

3 基于 OV5640 的 FPGA-DDR HDMI 显示

这部分主要是在上一节的基础上将 BRAM 换成更“廉价”的 DDR 内存，这样方便后续算法的开发和数据处理。

3.1 Xilinx 平台 DDR3 控制器使用

这部分详细的内容可以参考《\\...\Image\002_OV5640_DDR3_DEMO\DOC\DDR》目录下内容，本文档只简单介绍下使用方法。

本节介绍 7 系列 FPGA 存储器接口解决方案核心架构，概述了核心模块和接口。7 系列 FPGA 存储器接口解决方案核心如下图所示。

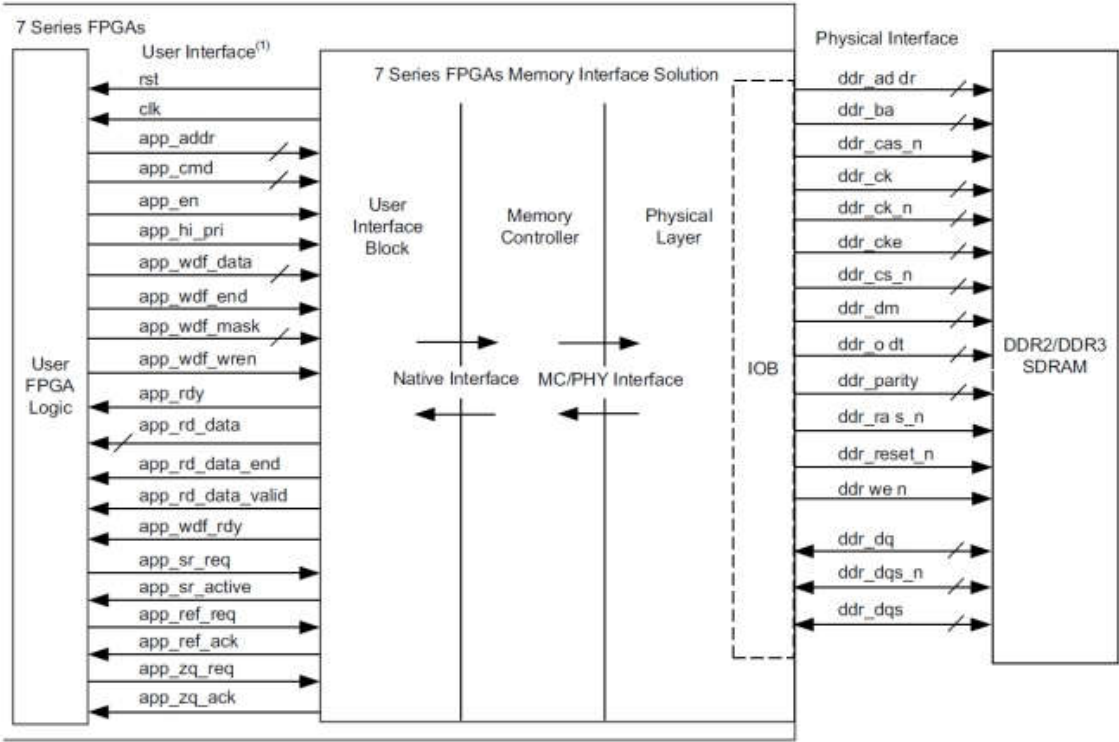
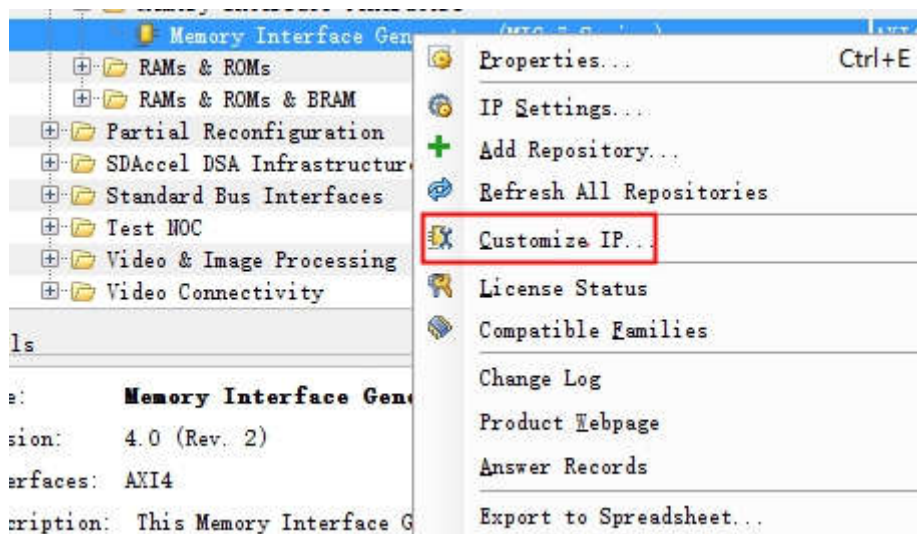


