# Altera FPGA 全速漂移 开发指南

## DDR2 控制器集成与读写测试

欢迎加入 FPGA/CPLD 助学小组一同学习交流：

EDN:

http://group.ednchina.com/GROUP_GRO_14596_1375.HTM

ChinaAET: http://group.chinaaet.com/273

淘宝店链接：http://myfpga.taobao.com/

技术咨询：orand_support@sina.com

**特权 HSC 最新资料例程下载地址：**

**http://pan.baidu.com/s/1pLmZaFx**

| 版本信息 | | |
|---|---|---|
| 时间 | 版本 | 状态 |
| 2016-06-18 | V1.00 | 创建。 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# 目录

# 1 功能概述

本实例对 Altera 提供的 DDR2 控制器 IP 核模块进行读写操作。每 1.78 秒执行一次 DDR2 的写入和读出操作。先是从 0 地址开始遍历写 256*64bits 数据到 DDR2 的地址 0-1023 中；在执行完写入后，执行一次相同地址的读操作，将读出的 256*64bits 数据写入到 FPGA 的片内 RAM 中。在 Quartus II 集成的 In-System Memory Content Editor 中可以查看片内 RAM 中规律变化的数据。

本实例系统功能框图如图 1 所示。

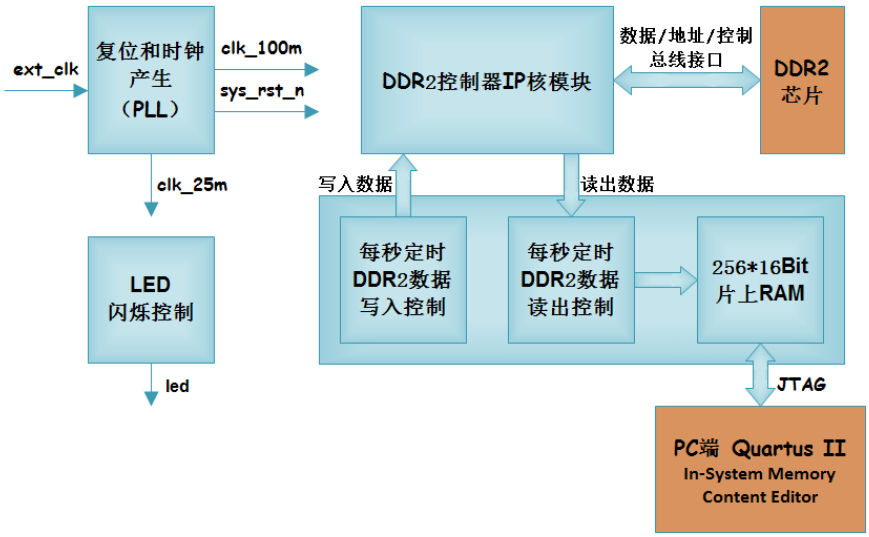



图 1 工程实例 2 功能框图

# 2 IP 核配置——片内 RAM

**FPGA 片内存储器概述**

片内存储器是基于 FPGA 的嵌入系统中最简单的存储器。因为存储是在 FPGA 内部完成的，电路板上无需外部连线。FPGA 的片内存储器可以根据需求定义存储器的大小、位宽、种类、及特殊的片内存储器特性，如 DDR 模式等。

片内存储器在基于 FPGA 的嵌入式系统的存储器中具有最高吞吐量和最低反应延时。它的反应延时通常仅为一个时钟周期。通过流水线操作访问存储器，可以使吞吐量达到每个时钟周期进行一次数据处理。

片内存储器的另一个好处是，由于它是在 FPGA 上直接实现的，它无需在板上或电路中进行写入。使用片内存储器可以节省开发时间和成本。

尽管速度很快，片内存储器在一定程度上会受到其容量的限制。FPGA 可用的片内存储器的数量由所使用的特定 FPGA 器件决定，如 Cyclone II 系列有低至 15KB 存储量的器件，Stratix III 系列却有高到 2MB 存储量的器件。

因为多数片内存储器都是易失性的，它在断电后丢失数据。然而，某些片内存储器可以在 FPGA 配置时自动初始化，相当于提供了一种非易失性的功能。

片内存储器的最佳应用场合包括作为常见的缓存、点到点的缓存、查找表以及 FIFO 等。

缓存，由于其具有低反应延时，片内存储器在微处理器中作为缓存表现良好。NIOS II 处理器使用片内存储器作为引导和数据缓存。片内存储器有限的容量作为缓存通常不是一个问题，因为缓存本身都相对很小。

点到点的缓存，低延时的存取也使得片内存储器适用于作为器件间的

缓存，即点到点的缓存。它是指处于正常的寻址空间，但与微处理器有专用接口的存储器。这些存储器主要用来实现缓存存储器的高速、低反应延时特性。

查找表，针对某些软件编程功能，尤其是数学上的功能。与在软件中进行计算相比，使用查找表储存所有可能的功能结果通常是最快的方法。片内存储器在这方面表现良好，前提是片内存储器的可用容量能够容纳可能的功能结果。

FIFO，嵌入系统经常需要管理从一个模块到另一个模块的数据流。FIFO 可以在以不同的高速运行着的模块间作为数据缓冲存储器。根据应用程序所需的 FIFO 的大小，片内存储器可以作高速和便利的 FIFO 存储。

但片内存储器不适用于需要大容量存储的应用中。因为片内存储器容量相对受限，应避免使用其储存大量的数据。然而，有些工作可以更好地利用片内存储器完成。如果应用程序使用多个小块数据，并且不是所有的数据块都适合使用片内存储器，设计者应当仔细考虑某些应用可以使用片内存储器。如果用户的目标是系统的高速性能，可以将最经常存取的数据放在片内存储器中。

**FPGA 片内 RAM 概述**

我们所使用的 Cyclone IV 系列 FPGA 器件内嵌丰富的 M9K 存储器，M9K 存储器支持以下特性。

- 每个 M9K 存储块有 8192 bits 的存储量。
- 每个端口拥有独立的读使能和写使能信号。
- 可变的端口配置。
- 所有位宽都支持单端口或者双端口模式。
- 每个端口都有可选的时钟使能信号。

- RAM 或 ROM 模式下可以初始化预加载存储数据。

本实例我们要用 Cyclone IV 的片内存储器配置一个 RAM。如图 2 所示，这是单端口 RAM 模式下的接口示意图。当然了，并非这里所罗列的所有接口都一定要用到，除了一般性的写入数据总线 data[]、地址总线 address[]、写使能信号 wren、写时钟 inclock（可以和 outclock 共用同一个时钟）、读出数据总线 q[]这几个接口外，其他接口都是可选接口，可以根据用户实际应用需求添加或删除。

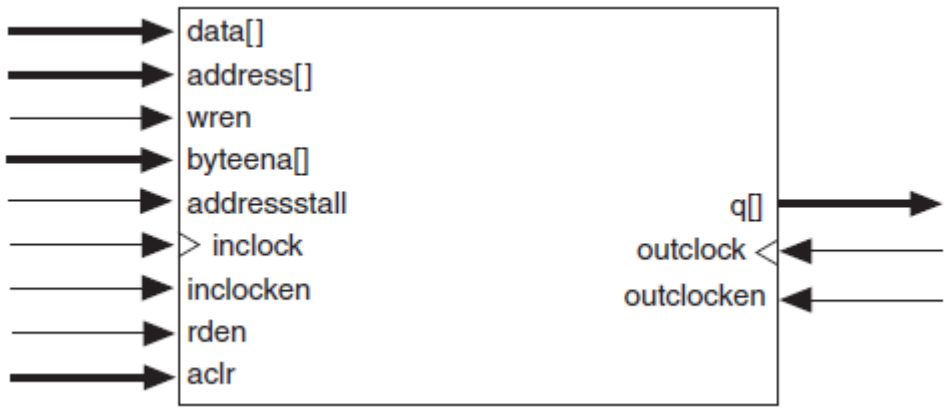



图 2 RAM 接口示意图

**FPGA 片内 RAM 设置与集成**

① Quartus II 工程中，点击菜单“Tools→MegaWizard Plug-In Manager”，创建 IP 核模块。

② 在弹出的选项卡中选择“Creat a new custom megafunction variation”，然后点击“Next”。

③ 选择我们所需要的 IP 核，如图 3 所示进行设置。

- 在“Select a megafunction from the list below”下面选择 IP 核为“Memory Compiler → RAM: 1-PORT”。

- 在“What device family will you be using”后面的下拉栏中选择我们所使用的器件系列为“Cyclone IV E”。
- 在“What type of output file do you want to create?”下面选择语言为“Verilog HDL”。
- 在“What name do you want for the output file?”下面输入工程所在的路径，并且在最后面加上一个名称，这个名称是我们现在正在例化的片内 RAM IP 核的名称，这里我们可以给他起名叫 onchipram_for_ddr，然后点击 Next 进入下一个页面。

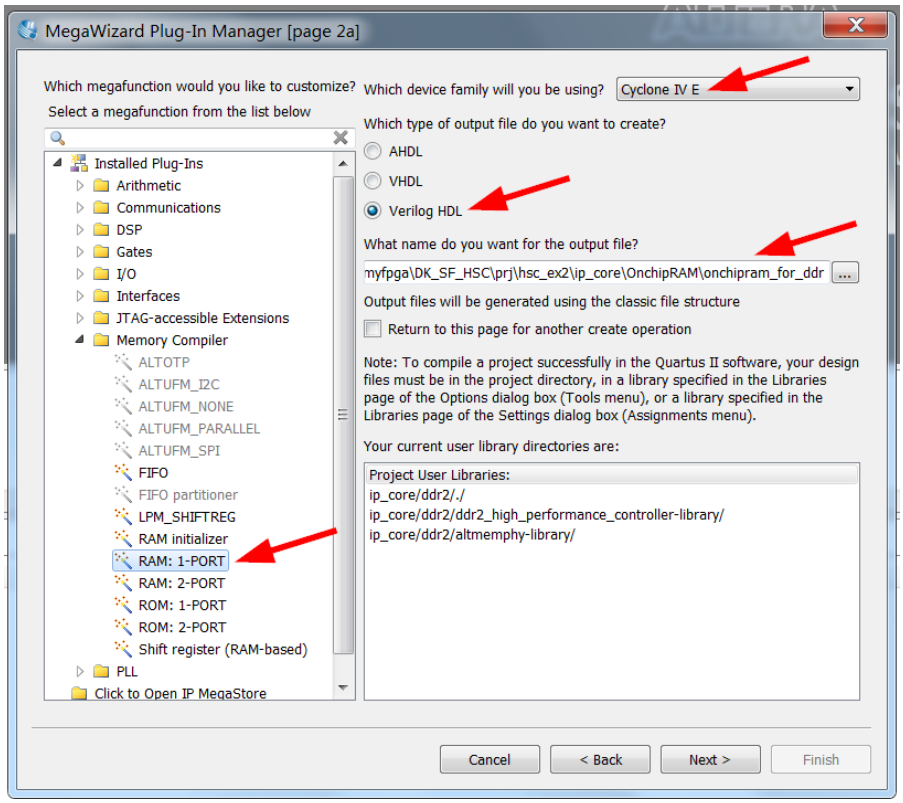



图 3 单口 RAM 创建向导

④ 如图 4 所示，在第一个配置页面“Parameter Settings → Widths/Blk Type/Clks”中，我们需要做如下的设置。

- 在“How wide should the ’q’ output bus be?”后面输入“64”bits，即该片内 RAM 的存储位宽是 64。
- 在“How many 64-bit words of memory?”后面输入“256”words，即该片内 RAM 的存储深度为 256。
- 在“What should the memory block type be?”下面可以选择“Auto”，也可以选择“M9K”，表示我们的 RAM 是使用 FPGA 固有的片内存储器资源还是逻辑（LCs）资源。
- 在“What clocking method would you like to use?”下面选择“Signal clock”，表示该 RAM 的读操作或写操作使用同一个时钟。

