1 CM33 处理器 AHB 总线转 AXI4 接口的 DDR 控制器开发与验证

章节导读

随着嵌入式系统对高性能存储访问需求的不断增长，将传统总线架构与现代高速接口进行适配已成为系统设计的关键挑战。本文针对 CM33 处理器（ARM Cortex-M33 内核）的 AHB 总线架构，通过 AHB 转 AXI IP，实现对 DDR 存储器的高速读写控制。

1.1 系统整体设计

外部 25MHz 时钟（CIB_CLK）输入至 FPGA 内部的 PLL_SYS 模块，经锁相环生成多路时钟（包括 50MHz 的 sys_clk、100MHz 的 usr_clk 以及 400MHz 的 phy_clk）。FPGA 核心包含 MCU 控制器（STAR_Processor）通过 AHB 总线连接至 AHB-to-AXI 桥接器，再经 AXI 总线与 DDR 控制器通信，最终控制外部 DDR 存储器。系统通过混合总线（AHB/AXI）和分级时钟设计，实现了从处理器控制（100MHz）到高速存储访问（400MHz）的完整数据通路。系统的整体结构如下所示。

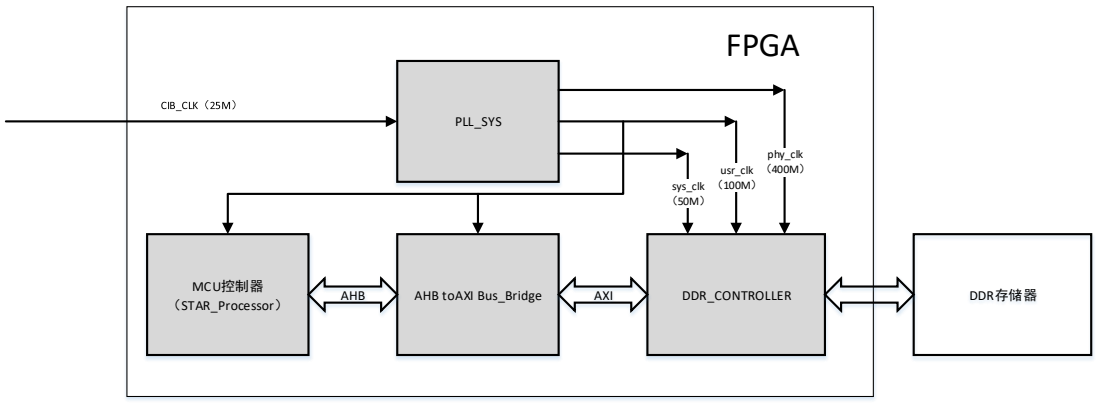


图 1-1 系统的整体结构

1.2 MCU 控制器

STAR IP 核是一种本系列 FPGA 的微控制器（MCU）的模块，用于图形化的端口定义，有利于设计者快速开发，兼容 Cortex-M33。在 IP Creator 中的嵌入式微处理器与控制器中找到 MCU 控制器，双击进行创建。

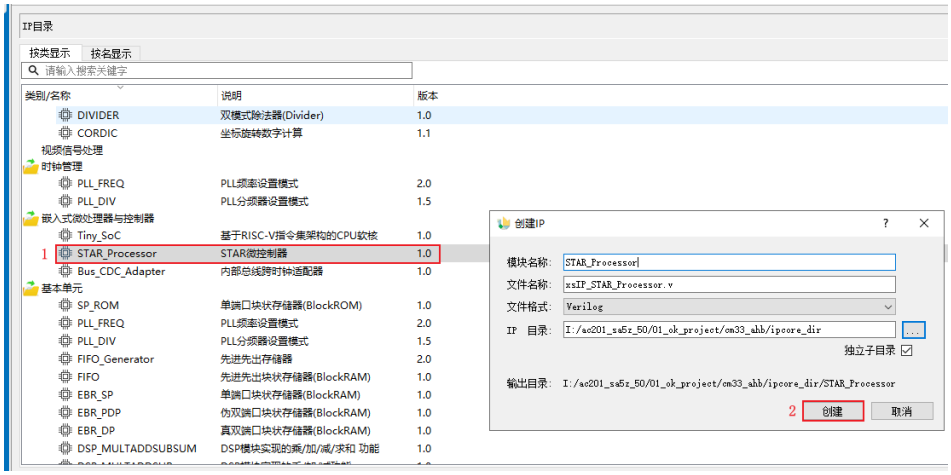


图 1-2 创建 MCU 控制器

创建完成之后，可以看到 IP 配置界面如下所示，从左到右一次是页标签和控制窗口。页标签分别是 STAR 架构设计、时钟、复位、GPIO、UART、ADC、SPI、I2C、中断、调试接口、AHB 主设备、AHB 从设备、DMA、STAR 核。按键包括加载、文档、确认、取消。加载用于加 IP 文件，文档是 IP 的数据手册，确认用于生成 IP 文件，取消用于关闭应用。

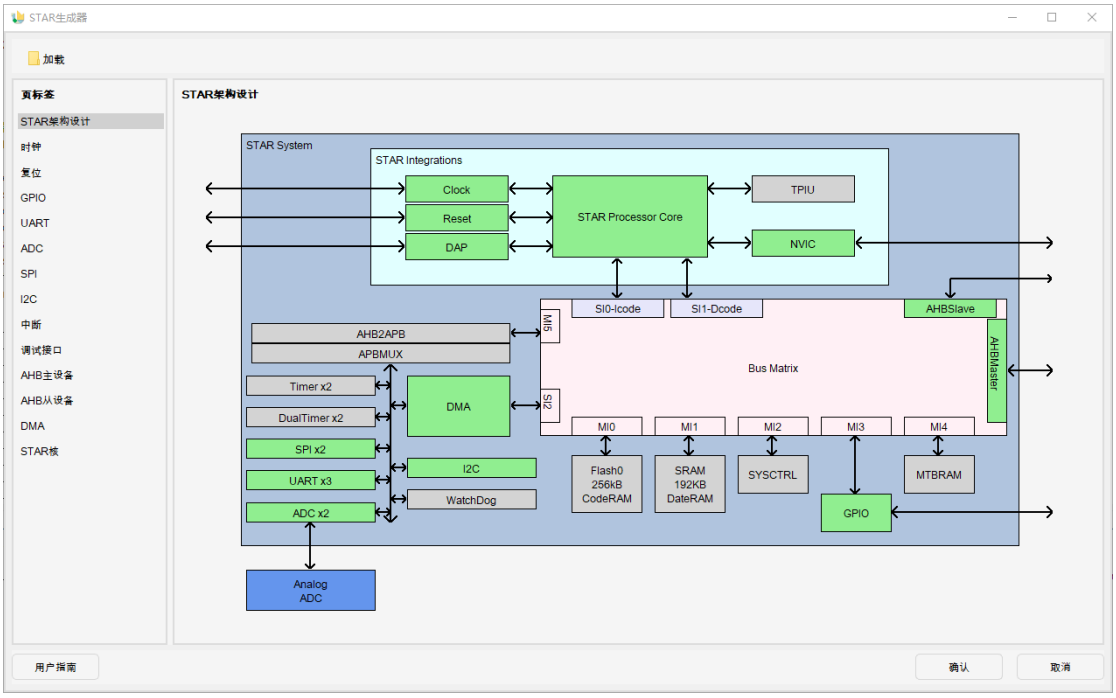


图 1-3 STAR IP 核配置界面

1.2.1 时钟

首先可以看到第一项就是系统时钟频率，本次实验我们设置为 100M，系统时钟源包括 CIB_CLK，CLK_TREE，输入的时钟只能选择其中一个。CIB_CLK

时钟来源于 FPGA 时钟树。CLK_TREE 时钟来源于 FPGA 内部 CIB 绕线资源。PCLK_DIV 是 PCLK 与 HCLK 的分频比，0:fPclk=1/1fHCLK，1:fPclk=1/2 fHCLK，2:fPclk=1/3fHCLK，4:fPclk=1/5fHCLK，5:fPclk=1/6fHCLK，6:fPclk=1/7fHCLK，7:fPclk=1/8。

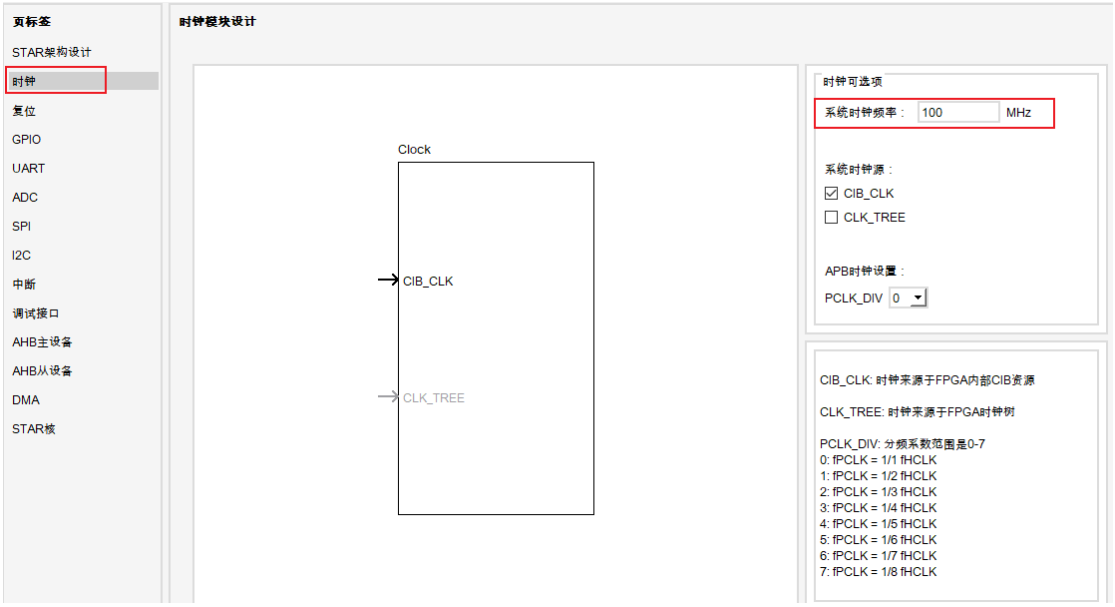


图 1-4 时钟设置界面

SEAL STAR 的时钟结构如下所示。

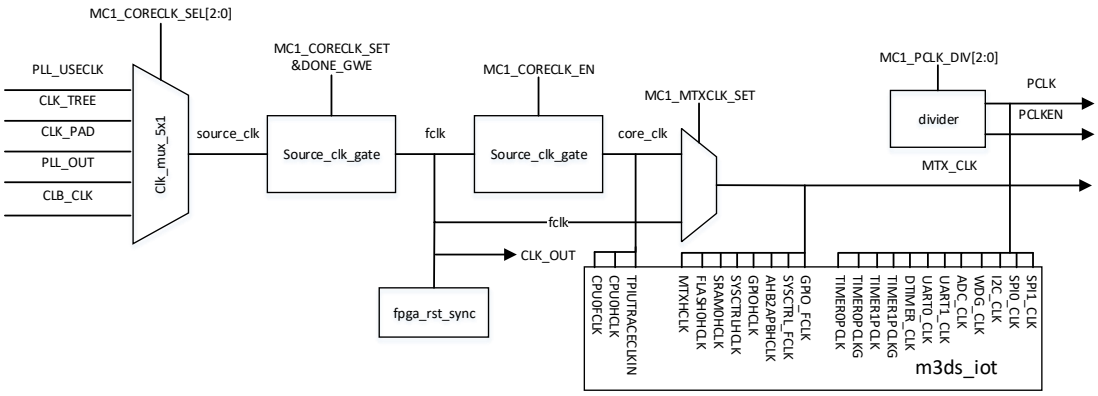


图 1-5 时钟结构图

对上述图中的部分信号说明如下所示。

表 1-1 时钟结构图部分信号说明表

NAME	Type	Width	Description
PLL_USRCLK	input	1	clock from OSC divider
CLK_TREE	input	1	clock from FPGA clock tree
CLK_PAD	input	1	clock from PAD(connect PAD PT22A)
PLL_OUT	input	1	clock from PLL output
CIB_CLK	input	1	clock from CIB block

DONE_GWE	input	1	source clock enable from config
CLKOUT	output	1	PLL generated clock
MTX_CLK	output	1	AHB Matrix clock output
PCLK	output	1	APB clock output
PCLKEN	output	1	APB clock output enable
MC1_CORE_SET (CORE_SET)	input	1	source clock enable from fuse: FALSE : Disable the STAR block TRUE : Enable the STAR block
MC1_CORECLK_SEL (CORECLK)	input	3	clock selection from fuse: 3'b000(PLL_USRCLK): choose PLL_USRCLK; 3'b001(CLK_TREE) : choose CLK_TREE; 3'b011(CLK_PAD): choose CLK from PAD; 3'b101(PLL_OUT): choose PLL_OUT 3'b111(CIB_CLK): choose CIB_CLK
MC1_CORECLK_EN (CORECLK_EN)	input	1	STAR Core clock enable from fuse: FALSE : Disable the STAR core clock TRUE : Enable the STAR core clock
MC1_MTXCLK_SET (MTXCLK)	input	1	matrix clock selection from fuse : 1(SOURCECLK): source_clk 0(CORECLK): core_clk
MC1_PCLK_DIV (PCLK_DIV)	input	3	the divide ratio ofAHB2APB: 0: f PCLK = f HCLK 1: f PCLK = 1/2 f HCLK 2: f PCLK = 1/3 f HCLK 3: f PCLK = 1/4 f HCLK 4: f PCLK = 1/5 f HCLK 5: f PCLK = 1/6 f HCLK 6: f PCLK = 1/7 f HCLK 7: f PCLK = 1/8 f HCLK

1.2.2 复位

复位配置界面如下所示。可以看到复位配置选项包括 CPU_RST，MTX_RSTN。CPU_RSTN 用于处理器内核复位，低电平有效。MTX_RSTN 用于 AHB Matrix 上外设的复位，低电平有效。这里我们暂时就勾选 MTX_RSTN。

