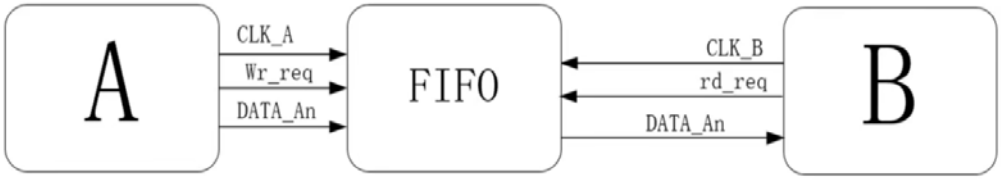
1. FIFO 简介

FIFO（First In First Out，即先入先出），是一种数据缓冲器，用来实现数据先入先出的读写方式。FIFO 存储器主要是作为缓存，应用在同步时钟系统和异步时钟系统中，在很多的设计中都会使用，如多比特数据做跨时钟域处理，前后带宽不同步等都用到了 FIFO。FIFO 本质上是由 RAM 加读写控制逻辑构成的一种先进先出的数据缓冲器。



图一：跨时钟域处理

图二：前后带宽不同步处理

Figure 1 跨时钟域与前后带宽不同步处理

1.1. FIFO 分类

同步 FIFO（sync_fifo），读写时钟相同。它的作用一般是做交互数据的一个缓冲，也就是说它的主要作用就是一个 buffer。

异步 FIFO（async_fifo），读写时钟不同，它有两个主要的作用，一个是实现数据在不同时钟域进行传递，另一个作用就是实现不同数据宽度的数据接口。异步 FIFO 的实现其实本质上和双口 RAM 是一样的，其实现思路就是将数据在 wr_clk 的时钟下写入自己设定大小的 ram 中，然后通过读时钟 rd_clk 从 ram 中将数据读出来即可。

1.2. 异步 FIFO 的主要参数

1） 位宽：用参数 DATA_WIDTH 表示，也就是 FIFO 存储的数据位宽（例如 DATA_WIDTH=8，能表示 0~255 的数）；

2） 指针位宽：用参数 PTR_WIDTH 表示，例如指针地址位宽为 8，地址位宽为 9（定义 9 为进位），FIFO 深度则为 2^8=256；

3） 深度：用参数 DATA_DEPTH 表示，也就是地址的大小，也就是说能存储多少个数据，由 PTR_WIDTH 计算得到；

4） 满标志：full，当 FIFO 中的数据满了以后将不再能进行数据的写入；

5） 将满标志：almost_full，当 FIFO 中的数据将要满了的时候将满标志跳变为 1，将满跳变的间隔阈值大小由 ALMOST_FULL_GAP 决定；

6） 空标志：empty，当 FIFO 为空的时候将不能进行数据的读出；

7） 将空标志：almost_empty，当 FIFO 中的数据将要空了的时候将满标志跳变为 1，将空跳变的间隔阈值大小由 ALMOST_EMPTY_GAP 决定；

8） 写地址：wr_addr，由自动加一生成，将数据写入该地址；

9） 读地址：rd_addr，由自动加一生成，将该地址上的数据读出；

10） 时钟：wr_clk，rd_clk，读写应用不同的时钟；

11） 复位信号：rd_rst_n，wr_rst_n，复位信号低电平有效；

12） 指针：wr_ptr，rd_ptr，比 rd_addr 多出来一位作为空满判断位。wr_ptr_gray，rd_ptr_gray，用格雷码指针来判断空满标识。wr_almost_ptr，rd_almost_ptr，用二进制码指针来判断将空将满标识；

13） 同步指针：r2w_r_ptr_gray 为读时钟域同步到写时钟域的读指针的格雷码，w2r_w_ptr_gray 为写时钟域同步到读时钟域的写指针的格雷码，指针的同步操作，用来做对比产生空满标志符；

1.3. 异步 FIFO 逻辑图

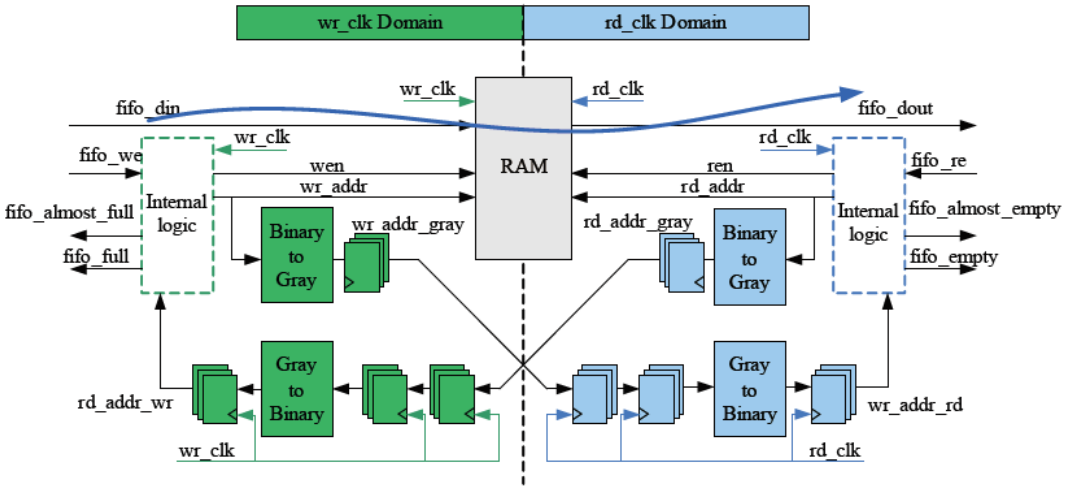


Figure 2 异步 FIFO 的逻辑图

1.4. 异步 FIFO 的模块划分

异步 FIFO 将模块划分为 4 个部分，fifo_mem（RAM）、wr_full、rd_empty、sync（synchronization）。RAM 模块根据读写地址进行数据的写入和读出，wr_full 模块根据 wr_clk 产生写地址，full 满信号以及 almost_full 将满信号，rd_empty 模块根 rd_clk 产生读地址，empty 空信号以及 almost_empty 将空信号，sync 模块用于同步 wr_ptr_gray 到读时钟域或者同步 rd_ptrr_gray 到写时钟域。

