1 使用 SPI 实现 MCU 与 FPGA 之间的数据交互

工程源码	|-ACX720 开发板 |-- acx720_spi_control
相关视频课程	无
说明	

章节导读

本章实验将运用 SPI 协议实现 STM32 和 ACX720 开发板之间的数据交互，实验中将会介绍 SPI 协议的原理、时序图以及如何通过 FPGA 实现 SPI 协议，关于 STM32 的编程我们只会简单说明，更详细的内容请用户自行学习。

1.1 SPI 原理

SPI 是串行外围设备接口（Serial Peripheral Interface）的缩写。串行外围设备接口是一种高速全双工同步的通信总线，被广泛应用于 ADC、LCD 等设备与 MCU 间要求通信速率较高的场合。

SPI 使用 4 根标准信号线进行通信：MISO（主输入-从输出)、MOSI（主输出-从输入)、时钟 SCLK、从机选择信号 SS（有时也称为片选信号 CS）。主从设备四种信号间的关系如图 1-1 所示：

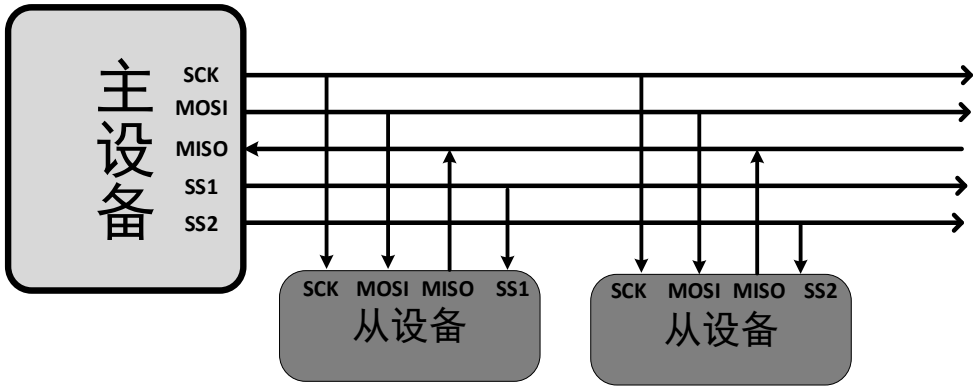

图 1-1 主从设备信号关系

SS（SlaveSelect）：片选信号线，低电平有效。当有多个 SPI 设备与 MCU 相连时，每个设备的这个片选信号线是与 MCU 的引脚单独相连的，而其他的 SCK、MOSI、MISO 线则由多个设备并联到相同的 SPI 总线上。

SCK（Serial Clock）：时钟信号线，由主通信设备产生，不同的设备支持的时钟频率不一样，如 STM32 的 SPI 时钟频率最大为 f PCLK /2。

MOSI（Master Output, Slave Input）：主设备输出/从设备输入引脚。主机的数据从这条信号线输出，从机由这条信号线读入数据，即这条线上数据的方向为主机到从机。

MISO（Master Input, Slave Output）：主设备输入/从设备输出引脚。主机从这条信号线读入数据，从机的数据则由这条信号线输出，即在这条线上数据的方向为从机到主机。

1.1.1 SPI 协议特点

主-从模式：两个 SPI 设备间必须由主机（master）控制从机（slave）。一个主机设备通过提供 SCLK 信号、选择 SS 信号（低电平有效）来控制多个从机设备。因此从机设备是无法主动向主机设备发送数据的，因为 SPI 是一种“时钟驱动”的协议，没有 SCLK 时无法正常工作。

同步传输：主机设备在要交换数据时输出时钟信号，相位 CLK_PHA 和极性 CLK_POL 的不同，配置组成了通常所说的 4 种 SPI 模式。只要主机和从机选择同样的配置，即可完成同步传输。

数据交换：图 1-2 为 CLK_PHA 为 0 模式下的时序图，可以看到每个时钟周期内，MOSI 和 MISO 两根信号上都有数据，即 SPI 设备会同时“发送”和“接收”1 bit 数据，完成数据交换。

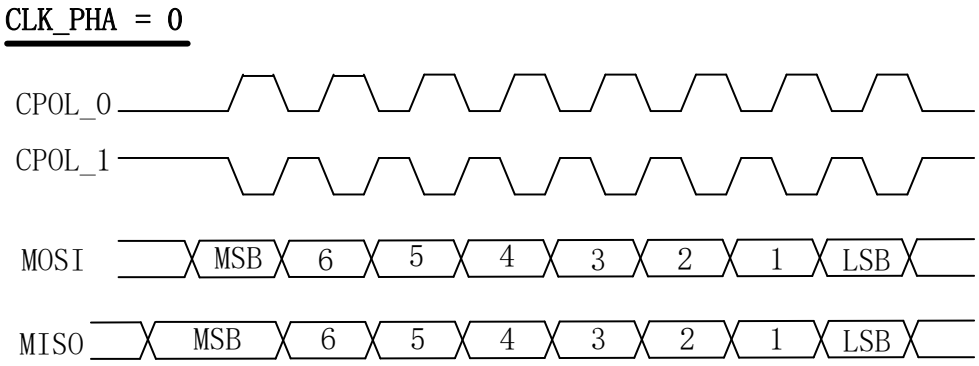

图 1-2 SPI 时序图

1.1.2 SPI 工作模式

SPI 一共有四种工作模式，分别由 CPOL 与 CPHA 决定，其中 CPOL 表示时钟的极性，即空闲状态下时钟线的电平状态，当 CPOL 的值为 0 时，则时钟空闲态的电平为低电平，当 CPOL 的值为 1 时则为高电平。CPHA 表示采样的相位，当 CPHA 为 0 时，数据会从第一个边沿开始采样（注意不是上升沿），当

CPHA 为 1 时，数据会从第二个边沿开始采样。通信双方必须是工作在同一模式下，通过 CPOL（时钟极性）和 CPHA（时钟相位）来控制设备的通信模式，具体如下：

Mode0：CPOL=0，CPHA=0

Mode1：CPOL=0，CPHA=1

Mode2：CPOL=1，CPHA=0

Mode3：CPOL=1，CPHA=1

SPI 工作时序如图 1-3，CS 即片选信号 SS，CPOL 代表时钟 SCK 的极性，CPHA 为 MOSI/MISO 数据传输过程，当 CPOL 为 0 时，则其有效状态为高电平，为 1 时，其有效状态为低电平；当 CPHA 为 0 时，会在第一个边沿对数据进行采样（虚线 1），第二个边沿对数据进行输出（虚线 2）；当 CPHA 为 1 时，会在第二个边沿对数据进行检测（虚线 2），第三个边沿输出（虚线 3）。

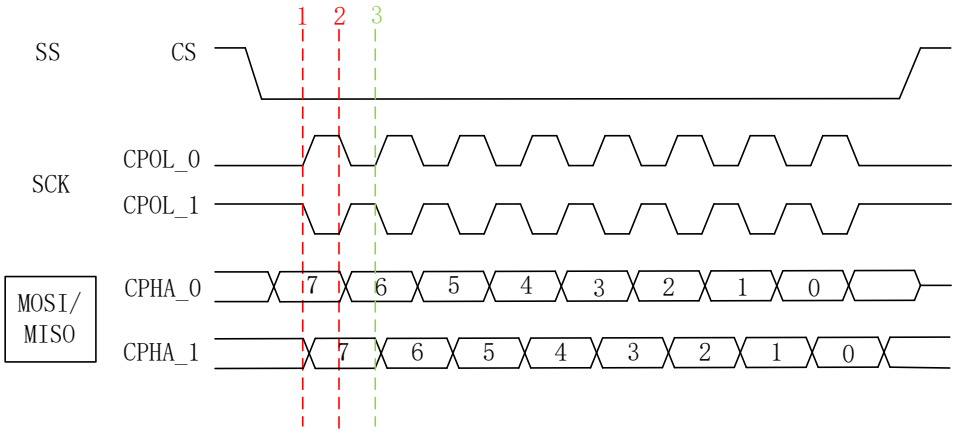

图 1-3 SPI 工作时序图

1.1.3 SPI 总线拓扑结构

SPI 在进行一主多从工作时其连接的拓扑结构有两种，如图 1-4，其中左边为 SPI 主机，通常为 MCU，右边为三个从机，通常为传感器，本次实验为 ACX720 开发板，具体如下：

（1）SPI 总线的 SCLK，MOSI，MISO，与从机共用，片选信号 SS 独立，分别对应连接到各个器件的片选上，由于片选信号相互独立，可以分别控制每一个从器件，通信相互独立，互不干涉，在使用中通常使用该拓扑结构。

（2）SPI 主线的 SCLK，SS 与从机共用，MOSI 连接到第一个从机的 MOSI，第一个从机的 MISO 并不连回主机，而是连接到第二个从机的 MOSI，

以此类推，最后通过最后一个从机的 MISO 连回主机，由于 SS 是共用的，所以从机会同时工作，在一次通信过程中便能把所有从机数据传输给主机，通常适用于串行转并行的场合。

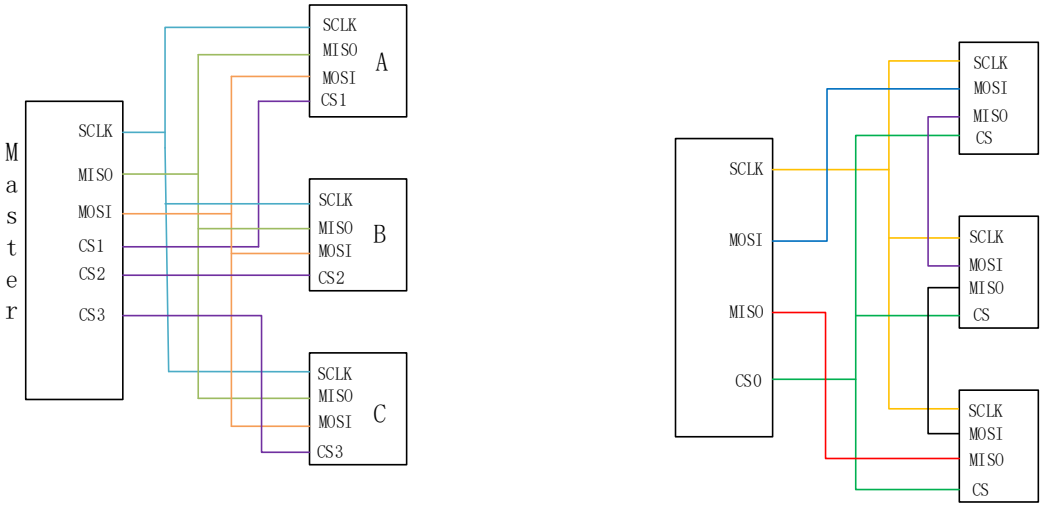

图 1-4 SPI 拓扑结构

1.2 系统整体设计

本次实验，我们将通过 STM32 将数据写入到 FPGA 芯片内部的 RAM 中，然后再将 RAM 中的数据回读，最终将写入和读出的数据通过串口打印出来，判断数据是否一致，从而判断实验是否成功，只有数据一致，才能说明实验成功，实现了 STM32 和 FPGA 之间的数据交互，系统的整体设计如下图 1-5 所示。

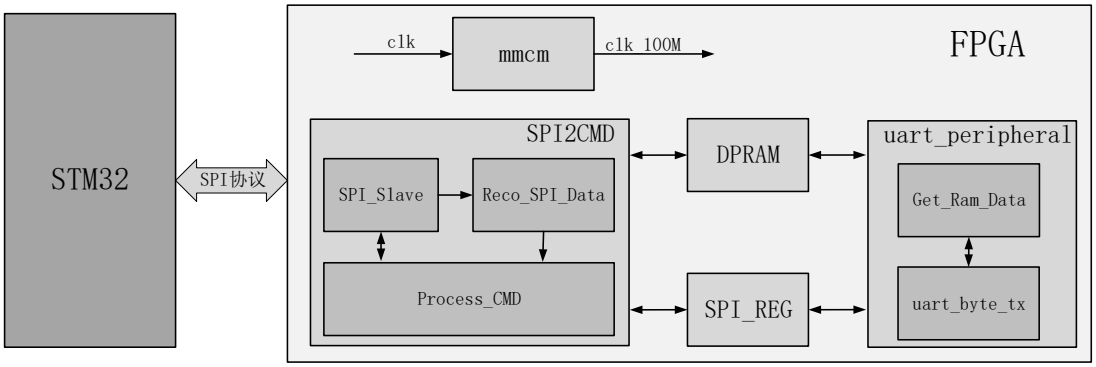

图 1-5 STM32 与 FPGA 数据交互整体设计框图

下面将对上述图中 FPGA 内部各个模块进行简要说明。

1. MMCM：锁相环模块，输入时钟 clk 为 50M，由开发板上的晶振提供，输出 100M 的时钟给到其他模块使用

2. SPI2CMD 模块：将接收的 SPI 的数据进行解析，然后写入进

DPRAM\SPI_REG 中，该模块中包含以下 3 个模块：

- SPI_Slave：SPI 从机模块，将输入的数据以 SPI 的时序发送出去，并解析收到的 SPI 数据。
- Reco_SPI_Data：将SPI_Slave 模块输出的数据按照协议解码，然后输出解码地址+解码数据+读写命令+执行信号(脉冲)交由下一个模块使用。
- Process_CMD：根据Reco_SPI_Data 模块输出的信号来执行相应的读写 RAM/REG 的操作。

3. SPI_REG：双端口 SPI 寄存器模块，内部包含 256 个 8 位寄存器，每个寄存器都支持读写功能。
4. DPRAM：真双端口 RAM IP 模块，用来存储数据。
5. uart_peripheral：串口设备模块，该模块中包含以下两个模块：
   - Get_Ram_Data：得到 RAM 和 REG 中的数据，生成控制串口发送模块的接口信号。
   - uart_byte_tx：串口发送模块，将存储在 RAM 中的数据通过串口发送出去。

