第 3 章　实验 3：自定制接口模块的设计

3.1　实验目的

在本实验中，在深入理解 MIPSfpga 处理器和 AXI 总线协议以及掌握了基于 Vivado IP 集成方法搭建 MIPSfpga 处理器系统的流程的基础上，实现一个基于 AXI4 总线接口标准的外设模块，并将该模块添加到 MIPSfpga 处理器系统中。需要按照下述步骤完成本实验：

（1）编写并封装一个带中断输出信号的基于 AXI4 总线接口标准的 PWM 外设模块（该模块的中断功能将在后续的实验中用到，在本实验中不会使用）。

（2）将该自定制接口模块添加到实验 2 搭建的 MIPSfpga 处理器硬件平台上。

（3）生成新建的包括该自定制接口模块的 MIPSfpga 处理器硬件平台比特流文件，并将其烧写到 Nexsy 4 DDR FPGA 开发板上。

（4）编写 C 语言程序，对 MIPSfpga 处理器硬件平台进行测试。

通过本实验，应加深对 AXI4 总线接口标准的理解，掌握基于 Vivado IP 集成开发流程设计实现处理器自定制外设模块的方法，学会编写、编译并调试 MIPSfpga 处理器外设驱动程序和应用测试程序，为后续设计实现更加复杂的 MIPSfpga 处理器硬件平台奠定基础。

3.2　实验内容

3.2.1　基于 AXI4 总线接口的自定制外设模块封装

开始实验前，先复制实验 2 的工程，即复制实验 2 的工程文件夹 MIPSfpga_axi4 并将复制后的文件夹更名为 MIPSfpga_CustomIP，然后按照下述步骤开始实验：

（1）启动 Vivado，打开 MIPSfpga_CustomIP 工程（因为只修改了该工程的文件夹名称，因此该工程的名称仍然是 MIPSfpga_axi4），然后选择 Tools→Create and Package IP 菜单命令，如图 3-1 所示。

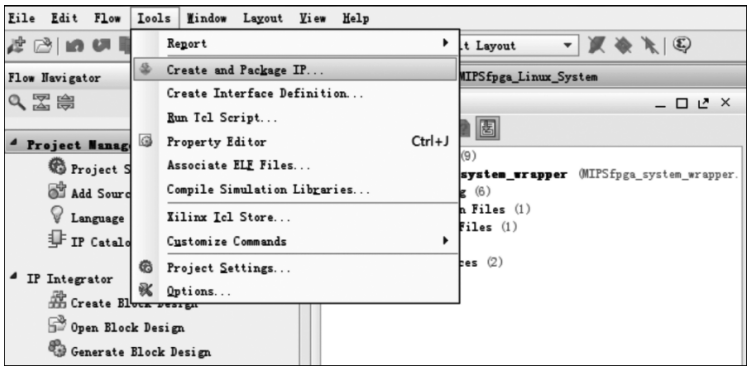


图 3-1　选择 Tools→Create and Package IP 菜单命令

(2) 出现 Create and Package New IP 对话框后,单击 Next 按钮,在下一步的界面中选择 Create a new AXI4 peripheral 单选按钮,如图 3-2 所示。

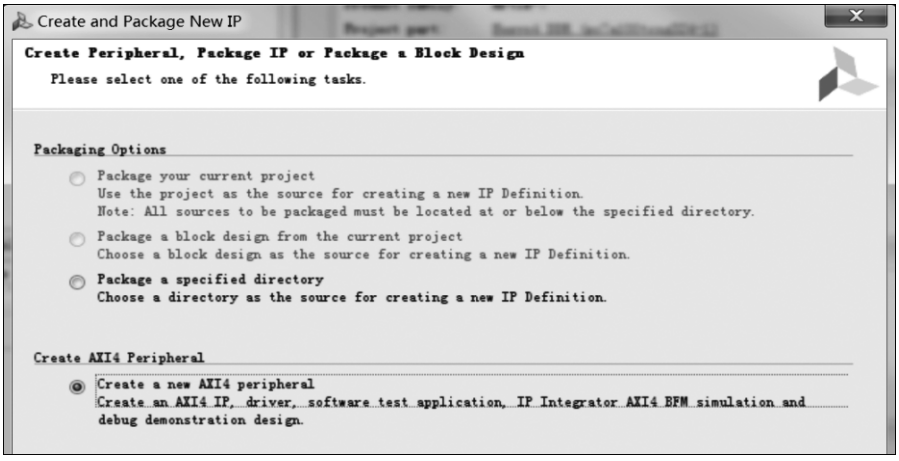


图 3-2　选择 Create a new AXI4 peripheral 单选按钮

(3) 如图 3-3 所示,在下一步的界面中输入模块的名称、版本号等信息,然后单击 Next 按钮。

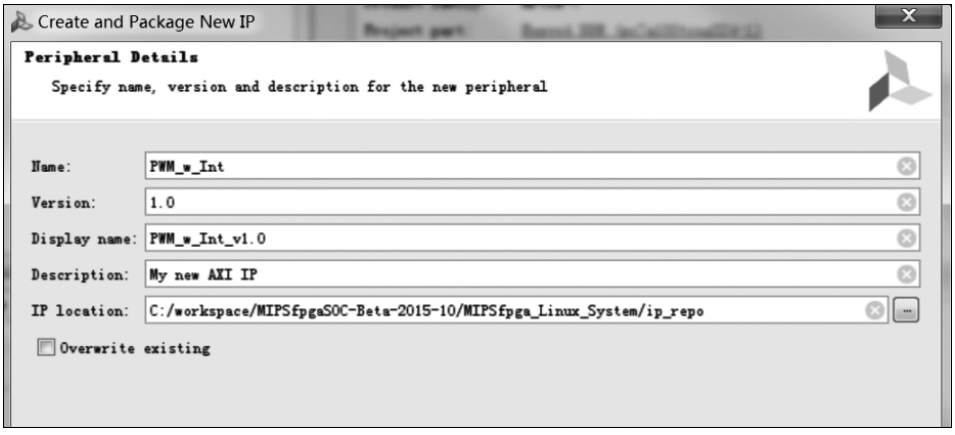


图 3-3　输入模块的相关信息

(4) 在下一步的界面中,设定该模块 AXI4 接口的类型和参数。这里选择接口类型为 AXI4-Lite 的从端口,该接口包含 4 个寄存器,寄存器的位数为 32 位,如图 3-4 所示。

(5) AXI4 接口类型和参数设置完成后,单击 Next 按钮,接下来选择 Edit IP 单选按钮,如图 3-5 所示,然后单击 Finish 按钮。

(6) 这时会启动一个新的 Vivado 界面,其中出现了一个名为 edit_PWM_w_Int_v1_0 的工程(该工程位于图 3-3 中 IP location 指定的文件夹下,且工程名与该模块的名称和版本号对应),如图 3-6 所示。

