爱了爱了，这篇寄存器讲的有点意思

CPP开发者 2021-12-29 19:55

The following article is from 程序员cxuan Author cxuan

程序员cxuan

cxuan 写的文章还不错。会分享计算机底层、计算机网络、操作系统，Java基础、框架、…

下面我们就来介绍一下关于寄存器的相关内容。我们知道，寄存器是 CPU 内部的构造，它主要用于信息的存储。除此之外，CPU 内部还有运算器，负责处理数据；控制器控制其他组件；外部总线连接 CPU 和各种部件，进行数据传输；内部总线负责 CPU 内部各种组件的数据处理。

那么对于我们所了解的汇编语言来说，我们的主要关注点就是寄存器。

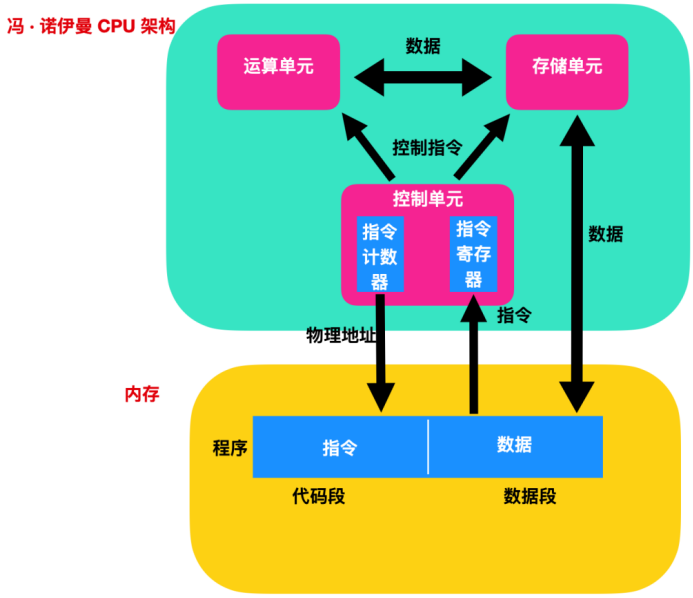
为什么会出现寄存器？因为我们知道，程序在内存中装载，由 CPU 来运行，CPU 的主要职责就是用来处理数据。

那么这个过程势必涉及到从存储器中读取和写入数据，因为它涉及通过控制总线发送数据请求并进入存储器存储单元，通过同一通道获取数据，这个过程非常的繁琐并且会涉及到大量的内存占用，而且有一些常用的内存页存在，其实是没有必要的，因此出现了寄存器，存储在 CPU 内部。

认识寄存器

寄存器的官方叫法有很多，Wiki 上面的叫法是 `Processing Register`，也可以称为 `CPU Register`，计算机中经常有一个东西多种叫法的情况，反正你知道都说的是寄存器就可以了。

认识寄存器之前，我们首先先来看一下 CPU 内部的构造。



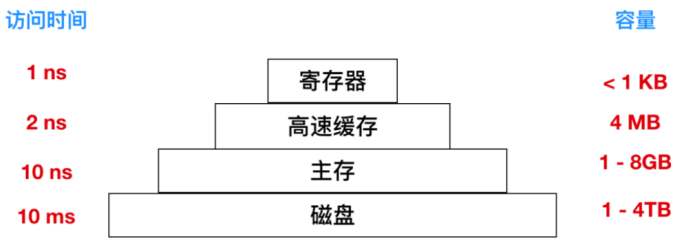
CPU 从逻辑上可以分为 3 个模块，分别是控制单元、运算单元和存储单元，这三部分由 CPU 内部总线连接起来。

几乎所有的冯·诺伊曼型计算机的 CPU，其工作都可以分为5个阶段：**「取指令、指令译码、执行指令、访存取数、结果写回」**。

- 取指令 阶段是将内存中的指令读取到 CPU 中寄存器的过程，程序寄存器用于存储下一条指令所在的地址
- 指令译码 阶段，在取指令完成后，立马进入指令译码阶段，在指令译码阶段，指令译码器按照预定的指令格式，对取回的指令进行拆分和解释，识别区分出不同的指令类别以及各种获取操作数的方法。
- 执行指令 阶段，译码完成后，就需要执行这一条指令了，此阶段的任务是完成指令所规定的各种操作，具体实现指令的功能。
- 访问取数 阶段，根据指令的需要，有可能需要从内存中提取数据，此阶段的任务是：根据指令地址码，得到操作数在主存中的地址，并从主存中读取该操作数用于运算。
- 结果写回 阶段，作为最后一个阶段，结果写回（Write Back，WB）阶段把执行指令阶段的运行结果数据写回到 CPU 的内部寄存器中，以便被后续的指令快速地存取；

计算机架构中的寄存器

寄存器是一块速度非常快的计算机内存，下面是现代计算机中具有存储功能的部件比对，可以看到，寄存器的速度是最快的，同时也是造价最高昂的。



存储层次结构

我们以 intel 8086 处理器为例来进行探讨，8086 处理器是 x86 架构的前身。在 8086 后面又衍生出来了 8088 。

在 8086 CPU 中，地址总线达到 20 根，因此最大寻址能力是 2^20 次幂也就是 1MB 的寻址能力，8088 也是如此。

在 8086 架构中，所有的内部寄存器、内部以及外部总线都是 16 位宽，可以存储两个字节，因为是完全的 16 位微处理器。8086 处理器有 14 个寄存器，每个寄存器都有一个特有的名称，即

