

东南大学电子科学与工程学院

实 验 报 告

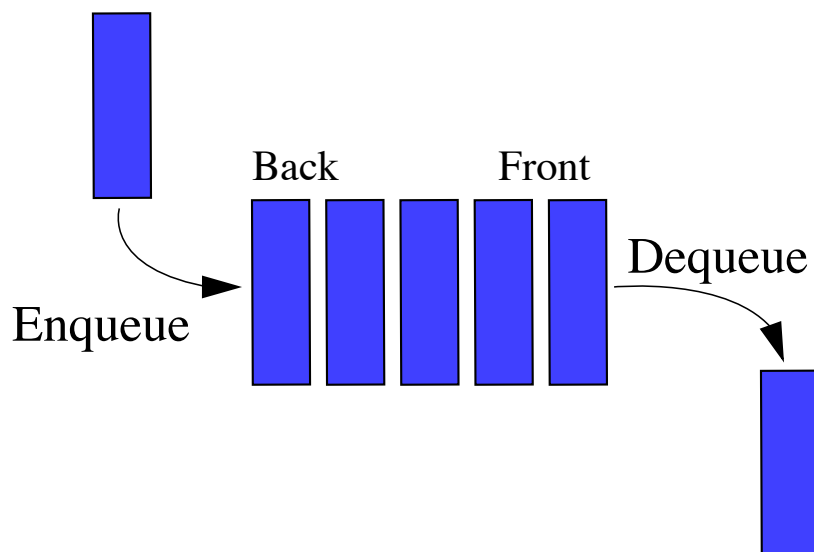
课程名称：数字模拟集成电路课程设计

| | |
|-------|------------|
| 实验名称： | Async-FIFO |
| 姓 名： | 孙寒石 |
| 学 号： | 06219109 |
| 实验地点： | 东南大学无锡国际校区 |
| 实验时间： | 2022-12-15 |
| 评定成绩： | |
| 审阅教师： | |

Async-FIFO

1 FIFO 概述

在计算和系统理论中，FIFO 是先进先出（先进先出）的首字母缩写词，是一种组织数据结构（通常，特别是数据缓冲区）操作的方法，其中最旧的（第一个）条目，或队列的“头”，首先被处理。这种处理类似于以先到先得 (FCFS) 的方式为排队区域中的人们提供服务，即按照他们到达队列尾部的相同顺序。FCFS 也是 FIFO 操作系统调度算法的行话，它按照需要的顺序为每个进程中央处理器 (CPU) 提供时间。FIFO 的对立面是 LIFO，后进先出，其中最年轻的条目或“栈顶”首先被处理。优先级队列既不是 FIFO 也不是 LIFO，但可以临时或默认采用类似的行为。排队论包含这些处理数据结构的方法，以及严格 FIFO 队列之间的交互。该存储器的特点是数据先进先出（后进后出）。其实，多位宽数据的异步传输问题，无论是从快时钟到慢时钟域，还是从慢时钟到快时钟域，都可以使用 FIFO 处理。



根据应用的不同，FIFO 可以实现为硬件移位寄存器，或使用不同的内存结构，通常是循环缓冲区或一种列表。有关抽象数据结构的信息，请参阅队列。大多数 FIFO 队列的软件实现都不是线程安全的，需要一种锁定机制来验证数据结构链是否一次仅由一个线程操作。

FIFO 通常用于电子电路中，用于硬件和软件之间的缓冲和流量控制。在其硬件形式中，FIFO 主要由一组读写指针、存储和控制逻辑组成。存储可以是静态随机存取存储器 (SRAM)、触发器、锁存器或任何其他合适的存储形式。对于非平凡大小的 FIFO，通常使用双端口 SRAM，其中一个端口专用于写入，另一个端口专用于读取。

Peter Alfke 于 1969 年在 Fairchild Semiconductor 实施了第一个已知的 FIFO。

2 原理

同步 FIFO 是一种 FIFO，其中相同的时钟用于读取和写入。异步 FIFO 使用不同的时钟进行读取和写入，它们会引入亚稳态问题。异步 FIFO 的常见实现使用格雷码（或任何单位距离码）来读取和写入指针，以确保可靠的标志生成。关于标志生成的另一注意事项是必须使用指针算法来为异步 FIFO 实现生成标志。相反，可以使用漏桶方法或指针算法在同步 FIFO 实现中生成标志。

硬件 FIFO 用于同步目的。它通常被实现为一个循环队列，因此有两个指针：

- 读指针/读地址寄存器

- 写指针/写地址寄存器

FIFO 状态标志的示例包括：已满、空、几乎已满和几乎为空。当读取地址寄存器到达写入地址寄存器时，FIFO 为空。当写入地址寄存器到达读取地址寄存器时，FIFO 已满。读取和写入地址最初都在第一个内存位置，FIFO 队列为空。

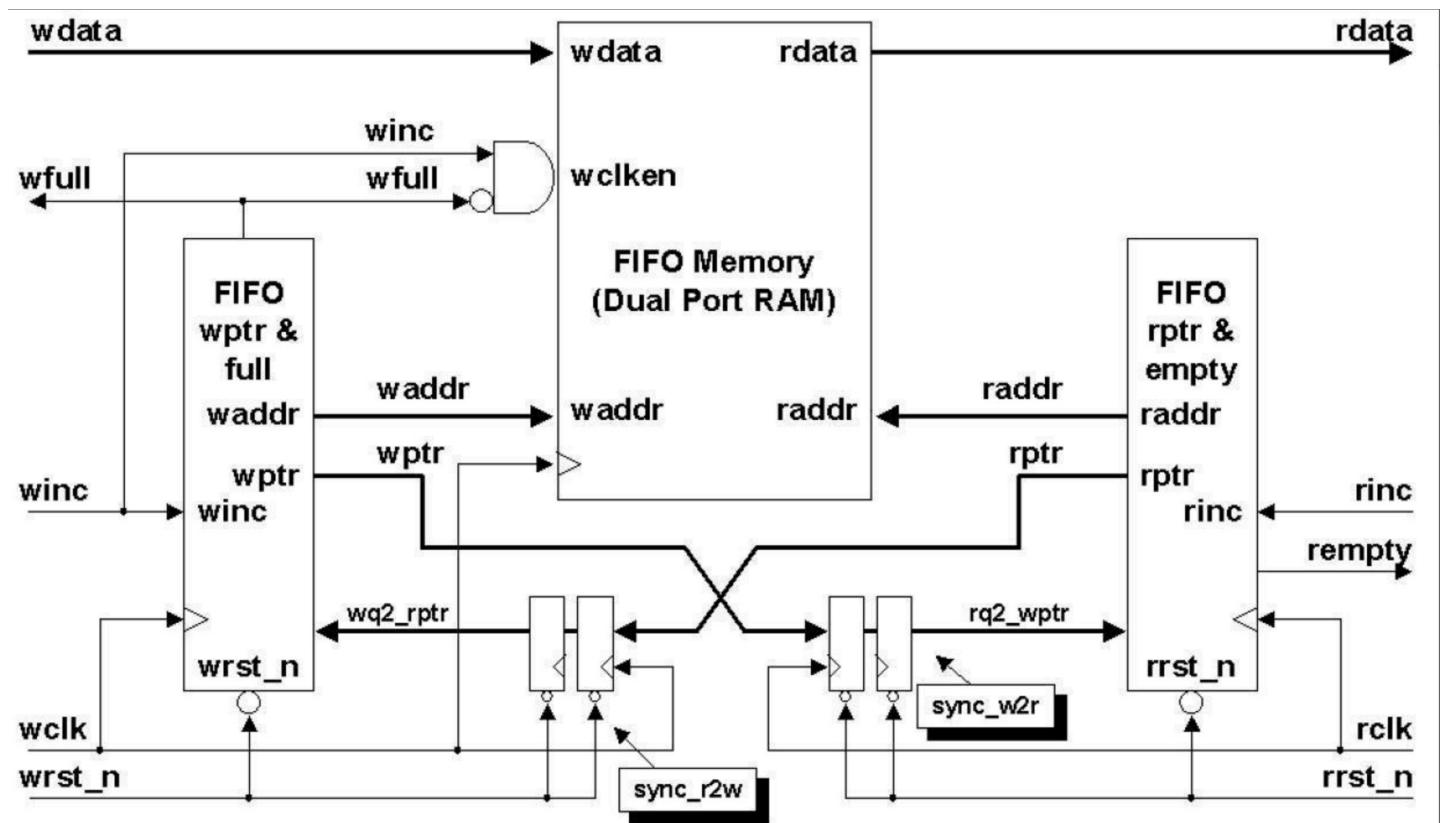
在这两种情况下，读地址和写地址最终都是相等的。为了区分这两种情况，一个简单而稳健的解决方案是为每个读写地址添加一个额外的位，该位在每次地址回绕时被反转。通过此设置，消歧条件为：

