第二讲 指令系统体系结构

201-设计自己的计算机

> 什么是指令系统体系结构呢?
>
> 要回答这个问题，其实非常的简单。但是想解释清楚，
>
> 也没有那么容易。我们还是从一个小故事开始吧。

> 有一天两个小伙伴碰了面，发现对方都很愁苦，一个就问另一个 "嘿你在愁苦什么呀?"
>
> 这个说"唉呀，最近遇到了很多运算上的问题。"
>
> "运算量好大呀。"对应说，"可不是吗，我也遇到了类似的问题。"
>
> "咱们怎么解决呢?"
>
> "不如咱们一起设计一个计算机吧!"
>
> 就说好啊，两人就一拍即和。
>
> "我们设计计算机怎么分工呢?"一个说，"我显然是软件 程序员，我来编写如何运算的软件指令。"
>
> 另一个说"那正好，我是硬件工程师，我 来设计计算机的硬件，主要是CPU，那咱们就分头工作吧!"
>
> "这事估计得花一年时间才能把CPU设计出来，也才能把软件
>
> 写好。真的就可以这么开始了吗?一年之后我们在碰面 怎么保证你写的软件就能在我做的CPU上运行起来呢?"
>
> "所以还不能着急，咱们得商量商量，得把咱们的共同规则订
>
> 好，然后呢才能分头去设计软件和硬件，这样保证之后我们在碰面的时候 软件和硬件能顺利的结合在一起。"
>
> 那好，这两个小伙伴就开始商量了，他们要商量的是什么呢?
>
> 就是我们要谈的指令系统体系结构。

> 他们要面临的计算任务并不复杂，所以只要一个很简单的计算机指令系统就可以了。
>
> 我们要设计多少指令、要设计哪些指令，首先要根据需求来确定。
>
> 那看上去我们只需要一些简单的加法。所以首先，我们需要设计一个加法指令。
>
> 那这些指令和要运算的数据，都是要放在存储器当中的，如果把两个存储器当中的数字相加，
>
> 可能会比较复杂。所以我们指令系统设计的时候呢，这个加法指令是这么做的:
>
> 将一个寄存器当中的数，和一个存储器当中的数相加，然后存到这个寄存器当中。
>
> 寄存器我们用R来表示，存储器当中的地址用M来表示。
>
> 所以我们这个指令系统当中，包含了一条运算类的指令。
>
> 但是寄存器当中的数从哪里来呢?自然是从存储器当中来。
>
> 把存储器当中的内容，装到寄存器当中来，这条指令就是LOAD。
>
> 就是将后面一个操作数M，所指向的内存单元中的内容放到前一个操作数R所指向的

寄存器当中。有了这两条指令之后，我们可以对内存当中的数 进行加法运算了。但是运算的结果还在寄存器当中，

所以我们还需要一条指令，把寄存器当中的数，再放回到存储器当中。

这条指令，我们记为STORE，它的作用是将寄存器R当中的数，

存入到M所指定的存储单元中。这两条指令用于

在存储器和寄存器中间传送数据，所以我们称之为传送类的指令。

我们已经知道CPU是从内存当中，按照地址，依次取出指令开始执行的，

那有时候我们想改变取指令的位子，这时候就需要用到这条指令，记作JMP L，

当CPU执行这条指令之后，就会转移到L所指向的存储器单元中去取出下一条指令来执行， 这样的指令我们称为转移类指令。

然后我们就设计好了我们想要的计算机的指令系统，虽然它很简单，但足够完成我们想要的运- 算任务了。

但这还是用英文单字和字母进行的描述，并不是计算机所能识别的二进式代码。

因此我们还要做近一步的规定。这就是指令的具体格式，

首先我们约定，每条指令都是等长的，都是两个字节。

其中第一个字节，我们取了高四位作为操作码，操作码

就是指明了这是一条什么类型的指令。我们现在有四条指令，

LOAD、ADD、STORE和JMP，我们分别给它分配了四个不同的操作码。

用十进制来表示就是0、1、2、3，

因为我们预留了四个二进制位，所以以后还可以扩展，最多可以扩展到16条指令。

但是现在我们只定义了四条指令，

接着看第一个字节的低四位，这个四位我们约定了作为寄存器号，

我们现在提供四个寄存器，编号从0000到

0011，分别指代CPU当中的R0到R3这四个寄存器，

由于我们预留了四个二进制位，所以以后还可以继续扩展，最多到十六个寄存器。

那就这为我们发展第二代、第三代以及之后的设计时，提供了扩展的空间。

这个指令的第二个字节， 我们约定作为存储单元的地址，这样有八个二进制位，

所以我们一共可以使用2的八次方，也就是256个字节的存储器。

我们看一个例子。如果软件设计人员写了这么一条指令:0001 0010

0000 1001，那从操作码 这四位我们就可以看出来它是一个加法指令。

从这四位指定的寄存器号，就可以看出来他想访问的是R2这个寄存器。

然后存储单元的地址，翻译成十进制的话就是9。所以这条指令的编码

想完成的操作，就是将R2的内容和存储单元9的内容相加，存到R2寄存器当中。

但如果软件成员写了这样的指令，0101 1010 0000

1001，如果CPU看到了这条指令， 那它是无法识别的，它不知道该做什么样的操作。

因为0101这个操作码并没有定义。

1010这个寄存器号，也没有定义。

当然也许在第二代或第三代的设计中，我们可以再定义了0101是减法，

1010可能是十号寄存器，那么在那个时候的CPU上，就

可以执行这条指令了。但在我们约定的第一代CPU上显然是无法执行的。

那约定好了指令的具体格式，我们再来看一看我们可以做的运算的任务。

假设我们要完成这样一个任务，将存储器地址M1中的内容，与存储器

地址M2中的内容相加，最后存到M3这个存储单元当中，

在完成运算之后，程序转向，存储器当中L所指向的位子继续执行。

但这样的任务，我们如何用我们现在有的这个指令系统来实现呢?

