**Chapter 12: Instruction Sets – Characteristics and Functions**

Table of Contents

[12.1 Machine Instruction Characteristics 4](#_Toc48999230)

[Elements of Machine Instruction 4](#_Toc48999231)

[Instruction Representation 5](#_Toc48999232)

[Instruction Types 7](#_Toc48999233)

[Number of Addresses 8](#_Toc48999234)

[Instruction Set Design 12](#_Toc48999235)

[12.2 Types of Operands 13](#_Toc48999236)

[Numbers 13](#_Toc48999237)

[Characters 14](#_Toc48999238)

[Logical Data 15](#_Toc48999239)

[12.4 Types of Operations 17](#_Toc48999240)

[Data Transfer 17](#_Toc48999241)

[Arithmetic 20](#_Toc48999242)

[Logical 21](#_Toc48999243)

[Conversion 25](#_Toc48999244)

[Input / Output 26](#_Toc48999245)

[System Control 27](#_Toc48999246)

[Transfer of Control 28](#_Toc48999247)

[Branch Instructions 30](#_Toc48999248)

[Skip Instructions 31](#_Toc48999249)

[Procedural Call Instructions 32](#_Toc48999250)

[Appendix 12A: Little-, Big- and Bi-Endian 36](#_Toc48999251)

[Byte Ordering 36](#_Toc48999252)

[Bit Ordering 40](#_Toc48999253)

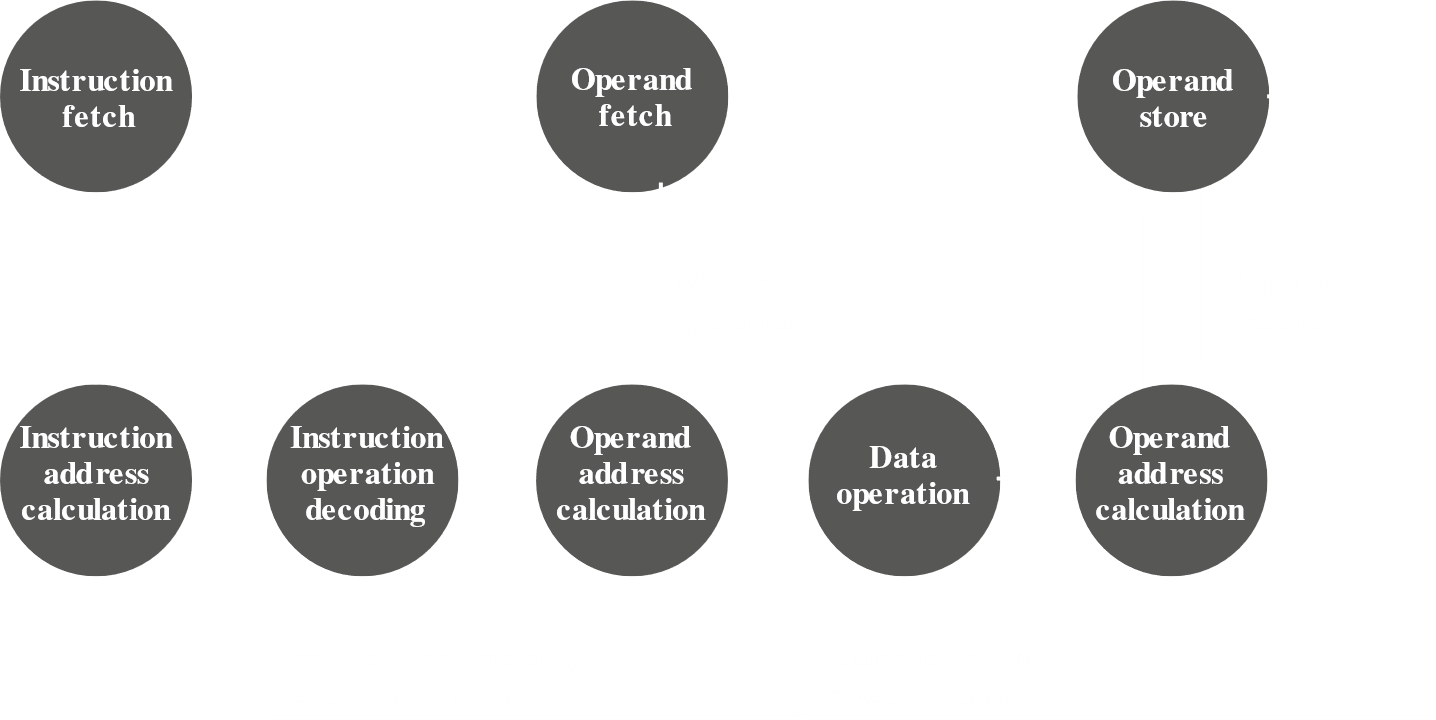
Much of what we discuss in this course is not apparent to the average programmer since they do not see it in everyday life. Machine instructions sets are where things become a little more visible. The machine instruction set provides the functional requirements for the processor. If a user were to program in machine language (assembly language), they would become aware of the register and memory structure, the types of data directly supported by the machine and the functioning of the ALU. We will be focusing on machine instructions here.

## 12.1 Machine Instruction Characteristics

The operation of the processor is determined by the instructions it executes, called machine instructions or computer instructions. The collection of different instructions that the processor can execute is called the processor’s instruction set.

### Elements of Machine Instruction

Each instruction must have the information needed by the processor for execution. We have seen the diagram for the instruction cycle states before. The same diagram helps us identify the different elements of a machine instruction.



* **Operation Code**: This specifies the operation to be performed (e.g. ADD, I/O). It is a binary code, and is also called an opcode.
* **Source Operand Reference**: These are references to any inputs that may be involved in the operation.
* **Result Operand Reference**: These are references to the outputs of the operation.
* **Next Instruction Reference**: This tells the processor where to get the next instruction.

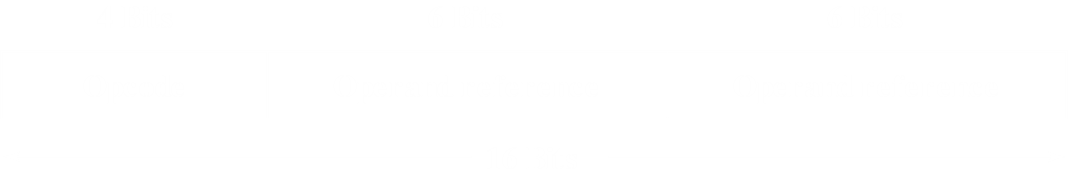
The address of the next instruction could be real or virtual. Generally, the difference is visible to instruction set architecture. Mostly, the next instruction immediately follows the current one so no explicit reference to the next instruction is needed. If an explicit reference is needed, the main memory or virtual memory address must be given. The form in which this is given is discussed in the next chapter.

