第三讲 算术逻辑单元

301-算术运算和逻辑运算

> 计算机的核心功能，就是运算。而运算的基本类型
>
> 包括算术运算，和逻辑运算。想要了解计算机是如何实现运算的
>
> 我们就得从算术运算指令，和逻辑运算指令开始说起。

> 这是我们非常熟悉的模型机。我们在最一开始介绍模型积时，就用了一条加法指令作为例子。
>
> 加法指令就是一种算术运算指令，当时这条例子中的，这条加法指令
>
> 是将R0寄存器当中的数，和存储器地址为6的存储单元的数相加， 并将结果放回到R0寄存器当中。
>
> 这条指令的格式比较接近X86指令，它可以在运算指令当中进行存储器的操作。
>
> 而现在，我们要选用MIPS指令作为示例进行讲解。
>
> 我们也知道，MIPS的运算指令是只能对寄存器进行操作的。例如这样的格式
>
> add表示是一条加法指令，dollar符号带上数字，是指对应的通用寄存器 所以这条指令要做的
>
> 是将九号寄存器的内容和十号寄存器的内容相加，并存到八号寄存器当中。如果我们写这么一段
>
> C语言程序，并且假设在程序中定义的这几个变量， 分别被分配到了八号、九号、十号的寄存器当中，
>
> 那在C语言当中所写的这条加法语句，就会经过MIPS的编译器，生成这样的
>
> MIPS的指令。那这条指令的操作数，全都是寄存器，所以它是一条R型指令。
>
> 首先我们查指令编码表，知道i的指令的opcode应该是0
>
> function是16进制的20，也就是10进制的32，
>
> 因为它不是一条移位指令，所以移位的域应该为0，然后根据
>
> 这条指令当中的操作数，我们可以知道，它的目的操作数，是八号寄存器， 所以rd对应是8，
>
> 第一个源操作数是九号寄存器，第二个源操作数是十号寄存器
>
> 所以这两个对应的域分别应该是9和10，根据这样的分析，我们就可以知道，
>
> 这条指令的具体编码。这条二进制的编码，就是存放在内存中的一条寄存机的指令了。
>
> 那CPU通过取址，就会把这条指令，从内存中取回来。
>
> 并放到ir寄存器中，也就是存放指令编码的寄存器。
>
> 指令一码电路，看到了这条指令的编码之后，
>
> 根据opcode域为全0，知道了这是一条r形的指令，再根据function域的值，
>
> 分析出这是一条add指令，因此，控制电路会像ALU，也就是运算器，
>
> 发出对应的控制信号，指明接下来要进行加法运算，于是同时，指令译码部件
>
> 分析出，第一个源操作数rs是九号寄存器，第二个源操作数rt是十号寄存器

我们注意看右边的内部总线，控制电路会选通对应的 源操作数寄存器，将其内容传送到ALU的输入，

同时指令译码部件还会分析出目的操作数rd，

为八号寄存器，因此控制电路还会将ALU的输出，联通到八号寄存器的输入，

在这样的设定下，ALU会根据控制电路给出的信号，

将两个输入的源操作数，进行加法运算，并送出运算的结果，

会被保存在八号寄存器中，这就完成了一次加法运算。

因此，对于MIPS指令来说，其核心的 算数运算指令，采用R型格式的有这几种:

