

第二十四章 IP 核之单端口 RAM 实验

RAM 的英文全称是 Random Access Memory, 即随机存取存储器, 简称**随机存储器**, 它可以随时把数据写入任一指定地址的存储单元, 也可以随时从任一指定地址的存储单元中读出数据, 其读写速度是由时钟频率决定的。本章我们将向大家介绍 Xilinx RAM IP 核的使用方法并对 RAM IP 核进行简单的读写测试。

本章包括以下几个部分:

24.1 RAM 简介

24.2 硬件设计

24.3 实验任务

24.4 程序设计

24.5 下载验证

24.6 本章总结

24.7 拓展训练

24.1 RAM 简介

在了解 RAM IP 核之前,我们先来看下存储器的大致分类,如下图所示:

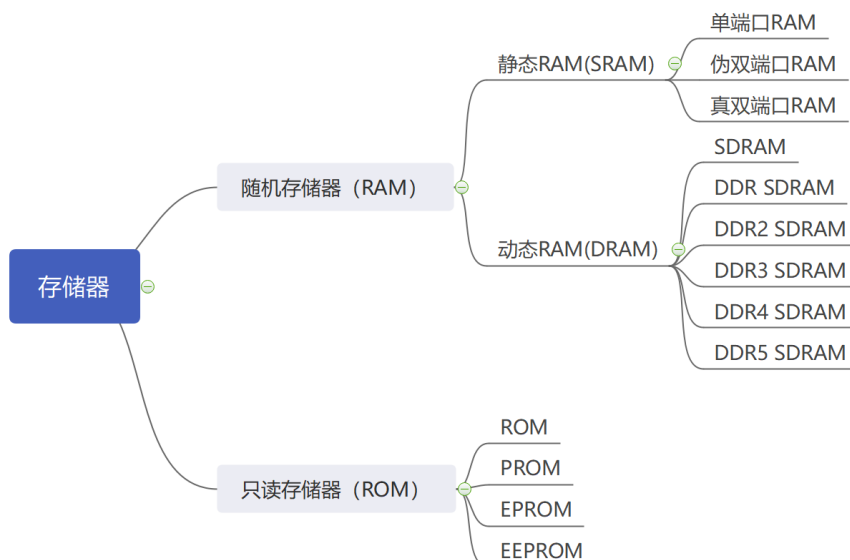


图 24.1.1 存储器分类

由上图可知,存储器包括随机存储器和只读存储器,随机存储器包括静态 RAM 和动态 RAM。静态 RAM 只要有供电,它保存的数据就不会丢失;而动态 RAM 在供电的情况下,还需要根据其要求的时间来对存储的数据进行刷新,才能保持存储的数据不会丢失。

静态 RAM 一般包括单端口 RAM (Single Port RAM, 缩写为 SPRAM)、简单双端口 RAM (Simple Dual Port RAM, 缩写为 SDP RAM, 也叫伪双端口 RAM) 和真双端口 RAM (True Dual Port RAM, 缩写为 TDP RAM)。静态 RAM 的特点是存储容量相对不是很大,但是读写速度非常高,其在 FPGA 或者 ASIC 设计中都属于非常重要的器件,可以说查找表、寄存器、组合逻辑和静态 RAM 构成了整个数字电路体系,足见静态 RAM 的重要性。

动态 RAM 一般包括 SDRAM 和 DDR SDRAM。目前 DDR SDRAM 已经从 DDR1 代发展到 DDR5 代了,DDR3 和 DDR4 SDRAM 是目前非常主流的存储器,大量使用在电脑、嵌入式和 FPGA 板卡上面,其特点是存储容量非常大、但是读写速度相比于静态 RAM 会稍低一些,这一点在数据量较少的情况下尤为明显。

只读存储器一般包括 PROM、EPROM 和 EEPROM 等,是非易失性的存储器。目前使用率较高的是 EEPROM,其特点是容量相对较小,存储的一般是器件的配置参数信息,例如 USB 2.0 芯片一般会配有一个 EEPROM 来存储相关的固件信息。

本章我们学习的 RAM 属于静态 RAM,我们重点看下几种静态 RAM 的特性与区别:

表 24.1.1 几种静态RAM的特性区别

RAM类型	特性说明	数据读写口数量	地址口数量
单端口RAM	1、允许通过一个端口对存储进行读写访问。 2、只有一组数据总线、地址总线、时钟信号以及读写使能信号。	1	1
伪双端口RAM	1、提供A和B两个端口，端口A只能进行写访问，端口B只能进行读访问。 2、有两组数据总线、地址总线、时钟信号以及读写使能信号。	1	2
真双端口RAM	1、提供A和B两个端口，两个端口均可对存储进行读写访问。 2、有两组数据总线、地址总线、时钟信号以及读写使能信号。	2	2

从上表可以看出，单端口 RAM 只有一个端口进行读写，即读/写只能通过这一个端口来进行。对于伪双端口 RAM 而言，其也有两个端口可以用于读写，但是其中一个端口只能读不能写，另一个端口只能写不能读；对于真双端口 RAM 而言，其有两个端口可以用于读写，且两个端口都可以进行读或写；

不同的特性决定不同的应用场景，在 RAM 的实际应用中，我们一般根据功能需求和带宽需求来选择合适的 RAM 类型，例如：

1、当我们需要读取一个配置，且这个配置只在上电的时候配置一次，其他时候不需要写操作，那么我们直接选择单端口 RAM 即可，通过一个端口要么进行写操作，要么进行读操作。

2、当我们需要使用 FIFO（先进先出存储器）来存储数据，就可以选择伪双端口 RAM，一个写端口，一个读端口，且读写可以同时进行。

3、当我们想要实现一个对 10000 节车厢的人数进行统计的功能，每节车厢有两个门，一个门仅用于上车，另一个门仅用于下车，有人上车时需要在原有的人数基础上加一，有人下车时需要在原有的人数基础上减一，每时每刻都可能有人上下车，那么要使用逻辑统计这么多节车厢的人数时，就需要有两个写端口的 RAM，深度为 10000（对应 10000 节车厢），这时就要选择真双端口 RAM 了。因为如果单纯的使用寄存器资源进行统计的话，仅这 10000 节车厢的计数就很可能把 FPGA 的寄存器资源消耗殆尽了。

通过上述的学习，想必大家已经对几种 RAM 的区别有了一定的了解，那么我们就回到实际应用中。在 FPGA 设计中 RAM 主要用于存放程序和数据，在 RAM 的实际使用过程中，我们一般将其作为关键数据的临时存储。接下来我们就介绍一下 Vivado 中的 RAM IP 核。

Vivado 软件自带的 Block Memory Generator IP 核（缩写为 BMG，中文名为块 RAM 生成器），可以用来配置生成 RAM 或者 ROM。RAM 是一种随机存取存储器，不仅可以读出存储的数据，同时还支持对存储的数据进行修改，而 ROM 是一种只读存储器，也就是说，在工作时只能读出数据，而不能写入数据。需要注意的是，配置生成的 RAM 或者 ROM 使用的都是 FPGA 内部的 BRAM 资源（Block RAM，即块随机存储器，是 FPGA 厂商在逻辑资源之外，给 FPGA 加入的专用 RAM 块资源），只不过配置成 ROM 时只用了嵌入式 BRAM 的读数据端口。本章我们主要介绍如何将 BMG IP 核配置成 RAM。

这里有个地方需要大家注意一下，Xilinx 7 系列器件内部的 BRAM 全部是真双端口 RAM，但是通过 BMG IP 核，我们还可以将其配置为伪双端口 RAM 或者单端口 RAM。

有关双端口 RAM 的内容我们将在下一章进行讲解，本章我们围绕单端口 RAM 进行讲解。

BMG IP 核配置成单端口 RAM 的框图如下图所示。

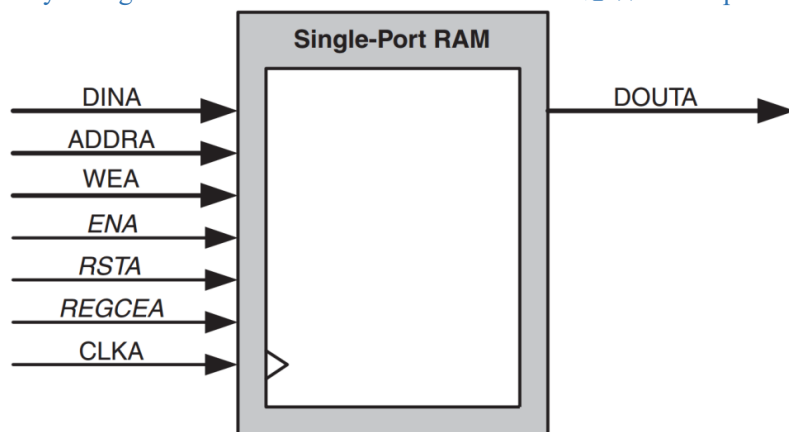


图 24.1.2 单端口 RAM 框图

各个端口的功能描述如下:

表 24.1.2 单端口RAM端口描述表

端口名	对应中文名	端口方向	端口描述
DINA	写数据信号	输入	写数据端口, 一般由多bit构成
ADDR A	地址信号	输入	RAM的地址信号, 由外部输入, 地址位宽决定了RAM的最大深度, 比如地址为 4, 则 RAM 最大深度是16
WEA	写使能信号	输入	高电平表示向RAM中写入数据, 低电平表示从RAM中读出数据
ENA	端口使能信号	输入	高电平表示使能端口A, 低电平表示禁用端口A, 禁止后端口A上的读写操作都会变为无效
RSTA	复位信号	输入	高电平有效
REGCEA	输出寄存器使能信号	输入	REGCEA为低电平时, 读端口保持最后一次输出的数据
CLKA	时钟信号	输入	由外部输入给RAM使用的时钟
DOUT A	读数据信号	输出	读数据端口, 由多bit构成, 一般和写数据位宽相同

24.2 硬件设计

本章实验只需要用到时钟信号和按键复位信号, 所以没有新的硬件设计内容需要进行讲解。

24.3 实验任务

本章实验任务是将 Xilinx BMG IP 核配置成一个单端口的 RAM 并对其进行读写操作, 然后通过仿真观察波形是否正确, 最后将设计下载到 FPGA 开发板中, 并通过在线调试工具对实验结果进行验证。

24.4 程序设计

24.4.1 RAM IP 核讲解

RAM IP 核配置

首先我们创建一个名为“ip_1port_ram”的空白工程, 然后点击 Vivado 软件左侧“Flow Navigator”栏中的“IP Catalog”, 如下图所示:

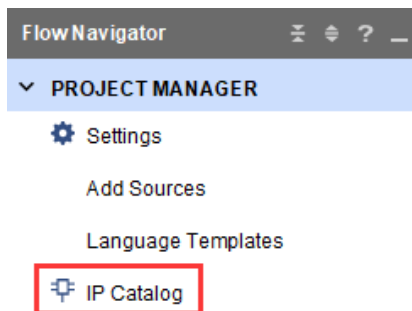


图 24.4.1 点击 “IP Catalog”

在 “IP Catalog” 窗口的搜索栏中输入 “Block Memory” 关键字后, 出现唯一匹配的 “Block Memory Generator”, 如下图所示 (图中出现的两个 IP 核为同一个 BMG IP 核):

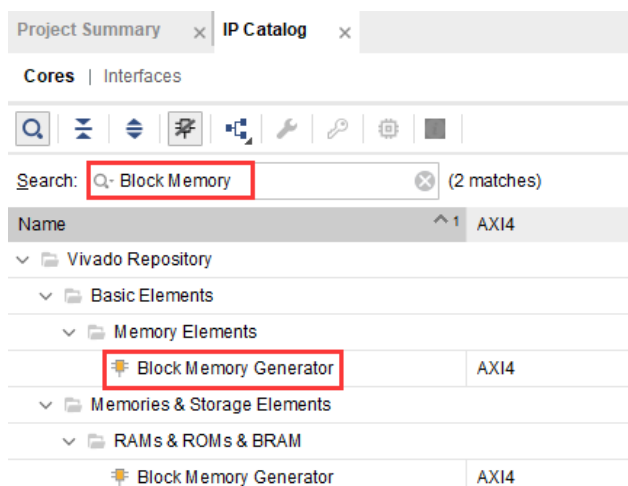


图 24.4.2 搜索栏中输入 “Block Memory” 关键字

双击 “Block Memory Generator” 后弹出 IP 核的配置界面, 接着我们就可以对 BMG IP 核进行配置了, “Basic” 选项卡配置界面如下图所示。

The image shows the 'Basic' tab of the 'blk_mem_gen_0' component configuration window. The 'Component Name' field is set to 'blk_mem_gen_0'. The 'Interface Type' is set to 'Native' and 'Memory Type' is set to 'Single Port RAM'. The 'Generate address interface with 32 bits' checkbox is unchecked. The 'Common Clock' checkbox is unchecked. The 'ECC Options' section shows 'ECC Type' set to 'No ECC' and 'Error Injection Pins' set to 'Single Bit Error Injection'. The 'Write Enable' section shows 'Byte Write Enable' unchecked and 'Byte Size (bits)' set to '9'. The 'Algorithm Options' section shows 'Algorithm' set to 'Minimum Area' and 'Primitive' set to '8lx2'.