除去使用 IP 和前面章节讲过的模块外，还需要设计的模块包括SPI2CMD 模块、SPI_REG 模块和 uart_peripheral 模块。

1.3 模块设计

1.3.1 SPI2CMD 模块设计

SPI2CMD 模块用于解析通过 SPI 协议传输过来的数据，该模块内部封装了 3 个模块，分别是 SPI_Slave 模块、Reco_SPI_Data 模块和 Process_CMD 模块。在模块内部例化这三个模块并进行信号连接，该模块的结构图如下图 1-6 所示。

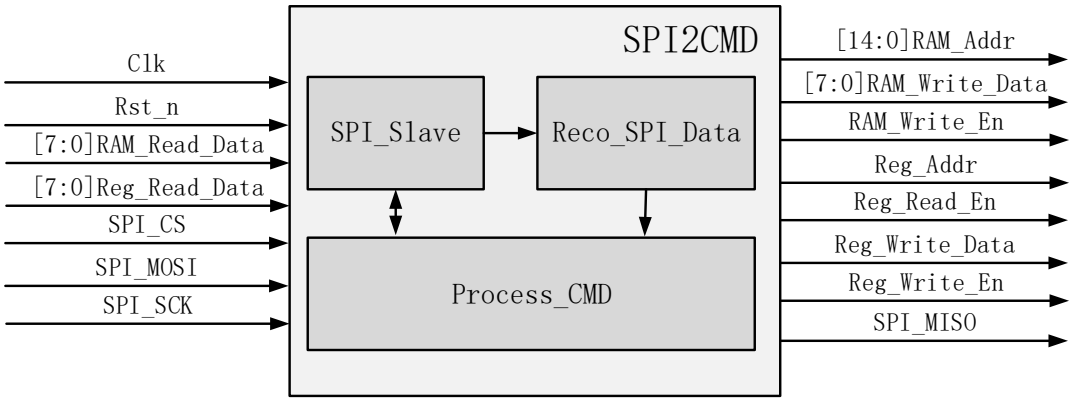

图 1-6 SPI2CMD 模块结构图

该模块的信号说明如下表 1-1 所示：

表 1-1 SPI2CMD 模块信号说明表

信号名称	I/O	信号意义
Clk	I	模块时钟信号，100M
Rst_n	I	模块复位信号，低电平复位
RAM_Read_Data[7:0]	I	从 RAM 中读出的 8 位数据信号
Reg_Read_Data[7:0]	I	从 REG 中读出的 8 位数据信号
SPI_CS	I	SPI 协议的片选信号
SPI_MOSI	I	SPI 协议的 MOSI 信号
SPI_SCK	I	SPI 协议的时钟信号
RAM_Write_Data[7:0]	O	向 RAM 写入的 8 位数据信号
RAM_Write_En	O	RAM 的写使能信号
Reg_Addr[7:0]	O	REG 地址信号
Reg_Read_En	O	REG 的读使能信号
Reg_Write_Data[7:0]	O	向 REG 写入的 8 位数据信号
Reg_Write_En	O	REG 的写使能信号
SPI_MISO	O	SPI 协议的 MISO 信号

在该模块内部，只是将三个模块进行例化然后将端口信号进行连接，所以这里就不贴出代码了，下面将对这 SPI_Slave 模块、Reco_SPI_Data 模块和 Process_CMD 模块进行介绍。

1.3.1.1 SPI_Slave 模块设计

SPI_Slave 模块的主要功能就是作为一个 SPI 从机，将需要发送的数据 Send_Data[7:0]以 SPI 的时序方式进行发送，并解析收到的 SPI 数据，它还会记录从 CS↓到 CS↑SPI 传输的字节数 Trans_Cnt[15:0]，该模块的基本框图如下图 1-7 所示：

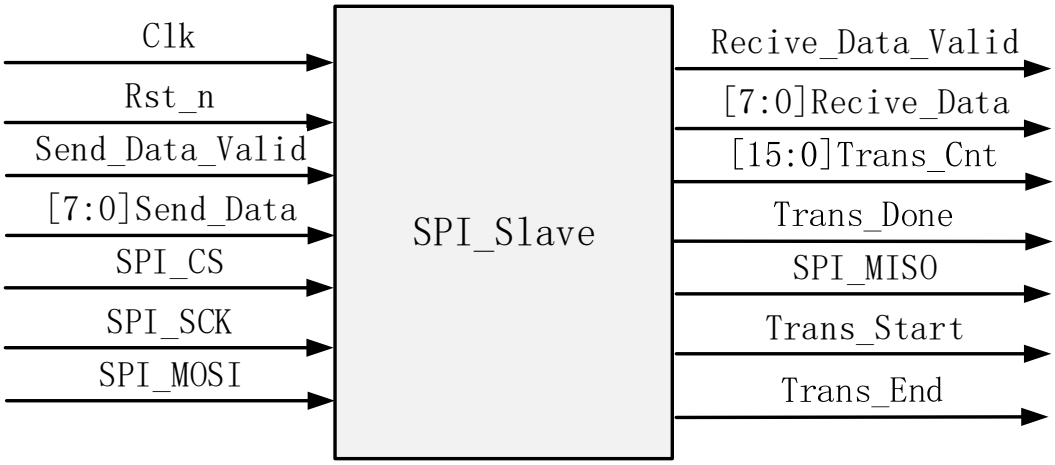

图 1-7 SPI_Slave 模块基本结构框图

该模块的信号说明如下表 1-2 所示。

表 1-2 SPI_Slave 模块的信号说明表

信号名称	I/O	信号意义
Clk	I	模块时钟信号，100M
Rst_n	I	模块复位信号，低电平有效
Send_Data_Valid	I	发送数据有效信号
Send_Data [7:0]	I	需要发送的 8 位数据
SPI_CS	I	SPI 协议的片选信号
SPI_SCK	I	SPI 协议的时钟信号
SPI_MOSI	I	SPI 协议的 MOSI 信号
Recive_Data_Valid	O	接收数据有效信号
Recive_Data[7:0]	O	接收到的 8 位数据
Trans_Cnt [15:0]	O	CS↓到 CS↑SPI 传输的字节数
Trans_Done	O	单字节传输完成标志信号
SPI_MISO	O	SPI 协议的 MISO 信号
Trans_Start	O	传输开始开始信号
Trans_End	O	整个数据传输完成标志信号

首先，我们将完成解码的功能，也就是解析主机通过 SPI_MOSI 传送过来的数据，然后将解析完成的信号 Recive_Data 进行输出。解码时以 SPI_SCK 信号为时钟信号，SPI_CS 为复位信号，在 SCK 每一个上升沿取一位数据，共 8 个上升沿，从高到低取 8 位，即得 Recive_Data[7:0]。解析时的时钟和复位信号代码如下所示：

```
assign SCK_Sel = (CPOL ^ CPHA) ? (SPI_SCK) : (~SPI_SCK);
assign SPI_Reset = (~Rst_n) | SPI_CS;
```

从上述代码中可以看出，SPI_Reset 信号高电平有效，并且与 CS 信号有关，只有当 CS 为低电平时，才没有处于复位状态；解码时的时钟信号还和 CPOL 和 CPHA 有关，也就是和 SPI 的工作模式有关，在前面“SPI 工作模式”介绍过，

SPI 一共有四种工作模式，分别是由 CPOL 和 CPHA 控制的，比如 SPI 工作在模式 0，也就是 CPOL=0、CPHA=0，此时时钟信号 SPI_SCK 空闲状态为低电平，在 SPI_SCK 第一个时钟边沿开始采样，此时 SPI 的工作时序图如下图 1-8 所示。

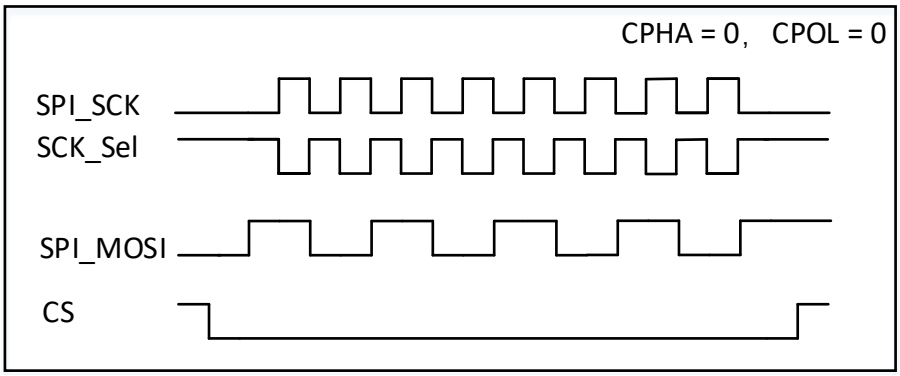

图 1-8 工作模式 0 从机接收时序图

从上图可以看出，此时 MOSI 传输过来的数据为“1010101010（55）”，切换成 SCK_Sel 和 SPI_Reset 信号，也就是检测到 SCK_Sel 下降沿和 SPI_Reset 上升沿的时候开始通过线性序列机的方式解析数据，解析时是从高位到低位开始的，解析完成之后将单字节传输完成信号 Trans_Done 拉高，代码如下所示：

```
//状态机加线性序列机采集 MOSI
always@(negedge SCK_Sel or posedge SPI_Reset)
begin
  if(SPI_Reset) begin
      In_Cnt <= 8'd0;
      Recive <= 8'h00;
      Trans_Done <= 1'b0;
      Trans_Cnt <= 8'h00;
  end
  else begin
    case (In_Cnt)
      8'd0:beginIn_Cnt<=In_Cnt+1'b1;Recive[7]<=SPI_MOSI;Trans_Done<=1'b0;end
      8'd1: begin In_Cnt <= In_Cnt + 1'b1; Recive[6] <= SPI_MOSI; end
      8'd2: begin In_Cnt <= In_Cnt + 1'b1; Recive[5] <= SPI_MOSI; end
      8'd3: begin In_Cnt <= In_Cnt + 1'b1; Recive[4] <= SPI_MOSI; end
      8'd4: begin In_Cnt <= In_Cnt + 1'b1; Recive[3] <= SPI_MOSI; end
      8'd5: begin In_Cnt <= In_Cnt + 1'b1; Recive[2] <= SPI_MOSI; end
      8'd6: begin In_Cnt <= In_Cnt + 1'b1; Recive[1] <= SPI_MOSI; end
      8'd7:begin In_Cnt <= 8'd0;  Recive[0] <= SPI_MOSI; Trans_Done <= 1'b1;
Trans_Cnt <= Trans_Cnt + 1'b1; end
      default: In_Cnt <= 8'd0;
    endcase
  end
end
```

每一位的数据解析完成之后，然后根据代码中设定 BITS_ORDER 的值（1：高位在前；0：低位在前），将每一位的数据组合赋值给 Recive_Data 信号进行输出，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Recive_Data <= 8'h00;
    else if(Done_POS) begin
        if(BITS_ORDER == 1'b1)
            Recive_Data <= Recive;
        else
            Recive_Data<={Recive[0],Recive[1],Recive[2],Recive[3],Recive[4],Recive[5],Recive[6],Recive[7]};
    end
    else
        Recive_Data <= Recive_Data;
    end
```

由于 SCK 与 Clk 存在跨时钟域的情况，因此将 SCK 时钟域里的 Trans_Done 信号打拍取上升沿，得到一个与 Clk 同步的脉冲，作为每个字节传输完成的标志，也就是上述代码中的 Done_POS 信号，实现方式就是将 Trans_Done 信号打两拍得到 Done_R1 和 Done_R2 信号，当 Done_R1 为低电平同时 Done_R2 为高电平时，得到 Trans_Done 的上升沿脉冲信号 Done_POS，如下图 1-9 所示。

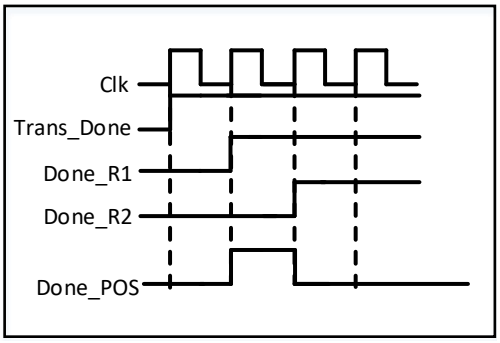

图 1-9 Done_POS 信号产生示意图

产生 Done_POS 信号的代码如下所示：

```
reg Done_R1,Done_R2;
//对 Trans_Done 信号打拍，取上升沿
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n) begin
        Done_R1 <= 8'h00;
        Done_R2 <= 8'h00;
    end
```

```
    else begin
        Done_R1 <= Trans_Done;
        Done_R2 <= Done_R1;
    end
end
wire Done_POS;
assign Done_POS = (~Done_R2) & Done_R1;
```

得到 Done_POS 信号之后，当该信号为高电平的时候，将接收数据有效信号拉高，代码如下所示：

```
//数据有效标志信号
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Recive_Data_Valid <= 1'b0;
    else if(Done_POS)
        Recive_Data_Valid <= 1'b1;
    else
        Recive_Data_Valid <= 1'b0;
end
```

通过前面的讲述，我们将主机传输过来的数据进行解析，并将解析的数据 Recive_Data 进行输出，现在我们需要将待发送的数据 Send_Data 以 SPI 时序的方式发送出去，也就是将数据通过 SPI_MISO 进行发送。

首先，当产生发送数据有效信号 Send_Data_Valid 之后，将需要发送的数据寄存，防止发送途中被修改，并且根据 BITS_ORDER 的值，判断是先发高位还是低位，代码如下所示：

