Instruction Fetch Execute Cycle

The Instruction Fetch Execute Cycle is one of the most important mental models of computation as aptly put by Prof. Rockford Ross. This embodies the basic principle of how all modern processors work. This functional model has remained more or less the same over the decades no matter how and when the development of processors have taken place ever since the days of Von Neumann architecture to today’s Super computers. This document is to be read in conjunction with the tutorial presented on the Instruction Fetch Execute Cycle of the Intel processor. The principles are fairly simple and can be easily generalized to any processor or Operating System. It further proceeds to explain what happens when a computer is first switched on till the time it is ready to accept instructions from the user.

It is very important to appreciate the fact that an Operating System (OS) is just like any other program albeit a little more complex as compared to user written programs. The OS provides the functionality to load and execute other programs and thus possesses certain privileges which user programs do not possess. A user program can and does request for OS intervention through a mechanism called system calls e.g. trap, page fault etc.
Once a computer has been powered on it performs a continuous cycle of the following:

- a. Fetch next instruction from memory
- b. Decode the instruction
- c. Execute the instruction

**What are these instructions?**

An instruction as the name instructs the computer what to do. In simple terms, every line of a program that we as users write instructs the computer to perform a series of operations. We may argue that our programs comprise of the instructions belonging to one of those of the High Level Languages like C, C++, and Java etc. A computer understand these high level instructions by converting them into a machine understandable form known as *machine language* comprising of 1’s and 0’s. Thus the following instruction in C

```c
int cnt;
cnt = cnt + 2;
```

may get translated into the corresponding machine language as:

```
000101010101111101010101010000000111111.....
```
We as users would not be able to understand this machine language. To make better sense we resort to Assembly Language where each machine language instruction is assigned a code known as a mnemonic. So instead of remembering some string of 0's and 1's, the programmer would only need to remember short codes like ADD, MOV etc.

e.g. mov al, 0x21

The above mnemonic means to move the value of the hexadecimal number 21 into a register called al. These mnemonics are in turn converted into machine instructions. Therefore, the Central Processing Unit (CPU) can now begin to execute the instructions by fetching them from the memory. The CPU does this continuously till it is shut down.

![Figure 1: Basic Instruction Fetch Execute Cycle](image-url)
Instruction Fetch Execute Cycle

A more complete form of the Instruction Fetch Execute Cycle can be broken down into the following steps:

1. Fetch Cycle
2. Decode Cycle
3. Execute Cycle
4. Interrupt Cycle

A detailed diagram is shown in the tutorial.

1. Fetch Cycle

The fetch cycle begins with retrieving the address stored in the Program Counter (PC). The address stored in the PC is some valid address in the memory holding the instruction to be executed. (In case this address does not exist we would end up causing an interrupt or exception). The Central Processing Unit completes this step by fetching the instruction stored at this address from the memory and transferring this instruction to a special register – Instruction Register (IR) to hold the instruction to be executed. The program counter is incremented to point to the next address from which the new instruction is to be fetched.

2. Decode Cycle

The decode cycle is used for interpreting the instruction that was fetched in the Fetch Cycle. The operands are retrieved from the addresses if the need be.
3. **Execute Cycle**

This cycle as the name suggests, simply executes the instruction that was fetched and decoded.

4. **Interrupt Cycle**

An interrupt can occur any time during the program execution. Whenever it is caused, a series of events take place so that the instruction fetch execute cycle can again resume after the OS calls the routine to handle the interrupt. Therefore, when an interrupt occurs, the following steps are performed by the OS:

- Suspend the execution of current instruction
- Push the address of current instruction on the system stack
- Loading the PC with the address of the interrupt handler
- This starts the Instruction Fetch Execute cycle again for the instructions in the Interrupt handler.
- Set the mode of operation as a privileged one often termed as the *Supervisor mode* so that the OS can execute the handler.
- Once the OS completes the execution of the interrupt handler, the address of the next instruction to be executed is obtained from popping the value of the address in the stack. The suspended instruction can now continue with its execution.

This cycle of fetching a new instruction, decoding it and finally executing it continues until the computer is turned off. Since we have said that it is mainly the operating system which aids the processor in executing the programs which holds the
different instructions, a question which is immediately raised is how does the OS start executing.