2. 异步 FIFO 设计关键技术问题

异步 FIFO 设计过程中存在两个关键技术问题：1）亚稳态；2）空/满状态

标志位判断及产生。在处理空/满标志位问题上，目前最常用的方案是增加一位读写指针附加位，当读写指针最高位相同其余位也相同时，认为读空，当读写指针最高位不相同其余位相同时，认为写满。文中以宽度为 16 位、深度为 8 位的异步 FIFO 为例，介绍亚稳态产生的原因以及降低亚稳态出现概率的方法，分析利用格雷码和同步转换来产生空/满标志位的方法。

2.1. 亚稳态产生的原因及解决方法

在所有数字器件中，寄存器都定义了一个信号时序要求，满足时序要求的寄存器才能正确地在输入端获取数据、在输出端产生数据。为确保操作可靠，输入数据在时钟沿之前必须稳定一段时间（建立时间），并且在时钟沿之后保持一段时间（保持时间），触发器经过一个特定时钟至输出延时后有效。如果一个数据信号在变化之前不满足触发器建立和保持时间要求，触发器输出可能会进入亚稳态。亚稳态触发器输出值会在高低电平之间徘徊不定，如下图第二个采样时钟来到时所示。

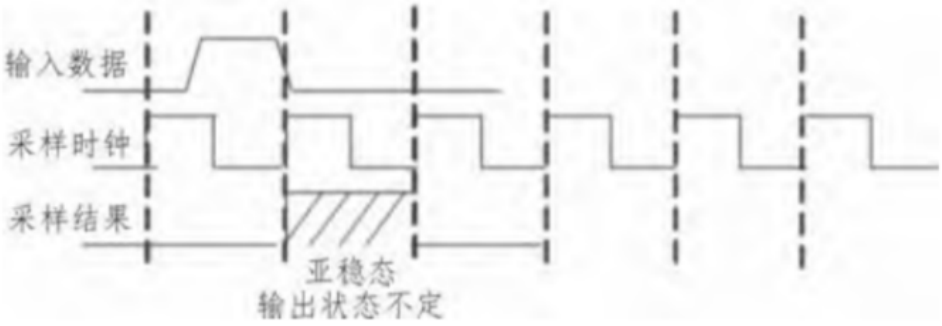


Figure 3 亚稳态产生的示意图

虽然亚稳态在异步电路中无法避免，但是 1）两级同步器和 2）格雷码计数器可以降低亚稳态概率到可以接受的程度。

2.1.1. 两级同步寄存器

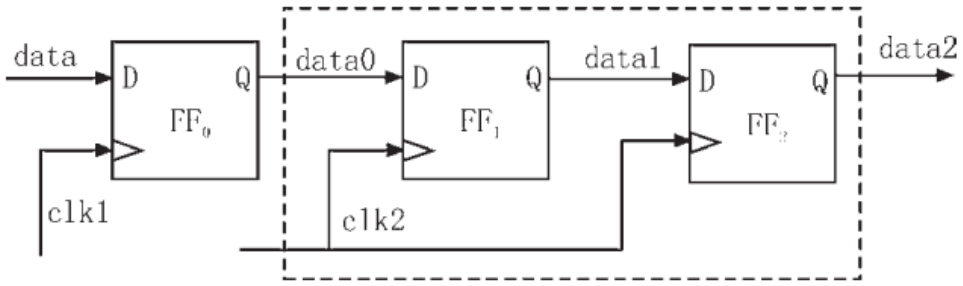


Figure 4 异步电路的同步化处理

图中两级寄存器对不同时钟域的输入数据锁存两拍。一般情况下，两级锁存同步器是一级同步器出现亚稳态概率的平方，在大部分同步设计中，两级同步器可以大大降低亚稳态出现概率，下图为通过两级寄存器消除亚稳态的示例图（当 clk1 和 clk2 上升沿很近时，data0 在变化时，此时 clk2 上升沿采集到一个正在变化的数值，data1 是个不确定值，FF1 触发器输出处于亚稳态，经过 1 个时钟延时，data1 值趋于稳定，FF2 在 clk2 上升沿对 data1 稳定值采样，输出 data2 为确定值。虽然 data1 在被 clk2 上升沿采样时也有处于亚稳态的可能，但是这种概率很小，经过两级同步器能大大降低亚稳态概率）。

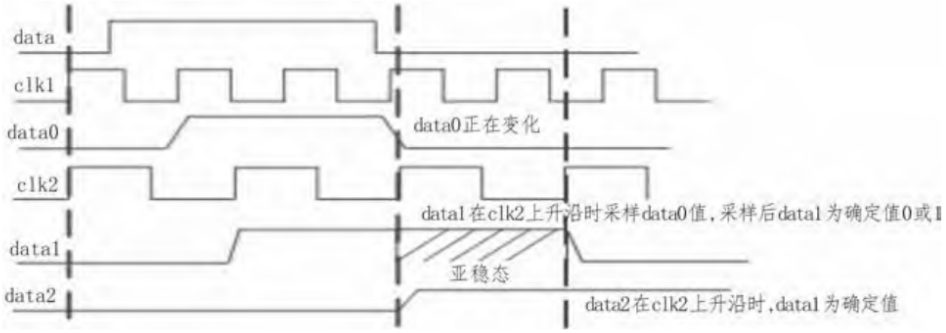


Figure 5 两级同步器消除亚稳态

两拍延时的数据同步对空满标志产生的影响：

由此信号 rd_ptr_gray 经过两级 D 触发器，就会有两拍的延时形成 r2w_r_ptr_gray 信号，所以在进行比较的时候就不是实时的 rd_ptr_gray 与 wr_ptr_gray 进行比较，而是两拍之前的 rd_ptr_gray 即 r2w_r_ptr_gray 与此刻的 wr_ptr_gray 进行比较。那么问题就来了这与我们的本意其实是不相符的，其实是这样的，这是一种最坏情况的考虑，将 r2w_r_ptr_gray 与 wr_ptr_gray 相比较是为了产生 full 信号，在用于数据同步的这两拍里面有可能再进行读操作，所以用于比较时的读地址一定小于或等于当前的读地址，就算此刻产生 full 信号，其实 FIFO 有可能还没有满。这也就为设计留了一些设计的余量。同理，就算有 empty 信号的产生，FIFO 有可能还有数据。这种留余量的设计在实际的工程项目中是很常见的。

