AAiT

Computer Architecture & Organization

Chapter 5

Instruction Sets and Addressing Modes

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

AAiT

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

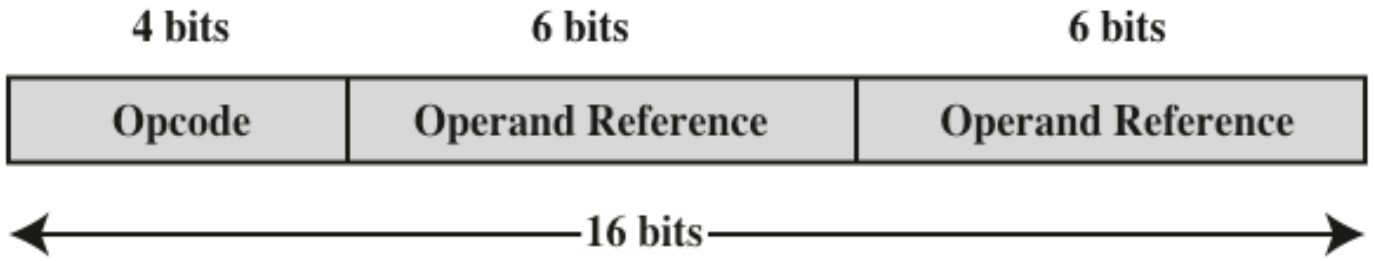


Figure 12.2 A Simple Instruction Format

Number of Addresses

- Figure 12.3 compares typical one, two, and three address instructions that could be used to compute Y = (A – B)/[C + (D * E)].
- Three-address instruction formats are not common because they require a relatively long instruction format to hold the three address references.
- The two-address format reduces the space requirement but also introduces some awkwardness.
- Simpler yet is the one-address instruction. For this to work, a second address must be implicit (AC).
- Zero-address instructions are applicable to a special memory organization called a *stack*.

Instruction	Comment
SUB Y, A, B	Y ← A – B
MPY T, D, E	T ← D × E
ADD T, T, C	T ← T + C
DIV Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction	Comment
MOVE Y, A	Y ← A
SUB Y, B	Y ← Y – B
MOVE T, D	T ← D
MPY T, E	T ← T × E
ADD T, C	T ← T + C
DIV Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction	Comment
LOAD D	AC ← D
MPY E	AC ← AC × E
ADD C	AC ← AC + C
STOR Y	Y ← AC
LOAD A	AC ← A
SUB B	AC ← AC – B
DIV Y	AC ← AC ÷ Y
STOR Y	Y ← AC

(c) One-address instructions

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

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

- 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
- There is an important threshold between one-address and multiple-address instructions.
- The design trade-offs involved in choosing the number of addresses per instruction are complicated by other factors. There is the issue of whether an address references a memory location or a register. Because there are fewer registers, fewer bits are needed for a register reference.
- The result is that most processor designs involve a variety of instruction formats.

Instruction Set Design

- One of the most interesting, and most analyzed, aspects of computer design is 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

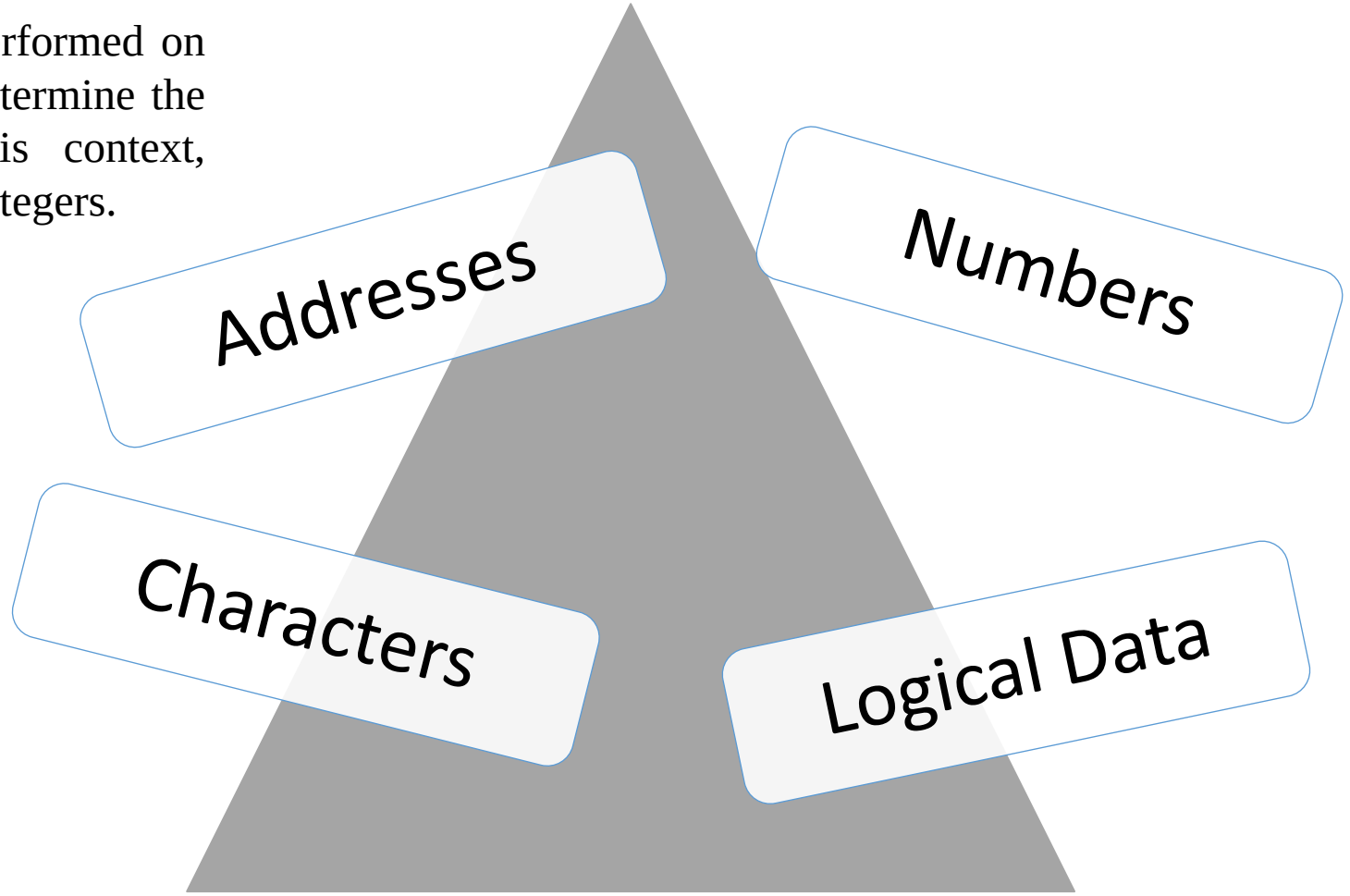
Fundamental design issues:

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

Types of Operands

- In many cases, some calculation must be performed on the operand reference in an instruction to determine the main or virtual memory address. In this context, addresses can be considered to be unsigned integers.

- Other common data types are numbers, characters, and logical data, and each of these is briefly examined in this section.
- Beyond that, some machines define specialized data types or data structures. For example, there may be machine operations that operate directly on a list or a string of characters.



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)
  - Note in Table F.1 (Appendix F) that for the IRA bit pattern 011XXXX, the digits 0 through 9 are represented by their binary equivalents, 0000 through 1001.
- Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC)
  - EBCDIC is used on IBM mainframes
  - As with IRA, EBCDIC is compatible with packed decimal. In the case of EBCDIC, the codes *1111*0000 through *1111*1001 represent the digits 0 through 9.

Logical Data

- Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. It is sometimes useful, however, to consider an *n-bit* unit as consisting of *n* 1-bit items of data, each item having the value 0 or 1. When data are viewed this way, they are considered to be *logical* data.
- 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

AAiT

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 contained 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

- The x86 can deal with data types of
  - 8 (byte),
  - 16 (word),
  - 32 (doubleword),
  - 64 (quad- word), and
  - 128 (double quadword) bits in length.
- To allow maximum flexibility in data structures and efficient memory utilization, words need not be aligned at even- numbered addresses.
  - E.g. quadwords need not be aligned at addresses evenly divisible by 8
- The x86 uses the little-endian style

