# 仲裁器

## 前言

对于总线协议中，由于具有多个主机要对总线进行请求，针对这一情况总会提到仲裁器用于判决当前时刻下那个主机可以获得总线的控制权。而仲裁器的处理延迟和处理逻辑算法是仲裁器的重要部分，设计一款处理速度更快的仲裁器至关重要。

## Round robin arbiter 轮询仲裁器

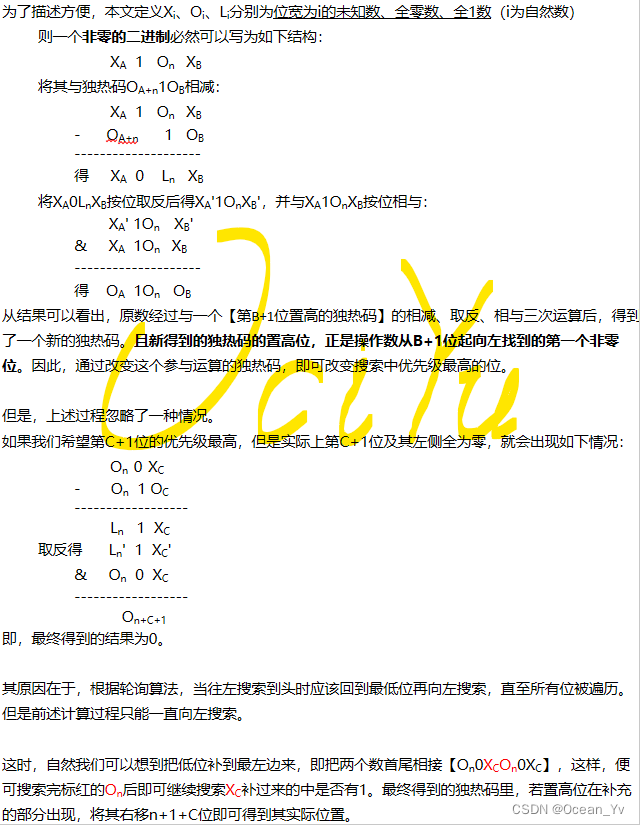
轮询仲裁器，顾名思义其优先级是变化的，它的优先级不仅与当前状态有关还与上一个状态有关。所以轮询仲裁器是时序电路，而不是纯组合电路就可以完成的。

假设总线上共有4个主机，对轮询仲裁器进行着请求，其开始时的优先级时高位大于低位。a > b > c > d 。

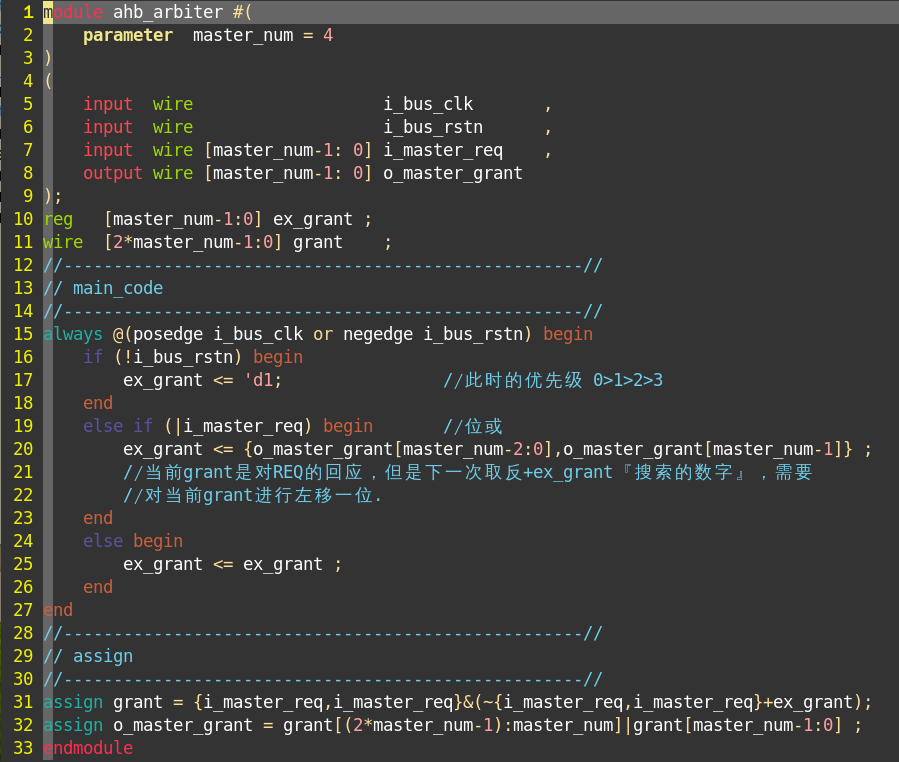
|  |  |  |  |
| --- | --- | --- | --- |
| 输入请求 | 此时的优先级 | 仲裁器判决结果 | 下一刻的优先级 |
| 0000 | 0>1>2>3 | 0000 | 0>1>2>3 |
| 0101 | 0>1>2>3 | 0001 | 1>2>3>0 |
| 1000 | 1>2>3>0 | 1000 | 0>1>2>3 |
| 1010 | 0>1>2>3 | 0010 | 2>3>0>1 |
| 1111 | 2>3>0>1 | 0100 | 3>0>1>2 |
| 1111 | 3>0>1>2 | 1000 | 0>1>2>3 |

