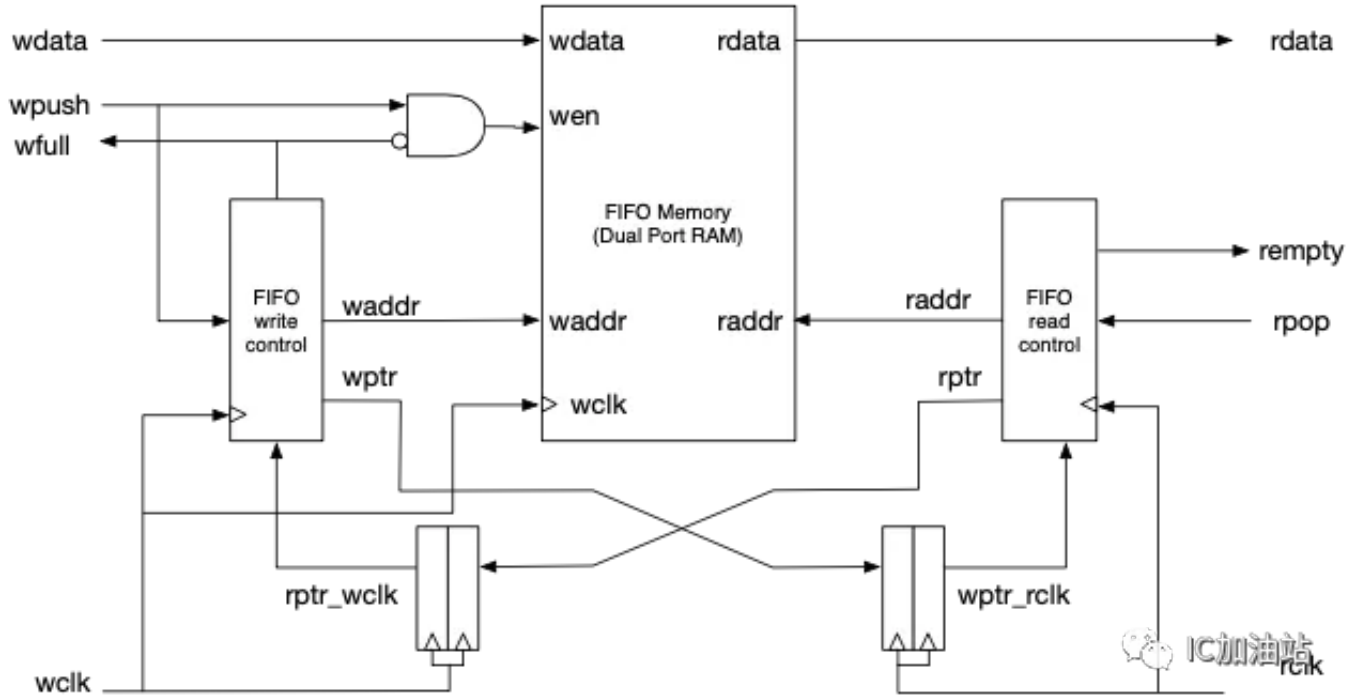
面试必杀技：异步FIFO（下）-- CDC的那些事（6）

原创 硅谷老李 IC加油站 6月22日

上一篇老李介绍了异步FIFO的基础部分，包括为什么用Gray Code来同步read pointer, write pointer。这一篇咱们从头一起过一遍异步FIFO的具体设计，然后再讨论几个常见的问题。

有的面试官可能上来让你先画异步FIFO的框图，老李建议大家自己手画一下，能够记住。



要注意，wptr和rptr都是gray code，在上一篇我们已经讨论过gray code是可以直接利用2flop synchronizer来同步的。而用来读写实际的memory必须是binary address，在FIFO write control和FIFO read control 里面我们进行binary to gray code的转换。

下面是上图中间fifo memory部分的简单实现

```
module fifomem #(parameter  DATASIZE = 8, // Memory data word width
                 parameter  ADDRSIZE = 4) // Number of mem address bits
        (output [DATASIZE-1:0] rdata,
         input  [DATASIZE-1:0] wdata,
         input  [ADDRSIZE-1:0] waddr, raddr,
         input                 wen, wfull, wclk);
        `ifdef VENDORRAM
         // instantiation of a vendor's dual-port RAM
         vendor_ram mem (.dout(rdata), .din(wdata),
                         .waddr(waddr), .raddr(raddr),
                         .wen(wen), .wclken_n(wfull), .clk(wclk));
        `else
         // RTL Verilog memory model
         localparam DEPTH = 1<<ADDRSIZE;
         reg [DATASIZE-1:0] mem [0:DEPTH-1];
         assign rdata = mem[raddr];
         always @(posedge wclk)
           if (wclken && !wfull) mem[waddr] <= wdata;
        `endif
endmodule
```


