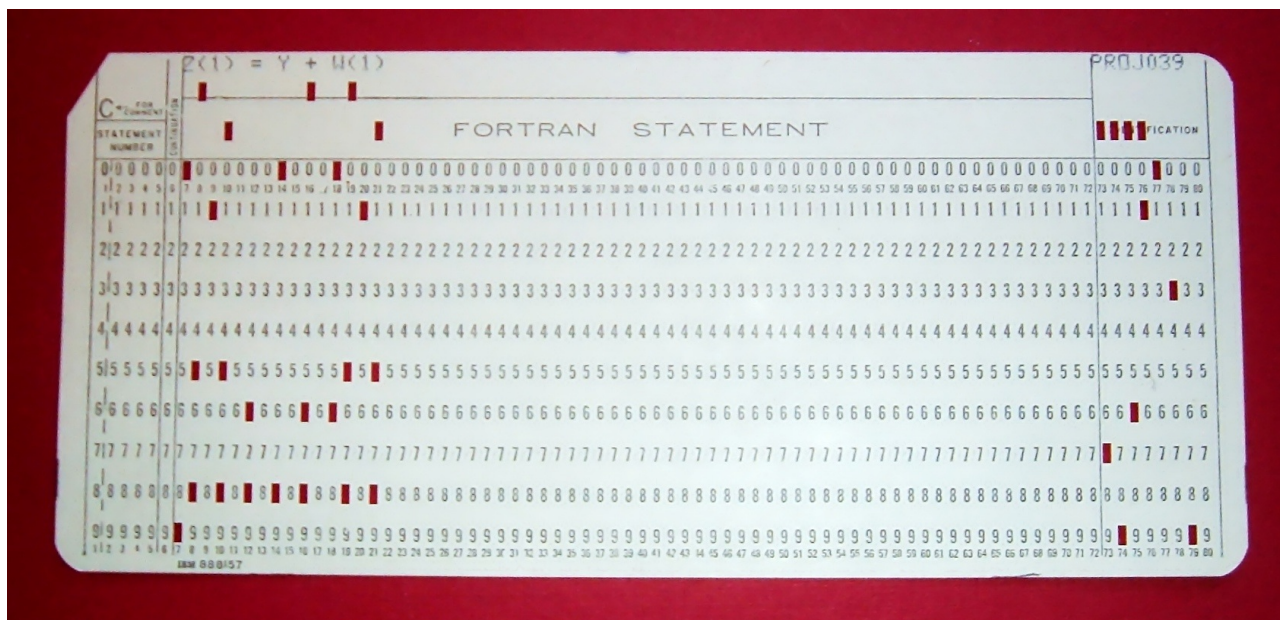


05-计算机指令：让我们试试用纸带编程

你在学写程序的时候，有没有想过，古老年代的计算机程序是怎么写出来的？

上大学的时候，我们系里教C语言程序设计的老师说，他们当年学写程序的时候，不像现在这样，都是用一种古老的物理设备，叫作“打孔卡（Punched Card）”。用这种设备写程序，可没法像今天这样，掏出键盘就能打字，而是要先在脑海里或者在纸上写出程序，然后在纸带或者卡片上打洞。这样，要写的程序、要处理的数据，就变成一条条纸带或者一张张卡片，之后再交给当时的计算机去处理。



上世纪60年代晚期或70年代初期，Arnold Reinold拍摄的FORTRAN计算程序的穿孔卡照片，[图片来源](#)

你看这个穿孔纸带是不是有点儿像我们现在考试用的答题卡？那个时候，人们在特定的位置上打洞或者不打洞，来代表“0”或者“1”。

为什么早期的计算机程序要使用打孔卡，而不能像我们现在一样，用C或者Python这样的高级语言来写呢？原因很简单，因为计算机或者说CPU本身，并没有能力理解这些高级语言。即使在2019年的今天，我们使用的现代个人计算机，仍然只能处理所谓的“机器码”，也就是一连串的“0”和“1”这样的数字。