至此,就有了一个基于 AXI4 总线接口的自定制外设模块的模板,接下来就需要对该

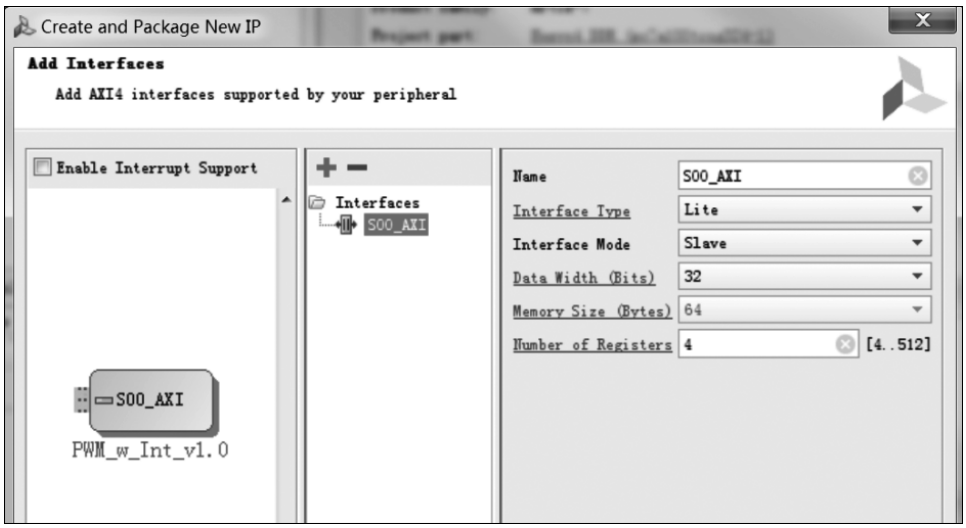


图 3-4　AXI4 接口类型和参数设置

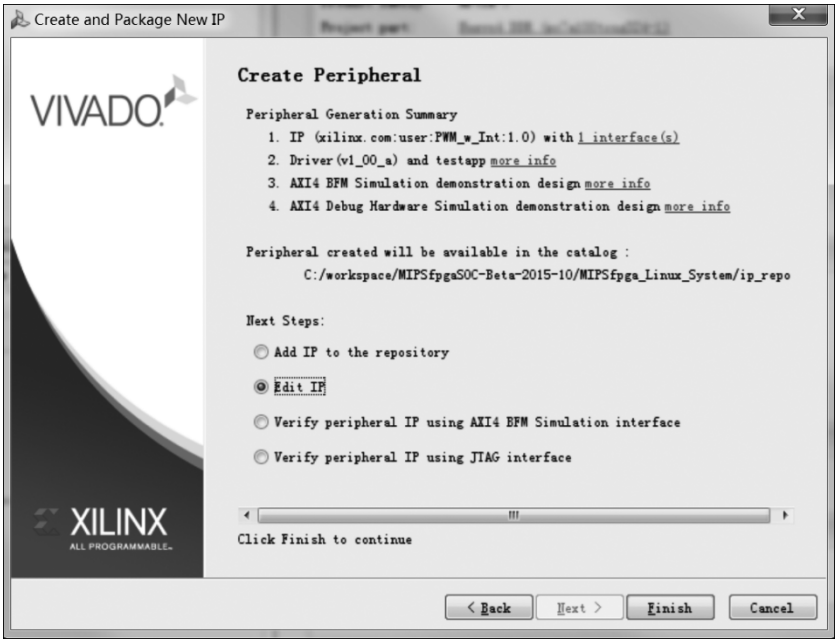


图 3-5　自定制模块设置完成

edit_PWM_w_Int_v1_0 工程进行相应的修改和完善，从而设计出需要的自定制外设模块。

以本实验的 PWM 外设模块的设计为例，需要进行以下修改（详细的程序代码请参看 3.3 节），具体步骤如下：

（1）在 edit_PWM_w_Int_v1_0 工程中打开名为 PWM_w_Int_v1_0.v 的文件，找到注释行//Users to add parameters here，在该行下面添加整型参数 PWM_PERIOD = 20，然后在注释行//Users to add ports here 下面添加线网型输出端口 Interrupt_out、LEDs、PWM_

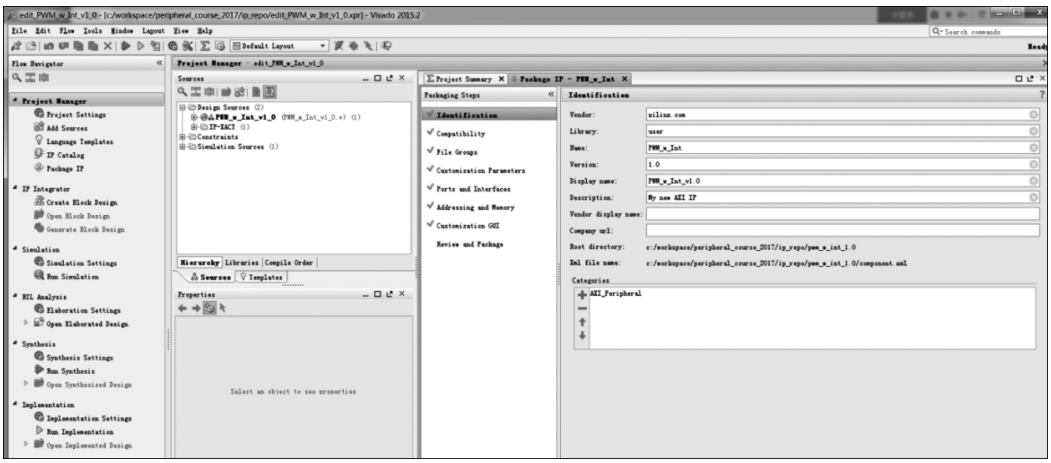


图 3-6　新建的 edit_PWM_w_Int_v1_0 工程

Counter 和 DutyCycle，如图 3-7(a)所示；在 PWM_w_Int_v1_0.v 文件中找到名为 PWM_w_Int_v1_0_S00_AXI_inst 的模块实例化代码段，在其中添加端口引用，即.slv_reg0(DutyCycle)，如图 3-7(b)所示；在注释行(//Add user logic here)后添加名为 PWM_Controller_Int 的模块实例化代码段，如图 3-7(c)所示。

(2) PWM_w_Int_v1_0.v 文件编辑完成后，再打开其下层的名为 PWM_w_Int_v1_0_S00_AXI.v 的文件，在注释行//Users to add parameters here 后将 slv_reg0 修改为输出端口，如图 3-8 所示。

(3) 在 edit_PWM_w_Int_v1_0 工程中通过右键快捷菜单中的 Add Source 命令添加一个名为 PWM_Controller_Int.v 的设计文件，如图 3-9 和图 3-10 所示。

(4) 添加完设计文件后，在工程中打开 PWM_Controller_Int.v 文件，根据需要输入程序

```
module PWM_w_Int_v1_0 #
(
    // Users to add parameters here
    parameter integer PWM_PERIOD = 20,
    // User parameters ends
    // Do not modify the parameters beyond this line


    // Parameters of Axi Slave Bus Interface S00_AXI
    parameter integer C_S00_AXI_DATA_WIDTH  = 32,
    parameter integer C_S00_AXI_ADDR_WIDTH  = 4
)
(
    // Users to add ports here
    output wire Interrupt_out,
    output wire [1:0] LEDs,
    output wire [PWM_PERIOD-1:0] PWM_Counter,
    output wire [31:0] DutyCycle,

    // User ports ends
    // Do not modify the ports beyond this line
```

(a) 修改 PWM_w_Int_v1_0.v 之一

图 3-7　PWM_w_Int_v1_0.v 文件的修改

```
// Instantiation of Axi Bus Interface S00_AXI
    PWM_w_Int_v1_0_S00_AXI # (
        .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
        .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
    ) PWM_w_Int_v1_0_S00_AXI_inst (
        .S_AXI_ACLK(s00_axi_aclk),
        .S_AXI_ARESETN(s00_axi_aresetn),
        .S_AXI_AWADDR(s00_axi_awaddr),
        .S_AXI_AWPROT(s00_axi_awprot),
        .S_AXI_AWVALID(s00_axi_awvalid),
        .S_AXI_AWREADY(s00_axi_awready),
        .S_AXI_WDATA(s00_axi_wdata),
        .S_AXI_WSTRB(s00_axi_wstrb),
        .S_AXI_WVALID(s00_axi_wvalid),
        .S_AXI_WREADY(s00_axi_wready),
        .S_AXI_BRESP(s00_axi_bresp),
        .S_AXI_BVALID(s00_axi_bvalid),
        .S_AXI_BREADY(s00_axi_bready),
        .S_AXI_ARADDR(s00_axi_araddr),
        .S_AXI_ARPROT(s00_axi_arprot),
        .S_AXI_ARVALID(s00_axi_arvalid),
        .S_AXI_ARREADY(s00_axi_arready),
        .S_AXI_RDATA(s00_axi_rdata),
        .S_AXI_RRESP(s00_axi_rresp),
        .S_AXI_RVALID(s00_axi_rvalid),
        .S_AXI_RREADY(s00_axi_rready),
        .slv_reg0(DutyCycle)
    );
```