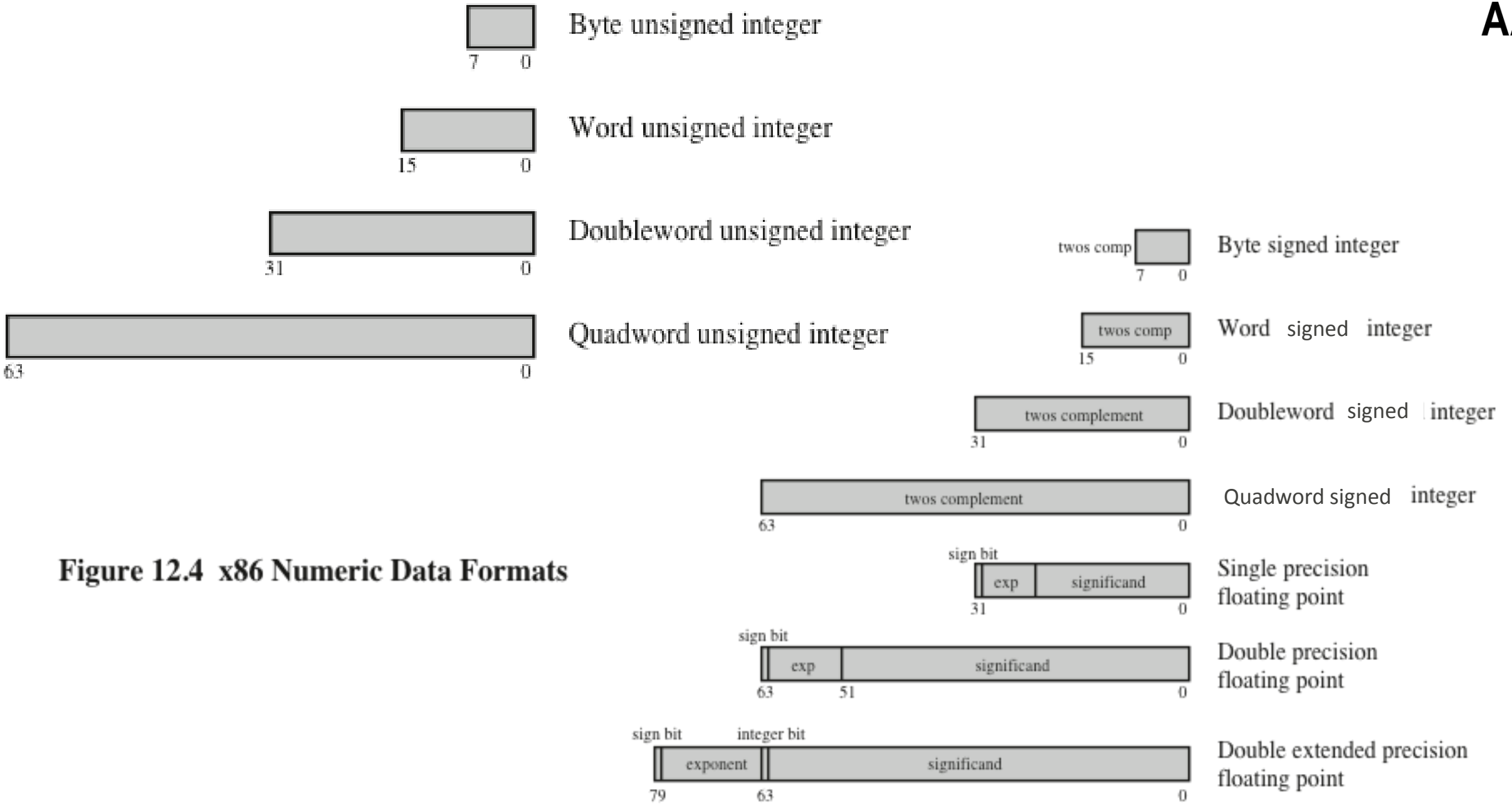


Figure 12.4 x86 Numeric Data Formats

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

AAiT

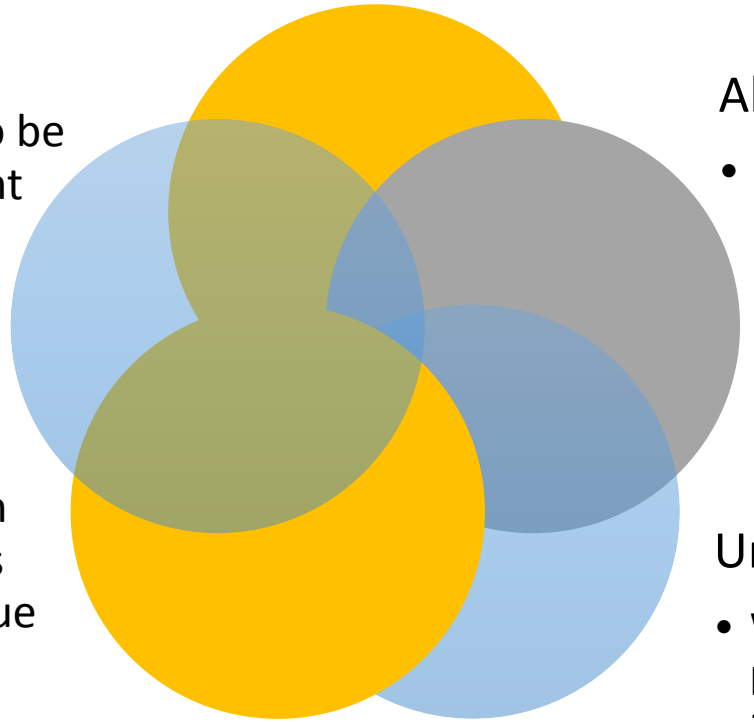
- Introduced to the x86 architecture as part of the extensions of the instruction set to optimize performance of multimedia applications
- These extensions include MMX (multimedia extensions) and SSE (streaming SIMD extensions)
- Data types:
  - **Packed byte and packed byte integer:** Bytes packed into 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 as an 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

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

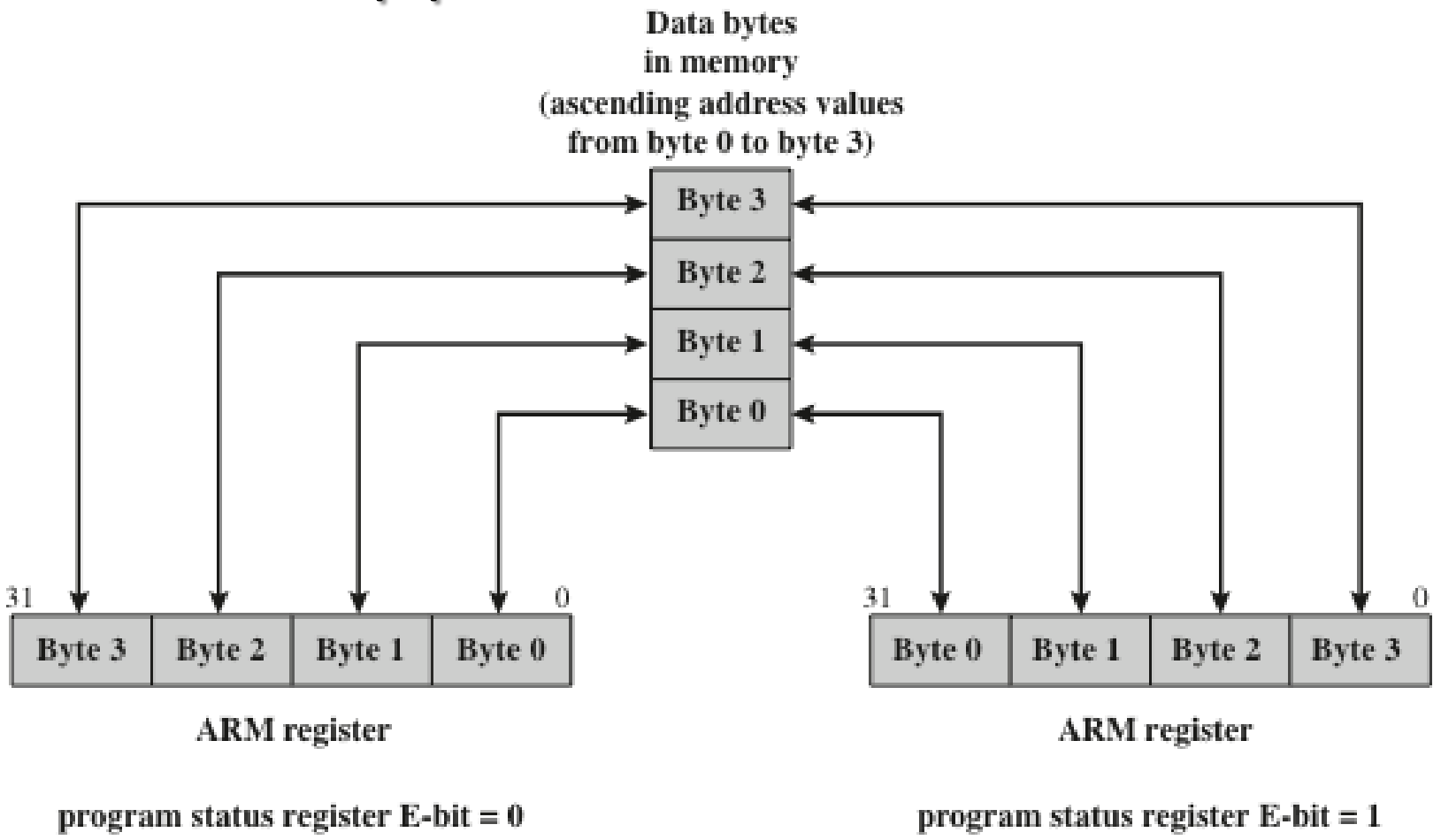
Alignment checking

- 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



- The ARM supports both little-endian and big-endian style

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

“

Danny Cohen introduced the terms
Little-Endian and **Big-Endian**
for byte ordering in an article from 1980.
In this technical and political examination of byte ordering issues, the "endian" names were drawn from Jonathan Swift's 1726 satire, *Gulliver's Travels*, in which civil war erupts over whether the **big end** or the **little end** of a boiled egg is the proper end to crack open, which is analogous to counting from the end that contains the

”

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

AAiT

Table 12.3

Common Instruction Set Operations

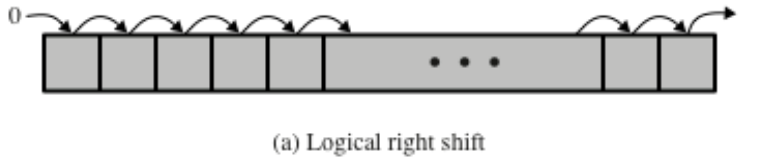
(page 2 of 2)

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

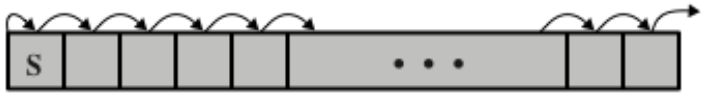
Data transfer	Transfer data from one location to another
	If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes 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