```
//寄存发送的数据，防止发送途中被修改
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Send_Data_R <= 8'h00;
    else if(Send_Data_Valid) begin
        if(BITS_ORDER == 1'b1)
            Send_Data_R <= Send_Data;
        else
            Send_Data_R<={Send_Data[0],Send_Data[1],Send_Data[2],Send_
Data[3],Send_Data[4],Send_Data[5],Send_Data[6],Send_Data[7]};
    end
    else
        Send_Data_R <= Send_Data_R;
end
```

在发送 MISO 数据的时候，数据需要提前准备，以便于 MISO 在 SCK 的上升沿发出数据，因此在 SCK 的第一个上升沿到达之前，取 Send_Data[7]到 MISO，

然后在 SCK 的前 7 个下降沿将 Send_Data[6:0]发送出去。在控制 MISO 发送数据时同样以 SCK_Sel 为时钟信号，SPI_Reset 为复位信号，代码如下所示：

```
assign SPI_MISO = SPI_CS ? 1'b0 : ((Out_Cnt | CPHA) ? MISO : Send_Data_R[7]);
//状态机加线性序列机控制 MISO
always@(posedge SCK_Sel or posedge SPI_Reset)
begin
    if(SPI_Reset) begin
        Out_Cnt <= 8'd0;
    end
    else begin
        case (Out_Cnt)
            8'd0: beginOut_Cnt<=Out_Cnt + 1'b1; MISO <= Send_Data_R[6+CPHA]; end
            8'd1: beginOut_Cnt<=Out_Cnt + 1'b1; MISO <= Send_Data_R[5+CPHA]; end
            8'd2: beginOut_Cnt<=Out_Cnt + 1'b1; MISO <= Send_Data_R[4+CPHA]; end
            8'd3: beginOut_Cnt<=Out_Cnt + 1'b1; MISO <= Send_Data_R[3+CPHA]; end
            8'd4: beginOut_Cnt<=Out_Cnt + 1'b1; MISO <= Send_Data_R[2+CPHA]; end
            8'd5: beginOut_Cnt<=Out_Cnt + 1'b1; MISO <= Send_Data_R[1+CPHA]; end
            8'd6: beginOut_Cnt<=Out_Cnt + 1'b1; MISO <= Send_Data_R[0+CPHA]; end
            8'd7: beginOut_Cnt<=8'd0; MISO <= Send_Data_R[0]; end
            default: Out_Cnt<=8'd0;
        endcase
    end
end
```

从上述代码中可以看出，根据 CPHA 的不同，发送最高字节的方式也不同，我们首先了解一下，根据 CPHA 不同，发送一字节的数据（55）时序，如下图 1-10 所示。

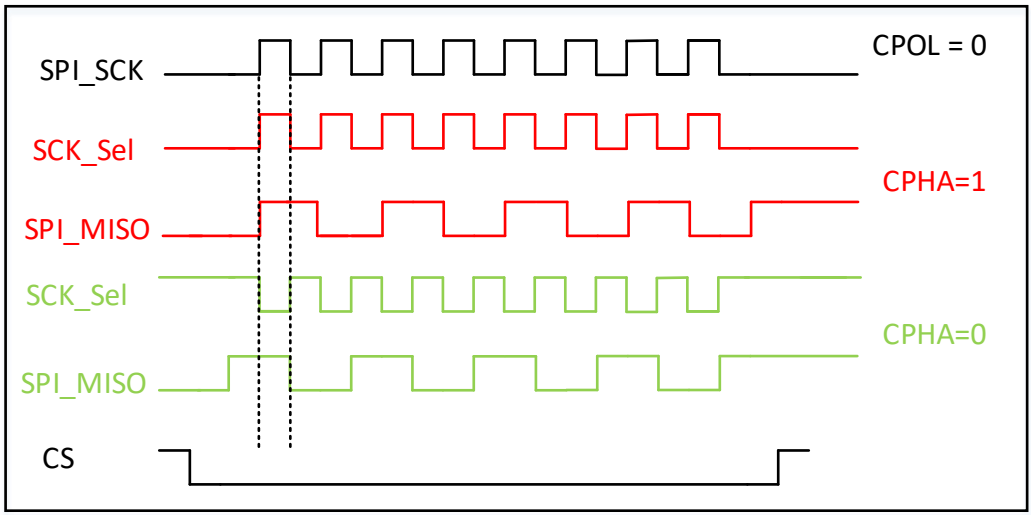

图 1-10 工作模式 0 从机发送时序图

从上图可以看出，当 CPHA=1 的时候，数据从 SPI_SCK 的第二个时钟边沿开始发送；CPHA=0 的时候，数据从 SPI_SCK 的第一个时钟边沿开始发送。这

里尤其需要注意的是当 CPHA 等于 0 的情况，如果使用时序逻辑的方式根据 Out_Cnt 值进行发送的话，在 SPI_SCK 的第一边沿发送数据的时候，会存在晚一拍的情况，导致提前准备的第一个需要发送的数据出错，但是如果我们采用组合逻辑的方式，就不会存在晚一拍的情况，当 Out_Cnt 和 CPHA 都为 0 的时候，将需要发送的数据最高位 Send_Data_R[7]发送出去，这样就确保我们数据到来之后，第一个数据能够准确传送给主机，然后就可以根据 Out_Cnt 的值依次发送剩下的 7 个数据，后面连续发送两次 Send_Data_R[0]不影响主机接收数据。

最后，我们还需要得到传输开始信号 Trans_Start 和传输结束信号 Trans_End，这两个信号可以根据 SPI_CS 的信号得到，当检测到 SPI_CS 下降沿时，得到 Trans_Start 信号；检测到 SPI_CS 的上升沿时，得到 Trans_End 信号，如下图 1-11 所示：

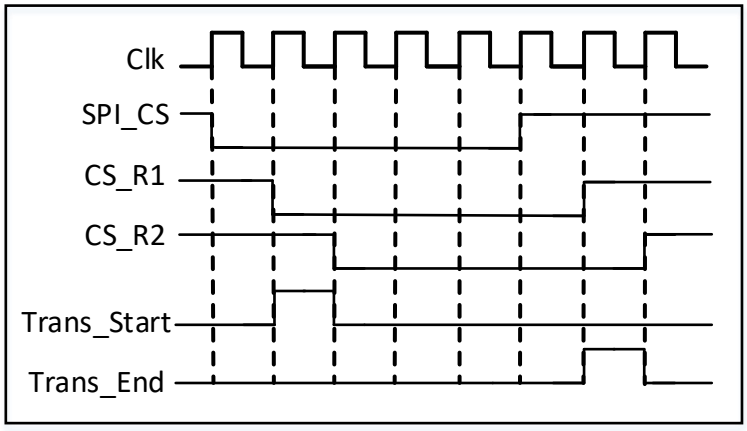

图 1-11 开始/完成信号时序图

从上述图中可以看出，将 SPI_CS 信号打两拍得到 CS_R1 和 CS_R2 信号，当 CS_R1 为低电平并且 CS_R2 为高电平的时候，得到 Trans_Start 信号；当 CS_R1 为高电平并且 CS_R2 为低电平的时候，得到 Trans_End 信号，代码如下所示：

```
reg CS_R1,CS_R2;
//对 CS 信号打拍，取下降沿
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n) begin
        CS_R1 <= 8'h00;
        CS_R2 <= 8'h00;
    end
    else begin
        CS_R1 <= SPI_CS;
```

```
        CS_R2 <= CS_R1;
    end
end
assign Trans_Start = (~CS_R1) & CS_R2;
assign Trans_End = (~CS_R2) & CS_R1;
```

1.3.1.2 Reco_SPI_Data 模块设计

Reco_SPI_Data 模块的功能就是将前面 SPI 收到的数据按照协议解码，然后输出解码地址+解码数据+读写命令+执行信号(脉冲)，它连接的下一个模块每接收到一个执行脉冲就会按照地址和数据以及读写命令来执行相应的操作，该模块的基本结构图如下图 1-12 所示：

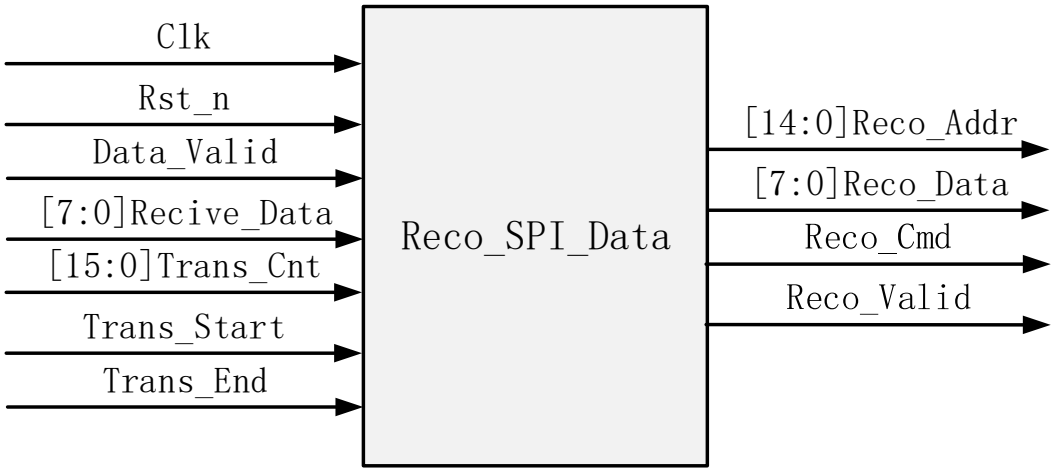

图 1-12 Reco_SPI_Data 模块的基本结构框图

该模块的信号说明如下表 1-3 所示：

表 1-3 Reco_SPI_Data 模块信号说明表

信号名称	I/O	信号意义
Clk	I	模块时钟信号，100M
Rst_n	I	模块复位信号，低电平有效
Data_Valid	I	数据有效标志信号
Recive_Data [7:0]	I	接收的 8 位数据
Trans_Cnt[15:0]	I	CS↓到 CS↑SPI 传输的字节数
Trans_Start	I	传输开始信号
Trans_End	I	传输结束信号
Reco_Addr[14:0]	O	解析出的地址信号
Reco_Data [7:0]	O	解析出的数据信号
Reco_Cmd	O	解析出的命令信号，1 表示读，0 表示写
Reco_Valid	O	解析出的执行脉冲信号

先简单的说明一下协议，如下表 1-4 所示。

表 1-4 协议说明表

起始位	读写位(1 读 0 写)+首地址的高 7 位	首地址的低 8 位	数据 1	……	数据 n	终止位
CS 的下降沿	第 1 字节	第 2 字节	第 3 字节	……	第 n 字节	CS 的上升沿

我们需要将接收的数据 Recive_Data 按照上表所示的协议进行解码，最终将解码完成的数据进行输出，解码的过程我们可以通过状态机的方式实现，状态转移图如下图 1-13 所示。

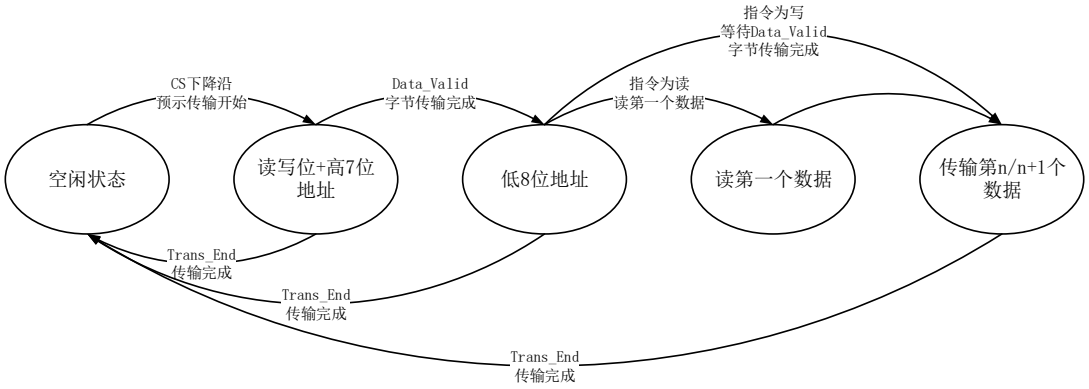

图 1-13 Reco_SPI_Data 模块状态转移图

根据上图所示，定义状态如下：

```
localparam S_IDLE    = 5'b00001;
localparam S_ADDR_H  = 5'b00010;
localparam S_ADDR_L  = 5'b00100;
localparam S_FR_DATA = 5'b01000;
localparam S_DATA    = 5'b10000;
```

下面将对每一个状态的的实现及其功能进行说明。

1. S_IDLE

空闲状态，当接收到传输开始信号 Trans_Start 之后，进入 S_ADDR_H 状态，否则一直处于 S_IDLE 状态，代码如下所示：

```
S_IDLE:
    begin
        if(Trans_Start)
            SM_STATE <= S_ADDR_H;
        else
            SM_STATE <= S_IDLE;
    end
```

2. S_ADDR_H

解析地址位和高 7 位地址位，当接收到数据有效信号 Data_Valid 并且 Trans_Cnt 等于 1 时，说明接收到了第一个字节的数据，此时接收的数据 Recive_Data 最高位对应着协议中的读/写位，将该位的数据给到 Reco_Cmd，并

将剩下的 7 位数据给到地址数据 Base_Addr 高七位；当接收到 Trans_End 状态，说明传输结束，进入到 S_IDLE 状态，代码如下所示：

```
S_ADDR_H:
    begin
        if(Data_Valid && (Trans_Cnt == 1)) begin
            Base_Addr[14:8] <= Recive_Data[6:0];
            Reco_Cmd <= Recive_Data[7];
            SM_STATE <= S_ADDR_L;
        end
        else if(Trans_End)
            SM_STATE <= S_IDLE;
        else
            SM_STATE <= S_ADDR_H;
    end
```

3. S_ADDR_L

解析地址低 8 位，进入该状态之后，当 Data_Valid 有效并且 Trans_Cnt 等于 2 的时候，说明此时接收到了第二个字节数据，将此时接收到的数据 Recive_Data 给到地址数据的低 8 位，此时当 Reco_Cmd 等于 1 的时候，代表读，进入到 S_FR_DATA 状态，否则为写，进入 S_DATA 状态，当接收到 Trans_End 信号，代表传输完成，进入到 S_IDLE 状态，代码如下所示：

