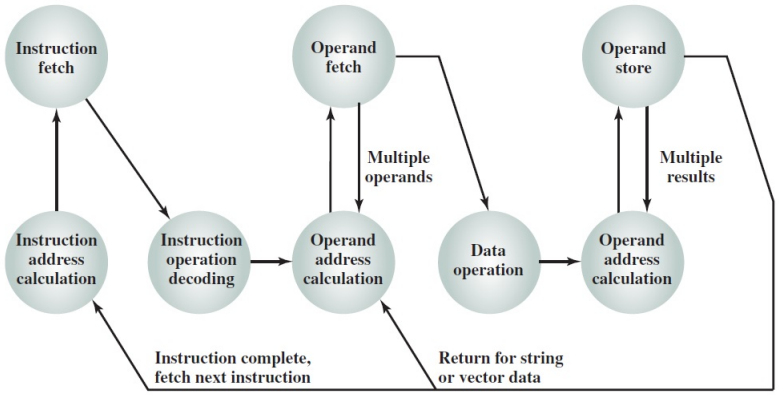
Lecture 4:
Instruction Sets: Characteristics & Functions

Machine Instruction Characteristics

The collection of different instructions known as *machine instructions* or *computer instructions* that the processor can execute is referred to as the processor's *instruction set*. Each instruction must contain the information required by the processor for execution. The state diagram form lecture 3 defines the elements of a machine instruction, which are as follows:

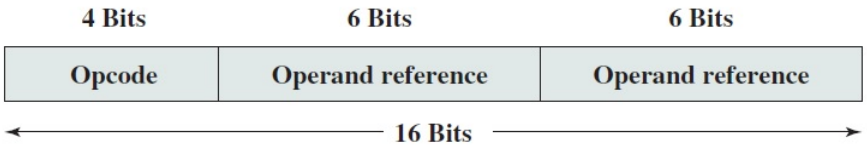
- **Operation code:** Specifies the operation to be performed. The operation is specified by a binary code, known as the operation code, or opcode.
- **Source operand reference:** The operation may involve one or more source operands, that is, operands that are inputs for the operation.
- **Result operand reference:** The operation may produce a result.
- **Next instruction reference:** This tells the processor where to fetch the next instruction after the execution of this instruction is complete.



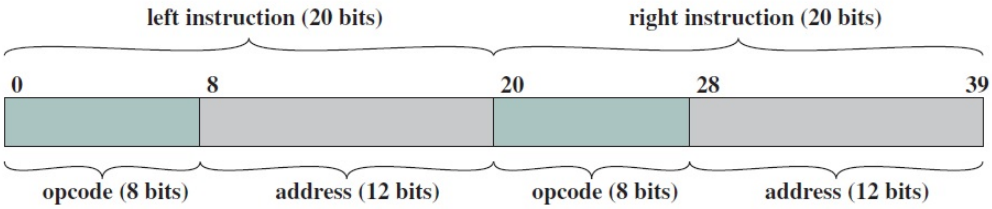
The address of the next instruction to be fetched can be either a real address ot a virtual address, and it can be explicitly referenced whenever it does not immediately follows the current instruction, which means the address most be supplied. Moreover instructions include references to operands. Source 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.

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent element of the instruction such as follows



or the IAS instruction



With most instruction sets, more than one format is used. During instructions 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.

Since it is difficult to deal with binary representations of machine instructions, it has become common practice to use a *symbolic representation* of the macine instructions. Opcodes are represented by abbreviations, called *mnenmonics*, 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 the register `R`, where `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.

It is possible to write a machine language program in symbolic form. Each symbolic opcode has a fixed binary representation, and the programmer specifes the location f each symbolic operand. Aftwerwards, a simple program would accept the symbolic inputs convert them to binary machine instructions. Although machine-language programming rarely exists now, symbolic machine language remains a useful tool for describing machine instructions.

Writing a statement in a high-level programming language will, typically, require several machine instructions. For instance, the statement

```
X = X + Y
```

can be accomplished by executing the instructions below where the content of `X` and `Y` are stored in the memory locations 513 and 514 respectively :

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.

