第四讲 乘法器和除法器

401-乘法的运算过程

> 乘法是我们日常生活中经常使用的运算，
>
> 如果是两个非常简单的事物，我们用口算就能解决， 即使是比较大的数我们也只要用一支笔和一张纸
>
> 就能够非常轻松的完成。这种方法我们在小学的时候就已经掌握了，
>
> 那么计算机又是如何实现乘法的呢?
>
> 这个问题就比较复杂了，不过今天我将用这样最基本的工具
>
> 为大家来揭示计算机实现乘法的秘密。

> 好了， 那么我们就回到小学的时代来看一看如何用笔在纸上进行乘法的运算。
>
> 我们要计算的这两个数是2345乘以9876。
>
> 首先我们要做的是最低位的乘法， 5乘以6，那么5
>
> 6 30，这还很简单， 但其实这里隐含了一个复杂的操作，那就是去查九九乘法表。
>
> 无论这个乘法表是牢记在了你的脑子里，还是藏在了你的口袋里，
>
> 那么总之我们要进行查表，发现5和6，对应的是30，
>
> 所以把这个30拿过来，放在这里，我们写上0，标上进位3。
>
> 那么接着做下一步，下一步是4 6 24，
>
> 我们就要加上这个进位的3得到27，在这里写上7并标上进位的2， 然后是3
>
> 6 18，标上了0，写上了进位2。
>
> 然后是2 6 12加上进位2，,是14，我们写上了4，然后进位是1。
>
> 当然又没有更高位了，所以直接把这个1也写上，
>
> 现在我们就得到了一个结果14070， 然后很可惜这只是一个中间的结果，
>
> 那么即使是这个中间的结果，我们也经过了四次查表运算，还进行了四次的加法，
>
> 中间总共产生了四次的进位输出， 这么繁琐的一个过程，而且同样呐中间结果我们一共有四个。
>
> 最后我们还要把这样的中间结果再加起来
>
> 才能得到我们最后的成绩，这样看来乘法的运算，实际上是非常繁琐的。
>
> 那么我们能不能不要用这样繁琐的过程呢? 可不可以有一些简单的方法，
>
> 能够更好的解决这样的问题。

> 想要找到简单的方法，我们不妨先来看一看简单的情况，
>
> 我们找两个简单的数字，1000乘以1001，
>
> 同时我们也把刚才用过的那张纸找回来，放在一边作为一个参照，
>
> 那么对于我们新选的这两个数字，我们同样也进行乘法的运算，方法是一样的，
>
> 先拿乘数的最低位和被乘数的每一位进行相乘，

