

作者：道哥，10+年的嵌入式开发老兵。

公众号：【[IOT物联网小镇](#)】，专注于：C/C++、Linux操作系统、应用程序设计、物联网、单片机和嵌入式开发等领域。 公众号回复【[书籍](#)】，获取 Linux、嵌入式领域经典书籍。

转载：欢迎转载文章，转载需注明出处。

【Linux 从头学】是什么
古老的 Intel8086 处理器
主存储器是什么？
寄存器是什么？
三个总线
CPU 如何对内存进行寻址？
我们是如何控制 CPU 的？
CPU 执行指令流程

【Linux 从头学】是什么

这两年多以来，我的本职工作重心一直是在 [x86 Linux](#) 系统这一块，从驱动到中间层，再到应用层的开发。

随着内容的不断扩展，越发觉得之前很多[基础](#)的东西都差不多忘记了，比如下面这张表(《深入理解 LINUX 内核》第 47 页)：

段	Base	G	Limit	S	Type	DPL	D/B	P
用户代码段	0x00000000	1	0xffff	1	10	3	1	1
用户数据段	0x00000000	1	0xffff	1	2	3	1	1
内核代码段	0x00000000	1	0xffff	1	10	0	1	1
内核数据段	0x00000000	1	0xffff	1	2	0	1	1

这张表描述了 Linux 系统中几个[段描述符](#)信息。

[数据段](#)和[代码段](#)，仔细看一下相关书籍就知道这些描述符代表什么意思，但是：

为什么这几个段的 [Base](#) 地址都是 0x00000000？

为什么 [Limit](#) 都是 0xfffff？

公众号【IOT物联网小镇】

为什么它们的 [Type](#) 类型和[优先级 DPL](#) 又各不相同？

如果没有对 x86 平台的一些基础知识的理解，要啃完这本书真的是[挺费力气](#)的！

更要命的是，随着 Linux 内核代码的体积不断膨胀，最新的 [5.13](#) 版本压缩档已经是一百多兆了：

linux-5.13.tar.gz	28-Jun-2021 05:34	181M
linux-5.13.tar.sign	28-Jun-2021 05:34	985
linux-5.13.tar.xz	28-Jun-2021 05:34	114M

