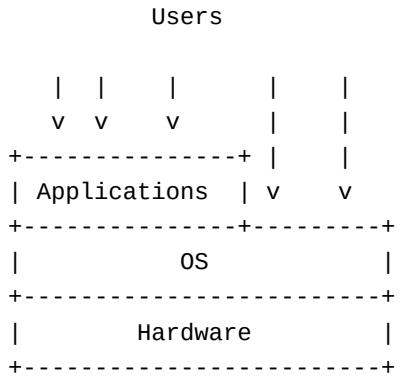
Computer System Overview: Part 1

1 What is operating system

We don't know what an OS is exactly until we have learned this course, but we may have some clues about the answer. Let's think about it. Whenever we want to use computers, we have to boot them up first. Whatever happens in this period, we know it is the OS that is in charge of it. After the computer is available for use, we then interact with the computer through a graphical interface or text-only console. We may run programs, install or uninstall applications in the OS as we need. Thus the following picture may be suitable for describing a computer system:



Thus, we may conclude that an operating system exploits and manages all kinds of computer hardware to provide a set of services directly or indirectly to the users.

Services may be functions the human users can use directly, e.g. file creation, user management, etc., and also those that may be used indirectly, which embody as application programming interface, all kinds of libraries and functions they provide.

2 Computer hardware overview

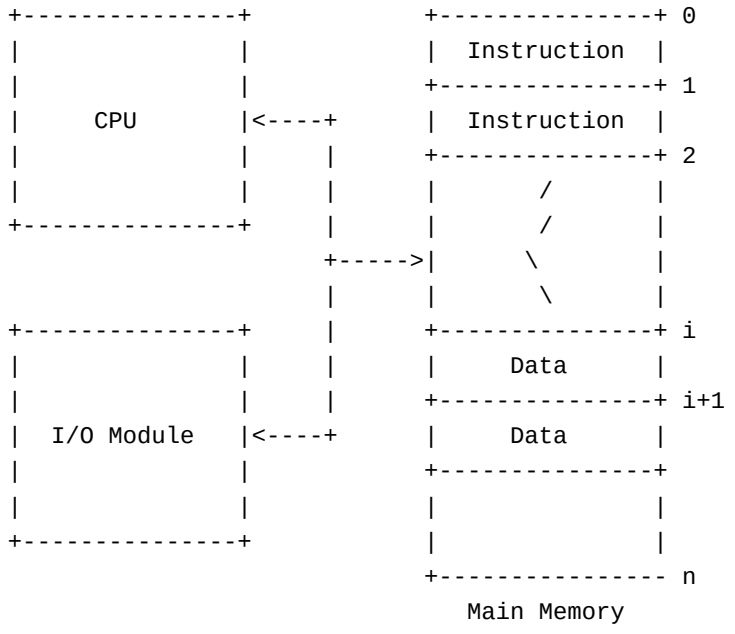
To build an OS, as we can see from the figure, I need to know more details about the hardware. In a computer system, there are all kinds of hardware, CPUs, mainboards, monitors, network adapters, sound cards, mice, keyboards, printers, hard disks, etc. Mainboard is not something that provides a specific function, instead it is a collection of all kinds of slots and modules. To make all these things work together, mainboard provides some kind of physical connections among them, i.e. what we call system bus. Thus based on our analysis, all these components may be divided into several groups: *CPU*, *memory*, *I/O modules* and *system bus*.

Instead of I/O devices, we use I/O modules because it is those I/O modules that communicate directly with CPU or memory.

As for memory, we may also say it is a kind of storage I/O module; however it has a special position in the system since we never heard of drivers for memory to work but I have known plenty of drivers for a variety of I/O modules. Those drivers are actually programs, which have to be loaded into memory to run. Obviously, memory cannot depend on a driver, instead the system includes physical circuits for accessing memory.

