

1 基于 DDR3 的串口传图帧缓存系统设计实现 (HDMI 和 TFT 显示)

工程源码	----02_设计实例 ----- acz702_uart_hp_ddr3_tft800x480_hdmi.zip
相关视频课程	
	本次设计需要使用到 EDA 拓展板，如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

本章将基于前面讲解的设计内容，继续讲解串口传图的基本内容。本章在 FPGA 学习内容中，处于立交桥的地位。本章的内容全面涵盖输入、缓存、输出，同时本章的内容又是基于 FPGA 片上图片缓存的内容深化。学完本章以后，既可以基于本章内容实现各种类型千变万化输入输出接口的替换设计，又可以在本章的数据接收与处理环节作文章，进行图像处理内容的理解与学习。

在学习本章之前，建议读者先复习串口接收模块、disp_driver 模块、基于 HDMI 和 TFT 显示屏的图片显示相关内容，同时，对 AXI 接口转换模块章节进行理解强化。

1.1 系统整体设计

在上一章中，我们已经完成了 AXI 转换接口模块的设计，通过该模块，我们只需操作 FIFO 便能完成对 DDR3 的读写操作。本章，我们趁热打铁，基于该模块，结合前面章节学习的串口接收模块和 tft 显示屏控制、SiI9022 初始化模块，设计一个基于 DDR3 的串口传图帧缓存系统。系统将会通过 EDA 拓展板上的串口接收电脑传输的 800*480 分辨率的 RGB565 图像数据，图像数据会被存储在 DDR3 中，在需要时，数据会被读出，用于在 TFT 和 HDMI 上显示。

本次设计系统的整体设计框图如图 1-1 所示：

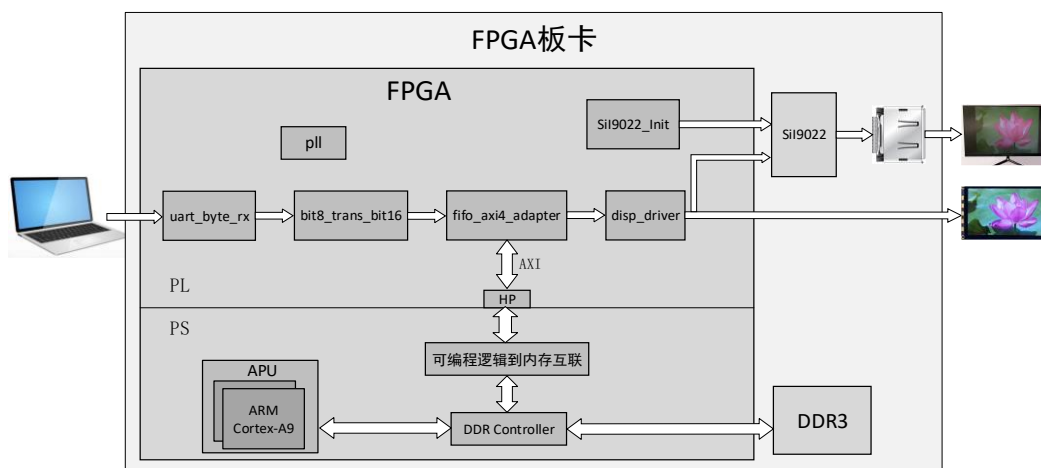


图 1-1 设计框图

其中，PS 侧是现成的硬件资源，但是要想使用这些资源，我们需要通过 ZYNQ 软核进行使能配置。PL 侧为需要用户设计的模块，这些模块绝大多数都在前面章节中有过讲解介绍，这里简单说明下它们的功能：

- (1) `uart_byte_rx` 模块:负责串口图像数据的接收。
- (2) `bit8_trans_bit16` 模块: 将串口接收的每两个 8bit 数据转换成一个 16bit 数据 (图像数据是 16bit 的 RGB565 的数据, 电脑是通过串口将一个像素点数据分两次发送到 FPGA, FPGA 需将串口接收数据重组成 16bit 的图像数据), 实现过程相对比较简单。
- (3) `disp_driver` 模块: tft 屏显示驱动控制, 对缓存在 DDR3 中的图像数据进行显示。
- (4) `fifo_axi4_adapter` 模块: 主要是用于接口的转换, 将 AXI4 接口换成与 FIFO 对接的接口。
- (5) `SiI9022_Init` 模块: 初始化并配置 SiI9022, 使芯片根据配置对输入的像素数据以及时序信号进行编码转换, 然后以符合 HDMI 接口的格式输出。
- (6) `pll`: 为系统提供 33MHz 和 100MHz 时钟。

如此, 我们便对整个系统有了一个简单的了解, 明白了系统的整体架构、各个模块的功能、数据的流向等等。接下来, 我们就基于该结构, 开始我们的设计。

1.2 系统工程搭建

本次设计由于涉及到 PS 与 PL，所以设计会分为两个部分。其中 PL 部分在 Vivado 软件中完成，用户需要构建本次设计的硬件逻辑系统，生成并导出用于 PS 端设计的硬件资源描述文件；PS 部分在 SDK 软件中完成，本次设计中主要是获取 PS 端配置脚本和相关文件，以及完成设计烧录工作。

首先创建一个名为 `uart_hp_ddr3_tft800x480_hdmi` 的 Vivado 工程，关于如何创建工程，如何选择芯片型号这里就不给大家赘述了。

工程创建好后，根据系统整体结构图，开始创建各个模块。其中，`fifo_axi4_adapter`、`disp_driver`、`SiI9022_Init` 在前面章节中都有过介绍，本次设计中我们直接使用即可，用户可以参考前面章节添加模块并创建模块所需的 IP 核即可。这里仅对 `bit8_trans_bit16`、`uart_byte_rx`、PLL 三个模块作简单介绍，并着重介绍如何通过 ZYNQ 核配置 PS 端相关硬件资源。

1.2.1 `uart_byte_rx` 模块

`uart_byte_rx` 模块负责接收通过串口传输过来的 800*480 分辨率的 RGB565 格式数据。因此，一帧图像包含 $800*480*16=6144000\text{bit}$ 数据。在前面章节的串口接收模块设计中，支持的最大波特率为 115200，在该波特率下传输一帧图像就需要 $6144000*10/8/115200 = 66.667\text{s}$ 。该波特率下串口传输一帧图像时间过长，将会导致整个系统效率十分低下。

在本次设计中，我们对 `uart_byte_rx` 模块进行修改，增加了一个自定义的 1562500 波特率。在该波特率下，串口接收一帧图像只需要 $6144000*10/8/1562500=4.9152\text{s}$ 。这里依旧使用一个选择器，来实现不同波特率与采样时钟分频计数值之间的对应关系。设计代码如下：

```
reg [8:0] Bps_DR;
parameter CLK_FRQ = 50000000;
always@(*)
    case(Baud_Set)
        0: Bps_DR = CLK_FRQ/9600/16 - 1;
        1: Bps_DR = CLK_FRQ/19200/16 - 1;
        2: Bps_DR = CLK_FRQ/38400/16 - 1;
        3: Bps_DR = CLK_FRQ/57600/16 - 1;
        4: Bps_DR = CLK_FRQ/115200/16 - 1;
        5: Bps_DR = CLK_FRQ/1562500/16 - 1;
        default: Bps_DR = CLK_FRQ/9600/16 - 1;
    endcase
```

1.2.2 bit8_trans_bit16 模块

bit8_trans_bit16 该模块主要功能是将串口传输过来（一次传 8bit）的每两个 8bit 数据拼接成一个 16bit 图像数据（RGB565）。功能相对比较简单。该模块的接口图及接口功能描述如下。

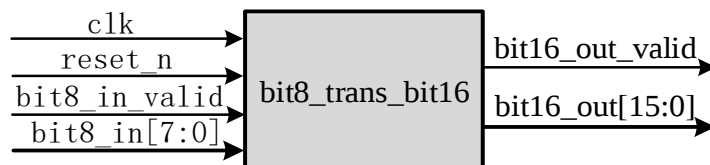


图 1-2 bit8_trans_bit16 模块接口图

表 1-1 bit8_trans_bit16 模块接口功能描述

接口名称	I/O	功能描述
clk	I	模块工作时钟
reset_n	I	模块复位，低有效
bit8_in[7:0]	I	8bit 数据输入
bit8_in_valid	I	8bit 数据有效标识
bit16_out[15:0]	O	转换后 16bit 数据输出
bit16_out_valid	O	转换后 16bit 数据有效标识

模块主要信号设计的时序要求如图 1-3。对输入的每两个 8bit 数据进行拼接，并产生一个拼接后数据输出有效标识信号。拼接后的数据是前一个数据在高字节位置，后一个数据在低字节位置。这个主要是串口传图上位机软件发送是先发送 16bit 图像数据的高字节，后发低字节，FPGA 上这样处理是为了保证拼接后图像数据的正确性。

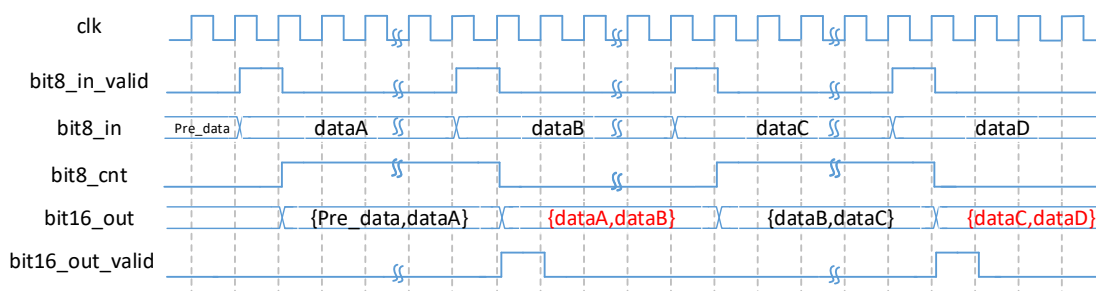


图 1-3 位宽转换模块时序图

有了时序图，代码设计就比较简单，具体代码如下。

```
module bit8_trans_bit16(  
    input          clk,  
    input          reset_p,  
  
    input          [7:0] bit8_in,
```

```
input          bit8_in_valid,

output reg [15:0] bit16_out,
output reg      bit16_out_valid
);

reg bit8_cnt;

always@(posedge clk or posedge reset_p)
begin
    if(reset_p)
        bit8_cnt <= 1'b0;
    else if(bit8_in_valid)
        bit8_cnt <= bit8_cnt + 1'b1;
    else
        bit8_cnt <= bit8_cnt;
end

always@(posedge clk or posedge reset_p)
begin
    if(reset_p)
        bit16_out <= 16'h0000;
    else if(bit8_in_valid)
        bit16_out <= {bit16_out[7:0], bit8_in};
    else
        bit16_out <= bit16_out;
end

always@(posedge clk or posedge reset_p)
begin
    if(reset_p)
        bit16_out_valid <= 1'b0;
    else if(bit8_in_valid && bit8_cnt)
        bit16_out_valid <= 1'b1;
    else
        bit16_out_valid <= 1'b0;
end

endmodule
```

1.2.3 PLL

PLL 用于产生模块所需时钟，对于设计来说，需要 33MHz、50MHz、100MHz 三种时钟。其中，33MHz 时钟用于 TFT 屏显示，50MHz 时钟用于 SiI9022 模块的初始化，100MHz 时钟用于 fifo_axi4_adapter 模块（AXI 接口常用时钟为 100MHz）、uart_byte_rx 模块、bit8_trans_bit16 模块。

店铺：<https://xiaomeige.taobao.com>

技术博客：<http://www.cnblogs.com/xiaomeige/>

官方网站：www.corecourse.cn

技术群组：

在 IP Catalog 中搜索 Clock，其中的 Clocking Wizard 便是我们要找的 IP。双击 IP 核进入配置界面，本次设计 PLL 的配置如图 1-4 和图 1-5 所示：