Source and result operands can be from one of four places:

* **Main or Virtual Memory**: The address must be given.
* **Processor Register**: A processor usually contains registers that can be referenced by machine instructions. If there is only one register, the reference may be implicit, but if there are more, then each register is given a unique name or number which is included in instructions.
* **Immediate**: The value of the operand is contained in a field in the instruction being executed.
* **I/O Device**: The instruction must provide 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

Each instruction is represented by a sequence of bits, divided into fields corresponding to the constituent elements of the instruction.



With most instruction sets, more than one format is used. During execution, an instruction is read into the instruction register (IR) in the processor. The processor must be able to extract the data from the different instruction fields to be able to perform the required operation.

Binary representations of machine instructions are difficult to read, so it is common practice to use symbolic representation. Opcodes are represented by abbreviations, called mnemonics. Common examples are ADD, SUB, MUL, DIV, LOAD (to load data from memory) and STOR (to store data in memory). Operands are also represented symbolically, such as ADD R, Y meaning add the value in data location Y to the register R.

Symbolic representation makes it possible to write machine-language programs in symbolic form. Each symbolic opcode has a fixed binary representation, and the location of each symbolic operand is specified beforehand such as X = 513 and Y = 514. A simple program accepts symbolic input, converts opcodes and operand references to binary form and constructs the binary machine instructions.

Machine-language programmers are almost non-existent nowadays. Most use high-level languages or assembly language at most. Symbolic machine language is still useful in describing machine instructions.

### Instruction Types

Consider a high-level language instruction, such as X = X + Y. This tells the computer to add the value stored in Y to the value stored in X and put the result in X. Say the variables X and Y represent the locations 513 and 514. With a simple set of machine instructions, the operation could be finished within 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.

A single high-level instruction needed three machine instructions. High-level instructions express operations in short, algebraic forms, using variables. Machine language expresses operations in a basic form involving movement of data to or from registers.

Consider the types of instructions that could be included in a real computer. There should be some instructions to let the user formulate any data processing task. Another way of looking at this is to look at the capabilities of a high-level programming language, since it must be translated into machine language. This means the machine instruction set needs to be sufficient to express any high-level language instructions.

With these things in mind, the categories of instruction type are:

* **Data Processing**: Arithmetic and logic instructions
* **Data Storage**: Movement of data to and from registers and memory locations
* **Data Movement**: I/O instructions
* **Control**: Test and branch instructions

Arithmetic instructions give computational capabilities to process numeric data.

Logic instructions allow processing for any type of data the user wants, since they operate on the bits of a word as bits rather than as numbers. They are used primarily with data in processor registers, and thus need the help of memory instructions.

I/O instructions are needed to transfer programs and data into memory, and results 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 different sets of instructions based on the decision made.

All the instruction types will be examined in greater detail a little later in this chapter.

### Number of Addresses

One of the traditional ways of describing processor architecture is in terms of the number of addresses in each instruction. This has become less significant now since processor design is much more complex, but it is still useful at this point to draw and analyse the distinction.

The maximum number of addresses needed in an instruction varies. Arithmetic and logic instructions need the most operands, and almost all of those are either unary or binary, meaning a maximum of two addresses are needed to reference source operands. A third address would be needed to store the results, and a fourth address is needed for the next instruction. In most architectures, this fourth address is not needed, since the address of the next instruction is implicitly obtained from the program counter. Most architectures also have a few special instructions with more operands, such as the load and store multiple instructions in the ARM architecture which can hold up to 17 operands in a single instruction.

Typical one-, two- and three-address instructions used to computer Y = (A – B) / [C + (D \* E)] is shown below.

|  |  |  |
| --- | --- | --- |
| **Three-Address Instructions** | | |
| **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 |

|  |  |  |
| --- | --- | --- |
| **Two-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 |

|  |  |  |
| --- | --- | --- |
| **One-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 |

With three addresses, each instruction refers to two source operands and one destination operand. A temporary location, T, is used to store some intermediate results. With five operands in the original expression, four instructions were needed. Three-address instruction formats are not common, since they need a long instruction format.

With two-address instructions, for binary operations, the result is stored in one of the source operand’s address. Space is saved, but it is somewhat awkward. To avoid altering the value of an operand, the MOVE instruction was used to move the value of the operand that would be overwritten to a temporary location. Six instructions are needed in total.

One-address instructions are the simplest, since the second address is implicit. These were common in earlier machines, where the implied address was a processor register called the accumulator (AC). The AC contains one of the operands and is used to store the result. Eight instructions are needed in total.

With some instructions, it is possible to have zero address. Zero-address instructions apply to a special memory organization called a stack. A stack is a last-in-first-out set of locations, and is kept in a known location. Often, at least the top two elements are in processor registers and are referenced by zero-address instructions. The use of stacks is discussed further later on in this chapter and in the next one.

Below, a table summarizing the interpretations of instructions for the different types of addresses is given. The next instruction’s address is assumed to be implicit in all cases. The test case is for an operation with two sources and one result, OP being the operation.

|  |  |  |
| --- | --- | --- |
| **Number of Addresses** | **Symbolic Representation** | **Interpretation** |
| 3 | OP A, B, C | A 🡨 B OP C |
| 2 | OP A, B | A 🡨 A OP C |
| 1 | OP A | AC 🡨 AC OP A |
| 0 | OP | T 🡨 (T – 1) OP T |

The number of addresses per instruction is a basic design decision. With fewer address per instruction, the instructions are more primitive and need a less complex processor. They are also shorter. However, they cause programs to have more instructions in total, which means loner execution times and longer, more complex programs.

The threshold between one-address and multi-address instructions is an important one. With one-address instructions, the programmer has only one general-purpose register available, the accumulator. With multi-address instructions, multiple general-purpose registers tend to be available, allowing some operations to be performed solely on registers. Since register references are faster than memory references, this reduces execution time. For the reasons of flexibility and availability of multiple registers, most contemporary machines use two- or three-address instructions.

There are also other factors complicating this design choice. The issue of whether an address references a memory location or a register comes up, since fewer registers means fewer bits are needed to reference a register. In the next chapter, we will also see that a machine may offer a variety of addressing modes and one or more bits is needed to specify the mode. All of this results in most processor designs having a variety of instruction formats.

### Instruction Set Design

Instruction set design is one of the most important and most analysed aspects of computer design. Since it affects so many aspects of the computer system, the design of an instruction set is very complex. The design choice 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 how the programmer controls the processor, so the programmer’s needs must also be considered when designing it.