**Loading of the Operating System**

The process by which the OS gets loaded into the memory so that it can start executing is known as the *System Bootup*. During the boot up sequence, a series of instructions need to be executed so that at the end of this sequence the OS is running and can in turn start executing user programs. The boot sequence begins with the following:

1. The *RESET* pin of the CPU is set to logical high.
2. The code which is found at some specific starting address (0xffffff0 in case of an Intel processor) is executed.
3. 0xffffff0 maps to the persistent memory chip of the computer known as the Read Only Memory (ROM).
4. The series of instructions stored in ROM is called the *Basic Input /Output System (BIOS)*.

![Figure 2: ROM](http://computer.howstuffworks.com/rom.htm)
Role of BIOS during System Bootup

Although the BIOS performs a number of functions e.g. making sure that all the different chips, hard drives, and the CPU function together as an entity, its most important function is loading the Operating System. When the computer is first powered on, the microprocessor attempts to execute the first instruction. For this purpose as mentioned earlier the processor needs to fetch this instruction. The processor cannot fetch it from the Operating System because the OS has not been loaded yet into the memory; it is still residing on the disk. It is the BIOS which provides the processor with the first instructions to be executed in order to load the OS.

The other functions performed by the BIOS during the system bootup apart from loading the OS are summarized as:

- A power-on self-test (POST) for the different hardware components in the system to make sure everything are functioning properly.

- Activating other BIOS entities e.g. graphics cards.

- Providing a set of low-level routines to enable the OS to interface different hardware devices keyboard, screen, and the ports (serial as well parallel).
Instruction Fetch Execute Cycle during System Bootup

When the computer is powered on, the PC as usual points to some specific address. This pointer holds the address of the instruction to be executed. Thus, the Program Counter is initialized to some specific address like 0xfffffff0 in the case of Intel CPUs. This address would be different for different Operating Systems. This is the address of Read Only Memory (ROM). ROM contains a special program called Boot Loader. Thus immediately, after we turn on the computer, the instructions comprising the Boot loader start executing because the PC was initialized with its address. The boot loader loads the OS from disk to the memory and then executes a JUMP instruction to that section of memory. The boot loader only loads a part of the OS and the instructions subsequent to the jump instruction read the remaining part of the OS into memory. The above process can be summarized as follows:

- The ROM contains the Basic Input Output System (BIOS) which makes a call to the boot loader program.
- The boot loader program loads the OS from disk to memory. It reads from the first sector of the disk which is termed as the Master Boot Record (MBR).
- The boot loader copies the code from the MBR to memory, starting at some specified location depending on the processor.
- The boot loader then performs a jump to that address, and subsequently the code for the OS starts executing.
The Operating system when completely loaded in the memory starts accepting and executing so that it can now in turn start executing the user instructions. It does this by continuously performing the Instruction Fetch Execute Cycle as mentioned before.

After seeing how a generic OS gets loaded, let us briefly try to use the same principle to see how Linux gets loaded in an Intel processor. Generally, Linux is loaded into the memory from the hard disk. The most commonly used Linux boot loader is known as **LILO (Linux Loader)**. This boot loader may be either present in the Master Boot record (MBR) or in the boot sector of some disk partition. When the boot loader (LILO) is executed through the Instruction Fetch Execute cycle, the user may choose which OS is to load in case the processor is dual bootable. This process is described in the following steps:

1. The boot loader moves itself from address 0x00007c00 to address 0x0009a000.
2. Loads the remaining part of the boot loader into RAM starting from address 0x0009b000.
3. This second part of the boot loader now reads a map of the available OS from disk to memory and offers the user with a prompt to choose the OS to load.
4. The boot loader now either copies the boot sector of the corresponding disk partition into RAM or copies the kernel image into RAM.
5. The loader displays the message “Loading Linux” and jumps to the setup().
6. The setup() is written in Assembly Language. It initializes the hardware devices and sets the environment for the kernel program. It further jumps to startup_32().
7. The return from setup() moves the copy of the kernel image to either 0x0001000 or 0x00100000 depending on whether kernel image loaded was high or low.
8. Another jump is made to start_kernel() which completes the initialization of the Linux kernel.

As we can see the basic steps performed are similar loading any generic OS. The differences lie as to what functions gets called and how they called. The final outcome is that the required OS gets loaded into the computer and now we as users can issue instructions to the computer so that it can continue with the Instruction fetch Execute Cycle as discussed before.

References

5. Web resource: http://computer.howstuffworks.com/rom.htm
6. Understanding Linux Kernel, O'Reilly, 2000, Daniel P. Bovet and Marco Cesati