面试必杀技：异步FIFO（上） -- CDC的那些事（5）

原创 硅谷老李 IC加油站 6月22日

这一篇老李终于要开始聊异步FIFO(Asynchronous FIFO)了。在知乎上曾经老李见过一个问题：

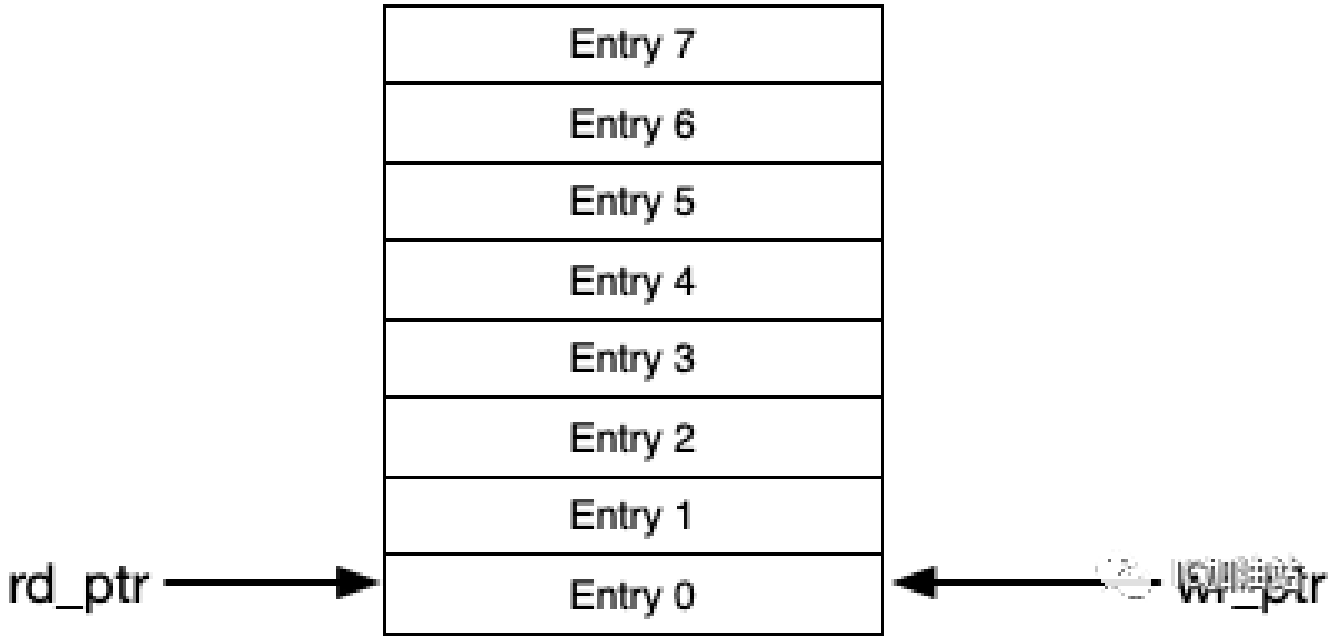
> 硕士生找工作的时候把异步fifo写成一个项目经历，是不是显得很low啊?
>
> 知乎：https://www.zhihu.com/question/38321271

下面回答很有意思，老李发现那些回答很low的以在校学生和刚毕业的应届生居多，大家都觉得简历里面一定要写上什么无线接收机，USB控制器等等才显得高大上。而几个来自真正工业界大佬的回答反而不说low，比如下面的这个回答来自海思的架构师说

> 我如果面试，不会介意。但会很乐意基于这个话题继续问你对异步原理的理解，例如格雷码的原理，两拍同步或者三拍同步的差异，或者FIFO内DATA如何保证绝对稳定等问题来进一步试探，这才是决定结果的。
>
> 夏晶晶 https://www.zhihu.com/question/38321271/answer/81593966

老李自己也基本认同这个观点，即异步FIFO里面其实需要了解的东西很多，几乎涵盖了CDC中所有相关的知识点，面试官可以就异步FIFO里面很多的点进行提问。老李当年面试硅谷各大芯片公司，几乎都被问到了异步FIFO。所以说，深刻理解异步FIFO，是一个合格的前端芯片设计工程师必须掌握的基本技能。