上面的code简化了rdata的逻辑，如果使用SRAM，可能需要加一级flop来存储SRAM读出来的值。

这里插一句，在设计异步FIFO或者使用异步FIFO的时候，需要计算清楚FIFO的深度，（如何FIFO的深度计算老李打算以后单独开一篇文章来讨论）然后要比较使用SRAM和flop array的cost。依据老李的经验，目前较新的7nm/5nm的工艺下，当存储位大于2k bit，使用vendor的compile memory在面积上开始划算起来，低于2k bit，使用flop array划算。这个部分需要大家在实际工作中自己去比较计算。

```
 module sync_r2w #(parameter ADDRSIZE = 4)
        (output reg [ADDRSIZE:0] rptr_wclk,
         input      [ADDRSIZE:0] rptr,
         input                   wclk, wrst_n);
        reg [ADDRSIZE:0] rptr_wclk;
//        always @(posedge wclk or negedge wrst_n)
//          if (!wrst_n) {wq2_rptr,wq1_rptr} <= 0;
//          else         {wq2_rptr,wq1_rptr} <= {wq1_rptr,rptr};
        sync_lib #(.STAGE(2), .WIDTH(4))
                 (.d    (rptr),
                  .q    (rptr_wclk),
                  .clk  (wclk),
                  .rstn (wrst_n)
                 );
endmodule
```


关于read pointer和write pointer 的同步很简单，用2flop synchronizer即可，老李这里要强调一下，大家在实际工作中不要自作聪明去用verilog的behaivor code去实现2flop syncronzier（上面注释掉的行），而是要直接例化现成的cdc库元件。只要你不是在创立不到一个月的创业公司，这种cdc library一定是你们公司已经有的，轮子已经造出来了，千万别自己再造。

下面我们再看一下write control部分的RTL实现。

```
module wptr_full  #(parameter ADDRSIZE = 4)
         (output reg                wfull,
          output     [ADDRSIZE-1:0] waddr,
          output reg [ADDRSIZE  :0] wptr,
          input      [ADDRSIZE  :0] rptr_wclk,
input
wpush, wclk, wrst_n);
reg  [ADDRSIZE:0] wbin;
wire [ADDRSIZE:0] wgraynext, wbinnext;
// GRAYSTYLE2 pointer
always @(posedge wclk or negedge wrst_n)
  if (!wrst_n) {wbin, wptr} <= 0;
  else         {wbin, wptr} <= {wbinnext, wgraynext};
// Memory write-address pointer (okay to use binary to address memory)
assign waddr = wbin[ADDRSIZE-1:0];
assign wbinnext  = wbin + (wpush & ~wfull);
assign wgraynext = (wbinnext>>1) ^ wbinnext;
//------------------------------------------------------------------
// Simplified version of the three necessary full-tests:
// assign wfull_val=((wgnext[ADDRSIZE] !=wq2_rptr[ADDRSIZE] ) &&
//                   (wgnext[ADDRSIZE-1]  !=wq2_rptr[ADDRSIZE-1]) &&
// (wgnext[ADDRSIZE-2:0]==wq2_rptr[ADDRSIZE-2:0]));
//------------------------------------------------------------------
assign wfull_val = (wgraynext=={~rptr_wclk[ADDRSIZE:ADDRSIZE-1],
                                rptr_wclk[ADDRSIZE-2:0]});
always @(posedge wclk or negedge wrst_n)
  if (!wrst_n) wfull  <= 1'b0;
    else
        wfull  <= wfull_val;
endmodule
```


这里的满的判断用到了我们上一篇讲的判断逻辑，即高两位相反，低位都相同。下面是read side判断空的逻辑。

```
module rptr_empty #(parameter ADDRSIZE = 4)
          (output reg                 rempty,
           output      [ADDRSIZE-1:0] raddr,
           output reg  [ADDRSIZE  :0] rptr,
           input       [ADDRSIZE  :0] wptr_rclk,
           input                      rpop, rclk, rrst_n);
   reg  [ADDRSIZE:0] rbin;
   wire [ADDRSIZE:0] rgraynext, rbinnext;
   //-------------------
   // GRAYSTYLE2 pointer
   //-------------------
   always @(posedge rclk or negedge rrst_n)
     if (!rrst_n) {rbin, rptr} <= 0;
     else         {rbin, rptr} <= {rbinnext, rgraynext};
   // Memory read-address pointer (okay to use binary to address memory)
   assign raddr     = rbin[ADDRSIZE-1:0];
   assign rbinnext  = rbin + (rpop & ~rempty);
   assign rgraynext = (rbinnext>>1) ^ rbinnext;
//---------------------------------------------------------------
// FIFO empty when the next rptr == synchronized wptr or on reset
//---------------------------------------------------------------
assign rempty_val = (rgraynext == wptr_rclk);
   always @(posedge rclk or negedge rrst_n)
     if (!rrst_n) rempty <= 1'b1;
     else         rempty <= rempty_val;
endmodule
```


下面我们来讨论几个面试中常见的问题。

问题：假设wclk速度比rclk快，那么当raddr+1，再同步到wclk后，如果这期间有了push操作，那会不会使得wptr超过了rptr，造成FIFO overflow呢?

