1 二端口 DDR3 控制器模块(ddr3_ctrl_2port)设计

章节导读

本节将主要介绍 DDR3 二端口控制器的一种设计方法。通过完成本次 DDR3 二端口控制器设计，读者能够简化大容量缓存的操作控制，从而实现仅需犹如 FIFO 般简便的操作，就可以享受 DDR 的巨大存储容量的这样一个现实应用。

1.1 DDR 控制器简要说明

智多晶为 Seal 5000 系列 FPGA 提供 DDRC（DDR Controller）IP 支持。DDRC 用于管理 FPGA 与外部 DDR（双倍数据速率）存储器之间的通信，实现高效、可靠的数据传输，并简化系统设计。

DDRC 的典型应用场景包括：

- 实时信号处理：雷达、通信系统中高速缓存数据流。
- 高性能计算：加速卡中暂存中间计算结果。
- 视频处理：帧缓冲存储与高分辨率视频流处理。

DDRC IP 所支持器件（规格信息）集成 DDR 控制器硬核（Hardcore），其中 SA5T-366 包含 2 个 DDRC 硬核，SA5Z-30、SA5Z-50、SA5T-100 包含 1 个 DDRC 硬核。不同器件的 DDRC 硬核存在一定差异。SA5T-100 与 SA5T-366 中的 DDRC 硬核支持双 FIFO 接口和 AXI4 接口；SA5Z-30 与 SA5Z-50 中的 DDRC 硬核只支持双 FIFO 接口，但可以通过 DDRC IP 扩展 AXI4 接口，消耗一定的逻辑资源。DDRC IP 额外提供了在硬核 FIFO 接口之上包装的 Native 接口。相比之下，硬核 FIFO 接口对指令时序的要求更严格，Native 接口则更加易用，除对资源占用极度敏感的场景下选择使用硬核 FIFO 接口，其他情况下都首推 Native 接口，可以显著降低调试成本。SA5Z-30 器件额外提供 DFI 接口支持，基于逻辑资源实现。

1.1.1 性能说明

支持以下 DDR2 速率：

- 400Mbps (4-4-4/3-3-3)；
- 533Mbps (4-4-4/3-3-3)；

- 667Mbps (5-5-5/4-4-4)；
- 800Mbps (6-6-6/5-5-5/4-4-4)。

支持以下 DDR3 速率：

- 800Mbps (6-6-6/5-5-5)；
- 1066Mbps (8-8-8/7-7-7/6-6-6)；
- 1333Mbps (10-10-10/9-9-9/8-8-8/7-7-7)；
- 1600Mbps (11-11-11/10-10-10/9-9-9/8-8-8)；
- 1866Mbps (13-13-13/12-12-12/11-11-11/10-10-10)。

最大 IO 速率：

- SA5Z-30：800Mbps；
- SA5Z-50：1066Mbps；
- SA5T-100：1600Mbps；
- SA5T-366：1866Mbps。

1.1.2 IP 配置

点击软件上方的 IP 管理，进入 IP 管理器，滑到最底端找到 DDR3 控制器，双击创建 IP，操作如下所示：

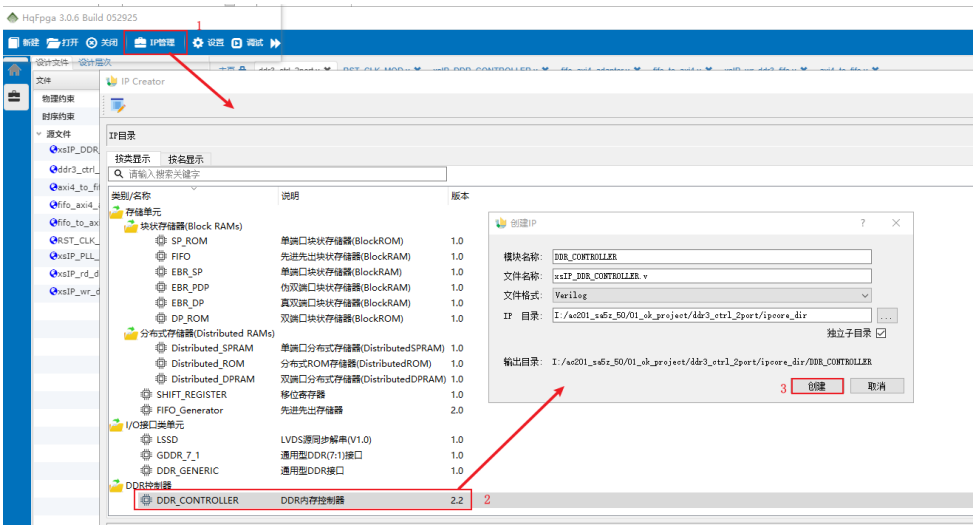


图 1-1 进入 IP 创建界面

进入，对 IP 进行配置如下所示：

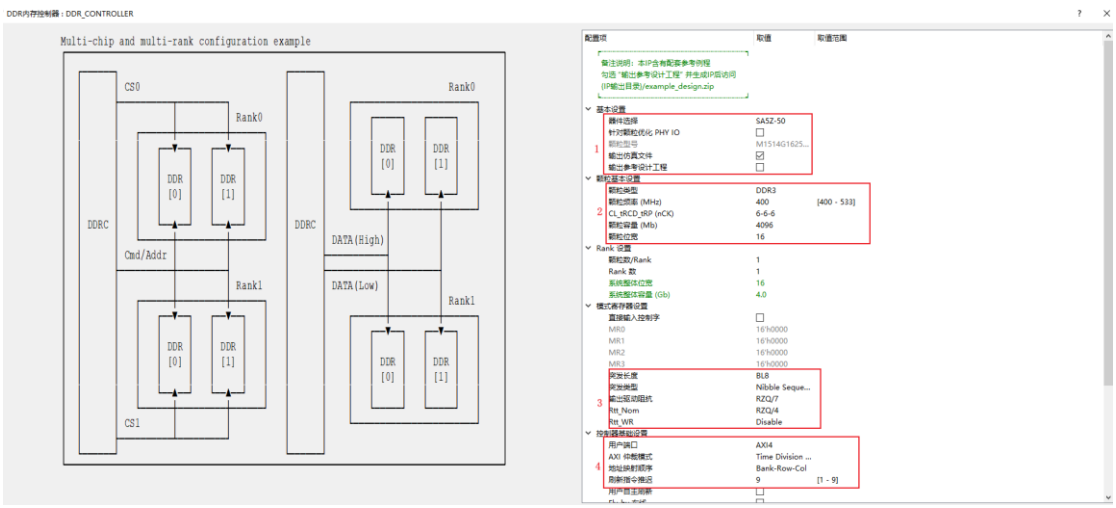


图 1-2DDR 控制器配置

1. 基本设置：我们实验用到的开发板时 AC201-SA5Z-50D0 的开发板，所以器件选择为 SA5Z-50，本次实验主要用到的是仿真，所以我们这里必须勾选输出仿真文件，方便我们进行仿真测试。
2. 颗粒基本设置：首先实验用到的开发板时 AC201-SA5Z-50D0 的开发板上的 DDR 芯片为 F60C1A0004-M79W（江波龙，单颗 512MB，工业级），该芯片为 DDR3 芯片，存储容量为 4GB。这里设置颗粒类型为 DDR3，颗粒频率为 400M，CL-tRCD-tRP（内存时序）为 6-6-6，颗粒容量 = 颗粒地址空间 × 颗粒位宽，颗粒容量为 4096M（4GB），颗粒位宽为 16。
3. 模式寄存器设置：突发长度设置为 BL8；突发类型为半字节顺序（Nibble Sequential），按照连续地址顺序访问存储单元；输出驱动阻抗选择 RZQ/7，Rtt_Nom 设置静态终端电阻值为 R ZQ/4。
4. 控制器基础设置：用户端口选择为 AXI4，仲裁模式选择为 Time Division Multiplexing，该设置读写优先级相同，读写交替进行。

建立完成之后，我们可以看到在 ipcore_dir\DDR_CONTROLLER 下有两个文件，如下所示：

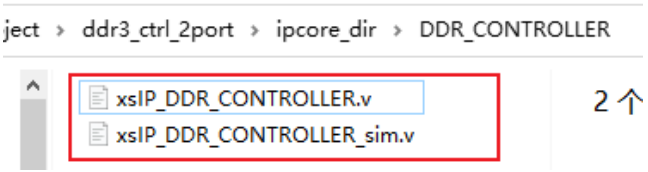


图 1-3 DDR3 控制器 IP

上述图中的两个文件，带 sim 的文件是仿真需要使用的，实际下板验证的

时候是不带 sim 的文件，这两个文件在端口输出上有一定的区别，如下所示：

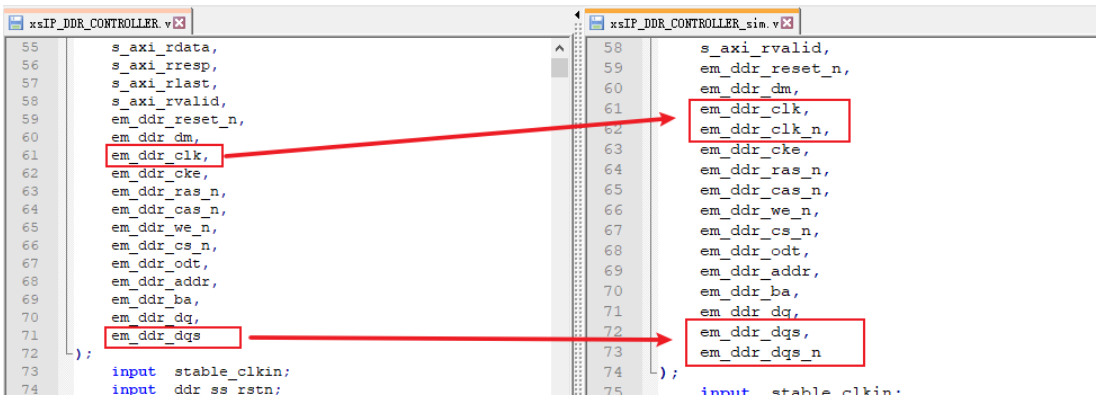


图 1-4 文件端口差别

可以看到实际用的时候，ddr_clk 和 ddr_dqs 都不是差分信号，分配引脚的时候直接接到开发板原理图的 P 端即可，本次实验主要是仿真，使用的时候，例化的是 sim 文件，在用 HQ 软件编译会报错，读者主要使用 Modelsim 查看仿真工程和仿真波形。

1.1.3 端口说明

1.1.3.1 时钟与复位

表 1-1 时钟与复位

信号	方向	描述
stable_clkin	I	在 PLL 锁定前保持稳定的低速时钟，频率需小于 ddr_work_clkin 的一半，我们在使用的时候，该时钟设置为 50M
ddr_ss_rstn	I	DDR 子系统复位，低电平有效
pll_lock	I	DDRC 外挂 PLL 的锁定信号
ddr_work_clkin	I	DDR 的工作时钟，即 ECLK，使用时设置为 100M
ddr_data_clkout	O	输出给用户的 DDR 数据时钟，即 SCLK，由 ECLK 分频（2/4 分频）而来

1.1.3.2 控制与状态

表 1-2 控制与状态

信号	方向	描述
init_status	O	init_status[2] = init_start 初始化启动 init_status[1] = init_done 初始化结束 init_status[0] = clocking_good 时钟正常
training_report	O	training_report[2] = wl_err 写均衡失败 training_report[1] = rl_err 读均衡失败 training_report[0] = rt_err 读训练失败

usr_ar_trigger	I	用户模式开启自刷新脉冲请求

1.1.3.3 AXI4 接口

表 1-3 AXI4 接口

信号	方向	描述
aclk	I	全局时钟
s_axi_awid	I	写地址 ID，位宽为 3
s_axi_awaddr	I	写地址，位宽为 32
s_axi_awlen	I	写突发长度,支持 0~255
s_axi_awsize	I	写突发大小
s_axi_awburst	I	写突发类型，只支持 INCR，给值 2’b10
s_axi_awlock	I	锁类型，不支持，给值 0
s_axi_awcache	I	内存类型，不支持，给值 0
s_axi_awprot	I	保护类型，不支持，给值 0
s_axi_awqos	I	服务质量，不支持，取值 0
s_axi_awvalid	I	写地址有效
s_axi_awready	O	写地址就绪，表明 slave 准备好接收地址
s_axi_wdata	I	写数据
s_axi_wstrb	I	写选通
s_axi_wlast	I	写数据最后一笔标记
s_axi_wvalid	I	写数据有效
s_axi_wready	O	写数据就绪，表明 slave 准备好接收数据
s_axi_bid	O	写响应 ID
s_axi_bresp	O	写响应
s_axi_bvalid	O	写响应有效
s_axi_bready	I	写响应就绪
s_axi_arid	I	读地址 ID
s_axi_araddr	I	读地址
s_axi_arlen	I	读突发长度，支持 0-255
s_axi_arsize	I	读突发大小
s_axi_arburst	I	读突发类型，只支持 INCR，给值 2’b10
s_axi_arlock	I	锁类型，不支持，给值 0
s_axi_arcache	I	内存类型，不支持，给值 0
s_axi_arprot	I	保护类型，不支持，给值 0
s_axi_arqos	I	服务质量，不支持，取值 0
s_axi_arcalid	I	读地址有效
s_axi_arready	O	读地址就绪，表明 slave 准备好接收地址
s_axi_rid	O	读数据 ID
s_axi_rdata	O	读数据
s_axi_rresp	O	读响应
s_axi_rlast	O	读数据最后一笔标记
s_axi_rvalid	O	读数据有效
s_axi_rready	I	读数据就绪，表明 master 准备好接收读数据

1.1.3.4 内存接口

表 1-4 内存接口

信号	方向	描述
em_ddr_clk	O	内存时钟
em_ddr_reset_n	O	内存异步复位，低电平有效
em_ddr_addr	O	内存访问地址（Row 地址与 Col 地址复用）
em_ddr_ba	O	内存访问 Bank
em_ddr_dq	IO	双向数据总线
em_ddr_dqs	IO	双向数据选通信号
em_ddr_dm	O	写数据字节掩码
em_ddr_cke	O	内存时钟使能
em_ddr_ras_n	O	内存行地址选中，低电平有效
em_ddr_cas_n	O	内存列地址选中，低电平有效
em_ddr_we_n	O	内存写使能，低电平有效
em_ddr_cs_n	O	内存片选，低电平有效
em_ddr_odt	O	内存片上端接（On-Die Termination）信号