在开始讲异步FIFO之前，老李先带大家简单回顾一下同步FIFO。FIFO就是一个存储的管道，有进的口，有出的口。同步FIFO就是说进口（写入端）和出口（读出端）是同一个时钟域。FIFO一般深度多于1，就需要两个指针: write pointer和read pointer。



对于write pointer和read pointer我们一般用2进制，写入操作(Push)使得write pointer + 1，读出操作(Pop)使得read pointer + 1。这就像是两个人在一个环形跑道上赛跑。当write pointer领先了read pointer一圈之后，也就是说FIFO里面所有的存储单元都存了数据，FIFO没有空余的存储单元了，我们就说FIFO满了。反过来，当read pointer追上了write pointer，所有的存储单元都空闲了，我们就说FIFO空了。

对于异步FIFO来说，Push和Pop分别在不同的时钟域，那么最核心的问题就是空满的判断了。在Pop的这一侧，FIFO空不空是关键，因为空的时候不能Pop，满不满反而不重要。在Push的这一侧，反过来，满不满才关键，因为满的时候不能继续往进Push。因此，我们就要在读的这一侧判断FIFO是否空，在写的这一侧来判断FIFO是否满。当然我们还是要有read pointer和write pointer，在pop这一侧更新read pointer，在push这一侧更新write pointer。那么当我们要把pointer同步到另外的时钟域进而去比较的时候，我们就遇到了上一讲讨论的multi-bit 同步的问题，即binary counter不能直接利用double flop来同步。

那么我们上一篇讲到的带反馈的asynchronous load模块可以用来同步pointer吗？可以是可以，但是缺点也很明显，即反馈的话要跨两次时钟域，对于效率很有影响，比如说push这一侧要等到反馈信号回来之后才能继续下一个push，哪怕FIFO里面还有很多空闲的单元。pop的这一侧也是一样。这样对于FIFO的整体性能影响太大。

那有没有更快的办法呢？答案就是老李上一期最后埋的坑 -- 用格雷码Gray Code。

格雷码是以美国学者Frank Gray于1947年提出的一种二进制编码方式，后面这种编码方式就以他的名字命名。这种编码方式的特点老李以两句话概况

1. 每相邻的两个编码之间有且只有一位不同

2. 当第N位从0变到1的时候，之后的数的N-1位会关于前半段轴对称，而比N位高的位是相同的。

记住了以上两点，老李保证你可以现场推出来Gray code是什么样的。说实话，老李自己也记不住二进制到Gray code的转换公式，每次都是写出来现场推。（相信老李，面试官问你的时候除非他自己面你之前背了公式，否则他也得现推。）如果你只记得1，你现场可能推不出来，所以老李还是建议你把2也记住。下面这幅图就是用10进制，传统2进制以及Gray code来表示0-15个数。

4'd0	4'b0000	4'b0000
4'd1	4'b0001	4'b0001
4'd2	4'b0010	4'b0011
4'd3	4'b0011	4'b0010
4'd4	4'b0100	4'b0110
4'd5	4'b0101	4'b0111
4'd6	4'b0110	4'b0101
4'd7	4'b0111	4'b0100
4'd8	4'b1000	4'b1100
4'd9	4'b1001	4'b1101
4'd10	4'b1010	4'b1111
4'd11	4'b1011	4'b1110
4'd12	4'b1100	4'b1010
4'd13	4'b1101	4'b1011
4'd14	4'b1110	4'b1001
4'd15	4'b1111	4'b1000

老李所说的轴对称是什么意思呢？请看Gray code前两个数4'b0000, 4'b0001，它们俩之间可以画一条对称轴，第1-3位都是相同的。再看前4个数，在4'b0001和4'b0011之间画一条对称轴，第2、3位是相同的，第0位则是轴对称的，从0-1到1-0。之后的规律老李也在图上标出来了，一看就懂。有了这两个规律，更多位数的Gray code老李相信你也可以现场直接写出来。

然后咱们就来推一推关系，你只要大概记住需要用到异或操作，就能推出来

```
Binary to Gray
g(3) = b(3) ^ 0
g(2) = b(3) ^ b(2)
g(1) = b(2) ^ b(1)
g(0) = b(1) ^ b(0)
```

```
g(n) = b(n+1) ^ b(n)
Gray to Binary
b(3) = g(3)
b(2) = g(3) ^ g(2)
b(1) = g(3) ^ g(2) ^ g(1)
b(0) = g(3) ^ g(2) ^ g(1) ^ g(0)