```
S_ADDR_L:
    begin
        if(Data_Valid && (Trans_Cnt == 2)) begin
            Base_Addr[7:0] <= Recive_Data;
            if(Reco_Cmd)
                SM_STATE <= S_FR_DATA;
            else
                SM_STATE <= S_DATA;
        end
        else if(Trans_End)
            SM_STATE <= S_IDLE;
        else
            SM_STATE <= S_ADDR_L;
    end
```

4. S_FR_DATA

读取第一个字节的数据，SPI 读写操作在时间上是有区别的，写 RAM/REG 时，先接收 SPI 数据，然后将接收的数据写入 RAM/REG 中；读 RAM/REG 时，先将 RAM/REG 数据取出，然后将取出的数据以 SPI 发送，因此数据要在 SPI 传输前准备好，也就是说读操作要比写操作提前一个 SPI 字节的时间，所以在读的时候，需要一个 S_FR_DATA 状态，进入该状态之后，第一个读取的数据需要

提前准备，无需等待 Data_Valid 信号，然后进入 S_DATA 状态，将 Reco_Valid 拉高，代表数据有效，代码如下所示：

```
S_FR_DATA://第一个读取的数据需要提前准备，无需等待 Data_Valid 信号
    begin
        Reco_Addr <= Base_Addr;
        Reco_Data <= Recive_Data;
        Reco_Valid <= 1'b1;
        SM_STATE <= S_DATA;
    end
```

5. S_DATA

传输第 n/n+1 个数据，进入 S_DATA 状态之后，当数据有效时，根据 Reco_Cmd 的值，计算出需要操作的地址 Reco_Addr，在前面一个状态中我们说过，读操作要写操作提前一个 SPI 字节的时间，那也就是说在写的时候，当计数值达到 3 之后，将第三个字节的数据写入至基地址 Base_Addr 中，但是在读的时候，在计数值到 2 的时候，执行从 RAM/REG 中读出数据的操作（因为要提前准备数据），此时地址为基地址 Base_Addr，如下图 1-14 所示：

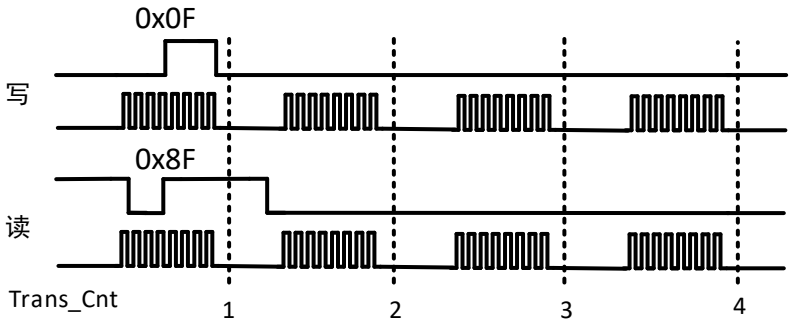

图 1-14 读写操作示意图

从上图可以看出，在执行写操作的时候，当计数值 Trans_Cnt 等于 3 的时候，执行将 SPI 传输过来的第三个数据写入 RAM/REG 的操作，地址为 0x0F00；执行读操作的时候，当计数值等于 2 的时候，执行从 RAM/REG 中读出数据的操作（因为要提前准备数据），此时地址为 0x0F00，从而可以得到：

写操作的地址 = 基地址 + 计数值 – 3；读操作的地址 = 基地址 + 计数值 – 2

在 S_DATA 状态的时候，当数据有效的时候，还需要将此时接收到的数据 Recive_Data 给到 Reco_Data 进行输出，并且将 Reco_Valid 信号拉高，代表数据有效，当得到 Trans_End 信号之后，代表传输完成，进入到 S_IDLE 状态，并将 Reco_Valid 信号拉低，S_DATA 的状态代码如下所示：

```
S_DATA:
    begin
```

```
        if(Data_Valid) begin
            if(Reco_Cmd)
                Reco_Addr <= Base_Addr + Trans_Cnt - 15'd2;
            else
                Reco_Addr <= Base_Addr + Trans_Cnt - 15'd3;

            Reco_Data <= Recive_Data;
            Reco_Valid <= 1'b1;
            SM_STATE <= S_DATA;
        end
        else if(Trans_End) begin
            SM_STATE <= S_IDLE;
            Reco_Valid <= 1'b0;
        end
        else begin
            SM_STATE <= S_DATA;
            Reco_Valid <= 1'b0;
        end
end
```

1.3.1.3 Process_CMD 模块设计

根据 Reco_SPI_Data 模块输出的解码地址+解码数据+读写命令+执行信号(脉冲)来执行相应的读写 RAM/REG 的操作，并根据地址以不同方式处理命令，15 位的地址位宽可操作的地址为 0x0000~0x7FFF，其中 0x0000~0x7EFF 为 RAM 地址，7F00~7FFF 共 256 个地址为 REG 地址，该模块的基本结构如下图 1-15 所示：

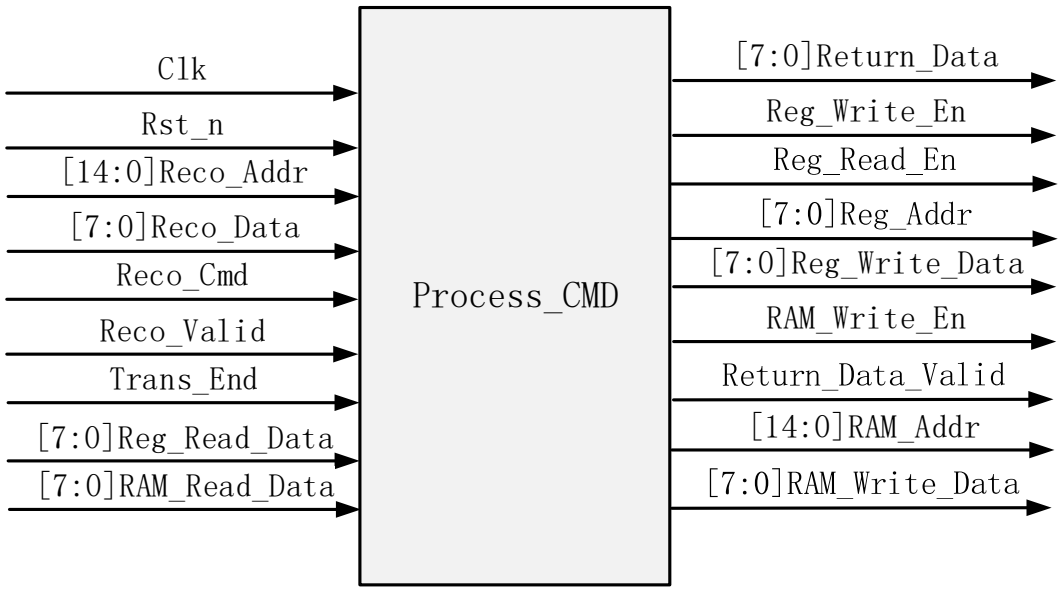

图 1-15 Process_CMD 模块基本结构框图

该模块的信号说明如下表 1-5 所示：

表 1-5 Process_CMD 模块信号说明表

信号名称	I/O	信号意义
Clk	I	模块时钟信号，100M
Rst_n	I	模块复位信号，低电平有效
Reco_Addr[14:0]	I	Reco_SPI_Data 模块输出的地址信号
Reco_Data [7:0]	I	Reco_SPI_Data 模块输出的数据信号
Reco_Cmd	I	Reco_SPI_Data 模块输出的命令信号，1 表示读，0 表示写
Reco_Valid	I	Reco_SPI_Data 模块输出的数据有效信号
Trans_End	I	传输完成标志信号
Reg_Read_Data [7:0]	I	从 REG 中读出的 8 位数据信号
RAM_Read_Data [7:0]	I	从 RAM 中读出的 8 位数据信号
Return_Data[7:0]	O	获取的返回的数据信号
Reg_Write_En	O	REG 的写使能信号
Reg_Read_En	O	REG 的读使能信号
Reg_Addr[7:0]	O	REG 的地址信号
Reg_Write_Data[7:0]	O	REG 的写数据信号
RAM_Write_En	O	RAM 的写使能信号
Return_Data_Valid	O	获取的返回的数据有效信号
RAM_Addr[14:0]	O	RAM 的地址信号
RAM_Write_Data[7:0]	O	RAM 的写数据信号

在 Process_CMD 模块中，首先根据地址以不同方式处理命令，当 Reco_Valid 有效，并且地址 Reco_Addr 小于 15'h7EFF 时，对 RAM 进行操作，此时若 Reco_Cmd 等于 1，从 RAM 对应的地址中读出数据，将 RAM 写使能 RAM_Write_En 拉低，RAM_En 信号拉高；若 Reco_Cmd 等于 0，将数据写入 RAM 对应的地址，将 RAM_Write_En 和 RAM_En 信号拉高；当地址 Reco_Addr 大于 15'h7EFF 时，此时若 Reco_Cmd 等于 1，从对应的寄存器读出数据，将 REG 的读使能信号 Reg_Read_En 拉高，否者将数据写入 RAM 对应的地址，将 REG 的写使能信号 Reg_Write_En 拉高，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n) begin
        Reg_Write_En <= 1'b0;
        Reg_Read_En <= 1'b0;
        RAM_Write_En <= 1'b0;
        RAM_En <= 1'b0;
    end
    else if(Reco_Valid)begin
        if(Reco_Addr <= 15'h7EFF) begin
            if(Reco_Cmd) begin  //从 RAM 对应的地址读出数据
                RAM_Write_En <= 1'b0;
```

```
                RAM_En <= 1'b1;
            end
            else begin          //将数据写入 RAM 对应的地址
                RAM_Write_En <= 1'b1;
                RAM_En <= 1'b1;
            end
        end
        else begin
            if(Reco_Cmd) begin  //从对应的寄存器读出数据
                Reg_Read_En <= 1'b1;
            end
            else begin          //将数据写入对应的寄存器
                Reg_Write_En <= 1'b1;
            end
        end
    end
    else begin
        Reg_Write_En <= 1'b0;
        Reg_Read_En <= 1'b0;
        RAM_Write_En <= 1'b0;
        RAM_En <= 1'b0;
    end
end
```

然后得到 RAM 和 REG 的读使能信号，当 RAM_En 和 RAM_Write_En 有效的时候，产生 RAM 的读使能信号，代码如下所示：

```
reg RAM_Read_En_R,Reg_Read_En_R;
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n) begin
        RAM_Read_En_R <= 1'b0;
        Reg_Read_En_R <= 1'b0;
    end
    else begin
        RAM_Read_En_R <= RAM_En & (~RAM_Write_En);
        Reg_Read_En_R <= Reg_Read_En;
    end
end
```

这里需要注意的是，当从 RAM 中读取数据时，数据出来是有延时的，具体延时几拍与你的 RAM IP 核配置有关，这将在后面讲到添加 RAM IP 核的时候进行说明，这里延迟了一拍。

最后当产生 REG 的读使能信号 Reg_Read_En_R 之后，将从 REG 中读出的数据 Reg_Read_Data 返回输出，并将 Return_Data_Valid 信号拉高，说明此时返回的数据有效；当产生 RAM 的读使能信号 RAM_Read_En_R 之后，将从 RAM

中读出的数据 RAM_Read_Data 返回输出，并将 Return_Data_Valid 信号拉高；当产生 Trans_End 信号之后，说明传输完成，将 Return_Data_Valid 信号拉低，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Return_Data <= 8'h00;
    else if(Reg_Read_En_R) begin
        Return_Data <= Reg_Read_Data;
        Return_Data_Valid <= 1'b1;
    end
    else if(RAM_Read_En_R) begin
        Return_Data <= RAM_Read_Data;
        Return_Data_Valid <= 1'b1;
    end
    else if(Trans_End) begin
        Return_Data <= 8'h00;
        Return_Data_Valid <= 1'b0;
    end
    else begin
        Return_Data_Valid <= 1'b0;
        Return_Data <= Return_Data;
    end
end
```

1.3.2 SPI_REG 模块设计

SPI 寄存器模块，该模块内部设计了两个端口 A 和 B，并在内部定义了 256 个 8 位的寄存器，该模块的基本结构框图如下图 1-16 所示。

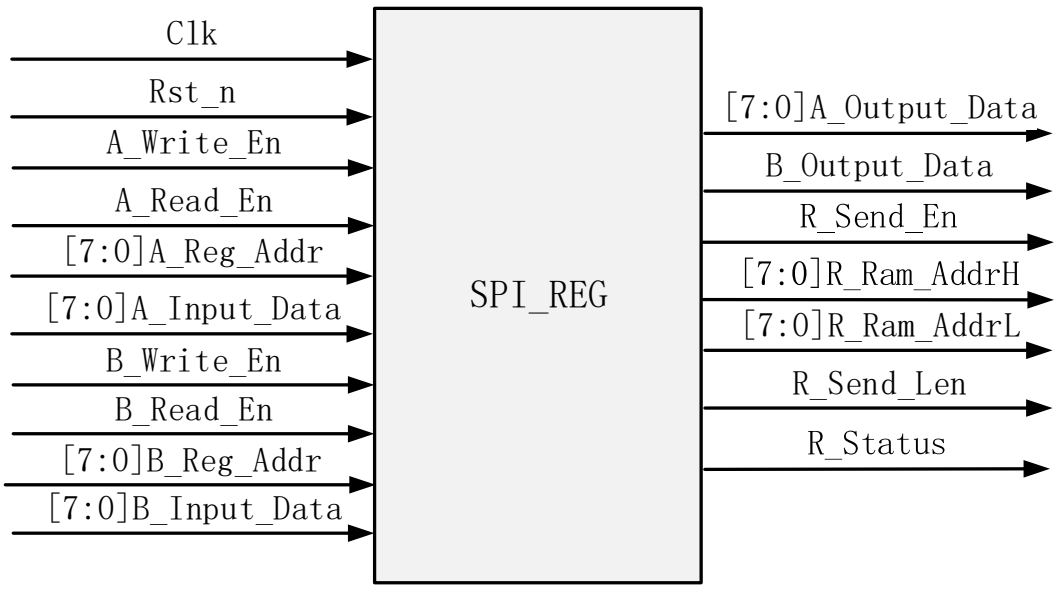

图 1-16 SPI_REG 模块的基本结构框图

对该模块的信号说明如下表 1-6 所示。

表 1-6 SPI_REG 模块信号说明表

类别	信号名称	I/O	信号意义
时钟及复位	Clk	I	模块时钟信号，100M
	Rst_n	I	模块复位信号，低电平有效
A 端口信号	A_Write_En	I	A 端口的写使能信号
	A_Read_En	I	A 端口的读使能信号
	A_Reg_Addr[7:0]	I	A 端口的地址信号
	A_Input_Data[7:0]	I	A 端口输入数据信号
	A_Output_Data[7:0]	O	A 端口输出数据信号
B 端口信号	B_Write_En	I	B 端口的写使能信号
	B_Read_En	I	B 端口的读使能信号
	B_Reg_Addr [7:0]	I	B 端口的地址信号
	B_Input_Data [7:0]	I	B 端口输入数据信号
	B_Output_Data[7:0]	O	B 端口输出数据信号
功能寄存器	R_Send_En	O	发送使能寄存器，地址为 0x00
	R_Ram_AddrH [7:0]	O	RAM 高 8 位地址寄存器，地址为 0x01
	R_Ram_AddrL[7:0]	O	RAM 低 8 位地址寄存器，地址为 0x02
	R_Send_Len	O	发送长度寄存器，地址为 0x03
	R_Status	O	状态寄存器，地址为 0x04

上表中功能寄存器的设定与我们想要实现的功能有关，用户可以根据自己的需求进行修改，下面将讲解 SPI_REG 模块的代码设计。

首先定义 256 个 8 位寄存器，将第一个寄存器定义为发送使能寄存器；第二个寄存器定义为 RAM 的高 8 位地址寄存器；第三个寄存器定义为 RAM 的低 8 位地址寄存器；第四个寄存器定义为发送长度寄存器；第五个寄存器定义为状态寄存器，代码如下所示：

