仅供参考

异 步 FIFO 结 构

（第一部分）

作者：　Vijay A.Nebhrajani

翻译：　Adam Luo

2006 年 7 月

设计一个 FIFO 是 ASIC 设计者遇到的最普遍的问题之一。本文着重介绍怎样设计 FIFO——这是一个看似简单却很复杂的任务。

一开始，要注意，FIFO 通常用于时钟域的过渡，是双时钟设计。换句话说，设计工程要处理（work off）两个时钟，因此在大多数情况下，FIFO 工作于独立的两个时钟之间。然而，我们不从这样的结构开始介绍—我们将从工作在单时钟的一个 FIFO 特例开始。虽然工作在同一时钟的 FIFO 在实际应用中很少用到，但它为更多的复杂设计搭建一个平台，这是非常有用的。然后再从特例推广到更为普通的 FIFO，该系列文章包括以下内容：

1．单时钟结构
2．双时钟结构——双钟结构 1
3．双时钟结构——双钟结构 2
4．双时钟结构——双钟结构 3
5．脉冲模式 FIFO

单时钟 FIFO 特例

FIFO 有很多种结构，包括波浪型（ripple）FIFO，移位寄存器型以及其他一些我们并不关心的结构类型。我们将集中讨论包含 RAM 存储器的结构类型。其结构如图 1 所示。

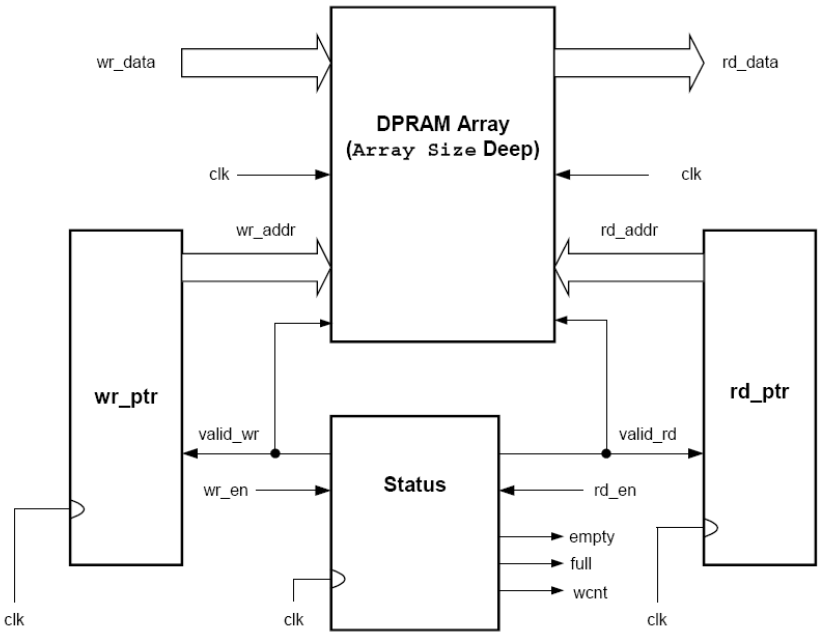


Figure 1. A FIFO Architecture

通过分析，我们看到图中有一个具有独立的读端口和独立的写端口的 RAM 存储器。这样选择是为了分析方便。如果是一个单端口的存储器，还应包含一个仲裁器保证同一时刻只能进行一项操作（读或写 ），我们选择双口 RAM（无需真正的双口 RAM，因为我们只是希望有一个简单的相互独立的读写端口）是因为这些实例非常接近实际情况。

读、写端口拥有又两个计数器产生的宽度为 $\log_2$(array_size)的互相独立的读、写地址。数据宽度是一个非常重要的参数将在在稍后的结构选择时予以介绍，而现在我们不必过分的关心它。为了一致，我们称这些计数器为“读指针”（read pointer）和“写指针”（write pointer）。写指针指向下一个将要写入的位置，读指针指向下一个将要读取的位置。每次写操作使写指针加 1，读操作使读指针加 1。

我们看到最下面的模块为“状态”(stauts) 模块。这个模块的任务实给 FIFO 提供“空”（empty）和“满”（full）信号。这些信号告诉外部电路 FIFO 已经达到了临界条件：如果出现“满”信号，那么 FIFO 为写操作的临界状态，如果出现“空”信号，则 FIFO 为读操作的临界状态。写操作的临界状态（“full is active”）表示 FIFO 已经没有空间来存储更多的数据，读操作的临界表示 FIFO 没有更多

的数据可以读出。status 模块还可告诉 FIFO 中“满”或“空”位置的数值。这是由指针的算术运算来完成了。

实际的“满”或“空”位置计算并不是为 FIFO 自身提供的。它是作为一个报告机构给外部电路用的。但是，“满”和“空”信号在 FIFO 中却扮演着非常重要的角色，它为了能实现读与写操作各自的独立运行而阻塞性的管理数据的存取。这种阻塞性管理的重要性不是将数据复写（或重读），而是指针位置可以控制整个 FIFO，并且使读、写操作改变着指针数值。如果我们不阻止指针在临界状态下改变状态，FIFO 还能都一边“吃”着数据一边“产生”数据，这简直是不可能的。

进一步分析：DPRAM 若能够寄存读出的信号，这意味着存储器的输出数据已被寄存。如果这样的话，读指针将不得不设计成“read 并加 1 ”，也就是说在 FIFO 输出数据有效之前，必须提供一个明确的读信号。另一方面，如果 DPRAM 没有寄存输出，一旦写入有效数据就可以读出；先读数据，然后使指针加 1。这将影响到从 FIFO 读出数据和实现空/满计算的逻辑。由于简化的缘故，我们仅论述 DPRAM 没有提供索锁存输出的情况。同理，将其推广到寄存输出的 DPRAM 并不是很复杂。

从功能上看，FIFO 工作原理如下所述：复位时，读、写指针均为 0。这是 FIFO 的空状态，空标志为高电平，（我们用高电平表示空标志）此时满标志为低电平。当 FIFO 出现空标志时，不允许读操作，只能允许写操作。写操作写入到位置 0，并使写指针加 1。此时，空标志变为低电平。假设没有发生读操作而且随后的一段时间 FIFO 中只有写操作。一定时间后，写指针的值等于 array_size-1。这就意味着在存储器中，要写入数据的最后一个位置就是下一个位置。在这种情况下，写操作将写指针变为 0，并将输出满标志。

