9 使用 CPU 实现对 FPGA 侧 RAM 数据读写

9.1. 背景介绍

在本章实验中，将 FPGA 侧的 RAM 作为从机挂到 AHB 总线上，Cortex-M3 作为主机对 RAM 空间进行读写操作。

实验中将通过 Verilog 语言对 AHB 总线的接口和时序进行描述，然后再通过 MDK 软件对 AHB 接口对应的寄存器地址进行读写操作。

9.1.1. AHB Slave 基本设计框架

本次实验的 AHB Slave 基本设计框图如下图 9. 1 所示。

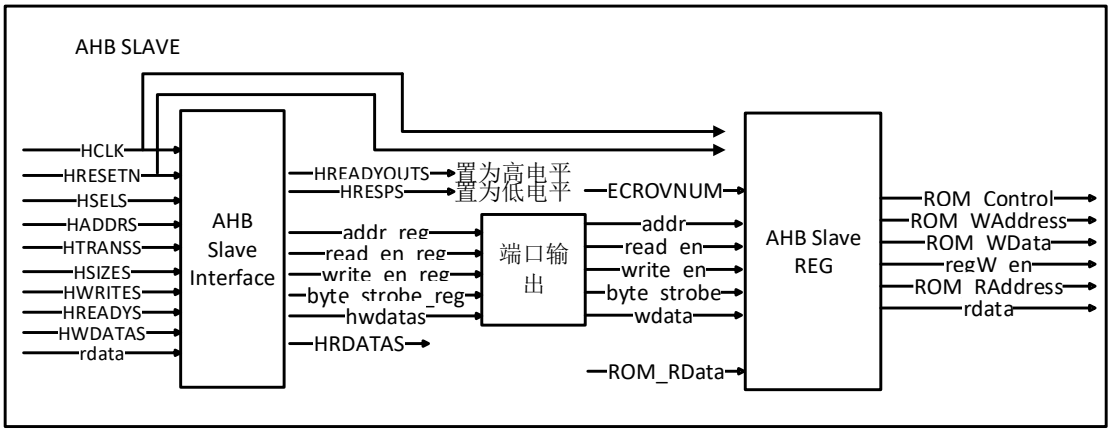

图 9. 1 AHB Slave 基本设计框图

从上图可以看出，AHB Slave 分成两个模块进行设计，一个是 AHB Slave Interface，另一个是 AHB Slave REG。下面就这两部分进行说明。

9.1.1.1. AHB Slave Interface

AHB Slave Interface 模块的作用是将 AHB 总线协议转换成简单的寄存器读写协议。该模块的输入信号连接的是 AHB 总线扩展端口 0，对于这些输入信号的说明如下表 9- 1 所示：

表 9- 1 AHB Slave Interface 模块输入信号说明表

信号名称	来源	信号描述
HCLK	时钟源	为所有总线传输提供基准频率，所有信号的时序都和 HCLK 的上升沿有关。
HRESETN	复位控制器	总线复位信号，低电平有效，用于复位系统和总线。
HSELS	译码器	从机选择信号，表示当前传输的是否是被选中的从机。
HADDRS	主机	32 位的地址总线，总线扩展端口 0 的基地址为 0xA0000000。
HTRANSS	主机	当前传输类型：

		00:IDLE：主设备占用总线，但没进行传输。 01:BUSY：主设备占用总线，但是在传输过程中还没有准备好进行下一次传输。 10:NONSEQ：表明一次单个数据的传输或者传输的第一个数据地址和控制信号与上一次传输无关。 11:SEQ：传输接下来的数据地址，和上一次传输的地址是相关的，这时总线上的控制信号应当与之前的保持一致，地址视情况递增或者回环。
HSIZES	主机	传输数据的比特位的大小。从 0-7 分别对应:8bits(byte), 16bits(halfword), 32bits(word),64bits, 128bits, 256bits, 512bits, 1024bits
HWRITES	主机	传输方向：高电平写，低电平读
HREADYS	多路选择器	为高电平时，表示主机和所有从机传输已完成
HWDATAS	主机	写数据总线用来在写操作期间将数据从主机传输到从机。建议最小的数据总线宽度为 32 位。
rdata	从机	读数据总线，Slave 到 Master。

上表中的信号是由总线主设备产生的，AHB Slave Interface 模块将接收到的信号进行处理，转换成正确的读写控制信号进行输出，输出信号说明如下表 9- 2 所示。

表 9- 2 AHB Slave Interface 模块输出信号说明表

信号名称	信号描述
hreadyouts	高电平表示传输已完成，低电平表示可以进行下一次传输。
hresps	Slave 发给 Master 的总线传输状态，为 0 表示从机处于 OKAY 状态
hrdatas	读数据总线，读取到的从机的数据
addr	地址信号的设定，32 位的地址信号
read_en	总线的读使能
write_en	总线的写使能
byte_strobe	处理器 AHB-Lite 读/写数据字节通道设定，由 hsizes 信号和 haddrs 的低 2 位共同控制
wdata	写数据总线，数据从主机传送至主机

9.1.1.2. AHB Slave REG

AHB Slave REG 模块就是将 AHB Slave Interface 模块输出的信号进行处理，得到控制 RAM 读写的信号。AHB Slave REG 模块信号说明如下表 9- 3 所示。

表 9- 3 AHB Slave REG 模块信号说明表

信号名称	信号类型	信号描述
hclk	输入	为所有总线传输提供基准频率，所有信号的时序都和 HCLK 的上升沿有关
hresetn	输入	总线复位信号，低电平有效，用于复位系统和总线。
addr	输入	地址信号的设定，32 位的地址信号
read_en	输入	总线的读使能
write_en	输入	总线的写使能
byte_strobe	输入	传输数据字节位数的选择，由 hsizes 信号和 haddrs 的低 2 位共同控制
wdata	输入	写数据总线，数据从主机传送至从机
ecorevnum	输入	工程变更顺序修正位

ROM_RData	输入	从从机中读取到的数据
rdata	输出	读数据总线，数据从从机传输至主机
ROM_Control	输出	寄存器的控制信号
ROM_WAddress	输出	寄存器写地址
ROM_WData	输出	寄存器写数据
regW_en	输出	寄存器写使能
ROM_RAddress	输出	寄存器需要读取的数据的地址