2.1.2. 格雷码计数器

格雷码是一种误差最小化的可靠性编码，可以极大地减少由一个状态变化到下一个状态时电路产生的误差。这种编码方式是两个相邻码之间只有一位变化，缺点是格雷码是无权码，不能直接用于计算、比较，计算将满将空信号时需要转换为二进制代码计算。亚稳态出现的原因是数据变化时建立和保持时间不够，数据地址经过二级同步器后，地址指针采用格雷码编码，地址指针一次只能变化一位，通过这种可以有效减少亚稳态出现概率。下图为格雷码与二进制代码相互转换的算法及代码。

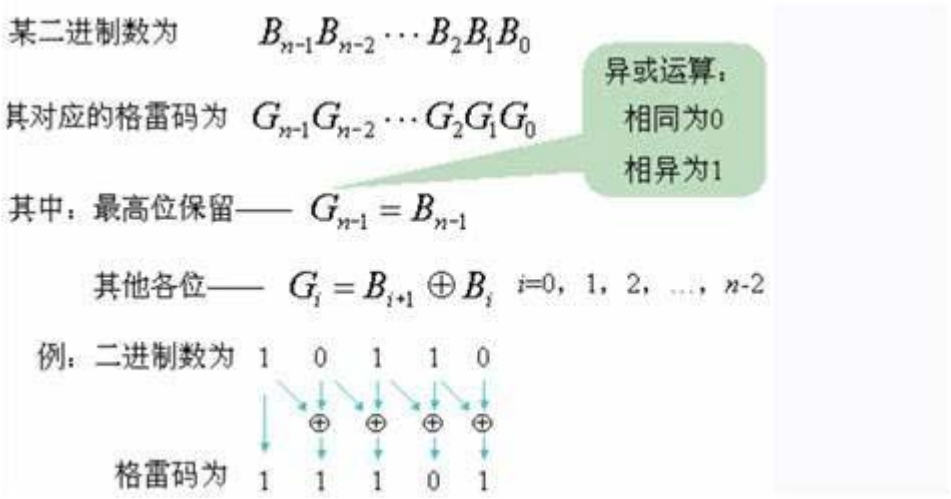


Figure 6 格雷码转二进制

代码：

```
//写指针格雷码转换
assign wr_ptr_gray = wr_ptr ^ (wr_ptr >> 1);
//读指针格雷码转换
assign rd_ptr_gray = rd_ptr ^ (rd_ptr>>1);
```

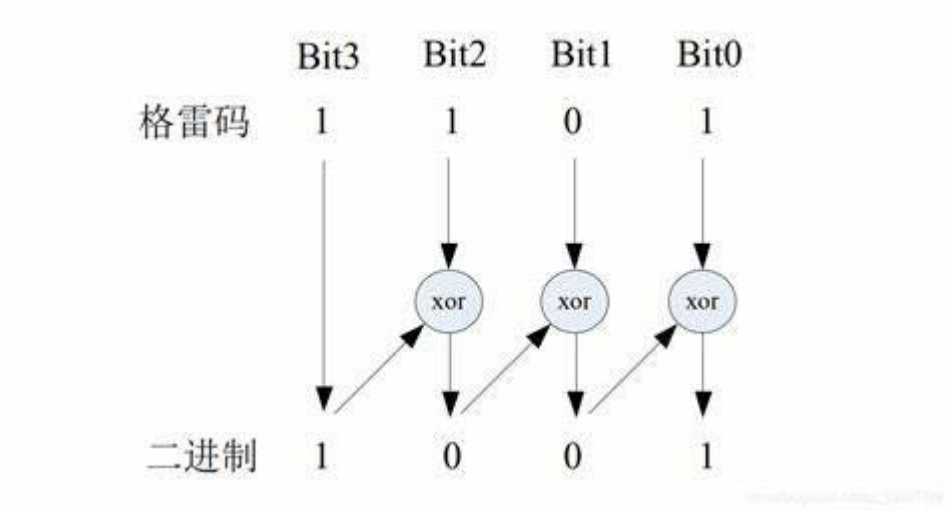


Figure 7 二进制转格雷码

代码：

```
//格雷码转二进制
assign rd_almost_ptr[PTR_WIDTH] = w2r_w_ptr_gray[PTR_WIDTH];
genvar i;
generate
    for(i=PTR_WIDTH-1;i>=0;i=i-1)begin:rdgray2bin
        assign rd_almost_ptr[i] = rd_almost_ptr[i+1] ^ w2r_w_ptr_gray[i];
    end
endgenerate

//格雷码转二进制
```

```
        assign wr_almost_ptr[PTR_WIDTH] = r2w_r_ptr_gray[PTR_WIDTH];
        genvar i;
        generate
            for (i=PTR_WIDTH-1;i>=0;i=i-1) begin:wrgray2bin   // <-- example block name
                assign wr_almost_ptr[i] = wr_almost_ptr[i+1] ^ r2w_r_ptr_gray[i];
            end
        endgenerate
```

在不同时钟域进行数据交换的时候我们一般采用格雷码的数据形式进行数据传递，这样能很大程度上降低出错的概率。

引入格雷码同时也引入一个问题，就是数据空满标志的判断不再是二进制时候的判断标准。

2.2. 空/满状态标志位判断及产生

我们知道 FIFO 的状态是满还是空，他们的相同的判断条件都是 wr_addr=rd_addr，但到底是空还是满我们还不能确定。在这里介绍一种方法来判断空满状态。我们设定一个指针 rd_ptr，wr_ptr，宽度为[PTR_WIDTH:0]，也就是说比传统的地址 rd_addr 和 wr_addr 多一位，我们就用这多出来的一位做空满判断。

1）如果是满状态的话，也就是说 wr_ptr 比 rd_ptr 多走了一圈，反应在二进制数值上就是 wr_ptr 和 rd_ptr 的最高位不相同，反应在格雷码上就是 wr_ptr_gray（rd_ptr_gray）和同步过来的 r2w_r_ptr_gray（w2r_w_ptr_gray）最高位和次高位不相同，其余位相同。

2）如果是空状态的话，也就是说 w_pointer_bin 和 r_pointer_bin 的路径相同，反应在二进制数值和格雷码上均为 wr_ptr 和 rd_ptr 的每一位相等。