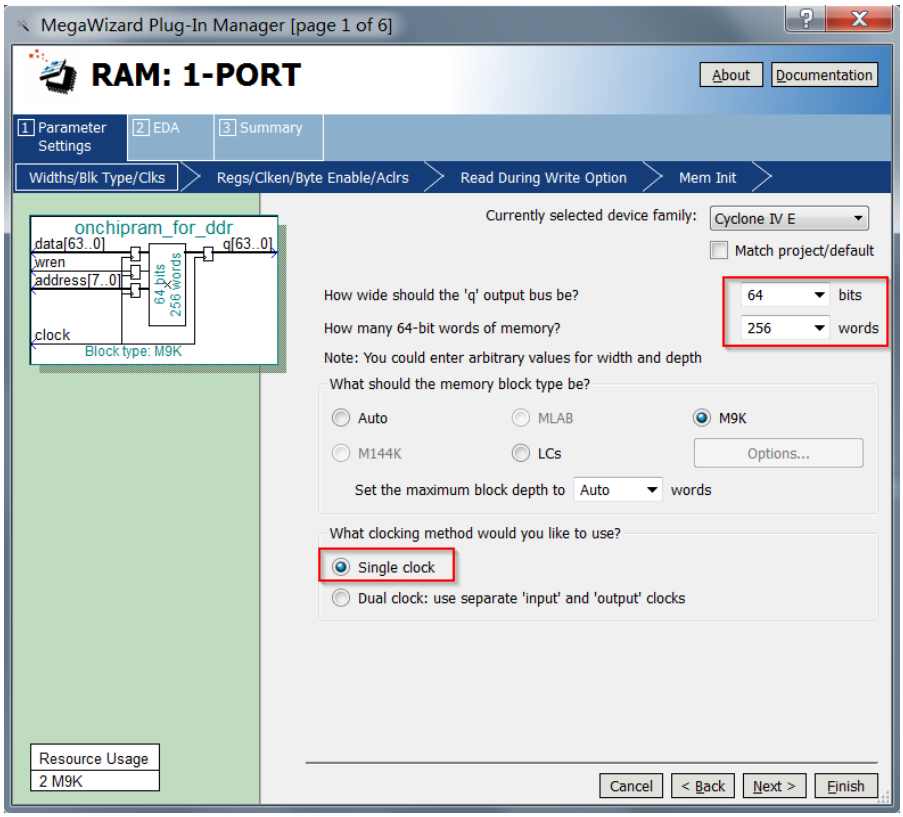



图 4 RAM 参数设置 1

⑤ 如图 5 所示，在第二个配置页面“Parameter Settings →

Regs/Clken/Byte Enable/Aclrs”中，做如下设置。

- 在“Which ports should be registered?”下面选择“‘q’ output port”，表示‘q’信号输出时会用时钟 clock 打一拍，这样更有利于时序收敛。

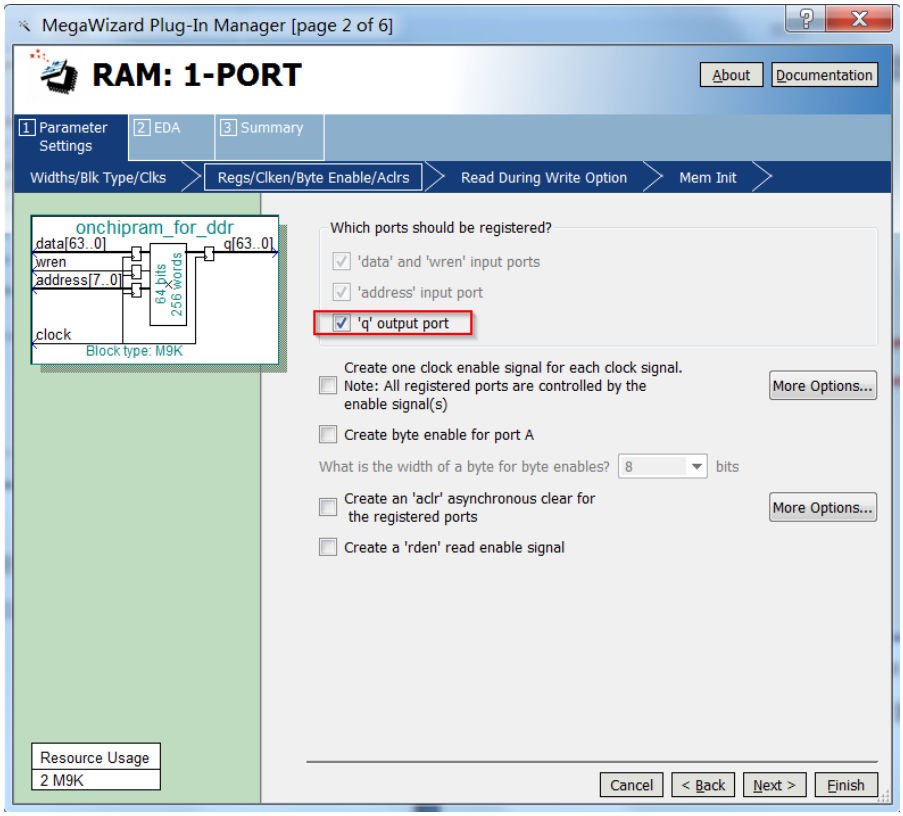



图 5 RAM 参数设置 2

⑥ 第三个配置页面“Parameter Settings → Read During Write Option”可以不用理会，默认设置就好。

⑦ 第四个配置页面“Parameter Settings → Mem Init”则需要特别设置一下，如图 6 所示。

- 在“Do you want to specify the initial content of the memory?”下面勾选“No,leave it blank”。表示我们不需要设置 RAM 的初始化文件，

当然了，在某些特殊应用中，设计者希望有初始化 RAM 文件，那么可以勾上“Yes,use this file for the memory content data”选项，然后在高亮的“File name”后面选择初始化 RAM 文件。

- 勾选上“Allow In-System Memory Content Editor to capture and update content independently of the system clock”，因为我们在板级调试时，将会用到 In-System Memory Content Editor 工具来实时查看 RAM 中的内容变化。“The ‘Instance ID’ of this RAM is:”后面随便输入一个数据，如我们输入“7788”，这个数据类似一个 ID 号，它主要是区分 In-System Memory Content Editor 工具查看不同的 RAM。

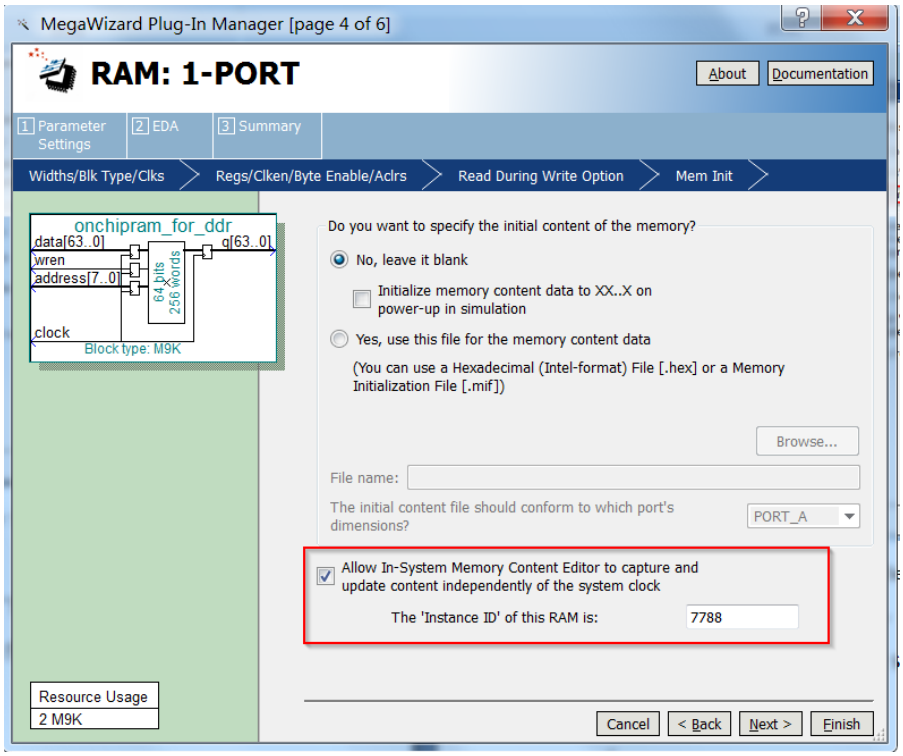



图 6 RAM 参数设置 3

⑧ 最后，在“Summary”页面中，如图 7 所示，勾选上“onchipram_for_ddr_inst.v”文件所在选项。点击“Finish”完成设置。

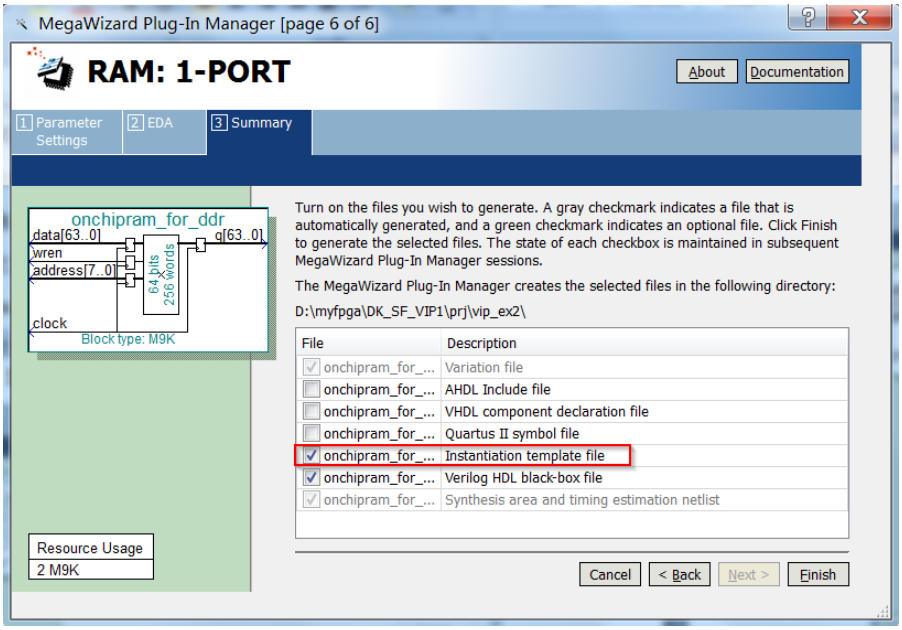



图 7 RAM 配置 Summary 页面

onchipram_for_ddr_inst.v 文件打开如图 8 所示，这里有新建 RAM 的接口例化模板，可以复制到工程顶层源码中重新做接口映射。

onchipram_for_ddr_inst.v

```
1  onchipram_for_ddr   onchipram_for_ddr_inst (
2      .address ( address_sig ),
3      .clock ( clock_sig ),
4      .data ( data_sig ),
5      .wren ( wren_sig ),
6      .q ( q_sig )
7      );
```

图 8 RAM IP 核例化模板

# 3 IP 核配置——DDR2 控制器

**DDR2 IP 核配置**

① 在新建的工程中，点击菜单“Tools→MegaWizard Plug-In Manager”。

② 在弹出的选项卡中选择“Creat a new custom megafunction variation”，然后点击“Next”。

③ 接着接着选择我们所需要的 IP 核，如图 9 所示进行设置。

- 在“Select a megafunction from the list below”下面选择 IP 核为“Interface → External Memory →DDR2 SDRAM→DDR2 SDRAM Controller with ALTMEMPHY v13.1”。
- 在“What device family will you be using”后面的下拉栏中选择我们所使用的器件系列为“Cyclone IV E”。
- 在“What type of output file do you want to create?”下面选择语言为“Verilog”。
- 在“What name do you want for the output file?”下面输入工程所在的路径，并且在最后面加上一个名称，这个名称是我们现在正在例化的 DDR2 Controller IP 核的名称，这里我们可以给他起名叫 ddr2_controller，然后点击 Next 进入下一个页面。