这条加法指令，是将rs和rt寄存器的内容相加，并存在rd寄存器中，

这个addu和add，从运算操作上本身看来是一样的，我们通过后面的注释也能看出来，

但区别在于add这条指令加了一个注释的标记，

他指的是上面这条add指令，在加法运算产生溢出时，会像

控制器报告异常，由控制器进行相关的处理，而addu这条指令，在

发生溢出时不会报出异常。关于溢出的问题， 在讲解加法器的实现时，还会进一步解释，

除了加法，还有对应的减法，也同样分为sub和

subu两种运算指令，其操作与对应的加法指令是类似的。

这一类R型指令，它的源操作数都是在寄存器当中的，但如果

有一个源操作数是立即数的话，就需要采用另一种格式的指令，也就是I型指令。

这一条I型格式的加法指令的例子，我们是将22寄存器的内容

加上一个立即数-50，并将结果存放在21号寄存器当中。

这是对应的I型指令的格式，

我们同样也来看一看这条指令的编码，透过查指令编码表我们可以得到，这批指令的opco- de是8，

然后根据这条指令当中的参数，可以知道，源操作数的寄存器编号rs是8，

等于22，目的操作数的寄存器编号rt，也等于21， 而立即数这个域，应该是-50，

将这些数都转换成二进制，填入对应的区域，我们就可以知道这条指令的编码，

让CPU将这条指令的编码，取回放入ir寄存器当中后， 指令译码部件会根据opcode域

发现这是一条立即数的加法指令，因此控制电路， 会将ALU发出控制信号，只是接下来要进行加法运算，

同时，指令译码部件分析也会发现，其中一个源操作数来自于22号寄存器，

我们注意看右边的内部走线区域，22号寄存器的内容 会在控制电路的控制下，通过内部走线走到ALU的一个入口，

而这个运算的另一个操作数是一个立即数， 这个立即数是存放在指令编码中的，

因此，控制电路会从指令编码中提取出这个立即数， 并将其传送到ALU的另一个入口，

而这条运算指令指示的目的操作数rt，是第21号寄存器， 因此ALU的输出会被传送到21号寄存器，

这就是这条立即数加法指令的运算过程。

但是我们要注意的一点是，这个立即数，是16位的。

而另一个源操作数以及目的寄存器，都是32位的，那如何将一个

16位数和32位数进行相加呢，我们就需要进行一些转换。

转换的方法在MIPS的指令说明中很清楚地指出了，

上面这部份是刚刚介绍过的R型指令，这是我们刚才介绍的I型指令，

我们可以发现，在指令说明中，指出将rs寄存器的内容，

加上这个立即数的符号扩展，然后再将结果存放在r寄存器当中。

关于这条指令有两个说明，第一条跟刚才介绍过的一样，就是这条指令会

在加法发生溢出时，向控制器报出异常，交由控制器来处理。

第二个说明，指的是对立即数的扩展的方法，

立即数是16位的，那它如何被扩展到32位呢?是让它的高16位

是由这低16位当中的最高位，复制了16次，填充到高16位当中。

也就是说这32位当中的高16位，都和低16位当中的最高位，完全一样。

这样的扩展方法我们称为符号扩展，熟悉的符码规则的话，我们都应该知道

进行符号扩展不会改变这个数做为一个有符号数的数值，

也就是说如果你本来是+5，经过符号扩展还是+5，如果你是-7，经过符号

扩展，那还是-7，如果不了解这一点的话，就应该了解i进制补码的相关知识。

那么回到这个I型指令的说明，还有对应的i 的iu这条指令，它的运算过程和i的i是一样的。

区别只在于它在溢出时，不会产生异常。

这就是MIPS指令系统当中，核心的算术运算指令。

除了算术运算指令，还有一大类就是逻辑运算指令。

与算术运算指令类似，逻辑运算指令也分成R型和I型两种格式，

在R型当中，有and指令，就两个寄存器 当中的数，进行与操作，并将结果放在第三个寄存器中。

还有or指令， 是对两个寄存器当中的数，进行或操作。nor指令，

是进行或非操作，也就是先进行或，然后再取反， I型指令当中，有andi指令，

也是进行与操作，但其中一个操作数是立即数， 这与算术运算指令类似，但其不同点在于，这16位立即数的扩展

方式，它不是采用符号扩展，而是采用0扩展。

同样还有ori指令，他在其中一个操作数也是立即数。

那么这个立即数的0扩展操作是这样的， 扩展后的高16位全都是0，而不是低16位当中

最高位的复制，这种扩展方式，就称为0扩展， 音位算术运算指令会考虑将操作数看做是一个有符号

的数，在运算时要考虑正数和负数的这个性质，

而逻辑运算指令，是将操作数，视为一组二进制的01串，

> 这就不存在正数或着负数这样的概念，因此它采用了0扩展，而不是符号扩展。
> 同样我们也来看一个例子， 这里选的是R型的与指令，如果在C语言当中
> 写了这样的语句，那就会对应产生这样的MIPS指令， 我们也来看看它的指令编码是什么样的，
> 通过查指令编码表，我们可以得到opcode域是0，function域是16进制- 的24，
> 移位域同样还是0，然后根据这条指令的 三个操作数得到，rd域，也就是目的操作数是8。
> 第一个源操作数是9，第二个源操作数是10，我们在
> 将这些数都转换成二进制，就可以得到这条指令的编码。
> 如果CPU将这条编码从内存中取回，并放回IR寄存器中，
> 那控制电路这一次就会发出要进行与操作的控制信号。
> 在这样的控制下ALU就会将九号十号寄存器当中
> 内容，进行与运算，并将结果送回到了8号寄存器当中。
> 最后，我们来看一看这些算术逻辑运算的总体需求， 对于算术运算，
> 我们可能需要将两个32位的数相加，其运算结果，是一个32位数。
> 或着进行两个32位数的减法，运算结果，也是一个32位数。
> 除了进行运算，我们还要检查运算的结果是否发生溢出，
> 而对于逻辑运算，我们需要进行两个32位数的与操作，
> 结果也是一个32位数。第二种情况是执行或操作， 第三种情况是执行或非操作，
> 这些就对应了MIPS的核心指令中，提供的加法 减法、与、或、或非这些指令的功能。

> 现在我们已经了解了这些运算指令的基本特点，
> 下一步，我们就得分析这些指令在硬件电路上，
> 是如何实现的。

302-门电路的基本原理

> 现代计算机的CPU和其它很多功能部件
> 都是基于晶体管的集成电路，想要了解计算机组成的基本原理
> 还是需要有一些集成电路的基本知识， 就让我们从最简单的门电路的实现开始吧！

> 晶体管是构成现代集成电路的基本元件，
> 通常使用的是MOS晶体管，MOS是
> 金属，氧化物，半导体的缩写。而MOS晶体管又主要有两种类型，
> 一种我们称为N型MOS管，也简称为NMOS，这就是NMOS的符号表示。
> 它对外有三个连接，一个是源，一个漏，
> 一个是门。那么只看这个符号是非常抽象的，我们还是来打个比方说明吧！
> MOS晶体管其工作原理其实很像这个水龙头，电路中的电流
> 就好比水管中的水流，水流的来源也就是源，
> 而水流出的地方就称为漏，控制水流的开关就是门。

如果把门打开，水流就会从源流到漏，

也就相当于晶体管导通，电流可以从源到漏。对于NMOS来说，

其导通的条件是gate端连接了高电平，而当gate端连接低电平时

这个晶体管是不导通的。于NMOS相对还有另一种类型叫做PMOS，

它和NMOS的区别就在于当gate端连接低电平时

这个晶体管导通，而连接高电平时这个晶体管不导通。

这就好比我们有两种类型的水龙头，一种是把这个

把手向上拉才会出水，另一种是将这个把手向下压才会出水。

用这两种功能相对应的晶体管就构成了互补型的MOS集成电路，

也简称为CMOS。那我们就来看一看如何用晶体管构建逻辑门。

最简单的一种逻辑就是取非， 那么首先来看非门，

这是非门的逻辑符号a是输入，y是输出， 中间用一个三角再加上一个小圆圈构成。

这是真值表列出了在各种输入情况下输出应该是怎么样的。

非门的真值表非常的简单，当输入a为0的时候，输出y为1， 输a为1的时候，输出y为零。

如果要写成逻辑函数表达式，应该是在a上面加一条横线，但这个表示用执笔比较容易些，

而用键盘输入就比较困难一些。所以在编程时 经常会用这两种表达式来替代，在a前面加波浪线

或者加感叹号来表示取非的操作。

这张图则显示了如何用晶体管来构成非门。

我们可以发现非门非常简单，只需要用两个晶体管。

我们一起来看一看非门的工作过程。

首先来看当输入a为0时如何让输出y变为1。先来看最上面，

这个电路上方连接的是电源，也就是高电平用1来表示。

然后再来看下面，这个符号代表的D， 也就是低电平，用0来表示，

当a为0时，我们注意到a同时连接到了两个

晶体管的gate端。上面这个注意这个符号，

这是一个PMOS晶体管，当它的gate端等于0时，请想一想，它是否处于导通状态。

对于PMOS来说，它的gate端为0时是导通的，

所以电源这一端的这个1我们就可以认为它传导了过来。

这个1会传给输出Y，但只有这一点是不够的，

因为我们发现Y还连接下面这个晶体管，这个符号代表了一个NMOS， 同样它的gate端也与a相连，

连接的是0，那对于NMOS来说gate端连接的是零，它是什么样的状态呢?