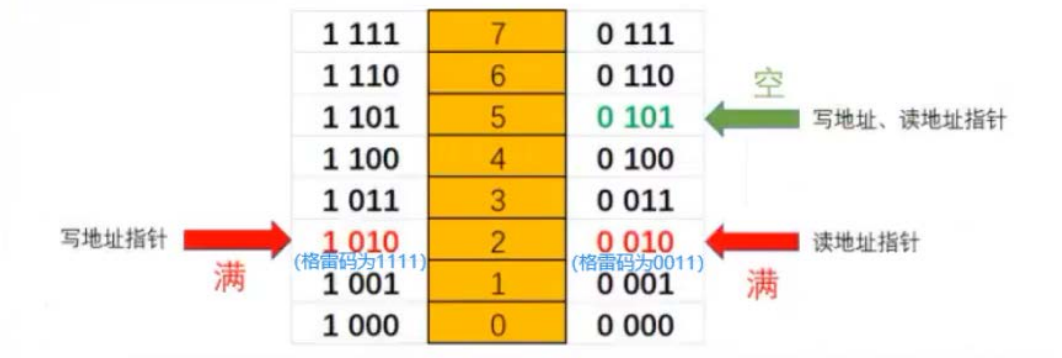


Figure 8 空/满状态标志位判断及产生

2.3. 将空/将满状态标志位判断及产生

将满（almost_full）和将空信号（almost_empty）实际上表示更加保守的满和空信号。基本思路是，设定一个间隔值 ALMOST_FULL_GAP（ALMOST_EMPTY_GAP），当读写地址之间的间隔小于或等于该间隔就产生将空或将满信号。

对于异步 FIFO 而言，由于同步过来的地址信号都是格雷码表示的，我们不能直接用格雷码去判断上述的这个间隔，所以需要先对接受到的格雷码进行解码变为二进制，再和当前时钟域下的另一个地址进行将满和将空的生成。

对于将空的判断和空一样，只需要检查写地址与读地址的差是否小于等于间隔

代码：

```
//产生将空信号  fifo_almost_empty
assign almost_empty = ((rd_almost_ptr - rd_ptr) <= ALMOST_EMPTY_GAP) ? 1'b1 : 1'b0;
```

对将满的判断则需要分两种情况：

1）最高位不同时：此时表示写地址有一个回卷，直接将读写地址除去符号位的部分做差与间隔比较。

2）最高位相同时：需要在差值上再加上 FIFO 深度（数据的总个数）。

代码：

```
//若最高位不相同则直接相减，最高位相同则需在此基础上加一个数据深度 DATA_DEPTH
assign wr_almost_val = (wr_almost_ptr[PTR_WIDTH] ^ wr_ptr[PTR_WIDTH]) ? wr_almost_ptr[PTR_WIDTH-1:0] - wr_ptr[PTR_WIDTH-1:0] : DATA_DEPTH + wr_almost_ptr - wr_ptr;

//产生将满信号
assign almost_full = (wr_almost_val <= ALMOST_FULL_GAP) ? 1'b1 : 1'b0;
```

3. 模块划分

异步 FIFO 将模块划分为 4 个部分，fifo_mem（RAM）、wr_full、rd_empty、sync（synchronization）。RAM 模块根据读写地址进行数据的写入和读出，wr_full 模块根据 wr_clk 产生写地址，full 满信号以及 almost_full 将满信号，rd_empty 模块根 rd_clk 产生读地址，empty 空信号以及 almost_empty 将空信号，sync 模块用于同步 wr_ptr_gray 到读时钟域或者同步 rd_ptrr_gray 到写时钟域。

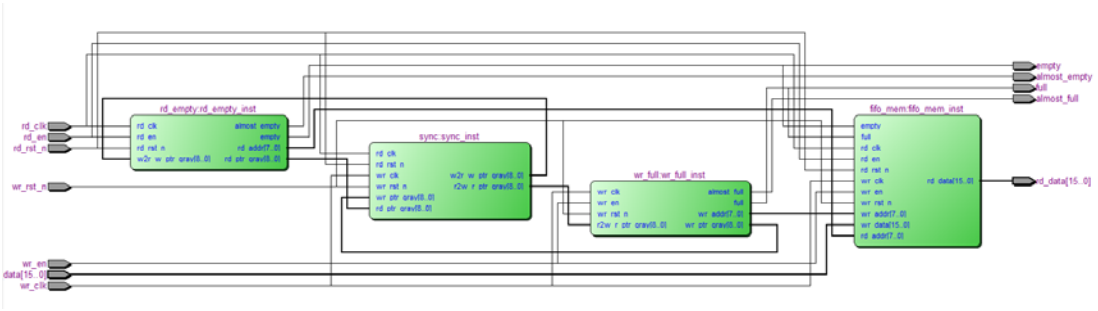


Figure 9　异步 FIFO 模块划分示意图

3.1. fifo_memory（RAM）模块

RAM 模块根据读写地址进行数据的写入和读出。

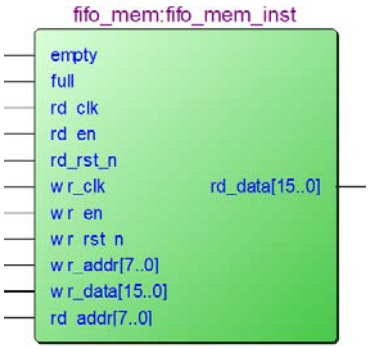


Figure 10 fifo_memory（RAM）模块

代码：

```
module fifo_mem #(
    parameter DATA_WIDTH = 16,          //RAM 存储器位宽
    parameter PTR_WIDTH    = 8          //指针地址位宽为 8，地址位宽为 9（定义 9 为进位），fifo 深度则为 2^8=256
    //parameter DATA_DEPTH = 256          //数据深度（数据个数）
)(
    input wr_clk,
    input wr_rst_n,
    input wr_en,
    input [PTR_WIDTH-1:0] wr_addr,    //写地址
    input [DATA_WIDTH-1:0] wr_data, //写数据
    input full,
    input rd_clk,
    input rd_rst_n,
    input rd_en,
    input [PTR_WIDTH-1:0] rd_addr,    //读地址
    input empty,

    output reg [DATA_WIDTH-1:0] rd_data //读数据
    );

    localparam DATA_DEPTH = 1 << PTR_WIDTH;//相当于将二进制数 1 在二进制中的表示向左移动 PTR_WIDTH 位，也就是将这个数乘以 2 的 PTR_WIDTH 次方

    //开辟存储空间
    reg [DATA_WIDTH-1:0] mem[DATA_DEPTH-1:0];

    integer i;
    always @(posedge wr_clk or negedge wr_rst_n) begin
        if (!wr_rst_n) begin
```

```
                for(i=0;i<DATA_DEPTH;i=i+1) begin
                    mem[i] <= 1'b0;
                end
            end
            else if(wr_en && !full) begin
                mem[wr_addr] <= wr_data;
            end
        end

        always @(posedge rd_clk or negedge rd_rst_n) begin
            if (!rd_rst_n) begin
                rd_data <= 'b0;
            end
            else if(rd_en && !empty) begin
                rd_data <= mem[rd_addr];
            end
        end
    endmodule
```