1.2 DDR 存储器的应用局限

我们知道，数据的存储和读出需满足如下基本原则：对于一个存储器来说，在进行写操作时，如果存储器已满或将满时，则不应再向其中写入数据，即使写入数据也是无效的。如果在进行读操作时，存储器已空或将空，则不应再从其中读出数据，否则读出的数据也是无效的。

在某些典型条件下，针对上述原则，我们如果不加额外的处理手段而单单只使用 DDR3 控制器 IP 核，则数据的读写有效性就有可能无法得到保证，具体如下：

首先，虽然 DDR3 控制器提供给用户的 ui 时钟频率为恒定，用户只需要按照该固定频率写入和读出数据即可，但是大部分硬件也会有自身的固定工作频率和数据读写频率，甚至有的器件固定工作频率、数据读写频率和 DDR3 的 ui 时钟频率相差极大。存储器和外设硬件的读写频率差异，很有可能将导致 DDR3 的 ui 时钟无法满足写入侧或读出侧的硬件读写速率需求，从而读写两端速率不匹配。如果读写速率不匹配，则不但 DDR3 与外设的数据交互会存在跨时钟域的问题，还会导致 DDR 读写出错。

其次，即使 DDR3 的读写速率刚好也可以满足硬件外设的读写速率，还会存在数据读写连续的问题。在有的外设发送或接收有效数据并不是连续的前提下，我们如果设置 DDR3 控制器 IP 核时，默认外设每一拍发送的数据都是有效

数据，而不针对数据读写不连续现象作其他收发控制，则也会导致数据读写出错。

此外，为了保证数据的交互速率高效，DDR3 IP 核典型的应用数据交互位宽为 128 位，而我们的外设又多以 8 位和 16 位等低位宽读写居多，这样，位宽的不匹配也会成为数据读写和缓存的一大障碍。

凡此种种，直接使用 DDR 控制器，会存在的问题可以总结如下图 1-5：

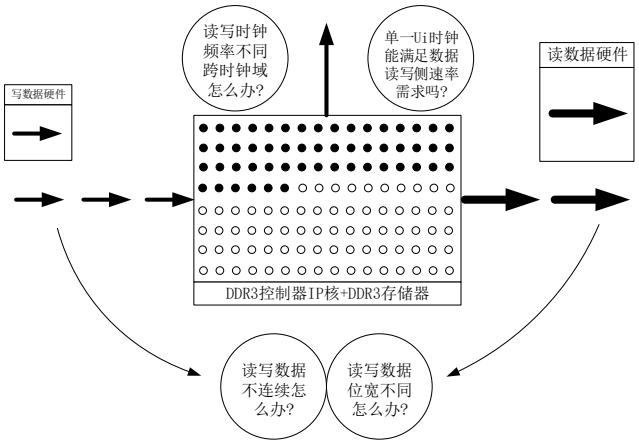


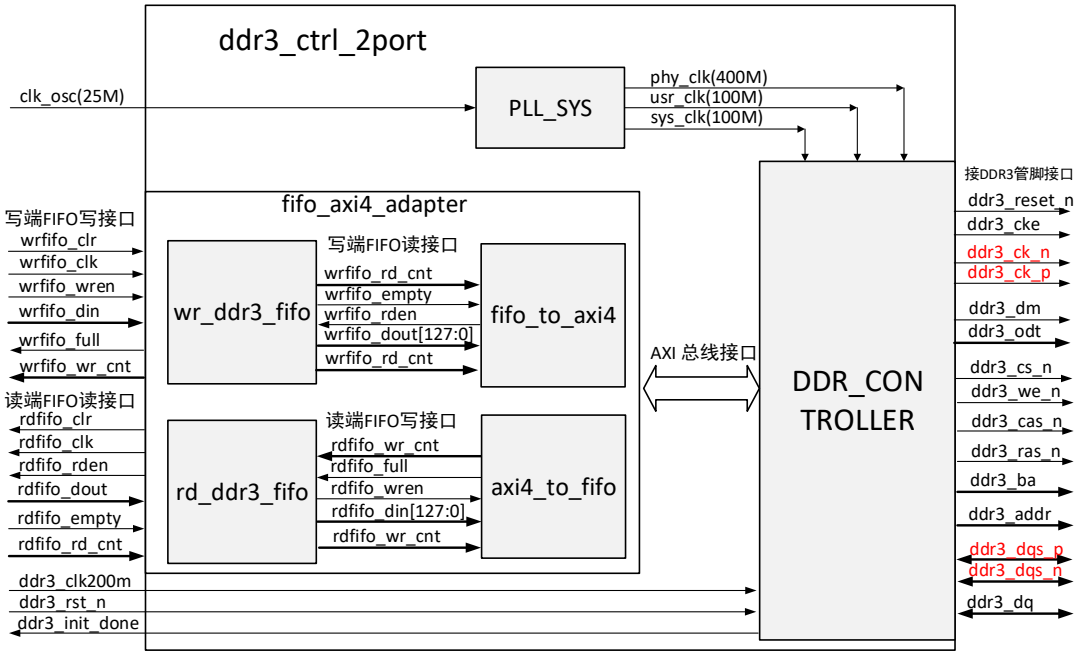
图 1-5 直接使 DDR 控制器 IP 核会遇到的实际问题

值得庆幸的是，以上种种 DDR3 控制器应用上的障碍，都能够通过在读写端各添加一个 FIFO 来获得较好解决。如果在读写端各添加一个 FIFO 对写入和读出数据进行缓存，既可以解决跨时钟域的读写问题，又可以适配多种多样的外设读写速率需求，同时，还可以满足数据在固定频率下非连续写和非连续读、位宽不匹配等种种问题。

因此，我们有必要借助前面讲解的内容，对 DDR3 控制器 IP 核进行深入设计。

1.3 模块整体结构框图

经过对交互数据的端口分析，我们可以绘制出 DDR3 二端口控制器设计框图：



注：图中红色标注部分为仿真使用时的端口，实际使用时该信号为单端信号，只需要接 P 端

图 1-6 DDR3 二端口控制器模块组框图

该控制器由如下几个部分构成：

1. DDR3 读写控制的 IP 核（DDR_CONTROLLER）
2. PLL IP 核（PLL_SYS）：生成 DDR3 控制器需要的时钟
3. 读写 FIFO 和 DDR3 IP 核数据交互控制模块（fifo_to_axi4 和 axi4_to_fifo）
4. 写 FIFO 模块（wr_ddr3_fifo）:负责将上游外设传递而来的数据写入 DDR3 控制器
5. 读 FIFO 模块（rd_ddr3_fifo）:负责将 DDR3 控制器内部数据读出到下游
6. 将模块组进行整合的顶层模块（ddr3_ctrl_2port）

其中，为了管理方便，将上述 2、3、4 进行了一个集成，命名集成模块为 fifo_axi4_adapter。

从以上模块组框图中我们不难发现，该模块组设计完成后，实际应用时，只需要在工程中合理描述写 FIFO 的写端口和读 FIFO 的读端口对应关系，就可以有效完成 DDR3 的读写控制。至于 DDR3 控制器和 DDR3 器件的数据交互、FIFO 和 DDR3 控制器 IP 核的数据交互，我们则不必过多关心。

1.4 模块参数及端口描述

按照上面的模块组设计方案，我们实际在对 DDR3 控制器进行读写控制时，只需要重点对写 FIFO 的写侧端口和读 FIFO 的读侧 FIFO 进行输入输出信号对应即可。

表 1-5 ddr3_ctrl_2port 模块参数说明

参数名	描述
DW	模块写 FIFO 的写数据和读 FIFO 的读数据位宽，即数据 wrfifo_din 和 rdfifo_dout 的位宽，设置值需被 128 整除
WR_ADDR_BEGIN	写数据存储空间的起始地址，1 个地址对应的数据位宽为（DW） bit
WR_ADDR_END	写数据存储空间的终止地址，1 个地址对应的数据位宽为（DW） bit
RD_ADDR_BEGIN	取数据存储空间的起始地址，1 个地址对应的数据位宽为（DW） bit
RD_ADDR_END	取数据存储空间的终止地址，1 个地址对应的数据位宽为（DW） bit
AXI_DATA_WIDTH	AXI 总线数据位宽
AXI_ADDR_WIDTH	AXI 总线地址位宽
AXI_ID	AXI 总线 ID 信号位宽
AXI_BURST_LEN	AXI 总线突发长度
FIFO_ADDR_DETH	读写 FIFO 面向 DDR3 侧接口的深度

表 1-6 ddr3_ctrl_2port 模块端口说明

端口名	方向	描述
clk_osc	I	输入的 25M 的时钟
ddr3_rst_n	I	提供给 DDR3 控制器的复位信号，低电平时为复位
ddr3_init_done	O	DDR3 控制器初始化及校准完成标识信号，为高表示初始化及校准成功
写 FIFO 的写接口，用户往 DDR3 写入数据的接口		
wrfifo_clr	I	写 FIFO 清空控制信号，给高电平表示执行清空，执行清空操作时，需保证给 3 个及以上个时钟（wrfifo_clk）周期的高电平
wrfifo_clk	I	写 FIFO 的写操作工作时钟
wrfifo_wren	I	写 FIFO 的写数据使能控制信号，给高电平表示往 FIFO 写入数据，为避免写入数据的丢失，确保在 FIFO 非满（wrfifo_full=0）情况下写入数据
wrfifo_din	I	写 FIFO 的写数据信号，数据位宽为 DW
wrfifo_full	O	写 FIFO 的写满标识信号，用于标识当前 FIFO 是否有被写满
wrfifo_wr_cnt	O	写 FIFO 的写数据计数，表示当前 FIFO 中缓存数据的个数
读 FIFO 的读接口，用户向 DDR3 读取数据的接口		
rdfifo_clr	I	读 FIFO 清空控制信号，给高电平表示执行清空，执行清空操作时，需保证给 3 个及以上个时钟（rdfifo_clk）周期的高电平
rdfifo_clk	I	读 FIFO 的读操作工作时钟
rdfifo_rden	I	读 FIFO 的读数据使能控制信号，给高电平表示往 FIFO 读数据，为避免读数据的丢失，确保在 FIFO 非空（rdfifo_empty =0）情况下读数据
rdfifo_dout	O	读 FIFO 的读数据输出，数据位宽为 DW，
rdfifo_empty	O	读 FIFO 的读空标识信号，用于标识当前 FIFO 是否为空（即 FIFO 内有

		无数据）
rdfifo_rd_cnt	O	读 FIFO 的读数据计数，表示当前 FIFO 中缓存数据的个数
接 DDR3 管脚接口（仿真中与 ddr3_model 对接）		
ddr3_dq[31:0]	IO	数据输入、输出：双向数据总线。若模式寄存器中使能了 CRC 功能，那么在数据 burst 结束时就会附加一段 CRC 码。
ddr3_dqs_n[1:0] ddr3_dqs_p[1:0]	IO	差分数据选通信号：差分信号对，作输入时与写数据同时有效，作输出时与读数据同时有效。读数据时与边沿对齐，但是跳变沿位于写数据的中心。DDR3 SDRAM 仅支持选通信号为差分信号，不支持单根信号的数据选通信号。
ddr3_addr[14:0]	O	地址输入，为 ACTIVATE 命令提供行地址和 READ/WRITE 命令的列地址和自动预充电（A10），以便从某个 bank 的内存阵列里选出一个位置
ddr3_ba[2:0]	O	Bank 地址输入，定义 ACTIVATE、READ、WRITE 或 PRECHARGE 命令是对哪一个 bank 操作的
ddr3_ras_n ddr3_cas_n ddr3_we_n	O	命令输入，这三个信号，连通 cs_n，定义一个命令
ddr3_reset_n	O	复位，低电平复位，复位是异步的
ddr3_ck_p ddr3_ck_n	O	时钟，差分时钟输入，所有控制和地址输入信号在 ck_p 上升沿和 ck_n 的下降沿交叉处被采样，输出数据选项（dqs_n、dqs_p）参考与 ck_p 和 ck_n 的交叉点
ddr3_cke	O	时钟使能：CKE 为高电平时，启动内部时钟信号、设备输入缓冲以及输出驱动单元。CKE 低电平时则关闭上述单元。当 CKE 为低电平时，可使设备进入 PRECHARGE POWER DOWN、SELF-REFRESH 以及 ACTIVE POWER DOWN 模式。CKE 与 SELF REFRESH 退出命令是同步的。在上电以及初始化序列过程中，VREFCA 与 VREF 将变得稳定，并且在后续所有的操作过程中都要保持稳定，包括 SELF REFRESH 过程中。CKE 必须在读写操作中保持稳定的高电平。在 POWER DOWN 过程中，除 CK_t，CK_c，ODT 以及 CKE 以外的所有输入缓冲都是关闭的。在 SELF REFRESH 过程中，除 CKE 以外的所有输入缓冲都是关闭的。在正时钟上升边沿采样。
ddr3_cs_n	O	片选信号，当 CS_n 锁存为高电平时，所有的命令都被忽略。在正时钟上升边沿采样。
ddr3_dm	O	输入数据掩码，dm 信号是作为写数据的掩码信号，当 dm 信号信号为低电平时，写命令的输入数据对应的位将被丢弃。dm 信号在 DQS 的两个条边沿都采样。
ddr3_odt	O	On-Die Termination，片上终端电阻；ODT 信号可使能 DDR SDRAM 内部的 RTT_NOM 终端电阻。该设计通过允许 DRAM 控制器独立地打开/关闭任一或所有 DRAM 设备的终端电阻来改善存储器通道的信号完整性。DRAM 通过 ODT 控制引脚为每个 DQ，DQS 和 DM 开启/关闭终端电阻。与其他输入命令不同，ODT 引脚直接控制 ODT 动作，不对其进行时钟采样。在自刷新模式下不支持 ODT。可以选择在 CKE 掉电期间通过模式寄存器启用 ODT 操作。 请注意，如果在掉电模式下启用 ODT，则在掉电期间可能无法关闭 VDDQ（I/O 供电），同时 DRAM 也会在读操作期间无法关闭。