```
//定义 256 个 8 位寄存器
reg [7:0]Reg_Data[255:0];
integer loop;
//功能寄存器
assign R_Send_En = Reg_Data[0];
assign R_Ram_AddrH = Reg_Data[1];
assign R_Ram_AddrL = Reg_Data[2];
assign R_Send_Len = Reg_Data[3];
assign R_Status = Reg_Data[4];
```

然后实现 A 端口的读功能，当检测到 A 端口的读使能信号 A_Read_En 之后，根据需要读取的寄存器地址，将对应寄存器的数据进行输出，代码如下所示：

```
//A 端口读
```

```
always @(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        A_Output_Data <= 8'h00;
    else if(A_Read_En)
        A_Output_Data <= Reg_Data[A_Reg_Addr];
    else
        A_Output_Data <= A_Output_Data;
end
```

B 端口的读功能的实现和 A 端口的读功能实现方法一样，当检测到 B 端口的读使能信号之后，根据需要读取的寄存器地址，将对应寄存器的数据进行输出，代码如下所示：

```
//B 端口读
always @(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        B_Output_Data <= 8'h00;
    else if(B_Read_En)
        B_Output_Data <= Reg_Data[B_Reg_Addr];
    else
        B_Output_Data <= B_Output_Data;
end
```

最后剩下 A、B 端口的写功能，当检测到复位信号之后，将所有的寄存器中存放的数据清零；当检测到 A 端口的写使能之后，根据需要写入的寄存器地址，将输入的数据写入进寄存器中；当检测到 B 端口的写使能之后，和 A 端口操作一样，将输入的数据根据需求写到对应的寄存器中；如果通过 A、B 两个端口操作同一个寄存器，寄存器中将会接收端口 A 输入的数据，代码如下所示：

```
//AB 端口写
always @(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        for (loop = 0;loop<256;loop=loop+1) begin
            Reg_Data[loop] <= 8'h00;
        end
    else if(A_Write_En)
        Reg_Data[A_Reg_Addr] <= A_Input_Data;
    else if(B_Write_En)
        Reg_Data[B_Reg_Addr] <= B_Input_Data;
    else begin
        if(A_Reg_Addr == B_Reg_Addr)
            Reg_Data[A_Reg_Addr] <= Reg_Data[A_Reg_Addr];
        else begin
            Reg_Data[A_Reg_Addr] <= Reg_Data[A_Reg_Addr];
```

```
            Reg_Data[B_Reg_Addr] <= Reg_Data[B_Reg_Addr];
        end
    end
end
```

1.3.3 添加 RAM IP

在本次实验中，我们运用了真双端口 RAM 存储数据，点击 IP Catalog 输入 RAM 然后选择 Block Memory Generator，进入界面之后选择 True Dual Port RAM，并勾选 Byte Write Enablie，由于 SPI 协议是按字节传输的，所以这里将 Byte Size 设置为 8，如下图 1-17 所示。

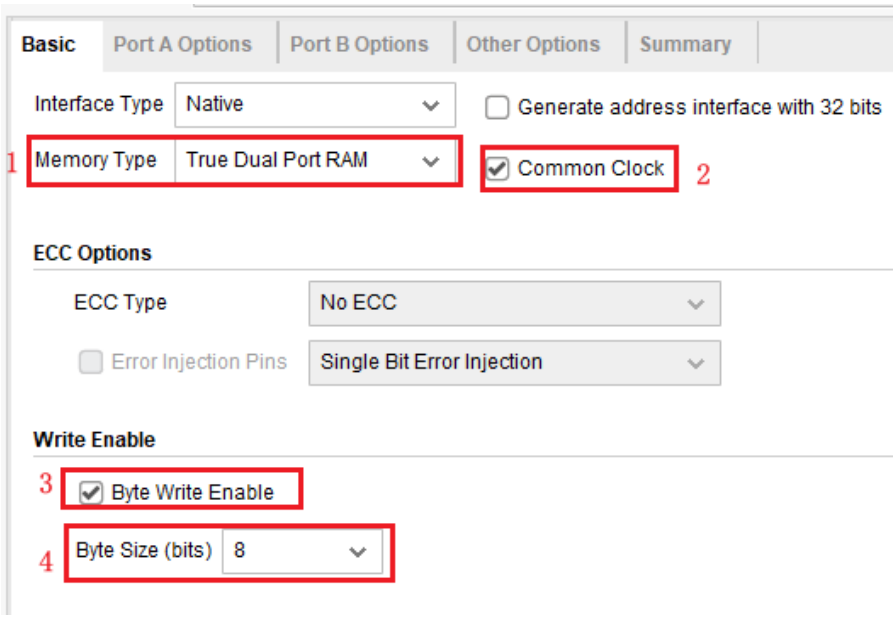

图 1-17 RAM 配置界面 1

然后配置端口 A 和端口 B，两个端口的配置方式一样，这里以端口 A 为例，为了和 SPI 传输方式保持一致，将读写位宽设置为 8 位，写深度设置为 32768，并一直使能端口，不勾选任何的输出寄存器，如下图 1-18 所示：

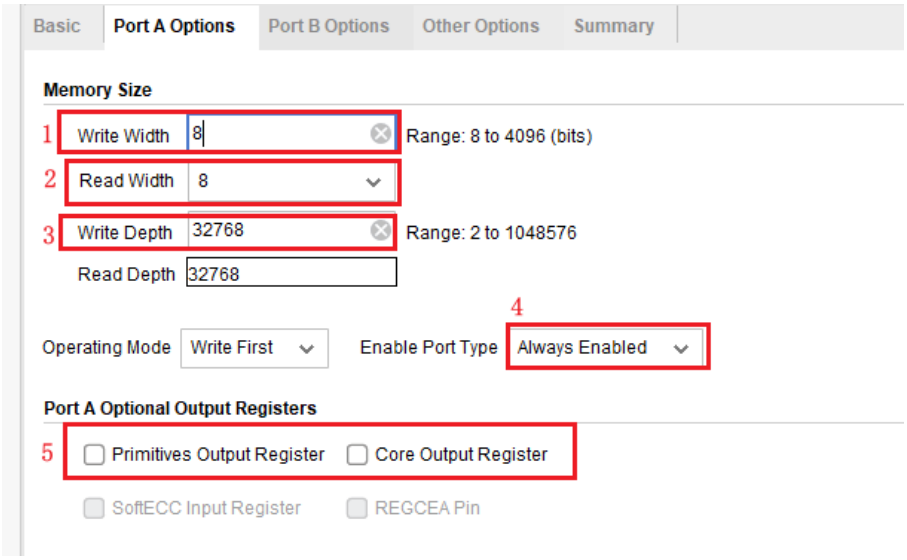

图 1-18 配置端口 A

在前面我们曾经说过，当从 RAM 中读取数据时，数据出来是有延时的，具体延时几拍与 RAM IP 核的配置有关，也就是上图中的 Port A Optional Output Registers 下面的勾选项有关，我们这里不勾选，将会有一拍的延时，勾选不同的选项，将会延迟不同的拍数，具体可以查看 Summary 界面下的 Total PortA Read Latency 的个数，如下图 1-19 所示：

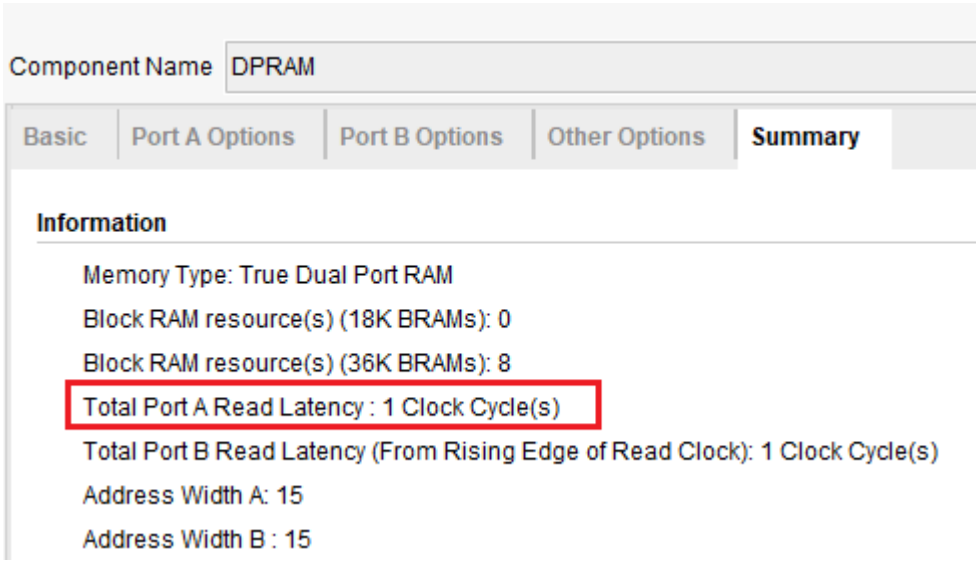

图 1-19 输出延迟查看

本章实验只是教大家如何添加 RAM IP 核，更详细的使用方式请查看前面“IP 核使用之 RAM”一节中的内容。

1.3.4 uart_peripheral 模块设计

串口设备模块（uart_peripheral）的功能主要根据寄存器中存储的数据内容去控制串口发送数据，串口需要发送从 RAM 中读出的 8 位数据，该模块内部还实现了初始化 RAM，清寄存器等操作。该模块内部主要包含 Get_Ram_Data 模块和 uart_byte_tx 模块，其基本结构如下图 1-20 所示：

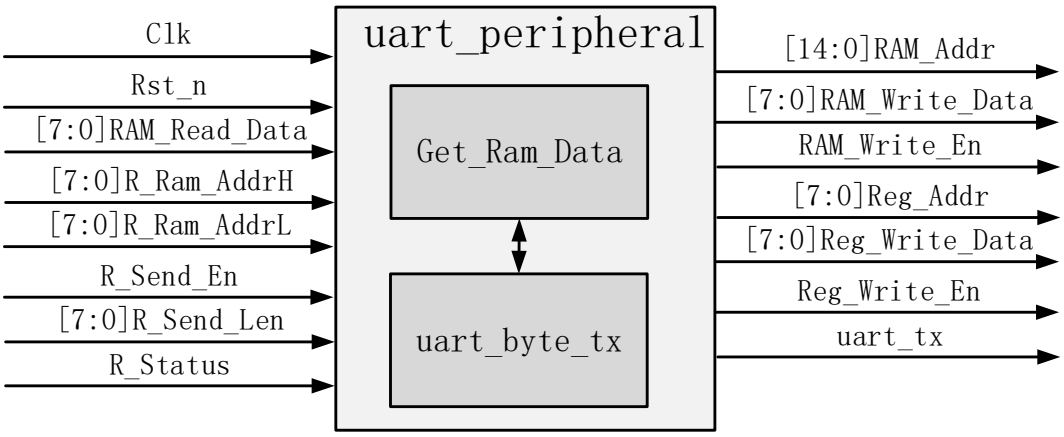

图 1-20 uart_peripheral 模块的基本结构框图

对该模块的信号说明如下表 1-7 所示。

表 1-7 uart_peripheral 模块信号说明表

信号名称	I/O	信号意义
Clk	I	模块时钟信号，100M
Rst_n	I	模块复位信号，低电平有效
RAM_Read_Data [7:0]	I	从 RAM 中读出的 8 位数据
R_Ram_AddrH [7:0]	I	RAM 高 8 位地址寄存器
R_Ram_AddrL[7:0]	I	RAM 低 8 位地址寄存器
R_Send_En	I	发送使能寄存器
R_Send_Len[7:0]	I	发送长度寄存器
R_Status	I	状态寄存器
RAM_Addr [14:0]	O	RAM 的地址信号
RAM_Write_Data [7:0]	O	RAM 的写数据信号
RAM_Write_En	O	RAM 的写使能信号
Reg_Addr[7:0]	O	REG 的地址信号
Reg_Write_Data[7:0]	O	REG 的写数据信号
Reg_Write_En	O	REG 的写使能信号
uart_tx	O	串口发送信号

在该模块内部，只是将两个模块进行例化然后将端口信号进行连接，所以这里将不进行代码显示了，串口发送模块（uart_byte_tx）在前面的实验中已经介绍过了，这里将不再进行说明，下面将介绍 Get_Ram_Data 模块的实现。

1.3.4.1 Get_Ram_Data 模块设计

Get_Ram_Data 模块根据 RAM 和 REG 中的数据，生成控制串口发送模块的接口信号，并具有初始化 RAM 和清寄存器的功能，该模块的基本结构框图如下图 1-21 所示：

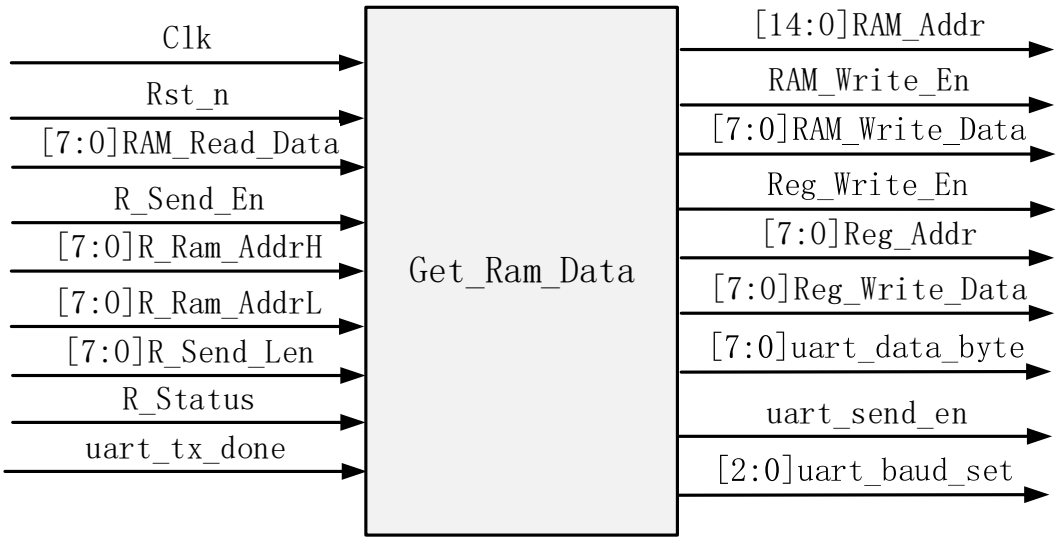

图 1-21 Get_Ram_Data 模块基本结构框图

该模块信号说明如下表 1-8 所示：

表 1-8 Get_Ram_Data 模块信号说明表

信号类型	信号名称	I/O	信号意义
时钟及复位	Clk	I	模块时钟信号，100M
	Rst_n	I	模块复位信号，低电平有效
RAM 端口	RAM_Read_Data [7:0]	I	RAM 的读数据信号
	RAM_Addr [14:0]	O	RAM 的地址信号
	RAM_Write_En	O	RAM 的写使能信号
	RAM_Write_Data[7:0]	O	RAM 的写数据信号
寄存器接口	Reg_Write_En	O	REG 的写使能信号
	Reg_Addr [7:0]	O	REG 的写地址信号
	Reg_Write_Data [7:0]	O	REG 的写数据信号
功能寄存器	R_Send_En	I	发送使能寄存器，地址为 0x00
	R_Ram_AddrH[7:0]	I	RAM 高 8 位地址寄存器，地址为 0x01
	R_Ram_AddrL[7:0]	I	RAM 低 8 位地址寄存器，地址为 0x02
	R_Send_Len [7:0]	I	发送长度寄存器，地址为 0x03
	R_Status	I	状态寄存器，地址为 0x04
串口发送模块接口	uart_tx_done	I	串口发送完成信号
	uart_data_byte[7:0]	O	串口需要发送的 8 字节数据
	uart_send_en	O	串口发送使能信号
	uart_baud_set [2:0]	O	串口波特率设置信号

首先我们初始化 RAM 中 0x0000~0x00FF 地址的数据为 0x00~0xFF，这样我们可以在 STM32 编程时，先读取 RAM 地址 0x0000~0x00FF 中的地址，看是否为 0x00~0xFF，验证 STM32 与 ACX720 之间通信是否正常，当初始化完成之后，将 Init_Data_Done 信号拉高，初始化 RAM 的代码如下所示：

