COM/BLM 376 Computer Architecture

Chapter 12 Instruction Sets: Characteristics and Functions

Asst. Prof. Dr. Gazi Erkan BOSTANCI

ebostanci@ankara.edu.tr

Slides are mainly based on

Computer Organization and Architecture: Designing for Performance by William Stallings, 9th Edition, Prentice Hall

Outline

1. Machine Intruction Characteristics
2. Types of Operands
3. Types of Operations

One boundary where the computer designer and the computer programmer can view the same machine is the machine instruction set.

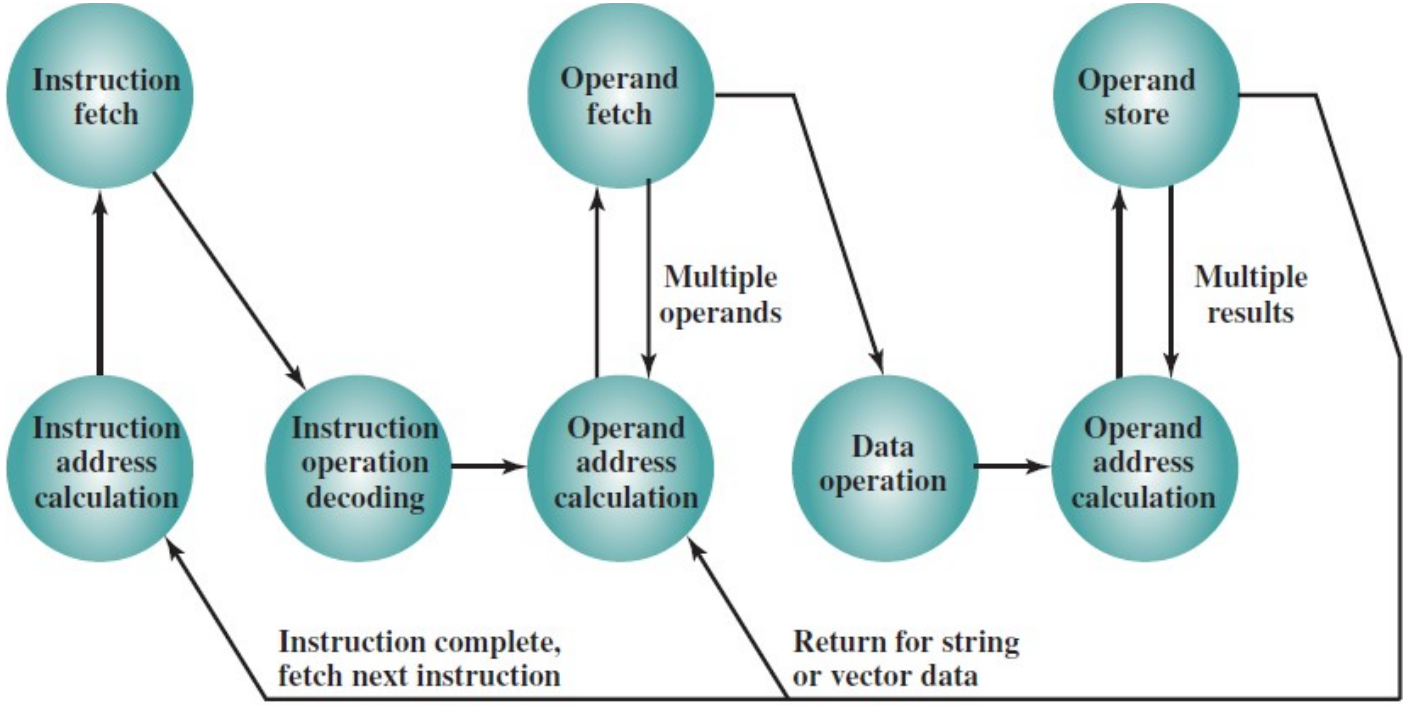
From the designer’s point of view, the machine instruction set provides the functional requirements for the processor: implementing the processor is a task that in large part involves implementing the machine instruction set.

The user who chooses to program in machine language becomes aware of the register and memory structure, the types of data directly supported by the machine, and the functioning of the ALU.

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*.

Recall previous diagram.



Elements of a Machine Instruction

- Each instruction must contain the information required by the processor for execution. Figure above shows the steps involved in instruction execution and, by implication, defines the elements of a machine instruction. These elements are as follows:
  - **Operation code:** Specifies the operation to be performed (e.g., ADD, I/O).
  - 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.

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

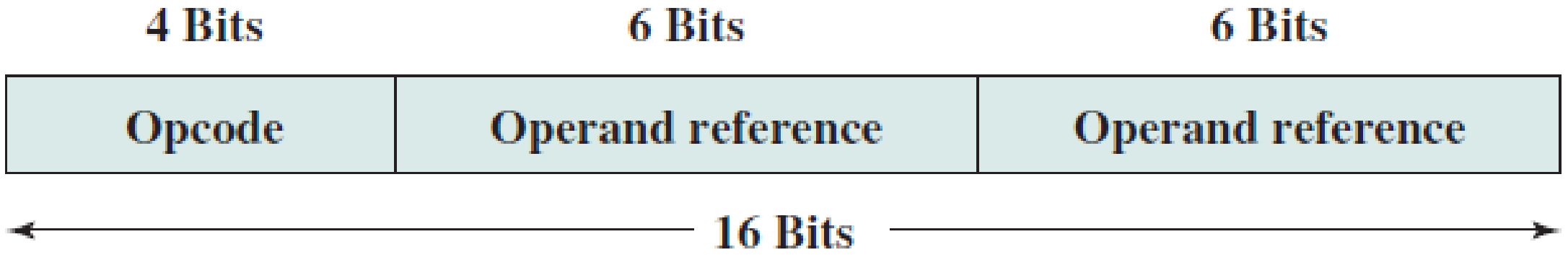
In most cases, the next instruction to be fetched immediately follows the current instruction. In those cases, there is no explicit reference to the next instruction. When an explicit reference is needed, then the main memory or virtual memory address must be supplied.