下方的代码给出的是模块例化模板。至于参数的设置和端口信号的连接方

法，可查看上述端口列表描述。

```
ddr3_ctrl_2port #(
    .FIFO_DW            (16  ),
    .WR_BYTE_ADDR_BEGIN (0   ),
    .WR_BYTE_ADDR_END   (2047),
    .RD_BYTE_ADDR_BEGIN (0   ),
    .RD_BYTE_ADDR_END   (2047),
    .FIFO_ADDR_DEPTH    (64  )
)
ddr3_ctrl_2port(
    // clock reset
    .clk_osc        (clk_osc),
    .ddr3_rst_n     (ddr3_rst_n     ),
    .ddr3_init_done(ddr3_init_done),
    // wr_fifo wr Interface
    .wrfifo_clr     (wrfifo_clr     ),
    .wrfifo_clk     (wrfifo_clk     ),
    .wrfifo_wren    (wrfifo_wren    ),
    .wrfifo_din     (wrfifo_din     ),
    .wrfifo_full    (wrfifo_full    ),
    .wrfifo_wr_cnt (wrfifo_wr_cnt ),
    // rd_fifo rd Interface
    .rdfifo_clr     (rdfifo_clr     ),
    .rdfifo_clk     (rdfifo_clk     ),
    .rdfifo_rden    (rdfifo_rden    ),
    .rdfifo_dout    (rdfifo_dout    ),
    .rdfifo_empty   (rdfifo_empty   ),
    .rdfifo_rd_cnt (rdfifo_rd_cnt ),
    //DDR3 Interface
    // Inouts
    .ddr3_dq        (ddr3_dq        ),
    .ddr3_dqs_n     (ddr3_dqs_n     ),
    .ddr3_dqs_p     (ddr3_dqs_p     ),
    // Outputs
    .ddr3_addr      (ddr3_addr      ),
    .ddr3_ba        (ddr3_ba        ),
    .ddr3_ras_n     (ddr3_ras_n     ),
    .ddr3_cas_n     (ddr3_cas_n     ),
    .ddr3_we_n      (ddr3_we_n      ),
    .ddr3_reset_n   (ddr3_reset_n   ),
    .ddr3_ck_p      (ddr3_ck_p      ),
    .ddr3_ck_n      (ddr3_ck_n      ),
    .ddr3_cke       (ddr3_cke       ),
    .ddr3_cs_n      (ddr3_cs_n      ),
    .ddr3_dm        (ddr3_dm        ),
    .ddr3_odt       (ddr3_odt       )
);
```

1.5 模块使用说明

从模块的整体结构框图可以看出，里面包含 6 个子模块，其中 wr_ddr3_fifo 和 rd_ddr3_fifo 为双时钟 FIFO IP，PLL_SYS 为 PLL IP，DDR_CONTROLLER 是 DDR 控制器 IP，fifo_to_axi4 和 axi4_to_fifo 模块是芯路恒开发的接口转换模块。

1.5.1 DDR_CONTROLLER 模块

配置已经在前面讲解过了，这里就不再重复介绍，请自行查看。

1.5.2 PLL_SYS 模块

PLL IP，生成 DDR 控制器需要的时钟信号，分别是 400M、100M、50M，配置如下所示。

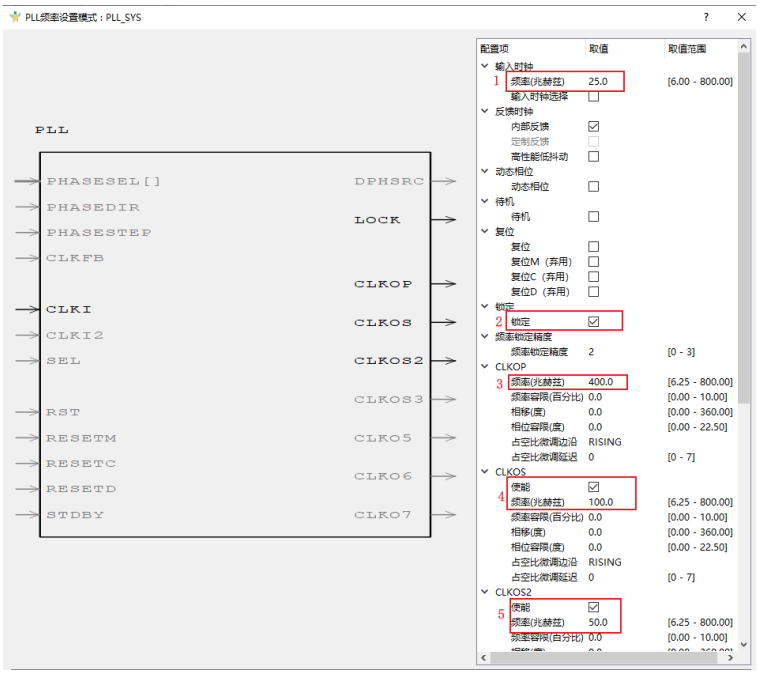


图 1-7 PLL IP 的配置界面

1.5.3 wr_ddr3_fifo

wr_ddr3_fifo 是双时钟 FIFO IP，我们进入 IP 管理器，选择存储单元中的 FIFO_Generator 命名为 wr_ddr3_fifo，进行配置如下所示：

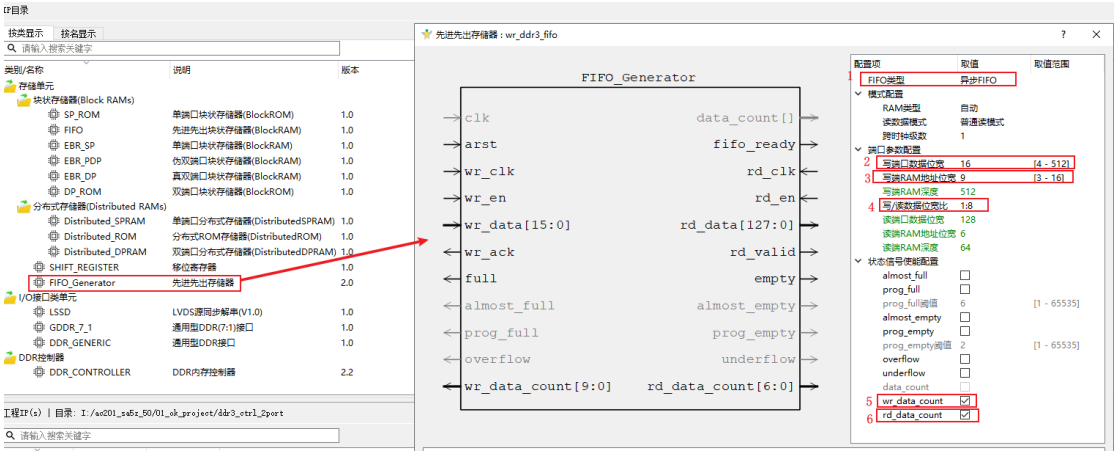


图 1-8 wr_ddr3_fifo 配置

1.5.4 rd_ddr3_fifo

rd_ddr3_fifo 也是双时钟 FIFO IP，具体配置如下：

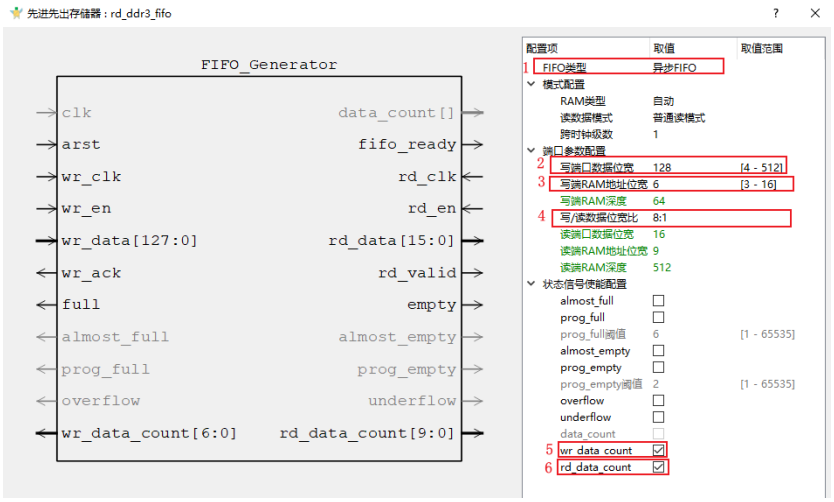


图 1-9 rd_ddr3_fifo 配置

1.5.5 fifo_to_axi4 转换模块

fifo_to_axi4 模块负责将写 fifo 中的数据通过 AXI4 接口发送出去，因此，这部分需要实现的是 AXI 接口的写事务部分。考虑模块设计实现的简单性（AXI 协议支持复杂的乱序读写操作等，这里就不做考虑），这里，我们将一次完成的写事务流程规定为①主机向写地址通道写入地址和控制信息——>②写数据通道突发写入数据——>③收到设备的写数据响应。

其中，突发写入数据又需要通过读取写 FIFO 得到，因此，结合以上内容，我们可以画出 fifo_to_axi4 模块写 AXI 接口操作的大致状态转移图。

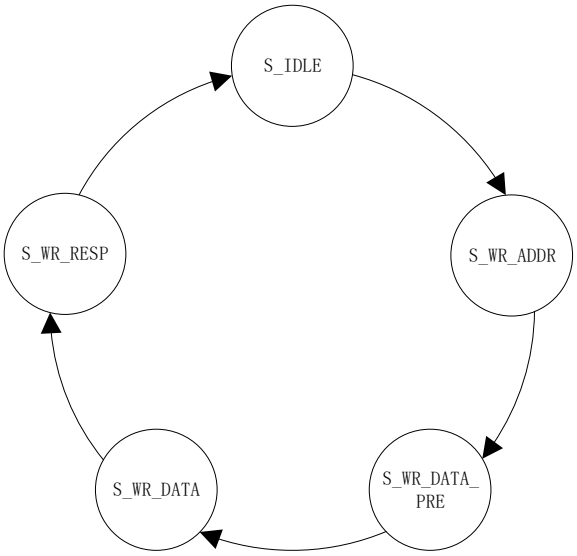


图 1-10 写操作状态转移图

这里的 S_WR_DATA_PRE 为预写数据态，代表从写 FIFO 中读取待突发的数据。状态机上电初始状态为 IDLE 状态，在该状态会判断是否可以开始写 DDR 操作，决定状态如何进行跳转；完成一次写操作流程后，状态回到 IDLE 状态进行下一次的操作。状态机采用三段式，第一二段的代码如下：

```
always@(posedge clk or posedge reset)
begin
    if(reset)
        curr_wr_state <= S_IDLE;
    else
        curr_wr_state <= next_wr_state;
end

always@(*)
begin
    case(curr_wr_state)
    //具体状态转移见下
    endcase
end
```

状态机在上电复位时处于初始状态 IDLE，在初始状态 IDLE 下，如果有写请求到来，则进入 AXI 写地址通道的操作状态 S_WR_ADDR。具体代码如下：

```
S_IDLE:
begin
    if(wr_ddr3_req == 1'b1)
        next_wr_state = S_WR_ADDR;
    else
        next_wr_state = S_IDLE;
end
```

在 S_WR_ADDR 状态，传输写操作的地址和控制信息，当 awready 和 awvalid 同时为高时表明地址已经传输成功，进入到读取数据的预写态。

```
S_WR_ADDR:
begin
    if(m_axi_awready && m_axi_awvalid)
        next_wr_state = S_WR_DATA_PRE;
    else
        next_wr_state = S_WR_ADDR;
end
```

预写数据态每次会读取 1 Beat（每一个 burst 可以被拆分为多次进行传输，每个被拆分的待传输数据称为 Beat；每个 Beat 被传输的过程称为 AXITransfer）数据，读取完成后马上便跳转到写操作的写数据通道操作。

```
S_WR_DATA_PRE:
begin
    next_wr_state = S_WR_DATA;
end
```

在写数据通道的操作中，每次发送 1 Beat 数据，发送完成后回到预写数据态读取数据，不断重复，直至发送完一次突发的最后一个数据后，进入到等待写响应的状态。

```
S_WR_DATA:
begin
if(m_axi_wready && m_axi_wvalid && m_axi_wlast)
    next_wr_state = S_WR_RESP;
else if(m_axi_wready && m_axi_wvalid)
    next_wr_state = S_WR_DATA_PRE;
else
    next_wr_state = S_WR_DATA;
end
```

在等待写响应状态，当主机接收到设备的写响应后，一次完整的写操作流程完成，状态回到 IDLE 状态等待下一次的操作。bresp 不同值表示不同的响应结果，bresp 为 2’b00 表示写数据成功，bid 需要与写地址通道传输的 awbid 一致。

```
S_WR_RESP:
begin
if(m_axi_bready && m_axi_bvalid && (m_axi_bresp == 2'b00) &&
(m_axi_bid == AXI_ID[AXI_ID_WIDTH-1:0]))
    next_wr_state = S_IDLE;
else
    next_wr_state = S_WR_RESP;
end
```

状态机设计完成后，剩下的就是在各个状态中产生各种信号。首先是写

DDR 请求信号的产生，这里我们用 wr_ddr3_req 表示。每当 FIFO 中有足够数据用于一次突发的时候，fifo_to_axi4 模块便能进行一次 DDR3 的写操作。因此，这里首先是通过突发长度计算出写 DDR3 所需的最少数据量（阈值），再通过阈值来判断 FIFO 中是否有足够的数据用来写 DDR3，进而产生写 DDR3 请求信号。

```
assign wr_req_cnt_thresh = (m_axi_awlen == 'd0) ? 1'b1 :
(AXI_BURST_LEN[7:0]+1'b1);
assign wr_ddr3_req       = (fifo_rst_busy == 1'b0) && (~fifo_empty) &&
(fifo_rd_cnt >= wr_req_cnt_thresh) ? 1'b1:1'b0;
```

然后是 AXI4 接口中，一些相对比较简单的信号，基本就是给固定值就可以：

```
assign m_axi_awid    = AXI_ID[AXI_ID_WIDTH-1:0];
assign m_axi_awsize  = DATA_SIZE;
assign m_axi_awburst = 2'b01;
assign m_axi_awlock  = 1'b0;
assign m_axi_awcache = 4'b0000;
assign m_axi_awprot  = 3'b000;
assign m_axi_awqos   = 4'b0000;
assign m_axi_awregion= 4'b0000;
assign m_axi_awlen   = AXI_BURST_LEN[7:0];
assign m_axi_wstrb   = 16'hffff;
assign m_axi_bready  = 1'b1;
```