那么0乘1还是0，1乘以1就是1， 进行这四次相乘以后得到了四个数，组成了我们第一个中间结果。

然后是乘数的第二位，也和被乘数的每一位相乘，

从而得到了第二个中间结果，然后是乘数的第三位，得到第三个中间结果，

最后是乘数的第四位和被乘数的每一位相乘之后得到最后一个中间结果，

我们再把这些中间结果加起来就得到了我们最后运算的乘积，

也就是1001000，我们用普通的乘法运算的方法自然就可以完成这次乘法的运算，

但是对于这个例子的特殊情况我们发现实际上它展示了一种更为简洁的过程。

在介绍这个过程之前我们先把几个名词确定下来。

这个称为被乘数，这个是乘数，最后的运算结果是乘积，

那么对于这个例子，每个中间结果是怎么生成的呢？ 原则其实可以很简单，

我们不用再去关心九九乘法表是怎么样的，也不用再去关心每一位产生的进位是不是要加起来，

我们只需要观察当前进行运算的乘数的位，

如果这一位是一，那我们就直接将被乘数放置在与它对齐的位置上就可以了。

如果当前参与运算的乘数位为零，

那我们则直接将零放置在与它对齐的位置上， 这就是我们经过简化后的运算方法，

针对这样的特殊情况每一个中间结果的产生都不需要去查找乘法表，都不需要进行加法的运算，

要么是被乘数本身，要么是全零，根据当前参与运算的乘数位 做一个非常简单的二选一的操作就可以了。

那么对于十进制来说这只是一种非常特殊的情况，并没有普遍的意义，

但是如果我们的运算当中只会出现一或零，那这就是一个通用的方法了。

那很好，二进制正好符合这个特点，所以我们现在显示的这个例子，

如果它表示的是一个二进制数的相乘，我们就可以使用这个非常简洁的运算方法了，

而这也正是计算机最终选择了二进制的一个重要的原因。

其实最开始的电子计算机也是使用十进制的， 我们不妨回到二十世纪四十年代去看两台我们非常熟悉的机器。

左边这一台就是ENIAC它采用的就是十进制，

而右边这一台是EDVAC，它采用的是二进制， 那么ENIAC采用的是十进制导致它内部设计的电路非常的复杂，

而EDVAC采用的二进制之后就大大简化了控制逻辑。

关于这一点，冯诺依曼在他的报告中有详细的描述。

冯诺依曼在关于EDVAC的报告草案中 对于计算机应该采用什么样的进制，进行了详细的分析。

他主要说了这么几点， 一，组成计算机的电子管是一种全或无的设备，它适合表示只有两个数值的系统。

那么二进制就是一个合适的选择，这也是我们通常说计算机为什么要采用二进制的原因，

但是仅仅有这一点是不够的，还有一个重要的原因，就是要有适合的运算方法，

在这份报告中冯诺依曼还提到，二进制可以大幅地

简化乘法和除法的运算过程，尤其是对于乘法，如果采用二进制，

就不再需要使用十进制的乘法表，也不再需要产生每一个中间结果时所使用的加法了。

另外冯诺依曼还强调了一点，那就是十进制才是真正适合人类来使用的，

无论计算机当中采用什么样的进制，最终和人进行交互时，都应该使用十进制。

因此输入输出设备需要承担二进制和十进制之间的转换工作，

由此看来，如果想让计算机使用二进制而不是十进制，就必须要保证

二进制在这个数值的表示和运算方面都需要有很好的性能的提升，

要远远的超过格外的进制转换带来的性能损失这样才是可以接受的。

那好，我们再回来看这个二进制乘法的运算过程。

现在我们有了非常简洁的运算方法， 但是这个运算方法是否是和硬件的电路来实现呐，我们还需要来进一步的分析。

计算机的硬件资源和我们在纸上运算时可以比较随意地书写是不一样的，

那么还是用在纸上进行计算作为例子，

与刚才不同的是我们现在没有这么大一张纸了，我们把其中一部分剪掉，

只留下了能写被乘数，乘数和乘积的这三行的位置， 不过还有另一个变化就是我们不但有笔，而且还有橡皮，

我们之前在纸上写的数可以用橡皮擦掉。

那么就来看看在这么小的纸面上我们是否还能够完成刚才的乘法运算。

因为我们已经没有地方可以去记录中间的结果了， 而在运算的最开始用来记录乘积的这个地方是空的，

所以我们要想一想如何充分的利用这个区域， 在运算开始的时候我们可以把乘积记为0，

然后我们开始运算。

第一步，对应的乘数位是1，所以我们现在 已经知道这个中间结果就直接是被乘数，

但是我们现在已经没有地方可以记录这个中间结果了，所以每产生一个中间结果我们就直接把- 它加到乘积上。

于是我们就用橡皮把现在的乘积的数给擦掉，然后写上新的乘积，

其实也就是一个临时的中间结果。

好我们这样就完成了第一步。

在产生完这个中间结果之后，我们注意下一个中间结果实际上

是往左侧一位的。为了不至于过一会儿忘记了中间结果与乘积之间的对齐关系，

我们现在就重新写一遍被乘数，把它放在下一个中间结果应该出现的对齐的位置。

好了，然后我们再开始执行第二步的运算，

这步中间结果是0，我们可以认为把这个0加到了现在的临时结果上，

也可以认为不做任何操作，然后呢我们会再把被乘数向左移动一位，

再检查最后的乘数位，还是0， 然后我们再把被乘数向左移动一位，

再检查乘数的对应位，这是1，所以这一次的中间结果就是被乘数1000。

我们到现在的被乘数和乘积之间的对齐关系，把这个中间结果加到当前的乘积上。

我们注意 现在我们已经检查完了乘数的每一位，所以这个运算已经完成了，

现在在这张纸的最下面这一行就是我们得到的最终的运算结果。

这样我们就可以在没有空间，记录中间结果的情况下完成了这个乘法的运算。

这就是一个适合硬件的乘法运算的过程。

现在我们已经将乘法运算改造成了适合硬件实现的方式，

那么下一步我们就要开始制造我们自己的乘法器了。

我们希望之后可以不再用纸和笔进行乘法的运算， 而用我们自己制造的乘法器自动地完成我们想要的

运算的乘法。

402-乘法器的实现

如果希望CPU支持某一种运算， 我们就需要用硬件电路来实现对应的功能部件。

现在呢我们现在已经有了一种乘法运算的方法，适合硬件来进行实现。

而且，我们还掌握了一些硬件电路的基本知识。

我们就可以一起来尝试如何构建一个硬件的乘法器。

现在我们来看乘法器是怎么实现的。

我们用一个简单的二进制乘法作为例子，1000乘以1001， 这是两个四位的二进制数相乘，

那么为此我们要实现一个四位的乘法器， 首先我们需要一个寄存器来保存被乘数，

被乘数寄存器是一个八位的寄存器。而且带有左移的功能。

它有一个左移的控制信号输入，当外部的控制逻辑将这个信号

视为有效时，在下一个时钟顺延到来的时候，被乘数寄存器当中的内容就会向左移动一位。

第二个寄存器是乘积寄存器，这也是一个八位的寄存器。

用来保存运算的结果。

被乘数寄存器当中的内容和乘积寄存器当中的内容需要进行加法运算， 这里我们就需要进行一个八位的加法器。

它会将被乘数寄存器当中的内容和乘积寄存器当中的内容进行相加，并将结果再送到乘积寄存- 器当中。

另外，我们还需要一个寄存器来 保存乘数。

这个寄存器只要四位就可以了。但有一点比较特殊，乘数寄存器的最低位被连到了控制逻辑，

也就说控制逻辑可以观察当前乘数寄存器的最低位，并据此来生成相关的控制信号，

这些控制信号就包括是否要让加法器进行 加法运算，以及是否要让乘积寄存器保存当前的运算结果。

另外，这个乘数寄存器还需要有向右移位的功能， 同样也是由控制逻辑发出的信号来进行控制的。

现在我们就有了一个完整的乘法器。就来看一看这样的一个乘法器是如何工作的。

首先我们要做的工作是初始化。

也就是对这三个寄存器的内容进行设置。

我们现在要进行的运算当中被乘数是1000， 也就要把1000放置到被乘数寄存器当中去。

那这个4bit的数是如何放到8bit的寄存器当中去的呢?

我们现在已经看到，我们需要在高位填上4个0，为了便于区分，

我在这里将我们运算所得的数用粗体表示， 那么在被乘数寄存器当中现在的第四位1000

就是我们这个运算本身所有的数， 而高四位的4个0是这个乘法器的系统搜补入的，所以这里用斜体来表示。

那这就是被乘数寄存器的内容了。然后我们看乘数寄存器。

乘数寄存器很简单，

它是四位宽的，对应了4位的乘数，直接填入即可，然后是乘积寄存器。

现在运算工作还没有开始，所以乘积寄存器中当前的内容是0.

现在初始化工作已经完成，我们就开始运算了。

第一步， 请大家回忆我们用纸笔是如何进行这个运算的。

第一步我们应该做什么呢? 首先是检查乘数寄存器的最低位，

看是否为1，现在发现它是1，那么就需要将 被乘数寄存器当中的内容和当前乘积寄存器当中的内容进行相加，

我们把这一步命名为第一步之后的一个小的步骤，称为1A，

在这一步控制逻辑会控制加法器，将

加法器的两个输入，也就是被乘数寄存器的内容和乘积寄存器当中的内容进行加法运算，

然后将加法器的运算结果送入 乘积寄存器，在下一个时钟上来临的时候，

因为乘积寄存器的写入信号是有效的， 所以它就会采样输入端的数据，饼保存到寄存器当中。

这样在这个时钟上升元之后，乘积寄存器当中的内容 就变成了00001000，

然后我们来看第二步，

第二步应该做什么呢?我们还是需要回忆之前讲过的适合硬件做的乘法运算。

这时候我们需要对被乘数进行左移，以便为下一次运算

进行准备。所以控制逻辑会给出左移的信号， 那么在下一个iii来临的时候，被乘数寄存器就会完成一次左移，

最左端的0被丢弃了，在最右端补入一个0，

然后我们再来看第三步，第三步又是要做什么呢?

我们注意乘数寄存器是有右移功能的， 控制逻辑会给出右移的控制信号，

当下一个时钟上升来临的时候，乘数寄存器就会向右移动一位。

这样做只是为了方便对乘数的对应位进行判断，

在硬件上，如果要依次的对乘数从低位到高位进行逐位的判断， 那就要用硬件来记住之前判断到了哪一位，

那与其记住这样的信息，补入把乘数每次向右移一位。

这样每次只需要判断乘数寄存器当中的最低位就可以了。

而且我们根据乘法运算的过程也可以知道， 我们对乘数的使用是由低到高逐位使用的。

一旦使用过之后就不会再用了。所以我们可以放心的把乘数进行右移， 并把刚才的最低位丢弃。

所以在做完加法运算之后的 第二步和第三步，实际上是在为下一轮的工作做准备。

但是我们还需要进行一个判断。那就是到底有没有下一轮。

这就是第四步，由控制逻辑进行判断， 现在是否已经完成了运算，对这个四位的乘法器来说，

一共要产生四个中间结果并将他们相加，所以与刚才类似这样的工作一共要执行四轮，

简单的计数对硬件来说是非常容易的， 所以控制逻辑可以很容易的判断出当前是否已经是第四次循环了，

当然现在不是，所以我们需要进行下一轮。

第二轮的运算方式和刚才第一轮是完全一样的。

首先判断乘数寄存器当前的最低位是否为1， 如果为1，则要进行加法并进行更新乘积寄存器当中的内容，

但是现在是0，所以不需要做这一步工作。

那控制逻辑就不会产生对应的加法控制信号，以及要写入乘积寄存器的写控制信号。

于是直接进入第二步，还记得第二步是什么吗?

第二步是让被乘数寄存器左移一位。

所以在下一个时钟上升元之后，被乘数寄存器就会左移。

然后是第三步，第三步 是让乘数寄存器右移一位。

再过了一个时钟上升沿，乘数寄存器就右移了一位。

然后是第四步，第四步是做什么?

