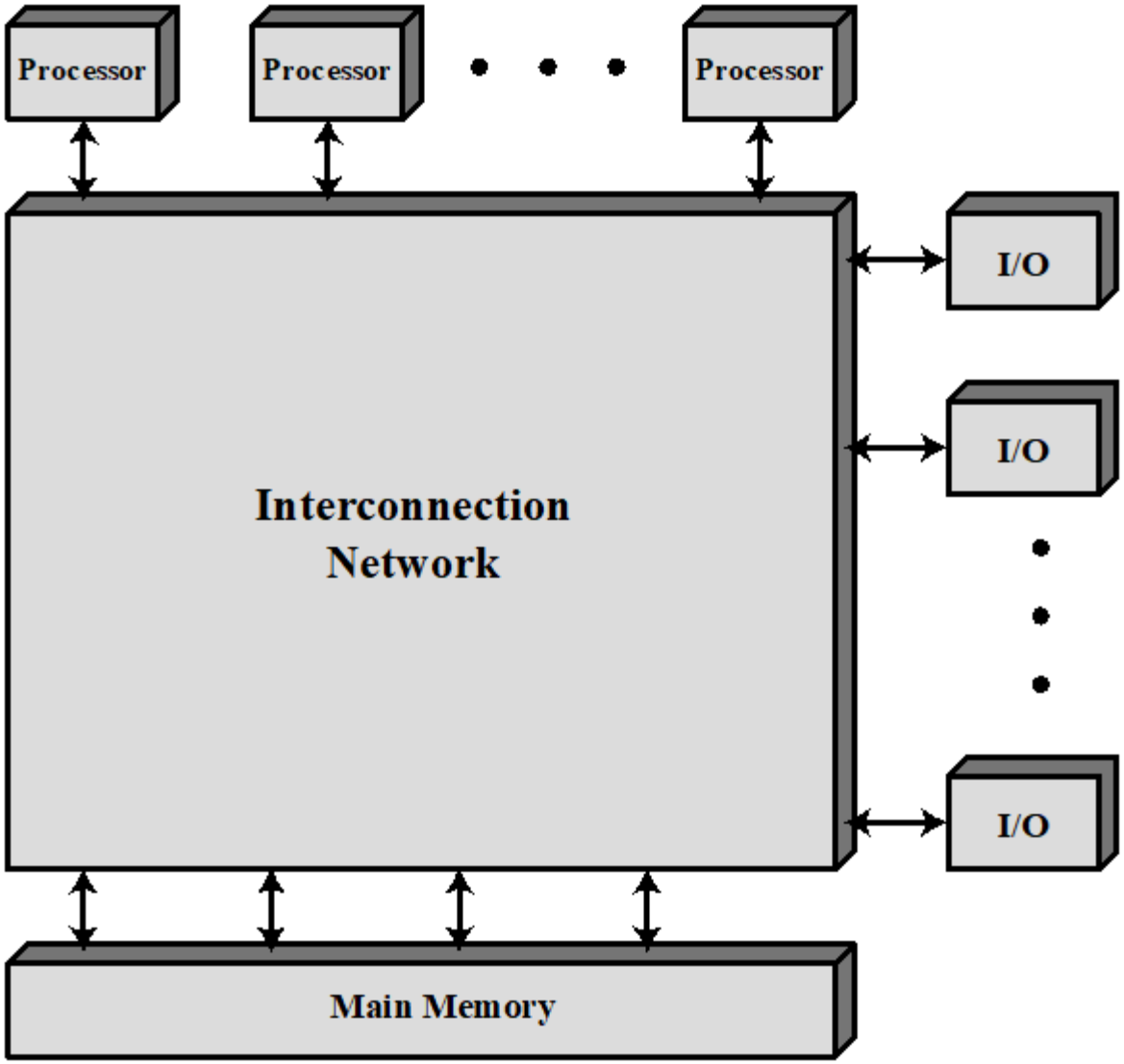
A stand alone computer with the following characteristics:

Two or more similar processors of comparable capacity	Processors share same memory and I/O facilities • Processors are connected by a bus or other internal connection • Memory access time is approximately the same for each processor	All processors share access to I/O devices • Either through same channels or different channels giving paths to same devices	All processors can perform the same functions (hence "symmetric")	System controlled by integrated operating system • Provides interaction between processors and their programs at job, task, file and data element levels

