06 | x86架构：有了开放的架构，才能打造开放的营商环境

2019-04-08 刘超

讲述：刘超
时长 17:52 大小 16.37M

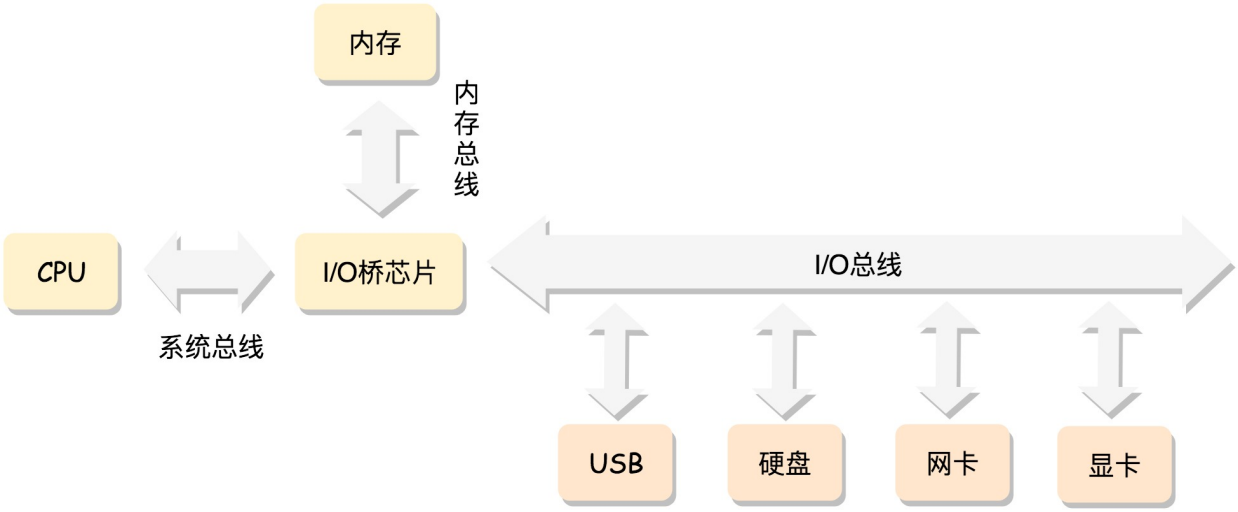
做生意的人最喜欢开放的营商环境，也就是说，我的这家公司，只要符合国家的法律，到哪里做生意，都能受到公平的对待，这样就不用为了适配各个地方的规则煞费苦心，只要集中精力优化自己的服务就可以了。

作为 Linux 操作系统，何尝不是这样。如果下面的硬件环境千差万别，就会很难集中精力做出让用户易用的产品。毕竟天天适配不同的平台，就已经够头大了。x86 架构就是这样一个开放的平台。今天我们就来解析一下它。

计算机的工作模式是什么样的?

还记得咱们攒电脑时买的那堆硬件吗？虽然你可以根据经验，把那些复杂的设备和线安装起来，但是你真的了解它们为什么要这么连接吗？

现在我就把硬件图和计算机的逻辑图对应起来，带你看看计算机的工作模式。



对于一个计算机来讲，最核心的就是**CPU**（Central Processing Unit，中央处理器）。这是这台计算机的大脑，所有的设备都围绕它展开。

对于公司来说，CPU 是真正干活的，将来执行项目都要靠它。

CPU 就相当于咱们公司的程序员，我们常说，二十一世最缺的是什么？是人才！所以，大量水平高、干活快的程序员，才是营商环境中最重要的部分。

CPU 和其他设备连接，要靠一种叫作**总线**（Bus）的东西，其实就是主板上密密麻麻的集成电路，这些东西组成了 CPU 和其他设备的高速通道。

在这些设备中，最重要的是**内存**（Memory）。因为单靠 CPU 是没办法完成计算任务的，很多复杂的计算任务都需要将中间结果保存下来，然后基于中间结果进行进一步的计算。CPU 本身没办法保存这么多中间结果，这就要依赖内存了。

