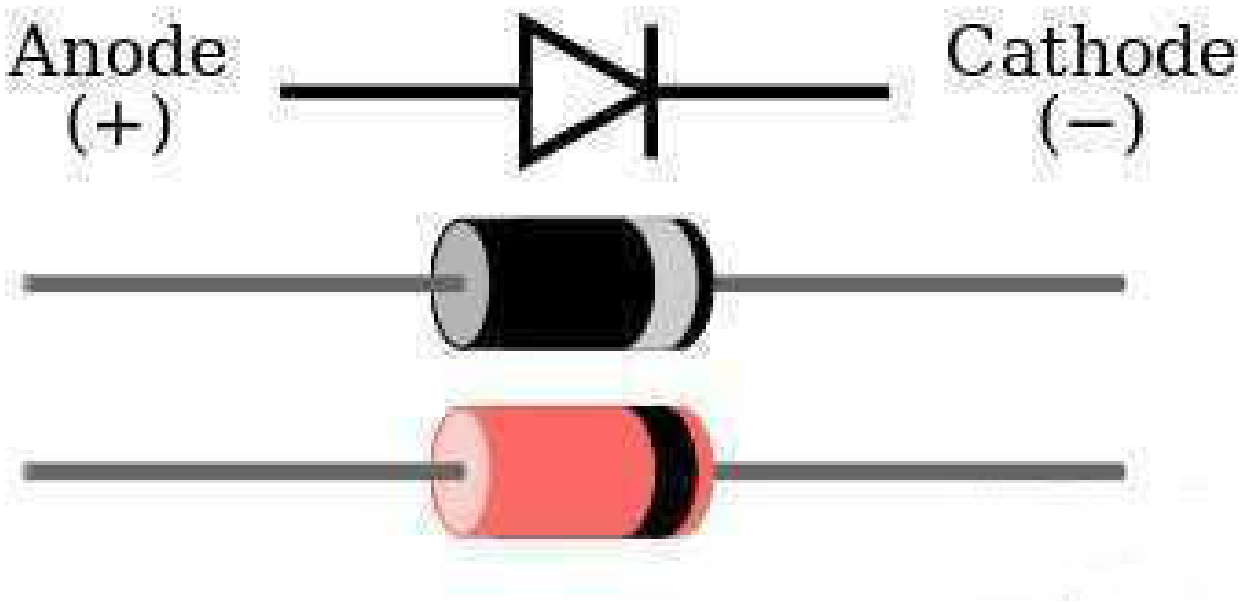
知乎神回复: 计算机的 CPU 是怎么认识代码的? 文章告诉你答案!

