# 特权同学 SDRAM-测试程序程序分析

知识点：sdram 控制器，时序约束，测试文件

第一部分：基本模块

Sdram 测试包括外部数据产生模块，fifo 模块，sdram 控制器，uart-RS232 传输模块。

Sdram 控制器模块分块三块：控制模块，命令模块，数据传输模块

具体来说，

1. 为什么要用 FIFO 模块呢？是因为外部数据与 SDRAM 控制器接收的速率不匹配，外部数据产生速率是 FPGA 系统时钟（25MHz），而 SDRAM 是 100MHz；同样，把 SDRAM 中数据通过串口传输出到上位机时，也会有一个速率匹配问题，所以就有两个 FIFO，其中一个写 FIFO，一个读 FIFO。
2. Sdram 控制器中控制模块完成初始化，自刷新，读写控制。命令模块有两个状态机，一个是初始化状态，另一个是工作状态。

第二部分：各个部分详解

1. Sdram 控制模块

输入引脚有：读、写请求，

输出引脚有：写应答（表明正在写），读应答，忙信号（表明正在工作），初始状态，工作状态，计数时钟（用于定义时间段，与数据传输模块紧密联系）

因为要用到好多时间参数，就直接建立一个参数文件。这个参数文件不是一个模块，所以不能用 module endmodule 来表示，里面所有参数有`define 来开始，只是最后命名为.v 文件。要用这个文件参数必须要前面加个“`”。`define end_trp cnt_clk_r= =TRP_CLK, 其中 cnt_clk_r 与控制模块中 cnt_clk_r 一致。（这点是新学的）

初始化过程分为 200us 等待，所有 L-band 预充电，8 个预刷新，模式寄存器设置。

```
//------------------------------------------------------------------------
//上电后 200us 计时,计时时间到,则 done_200us=1
//------------------------------------------------------------------------
reg[14:0] cnt_200us;
always @ (posedge clk or negedge rst_n)
    if(!rst_n) cnt_200us <= 15'd0;
    else if(cnt_200us < 15'd20_000) cnt_200us <= cnt_200us+1'b1; //计数

assign done_200us = (cnt_200us == 15'd20_000);      //条件满足则 done_200us=1，此后一直保持高电平，因为计数器保持不变，这不同于一个脉冲信号

//------------------------------------------------------------------------
//SDRAM 的初始化操作状态机
//------------------------------------------------------------------------
```

主要分析一个这个时间是怎么计算的?这个时间计算相当于用一个计时开关，这个开关就是 cnt_rst_n(低电平计数器复位)。在参数文件中，`define end_trp cnt_clk_r= =TRP_CLK 。cnt_clk_r 在启动时就一直递增，而达到 TRP_CLK 时，也就是 end_trp 成立，完成标志，关闭开关 cnt_rst_n.

```
always @ (posedge clk or negedge rst_n)
    if(!rst_n) cnt_clk_r <= 9'd0;              //计数寄存器复位
    else if(!cnt_rst_n) cnt_clk_r <= 9'd0;     //计数寄存器清零
    else cnt_clk_r <= cnt_clk_r+1'b1;          //启动计数延时

    //计数器控制逻辑
always @ (init_state_r or work_state_r or cnt_clk_r) begin
    case (init_state_r)
            `I_NOP:  cnt_rst_n <= 1'b0;
            `I_PRE:  cnt_rst_n <= (TRP_CLK != 0);//预充电延时计数启动
            `I_TRP:  cnt_rst_n <= (`end_trp) ? 1'b0:1'b1;      //预充电计数正好是 TRP_CLK,计数结束，清零计数器

//---------------------------------------------------------------------------
//15us 计时,产生自刷新请求。//每 60ms 全部 4096 行存储区进行一次自刷新，一行刷新时间是 15us，这是一直在循环
// (  存储体中电容的数据有效保存期上限是 64ms )
//---------------------------------------------------------------------------
```