内存就相当于办公室，我们要看看方不方便租到办公室，有没有什么创新科技园之类的。有了共享的、便宜的办公位，公司就有注册地了。

当然总线上还有一些其他设备，例如显卡会连接显示器、磁盘控制器会连接硬盘、USB 控制器会连接键盘和鼠标等等。

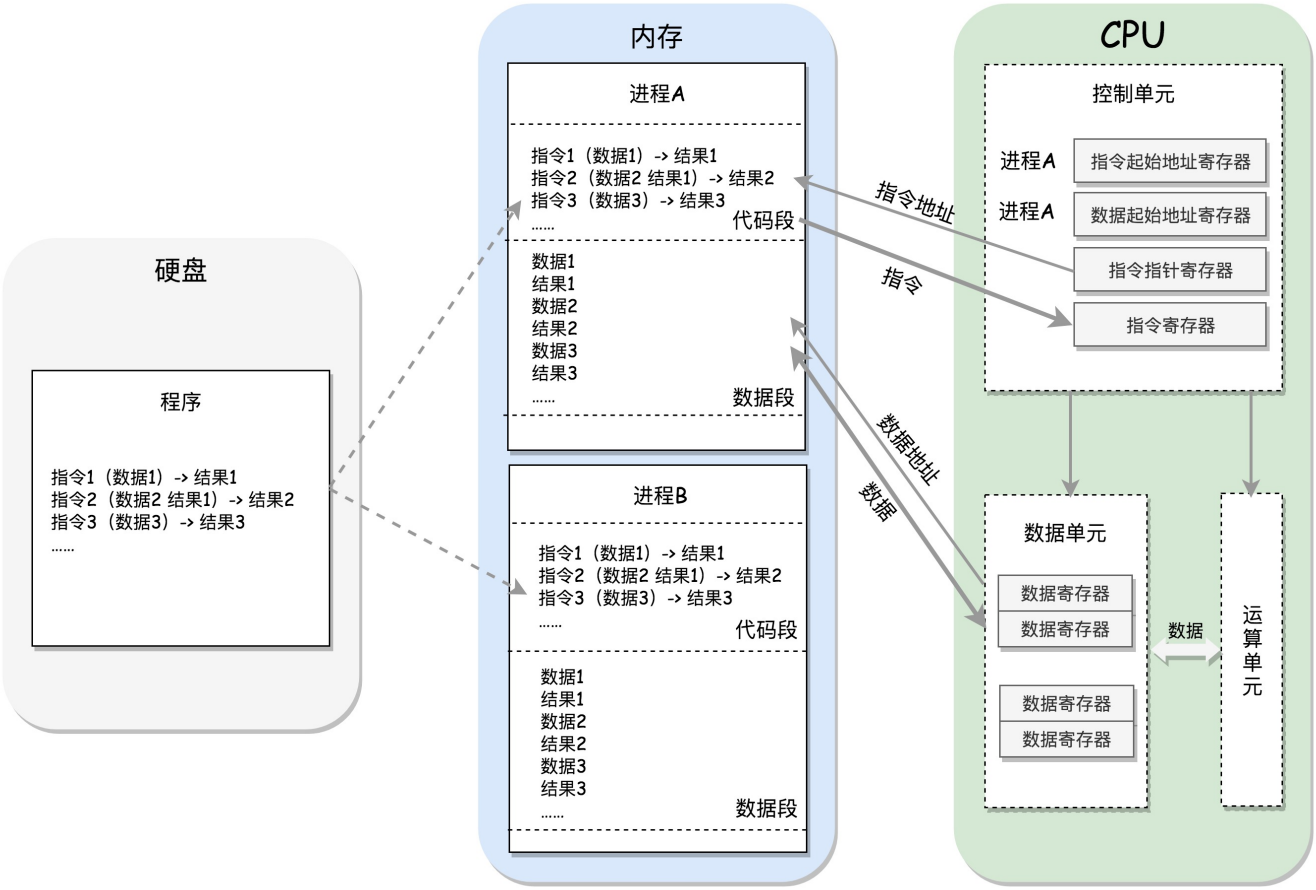
CPU 和内存是完成计算任务的核心组件，所以这里我们重点介绍一下**CPU 和内存是如何配合工作的。**

CPU 其实也不是单纯的一块，它包括三个部分，运算单元、数据单元和控制单元。

运算单元只管算，例如做加法、做位移等等。但是，它不知道应该算哪些数据，运算结果应该放在哪里。

运算单元计算的数据如果每次都要经过总线，到内存里面现拿，这样就太慢了，所以就有了**数据单元。**数据单元包括 CPU 内部的缓存和寄存器组，空间很小，但是速度飞快，可以暂时存放数据和运算结果。

有了放数据的地方，也有了算的地方，还需要有个指挥到底做什么运算的地方，这就是**控制单元。**控制单元是一个统一的指挥中心，它可以获得下一条指令，然后执行这条指令。这个指令会指导运算单元取出数据单元中的某几个数据，计算出个结果，然后放在数据单元的某个地方。



每个项目都有一个项目执行计划书，里面是一行行项目执行的指令，这些都是放在档案库里面的。每个进程都有一个程序放在硬盘上，是二进制的，再里面就是一行行的指令，会操作一些数据。

进程一旦运行，比如图中两个进程 A 和 B，会有独立的内存空间，互相隔离，程序会分别加载到进程 A 和进程 B 的内存空间里面，形成各自的代码段。当然真实情况肯定比我说的要复杂的多，进程的内存虽然隔离但不连续，除了简单的区分代码段和数据段，还会分的更细。

