Low Level Programming Basic Concepts

Introduction

Hi! Welcome to low-level world! This time I would like to explain the basic concepts of low level. This part is merely directed toward Intel PC. I'd like to add more for other computers like Mac and Amiga, but I think time wouldn't allow me to do this in near future. :-)

Registers

What is registers exactly? You can consider it as variables inside the CPU chip. Yeah! That depicts registers so close. There are several registers exist in PC:

AX, BX, CX, DX, CS, DS, ES, SS, SP, BP, SI, DI, Flags, and IP

They are all 16-bits. You can treat it as if they are word (or unsigned integer) variables. However, each registers has its own use.

AX, BX, CX, and DX are **general purpose registers**. They can be assigned to any value you want. Of course you need to adjust it into your need. AX is usually called **accumulator register**, or just accumulator. Most of arithmatical operations are done with AX. Sometimes other general purpose registers can also be involved in arithmatical operation, such as DX. The register BX is usually called **base register**. The common use is to do array operations. BX is usually worked with other registers, most notably SP to point to stacks. The register CX is commonly called **counter register**. This register is used for counter purposes. That's why our PC can do looping. DX register is the **data register**. It is usually for reserving data value.

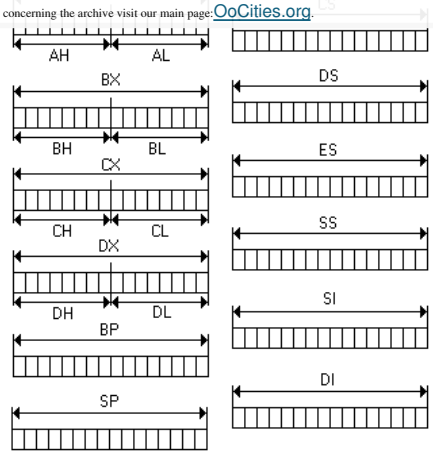
The registers CS, DS, ES, and SS are called **segment registers**. You may not fiddle with these registers. You can only use them in the correct ways only. CS is called **code segment register**. It points to the segment of the running program. We may NOT modify CS directly. Oh yes, what is "segment" anyway? It's discussed later. :-) DS is called **data segment register**. It points to the segment of the data used by the running program. You can point this to anywhere you want as long as it contains the desired data. ES is called **extra segment register**. It is usually used with DI and doing pointers things. The couple DS:SI and ES:DI are commonly used to do string operations. SS is called **stack segment register**. It points to stack segment.

The register SI and DI are called **index registers**. These registers are usually used to process arrays or strings. SI is called **source index** and DI is **destination index**. As the name follows, SI is always pointed to the source array and DI is always pointed to the destination. This is usually used to move a block of data, such as records (or structures) and arrays. These register is commonly coupled with DS and ES.

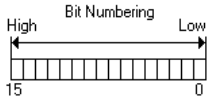
The register BP, SP, and IP are called **pointer registers**. BP is **base pointer**, SP is **stack pointer**, and IP is **instruction pointer**. Usually BP is used for preserving space to use local variables. SP is used to point the current stack. Although SP can be modified easily, you must be cautious. It's because doing the wrong thing with this register could cause your program in ruin. IP denotes the current pointer of the running program. It is always coupled with CS and it is NOT modifiable. So, the couple of CS:IP is a pointer pointing to the current instruction of running program. You can NOT access CS nor IP directly.

The **flag** register is used to store the current status of the processor. It holds the value of which the programmers may need to access. These involves detecting whether the last arithmatic holds zero result or may be overflow. You can only modify flag from stack.

The general registers AX, BX, CX, and DX are 16-bit. However, they are composed from two smaller registers. For example: AX. **The high 8-bit** is called AH, and **the low 8-bit** is called AL. Both AH and AL can be accessed directly. However, since they altogether embodied AX, modifying AH is modifying the high 8-bit of AX. Modifying AL is modifying the low 8-bit of AX. Here's a picture that may enlighten you :-)



Bit numbering of a register begins from the lower part. The lowest bit is numbered as bit 0, the highest bit is numbered as bit 15. So, there are 16 bits. Therefore, AL occupy bit 0 to bit 7 of AX, AH occupy bit 8 to bit 15 of AX.