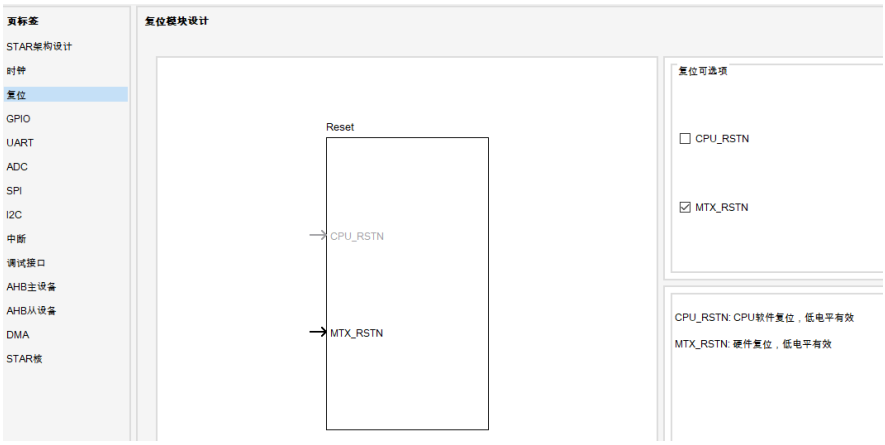


图 1-6 复位设置界面

1.2.3 GPIO

通用输入输出接口配置界面如下所示。

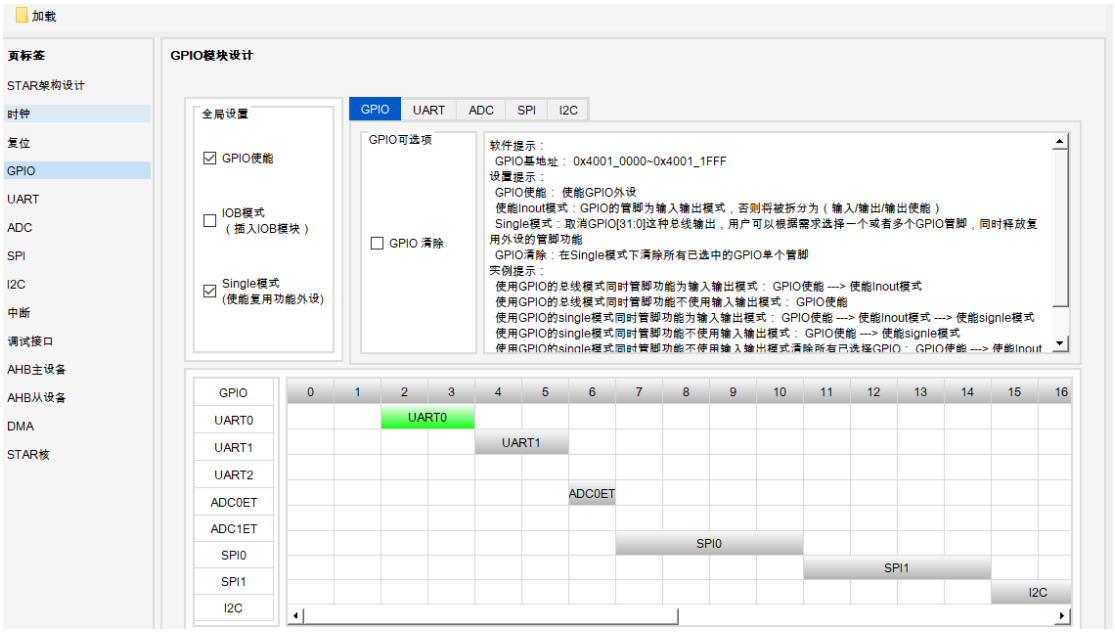


图 1-7GPIO 配置界面

从上图可以看出，配置界面分为上下两个部分，全局设置包括 GPIO 使能，使能 Inout 模式，Single 模式。下面部分是 GPIO 管脚与其功能的可视化表格。

GPIO 使能选项是 GPIO 的总线模式（输入/输出/输出使能）。使能 Inout 模式选项是输入输出模式。Single 模式选项可以选择一个或多个 GPIO 管脚。GPIO 清除选项用于清除 Single 模型下 GPIO 的选择。

选中 GPIO 使能选项后，生成的模块端口是 input wire [31:0] GPIO_IN，output wire [31:0]GPIO_OUT， output wire [31:0] GPIO_OUT_EN。选中 GPIO 使能选项且选中使能 Inout 模式选项后，生成的端口是 inout wire [31:0] GPIO。选中 GPIO 使能选项，选中 Single 模式选项，且选中表格的 GPIO 管脚后，根据选择的 GPIO 管脚生成模块的端口，例如(input wire GPIO_IN0，output wire GPIO_OUT0，output wire GPIO_OUT_EN0)，(input wire GPIO_IN1，output wire GPIO_OUT1，output wire GPIO_OUT_EN1)，…，(input wire GPIO_IN31，output wire GPIO_OUT31， output wire GPIO_OUT_EN31)。选中 GPIO 使能选项，选中使能 Inout 模式选项，选中 Single 模式选项，且选中表格中的 GPIO 管脚后，根据选择的 GPIO 管脚生成模块的端口，例如 inout wire GPIO0，inout wire GPIO1，…，inout wire GPIO31。

GPIO 复用功能用于支持两套外围设备，对于每个引脚，必须设置

ALTFUNC 的相关位用于复用功能，对应的就是每个 GPIO 的 ALTFUNCSET 位和 ALTFUNCCLR 位。在 Seal STAR 块中，GPIO 复用功能使用以下引脚，如下表所示。

表 1-2 GPIO 复用功能表

GPIO bit	INPUTALT Function	OUTPUTALT Function
0	timer0extin	value 0
1	timer1extin	value 0
2	uart0_rxd	value 0
3	unused	uart0_txd
4	uart1_rxd	value 0
5	unused	uart1_txd
6	adc_etr	value 0
7	unused	spi0_clk_out
8	unused	spi0_sel
9	spi0_data_in[0]	spi0_data_out[0]
10	spi0_data_in[1]	spi0_data_out[1]
11	unused	spi1_clk_out
12	unused	spi1_sel
13	spi1_data_in[0]	spi1_data_out[0]
14	spi1_data_in[1]	spi1_data_out[1]
15	i2c_scl_in	i2c_scl_out
16	i2c_sda_in	i2c_sda_out
17	spi0_data_in[2]	spi0_data_out[2]
18	spi0_data_in[3]	spi0_data_out[3]
19	spi1_data_in[2]	spi1_data_out[2]
20	spi1_data_in[3]	spi1_data_out[3]
21	adc1_etr	value 0
22	uart2_rxd	value 0
23	unused	Uart2_txd
24	unused	unused
25	unused	unused
26	Spi1_clk_in	unused
27	Spi1_sel_in	unused
28	Spi1_sel_in	unused
29	Spi0_sel_in	unused
30~31	unused	unused

如果需要使用 GPIO 的复用功能，必须勾选其中两项：GPIO 使能和 Single 模式（使能复用功能外设），这样下方的可视化窗口就可以点击勾选了，本次实验需要使用到串口，这里我们需要使能这两个选项。如果需要使用对应的复用功能，我们可以通过三种方式进行选择，这里我们使能 UART0，如下所示。

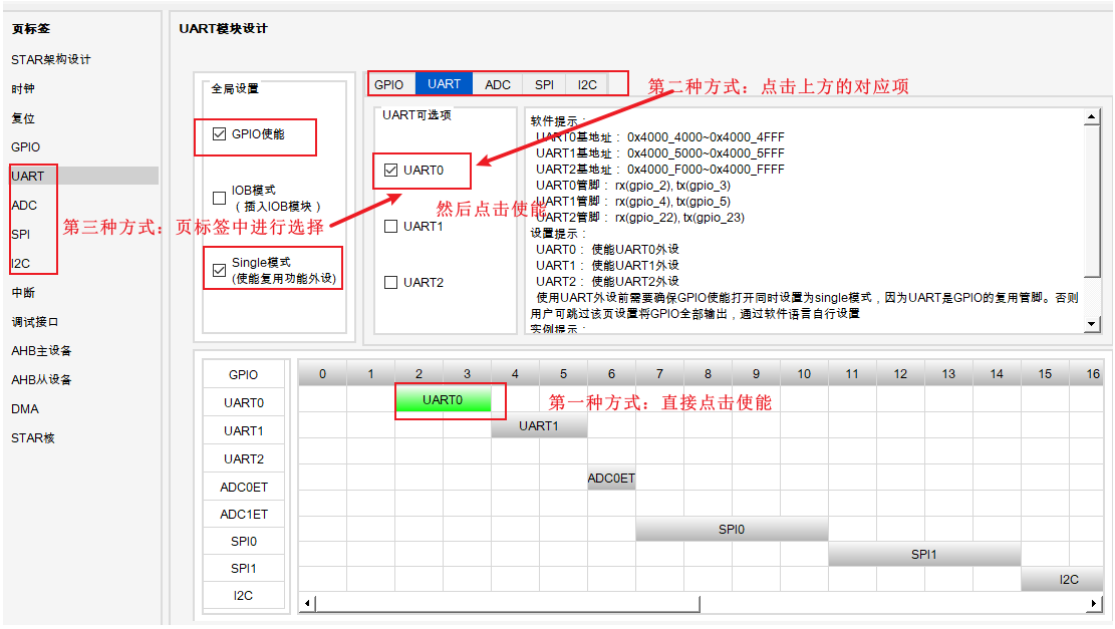


图 1-8 使能复用功能

1.2.4 UART

本次实验我们使能了 UART0，生成的端口是 input wire UART0_RX 和 output wire UART0_TX。UART0 的基址是 0x4000_4000~0x4000_4FFF，复用 GPIO2,3 管脚，对应关系是 RX(GPIO2), TX(GPIO3)。下表显示了 APB UART 的寄存器映射。寄存器地址=基地址+基地址偏移量。

表 1-3 UART 的寄存器映射表

Name	Base offset	Type	Width	Reset value	Description
DATA	0x000	RW	8	0x--	[7:0] Data value. Read Received data. Write Transmit data.
STATE	0x004	RW	4	0x0	[3] RX buffer overrun, write 1 to clear. [2] TX buffer overrun, write 1 to clear. [1] RX buffer full, read-only. [0] TX buffer full, read-only.
CTRL	0x008	RW	7	0x0	[6] High-speed test mode for TX only. [5] RX overrun interrupt enable. [4] TX overrun interrupt enable. [3] RX interrupt enable. [2] TX interrupt enable. [1] RX enable. [0] TX enable.
INTSTATUS INTCLEAR	0x00C	RW	4	0x0	[3] RX overrun interrupt. Write 1 to clear. [2] TX overrun interrupt. Write 1 to clear. [1] RX interrupt. Write 1 to clear. [0] TX interrupt. Write 1 to clear.
BAUDDIV	0x010	RW	20	0x00000	[19:0] Baud rate divider. The minimum

					number is 16.
PID4	0xFD0	RO	8	0x04	Peripheral ID Register 4: value = 8'h4
PID5	0xFD4	RO	8	0x00	Peripheral ID Register 5: value = 8'h0
PID6	0xFD8	RO	8	0x00	Peripheral ID Register 6: value = 8'h0
PID7	0xFDC	RO	8	0x00	Peripheral ID Register 7: value = 8'h0
PID0	0xFE0	RO	8	0x21	Peripheral ID Register 0: value = 8'h21
PID1	0xFE4	RO	8	0xB8	Peripheral ID Register 1: value = 8'hb8
PID2	0xFE8	RO	8	0x1B	Peripheral ID Register 2: value = 8'h1b
PID3	0xFEC	RO	8	0x00	Peripheral ID Register 3: value = 8'h0
CID0	0xFF0	RO	8	0x0D	Component ID Register 0: value = 8'hd
CID1	0xFF4	RO	8	0xF0	Component ID Register 1: value = 8'hf0
CID2	0xFF8	RO	8	0x05	Component ID Register 2: value = 8'h5
CID3	0xFFC	RO	8	0xB1	Component ID Register 3: value = 8'hb1

其他复用功能我们这里不进行说明，请自行查看相关文档。

1.2.5 中断

中断配置页面如下所示。可以左侧是可视化窗口，右侧是控制区。控制区包括外部中断使能和 Single 模式。外部中断使能选项用于使能外部中断扩展接口，Single 模式用于选择单个外部中断接口。本次实验暂不使用。

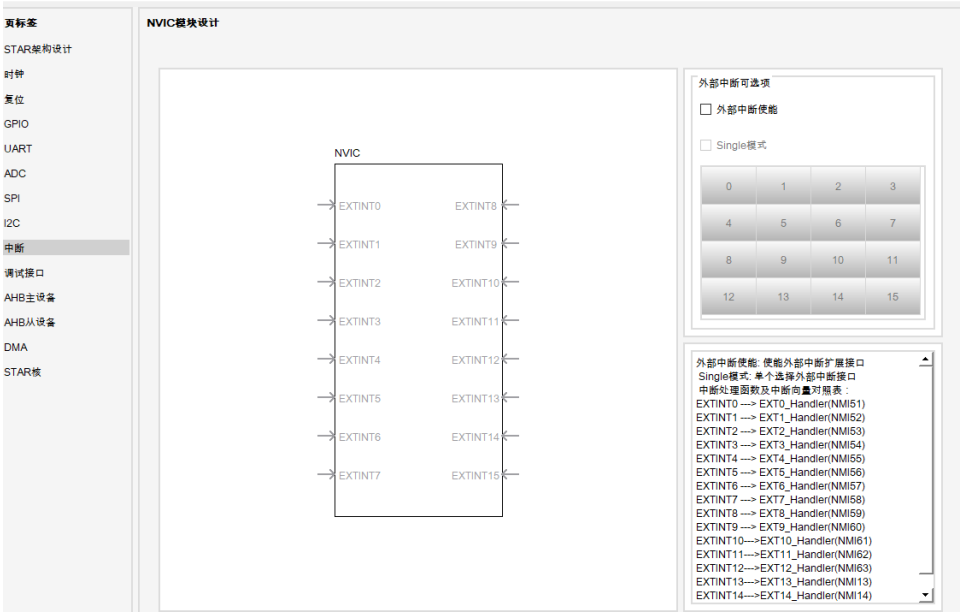


图 1-9 中断配置界面

1.2.6 调试接口

调试接口标签页如下图所示，左侧是端口的可视化框图，右侧是控制区。用户可以根据是否需要软件调试来选择性地打开它。JTAG Enable 使能选项用于启用 JTAG 接口。SW 选项使能用于启用 SW 接口。本次实验选用 SW 接口

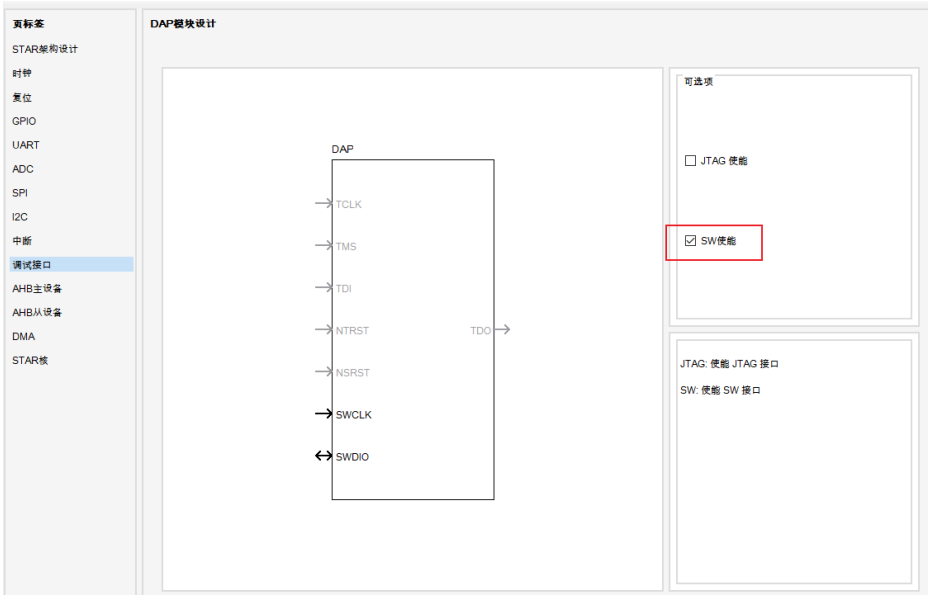


图 1-10 使能 SW 接口

1.2.7 AHB 主设备

AHB 主设备标签页如下图所示，左侧是端口的可视化框图，右侧是控制区。AHBMaster0 使能选项用于使能 AHBMaster0 总线，AHB Master1 使能选项用于使能 AHBMaster1 总线。本次实验使能 Mastrer0。

