Design of Control Unit

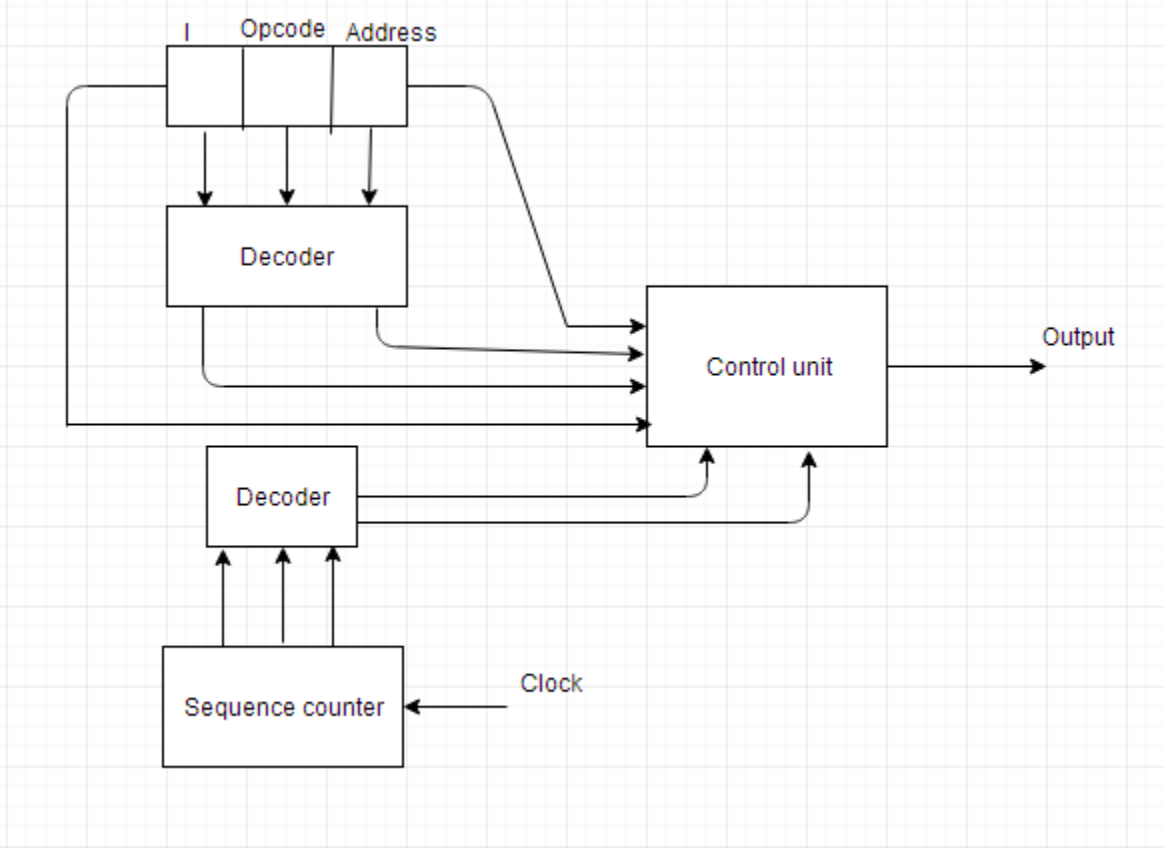
Control unit generates timing and control signals for the operations of the computer. The control unit communicates with ALU and main memory. It also controls the transmission between processor, memory and the various peripherals. It also instructs the ALU which operation has to be performed on data.

Control unit can be designed by two methods which are given below:

Hardwired Control Unit

It is implemented with the help of gates, flip flops, decoders etc. in the hardware. The inputs to control unit are the instruction register, flags, timing signals etc. This organization can be very complicated if we have to make the control unit large.

If the design has to be modified or changed, all the combinational circuits have to be modified which is a very difficult task.



Microprogrammed Control Unit

It is implemented by using programming approach. A sequence of micro operations is carried out by executing a program consisting of micro-instructions. In this organization any modifications or changes can be done by updating the micro program in the control memory by the programmer.

