**Computer Organization**

**UNIT-II**

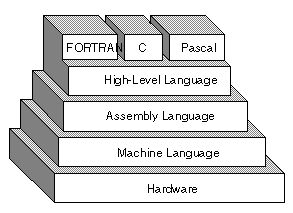
**Programming the basic control:**

**Machine language:**

Machine language is the lowest-level [programming language](http://www.webopedia.com/TERM/P/programming_language.html) (except for computers that utilize programmable [microcode](http://www.webopedia.com/TERM/M/microcode.html)). Machine languages are the only [languages](http://www.webopedia.com/TERM/L/language.html) understood by [computers](http://www.webopedia.com/TERM/C/computer.html).

## Why Humans Don't Use Machine Language?

While easily understood by computers, machine languages are almost impossible for humans to use because they consist entirely of numbers. [Programmers](http://www.webopedia.com/TERM/P/programmer.html), therefore, use either a high-level programming language or an [assembly language](http://www.webopedia.com/TERM/A/assembly_language.html). An assembly language contains the same [instructions](http://www.webopedia.com/TERM/I/instruction.html) as a machine language, but the instructions and [variables](http://www.webopedia.com/TERM/V/variable.html) have [names](http://www.webopedia.com/TERM/N/name.html) instead of being just numbers.



**Assembly language:**

Assembly language is a low-level programming language for a computer or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.

**Sample code:**

section .text

global \_start ;must be declared for linker (ld)

\_start: ;tells linker entry point

mov edx,len ;message length

mov ecx,msg ;message to write

mov ebx,1 ;file descriptor (stdout)

mov eax,4 ;system call number (sys\_write)

int 0x80 ;call kernel

mov eax,1 ;system call number (sys\_exit)

int 0x80 ;call kernel

section .data

msg db 'Hello, world!', 0xa ;string to be printed

len equ $ - msg ;length of the string

**Assembler:**

An assembler is a [program](http://searchsoftwarequality.techtarget.com/definition/program) that takes basic computer [instruction](http://searchcio-midmarket.techtarget.com/definition/instruction)s and converts them into a pattern of [bit](http://searchcio-midmarket.techtarget.com/definition/bit)s that the computer's [processor](http://searchcio-midmarket.techtarget.com/definition/processor) can use to perform its basic operations. Some people call these instructions assembler language and others use the term *assembly language*.

An assembler is a [program](http://searchsoftwarequality.techtarget.com/definition/program) that takes basic computer [instruction](http://searchcio-midmarket.techtarget.com/definition/instruction)s and converts them into a pattern of [bit](http://searchcio-midmarket.techtarget.com/definition/bit)s that the computer's [processor](http://searchcio-midmarket.techtarget.com/definition/processor) can use to perform its basic operations. Some people call these instructions assembler language and others use the term *assembly language*.

Here's how it works:

* Most computers come with a specified set of very basic instructions that correspond to the basic machine operations that the computer can perform. For example, a "Load" instruction causes the processor to move a string of bits from a location in the processor's [memory](http://searchmobilecomputing.techtarget.com/definition/memory) to a special holding place called a [register](http://whatis.techtarget.com/definition/register). Assuming the processor has at least eight registers, each numbered, the following instruction would move the value (string of bits of a certain length) at memory location 3000 into the holding place called register 8:

L 8,3000

* The programmer can write a program using a sequence of these assembler instructions.
* This sequence of assembler instructions, known as the [source code](http://searchsoa.techtarget.com/definition/source-code) or source program, is then specified to the assembler program when that program is started.
* The assembler program takes each program statement in the source program and generates a corresponding bit stream or pattern (a series of 0's and 1's of a given length).
* The output of the assembler program is called the [object code](http://searchcio-midmarket.techtarget.com/definition/object-code) or object program relative to the input source program. The sequence of 0's and 1's that constitute the object program is sometimes called [machine code](http://searchcio-midmarket.techtarget.com/definition/machine-code).
* The object program can then be run (or executed) whenever desired.

In the earliest computers, programmers actually wrote programs in machine code, but assembler languages or instruction sets were soon developed to speed up programming. Today, assembler programming is used only where very efficient control over processor operations is needed. It requires knowledge of a particular computer's instruction set, however. Historically, most programs have been written in "higher-level" languages such as COBOL, FORTRAN, PL/I, and C. These languages are easier to learn and faster to write programs with than assembler language. The program that processes the source code written in these languages is called a [compiler](http://whatis.techtarget.com/definition/compiler). Like the assembler, a compiler takes higher-level language statements and reduces them to machine code.

