

DDR3 读写测试及仿真实验

黑金动力社区 2020-03-13

1 实验简介

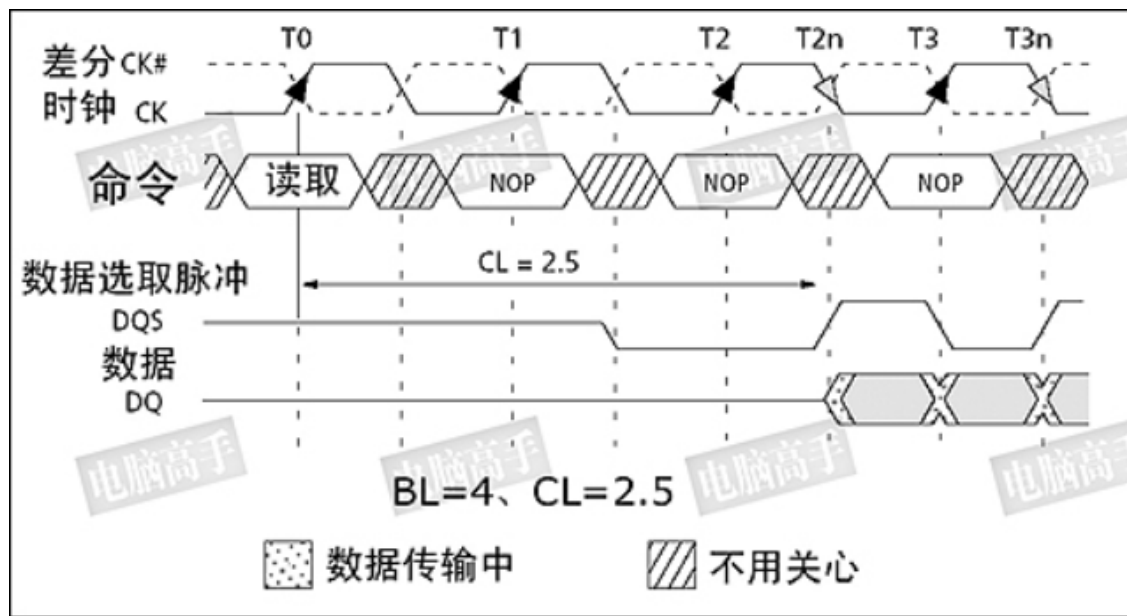
本实验为后续使用 DDR3 内存的实验做铺垫,通过循环读写 DDR3 内存,了解其工作原理和 DDR3 控制器的写法,由于 DDR3 控制复杂,控制器的编写难度高,这里笔者介绍 XILINX 的 MIG 控制器情况下应用,是后续音频、视频等需要用到 SDRAM 实验的基础。

2 实验原理

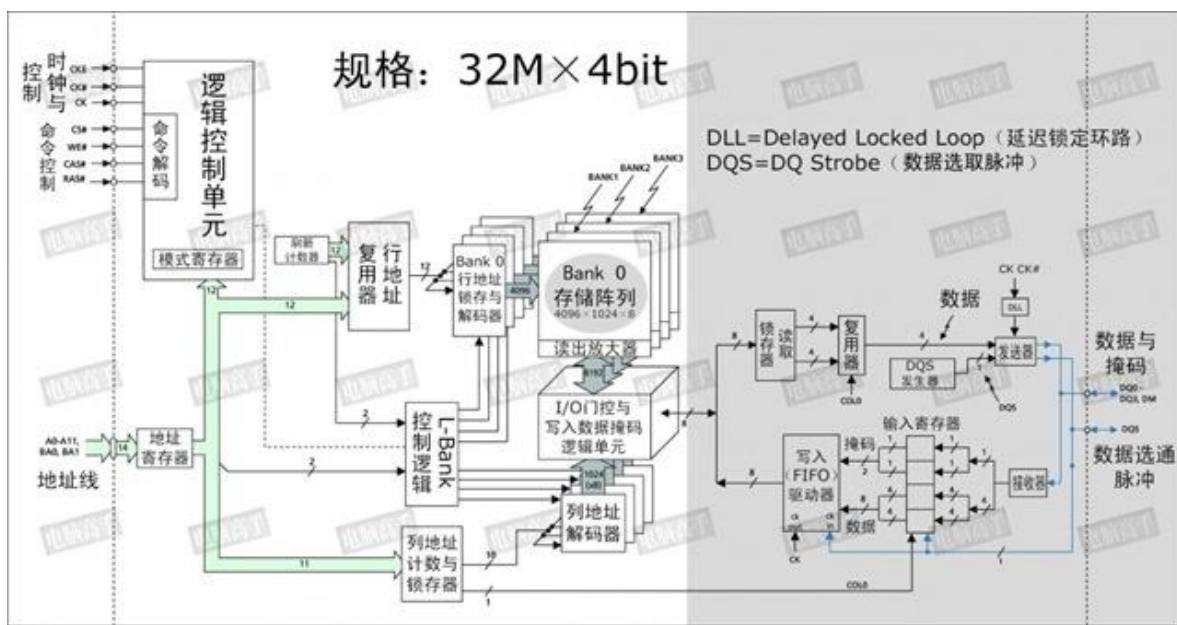
DDR SDRAM 全称为 Double Data Rate SDRAM,中文名为“双倍数据流 SDRAM”。DDR SDRAM 在原有的 SDRAM 的基础上改进而来。也正因为如此,DDR 能够凭借着转产成本优势来打败昔日的对手 RDRAM,成为当今的主流。本文只着重讲 DDR 的原理和 DDR SDRAM 相对于传统 SDRAM(又称 SDR SDRAM)的不同。

(一) DDR 的基本原理

有很多文章都在探讨 DDR 的原理,但似乎也不得要领,甚至还带出一些错误的观点。首先我们看看一张 DDR 正规的时序图。



从中可以发现它多了两个信号：CLK#与DQS，CLK#与正常CLK时钟相位相反，形成差分时钟信号。而数据的传输在CLK与CLK#的交叉点进行，可见在CLK的上升与下降沿（此时正好是CLK#的上升沿）都有数据被触发，从而实现DDR。在此，我们可以说通过差分信号达到了DDR的目的，甚至讲CLK#帮助了第二个数据的触发，但这只是对表面现象的简单描述，从严格的定义上讲并不能这么说。之所以能实现DDR，还要从其内部的改进说起。



DDR 内存芯片的内部结构图

这是一颗 128Mbit 的内存芯片，从图中可以看出来，白色区域内与 SDRAM 的结构基本相同，但请注意灰色区域，这是与 SDRAM 的不同之处。首先就是内部的 L-Bank 规格。SDRAM 中 L-Bank 存储单元的容量与芯片位宽相同，但在 DDR SDRAM 中并不是这样，存储单元的容量是芯片位宽的一倍，所以在此不能再套用讲解 SDRAM 时“芯片位宽=存储单元容量”的公式了。也因此，真正的行、列地址数量也与同规格 SDRAM 不一样了。

以本芯片为例，在读取时，L-Bank 在内部时钟信号的触发下一次传送 8bit 的数据给读取锁存器，再分成两路 4bit 数据传给复行地址，由后者将它们合并为一路 4bit 数据流，然后由发送器在 DQS 的控制下在外部时钟上升与下降沿分两次传输 4bit 的数据给北桥。这样，如果时钟频率为 100MHz，那么在 I/O 端口处，由于是上下沿触发，那么就是传输频率就是 200MHz。

现在大家基本明白 DDR SDRAM 的工作原理了吧，这种内部存储单元容量（也可以称为芯片内部总线位宽）=2×芯片位宽（也可称为芯片 I/O 总线位宽）的设计，就是所谓的两位预取（2-bit Prefetch），有的公司则贴切的称之为 2-n Prefetch（n 代表芯片位宽）。

(二) DDR SDRAM 与 SDRAM 的不同

DDR SDRAM 与 SDRAM 的不同主要体现在以下几个方面。

DDR SDRAM 与 SDRAM 的主要不同对比表

比较项 \ 内存类型	SDRAM	DDR SDRAM
命令		
全页式突发传输	支持	不支持
时钟信号挂起	支持	不支持
读出数据屏蔽	支持	不支持
写入数据屏蔽	支持	支持
功能		
时钟	单一时钟	差分时钟
预取设计	1-bit	2-bit
数据传输率	1/时钟周期	2/时钟周期
CAS 潜伏期	2、3	1.5、2、2.5、3
写入潜伏期	0	可变
突发长度	1、2、4、8、全页	2、4、8
延迟锁定回路	可选	工作时必需
自动刷新间隔周期	固定	弹性设计（最大值与 SDRAM 的固定值相同）
数据选取脉冲	无	必需
封装与电气特性		
封装类型 (≥64Mbit)	TSOP-II 54pin (400mil)	TSOP-II 66pin (400mil)
		CSP 60pin
工作电压	3.3V (LVTTTL 接口)	2.5V (SSTL_2 接口)
模组	168pin DIMM	184pin DIMM

注：LVTTTL=Low Voltage Transistor-Transistor Logic（低电压晶体管-晶体管逻辑电路）、SSTL=Stub Series Terminated Logic（短线串联终止逻辑电路）

提示：TSOP-II 与 CSP

TSOP-II：是指小外形薄型封装（Thin Small Outline Package）的第二种方式，引脚在封装的长边两侧，TSOP-I 的引脚则在短边的两侧。

CSP：是指芯片尺寸封装（Chip Scale Package），其封装尺寸与芯片核心尺寸基本相同，所以称 CSP，其内核面积与封装面积的比例约为 1:1.1，凡是符合这一标准的封装都可称之为 CSP。

DDR SDRAM 与 SDRAM 一样，在开机时也要进行 MRS，不过由于操作功能的增多，DDR SDRAM 在 MRS 之前还多了一 EMRS 阶段（Extended Mode Register Set，扩展模式寄存器设置），这个扩展模式寄存器控制着 DLL 的有效/禁止、输出驱动强度、QFC 有效/无效等。

提示与误区：QFC 的含义与作用

QFC 是指 FET Switch Controller（FET 开关控制），低电平有效。用于借助外部 FET（场效应管）开关控制模组上芯片间的相互隔离，没有读写操作时进入隔离状态，以确保芯片间不受相互干扰。QFC 是一个特选功能，厂商都是在接到芯片买家的指定要求后，才在芯片中加入这一功能，并且需要在模装配时进行相关的设计改动（如增加 VddQ 的上拉电阻），所以 DIY 市场上几乎很少见到支持这一功能的 DDR 内存。而在 2002 年 5 月，JEDEC 最新发布的 DDR 规范中，已经不存在 QFC 的定义，而且即使有 FET 开关也将由北桥控制（此时是用来隔离模组的），因此 QFC 已经成为历史。可是，在很多“深入”性介绍中，都将 QFC 认为是一个必要的过程，这显然是错的。

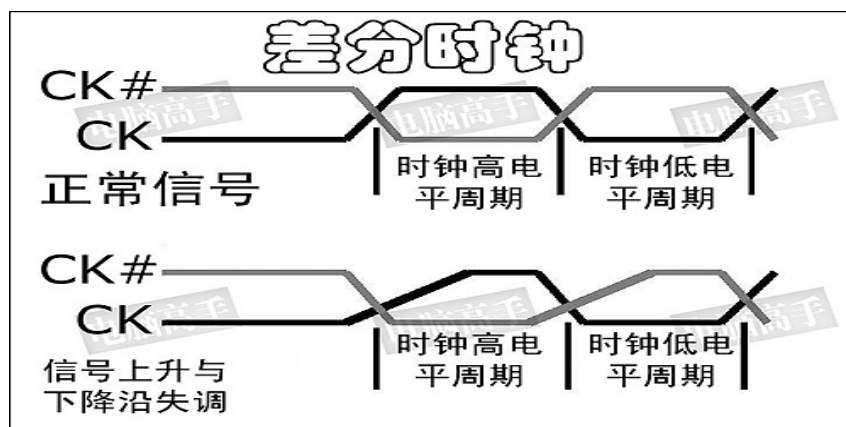
由于 EMRS 与 MRS 的操作方法与 SDRAM 的 MRS 大同小异，在此就不再列出具体的模式表了，有兴趣的话可查看相关的 DDR 内存资料。下面我们就着重说说 DDR SDRAM 的新设计与新功能。

误区：CSP 与 uBGA、WBGA、TinyBGA、FBGA 等是不同的封装技术

实际上，CSP 只是一种封装标准/类型，不涉及具体的封装技术，只要达到它的尺寸标准都可称之为 CSP 封装。近几年出现的 uBGA、WBGA、TinyBGA、FBGA 小型芯片封装技术则是 CSP 的具体表现形式（其实都是 BGA 封装技术的一种），由此可以看出 CSP 并没有固定的封装技术，它自己更不是一个封装技术，只要厂商愿意或有实力，可以开发出更多的符合 CSP 标准的封装技术。

1、差分时钟

差分时钟（参见上文“DDR SDRAM 读操作时序图”）是 DDR 的一个必要设计，但 CK# 的作用，并不能理解为第二个触发时钟（你可以在讲述 DDR 原理时简单地这么比喻），而是起到触发时钟校准的作用。由于数据是在 CK 的上下沿触发，造成传输周期缩短了一半，因此必须要保证传输周期的稳定以确保数据的正确传输，这就要求 CK 的上下沿间距要有精确的控制。但因为温度、电阻性能的改变等原因，CK 上下沿间距可能发生变化，此时与其反相的 CK# 就起到纠正的作用（CK 上升快下降慢，CK# 则是上升慢下降快）。而由于上下沿触发的原因，也使 CL=1.5 和 2.5 成为可能，并容易实现。与 CK 反相的 CK# 保证了触发时机的准确性。



2、数据选取脉冲 (DQS)

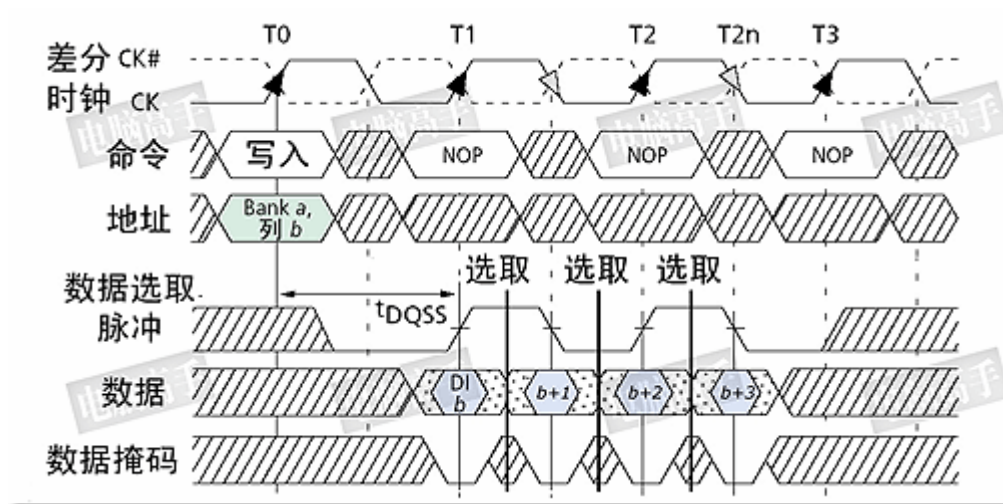
总结 DQS：它是双向信号；读内存时，由内存产生，DQS 的沿和数据的沿对齐；写入内存时，由外部产生，DQS 的中间对应数据的沿，即此时 DQS 的沿对应数据最稳定的中间时刻。

DQS 是 DDR SDRAM 中的重要功能，它的功能主要用来在一个时钟周期内准确的区分出每个传输周期，并便于接收方准确接收数据。每一颗芯片都有一个 DQS 信号线，它是双向的，在写入时它用来传送由北桥发来的 DQS 信号，读取时，则由芯片生成 DQS 向北桥发送。完全可以说，它就是数据的同步信号。

在读取时，DQS 与数据信号同时生成（也是在 CK 与 CK#的交叉点）。而 DDR 内存中的 CL 也就是从 CAS 发出到 DQS 生成的间隔，数据真正出现在数据 I/O 总线上相对于 DQS 触发的时间间隔被称为 tAC。注意，这与 SDRAM 中的 tAC 的不同。实际上，DQS 生成时，芯片内部的预取已经完毕了，tAC 是指上文结构图中灰色部分的数据输出时间，由于预取的原因，实际的数据传出可能会提前于 DQS 发生（数据提前于 DQS 传出）。由于是并行传输，DDR 内存对 tAC 也有一定的要求，对于 DDR266，tAC 的允许范围是 $\pm 0.75\text{ns}$ ，对于 DDR333，则是 $\pm 0.7\text{ns}$ ，有关它们的时序图示见前文，其中 CL 里包含了一段 DQS 的导入期。