第四步是判断当前的运算是否已经结束了。

当然现在还没有到第四轮，所以运算还没有结束。

因此我们还继续执行第三轮，每轮的第一步都是一样的，

就是判断乘数寄存器的最低位，在这里我们就可以看出来这样机制的好处。

就是每轮的操作内容完全是一样的， 硬件上不需要设计额外的机制去知道当前是进行到哪一轮，

也不需要智力因素的介入去判断 当前到底要分析乘数的哪一位或者如何进行中间结果的对齐等工作。

它所需要做的都是很机械的左移， 右移，判断当前 乘数寄存器的最低位，这些非常简洁明了的工作。

那在第三轮的第一步我们发现乘数寄存器的最低位还是0，

那我们就不再详细描述第三轮当中的每一步了。我们会直接跳到第四轮。

但请先想一想当我们到了第四轮的第一步的时候，被乘数寄存器当中的内容

应该是什么？乘积寄存器当中的内容应该是什么， 而乘数寄存器当中的内容

又会是什么？如果你有纸笔的话不妨在纸上把你所设想的这三个寄存器当中的内容

写下来，然后我们一起对比一下你的判断是否正确。

好来现在我们假设硬件已经依次的做完了

第三轮的第二步第三步和第四步。

现在我们到了第四轮的第一步，我们注意到被乘数寄存器又左移了一位。

它的最低3位现在都是移位补入的，”乘积寄存器“内容没有变化

而”乘数寄存器“又往右移动了一位。

现在只剩下最低位的这个1，是我们初始化的时候填入的，而它

高位的3个0，我们看到它都是斜体的，就标明了是后来移位补入的。

那么在第4轮的第1步，我们还是要判断乘数寄存器的最低位 这时候我们发现是1，这时候控制逻辑会做什么工作呢?

其实和第1轮一样，它就会控制加法器进行加法运算，将当前”被

乘数寄存器“的内容和”乘积寄存器“的内容进行相加，并向”乘积寄存器“发出信号

让”乘积寄存器“在下一个时钟上升沿的时候，采样加法器的输出结果。

我们要注意，”被乘数寄存器“当中的内容已经移到了位置。

从这里我们也可以看出 用这样的方法，控制逻辑不需要关心当前”被乘数寄存器“和”乘积寄存器“

的内容进行相加时如何进行对齐，它只要保证每一次往左移一位

然后再需要进行加法的时候，它们的对齐关系一定是正确的。

那么在执行完加法运算，并将运算结果保存到”乘积寄存器“当中之后

我们还需要对”被乘数寄存器“和”乘数寄存器“进行一个移位的操作。

那么还是和刚才一样，你可以在纸上写一下你所预想的这三个寄存器当中的内容

我们将直接跳到第4轮的第4步。

这时候我们注意，”乘积寄存器“当中的内容已经发生了变化 而被乘数和乘数寄存器也都发生了移位

这时候”被乘数寄存器“当中，真正的被乘数1000,已经移到了高4位

而低4位都是后来移位补入的。我们再看”乘数寄存器“

现在”乘数寄存器”当中这4个0全都是后来移位补入的，原先的乘数已经全部移出了

这也说明，我们对乘数的每一位都进行了判断，我们已经完成了所有的中间结果。

当然对于硬件控制逻辑来说，它不可能像我们这样用人眼 去观察“乘数寄存器”当中乘数是否已经全部的移出

但它的判断方法更为简单。

还是看这是否是第4次循环，当然我们知道，这确实是第4次 循环，所以在这个时候，这个乘法运算就已经完成了。

所以在最后一轮当中，被乘数和乘数这两个寄存器的移位并没有实际的意义。

好了，现在我们已经得到了这个乘法的乘积，它就在现在的”乘积寄存器“当中。

这个运算结果是01001000。

我们可以看到，这和我们用纸笔算出来的结果是一样的。

现在，我们用这个四位乘法器的例子，知道了一个乘法器的运算过程 那我们就来对这个运算的过程进行一个总结。

这是一个N位的乘法器的工作流程图。当我们做好了初始化工作之后

第1步，就是检查“乘数寄存器”的最低位 如果这个最低位是1，那么就将“被乘数寄存器”和

“乘积寄存器”的内容相加，运算结果还放入“乘积寄存器”当中。

之后，再执行第2步，也就是将“被乘数寄存器"的内容左移

一位，当然如果在第1步的检查当中，最低位等于0

那就跳过第1a步，直接执行第2步；执行完第2步之后

第3步，是将”乘数寄存器“的内容右移一位

第4步，是检查是否已经到了第N次循环 如果不是，那就意味着运算还没有结束，我们只需要

回到第1步，再检查当前”乘数寄存器“的最低位 并执行相应的操作就可以了。如此往复循环，直到

有一次在第4步检查，发现已经是第N次循环了。

那这就意味着这个乘法运算的工作就已经完成了 我们所需的运算结果就放在”乘积寄存器“当中。

这就是一个N位乘法器的工作流程。

[music]

[music]