AHB Mastrer0:使能 AHB Master0 总线，总线地址为 0x60000000~0x9FFFFFFF，大小：1GB；

AHB Mastrer1:使能 AHB Master1 总线，总线地址为 0xA0000000~0xDFFFFFFF，大小：1GB；

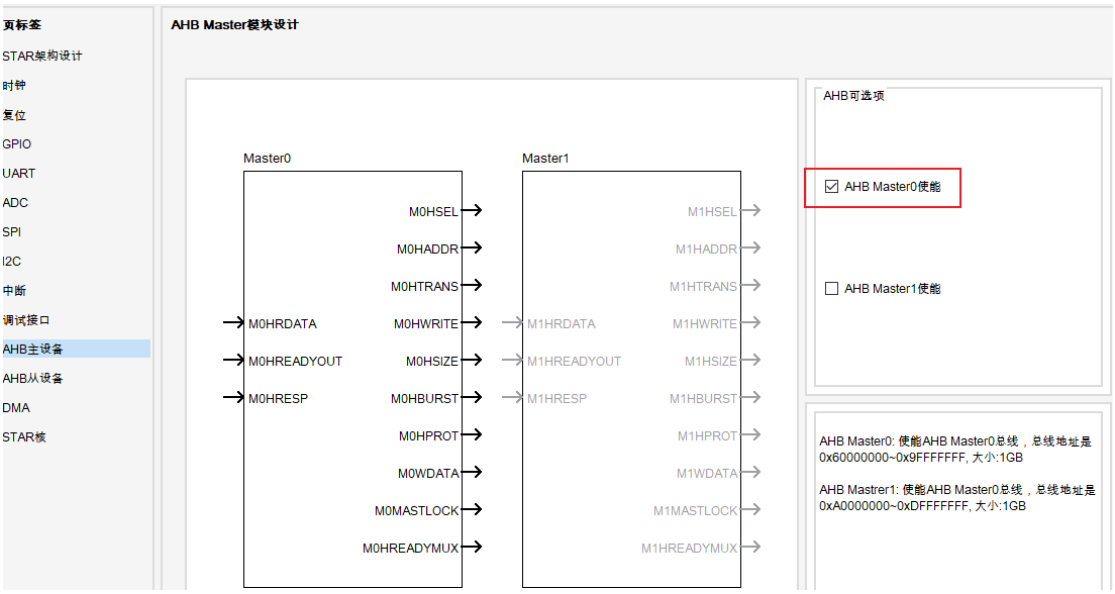


图 1-11AHB 主设备设置

1.2.8 AHB 从设备

AHB 从设备标签页如图 2-12 所示，左侧是端口的可视化框图，右侧是控制区。AHB Slave0 使能选项用于使能 AHBSlave0 总线，AHB Slave1 使能选项用于使能 AHB Slave1 总线。本次实验不使能该功能。

AHB Slave0 : 使能 AHB Slave0 总线，总线地址受控于连接的 AHB Master；

AHB Slave1 : 使能 AHB Slave1 总线，总线地址受控于连接的 AHB Master。

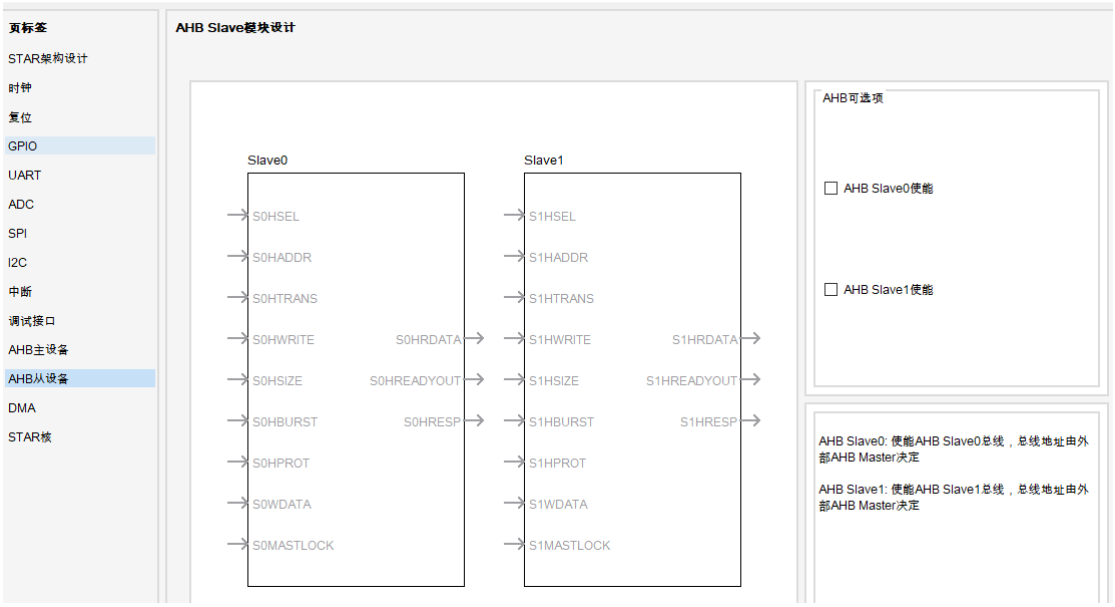


图 1-12 AHB 从设备设置界面

1.2.9 DMA

DMA 标签页如图所示，左侧是端口的可视化框图，右侧是控制区。DMA Enable 使能选项用于启用 DMA 接口。本次实验不使能该功能。

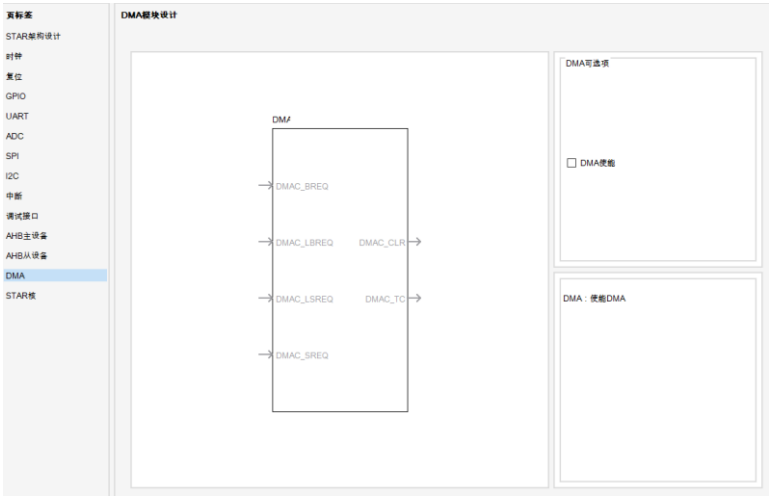


图 1-13DMA 标签页

1.2.10STAR 核

STAR 核标签页如图所示，左侧是端口的可视化框图，右侧是控制区。右侧控制区域有五个标签，分别是 REMAP，Coprocessor， FPU， QSPI 和 MPU。选择其中一个选项之后，在右下角对于该选项的设置都有详细的说明，读者请自行在软件中进行查看。

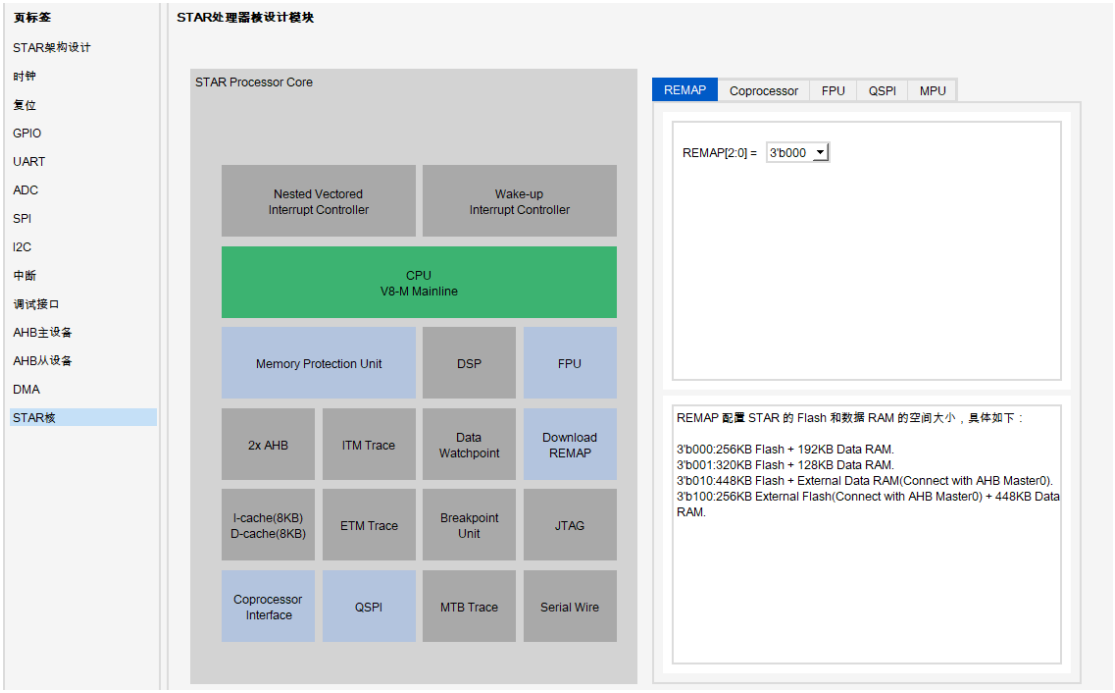


图 1-14STAR 核配置界面

对 MCU 控制器 IP 核配置完成之后，需要将其 IP 文件添加至工程中，操作如下所示。

图 1-15 添加 IP 文件

添加 IP 成功之后，右击 IP 文件，点击打开，可以看到具体的文件内容，双击可以进入 IP 配置界面。

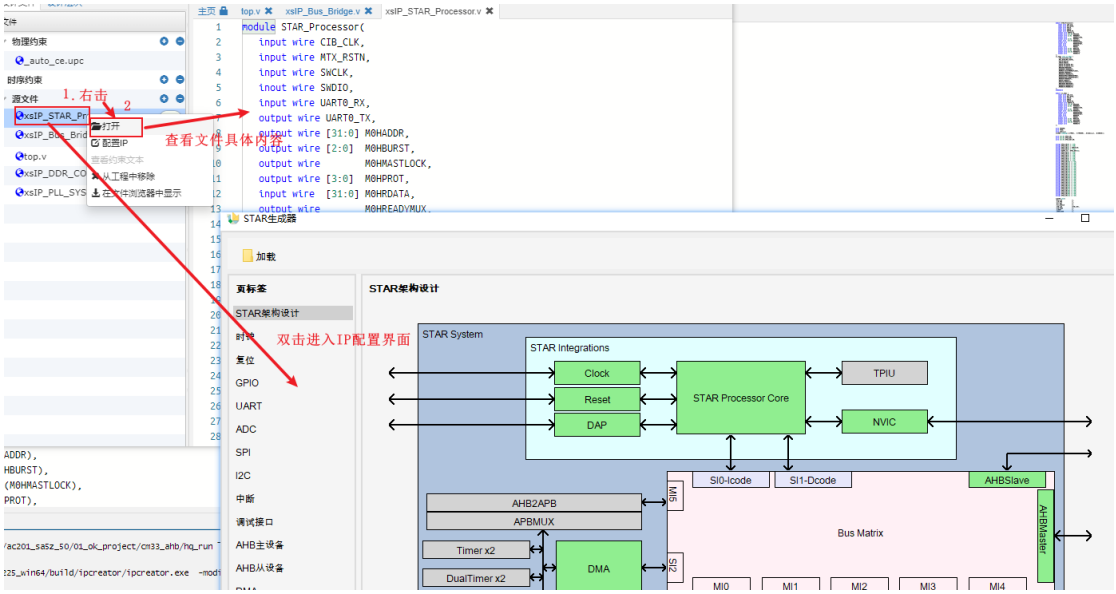


图 1-16 查看 IP 文件内容及进入 IP 配置界面的操作方式

1.3 AHB to AXI Bus_Bridge

智多晶 AHB to AXI Bus_Bridge IP 的功能是实现 AHB_Lite 总线与 AXI_Full/AXI_Lite 总线之间的桥接。用户使用此 IP 可以便捷的将 AHB-Lite Master 设备和 AXI Slave 设备连接起来，从而实现 AHB-Lite Master 设备到 AXI Slave 设备之间的数据交互。本次实验就是通过该 IP 实现 MCU 控制器和 DDR3 之间的数据交互。

AHB to AXI Bus_Bridge IP 主要由 AHB_Slave_Interface 模块、AHB_Data_Counter 模块、AXI_Master_Interface 模块，Control_Logic 等模块组成。下图是 AHB to AXI Bus_Bridge IP 的系统框图。

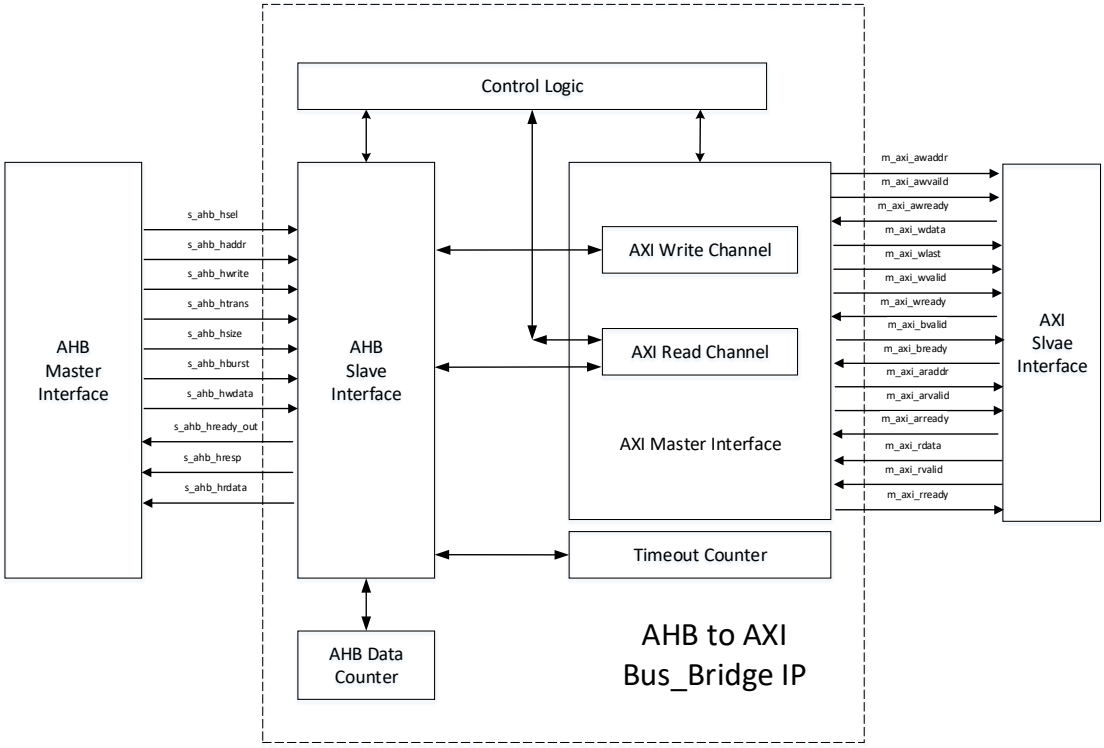


图 1-17AHB to AXI Bus_Bridge IP 系统框图

对上述图中的模块进行简要说明：

- AHB Interface 模块：AHB Interface 模块用于接收 AHB 接口的数据信息，当 AHB Master 发起一次传输后，该模块会寄存 AHB 侧的控制信号，同时会将 AHB Master 产生的控制信号处理后转发至 Control Logic 逻辑模块用于状态机的控制，来完成 AHB 侧突发传输到 AXI 侧突发传输的控制。

- Control Logic 模块：Control Logic 模块是整个 IP 的核心控制单元，负责指示 AHB Interface 模块及 AXIInterface 传输的进度控制。当 Control Logic 模块检测在 AHB Master 端发起的传输模式，如（读/写突发传输、读/写单次传输），会指示子模块将当前的 AHB 传输模式适当地映射成 AXI 传输模式。