程序运行的过程中要操作的数据和产生的计算结果，都会放在数据段里面。**那 CPU 怎么执行这些程序，操作这些数据，产生一些结果，并写入回内存呢?**

CPU 的控制单元里面，有一个**指令指针寄存器**，它里面存放的是下一条指令在内存中的地址。控制单元会不停地将代码段的指令拿进来，先放入指令寄存器。

当前的指令分两部分，一部分是做什么操作，例如是加法还是位移；一部分是操作哪些数据。

要执行这条指令，就要把第一部分交给运算单元，第二部分交给数据单元。

数据单元根据数据的地址，从数据段里读到数据寄存器里，就可以参与运算了。运算单元做完运算，产生的结果会暂存在数据单元的数据寄存器里。最终，会有指令将数据写回内存中的数据段。

你可能会问，上面算来算去执行的都是进程 A 里的指令，那进程 B 呢？CPU 里有两个寄存器，专门保存当前处理进程的代码段的起始地址，以及数据段的起始地址。这里面写的都是进程 A，那当前执行的就是进程 A 的指令，等切换成进程 B，就会执行 B 的指令了，这个过程叫作**进程切换**（Process Switch）。这是一个多任务系统的必备操作，我们后面有专门的章节讲这个内容，这里你先有个印象。

到这里，你会发现，CPU 和内存来来回回传数据，靠的都是总线。其实总线上主要有两类数据，一个是地址数据，也就是我想拿内存中哪个位置的数据，这类总线叫**地址总线**（Address Bus）；另一类是真正的数据，这类总线叫**数据总线**（Data Bus）。

所以说，总线其实有点像连接 CPU 和内存这两个设备的高速公路，说总线到底是多少位，就类似说高速公路有几个车道。但是这两种总线的位数意义是不同的。

地址总线的位数，决定了能访问的地址范围到底有多广。例如只有两位，那 CPU 就只能认 00，01，10，11 四个位置，超过四个位置，就区分不出来了。位数越多，能够访问的位置就越多，能管理的内存的范围也就越广。

而数据总线的位数，决定了一次能拿多少个数据进来。例如只有两位，那 CPU 一次只能从内存拿两位数。要想拿八位，就要拿四次。位数越多，一次拿的数据就越多，访问速度也就越快。

x86 成为开放平台历史中的重要一笔

那 CPU 中总线的位数有没有个标准呢？如果没有标准，那操作系统作为软件就很难办了，因为软件层没办法实现通用的运算逻辑。这就像很多非标准的元器件一样，你烧你的电路板，我烧我的电路板，谁都不能用彼此的。

早期的 IBM 凭借大型机技术成为计算机市场的领头羊，直到后来个人计算机兴起，苹果公司诞生。但是，那个时候，无论是大型机还是个人计算机，每家的 CPU 架构都不一样。如果一直是这样，个人电脑、平板电脑、手机等等，都没办法形成统一的体系，就不会有我们现在通用的计算机了，更别提什么云计算、大数据这些统一的大平台了。

好在历史将 x86 平台推到了**开放、统一、兼容**的位置。我们继续来看 IBM 和 x86 的故事。

IBM 开始做 IBM PC 时，一开始并没有让最牛的华生实验室去研发，而是交给另一个团队。一年时间，软硬件全部自研根本不可能完成，于是他们采用了英特尔的 8088 芯片作为 CPU，使用微软的 MS-DOS 做操作系统。

谁能想到 IBM PC 卖的超级好，好到因为垄断市场而被起诉。IBM 就在被逼的情况下公开了一些技术，使得后来无数 IBM-PC 兼容机公司的出现，也就有了后来占据市场的惠普、康柏、戴尔等等。

能够开放自己的技术是一件了不起的事。从技术和发展的层面来讲，它会使得一项技术大面积铺开，形成行业标准。就比如现在常用的 Android 手机，如果没有开放的 Android 系统，我们也没办法享受到这么多不同类型的手机。