现在，我们只要有足够的晶体管 构造出合适的逻辑门和触发器

就可以制造出一个完整的、可以正常工作的乘法器了。

虽然它的实现还非常地初级，但

它肯定可以正确地完成我们想要的乘法运算。

403-乘法器的优化1

计算机的最大优势在于它的运算速度。

因此，当我们设计计算机当中的一个功能部件时

在保证其功能正确性的前提下，就得考虑，它是否拥有足够好的性能。

那么在这一节，我们就要一起对这个乘法器，进行性能上的分析和优化。

首先，我们来快速回顾一下N位乘法器的工作流程。

当我们做好初始化工作之后，首先检查“乘数寄存器”的最低位

如果最低位为1，那么就将“被乘数寄存器”和“乘积寄存器” 的内容相加，并将结果放入到“乘积寄存器”当中。

那么这一步实际上是控制逻辑给出了若干个控制信号

包括，让加法器完成加法运算，并给了“乘积寄存器”写入的控制信号。

因为加法器的输出是连接到“乘积寄存器”的输入端的，所以在下一个时钟上升沿来临

的时候，“乘积寄存器”就会对加法器的输出进行采样，并保存起来。

这就完成了这一步所指的工作。

然后控制逻辑会向“被乘数寄存器”发出“左移”的控制信号

在下一个时钟上升沿来临的时候，“被乘数寄存器”就会完成左移一位的工作。

当然，如果在第一步中检查的“乘数寄存器”的最低位是0

那控制逻辑就不会给出让加法器进行运算，和让“乘积寄存器”进行写入的控制信号

而会进入到第二步，完成“被乘数寄存器”左移的工作。

再到下一个时钟上升沿，“乘数寄存器”会发现其右移的控制 信号有效，所以它就完成一次右移的工作。

然后判断是否已经到了第N次循环，如果没有，说明运算还没有结束

再回到第1步继续执行；如果发现已经是第N次循环，说明运算已经完成

运算的结果就在当前的“乘积寄存器”当中啦。

我们分析这个工作流程，我们可以发现第1a步执行这个加法

我们需要一个时钟周期，第2步的左移又需要一个时钟周期 第3步的右移还需要一个时钟周期。

因此这个乘法器工作时，每执行一轮，都需要3个时钟周期

那如果是一个32位的乘法器，那就大约需要100个时钟周期。

我们来想一想有没有优化的办法。

首先，我们很容易想到的优化，就是刚才 说的这些加法和移位的操作，是否能够并行起来。

虽然我们在介绍流程时 这些步骤是一步一步进行的，这样只是便于我们理解。

在实际的实现上，它们并不需要一定有一个先后的顺序。

因此，问题就在于我们的实现结构能否支持让这些加法和移位并行操作。

我们知道，寄存器的内容只会在时钟上升沿来临的时候发生变化

而在其他时候，无论输入端如何变化，寄存器的内容都不会发生变化。

因此，我们就用这个乘法器的当前状态为例，进行分析。

假设现在时钟上升沿还没有来临，那“被乘数寄存器”的输出就是它当前所保存的内容

00001000，这八位信号会被送到加法器的输入端，而加法器的另一个输入端

连接的是“乘积寄存器”的输出端，因此，现在的信号值是全0。

而当前“乘积寄存器”的最低位为1，控制逻辑会据此 产生相关的控制信号，包括让加法器进行加法运算

这样加法器就会产生对应的运算结果。

与此同时，控制逻辑还会给出写输入信号 但是现在时钟上升沿还没有来，所以“乘积寄存器”其实什么也不会做。

那我们注意，在这个时候，控制逻辑实际上可以同时给出“被乘数寄存器”的移位信号

还可以给出“乘数寄存器”的移位信号。

那现在，控制逻辑就将刚才流程图当中的第1a步

第2步和第3步所对应的控制信号都置为有效了。

但是因为时钟上升沿还没有来，所以这些寄存器的内容都不会发生变化。

那么当时钟上升沿到来的时候，这些寄存器就会根据输入改变其内容。

“乘积寄存器”会将输入端的数保存起来，“被乘数寄存器” 会向左移动一位，“乘数寄存器“会向右移动一位。

而这些操作都是在同一个时钟上升沿到来的时候完成的。

请注意，我们现在假设这个时钟上升沿来了，注意观察这3个寄存器 它们是同时发生变化的。

而且根据我们对寄存器的了解，我们已经知道，在时钟上升沿之后很短的一个时间

寄存器所改变的内容就会传送到输出端。

因此我们可以注意到，”被乘数寄存器“的内容现在已经改变，因此

很快，它的输出也会变成00010000，并传递到加法器。

但是我们可以放心的是，即使这个信号经过了加法器，并产生了新的运算结果 这个运算结果也不会改变”乘积寄存器“的内容。

因为等这个信号传递到了”乘积寄存器“的输入端时 已经过去了一段时间，这时候”乘积寄存器“对输入端的采样工作已经完成

其输入端信号的改变不会影响这个寄存器的内容了。

这样我们就在同一个时刻完成了所有的加法和移位的操作。

那我们再回到刚才的那个流程图。

在这个流程图当中，这3个顺序执行的操作就可以进行优化

而我们这32位的乘法，也就不应该再需要100个时钟周期了。

经过优化以后的工作流程，在最开始的第1步，和最后的第4步，还是跟原来一样的

而中间的这3步，我们可以把它并排地放在一起。

当第1步检查”乘数寄存器“的最低位，如果最低位是1的时候 第1a步、第2步和第3步将同时进行

如果最低位为0，则通过控制信号的不同，不执行第1a步，而同时执行第2步和第3步

完成之后，就直接进入第4步。

这样我们每一次循环只需要一个周期 我们用这个很简单的优化，就把性能提升为了之前的3倍。

这样我们就有了一个速度更快的乘法器。

现在，我们已有了一个经过优化后的乘法器了。

当然，这个乘法器其实还有很大的性能提升空间

比如说，对于乘法运算，每一个中间结果实际上不需要依赖其他的中间结果的运算

那我们可不可以把所有的中间结果一次性都算出来?