(b) 修改 PWM_w_Int_v1_0.v 之二

```
// Add user logic here
PWM_Controller_Int #(
    .period(PWM_PERIOD)
) PWM_inst (
    .Clk(s00_axi_aclk),
    .DutyCycle(DutyCycle),
    .Reset(s00_axi_aresetn),
    .PWM_out(LEDs),
    .Interrupt(Interrupt_out),
    .count(PWM_Counter)
);
// User logic ends

endmodule
```

(c) 修改 PWM_w_Int_v1_0.v 之三

图 3-7　（续）

```
// Users to add ports here
output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg0,
// User ports ends
// Do not modify the ports beyond this line
```

图 3-8　PWM_w_Int_v1_0_S00_AXI.v 文件的修改

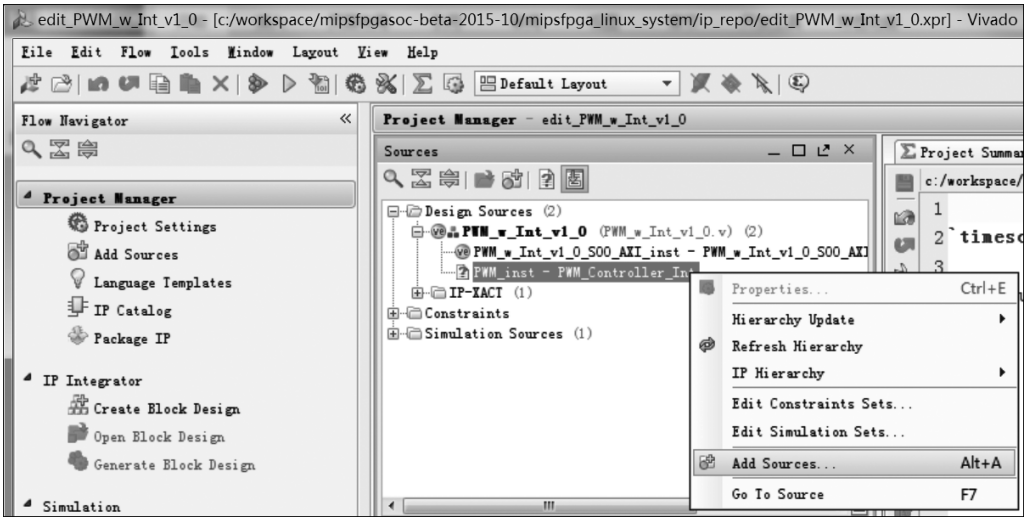


图 3-9　在 edit_PWM_w_Int_v1_0 工程中添加设计文件

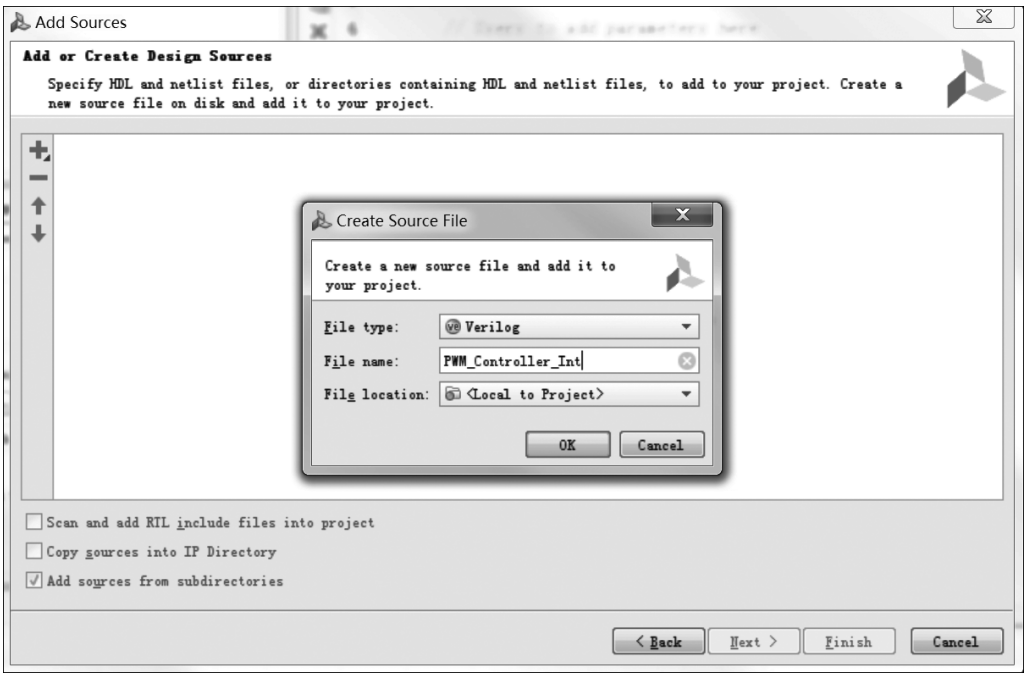


图 3-10　创建名为 PWM_Controller_Int.v 的设计文件

源码(具体参看 3.3 节)。

(5) 然后编写相应的仿真测试程序,对外设模块设计的正确性进行功能仿真验证。仿真验证正确后可以综合。综合无误后就可以对该模块进行 IP 封装了。

(6) IP 封装通过 Vivado 的 Package IP 功能实现,其界面如图 3-11 所示(如果图 3-11 所示的界面没有在工程中显示,则可以通过选择 Package IP 菜单命令打开该界面)。

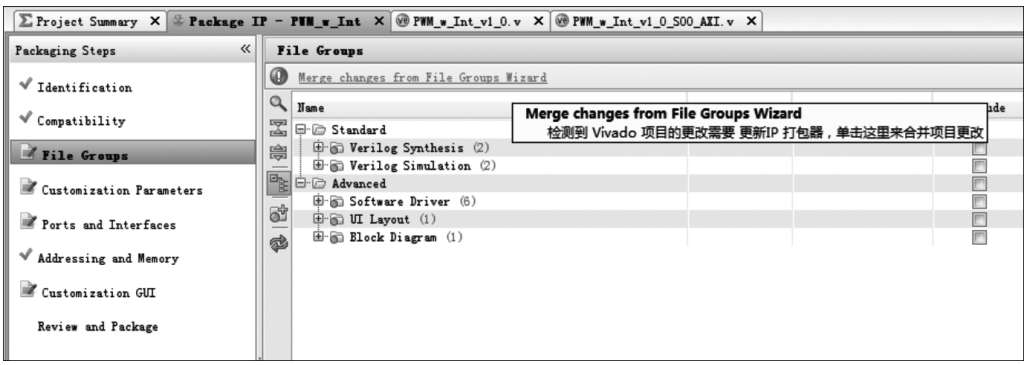


图 3-11　Package IP 界面

(7) 在 Package IP 界面左侧找到 File Groups 选项。如果 File Groups 选项的图标上不是绿色的钩,则在右侧选择 Merge changes from File Groups Wizard,然后检查 PWM_Controller_Int.v 文件是否已经加入 File Groups 中。如果选择 Merge changes from File Groups Wizard 后 PWM_Controller_Int.v 文件没有自动加入,则需手工添加该文件。

（8）PWM_Controller_Int.v 文件添加成功后的 File Groups 窗口如图 3-12 所示。如果添加的 PWM_Controller_Int.v 文件不是如图 3-12 所示的相对路径，而是绝对路径，则需要手工删除该文件，然后重新添加该文件。