Shift and Rotate Operations



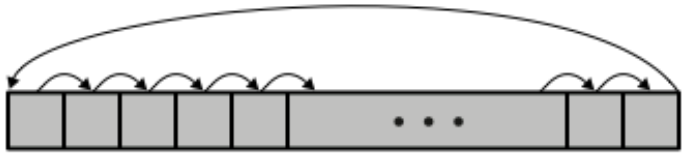
(a) Logical right shift

(b) Logical left shift

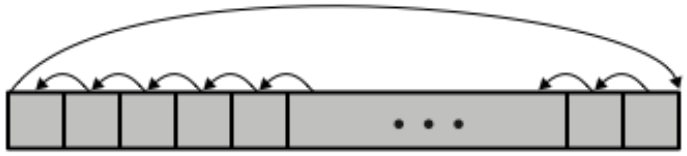


(c) Arithmetic right shift

(d) Arithmetic left shift



(e) Right rotate



(f) Left rotate

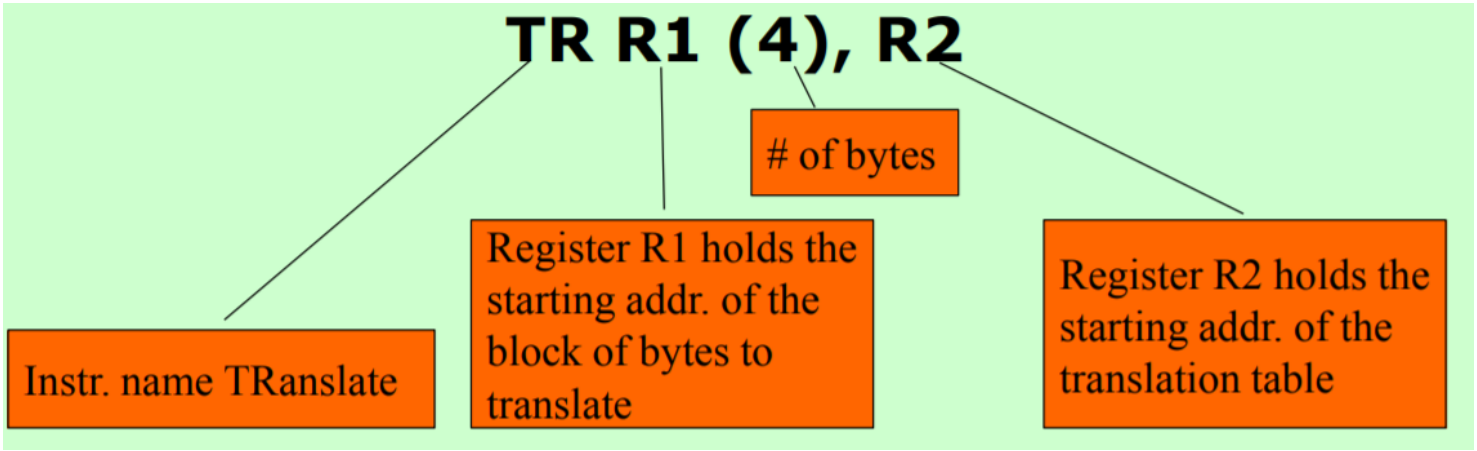
Figure 12.6 Shift and Rotate Operations

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

Conversion instructions

- Instructions 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: TR R1 (L), R2



- **For example,**
  Translate 1984 in EBCDIC to IRA,
  - Create a 256-byte table in storage locations, say, 1000-10FF.
  - This table contains the IRA translation, locations 10F0 through 10F9 will contain the values 30 through 39
  - Locations 2100–2103 contain F1 F9 F8 F4.
  - R1 contains 2100.
  - R2 contains 1000.
    Then, if we execute TR R1 (4), R2
  - locations 2100–2103 will contain 31 39 38 34.

Exercise:
What will be content of the translation table if we want to translate (R2) 1984 in IRA to EBCDIC

System Control Instructions

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

- A branch instruction, also called a jump instruction, has as one of its operands the address of the next instruction to be executed.
- Most often, the instruction is a conditional branch instruction. That is, the branch is made (update program counter to equal address specified in operand) only if a certain condition is met. Otherwise, the next instruction in sequence is executed (increment program counter as usual).
- A branch instruction in which the branch is always taken is an unconditional branch

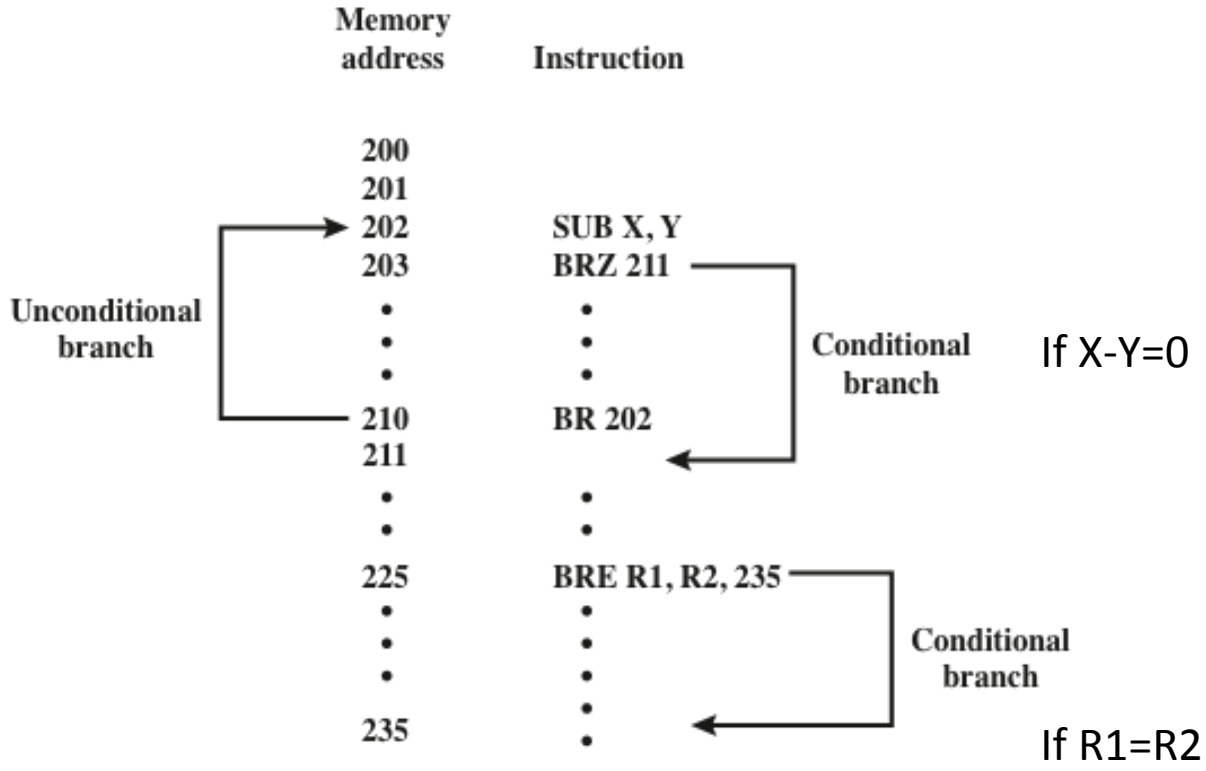
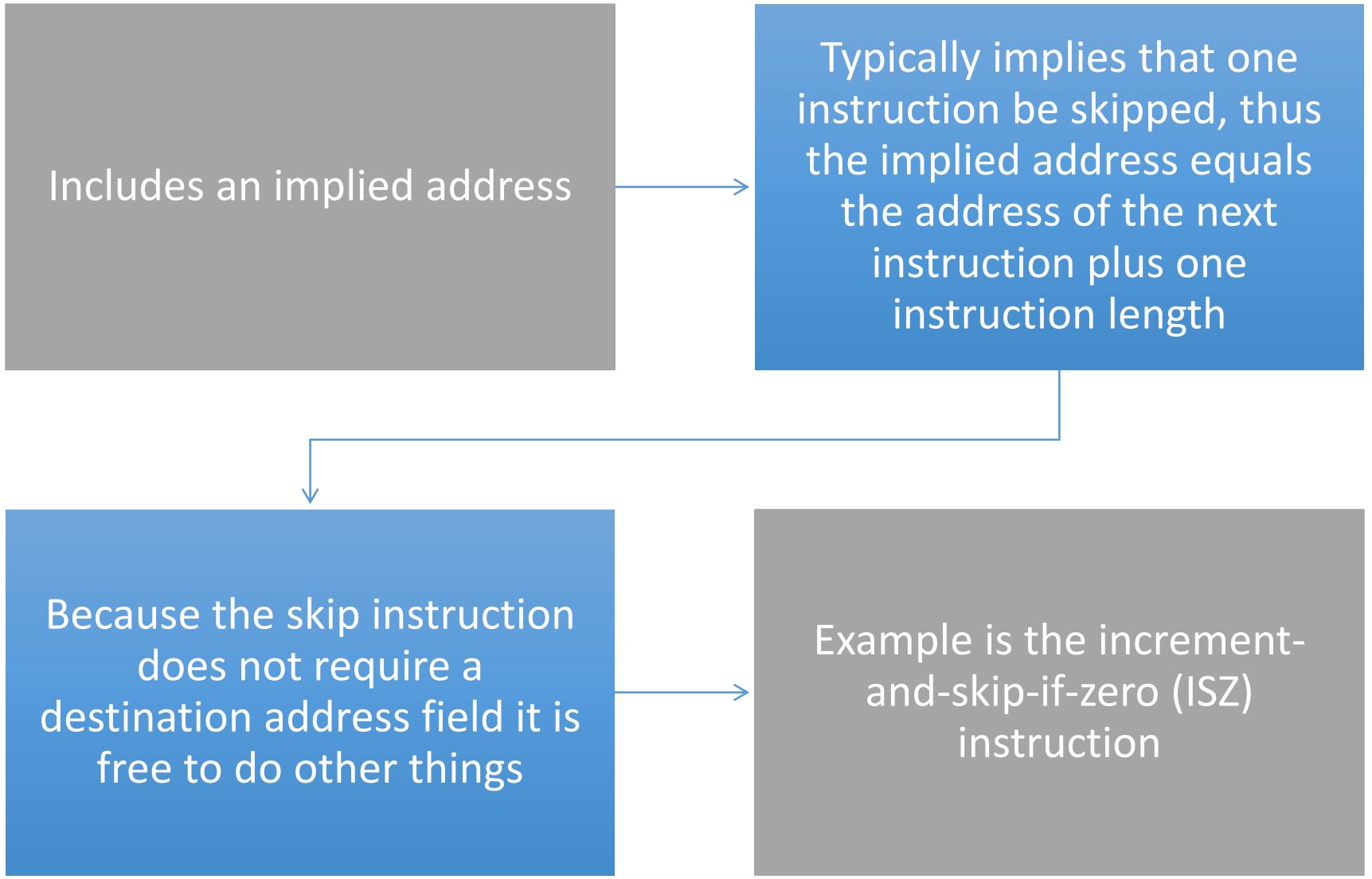


