Akdeniz University
Computer Engineering Department

CSE206 Computer Organization

Week12: Instruction Sets

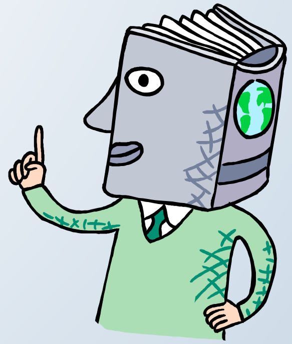
Assoc.Prof.Dr. Taner Danışman
tdanisman@akdeniz.edu.tr

Course program (Textbook: Stalling 10th Edt.)

Week 1	19-Feb-24	Introduction	Ch1
Week 2	26-Feb-24	Computer Evolution	Ch2
Week 3	4-Mar-24	Computer Systems	Ch3
Week 4	11-Mar-24	Cache Memory, Direct Cache Mapping	Ch4
Week 5	18-Mar-24	Associative and Set Associative Mapping	Ch4
Week 6	25-Mar-24	Internal Memory, External Memory, I/O	Ch5-Ch6-Ch7
Week 7	1-Apr-24	Number Systems, Computer Arithmetic	Ch9-Ch10
Week 8	8-Apr-24	Holiday	
Week 9	15-Apr-24	Number Systems, Computer Arithmetic	Ch11
Week 10	26-Apr-24	Midterm	Ch1-Ch11
Week 11	29-Apr-24	Digital Logic	Ch11
Week 12	6-May-24	Instruction Sets	Ch12
Week 13	13-May-24	Addressing Modes	Ch13
Week 14	20-May-24	Processor Structure and Function	Ch14
Week 15	27-May-24	Assembly Language (TextBook : Assembly Language for x86 Processors)	Kip Irvine

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



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

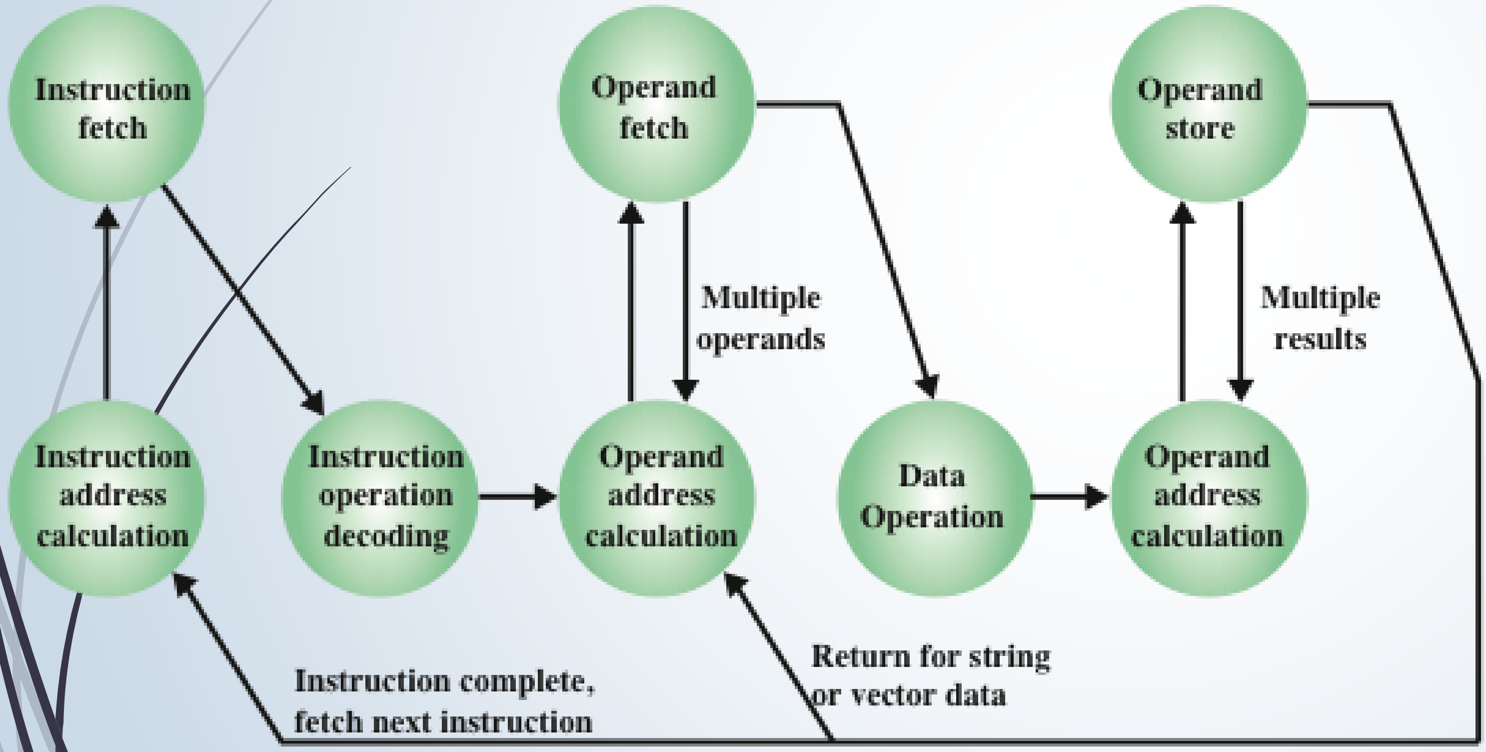
Result operand reference

- The operation may produce a **result**

Next instruction reference

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

Instruction Cycle State Diagram



The address of the next instruction to be fetched could be either a **real address** or a **virtual address**. Generally, the distinction is *transparent* to the instruction set architecture.

