Instruction Set: Characteristics and Functions

Bôi:
Hoang Lan Nguyen

Machine Instruction Characteristics

What is an Instruction Set?

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

From the user's side, the user who chooses to program in machine language (actually, in assembly language) becomes aware of the register and memory structure, the types of data directly supported by the machine, and the functioning of the ALU.

Elements of an Instruction

Each instruction must have elements that contain the information required by the CPU for execution. 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 CPU where to fetch the next instruction after the execution of this instruction is complete.

The next instruction to be fetched is located in main memory or, in the case of a virtual memory system, in either main memory or secondary memory (disk). 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. Source and result operands can be in one of three areas:

- Main or virtual memory: As with next instruction references, the main or virtual memory address must be supplied.
- CPU register: With rare exceptions, a CPU 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 number, and the instruction must contain the number of the desired register.
- 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 Cycle State Diagram**

![Instruction Cycle State Diagram](image)

**Figure 10.1** Instruction Cycle State Diagram
**Instruction Representation**

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the instruction. During instruction execution, an instruction is read into an instruction register (IR) in the CPU. The CPU 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>MPY</td>
<td>Multiply</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide</td>
</tr>
<tr>
<td>LOAD</td>
<td>Load data from memory</td>
</tr>
<tr>
<td>STOR</td>
<td>Store data to memory</td>
</tr>
</tbody>
</table>

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.

Simple Instruction Format
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: Memory instructions
- Data movement: I/O instructions
- Control: Test and branch instructions

**Number of Addresses**

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 operand) or binary (two operands). Thus, we would need a maximum of two addresses to reference operands. The result of an operation must be stored, suggesting a third address. 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 operands, one result and the address of the next instruction. In practice, four-address instructions are extremely rare. Most instructions have one, two, or three operand addresses, with the address of the next instruction being implicit (obtained from the program counter).

- Three addresses:
  - Operand 1,Operand 2, Result

Example: \( a = b + c \)
Three-address instruction formats are not common, because they require a relatively long instruction format to hold the three address references.

- Two addresses:
- One address doubles as operand and result

Example: \( a = a + b \)

- The two-address formal 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.
- One addresses:
  - a second address must be implicit. This was common in earlier machines, with the implied address being a CPU register known as the accumulator, or AC. The accumulator contains one of the operands and is used to store the result.
- Zero addresses
  - Zero-address instructions are applicable to a special memory organization, called a Stack. A stack is a last-in-first-out set of locations.

**How Many Addresses?**

The number of addresses per instruction is a basic design decision.

Fewer addresses:

- Fewer addresses per instruction result in more primitive instructions, which requires a less complex CPU.
- 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

Multiple-address instructions:

- 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.

**Design Decisions**

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 CPU and thus has a significant effect on the implementation of the CPU. The instruction set is the programmer's means of controlling the CPU. Thus, programmer requirements must be considered in designing the instruction set. The most important 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 CPU registers that can be referenced by instructions, and their use.
- Addressing: The mode or modes by which the address of an operand is specified

**Types of Operands**

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

- Addresses
- Numbers
- Characters
- Logical data

**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. Thus, the programmer is faced with understanding the consequences of rounding, overflow and underflow.

Three types of numerical data are common in computers:

- Integer or fixed point
- 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), referred to in the United States as the American Standard Code for Information Interchange (ASCII). IRA is also widely used outside the United States. Each character in this code is represented by a unique 7-bit pattern, thus, 128 different characters can be represented. This is a larger number than is necessary to represent printable characters, and some of the patterns represent control characters. Some of these control characters have to do with controlling the printing of characters on a page. Others are concerned with communications procedures. IRA-encoded characters are almost always stored and transmitted using 8 bits per character. The eighth bit may be set to 0 or used as a parity bit for error detection. In the latter case, the bit is set such that the total number of binary 1s in each octet is always odd (odd parity) or always even (even parity).

Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC). EBCDIC is used on IBM S/390 machines. It is an 8-bit code. As with IRA, EBCDIC is compatible with packed decimal. In the case of EBCDIC, the codes 11110000 through 11111001 represent the digits 0 through 9.

**Logical Data**

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

**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
Data transfer

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

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

In terms of CPU action, data transfer operations are perhaps the simplest type. If both source and destination are registers, then the CPU simply causes data to be transferred from one register to another; this is an operation internal to the CPU. If one or both operands are in memory, then the CPU 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 actual memory address.
3. Determine whether the addressed item is in cache.
4. If not, issue a command to the memory module.

Example:

<table>
<thead>
<tr>
<th>Operation mnemonic</th>
<th>Name</th>
<th>Number of bits transferred</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Load</td>
<td>32</td>
<td>Transfer from memory in register</td>
</tr>
<tr>
<td>LH</td>
<td>Load half-word</td>
<td>16</td>
<td>Transfer from memory to register</td>
</tr>
<tr>
<td>ST</td>
<td>Store</td>
<td>32</td>
<td>Transfer from register to memory</td>
</tr>
<tr>
<td>STH</td>
<td>Store half-word</td>
<td>16</td>
<td>Transfer from register to memory</td>
</tr>
</tbody>
</table>
**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

**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 AND, OR, NOT, XOR, …

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

In addition to bitwise logical operations, most machines provide a variety of shifting and rotating functions such as shift left, shift right, right rotate, left rotate…
Conversion

Conversion instructions are those that change the format or operate on the format of data. An example is converting from decimal to binary.

Figure 10.5 Shift and Rotate Operations
**Input/Output**

As we saw, there are a variety of approaches taken, including isolated programmed IO, 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 Controls**

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 S/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 CPU 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.

We now turn to a discussion of the most common transfer-of-control operations found in instruction sets: branch, skip, and 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).

**Skip instructions**

Another common 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.

**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.
The above figure 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 CPU suspends execution of the main program and begins execution of PROC1 by fetching the next instruction from location 4500. Within PROC1, there are two calls to PROC2 at location 4800. In each case, the execution of PROC1 is suspended
and PROC2 is executed. The RETURN statement causes the CPU to go back to the calling program and continue execution at the instruction after the corresponding CALL instruction. This behavior is illustrated in the right of this figure.

Several 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 CPU 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

**Addressing Modes**

The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references and/or the complexity of address calculation, on the other. In this section, we examine the most common addressing techniques:

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

**Immediate Addressing**

The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction:
Operand is part of instruction
Operand = address field
e.g. ADD 5 ;Add 5 to contents of accumulator ;5 is operand

### Instruction

<table>
<thead>
<tr>
<th>Op-code</th>
<th>Operand</th>
</tr>
</thead>
</table>

The advantage of immediate addressing is that 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.

The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

### Direct Addressing

A very simple form of addressing is direct addressing, in which:

- Address field contains address of operand
- Effective address (EA) = address field (A)
e.g. ADD A ;Add contents of cell A to accumulator
The technique was common in earlier generations of computers but is not common on contemporary architectures. It requires only one memory reference and no special calculation. The obvious limitation is that it provides only a limited address space.

**Indirect Addressing**

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as indirect addressing.
Register Addressing

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address.
The advantages of register addressing are that:

- Only a small address field is needed in the instruction
- No memory references are required, faster instruction fetch

The disadvantage of register addressing is that the address space is very limited.

**Register Indirect Addressing**

Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register. Thus, for register indirect address: Operand is in memory cell pointed to by contents of register.
The advantages and limitations of register indirect addressing are basically the same as for indirect addressing. In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address. In addition, register indirect addressing uses one less memory reference than indirect addressing.

**Displacement Addressing**

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context of its use but the basic mechanism is the same. We will refer to this as displacement addressing, address field hold two values:

- A = base value
- R = register that holds displacement
Instruction Set: Characteristics and Functions

Instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Register R</th>
<th>Address A</th>
</tr>
</thead>
</table>

Registers

Pointer to Operand

Memory

Operand