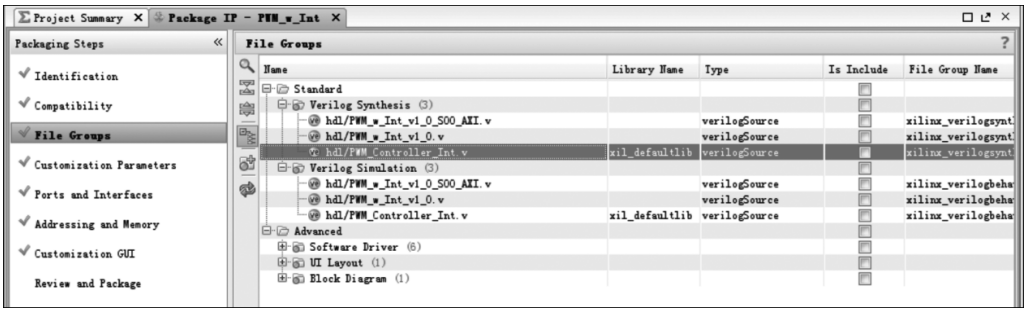


图 3-12　PWM_Controller_Int.v 文件已正确添加到 File Groups 中

（9）在 Package IP 界面左侧选择 Ports and Interfaces 选项，检查模块的 Interrupt_out 和 LEDs 信号引脚是否已经设置为输出引脚。如果 Ports and Interfaces 选项的图标上不是绿色的钩，则在该选项上单击，即可自动加入；也可以右击该选项，在弹出的快捷菜单中选择 Import IP Ports 命令，找到模块的顶层设计文件，即 PWM_w_Int_v1_0.v，通过选定顶层设计文件添加输出引脚，如图 3-13 所示。引脚正确添加后的结果如图 3-14 所示。

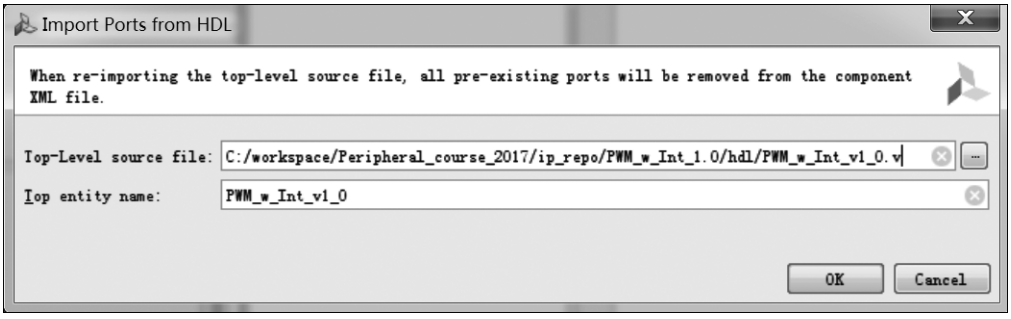


图 3-13　通过选定顶层设计文件添加输出引脚

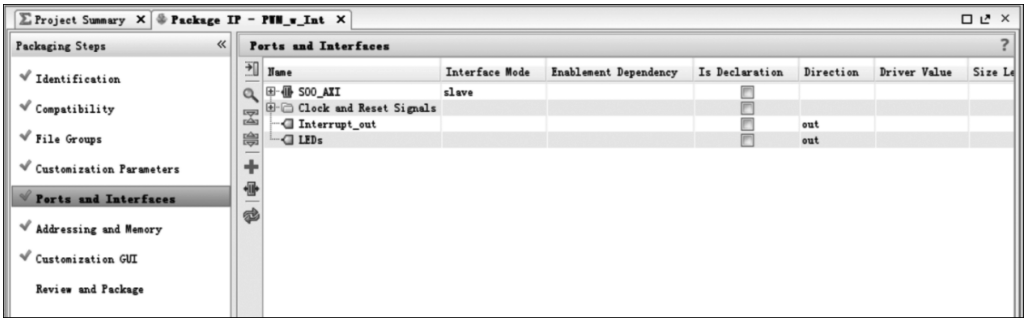


图 3-14　引脚正确添加后的结果

（10）最后选择 Review and Package 选项，确认设置正确无误后单击 Re-Package IP 按钮，等待 Vivado 完成 edit_PWM_w_Int_v1_0 工程模块的封装（IP 封装完成后，通常当前的 Vivado 工程会自动关闭并退出），如图 3-15 所示。

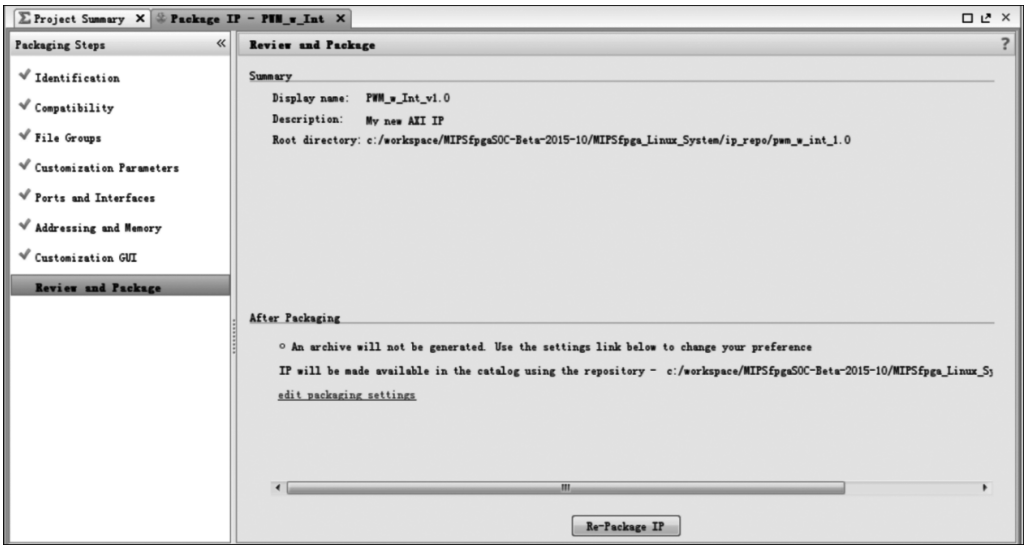


图 3-15　完成 IP 封装

IP 封装完成后，需要回到 MIPSfpga_CustomIP 工程，选择 IP Catalog 菜单命令打开 IP 库，查看刚才封装的 PWM 外设模块是否在 IP 库中，如图 3-16 所示，此时 IP Catalog 窗口中的 User Repository 下的 AXI Peripheral 文件夹下名为 PWM_w_Int_v1.0 的模块就是刚刚自定制的基于 AXI4 总线接口标准的外设模块。

如果需要，可以再次打开 edit_PWM_w_Int_v1_0 工程，对模块进行修改和重新封装。

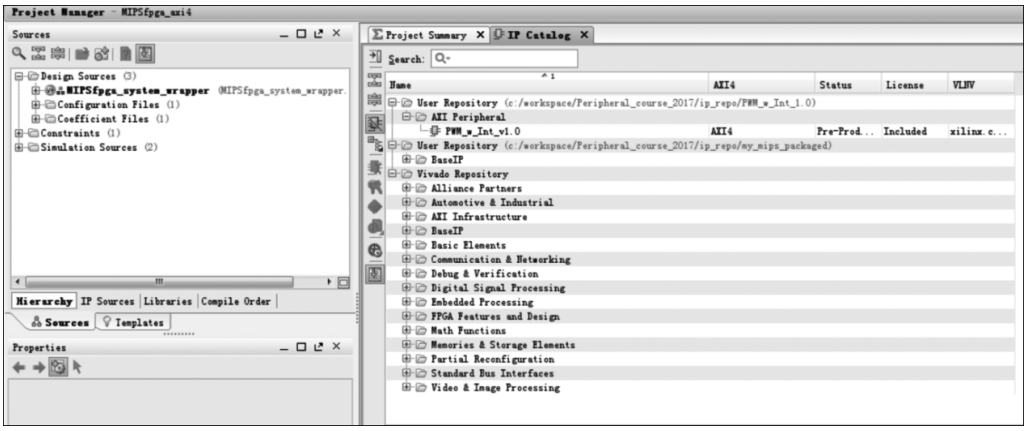


图 3-16　IP Catalog 中新增的用户自定制模块

3.2.2　在 MIPSfpga 硬件平台中使用自定制模块

具体实验步骤如下：

(1) 在 Vivado 的 Project Manager 下选择 Open Block Design，打开 MIPSfpga_CustomIP 工程，如图 3-17 所示。