那么，我们每天用高级语言的程序，最终是怎么变成一串串“0”和“1”的？这一串串“0”和“1”又是怎么在CPU中处理的？今天，我们就来仔细介绍一下，“机器码”和“计算机指令”到底是怎么回事。

在软硬件接口中，CPU帮我们做了什么事？

我们常说，CPU就是计算机的大脑。CPU的全称是Central Processing Unit，中文是中央处理器。

我们上一节说了，从**硬件**的角度来看，CPU就是一个超大规模集成电路，通过电路实现了加法、乘法乃至各种各样的处理逻辑。

如果我们从**软件工程师**的角度来讲，CPU就是一个执行各种**计算机指令**（Instruction Code）的逻辑机器。这里的计算机指令，就好比一门CPU能够听得懂的语言，我们也可以把它叫作**机器语言**（Machine Language）。

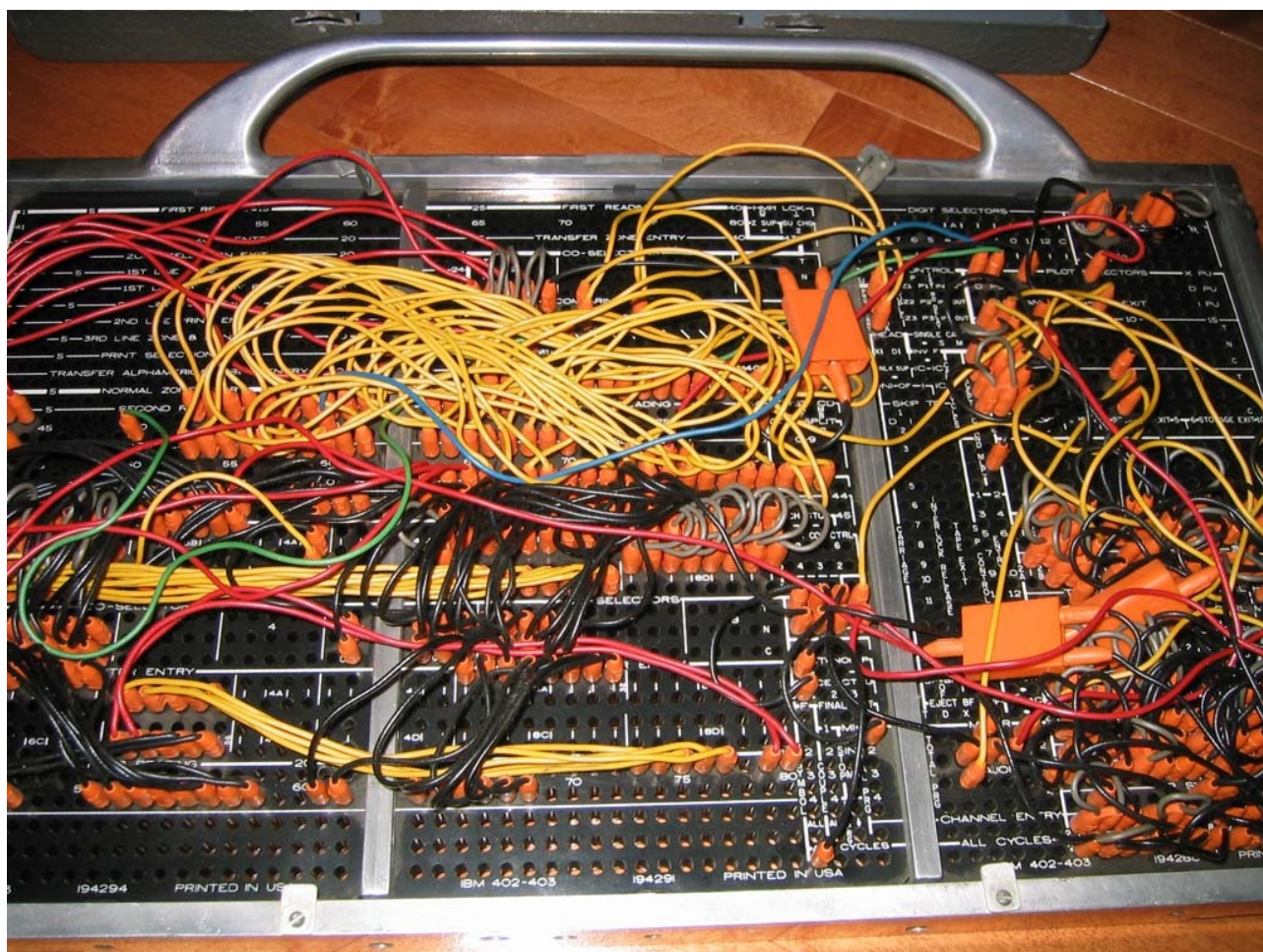
不同的CPU能够听懂的语言不太一样。比如，我们的个人电脑用的是Intel的CPU，苹果手机用的是ARM的