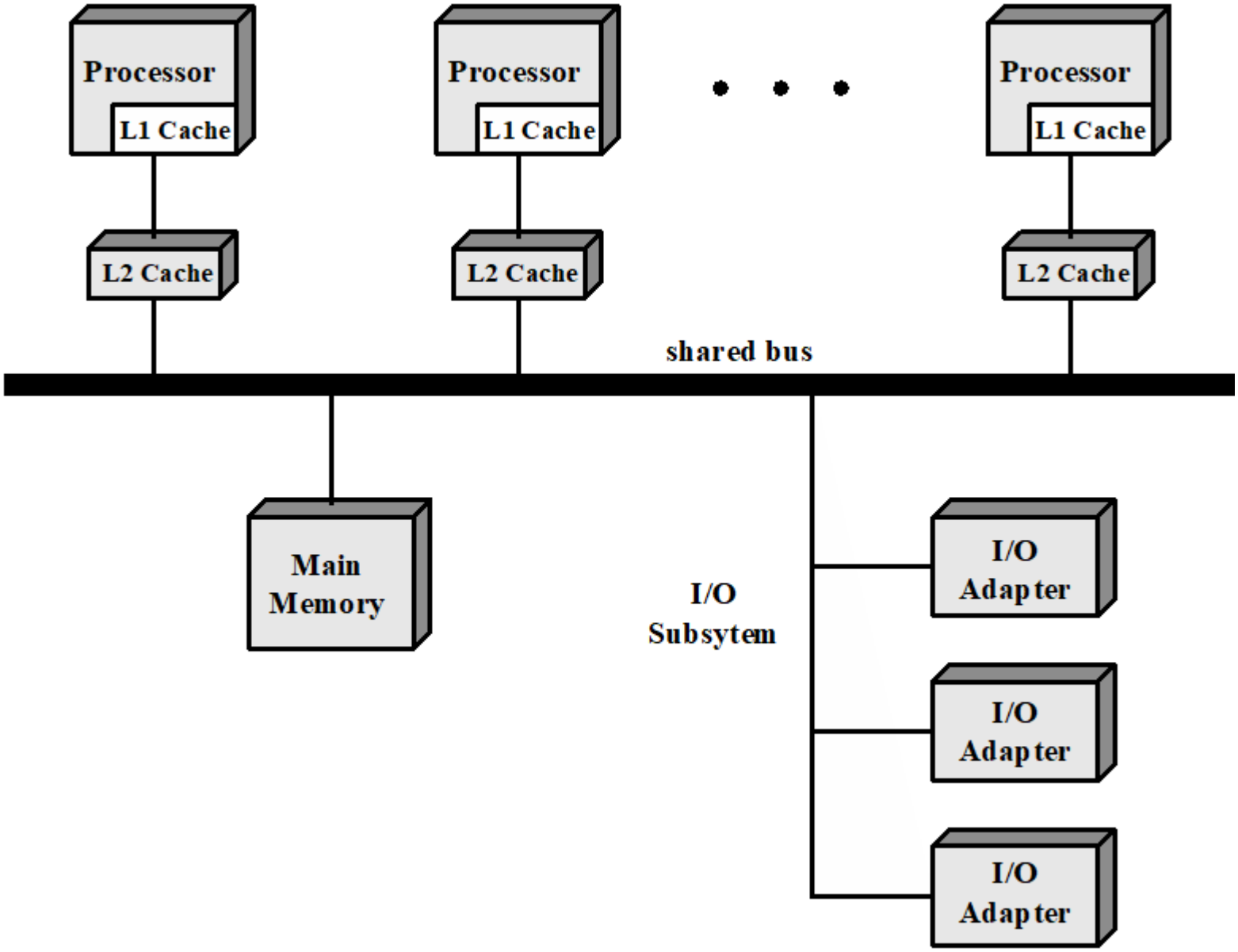
Multiprogramming and Multiprocessing



Generic Block Diagram of a Tightly Coupled Multiprocessor



Symmetric Multiprocessor Organization



Organization

The bus organization has several attractive features:

- Simplicity
  - Simplest approach to multiprocessor organization
- Flexibility
  - Generally easy to expand the system by attaching more processors to the bus
- Reliability
  - The bus is essentially a passive medium and the failure of any attached device should not cause failure of the whole system