注意，在这种情况下，写指针和读指针是相等的，但是 FIFO 已满，而不是空。这意味着“满”或“空”的决定并不是仅仅基于指针的值，而是基于引起指针值相等的操作。如果指针值相等的原因是复位或者读操作，FIFO 认为是空；如果原因是写操作，那么 FIFO 认为是满。

现在，假设我们开始一系列的读操作，每次读操作都将增加读指针的值，直到读指针的位置等于 array_size-1。在该点，从这个位置读出的 FIFO 输出总线上的数据是有效的。随后的逻辑读取这些数据并提供一个读信号 （在一个时钟周期内有效）。这将导致读指针再次等于写指针（在两个指针走完存储器一圈后）。然而，由于这次相等是由于一个读操作，将会输出空标志。

因此，我们将得到如下的空标志：写操作无条件的清除空标志。

Read pointer=(array_size-1) ， 读操作置空标志。

以及如下的满标志：读操作无条件的清除满标志，

Write pointer= (array_size-1)， 写操作置满标志。

然而，这是一个特殊的例子，由于一般情况下，读操作在 FIFO 不是空的情

况下就开始了（读操作逻辑不需要等待 FIFO 变满），因此这些条件不得不修改来存储读指针和写指针的每一个值。

有这样一个想法，那就是我们可以将存储器组织成一个环形列表。因此，如果写指针与读指针差值大于 1 或更多，就进行读操作，FIFO 为空，这种工作方式对于用无符号（n-bit）结构来描述的临界状态非常适合。同样的，如果读指针与写指针的差值大于 1，就进行写操作，直到 FIFO 为满。

这将带来如下的条件：
写操作无条件的清除空标志。
write_pointer=(read_pointer+1)，读操作置空。
读操作无条件的清除满标志，
read_pointer= (write_pointer+1)，写操作置满。

注意，读操作和写操作同时都在使其指针增加，但不改变空标志和满标志的状态。在空或满的临界状态同时读操作和写操作都是不允许的。

综上所述，我们现在能够定义 FIFO 的 status 模块，这里提供了用 VHDL 编写的代码，由于是同步的，很容易转换成 Verilog HDL 代码。

```
library IEEE, STD;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity status is
port (reset : in std_logic;
clk : in std_logic;
fifo_wr : in std_logic;
fifo_rd : in std_logic;
valid_rd : out std_logic;
valid_wr : out std_logic;
rd_ptr : out std_logic_vector(4 downto 0);
wr_ptr : out std_logic_vector(4 downto 0);
empty : out std_logic;
full : out std_logic
);
end status;
architecture status_A of status is
signal rd_ptr_s : std_logic_vector(4 downto 0);
signal wr_ptr_s : std_logic_vector(4 downto 0);
signal valid_rd_s : std_logic;
signal valid_wr_s : std_logic;
begin
empty_P : process(clk, reset)
```

```
begin
if (reset = '1') then
empty <= '1';
elsif (clk'event and clk = '1') then
if (fifo_wr = '1' and fifo_rd = '1') then
-- do nothing
null;
elsif (fifo_wr = '1') then
-- write unconditionally clears empty
empty <= '0';
elsif (fifo_rd = '1' and (wr_ptr_s = rd_ptr_s + '1')) then
-- set empty
empty <= '1';
end if;
end if;
end process;
full_P : process(clk, reset)
begin
if (reset = '1') then
full <= '0';
elsif (clk'event and clk = '1') then
if (fifo_rd = '1' and fifo_wr = '1') then
-- do nothing
null;
elsif (fifo_rd = '1') then
-- read unconditionally clears full
full <= '0';
elsif (fifo_wr = '1' and (rd_ptr_s = wr_ptr_s + '1')) then
-- set full
full <= '1';
end if;
end if;
end process;
valid_rd_s <= '1' when (empty = '0' and fifo_rd = '1');
valid_wr_s <= '1' when (full = '0' and fifo_wr = '1');
wr_ptr_s_P : process(clk, reset)
begin
if (reset = '1') then
wr_ptr_s_P <= (others => '0');
elsif (clk'event and clk = '1') then
if (valid_wr_s = '1') then
wr_ptr_s <= wr_ptr_s + '1';
end if;
end if;
```

```
end process;
rd_ptr_s_P : process(clk, reset)
begin
if (reset = '1') then
rd_ptr_s_P <= (others => '0');
elsif (clk'event and clk = '1') then
if (valid_rd_s = '1') then
rd_ptr_s <= rd_ptr_s + '1';
end if;
end if;
end process;
rd_ptr <= rd_ptr_s;
wr_ptr <= wr_ptr_s;
end status_A;
```

电路图如图 2 所示：

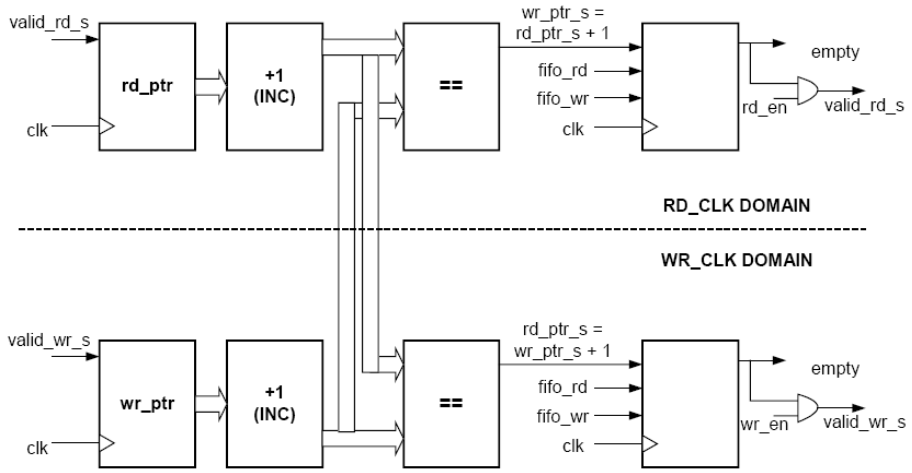


Figure 2: Circuit for Status block and pointers of FIFO of Figure 1.