CPU。这两者能听懂的语言就不太一样。类似这样两种CPU各自支持的语言，就是两组不同的**计算机指令集**，英文叫Instruction Set。这里的“Set”，其实就是数学上的集合，代表不同的单词、语法。

所以，如果我们在自己电脑上写一个程序，然后把这个程序复制一下，装到自己的手机上，肯定是没办法正常运行的，因为这两者语言不通。而一台电脑上的程序，简单复制一下到另外一台电脑上，通常就能正常运行，因为这两台CPU有着相同的指令集，也就是说，它们的语言相通的。

一个计算机程序，不可能只有一条指令，而是由成千上万条指令组成的。但是CPU里不能一直放着所有指令，所以计算机程序平时是存储在存储器中的。这种程序指令存储在存储器里面的计算机，我们就叫作**存储程序型计算机**（Stored-program Computer）。

说到这里，你可能要问了，难道还有不是存储程序型的计算机么？其实，在没有现代计算机之前，有着聪明才智的工程师们，早就发明了一种叫Plugboard Computer的计算设备。我把它直译成“插线板计算机”。在一个布满了各种插口和插座的板子上，工程师们用不同的电线来连接不同的插口和插座，从而来完成各种计算任务。下面这个图就是一台IBM的Plugboard，看起来是不是有一股满满的蒸汽朋克范儿？



一台IBM的Plugboard，[图片来源](#)

从编译到汇编，代码怎么变成机器码？

了解了计算机指令和计算机指令集，接下来我们来看看，平时编写的代码，到底是怎么变成一条条计算机指令，最后被CPU执行的呢？我们拿一小段真实的C语言程序来看看。

```
// test.c
int main()
```



```
{
    int a = 1;
    int b = 2;
    a = a + b;
}
```

这是一段再简单不过的C语言程序，即便你不了解C语言，应该也可以看懂。我们给两个变量 a、b 分别赋值 1、2，然后再将a、b两个变量中的值加在一起，重新赋值给了a整个变量。

要让这段程序在一个Linux操作系统上跑起来，我们需要把整个程序翻译成一个**汇编语言**（ASM，Assembly Language）的程序，这个过程我们一般叫编译（Compile）成汇编代码。

针对汇编代码，我们可以再用汇编器（Assembler）翻译成机器码（Machine Code）。这些机器码由“0”和“1”组成的机器语言表示。这一条条机器码，就是一条条的**计算机指令**。这样一串串的16进制数字，就是我们CPU能够真正认识的计算机指令。

在一个Linux操作系统上，我们可以简单地使用gcc和objdump这样两条命令，把对应的汇编代码和机器码都打印出来。

```
$ gcc -g -c test.c
$ objdump -d -M intel -S test.o
```

可以看到，左侧有一堆数字，这些就是一条条机器码；右边有一系列的push、mov、add、pop等，这些就是对应的汇编代码。一行C语言代码，有时候只对应一条机器码和汇编代码，有时候则是对应两条机器码和汇编代码。汇编代码和机器码之间是一一对应的。