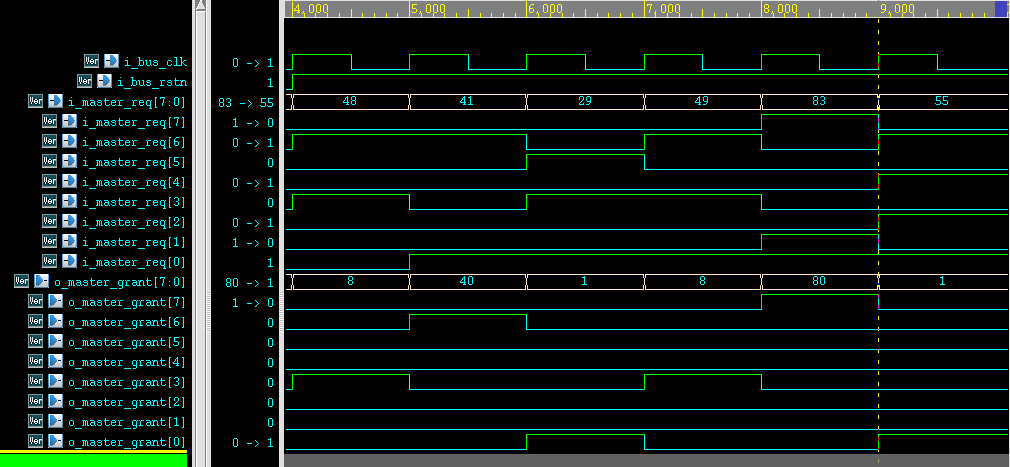
可以看出，此刻的优先级至于上次的判决结果有关。

此流程是轮询算法的精髓，而这里的先减1Ob是表示从哪一位开始获得第一个非零位。这与我们的优先级不谋而合，我们上次获得的Grant=4‘b0010，那么下次他的优先级就是2301，此时1的优先级应该为最低，如果直接采用Grant=4‘b0010作为这个1Ob是不合理的，所以对Grant进行左移一位，即表示对原优先级中下一级“2”作为最高的优先级。



注：-1 取反 与 取反 +1 得到的结果是相同的。





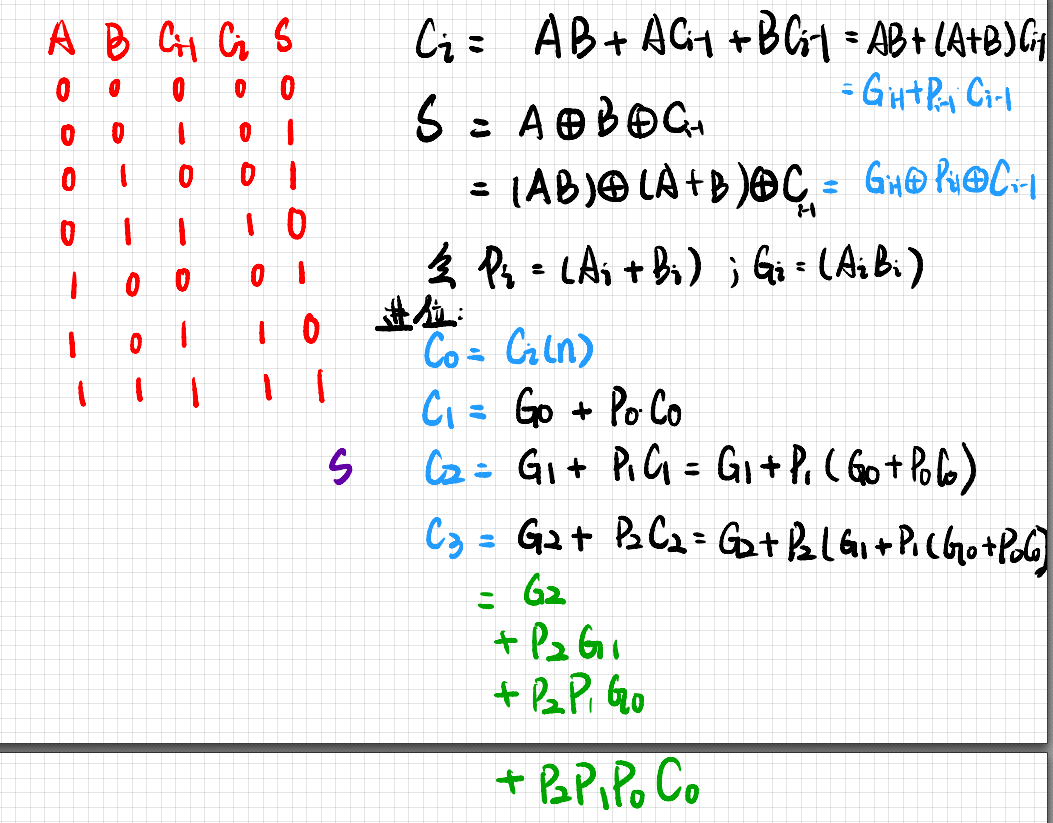
## Parallel prefix round robin arbiter 平行前缀轮询仲裁器