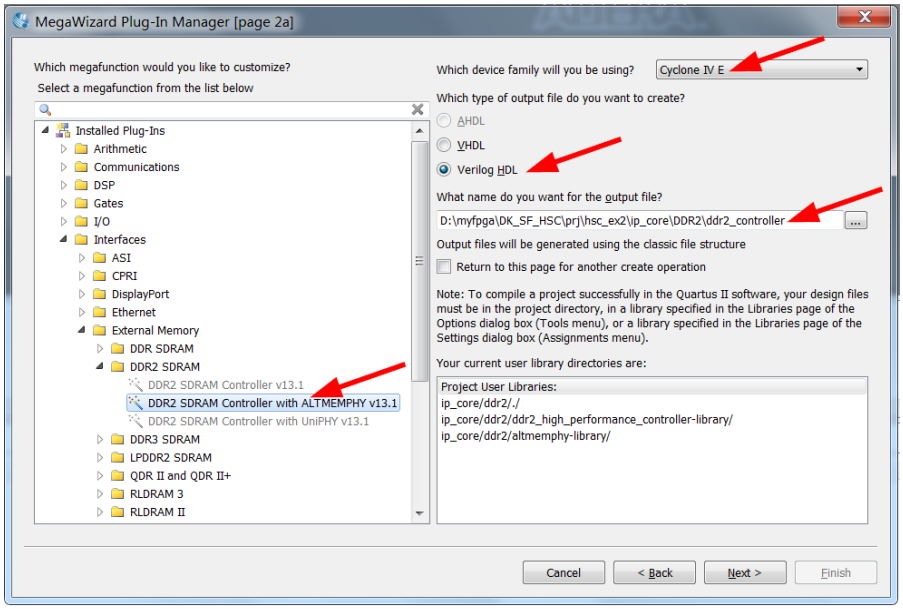



图 9 DDR2 IP 核创建向导

④ 进入配置页面后，首先是“Parameter Settings →Memory Settings”选项卡，如图 10 进行设置。

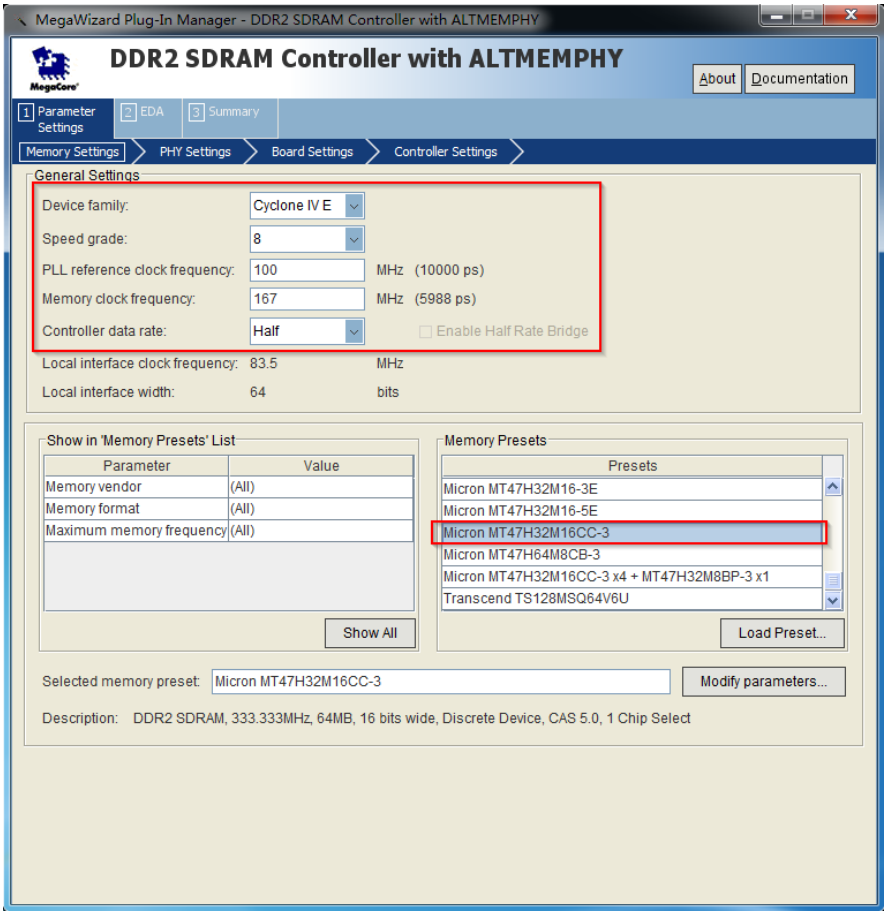



图 10 DDR2 存储器参数配置

- “Device family”后面的下拉框中选择工程所使用的器件系列，本实例工程选择“Cyclone IV E”。
- “Speed grade”后面的下拉框中选择工程所使用器件的速度等级，根据实际器件的情况，我们选择“8”。
- “PLL reference clock frequency”后面设置输入到该 IP 核的 PLL 模块的时钟频率，本实例工程输入时钟频率为“100MHz”。
- “Memory clock frequency”后面输入 DDR2 实际操作的差分时钟频率，我们设置为“167MHz”。
- “Controller data rate”为该 IP 核与 FPGA 内部逻辑接口之间的频

率，我们设定为“Half”，表示这个频率为 DDR2 时钟频率 167MHz 的一半。

- “Memory Presets”下面有很多可选的 DDR2 型号，我们根据实际使用型号选择“Micron MT47H32M16CC-3”。

关于“Memory Presets”部分，如果我们找不到自己使用的 DDR2 芯片信号，那么可以现在“Presets”中先选择一个和实际使用型号参数相近的型号，然后点击“Modify parameters…”按钮对一些具体的参数进行修改，以匹配实际使用的 DDR2 型号。如图 11 所示，打开的“Preset Editor”中几乎可以更改所有的 DDR2 相关参数。这些参数的具体含义可以参考 Altera 官方文档“ug_altmemphy.pdf”。

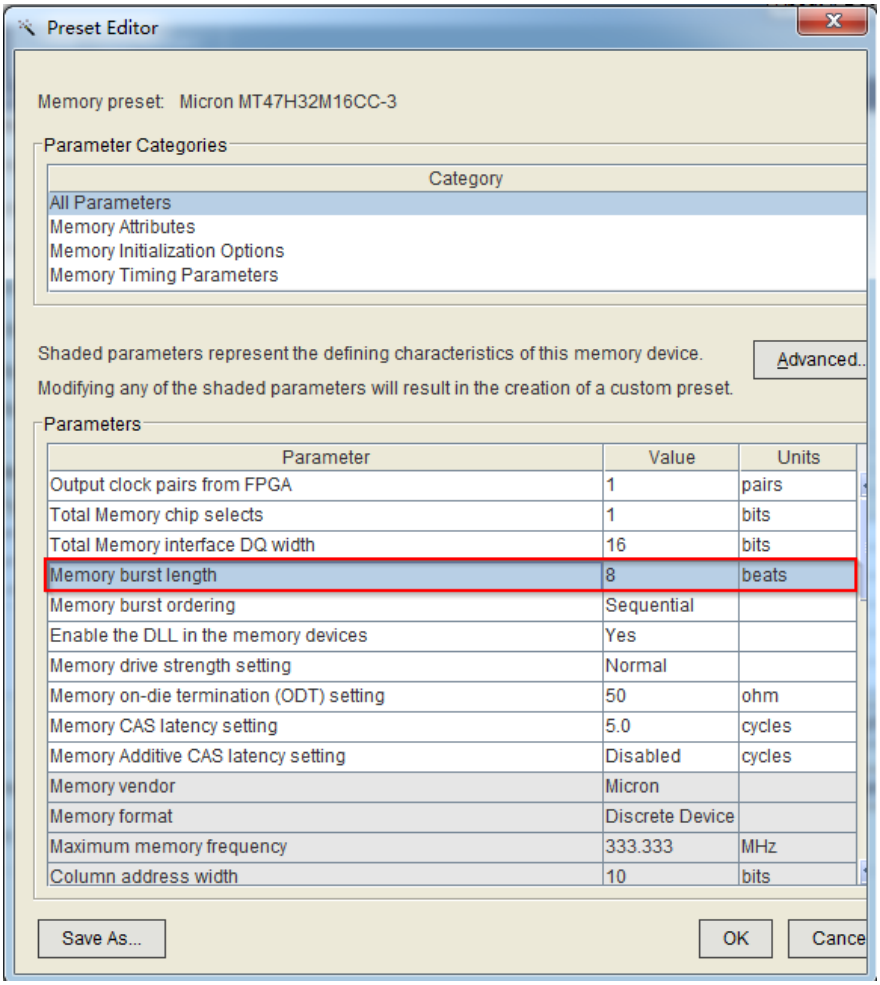



图 11 DDR2 自定义参数设置

- “Preset Editor”选项卡中需要更改一个设置，即 Parameter 为“Memory burst length”行对应的“Value”更改为“8”。这是由于我们使用了 Half 模式的 Controller data rate，若不更改此 burst 值，则 IP 核生成时将会报错。

⑤ “Parameter Settings →PHY Settings”选项卡、“Parameter Settings →Board Settings”选项卡以及“Parameter Settings →Controller Settings”选项卡，通常都不需要额外设置，使用默认设置就可以。

⑥ “EDA”选项卡和仿真有关，可以选择是否生成响应的网表文件。

⑦ “Summary”选项卡可以选择是否生成一些 IP 核相关的文件，点击“Finish”就可以生成 IP 核。如图 12 所示，至此 IP 核以及生成完毕。

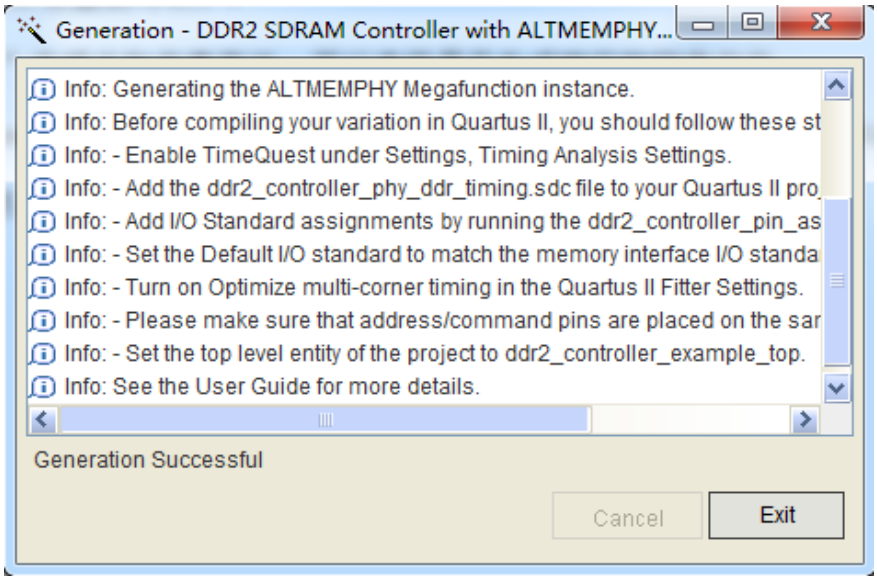



图 12 DDR2 IP 核生成

**DDR2 IP 核接口描述**

如图 13 所示，这是 DDR2 IP 核与外部接口的功能框图。

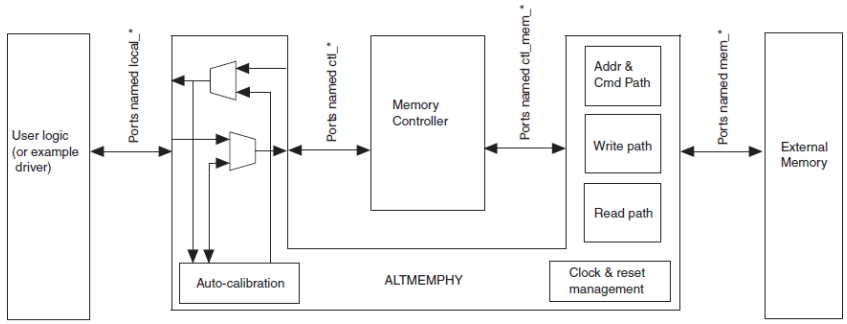



图 13 DDR IP 核功能框图

- 图左侧为用户逻辑（User logic），它与 DDR2 IP 核的接口通常命名为“local_*”；图右侧为 FPGA 外部的 DDR2 芯片，它与 DDR2 IP 核的接口通常命名为“mem_*”。