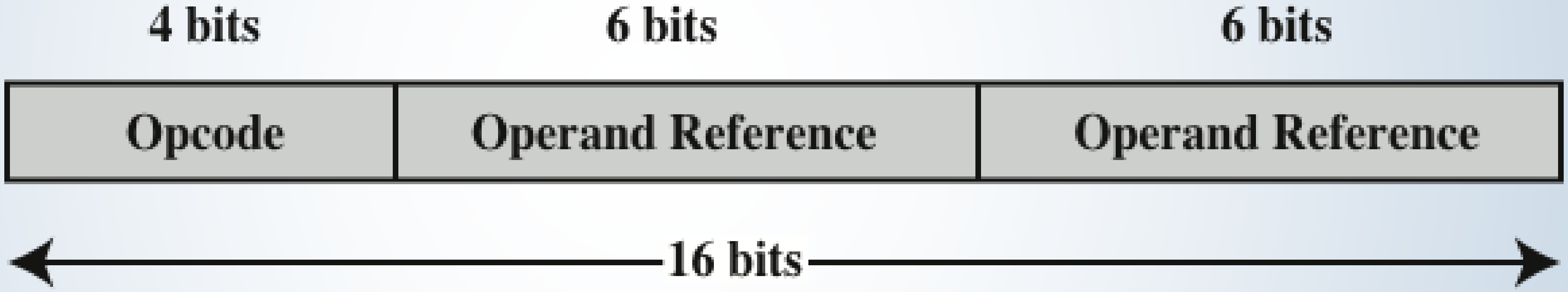
In most cases, the next instruction to be fetched immediately follows the current instruction. In those cases, there is no explicit reference to the next instruction. When an explicit reference is needed, then the main memory or virtual memory address must be supplied.

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

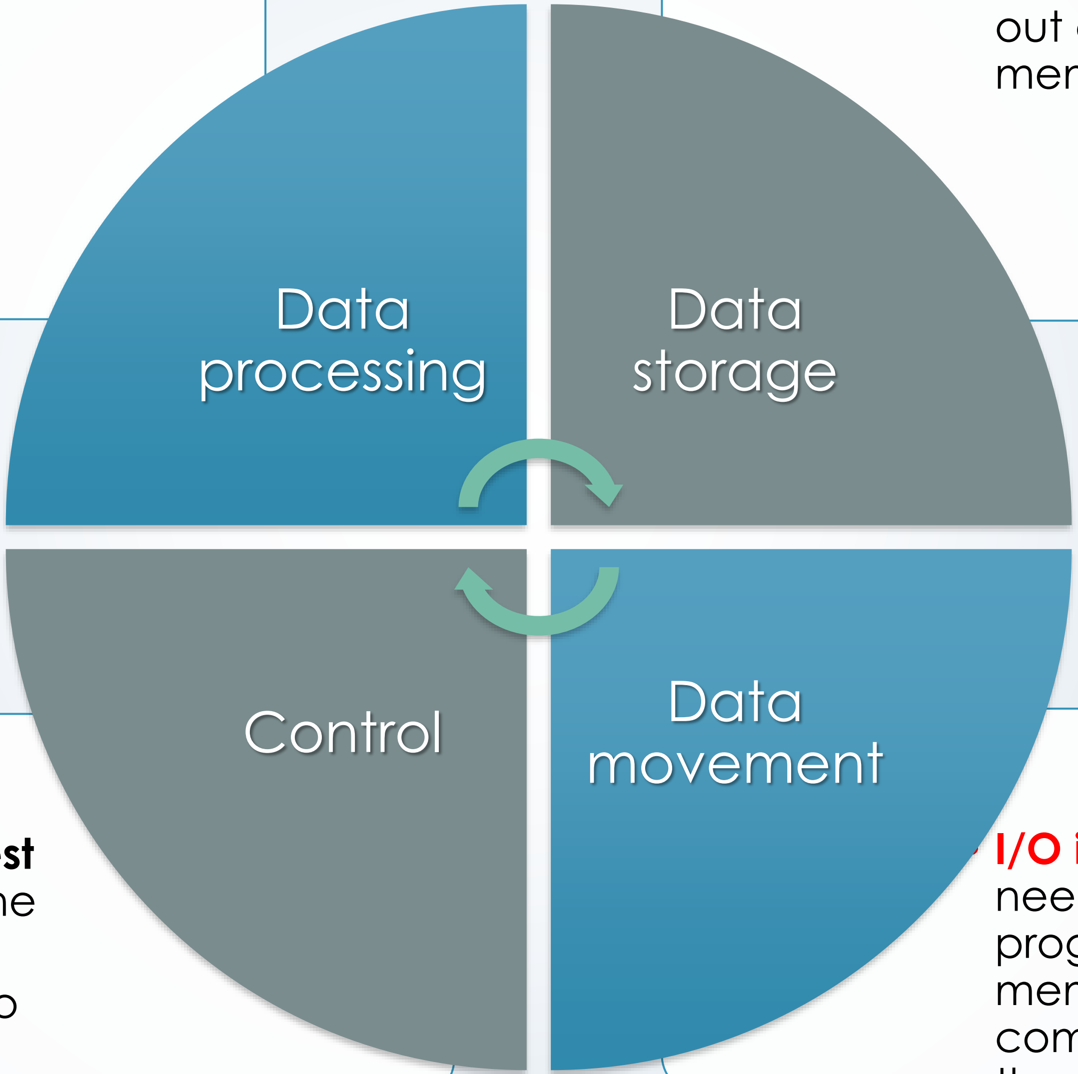


During instruction execution, an instruction is read into an instruction register (IR) in the processor. The processor must be able to extract the data from the various instruction fields to perform the required operation.

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 for 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 used to **test the value** of a data 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

Number of Addresses

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

In one-address instruction, a second address must be implicit. This was common in earlier machines, with the implied address being a processor register known as the **accumulator** (AC). The accumulator contains one of the operands and is used to store the result.

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

zero addresses for some instructions?

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

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

The number of addresses per instruction is a basic design decision. Fewer addresses per instruction result in instructions that are more primitive, requiring a less complex processor. It also results in instructions of shorter length. On the other hand, programs contain more total instructions, which in general results in longer execution times and longer, more complex programs.

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

- How many and which operations to provide and how complex operations should be

Data types

- The various types of data upon which operations are performed

Instruction format

- Instruction length in bits, number of addresses, size of various fields, etc.