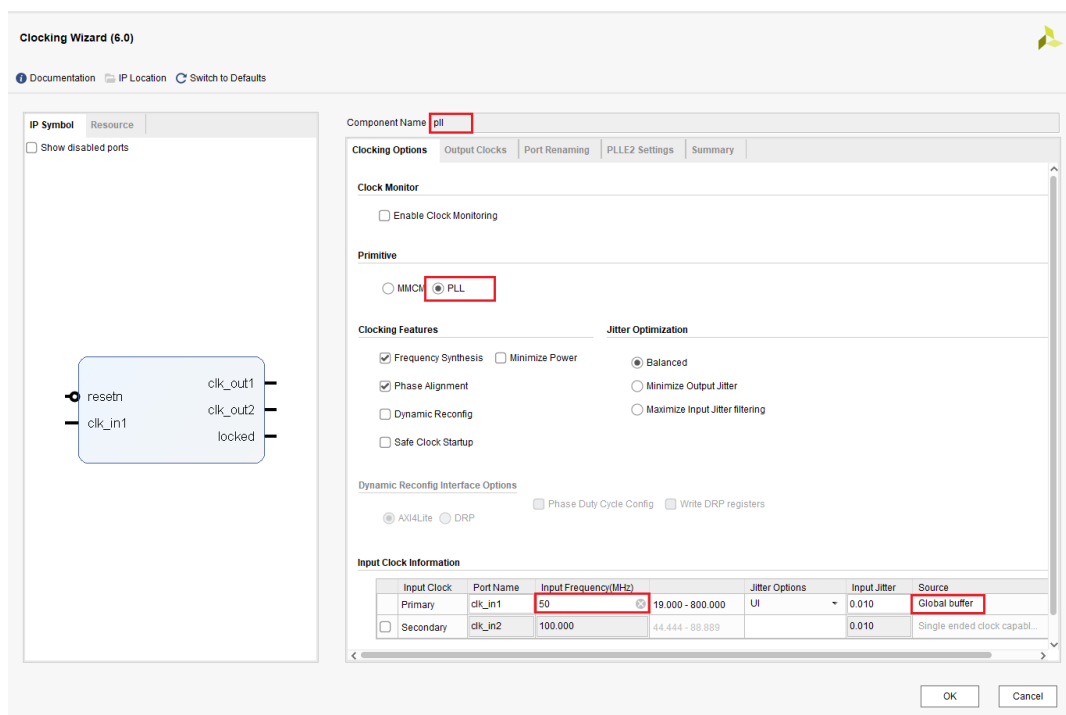


图 1-4 配置 PLL（其一）

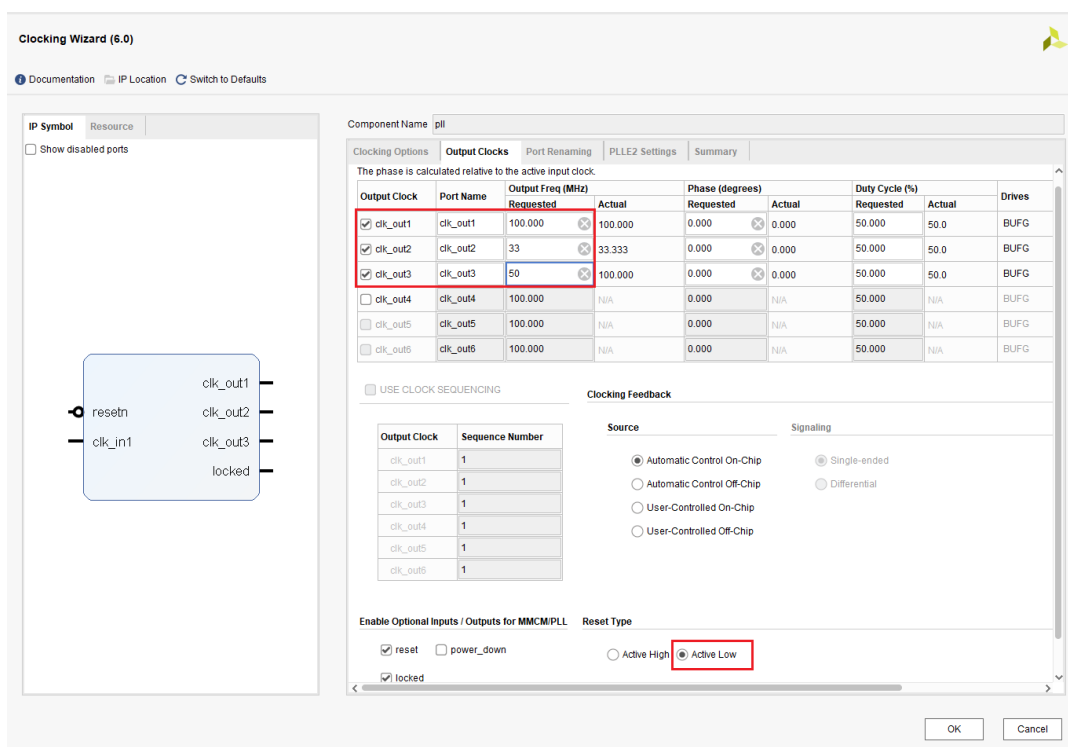


图 1-5 配置 PLL（其二）

配置完成后点击 OK，在弹出的窗口选择 Generate，对 IP 核生成输出，这样

店铺：<https://xiaomeige.taobao.com> 官方网站：www.corecourse.cn
技术博客：<http://www.cnblogs.com/xiaomeige/> 技术群组：

我们便成功添加了 PLL。

1.2.4 Block Design 与 ZYNQ 核

ZYNQ 核的全称为 ZYNQ7 Processing System，是围绕 Zynq-7000 处理系统的软件接口。该 IP 核充当 PS 和 PL 之间的逻辑连接，本次设计对 PS 端相关硬件资源的配置便需要通过该 IP 核完成。

在工程中添加并配置 ZYNQ 核的方式有两种，一种是通过 IP Catalog 创建，另一种则是通过 IP INTEGRATOR 的 Block Design 创建。考虑到设计的直观性，为了使设计更易理解，接下来我们通过创建 Block Design 的方式来为设计添加并配置 ZYNQ 核。

1.2.4.1 创建 Block Design

点击 IP INTEGRATOR 下的 Create Block Design 新建一个模块设计，在弹出的窗口中为模块设计命名为 system，如图 1-6 所示：

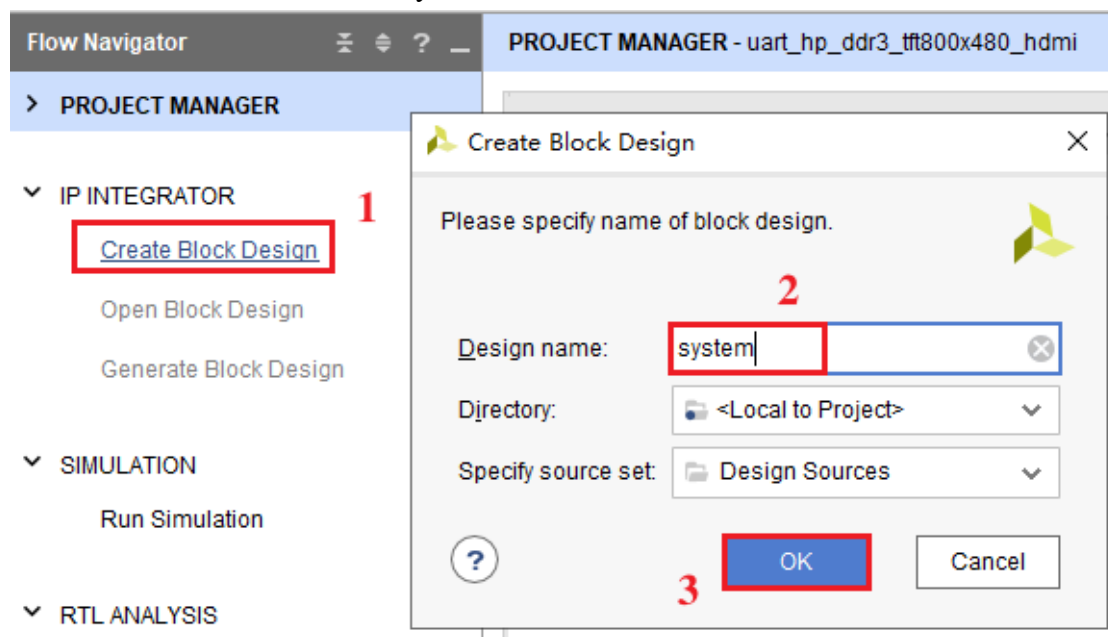


图 1-6 创建模块设计

创建好的 Block Design 界面如图 1-7 所示，左侧为工程资源窗口，能够查看工程设计结构、模块资源、接口信号等。右侧为视图窗口，用户添加的 IP 会直接展示在该窗口中，用户通过单击“+”为设计添加 IP。