图 3-1 Series FPGAs Memory Interface Solution

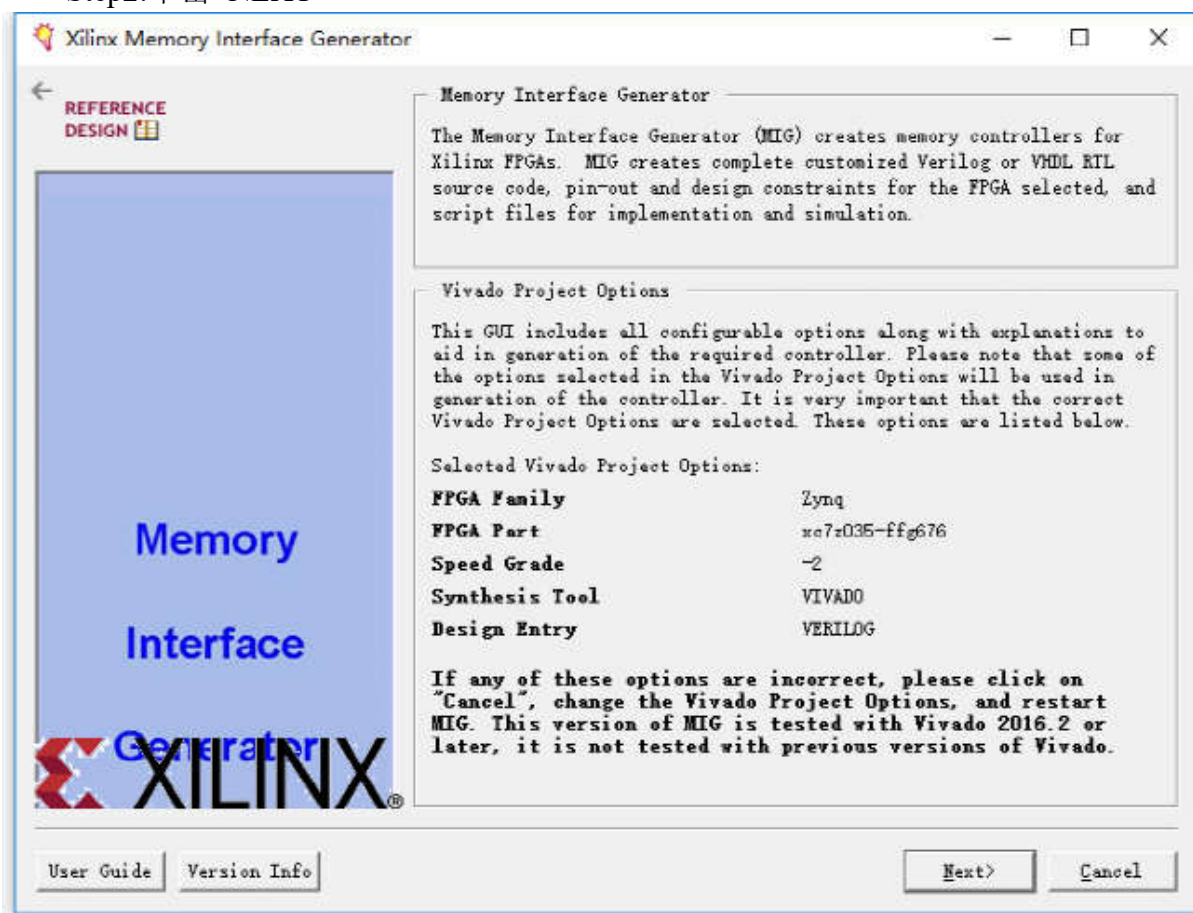
3.1.1 Setp By Step 搭建 FPGA 工程

Step1:任意创建一个新的空的工程（创建工程的具体工程如果还不清楚的看我们教程第一季部分），并且进入 IP CORE 列表 右击 Customize ip。

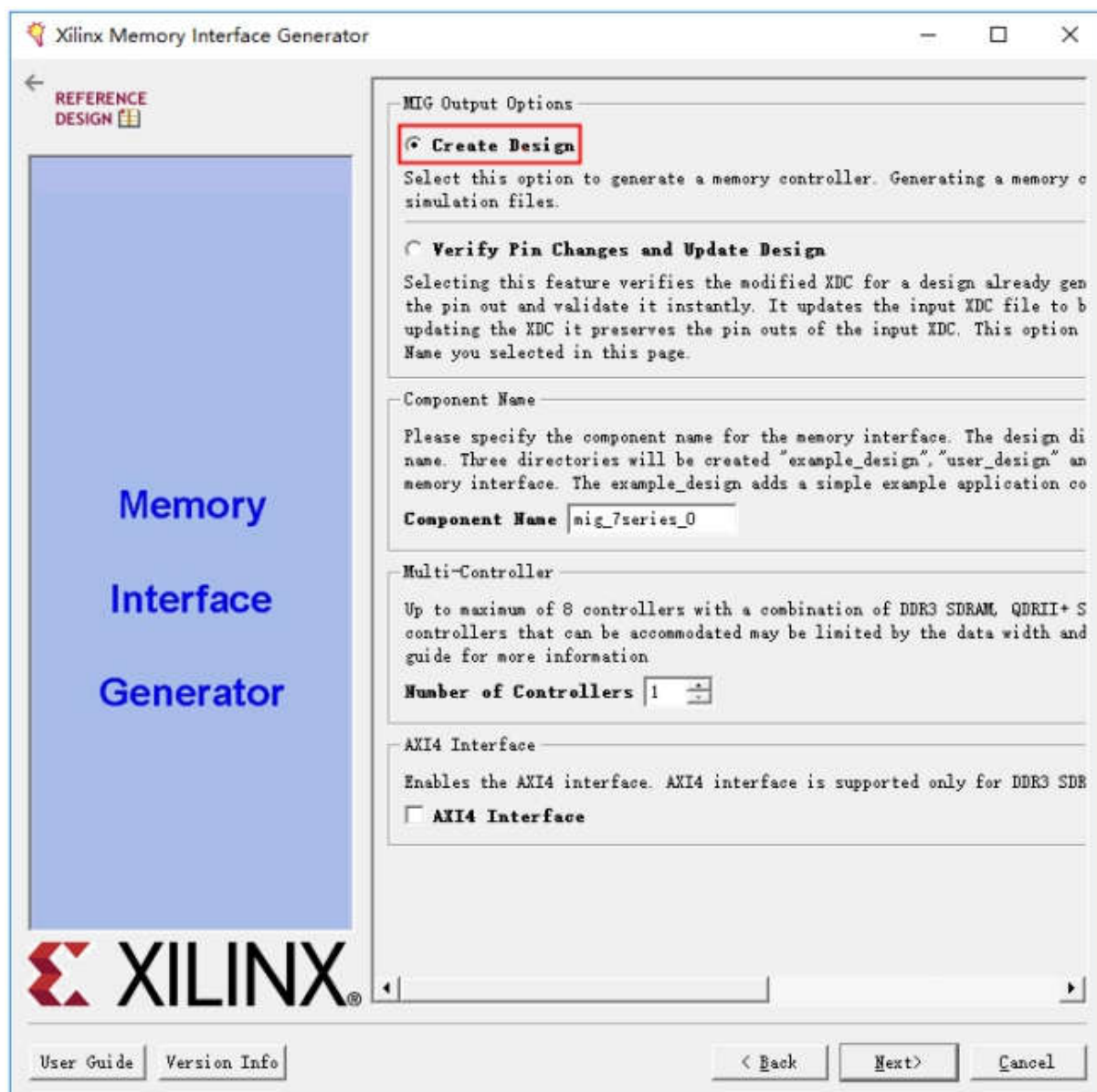




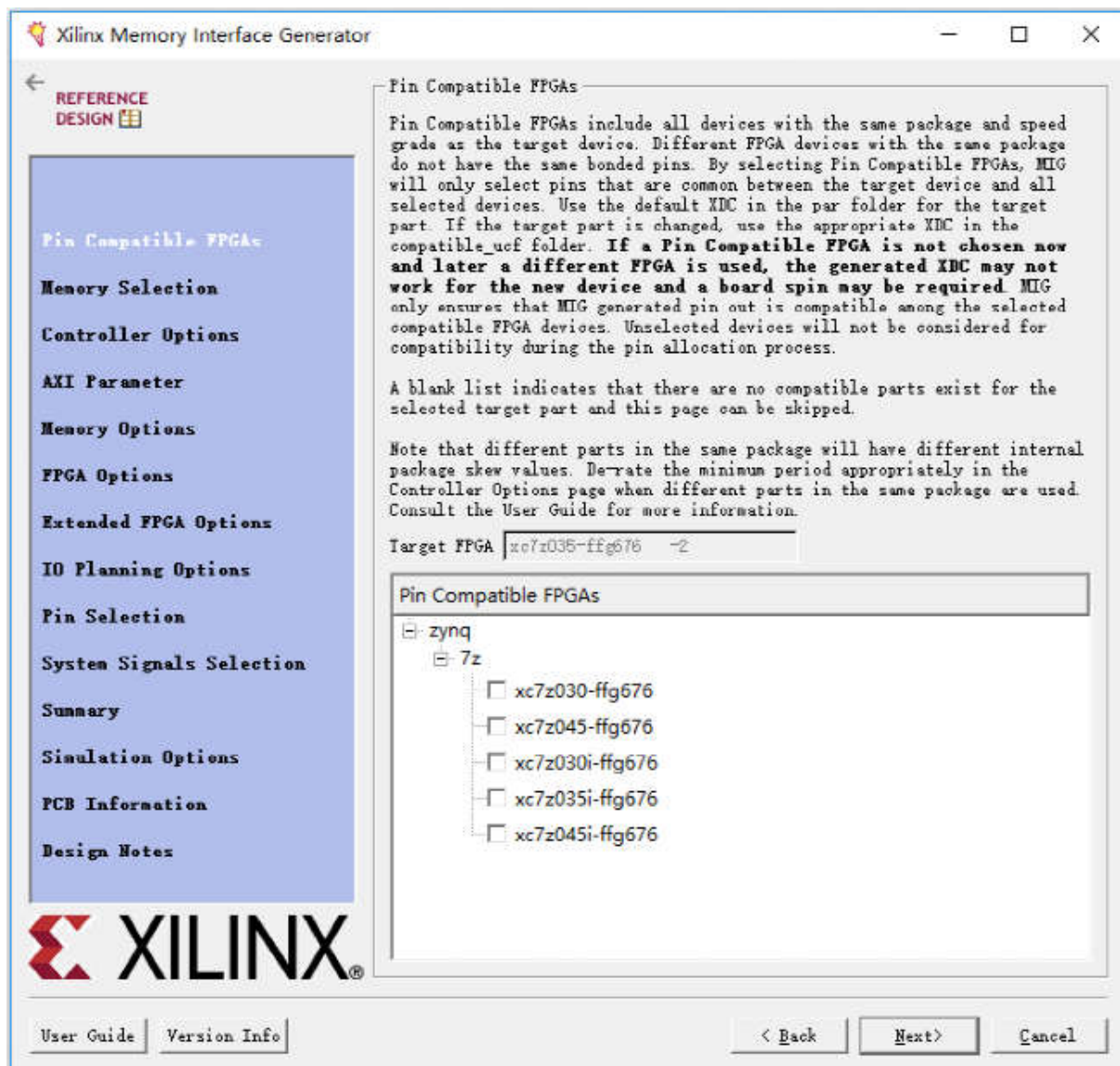
Step2:单击 NEXT



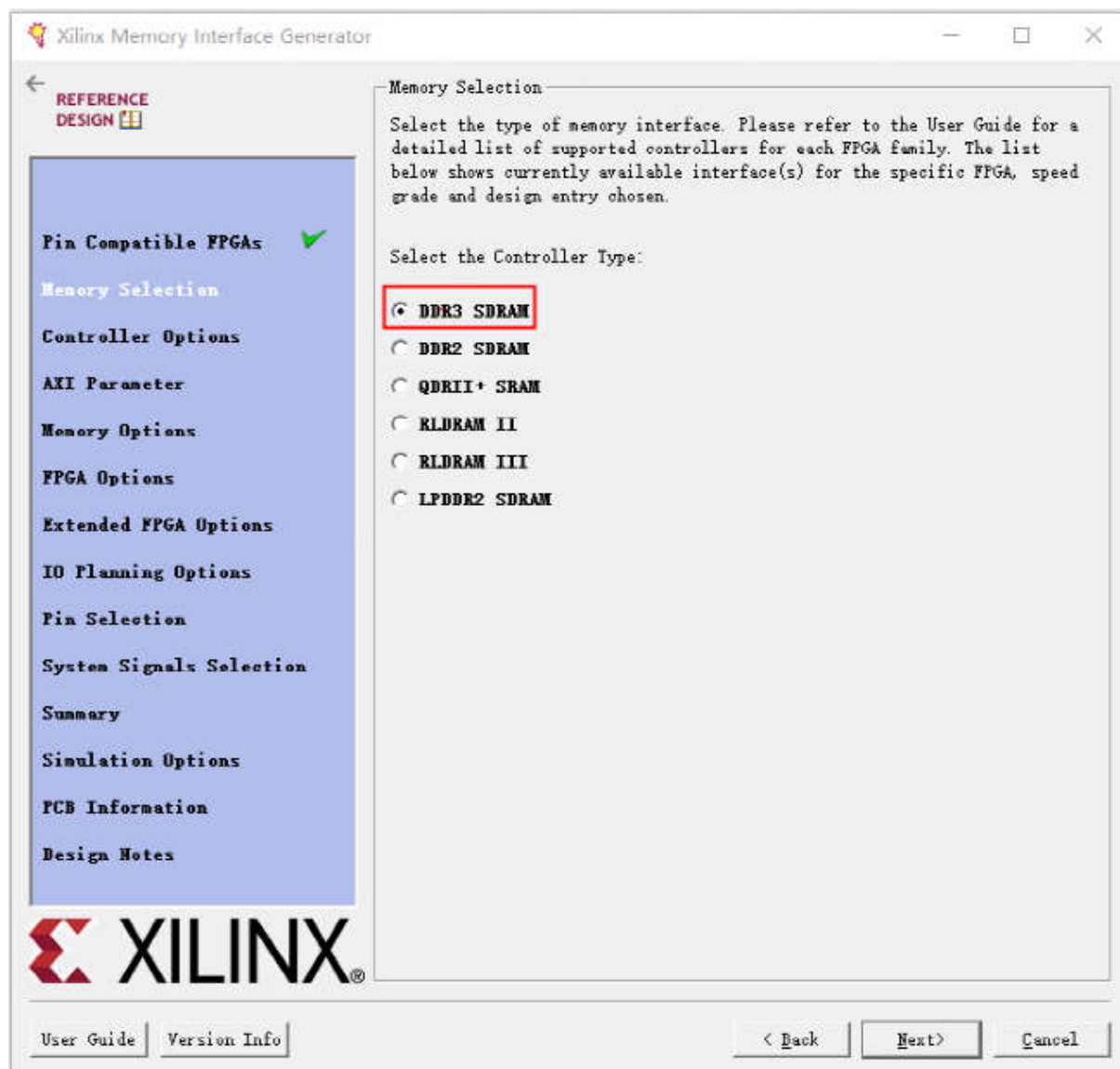
Step3:选择 Create Design 单击 NEXT



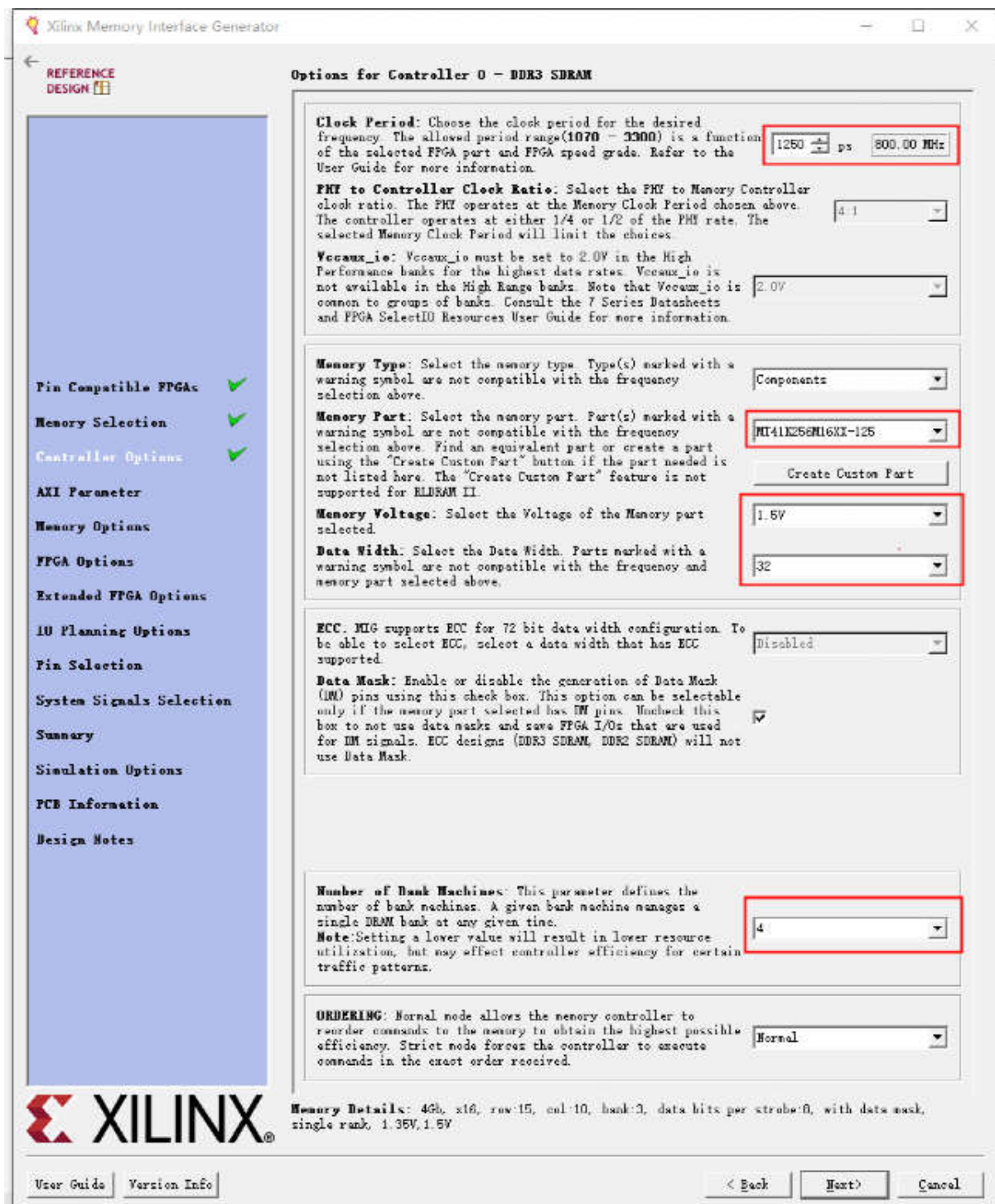
Step4:单击 NEXT



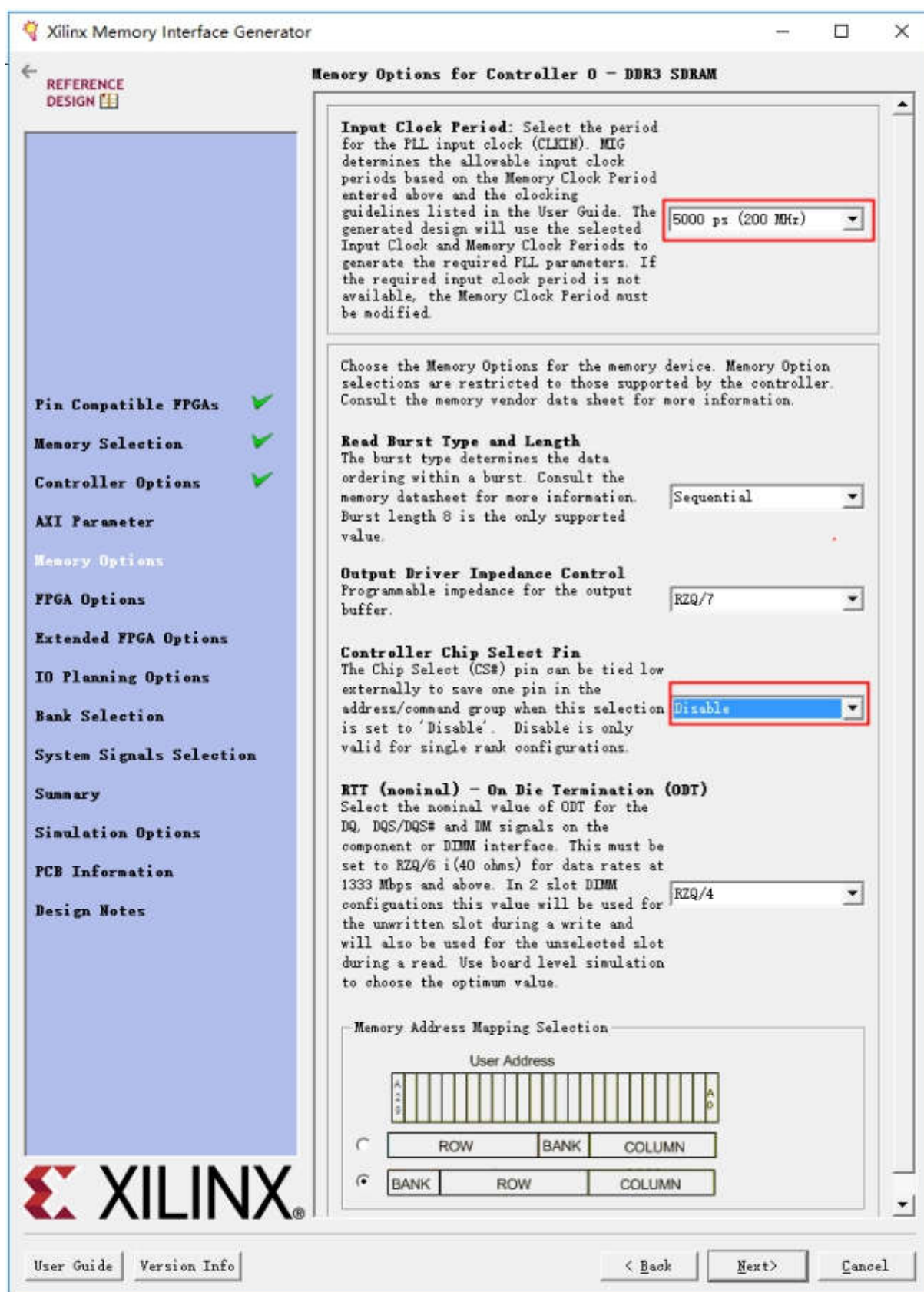
Step5:选择 DDR3 单击 NEXT



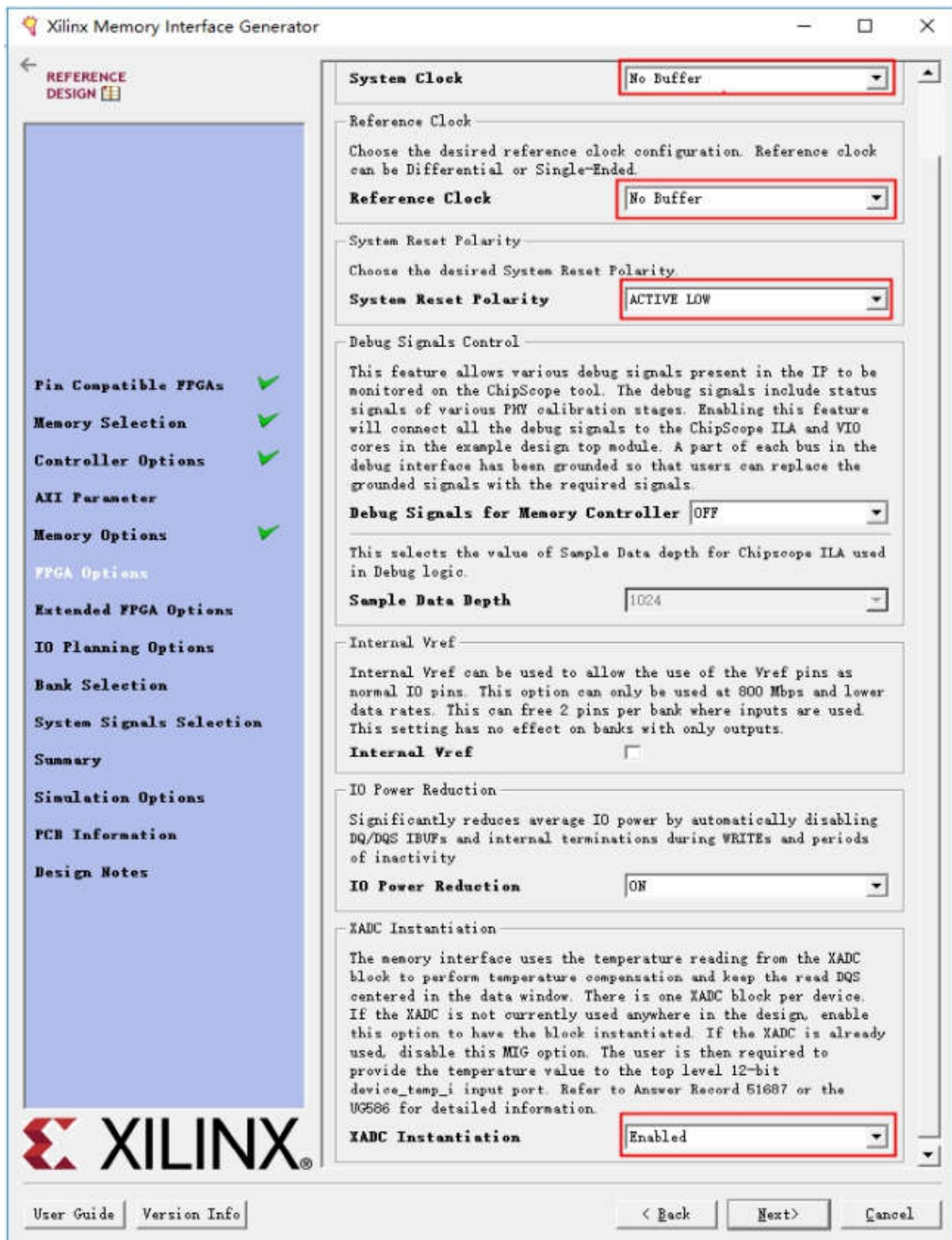
Step6:设置 MIG 内核时钟频率为 800M(数据: 1600MX32bit)、内存型号、内存的数据位宽



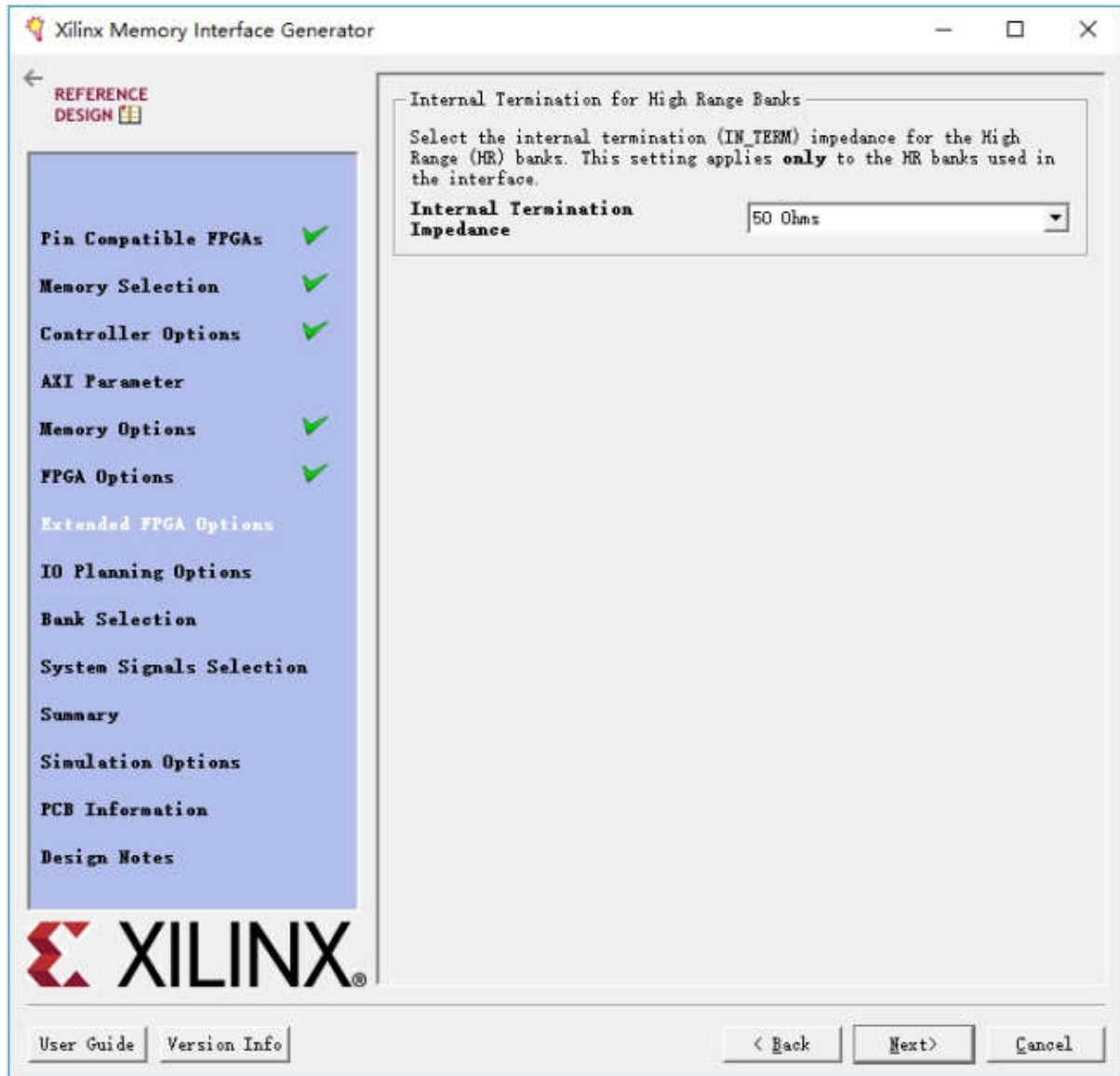
Step7:设置输入频率为 200M,不使用调试信号



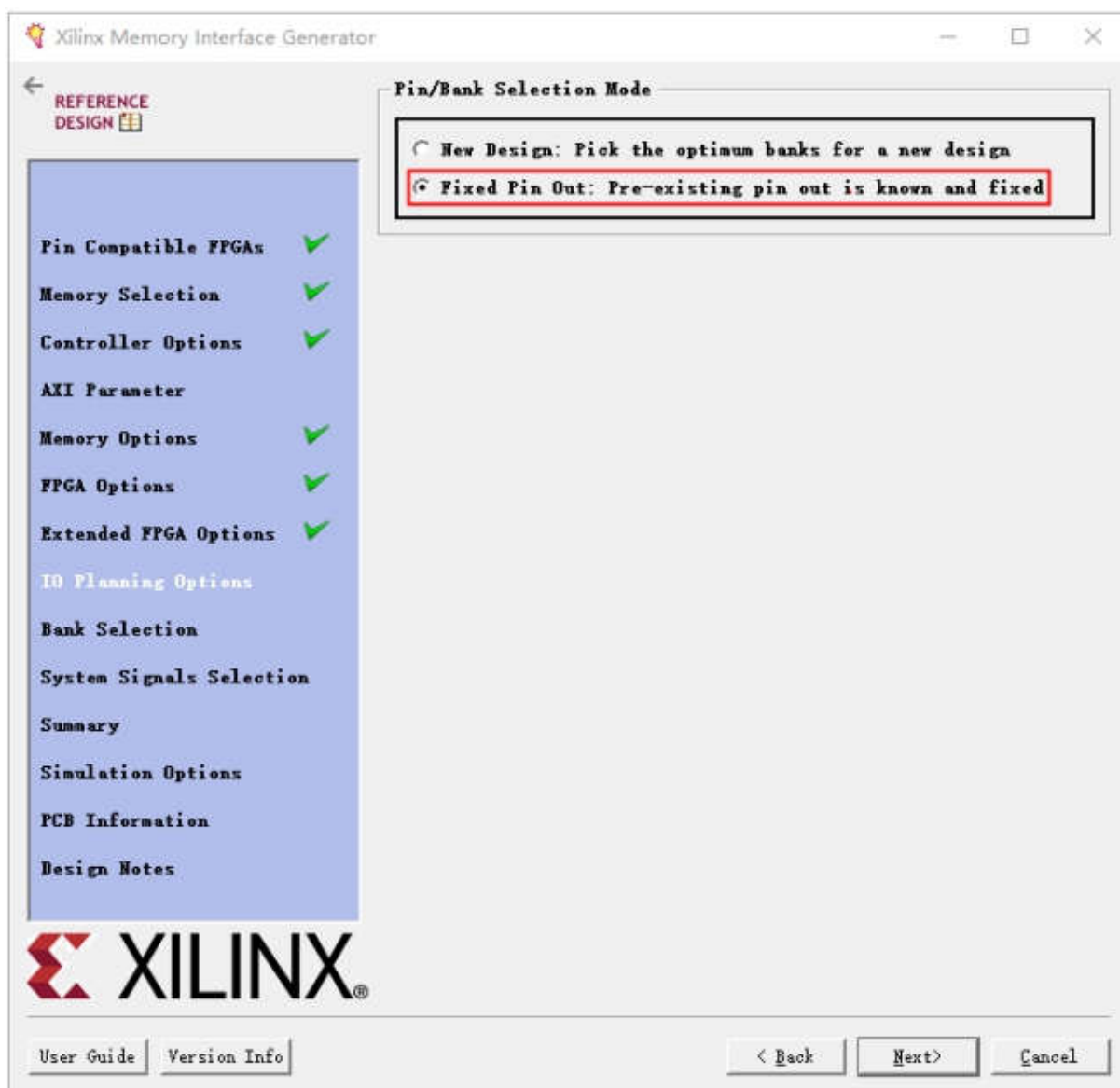
Step8:系统和参考时钟选择 no buffer, MIG 低电平复位, XADC 补偿使能(本教程没有使用到, 但是仍然势能)



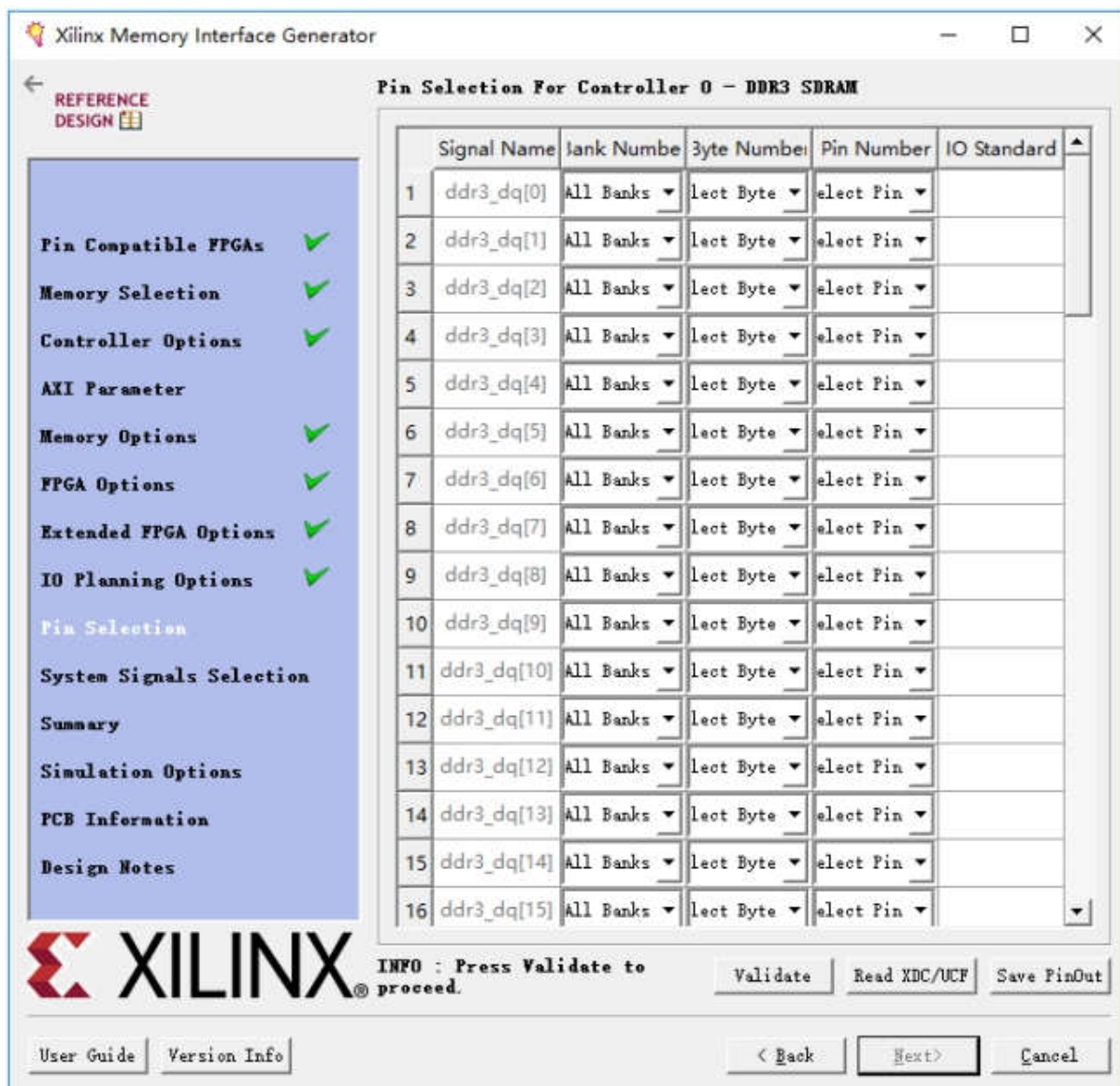
Step9:终端阻抗选择 50hms



Step10:选择 Fixed Pin Out



Step11:根据原理图手动填写 PIN 脚定义



The screenshot shows the 'Pin Selection For Controller 0 - DDR3 SDRAM' window in the Xilinx Memory Interface Generator. The left sidebar contains a list of configuration steps, with 'Pin Selection' highlighted. The main area displays a table for pin selection for 16 signals (ddr3_dq[0] to ddr3_dq[15]). Each row has dropdown menus for 'Bank Number' (set to 'All Banks'), 'Byte Number' (set to '1st Byte'), and 'Pin Number' (set to 'select Pin'). The 'IO Standard' column is empty. At the bottom, there are buttons for 'Validate', 'Read XDC/UCF', 'Save PinOut', '< Back', 'Next >', and 'Cancel'. A status bar at the bottom left shows 'User Guide' and 'Version Info'.

	Signal Name	Bank Number	Byte Number	Pin Number	IO Standard
1	ddr3_dq[0]	All Banks	1st Byte	select Pin	
2	ddr3_dq[1]	All Banks	1st Byte	select Pin	
3	ddr3_dq[2]	All Banks	1st Byte	select Pin	
4	ddr3_dq[3]	All Banks	1st Byte	select Pin	
5	ddr3_dq[4]	All Banks	1st Byte	select Pin	
6	ddr3_dq[5]	All Banks	1st Byte	select Pin	
7	ddr3_dq[6]	All Banks	1st Byte	select Pin	
8	ddr3_dq[7]	All Banks	1st Byte	select Pin	
9	ddr3_dq[8]	All Banks	1st Byte	select Pin	
10	ddr3_dq[9]	All Banks	1st Byte	select Pin	
11	ddr3_dq[10]	All Banks	1st Byte	select Pin	
12	ddr3_dq[11]	All Banks	1st Byte	select Pin	
13	ddr3_dq[12]	All Banks	1st Byte	select Pin	
14	ddr3_dq[13]	All Banks	1st Byte	select Pin	
15	ddr3_dq[14]	All Banks	1st Byte	select Pin	
16	ddr3_dq[15]	All Banks	1st Byte	select Pin	

Step12:填写完成后，点击 Vivald 验证一下合理性，然后最后保存一版 PinOut，方便后续这部分开发。

Xilinx Memory Interface Generator

REFERENCE DESIGN

Pin Selection For Controller 0 - DDR3 SDRAM

	Signal Name	Bank Number	Byte Number	Pin Number	IO Standard
1	ddr3_dq[0]	33	T1	G1	SSTL15_T_DCI
2	ddr3_dq[1]	33	T1	J4	SSTL15_T_DCI
3	ddr3_dq[2]	33	T1	H1	SSTL15_T_DCI
4	ddr3_dq[3]	33	T1	H4	SSTL15_T_DCI
5	ddr3_dq[4]	33	T1	H2	SSTL15_T_DCI
6	ddr3_dq[5]	33	T1	L3	SSTL15_T_DCI
7	ddr3_dq[6]	33	T1	J1	SSTL15_T_DCI
8	ddr3_dq[7]	33	T1	K1	SSTL15_T_DCI
9	ddr3_dq[8]	33	T0	F3	SSTL15_T_DCI
10	ddr3_dq[9]	33	T0	C1	SSTL15_T_DCI
11	ddr3_dq[10]	33	T0	B2	SSTL15_T_DCI
12	ddr3_dq[11]	33	T0	D3	SSTL15_T_DCI
13	ddr3_dq[12]	33	T0	G4	SSTL15_T_DCI
14	ddr3_dq[13]	33	T0	D1	SSTL15_T_DCI
15	ddr3_dq[14]	33	T0	B1	SSTL15_T_DCI
16	ddr3_dq[15]	33	T0	F4	SSTL15_T_DCI
17	ddr3_dq[16]	33	T2	M1	SSTL15_T_DCI
18	ddr3_dq[17]	33	T2	L5	SSTL15_T_DCI
19	ddr3_dq[18]	33	T2	M4	SSTL15_T_DCI
20	ddr3_dq[19]	33	T2	M5	SSTL15_T_DCI
21	ddr3_dq[20]	33	T2	M2	SSTL15_T_DCI
22	ddr3_dq[21]	33	T2	M4	SSTL15_T_DCI
23	ddr3_dq[22]	33	T2	L2	SSTL15_T_DCI
24	ddr3_dq[23]	33	T2	M1	SSTL15_T_DCI
25	ddr3_dq[24]	33	T3	N8	SSTL15_T_DCI
26	ddr3_dq[25]	33	T3	K7	SSTL15_T_DCI
27	ddr3_dq[26]	33	T3	N7	SSTL15_T_DCI
28	ddr3_dq[27]	33	T3	J5	SSTL15_T_DCI

Pin Compatible FPGAs ✓
Memory Selection ✓
Controller Options ✓
AXI Parameter
Memory Options ✓
FPGA Options ✓
Extended FPGA Options ✓
IO Planning Options ✓
Pin Selection
System Signals Selection
Summary
Simulation Options
PCB Information
Design Notes

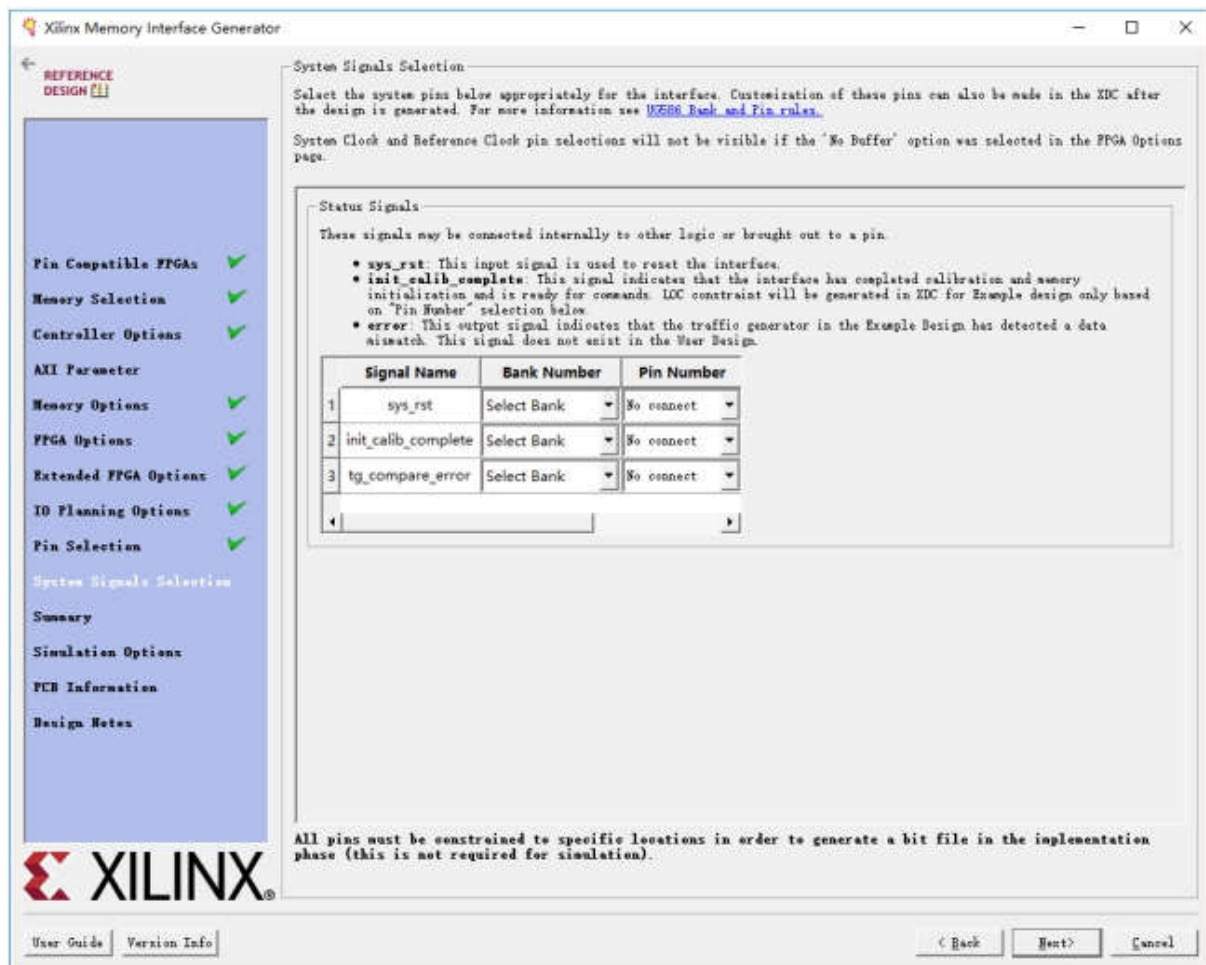
XILINX INFO : Press Validate to proceed.