- DDR2 IP 核内部分两个部分，即图示的“ALTMEMPHY”和“存储控制器（Memory Controller）”，正如图中所示，它们各有分工。存储控制器产生 DDR2 芯片实际读写操作需要的时序；ALTMEMPHY 有两部分功能，一个功能是对 DDR2 做自动校正（Aoto-Calibration），另一个功能是实现 DDR2 所需的物理接口。
- DDR2 的自动校正是在初始化阶段进行的，此时 ALTMEMPHY 断开用户逻辑和存储控制器之间的接口，ALTMEMPHY 产生存储控制器所需的 DDR2 读写控制，直到校正完成。在初始化过后，ALTMEMPHY 将不再需要控制存储控制器，而是一直保持用户逻辑和存储控制器的连通。
- 另外，图中未明确示意，实际上这个 DDR2 IP 核还包括了一个 PLL，用于时钟的管理。

现在我们来看 ddr2_controller 模块例化的接口。这里可以分为三大类，第一类为系统类接口，主要是一些系统或 PLL 的复位、时钟等接口；第二类为带“local_*”的接口，是 DDR2 IP 核与用户逻辑间的接口；第三类为带“mem_*”的接口，是 DDR2 IP 核与 FPGA 外部 DDR2 芯片的接口。

对于第一类接口，功能描述如表 1 所示。

表 1 DDR2 IP 核系统接口列表

| 信号名 | 方向 | 功能描述 |
| --- | --- | --- |
| global_reset_n | Input | IP 核的全局异步复位信号，低电平有效。该信号有效时，将使得 ALTMEMPHY（包括 PLL）都进入复位状态。 |
| pll_ref_clk | Input | PLL 的输入参考时钟信号。 |

| | | |
|---|---|---|
| soft_reset_n | Input | IP 核的全局异步复位信号，低电平有效。该信号只能复位 ALTMEMPHY，而不能复位 PLL。 |
| aux_half_rate_clk | Output | phy_clk 时钟信号的引出，时钟频率与 phy_clk 一样，可用于用户逻辑使用。 |
| aux_full_rate_clk | Output | phy_clk 时钟信号的引出，时钟频率是 phy_clk 的两倍，可用于用户逻辑使用。 |
| reset_request_n | Output | 复位输出，用于指示用户逻辑 DDR2 IP 核的内部 PLL 输出 locked 还未完成。 |
| phy_clk | Output | ALTMEMPHY 产生供用户逻辑使用的半速率时钟信号。所有输入和输出到 ALTMEMPHY 的用户逻辑接口信号，都与此时钟同步。 |
| reset_phy_clk_n | Output | 与 phy_clock 时钟域相关的复位信号，低电平有效。可用此时钟复位所有 DDR IP 核和用户逻辑接口相关的信号。 |

注：方向是相对 DDR2 IP 核而言的。

对于第二类带“local_*”的用户逻辑接口，功能描述如表 2 所示。

表 2 DDR2 IP 核本地接口列表

| 信号名 | 方向 | 功能描述 |
|---|---|---|
| local_address[22:0] | Input | 本地逻辑对 DDR2 IP 核的数据读出或写入地址。 |
| local_write_req | Input | 本地逻辑对 DDR2 IP 核的数据写入请求信号，高电平有效。 |
| local_read_req | Input | 本地逻辑对 DDR2 IP 核的数据读出请求信号，高电平有效。 |

| | | |
|---|---|---|
| local_burstbegin | Input | 本地逻辑对 DDR2 IP 核的数据突发传输起始标志信号。多个数据传输时，该信号在 local_write_req 或 local_read_req 信号拉高的第一个时钟周期时保持高电平，用于指示传输的起始。 |
| local_wdata[63:0] | Input | 本地逻辑写入到 DDR IP 核的数据总线信号，每次写入 4 个 16bit 数据。 |
| local_be[7:0] | Input | 读写数据字节使能标志信号。Local_be 的每个位对应 local_wdata 或 local_rdata 的 8bit 数据是否有效。 |
| local_size[2:0] | Input | 突发传输的有效数据数量，即传输多少个 local_wdata 或 local_rdata 数据。 |
| local_ready | Ouput | DDR2 IP 核输出的当前读写请求已经被接收的指示信号，高电平有效。 |
| local_rdata[63:0] | Output | DDR IP 核输出的本地逻辑读出数据总线信号，每次读出 4 个 16bit 数据。 |
| local_rdata_valid | Output | local_rdata 数据总线输出有效信号，高电平有效。 |
| local_refresh_ack | Input | 本地逻辑输入到 DDR IP 核的刷新请求信号。 |
| local_init_done | Output | ALTMEMPHY 完成 DDR 存储控制器的自动校准操作，拉高该信号。该信号可以作为用户逻辑的复位信号。 |

注：方向是相对 DDR2 IP 核而言的。

第三类是带“mem_*”的 DDR2 芯片接口，前面已经给出基本的功能描述，这里不再赘述。DDR2 的在我们设计中例化的接口映射代码如下所示。

```
////////////////////////////////////////////////////
    //DDR2 controller and phy IP core

ddr2_controller     ddr2_controller_inst (
            .local_address(local_address),
            .local_write_req(local_write_req),
            .local_read_req(local_read_req),
            .local_burstbegin(local_read_req | local_write_req),
            .local_wdata(local_wdata),
            .local_be(8'hff),
            .local_size(3'd1),
            .global_reset_n(sys_rst_n),
            .pll_ref_clk(clk_100m),
            .soft_reset_n(1'b1),
            .local_ready(local_ready),
            .local_rdata(local_rdata),
            .local_rdata_valid(local_rdata_valid),
            .local_refresh_ack( ),
            .local_init_done(local_init_done),
            .reset_phy_clk_n(reset_phy_clk_n),
            .mem_odt(mem_odt),
            .mem_cs_n(mem_cs_n),
            .mem_cke(mem_cke),
            .mem_addr(mem_addr),
            .mem_ba(mem_ba),
            .mem_ras_n(mem_ras_n),
```

```
        .mem_cas_n(mem_cas_n),
        .mem_we_n(mem_we_n),
        .mem_dm(mem_dm),
        .phy_clk(phy_clk),
        .aux_full_rate_clk( ),
        .aux_half_rate_clk( ),
        .reset_request_n( ),
        .mem_clk(mem_clk),
        .mem_clk_n(mem_clk_n),
        .mem_dq(mem_dq),
        .mem_dqs(mem_dqs)
    );
```

**DDR2 IP 核接口时序**

用户逻辑和 DDR2 IP 核之间的接口并不是什么新发明的特殊接口，不过是 Avalon-MM 总线而已。有人说这个美眉（Memory-Map）会不会太慢了，关键时刻耽误事？非也，MM 总线的 burst 模式也可以流水线式连续传输数据，丝毫不逊色于 ST(stream)传输方式。

这里我们可以简单了解一下带“local_*”的 Avalon-MM 总线 burst 模式传输协议的使用方法。

可以比较简单山寨的理解前面已经给出的带“local_*”的 Avalon-MM 信号接口：

- local_size 为 burst 读写的最大数据数量。通常 IP 核内部有 FIFO 用于支持这样的连续数据读写，在 Megafunction 中设定好的最大数据数量是 Avl_size 的上限值。
- local_be 为 byte enable 信号，用于使能或说是屏蔽读写数据的各个高低字节。

- local_ready 为总线当前状态指示。这里高电平表示 ready，此时的 local_read_req 和 local_write_req 能够被锁存。
- local_burstbegin 为突发传输起始标志位。它不受 local_ready 的影响，在发起一次读或写操作的第一个时钟周期，只需保持一个时钟周期的 local_burstbegin 为高电平状态，并且不用管此时的 local_ready 状态如何。
- local_addr 为读写共用的总线地址，位宽由 DDR2 的存储总量和总线上读写数据的位宽来决定。如 1Gbit 的 DDR2，外部芯片的数据位宽为 16bit，Avalon-MM 读写的数据位宽 64bit，那么它的地址不是以 16bit 位宽来计算的，而是以 64bit 位宽来计算的，即 16M（24 位）。
- local_read_req 为读请求，配合地址 local_addr 和突发传输起始标志位 local_burstbegin 发起一次 burst 读操作。在 local_burstbegin 拉高后，只需要确保在同一个时钟周期或其后第一次 local_ready 有效的时钟周期拉高一次 local_read_req 信号即可。
- local_rdata_valid 为读出数据的有效标志位。IP 核在收到 burst 读请求（local_read_req）后的若干个时钟周期开始连续送出数据（数据可能分多次连续送出），该信号和读出数据配合，高电平表示当前读出数据有效。
- local_rdata 为读出数据。和 local_rdata_valid 配合送给用户逻辑。
- local_write_req 为写请求信号。若发起一次 n 个数据写入的 burst 传输，第一个传输时钟周期首先拉高 local_burstbegin 以及 local_write_req，且 local_write_req 必须保持到 n 个数据写入完成。只有在 local_ready 有效时，当前的 local_write_req、local_addr 和 local_wdata 才是有效的。

- local_wdata 为写入数据。

接着我们再用几张时序图来解析前面的接口。

① 如图 14 所示，正常 4 个数据的 burst 读操作。默认情况下，local_addr 为读出的首个数据对应的地址，随后将读出递增地址的数据。

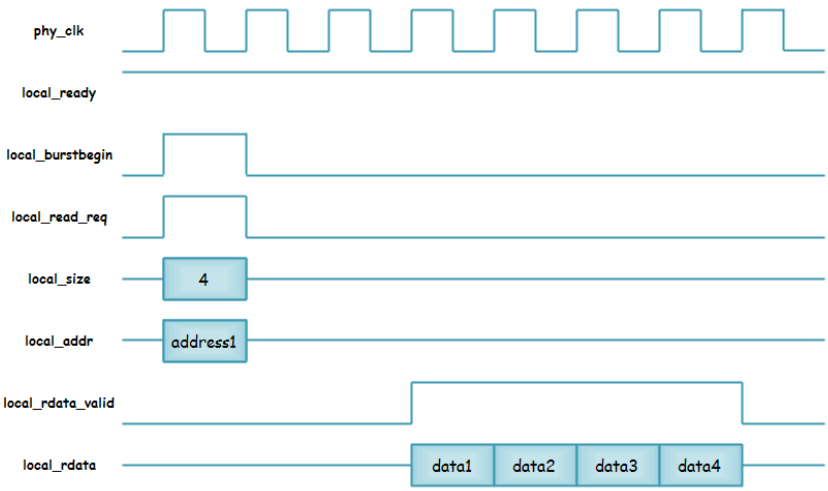



图 14 四个数据读取操作时序

② 如图 15 所示，遇到 local_ready 拉低的读操作，必须保持 local_read_req、local_size 和 local_addr 到 local_ready 拉高为止。

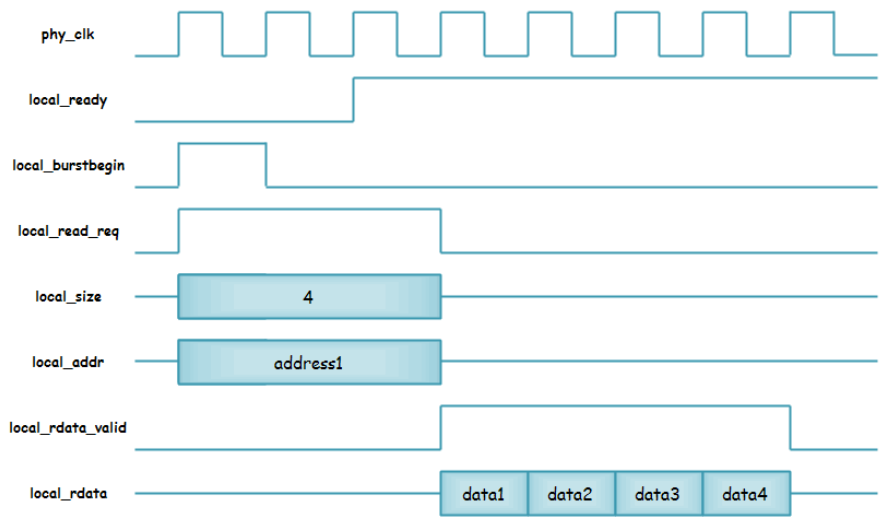



图 15 读忙时的四个数据读取时序

③ 如图 16 所示，正常 4 个数据的 burst 写操作。默认情况下，local_addr 设定的是写入的首个数据对应的地址，随后每次写入数据后地址自动递增。