Figure 12.7 Branch Instructions

- Figure 12.7 shows examples of these operations

Skip Instructions



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

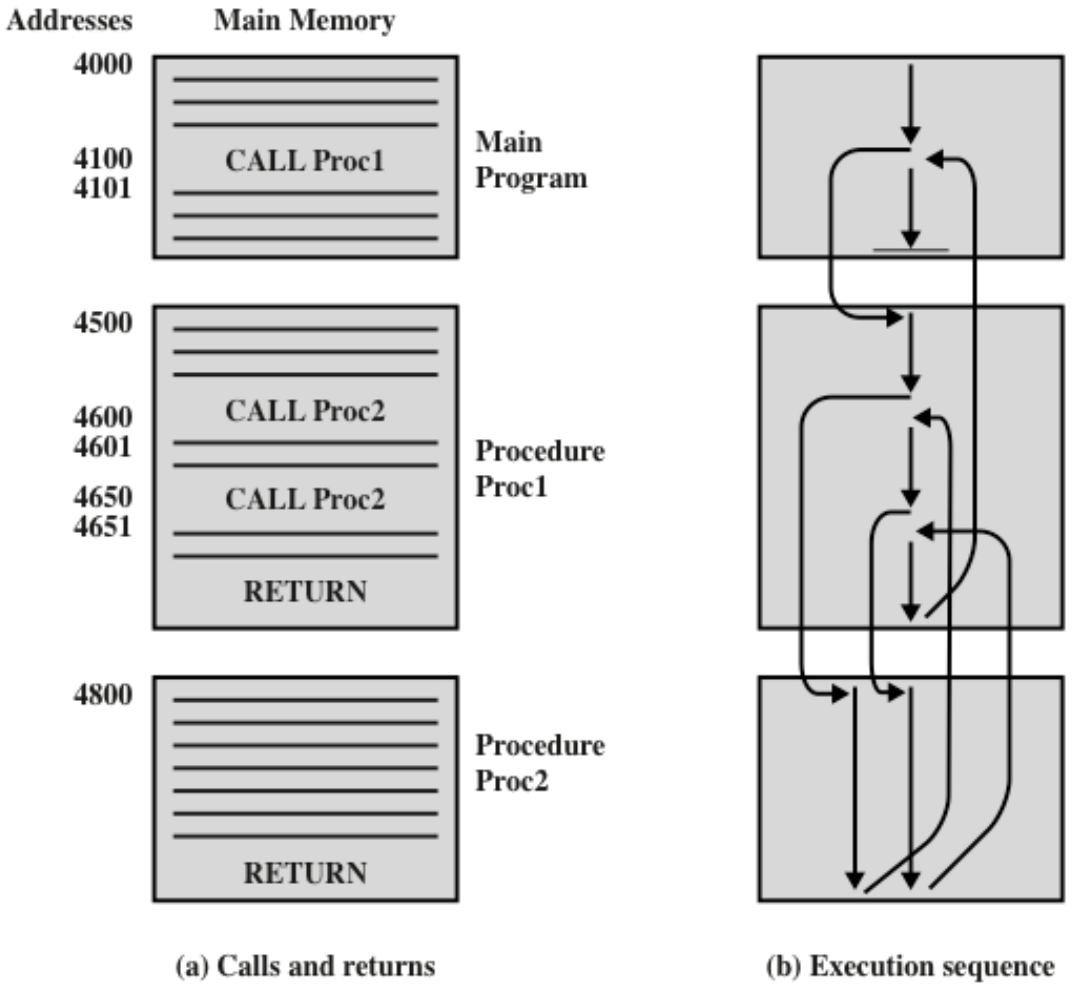


Figure 12.8 Nested Procedures

AAiT

Use of Stack to Implement Nested Procedures

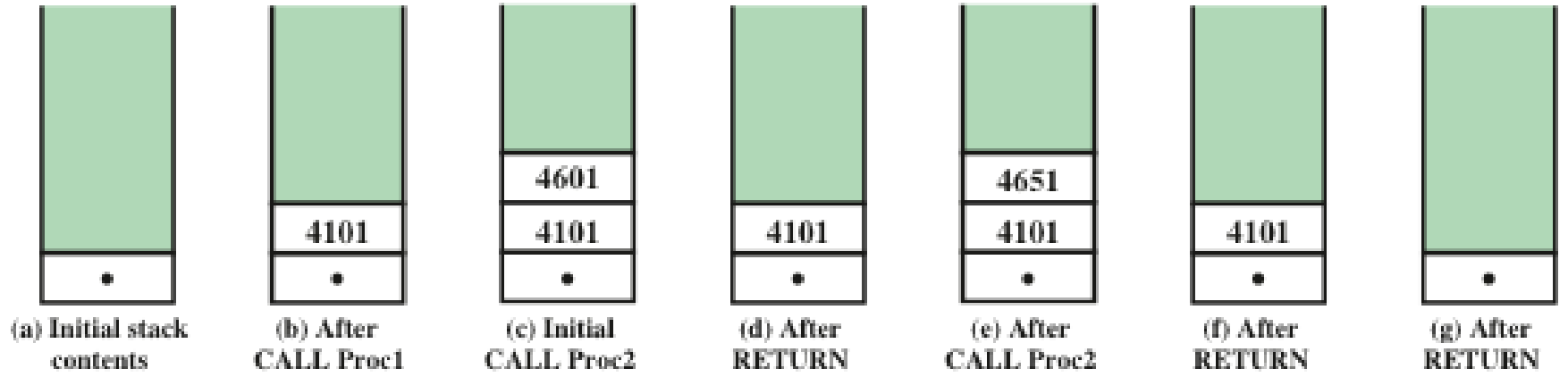


Figure 12.9 Use of Stack to Implement Nested Procedures of Figure 12.8

Exercise:

Show the state of the stack at every recursive call while solving Fibonacci (6)

```
Fibonacci(n)
{
    if (n < 1)
        return 0;
    else
        return (Fibonacci (n - 1) + n );
}
```

Stack Frame Growth Using Sample Procedures P and Q

AAiT

- 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 12.10 example refers to procedure P in which the local variables *x1* and *x2* are declared, and procedure Q, which P can call and in which the local variables *y1* and *y2* are declared.

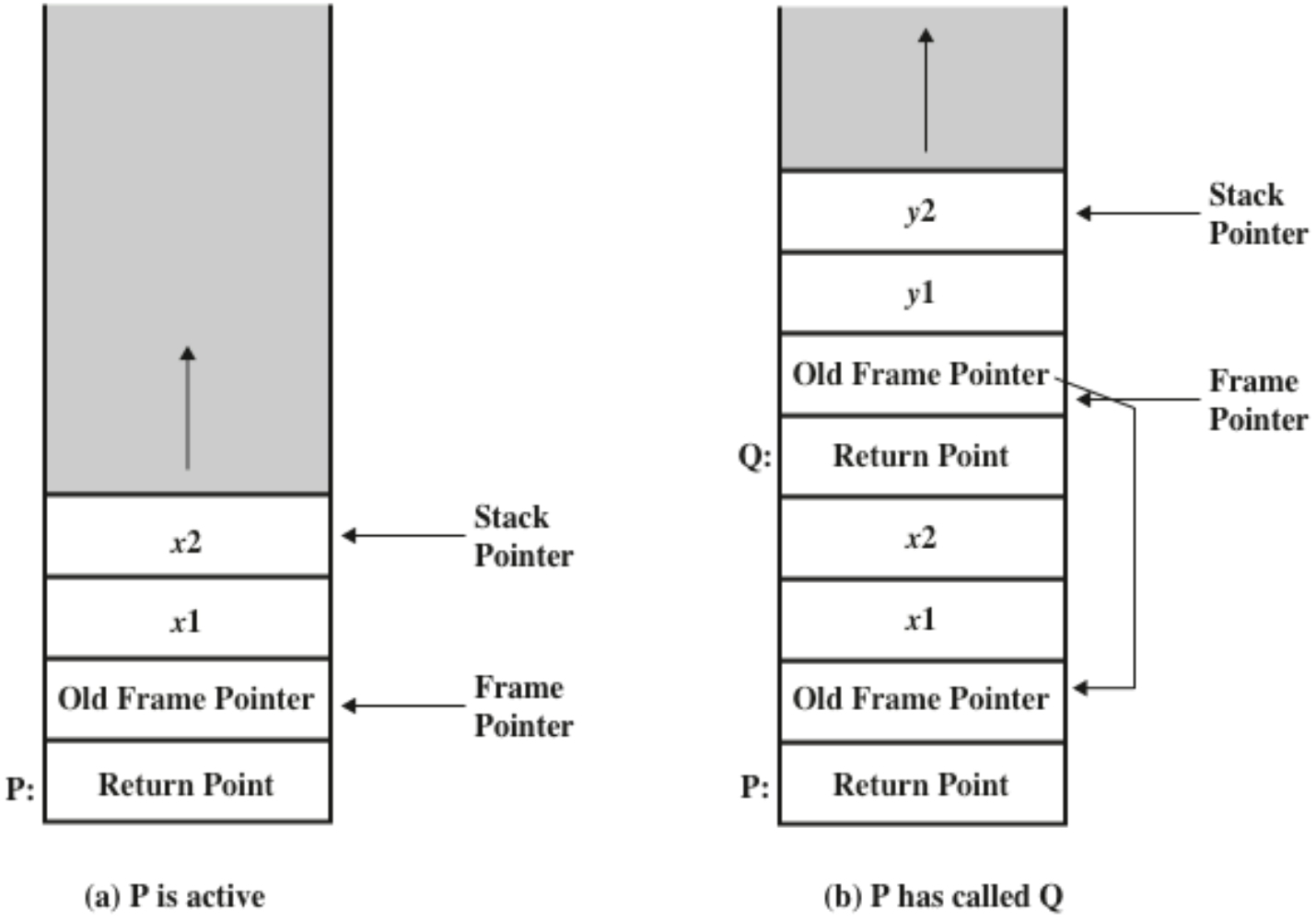


Figure 1.10 Stack Frame Growth Using Sample Procedures P and Q

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.