然后再将这些中间结果进行相加，而不要经过一轮又一轮的迭代

这样是不是可以更好地提升性能呢？至于具体怎么做

就留给你自己分析吧。

404-乘法器的优化2

当我们购买一台计算机时，不仅希望 它的性能足够好，还希望他的价格足够低。

那对于CPU这样的集成电路芯片来说， 它的价格的一个重要因素就是其中晶体管的数量。

或者说是芯片的面积。因此，在我们设计各个功能部件的时候，

如何减小芯片的面积也是一个重要的优化方向。

我们还是对照这个乘法器的结构图， 我们来一起分析在哪些地方我们只是为了实现的方便

而设置了不必要的硬件资源。这些都是我们可以优化的方向。

首先我们来看被乘数寄存器。这是一个八位的寄存器。

但是实际上我们的被乘数只有四位。

那我们可能要说那是因为我们要进行移位啊，而移位是为了和乘积的中间结果进行对齐

然后进行运算，这就是这个乘法运算的要点之一啊，

我们先不管这些，在这里显然用八位的寄存器来保存了一个四位的被乘数

这就是浪费。至于如何解决，或者能不能解决，我们一会再说，我们先把所有的浪费的情况都- 找出来。

都找出来。好这里是一个，然后第二个， 这个乘数寄存器是个四位的，那这个就没有浪费了，因为乘数本来就是四位的，

但我们仔细想一想我们会发现这个乘数实际上每一个周期都会往右移动一位，

所以在这个乘数寄存器当中虽然一开始是没有浪费的， 但是在运算的过程中每个周期有效的数字会减少一位，

所以逐渐就会出现浪费的位数，而且越来越多， 那好我们又找到了一个浪费的地方。再来看第三个，

乘积寄存器，最后的乘积确实需要8位才能保存， 但是问题在于这个乘积寄存器在不断的保存中间结果，

而中间结果一开始并不是8位的。

当被乘数还集中在最低四位的时候，这个运算结果实际的有效数字

只有最低的四位，只有随着被乘数寄存器不断的左移，

那乘积寄存器当中的有效数字还在不断地增加，最后才打到了8位。

所以说对于乘积寄存器，虽然最终它没有浪费， 但是在最开始，它的高四位是浪费的。

而且其中的有效数字每个周期也是增加一位。

这样我们就找到了第三个浪费的点。第四点，在这里加法器，

实际上每一次在将被乘数和当前的乘积进行相加的时候，无论被乘数左移到什么程度，

实际上只有真正有效的那四位的被乘数需要和乘积寄存器当中对应的位进行相加，

被乘数寄存器当中从右边补入的那些0实际上 是不用参加运算的。所以这个8位的加法器也是一个浪费的点。

那好，现在我们就找到了四个浪费的点。

那我们就来看一看如何进行优化。

这就是刚才我们找到的那四个浪费的店。

我们先来看第一个。被乘数寄存器有8位宽，而且带左移的功能。

但是他有效的数字时钟只有4位，那我们用一个非常直截了当的方法，

既然你的有效数字只有四位，那么就给你一个四位的寄存器，用来保存这个被乘数。

那我们可能要说，左移怎么办呢？我现在只有一个四位的寄存器，那一左移

岂不把一个有效的数字给丢掉了吗？被乘数这四位可都一直要保持参与运算呢？ 那方法也很简单。

既然我削减了你的地盘，那我也相应的减少一点你的任务， 你就不需要左移了。所以这个寄存器也被取消了左移的功能，

从实现上来说，没有了左移的功能，它的实现还会变得更为简单，

但是我们不能一味的为了减少浪费而削减功能。被乘数左移是为了和中间的结果进行对齐，以便

进行运算。这件事情总得有一个功能部件来做。那么等会再去看看别的地方。

至少现在第一个问题已经解决了。被乘数寄存器缩减为4位

而且取消了左移的功能，在被乘数寄存器这个地方已经没有了浪费的现象。

然后我们来看乘积寄存器。乘积寄存器是8位宽

的，但是初始时有效数字只有四位，而且每周期增加一位。

对于乘积寄存器，虽然它开始时有浪费，但是因为它最后的结果需要8位宽，

所以我们没有削减它的宽度，

但是一样，我既然保留了你的地盘，就得增加你的任务，

刚才我们不是取消了被乘数寄存器的左移功能吗？

被乘数寄存器左移就为了跟乘积寄存器当中的内容进行某种方式的对齐。

那既然它不能左移了，那你就右移呗，反正都是你们两对齐， 要么它向左，要么你向右，效果都是一样的。

所以我们就在乘积寄存器上增加一个功能。

就是右移的功能。既然我们用乘积寄存器的右移

代替被乘数的左移，那么乘积寄存器初始值的放置位置也发生了一点变化，

我们把它放置在高四位的地方，

并在这个寄存器中间加一条虚拟的线来标志高四位和第四位的分界。

那好，我们对乘积寄存器的改造暂时先到这里。我们给增加了右移的功能，

而且乘积的初始值防在其中的高四位，随着运算的过程不断的右移，

现在从运算功能上来说我们已经解决了被乘数寄存器被取消左移的这件事情。

虽然乘积寄存器本身的浪费问题我们还没有解决。

现在我们来看下一个问题。就是 加法器有8位宽，但是参与运算的有效数字实际只有4位。

那现在这个问题已经很好解决了。

因为乘积寄存器已经被削减到了四位，而乘积寄存器，初始值放在了高四位的地方，

所以我们就将加法运算改成被乘数寄存器的这四位和乘积寄存器的高四位进行运算。

这里我们需要仔细想一下。如果想不明白可以拿出纸笔再重现一下

乘法运算的过程。这样我们就解决了三个问题。现在还有最后一个问题。

我们还需要一个乘数寄存器。它是四位宽的，而且有右移的 功能。但是其中有效的数字每周期是减少一位的。

我们仔细看一看这个功能的描述，再看一看现在这个结构，我们真的还有必要设置这个寄- 存器吗?

如果没想明白，就请多看几遍这个乘数寄存器的功能描述。

一开始是4位宽，而且带右移，

它所需要的有效数字每周期减少一位，再仔细想一想，并且可以看一看

乘积寄存器还空余的这一部分，它是不是正好是一个最开始有四位宽的寄存器而且带右移，

当然因为乘积的有效位每个周期会右移一位，

因此这个控制的部分每个周期会减少一位，但这不正好是乘数寄存器所需要的功能吗?

所以，我们其实就可以把乘数放在乘积寄存器的第四位，

从而可以取消乘积寄存器这个部件。同时也解决了乘积寄存器在一开始有浪费的这个

问题。这样我们就完全解决了刚才分析出的这四个

硬件资源浪费的问题。与此同时，我们的控制逻辑也需要 做一些调整。现在需要把乘积寄存器的最低位连

接到控制逻辑，这也就是我们当前所需要观察的乘数的对应位，

如果发现这位为1，控制逻辑就需要向加法运算的控制信号，并向乘积寄存器发出写入的控- 制信号，

并向乘积寄存器发出写入的控制信号，但这个写入只发生在 但这个写入只发生在这个寄存器的高四位，

然后每个周期会发出右移的控制信号，那么随着乘积寄存器的右移， 原先存放在第四位的乘数就在逐渐的被移出，

而原先只在高四位的乘积则逐渐占满了乘积寄存器。

而在这个过程中，控制逻辑每一轮都可以观察到 乘数的对应位，从而产生对应的控制信号，

因此这个结构可以完成刚才我们所介绍的那个乘法运算流程的所有工作。

当然这只是一个四位乘法器的例子。

但我们可以很容易的推广到N位，这就是一个经过优化后的N位乘法器。

我们需要一个N位的寄存器保存被乘数，还需要一个N位的加法器

进行中间结果的运算，再需要一个2N位的带右移的寄存器同时保存乘积和乘数。

这就是我们进行了性能优化和面积优化之后的乘法器的实现结构。

通过对乘法器的分析和优化，

我们要意识到两件事情。首先，这些原理性的描述虽然易于理解，

但是不见得适合真正的实现。而从另一个角度来看， 对于这些原理性的实现，我们只要采用一些很简单的方法，

就能够在性能和面积上获得很好的优化效果。

当然，我们现在有的这个乘法器和 现实中我们真正使用的乘法器还是有相当大的距离的。

也就是说，还有更多的其他的优化方法可以使用。

如果你对这些方法有兴趣，可以进一步的深入挖掘 和学习。

405-除法的运算过程

> 在加、减、乘、除这样的基本算数运算当中，
>
> 除法是最为复杂的。因此，我们想要实现硬件的除法器， 还是从最简单的情况开始说起。

> 我们还是采用模仿纸笔进行除法运算 的方式，来回顾一下除法的运算过程。
>
> 这里是两个十进制的数， 被除数是1001010， 除数是1000，
>
> 这是两个经过精心挑选的数，用它们进行除法运算， 运算的过程中只会出现0或者1，
>
> 所以看上出又像是二进制表示的数。所以我们通过这个例子可以 看出十进制的除法运算，和二进制的除法运算之间的联系。
>
> 这个运算的过程，我们已经非常熟悉了，所以我们快速地来看一遍。
>
> 首先将除数和被除数，从高位开始对齐，然后将对齐的这部分进行一个减法，
>
> 当我们发现减完的结果是一个正数，也就是所谓的够减的时候，我们就在上面标一个1，
>
> 然后把减完的差放在下面，再从被除数后面的位拿一个数下来，
>
> 这时候我们用眼睛看一看就可以发现，现在这个数还不够去减这个除数，所以我们在上面标- 一个0，
>
> 然后再多拿一位，还不够，那么我们在最上面再标一个0，然后再拿一位下来， 现在我们发现，这时候已经够减这个除数了。
>
> 好，那我们在这上面标一个1，然后把除数写在这里，
>
> 再执行一次减法，减完的结果已经比除数小了，而且
>
> 被除数那里也没有多余的位可以拿下来继续运算，这样我们这个除法就已经完成了。
>
> 先得到最上面的这个数，就是这个除法运算的商， 而最下面的这个数，我们称为余数。
>
> 如果用一个式子来表达这四个数的关系，那就是被除数，等于商乘以
>
> 除数，再加上余数。当然我在描述这个除法运算的时候，非常的口语化，
>
> 而且也有人的智力因素参与其中，比如说用眼睛看一看够不够减，之类的。
>
> 这样的描述过程是很难用硬件去实现的。
>
> 想要设计一个硬件的除法器，首先我们要从硬件实现的 角度出发，来考虑如何描述除法的运算过程。
>
> 那我们用这样的方式再来看另一个例子。
>
> 这里就是两个二进制数的除法了，被除数是 00000111，这就是十进制的7，
>
> 而除数是0010，这就是十进制的2， 所以这就是做7除以2。
>
> 对于硬件实现，我们首先要考虑的，是这个运算的操作数的宽度。
>
> 这里我们可以看到，被除数是一个8位宽的数，而除数是一个4位宽的数，
>
> 因此在这样的情况下，即使高位是0，我们也不能将这个0 省略，因为它们实实在在地，在硬件中占据了一个位置。
>
> 很显然，我们可以把被除数放在一个8位宽的寄存器当中，
>
> 同时我们还要注意，被除数是在不断的和除数进行减法的操作。
>
> 在经过几轮之后，减法的运算结果最后就产生了余数。

所以，如果我们将每次减法运算的结果都放回到被除数的寄存器当中， 那么这个寄存器又可以被称为余数寄存器。

现在我们就有了第一个部件，就是一个8位宽的余数寄存器。

然后我们还需要一个部件来保存除数， 还需要一个部件来保存商，那我们在纸上进行除法运算的时候，

第1步，会把除数抄写到这里， 并将被除数当中最高的四位和这个除数进行对比。

经过比较，我们发现被除数的这一部分是比除数小的， 所以不能执行这个减法。

因此，我们在于当前除数最低位对齐的这个地方，

标上我们得到的第一位的商，也就是0，目前执行的这个过程， 我们就称为第一轮。然后我们会怎么做?