- AHB Data Counter 模块：AHB Data Counter 模块对从 AHB Master 接收到的写入有效数据进行计数，同时将计数值发送给 AHB Interface 模块，AHB Interface 模块判断计数值满足当前传输所需数量的后拉低 s_ahb_hready_out 信号，直到 AXI 侧处理完当前突发传输后，再拉高 s_ahb_hready_out 信号，表示当前传输完成，AHB Master 可以开始下一轮 AHB 传输。

- AXI Interface 模块：AXI Interface 模块包括 WR Channel 和 RD Channel，分别处理 AXI 写和读逻辑。AXI 写、读通道根据控制逻辑的指令，分别管理 AXI 写事务相关的各个通道（包括写地址通道、写数据通道和写响应通道）和控制 AXI 读事务通道（读地址通道和读数据通道）在 AHB 总线突发终止操作时，该通道会对写、读选等通信号进行精准控制。

1.3.1 AHB 总线

1.3.1.1 AHB 总线简介

AHB 总线规范是 AMBA 总线规范的一部分，AMBA 总线规范是 ARM 公司提出的，由于其规范严谨、功能丰富、总线效率高被大多数 SOC 设计采用。完整的 AHB 总线由四部分组成：

1. AHB 主设备 Master：发起依次读/写操作；某一时刻只允许一个主设备使用总线。
2. AHB 从设备 Slave：响应一次读/写操作；通过地址映射来选择使用哪一个从设备。
3. AHB 仲裁器 Arbiter：允许某一个主设备控制总线。
4. AHB 译码器 Decoder：通过地址译码来决定选择哪一个从设备。

其基本的组成框图如下所示。

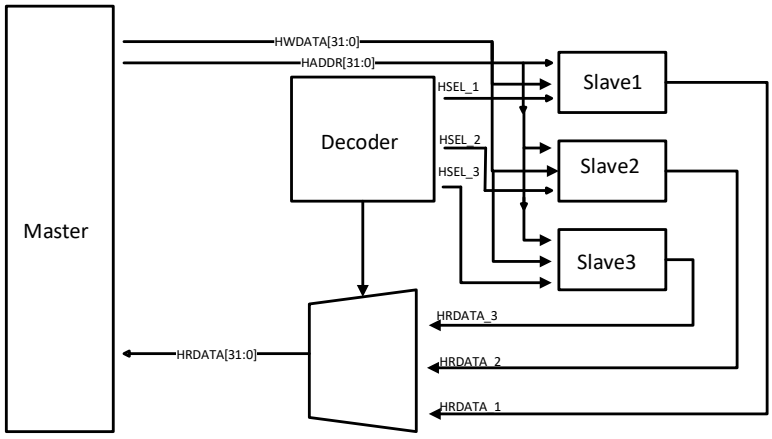


图 1-18 AHB 总线组成框图

1.3.1.2 AHB 操作概述

当需要占用总线的时候，总线的 Master 向 Arbiter 发出请求，Arbiter 授权给指定的 Master。任一时间周期只允许一个 Master 接入总线，对其指定的 Slave 进行读写操作。

总线授权的 Master 向 AHB 传输的步骤：首先发出地址和控制信号，然后提供地址信息、传输方向、带宽和 burst 类型。总线将会统一规划 Slave 的地址，译码器根据地址和控制信号确定哪个 Slave 与 Master 进行数据通信。为了避免出现三态总线，AHB 将读写总线分开，写数据总线将用于从 Master 到 Slave 的数据传输，读数据总线用于从 Slave 到 Master 的数据传输。每笔传输包括一个地址和控制周期，一个或者多个数据周期。地址和控制周期不能被扩展，因此 Slave 必须在一个周期内采样地址信号。数据周期可以通过 HREADY 信号扩展，当 HREADY 为低时，给传输加入等待状态以使 Slave 获得额外的时间来提供或采样数据，另外 Slave 通过可以响应信号 HRESP 反映传输状态。

AHB 支持批量式数据传送，可以自动递增地址。递增地址的方式分为：持续递增与回绕传送。一般情况下 Master 完成完整的 burst 传输，Arbiter 才会授权给其他的 Master 接入总线，然而为避免过大的判决延迟，Arbiter 也可能打断 burst 传输。在这种情况下 Master 必须再次接入总线以进行中断的 burst 剩余部分的传输。

1.3.1.3 AHB 信号描述

对于 AHB 信号描述如下所示。

表 1-4 AHB 信号描述表

信号名	含义	源	I O	描述
HCLK	总线时钟	clock source	各 module	时钟为所有总线传输提供基准频率。所有信号时序都和 HCLK 的上升沿有关
HRESETn	复位	reset controller	各 module	总线复位信号，低电平有效，用于复位系统和总线
HSEL	从选择	Decoder	slave	每隔从机都有自己独立的从机选择信号，并且该信号可以表示当前传输的是否为选中的从机。
HADDR[31:0]	地址总线	Master	decoder;mux to slave;arbiter	32 位系统地址总线
HTRANS[1:0]	传送状态	Master	mux to slave	当前传输状态。
HSIZE[2:0]	传送带宽	Master	mux to slave	每一个 transfer 传输的数据大小，以字节为单位，最高支持 1024 位。

HWRITE	传送方向	Master	mux to slave	1 为写，0 为读
HWDATA[31:0]	写数据总线	Master	mux to slave	写数据总线，用来在写操作期间将数据从主机传送到从机，建议最小的数据总线宽度为 32 位。
HREADY	传送完成	Slave	mux to master; arbiter	该信号为高电平时，表示主机和所有的从机传输已完成
HREADYOUT	传送完成	Slave	mux to master; arbiter	高电平表示传输已完成，低电平表示可以继续传输
HRDATA[31:0]	读数据总线	Slave	mux to master	读数据总线，通过译码选择通道
HRESP	传送响应	Slave	mux to master; arbiter	Slave 发给 Master 的总线传输状态。00：OKAY，传输完成；01：ERROR：传输错误；10：RETRY，传输未完成，请求主设备重新开始一个传输，Arbiter 会继续使用通常的优先级；11：SPLIT，传输未完成，请求主设备分离一次传输，Arbiter 会调整优先级方案以便其他请求总线的主设备可以访问总线。
HBURST[2:0]	批量传送	Master	mux to slave	表示传输是否组成了突发的一部分。支持 4 个，8 个，16 个节拍的突发传输，突发传输可以使增量或回环。
HPROT[3:0]	保护控制	Master	mux to slave	保护控制信号，需要 slave 带保护功能，一般不用

对 AHB 控制信号说明如下：

1. 传送状态 HTRANS[1:0]

在 AHB 总线上，M 的传送状态可以由 HTRANS[1:0]来表示，关于这两位的说明如下所示。

表 1-5 传送状态位说明表

HTRANS[1:0]	类型	含义	描述
00	IDLE	空闲状态	此时控制 AHB 总线的 M（Master 得到允许后的信号），M 以 IDLE 表示没有数据传送请求。S 必须忽略此时的传送（已收到的地址与其它控制信号），并立刻以零等待 OKAY 信号来对 M 进行响应
01	BUSY	忙状态	M 以此信号表示正在处理数据，此时 SLAVE 要响应 OKAY 的信号（零等待状态），此时 SLAVE 会忽略已收到的地址和其它控制信号。一般在一个批量传送的中间，表示 M 发起一次批量传送，但是下一个传送不能立即进行，这样就使 AHB M 能在传送的中间插入空闲周期，这时候的地址和控制信号必须反映下一个传送数据的情况。作为 AHB 从设备必须忽略此时的传送，并以零等待的 OKAY 状态来响应 M。
10	NONSEQ	非连续状态	表示当前是单笔数据或突发传送的第一笔数据的传送，此时

			所送出的地址及控制信号与上一笔的传送无关。
11	SEQ	连续状态	表示突发传送剩余的数据传送时顺序传送，地址及控制信号与前面的操作相关。地址等于上一笔地址加控制信号 HSIZE 字节，控制信号与上一笔数据相同。

2. 批量传送 HBURST[2:0]

每次传送的数据大小由 HBURST[2:0]定义，说明如下所示。

表 1-6 批量传送状态位说明表

HBURST[2:0]	类型	描述
000	SINGLE	单笔数据传送
001	INCR	不定长的递增方式的批量传送
010	WRAP4	4 个数据的回绕方式的批量传送
011	INCR4	4 个数据的递增方式的批量传送
100	WRAP8	8 个数据的回绕方式的批量传送
101	INCR8	8 个数据的递增方式的批量传送
110	WRAP16	16 个数据的回绕方式的批量传送
111	INCR16	16 个数据的递增方式的批量传送

AHB 从设备通常支持 SINGLE，INCR，INCR4，WRAP8，INCR8，WRAP16，INCR16。AHB 对传送范围规定不超过 1KB，递增传送的大小为任意数目，但不可超过 1KB 的范围。

3. 传送方向 HWRITE

HWRITE 拉高时（写），M 必须对写入动作初始化，数据会由 M 放到 HWDATA[31:0]总线上。HWRITE 拉低时（读），M 会对读取动作初始化，被寻址到的 S 会将数据放到 HRDATA[31:0]总线上。

4. 传送大小 HSIZE[2:0]

传输数据大小由 HSIZE[2:0]控制，表示每次传送的字节数目，对其说明如下所示。

表 1-7 传输数据大小状态位说明表

HSIZE[2:0]	大小	描述
000	8 位	字节
001	16 位	半字
010	32 位	字
011	64 位	-
100	128 位	4-字数据线
101	256 位	8-字数据线
110	512 位	-
111	1024 位	-

HSIZE[2:0]和 HBURST[2:0]两个信号可合起来定义回绕地址的范围。例如：

HSIZE[2:0]=32 位，HBURST[2:0]长度为 4 字节，则回绕地址必须对齐在 16 字节。通常的 AHB 从设备是 32 位数据线，所以它支持 8 位，16 位和 32 位 3 种大小。

5. 保护控制 HPROT[3:0]

提供总线访问的附加信息，主要是给那些希望执行某种保护级别的模块使用的。这个信号指示当前传输是否为预取指令或者数据传输，同时也表示传输是某种保护模式访问还是用户模式访问，对带存储器管理端元的总线主机而言，这些信号也用来指示当前传输时高速缓存（cache）还是缓冲的（buffer）。

1.3.1.4 AHB 模块接口

1. 从接口框图

从接口的框图如下所示。

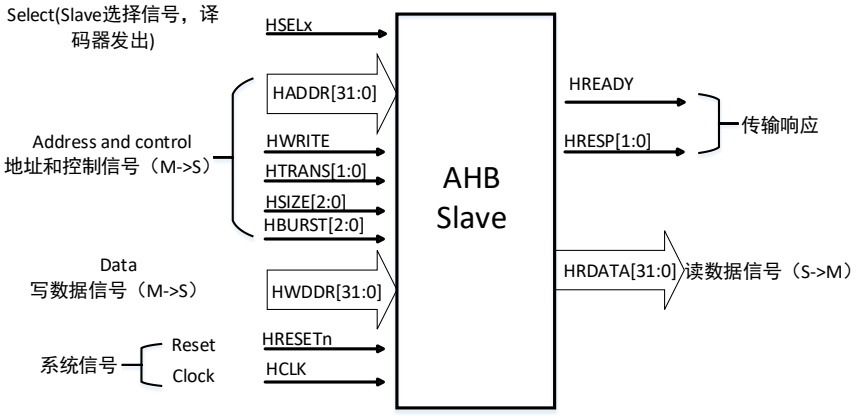


图 1-19 从接口的框图

AHB 总线中的仲裁器决定哪一个 Master 拥有总线使用权之后，会将这个 Master 的数据地址、控制信号及预写入 Slave 的数据选出，并且送至每一个 Slave，所选出的数据地址会经由 AHB 译码器产生唯一的 HSELx 使能信号来启动 Slave 进行数据传送。Master 启动一个数据传送之后，被使能的 Slave 会发出 HREADY 信号来决定是否延长当前的数据传送，若 HREADY 为 0，表示此笔数据的传递必须被延迟，若 HREADY 为 1，表示能够完成此笔数据的传递。

由上图可以发现，Slave 除了用 HREADY 信号来告知此笔数据是否需要额外的延迟时间之外，还会通过 HRESP[1:0]信号响应当前传送的情形。

2. 主接口框图

主接口的框图如所示。

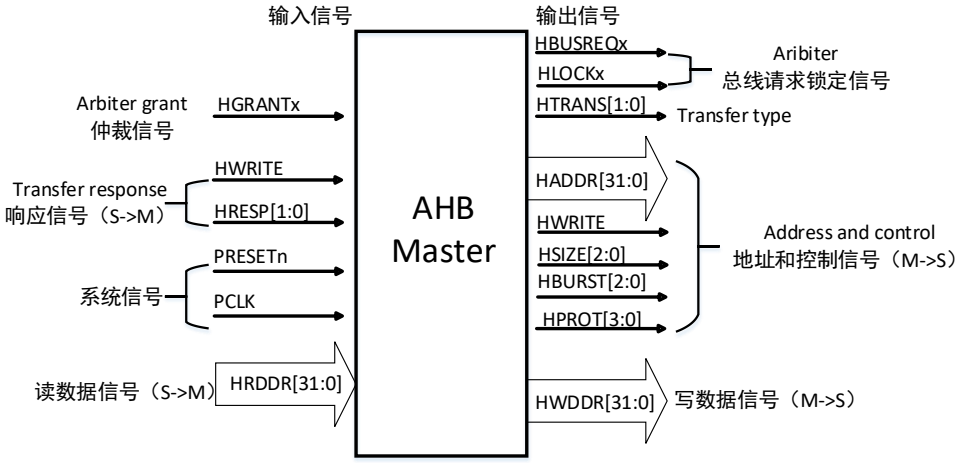


图 1-20 主接口框图

每一次的数据传送可以分为四种型态，Master 用 HTRANS[1:0]信号来决定此次传送数据的型态。

1.3.2 AXI 总线

AXI （Advanced eXtensible Interface）是 ARM 公司推出的 AMBA（Advanced Microcontroller Bus Architecture）协议家族中的新一代高性能总线标准，专为高带宽、低延迟的多核系统设计。AXI 在 AHB 基础上进一步优化，支持更复杂的互联结构，广泛应用于现代 SoC（如 ARM Cortex-A 系列处理器、GPU、高速外设等）。

AXI 最早的版本为 AXI3，协议中地址/控制和数据相位是分离的，支持不对齐的数据传输、Outstanding 传输访问和乱序访问。数据以突发（burst）的形式组织，只需要首地址，就能完成一次多数据的突发传输。

2010 年，ARM 公司发布了 AMBA 4.0 协议，AXI 协议也由 AXI3 升级为 AXI4。相较于 AXI3，AXI4 协议移除了一些不太实用的信号，比如移除了用于标志写指令 ID 的 WID 信号，因此 AXI4 不再支持乱序写；除此之外添加了一些新的信号，比如说用户信号和 Qos 信号（Quality of Service）；对一些功能也进行了修改，比较有代表的就是突发长度由原来的最高 16，变为了最高 256；除了这些之外，AXI4 还定义了一种新的协议——AXI4-Lite，这是一种简化版的 AXI4 协议，应用于一些总线性能要求较低的场景。

在本次实验中，DDR3 控制器使用的就是 AXI4 协议，所以本章我们学习 AXI4 协议。AXI4 接口协议可以分为 AXI4、AXI4-Lite、AXI4-Stream 三种总线协议。AXI4 接口协议可以说是 AXI4 协议的完整版，AXI4-Lite 是 AXI4 的简化