细心的读者会注意到图 2 中产生满或空标志需要同时用到两个指针。在双时钟设计的情况下，希望用读指针处理(work off) 读时钟，写指针处理(work off) 写时钟。这会引起不希望发生的毛刺问题——自己可以去试一试，看一看。这些问题以及一些解决方案将在后续的该系列文章中提及。

PS： 文章中三次提到 work off clock，分别在开头和结尾处，work off 字面意思是“去除，消除，出售”的意思，可是在 FIFO 中，不应该是去除的意思，故根据前后文和常识，将其翻译为“处理”，有不对的地方请批评指正！还有两部分，会尽快翻译出来，敬请期待……

<未完待续>

Adam Luo BEJING

异步 FIFO 结构

（第二部分）

作者： *Vijay A. Nebhrajani*

翻译： *Adam Luo*

2006 年 7 月

在先前的该系列文章中，我们看到了怎样用双端口、无寄存器输出的 RAM 设计同步 FIFO。这部分我们将探讨同样的概念，并将其推广到怎样产生具有相互独立、自由工作的读、写时钟的 FIFO。拥有自由工作时钟简化了很多问题，但是这导致了一个特殊情况下的解决方法。普通情况下不对时钟进行假设，甚至不假设其自由工作。我将在本系列文章的最后一部份讨论最普通的情况。

如果你看过先前的文章，你会发现只有 status 模块工作在两个时钟。存储器没有寄存输出，所以它确实不需要用读时钟；即使它是寄存输出，也可毫无问题的运行于读时钟上。Status 模块本质的功能是对两个指针进行操作，而且这两个指针工作在不同的时钟域。这也是真正的困难所在。如果打算用写时钟来取样读指针或用读时钟来取样写指针，将不可避免的遇到一个问题：***亚稳态***。它将导致空/满标志的计算错误，并导致设计的失败。

1 亚稳态

接下来我们将系统地开始探讨亚稳态，并解决由亚稳态产生的问题。首先我们来了解一下什么是亚稳态。亚稳态是一种物理现象的名称，它发生在一个事件试图取样[1]（sample）另一事件的时候。亚稳态可以描述如下：假设一个信号在 t = 0 时刻瞬间从 0 变为 1，那么信号在 t = 0 时刻的值究竟是多少？是 0 还是 1，或者在两者之间？在亚稳态中，这个问题被定义的两个时刻回避了，分别是 0-和 0+。在 t = 0-时刻，规定信号的取值为 0，t = 0+ 时刻规定信号的取值为 1。显然，0- = 0 - 0，0+ =0 + 0。注意，这仅仅是一个数学定义，如果你正在用实际的电路做同样的事，输出将有可能是逻辑 0（0 伏）或者逻辑 1（5 伏），或者是介于 0 ~5 伏中间的某个值。正如在数学中描述的一样，物理系统中一个事件取样另一个事件产生了不可预知的结果。不可预知性也就意味着另一个迹象——亚稳态很危险。

1.1 时间分辨率（Resolution Time 翻转时间？）

当一个事件取样一个稳定值时（或者一个能稳定一段时间的值），取样值就随这个稳定值而变化。假设在 D 触发器情况下，就是 Q 值随 D 值变化。这段能够稳定取样的时间用相关的取样事件来定义，称之为时间分辨率（翻转时间？）。也就是我们所熟悉的“clock-to Q time”，或 t_{cq}。如果遇到触发器的 setup time[2] 和 hold time[2]，这将是 cell 设计者保证输入能够正确变为输出的时间。亚稳态影响物理系统的时间分辨率，同样也影响输出值。在“不稳定平衡”情况下考虑这些问题，就像“山上的球”（或者球面上的球体）你不知道它会向哪个方向滚，这个球就处于不稳定平衡状态。如果球完全不受干扰，它有可能一直呆在原地，但是微小的晃动会使球滚到山的一边或另一边。这将无法计算球从山上滚下的距离，或者无法计算球从山的哪一边滚下来。这就是亚稳态的一个准确的例子——你无法预知物理系统输出的值将会变成什么样，多久会变化，并且相当危险。换句话说，输出永远保持一个有限非零概率的亚稳态。在现实中，尽管很少有这种情况发生，但 20 倍的 clock-to Q 时间是一个合理的时间分辨率数值。在理论上，当取样操作接近被取样事件的时候，时间分辨率是无限的渐进曲线。

1.2 MTBF（平均无故障时间）与可靠性

如果一个设计中包含同步组件，无论是否愿意它都会出现亚稳态。亚稳态无法彻底消除，因此我们所做的就是计算错误概率以及在时间上来描述它。让我们来看一下，假设这里有一个物理系统亚稳态错误发生的概率为 1/1000。换句话说，每一千次采样就会因为亚稳态发生一次错误。这也意味着，每一千次，输出就会在下一个时钟沿到来时，无法变化。如果时钟频率为 1KHz，那么每秒都会有一次错误出现，MTBF 值就为 1 秒。当然，这个假设过于简单；MTBF 是一种故障概率的统计度量，并且需要更为复杂、经验化、实验化的数据来计算。对于触发器来说，这种关系依赖于电路自身的物理常数和时钟频率，记住亚稳态本身与时钟没有任何关系，但是它和 MTBF 相关[3]。自然的，我们会说一个可靠性好的电路具有很高的 MTBF 值。