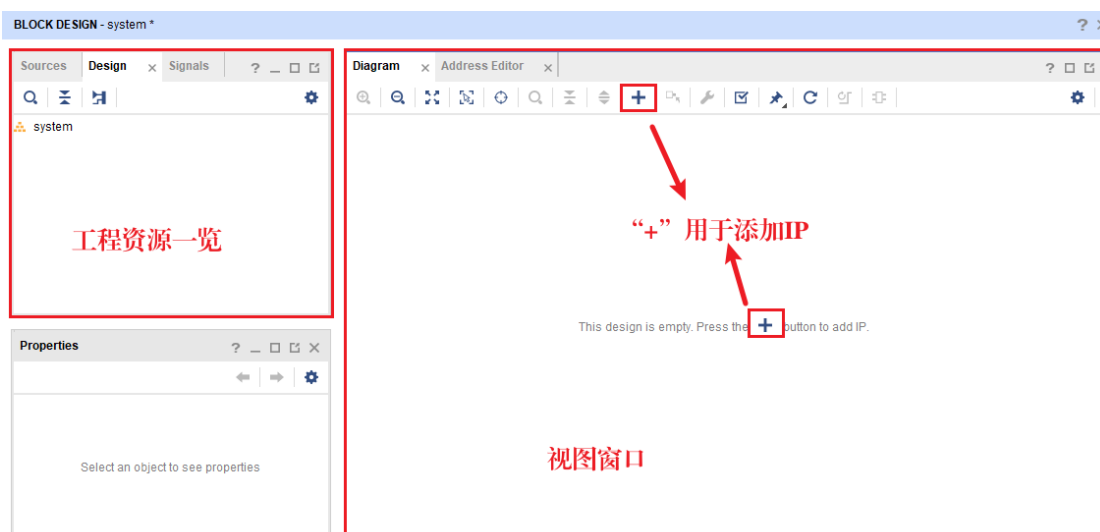


图 1-7 Block Design 界面

1.2.4.2 添加并配置 ZYNQ 核

点击“+”，在弹出的窗口中搜索 ZYNQ，会出现图 1-8 所示名为 ZYNQ7 Processing System 的 IP 核，双击添加 IP。

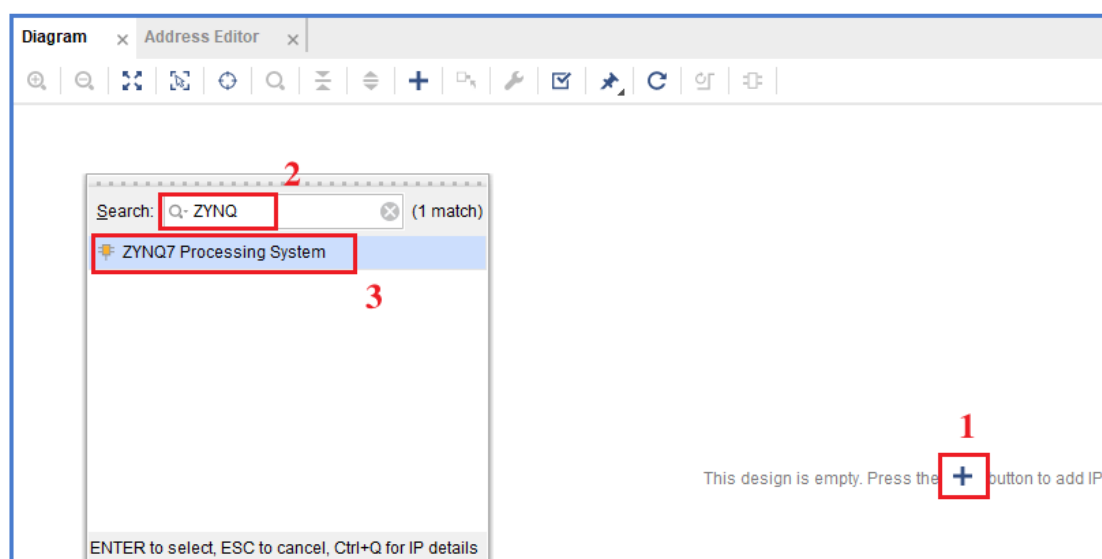


图 1-8 添加 ZYNQ 核

添加完成后的 ZYNQ 核如图 1-9 所示：

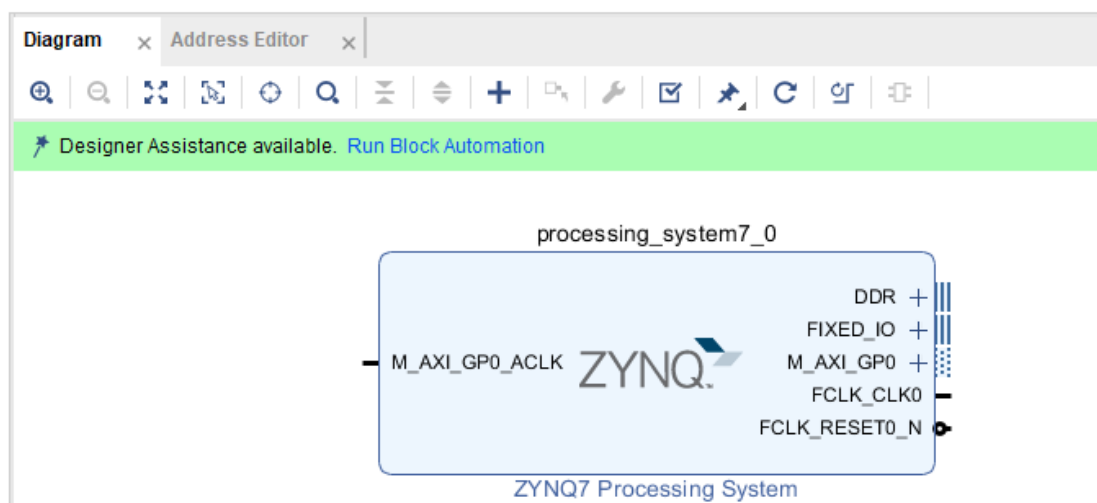


图 1-9 ZYNQ 核

此时的 ZYNQ 核还未配置，双击 IP 核进入到配置界面，如图 1-10 所示：

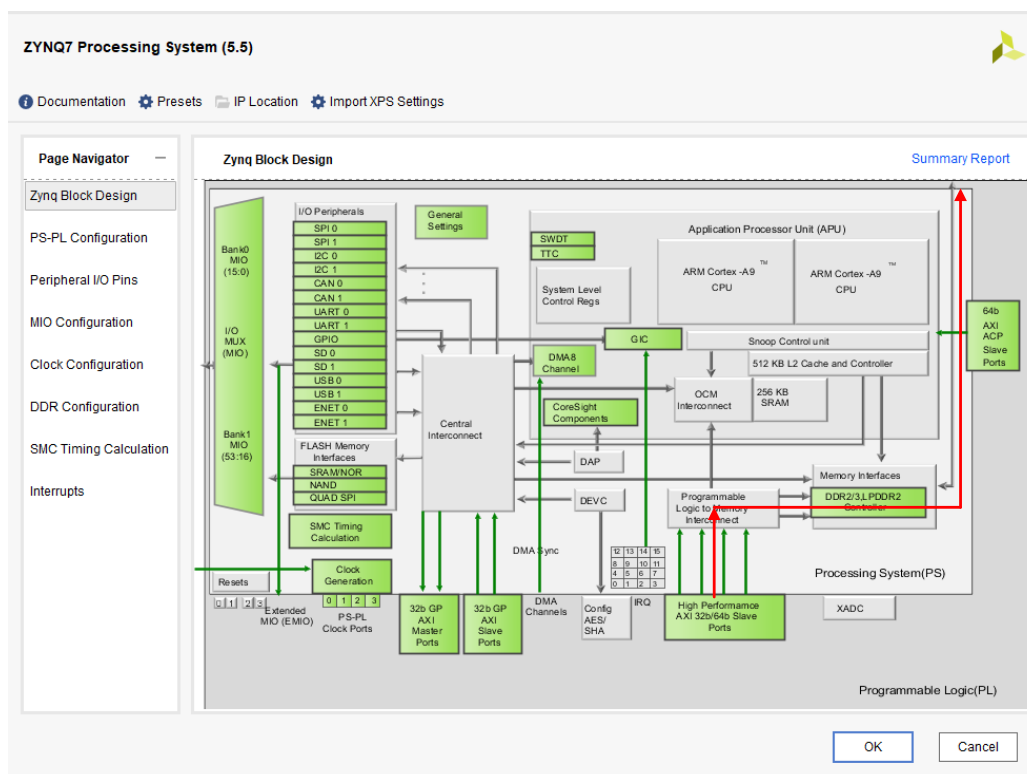


图 1-10 ZYNQ 核配置界面

本次设计我们需要借助 HP 接口来实现数据写入到 PS 侧的 DDR3，如图 1-10 中红色线条。点击图中绿色的 High Performance AXI 32b/64b Slave Ports 实现快速跳转，如图 1-11 所示：

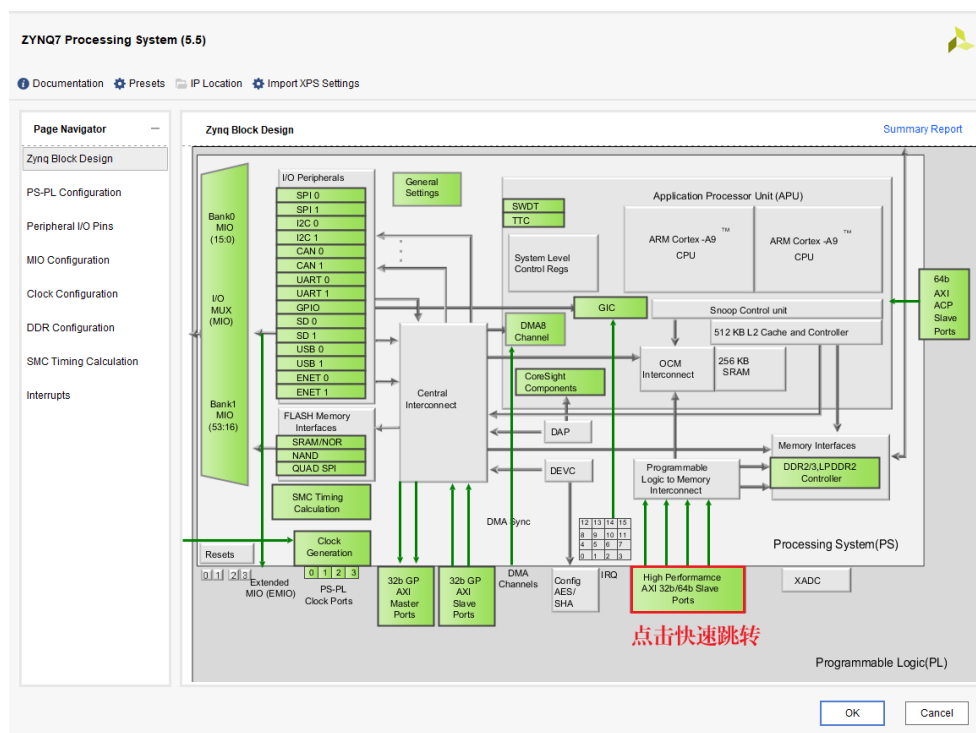


图 1-11 快速跳转

在跳转后的界面中勾选 S AXI HP0 interface，如图 1-12 所示：

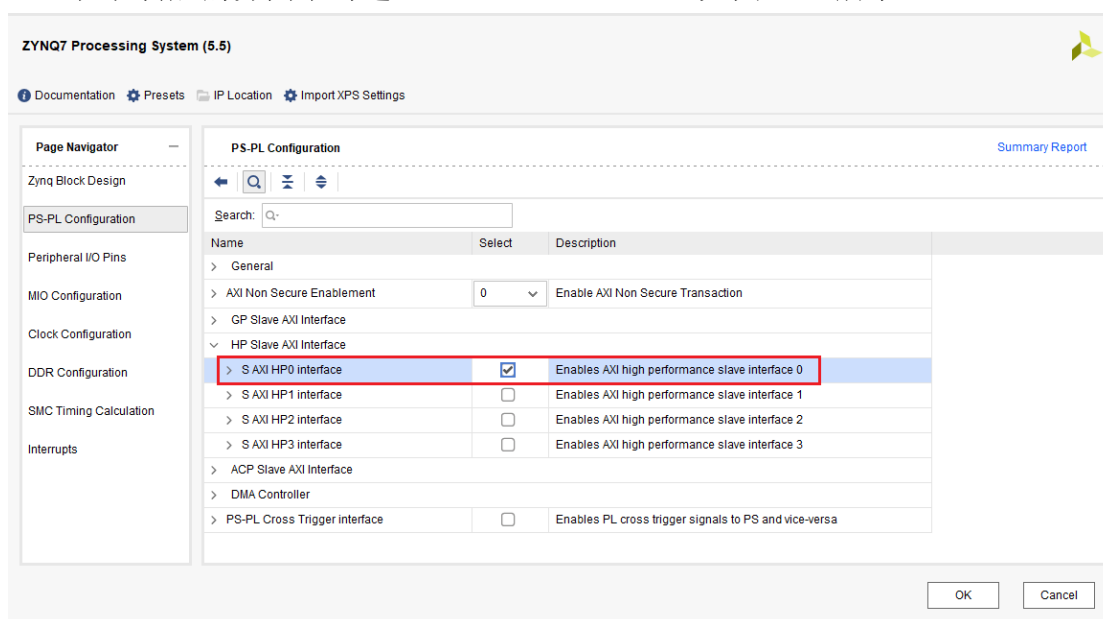


图 1-12 使能 HP0 接口

由于设计需要使用到 DDR3，接下来我们还需要配置 DDR 控制器。点击 DDR Configuration 跳转到对应界面，这里我们直接配置 DDR 型号为“MT41K128M16 JT-125”，如图 1-13 所示：

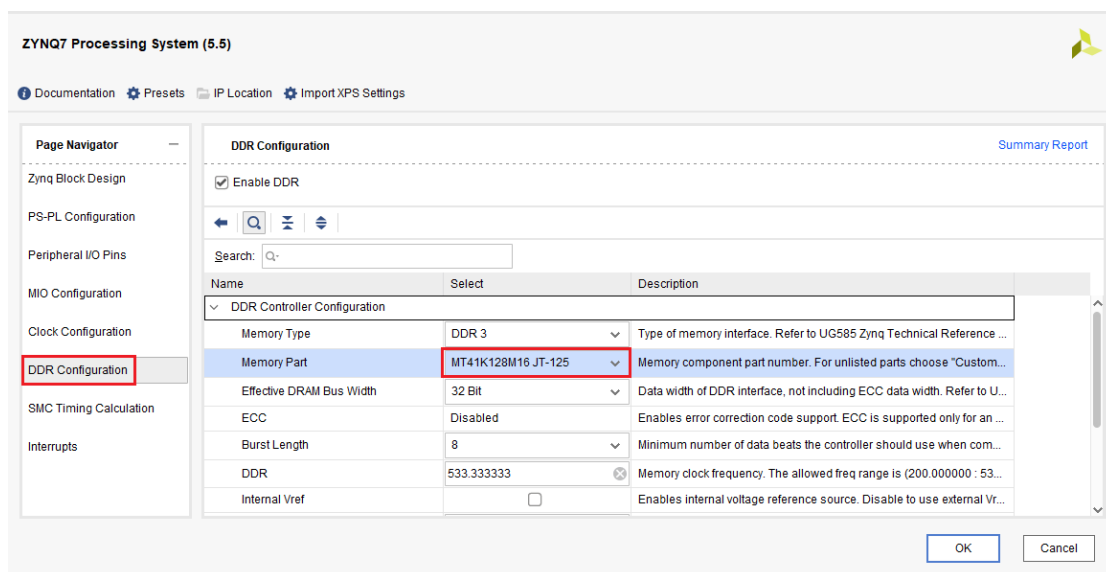


图 1-13 配置 DDR 型号

接着打开 Clock Configuration 页，勾选 PL Fabric Clocks 项下的 FCLK_CLK0 (默认勾选)，如图 1-14 所示：

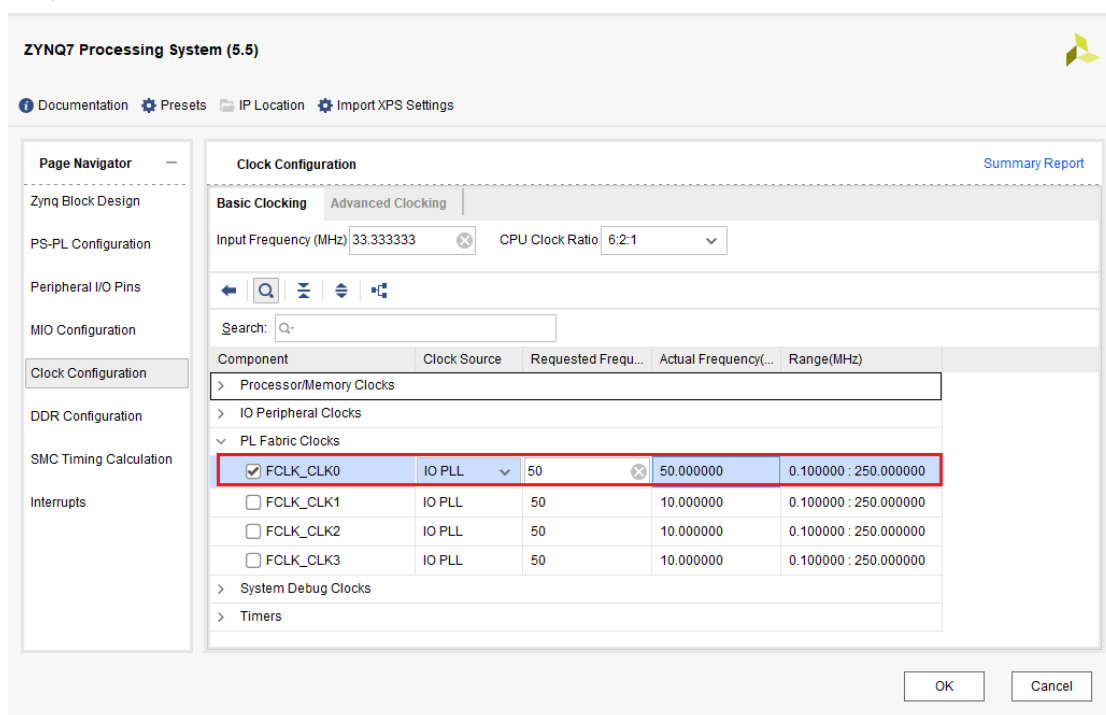


图 1-14 使能 PS 端输出时钟

该接口为 PS 端时钟输出接口，用来为 PL 提供时钟，这里我们保持默认的 50MHz 即可。关于该时钟如何实现，我们可以在 Advanced Clocking 中看到，计算方法如图 1-15 所示：