版，AXI4-Stream 是一种特殊的数据流范式协议，结构较为简单。因此，本节内容主要以带领大家学习 AXI4 接口协议为主，通过 AXI4 接口协议带大家了解 AXI4 协议的工作流程及原理，并结合具体的波形信号分析验证。

1.3.2.1 读写事务通道

AXI4 和 AXI4-Lite 接口由五个不同的通道组成，即：

1. 写地址通道（write address channel，AW）
2. 写数据通道（write data channel，W）
3. 写响应通道（write response channel，B）
4. 读地址通道（read address channel，AR）
5. 读数据通道（read data channel，R）

这五个通道一起组合成了读事务与写事务，用以在主设备与从设备间传输数据。读事务与写事务的结构如图所示。

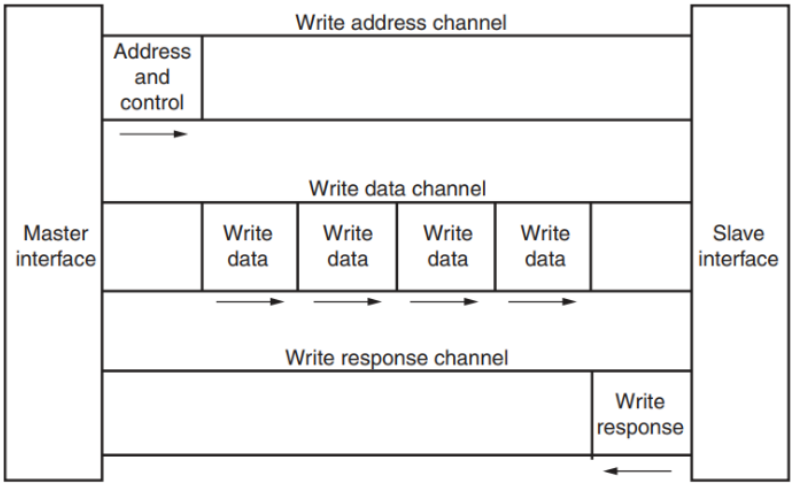


图 1-21 写事务结构

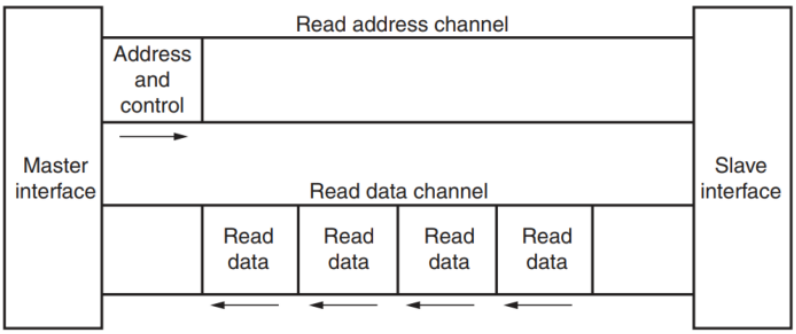


图 1-22 读事务结构

这两种事务包含以下特点：

1. 这 5 条独立的通道都包含一个双路的 VALID、READY 握手机制。信息源通过 VALID 信号来指示通道中的数据和控制信息什么时候有效。目地源用 READY 信号来表示何时准备好接收数据。传输地址信息和数据都是在 VALID 和 READY 同时为高时有效。
2. 读数据和写数据通道都包括一个 LAST 信号，用来指明一个事务传输的最后一个数据。
3. 读/写事务都有自己的地址通道，地址通道携带着传输事务所必须的地址和信息。
4. 读数据通道传送着从设备到主机的读数据和读响应信息。读响应信息指明读事务的完成状态。
5. 写数据通道传送着主机向设备的写数据和写控制信息（TLAST）。写响应通道提供了设备响应写事务的一种方式。在每一次突发式写会产生一个完成信号。

而每个 AXI4-Stream 都充当具有握手数据流的单个单向通道。因此，AXI4-Stream 接口没有以上五个通道，但是传输前需要先进行上述握手过程，在握手完成后，数据会被直接传输。

1.3.2.2 通道信号介绍

AXI4 接口的每个通道都包含了多个信号线，这些信号线的位宽大多都可以自定义，根据设计的需求，并不是所有的信号都需要被使用到，各个通道信号以及全局信号描述如下：

1. 全局信号

表 1-8 全局信号说明表

信号	源	描述
ACLK	时钟源	公共时钟信号
ARESETn	复位源	公共复位信号

2. 写地址通道信号

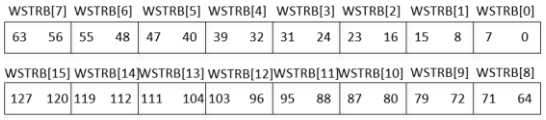
表 1-9 写地址通道信号说明表

信号	源	描述
AWID	主机	写地址 ID，这个信号是写地址信号组的 ID tag

AWADDR	主机	写地址，给出一次写突发传输的写地址
AWLEN	主机	突发式写的长度，突发式写所传输的数据的个数。数据个数 = AWLEN + 1
AWSIZE	主机	突发式写的大小，给出每个突发传输数据的字节数
AWBURST	主机	突发式写的类型
AWLOCK	主机	锁类型
AWCACHE	主机	Cache 类型。这信号指明事务的 bufferable、cacheable、write-through、write-back、allocate attributes 信息
AWPROT	主机	保护类型，表明一次传输的特权级及安全等级
AWQOS	主机	质量服务 QoS
AWUSER	主机	用户自定义信号
AWVALID	主机	写地址有效。 1 = 地址和控制信息有效 0 = 地址和控制信息无效 这个信号会一直保持，直到 AWREADY 变为高
AWREADY	主机	写地址准备好。这个信号用来指明设备已经准备好接受地址和控制信息了。 1 = 设备准备好 0 = 设备没准备好

3. 写数据通道信号

表 1-10 写数据通道信号说明表

信号	源	描述
WDATA	主机	写的数据
WSTRB	主机	写数据有效的字节阀门，用来表明哪 8bits 数据是有效的。WSTRB[n]标示的区间为 WDATA[(8*n)+7:(8*n)] 
WLAST	主机	此次传输是最后一个数据传输
WUSER	主机	用户自定义信号
WVALID	主机	写有效 1 = 写数据和阀门有效 0 = 写数据和阀门无效
WREADY	从机	写就绪。指明设备已经准备好接受数据了 1 = 设备就绪 0 = 设备未就绪

4. 写响应通道信号

表 1-11 写响应通道信号说明表

信号	源	描述
BID	从机	写响应 ID， 这个数值必须与 AWID 的数值匹配
BRESP	从机	写响应。这个信号指明写事务的状态。 2’b00：OKAY，2’b01：EXOKAY，2’b10：SLVERR，2’b11：DECERR
BUSER	从机	用户自定义信号

BVALID	从机	写响应有效。 1 = 写响应有效 0 = 写响应无效
BREADY	主机	接受响应就绪。该信号表示主机已经能够接受响应信息。 1 = 主机就绪 0 = 主机未就绪

5. 读地址通道信号

表 1-12 读地址通道信号说明表

信号	源	描述
ARID	主机	读地址 ID，用来标志一组读信号
ARADDR	主机	读地址，一次读突发传输的读地址
ARLEN	主机	突发式读长度，突发传输数据的个数，数据个数 = ARLEN + 1
ARSIZE	主机	突发式读大小，每个突发传输数据的字节数
ARBURST	主机	突发式读类型
ARLOCK	主机	总线锁信号，可提供操作的原子性
ARCACHE	主机	Cache 类型
ARPROT	主机	保护类型，一次传输的特权级及安全等级
ARQOS	主机	质量服务 QoS
ARUSER	主机	用户自定义信号
ARVALID	主机	读地址有效。信号一直保持，直到 ARREADY 为高。 1 = 地址和控制信息有效 0 = 地址和控制信息无效
ARREADY	从机	读地址就绪。指明设备已经准备好接受数据了。 1 = 从机就绪 0 = 从机未就绪

6. 读数据通道信号

表 1-13 读数据通道信号说明表

信号	源	描述
RID	从机	读 ID tag。RID 的数值必须与 ARID 的数值匹配
RDATA	从机	读数据
RRESP	从机	读响应。这个信号指明读传输的状态：OKAY、EXOKAY、SLVERR、DECERR
RLAST	从机	读事务传送的最后一个数据
RUSER	从机	用户自定义信号
RVALID	从机	读数据有效。 1 = 读数据有效。 0 = 读数据无效。
RREADY	主机	读数据就绪。 1 = 主机就绪 0 = 主机未就绪

AXI4 中所有信号都在全局时钟 ACLK 的上升沿采样，所有输出信号都必须在 ACLK 上升沿之后发生变化。从各个通道信号的名称可以看到，它们的命名

是有规律的：读地址信号都是以 AR 开头（A：address；R：read）；写地址信号都是以 AW 开头（A：address；W：write）；读数据信号都是以 R 开头（R：read）；写数据信号都是以 W 开头（W：write）；应答信号都是以 B 开头（B：back（answer back））。

这些信号互相配合，进而实现各个通道的传输，下面以 AXI4 接口的典型时序为例，首先带大家简单了解 AXI4 如何实现读写事务的传输。

1.3.2.3 典型时序

典型的突发写和读时序如下所示，其中绿色信号为全局信号，红色信号为主机发送给从机的信号，蓝色信号为从机发送给主机的信号。

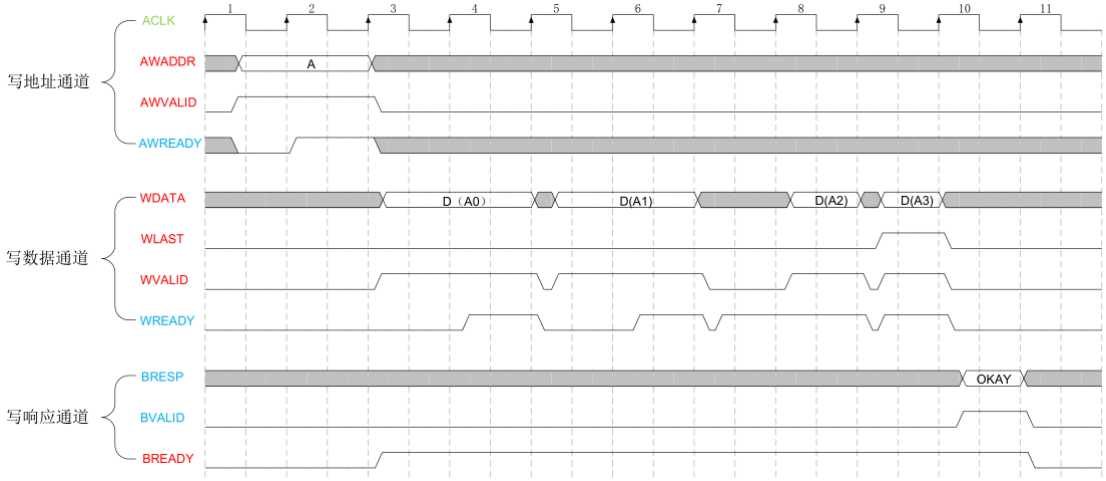


图 1-23 写事务典型时序图

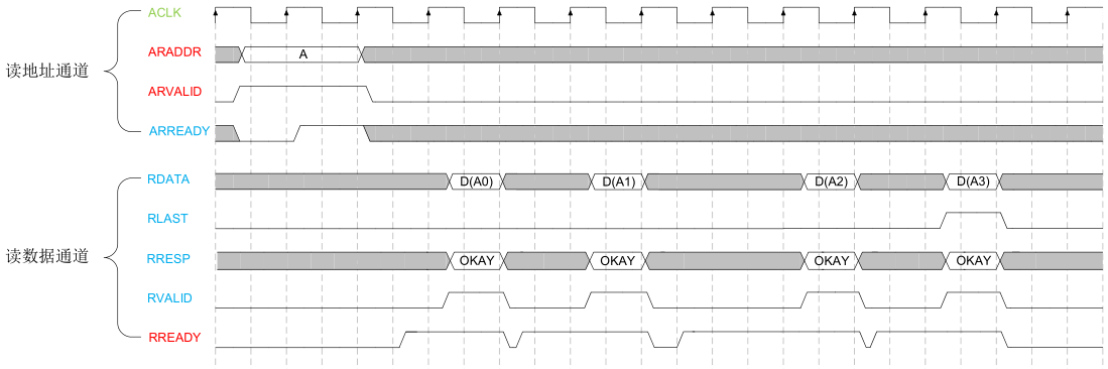


图 1-24 读事务典型时序图

在进行写事务时，主机首先会将待写入数据的地址以及控制信号放入写地址通道，随后进行握手，握手成功后，地址和控制信号被写入从机。这里的控制信号包括突发长度、数据包大小、突发类型等等。接着主机通过写数据通道

向从机传输数据，每传输一次数据就要进行一次握手，只有握手成功，数据才会被写入到从机中。当最后一组数据被放到写数据通道时，主机还会产生一个写控制信号 WLAST，用来告诉从机这是最后的数据。从机在接收了最后一 Beat 以及 WLAST 信号后，会通过写应答通道向主机发送写响应信号 BRESP，以告知主机，本次传输状态。

在进行读事务时，主机同样首先将待读取的地址以及控制信号放入读地址通道，随后进行握手，握手成功后，地址和控制信号被写入从机。接着从机将指定地址的读数据放入写数据通道，并产生读响应信号，当读数据通道握手成功时，数据与响应信号被发送给主机。当最后一 Beat 数据被放到读数据通道时，从机会拉高读控制信号 RLAST，以告知主机本次读传输结束。

我们可以看到，AXI4 的读事务只有两个通道，相对于写事务而言，没有了应答通道。这里的读地址通道同样被主机用来传输地址和控制信号，而读数据通道，除了用来传输待读取数据外，还用来传输读响应信号，指示读事务的完成状态，也因此，读事务不需要读响应通道。

至于写响应信号为什么要单独设立一个通道，这是因为 AXI 协议中存在握手机制， 每个通道在进行数据交互前，都需要先进行双向握手（VALID 和 READY 信号）。当两个信号都为高时，传输线上的数据才会有效，这种握手机制只允许一个方向的数据流，对于读事务，数据方向与响应方向一致，由从机流向主机；而对于写事务，数据方向与响应方向相反，所以就需要单独的响应通道。

1.3.3 AHB 与 AXI 突发映射关系

AHB 和 AXI 支持的突发传输类型如下表所示。

表 1-14 AHB 和 AXI 突发传输类型对比说明表

AHB 突发传输类型				AXI 突发传输类型		
HBURST[2:0]	Type	Description		AxBURST	Label	Meaning
3’b000	SINGLE	Single transfer brust		2’b00	FIXED	Fixed burst
3’b001	INCR	Incrementing burst of underfined length		2’b01	INCR	Incrementing burst
3’b010	WRAP4	4-beat wrapping burst		2’b10	WRAP	wrapping burst
3’b011	INCR4	4-beat incrementing burst		2’b11	RESERVED	-
3’b100	WRAP8	8-beat wrapping burst				
3’b101	INCR8	8-beat incrementing burst				

AHB to AXI Bus_Bridge IP 的主要功能是将标准 AHB 传输模式映射成为 AXI 传输模式，对于 AHB 不同的传输模式，经过转换后的 AXI 传输模式也不同。

AHB to AXIBus_Bridge IP 对 AHB 和 AXI 传输模式的转换如表所示。

表 1-15 AHB 到 AXI 突发映射关系

AHB Transaction	AXI Transaction
Single 类型 Transaction	Length 为 1 的 INCR 类型 Transaction
INCR4、INCR8 和 INCR16 类型 Transaction	Length 为 4、8 和 16 的 INCR 类型 Transaction
未定义 Length 长度的 INCR 类型 Transaction	根据 AHB 侧实际 Length 长度拆分为多个 Length 为 1 的 INCR 类型 Transaction