9.1.2. AHB 总线时序简介

基本的 AHB 数据传输时序，大体可以分为三种：零等待传输（no wait state transfer）；等待传输（transfers with wait states）；多重传输（multiple transfer pipeline）。下面就这三种传输的时序简单的介绍一下。

9.1.2.1. 零等待传输

零等待传输的时序图如下图 9. 2 所示。

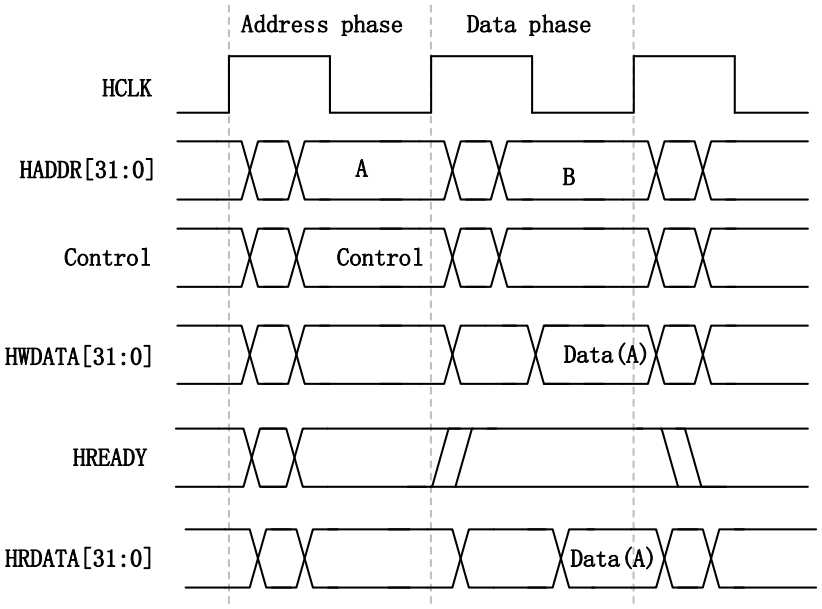

图 9. 2 零等待传输时序图

对于上述时序图的描述如下所示：

第一个周期的上升沿，Master 驱动地址和控制信号；

第二个周期的上升沿，Slave 采样地址和控制信号，并将 HREADY 拉高；如果是写操作，Master 会在第二个周期的上升沿传输要写入的数据；如果是读操作，Slave 会在 HREADY 信号拉高后将读取的数据写入总线；

第三个周期的上升沿，如果是写操作，Master 获取 HREADY 高信号，表明 Slave 已成功接收数据，操作成功；如果是读操作，Master 获取 HREADY 高信号，表明此时的读数据有效并且接收下来，操作成功。

需要注意，HREADY 信号在数据有效期间必须为高，并且延续到第三个周期的上升沿之后，确保 Master 的正确采样。

当 HCLK 上升沿触发之后，Master 送出地址和控制信号；Slave 在 HCLK 下一个上升沿触发时，收到地址和控制信号；在 HCLK 的第三个上升沿触发时，Master 收到 Slave 的响应信号。AHB 支持流水线动作，在收到上一笔数据的同时，可以将下一笔的数据地址送出。

9.1.2.2. 等待传输

当遇到寻址比较慢的 Slave 的时候，传送完成时间会因为等待而延迟。当 HREADY 被 Slave 拉低时，会发生等待状态。需要延迟传送的周期时，就会在数据期间将 HREADY 拉低，此时地址和数据都必须延伸。等待传输的时序如下图 9. 3 所示。

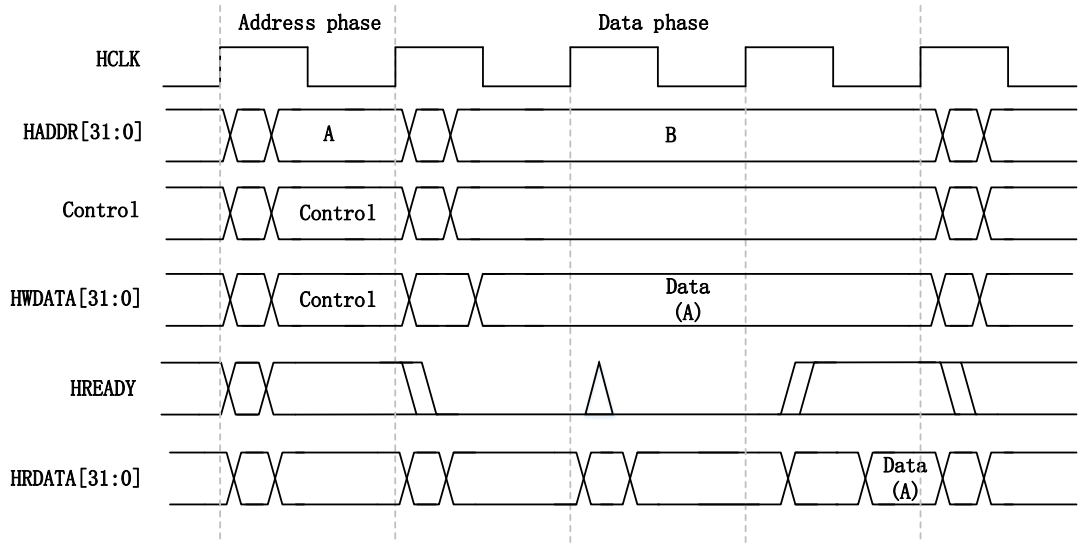

图 9. 3 等待传输时序图

从上图可以看出，在数据模式时，时序必须多花两个周期的延迟等待时间（HREADY 为 0），Slave 才能正常接收到 Master 写的数据或准备好 Master 欲读取的数据。如果时写操作的话，Master 需要在等待期间保持写数据不变，直到本次传输完成；如果是读操作的话，Slave 不需要一开始就给出数据，仅当 HREADY 拉高后才给出有效数据。