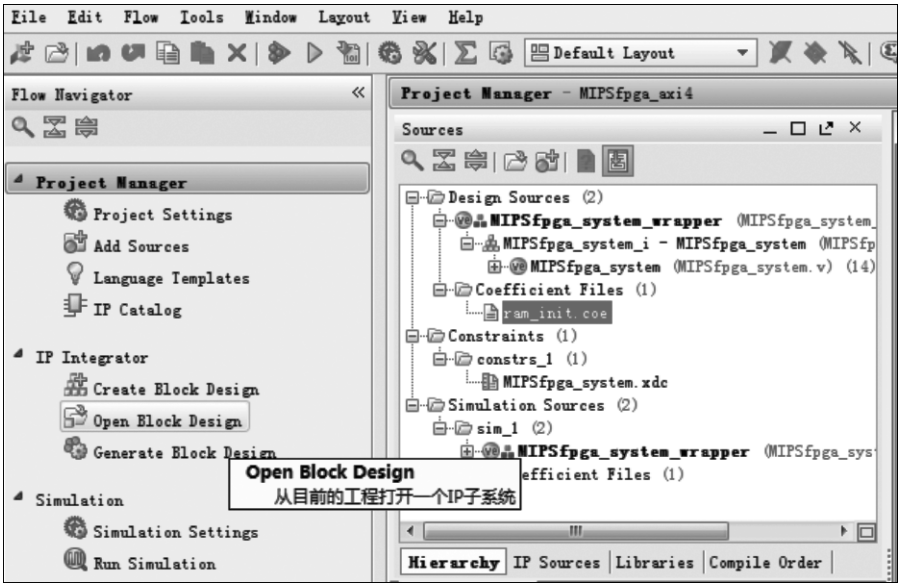


图 3-17　选择 Open Block Design 打开工程

(2) 右击工作空间，在弹出的快捷菜单中选择 Add IP 命令，在 IP 库中找到用户自定制模块 PWM_w_Int_v1.0，双击该模块，将其添加到工程中，并按照图 3-18 所示将其连接到 MIPSfpga 处理器系统中(具体连接方法参看实验 2)。

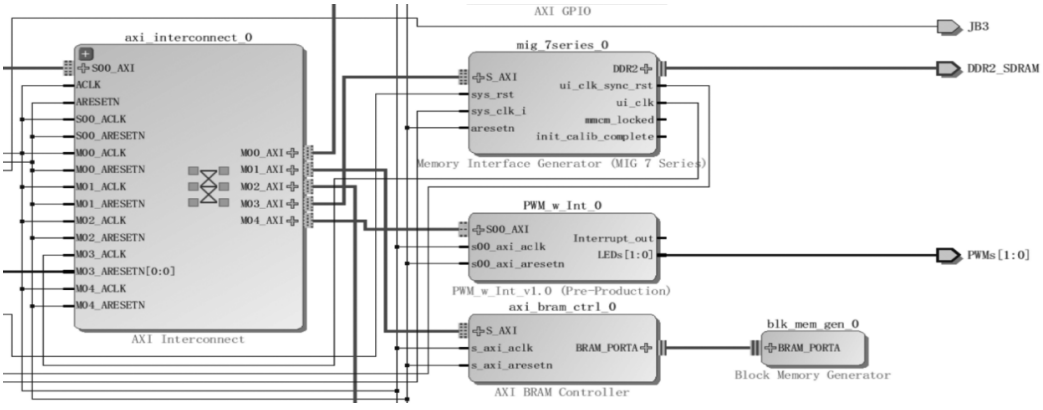


图 3-18　添加 PWM_w_Int_v1.0 模块

(3) PWM_w_Int_v1.0 模块连接完成后，切换到 Address Editor，将模块的偏移地址(Offset Address，即该 IP 模块的物理地址)设置为 0x10C0_0000，如图 3-19 所示。

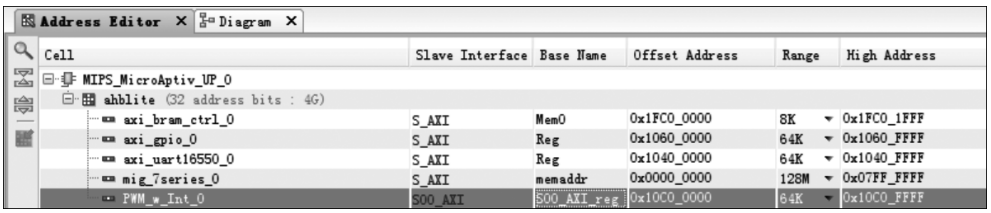


图 3-19　设置 PWM_w_Int_v1.0 模块的偏移地址

完成上面的操作后，MIPSfpga 处理器硬件平台如图 3-20 所示。

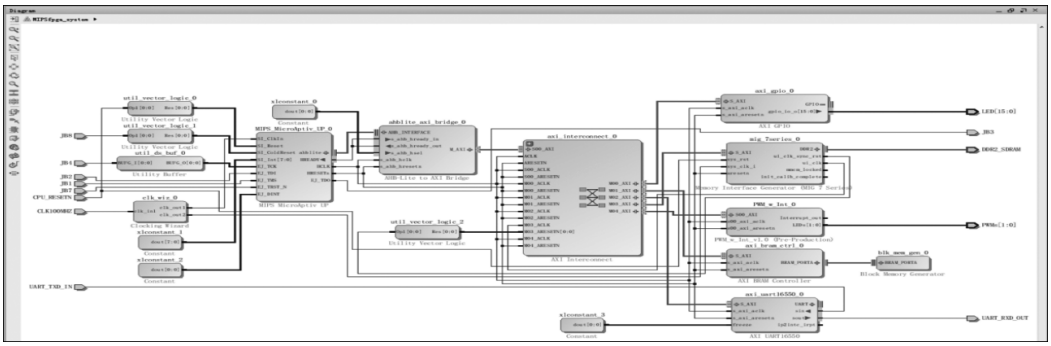


图 3-20　添加自定制模块后的 MIPSfpga 处理器硬件平台

（4）在 Project Manager 中选择 Validate Design，对设计的正确性进行检验。检验过程中如果只是出现警告，单击 OK 按钮忽略即可，然后在 Project Manager 中选择 Generate Block Design，在弹出的对话框中单击 Generate 按钮，更新 MIPSfpga_system_wrapper 文件。

（5）相应地修改约束文件，为新添加的 PWM 外设模块增加引脚绑定。

（6）单击 Generate Bitstream 按钮，生成比特流文件。比特流文件生成后，注意观察时序能否满足设计要求。

3.2.3　MIPSfpga 硬件平台测试

具体实验步骤如下：

（1）在下载的 system_ability 文件夹下找到 MIPSfpga_CustomIP_C 文件夹，仔细阅读该文件夹下的所有程序和文件。

（2）在主机上打开一个 cmd 命令窗口，切换到 MIPSfpga_CustomIP_C 文件夹，然后输入 make 命令，对程序进行编译（如果需要，可以用 make clean 命令先对编译的程序进行清除，详见 1.2.3 节）生成 ELF 文件。

（3）连接 Nexys 4 DDR FPGA 开发板的下载线和 JTAG 调试器。打开 Vivado，向 Nexys 4 DDR FPGA 开发板烧写添加自定制外设模块后的 MIPSfpga 硬件平台的比特流文件。