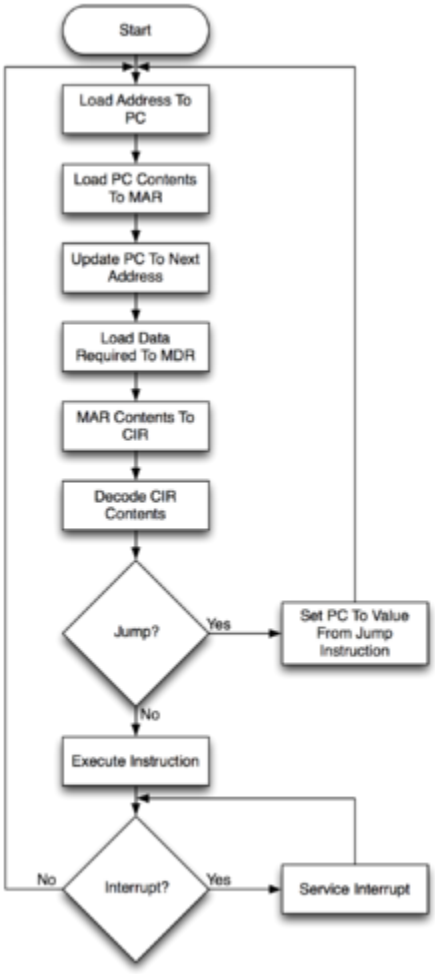
Difference between Hardwired Control and Microprogrammed Control

Hardwired Control	Microprogrammed Control
Technology is circuit based.	Technology is software based.
It is implemented through flip-flops, gates, decoders etc.	Microinstructions generate signals to control the execution of instructions.
Fixed instruction format.	Variable instruction format (16-64 bits per instruction).
Instructions are register based.	Instructions are not register based.
ROM is not used.	ROM is used.
It is used in RISC.	It is used in CISC.
Faster decoding.	Slower decoding.
Difficult to modify.	Easily modified.
Chip area is less.	Chip area is large.

Instruction cycle

The **instruction cycle** (also known as the **fetch–decode–execute cycle** or the **fetch-execute cycle**) is the basic operational process of a computer system. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the

instruction describes, and then carries out those actions. This cycle is repeated continuously by a computer's central processing unit (CPU), from boot-up until the computer has shut down.



A diagram of the instruction cycle.

In simpler CPUs the instruction cycle is executed sequentially, each instruction being processed before the next one is started. In most modern CPUs the instruction cycles are instead executed concurrently, and often in parallel, through an instruction pipeline: the next instruction starts being processed before the previous instruction has finished, which is possible because the cycle is broken up into separate steps.

Components

Program counter (PC)
: An incrementing counter that keeps track of the memory address of the instruction that is to be executed next or in other words, holds the address of the instruction to be executed next.

Memory address register (MAR)
: Holds the address of a block of memory for reading from or writing to.

Memory data register (MDR)
: A two-way register that holds data fetched from memory (and ready for the CPU to process) or data waiting to be stored in memory. (This is also known as the memory buffer register (MBR).)

Instruction register (IR)
: A temporary holding ground for the instruction that has just been fetched from memory.

Control unit (CU)
: Decodes the program instruction in the IR, selecting machine resources, such as a data source register and a particular arithmetic operation, and coordinates activation of those resources.

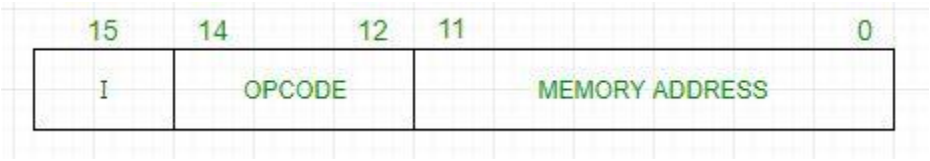
Arithmetic logic unit (ALU)
: Performs mathematical and logical operations.

Floating-point unit (FPU)
: Performs floating-point operations.