「AX，BX，CX，DX，SP，BP，SI，DI，IP，FLAG，CS，DS，SS，ES」

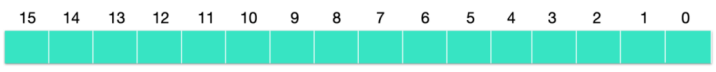
这 14 个寄存器有可能进行具体的划分，按照功能可以分为三种

- 通用寄存器
- 控制寄存器
- 段寄存器

下面我们分别介绍一下这几种寄存器

通用寄存器

通用寄存器主要有四种 ，即**「AX、BX、CX、DX」**同样的，这四个寄存器也是 16 位的，能存放两个字节。AX、BX、CX、DX 这四个寄存器一般用来存放数据，也被称为 数据寄存器 。它们的结构如下



8086 CPU 的上一代寄存器是 8080 ，它是一类 8 位的 CPU，为了保证兼容性，8086 在 8080 上做了很小的修改，8086 中的通用寄存器 AX、BX、CX、DX 都可以独立使用两个 8 位寄存器来使用。

在细节方面，AX、BX、CX、DX 可以再向下进行划分

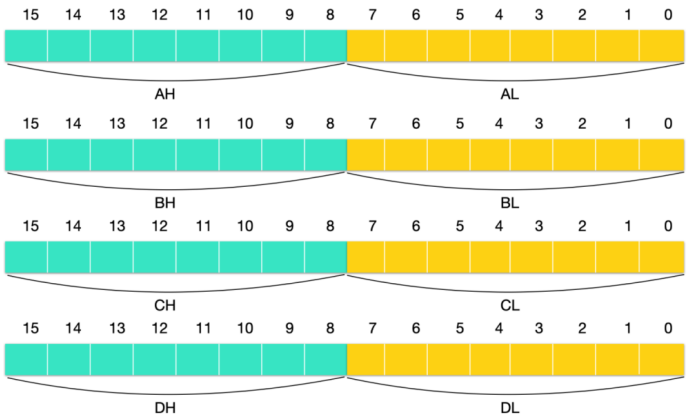
- AX(Accumulator Register)：累加寄存器，它主要用于输入/输出和大规模的指令运算。
- BX(Base Register)：基址寄存器，用来存储基础访问地址
- CX(Count Register)：计数寄存器，CX 寄存器在迭代的操作中会循环计数
- DX(data Register)：数据寄存器，它也用于输入/输出操作。它还与 AX 寄存器以及 DX 一起使用，用于涉及大数值的乘法和除法运算。

这四种寄存器可以分为上半部分和下半部分，用作八个 8 位数据寄存器

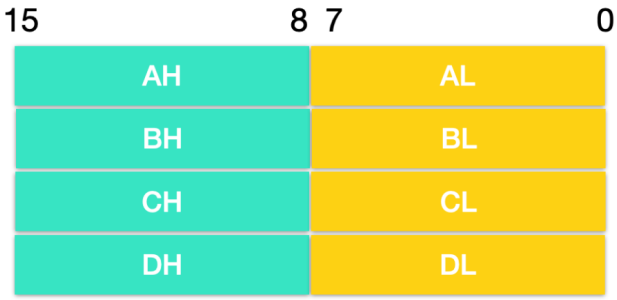
- **「AX 寄存器可以分为两个独立的 8 位的 AH 和 AL 寄存器；」**
- **「BX 寄存器可以分为两个独立的 8 位的 BH 和 BL 寄存器；」**
- **「CX 寄存器可以分为两个独立的 8 位的 CH 和 CL 寄存器；」**
- **「DX 寄存器可以分为两个独立的 8 位的 DH 和 DL 寄存器；」**

除了上面 AX、BX、CX、DX 寄存器以外，其他寄存器均不可以分为两个独立的 8 位寄存器

如下图所示。



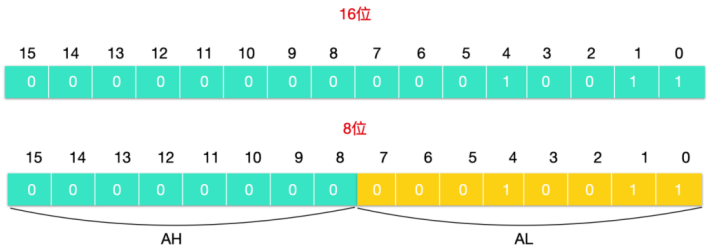
合起来就是



AX 的低位（0 - 7）位构成了 AL 寄存器，高 8 位（8 - 15）位构成了 AH 寄存器。AH 和 AL 寄存器是可以使用的 8 位寄存器，其他同理。

在认识了寄存器之后，我们通过一个示例来看一下数据的具体存储方式。

比如数据 19 ，它在 16 位存储器中所存储的表示如下



寄存器的存储方式是先存储低位，如果低位满足不了就存储高位，如果低位能够满足，高位用 0 补全，在其他低位能满足的情况下，其余位也用 0 补全。

8086 CPU 可以一次存储两种类型的数据

- 字节(byte)：一个字节由 8 bit 组成，这是一种恒定不变的存储方式
- 字(word)：字是由指令集或处理器硬件作为单元处理的固定大小的数据，对于 intel 来说，一个字长就是两个字节，字是计算机一个非常重要的特征，针对不同的指令集架构来说，计算机一次处理的数据也是不同的。也就是说，针对不同指令集的机器，一次能处理不用的字长，有字、双字（32位）、四字（64位）等。

AX 寄存器

我们上面探讨过，AX 的另外一个名字叫做累加寄存器或者简称为累加器，其可以分为 2 个独立的 8 位寄存器 AH 和 AL；在编写汇编程序中，AX 寄存器可以说是使用频率最高的寄存器。

下面是几段汇编代码

```
mov ax,20   /* 将 20 送入寄存器 AX*/
mov ah,80   /* 将 80 送入寄存器 AH*/
add ax,10   /* 将寄存器 AX 中的数值加上 8 */
```

