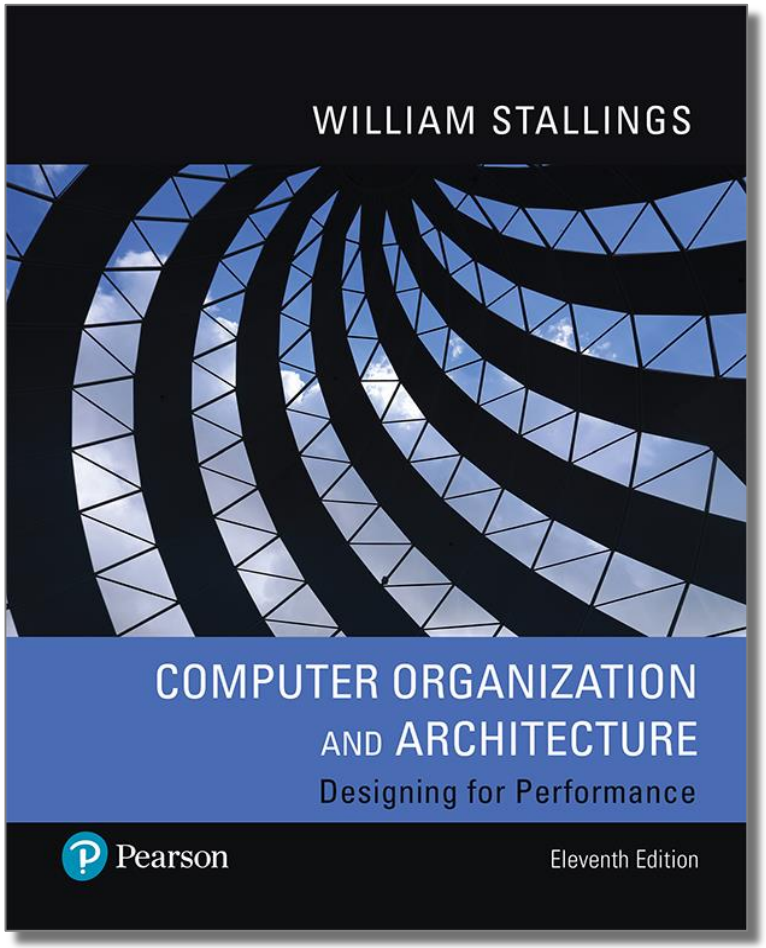
# Computer Organization and Architecture

## Designing for Performance

11th Edition




## Chapter 13

Instruction Sets: Characteristics and Functions

# Table 13.4
# Processor Actions for Various Types of Operations

| | |
|---|---|
| Data transfer | Transfer data from one location to another |
| | If memory is involved:<br>Determine memory address<br>Perform virtual-to-actual-memory address transformation<br>Check cache<br>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 |

(Table can be found on page 447 in the textbook.)

# Table 13.3 Common x86 Instruction Set Operations (1 of 3)

(a) Data Transfer

| Operation Name | Description |
|---|---|
| MOV Dest, Source | Move data between registers or between register and memory or immediate to register. |
| XCHG Op1, Op2 | Swap contents between two registers or register and memory. |
| PUSH Source | Decrements stack pointer (ESP register), then copies the source operand to the top of stack. |
| POP Dest | Copies top of stack to destination and increments ESP. |

(b) Arithmetic

| Operation Name | Description |
|---|---|
| ADD Dest, Source | Adds the destination and the source operand and stores the result in the destination. Destination can be register or memory. Source can be register, memory, or immediate. |
| SUB Dest, Source | Subtracts the source from the destination and stores the result in the destination. |
| MUL Op | Unsigned integer multiplication of the operand by the AL, AX, or EAX register and stores in the register. Opcode indicates size of register. |
| IMUL Op | Signed integer multiplication. |
| DIV Op | Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. |
| IDIV Op | Signed integer division. |
| INC Op | Adds 1 to the destination operand, while preserving the state of the CF flag. |
| DEC Op | Subtracts 1 from the destination operand, while preserving the state of the CF flag. |
| NEG Op | Replaces the value of operand with (0 – operand), using twos complement representation. |
| CMP Op1, Op2 | Compares the two operands by subtracting the second operand from the first operand and sets the status flags in the EFLAGS register according to the results. |

(Table can be found on page 446-447 in the textbook.)

# Table 13.3 Common x86 Instruction Set Operations (2 of 3)

(c) Shift and Rotate

| Operation Name | Description |
|---|---|
| SAL Op, Quantity | Shifts the source operand left by from 1 to 31 bit positions. Empty bit positions are cleared. The CF flag is loaded with the last bit shifted out of the operand. |
| SAR Op, Quantity | Shifts the source operand right by from 1 to 31 bit positions. Empty bit positions are cleared if the operand is positive and set if the operand is negative. The CF flag is loaded with the last bit shifted out of the operand. |
| SHR Op, Quantity | Shifts the source operand right by from 1 to 31 bit positions. Empty bit positions are cleared and the CF flag is loaded with the last bit shifted out of the operand. |
| ROL Op, Quantity | Rotate bits to the left, with wraparound. The CF flag is loaded with the last bit shifted out of the operand. |
| ROR Op, Quantity | Rotate bits to the right, with wraparound. The CF flag is loaded with the last bit shifted out of the operand. |
| RCL Op, Quantity | Rotate bits to the left, including the CF flag, with wraparound. This instruction treats the CF flag as a one-bit extension on the upper end of the operand. |
| RCR Op, Quantity | Rotate bits to the right, including the CF flag, with wraparound. This instruction treats the CF flag as a one-bit extension on the lower end of the operand. |