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

- **Main or virtual memory:** As with next instruction references, the main or virtual memory address must be supplied.
- **Processor register:** With rare exceptions, a processor contains one or more registers that may be referenced by machine instructions. If only one register exists, reference to it may be implicit. If more than one register exists, then each register is assigned a unique name or number, and the instruction must contain the number of the desired register.
- **Immediate:** The value of the operand is contained in a field in the instruction being executed.
- **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.

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. A simple example of an instruction format is shown in the figure. With most instruction sets, more than one format is used. 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.



It is difficult for both the programmer and the reader of textbooks to deal with binary representations of machine instructions. Thus, it has become common practice to use a *symbolic representation* of machine instructions.

Opcodes are represented by abbreviations, called *mnemonics*, that indicate the operation. Common examples include

```
ADD Add
SUB Subtract
MUL Multiply
DIV Divide
LOAD Load data from memory
STOR Store data to memory
```

Operands are also represented symbolically. For example, the instruction

```
ADD R, Y
```

may mean add the value contained in data location Y to the contents of register R.

In this example, Y refers to the address of a location in memory, and R refers to a particular register. Note that the operation is performed on the contents of a location, not on its address.

Thus, it is possible to write a machine-language program in symbolic form.

Each symbolic opcode has a fixed binary representation, and the programmer specifies the location of each symbolic operand. For example, the programmer might begin with a list of definitions:

```
X = 513
Y = 514
```

and so on. A simple program would accept this symbolic input, convert opcodes and operand references to binary form, and construct binary machine instructions.

Machine-language programmers are rare to the point of nonexistence. Most programs today are written in a high-level language or, failing that, assembly language.

However, symbolic machine language remains a useful tool for describing machine instructions, and we will use it for that purpose.

Instruction Types

Consider a high-level language instruction that could be expressed in a language such as BASIC or FORTRAN. For example,

$$X = X + Y$$

This statement instructs the computer to add the value stored in Y to the value stored in X and put the result in X. How might this be accomplished with machine instructions? Let us assume that the variables X and Y correspond to locations 513 and 514. If we assume a simple set of machine instructions, this operation could be accomplished with three instructions:

1. Load a register with the contents of memory location 513.
2. Add the contents of memory location 514 to the register.
3. Store the contents of the register in memory location 513.

As can be seen, the single BASIC instruction may require three machine instructions. This is typical of the relationship between a high-level language and a machine language. A high-level language expresses operations in a concise algebraic form, using variables. A machine language expresses operations in a basic form involving the movement of data to or from registers.

With this simple example to guide us, let us consider the types of instructions that must be included in a practical computer. A computer should have a set of instructions that allows the user to formulate any data processing task.

Another way to view it is to consider the capabilities of a high-level programming language. Any program written in a high-level language must be translated into machine language to be executed. Thus, the set of machine instructions must be sufficient to express any of the instructions from a high-level language.

With this in mind we can categorize instruction types as follows:

- **Data processing:** Arithmetic and logic instructions
- **Data storage:** Movement of data into or out of register and or memory locations
- **Data movement:** I/O instructions
- **Control:** Test and branch instructions

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. These operations are performed primarily on data in processor registers.

Therefore, there must be *memory* instructions for moving data between memory and the registers.

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

Test instructions are used to test the value of a data word or the status of a computation.

Branch instructions are then used to branch to a different set of instructions depending on the decision made.

Number of Addresses

One of the traditional ways of describing processor architecture is in terms of the number of addresses contained in each instruction. This dimension has become less significant with the increasing complexity of processor design. Nevertheless, it is useful at this point to draw and analyze this distinction.

What is the maximum number of addresses one might need in an instruction? Evidently, arithmetic and logic instructions will require the most operands. Virtually all arithmetic and logic operations are either unary (one source operand) or binary (two source operands).

Thus, we would need a maximum of two addresses to reference source operands. The result of an operation must be stored, suggesting a third address, which defines a destination operand. Finally, after completion of an instruction, the next instruction must be fetched, and its address is needed.

This line of reasoning suggests that an instruction could plausibly be required to contain four address references: two source operands, one destination operand, and the address of the next instruction. In most architectures, most instructions have one, two, or three operand addresses, with the address of the next instruction being implicit (obtained from the program counter).

Most architectures also have a few special-purpose instructions with more operands.

- For example, the load and store multiple instructions of the ARM architecture designate up to 17 register operands in a single instruction.

Figure compares typical one-, two-, and three-address instructions that could be used to compute

Y = (A - B)/[C + (D x E)].

With three addresses, each instruction specifies two source operand locations and a destination operand location.

Because we choose not to alter the value of any of the operand locations, a temporary location, T, is used to store some intermediate results. Note that there are four instructions and that the original expression had five operands.

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

Three-address instruction formats are not common because they require a relatively long instruction format to hold the three address references.

With two-address instructions, and for binary operations, one address must do double duty as both an operand and a result.

Thus, the instruction SUB Y, B carries out the calculation Y - B and stores the result in Y. The two-address format reduces the space requirement but also introduces some awkwardness.

To avoid altering the value of an operand, a MOVE instruction is used to move one of the values to a result or temporary location before performing the operation. Our sample program expands to six instructions.

Simpler yet is the one-address instruction. For this to work, 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. In our example, eight instructions are needed to accomplish the task.

It is, in fact, possible to make do with zero addresses for some instructions. Zero-address instructions are applicable to a special memory organization called a *stack*. A stack is a last-in-first-out set of locations. The stack is in a known location and, often, at least the top two elements are in processor registers. Thus, zero-address instructions would reference the top two stack elements.