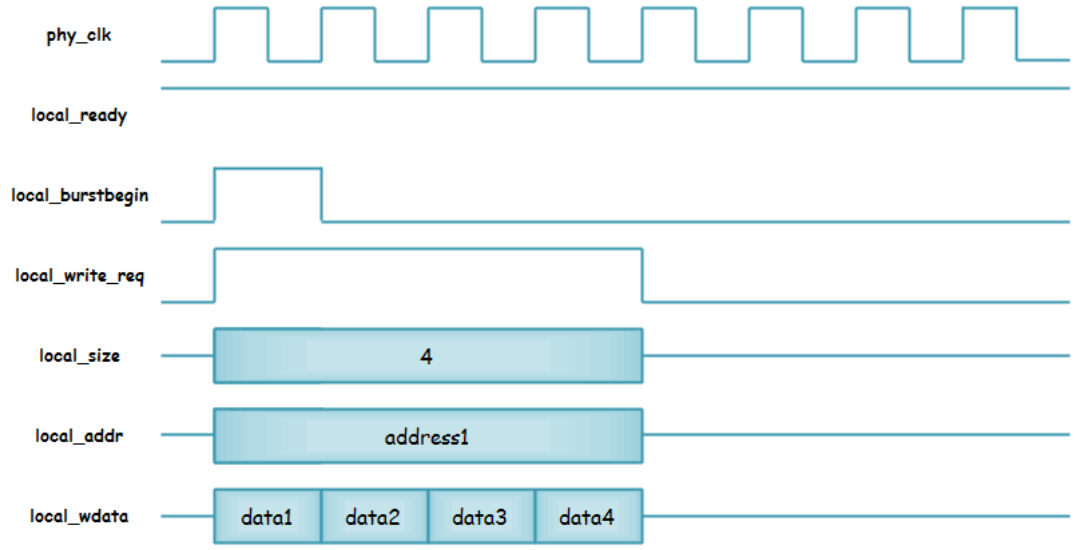



图 16 四个数据写取操作时序

④ 如图 17 所示，遇到 local_ready 拉低的 4 个数据的 burst 写操作。Local_ready 拉高时，local_write_req、local_addr 和 local_wdata 所对应的地址和数据才是有效的。

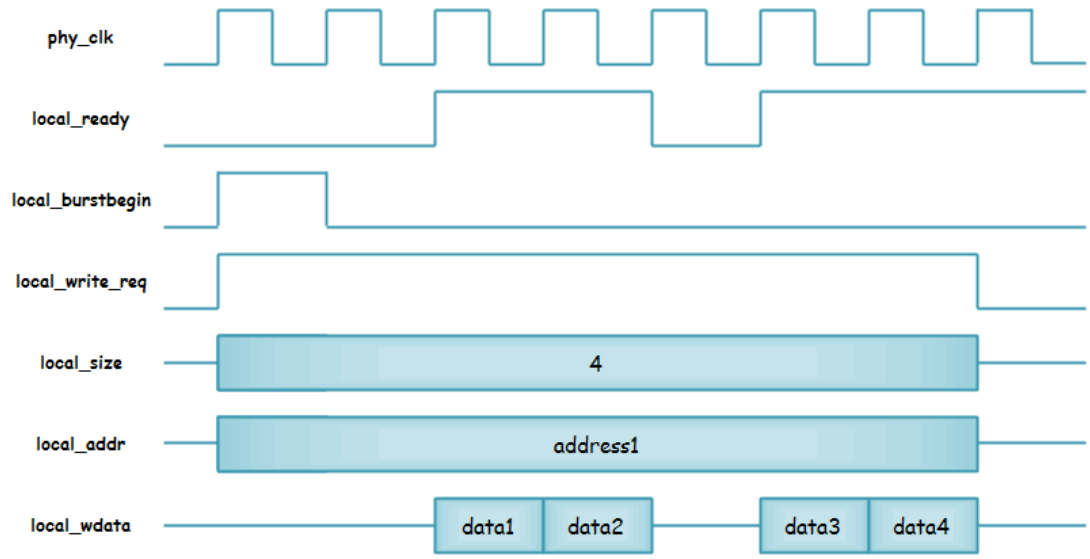



图 17 写忙时的四个数据写时序

# 4 DDR2 引脚电平设置

DDR2 的标准 IO 电平不是我们接触最多的 3.3V TTL，而是 1.8V 的“SSTL-18 Class I”标准，那么这个电平标准，在 FPGA 中如何设置？

我们可以先点击 Quartus II 的菜单“Assignments → Pin Planner”，打开 Pin Planner 如图 18 所示。这里可以做 FPGA 信号和实际芯片 die 引脚的映射，包括引脚的电平标准、电流强度、片内上下拉、偏斜率等等参数都可以在这里配置。这种灵活的引脚可配置性，其实也是 FPGA 可编程灵活性的一大体现。

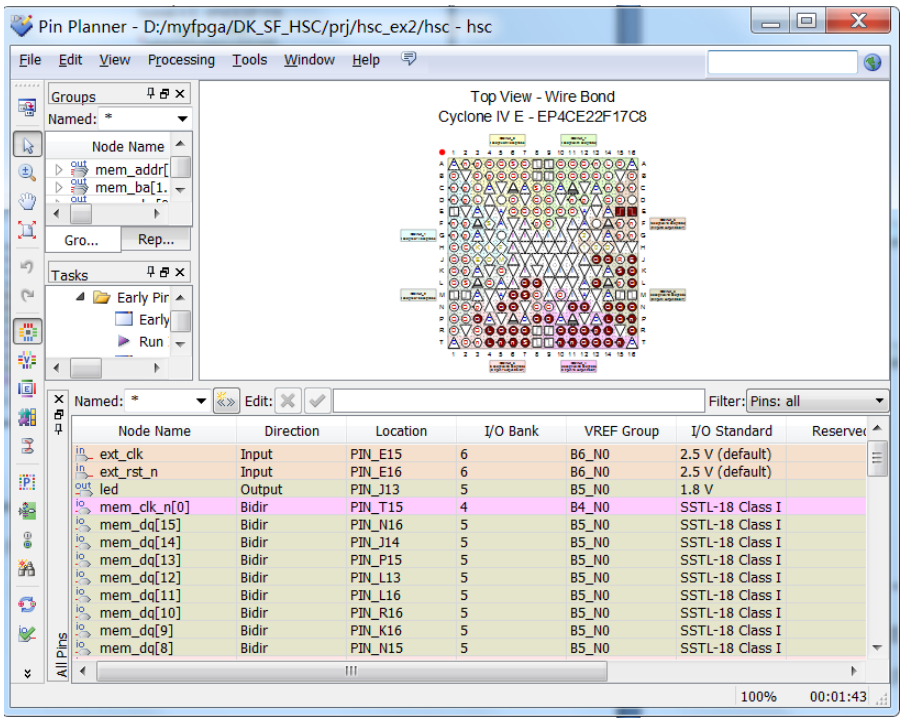



图 18 Pin Planner 界面

在这个 Pin Planner 界面中，我们可以先看看“Top View – Wire Bond”这个视图。如图 19 所示，在芯片引脚视图的外侧，有 8 个小矩形框分别标示了 IO BANK 号，图中对应不同的颜色表示了不同的 IO BANK。

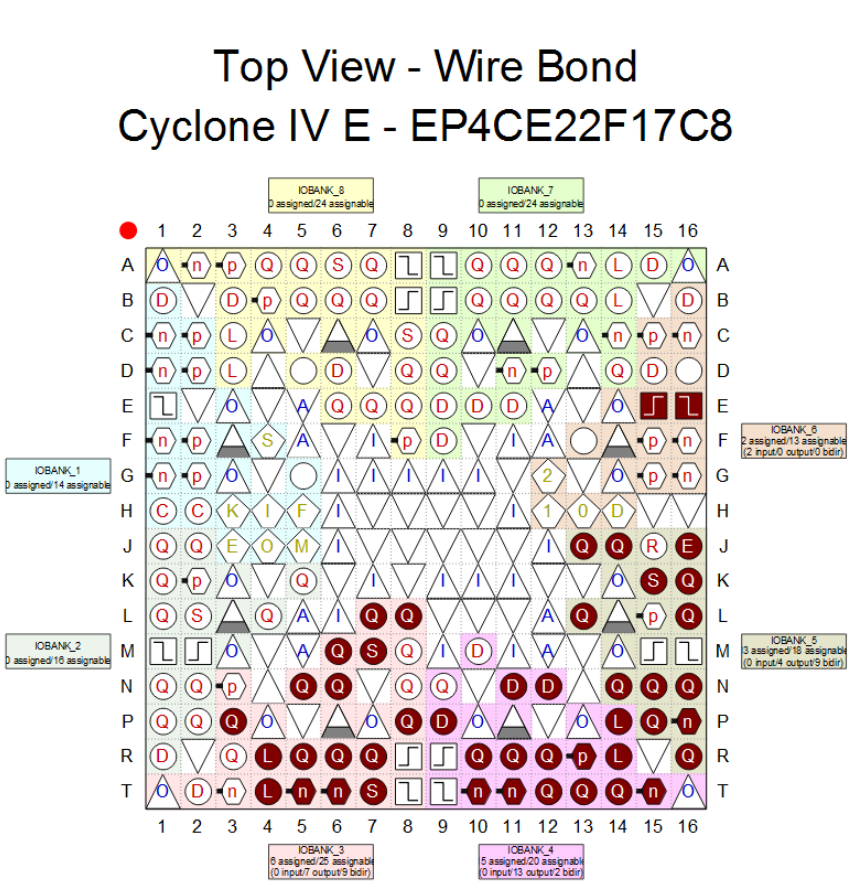



图 18 引脚视图

若是“Top View – Wire Bond”没有出现不同的 IO BANK 标示，那么我们可以在这个视图的空白处点击右键，如图 19 所示，然后单击“Show I/O Banks”选项打开。

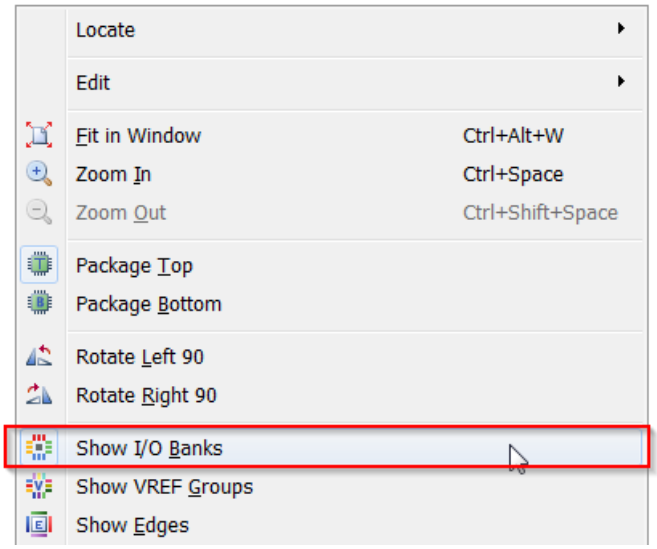



图 19 引脚视图的右键菜单

FPGA 的电平标准，通常是以 IO BANK 为单位进行划分的，对于一颗 FPGA 芯片，它通常会根据实际情况，将一些就近的引脚划分在同一个 IO BANK 中，并且一颗 FPGA 芯片总会或多或少划分出几个 IO BANK 来。不同的 IO BANK 可以有不同的 IO 电平标准，但是在同一个 IO BANK 内的引脚，它们的 IO 电平标准一定是一样的。

回到我们的应用中来，在我们的核心板上，大都是的 IO BANK 的电平都是 3.3V 的，而 DDR2 引脚所使用的 IO BANK 则是 1.8V。如图 20 所示，在原理图设计上，VCCIO 电压供 1.8V 的 BANK3、BANK4 和 BANK5，都连接了 DDR2 的引脚。