图 24.4.3 “Basic”选项卡配置界面

最上面的“Component Name”一栏可以设置该 IP 元件的名称，这里我们保持默认命名，当然也可以命名为其它方便自己一眼看出其功能的名称。

接着目光回到“Basic”选项卡上，该选项卡下各参数（我们重点关注（1）和（3）中的内容）含义如下：

（1）、“Interface Type（接口模式）”：有两种接口模式可选，分别为 Native（常规）接口和 AXI4 接口。AXI4 模式一般是在处理器中的数据需要和 BRAM 交互（如 ZYNQ 中的 PS 和 PL 交互）时才会使用，当不需要与处理器数据进行交互时，我们一般采用 Native 模式，所以本次实验我们选择 Native 模式。

（2）、“Generate address interface with 32 bits”选项为是否生成位宽为 32 的地址接口，勾选后内存的数据位宽必须设置成 32 的整数倍，在实际应用中，我们一般不做勾选。

（3）、“Memory Type（存储类型）”：有五种类型可选，分别为“Single Port RAM（单端口 RAM）”、“Simple Dual Port RAM（伪双端口 RAM）”、“True Dual Port RAM（真双端口 RAM）”、“Single Port ROM（单端口 ROM）”和“Dual Port ROM（双端口 ROM）”，因为本章的试验任务是学习单端口 RAM，所以我们选择“Single Port RAM”。

（4）、“Common Clock”选项为是否启用同步时钟，在单端口模式下不需要考虑这个问题，所以该选项也无法进行设置。

（5）、“ECC（全称 Error Correcting Code，即纠错码）Options”，纠错码选项只有在伪双端口 RAM 类型下才可以进行配置。

（6）、“Write Enable（写使能）”：可以选择是否使用字节写使能功能，启用后可设置字节大小为 8 位（无奇偶校验）或 9 位（包括奇偶校验），需要注意的是启用后内存的数据位宽必须设置为所选字节大小的整数倍。这里我们不启用字节写使能功能。

（7）、“Algorithm Options（算法选项）”：算法选项主要用于决定 BRAM 的拼接的方式，一般在 BRAM 深度、宽度较大的时候起作用，有三种算法可选，分别为“Minimum Area（最小面积算法）”、“Low Power（低功耗算法）”和“Fixed Primitives（固定单元算法）”。这里我们保持默认即可。

接下来我们对“Port A Options”选项卡进行配置, 如下图所示:

图 24.4.4 “Port A Options”选项卡配置界面

“Port A Options”选项卡下各参数(我们重点关注(1)中的内容)含义如下:

(1)、“Memory Size (内存大小)”, 用于指定端口读/写宽度和深度、运行模式和使能端口类型, 具体描述如下:

Write Width: 写数据位宽, 单位 bit, 本次实验我们设置成 8, 可设置的位宽范围为 1~4608。

Read Width: 读数据位宽, 其位宽设置必须与写数据位宽存在倍数关系(倍数比仅支持 1: 1、1: 2、1: 4、1: 8、2: 1、4: 1、8: 1 和 16: 1), 通常情况下读/写数据位宽保持一致, 所以这里设置成 8。

Write Depth: 写深度, 这里设置成 32, 即 RAM 所能访问的地址范围为 0~31。这里有一点需要大家注意, 那就是写深度和写位宽的乘积不要超过器件本身的 ram 资源的大小, 否则软件会报错。

Read Depth: 读深度, 当写数据位宽、读数据位宽和写深度的值确定后, 读深度的值就会自动确定。

Operating Mode: RAM 运作模式。该选项决定了我们在进行写操作期间, DOUT (读数据线) 上的数据变化, 有三种模式可选, 分别为:

- **Write First (写优先模式):** 当我们在对某个地址进行写操作时, 则会将写入的数据传递到读数据线上。
- **Read First (读优先模式):** 当我们在对某个地址进行写操作时, 则会将上次写入该地址的数据递到读数据线上。
- **No Change (不变模式):** 在该模式下, 进行写操作时, 读数据线上的数据保持不变。

这里我们选择 No Change 模式。

Enable Port Type: 使能端口类型。有两种可选, 分别为 Use ENA pin (添加使能端口 A 信号) 和 Always Enabled (取消使能信号, 端口 A 一直处于使能状态), 这里选择默认的 Use ENA pin。

(2)、“Port A Optional Output Register”用于为 RAM 的输出端添加寄存器。其作用是提高 BRAM 的运行频率和改善时序, 当然为此付出的代价就是每勾选一个寄存器, 输出就会延迟一拍。其中:

- **Primitives Output Register,** 使用 BRAM 内部的寄存器打拍输出。
- **Core Output Register,** 使用 SLICE 的寄存器打拍输出。
- **SoftECC Input Register,** 当使用软 ECC 的时候, 用 SLICE 的寄存器打拍。

• REGCEA, 当使用 Primitives Output Register 或者 Core Output Register 时, 可以用 REGCEA 来使能相应的输出。

本次实验中, 建议生成 ip 的时候, 取消 primitives output register 配置, 这样我们在当前时钟周期发起读操作的话, 在下一个时钟周期就可以读出有效数据了。

(3)、“Port A Output Reset Options”用于设置端口的复位信号, 可以添加一个复位信号 (RSTA Pin)、设置复位时 ram 输出总线上的数据值 (Output Reset Value)、设置是否复位内置锁存器 (Reset Memory Latch) 和设置复位信号与时钟使能之间的优先级 (Reset Priority), 这里我们不需要添加, 所以不做勾选。

(4)、“READ Address Change A”用于更改端口的读地址, 这个功能只在 UltraScale 设备上使用, 一般在某些低功耗场景中会应用到。

接下来是“Other Options”选项卡, 用于设置 RAM 的初始值等信息, 虽然本次实验不需要设置, 但是我们可以简单的了解一下, 其界面如下所示:

图 24.4.5 “Other Options”选项卡配置界面

“Other Options”选项卡下各参数含义如下:

(1)、“Pipeline Stages within Mux”输出端 Mux 选择器的流水线级数。在大位宽、大深度的 BRAM 拼接场景中, 我们会用 MUX 来选择输出地址所对应的数据, 勾选该选项后可以使输出数据具有更好的时序。

(2)、“Memory Initialization (初始化文件)”, 选择是否使用本地初始化文件 (.coe 文件) 来对存储空间进行初始化。

(3)、“Structural/UniSim Simulation Model Options”, 用于选择结构仿真模型发生碰撞时生成的警告消息和输出的类型。

(4)、“Behavioral Simulation Model Options”, 用于关闭仿真时的冲突告警和超出范围告警。

最后一个是“Summary”选项卡, 该界面显示了我们配置的存储器的类型, 消耗的 BRAM 资源等信息, 我们直接点击“OK”按钮完成 BMG IP 核的配置, 如下图所示:

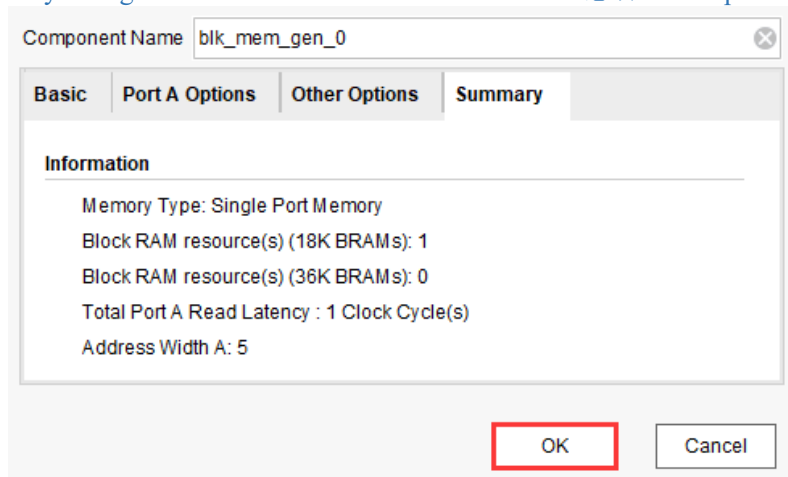


图 24.4.6 “Summary” 界面

接着就弹出了“Generate Output Products”窗口，我们直接点击“Generate”即可，如下图所示：

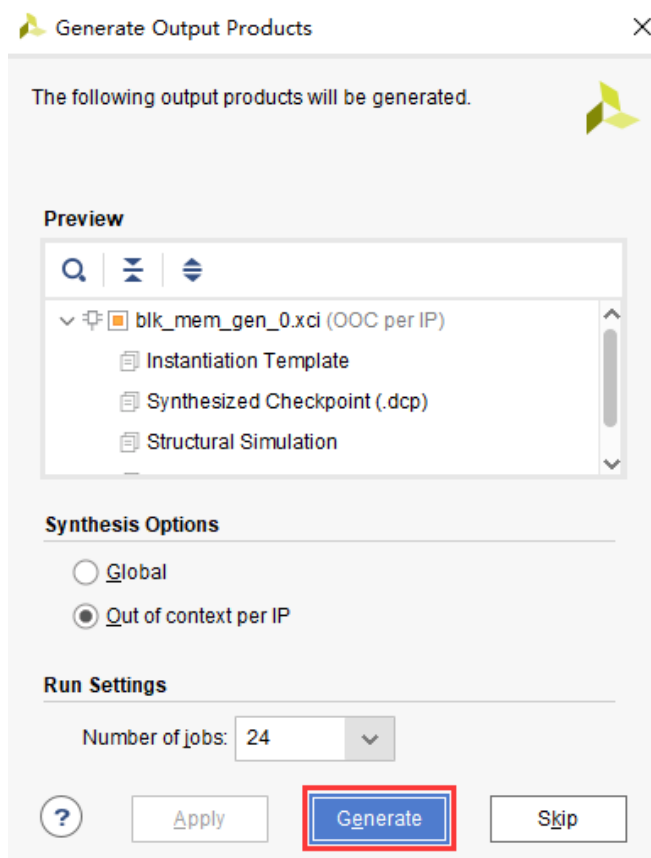


图 24.4.7 “Generate Output Products” 窗口

之后我们就可以在“Design Run”窗口的“Out-of-Context Module Runs”一栏中看到该 IP 核对应的 run “blk_mem_gen_0_synth_1”，其综合过程独立于顶层设计的综合，所以我们可以看到其正在综合，如下图所示：

Tcl Console

Messages

Log

Reports

Design Runs

×

🔍

⌵

⌶

⏮

⏪

⏩

⏭

+

%

Name	Constraints	Status
<div>▼ ▶ synth_1 (active)</div>	constrs_1	Not started
▶ impl_1	constrs_1	Not started
<div>▼ 📁 Out-of-Context Module Runs</div>		
🟢 blk_mem_gen_0_synth_1	blk_mem_gen_0	Running synth_design...

图 24.4.8 “blk_mem_gen_0_synth_1” run

接下来我们看下 IP 核的接口时序。

时序图讲解

Xilinx BMG IP 核提供了三种 RAM 读写操作模式，分别为写优先模式、读优先模式和不变模式，针对不同的模式，Xilinx 也给出了不同的时序图。我们想要对 BMG IP 核进行读写操作，首先就需要了解其时序，如果读写操作不满足其时序，那么设计的代码就不能达到我们预期的效果，所以本小节就带着大家一起学习在不同 RAM 读写操作模式下的读写时序。

1、写优先模式的时序图如下所示：

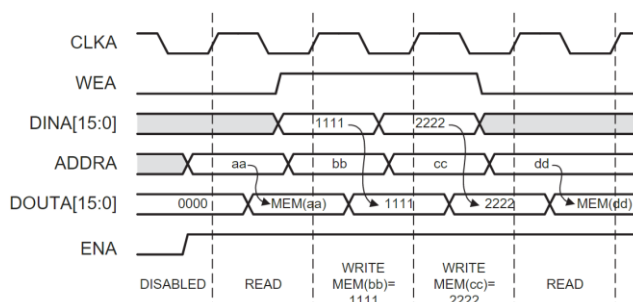


图 24.4.9 写优先模式时序图

在写优先模式下，当 ENA（使能信号）为高后，第一个时钟上升沿 WEA（读/写使能信号）为低电平，表示读数据，此时的地址为 aa，所以读出的就是 aa 内的数据；第二个时钟上升沿 WEA 为高电平，表示写数据，此时的地址为 bb，即先将数据(DAIN)“1111”写入地址 bb 后再读出，所以读出的数据也是“1111”；第三个时钟上升沿同理，将 DINA 数据写入存储器后再将更新后的数据送到 DOUTA 上进行输出；第四个时钟上升沿 WEA 为低电平，表示读数据，读取的就是地址 dd 内的数据。

2、读优先模式的时序图如下所示：

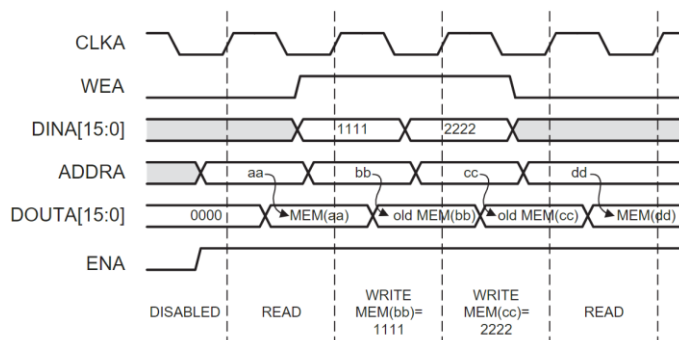


图 24.4.10 读优先模式时序图

在读优先模式下，当 ENA（使能信号）为高后，第一个时钟上升沿 WEA 为低电平，表示读数据，此时的地址为 aa，所以读出的就是 aa 内的数据；第二个时钟上升沿 WEA 为高电平，表示写数据，此时的地址为 bb，即先读出地址 bb 中的旧数据，然后再将数据(DAIN)“1111”写入地址 bb；第三个时钟上升沿同

理, 先将当前存储器中的旧数据送到 DOUTA 上进行输出, 然后再将 DINA 数据写入当前存储地址; 第四个时钟上升沿 WEA 为低电平, 表示读数据, 读取的就是地址 dd 内的数据。

3、不变模式的时序图如下所示:

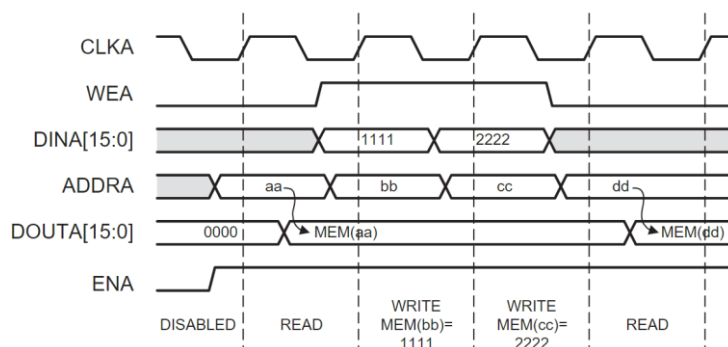


图 24.4.11 不变模式时序图

在不变模式下, 当 ENA 和 WEA 同时为高时, 只进行写操作, 读数据保持不变, 也就是说 DOUTA 保持前一拍数据, 直到 WEA 为低。

总结一下就是在进行写操作时, 读数据线 (DOUT) 上的数据在写优先模式和读优先模式下会随着地址的变化而输出相应的数据, 在不变模式下则会保持最后一次读操作输出的数据。

24.4.2 顶层模块设计

本次实验的目的是为了将 Xilinx BMG IP 核配置成一个单端口的 RAM 并对其进行读写操作, 因此可以给模块命名为 `ip_1port_ram`。由于单端口 RAM 的读写是共用一组信号总线的, 所以这里我们将读写操作合成一个模块, 可以将其命名为 `ram_rw`; 系统时钟和系统复位是一个完整的工程中必不可少的输入端口信号, 这里就不再多讲了; 因为 RAM 主要用于存储程序中的数据, 并不需要输出到 IO 引脚上, 所以本实验不需要输出端口。由上述分析我们可以画出一个大致的模块框图, 如下图所示:

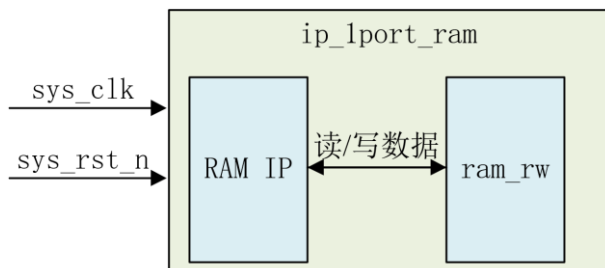


图 24.4.12 顶层模块框图与端口

模块端口与功能描述如下表所示:

表 24.4.1 `ip_1port_ram` 模块端口与功能描述

信号名	位宽	方向	端口说明
<code>sys_clk</code>	1	输入	系统时钟, 50MHz
<code>sys_rst_n</code>	1	输入	系统复位按键, 低电平有效

代码编写

因为本次实验除了调用 BMG IP 核外还需要例化一个读写模块 (这里我们将其命名为 `ram_rw`), 所以我们需要创建一个顶层模块来例化 IP 核与读写模块, 这里我们可以将顶层模块命名为 `ip_1port_ram`, 代码

如下:

```

1  module ip_lport_ram(
2      input      sys_clk ,    //系统时钟
3      input      sys_rst_n    //系统复位, 低电平有效
4  );
5
6  //wire define
7  wire          ram_en       ; //RAM 使能
8  wire          ram_we       ; //ram 读写使能信号, 高电平写入, 低电平读出
9  wire  [4:0]    ram_addr    ; //ram 读写地址
10 wire  [7:0]    ram_wr_data ; //ram 写数据
11 wire  [7:0]    ram_rd_data ; //ram 读数据
12
13 //*****
14 /**                                main code
15 //*****
16
17 //ram 读写模块
18 ram_rw  u_ram_rw(
19     .clk      (sys_clk      ),
20     .rst_n    (sys_rst_n    ),
21     //RAM
22     .ram_en   (ram_en       ),
23     .ram_we   (ram_we       ),
24     .ram_addr (ram_addr     ),
25     .ram_wr_data (ram_wr_data ),
26     .ram_rd_data (ram_rd_data ),
27 );
28
29 //ram ip 核
30 blk_mem_gen_0 blk_mem_gen_0 (
31     .clka (sys_clk      ), // input wire clka
32     .ena  (ram_en       ), // input wire ena
33     .wea  (ram_we       ), // input wire [0 : 0] wea
34     .addra (ram_addr     ), // input wire [4 : 0] addra
35     .dina  (ram_wr_data  ), // input wire [7 : 0] dina
36     .douta (ram_rd_data  ) // output wire [7 : 0] douta
37 );
38
39 endmodule

```

可以看出 ip_lport_ram 顶层模块只是例化了 IP 核 (blk_mem_gen_0) 和读写模块 (ram_rw), 其中读写模块负责产生对 RAM IP 核读/写所需的所有数据、地址和读写使能信号, 同时将 RAM IP 读出的数据也

24.4.3 RAM 读写模块设计

首先介绍下 RAM 读写模块的设计，在 RAM 读写模块中，我们的输入信号主要有系统时钟信号、系统复位信号和 RAMIP 核输出的读数据（ram_rd_data），输出有控制 RAM 所需的 ram_en（RAM 端口使能）、ram_we（读写使能）、ram_addr（读写地址）和 ram_wr_data（写数据）这四个信号。由上述分析绘制出如下图所示的模块框图：

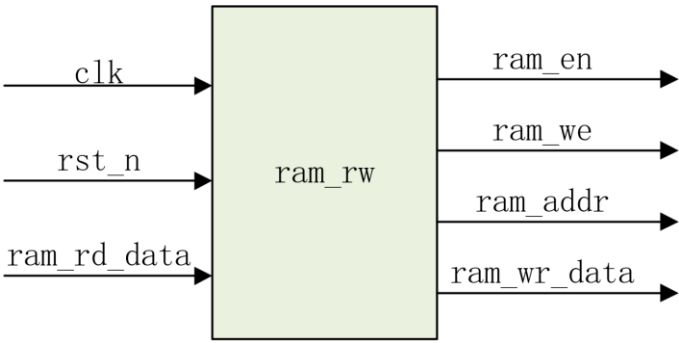


图 24.4.13 RAM 读写模块框图与端口

模块端口与功能描述如下表所示：

表 24.4.2 ram_rw模块端口与功能描述

信号名	位宽	方向	端口说明
clk	1	输入	系统时钟，50MHz
rst_n	1	输入	系统复位按键，低电平有效
ram_rd_data	8	输入	ram读数据
ram_en	1	输出	ram端口使能信号，高有效
ram_we	1	输出	ram读写使能信号，1为写，0为读
ram_addr	8	输出	ram读写地址
ram_wr_data	8	输出	ram写数据

绘制波形图

在编写代码之前，我们可以先绘制一个大致模块端口信号的波形图，其中时钟和复位信号是必不可少的，ram_en（RAM 端口使能信号）、ram_we（读写使能信号）、ram_addr（读写地址）、ram_wr_data（写数据）信号是我们使用 RAM 时要控制的信号，ram_rd_data 是 RAM 输出的数据，rw_cnt（读写计数器）是用来改变读写使能状态的，以读写 32 个数据为例，绘制出如下波形图：

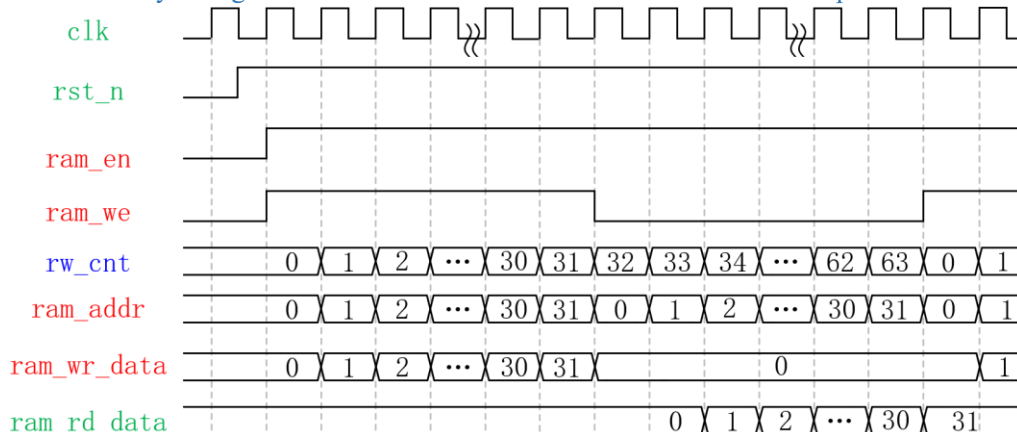


图 24.4.14 读写模块波形图

当系统复位结束后,我们便一直对 RAM 进行读写操作,所以 `ram_en` 在复位结束后一直处于使能状态。因为是读写 32 个数据,所以地址一直在 0~31 之间循环计数。接着我们用一个计数器 (`rw_cnt`) 循环计数 0~63,以此来对读写使能信号 (`ram_we`) 进行切换。当计数器值在 0~31 时进行写操作,写操作期间我们需要将 `ram_we` 置为高;当计数器值在 32~63 时进行读操作,读操作期间我们需要将 `ram_we` 置为低。

在写操作期间,写数据 (`ram_wr_data`) 由 0 开始累加 (步进为 1),直到累加值等于 31 (即写入 32 个数据) 时清零。而读数据只需要我们给出地址即可,需要注意的是读数据的输出会比地址晚一个时钟周期,当我们由读状态切换至写状态时,读数据会保持最后一次读出的数据。

代码编写

`ram_rw` 模块用于产生 RAM 读/写操作所需的信号,其代码如下所示:

```

1  module ram_rw(
2      input          clk          , //时钟信号
3      input          rst_n        , //复位信号,低电平有效
4
5      output reg      ram_en       , //ram 使能信号
6      output          ram_we       , //ram 读写选择
7      output reg [4:0] ram_addr    , //ram 读写地址
8      output reg [7:0] ram_wr_data, //ram 写数据
9      input  [7:0] ram_rd_data    //ram 读数据
10 );
11
12 //reg define
13 reg [5:0] rw_cnt ; //读写控制计数器
14
15 //*****
16 /**                               main code
17 //*****
18
19 //rw_cnt 计数范围在 0~31,写入数据;32~63 时,读出数据
20 assign ram_we = (rw_cnt <= 6'd31 && ram_en == 1'b1) ? 1'b1 : 1'b0;
21
22 //控制 RAM 使能信号

```



```
23 always @(posedge clk or negedge rst_n) begin
24     if(rst_n == 1'b0)
25         ram_en <= 1'b0;
26     else
27         ram_en <= 1'b1;
28 end
29
30 //读写控制计数器, 计数器范围 0~63
31 always @(posedge clk or negedge rst_n) begin
32     if(rst_n == 1'b0)
33         rw_cnt <= 6'b0;
34     else if(rw_cnt == 6'd63 && ram_en)
35         rw_cnt <= 6'b0;
36     else if(ram_en)
37         rw_cnt <= rw_cnt + 1'b1;
38     else
39         rw_cnt <= 6'b0;
40 end
41
42 //读写地址信号 范围: 0~31
43 always @(posedge clk or negedge rst_n) begin
44     if(rst_n == 1'b0)
45         ram_addr <= 5'b0;
46     else if(ram_addr == 5'd31 && ram_en)
47         ram_addr <= 5'b0;
48     else if (ram_en)
49         ram_addr <= ram_addr + 1'b1;
50     else
51         ram_addr <= 5'b0;
52 end
53
54 //在 WE 拉高期间产生 RAM 写数据, 变化范围是 0~31
55 always @(posedge clk or negedge rst_n) begin
56     if(rst_n == 1'b0)
57         ram_wr_data <= 8'b0;
58     else if(ram_wr_data < 8'd31 && ram_we)
59         ram_wr_data <= ram_wr_data + 1'b1;
60     else
61         ram_wr_data <= 8'b0 ;
62 end
63
64 endmodule
```

模块中定义了一个读写控制计数器 (rw_cnt)，当计数范围在 0~31 之间时，向 ram 的 0~31 地址写入数据 (0~31)；当计数范围在 32~63 之间时，从 ram 的 0~31 地址中读出数据。

24.4.4 仿真验证

编写 TB 文件

我们接下来先对代码进行仿真，因为本章实验我们只有系统时钟和系统复位这两个输入信号，所以仿真文件也只需要编写这两个信号的激励即可，TestBench 代码如下：

```

1  `timescale 1ns / 1ps          //仿真单位/仿真精度
2
3  module tb_ip_lport_ram();
4
5  //parameter define
6  parameter CLK_PERIOD = 20; //时钟周期 20ns
7
8  //reg define
9  reg      sys_clk;
10 reg      sys_rst_n;
11
12 //信号初始化
13 initial begin
14     sys_clk = 1'b0;
15     sys_rst_n = 1'b0;
16     #200
17     sys_rst_n = 1'b1;
18 end
19
20 //产生时钟
21 always #(CLK_PERIOD/2) sys_clk = ~sys_clk;
22
23 ip_lport_ram u_ip_lport_ram(
24     .sys_clk      (sys_clk      ),
25     .sys_rst_n    (sys_rst_n    )
26 );
27
28 endmodule

```

仿真验证

通过仿真我们得到了以下波形图：

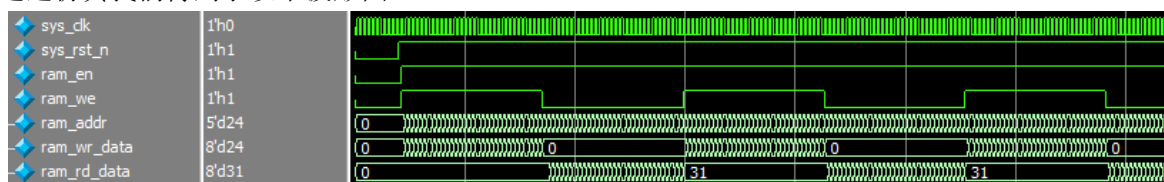


图 24.4.15 RAM 仿真波形图

可以看出我们的波形是满足我们所选择的不变模式的，即读写不同时进行。接下来我们展开波形图，分别看一下写操作与读操作时的波形。

写操作波形图如下所示：

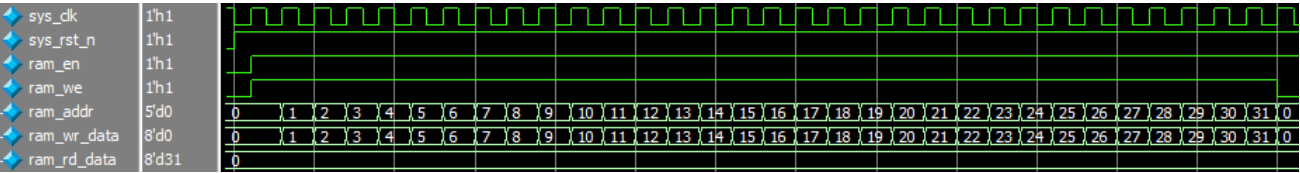


图 24.4.16 写操作仿真波形图

由上图可知，ram_wea 信号拉高，说明此时是对 ram 进行写操作。ram_wea 信号拉高之后，地址和数据都是从 0 开始累加，也就是说当 ram 地址为 0 时，写入的数据也是 0；当 ram 地址为 1 时，写入的数据也是 1，我们总共向 ram 中写入 32 个（即 0~31）数据。