Hence, the types of instrutions that a practical computer needs must allows the user to formulate any data processing task. In other words, every program wirtten in a high-level programming language must be able to be translated into machine language to be executed. Considering this, we can categorize instruction types as follows:

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

Arithmetic instructions provide computational capabilities for processing numerical 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 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.

One of the traditional ways of describing processor architecture, which is less significant now due to the increasing complexity of processor design, is in terms of the number of addresses cintained in each instruction. Determining the number of addresses needed in an instruction can vary. For instance, arithmetic and logic operations need the most operands. Likewise, you may want to include the address of the result of an operation. Furthermore, you may want to include the address of the next instruction.

From this reasoning, you may want an instruction to contain four address references: two source operands, one destination operand, and the address of the next instruction. In most architectures, many 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.

The following figure presents how typical one-, two-, and three-address instructions would compute the equation

$$Y = \frac{(A - B)}{(C + (D \times E))}$$

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

With three addresses, each instruction specifies two source operand locations and a destination operand location. Because none of the values of the operands are being modifed, we use a temporary location, `T`, to store some intermediate results. To compute the equation, four instructions are needed with three-address instruction. Although the program with three-address instructions is short, 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`, `B` carries out the calculatation `Y - B` and store 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.

With one-address instructions, 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 anf is used to store the result. In is also possible to make zero-address instructions. They are applicable to a special memory organization called a *stack*. A stack is a list-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.

The table below show the interpretations to be placed on instructions with zero, one, two and three addresses. In each case, it is assumed 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; hence, a less complex processor is required. However, the programs contain more total instructions, which in general results in longer execution times and longer, more complex programs. Also it is important to note that with one-address instructions, the programmer generally has available only one general-purpose register, the accumulator; whereas, with multiple-address instructions, it is common to have multiple general-purpose registers, which allows some operations to be performed solely on registers. 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 invloved in choosing the number of addresses per instruction are complicated by other factors, which results in most processor designs invloving a variety of instruction formats.

One of the most interesting, and most anlyzed, aspects of computer design is instruction set design. The design of an instructon 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. It is a programmer's means of controlling the processor. Thus, programmer reuirements must be considered in designing the instruction set.

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 the instructions, and their use.
- **Addressing:** The mode of 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.

Types Of Operands

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

- **Addresses**
- **Numbers**
- **Characters**
- **Logical Data**

There can also be machine operations that operate directly on a list or a string of characters.

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. Unlike mathematics numbers, computer numbers have a limited range. 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

Although all internal computer operations are binary, the human users deal with decimal numbers. Thus, it is necessary to convert from binary to decimal and vica versa. Hence, it is preferable to store and operate on the numbers in decimal form. The most common representation for this purpose is *packed decimal*. With packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way, with two digits stored per byte. Thus, `0 = 0000`, `1 = 0001`, `2 = 0010` and so on. Note that this is a rather inefficient code because only 10 of 16 possible 4-bit values are used. To form numbers, 4-bit codes are strung together, usually in multiples of 8 bits. Thus, the code for 246 is `0000 0010 0100 0110`. Although the code is less compact than a standard binary representation, it avoids the conversion overhead. Negative numbers can be represented by including a 4-bit sign digit at either the left or right end of a string of packed decimal digits. Standard sign values are `1100` for positive (+) and `1101` for negative (-). Many machines provide arithmetic instructions for performing operations directly on packed decimal numbers.

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 that are designed for binary data. Thus, a number of codes have been devised by which characters are represented by a sequence of bits similar to Morse code. 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). Each character in the code is represented by a unique 7-bit pattern; hence, there are 128 different characters. These characters are either printable characters, *control characters*, which are characters that deals with controlling the printing of characters on a page, or characters concerned with

communications procedures. IRA-encoding characters are almost always stored and transmitted using 8 bits per character where the eighth bit is either 0 or used as a parity but for error detection.

Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data; however, it is convenient to consider an n-bit unit as an item consisting of n 1-bit of data that are either 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 value 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 instance, if floating-point operations are implemented in software, we need to be able to shift significant bits in some operations. Another example is to convert IRA ro packed decimal by extracting the rightmost 4 bits of each byte.