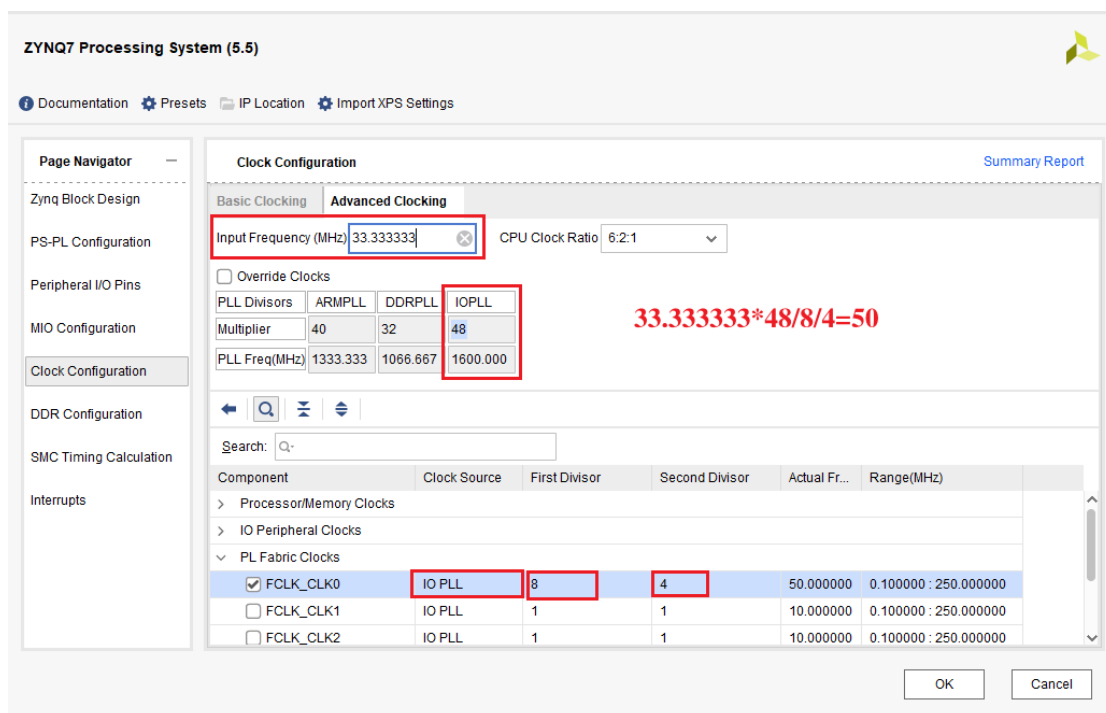


图 1-15 输出时钟分频与倍频系数

除了以上这些配置外，ZYNQ 核默认还为我们使能了一些接口，为了减少不必要的麻烦，这里我们需要取消其中 M AXI GP0 接口的使能，如图 1-16 所示：

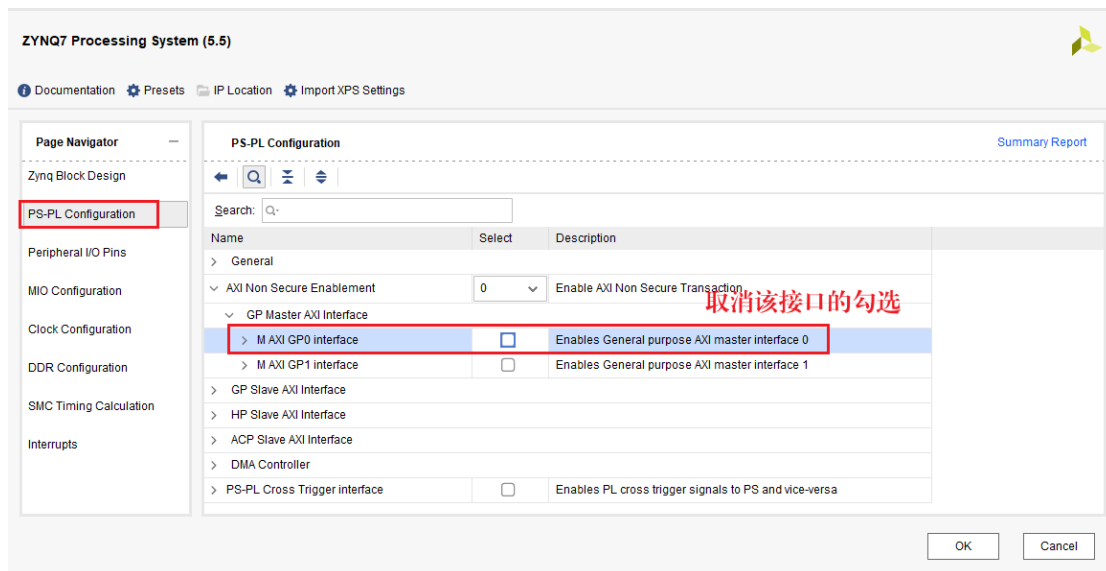


图 1-16 取消 M AXI GP0 接口

至此，我们便完成了对 ZYNQ 核的配置，点击 OK，配置完成后的 ZYNQ 核如图 1-17 所示：



图 1-17 配置完成后 ZYNQ 核

1.2.4.3 添加并配置互联及复位 IP

在本次设计中，PL 侧的数据在经过 fifo_axi4_adapter 模块转换成 AXI4 接口协议后通过 HP0 接口写入 PS 侧。但是，HP 接口使用的 AXI3 协议（双击 HP0 接口即可看到其接口协议），二者之间由于所使用的协议不同，不能直接连接。对于这种情况，就需要添加一个 AXI Interconnect 核，用以实现协议之间的转换。

使用快捷键 ctrl+i 搜索并添加 AXI Interconnect 核，随后配置 IP 的主从接口数为 1，其余项保持默认即可，如图 1-18 所示：

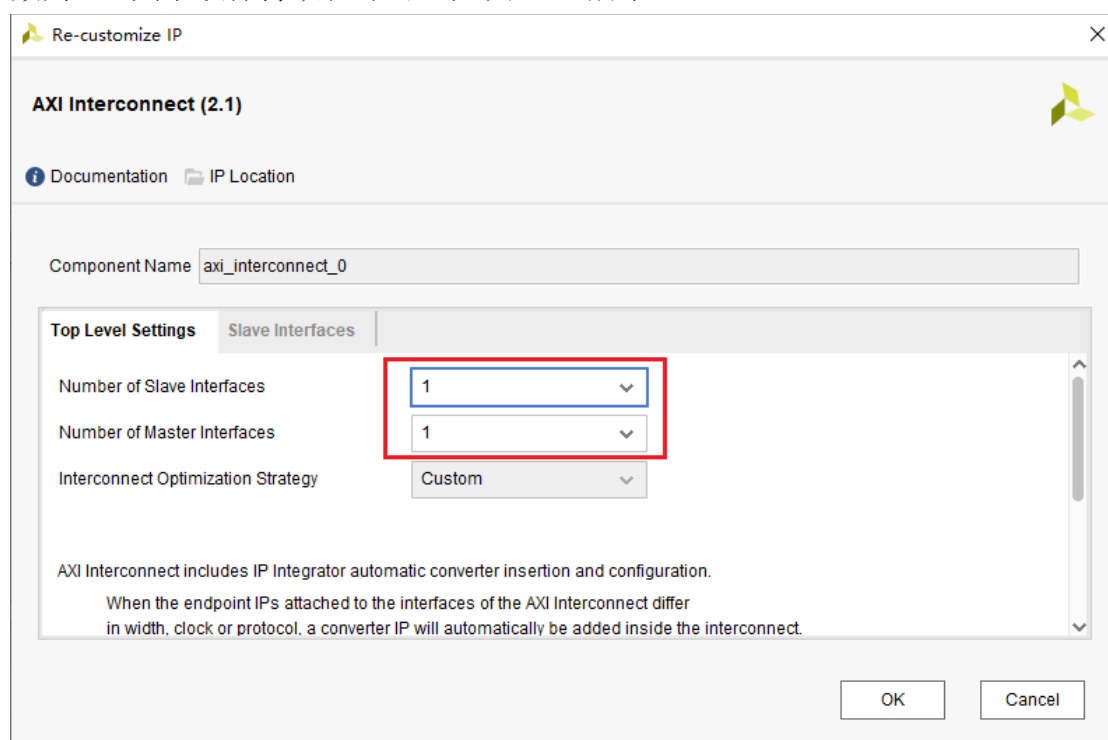


图 1-18 配置 AXI Interconnect 核

配置完成后的 AXI Interconnect 核如图 1-19 所示：

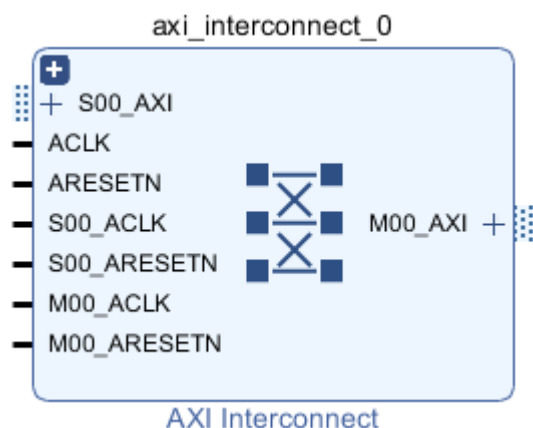


图 1-19 配置完成后的 AXI Interconnect 核

除了互联之外，复位信号也同样重要，xilinx 中有提供专门用于管理复位的 IP 核 Processor System Reset。ctrl+i 搜索并添加该 IP，本次设计中，该 IP 核无需配置，直接使用即可，添加完成后的 IP 如图 1-20 所示：

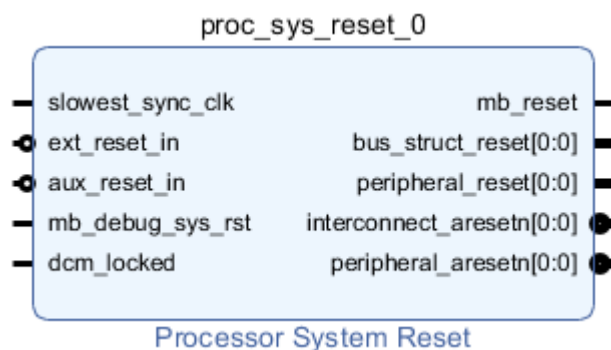


图 1-20 Processor System Reset 核

1.2.4.4 端口连接

对于设计来说，我们创建的 Block Design 就是一个模块，而我们添加在其中的 IP 核则是这个模块中的一个子模块。为了让模块能够正常的工作，在配置完子模块后，就需要将它们彼此之间的接口联系起来，具体操作步骤如下：

1. 选中 ZYNQ 核的 FCLK_RESET0_N 接口，右键选择 Make External 导出该接口。该接口将作为 PS 侧输出给 PL 的复位信号，为了方便区分，在左侧的 External Port Properties 中将端口重命名为 ps2pl_resetn_0。操作步骤参考图 1-21：

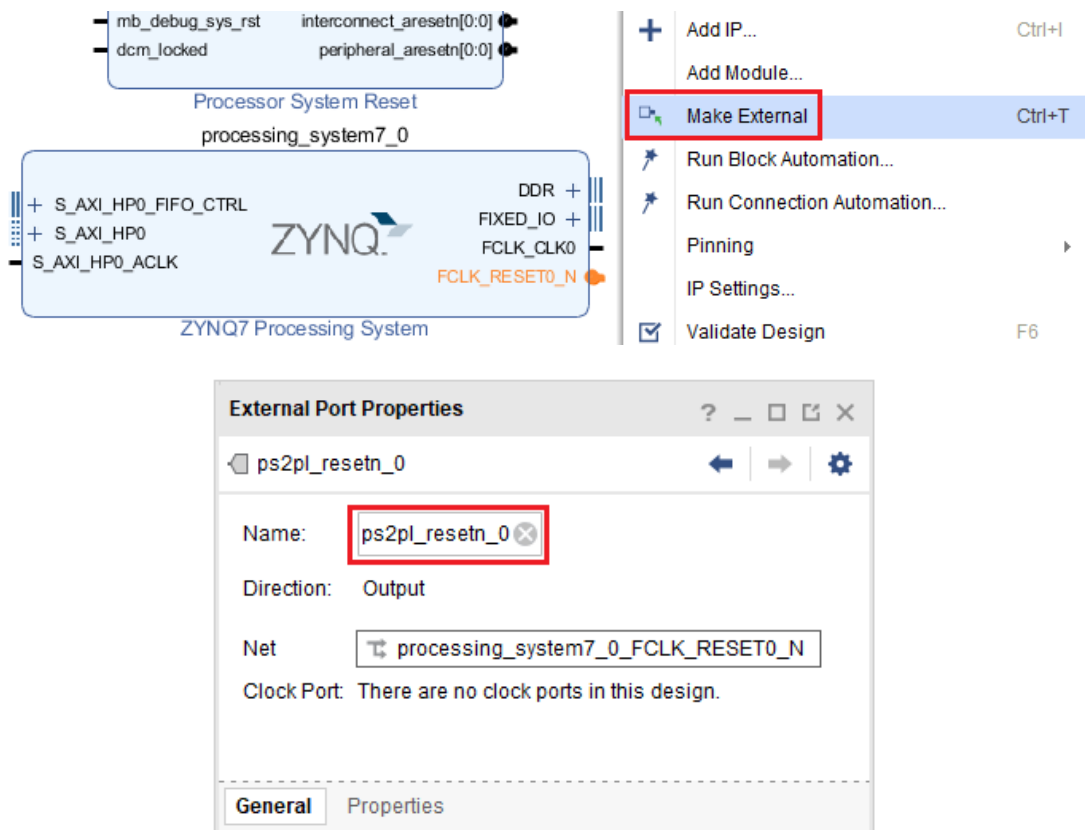


图 1-21 导出引脚与重命名

2. 依据步骤 1，导出 ZYNQ 核的 FCLK_CLK0 接口和 S_AXI_HP0_ACLK 接口，分别用于为 PL 端提供工作时钟、为 HP0 接口提供时钟。为了方便区分，将接口分别重命名为 ps2pl_clk50m_0、pl2ps_axi_aclk_0；导出 Processor System Reset 核的 ext_reset_in 接口，用于 IP 核的复位信号输入接口。为了方便区分，将信号重命名为 pl2ps_axi_resetn_0；导出 AXI Interconnect 核的 S00_AXI 接口，用于输入 AXI4 总线信号。为了方便区分，将信号重命名为 pl2ps_axi_0。导出端口后的设计如图 1-22 所示：