1.3.4 AHB 与 AXI 数据传输转换时序图

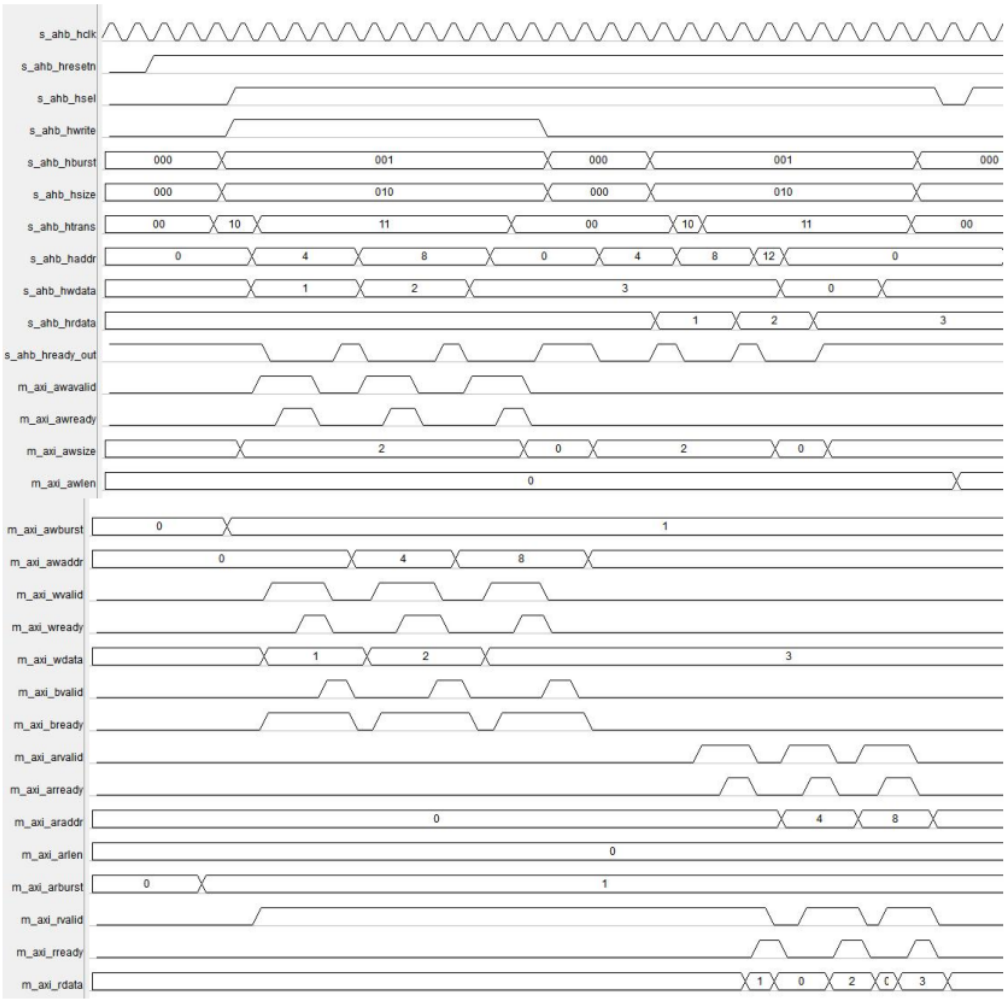


图 1-25 INCR 写/读传输时序图

AHB Master 侧发起一次 INCR 不定长度的写突发传输，如上图所示，s_ahb_hsel 为高、s_ahb_hwrite 为高、s_ahb_htrans 为 10 表示一次写突发传输开始，s_ahb_hburst 为 001 表示的是 INCR 不定长度的突发传输，当 AHB to AXI Bus_Bridge IP 采样到上述信号后，会拉高 m_axi_awvalid 给 AXI Slave 表示写控制信号有效，当 AXI Slave 正确接收到 AHB to AXIBus_Bridge IP 产生的写控制

信号 m_axi_awaddr、m_axi_awlen、m_axi_awburst 等信号后，AXI Slave 会拉高 m_axi_awready 信号给 AHB to AXI Bus_Bridge IP 指示写控制完成；同时当 AHB 写数据有效时，AHB to AXI Bus_Bridge IP 会拉高 m_axi_wvalid 信号给 AXI Slave 表示写数据信号有效，当 AXI Slave 正确接收到写数据后，会拉高 m_axi_wready 给 AHB to AXI Bus_Bridge IP 指示写数据完成；当写控制信号和写数据传输完成后，AHB to AXI Bus_Bridge IP 会拉高 m_axi_bready 给 AXI Slave，当 AXI Slave 拉高 m_axi_bvalid 时，表示一次写传输完成。

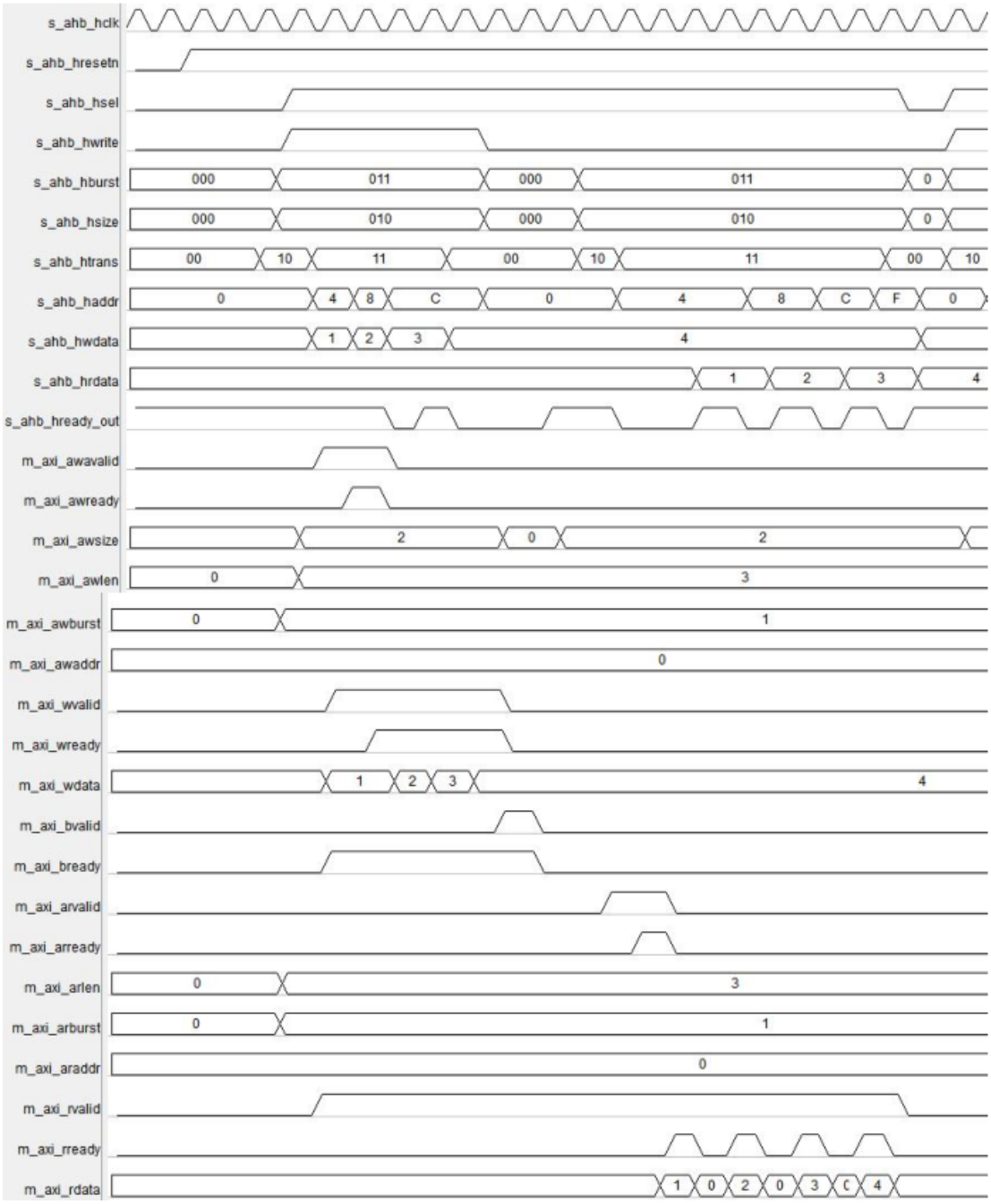


图 1-26 INCR4 写/读传输时序图

AHB Master 侧发起一次 INCR 不定长度的读突发传输，如上图所示，s_ahb_hsel 为高、s_ahb_hwrite 为低、s_ahb_htrans 为 10 表示一次读突发传输开始，s_ahb_hburst 为 001 表示的是 INCR 不定长度的突发传输，当 AHB to AXI Bus_Bridge IP 采样到上述信号后，会拉高 m_axi_arvalid 给 AXI Slave 表示读控制信号有效，当 AXI Slave 正确接收到 AHB to AXI Bus_Bridge IP 产生的读控制信号 m_axi_araddr、m_axi_arlen、m_axi_arburst 等信号后，AXI Slave 会拉高 m_axi_arready 信号给 AHB to AXI Bus_Bridge IP 指示读控制完成；同时当 AHB 读数据有效时，AXI Slave 会拉高 m_axi_rvalid 信号给 AHB to AXI Bus_Bridge IP 表示读数据信号有效，当 AHB to AXI Bus_Bridge IP 正确接收到读数据后，会拉高 m_axi_rready 信号表示一次读传输完成。

将 AHB Master 侧固定写、读突发传输模式 INCR4 映射成 AXI 侧写、读突发传输模式，与上述 INCR 映射类似，只是当 AHB Master 侧发起写、读突发传输 INCR4 模式时，AHBMaster 侧信号 s_ahb_hburst 的值为 011，AHB to AXI Bus_Bridge IP 接收到该信号会 AHB 传输映射成突发长度为 4 的 AXI 突发传输（m_axi_awlen、m_axi_awlen 值为 3）。

1.3.5 IP 配置

进入 IP 管理器界面，找到 Bus_Bridge，双击进行创建 IP，进入 IP 配置界面之后，Bus_Bridge 模式设置为 AHB to AXI_FULL。