读操作波形图如下所示：

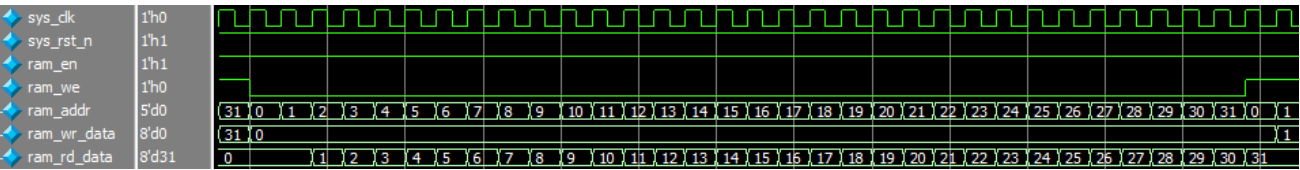


图 24.4.17 读操作仿真波形图

由上图可知，ram_wea 信号拉低，说明此时是对 ram 进行读操作。ram_wea 信号拉低之后，ram_addr 从 0 开始增加，即从 ram 的地址 0 开始读数据；ram_rd_data 在给出地址的下一个时钟周期后，开始输出数据，输出的数据为 0、1、2……30、31，和我们写入的值是相等的，也就是说，我们创建的 RAM IP 核从仿真结果上来看是正确的。

24.5 下载验证

引脚约束

在仿真验证完成后，接下来对引脚进行分配，并上板验证。本实验中，系统时钟、按键复位的管脚分配如下表所示：

表 24.5.1 IP核之单端口RAM实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50MHz	LVC MOS33
sys_rst_n	input	N16	系统复位按键，低电平有效	LVC MOS33

对应的 XDC 约束语句如下所示：

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVC MOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN N16 IOSTANDARD LVC MOS33} [get_ports sys_rst_n]
```

Vivado 软件中 IO Planning 界面如下图所示：



Scalar ports (2)															
 sys_clk	IN		U18	▼	✓	34	LVC MOS33*	▼	3.300			NONE	▼	NONE	▼
 sys_rst_n	IN		N16	▼	✓	35	LVC MOS33*	▼	3.300			NONE	▼	NONE	▼

图 24.5.1 IO Planning 引脚约束界面

添加 ila IP 核进行在线调试

接下来添加 ILA IP 核, 将 ram_en (1 位)、ram_wea (1 位)、ram_addr (5 位)、ram_wr_data (8 位) 和 ram_rd_data (8 位) 信号添加至观察列表中, 添加 ILA IP 核的方法这里就不再赘述了, 本例程是将 ILA 例化在了 ram_rw 模块中, 例化代码如下所示:

```
ila_0 u_ila_0 (
    .clk(clk),           // input wire clk

    .probe0(ram_en),     // input wire [0:0] probe0
    .probe1(ram_we),     // input wire [0:0] probe1
    .probe2(ram_addr),   // input wire [4:0] probe2
    .probe3(ram_wr_data), // input wire [7:0] probe3
    .probe4(ram_rd_data) // input wire [7:0] probe4
);
```

上板验证

编译工程并生成比特流文件后, 将下载器一端连接电脑, 另一端与开发板上的 JTAG 下载口连接, 连接电源线, 并打开开发板的电源开关。

点击 Vivado 左侧 “Flow Navigator” 窗口最下面的 “Open Hardware Manager”, 如果此时 Vivado 软件识别到了下载器, 则点击 “Hardware” 窗口中 “Program Device”, 在弹出的界面中选择 “Program” 下载程序。

RAM 写操作在 ILA 中观察到的波形如下图所示:

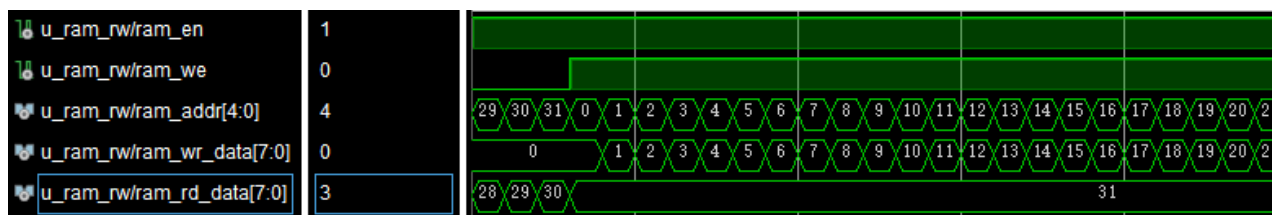


图 24.5.2 写操作 ILA 波形图

由上图可知, ram_wea 信号拉高之后, 地址和数据都从 0 开始累加, 也就是说当 ram 地址为 0 时, 写入的数据也是 0; 当 ram 地址为 1 时, 写入的数据也是 1。可以发现, 写操作的验证结果与我们的仿真波形是一致的。

接着我们再来看一下 RAM 读操作在 ILA 中观察到的波形, 如下图所示:

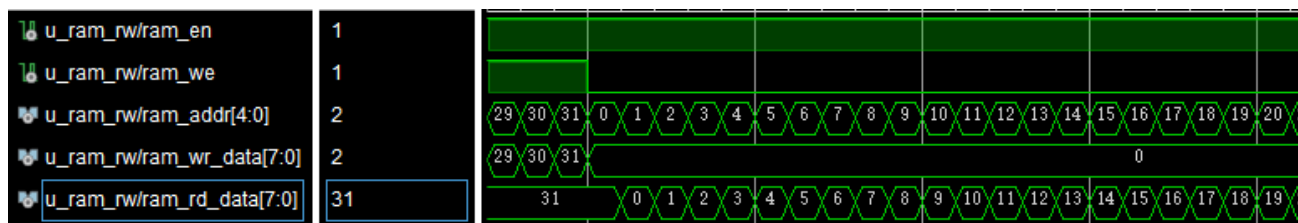


图 24.5.3 读操作 ILA 波形图

由上图可知, ram_wea 信号拉低之后, ram_addr 从 0 开始增加, 即从 ram 的地址 0 开始读数据; ram_rd_data 在给出地址的下一个时钟周期后, 开始输出数据, 输出的数据为 0, 1, 2.....30, 31, 和我们写入的值是相等的。我们可以发现, 读操作的验证结果和仿真的波形也是一致的, 至此说明了 IP 核之单端口 RAM 实验验证成功。

24.6 本章总结

本章节我们主要介绍了存储器的分类、静态 RAM 的种类和特性,重点讲解了 Vivado 软件中如何将 BGM IP 核配置成单端口 RAM 的方法,并调用其进行读写操作,读者要牢记于心,这对后面章节的学习至关重要,能大大提高大家的开发效率。IP 核调用(例化)格式如下所示:

```
blk_mem_gen_0 blk_mem_gen_0 (  
    .clka  (sys_clk      ), // input wire clka  
    .ena   (ram_en       ), // input wire ena  
    .wea   (ram_wea      ), // input wire [0 : 0] wea  
    .addra (ram_addr     ), // input wire [4 : 0] addra  
    .dina  (ram_wr_data  ), // input wire [7 : 0] dina  
    .douta (ram_rd_data  )  // output wire [7 : 0] douta  
);
```

24.7 拓展训练

在经过本章的学习后,相信大家已经掌握了单端口 RAM 的使用方法,我们可以尝试进行以下拓展训练:

- 1、BMG IP 核在未来的工程项目中会频繁运用,大家可以在课后多多熟悉该 IP 核的使用。
- 2、大家可以提前尝试一下如何实现双端口 RAM 的读写,这部分内容我们会在下一章进行讲解。
- 3、如果大家想了解 ROM 的详细创建,可以看一下“RGB LCD 字符和图片显示实验”章节的讲解。

第二十五章 IP 核之双端口 RAM 实验

上一章我们学习了单端口 RAM 的概念以及 BMG IP 核配置成单端口 RAM 的方法, 并成功实现了单端口 RAM 的读写操作, 本章我们将带着大家继续学习双端口 RAM 的概念以及如何将 BMG IP 核配置成双端口 RAM, 并对双端口 RAM 进行简单的读写测试。

本章包括以下几个部分:

25.1 RAM IP 核简介

25.2 实验任务

25.3 程序设计

25.4 下载验证

25.5 本章总结

25.6 拓展训练

25.1 RAM IP 核简介

双端口 RAM 是指拥有两个读写端口的 RAM, 有伪双端口 RAM (一个端口只能读, 另一个端口只能写) 和真双端口 RAM (两个端口都可以进行读写操作) 之分。一般当我们需要同时对存储器进行读写操作时会使用到双端口 RAM, 例如有一个 FIFO 存储器, 我们需要同时对其进行数据的写入和读出, 这时候就需要一个写端口和一个读端口了。接下来我们看下双端口 RAM 的框图。

在上一章中我们介绍过了单端口 RAM 的框图, 本章将带着大家一起了解一下双端口 RAM 的框图, 为了方便大家进行对比, 这里我们将上一章介绍的单端口 RAM 的框图也展示出来, 如下图所示:

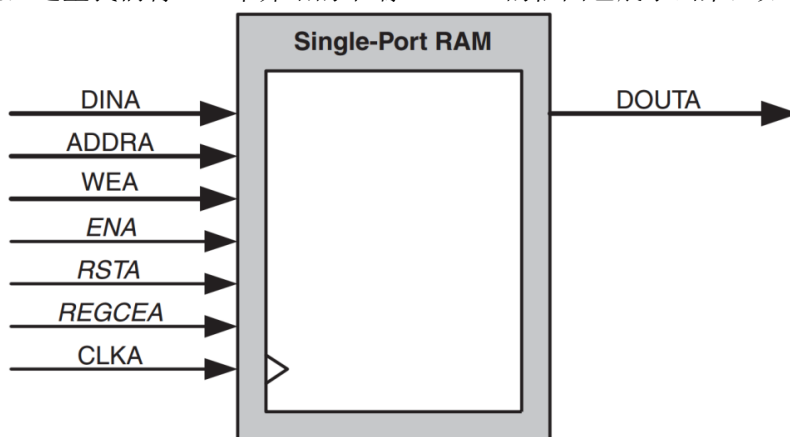


图 25.1.1 单端口 RAM 框图

首先介绍的是简单双端口 (也称为伪双端口) RAM, 需要注意的是简单双端口 RAM 的端口 A 只能写不能读, 端口 B 只能读不能写。BMG IP 核配置成简单双端口 RAM 的框图如下图所示。

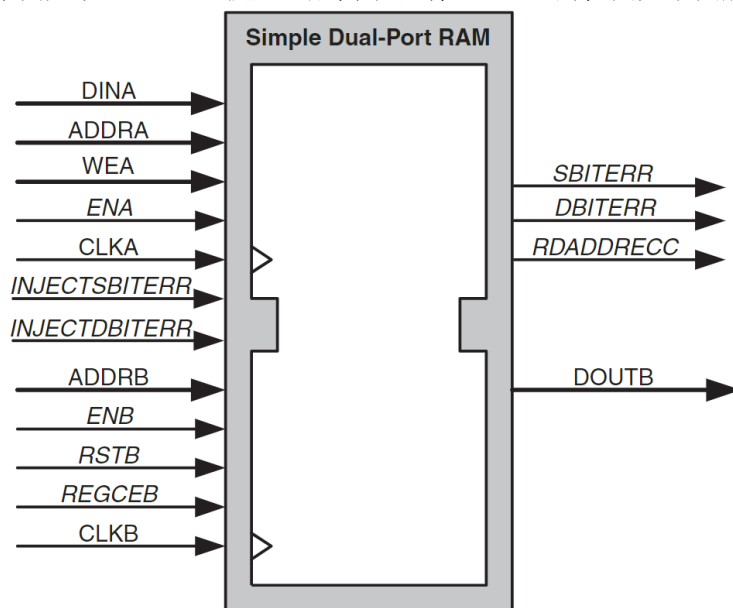


图 25.1.2 简单 (伪) 双端口 RAM 框图

与单端口 RAM 不同的是, 伪双端口 RAM 输入有两路时钟信号 CLKA/CLKB; 独立的两组地址信号 ADDR A/ADDRB; Port A 仅提供 DINA 写数据总线, 作为数据的写入口; Port B 仅提供数据读的功能, 读出的数据为 DOUTB。这里我们仅对新出现的信号进行讲解, 其它信号在单端口 RAM 中已经讲解过了, 其中不同端口的同名 (同功能) 信号以 A 和 B 做为区分, 各个新端口 (这些信号很少使用, 我们一般不用关注) 的功能描述如下:

INJECTSBITERR: Inject Single-Bit Error 的简写, 即注入单 bit 错误, 仅适用于 Xilinx Zynq-7000 和 7 系

列芯片的 ECC 配置。

INJECTDBITERR: Inject Double-Bit Error 的简写, 即注入双 bit 错误, 同样仅适用于 Xilinx Zynq-7000 和 7 系列芯片的 ECC 配置。

SBITERR: Single-Bit Error 的简写, 即单 bit 错误, 标记内存中存在的单 bit 错误, 该错误已在输出总线上自动更正。

DBITERR: Double-Bit Error 的简写, 即双 bit 错误, 标记内存中存在双 bit 错误, 需要注意的是内置的 ECC 解码模块不能自动纠正双 bit 错误。

RDADDRECC: Read Address for ECC Error output 的简写, 即读地址 ECC 错误输出, 同样仅适用于 Xilinx Zynq-7000 和 7 系列芯片的 ECC 配置。

接着介绍一下真双端口 RAM, 其两个端口 (A 和 B) 均可进行读/写操作。BMG IP 核配置成真双端口 RAM 的框图如下图所示。

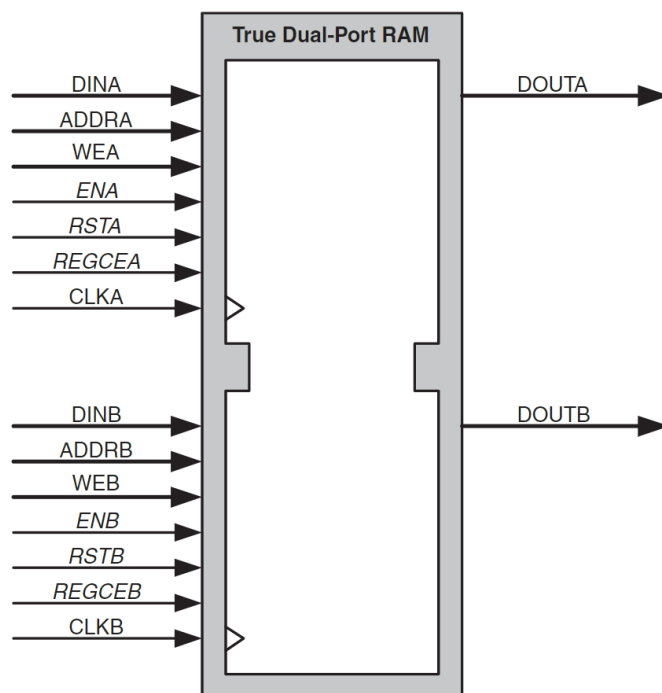


图 25.1.3 真双端口 RAM 框图

真双端口 RAM 提供了两个独立的读写端口 (A 和 B), 既可以同时读, 也可以同时写, 也可以一个读一个写。通过框图对比可以发现, 真双端口 RAM 只是将单端口 RAM 的所有信号做了一个复制处理, 不同端口的同一信号以 A 和 B 作为区分, 所以这里我们就不再赘述各个端口的功能了。

下方我们列出了三种静态 RAM 的端口对比表, 让大家能更直观的看出各静态 RAM 的端口差异, 其中“√”表示有, “×”表示无。

表 25.1.1 静态RAM端口对比表

端口名	对应中文名	方向	单端口 RAM	伪双端口 RAM	真双端口 RAM
DINA	端口A写数据信号	输入	√	√	√
ADDRA	端口A地址信号	输入	√	√	√
WEA	端口A写使能信号	输入	√	√	√
ENA	端口A端口使能信号	输入	√	√	√
RSTA	端口A复位信号	输入	√	×	√
REGCEA	端口A输出寄存器使能信号	输入	√	×	√
CLKA	端口A时钟信号	输入	√	√	√
DOUTA	端口A读数据信号	输出	√	×	√
DINB	端口B写数据信号	输入	×	×	√
ADDRB	端口B地址信号	输入	×	√	√
WEB	端口B写使能信号	输入	×	×	√
ENB	端口B端口使能信号	输入	×	√	√
RSTB	端口B复位信号	输入	×	√	√
REGCEB	端口B输出寄存器使能信号	输入	×	√	√
CLKB	端口B时钟信号	输入	×	√	√
DOUTB	端口B读数据信号	输出	×	√	√
INJECTSBITERR	注入单bit错误	输入	×	√	×
INJECTDBITERR	注入双bit错误	输入	×	√	×
SBITERR	单bit纠错	输出	×	√	×
DBITERR	双bit检错	输出	×	√	×
RDADDRECC	读地址ECC错误输出	输出	×	√	×

通过对比我们可以发现无论是哪种双端口 RAM, 其地址线、时钟线和使能线等控制信号都有两组, 所以双端口 RAM 可以实现在不同时钟域下的读/写, 且可以同时对不同的地址进行读/写, 这便大大提高了我们数据处理的灵活性。但是两组信号线也相应的加大了双端口 RAM 的使用难度, 因为端口使能, 读写使能, 地址和写数据等控制信号我们都需要分别给出两组, 这样才能驱使两个端口都处于我们需要的工作状态, 这里仅凭文字描述大家理解起来可能会有些吃力, 所以在稍后的小节中我们会结合波形图(图 25.3.9)进行更详细的讲解。

需要注意的是在伪双端口模式下我们需要避免读写冲突; 在真双端口模式下我们需要避免读写冲突和

写写冲突。下面我们分别看下读写冲突和写写冲突是什么。

1、读写冲突: 即同时刻读写同一地址所出现的冲突, 例如理论上我们已经向某个地址写入了新的数据, 我们也希望可以同时读到这个地址内新写入的数据, 但实际上, 这个新数据还没有写入 RAM 中, 所以我们读出来的可能是 RAM 默认值, 或者是 RAM 该地址中上一次的值, 这便是读写冲突。读写冲突示意图如下所示:

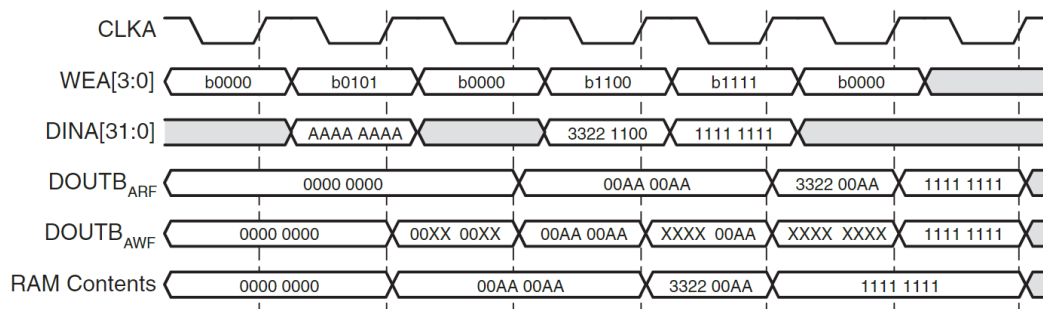


图 25.1.4 RAM 读写冲突示意图

由上图可知当发生读写冲突时, 读优先的模式下读出的是读地址中存储的上一个数据; 写优先模式下读出的是未知的数据“XX”。

2、写写冲突: 表示两个端口写使能同时有效且写地址相同, 此时需要关断一个写, 把两个写端口都需要更新的值处理到一个写端口上。切记任何双端口 RAM 都不支持写写冲突。写写冲突示意图如下所示:

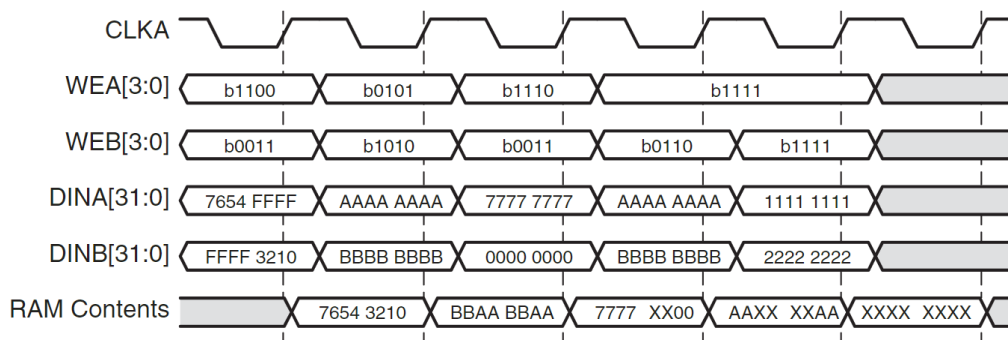


图 25.1.5 RAM 写写冲突示意图