3.2. wr_full 模块

wr_full 模块根据 wr_clk 产生写地址，full 满信号以及 almost_full 将满信号。

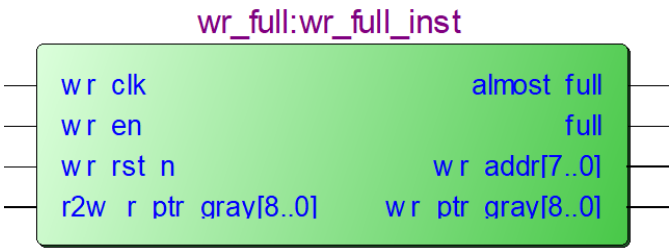


Figure 11 wr_full 模块

```
代码：
module wr_full #(
    parameter   DATA_WIDTH = 16,
    parameter   PTR_WIDTH  = 8,
    parameter   ALMOST_FULL_GAP = 3
)
(
    input wr_clk,
    input wr_rst_n,
    input wr_en,
    input [PTR_WIDTH:0] r2w_r_ptr_gray,
```

```
        output full,
        output [PTR_WIDTH-1:0] wr_addr,
        output [PTR_WIDTH:0] wr_ptr_gray,
        output wire almost_full
    );
        localparam DATA_DEPTH = 1 << PTR_WIDTH;//相当于将二进制数 1 在二进制中的表示向左移动 PTR_WIDTH 位，也就是将这个数乘以 2 的 PTR_WIDTH 次方
        reg [PTR_WIDTH:0] wr_ptr;
        wire [PTR_WIDTH:0] wr_almost_ptr;
        wire [PTR_WIDTH:0] wr_almost_val;

        //fifo_mem 的地址
        assign wr_addr = wr_ptr[PTR_WIDTH-1:0];
        //写指针格雷码转换
        assign wr_ptr_gray = wr_ptr ^ (wr_ptr >> 1);
        //产生 full，最高位和此高位不同而其他位均相同则判断为 full
        assign full =
({~wr_ptr_gray[PTR_WIDTH:PTR_WIDTH-1],wr_ptr_gray[PTR_WIDTH-2:0]} ==

{r2w_r_ptr_gray[PTR_WIDTH:PTR_WIDTH-1],r2w_r_ptr_gray[PTR_WIDTH-2:0]}) ?
1'b1:1'b0;
        //格雷码转二进制
        assign wr_almost_ptr[PTR_WIDTH] = r2w_r_ptr_gray[PTR_WIDTH];
        genvar i;
        generate
            for (i=PTR_WIDTH-1;i>=0;i=i-1) begin:wrgray2bin  // <-- example block name
                assign wr_almost_ptr[i] = wr_almost_ptr[i+1] ^ r2w_r_ptr_gray[i];
            end
        endgenerate

        //若最高位不相同则直接相减，最高位相同则需在此基础上加一个数据深度 DATA_DEPTH
        assign wr_almost_val = (wr_almost_ptr[PTR_WIDTH] ^ wr_ptr[PTR_WIDTH]) ?
wr_almost_ptr[PTR_WIDTH-1:0] - wr_ptr[PTR_WIDTH-1:0] : DATA_DEPTH +
wr_almost_ptr - wr_ptr;

        //产生将满信号
        assign almost_full = (wr_almost_val <= ALMOST_FULL_GAP) ? 1'b1 : 1'b0;
        //产生写指针
        always@(posedge wr_clk or negedge wr_rst_n)
            if (!wr_rst_n)begin
                wr_ptr <= 'b0;
                end
```

```
            else if (~full && wr_en)begin
                wr_ptr <= wr_ptr + 1'b1;
                end
        endmodule
```