前文已经说了 DQS 是为了保证接收方的选择数据，DQS 在读取时与数据同步传输，那么接收时也是以 DQS 的上下沿为准吗？不，如果以 DQS 的上下沿区分数据周期的危险很大。由于芯片有预取的操作，所以输出时的同步很难控制，只能限制在一定的时间范围内，数据在各 I/O 端口的出现时间可能有快有慢，会与 DQS 有一定的间隔，这也就是为什么要有一个 tAC 规定的原因。而在接收方，一切必须保证同步接收，不能有 tAC 之类的偏差。这样在写入时，芯片不再自己生成 DQS，而以发送方传来的 DQS 为基准，并相应延后一定的时间，在 DQS 的中部为数据周期的选取分割点（在读取时分割点就是上下沿），从这里分隔开两个传输周期。这样做的好处是，由于各

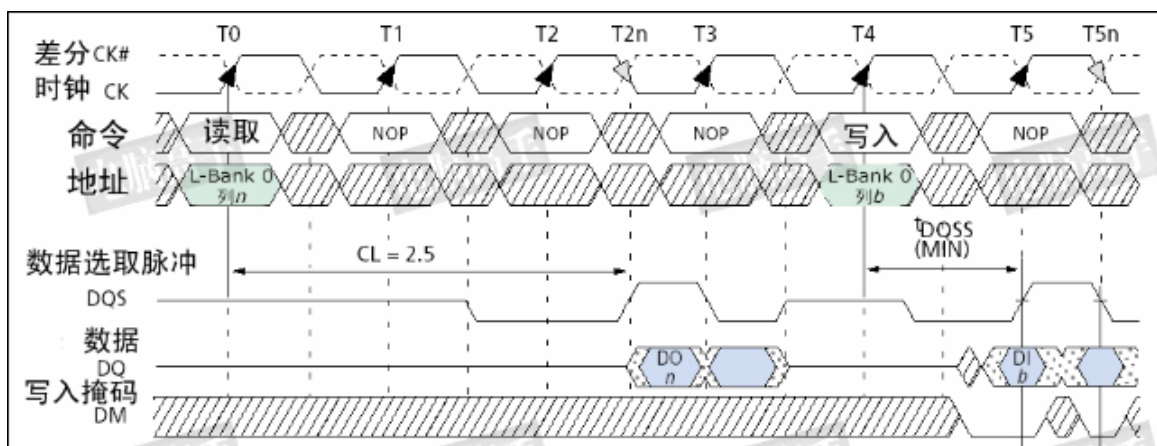
数据信号都会有一个逻辑电平保持周期，即使发送时不同步，在 DQS 上下沿时都处于保持周期中，此时数据接收触发的准确性无疑是最高的。在写入时，以 DQS 的高/低电平期中部为数据周期分割点，而不是上/下沿，但数据的接收触发仍为 DQS 的上/下沿。



3、写入延迟

在上面的 DQS 写入时序图中，可以发现写入延迟已经不是 0 了，在发出写入命令后，DQS 与写入数据要等一段时间才会送达。这个周期被称为 DQS 相对于写入命令的延迟时间（ t_{DQSS} ，WRITE Command to the first corresponding rising edge of DQS），对于这个时间大家应该很好理解了。

为什么要有这样的延迟设计呢？原因也在于同步，毕竟一个时钟周期两次传送，需要很高的控制精度，它必须要等接收方做好充分的准备才行。 t_{DQSS} 是 DDR 内存写入操作的一个重要参数，太短的话恐怕接受有误，太长则会造成总线空闲。 t_{DQSS} 最短不能小于 0.75 个时钟周期，最长不能超过 1.25 个时钟周期。有人可能会说，如果这样，DQS 不就与芯片内的时钟不同步了吗？对，正常情况下， t_{DQSS} 是一个时钟周期，但写入时接受方的时钟只用来控制命令信号的同步，而数据的接受则完全依靠 DQS 进行同步，所以 DQS 与时钟不同步也无所谓。不过， t_{DQSS} 产生了一个不利影响——读后写操作延迟的增加，如果 $CL=2.5$ ，还要在 t_{DQSS} 基础上加入半个时钟周期，因为命令都要在 CK 的上升沿发出。



当 $CL=2.5$ 时，读后写的延迟将为 $t_{DQSS}+0.5$ 个时钟周期（图中 $BL=2$ ）

另外，DDR 内存的数据真正写入由于要经过更多步骤的处理，所以写回时间（ t_{WR} ）也明显延长，一般在 3 个时钟周期左右，而在 DDR-II 规范中更是将 t_{WR} 列为模式寄存器的一项，可见它的重要性。

误区：DDR SDRAM 各种延迟与潜伏期的单位时间减半

一些文章认为，DDR 使数据传输率加倍，那么与之相关的延迟与潜伏期的单位时间也减半，比如 DDR-266 内存， t_{RCD} 、 CL 、 t_{RP} 的单位周期为 3.75ns，比 PC133 内存少了一半。这是严重的概念性错误，从我们列举的时序图中可以看出， t_{RCD} 、 CL 、 t_{RP} 是以时钟信号来界定的，不能用传输周期去表示，否则 $CL=2.5$ 的参考基准是什么？对于 DDR-266，时钟频率是 133MHz，时钟周期仍是 7.5，和 PC133 的标准一样。那些文章的作者显然是将时钟周期与传输周期弄混了

4、突发长度与写入掩码

在 DDR SDRAM 中，突发长度只有 2、4、8 三种选择，没有了随机存取的操作（突发长度为 1）和全页式突发。这是为什么呢？因为 L-Bank 一次就存取两倍于芯片位宽的数据，所以芯片至少也要进行两次传输才可以，否则内部多出来的数据怎么处理？而全页式突发事实证明在 PC 内存中是很难用得上的，所以被取消也不稀奇。

但是，突发长度的定义也与 SDRAM 的不一样了（见本章节最前那幅 DDR 简示图），它不再指所连续寻址的存储单元数量，而是指连续的传输周期数，每次是一个芯片位宽的数据。对于突发写入，如果其中有不存入的数据，仍可以运用 DM 信号进行屏蔽。DM 信号和数据信号同时发出，接收方在 DQS 的上升与下降沿来判断 DM 的状态，如果 DM 为高电平，那么之前从 DQS 中部选取的数据就被屏蔽了。有人可能会觉得，DM 是输入信号，意味着芯片不能发出 DM 信号给

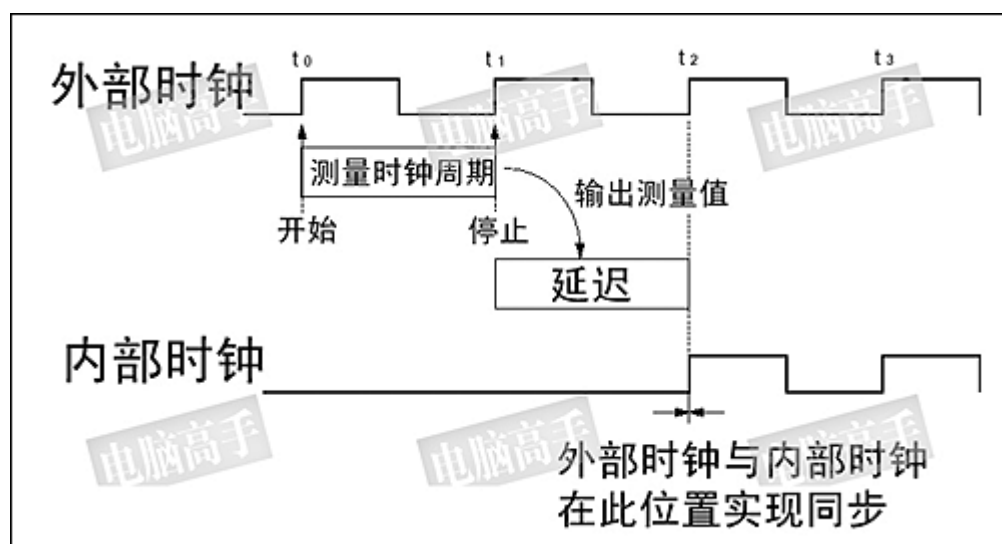
北桥作为屏蔽读取数据的参考。其实，该读哪个数据也是由北桥芯片决定的，所以芯片也无需参与北桥的工作，哪个数据是有用的就留给北桥自己去选吧。

5、延迟锁定回路 (DLL)

DDR SDRAM 对时钟的精确性有着很高的要求，而 DDR SDRAM 有两个时钟，一个是外部的总线时钟，一个是内部的工作时钟，在理论上 DDR SDRAM 这两个时钟应该是同步的，但由于种种原因，如温度、电压波动而产生延迟使两者很难同步，更何况时钟频率本身也有不稳定的情况（SDRAM 也内部时钟，不过因为它的工作/传输频率较低，所以内外同步问题并不突出）。DDR SDRAM 的 tAC 就是因为内部时钟与外部时钟有偏差而引起的，它很可能造成因数据不同步而产生错误的恶果。实际上，不同步就是一种正/负延迟，如果延迟不可避免，那么若是设定一个延迟值，如一个时钟周期，那么内外时钟的上升与下降沿还是同步的。鉴于外部时钟周期也不会绝对统一，所以需要根据外部时钟动态修正内部时钟的延迟来实现与外部时钟的同步，这就是 DLL 的任务。

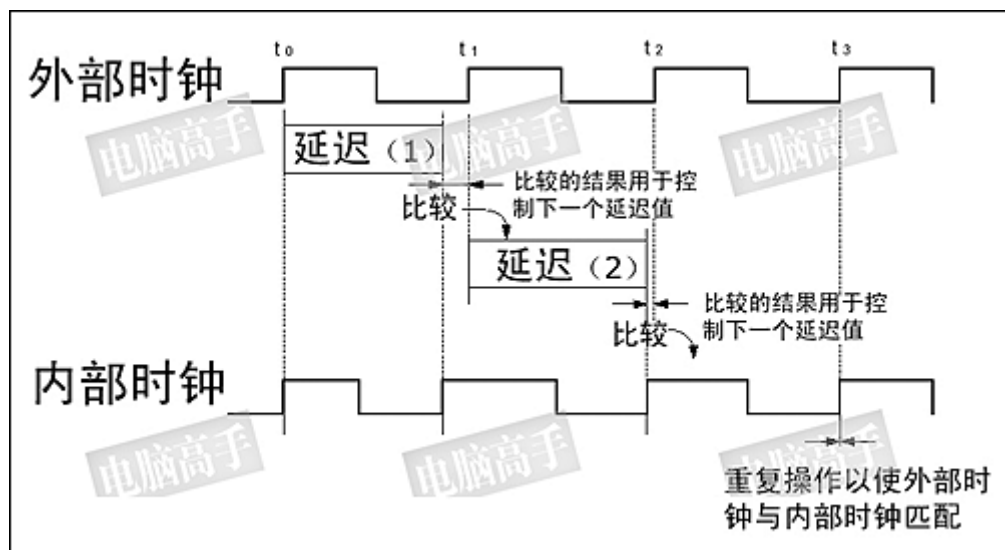
DLL 不同于主板上的 PLL，它不涉及频率与电压转换，而是生成一个延迟量给内部时钟。目前 DLL 有两种实现方法，一个是时钟频率测量法（CFM，Clock Frequency Measurement），一个是时钟比较法（CC，Clock Comparator）。

CFM 是测量外部时钟的频率周期，然后以此周期为延迟值控制内部时钟，这样内外时钟正好就相差了一个时钟周期，从而实现同步。DLL 就这样反复测量反复控制延迟值，使内部时钟与外部时钟保持同步。



CFM 式 DLL 工作示意图

CC 的方法则是比较内外部时钟的长短，如果内部时钟周期短了，就将所少的延迟加到下一个内部时钟周期里，然后再与外部时钟做比较，若是内部时钟周期长了，就将多出的延迟从下一个内部时钟中刨除，如此往复，最终使内外时钟同步。



CC 式 DLL 工作示意图

CFM 与 CC 各有优缺点，CFM 的校正速度快，仅用两个时钟周期，但容易受到噪音干扰，并且如果测量失误，则内部的延迟就永远错下去了。CC 的优点则是更稳定可靠，如果比较失败，延迟受影响的只是一个数据（而且不会太严重），不会涉及到后面的延迟修正，但它的修正时间要比 CFM 长。DLL 功能在 DDR SDRAM 中可以被禁止，但仅限于除错与评估操作，正常工作状态是自动有效的。

误区：DLL 是实现 DDR 传输的关键

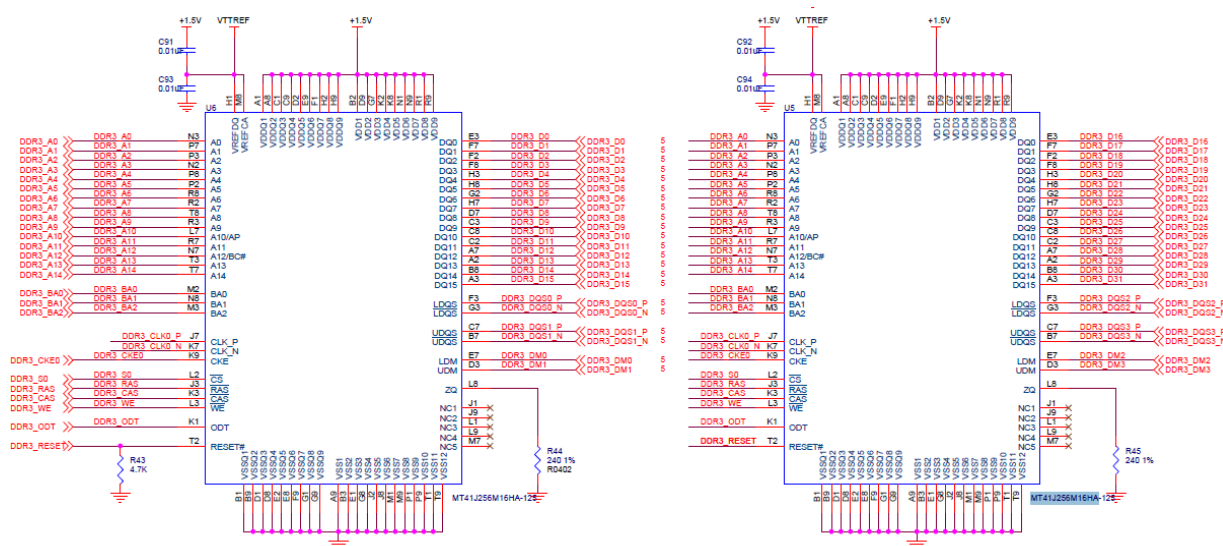
“DDR 内存通过内部的 DLL 延时锁相环提供精确的时钟定位，这样就可以在每个时钟周期的上升沿和下降沿传输数据”。“DDR 使用了 DLL 来提供一个数据滤波信号 DQS 来选取数据”。

以上是当前较为流行的对 DDR 工作原理的两种解释，现在大家能看出错误所在吗？两者都把 DLL 的功能夸大了，DLL 只是一个重要的辅助校准设计，与能否实现双沿触发没有关系，它只是保证数据的输出尽量与外部时钟同步。后者则是概念性错误，从 DDR 内存结构图可看出，DQS 不是 DLL 生成的，只是由 DLL 保证其与外部时钟的同步，DQS 由单独的 DQS 发生器生成。

3 硬件介绍

AX7101/AX7102/AX7103 开发板上使用了 2 个 Micron DDR3 的颗粒 MT41J256M16HA,每个 DDR 芯片的容量为 4Gb。两个 DDR3 芯片组合成 32 位的数据总线宽度和 FPGA 相连接。开发板板上对 DDR3 的地址线和控制线都做了端接电阻上拉到 VTT 电压, 保证信号的质量。在 PCB 的设计上, 完全遵照 XILINX 的 DDR3 设计规范, 严格保证等长设计和阻抗控制。在进行 DDR3 与 Artix-7 设计时, FPGA 的 DDR 管脚分配是要有所考虑的, 而不能随意分配。如果用户自己实在不清楚怎么连接, 那就请完全参考我们的原理图来设计, 依样画葫芦总会的吧?