2.1 Basic elements

To depict these components and their connections, we present the following top-level view:



According to the figure, data may be transferred between CPU and memory, or CPU and I/O modules, or even memory and I/O modules.

As we know, a memory consists of a set of locations, defined by a sequentially numbered addresses. Each location may be a byte of 8 bits, or a word of 16 bits. It contains a binary number that can be interpreted as either an instruction or pure data.

To access data in memory, CPU makes use of two internal registers: *MAR* (*memory address register*) and *MBR* (*memory buffer register*). MAR specifies the address in memory for the next read/write; MBR otherwise contains the data to be written into memory, or data to be read from memory. Similarly, I/OAR specifies a particular I/O module, and I/OBR is used for the exchange of data between an I/O module and the processor.

An I/O module transfers data from external devices to CPU and memory, and vice versa. It contains buffers for temporarily holding data until they can be sent on.

2.2 Registers

Computers compute. The component that performs computation is CPU, or more concretely it is the *ALU* (*arithmetic logical unit*) in CPU that do the computation. To compute, we need first prepare input; however ALU cannot access memory directly, Instead, a set of registers are provided as a cache that is faster but smaller than main memory. (They are smaller between they are much more expensive than regular memory.)

Except for the registers directly involved in computation, CPU also has some registers in the purpose of control and recording status.

The textbook categorizes registers into two types: user-visible registers, and control and status registers. Since this separation is not common, so here we just explain registers one by one without labelling them as one of which kind.

2.2.1 Data registers

```
MOV AX, 1234H
MOV [4321H], AL
```

2.2.2 Address registers

Segmented addressing registers

We may naturally assume that, to access some location of memory, we simply use an address register to contain the address of that location, but the actual practice is kind of much more complex. One popular addressing method is *segmented addressing*. With this method, memory is divided into segments, and each segment are variable-length blocks of words. To refer to a location in such a memory system, we need to give two pieces of

information. One is which segment, and the other is which item in that segment we are visiting. That is the address consists of two parts, segment address, and the offset within the segment. Accordingly there are two kinds of registers: *segment address registers* and *offset address registers*. For example, CPU x8086, shifts the content of `CS` to the left by 4 bits, and then adds up the result and the content of `IP`. Finally the sum is used as the effective address.

```
CS : IP
DS : DI
DS : SI
```

It should be made clear that segment is just a logical concept, not an physically existing entity in memory. We may simply write to `CS` to change the segment it points to.

Stack pointers

Due to the popularity of stack in programs execution, computer systems provide registers to access memory segment in the way of accessing stacks. For example, in x8086, we have

```
SS : SP
```

where `SS` gives the stack segment, and `SP` always points to the top of the stack. Thus the following two sets of instructions have the same effect:

```
PUSH AX
```

```
SUB SP, 2
MOV [SS:SP], AX
```

2.2.3 Control registers

All the registers we discuss above are related to data access, however we know, memory also contains instructions for CPU to execute. In this purpose, CPU provides

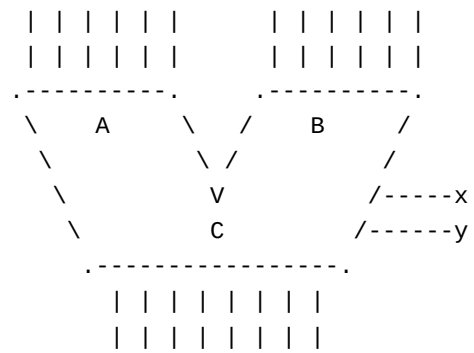
Program counter (PC)

contains the address of an instruction to be fetched from memory

Instruction register (IR)

contains the instruction most recently fetched.

The execution of an instruction is actually to interpret the operation code in the instruction and generate signals for ALU or other components in CPU. For example, when `xy=00`, ALU does `A+B => C`, and when `xy=01`, ALU does `A-B => C`, etc.