其实它是关闭的，所以我们可以认为 连接的这个D的这个0是无法通过这个晶体管传导出来，

所以现在y这条线就由上面这个晶体管驱动，因此Y就输出为1了。

这就是a为0时，通过非门，让Y变为了1。

我们再来看另一种情况也就是a等于1时，y等于零， 同样电源是1，D是零，

a等于1时，上面这个PMOS的gate端是1，所以处于关闭状态。

而下面这个NMOS的gate端为1时，它就处于导通状态，我们可以认为连接

d的这个0通过这个晶体管传导了出来，因此这时y的输出就是0。

这两张图就说明了非门的工作过程。

我们再来看下一个非常常用的逻辑门也就是与门。

与门是有这个符号来表示的， 它的逻辑函数可以写成y等于a点运算符b，

其实这个点儿也是乘号的一种表示方式。

我们来看它的真值表。所谓与操作就是两个输入端都为1时， 输出才会为1，我们可以看第四行，

在其他的情况下输出全为0。

右边这个图是一个与非门实现原理，

它有四个晶体管构成，我们要注意虽然我们需要的逻辑功能是与门，

但是在实现中，与非门，比与门的实现更为简单。

所谓与非门，它的输出结果于与门正好相反，

也就是两个输入均为一时输出为0，其它时候输出都为1，

所以可以很简便的用一个与非门和非门相连就实现了与门的功能。

我们再来看一看与非门的工作过程，假设a等于1，b等于1，

对于与非门来说，这时候的输出结果应该是0，我们来看这个工作过程是怎么样的。

这时候a为1，我们首先来看b所连接的晶体管，

上面这是一个PMOS，它的gate端为1， 所以这个晶体管是关闭的，

那好我们暂时可以不管它，我们再来看b连接的另一个晶体管，在最下面这里，

它的gate端是1，所以它是导通的，那么它就会把0传导出来，

但这个0还需要经过一个晶体管才能传导到y， 这个晶体管是由输入a来控制的，

这也是一个NMOS晶体管，而我们发现现在输入a也是1,所以这个晶体管也是导通的，因- 此这个0

被继续往外传输就传递到了y，与此同时 我们还要看到上面这个PMOS晶体管也是由a控制的，

当gate端为1时，这个晶体管是关闭的， 因此信号y与电源之间并联的两个晶体管都处于关闭状态，

而与D之间串联的两个晶体管都处于导通状态。

所以y的值就是0。

右边这个例子则说明了a等于1， b等于零时，与非门的输出应该是1。

对于这个例子我就不详加描了。

简单的说就是下面两个串联的NMOS，有一个处于关闭状态，

因此y与D不联通，而上面两个并联的PMOS， 有一个处于导通状态。所以y与电源连通，

因此y的值就是1，这就是与非门的工作过程。

将它的输出y再链接一个非门就构成了与门。

然后我们来看或门， 我们与加号相同的这个符号来表示或操作。

这是或门的真值表，它的规则也很简单。

只有当两个输入都是0的时候，输出为0， 只要有任何一个输入为1，或者两个输入都为1，

输出都是1，那或门也是有几个PMOS和NMOS构成的。

在这里我们就不详细描述了。

最后来看一个逻辑门称为异或门，异或操作是指

它的两个操作数的值如果不相同则结果为真，也就是1，

如果两个操作数的值相同，则结果为假，也就是0，

它其实可以用我们刚才学过的与或非的操作组合出来，a的非和b进行与，

这个意思就是当a等于零时，a的非为1，而且b也等于1时，

这第一个括号中的表达式就为1，而当a等于1，b等于0时，

第二个括号中的表达式就会为1，中间进行或， 也就是这两个表达式只要其中一个为1，结果就是1，

而当a和b都等于0，或者a和b都等1时，这两个表达式

的结果都是0。取或之后最终的结果也还是0。

这就体现了异或运算的要求，这是异或运算的逻辑符号，

主体与或门类似，只是在左边多了一条线。

这是异或的逻辑函数表示，用一个圈内部带一个加号 当然这个符号很难直接用键盘输入，

所以在编程时也会用这个符号表示异或。

这是异或运算的真值表，当输入相同时， 比如都等于零，或者都等于1，

则输出为0，当输入不同时，也就是一个0一个1， 这样的形式输出则为1，

这就是异或运算。

现在我们已经学习了NMOS晶体管， PMOS晶体管，以及由它们构成的逻辑门，

包括非门， 与门，我们用与门则可以

进一步去实现计算机当中所要求的各种与相关的操作。

我们还实现了或门，用或门可以去实现各种或

运算指令，以及更为复杂的异或门，可以实现更多的功能。

现在我们已经学习的最基础的门电路，

别看它们简单实际上可以构造出非常复杂的功能了。

但是还有一个问题，这些电路是如何存储信息的呢?