Table summarizes the interpretations to be placed on instructions with zero, one, two, or three addresses. In each case in the table, it is assumed that the address of the next instruction is implicit, and that one operation with two source operands and one result operand is to be performed.

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.
- Also, there is an important threshold between one-address and multiple-address instructions. With one-address instructions, the programmer generally has available only one general-purpose register, the accumulator.
- With multiple-address instructions, it is common to have multiple general-purpose registers. This allows some operations to be performed solely on registers. Because register references are faster than memory references, this speeds up execution.

For reasons of flexibility and ability to use multiple registers, most contemporary machines employ a mixture of two- and three-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.

Also, as we shall see next week, a machine may offer a variety of addressing modes, and the specification of mode takes one or more bits. 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. The design of an instruction set is very complex because it affects so many aspects of the computer system. The instruction set defines many of the functions performed by the processor and thus has a significant effect on the implementation of the processor. The instruction set is the programmer's means of controlling the processor. Thus, programmer requirements must be considered in designing the instruction set.

It may surprise you to know that some of the most fundamental issues relating to the design of instruction sets remain in dispute. Indeed, in recent years, the level of disagreement concerning these fundamentals has actually grown.

- The most important of these fundamental design issues include the following:
  - **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, and so on
  - **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

These issues are highly interrelated and must be considered together in designing an instruction set.

Following this overview section, this chapter examines data types and operation repertoire.

TYPES OF OPERANDS

Machine instructions operate on data. The most important general categories of data are

- Addresses
- Numbers
- Characters
- Logical data

We shall see, in discussing addressing modes later, that addresses are, in fact, a form of data. 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. Even in nonnumeric data processing, there is a need for numbers to act as counters, field widths, and so forth. An important distinction between numbers used in ordinary mathematics and numbers stored in a computer is that the latter are limited. This is true in two senses.

First, there is a limit to the magnitude of numbers representable on a machine and second, in the case of floating-point numbers, a limit to their precision. Thus, the programmer is faced with understanding the consequences of rounding, overflow, and underflow.

Three types of numerical data are common in computers:

- Binary integer or binary fixed point
- Binary floating point
- Decimal

Characters

A common form of data is text or character strings. While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems. Such systems are designed for binary data.

Thus, a number of codes have been devised by which characters are represented by a sequence of bits. Perhaps the earliest common example of this is the Morse code. Today, the most commonly used character code in the International Reference Alphabet (IRA),

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.

There are two advantages to the bit-oriented view.

- First, we may sometimes wish to store an array of Boolean or binary data items, in which each item can take on only the values 1 (true) and 0 (false). With logical data, memory can be used most efficiently for this storage.
- Second, there are occasions when we wish to manipulate the bits of a data item.
  - For example, if floating-point operations are implemented in software, we need to be able to shift significant bits in some operations.
  - Another example: To convert from IRA to packed decimal, we need to extract the rightmost 4 bits of each byte.

TYPES OF OPERATIONS

The number of different opcodes varies widely from machine to machine. However, the same general types of operations are found on all machines. A useful and typical categorization is the following:

- Data transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System control
- Transfer of control

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

Type	Operation Name	Description
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

Type	Operation Name	Description
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

Type	Operation Name	Description
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

Type	Operation Name	Description
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

The most fundamental type of machine instruction is the data transfer instruction. The data transfer instruction must specify several things.

- First, the location of the source and destination operands must be specified. Each location could be memory, a register, or the top of the stack.
- Second, the length of data to be transferred must be indicated.
- Third, as with all instructions with operands, the mode of addressing for each operand must be specified.

The choice of data transfer instructions to include in an instruction set exemplifies the kinds of trade-offs the designer must make. For example, the general location (memory or register) of an operand can be indicated in either the specification of the opcode or the operand.

Figure shows 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

In terms of processor action, data transfer operations are perhaps the simplest type.

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:

1. Calculate the memory address, based on the address mode.
2. If the address refers to virtual memory, translate from virtual to real memory address.
3. Determine whether the addressed item is in cache.
4. If not, issue a command to the memory module.

Arithmetic

Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. These are invariably 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; for example,

- **Absolute:** Take the absolute value of the operand.
- **Negate:** Negate the operand.
- **Increment:** Add 1 to the operand.
- **Decrement:** Subtract 1 from the operand.

The execution of an arithmetic instruction may involve data transfer operations to position operands for input to the ALU, and to deliver the output of the ALU.

Logical

Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable units, often referred to as “bit twiddling.” They are based upon Boolean operations.

Some of the basic logical operations that can be performed on Boolean or binary data are shown in the table.

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

The NOT operation inverts a bit. AND, OR, and Exclusive-OR (XOR) are the most common logical functions with two operands. EQUAL is a useful binary test. These logical operations can be applied bitwise to *n*-bit logical data units. Thus, if two registers contain the data

(R1) = 10100101

(R2) = 00001111

then

(R1) AND (R2) = 00000101

where the notation (X) means the contents of location X. Thus, the AND operation can be used as a *mask* that selects certain bits in a word and zeros out the remaining bits.