9.1.2.3. 多重传送

一次完整的数据传送会有多个传送周期，时序图如下图 9. 4 所示。

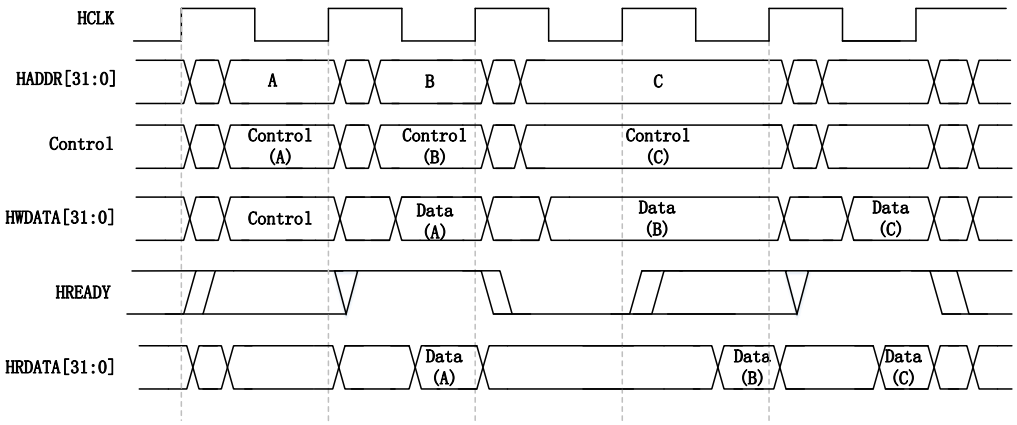

图 9.4 多重传输时序图

由上图可以看出，扩展数据周期就必须延长相应的下一笔传输的地址周期。图中 A 和 C 的数据传输为零等待传输， B 的数据传输加入了一个等待传输，那么相应的 C 的地址周期需要进行扩展。

上图中具体时序描述如下：

第一个周期，Master 发起一个操作 A，并驱动地址和控制信号；

第二个周期，Slave 收到了来自总线的请求，将 HREADY 信号拉高；

第二个周期上升沿后，Master 发现有操作 B 需要执行，并且检查到上一周期的 HREADY 为高，则发起第二个操作 B；

第三个周期，Master 获取 HREADY 信号为高，表示操作 A 已经完成；

第三个周期上升沿后，Master 发现有操作 C 需要执行，并且检查到上一周期的 HREADY 为高，则发起第三个操作 C；

第三个周期上升沿后，Slave 由于繁忙插入了一个等待状态，将 HREADY 拉低；

第四个周期，Master 获取 HREADY 信号为低，知道 Slave 希望等待，于是 Master 保持和上一拍一样的信号；

第四个周期，Slave 处理完了事务，将 HREADY 信号拉高，表示可以继续处理；

第五个周期，Master 获取 HREADY 信号为高，知道 Slave 已经可以处理 B 操作；

第五个周期上升沿后，B 操作完成；

第六个周期上升沿后，C 操作完成。

9.2. 实验介绍

本次实验将通过 Verilog 语言实现对 AHB 协议的时序描述，并且对 CM3 的

AHB 接口信号进行解析，然后利用 MDK 软件对 AHB 接口对应的寄存器地址进行读写操作。将 FPGA 侧的 RAM 作为从机挂载到 AHB 总线上，CM3 作为主机对 RAM 进行读写操作，从而实现 FPGA 和 CM3 的数据交互。实验中将写入和读取到的数据通过串口打印出来，从而判断读写是否成功。

9.3. 建立 HQFPGA 工程

将串口回环实验的工程复制至存放本次实验的文件夹之下，并且对其进行修改，修改方式参考实验二中 2.3 节的内容。

9.4. 添加新的模块文件

9.4.1. 复制 AHB Slave 模块至工程目录中

从我们提供的例程中将 AHB Slave 模块代码复制至自己 HQ 工程的 rtl 文件夹下，如下图 9. 5 所示。

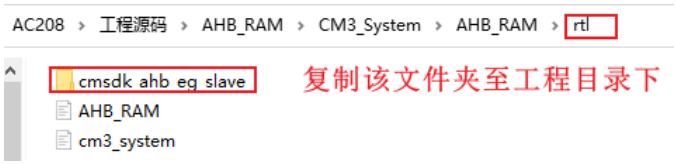

图 9. 5 复制 AHB Slave 模块文件至工程目录下

9.4.2. HQ 软件中添加模块

打开本次工程的 HQ 工程，点击工程属性，在源文件一栏中点击“+”，找到模块代码进行添加，模块中一共有 3 个.v 文件，都需要添加，操作方式如下图 9. 6 所示。

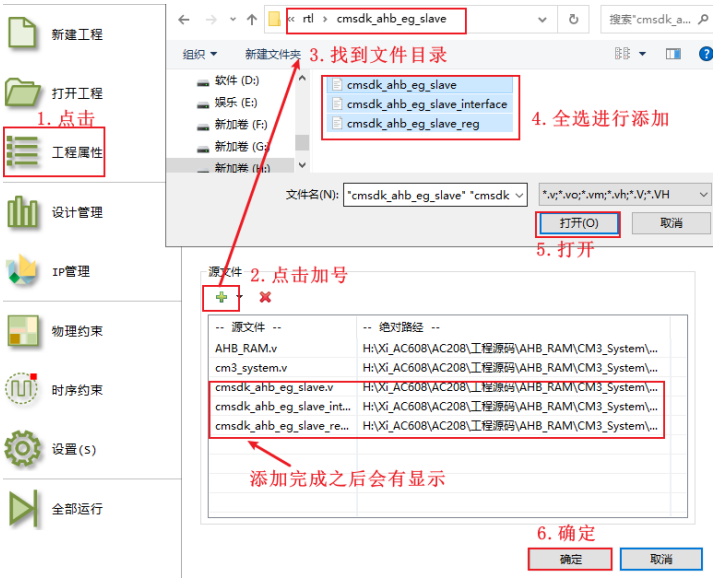

图 9. 6 在 HQ 工程中添加模块代码

9.4.3. 添加 RAM IP

本次实验需要添加 RAM IP 用于存储 CM3 发送过来的数据，点击 IP 管理，找到 Distrubuted_DPRAM 进行添加，具体操作方式如下图 9. 7 所示。