我们能用的只有这四条指令，我们没有 提供直接将两个存储单元的数进行香加的指令。

这个程序应该这么写，第一步将M1中 的内容送到了一个寄存器，暂且记为RX，

然后将RX的内容和M2的内容相加，运算结果存入

RX，再将RX的内容送到存储器M3当中，

这就完成了运算。再转移到L，取出下一条指令继续执行。

我们假设现实在第一个任务中，M1指向存储单元地址五，M2

指向6，M3指向7，最后要转移的目标地址L是18，

这几个都是十进制的描述。那基于这样的任务，我们的软件程序员就可以编写出这样的机器语- 言的程序，

当然初看上去全是01的代码，很难分辨，为了便于学习，

我将它按照不同的含义用不同的颜色表示出来。

相同的颜色代表着它们对应的关系，这样我们就可以看出来第一条指令

0000的操作码，应该是一个LOAD指令， 接下来0011指定了寄存器的编号，所以它指向那个R3。

第三部份是存储器的地址，这个二进制数实际上就是十进制的5，这就可以看出来

我们是用LOAD这条指令，将存储单元当中，地址5所对应的内容，传送到了R3这个寄存- 器当中。

这样我们就可以依次分析出每条指令的功能，当然 我还是要先强调一下，编写程序的实际的顺序。

最早的软件程序员，是需要直接编写机器语言指令的， 那么它要编写的就是中间这类的01编码，并把这样的01编码

在穿孔纸带上，鑽上对应的小孔，送到计算机当中去， 这样的工作，显然是非常的繁琐、容易出错，效率也很低。

那后来随着技术的进步，程序员就可以编写类似左边这一类的汇编语言程序。

汇编语言程序基本上和机器语言程序可以作到一一对应，那我们通过一些工具，就可以

先将汇编语言程序转换成对应的机器语言程序，然后再将机器语言程序输送到计算机当中。

再进一步，人们又可以编写各种高级语言的程序。那当然还是需要更为复杂的工具，

将高级语言经过若干个步骤，最后还是要转换成我们这个表格中间这一列所展示的机器语言。

也就是一系列二进制代码。

现在我们已经写好了这样的程序，我们需要把这样的程序放到存储器当中去。

我们要注意的是，这只是存储器的一个片段，最左边这一列只是要存储器的地址。

我们展示了存储单元从地址5一直到地址18，所保存的内容。

我们从上往下依次看，在地址5和6这两个存储单元中，

存放了两个我们需要进行运算的源操作数，现在存放的是12和34.

地址单元7准备用来存放运算结果的。

现在被初始化为0。地址单元8和9这两个字节，存在了LOAD这条指令。

为了便于观看，我还是用不同的颜色标出了不同含义的二进制位， 再往下的两个字节，是加法这条指令。

然后是STORE这条指令。然后是转移JMP这条指令。我们知道JMP这条指令会让CPU

转向地址单元18，去取出指令来。所以这里所描述的第五条指令是不会被执行的。

在执行完JMP这条指令以后，CPU会取出第六条指令进行执行。

我们还是用大家熟悉的模型机的图示进行说明。

假想我们已经把刚才的程序和数据输送到了这个模型机的存储器当中，

与此同时，硬件工程师也设计完成了CPU，并将

CPU和存储器进行连接，构建成了完整的计算机的系统。

为了便于阅读，我将存储器当中二进制码进行了转换。我们可以看到，在存储

单元中，依次存放着我们要进行运算的源操作数，准备存放结果的空间，

还有包含了四条指令的这段程序。然后看右边CPU当中， 如果PC寄存器已经装入了0000

1000这样一个地址， 接下来的过程大家应该很熟悉了，PC中的这个地址会通过MAR寄存器，再通过

地址总线送到存储器，存储器就会找到这个单元所对应的那条指令，

也就是LOAD这条指令，把这条指令的编码送回CPU。接下来就会依次完成我们所编写的- 这个程序。

当然最后还有一个问题，为什么PC寄存器当中的地址是这个呢?

其实这是不一定的，这也是我们进行指令系统体系结构设计时。必须要约定的一个内容。

就是CPU在启动时，或者说在复位完成之后，第一条指令从哪里开始取出。

这也是最开始，软硬件双方必须商量好的事情。

至于这个地址到底应该是什么，并没有明确的规则。

但通常情况下，我们会约定为这个体系结构所能访问的存储单元

的最小地址，也就是0，或者是接近最高地址的地方。

那大家也可以思考一个很简单的问题，如果我们这个体系结构约定的CPU的

起始地址是全0的话，存储器当中的程序应该做怎么样的改动呢?

> 如果要求现在左边这个图中，已经存放的程序和数据的位置都不允许变动呢。
>
> 其实解决方法很简单。
>
> 留给大家思考。

> 现在我们已经对什么是指令系统体系结构有了初步的了解，
>
> 也知道如何着手开始设计一个属于自己的计算机。
>
> 从下一节开始，我们将一起分析几个真实 的体系结构。

202-x86体系结构

> X86是商业上最为成功，影响力最大的一种体系结构。
>
> 但从技术的角度看，它又存在着很多的问题， 那我们就来一起分析X86，这种体系结构的特点。

> 这张表列出来X86体系结构当中，具有代表性的一些微处理器的型号。
>
> 主要分成了16位、32位和64位三大类。
>
> 我们先来看最早推出来的8086， 8086是一款16位的CPU，
>
> 所谓16位CPU，主要是指CPU当中的运算部件可以支持16位数据的运算。
>
> 因为运算当中所需要的数据，一般会放在通用寄存器中。
>
> 所以通用寄存器的位宽通常和运算单元的位宽是相同的。
>
> 而运算单元产生的数据又经常会用做访问存储器的地址。
>
> 所以CPU访问存储器地址的宽度，也常常和运算单元的位宽相同。
>
> 那么对于8086来说，它是一个16位的CPU，它内部的通用寄存器也是16位的。
>
> 但是它连接存储器的地址线的宽度却是20位的。
>
> 那么它生成访问存储器的地址就需要采用一些特殊的方式。
>
> 下面我们先来看8086内部的通用寄存器的情况，再来介绍它生成地址的方式。
>
> 这就是8086体系结构所规定的寄存器。
>
> 都是16位宽的，主要可分为这几类：通用寄存器、
>
> 指令指针寄存器、标志寄存器，还有段寄存器。
>
> 我们首先来看通用寄存器，结合我们之前说过的模型机的例子，
>
> 通用寄存器就在这里，CPU从存储器当中取回一个数， 很可能就会放在某一个通用寄存器当中。
>
> 而CPU执行运算指令，其操作数的来源也往往会在寄存器中。
>
> 对于8086来说，这些用于存储数据的通用寄存器， 主要有这四个：AX、BX、CX和DX。
>
> 这四个寄存器都是16位寄存器。
>
> 但是这些16位寄存器还可以被分为两个8位的寄存器来使用。
>
> 大多数的算术运算和逻辑运算的指令，都可以使用这些数据寄存器。
>
> 那么这些寄存器除了可以一般性的存放数据之外，
>
> 还会有一些专门的用途，那我们在后面介绍到相关的指令时再做具体的讲解。
>
> 除了这四个寄存器还有SP、BP、SI、DI这四个通用寄存器。