这就是我们下一节要讨论的问题。

303-寄存器的基本原理

CPU中有很多的寄存器， 有临时保存数据的通用寄存器，

也有专门保存指令编码或者指令地址的寄存器， 它们都有存储信息的能力，

那在这一节，我们就来分析这些寄存器是如何实现的。

在CPU当中用来存放信息的非常重要的部件就是通用寄存器，

比如说零号通用寄存器，在mips的体系结构中，它就是一个32位的寄存器，

从电路实现上来说这32个比特都是同样的。

我们来看其中一个，它就可以用这样一个结构来实现。

这个结构就是我们将要介绍的D触发器。

触发器是具有存储信息能力的基本单元，

它有很多种类型，d触发器是其中一种。

其实触发器也没有那么神秘，它也是由与或非这样的逻辑门构成的，

仅就D触发器而言，以后有很多的实现方式，

在这里我们就不深入到它的内部实现细节，而重点关注它对外提供的功能。

这个D触发器主要有这三个接孔，一个数据输入， 一个数据输出和一个时钟输入。

它的功能表现是这样的，在时钟clock的 在时钟clock的上升沿，也就是clock从低电平变为高电平，

也可以说是从0变到1的时候，这是一个很短的时间， 在这时D触发器会采样输入端口D的值，

经过一段很短的时间会将这个值传送到输出端口Q， 在其它的时候也就是不在时钟信号的上升沿的时候，

无论输入端口D如何变化，其输出端口Q的值都是保持不变的。

当然这样说起来还是很抽象，我们通过一个形象化的描述来做进一步的说明。

就好比我们平常使用的照相机，我们把镜头

比作d触发器的输入端D，拍照的快门比作时钟端clock，

这台相机内部带有无线传输的模块，可以将其拍摄的照片传送到一个显示器上，这个显示器

就好比D触发器的输出端Q，那这样我们就把D触发器比作这个照相机和这个显示器。

当按下照相机的快门后，照相机会拍一张照片， 过一秒钟后显示器上显示出这张照片，

对于D触发器来说，这就叫做clock to

Q的时间， 也就是从时钟的上升沿到来开始直到数据出现在输出端为止，

这个时间是属于D触发器自身的特性，那对于这个D触发器所在的整个芯片还有一个

重要的特性就是时钟频率，在这里就好比我们约定 每十秒钟会来按一次快门，那这个系统的时钟频率

就是零点一赫兹。

最后我们再给这一套由相机和显示器构成的 D触发器加上一个输入也就是另一台显示器，

这样就会拍摄左边这台显示器上的画面， 并将它显示在右边这台显示器上。那好，现在我们就假设十秒钟到了，

我们会按动一次快门， 按动快门后照相机会拍摄下左边这台显示器上的画面，

并在一秒钟后将拍摄的照片传送到右边这台显示器上，这个过程就体现了在

时钟的上升沿D触发器采用输入端的信号，并在时钟上升沿之后clock to

Q的时间 将采样的信号从输出端口送出来，那之后输入端 也就是左边这台显示器上的画面可能会发生变化，

但是只要不按动快门，右边显示器上的内容是不会发生变化的。

也就是在下一个时钟上升沿来临之前无论输入怎么变化，输出都不会发生变化。

现在假设十秒钟又到了，我们要按动一次快门，

需要注意的是就和现实中的照相一样， 在按动快门的前后很短的时间内通过镜头看到的画面不应该发生变动，

否则就可能造成拍出的照片是模糊的。

对于D触发器来说，在时钟上升沿前后很短的时间内，输入端的

信号也不能发生变化，否则就可能造成无法正确的采样。

那么这也是D触发器的一个重要的特性，要求输入信号在时钟上升沿之前有一段很短的稳- 定时间，

很短的稳定时间，称为set up时间，在时钟上升沿之后也需要有一段很短的稳定时间，称为hold时间。

好我们假设在这一次按动快门时左边显示器上的画面是稳定的，

现在我们来按动快门，再过一秒钟， 这次采样的信号就被送到了输出端。

这就是一个D触发器简单的工作原理。

我们再来看一看两个D触发器相连的情况， 左边这个a相机就是第一个触发器，

它的输出连到了右边这个b相机的输入， 我们还是用同样的约定，每隔十秒钟来按动一次快门，

那当十秒钟到了的时候，这两台相机的快门会被同时按动，一秒钟后，

它们的输出画面都发生了改变，但我们要注意的是a相机拍摄下来的这个绿叶 是经过了clock

to Q时间才传送到了它的输出，而在此之前b相机

已经拍摄到了a相机此前的输出，也就是现在显示在最右边的显示器上的这个红色的叶子。

所以我们要注意的是这两个相连的相机虽然是同时按动了快门，但并不意味着最左边的画面

最左边的画面会一直传递到最右边的显示器上，而是将左边的相机之前存储的画面依次向- 右传递，

那么在一个复杂的系统中是有有很多的D触发器用各种不同的方式相连， 这样就可以在不同的触发器中存放不同的信息，

并且可以在时钟的控制下进行传递，而同样在时钟上升沿没有到来的时候，

无论输入发生什么样的变化都不会影响到后面触发器的输出。

通过这个例子，我们应该对D触发器的行为有了更加深入的了解。

我们再来看这些解释的时候应该不会觉得那么的陌生。

我把刚才说的这个过程用一个时序图的方式表达出来。

第一行是时钟信号，它是有规律的进行变化， 两个上升沿之间的间隔时间就称为时钟周期，

输入信号D则可能在任何时候发生变化， 比如在这个时候它由0变到了1，

但是因为时钟上升沿没有到来，所以输出端Q 并没有发生变化，直到时钟上升沿到来的时候，

> D触发器会采样输入端的信号， 并经过很短的clock
>
> to Q的时间在输出端体现出来， 因为这时候输入端是1，所以输出端也变成了1，
>
> 然后时间再继续，在这个时钟周期内，输入D又发生了变化，由1变成了0，
>
> 但是同样输出端Q没有发生变化，
>
> 直到下一个时钟上升沿到来，采样到了新的输入端的指令， 再经过Clock
>
> to Q输出端也变成了0这就是一个D触发器的基本工作行为。
>
> 那我们如果把很多个D触发器
>
> 组合起来，比如就是这32个D触发器，
>
> 那就可以构成一个32位的寄存器，当然这只是一个很简单的原理性实现。
>
> 用这样一个32位的寄存器就可以做成CPU当中的一个通用寄存器，
>
> 用同样的方法可以作出其它的通用寄存器以及PC，IR
>
> 这样的寄存器，再将这些寄存器于其它的由逻辑门构成的电路相连， 就构成了我们这个复杂的CPU了。

> 现在我们对寄存器的工作原理有了一个基本的了解，
>
> 我们也对寄存电路是如何运行的有一点点非常浅显的认识，
>
> 如果你还想了解更多就需要去学习其他的 专门课程了。

304-逻辑运算的实现

> ALU
>
> 也就是算术逻辑单元，它能够提供基本的算术运算和逻辑运算的功能。
>
> 那从硬件层面上，它是如何实现的呢？ 就让我们先从相对简单的逻辑运算开始分析。

> 现在我们已经掌握了门单元的功能，可以提供基本的逻辑运算。
>
> 例如这个与门就可以对两个二进制的输入进行与操作，并输出结果。
>
> 但这和计算机当中与运算指令所需的功能还是有差距的。
>
> 例如这条and指令，它的两个源操作数都是32位的寄存器， 它的目的操作数也是一个32位的寄存器。
>
> 也就是说它要同时对两个32位的二进制数进行与操作运算，
>
> 并得到一个32位的运算结果。那么怎么用与门来完成呢？
>
> 其实也很简单，我们就把32个与门并排
>
> 连起来，将32位的输入分别连接到这32个与门上。
>
> 可以标记为A0，A1一直到A31，这代表第一个操作数，而从B0，B1
>
> 一直到B31，这代表第二个操作数。这32个 与门的输出，分别记为Y0，Y1直到Y31。
>
> 它们就组成了一个32位的数。在这条与运算的指令控制下，
>
> rs所指定的寄存器的内容会被传送到这个端口。
>
> 而rt所指向的内容会被传送到另一个端口。
>
> 经过这些与门之后，得到的运算结果，会被传送到rd所指定的寄存器当中。

这样就完成了这条指令所要求的与运算。

与之类似，如果我们要完成这条或运算指令，则需要32个或门，

这样就构成一个可以完成32位或运算的单元。

那在ALU当中，实际上是包含了多钟不同的功能部件。

包括我们刚才提到的32位的与运算， 32位的或运算，还会有其它的逻辑运算以及算术运算单元。

那它们是怎样合成一个整体的呢?

对于这个整体的运算单元，它需要有一个32位的输入，然后在运算单元的内部，

分别连接到各个不同的运算功能部件的A输入端口，然后将另一个32位的

输入也在运算单元的内部分别连接到各个不同功能单元的B输入端口。

这样每个功能部件都按照各自的功能完成对应的与操作，或操作，

以及其他的操作，并产生对应的运算结果。

那现在的问题是，我们到底需要哪个运算结果作为输出呢?