(d) Logical

| Operation Name | Description |
|---|---|
| NOT Op | Inverts each bit of the operand. |
| AND Dest, Source | Performs a bitwise AND operation on the destination and source operands and stores the result in the destination operand. |
| OR Dest, Source | Performs a bitwise OR operation on the destination and source operands and stores the result in the destination operand. |
| XOR Dest, Source | Performs a bitwise XOR operation on the destination and source operands and stores the result in the destination operand. |
| TEST Op1, Op2 | Performs a bitwise AND operation on the two operands and sets the S, Z, and P status flags. The operands are unchanged. |

(Table can be found on page 446-447 in the textbook.)

# Table 13.3 Common x86 Instruction Set Operations (3 of 3)

(e) Transfer of Control

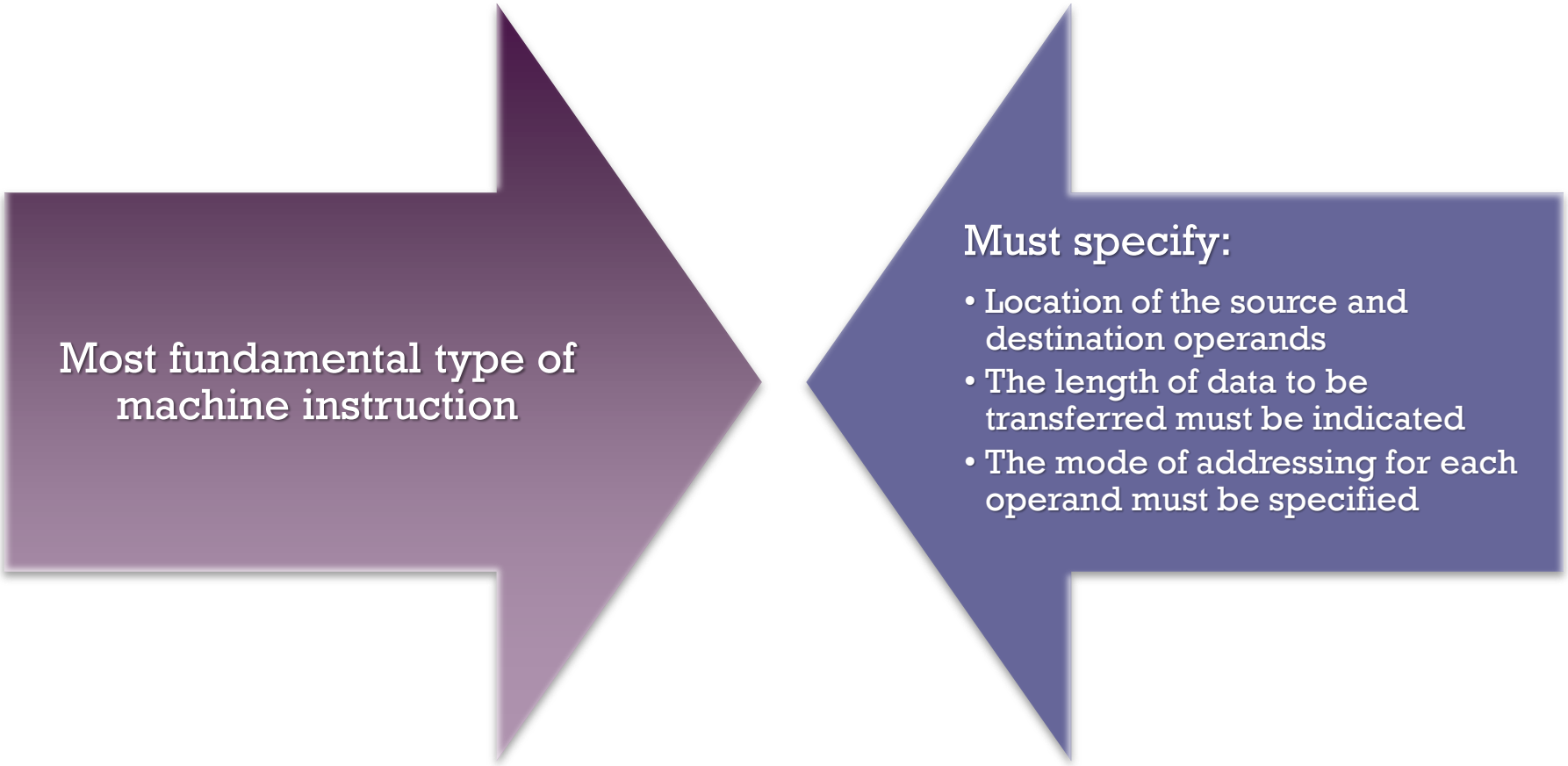
| Operation Name | Description |
|---|---|
| CALL proc | Saves procedure linking information on the stack and branches to the called procedure specified using the operand. The operand specifies the address of the first instruction in the called procedure. |
| RET | Transfers program control to a return address located on the top of the stack. The return is made to the instruction that follows the CALL instruction. |
| JMP Dest | Transfers program control to a different point in the instruction stream without recording return information. The operand specifies the address of the instruction being jumped to. |
| Jcc Dest | Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. See Tables 13.8 and 13.9. |
| NOP | This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register. |
| HLT | Stops instruction execution and places the processor in a HALT state. An enabled interrupt, a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. |
| WAIT | Causes the processor to repeatedly check for and handle pending, unmasked, floating-point exceptions before proceeding. |
| INT Nr | Interrupts current program, runs specified interrupt program |