- 当读地址寄存器等于写地址寄存器时，FIFO 为空。
- 当读写地址寄存器仅在额外的最高有效位不同而其余位相同时，FIFO 已满。

复位之后，在写时钟和状态信号的控制下，数据写入 FIFO 中。RAM 的写地址从 0 开始，每写一次数据写地址指针加一，指向下一个存储单元。当 FIFO 写满后，数据将不能再写入，否则数据会因覆盖而丢失。

FIFO 数据为非空、或满状态时，在读时钟和状态信号的控制下，可以将数据从 FIFO 中读出。RAM 的读地址从 0 开始，每读一次数据读地址指针加一，指向下一个存储单元。当 FIFO 读空后，就不能再读数据，否则读出的数据将是错误的。

FIFO 的存储结构为双口 RAM，所以允许读写同时进行。典型异步 FIFO 结构图如下所示。端口及内部信号将在代码编写时进行说明。



读写时刻

关于写时刻，只要 FIFO 中数据为非满状态，就可以进行写操作；如果 FIFO 为满状态，则禁止再写数据。关于读时刻，只要 FIFO 中数据为非空状态，就可以进行读操作；如果 FIFO 为空状态，则禁止再读数据。不管怎样，一段正常读写 FIFO 的时间段，如果读写同时进行，则要求写 FIFO 速率不能大于读速率。

读空状态

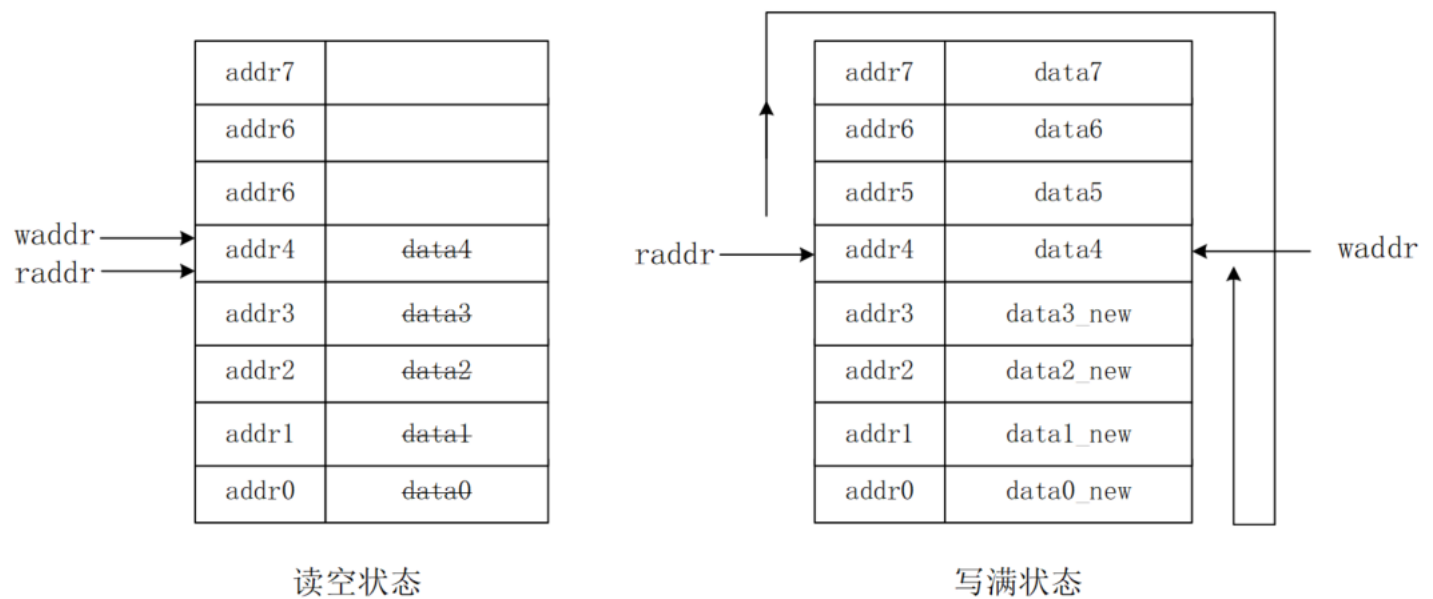
开始复位时，FIFO 没有数据，空状态信号是有效的。当 FIFO 中被写入数据后，空状态信号拉低无效。当读数据地址追赶上写地址，即读写地址都相等时，FIFO 为空状态。

因为是异步 FIFO，所以读写地址进行比较时，需要同步打拍逻辑，就需要耗费一定的时间。所以空状态的指示信号不是实时的，会有一定的延时。如果在这段延迟时间内又有新的数据写入 FIFO，就会出现空状态指示信号有效，但是 FIFO 中其实存在数据的现象。

严格来讲该空状态指示是错误的。但是产生空状态的意义在于防止读操作对空状态的 FIFO 进行数据读取。产生空状态信号时，实际 FIFO 中有数据，相当于提前判断了空状态信号，此时不再进行读 FIFO 数据操作也是安全的。所以，该设计从应用上来说是没有问题的。