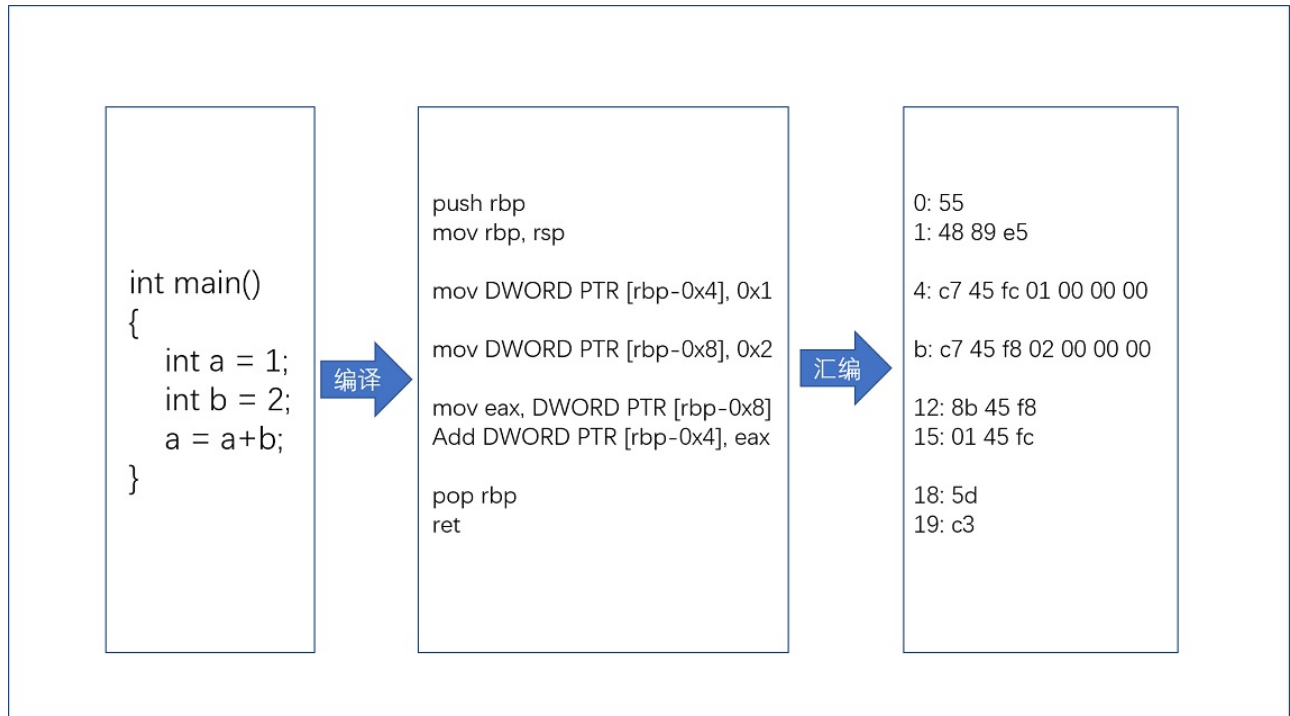
```
test.o:      file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
int main()
{
    0:  55                push    rbp
    1:  48 89 e5          mov     rbp, rsp
    int a = 1;
    4:  c7 45 fc 01 00 00 00 mov     DWORD PTR [rbp-0x4], 0x1
    int b = 2;
    b:  c7 45 f8 02 00 00 00 mov     DWORD PTR [rbp-0x8], 0x2
    a = a + b;
    12:  8b 45 f8          mov     eax, DWORD PTR [rbp-0x8]
    15:  01 45 fc          add     DWORD PTR [rbp-0x4], eax
}
    18:  5d                pop     rbp
    19:  c3                ret
```