> 这里注意下：上面代码中出现的是 ax、ah，而注释中确是 AX、AH ，其实含义是一样的，不区分大小写。

AX 相比于其他通用寄存器来说，有一点比较特殊，AX 具有一种特殊功能的使用，那就是使用 DIV 和 MUL 指令式使用。

> DIV 是 8086 CPU 中的 除法 指令。
>
> MUL 是 8086 CPU 中的 乘法 指令。

BX 寄存器

BX 被称为数据寄存器，即表明其能够暂存一般数据。同样为了适应以前的 8 位 CPU ，而可以将 BX 当做两个独立的 8 位寄存器使用，即有 BH 和 BL。BX 除了具有暂存数据的功能外，还用于 寻址 ，即寻找物理内存地址。

BX 寄存器中存放的数据一般是用来作为 偏移地址 使用的，因为偏移地址当然是在基址地址上的偏移了。偏移地址是在段寄存器中存储的，关于段寄存器的介绍，我们后面再说。

CX 寄存器

CX 也是数据寄存器，能够暂存一般性数据。同样为了适应以前的 8 位 CPU ，而可以将 CX 当做两个独立的 8 位寄存器使用，即有 CH 和 CL。

除此之外，CX 也是有其专门的用途的，CX 中的 C 被翻译为 Counting 也就是计数器的功能。当在汇编指令中使用循环 LOOP 指令时，可以通过 CX 来指定需要循环的次数，每次执行

循环 LOOP 时候，CPU 会做两件事

- 一件事是计数器自动减 1
- 还有一件就是判断 CX 中的值，如果 CX 中的值为 0 则会跳出循环，而继续执行循环下面的指令，

  当然如果 CX 中的值不为 0 ，则会继续执行循环中所指定的指令 。

DX 寄存器

DX 也是数据寄存器，能够暂存一般性数据。同样为了适应以前的 8 位 CPU ，DX 的用途其实在前面介绍 AX 寄存器时便已经有所介绍了，那就是支持 MUL 和 DIV 指令。同时也支持数值溢出等。

段寄存器

CPU 包含四个段寄存器，用作程序指令，数据或栈的基础位置。实际上，对 IBM PC 上所有内存的引用都包含一个段寄存器作为基本位置。

段寄存器主要包含

- CS(Code Segment)：代码寄存器，程序代码的基础位置
- DS(Data Segment)：数据寄存器，变量的基本位置
- SS(Stack Segment)：栈寄存器，栈的基础位置
- ES(Extra Segment)：其他寄存器，内存中变量的其他基本位置。

索引寄存器

索引寄存器主要包含段地址的偏移量，索引寄存器主要分为

- BP(Base Pointer)：基础指针，它是栈寄存器上的偏移量，用来定位栈上变量
- SP(Stack Pointer)：栈指针，它是栈寄存器上的偏移量，用来定位栈顶
- SI(Source Index)：变址寄存器，用来拷贝源字符串
- DI(Destination Index)：目标变址寄存器，用来复制到目标字符串

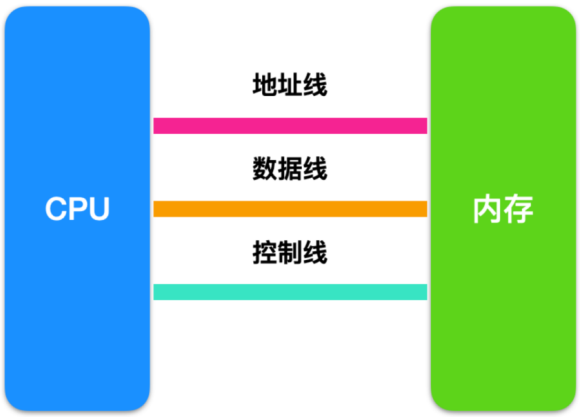
状态和控制寄存器

就剩下两种寄存器还没聊了，这两种寄存器是指令指针寄存器和标志寄存器：

- IP(Instruction Pointer)：指令指针寄存器，它是从 Code Segment 代码寄存器处的偏移来存储执行的下一条指令
- FLAG：Flag 寄存器用于存储当前进程的状态，这些状态有
  - 位置 (Direction)：用于数据块的传输方向，是向上传输还是向下传输
  - 中断标志位 (Interrupt)：1 - 允许；0 - 禁止
  - 陷入位 (Trap)：确定每条指令执行完成后，CPU 是否应该停止。1 - 开启，0 - 关闭
  - 进位 (Carry)：设置最后一个无符号算术运算是否带有进位
  - 溢出 (Overflow)：设置最后一个有符号运算是否溢出
  - 符号 (Sign)：如果最后一次算术运算为负，则设置 1 =负，0 =正
  - 零位 (Zero)：如果最后一次算术运算结果为零，1 = 零
  - 辅助进位 (Aux Carry)：用于第三位到第四位的进位
  - 奇偶校验 (Parity)：用于奇偶校验

物理地址

我们大家都知道， CPU 访问内存时，需要知道访问内存的具体地址，内存单元是内存的基本单位，每一个内存单元在内存中都有唯一的地址，这个地址即是 物理地址 。而 CPU 和内存之间的交互有三条总线，即数据总线、控制总线和地址总线。



CPU 通过地址总线将物理地址送入存储器，那么 CPU 是如何形成的物理地址呢？这将是我们接下来的讨论重点。

现在，我们先来讨论一下和 8086 CPU 有关的结构问题。

cxuan 和你聊了这么久，你应该知道 8086 CPU 是 16 位的 CPU 了，那么，什么是 16 位的 CPU 呢？

你可能大致听过这个回答，16 位 CPU 指的是 CPU 一次能处理的数据是 16 位的，能回答这个问题代表你的底层还不错，但是不够全面，其实，16 位的 CPU 指的是