参考文章 1 . Fast Arbiters for On-Chip Network Switches

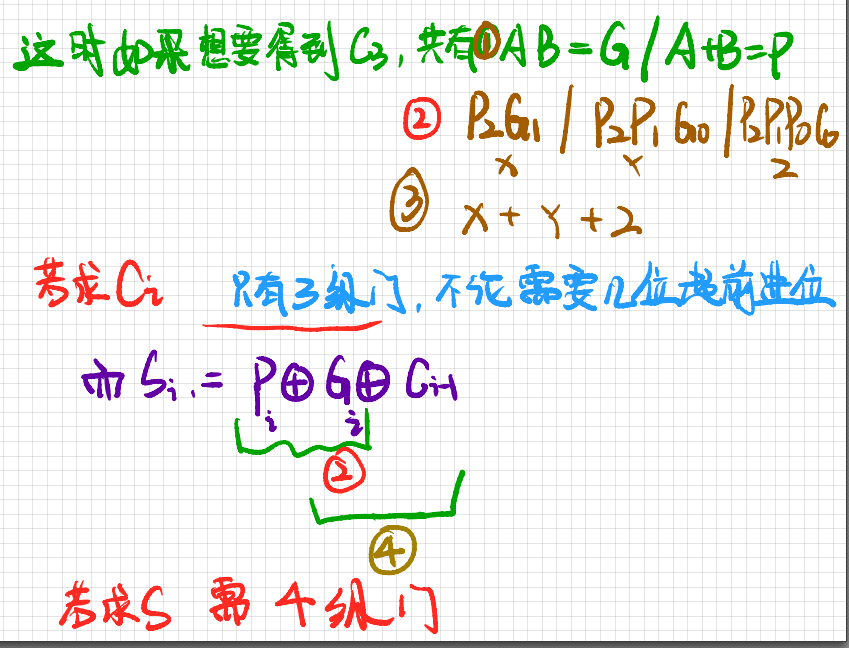
2. R. P . Brent and H. T. Kung, “A Regular Layout for Parallel Adders,”IEEE Trans. on Comp., vol. 31, no. 3, pp. 260–264, Mar. 1982.

### Parallel prefix arbiter并行前缀仲裁器

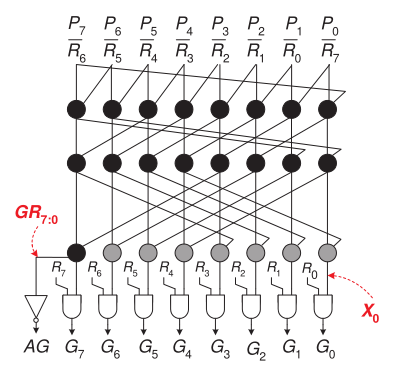
为了方便理解并行前缀的含义，首先引入超前进位加法器CLA（Carry lookahead adder）



可以看出超前进位可以在输入信号发起时，就可以计算出来，并且计算的时间时固定的。



可以看出虽然超前进位中有G P的计算，相与的计算和相或的计算，并且每一项的与都不同，其最大延迟由最大的与项决定。而并行前缀parallel prefix的含义与超前进位的含义类似但不相同，就是为了减少计算时间，将计算分解成几个步骤，其中每个步骤可以使用相同组成单元===每段处理的时间是相同的。



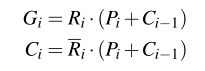
《Fast Arbiters for On-Chip Network Switches》一文找到了一种简单的方法，可以做到快速且节省功耗。