这个时候你可能又要问了，我们实际在用GCC（GUC编译器套装，GUI Compiler Collectipon）编译器的时

候，可以直接把代码编译成机器码呀，为什么还需要汇编代码呢？原因很简单，你看着那一串数字表示的机

器码，是不是摸不着头脑？但是即使你没有学过汇编代码，看的时候多少也能“猜”出一些这些代码的含义。

因为汇编代码其实就是“给程序员看的机器码”，也正因为这样，机器码和汇编代码是一一对应的。我们人类很容易记住add、mov这些用英文表示的指令，而8b 45 f8这样的指令，由于很难一下子看明白是在干什么，所以会非常难以记忆。尽管早年互联网上到处流传，大神程序员拿着小刀在光盘上刻出操作系统的梗，但是要让你用打孔卡来写个程序，估计浪费的卡片比用上的卡片要多得多。



从高级语言到汇编代码，再到机器码，就是一个日常开发程序，最终变成了CPU可以执行的计算机指令的过程。

解析指令和机器码

了解了这个过程，下面我们放大局部，来看看这一行行的汇编代码和机器指令，到底是什么意思。

我们就从平时用的电脑、手机这些设备来说起。这些设备的CPU到底有哪些指令呢？这个还真有不少，我们日常用的Intel CPU，有2000条左右的CPU指令，实在是太多了，所以我没法一一来给你讲解。不过一般来说，常见的指令可以分成五大类。

第一类是**算术类指令**。我们的加减乘除，在CPU层面，都会变成一条条算术类指令。

第二类是**数据传输类指令**。给变量赋值、在内存里读写数据，用的都是数据传输类指令。

第三类是**逻辑类指令**。逻辑上的与或非，都是这一类指令。

第四类是**条件分支类指令**。日常我们写的“if/else”，其实都是条件分支类指令。

最后一类是**无条件跳转指令**。写一些大一点的程序，我们常常需要写一些函数或者方法。在调用函数的时候，其实就是发起了一个无条件跳转指令。

你可能一下子记不住，或者对这些指令的含义还不能一下子掌握，这里我画了一个表格，给你举例子说明一下，帮你理解、记忆。

指令类型	示例指令	示例汇编代码	含义	注释
算术类指令	add	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	将s2和s3寄存器中的数相加后的结果放到寄存器s1中
逻辑类指令	or	or \$s1, \$s2, \$s3	$\$s1 = \$s2 \mid \$s3$	将s2和s3寄存器中的数按位取或后的结果放到寄存器s1中
数据传输指令	load word	load \$s1, 10(\$s2)	$\$s1 = \text{memory}[\$s2 + 10]$	取s2寄存器中的数，加上10偏移量后，找到内存中的字，存入到s1寄存器中
条件分支指令	branch on equal	beq \$s1, \$s2, 10	if ($\$s1 == \$s2$) go to PC+4+10	如果s1和s2寄存器内的值相等，从程序计数器往后跳10
无条件跳转指令	jump	j 1000	go to 1000	跳转到1000这个目标地址

下面我们来看看，汇编器是怎么把对应的汇编代码，翻译成为机器码的。

我们说过，不同的CPU有不同的指令集，也就对应着不同的汇编语言和不同的机器码。为了方便你快速理解这个机器码的计算方式，我们选用最简单的MIPS指令集，来看看机器码是如何生成的。

MIPS是一组由MIPS技术公司在80年代中期设计出来的CPU指令集。就在最近，MIPS公司把整个指令集和芯片架构都完全开源了。想要深入研究CPU和指令集的同学，我这里推荐[一些资料](#)，你可以自己了解下。

指令类型	6位	5位	5位	5位	5位	6位	解释
R	opcode	rs	rt	rd	shamt 位移量	funct 功能码	算术操作、逻辑操作
I	opcode	rs	rt	address/immediate 地址/立即数			数据传输、条件分支、立即数操作
J	opcode	target address 目标地址					无条件跳转