```

Gray code有什么魔法之处，能够突破muti-bit不能用2flop synchronzier的限制呢？关键就在于它的第一个特点：相邻两个编码之间有且只有1位不同。我们说multi-bit如果在一个时钟沿有多个bit同时翻转，在另外一个时钟域采到的时候由于2flop 稳定需要1个或2个周期，所以可能会出现错误的值。Gray code这种编码，从根本上就没有这个问题，因为以Gray code编码作为计数器，每个时钟沿来的时候**只会有1个bit发生了翻转**，其余所有bit都是稳定的！这样即使这一个bit在用2flop synchronizer同步到另外一个时钟域时，可能需要1个周期发生变化，或者2个周期，在发生变化前，另一个域的值就是之前的稳定值，变化后就是新的值，而不会出现其他不该出现的值。

用了2flop synchronizer来同步，省去了反馈，把read pointer同步到write domain来判断满，把write pointer 同步到read domain来判断空，只需要跨一次domain，就可以判断，这样可以提高push和pop的效率。

而对于memory的取址还是得用2进制编码，我们需要做的就是在同步pointer的时候把binary pointer 转化为gray code pointer，然后用2flop synchronizer同步到对面时钟域之后，再来判断空满。这里有个问题，我们需不需要把gray code再转化为binary code之后再来比较空满呢？当然可以，我们学习新知识就是不断把问题转换为已经解决的问题，在同步FIFO的时候我们已经知道怎么用read pointer和write pointer判断空满，那么自然而然我们可以这样做。那么有没有更快的办法呢？我们能不能直接利用gray code来判断空满呢？

有！我们再仔细观察gray code。和同步FIFO一样，我们对于2^n个entry的FIFO，需要N+1个bit来表示address和gray code。以下面的编码为例，假设FIFO有8个entry，我们用4位来表示。

FIFO空比较好判断，write pointer == read pointer，用binary或者gray code都行，要求每位都相同。

满稍微复杂一点，我们举例来说，假设FIFO一开始一直写，不读，写满8个entry后write pointer 的binary变成4'b1000，gray code是4'b1100， 而read pointer的

gray code是4’b0000，可以看到高两位是相反的，之后的低位是相同的。再举个例子，假设write pointer 的gray code到了4'b1011，而这个时候read pointer如果是4'b0111，那么也是8个entry满了。

4’d0	4’b0000	4'b0000
4’d1	4’b0001	4'b0001
4’d2	4’b0010	4'b0011
4’d3	4’b0011	4'b0010
4’d4	4’b0100	4'b0110
4’d5	4’b0101	4'b0111
4’d6	4’b0110	4'b0101
4’d7	4’b0111	4'b0100
4’d8	4’b1000	4'b1100
4’d9	4’b1001	4'b1101
4’d10	4’b1010	4'b1111
4’d11	4’b1011	4'b1110
4’d12	4’b1100	4'b1010
4’d13	4’b1101	4'b1011
4’d14	4’b1110	4'b1001
4’d15	4’b1111	4'b1000

所以我们归纳出，利用gray code判断满的条件为:

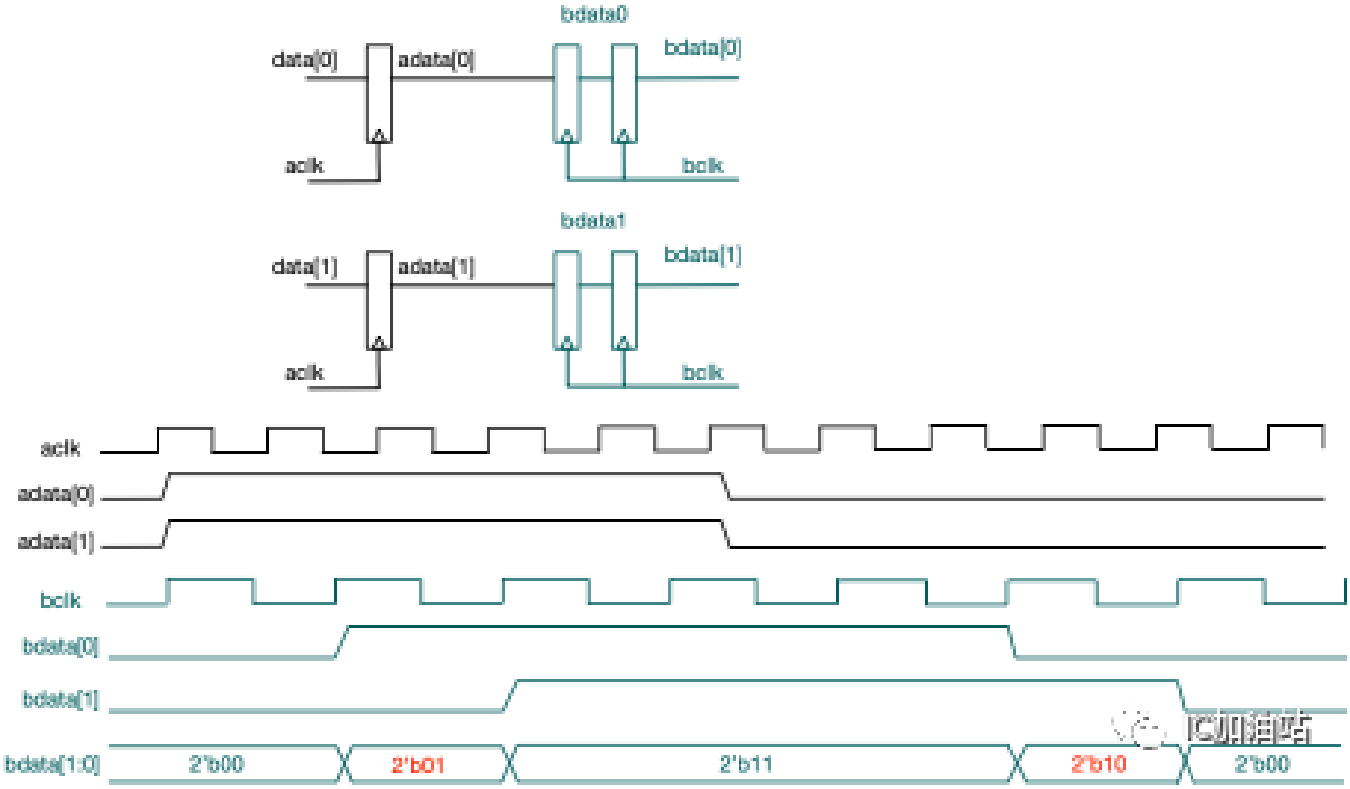
```
assign full = (write_ptr_gray[N:N-1] == ~read_ptr_gray[N:N-1])
            &&(write_ptr_gray[N-2:0] == read_ptr_gray[N-2:0]);
```

最后再说一个面试中的常见问题，这个问题知乎上也有人提

> 慢时钟域同步快时钟域格雷码时候，在慢时钟域的一个周期中，经历了两次或多次快时钟域的上升沿，那么对应的格雷码就会有两个或多个bits发生变化，这个不会产生多个bits同步的问题吗?
>
> 知乎：https://www.zhihu.com/question/290661321

说实话老李当年面试就被问过好几次这个问题。这个问题很有迷惑性，但是回答起来也很简单，其实老李上面已经回答了，这里再复习一下，继续搬出我们之前看过的图。我们说多个bit发生变化其实是针对source clock的每一个edge来说的，因为不同bit之间发生翻转的时间不能严格对齐，所以会导致destination clock可能看到不同的值，导致最后synchronizer输出会出现错误的值，从而影响FIFO的空满判断。而gray code在每个source clock的沿只会有一个bit发生翻转，其余bit保持稳定，这样**每个**

destination clock edge来的时候最多也只可能碰到1bit在翻转，这个翻转的bit可能会给synchronizer的第一级引入metastable，但是最后synchronizer的输出无非就是保持前值或者是更新后的值，而这两个值都是合理的值，不会出现一个错误的值从而导致FIFO空满判断逻辑错误。虽然慢时钟域同步过来的值可能和之前的值相比有多个bit发生变化，但是**这些bit的翻转不是同时发生**的，这是回答这道题的关键。



这周老李工作较忙，上周的推送晚了几天，拖到了这周末。下周老李继续带大家了解异步FIFO里的其他细节内容。比如前面讲的判断空满是真满、真空还是假满、假空？如果FIFO的深度不是2^n，还能用gray code吗？

如果你觉得这篇文章对你有所帮助，不妨点个右下角的“在看”，最好能够分享到群里或者朋友圈，你的支持就是老李更新的动力！