它们在早期都有一些特殊的用途。

而随着X86体系结构的不断更新，它们也大多成为了可以用于保存普通数据的寄存器。

然后我们再来看标志 寄存器，之前分析模型机时我们提到，

当执行运算指令时，ALU会将X和Y两个 寄存器当中的内容相加，并将运算结果放在Z这个寄存器中。

同时将运算结果的一些特性保持在标志寄存器中。例如这个

加法运算如果产生了进位，那就可以将这个进位保持在标志寄存器中，

以免丢失，后续的运算也可以知道之前的运算产生了这样一个进位。

那在8086中，也有这样一个标志寄存器。

称为FLAGS，这个寄存器当中，包含了若干个标志位。

主要可以分为量大类，一类称为状态标志，它反映的是CPU的工作状态。

另一大类称为控制标志。

这是对CPU的运行起到特定的控制的作用。

那8086的这个标志寄存器也是16位的。

但实际只有其中一部分有具体的含义。

在图中标为红色的都是状态标志，标为紫色的这三个是控制标志。

例如我们刚才提到的加法的进位标志。

当CPU执行完一条加法指令， 而这次加法运算的结果产生了一个进位，

那CPU内部除了将加法的运算结果保存到对应的寄存器之外， 还同时会将这个CF标志置为1，

这个动作是由硬件自动完成的，不需要由编程人员来设置。

我们再来看下一个寄存器。

在模型机上，CPU要去取下一条指令之前， 都会先从PC寄存器当中，取出下一条指令的地址，

将这个地址发到存储器中，才能取回下一条指令的编码。

那在8086当中，这个寄存器称为IP寄存器。

IP是指令指针的缩写。

编程人员是不能直接修改IP寄存器的， 除了顺序取出指令，IP寄存器会自动增加以外，

如果遇到了转移指令，这些会改变程序流向的指令， 那CPU会自动修改IP寄存器的内容。

这里我们还需要注意一个问题，因为IP寄存器是16位宽的，

所以它能够指向的内存单元的数量是2的16次方。

也就是64K个字节单元，那么即使在那个时代，64K的内存也是太小了，

无法满足当时大多数程序的需求。

因此，实际上8086在外部连接的是一个1兆字节的内存。

这样就需要8086对外有20位的地址线。

那多出来的这4位地址线，从哪里来呢？ 那8086采用的是一个很巧妙，也很繁琐的解决方案。

这就是用段寄存器的方式，

段寄存器是用来和其它寄存器一起联合生成存储器地址的， 8086当中有4个段寄存器。

CS是代码段寄存器，DS是数据段寄存器。

ES是附加段寄存器，SS是堆栈段寄存器。

我们以代码段寄存器为例，来看一看地址生成的方式。

假设8086CPU要从这个1M的内存中取出一条指令，

那就需要现在段寄存器当中保存这个地址的一部分。

然后地址的另一部分根据这个程序的本身来产生。这样的组合，就称为逻辑地址。

我们假设已经在这个代码段寄存器当中存放了一个16位数， 那用16进制来表示，就是2000H，

而根据程序运行的状况，当前IP寄存器当中的值是3000H，

那下一条指令的地址是怎么产生的呢？ 那在CPU内部就会有一个硬件单元，负责移位，

先将段寄存器当中的16位数向左移4位。

那新产生的这个数用16进制来表示就是20000H，

然后再将这个移位后的数与IP寄存器当中的内容相加， 这又需要用到一个加法器。

相加之后就得到一个20位的地址， 在这里就是23000H，这时CPU才

可以将这个地址发送到存储器去，从而取回下一条指令的编码。

而这个地址则被称为物理地址，从逻辑地址到物理地址，就是用

段寄存器当中的内容乘以16再加上程序中产生的偏移地址。

那再用我们已经很熟悉的模型机来看另一个例子。

假设这时CPU已经把下一条指令的编码取回了， 放在IR寄存器当中，那这条指令是要将3000H

所指向的内存地址当中的数取出来，放在AX寄存器当中。

那如果是在我们之前讲过的模型机上运行， CPU就会将3000H这个数，

放到MIR寄存器当中去，然后再传送到地址总线上。

但是对于8086来说，它要发出的是一个20位的地址，

必须先要用段加偏移的方式进行计算。

那我们假设之前已经在DS寄存器当中保存了2000H这个数， 那CPU的硬件就会将DS当中的数取出来。

送到一个移位的部件，向左移4位。

然后再和3000H相加。

这样就得到了一个20位的地址，23000H， 然后才能把

这个生成的地址放在MIR寄存器当中，再传送到地址总线上。

然后存储器则会返回23000H这个地址所对应的内容，

并放到数据总线上，进一步保存到了MDR寄存器中，最后CPU的硬件

会将MDR当中的内容，再传送到AX计算器当中，从而完成了这条指令所执行的操作。

那么结合上一页我们所介绍的内容，我们会发现，对于8086来说，

它在取指令的时候，就要执行一次段加偏移的这样的计算。

那么在执行指令的时候，还要执行这样一次计算，那它执行一条指令的过程

就比我们之前在模型机上学习的例子要复杂得多了。

当然，虽然很繁琐， 但在那个时期，确实在一定程度上解决了16位地址空间太小的问题。

但是想要提供更高的性能，以满足当时蓬勃发展的个人计算机的需要， 还是要从体系结构上做大的改进。

而1985年推出的80386就是这样一款跨时代的作品。

80386是x86系列当中第一款32位的微处理器，

也就是说它的运算部件可以支持32位数据的运算，

同时也提供32位的通用寄存器， 那么自然它也可以产生32位的地址，

从而可以指向2的32次方，也就是4G字节的内存空间。

这样大容量的内存空间在之后相当长的 时间里，都让编程几乎不受内存空间的限制。

而英特尔也凭借80386确立了它在个人计算机CPU领域的优势地位。

此外，80386还对运行模式进行了改进，

以便更好和更稳定地支持操作系统，以及越来越丰富的软件。

32位x86的体系结构也被称为IA-32，

它所提供的32位寄存器是在8086 16位寄存器的基础上扩展而来的。

例如8086中的AX寄存器 在为它增加了16位之后就变成了32位的EAX寄存器。

在指令中如果使用EAX， 就是指这个32位的寄存器，但与此同时，指令中

还可以继续使用AX来指定其中的低16位。

同样，也可以继续使用AH和AL这两个8位的寄存器编号，

那这样IA-32中就有了8个32位的通用寄存器，

还有一个32位的标志寄存器。

指令指针寄存器也扩展到了32位，

用这个寄存器就可以指向2的32次方，也就是4G字节的内存空间。

从这里看来，386只要使用这个EIP寄存器就足够了，

但实际上386不但保留了原先的4个段寄存器， 还增加了2个段寄存器。而运行在保护模式下，

这些段寄存器的使用方法是不一样的。

你如果有兴趣可以查阅保护模式相关的资料进行学习， 在这里就不再详细描述了。

那到了上世纪90年代后期，即使在个人计算机领域，

32位CPU也逐渐出现了难以满足性能需求的情况，

尤其是4G内存的空间限制了大规模程序的应用，

那在这时，一贯主导x86体系结构改进的英特尔，它提出了名为IA-64的体系结构。

这个64位的体系结构和之前的x86体系结构并不兼容，

那由于种种原因，这个新的结构并未获得成功。

那趁着这个机会，AMD后来居上， 提出了与原先兼容的64位的x86的方案，

从而在64位的时代占据了先机， 当然后来英特尔也转回来支持这个兼容的方案。

那这个方案有很多不同的名字， 比如说AMD64，Intel64，

通常我们更多地把它称为x86-64。

那x86-64的寄存器模型 则是在IA-32的32位寄存器模型的基础上进行了扩展。

那与之前类似，在原先32位的EAX寄存器的基础上

再增加32位，形成了64位的RAX寄存器。