这么一个庞然大物，如何下手才能真正的学好 Linux 呢？！

即便是从 [Linux 0.11](#) 版本开始，其中的很多代码看起来也是非常费劲的！

周末在整理一些吃灰的书籍时，发现几本以前看过的好书：王爽的《汇编语言》，李忠的《从实模式到保护模式》，马朝晖翻译的《汇编语言程序设计》等等。

都是非常-非常-老的书籍，再次翻了一下，真心觉得内容[写得真好](#)！

对一些概念、原理、设计思路的描述，清晰而透彻。

Linux 系统中的很多关于分段、内存、寄存器相关的设计，都可以在这些书籍中找到基础支撑。

于是乎，我就有了一个[想法](#)：是否可以把这些书籍中，与 Linux 系统相关的内容进行一次重读和整理，但[绝不是](#)简单的知识搬运。

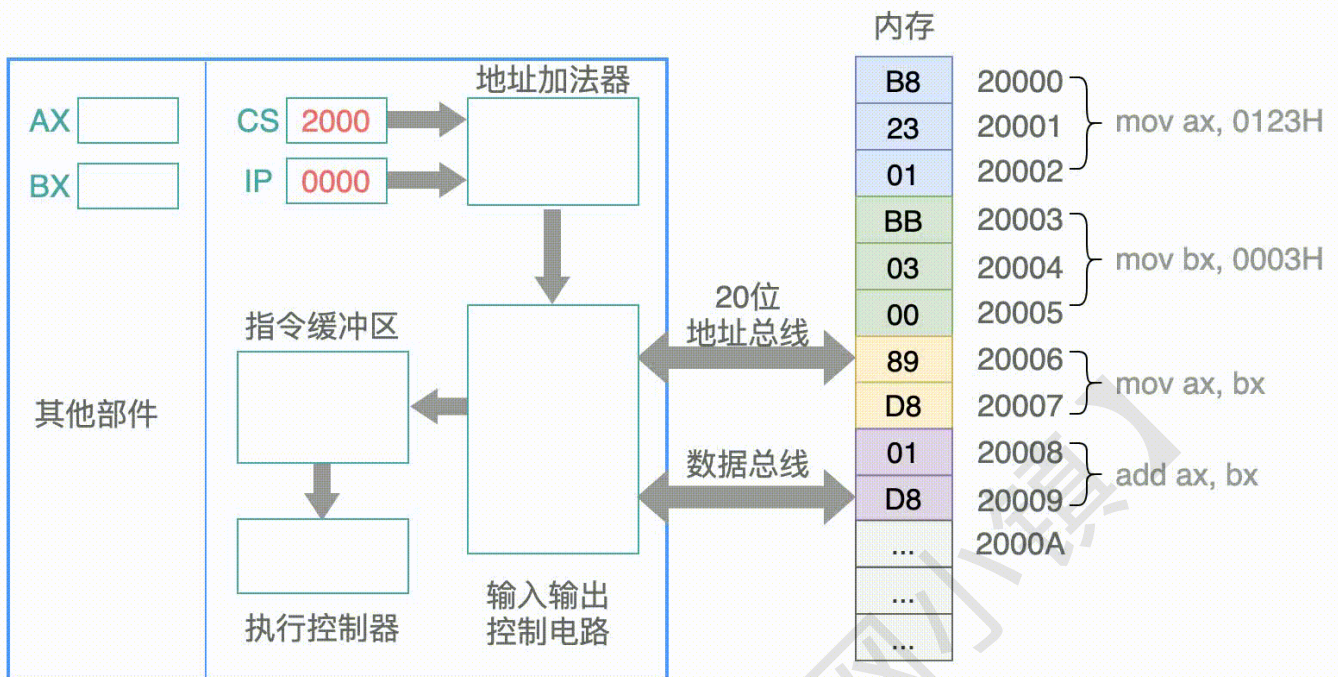
考虑了一下，大概有下面几个想法：

1. 先确定最终目标的目标：学习 Linux 操作系统；
2. 这几本书写的都是汇编语言，以及比较基础的底层知识。我们会淡化汇编语言部分，把重点放在与 Linux 操作系统有关联的原理部分；
3. 不会严格按照书中的内容、顺序来输出文章，而是把几本书中内容相关的部分放在一起学习、讨论；
4. 有些内容，可以与 Linux 2.6 版本中的相关部分进行对比分析，这样的话在以后学习 Linux 内核部分时，可以找到底层的支撑；
5. 最后，希望我自己能坚持这个系列，也算是给自己的一个梳理吧。

一句话：[以基础知识为主](#)！

作为开篇第一章，本文将会描述下面这张图的执行步骤：

初始状态: CS:IP = 2000: 0000



现在就开始吧！

古老的 Intel8086 处理器

8086 是 Intel 公司的第一款 16 位处理器，诞生于 1978 年，应该比各位小伙伴的年龄都大一些。

在 Intel 公司的所有处理器中，它占有很重要的地位，是整个 Intel 32 位架构处理器(IA-32)的**开山鼻祖**。

那么，问题来了，**什么叫 16 位的处理器？**

有些人会把**处理器的位数**与**地址总线**的位数搞混在一起！

我们知道，CPU 在访问内存的时候，是通过**地址总线**来传送**物理地址**的。

8086 CPU 有 20 位的地址线，可以传送 20 位地址。

每一根地址线都表示一个 bit，那么 20 个 bit 可以表示的最大值就是 **2 的 20 次方**。

也就是说：最大可以定位到 1M 地址的内存，这称作 CPU 的**寻址能力**。

但是，8086 处理器却是 16 位的，因为：

1. 运算器一次最多可以处理 16 位的数据；
2. 寄存器的最大宽度为 16 位；
3. 寄存器和运算器之间的通路为 16 位；

也就是说：在 8086 处理器的内部，能够一次性**处理、传输、暂时存储**的最大长度是 16 位，因此，我们说它是 **16 位结构的 CPU**。

主存储器是什么？

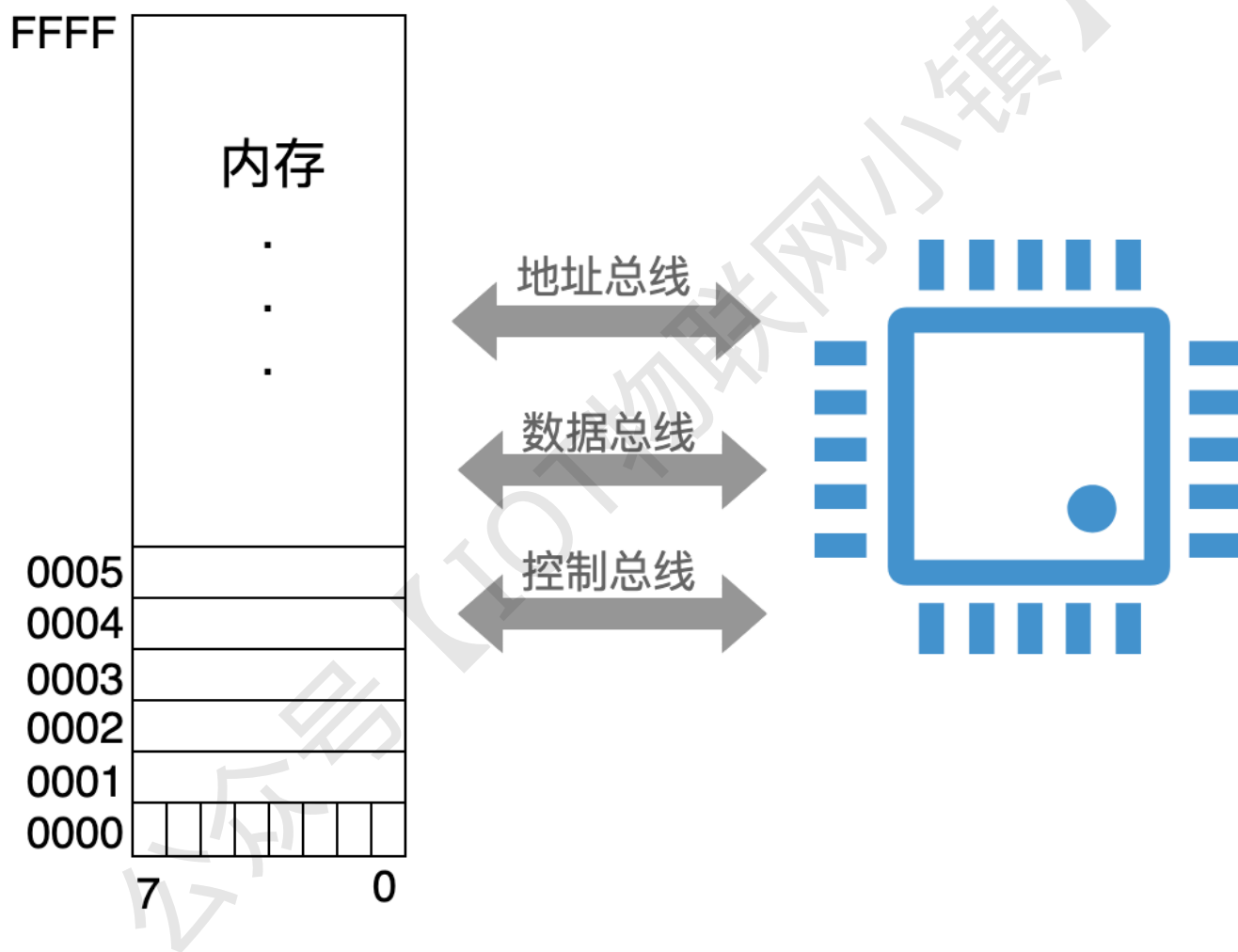
计算机的本质就是对数据的存储和处理，那么参与计算的数据是从哪里来的呢？那就是一个称作 **存储器(Storage 或 Memory)** 的物理器件。