叶叶旅游记 12-15

又是读个大学就能懂系列的。

行吧，老规矩，尽量简单的语言来解释一下。

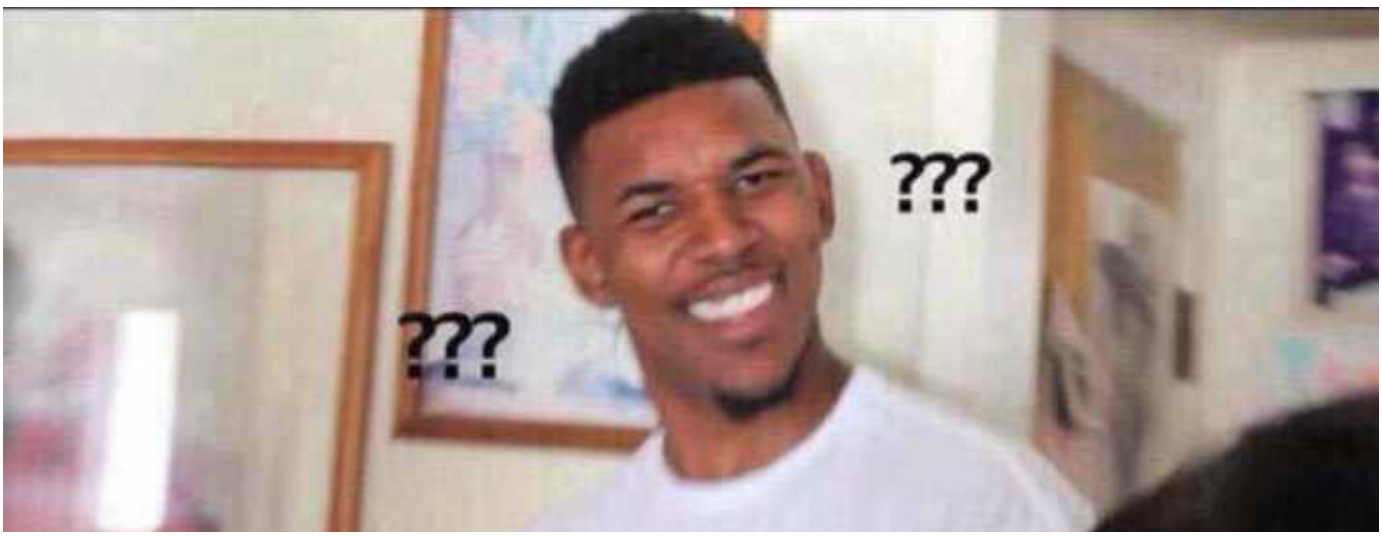
先说一下半导体，啥叫半导体？就是介于导体和绝缘体中间的一种东西，比如二极管。



电流可以从A端流向C端，但反过来则不行。你可以把它理解成一种防止电流逆流的东西。

当C端10V，A端0V，二极管可以视为断开。

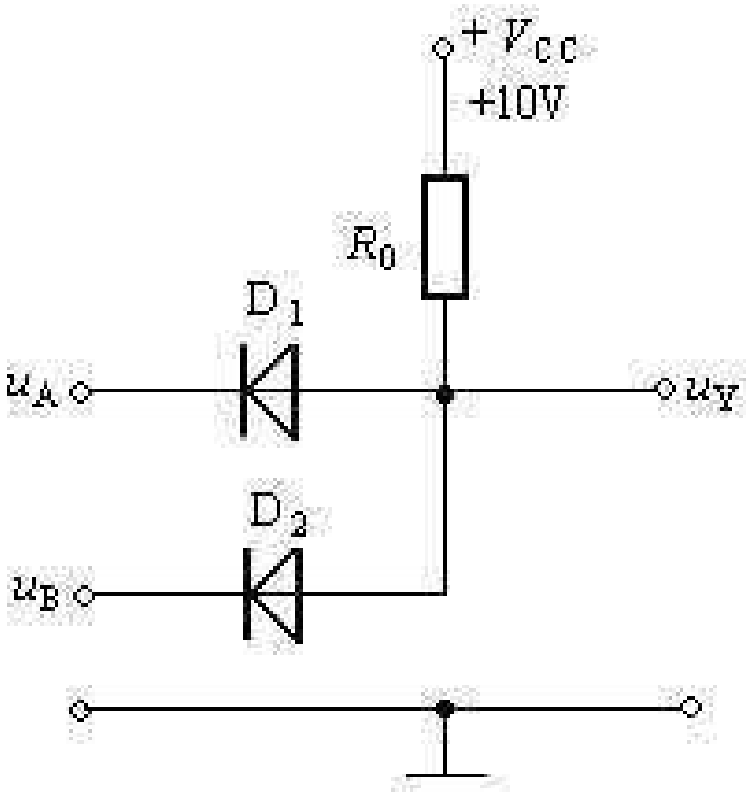
当C端0V，A端10V，二极管可以视为导线，结果就是A端的电流源源不断的流向C端，导致最后的结果就是A端=C端=10V

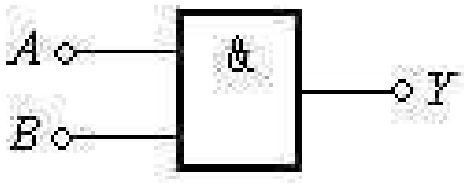


等等，不是说好的C端0V，A端10V么？咋就变成结果是A端=C端=10V了？你可以把这个理解成初始状态，当最后稳定下来之后就会变成A端=C端=10V。

文科的童鞋们对不住了，实在不懂问高中物理老师吧。反正你不能理解的话就记住这种情况下它相当于导线就行了。

利用半导体，我们可以制作一些有趣的电路，比如【与门】





此时A端B端只要有一个是0V，那Y端就会和0V地方直接导通，导致Y端也变成0V。只有AB两端都是10V，Y和AB之间才没有电流流动，Y端也才是10V。

我们把这个装置成为【与门】，把有电压的地方计为1，0电压的地方计为0。至于具体几V电压，那不重要。

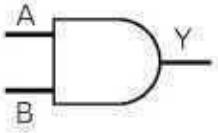
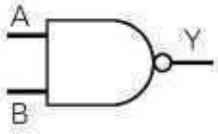
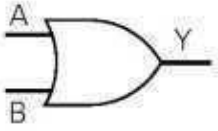
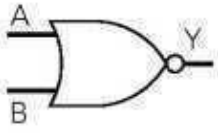
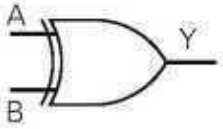
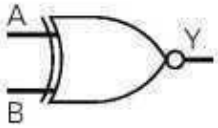
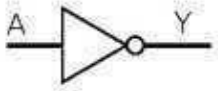
也就是AB必须同时输入1，输出端Y才是1;AB有一个是0，输出端Y就是0。

其他还有【或门】【非门】和【异或门】，跟这个都差不多，或门就是输入有一个是1输出就是1，输入00则输入0。

非门也好理解，就是输入1输出0，输入0输出1。

异或门难理解一些，不过也就那么回事，输入01或者10则输出1，输入00或者11则输出0。（即输入两个一样的值则输出0，输入两个不一样的值则输出1）。

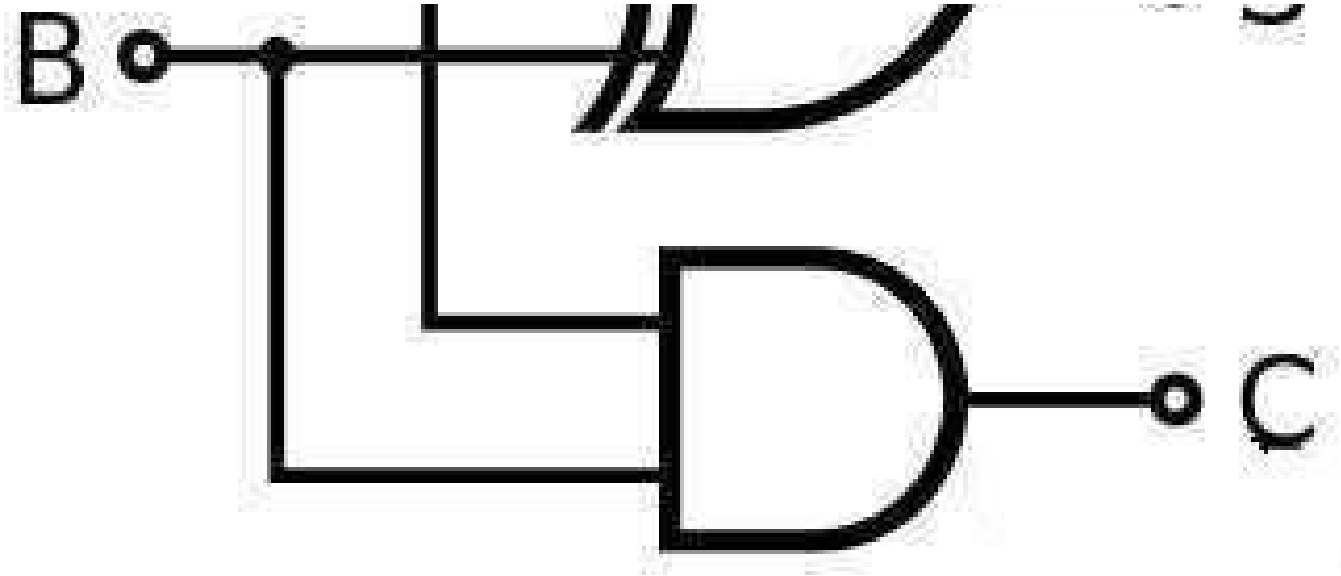
这几种门都可以用二极管做出来，具体怎么做就不演示了，有兴趣的童鞋可以自己试试。每次都画二极管也是个麻烦，我们就把门电路简化成下面几个符号。