As another example, if two registers contain

(R1) = 10100101

(R2) = 11111111

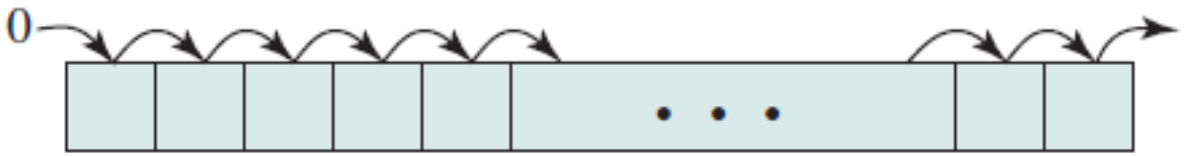
then

(R1) XOR (R2) = 01011010

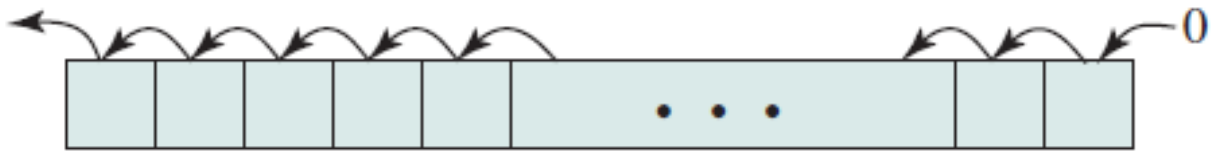
With one word set to all 1s, the XOR operation inverts all of the bits in the other word (one’s complement).

In addition to bitwise logical operations, most machines provide a variety of shifting and rotating functions. The most basic operations are illustrated in figure below.

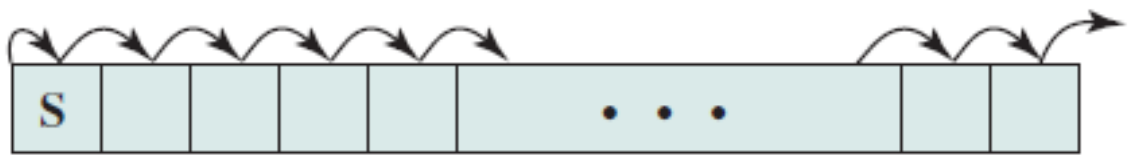
With a **logical shift**, the bits of a word are shifted left or right. On one end, the bit shifted out is lost. On the other end, a 0 is shifted in. Logical shifts are useful primarily for isolating fields within a word. The 0s that are shifted into a word displace unwanted information that is shifted off the other end.



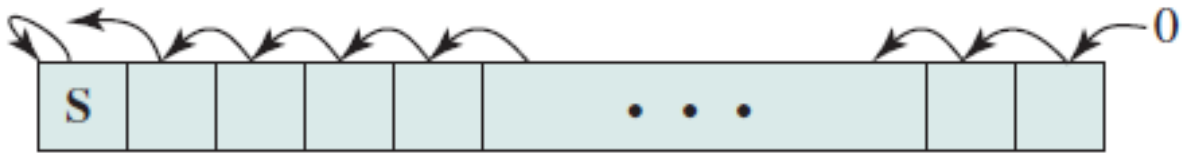
(a) Logical right shift



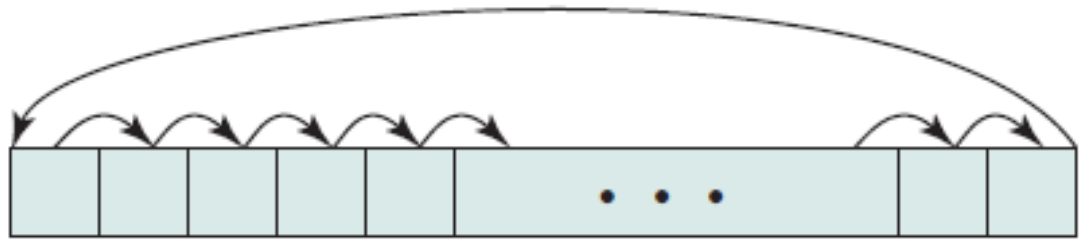
(b) Logical left shift



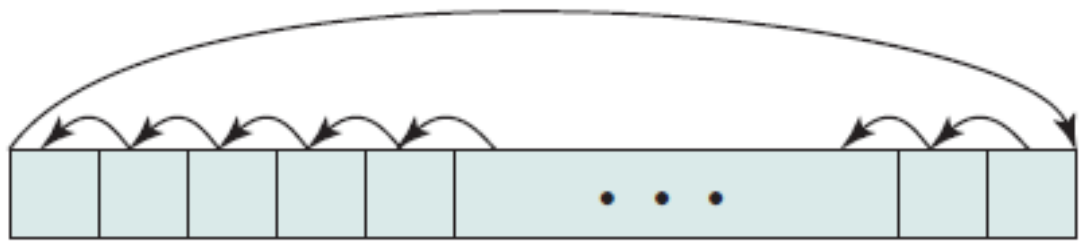
(c) Arithmetic right shift



(d) Arithmetic left shift



(e) Right rotate



(f) Left rotate

As an example, suppose we wish to transmit characters of data to an I/O device 1 character at a time. If each memory word is 16 bits in length and contains two characters, we must *unpack* the characters before they can be sent. To send the two characters in a word,

1. Load the word into a register.
2. Shift to the right eight times. This shifts the remaining character to the right half of the register.
3. Perform I/O. The I/O module reads the lower-order 8 bits from the data bus.

The preceding steps result in sending the left-hand character. To send the righthand character,

1. Load the word again into the register.
2. AND with 0000000011111111. This masks out the character on the left.
3. Perform I/O.

The **arithmetic shift** operation treats the data as a signed integer and does not shift the sign bit. On a right arithmetic shift, the sign bit is replicated into the bit position to its right. On a left arithmetic shift, a logical left shift is performed on all bits but the sign bit, which is retained.

These operations can speed up certain arithmetic operations.

- With numbers in twos complement notation, a right arithmetic shift corresponds to a division by 2, with truncation for odd numbers.
- Both an arithmetic left shift and a logical left shift correspond to a multiplication by 2 when there is no overflow. If overflow occurs, arithmetic and logical left shift operations produce different results, but the arithmetic left shift retains the sign of the number.

Rotate, or cyclic shift, operations preserve all of the bits being operated on.

One use of a rotate is to bring each bit successively into the leftmost bit, where it can be identified by testing the sign of the data (treated as a number).