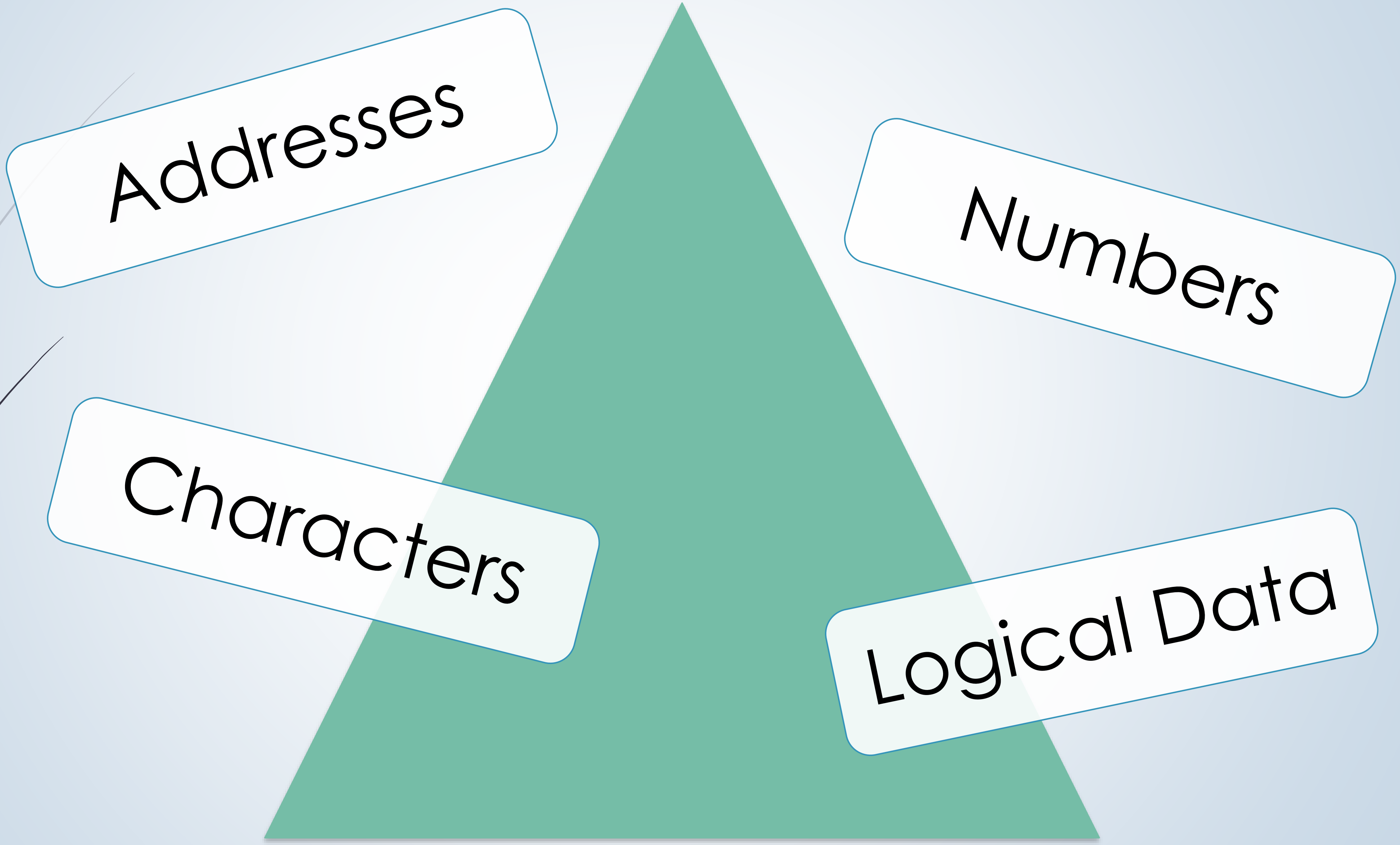
Registers

- Number of processor registers that can be referenced by instructions and their use

Addressing

- The mode or modes by which the address of an operand is specified

Types of Operands



Numbers

- All machine languages include **numeric** data types
- Numbers stored in a computer are limited:
  - Limit to the magnitude of numbers representable on a machine
  - 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
- Most 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:
  - Memory can be used most 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)
  - 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

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

x86 Numeric Data Formats

float a;

a = 0.2;

0 | 01111100 | 1001100110011001100 ...

+ 127

$2^{k-1} - 1$

$0.2 \times 2 = 0.4$

$0.4 \times 2 = 0.8$

$0.8 \times 2 = 1.6$

$0.6 \times 2 = 1.2$

$0.2 \times 2 = 0.4$

$0.4 \times 2 = 0.8$

0 0 1 1 0 0 ...

0.0011001100110011 ...