而对于 AXI 接口中写事务相关的一些信号，在写操作的写地址通道比较关键的是产生 awaddr 和 awvalid。其中，awaddr 除了在复位和清除时变为起始地址外，在完成一次写操作流程后，地址就需要增加一次突发写入的数据量，需要注意的是这里的地址是以字节为单位的，则每次地址增加量应该是突发写数据个数*每个数据的字节数。这里每次突发长度为 AWLEN 加 1，每个数据字节数为 AXI_DATA_WIDTH/8，所以每完成一次写操作，地址增加(m_axi_awlen + 1'b1)*（AXI_DATA_WIDTH/8）。

```
always@(posedge clk or posedge reset)
begin
if(reset)
    m_axi_awaddr <= WR_AXI_BYTE_ADDR_BEGIN;
else if(wr_addr_clr || axi_awaddr_clr)
    m_axi_awaddr <= WR_AXI_BYTE_ADDR_BEGIN;
else if(m_axi_awaddr >= WR_AXI_BYTE_ADDR_END)
    m_axi_awaddr <= WR_AXI_BYTE_ADDR_BEGIN;
else if((curr_wr_state == S_WR_RESP) && m_axi_bready && m_axi_bvalid
&& (m_axi_bresp == 2'b00) && (m_axi_bid == AXI_ID[AXI_ID_WIDTH-1:0]))
    m_axi_awaddr <= m_axi_awaddr + ((m_axi_awlen + 1'b1)*
    (AXI_DATA_WIDTH/8));
else
```

```
    m_axi_awaddr <= m_axi_awaddr;
end
```

对于 awvalid 产生就相对简单些，在进入 WR_ADDR 状态到就将其输出为高，等到 awready 和 awvalid 同时高的时候，就将 awvalid 输出为低，保证 awready 和 awvalid 信号只有一个时钟周期的同时高。

```
always@(posedge clk or posedge reset)
begin
if(reset)
    m_axi_awvalid <= 1'b0;
else if((curr_wr_state == S_WR_ADDR) && m_axi_awready &&
m_axi_awvalid)
    m_axi_awvalid <= 1'b0;
else if(curr_wr_state == S_WR_ADDR)
    m_axi_awvalid <= 1'b1;
else
    m_axi_awvalid <= m_axi_awvalid;
end
```

在写操作的写数据通道比较关键的是产生 wvalid 和 wlast 信号。wvalid 用于表示写入的数据有效，对于 fifo_to_axi4 模块来说，数据是从 FIFO 读出，通过 AXI4 接口写入到 DDR3。又由于 FIFO 使用的是标准读模式，数据会在读使能信号后一拍被读出，因此，这里我们将 fifo 读请求信号打拍后，用来作为 wvalid 信号拉高的条件。一旦 wvalid 信号拉高，根据 AXI4 协议，应当一直保持不变，直到与 wready 信号握手成功，才能被拉低。所以，最终 wvalid 信号的代码实现如下：

```
always@(posedge clk)
begin
    fifo_rddata_valid <= fifo_rdreq;
end

always@(posedge clk or posedge reset)
begin
if(reset)
    m_axi_wvalid <= 1'b0;
else if(m_axi_wready && m_axi_wvalid)
    m_axi_wvalid <= 1'b0;
else if(fifo_rddata_valid)
    m_axi_wvalid <= 1'b1;
else
    m_axi_wvalid <= m_axi_wvalid;
end
```

wlast 信号是主机向设备传输最后一个数据的标识信号，这个信号的产生依赖于单次突发写入数据个数和当前已经传输的数据个数。主机在传输最后一个

数据同时将其输出为高，在发送完最后一个数据后立马将其输出为低。这个过程首先需要对传输数据个数进行计数，当 wready 和 m_axi_wvalid 同时为高时代表传输一个数据，传输数据个数计数器代码如下。

```
always@(posedge clk or posedge reset)
begin
if(reset)
    wr_data_cnt <= 1'b0;
else if(curr_wr_state == S_IDLE)
    wr_data_cnt <= 1'b0;
else if(m_axi_wready && m_axi_wvalid)
    wr_data_cnt <= wr_data_cnt + 1'b1;
else
    wr_data_cnt <= wr_data_cnt;
end
```

在产生 wlast 时，分两种情况，一是当突发写数据个数为 1，也就是 wlen 等于 0 时，那么传输的第一个数就是传输的最后一个数据，这种情况下，在每次传输时都将 wlast 拉高即可；二是当突发写数据个数大于 1，也就是 wlen 大于 0 时，就在传输完倒数第二个数（即 wr_data_cnt 为 m_axi_awlen -1'b1）后将 wlast 变为高电平。当最后一个数据传输完成（m_axi_wready、m_axi_wvalid 和 m_axi_wlast 同时为高电平）后将 wlast 变为低，具体代码如下。

```
always@(posedge clk or posedge reset)
begin
if(reset)
    m_axi_wlast <= 1'b0;
else if(m_axi_wready && m_axi_wvalid && m_axi_wlast)
    m_axi_wlast <= 1'b0;
else if(m_axi_awlen == 8'd0)
    m_axi_wlast <= 1'b1;
else if(m_axi_wready && m_axi_wvalid && (wr_data_cnt == m_axi_awlen -
1'b1))
    m_axi_wlast <= 1'b1;
else
    m_axi_wlast <= m_axi_wlast;
end
```

当然，除了写事物接口相关的信号外，还有预写数据态时，需要产生的写 FIFO 读请求信号。由于设计中使用的是标准模式 FIFO，数据会在读请求信号有效的一拍后输出，所以这里对 FIFO 读请求信号进行打拍，用来对数据进行锁存，确保每次通过写数据通道发送的数据是正确值。

```
always@(posedge clk or posedge reset)
begin
if(reset)
```

```
    fifo_rdreq <= 1'b0;
else if((curr_wr_state == S_WR_ADDR) && m_axi_awready &&
m_axi_awvalid)
    fifo_rdreq <= 1'b1;
else if((curr_wr_state == S_WR_DATA) && m_axi_wready && m_axi_wvalid
&& (~m_axi_wlast))
    fifo_rdreq <= 1'b1;
else
    fifo_rdreq <= 1'b0;
end

always@(posedge clk)
begin
    fifo_rddata_valid <= fifo_rdreq;
end

always@(posedge clk or posedge reset)
begin
if(reset)
    fifo_rddata_latch <= {AXI_DATA_WIDTH{1'b0}};
else if(fifo_rddata_valid)
    fifo_rddata_latch <= fifo_rddata;
end

assign m_axi_wdata   = fifo_rddata_latch;
```

那么这是 fifo_to_axi4 模块的实现，接下来我们再来看看 axi4_to_fifo 模块的代码实现。

1.5.6 axi4_to_fifo 转换模块

axi4_to_fifo 模块负责在读 FIFO 中有足够空间的时候，通过 AXI4 接口读取 DDR 中的数据，并将数据写进读 FIFO 中。也就是 axi4_to_fifo 模块只负责 AXI4 接口中读事务的实现。

而对于一次完整的读事务流程，这里我们可以规定为两步：

①主机向读地址通道写入地址和控制信息——>②收到设备的读数据响应和读的数据。

因此，axi4_to_fifo 模块读操作状态机设计如所示：

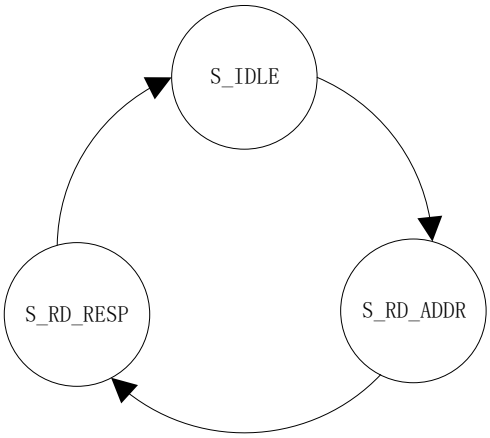


图 1-11 读操作状态转移图

上电初始状态为 IDLE 状态，在该状态根据 FIFO 是否有足够空间存储数据，决定状态如何跳转；完成一次读操作流程后，状态回到 IDLE 状态进行下一次的操作。状态机采用三段式，第一二段的代码如下。

```
always@(posedge clk or posedge reset)
begin
    if(reset)
        curr_state <= S_IDLE;
    else
        curr_state <= next_state;
end

always@(*)
begin
    case(curr_state)
    //具体状态转移见下
    endcase
end
```

状态机在上电复位处于初始状态 IDLE，在初始状态 IDLE 如果有读请求到来，则进入 AXI 读地址通道的操作状态 S_RD_ADDR，具体代码如下。

```
S_IDLE:
begin
if(rd_ddr3_req == 1'b1)
    next_state = S_RD_ADDR;
else
    next_state = S_IDLE;
end
```

在 S_RD_ADDR 状态，传输读操作的地址和控制信息，当 aready 和 arvalid 同时为高时表明地址已经传输完成，进入到读操作的读响应通道的操作。

```
S_RD_ADDR:
```

```
begin
if(m_axi_arready && m_axi_arvalid)
    next_state = S_RD_RESP;
else
    next_state = S_RD_ADDR;
end
```

在等待读响应状态，当主机接收到设备的读响应后，一次完整的读操作流程完成，状态回到仲裁状态进行下一次的操作，bresp 不同值表示不同的响应结果，bresp 为 2’b00 表示读数据成功，last 表示读取的最后一个数据的标识，rid 需要与读地址通道传输的 arid 一致。

```
S_RD_RESP:
begin
if(m_axi_rready && m_axi_rvalid && m_axi_rlast && (m_axi_rresp ==
2'b00) && (m_axi_rid == AXI_ID[AXI_ID_WIDTH-1:0]))
    next_state = S_IDLE;
else
    next_state = S_RD_RESP;
end
```

那么这是状态机的状态跳转，剩下的就是各个状态中产生的各种信号。与 fifo_to_axi4 模块类似，首先是 DDR 读请求信号的产生，这里我们用 rd_ddr3_req 表示。每当读 FIFO 中有足够空间存储一次突发数据，axi4_to_fifo 模块便会进行一次 DDR3 读操作。因此，这里首先计算出 FIFO 写计数的阈值，随后再将 FIFO 写计数与阈值作比较，产生对应的读 DDR3 请求信号。

```
assign rd_req_cnt_thresh = 2**FIFO_ADDR_WIDTH -
(AXI_BURST_LEN[7:0]+1'b1);
assign rd_ddr3_req   = (fifo_rst_busy == 1'b0) && (fifo_wr_cnt <
rd_req_cnt_thresh-2'd1) ? 1'b1:1'b0;
```

然后是 AXI4 接口中读事务相关信号，其中一些比较简单的信号，这里也是直接给固定值：

```
assign m_axi_arid    = AXI_ID[AXI_ID_WIDTH-1:0];
assign m_axi_arsize  = DATA_SIZE;
assign m_axi_arburst = 2'b01;
assign m_axi_arlock  = 1'b0;
assign m_axi_arcache = 4'b0000;
assign m_axi_arprot  = 3'b000;
assign m_axi_arqos   = 4'b0000;
assign m_axi_arregion= 4'b0000;
assign m_axi_arlen   = AXI_BURST_LEN[7:0];
assign m_axi_rready  = ~fifo_alfull;
```

接着是 AXI4 接口中相对复杂一点的信号，也就是写地址通道信号中比较关键的 araddr 和 arvalid 信号。产生过程与 awaddr 和 awvalid 类似。awaddr 除了在

复位和清除时变为起始地址外，在完成一次读操作流程后，地址需要增加一次突发写入的数据量。即每完成一次读操作，地址增加(m_axi_arlen + 1'b1)*（AXI_DATA_WIDTH/8），计算与写操作一样。具体代码如下：

```
always@(posedge clk or posedge reset)
begin
if(reset)
    m_axi_araddr <= RD_AXI_BYTE_ADDR_BEGIN;
else if(rd_addr_clr || axi_araddr_clr)
    m_axi_araddr <= RD_AXI_BYTE_ADDR_BEGIN;
else if(m_axi_araddr >= RD_AXI_BYTE_ADDR_END)
    m_axi_araddr <= RD_AXI_BYTE_ADDR_BEGIN;
else if((curr_rd_state == S_RD_RESP) && m_axi_rready && m_axi_rvalid
&& m_axi_rlast && (m_axi_rresp == 2'b00) && (m_axi_rid ==
AXI_ID[AXI_ID_WIDTH-1:0]))
    m_axi_araddr <= m_axi_araddr + ((m_axi_arlen +
1'b1)*(AXI_DATA_WIDTH/8));
else
    m_axi_araddr <= m_axi_araddr;
end

//m_axi_arvalid
always@(posedge clk or posedge reset)
begin
if(reset)
    m_axi_arvalid <= 1'b0;
else if((curr_state == S_RD_ADDR) && m_axi_arready && m_axi_arvalid)
    m_axi_arvalid <= 1'b0;
else if(curr_state == S_RD_ADDR)
    m_axi_arvalid <= 1'b1;
else
    m_axi_arvalid <= m_axi_arvalid;
end
```

除了这些信号外，还有读 FIFO 的写请求信号以及写数据的产生。

```
always@(posedge clk or posedge reset)
begin
    if(reset)
        begin
            fifo_wrreq  <= 1'b0;
            fifo_wrdata <= {AXI_DATA_WIDTH{1'b0}};
        end
    else if(addr_clr || axi_araddr_clr)
        begin
            fifo_wrreq  <= 1'b0;
            fifo_wrdata <= {AXI_DATA_WIDTH{1'b0}};
        end
```

```
    else if(m_axi_rvalid && m_axi_rready)
        begin
            fifo_wrreq  <= 1'b1;
            fifo_wrdata <= m_axi_rdata;
        end
    else
        begin
            fifo_wrreq  <= 1'b0;
            fifo_wrdata <= {AXI_DATA_WIDTH{1'b0}};
        end
end
```