在 PCB 的设计上, 考虑高速信号的数据传输的可靠性, 走线上严格保证等长设计和阻抗控制。开发板 DDR 部分的原理图如下:

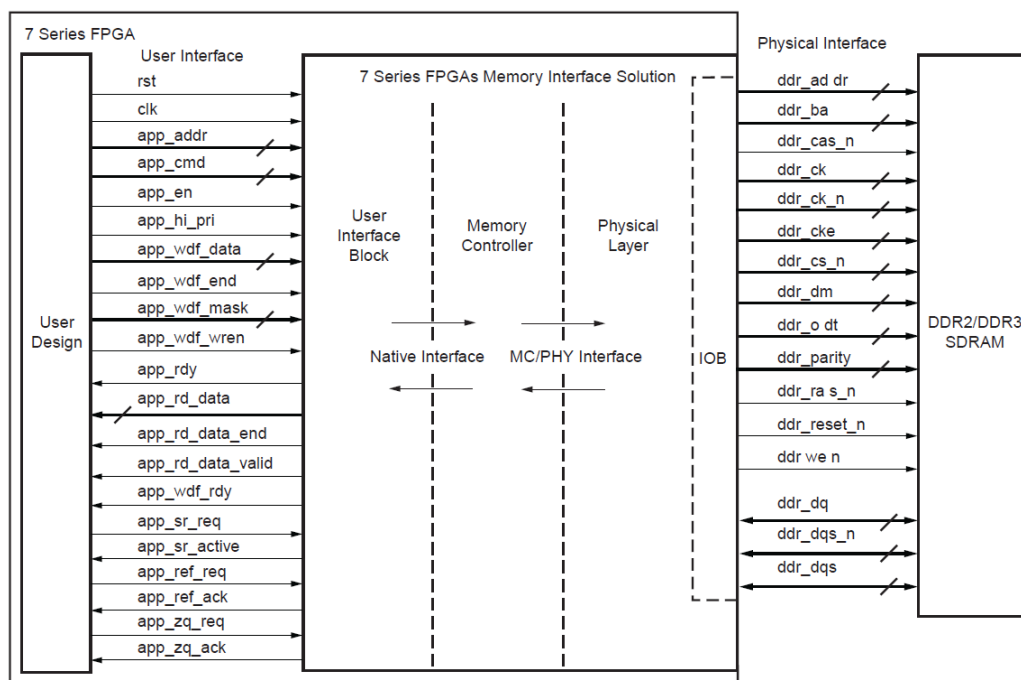


AX7101/AX7102/AX7103 开发板 DDR3

4 程序设计

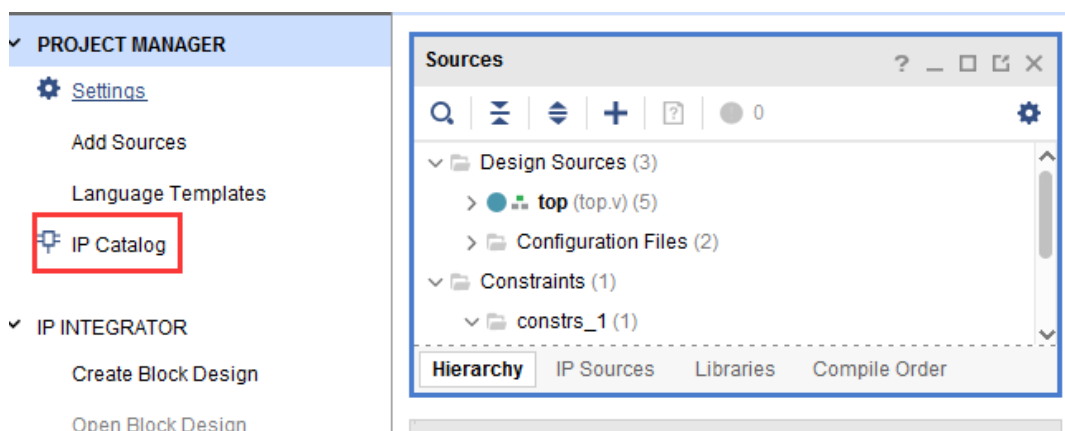
4.1 MIG 控制器设计

MIG IP 控制器是 Xilinx 为用户提供的—个 DDR 控制的 IP, 这样用户即使不了解 DDR 的控制和读写时序也能通过 DDR 控制器方便的读写 DDR 存储器。7 系列的 DDR 控制器的解决方案如下所示:

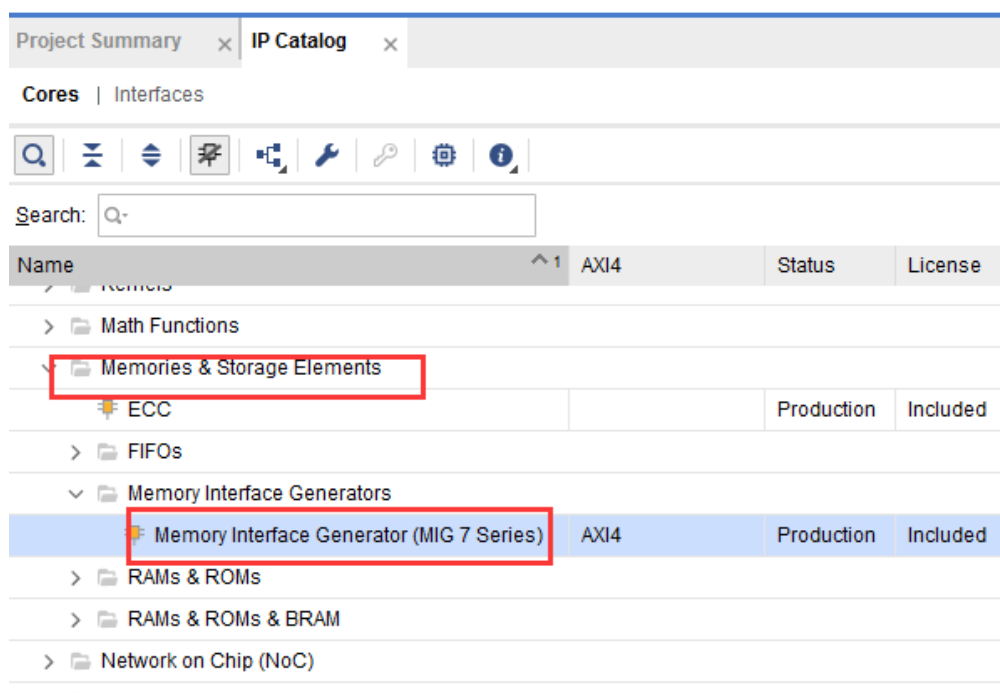


DDR3 控制器包含 3 部分:用户接口模块(User interface Block), 存储器控制模块(Memory Controller)和 DDR3 的物理接口(Physical Layer)。开发人员只需要开发用户的逻辑设计跟 DDR 控制器的用户接口对接来读写 DDR3 的数据。关于 DDR3 控制器用户端的接口定义和时序的更多介绍,大家还是参考 Xilinx 提供的文档 (UG586), 接下来为大家介绍如何生成和配置 DDR3 控制器吧!

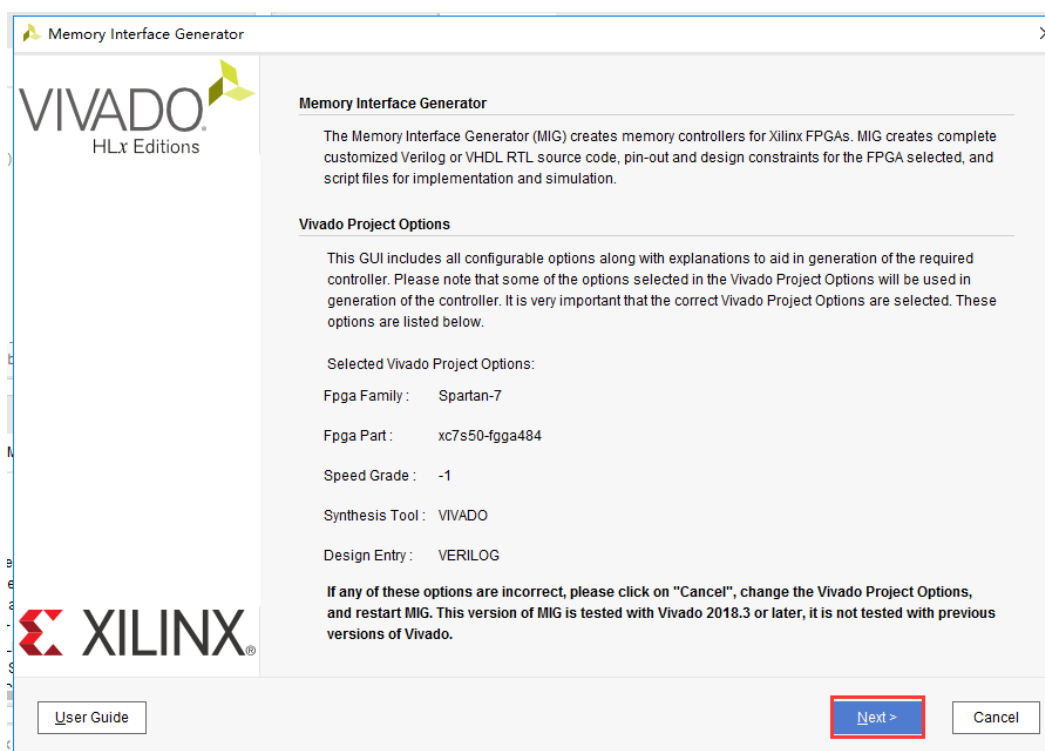
1. 首先在 Vivado 环境里新建一个项目, 取名为 ddr3_test。再点击 Project Manager 界面下的 IP Catalog, 打开 IP Catalog 界面。



2. 在 IP Catalog 界面里双击 Memories & Storage Elements\Memory Interface Generators 下的 Memory Interface Generator (MIG 7 Series)。



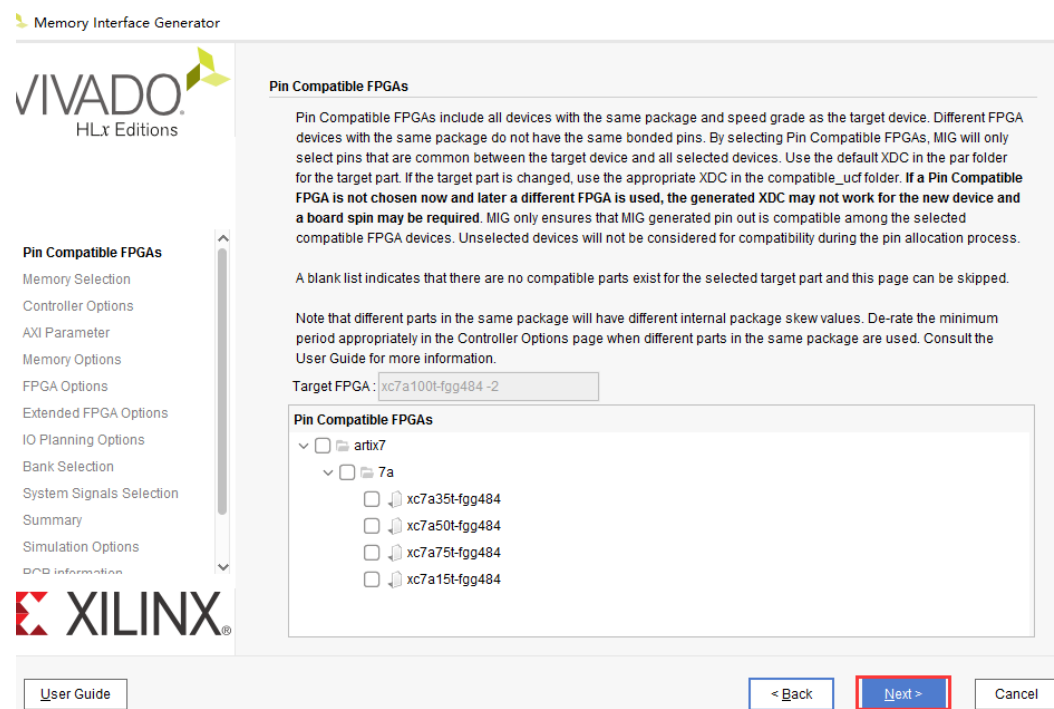
3. 点击 Next, 如果有同学想了解更多的 MIG 的信息, 可以点击左边的 User Guide 按钮来打开 Xilinx 的相关文档来查看。