由上图可知当发生写写冲突时, 发生冲突的地址写入的是未知的数据“XX”。

综上所述, 真双端口 RAM 的读写更为灵活, 但相应的也更加难以驾驭, 因为真双端口不仅需要考虑读写冲突, 还要考虑写写冲突, 而在大部分设计中我们用的都是伪双端口 RAM, 所以本章我们以伪双端口 RAM 的读写为例, 讲解一下如何避免程序中的读写冲突。一般发生读写冲突的时候, 我们可以通过错开读写地址的方法来避免读写冲突, 我们在本次实验中所使用的就是这种方法, 在后文中会进行讲解; 或者通过写穿通到读方法来处理冲突数据, 这里我们简单做下讲解, 如下图所示:

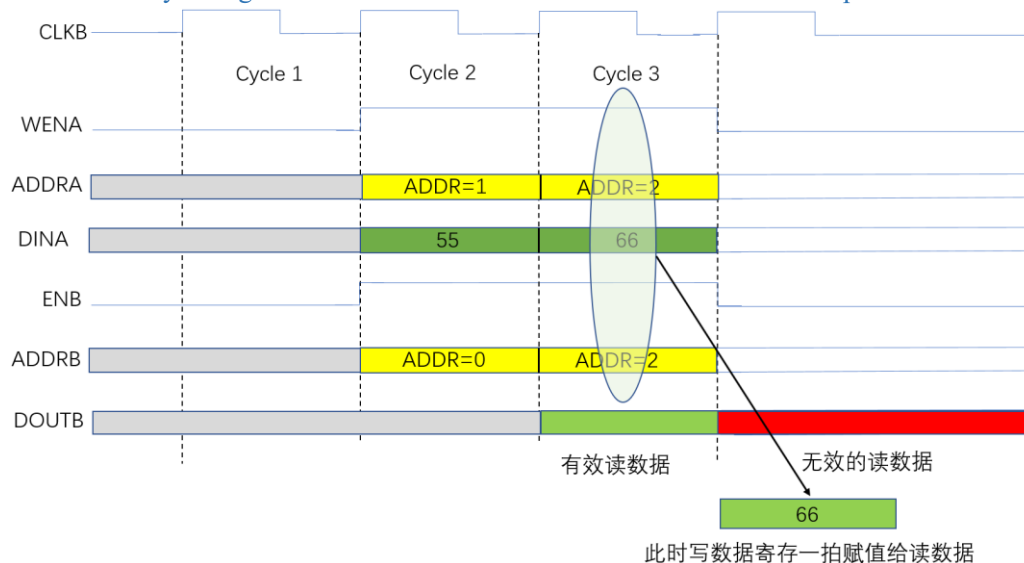


图 25.1.6 RAM 读写冲突处理

从图中我们可以看出, Cycle 2 时, 读和写地址不同, 读可以正常读到数据, 但是到 Cycle 3 时, 读和写地址相同且读写都有效, 此处如果不做特殊处理, 那么读数据是无效的。需要我们把写数据寄存一拍同步到读侧, 即把最新的写数据直接赋给读数据, 这便是写穿通到读。

25.2 实验任务

本章实验任务是将 Xilinx BMG IP 核配置成一个同步的伪双端口 RAM 并对其进行读写操作, 然后通过仿真观察波形是否正确, 最后将设计下载到 FPGA 开发板中, 并通过在线调试工具对实验结果进行观察。

25.3 程序设计

25.3.1 RAM IP 核配置

首先我们创建一个名为 “ip_2port_ram” 的空白工程, 然后点击 Vivado 软件左侧 “Flow Navigator” 栏中的 “IP Catalog”, 如下图所示:

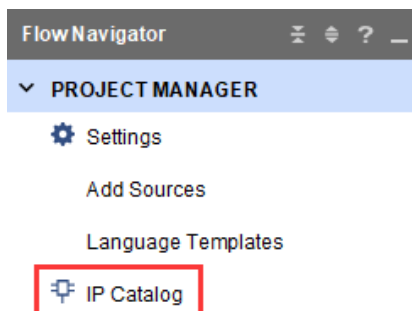


图 25.3.1 点击 “IP Catalog”

在 “IP Catalog” 窗口的搜索栏中输入 “Block Memory” 关键字后, 出现唯一匹配的 “Block Memory Generator”, 如下图所示 (图中出现的两个 IP 核为同一个 BMG IP 核):

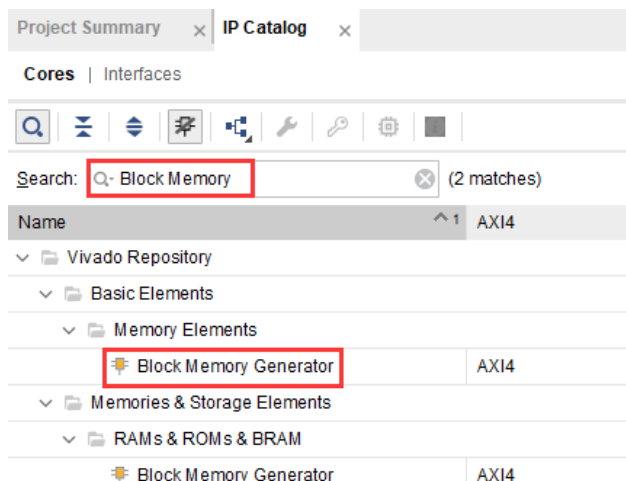


图 25.3.2 搜索栏中输入“Block Memory”关键字

双击“Block Memory Generator”后弹出 IP 核的配置界面，接着我们就可以对 BMG IP 核进行配置了，“Basic”选项卡配置界面如下图所示。

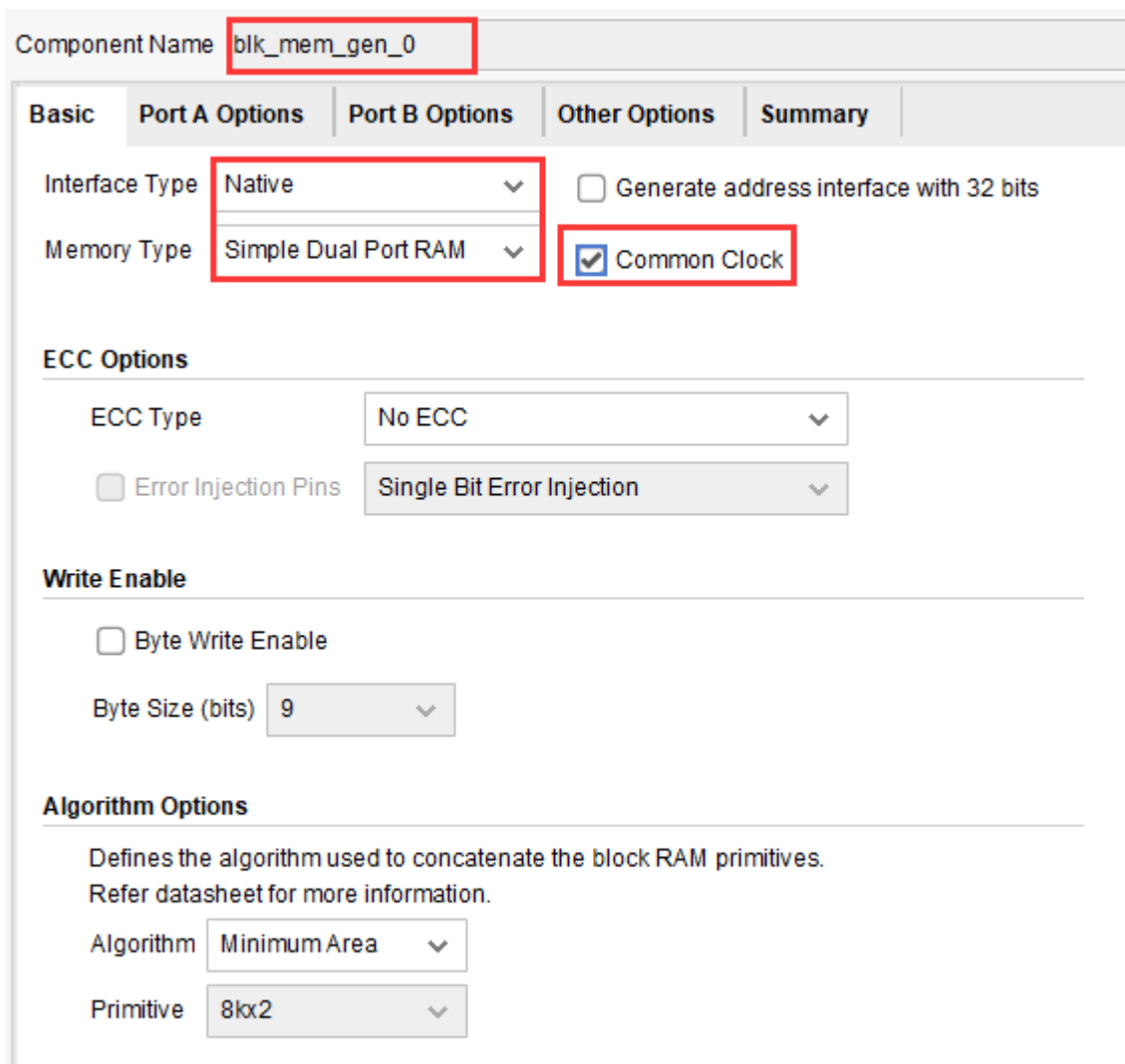


图 25.3.3 “Basic”选项卡配置界面

最上面的“Component Name”一栏可以设置该 IP 元件的名称，这里我们保持默认命名，当然也可以命

名为其它方便自己一眼看出其功能的名称。

接着目光回到“Basic”选项卡上,该选项卡下各参数含义我们在上一章已经做过详细的讲解,本章就不再赘述了。因为本章是创建一个同步的伪双端口 RAM,所以“Memory Type (存储类型)”我们选择“Simple Dual Port RAM (伪双端口 RAM)”,并勾选“Common Clock (同步时钟)”选项,其余设置保存默认即可。下面我们对“ECC Options”做一下扩充讲解,感兴趣的同学可以看一下。

当存储类型 (Memory Type) 设置为伪双端口时才可以用 ECC, ECC 的主要作用是单 bit 纠错和双 bit 检错,在启用后我们可以在写操作期间将单 bit (Single Bit Error Injection) 或双 bit (Double Bit Error Injection) 错误注入到指定位置。这里我们不需要注入错误码,所以保存默认选项“No ECC”即可。

接下来我们对“Port A Options”和“Port B Options”选项卡进行配置,如下图所示:

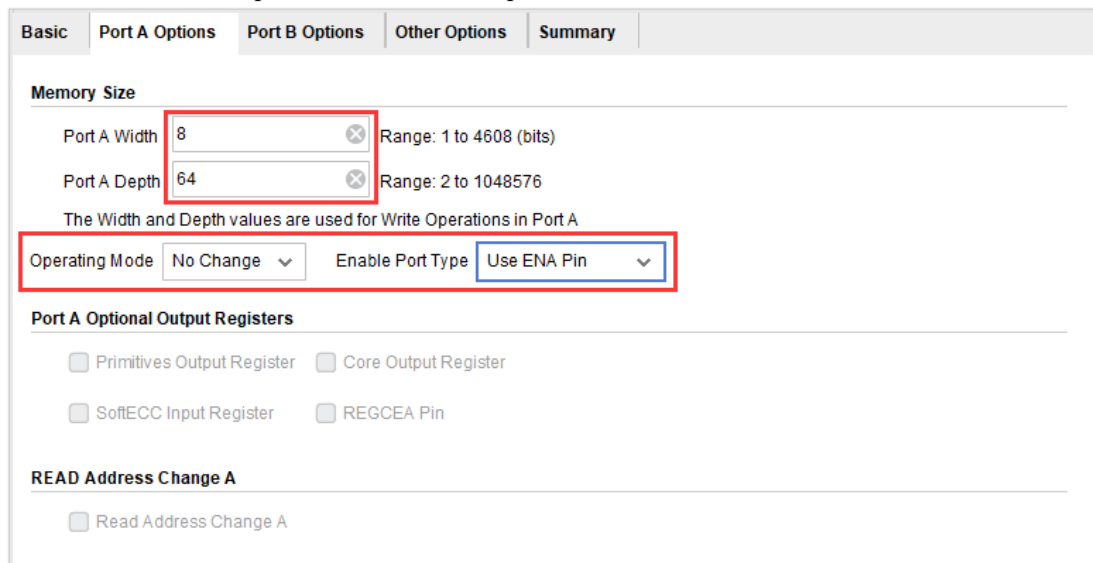


图 25.3.4 “Port A Options”选项卡配置界面

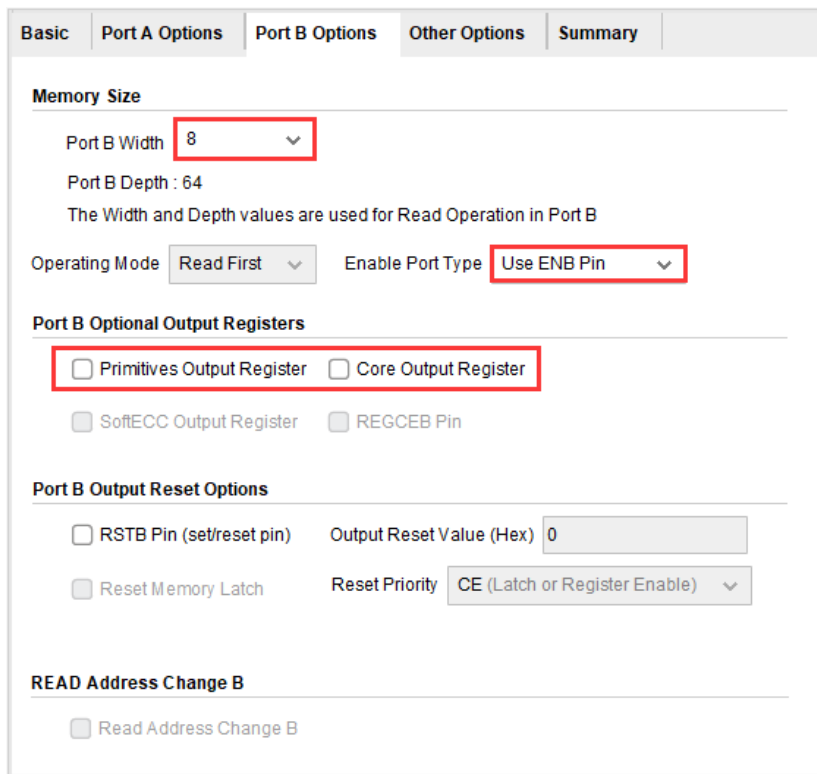


图 25.3.5 “Port B Options”选项卡配置界面

“Port A Options”和“Port B Options”选项卡下各参数含义在上一章中我们已经做过详细的讲解了,还不熟悉的同学可以回顾一下上一章的内容。

需要注意的是,只有端口 A 的写数据位宽和写深度是可以任意配置的。端口 B 的读数据位宽必须与端口 A 的写数据位宽存在比例关系(上一节中已讲解过支持的比例关系);端口 B 的读深度是当端口 A 的写数据位宽、端口 A 的写深度和对端口 B 的读数据位宽确定后,自动确定的。

接下来是“Other Options”选项卡,同上一章一样,该选项卡无需配置,保存默认即可。

最后一个是“Summary”选项卡,该界面显示了我们配置的存储器的类型,消耗的 BRAM 资源等信息,我们直接点击“OK”按钮完成 BMG IP 核的配置,如下图所示:

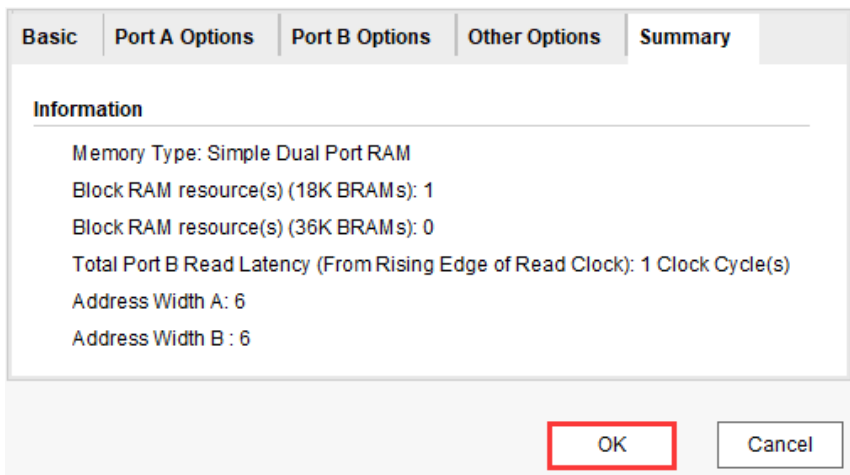


图 25.3.6 “Summary”界面

接着就弹出了“Generate Output Products”窗口,我们直接点击“Generate”即可,如下图所示:

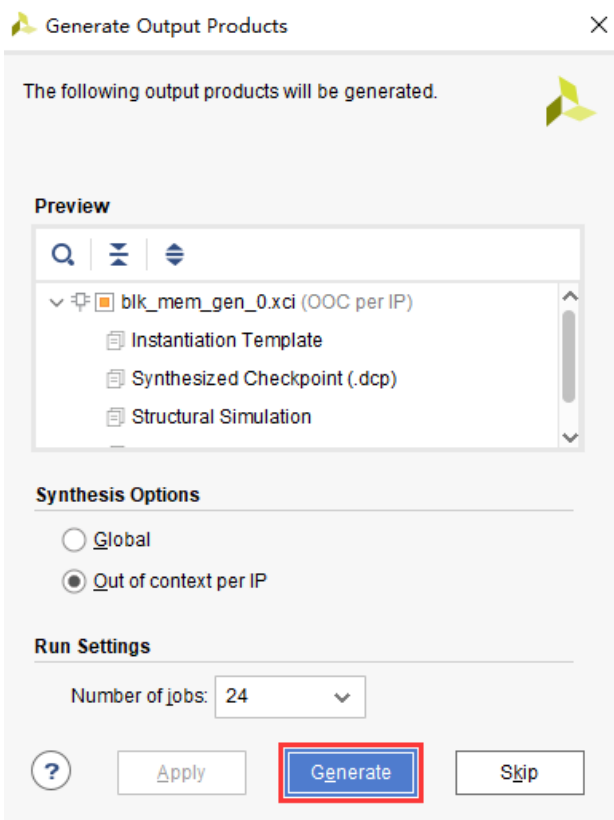


图 25.3.7 “Generate Output Products”窗口

接下来我们就可以开始编写代码来调用我们设置好的 IP 核了,有关代码部分的讲解,请看后续小

节的内容。

25.3.2 顶层模块设计

本次实验的目的是为了将 Xilinx BMG IP 核配置成一个伪双端口 RAM 并对其进行读写操作，因此可以给模块命名为 ip_2port_ram。因为伪双端口的数据线，地址线及其他信号线都是相互独立的，所以这里我们将读写分为两个子模块，分别命名为 ram_rd（读模块）和 ram_wr（写模块）；系统时钟和系统复位信号大家已经很熟悉了，这里就不再多讲了；本次的实验结果我们仍是通过在线调试工具进行观察，所以没有需要输出到 IO 上的信号。由上述分析我们可以画出一个大致的模块框图，如下图所示：

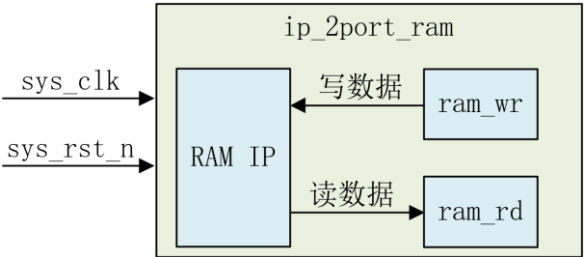


图 25.3.8 模块框图与端口

模块端口与功能描述如下表所示：

表 25.3.1 ip_2port_ram模块端口与功能描述

信号名	位宽	方向	端口说明
sys_clk	1	输入	系统时钟，50MHz
sys_rst_n	1	输入	系统复位按键，低电平有效

绘制波形图

首先我们梳理一下顶层模块中各个信号的波形变化。因为本次实验中的信号皆为内部信号，所以为了方便区分，我们用绿色标记端口 A（写端口）相关的信号名，用红色标记端口 B（读端口）的信号名，用蓝色标记两个端口之间的交互信号。这里我们以读写 64 个（0~63）数据为例，绘制出如下波形图：

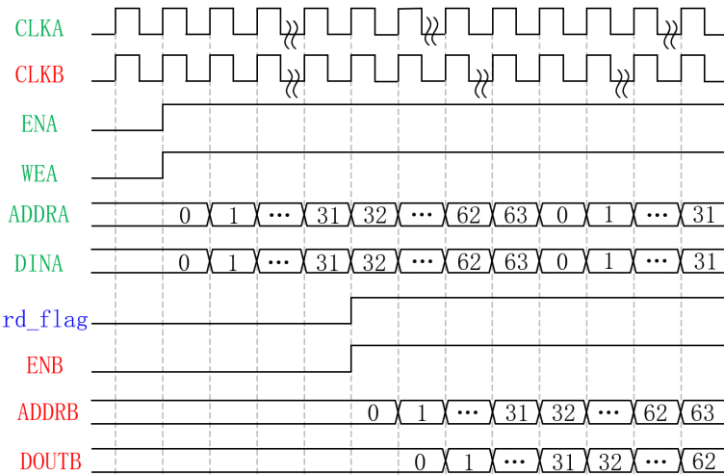


图 25.3.9 顶层模块信号波形图

由上图可以看出，除了和伪双端口有关的信号外，我们还引入了一个读启动信号（rd_flag），该信号的作用是错开两个端口的启动时间，以此达到错开读写地址，防止读写冲突的目的。

回到波形图上，为了能读出有效数据，我们先使能了端口 A，因为端口 A 只能进行写操作，且我们也

需要它一直向 RAM 中写入数据, 所以当端口 A 使能后, 我们就一直拉高 WEA 信号, 让其一直处于写状态, 并从 0 开始累加地址, 向 0~63 地址中循环写入数据。当写完一半的数据时, 我们通过拉高 rd_flag 信号来使能端口 B, 让端口 B 开始工作, 并从 0 地址开始循环读出 0~63 地址内存储的数据。因为启动两个端口的时间存在误差, 且读写的时钟速率相同, 所以读写地址就会相应的错开, 以此达到避免读写冲突的目的。

这里有几点需要大家注意:

1、伪双端 RAM 时, 因为端口 A 只能写不能读, 所以 WEA 可以理解为端口 A 的写使能信号 (高有效)。

2、哪怕我们只错开一个地址, 也是可以避免读写冲突的, 这里我们在写完一半的数据时拉高 rd_flag 信号, 只是为了方便我们进行观察而多错开了一些地址而已。

3、端口 B 只能读, 其没有像 WEA 一样的读写使能信号, 所以当端口使能后, 便一直处于读状态。

编写代码

因为本次实验除了调用 BMG IP 核外还需要例化一个读模块 (ram_rd) 和一个写模块 (ram_wr), 所以我们需要创建一个顶层模块来例化 IP 核与读/写模块, 这里我们可以将顶层模块命名为 ip_2port_ram, 代码如下:

```

1 module ip_2port_ram(
2     input      sys_clk    ,    //系统时钟
3     input      sys_rst_n   //系统复位, 低电平有效
4 );
5
6 //wire define
7 wire          ram_wr_en   ; //端口 A 使能
8 wire          ram_wr_we   ; //ram 端口 A 写使能
9 wire          ram_rd_en   ; //端口 B 使能
10 wire         rd_flag     ; //读启动标志
11 wire [5:0]    ram_wr_addr; //ram 写地址
12 wire [7:0]    ram_wr_data; //ram 写数据
13 wire [5:0]    ram_rd_addr; //ram 读地址
14 wire [7:0]    ram_rd_data; //ram 读数据
15
16 //*****
17 /**                                main code
18 //*****
19
20 //RAM 写模块
21 ram_wr u_ram_wr(
22     .clk          (sys_clk    ),
23     .rst_n        (sys_rst_n  ),
24
25     .rd_flag      (rd_flag    ),
26     .ram_wr_en    (ram_wr_en  ),
27     .ram_wr_we    (ram_wr_we  ),
28     .ram_wr_addr  (ram_wr_addr),

```

```
29     .ram_wr_data    (ram_wr_data)
30 );
31
32 //简单双端口 RAM
33 blk_mem_gen_0 u_blk_mem_gen_0 (
34     .clka    (sys_clk    ), // input wire clka
35     .ena     (ram_wr_en  ), // input wire ena
36     .wea     (ram_wr_we  ), // input wire [0 : 0] wea
37     .addra   (ram_wr_addr), // input wire [5 : 0] addra
38     .dina    (ram_wr_data), // input wire [7 : 0] dina
39     .clkb    (sys_clk    ), // input wire clkb
40     .enb     (ram_rd_en  ), // input wire enb
41     .addrb   (ram_rd_addr), // input wire [5 : 0] addrb
42     .doutb   (ram_rd_data) // output wire [7 : 0] doutb
43 );
44
45 //RAM 读模块
46 ram_rd u_ram_rd(
47     .clk        (sys_clk    ),
48     .rst_n      (sys_rst_n  ),
49
50     .rd_flag     (rd_flag    ),
51     .ram_rd_en   (ram_rd_en  ),
52     .ram_rd_addr (ram_rd_addr),
53     .ram_rd_data (ram_rd_data)
54 );
55
56 endmodule
```

可以看出 ip_2port_ram 顶层模块只是例化了 IP 核 (blk_mem_gen_0)、读模块 (ram_rd) 和写模块 (ram_wr)，其中写模块负责产生 RAM IP 核写操作所需的所有数据、地址、写使能信号以及启动读模块的标志信号 (rd_flag)；读模块负责产生 RAM IP 核读操作所需的地址，并将读出的数据也连接至读模块。

25.3.3 RAM 写模块设计

首先介绍下 RAM 写模块的设计，在 RAM 写模块中，我们的输入信号主要有系统时钟信号和系统复位信号；输出有控制写 RAM 所需的 ram_wr_en（写端口使能）、ram_wr_we（写使能）、ram_wr_addr（写地址）和 ram_wr_data（写数据）这四个信号，以及控制读模块启动 rd_flag（读启动）信号。由上述分析绘制出如下图所示的模块框图：

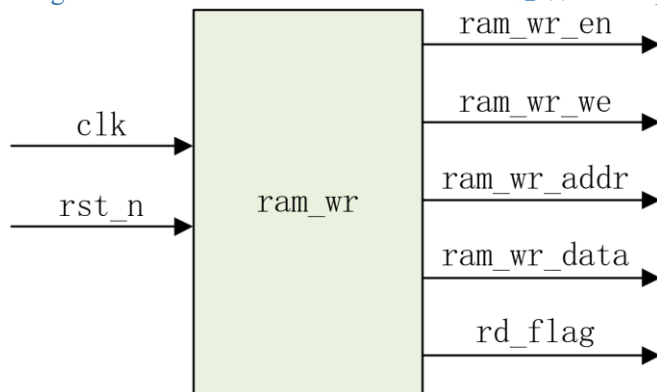


图 25.3.10 RAM 写模块框图

模块端口与功能描述如下表所示:

表 25.3.2 ram_wr 模块端口与功能描述

信号名	位宽	方向	端口说明
clk	1	输入	系统时钟, 50MHz
rst_n	1	输入	系统复位按键, 低电平有效
ram_wr_en	1	输出	ram写端口使能信号, 高有效
ram_wr_we	1	输出	ram写使能信号, 高有效
ram_wr_addr	8	输出	ram写地址
ram_wr_data	8	输出	ram写数据
rd_flag	1	输出	读启动信号, 高有效

绘制波形图

在编写代码前, 我们先大致梳理一下模块的端口时序, 并绘制出如下波形图:

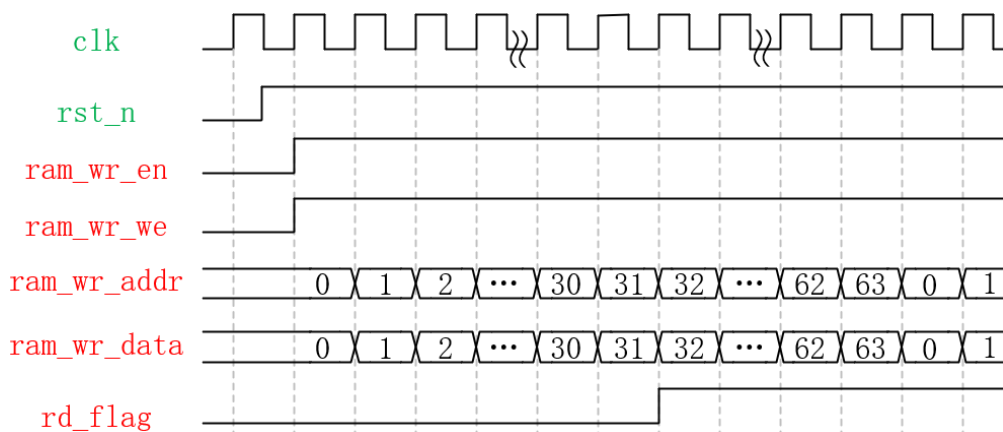


图 25.3.11 RAM 写模块波形图

因为我们需要一直向 RAM 中写入数据, 所以当复位结束后, 我们就将 ram_wr_en (写端口使能) 和 ram_wr_we (ram 写使能) 一直置为高。当写使能拉高后, 写地址一直在 0~63 之间循环计数, 并向对应的 RAM 地址中写入数据, 当写地址第一次计数到 31 时, 将 rd_flag 信号拉高并保持, 以启动读模块进行读操作。

代码编写

ram_wr 模块用于产生 RAM 写操作所需的信号以及读启动信号, 其代码如下所示:

```
1 module ram_wr(
2     input          clk          , //时钟信号
3     input          rst_n        , //复位信号, 低电平有效
4
5     //RAM 写端口操作
6     output          ram_wr_we    , //ram 写使能
7     output reg      ram_wr_en    , //端口使能
8     output reg      rd_flag      , //读启动信号
9     output reg [5:0] ram_wr_addr , //ram 写地址
10    output [7:0] ram_wr_data     //ram 写数据
11 );
12
13 //*****
14 /**                                main code
15 //*****
16
17 //ram_wr_we 为高电平表示写数据
18 assign ram_wr_we = ram_wr_en ;
19
20 //写数据与写地址相同, 因位宽不等, 所以高位补 0
21 assign ram_wr_data = {2'b0, ram_wr_addr} ;
22
23 //控制 RAM 使能信号
24 always @(posedge clk or negedge rst_n) begin
25     if(!rst_n)
26         ram_wr_en <= 1'b0;
27     else
28         ram_wr_en <= 1'b1;
29 end
30
31 //写地址信号 范围:0~63
32 always @(posedge clk or negedge rst_n) begin
33     if(!rst_n)
34         ram_wr_addr <= 6'd0;
35     else if(ram_wr_addr < 6'd63 && ram_wr_we)
36         ram_wr_addr <= ram_wr_addr + 1'b1;
37     else
38         ram_wr_addr <= 6'd0;
39 end
40
```