我们接下来要做的就是往右挪一个位置，把除数再抄一遍。

好，我们把除数抄在这里，如果我们忽略这些数字底下衬着的这些带颜色

的框，那它实际上和在纸上进行除法运算我们所写的内容是一样的。

而如果我们带上了这个写了除数的长条形的方框，那我们又可以发现

这个除数0010好像是在这个方框中右移了一位，

这样我们就可以考虑在硬件上用一个带右移的寄存器来实现这个功能。

那么这个时候，又需要用被除数和除数进行一个减法运算了。

那现在够不够减呢? 实际上还是不够的。其实我们不用去找应该对其哪几位进行减法的比较，

如果我们直接把这个写的除数的长条方框看成一个8位的寄存器的话，

那就可以把其中没有标明数字的地方都看做是0，每次都是把这个标着

除数的8位的寄存器的内容和上面标着余数的8位寄存器的内容进行比较，

那么现在这个除数还是更大一些，所以这一轮也不能执行减法，这样我们在商这个位置再标- 一个0。

我们要注意，在纸上运算时，我们直接就写上了第二个0， 而现在，我们在硬件上只给商预留了四个位置，也就是说，

是一个四位的寄存器。所以这个操作就好比将商的这个寄存器

往左进行了一个移位，并在最右端补入了一个0，

那这就是第二轮。然后我们继续，再将除数往右边挪一位，

再和被除数进行比较，现在还是除数更大，所以这一次 还是不能执行减法，而商再写上一个0，

这就是第三轮。我们再看下一轮，再向右移一位除数之后， 我们发现现在被除数更大了，可以执行一次减法了，

减完的结果就是00000011。

因为执行了一次减法，所以我们在商对应的位置标上了1，

也就相当于对于这个商的寄存器往左移动之后，在最右端补上了一个1，

然后我们把这个减法的结果更新到余数寄存器当中去， 这就是第四轮。但现在还没有结束，我们还需要

把除数再往右移一位，并和当前的余数进行比较， 我们发现还可以进行一次减法运算，

> 所以在最上面，我们再把商从最右边移入一个1，
>
> 然后再回来看下边，我们再执行一次减法，得到了一个结果为1， 并把这个结果再更新到余数寄存器当中去，
>
> 这就是第五轮。现在我们就已经得到了最终的结果，
>
> 最上面是商的最终结果：0011， 最下面是余数的最终结果，也就是1。
>
> 0011是十进制的3，所以7除以2，商为3，余数为1，
>
> 这样的结果显然是正确的。我们再来整理一下这个例子。
>
> 对于被除数，我们选择的是十进制的7，对于二进制，它可以用8个比特来表示，
>
> 而且被除数我们可以视为初始化时候的余数 除数12，我们用4个比特的寄存器来保存，
>
> 而且在运算的过程中，我们可以看作将除数不断地右移，并和被除数进行相减。
>
> 对于余数，虽然它最终的结果很小，但是因为它可以与被除数共享一个寄存器，
>
> 所以我们也用8个比特来表示。
>
> 对于商，在这个例子中是3，我们要注意到的是，在运算过程中，它的每一个比特是从高
>
> 到低，也就是从左往右依次生成的，可以视为不断地左移而形成，
>
> 在这里我们也用4个比特的寄存器来保存。
>
> 我们要注意，这个位宽和乘法运算正好是相反的，
>
> 在乘法当中，我们一般会约定乘数和被乘数的宽度是一样的，如果
>
> 两个都是4位宽的，则乘积最多为8位宽， 所以在除法当中，我们也用了类似的约定。
>
> 如果被除数是8位宽，则除数和商都约定为4位宽。
>
> 现在我们就有了一个适合硬件实现的除法运算过程了。

> 从这个事例我们就 能够看出，除法运算确实是非常复杂。
>
> 不过现在我们应该很有信心， 只要我们能够用适合硬件的方式把这个运算过程描述出来，
>
> 我们就应该能够把这个除法器设计出来。

406-除法器的实现

> 现在，我们已经将除法的运算过程，用适合硬件实现的方法描述出来了，
>
> 那么就可以着手开始设计真正的硬件的除法器了。
>
> 那么在这一节，我们将首先整理出一个除法器的工作流程，
>
> 然后通过一个事例来分析除法器的结构和它的工作原理。

> 我们首先来看一个32位除法器的工作流程。
>
> 首先进行初始化工作，然后进入正式的工作步骤。
>
> 我们要注意，余数和被除数是共享一个寄存器的。
>
> 所以第1步，是用余数寄存器的内容减去除数 寄存器中的内容，并将结果保存到余数寄存器当中；
>
> 第2步，是检查余数寄存器的内容，如果当前的余数大于等于0
>
> 那说明这次减法操作是正确的； 那下一步执行2a这个分支，将商寄存器中的内容