图 1-27 Bus_Bridge IP 配置

配置完成之后，还需要将生成的 IP 文件添加工程，可以参考前面添加 MCU 控制器文件的操作。

1.4 DDR 内存控制器

DDR 的详细配置可以参考二端口 DDR3 控制器模块设计一节的内容，本次实验的配置如下所示。

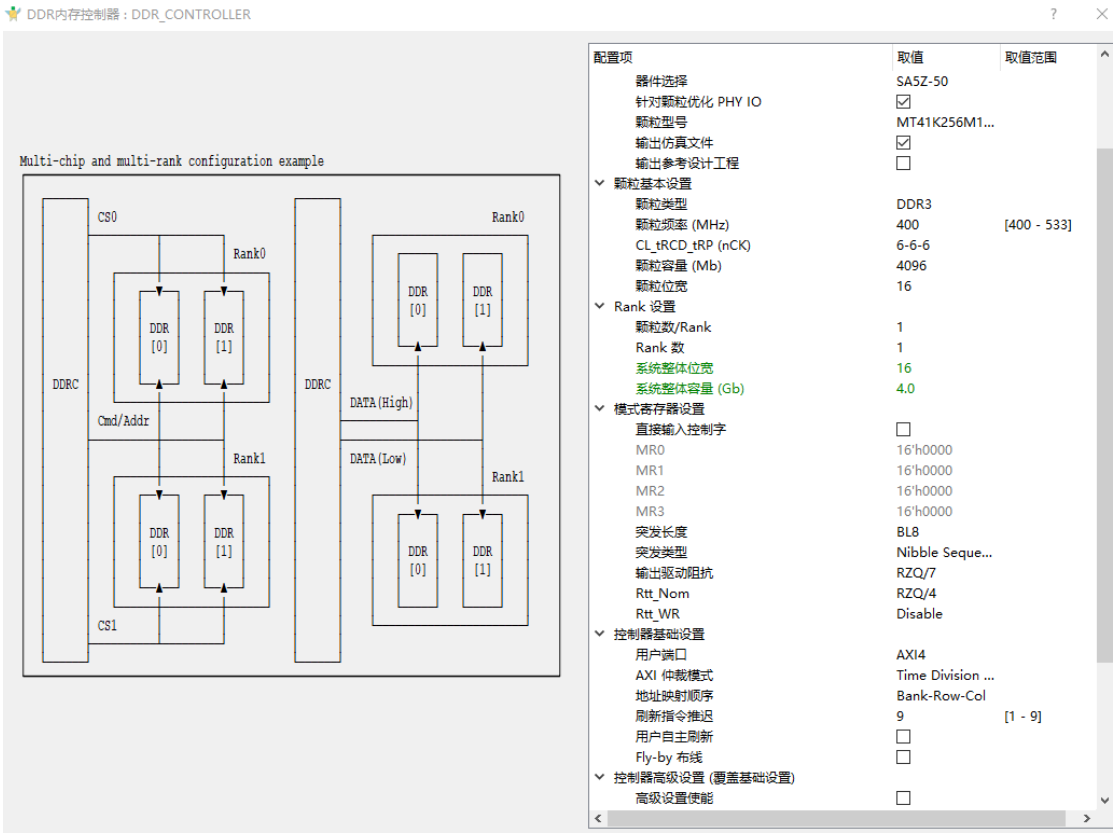


图 1-28 DDR 内存控制器配置

配置完成之后，也需要手动将文件添加至工程中。

1.5 PLL 锁相环配置

我们通过锁相环生成本次实验需要的时钟频率，输入时钟为 25M，由外部晶振提供，分别输出 400M、100M 和 50M 的时钟，配置如下所示。

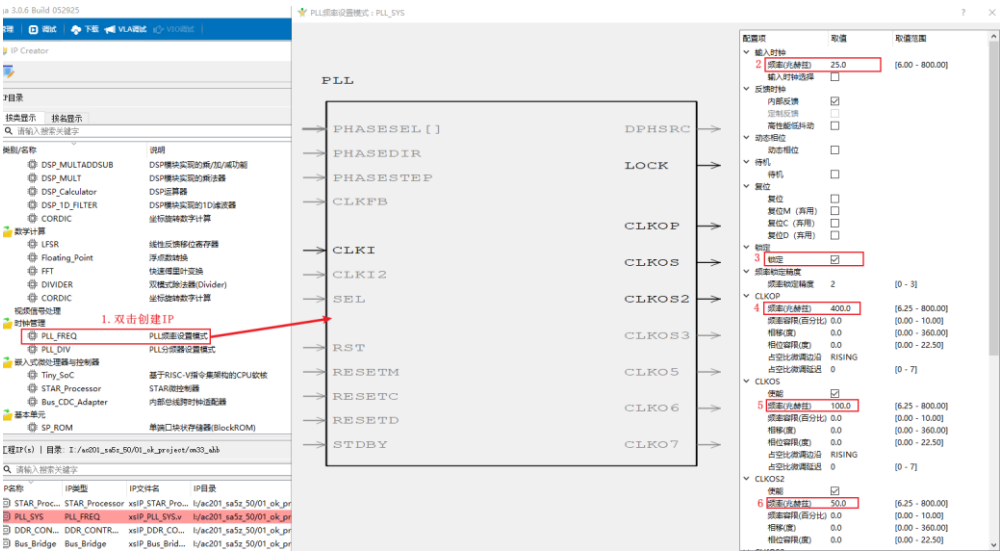


图 1-29 PLL 锁相环配置

1.6 创建顶层文件

创建顶层文件，将生成的文件例化到顶层文件，代码如下所示：

```
module top
(
    input wire CIB_CLK,                 //25M
    input wire MTX_RSTN,
    input wire SWCLK,
    inout wire SWDIO,

    //UART
    input wire UART0_RX,
    output wire UART0_TX,

    //DDR3 Interface
    // Inouts
    inout  [15:0]        ddr3_dq       ,
    inout [1:0]          ddr3_dqs      ,
    // Outputs
    output [14:0]        ddr3_addr     ,
    output [2:0]         ddr3_ba       ,
    output               ddr3_ras_n    ,
    output               ddr3_cas_n    ,
    output               ddr3_we_n     ,
    output               ddr3_reset_n  ,
    output               ddr3_clk      ,
    output [0:0]         ddr3_cke      ,
    output [0:0]         ddr3_cs_n     ,
    output [1:0]         ddr3_dm       ,
```

```
    output [0:0]         ddr3_odt      ,

    output               led

);

    //pll
    wire                    pll_lock           ;
    wire                    sys_clk            ;
    wire                    usr_clk            ;
    wire                    phy_clk            ;

    wire [2:0]  init_status;

    //ahb
    wire [31:0] M0HADDR;
    wire [2:0]  M0HBURST;
    wire        M0HMASTLOCK;
    wire [3:0]  M0HPROT;
    wire  [31:0] M0HRDATA;
    wire        M0HREADYMUX;
    wire         M0HREADYOUT;
    wire         M0HRESP;
    wire        M0HSEL;
    wire [2:0]  M0HSIZE;
    wire [1:0]  M0HTRANS;
    wire [31:0] M0HWDATA;
    wire        M0HWRITE;

    //axi
    wire        m_axi_arready   ;
    wire        m_axi_awready   ;
    wire        m_axi_bvalid    ;
    wire        m_axi_rlast     ;
    wire        m_axi_rvalid    ;
    wire        m_axi_wready    ;
    wire [3:0]  m_axi_bid       ;
    wire [1:0]  m_axi_bresp     ;
    wire [31:0] m_axi_rdata     ;
    wire [3:0]  m_axi_rid       ;
    wire [1:0]  m_axi_rresp     ;
    wire        m_axi_aclk      ;
    wire        m_axi_aresetn   ;
    wire        m_axi_arlock    ;
    wire        m_axi_arvalid   ;
    wire        m_axi_awlock    ;
    wire        m_axi_awvalid   ;
```

```
    wire        m_axi_bready    ;
    wire        m_axi_rready    ;
    wire        m_axi_wlast     ;
    wire        m_axi_wvalid    ;
    wire [31:0] m_axi_araddr    ;
    wire [1:0]  m_axi_arburst   ;
    wire [3:0]  m_axi_arcache   ;
    wire [3:0]  m_axi_arid      ;
    wire [7:0]  m_axi_arlen     ;
    wire [2:0]  m_axi_arprot    ;
    wire [2:0]  m_axi_arsize    ;
    wire [31:0] m_axi_awaddr    ;
    wire [1:0]  m_axi_awburst   ;
    wire [3:0]  m_axi_awcache   ;
    wire [3:0]  m_axi_awid      ;
    wire [7:0]  m_axi_awlen     ;
    wire [2:0]  m_axi_awprot    ;
    wire [2:0]  m_axi_awsize    ;
    wire [31:0] m_axi_wdata     ;
    wire [3:0]  m_axi_wstrb     ;

//CM33 控制器
STAR_Processor STAR_Processor(
    .CIB_CLK(usr_clk),  //25M
    .MTX_RSTN(MTX_RSTN),
    .SWCLK(SWCLK),
    .SWDIO(SWDIO),
    .UART0_RX(UART0_RX),
    .UART0_TX(UART0_TX),
    .M0HADDR(M0HADDR),      //32
    .M0HBURST(M0HBURST),     //3
    .M0HMASTLOCK(M0HMASTLOCK), //output
    .M0HPROT(M0HPROT),      //3
    .M0HRDATA(M0HRDATA),     //32
    .M0HREADYMUX(M0HREADYMUX),   //output
    .M0HREADYOUT(M0HREADYOUT),
    .M0HSEL(M0HSEL),
    .M0HRESP(M0HRESP), //M0HRESP Slave 发给 Master 的总线传输状态,0：OKAY，传输完成
    .M0HSIZE(M0HSIZE),      //2
    .M0HTRANS(M0HTRANS),     //2
    .M0HWDATA(M0HWDATA),     //32
    .M0HWRITE(M0HWRITE)
);

//AHB 转 AXI
Bus_Bridge Bus_Bridge
(
```

```
    .s_ahb_hclk         (usr_clk),  //系统时钟 CIB_CLK
    .s_ahb_hready_in    (M0HREADYOUT),  //input 该信号为高时，表示总线上的传输完成
    .s_ahb_hresetn      (MTX_RSTN),  //复位信号，低电平有效
    .s_ahb_hsel         (M0HSEL),  //从机选通信号，该信号为高时表示本从机被选通
    .s_ahb_hwrite       (M0HWRITE),  //读写信号，该信号为高表示写操作，为低表示读操作
    .s_ahb_haddr        (M0HADDR            ),  //32 位地址总线
    .s_ahb_hburst       (M0HBURST           ),  //突发传输类型
    .s_ahb_hprot        (M0HPROT      ),  //保护控制信号，为总线访问提供附加的信息
    .s_ahb_hsize        (M0HSIZE),  //表示传输数据的大小
    .s_ahb_htrans       (M0HTRANS   ),  //表示当前传输的类型，如连续、不连续、空闲或忙
    .s_ahb_hwdata       (M0HWDATA           ),  //32 位写数据总线
    .s_ahb_hready_out   (M0HREADYOUT        ),
    .s_ahb_hresp        (M0HRESP            ),
    .s_ahb_hrdata       (M0HRDATA           ),
    .m_axi_arready      (m_axi_arready      ),  //
    .m_axi_awready      (m_axi_awready      ),
    .m_axi_bvalid       (m_axi_bvalid       ),
    .m_axi_rlast        (m_axi_rlast        ),
    .m_axi_rvalid       (m_axi_rvalid       ),
    .m_axi_wready       (m_axi_wready       ),
    .m_axi_bid          (m_axi_bid          ),
    .m_axi_bresp        (m_axi_bresp        ),
    .m_axi_rdata        (m_axi_rdata        ),
    .m_axi_rid          (m_axi_rid          ),
    .m_axi_rresp        (m_axi_rresp        ),
    .m_axi_aclk         (m_axi_aclk         ),//output
    .m_axi_aresetn      (m_axi_aresetn      ),//output
    .m_axi_arlock       (m_axi_arlock       ),
    .m_axi_arvalid      (m_axi_arvalid      ),
    .m_axi_awlock       (m_axi_awlock       ),
    .m_axi_awvalid      (m_axi_awvalid      ),
    .m_axi_bready       (m_axi_bready       ),
    .m_axi_rready       (m_axi_rready       ),
    .m_axi_wlast        (m_axi_wlast        ),
    .m_axi_wvalid       (m_axi_wvalid       ),
    .m_axi_araddr       (m_axi_araddr       ),
    .m_axi_arburst      (m_axi_arburst      ),
    .m_axi_arcache      (m_axi_arcache      ),
    .m_axi_arid         (m_axi_arid         ),
    .m_axi_arlen        (m_axi_arlen        ),
    .m_axi_arprot       (m_axi_arprot       ),
    .m_axi_arsize       (m_axi_arsize       ),
    .m_axi_awaddr       (m_axi_awaddr       ),
    .m_axi_awburst      (m_axi_awburst      ), //output
    .m_axi_awcache      (m_axi_awcache      ),
    .m_axi_awid         (m_axi_awid         ),
    .m_axi_awlen        (m_axi_awlen        ),
```

```
    .m_axi_awprot       (m_axi_awprot       ),
    .m_axi_awsize       (m_axi_awsize       ),
    .m_axi_wdata        (m_axi_wdata        ),
    .m_axi_wstrb        (m_axi_wstrb        )
);

PLL_SYS  U_PLL_SYS(
    .CLKI           (CIB_CLK    ),
    .CLKOP          (phy_clk    ), //ddr3_phyio_clk
    .CLKOS          (usr_clk    ), //user_clk
    .CLKOS2         (sys_clk    ),
    .LOCK           (pll_lock   )
);

assign led = init_status[1];
//DDR3 控制器
DDR_CONTROLLER  ddrc_top_axi_inst
(
    .stable_clkin           (sys_clk),//I 本地稳定时钟 f<1/2 ddr_work_clkin  50M
    .ddr_ss_rstn            (MTX_RSTN),//I 复位 DDR 子系统，包含 DDR
    .pll_lock               (pll_lock),//I 提供 ddr_work_clkin 的 PLL lock 标志
    .ddr_work_clkin         (phy_clk),// I DDR 工作时钟 Phy_io_clk  400m
    .ddr_data_clkout        (),//
    .init_status            (init_status),//3o init_status[1] = init_done;
    .training_report        (                   ),
    .usr_ar_trigger         (1'b0               ),
    .aclk                   (usr_clk            ),//I 100M
    .s_axi_awid             (m_axi_awid         ),
    .s_axi_awaddr           (m_axi_awaddr       ),
    .s_axi_awlen            (m_axi_awlen        ),
    .s_axi_awsize           (m_axi_awsize       ),
    .s_axi_awburst          (m_axi_arburst      ),
    .s_axi_awlock           (m_axi_awlock       ),
    .s_axi_awcache          (m_axi_awcache      ),
    .s_axi_awprot           (m_axi_awprot       ),
    .s_axi_awqos            (4'b0000            ),
    .s_axi_awvalid          (m_axi_awvalid      ),
    .s_axi_awready          (m_axi_awready      ),
    .s_axi_wdata            (m_axi_wdata        ),
    .s_axi_wstrb            (m_axi_wstrb        ),
    .s_axi_wlast            (m_axi_wlast        ),
    .s_axi_wvalid           (m_axi_wvalid       ),
    .s_axi_wready           (m_axi_wready       ),
    .s_axi_bid              (m_axi_bid          ),
    .s_axi_bresp            (m_axi_bresp        ),
    .s_axi_bvalid           (m_axi_bvalid       ),
    .s_axi_bready           (m_axi_bready       ),
```

```
    .s_axi_arid             (m_axi_arid         ),
    .s_axi_araddr           (m_axi_araddr       ),
    .s_axi_arlen            (m_axi_arlen        ),
    .s_axi_arsize           (m_axi_arsize       ),
    .s_axi_arburst          (m_axi_arburst      ),
    .s_axi_arlock           (m_axi_arlock       ),
    .s_axi_arcache          (m_axi_arcache      ),
    .s_axi_arprot           (m_axi_arprot       ),
    .s_axi_arqos            (4'b0000            ),
    .s_axi_arvalid          (m_axi_arvalid      ),
    .s_axi_arready          (m_axi_arready      ),
    .s_axi_rid              (m_axi_rid          ),
    .s_axi_rdata            (m_axi_rdata        ),
    .s_axi_rresp            (m_axi_rresp        ),
    .s_axi_rlast            (m_axi_rlast        ),
    .s_axi_rvalid           (m_axi_rvalid       ),
    .s_axi_rready           (m_axi_rready       ),
    // `ifndef DDR2
    .em_ddr_reset_n         (ddr3_reset_n),
    // `endif
    .em_ddr_dq              (ddr3_dq),
    .em_ddr_dqs             (ddr3_dqs),
    .em_ddr_clk             (ddr3_clk),
    .em_ddr_cke             (ddr3_cke),
    .em_ddr_ras_n           (ddr3_ras_n),
    .em_ddr_cas_n           (ddr3_cas_n),
    .em_ddr_we_n            (ddr3_we_n),
    .em_ddr_cs_n            (ddr3_cs_n),
    .em_ddr_odt             (ddr3_odt),
    .em_ddr_dm              (ddr3_dm),
    .em_ddr_ba              (ddr3_ba),
    .em_ddr_addr            (ddr3_addr)
);
endmodule
```

1.7 物理约束

对工程进行综合完成之后，进行物理约束，在软件左边的设计文件中找到物理约束，点击+号，进入约束编辑器。

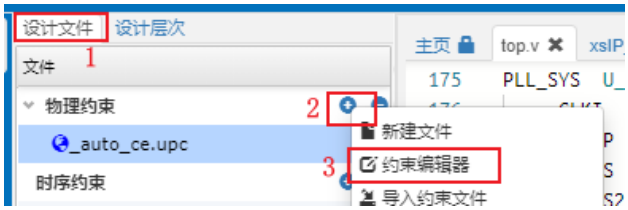
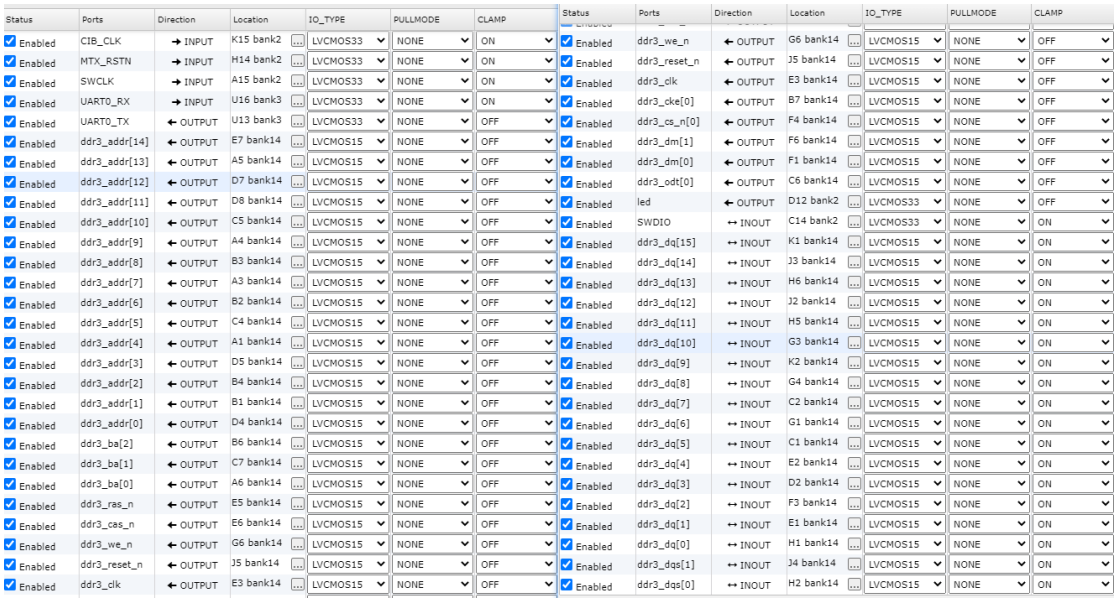


图 1-30 进入约束编辑器操作

上图所示已经有 upc 文件，没有 upc 文件的时候，按照上述方式进行创建，已有 upc 文件的工程，双击 upc 文件即可进入物理约束管理器，本次实验的约束如下所示。