Validate Read KDC/UCF Save PinOut

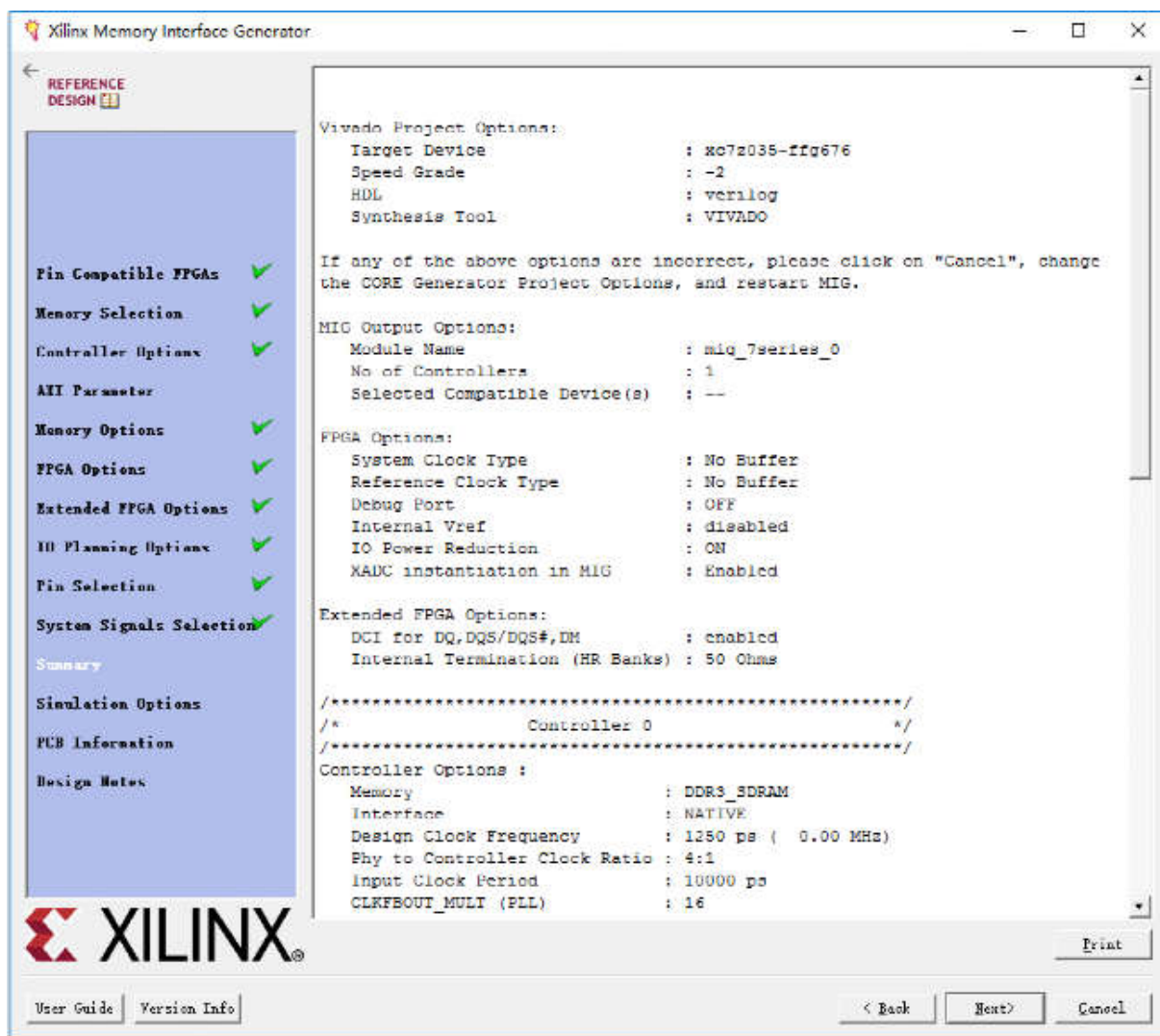
User Guide Version Info

< Back Next? Cancel

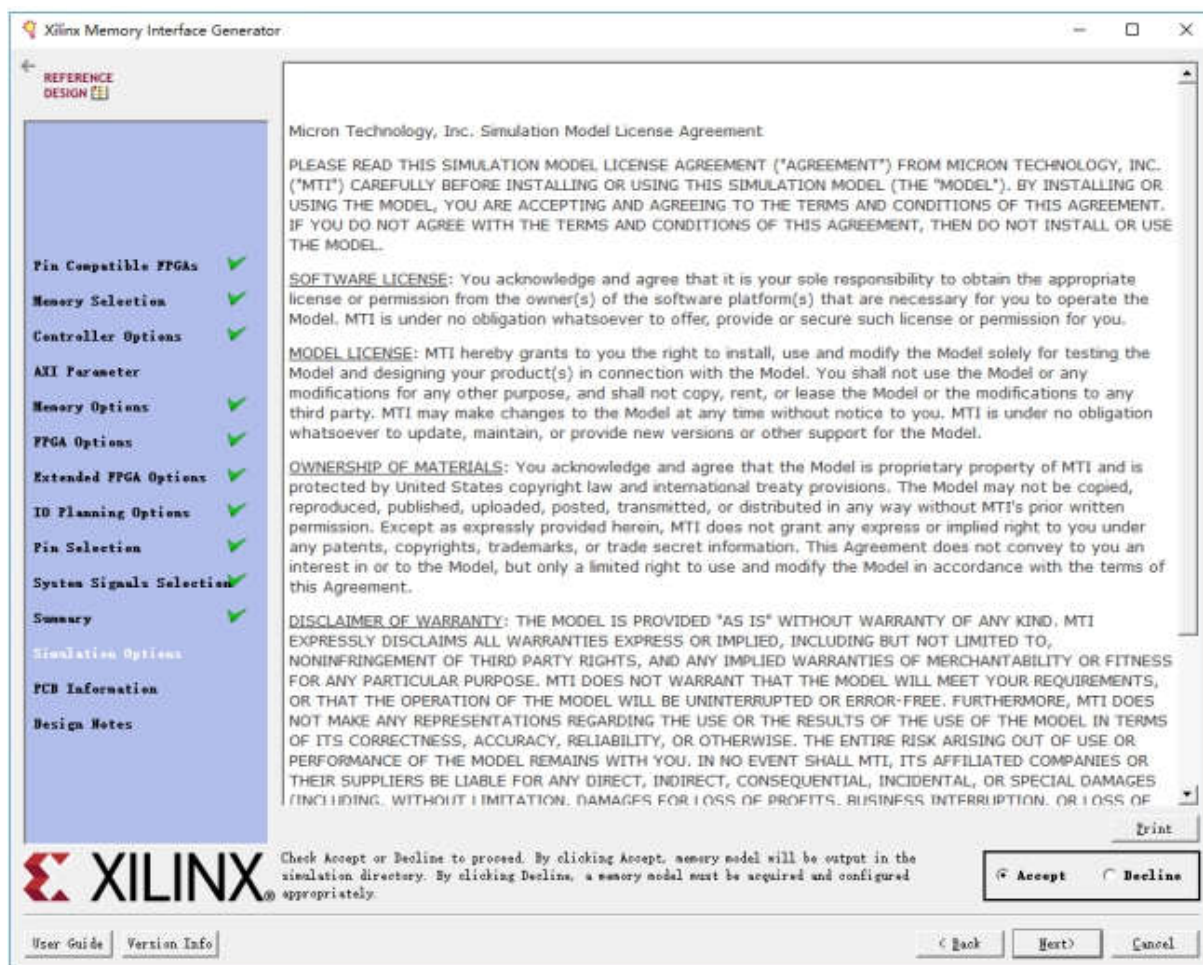
Step13:单击 NEXT



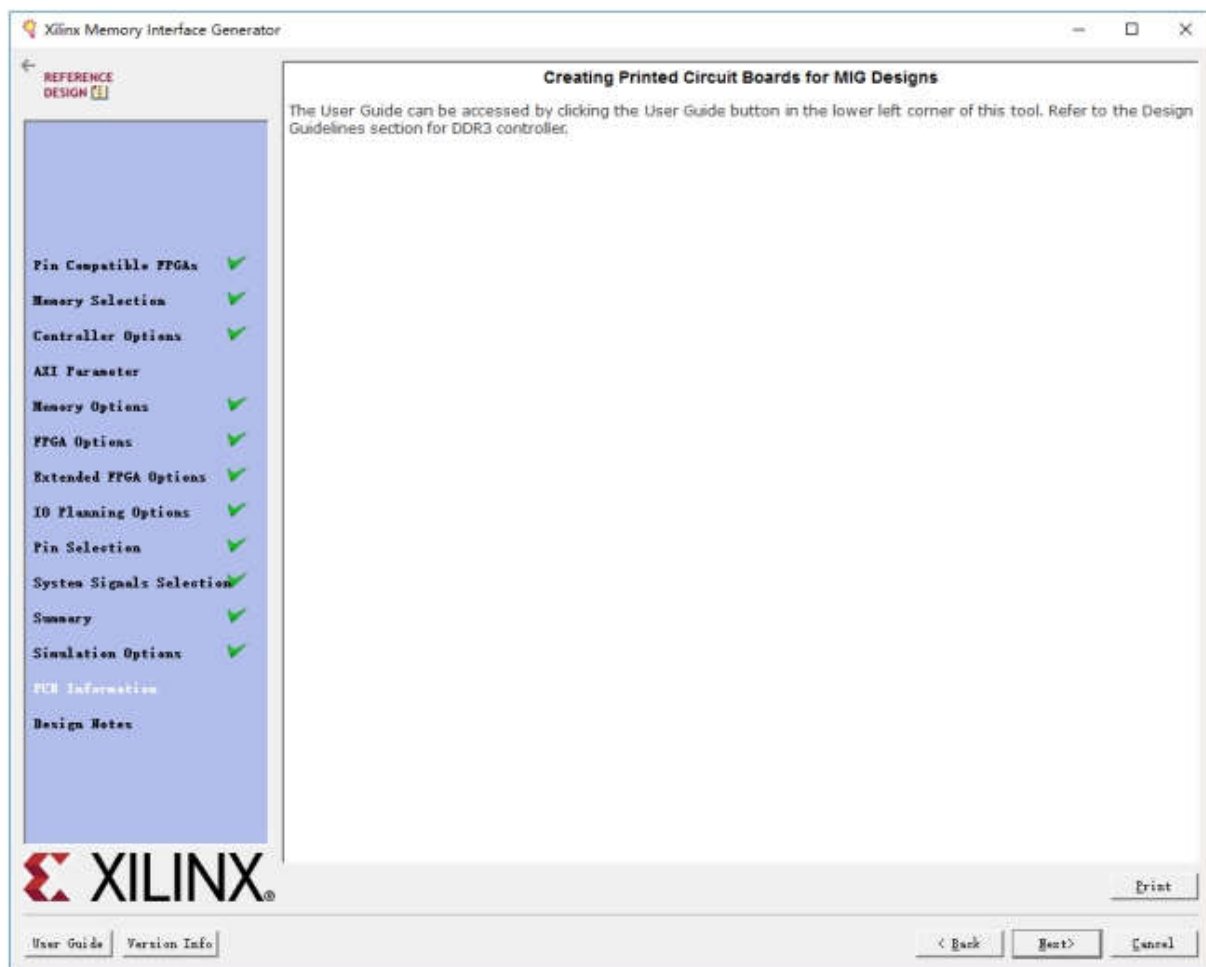
Step14:单击 NEXT



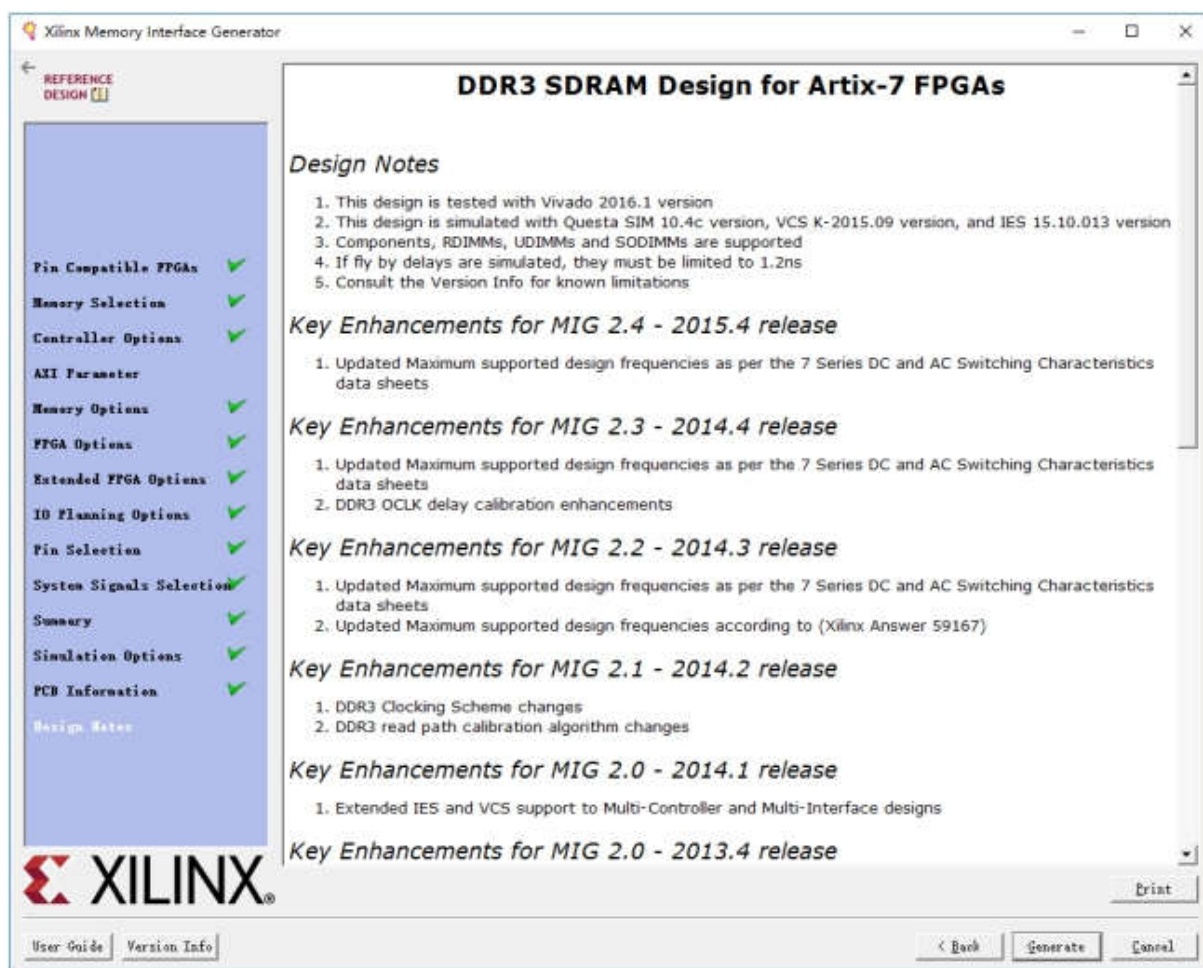
Step15:单击 NEXT



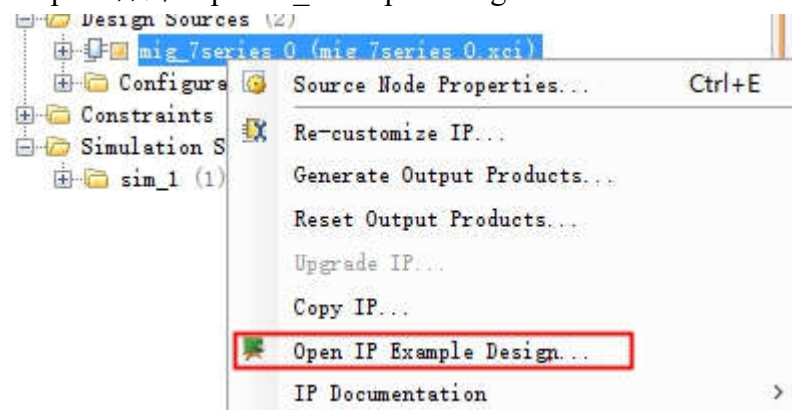
Step16:单击 NEXT



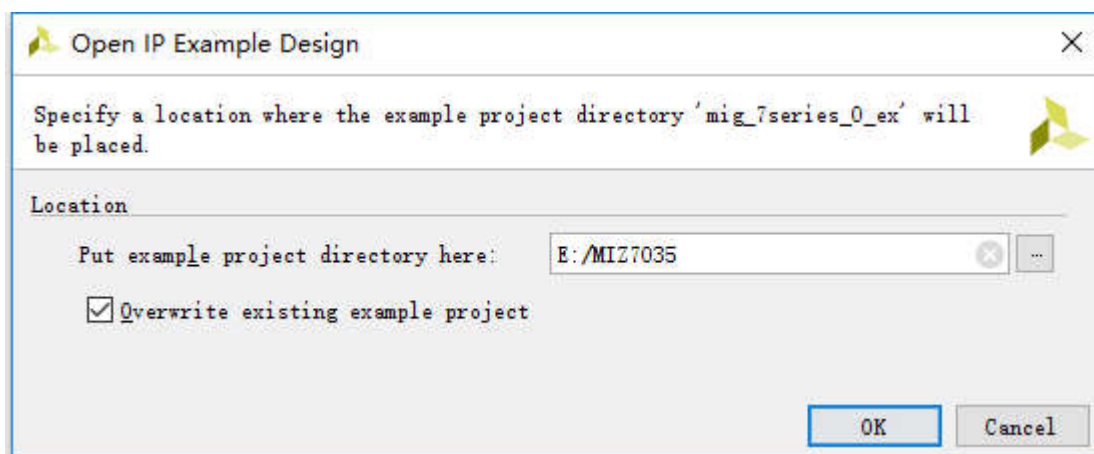
Step17:单击 Generate



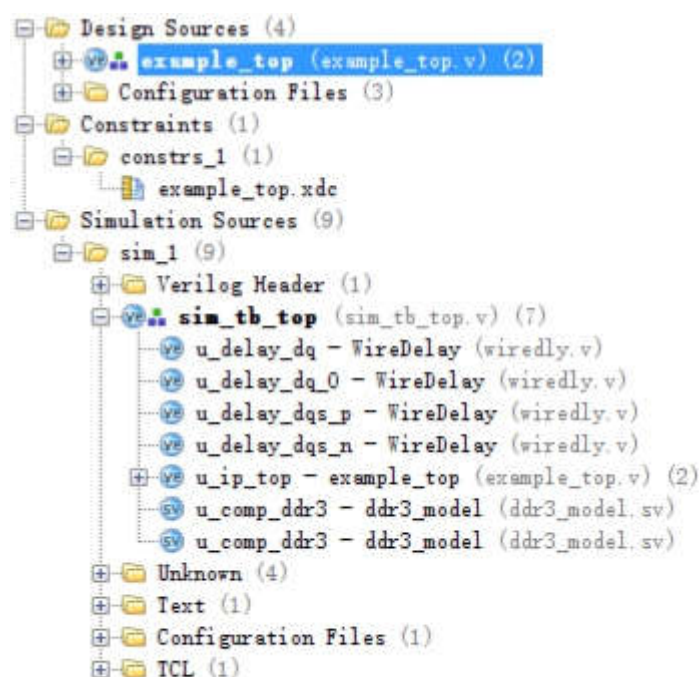
Step17:右击 Open IP Example Design...



Step18:设置好 IP 的路径



Step19:创建完成后的代码



接下来就是怎么使用这个控制器了，下面介绍一下关于这个控制器的接口。

3.1.2.7 系列 FPGA 存储器接口相关参数介绍

3.1.2.1 用户 FPGA 逻辑接口

图 3-1 所示的用户 FPGA 逻辑模块是需要连接到外部 DDR2 或 DDR3 SDRAM 的任何 FPGA 设计。用户 FPGA 逻辑通过用户界面连接到内存控制器。IPCORE 提供了一个用户 FPGA 逻辑示例。

AXI4 从接口块

AXI4 从站接口将 AXI4 事务映射到 UI，以向内存控制器提供行业标准总线协议接口。

用户界面块和用户界面

UI 块向用户提供 FPGA 逻辑块。它通过呈现平面地址空间和缓冲读写数据来提供对本机接口的简单替代。

内存控制器和本机接口

内存控制器（MC）的前端显示 UI 块的本机接口。本地接口允许用户设计提交存储器读写请求，并提供将数据从用户设计移动到外部存储器件的机制，反之亦然。内存控制器的后端连接到物理接口，并处理该模块的所有接口要求。内存控制器还提供了重新排序选项，重新排序接收的请求以优化数据吞吐量和延迟。

PHY 和物理接口 PHY 的前端连接到存储控制器。PHY 的后端连接到外部存储设备。PHY 处理所有存储器件信号顺序和定时。IDELAYCTRL 任何使用 IDELAYs 的银行都需 IDELAYCTRL。IDELAY 与数据组（DQ）相关联。任何使用这些信号的 BANK/时钟区域都需 IDELAYCTRL。MIG 工具实例化一个 IDELAYCTRL，然后使用 IODELAY_GROUP 属性（参见 iodelay_ctrl.v 模块）。基于此属性，Vivado Design Suite 可根据需要在设计中正确复制 IDELAYCTRL。IDELAYCTRL 参考频率由 MIG 工具设置为 200 MHz，300 MHz 或 400 MHz，具体取决于 FPGA 的存储器接口频率和速度等级。根据设置的 IODELAY_GROUP 属性，Vivado Design Suite 会复制 IDELAY 块所在区域的 IDELAYCTRL。

当用户自己创建一个多控制器设计时，每个 MIG 输出都具有用原语实例化的组件。这违反了 IDELAYCTRLs 的规则和 IODELAY_GRP 属性的使用。IDELAYCTRL 需要只有一个组件的实例化才能正确设置属性，并允许工具根据需要进行复制。

用户接口

UI 如表 3-1 所示，并连接到 FPGA 用户设计，以允许访问外部存储设备。

表 3-1 User Interface

Signal	Direction	Description
app_addr[ADDR_WIDTH - 1:0]	Input	This input indicates the address for the current request.
app_cmd[2:0]	Input	This input selects the command for the current request.
app_en	Input	This is the active-High strobe for the app_addr[], app_cmd[2:0], app_sz, and app_hi_pri inputs.
app_rdy	Output	This output indicates that the UI is ready to accept commands. If the signal is deasserted when app_en is enabled, the current app_cmd and app_addr must be retried until app_rdy is asserted.
app_hi_pri	Input	This active-High input elevates the priority of the current request.
app_rd_data [APP_DATA_WIDTH - 1:0]	Output	This provides the output data from read commands.
app_rd_data_end	Output	This active-High output indicates that the current clock cycle is the last cycle of output data on app_rd_data[]. This is valid only when app_rd_data_valid is active-High.
app_rd_data_valid	Output	This active-High output indicates that app_rd_data[] is valid.
app_sz	Input	This input is reserved and should be tied to 0.
app_wdf_data [APP_DATA_WIDTH - 1:0]	Input	This provides the data for write commands.
app_wdf_end	Input	This active-High input indicates that the current clock cycle is the last cycle of input data on app_wdf_data[].
app_wdf_mask [APP_MASK_WIDTH - 1:0]	Input	This provides the mask for app_wdf_data[].
app_wdf_rdy	Output	This output indicates that the write data FIFO is ready to receive data. Write data is accepted when app_wdf_rdy = 1'b1 and app_wdf_wren = 1'b1.
app_wdf_wren	Input	This is the active-High strobe for app_wdf_data[].
app_correct_en_i	Input	When asserted, this active-High signal corrects single bit data errors. This input is valid only when ECC is enabled in the GUI. In the example design, this signal is always tied to 1.
app_sr_req	Input	This input is reserved and should be tied to 0.
app_sr_active	Output	This output is reserved.

Signal	Direction	Description
app_ref_req	Input	This active-High input requests that a refresh command be issued to the DRAM.
app_ref_ack	Output	This active-High output indicates that the Memory Controller has sent the requested refresh command to the PHY interface.
app_zq_req	Input	This active-High input requests that a ZQ calibration command be issued to the DRAM.
app_zq_ack	Output	This active-High output indicates that the Memory Controller has sent the requested ZQ calibration command to the PHY interface.
ui_clk	Output	This UI clock must be a half or quarter of the DRAM clock.
init_calib_complete	Output	PHY asserts init_calib_complete when calibration is finished.
app_ecc_multiple_err[7:0](1)	Output	This signal is applicable when ECC is enabled and is valid along with app_rd_data_valid. The app_ecc_multiple_err[3:0] signal is non-zero if the read data from the external memory has two bit errors per beat of the read burst. The SECDED algorithm does not correct the corresponding read data and puts a non-zero value on this signal to notify the corrupted read data at the UI.
ui_clk_sync_rst	Output	This is the active-High UI reset.
app_ecc_single_err[7:0]	Output	This signal is applicable when ECC is enabled and is valid along with app_rd_data_vali. The app_ecc_single_err signal is non-zero if the read data from the external memory has a single bit error per beat of the read burst.