写满状态

开始复位时，FIFO 没有数据，满信号是无效的。当 FIFO 中被写入数据后，此时读操作不进行或读速率相对较慢，只要写数据地址超过读数据地址一个 FIFO 深度时，便会产生满状态信号。此时写地址和读地址也是相等的，但是意义是不一样的。



此时经常使用多余的 1bit 分别当做读写地址的拓展位，来区分读写地址相同的时候，FIFO 的状态是空还是满状态。当读写地址与拓展位均相同的时候，表明读写数据的数量是一致的，则此时 FIFO 是空状态。如果读写地址相同，拓展位为相反数，表明写数据的数量已经超过读数据数量的一个 FIFO 深度了，此时 FIFO 是满状态。当然，此条件成立的前提是空状态禁止读操作、满状态禁止写操作。

同理，由于异步延迟逻辑的存在，满状态信号也不是实时的。但是也相当于提前判断了满状态信号，此时不再进行写FIFO 操作也不会影响应用的正确性。

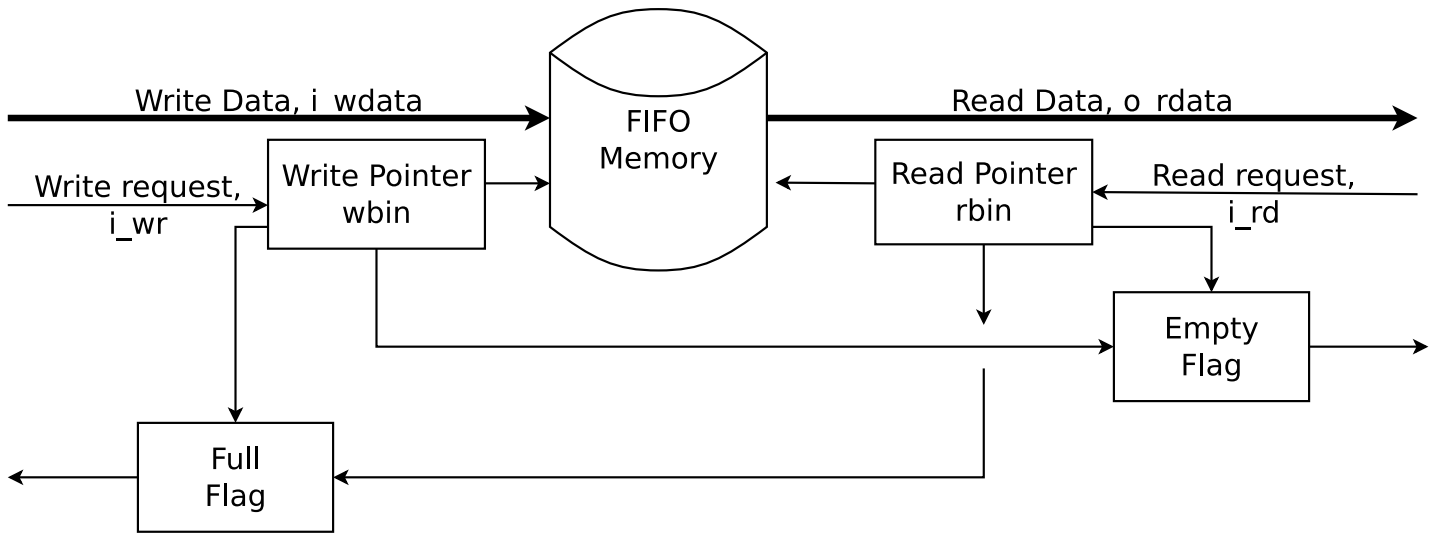
3 设计分析

在大规模ASIC或FPGA设计中，多时钟系统往往是不可避免的，这样就产生了不同时钟域数据传输的问题，其中一个比较好的解决方案就是使用异步FIFO来作不同时钟域数据传输的缓冲区，这样既可以使相异时钟域数据传输的时序要求变得宽松，也提高了它们之间的传输效率。此文内容就是阐述异步FIFO的设计。同步和异步 FIFO 都有写指针。我们称之为 `wbin`。然后，在任何写入时，我们都会将此指针增加一个——但前提是 FIFO 尚未满。同步模块 `synchronize to write clk`，其作用是把读时钟域的读指针`rd_ptr`采集到写时钟 `wr_clk` 域，然后和写指针 `wr_ptr`进行比较从而产生或撤消写满标志位 `wr_full`；类似地，同步模块 `synchronize to read clk` 的作用是把写时钟域的写指针 `wr_ptr` 采集到读时钟域，然后和读指针`rd_ptr`进行比较从而产生或撤消读空标志位 `rd_empty`。另外还有写指针 `wr_ptr` 和写满标志位 `wr_full` 产生模块，读指针 `rd_ptr` 和读空标志位 `rd_empty` 产生模块，以及双端口存储RAM模块。

为设计应用于各种场景的 FIFO，这里对设计提出如下要求：

- (1) FIFO 深度、宽度参数化，输出空、满状态信号，并输出一个可配置的满状态信号。当 FIFO 内部数据达到设置的参数数量时，拉高该信号。
- (2) 输入数据和输出数据位宽可以不一致，但要保证写数据、写地址位宽与读数据、读地址位宽的一致性。例如写数据位宽 8bit，写地址位宽为 6bit（64 个数据）。如果输出数据位宽要求 32bit，则输出地址位宽应该为 4bit（16 个数据）。
- (3) FIFO 是异步的，即读写控制信号来自不同的时钟域。输出空、满状态信号之前，读写地址信号要用格雷码做同步处理，通过减少多位宽信号的翻转来减少打拍法同步时数据的传输错误。格雷码与二进制之间的转换如下图所示。

以下是同步FIFO的图，为了比较和异步FIFO的区别，这里我们就给出。



```

1  initial wbin = 0;
2  always @(posedge i_wclk or negedge i_wrst_n)
3  if (!i_wrst_n)
4      wbin <= 0;
5  else ((i_wr) && (!o_wfull))
6      wbin <= wbin + 1;

```

注意到此逻辑是使用写时钟域发生的 `i_wclk` 或者复位也是写时钟域中的下降沿驱动的异步复位。您希望从异步 FIFO 中得到这一点。同步 FIFO 中的写指针逻辑 将是相同的，只是只使用一个时钟，很可能还有一个同步复位。它们都需要在任何写入时将传入数据放入存储器（块 RAM）中。

```

1  always @(posedge i_wclk)
2  if ((i_wr) && (!o_wfull))
3      mem[wbin[AW-1:0]] <= i_wdata;