```
41 //当写入 32 个数据 (0~31) 后, 拉高读启动信号
42 always @(posedge clk or negedge rst_n) begin
43     if(!rst_n)
44         rd_flag <= 1'b0;
45     else if(ram_wr_addr == 6'd31)
46         rd_flag <= 1'b1;
47     else
48         rd_flag <= rd_flag;
49 end
50
51 endmodule
```

因为写模块要循环向 ram 的 0~63 地址中写入数据, 所以当写端口使能后, 写使能信号就一直为高, 如代码第 18 行所示。

当写使能信号拉高后, 写地址就会在 0~63 之间循环计数 (如代码第 32~39 行所示), 并向 RAM 中写入相应的数据 (如代码第 21 行所示)。

在第 42~49 行代码中, 为了避免读写冲突, 我们引入了读启动信号 (rd_flag), 当地址第一次计数到 31 时, 拉高 rd_flag 信号, 使读模块开始工作, 这样做就可以将读地址和写地址错开, 防止在同一时刻对同一地址进行读和写。

25.3.4 RAM 读模块设计

首先介绍下 RAM 读模块的设计, 在 RAM 读模块中, 我们的输入信号主要有系统时钟信号、系统复位信号、从 RAM 中读出的数据 (ram_rd_data) 以及我们自己定义的读启动标志信号 (rd_flag); 输出有控制读 RAM 所需的 ram_rd_en (读端口使能) 和 ram_rd_addr (读地址) 这两个信号。由上述分析绘制出如下图所示的模块框图:

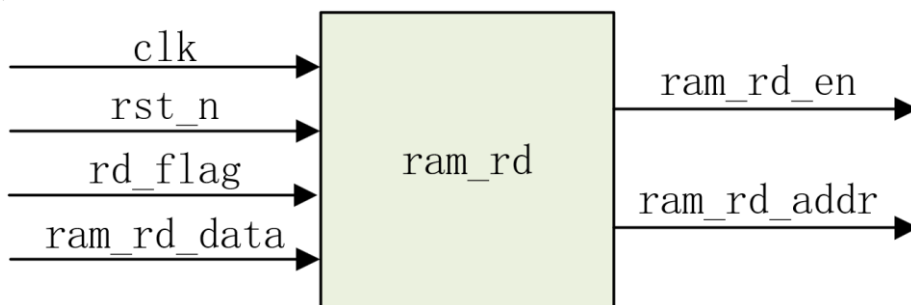


图 25.3.12 RAM 读模块框图

模块端口与功能描述如下表所示:

表 25.3.3 ram_rd模块端口与功能描述

信号名	位宽	方向	端口说明
clk	1	输入	系统时钟, 50MHz
rst_n	1	输入	系统复位按键, 低电平有效
rd_flag	1	输入	读启动标志, 高有效
ram_rd_data	8	输入	ram读数据
ram_rd_en	1	输出	ram读端口使能信号, 高有效
ram_rd_addr	8	输出	ram读地址

绘制波形图

在编写代码前, 我们先大致梳理一下模块的端口时序, 并绘制出如下波形图:

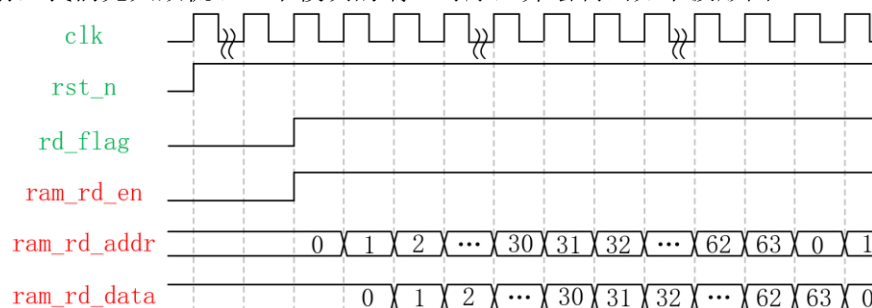


图 25.3.13 RAM 读模块波形图

因为我们需要一直从 RAM 中读出数据, 所以当复位结束且读启动信号拉高后, 我们就将 ram_rd_en (读端口使能) 一直置为高。当读端口使能后, 读地址就会一直在 0~63 之间循环计数, 并读出对应 RAM 地址中的数据, 需要注意的是读数据的输出会比读地址晚一个时钟周期。

代码编写

ram_rd 模块用于产生 RAM 读操作所需的信号, 并接引从 RAM 中读出的数据, 其代码如下所示:

```

1 module ram_rd(
2     input        clk        ,    //时钟信号
3     input        rst_n      ,    //复位信号, 低电平有效
4
5     //RAM 读端口操作
6     input        rd_flag    ,    //读启动标志
7     input [7:0]  ram_rd_data,    //ram 读数据
8     output       ram_rd_en  ,    //端口使能
9     output reg [5:0] ram_rd_addr //ram 读地址
10 );
11 //*****
12 /**                                main code
13 //*****
14
15 //控制 RAM 使能信号

```

```
16 assign ram_rd_en = rd_flag;
17
18 //读地址信号 范围:0~63
19 always @(posedge clk or negedge rst_n) begin
20     if(!rst_n)
21         ram_rd_addr <= 6'd0;
22     else if(ram_rd_addr < 6'd63 && ram_rd_en)
23         ram_rd_addr <= ram_rd_addr + 1'b1;
24     else
25         ram_rd_addr <= 6'd0;
26 end
27
28 endmodule
```

如代码第 16 行所示, 当读启动信号拉高后, 读端口将被使能; 当读端口使能后, 读地址会在 0~63 之间循环计算, 从而读出对应 RAM 地址中的数据 (如代码第 18~26 行)。

25.3.5 仿真验证

编写 TB 文件

我们接下来先对代码进行仿真, 因为本章实验我们只有系统时钟和系统复位这两个输入信号, 所以仿真文件也只需要编写这两个信号的激励即可, TestBench 代码如下:

```
1 `timescale 1ns / 1ps //仿真单位/仿真精度
2
3 module tb_ip_2port_ram();
4
5 //parameter define
6 parameter CLK_PERIOD = 20; //时钟周期 20ns
7
8 //reg define
9 reg sys_clk;
10 reg sys_rst_n;
11
12 //信号初始化
13 initial begin
14     sys_clk = 1'b0;
15     sys_rst_n = 1'b0;
16     #200
17     sys_rst_n = 1'b1;
18 end
19
20 //产生时钟
21 always #(CLK_PERIOD/2) sys_clk = ~sys_clk;
22
```

```

23 ip_2port_ram u_ip_2port_ram(
24     .sys_clk          (sys_clk          ),
25     .sys_rst_n        (sys_rst_n        ),
26 );
27
28 endmodule

```

仿真验证

通过仿真我们得到了以下波形图:

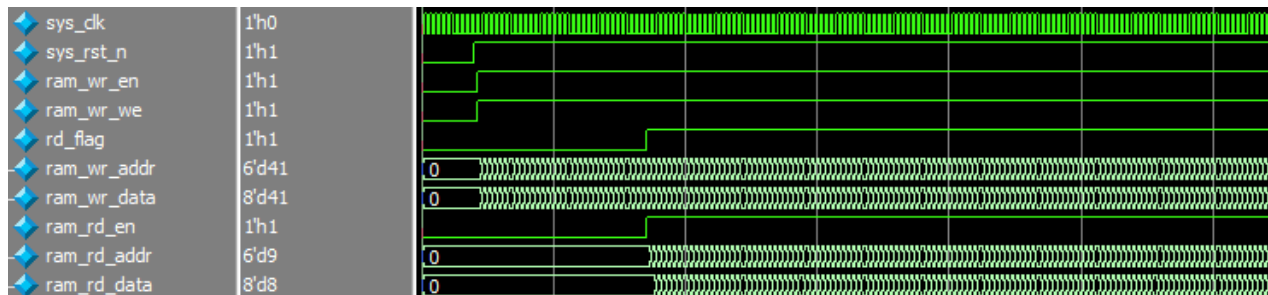


图 25.3.14 双端口 RAM 仿真波形图 1

可以看出在写入一定的数据后,读启动信号 (rd_flag) 被拉高,这时候读端口才被使能 (即 ram_rd_en 拉高),此后读模块开始读出 ram 中的数据。

接着我们展开波形图看一下细节,如下图所示:

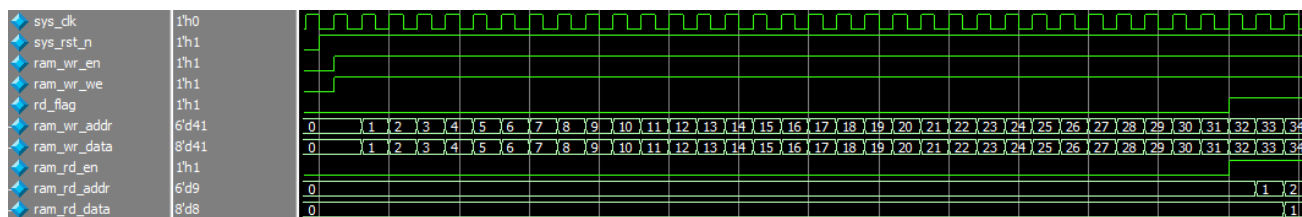


图 25.3.15 双端口 RAM 仿真波形图 2

由上图可知, ram_wr_we 信号拉高,说明此时是对 ram 进行写操作。ram_wr_we 信号拉高之后,地址和数据都是从 0 开始累加,也就是说当 ram 地址为 0 时,写入的数据也是 0;当 ram 地址为 1 时,写入的数据也是 1。当地址为 32 时,读启动信号 (rd_flag) 被拉高,此时将读端口使能,开始读出 ram 中的数据,这里大家可能会有一个疑惑,明明我们代码中是在写地址计数到 31 的时候给 rd_flag 赋值为 1 (即拉高) 的,但是仿真结果为什么是在 32 的时候才拉高呢?这是因为在时序逻辑中,rd_flag 的赋值是在写地址计数到 31 时的下一个时钟信号的上升沿完成的,所以我们在仿真中看到的结果就是在写地址为 32 时,rd_flag 才被拉高。

我们接着看后续的波形,如下图所示:

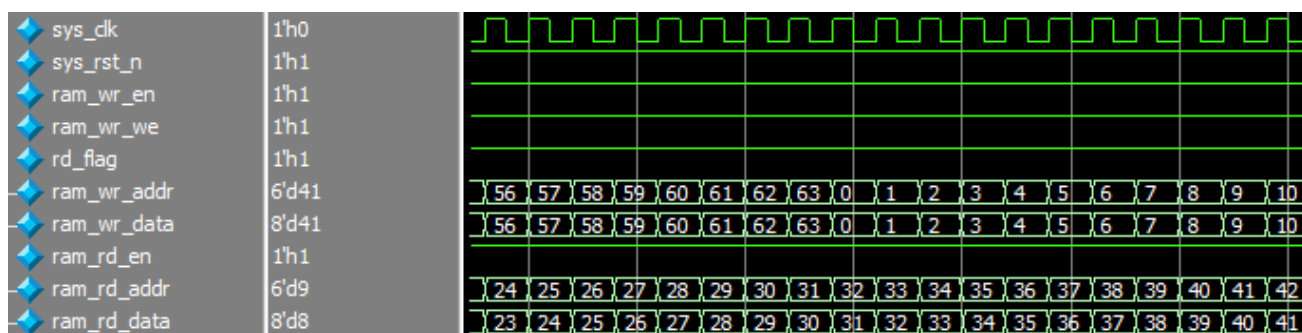


图 25.3.16 双端口 RAM 仿真波形图 3

由上图可知, 经过 `rd_flag` 将读写地址错开后, 写端口和读端口在同时对 `ram` 进行相应的操作, 并没有发生读写冲突的情况, 且读操作和写操作恒定相差 32 个地址。

由仿真分析可知我们不仅成功实现了双端口 `ram` 的读写功能, 还有效避免了读写冲突的问题。

25.4 下载验证

引脚约束

在仿真验证完成后, 接下来对引脚进行分配, 并上板验证。本实验中, 系统时钟、按键复位的管脚分配如下表所示:

表 25.4.1 IP核之双端口RAM实验管脚分配

信号名	方向	管脚	端口说明	电平标准
<code>sys_clk</code>	input	U18	系统时钟, 50MHz	LVC MOS33
<code>sys_rst_n</code>	input	N16	系统复位按键, 低电平有效	LVC MOS33

对应的 XDC 约束语句如下所示:

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVC MOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN N16 IOSTANDARD LVC MOS33} [get_ports sys_rst_n]
```

Vivado 软件中 IO Planning 界面如下图所示:

Scalar ports (2)										
sys_clk	IN	U18	▼	✓	34	LVC MOS33*	▼	3.300		NONE ▼ NONE ▼
sys_rst_n	IN	N16	▼	✓	35	LVC MOS33*	▼	3.300		NONE ▼ NONE ▼

图 25.4.1 IO Planning 引脚约束界面

添加 ila IP 核进行在线调试

接下来添加 ILA IP 核, 将 `ram_wr_en` (1 位)、`ram_wr_we` (1 位)、`ram_rd_en` (1 位)、`rd_flag` (1 位)、`ram_wr_addr` (6 位)、`ram_wr_data` (8 位)、`ram_rd_addr` (6 位) 和 `ram_rd_data` (8 位) 信号添加至观察列表中, 添加 ILA IP 核的方法这里不再赘述, 本例程是将 ILA 例化在了顶层模块 (`ip_2port_ram`) 中, 例化代码如下所示:

```
ila_0 u_ila_0 (
    .clk      (sys_clk      ), // input wire clk

    .probe0   (ram_wr_en   ), // input wire [0:0] probe0
    .probe1   (ram_wr_we   ), // input wire [0:0] probe1
    .probe2   (ram_rd_en   ), // input wire [0:0] probe2
    .probe3   (rd_flag     ), // input wire [0:0] probe3
    .probe4   (ram_wr_addr ), // input wire [5:0] probe4
    .probe5   (ram_wr_data ), // input wire [7:0] probe5
    .probe6   (ram_rd_addr ), // input wire [5:0] probe6
    .probe7   (ram_rd_data ), // input wire [7:0] probe7
);
```

上板验证

编译工程并生成比特流文件后, 将下载器一端连接电脑, 另一端与开发板上的 JTAG 下载口连接, 连接

考, 如下图所示:

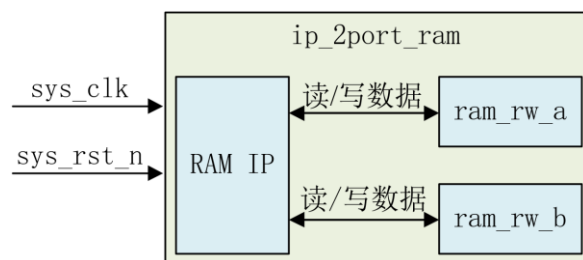


图 25.6.1 模块框图与端口

框图中 `ram_rw_a` 为端口 A 的读写模块, `ram_rw_b` 为端口 B 的读写模块, 其余模块和信号大家应该已经很熟悉了, 这里就不再多说了。

有了框图后, 再为大家提供一个思路: 在简介中我们提到过真双端口 RAM 只是将单端口 RAM 的所有信号做了一个加倍处理, 所以这里两个读写模块也可以理解为两个单端口 RAM, 但是为了避免读写冲突和写写冲突, 这两个模块并不完全一样, 所以不能通过例化两次单端口实验中的读写模块来完成, 而是要对其稍作修改, 编写为两个机制不同的读写模块。

举个例子: 假如我们两个端口的读写数据位宽 (8 位) 和深度 (32) 都相同, 那么我们可以做一个能够计数到 32 (0~31) 的计数器, 当计数器范围在 0~15 时, 端口 A 向 0~15 地址中写入数据, 端口 B 向 16~31 地址中写入数据; 当计数器范围在 16~31 时, 端口 A 从 16~31 地址中读出数据, 端口 B 从 0~15 地址中读出数据, 这么做便避免了两个端口之间的读写冲突和写写冲突。

当然并不是只有上述的一种思路, 如果大家有其他更好的思路也不妨自己尝试看看。