逻辑	真值表	MIL 逻辑符号	逻辑	真值表	MIL 逻辑符号
与	A B Y 0 0 0 0 1 0 1 0 0 1 1 1		与非	A B Y 0 0 1 0 1 1 1 0 1 1 1 0	
或	A B Y 0 0 0 0 1 1 1 0 1 1 1 1		或非	A B Y 0 0 1 0 1 0 1 0 0 1 1 0	
异或	A B Y 0 0 0 0 1 1 1 0 1 1 1 0		异或非	A B Y 0 0 1 0 1 0 1 0 0 1 1 1	
非	A Y 0 1 1 0				

想要学习C/C++编程的可以关注私信小编“编程”二字交流

然后我们就可以用门电路来做CPU了。当然做CPU还是挺难的，我们先从简单的开始：加法器。

加法器顾名思义，就是一种用来算加法的电路，最简单的就是下面这种。

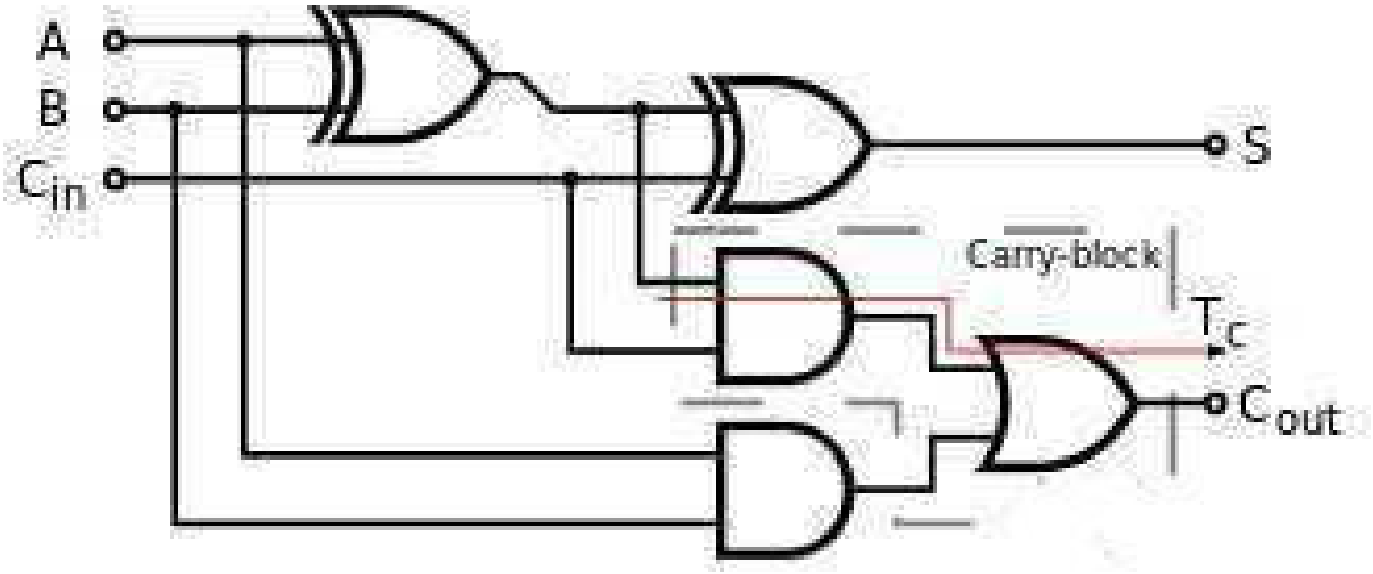


AB只能输入0或者1，也就是这个加法器能算0+0，1+0或者1+1。

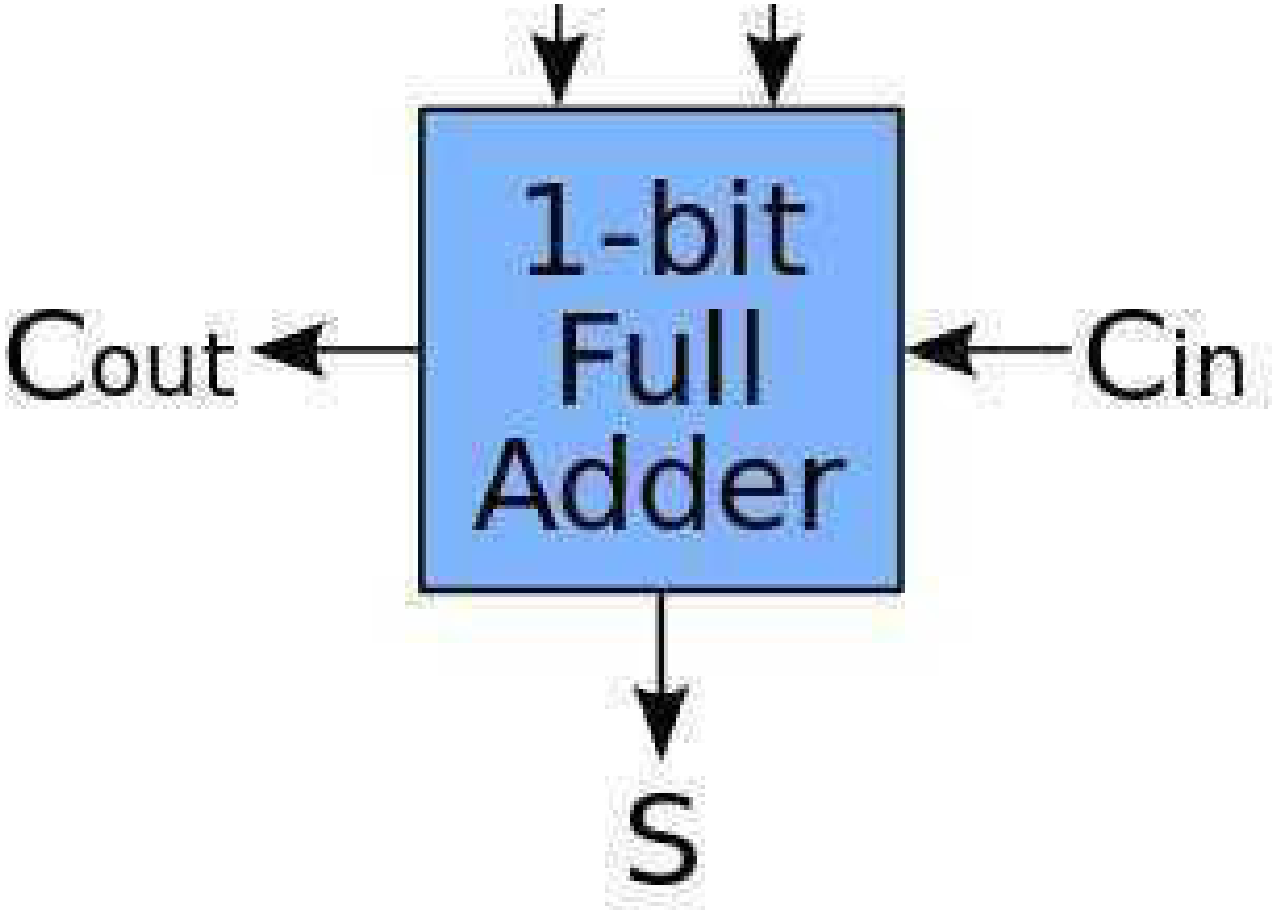
输出端S是结果，而C则代表是不是发生进位了，二进制1+1=10嘛。这个时候C=1，S=0