（4）再打开一个 cmd 命令窗口，切换到 Codescape_Scripts 文件夹，然后在该 cmd 命令

窗口中输入如下命令：

```
loadMIPSfpga.bat C:\workspace\Peripheral_course_2017\MIPSfpga_CustomIP_C
```

在上面的命令中，loadMIPSfpga.bat 后面给出的是 MIPSfpga_CustomIP_C 文件夹所在的路径。

(5) 在主机上打开一个串口终端(例如 PuTTY.exe)，将波特率设置为 115 200baud，然后观察程序运行情况，特别要注意观察 PWM 模块的输出。

(6) 如果程序运行不正确，使用 GDB 工具进行调试，找出原因后进行修正。

(7) 对 MIPSfpga_CustomIP_C 文件夹下的程序进行修改和完善，尝试进行更全面的测试或添加新的功能，思考如何为 PWM 外设模块提供相应的驱动程序。

3.3　实验背景及源码

3.3.1　AXI 总线协议

1. AMBA 标准

AMBA(Advanced Microcontroller Bus Architecture，高级微控制器总线体系结构)是由 ARM 公司推出的片上总线。它提供了一种特殊的机制，可将 RISC 处理器方便地与其他 IP 核或外设集成。第一代 AMBA 标准(即 AMBA 1.0)定义了两组总线协议，即高级系统总线(Advanced System Bus，ASB)和高级外设总线(Advanced Peripheral Bus，APB)。AMBA 2.0 则在此基础上增加了高级高性能总线(Advanced High-performance Bus，AHB)。

AMBA 3.0 标准定义了 4 个总线协议，这些协议针对高数据吞吐量、低带宽通信，要求低门数、低功耗以及执行片上测试和调试访问的数据集中处理。这 4 个总线协议除 AHB、ASB、APB 外，还包括新增的 AXI(Advanced eXtensible Interface，高级扩展接口)总线协议，它丰富了 AMBA 标准的内容，能够满足超高性能和复杂的片上系统设计的需求。

AMBA 4.0 标准在 AMBA 3.0 的基础上对 AXI 总线协议进行了扩充，分别定义了 3 个总线接口协议，即 AXI4、AXI4-Lite 和 AXI4-Stream。

AXI4 协议是对 AXI3 协议的更新，在用于多个主端口系统时，可提高互连的性能和利用率。其主要特点是增强了以下功能：

(1) 对于突发长度，支持最多 256 位。

(2) 能够发送服务质量信号。

(3) 支持多区域接口。

AXI4-Lite 协议是 AXI4 协议的子协议，能够较好地用于连接更简单、寄存器更少的接口设备。其主要功能和特点如下：

(1) 所有事务的突发长度均为 1 位。

(2) 所有数据存取的大小均与数据总线的宽度相同。

(3) 不支持独占访问。

AXI4-Stream 协议可用于从主接口到辅助接口的单向数据传输，能够显著降低信号传输的路由开销。该协议的主要功能和特点如下：

(1) 使用同一组共享线支持单数据流和多数据流。

(2) 在同一互连结构内支持多种数据宽度。

(3) 方便在 FPGA 中实现和使用。

2. AXI4 总线协议简介

AXI4 总线协议定义了 5 个独立进行数据传输的通道,分别是读地址(Read Address,RA)通道、读数据(Read Data,RD)通道、写地址(Write Address,WA)通道、写数据(Write Data,WD)通道和写响应(Write Response,WR)通道。

读/写地址通道提供数据传输需要的地址信息。数据传输通过写数据通道和读数据通道在主设备和从设备之间进行:写数据时,写数据通道用于将主设备的数据传输到从设备,此时写响应通道用于数据传输结束后从设备向主设备提供写数据完成确认信号(即写响应信号),如图 3-21 所示;读数据时,读数据通道用于将从设备的数据传输到主设备,如图 3-22 所示。

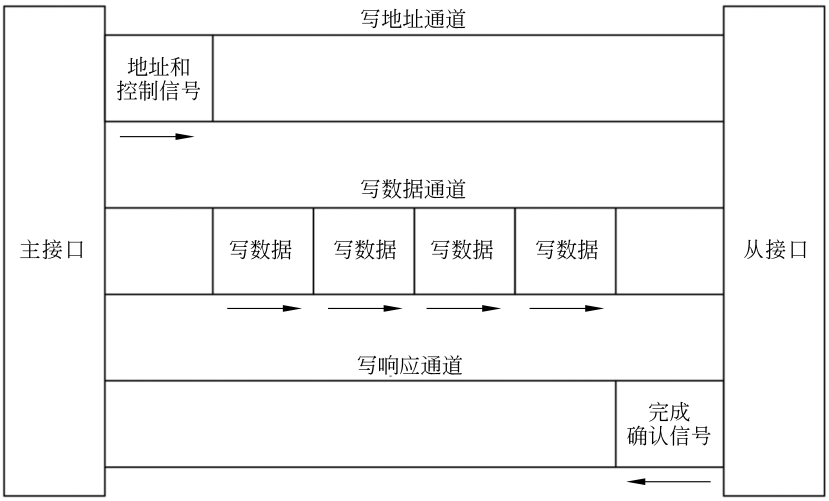


图 3-21　AXI4 总线协议写数据传输方式

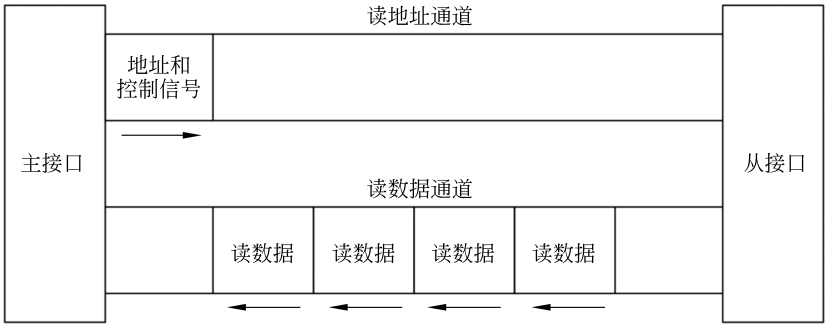


图 3-22　AXI4 总线协议读数据传输方式

不管是哪种传输方式,AXI4 总线协议都有如下规定:

(1) 数据传输之前允许先提供地址信息。

(2) 支持多个未完成传输过程同时进行。

(3) 支持数据传输乱序完成。

读数据通道用于传输读的数据以及从设备对主设备读操作的响应信息。其数据线的宽度可以是 8、16、32、64、128、256、512 或 1024 位之一。响应信息主要用于表明本次读操作的结束。

写数据通道用于传输主设备要写到从设备的数据。其数据线的宽度可以是 8、16、32、64、128、256、512 或 1024 位之一，同时用字节使能信号标明有效的写数据字节。写数据通道中传输的信息通常被当作缓存处理，因此主设备可以在本次数据传输未被从设备确认的情况下就开始下一次数据传输。

从设备利用写响应通道向主设备反馈写操作的信息，所有写操作的完成确认信号都通过写响应通道传输。完成确认信号仅用于表明本次写操作的完成，并不用于表明本次写操作过程中每个数据的传输。