MIPS的指令是一个32位的整数，高6位叫**操作码**（Opcode），也就是代表这条指令具体是一条什么样的指令，剩下的26位有三种格式，分别是R、I和J。

R指令是一般用来做算术和逻辑操作，里面有读取和写入数据的寄存器的地址。如果是逻辑位移操作，后面还有位移操作的位移量，而最后的功能码，则是在前面的操作码不够的时候，扩展操作码表示对应的具体指令的。

I指令，则通常是用在数据传输、条件分支，以及在运算的时候使用的并非变量还是常数的时候。这个时候，没有了位移量和操作码，也没有了第三个寄存器，而是把这三部分直接合并成了一个地址值或者一个常数。

J指令就是一个跳转指令，高6位之外的26位都是一个跳转后的地址。

```
add $t0,$s2,$s1
```

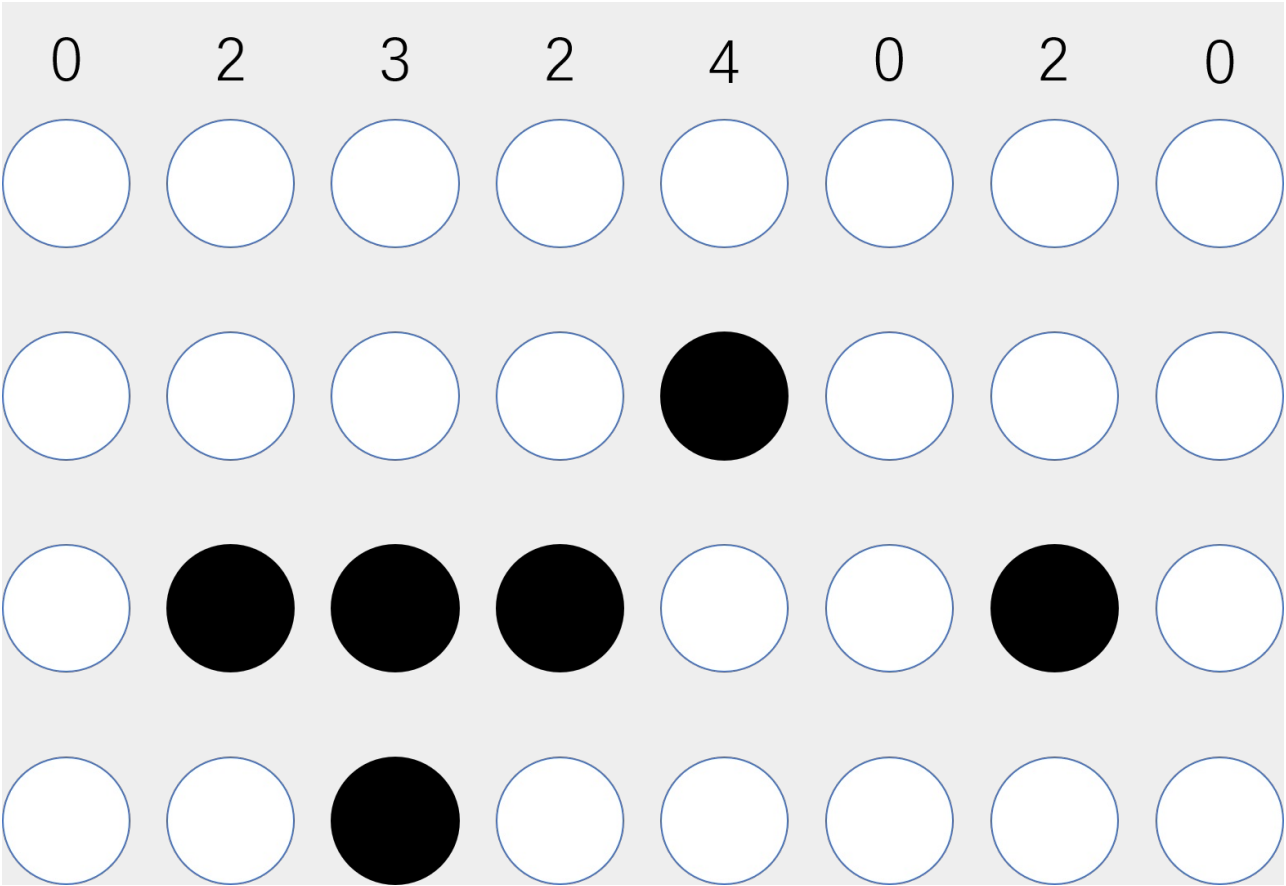
我以一个简单的加法算术指令add \$t0, \$s1, \$s2,为例，给你解释。为了方便，我们下面都用十进制来表示对应的代码。