费了大半天的力气，算个1+1是不是特别有成就感？

那再进一步算个1+2吧（二进制01+10），然后我们就发现了一个新的问题：第二位需要处理第一位有可能进位的问题，所以我们还得设计一个全加法器。

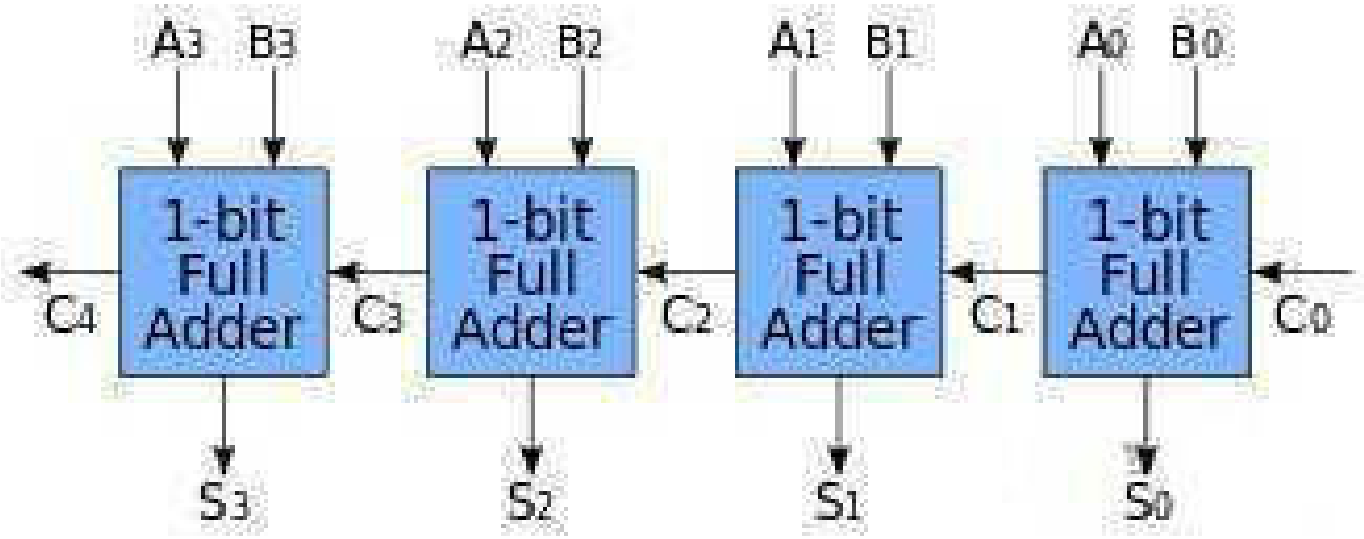


每次都这么画实在太麻烦了，我们简化一下



也就是有3个输入2个输出，分别输入要相加的两个数和上一位的进位，然后输入结果和是否进位。

然后我们把这个全加法器串起来



我们就有了一个4位加法器，可以计算4位数的加法也就是15+15，已经达到了幼儿园中班水平，是不是特别给力？

做完加法器我们再做个乘法器吧，当然乘任意10进制数是有点麻烦的，我们先做个乘2的吧。

乘2就很简单了，对于一个2进制数数我们在后面加个0就算是乘2了

比如

5=101（2）

10=1010（2）

所以我们只要把输入都往前移动一位，再在最低位上补个零就算是乘2了。具体逻辑电路图我就不画，你们知道咋回事就行了。

那乘3呢？简单，先位移一次（乘2）再加一次。乘5呢？先位移两次（乘4）再加一次。