The most fundamental issues related to the design of instruction sets is still disputed. The most important of these include:

* **Operation Repertoire**: How many and which operations to provide, and how complex they should be
* **Data Types**: The 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**: How many processor registers can be referenced by instructions, and their use
* **Addressing**: The mode or modes by which the address of an operand is specified

The issues are all interrelated and must all be considered together. They will be discussed in some sequence in this chapter and the ones after this, but the interrelations must be noted nonetheless.

## 12.2 Types of Operands

The most important general categories of data are:

* Addresses
* Numbers
* Characters
* Logical Data

When discussing addressing modes in the next chapter, we shall see that address are actually a form of data. In many cases, calculations need to be performed on the operand reference in an instruction to determine the main or virtual memory address. In this context, the address can be considered to be an unsigned integer.

The other common data types are briefly examined here. Some machines have further special data types and data structures, such as machine operations that operate directly on a list or a string.

### Numbers

All machine languages include numeric data types. Even in non-numeric processing, numbers are needed to use as counters, field widths, etc. An important difference between numbers in ordinary mathematics and numbers in computers, is that the latter is limited. There is a limit to the magnitude of numbers representable on a machine and, in the case of floating-point numbers, also a limit to their precision. This is why programmers are forced to deal with rounding, overflow and underflow.

The three common types of numeric data are:

* Binary Integer or Binary Fixed-Point
* Binary Floating-Point
* Decimal

Binary fixed-point and floating-point numbers were discussed thoroughly in a previous chapter, but there are a few more things to say about decimal numbers. Decimals are presented to the user, but are not used by the computer. This means some conversion has to take place in both directions. For applications with a great deal of I/O but only a little bit of simple computation, it is easier to store and operate on numbers in decimal form. This is where packed decimals come in.

Packed decimals are numbers where each decimal digit is represented as a 4-bit code in the obvious way, storing two digits per byte. 0 is 0000, 1 is 0001 and so on. This is inefficient, because only 10 of the 16 possible 4-bit values are used. Numbers are formed by stringing the 4-bit codes together, usually in multiples of 8 bits. Thus, 246 is 0000 0010 0100 0110. This is of course less compact than binary representation, but it avoids the need for conversion. Negative numbers are represented by including a 4-bit sign digit at either the left or right end of a string of packed decimal digits, 1100 for positive and 1101 for negative.

Many machines have arithmetic instructions to operate directly on packed decimal numbers. The algorithms are similar to those for binary representations of numbers, but take into account the decimal carry operation.

### Characters

Character data refers to text or character strings. While these are convenient for humans, they cannot be stored or transmitted as they are by data processing and communications systems, which are designed for binary data. This has resulted in a number of codes being made to represent characters as a sequence of bits. This began with Morse code, and has cumulated in the International Reference Alphabet (IRA) or American Standard Code for Information Interchange (ASCII) used today.

In ASCII code, each character is given a unique 7-bit pattern, with 128 characters in total being represented. This is more than enough for printable characters, and includes control characters, which control how characters are printed, and other concerned with communication procedures. IRA-encoded characters are almost always stored and transmitted using 8-bits per character, with the 8th bit 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 (for odd parity) or even (for even parity).

The IRA bit pattern for the digits 0 through 9 follow the pattern 011XXXX, with the last 4 bits being the same as those used in packed decimal representation. This makes it easy to convert between the two.

Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC), used on IBM mainframes. It is an 8-bit code as well, and is also compatible with packed decimal representation, following the pattern 1111XXXX.

### Logical Data

Normally, each word or addressable unit (byte, halfword, etc.) is treated as a single unit of data. Sometimes however, it is useful to consider an -bit unit as 1-bit items of data, each item having a value of 0 or 1. Data viewed in this way is considered to be logical data.

Logical data has two advantages. Firstly, we may wish to store an array of Boolean or binary data items. This allows more efficient use of memory. Secondly, we may want to manipulate the bits of a data item, such as to shift significant bits for a floating-point number. This is also useful to extract the rightmost 4-bits of an IRA code to use as packed decimal.

Notice how the same data is being treated sometimes as logical data and sometimes as numerical data or text. The ‘type’ of a unit of data is determined by the operation being performed on it. This is not the case in high-level languages, but it is almost always the case in machine language.

## 12.4 Types of Operations

Though the number of different opcodes vary, the same general operations are found on all machines. These include:

* Data Transfer
* Arithmetic
* Logical
* Conversion
* I/O
* System Control
* Transfer of Control

### Data Transfer

|  |  |
| --- | --- |
| **Operation Name** | **Description** |
| Move (Transfer) | Transfer a word or block from source to destination |
| Store | Transfer a word from processor to memory |
| Load (Fetch) | Transfer a word from memory to processor |
| Exchange | Swap contents of source and destination |
| Clear (Reset) | Transfer a word of 0s to destination |
| Set | Transfer a word of 1s to destination |
| Push | Transfer a word from source to top of stack |
| Pop | Transfer a word from top of stack to destination |

Data transfer is the most fundamental type of machine instruction. The data transfer instruction must specify several things. The location of the source and destination operands must be specified, and each location could be from memory, a register or the top of the stack. Secondly, the length of data to be transferred must be indicated. Thirdly, the mode of addressing for each operand must be specified. This last part is discussed in the next chapter.

The choice of data transfer instructions to include in an instruction set shows 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.

The table below shows some of the most common IBM EAS/390 data transfer instructions. There are variants for different amounts of data, and different instructions for register to register, register to memory, memory to register and memory to memory transfers. In contrast, the VAX has a move instruction with variants for different amounts of data to be transferred, but which specifies whether an operand is from the register or memory as part of the operand. The second approach is easier for the programmer, since there are fewer mnemonics, but it is also less compact because the location of each operand must be specified separately in the instruction. This topic is further discussed under instruction formats in the next chapter.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 the simplest. If both source and destination are registers, then the processor must simply transfer data from one register to another, which is an operation internal to the processor. If one or both operands are in memory, some or all of the following actions are needed:

1. Calculate the memory address, based on the address mode (discussed in the next chapter)
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

|  |  |
| --- | --- |
| **Operation Name** | **Description** |
| 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 |

Most machines provide basic arithmetic operations like 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.

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. In addition, of course, the ALU portion of the processor performs the desired operation. The processor may also need to set condition codes and flags.

### Logical

|  |  |
| --- | --- |
| **Operation Name** | **Description** |
| AND | Perform logical AND |
| OR | Perform logical OR |
| NOT | Perform logical NOT |
| Exclusive OR | Perform logical XOR |
| Test | Test specified condition and set flags based on outcome |
| Compare | Make logical or arithmetic comparisons of two or more operands and set flags based on outcome |
| Set Control Variables | Set controls for protection purposes, interrupt handling, timer control, etc. |
| Shift | Left or right shift operand and introduce constants at the other end |
| Rotate | Left or right shift operand and wraparound |

Like arithmetic operations, logical operations involve the ALU and may involve data transfer.