1.3 同步

由于亚稳态无法彻底避免，在设计电路时一定要——

- 很好的处理错误。
- 将错误发生的概率降到最低。

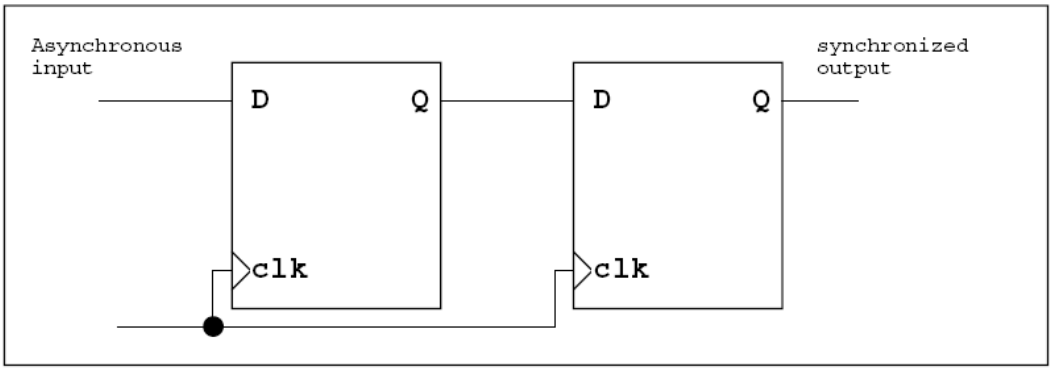


Figure 1 Dual Stage Synchronization

首先要求，设计与设计之间要有很大的区别，它并不在本文介绍的范围。第二个要求，使用 “同步”技术。这种技术由两个触发器简单的组合在一起如图 1 所示。仅当 Q1 的出现非常接近时钟沿的时候，Q2 才会进入亚稳态。如果在亚稳态情况下我们将 20 倍的 t_{cq} 作为时间分辨率，那么时钟周期将为 $t_{clk} = 20t_{cq} + t_{setup}$。这说明经过 20 倍的 clock-to Q 时间，输出仍然随输入改变的概率大大减小。因此，在时钟沿到来时 Q2 没有被改变的概率接近 P^2，这里 P 是第一级输出没有在时钟沿到来时随输入而改变的概率。这称为两级同步。当使用这个时钟频率下概率来计算 MTBF 时， MTBF 值会提高很多。如果愿意的话，可以通过三级同步进一步增加 MTBF 值。但这在实际中很少需要。

如图 2 所示，可以在电路中增加冗余的同步来很好的抵御亚稳态。在三冗余和等同于两冗余的状态下，最终的输出大部分（三分之二）可以计算出来。在这个实例中，小尺寸的布局布线与器件的差异说明了如果一个同步器产生亚稳态错误，其他的两级也会产生亚稳态错误，所有的概率将随之改变。这种技术仅在要求非常苛刻时候的用到。欲更多了解亚稳态与同步的知识，请参阅 Grosse，Debora 的“Keep metastabiliy from killing your digital design”
http://www.edn.com/archives/1942/062394/13df2.htm

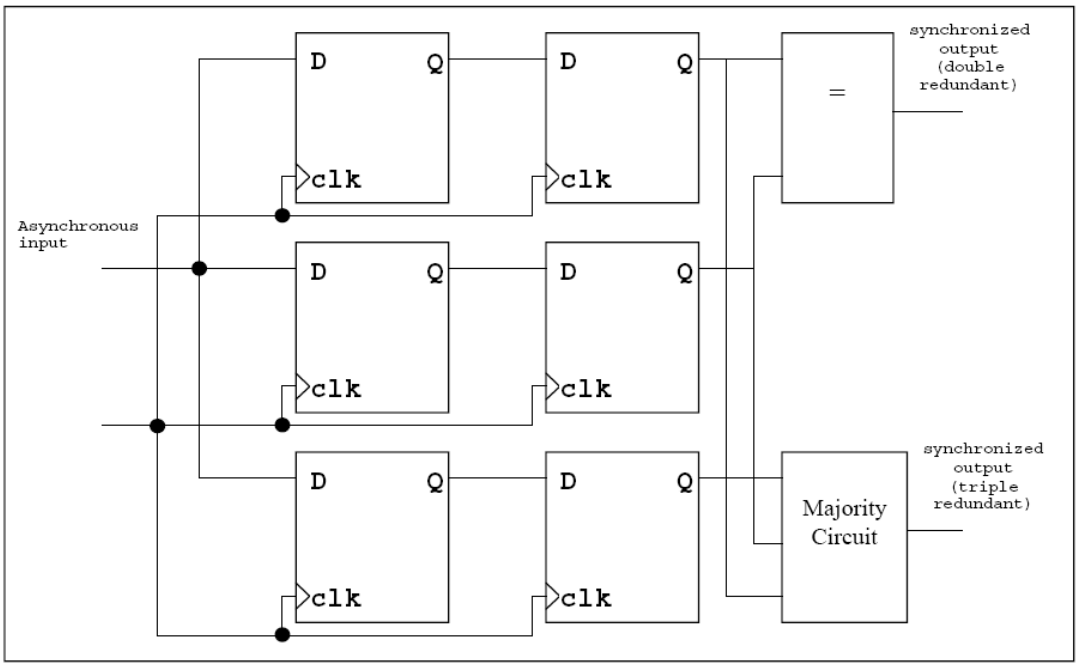


Figure 2 Double and Triple redundant 2-stage synchronization

2 采样计数器

2.1 同步：解决可靠性问题

现在我们回到有关 FIFO 的问题上来。如果要用时钟取样计数器的值，这相对于计数器时钟来说是异步的。因此，到最后不得不考虑计数器到底在哪个范围变化，假定从 FFFF 到 0000。每个单独的位（bit）都处于亚稳态。这种变化意味着有可能读数为 0000 到 ffff 之间（包含两者）的任何可能的值。当然这也说明该情况下 FIFO 将无法工作。同步可以保存处于亚稳态时的计数器取样，尽管看似很离谱但仍然可以得到取样值。换句话说，仅靠计数器同步是不够的。

重要的是我们必须确保不是所有的计数器位（bits）同时改变。实际上，不得不保证每一次计数器的增加正好改变一位。这说明计数器变化时出现错误的只可能有一位。如果计数器准备开始工作，那么至少需要一位的变化，这就是我们所能做的最好的办法。我们所需要的是用格雷码来表示的计数器。这是因为格雷码是最小距离码[4]，相邻码元之间的只有 1 位不同。

让我们来分析一下格雷码（GRAY）对于 FIFO 的指针设计有什么作用。首先，同步意味着计数器的取样值很少处于亚稳态，其次，我们取样的值最多只会有一位发生错误。这就是说计数器的真实值从 N-1 变到 N，那么无论是否发生错误读取的数不是 N-1 就是 N，而不会是其它的值。由于在变化的那一时刻，必须确定输出的值是多少，这对于读出计数器值来说是完全正确的举动。只要能够确定读出的值是旧还是新就可以了。出现其它值则是不对的。如果进一步考虑，将会发现如果在改变值的瞬间取样计数器的值，两个答案（N-1，N）对于计数器的值都是正确的。

2.2 保守的[5]报告——很好的处理错误

了解了这么多，接下来分析一下怎样将这些知识用于 FIFO 的读写指针操作。人们通常希望知道 FIFO 是否为满。如果它满了，必须阻止写操作再次发生。这

很关键，因为当 FIFO 已满时，必须停止写指针加 1。将（格雷码的）读指针与写时钟同步。因为每当同步读指针的时候，实际的读指针可能会变为不同的值。这意味着读指针可能会是一个失效的值。如果是这样，从写操作的角度考虑会发生少读现象（相比实际情况），如果条件吻合，FIFO 为满。实际上，FIFO 可能未满，因为有可能读操作发生，而从写操作的角度是“看不到”的。然而，我们只要阻止额外的写操作就 OK 了。如果当 FIFO 真的满了时我们不去阻止写操作将会出现错误。