对于当年的 PC 机来说，其实也是这样。英特尔的技术因此成为了行业的开放事实标准。由于这个系列开端于 8086，因此称为 x86 架构。

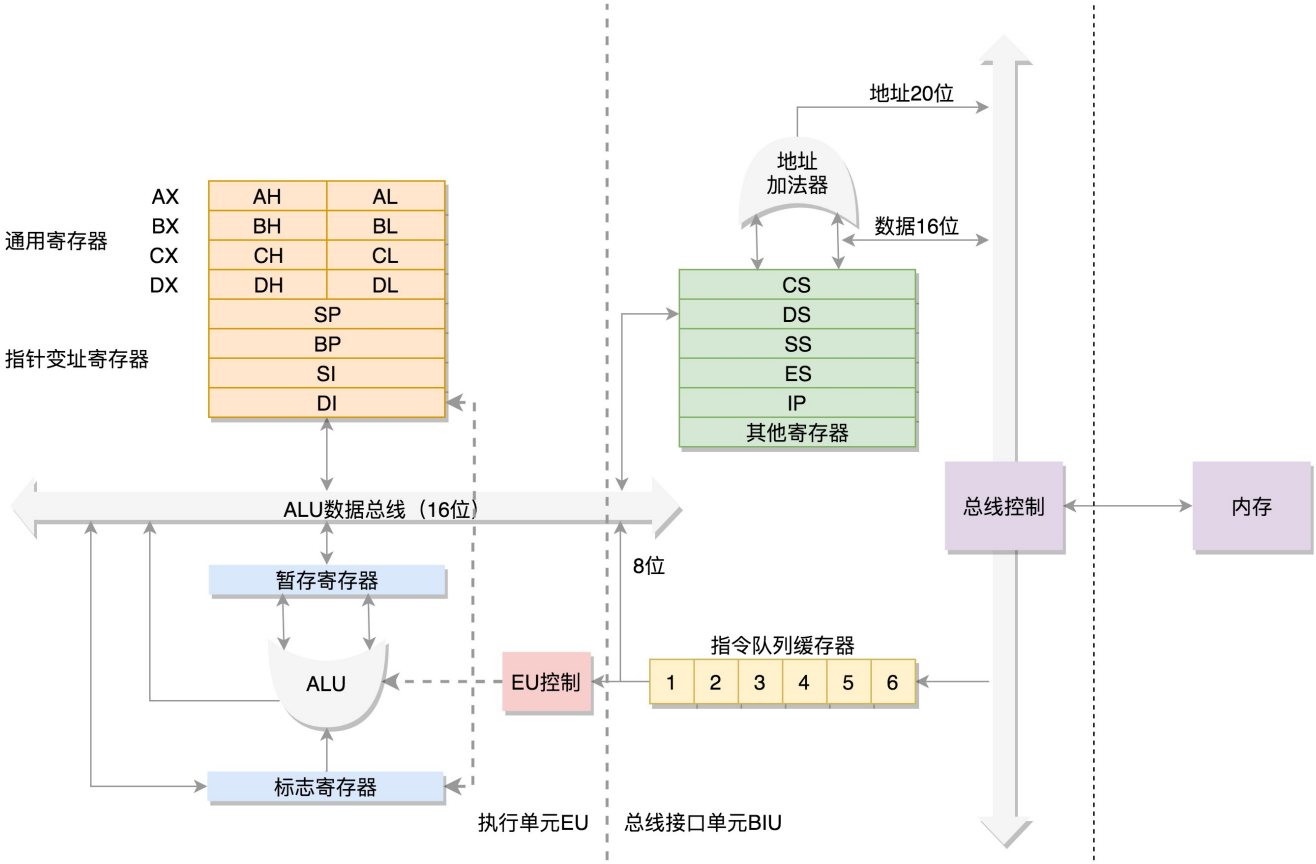
后来英特尔的 CPU 数据总线和地址总线越来越宽，处理能力越来越强。但是一直不能忘记三点，一是标准，二是开放，三是兼容。因为要想如此大的一个软硬件生态都基于这个架构，符合它的标准，如果是封闭或者不兼容的，那谁都不答应。

型号	总线位宽	地址位	寻址空间
8080	8	16	64k（2^16）
8086	16	20	1M（2^20）
8088	8	20	1M
80386	32	32	4G

从 8086 的原理说起

说完了 x86 的历史，我们再来看 x86 中最经典的一款处理器，8086 处理器。虽然它已经很老了，但是咱们现在操作系统中的很多特性都和它有关，并且一直保持兼容。

我们把 CPU 里面的组件放大之后来看。你可以看我画的这幅图。



我们先来看数据单元。

为了暂存数据，8086 处理器内部有 8 个 16 位的通用寄存器，也就是刚才说的 CPU 内部的数据单元，分别是 AX、BX、CX、DX、SP、BP、SI、DI。这些寄存器主要用于在计算过程中暂存数据。

这些寄存器比较灵活，其中 AX、BX、CX、DX 可以分成两个 8 位的寄存器来使用，分别是 AH、AL、BH、BL、CH、CL、DH、DL，其中 H 就是 High（高位），L 就是 Low（低位）的意思。