Most machines provide many ways to manipulate the individual bits of a word or other addressable units, often called ‘bit twiddling’. These are based on Boolean operations. Some of the basic operations are shown below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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 |

These logical operations can be applied bitwise to -bit logical data units.

R1 = 10100101

R2 = 00001111

R1 AND R2 = 00000101

Here, the AND operation is being used as a mask to select certain bits in a word and zero out the remaining bits. In a similar manner,

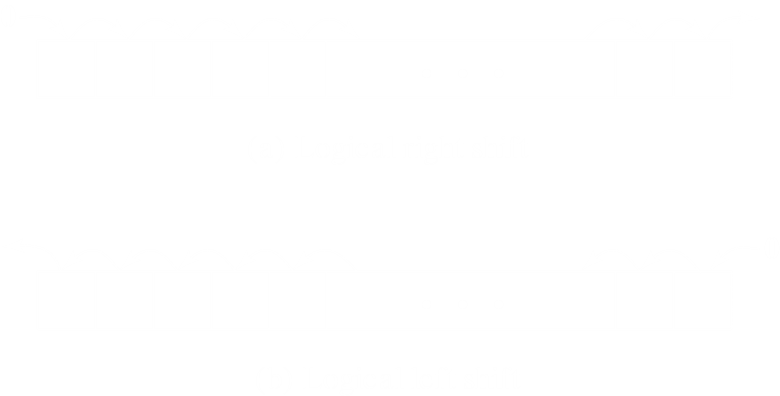
R1 = 10100101

R2 = 11111111

R1 XOR R2 = 01011010

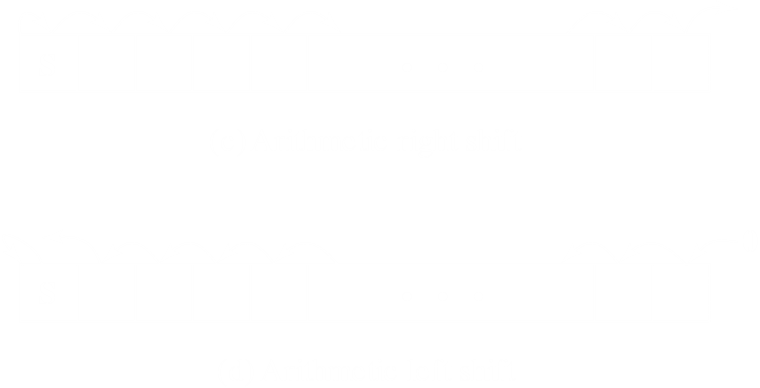
The XOR operation has been used to invert all the bits of R1. This is basically the 1s complement operation.

Most machines also provide a variety of shifting and rotating functions. In a logical shift, the bits of a word are shifted left or right. On one end, the bits shifted out are lost. On the other end, 0s are shifted in.



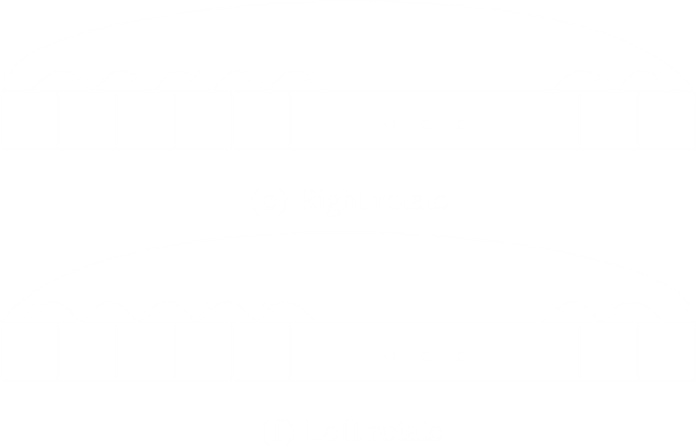
Logical shifts help to isolate fields within a word. The 0s shifted into a word displace unwanted information that is shifted off the other end. For example, say we want to transmit characters of data to an I/O device one character at a time. If each memory word is 16 bits in length and has two characters, we need to load the word into a register and shift to the right 8 times to leave just the word on the left. We can send this to I/O now. Then we load the word into the register again, AND with 0000000011111111 which ‘removes’ the left word and leaving the word on the right. We can then send this to I/O as well.

In arithmetic shift, the data is treated as a signed integer and the sign bit is not shifted. On a right shift, the sign bit is replicated into the bit position on its right. On a left shift, a logical left shift is performed on all bits except the sign bit.



Arithmetic shifting can make some arithmetic operations faster. With numbers in twos complement notation, a right arithmetic shift is the same as division by 2, with truncation for odd numbers. Both arithmetic and logical left shifts correspond to multiplication by 2 when there is no overflow. If overflow occurs, arithmetic and logical left shifts give different results, but arithmetic left shift retains the sign of the number. Due to the potential for overflow, many processors do not include the instruction at all.

Rotation, or cycle shift, preserves all 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, treating the data as a number.



The table below gives examples of all the shift and rotate operations discussed here.

|  |  |  |
| --- | --- | --- |
| **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 |

The functions of the processor for logical operations is the same as for arithmetic operations.

### Conversion

|  |  |
| --- | --- |
| **Operation Name** | **Description** |
| Translate | Translate values in a section of memory based on a table of correspondence |
| Convert | Convert the contents of a word from one form to another (e.g. packed decimal to binary) |

Conversion instructions change the format or operate on the format of data. Decimal to binary conversion is an example of a conversion instruction being executed.

To properly understand this, we need to look at a more complicated example. Take the Translate (TR) instruction. It is used to convert between 8-bit codes and takes three operands.

TR R1 (L), R2

The operand R2 has the address of the start of a table of 8-bit codes. The L bytes starting at the address 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 need to create a 256-byte table in storage locations, say 1000-10FF (hexadecimal). The table will contain the characters of IRA code in the sequence of the binary representation of the EBCDIC code, meaning the IRA code is placed in the table at the relation equivalent location to the binary value of the EBCDIC code of the same character.

Thus, locations 10F0 through 10F9 will contain the values 30 through 39, since F0 is the EBCDIC code for the digit 0 and 30 is the IRA code for the digit 0 and so on. Similarly, for the digits 1984 starting at location 2100, the following scenario will be present:

* Locations 2100-2103 contain F1, F9, F8 and F4 in that order
* R1 contains 2100
* R2 contains 1000

Thus, when the instruction TR R1 (4), R2 is executed, 2100-2103 will contain 31, 39, 38 and 34 in that order.