同样的从读操作的角度看——实际上当 FIFO 中还有一些数据时，读操作一方看到“被延迟的”写操作，可能会认为 FIFO 为空。这种情况读操作被阻止直到写操作“变得可被读操作一方所看见”，它将不允许进一步的读操作。

上述被称为保守的报告。简而言之，当 FIFO 未满时，对于写操作一方报告称 FIFO 已满，当 FIFO 未空时，报告对读操作一方称 FIFO 已空。这种现象好比 FIFO 动态的缩小了一点，这毫无坏处。在字节计算的情况下，我们用同样的技术，提供写操作一方的字数计算和读操作一方的字数计算。写操作一方计算的字数可能大于 FIFO 中的真实字数。这已令人相当满意了，因为影响它的仅仅是允许其阻止下一步的写操作。同理，读操作一方字数计算会少于实际字数，那也没关系，只要确认不要将写操作一方计算的字数用于读操作一方即可，反之亦然。

这种保守的报告机构在被同步的值中能很好的处理错误。事实上，即使取样的读指针值将处于亚稳态一段时间，其影响只是阻止写操作，使 FIFO 暂停写操作，而不会引起数据错误。同理适于读操作。

3 结构 1

3.1 产生空/满标志的条件

记得上篇文章中，我们提到了指针不是影响空/满标志唯一的条件。空标志的条件是由读操作引起的读写指针相等，满标志的条件是写操作引起的读写指针相等。换句话说，要正确地产生空/满标志信号，需要用写时钟对读信号进行取样，同时用读时钟对写信号进行取样。这不同于球的游戏，因为我们不希望对时钟的频率做出假设。设想一个 10ns 的写信号（100MHz）被一个 1KHz 的读时钟取样，若无脉冲宽度延展的话就不能这样做，而且这也意味着已知（或假设已知）两个时钟之间具有某种关联。

当然，我们也不希望假设时钟之间有任何的关系。这就引起了分别围绕三种方法的问题，并且，它将引出我们即将讨论的三种不同的结构。第一种结构相当不错，将在下面描述。第二种结构也还行，但不是很好，第三种结构性能超强，但在在面积占用方面没有优势。选择哪种结构要根据自己的需求。

3.2 第一个方案

由于不可能设计出一个不考虑频率的满足脉冲采样的电路，通过对读/写指针的编码我们绕过了这个问题。构造一个指针宽度为 N+1，深度为 2^N 字节的 FIFO。为便方比较还可以将格雷码指针转换为二进制指针。

当（正被讨论的已被时钟同步的）指针的二进制码中最高位不一致而其它 N 位都相等时，FIFO 为满。当（已经过二进制转换的）指针完全相等时，FIFO 为空。这也许不容易看出，因此让我们举个例子来分析一下。

思考一下一个深度为 8 字节的 FIFO 怎样工作（使用已转换为二进制的指针）

（译者注：FIFO_WIDTH=8，FIFO_DEPTH= 2^N = 8，N = 3，指针宽度为 N+1=4 ）。起初 rd_ptr_bin 和 wr_ptr_bin 均为“0000”。此时 FIFO 中写入 8 个字节的数据。wr_ptr_bin = “1000”，rd_ptr_bin=“0000”。当然，这就是满条件。现在，假设执行了 8 次的读操作，使得 rd_ptr_bin =“1000”，这就是空条件。另外的 8 次写操作将使 wr_ptr_bin 等于“0000”，但 rd_ptr_bin 仍然等于“1000”，因此 FIFO 为满条件。

显然起始指针无需为“0000”。假设它为“0100”，并且 FIFO 为空，那么 8 个字节会使 wr_ptr_bin =“1100”，，rd_ptr_bin 仍然为“0100”。这又说明 FIFO 为满。

这个例子的意义就在它生动地说明了读/写指针怎样产生空/满标志的。我曾说过第一个方案是最好的？你到不如将其这种技术与同步 FIFO 一起使用。它可以避免算数运算，提高 FIFO 的速度。

3.3 实现

我们知道在 FIFO 中要用到格雷码计数器。而不是用由格雷码换算的二进制码计数器（它不能实现每个计数器换后只有 1 位发生变化），必须使用真正的格雷码计数器。如果想实现格雷码计数，你会发现它并不像看起来那么容易。当然，你可以创建一个定制的机构来完成这项工作，但还是让我来提供一个更为普遍的解决问题的方法。大家知道格雷码于二进制码之间能够相互转换用到一个简单的公式：

二进制码转格雷码

$g_n=b_n$

$G_i=b_i \oplus b_i+1 \qquad \forall \ i \neq n$

格雷码转二进制码

$b_n=g_n$

$b_i=g_i \oplus b_i+1 \qquad \forall \ i \neq n$

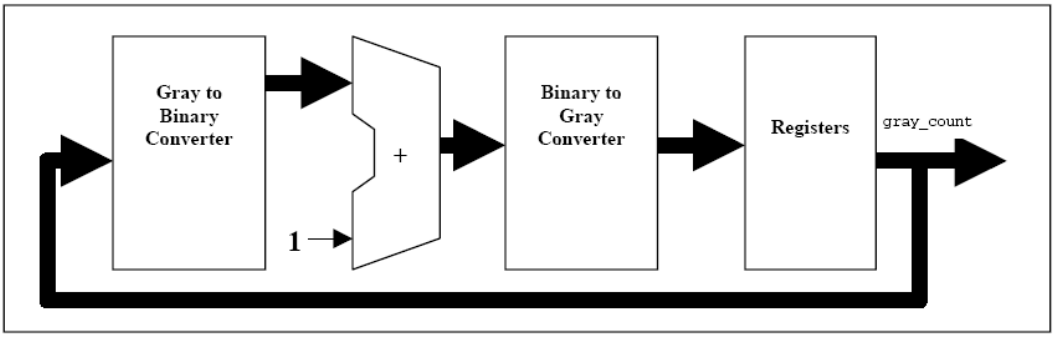


Figure 3 Generalized Gray counter architecture

在上面的公式中，下标表示 n+1 位二进制码或格雷码的位数。

我们还知道计数器不过是一个触发器组和一个累加器而已，我们可以按照下面的方法来做——将格雷码码元转换为二进制码元，然后加 1，再它转换回格雷码并存储。这是解决产生 n-bit 格雷码算法棘手问题一个普遍的方法。由它生成的计数器如图 3 所示。