$1.10011001100110011 \ldots \times 2^{-3}$

127 − 3 = 124

0	1	1	1	1	1	0	0
128	64	32	16	8	4	2	1

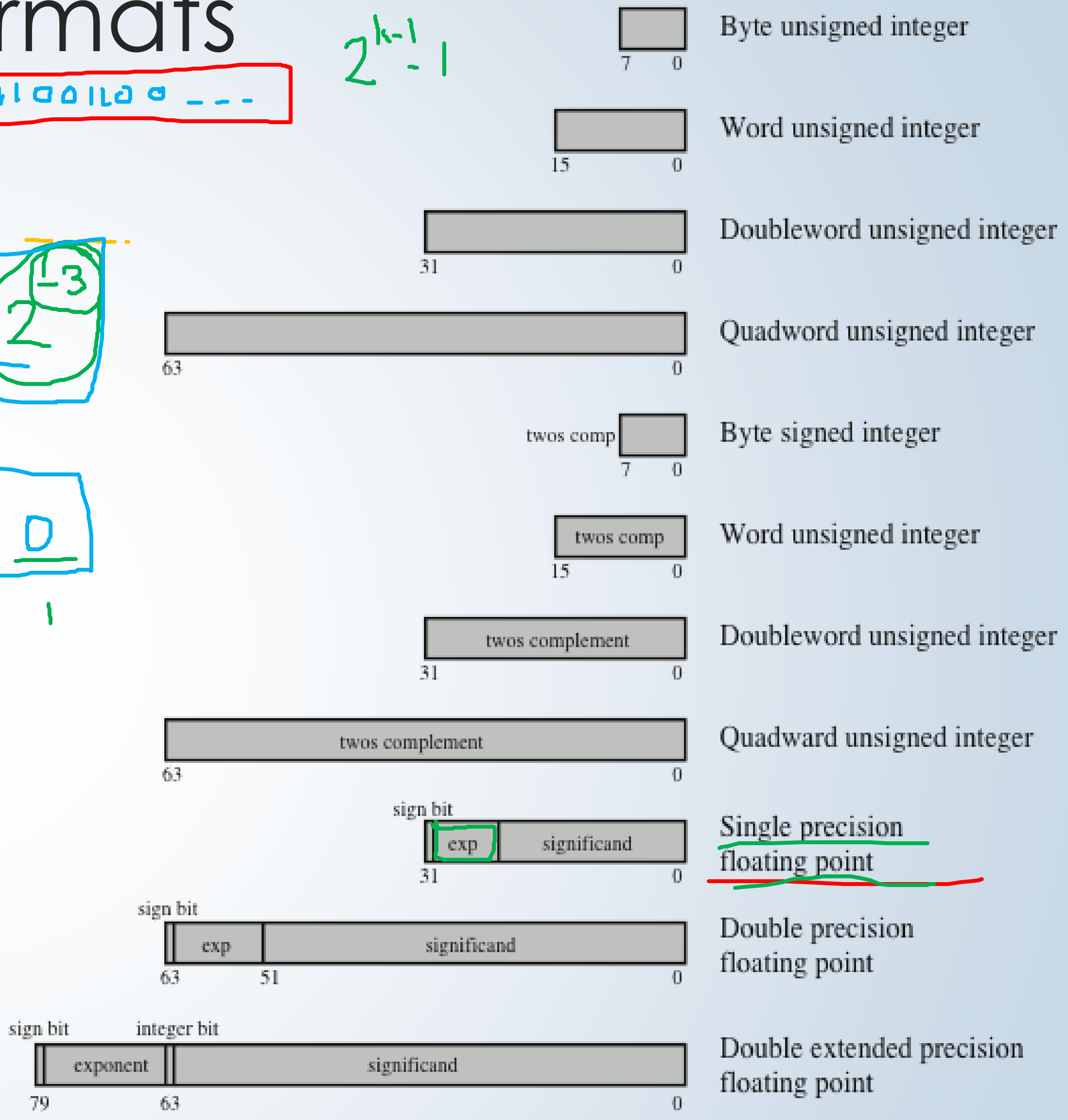


Figure 12.4 x86 Numeric Data Formats

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

- 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)
- Data types:
  - Packed byte and packed byte integer
  - Packed word and packed word integer
  - Packed doubleword and packed doubleword integer
  - Packed quadword and packed quadword integer
  - Packed single-precision floating-point and packed double-precision floating-point

The basic concept is that multiple operands are packed into a single referenced memory item and that these multiple operands are operated on in parallel.

ARM Data Types

ARM processors support data types of:

- 8 (byte)
- 16 (halfword)
- 32 (word) bits in length

All three data types can also be used for twos complement signed integers

For all three data types an unsigned interpretation is supported in which the value represents an unsigned, nonnegative integer

Alignment checking

- When the appropriate control bit is set, a data abort signal indicates an alignment fault for attempting unaligned access

Unaligned access

- When this option is enabled, the processor uses one or more memory accesses to generate the required transfer of adjacent bytes transparently to the programmer

ARM Endian Support

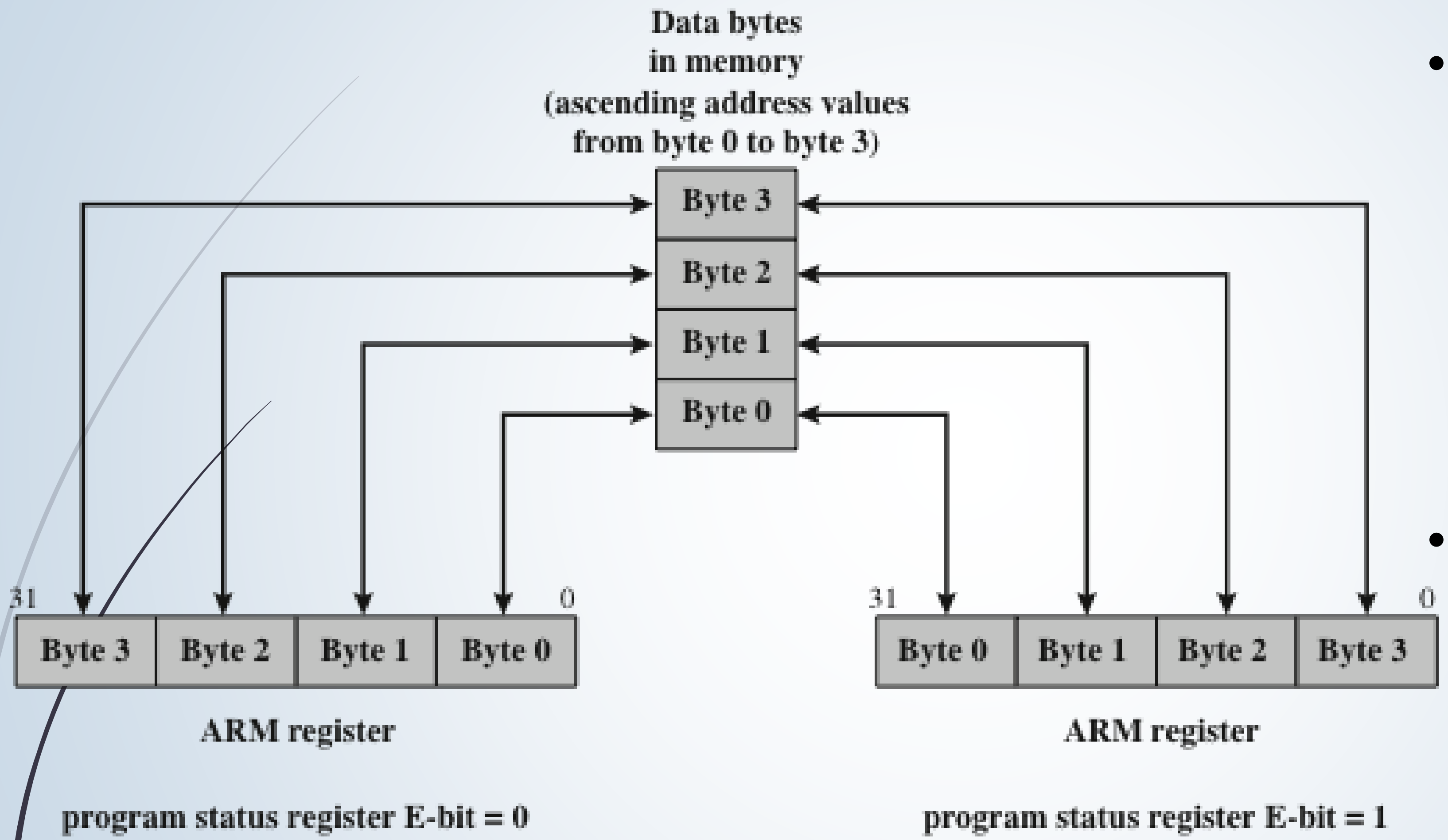


Figure 12.5 ARM Endian Support - Word Load/Store with E-bit

- The E-bit defines which endian to load and store data. Figure 12.5 illustrates the functionality associated with the E-bit for a word load or store operation.
- This mechanism enables efficient dynamic data load/store for system designers who know they need to access data structures in the opposite endianness to their OS/environment