- CPU 内部的运算器一次最多能处理 16 位的数据

> 运算器其实就是 ALU，运算控制单元，它是 CPU 内部的三大核心器件之一，主要负责数据的运算。

- 寄存器的最大宽度为 16 位

> 这个寄存器的最大宽度值就是通用寄存器能处理的二进制数的最大位数

- 寄存器和运算器之间的通路为 16 位

> 这个指的是寄存器和运算器之间的总线，一次能传输 16 位的数据

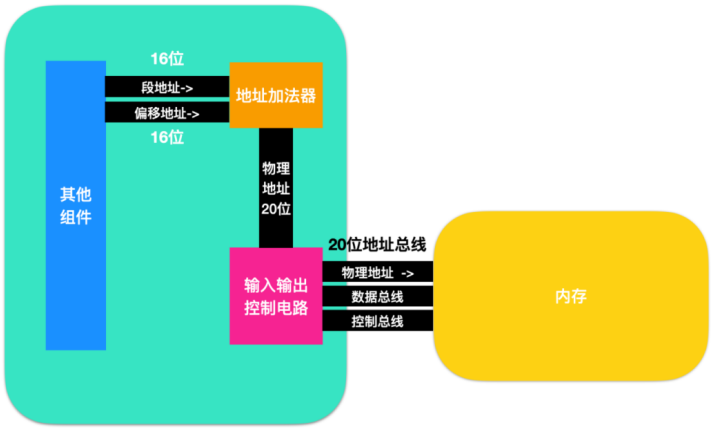
好了，现在你应该知道为什么叫做 16 位 CPU 了吧。

在你知道上面这个问题的答案之后，我们下面就来聊一聊如何计算物理地址。

8086 CPU 有 20 位地址总线，每一条总线都可以传输一位的地址，所以 8086 CPU 可以传送 20 位地址，也就是说，8086 CPU 可以达到 2^20 次幂的寻址能力，也就是 1MB。

8086 CPU 又是 16 位的结构，从 8086 CPU 的结构看，它只能传输 16 位的地址，也就是 2^16 次幂也就是 64 KB，那么它如何达到 1MB 的寻址能力呢？

原来，8086 CPU 的内部采用两个 16 位地址合成的方式来传输一个 20 位的物理地址，如下图所示

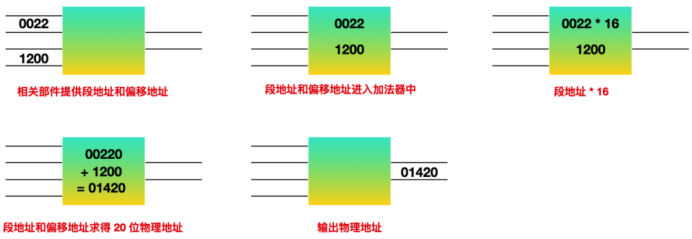


叙述一下上图描述的过程

CPU 相关组件提供两个地址：段地址和偏移地址，这两个地址都是 16 位的，他们经由 地址加法器 变为 20 位的物理地址，这个地址即是输入输出控制电路传递给内存的物理地址，由此完成物理地址的转换。

地址加法器采用**「物理地址 = 段地址 * 16 + 偏移地址」**的方法用段地址和偏移地址合成物理地址。

下面是地址加法器的工作流程



其实段地址 * 16 ，就是左移 4 位。在上面的叙述中，物理地址 = 段地址 * 16 + 偏移地址，其实就是**「基础地址 + 偏移地址 = 物理地址」**寻址模式的一种具体实现方案。基础地址其实就等于段地址 * 16。

你可能不太清楚 段 的概念，下面我们就来探讨一下。

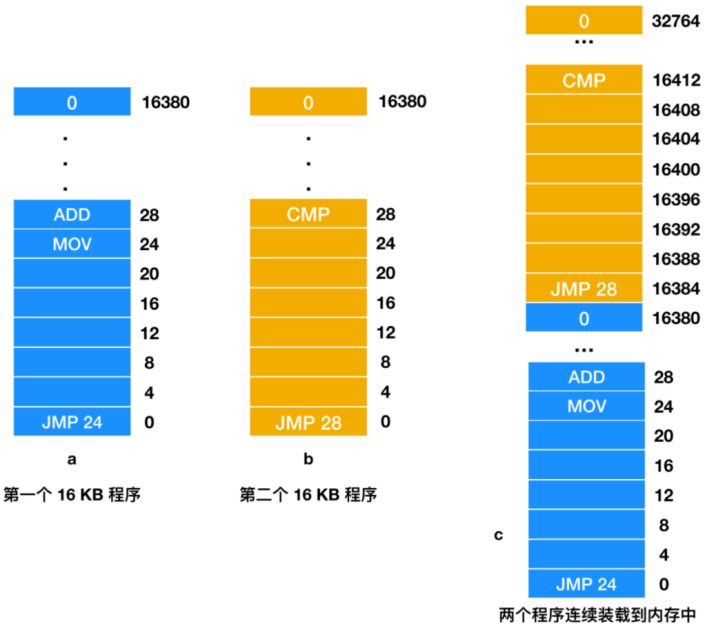
什么是段

段这个概念经常出现在操作系统中，比如在内存管理中，操作系统会把不同的数据分成 段 来存储，比如**「代码段、数据段、bss 段、rodata 段」**等。

但是这些的划分并不是内存干的，cxuan 告诉你是谁干的，这其实是幕后 Boss CPU 搞的，内存当作了声讨的对象。

其实，内存没有进行分段，分段完全是由 CPU 搞的，上面聊过的通过基础地址 + 偏移地址 = 物理地址的方式给出内存单元的物理地址，使得我们可以分段管理 CPU。

如图所示



这是两个 16 KB 的程序分别被装载进内存的示意图，可以看到，这两个程序的段地址的大小都是 16380。

> 这里需要注意一点， 8086 CPU 段地址的计算方式是段地址 * 16，所以，16 位的寻址能力是 2^16 次方，所以一个段的长度是 64 KB。

段寄存器

cxuan 在上面只是简单为你介绍了一下段寄存器的概念，介绍的有些浅，而且介绍段寄存器不介绍段也有**「不知庐山真面目」**的感觉，现在为你详细的介绍一下，相信看完上面的段的概念之后，段寄存器也是手到擒来。