注意：

1.该信号仅提供到 memc_ui_top 模块级别。 只有当 ECC 被使能时才应使用该信号。

app_addr [ADDR_WIDTH - 1: 0]

此输入表示当前正在提交给用户界面的请求的地址。UI 聚合外部 SDRAM 的所有地址字段，并向您显示一个平面地址空间。

app_cmd[2: 0]

此输入指定请求的命令如下表所示

表 3-2 Commands for app_cmd[2:0]

Operation	app_cmd[2:0] Code
Read	001
Write	000

app_en

此信号在输入请求中使用， 用户必须赋值于 app_addr [], app_cmd [2: 0]和 app_hi_pri， 然后断言 app_en 将该请求提交给 UI。 这将通过断言 app_rdy 启动 UI 确认的握手。

app_hi_pri

该输入表示当前请求是高优先级。

app_wdf_data [APP_DATA_WIDTH - 1: 0]

该总线提供当前正在写入外部存储器的数据。

app_wdf_end

该输入表示当前周期中 app_wdf_data [] 总线上的数据是当前请求的数据。

app_wdf_mask [APP_MASK_WIDTH - 1: 0]

该总线指示 app_wdf_data [] 的哪些位被写入外部存储器，哪些位保持在当前状态。

app_wdf_wren

该输入表示 app_wdf_data [] 总线上的数据有效。

app_rdy

此输出向用户显示当前正在提交给 UI 的请求是否被接受。如果在 app_en 被断言之后，UI 不会声明此信号，则必须重试当前的请求。如果以下情况，则不会声明 app_rdy 输出：

- 1) PHY / 内存初始化尚未完成
- 2) 所有的银行机器都被占用（ 可以看作命令缓冲区已满）
 - 请求读取， 读取缓冲区已满
 - 请求写入， 并且没有写缓冲区指针可用
- 3) 正在插入定期读取

app_rd_data [APP_DATA_WIDTH - 1: 0]

该输出包含从外部存储器读取的数据。

app_rd_data_end

此输出表示当前周期中 app_rd_data [] 总线上的数据为当前请求的最后数据。

app_rd_data_valid

该输出表示 app_rd_data [] 总线上的数据有效。

app_wdf_rdy

该输出表示写入数据 FIFO 准备好接收数据。 接受写入数据当 app_wdf_rdy 和 app_wdf_wren 都被断言时。

app_ref_req

当被置位时，该高电平有效输入请求存储器控制器向 DRAM 发送刷新命令。必须对单个周期进行脉冲以进行请求，然后至少断言，直到 app_ref_ack 信号被断言以确认请求并指示已经发送请求。

app_ref_ack

当置位时，此高电平有效输入确认刷新请求，并指示命令已从存储控制器发送到 PHY。

app_zq_req

当置位时，该高电平有效输入请求存储器控制器向 DRAM 发送 ZQ 校准命令。必须对单个周期进行脉冲以进行请求，然后至少断言，直到 `app_zq_ack` 信号被断言以确认请求并指示已经发送请求。

app_zq_ack

当有效时，此高电平有效输入确认 ZQ 校准请求，并指示命令已从存储控制器发送到 PHY。

ui_clk_sync_rst

这是从与 `ui_clk` 同步的 UI 重置。

ui_clk

这是 UI 的输出时钟。它必须是出口到外部 SDRAM 的时钟频率的一半或四分之一，这取决于 GUI 中选择的 2: 1 或 4: 1 模式。

init_calib_complete

当校准完成时，PHY 会断言 `init_calib_complete`。在向内存控制器发送命令之前，应用程序无需等待 `init_calib_complete`。

AXI4 从接口块

AXI4 从接口模块将 AXI4 事务映射到 UI 界面，为内存控制器提供业界标准的总线协议接口。在通过 MIG 工具提供的设计中，AXI4 从站接口是可选的。两种工具之间的 RTL 是一致的。MIG AXI4 总线接口的方式更多用于 SOC 的开发，本章教程暂时不讲解 AXI4 总线方式，读写 MIG。

3.1.2.2 时钟架构(Clocking Architecture)

PHY 设计要求使用 PLL 模块来生成各种时钟，全局和本地时钟网络都用于在整个设计中分配时钟。PHY 还需要与 PLL 相同的一个 MMCM。该 MMCM 补偿 BUFG 到 PHY 的插入延迟。时钟生成和分配电路和网络驱动 PHY 内的块，大致分为四个单独的一般功能：

- 内部（FPGA）逻辑
- 写入路径（输出）I/O 逻辑
- 读路径（输入）和延迟 I/O 逻辑
- IDELAY 参考时钟

对于 DDR3 设计，IDELAY 参考时钟生成需要一个 MMCM。如果设计频率 > 667 MHz，则 IDELAY 参考时钟为 300MHz 或 400MHz（取决于 FPGA 速度等级）。MIG 实例化一个 MMCM，用于 300MHz 和 400MHz 时钟生成。

PHY 需要一个 MMCM 和一个 PLL。PLL 用于为大多数内部逻辑，相位器的频率参考时钟以及在多 I/O bank 实现中保持 PHY 控制块保持同步所需的同步脉冲生成时钟。

对于 400MHz 和 933MHz 之间的 DDR3 SDRAM 时钟频率，两个相位器频率参考时钟的频率都与存储器时钟频率相同。对于低于 400MHz 的 DDR2 或 DDR3 SDRAM 时钟频率，其中一个移相器频率参考时钟以与存储器时钟相同的频率运行，而第二个频率参考时钟必须是存储器时钟频率的 2x 或 4x，使其满足范围要求 400 MHz 至 933 MHz。两个相位器频率参考时钟必须由相同的 PLL 产生，因此它们彼此同相。时钟架构的框图如图 3-2 所示。freq_refclk 的相位根据操作频率和为存储器接口引脚选择的 bank 而有所不同。

- 在 GUI 中为存储器接口引脚选择 HP 存储区并且存储频率 $\geq 400\text{MHz}$ 时，相位为 331.5° 。

- 在 GUI 中为存储器接口引脚选择 HP 存储区时，存储器频率在 200-400 MHz（不包括 400MHz）之间，相位为 315° 。

- 对于低电压设备，当选择 HP 存储区用于 GUI 中的存储器接口引脚并且存储器频率 $\geq 400\text{MHz}$ 时，相位为 331.5° 。

- 对于在 GUI 中为存储器接口引脚选择 HP 存储区并且存储器频率在 200-400 MHz（不包括 400MHz）之间的低压设备时，相位为 0° 。

- 当在 GUI 中为存储器接口引脚选择 HR 库并且存储器频率 $\geq 400\text{MHz}$ 时，相位为 331.5° 。

- 当在 GUI 中为存储器接口引脚选择 HR 库时，存储器频率在 200-400MHz（不包括 400MHz）之间，相位为 0° 。

PLL 乘法 (M) 和除 (D) 值的默认设置是系统时钟输入频率等于存储器时钟频率。不需要 1: 1 的比例。只要符合 PLLE2 操作条件，并遵守列出的其他约束，PLL 输入分频器 (D) 可以是“7 系列 FPGA 时钟资源用户指南” (UG472) [参考文献 10]中列出的任何值。PLL 乘法 (M) 值必须在 1 和 16 之间，包括 1 和 16。用于存储器时钟的 PLL 输出分频器 (O) 对于 800Mb/s 及以上必须为 2，对于 400 至 800Mb/s，必须为 2。PLL VCO 频率范围必须保持在硅数据表中指定的范围内。sync_pulse 必须是 mem_refclk 频率的 1/16，必须具有 1/16 或 6.25% 的占空比。有关 PLL 和系统时钟 CCIO 输入的物理放置的信息，请参阅设计指南，第 192 页。

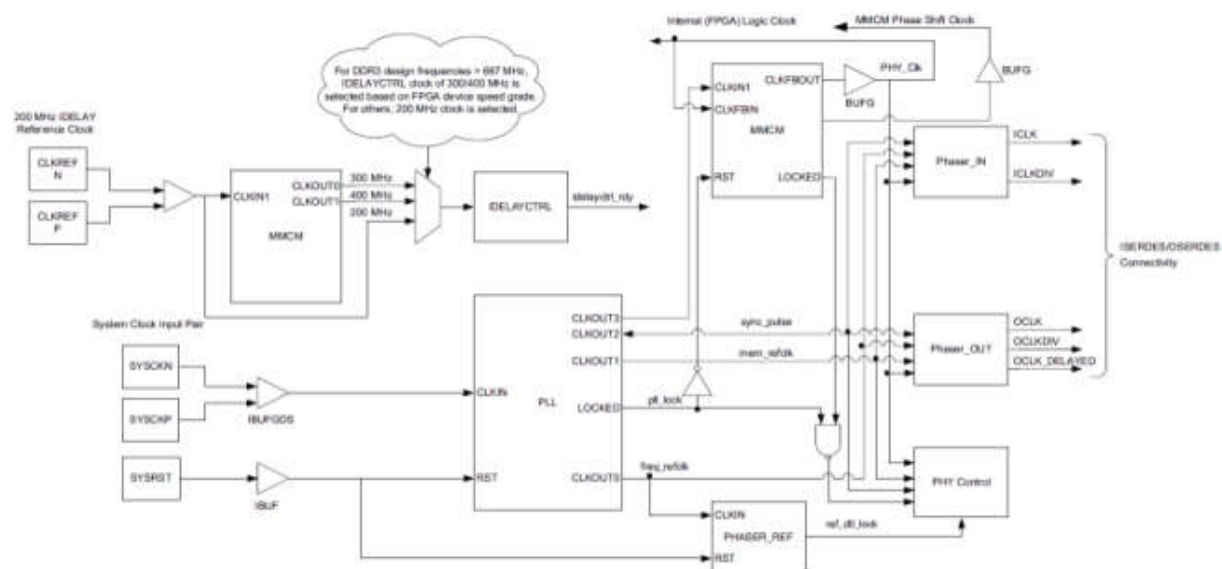


图 3-2 Clocking Architecture

ISERDES/OSERDES 连接的详细信息如图 3-3 和图 3-4。

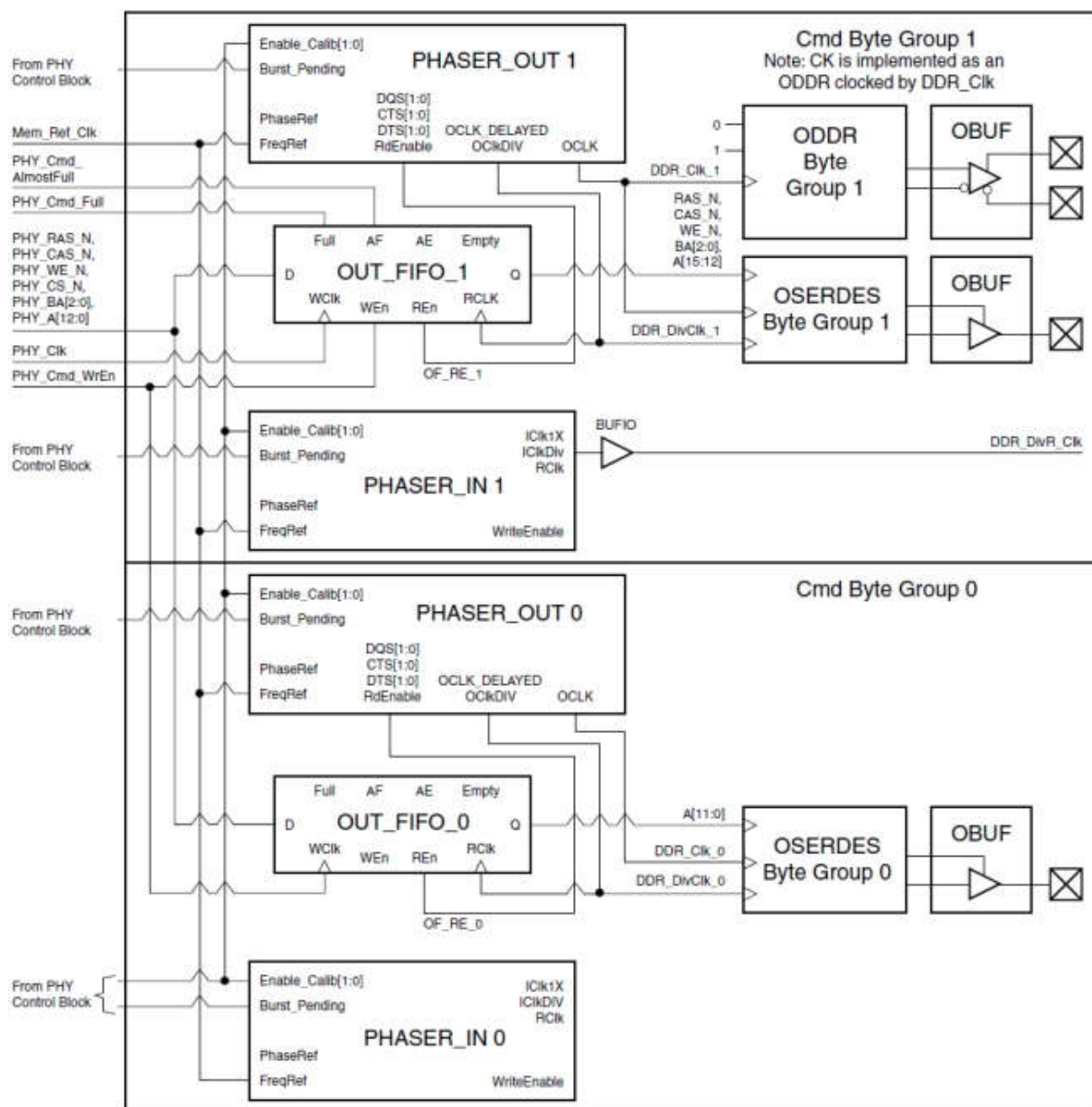


图 3-3 Address/Command Path Block Diagram

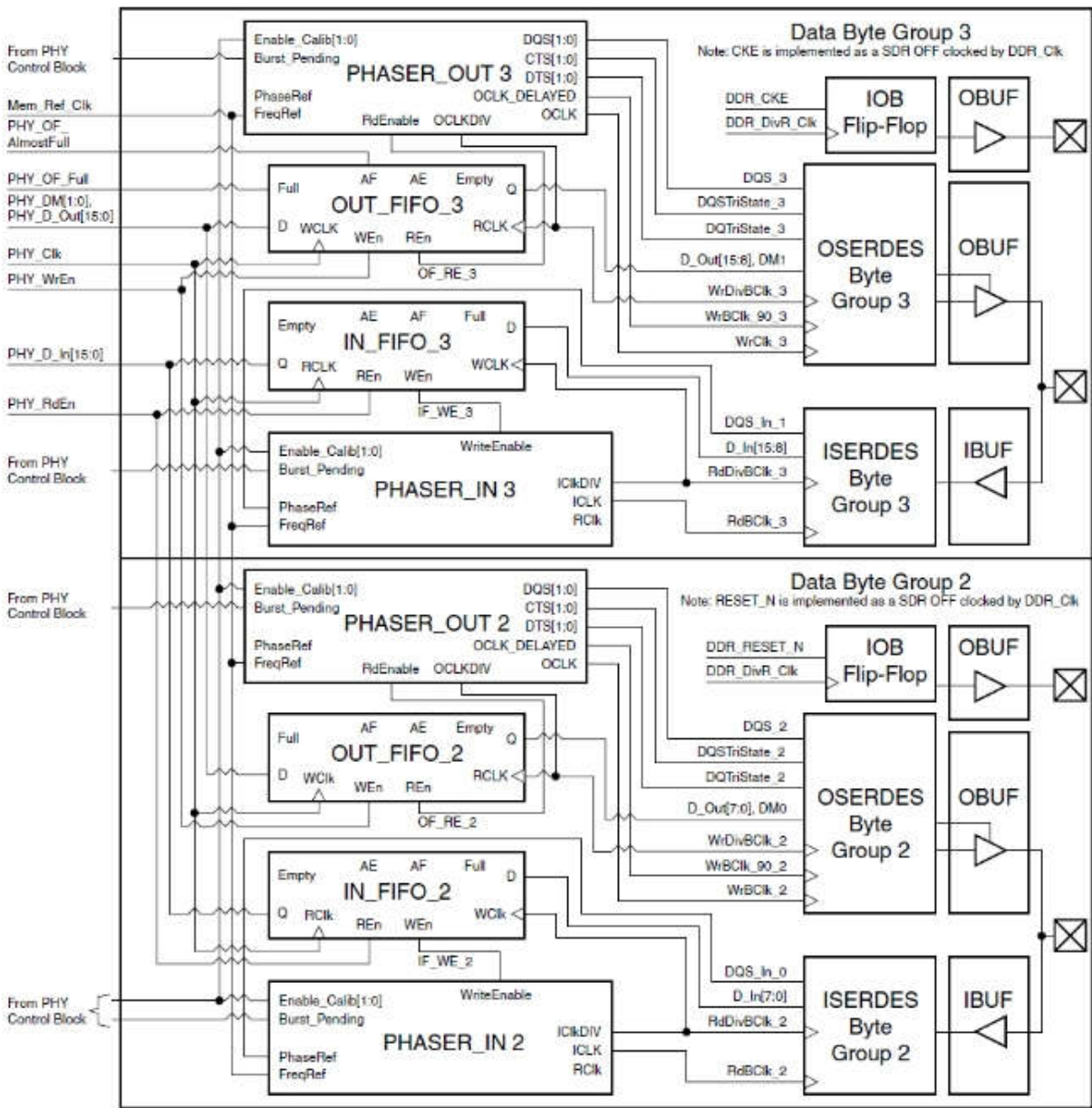


图 3-4 Datapath Block Diagram

内部（FPGA） 逻辑时钟(Internal (FPGA) Logic Clock)

内部 FPGA 逻辑由 DDR2 或 DDR3 SDRAM 时钟频率的一半或四分之一频率的全局时钟资源提供时钟，这取决于 MIG 工具中选择的 4：1 或 2：1 模式。该 PLL 还输出高速 DDR2 或 DDR3。

内存时钟。

写路径（输出） I/O 逻辑时钟(Write Path (Output) I/O Logic Clock)包含数据和控制的输出路径由 PHASER_OUT 计时。 PHASER_OUT 为 OUT_FIFO 和 OSERDES/ODDR 的每个字节组提供同步时钟。HASER_OUT 为其相关联的字节组生成字节时钟（COCLK），分频字节时钟（OCLKDIV）和延迟字节时钟（OCLK_DELAYED）。这些时钟直接从频率参考时钟生成，并且彼此同相。字节时钟

频率与频率参考时钟频率相同，分频字节时钟是频率参考时钟频率的一半。OCLK_DELAYED 用于对 DQS ODDR 进行时钟以实现写 DQS 与其相关 DQ 位之间所需的 90° 相位偏移。PHASER_OUT 还驱动在写入期间产生 DQS 所需的信号，与数据字节组相关联的 DQS 和 DQ 3 状态以及字节组的 OUT_FIFO 的读取使能。使用 PHASER_OUT 的地址/控制和写入路径的时钟细节如图 3-3 和图 3-4 所示。

读路径（输入） I/O 逻辑时钟(Read Path (Input) I/O Logic Clock)

输入读取数据路径由 PHASER_IN 块计时。PHASER_IN 块为 IN_FIFO 和 IDDR / ISERDES 的每个字节组提供同步时钟。PHASER_IN 块接收相关字节组的 DQS 信号，并为 DDR2 或 DDR3SDRAM 数据捕获产生两个延迟时钟：读字节时钟（ICLK）和读分频字节时钟（ICLKDIV）。ICLK 是与相关 DQS 相对齐的频率参考时钟的延迟版本。ICLKDIV 用于将数据捕获到 ISERDES 中的第一级触发器中。ICLKDIV 与 ICLK 对齐，是 ISERDES 中触发器最后一级的并行传输时钟。ICLKDIV 也用作与字节组相关联的 IN_FIFO 的写时钟。PHASER_IN 块还驱动字节组的 IN_FIFO 的写使能（WrEnable）。使用 PHASER_IN 的读取路径的时钟细节如图 3-3 所示。

IDELAY 参考时钟(IDELAY Reference Clock)

用户必须提供 200 MHz ref_clk，然后 MIG 会使用附加的 MMCM 创建相应的 IDELAYCTRL 频率。IDELAYCTRL 模块会持续校准 I/O 区域中的 IDELAY 元件，以解决不同的环境条件。IP 内核假定外部时钟信号正在驱动 IDELAYCTRL 模块。如果 PLL 时钟驱动 IDELAYCTRL 输入时钟，PLL 锁定信号需要并入 IODELAY_CTRL.v 模块中的 rst_tmp_idelay 信号中。这样可以确保时钟在使用前是稳定的。

3.1.2.3 MIG 内存控制器用户逻辑时序

本小节讲解控制命令、读命令、写命令极其时序。

命令路径(Command Path)

当用户逻辑 app_en 信号被断言并且 app_rdy 信号从 UI 被断言时，命令被 UI 接受并写入 FIFO。当 app_rdy 被取消置位时，UI 会忽略该命令。用户逻辑需要将 app_en 保持为高电平以及有效的命令和地址值，直到 app_rdy 被断言，如图所示。

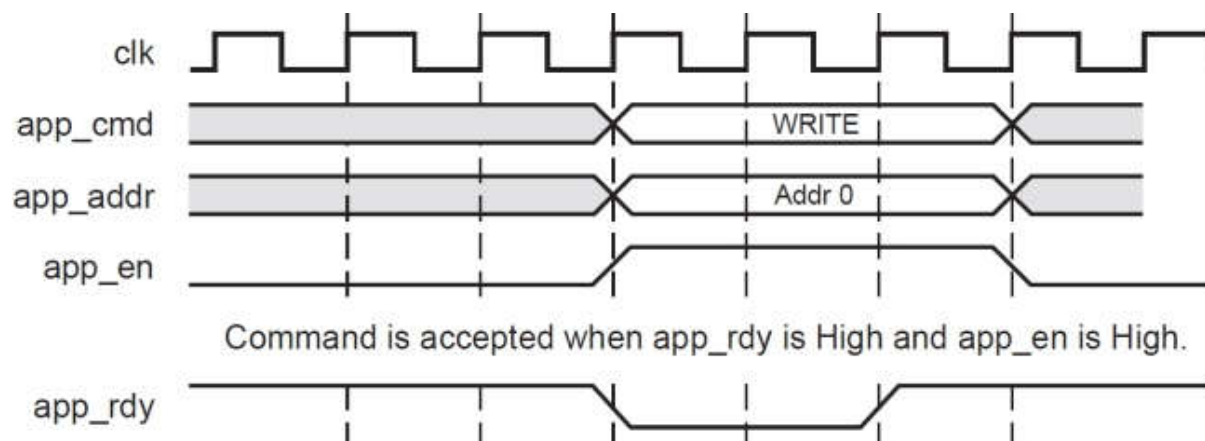


图 3-5 UI Command Timing Diagram with app_rdy Asserted

可以发出非背靠背写入命令，如图 1-54 所示。此图描述了 `app_wdf_data`，`app_wdf_wren` 和 `app_wdf_end` 信号的三种场景，如下所示：

1. 写入数据以及相应的写入命令（BL8 的下半部分）。
2. 写入数据在相应的写入命令之前。
3. 写入数据在相应的写命令之后，不应超过两个时钟周期的限制。

对于在写入命令后输出的写入数据，如注 3 所示，最大延迟为两个时钟周期。

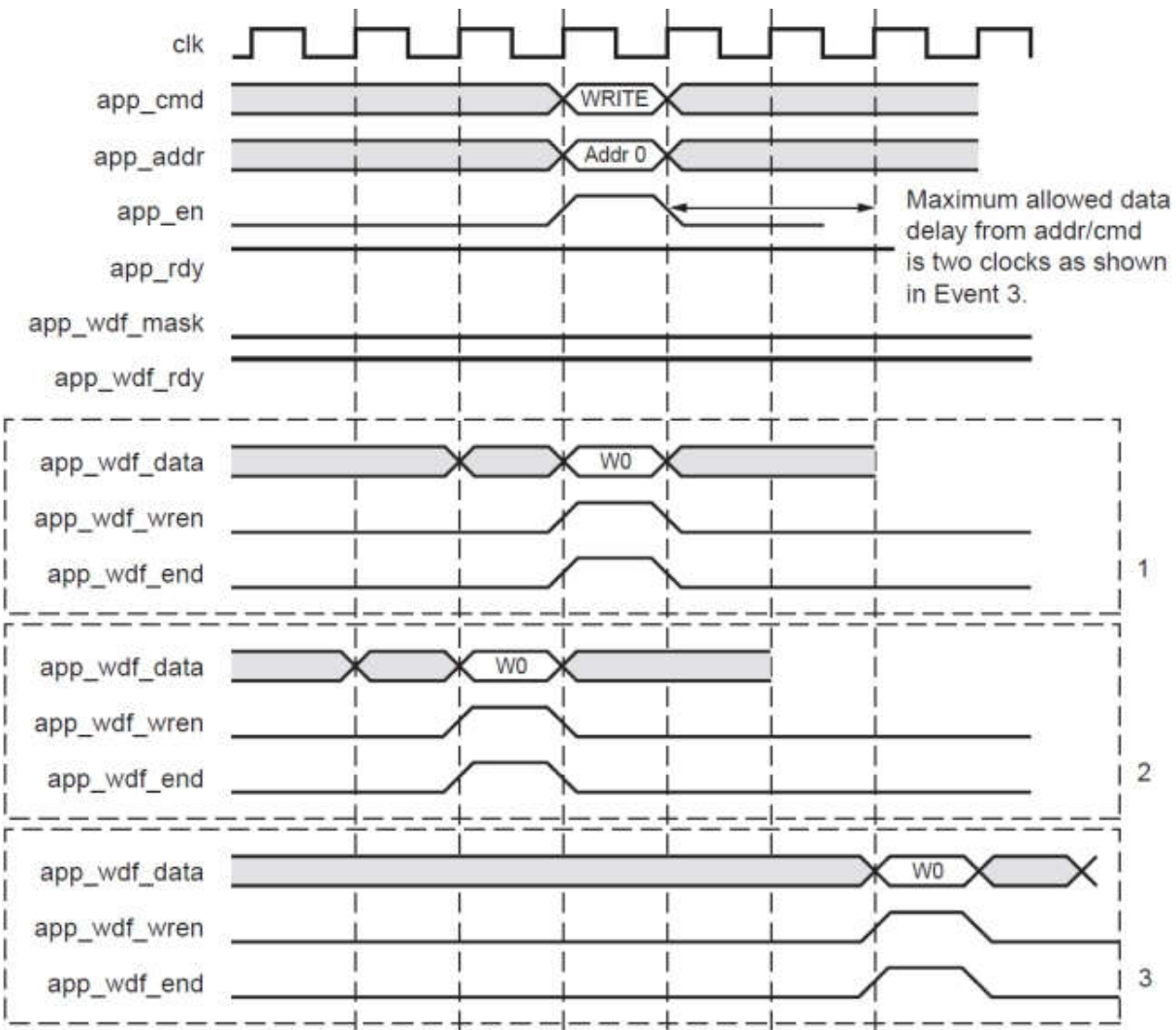


图 3-6 Mode UI Interface Write Timing Diagram (Memory Burst Type = BL8)