当用综合工具优化时，相信综合工具能够为格雷码计数器提供一个相当快速

的电路。当然，如果希望拥有一个深度为 32 字节的 FIFO 时，可在格雷码编码的状态机中手工编写计数器代码。

最终的 FIFO 设计如图 4 所示。这次我不再提供代码，因为我相信无论用 VHDL 或 Verilog HDL 编写，那都是一件非常容易的事。

在这里我要补充一点：如果你观察图 4，会注意到有 4 个格雷码~二进制码转换器，这并不浪费。但通过保留格雷码指针的低 n-bit 和二进制最高位将有可能避免使用这些转换器。这是一种“混合”计数器，我将它作为练习留给读者。

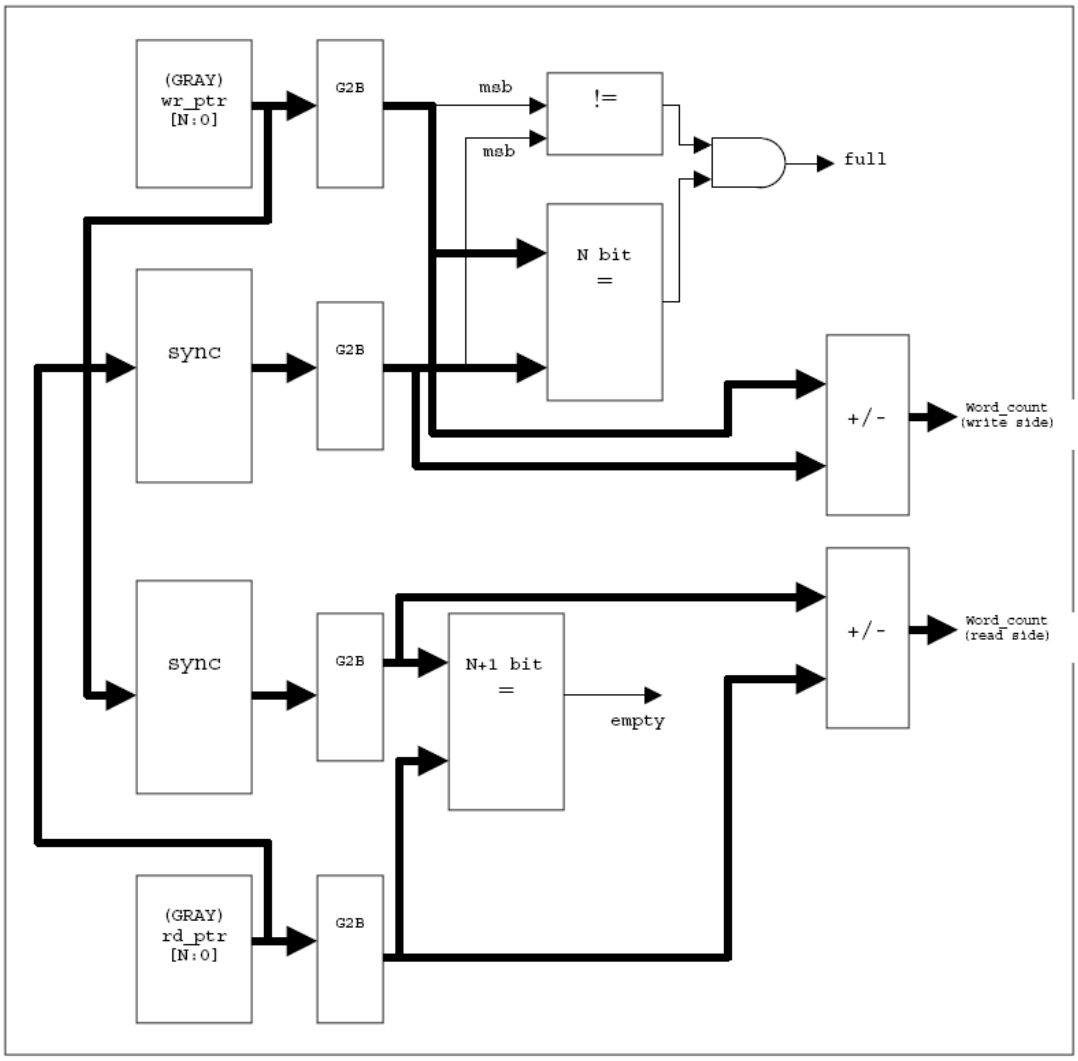


Figure 4 Status Block for Architecture 1

3.4 时间考量

管理 FIFO 工作的首要时间条件是时钟的最高频率。在上述的 FIFO 条件下，不得不面对几个参量——时钟频率不能大于存储器所需频率，必须满足亚稳态时间关系 $t_{clk}=20t_{cq}+t_{setup}$。当然，在公式中 20 这个因数完全凭借经验，倘若已经完成系统 MTBF 的计算，也可以选择其它值。另外，还应考虑格雷码计数器能够运行多快，因为上述公式要求受制于 XOR（异或）门的速度。由于本文没有做任何设想（除了同步，而这并非真正意义上的设想），时间不会引起很多同样的问题。

在这个设计中最应考虑的是用格雷码计数器和同步时避免与亚稳态相关的

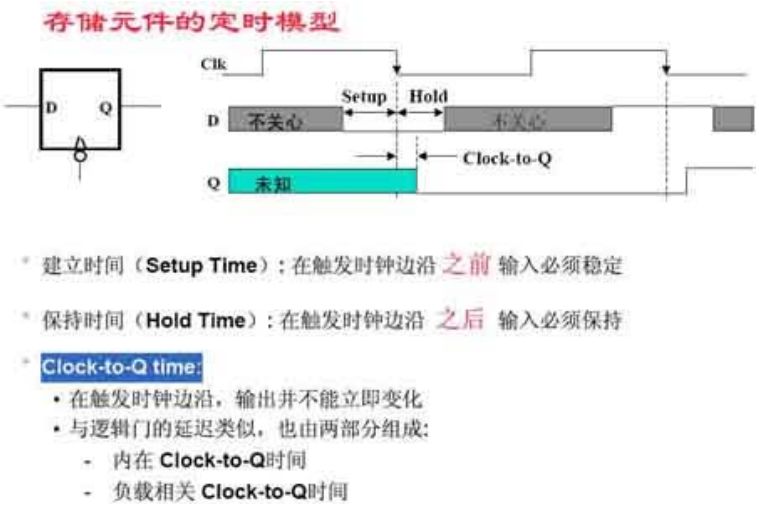
错误。要认识到发生同步错误，整个 FIFO 将无法工作（2bit 错误就将意味着在 DPRAM 中一个完全不同的地址，因为地址也用格雷码表示）——也就是 FIFO 即可以吃入数据也可以吐出数据。所以，我无法列出更多的要点，而这些要点都基于用户设计中 MTBF 值，要做好你所能承担的最坏的打算。