Organization

Disadvantages of the bus organization:

- Main drawback is performance
  - All memory references pass through the common bus
  - Performance is limited by bus cycle time
- Each processor should have cache memory
  - Reduces the number of bus accesses
- Leads to problems with *cache coherence*
  - If a word is altered in one cache it could conceivably invalidate a word in another cache
  - To prevent this other processors must be alerted that an update has taken place
  - Typically addressed in hardware rather than the operating system

Multiprocessor Operating System Design Considerations

- **Simultaneous concurrent processes**
  - OS routines need to be reentrant to allow several processors to execute the same IS code simultaneously
  - OS tables and management structures must be managed properly to avoid deadlock or invalid operations
- **Scheduling**
  - Any processor may perform scheduling so conflicts must be avoided
  - Scheduler must assign ready processes to available processors
- **Synchronization**
  - With multiple active processes having potential access to shared address spaces or I/O resources, care must be taken to provide effective synchronization
  - Synchronization is a facility that enforces mutual exclusion and event ordering
- **Memory management**
  - In addition to dealing with all of the issues found on uniprocessor machines, the OS needs to exploit the available hardware parallelism to achieve the best performance
  - Paging mechanisms on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement
- **Reliability and fault tolerance**
  - OS should provide graceful degradation in the face of processor failure
  - Scheduler and other portions of the operating system must recognize the loss of a processor and restructure accordingly

Parallel Processing

- Multiple Processor Organizations
- Symmetric Multiprocessors
- **Cache Coherence**
- MESI Protocol

Cache Coherence

Software Solutions

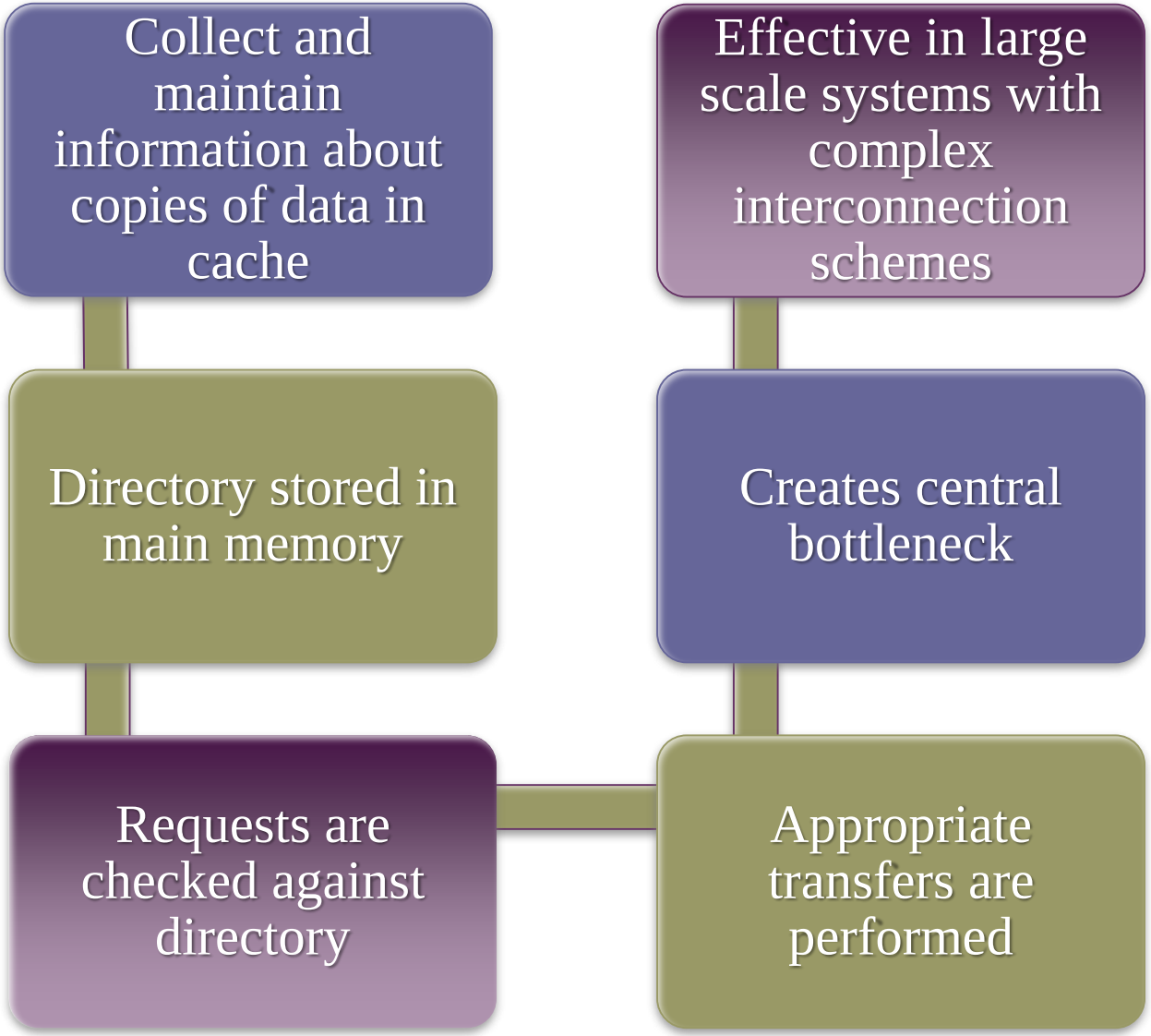
- Attempt to avoid the need for additional hardware circuitry and logic by relying on the compiler and operating system to deal with the problem.
- Attractive because the overhead of detecting potential problems is transferred from run time to compile time, and the design complexity is transferred from hardware to software.
  - Compile-time software approaches generally must make conservative decisions, leading to inefficient cache utilization