写路径

当 `app_wdf_wren` 被断言并且 `app_wdf_rdy` 为高时，写数据被写入写入 FIFO（图 3-6）。如果 `app_wdf_rdy` 被取消置位，则用户逻辑需要保留 `app_wdf_wren` 和 `app_wdf_end` 以及有效的 `app_wdf_data` 值，直到 `app_wdf_rdy` 被断言。`app_wdf_mask` 信号可用于屏蔽写入外部存储器的字节。

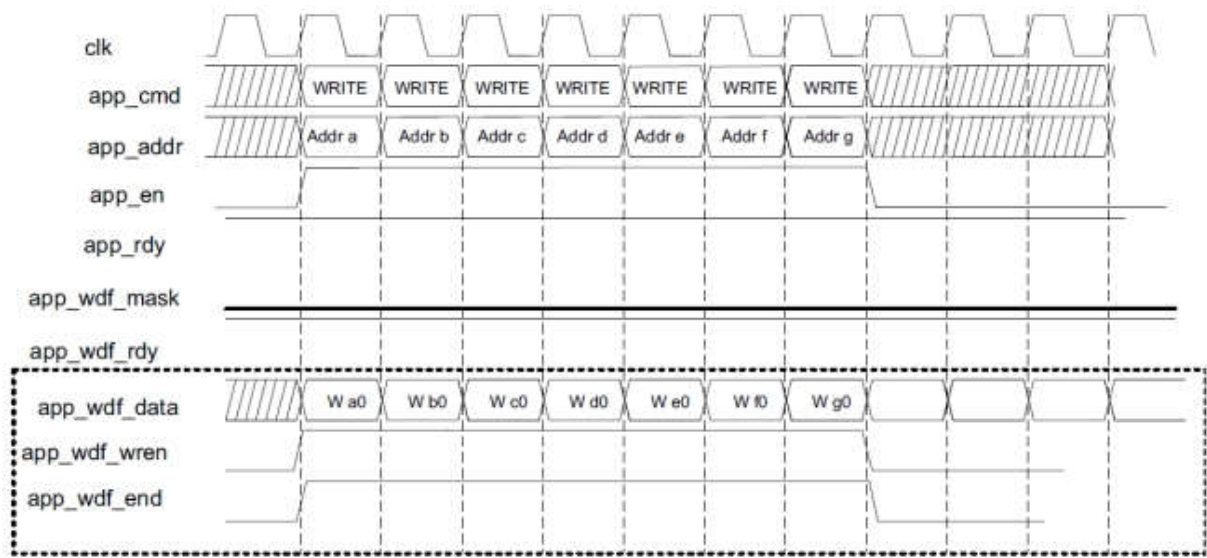


图 3-7 Mode UI Interface Back-To-Back Write Commands Timing Diagram (Memory Burst Type = BL8)

如图 3-5 所示， 写入数据和相关联的写入命令之间的单次写入的最大延迟是两个时钟周期。 当发出背靠背写入命令时， 写入数据和相关的背靠背写命令之间没有最大延迟， 如图 3-8 所示。

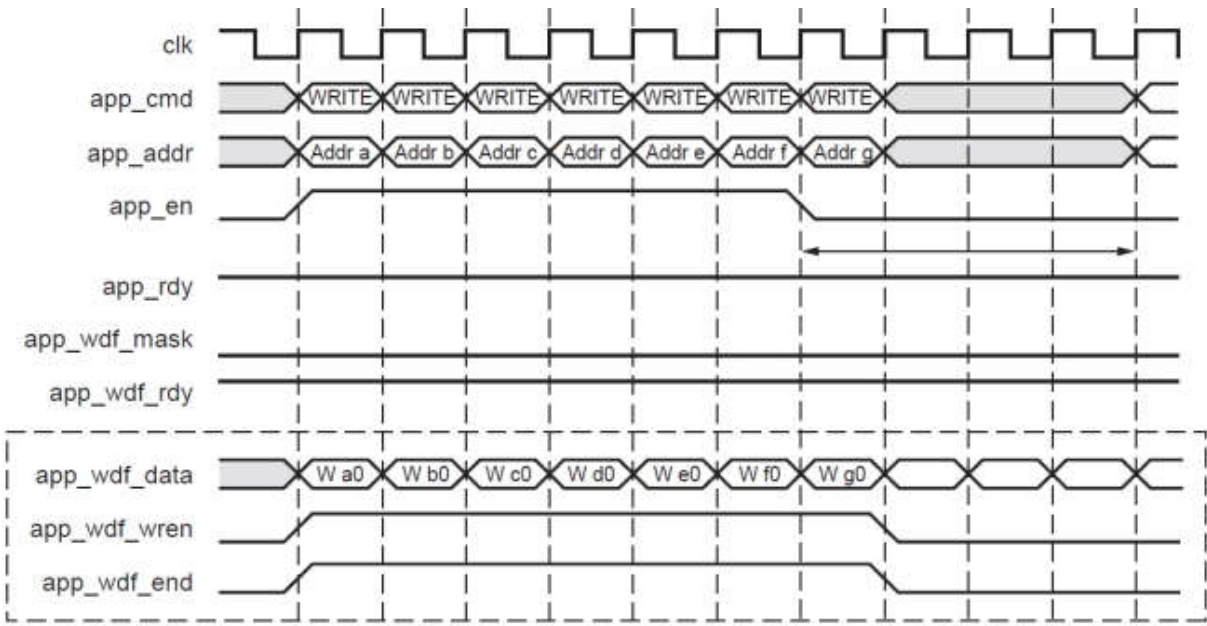


图 3-8 Mode UI Interface Back-to-Back Write Commands Timing Diagram(Memory Burst Type = BL8)

必须使用 `app_wdf_end` 信号来指示存储器写入突发的结束。 对于 8 位的内存突发类型， 应该在第二个写入数据字上断言 `app_wdf_end` 信号。 应用程序接口数据到 DRAM 输出数据的映射可以是一个例子解释。

读路径

读取的数据由 UI 以请求的顺序返回， 并且在 `app_rd_data_valid` 被断言时有效（图 3-9 和图 3-10）。 `app_rd_data_end` 信号表示每个读命令脉冲串的结束， 在用户逻辑中不需要。

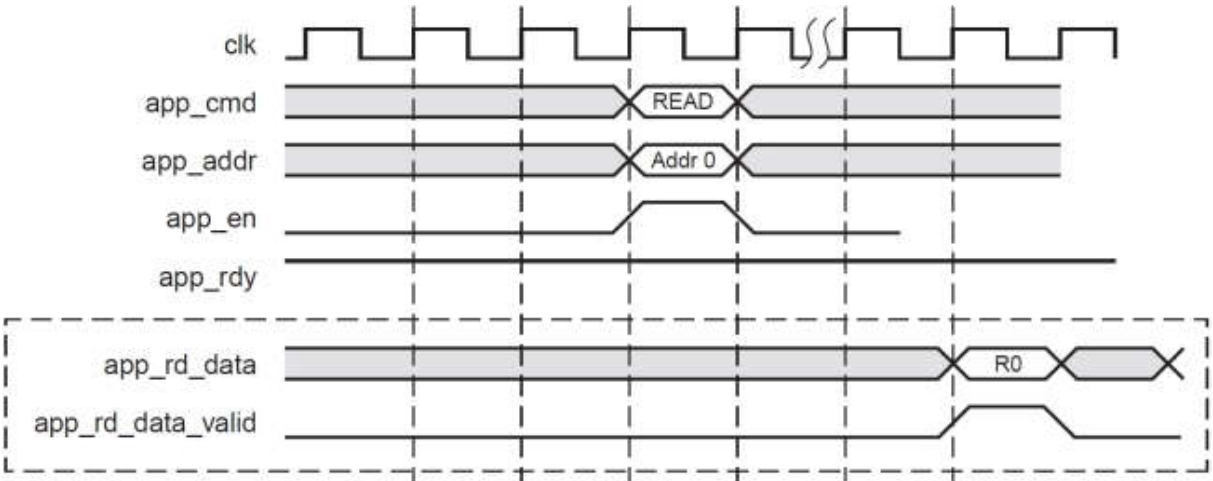


图 3-9 4:1 Mode UI Interface Read Timing Diagram(Memory Burst Type = BL8)

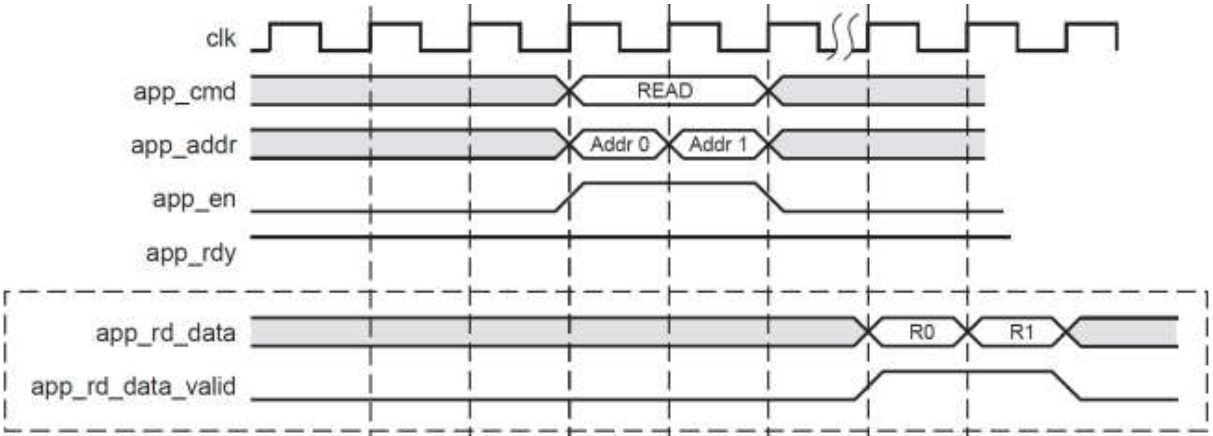


图 3-10 4:1 Mode UI Interface Read Timing Diagram(Memory Burst Type = BL4 or BL8)

在图 3-10 中， 返回的读取数据总是与地址/控制总线上的请求顺序相同。

用户刷新

对于用户控制的刷新， 应通过将 `USER_REFRESH` 参数设置为“ON” 来禁用内存控制器管理的维护。 要请求 REF 命令， `app_ref_req` 会选通一个周期。 当存储器控制器将命令发送到 PHY 时， 它会将 `app_ref_ack` 选通一个周期， 之后可以发送另一个请求。 界面如图 3-11 所示。

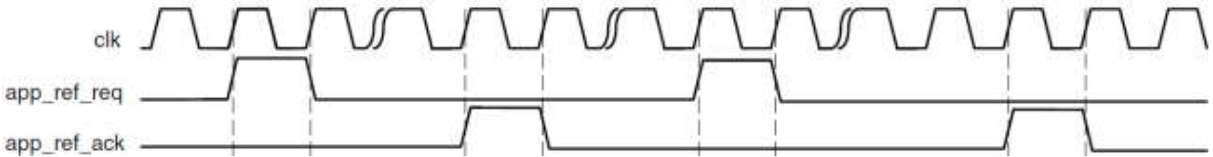


图 3-11 User-Refresh Interface

只要遵循上述定义的握手，可以随时执行用户刷新操作。对于其他命令，没有额外的接口要求。但是，待处理的请求会影响操作何时出现。内存控制器在发出刷新命令之前完成所有挂起的数据请求。在确定何时选择 `app_ref_req` 以避免 `tREFI` 违规时，必须考虑每个待处理请求的时序参数。为了解决最坏的情况，减去每个银行机器的 `tRCD`，`CL`，数据传输时间 `tRP`，以确保在 `tREFI` 到期之前所有事务都可以完成。
$$(tREFI - (tRCD + ((CL + 4) \times tCK) + tRP) \times nBANK_MACHS)$$
 公式 1-1 显示了 REF 请求间隔的最大值。公式 1-1 在校准后应立即发出用户 REF，以建立确定何时发送后续请求的时间基准。

3.1.3 Step By Step 修改代码

MIG 控制器官方产生的代码过于复杂，不利于学习，所以笔者将对代码做一些修改，使之适配我们开发板，以及有利于学习和消化。

Step1:修改顶层接口信号，由于开发板的核心板上是 100MHZ 单端时钟，以及复位信号是通过按钮提供，所以修改以下接口，并且增加 `clk100m_i` 信号以及 `rst_key` 信号。

```

237     output [0:0]          ddr3_cs_n,
238
239     output [3:0]          ddr3_dm,
240
241     output [0:0]          ddr3_odt,
242
243
244     // Inputs
245
246     output                tg_compare_error,
247     output                init_calib_complete,
248     input                 clk100m_i,
249     input                 rst_key
250

```

Step2:增加 PLL 时钟管理模块，从 100MHZ 输入，输出 200MHZ 时钟提供给 MIG 系统时钟和系统参考时钟,这个 200MHZ 时钟就是刚才设置的，必须确保一致。

Re-customize IP

Clocking Wizard (3.3)

Documentation IP Location Switch to Defaults

Component Name: clk_wiz_0

IF Symbol Resources

Show disabled ports

Reset clk_out1
clk_100m locked

Clock Monitor

☐ Enable Clock Monitoring

Primitive

☒ MMCM ☐ PLL

Clocking Features

☒ Frequency Synthesis ☐ Minimize Power

☒ Phase Alignment ☐ Spread Spectrum

☐ Dynamic Reconfig ☐ Dynamic Phase Shift

☐ Safe Clock Startup

Jitter Optimization

☒ Balanced ☐ Minimize Output Jitter ☐ Minimize Input Jitter Filtering

Dynamic Reconfig Interface Options

☐ SERIAL ☐ DQ ☐ Phase Duty Cycle Config ☐ Trips BIF registers

Input Clock Information

Input Clock	Port Name	Input Frequency (MHz)	Jitter Options	Input Jitter	Source
<input checked="" type="checkbox"/> Primary	clk_in1	100.000	10.000 - 200.000	UI	Global buffer
<input type="checkbox"/> Secondary	clk_in2	100.000	20.000 - 100.000	0.010	Single ended clock

Component Name: clk_wiz_0

Clocking Options Output Clocks Port Renaming MMCM Settings Summary

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)	Actual	Phase (degrees)	Actual	Duty Cycle (%)	Actual	Drives	Matched Routing
<input checked="" type="checkbox"/> clk_out1	clk_out1	200.000	200.000	0.000	0.000	50.000	50.0	SDPG	<input type="checkbox"/>
<input type="checkbox"/> clk_out2	clk_out2	100.000	N/A	0.000	N/A	50.000	N/A	SDPG	<input type="checkbox"/>
<input type="checkbox"/> clk_out3	clk_out3	100.000	N/A	0.000	N/A	50.000	N/A	SDPG	<input type="checkbox"/>
<input type="checkbox"/> clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000	N/A	SDPG	<input type="checkbox"/>
<input type="checkbox"/> clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000	N/A	SDPG	<input type="checkbox"/>
<input type="checkbox"/> clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000	N/A	SDPG	<input type="checkbox"/>
<input type="checkbox"/> clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000	N/A	SDPG	<input type="checkbox"/>

☐ USE CLOCK SEQUENCING

Clocking Feedback

Source

☒ Automatic Control On-Chip ☐ Automatic Control Off-Chip ☐ User-Controlled On-Chip ☐ User-Controlled Off-Chip

Signaling

☒ Single-ended ☐ Differential

Enable Optional Inputs / Outputs for MMCM/PLL

☒ reset ☐ power_down ☐ input_clk_stopped

☒ locked ☐ clkfbstopped

Reset Type

☒ Active High ☐ Active Low

Step3:修补并且增加如下代码


```
wire sys_rst;
wire locked;
wire clk_ref_i;
wire sys_clk_i;
wire clk_200;

assign sys_rst = ~rst_key; //复位信号
assign clk_ref_i = clk_200; //200M的参考时钟
assign sys_clk_i = clk_200; //200M的系统时钟

//时钟管理产生DDR需要的时钟
clk_wiz_0 CLK_WIZ_DDR( .clk_out1(clk_200), .reset(sys_rst), .locked(locked), .clk_in1(clk100m_i));
```

Step4:修改 MIG IP CORE 输入信号如下图红框所示

```

327 mig_7series_0 u_mig_7series_0
328 (
329 // Memory interface ports
330     .ddr3_addr          (ddr3_addr),
331     .ddr3_ba            (ddr3_ba),
332     .ddr3_cas_n         (ddr3_cas_n),
333     .ddr3_ck_n          (ddr3_ck_n),
334     .ddr3_ck_p          (ddr3_ck_p),
335     .ddr3_cke            (ddr3_cke),
336     .ddr3_ras_n         (ddr3_ras_n),
337     .ddr3_we_n          (ddr3_we_n),
338     .ddr3_dq            (ddr3_dq),
339     .ddr3_dqs_n         (ddr3_dqs_n),
340     .ddr3_dqs_p         (ddr3_dqs_p),
341     .ddr3_reset_n       (ddr3_reset_n),
342     .init_calib_complete (init_calib_complete),
343     .ddr3_cs_n          (ddr3_cs_n),
344     .ddr3_dm            (ddr3_dm),
345     .ddr3_odt           (ddr3_odt),
346 // Application interface ports
347     .app_addr           (app_addr),
348     .app_cmd            (app_cmd),
349     .app_en             (app_en),
350     .app_wdf_data       (app_wdf_data),
351     .app_wdf_end        (app_wdf_end),
352     .app_wdf_wren       (app_wdf_wren),
353     .app_rd_data        (app_rd_data),
354     .app_rd_data_end    (app_rd_data_end),
355     .app_rd_data_valid  (app_rd_data_valid),
356     .app_rdy            (app_rdy),
357     .app_wdf_rdy        (app_wdf_rdy),
358     .app_sr_req         (1'b0),
359     .app_ref_req        (1'b0),
360     .app_zq_req         (1'b0),
361     .app_sr_active      (app_sr_active),
362     .app_ref_ack        (app_ref_ack),
363     .app_zq_ack         (app_zq_ack),
364     .ui_clk             (clk),
365     .ui_clk_sync_rst    (rst),
366     .app_wdf_mask       (32'd0),
367 // System Clock Ports
368     .sys_clk_i          (sys_clk_i),
369 // Reference Clock Ports
370     .clk_ref_i          (clk_ref_i),
371     .device_temp        (device_temp),
372     .sys_rst            (locked),
373 );

```

Step5:添加内存测试的读写逻辑控制代码

代码 3-1 内存测试的读写逻辑控制代码

```

1.    //以下是读写测试
2.    // End of User Design top instance
3.    parameter    [2:0]CMD_WRITE    =3'd0;
4.    parameter    [2:0]CMD_READ    =3'd1;
5.    //parameter TEST_DATA_RANGE=24'd16777215;//全地址测试
6.    parameter TEST_DATA_RANGE=24'd2000;//部分测试
7.
8.    (*mark_debug="true"*)    wire init_calib_complete;
9.    (*mark_debug="true"*)    reg    [3:0]state=0;
10.   (*mark_debug="true"*)    reg    [23:0]Count_64=0;// 128M*2*16/256
11.   (*mark_debug="true"*)    reg    [23:0]Count_64_1=0;
12.   (*mark_debug="true"*)    reg    ProsessIn=0;//表示读写操作的包络
13.   (*mark_debug="true"*)    reg    WriteSign=0;//表示是写操作
14.   (*mark_debug="true"*)    reg    ProsessIn1=0;//表示写操作的包络
15.   reg    [ADDR_WIDTH-1:0]app_addr_begin=0;
16.   reg    [29:0]CountWrite_tem=0;
17.   reg    [29:0]CountRead_tem=0;
18.
19.
20.   (*mark_debug="true"*)    reg    [29:0]                CountWrite=0;
21.   (*mark_debug="true"*)    reg    [29:0]                CountRead=0;
22.   (*mark_debug="true"*)    wire                error_rddata=0;
23.
24.   assign    app_wdf_end                =app_wdf_wren;//两个相等即可
25.   assign    app_en                =ProsessIn?(WriteSign?app_rdy&&app_wdf_rdy:app_rdy):1'd0;//控制命令使能
26.   assign    app_cmd                =WriteSign?CMD_WRITE:CMD_READ;
27.   assign    app_addr                =app_addr_begin;
28.   assign    app_wdf_data                =Count_64_1;//写入的数据是计数器
29.   assign    app_wdf_wren                =ProsessIn1?app_rdy&&app_wdf_rdy:1'd0;
30.   always@(posedge clk)
31.       if(rst&!init_calib_complete)//
32.       begin
33.           state                <=4'd0;
34.           app_addr_begin                <=28'd0;
35.           WriteSign                <=1'd0;
36.           ProsessIn                <=1'd0;
37.           Count_64                <=24'd0;
38.       end
39.   else case(state)
40.       4'd0:    begin
41.           state                <=4'd1;
42.           app_addr_begin                <=28'd0;

```

```

43.      WriteSign          <=1'd0;
44.      ProsessIn          <=1'd0;
45.      Count_64           <=24'd0;
46.      CountWrite_tem     <=30'd0; //??0
47.      CountRead_tem      <=30'd0;
48.      CountWrite         <=CountWrite_tem; //"?D?
49.      CountRead          <=CountRead_tem;
50.      end
51.  4'd1:  begin
52.      state              <=4'd2;
53.      WriteSign          <=1'd1;
54.      ProsessIn          <=1'd1;
55.      Count_64           <=24'd0;
56.      app_addr_begin     <=28'd0;
57.      CountWrite_tem     <=CountWrite_tem+30'd1;
58.      end
59.  4'd2:  begin//写整片的 DDR3
60.      state              <=(Count_64==TEST_DATA_RANGE)&&app_rdy&&app_
wdf_rdy?4'd3:4'd2; //最后一个地址写完之后跳出状态
61.      WriteSign          <=(Count_64==TEST_DATA_RANGE)&&app_rdy&&app_
wdf_rdy?1'd0:1'd1; //写数据使能
62.      ProsessIn          <=(Count_64==TEST_DATA_RANGE)&&app_rdy&&app_
wdf_rdy?1'd0:1'd1; //写命令使能
63.      Count_64           <=app_rdy&&app_wdf_rdy?(Count_64+24'd1):Count
_64;
64.      app_addr_begin     <=app_rdy&&app_wdf_rdy?(app_addr_begin+28'd
8):app_addr_begin; //跳到下一个 (8*32=256) bit 数据地址
65.      CountWrite_tem     <=CountWrite_tem+30'd1;
66.      end
67.  4'd3:  begin
68.      state              <=(state1==4'd0)?4'd4:state;
69.      WriteSign          <=1'd0;
70.      ProsessIn          <=(state1==4'd0)?1'd1:1'd0;
71.      Count_64           <=24'd0;
72.      app_addr_begin     <=28'd0;
73.      CountWrite_tem     <=CountWrite_tem+30'd1;
74.      end
75.  4'd4:  begin//读整片的 DDR3
76.      state              <=(Count_64==TEST_DATA_RANGE)&&app_rdy?4'd0:
state;
77.      WriteSign          <=1'd0;
78.      ProsessIn          <=(Count_64==TEST_DATA_RANGE)&&app_rdy?1'd0:
1'd1;
79.      Count_64           <=app_rdy?(Count_64+24'd1):Count_64;

```

```

80.         app_addr_begin          <=app_rdy?(app_addr_begin+28'd8):app_addr_b
egin;
81.         CountRead_tem           <=CountRead_tem+30'd1;
82.         end
83.     default:begin
84.         state                    <=4'd1;
85.         app_addr_begin           <=28'd0;
86.         WriteSign                <=1'd0;
87.         ProsessIn               <=1'd0;
88.         Count_64                <=24'd0;
89.         end
90.     endcase
91.
92.     (*mark_debug="true"*)    reg    [3:0]state1=0;
93.     always@(posedge clk)//单独将写操作从上面的状态机提出来，当然也可以和上面的状态机合并到一起
94.         if(rst&!init_calib_complete)//
95.             begin
96.                 state1          <=4'd0;
97.                 ProsessIn1      <=1'd0;
98.             end
99.         else case(state1)
100.            4'd0:    begin
101.                state1          <=(state==4'd1)?4'd1:4'd0;
102.                ProsessIn1      <=(state==4'd1)?1'd1:1'd0;
103.                Count_64_1      <=24'd0;
104.            end
105.            4'd1:    begin
106.                state1          <=(Count_64_1==TEST_DATA_RANGE)&&app_rdy&&a
pp_wdf_rdy?4'd0:4'd1;
107.                ProsessIn1      <=(Count_64_1==TEST_DATA_RANGE)&&app_rdy&&a
pp_wdf_rdy?1'd0:1'd1;
108.                Count_64_1      <=app_rdy&&app_wdf_rdy?(Count_64_1+24'd1):C
ount_64_1;
109.            end
110.        default:begin
111.            state1          <=(state==4'd1)?4'd1:4'd0;
112.            ProsessIn1      <=(state==4'd1)?1'd1:1'd0;
113.            Count_64_1      <=24'd0;
114.        end
115.    endcase
116.
117.    (*mark_debug="true"*)    reg    [23:0]app_rd_data_tem=0;
118.    (*mark_debug="true"*)    reg    [23:0]cmp_data_r1=0;