而指令指针寄存器也被扩展到了64位，因此

理论上我们就可以访问2的64次方个字节这么大的内存空间。

此外，因为把常用的操作数 放在寄存器当中比放在存储器当中性能要好得多。

因此有更多的寄存器，编程就会更加地方便。

那么在x86-64当中，另外还新增了8个64位的通用寄存器，

这8个新增的寄存器的名称依次为R8，R9，一直到R15。

因为之前我们就已经有了8个通用寄存器， 如果要给它们编号的话，就正好是从R0到R7，

所以新增的寄存器就从R8开始编号，

这就是x86体系结构从16位直到64位的大致情况。

现在我们已经了解了x86体系结构的基本特点，

之后我们将通过分析x-86的具体指令来进一步学习 这种体系结构。

203-x86指令简介

x86指令种类繁多，数量庞大，

在这一节我们将会学习x86指令的分类， 并分析其中最为基础的一部分指令。

通常一个指令系统主要包括这几类指令。

运算指令，比如加、减、乘、除这样的算术运算，以及与、或、非这样的逻辑运算。

还有传送类指令，比如把数据从存储器送到通用 寄存器，或者从通用寄存器送到I/O接口等等。

有了这两类指令，计算机就可以从外界获取数据，

并在内部完成运算，最后将结果输出到外界。但是如果你想编制比较复杂的程序， 例如像高级语言当中if

else这样的语句， 或者是for while这样的循环语句，

那就需要用到转移类指令， 另外还需要有一些对CPU进行控制的指令。

那无论是哪一类指令，我们首先要关心的就是它究竟改变了什么。

例如一条加法指令，它会改变通用寄存器的内容，

或者有可能改变标志位， 再有是改变存储器单元的内容，或者改变外设端口的内容，

还有可能改变指令指针以及其他的情况。

那我们在学习到新的指令的时候，一定要认真地想清楚这条指令

究竟改变了哪些地方，又对后续的指令会产生什么样的影响。

现在我们就通过一个示例程序来讲解几个常用的指令。

这个程序的目的是进行两个数的求和运算，

这两个数比较大，可能有很多个字节，第一个数存放在2000H开始的

存储器空间中，第二个数存放在3000H开始的存储器空间中。

而且我们希望这个程序有一定的灵活性，可以适应不同长度的数。

这两个数的长度存放在2500H这个字节单元里。

那我们就可以假设存储器单元中存放的数的情况是这样的，

从2000H开始，若干个字节存放在第一个数，3000H开始的

若干个字节存放在第二个数，而2500H地址对应的这个字节

则告诉我们这两个数有多长。从这里我们可以看出是16进制的12H，

也就是十进制的18。现在这个程序已经编写完毕， 那我们就来一起逐条指令地分析这个程序。

首先我们要用到的是传送类指令， 传送类指令的作用是把数据或者地址

传送到寄存器或者存储器单元中。

这些是x86指令系统当中常用的 传送类指令，那我们就来看一看第一条MOV指令。

MOV指令带两个操作数，第一个是目的操作数，

第二个是源操作数。这条指令所做的操作，就是将源操作数中的内容

传送到目的操作数中。看似简单，但实际上 这条指令的格式有非常丰富的变化。

例如这条MOV指令就是将40这个数送到EBX寄存器当中，

那对于它的源操作数来说，就是使用了直接给出操作数的 方式，这个操作数的值就会体现在指令编码中，

CPU在从存储器取址的时候， 就会将40这个数作为指令编码的一部分取回来，

然后就可以直接将这个数送到EBX寄存器中了。

第二条MOV指令是将BL寄存器的内容传送到AL寄存器中，

那对于源操作数来说，这里给出的是存放操作数的寄存器的名称。

第三条MOV指令则是将1000H所指向的存储器单元中的内容取出，

传送到ECX寄存器中。这里源操作数则是给出了存放操作数的存储器的地址。

第四条MOV指令是将AX寄存器中的内容传送到存储器的某个单元，

这个单元的地址是存放在DI寄存器中，所以在执行这条指令的过程中，

CPU需要先从DI寄存器当中取出一个数，把这个数作为访问存储器的地址，

再从AX当中取出一个数，作为访问存储器的 数据，再执行写存储器的操作。

因此对于这个目的操作数，

是给出了一个寄存器的名称，而这个寄存器当中存放了操作数的存储器地址。

最后是一个更复杂的情况，这里给出的是存放操作数的存储器地址的寄存方法。

这里WORD PTR这个关键词所表明的意思是

这个内存地址指向的是长度为1个字的内存单元，

也就是两个字节。那我们要把01H这个数存放到这个

内存单元当中去，而计算这个内存单元地址的过程是这样的。

CPU要从SI寄存器当中取出一个数， 并将它乘以2，然后从BX寄存器当中取出一个数，

二者相加，还要再加上200H这个数。在完成了这么多次运算之后，

我们才能得到这个存储器的地址， 然后才能发起存储器写的操作，将01H这个数送到指定的内存单元当中去。

从这几个例子我们可以看出，x86提供了非常丰富的访问存储器的方法，

这为编写程序带来了很大的便利，但这也让CPU的设计变得非常地复杂。

那我们再来看看MOV指令的编码，

这就是一条MOV指令，它有3个字节。

第一行是这3个字节的含义， 第二行是我们举的一个例子的具体的编码，

那这个编码实际上是MOV AX，10EEH这条指令。

我们可以发现第2个字节和第3个字节就是这条指令当中的这个立即数10EE。

而第1个字节的最后3个比特是指定了寄存器的编号，

000代表是AX，而前面1011则是代表了这个类型的MOV指令。

因此CPU取回这条指令编码就知道是将后面 这两个字节的内容写入到AX这个寄存器当中。

我们再来看另一条MOV指令。

这条MOV指令执行的是一个存储器到寄存器的传送，这个存储器的地址

是由BX寄存器的内容和立即数1004H相加得到的，

那我们在指令编码当中，也能找到1004这个立即数，还有

在寄存器这个位域所对应的能看到011，这是BX寄存器的编号。

那这条MOV指令比上面这条要复杂一些，所以它是4个字节的。

从这里我们也可以看出，x86指令系统是一种变长的指令，

它可以根据需要设定指令编码的长度，这样就比较灵活。

但是从另一个方面来看，这对CPU取指令的操作会带来很多的麻烦。

因为CPU在取到这条指令之前， 它无法判断这条指令究竟由几个字节组成，

取得少了，那指令编码不全，无法执行； 取得太多，又会浪费时间，还会多占用CPU内部的空间。

这就是变长指令的不利之处。

那我们还是回到这个程序的例子，前三条都是MOV指令，

第一条是将2500H这个内存地址中的内容传送到CL寄存器中，

这个内存地址当中保存的是我们要运算的数的长度。

第二条是将2000H这个立即数传送到SI寄存器中。

第三条是将3000H这个立即数传送到DI寄存器中。

这样SI和DI这两个寄存器就分别保存了我们要计算的这两个数的起始地址。

接下来我们就可以开始运算了，这就需要用到运算类指令。

运算类指令包括逻辑运算指令，移位指令，

还有算术运算指令。我们就选择加法指令为例进行介绍。

这里有三条加法指令，第一条是ADD指令，

它有两个操作数，所做的操作是将这两个操作数中的内容相加，

并将结果存放到第一个操作数当中去，这里前两条就是ADD指令的示例，

这我们应该比较熟悉了，就不再详细介绍。

另外一条特殊的加法指令是INC指令，这条 指令只有一个操作数，它要做的就是将这个操作数加1。

就像这个例子，就是把CL寄存器当中的数加1，结果还保存在CL寄存器中。

INC指令的功能很简单，它的指令编码也很短，这条指令只需要一个字节，

是最短的x86指令之一。

那加1其实就是加法的一种特殊的情况，为什么要单独设一种指令呢?