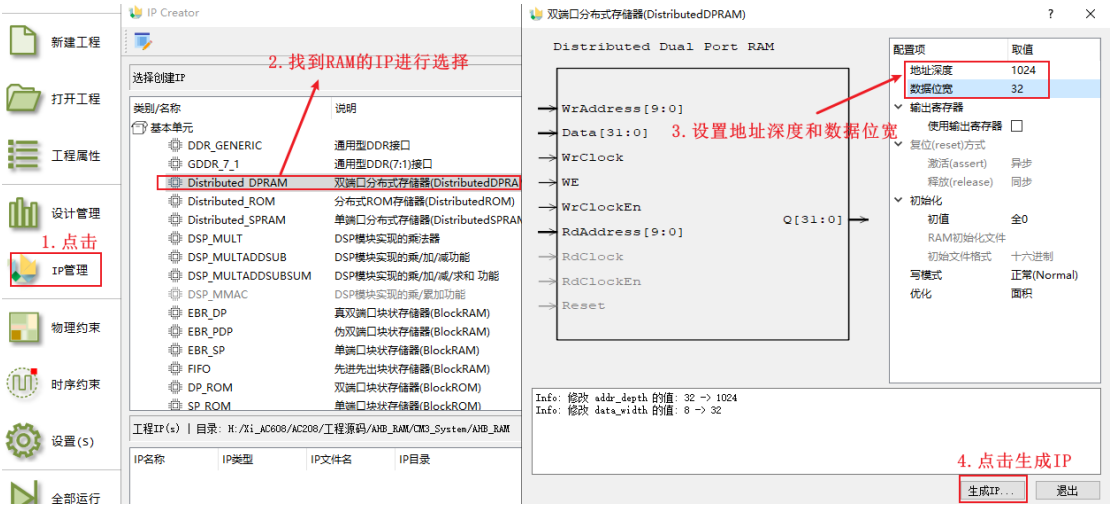

图 9. 7 添加 RAM IP

点击生成 IP 之后，会如下图 9. 8 所示的保存 IP 的界面，保存 IP 之后会在工程目录下生成 ipcore_dir 文件夹用以存放生成的 IP 文件。

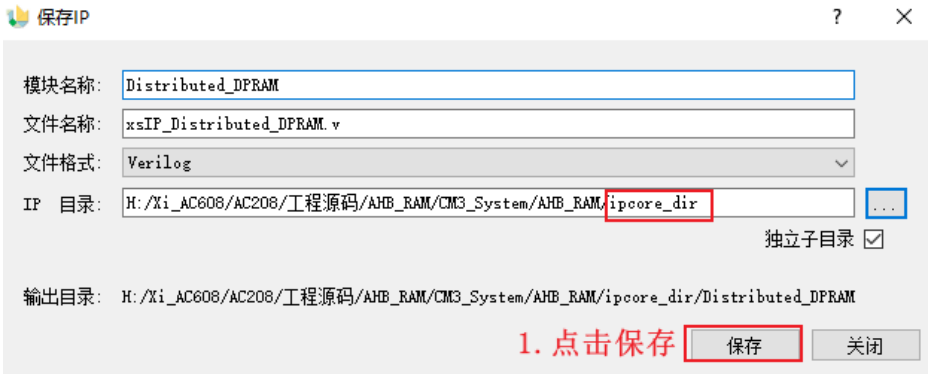

图 9. 8 保存 IP

保存完 IP 之后，在 IP 配置界面会显示完成 IP 生成，然后点击退出，如下图 9. 9 所示。

图 9. 9 退出 IP 配置界面

通过上述步骤完成了 IP 的建立，然后点击工程属性找到 IP 所在位置的.v 文件进行添加，这样在工程中我们才可以调用该 IP，添加步骤如下图 9. 10 所示。实验中还需要添加 PLL 模块，添加方式参考实验二。