这怎么与行选通配合呢？应该是在 15us 内进行了一次行有效跳转。

刷新操作分为两种：自动刷新（AR）和自刷新（self refresh,SR）,不论是何种刷新方式，都不需要外部提供行地址信息，因为这是一个内部的自动操作。对于 AR，SDRAM 内部有一个行地址生成器（也叫刷新计数器）用来自动的依次生成行地址。由于刷新是针对一行中所有存储体进行，所以无需列寻址。由于刷新是涉及到所有 L-Bank，因此，在刷新过程中，所有的 L-Bank 都停止工作，占用 N（PC133 为 9）个时钟后就可以进行正常的工作状态。

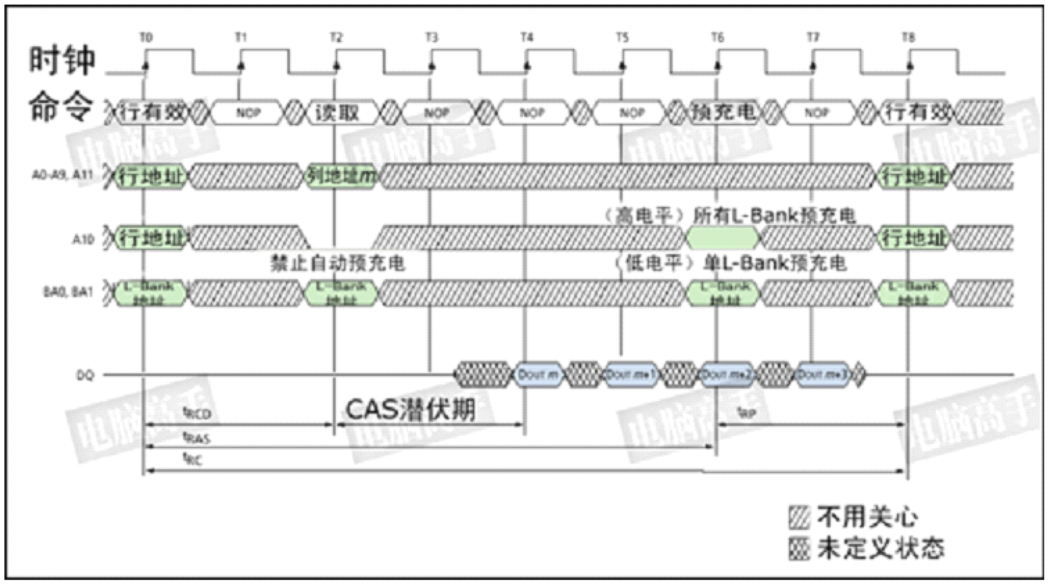
SR 则主要用于休眠模式低功耗状态下的数据保存，这方面最著名的应用就是 STR（Suspend to RAM，休眠挂起于内存）。在发出 AR 命令时，将 CKE 置于无效状态，就进入了 SR 模式，此时不再依靠系统时钟工作，而是根据内部的时钟进行刷新操作。在 SR 期间除了 CKE 之外的所有外部信号都是无效的（无需外部提供刷新指令），只有重新使 CKE 有效才能退出自刷新模式并进入正常操作状态。

AR 与 SR 的区别，外在表现：自动刷新 CKE 始终保持有效（置高），而自动刷新是 CKE 前一个时钟为高，后一个时钟为低。本源程序用的就是 AR。内在表现：自动刷新就是需要 CPU 提供刷新的时钟，自刷新就是 SDRAM 芯片自己刷新，不需要 CUP 提供刷新的时钟

如果要对同一 L-Bank 的另一行进行寻址，就要将原来工作的行关闭，重新发送行/列地址，L-Bank 关闭现有的工作行，准备打开新行的操作就是预充电（precharge）。预充电可以通过命令控制（需要提供 L-Bank 的地址），也可以通过设置 A10（置高）来让芯片在每次读写操