A newer idea in program preparation and portability is the concept of a [virtual machine](http://searchservervirtualization.techtarget.com/definition/virtual-machine). For example, using the [Java](http://searchsoa.techtarget.com/definition/Java) programming language, language statements are compiled into a generic form of machine language known as [bytecode](http://searchcio-midmarket.techtarget.com/definition/bytecode) that can be run by a virtual machine, a kind of theoretical machine that approximates most computer operations. The bytecode can then be sent to any computer platform that has previously downloaded or built in the Java virtual machine. The virtual machine is aware of the specific instruction lengths and other particularities of the platform and ensures that the Java bytecode can run.

**Programming –Arithmetic & logic operations:**

An operator in a programming language is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce final result. This chapter will explain the concept of **operators** and it will take you through the important arithmetic and relational operators available in C, Java, and Python.

## Arithmetic Operators:

Computer programs are widely used for mathematical calculations. We can write a computer program which can do simple calculation like adding two numbers (2 + 3) and we can also write a program, which can solve a complex equation like P(x) = x4 + 7x3 - 5x + 9. If you have been even a poor student, you must be aware that in first expression 2 and 3 are operands and + is an operator. Similar concepts exist in Computer Programming.

Take a look at the following two examples −

2 + 3

P(x) = x4 + 7x3 - 5x + 9.

These two statements are called arithmetic expressions in a programming language and **plus**, **minus** used in these expressions are called arithmetic operators and the values used in these expressions like 2, 3 and x, etc., are called operands. In their simplest form, such expressions produce numerical results.

Similarly, a programming language provides various arithmetic operators. The following table lists down a few of the important arithmetic operators available in C programming language. Assume variable A holds 10 and variable B holds 20, then –

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| \* | Multiplies both operands | A \* B will give 200 |
| / | Divides numerator by de-numerator | B / A will give 2 |
| % | This gives remainder of an integer division | B % A will give 0 |

## Relational Operators

Consider a situation where we create two variables and assign them some values as follows −

A = 20

B = 10

Here, it is obvious that variable A is greater than B in values. So, we need the help of some symbols to write such expressions which are called relational expressions. If we use C programming language, then it will be written as follows −

(A > B)

Here, we used a symbol > and it is called a relational operator and in their simplest form, they produce Boolean results which means the result will be either true or false. Similarly, a programming language provides various relational operators. The following table lists down a few of the important relational operators available in C programming language. Assume variable **A** holds 10 and variable **B** holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

Here, we will show you one example of C Programming which makes use of **if conditional statement**. Though this statement will be discussed later in a separate chapter, but in short, we use **if statement** to check a condition and if the condition is true, then the body of **if statement** is executed, otherwise the body of **if statement** is skipped.

## Logical Operators

Logical operators are very important in any programming language and they help us take decisions based on certain conditions. Suppose we want to combine the result of two conditions, then logical AND and OR logical operators help us in producing the final result.

The following table shows all the logical operators supported by the C language. Assume variable **A** holds 1 and variable **B** holds 0, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

**Subroutines:**

The subroutine is an important part of any computer systems architecture. A subroutine is a group of instructions that usually performs one task, it is a reusable section of the software that is stored in memory once, but used as often as necessary. This saves memory space and makes it easier to develop software. The only disadvantage of a subroutine is that it takes the computer a small amount of time to link to the subroutine and return from it.

In the 6800 microprocessor a subroutine can be called by a Jump or Branch to subroutine instruction and when the subroutine is completed the Return from Subroutine (RTS) instruction returns from the subroutine.

The Stack stores the return address whenever a subroutine is called during the execution of a programme. The Jump or Branch to subroutine instruction pushes the address of the next instruction following it onto the Stack. The RTS instruction removes the address from the Stack so the programme returns to the next instruction following the subroutine call.

In the PIC 16F84 the CALL instruction is used when Calling a subroutine,  saving the current program counter so that the Return operation knows where to restore the program counter. This is accomplished automatically (as part of the CALL instruction) pushing the return address onto the Stack, then when a Return instruction is executed, this address is popped off the stack and put into the program counter.