这样，比较长的数据也能暂存，比较短的数据也能暂存。你可能会说 16 位并不长啊，你可别忘了，那是在计算机刚刚起步的时代。

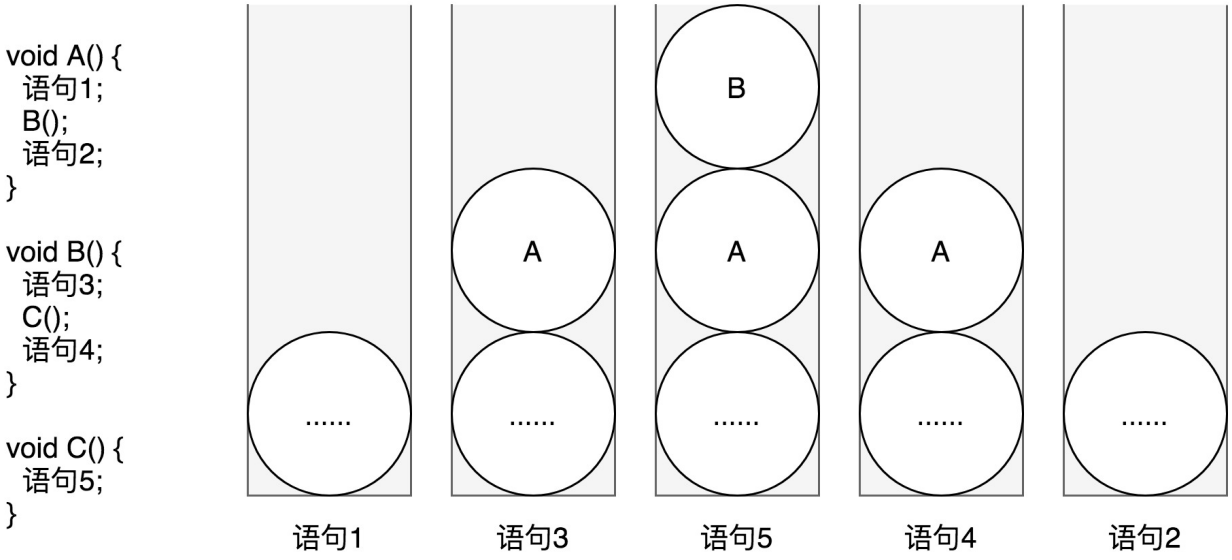
接着我们来看控制单元。

IP 寄存器就是指令指针寄存器（Instruction Pointer Register），指向代码段中下一条指令的位置。CPU 会根据它来不断地将指令从内存的代码段中，加载到 CPU 的指令队列中，然后交给运算单元去执行。

如果需要切换进程呢？每个进程都分代码段和数据段，为了指向不同进程的地址空间，有四个 16 位的段寄存器，分别是 CS、DS、SS、ES。

其中，CS 就是代码段寄存器（Code Segment Register），通过它可以找到代码在内存中的位置；DS 是数据段的寄存器，通过它可以找到数据在内存中的位置。

SS 是栈寄存器（Stack Register）。栈是程序运行中一个特殊的数据结构，数据的存取只能从一端进行，秉承后进先出的原则，push 就是入栈，pop 就是出栈。



凡是与函数调用相关的操作，都与栈紧密相关。例如，A 调用 B，B 调用 C。当 A 调用 B 的时候，要执行 B 函数的逻辑，因而 A 运行的相关信息就会被 push 到栈里面。当 B 调用 C 的时候，同样，B 运行相关信息会被 push 到栈里面，然后才运行 C 函数的逻辑。当 C 运行完毕的时候，先 pop 出来的是 B，B 就接着调用 C 之后的指令运行下去。B 运行完了，再 pop 出来的就是 A，A 接着运行，直到结束。

如果运算中需要加载内存中的数据，需要通过 DS 找到内存中的数据，加载到通用寄存器中，应该如何加载呢？对于一个段，有一个起始的地址，而段内的具体位置，我们称为**偏移量**（Offset）。例如 8 号会议室的第三排，8 号会议室就是起始地址，第三排就是偏移量。

在 CS 和 DS 中都存放着一个段的起始地址。代码段的偏移量在 IP 寄存器中，数据段的偏移量会放在通用寄存器中。

这时候问题来了，CS 和 DS 都是 16 位的，也就是说，起始地址都是 16 位的，IP 寄存器和通用寄存器都是 16 位的，偏移量也是 16 位的，但是 8086 的地址总线地址是 20 位。怎么凑够这 20 位呢？方法就是“**起始地址 *16+ 偏移量**”，也就是把 CS 和 DS 中的值左移 4 位，变成 20 位的，加上 16 位的偏移量，这样就可以得到最终 20 位的数据地址。

从这个计算方式可以算出，无论真正的内存多么大，对于只有 20 位地址总线的 8086 来讲，能够区分出的地址也就 2^20=1M，超过这个空间就访问不到了。这又是为啥呢？如果你想访问 1M+X 的地方，这个位置已经超过 20 位了，由于地址总线只有 20 位，在总线上超过 20 位的部分根本是发不出去的，所以发出去的还是 X，最后还是会访问 1M 内的 X 的位置。

那一个段最大能有多大呢？因为偏移量只能是 16 位的，所以一个段最大的大小是 2^16=64k。

是不是好可怜？对于 8086CPU，最多只能访问 1M 的内存空间，还要分成多个段，每个段最多 64K。尽管我们现在看来这不可想象的小，根本没法儿用，但是在当时其实够用了。