这就还需要增加一个部件就是多选器。这里我们假定这个运算单元

当中包含四种功能，所以我们会有四个运算的结果，要经过一个四选一的多选器。

那样从四个选择当中选出一个来， 我们就需要一个两位的选择信号，当这个选择信号为00时，

在这个图中就会选择与运算功能部件输出的结果。

如果它是01时，就会选择这个或运算功能部件输出的结果。

这个多选器实际上也是由若干个门组成的。当然如果这个运算单元中包含

着更多数量功能单元，比如说8个，那么就要使用一个8选1的多选器， 而运算类型的选择信号也要扩大为3个bit。

那好经过这个多选器之后，我们就可以产生一个32位的输出。

那对于这一个完整运算单元来说， 当我们通过这个选择信号选择了对应的运算类型之后，

从运算单元的输出端口，我们就可以看到经过指定运算之后产生的输出。

回到之前提到过的在模型机上进行逻辑运算的这个例子，

如果我们要执行这条指令，实际上是在控制电路的控制下

将9号和10号寄存器的内容分别连接到ALU的两个输入端， 这里我们需要进行的与运算。如果按照上一页给出的例子，

这时候控制电路给出的选择信号应该是00。

最后控制电路还会将ALU的输出与8号寄存器的输入相连，

这就相当于左边这张图所显示的电路的连接。

最上面是由32个D触发器组成的8号寄存器，中间是9号寄存器，

然后是10号寄存器。9号，10号寄存器的Q端的输出会被链接到ALU的两个输入，

同时ALU的功能选择信号输入了与运算所对应的编码。

然后ALU的输出会被连接到8号寄存器的输入D端， 所以在某一个时钟周期内，ALU会按照输入的要求

> 完成相关的运算，并将运算结果送到输出信号上。
>
> 输入结果经过连线的传送， 会送到8号寄存器的输入端，
>
> 等到下一个时钟上升沿来临的时候，
>
> 8号寄存器就会将输入端的信号采样，存入到寄存器内部，并会在输出端表现出来。
>
> 之后的运算如果使用8号寄存器作为输入的话，就会使用这个新的值。

> 这一节我们讲解了 如何通过简单的门电路构造出不同的逻辑运算单元，
>
> 而且我们还知道了如何将不同的运算单元整合成一个完整的ALU。
>
> 但是ALU当中的算术运算单元又是如何实现的呢?
>
> 我们从下一节开始分析。

305-加法和减法的实现

> 加法和减法是两种基本的算术运算。
>
> 在计算机进行二进制的加减法的运算 和我们平时用纸笔进行的运算方法肯定是不一样的。
>
> 那它是如何实现的呢？让我们一起来分析。

> 我们先来看如果用手工运算的话如何进行二进制加法的。
>
> 这里有两个四位数的二进制数相加，被加数是1101，加数0110。
>
> 其实和我们做十进制的加法是一样的，首先从最低位开始
>
> 1加0等于1，然后第二个0加1等于1；
>
> 第三个1加1，因为我们是二进制就是逢二进一
>
> 座椅这里应该计为0并有一个进位。我们暂且记下，然后再算最高位
>
> 再算最高位1加0本来应该等于1，还要加上一个进位，所以应该等于0。
>
> 再往上进一位，这就是最后的（和）。因此，对每一位的相加来说
>
> 实际上要做这么几项工作，我们要完成两个一位的二进制数的相加。
>
> 而且可能有从进位传进过来的进位要参加运算，最后还有可能产生进位的输出。
>
> 那么现在就来看两个一位的二进制数的相加是如何用硬件电路实现的。
>
> 首先我们来介绍一个电路叫半加器，半加器的功能是将两个1位的二进制数相加。
>
> 这就是半加器，由一个异或门和与门组成。
>
> 它由两个输入端口 A和B，就是我们要相加的两个一位二进制数。
>
> 两个输出端口其中一个是S，相当于这两个一位二进制数
>
> 相加对应产生的（和），而C则是它们产生的进位； 有进位就是1，没有进位就是0
>
> 。那如果A等于0 C等于1那经过这个异或门之后的输出应该是多少呢?
>
> 我们知道对于异或门如果两个输入是不相同的，那输出为1；
>
> 两个输入是相同的时候输出为0。所以，这个时候输出应该为1。
>
> 而下面这个语文只有两个输入都为1时，输出才是1。
>
> 那么这时，输出应该是0。

这个结果也正好符合我们对这两个二进制数进行相加所得的和及

进位，那么对于AB来说总共有四种情况的组合 对应产生四种运算结果。除了我们刚才介绍的

AB为0、1的情况，我们可以再来看当AB均为1时

这时候运算的结果是十进制的2，也就是二进制的1、0；所以对应这位的和是0，

产生一个进位1。这就是半加器，但半加器相比我们刚才提到的功能需求还差一点。

它不能将第一位产生的进位作为输入参与运算，所以我们还需要增加这个功能。

这就是全加器，实际上全加器是由两个半加器构成的。这个图中

这两个绿色的异或门和与门就是一个半加器，而这两个橙色的异或门

和与门是另一个半加器，最后再添加一个或门就构成了全加器。

这个全加器有三个输入，除了两个原操作数

A和B，还有一个进位输入Cin。它的输出则有两个，

运算结果S和进位输出Cout。

这样三个输入总共有8种组合的情况，这是他们对应的输出，我么来看

其中的一个例子。比如A和B均为1热切进位输入也是1的时候

相当于三个1相加，用十进制来说运算结果就是3。

那么当前的位和进位组合在一起我们可以看到 二进制的1、11实际上也就是十进制的3.