以下是推导过程的个人总结：

定义：

P [7:0] = 8’b 00000010 表示 第1位具有最高优先级，左边依次优先级降低，之后循环。请求i能否获得授予信号需要判断高优先级位有无请求输入，或当前请求是否是最高优先级位。

按照上述仲裁器的优先级判断，可以总结出两个公式：



Ri：表征当前位置是否存在请求

Gi：表征当前位置的Request是否可以获得Grant

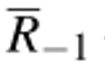
Ci：表征该位置之前的具有高优先级的位置，是否存在请求，如果高优先没有请求，那么才可以轮到该位置的Request

至此，开始化简。令，将C的表达式代入，可得：

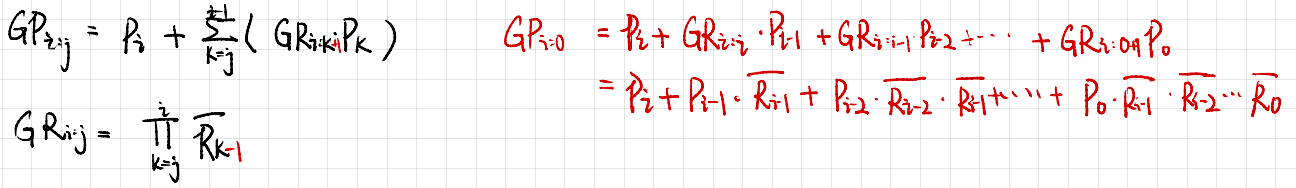


此形式与超前进位加法器的形式类似，X2可以展开得到：



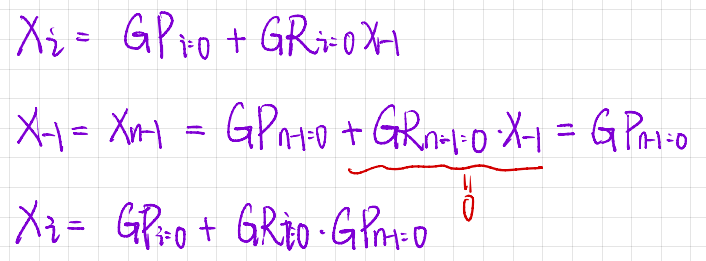
由于优先级是循环的，所以此时的是最高位的请求，例如8位请求中就是R7非，Xin也是最高位的X，X7。