Memory reference instruction:

The basic computer has 16 bit instruction register (IR) which can denote either memory reference or register reference or input-output instruction.

1. **Memory Reference –** These instructions refer to memory address as an operand. The other operand is always accumulator. Specifies 12 bit address, 3 bit opcode (other than 111) and 1 bit addressing mode for direct and indirect addressing.

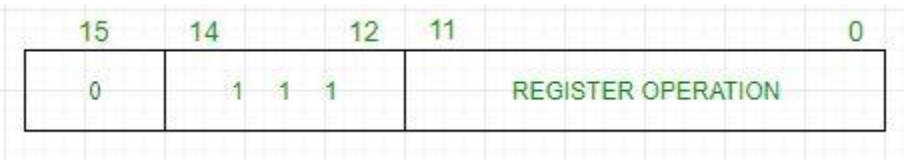


Example –
IR register contains = 0001XXXXXXXXXXX, i.e. ADD after fetching and decoding of instruction we find out that it is a memory reference instruction for ADD operation.

```
Hence, DR <- M[AR]
AC <- AC+ DR, SC <- 0
```

2. **Register Reference –** These instructions perform operations on registers rather than memory addresses. The IR(14-12) is 111 (differentiates it from memory reference) and

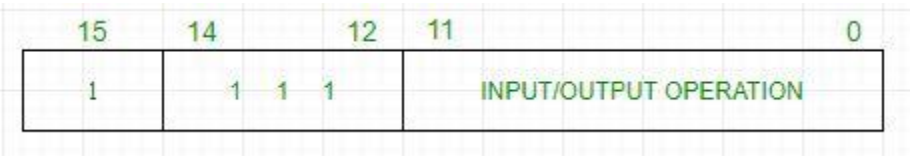
IR(15) is 0 (differentiates it from input/output instructions). The rest 12 bits specify register operation.



Example –
IR register contains = 0111001000000000, i.e. CMA after fetch and decode cycle we find out that it is a register reference instruction for complement accumulator.

```
Hence, AC <- ~AC
```

3. **Input/Output –** These instructions are for communication between computer and outside environment. The IR(14-12) is 111 (differentiates it from memory reference) and IR(15) is 1 (differentiates it from register reference instructions). The rest 12 bits specify I/O operation.



Example –
IR register contains = 1111100000000000, i.e. INP after fetch and decode cycle we find out that it is an input/output instruction for inputing character. Hence, INPUT character from peripheral device.

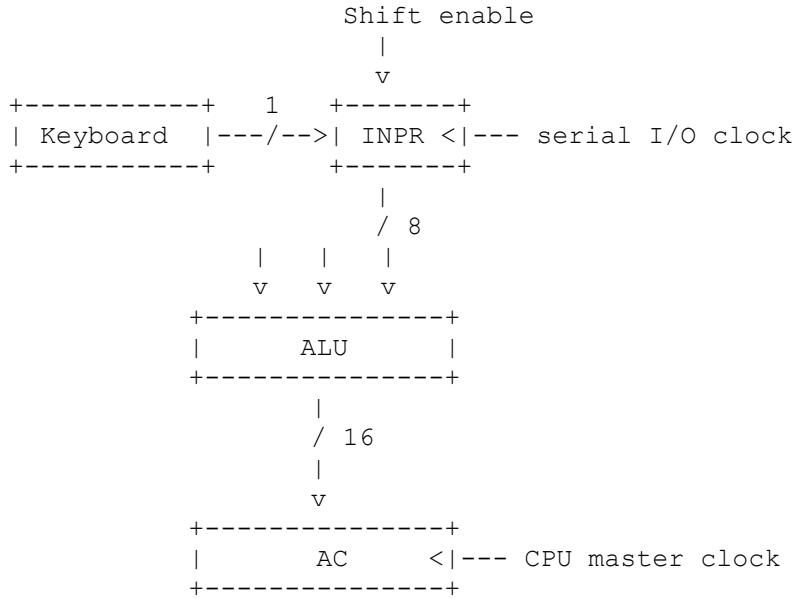
The set of instructions incorporated in16 bit IR register are:

1. Arithmetic, logical and shift instructions (and, add, complement, circulate left, right, etc)
2. To move information to and from memory (store the accumulator, load the accumulator)
3. Program control instructions with status conditions (branch, skip)
4. Input output instructions (input character, output character)

Input-Output and Interrupt

The Basic Computer I/O consists of a simple terminal with a keyboard and a printer/monitor.