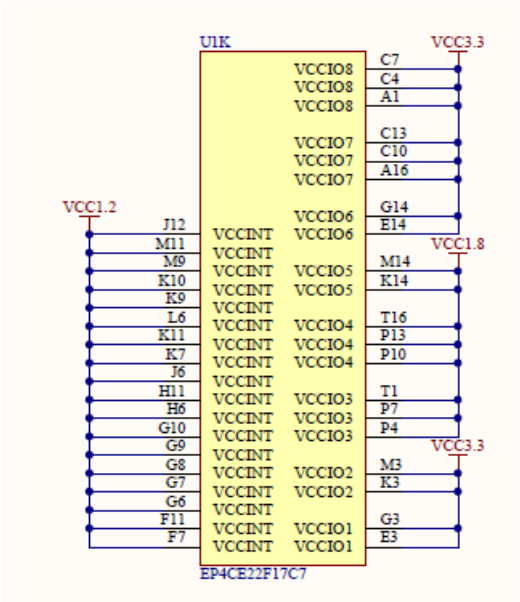



图 20 FPGA 供电电路

基于此，我们在 FPGA 内部的引脚电平标准的设定上，我们也需要做相应的设定。如图 21 所示，我们在“I/O Standard”一列中，对应 DDR2 引脚，选择它们的电平标准为“SSTL-18 Class I”。

| Node Name | Direction | Location | I/O Bank | VREF Group | I/O Standard | Reserved |
|---|---|---|---|---|---|---|
| altera_reserved_tck | Input | | | | 2.5 V (default) | |
| altera_reserved_tdi | Input | | | | 2.5 V (default) | |
| altera_reserved_tdo | Output | | | | 2.5 V (default) | |
| altera_reserved_tms | Input | | | | 2.5 V (default) | |
| ext_clk | Input | PIN_E15 | 6 | B6_N0 | 2.5 V (default) | |
| ext_rst_n | Input | PIN_E16 | 6 | B6_N0 | 2.5 V (default) | |
| led | Output | PIN_J13 | 5 | B5_N0 | 1.8 V | |
| mem_addr[12] | Output | PIN_T4 | 3 | B3_N0 | SSTL-18 Class I | |
| mem_addr[11] | Output | PIN_R4 | 3 | B3_N0 | | |
| mem_addr[10] | Output | PIN_T7 | 3 | B3_N0 | | |
| mem_addr[9] | Output | PIN_J16 | 5 | B5_N0 | | |
| mem_addr[8] | Output | PIN_R12 | 4 | B4_N0 | | |
| mem_addr[7] | Output | PIN_T5 | 3 | B3_N0 | | |
| mem_addr[6] | Output | PIN_P8 | 3 | B3_N0 | | |
| mem_addr[5] | Output | PIN_P16 | 5 | B5_N0 | | |
| mem_addr[4] | Output | PIN_T12 | 4 | B4_N0 | | |
| mem_addr[3] | Output | PIN_T6 | 3 | B3_N0 | | |
| mem_addr[2] | Output | PIN_N11 | 4 | B4_N0 | | |
| mem_addr[1] | Output | PIN_R14 | 4 | B4_N0 | | |
| mem_addr[0] | Output | PIN_R11 | 4 | B4_N0 | | |
| mem_ba[1] | Output | PIN_R13 | 4 | B4_N0 | SSTL-18 Class I | |

Differential 2.5-V SSTL Class II
LVDS_E_3R
PPDS_E_3R
RSDS_E_1R
RSDS_E_3R
SSTL-2 Class I
SSTL-2 Class II
SSTL-18 Class I
SSTL-18 Class II
mini-LVDS_E_3R

图 21 DDR2 引脚电平设置

# 5 Verilog 代码解析

本实例有 8 个模块，3 个层级。其层次结构如图 22 所示。

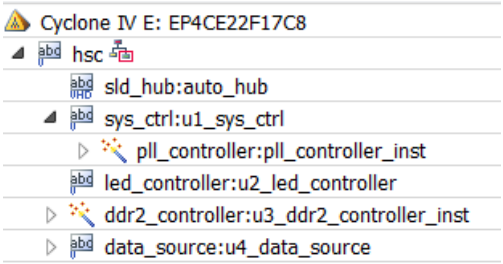



图 22 代码层次图

- hsc.v 是顶层模块，其下例化了 5 个子模块，即 sld_hub.vhd 模块、ddr2_controller.v 模块、data_source.v 模块、onchipram_for_ddr.v 模块、led_controller.v 模块和 sys_ctrl.v 模块。该模块仅仅用于子模块间的接口连接，以及和 FPGA 外部的接口定义，该模块中未作任何的逻辑处理。
- sld_hub.vhd 二级子模块是一个软核 IP，编译时自动产生，不是我们在顶层代码 hsc.v 中例化的。它用于我们 FPGA 的片内 RAM 和 Quartus II 之间通过 JTAG 进行数据交互。换句话说，这个模块其实是一个 JTAG 协议的接口实现模块。该模块是 VHDL 编写，并且内部代码对用户是一个“黑盒”，我们只要会使用它就可以，无需关心它的代码内容。
- ddr2_controller.v 二级子模块也是一个软核 IP，实现 DDR2 的时序控制功能，并且通过一个简单的 Avalon 接口实现 DDR2 和 FPGA 逻辑之间的读写数据传输。其实该模块下还有多个子模块，但由于只是一个 IP 核，内部代码不直接开放，所以我们姑且认为它是我们工程中的一个模块。

- data_source.v 二级子模块是用户逻辑，我们用它定时产生一组写入 DDR2 地址 0-1023 的连续的有规律的数据；并且我们也会定时的读出 DDR2 地址 0-1023 地址的数据，将这些数据再写入到 FPGA 例化的片内 RAM 里。
- onchipram_for_ddr.v 三级子模块是硬核 IP，它例化实现了 FPGA 的一块片内 RAM 空间。
- sys_ctrl.v 二级子模块中例化了 PLL 模块，并且对输入 PLL 的复位信号以及 PLL 锁定后的复位信号进行“异步复位，同步释放”的处理，确保系统的复位信号稳定可靠。
- pll_controller.v 三级子模块为 FPGA 器件特有的 IP 硬核模块，其主要功能是产生多个特定输入时钟的分频、倍频、相位调整后的输出时钟信号。
- led_controller.v 二级子模块进行 24 位计数器的循环计数，产生分频信号用于实现 LED 指示灯的闪烁。

**hsc.v 模块代码解析**

顶层模块的代码如下。

```
module hsc(
ext_clk,ext_rst_n,mem_odt,mem_cs_n,mem_cke,mem_addr,
mem_ba,mem_ras_n,mem_cas_n,mem_we_n,mem_dm,mem_clk,mem_clk_n,me
m_dq,mem_dqs,led
        );
    //外部输入时钟和复位接口
input ext_clk;       //外部 25MHz 输入时钟
input ext_rst_n;     //外部低电平复位信号输入
    //LED 指示灯接口
output led;      //用于测试的 LED 指示灯
```

```
    //DDR2 芯片接口
output  [1:0]   mem_ba;
output  [12:0]  mem_addr;
output  [0:0]   mem_cke;
output  [0:0]   mem_cs_n;
output      mem_ras_n;
output      mem_cas_n;
output      mem_we_n;
output  [1:0]   mem_dm;
output  [0:0]   mem_odt;
inout   [0:0]   mem_clk;
inout   [0:0]   mem_clk_n;
inout   [15:0]  mem_dq;
inout   [1:0]   mem_dqs;

////////////////////////////////////////////////////
//系统内部时钟和复位产生模块例化
    //PLL 输出复位和时钟，用于 FPGA 内部系统
wire sys_rst_n; //系统复位信号，低电平有效
wire clk_25m;       //PLL 输出 25MHz
wire clk_33m;       //PLL 输出 33MHz
wire clk_50m;       //PLL 输出 50MHz
wire clk_65m;       //PLL 输出 65MHz
wire clk_100m;      //PLL 输出 100MHz

sys_ctrl    uut_sys_ctrl(
                .ext_clk(ext_clk),
                .ext_rst_n(ext_rst_n),
                .sys_rst_n(sys_rst_n),
                .clk_25m(clk_25m),
                .clk_33m(clk_33m),
                .clk_50m(clk_50m),
```

```
                .clk_65m(clk_65m),
                .clk_100m(clk_100m)
            );

////////////////////////////////////////////////////
//LED 闪烁逻辑产生模块例化

led_controller      uut_led_controller(
                        .clk(clk_25m),
                        .rst_n(sys_rst_n),
                        .led(led)
                    );

////////////////////////////////////////////////////
//local DDR2 SDRAM interface
    //Clock and Reset Signals for DDR2/DDR SDRAM
wire        phy_clk;
wire        reset_phy_clk_n;
    //Local Interface Signals for DDR2 and DDR SDRAM
wire    [22:0]  local_address;
wire        local_write_req;
wire        local_read_req;
wire    [63:0]  local_wdata;
wire        local_ready;
wire    [63:0]  local_rdata;
wire        local_rdata_valid;
wire        local_init_done;

////////////////////////////////////////////////////
    //DDR2 controller and phy IP core
    //工作于 8*16Bits 模式
ddr2_controller     ddr2_controller_inst (
```

```
        .local_address(local_address),
        .local_write_req(local_write_req),
        .local_read_req(local_read_req),
        .local_burstbegin(local_read_req | local_write_req),
        .local_wdata(local_wdata),
        .local_be(8'hff),
        .local_size(3'd1),
        .global_reset_n(sys_rst_n),
        .pll_ref_clk(clk_100m),
        .soft_reset_n(1'b1),
        .local_ready(local_ready),
        .local_rdata(local_rdata),
        .local_rdata_valid(local_rdata_valid),
        .local_refresh_ack(),
        .local_init_done(local_init_done),
        .reset_phy_clk_n(reset_phy_clk_n),
        .mem_odt(mem_odt),
        .mem_cs_n(mem_cs_n),
        .mem_cke(mem_cke),
        .mem_addr(mem_addr),
        .mem_ba(mem_ba),
        .mem_ras_n(mem_ras_n),
        .mem_cas_n(mem_cas_n),
        .mem_we_n(mem_we_n),
        .mem_dm(mem_dm),
        .phy_clk(phy_clk),
        .aux_full_rate_clk(),
        .aux_half_rate_clk(),
        .reset_request_n(),
        .mem_clk(mem_clk),
        .mem_clk_n(mem_clk_n),
        .mem_dq(mem_dq),
```

```
        .mem_dqs(mem_dqs)
    );

////////////////////////////////////////////////////
    //产生数据源，用于测试 DDR2 的读写
data_source     uut_data_source(
                    .clk(phy_clk),  //75MHz
                    .rst_n(reset_phy_clk_n),
                    .local_address(local_address),
                    .local_write_req(local_write_req),
                    .local_read_req(local_read_req),
                    .local_wdata(local_wdata),
                    .local_ready(local_ready),
                    .local_rdata(local_rdata),
                    .local_rdata_valid(local_rdata_valid),
                    .local_init_done(local_init_done)
                );

endmodule
```

以下是 DDR2 和 FPGA 接口的信号，包括数据总线、地址总线、控制总线和差分时钟等。

```
    //DDR2 芯片接口
output  [1:0]   mem_ba;
output  [12:0]  mem_addr;
output  [0:0]   mem_cke;
output  [0:0]   mem_cs_n;
output      mem_ras_n;
output      mem_cas_n;
output      mem_we_n;
output  [1:0]   mem_dm;
output  [0:0]   mem_odt;
```

```
inout   [0:0]   mem_clk;
inout   [0:0]   mem_clk_n;
inout   [15:0]  mem_dq;
inout   [1:0]   mem_dqs;
```