```

在异步 FIFO 的情况下，这也专门使用与写通道关联的时钟完成 `i_wclk`。读取逻辑与上面的写入逻辑非常相似。该逻辑首先调整将调用的读取地址指针 `rbin`。像 `wbin` 上面一样，这比实际寻址缓冲区中的值所需的位多了一位——因此它有 `N+1` 位来访问 `2^N` 数据点。以类似于写指针的方式，这个指针也需要递增：任何时候有读请求并且缓冲区不为空。

```

1  initial rbin = 0;
2  always @(posedge i_rclk or negedge i_rrst_n)
3  if (!i_rrst_n)
4      rbin <= 0;
5  else if ((i_rd) && (!o_empty))
6      rbin <= rbin + 1;

```

作为最后一步，我们将从内存中读取并返回结果。

```

1  assign o_rdata = mem[rbin[AW-1:0]];

```

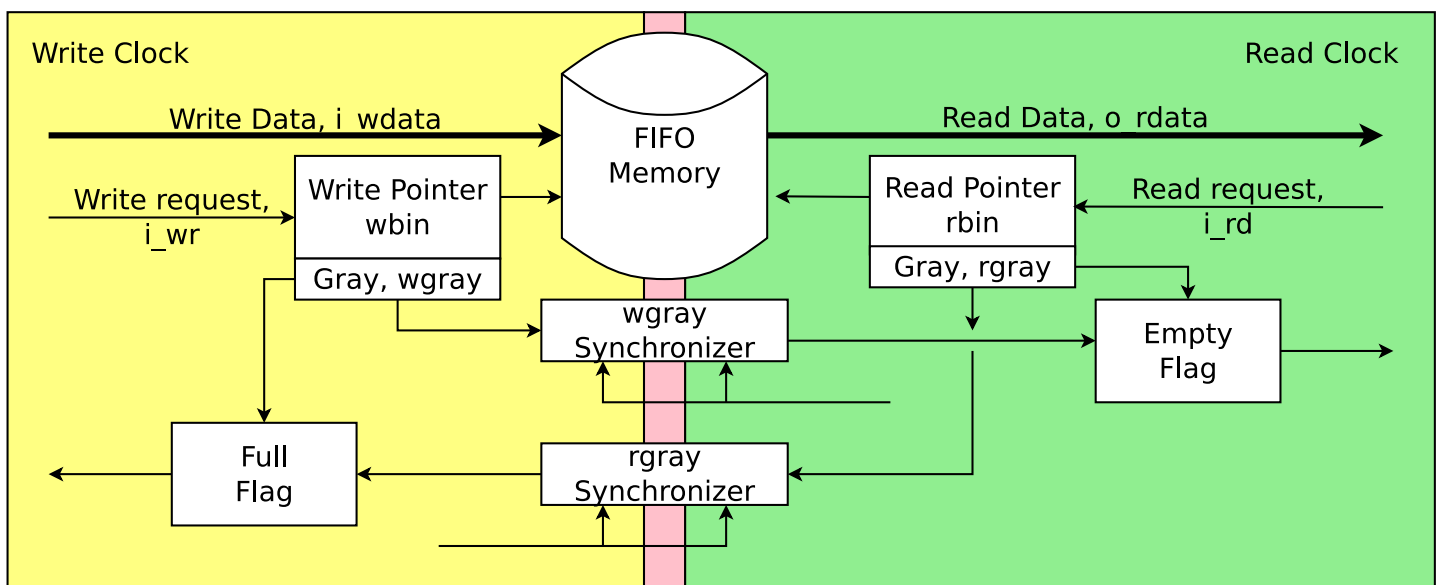
与大多数数字设计问题一样，问题在于细节。在这种情况下，仔细看一下两个标志，`o_wfull` 表示 FIFO 满了，`o_empty` 表示空了。作为计算这些的第一次尝试，可以用组合逻辑来表达它们，如下所示：

```

1 // The FIFO is empty when both read and write pointers point to the
2 // same location.
3 assign o_empty = (wbin == rbin);
4
5 // It is full when wbin-rbin = 2^N. In that case, the bottom AW
6 // address bits are identical, but the top bit is different.
7 assign o_wfull = (wbin[AW] != rbin[AW])
8               && (wbin[AW-1:0]==rbin[AW-1:0]);

```

对于同步 FIFO，两个 `AW+1` 位指针都是在同一个时钟上生成的，因此不会立即出现明显的问题。当然，可以调整此逻辑以便注册 `o_empty` 和 `o_wfull` 标志，但它们仍将具有这些相同的基本值。

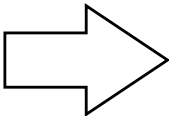


在这种一位差异的情况下，一位变化是早一个时钟到达还是晚一个时钟到达并不重要——它只是一个没有依赖性的慢信号。

我们并没有尝试将 1 位信号从一个时钟域交叉到另一个时钟域，而是从一个时钟域 `N` 到下一个时钟域的位（即位 `AW`）信号——无论是读时钟端还是写时钟边。如果我们将整个字放入一个同步器中，具有更多位，但它们也可能不再稳定。例如，如果 `wbin` `rbin` `rbin` 都是 1 并过渡到全 0，然后可能会在同步器的输出端设置一些不受控制的随机数的 1——这取决于位的路由方式，因此哪些位在新时钟信号之前到达，哪些位在新时钟信号之后到达。例如，`8'hff` 过渡到 `8'h00` 可能被理解为 `8'h52`（在许多其他可能性中）。

解决方案是以一种选定的形式将地址从一个传递到另一个时钟域，以便任何时候只有一位会发生变化。形式上，我们可以描述这样一个，要求之间的差异不能超过一位。使用格雷编码计数器，我们可以跨时钟域同时读取和写入地址指针。这意味着我们将在我们的 FIFO 设计中添加格雷编码指针，以便在时钟域分界线上桥接它们。首先，读指针，现在格雷编码并表示为 `rgray`，将交叉到写时钟域。

指向存储器的地址指针由二进制计数器产生，而用于跨时钟域传播的格雷码指针是对二进制指针的实时转换并用寄存器采集获得的。这里要注意的是，计数器的位宽比实际所需的位宽要多出一位，这样做的目的是方便判断FIFO的空或满。

| Counter | | Gray Code |
|---------|---|-----------|
| 00000 | | 00000 |
| 00001 | | 00001 |
| 00010 | | 00011 |
| 00011 | | 00010 |
| 00100 | | 00110 |
| 00101 | | 00111 |
| 00110 | | 00101 |
| 00111 |  | 00100 |
| 01000 | | 01100 |
| 01001 | | 01101 |
| 01010 | | 01111 |
| 01011 | | 01110 |
| 01100 | | 01010 |
| 01101 | | 01011 |
| 01110 | | 01001 |
| 01111 | | 01000 |
| ... | | ... |

```
gray_value = (counter>>1)^counter;
```

如果我们有一个计数器，例如，

```
1 | always @(posedge i_clk)
2 |     counter <= counter + 1;
```