The keyboard is connected serially (1 data wire) to the INPR register. INPR is a shift register capable of shifting in external data from the keyboard one bit at a time. INPR outputs are connected in parallel to the ALU.



I/O Operations

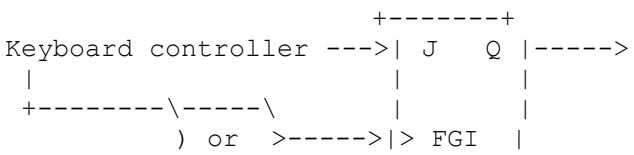
Since input and output devices are not under the full control of the CPU (I/O events are asynchronous), the CPU must somehow be told when an input device has new input ready to send, and an output device is ready to receive more output.

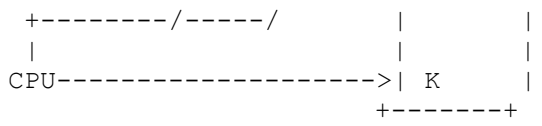
The FGI flip-flop is set to 1 after a new character is shifted into INPR. This is done by the I/O interface, not by the control unit. This is an example of an asynchronous input event (not synchronized with or controlled by the CPU).

The FGI flip-flop must be cleared after transferring the INPR to AC. This must be done as a microoperation controlled by the CU, so we must include it in the CU design.

The FGO flip-flop is set to 1 by the I/O interface after the terminal has finished displaying the last character sent. It must be cleared by the CPU after transferring a character into OUTR.

Since the keyboard controller only sets FGI and the CPU only clears it, a JK flip-flop is convenient:





How do we control the CK input on the FGI flip-flop? (Assume leading-edge triggering.)

There are two common methods for detecting when I/O devices are ready, namely *software polling* and *interrupts*. These two methods are discussed in the following sections.

Interrupts

To alleviate the problems of software polling, a hardware solution is needed.

Analogies to software polling in daily life tend to look rather silly. For example, imagine a teacher is analogous to a CPU, and the students are I/O devices. The students are working asynchronously, as the teacher walks around the room constantly asking each individual student "are you done yet?".

What would be a better approach?

With interrupts, the running program is not responsible for checking the status of I/O devices. Instead, it simply does its own work, and assumes that I/O will take care of itself!

When a device becomes ready, the CPU *hardware* initiates a branch to an I/O subprogram called an *interrupt service routine (ISR)*, which handles the I/O transaction with the device.

An interrupt can occur during *any* instruction cycle as long as interrupts are enabled. When the current instruction completes, the CPU interrupts the flow of the program, executes the ISR, and then resumes the program. The program itself is not involved and is in fact unaware that it has been interrupted.

What Is an ALU?

An **arithmetic logic unit (ALU)** is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the **central processing unit (CPU)** of a computer. Modern CPUs contain very powerful and complex ALUs. In addition to ALUs, modern CPUs contain a control unit (CU).

Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers. A **register** is a small amount of storage available as part of a CPU. The control unit tells the ALU what operation to perform on that data, and the ALU stores the result in an output register. The control unit moves the data between these registers, the ALU, and memory.