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

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

TR R1 (L), R2

The operand R2 contains the address of the start of a table of 8-bit codes. The L bytes starting at the address specified in R1 are translated, each byte being replaced by the contents of a table entry indexed by that byte.

For example, to translate from EBCDIC to IRA, we first create a 256-byte table in storage locations, say, 1000-10FF hexadecimal.

The table contains the characters of the IRA code in the sequence of the binary representation of the EBCDIC code; that is, the IRA code is placed in the table at the relative location equal to the binary value of the EBCDIC code of the same character.

Thus, locations 10F0 through 10F9 will contain the values 30 through 39, because F0 is the EBCDIC code for the digit 0, and 30 is the IRA code for the digit 0, and so on through digit 9. Now suppose we have the EBCDIC for the digits 1984 starting at location 2100 and we wish to translate to IRA.

Assume the following:

- 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.

Input/Output

Input/output instructions were discussed in some detail in Chapter 7. As we saw, there are a variety of approaches taken, including isolated programmed I/O, memory-mapped programmed I/O, DMA, and the 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

System control instructions are those 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.

Some examples of system control operations are as follows.

- A system control instruction may read or alter a control register.
- Another example is an instruction to read or modify a storage protection key, such as is used in the EAS/390 memory system.
- Another example is access to process control blocks in a multiprogramming system.

Transfer of Control

For all of the operation types discussed so far, the next instruction to be performed is the one that immediately follows, in memory, the current instruction.

However, a significant fraction of the instructions in any program have as their function changing the sequence of instruction execution. For these instructions, the operation performed by the processor is to update the program counter to contain the address of some instruction in memory.

There are a number of reasons why transfer-of-control operations are required. Among the most important are the following:

1. In the practical use of computers, it is essential to be able to execute each instruction more than once and perhaps many thousands of times. It may require thousands or perhaps millions of instructions to implement an application. This would be unthinkable if each instruction had to be written out separately. If a table or a list of items is to be processed, a program loop is needed. One sequence of instructions is executed repeatedly to process all the data.
2. Virtually all programs involve some decision making. We would like the computer to do one thing if one condition holds, and another thing if another condition holds.
   - For example, a sequence of instructions computes the square root of a number. At the start of the sequence, the sign of the number is tested. If the number is negative, the computation is not performed, but an error condition is reported.
3. To compose correctly a large or even medium-size computer program is an exceedingly difficult task. It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time.

BRANCH INSTRUCTIONS 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**.

There are two common ways of generating the condition to be tested in a conditional branch instruction. First, most machines provide a 1-bit or multiple-bit condition code that is set as the result of some operations. This code can be thought of as a short user-visible register. As an example, an arithmetic operation (ADD, SUBTRACT, and so on) could set a 2-bit condition code with one of the following four values: 0, positive, negative, overflow. On such a machine, there could be four different conditional branch instructions:

BRP X Branch to location X if result is positive.

BRN X Branch to location X if result is negative.

BRZ X Branch to location X if result is zero.

BRO X Branch to location X if overflow occurs.

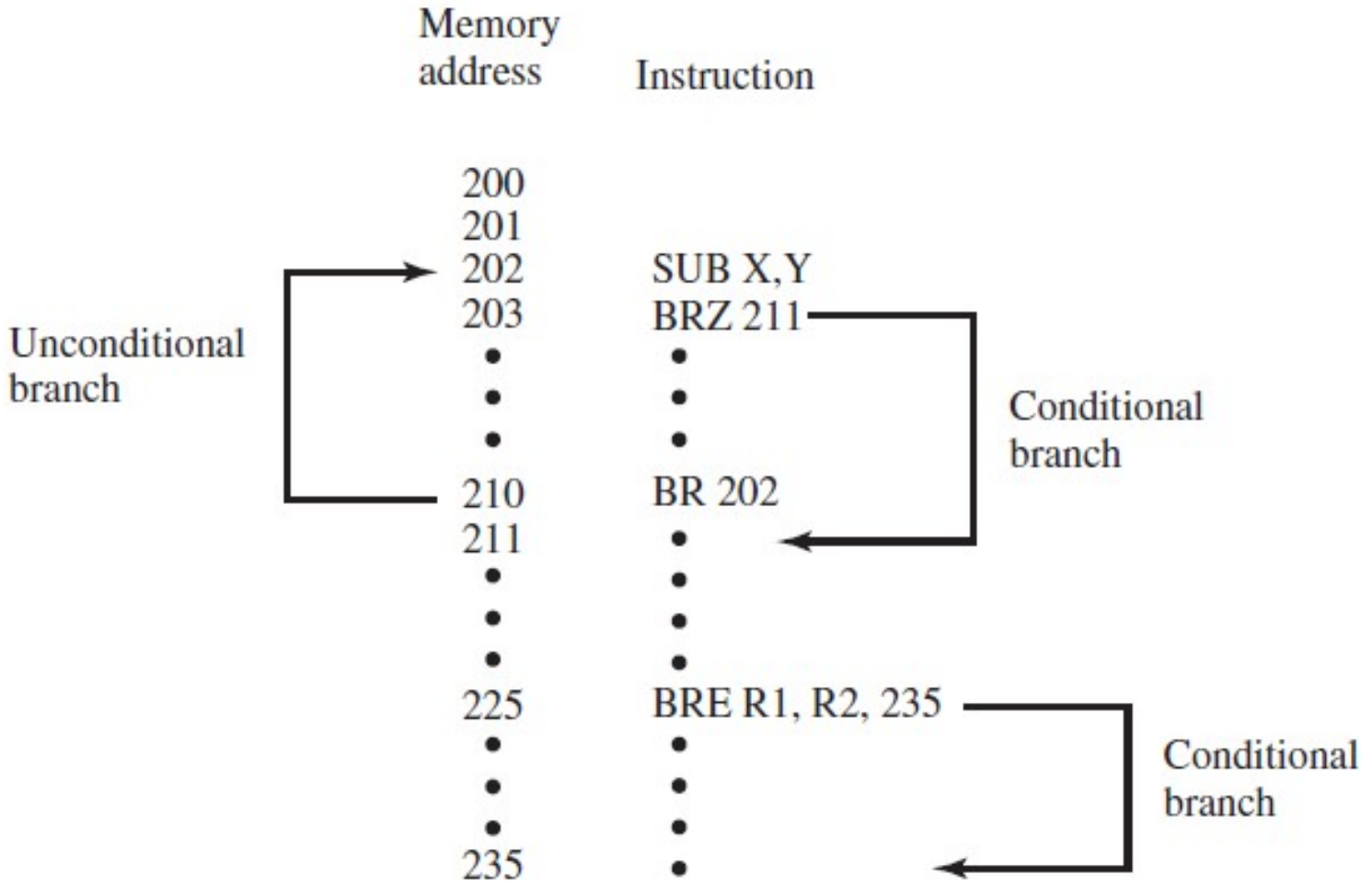
In all of these cases, the result referred to is the result of the most recent operation that set the condition code.