作之后自动进行预充电。

自刷新与预充电的区别？自刷新是对所有的行，而且不需要提供行地址信息。预充电是对指定的行（工作的行，工作的列）进行。具体如下图




```
reg[10:0] cnt_15us;//计数寄存器
always @ (posedge clk or negedge rst_n)
    if(!rst_n) cnt_15us <= 11'd0;
    else if(cnt_15us < 11'd1499) cnt_15us <= cnt_15us+1'b1;  // 60ms(64ms)/4096=15us 循环计数，这是一个脉冲信号，高电平只保留一个时钟周期
    else cnt_15us <= 11'd0;

//每 15us 就产生一个自刷新求，sdram_ref_ack 是自刷新工作指示
always @ (posedge clk or negedge rst_n)
    if(!rst_n) sdram_ref_req <= 1'b0;
    else if(cnt_15us == 11'd1498) sdram_ref_req <= 1'b1; //产生自刷新请求
    else if(sdram_ref_ack) sdram_ref_req <= 1'b0;          //已响应自刷新

assign sdram_ref_ack = (work_state_r == `W_AR);          // SDRAM 自刷新应答信号
//-------------------------------------------------------------------------
//SDRAM 的读写以及自动刷新操作状态机
//-------------------------------------------------------------------------
```

读写控制信号是在完成初始化后进行的，自刷新与自动刷新不一样，它是手动操作(这种想法是错误的，它们都是自动操作)，每隔 15us 产生一个刷新请求，自刷新优先级高于读写操作，这个很好理解，要是超过这个时间，信息就会丢失。读写开始状态是行有效，根据命令进行读写。读写后都有一个等待时间进行预充电以便进行下一次读写操作。为什么这里没有预充电操作呢？这是因为本源程序使用的是自动预充电（A10 置高），所以这里就只用一个等待就行。

```
reg[3:0] work_state_r;   // SDRAM 读写状态
reg sys_r_wn;            // SDRAM 读/写控制信号
```

```
        case (work_state_r)
            `W_RD:      work_state_r <= (`end_tread) ? `W_RWAIT:`W_RD;  // 后面需要添加一个读完成后的预充电等待状态
            `W_RWAIT:   work_state_r <= (`end_trwait) ? `W_IDLE:`W_RWAIT;
                        // SDRAM 写数据状态
            `W_WRITE:   work_state_r <= `W_WD;
            `W_WD:      work_state_r <= (`end_twrite) ? `W_TDAL:`W_WD;
            `W_TDAL:        work_state_r <= (`end_tdal) ? `W_IDLE:`W_TDAL;
                        // SDRAM 自动刷新状态
            `W_AR:      work_state_r <= (TRFC_CLK == 0) ? `W_IDLE:`W_TRFC;
            `W_TRFC:    work_state_r <= (`end_trfc) ? `W_IDLE:`W_TRFC;
            default:    work_state_r <= `W_IDLE;
            endcase
end

assign sdram_busy = (sdram_init_done && work_state_r == `W_IDLE) ? 1'b0:1'b1; // SDRAM 忙标志位，在进行初始化也是工作的一部分。

//`define  end_twrite          cnt_clk_r == TWRITE_CLK-2（为 6），这就好理解下面这些数字是怎么来，也就是在写没完成时就就指示在写。
assign sdram_wr_ack = ((work_state == `W_TRCD) & ~sys_r_wn) | (work_state == `W_WRITE)
                            | ((work_state == `W_WD) & (cnt_clk_r < 9'd6));      //写 SDRAM 响应信号,作为 wrFIFO 的输出有效信号

//`define  end_tread      cnt_clk_r == TREAD_CLK+2(为 10)
assign sdram_rd_ack = (work_state_r == `W_RD) & (cnt_clk_r > 9'd1) & (cnt_clk_r < 9'd10);
    //读 SDRAM 响应信号
assign sys_dout_rdy = (work_state_r == `W_RD && `end_tread);          // SDRAM 数据输出完成标志
```