再来说 32 位处理器

当然，后来计算机的发展日新月异，内存越来越大，总线也越来越宽。在 32 位处理器中，有 32 根地址总线，可以访问 2^32=4G 的内存。使用原来的模式肯定不行了，但是又不能完全抛弃原来的模式，因为这个架构是开放的。

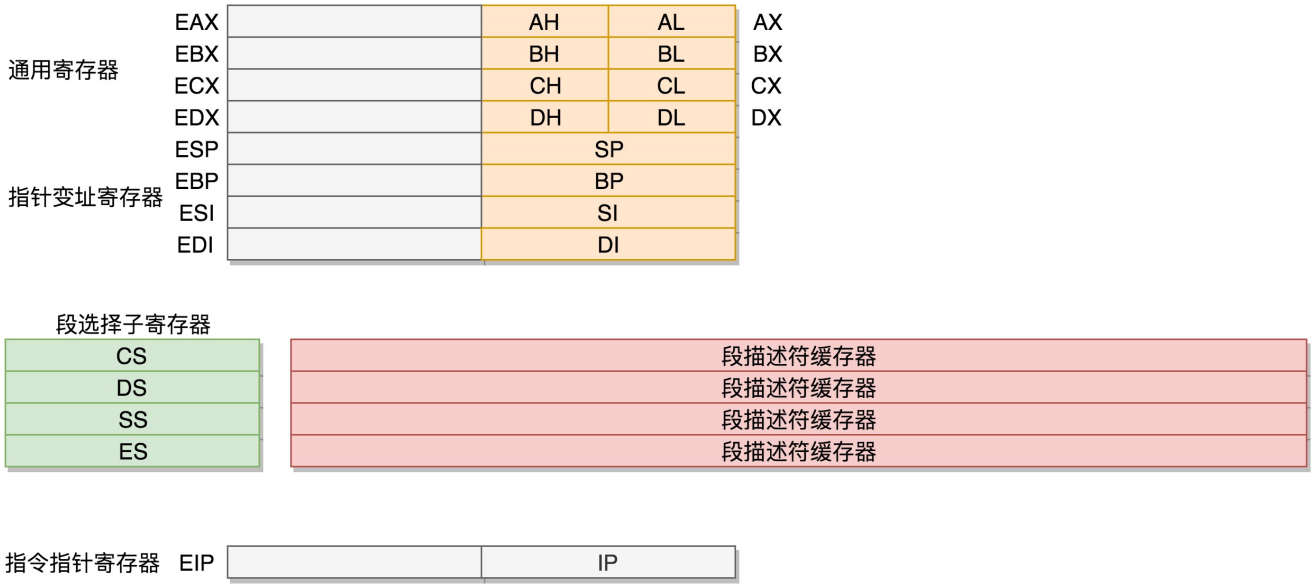
“开放”，意味着有大量其他公司的软硬件是基于这个架构来实现的，不能为所欲为，想怎么改怎么改，一定要和原来的架构兼容，而且要一直兼容，这样大家才愿意跟着你这个开放平台一直玩下去。如果你朝令夕改，那其他厂商就惨了。

如果是不开放的架构，那就没有问题。硬件、操作系统，甚至上面的软件都是自己搞的，你想怎么改就可以怎么改。

我们下面来说说，在开放架构的基础上，如何保持兼容呢？

首先，通用寄存器有扩展，可以将 8 个 16 位的扩展到 8 个 32 位的，但是依然可以保留 16 位的和 8 位的使用方式。你可能会问，为什么高 16 位不分成两个 8 位使用呢？因为这样就不兼容了呀！

其中，指向下一条指令的指令指针寄存器 IP，就会扩展成 32 位的，同样也兼容 16 位的。



而改动比较大，有点不兼容的就是**段寄存器**（Segment Register）。

因为原来的模式其实有点不伦不类，因为它没有把 16 位当成一个段的起始地址，也没有按 8 位或者 16 位扩展的形式，而是根据当时的硬件，弄了一个不上不下的 20 位的地址。这样每次都要左移四位，也就意味着段的起始地址不能是任何一个地方，只是能整除 16 的地方。

如果新的段寄存器都改成 32 位的，明明 4G 的内存全部都能访问到，还左移不左移四位呢？

那我们索性就重新定义一把吧。CS、SS、DS、ES 仍然是 16 位的，但是不再是段的起始地址。段的起始地址放在内存的某个地方。这个地方是一个表格，表格中的一项一项是**段描述符**（Segment Descriptor）。这里面才是真正的段的起始地址。而段寄存器里面保存的是在这个表格中的哪一项，称为**选择子**（Selector）。

这样，将一个从段寄存器直接拿到的段起始地址，就变成了先间接地从段寄存器找到表格中的一项，再从表格中的一项中拿到段起始地址。

这样段起始地址就会很灵活了。当然为了快速拿到段起始地址，段寄存器会从内存中拿到 CPU 的描述符高速缓存器中。

这样就不兼容了，咋办呢？好在后面这种模式灵活度非常高，可以保持将来一直兼容下去。前面的模式出现的时候，没想到自己能够成为一个标准，所以设计就没这么灵活。

因而到了 32 位的系统架构下，我们将前一种模式称为**实模式**（Real Pattern），后一种模式称为**保护模式**（Protected Pattern）。