从广义上来说，只要能存储数据的器件都可以称作存储器，比如：**硬盘、U盘**等。

但是，在计算机**内部**，有一种专门与 CPU 相连接，用来存储正在执行的程序 and 数据的存储器，一般称作**内存储器或者主存储器**，简称：**内存或主存**。

内存按照**字节**来组织，单次访问的**最小单位**是 1 个字节，这是最基本的**存储单元**。

每一个存储单元，也就是一个字节，都对应着一个地址，如下图所示：



CPU 就通过**地址总线**来确定：对内存中的哪一个存储单元中的数据进行访问。

第 1 个字节的地址是 0000H，第 2 个字节的地址是 0001H，后面以此类推。

图中的这个内存，最大存储单元的地址是 FFFFH，换算成十进制就是 65535，因此这个内存的容量是 65536 字节，也就是 64 KB。

这里有一个**原子操作**的问题可以考虑一下。

在 Linux 内核代码中，很多地方使用了原子操作，比如：**互斥锁的实现代码**。

公众号【IOT物联网小镇】

为什么原子操作需要对变量的类型限制为 int 型呢？这就涉及到对内存的读写操作了。

尽管内存的**最小**组成单位是**字节**，但是，经过精心的设计和安排，不同位数的 CPU，能够按照**字节**、**字**、**双字**进行访问。

换句话说，仅通过单次访问，16 位处理器就能处理 16 位的二进制数，32 位处理器就能处理 32 位的二进制数。

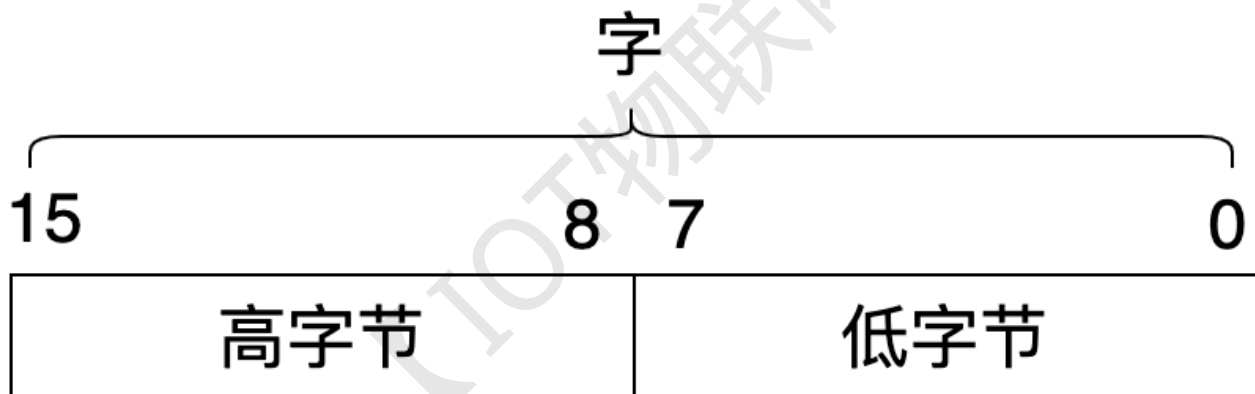
寄存器是什么？

在 CPU 内部，一些都是代表 0 或 1 的电信号，这些二进制数字的**一组电信号**出现在处理器内部线路上，它们是一排高低电平的组合，代表着二进制数中的每一位。

在处理器内部，必须用一个称为**寄存器**的电路把这些数据锁存起来。

因此，寄存器本质上也属于存储器的一种。只不过它们位于**处理器的内部**，CPU 访问寄存器比访问内存的速度更快。

处理器总是很忙的，在它操作的过程中，所有数据在寄存器里面只能是**临时**存在一小会，然后再被送往别处，这就是为什么它被叫做“**寄存器**”。



8086 中的寄存器都是 16 位的，可以存放 2 个字节，或者说 1 个字。**高**字节在前(bit8 ~ bit15)，**低**字节在后(bit0 ~ bit7)。