从这里我们也可以看出x86的设计思想。

因为在程序当中，我们经常会进行每次加1的计数的操作，

那为这种常见的情况设计一种很短的指令，就可以大大减小程序代码的长度，

这在存储空间非常有限的情况下，是非常有意义的。

那第三个加法指令是ADC指令，就是带进位的加法，

ADC指令看上去也只有两个操作数，

但实际它的加法运算是将这两个操作数相加，再加上CF标志位，

运算的结果放回到第一个操作数中去。

我们结合这个模型机来看一看。

对于一般的加法指令，ADD指令会用到ALU，如果这个加法运算产生了进位，

就会去改写标志寄存器当中的CF位。

而如果当前执行的是ADC指令，那标志寄存器当中的 CF位也会被送到ALU参与运算，

这样之前的运算指令的结果实际就影响了现在这条加法指令。

当然ADC指令的进位也同样会影响标志寄存器当中的CF标志位。

所以我们要记住ADD指令和ADC指令

都会根据自己的运算结果来改变标志寄存器当中的CF位。

而ADC指令还会将CF标志位的值加入到运算当中。

那我们接着来看这段程序，这条MOV指令

是将SI寄存器所指向的内存单元的数传送到AX寄存器中，

也是将第一个数的第一个字，注意是两个字节，

传送到AX寄存器当中，然后用ADC指令将 AX寄存器当中的内容和DI所指向的内存单元中的内容，

也就是第二个数的头两个字节相加， 结果还保存在AX寄存器中。

然后再将AX寄存器中的内容传送到SI所指向的内存单元。

那我们要注意这里用的是ADC指令，为什么要用这条指令呢?

实际上过一会我们还会跳回到这里反复地 执行这段指令，从而将这两个很长的数累加起来。

因此在累加的过程中，低位相加如果产生了进位， 我们就得让这个进位传递到下一次的加法当中，

这样运算结果才不会发生错误，但我们还要注意第一次加这两个数的最低字节的时候，

本来是不应该带上进位的，所以我们得提前把CF标志位清零。

这里就用到了一条CPU的控制指令CLC， 它的作用就是把标志寄存器当中的CF位清零。

这样我们就完成了第一个字的累加。

然后我们执行了两次INC指令去递增SI寄存器，

然后用两个INC指令递增了DI寄存器，这就为下一轮的累加做好了准备。

不过这里有一个小问题，我们是否可以用ADD SI

2这样一条指令来代替这两条INC指令? 是否可以就留给你来思考。

那做好了准备之后，我们就应该想办法跳回到前面的指令，继续进行累加的操作，

这就会用到转移类指令。转移类指令的作用是改变指令的执行顺序。

我们现在要用到的是条件转移指令，而且是直接转移。

这里我们首先执行了DEC指令， 这条指令的操作是将CL寄存器的内容减1，

那CL寄存器中存放的是这个数的长度， 将它减1就说明我们已经完成了其中一个字的累加工作。

那如果减完之后，CL寄存器当中的值不为0， 这就说明我们还需要继续累加。

那这时就应该跳转到LOOP1这个标号继续执行， 这个操作就是由这条JNZ指令完成的。

这是一条条件转移指令，它所检查的条件 就是之前指令的运算结果是否为0，

其实准确地说，它并不是真的去检查之前一条指令的 运算，而是去检查标志寄存器当中的标志位。

标志寄存器当中有一个ZF标志位，如果DEZ指令的运算结果为0，

就会将ZF标志位置为1，代表这次运算的结果为0， 否则就会把ZF标志位置为0。

从模型机上来看，当执行刚才那条JNZ的转移指令时，

CPU会来检查标志寄存器当中的ZF位，从而决定如何改变下一条指令的地址。

根据我们刚才那个程序所需要的功能， 如果DEZ指令运算的结果为0，我们希望不转移，

而如果运算的结果不为0，那我们应该将下一条 指令的地址改为LOOP1那个标号所指向的指令的地址。

那么在这种情况下，我们就要从 这么多条件转移指令当中选择我们合适的指令。

根据刚才的分析，我们就应该选择这条JNZ指令， 它是在ZF=0的时候转移。

我们也注意到x86提供了很多种不同的条件转移指令，比如说有在CF为1的时候转移，

其实还有更复杂的条件，可以将多个标志位的组合作为转移的判断条件，

这样对于编程是非常方便的。

> 但同时我们也要想到CPU要提供这么多不同的条件转移的判断方式， 它内部的电路就会变得非常的复杂。
>
> 那我们还是回到这个程序，当CL寄存器的内容
>
> 不为0的时候，说明这个数的累加工作还没有做完， 那我们会跳回到LOOP1的标号这里继续做下一次的累加，
>
> 直到某一次CL减到0了，那这个条件转移指令的条件不满足，
>
> 因此会继续执行后面的指令。那我们发现后面还有三条指令，
>
> 那最后的这三条指令又是想做什么呢？这其实很简单， 就留给你来思考吧。
>
> 那最后一类就是控制类指令。
>
> 这里就包括我们刚才已经用过的CLC指令，
>
> 就是将CF标志位清零，还有一些对其他标志位的操作， 以及其他一些对CPU进行控制的指令。
>
> 那现在我们就使用了 这些简单的指令完成了这个累加两个数的程序。

> 即使是作为 基础的x86指令也很难在短时间内一一介绍，
>
> 而且也没有那个必要。大部分指令还是非常容易理解和掌握的， 能够读懂最基础的代码就可以了。
>
> 至于那些复杂的变化，用到的时候再查手册也来得及。

204-复杂的x86指令举例

> x86作为复杂指令系统的代表，自然会有不少相当复杂的指令。
>
> 在这一节我们将会看到其中有代表性的一些例子。

> 串操作指令是将存储器中的数据串进行每次一个元素的操作。
>
> 所谓一个元素可以是字节或者是字。
>
> 这个串可以很长，能够达到64KB， x86提供了5种不同的串操作指令，
>
> 并且还有3种重复前缀，可以与串操作指令配合使用。
>
> 这张表就展示了这5种串操作指令和3种重复前缀。
>
> 我们来选择其中一组进行介绍。这个指令的格式非常简单，没有任何的操作数，
>
> 它的功能就是在存储器中将指定位置的 一个字节单元传送到存储器的另一个制定的位置。
>
> 与它配合的经常是这个重复前缀REP，
>
> x86的体系结构中有很多种的前缀，这个前缀的
>
> 涵义是当CX寄存器的值不等于0时，就重复执行这个串操作指令。
>
> 那么很奇怪的是这个指令没有任何操作数。其实大家要注意x86当中有很多
>
> 这样的没有操作数的指令，但这并不意味着它们比那些有操作数的指令要简单。
>
> 因为它们不写操作数，不是因为没有操作数， 很可能是因为操作数太多了，实在在指令中写不下，
>
> 因此它们实际上是有一些隐含的操作数。
>
> 对于这这条串传送指令，它要传送的数据串称为源串。