那么同一计数器的格雷编码)版本将具有一次仅更改一位的属性。我们可以 通过将计数器本身向下移动一个来创建这个格雷编码。

```
1 | assign graycounter = counter ^ (counter >> 1);

1 | initial { wq2_rgray, wq1_rgray } = 0;
2 | always @(posedge i_wclk or negedge i_wrst_n)
3 | if (!i_wrst_n)
4 |     { wq2_rgray, wq1_rgray } <= 0;
5 | else
6 |     { wq2_rgray, wq1_rgray } <= { wq1_rgray, rgray };
```


这只是一个 N 位宽的双触发器同步器。

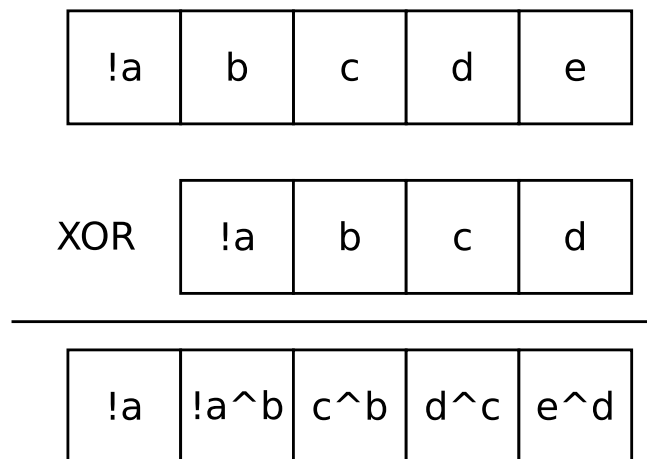
其次，写入指针，格雷编码为 `wgray`，将从写入时钟域跨越到读取。

```
1  initial { rq2_wgray,  rq1_wgray } = 0;
2  always @(posedge i_rclk or negedge i_rrst_n)
3  if (!i_rrst_n)
4      { rq2_wgray, rq1_wgray } <= 0;
5  else
6      { rq2_wgray, rq1_wgray } <= { rq1_wgray, wgray };
```

这将我们的两个指针 `rbin` 和作为 `wbin` 带入另一个时钟域，但这些值不再是计数器。让我们研究一下如何使用这两个格雷指针。请记住，我们需要确定 FIFO 何时为空，何时为满。

```
1  assign o_reempty = (wbin == rbin);
2  assign o_wfull   = (wbin[AW] != rbin[AW])
3                    && (wbin[AW-1:0]==rbin[AW-1:0]);
```

如果检查格雷编码计数器，会注意到格雷编码值是唯一的——就像它们所代表的计数器一样。此外， N 位计数器可以用 N 位格雷编码来表示。也就是说，如果要检查两个指针是否相同，只需要检查两个格雷编码指针是否相同即可。写指针唯一不同的位是最高有效位，如图，读取指针和写入指针之间除了前两位之外的所有位都是相同的。因此，我们可以通过测试是否最高两位相反，但其余位相同来测试两个指针是否除最高位外其他所有位都相同。



```
1  assign o_wfull   = (wgray[AW:AW-1] == ~wq2_rgray[AW:AW-1])
2                    && (wgray[AW-2:0]==wq2_rgray[AW-2:0]);
```

```

1  always @(posedge i_rclk or negedge i_rrst_n)
2  if (!i_rrst_n)
3      o_empty <= 1'b0;
4  else
5      o_empty <= (rq2_wgray == rgray);

```

和

```

1  always @(posedge i_wclk or negedge i_wrst_n)
2  if (!i_wrst_n)
3      o_full <= 1'b0;
4  else
5      o_full <= (wgray[AW:AW-1] == ~wq2_rgray[AW:AW-1])
6      && (wgray[AW-2:0] == wq2_rgray[AW-2:0]);

```

异步FIFO最核心的部分就是精确产生空满标志位，这直接关系到设计的成败。本文采用比较读写指针来判断FIFO的空满，如果FIFO的深度是n-1位线所能访问到的地址空间，那么此设计所要用的指针位宽就比实际多出一位，也就是n位，这样做有助于判断FIFO是空还是满。当读地址`rd_ptr`赶上写地址`wr_ptr`，也就是`rd_ptr`完全等于`wr_ptr`时，可以断定，FIFO里的数据已被读空，而且只有在两种情况下，FIFO才会为空：第一种是系统复位，读写指针全部清零；另一种情况是在FIFO不为空时，数据读出的速率快于数据写入的速率，读地址赶上写地址时FIFO为空。空标志位的产生需要在读时钟域里完成，这样不至于发生FIFO已经为空了而空标志位还没有产生的情况，但是可能会发生FIFO里已经有数据了而空标志位还没有撤消的情况，不过就算是在最坏情况下，空标志位撤消的滞后也只有三个时钟周期，这个问题不会引起传输错误；还有一种情况就是空标志比较逻辑检测到读地址和写地址相同后紧接着系统产生了写操作，写地址增加，FIFO内有了新数据，由于同步模块的滞后性，用于比较的写地址不能及时更新，这样，一个本不应该有的空标志信号就产生了，不过这种情况也不会导致错误的发生，像这种FIFO非空而产生空标志信号的情况称为“虚空”。

写时钟域的写指针通过两级寄存被同步到读时钟域之后与读指针进行比较，如果完全相等，则会产生空标志信号；同步模块用两级寄存器来实现是为了消除可能的亚稳态，正如前面所述，因为`wr_ptr_gray`是用格雷码实现的，即使同步模块是在`wr_ptr_gray`跳变的时刻进行采集，其采集到的所有可能值也只有两个，一个是跳变之前的值，一个是跳变之后的值，它们只相差1，最坏情况也只是产生了“虚空”信号，而这不会引起错误传输。

和读空标志位产生机制一样，写满标志位也是通过比较读写地址产生的。和读空标志产生一样，写满标志也是读写指针相同时产生。但是如果地址的宽度和FIFO实际深度所需的宽度相等，某一时刻读写地址相同了，那FIFO是空还是满就难以判断了。所以读写指针需要增加一位来标记写地址是否超前读地址（在系统正确工作的前提下，读地址不可能超前于写地址），比如FIFO的深度为8，我们需要用宽度为4的指针。

如果读指针的最高位为0，而写指针的最高位为1，说明写指针超前于读指针，这时如果读写指针指向同一存储空间则可判断为FIFO被写满。写满标志位产生逻辑只需关心格雷码指针最高位不同（写超前于读）且它们指向同一存储空间的情况。