对应的MIPS指令里opcode是0，rs代表第一个寄存器s1的地址是17，rt代表第二个寄存器s2的地址是18，rd代表目标的临时寄存器t0的地址，是8。因为不是位移操作，所以位移量是0。把这些数字拼在一起，就变成了一个MIPS的加法指令。

为了读起来方便，我们一般把对应的二进制数，用16进制表示出来。在这里，也就是0X02324020。这个数字也就是这条指令对应的机器码。

指令	格式	opcode	rs	rt	rd	shamt	funct
add	R	0	17	18	8	0	32
二进制表示		0	10001	10010	01000	00000	100000
十六进制表示		0X02324020					

回到开头我们说的打孔带。如果我们用打孔代表1，没有打孔代表0，用4行8列代表一条指令来打一个穿孔纸带，那么这条命令大概就长这样：



好了，恭喜你，读到这里，你应该学会了怎么作为人肉编译和汇编器，给纸带打孔编程了，不用再对那些用过打孔卡的前辈们顶礼膜拜了。

总结延伸

到这里，想必你也应该明白了，我们在这一讲的开头介绍的打孔卡，其实就是一种存储程序型计算机。

只是这整个程序的机器码，不是通过计算机编译出来的，而是由程序员，用人脑“编译”成一张张卡片的。对应的程序，也不是存储在设备里，而是存储成一张打好孔的卡片。但是整个程序运行的逻辑和其他CPU的机器语言没有什么分别，也是处理一串“0”和“1”组成的机器码而已。

这一讲里，我们看到了一个C语言程序，是怎么被编译成为汇编语言，乃至通过汇编器再翻译成机器码的。

除了C这样的编译型的语言之外，不管是Python这样的解释型语言，还是Java这样使用虚拟机的语言，其实最终都是由不同形式的程序，把我们写好的代码，转换成CPU能够理解的机器码来执行的。

只是解释型语言，是通过解释器在程序运行的时候逐句翻译，而Java这样使用虚拟机的语言，则是由虚拟机对编译出来的中间代码进行解释，或者即时编译成为机器码来最终执行。

然而，单单理解一条指令是怎么变成机器码的肯定是不够的。接下来的几节，我会深入讲解，包含条件、循环、函数、递归这些语句的完整程序，是怎么在CPU里面执行的。

推荐阅读

这一讲里，我们用的是相对最简单的MIPS指令集作示例。想要对我们日常使用的Intel CPU的指令集有所了解，可以参看《计算机组成与设计：软/硬件接口》第5版的2.17小节。

课后思考

我们把一个数字在命令行里面打印出来，背后对应的机器码是什么？你可以试试通过GCC把这个的汇编代码和机器码打出来。

欢迎你在留言区写下你的思考和疑问，你也可以把今天的文章分享给你朋友，和他一起学习和进步。




深入浅出计算机组成原理

带你掌握计算机体系全貌

徐文浩 bothub 创始人



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言：

- 梨子🍏 2019-05-03 01:50:53
没有理解 `0X02324020` 是如何计算出来的？

作者回复2019-05-03 04:15:20

梨子同学你好，你把把上一行的二进制，四位四位一读，每四位当成是一个16进制数，就会得到这个结果