3.3. rd_empty 模块

rd_empty 模块根据 rd_clk 产生读地址，empty 空信号以及 almost_empty 将空信号。

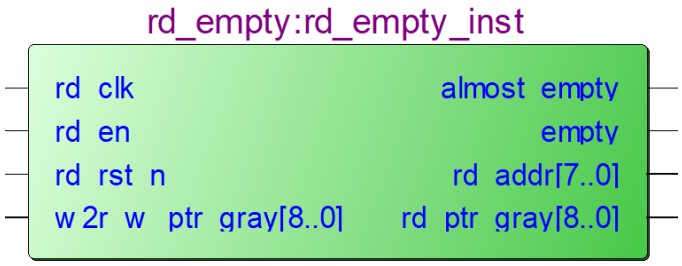


Figure 12 rd_empty 模块

代码：

```
module rd_empty #(
    parameter DATA_WIDTH = 16,
    parameter PTR_WIDTH  = 8,
    parameter ALMOST_EMPTY_GAP = 3   //将空信号间隔阈值为 3
)
(
    input   rd_clk,
    input   rd_rst_n,
    input   rd_en,
    input   [PTR_WIDTH:0]  w2r_w_ptr_gray,      //写时钟域同步到写时钟域的读指针的格雷码
    output [PTR_WIDTH-1:0] rd_addr,             //fifo_mem 的地址
    output [PTR_WIDTH:0] rd_ptr_gray,           //格雷码形式的读指针
    output wire empty,
    output wire almost_empty
);
    reg [PTR_WIDTH:0] rd_ptr; //比 rd_addr 多出来一位作为空满判断位
    wire [PTR_WIDTH:0] rd_almost_ptr; //用来判断将空信号的读指针


    //直接作为存储实体的地址，比如连接到 RAM 存储实体的读地址端
    assign rd_addr = rd_ptr[PTR_WIDTH-1:0];

    //读指针格雷码转换
    assign rd_ptr_gray = rd_ptr ^ (rd_ptr>>1);
```

```
        //产生读空信号 empty
        assign empty = (rd_ptr_gray == w2r_w_ptr_gray) ? 1'b1:1'b0;

        //格雷码转二进制
        assign rd_almost_ptr[PTR_WIDTH] = w2r_w_ptr_gray[PTR_WIDTH];
        genvar i;
        generate
            for(i=PTR_WIDTH-1;i>=0;i=i-1)begin:rdgray2bin    // <-- example block name
                assign rd_almost_ptr[i] = rd_almost_ptr[i+1] ^ w2r_w_ptr_gray[i];
            end
        endgenerate

        //产生将空信号 fifo_almost_empty
        assign almost_empty = ((rd_almost_ptr - rd_ptr) <= ALMOST_EMPTY_GAP) ? 1'b1 : 1'b0;

        //读指针产生
        always @(posedge rd_clk or negedge rd_rst_n) begin
            if (!rd_rst_n) begin
                rd_ptr <= 'b0;
            end
            else if(rd_en && !empty) begin
                rd_ptr <= rd_ptr + 1'b1;
            end
        end
        endmodule
```

3. 4. sync 模块

sync 模块用于同步 wr_ptr_gray 到读时钟域或者同步 rd_ptrr_gray 到写时钟域。

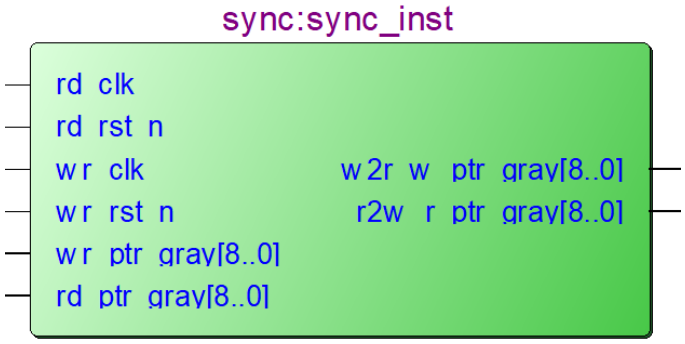


Figure 13 sync 模块

代码：

```
module sync #(
    parameter DATA_WIDTH = 16,
    parameter PTR_WIDTH    = 8
)(
    input    wr_clk,
    input    wr_rst_n,
    input    rd_clk,
    input    rd_rst_n,
    input    [PTR_WIDTH:0] wr_ptr_gray,
    input    [PTR_WIDTH:0] rd_ptr_gray,

    output wire [PTR_WIDTH:0] w2r_w_ptr_gray,
    output wire [PTR_WIDTH:0] r2w_r_ptr_gray
    );

    reg [PTR_WIDTH:0] wr_ptr_gray_r[1:0];
    reg [PTR_WIDTH:0] rd_ptr_gray_r[1:0];

    assign w2r_w_ptr_gray = wr_ptr_gray_r[1];
    assign r2w_r_ptr_gray = rd_ptr_gray_r[1];

    //写指针同步到读时钟域
    always @(posedge rd_clk or negedge rd_rst_n) begin
        if (!rd_rst_n) begin
            wr_ptr_gray_r[0] <= 'b0;
            wr_ptr_gray_r[1] <= 'b0;
        end
        else begin
            wr_ptr_gray_r[0] <= wr_ptr_gray;
            wr_ptr_gray_r[1] <= wr_ptr_gray_r[0];
        end
    end

    //读指针同步到写时钟域
    always @(posedge wr_clk or negedge wr_rst_n) begin
        if (!wr_rst_n) begin
            rd_ptr_gray_r[0] <= 'b0;
            rd_ptr_gray_r[1] <= 'b0;
        end
        else begin
            rd_ptr_gray_r[0] <= rd_ptr_gray;
            rd_ptr_gray_r[1] <= rd_ptr_gray_r[0];
        end
```

```
        end
endmodule
```

4. 仿真测试

4.1. TestBench 测试文件

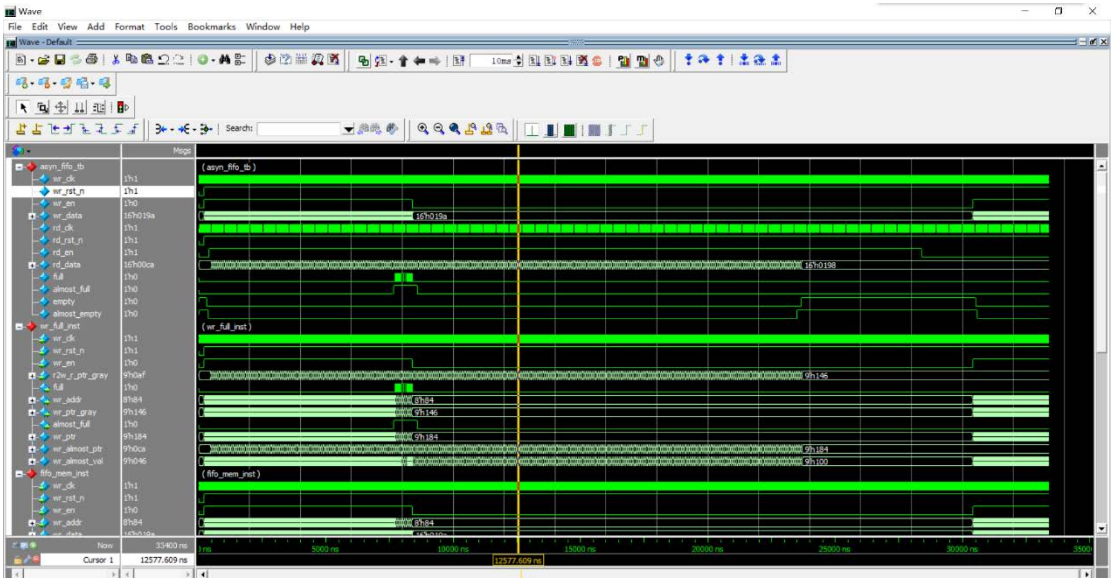
```
`timescale 1ns/1ns
module asyn_fifo_tb;
    localparam DATA_WIDTH = 16;

    reg wr_clk;
    reg wr_rst_n;
    reg wr_en;
    reg [DATA_WIDTH-1:0] wr_data;
    reg rd_clk;
    reg rd_rst_n;
    reg rd_en;

    wire [DATA_WIDTH-1:0] rd_data;
    wire full;
    wire almost_full;
    wire empty;
    wire almost_empty;

asyn_fifo #(
    .DATA_WIDTH(16),
    .PTR_WIDTH(8)
)
asyn_fifo_inst
(
    .wr_clk(wr_clk),
    .wr_rst_n(wr_rst_n),
    .wr_en(wr_en),
    .wr_data(wr_data),
    .rd_clk(rd_clk),
    .rd_rst_n(rd_rst_n),
    .rd_en(rd_en),

    .rd_data(rd_data),
    .full(full),
    .almost_full(almost_full),
```