这样前假期就能满足我们刚才分析的那三点要求。

那我们再回到刚才那个例子 来看看如何用全加器构成一个4比层的加法器

这是一个全加器，我们将四个全加器串联起来，前一个全加器的

进位输出镰刀后一个全加器的进位输入，这样就构成了一个四位的加法器。

那这个思维的加法器是如何工作的呢？ 还是用过这个例子来看，A有四个二进制位，我们用橙色来表示。

B也有四个二进制位，我们用蓝色来来表示。在运算时，需要将A对应的二进制位

连到这四个全加器的A输入端口，而将B对应的四个二进制位镰刀这四个全加器的B端口。

我们用不同的颜色标出来，对于整个加法器的进位输入我们视为0。这样最右边全加器的

三个输入都已经确定，很快就可以得到输出。S为1，Cout则为0。

这样第二个全加器的三个输入也都已经确定，从而可以产生对应的输出。

接下来S是1，进位是 0。对于第三个全加器，这时候三个输入也已经确定了。

那这时候因为有两个1相加，所以S应该是0，进位是1。

对于最后一个全加器，运算结果S是0，Cout是1。

这样作为一个整体 这个四位的加法器就得到了运算的结果，包括两个部分:

一部分是用S的端口出来的， 成为这个运算的（和）。还有一部分是从最左边的Cout出来，作为整个加法器的进位。

因此，这个四位二进制数的运算结果就包括这两个部分。

和这个四位的加法器一样，我们可以很容易的构建出32位的

加法器。也就是用32个全加器串联而成。

它的输入是两个32位数A和B在加法器的内部， 会分别连接到了32个全加器的A输入端口和B输入端口。

输入全加器的输出成为一个32位的加法器的输出。

整个加法器的进位输入连接到了最低位的全加器。

而最高位全加器的进位输出作为整个加法器的进位输出。

这样的加法器就可以满足加法运算指令的需求，对于这道指令我们

只需要将rs所指向的寄存器和rt所指向的寄存器的内容分贝连接到A端口和B端口

并将S送到RD所指向的寄存器 这就完成了对应的加法运算。那么ui的指令类似

addu的指令也完成了同样的运算功能，这两条指令的区别就在于

对溢出的处理不同。溢出是指运算结果超出了正常的表示范围。

由于在计算机中我们总是用有限的二进制位来表示一个数，

因此加法运算的结果有可能或超出这个有限的位数所能表示的范围。

溢出这种情况仅是针对有符号运算数来说的。

具体表现就是如果两个正数相加，结果变成了一个负数； 或者两个负数相加结果变成了一个正数 这显然是不正确的。

这种情况就是由溢出造成的。我们来看一个例子， 0011

加上0101如果用个四位加法器进行运算，

就会得到1000.如果我们把它们都看成无符号数那么0011就是3，0101就是5； 而1000就是8

。那么这个运算结果是正确的。

但如果我们把它看作有符号数

原操作数仍然是3和5单一运算结果1000他其实代表的是-8。

这样子的结果就是不正确的，也就是发生了溢出。

那么还需要注意的是进位和溢出的差别。

有溢出的时候不一定有进位，而有进位的时候也不一定有溢出。

我们分别通过例子说明，这还是刚才这个例子：

这个例子中的加法是加上是没有发生进位的，所以这就是一种有溢出无进位的情况。

我们再看一个例子：1110加上1100送到 四位的加法器中运算的结果1010

，同时有一个进位1。

这显然是有进位 但它却是无溢出的情况。为什么呢？我们来看，如果把它当作无符号数进行运算

1110是14，1100是12 如果带上进位一起考虑那就是正确的运算结果26。

如果不带进位则是为10 。因为四位的二进制位只能表示小于16的数，

所以无符号数的的运算结果可以默认为对结果进行 模16的运算。那只看着四位的话结果是10

； 相当于26 模16的值。

所以这也是正确的。而对于有符号数来说相当于-2加上-4 而这四位的结果就是-6

，这个运算也是正确的。

当然，从另一个角度来看因为溢出是针对有符号的数目的运算结果，超出了能表示的

范围，我们也可以把有进位这种情况看作无法iii的运算超出了能够表示的范围。

但是进位是很容易判断的，加法器本身就提供了这样的机制。

最高位的全加器的进位输出就是整个加法器的进位。那溢出又该如何判断呢？ 其实很简单，

就是当最高位的全加器的进位输入不等于它的进位输出的时候，这就是发生溢出了。

这是刚才的全加器，这个c31就是最高位的全加器的进位输入。

这儿，就是最高位全加器的进位输出。

把这两个信号连出来，判断它们是否不相等。那么用什么

方法来判断不相等呢？其实非常简单，就连接一个异或门就可以了。

当它们不相等时，异或门的输出为1 当他们两个都为0或者都为1时，异或门的输出为0

因此，就可以用这个信号来表示是否发生了溢出。

至于为什么用这个方法就可以判断溢出，留给大家自己思考。然后还需要说明的是，对于

一个加法器的硬件实现，它并不关心这两个输入数是有符号数还是

无符号数。或者说，它对于有符号数和无符号数的处理都是一样的。

全都是通过这套同样的硬件逻辑，进行运算，产生相同的结果。

至于参与运算的数到底是有符号数还是无符号数，取决于编程人员如何去看待它。

因此，是不是要处理溢出以及如何处理溢出，就不能只交给硬件来做。

不同体系结构有不同的方法。我们先来开MIPS处理溢出的方式。

它提供了两种不同的指令。如果编程人员想将操作数看作有符号数，需要处理溢出，

则需要使用add和addi这样的指令。

当这样的运算发生溢出时，会产生异常。

就是说，控制电路会检查加法器产生的overflow的信号。

如果overflow信号有效，控制电路就会当做一个异常的情况进行处理。

至于如何处理，我们会在讲到中断和异常时再做分析。

如果编程人员希望将操作数看作无符号数，也就是不对溢出进行处理。

那就需要使用这两条指令，addu和addiu。它们分别和上面这两条指令是对应的。

唯一的区别就是，在执行这两条指令时，控制电路不会检查加法器输出的overflo- w信号。

所以说，MIPS处理溢出的方式是提前做准备。

按照是否要处理溢出，采用不同的指令进行运算。

而x86则采用了另一种方式。它并没有根据是否处理溢出分为两种

运算指令，x86的运算指令，如果产生了溢出， 并不会直接由控制电路检查到并进行处理。

而是将加法器产生的溢出信号传送到了标志寄存器。

如果发生溢出，则会致标志寄存器当中的OF位为1.

如果没有发生溢出，则致OF位为0.

这就是x86的标志寄存器，其中第十一位是 OF位，这就是溢出标志。

则在后续的指令中，需要检查标志寄存器的OF位是否为1并进行相应的操作。

最后我们再来看一看减法运算。

其实减法是可以很容易的转换位加法的。例如我们要进行A-B的运算，那就相当

于进行A加上负B的运算，但我们需要注意的是，我们如何将B转换为负B呢?

我们知道，计算机当中使用补码来保存二进制数的。

B转换为负B，可不是在前面加一个负号这么简单。

那么补码表示二进制数取相反数，有这样一个转换的规则。

叫做按位取反，末位加一。这是什么意思呢？ 我们来看一个例子

例如我们有一个数X，那么所谓按位取反，就是将X中的每一位由0变成

1，由1变成0，那么得到了X按位取反以后的值。如果我们把这两个值相加，

那么它们的和，显而易见，每一位都是1.