In [computer programming](https://en.wikipedia.org/wiki/Computer_programming), a **subroutine** is a sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular [task](https://en.wikipedia.org/wiki/Task_%28computing%29) should be performed. Subprograms may be defined within programs, or separately in [libraries](https://en.wikipedia.org/wiki/Library_%28computer_science%29) that can be used by multiple programs. In different programming languages, a subroutine may be called a **procedure**, a **function**, a **routine**, a [method](https://en.wikipedia.org/wiki/Method_%28computing%29), or a **subprogram**. The generic term **callable unit** is sometimes used.

The name *subprogram* suggests a subroutine behaves in much the same way as a computer program that is used as one step in a larger program or another subprogram. A subroutine is often coded so that it can be started (called) several times and from several places during one execution of the program, including from other subroutines, and then branch back (*return*) to the next instruction after the *call*, once the subroutine's task is done. Subroutines are a powerful [programming](https://en.wikipedia.org/wiki/Programming_language) tool, and the [syntax](https://en.wikipedia.org/wiki/Syntax_%28programming_languages%29) of many [programming languages](https://en.wikipedia.org/wiki/Programming_language) includes support for writing and using them. Judicious use of subroutines (for example, through the [structured programming](https://en.wikipedia.org/wiki/Structured_programming) approach) will often substantially reduce the cost of developing and maintaining a large program, while increasing its quality and reliability. Subroutines, often collected into [libraries](https://en.wikipedia.org/wiki/Library_%28computing%29), are an important mechanism for sharing and trading software. The discipline of [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) is based on [objects](https://en.wikipedia.org/wiki/Object_%28computer_science%29) and [methods](https://en.wikipedia.org/wiki/Method_%28computer_programming%29) (which are subroutines attached to these objects or object [classes](https://en.wikipedia.org/wiki/Class_%28computer_programming%29)).

In the [compiling](https://en.wikipedia.org/wiki/Compiler) method called [threaded code](https://en.wikipedia.org/wiki/Threaded_code), the executable program is basically a sequence of subroutine calls.

**The advantages of breaking a program into subroutines include:**

* [Decomposing](https://en.wikipedia.org/wiki/Decomposition_%28computer_science%29) a complex programming task into simpler steps: this is one of the two main tools of [structured programming](https://en.wikipedia.org/wiki/Structured_programming), along with [data structures](https://en.wikipedia.org/wiki/Data_structure)
* Reducing [duplicate code](https://en.wikipedia.org/wiki/Duplicate_code) within a program
* Enabling [reuse of code](https://en.wikipedia.org/wiki/Code_reuse) across multiple programs
* Dividing a large programming task among various programmers, or various stages of a project
* [Hiding implementation details](https://en.wikipedia.org/wiki/Information_hiding) from users of the subroutine
* Improving [traceability](https://en.wikipedia.org/wiki/Traceability#Software) (i.e. most languages offer ways to obtain the call trace which includes the names of the involved subroutines and perhaps even more information such as file names and line numbers); by not decomposing the code into subroutines, debugging would be impaired severely.

## Disadvantages

Invoking a subroutine (versus using in-line code) imposes some [computational overhead](https://en.wikipedia.org/wiki/Computational_overhead) in the call mechanism.

The subroutine typically requires standard [housekeeping](https://en.wikipedia.org/wiki/Housekeeping_%28computing%29) code – both at entry to, and exit from, the function ([function prologue and epilogue](https://en.wikipedia.org/wiki/Function_prologue) – usually saving [general purpose registers](https://en.wikipedia.org/wiki/General_purpose_register) and return address as a minimum).

**Micro programmed control:**

## Control Memory

### Introduction:-

Control  memory  is  a  random  access  memory(RAM) consisting of addressable storage registers. It is primarily used in mini and mainframe computers. It is used as a temporary storage for data. Access to control memory data requires less time than to main memory; this speeds up CPU operation by reducing the numberof memory references for data storage and retrieval. Access  is  performed  as  part  of  a  control  section sequence while the master clock oscillator is running. The control memory addresses are divided into two groups: a task mode and an executive   (interrupt)  mode. Addressing words stored in control memory is via the address select logic for each of the register groups. There can be up to five register groups in control memory. These groups select a register for fetching data for programmed   CPU operation or for maintenance console or equivalent display or storage of data via a maintenance   console   or   equivalent.   During programmed   CPU  operations,  these  registers  are accessed directly by the CPU logic. Data routing circuits  are  used  by  control  memory  to  interconnect  the registers used in control memory. Some  of  the  registers  contained  in  a  control memory that operate in the task and the executive modes include the following: Accumulators ,Indexes, Monitor clock status indicating registers Interrupt data registers.