The work of the processor for conversion operations is similar to that for arithmetic and logical operations. Additionally, there may be special logic to perform conversions.

### Input / Output

|  |  |
| --- | --- |
| **Operation Name** | **Description** |
| 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 |

Input / output instructions have already been discussed in a previous chapter. There were a variety of approaches, such as 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.

For I/O operations, the processor may need to issue commands to I/O modules and determine memory-mapped addresses for memory-mapped I/O.

### System Control

System control instructions can only be executed 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 operating system.

Some examples of system control operations include:

* A system control instruction that may read or alter a control register (control registers are discussed in a future chapter)
* An instruction to read or modify a storage protection key
* Access to process control blocks in a multiprogramming system

### Transfer of Control

|  |  |
| --- | --- |
| **Operation Name** | **Description** |
| 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, then jump to specified address |
| Return | Replace contents of PC and other registers 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 |

For all the operation types we discussed so far, the next instruction is always the one that immediately follows the current one. However, many instructions in any program are meant to change 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 many reasons why transfer-of-control operations are needed. Some of the most important ones are:

1. In the practical use of computers, it is essential to be able to execute each instruction many times. An application may need thousands, maybe millions of instructions and having to write each instance of an instruction separately is unthinkable. To processor a table or list, a program loop is needed where one sequence of instructions is executed repeatedly to process all the data.
2. Almost all programs include some decision making. If one condition holds, one thing should be done and if another condition holds, something else should be done.
3. To correctly compose a medium or large computer program is a very difficult task. It helps if there are mechanisms for breaking the task into smaller pieces that can be worked on one at a time.

For transfer-of-control operations, the processor may need to update the program counter, and manage parameter passing and linkage for subroutine calls and returns.

Next, we will look at the most common transfer-of-control operations found in instruction sets: branch, skip and procedure call.

#### Branch Instructions

A branch or 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, meaning the branch is made (i.e. the program counter is updated to the address specified in the operand) only if a certain condition is met. Otherwise, the next instruction in sequence is executed as normal. A branch instruction where the branch is always taken is an unconditional branch.

There are two common ways to generate 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. For example, an arithmetic operation could set a 2-bit condition code with one of four values: 0, positive, negative or 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

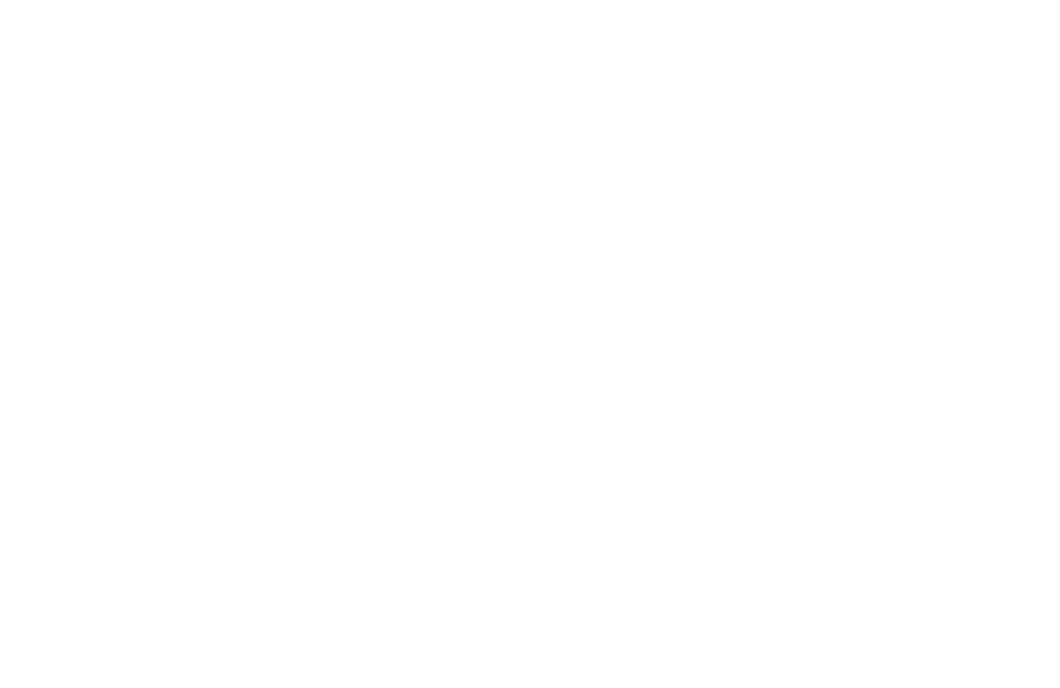
BRZ X Branch to location X if result is zero

BRO X Branch to location X if overflow occurs

In all 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.

BRE R1, R2, X Branch to X if contents of R1 and R2 are the same



The figure above shows examples of these operations. Note that a branch can be either forward, if the instruction has a higher address, or backward, if the instruction has a lower address. The example specifically shows how an unconditional and a conditional branch can be used to create a loop that keeps repeating until the condition is met.

#### Skip Instructions

The skip instruction includes an implied address. It implies that one instruction be skipped, so the implied address is the next address plus one instruction length. Because the skip instruction does not need a destination address field, it is free to do other things.