源串的地址默认放在DS:SI这组寄存器指向的位置。

而要传送的目的，我们称为目的串地址，默认放在 ES:DI这组寄存器指向的位置，

而要传送的串的长度则放在CX寄存器当中。

我们可以看到，虽然没有写操作数，但是它实际有5个寄存器作为它的操作数。

不仅它有隐含的操作数，还有一些隐含的操作，除了

进行串的传送之外，在完成这个操作之后，硬件上还会自动完成这些操作：

第一修改SI和DI寄存器，以指向下一个串元素。

然后再判断是否使用了重复前缀， 如果是，则将CX寄存器的内容减1，

需要注意的是这些操作都是硬件自动完成的， 不需要程序员在软件中特别指定。

我们来看一个例子。假设我们在存储器中要进行一次

数据串的传送。源串的位置在12040这个地址开始，

一共三个字节，我们希望传送到12060开始的地方。

那我们编写的程序是这样的，假设事先已配置好了数据段寄存器DS为1000，

这个程序的前两条指令实际是将数据段寄存器的内容传送到

附加段寄存器当中。只不过段寄存器之间不能直接传送，所以借用了AX，

然后在SI寄存器当中保存源串的偏移地址， 在DI寄存器当中放入目的串的偏移地址，

这样DS和SI这组寄存器就指向了源串。

而ES和DI这组寄存器就指向了目的串。

下一条指令CLD，这是确定传送的方向，一会再进行解释。

然后在CX寄存器当中存入3，然后才是这条串传送指令。

前面加上了重复前缀，这样的配置就相当于 连续执行了三次这条串传送指令。

当执行第一次传送之后，第一个字节被传送到了目的串的位置，

传送完成后，SI和DI自动被增加，CX 自动被减1.这些操作都是由CPU完成的。

同时我还要说明，所谓的传送这个字节实际上是被CPU发起的向12040

地址的读操作，读入到CPU中，再发起一次向12060地址的存储器写操作，

写入到对应的字节单元。在第二次传送后， SI和DI又被加1，CX又被减1，

第三次传送完之后，虽然SI和DI继续加1，

但CX已经减为0，所以不再继续执行。

还需要说明一点的是串传送的方向也是可以设置的。

如果设置DF=0，则是从源串的低地址开始传送， 在传送过程中，SI和DI是自动增量的修改。

如果设置DF-=1，则是从源串的高地址开始传送， 传送过程中，SI和DI自动减量的修改。

这个表格就说明了SI和DI的修改方法。那如何修改DF标志位呢?

其实x86提供了两条控制指令，对标志位进行操作。

STD就是把DF标志置1。CLD 就是我们刚才的例子中的那条指令，是把DF清0。

> 这就可以确定串传送的方向。设置这样的方向
>
> 实际上是为了应对源串和目的串有可能重叠的问题。
>
> 我们简单来看一个释意。如果源串和目的串在内存中是互
>
> 相不重叠的，那这时候设置DF为0，或者为1，都没有关系。
>
> 但是如果你的源串和目的串有一个重叠，
>
> 那必须设置DF为1，从高地址依次向低地址开始传送，不然
>
> 图中绿色的重叠部分，就会在传送的一开始被覆盖，从而导致结果的错误。
>
> 那如果源串和目的串是这样的重叠的形式，则必须设置DF为0。
>
> 从低地址开始传送，原因也是一样的。
>
> 除了串传送指令，还有其他类型的串操作。例如在一个数据
>
> 串种，查找特定的数据，或者比较两个数据串是否相同。
>
> 这样程序员有了很便利的手段，对一大块数据进行操作。
>
> 因此串操作指令是功能非常强大的指令，不过由于数据串当中的
>
> 元素数量有可能很多，因此串操作指令的执行时间也可能很长，
>
> 这是需要注意的。最后我们从 一个有趣的例子来看一看x86指令的复杂程度。
>
> 这张图是x86指令的通用格式。
>
> 每一个小格都是指令格式中特定的位域。
>
> 那么我们可以人为写出一条指令来，这条指令是一个加法，而且
>
> 有一个前缀LOCK，这和我们刚才学到的REP一样，都是指令的前缀。
>
> 这个加法，其中一个源操作数是32位的立即数。另一个源
>
> 操作数以及目的操作数，是内存当中的一个32位的存储单元。
>
> 这个存储单元本应默认在数据段， 但这里强制指定为在附加段，
>
> 这个存储单元的地址由EAX寄存器，ECX寄存器
>
> 和一个立即数计算而得。要计算这个内存地址
>
> 需要一次乘法，两次加法得到偏移地址，再和段机制
>
> 进行移位并相加的操作，然后访问这个存储单元得到32位数。
>
> 在与1 2 3 4 5 6 7 8这个立即数相加， 然后再访问这个存储单元，将这个数存进去，
>
> 这条指令的编码一共有15个字节， 可以认为是一条最长的x86指令
>
> ，x86指令的复杂程度由此可见一斑。

> 编程人员只用给出一条简短的指令，
>
> 计算机就可以完成非常复杂的工作，这自然是一件很好的事情。
>
> 计算机似乎就应该这么设计，可惜世界没有这么简单， 有人提出了完全相反的做法，我们下一节再说。