至此关于 axi4_to_fifo 模块的设计就完成了，上述设计的完整代码可参考工程 fifo_axi4_adapter。

1.5.1 封装接口转换模块

根据设计框图可知，接口转换 fifo_axi4_adapter 模块由 fifo_to_axi4 模块、axi4_to_fifo 模块、rd_ddr3_fifo 模块和 wr_ddr3_fifo 模块组成。只需要简单的进行端口信号连接即可。具体代码如下：

```
module fifo_axi4_adapter #(
  parameter FIFO_DW                = 16     ,
  parameter WR_AXI_BYTE_ADDR_BEGIN = 0      ,
  parameter WR_AXI_BYTE_ADDR_END   = 1023   ,
  parameter RD_AXI_BYTE_ADDR_BEGIN = 0      ,
  parameter RD_AXI_BYTE_ADDR_END   = 1023   ,

  parameter AXI_DATA_WIDTH         = 128    ,
  parameter AXI_ADDR_WIDTH         = 32     ,
  parameter AXI_ID_WIDTH           = 3      ,
  parameter AXI_ID                 = 4'b0000,
  parameter AXI_BURST_LEN          = 8'd31  , //burst length =
AXI_BURST_LEN+1
  parameter FIFO_ADDR_DEPTH        = 64
)
(
  // clock reset
  input                            clk           ,
  input                            reset         ,
  // wr_fifo wr Interface
  input                            wrfifo_clr    ,
  input                            wrfifo_clk    ,
  input                            wrfifo_wren   ,
  input   [FIFO_DW-1:0]            wrfifo_din    ,
  output                           wrfifo_full   ,
  output  [15:0]                   wrfifo_wr_cnt ,
```

```
    // rd_fifo rd Interface
    input                           rdfifo_clr    ,
    input                           rdfifo_clk    ,
    input                           rdfifo_rden   ,
    output  [FIFO_DW-1:0]           rdfifo_dout   ,
    output                          rdfifo_empty  ,
    output  [15:0]                  rdfifo_rd_cnt ,
    // Master Interface Write Address Ports
    output  [AXI_ID_WIDTH-1:0]      m_axi_awid    ,
    output  [AXI_ADDR_WIDTH-1:0]    m_axi_awaddr  ,
    output  [7:0]                   m_axi_awlen   ,
    output  [2:0]                   m_axi_awsize  ,
    output  [1:0]                   m_axi_awburst ,
    output  [0:0]                   m_axi_awlock  ,
    output  [3:0]                   m_axi_awcache ,
    output  [2:0]                   m_axi_awprot  ,
    output  [3:0]                   m_axi_awqos   ,
    output  [3:0]                   m_axi_awregion,
    output                          m_axi_awvalid ,
    input                           m_axi_awready ,
    // Master Interface Write Data Ports
    output  [AXI_DATA_WIDTH-1:0]    m_axi_wdata   ,
    output  [AXI_DATA_WIDTH/8-1:0]  m_axi_wstrb   ,
    output                          m_axi_wlast   ,
    output                          m_axi_wvalid  ,
    input                           m_axi_wready  ,
    // Master Interface Write Response Ports
    input   [AXI_ID_WIDTH-1:0]      m_axi_bid     ,
    input   [1:0]                   m_axi_bresp   ,
    input                           m_axi_bvalid  ,
    output                          m_axi_bready  ,
    // Master Interface Read Address Ports
    output  [AXI_ID_WIDTH-1:0]      m_axi_arid    ,
    output  [AXI_ADDR_WIDTH-1:0]    m_axi_araddr  ,
    output  [7:0]                   m_axi_arlen   ,
    output  [2:0]                   m_axi_arsize  ,
    output  [1:0]                   m_axi_arburst ,
    output  [0:0]                   m_axi_arlock  ,
    output  [3:0]                   m_axi_arcache ,
    output  [2:0]                   m_axi_arprot  ,
    output  [3:0]                   m_axi_arqos   ,
    output  [3:0]                   m_axi_arregion,
    output                          m_axi_arvalid ,
    input                           m_axi_arready ,
    // Master Interface Read Data Ports
    input   [AXI_ID_WIDTH-1:0]      m_axi_rid     ,
    input   [AXI_DATA_WIDTH-1:0]    m_axi_rdata   ,
```

```
  input   [1:0]                   m_axi_rresp   ,
  input                           m_axi_rlast   ,
  input                           m_axi_rvalid  ,
  output                          m_axi_rready
);

  localparam FIFO_ADDR_WIDTH = clogb2(FIFO_ADDR_DEPTH-1);

  wire                        wrfifo_rden;
  wire [AXI_DATA_WIDTH-1:0] wrfifo_dout;
  wire [FIFO_ADDR_WIDTH-1:0]  wrfifo_rd_cnt;
  wire                        wrfifo_empty;
  wire                        wrfifo_wr_rst_busy;
  wire                        wrfifo_rd_rst_busy;

  wire                        rdfifo_wren;
  wire [AXI_DATA_WIDTH-1:0] rdfifo_din;
  wire [FIFO_ADDR_WIDTH-1:0]  rdfifo_wr_cnt;
  wire                        rdfifo_full;
  wire                        rdfifo_wr_rst_busy;
  wire                        rdfifo_rd_rst_busy;

  reg                         wrfifo_clr_sync_clk;
  reg                         wr_addr_clr;
  reg                         rdfifo_clr_sync_clk;
  reg                         rd_addr_clr;

  wire fifo_ready;
  wr_ddr3_fifo wr_ddr3_fifo
  (
      .arst(wrfifo_clr),
      .wr_clk(wrfifo_clk),
      .wr_en(wrfifo_wren),
      .wr_data(wrfifo_din),
      .rd_clk(clk),
      .rd_en(wrfifo_rden),
      .fifo_ready(fifo_ready),
      .wr_ack(),
      .wr_data_count(wrfifo_wr_cnt),
      .full(wrfifo_full),
      .rd_data(wrfifo_dout),
      .rd_valid(),
      .rd_data_count(wrfifo_rd_cnt),
      .empty(wrfifo_empty)
  );
  assign wrfifo_rd_rst_busy = ~fifo_ready;
```

```
  wire rd_fifo_ready;
  rd_ddr3_fifo rd_ddr3_fifo
  (
      .arst(rdfifo_clr),
      .wr_clk(clk),
      .wr_en(rdfifo_wren),
      .wr_data(rdfifo_din),
      .rd_clk(rdfifo_clk),
      .rd_en(rdfifo_rden),
      .fifo_ready(rd_fifo_ready),
      .wr_ack(),
      .wr_data_count(rdfifo_wr_cnt),
      .full(rdfifo_full),
      .rd_data(rdfifo_dout),
      .rd_valid(),
      .rd_data_count(rdfifo_rd_cnt),
      .empty(rdfifo_empty)
  );

  assign rdfifo_wr_rst_busy = ~rd_fifo_ready;

  always@(posedge clk)
  begin
    wrfifo_clr_sync_clk <= wrfifo_clr;
    wr_addr_clr <= wrfifo_clr_sync_clk;
  end

  always@(posedge clk)
  begin
    rdfifo_clr_sync_clk <= rdfifo_clr;
    rd_addr_clr <= rdfifo_clr_sync_clk;
  end

  fifo_to_axi4
  #(
    .WR_AXI_BYTE_ADDR_BEGIN (WR_AXI_BYTE_ADDR_BEGIN ),
    .WR_AXI_BYTE_ADDR_END   (WR_AXI_BYTE_ADDR_END   ),

    .AXI_DATA_WIDTH         (AXI_DATA_WIDTH         ),
    .AXI_ADDR_WIDTH         (AXI_ADDR_WIDTH         ),
    .AXI_ID_WIDTH           (AXI_ID_WIDTH           ),
    .AXI_ID                 (AXI_ID                 ),
    .AXI_BURST_LEN          (AXI_BURST_LEN          ),//burst length =
AXI_BURST_LEN+1
    .FIFO_ADDR_WIDTH        (FIFO_ADDR_WIDTH        )
  )fifo_to_axi4_inst
  (
```

```
    //clock reset
    .clk                    (clk                    ),
    .reset                  (reset                  ),
    //FIFO Interface ports
    .addr_clr               (wr_addr_clr            ), //1:clear, sync
clk
    .fifo_rdreq             (wrfifo_rden            ),
    .fifo_rddata            (wrfifo_dout            ),
    .fifo_empty             (wrfifo_empty           ),
    .fifo_rd_cnt            (wrfifo_rd_cnt          ),
    .fifo_rst_busy          (wrfifo_rd_rst_busy     ),
    // Slave Interface Write Address Ports
    .m_axi_awid             (m_axi_awid             ),
    .m_axi_awaddr           (m_axi_awaddr           ),
    .m_axi_awlen            (m_axi_awlen            ),
    .m_axi_awsize           (m_axi_awsize           ),
    .m_axi_awburst          (m_axi_awburst          ),
    .m_axi_awlock           (m_axi_awlock           ),
    .m_axi_awcache          (m_axi_awcache          ),
    .m_axi_awprot           (m_axi_awprot           ),
    .m_axi_awqos            (m_axi_awqos            ),
    .m_axi_awregion         (m_axi_awregion         ),
    .m_axi_awvalid          (m_axi_awvalid          ),
    .m_axi_awready          (m_axi_awready          ),
    // Slave Interface Write Data Ports
    .m_axi_wdata            (m_axi_wdata            ),
    .m_axi_wstrb            (m_axi_wstrb            ),
    .m_axi_wlast            (m_axi_wlast            ),
    .m_axi_wvalid           (m_axi_wvalid           ),
    .m_axi_wready           (m_axi_wready           ),
    // Slave Interface Write Response Ports
    .m_axi_bid              (m_axi_bid              ),
    .m_axi_bresp            (m_axi_bresp            ),
    .m_axi_bvalid           (m_axi_bvalid           ),
    .m_axi_bready           (m_axi_bready           )
  );

  axi4_to_fifo
  #(
    .RD_AXI_BYTE_ADDR_BEGIN (RD_AXI_BYTE_ADDR_BEGIN ),
    .RD_AXI_BYTE_ADDR_END   (RD_AXI_BYTE_ADDR_END   ),

    .AXI_DATA_WIDTH         (AXI_DATA_WIDTH         ),
    .AXI_ADDR_WIDTH         (AXI_ADDR_WIDTH         ),
    .AXI_ID_WIDTH           (AXI_ID_WIDTH           ),
    .AXI_ID                 (AXI_ID                 ),
```

```
    .AXI_BURST_LEN          (AXI_BURST_LEN          ),//burst length =
AXI_BURST_LEN+1
    .FIFO_ADDR_WIDTH        (FIFO_ADDR_WIDTH        )
  )axi4_to_fifo_inst
  (
    //clock reset
    .clk                    (clk                    ),
    .reset                  (reset                  ),
    //FIFO Interface ports
    .addr_clr               (rd_addr_clr            ), //1:clear, sync
clk
    .fifo_wrreq             (rdfifo_wren            ),
    .fifo_wrdata            (rdfifo_din             ),
    .fifo_alfull            (rdfifo_full            ),
    .fifo_wr_cnt            (rdfifo_wr_cnt          ),
    .fifo_rst_busy          (rdfifo_wr_rst_busy     ),
    // Slave Interface Read Address Ports
    .m_axi_arid             (m_axi_arid             ),
    .m_axi_araddr           (m_axi_araddr           ),
    .m_axi_arlen            (m_axi_arlen            ),
    .m_axi_arsize           (m_axi_arsize           ),
    .m_axi_arburst          (m_axi_arburst          ),
    .m_axi_arlock           (m_axi_arlock           ),
    .m_axi_arcache          (m_axi_arcache          ),
    .m_axi_arprot           (m_axi_arprot           ),
    .m_axi_arqos            (m_axi_arqos            ),
    .m_axi_arregion         (m_axi_arregion         ),
    .m_axi_arvalid          (m_axi_arvalid          ),
    .m_axi_arready          (m_axi_arready          ),
    // Slave Interface Read Data Ports
    .m_axi_rid              (m_axi_rid              ),
    .m_axi_rdata            (m_axi_rdata            ),
    .m_axi_rresp            (m_axi_rresp            ),
    .m_axi_rlast            (m_axi_rlast            ),
    .m_axi_rvalid           (m_axi_rvalid           ),
    .m_axi_rready           (m_axi_rready           )
  );


  //*********************************
  //The following function calculates the awsize/arsize width based on
AXI_DATA_WIDTH
  //*********************************
  function integer clogb2;
    input integer axi_data_byte;
      for (clogb2=0; axi_data_byte>0; clogb2=clogb2+1)
        axi_data_byte = axi_data_byte >> 1;
```

```
  endfunction
endmodule
```

至此，我们便完成了 AXI 接口转换模块的设计，为了验证设计是否能够正常工作，接下来我们还需要对模块进行仿真。

1.6 fifo_axi4_adapter 模块仿真

将上面 fifo_axi4_adapter 模块设计内容进行分析和综合直至没有错误以及警告。新建名称为 fifo_axi4_adapter_tb 的仿真文件。为了使仿真更加贴近我们的系统设计，这里 AXI 接口的仿真使用 DDR 控制器 MIG IP 和 DDR 仿真模型，具体仿真 testbench 设计的结构框图如下。

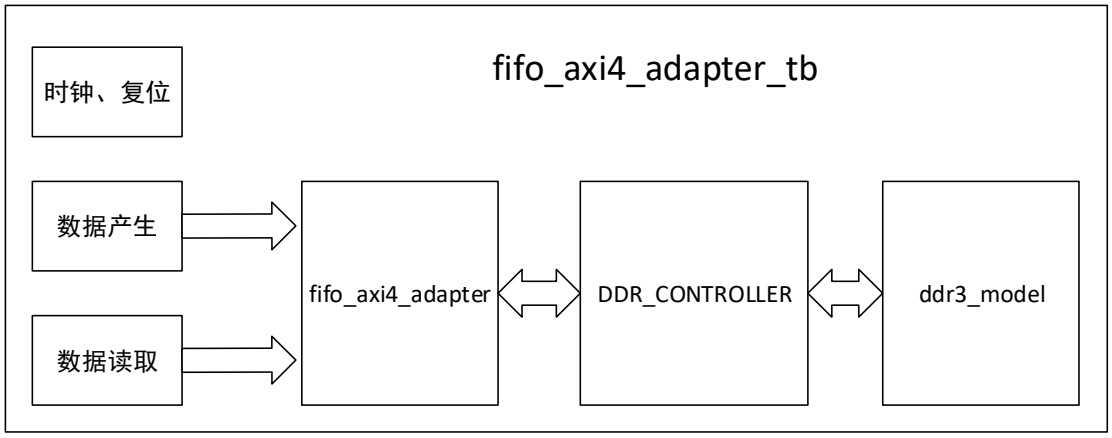


图 1-12 fifo_axi4_adapter_tb 仿真架构

这里仿真文件中除了需要例化 fifo_axi4_adapter 模块，还需要分别例化 DDR_CONTROLLER 模块以及 DDR3 仿真模型。DDR_CONTROLLER 模块就是 DDR 控制器 MIG IP。我们在前面章节也介绍过该 IP 核的配置方法，这里就不重复讲解了。还需要例化 PLL IP，产生 DDR 控制器需要的时钟，需要产生 PLL 的输入时钟 25M，如下所示：

```
initial clk_osc = 1'b1;
always #20 clk_osc = ~clk_osc;
```