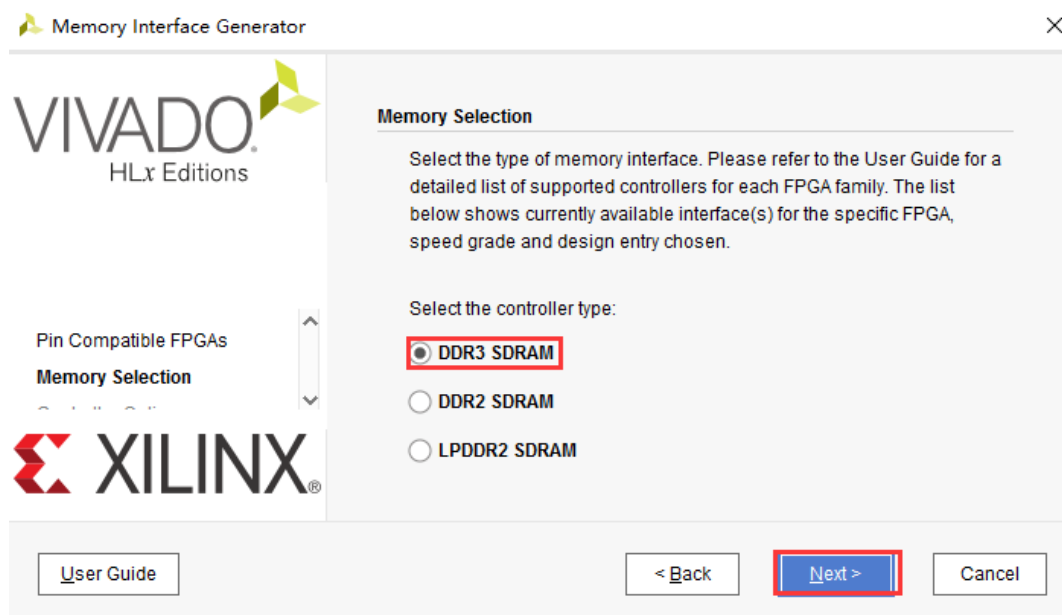
4. 修改 Component Name 为 "ddr3", 点击 Next。



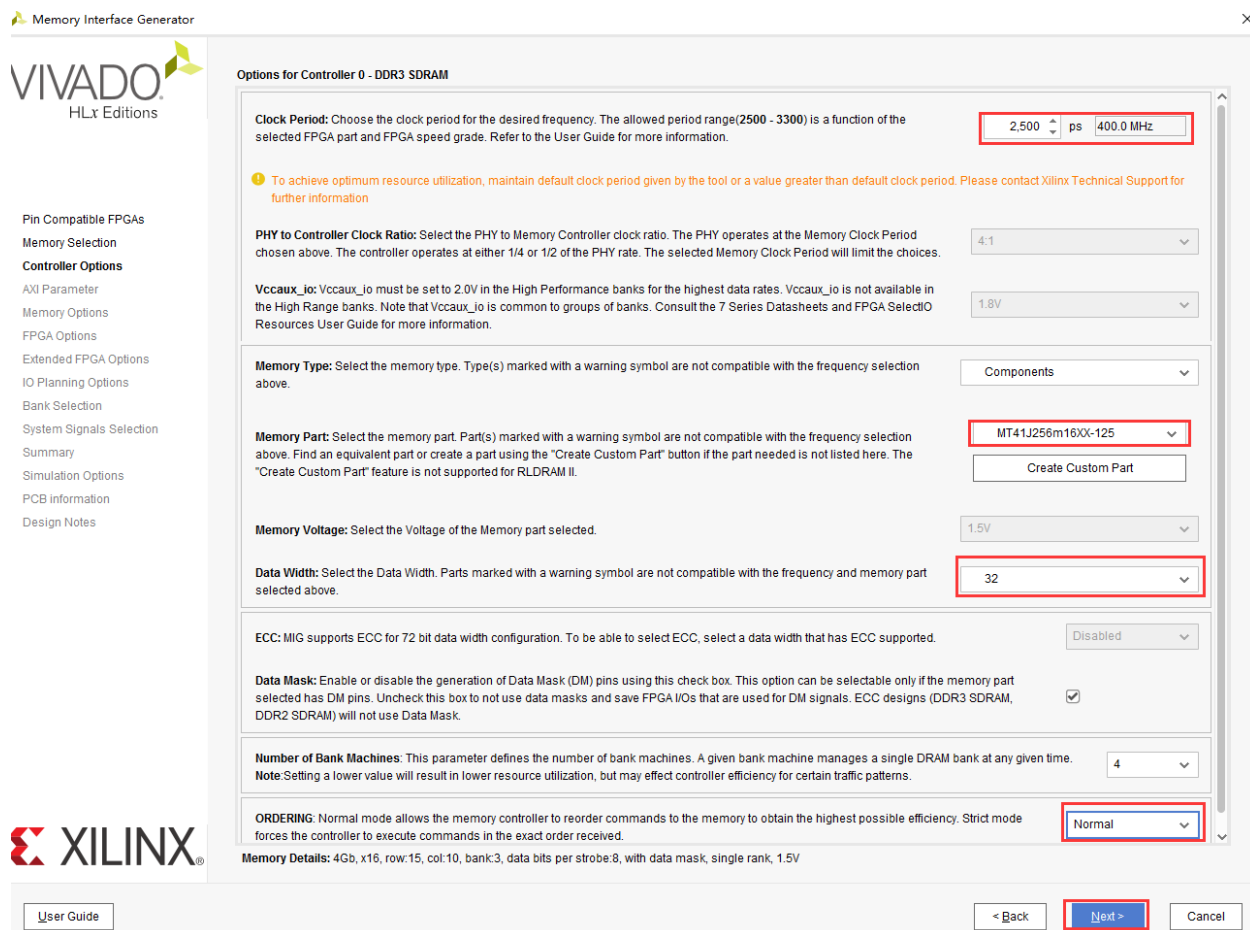
5. 这里可以选择兼容芯片，我们不需要，点 Next。



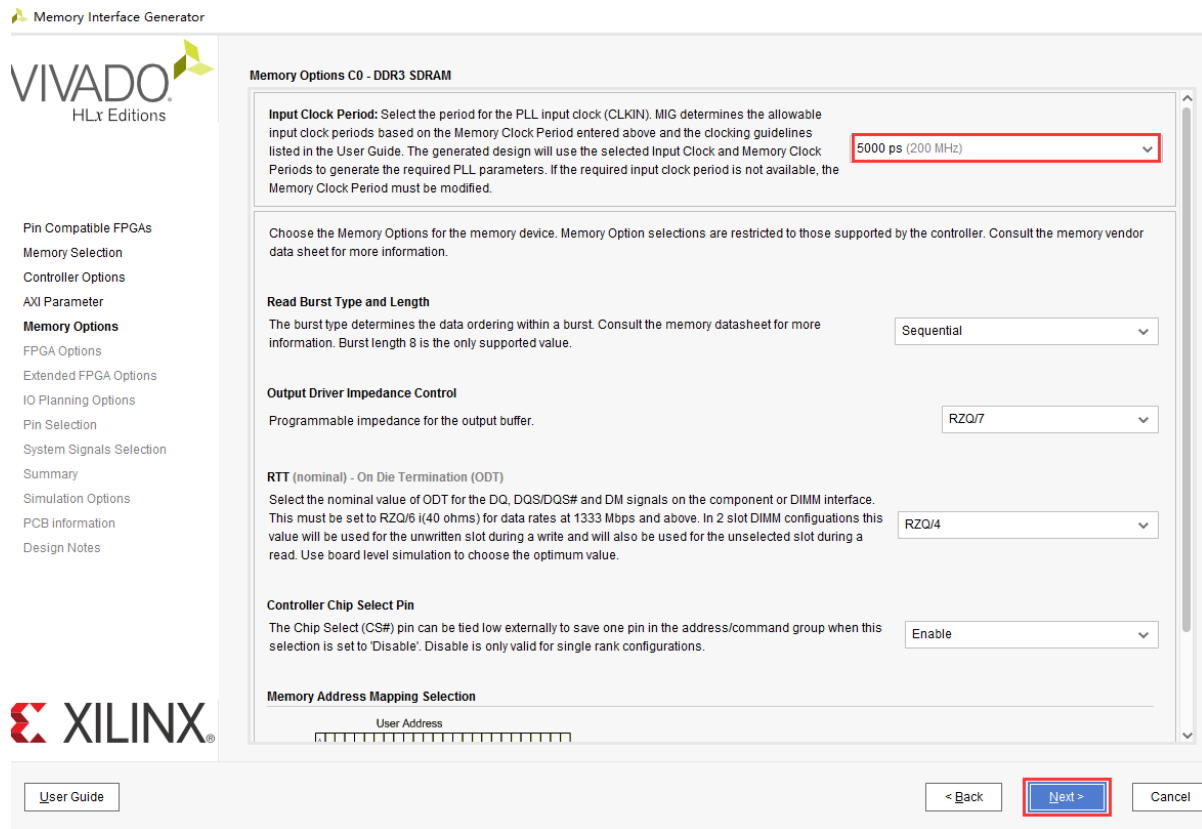
6. 选择默认的 DDR3 SDRAM。



7. Memory Part 选择开发板上的型号"MT41J256m16xx-125", Data Width 数据宽度选择 32 位。



8. 选择 PLL 输入时钟的频率为 200MHz, 这个时钟需要跟开发板上的时钟频率一致, 其它设置输出阻抗值和内部的 ODT 内部上拉电阻值来改善 DDR3 的信号完整性, 一般不需要修改。



9. System Clock 选择差分"No Buffer", Reference Clock 因为开发板上没有提供单独的 DDR 参考时钟, 所以选择"Use System Clock". System Reset Polarity 选择"ACTIVE LOW", 其它保留默认配置。

Memory Interface Generator

VIVADO[®]
HLx Editions

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

System Clock
Choose the desired input clock configuration. Design clock can be Differential or Single-Ended.
System Clock: No Buffer

Reference Clock
Choose the desired reference clock configuration. Reference clock can be Differential or Single-Ended.
Reference Clock: Use System Clock

System Reset Polarity
Choose the desired System Reset Polarity.
System Reset Polarity: ACTIVE LOW

Debug Signals Control
This feature allows various debug signals present in the IP to be monitored on the ChipScope tool. The debug signals include status signals of various PHY calibration stages. Enabling this feature will connect all the debug signals to the ChipScope ILA and VIO cores in the example design top module. A part of each bus in the debug interface has been grounded so that users can replace the grounded signals with the required signals.
Debug Signals for Memory Controller: OFF

Sample Data Depth
This selects the value of Sample Data depth for Chipscope ILA used in Debug logic.
Sample Data Depth: 1024

Internal Vref
Internal Vref can be used to allow the use of the Vref pins as normal IO pins. This option can only be used at 800 Mbps and lower data rates. This can free 2 pins per bank where inputs are used. This setting has no effect on banks with only outputs.
Internal Vref: ☐

IO Power Reduction
Significantly reduces average IO power by automatically disabling DQ/DQS IBUFs and internal terminations during WRITES and periods of inactivity
IO Power Reduction: ON

XADC Instantiation

User Guide < Back Next > Cancel

10. HR bank 的内部端接阻抗，这里为 50 ohms，不用修改。

Memory Interface Generator

VIVADO[®]
HLx Editions

Pin Compatible FPGAs
Memory Selection
Controller Options
AXI Parameter
Memory Options
FPGA Options
Extended FPGA Options

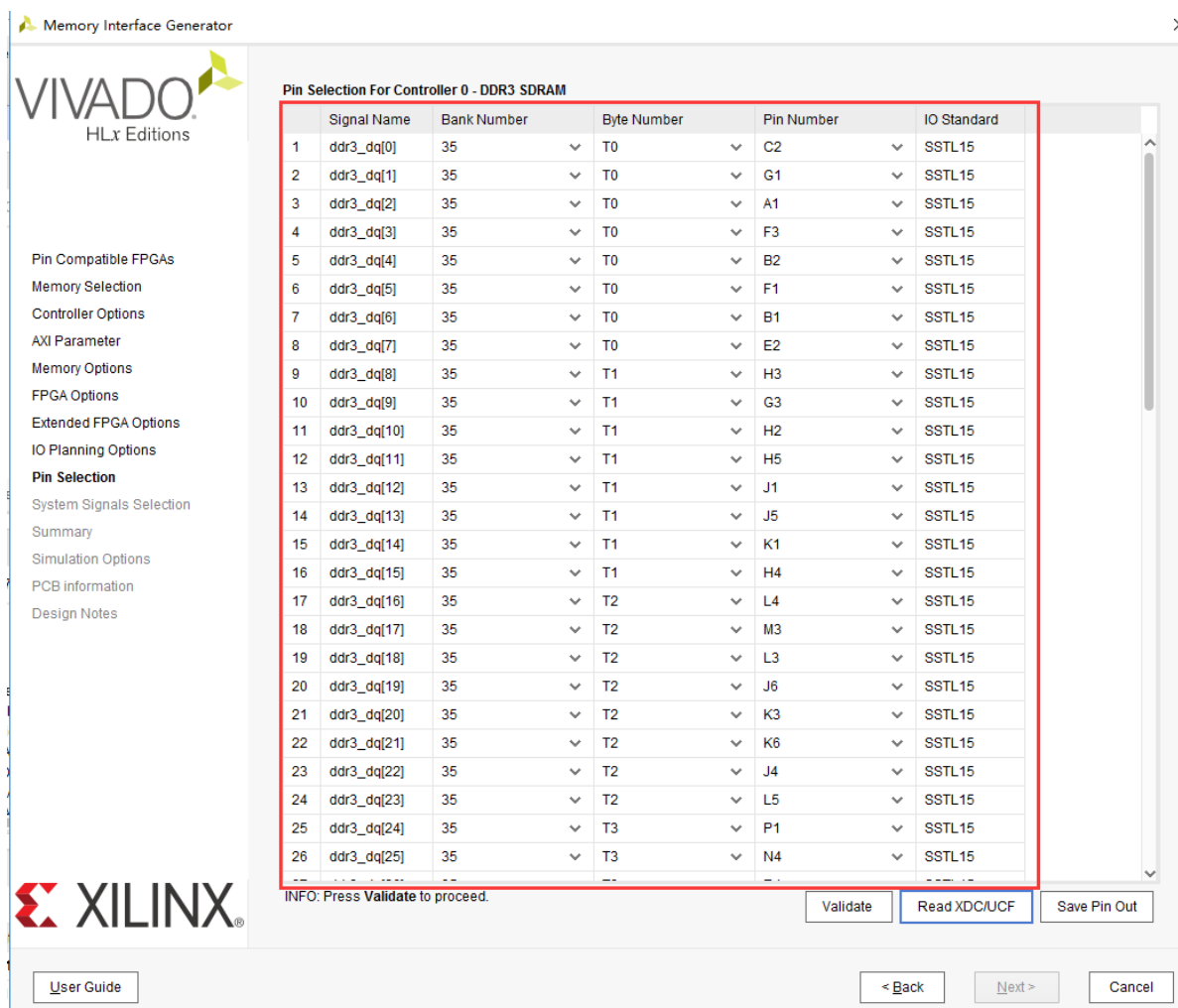
Internal Termination for High Range Banks
Select the internal termination (IN_TERM) impedance for the High Range (HR) banks. This setting applies **only** to the HR banks used in the interface.
Internal Termination Impedance: 50 Ohms

User Guide < Back Next > Cancel

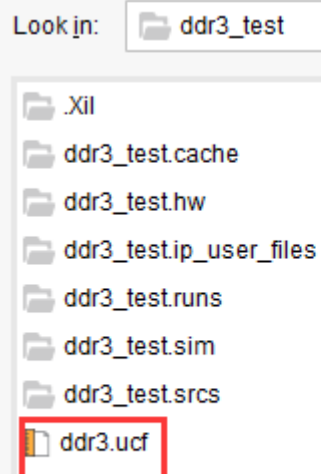
11. 点击第二项，我们需要设定一下 DDR 的管脚，点击 Next。



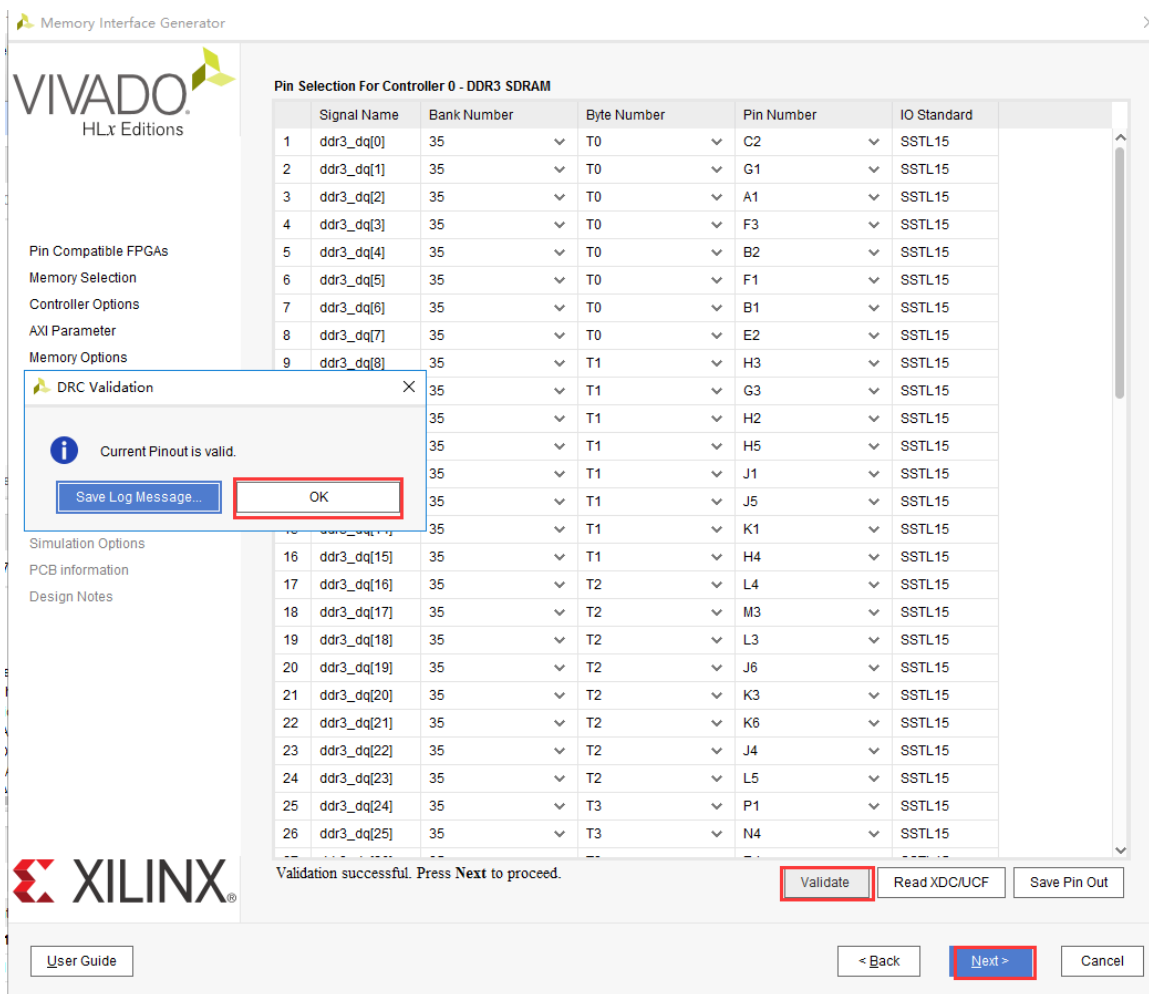
12. 在这个界面里设置 DDR3 的数据、地址和控制信号的 FPGA 管脚分配和 IO 电平。这个手工分配起来还是比较费劲的，用户可以使用点击“Read XDC/UCF”按键直接导入管脚分配文件。导入后 ddr3 的管脚分配如下。




在 ddr3_test 例程中我们已经为大家准备好了一个 ddr3.ucf 文件，用户只要直接导入这个 ucf 文件就可以完成 ddr3 的管脚分配。




再点击"validate" 按验证一下，通过后点击 Next。



13. 这里软件默认设置，直接点击 Next。



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



System Signals Selection

Select the system pins below appropriately for the interface. Customization of these pins can also be made in the XDC after the design is generated. For more information see [UG586 Bank and Pin rules](#).

System Clock and Reference Clock pin selections will not be visible if the 'No Buffer' option was selected in the FPGA Options page.

System Signals

These signals may be connected internally to other logic or brought out to a pin.

- **sys_rst**: This input signal is used to reset the interface.
- **init_calib_complete**: This signal indicates that the interface has completed calibration and memory initialization and is ready for commands. LOC constraint will be generated in XDC for Example design only based on "Pin Number" selection below.
- **error**: This output signal indicates that the traffic generator in the Example Design has detected a data mismatch. This signal does not exist in the User Design.

Signal Name	Bank Number	Pin Number
sys_rst	Select Bank	No connect
init_calib_complete	Select Bank	No connect
tg_compare_error	Select Bank	No connect

All pins must be constrained to specific locations in order to generate a bit file in the implementation phase (this is not required for simulation).