```
reg [14:0]Init_Addr;
//初始化 RAM 中 0x0000~0x00FF 地址的数据为 0x00~0xFF
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n) begin
        RAM_Write_En <= 1'b0;
        Init_Addr <= 8'h00;
        RAM_Write_Data <= 8'h00;
        Init_Data_Done <= 1'b0;
    end
    else if(Init_Addr == 15'h00FF) begin
        Init_Data_Done <= 1'b1;
        RAM_Write_En <= 1'b0;
    end
    else begin
        Init_Addr <= Init_Addr + 1'b1;
        RAM_Write_Data <= RAM_Write_Data + 1'b1;
        RAM_Write_En <= 1'b1;
    end
end
```

然后清零发送使能寄存器，写状态寄存器。当检测到发送使能寄存器 R_Send_En 为 1 时，说明一轮传输开始了，这时需要将 R_Send_En 清 0，也就是拉高寄存器写使能信号，向寄存器地址 0 中写入 0，这样操作是为了方便进行下一次传输；当 R_Send_En 为 0 时，代表发送使能寄存器清零完成，取消寄存器写使能信号；当串口发送的数据个数等于需要传输的数据长度时，拉高寄存器写使能信号，向地址 0x04 中写入 1，也就是向状态寄存器中写入 1，代表数据传输完成，代码如下所示：

```
//清零 R_Send_En 寄存器，写状态寄存器
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n) begin
        Reg_Write_En <= 1'b0;
        Reg_Addr <= 8'h00;
        Reg_Write_Data <= 8'h00;
    end
    else if(R_Send_En) begin
        //将 R_Send_En 清 0
        Reg_Write_En <= 1'b1;
```

```
            Reg_Addr <= 8'h00;
            Reg_Write_Data <= 8'h00;
        end
        else if(R_Send_En == 1'b0) begin    //将 R_Send_En 清 0 之后取消写使能
            Reg_Write_En <= 1'b0;
        end
        else if(Send_Cnt == Send_Len) begin
            //将 R_Status 置 1
            Reg_Write_En <= 1'b1;
            Reg_Addr <= 8'h04;
            Reg_Write_Data <= 8'h01;
        end
    end
```

上述代码中的 Send_Len 信号就是需要传输的数据长度，当检测到 R_Send_En 为 1 时，记录本次需要传输的地址和长度，防止中途被改变，代码如下所示：

```
    //存储此次传输的地址与长度
    always@(posedge Clk or negedge Rst_n)
    begin
        if(!Rst_n) begin
            Send_Len <= 8'd0;
            Base_Addr <= 15'h00;
        end
        else if(R_Send_En) begin     //记录长度与地址
            Send_Len <= R_Send_Len;
            Base_Addr <= {R_Ram_AddrH[6:0],R_Ram_AddrL};
        end
        else begin
            Send_Len <= Send_Len;
            Base_Addr <= Base_Addr;
        end
    end
```

串口发送的数据个数 Send_Cnt 可以根据串口发送完成标志信号 uart_tx_done 进行计数，当检测到 uart_tx_done 信号为高电平时，说明串口单字节传输完成，Send_Cnt 计数加 1，当 Send_Cnt 的计数值与 Send_Len 的值一样时，清零 Send_Cnt，方便进行下一次的串口发送数据计数，代码如下所示：

```
    //Send_Cnt 计数
    always@(posedge Clk or negedge Rst_n)
    begin
        if(!Rst_n)
            Send_Cnt <= 8'd0;
        else if(Send_Cnt == Send_Len) begin
            Send_Cnt <= 8'd0;
        end
```

```
    else if(uart_tx_done) begin
        Send_Cnt <= Send_Cnt + 1'b1;
    end
end
```

当检测到 R_Send_En 的上升沿之后，产生串口发送使能信号，开始发送数据。首先我们得获取 R_Send_En 的上升沿，也就是将 R_Send_En 信号打两拍，得到 R_Send_En_R1 和 R_Send_En_R2，当 R_Send_En_R2 为低电平并且 R_Send_En_R1 为高电平的时候，此时就得到了 R_Send_En 的上升沿脉冲信号 R_Send_En_POS，如下图 1-22 所示：

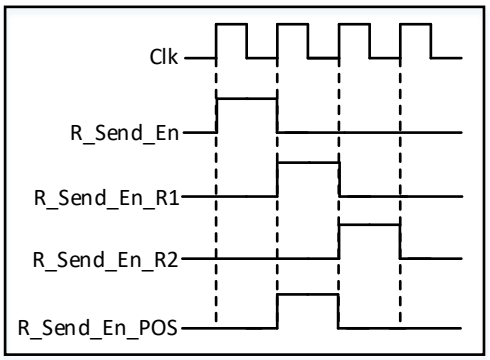

图 1-22 R_Send_En_POS 信号产生时序图

R_Send_En_POS 信号的实现代码如下所示：

```
wire R_Send_En_POS;
reg R_Send_En_R1,R_Send_En_R2;
//对 R_Send_En 打拍，取上升沿
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n) begin
        R_Send_En_R1 <= 1'b0;
        R_Send_En_R2 <= 1'b0;
    end
    else begin
        R_Send_En_R1 <= R_Send_En;
        R_Send_En_R2 <= R_Send_En_R1;
    end
end
assign R_Send_En_POS = (~R_Send_En_R2) & R_Send_En_R1;
```

当检测到 R_Send_En_POS 信号，将串口发送使能信号拉高，当串口发送数据计数 Send_Cnt 达到需要发送的数据个数的时候，将串口发送使能信号拉低，代码如下所示：

```
//在 R_Send_En 的上升沿之后连续发送数据
always@(posedge Clk or negedge Rst_n)
begin
```

```
    if(!Rst_n) begin
        uart_send_run <= 1'b0;
    end
    else if(R_Send_En_POS)
        uart_send_run <= 1'b1;
    else if(Send_Cnt == Send_Len - 1)
        uart_send_run <= 1'b0;
    else
        uart_send_run <= uart_send_run;
end
```

这里需要注意的是，RAM 在读取数据时有一拍的延时，所以串口发送时也得延迟一拍，也就是还需要将 uart_send_run 信号打一拍，这时得到的信号才是串口发送模块的发送使能信号，代码如下所示：

```
//uart_send_en 比 uart_send_run 慢一拍，是因为 Ram 读取有一拍的延时
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        uart_send_en <= 1'b0;
    else
        uart_send_en <= uart_send_run;
end
```

最后将串口的波特率信号设置为 3'd4，对应的波特率就是 115200；当 Init_Data_Done 为高电平时，说明初始化完成，此时 RAM 的地址为基地址 Base_Addr 加上传输数据的计数值 Send_Cnt，否则就为初始化地址，RAM 读出的数据交由串口发送模块的数据信号 uart_data_byte 进行输出，代码如下所示：

```
assign uart_baud_set = 3'd4;
assign RAM_Addr = Init_Data_Done ? (Base_Addr + Send_Cnt) : Init_Addr;
assign uart_data_byte = RAM_Read_Data;
```

通过上述描述，FPGA 内部各个模块的设计就完成了，接下来就需要通过在 STM32 上编程，实现对 FPGA 的读写操作。

1.4 MCU 编程

本次实验将使用 MDK 软件对 STM32 进行编程，首先，初始化 STM32 的串口，并初始化 STM32 的 SPI1 为主机模式、使能 SPI 外设，然后初始化发送数据的值，这里设置发送长度为 255，需要发送的数据就是 0~254，然后从 FPGA 的 RAM 中读取初始化的数据，并通过串口打印，再将需要写入的 0~254 写入 FPGA 的 RAM 中，并从 FPGA 的 RAM 中将数据读回，然后通过串口将数据打印，最后设置 FPGA 的串口从 RAM 取数据的地址与长度并启动 FPGA 的串口打

印，由此得到 main 函数的程序，代码如下所示：

```
int main(void)
{
    uint16_t i;
    uint8_t Read_Data[256],Write_Data[256];
    uart_init(115200);      //串口初始化为 115200
    SPI1_Master_Init(); //初始化 SPI1 为主机模式
    SPI_Cmd(SPI1, ENABLE); //使能 SPI 外设
    //初始化发送数据的值
    for(i=0;i<TEST_LENGTH;i++) {
        Write_Data[i] = i;
    }
    //从 FPGA 的 RAM 中读取初始化的数据
    Read_Data_From_RAM(Read_Data, 256, 0x00);
    //串口打印读取的数据
    for(i=0;i<256;i++) {
        printf("RAM_Data[0x%04X]:%d\n",i,Read_Data[i]);
    }
    //将数据写入 FPGA 的 RAM 中
    Write_Data_To_RAM(Write_Data, TEST_LENGTH, TEST_RAM_ADDRESS);
    //从 FPGA 的 RAM 中将数据读回
    Read_Data_From_RAM(Read_Data, TEST_LENGTH, TEST_RAM_ADDRESS);
    //串口打印读取的数据
    for(i=0;i<TEST_LENGTH;i++) {
      printf("RAM_Data[0x%04X]:%d\n",i+TEST_RAM_ADDRESS,Read_Data[i]);
    }
    //设置 FPGA 的串口从 RAM 取数据的地址与长度
    Set_UART_RAM_Conf(TEST_LENGTH, TEST_RAM_ADDRESS);
    //启动 FPGA 的串口打印
    Start_FPGA_UART_Send();
}
```

上述代码中的 TEST_RAM_ADDRESS 代表的 STM32 需要写入的 RAM 初始地址，这里定义为 0x0F00，TEST_LENGTH 为需要写入的长度，本次实验设定为 255，需要写入的数据为 0~254，下面对上述使用到的部分函数进行说明。

1.4.1 Read_Data_From_RAM 函数

Read_Data_From_RAM 函数的功能是将数据从 RAM 中读出，首先需要将片选信号拉低，开始一轮数据传输，然后配置起始地址和模式，也就是通过 SPI 协议写入需要读取 RAM 地址和指令，首先需要传输 RAM 地址的高 7 位和读指令，然后传输 RAM 地址的低 7 位地址，最后根据需要读取的数据长度，读取 RAM 中的数据，数据读完之后，将 CS 拉高，代表一轮传输完成，由此，得到 Read_Data_From_RAM 函数代码如下所示：