所以一般简单的CPU是没有乘法的，而乘法则是通过位移和加算的组合来通过软件来实现的。这说的有点远了，我们还是继续做CPU吧。

现在假设你有8位加法器了，也有一个位移1位的模块了。串起来你就能算

（A+B）X2

了！激动人心，已经差不多到了准小学生水平。

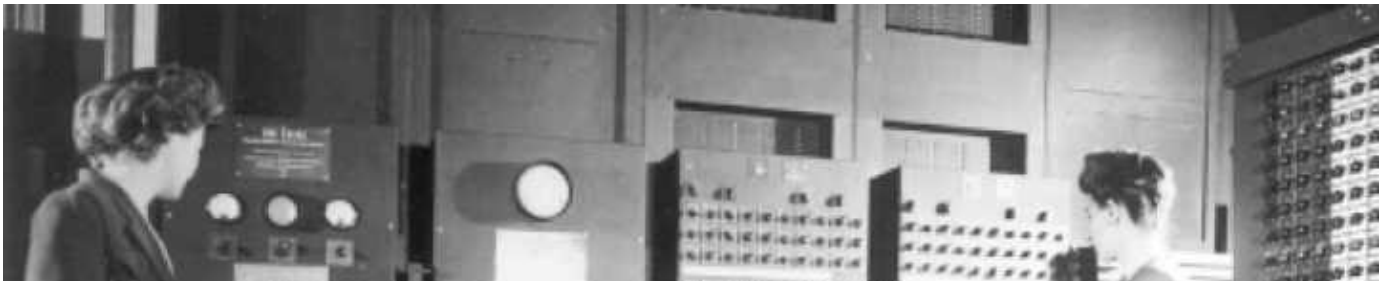
那我要是想算

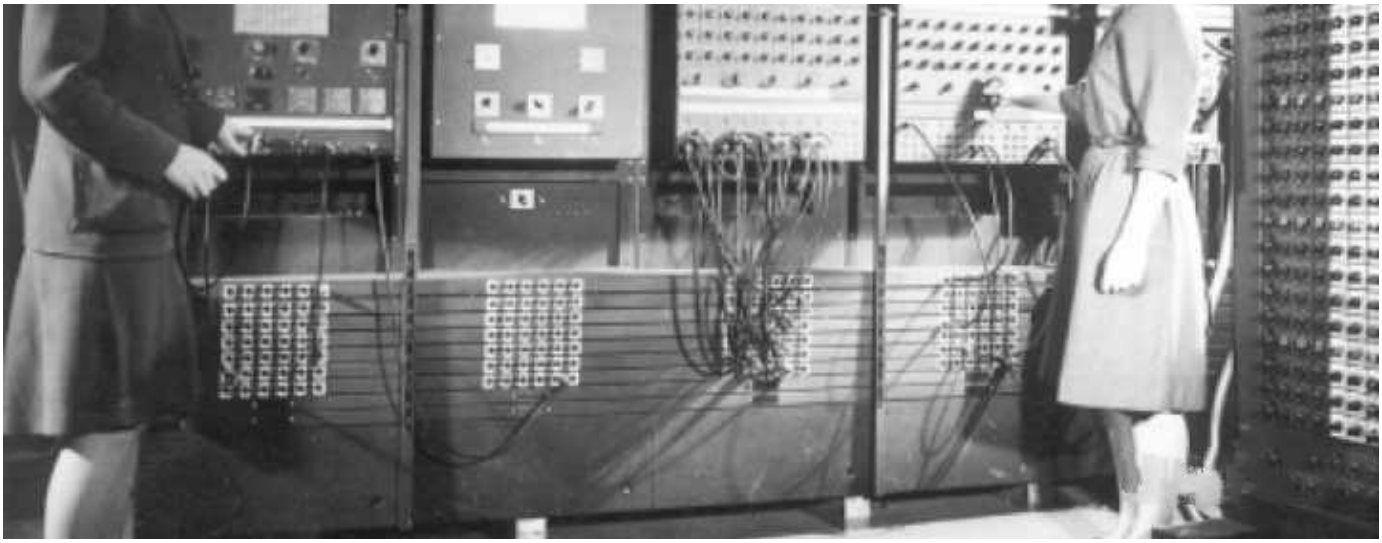
AX2+B

呢？简单，你把加法器模块和位移模块的接线改一下就行了，改成输入A先过位移模块，再进加法器就可以了。

啥？？？？你说啥？？？你的意思是我改个程序还得重新接线？

所以你以为呢？编程就是把线来回插啊。



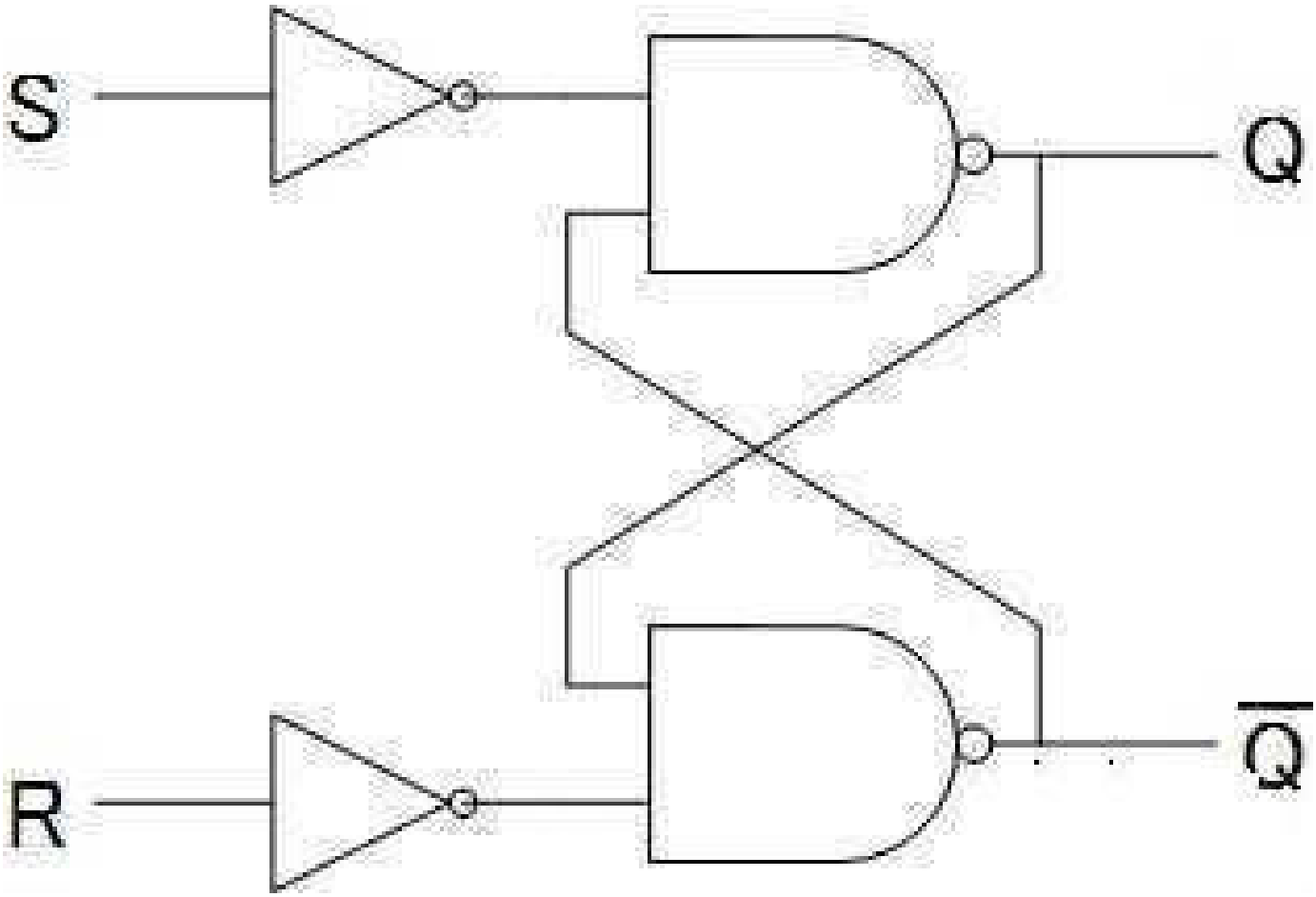


惊喜不惊喜？意外不意外？

早期的计算机就是这样编程的，几分钟就算完了但插线好几天。而且插线是个细致且需要耐心的工作，所以那个时候的程序员都是清一色的漂亮女孩子，穿制服的那种，就像照片上这样。是不是有种生不逢时的感觉？