2.2.4 Status registers

Besides the above types of registers, CPU also includes registers, that contain status information. They are known as *PSW* (*program status word*). PSW typically contains condition codes plus other status information.

Condition codes are bits set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set following the execution of the instruction. The code may subsequently be tested as part of conditional branch operation. Let's say

```
    CMP AX, BX
    JGE exit

    ...

  exit:

    ...
```

Generally, these condition codes cannot be altered by explicit reference because they are intended for feedback regarding the execution of an instruction, and are updated automatically whenever a related instruction is executed. For example, we aren't supposed to use the following instruction to clear the lowest bit of PSW register.

```
    OR PSW, FEH
```

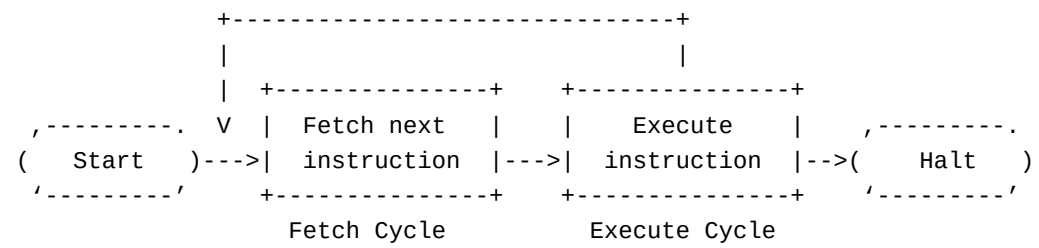
The registers we give here are all very common ones. A CPU may actually provide much more registers. There are a number of factors that have to be taken into account. One is operating-system support. Certain types of control information are of specific utility to the operating system. If the processor designer has a functional understanding of the operating system to be used, then register organization can be designed to provide hardware support for particular features such as memory protection and switching between user program. These features may originally be implemented in software.

Another key factor is the allocation of control information between registers and memory. Although registers are much faster, but due to the price reason, a computer system doesn't have many registers, so at least part of control information has to be put into memory. Thus here comes a problem of balance. You need to consider what control information is more frequently used and in which order.

2.3 Instruction execution

The previous section mainly addresses the static characteristics of a processor, this section otherwise talks about its dynamic side - instruction execution.

A program to be executed a CPU consists of a set of instructions stored in memory. Roughly, the execution of an instruction may be looked on as a process of two steps. At the first step, the instruction is read (fetched) from memory into IR, whose address is specified by PC register. Then at the second step, CPU executes the instruction, i.e. interpreting the instruction and performing the action specified. The first step is called *fetch cycle* and the second *execute cycle*. The whole process of the two steps is called *instruction cycle*. Thus program execution is actually repeating instruction cycles until either the computer is turned off, or an instruction that asks CPU to halt is encountered. The following figure depicts this process:



In a typical processer, the register PC contains the address of the instruction to be fetched next. Whenever an instruction is obtained, the content of PC will increment automatically so that it will fetch the next instruction in sequence.

The fetched instruction is loaded into IR. The instruction contains bits that specify the action the processor is to take. In general, these actions fall into four categories:

- **data exchange between CPU and memory**
- **data exchange between CPU and I/O modules**

  There are two popular methods to address I/O modules. One is allocating part of memory address space for I/O modules, thus to read/write from/to an I/O module, same instructions as used to access memory will do without any change. Of course, you have to specify addresses in the instructions that are corresponding to I/O modules. The

other method is using a separate set of instructions for I/O module. For example, in CPU x8086, the following instructions are used:

```
MOV DX, 61H
OUT DX, AL
...
MOV DX, 60H
IN AL, DX
```

- **data processing**

  The processer may perform some arithmetic or logic operation on data.

- **control**

  Some instructions may affect the sequence of execution. For example:

```
JMP 0700H
...
CMP AL, 08H
JNE different
...
```

An example is given in the textbook to show how a partial program is executed step by step to add two numbers up.