[User Guide](#)

< Back

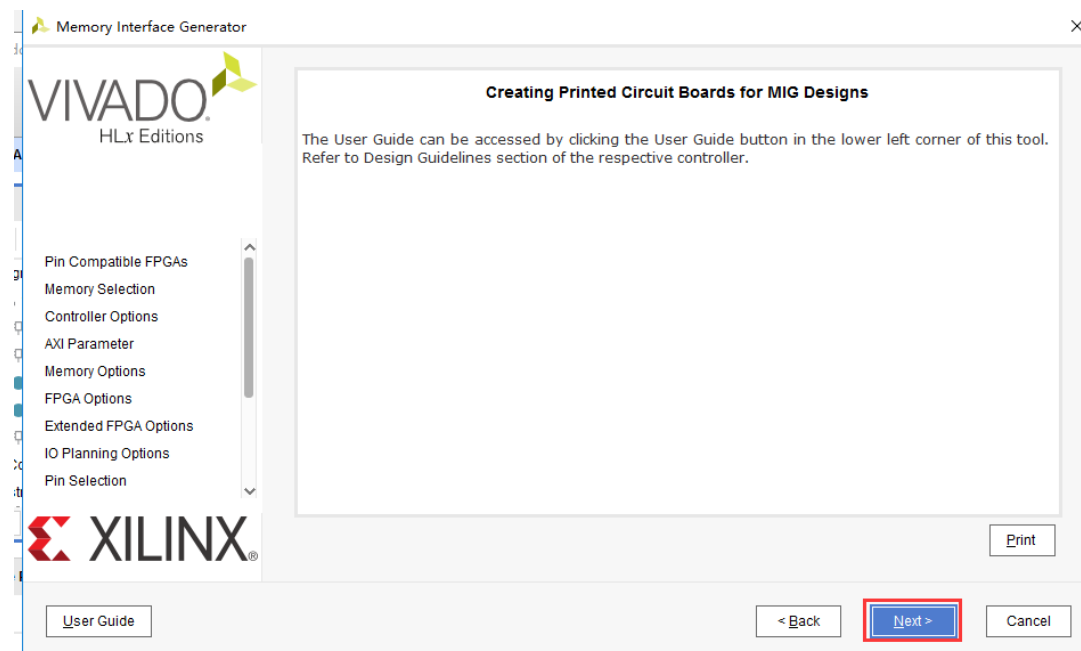
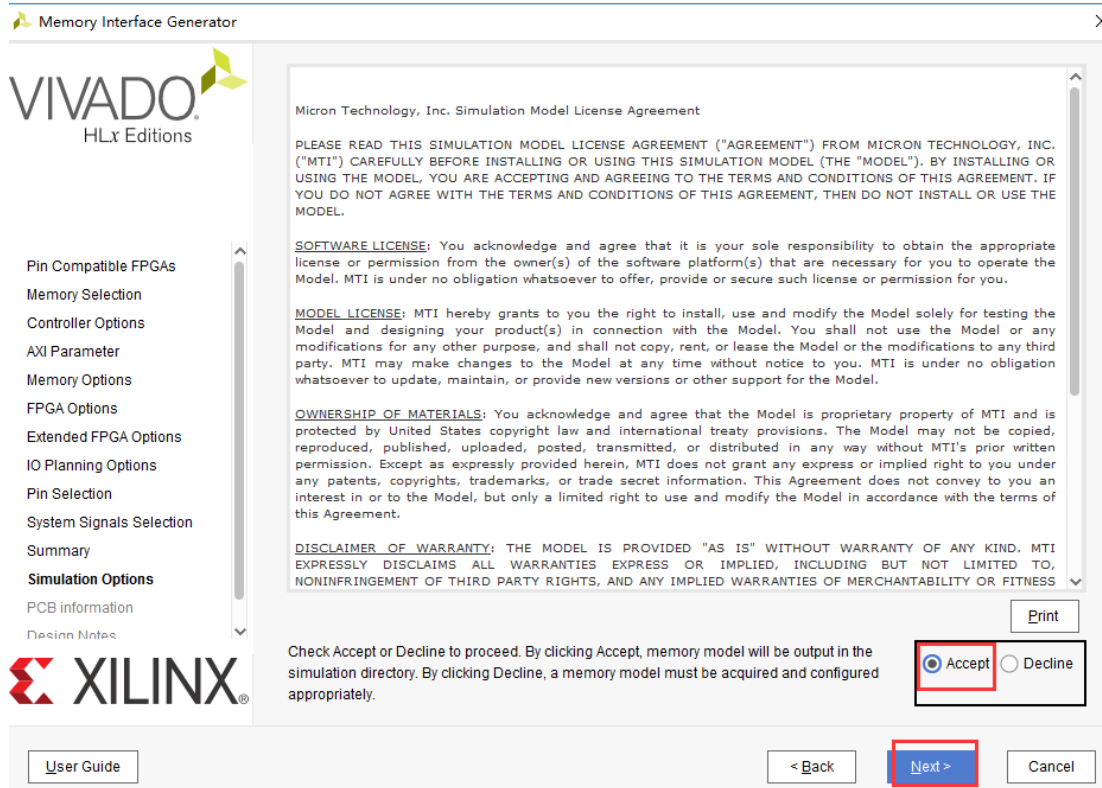
Next >

Cancel

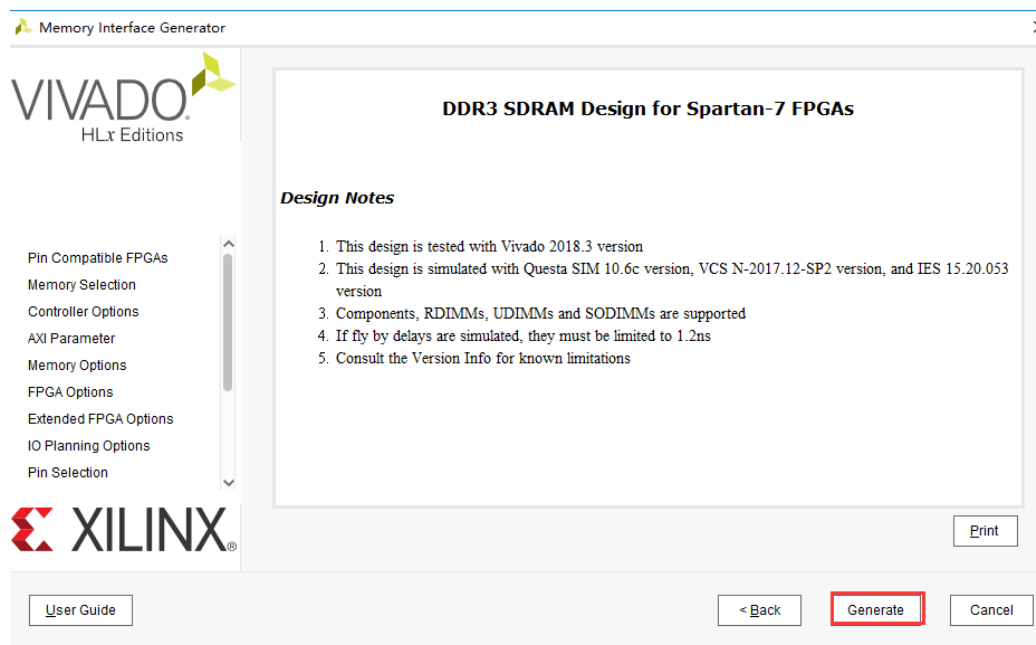
14. 显示 ddr3 IP 配置的概况，检查一下，没有问题就点击 Next



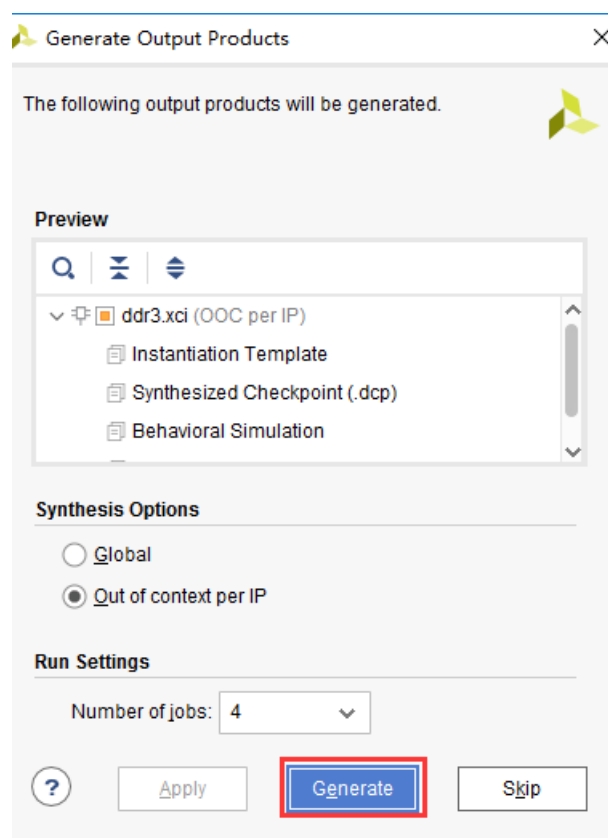
15. 选择 Accept , 点击 Next.



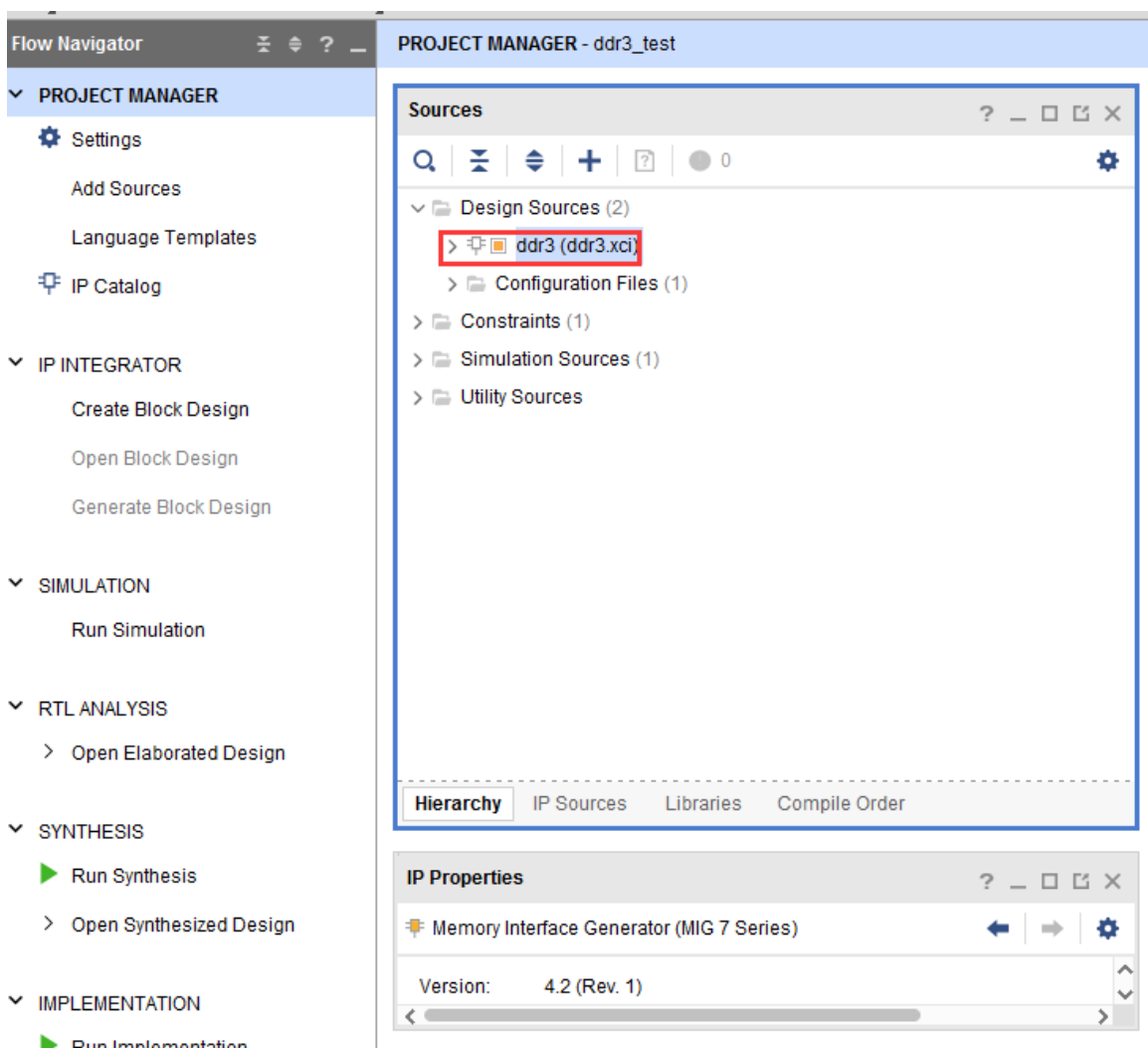
16. 点击 Generate 按钮生成 MIG 控制器。



17. 点击 Generate 按钮生成 MIG 相关的设计文档。



18. 这时在项目中添加了一个 ddr3 的 IP。



关于 MIG 7 Series 详细介绍请大家参考 Xilinx 提供的文档 “ug586_7Series_MIS.pdf”。

4.2 ddr3 时钟设计

在 AX7101/AX7102/AX7103 开发板中，FPGA 采用的差分晶振频率 200M,而上节设计的 ddr3 IP 需要的时钟为单端 200MHz,这里通过 IBUFDS 产生 DDR3 时钟 sys_clk_200MHz：代码如下：

```
IBUFDS sys_clk_ibufgds
(
    .O          (sys_clk_200MHz),
    .I          (sys_clk_p),
    .IB        (sys_clk_n)
);
```

5 DDR3 测试程序设计

如果让大家在自己编写一个 DDR3 的测试程序，或者在自己的程序中使用这个 DDR3 IP 的话，相信很多同学还是会找不到北，不知从何下手了。那这节我们就来学习自己编写一个 DDR3 的驱动程序来跟 DDR3 IP 进行通信，来实现 DDR3 的数据读写。

1. 首先我们来编写一个 DDR 的驱动程序 mem_burst.v

虽然 DDR3 IP 产生的 ddr3.v 程序里有用户部分的接口，但这部分的接口时序对我们来说还是过于复杂，读写数据时用的接口信号也比较多。所以需要编写一个比较通用，接口比较简单的程序来封装 ddr3.v，这就是 mem_burst.v 程序是的功能和目的。

通过 mem_burst 程序，用户读写 DDR3 数据就变得简单。写入数据的时候，只要控制写请求信号 wr_burst_req，写长度 wr_burst_len，写地址 wr_burst_addr 和写数据 wr_burst_data。同样，读数据的时候，只要控制读请求信号 rd_burst_req，读长度 rd_burst_len，读地址 rd_burst_addr，读数据有效 rd_burst_data_valid 和读数据 rd_burst_data。

下面来编写 mem_burst.v 的具体程序吧！

```
module mem_burst
#(
parameter          MEM_DATA_BITS    = 64,
parameter          ADDR_BITS       = 24
)
(
input              rst,                //reset
input              mem_clk,            //clock
input              rd_burst_req,       //read burst request
input              wr_burst_req,       //write burst request
input[9:0]         rd_burst_len,       //read burst length
input[9:0]         wr_burst_len,       //write burst length
input[ADDR_BITS - 1:0] rd_burst_addr,  //read burst address
input[ADDR_BITS - 1:0] wr_burst_addr,  //write burst address
output            rd_burst_data_valid, //read burst data valid
output            wr_burst_data_req,   // write burst data request
output[MEM_DATA_BITS - 1:0] rd_burst_data, //read burst data
input[MEM_DATA_BITS - 1:0] wr_burst_data, //write burst data
output            rd_burst_finish,     // read burst finish flag
output            wr_burst_finish,     // write burst finish flag
output            burst_finish,        // write burst finish flag
)
//*****
// xilinx MIG IP application interface ports
//*****
output[ADDR_BITS-1:0] app_addr,
output[2:0]          app_cmd,
output              app_en,
output [MEM_DATA_BITS-1:0] app_wdf_data,
output              app_wdf_end,
output [MEM_DATA_BITS/8-1:0] app_wdf_mask,
output              app_wdf_wren,
input [MEM_DATA_BITS-1:0] app_rd_data,
input              app_rd_data_end,
input              app_rd_data_valid,
input              app_rdy,
```

```

input      app_wdf_rdy,
input      ui_clk_sync_rst,
input      init_calib_complete
);
assign app_wdf_mask = {MEM_DATA_BITS/8{1'b0}};
/*****
Define the state of the state machine
*****/
localparam IDLE = 3'd0;
localparam MEM_READ = 3'd1;
localparam MEM_READ_WAIT = 3'd2;
localparam MEM_WRITE = 3'd3;
localparam MEM_WRITE_WAIT = 3'd4;
localparam READ_END = 3'd5;
localparam WRITE_END = 3'd6;
localparam MEM_WRITE_FIRST_READ = 3'd7;

reg[2:0] state;
reg[9:0] rd_addr_cnt; //read address count
reg[9:0] rd_data_cnt; //read data count
reg[9:0] wr_addr_cnt; //write address count
reg[9:0] wr_data_cnt; //write data count

reg[2:0] app_cmd_r;
reg[ADDR_BITS-1:0] app_addr_r;
reg app_en_r;
reg app_wdf_end_r;
reg app_wdf_wren_r;
assign app_cmd = app_cmd_r;
assign app_addr = app_addr_r;
assign app_en = app_en_r;
assign app_wdf_end = app_wdf_end_r;
assign app_wdf_data = wr_burst_data;
assign app_wdf_wren = app_wdf_wren_r & app_wdf_rdy;
assign rd_burst_finish = (state == READ_END);
assign wr_burst_finish = (state == WRITE_END);
assign burst_finish = rd_burst_finish | wr_burst_finish;

assign rd_burst_data = app_rd_data;
assign rd_burst_data_valid = app_rd_data_valid;

assign wr_burst_data_req = (state == MEM_WRITE) & app_wdf_rdy ;

always@(posedge mem_clk or posedge rst)
begin
    if(rst)
    begin
        app_wdf_wren_r <= 1'b0;
    end
    else if(app_wdf_rdy)
        app_wdf_wren_r <= wr_burst_data_req;
end
/*****
Generate read and write burst state machine
*****/
always@(posedge mem_clk or posedge rst)
begin
    if(rst)
    begin
        state <= IDLE;
        app_cmd_r <= 3'b000;
        app_addr_r <= 0;
        app_en_r <= 1'b0;
        rd_addr_cnt <= 0;
        rd_data_cnt <= 0;
        wr_addr_cnt <= 0;
        wr_data_cnt <= 0;
        app_wdf_end_r <= 1'b0;
    end
    else if(init_calib_complete == 1'b1) //Jump to read and write state machine when ddr initialization
    completed
    begin
        case(state)
            IDLE:
            begin
                if(rd_burst_req) //jump to MEM_READ state when rd_burst_req is high