左移一位，并将空出的最右边这位设为1；

然后执行第3步，那就是将除数寄存器右移一位

但如果在第2步检查余数时，发现余数寄存器的内容小于0，

那这就意味着，刚才这次减法操作是不应该进行的，

所以必须回退、消除这个影响，那么2b的这个分支，就是回退第1步的减法操作

然后还会将商寄存器左移一位，但是空出来的最右位是设为0。

这一步做完之后，同样也会进入第3步；

完成第3步之后，第4步是判断当前是否为最后一轮循环。

这是一个32位的除法器，也就是除数是32位的，那么一共要进行33轮循环，

如果不到33轮循环，那就回到第1步继续执行； 如果已经是33轮循环，说明运算已经完成，

最终结果存放在当前的商寄存器和余数寄存器当中。

这就是32位除法器的工作流程。

为了简便起见，我们还是用一个4位的除法器作为例子来进行说明。

一个4位的除法器需要一个8位的"余数"寄存器， 需要一个8位的"除数"寄存器，而且带右移的功能，

还需要一个4位的"商"寄存器，带左移的功能， 最后需要一个8位的ALU，支持加法和减法运算。

这里我们要注意的是，在乘法器当中，只需要一个加法器， 而在除法器当中，需要支持加法和减法两种运算。

因为正常的流程中，是需要减法运算。

而刚才我们在讲解工作流程时提到，有时候需要回退已经完成的减法操作，这就需要用到加法。

那么这个ALU的输入，就来自除数寄存器和余数寄存器， 它的输出还连到了余数寄存器。

那除了这些部件，我们同样还需要控制逻辑，控制这几个寄存器和ALU的工作。

现在我们就来看一看这个除法器的工作过程。

我们注意右下角，我们还是结合这个7除以2的例子来进行说明。

首先是初始化工作。我们将8位的被除数放入"余数"寄存器，

然后将4位的除数放入"除数"寄存器的高4位， 并将"除数"寄存器的低4位补上0，

最后是将4位的"商"寄存器置为0。这就完成了初始化的工作。

现在正式进入工作的流程。第1步，

是执行减法运算，将当前余数寄存器的内容，减去除数寄存器的内容。

我们注意到，除数寄存器当中，现在保存的是00100000，

而余数寄存器当中，保存的是00000111。

那么控制逻辑会向ALU发出执行减法的控制信号， ALU将输入的两个数进行减法运算，

得到结果是11100111。

这个时候，控制逻辑还会向余数寄存器发出要写入的控制信号，

在下一个时钟上升沿到来的时候，ALU的输出就会保存到余数寄存器当中。

现在余数寄存器已经更新了，然后执行第2步， 就是检查余数寄存器，

如果大于等于零，就执行2a这个分支的操作； 如果小于零，则执行2b这个分支的操作；

那怎么检查是大于等于零，还是小于零呢？其实只要看余数寄存器的最高位就可以了。

根据补码的表示，我们知道，如果最高位为1，则代表这个数小于零。

那现在我们就要执行2b这个分支。

因为余数寄存器的内容小于零，意味着我们不应该做刚才那次减法。

所以首先需要回退第一步的操作，但是硬件的操作已经完成，

刚才的减法结果也保存到了余数寄存器当中，而余数寄存器之前的值，我们现在已经不知道了。

怎么回退呢？

所以实际上没有真正的往回退的方法， 只是还好，我们知道如何找回跟刚才一样的那个数。

因为刚才执行的是一次减法，所以现在只要把余数寄存器的内容加上

除数寄存器的内容，就可以得到刚才在余数寄存器当中的内容了。

所以现在，在ALU的输入端，两个输入信号早已准备好，

而控制逻辑会给出执行加法的控制信号， 很快ALU会完成加法运算，并得到运算结果。

当然，控制逻辑也会给出，让余数寄存器写入这个结果的控制信号，然后

等到下一个时钟上升沿到来的时候，余数寄存器就把这个值保存进去了。

我们发现，这个值就是我们刚才执行第1步之前的余数寄存器的内容， 也就是十进制的7。

现在我们已经完成了回退的工作，那我们还要记录这一步 所对应的商，也就是将商寄存器要左移一位，并

将新的最右位设为0，我们注意右边的这个商寄存器。

那好，现在这一步的工作就已经完成了。

第3步， 是将除数寄存器的内容右移一位，为下一次的减法操作做好准备。

我们注意除数寄存器，除数寄存器右移之后，最左边会补入一个0，

而最右边移出的数就直接丢掉了。我们再来看第4步，

第4步，是检查这是不是最后一轮循环，因为这是一个4位的除法器， 所以我们要检查的是，当前是不是第5轮循环，

那经过检查，现在不是第5轮循环，所以我们还需要继续除法器的工作。

现在我们进入第2轮，第2轮的第1步和第1轮的第1步实际上是一样的，

也是执行减法操作，余数寄存器减去除数寄存器。

在控制逻辑的控制下，ALU很快就可以得到减法的结果。

我们现在就可以注意一下，这个减法运算的结果最高位是1， 所以等一会儿我们还得回退这个操作。

后面要做的事情，就和我们刚才第一轮所解释过的一样，我们就不再一步一步详细描述了。

那我们用一个表，来总体说明从第二轮到第四轮发生的事情。

刚才我们看到的

是第二轮当中的第1步，将余数减除数，得到的结果会保存到余数寄存器当中。

这个表中，红颜色的部分，代表在这一步当中发生改变的数值。

在这里我们会注意到，余数寄存器当中的最高位为1，说明这是一个小于0的数。

所以在下一步，会进行回退的操作， 也就是将余数加上除数，再放回到余数寄存器当中，

同时将商左移，最右边补0， 然后将除数寄存器右移，这就完成了第二轮的操作。

然后是第三轮，我们同时也注意到第三轮进行的减法，得到的结果最高位还是1，

所以还要执行回退，然后将商的最右边移入一个0，

再将除数进行右移。然后是第四轮， 第四轮执行完了减法，我们注意到最高位是0，

那这就表明这次减法运算不用回退，

所以在下一轮余数寄存器并没有发生变化，而商左移之后，最右边补入了一个1，

然后再将除数进行右移，这样就完成了四轮的工作。

现在我们回到这个结构图，来看第五轮的第1步。这时候继续执行减法操作，

除数寄存器的输出，余数寄存器的输出，控制逻辑发出减法的控制信号，

ALU得出运算的结果，运算的结果是1， 在下一个时钟沿到来的时候，它会被存到余数寄存器当中。

我们注意，余数寄存器现在发生了改变。

然后是第五轮的第2步，检查余数寄存器的最高位，

发现最高位为0，说明这是一个大于等于0的数，进入2a这个分支进行操作。

2a这个分支，就是将商寄存器左移一位，并将新的最右位设为1。

我们注意商寄存器，左移后，最右边设了1。

然后是第五轮的第3步，将除数寄存器右移1位。

我们从一个旁观者的角度可以看到，因为这已经是最后一轮了，所以

除数寄存器的这一次右移，是没有必要的，但是作为

硬件来说，它现在并不知道这是最后一轮，所以它仍然在为下一轮进行准备工作，

直到在第4步，在进行检查是否为第五轮的时候，硬件的控制

逻辑才会知道，这已经是最后一轮循环，这个除法运算已经完成了。

现在除法运算的结果就已经在余数寄存器和商寄存器当中。

注意我们要做的是这个例子，7除以2，商为 3，也就是现在的0011，而余数为1，

之后我们用纸笔运算的结果是一样的。

那这就是一个四位的除法器的结构和工作的流程。

与之类似，我们就可以得到一个32位的除法器， 那这个32位的除法器当中，有一个

64位的余数寄存器，一个64位的除数寄存器，带右移功能，

一个32位的商寄存器，带左移的功能， 一个64位的ALU，支持加法和减法运算。

那这样一个除法器，用刚才同样的流程就可以完成32位除法的运算。

现在这么复杂的除法，都已经被我们解决了，

> 我们已经有了一个可以正常工作的除法器，这可真是一个了不起的成果！
>
> 当然，这当中还有一些细节的问题我们没有谈到。
>
> 比如说，当一个正数和一个负数相除的时候 那商应该是正数还是负数？余数又应该是正数还是负数呢？
>
> 类似这样的小问题，就留给大家回去自己思考了。

407-除法器的优化

> 我们现在的这个除法器已经可以正常的工作了。
>
> 但是距离实用，还有相当大的距离，
>
> 必须要经过优化。不过，除法的优化就比较复杂，
>
> 因此，在这一节，我们只是对它的优化方法和优化的方向做一个非常基本的探讨。