Status	Ports	Direction	Location	IO_TYPE	PULLMODE	CLAMP
Enabled	CIB_CLK	INPUT	K15 bank2	LVCMOS33	NONE	ON
Enabled	MTX_RSTN	INPUT	H14 bank2	LVCMOS33	NONE	ON
Enabled	SWCLK	INPUT	A15 bank2	LVCMOS33	NONE	ON
Enabled	UART0_RX	INPUT	U16 bank3	LVCMOS33	NONE	ON
Enabled	UART0_TX	OUTPUT	U13 bank3	LVCMOS33	NONE	OFF
Enabled	ddr3_addr[14]	OUTPUT	E7 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[13]	OUTPUT	A5 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[12]	OUTPUT	D7 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[11]	OUTPUT	D8 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[10]	OUTPUT	C5 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[9]	OUTPUT	A4 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[8]	OUTPUT	B3 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[7]	OUTPUT	A3 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[6]	OUTPUT	B2 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[5]	OUTPUT	C4 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[4]	OUTPUT	A1 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[3]	OUTPUT	D5 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[2]	OUTPUT	B4 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[1]	OUTPUT	B1 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_addr[0]	OUTPUT	D4 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_ba[2]	OUTPUT	B6 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_ba[1]	OUTPUT	C7 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_ba[0]	OUTPUT	A6 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_ras_n	OUTPUT	E5 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_cas_n	OUTPUT	E6 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_we_n	OUTPUT	G6 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_reset_n	OUTPUT	J5 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_clk	OUTPUT	E3 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_we_n	OUTPUT	G6 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_reset_n	OUTPUT	J5 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_clk	OUTPUT	E3 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_cke[0]	OUTPUT	B7 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_cs_n[0]	OUTPUT	F4 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_dm[1]	OUTPUT	F6 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_dm[0]	OUTPUT	F1 bank14	LVCMOS15	NONE	OFF
Enabled	ddr3_odt[0]	OUTPUT	C6 bank14	LVCMOS15	NONE	OFF
Enabled	led	OUTPUT	D12 bank2	LVCMOS33	NONE	OFF
Enabled	SWDIO	INOUT	C14 bank2	LVCMOS33	NONE	ON
Enabled	ddr3_dq[15]	INOUT	K1 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[14]	INOUT	J3 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[13]	INOUT	H6 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[12]	INOUT	J2 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[11]	INOUT	H5 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[10]	INOUT	G3 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[9]	INOUT	K2 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[8]	INOUT	G4 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[7]	INOUT	C2 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[6]	INOUT	G1 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[5]	INOUT	C1 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[4]	INOUT	E2 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[3]	INOUT	D2 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[2]	INOUT	F3 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[1]	INOUT	E1 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dq[0]	INOUT	H1 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dqs[1]	INOUT	J4 bank14	LVCMOS15	NONE	ON
Enabled	ddr3_dqs[0]	INOUT	H2 bank14	LVCMOS15	NONE	ON

图 1-31 管脚约束示意图

然后设置 IO 电压，将 DDR 所在 bank14 的电压设置为 1.5V，如下所示。

图 1-32 设置 bank 电压

这里特别说明一点，关于 SW 调试接口的引脚分配，这里的引脚并不是固定的，可以自己根据需要进行修改，我们设置的是扩展 GPIO0 口上 GPIO0_0 和 GPIO0_1，分别对应 SWCLK 和 SWDIO。

1.8 MDK 工程建立

我们 MDK 中实现的功能是：循环写入 5000 个数据，然后读取数据，将读取的数据和写入的数据进行对比，对错误数据进行统计，如果没错则串口打印“Verification passed: All data matched successfully”，否则打印出错误个数，代

码如下所示：

```
//通过 AHB 写操作函数
static void ahbslave_write(uint32_t *address, uint32_t data)
{
    *address = data;
}
//通过 AHB 读操作函数
static uint32_t ahbslave_read(uint32_t address)
{
    uint32_t *data;
    data = (unsigned int *)address;
    return *data;
}
int main(void)
{
    uint32_t ts = 0;
    uint32_t rd = 0;
    uint32_t rData = 0;
    uint32_t errorCount = 0; // 新增：错误计数器

    uart_init(115200);

    // 写入数据
    for(ts=0;ts<50000;ts++){
        ahbslave_write(STAR_TARGEXP0_BASE+4*ts, ts);
        //printf("Write: Address:%d, Data: %d\r\n", 4*ts, ts);
    }

    // 读取数据并验证
    for(rd=0;rd<50000;rd++){
        rData = ahbslave_read(STAR_TARGEXP0_BASE+4*rd);
        //printf("Read: Address:%d, Data: %d\r\n", 4*rd, rData);

        // 新增：验证读写数据是否一致
        if(rData != rd) {
            printf("ERROR: Address:%d - Written:%d, Read:%d\r\n",
                   4*rd, rd, rData);
            errorCount++;
        }
    }

    // 新增：输出错误统计
    if(errorCount > 0) {
        printf("Verification failed: %d errors detected\r\n", errorCount);
    } else {
        printf("Verification passed: All data matched successfully\r\n");
    }
```

```
    return errorCount; // 返回错误数量，非零表示有错误
}
```

1.9 板级验证

1.9.1 实验所需硬件

（1）AC201-SA5Z-50D0 开发板

（2）FPGA 下载器：XIST USB Cable

（3）CM3 仿真器：DAP Link

（4）电源线

（5）Type-C 线

1.9.2 硬件连接

本次实验的硬件连接如下所示。

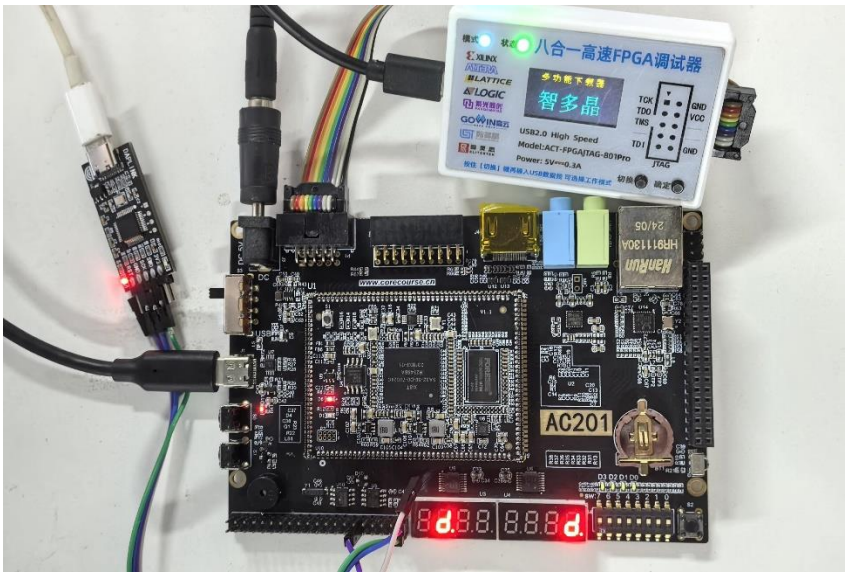


图 1-33 硬件连接

1.9.3 下载文件至目标板

下载文件的方式总共分为两种，一种是分别下载硬件设计代码和软件设计代码，当我们需要调试软件程序代码的时候，这种方式比较适合；另外一种是合并成一个 bin 文件进行下载，这种方式适合在代码已经测试没有问题，需要量产的时候使用。

1.9.3.1 分别下载

1. 下载硬件设计文件，硬件测试文件在 HQ 工程目录下的 hq_run 文件夹下，具体操作方式如下所示。

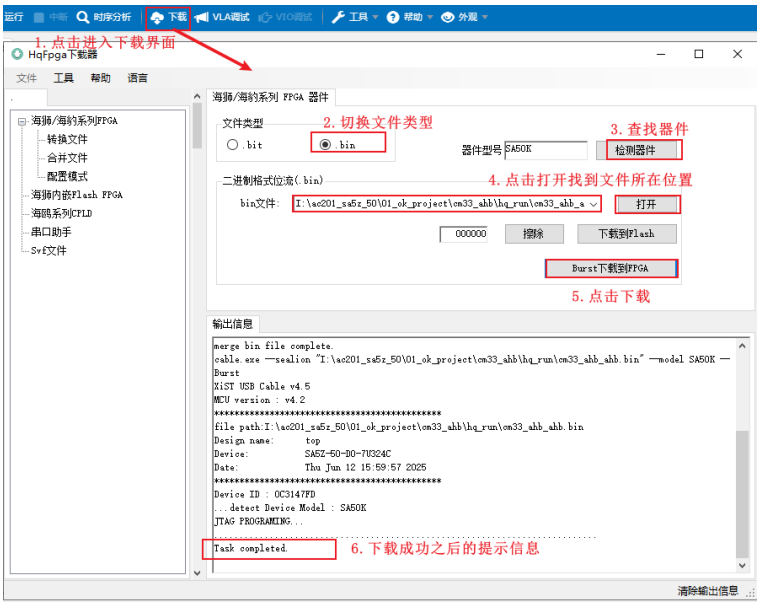


图 1-34 下载硬件设计文件

2. 下载软件设计文件，操作方式如下所示。

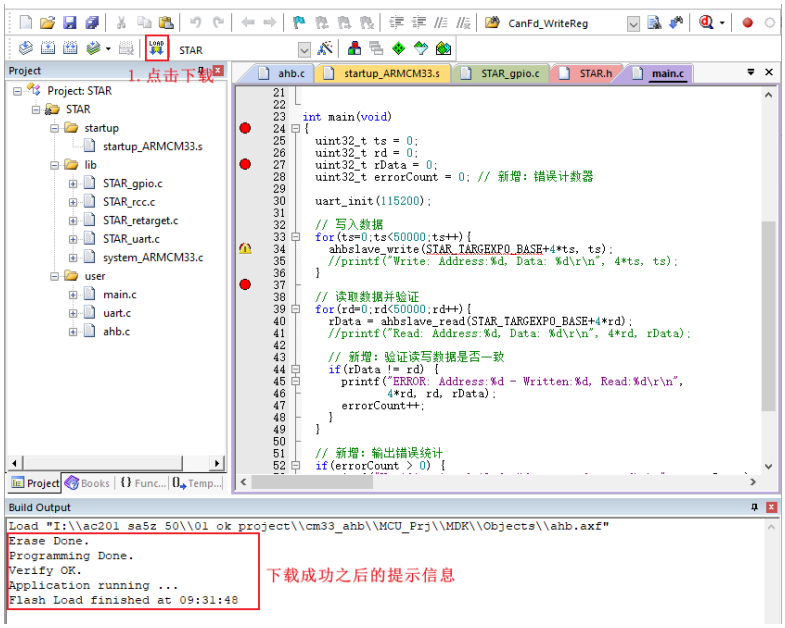


图 1-35 下载软件设计文件

1.9.3.2 合并下载

将 FPGA 的 bin 文件和 CM3 的 bin 文件合成一个 bin 文件下载至开发板，合并文件操作步骤如下所示。

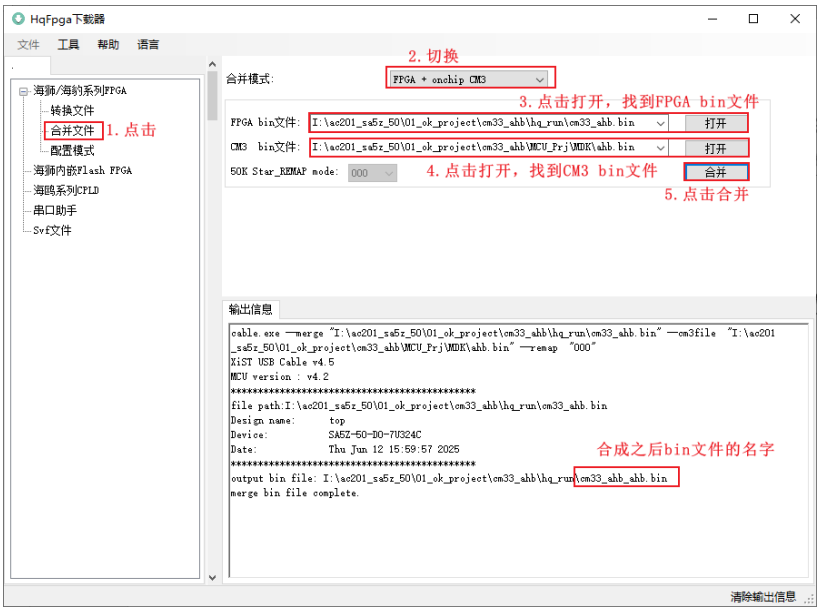


图 1-36 合并文件操作示意图

合并成功之后进行下载，合并之后的文件同样也存放在 hq_run 文件下，下载步骤如下图所示。

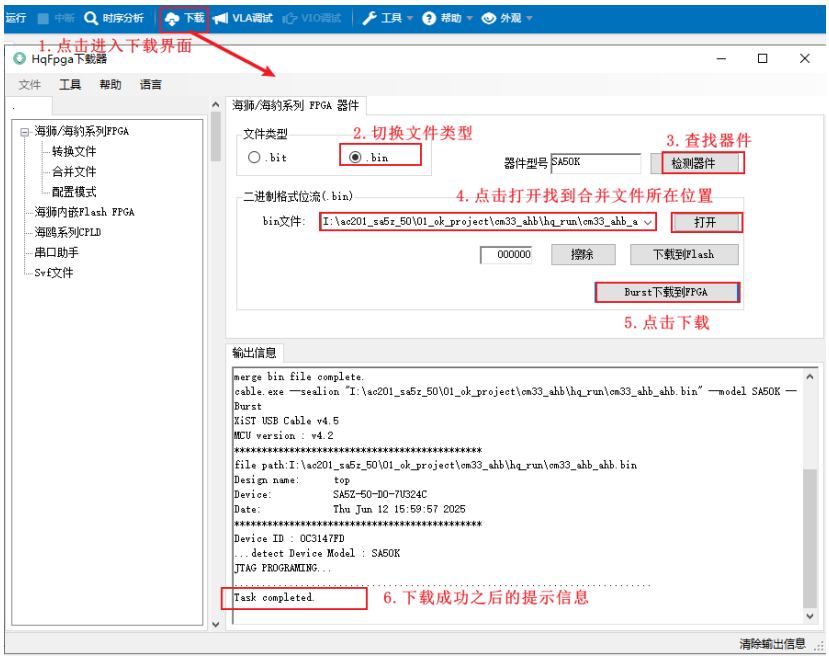


图 1-37 合并文件下载步骤示意图

1.9.4 功能演示

打开串口软件，波特率设置为 115200，如下所示。

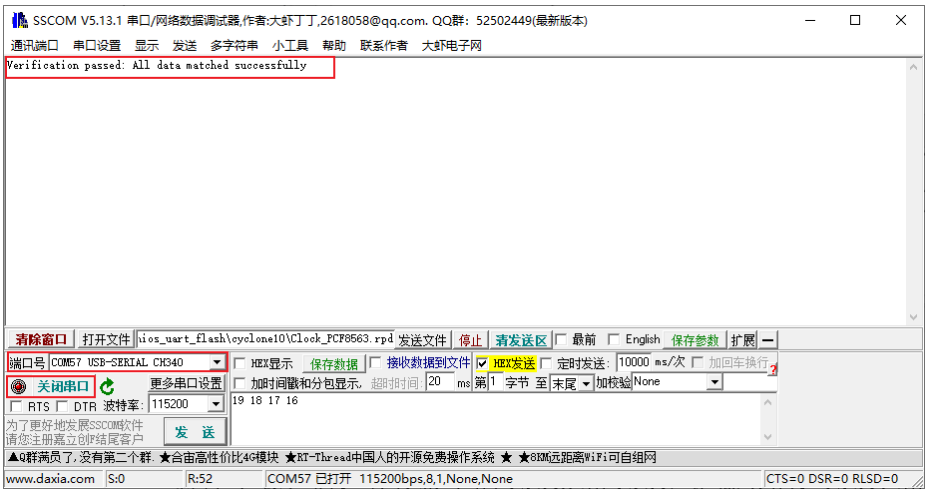


图 1-38 串口打印相关信息

从上图可以看出，程序下载完成之后，串口打印“Verification passed: All data matched successfully”，说明读写数据一致。