8086 中有下面这些寄存器：

通用寄存器

AX

AH

AL

BX

BH

BL

CX

CH

CL

DX

DH

DL

SI

DI

BP

SP

段寄存器

CS

DS

ES

SS

指令指针寄存器

IP

刚才说了，这些寄存器都是 16 位的。由于需要与以前更古老的处理器兼容，其中的 4 个寄存器：AX、BX、CX、DX 还可以当成 2 个 8 位的寄存器来使用。

比如：AX 代表一个 16 位的寄存器，AH、AL 分别代表一个 8 位的寄存器。

mov AX, 5D 表示把 005D 送入 AX 寄存器(16 位)

mov AL, 5D 表示把 5D 送入 AL 寄存器(8 位)

三个总线

当我们启动一个应用程序的时候，这个程序的代码和数据都被加载到物理内存中。

CPU 无论是读取指令，还是操作数据，都需要与内存进行信息的交互：

1. 确定存储单元的地址(地址信息);
2. 器件的选择，读或写的命令(控制信息);
3. 读或写的数据(数据信息);

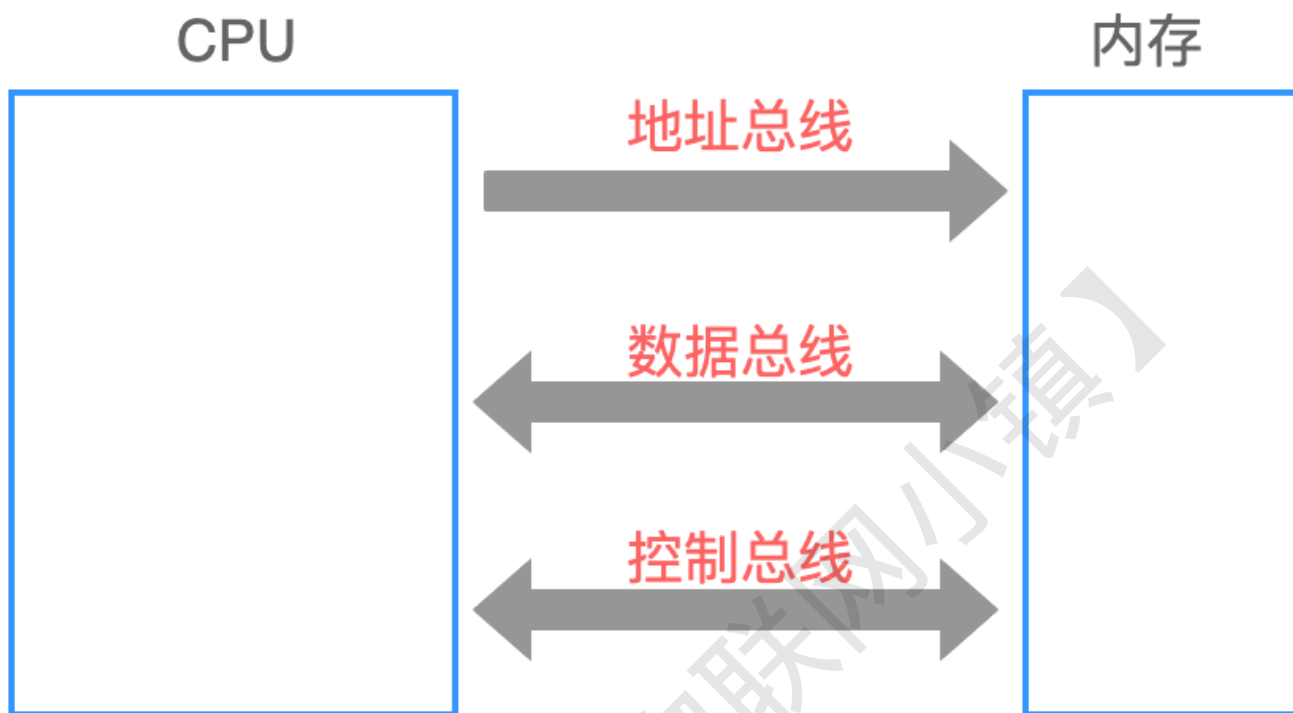
在计算机中，有专门连接 CPU 和其他芯片的数据，称为总线。

从逻辑上来分类，包括下面 3 种总线：

地址总线：用来确定存储单元的地址；

控制总线：CPU 对外部期间进行控制；

数据总线：CPU 与内存或其他器件之间传送数据；



8086 有 20 根地址线，称作地址总线的宽度，它可以寻址 2 的 20 次方个内存单元。

同样的道理，8086 数据总线的宽度是 16，也就是一次性可以传送 16 bit 的数据。

控制总线决定了 CPU 可以对外进行多少种控制，决定了 CPU 对外部器件的控制能力。

CPU 如何对内存进行寻址？

在 Linux 2.6 内核代码中，编译器产生的地址叫做虚拟地址(也称作：逻辑地址)，这个逻辑地址经过段转换之后，变成线性地址，线性地址再经过分页转换，就得到最终物理内存上的物理地址。



还记得文章开头的那张[段描述符](#)的表格吗？

其中的代码段和数据段描述符的[起始](#)地址都是 $0x00000000$ ，也就是说：在数值上[虚拟地址](#)和转换后的[线性地址](#)是相等的（稍后就会明白为什么是这样）。