205-MIPS体系结构

MIPS是精简指令系统的代表，采用了与X86相反的设计理念， 并引领了精简指令系统的潮流，

那就让我们一起来看一看这 究竟是怎么一回事。

要探讨MIPS指令系统，就得从它的设计者John Hennessy开始说起。

Hennessy 1977年进入斯坦福大学，

81年领导了RISC微处理器研究小组，他也被称为RISC的先驱。

RISC是精简指令系统计算机的简称，与之相对，之前的计算机上的指令系统就被称为

复杂指令系统，X86就是其中的代表。

后来，Hennessy共同创立了MIPS计算机系统公司，

90年代，他主要是在斯坦福计算机系担任系主任等职务，

2000年起开始担任斯坦福大学的校长。

由于他所设计的MIPS体系结构引领了精简指令系统的潮流，

后来还获得了IEEE的荣誉奖章，

而他所创立的MIPS公司也一度创造了辉煌。

在80年代末上市，后来被收购， 然后再一次上市，再一次被收购，

现在MIPS处理器已经不再应用在计算机产品中了，

但是在广义的计算设备包括数字电视，游戏机，网络设备

等领域仍然有广泛的应用，其实MIPS公司商业上的兴衰

也是诸多 RISC微处理器公司的命运写照。

第一代的MIPS是32位的， 在1985年推出了对应的处理器，R2000，90年，

R3000处理器对应着第二代的MIPS， 92年，MIPS扩展到了64位，94年，64位的MIPS又

进一步升级，96年的MIPS5并没有对应的处理器，

然后在99年，MIPS指令系统进行了较大的调整， 形成了MIPS32，到了99年，以MIPS5为基础，

推出了MIPS64指令系统， MIPS的设计指导思想非常的简单，

从它的名字就可以看出来。MIPS全称的含义 是一个流水线不会互锁的微处理器，流水线是

现代微处理器为提高性能而采用的一项技术，而流水线中的互锁

则是导致流水线性能降低的一个非常重要的因素。从这个名称也可以看出，MIPS的

指导思想是希望其指令的设计能让微处理器运行的更快，性能更好。

所以它主要的关注点是减少指令的类型，并且降低 指令的复杂度，所以在MIPS指令系统当中，

指令的总数是很少的。

而且每条指令都比较简单。它的主要目的就是希望可以用

一个非常简单的CPU来支持这样的指令系统。而CPU越简单

就可以运行的更快。假设要编写程序完成同样的任务，用MIPS指令编写，其指令数量是X- 86指令的5倍，

但是如果MIPS的CPU能够做到比X86 CPU快10倍，那它仍然获得了明显的性能优势。

这就是MIPS，同时也是RISC的设计思想。

那MIPS的指令是怎么体现它这样的设计思想的呢？ 第一，MIPS固定了指令的长度，都是32个比特，

也就说MIPS中的一个WORD,我们要注意这和X86中一个WORD是16位是不同的。

固定的指令长度，大大简化了CPU从存储器中取指令的工作。

不用像X86 CPU那样需要判断每条指令的长度。

第二，MIPS采用了非常简单的寻址模式，相比于X86提供的复杂多样的寻址模式，

虽然给编程带来了不变，但是大大简化了CPU访问存储器的控制逻辑。

第三MIPS指令的数量比较少，每条指令的工作也很简单，

基本上一条指令只完成一个操作，不像X86的指令，一条指令往往完成丰富的功能，

这样可以简化指令的执行过程。不但简化了CPU的控制逻辑，而且可以方便的

实现各种让指令并行执行的技术，从而提高CPU的性能。

第四，在MIPS指令系统中只允许LOAD和STALL这两种指令访问存储器，

而不支持X86指令中这些让算术指令访问存储器的操作。

因为访存是一个相对复杂的工作，

这种限制就可以让运算指令的实现变得非常的简单，但是我们要注意，MIPS

的这些特点让直接使用MIPS指令进行编程变得非常的困难， 因此，想要有高效率的MIPS程序，

必须要有优秀的编译器的支持。我们来看几个MIPS指令的例子。

例如加法指令，它的格式是ADD A,B,C

我们注意，与X86指令不同，MIPS的加法指令是三个操作数，这A,B,C可以是三个- 寄存器，

除了加法运算，这里还列出了减法，乘法，除法等等，还有逻辑运算，

还有左移和右移这样的移位运算。这类我们可以看出MIPS的运算指令

格式都非常简洁和统一，而且这些指令的操作数都不可以是存储器操作数。

要访问存储器，就必须使用专门的访存指令。我们来看一个例子，假设A是一个

100个字的数组，它的首地址存放在19号寄存器中， MIPS的寄存器编号用$符进行标记,

那如果我们想完成这样一个运算，也就说将一个变量H加上

数组A中的第三个元素，并赋给数组A的第十个元素，我们注意，

变量H放在18号寄存器中，而我们可以使用8号寄存器用来存放临时数据，

那么对应的MIPS指令需要如下几条，首先要用LOAD指令 将19号寄存器对应的地址加上偏移量12，因为MIPS当中

一个字是32位，所以第三个元素与首地址之间的偏移是12，

将存储器中的这个字装入到8号寄存器中，

这就相当于将数组A中的第三个元素赋给了一个临时变量，

第二句是一个加法指令，将8号寄存器与18号寄存器相加， 并将结果存放在8号寄存器。这就

相当于将数组A的第三个元素与变量H相加，还存在这个临时变量中，

第三条指令是将8号寄存器中的数

> 也就是运算的结果存到以19号寄存器为首地址，偏移量为40的内存单元， 这就是数组A的第十个元素的位置。
>
> 那我们用这三条MIPS指令就完成了这个功能。
>
> 而这张表列出了MIPS的所有的通用寄存器，总共有32个，
>
> 每个都是32位，相比于X86的寄存器，MIPS的通用寄存器是非常规正的。
>
> 这32个寄存器的编号从0，
>
> 一直到31，那我们可以用$符加上编号 进行指示，同时每个寄存器还有一个符号的名称，
>
> 并且约定了一些特定的用途，例如8号到15号寄存器，
>
> 又被称为T0，到T7的寄存器，用来保存临时的变量，
>
> 而1号寄存器，它的名称是AT,专门留给汇编器使用。
>
> 在编写汇编程序时， 我们可以用数字，也可以用名称，来表示这些
>
> 寄存器，例如这两条指令所表达的含义以及对应的二进制编码都是一样的。这两条指令
>
> 也是如此。如果我们直接用MIPS指令进行汇编语言的编程， 从T0到T7，S0到S7，这些
>
> 寄存器都是我们经常使用的。MIPS 的体系结构简明扼要，需要说明的就是这些。

> 我们已经了解了MIPS的设计理念，
>
> 那这样的理念究竟是如何实现的呢？在下一节我们将一起来探究
>
> MIPS的指令。

206-MIPS指令简介

> MIPS秉承着指令数量少，
>
> 指令功能简单的设计理念，那这样的设计理念
>
> 是如何实现的呢？在这一节，我们就将来分析MIPS指令的特点。

> 相比于X86指令所提供的动辄上千页的指令说明，MIPS指令
>
> 只用这两页纸就可以说清楚了。MIPS指令的基本格式
>
> 就分为这三种，R型，I型和J型。R型指的是寄存器型，
>
> I型指的是立即数型，J型指的是转移型。
>
> 我们用这张表对MIPS的指令进行不同纬度的分类，横轴是按照指令的格式
>
> 分为R型、I型和J型， 纵轴则是根据指令的功能类型分为运算指令、
>
> 访存指令和分支指令， 首先，我们来看指令格式为R型的运算指令。
>
> R型指令总共包含六个域，其中最高位的opcode域
>
> 是六个比特，最低位的funct域也是六个比特，
>
> 中间的四个域，均为五个比特， 我们分别来看各个域的用途。
>
> opcode域，用于指定指令的类型，对于所有的R型指令，这个域的值，均为零，
>
> 但这并不是说明R型指令只有一种，它还需要用funct域来更为精确的指定指令的类型。- 所以说，
>
> 对于R型指令，实际上一共有12个比特操作码，

那大家可以思考一下，为什么不将opcode域和funct域合并成一个12比特的域呢?

那样岂不是更直观明了吗？我们再来看这些5比特的域。

RS域，这个域通常用来指定第一个源操作数所在的寄存器编号，

rt域通常用来指定第二个源操作数所在的寄存器的编号，

rd域通常用来指定目的操作数的寄存器编号， 也就是保存运算结果的地方。

5个比特的域可以表示0-31的数， 正好对应Mix的体系结构中的32个通用寄存器，

还剩下最后一个域，它指示的是一位操作的位数。因为对于

32比特的数，5比特的域正好可以表示0-31的移位位数。

那这个域只是对于移位指令有用，对于非移位指令，这个域被设为0，

我们来看一个例子，这是将9号寄存器和10号寄存器 中的数相加，把运算结果保存在8号寄存器中，

那我们通过这条汇编指令的描述，如何得到mix指令的二进制编码呢?