```
/** 将数据从 RAM 读出 **/
void Read_Data_From_RAM(uint8_t *Read_Data,uint16_t Length,uint16_t Address)
{
    uint16_t i;

    SPI1_CS(0);
     //配置起始地址和模式
    SPI_ReadWriteByte(SPI1,(Address >> 8) | 0x80);
    SPI_ReadWriteByte(SPI1,Address & 0xFF);
    //连续读数据
    for(i=0;i<Length;i++) {
        Read_Data[i] = SPI_ReadWriteByte(SPI1,0x00);//启动传输
    }
    SPI1_CS(1);
}
```

在配置地址和模式时，需要注意，函数中的变量 Address 代表的是：15 位地址（0x0000 到 0x7EFF 为 RAM 地址，0x7F00 到 0x7FFF 为寄存器地址）+读指令（最高位置 1 为读），上述代码中的 SPI_ReadWriteByte 函数代表通过 SPI 读写一个字节，代码如下所示：

```
u8 SPI_ReadWriteByte(SPI_TypeDef* SPIx,u8 TxData)
{
    u8 retry=0;
    //检查指定的 SPI 标志位设置与否:发送缓存空标志位
    while (SPI_I2S_GetFlagStatus(SPIx, SPI_I2S_FLAG_TXE) == RESET)
    {
        retry++;
        if(retry>200)return 0;
    }
    SPI_I2S_SendData(SPIx, TxData); //通过外设 SPIx 发送一个数据
    retry=0;
    //检查指定的 SPI 标志位设置与否:接受缓存非空标志位
    while (SPI_I2S_GetFlagStatus(SPIx, SPI_I2S_FLAG_RXNE) == RESET)
    {
        retry++;
        if(retry>200)return 0;
    }
    return SPI_I2S_ReceiveData(SPIx); //返回通过 SPIx 最近接收的数据
}
```

1.4.2 Write_Data_To_RAM

Write_Data_To_RAM 函数的功能是将数据写入到 RAM 中，与 Read_Data_From_RAM 函数的实现方式大致一样，唯一不同的就是 Address 的最高位应该为 0，也就是代表值写指令，该函数的实现代码如下所示：

```
/** 将数据写入 RAM **/
void Write_Data_To_RAM(uint8_t *Write_Data,uint16_t Length,uint16_t
Address)
{
    uint16_t i;

    SPI1_CS(0);
    //配置起始地址和模式（15 位地址）+写命令（最高位置 0 为写）
    SPI_ReadWriteByte(SPI1,(Address >> 8));
    SPI_ReadWriteByte(SPI1,Address & 0xFF);
    //连续写数据
    for(i=0;i<Length;i++) {
        SPI_ReadWriteByte(SPI1,Write_Data[i]);//启动传输
    }
    SPI1_CS(1);
}
```

1.4.3 Set_UART_RAM_Conf

Set_UART_RAM_Conf函数的功能是设置FPGA的串口发送的地址和长度。在设计 Process_CMD 模块的时候，定义其中 0x0000~0x7EFF 为 RAM 地址，0x7F00~0x7FFF 为寄存器地址，所以代码中寄存器的基地址 REG_BASE_ADDRESS 为 0x7F00，在“SPI_REG 模块设计”一节中，我们一共设计了 4 个寄存器，其说明如下表 1-9 所示。

表 1-9 寄存器说明表

寄存器地址	寄存器功能
0x7F00	串口发送使能寄存器
0x7F01	RAM 起始数据地址高字节寄存器
0x7F02	RAM 起始数据地址低字节寄存器
0x7F03	串口发送的数据字节长度寄存器

根据上表所示内容，在 Set_UART_RAM_Conf 函数内部首先拉低 CS 启动一轮传输，然后通过 SPI_ReadWriteByte 函数传输数据 0x7F01，然后依次传输需要发送的数据地址 Address 及长度，也就是依次设置了 RAM 起始数据地址高字节寄存器、RAM 起始数据地址低字节寄存器以及串口发送的数据字节长度寄存器的值，传输完成之后，拉高 CS，函数代码如下所示：

```
void Set_UART_RAM_Conf(uint16_t Length,uint16_t Address)
{
    //使用 FPGA 的串口发送从 RAM 的地址 0x0F10 开始的连续 255 个数据
    SPI1_CS(0);

    //写起始地址，0x0000 到 0x7EFF 为 RAM 地址，0x7F00 到 0x7FFF 为寄存器地址
    SPI_ReadWriteByte(SPI1,REG_BASE_ADDRESS >> 8);
```

```
    SPI_ReadWriteByte(SPI1,(REG_BASE_ADDRESS + 1) & 0xFF);

    SPI_ReadWriteByte(SPI1,(Address >> 8));
    SPI_ReadWriteByte(SPI1,Address & 0xFF);
    SPI_ReadWriteByte(SPI1,Length);

    SPI1_CS(1);
}
```

1.4.4 Start_FPGA_UART_Send

Start_FPGA_UART_Send 函数的功能是启动 FPGA 的串口进行数据打印。首先，通过 SPI_ReadWriteByte 函数传输串口发送使能寄存器的地址 0x7F00，然后向寄存器中写入 0x01，代表启动一轮传输，代码如下所示：

```
void Start_FPGA_UART_Send()
{
    SPI1_CS(0);
    SPI_ReadWriteByte(SPI1,(REG_BASE_ADDRESS >> 8));
    SPI_ReadWriteByte(SPI1,REG_BASE_ADDRESS & 0xFF);

    SPI_ReadWriteByte(SPI1,0x01);
    SPI1_CS(1);
}
```

综上所述，我们对部分函数进行了说明，关于串口初始化和 SPI 初始化的内容，请用户自行学习，这里将不再进行说明。

经过以上工作，代码设计部分的任务已经全部完成，接下来将进行板级验证。这里需要特别注意的就是 SPI 工作模式的设定，FPGA 侧的 SPI 工作模式必须和 STM32 的 SPI 工作模式以及数据传输位序都保持一致，比如都使用模式 0，也就是 CPOL=0，CPHA=0，FPGA 侧模式的设置对应的就是 SPI2CMD 模块的 CPOL 和 CPHA 这两个参数，数据传输位序与 SPI2CMD 模块的 BITS_ORDER 参数有关，1 为高位在前，0 为低位在前，这里我们将工作模式设置为模式 0，数据传输位序设置为 1，也就是高位在前，由此得到 SPI2CMD 模块例化时的代码如下所示：

```
SPI2CMD
#(
    .CPOL(1'D0),
    .CPHA(1'D0),
    .BITS_ORDER(1'D1)
 )SPI2CMD(
    .Clk(clk_100M),
    .RAM_Addr(RAM_Addr_A),
```

```
    .RAM_Read_Data(RAM_Read_Data_A),
    .RAM_Write_Data(RAM_Write_Data_A),
    .RAM_Write_En(RAM_Write_En_A),
    .Reg_Addr(Reg_Addr_A),
    .Reg_Read_Data(Reg_Read_Data_A),
    .Reg_Read_En(Reg_Read_En_A),
    .Reg_Write_Data(Reg_Write_Data_A),
    .Reg_Write_En(Reg_Write_En_A),
    .Rst_n(Rst_n),
    .SPI_CS(SPI_CS),
    .SPI_MISO(SPI_MISO),
    .SPI_MOSI(SPI_MOSI),
    .SPI_SCK(SPI_SCK)
);
```

STM32 侧 SPI 模式以及数据位序的设置就是修改 SPI 初始化函数 SPI1_Master_Init 中的 SPI_CPOL 参数、SPI_CPHA 参数和 SPI_FirstBit 参数，比如这里和 FPGA 侧保持一致，使用模式 0 和高位在前，代码如下所示：