虽然和美女作伴是个快乐的事，但插线也是个累死人的工作。所以我们需要改进一下，让CPU可以根据指令来相加或者乘2。

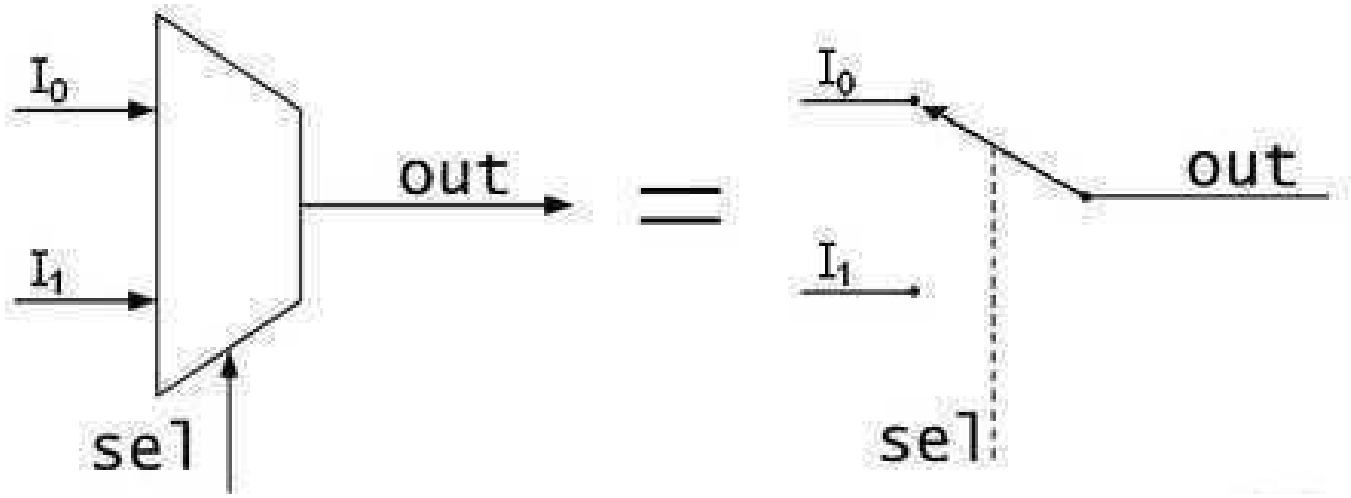
这里再引入两个模块，一个叫flip-flop，简称FF，中文好像叫触发器。



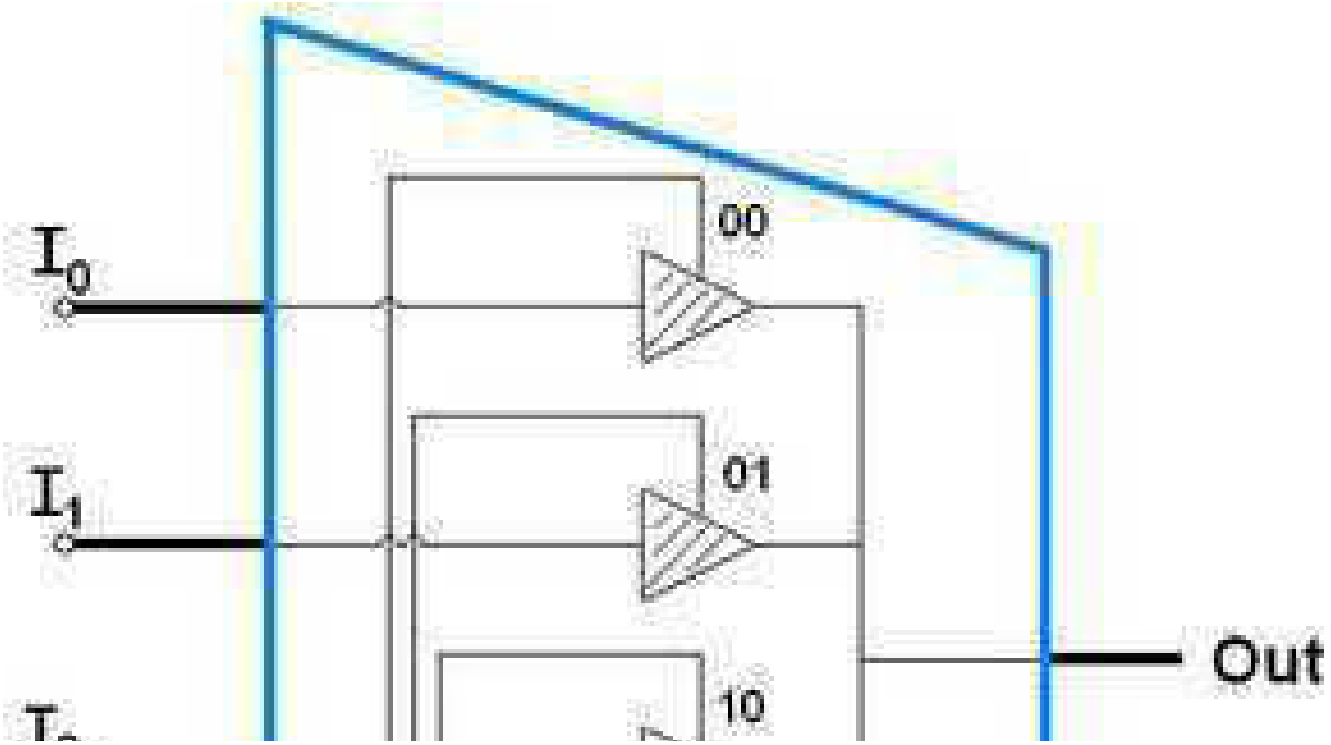