```



```

119.    reg    [23:0]cmp_data_r=0;
120.    always@(posedge clk)
121.        if(rst&!init_calib_complete)begin
122.            cmp_data_r <= 24'd0;
123.        end
124.        else if(cmp_data_r==TEST_DATA_RANGE)begin
125.            cmp_data_r <= 24'd0;
126.        end
127.        else if(app_rd_data_valid) begin
128.            cmp_data_r<=cmp_data_r+1'b1;
129.        end
130.
131.    reg app_rd_data_valid_r=1'b0;
132.    always@(posedge clk) begin
133.        app_rd_data_valid_r <= app_rd_data_valid;
134.        app_rd_data_tem    <= app_rd_data;
135.        cmp_data_r1 <= cmp_data_r;
136.    end
137.
138.    assign error_rddata=(app_rd_data_tem!=cmp_data_r1)&app_rd_data_valid_r;

```

Step6:修改仿真代码中的系统时钟，默认是 200MHZ(5ns) 改为 100MHZ (100ns)

```

//*****
// The following parameters are multiplier and divisor factors for PLLE2.
// Based on the selected design frequency these parameters vary.
//*****
parameter CLKIN_PERIOD    = 20000;
                                // Input Clock Period

```

Step7:修改仿真接口

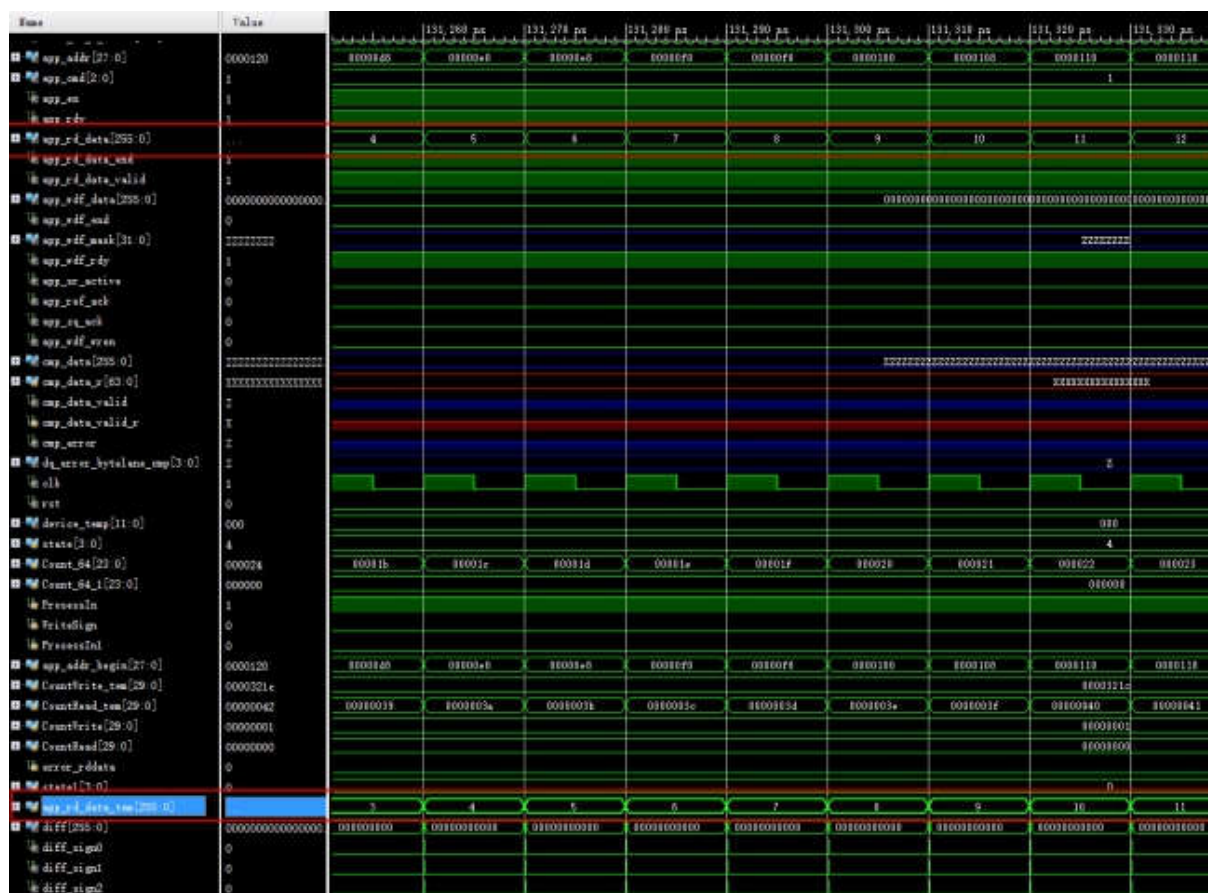
```

476     u_ip_top
477     (
478         .ddr3_dq          (ddr3_dq_fpga),
479         .ddr3_dqs_n       (ddr3_dqs_n_fpga),
480         .ddr3_dqs_p       (ddr3_dqs_p_fpga),
481
482         .ddr3_addr        (ddr3_addr_fpga),
483         .ddr3_ba          (ddr3_ba_fpga),
484         .ddr3_ras_n       (ddr3_ras_n_fpga),
485         .ddr3_cas_n       (ddr3_cas_n_fpga),
486         .ddr3_we_n        (ddr3_we_n_fpga),
487         .ddr3_reset_n     (ddr3_reset_n),
488         .ddr3_ck_p        (ddr3_ck_p_fpga),
489         .ddr3_ck_n        (ddr3_ck_n_fpga),
490         .ddr3_cke          (ddr3_cke_fpga),
491         .ddr3_cs_n        (ddr3_cs_n_fpga),
492
493         .ddr3_dm           (ddr3_dm_fpga),
494
495         .ddr3_odt          (ddr3_odt_fpga),
496
497         .clk50m_i          (sys_clk_i),
498
499         // .clk_ref_i       (clk_ref_i),
500
501         .init_calib_complete (init_calib_complete),
502         .tg_compare_error    (tg_compare_error),
503         .rst_key              (sys_rst)
504     );
505

```

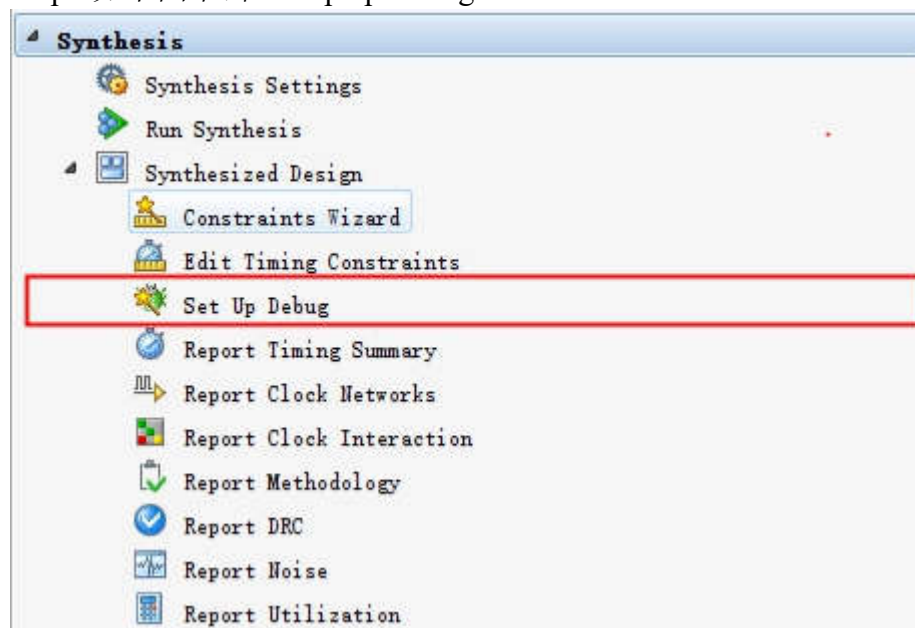
3.1.4 Step By Step RTL 仿真

Step1:在下图中可以看到笔者编写的控制代码往 MIG 写入计数器的值。

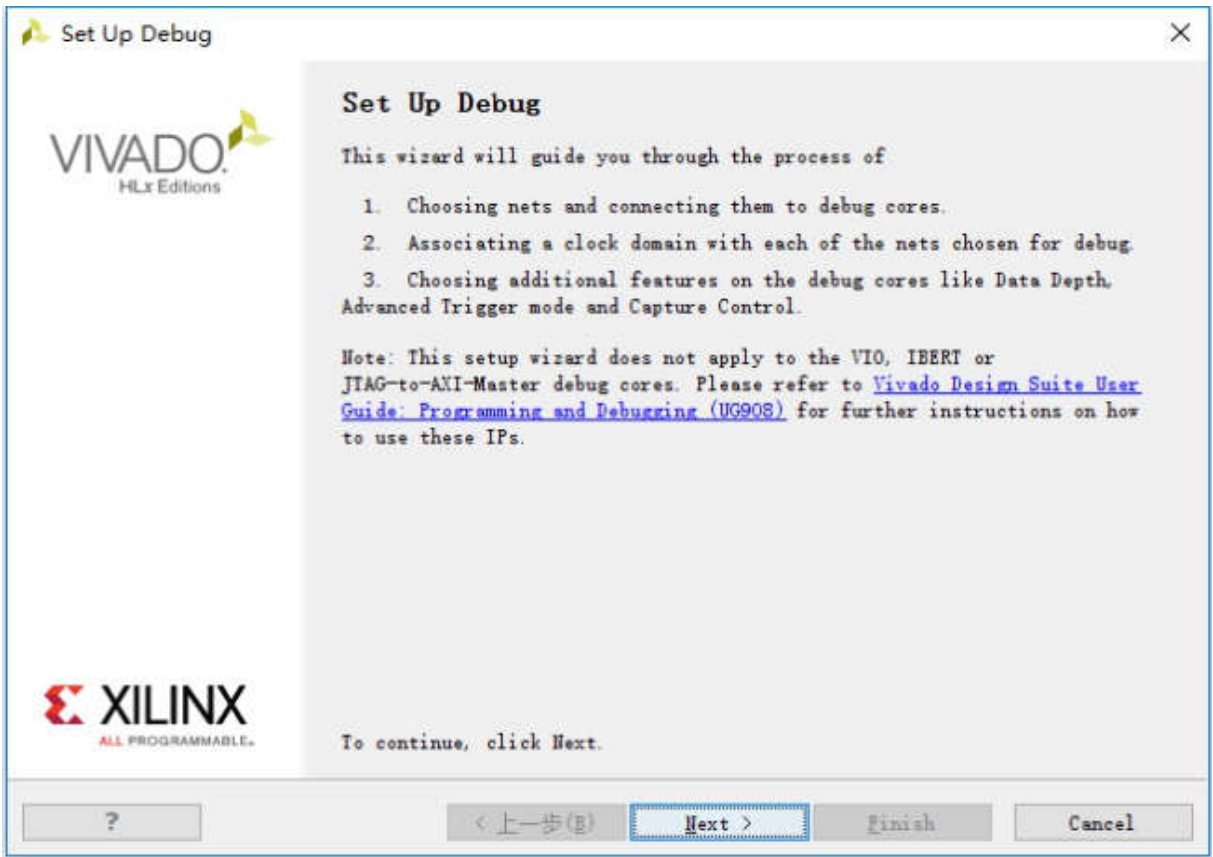


3.1.5 Step By Step 下载以及在线仿真

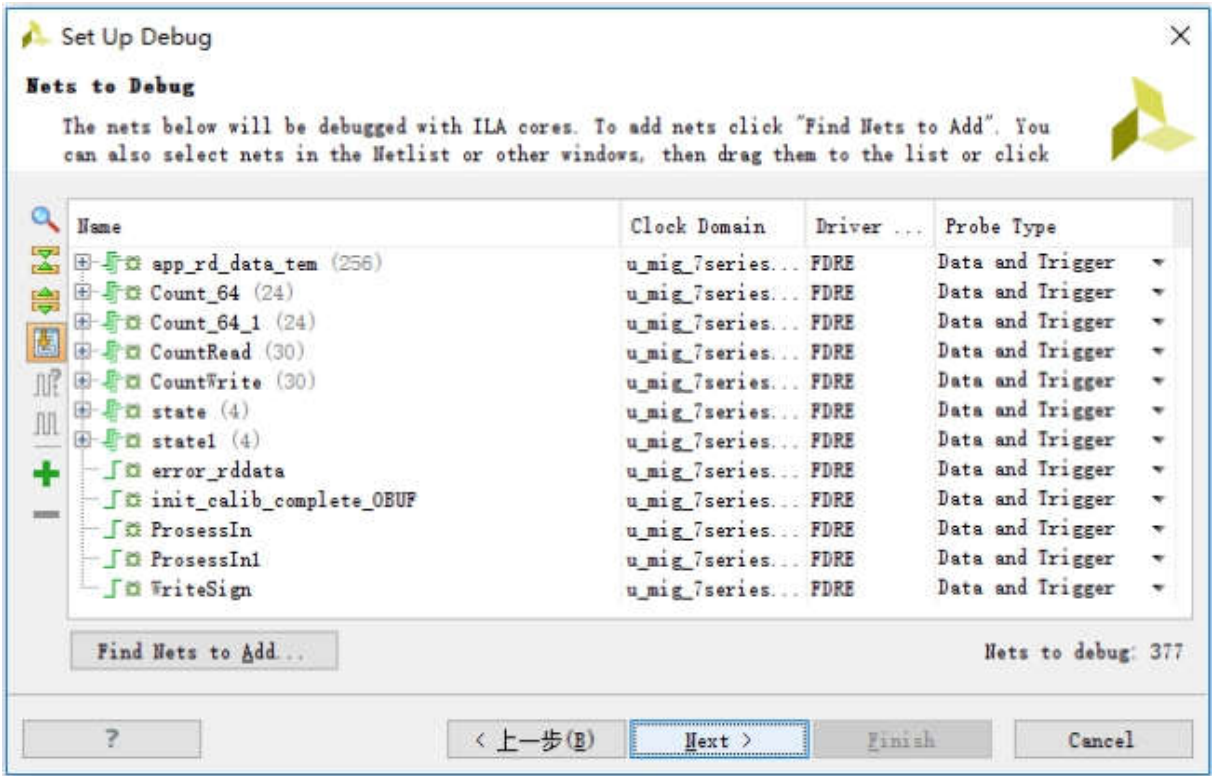
Step1:如下图单击 Setup up Debug



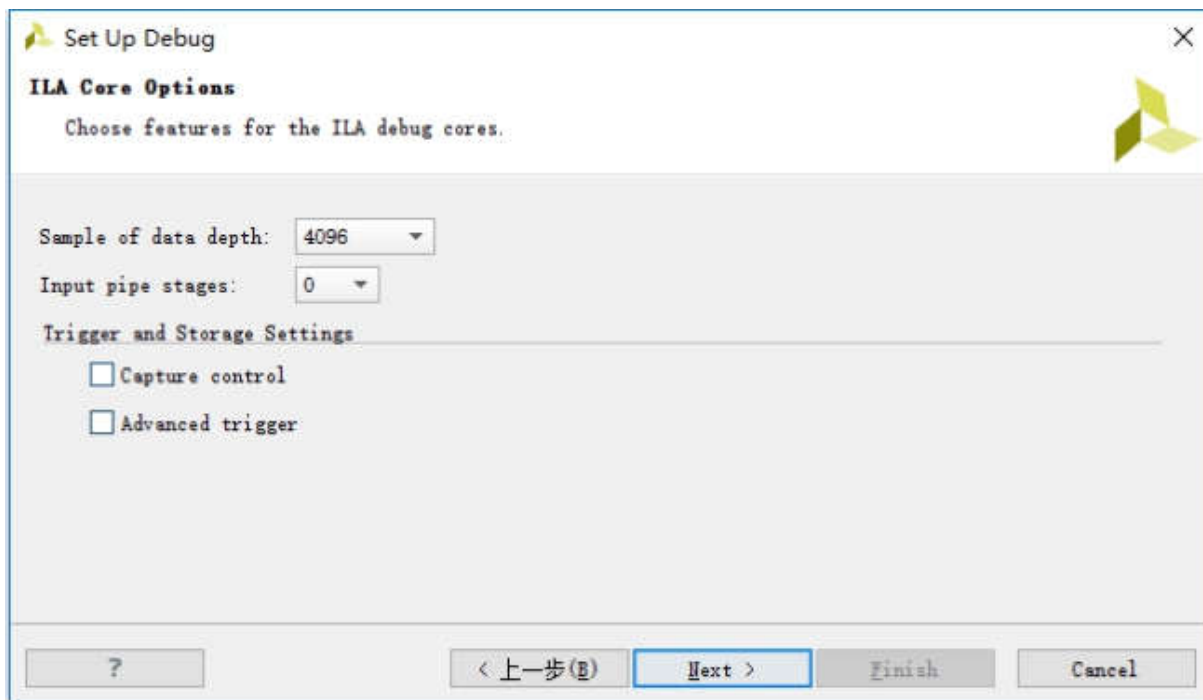
Step2:单击 NEXT



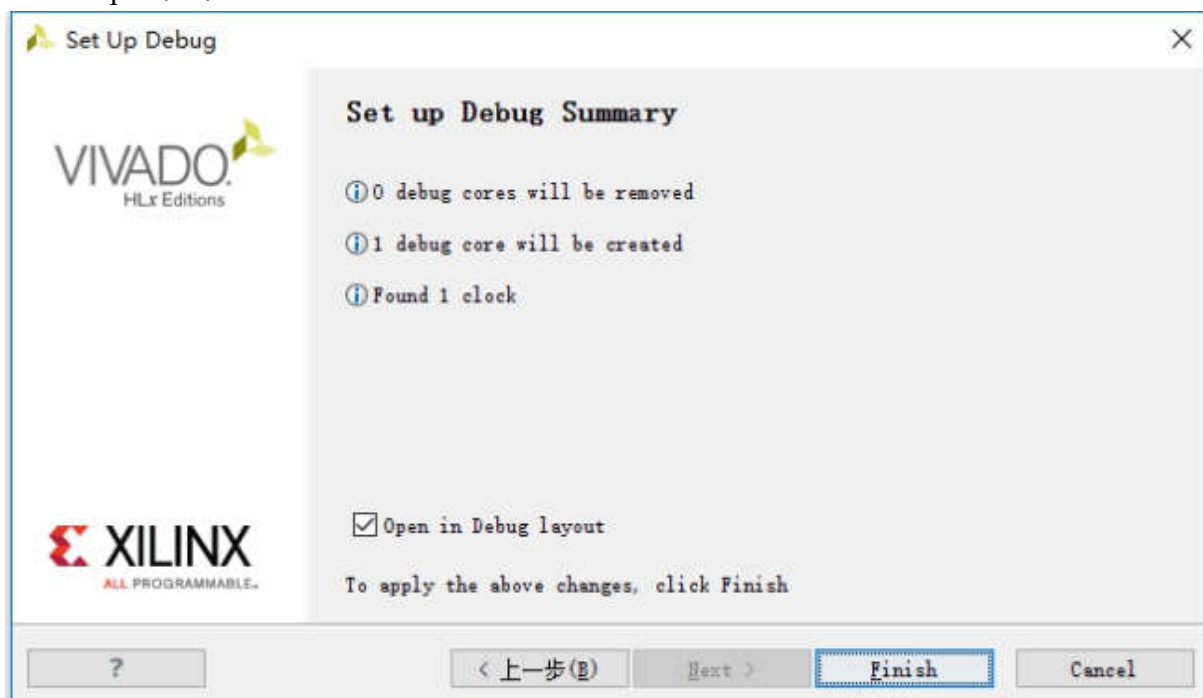
Step3:单击 NEXT



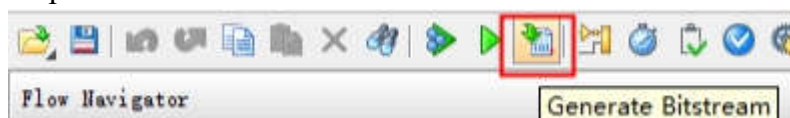
Step4:设置采样深度 4096 单击 NEXT



Step5:单击 Finish



Step6:单击 Generate Bitstream



Step7:编译完成后下载程序，并且再波形串口查看从内存读取的数据，以及是否有错误产生，可以看到数据全部正确。



对应代码《002_MIG_DDR_TEST》。

3.2 基于 OV5640 的 FPGA-DDR HDMI 显示

3.2.1 构架设计

对于稍微复杂一点的工程项目必须要有优秀的构架设计，优秀的设计构架的意义比具体写代码更有价值， 所以笔者建议读者在以后自己的项目中要多学习和思考构架的设计。笔者设计了一种基于消息缓存模式的构架，可以广泛用于图像缓存，数据采集缓存， 通信数据缓存等项目中。如下图所示，摄像头数据经过 sensor_decode 模块解码摄像头数据后将 RGB 时序图像数据输入到 MIG_BURTS_IMAGE，然后再流出来到 vga_lcd_driver 模块，vga_lcd_driver 模块输出的是 RGB 时序，进过 HDMI IP 模块后接到 HDMI 显示器显示输出。

其中，OV5640 驱动，DDR 控制器，以及 HDMI 模块这些全部都有单独实现过，所以本节主要实现的是各个 IP 核之间数据的流动，各个模块不会再重点介绍，但是 DDR 作为各个模块的中转点，还需要再介绍一些内容。

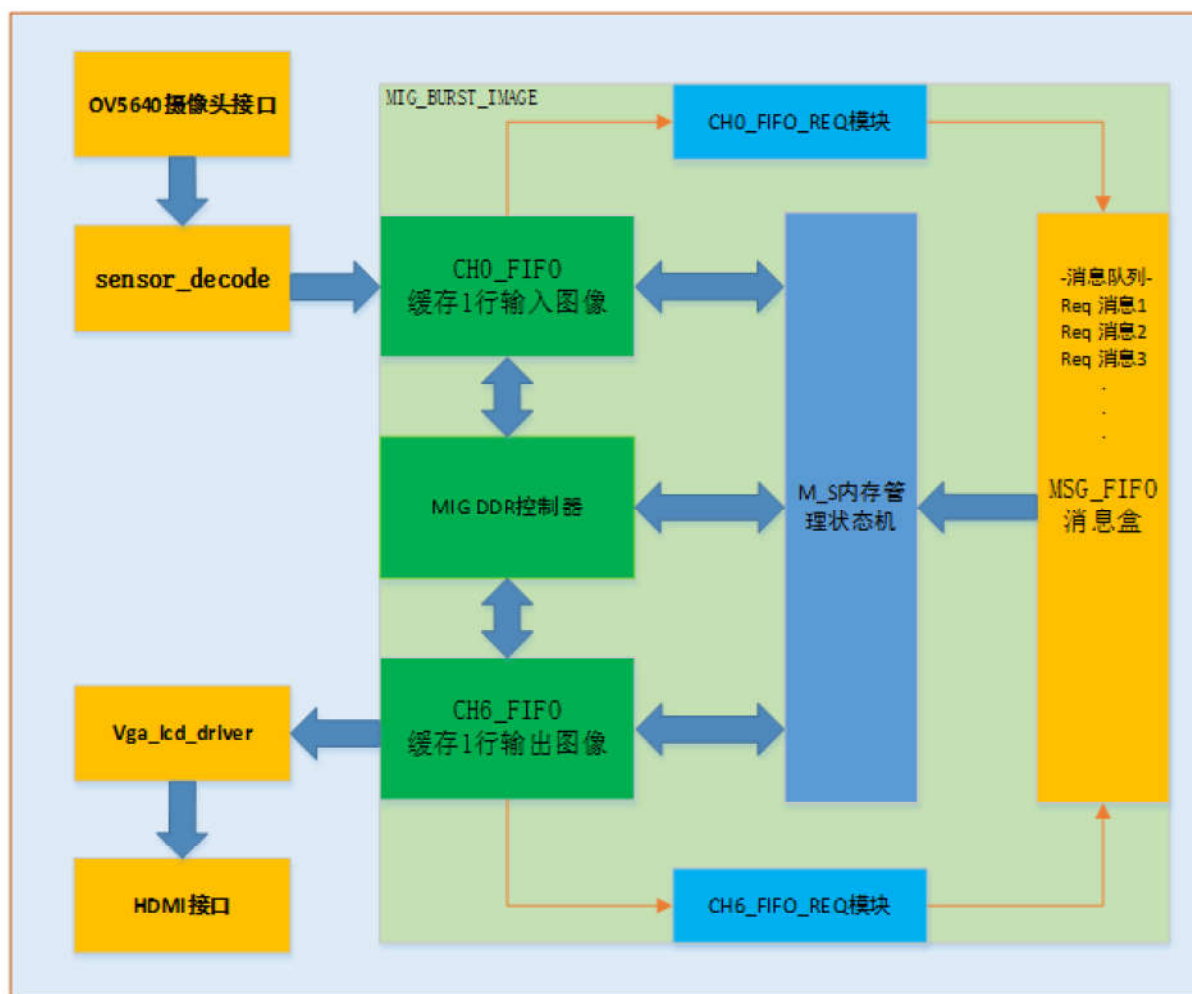


图 3-12 驱动架构即数据流向图

1)、OV5640 摄像头接口：是连接到开发板的物理接口，摄像头模块安装在这个接口上

2)、HDMI 接口：是图形的输出接口，图像数据从 HDMI 接口输出

3)、sensor_decode 模块：解码 OV5640 摄像头的图像数据，然后转为 R/G/B 数据，同时需要产生 HS 信号、VS 信号、de 信号提供给后续模块使用。如下面一段代码

```
1. assign rgb_o = {rgb565[15:11],3'd0 ,rgb565[10:5] ,2'd0,rgb565[4:0],3'd0};
2. //assign rgb_o = {grid_data_2,grid_data_2,grid_data_2};
3. assign clk_ce =out_en? byte_flag_r0:1'b0;
4. assign vs_o = out_en ? vsync_d[1] : 1'b0;
5. assign hs_o = out_en ? href_d[1] : 1'b0;
```