我们再来看看一下 8086 中更简单的地址转换。

刚才说到，内存是一个[线性](#)的存储器件，CPU 依赖地址来[定位](#)每一个存储单元。

公众号【IOT物联网小镇】

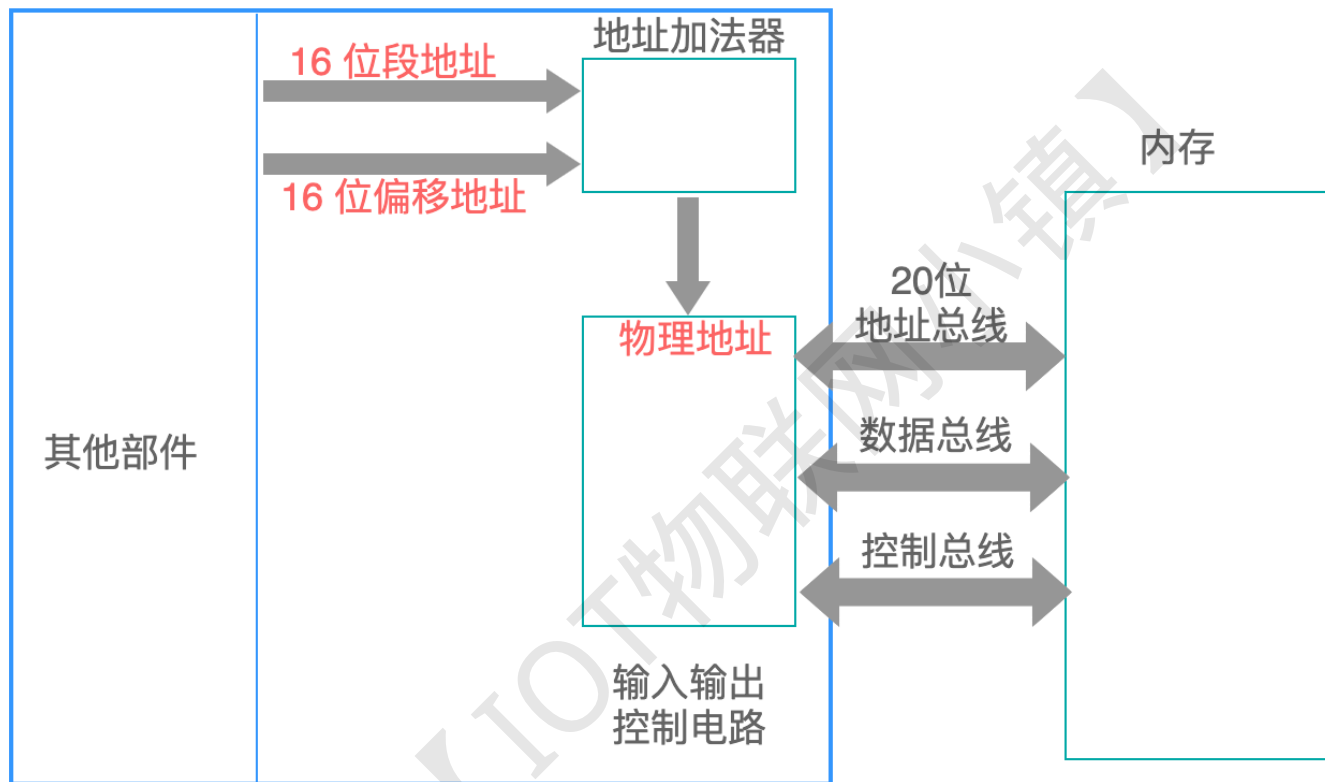
对于 8086 CPU 来说，它有 20 根地址线，可以传送 20 位地址，达到 1MB 的寻址能力。

但是 8086 又是 16 位的结构，在内部一次性处理、传输、暂时存储的地址只有 16 位。

从内部结构来看，如果将地址从内部简单的发出到地址总线上，只能送出 16 位的地址，这样的话，寻址能力只有 64KB。

那么应该怎样才能充分利用 20 根地址线呢？

8086 CPU 采用: 在内部使用两个 16 位地址合成的方法，来形成一个 20 位的物理地址，如下所示：



第一个 16 位的地址称为段地址，第二个 16 位的地址称为偏移地址。

地址加法器采用下面的这个公式，来“合成”得到一个 20 位的物理地址：

$$\text{物理地址} = \text{段地址} \times 16 + \text{偏移地址}$$

例如：我们编写的程序，在加载到内存中之后，放在一个内存空间中。

CPU 在执行这些指令的时候，把 CS 寄存器当做段寄存器，把 IP 寄存器当做偏移寄存器，然后计算 $CS \times 16 + IP$ 的值，就得到了指令的物理地址。