AAiT

Table 12.8

x86
Operation Types (With Examples of Typical Operations)

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
- The ENTER instruction was added to the instruction set to provide direct support for the compiler

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.

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

x86 Condition Codes for Conditional Jump and SETcc Instructions

AAiT

x86 Single-Instruction, Multiple-Data (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

ARM Operation Types

- Load and store instructions
- Branch instructions
- Data-processing instructions
- Multiply instructions
- Parallel addition and subtraction instructions
- Extend instructions
- Status register access instructions

Code	Symbol	Condition Tested	Comment
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS/HS	C = 1	Carry set/unsigned higher or same
0011	CC/LO	C = 0	Carry clear/unsigned lower
0100	MI	N = 1	Minus/negative
0101	PL	N = 0	Plus/positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 AND Z = 0	Unsigned higher
1001	LS	C = 0 OR Z = 1	Unsigned lower or same
1010	GE	N = V [(N = 1 AND V = 1) OR (N = 0 AND V = 0)	Signed greater than or equal
1011	LT	N ≠ V [(N = 1 AND V = 0) OR (N = 0 AND V = 1)]	Signed less than
1100	GT	(Z = 0) AND (N = V)	Signed greater than
1101	LE	(Z = 1) OR (N ≠ V)	Signed less than or equal
1110	AL	—	Always (unconditional)
1111	—	—	This instruction can only be executed unconditionally

ARM Conditions for Conditional Instruction Execution

Addressing Modes

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

Addressing Modes

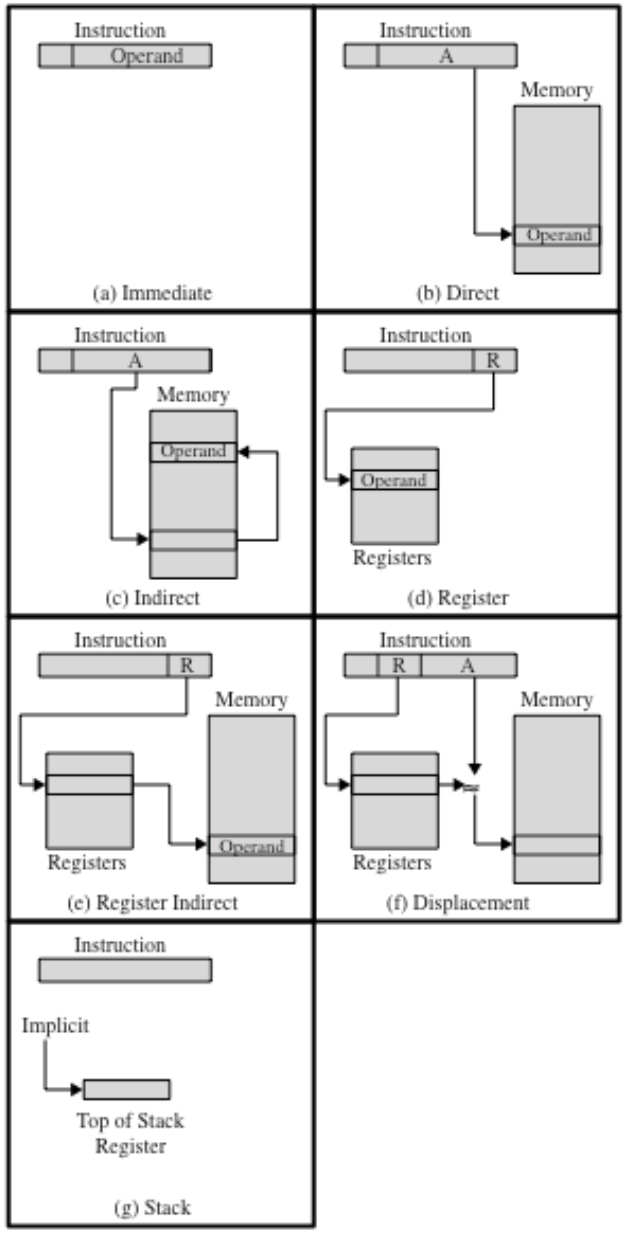


Figure 13.1 Addressing Modes

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
- This mode can be used to define and use constants or set initial values of variables
  - Typically 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

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 (also called PC-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
  - Typically the address field is treated as a twos complement number for this operation
  - Thus 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

x86 Addressing Mode Calculation

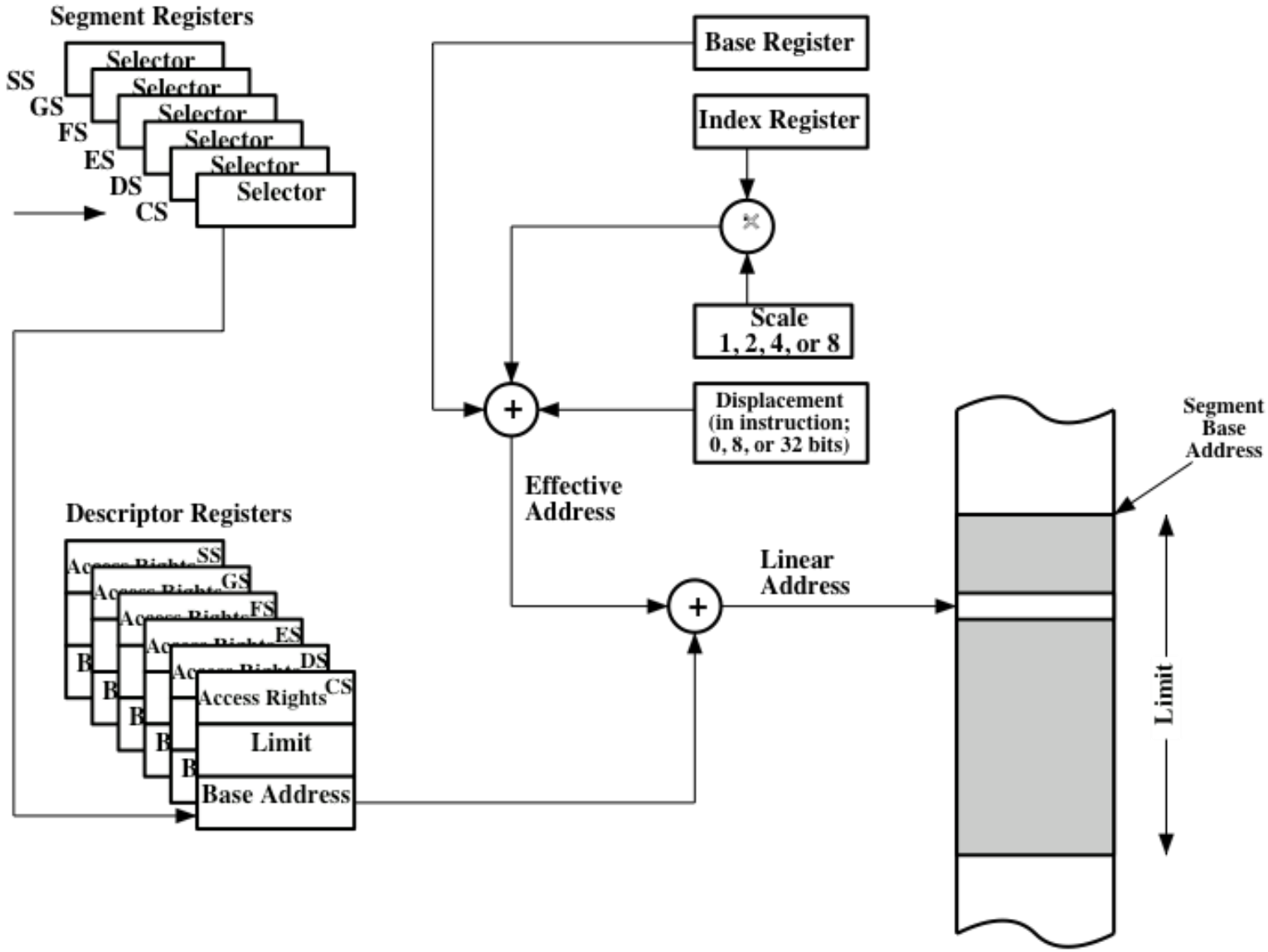


Figure 13.2 x86 Addressing Mode Calculation

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

ARM Indexing Methods

- In ARM, load and store instructions are the only instructions that reference memory.
- This is always done indirectly through a base register plus offset. There are three alternatives with respect to indexing (Figure 13.3):

- **Offset**:

  STRB r0, [r1, #12].

  An offset value is added to or subtracted from the value in the base register to form the memory address.
- **Preindex**:

  STRB r0, [r1, #12]!

  Same but the memory address is also written back to the base register. The exclamation point signifies preindexing.
- **Postindex**:

  STRB r0, [r1], #12.