Overall, data is stored in binary; hence, the type of data is determined by the operation being performed on it. While this is not normally the case in high-level languages, it is almost always the case with machine language.

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 categaorization is the following:

Three types of numerical data are common in computers:

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

The following table provides a discussion of the actions taken by the processor to execute a particular type of operation

Type	Operation Name	Description
Data transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand

Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT	(complement) Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end
Transfer of control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued
Input/output	Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

The most fundamental type of machne instruction is the data transfer instruction. The data tranfer 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 such as indicating the general location of an operand by either the specification of the opcode or the operand.

There are variants to indicate the amount of data to be transferred. Likewise, there are different instructions for register to register, register to memory, memory to register, and memory to memory transfers. 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.

Most machines provide the basic arithmetic operations. These are invariably provided for signed integer numbers, and often, they are provided for floating point and packed decimal numbers. Other possible operations include a variety of single-operand instructions; ofr 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. Furthermore, of course, the ALU portion of the processor perfroms the desired operations.

Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable unts, 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 in the table 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 logcal operations can be applied bitwise to $n$bit logical data unts. Thus, if two registers contain the data

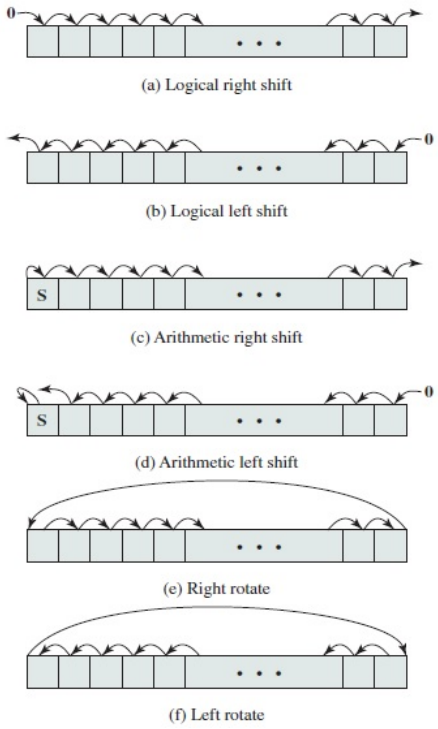
$$\texttt{(R1)} = \texttt{10100101}$$
$$\texttt{(R2)} = \texttt{00001111}$$
$$\texttt{then}$$
$$\texttt{(R1) AND (R2)} = \texttt{00000101}$$

where the notation `(X)` means the contents of the location `X`. Thus, the `AND` operation can be used as a *mask* that selects certain bits in a word and zeroes out the remaining bits. An another example is as follows

$$\texttt{(R1)} = \texttt{10100101}$$
$$\texttt{(R2)} = \texttt{11111111}$$
$$\texttt{then}$$
$$\texttt{(R1) XOR (R2)} = \texttt{01011010}$$

Here given a word of all 1s, the `XOR` operation will inverts all the bits in the other word (*ones complement*).

In addition to bitwise logical operations, most machines provide a variety of shifting and rotating functions. The most basic operations are illustrated in the 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 usedful 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. For instance, 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 data bus.

The preceeding steps result in sending the left-hand character. To send the right-hand 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 but 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 corresponds 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. Because of the potential for overflow, many processors do not include this instruction.

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).

As with arithmetic operations, logical operations involve ALU activity and may involve data transfer operations. The table below gives examples of all of the shift and rotate operations

Input	Operation	Result
10100110	Logical right shift (3 bits)	00010100
10100110	Logical left shift (3 bits)	00110000
10100110	Arithmetic right shift (3 bits)	11110100
10100110	Arithmetic left shift (3 bits)	10110000
10100110	Right rotate (3 bits)	11010100
10100110	Left rotate (3 bits)	00110101

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

There are a variety of approaces taken to handle input/output instructions 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.

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; Other examples are an instruction to read or modify a storage protection key, or access to process control blocks in a multiprogramming system.

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.

The most common transfer-of-control operations found in instruction sets are *branch*, *skip* and *procedure call* instructions.