数据产生的激励设计上将其封装成任务 task 形式，方便调用。具体代码如下。该任务的具体功能是在调用这个任务和给定的输入参数 data_begin 和 wr_data_cnt 时，向 wr_ddr3_fifo 中写入以起始数据 data_begin 开始递增的 wr_data_cnt 个数据。

```
task wr_data;
    input [15:0]data_begin;
    input [15:0]wr_data_cnt;
    begin
```

```
    wrfifo_wren = 1'b0;
    wrfifo_din  = data_begin;
    @(posedge wrfifo_clk);
    #1 wrfifo_wren = 1'b1;
    repeat(wr_data_cnt)
    begin
        @(posedge wrfifo_clk);
        #1 wrfifo_din = wrfifo_din + 1'b1;
    end
    #1 wrfifo_wren = 1'b0;
    end
endtask
```

数据读取的激励设计上采用与数据产生类似的方式，封装成任务 task 形式，具体代码如下。任务的具体功能是在调用这个任务和给定的输入参数 rd_data_cnt 时，向 rd_ddr3_fifo 中读出个 rd_data_cnt 数据。同时为了验证读出数据的正确性，task 输入给定预期读出第一个数据数值，内部根据第一个数据产生接下来期望读出数据，同时对实际读出数据和期望读出数据做比较，如果出现实际读出数据不等于期望读出数据，则提前结束仿真并打印“SIM is failed”信息。

```
task rd_data;
    input [15:0]data_begin;
    input [15:0]rd_data_cnt;
    begin
        rdfifo_rden = 1'b0;
        expect_rd_data = data_begin;
        @(posedge rdfifo_clk);
        #1 rdfifo_rden = 1'b1;
        repeat(rd_data_cnt)
        begin
        @(posedge rdfifo_clk);
        #1;
        if(rdfifo_dout != expect_rd_data)
        begin
            $display("SIM is failed");
            $finish;
        end
        expect_rd_data = expect_rd_data + 1'b1;
        end
        #1 rdfifo_rden = 1'b0;
    end
endtask
```

仿真中加入了 DDR 控制器 IP 模块，在写入数据前需要等 DDR 控制器初始化完成。仿真设计上整体流程是，先产生 DDR 控制器的复位以及 FIFO 的复位。

等 DDR 控制器内部锁相环锁定后，延时 200ns 后对 FIFO 解复位。等待 DDR 初始化校准完成后，延时 200n 往 wr_ddr3_fifo 写入 1024 个数据，数据写完后，对 rd_ddr3_fifo 进行复位，清空里面的缓存，等 FIFO 复位结束一段时间后，等待 rd_ddr3_fifo 非空后开始对 rd_ddr3_fifo 进行读取数据，读取 1024 个数据。整个仿真结束后打印“SIM is successfully”信息，具体代码如下。

```
initial begin
    sys_rst = 1'b0;
    aresetn = 1'b0;
    expect_rd_data = 16'd0;
    wrfifo_clr  = 1'b1;
    wrfifo_wren = 1'b0;
    wrfifo_din  = 16'd0;
    rdfifo_clr  = 1'b1;
    rdfifo_rden = 1'b0;
    #201;
    sys_rst = 1'b1;
    aresetn = 1'b1;
    #200;
    wrfifo_clr  = 1'b0;
    rdfifo_clr  = 1'b0;
    @(posedge init_status[1]);
    #200;

    wr_data(SIM_DATA_BEGIN,SIM_DATA_CNT);
    #2000;
    rdfifo_clr  = 1'b1;
    #20;
    rdfifo_clr  = 1'b0;
    #2000;
    wait(rdfifo_empty == 1'b0)
    rd_data(SIM_DATA_BEGIN,SIM_DATA_CNT);

    #5000;
    $display("SIM is successfully");
    $stop;
end
```

然后使用 Modelsim 建立仿真工程，添加文件，得到仿真波形如下所示：

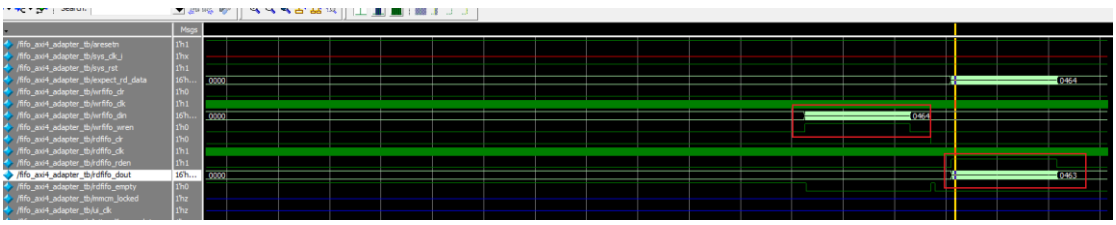


图 1-13 仿真整体效果图

在初始化校准完成信号变高后，就立马产生了一次读 DDR 操作，同时读出数据写入到 rd_ddr3_fifo，这个时候从 DDR 读出的数据是无效的，因为在此之前还没有往 DDR 里存入数据。这就是为什么在仿真上在往 DDR 存储数据后，读数据之前先对 rd_ddr3_fifo 进行一次复位清空缓存的操作，同时对把 DDR 的地址也复位到起始地址。这样可保证读取的数据都是从 DDR 起始地址开始，同时保证读之前读 FIFO 没有无效的缓存数据。在串口传图系统上使用也是采用仿真类似的方式进行的，为了达到显示屏显示图像的帧同步（不出现显示偏移问题），会在显示每帧数据前将读 FIFO 模块 rd_ddr3_fifo 进行一次复位清空缓存和读 DDR 操纵的地址复位到起始。保证每次屏幕显示图像数据都是从 DDR 内存中起始地址开始读出的数据。

之后再往写 FIFO 写入 1024 个数据（位宽为 16bit）过程中，AXI 总线上产生了 4 次突发写，每次突发写是 32 个数据（位宽为 128bit），这个与预期的突发写次数（写 FIFO 数据个数*FIFO 写数据位宽/AXI 数据位宽/AXI 一次突发写数据个数 = 1024*16/128/32 =4，根据这个公式，写 FIFO 数据个数不是随意设置，否则会出现不能满足写入数据正好是 AXI 突发写数据个数的整数被）一致。

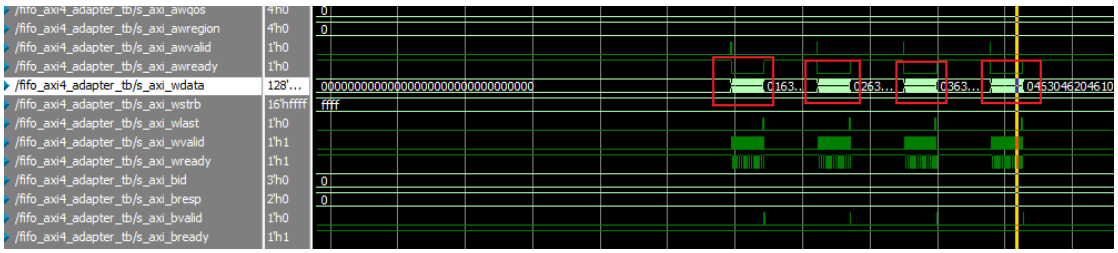


图 1-14 突发写 4 次的仿真体现

对第一次突发写 DDR 操作部分 AXI 写事务波形放大来看。

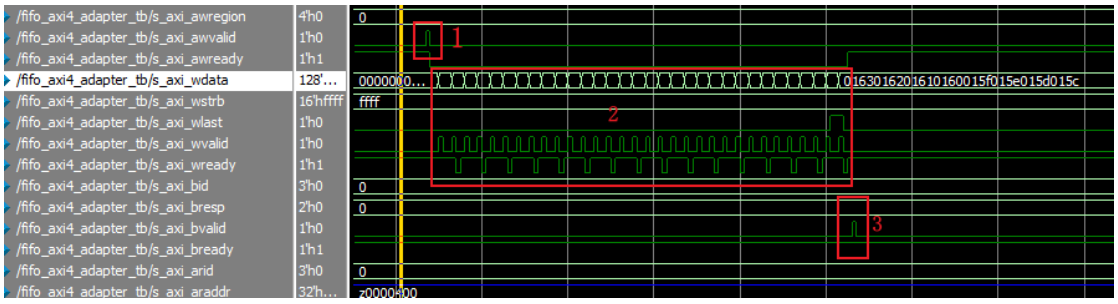


图 1-15 第一次突发写的放大观察

从波形图可以看到一次 AXI 写数据操作流程如下：

（1）在地址通道写入接下来要写入数据的起始地址和写突发长度等信息，在 awready 和 awvalid 同时为高时，这些信息被传输给 DDR 控制器；

（2）在写数据通道写入指定突发长度的数据，当 wready 和 wvalid 同时为高时表示写入数据被传输给 DDR 控制器，在写最后一个数据时，wlast 变为高，写完最后一个 wlast 变为低；

（3）在写响应通道等待设备的写响应，当 bready 和 bvalid 同时为高时表示响应的到来。

一次操作过程中，数据写入流程与我们设计预期是一致的。同时可以对写入的数据波形放大了看，写入的每个数据是由写 FIFO 写入 8 个 16bit 数据拼接而成的 128bit 数据，具体拼接过程的位宽转换是由写 FIFO 完成。可以通过类似方法去对其他几次的写操作波形进行观察，从而验证设计的正确性。

将 1024 个数据（位宽为 16bit）写完之后，对读 FIFO 进行一次清零，清零的作用前面已经说过。等待一段时间后，对读 FIFO 进行读数据操作，读取前面写入的 1024 个数据。读数据过程中，AXI 总线上就会产生突发读的操作。

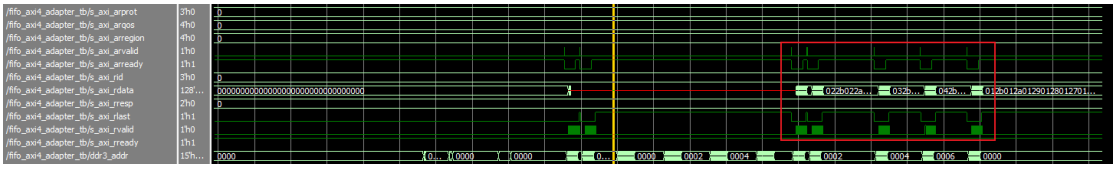


图 1-16 突发读操作

对红色圈中第一次突发读 DDR 操作部分 AXI 读事务波形放大来看。

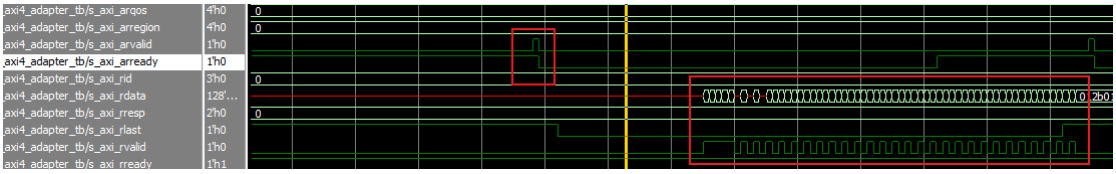


图 1-17 第一次突发读操作放大观察

（1）在地址通道写入接下来要读取数据的起始地址和读突发长度等信息，在 arready 和 arvalid 同时为高时，这些信息被传输给 DDR 控制器；

（2）在读响应通道，当 rready 和 rvalid 同时为高时表示读出的有效数据，当 rlast 变为高时表示读出的最后一个数据。图中波形 rdata 有部分地方数据是未知，这些地方并非是有效数据地方，可不用关心。

从整体上看，写 FIFO 中写入数据是 100~1123 的 1024 个连续递增数据，波形与仿真代码是一致的，下图是整个写入数据和写入第一个数据与最后一个数据的波形图。