**Control memory**

Control data register reregister register

Control address reigister

Control memory

Sequencer

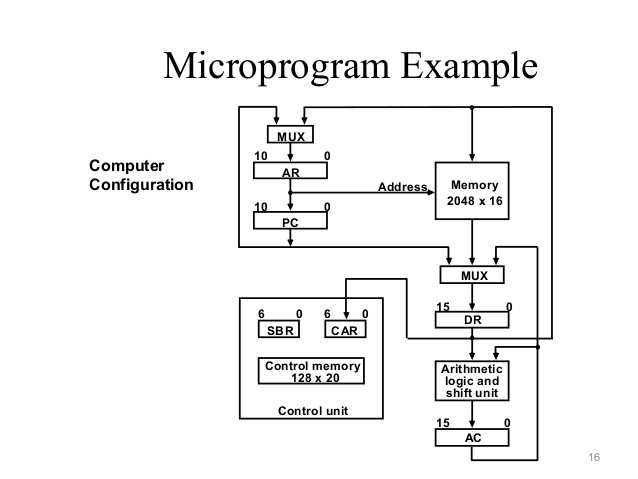
External inputs

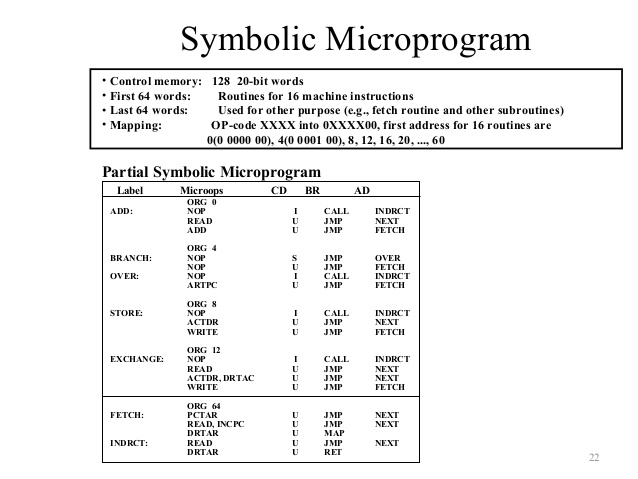
Control word

Next address information

• The control unit in a digital computer initiates sequences of microoperations   
• The complexity of the digital system is derived form the number of sequences that   
are performed   
• When the control signals are generated by hardware, it is hardwired   
• In a bus-oriented system, the control signals that specify microoperations are   
groups of bits that select the paths in multiplexers, decoders, and ALUs.   
  
• The control unit initiates a series of sequential steps of microoperations   
• The control variables can be represented by a string of 1’s and 0’s called a control   
word   
• A microprogrammed control unit is a control unit whose binary control variables   
are stored in memory   
• A sequence of microinstructions constitutes a microprogram   
• The control memory can be a read-only memory   
• Dynamic microprogramming permits a microprogram to be loaded and uses a   
writable control memory   
  
• A computer with a microprogrammed control unit will have two separate   
memories: a main memory and a control memory   
• The microprogram consists of microinstructions that specify various internal   
control signals for execution of register microoperations   
• These microinstructions generate the microoperations to:   
o fetch the instruction from main memory   
o evaluate the effective address   
o execute the operation   
o return control to the fetch phase for the next instruction   
  
• The control memory address register specifies the address of the microinstruction   
• The control data register holds the microinstruction read from memory   
• The microinstruction contains a control word that specifies one or more   
microoperations for the data processor   
  
• The location for the next microinstruction may, or may not be the next in   
sequence   
• Some bits of the present microinstruction control the generation of the address of   
the next microinstruction   
• The next address may also be a function of external input conditions   
• While the microoperations are being executed, the next address is computed in the   
next address generator circuit (sequencer) and then transferred into the CAR to   
read the next microinstructions   
• Typical functions of a sequencer are: o incrementing the CAR by one   
o loading into the CAR and address from control memory   
o transferring an external address   
o loading an initial address to start the control operations   
• A clock is applied to the CAR and the control word and next-address information   
are taken directly from the control memory   
• The address value is the input for the ROM and the control work is the output   
• No read signal is required for the ROM as in a RAM   
  
• **The** **main advantage of the microprogrammed control** is that once the hardware   
configuration is established, there should be no need for h/w or wiring changes   
• To establish a different control sequence, specify a different set of   
microinstructions for control memory .