从以上的描述中可以看出：8086 CPU 似乎是因为寄存器无法直接输出 20 位的物理地址，不得已才使用这样的地址合成方式。

其实更本质的原因是：8086 CPU 就是想通过 基地址 + 偏移量 的方式来对内存进行寻址(这里的基地址，就是段地址左移 4 位)。

也就是说，即使 CPU 有能力直接输出一个 20 位的地址，它仍然可能会采用 基地址 + 偏移量 的方式来进行内存寻址。

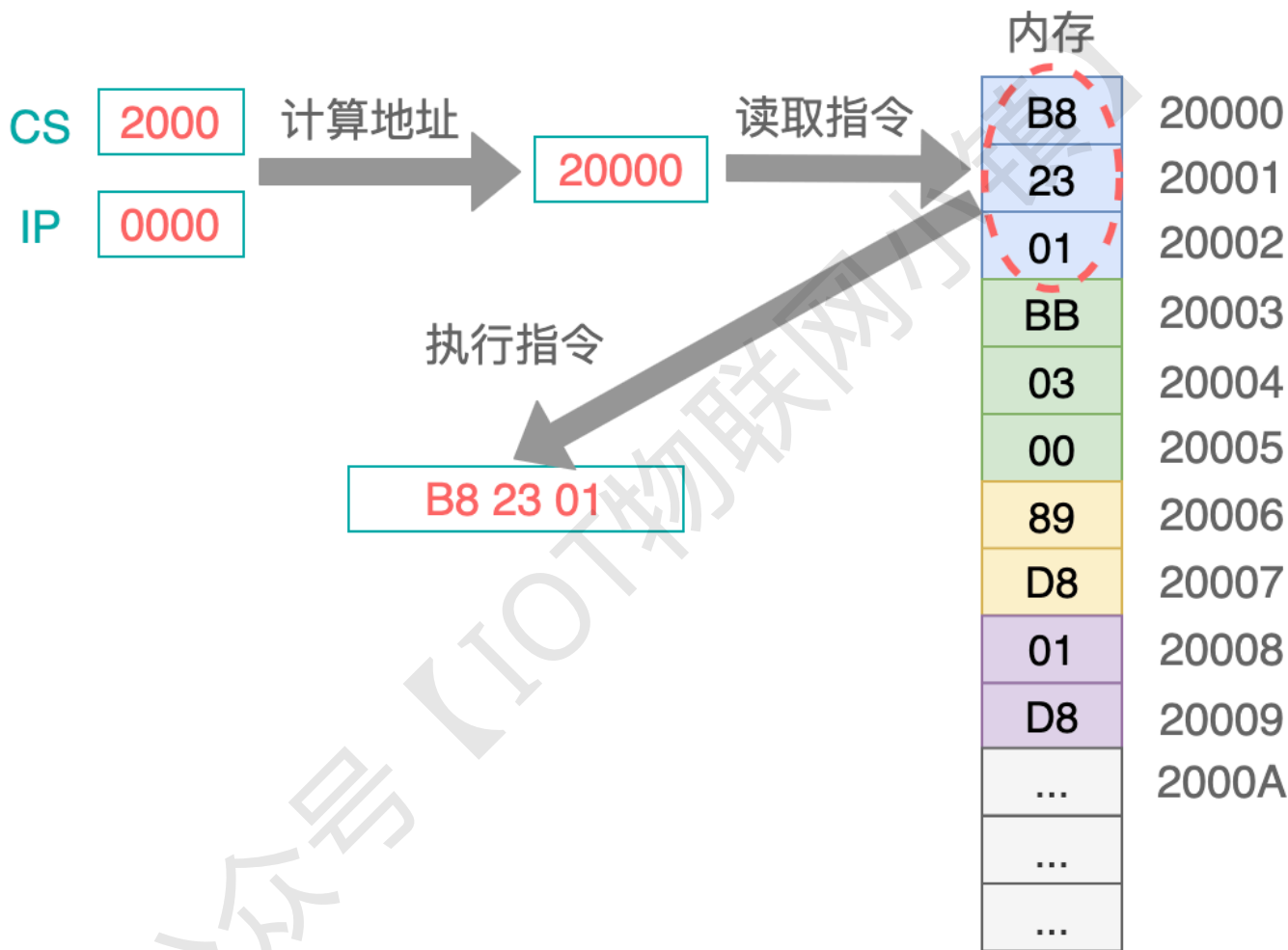
公众号【IOT物联网小镇】

想一下：我们在 Linux 系统中编译一个库文件的时候，一般都会在编译选项中添加 `-fPIC` 选项，表示编译出来的动态库是地址无关的，在被加载到内存时需要被重定位。

而基地址+偏移量的寻址模式，就为重定位提供了底层支撑。

我们是如何控制 CPU 的？

CPU 其实是一个很纯粹、很呆板的一个东西，它唯一做的事情就是：到 `CS:IP` 这两个寄存器指定的内存单元中取出一条指令，然后执行这条指令：



当然了，还需要预先定义一套指令集，在内存中的指令区中，存储的都必须是合法的指令，否则 CPU 就不认识了。

每一条指令都是用某些特定的数(指令码)来指示 CPU 进行特定的操作。

CPU 认识这些指令，一看到这些指令码，CPU 就知道这个指令码后面还有几个字节的操作数、需要进行什么样的操作。

例如：指令码 `F4H` 表示让处理器停机，当 CPU 执行这条指令的时候，就停止工作。

公众号【IOT物联网小镇】

(其实这里说 CPU 已经有点不准确了, 因为 CPU 是囊括了很多器件的一个整体, 也许这里说 CPU 中的执行单元会更准确些。)

另外有一点可以提前说一下: 内存中的一切都是数据, 至于把其中的哪一部分数据当做指令来执行, 哪一部分数据当做被指令操作的“变量”, 这完全是由操作系统的设计者来规划的。