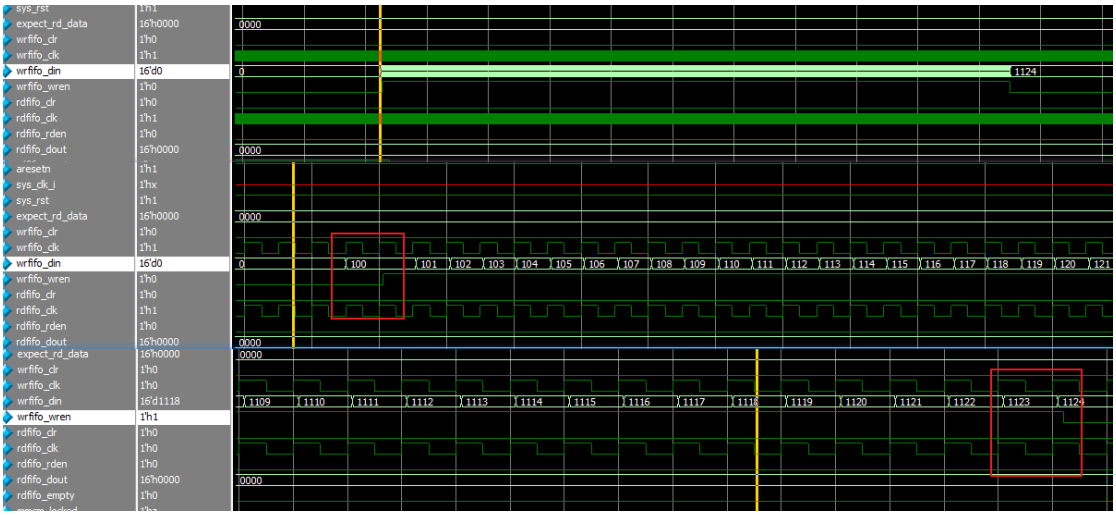


图 1-18 写入数据整体效果与关键信息

读 FIFO 中读出的数据同样也是 100~1123 的 1024 个连续递增数据，如下所示。

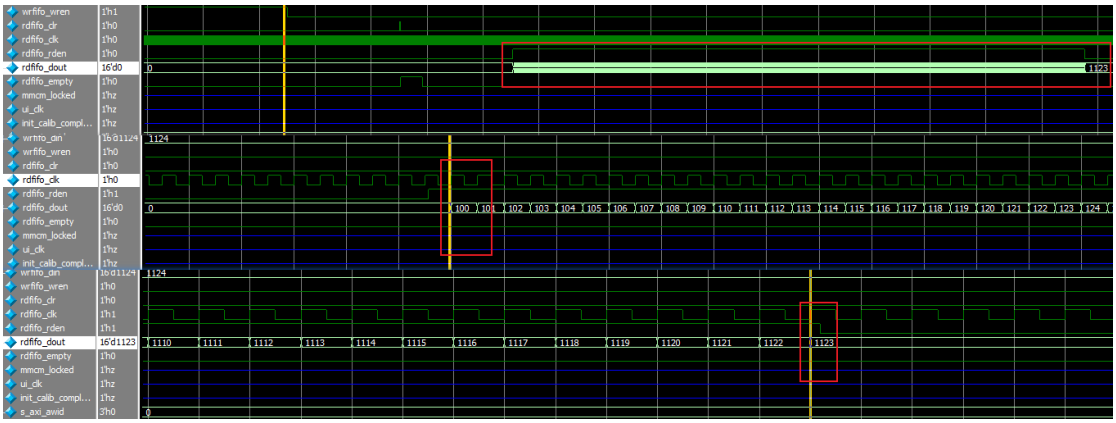


图 1-19 读出数据整体效果与关键信息

从写入数据与最后读出数据的一致性说明 fifo_axi4_adapter 模块功能是正常的。

1.7 ddr3_ctrl_2port 模块仿真说明

在上一小节完成了 fifo_axi4_adapter 设计及仿真后，我们给上述各模块添加顶层，并对顶层端口再进一步进行仿真。

双端口 FIFO DDR3 控制器模块的仿真需要注意的是，需将 DDR3 仿真模型（即 ddr3_model 模块）例化到 tb 文件中。DDR3 仿真模型充当的是实际硬件存储器的代表，它的作用在于模拟 DDR3 硬件存储器，和新设计的二端口控制器实现数据交互。

例化框图可参考如下：

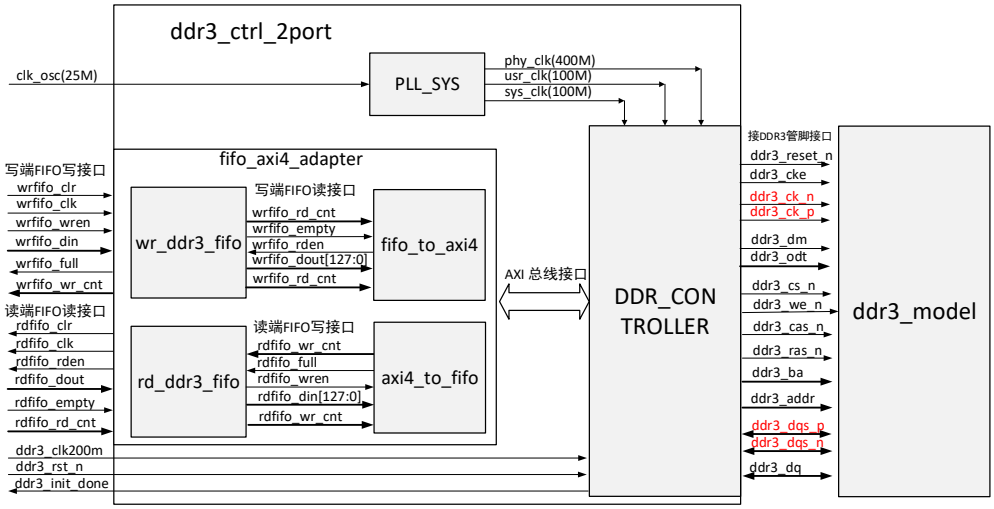


图 1-20 DDR3 例化模型

如果我们在设计中需要使用 ddr3 模型，只需要将接 ddr3 管脚接口的信号与 ddr3_model 模型端口信号一一连接即可。我们将 ddr3_model.v 文件和 ddr3_model_parameters.vh 文件复制粘贴到需要仿真的工程下，然后添加仿真文件 ddr3_model.v。相关代码名称为：

ddr3_model.sv

ddr3_model_parameters.vh

关于 ddr3_ctrl_2port 模块和模块仿真文件 ddr3_ctrl_2port_tb 的代码名称为：

设计文件：ddr3_ctrl_2port.v

仿真文件：ddr3_ctrl_2port_tb.v

下图是对 ddr3_ctrl_2port 模块仿真的波形图，注意波形中 ddr3_init_done 信号大约在 0.5ms 之后某一时刻出现由低电平变高电平，如果超过这个很长时间没有变为高电平，那说明在仿真层面 DDR3 初始化可能存在问题，需要检查 ddr3_model 模块是否例化好，相关信号的数据位宽是否正确。

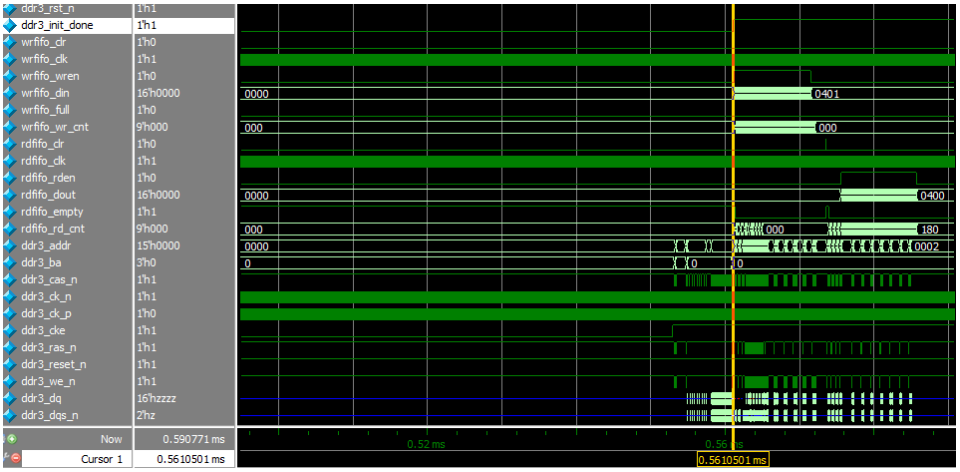


图 1-21 DDR3 仿真波形

从最终顶层输出端口来看，仿真结果也是能较好的体现本次设计的二端口 DDR3 控制器的读写功能，这样，我们整个二端口 DDR3 控制器模块的设计就完成了。

2 基于 DDR3 的串口传图帧缓存系统设计实现

章节导读

本章将基于前面讲解的设计内容，继续讲解串口传图的基本内容。本章在 FPGA 学习内容中，处于立交桥的地位。本章的内容全面涵盖输入、缓存、输出，同时本章的内容又是基于 FPGA 片上图片缓存的内容深化。学完本章以后，既可以基于本章内容实现各种类型千变万化输入输出接口的替换设计，又可以在本章的数据接收与处理环节作文章，进行图像处理内容的理解与学习。

2.1 系统整体设计

本章我们将设计一个基于 DDR3 的串口传图帧缓存系统。该系统的整体设计框图如下。

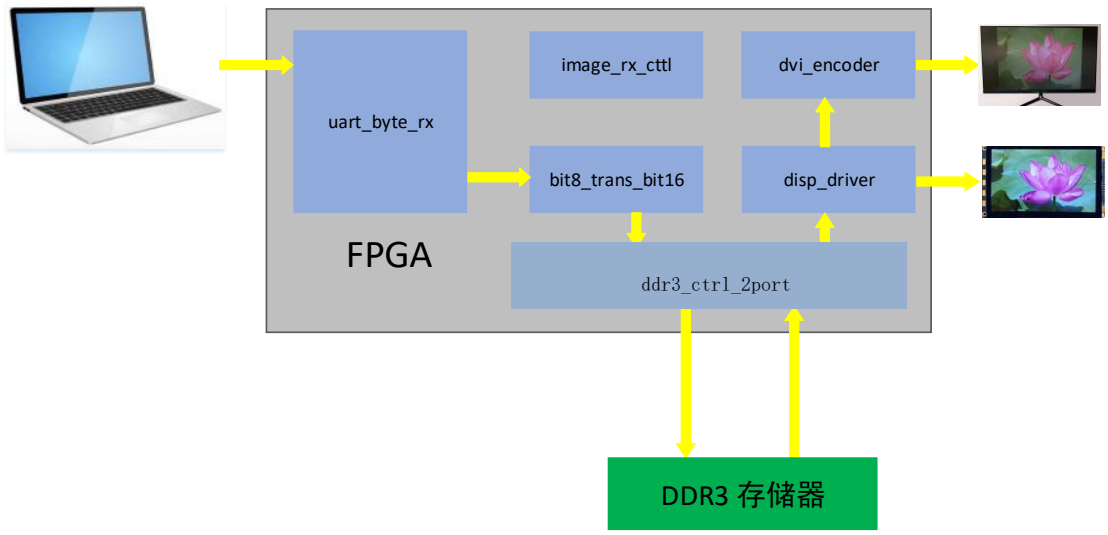


图 2-1 串口传图顶层系统设计框图

其中，

（1）uart_byte_rx 模块:负责串口图像数据的接收，该模块的设计前面章节已经有讲。

（2）bit8_trans_bit16 模块：将串口接收的每两个 8bit 数据转换成一个 16bit 数据（图像数据是 16bit 的 RGB565 的数据，电脑是通过串口将一个像素点数据分两次发送到 FPGA，FPGA 需将串口接收数据重组成 16bit 的图像数据），实现过程相对比较简单。

（3）disp_driver 模块：tft 屏显示驱动控制，对缓存在 DDR3 中的图像数据

进行显示。

（4）ddr3_ctrl_2port 模块组：包含 wr_ddr3_fifo、rd_ddr3_fifo、fifo_to_axi4、axi4_to_fifo 以及 DDR_CONTROLLER 模块。完成采集的图像数据缓存。其中，wr_ddr3_fifo、rd_ddr3_fifo、fifo_to_axi4、axi4_to_fifo 已集成为 fifo_axi4_adapter。

（5）pll 模块：上述各个模块所需时钟的产生，使用 PLL IP。

（6）dvi_encoder 模块：用于将生成信号转换成 HDMI 输出信号作 HDMI 显示。

除去使用 IP 和前面章节讲过的模块（组）外，还需要设计 bit8_trans_bit16 模块和 image_rx_ctrl 模块。

2.2 接收控制模块设计(image_rx_ctrl)

为了对接收的信号进行分析和整理，我们设计了接收控制模块，主要内容是描述图像的行场同步信号生成。

```
module   image_rx_ctrl # (
    parameter DISP_WIDTH  = 800 ,
    parameter DISP_HEIGHT = 480
   )
   (
    input clk,
    input reset_p,
    input  image_data_valid,
    output reg [15:0]image_data_hcnt,
    output reg [15:0]image_data_vcnt,
    output reg image_data_hs,
    output reg image_data_vs,
    output reg frame_rx_done_flip
   );

     //generate image data hs or vs
  always@(posedge clk or posedge reset_p)
    if(reset_p)
      image_data_hcnt <= 'd0;
    else if(image_data_valid) begin
      if(image_data_hcnt == (DISP_WIDTH - 1'b1))
        image_data_hcnt <= 'd0;
      else
        image_data_hcnt <= image_data_hcnt + 1'b1;
    end
```

```
  always@(posedge clk or posedge reset_p)
    if(reset_p)
      image_data_vcnt <= 'd0;
    else if(image_data_valid) begin
      if(image_data_hcnt == (DISP_WIDTH - 1'b1)) begin
        if(image_data_vcnt == (DISP_HEIGHT - 1'b1))
          image_data_vcnt <= 'd0;
        else
          image_data_vcnt <= image_data_vcnt + 1'b1;
      end
    end
  //hs
  always@(posedge clk or posedge reset_p)
    if(reset_p)
      image_data_hs <= 1'b0;
    else if(image_data_valid && image_data_hcnt == (DISP_WIDTH - 1'b1))
      image_data_hs <= 1'b0;
    else
      image_data_hs <= 1'b1;
  //vs
  always@(posedge clk or posedge reset_p)
    if(reset_p)
      image_data_vs <= 1'b0;
    else if(image_data_valid && image_data_hcnt == (DISP_WIDTH - 1'b1) &&
            image_data_vcnt == (DISP_HEIGHT - 1'b1))
      image_data_vs <= 1'b0;
    else
      image_data_vs <= 1'b1;

  always@(posedge clk or posedge reset_p)
    if(reset_p)
      frame_rx_done_flip <= 1'b0;
    else if(image_data_valid && image_data_hcnt == (DISP_WIDTH - 1'b1) &&
            image_data_vcnt == (DISP_HEIGHT - 1'b1))
      frame_rx_done_flip <= ~frame_rx_done_flip;
endmodule
```

在该模块之中，完成了图像行同步信号描述及计数，图像场同步信号描述及计数，图像接收完成信号描述。通过这些信息，就可以定位图像传输的进度。同时，如果图像传输完成，则给出传输完成的 led 点亮信号。

2.3 位宽转换模块设计

bit8_trans_bit16 该模块主要功能是将串口传输过来（一次传 8bit）的每两个 8bit 数据拼接成一个 16bit 图像数据（RGB565）。功能相对比较简单。该模块的

接口图及接口功能描述如下。

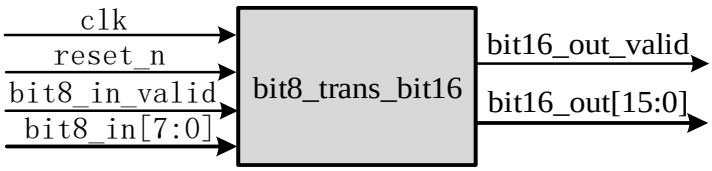


图 2-2 bit8_trans_bit16 模块接口图

表 2-1 bit8_trans_bit16 模块接口功能描述

接口名称	I/O	功能描述
clk	I	模块工作时钟
reset_n	I	模块复位，低有效
bit8_in[7:0]	I	8bit 数据输入
bit8_in_valid	I	8bit 数据有效标识
bit16_out[15:0]	O	转换后 16bit 数据输输出
bit16_out_valid	O	转换后 16bit 数据有效标识