3. AXI4 互连方式

使用 AXI4 总线进行设备互连时，其结构如图 3-23 所示，通常由一个互连结构(interconnect)将多个主、从设备连接起来。

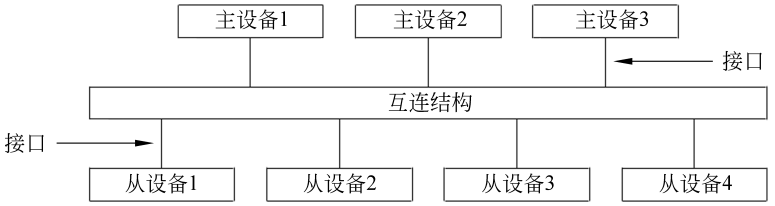


图 3-23　AXI4 总线互连结构

主设备与互连结构以及互连结构与从设备之间都通过一对 AXI4 总线接口连接，如图 3-24 所示，这些接口也都遵循 AXI4 总线协议。

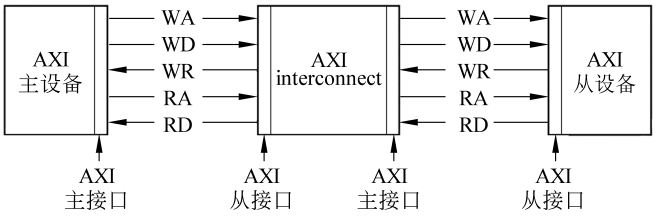


图 3-24　AXI4 总线系统互连方式

4. AXI4 总线信号

下面对 AXI4 总线的信号进行描述，包括全局信号、写地址通道信号、写数据通道信号、写响应通道信号、读地址通道信号、读数据通道信号以及低功耗接口信号，如表 3-1～表 3-7 所示。在下面的表中，所有通道信号都是以 32 位数据总线、4 位写数据使能以及 4 位 ID 段为例列出。

表 3-1　AXI4 总线全局信号

信号名称	源	描　　述
ACLK	时钟	全局时钟信号
ARESETn	复位	全局复位信号，低电平有效

表 3-2　AXI4 总线写地址通道信号

信号名称	源	描　　述
AWID[3:0]	主机	写地址 ID
AWADDR[31:0]	主机	写地址
AWLEN[3:0]	主机	突发式写的长度，此长度决定突发式写传输数据的个数
AWSIZE[2:0]	主机	突发式写的大小
AWBURST[1:0]	主机	突发式写的类型
AWLOCK[1:0]	主机	锁类型
AWCACHE[3:0]	主机	高速缓存类型。该信号指明事务的 bufferable、cacheable、write-through、write-back、allocate attributes 等信息
AWPROT[2:0]	主机	保护类型
AWVALID	主机	写地址有效。取值如下： 1 ＝地址和控制信息有效 0 ＝地址和控制信息无效 该信号会一直保持，直到 AWREADY 信号变为高
AWREADY	设备	写地址准备好。该信号用来指明设备已经准备好接收地址和控制信息。取值如下： 1 ＝设备已准备好 0 ＝设备未准备好

表 3-3　AXI4 总线写数据通道信号

信号名称	源	描　　述
WID[3:0]	主机	写 ID。WID 的值必须与 AWID 的值匹配
WDATA[31:0]	主机	写的数据
WSTRB[3:0]	主机	写触发。WSTRB[n]标示的区间为 WDATA[8n＋7:8n]
WLAST	主机	写的最后一个数据
WVALID	主机	写有效。取值如下： 1 ＝写数据和写触发有效 0 ＝写数据和写触发无效
WREADY	设备	写就绪。指明设备已经准备好接收数据。取值如下： 1 ＝设备已就绪 0 ＝设备未就绪

表 3-4　AXI4 总线写响应通道信号

信号名称	源	描　　述
BID[3:0]	设备	写响应 ID。BID 值必须与 AWID 的值匹配
BRESP[1:0]	设备	写响应。该信号指明写事务的状态。可能的响应有 OKAY、EXOKAY、SLVERR、DECERR
BVALID	设备	写响应有效。取值如下： 1 ＝写响应有效 0 ＝写响应无效
BREADY	主机	接收响应就绪。该信号表示主机已经能够接收响应信息。取值如下： 1 ＝主机已就绪 0 ＝主机未就绪

表 3-5　AXI4 总线读地址通道信号

信号名称	源	描　　述
ARID[3:0]	主机	读地址 ID
ARADDR[31:0]	主机	读地址
ARLEN[3:0]	主机	突发式读长度
ARSIZE[2:0]	主机	突发式读大小
ARBURST[1:0]	主机	突发式读类型
ARLOCK[1:0]	主机	锁类型
ARCACHE[3:0]	主机	高速缓存类型
ARPROT[2:0]	主机	保护类型
ARVALID	主机	读地址有效。该信号一直会保持，直到 ARREADY 为高。取值如下： 1 ＝地址和控制信息有效 0 ＝地址和控制信息无效
ARREADY	设备	读地址就绪。指明设备已经准备好接收数据。取值如下： 1 ＝设备已就绪 0 ＝设备未就绪

表 3-6　AXI4 总线读数据通道信号

信号名称	源	描　　述
RID[3:0]	设备	读 ID。RID 的值必须与 ARID 的值匹配
RDATA[31:0]	设备	读的数据
RRESP[1:0]	设备	读响应。该信号指明读传输的状态，包括 OKAY、EXOKAY、SLVERR、DECERR
RLAST	设备	读的最后一个数据

续表

信号名称	源	描　　述
RVALID	设备	读数据有效。取值如下： 1 ＝读数据有效 0 ＝读数据无效
RREADY	主机	读数据就绪。取值如下： 1 ＝主机已就绪 0 ＝主机未就绪

表 3-7　AXI4 总线低功耗接口信号

信号名称	源	描　　述
CSYSREQ	时钟控制器	系统低功耗请求。该信号使外部设备进入低功耗状态
CSYSACK	外部设备	低功耗请求应答
CACTIVE	外部设备	取值如下： 1 ＝外部设备时钟有请求 0 ＝外部设备时钟无请求

5. AXI4 基本的读写传输过程

AXI4 总线协议的 5 个通道都使用相同的 VALID/READY 握手机制来传输数据或控制信息。传输时，主机源产生 VALID 信号来指明何时数据或控制信息有效，而设备源则产生 READY 信号来指明已经准备好接收数据或控制信息。只有当 VALID 和 READY 信号同时有效时，才会发生真正的数据或控制信息传输。

VALID 和 READY 信号之间可能出现以下 3 种时序关系：

(1) VALID 先变高，READY 后变高，如图 3-25(a)所示。

(2) READY 先变高，VALID 后变高，如图 3-25(b)所示。

(3) VALID 和 READY 信号同时变高，在这种情况下，数据或控制信息可立即进行传输，如图 3-25(c)所示。

读/写地址通道、读/写数据通道和写响应通道之间的关系是灵活的。例如，写数据通道中的数据可以早于与其相关联的写地址信号出现在接口上，也有可能写数据通道中的数据与写地址信号在同一个时钟周期中同时出现。但是，不管是什么情况，下述两种关系必须被保持：

(1) 读数据通道中的数据必须总是跟在与其相关联的地址之后。

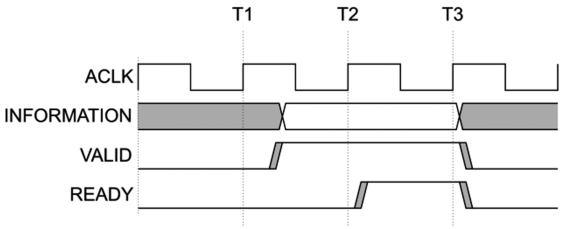
(2) 写响应通道中的信息必须总是跟在与其相关联的写事务的最后出现。

读传输事务信号之间的依赖关系如图 3-26 所示。它们之间必须满足下面的依赖关系：

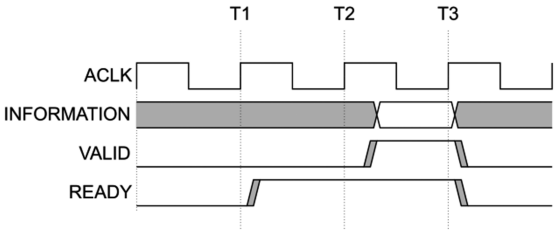
(1) 从设备可以在 ARVALID 信号出现时再给出 ARREADY 信号；也可以先给出 ARREADY 信号，再等待 ARVALID 信号有效。