在 8086 处理器的层面来说, 只要是 CS:IP “指向”的内存区域, 都被当做指令来执行。

从以上描述可以看出: 在 CPU 中, 程序员能够用指令读写的器件只有寄存器, 我们可以通过改变寄存器中的内容, 来实现对 CPU 的控制。

更直白的说就是: 我们可以通过改变 CS、IP 寄存器中的内容, 来控制 CPU 执行目标指令。

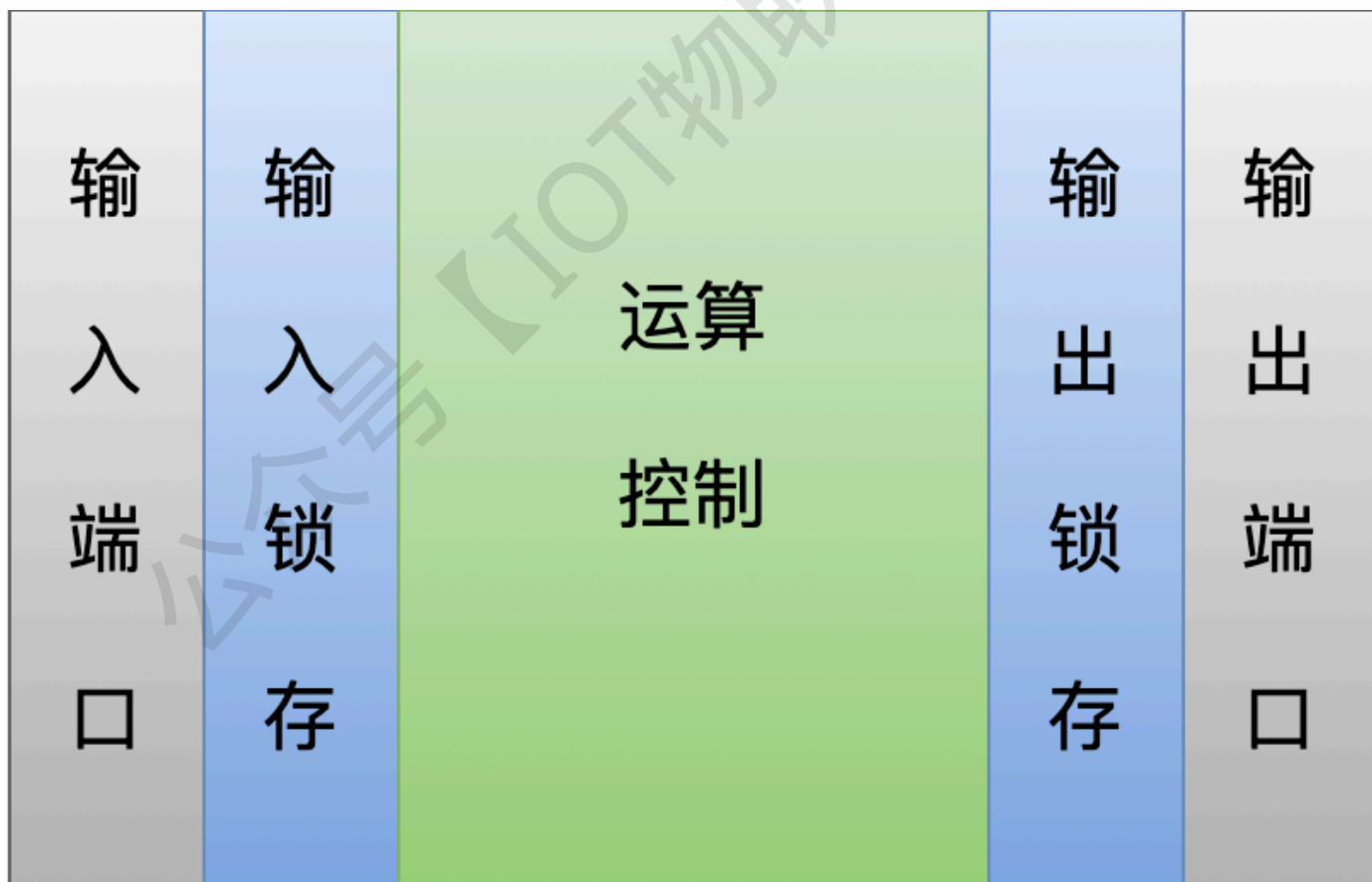
作为一名合格的嵌入式开发者, 大家估计都配置过一些单片机里的寄存器, 以达到一些功能定义、端口复用的目的, 其实这些操作, 都可以看做是我们对 CPU 的控制。

如果把 CPU 比作木偶, 那么 寄存器就是控制木偶的绳索。

我们再把 CPU 与 工控领域的 PLC 编程进行类比一下。

我们在拿到一个新的 PLC 设备之后, 其中只有一个运行时(runtime), 这个运行时执行的本职工作就是:

1. 扫描所有的输入端口, 锁存在输入映象区;
2. 执行一个运算、控制逻辑, 得到一些列输出信号, 锁存到输出映象区;
3. 把输出映象区的信号, 刷新到输出端口;



在一个全新的 PLC 中, 其中第 2 个步骤中需要的运算、控制逻辑可能就不存在。

因此, 单单一个 runtime, PLC 是无法完成一件有意义的工作的。

公众号【IOT物联网小镇】

为了让 PLC 完成一个具体的控制目标，我们还需要利用 PLC 厂家提供的[上位机编程软件](#)，开发一个运算、控制逻辑程序，编程语言一般都是[梯形图](#)居多。