Another approach that can be used with a three-address instruction format is to perform a comparison and specify a branch in the same instruction. For example,

```
BRE R1, R2, X  ;Branch to X if contents of R1 = contents of R2.
```

Figure shows examples of these operations. Note that a branch can be either *forward* (an instruction with a higher address) or *backward* (lower address).

The example shows how an unconditional and a conditional branch can be used to create a repeating loop of instructions. The instructions in locations 202 through 210 will be executed repeatedly until the result of subtracting Y from X is 0.



SKIP INSTRUCTIONS Another form of transfer-of-control instruction is the skip instruction. The skip instruction includes an implied address. Typically, the skip 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. A typical example is the increment-and-skip-if-zero (ISZ) instruction. Consider the following program fragment:

```
301
.
.
.
309 ISZ R1
310 BR 301
311
```

In this fragment, the two transfer-of-control instructions are used to implement an iterative loop. R1 is set with the negative of the number of iterations to be performed. At the end of the loop, R1 is incremented. If it is not 0, the program branches back to the beginning of the loop. Otherwise, the branch is skipped, and the program continues with the next instruction after the end of the loop.

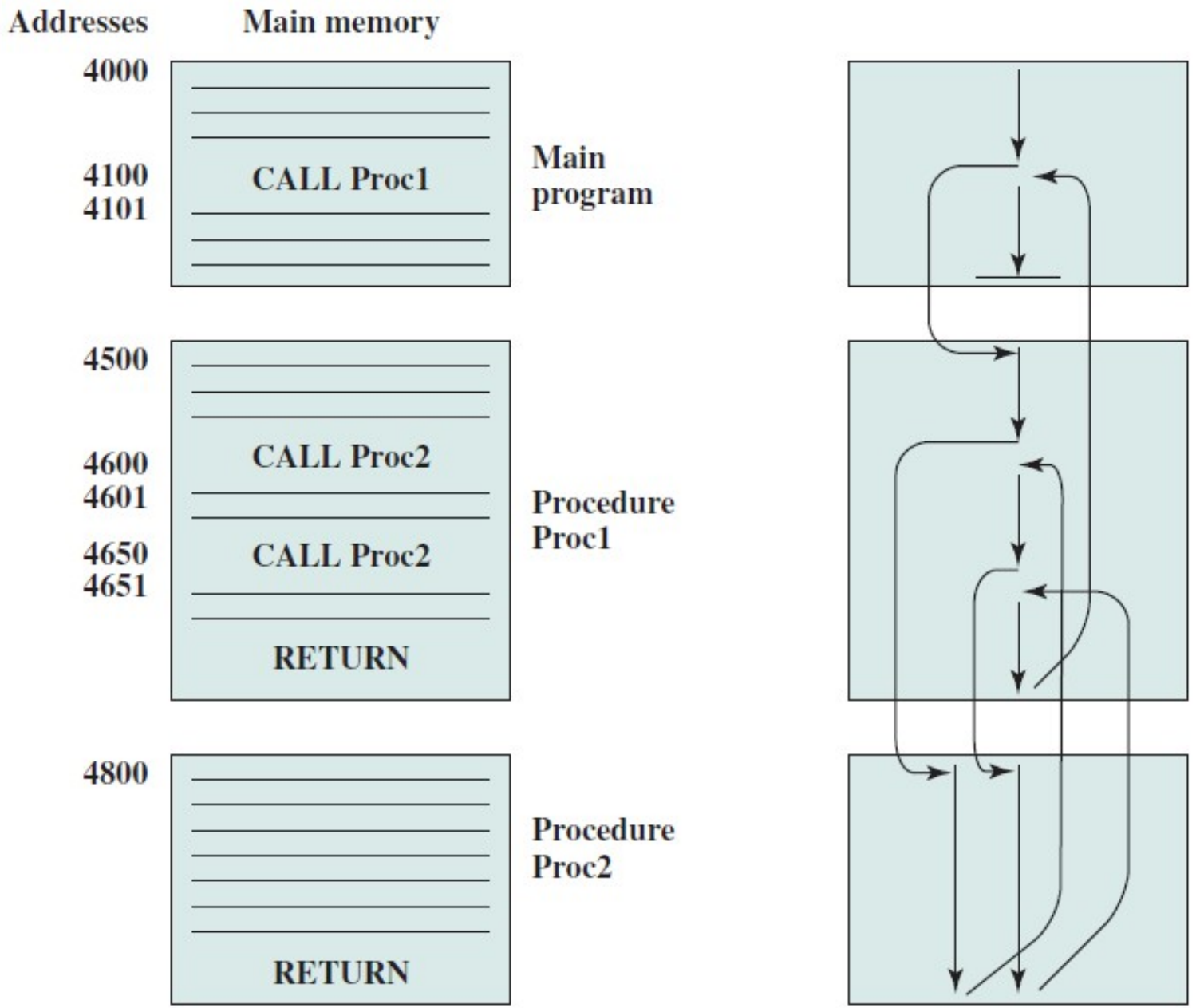
PROCEDURE CALL INSTRUCTIONS Perhaps the most important innovation in the development of programming languages is the *procedure*. A procedure is a self-contained computer program that is incorporated into a larger program. At any point in the program the procedure may be invoked, or *called*. The processor is instructed to go and execute the entire procedure and then return to the point from which the call took place.