这个模块的作用是存储1bit数据。比如上面这个RS型的FF，R是Reset，输入1则清零。S是Set，输入1则保存1。RS都输入0的时候，会一直输出刚才保存的内容。

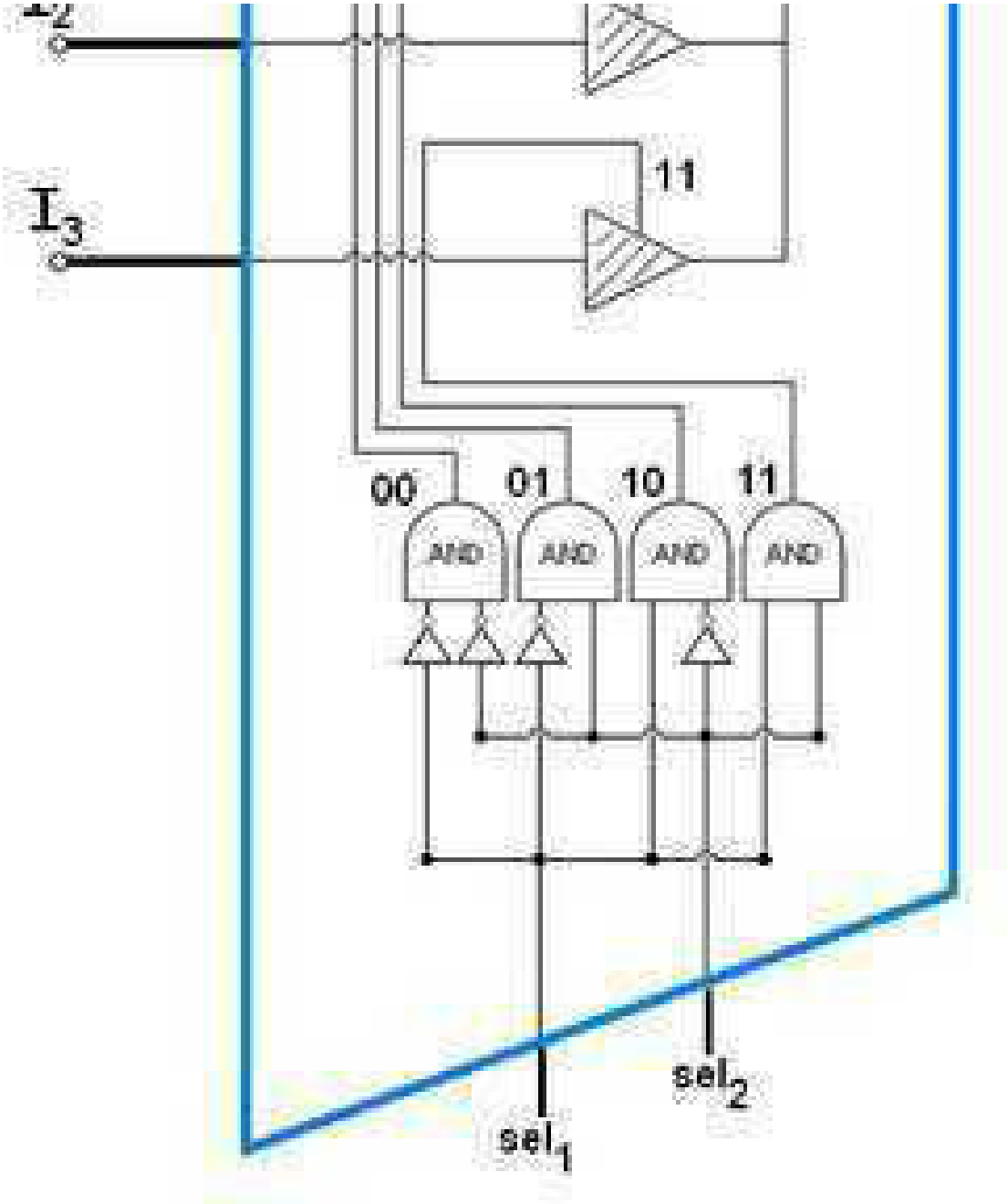
我们用FF来保存计算的中间数据（也可以是中间状态或者别的什么），1bit肯定是不够的，不过我们可以并联嘛，用4个或者8个来保存4位或者8位数据。这种我们称之为寄存器（Register）。