当这个程序被下载到 PLC 中之后，它就可以控制[运行时](#)来做一些有意义的工作了。

我们可以简单的认为：[梯形图就是用来控制 PLC 的运行时](#)。



对于 CPU 来说，想让它执行某个内存单元的指令，只要修改寄存器 CS 和 IP 即可。

换句话说：只要对一个程序的内存布局足够的清楚，可以把 CPU 玩弄于股掌之间，让它执行哪里的代码都可以。

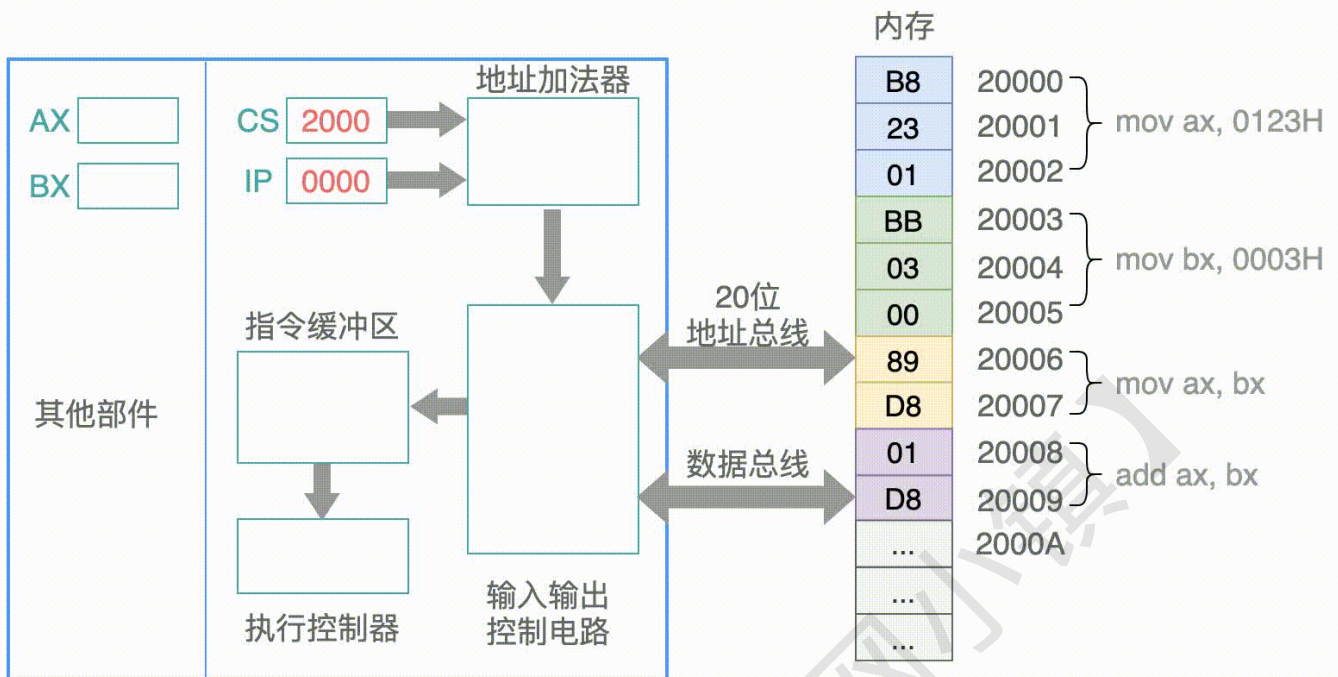
CPU 执行指令流程

现在我们已经明白了地址转换、内存的寻址，距离 CPU 执行一条指令需要的最小单元还剩下：[指令缓冲区和控制电路](#)。

简单来说：指令缓冲区用来[缓存](#)从内存中读取的指令，控制电路用来协调各种器件对总线等资源的使用。

对于下面这张图来说，它一共有 4 条指令：

初始状态: CS:IP = 2000: 0000



以第一条指令来举例，它一共经过 5 个步骤：

1. 把 CS:IP 内容送入地址加法器，计算得到 20 位的物理地址 20000H;
2. 控制电路把 20 位的地址，送入到地址总线;
3. 内存中 20000H 单元处的指令 B8 23 01，经过数据总线被送到指令缓冲区;
4. 指令偏移寄存器 IP 的值要加 3，指向下一条等待被执行的偏移地址(因为指令码 B8 代表当前指令的长度是 3 个字节);
5. 执行指令缓冲区中的指令: 把数值 0123H 送入寄存器 AX 中;

以上就是一条指令的执行最基本步骤，当然，现代处理器的指令执行流程，比这里的要复杂的多得多。

----- End -----

万丈高楼平地起！

这篇文章，仅仅描述了 CPU 执行一条指令所需要的最小知识点。

下一篇文章，我们再继续对内存的[分段机制](#)进行更进一步的窥探。

推荐阅读

专辑0: 精选文章

专辑1: C 语言

公众号【IOT物联网小镇】

专辑2: 应用程序设计

专辑3: Linux 操作系统

专辑4: 物联网



微信搜一搜



IOT物联网小镇

星标公众号，能更快找到我！

C/C++、物联网、嵌入式、Lua语言
Linux 操作系统、应用程序开发设计



扫码关注公众号



道哥 个人微信

喜欢请**分享**，满意点个**赞**，最后点**在看**。