很明显上面这个等式能一直循环无限套娃，那么如何获得一个具有普适性的X等式呢？



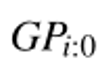


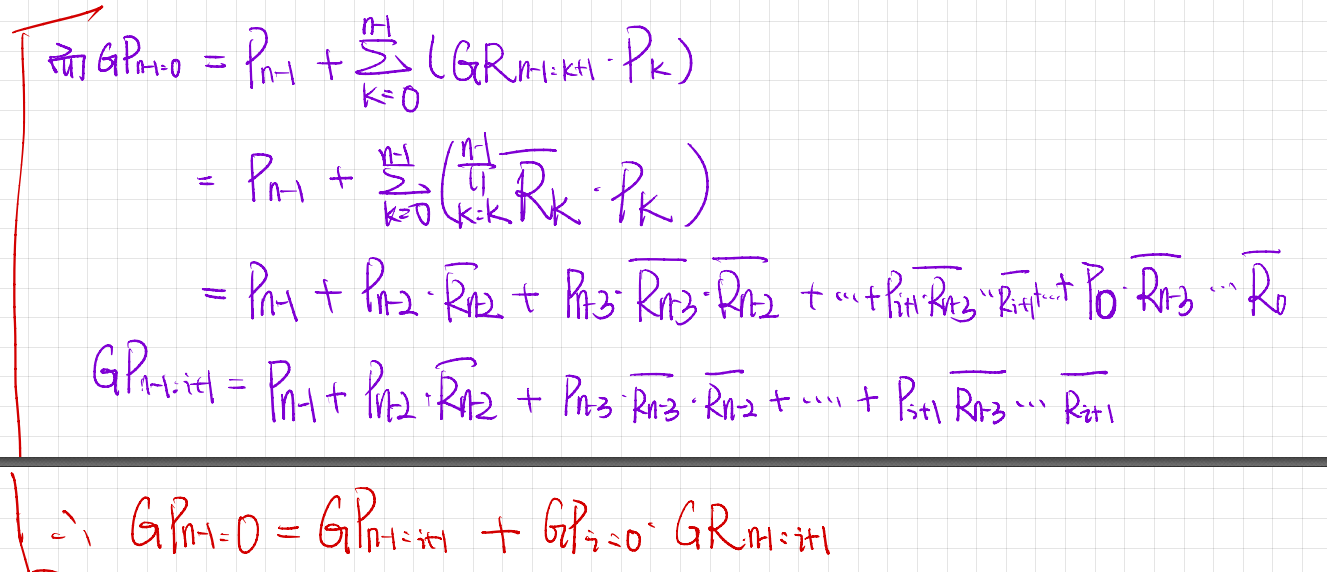
而Xin = X-1= Xn-1

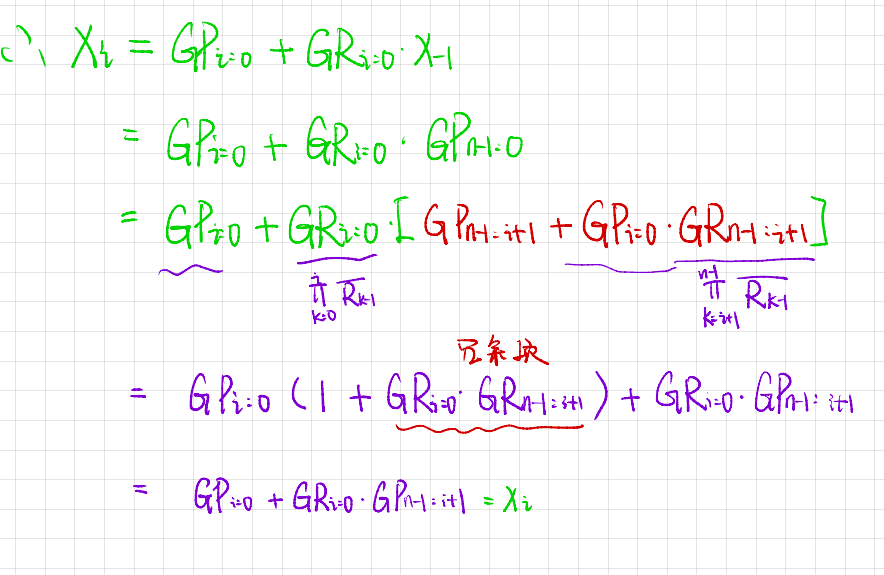


上面可以看出，X-1并不是一直套娃下去，套娃项可以忽略。

于是Xi更新为：

与之间存在包含关系，下面开始如何化简：

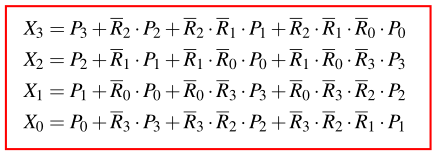




得到最终式：

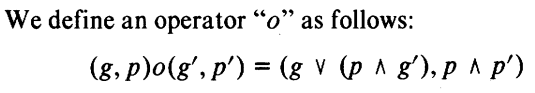


Eg ： 4输入仲裁器中的X 计算公式



如果只是到这一步，还不能称其Parallel prefix ，因为这里的每一项计算时间并不相同，无法做到并行前缀。

所以引入一种新的计算公式，对上述计算进行化简（参考文章：R. P . Brent and H. T. Kung, “A Regular Layout for Parallel Adders,”IEEE Trans. on Comp., vol. 31, no. 3, pp. 260–264, Mar. 1982.）