2. Sdram 命令控制模块

这个比较简单，只是利用命令。注意只有上电初始化时使时钟使 CKE 为无效（复位为 0），其它命令 CKE 均置 1.在模式寄存器设置（MRS）时，设置突发长度为 8，即是一次读或写均是 8Words.

```
    assign {sdram_cke,sdram_cs_n,sdram_ras_n,sdram_cas_n,sdram_we_n} = sdram_cmd_r;
        //SDRAM 控制信号命令
    `define         CMD_INIT     5'b01111     //上电初始化命令端口
    `define         CMD_NOP          5'b10111     // NOP COMMAND
    `define         CMD_ACTIVE       5'b10011     // ACTIVE COMMAND
    `define         CMD_READ   5'b10101     // READ COMMADN
    `define         CMD_WRITE  5'b10100     // WRITE COMMAND
    `define         CMD_B_STOP       5'b10110     // BURST      STOP
    `define         CMD_PRGE   5'b10010     // PRECHARGE
    `define         CMD_A_REF  5'b10001     // AOTO REFRESH,CKE=1 为自动刷新
```

`define CMD_LMR 5'b10000 // LODE MODE REGISTER

| 命令名（功能） | CS# | RAS# | CAS# | WE# | DQM | 地址线 | DQs |
|---|---|---|---|---|---|---|---|
| 命令禁止（NOP） | H | X | X | X | X | X | X |
| 无操作（NOP） | L | H | H | H | X | X | X |
| 有效/活动（使指定L-Bank中的指定行有效） | L | L | H | H | X | Bank/行 | X |
| 读取（从指定L-Bank中的指定列开始读取数据） | L | H | L | H | UH | Bank/列 | X |
| 写入（从指定L-Bank中的指定列开始写入数据） | L | H | L | L | UH | Bank/列 | 有效 |
| 突发传输终止 | L | H | H | L | X | X | 活动 |
| 预充电（关闭指定或全部L-Bank中的工作行） | L | L | H | L | X | 相应编码 | X |
| 自动刷新或自刷新（后者进入自刷新模式） | L | L | L | H | X | X | X |
| 模式寄存器加载（写入） | L | L | L | L | X | 操作码 | X |
| 写允许/输出允许 | - | - | - | - | L | - | 有效 |
| 写禁止/输出屏蔽（High-Z） | - | - | - | - | H | - | 屏蔽 |

3. Sdram 数据传输模块

难点在于 sdram_datar 的控制。由于 sdram_data 是双向口，故要有一个方向控制位，也就是当作为输入口时，输出就是高阻，同理，当作为输出口时，输入就为高阻。

```
//将待写入数据送到 SDRAM 数据总线上
always @ (posedge clk or negedge rst_n)
    if(!rst_n) sdr_dlink <= 1'b0;
  else if((work_state == `W_WRITE) | (work_state == `W_WD)) sdr_dlink <= 1'b1;
    else sdr_dlink <= 1'b0;

assign sdram_data = sdr_dlink ? sdr_din:16'hzzzz;      //此时作为输入端口
```

不可能再同样定义一个 sdr_dlink_out，然后再用 assign 语句赋值，这样不对的，不能对同一个信号进行两次赋值

```
//将数据从 SDRAM 读出
always @ (posedge clk or negedge rst_n)
    if(!rst_n) sdr_dout <= 16'd0;  //突发数据读寄存器复位
    else if((work_state == `W_RD) & (cnt_clk > 9'd0) & (cnt_clk < 9'd10))
sdr_dout <= sdram_data;      //连续读出 256B 的 16bit 数据存储到 rdFIFO 中？？？？

assign sys_data_out = sdr_dout;
```