<未完待续>

译者注释：

[1] sample 在这里翻译为“取样”，而不是“采样”、“抽样”，是想有别于奈奎斯特定理中抽样的概念。汉语中在触发器构成的电路里很少听到“取样”这种说法，但文中用到了 sample 一词，姑且译作“取样”，其实它表示了时序电路中两个信号相遇时的状态、稳定程度及先后关系。

[2] 关于 setup time 和 hold time 借用北大的一个 PPT 来解释其意义。



[3] 实际中，MTBF 与时钟频率也有关，若一个时钟为 1MHz 的系统其 MTBF 为 10 年，当时钟变为 10 MHz 时，其 MTBF 有可能变为 1 秒钟。

[4] 最小距离码：码的距离定义为相邻两个码元之间互异元素的个数比如 101010 与 110011，从左到右，互异的元素有，第 2 个，第 3 个，第 6 个，所以码距为 3 。这种定义又称为码的汉明距离。格雷码相邻码只有一位不同，因此格雷码的距离为 1。

[5] 原文为 Pessimistic Reporting，字面翻译为悲观的报告，但综合上下文的意思来看，作者想表述得的意思是宁少不多，宁慢不快。应该是一种出于保险的，保守的考虑，宁可少读一次 FIFO ，也不让 FIFO 出错，顾翻译为“保守的报告”

限于水平，以上翻译中的错误在所难免，欢迎大家批评指正！

Adam Luo

异步 FIFO 结构

（第三部分）

作者：Vijay A.Nebhrajani

翻译：Adam Luo

2006 年 7 月 30 日

在本系列文章的第一部分我们了解了FIFO的一般结构，并分析了单时钟FIFO的一个特例[1]。第二部分描述了双时钟设计的一种可能的结构。在第三部分我们将探究一种具有新颖结构的双时钟FIFO。这种结构未必更好——只是另一种实现的方法而已。

1 工作原理

至此我们已经解决了用格雷码表示的不同时钟域的所有计算，包括多位二进制计算。本篇所介绍的结构与以往并没有什么不同，唯一的区别在判断引起读写指针相等条件的方法。

如果还记得先前的文章，文中提到读写指针相等意味着无论是满条件还是空条件，依赖于读操作还是写操作导致了指针的相等。在同步 FIFO 的第一个例子中，这很容易判别，因为两种操作均与一个时钟有关。在第二种结构中，这个条件已被编码于指针中。我们现在将探究双时钟设计的第二种方法。

1.1 方向标志（Direction flags）

在这种结构中，我们让指针轨迹的标志相等。我们称其为“Direction flags（方向标志）”。这个标志告诉状态电路 FIFO “当前朝向（headed）”。它假设写操作引起的 FIFO 朝越来越满的方向与读操作引起的 FIFO 朝越来越空的方向为 FIFO 的朝向。

不用说，每边（读操作或写操作）都必须保留独立的方向标志复本(copies)并且维持在保守状态。因此对于写操作一方将有其自己的方向标志来维护保守性。也就是，从读操作一方可能会看到写操作被延迟并且读操作一边也将维持方向标志，它可以根据延迟的写操作来计算。就像先前的双时钟结构，这将确认 FIFO 没有在吞或吐数据，但这样做是以 FIFO 尺寸动态缩小为代价的。

FIFO 满/空标志的计算基于这些方向标志，其思想是如果 FIFO 的朝向为向越来越满的方向，并且指针相等，则 FIFO 真正为满。如果 FIFO 朝向越来越空的方向，并且指针相等，则 FIFO 确实为空。

1.2 方向标志的实现

有很多不同的方法实现方向标志：一般的想法是当 FIFO 的字节计算超过某个预定上限，就认为 FIFO “going towards full(趋向满) ”，当字节计算低于预定下限是，就认为 FIFO “going towards empty（趋向空）”。

一些设计人员选择“going towards full”的门限为 FIFO 容量的 75%，“going towards empty”的门限为 FIFO 容量的 25%。还有人选择两个门限都为 FIFO 容量的 50%。也有选择 80% 和 20%的。门限的选择可由自己来决定，要根据设计选择最适合的门限。也可以根据时钟的速度与门限值得关系来确定以便使标志失效的可能性最小，但我不确定门限的选择会让设计的系统变得更好。我认为上限与下限之间有滞后或许更好（滞后的意思是上限与下限之间的差并且“going full”门限要大大超于“going empty”门限）。

我们不妨选择 FIFO 容量的 75%和 25%作为门限。这样做比较有效，因为你只需比较指针的高两位就能决定是否越过门限。若用另一些值，你将不得不比较指针的所有位，而这有可能影响你所设计的系统的速度。像以前一样，写操作的

一方可以看到写指针和一个被同步的读指针，两个指针均为格雷码。然后，将格雷码指针转换为二进制指针并计算出 FIFO 中有多少数据。如果 FIFO 中的数据量大于“going full”门限，就置位方向标志。当 FIFO 中的数据小于“going empty”门限，就清除方向标志。

同理，读操作看到（格雷码的）读指针和一个被同步的（格雷码的）写指针。在完成格雷码到二进制码的转换后，计算 FIFO 中的字节数；如果字数小于“going empty”门限，就置位方向标志（此时方向标志的反指向(the opposite sense) 作为写操作一方的方向标志），当字数增加，大于“going full”门限时就清除方向标志。