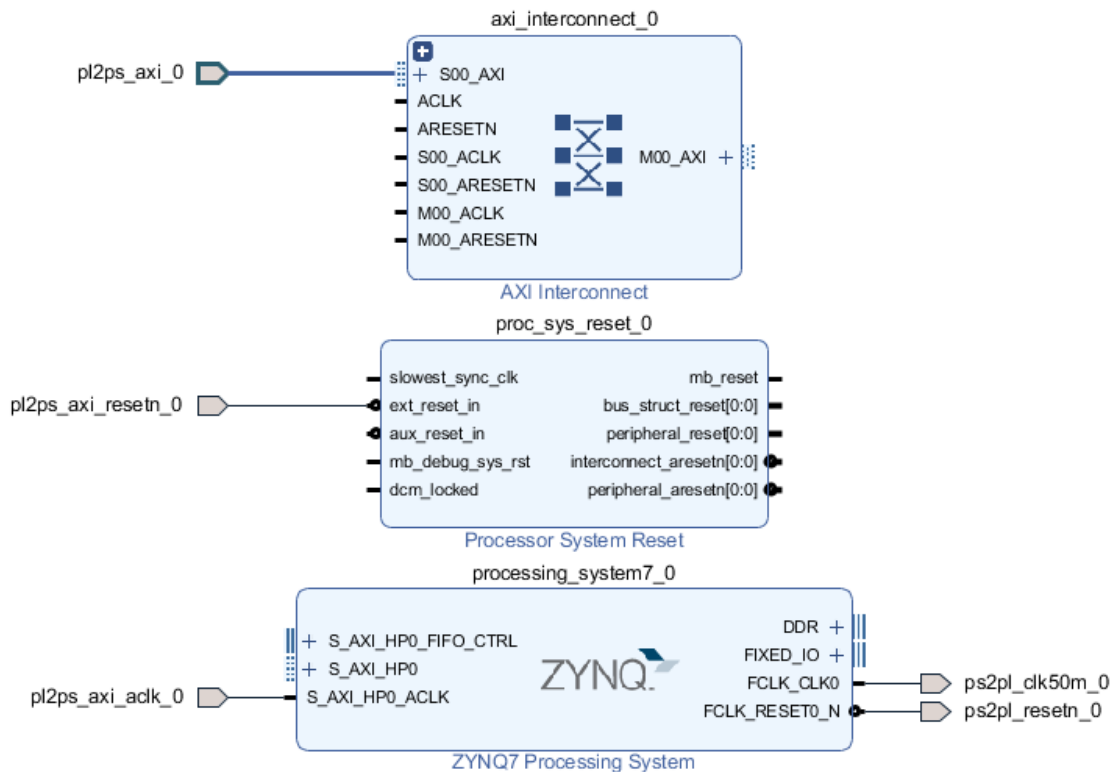


图 1-22 导出端口

3. 连接 AXI Interconnect 核的 M00_AXI 接口和 ZYNQ 核的 S_AXI_HP0 接口，如图 1-23 所示：

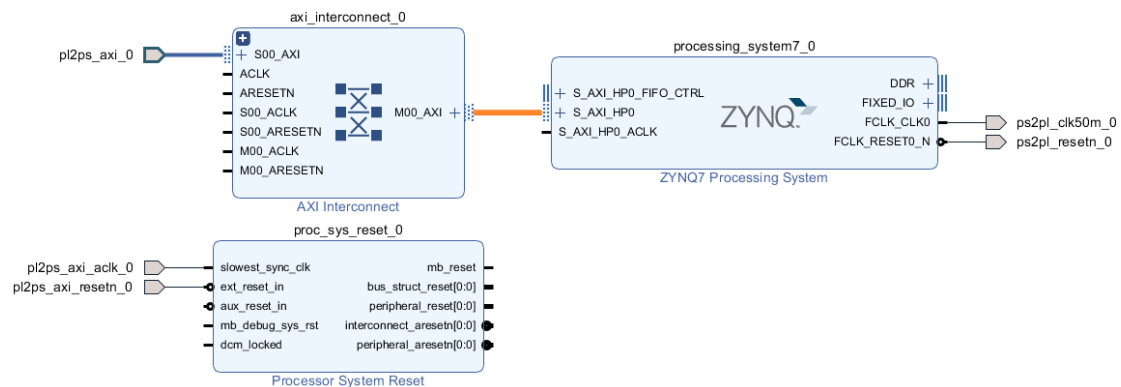


图 1-23 连接 HP 接口

4. 点击上方的绿色字样“Run Block Automation”，导出 ZYNQ 核的 DDR 以及 FIXED_IO 引脚，软件会弹出配置窗口，保持默认即可，如图 1-24 所示：

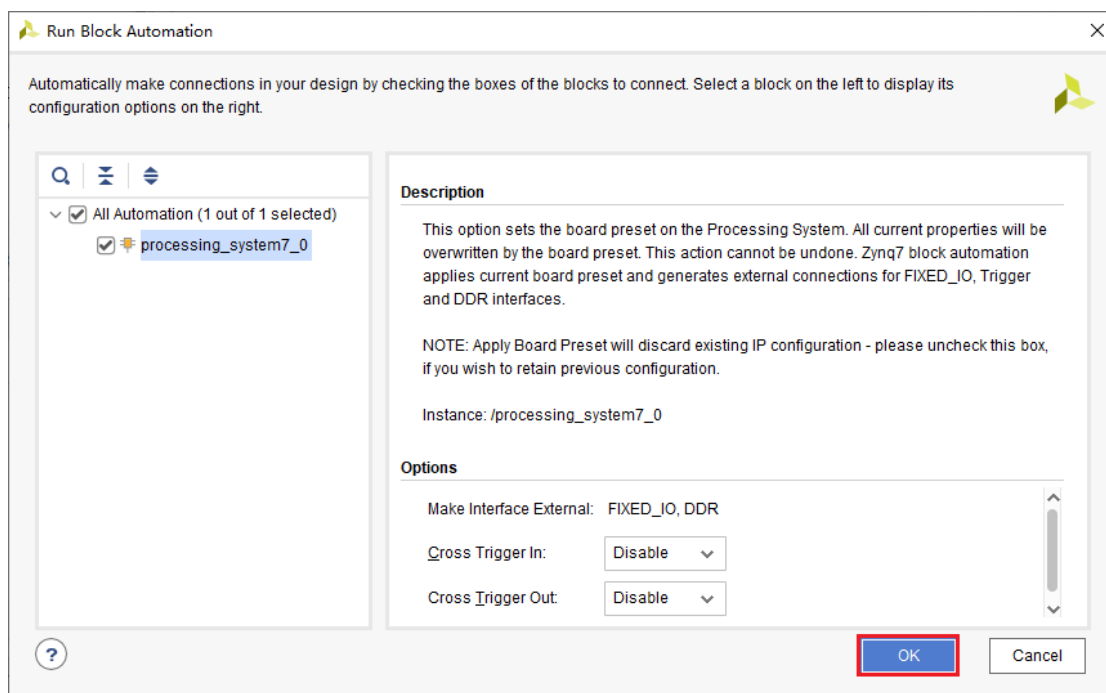


图 1-24 运行模块自动化

5. 点击上方的“Run Connection Automation”，在弹出的窗口中勾选所有接口，并将所有接口的时钟源全部设置为 pl2ps_axi_aclk_0，如图 1-25 所示：

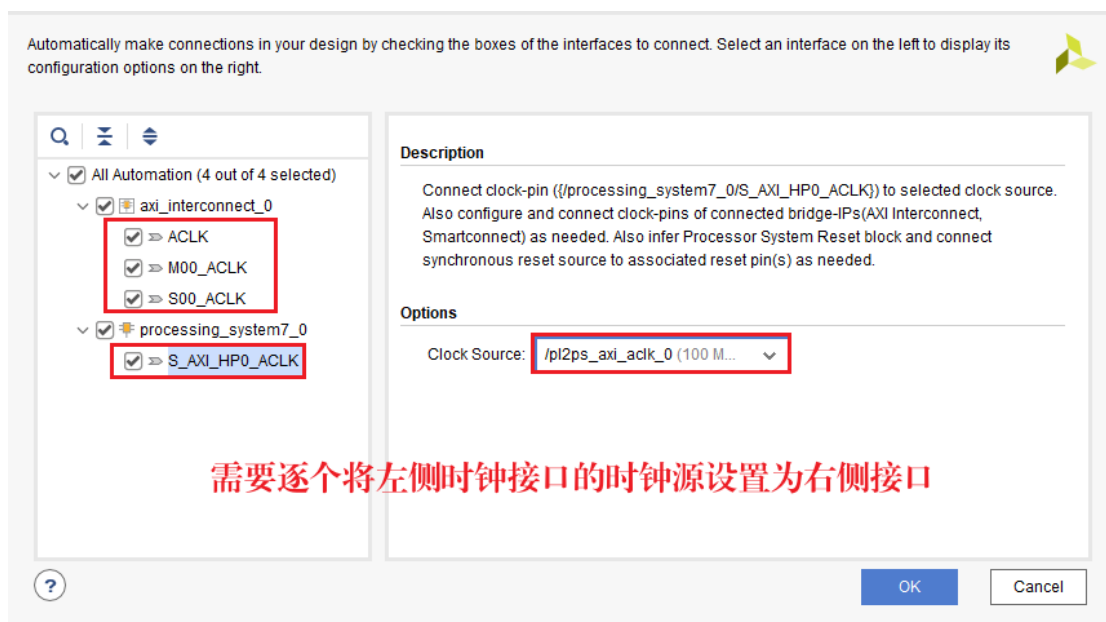


图 1-25 配置自动连接

连接完成的 Block Design 如图 1-26 所示：

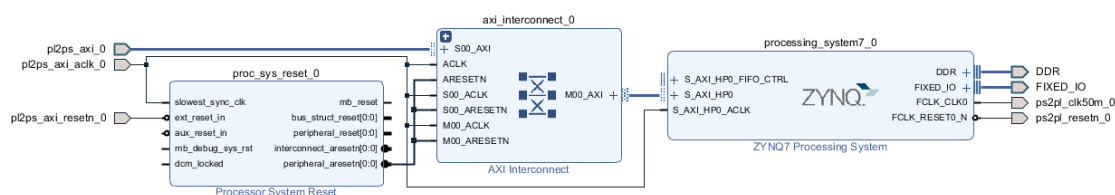


图 1-26 Block Design 完整结构

此时我们并不知道设计是否正确，因此点击上方的√来验证设计，如图 1-27 所示：

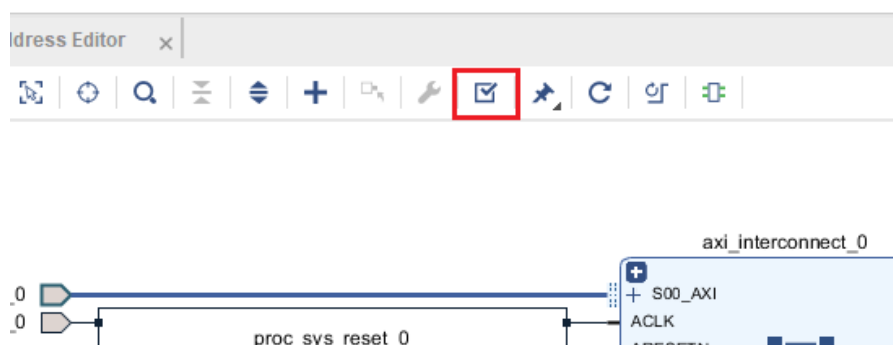


图 1-27 验证设计

此时可以看到，软件弹出严重警告，告诉我们 HP0_DDR_LOWOCM 接口并没有被映射到 pl2ps_axi_0 接口上，并告诉了我们解决方法，如图 1-28 所示：