4. rdfifo 模块（这是个难点）

因为要把 SDRAM 数读出去，需要一个 rdfifo，我们刚分析完 sdram,，紧接着分析 rdfifo 这样好理解些。其实把 SDRAM 中的数读出，就是把读出的数写入到 rdfifo 中去，故写时钟就是 sdram 的工作频率，也就是 100MHz,。分析一下 rdfifo 中写请求，为什么接 sdram_rd_ack 呢？其实分析一下原理就知道了，这个过程是把 SDRAM 中的数读出，把读出的数写入到 rdfifo 中去，故 sdram 正在读出时，就把这个信号启动 rdfifo 写请求，就一直写。

注意比较为什么不把 sdram_rd_req 作为启动信号？前者是一高电平，后者是一个脉冲，只保持一个时钟的高电平。（也解释不通呀!？？）

第二点要注意的是使用写使用量（wrusedw）经过运算得到输出控制信号（没有有 full,empty 标志信号），而不是用 rdusedw.这两个有什么区别吗？区别在于 wusedw 表示当前

fifo 已经被使用了多少 word(就是深度),而 rdusedw 表示当前 fifo 还有多少 word 可供读取(好像感觉一样哦，都表示当前有多少个）,只是表达的侧重点不同，我知道的一个区别是它们同步的时钟不同，前者与写时钟同步。

关于 wrusedw 与 rdusedw 再多讲一点，我们就以一个例子来讲，例如，设置 fifo 有 16 个 word,假设现在只写不读，当 wrusedw 等于是 15 时，fifo 已经写满，此时若继续写入数据，则 wrusedw 在下一个有效时钟边沿到来时，wrused 会等于 0,，而不是保持 15 不变。Rdused 也是一样的，当为 0 时还读，下一个时刻 rdusedw 为 15，而不是保持 0 不变。

```
rdfifo          uut_rdfifo(
                    .data(sys_data_out),
                    .rdclk(clk_25m),
                    .rdreq(rdf_rdreq),
                    .wrclk(clk_100m),
                    .wrreq(sdram_rd_ack),
                    .q(rdf_dout),
                    .wrusedw(rdf_use)
                    );
```

//Syswr_done 是产生数据完成标志。与 syswr_done 的关系怎么理解？写地址完成了，只要 sdram 处在在读状态（要读了），就开始处于读地址状态（读地址从 0 开始递增）

```
assign sdram_rd_req = ((rdf_use <= 9'd256) & syswr_done);      //sdram 写入完成且 FIFO 半空（256 个 16bit 数据）即发出读 SDRAM 请求信号（这样理解错了）
```

我们暂且不考虑 syswr_done 的影响。Wrusedw 开始为 0，就启动 sdram 读请求，Sdram 马上进入在读状态，启动 rdfifo 的写请求，8 个一次的数据进入 rdfifo 中（sdram 突发读长度为 8），当 rdfifo 写入大于 256 时，sdram 就不读了，随后 rdfifo 也不写了。

Rdfifo 中读请求信号来自于串口发送模块的 fifo232_rdreq,丨 串口发送开始时就启动 fifo232_rdreq,（这中间时间要打几拍，取上升沿），即启动 rdfifo 读请求。简单地理解是，串口发送准备好了，就打开 rdfido 的读操作。

5. wrfifo 模块

分析完了 rdfifo，分析 wrfifo 就比较简单了，sdram 要在写状态时，就启动 wrfifo 读请求，把 wrfifo 的数输入到 sdram 中。

```
wrfifo          uut_wrfifo(
                    .data(wrf_din),
                    .rdclk(clk_100m),
                    .rdreq(sdram_wr_ack),
                    .wrclk(clk_25m),
                    .wrreq(wrf_wrreq),
                    .q(sys_data_in),
                    .wrusedw(wrf_use)
                    );
```

```
assign sdram_wr_req = ((wrf_use >= 9'd8) & ~syswr_done);      //FIFO（8 个 16bit 数据）即发出写 SDRAM 请求信号
```