Common Instruction Set Operations

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

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)

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

Most fundamental type of machine instruction

Must specify:

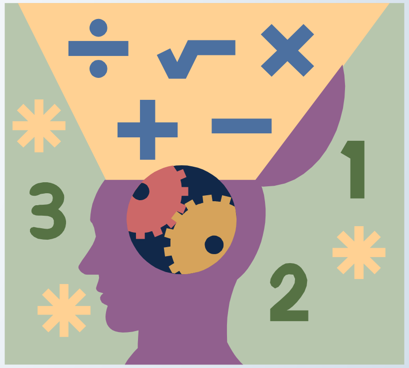
- Location of the source and destination operands
- The length of data to be transferred must be indicated
- The mode of addressing for each operand must be specified

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
- Often they are also provided for floating-point and packed decimal numbers
- Other possible operations include a variety of 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

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

Table 12.6 Basic Logical Operations

Shift and Rotate Operations

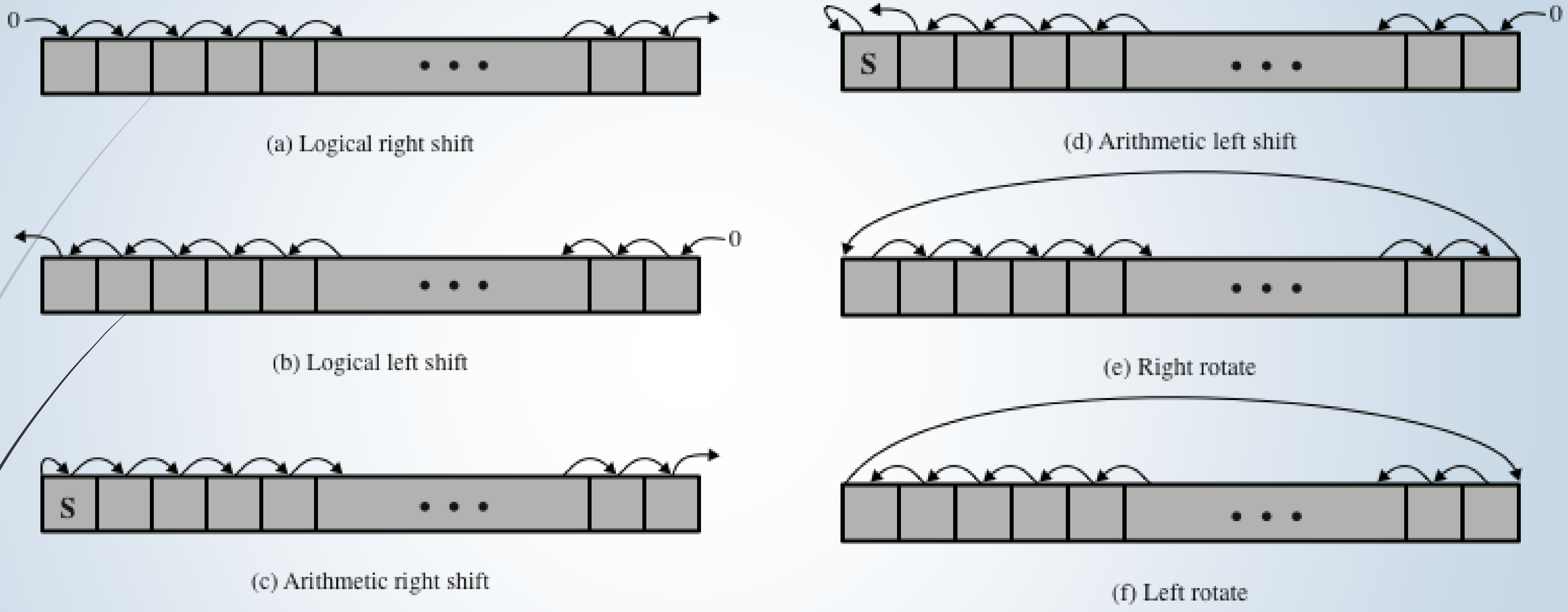


Figure 12.6 Shift and Rotate Operations

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

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

Programmed I/O:

- Programmed I/O is the simplest and most basic I/O technique.
- In programmed I/O, the CPU performs all the tasks related to data transfer between the I/O device and memory.
- The CPU sends commands to the I/O device, waits for the device to complete the operation, and then transfers data between the device and memory.
- It involves frequent polling or busy waiting by the CPU, which can be inefficient as the **CPU is occupied during the entire I/O operation**.
- Programmed I/O is suitable for low-speed devices, where the CPU can afford to spend time waiting for the I/O operations to complete.

Memory-Mapped I/O:

- Memory-mapped I/O uses the same address space for both memory and I/O devices.
- The I/O devices are assigned specific memory addresses, and the CPU accesses them through memory read and write operations.
- The advantage of memory-mapped I/O is that it allows I/O devices to be accessed using the same load and store instructions used for memory access.
- Memory-mapped I/O eliminates the need for separate I/O instructions and dedicated I/O registers, simplifying the programming model.
- However, it can lead to contention for memory access if the CPU and I/O devices access the same memory simultaneously.

DMA (Direct Memory Access):

- DMA is an advanced I/O technique that offloads the data transfer task from the CPU to a dedicated DMA controller.
- With DMA, the I/O device transfers data directly to or from memory without CPU intervention, freeing the CPU to perform other tasks.
- The DMA controller takes over the bus control and coordinates the data transfer between the I/O device and memory.
- DMA is particularly beneficial for high-speed devices or large data transfers as it reduces CPU overhead and improves overall system performance.
- However, setting up DMA transfers and managing shared resources requires careful synchronization and coordination to avoid conflicts.