当系统刚刚启动的时候，CPU 是处于实模式的，这个时候和原来的模式是兼容的。也就是说，哪怕你买了 32 位的 CPU，也支持在原来的模式下运行，只不过快了一点而已。

当需要更多内存的时候，你可以遵循一定的规则，进行一系列的操作，然后切换到保护模式，就能够用到 32 位 CPU 更强大的能力。

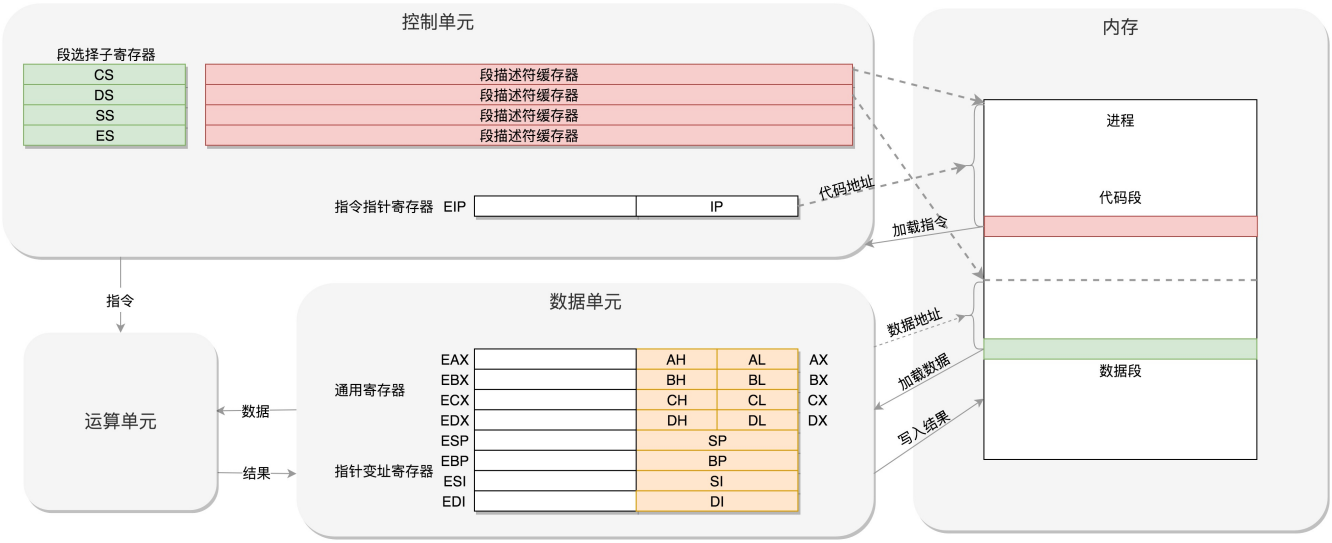
这也就是说，不能无缝兼容，但是通过切换模式兼容，也是可以接受的。

在接下来的几节，我们就来看一下，CPU 如何从启动开始，逐渐从实模式变为保护模式的。

总结时刻

这一节，我们讲了 x86 架构。在以后的操作系统讲解中，我们也是主要基于 x86 架构进行讲解，只有了解了底层硬件的基本工作原理，将来才能理解操作系统的工作模式。

x86 架构总体来说还是很复杂的，其中和操作系统交互比较密切的部分，我画了个图。在这个图中，建议你重点牢记这些寄存器的作用，以及段的工作模式，后面我们马上就能够用到了。



课堂练习

操作这些底层的寄存器往往需要使用汇编语言，操作系统的一些底层的模块也是用汇编语言写的，因而你需要简单回顾一些汇编语言中的一些简单的命令的作用。所以，今天给你留个练习题，简单了解一下这些命令。

mov, call, jmp, int, ret, add, or, xor, shl, shr, push, pop, inc, dec, sub, cmp。

欢迎留言和我分享你的疑惑和见解，也欢迎你收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习、进步。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　05 | 学会几个系统调用：咱们公司能接哪些类型的项目？

下一篇　07 | 从BIOS到bootloader：创业伊始，有活儿老板自己上

精选留言 (95)

写留言

一命赌快乐 置顶

2019-04-08

84

move a b :把b值赋给a,使a=b
call和ret :call调用子程序，子程序以ret结尾
jmp :无条件跳
int :中断指令
add a b : 加法,a=a+b...