(2) 从设备必须等待 ARVALID 和 ARREADY 信号都有效时才能给出 RVALID 信号，并且开始数据传输。

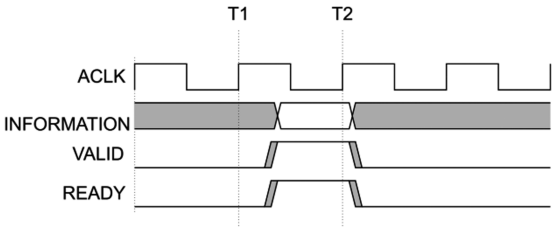
写传输事务信号之间的依赖关系如图 3-27 所示。它们之间必须满足下面的依赖关系：



(a) VALID 先变高，READY 后变高



(b) READY 先变高，VALID 后变高



(c) VALID 和 READY 信号同时变高

图 3-25　VALID 和 READY 信号的 3 种时序关系

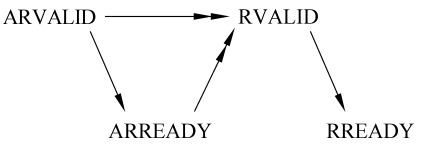


图 3-26　读传输事务信号之间的关系

（1）主设备要确保不能等待从设备先给出 AWREADY 或 WREADY 信号后再给出 AWVALID 或 WVALID 信号。

（2）从设备可以等待 AWVALID 或 WVALID 信号中的一个或者两个有效之后再给出 AWREADY 信号。

（3）从设备可以等待 AWVALID 或 WVALID 信号中的一个或者两个有效之后再给出 WREADY 信号。

6. AXI4 突发式读写传输过程

AXI4 突发式读写传输过程需要满足以下几个条件：

（1）突发式读写的地址必须以 4KB 为基准对齐。

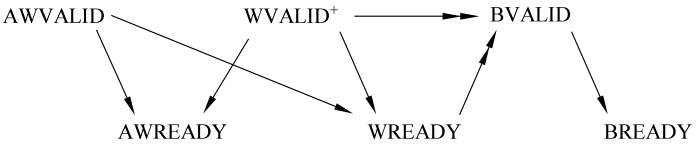


图 3-27　写传输事务信号之间的关系

(2) 由 AWLEN 信号或 ARLEN 信号指定每一次突发式读写传输的数据个数。

(3) 由 ARSIZE 信号或 AWSIZE 信号指定每一个时钟节拍传输的数据的最大位数。要注意的是,任何一次传输的数据位数都不能超过数据总线的总宽度。

AXI4 总线协议定义了 3 种突发式读写的类型,用 ARBURST 或 AWBURST 信号选择突发式读写的类型:

(1) 固定式突发读写(FIX)。地址是固定的,每一次传输的地址都不变。这种突发式读写对一个相同的位置重复进行存取,例如 FIFO。

(2) 增值式突发读写(INCR)。每一次读写的地址都比上一次的地址增加一个固定的值。

(3) 包装式突发读写(WRAP)。与增值式突发读写类似,只不过包装式突发读写的地址不一定从包数据的低地址开始,而且当地址增加到包数据边界时会自动转换到包数据的低地址。

包装式突发读写有两个限制:

(1) 起始地址必须以传输的位数为基准对齐。

(2) 突发式读写的长度必须是 2、4、8 或者 16 位。

7. AXI4 读写传输事务的设备响应

AXI4 总线协议规定对于读事务和写事务都必须有响应。读事务将读响应信号与读的数据一起发送给主设备,而写事务将写响应信号通过写响应通道传送。读写传输事务通过 RRESP[1:0]和 BRESP[1:0]信号对响应信息进行编码。

AXI4 总线协议的响应类型有正常存取成功(OKAY)、独占式存取(EXOKAY)、从设备错误(SLVERR)和译码错误(DECERR)4 种。

AXI4 总线协议还规定,当有错误响应信息时,需要传输的数据会被正常传输,即在一次突发式读写传输事务中,即使有错误响应发生,剩余数据的传输也不会被取消。

8. AXI4 读写传输事务 ID

AXI4 总线协议用事务 ID 标签(tag)支持多地址和乱序传输。事务 ID 标签主要由以下 5 个事务 ID 构成:

(1) AWID。写地址 ID 标签。

(2) WID。在写事务中,主设备传送 WID 以匹配与地址一致的 AWID。

(3) BID。在写响应事务中,从设备会传送 BID 以匹配与 AWID 和 WID 一致的事务。

(4) ARID。读地址 ID 标签。

(5) RID。在读事务中,从设备传送 RID 以匹配与 ARID 一致的事务。

主设备可以使用一个事务的 ARID 或者 AWID 提供的附加信息对请求进行排序。事务

排序规则如下：

（1）从不同主设备传输的事务没有先后顺序限制，它们可以以任意顺序完成。

（2）从同一个主设备传输的不同 ID 的事务也没有先后顺序限制，它们可以以任意顺序完成。

（3）AWID 相同的写事务的数据必须按照顺序依次写入主设备发送的地址。

（4）ARID 相同的读事务的数据必须遵循下面的顺序读出：当从相同从设备读 ARID 相同的读事务的数据时，从设备必须确保数据按照相同的顺序接收；当从不同的从设备读 ARID 相同的读事务的数据时，在接口处必须确保数据按照主设备发送顺序相同的顺序接收。

（5）在 AWID 和 ARID 相同的读事务和写事务之间没有先后顺序限制。如果主设备有顺序要求，那么必须确保第一个事务完全完成后才开始执行第二个事务。

当一个主设备端口与互连结构相连时，互连结构会在 ARID、AWID、WID 信号中添加一位，从而使得每一个主设备端口都是独一无二的。这样做有两个作用：

（1）主设备不必知道其他主设备的 ID。

（2）从设备端口处的 ID 段比主设备端口处的 ID 段宽。

对于读数据，互连结构会附加一位到 RID 中，用来判断哪个主设备端口读取数据。互连结构在将 RID 的值送往正确的主设备端口之前会移除 RID 中后添加的一位。

3.3.2　PWM_w_Int_v1_0 模块部分源码

```
module PWM_w_Int_v1_0 #
(
    // Users to add parameters here
    parameter integer PWM_PERIOD =20,
    // User parameters ends
    // Do not modify the parameters beyond this line


    // Parameters of Axi Slave Bus Interface S00_AXI
    parameter integer C_S00_AXI_DATA_WIDTH  =32,
    parameter integer C_S00_AXI_ADDR_WIDTH  =4
)
(
    // Users to add ports here
    output wire Interrupt_out,
    output wire [1:0] LEDs,
    output wire [PWM_PERIOD-1:0] PWM_Counter,
    output wire [31:0] DutyCycle,
    // User ports ends
   …
    .slv_reg0(DutyCycle)
);
```

```
// Add user logic here
PWM_Controller_Int #(
  .period(PWM_PERIOD)
) PWM_inst (
  .Clk(s00_axi_aclk),
  .DutyCycle(DutyCycle),
  .Reset(s00_axi_aresetn),
  .PWM_out(LEDs),
  .Interrupt(Interrupt_out),
  .count(PWM_Counter)
);
// User logic ends
```

3.3.3 PWM_w_Int_v1_0_S00_AXI 模块部分源码

```
// Users to add ports here
output reg [C_S_AXI_DATA_WIDTH-1:0]     slv_reg0,
// User ports ends
…
//--Number of Slave Registers 4
//reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg0;
reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg1;
reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg2;
reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg3;
```

3.3.4 PWM_Controller_Int 模块部分源码

```
`timescale 1ns / 1ps
module PWM_Controller_Int #(parameter integer period =20)
(
    input Clk,
    input [31:0] DutyCycle,
    input Reset,
    output reg [1:0] PWM_out,
    output reg Interrupt,
    output reg [period-1:0] count
);
    // Sets PWM Period.  Must be calculated vs. input clk period
    // For example, setting this to 20 will divide the input clock by 2^20, or 1 Million
    // So a 50 MHz input clock will be divided by 1e6, thus this will have a period of 1/50
    always @(posedge Clk)
        if (!Reset)
            count <=0;
        else
            count <=count +1;
```