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

Typically 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
- Most common transfer-of-control operations found in instruction sets:
  - **Branch**
  - **Skip**
  - **Procedure call**

Branch Instruction

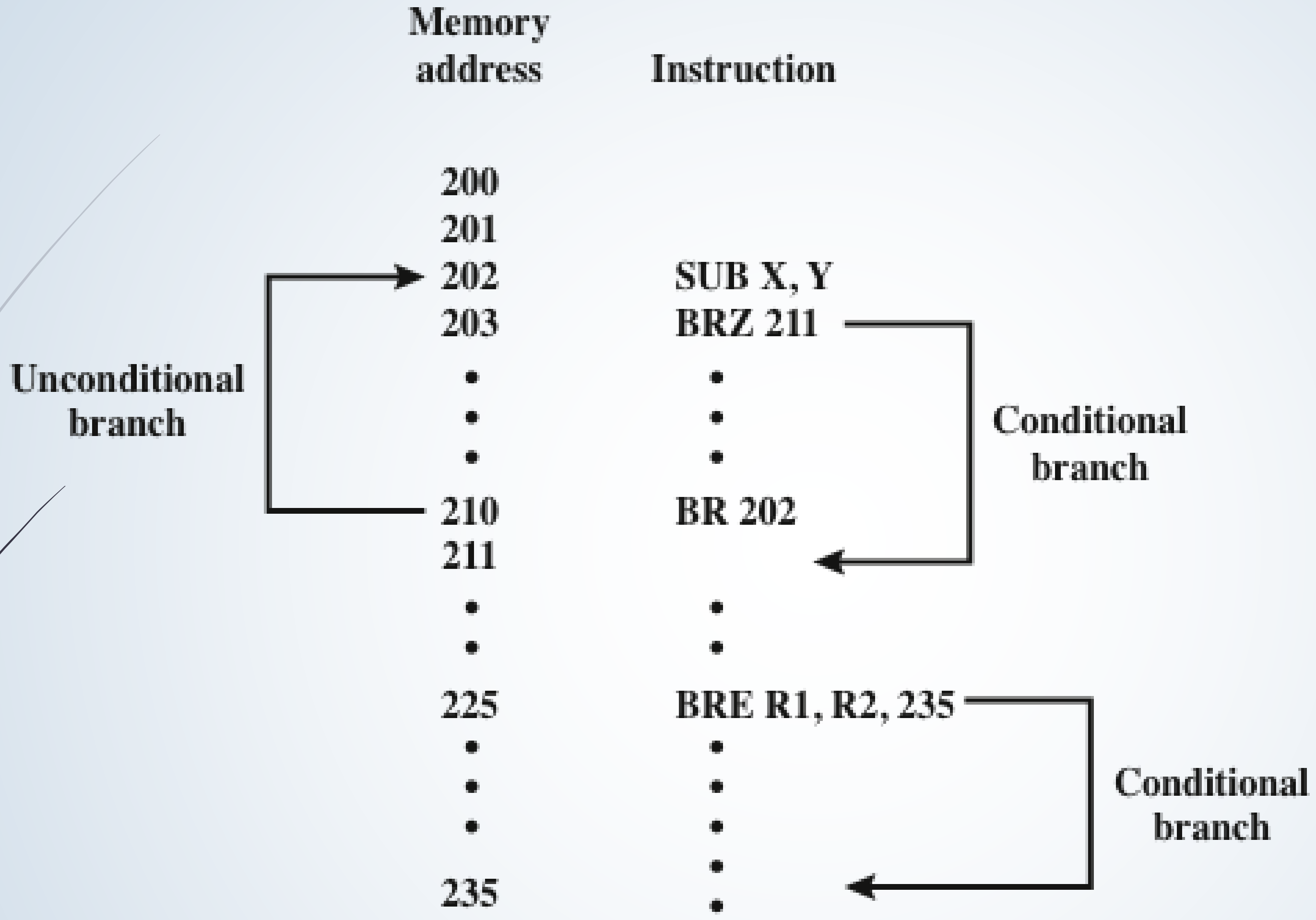


Figure 12.7 Branch Instructions

JUMP

Note that a branch can be either *forward* (an instruction with a higher address) or *backward* (lower address).

Skip Instructions

Includes an implied address

Typically implies that one instruction be skipped, thus the implied address equals the address of the next instruction plus one instruction length

Because the skip instruction does not require a destination address field it is free to do other things

Example is the increment-and-skip-if-zero (ISZ) instruction

Procedure Call Instructions

- 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
  - Modularity
- **Involves two basic instructions:**
  - A call instruction that branches from the present location to the procedure
  - Return instruction that returns from the procedure to the place from which it was called

Nested Procedures

- Main program starting at location 4000.
- This program includes a call to procedure PROC1, starting at location 4500.
- When this call instruction is encountered, the processor suspends execution of the main program and begins execution of PROC1 by fetching the next instruction from location 4500.
- Within PROC1, there are two calls to PROC2 at location 4800. In each case, the execution of PROC1 is suspended and PROC2 is executed
- The RETURN statement causes the processor to go back to the calling program and continue execution at the instruction after the corresponding CALL instruction.