The two principal reasons for the use of procedures are economy and modularity. A procedure allows the same piece of code to be used many times. This is important for economy in programming effort and for making the most efficient use of storage space in the system (the program must be stored). Procedures also allow large programming tasks to be subdivided into smaller units. This use of *modularity* greatly eases the programming task.

The procedure mechanism involves two basic instructions: a call instruction that branches from the present location to the procedure, and a return instruction that returns from the procedure to the place from which it was called. Both of these are forms of branching instructions.

Figure (a) illustrates the use of procedures to construct a program. In this example, there is a 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.



Three points are worth noting:

1. A procedure can be called from more than one location.
2. A procedure call can appear in a procedure. This allows the *nesting* of procedures to an arbitrary depth.
3. Each procedure call is matched by a return in the called program.

Because we would like to be able to call a procedure from a variety of points, the processor must somehow save the return address so that the return can take place appropriately. There are three common places for storing the return address:

- Register
- Start of called procedure
- Top of stack

Consider a machine-language instruction CALL X, which stands for *call procedure at location X*. If the register approach is used, CALL X causes the following actions:

RN <- PC + Δ

PC <- X

where RN is a register that is always used for this purpose, PC is the program counter, and Δ is the instruction length. The called procedure can now save the contents of RN to be used for the later return.

A second possibility is to store the return address at the start of the procedure.

In this case, CALL X causes

X <- PC + Δ

PC <- X + 1

This is quite handy. The return address has been stored safely away.

Both of the preceding approaches work and have been used. The only limitation of these approaches is that they complicate the use of *reentrant* procedures. A reentrant procedure is one in which it is possible to have several calls open to it at the same time.

A recursive procedure (one that calls itself) is an example of the use of this feature. If parameters are passed via registers or memory for a reentrant procedure, some code must be responsible for saving the parameters so that the registers or memory space are available for other procedure calls.

A more general and powerful approach is to use a stack. When the processor executes a call, it places the return address on the stack. When it executes a return, it uses the address on the stack.

Figure below illustrates the use of the stack.

Use of stack to implement nested subroutines given above.

In addition to providing a return address, it is also often necessary to pass parameters with a procedure call. These can be passed in registers. Another possibility is to store the parameters in memory just after the CALL instruction.

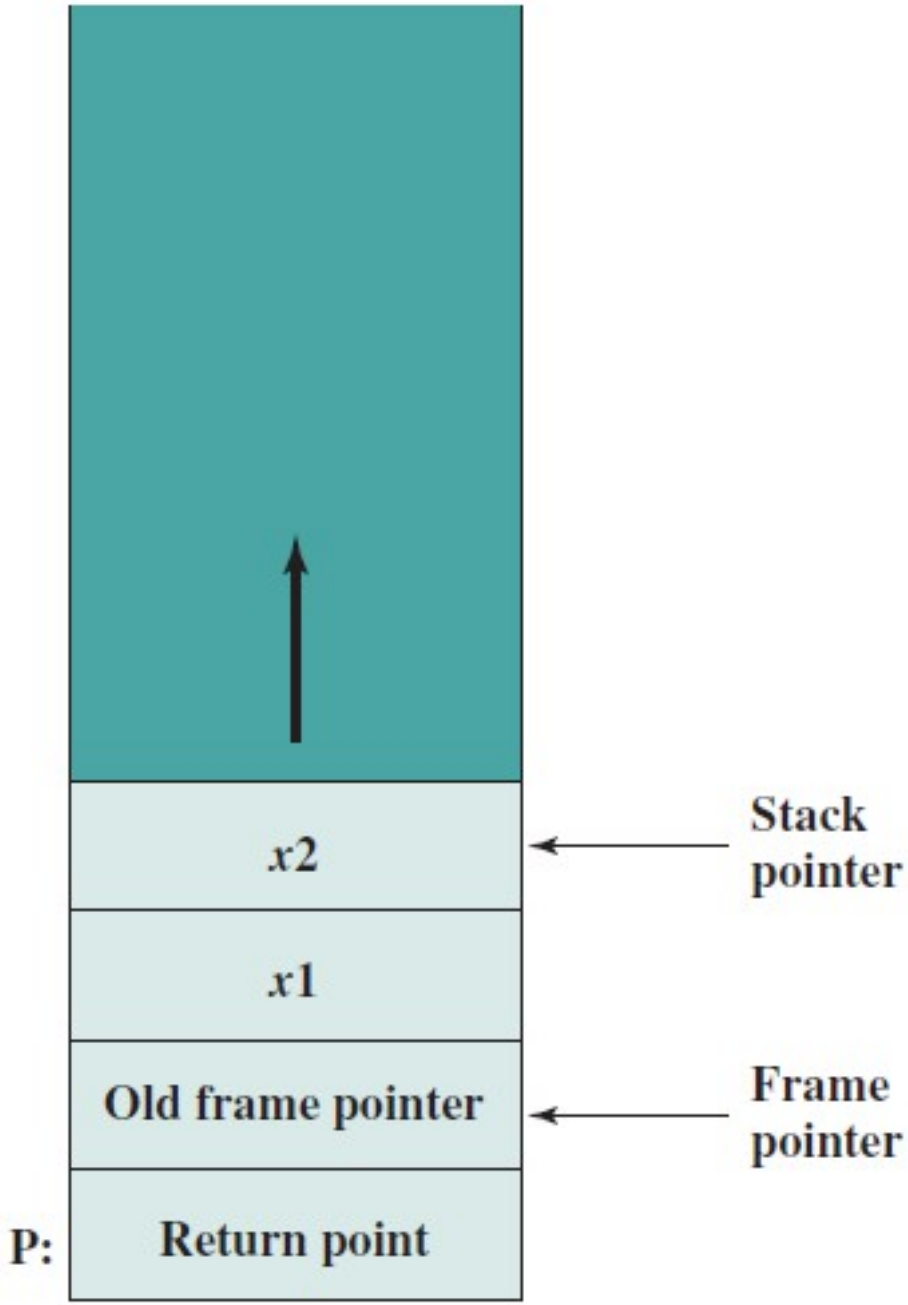
In this case, the return must be to the location following the parameters. Again, both of these approaches have drawbacks. If registers are used, the called program and the calling program must be written to assure that the registers are used properly. The storing of parameters in memory makes it difficult to exchange a variable number of parameters. Both approaches prevent the use of reentrant procedures.