回答：不会，当rptr在传过去之前，如果wptr已经追上了rptr-1，那么wfull已经是1了，FIFO是不允许在FIFO 满的时候进行push操作的（在实际工程中我们通常要利用assertion来check保证在wfull为1的时候push不能为1）。而如果这个时候有了pop操作，raddr+1，这个时候实际上FIFO有了一个free entry，但是push这一侧看到的FIFO依然是满的。这就是我们所说的异步FIFO的**假满**。相应的，FIFO的empty为1时，也可能FIFO此时有个push操作，导致FIFO为**假空**。假空和假满并不会影响FIFO的正确性，无非就是早一点告诉push side停止push，或者早一点告诉pop side停止pop，但是FIFO是不会产生overflow和underflow的。如果要说有什么缺点的话，就是在性能上有一些损失，当FIFO的深度很大的时候，这通常不是什么问题。

问题：如何判断FIFO是真空/真满呢?
回答：判断假空假满刚好相反，在push side我们来判断空，在pop side来判断满，为什么要这样留给大家自己思考。

问题：设计一个depth=1的异步FIFO

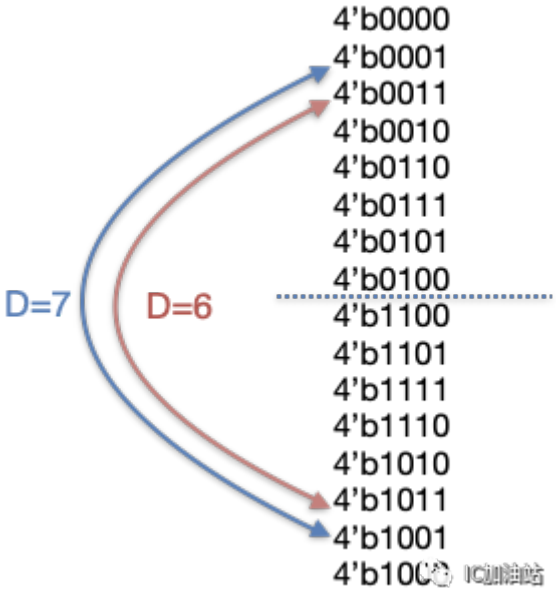
回答：这个大家就是要活学活用了，不能死板套用。只需要考虑一个问题，只有1个entry，那么需要几位的address 或者pointer呢？当然是1位就够了，那我们真的还需要一个pointer吗？因为只有一个entry，当一次push，FIFO就满了，一次pop，FIFO就空了。1个bit用来表示满和空就足够了。其实这样的FIFO我们已经见过了，老李在多bit信号跨时钟域怎么办？-- CDC的那些事（4）里面讲到的带反馈的asynchronous load其实就是depth=1的异步FIFO！

问题：如何设计depth不是2的幂次的异步FIFO？
回答：我们在上一讲里面看到的gray code，只有当depth=2的幂次个数的时候，才能做到wrap around时继续保持gray code的性质：即连续两个码之间只有1位不同。下面这个图是表示depth=8的时候我们利用16个gray code来表示pointer。

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

比如从4'd15到4'd0，也只有1位不同。但是如果不是2的幂次，比如DEPTH=7，那我们怎么样来利用Gray code呢？直接从4'b0000到4'b0101肯定是不行的，因为4'b0101变到4'b0000有两个bit发生了变化，这样我们就没法利用2flop synchronizer来同步了。解决这个办法的诀窍其实就是老李上一篇提到的gray code的第二个性质：gray code每一位是有个对称轴的。我们可以这样编码，addr==0的时候gray code不从4'b0000开始，而是从4'b0001开始，直到4'b1001来wrap around，这样从4'b1001->4'b0001依然只有一个bit翻转。同理，如果是depth=6，那么我们继续往

里收缩1位，只利用gray code关于对称轴两侧的部分编码，从4'b0011到4'b1011，我们可以看到，这样的编码依然可以保证相邻两个码之间只会有1位变化。



注意，利用这种编码，FIFO的满判断逻辑就不是简单的高两位取反，低位相同了，比如depth=7, rd_ptr=4'b0001, wr_ptr=4'b1100表示7个entry已经满了，如何得出正确的满判断逻辑就交给大家思考吧，欢迎大家在评论区留言讨论。

下一篇老李会带大家过一下工业界最常用的CDC tool Spyglass CDC的基本介绍，同时会奉上一些工程经验总结，敬请期待。

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！