展开

作者回复: 赞，认真学习的典范

Colin.Tao
2019-04-08

22

妥妥地复习了一把“计算机体系结构”

展开

作者回复: 后面很多地方要用到寄存器，所以必须先讲一下

Frank_51
2019-04-08

19

作为一个搞硬件的，这章是我学习最轻松的，推荐一个入门的系统学习汇编的视频课，网易云课堂上的一个课程，《汇编从零开始到C语言》，通俗易懂，小白入门必备

展开

作者回复: 推荐书籍

why
2019-04-08

12

Guide to x86 Assembly: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

TeFuir
2019-04-09

9

老师您好，我想问下 为什么高16位分成两个8位就不兼容列呀。

展开

MJ
2019-04-08

9

“起始地址 *16+ 偏移量”，也就是把 CS 和 DS 中的值左移 4 位，变成 20 位的，加上 16 位的偏移量，这样就可以得到最终 20 位的数据地址。

求问老师，左移四位，如何理解?

作者回复: 就是乘以十六

why
2019-04-10

8

- CPU 包括: 运算单元, 数据单元, 控制单元
  - 运算单元 不知道算哪些数据, 结果放哪
  - 数据单元 包括 CPU 内部缓存和寄存器, 暂时存放数据和结果
  - 控制单元 获取下一条指令, 指导运算单元取数据, 计算, 存放结果
- 进程包含代码段, 数据段等, 以下为 CPU 执行过程:...

展开

谭
2019-04-08

8

老师讲地太精彩了 点赞。由于基础弱，还有几个问题希望老师万忙中答疑一下，谢谢：

问题1：程序编译成二进制代码的时候，包含有指令起始地址吗？若包含那么后续每一行指令的涉及到的指令地址是计算出来的？ 或者说加载进程的程序的时候才会确定起始地址？很好奇这个指令的指针寄存器里的值是什么时候、怎么放进去的？...

展开

作者回复: 1.会有指令起始地址，后面讲ELF格式的时候会说这个事情。当执行一个程序的时候，会加载ELF格式，加载的时候会设置指令指针

2.多线程共享同一个进程内存空间，所以代码段的起始地址还是一样的。只不过每个线程执行不同的func，指令指针会不一样，在内核中，线程也是有独立的task_struct

3.寄存器是有限的，如果把程序编译成汇编看的话，再大的数据，也是要转换为对寄存器的操作。当然寄存器里面可以包含对内存的访问地址，这样内存里面的数据就很多了

CHEN
2019-04-08

4

已经有童鞋回复，再补充点
MOV: load value. this instruction name is misnomer, resulting in some confusion (data is not movedbut copied), in other architectures the same instructions is usually named “LOAD” and/or “STORE”or something like that.One important thing: if you set the low 16-bit part of a 32-bit register in 32-bit mode, the high...
展开

YoungMarsh...
2019-04-08

4

几乎完全看懵了，发现基础知识极其匮乏，努力补课，夯实根基太重要了。
展开

Pluto
2019-04-08

4

原来 x86 架构是指 8086，而 x86 是代表 32 位操作系统是因为 80386，原来这两个 x86 不是同一个意思啊，以前学操作系统的时候一直想不明白 x86 为什么是指代 32 位操作系统
展开

CHEN
2019-04-08

3

补充2
ADD: add two values.
OR: logical “or”.
XOR op1, op2:is in fact just “eXclusive OR”,but the compilers often use it instead of MOV EAX, 0—again becauseit is a slightly shorter opcode (2 bytes for XOR...
展开

TinnyFlam...
2019-04-08

3

x86是甜蜜的历史包袱，它的兼容性让它一统市场，但是当时很多性能上更好的尝试在商业上都失败了，因为兼容性的问题客户不买单……

作者回复: 是的，所以兼容比优雅要重要

Luciano李...
2019-04-08

3

准时

展开

楽□天
2019-05-08

2

那64位的还有实模式吗?

展开

布凡
2019-04-12

2

本文结构：https://www.cnblogs.com/bindot/p/linux6.html

展开

一笔一画
2019-04-10

2

“段的起始地址放在内存的某个地方。”
请问下老师，这个地方具体在哪里？有什么方法设置和查看吗？跟elf格式有关系吗？

展开

作者回复: 段的起始地址后面会特别讲，linux的处理有些特别，其实不分段，是linux事先放在初始化为固定值的。和elf没有关系。elf看到的是虚拟地址

尚墨
2019-04-10

2

上周看了 Go 夜读社区 《Go Plan 9汇编入门》 https://www.bilibili.com/video/av46494102 听的云里雾里的，在来看这篇文章好像感觉出了点门道。

作者回复: 呜哇，没想到能和go联系在一起，看来底层原理比较容易互通

小智e
2019-04-08

2

哇，超哥这么一讲，温习了大二学的数字逻辑，想起当时设计模型机的痛苦，也发现当初学的知识是有用的。还有，超哥画的图太棒了

嘉木
2019-05-05

1

sp是栈指针，ss是栈寄存器 ，这两个有什么区别和关联呢?

展开

作者回复: 一个是段的起始地址，一个是栈顶指针