模块主要信号设计的时序要求如下。对输入的每两个 8bit 数据进行拼接，并产生一个拼接后数据输出有效标识信号。拼接后的数据是前一个数据在高字节位置，后一个数据在低字节位置。这个主要是串口传图上位机软件发送是是先发送 16bit 图像数据的高字节，后发低字节，FPGA 上这样处理为了保证拼接后图像数据的正确性。

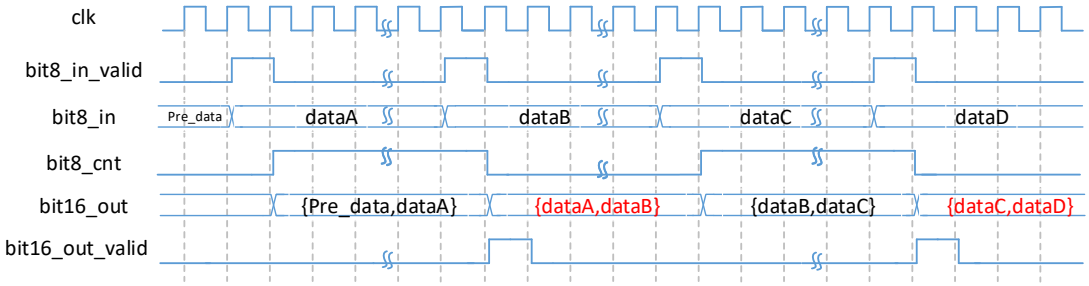


图 2-3 bit8_trans_bit16 数据转换时序图

有了时序图，代码设计就比较简单，具体代码如下。

```
module bit8_trans_bit16(
  input             clk,
  input             reset_p,

  input      [7:0] bit8_in,
  input            bit8_in_valid,

  output reg [15:0]bit16_out,
  output reg       bit16_out_valid
);
```

```
  reg bit8_cnt;

  always@(posedge clk or posedge reset_p)
  begin
    if(reset_p)
      bit8_cnt <= 1'b0;
    else if(bit8_in_valid)
      bit8_cnt <= bit8_cnt + 1'b1;
    else
      bit8_cnt <= bit8_cnt;
  end

  always@(posedge clk or posedge reset_p)
  begin
    if(reset_p)
      bit16_out <= 16'h0000;
    else if(bit8_in_valid)
      bit16_out <= {bit16_out[7:0],bit8_in};
    else
      bit16_out <= bit16_out;
  end

  always@(posedge clk or posedge reset_p)
  begin
    if(reset_p)
      bit16_out_valid <= 1'b0;
    else if(bit8_in_valid && bit8_cnt)
      bit16_out_valid <= 1'b1;
    else
      bit16_out_valid <= 1'b0;
  end
endmodule
```

通过以上设计，位宽转换模块就设计完成。

2.4 系统仿真验证与板级测试

各个子模块设计完成后，顶层的设计就相对容易些，根据整体设计框图对子模块端口信号进行连接即可，具体代码参见本章工程源码。顶层的仿真与 fifo_axi4_adapter 模块仿真类似，这里就不做详细讲解，读者可以参考本章工程源码完成顶层的仿真。注意本次设计的顶层仿真耗时可能较长，实测在 1ms 之后才逐渐开始出现 TFT_rgb 数据。

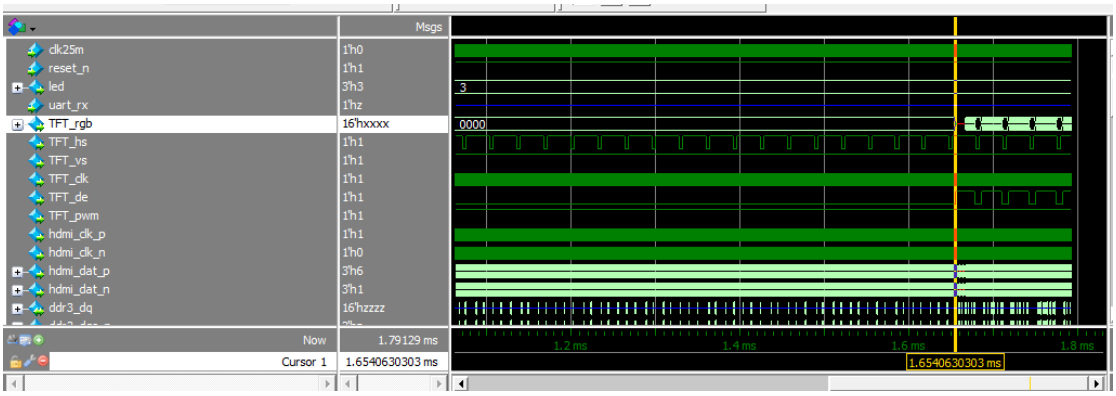


图 2-4 顶层仿真（局部）

2.4.1 管脚绑定

根据 AC201-SA5Z-50D0 开发板提供的管脚绑定表，我们对 TFT 显示屏和串口的管脚，直接进行绑定即可。

由于涉及到 DDR3 的工程，管脚数量都不少，受制于篇幅，读者可以自己参考教程配套的工程代码完成管脚绑定。本工程涉及到绑定的部分，主要为时钟、复位基础管脚，有 DDR 相关管脚，串口相关管脚，TFT 相关管脚等几个部分。

2.4.2 串口传图工程的 TFT 显示

在顶层设计分析综合没有错误并且顶层仿真确认设计功能没有问题后，准备进行上板验证。由于本工程涉及管脚数量较多，我们也就不以列表的形式展示本章例程的管脚绑定了。如需参考，读者可以直接从例程中赋值本工程的管脚绑定 upc 文件到自己设计的工程中对自己的设计进行验证。

对工程的管脚和时钟进行约束后，生成 Bit 文件。上板调试硬件平台基于 AC201-SA5Z-50D0 开发板，对应硬件连接如下图所示:

图 2-5 串口传图开发板连接图

连接好开发板后，下载生成的 Bit 文件。

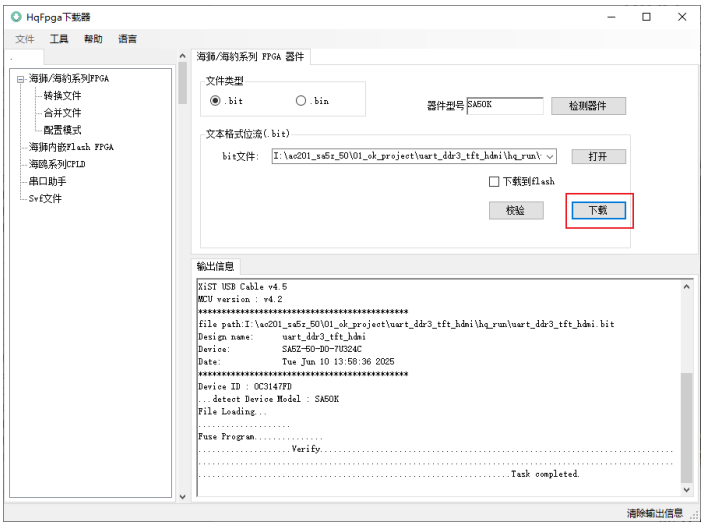


图 2-6 下载串口传图 bit 文件

下载完成后，开发板上 LED0~LED1 会亮，TFT5.0 寸屏上显示花屏状态。如下图所示。

图 2-7 下载完程序后产生的碎花屏效果

出现这种花屏是因为这个时候，DDR3 中并未写入数据，显示的数据是不可知的一些数据。LED0 和 LED1 分别表示的是 DDR 控制器内时钟锁相环的 locked 信号和 DDR 初始化校准完成信号的状态，亮表示这个两个信号均变为高电平，说明 DDR 已经正常完成初始化和校准操作。接下来通过小梅哥串口传图工具向 FPGA 传输图片数据。小梅哥串口传图工具可在论坛下载。

图 2-8 启动串口传图工具

双击打开串口传图工具，通过点击“打开图片”按钮设置图片存放路径；图片宽度和高度设置成与显示屏分辨率一致（TFT5.0 寸屏是 800*480）。下图是待传的图片。

图 2-9 需上传的图片

关于图片的信息可通过右键图片查看其属性，在属性的详细信息窗口可以看到图片大小，位深度等信息。注：这里提供的上位机要求图片为位深度为 24

或 16 的 bmp 格式图片，图片的宽度和高度需要为 800*480（插 5 寸屏情况下）或 480*272（插 4.3 寸屏情况下）。

图 2-10 需传图片的属性信息

串口波特率设置与 FPGA 串口接收波特率一致（FPGA 串口接收波特率是 1562500bps），串口端口号根据实际连接电脑的串口号进行设置（这里使用的是 COM14），设置好传图工具后，点击“连接设备”。

图 2-11 软件连接图片

连接成功后，“连接设备”会变成“断开设备”，通过点击“发送图片”按钮开始发送图片数据。

图 2-12 设置完图片信息后向对应的 com 端口号发送图片

传图过程中，可以看到 TFT 屏上开始显示发送的图片。图片传送完成后，TFT 屏显示效果如下。

图 2-13 串口传图完成后的图片效果

以上就完成了串口传图 TFT 显示的工程设计。

2.4.3 串口传图工程添加 HDMI 显示

该显示内容还可以通过 HDMI 接口输出到 HDMI 显示器上。当然，添加 HDMI 显示后，TFT 显示也可以保留。

为了实现以上目标，我们可以对工程作如下改进：

1. 在锁相环中，添加 165M 时钟信号输出。

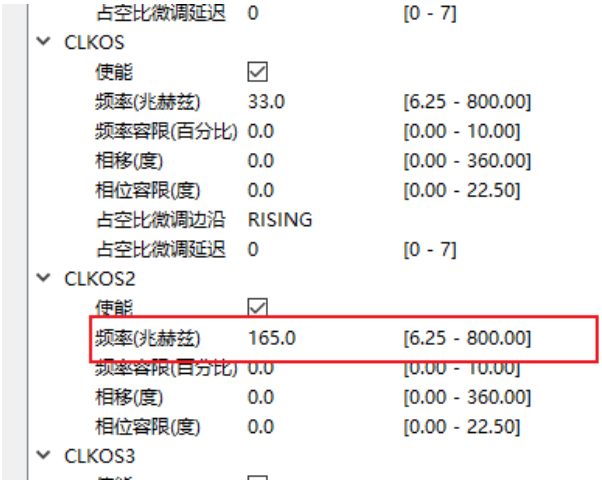


图 2-14 锁相环添加 165MHz 输出频率

2. 在端口处添加 HDMI 输出管脚，并在 XDC 文件或管脚配置界面进行

HDMI 管脚绑定。

```
//hdmi1 interface
output          hdmi1_clk_p   ,
output          hdmi1_clk_n   ,
output [2:0]    hdmi1_dat_p   ,
output [2:0]    hdmi1_dat_n   ,
output          hdmi1_oe
```

3. 添加 HDMI 接口例化：

```
//HDMI
dvi_encoder dvi_encoder(
    .pixelclk    (pixelclk    ),
    .pixelclk5x  (pixelclk5x  ),
    .rst_p       (g_rst_p     ),
    .blue_din    (disp_blue   ),
    .green_din   (disp_green  ),
    .red_din     (disp_red    ),
    .hsync       (disp_hs     ),
    .vsync       (disp_vs     ),
    .de          (disp_de     ),
    .tmds_clk_p  (hdmi_clk_p ),
    .tmds_clk_n  (hdmi_clk_n ),
    .tmds_data_p (hdmi_dat_p ),
    .tmds_data_n (hdmi_dat_n )
);
```

4. RGB565 和 RGB888 的转换

```
assign hdmi1_oe = 1'b1;
wire [7:0] disp_red;
wire [7:0] disp_green;
wire [7:0] disp_blue;


assign disp_red = {TFT_rgb[15:11],3'b0};
assign disp_green = {TFT_rgb[10:5],2'b0};
assign disp_blue = {TFT_rgb[4:0],3'b0};

wire[15:0] image_data;
assign wrfifo_din  = image_data;
assign wrfifo_wren = image_data_valid;

wire disp_hs = TFT_hs;
wire disp_vs = TFT_vs;
wire disp_de = TFT_de;
```

5. 添加 HDMI 管脚绑定

Enabled	TFT_pwm	← OUTPUT	P4 bank12	...	LVCMOS33	NONE
Enabled	hdmi_clk_p	← OUTPUT	T3 bank12	...	LVCMOS33	NONE
Enabled	hdmi_clk_n	← OUTPUT	R3 bank12	...	LVCMOS33	NONE
Enabled	hdmi_dat_p[2]	← OUTPUT	V4 bank12	...	LVCMOS33	NONE
Enabled	hdmi_dat_p[1]	← OUTPUT	T6 bank12	...	LVCMOS33	NONE
Enabled	hdmi_dat_p[0]	← OUTPUT	U3 bank12	...	LVCMOS33	NONE
Enabled	hdmi_dat_n[2]	← OUTPUT	R7 bank12	...	LVCMOS33	NONE
Enabled	hdmi_dat_n[1]	← OUTPUT	U4 bank12	...	LVCMOS33	NONE
Enabled	hdmi_dat_n[0]	← OUTPUT	N6 bank12	...	LVCMOS33	NONE

图 2-15 HDMI 相关管脚绑定

6. 完成以上配置后，在前面给出的线缆连接图中，添加开发板 HDMI 接口和液晶显示器的连接。

7. 将改造好的工程下载到 FPGA 开发板，HDMI 显示如下：

图 2-16 串口传图 TFT 和 HDMI 联合显示实验效果

至此，串口传图帧缓存系统在 ACX750 开发板上的设计及验证就成功完成了。

2.5 总结

本章讲解了基于 DDR3 的串口传图帧缓存系统的设计与实现方法。通过本章的学习，各位读者可以更深入的对输入控制器，缓存控制器，输出控制器的设计有一个更为全面宏观的认识，在这些控制器协调工作时，既需要关注数据在流动中形态的不同，又需要关注数据收发控制信号和不同时钟频率信号的协调。