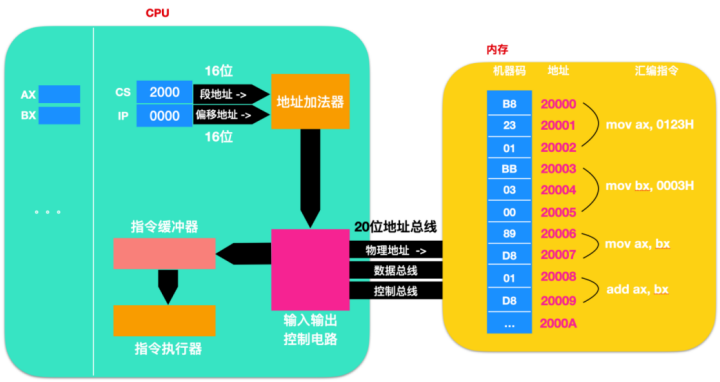
我们在合成物理地址的那张图提到了 相关部件 的概念，这个相关部件其实就是 段寄存器 ，即**「CS、DS、SS、ES」**。8086 的 CPU 在访问内存时，由这四个寄存器提供内存单元的段地址。

CS 寄存器

要聊 CS 寄存器，那么 IP 寄存器是你绕不过去的曾经。CS 和 IP 都是 8086 CPU 非常重要的寄存器，它们指出了 CPU 当前需要读取指令的地址。

> CS 的全称是 Code Segment，即代码寄存器；而 IP 的全称是 Instruction Pointer ，即指令指针。现在知道这两个为什么一起出现了吧！

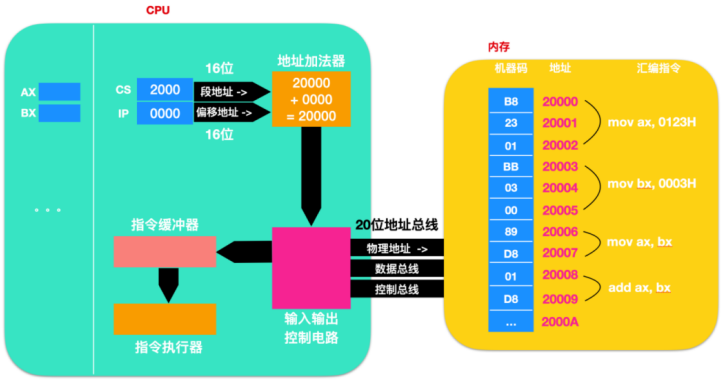
在 8086 CPU 中，由 CS:IP 指向的内容当作指令执行。如下图所示



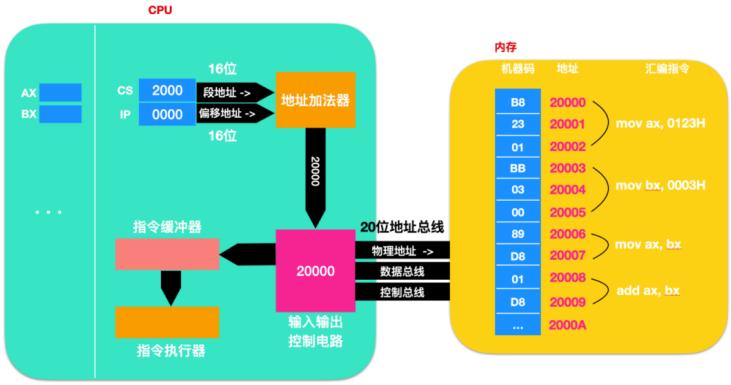
说明一下上图

在 CPU 内部，由 CS、IP 提供段地址，由加法器负责转换为物理地址，输入输出控制电路负责输入/输出数据，指令缓冲器负责缓冲指令，指令执行器负责执行指令。在内存中有一段连续存储的区域，区域内部存储的是机器码、外面是地址和汇编指令。

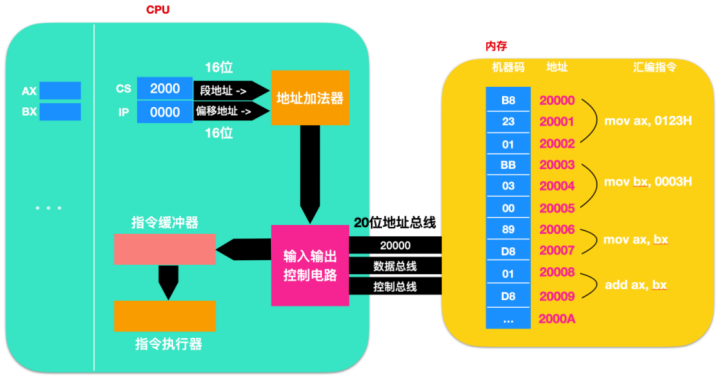
上面这幅图的段地址和偏移地址分别是 2000 和 0000，当这两个地址进入地址加法器后，会由地址加法器负责将这两个地址转换为物理地址