(f) Input/Output

| Operation Name | Description |
|---|---|
| IN Dest, Source | Copies the data from the I/O port specified by the source operand to the destination operand, which is a register location. |
| INS Dest, Source | Copies the data from the I/O port specified by the source operand to the destination operand, which is a memory location. |
| OUT Dest, Source | Copies the byte, word, or doubleword value from the source register to the I/O port specified by the destination operand. |
| XOR Dest, Source | Copies byte, word, or doubleword from the source operand to the I/O port specified with the destination operand. The source operand is a memory location. |

(Table can be found on page 446-447 in the textbook.)

# Data Transfer



# Table 13.5
# 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 |

(Table can be found on page 448 in the textbook.)

# Data Transfer

- If both source and destination are registers, then the processor simply 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:
  - Calculate the memory address, based on the address mode (discussed in Chapter 14).
  - If the address refers to virtual memory, translate from virtual to real memory address.
  - Determine whether the addressed item is in cache.
  - If not, issue a command to the memory module.

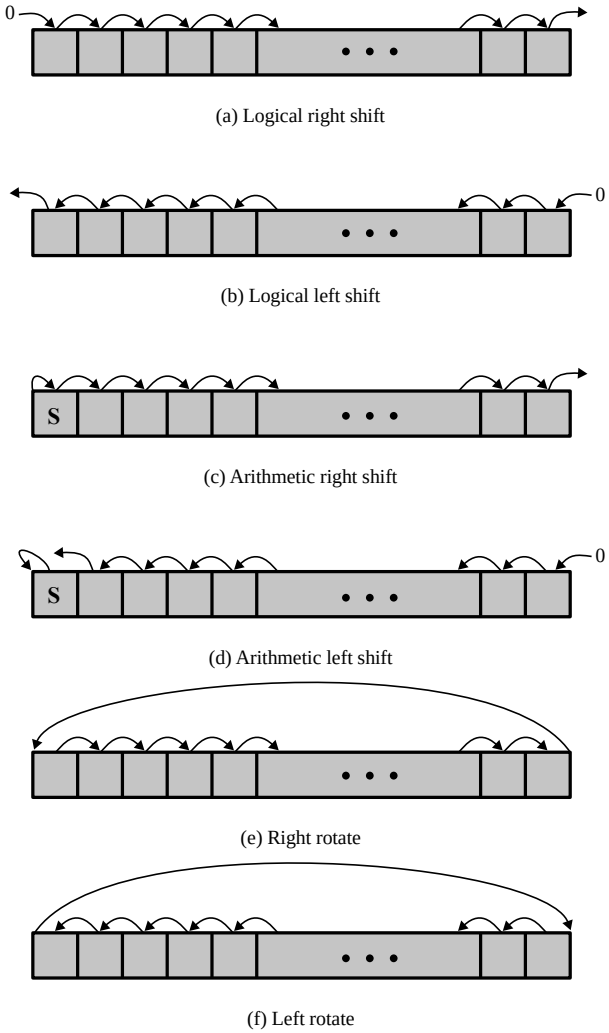
# Arithmetic

- 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

# Table 13.6
# 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 |

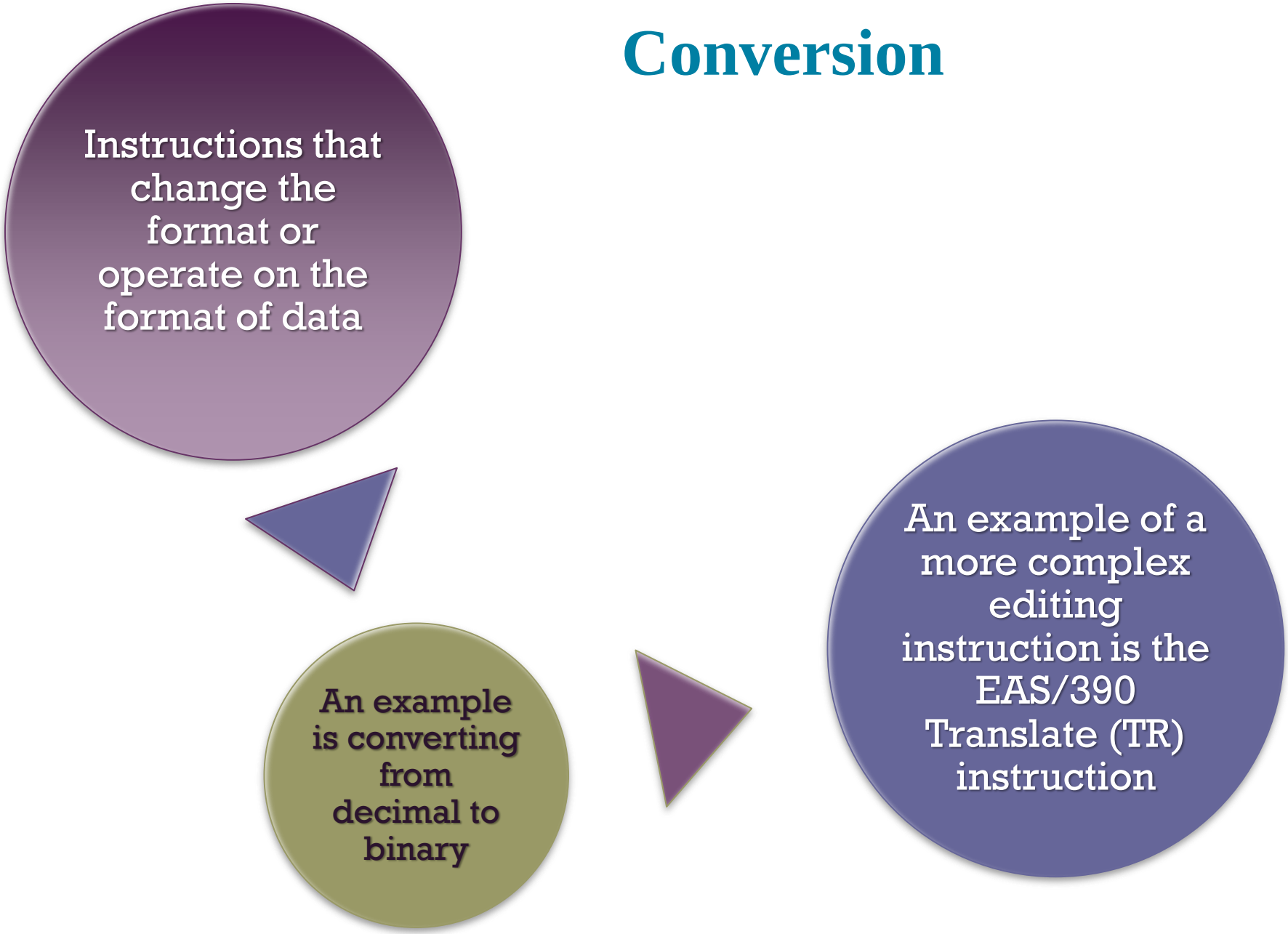
# Figure 13.6
# Shift and Rotate Operations



# Table 13.7
# 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



# 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

Typically these instructions are reserved for the use of the operating system

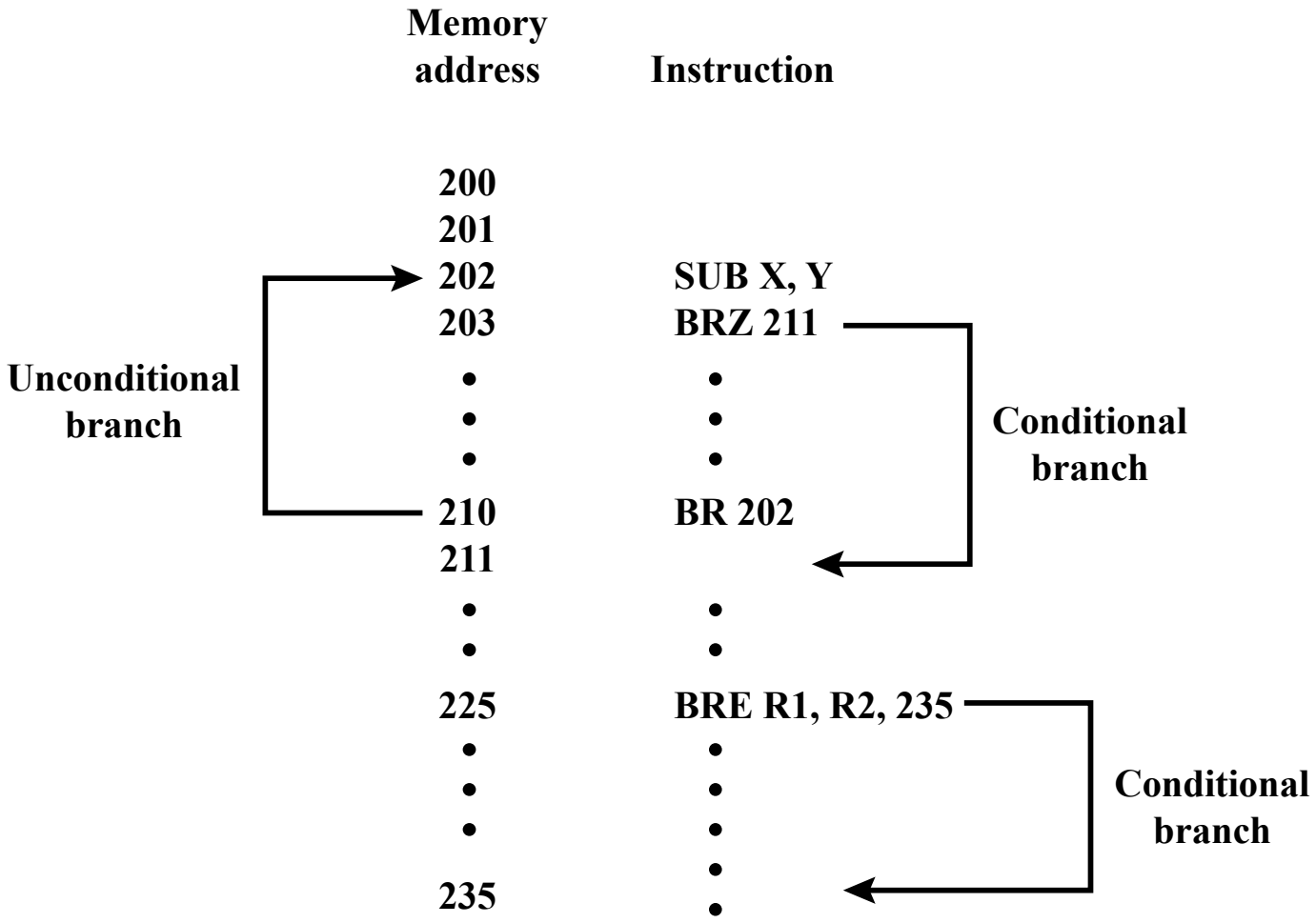
Examples of system control operations:

- 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

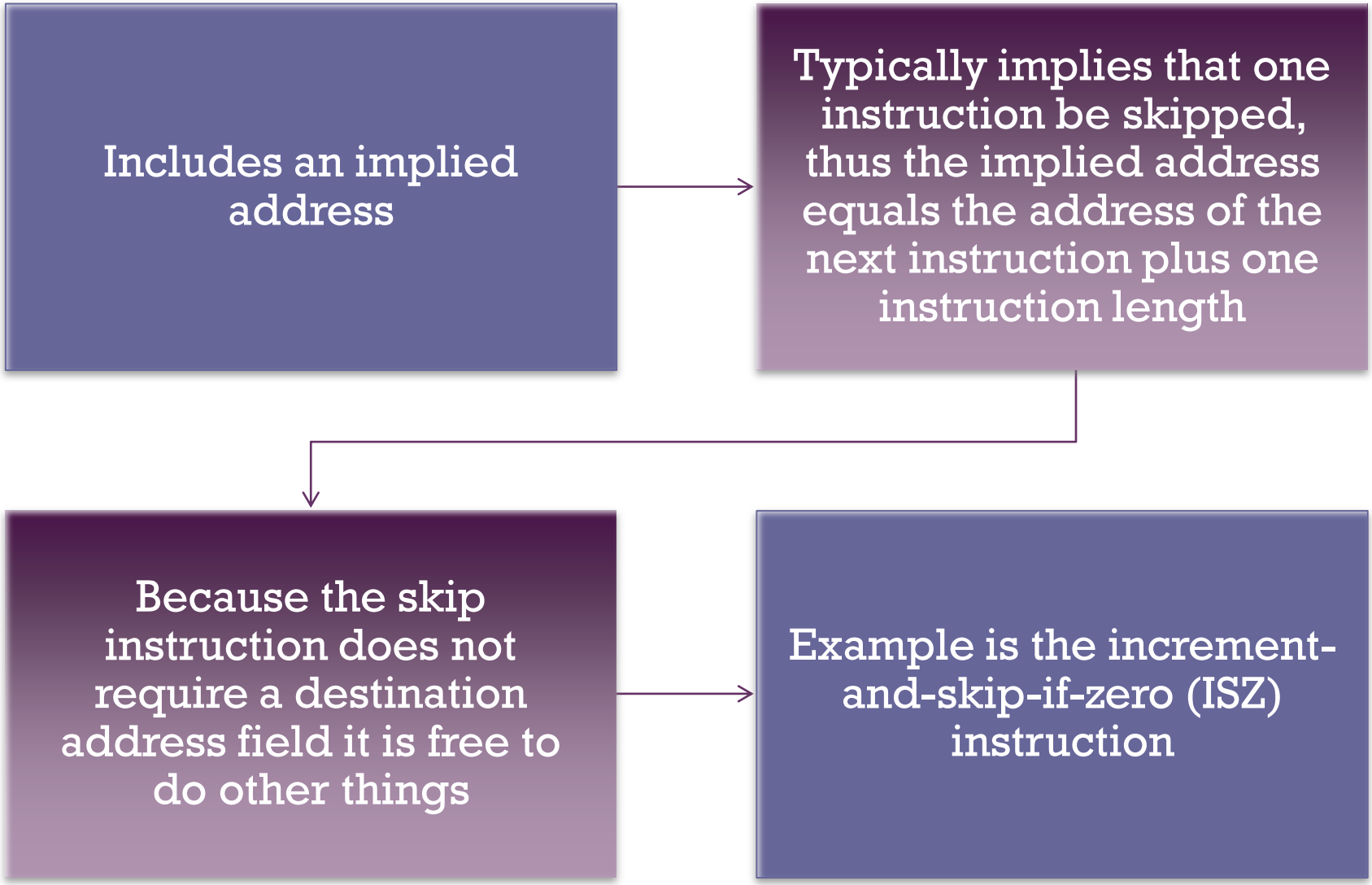
# Transfer of Control

- 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

# Figure 13.7
# Branch Instructions



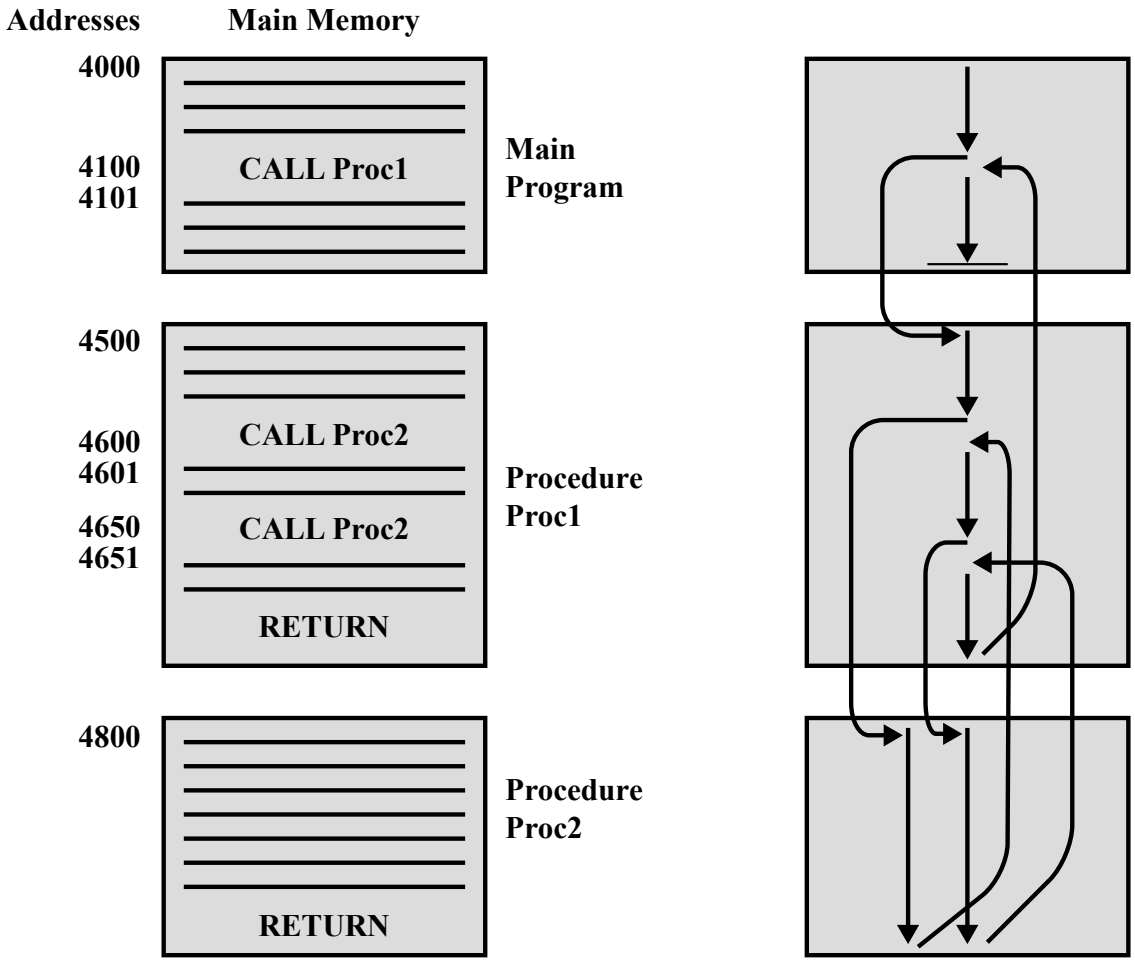
# Skip Instructions



# 