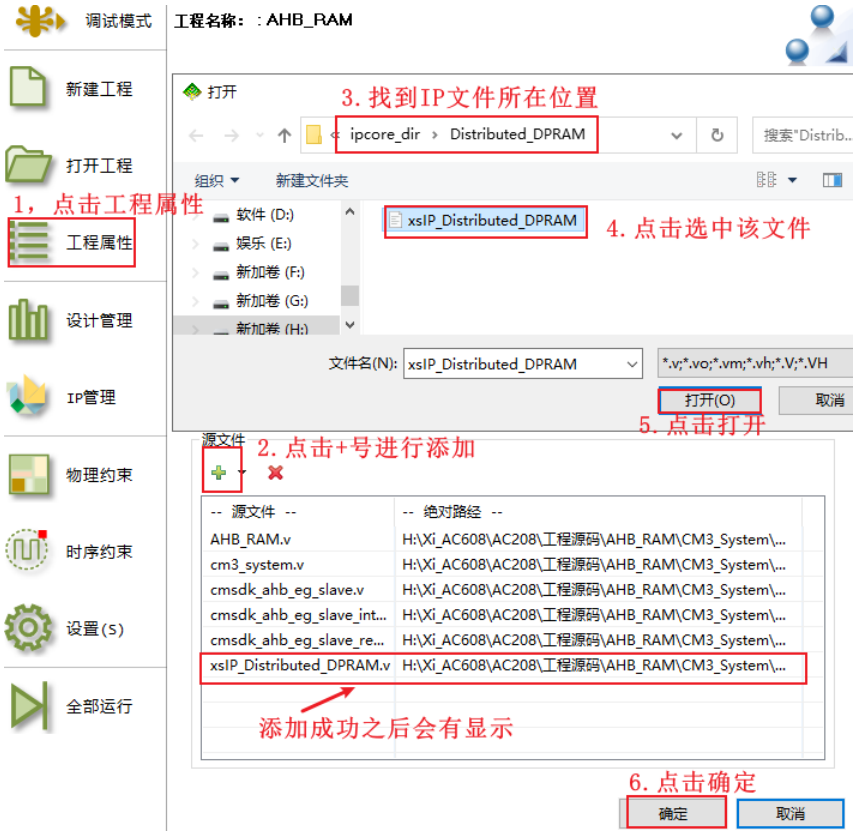

图 9. 10 在工程中添加 IP 文件

9.4.4. AHB SLAVE 模块代码简介

点击设计管理，进入设计管理器，点击设计文件，可以看到我们添加的 AHB Slave 模块的代码文件，如下图 9. 11 所示。

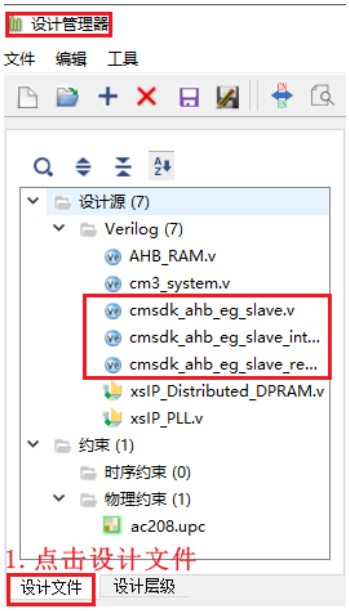

图 9. 11 查看模块代码文件的方式

在 9.1.1 一节中，我们介绍了 AHB Slave 模块的基本设计框架，代码主要就是基于该框架进行实现的。

9.4.4.1. AHB Slave Interface 模块代码实现

AHB Slave Interface 模块的代码通过点击“cmsdk_shb_eg_slave_interface.v”文件进行查看，首先就是对该模块的输入输出信号的定义，具体的输入输出信号的说明请查看 9.1.1.1 节中的内容。信号定义完成之后，根据输入的 AHB 总线上主机给的信号，将其转换成寄存器的基本读写信号，主要进行的操作说明如下：

1. 地址信号的注册

产生传输请求信号之后，将 AHB 总线主机传送过来的总线地址信号设置为寄存器的地址信号。代码设计如下所示：

```
always @(posedge hclk or negedge hresetn)
begin
    if (~hresetn)
        addr_reg <= {(ADDRWIDTH){1'b0}};
    else if (trans_req)
        addr_reg <= haddrs[ADDRWIDTH-1:0];
    end
```

上诉代码中的传输请求信号 trans_reg 的产生和 hreadys、hsels、htranss 信号

有关，当传输结束（hreadys 为 1）、从机被选中（hsels 为 1）并且传输类型为 SEQ 和 NONSEQ（htranss 的第[1]位为 1）的时候产生，代码实现如下所示：

```
wire   trans_req= hreadys & hsels & htranss[1];//传输请求
```

2. 寄存器读取信号的产生

当产生更新读请求信号之后，将总线的读请求信号转换成寄存器的读使能信号，代码实现如下所示：

```
always @(posedge hclk or negedge hresetn)
begin
    if (~hresetn)
    begin
        read_en_reg <= 1'b0;
    end
    else if (update_read_req)
    begin
        read_en_reg  <= ahb_read_req;//总线读请求
    end
end
```

上述代码中的 update_read_req 更新读请求信号分为两种情况产生：第一种就是产生一个有效的读请求的信号，也就是产生了 trans_req 信号并且 AHB 总线产生读请求（hwrites 为低电平）；第二种就是当存在读请求的时候，等待传输结束之后（hreadys 为 1）产生。代码实现如下所示：

```
wire   ahb_read_req  = trans_req & (~hwrites);
assign update_read_req = ahb_read_req | (read_en_reg & hreadys);
```

3. 寄存器写信号的产生

当产生更新写请求信号之后，将总线的写请求信号转换成寄存器的写使能信号，代码实现如下所示

```
always @(posedge hclk or negedge hresetn)
begin
    if (~hresetn)
    begin
        write_en_reg <= 1'b0;
    end
    else if (update_write_req)
    begin
        write_en_reg  <= ahb_write_req;//总线写请求
    end
end
```

上述代码中 update_write_req 写使能信号的产生分为两种方式：第一种就是产生有效的写请求信号，也就是产生了 trans_req 信号并且 AHB 总线产生写请求

（hwrites 为高电平）；第二种就是当已经有写使能信号的时候，等待传输信号产生（hreadys 为 1）。代码实现如下所示：

```
wire   ahb_write_req = trans_req &  hwrites;
assign update_write_req = ahb_write_req |( write_en_reg & hreadys);
```

4. 读/写数据字节通道选择

根据 hsizes 信号和 haddrs 信号的低 2 位设定的不同的值，可以决定传输类型，是字、字节还是半字，具体说明如下表 9- 4 所示。

表 9- 4 处理器 AHB-Lite 读/写数据字节通道

地址阶段		数据阶段				数据传输位宽
HSIZE[2:0]	HADDR[1:0]	HXDATA[31:24]	HXDATA[23:16]	HXDATA[15:8]	HXDATA[7:0]	
000	00	----	----	----	RD[7:0]	8bits
000	01	----	----	RD[15:8]	----	
000	10	----	RD[23:16]	----	----	
000	11	RD[31:24]	----	----	----	
001	00	----	----	RD[15:8]	RD[7:0]	16bits
001	1x	RD[31:24]	RD[15:8]	----	----	
…..	…..	RD[31:24]	RD[15:8]	RD[15:8]	RD[7:0]	32bits

关于上诉表格的代码实现方式请自行对应的源文件，最后在产生更新读请求或者更新写请求的时候，将对应的字节通道给到寄存器的字节通道选择。代码实现如下所示：

```
always @(posedge hclk or negedge hresetn)
begin
    if (~hresetn)
    byte_strobe_reg <= {4{1'b0}};
    else if (update_read_req|update_write_req)
    byte_strobe_reg  <= byte_strobe_nxt;
end
```

最后将主从信号直接相连，将信号输出给 AHB Slave REG 模块进行处理。代码如下所示：

```
assign addr        = addr_reg[ADDRWIDTH-1:0];
assign read_en     = read_en_reg;
assign write_en    = write_en_reg;
assign wdata       = hwdatas;
assign byte_strobe = byte_strobe_reg;
assign hreadyouts  = 1'b1;  // slave always ready
assign hresps      = 1'b0;  // OKAY response from slave
assign hrdatas     = rdata;
```