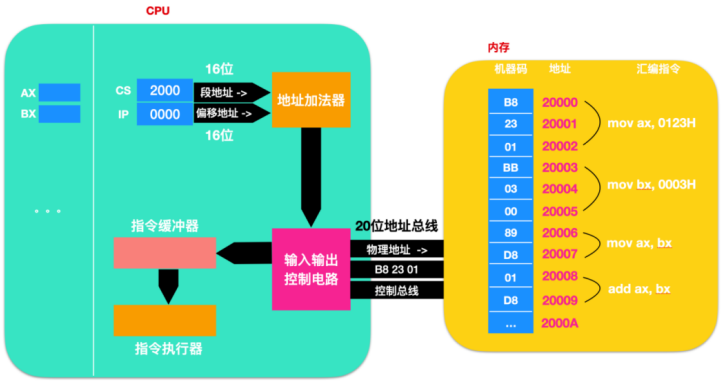
然后地址加法器负责将指令输送到输入输出控制电路中



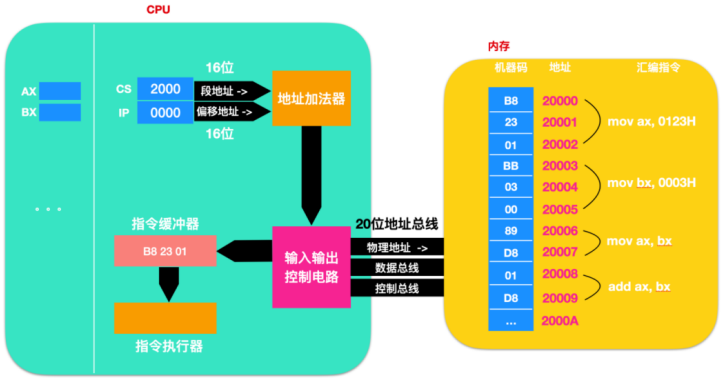
输入输出控制电路将 20 位的地址总线送到内存中。



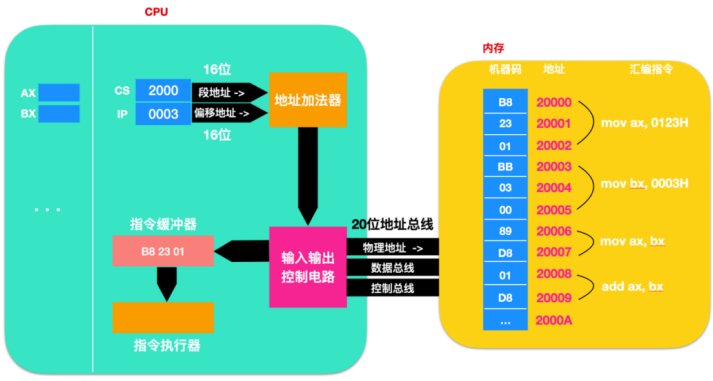
然后取出对应的数据，也就是「**B8、23、01**」，图中的 B8、BB 都是操作数。



控制输入/输出电路会将 B8 23 01 送入指令缓存器中。



此时这个指令就已经具备执行条件，此时 IP 也就是指令指针会自动增加。我们上面说到 IP 其实就是从 Code Segment 也就是 CS 处偏移的地址，也就是偏移地址。它会知道下一个需要读取指令的地址，如下图所示



在这之后，指令执行执行取出的 B8 23 01 这条指令。

然后下面再把 2000 和 0003 送到地址加法器中再进行后续指令的读取。后面的指令读取过程和我们上面探讨的如出一辙，这里 cxuan 就不再赘述啦。

通过对上面的描述，我们能总结一下 8086 CPU 的工作过程

- 段寄存器提供段地址和偏移地址给地址加法器

- 由地址加法器计算出物理地址通过输入输出控制电路将物理地址送到内存中
- 提取物理地址对应的指令，经由控制电路取回并送到指令缓存器中
- IP 继续指向下一条指令的地址，同时指令执行器执行指令缓冲器中的指令

什么是 Code Segment

Code Segment 即代码段，它就是我们上面聊到就是 CS 寄存器中存储的基础地址，也就是段地址，段地址其本质上就是一组内存单元的地址，例如上面的**「mov ax,0123H 、mov bx,0003H」**。

我们可以将长度为 N 的一组代码，存放在一组连续地址、其实地址为 16 的倍数的内存单元中，我们可以认为，这段内存就是用来存放代码的。

DS 寄存器

CPU 在读写一个内存单元的时候，需要知道这个内存单元的地址。在 8086 CPU 中，有一个 DS 寄存器，通常用来存放访问数据的段地址。如果你想要读取一个 10000H 的数据，你可能会需要下面这段代码

```
mov bx,10000H
mov ds,bx
mov a1,[0]
```

上面这三条指令就把 10000H 读取到了 a1 中。

在上面汇编代码中，mov 指令有两种传送方式

- 一种是把数据直接送入寄存器
- 一种是将一个寄存器的内容送入另一个寄存器

但是不仅仅如此，mov 指令还具有下面这几种表达方式

描述	举例
mov 寄存器，数据	比如：mov ax,8
mov 寄存器，寄存器	比如：mov ax,bx
mov 寄存器，内存单元	比如：mov ax,[0]

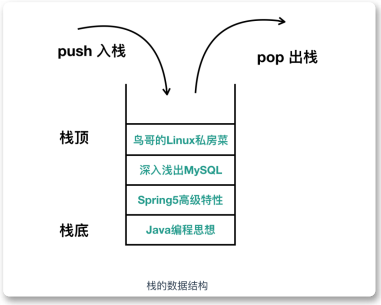
描述	举例
mov 内存单元，寄存器	比如：mov[0], ax
mov 段寄存器，寄存器	比如：mov ds,ax

栈

栈我相信大部分小伙伴已经非常熟悉了， 栈 是一种具有特殊的访问方式的存储空间。它的特殊性就在于，先进入栈的元素，最后才出去，也就是我们常说的 先入后出 。

它就像一个大的收纳箱，你可以往里面放相同类型的东西，比如书，最先放进收纳箱的书在最下面，最后放进收纳箱的书在最上面，如果你想拿书的话， 必须从最上面开始取，否则是无法取出最下面的书籍的。