4 设计

4.1 双口 RAM 设计

RAM 端口参数可配置，读写位宽可以不一致。建议 memory 数组定义时，以长位宽地址、短位宽数据的参数为参考，方便数组变量进行选择访问。Verilog 描述如下。

```
1 module ramdp
2     #( parameter      AWI      = 10 ,
3         parameter      AWO      = 10 ,
4         parameter      DWI      = 10 ,
5         parameter      DWO      = 10
6     )
7     (
8         input           CLK_WR , //写时钟
9         input           WR_EN ,  //写使能
10        input [AWI-1:0] ADDR_WR ,//写地址
11        input [DWI-1:0] D ,      //写数据
12        input           CLK_RD , //读时钟
13        input           RD_EN ,  //读使能
14        input [AWO-1:0] ADDR_RD ,//读地址
15        output reg [DWO-1:0] Q   //读数据
16    );
17    //输出位宽大于输入位宽，求取扩大的倍数及对应的位数
18    parameter      EXTENT      = DWO/DWI ;
19    parameter      EXTENT_BIT   = AWI-AWO > 0 ? AWI-AWO : 'b1 ;
20    //输入位宽大于输出位宽，求取缩小的倍数及对应的位数
21    parameter      SHRINK      = DWI/DWO ;
22    parameter      SHRINK_BIT   = AWO-AWI > 0 ? AWO-AWI : 'b1;
23
24    genvar i ;
25    generate
26        //数据位宽展宽（地址位宽缩小）
27        if (DWO >= DWI) begin
28            //写逻辑，每时钟写一次
29            reg [DWI-1:0] mem [(1<<AWI)-1 : 0] ;
30            always @(posedge CLK_WR) begin
31                if (WR_EN) begin
32                    mem[ADDR_WR] <= D ;
33                end
34            end
35        end
```

```

36         //读逻辑, 每时钟读 4 次
37         for (i=0; i<EXTENT; i=i+1) begin
38             always @(posedge CLK_RD) begin
39                 if (RD_EN) begin
40                     Q[(i+1)*DWI-1: i*DWI]  <= mem[(ADDR_RD*EXTENT) + i ] ;
41                 end
42             end
43         end
44     end
45
46     //=====
47     //数据位宽缩小(地址位宽展宽)
48     else begin
49         //写逻辑, 每时钟写 4 次
50         reg [DWO-1:0]          mem [(1<<AWO)-1 : 0] ;
51         for (i=0; i<SHRINK; i=i+1) begin
52             always @(posedge CLK_WR) begin
53                 if (WR_EN) begin
54                     mem[(ADDR_WR*SHRINK)+i]  <= D[(i+1)*DWO -1: i*DWO] ;
55                 end
56             end
57         end
58
59         //读逻辑, 每时钟读 1 次
60         always @(posedge CLK_RD) begin
61             if (RD_EN) begin
62                 Q <= mem[ADDR_RD] ;
63             end
64         end
65     end
66     endgenerate
67
68 endmodule

```

计数器设计

计数器用于产生读写地址信息, 位宽可配置, 不需要设置结束值, 让其溢出后自动重新计数即可。Verilog 描述如下。

```

1 module  ccnt #(parameter W = 11 )
2     (
3         input          rstn ,
4         input          clk ,

```

```

5      input                en ,
6      output [W-1:0]      count
7  );
8
9      reg [W-1:0]          count_r ;
10     always @(posedge clk or negedge rstn) begin
11         if (!rstn) begin
12             count_r      <= 'b0 ;
13         end
14         else if (en) begin
15             count_r      <= count_r + 1'b1 ;
16         end
17     end
18     assign count = count_r ;
19
20 endmodule

```

4.2 FIFO 设计

该模块为 FIFO 的主体部分，产生读写控制逻辑，并产生空、满、可编程满状态信号。这里只给出读数据位宽大于写数据位宽的逻辑代码。

```

1  module  fifo
2      #(  parameter      AWI          = 10 ,
3          parameter      AWO          = 10 ,
4          parameter      DWI          = 16 ,
5          parameter      DWO          = 16 ,
6          parameter      PROG_DEPTH = 1024) //可设置深度
7      (
8          input           rstn,      //读写使用一个复位
9          input           wclk,      //写时钟
10         input           winc,      //写使能
11         input [DWI-1: 0] wdata,    //写数据
12
13         input           rclk,      //读时钟
14         input           rinc,      //读使能
15         output [DWO-1 : 0] rdata,  //读数据
16
17         output          wfull,     //写满标志
18         output          rempty,    //读空标志
19         output          prog_full  //可编程满标志
20     );

```

```

21
22 //输出位宽大于输入位宽，求取扩大的倍数及对应的位数
23 parameter          EXTENT          = DWO/DWI ;
24 parameter          EXTENT_BIT      = AWI-AWO ;
25 //输出位宽小于输入位宽，求取缩小的倍数及对应的位数
26 parameter          SHRINK          = DWI/DWO ;
27 parameter          SHRINK_BIT      = AWO-AWI ;
28
29 //===== push/wr counter =====
30 wire [AWI-1:0]      waddr ;
31 wire                wover_flag ; //多使用一位做写地址拓展
32 ccnt                #(.W(AWI+1))
33 u_push_cnt(
34     .rstn            (rstn),
35     .clk             (wclk),
36     .en              (winc && !wfull), //full 时禁止写
37     .count           ({wover_flag, waddr})
38 );
39
40 //===== pop/rd counter =====
41 wire [AWO-1:0]      raddr ;
42 wire                rover_flag ; //多使用一位做读地址拓展
43 ccnt                #(.W(AWO+1))
44 u_pop_cnt(
45     .rstn            (rstn),
46     .clk             (rclk),
47     .en              (rinc & !rempty), //empty 时禁止读
48     .count           ({rover_flag, raddr})
49 );
50
51 //=====
52 //窄数据进，宽数据出
53 generate
54     if (DWO >= DWI) begin : EXTENT_WIDTH
55
56         //格雷码转换
57         wire [AWI:0] wptr      = ({wover_flag, waddr}>>1) ^ ({wover_flag, waddr})
58 ;
59
60         //将写数据指针同步到读时钟域
61         reg [AWI:0]  rq2_wptr_r0 ;
62         reg [AWI:0]  rq2_wptr_r1 ;
63         always @(posedge rclk or negedge rstn) begin

```

```

62         if (!rstn) begin
63             rq2_wptr_r0    <= 'b0 ;
64             rq2_wptr_r1    <= 'b0 ;
65         end
66         else begin
67             rq2_wptr_r0    <= wptr ;
68             rq2_wptr_r1    <= rq2_wptr_r0 ;
69         end
70     end
71
72     //格雷码转换
73     wire [AWI-1:0] raddr_ex = raddr << EXTENT_BIT ;
74     wire [AWI:0]   rptr      = ({rover_flag, raddr_ex}>>1) ^ ({rover_flag,
raddr_ex}) ;
75     //将读数据指针同步到写时钟域
76     reg [AWI:0]    wq2_rptr_r0 ;
77     reg [AWI:0]    wq2_rptr_r1 ;
78     always @(posedge wclk or negedge rstn) begin
79         if (!rstn) begin
80             wq2_rptr_r0    <= 'b0 ;
81             wq2_rptr_r1    <= 'b0 ;
82         end
83         else begin
84             wq2_rptr_r0    <= rptr ;
85             wq2_rptr_r1    <= wq2_rptr_r0 ;
86         end
87     end
88
89     //格雷码反解码
90     //如果只需要空、满状态信号，则不需要反解码
91     //因为可编程满状态信号的存在，地址反解码后便于比较
92     reg [AWI:0]    wq2_rptr_decode ;
93     reg [AWI:0]    rq2_wptr_decode ;
94     integer        i ;
95     always @(*) begin
96         wq2_rptr_decode[AWI] = wq2_rptr_r1[AWI];
97         for (i=AWI-1; i>=0; i=i-1) begin
98             wq2_rptr_decode[i] = wq2_rptr_decode[i+1] ^ wq2_rptr_r1[i] ;
99         end
100     end
101     always @(*) begin
102         rq2_wptr_decode[AWI] = rq2_wptr_r1[AWI];