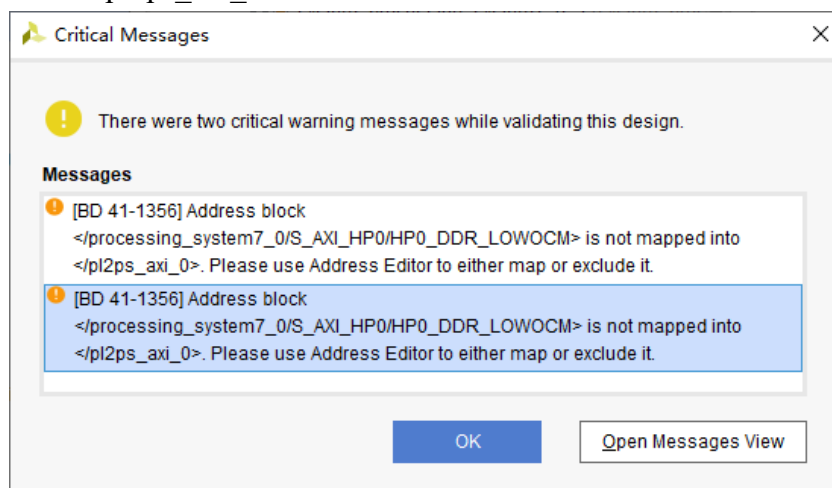


图 1-28 严重警告

此时点击上方的 Address Editor，展开 Unmapped Slaves 就能看到，此时的 HP0 并未被分配任何地址，如图 1-29 所示：



图 1-29 查看地址编辑器

点击上方的 Auto Assign Address，让软件自动帮我们分配地址，如图 1-30 所示：

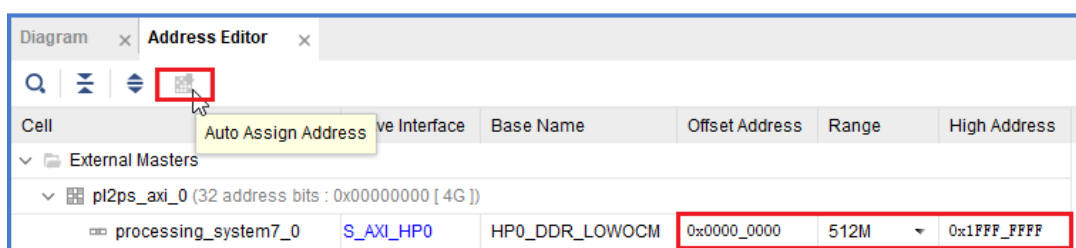


图 1-30 自动分配地址

再次点击√验证设计，可以看到，此时已经没有了警告，如图 1-31 所示：

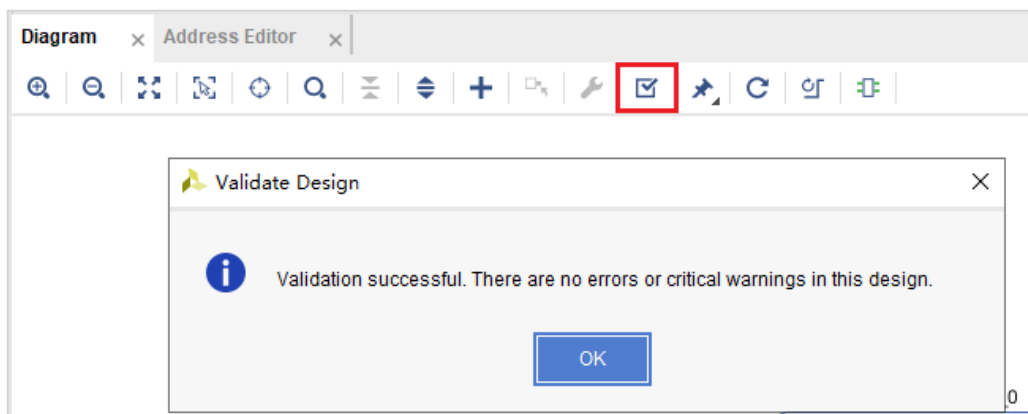


图 1-31 验证设计

1.2.4.5 创建 Block design 封装

经过以上操作，我们已经完成对“Block Design 模块”中各个“子模块”的配置和连接。但是，要想将“Block Design 模块”添加到我们整个设计中，我们还需要为 Block Design 创建一个封装。