```

```

begin
    state <= MEM_READ;
    app_cmd_r <= 3'b001;
    app_addr_r <= {rd_burst_addr, 3'd0};
    app_en_r <= 1'b1;
end
else if(wr_burst_req) //jump to MEM_WRITE state when wr_burst_req is high
begin
    state <= MEM_WRITE;
    app_cmd_r <= 3'b000;
    app_addr_r <= {wr_burst_addr, 3'd0};
    app_en_r <= 1'b1;
    wr_addr_cnt <= 0;
    app_wdf_end_r <= 1'b1;
    wr_data_cnt <= 0;
end
end
MEM_READ:
begin
    if(app_rdy)
    begin
        app_addr_r <= app_addr_r + 8;
        if(rd_addr_cnt == rd_burst_len - 1) //wait after reading burst address end
        begin
            state <= MEM_READ_WAIT;
            rd_addr_cnt <= 0;
            app_en_r <= 1'b0;
        end
        else
            rd_addr_cnt <= rd_addr_cnt + 1; //read address count
        end
    end
    if(app_rd_data_valid)
    begin
        if(rd_data_cnt == rd_burst_len - 1)
        begin
            rd_data_cnt <= 0;
            state <= READ_END;
        end
        else
        begin
            rd_data_cnt <= rd_data_cnt + 1; //read data count
        end
    end
end
MEM_READ_WAIT:
begin
    if(app_rd_data_valid)
    begin
        if(rd_data_cnt == rd_burst_len - 1) //jump to READ_END state
        begin
            rd_data_cnt <= 0;
            state <= READ_END;
        end
        else
        begin
            rd_data_cnt <= rd_data_cnt + 1; //data count
        end
    end
end
MEM_WRITE_FIRST_READ:
begin
    app_en_r <= 1'b1;
    state <= MEM_WRITE;
    wr_addr_cnt <= 0;
end
MEM_WRITE:
begin
    if(app_rdy)
    begin
        app_addr_r <= app_addr_r + 'b1000;
        if(wr_addr_cnt == wr_burst_len - 1) // write burst address end
        begin
            app_wdf_end_r <= 1'b0;
            app_en_r <= 1'b0;
        end
        else
        begin

```

```

        wr_addr_cnt <= wr_addr_cnt + 1; //write address count
    end
end

if(wr_burst_data_req)
begin
    if(wr_data_cnt == wr_burst_len - 1)
    begin
        state <= MEM_WRITE_WAIT;
    end
    else
    begin
        wr_data_cnt <= wr_data_cnt + 1; //write data count
    end
end

end

end
READ_END:
    state <= IDLE;
MEM_WRITE_WAIT:
begin
    if(app_rdy)
    begin
        app_addr_r <= app_addr_r + 'b1000;
        if(wr_addr_cnt == wr_burst_len - 1) //wait after writing burst address end
        begin
            app_wdf_end_r <= 1'b0;
            app_en_r <= 1'b0;
            if(app_wdf_rdy)
                state <= WRITE_END;
        end
        else
        begin
            wr_addr_cnt <= wr_addr_cnt + 1;
        end
    end
    else if(~app_en_r & app_wdf_rdy)
        state <= WRITE_END;
    end
    WRITE_END:
        state <= IDLE;
    default:
        state <= IDLE;
endcase
end
endmodule

```

mem_burst.v 程序的功能就是把外部的 burst 读请求和写请求转化成 DDR3 IP 用户接口的所需的信号和时序。下面分别是读和写的流程:

DDR Burst 读:

当程序在 IDLE 状态接收到读请求(rd_burst_req 为高)时, 会向 DDR3 IP 的用户接口发送第一个数据读命令 (读命令、地址、命令有效) 信号。并会进入 MEM_READ 状态, 在 MEM_READ 状态里, 如果判断 DDR3 IP 的用户接口空闲的话, 会发送剩余的数据读命令 (地址增加), 发送完成转到 MEM_READ_WAIT 状态。另外在这个 MEM_READ 状态里, 还需要判断 DDR3 IP 从 DDR3 里读出的数据是否有效来统计读出的数据是否为读 burst 长度。

在 MEM_READ_WAIT 状态里读取读 burst 长度的 DDR3 的数据。数据全部读出完成后进入 READ_END 状态, 再返回 IDLE 状态。

DDR Burst 写:

当程序在 IDLE 状态接收到写请求(wr_burst_req 为高)时, 会向 DDR3 IP 的用户接口发送第一个数据写命令 (写命令、地址、命令有效) 信号。并会进入 MEM_WRITE 状态, 在 MEM_WRITE 状态里, 如果判断 DDR3 IP 的用户接口空闲的话, 会发送剩余的数据写命令 (地址增加), 同时在这个 MEM_WRITE 状态里, 还会向 DDR3 IP 的数据 FIFO 里写入要写到 DDR 的数据。当写入到 DDR3 IP 的 FIFO 的数据长度长度为写 burst 长度时转到 MEM_WRITE_WAIT 状态。在 MEM_WRITE_WAIT 状态, 判断 DDR 数据写命令是否发送完成, 发送完成就进入 WRITE_END 状态, 再返回 IDLE 状态。

这里需要注意的是, 向 DDR3 IP 的用户接口发送写命令和写入 DDR 的数据是独立的, 因为写入到 DDR 的数据是需要先存储到 DDR3 IP 的 FIFO 中。只要 app_wdf_rdy 有效, 就可以往 DDR3 IP 里写入数据。写入到 DDR3 的数据需要提前准备好, 这里有一个 wr_burst_data_req 信号来请求用户提前准备要写入的 DDR3 的数据。

2. 接下去我们来编写一个 DDR3 的测试程序 mem_test.v

```
module mem_test
#(
    parameter MEM_DATA_BITS = 64,
    parameter ADDR_BITS = 24
)
(
    input rst, //reset
    input mem_clk, //input clock
    output reg rd_burst_req, //read burst request
    output reg wr_burst_req, //write burst request
    output reg[9:0] rd_burst_len, //read burst length
    output reg[9:0] wr_burst_len, //write burst length
    output reg[ADDR_BITS - 1:0] rd_burst_addr, //read burst address
    output reg[ADDR_BITS - 1:0] wr_burst_addr, //write burst address
    input rd_burst_data_valid, //read burst data valid
    input wr_burst_data_req, // write burst data request
    input[MEM_DATA_BITS - 1:0] rd_burst_data, //read burst data
    output[MEM_DATA_BITS - 1:0] wr_burst_data, //write burst data
    input rd_burst_finish, // read burst finish flag
    input wr_burst_finish, // write burst finish flag
    output reg error
);
localparam IDLE = 3'd0;
localparam MEM_READ = 3'd1;
localparam MEM_WRITE = 3'd2;

reg[2:0] state;
reg[7:0] wr_cnt;
reg[MEM_DATA_BITS - 1:0] wr_burst_data_reg;
assign wr_burst_data = wr_burst_data_reg;
reg[7:0] rd_cnt;
always@(posedge mem_clk or posedge rst)
begin
    if(rst)
        error <= 1'b0;
    else
        error <= (state == MEM_READ) && rd_burst_data_valid && (rd_burst_data !=
{ (MEM_DATA_BITS/8){rd_cnt}});
end
```

```

end
/*****
Generate write burst data when writing burst data request is valid
*****/
always@(posedge mem_clk or posedge rst)
begin
    if(rst)
    begin
        wr_burst_data_reg <= {MEM_DATA_BITS{1'b0}};
        wr_cnt <= 8'd0;
    end
    else if(state == MEM_WRITE)
    begin
        if(wr_burst_data_req)//Generate write burst data
        begin
            wr_burst_data_reg <= {(MEM_DATA_BITS/8){wr_cnt}};
            wr_cnt <= wr_cnt + 8'd1;
        end
        else if(wr_burst_finish)
            wr_cnt <= 8'd0;
    end
end
/*****
reading burst data count when data is valid
*****/
always@(posedge mem_clk or posedge rst)
begin
    if(rst)
    begin
        rd_cnt <= 8'd0;
    end
    else if(state == MEM_READ)
    begin
        if(rd_burst_data_valid)//reading burst data count
        begin
            rd_cnt <= rd_cnt + 8'd1;
        end
        else if(rd_burst_finish)
            rd_cnt <= 8'd0;
    end
    else
        rd_cnt <= 8'd0;
end
/*****
Generate a state machine that cyclically reads and writes MIG ip
*****/
always@(posedge mem_clk or posedge rst)
begin
    if(rst)
    begin
        state <= IDLE;
        wr_burst_req <= 1'b0;
        rd_burst_req <= 1'b0;
        rd_burst_len <= 10'd128;
        wr_burst_len <= 10'd128;
        rd_burst_addr <= 0;
        wr_burst_addr <= 0;
    end
    else
    begin
        case(state)
            IDLE:
            begin
                state <= MEM_WRITE;
                wr_burst_req <= 1'b1;
                wr_burst_len <= 10'd128;
            end
            MEM_WRITE:
            begin
                if(wr_burst_finish)//Start jumping to MEM_READ when writing burst finish
                begin
                    state <= MEM_READ;
                    wr_burst_req <= 1'b0;
                    rd_burst_req <= 1'b1;
                    rd_burst_len <= 10'd128;
                    rd_burst_addr <= wr_burst_addr;
                end
            end
        endcase
    end
end

```

```

        end
    end
    MEM_READ:
    begin
        if(rd_burst_finish)//Start jumping to MEM_WRITE when reading burst finish
        begin
            state <= MEM_WRITE;
            wr_burst_req <= 1'b1;
            wr_burst_len <= 10'd128;
            rd_burst_req <= 1'b0;
            wr_burst_addr <= wr_burst_addr + 128;
        end
    end
    default:
        state <= IDLE;
    endcase
end
endmodule

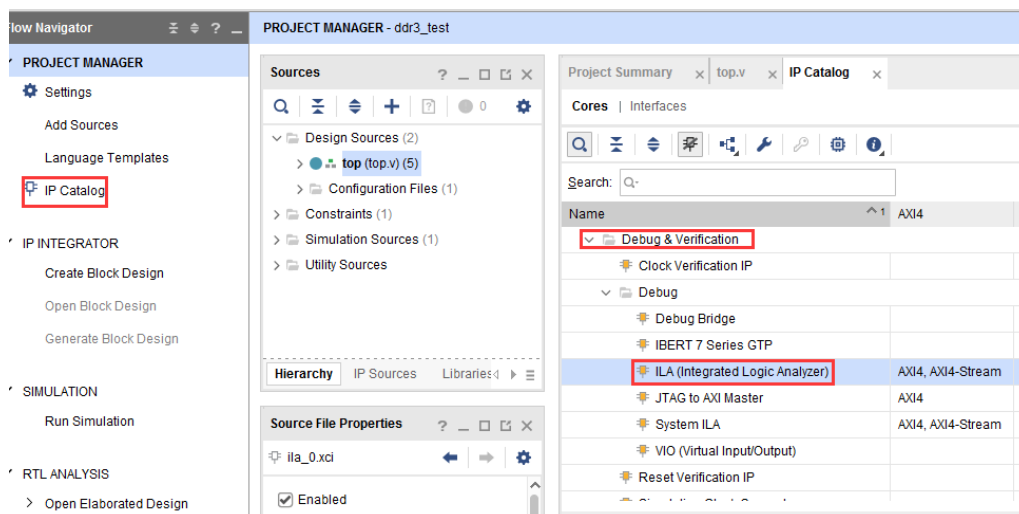
```

mem_test.v 测试程序里主要实现 ddr 的 burst 读和 burst 写的功能, 程序里产生读写请求信号, 地址和测试数据并校验读和写的数据是否正确, 这里 burst 的数据长度是 128。如果 DDR 的读写数据错误(写入的数据和读出的数据不一致), error 信号会变高。

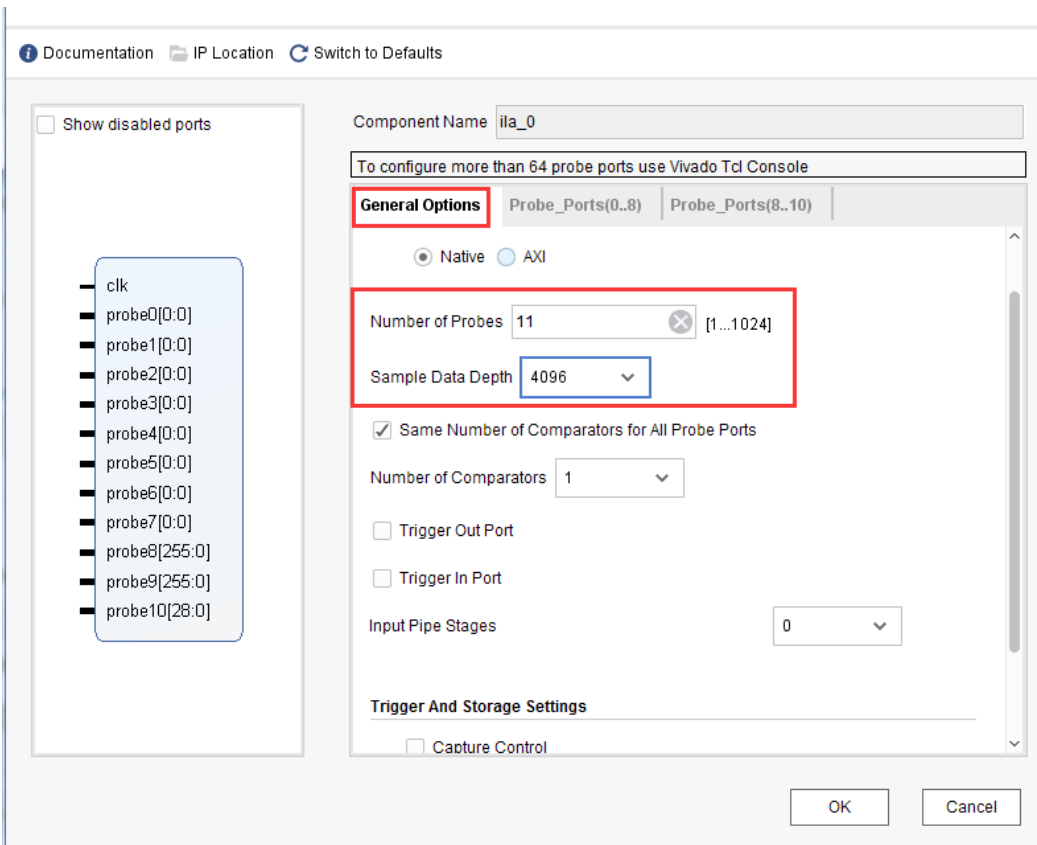
3. 接下去我们来编写一个顶层程序 top.v

top.v 程序里例化 mem_burst 模块、mem_test 模块和 DDR3 IP 模块 ddr3.v, DDR3 的时钟产生模块, 另外实例化了一个 ila_0 IP, 用来观察 DDR Burst 读和 Burst 写的数据、地址和控制信号。所以我们还需要向工程中再添加一个 ila 的 IP。

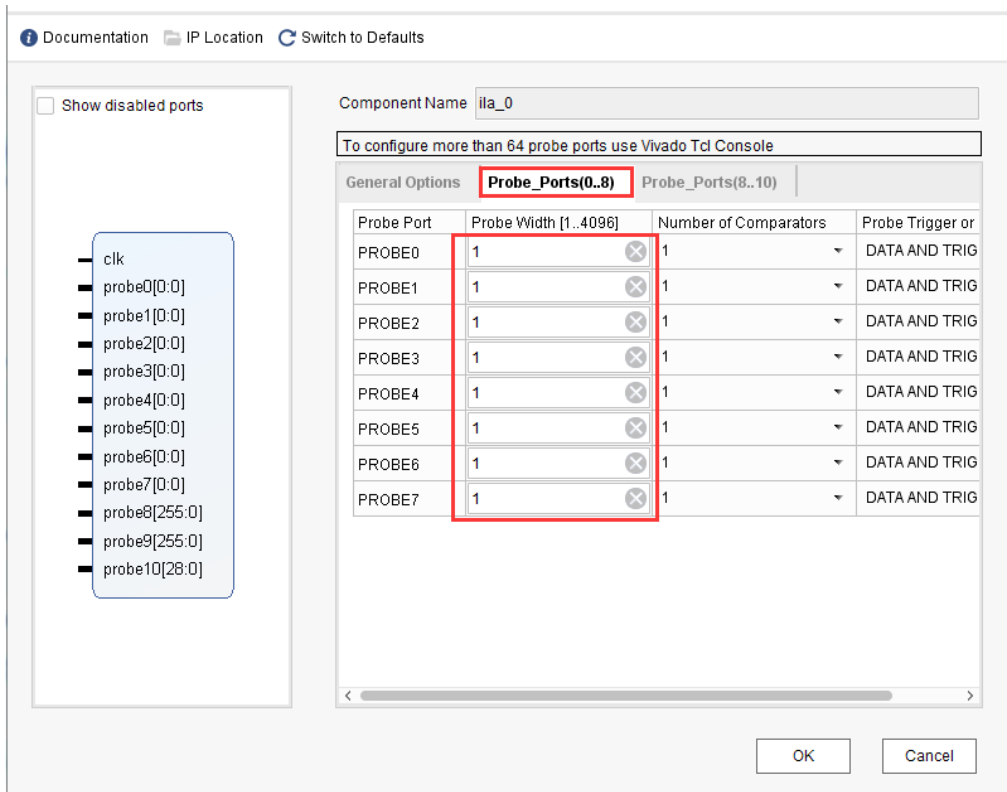
4. 添加 ila IP, 打开 IP Catalog 界面, 选择 Debug & Verification\Debug 下的 ILA (Integrated Logic Analyzer), 双击打开。