在补码表示中，全1的这个二进制数，就代表着-1

那么由这个运算我们可以得到，如果将X和它按位取反后的值相加，

就等于-1，我们把这个等式进行一些变换，

就可以得到，X相反数，就等于对X进行按位取反。

然后再加上1

那么我们在加法器的基础上，用这样的方式，就可以很容易的实现减法器。

也就是A减B，相当于A加上负B， 实际上就等于A加上B的按位取反然后再加1

那么我们应该对加法器进行怎样的改造就能实现这样的操作呢？ 我们来看，这就是刚才的那个加法器。

原来的输入A和B都不变，我们增加了一个新的输入， 只有一个比特，称为减法模式。

它首先控制了一个二选一的多选器，如果这个信号为0

代表是执行加法操作，那么会将多选器的这一条通路选通， 也就是直接将B传送的下面的这些全加器。

这就跟刚才的加法运算是一样的。而且我们还注意到这个选择信号还连接到了最低位全加- 器的进位

输入，但是因为它现在是0，所以仍然和刚才的加法 操作是一样的，这时候就该执行一个加法的运算。

如果这个信号为1，代表要执行一个减法的运算。

那这个二选一的多选器会选择这条通路。

这条通路是将B这个信号的输入每一位都接上一个非门， 相当于按位取反，将按位取反的B送到每一个全加器

与A相加，我们还要注意，因为这个选择信号为1， 所以最低位的全加器的进位输入也是1.

这样就实现了对B进行按位取反，末位加一的操作，于是这个加法器也就变成了减法器。

> 这样我们通过这个改动，这个功能部件又能实现加法，又能实现减法，通过这个选择信号来进- 行选择。

> 我们已经实现了加法器和减法器
>
> 那现在是否可以构造出一个完整的ALU呢？先别着急，
>
> 对于计算机系统来说，我们不仅要求它功能正确，
>
> 而且希望有着更好的性能，那么这个加法器和减法器是否还有更大的提升空间呢？
>
> 我们下一节进行分析

306-加法器的优化

> ALU提供的加法，和减法， 就基本都是由加法器来实现的，
>
> 我们现在学习的加法器，是由一个一个的全加器串连而成，
>
> 它在性能上存在着很大的问题，而这个问题究竟是什么，
>
> 我们又该如何解决，在这一节，我们将来一起探讨这件事情。
>
> 我们还是来看这个四位加法器的
>
> 例子，在实现中，我们是用四个全加器，构成了这个四位的加法器，
>
> 我们注意到，当把这个加法器的输入都准备好时， 其实只有最右边的这个全加器的三个输入都准备好了，
>
> 左边这三个全加器，它的进位输入都还没有产生， 所以实际上只有最右边这个全加器，可以得到正确运算结果，
>
> 等他将进位的输出，传接到下一个全加器后，下一个全加 器才可以开始运算，进而产生新的进位输出。
>
> 这样进位输出，像波浪一样，依次从低位到高位传递，
>
> 这样的加法器，也因此得名为行波进位加法器。
>
> 它的英文名称也简称为RCA， 这个加法器，从结构上来看，低位全加器
>
> 进位输出，都连接到高一位全加器的进位输入， 它的优点是电路布局简单，设计方便，
>
> 我们只要设计好了全加器，连接起来就构成了多位的加法器。
>
> 但这缺点也很明显，也就是高位的运算必须等待低位的运算完成， 这样造成了整个加法器的延迟时间很长。
>
> 因此，我们要来分析一下，这个行波进位加法器延迟的情况。
>
> 我们还是以四位的形波进位加法器为例，我们把构成这个加法器的四个全加器，
>
> 内部结构打开，每个全加器个进位输出连接到下一个全加器的进位输入，
>
> 因此，我们现在看到的就是一个四位的形波进位加法器的门电路的实现。
>
> 要对一个电路的性能进行分析，我们就要找出其中的最长路径。
>
> 也就是找出所有的从输入到输出的电路连接中，经过
>
> 的门数最多的那一条，经过分析，我们可以发现，
>
> 实际上红色标明的这条路径，就是这个加法器延迟最长的路径，

也被称为关键路径。我们来做一个简单的分析， 对于最低位的全加器，它在A、B和Cin都已经准备好，

其实，输入信号进入到这块电路之后，在连接线上传递需要花时间。

称为线延迟，而经过这样的门，也需要花时间，称为门延迟。

在进行设计原理分析时，我们主要关注门延迟。

因此从这条通路来看，产生第一个s输出，需要通过两个门的延迟。

所以它显然不是最长的路径，因为往下，这里还会经过一个门， 然后再经过一个门，

要经过三个门，才会产生这个全加器的进位输出， 当然我们还要注意，为什么是从这里来，

而不是从这个与门的另一个输入，我们可以发现，这个与门的另一个输入，直接连接了Cin，

所以它不如从上面来的这条路径更长。当然，从A出发或着从B出发都是一样的，

所以对于第一个全加器，它的最长路径，是这一条。

然后进入第二个全加器，那么等传递到这个与门的时候，

我们会发现，这个与门下面这个输入，经过了三级门才到达。

而上一个输入，只要经过一级门就到达了， 所以显然最长路径需要从下面这条通路开始计算的。

然后，在这个全加器中，经过两个逻辑门，产生了进位输出，

依次往下传递，在每一个全加器中，都经过了两个逻辑门，

最后产生了进位输出。我们要注意，

在这个进位输出产生时，所有的全加器的S都已经产生了。

即使最晚产生的这个S，也比这个Cout早了一级门，

那如果我们把每一个门延迟都记为T的话，

首先在每个全加器内部，都要经过2T个延迟，然后还

要加上最开始的这一个门延迟，因此我们就可以计算出，总门延迟时间

就是2T乘以全加器的数量，在加上1T。

对于四位的形波进位加法器，一共是九个门延迟， 那如果是N位的形波进位加法器

那就一共是2N+1个门延迟， 这样对于32位的形波进位加法器来说，

N就等于32，所以一共是65个门延迟。

那65个门延迟，到底代表着什么呢?

这样看起来还是太抽象了，我们选一个大家熟悉的物品，有一些感性的认识。

比如说一款著名的智能手机，它内部的CPU，采用着28nm的制造工艺，

主频是1.3GHz，也就是时钟周期0.66ns，

这就是最近的两个时钟上升延之间的时间长度，因为加法器的输入

是来自寄存器，而且加法器的输出，包括运算的和，和进位输出， 都是要传递到寄存器保存起来。

所以说这些信号从前一级的寄存器，经过加法器的所有逻辑，一直到下一级寄存器的输入，

不能超过0.66ns，那实际情况又是怎么样的呢?