栈的数据结构就是这样，你把书籍压入收纳箱的操作叫做 压入（push） ，你把书籍从收纳箱取出的操作叫做 弹出（pop） ，它的模型图大概是这样

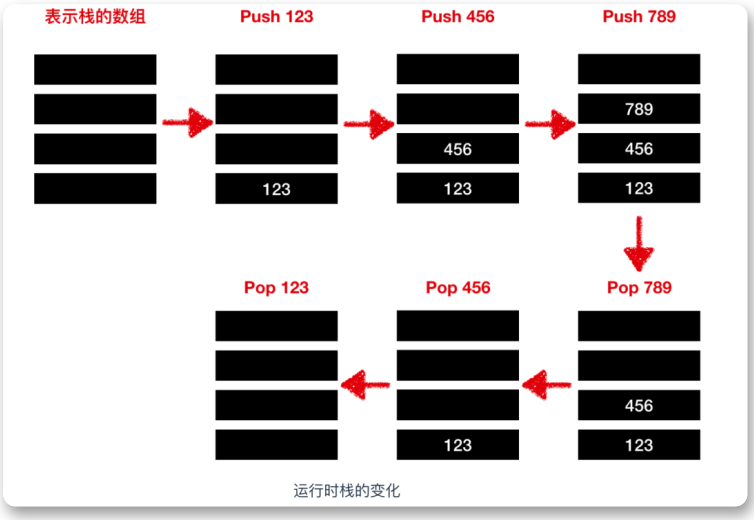


入栈相当于是增加操作，出栈相当于是删除操作，只不过叫法不一样。栈和内存不同，它不需要指定元素的地址。它的大概使用如下

```
// 压入数据
Push(123);
Push(456);
Push(789);

// 弹出数据
j = Pop();
k = Pop();
l = Pop();
```