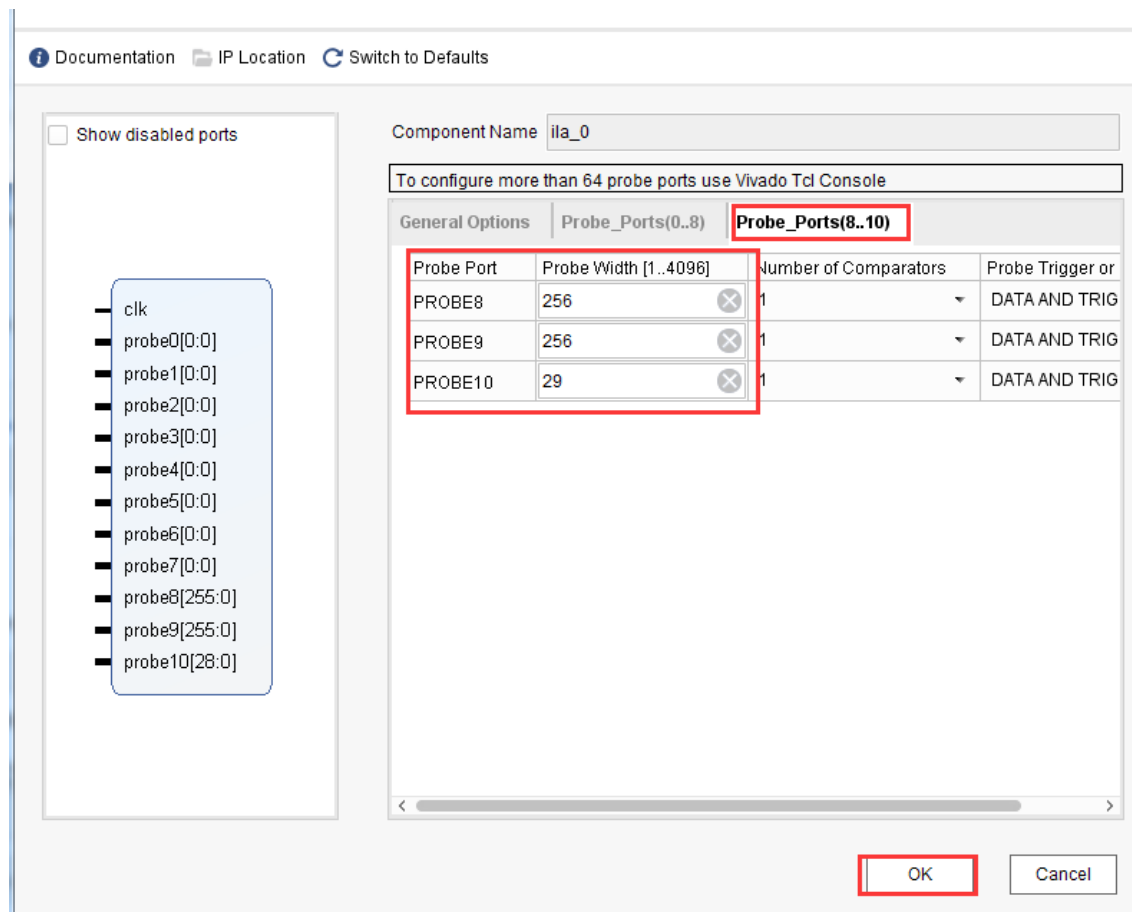
Component 名字为 ila_0, 名字需要跟程序里的一样。Probe 的数量为 11, 就是 11 个采样通道, 采样的数据深度为 4096, 采样深度越深, 采样的数据量越大, 但会消耗更多的 FPGA 逻辑资源。



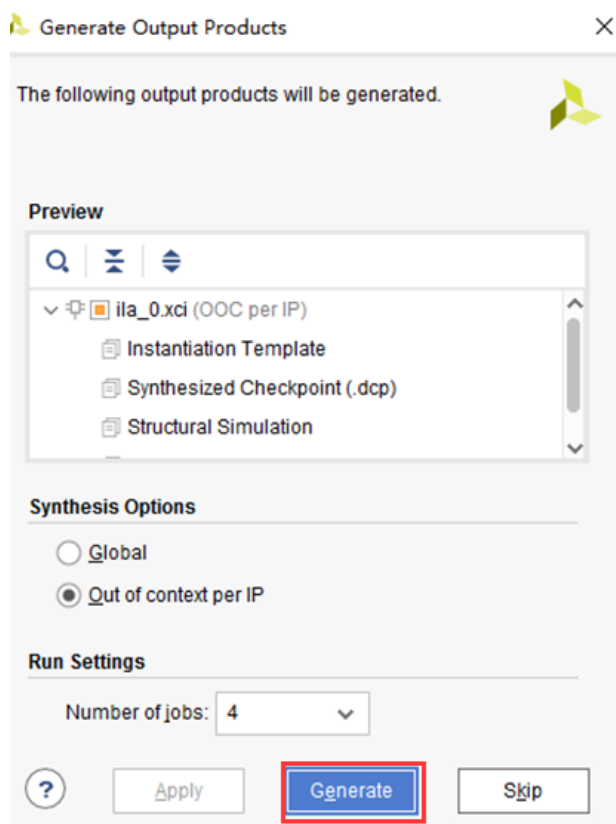
再对前面 8 个通道设置数据宽度，这里因为我们每个通道只是采样一个 bit 的信号，所以数据宽度都为 1。



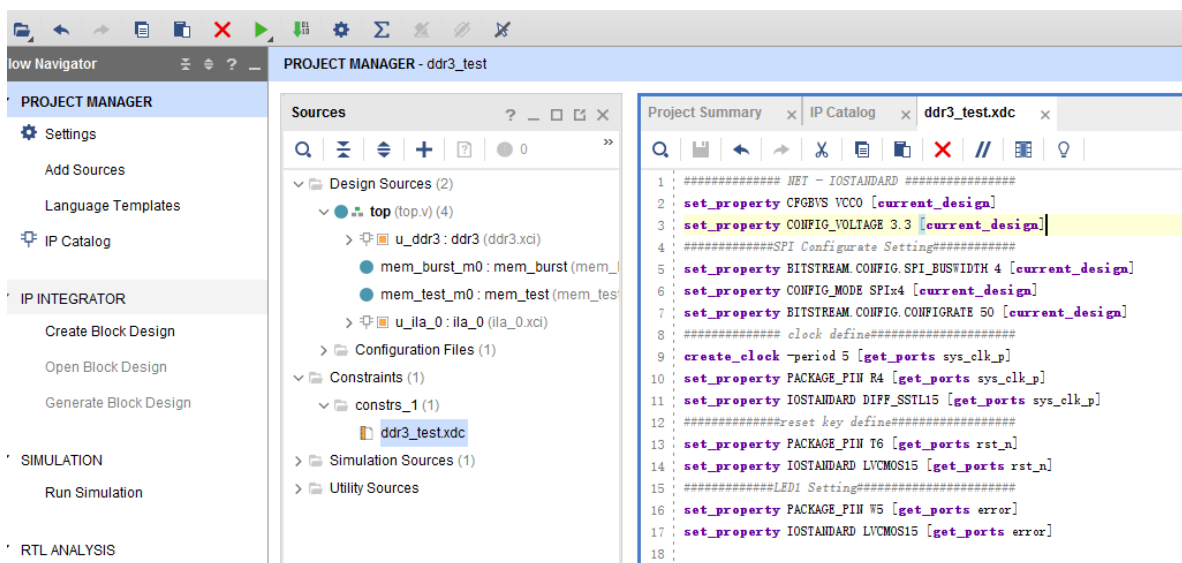
再设置后面 8~10 通道的数据宽度，8 通道和 9 通道设置成 256 数据宽度，用来采样 DDR 读写数据，10 通道设置成 29，用来采样 DDR 的地址。点击 OK 完成。



点击 Generate 按钮生成 IP 的设计文件。

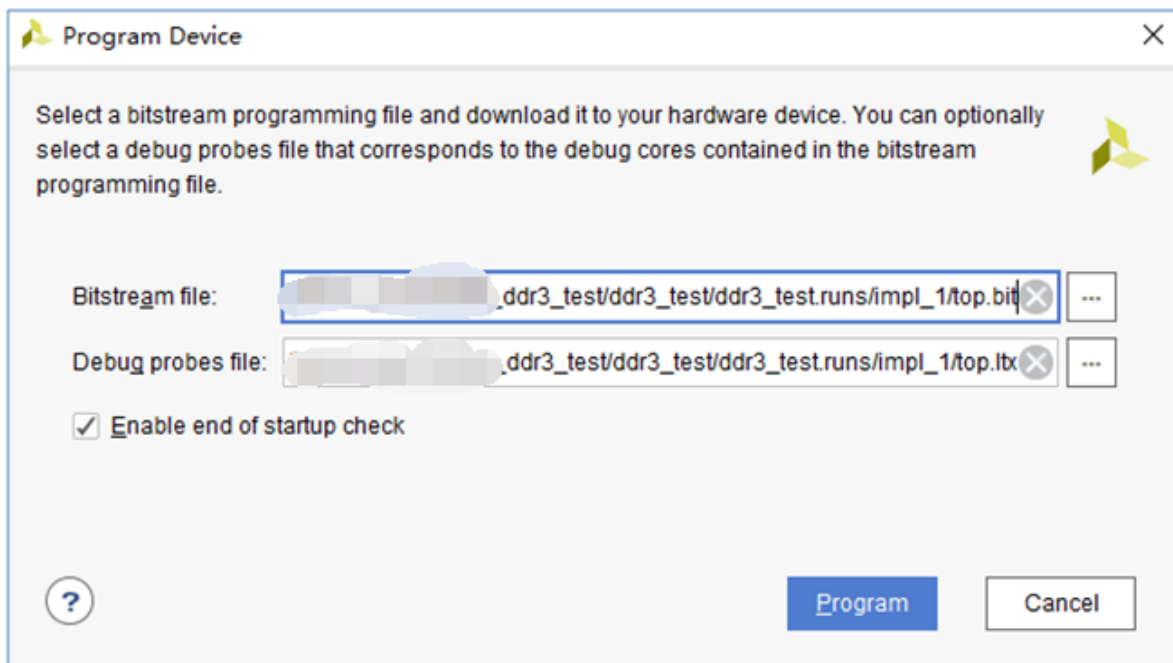


再来添加 xdc 管脚约束文件 ddr3_test.xdc，把 error 信号连接到核心板的 LED1 上，如果 error 为高电平，LED1 亮。另外设置系统的复位信号 rst_n 和 FPGA 时钟输入 sys_clk，如下图：

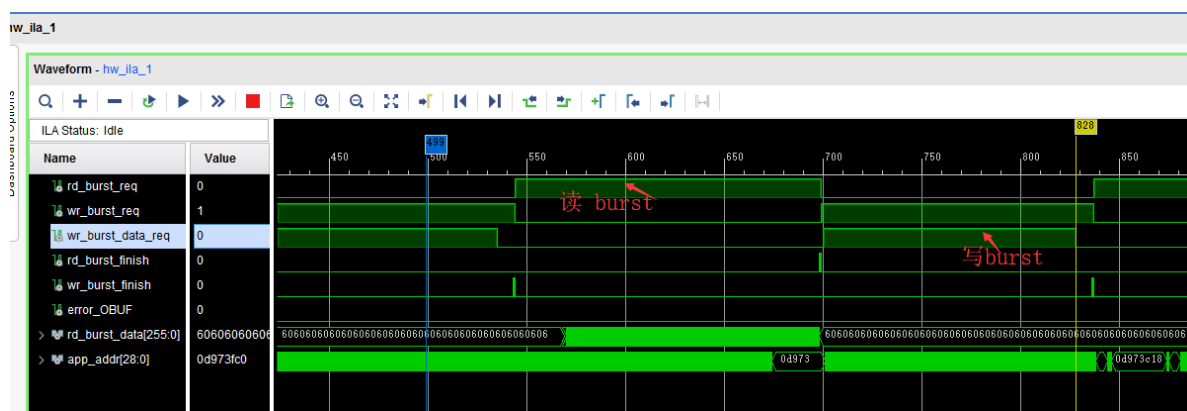


6 下载和测试

编译程序生成 top.bit 文件，然后下载 bit 文件到 FPGA。



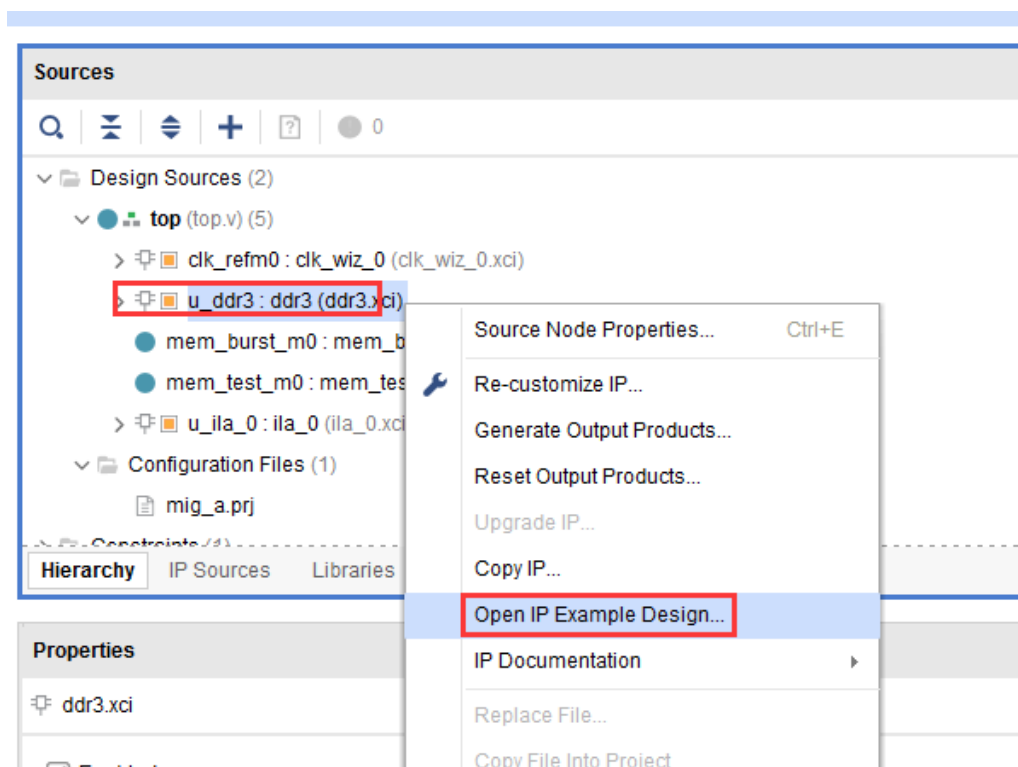
下载后软件会自动打开 ila 的波形调试界面，点击“Run trigger for ILA core”按钮运行 ila 采集数据。



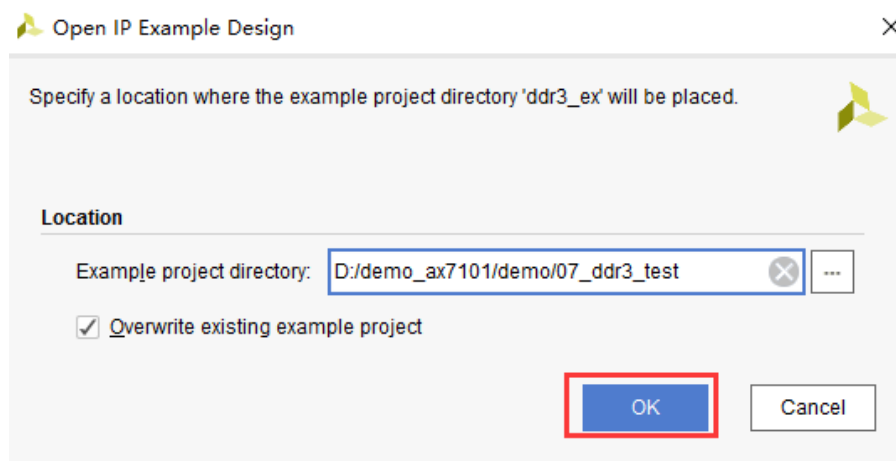
另外我们也可以检查核心板上的 LED1 是否点亮，如果点亮，说明 error 信号为高，说明 DDR3 的读写数据错误，如果一直熄灭，说明 DDR3 读写数据正确。