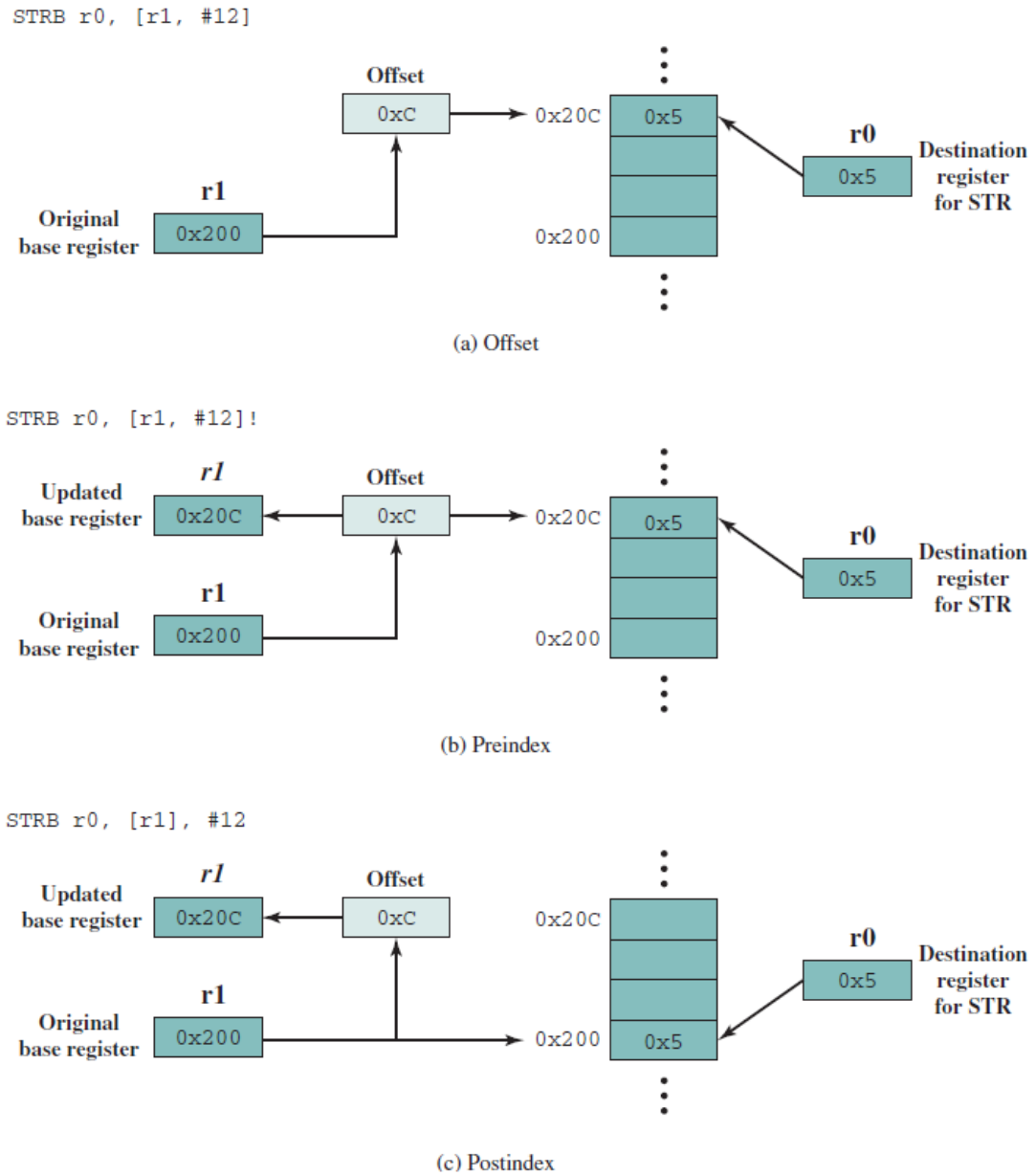


Figure 13.3 ARM Indexing Methods

ARM Data Processing Instruction Addressing and Branch Instructions

- Data processing instructions (Direct Memory addressing is not allowed, uses load /Store instructions instead)
  - Use either register addressing or a mixture of register and immediate addressing
  - For register addressing the value in one of the register operands may be scaled using one of the five shift operators

- Branch instructions
  - The only form of addressing for branch instructions is immediate
  - Instruction contains 24 bit value
    - Shifted 2 bits left so that the address is on a word boundary
    - Effective range +/-32MB from from the program counter

Instruction Formats

- Define the layout of the bits of an instruction, in terms of its constituent fields
- Must include an opcode and, implicitly or explicitly, indicate the addressing mode for each operand
- For most instruction sets more than one instruction format is used

Instruction Length

The most basic design issue to be faced is the instruction format length.

- Most basic design issue
- Affects, and is 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

We've looked at some of the factors that go into deciding the length of the instruction format. An equally difficult issue is how to allocate the bits in that format. The trade-offs here are complex.

- More **Number of opcodes** obviously reduces the number of bits available for addressing
  - **Number of addressing modes**: Some bits to explicitly indicate the addressing modes an opcode is using
  - **Number of operands:** Typical instruction formats on today's machines include two operands. Each operand address in the instruction might require its own mode indicator.
  - **Register versus memory:** With a single user-visible register (usually called the accumulator), one operand address is implicit and consumes no instruction bits. However, single-register programming is awkward and requires many instructions. A number of studies indicate that a total of 8 to 32 user-visible registers is desirable
  - **Number of register sets:** Some architectures, including that of the x86, have a collection of two or more specialized sets (such as data and displacement). One advantage of this latter approach is that, for a fixed number of registers, a functional split requires fewer bits to be used in the instruction. For example, with two sets of eight registers, only 3 bits are required to identify a register
  - **Address range:** For addresses that reference memory, the range of addresses that can be referenced is related to the number of address bits.
  - **Address granularity:** For addresses that reference memory rather than registers, another factor is the granularity of addressing. In a system with 16- or 32-bit words, an address can reference a word or a byte at the designer's choice. Byte addressing is convenient for character manipulation but requires, for a fixed-size memory, more address bits.

PDP-8 Instruction Format

- The PDP-8 uses 12-bit instructions and operates on 12-bit words. There is a single general-purpose register, the accumulator.

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

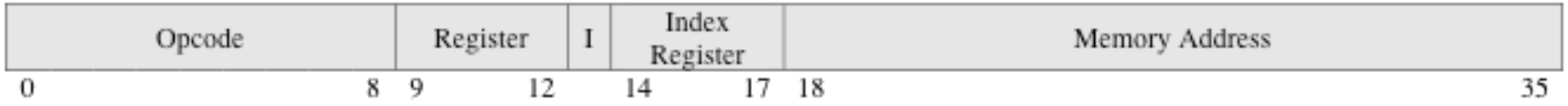
1	1	1	1	CLA	MQA	0	MQL	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11

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 11.5 PDP-8 Instruction Formats

PDP-10 Instruction Format

- The PDP-10 was designed to be a large-scale time-shared system, with an emphasis on making the system easy to program, even if additional hardware expense was involved.
- Base plus displacement addressing, which places a memory organization burden on the programmer, was avoided in favor of direct addressing.
- A 36-bit instruction length is true luxury. There is no need to do clever things to get more opcodes; a 9-bit opcode field is more than adequate.
- Addressing is also straightforward. An 18-bit address field makes direct addressing desirable. For memory sizes greater than 2^{18}, indirection is provided.



I = indirect bit

Figure 11.6 PDP-10 Instruction Format

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

PDP-11 Instruction Format

- The PDP-11 was designed to provide a powerful and flexible instruction set within the constraints of a 16-bit minicomputer .
- Figure 13.7 shows the PDP-11 instruction formats. Thirteen different formats are used, encompassing zero-, one-, and two-address instruction types
- PDP-11 instructions are usually one word (16 bits) long. For some instructions, one or two memory addresses are appended, so that 32-bit and 48-bit instructions are part of the repertoire. This provides for further flexibility in addressing.