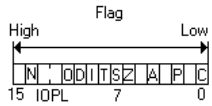
386 processors introduces extended register. Most of the registers, except segment registers are enhanced into 32-bit. So, we have extended registers EAX, EBX, ECX, and so on. AX is only the low 16-bit (bit 0 to 15) of EAX. BX is only the low 16-bit (bit 0 to 15) of EBX and so on. There are no special direct access to the upper 16-bit (bit 16 to 31) in extended register. Segment registers are not extended. There are no ECS, or EDS or so.

Flag is discussed separately on the next caption.

Flag

Flag is actually 16-bit register that contains processor status. Intel doesn't provide a direct access to it, rather it is accessed via stack. (via `POPF` and `PUSHF`) However, for some reason you can then access flag using the assembly instruction `SAHF` and `LAHF` for just some flag attributes.

You can access each flag attribute by using bitwise `AND` operation since each status is mostly represented by just 1 bit. Here's the flag lay-out:



1. C denotes carry flag (bit 0). It is turned to 1 whenever the last arithmatical operation, such as adding and subtracting, has carry or borrow, otherwise 0. DOS often uses this to indicate errors.
2. P denotes parity flag (bit 2). Seldomly used. It will set to 1 if the last operation (any operation) results even number of bit 1. It is usually used in communication things.
3. A denotes auxilarry flag (bit 4). Seldomly used. It is set in Binary Coded Decimal (BCD) operations.
4. Z denotes zero flag (bit 6). Usually used to detect whether the last operation (any operation) holds zero result.
5. S denotes sign flag (bit 7). It is often used to detect whether the last operation holds negative result. It is set to 1 if the highest bit (bit 7 in bytes, or bit 15 in words) of the last operation is 1.
6. T denotes trap flag (bit 8). It is only used in debuggers to turn on the step-by-step feature.
7. I denotes interrupt flag (bit 9). It is used to toggle the interrupt enable or not. If the bit is set (= 1), then the interrupts are enabled, otherwise disabled. The default is on.
8. D denotes interrupt flag (bit 10). It is used for directions of string operations. If the bit is set, then all string operations are done backward. Otherwise, forward. The default is forward (= 0).
9. O denotes the overflow flag (bit 11). It is used to detect whether the last arithmatic operation result has overflowed or not. If the bit is set, then it has been an overflow.
10. IOPL denotes the I/O Previlege Level flag (bit 12 to 13). It is used to denote the previlege level of the running programs. It is rarely used in real mode programming. This flag is exist on 286 or better CPUs.
11. N denotes the Nested Task flag (bit 14) this flag is exist on 286 or better CPU. This is to detect whether it has been multiple task (or exceptions) occur. Rarely used in practical programming.

Upon the most often used flag is O, D, I, S, Z, and C.

386 or better CPUs has enhanced the flag into 32 bit. But that's out of the scope. We talked about the basics, didn't we? :-)

Memory

memory, CPU uses registers. Originally, CPU only has 16-bit registers, so the maximum amount of memory that can be addressed is 2^{16} = 65536 (64K). However, after XT arrives, the memory is extended to 1 MB. That is 16 times bigger than the original.Unfortunately, the CPU still has 16 bit registers which is, in fact, can not handle all the memory. Then, the engineers have to get around with this. Therefore, the technique called **segmentation** is invented. That means the memory is divided virtually into several areas called **segment**.

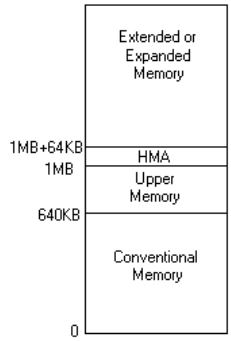
Upon the arrival of segmentation, the segment registers are also exist to corporate the idea. The segment registers are 16 bit, too. The idea of the segmentation is NOT dividing 1 MB into 16 exact parts. That means that segment registers are only allowed to have the value of 0 to 15, right? That only use 4 bits. For the sake of the ease for memory maintenance of operating system (that is DOS), the engineers don't just make the other 12 bits in segment registers just wasted, the segmentation is then **interleaved**.

It means that if we say the segment number 0, then we can access the memory 0 to 65536. Segment number 1 allows us to access memory number 16 to 65552. Segment 2 from 32 to 65568, and so on **with the increment of 16**. Therefore, all the 1 MB memory is addressable. Why did they do that? It is for the sake of the operating system memory management stuff. Therefore, DOS align the executed code to the nearest 16 bytes alignment.