7 DDR3 仿真

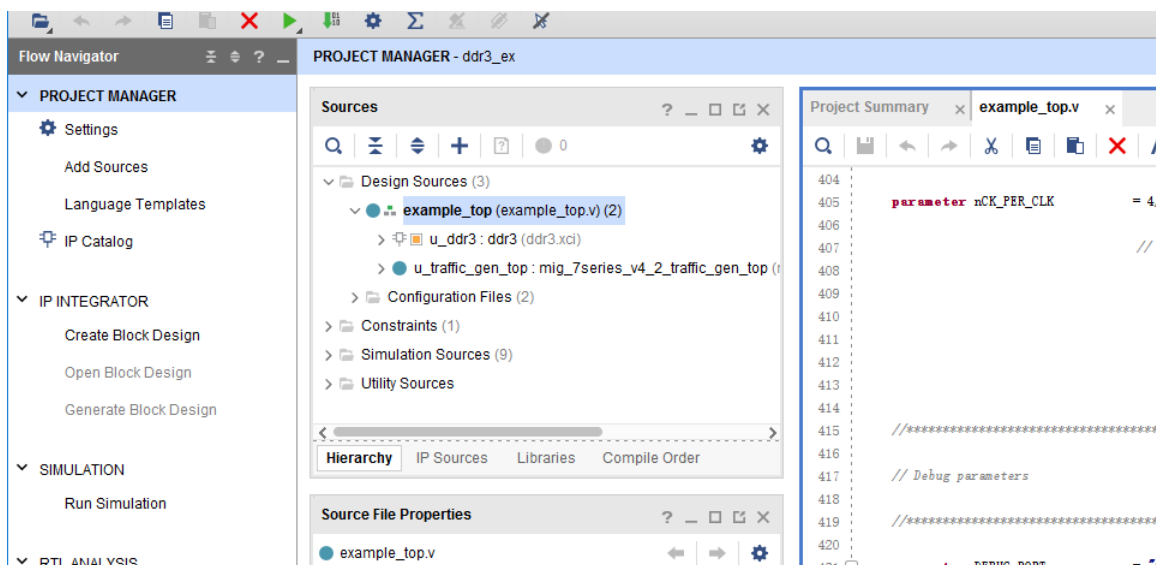
1. 在上面的例程中右键选择 ddr3 IP，在弹出的下拉菜单里选择 Open IP Example Design。



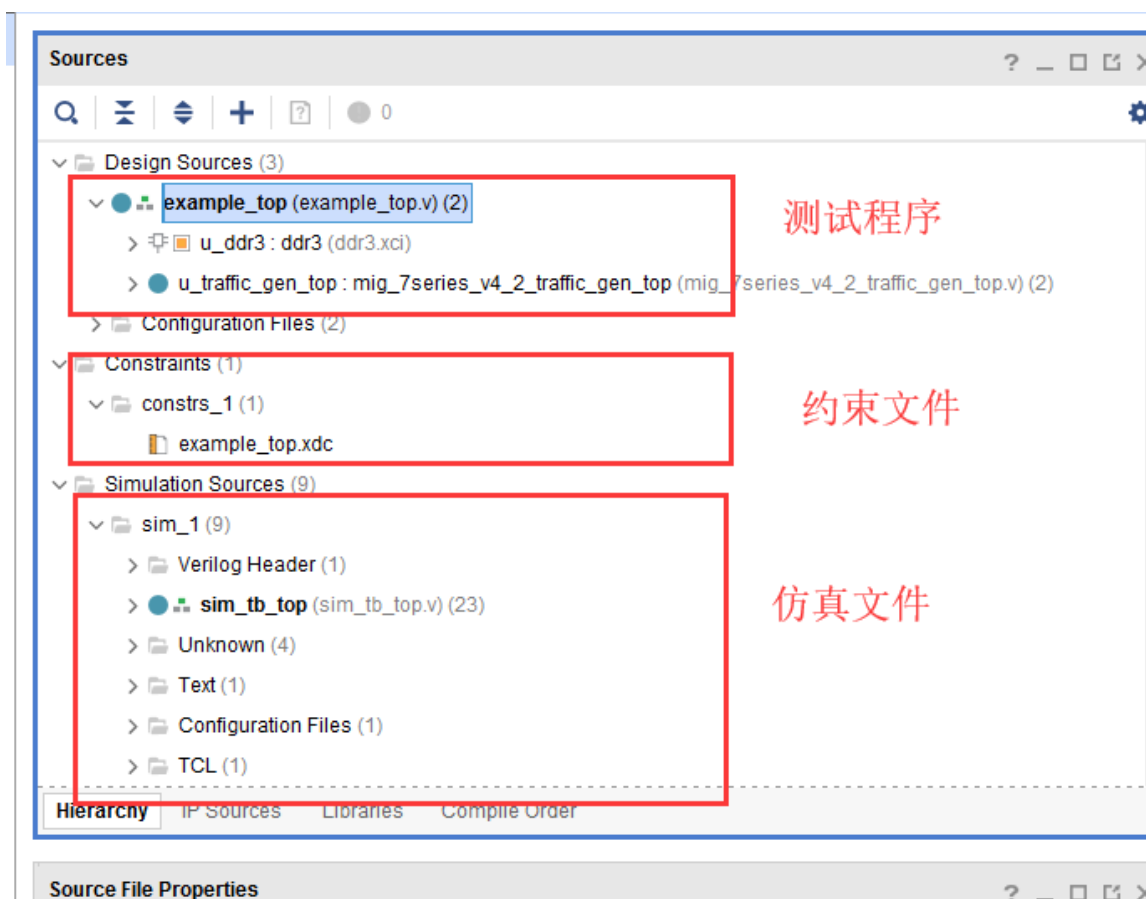
ddr3_ex 项目生成的路径是选择默认，可以不用修改，点击 OK。



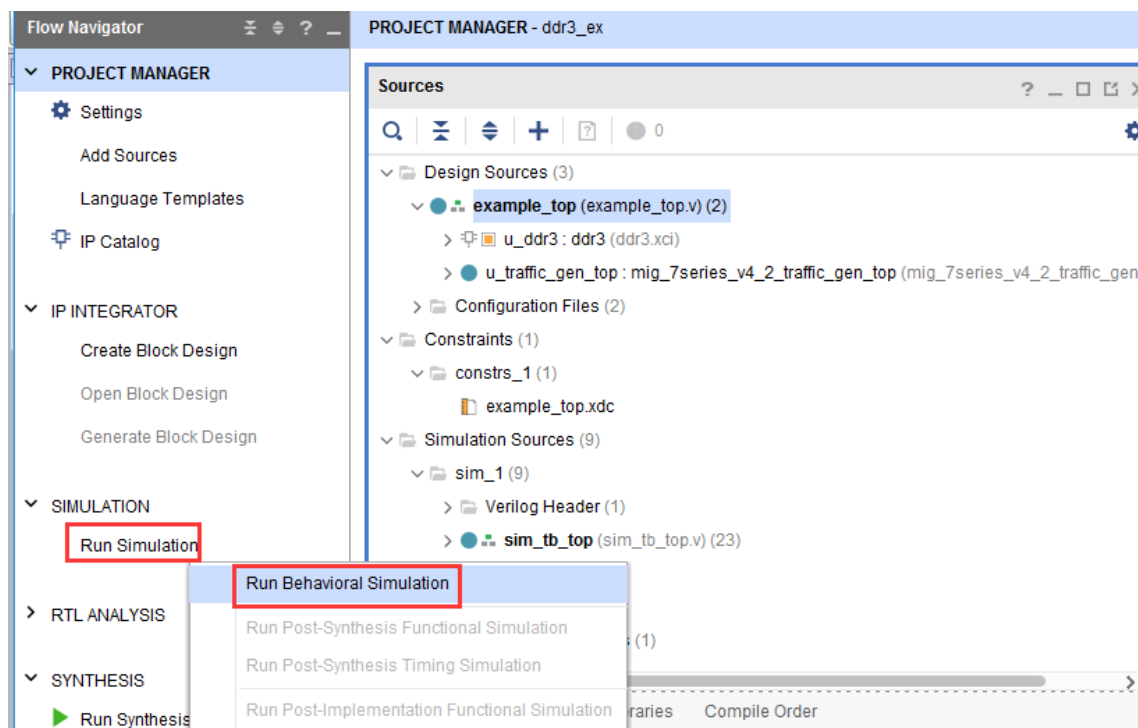
软件会自动打开生成的 ddr3_ex 工程，如下图：



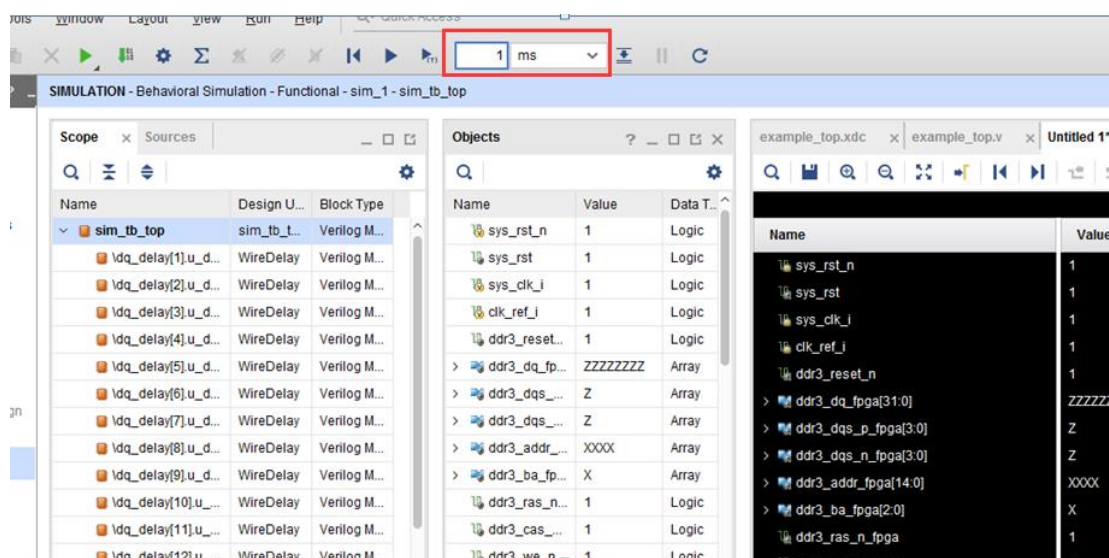
这个 ddr3_ex 工程里，软件已经自动编写了 ddr3 的测试程序、管脚定义文件 xdc 文件和仿真程序。



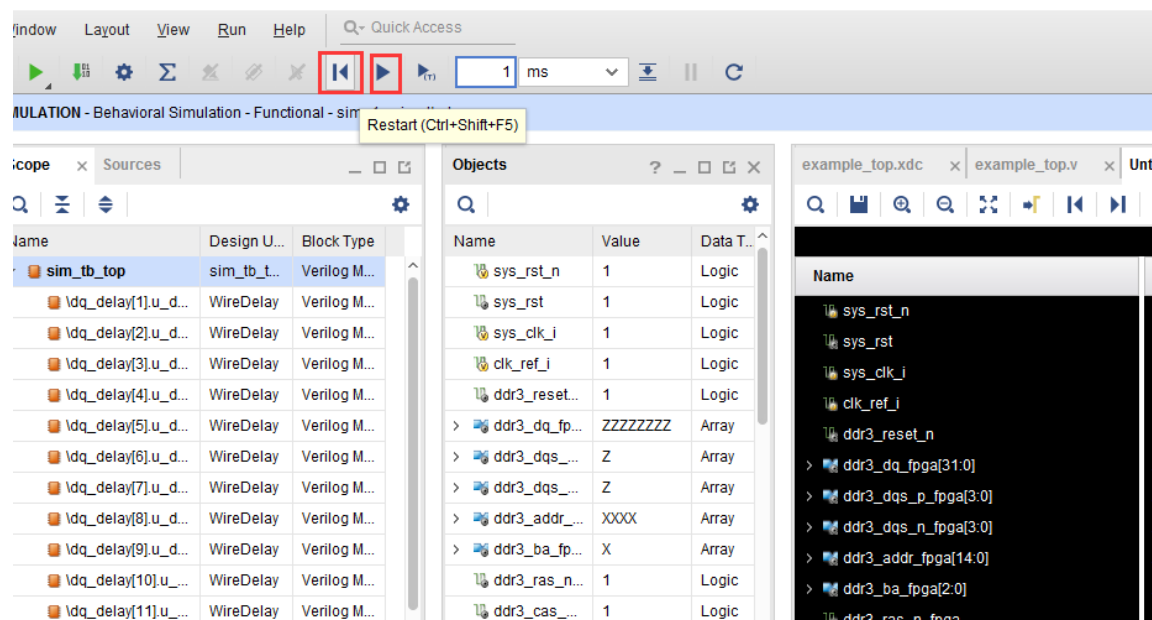
2. 点击 Run Simulation 按钮，再选择 Run Behavioral Simulation。这里我们做一下行为级的仿真就可以了。



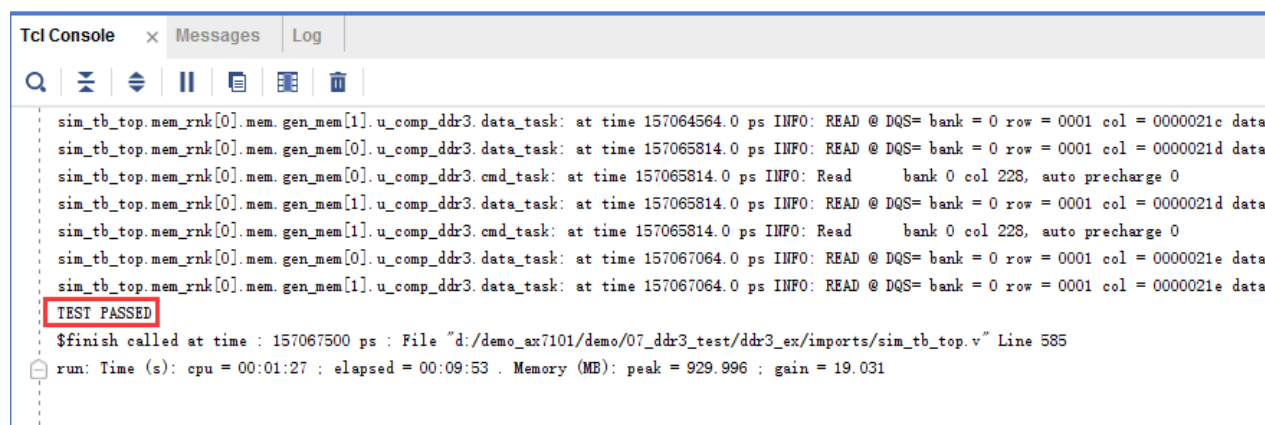
3. 打开仿真环境后设置仿真的时间长度为 1ms。



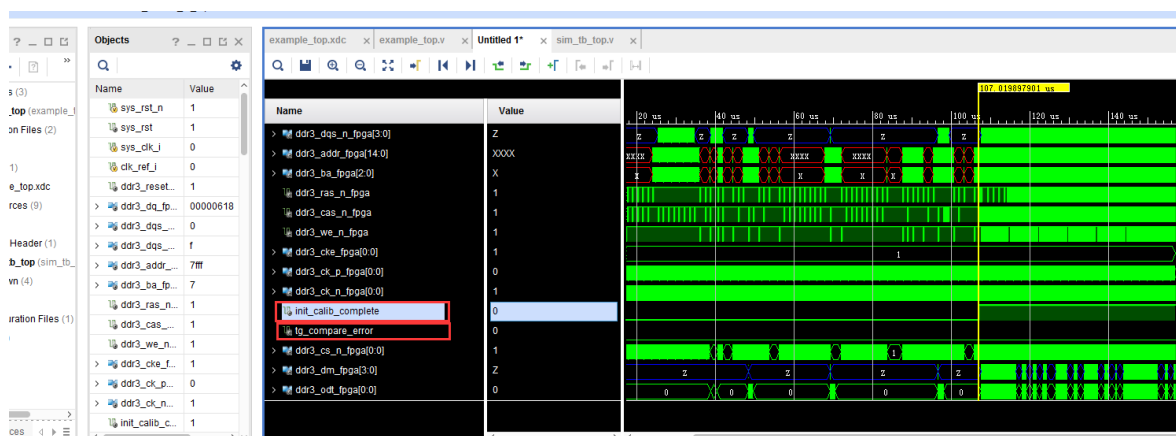
4. 分别点击 “Restart” 按钮和 “Run all” 按钮开始波形仿真。



5. 仿真的时间会比较长，仿真大概会在 155us 的时候完成。在 Tcl Console 窗口会显示 TEST PASSED 字样，说明 DDR3 的测试成功，仿真没有问题。



6. 我们再来看一下波形，在 100us 左右的地方，init_calib_done 信号变高，说明 ddr3 初始化成功，后面的时间是 DDR 数据读写的波形仿真，tg_compare_error 一直为低，说明读写的数据正确。



有兴趣的同学可以结合工程中的程序去看 ddr3 仿真波形中的每一个变量，这里就不具体介绍了。

到这里为止，我们的 DDR3 的测试和仿真全部完成了，本实验中测试例程的 mem_burst 模块非常重要，它简化了我们对 DDR3 IP 接口和时序控制的难度，我们也将在今后的例程中使用它。另外关于更多的 DDR3 IP 控制器的资料大家可以参考 Xilinx 提供的 “ug586_7Series_MIS.pdf” 文档资料。