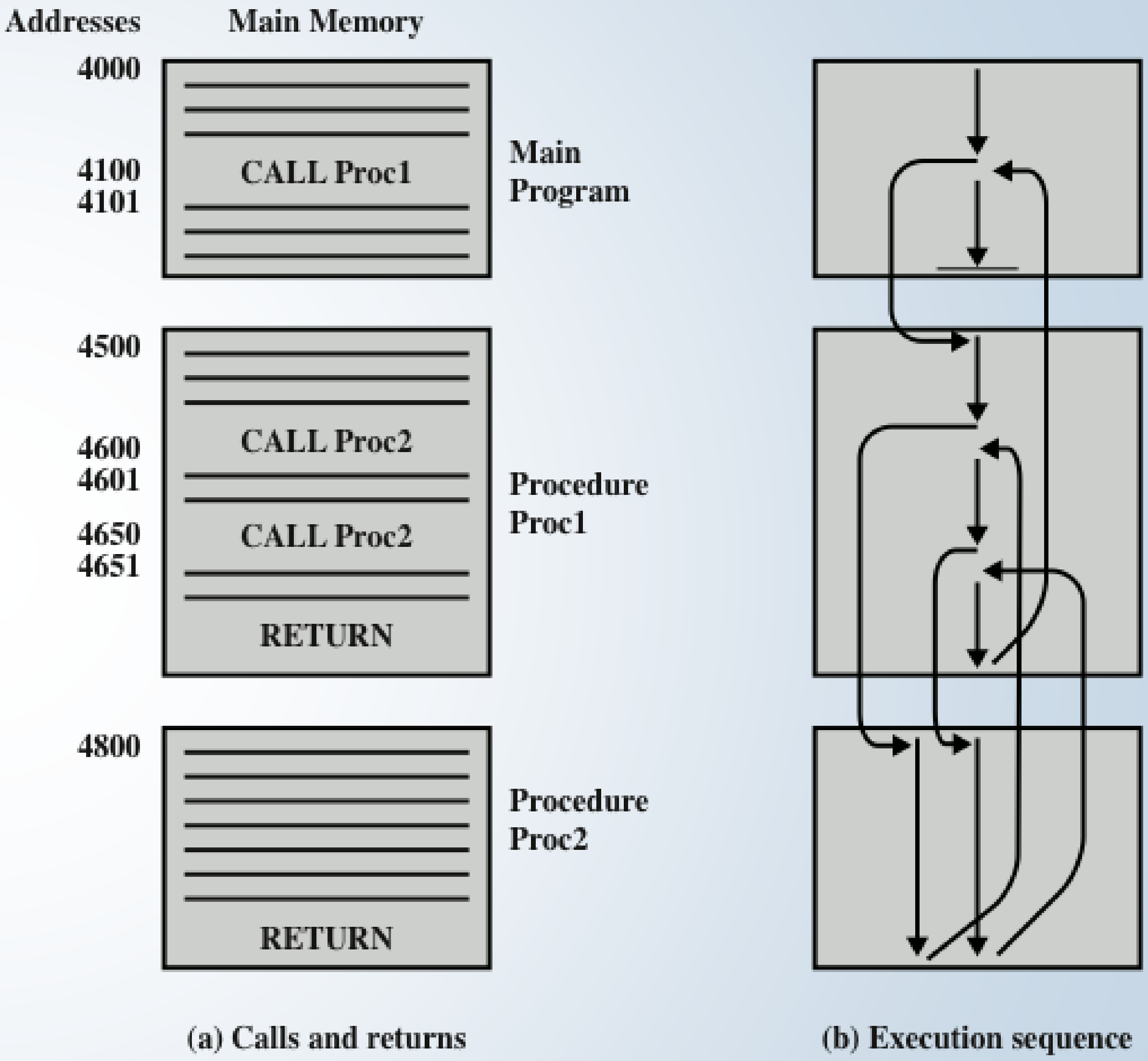


Figure 12.8 Nested Procedures

Use of Stack to Implement Nested Procedures

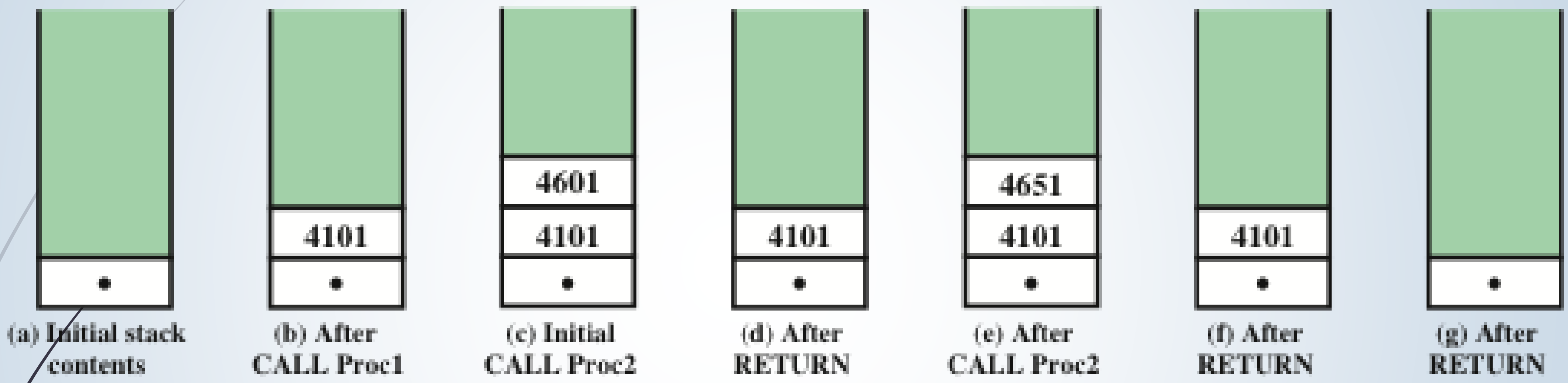


Figure 12.9 Use of Stack to Implement Nested Procedures of Figure 12.8

Stack Frame

- A stack frame, also known as an activation record or stack frame record, is a data structure used by a computer program during its execution to manage function calls and local variables.
- When a function is called, a new stack frame is created and pushed onto the top of the call stack. The stack frame contains the following information:
  - **Return Address:** memory address where the program should return after the function call completes
  - **Function Parameters:** Stores the parameters passed to the function.
  - Local variables declared within the function
  - Saved Registers: Some processor architectures require saving certain registers before executing a function call

Stack Frame Growth Using Sample Procedures P and Q

- A more flexible approach to parameter passing is the stack.
- When the processor executes a call, it not only stacks the return address, it stacks parameters to be passed to the called procedure.
- The called procedure can access the parameters from the stack. Upon return, return parameters can also be placed on the stack.
- The entire set of parameters, including return address, that is stored for a procedure invocation is referred to as a *stack frame.*