对于一个32位的形波进位加法器， 如果采用28nm的制造工艺，没延迟大约为0.02ns，

因为总共需要65个门的延迟，所以一共需要1.3ns， 这远远超过了我们所要求的0.66ns，

采用这样的加法器，它的主时钟频率，最多也只能达到769MHz，

这里，我们都还没有考虑寄存器的建立保持时间， 还有连线的延迟。实际的频率只会比这个更低，

所以说，这样的加法器，与我们现实中实际使用的加法器，性能差距是非常大的。

那我们应该如何进行优化呢?

首先，还是要分析一下这个加法器的问题所在，

影响性能的核心问题，就在于高位的运算，必须要等待低位的进位输出讯号

那我们是否可以提前计算出这些进位信号，以提高性能呢?

那我们就对进位输出信号，进行重点的分析，

它的进位输出信号记为Ci+1，Ci+1可以通过这样的逻辑表达式计算出来，

它的意思就是，如果这个全加器的三个输入， Ai，Bi或着Ci，只要任义其中两个为一，

则进位就是1，这是很自然的，三个一位的二进制数相加，

有两个或着两个以上的1，则自然会产生进位。我们再将这个算式当中的Ci提取出来，

就可以得到这样一个表达式。在这个表达式当中，Ai

和Bi都是在运算之初就是确定的，所以为了表达的简便，

我们再设定两个信号，一个称为Gi，它就等于Ai和Bi的与，

另一个是Pi，它就等于Ai和Bi的或，这样将Gi和Pi，代入到这个算式当中，

就得到了一个更为简化的算式。我们只需要记住的是，Pi

和Gi是由Ai和Bi产生的，他们都是在运算之初，就可以确定了的信号。

那么通过这样一个通用的表达式，我们来看，每一个进位输出讯号，是如何计算出来的。

首先，C1是通过G0、P0和C0计算出来的，

那么在运算之初，G0和P0都是已知的，C0也是已知的。

所以C1不用进行任何的等待，直接就可以计算出来， 而Ci，是由G1、P1和C1运算而得的。

在这里G和P都是已知的， 但是C1不是这个加法器的直接输入，

但如果我们想提前计算出C2，我们可以将上一个算式，代入到这个算式当中。

也就是C1转换成上一个算式，我们再将这个算式展开，

除了G和P之外，只有C0，所有的信号都是在运算之初就可以确定的，

这样，我们就不用等待最低位的全加器产生C1，

而是通过整个加法器的输入，直接计算出C2。与之类似，

原本C3也要等待C2的运算，那么现在将C2这个算式带进来，

变成这样，然后再进行展开。展开后的算式中，除了G和P

也只有C0，所以所有的信号也都是在运算之初就可以确定的了。

同样，我们还可以得到C4，越高位的进位输出信号，就越复杂。

但是我们可以看到，C4的算式当中，除了G和P之外，也只有C0

所以它也不用依赖低位的进位信号，通过这样的转换， 我们就有了提前计算所有的进位输出信号的方法。

那它在硬件上是如何实现的呢？我们以C4为例，

我们将刚才C4的这个表达式写成竖排的形式，它对应的硬件电路是这样的，

我们可以看到，最外层一共有五组元素进行 或运算，对应了最后的这个五数字的或门。

而其中第一个元素是G3，G3就是A3和B3的与， 所以在这里我们可以看到，A3和B3经过了一个与门，

与门的输出直接连到了最后的这个或门上。

第二个元素是P3和G2的与， 它就对应了这个与门，其中一个输入是P3，

而P3是A3和B3的或。另一个输入是G2， G2是A2和B2的与。

再往下看，第三组元素对应了这个3输入的与门。

第一个输入是P3，来自于这里。

第二个输入是P2，来自这里。第三个输入是G1，来自这里。

好，依此类推，就可以达到第四组元素，第五组元素。

这样我们就可以发现C4只需要通过A，B和C0就可以计算出来，

而且计算C4的延迟只需要三级门，这是一级，这是第二级，

这是第三级。其实由此类推，计算任意一级的进位输出

都只需要三级门延迟，与加法器的位数无关。

但只是如果进一步拓宽加法器的位数，像这样最后的算式 会变得越来越长，对应的电路就会变得越来越复杂。

那先不谈这个缺点，至少我们有了提前计算进位输出的方法， 用这样的方法实现了加法器

就被称为超前进位加法器。它的英文缩写也可以写成CLA。

这就是一个四位的超前进位加法器。

它仍然由四个全加器构成，但是每个全加器的进位

输入并不来自于前一级的全加器，而是来自一个统一的逻辑，

这就是刚才我们展示的超前进位的逻辑。对于每一个进位，

其实都只需要三级门延迟就可以计算出来。然后进入到全加器当中，

还需要经过一级门延迟才可以计算出对应的S信号。

因此，对于超前进位加法器总的延迟时间为4级的门延迟。

而四位的行波进位加法器总延迟时间为9级的门延迟。这个性能的提升就是非常可观的。

而且更为重要的是超前进位加法器它的门延迟与加法的位宽是没有关系的。

所以对于32位的加法器如果采用行波进位的方式，我们已经分析过需要65级的门延迟，

那如果采用超前进位的方式，理想情况下也只需要四级的门延迟，但可惜的是，

这也只是一个理想。因为要实现32位的完全的超前进位，电路就会变得非常的复杂。

> 我们可以看一看C31的超前进位的表达式就会变成这个样子。
>
> 也许这样就需要32输入的与门和32输入的或门。
>
> 这个在实现上就不太可行了。因此通常的实现方法，
>
> 是采用多个小规模的超前进位加法器拼接而成一个较大的加法器，
>
> 比如说，要生成一个32位的加法器，我们可以先做一个8位的超前进位加法器，
>
> 然后将四个八位的超前进位加法器用行波进位的方式连接起来，从而构成一个32位的加法器。
>
> 那么这样分析，一个32位的行波进位加法器它的门延迟是1.3个ns，
>
> 而一个超前进位加法器只需要0.08个ns，也就是四级门延迟。
>
> 如果我们采用四个超前进位加法器拼接成一个32位的加法器，那也只需要0.26ns。
>
> 这个0.26ns是怎么算出来的呢？就留给大家自己来计算了。
>
> 那么因为这个加法器的关键路径是0.26ns，
>
> 那么它就可以运行在3.84GHz的时钟频率下，那么至少它就不会
>
> 成为我们整个复杂的CPU设计的关键路径，不会降低整个芯片的使用频率了。

> 这个经过改进的加法器，
>
> 不仅功能正确，而且在性能和可实现性两个方面达到了比较好的平衡。
>
> 那现在把它和之前实现的逻辑运算部件整合在一起，
>
> 就构成了一个ALU可以提供基本的算术运算和逻辑运算的功能了。