```

```

103         for (i=AWI-1; i>=0; i=i-1) begin
104             rq2_wptr_decode[i] = rq2_wptr_decode[i+1] ^ rq2_wptr_r1[i] ;
105         end
106     end
107
108     //读写地址、拓展位完全相同是，为空状态
109     assign rempty    = (rover_flag == rq2_wptr_decode[AWI]) &&
110                     (raddr_ex >= rq2_wptr_decode[AWI-1:0]);
111     //读写地址相同、拓展位不同，为满状态
112     assign wfull     = (wover_flag != wq2_rptr_decode[AWI]) &&
113                     (waddr >= wq2_rptr_decode[AWI-1:0]) ;
114     //拓展位一样时，写地址必然不小于读地址
115     //拓展位不同时，写地址部分比如小于读地址，实际写地址要增加一个FIFO深度
116     assign prog_full = (wover_flag == wq2_rptr_decode[AWI]) ?
117                     waddr - wq2_rptr_decode[AWI-1:0] >= PROG_DEPTH-1 :
118                     waddr + (1<<AWI) - wq2_rptr_decode[AWI-1:0] >=
119     PROG_DEPTH-1;
120
121     //双口 ram 例化
122     ramdp
123     # ( .AWI      (AWI),
124         .AWO      (AWO),
125         .DWI      (DWI),
126         .DWO      (DWO))
127     u_ramdp
128     (
129         .CLK_WR      (wclk),
130         .WR_EN       (winc & ! wfull), //写满时禁止写
131         .ADDR_WR     (waddr),
132         .D           (wdata[DWI-1:0]),
133         .CLK_RD      (rclk),
134         .RD_EN       (rinc & !rempty), //读空时禁止读
135         .ADDR_RD     (raddr),
136         .Q           (rdata[DWO-1:0])
137     );
138 end
139
140 //=====
141 //big in and small out
142 /*
143 else begin: SHRINK_WIDTH

```



```

144         .....
145     end
146     */
147 endgenerate
148 endmodule

```

4.3 FIFO 调用

下面可以调用设计的 FIFO，完成多位宽数据传输的异步处理。

```

1  module  fifo_s2b(
2      input                rstn,
3      input  [4-1: 0]      din,      //异步写数据
4      input                din_clk,  //异步写时钟
5      input                din_en,   //异步写使能
6
7      output [16-1 : 0]     dout,     //同步后数据
8      input                dout_clk,  //同步使用时钟
9      input                dout_en ); //同步数据使能
10
11  wire      fifo_empty, fifo_full, prog_full ;
12  wire      rd_en_wir ;
13  wire [15:0]  dout_wir ;
14
15  //读空状态时禁止读，否则一直读
16  assign rd_en_wir      = fifo_empty ? 1'b0 : 1'b1 ;
17
18  fifo  #(.AWI(10), .AWO(10), .DWI(16), .DWO(16), .PROG_DEPTH(1024))
19      u_buf_s2b(
20          .rstn            (rstn),
21          .wclk             (din_clk),
22          .winc             (din_en),
23          .wdata            (din),
24
25          .rclk             (dout_clk),
26          .rinc             (rd_en_wir),
27          .rdata            (dout_wir),
28
29          .wfull            (fifo_full),
30          .rempty           (fifo_empty),
31          .prog_full        (prog_full));
32

```

```

33 //缓存同步后的数据和使能
34 reg          dout_en_r ;
35 always @(posedge dout_clk or negedge rstn) begin
36     if (!rstn) begin
37         dout_en_r      <= 1'b0 ;
38     end
39     else begin
40         dout_en_r      <= rd_en_wir ;
41     end
42 end
43 assign      dout      = dout_wir ;
44 assign      dout_en    = dout_en_r ;
45 endmodule

```

4.4 testbench

```

1  `timescale 1ns/1ns
2  `define      SMALL2BIG
3  module test ;
4
5  `ifdef SMALL2BIG
6      reg          rstn ;
7      reg          clk_slow, clk_fast ;
8      reg [15:0]   din ;
9      reg          din_en ;
10     wire [15:0]   dout ;
11     wire          dout_en ;
12
13     //reset
14     initial begin
15         clk_slow  = 0 ;
16         clk_fast  = 0 ;
17         rstn      = 0 ;
18         #50 rstn  = 1 ;
19     end
20
21     //读时钟 clock_slow 较快于写时钟 clk_fast 的 1/4
22     //保证读数据稍快于写数据
23     parameter CYCLE_WR = 40 ;
24     always #(CYCLE_WR/2/4) clk_fast = ~clk_fast ;
25     always #(CYCLE_WR/2-1) clk_slow = ~clk_slow ;

```

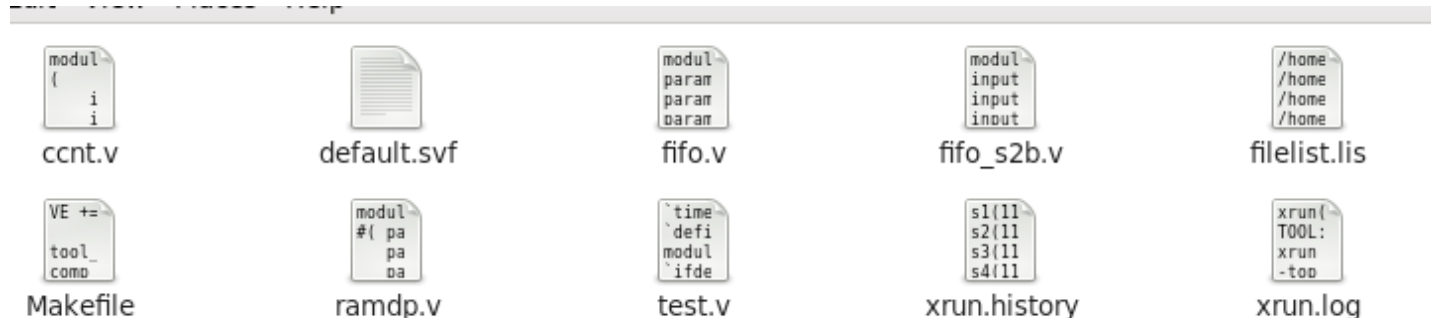
```

26
27 //data generate
28 initial begin
29     din      = 16'h0000 ;
30     din_en   = 1'b0 ;
31     wait (rstn) ;
32     repeat(1500) begin
33         @(negedge clk_fast) ;
34         din_en = 1'b1 ;
35         din    = din+16'h0001;
36         #10;
37     end
38
39 end
40 fifo_s2b u_data_buf2(
41     .rstn      (rstn),
42     .din       (din),
43     .din_clk   (clk_fast),
44     .din_en    (din_en),
45
46     .dout      (dout),
47     .dout_clk  (clk_slow),
48     .dout_en   (dout_en));
49 `else
50 `endif
51 //stop sim
52 initial begin
53     forever begin
54         #100;
55         if ($time >= 55000) $finish ;
56     end
57 end
58
59 initial begin
60     $shm_open("test.shm");
61     $shm_probe("AC");
62 endmodule