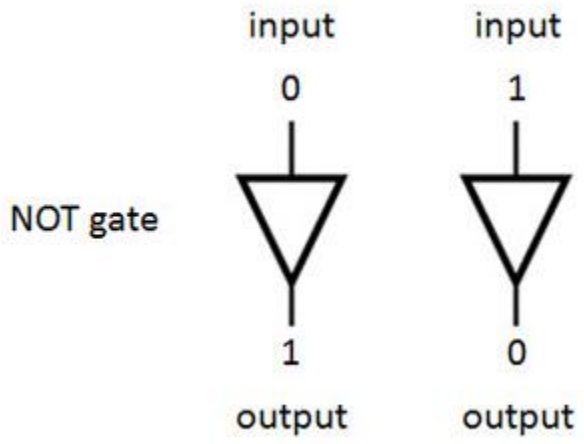
How an ALU Works

An ALU performs basic arithmetic and logic operations. Examples of arithmetic operations are addition, subtraction, multiplication, and division. Examples of logic operations are comparisons of values such as NOT, AND, and OR.

All information in a computer is stored and manipulated in the form of **binary numbers**, i.e. 0 and 1. **Transistor** switches are used to manipulate binary numbers since there are only two possible states of a switch: open or closed. An open transistor, through which there is no current, represents a 0. A closed transistor, through which there is a current, represents a 1.

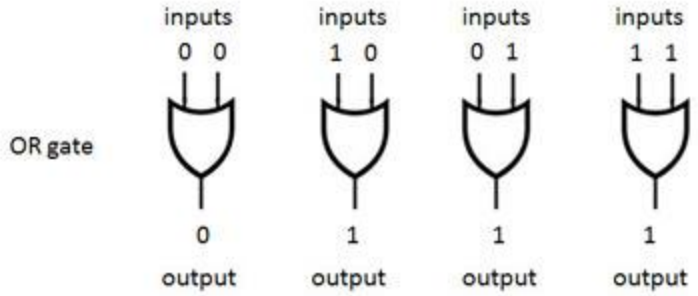
Operations can be accomplished by connecting multiple transistors. One transistor can be used to control a second one - in effect, turning the transistor switch on or off depending on the state of the second transistor. This is referred to as a **gate** because the arrangement can be used to allow or stop a current.

The simplest type of operation is a NOT gate. This uses only a single transistor. It uses a single input and produces a single output, which is always the opposite of the input. This figure shows the logic of the NOT gate:



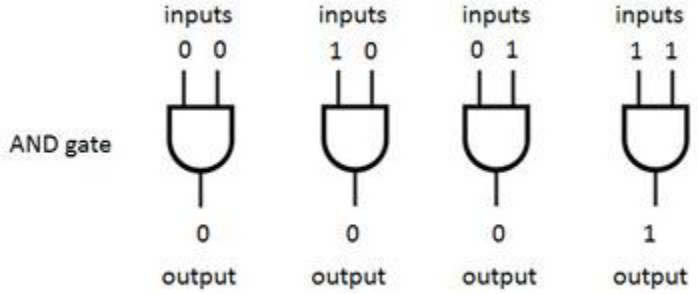
How a NOT gate processes binary data

Other gates consist of multiple transistors and use two inputs. The OR gate results in a 1 if either the first or the second input is a 1. The OR gate only results in a 0 if both inputs are 0. This figure shows the logic of the OR gate:



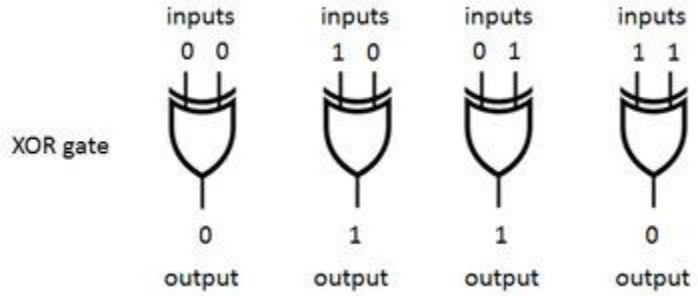
How an OR gate processes binary data

The AND gate results in a 1 only if both the first and second input are 1s. This figure shows the logic of the AND gate:



How an AND gate processes binary data

The XOR gate, also pronounced X-OR gate, results in a 0 if both the inputs are 0 or if both are 1. Otherwise, the result is a 1. This figure shows the logic of the XOR gate:



How an XOR gate processes binary data.