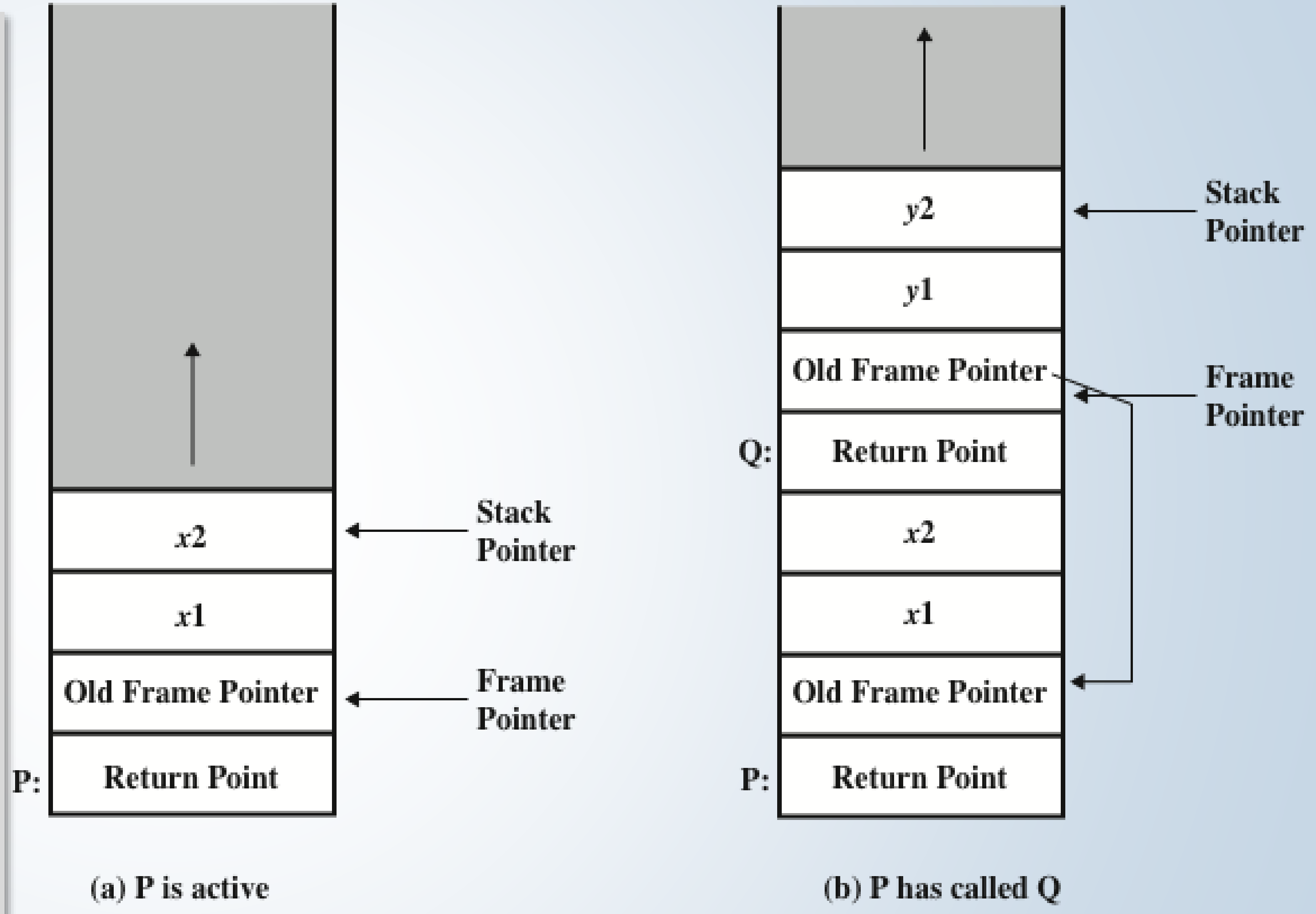


Figure 1.10 Stack Frame Growth Using Sample Procedures P and Q

Instruction	Description
Data Movement	
MOV	Move operand, between registers or between register and memory.
PUSH	Push operand onto stack.
PUSHA	Push all registers on stack.
MOVSX	Move byte, word, dword, sign extended. Moves a byte to a word or a word to a doubleword with twos-complement sign extension.
LEA	Load effective address. Loads the offset of the source operand, rather than its value to the destination operand.
XLAT	Table lookup translation. Replaces a byte in AL with a byte from a user-coded translation table. When XLAT is executed, AL should have an unsigned index to the table. XLAT changes the contents of AL from the table index to the table entry.
IN, OUT	Input, output operand from I/O space.
Arithmetic	
ADD	Add operands.
SUB	Subtract operands.
MUL	Unsigned integer multiplication, with byte, word, or double word operands, and word, doubleword, or quadword result.
IDIV	Signed divide.
Logical	
AND	AND operands.
BTS	Bit test and set. Operates on a bit field operand. The instruction copies the current value of a bit to flag CF and sets the original bit to 1.
BSF	Bit scan forward. Scans a word or doubleword for a 1-bit and stores the number of the first 1-bit into a register.
SHL/SHR	Shift logical left or right.
SAL/SAR	Shift arithmetic left or right.
ROL/ROR	Rotate left or right.
SETcc	Sets a byte to zero or one depending on any of the 16 conditions defined by status flags.
Control Transfer	
JMP	Unconditional jump.
CALL	Transfer control to another location. Before transfer, the address of the instruction following the CALL is placed on the stack.
JE/JZ	Jump if equal/zero.
LOOPE/LOOPZ	Loops if equal/zero. This is a conditional jump using a value stored in register ECX. The instruction first decrements ECX before testing ECX for the branch condition.
INT/INTO	Interrupt/Interrupt if overflow. Transfer control to an interrupt service routine

Table 12.8

x86
Operation Types
(With Examples of
Typical Operations)

(page 1 of 2)

String Operations	
MOVS	Move byte, word, dword string. The instruction operates on one element of a string, indexed by registers ESI and EDI. After each string operation, the registers are automatically incremented or decremented to point to the next element of the string.
LODS	Load byte, word, dword of string.
High-Level Language Support	
ENTER	Creates a stack frame that can be used to implement the rules of a block-structured high-level language.
LEAVE	Reverses the action of the previous ENTER.
BOUND	Check array bounds. Verifies that the value in operand 1 is within lower and upper limits. The limits are in two adjacent memory locations referenced by operand 2. An interrupt occurs if the value is out of bounds. This instruction is used to check an array index.
Flag Control	
STC	Set Carry flag.
LAHF	Load AH register from flags. Copies SF, ZF, AF, PF, and CF bits into A register.
Segment Register	
LDS	Load pointer into DS and another register.
	System Control
HLT	Halt.
LOCK	Asserts a hold on shared memory so that the Pentium has exclusive use of it during the instruction that immediately follows the LOCK.
ESC	Processor extension escape. An escape code that indicates the succeeding instructions are to be executed by a numeric coprocessor that supports high-precision integer and floating-point calculations.
WAIT	Wait until BUSY# negated. Suspends Pentium program execution until the processor detects that the BUSY pin is inactive, indicating that the numeric coprocessor has finished execution.
Protection	
SGDT	Store global descriptor table.
LSL	Load segment limit. Loads a user-specified register with a segment limit.
VERR/VERW	Verify segment for reading/writing.
Cache Management	
INVD	Flushes the internal cache memory.
WBINVD	Flushes the internal cache memory after writing dirty lines to memory.
INVLPG	Invalidates a translation lookaside buffer (TLB) entry.

Table 12.8

x86
Operation Types
(With Examples of
Typical Operations)

(page 2 of 2)

Call/Return Instructions

- The x86 provides four instructions to support procedure call/return:
  - CALL
  - ENTER
  - LEAVE
  - RETURN
- Common means of implementing the procedure is via the use of stack frames
- The CALL instruction pushes the current instruction pointer value onto the stack and causes a jump to the entry point of the procedure by placing the address of the entry point in the instruction pointer

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.

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)

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 and ARM data types
- Types of operations
  - Data transfer
  - Arithmetic
  - Logical
  - Conversion
  - Input/output
  - System control
  - Transfer of control