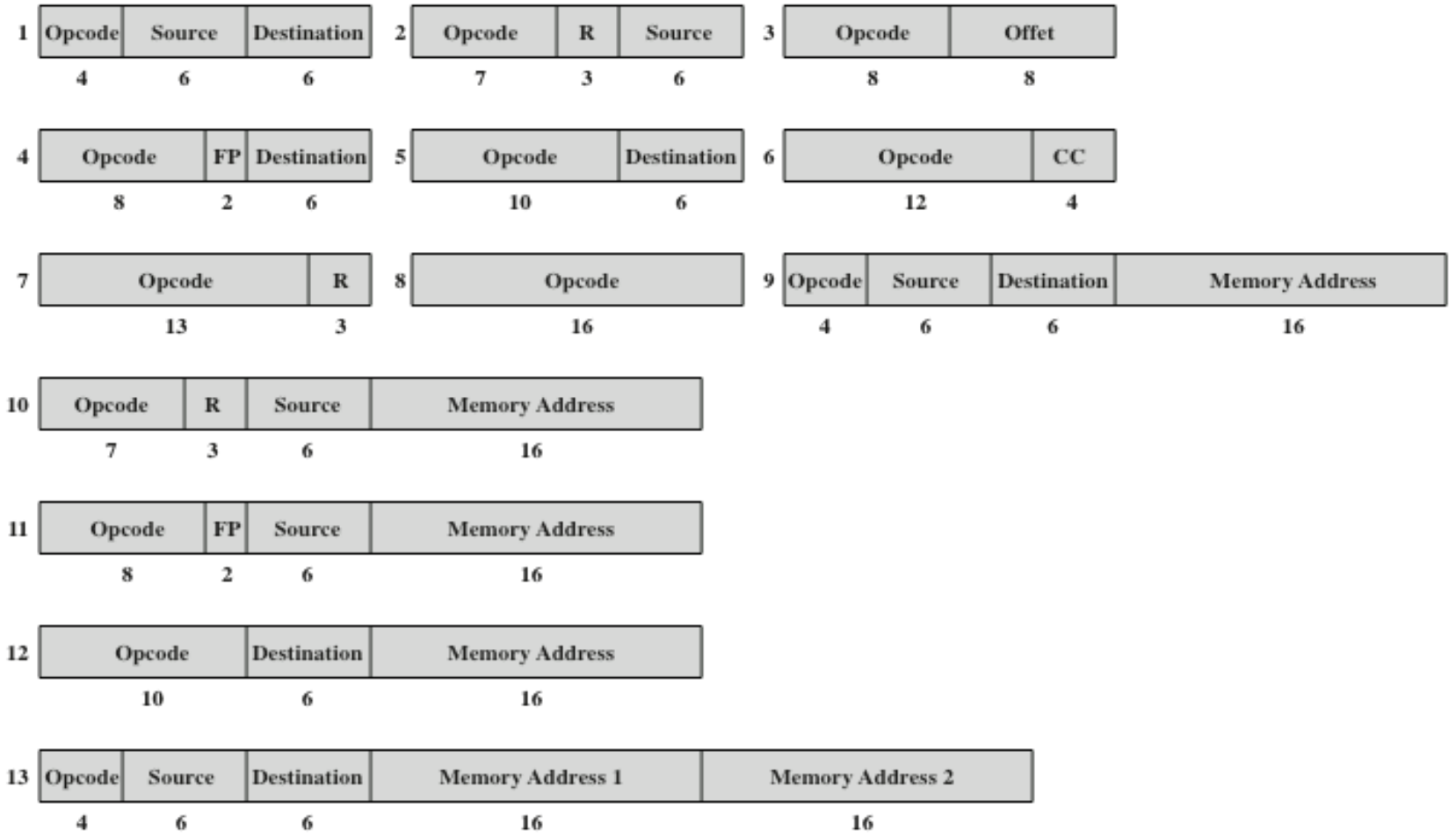


Figure 13.7 Instruction Formats for the PDP-11

VAX Instruction Examples

- VAX is a highly variable instruction format developed, like all PDPs, by DEC (Digital Equipment Corporation)
- An instruction consists of a 1- or 2-byte opcode followed by from zero to six operand specifiers, depending on the opcode. The minimal instruction length is 1 byte, and instructions up to 37 bytes can be constructed. Figure 13.8 gives a few examples.