```

5 仿真实验

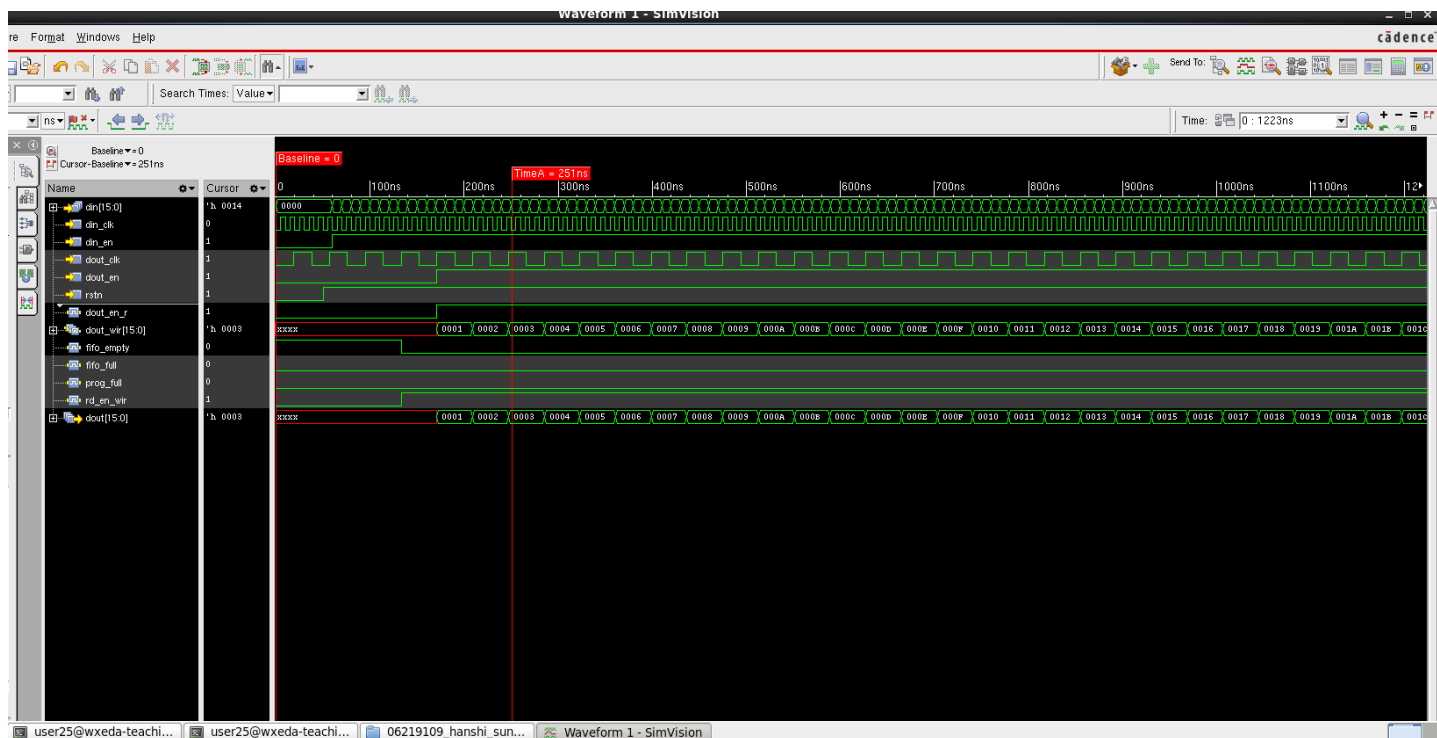
文件结构:



首先用 `make xrun` 编译debug

```
user25@wxeda-teaching:/home/teaching/user25/Desktop/06219109_hanshi_sun_FinalWork
file Edit View Search Terminal Help
root@wxeda-teaching 06219109 hanshi sun FinalWork]# make xrun
run -top test -access +rwc -f filelist.lis -timescale '1ns/1ns'
JOL: xrun 19.03-s001: Started on Dec 16, 2022 at 10:39:46 CST
JOL: xrun(64) 19.03-s001: Started on Dec 16, 2022 at 10:39:46 CST
run(64): 19.03-s001: (c) Copyright 1995-2019 Cadence Design Systems, Inc.
ile: /home/teaching/user25/Desktop/06219109_hanshi_sun_FinalWork/ccnt.v
module worklib.ccnt:v
errors: 0, warnings: 0
ile: /home/teaching/user25/Desktop/06219109_hanshi_sun_FinalWork/fifo_s2b.v
module worklib.fifo_s2b:v
errors: 0, warnings: 0
ile: /home/teaching/user25/Desktop/06219109_hanshi_sun_FinalWork/ramdp.v
module worklib.ramdp:v
errors: 0, warnings: 0
ile: /home/teaching/user25/Desktop/06219109_hanshi_sun_FinalWork/test.v
module worklib.test:v
errors: 0, warnings: 0
ile: /home/teaching/user25/Desktop/06219109_hanshi_sun_FinalWork/fifo.v
module worklib.fifo:v
errors: 0, warnings: 0
Elaborating the design hierarchy:
Caching library 'worklib' ..... Done
Top level design units:
test
Building instance overlay tables: ..... Done
Generating native compiled code:
worklib.ccnt:v <0x252f39e4>
streams: 2, words: 636
worklib.ccnt:v <0x12073667>
streams: 2, words: 636
worklib.fifo:v <0x3bc5f994>
streams: 20, words: 5832
worklib.fifo_s2b:v <0x78f55bcf>
streams: 3, words: 531
worklib.test:v <0x5870cd69>
streams: 11, words: 5050
worklib.ramdp:v <0x426faf5a>
streams: 4, words: 1649
Building instance specific data structures.
Loading native compiled code: ..... Done
Design hierarchy summary:
          Instances Unique
Modules:          6      5
Registers:        17     24
Scalar wires:      14      -
Expanded wires:    22      2
```

之后用 `make sim` 来看仿真看波形:



6 设计总结

本实验从理论原理，到代码编写debug，到实验仿真都做成功了。介绍FIFO 的定义，工作原理并对其功能实现进行具体操作和演示。通过本次设计为以后数字电路设计中处理连续数据流和信号同步等问题打下良好基础。