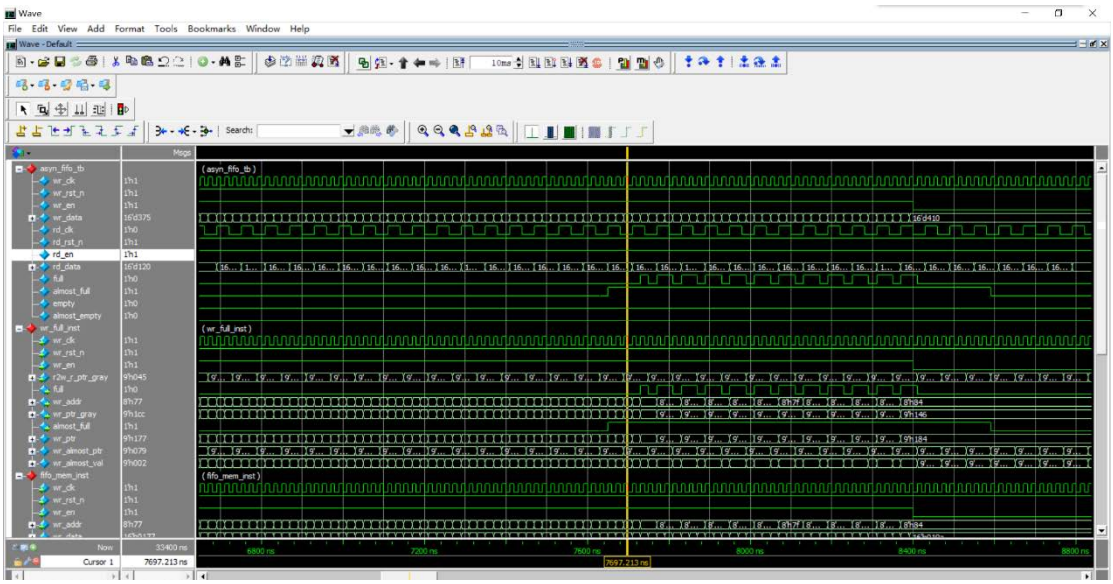
```
        .empty(empty),
        .almost_empty(almost_empty)
        );

initial wr_clk = 0;
always #10 wr_clk = ~wr_clk;

initial rd_clk = 0;
always #30 rd_clk = ~rd_clk;

//写数据
always @(posedge wr_clk or negedge wr_rst_n)
        begin
        if (!wr_rst_n)
                begin
                wr_data <= 1'b0;
                end
        else if(wr_en)
                begin
                wr_data <= wr_data + 1'b1;
                end
        else
                begin
                wr_data <= wr_data;
                end
        end

initial begin
        wr_rst_n = 0;
        rd_rst_n = 0;
        wr_en       = 0;
        rd_en       = 0;
        #200;
        wr_rst_n = 1;
        rd_rst_n = 1;
        wr_en       = 1;
        #200
        rd_en       = 1;
        #8000;
        wr_en       = 0;
        #20000
        rd_en       = 0;
        #2000
        wr_en       = 1;
```

```
        #3000
        $stop;
        End
    endmodule
```

4.2. Modelsim 仿真结果

仿真结果显示，在 wr_en 写使能有效的情况下，依次向 FIFO 写入数据，在 rd_en 读使能的情况下从 rd_data 能依次从 FIFO 读出数据。当数据达到设定深度时，full 判断为满，此时停止写入，当数据读完时，empty 判断为空，此时停止数据读出。与此同时达到设定的阈值 ALMOST_FULL_GAP 和 ALMOST_EMPTY_GAP 时将空将满信号也能正常输出。说明基本实现异步 FIFO 的功能。





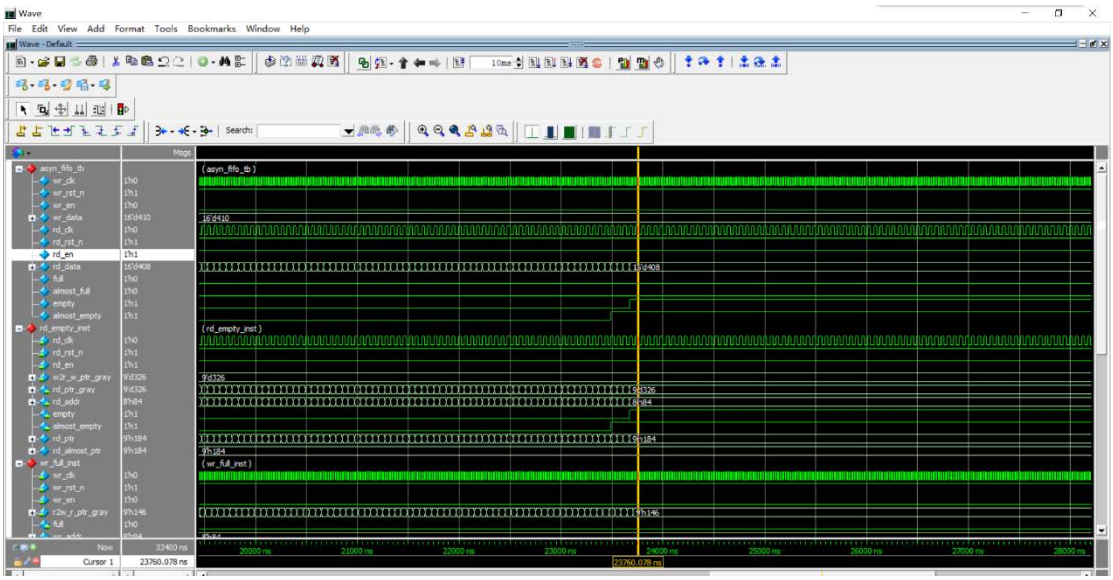


Figure 14　仿真测试图

5. 拓展：实际的系统中如何设计一个 FIFO 的深度

当写入频率小于读出频率时，FIFO 深度没有特殊限制，当写入频率大于读出频率时，需要 FIFO 缓存数据，FIFO 深度有要求，保证能把尚未读出的数据存储在 FIFO 中。