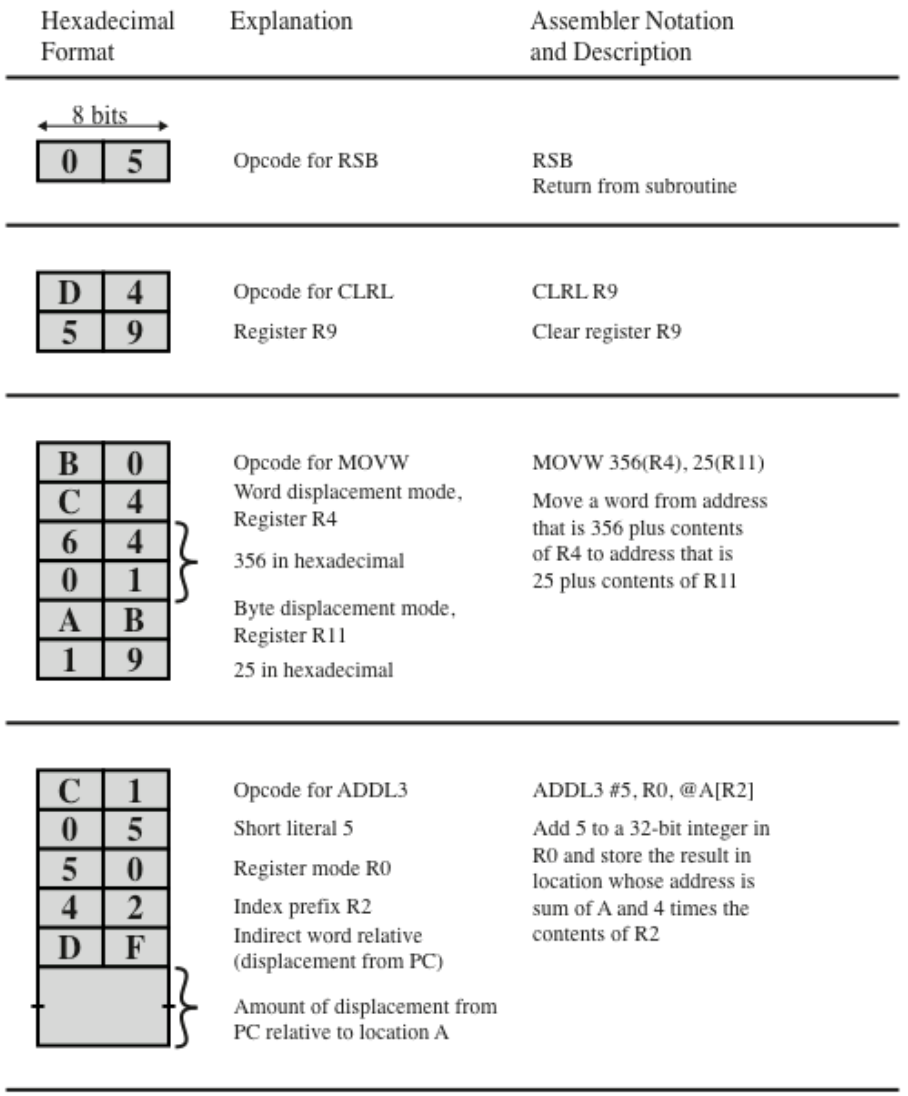


Figure 13.8 Examples of VAX Instructions

x86 Instruction Format

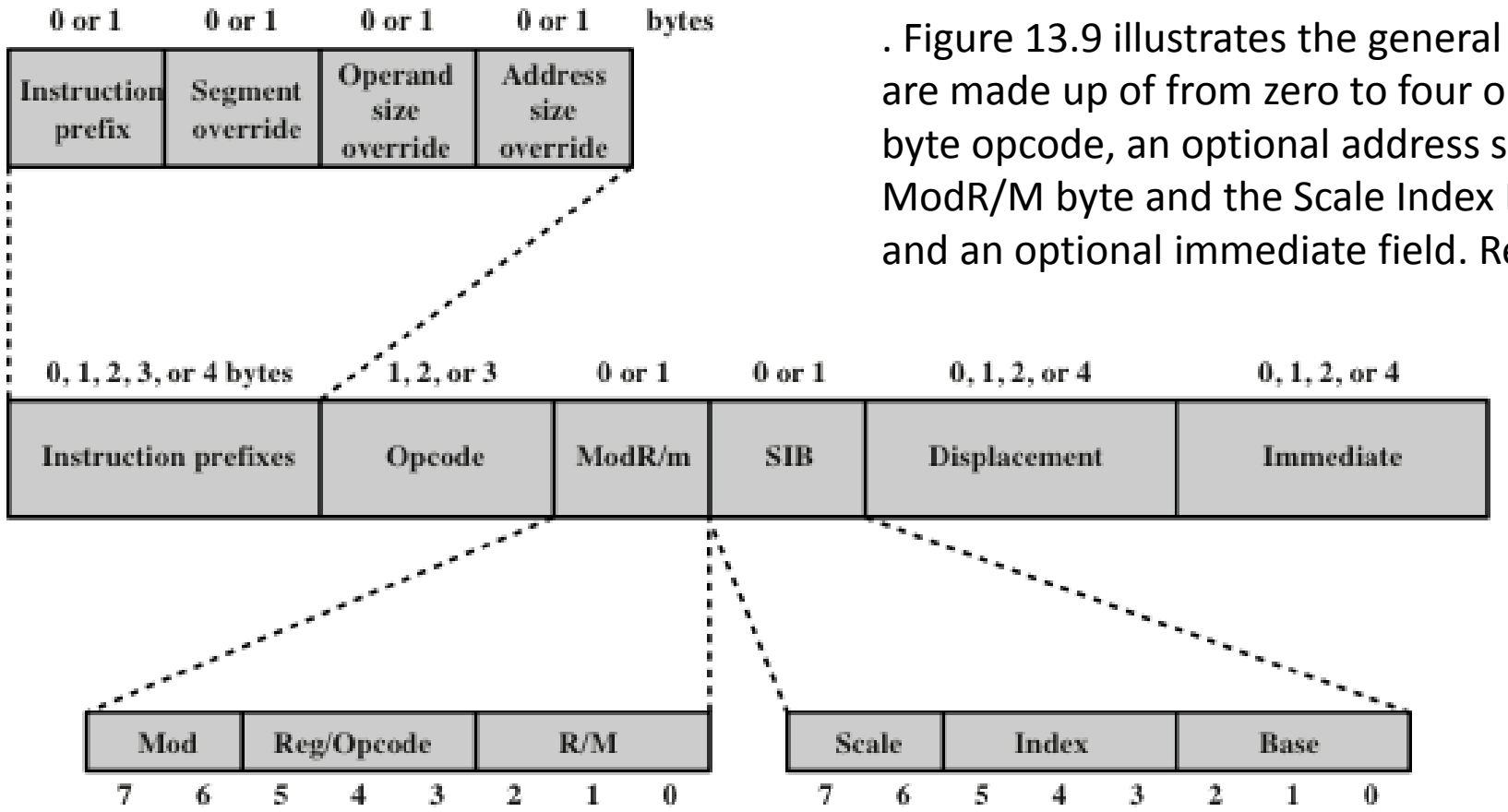


Figure 13.9 x86 Instruction Format

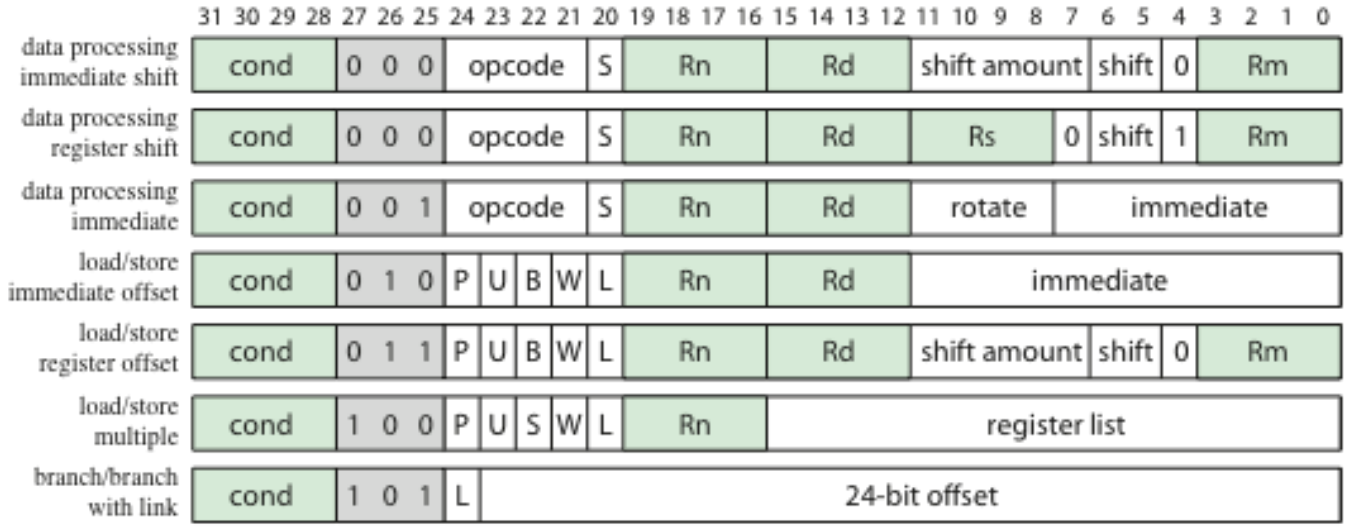
. Figure 13.9 illustrates the general x86 instruction format. Instructions are made up of from zero to four optional instruction prefixes, a 1- or 2-byte opcode, an optional address specifier (which consists of the ModR/M byte and the Scale Index Base byte) an optional displacement, and an optional immediate field. Refer here for Intel 64 and IA-32 ISA.

As can be seen, the encoding of the x86 instruction set is very complex. This has to do partly with the need to be backward compatible with the 8086 machine and partly with a desire on the part of the designers to provide every possible assistance to the compiler writer in producing efficient code. It is a matter of some debate whether an instruction set as complex as this is preferable to the opposite extreme of the RISC (as in ARM) instruction sets.

Q: Ideally, what would the longest instruction size be in x86?

ARM Instruction Formats

- All instructions are 32 bits long, unlike x86 which is variable.
- Most instructions execute in a single cycle (unlike x86 which is variable).
- Most instructions can be conditionally executed
- Refer here for full instruction set



S = For data processing instructions, signifies that the instruction updates the condition codes
S = For load/store multiple instructions, signifies whether instruction execution is restricted to supervisor mode
P, U, W = bits that distinguish among different types of addressing_mode
B = Distinguishes between an unsigned byte (B==1) and a word (B==0) access
L = For load/store instructions, distinguishes between a Load (L==1) and a Store (L==0)
L = For branch instructions, determines whether a return address is stored in the link register

Figure 13.10 ARM Instruction Formats

Thumb Instruction Set

- The Thumb instruction set contains a subset of the ARM 32-bit instruction set recoded into 16-bit instructions.
- Thumb instructions are unconditional, so the condition code field is not used. Referee here for full instruction set
- The ARM processor can execute a program consisting of a mixture of Thumb instructions and 32-bit ARM instructions.

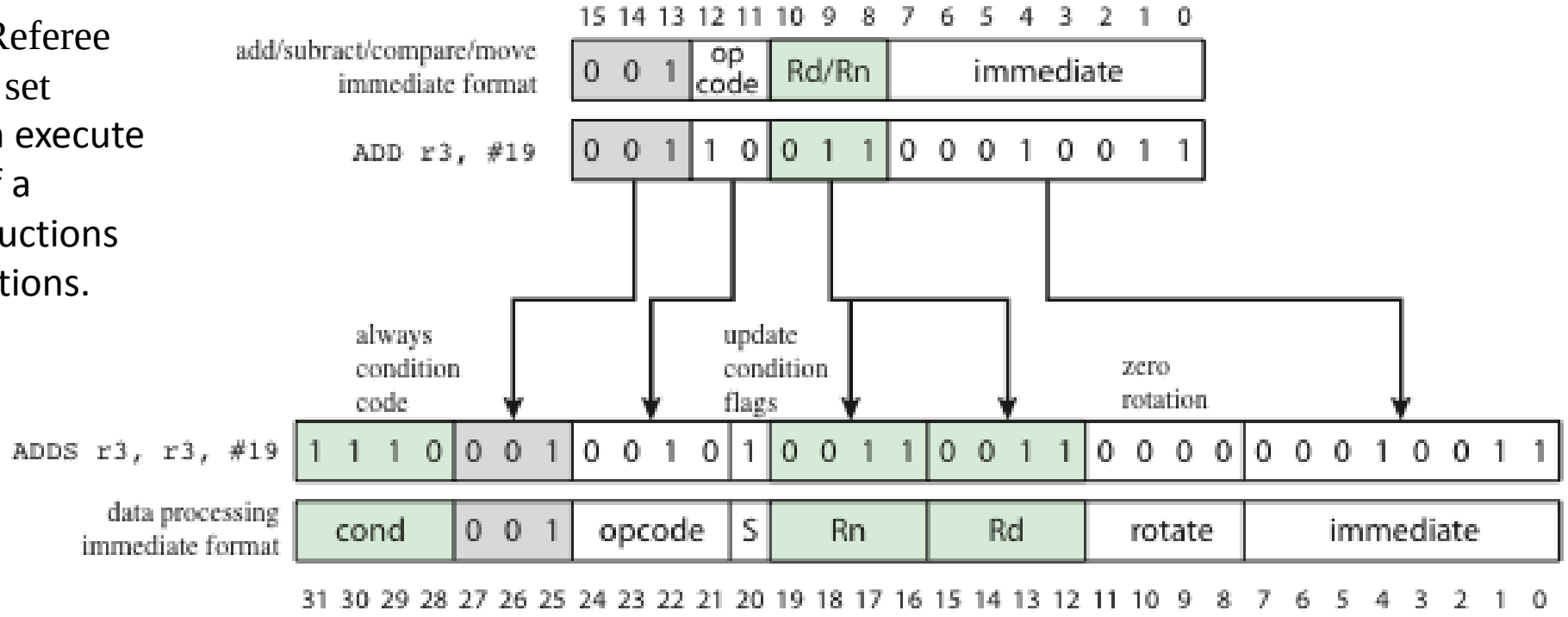


Figure 13.12 Expanding a Thumb ADD Instruction into its ARM Equivalent

Assembler

- A processor can understand and execute machine instructions. Such instructions are simply binary numbers stored in the computer.
- The development of assembly language was a major milestone in the evolution of computer technology. It was the first step to the high-level languages in use today. Although few programmers use assembly language, virtually all machines provide one.

Consider the simple BASIC statement

N=I+J+K

- Suppose we wished to program this statement in machine language and to initialize I, J, and K to 2, 3, and 4, respectively. This is shown in Figure 13.13a. The program starts in location 101 (hexadecimal). Memory is reserved for the four variables starting at location 201. The program consists of four instructions:

Address	Contents			
101	0010	0010	101	2201
102	0001	0010	102	1202
103	0001	0010	103	1203
104	0011	0010	104	3204
201	0000	0000	201	0002
202	0000	0000	202	0003
203	0000	0000	203	0004
204	0000	0000	204	0000

(a) Binary program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

(c) Symbolic program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program

Figure 11.13 Computation of the Formula N = I + J+ K

AAiT

Summary

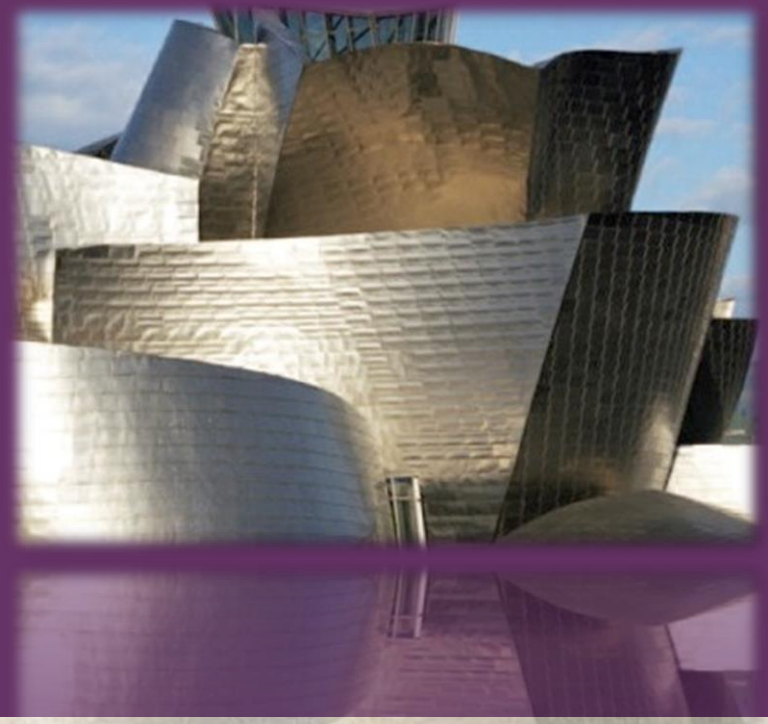
Instruction Sets:

Characteristics, Functions & Addressing mode

- 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

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

- Addressing modes
  - Immediate addressing
  - Direct addressing
  - Indirect addressing
  - Register addressing
  - Register indirect addressing
  - Displacement addressing
  - Stack addressing
- x86 & ARM addressing modes
- Instruction formats
  - Instruction length
  - Allocation of bits
  - Variable-length instructions
- X86 & ARM instruction formats



William Stallings
Computer Organization and Architecture
9th Edition