记住，当选择 75％和 25%作为门限时，上述计算无需比较指针的全部位。只用指针的高两位就足够了。

1.3 空/满的计算

在写操作一方，如果指针相等并且方向标志置位，则 FIFO 的满标志置位，同理，在读操作一方，如果方向标志置位并且指针相等，FIFO 的空标志置位。注意，这意味着我们不排除空/满标志同时置位的可能性。尽管听起来不合常理，但对于 FIFO 是正确的条件。你也许会想 FIFO 怎么可能同时既满又空呢。然而如果你进一步分析，就会发现“满”只是一个写操作一方的流控制机构，“空”只是一个读操作一方的流控制机构。如果 FIFO 的读写操作两边的 Blocks 同时流动那么这就是正确的——它并未破坏存储器或指针。当 FIFO 真的不能再接收数据或当不能再提供更多的数据时 FIFO 没有报告是非常危险的。仔细分析以前所讲的结构，它证明这种可能性不排除存在于其它结构中。

下面列出了计算方向标志每一边（读或写）的计算（注意公式中的指针已被适当的同步并且转换为二进制）：

```
word_count       = wr_ptr - rd_ptr + 1                    if wr_ptr > rd_ptr
                   fifo_size - (rd_ptr - wr_ptr) + 1      if rd_ptr > wr_ptr
direction_flagwr = 1                                      if word_count > going_full_threshold
                   0                                      if word count < going_empty_threshold
direction_flagrd = 1                                      if word_count < going_empty_threshold
                   0                                      if word_count > going_full_threshold
```

如图 1 所示为 75%和 25%门限的特列。在这个特例中，上述字节计算公式仅需要二进制码指针的最高两位，并且不需要加 1 得到字数的精确值。只需知道是否越过门限而已。

还要记住，在写操作和读操作一边的门限不必相同；可以根据读写时钟频率调整门限值以优化性能。

2 结论

这种结构是在同步 FIFO 情况下提出的命题中的一种变异。这种同步 FIFO 是这个结构的一个特例——“going full”和“going empty”的门限分别等于（fifo_size-1）和 1。

这种结构的表现比先前所提到的异步结构有着明显的优点码？未必。在一些临界情况下，这种结构可能会有优势——对于时间紧张的情况下 N-bit 格雷码转

换为二进制码也没有问题，但在 N+1 位（在先前的结构中需要的）格雷码情况下就没有这种优势，或当面积比较紧张的时候为了N＋1 位转换而占用额外的面积这种结构就不适用了。我的观点是，它们都是真正的非常优秀的结构，因此选择哪种结构取决于你的偏好。

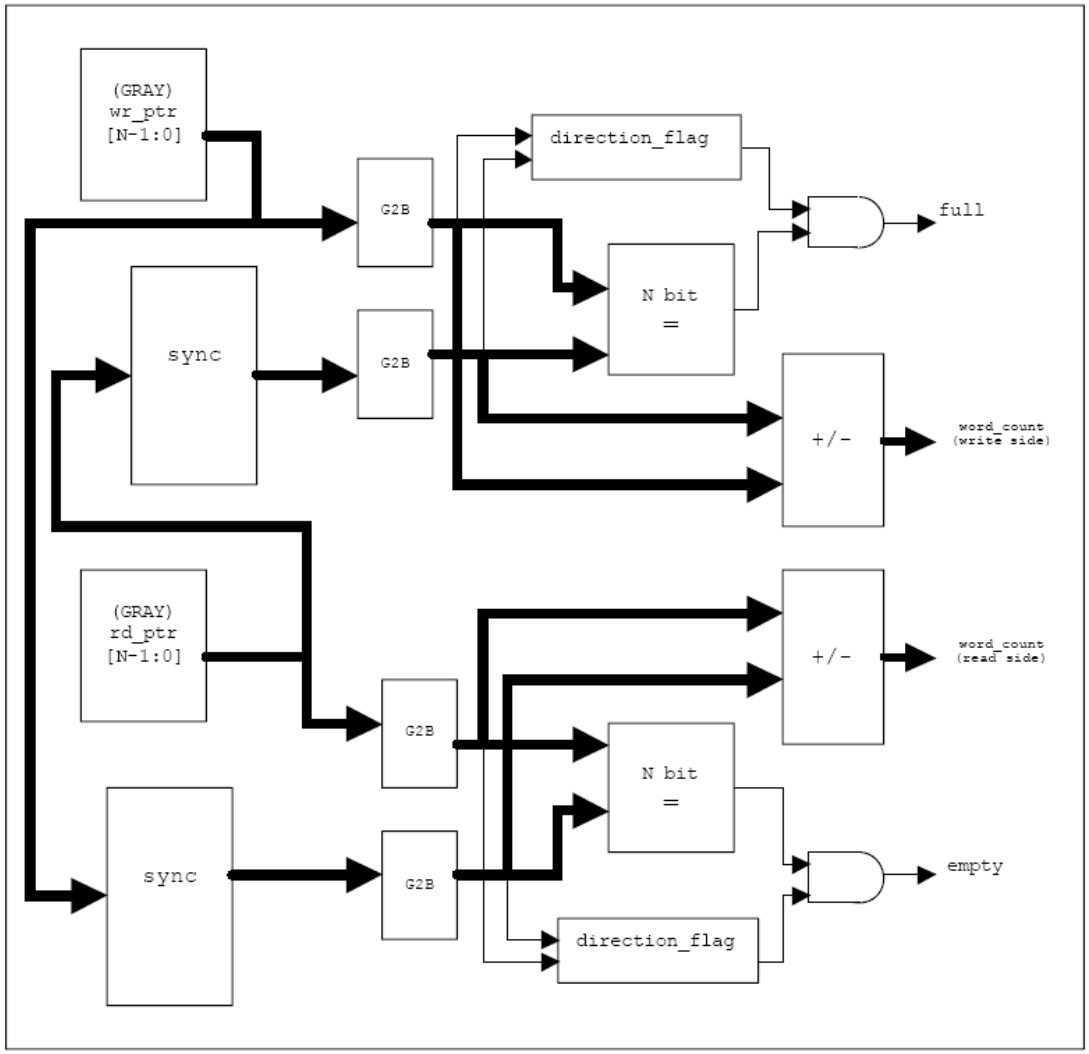


Figure 1 Status block for second dual clock FIFO architecture.

< The end>

[1] 勘误：第一篇中关于 Trivial case,应当翻译为“特例”trivial case 指其解不影响全局的事件。一般译作琐事，但很显然在此为琐事不合适，取其意义。

[2] 第二篇文章中出现了 the most significant bit.简称 MSB，与 LSB 对应，意为最高位。我们常用简称，往往看到全程不知道怎么翻译了，抱歉。

Adam Luo
2006.07.30