Cache Coherence

Hardware-Based Solutions

- Generally referred to as cache coherence protocols
- These solutions provide dynamic recognition at run time of potential inconsistency conditions
- Because the problem is only dealt with when it actually arises there is more effective use of caches, leading to improved performance over a software approach
- Approaches are transparent to the programmer and the compiler, reducing the software development burden
- Can be divided into two categories:
  1. Directory protocols
  2. Snoopy protocols

1. Directory Protocols



1. Snoopy Protocols

- Distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor:
  - A cache must recognize when a line that it holds is shared with other caches
  - When updates are performed on a shared cache line, it must be announced to other caches by a broadcast mechanism
  - Each cache controller is able to “snoop” on the network to observe these broadcast notifications and react accordingly
- Suited to bus-based multiprocessor because the shared bus provides a simple means for broadcasting and snooping
  - Care must be taken that the increased bus traffic required for broadcasting and snooping does not cancel out the gains from the use of local caches
- Two basic approaches have been explored:
  - Write invalidate
  - Write update (or write broadcast)

Write Invalidate

- Multiple readers, but only one writer at a time
- When a write is required, all other caches of the line are invalidated
- Writing processor then has exclusive (cheap) access until line is required by another processor
- Widely used in commercial multiprocessor systems such as the x86 architecture
- State of every line is marked as modified, exclusive, shared or invalid, for this reason the write-invalidate protocol is called *MESI*.

Write Update

- With a write-update protocol, there can be multiple writers as well as multiple readers.
- When a processor wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it.
- Neither of these two approaches is superior to the other under all circumstances.
- Performance depends on the number of local caches and the pattern of memory reads and writes.
- Some systems implement adaptive protocols that employ both write-invalidate and write-update mechanisms.

Parallel Processing

- Multiple Processor Organizations
- Symmetric Multiprocessors
- Cache Coherence
- **MESI Protocol**

MESI Protocol

- To provide cache consistency on an SMP the data cache supports a protocol known as MESI.
- For MESI, the data cache includes two status bits per tag, so that each line can be in one of four states:
  - Modified: The line in the cache has been modified and is available only in this cache
  - Exclusive: The line in the cache is the same as that in main memory and is not present in any other cache
  - Shared: The line in the cache is the same as that in main memory and may be present in another cache
  - Invalid: The line in the cache does not contain valid data
- Table next summarizes the meaning of the four states.

MESI Protocol

Table: MESI Cache Line States

	M **Modified**	**E** **Exclusive**	**S** **Shared**	**I** **Invalid**
This cache line valid?	Yes	Yes	Yes	No
The memory copy is…	out of date	valid	valid	—
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line…	does not go to bus	does not go to bus	goes to bus and updates cache	goes directly to bus

- Figure next displays a state diagram for the MESI protocol.
- Each line of the cache has its own state bits and therefore its own realization of the state diagram.
- **Figure a** shows the transitions that occur due to actions initiated by the processor attached to this cache.
- **Figure b** shows the transitions that occur due to events that are snooped on the common bus.