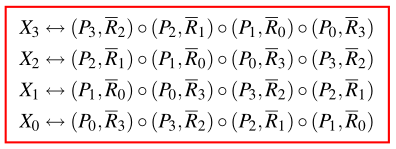
于是得到如下结论：





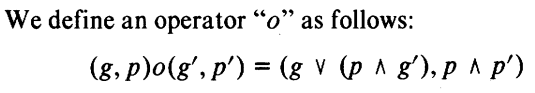


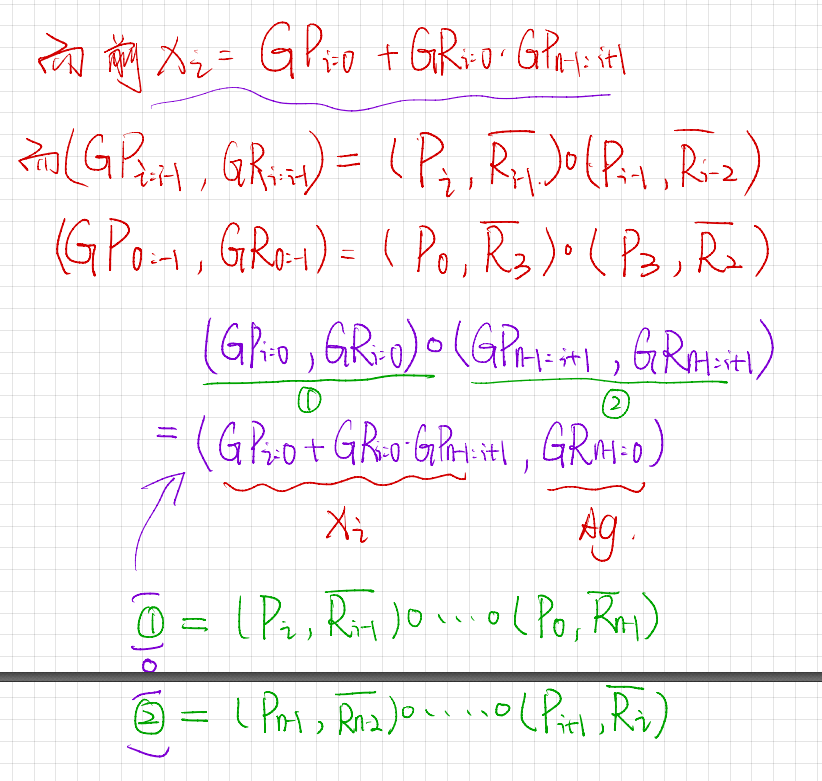
所以可得X的Parallel prefix公式：



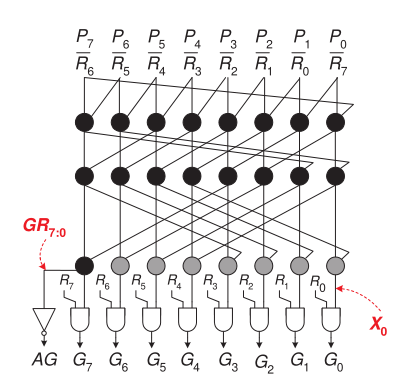
而X与GP GR的关系又如下：

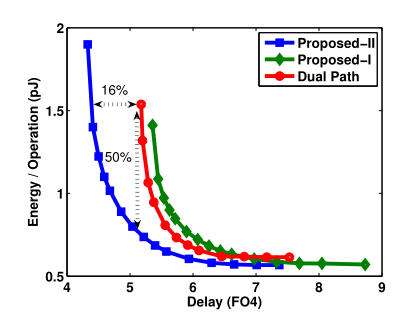


上述计算公式，套入，便可验证：



至此，数学上的推导与化简终于结束了。PPA（Parallel prefix arbiter）就可以设计了。下图就是对8输入进行仲裁，圆表征一次o操作，而AG也在每次的o操作中可以计算出来。





延迟指标的比较

### Parallel prefix round robin arbiter并行前缀轮询仲裁器

有了并行前缀的先验知识，轮询仲裁的功能，只是做一个Gnow与Pnext之间的反馈通路，当前的Grant不但作为输出，同时也会作为下一次仲裁的优先级。

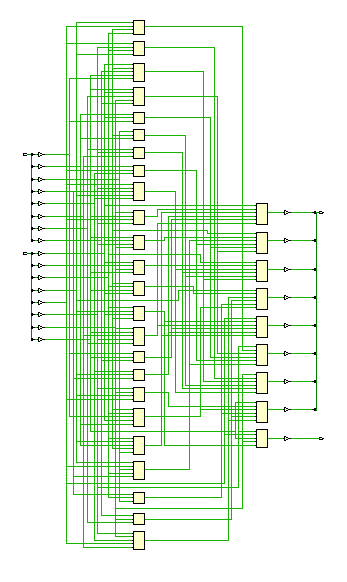
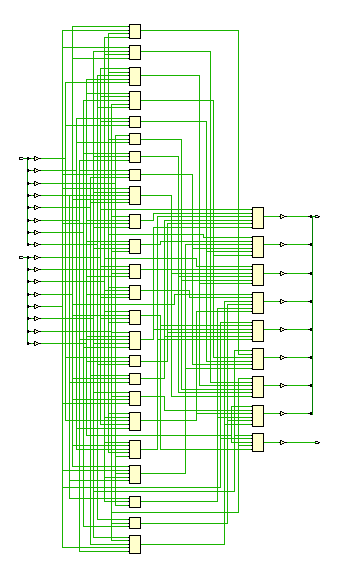
我之前实现的RRA仲裁机制是通过扩展取反+1位与相或来获得，其中优先级的变更只是对当前Grant向左移位。