**Examples of microprogram:**





**Design of Control Unit**

The **control unit** (CU) is a component of a computer's [central processing unit](https://en.wikipedia.org/wiki/Central_processing_unit) (CPU) that directs the operation of the processor. It tells the computer's memory, arithmetic/logic unit and input and output devices how to respond to a program's instructions.

It directs the operation of the other units by providing timing and control signals. Most computer resources are managed by the CU. It directs the flow of data between the CPU and the other devices. John von Neumann included the control unit as part of the [von Neumann architecture](https://en.wikipedia.org/wiki/Von_Neumann_architecture). In modern computer designs, the control unit is typically an internal part of the [CPU](https://en.wikipedia.org/wiki/Central_processing_unit) with its overall role and operation unchanged since its introduction.

## Functions of the control unit

The Control Unit (CU) is digital circuitry contained within the processor that coordinates the sequence of data movements into, out of, and between a processor's many sub-units. The result of these routed data movements through various digital circuits (sub-units) within the processor produces the manipulated data expected by a software instruction (loaded earlier, likely from memory). In a way, the CU is the "brain within the brain", as it controls (conducts) data flow inside the processor and additionally provides several external control signals to the rest of the computer to further direct data and instructions to/from processor external destinations (i.e. memory).

Examples of devices that require a CU are CPUs and graphics processing units (GPUs). The CU receives external instructions or commands which it converts into a sequence of control signals that the CU applies to the data path to implement a sequence of [register-transfer level](https://en.wikipedia.org/wiki/Register-transfer_level) operations.

More precisely, the Control Unit (CU) is generally a sizable collection of complex digital circuitry interconnecting and controlling the many execution units (i.e. ALU, data buffers, registers) contained within a CPU. The CU is normally the first CPU unit to accept from an externally stored computer program, a single instruction (based on the CPU's [instruction set](https://en.wikipedia.org/wiki/Instruction_set)). The CU then decodes this individual instruction into several sequential steps (fetching addresses/data from registers/ memory, managing execution [i.e. data sent to the ALU or I/O], and storing the resulting data back into registers/memory) that controls and coordinates the CPU's inner works to properly manipulate the data. The design of these sequential steps are based on the needs of each instruction and can range in number of steps, the order of execution, and which units are enabled. Thus by only using a program of set instructions in memory, the CU will configure all the CPU's data flows as needed to manipulate the data correctly between instructions. This results in a computer that could run a complete program and requiring no human intervention to make hardware changes between instructions (as had to be done when using only [punch cards](https://en.wikipedia.org/wiki/Plugboard#Unit_record_equipment) for computations before stored programmed computers with CUs where invented). These detailed steps from the CU dictate which of the CPU's interconnecting hardware control signals to enable/disable or which CPU units are selected/de-selected and the unit's proper order of execution as required by the instruction's operation to produce the desired manipulated data. Additionally, the CU's orderly hardware coordination properly sequences these control signals then configures the many hardware units comprising the CPU, directing how data should also be moved, changed, and stored outside the CPU (i.e. memory) according to the instruction's objective. Depending on the type of instruction entering the CU, the order and number of sequential steps produced by the CU could vary the selection and configuration of which parts of the CPU's hardware are utilized to achieve the instruction's objective (mainly moving, storing, and modifying data within the CPU). This one feature, that efficiently uses just software instructions to control/select/configure a computer's CPU hardware (via the CU) and eventually manipulates a program's data, is a significant reason most modern computers are flexible and universal when running various programs. As compared to some 1930s or 1940s computers without a proper CU, they often required rewiring their hardware when changing programs. This CU instruction decode process is then repeated when the Program Counter is incremented to the next stored program address and the new instruction enters the CU from that address, and so on till the programs end.