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.

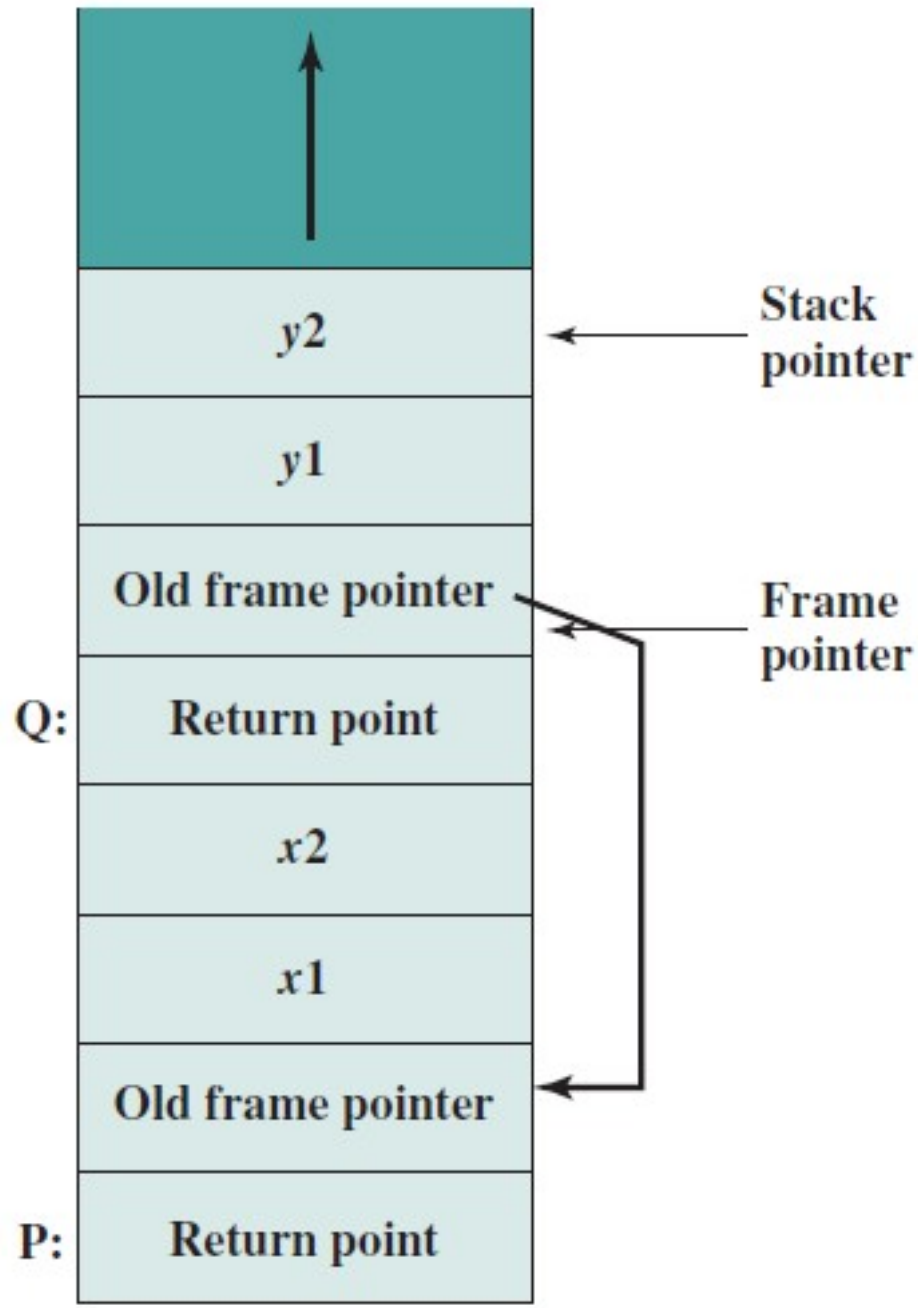
The called procedure can access the parameters from the stack. Upon return, return parameters can also be placed on the stack. The entire set of parameters, including return address, that is stored for a procedure invocation is referred to as a *stack frame*.

An example is provided in the figure below. The example refers to procedure P in which the local variables *x*1 and *x*2 are declared, and procedure Q, which P can call and in which the local variables *y*1 and *y*2 are declared. In this figure, the return point for each procedure is the first item stored in the corresponding stack frame.

Next is stored a pointer to the beginning of the previous frame. This is needed if the number or length of parameters to be stacked is variable.



(a) *P* is active



(b) *P* has called *Q*