那么并行前缀轮询仲裁器，不过是对Grant进行位移，存储罢了。

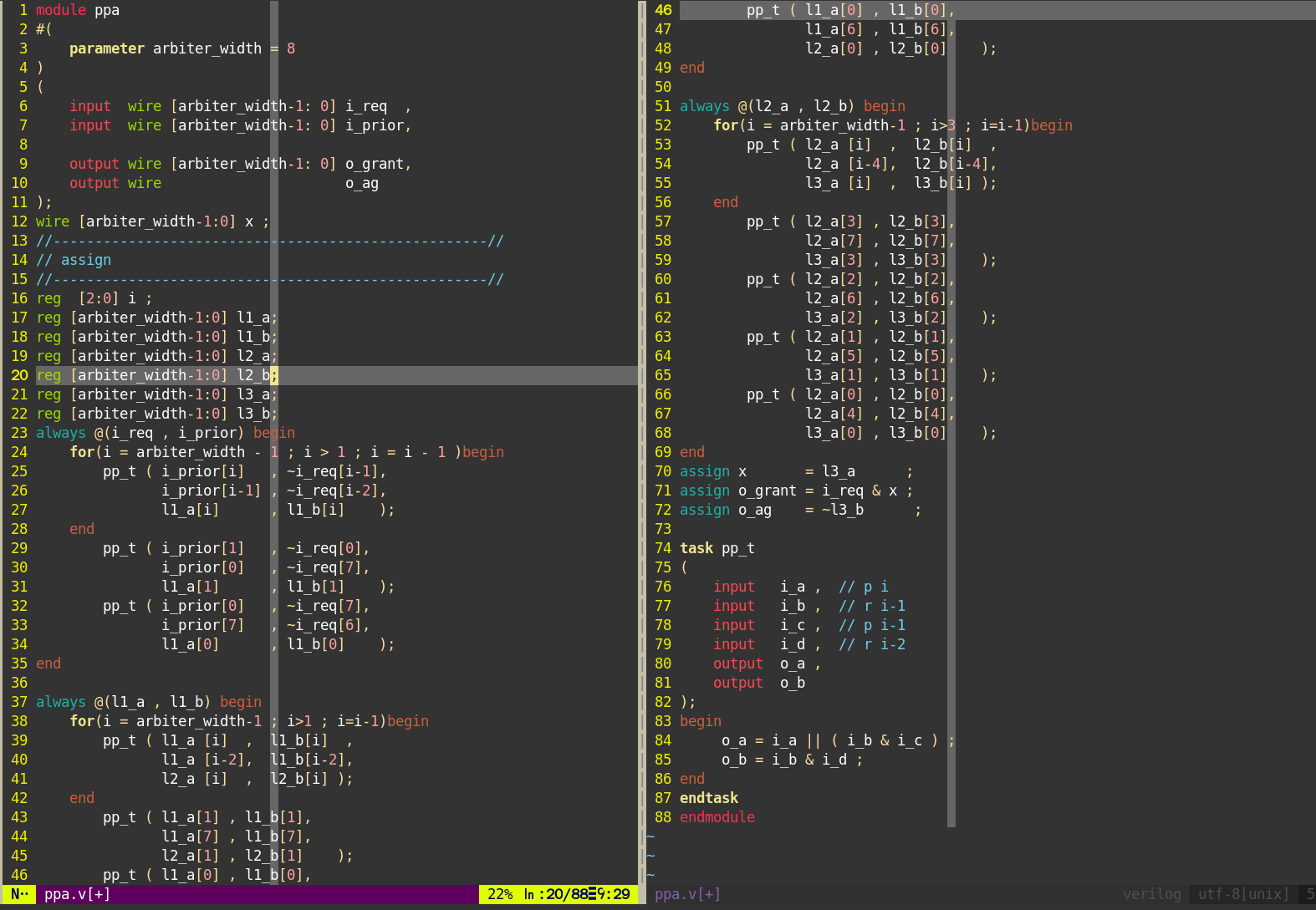
### 代码实现

先对PPA进行实现：

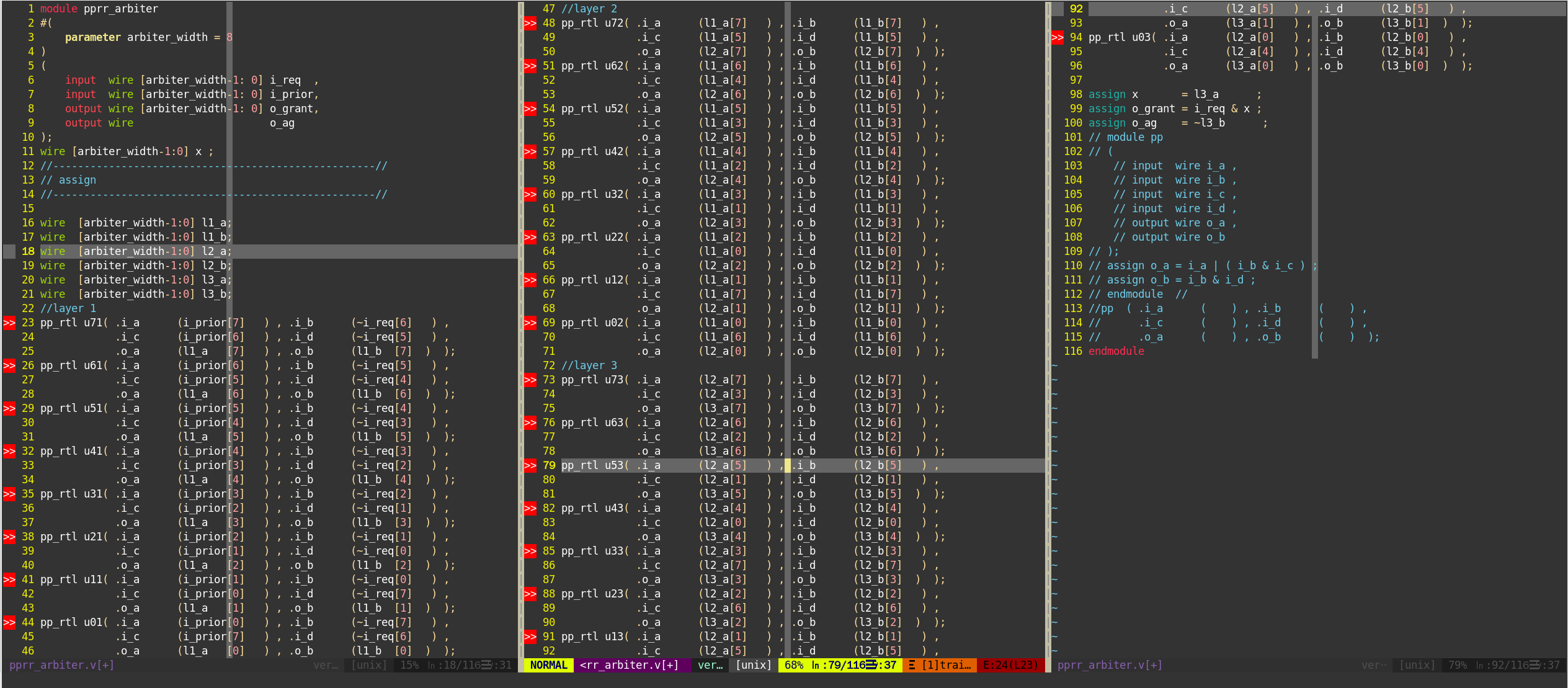
首先对TASK函数综合出的电路与直接MODULE调用综合出的电路进行对比。（相同功能，无轮询，所综合出的电路是一样的。）左边是TASK综合出的电路，右边是MODULE。