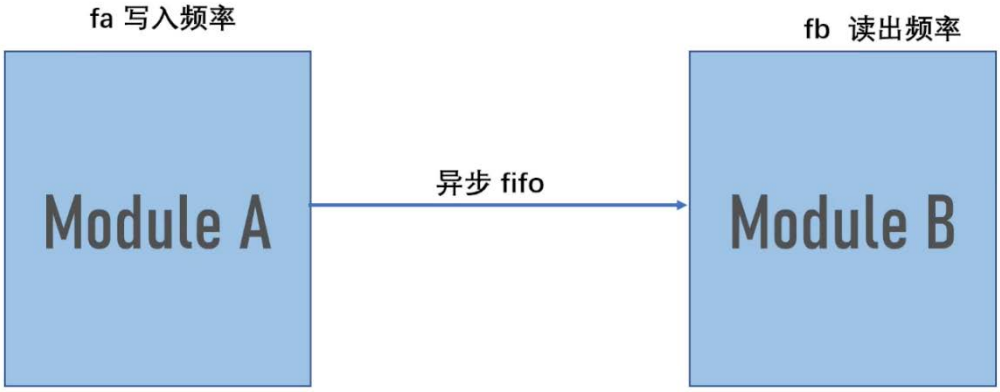


Figure 15　写入频率与读出频率

基本大原则： 写入的数据量=读出的数据量

FIFO不能用在连续数据流的缓存中。
例如:AD采样的结果连续不断的送入FIFO，深度只有无穷才能满足需求。

↓

在出现Burst的时候，凸显FIFO的功能与用途

Burst：**是指在同一行中相邻的存储单元连续进行数据传输的方式，连续传输的周期数就是突发长度（Burst Lengths，简称BL）**

发送方"突发"发送数据的时间T内，读不完所有的数据。
T时间后，FIFO内有足够的空间存储尚未被读走的数据，
在发送方不发数据的空闲时间内，将剩余的数据读走。 → FIFO深度的设置

如何计算

- fa>fb 基础情况
- fa>fb 读写存在idle cycle
- fa<fb 读写存在idle cycle
- fa<fb 随机读写情况

fa>fb 基础情况

Case:fa=80MHz，fb=50Mhz，每一次burst传送120个数据

写入120数据所用时间：
120/fa=1500ns
1500ns可以读出的数据：
1500*fb=75
仍未被读出数据：
120-75=45 -----FIFO最少深度

↓

格雷码比较时Delay的存在

FIFO最少深度>45

idle cycle：闲置循环
读写过程不是在每个周期都连续进行的，可能会存在某些个闲暇的循环。

↓

计算时候相当于变相减小了频率

换个说法：wr_en / re_en 占空比
计算方法相同

fa>fb 读写存在idle cycle

Case: fa=80MHz，fb=50Mhz，每一次burst传送120个数据
两次写之间idle cycle=1
两次读之间idle cycle=3

写入一次数据时间=2*(1/fa) ---40MHz(fa'
读出一次数据时间=4*(1/fb) ----12.5MHz(fb'

写入120数据所用时间：
120/fa'=3000ns
3000ns可以读出的数据：
3000*fb'=37.5
仍未被读出数据：
120-37.5=82.5->83 -----FIFO最少深度

fa<fb 读写存在idle cycle

Case: fa=50MHz，fb=60Mhz，每一次burst传送120个数据
两次写之间idle cycle=1
两次读之间idle cycle=2

写入一次数据时间=2*(1/fa) ---25MHz(fa'
读出一次数据时间=3*(1/fb) ----20MHz(fb'

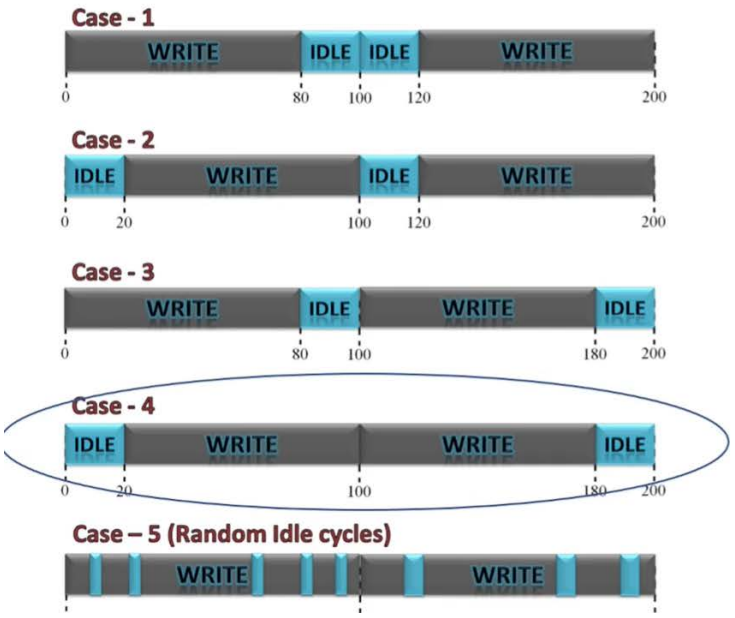
↓

相当于fa＞fb 计算方法同第一页

fa<fb 随机读写情况

Case:写入数据= 80 DATA/100 Clock
读出数据= 8 DATA/10 Clock. Burst数据量160

写频率80MHz,读频率50MHz



写入160数据所用时间:
160/fa=2000ns
2000ns可以读出的数据:
fb'=(1/50MHz)*10/8=40MHz
2000ns*fb=80
仍未被读出数据:
160-80=80 -----FIFO最少深度

→ 最差的情况：写最快，读最慢