9.4.4.2. AHB Slave REG 模块代码实现

AHB Slave REG 模块的输入信号是由 AHB Slave Interface 模块的，最后将得

到的信号输出给 RAM，对于其中一部分代码进行说明：

1. 地址写操作地址的解码

在之前我们介绍过 haddrs 的最低 2 位两位传递的信号不是用来表示数据的地址，表示数据地址的只有除去低两位的值。我们通过 addr[11:2]的值以及 write_en 的值确定执行的写操作的寄存器，代码如下所示：

```
assign wr_sel[0] = ((addr[11:2]==10'b0000000000)&(write_en)) ? 1'b1: 1'b0;  //0xA0000000
assign wr_sel[1] = ((addr[11:2]==10'b0000000001)&(write_en)) ? 1'b1: 1'b0;  //0xA0000004
assign wr_sel[2] = ((addr[11:2]==10'b0000000010)&(write_en)) ? 1'b1: 1'b0;  //0xA0000008
assign wr_sel[3] = ((addr[11:2]==10'b0000000011)&(write_en)) ? 1'b1: 1'b0;  //0xA000000c
assign wr_sel[4] = ((addr[11:2]==10'b0000000100)&(write_en)) ? 1'b1: 1'b0;  //0xA0000010
```

2. 寄存器写操作

我们这里以写数据为例，写数据对应的地址设定为 wr_sel[2]，也就是当从机接收的地址信号为 0xA0000008 的时候，表明此时需要对应的地址写入数据，写入的数据根据字节进行传输，代码如下所示：

```
always @(posedge hclk or negedge hresetn)
begin
    if (~hresetn)
    begin
        regWData <= {32{1'b0}}; // reset data register to 0x00000000
    end
    else if (wr_sel[2])//写数据
    begin
        if (byte_strobe[0])
            regWData[ 7: 0] <= wdata[ 7: 0];
        if (byte_strobe[1])
            regWData[15: 8] <= wdata[15: 8];
        if (byte_strobe[2])
            regWData[23:16] <= wdata[23:16];
        if (byte_strobe[3])
            regWData[31:24] <= wdata[31:24];
    end
end
```

3. 寄存器读操作

当检测到读使能的时候，根据传送过来的 addr 的值，确定需要执行的操作，对于寄存器的定义如下所示：

```
 if (addr[11:5] == 8'h00) begin
    case(addr[4:2])
        3'b000: rdata =  regControl;    //0xA0000000
        3'b001: rdata =  regWAddress;   //0xA0000004
        3'b010: rdata =  regWData;      //0xA0000008
```

```
        3'b011: rdata =  regRAddress;   //0xA000000c
        3'b100: rdata =  regRData;      //0xA0000010
        default: rdata = {32{1'bx}};
    endcase
end
```

4. 写使能

当收到写操作数据寄存器的地址，会产生写使能信号，将该信号最终连接至 RAM 的写使能 WE 上。代码如下所示：

```
always @(posedge hclk or negedge hresetn)
begin
    if (~hresetn)
    begin
        regW_en <= 1'b0; // reset data register to 0x00000000
    end
    else if (wr_sel[2])
    begin
        if(byte_strobe == 4'b1111)//数据为 32bit
        regW_en <= 1'b1;//写使能
        else
        regW_en <= 1'b0;
    end
    else
        regW_en <= 1'b0;
end
```

9.4.5. 例化模块端口

9.4.5.1. cm3_system 文件添加例化代码

点开“cmsdk_shb_eg_slave 文件”，可以看到该模块的输入和输出端口，将该模块例化到 cm3_system 文件中，例化代码如下所示。

```
cmsdk_ahb_eg_slave #(32) ahb_slave(
  .HCLK      (PLL_OUT),// Clock
  .HRESETn   (MTXRSTN),  // Reset
  .ECOREVNUM (4'b0), // Engineering-change-order revision bits

  // AHB connection to master
  .HSELS     (TARGEXP0HSEL),
  .HADDRS    (TARGEXP0HADDR),
  .HTRANSS   (TARGEXP0HTRANS),
  .HSIZES    (TARGEXP0HSIZE),
  .HWRITES   (TARGEXP0HWRITE),
  .HREADYS   (TARGEXP0HREADYMUX),
```

```
  .HWDATAS    (TARGEXP0HWDATA),
  .HREADYOUTS(TARGEXP0HREADYOUT),
  .HRESPS     (TARGEXP0HRESP),
  .HRDATAS    (TARGEXP0HRDATA),

  //ROM W/R Data
  .ROMCONTROL   (),
  .ROMWADDRESS  (ROMWADDRESS),
  .ROMWDATA     (ROMWDATA),
  .regW_en      (regW_en),
  .ROMRADDRESS  (ROMRADDRESS),
  .ROMRDATA     (ROMRDATA)
);
```

上述例化代码中，将地址位宽设置为 32 位的，将总线时钟源和总线复位信号与我们整个系统的时钟和复位信号相连接，AHB 接口信号连接至 AHB 扩展端口 0，将最后得到的寄存器控制端口信号进行导出，在顶层模块中将这些信号连接至 RAM 中。在 cm3_system 文件中需要新加的信号如下所示：

```
output wire [31:0] ROMCONTROL,
output wire [31:0] ROMWADDRESS,
output wire [31:0] ROMWDATA,
output wire        regW_en,
output wire [31:0] ROMRADDRESS,
input  wire [31:0] ROMRDATA
```

9.4.5.2. 顶层模块中添加例化代码

在顶层模块中，首先定义 RAM 端口信号，如下所示：

```
wire[31:0]   ROMWADDRESS;
wire[31:0]   ROMWDATA;
wire         regW_en;
wire[31:0]   ROMRADDRESS;
wire[31:0]   ROMRDATA1;
reg[31:0]    ROMRDATA;
```