> 这是我们已经有了的这一版除法器， 我们不妨称之为第一版的实现。
>
> 在这个除法器当中，有一个64位的余数寄存器，一个64位的
>
> 除数寄存器，一个32位的商寄存器，和一个64位的ALU。
>
> 那我们首先来考虑面积方面的优化，我们先来看看在哪些地方存在着浪费。
>
> 首先，我们的除数是只有32位的， 而我们的除数寄存器使用了64位的，
>
> 实际只使用了其中的一半，这是第一个可以优化的点。
>
> 第二，商寄存器在初始化的时候是空的， 每执行完一轮会产生移位，从右向左逐位填满，
>
> 所以这里也存在着浪费的情况。
>
> 第三，余数寄存器初始时是满的，也就是最开始的被除数。
>
> 但是在不断进行和除数的减法之后，这个余数会变得越来越小， 它有实际意义的位从左往右在逐渐地减少。
>
> 所以越往后，余数寄存器当中浪费的位就会越多， 那我们就尝试从这三个方面进行面积上的优化。
>
> 那我们把经过面积优化的除法器称为第二版，我们就来看一看第二版的除法器是什么样的。
>
> 为了方便对比，我们把优化方案和原来结构的描述放在一起。
>
> 首先，原先有一个64位的除数寄存器，现在我们将除数寄存器缩小为32位，
>
> 因为除数本来就是32位的，所以这样，除数寄存器就没有了浪费的情况。
>
> 但是我们要注意，除数寄存器是在整个运算过程中一直要使用的，
>
> 它的移位只是为了和余数寄存器进行对齐，以方便运算。所以
>
> 原先才准备了一个64位的寄存器，以便于除数在移位的过程中也不会丢失。
>
> 那现在将它缩小为32位了，就不能再有移位的功能，否则其中
>
> 有效位就会丢失了，因此这个除数寄存器也就不用再支持移位。
>
> 那么在运算中，需要将除数和余数进行对齐的这项功能，就得放到别的地方来完成了。
>
> 第二，我们知道原来有一个32位的商寄存器， 它一开始是空的，在运算的过程中逐位填满，

那既然它一开始没有用，所以我们干脆先取消这个寄存器， 再看一看可不可以在别的地方实现类似的功能。

第三，原来有一个64位的ALU，

而我们知道，实际参与运算的其中一个操作数是除数， 另一个操作数，则是余数寄存器当中和除数对应的一些位，

实际上只有32位，所以我们就将这个64位的ALU缩小为32位， 也就是说，只要让其中有效位数参与运算。

那么对于余数寄存器，或者说一开始是被除数的这个寄存器，

它在和除数进行减法运算时，最开始只有最高的32位参与运算，

之后才逐次地往低位移动，所以我们先规定， 这个余数寄存器只有高32位参与这个加减法运算，

我们也在这个余数寄存器当中，画一条半透明的线作为标记，而整个余数寄存器仍然保留为- 64位，

它其中的高32位被连接到ALU的一个输入， 而ALU的输出也连接到了余数寄存器的高32位，

但是原来除数寄存器是带有右移的功能，从而实现了 余数寄存器中参与运算的数逐渐向低位移动这样一个情况。

那现在除数寄存器已经不能右移了，与之相对，余数寄存器那就得支持一个左移的功能，

而且我们再去回顾除法的运算过程，可以发现，余数，或者说被除数的高位一旦退出了

运算，就不再会有机会重新参与运算了。所以把余数寄存器的高位向左移位，并将

移出的位丢弃，是不会对运算的过程造成影响的，所以我们将余数寄存器加上左移的功能。

而实际上我们发现，现在这个余数寄存器不仅支持左移，还支持右移的功能。

为什么支持左移？刚才我已经介绍了；而为什么支持右移呢？ 就留给大家自己来思考。

当然，什么时候进行移位，也都需要由控制逻辑进行控制， 所以在余数寄存器上也需要有相应的控制输入。

那现在，我们要注意的是， 余数寄存器的浪费问题仍然没有解决。

随着运算的进行，每一轮余数寄存器都会 向左移位一次，它的右边则会多空出一位来，

而且空出的位会越来越多。那么我们回头来看一看，其中我们还需要一个32位

的商寄存器，而且这个商在运算的一开始是不需要占据任何空间的，

只需要每一轮采用左移的方式，给它多分配一个比特就可以了。

那就正好是余数寄存器现在所浪费的情况。

那我们就可以很自然地将商寄存器合并到余数寄存器当中，

让每一轮产生的商，从余数寄存器的右端，逐个移入。

这样当运算 结束时，商就占据了余数寄存器的低32位，

而余数寄存器的高32位，则是最后真正的余数。这样

再连上对应的控制逻辑之后，我们就有了一个经过面积优化的除法器。

那么在实现了面积优化之后，那我们就要考虑，在性能上是否可以进行优化。

想进行除法器的性能优化，我们就要先来回顾乘法是如何进行优化的。

其实现在的乘法器可以做到非常好的优化，

这和乘法运算自身的特点是分不开的。我们来看之前提到过的例子。

其实仔细分析就可以发现，虽然乘法和除法都要产生很多中间的结果，

也都需要通过移位等操作进行对齐，再进行最后的运算。

但是很大的不同在于，乘法的每一个中间结果，都是独立的，每一个中间结果，

要么和被乘数相等，要么是0，而且它究竟是哪一种，

只有乘数当中固定的一位决定，不受其他位的影响。所以如果我们人工进行

乘法的计算，当我们有了被乘数和乘数之后，可以交给很多人来协作运算，

每个人只计算其中一个中间结果，然后再由一些人将这些中间结果加起来，

这样就可以通过并行的计算，大幅度地提高性能。

而我们来看除法的这个工作流程，

中间有一个检查余数的工作，而且当余数小于0时，还需要回退第一步的操作。

这实际上就是因为除法的这些中间结果，并不是各个独立的。那 我们可能会想到，乘法的那个流程图当中，不也有一个要进行检查

的分支操作吗，看上去和除法这个流程图是非常相似的， 那么不妨把乘法的这个流程图找出来，仔细地看一看。

的确，在开始的这个地方，确实有一个检查乘数寄存器的最低位的操作，

并且根据检查结果的不同，会走向两个不同的分支。但是 我们仔细分析就会发现，这个分支实际上是不存在的。

因为当最低位为1时，会执行将被乘数寄存器和乘积寄存器相加，这样一个操作。

那么我们不妨可以理解为，将被乘数寄存器的内容和乘数寄存器的

最低位进行一个”与操作“，然后再和乘积寄存器相加。

因为现在乘数寄存器的最低位是1， 任何数和1进行”与操作“，结果还是这个数本身，

所以这样的改动并不会影响这里的操作。而对于这一边， 当最低位为0时，如果我们也做同样的操作，也就是将

乘数寄存器的最低位和被乘数寄存器先进行”与操作“，然后 再和乘积寄存器相加。

那因为任何数和0进行”与操作“，结果都是0，0再和乘积寄存器相加，

那就相当于没有执行加法操作，乘积寄存器的内容并没有改变。

所以实际上，我们可以将这个分支取消，

在每轮的最开始，都直接将乘数寄存器的最低位和被 乘数寄存器进行”与操作”，然后和乘积寄存器相加，

这样无论最低位是1，还是0，都能完成这个工作流程图所表达的工作。

那即使对于这个优化后的工作流程，效果也是一样的。

因为第一步的检查之后，2和3，这两步操作都会执行， 唯一的区别在于1a这个操作。

所以采用刚才所描述的那种修改，也可以达到同样的效果。

因此，对于这个工作流程来说，在实际 的实现当中，也可以不存在这个分支的操作。

> 然而对于除法的这个工作流程， 这一次减法的运算结果是事先无法预知的，
> 因此无法预知下一步将执行哪一个分支，而且其中一种分支
> 还需要将这个减法操作进行回退，所以也没有办法将两个分支进行合并。
> 因此，在这个除法运算流程的大框架之下， 是很难进行进一步的性能优化了。

> 正是由于除法的优化非常的困难，
> 因而也引来了很多人对除法进行深入的研究， 也产生了很多很有趣的、快速除法的实现方法，
> 如果你对此有兴趣，可以进一步深入地探索。