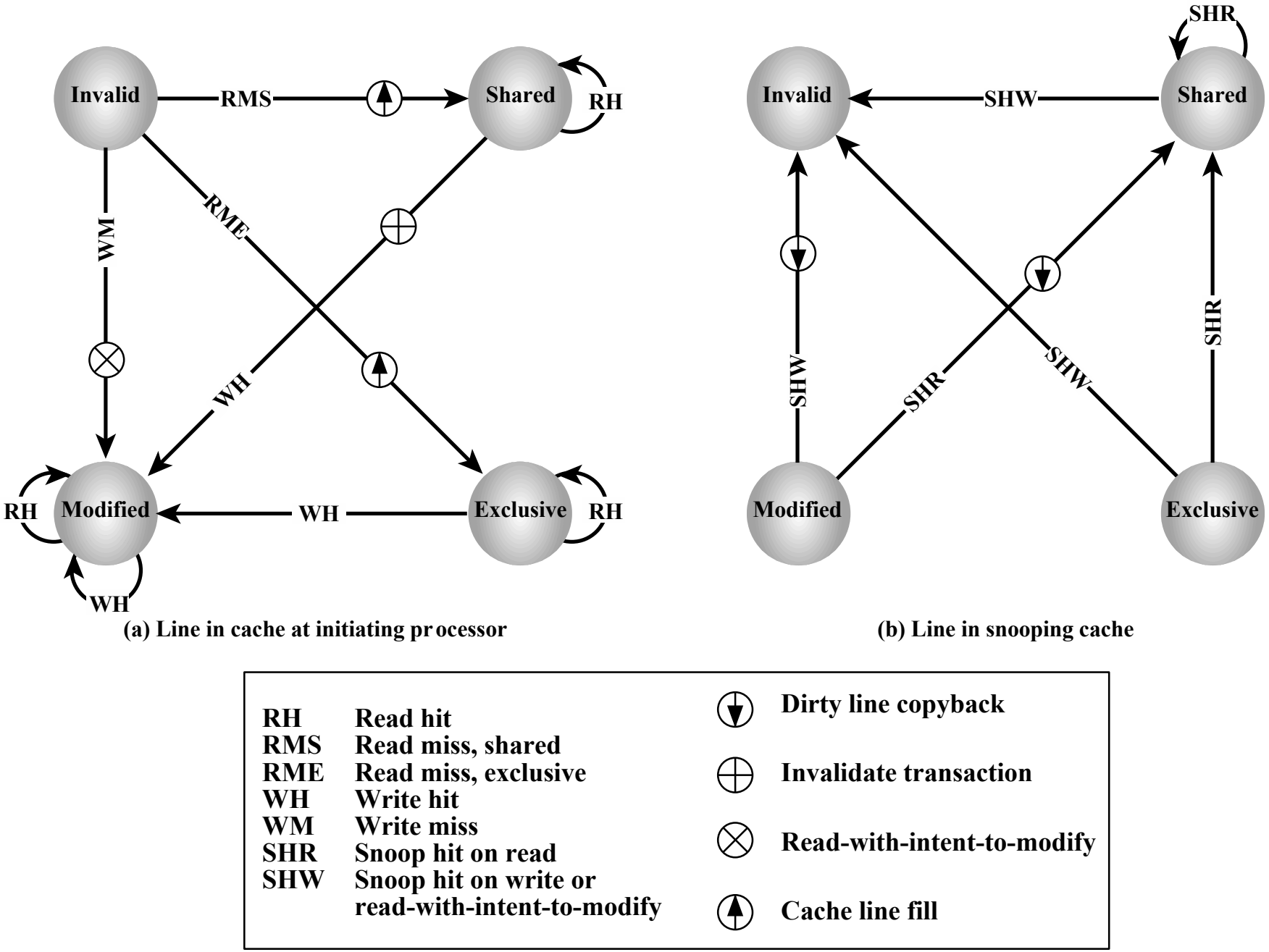


Figure 17.6 MESI State Transition Diagram

MESI Protocol

- Read miss
- Read hit
- Write miss
- Write hit
- L1-L2 Cache consistency

Thank You