这其实很容易。首先，我们查询mix指令编码表，就可以得到

加法指令的opcode域应该是0，funct域应该是32，

因为它不是移位指令，所以移位的域被设为0，然后我们根据这条指令的操作数

可以得到目的操作数，也就说rd这个域

等于8，第一个源操作数应该是9，第二个源操作数应该是10，

这样我们把各个域的数值转换成二进制数，填写到对应的位置， 就可以得到这条指令的二进制编码了。

MIPS指令系统简洁明了的规则可以让我们非常容易的对指令进行这样的手工编码转换，

同样也说明了CPU对这样的指令进行硬件的译码也会非常的方便。

如果指令中需要用到立即数，那么就要用到I型指令，

因为R型指令当中只有一个5比特的域，也就说移位这个域 可以用来表示立即数，那能表示的数的范围为0-31，

在程序中常用的立即数远大于这个范围，所以R型指令

并不适用，我们需要新的指令格式。这就是I型指令，I型指令的大部分域与R型指令是- 相同的，

I型指令的第一个域，也是opcode域，用于

指定指令的类型，但它没有funct域，所以不同的I型指令，及opcode域是不- 一样的。

第二个域rs，指定了第一个源操作数所在的寄存器编号，

第三个数rt用于指定目的操作数，I型指令与R型指令不同，它只有两个寄存器数域，

剩下的16位被整合成了一个完整的域，可以存放16位的立即数，

可以表示2的十六次方个不同的数值。对一般的访存指令，我们需要 用一个寄存器，加上一个立即数来指示一个内存单元，那么这个

立即数就是访存地址的偏移量，16位的立即数，可以访问正负32K的空间，

对于一般的访存指令来说，就可以满足了。而对于运算指令，虽然无法满足全部的需求，但是- 大多数情况下，

16位也可以使用了。在这一点上， 就可以体现出X86这样的CISC指令系统的优势，对于X86指令来说，

如果它想使用更大宽度的立即数，它可以很容易的扩展，因为它的指令本来就没有限制长- 度，但是

MIPS指令就不行。它的指令总长度就是32位的，再加上各个寄存器位域的使用，

所以I型指令最多只能使用十六位的立即数。

我们来看一个例子，对于加法，如果我们想让其中的 源操作数是一个立即数的话，就可以用add

i这个指令， 注意它和add指令是不一样的。add指令的操作数必须都是寄存器。

我们再来练习一下手工转换指令的编码。我们通过查指令编码表， 可以发现add

i指令的opcode域是8，从这一点我们也可以看出， add

i和add虽然只有一个字母的差别，但是他们指令格式是完全不一样的。

剩下的域我们通过分析这条指令的操作数就可以得到，

rs域，等于22，rt域=21，立即数域

等于-50，我们将这些数转换成二进制，就可以得到这条指令的编码了。

然后我们来看所有的分支指令，分支指令

是用于改变控制流的指令，其实就相当于X86当中的转移指令。

在MIPS中，分支指令也分为条件分支，和非条件分支

两种。对于条件分支有两条指令，beq和bne， 对于非条件分支，只有一条指令，j，

我们先来看条件分支指令，条件分支指令实际上是i型指令。

这就是两条条件分支指令，他们的opcode域分别是4和5，

我们以beq指令为例，它共有三个操作数，前两个是寄存器操作数，

第三个操作数是存储器地址，也就说一个立即数，CPU会判断第一个寄存器当中的数

和第二个寄存器当中的数是否相等。如果相等就跳转到 LE所指向的寄存器单元取出下一条指令，否则，

顺序执行deq之后的那条指令。我们需要注意，

这里和X86的条件转移指令有很大的不同。MIPS没有标志寄存器，它就在

一条指令当中即进行了比较，又完成了转移，

我们还记得MIPS的全称，就是为了减少指令流水线的互锁，也就说要尽量

避免不同指令之间相互的影响。而标志位这件事，很明显就是前一条指令运行的结果，可能

会对后面的某一条指令产生影响，这是MIPS指令设计时 要尽量避免的。所以BEQ指令也很好的体现了MIPS的这一设计理念。

我们来看一个例子。这段C语言代码是我们经常会写的。

如果把它转换为MIPS指令，是这样的，第一条BEQ指令，

如果S3寄存器和S4寄存器内容相同，则转移到

Q所对应的这行指令。那么S3和S4中保存了I和J这两个变量，

如果他们内容相同，会转移到这里，执行加法指令， 也就对应于F=G+H，如果他们不等，

则会顺序的执行下一条指令，也就一条减法指令对应于F=G-H

执行完之后，会跳过这条加法指令，然后进入后面的代码，

从条件分支指令的格式可以看出，目标地址只能使用十六位的位移量，

这是一个很大的局限，但是我们还得考虑如何充分发挥这十六位的作用。

如果以当前的PC寄存器 为基准，在MIPS中，指向下一条指令地址的寄存器称为PC，

类似于X86中的IP寄存器。这个寄存器，是指向32位寄存地址的。

如果以它为基准，十六位位移量可以表示出 当前指令前后2的15次方字节这么一个范围，

但是我们要注意一点，MIPS的指令长度固定位32个比特，因此每条指令的位置，

一定会在四个字节对齐的地方，这样地址，最低两位肯定为0,。

所以我们实际上可以用十六位的位移量去指示每四个字节为一个单位的地址。

这样就可以把目标地址的范围扩大四倍，可以达到前后128kB。

在这样的条件下，目标地址应该这么计算，

当分支条件不成立时，下一条指令的地址就等于当前的pc+4。

如果分支条件成立，那下一条指令的地址就等于已经加了4的pc， 再加上这个立即数乘以四。

然后我们来看非条件分支指令，相比于条件分支指令，有两个寄存器域用于

比较条件，那如果我们不需要判断条件，我们就可以想办法扩大目标地址的范围。

当然理想情况下是直接使用32位的地址， 但还是因为MIPS的指令长度固定为32位，而每条指令

至少需要有opcode域，指示它指令类型。

这就占用了六个bit。那我们把剩下的26个bit全都用于目标地址，

这就是J型指令。在考虑到MIPS指令是四字节对齐的这个情况，

对于这一行指令，下一条指令的地址的计算方法可以是将当前的pc加四之后，

取最高的四位，再加上J型指令编码中的26位，

然后在末尾填上两个零，虽然目标地址的范围还不能达到整个 4G的空间，但比之前的条件分支指令已经扩大了很多。

我们用一个例子来进行进一步的说明。

假设我们在高级语言中用的若干变量与寄存器的对应关系是这样的，

那我们就可以用这样一种方式来实现这段c语言的代码，

第一条指令是判断i和j是否相等，如果不相等， 则转移到else这个标号所对应的位置，

也就是执行一条减法指令对应于f=g-h，如果判断条件不成立，

也就是i=j的时候，顺序地执行下一条加法指令，

也就对应于f=g+h。然后用无条件分支指令

跳到else条件之后继续执行后面的程序。

我们现在已经知道这个J型指令的目标地址可以是当前指令

前后256MB的范围，那如果我们还想跳转到更远的地址， 应该怎么办呢？有一个很简单的方法

就是两次调用J指令，第一条J指令尽可能跳到最远的地方，

> 然后在那个目标地址再放一条J指令，像接力一样再跳一次。
>
> 这个方法很简单，但是用起来不算太方便，那么还可以用什么方法呢？
>
> 大家还记得我们曾说过间接转移指令吗？MIPS中也可以用同样的方法，
>
> 这就是jr指令。jr指令有一个寄存器操作数，
>
> 可以把要转移的目标地址放到寄存器当中，这样就可以使用32位的目标地址了，
>
> 但是这样的指令显然无法用J型指令来实现，
>
> 那么需要新增一种指令类行吗？其实也不需要，我们就用原来的r型指令就可以很好的实现。
>
> 只用占用其中的一个寄存器位域，然后新增一种function的编码就可以了。
>
> 这就是MIPS指令系统的核心内容，我们只用 熟悉这两页的内容就可以轻松的掌握MIPS的指令了。

> 我们已经介绍完了MIPS制定系统体系结构，
>
> 它不愧为精简指令系统的经典设计，指令简洁，而且精巧。