另外一个叫MUX，中文叫选择器。



这个就简单了，sel输入0则输出i0的数据，i0是什么就输出什么，01皆可。同理sel如果输入1则输出i1的数据。当然选择器可以做的很长，比如这种四进一出的





具体原理不细说了，其实看看逻辑图琢磨一下就懂了，知道有这个东西就行了。

有这个东西我们就可以给加法器和乘2模块（位移）设计一个激活针脚。

这个激活针脚输入1则激活这个模块，输入0则不激活。这样我们就可以控制数据是流入加法器还是位移模块了。

于是我们给CPU先设计8个输入针脚，4位指令，4位数据。

我们再设计3个指令：

0100，数据读入寄存器

0001，数据与寄存器相加，结果保存到寄存器

0010，寄存器数据向左位移一位（乘2）

为什么这么设计呢，刚才也说了，我们可以为每个模块设计一个激活针脚。然后我们可以分别用指令输入的第二第三第四个针脚连接寄存器，加法器和位移器的激活针脚。

这样我们输入0100这个指令的时候，寄存器输入被激活，其他模块都是0没有激活，数据就存入寄存器了。同理，如果我们输入0001这个指令，则加法器开始工作，我们就可以执行相加这个操作了。

这里就可以简单回答这个问题的第一个小问题了：

那cpu 是为什么能看懂这些二级制的数呢？

为什么CPU能看懂，因为CPU里面的线就是这么接的呗。你输入一个二进制数，就像开关一样激活CPU里面若干个指定的模块以及改变这些模块的连同方式，最终得出结果。

几个可能会被问道的问题

Q：CPU里面可能有成千上万个小模块，一个32位/64位的指令能控制那么多吗？

A：我们举例子的CPU里面只有3个模块，就直接接了。真正的CPU里会有一个解码器（decoder），把指令翻译成需要的形式。

Q：你举例子的简单CPU，如果我输入指令0011会怎么样？

A：当然是同时激活了加法器和位移器从而产生不可预料的后果，简单的说因为你使用了没有设计的指令，所以后果自负呗。（在真正的CPU上这么干大概率就是崩溃呗，当然肯定会有各种保护性的设计，死也就死当前进程）

细心的小伙伴可能发现一个问题：你设计的指令

【0001，数据与寄存器相加，结果保存到寄存器】

这个一步做不出来吧？毕竟还有一个回写的过程，实际上确实是这样。我们设计的简易CPU执行一个指令差不多得三步，读取指令，执行指令，写寄存器。

经典的RISC设计则是分5步：读取指令(IF)，解码指令(ID)，执行指令(EX)，内存操作(MEM)，写寄存器(WB)。我们平常用的x86的CPU有的指令可能要分将近20个步骤。

你可以理解有这么一个开关，我们啪的按一下，CPU就走一步，你按的越快CPU就走的越快。咦？听说你有个想法？少年，你这个想法很危险啊，姑且不说你有没有麒麟臂，能不能按那么快（现代的CPU也就2GHz多，大概也就一秒按个20亿下左右吧）

就算你能按那么快，虽然速度是上去了，但功耗会大大增加，发热上升稳定性下降。江湖上确实有这种玩法，名曰超频，不过新手不推荐你尝试哈。

那CPU怎么知道自己走到哪一步了呢？前面不是介绍了FF么，这个不光可以用来存中间数据，也可以用来存中间状态，也就是走到哪了。

具体的设计涉及到FSM（finite-state machine），也就是有限状态机理论，以及怎么用FF实装。这个也是很重要的一块，考试必考哈，只不过跟题目关系不大，这里就不展开讲了。

我们再继续刚才的讲，现在我们有3个指令了。我们来试试算个（1+4）X2+3吧。

0100 0001 ；寄存器存入1

0001 0100 ；寄存器的数字加4

0010 0000 ；乘2

0001 0011 ；再加三

太棒了，靠这台计算机我们应该可以打败所有的幼儿园小朋友，称霸大班了。而且现在我们用的是4位的，如果换成8位的CPU完全可以吊打低年级小学生了！

实际上用程序控制CPU是个挺高级的想法，再此之前计算机（器）的CPU都是单独设计的。

1969年一家日本公司BUSICOM想搞程控的计算器，而负责设计CPU的美国公司也觉得每次都重新设计CPU是个挺傻X的事，于是双方一拍即合，于1970年推出一种划时代的产品，世界上第一款微处理器4004。

这个架构改变了世界，那家负责设计CPU的美国公司也一步一步成为了业界巨头。哦对了，它叫Intel，对，就是噔噔噔噔的那个。

我们把刚才的程序整理一下，

你来把它输入CPU，我去准备一下去幼儿园大班踢馆的工作。

神马？等我们输完了人家小朋友掰手指都能算出来了？？

没办法机器语言就是这么反人类。哦，忘记说了，这种只有01组成的语言被称之为机器语言（机器码），是CPU唯一可以理解的语言。不过你把机器语言让人读，绝对一秒变典韦，这谁也受不了。

想要学习C/C++编程的可以关注私信小编“编程”二字交流

所以我们还是改进一下吧。不过话虽这么讲，也就往前个30年，直接输入01也是个挺普遍的事情。

于是我们把我们机器语言写成的程序

0100 0001 ；寄存器存入1

0001 0100 ；寄存器的数字加4

0010 0000 ；乘2

0001 0011 ；再加三

改写成

MOV 1 ；寄存器存入1

ADD 4 ；寄存器的数字加4

SHL 0 ；乘2

ADD 3 ；再加三

是不是容易读多了？这就叫汇编语言。