Wrfifo 中写请求信号来自于数据产生 datagene 模块的 wrf_wrreq

6. 数据产生模块

问题：1.输入引脚用 sdram_rd_ack,有什么作用？启动把产生的数读出去（读地址就行）

2.系统写完成 syswr_done 有什么作用？对两个 fifo 的控制

```
input sdram_rd_ack;                    //系统读 SDRAM 响应信号,作为 rdFIFO 的输写有效信号,这里捕获它的下降沿作为读地址自增加标志位

reg sdr_rdackr1,sdr_rdackr2;

//----------------------------------------
//捕获 sdram_rd_ack 下降沿标志位
always @(posedge clk or negedge rst_n)
        if(!rst_n) begin
                sdr_rdackr1 <= 1'b0;
                sdr_rdackr2 <= 1'b0;
            end
        else begin
                sdr_rdackr1 <= sdram_rd_ack;
                sdr_rdackr2 <= sdr_rdackr1;
            end

wire neg_rdack = ~sdr_rdackr1 & sdr_rdackr2;

//----------------------------------------
//上电 500us 延时等待 sdram 就绪
reg[13:0] delay;     //500us 延时计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) delay <= 14'd0;
    else if(delay < 14'd12500) delay <= delay+1'b1;//后来一直保持 12500 不变

wire delay_done = (delay == 14'd12500);     //500us 延时结束.25MHz

//----------------------------------------
//每 640ns 写入 8 个 16bit 数据到 sdram? sdram 一次写请求就写进 8words,地址自动增加
//上电后所有地址写入完毕时间需要不到 360ms 时间?2^19*640ns=336ms
reg[5:0] cntwr;      //写 sdram 定时计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) cntwr <= 6'd0;
    else if(delay_done) cntwr <= cntwr+1'b1;
//----------------------------------------
always @(posedge clk or negedge rst_n)
    if(!rst_n) addr <= 19'd0;
```

else if(!wr_done && cntwr == 6'h3f) addr <= addr+1'b1;// 写地址每个产生时间为0.5us*64=32us,这是错误的，这是理解错了 delay_doney 为脉冲信号，它其实是一个前 0.5us 为低电平，后来一直保持高电平。所以写一个地址需要时间为 10ns*64=640ns

else if(wr_done && neg_rdack) addr <= addr+1'b1;  //读地址产生

assign moni_addr = {addr,3'b000}; //这里体会到只写一个地址达到的效果是可以写 8 个地址，这与 SDRAM 突发写长度为 8

当地址为 19'h7ffff,下一时钟采集到 wr_done=1,不进行地址加 1，此到等到 sdram 正在读（要读），地址加 1 重新回到零了，然后读地址从零开始递增。

```
always @(posedge clk or negedge rst_n)
    if(!rst_n) wr_done <= 1'b0;
    else if(addr == 19'h7ffff) wr_done <= 1'b1;

assign syswr_done = wr_done;

//----------------------------------------
//写 sdram 请求信号产生，即 wrfifo 的写入有效信号
写地址时间每个是 64*10ns,而写入的数据是在第 50ns~120 之间
always @(posedge clk or negedge rst_n)
    if(!rst_n) wrf_wrreqr <= 1'b0;
    else if(!wr_done) begin  //上电 0.5ms 延时完成
        if(cntwr == 6'h05) wrf_wrreqr <= 1'b1; //写请求信号产生
        else if(cntwr == 6'h0d) wrf_wrreqr <= 1'b0;  //请求信号撤销 8
    end

always @(posedge clk or negedge rst_n)
    if(!rst_n) wrf_dinr <= 16'd0;
    else if(!wr_done && ((cntwr > 6'h05) && (cntwr <= 6'h0d))) begin   //上电 0.5ms 延时完成
            wrf_dinr <= wrf_dinr+1'b1;   //写入数据递增，其实是模块中产生的数，是要读出去的
        end
```

第四部分：时序分析（这点难点）
本例程在 Quarters9.0 中运行时，出现不满足时序。