添加 cm3_system 模块新添加的端口信号，得到 cm3_system 模块的例化代码如下所示（ROMCONTROL 在本次实验中并未用到可不进行添加）：

```
cm3_system cm3_system(
  .MAIN_CLK(MAIN_CLK),
  .CLKOUT(CLKOUT),
  .PCLK(PCLK),
  .MTX_CLK(MTX_CLK),
  .MTXRSTN(MTXRSTN),
  .PLL_OUT(PLL_Clock),
```

```
  .swdclk(swdclk),
  .swddio(swddio),
  .GPIO(GPIO),
  .ROMCONTROL(),
  .ROMWADDRESS(ROMWADDRESS),
  .ROMWDATA(ROMWDATA),
  .regW_en(regW_en),
  .ROMRADDRESS(ROMRADDRESS),
  .ROMRDATA(ROMRDATA)
);
```

在顶层模块中，首先找到之前添加 RAM IP，右击点击打开，就可以看到具体的代码内容，操作如下图 9. 12 所示。

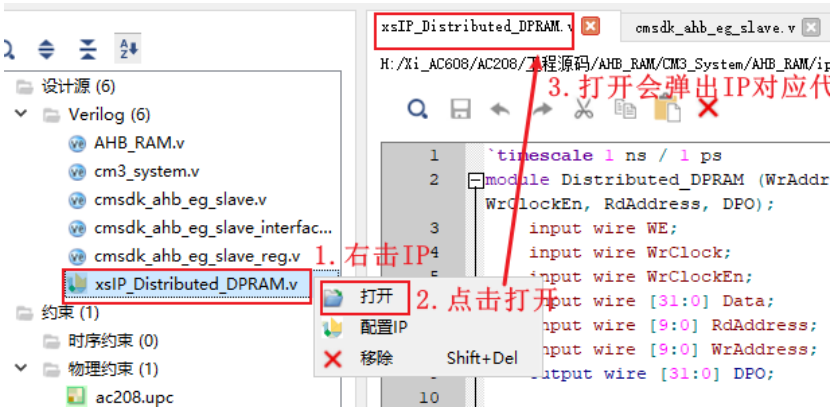

图 9. 12 HQ 软件中查看 IP 具体代码的操作示意图

将添加的 RAM IP 端口信号例化至顶层代码中，例化代码如下所示，模块的输入时钟为 PLL 的输出时钟：

```
Distributed_DPRAM U_Distributed_DPRAM(
  .WrAddress(ROMWADDRESS[9:0]),
  .Data(ROMWDATA),
  .WrClock(PLL_Clock),
  .WE(regW_en),
  .WrClockEn(1'b1),
  .RdAddress(ROMRADDRESS[9:0]),
  .Q(ROMRDATA)
);
```

9.5. 物理管脚约束

本次实验需要使用到的就是串口，确保串口的引脚未分配错误。这里我们选择串口 0，对于串口 0 的引脚分配如下图 9. 13 所示：

```
#-------uart1---------
#uart1_rxd
phycst.pin.set {GPIO[4]}
#uart1_txd
phycst.pin.set {GPIO[5]}
#-------uart0---------
#uart0_rxd
phycst.pin.set {GPIO[2]} C15 -attr "PULLMODE=UP"
#uart0_txd
phycst.pin.set {GPIO[3]} E12 -attr "PULLMODE=UP"
```

图 9. 13 串口引脚分配确定

9.6. 编译设计

保存完修改后的顶层文件和物理约束文件之后，点击全部运行，生成本次实验需要的 FPGA 侧的 bin 文件，操作如图 9. 14 所示。

图 9. 14 编译设计

9.7. 建立 MDK 工程

将已有的 MDK 工程复制至本次工程的文件下，并对其进行修改，操作方式参考实验二中 2.8 节的内容。

9.8. 软件设计

在本次实验中，HADRWARE 文件夹下只需要有 uart 库就可以，其它的不需要使用的都可以删除，删除方式就是将光标放在需要删除的文件上，然后右键 Remove File。删除不需要的文件可以减少代码工程量，从而节省空间，加快代码运行速度。

在本次实验中，Cortex-M3 作为主机挂载在 AHB 总线上，然后通过读写寄存器的方式去控制从机 RAM 内存空间的读写。Coretx-M3 处理器中有两个 AHB 扩展口，对其说明如下表 9- 5 所示。

表 9- 5Coretx-M3 上 AHB 扩展端口说明表

Type	Start	End	Peripheral	Size	Subsystem	Comment

					connection	
Peripherals	0xA000_0000	0xBFFF_FFFF	AHB Expansion0	512MB	TARGEXP0	connected to FPGA fabric
	0xC000_0000	0xDFFF_FFFF	AHB Expansion1	512MB	TARGEXP1	connected to FPGA fabric

本次实验使用到的是扩展端口 0，由上表可以看出其基地址为 0xA0000000。在本次实验中使用 Coretx-M3 去控制不同寄存器读写的时候，就是向对应的寄存器地址进行读写操作（寄存器地址=基地址+地址偏移=0xA0000000+地址偏移），Coretx-M3 需要操作的寄存器地址如何确定，这里以写操作地址寄存器为例：

首先，在 AHB Slave 模块代码中对于写操作地址定义如下所示：

```
if(addr[11:5] == 8'h00) begin
    case(addr[4:2])
        3'b001: rdata =  regWAddress;
    endcase
end
```

可以看到当 addr[4:2]=001 的时候，对应的是写操作地址寄存器，此时对应的寄存器地址=32’b10100000000000000000000000001100=0xA0000004=0xA0000000+0x04，其中 0x04 为写操作寄存器的偏移地址。

程序具体编写思路如下所示：

1. 定义寄存器的偏移地址

在 AHB Slave 模块代码中，对于传输地址进行解析，设置了写地址、写数据、读地址、读数据对应的偏移地址以及对应的 Slave ID（当偏移地址为 0xFFC 的时候，对应的 ID 为 ARM_CMSDK_AHB_EG_SLAVE_CID3，也就是 0xB1），具体设置的代码请自行查看“cmsdk_ahb_eg_slave_reg”文件中的内容。对于寄存器偏移地址定义如下所示：

```
#define WR_ADDR_OFFSET  0x04            //写操作地址寄存器偏移
#define WR_DATA_OFFSET  0x08            //写操作数据寄存器偏移
#define RD_ADDR_OFFSET  0x0C            //读操作地址寄存器偏移
#define RD_DATA_OFFSET  0x10            //读操作数据寄存器偏移
#define FPGA_AHB_SLAVE_ID_OFFSET 0xFFC //FPGA AHB SLAVE ID 偏移
```