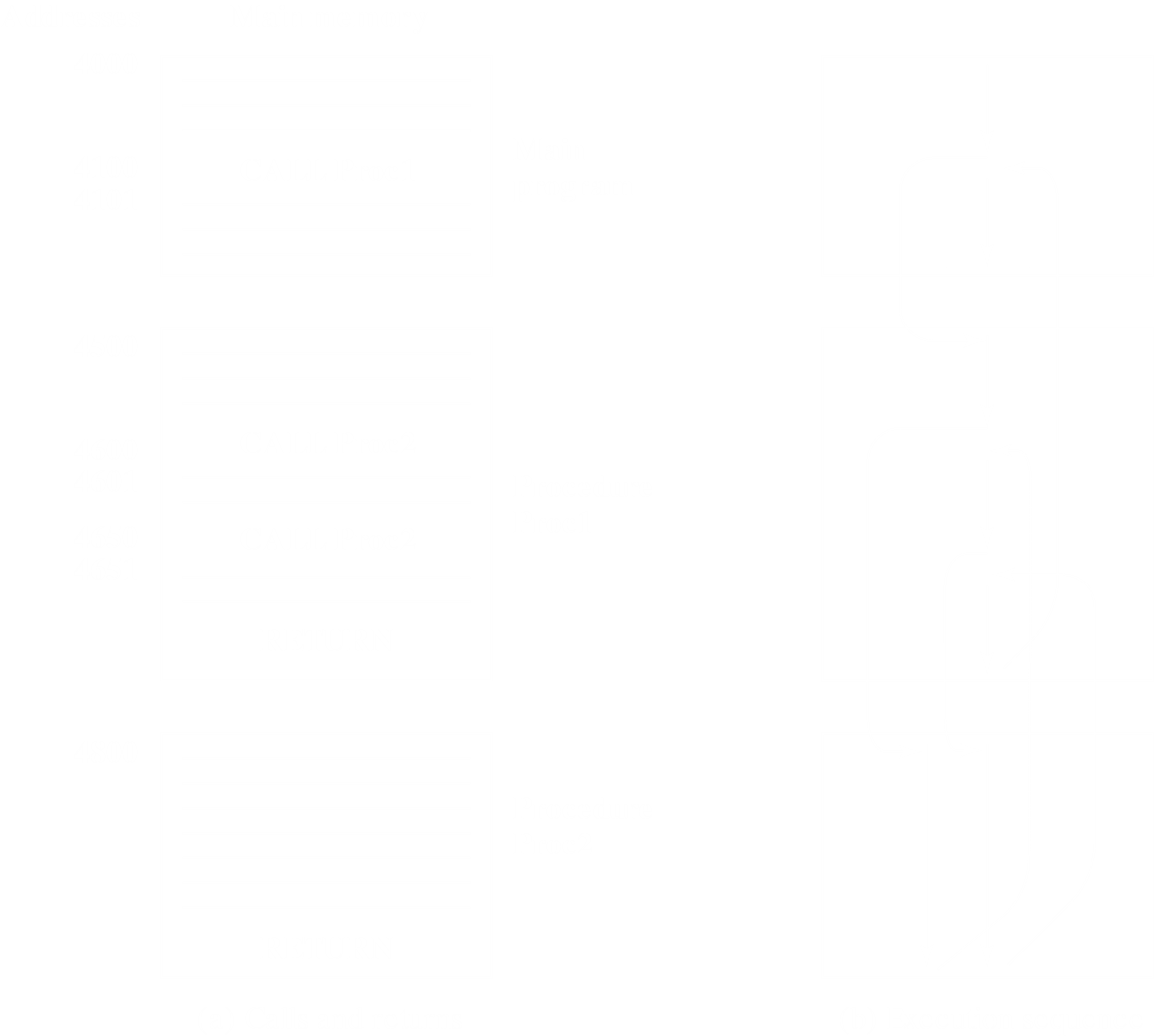
A typical example is the increment-and-skip-if-zero (ISZ) instruction. Consider the following situation. An ISZ instruction is given at the memory location 309, where R1 resides. R1 has the negative of the count of an iterative loop from 301 to 310. At the memory location 310, there is a BR instruction that takes the program back to 301. R1 is incremented on every loop. Thus, the loop will continue until R1 is 0. Once R1 is 0, the ISZ instruction will detect it and skip the BR instruction at 310, ending the loop.

#### Procedural Call Instructions

The procedure is perhaps the most important innovation in the development of programming languages. A procedure is a self-contained program inside a larger program. At any point of the larger program, the procedure may be invoked, or called, and 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 main reasons for using procedures is reusability and organization. A procedure lets the same piece of code be used as many times as needed, which makes it economical and is an efficient use of storage space in the system. Procedures also allow large programs to be divided into smaller units, which makes the task of programming easier.

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 where it was called. Both of these are forms of branching instructions.



Notice that a procedure can be called in more than one location, a procedure can include other procedure calls, allowing nested procedures to an arbitrary depth, and that each procedure call is matched with a return in the called program.

To allow a procedure to be called from different points, the processor must save the return address. This is commonly stored in either the register, the start of the called procedure or the top of the stack.

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

RN 🡨 PC +

PC 🡨 X

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

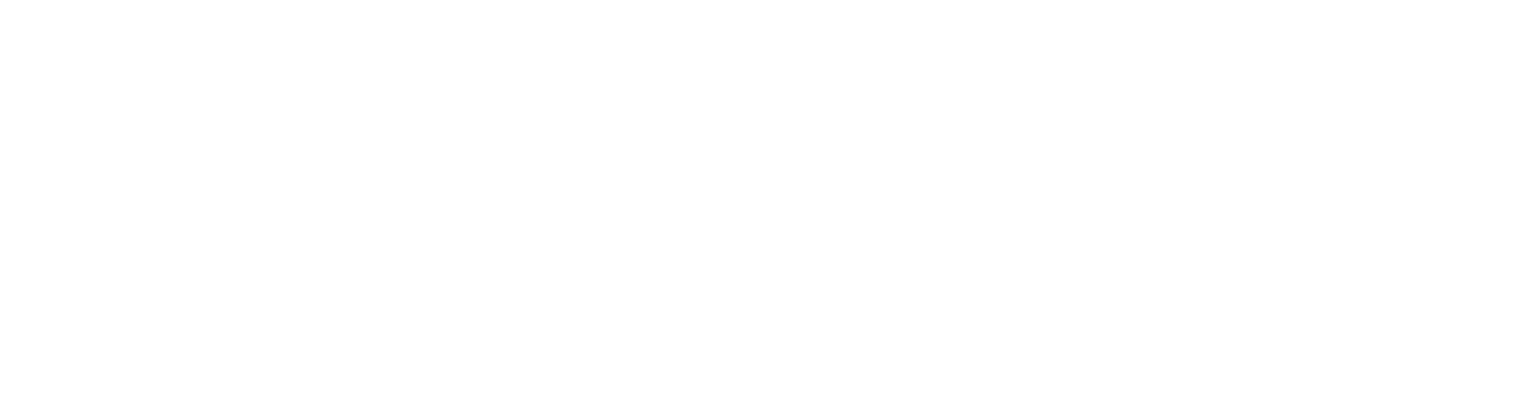
If instead, the return address is stored at the start of the procedure, CALL X causes this:

X 🡨 PC +

PC 🡨 X + 1

Both of the approaches above work and have been used, but 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 is an example 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 is available for other procedure calls.