在左侧的 Source 栏中，右键单击 system，在弹出的选项中选择生成输出产品，如图 1-32 所示：

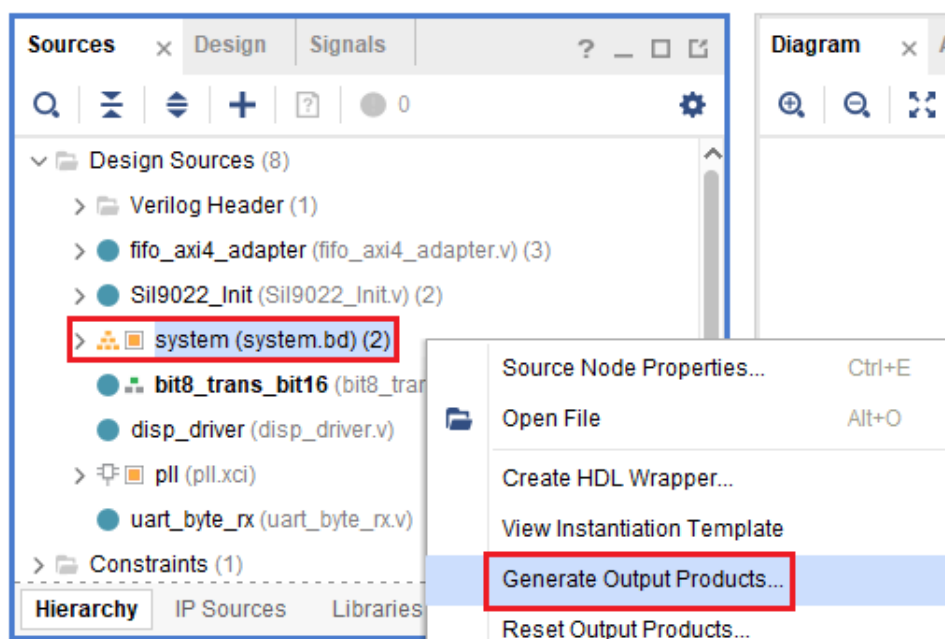


图 1-32 生成输出

在弹出的窗口中我们只需要设置 jobs 的数量为最大值即可（该值与电脑 CPU 性能有关），其余项保持默认，如图 1-33 所示：

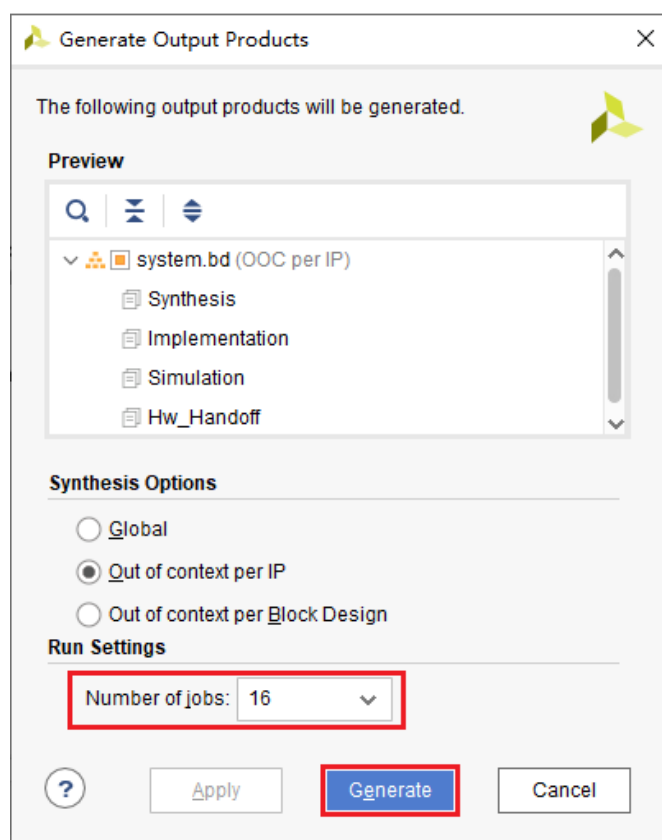


图 1-33 生成输出设置

随后等待输出生成完成，再次右键单击 system，选择 Create HDL Wrapper...，创建封装，如图 1-34 所示：

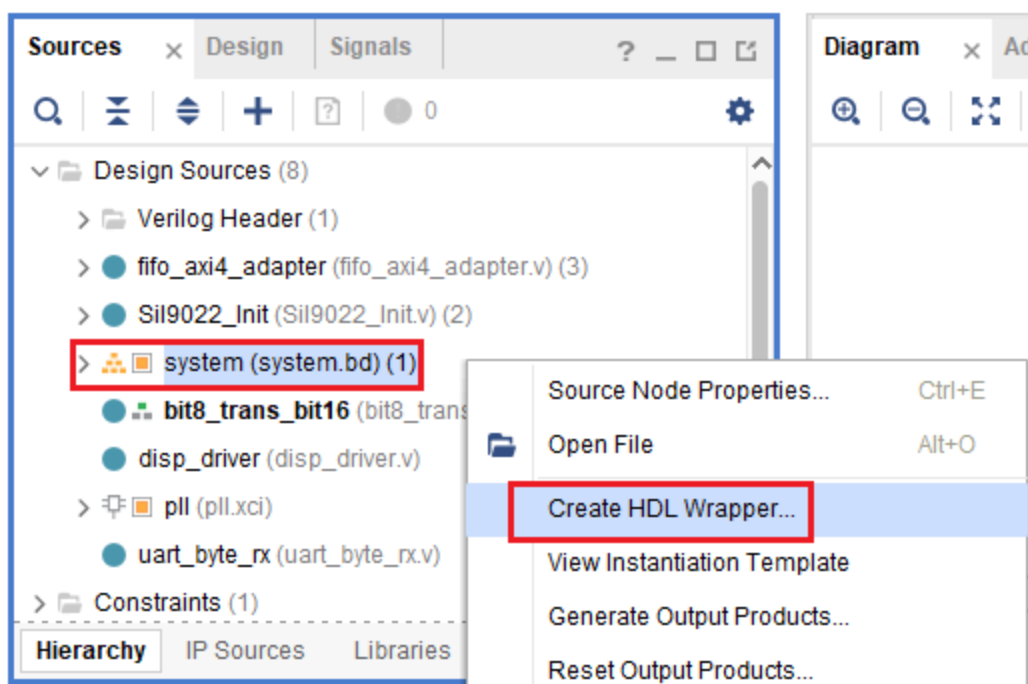


图 1-34 创建封装

在弹出的窗口中，我们选择让 vivado 来管理封装并自动更新，如图 1-35 所示：

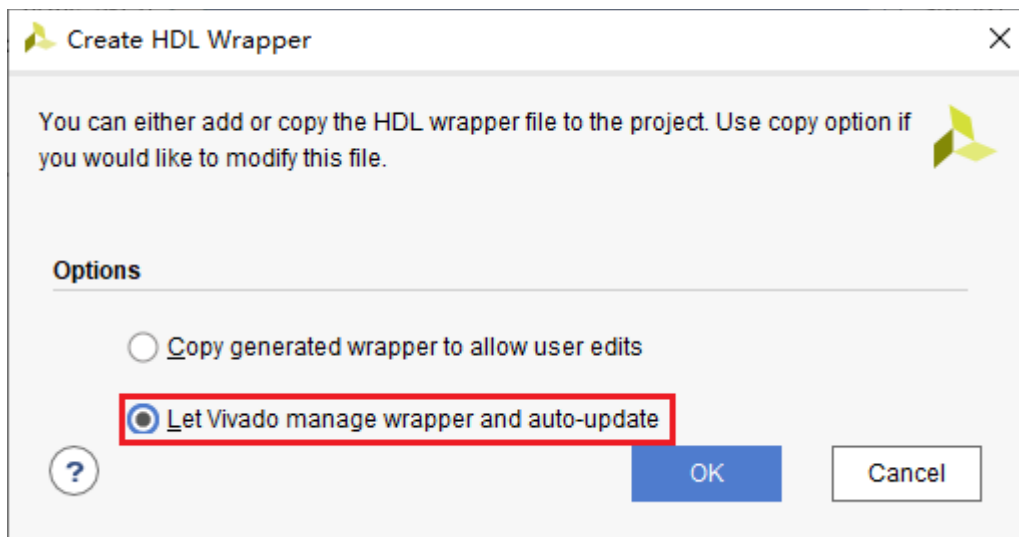


图 1-35 创建封装

点击 OK 后稍等片刻便能看到软件创建好的 system 的封装，如图 1-36 所示：

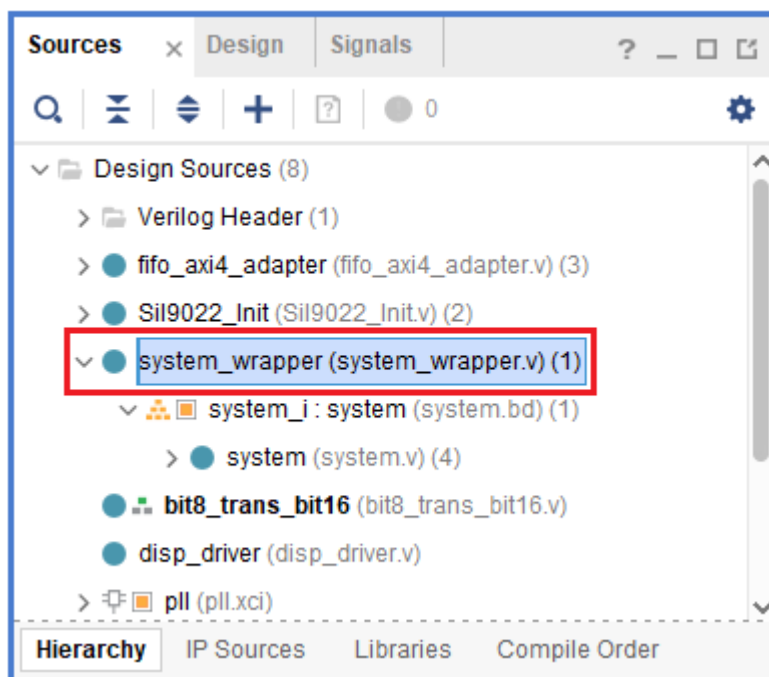


图 1-36 Block Design 封装

此时双击打开 system_wrapper.v，其中例化的便是我们的 system，例化部分代码如下：

```
system system_i
(
    .DDR_addr(DDR_addr),
    .DDR_ba(DDR_ba),
    .DDR_cas_n(DDR_cas_n),
    .DDR_ck_n(DDR_ck_n),
    .DDR_ck_p(DDR_ck_p),
    .DDR_cke(DDR_cke),
    .DDR_cs_n(DDR_cs_n),
    .DDR_dm(DDR_dm),
    .DDR_dq(DDR_dq),
    .DDR_dqs_n(DDR_dqs_n),
    .DDR_dqs_p(DDR_dqs_p),
    .DDR_odt(DDR_odt),
    .DDR_ras_n(DDR_ras_n),
    .DDR_reset_n(DDR_reset_n),
    .DDR_we_n(DDR_we_n),
    .FIXED_IO_ddr_vrn(FIXED_IO_ddr_vrn),
    .FIXED_IO_ddr_vrp(FIXED_IO_ddr_vrp),
    .FIXED_IO_mio(FIXED_IO_mio),
    .FIXED_IO_ps_clk(FIXED_IO_ps_clk),
    .FIXED_IO_ps_porb(FIXED_IO_ps_porb),
    .FIXED_IO_ps_srstb(FIXED_IO_ps_srstb),
    .pl2ps_axi_0_araddr(pl2ps_axi_0_araddr),
    .pl2ps_axi_0_arburst(pl2ps_axi_0_arburst),
```

```
.pl2ps_axi_0_arcache(pl2ps_axi_0_arcache),
.pl2ps_axi_0_arlen(pl2ps_axi_0_arlen),
.pl2ps_axi_0_arlock(pl2ps_axi_0_arlock),
.pl2ps_axi_0_arprot(pl2ps_axi_0_arprot),
.pl2ps_axi_0_arqos(pl2ps_axi_0_arqos),
.pl2ps_axi_0_arready(pl2ps_axi_0_arready),
.pl2ps_axi_0_arregion(pl2ps_axi_0_arregion),
.pl2ps_axi_0_arsize(pl2ps_axi_0_arsize),
.pl2ps_axi_0_arvalid(pl2ps_axi_0_arvalid),
.pl2ps_axi_0_awaddr(pl2ps_axi_0_awaddr),
.pl2ps_axi_0_awburst(pl2ps_axi_0_awburst),
.pl2ps_axi_0_awcache(pl2ps_axi_0_awcache),
.pl2ps_axi_0_awlen(pl2ps_axi_0_awlen),
.pl2ps_axi_0_awlock(pl2ps_axi_0_awlock),
.pl2ps_axi_0_awprot(pl2ps_axi_0_awprot),
.pl2ps_axi_0_awqos(pl2ps_axi_0_awqos),
.pl2ps_axi_0_awready(pl2ps_axi_0_awready),
.pl2ps_axi_0_awregion(pl2ps_axi_0_awregion),
.pl2ps_axi_0_awsized(pl2ps_axi_0_awsized),
.pl2ps_axi_0_awvalid(pl2ps_axi_0_awvalid),
.pl2ps_axi_0_bready(pl2ps_axi_0_bready),
.pl2ps_axi_0_bresp(pl2ps_axi_0_bresp),
.pl2ps_axi_0_bvalid(pl2ps_axi_0_bvalid),
.pl2ps_axi_0_rdata(pl2ps_axi_0_rdata),
.pl2ps_axi_0_rlast(pl2ps_axi_0_rlast),
.pl2ps_axi_0_rready(pl2ps_axi_0_rready),
.pl2ps_axi_0_rresp(pl2ps_axi_0_rresp),
.pl2ps_axi_0_rvalid(pl2ps_axi_0_rvalid),
.pl2ps_axi_0_wdata(pl2ps_axi_0_wdata),
.pl2ps_axi_0_wlast(pl2ps_axi_0_wlast),
.pl2ps_axi_0_wready(pl2ps_axi_0_wready),
.pl2ps_axi_0_wstrb(pl2ps_axi_0_wstrb),
.pl2ps_axi_0_wvalid(pl2ps_axi_0_wvalid),
.pl2ps_axi_acl_0(pl2ps_axi_acl_0),
.pl2ps_axi_resetsn_0(pl2ps_axi_resetsn_0),
.ps2pl_clk50m_0(ps2pl_clk50m_0),
.ps2pl_resetsn_0(ps2pl_resetsn_0));
```

其中，DDR 与 FIXED_IO 相关端口为 PS 侧端口；pl2ps_axi 相关端口为我们导出的用于 PL 侧控制 AXI 总线将数据写入 PS 的相关端口；ps2pl 相关端口为我们导出的，PS 端输出给 PL 的控制信号端口。

1.2.5 创建顶层设计

经过上述步骤，我们已经创建好了各个模块，接下来需要做的就是创建设计顶层，例化并连接各个模块了。

本次顶层设计内容如下：

```
module uart_hp_ddr3_tft800x480_hdmi(  
    //user  
    input          reset_n          ,  
    output         led              ,  
    //TFT Interface  
    output [15:0]   TFT_rgb          , //TFT 数据输出  
    output         TFT_hs            , //TFT 行同步信号  
    output         TFT_vs            , //TFT 场同步信号  
    output         TFT_clk           , //TFT 像素时钟  
    output         TFT_de            , //TFT 数据使能  
    output         TFT_pwm           , //TFT 背光控制  
    //Rx uart Interface  
    input          uart_rx           ,  
    //ov5640 init IIC interface  
    output         SiI9022_sclk      ,  
    inout          SiI9022_sdat      ,  
    //DDR3 Interface  
    // Inouts  
    inout [31:0]   ddr3_dq           ,  
    inout [3:0]    ddr3_dqs_n        ,  
    inout [3:0]    ddr3_dqs_p        ,  
    // 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 [0:0]    ddr3_ck_p         ,  
    output [0:0]    ddr3_ck_n         ,  
    output [0:0]    ddr3_cke         ,  
    output [0:0]    ddr3_cs_n         ,  
    output [3:0]    ddr3_dm           ,  
    output [0:0]    ddr3_odt         ,  
    inout          FIXED_IO_ddr_vrn   ,  
    inout          FIXED_IO_ddr_vrp   ,  
    inout [53:0]    FIXED_IO_mio      ,  
    inout          FIXED_IO_ps_clk    ,  
    inout          FIXED_IO_ps_porcb  ,  
    inout          FIXED_IO_ps_srstb  ,  
);  
//Resolution_800x480 像素时钟为 33MHz  
parameter IMAGE_WIDTH  = 800;  
parameter IMAGE_HEIGHT = 480;  
  
//PL 使用 DDR 的基地址，留一定空间给 PS 用
```

```
parameter DDR_BASE_ADDR = 32'h1800000;

//*****
//Internal connect
//*****

//clock
wire      ps2pl_clk50m_0; //系统时钟输入, 50MHz
wire      ps2pl_resetrn_0; //复位信号输入
wire      pll_locked;
wire      loc_clk50m;
wire      loc_clk100m;
wire      disp_clk;
wire      disp_clk5x;
wire      reset;
//camera interface
wire [15:0] image_data;
wire      image_data_valid;
wire      image_data_hs;
wire      image_data_vs;
//tft
wire      frame_begin;
wire      disp_data_req;
wire [15:0] disp_data;
//VGA
wire [4:0] Disp_Red;
wire [5:0] Disp_Green;
wire [4:0] Disp_Blue;
//wr_fifo Interface
wire      wrfifo_clr;
wire [15:0] wrfifo_din;
wire      wrfifo_wren;
//rd_fifo Interface
wire      rdfifo_clr;
wire      rdfifo_rden;
wire [15:0] rdfifo_dout;
//axi
wire [3:0] s_axi_awid;
wire [31:0] s_axi_awaddr;
wire [7:0] s_axi_awlen;
wire [2:0] s_axi_awsize;
wire [1:0] s_axi_awburst;
wire [0:0] s_axi_awlock;
wire [3:0] s_axi_awcache;
wire [2:0] s_axi_awprot;
wire [3:0] s_axi_awqos;
wire [3:0] s_axi_awregion;
wire      s_axi_awvalid;
```

```
wire          s_axi_awready;
//
wire [63:0]    s_axi_wdata;
wire [7:0]     s_axi_wstrb;
wire          s_axi_wlast;
wire          s_axi_wvalid;
wire          s_axi_wready;
//
wire [3:0]     s_axi_bid;
wire [1:0]     s_axi_bresp;
wire          s_axi_bvalid;
wire          s_axi_bready;
//
wire [3:0]     s_axi_arid;
wire [31:0]    s_axi_araddr;
wire [7:0]     s_axi_arlen;
wire [2:0]     s_axi_arsize;
wire [1:0]     s_axi_arburst;
wire [0:0]     s_axi_arlock;
wire [3:0]     s_axi_arcache;
wire [2:0]     s_axi_arprot;
wire [3:0]     s_axi_arqos;
wire [3:0]     s_axi_arregion;
wire          s_axi_arvalid;
wire          s_axi_arready;
//
wire [3:0]     s_axi_rid;
wire [63:0]    s_axi_rdata;
wire [1:0]     s_axi_rresp;
wire          s_axi_rlast;
wire          s_axi_rvalid;
wire          s_axi_rready;
//
wire          s_axi_aclk;
wire          s_axi_reseten;
//
wire [7:0]     uart_byte;
wire          uart_byte_vaild;

assign loc_clk50m      = ps2pl_clk50m_0;
assign s_axi_aclk      = loc_clk100m;

wire          pl_reset_n;
wire          reset_pre;
reg  [19:0]    reset_sync;
assign pl_reset_n = ps2pl_reseten_0 & reset_n;
assign reset_pre  = ~pll_locked;
```

```
//PS 先释放复位，PL 的逻辑复位释放往后延迟 20 个时钟周期
always@(posedge loc_clk100m or posedge reset_pre)
begin
    if(reset_pre)
        reset_sync <= {20{1'b1}};
    else
        reset_sync <= reset_sync << 1;
end

assign reset = reset_sync[19];
assign s_axi_resetn = ~reset;

uart_byte_rx#(
    .CLK_FRQ(100_000_000)
)
uart_byte_rx(
    .Clk      (loc_clk100m      ),
    .Reset_n   (!reset          ),

    .Baud_Set  (3'd5             ), //1562500bps
    .uart_rx   (uart_rx         ),

    .Data(uart_byte      ),
    .Rx_Done  (uart_byte_vaild )
);

bit8_trans_bit16 bit8_trans_bit16
(
    .clk      (loc_clk100m      ),
    .reset_p   (reset           ),

    .bit8_in   (uart_byte       ),
    .bit8_in_valid (uart_byte_vaild ),

    .bit16_out  (image_data      ),
    .bit16_out_valid (image_data_valid)
);

pll pll
(
    // Clock out ports
    .clk_out1 (loc_clk100m      ), // output clk_out1
    .clk_out2 (disp_clk         ), // output clk_out2
    // Status and control signals
    .resetn   (pl_reset_n      ), // input reset
    .locked    (pll_locked      ), // output locked
```

```
// Clock in ports
.clk_in1 (ps2pl_clk50m_0 ) // input clk_in1
);

assign wrfifo_clr  = reset;
assign wrfifo_wren = image_data_valid;
assign wrfifo_din  = image_data;

assign rdfifo_clr  = reset || frame_begin;
assign rdfifo_rden = disp_data_req;
assign disp_data   = rdfifo_dout;

//显示模块
disp_driver disp_driver(
    .ClkDisp      (disp_clk      ),
    .Rst_p        (reset        ),

    .Data          (disp_data     ),
    .DataReq       (disp_data_req ),

    .H_Addr        (              ),
    .V_Addr        (              ),

    .Disp_HS       (TFT_hs        ),
    .Disp_VS       (TFT_vs        ),
    .Disp_Red      (Disp_Red ),
    .Disp_Green    (Disp_Green ),
    .Disp_Blue     (Disp_Blue ),
    .Frame_Begin   (frame_begin ),
    .Disp_DE       (TFT_de        ),
    .Disp_PCLK     (TFT_clk       )
);
assign TFT_rgb = {Disp_Red,Disp_Green,Disp_Blue};
assign TFT_pwm = 1'b1;

fifo_axi4_adapter #(
    .FIFO_DW          (16          ),
    .WR_AXI_BYTE_ADDR_BEGIN (DDR_BASE_ADDR + 1'b1 ),
    .WR_AXI_BYTE_ADDR_END   (DDR_BASE_ADDR +
IMAGE_WIDTH*IMAGE_HEIGHT*2),
    .RD_AXI_BYTE_ADDR_BEGIN (DDR_BASE_ADDR + 1'b1 ),
    .RD_AXI_BYTE_ADDR_END   (DDR_BASE_ADDR +
IMAGE_WIDTH*IMAGE_HEIGHT*2),

    .AXI_DATA_WIDTH      (64          ),
    .AXI_ADDR_WIDTH      (32          ),
```

```
.AXI_ID          (4'b0000          ),
.AXI_BURST_LEN   (8'd15           ) //axi burst
length = 16
)fifo_axi4_adapter_inst
(
  //clock reset
  .clk            (loc_clk100m      ),
  .reset          (reset            ),
  //wr_fifo Interface
  .wrfifo_clr     (wrfifo_clr       ),
  .wrfifo_clk     (loc_clk100m      ),
  .wrfifo_wren    (wrfifo_wren      ),
  .wrfifo_din     (wrfifo_din       ),
  .wrfifo_full    (                 ),
  .wrfifo_wr_cnt  (                 ),
  //rd_fifo Interface
  .rdfifo_clr     (rdfifo_clr       ),
  .rdfifo_clk     (disp_clk         ),
  .rdfifo_rden    (rdfifo_rden      ),
  .rdfifo_dout    (rdfifo_dout      ),
  .rdfifo_empty   (                 ),
  .rdfifo_rd_cnt  (                 ),
  // Master Interface Write Address Ports
  .m_axi_awid     (s_axi_awid       ),
  .m_axi_awaddr   (s_axi_awaddr     ),
  .m_axi_awlen    (s_axi_awlen      ),
  .m_axi_awsz     (s_axi_awsz       ),
  .m_axi_awburst  (s_axi_awburst    ),
  .m_axi_awlock   (s_axi_awlock     ),
  .m_axi_awcache  (s_axi_awcache    ),
  .m_axi_awprot   (s_axi_awprot     ),
  .m_axi_awqos    (s_axi_awqos      ),
  .m_axi_awregion (s_axi_awregion   ),
  .m_axi_awvalid  (s_axi_awvalid    ),
  .m_axi_awready  (s_axi_awready    ),
  // Master Interface Write Data Ports
  .m_axi_wdata    (s_axi_wdata      ),
  .m_axi_wstrb    (s_axi_wstrb      ),
  .m_axi_wlast    (s_axi_wlast      ),
  .m_axi_wvalid   (s_axi_wvalid     ),
  .m_axi_wready   (s_axi_wready     ),
  // Master Interface Write Response Ports
  .m_axi_bid      (4'b0000         ),
  .m_axi_bresp    (s_axi_bresp      ),
  .m_axi_bvalid   (s_axi_bvalid     ),
  .m_axi_bready   (s_axi_bready     ),
  // Master Interface Read Address Ports
```

```

.m_axi_arid      (s_axi_arid      ),
.m_axi_araddr    (s_axi_araddr    ),
.m_axi_arlen     (s_axi_arlen     ),
.m_axi_arsize    (s_axi_arsize    ),
.m_axi_arburst   (s_axi_arburst   ),
.m_axi_arlock    (s_axi_arlock    ),
.m_axi_arcache   (s_axi_arcache   ),
.m_axi_arprot    (s_axi_arprot    ),
.m_axi_arqos     (s_axi_arqos     ),
.m_axi_arregion  (s_axi_arregion  ),
.m_axi_arvalid   (s_axi_arvalid   ),
.m_axi_arready   (s_axi_arready   ),
// Master Interface Read Data Ports
.m_axi_rid       (4'b0000        ),
.m_axi_rdata     (s_axi_rdata     ),
.m_axi_rresp     (s_axi_rresp     ),
.m_axi_rlast     (s_axi_rlast     ),
.m_axi_rvalid    (s_axi_rvalid    ),
.m_axi_rready    (s_axi_rready    )
);