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

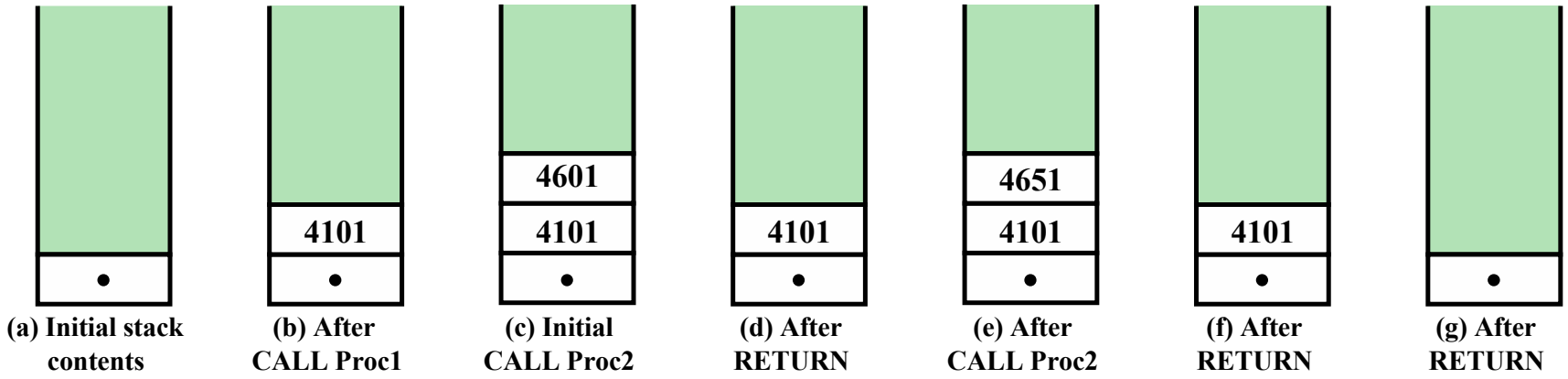
# Figure 13.8
# Nested Procedures



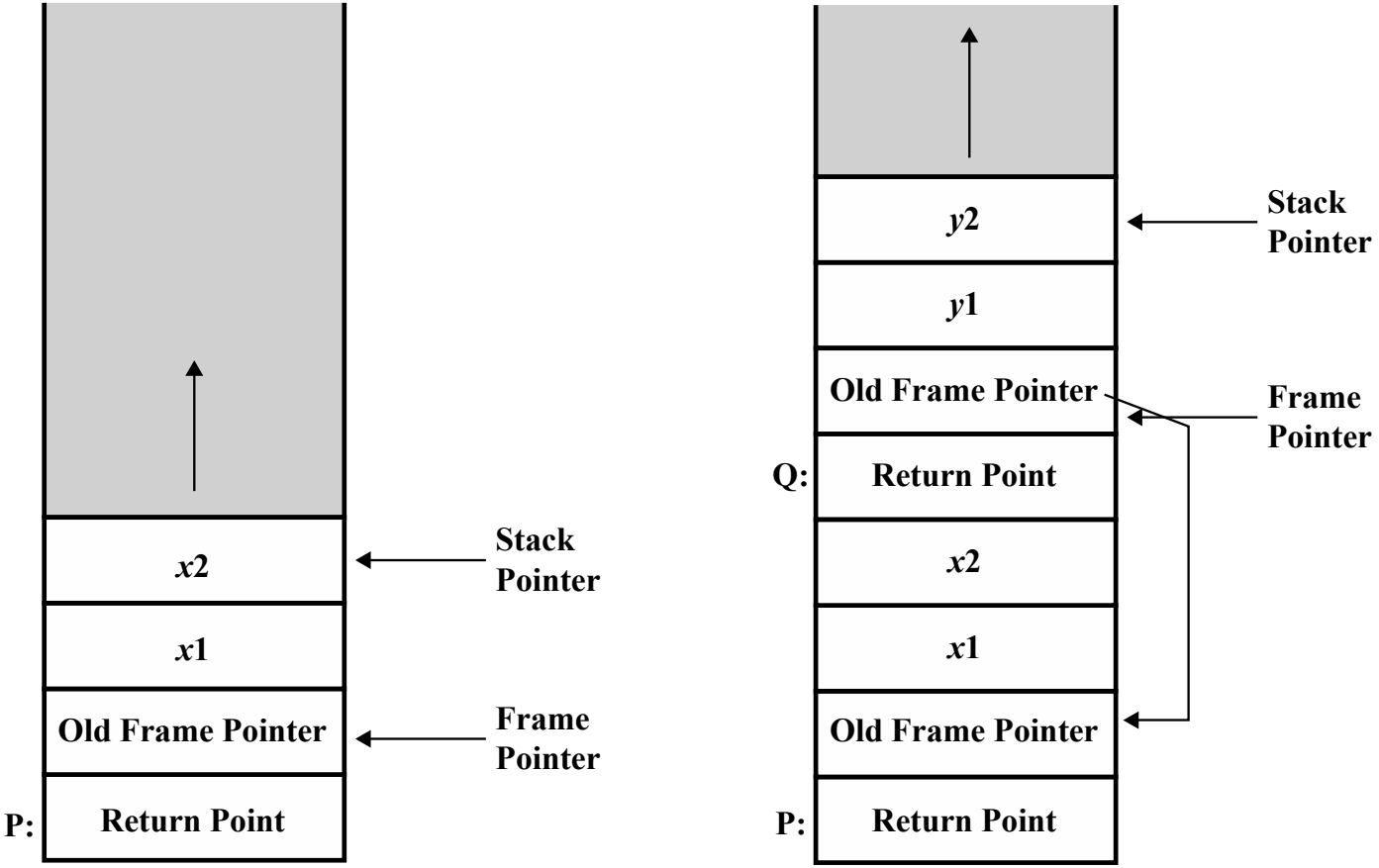

(a) Calls and returns

(b) Execution sequence

# Figure 13.9
# Use of Stack to Implement Nested Subroutines of Figure 13.8



# Figure 13.10
# Stack Frame Growth Using Sample Procedures P and Q




(a) P is active

(b) P has called Q

# x86 Operation Types

- The x86 provides a complex array of operation types including a number of specialized instructions
- The intent was to 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 13.8
# x86 Status Flags