在栈中，LIFO 方式表示栈的数组中所保存的最后面的数据（Last In）会被最先读取出来（First Out）。



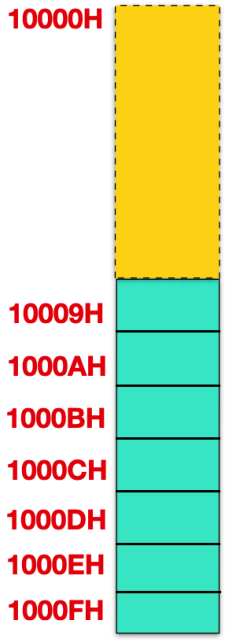
栈和 SS 寄存器

下面我们就通过一段汇编代码来描述一下栈的压入弹出的过程

8086 CPU 提供入栈和出栈指令，最基本的两个是 PUSH(入栈) 和 POP(出栈)。比如 push ax 会把 ax 寄存器中的数据压入栈中，pop ax 表示从栈顶取出数据送入 ax 寄存器中。

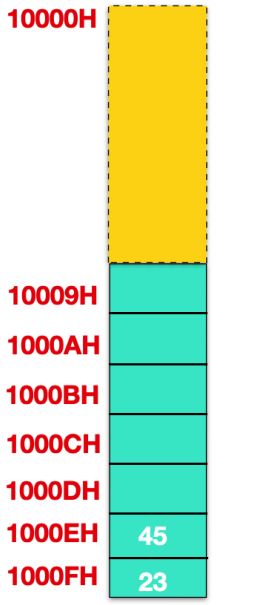
> 这里注意一点：8086 CPU 中的入栈和出栈都是以字为单位进行的。

我这里首先有一个初始的栈，没有任何指令和数据。



栈初始化

然后我们向栈中 push 数据后，栈中数据如下



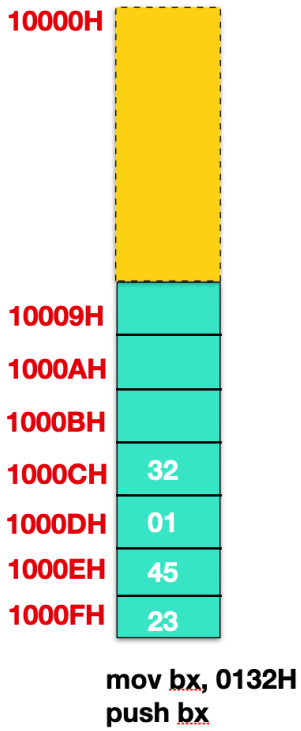
mov ax, 2345H
push ax

涉及的指令有

```
mov ax,2345H
push ax
```

> 注意，数据会用两个单元存放，高地址单元存放高 8 位地址，低地址单元存放低 8 位。

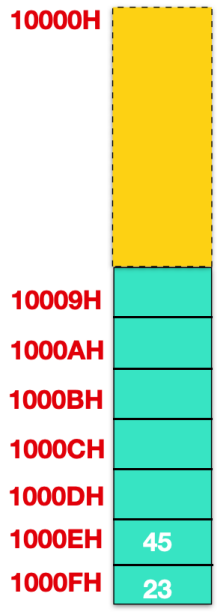
再向栈中 push 数据



其中涉及的指令有

```
mov bx,0132H
push bx
```

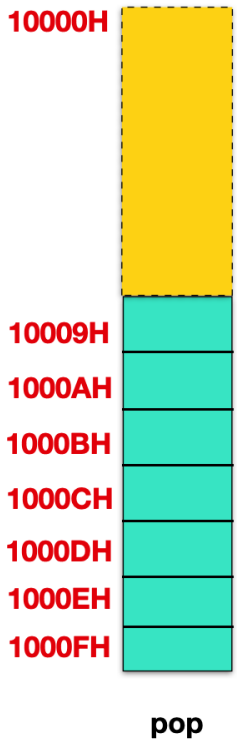
现在栈中有两条数据，现在我们执行出栈操作



其中涉及的指令有

```
pop ax
/* ax = 0132H */
```

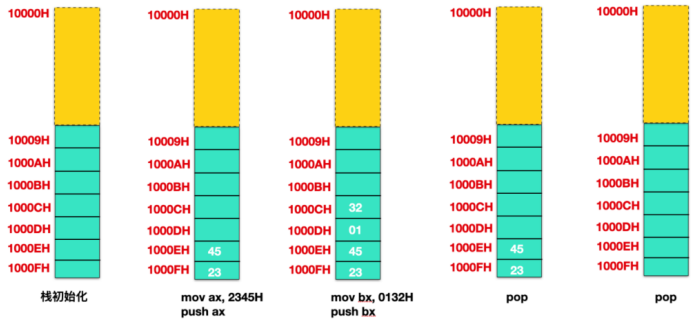
再继续取出数据



涉及的指令有

```
pop bx
/* bx = */
```

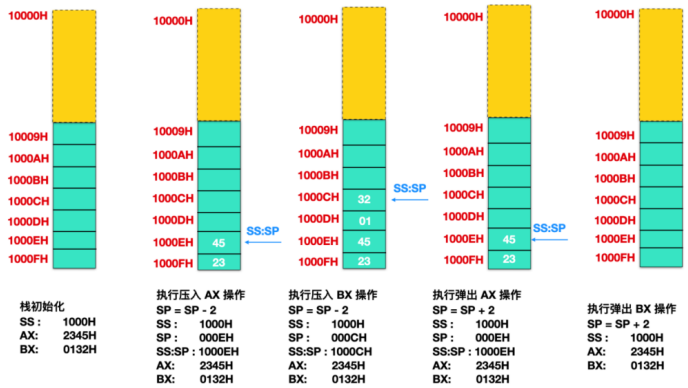
完整的 push 和 pop 过程如下



现在 cxuan 问你一个问题，我们上面描述的是 10000H ~ 1000FH 这段空间来作为 push 和 pop 指令的存取单元。但是，你怎么知道这个栈单元就是 10000H ~ 1000FH 呢？也就是说，你如何选择指定的栈单元进行存取？

事实上，8086 CPU 有一组关于栈的寄存器 SS 和 SP 。SS 是段寄存器，它存储的是栈的基础位置，也就是栈顶的位置，而 SP 是栈指针，它存储的是偏移地址。在任意时刻， SS:SP 都指向栈顶元素。push 和 pop 指令执行时，CPU 从 SS 和 SP 中得到栈顶的地址。

现在，我们可以完整的描述一下 push 和 pop 过程了，下面 cxuan 就给你推导一下这个过程。



上面这个过程主要涉及到的关键变化如下。

当使用「PUSH」指令向栈中压入 1 个字节单元时，SP = SP - 1；即栈顶元素会发生变化；

而当使用「PUSH」指令向栈中压入 2 个字节的字单元时，SP = SP − 2 ；即栈顶元素也要发生变化；

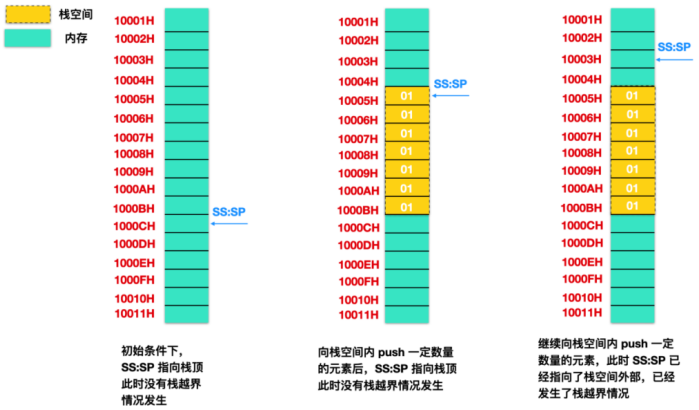
当使用「POP」指令从栈中弹出 1 个字节单元时， SP = SP + 1；即栈顶元素会发生变化；

当使用「POP」指令从栈中弹出 2 个字节单元的字单元时， SP = SP + 2 ；即栈顶元素会发生变化；

栈顶越界问题

现在我们知道，8086 CPU 可以使用 SS 和 SP 指示栈顶的地址，并且提供 PUSH 和 POP 指令实现入栈和出栈，所以，你现在知道了如何能够找到栈顶位置，但是你如何能保证栈顶的位置不会越界呢？栈顶越界会产生什么影响呢？

比如如下是一个栈顶越界的示意图



第一开始，SS：SP 寄存器指向了栈顶，然后向栈空间 push 一定数量的元素后，SS:SP 位于栈空间顶部，此时再向栈空间内部 push 元素，就会出现栈顶越界问题。

栈顶越界是危险的，因为我们既然将一块区域空间安排为栈，那么在栈空间外部也可能存放了其他指令和数据，这些指令和数据有可能是其他程序的，所以如此操作会让计算机 懵逼 。

我们希望 8086 CPU 能自己解决问题，毕竟 8086 CPU 已经是个成熟的 CPU 了，要学会自己解决问题了。

然鹅（故意的），这对于 8086 CPU 来说，这可能是它一辈子的 夙愿 了，真实情况是，8086 CPU 不会保证栈顶越界问题，也就是说 8086 CPU 只会告诉你栈顶在哪，并不会知道栈空间有多大，所以需要程序员自己手动去保证。。。

- EOF -

推荐阅读 — 点击标题可跳转

1、C++ 为什么要弄出虚表这个东西?

2、深入理解 Linux 异步 I/O 框架 io_uring

3、如何正确的理解指针和结构体指针、指针函数、函数指针这些东西?

关注『CPP开发者』

看精选C++技术文章 . 加C++开发者专属圈子

CPP开发者

我们在 Github 维护着 9000+ star 的C语言/C++开发资源。日常分享 C语言 和 C++ 开发…

24篇原创内容

公众号

点赞和在看就是最大的支持❤️

People who liked this content also liked

“C++ 继任者”火到 GitHub 趋势榜一，C++ 之父：规范不足，无法评价

CPP开发者