2. 写寄存器操作

通过 Coretx-M3 向寄存器中写入值，就是向对应的地址中写入对应的值，函数代码如下所示。

```
void AHB_SlaveWrite(uint32_t *address,uint32_t data)
{
    *address = data;
}
```

3. 读寄存器操作

读寄存器中的值就是读取对应寄存器地址中的值，函数代码如下所示。

```
uint32_t AHB_SlaveRead(uint32_t address)
{
    uint32_t *data;
    data = (unsigned int *)address;
    return *data;
}
```

4. 主函数代码编写

本次实验最终需要实现的功能就是通过 Coretx-M3 去控制 FPGA 侧的 RAM 的读写，并且通过串口打印写入和读出的值。

代码编写：初始化串口，设置波特率为 9600，读取 AHB Slave ID，当 ID 读取成功之后，通过 AHB 向 FPGA 侧 RAM 中循环写入 512 个字节的数据，并且在写入的时候回读写入的数据，将写入和读取的值通过串口打印，看写入和回读的数据是否一致。

FPGA RAM 读写操作流程：写操作的时候，首先是写入需要写的 RAM 地址，然后是写入需要写入地址的数据；读操作的时候，先是写入需要读取的 RAM 的地址，然后读取该地址中的数据。主函数中的代码如下所示：

```
int main()
{
    int i = 0;
    uint32_t rData = 0;
    uint32_t SlaveID = 0;
    //串口初始化
    Uart_Init(CM3DS_MPS2_UART0,9600);
    //读取 AHB Slave ID
    SlaveID = AHB_SlaveRead(CM3DS_MPS2_TARGEXP0_BASE+FPGA_AHB_SLAVE_ID_OFFSET);
    if (SlaveID == 0xB1){
        printf("ahb slave ID ok.\r\n");
        //通过 AHB 读写 FPGA RAM 数据
        for (i = 0; i < 512; i++){
            //往 FPGA RAM 写入数据
            AHB_SlaveWrite((uint32_t *)(CM3DS_MPS2_TARGEXP0_BASE + WR_ADDR_OFFSET), i); //wr_addr
            AHB_SlaveWrite((uint32_t *)(CM3DS_MPS2_TARGEXP0_BASE + WR_DATA_OFFSET), i); //wr_data
            printf("Write:wAddress:%d, wData: 0x%x\r\n", i, i);
            //将写入的数据从 FPGA RAM 回读
            AHB_SlaveWrite((uint32_t *)(CM3DS_MPS2_TARGEXP0_BASE + RD_ADDR_OFFSET), i);
            rData = AHB_SlaveRead(CM3DS_MPS2_TARGEXP0_BASE + RD_DATA_OFFSET);
            printf("Read:rAddress:%d, rData: 0x%x\r\n", i, rData);
        }
    }
```

```
    else{
        printf("ahb slave ID error.\r\n");
    }
}
```

9.9. 板级验证

9.9.1. 实验所需硬件

(1) AC208 开发板
(2) FPGA 下载器：XIST USB Cable
(3) CM3 仿真器：DAP Link
(4) 电源线一根

9.9.2. 硬件连接

硬件连接参考实验一。

9.9.3. 下载文件至目标板

下载文件的方式参考实验一。

9.9.4. 功能演示

将程序下载完成之后，打开串口助手，可以看到如下图 9. 15 所示的界面。

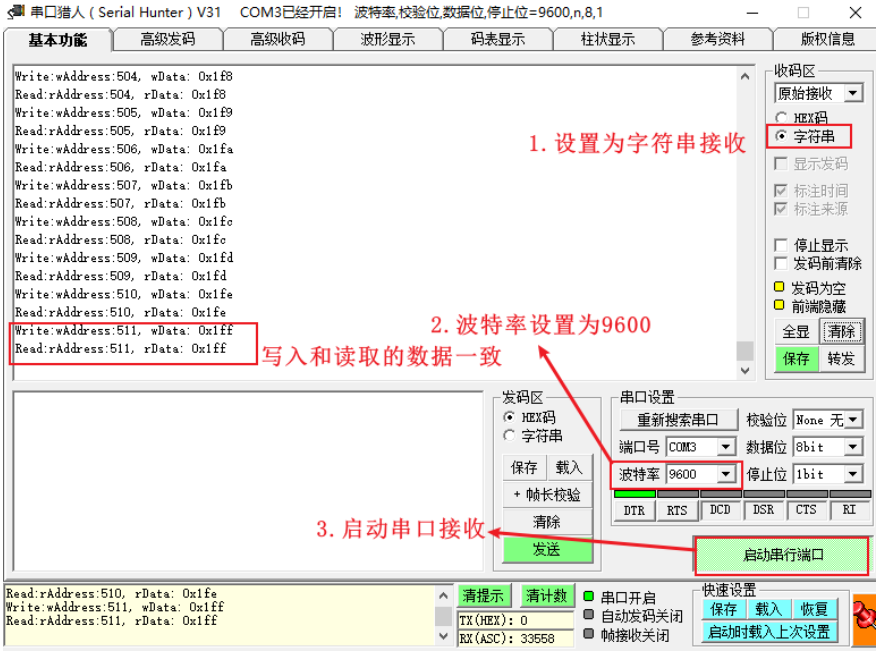

图 9. 15 读写 RAM 结果显示图

由上图可以看出，写入寄存器地址的数据和从该地址回读的数据一致的，这也就说明实验成功实现了读写 RAM 的功能。

9.10. 思考与总结

本次实验将 FPGA 侧的 RAM 作为从机挂载到 AHB 总线上，CM3 作为主机对 RAM 进行读写操作，从而实现 FPGA 和 CM3 的数据交互。最终通过串口显示写入的数据和读取到的数据是一致的。实验中需要注意的是本例程中 FPGA RAM 的地址空间为 0-1023，读写地址不能超过这个范围。