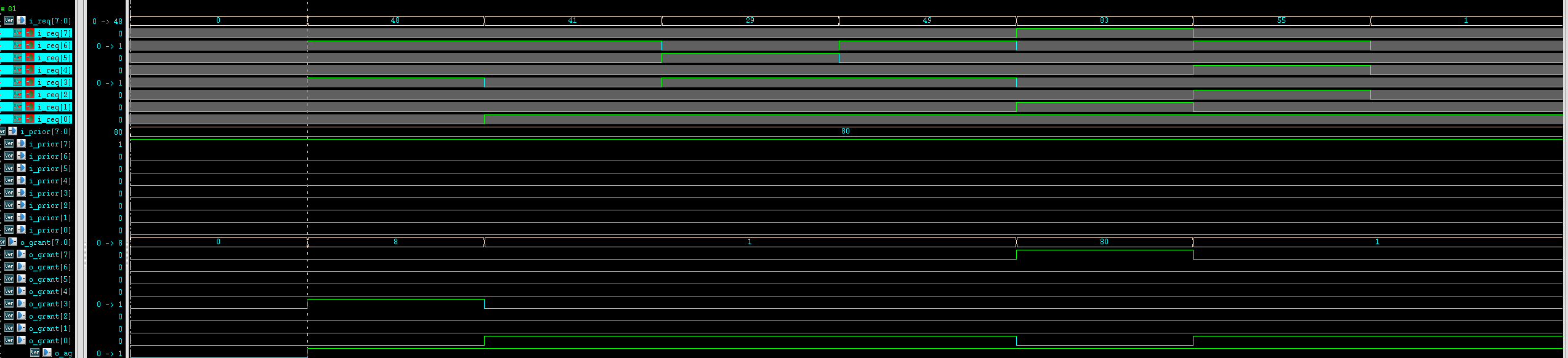
TASK代码（可综合）：



Module模块调用：



可见相同综合电路，合理使用TASK函数可以简化代码，使代码风格更容易读懂，且对综合电路无影响。但是需要注意，FOR循环不能一直循环，需要合理设计条件。



于是为了简化代码，可以合理使用TASK。

PPRRA实现：

下面是o运算