| Status Bit | Name | Description |
|---|---|---|
| C | 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. |
| P | Parity | Parity of the least-significant byte of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity. |
| A | Auxiliary Carry | Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation. Used in binary-coded decimal arithmetic. |
| Z | Zero | Indicates that the result of an arithmetic or logic operation is 0. |
| S | Sign | Indicates the sign of the result of an arithmetic or logic operation. |
| O | Overflow | Indicates an arithmetic overflow after an addition or subtraction for twos complement arithmetic. |

# Table 13.9 x86 Condition Codes for Conditional Jump and SETcc Instructions

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

(Table can be found on page 460 in the textbook.)

# 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

# Table 13.10 MMX Instruction Set

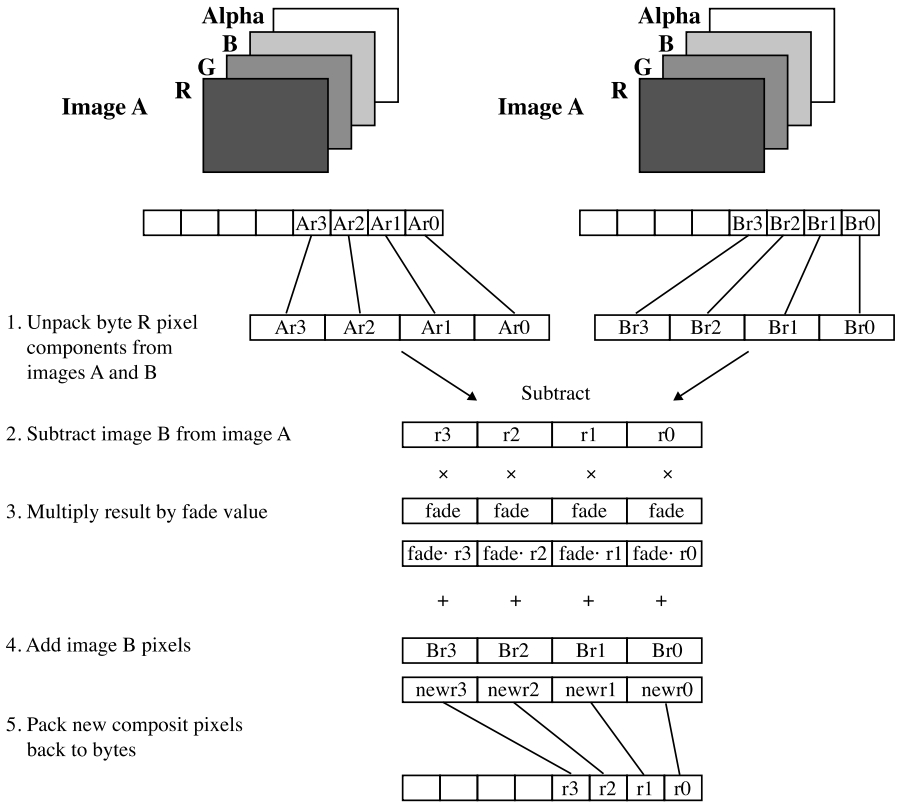
| Category | Instruction | Description |
|---|---|---|
| Arithmetic | PADD [B, W, D] | Parallel add of packed eight bytes, four 16-bit words, or two 32-bit doublewords, with wraparound. |
| | PADDS [B, W] | Add with saturation. |
| | PADDUS [B, W] | Add unsigned with saturation. |
| | PSUB [B, W, D] | 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. |
| Statemgt | EMMS | Empty MMX state (empty FP registers tag bits). |

*Note:* If an instruction supports multiple data types [byte (B), word (W), doubleword (D), quadword (Q)], the data types are indicated in brackets.

(Table can be found on page 462 in the textbook.)

Pearson

# Figure 13.11 Image Compositing on Color Plane Representation




MMX code sequence performing this operation:

```
pxor      mm7, mm7      ;zero out mm7
movq      mm3, fad_val  ;load fade value replicated 4 times
movd      mm0, imageA   ;load 4 red pixel components from image A
movd      mm1, imageB   ;load 4 red pixel components from image B
punpckblw mm0, mm7      ;unpack 4 pixels to 16 bits
punpckblw mm1, mm7      ;unpack 4 pixels to 16 bits
psubw     mm0, mm1      ;subtract image B from image A
pmulhw    mm0, mm3      ;multiply the subtract result by fade values
padddw    mm0, mm1      ;add result to image B
packuswb  mm0, mm7      ;pack 16-bit results back to bytes
```

**Figure 13.11 Image Compositing on Color Plane Representation**

# Copyright