The memory access must be done in a pair of register. The first is the segment register and next is any register, usually BX, DX, SI or DI. The register pair usually written like this: ES:DI with a colon between them. The pair is called the **segment:offset** pair. So, ES:DI means that the segment part is addressed by ES, and the offset part is addressed by DI.

If the ES contains 0, and DI is 5, means that we access the memory 5. If ES:DI = 0001:0005 then it actually access the actual address 21 (1 * 16 + 5 = 21). Remember the interleaving I've mentioned above. So, 0000:0021 and 0001:0005 is actually the same address. How could the processor do that? The register pair segment:offset contains the **logical address**. The actual address or the **absolute address** need to be calculated from the logical address. Since the increment of the interleaving is 16, then we need to multiply the segment value with 16 first, then add it with the offset part.

Usually programmers refer the memory 0 to 640 KB as the **low memory**. Sometimes it is called **conventional memory**. The area above the first 640 KB until 1 MB is called **upper memory**. Then the 64KB after the border 1 MB is called **high memory area (HMA)**. After that is either **extended** or **expanded memory**. Look at this picture.



Nowadays, the difference between extended memory and expanded memory is not too clear. It's fully depends on the driver. `HIMEM.SYS` provide access to extended memory. `EMM386.EXE`, `QEMM`, `386MAX`, or so provides the access to expanded memory. It's fully depends on you which one you'd like to use, but programmers prefer expanded memory rather than extended one. It is because of its speed.

Pointers are actually integers (or long integers) that contains the address of specific location in the memory. That's why pointers can access memory indirectly.

That's the general memory lay out.

Stacks

Now, let's talk about specific memory lay out. DOS, load the program code into memory. A specific amount of memory is reserved in order to make the program runs as expected. Each the program memory mode behaves differently. However, there is one thing: there must be a room for the code itself, then there must be a room for data, and the last thing is there must be a room for **stack**.

What is stack exactly? You can say that it is a temporary area to store temporary things. :-) It is mainly used to pass the parameter value to procedures or functions. Sometimes, it also act as temporary space to allocate for local variables. Therefore, the role of the stack is very important.

How the stack works? It works exactly as the stack in linked list! The last item pushed into stack is going to be popped first. LIFO concept works here. At this moment, you don't have to know how stack work in depth. The main thing is that you know that stack here uses LIFO concept, but it is **NOT a linked list**.

How to adjust stacks? Reserve as much memory as needed for stack. If you use many parameters in your procedure or functions, you need to reserve bigger stack. Usually 2 KB or 4 KB is enough for many programs. However, if you use a lot of local variables, you need to reserve more.

That's all about the stacks, let's advance.

Interrupts

What is interrupts exactly? It is like its name: interrupts. It interrupts processes. Upon a request of an interrupt, the processor usually **stores only the CS:IP and flag** state of the running program, then it goes to the interrupt routine. After processing the interrupt, the processor restores all states stored and resume the program. There are three kind of interrupts: hardware (other than CPU) interrupts, software interrupts, and CPU-generated interrupts.

Hardware interrupts occurs if one of the hardware inside your computer needs immediate processing. Delaying the process could cause unpredictable, or even,

Software interrupts occurs if the running program requests the program to be interrupted and do something else. It is usually like waiting the user input from keyboard, or may be request the graphic driver to initialize itself to graphic screen.

CPU-generated interrupts occurs if the processor knows that is something wrong with the running code. It is usually directed for crash protection. If your program contains instructions that processor doesn't know, the processor interrupts your program. It also happens if you divide a number with 0.

Interrupts has a lot of use. It has routines that ease programming life. Changing into graphic screen, waiting for a key, accessing files, disks and so on are done through interrupts.

Notes

Finally, you get all the **basic** concepts. There are still a lot more to learn in low-level programming. Now, go to the implementation. Pascal, C / C++, or even assembler. You can choose from the link below. If you don't understand, don't hesitate to mail me.

Where to go?

Back to main page
Back to Pascal Tutorial Lesson 2 contents
Pascal implementation
C / C++ implementation
Assembly implementation
My page of programming link
Contact me here