我们查看 DDR2 芯片的 spec，可以看到如图 23 所示的内部结构和外部接口框图，这些外部接口基本都是需要连接到 FPGA 进行数据和地址的传输。

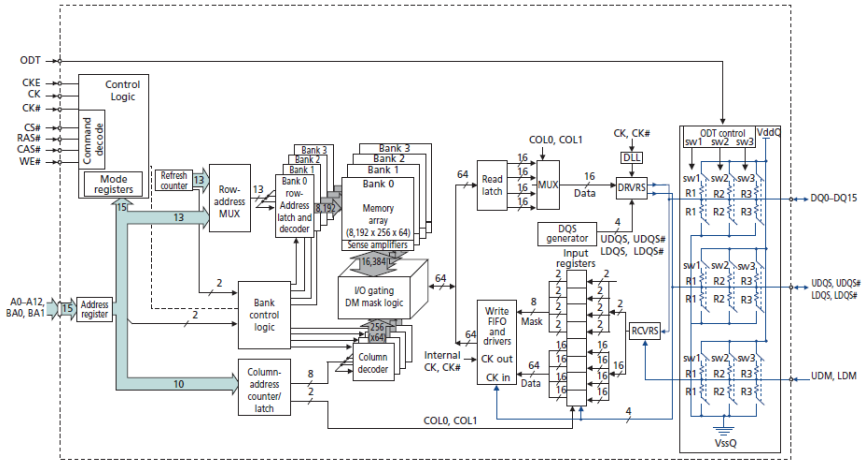



图 23 DDR2 内部结构与外部接口

对于这些信号，我们也可以简单的通过表 3 的描述来了解它们的基本功能和用途。其中方向是相对 FPGA 而言的。

表 3 DDR2 芯片引脚定义

| 信号名 | 方向 | 功能描述 |
|---|---|---|
| mem_ba[1:0] | Output | DDR2 的 BANK 地址信号。 |
| mem_addr[12:0] | Output | DDR2 的行和列地址总线。在 DDR2 读写操作的不同时刻，通过其他信号的控制，实现列地址和行地址的传输。 |
| mem_cke | Output | DDR2 时钟使能信号，高电平有效。即该信 |

| | | |
|---|---|---|
| | | 号拉高表示 mem_clk/mem_clk_n 信号可用，拉低表示不可用。 |
| mem_cs_n | Output | DDR2 片选信号，低电平有效。 |
| mem_ras_n | Output | DDR2 行选通信号，低电平有效。作为 DDR2 的命令接口。 |
| mem_cas_n | Output | DDR2 列选通信号，低电平有效。作为 DDR2 的命令接口。 |
| mem_we_n | Output | DDR2 写选通信号，低电平有效。作为 DDR2 的命令接口。 |
| mem_dm[1:0] | Output | DDR2 数据写入屏蔽信号，高电平有效。拉高表示当前的数据无效。mem_dm[1]对应 mem_dq[15:8] 的屏蔽； mem_dm[0] 对应 mem_dq[7:0]的屏蔽。 |
| mem_odt | Output | DDR2 的片内阻抗设置。 |
| mem_clk | Inout | DDR2 差分时钟正极。 |
| mem_clk_n | Inout | DDR2 差分时钟负极。 |
| mem_dq[15:0] | Inout | DDR2 数据总线。 |
| mem_dqs[1:0] | Inout | DDR2 数据锁存时钟。 |

**sld_hub.vhd 模块代码解析**

略。该模块为 FPGA 内部自动产生，且只提供网表文件，没有代码可供设计者查看。

**ddr2_controller.v 模块代码解析**

略。关于 DDR2 控制器部分的 IP 核配置请参考“IP 核配置——DDR2 控制器”部分的内容。

**data_source.v 模块代码解析**

如图 24 所示，该模块实现定时的 DDR2 数据写入和读出操作。大约每 3.6s 为一个读写周期，在这个周期中，先执行一次 DDR2 地址为 0-1023 的数据写入，随后再执行一次 DDR2 地址为 0-1023 的数据读出，这些读出的数据将被送入 FPGA 的片内 RAM 中。

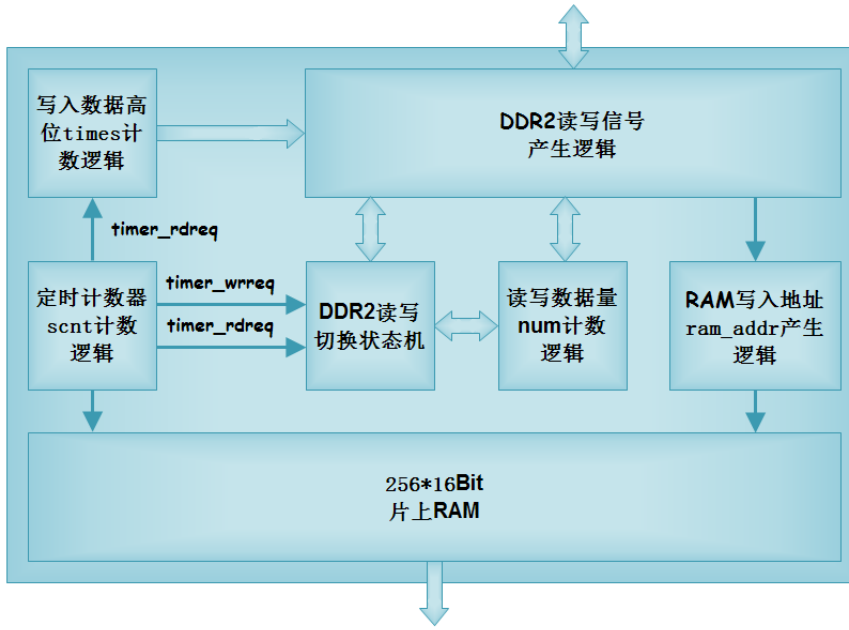



图 24 DDR2 读写控制逻辑产生功能框图

该模块的 Verilog 代码如下。

```
module data_source(
            clk,     //75MHz
            rst_n,
            local_address,
            local_write_req,
```

```
            local_read_req,
            local_wdata,
            local_ready,
            local_rdata,
            local_rdata_valid,
            local_init_done
        );

    //Clock and Reset Signals for DDR2/DDR SDRAM
input       clk;
input       rst_n;
    //Local Interface Signals for DDR2 and DDR SDRAM
output  [22:0]  local_address;
output      local_write_req;
output      local_read_req;
output  [63:0]  local_wdata;
input       local_ready;
input   [63:0]  local_rdata;
input       local_rdata_valid;
input       local_init_done;

////////////////////////////////////////////////////
//1s 定时计数逻辑     时钟为 75MHz(13.3333ns)
reg[26:0] scnt;
reg[7:0] times;

always @(posedge clk or negedge rst_n)
    if(!rst_n) scnt <= 27'd0;
    else if(local_init_done) scnt <= scnt+1'b1;

    //定时 DDR2 写数据信号
wire timer_wrreq = (scnt == 27'h00_001_000);
```

```
    //定时 DDR2 读数据信号
wire timer_rdreq = (scnt == 27'h00_005_000);

always @(posedge clk or negedge rst_n)
    if(!rst_n) times <= 8'd0;
    else if(timer_rdreq) times <= times+1'b1;

////////////////////////////////////////////////////
//产生读写 DDR 操作的状态
parameter SIDLE = 4'd0;
parameter SWRDB = 4'd1;
parameter SRDDB = 4'd2;
parameter SSTOP = 4'd3;

reg[3:0] cstate;
reg[8:0] num;

always @(posedge clk or negedge rst_n)
    if(!rst_n) cstate <= SIDLE;
    else begin
        case(cstate)
            SIDLE: begin
                if(timer_wrreq) cstate <= SWRDB;
                else if(timer_rdreq) cstate <= SRDDB;
                else cstate <= SIDLE;
            end
            SWRDB: begin
                if((num == 9'd255) && local_ready) cstate <= SSTOP;
                else cstate <= SWRDB;
            end
            SRDDB: begin
```

```
                if((num == 9'd255) && local_ready) cstate <= SSTOP;
                else cstate <= SRDDB;
            end
            SSTOP: cstate <= SIDLE;
            default: cstate <= SIDLE;
        endcase
    end

////////////////////////////////////////////////////
// (1024/4) 256 次的数据读写请求计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) num <= 9'd0;
    else if((cstate == SWRDB) || (cstate == SRDDB)) begin
        if(local_ready) num <= num+1'b1;
        else ;
    end
    else num <= 9'd0;

assign local_address = (cstate == SWRDB) ?
{13'h0a55,2'd1,num[7:0]}:{13'h0a55,2'd1,num[7:0]};

assign local_wdata =
{times,{num[5:0],2'b00},times,{num[5:0],2'b01},times,{num[5:0],
2'b10},times,{num[5:0],2'b11}};

assign local_write_req = (cstate == SWRDB);
assign local_read_req = (cstate == SRDDB);

////////////////////////////////////////////////////
//片内 RAM 例化，写入当前 DDR2 读出的 256*64bit 数据
reg[7:0] ram_addr;
```

```
always @(posedge clk or negedge rst_n)
    if(!rst_n) ram_addr <= 8'd0;
    else if(timer_rdreq) ram_addr <= 8'd0;
    else if(local_rdata_valid) ram_addr <= ram_addr+1'b1;
    else ;

onchipram_for_ddr   onchipram_for_ddr_inst (
                        .address ( ram_addr ),
                        .clock ( clk ),
                        .data ( local_rdata ),
                        .wren ( local_rdata_valid ),
                        .q (  )
                    );

endmodule
```

① DDR2 控制器 IP 核模块和用户逻辑部分的接口如下。当然这并非全部的可用接口，但我们的应用中只需要对这些接口进行操作即可完成实验。关于这部分的接口定义，我们将会在随后的“IP 核配置——DDR2 控制器”部分内容进行描述。

```
    //Local Interface Signals for DDR2 and DDR SDRAM
output  [22:0]  local_address;
output      local_write_req;
output      local_read_req;
output  [63:0]  local_wdata;
input       local_ready;
input   [63:0]  local_rdata;
input       local_rdata_valid;
input       local_init_done;
```

② 1.8s 的周期定时逻辑如下。

```
always @(posedge clk or negedge rst_n)
    if(!rst_n) scnt <= 27'd0;
    else if(local_init_done) scnt <= scnt+1'b1;

    //定时 DDR2 写数据信号
wire timer_wrreq = (scnt == 27'h00_001_000);

    //定时 DDR2 读数据信号
wire timer_rdreq = (scnt == 27'h00_005_000);
```

- 1.8s 的定时时间如何算得？注意这个模块的时钟是 75MHz，对应周期为 13.3333ns，计数器为 27 位，最大的计数值即（2 的 27 次方），那么我们把 13.3333ns 乘以（2 的 27 次方）即可得到一个计数周期的总时间为 1.789s，约为 1.8s。
- 另外，我们注意以下代码：

```
wire timer_wrreq = (scnt == 27'h00_001_000);
```

实际上等同于：

```
wire timer_wrreq;
assign timer_wrreq = (scnt == 27'h00_001_000) ? 1'b1 : 1'b0;
```

③ 状态机部分代码如下，这里实际上是一个一段式的状态机。虽然各种语法书籍可能对状态机的设计都有一套推荐套路，并且对一段式、两段式和三段式都有一些优劣的方法。不过，具体应用应该具体分析。对于我们这种非常简单的应用，一段式状态机不失为一种“偷懒”的写法。实际上，一段时非常简单明了，对新手来说非常容易搞明白。

```
//////////////////////////////////////////////////////
//产生读写 DDR 操作的状态
parameter SIDLE = 4'd0;
parameter SWRDB = 4'd1;
parameter SRDDB = 4'd2;
parameter SSTOP = 4'd3;
```

```
reg[3:0] cstate;
reg[8:0] num;

always @(posedge clk or negedge rst_n)
    if(!rst_n) cstate <= SIDLE;
    else begin
        case(cstate)
            SIDLE: begin
                if(timer_wrreq) cstate <= SWRDB;
                else if(timer_rdreq) cstate <= SRDDB;
                else cstate <= SIDLE;
            end
            SWRDB: begin
                if((num == 9'd255) && local_ready) cstate <= SSTOP;
                else cstate <= SWRDB;
            end
            SRDDB: begin
                if((num == 9'd255) && local_ready) cstate <= SSTOP;
                else cstate <= SRDDB;
            end
            SSTOP: cstate <= SIDLE;
            default: cstate <= SIDLE;
        endcase
    end
```

- parameter 是定义一个参数，和 C 语言中的宏定义类似，当然了，Verilog 中也有宏定义。只不过这个 parameter 只能定义一些具体的数据值。参数的名称大家是可以随便定义的，这里用大写字符“SIDLE”定义。Parameter 的名称对大小写也没有要求，这里以及今后的设计中对 parameter 统一用大写字母，只是为了规范代码，

便于阅读和管理。

```
parameter SIDLE = 4'd0;
```

- 状态机中判断某些输入信号的变化，若是某些切换条件满足，那么就切换到另一个状态。例如，在 SIDLE 状态下，我们就判断 timer_wrreq 和 timer_rdreq 信号是否拉高，若是则相应切换到 SWRDB 和 SRDDB 状态，若否则继续保持当前状态不变。

```
            SIDLE: begin
                if(timer_wrreq) cstate <= SWRDB;
                else if(timer_rdreq) cstate <= SRDDB;
                else cstate <= SIDLE;
```