4)、vga_lcd_driver 模块：产生 RGB(VGA)输出时序。

5)、MIG_BURST_IMAGE 模块:管理图像数据和内存管理，MIG_BURST_IMAGE 模块中包括了 CH0_FIFO 模块、CH6_FIFO 模块、MIG_DDR 控制器、MSG_FIFO 消

息盒、M_S 内存管理状态机,此外还包括 CH0_FIFO 的读请求, 以及 CH6_FIFO 的写请求。

3.2.2 主要代码分析

本节不附源码,所有的源码请参考《003_OV5640_DDR3_DEMO\OV5640_DEMO》,每个源码分析时会附上源码文件名称,只需要找到相关文件即可。

3.2.2.1 OV5640 的寄存器配置

《i2c_timing_ctrl.v》

本文件主要实现 IIC 的时序,具体可以参考上面章节关于 IIC 的介绍。

《I2C_OV5640_RGB565_Config.v》

本文件主要是 OV5640 所有的寄存器配置,主要的寄存器都有注释,可以参考源文件。

3.2.2.2 sensor_decode.v

《sensor_decode.v》

OV5640 输入的图像是 640X480 分辨率的图形,为了观察数据的方便,未来观察行场信号是否正确,笔者先利用上一课中用到的测试图形方案数据代码 OV5640 产生的 RGB 数据,而行场信号继续使用 OV5640 产生的行场信号。这么做的目的,主要是我们在对 OV5640 解码的时候可能采样的颜色时序会出问题,但是一开始又不能定位问题,这样可以循序渐进式解决问题。

3.2.2.3 vga_lcd_driver.v

《vga_lcd_driver.v》

vga_lcd_driver 模块的作用是产生一副 640X480 分辨率的 RGB 使用时序,RGB 的数据来源来自 CH6_FIFO。读者可以修改不同分辨率输出不同的测试图形。

3.2.2.4 CH0_FIFO 模块

《MIG_BURST_IMAGE.v》

CH0_FIFO 是 IP CORE 在 MIG_BURST_IMAGE 模块中被调用，输入的图像数据需要经过这个 FIFO 把宽度是 32bit 的像素点转为 256bit(32bitX8)。这个模块主要的信号介绍如下：

CH0_FIFO_RST：用于复位 FIFO,image_data_gen 模块每次新的一帧数据过来的时候，会复位 FIFO，这样实现对 FIFO 的清除。

CH0_wclk_i：是写 FIFO 的时钟，这个时钟来自于顶层的模块和 Image_data_gen 时钟一致。

CH0_rclk_i：同 MIG 控制的用户时钟一致。

CH0_data_i：一个 32bit 的数据，测试数据一个像素是 24bit 所以 CH0_data_i 的 31bit~24bit 这里是无效的。

CH0_wren_i：写 FIFO 使能，连接到 Image_data_gen 的 de 信号上，通过 Image_data_gen 实现对 FIFO 的写。

CH0_rden_i：读 FIFO 使能，这个信号用于从 FIFO 读取 256bit 的数据，写入到 MIG 控制器

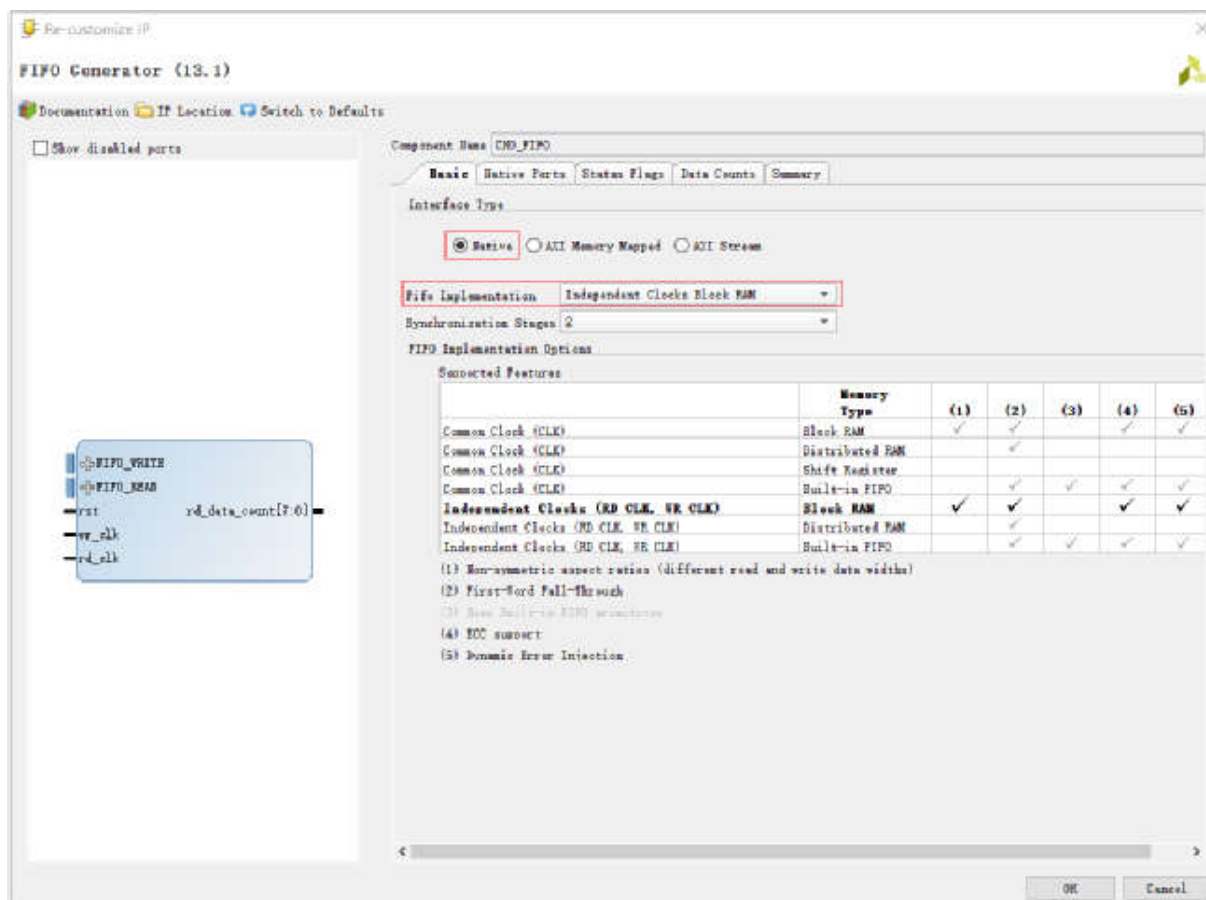
CH0_data_o：256bit 长度的数据，用于输出到 MIG 控制的，所以可以看出每一次写命令写入到 MIG 控制点的数据量是 8 个像素点。

CH0_empty：笔者 debug 的时候观察的信号，对程序没有任何作用。

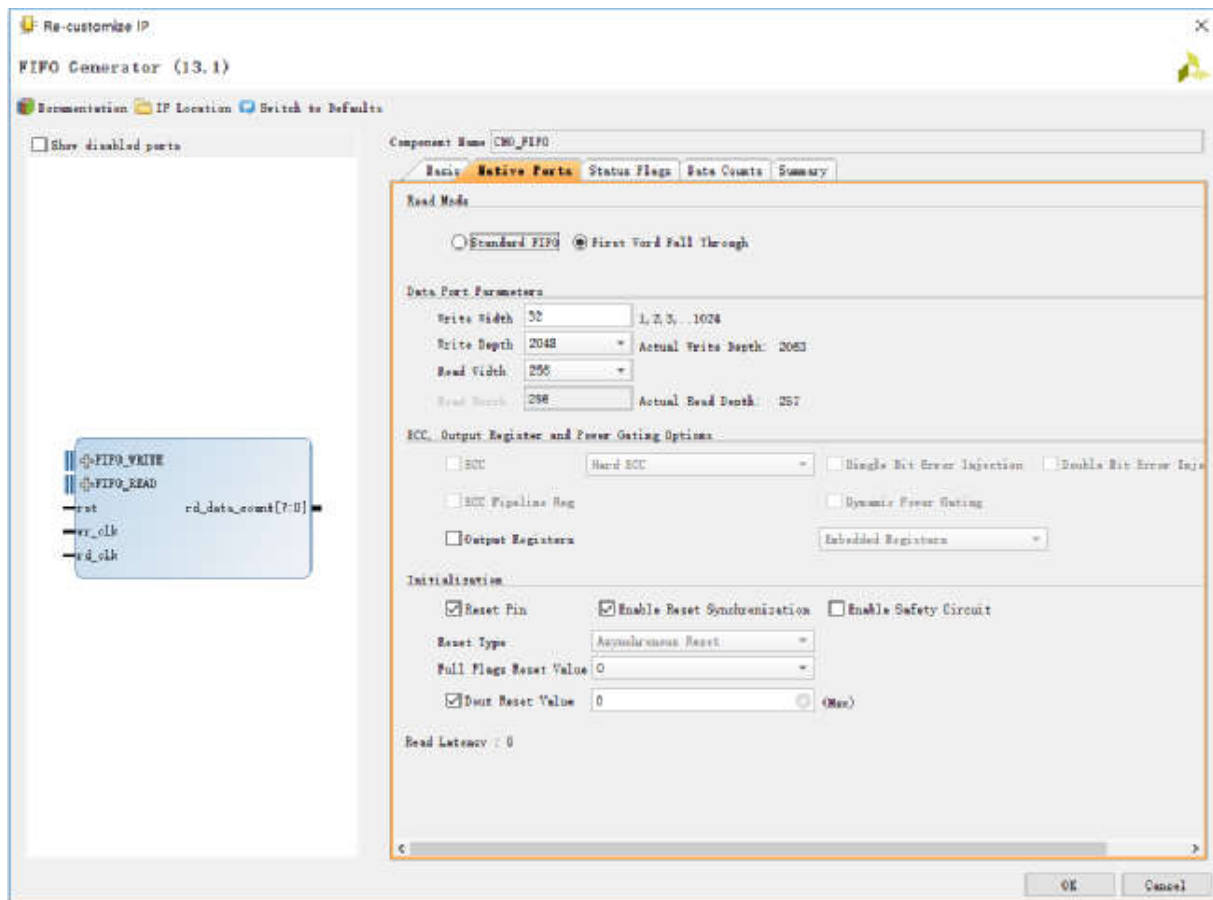
CH0_rusdw_o：用来观察 CH0_FIFO 中有多少数据可以读出来的，也是用来产生 MIG 控制器写 MIG 请求的信号。这个信号后面再讲解。

CH0_FIFO 的配置界面：

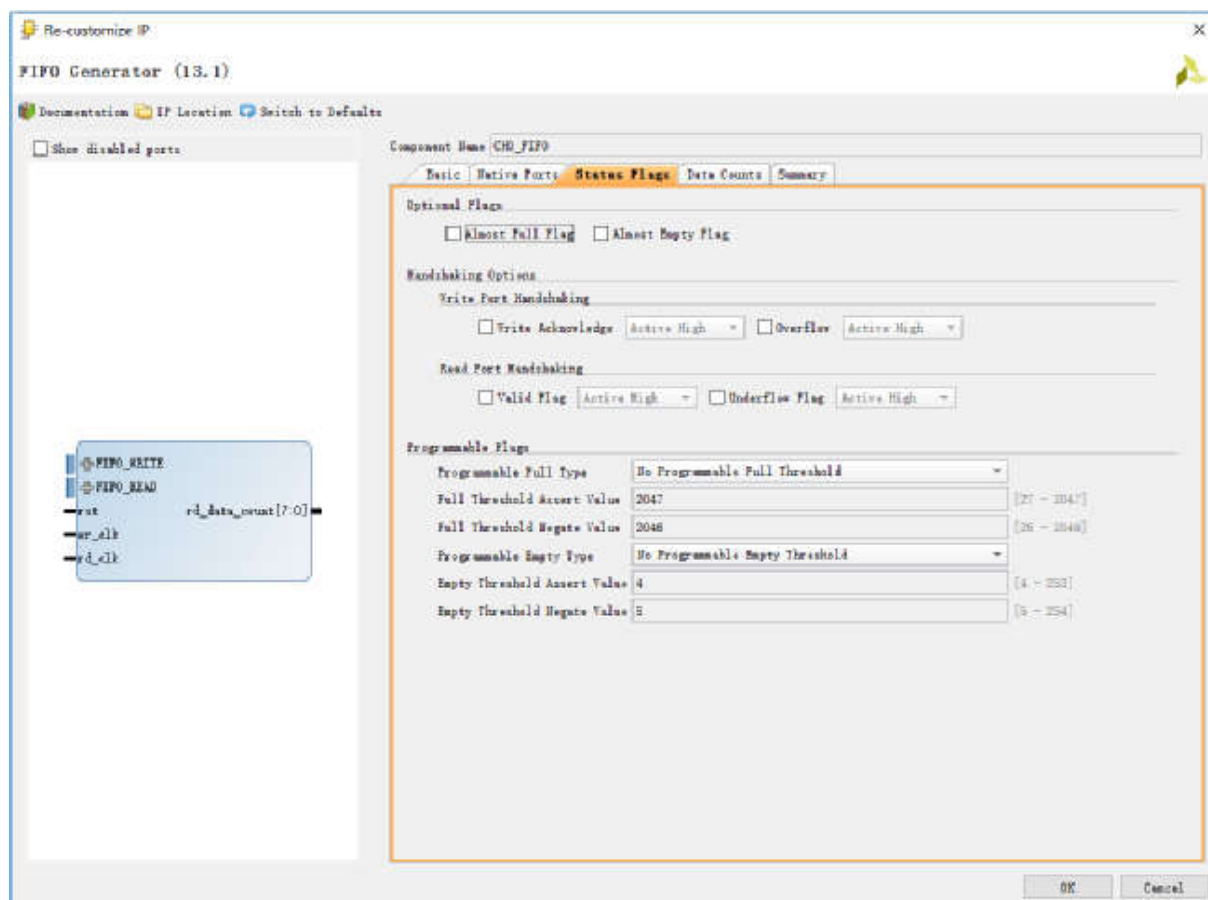
Step1:设置 Native 和 Independent Clocks Block RAM



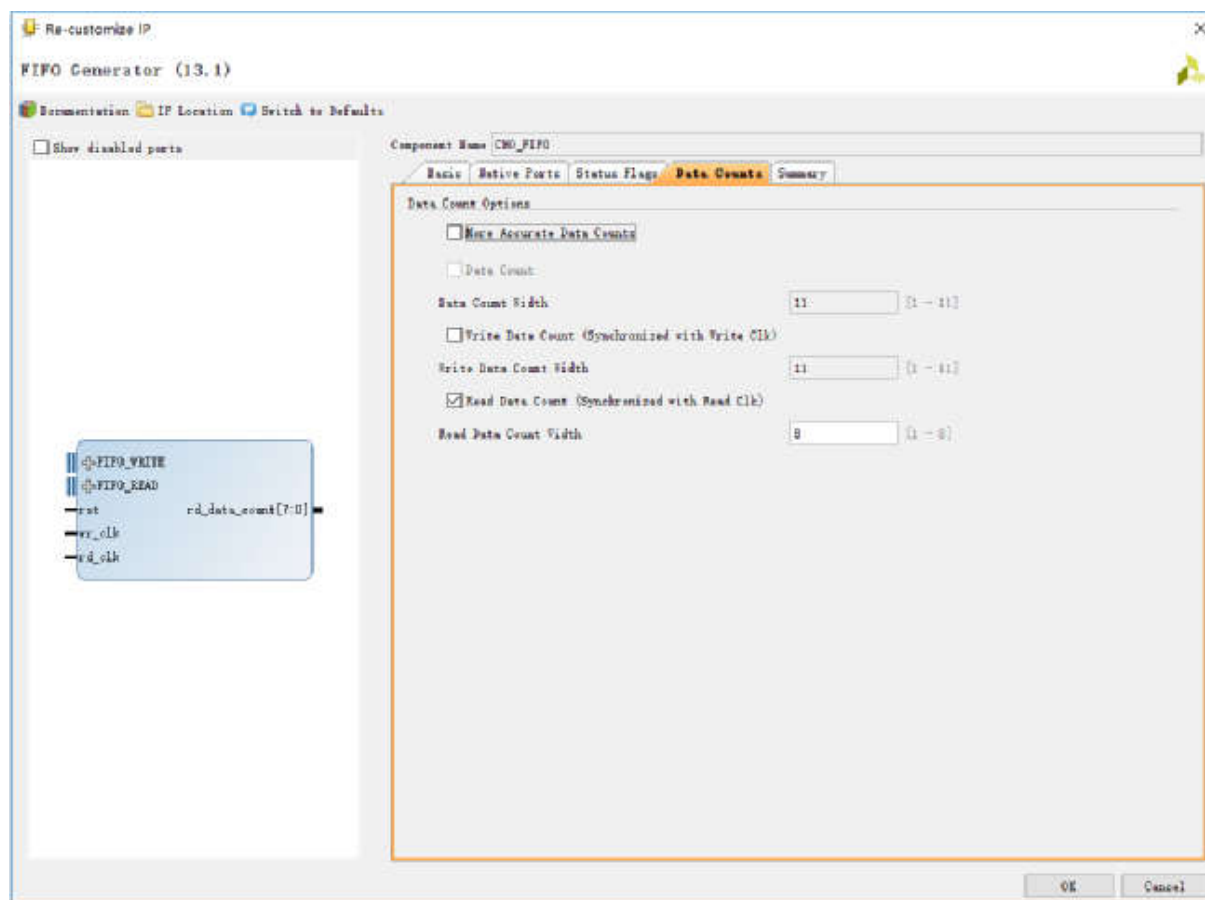
Step2:设置 FIFO 采用 First Word Fall Through 模式, FIFO 的写是 32bit 2048 深度 FIFO 的读是 256bit 256 深度



Step3:这一页默认不用设置



Step4:设置读 FIFO 的 counter 计数器宽度是 8bit



3.2.2.5 CH6_FIFO 模块

《MIG_BURST_IMAGE.v》

CH6_FIFO 是 IP CORE 在 MIG_BURST_IMAGE 模块中被调用，输出的图像数据需要经过这个 FIFO 把宽度是 256bit(32bitX8)的数据转为 32bit 的 8 个像素点。这个模块主要的信号介绍如下：

CH6_FIFO_RST 用于复位 FIFO,每次 vga_lcd_driver 模块的 VS 信号到来的时候，会复位 FIFO，这样也实现了对 FIFO 的清除。

CH6_wclk_i: 写 FIFO 的时钟，同 MIG 控制的用户时钟一致。

CH6_rclk_i: 读 FIFO 的时钟，这个时钟来自于顶层的模块和 vga_lcd_driver 时钟一致

CH6_data_i: 一个 256bit 的数据，是从 MIG 控制器读取到的内存数据。

CH6_wren_i: FIFO 使能，连接到 M_S 信号的 app_rd_data_valid 信号。

CH6_rden_i: 读 FIFO 使能，这个信号用于从 FIFO 读取 32bit 的像素数据，这个信号连接到

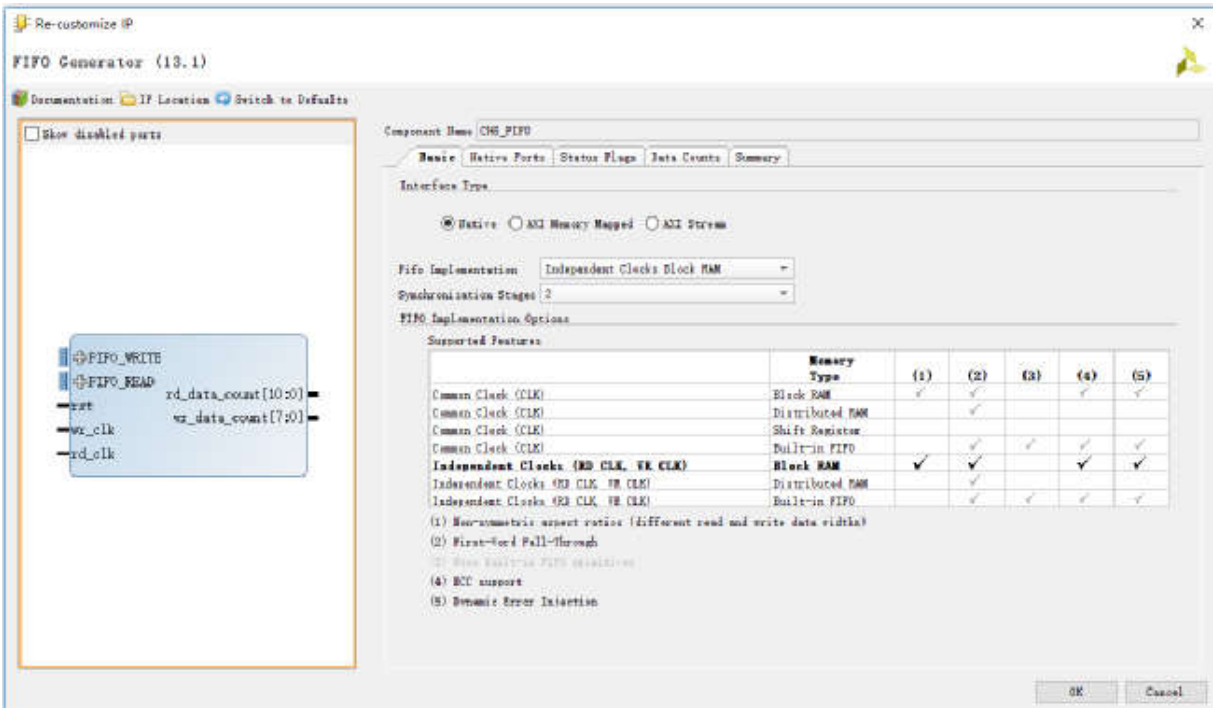
vga_lcd_driver 模块的 de 信号。

CH6_data_o 是 32bit 长度的数据， 是输出到 vga_lcd_driver 模块的像素数据。

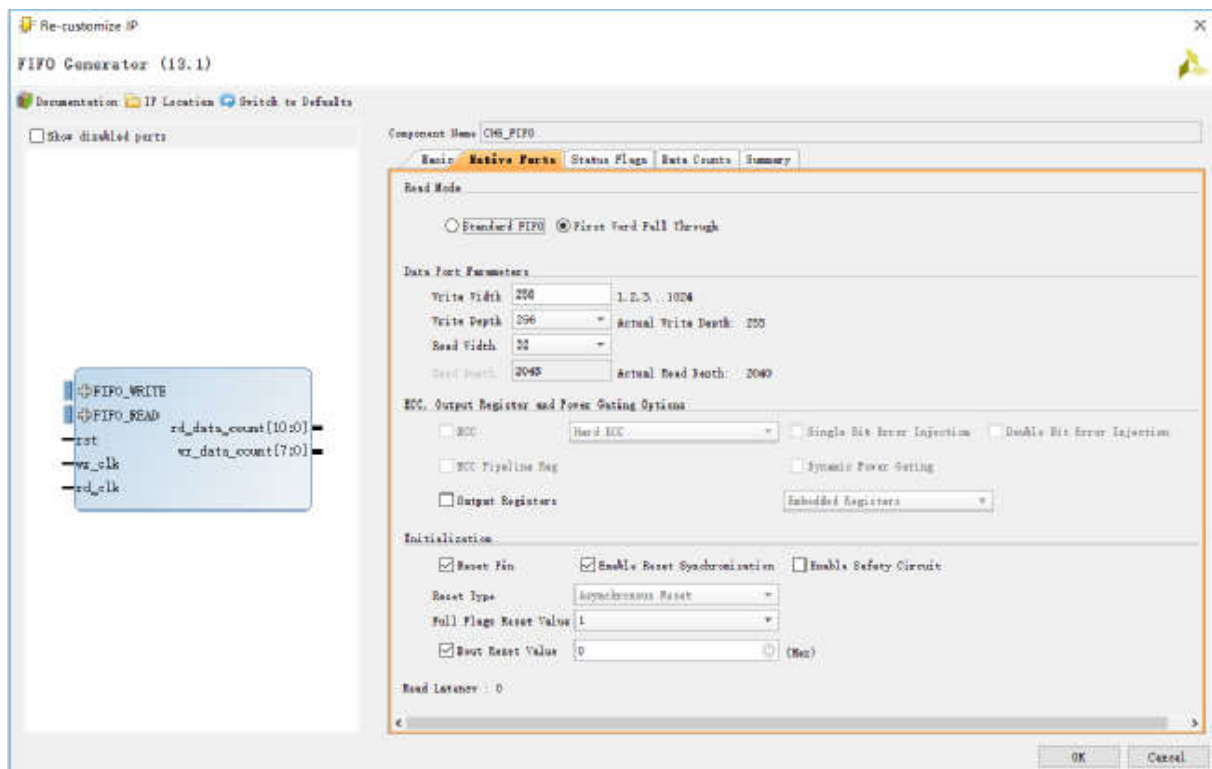
CH6_wusdw_o 是用来观察 CH6_FIFO 已经写入了多少数据。 这个信号后面再讲解。