```
SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low; //时钟悬空高
SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge; //数据捕获于第二个时钟沿
SPI_InitStructure.SPI_FirstBit=SPI_FirstBit_MSB;//数据传输从 MSB 位开始
```

读者可以根据如下表 1-10 所示的内容，进行模式和位序的设置。

表 1-10 模式和位序的设置表

SPI 工作模式及位序	FPGA 侧设置（SPI2CMD 模块）	STM32 侧设置（SPI1_Master_Init 函数）
工作模式 0，高位在前	CPOL=0 CPHA=0 BITS_ORDER=1	SPI_CPOL=SPI_CPOL_Low SPI_CPHA=SPI_CPHA_1Edge SPI_FirstBit=SPI_FirstBit_MSB
工作模式 0，低位在前	CPOL=0 CPHA=0; BITS_ORDER=0	SPI_CPOL=SPI_CPOL_Low SPI_CPHA=SPI_CPHA_1Edge SPI_FirstBit= SPI_FirstBit_LSB
工作模式 1，高位在前	CPOL=0 CPHA=1; BITS_ORDER=1	SPI_CPOL=SPI_CPOL_Low SPI_CPHA= SPI_CPHA_2Edge SPI_FirstBit=SPI_FirstBit_MSB
工作模式 1，低位在前	CPOL=0 CPHA=1; BITS_ORDER=0	SPI_CPOL= SPI_CPOL_Low SPI_CPHA= SPI_CPHA_2Edge SPI_FirstBit= SPI_FirstBit_LSB
工作模式 2，高位在前	CPOL=1 CPHA=0; BITS_ORDER=1	SPI_CPOL= SPI_CPOL_High SPI_CPHA=SPI_CPHA_1Edge SPI_FirstBit=SPI_FirstBit_MSB
工作模式 2，低位在前	CPOL=1 CPHA=0; BITS_ORDER=0	SPI_CPOL= SPI_CPOL_High SPI_CPHA=SPI_CPHA_1Edge SPI_FirstBit= SPI_FirstBit_LSB
工作模式 3，高位在前	CPOL=1 CPHA=1; BITS_ORDER=1	SPI_CPOL= SPI_CPOL_High SPI_CPHA=SPI_CPHA_2Edge SPI_FirstBit=SPI_FirstBit_MSB
工作模式 3，低位在前	CPOL=1 CPHA=1;	SPI_CPOL= SPI_CPOL_High SPI_CPHA=SPI_CPHA_2Edge

	BITS_ORDER=0	SPI_FirstBit= SPI_FirstBit_LSB

1.5 板级验证

本次实验的板级验证环节，主要验证：通过 STM32 将数据写入到 FPGA 芯片内部的 RAM 中，然后再将 RAM 中的数据回读，最终将写入和读出的数据通过串口打印出来，当数据一致的时候，就说明实验成功，STM32 成功实现了读写 FPGA 的功能。

1.5.1 引脚分配

本次实验的引脚分配表如下表 1-11 所示：

表 1-11 引脚分配表

Pin Name	Signial Name	Pin NO
FPGA_GCLK1	clk	Y18
S0	reset_n	F15
FPGA_UART_TX	uart_tx	M15
GPIO2_6	SPI_CS	N17
GPIO2_8	SPI_SCK	T18
GPIO2_11	SPI_MISO	P17
GPIO2_12	SPI_MOSI	R16

按照上述表中的内容进行引脚分配， SPI 的引脚分配与 STM32 和 ACX720 开发板的连接有关，本次实验我们连接在开发板右边 LCD 屏的连接口上，也就是 GPIO2 上，用户可以根据自己的需求进行修改。

上述工作完成后，接下来对我们的设计生成比特流，然后便可以开始连接硬件，准备烧录比特流到 ACX720 开发板上进行板级验证了

1.5.2 系统所需硬件

1. STM32 最小系统板
2. ACX720 开发板
3. xilinx 下载器
4. 串口线
5. 电源线
6. DAP Link

1.5.3 硬件连接

将xilinx 下载器、电源线、STM32 最小系统板依次连接在 ACX720 开发板上，然后将串口线一端连接 ACX720 开发板，一端连接电脑；将 DAP Link 一端连接 STM32 开发板，一端连接电脑；将 DAP Link 的 RXD 与 STM32 串口的 TXD 相连（实验中用红色杜邦线连接），在 uart_init 函数中可以看到，本次实验对应的 STM32 的 GPIO_Pin_9，整体硬件连接如下图 1-23 所示。

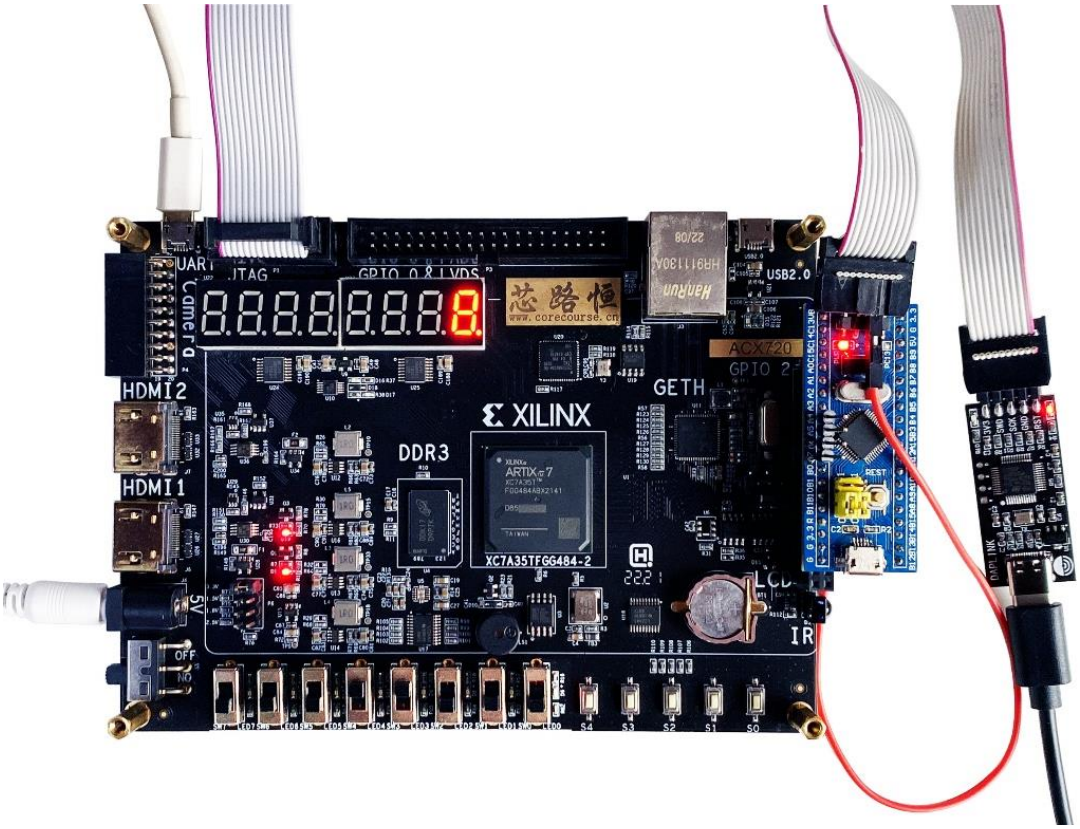

图 1-23 硬件连接图

1.5.4 下载 bit 文件并观察现象

我们首先给 ACX720 开发板上电，然后 FPGA 侧生成的 bit 文件下载至开发板中，如下图 1-24 所示。

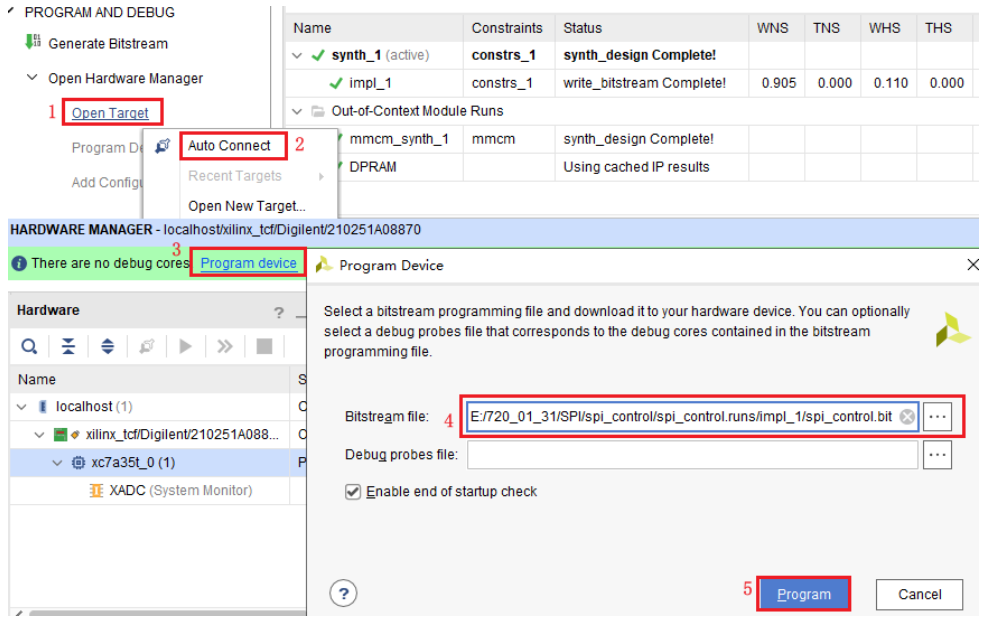

图 1-24 下载 FPGA 侧 bit 文件

然后打开两个名叫串口猎人的串口工具（该工具软件在开发板资料的 05_常用软件中有提供，用户也可自行选择其他串口软件），一个用来显示 FPGA 侧串口打印的数据，一个用来显示 STM32 侧串口打印的数据，配置如下图 1-25 和图 1-26 所示。

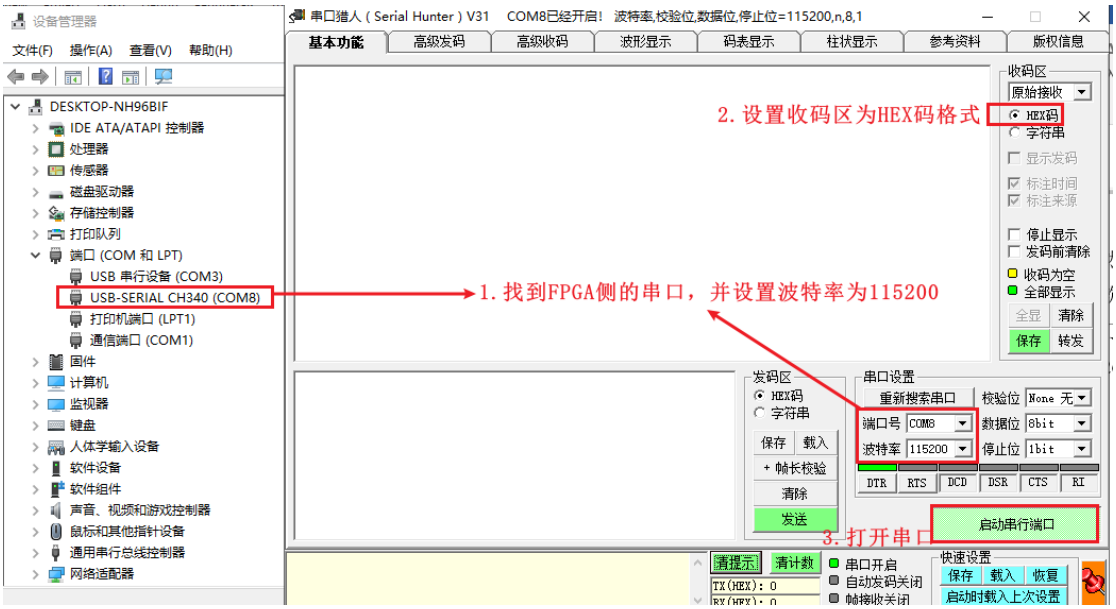

图 1-25 FPGA 侧串口助手配置界面

图 1-26 STM32 侧串口助手配置界面

编译 MDK 工程，下载程序，如下图 1-27 所示：

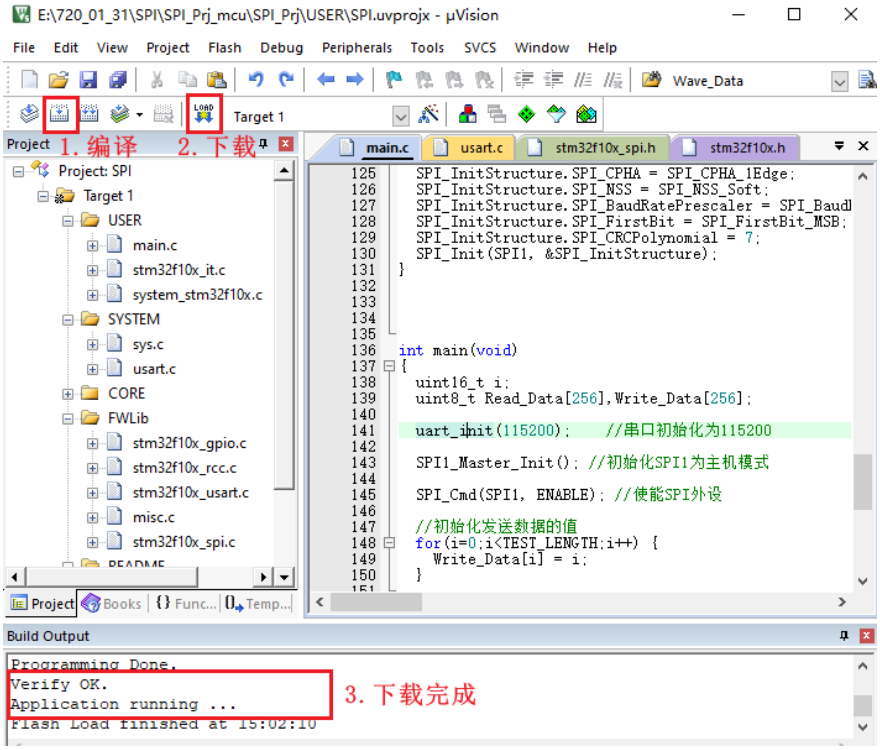

图 1-27 下载 STM32 侧程序

此时我们观察到 STM32 串口打印的数据如下图 1-28 所示。

图 1-28 STM32 侧串口打印的数据显示

点击全显之后，翻到最开始打印的数据，显示的是 RAM_Data[0x0000]~RAM_Data[0x00FF]中存储的数据，也就是 FPGA 侧初始化写入的数据，如下图 1-29 所示，可以看出从 RAM 中读出的初始化数据为 0~255，与我们设定一致，代表 STM32 和 FPGA 通信成功。

图 1-29 串口打印 RAM 初始化的数据

然后打印的是 RAM_Data[0x0F00]~ RAM_Data[0x0FFE]的数据，按照代码中的设定应该是 0~254 一共 255 个数据，串口打印的数据如下图 1-30 所示，从图中可以看出串口打印的数据与代码中设定一致。

图 1-30 串口打印从 RAM 回读的 STM32 写入的数据

最后查看 FPGA 侧串口打印，如下图 1-31 所示。

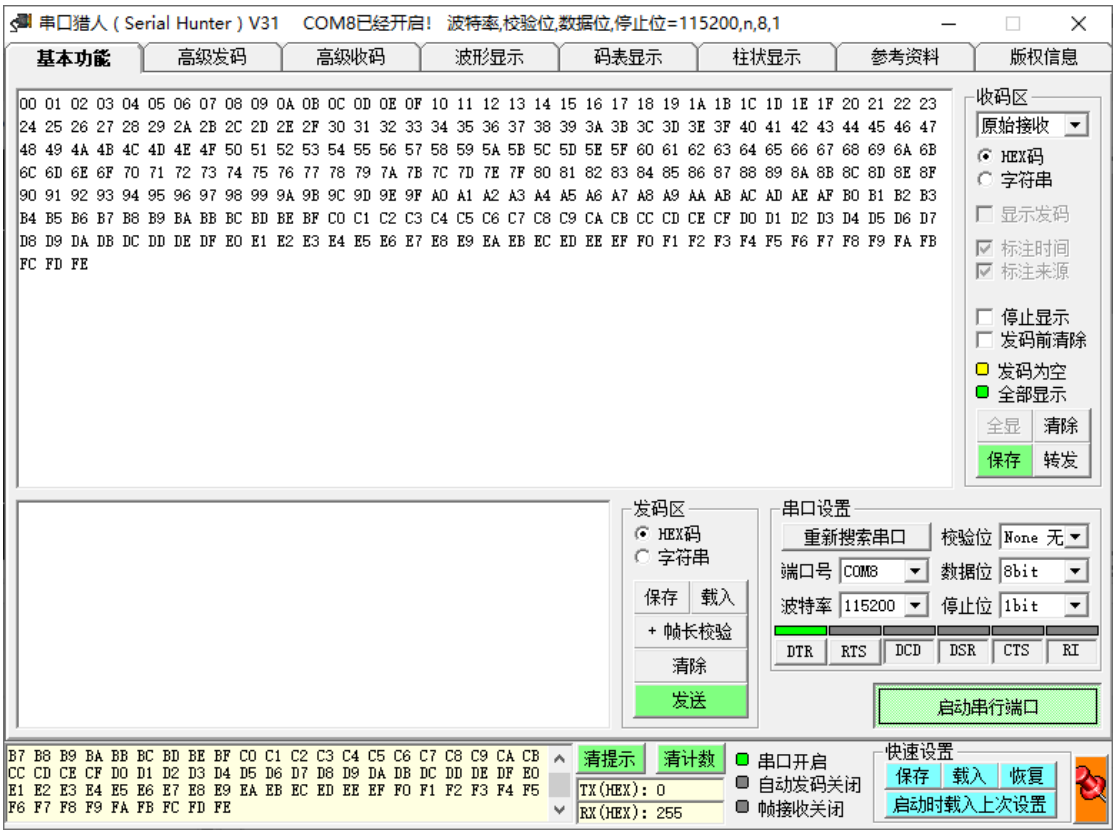

图 1-31FPGA 侧串口打印数据

从上述图中可以看出，FPGA 发送的数据为 0~FE（255），数据个数为 255 个，与 STM32 侧写入的数据一致，说明本次实验成功完成了 STM32 读写 FPGA 的功能。

1.6 总结

本次实验通过 STM32 实现了对 FPGA 内部 RAM 的读写功能，并通过向 FPGA 内部 REG 写值的方式，实现控制 FPGA 侧串口发送的功能。读者可以修改 SPI_REG 模块和 uart_peripheral 模块的内容去实现自己想要的功能。

实验中需要注意的是，FPGA 侧 SPI 的工作模式及数据传输位序一定要与 STM32 侧保持一致，否者会出现数据传输错误的情况，读者可以根据表 1-10 中的内容进行设置。