- 这里还有一个“{}”的语句，代码如下所示。

```
assign local_wdata =
{times,{num[5:0],2'b00},times,{num[5:0],2'b01},times,{num[5:0],
2'b10},times,{num[5:0],2'b11}};
```

我们先用一个更简单的例子普及一下“{}”符号在 Verilog 语法中的意义。

```
Input[7:0] a;
Input[7:0] b;
output[15:0] c;
assign c = {a,b};
```

该段代码表示 16 位宽的输入信号 c 的值为 8 位输入信号 a 和 b 拼接而成。即输入信号 a 的值赋给输出信号 c 的高 8 位，输入信号 b 的值赋给输出信号 c 的低 8 位。若 a=8'haa，b=8'h55，那么 c=16'haa55。

解释完这个例子，想必大家回头看工程中的代码就能明白了。

④ 片内 RAM 的例化代码如下。

```
onchipram_for_ddr   onchipram_for_ddr_inst (
                        .address ( ram_addr ),
                        .clock ( clk ),
```

```
                .data ( local_rdata ),
                .wren ( local_rdata_valid ),
                .q ( )
            );
```

**onchipram_for_ddr.v 模块代码解析**

略。关于片内 RAM 的 IP 核配置请参考“IP 核配置——片内 RAM”部分的内容。

**sys_ctrl.v 模块代码解析**

略。

**pll_controller.v 模块代码解析**

略。

**led_controller.v 模块代码解析**

略。

# 6 板级调试

① 打开“…\prj\hsc_ex2”文件夹下的工程。

② 点击 Quartus II 菜单栏的“Tools→In-System Memory Content Editor”，在界面的右侧，如图 25 所示，选择“…\prj\hsc_ex2\output_files”文件夹下的 hsc.sof，执行下载操作，即点击 File 右侧的小按钮。

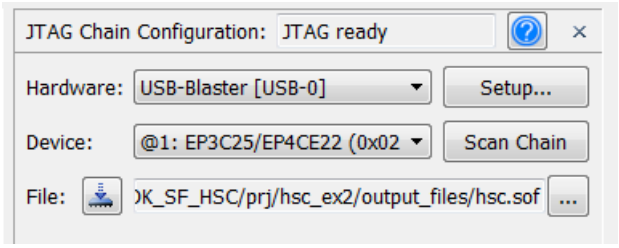



图 25 JTAG 下载配置界面

③ 下载完成后可以看到 HSC 开发板上的指示灯 D1 闪烁。此时我们接着选中 Index 下面的 Memory 项，然后单击循环读取按钮，如图 26 所示。

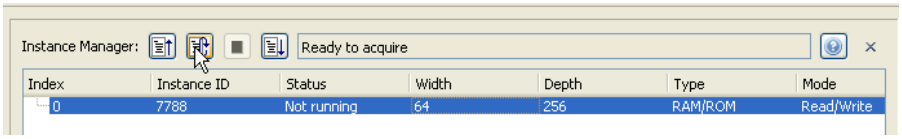
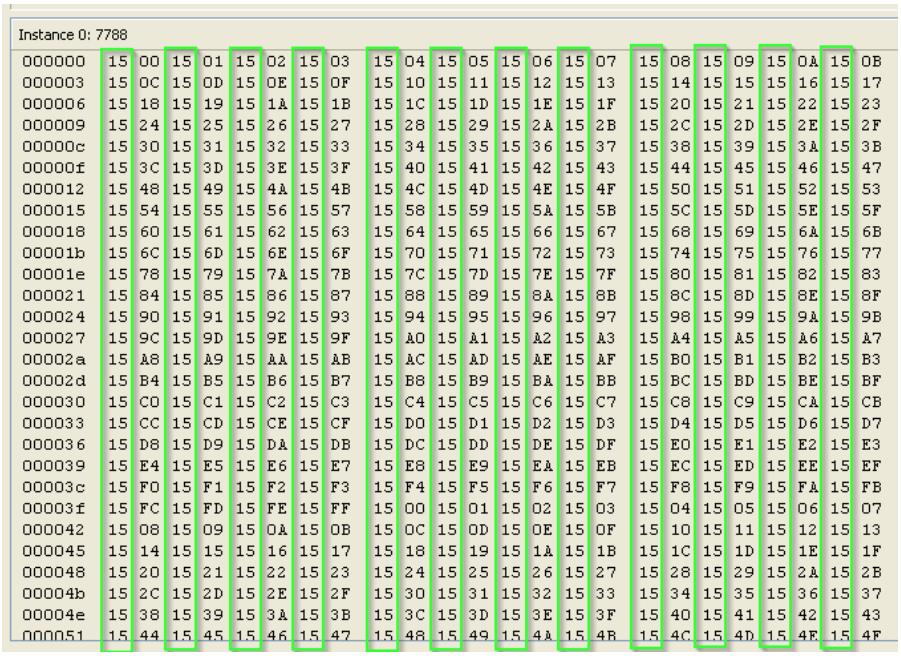



图 26 可在线查看的片内 RAM

④ 接着我们可以观察 Memory 当前的数据变化。如图 27 所示，矩形框起来的高字节数据，即我们每隔一秒多统一递增一次的数据，而其后的低字节数据则始终是从 0 开始递增和地址一一对应的递增数据。因此，我们看到的实验结果是，这个 onchip RAM 的所有 16bit 的高字节每隔一秒都会递增一，而其后的数据则一直保持当前状态不变。


Instance 0: 7788

| | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 000000 | 15 | 00 | 15 | 01 | 15 | 02 | 15 | 03 | 15 | 04 | 15 | 05 | 15 | 06 | 15 | 07 | 15 | 08 | 15 | 09 | 15 | 0A | 15 | 0B |
| 000003 | 15 | 0C | 15 | 0D | 15 | 0E | 15 | 0F | 15 | 10 | 15 | 11 | 15 | 12 | 15 | 13 | 15 | 14 | 15 | 15 | 15 | 16 | 15 | 17 |
| 000006 | 15 | 18 | 15 | 19 | 15 | 1A | 15 | 1B | 15 | 1C | 15 | 1D | 15 | 1E | 15 | 1F | 15 | 20 | 15 | 21 | 15 | 22 | 15 | 23 |
| 000009 | 15 | 24 | 15 | 25 | 15 | 26 | 15 | 27 | 15 | 28 | 15 | 29 | 15 | 2A | 15 | 2B | 15 | 2C | 15 | 2D | 15 | 2E | 15 | 2F |
| 00000c | 15 | 30 | 15 | 31 | 15 | 32 | 15 | 33 | 15 | 34 | 15 | 35 | 15 | 36 | 15 | 37 | 15 | 38 | 15 | 39 | 15 | 3A | 15 | 3B |
| 00000f | 15 | 3C | 15 | 3D | 15 | 3E | 15 | 3F | 15 | 40 | 15 | 41 | 15 | 42 | 15 | 43 | 15 | 44 | 15 | 45 | 15 | 46 | 15 | 47 |
| 000012 | 15 | 48 | 15 | 49 | 15 | 4A | 15 | 4B | 15 | 4C | 15 | 4D | 15 | 4E | 15 | 4F | 15 | 50 | 15 | 51 | 15 | 52 | 15 | 53 |
| 000015 | 15 | 54 | 15 | 55 | 15 | 56 | 15 | 57 | 15 | 58 | 15 | 59 | 15 | 5A | 15 | 5B | 15 | 5C | 15 | 5D | 15 | 5E | 15 | 5F |
| 000018 | 15 | 60 | 15 | 61 | 15 | 62 | 15 | 63 | 15 | 64 | 15 | 65 | 15 | 66 | 15 | 67 | 15 | 68 | 15 | 69 | 15 | 6A | 15 | 6B |
| 00001b | 15 | 6C | 15 | 6D | 15 | 6E | 15 | 6F | 15 | 70 | 15 | 71 | 15 | 72 | 15 | 73 | 15 | 74 | 15 | 75 | 15 | 76 | 15 | 77 |
| 00001e | 15 | 78 | 15 | 79 | 15 | 7A | 15 | 7B | 15 | 7C | 15 | 7D | 15 | 7E | 15 | 7F | 15 | 80 | 15 | 81 | 15 | 82 | 15 | 83 |
| 000021 | 15 | 84 | 15 | 85 | 15 | 86 | 15 | 87 | 15 | 88 | 15 | 89 | 15 | 8A | 15 | 8B | 15 | 8C | 15 | 8D | 15 | 8E | 15 | 8F |
| 000024 | 15 | 90 | 15 | 91 | 15 | 92 | 15 | 93 | 15 | 94 | 15 | 95 | 15 | 96 | 15 | 97 | 15 | 98 | 15 | 99 | 15 | 9A | 15 | 9B |
| 000027 | 15 | 9C | 15 | 9D | 15 | 9E | 15 | 9F | 15 | A0 | 15 | A1 | 15 | A2 | 15 | A3 | 15 | A4 | 15 | A5 | 15 | A6 | 15 | A7 |
| 00002a | 15 | A8 | 15 | A9 | 15 | AA | 15 | AB | 15 | AC | 15 | AD | 15 | AE | 15 | AF | 15 | B0 | 15 | B1 | 15 | B2 | 15 | B3 |
| 00002d | 15 | B4 | 15 | B5 | 15 | B6 | 15 | B7 | 15 | B8 | 15 | B9 | 15 | BA | 15 | BB | 15 | BC | 15 | BD | 15 | BE | 15 | BF |
| 000030 | 15 | C0 | 15 | C1 | 15 | C2 | 15 | C3 | 15 | C4 | 15 | C5 | 15 | C6 | 15 | C7 | 15 | C8 | 15 | C9 | 15 | CA | 15 | CB |
| 000033 | 15 | CC | 15 | CD | 15 | CE | 15 | CF | 15 | D0 | 15 | D1 | 15 | D2 | 15 | D3 | 15 | D4 | 15 | D5 | 15 | D6 | 15 | D7 |
| 000036 | 15 | D8 | 15 | D9 | 15 | DA | 15 | DB | 15 | DC | 15 | DD | 15 | DE | 15 | DF | 15 | E0 | 15 | E1 | 15 | E2 | 15 | E3 |
| 000039 | 15 | E4 | 15 | E5 | 15 | E6 | 15 | E7 | 15 | E8 | 15 | E9 | 15 | EA | 15 | EB | 15 | EC | 15 | ED | 15 | EE | 15 | EF |
| 00003c | 15 | F0 | 15 | F1 | 15 | F2 | 15 | F3 | 15 | F4 | 15 | F5 | 15 | F6 | 15 | F7 | 15 | F8 | 15 | F9 | 15 | FA | 15 | FB |
| 00003f | 15 | FC | 15 | FD | 15 | FE | 15 | FF | 15 | 00 | 15 | 01 | 15 | 02 | 15 | 03 | 15 | 04 | 15 | 05 | 15 | 06 | 15 | 07 |
| 000042 | 15 | 08 | 15 | 09 | 15 | 0A | 15 | 0B | 15 | 0C | 15 | 0D | 15 | 0E | 15 | 0F | 15 | 10 | 15 | 11 | 15 | 12 | 15 | 13 |
| 000045 | 15 | 14 | 15 | 15 | 15 | 16 | 15 | 17 | 15 | 18 | 15 | 19 | 15 | 1A | 15 | 1B | 15 | 1C | 15 | 1D | 15 | 1E | 15 | 1F |
| 000048 | 15 | 20 | 15 | 21 | 15 | 22 | 15 | 23 | 15 | 24 | 15 | 25 | 15 | 26 | 15 | 27 | 15 | 28 | 15 | 29 | 15 | 2A | 15 | 2B |
| 00004b | 15 | 2C | 15 | 2D | 15 | 2E | 15 | 2F | 15 | 30 | 15 | 31 | 15 | 32 | 15 | 33 | 15 | 34 | 15 | 35 | 15 | 36 | 15 | 37 |
| 00004e | 15 | 38 | 15 | 39 | 15 | 3A | 15 | 3B | 15 | 3C | 15 | 3D | 15 | 3E | 15 | 3F | 15 | 40 | 15 | 41 | 15 | 42 | 15 | 43 |
| 000051 | 15 | 44 | 15 | 45 | 15 | 46 | 15 | 47 | 15 | 48 | 15 | 49 | 15 | 4A | 15 | 4B | 15 | 4C | 15 | 4D | 15 | 4E | 15 | 4F |

图 27 递增的 DDR2 读出数据