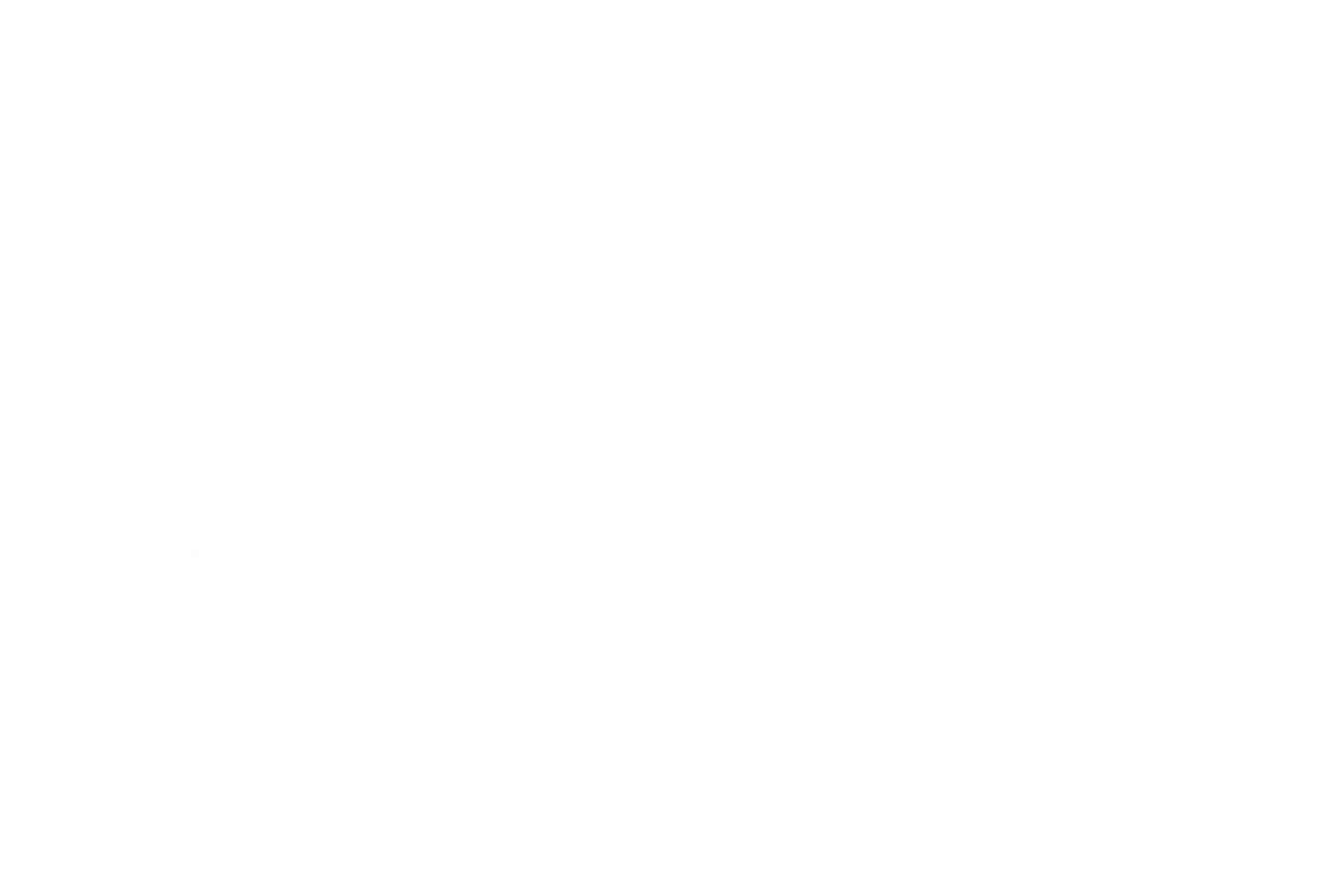
A more general and powerful approach is to use a stack. When the processor executes a call, it places the return address on the stack. When it executes a return, it uses the address on the stack.



In addition to the return address, parameters must also often be passed to a procedure call. These can be passed in registers. Alternatively, the parameters could be stored in memory just after the CALL instruction. In this case, the return must be to the location following the parameters.

Again, both of these methods have flaws. 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 on the other hand, makes it difficult to exchange a variable number of parameters. Both of these drawbacks make it impossible to use reentrant procedures.

Again, stacks allow us an escape route. When a processor executes a call, it stacks parameters to be passed to the called procedures along with the return address. The called procedure can access the parameters from the stack. Upon return, the return parameters can also be placed on the stack. The entire set of parameters and the return addresses is called a stack frame.



In the figure above, the old frame pointer is needed if the number of parameters to be passed is variable.

## Appendix 12A: Little-, Big- and Bi-Endian

An annoying problem we face is how the bytes within a word and the bits within a byte are referenced and represented.

### Byte Ordering

We shall now look at the concept of endianness. Suppose we have a 32-bit hexadecimal value 12345678 that is stored in a 32-bit word in byte-addressable memory. The value consists of 4 bytes. The bytes can be stored in 4 consecutive memory locations, either as 12, 34, 56, 78 or as 78, 56, 34, 12. The first format has the most significant byte in the lowest numerical address, and is called big endian. The second format has the least significant byte in the lowest numerical byte address and is called little endian. For a single given multibyte scalar value, big endian and little endian are byte reversed mappings of each other.

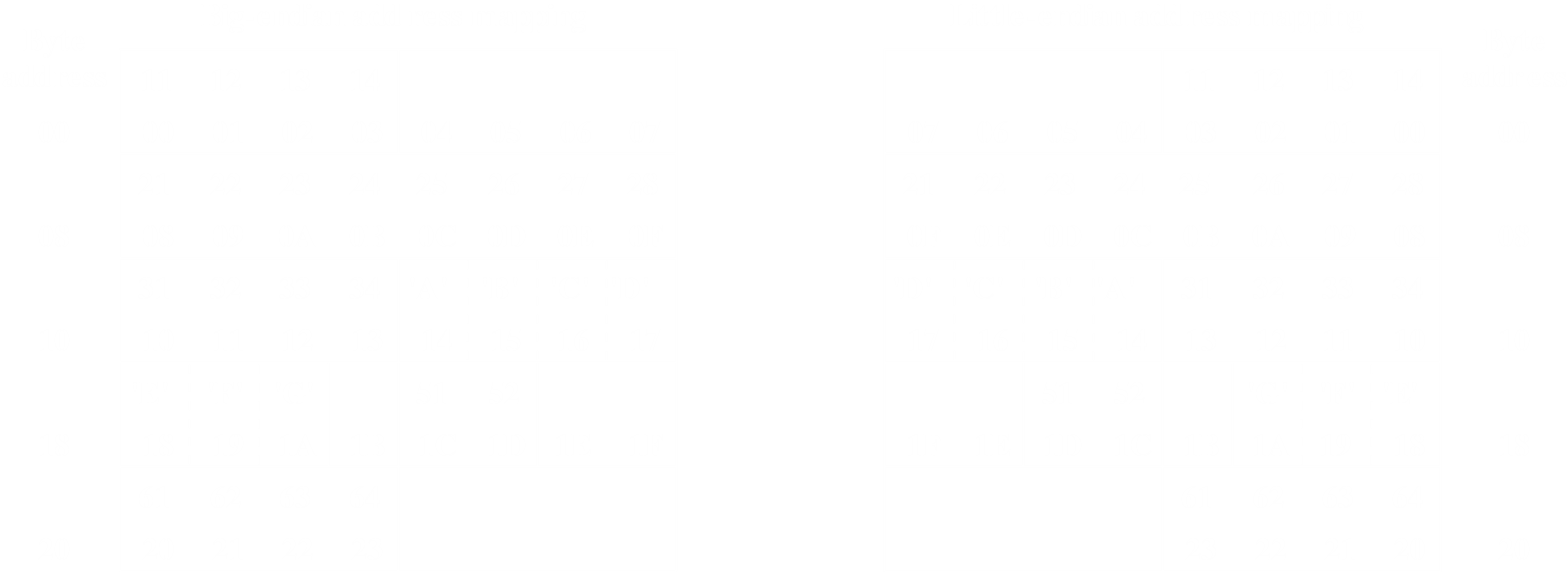
The concept of endianness arises when we need to treat a multi-byte entity like the one above as a single data item with a single address, even though it has smaller addressable units. Some machines are little-endian, while others are big-endian. As a result, there are problems when data is transferred between machines of different endian types and someone wants to manipulate the individual bytes within the multi-byte entity.