CH6_FIFO 的配置界面：

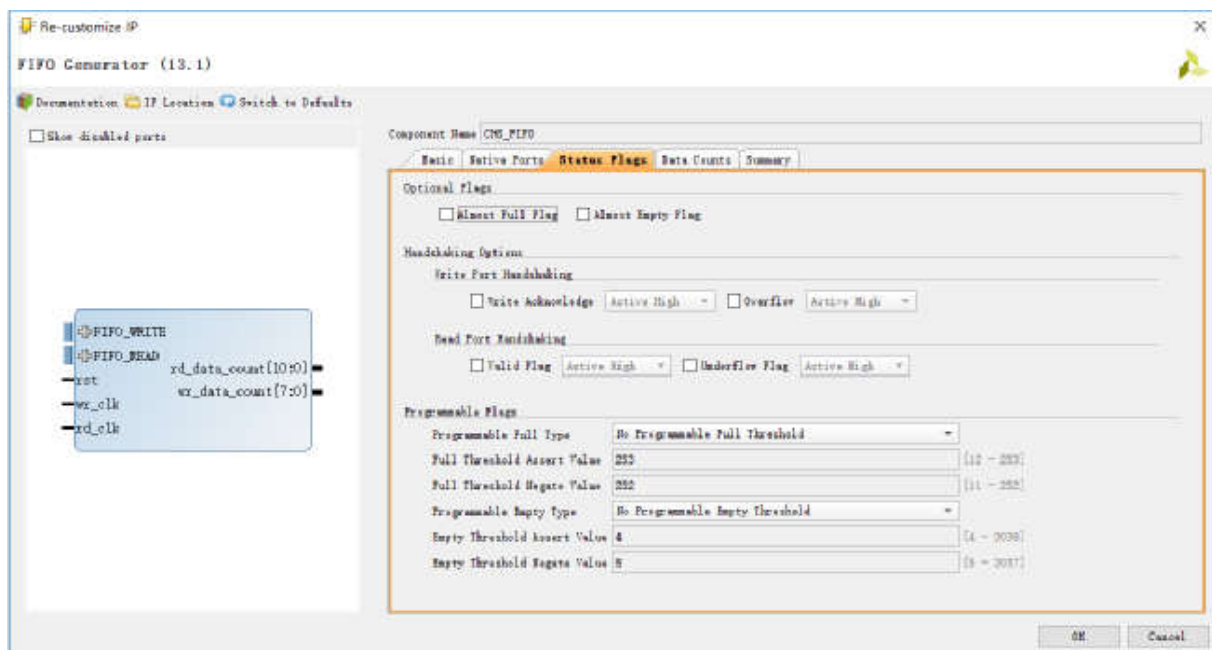
Step1:设置 Native 和 Independent Clocks Block RAM



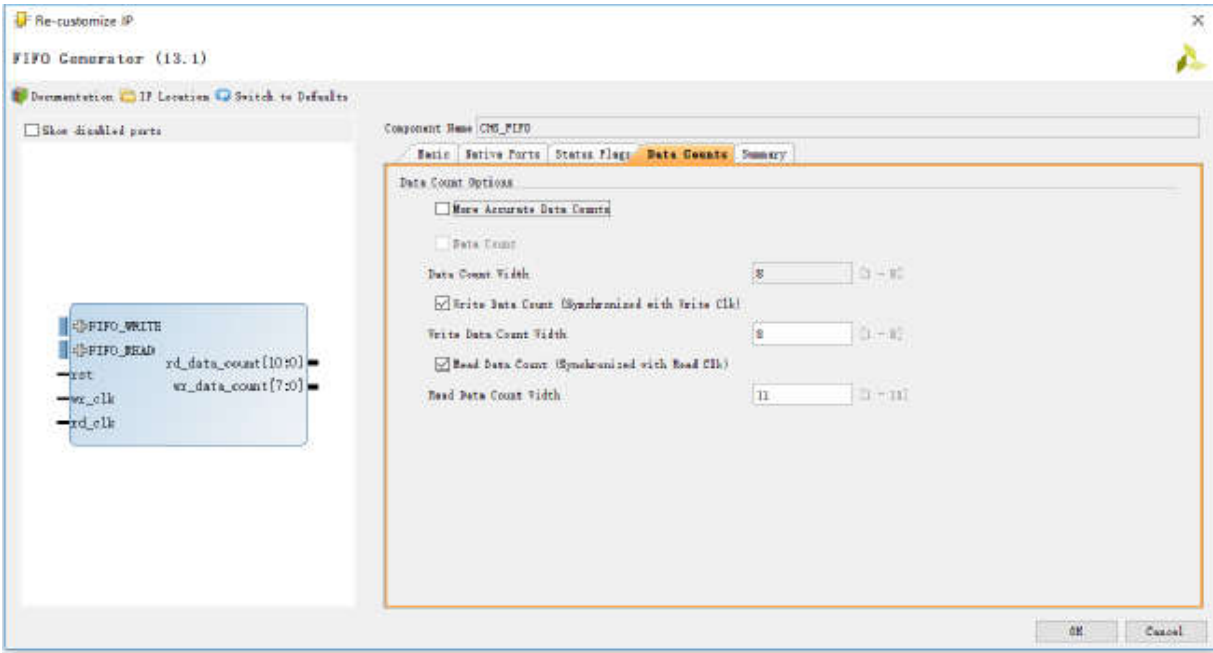
Step2:设置 FIFO 采用 First Word Fall Through 模式， FIFO 的写是 256bit 256 深度读 FIFO 是 32bit 2048 深度。



Step3:这一页默认不用设置



Step4:设置写 FIFO 的 counter 计数器宽度 8bit



3.2.2.6 MSG_FIFO 模块

《MIG_BURST_IMAGE.v》

MSG_FIFO 模块的作用是设计了一个消息的缓存， 所有读写请求都会先保存在这消息队列里面然后再按照先后顺序出列。 M_S 模块会对 MSG_FIFO 里面的消息进行处理。

MSG_FIFO_CLK： MSG_FIFO 的时钟信号 这里就是系统本模块的系统时钟和 MIG 控制器时钟一致。

MSG_FIFO_WRDATA： 长度为 8 代表同时可以一次性存储 8 个不同信号

MSG_FIFO_WREN： 写消息使能， 当读 image_data_gen 和 vga_lcd_driver 模块的场信号过来的时候会使能写 MSG_FIFO ， 把消息的状态进入 MSG_FIFO

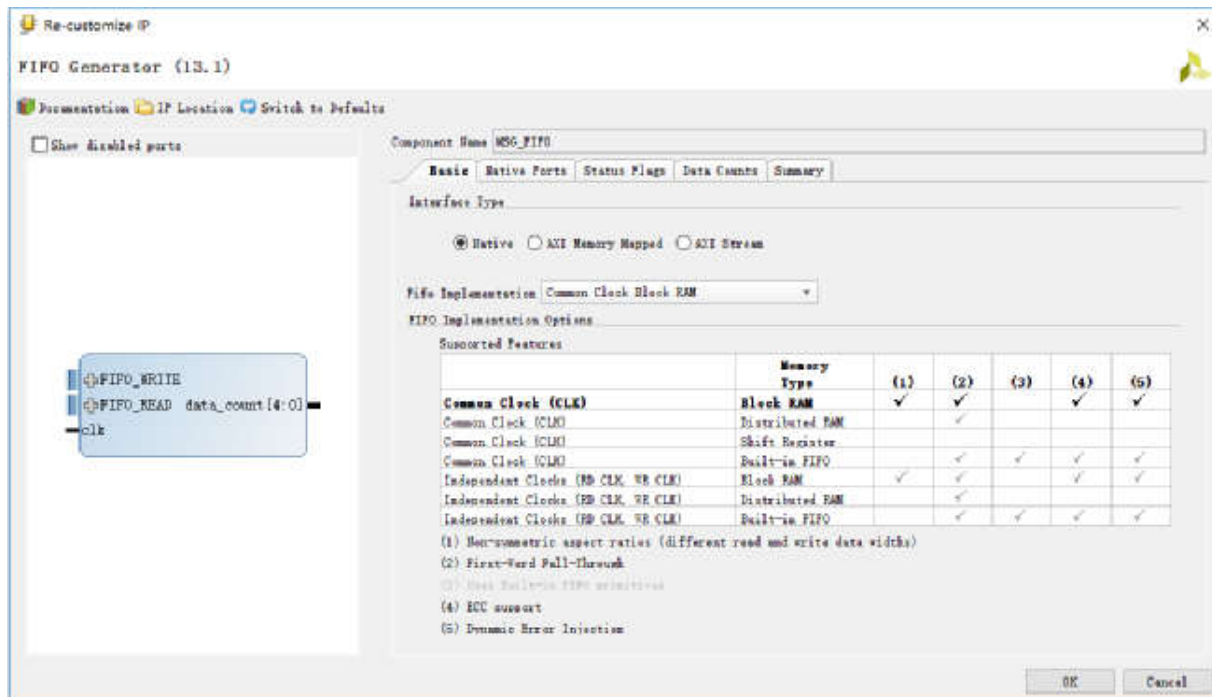
MSG_FIFO_RDDATA： 读消息使能， M_S 状态机会逐条取出 MSG_FIFO 队列中的消息。

MSG_FIFO_FULL： 消息满信号， 不应该让此信号满， 如果满必然发出消息溢出

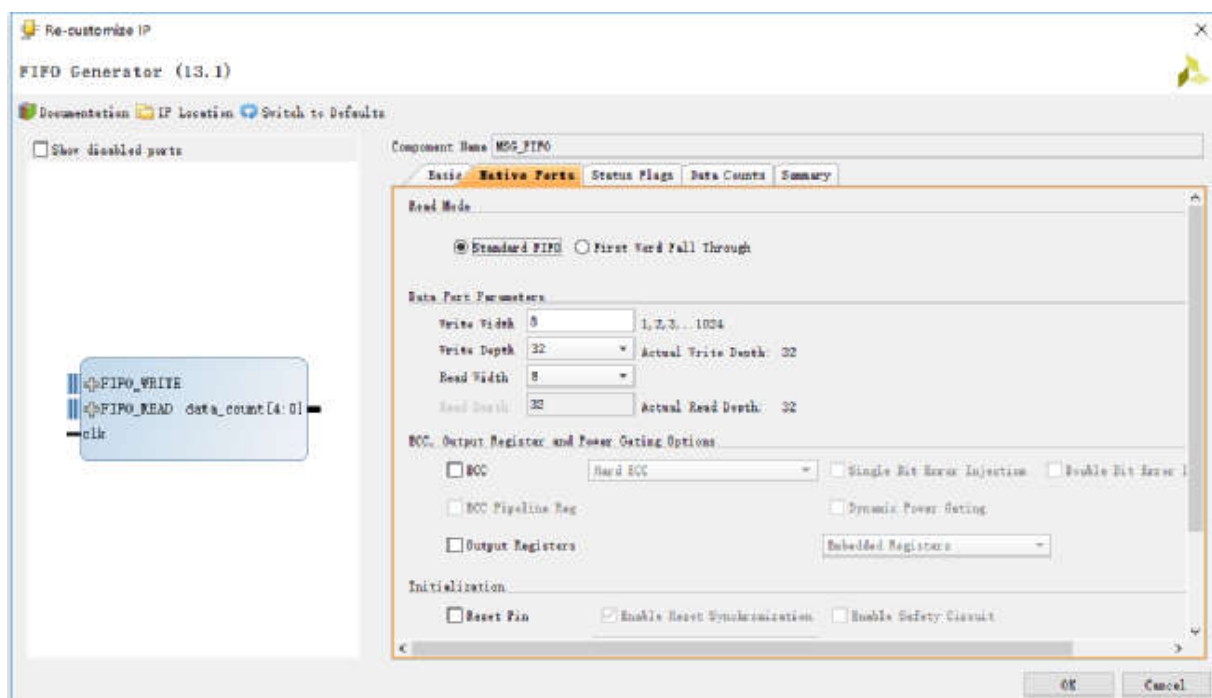
MSG_FIFO_USEDW： 记录目前已经写入的消息条数， 这里最多记录 32 条消息

CH6_FIFO 的配置界面：

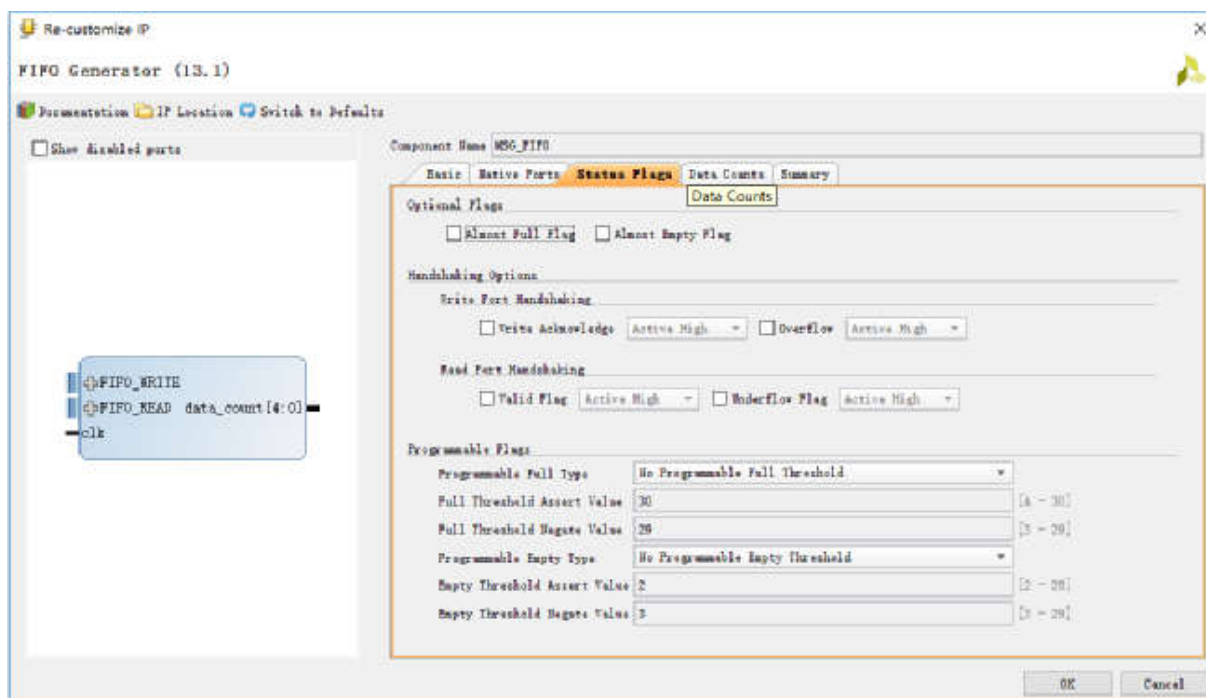
Step1:设置 Native 和 Common Clocks Block RAM



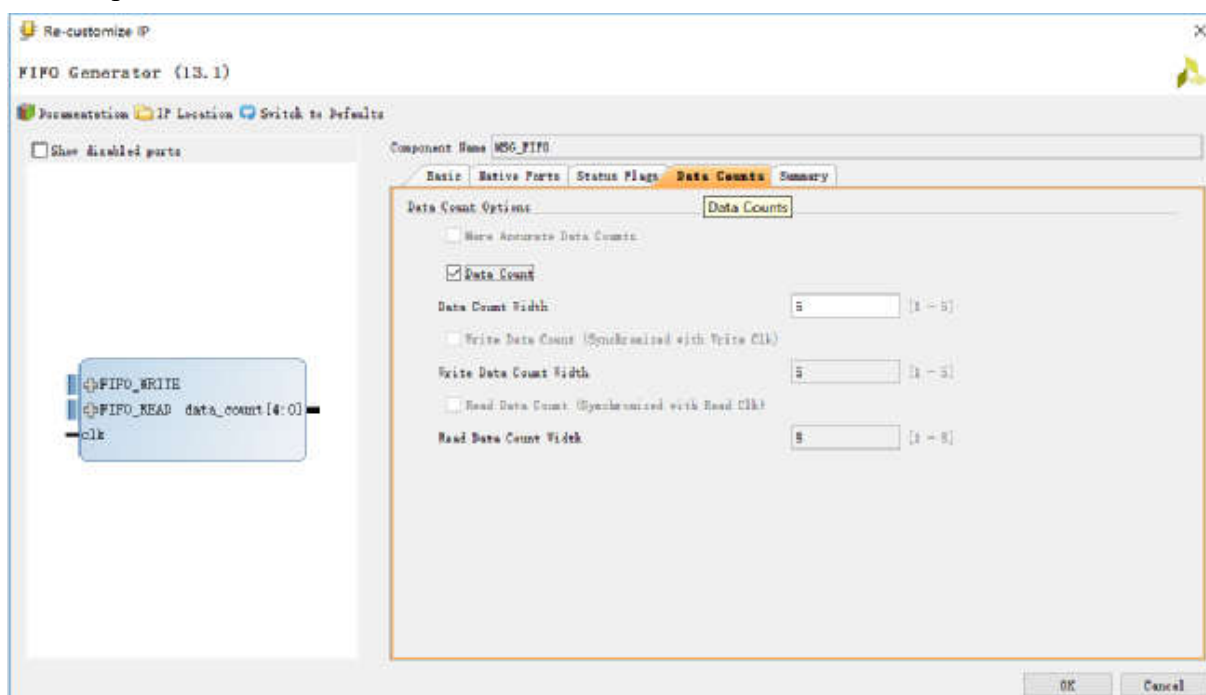
Step2:设置 FIFO 采用 standard 模式，FIFO 的写是 8bit 32 深度读 FIFO 是 8bit 32 深度。



Step3:这一页默认不用设置



Step4:设置写 FIFO 的 counter 计数器宽度 5bit



3.2.2.7 CH0_FIFO_REQ/CH6_FIFO_REQ 模块

《MIG_BURST_IMAGE.v》

CH0_REQ: 这个模块中, 信号在 CH0_FIFO 缓存一行数据后有效

CH6_REQ:这个模块中, 当 CH6_FIFO 的数据小于一行开始读 DDR 并且写入。

3.2.2.8 M_S 内存管理状态机

《MIG_BURST_IMAGE.v》

剩余所有的代码，具体参考注释。

3.2.2.9 图像帧三缓存的实现

首先是地址空间的分配的位置，如下面的程序。

```
1. parameter [ADDR_WIDTH-1'b0:0]CH0_RANGE =28'd921600;//写一副图像的地址空间
2. parameter [ADDR_WIDTH-1'b0:0]CH0_BASE_ADDR0 =28'd0;
3. parameter [ADDR_WIDTH-1'b0:0]CH0_BASE_ADDR1 =28'd921600;
4. parameter [ADDR_WIDTH-1'b0:0]CH0_BASE_ADDR2 =28'd921600*2;
5. parameter [ADDR_WIDTH-1'b0:0]CH6_RANGE =28'd921600;//读一副图像的地址空
6. parameter [ADDR_WIDTH-1'b0:0]CH6_BASE_ADDR0 =28'd0;
7. parameter [ADDR_WIDTH-1'b0:0]CH6_BASE_ADDR1 =28'd921600;
8. parameter [ADDR_WIDTH-1'b0:0]CH6_BASE_ADDR2 =28'd921600*2;
```

为了更好的实现三缓存设计， 本节设计了两个变量分别为 CH0_Frame_buf 和 CH6_Frame_buf。 原理就是让 CH6 通道确保可以输出 CH0 通道进来的最新数据。

```
1. M_S_MSG_FIFO2://--相对地址处理--//
2. begin
3.     M_S          <=M_S_RST_FIFO0;
4.     if(MSG_FIFO_RDDATA[7])begin
5.         if(CH0_Frame_buf==2'd2)
6.             CH0_Frame_buf=2'd0;
7.         else
8.             CH0_Frame_buf=CH0_Frame_buf+1'b1;
9.     end
10.
11.     if(MSG_FIFO_RDDATA[6])begin
12.         if(CH0_Frame_buf==2'd0)
13.             CH6_Frame_buf<=2'd2;
14.         else
15.             CH6_Frame_buf<=CH0_Frame_buf-1'b1;
16.     end
```

三缓存地址空间的切换

```
1. always@(*) begin
```



```

2.    case(CH0_Frame_buf)
3.        0:begin
4.            CH0_BASE_ADDR <= CH0_BASE_ADDR0;
5.        end
6.        1:begin
7.            CH0_BASE_ADDR <= CH0_BASE_ADDR1;
8.        end
9.        2:begin
10.           CH0_BASE_ADDR <= CH0_BASE_ADDR2;
11.        end
12.    default: begin
13.        CH0_BASE_ADDR <= CH0_BASE_ADDR0;
14.    end
15. endcase
16. end
17.
18. always@(*) begin
19.    case(CH6_Frame_buf)
20.        0:begin
21.            CH6_BASE_ADDR <= CH6_BASE_ADDR0;
22.        end
23.        1:begin
24.            CH6_BASE_ADDR <= CH6_BASE_ADDR1;
25.        end
26.        2:begin
27.            CH6_BASE_ADDR <= CH6_BASE_ADDR2;
28.        end
29.    default: begin
30.        CH6_BASE_ADDR <= CH6_BASE_ADDR0;
31.    end
32. endcase
33. end

```

MIG 地址空间的赋值

```
assign app_addr =(M_S==M_S_WR) ? (CH0_PTR+CH0_BASE_ADDR) : (CH6_PTR+CH6_BASE_ADDR);
```

3.2.2.10 HDMI 输出模块

《OV5640_TOP.v》

这部分，没什么难度，主要将《vga_lcd_driver.v》产生的时序及数据传输到 HDMI 的 IP 即可。

3.2.3 编译下载观察波形



图 3-13 摄像头安装

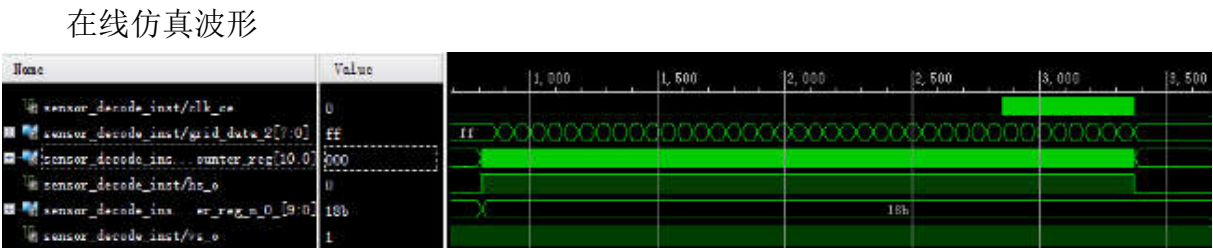


图 3-14 数据

这里笔者只给出 OV5640 部分添加测试代码和数据的部分。
测试输出结果

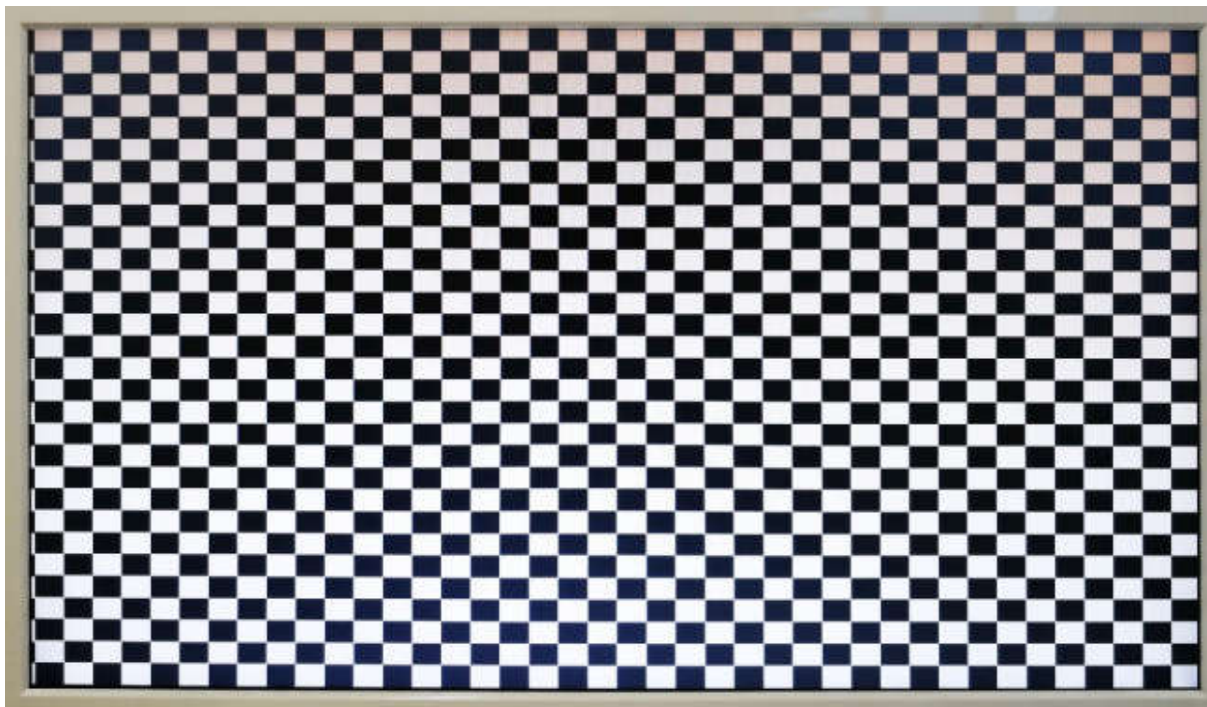


图 3-15 测试代码输出结果



图 3-16 摄像头输出结果

3.2.4 本章节文件夹内容

本章实验代码有 2 个如下图所示， 分别对应本课程中每个实验步骤

OV5640_TS_DATA: 增加测试数据代码的部分

OV5640_DEMO: 去掉测试数据采用真实摄像头数据的部分

DOC: 相关参考资料

 DOC	2019-12-14 23:45	文件夹
 OV5640_DEMO	2019-11-25 21:06	文件夹
 OV5640_TS_DATA	2019-11-25 21:06	文件夹

图 3-17 本章节文件夹内容

3.3 本章小结

。