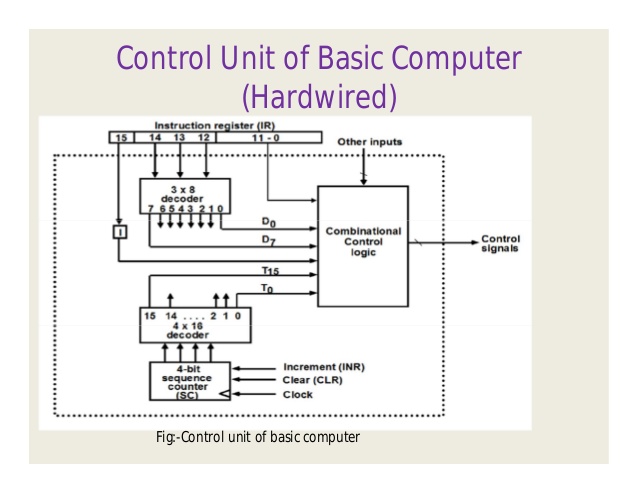
Other more advanced forms of Control Units manage the translation of instructions (but not the data containing portion) into several micro-instructions and the CU manages the scheduling of the micro-instructions between the selected execution units to which the data is then channeled and changed according to the execution unit's function (i.e., ALU contains several functions). On some processors, the Control Unit may be further broken down into additional units, such as an [instruction unit](https://en.wikipedia.org/wiki/Instruction_unit) or scheduling unit to handle scheduling, or a retirement unit to deal with results coming from the [instruction pipeline](https://en.wikipedia.org/wiki/Instruction_pipeline). Again, the Control Unit orchestrates the main functions of the CPU: carrying out stored instructions in the software program then directing the flow of data throughout the computer based upon these instructions (roughly likened to how traffic lights will systematically control the flow of cars [containing data] to different locations within the traffic grid [CPU] until it parks at the desired parking spot [memory address/register]. The car occupants [data] then go into the building [execution unit] and comes back changed in some way then get back into the car and returns to another location via the controlled traffic grid).

To execute an instruction, the control unit of the CPU must generate the required control signal in the proper sequence. As for example, during the fetch phase, CPU has to generate PCout signal along with other required signal in the first clock pulse. In the second clock pulse CPU has to generate PCin signal along with other required signals. So, during fetch phase, the proper sequence for generating the signal to retrieve from and store to PC is PCout  and   PCin.

To generate the control signal in proper sequence, a wide variety of techniques exist. Most of these techniques, however, fall into one of the two categories,

* + - 1. Hardwired Control
      2. Microprogrammed Control.

**Hardwired Control:**



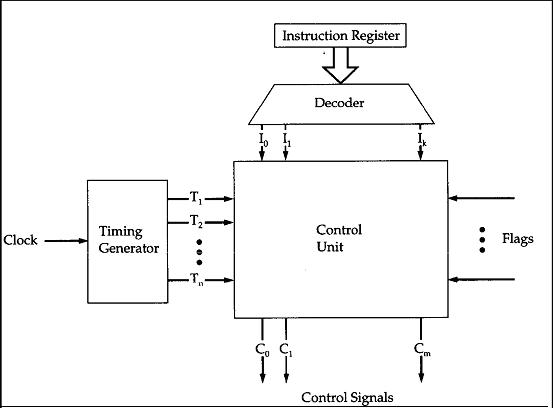
**Hardwired control units** are implemented through use of [sequential logic](https://en.wikipedia.org/wiki/Sequential_logic) units, featuring a finite number of gates that can generate specific results based on the instructions that were used to invoke those responses. Hardwired control units are generally faster than microprogrammed designs.

Their design uses a fixed architecture—it requires changes in the wiring if the [instruction set](https://en.wikipedia.org/wiki/Instruction_set) is modified or changed. This architecture is preferred in [reduced instruction set computers](https://en.wikipedia.org/wiki/Reduced_instruction_set_computing) (RISC) as they use a simpler instruction set.

A controller that uses this approach can operate at high speed; however, it has little flexibility, and the complexity of the instruction set it can implement is limited.

The hardwired approach has become less popular as computers have evolved. Previously, control units for CPUs used ad-hoc logic, and they were difficult to design.

**Microprogrammed Control:**



The idea of microprogramming was introduced by [Maurice Wilkes](https://en.wikipedia.org/wiki/Maurice_Wilkes) in 1951 as an intermediate level to execute computer program instructions. Microprograms were organized as a sequence of *microinstructions* and stored in special control memory. The algorithm for the microprogram control unit is usually specified by [flowchart](https://en.wikipedia.org/wiki/Flowchart) description.The main advantage of the microprogram control unit is the simplicity of its structure. Outputs of the controller are organized in microinstructions and they can be easily replaced.

**Address sequencing:**

Microinstructions are stored in control memory in groups, with each group specifying routine. Each computer instruction has its own microprogram routine specifying routine. Each computer instruction has its own microprogram routine.

The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another. To appreciate the address sequencing in a microprogram control unit, let us enumerate the steps that the control must undergo during the execution of a single computer instruction. An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions. At the end of the fetch routine, the instruction is in the instruction register of the computer. The control memory next must go through the routine that determines the effective address of the operand. A machine instruction may have bits that specify various addressing modes, such as **indirect address and index registers**. The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.