The problem of endianness differences does not go beyond an individual unit, even when aggregates such as files, data structures and arrays are composed of multiple data units, each with endianness. Consider the following structure.

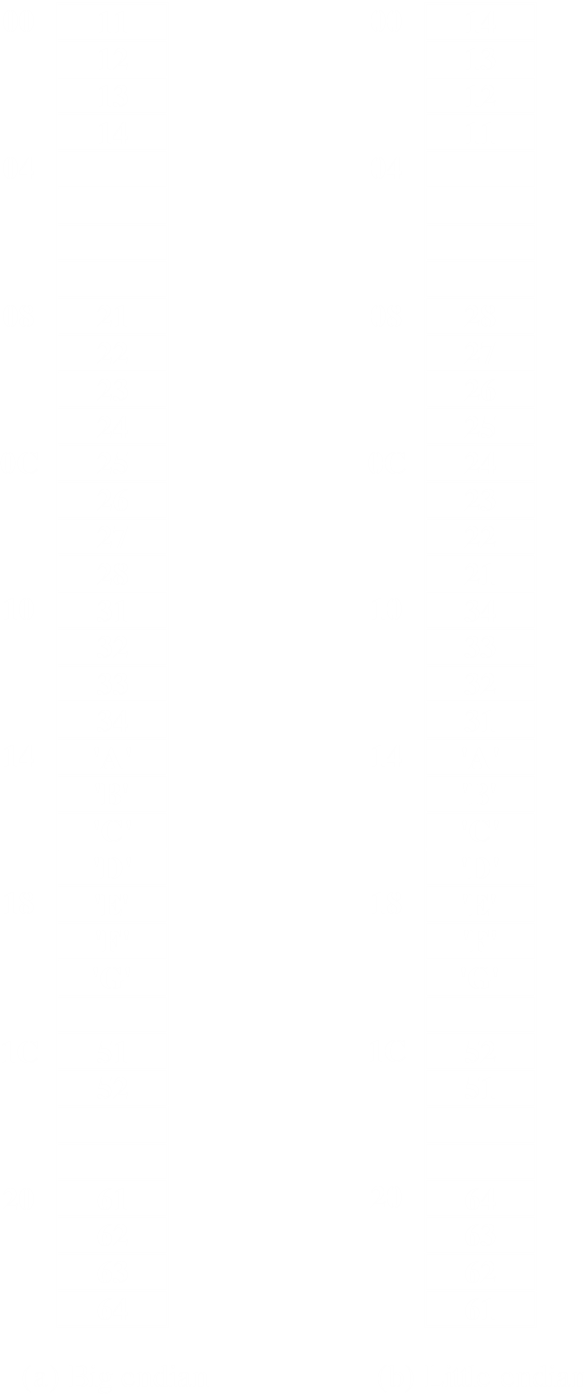
struct  
{  
 int a; *// 0x1112\_1314 word* int pad; *//* double b; *// 0x2122\_2324\_2526\_2728 doubleword* char\* c; *// 0x3132\_3334 word* char d[7]; *// 'A', 'B', 'C', 'D', 'E', 'F', 'G' byte array* short e; *// 0x5152 halfword* int f; *// 0x6162\_6364 word*} s;

C++

Here, the integer a has the hexadecimal values 11121314, the integer pad is uninitialized (blank), and so on. Now consider how the structure is stored in the two endian forms.



Notice that for each individual integer, double or other data unit, the order flips. However, this reversal does not go beyond that. Each piece of data is taking up the same group of byte addresses in either case. Looking at this as an array should make the effect clearer.



There is no clear favourite between the two forms of endianness, and each has its advantages. For the big-endian style, advantages include:

* **Character-String Sorting**: A big-endian processor is faster in comparing integer-aligned character strings; the integer ALU can compare multiple bytes in parallel.
* **Decimal/IRA Dumps**: All values can be printed left to right without causing confusion.
* **Consistent Order**: Big-endian processors store their integers and character strings in the same order (most significant byte first).

For the little-endian style, the advantages include:

* A big-endian processor has to perform addition when it converts a 32-bit integer address to a 16-bit integer address, to use the least significant bytes.
* It is easier to perform higher-precision arithmetic with the little-endian style; we do not have to find the least-significant byte and move backwards.

The differences are minor and the choice of which style to use is more about accommodating previous machines than anything else.

A bi-endian processor supports both big-endian and little-endian modes. Software developers can choose either mode when migrating operating systems and applications from other machines. The OS establishes the endian mode in which processes execute. Once a mode is selected, all further memory loads and stores are determined by the memory-addressing model of that mode.

To support his hardware feature, 2 bits are maintained in the machine state register (MSR) by the OS as part of the process state. One bit specified the endian mode in which the kernel runs, while the other specifies the processor’s current operating mode. Thus, mode can be changed on a per-process basis.

### Bit Ordering

In ordering the bits within a byte, we must face two questions.

1. Do we count the first bit as bit 0 or bit 1?
2. Do we use little endian or bit endian style?

These questions are answered in different ways in different machines, and even under different circumstances. Also, the endian style for bits within a byte might not be the same as for bytes within a multibyte scalar. The programmer needs to keep all of this in mind when manipulating individual bits.

Another area of concern is when the data is transferred over a bit-serial line. When a byte is transmitted, does the system transmit the most significant bit first or the least significant bit first? How incoming bits are handled will depend on this.