```

```

system_wrapper system_wrapper
(

```

```

.DDR_addr      (ddr3_addr      ),
.DDR_ba       (ddr3_ba       ),
.DDR_cas_n     (ddr3_cas_n     ),
.DDR_ck_n     (ddr3_ck_n     ),
.DDR_ck_p     (ddr3_ck_p     ),
.DDR_cke      (ddr3_cke      ),

.DDR_cs_n     (ddr3_cs_n     ),
.DDR_dm       (ddr3_dm       ),
.DDR_dq       (ddr3_dq       ),
.DDR_dqs_n    (ddr3_dqs_n    ),
.DDR_dqs_p    (ddr3_dqs_p    ),
.DDR_odt      (ddr3_odt      ),
.DDR_ras_n    (ddr3_ras_n    ),
.DDR_reset_n  (ddr3_reset_n  ),
.DDR_we_n     (ddr3_we_n     ),
.FIXED_IO_ddr_vrn (FIXED_IO_ddr_vrn ),
.FIXED_IO_ddr_vrp (FIXED_IO_ddr_vrp ),
.FIXED_IO_mio  (FIXED_IO_mio  ),
.FIXED_IO_ps_clk (FIXED_IO_ps_clk ),
.FIXED_IO_ps_porpb (FIXED_IO_ps_porpb ),
.FIXED_IO_ps_srstb (FIXED_IO_ps_srstb ),
.ps2pl_clk50m_0 (ps2pl_clk50m_0 ),

```

```
.ps2pl_resetrn_0      (ps2pl_resetrn_0      ),
//Slave Interface Read Address Ports
.pl2ps_axi_0_araddr   (s_axi_araddr        ),
.pl2ps_axi_0_arburst  (s_axi_arburst       ),
.pl2ps_axi_0_arcache  (s_axi_arcache       ),
.pl2ps_axi_0_arlen    (s_axi_arlen        ),
.pl2ps_axi_0_arlock   (s_axi_arlock       ),
.pl2ps_axi_0_arprot   (s_axi_arprot       ),
.pl2ps_axi_0_arqos    (s_axi_arqos       ),
//  .pl2ps_axi_0_arregion(s_axi_arregion    ),
.pl2ps_axi_0_arready  (s_axi_arready       ),
.pl2ps_axi_0_arsize   (s_axi_arsize       ),
.pl2ps_axi_0_arvalid  (s_axi_arvalid      ),
//Slave Interface Write Address Ports
.pl2ps_axi_0_awaddr   (s_axi_awaddr       ),
.pl2ps_axi_0_awburst  (s_axi_awburst      ),
.pl2ps_axi_0_awcache  (s_axi_awcache      ),
.pl2ps_axi_0_awlen    (s_axi_awlen       ),
.pl2ps_axi_0_awlock   (s_axi_awlock      ),
.pl2ps_axi_0_awprot   (s_axi_awprot      ),
.pl2ps_axi_0_awqos    (s_axi_awqos      ),
//  .pl2ps_axi_0_awregion(s_axi_awregion    ),
.pl2ps_axi_0_awready  (s_axi_awready      ),
.pl2ps_axi_0_awszise  (s_axi_awszise      ),
.pl2ps_axi_0_awvalid  (s_axi_awvalid      ),
//Slave Interface Write Response Ports
.pl2ps_axi_0_bready   (s_axi_bready       ),
.pl2ps_axi_0_bresp    (s_axi_bresp       ),
.pl2ps_axi_0_bvalid   (s_axi_bvalid      ),
//Slave Interface Read Data Ports
.pl2ps_axi_0_rdata    (s_axi_rdata       ),
.pl2ps_axi_0_rlast    (s_axi_rlast       ),
.pl2ps_axi_0_rready   (s_axi_rready      ),
.pl2ps_axi_0_rresp    (s_axi_rresp      ),
.pl2ps_axi_0_rvalid   (s_axi_rvalid      ),
//Slave Interface Write Data Ports
.pl2ps_axi_0_wdata    (s_axi_wdata       ),
.pl2ps_axi_0_wlast    (s_axi_wlast       ),
.pl2ps_axi_0_wready   (s_axi_wready      ),
.pl2ps_axi_0_wstrb    (s_axi_wstrb      ),
.pl2ps_axi_0_wvalid   (s_axi_wvalid      ),
//Slave Interface ACLK RESET
.pl2ps_axi_aclk_0     (s_axi_aclk        ),
.pl2ps_axi_resetrn_0  (s_axi_resetrn     )
);

reg [20:0]cnt;
```

```
reg Go;

always@(posedge ps2pl_clk50m_0 or negedge ps2pl_resetrn_0)
if(!ps2pl_resetrn_0)
    cnt <= 0;
else if(cnt <= 499999)
    cnt <= cnt + 1 ;
else
    cnt <= 500001;

always@(posedge ps2pl_clk50m_0 or negedge ps2pl_resetrn_0)
if(!ps2pl_resetrn_0)
    Go <= 0;
else if(cnt == 499999)
    Go <= 1'b1;
else
    Go <= 0;

SiI9022_Init SiI9022_Init(
    .Clk(ps2pl_clk50m_0),
    .Rst_n(ps2pl_resetrn_0),

    .Go(Go),
    .device_id(8'h72),
    .Init_Done(led),

    .i2c_sclk(SiI9022_sclk),
    .i2c_sdat(SiI9022_sdat)
);

endmodule
```

顶层中对于各个模块的例化、连接并没有过多要讲的内容，这里主要来说复位。顶层中涉及到很多的复位相关信号，这些信号都是围绕以下五个复位信号展开：

1. 从 ZYNQ 核中导出的，PS 提供给 PL 的复位信号 ps2pl_resetrn_0
2. 用户输入的复位信号 reset_n
3. 基于 ps2pl_resetrn_0 和 reset_n 产生的 PL 复位信号 pl_reset_n
4. 确保 PL 的逻辑复位在 PS 之后的复位信号 reset
5. 用于复位 AXI 总线的复位信号 s_axi_resetrn

复位信号 3 基于复位信号 1 和 2，当 PS 复位或用户复位时，pl_reset_n 复位，

代码实现如下：

```
assign pl_reset_n = ps2pl_resetrn_0 & reset_n;
```

pl_reset_n 信号被用作 PLL 的复位信号，确保 PLL 能够第一时间开始工作。PLL 从复位完成到输出稳定时钟需要一定的时间，在时钟稳定后，PLL 会输出高电平的 locked 信号。此时，时钟稳定，系统能够稳定运行，便可以开始其他模块的复位。通过对 locked 信号取非，我们就能知道何时可以准备复位设计模块。

```
assign reset_pre = ~pll_locked;
```

因为本次设计涉及到了 PS，所以系统的启动流程与以往的纯逻辑设计不同。系统首先会对 PS 与 PL 复位，随后根据 bit 文件配置 PL，接着再回头配置 PS，运行 PS 的用户程序。

而在这一整套启动过程中，如果 PL 端先于 PS 端工作，就会出现 PL 端想要读写 DDR3 但是 PS 端还未配置完成，进而导致卡死的情况。为了避免这种情况我们需要让 PL 端的模块复位延迟一段时间，进而得到复位信号 reset，代码实现如下：

```
//PS 先释放复位，PL 的逻辑复位释放往后延迟 20 个时钟周期
always@(posedge loc_clk100m or posedge reset_pre)
begin
    if(reset_pre)
        reset_sync <= {20{1'b1}};
    else
        reset_sync <= reset_sync << 1;
end

assign reset = reset_sync[19];
```

reset 信号被用于除 PLL 外各个模块的复位，为了使 Block design 中用于协议转换的 AXI Interconnect 模块也能同步复位，需要将 reset 连接到我们导出的 s_axi_resetrn 接口上，代码实现如下：

```
assign s_axi_resetrn = ~reset;
```

以上便是本次顶层设计以及顶层中各个复位信号的由来及作用。添加完顶层后，我们需要为设计分配引脚并进行电平约束，随后生成 bit 文件后，才能导出硬件资源描述文件。

1.2.6 引脚约束

打开管脚约束窗口，其中 Package Pin 和 I/O Std 是我们要约束的内容。进行管脚分配时可以参考表 1-2，分配完成后如图 1-37 所示。

表 1-2 引脚分配表

Pin Name	Signal Name	Pin NO.	Pin Name	Signal Name	Pin NO.
FPGA_UART_RX	uart_rx	K16	TFT_rgb[12]	Display_R1	V16
FPGA_KEY0	reset_n	F20	TFT_rgb[11]	Display_R0	T15
AUD_I2C_SCL	SiI9022_sclk	T10	TFT_rgb[10]	Display_G5	V20
AUD_I2C_SDA	SiI9022_sdat	R14	TFT_rgb[9]	Display_G4	U17
FPGA_LED0	led	T14	TFT_rgb[8]	Display_G3	V18
TFT_clk	Display_PCLK	U15	TFT_rgb[7]	Display_G2	T16
TFT_de	Display_DE	W15	TFT_rgb[6]	Display_G1	R16
TFT_pwm	Display_BL	R17	TFT_rgb[5]	Display_G0	U19
TFT_hs	Display_HSYNC	U14	TFT_rgb[4]	Display_B4	Y19
TFT_vs	Display_VSYNC	W14	TFT_rgb[3]	Display_B3	W18
TFT_rgb[15]	Display_R4	W20	TFT_rgb[2]	Display_B2	Y18
TFT_rgb[14]	Display_R3	W19	TFT_rgb[1]	Display_B1	W16
TFT_rgb[13]	Display_R2	V17	TFT_rgb[0]	Display_B0	Y17

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	IN_TERM
DDR_38849 (71)	(Multiple)			✓	502	(Multiple)*	(Multiple)	(Multiple)	(Multiple)	(Multiple)	NONE	FP_VTT_50	
FIFO_READ_3567 (1)	OUT			✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
FIXED_IO_38849 (59)	INOUT			✓	(Multiple)	(Multiple)*	(Multiple)	(Multiple)		(Multiple)	NONE	FP_VTT_50	
read_clk_3567 (1)	OUT			✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
Scalar ports (1)													
TFT_clk	OUT		U15	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[15]	OUT		W20	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[14]	OUT		W19	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[13]	OUT		V17	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[12]	OUT		V16	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[11]	OUT		T15	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[10]	OUT		V20	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[9]	OUT		U17	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[8]	OUT		V18	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[7]	OUT		T16	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[6]	OUT		R16	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[5]	OUT		U19	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[4]	OUT		Y19	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[3]	OUT		W18	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[2]	OUT		Y18	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[1]	OUT		W16	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_rgb[0]	OUT		Y17	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
Scalar ports (8)													
led	OUT		T14	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
reset_n	IN		F20	✓	35	LVCMS33*	3.300				NONE	NONE	
SiI9022_sclk	OUT		T10	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
SiI9022_sdat	INOUT		R14	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_hs	OUT		U14	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_pwm	OUT		R17	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
TFT_vs	OUT		W14	✓	34	LVCMS33*	3.300		12	SLOW	NONE	FP_VTT_50	✓
uart_rx	IN		K16	✓	35	LVCMS33*	3.300				NONE	NONE	✓

图 1-37 引脚分配图

完成约束后生成 bit 流，在 bit 生成完成后，点击 file->Export->Export Hardware...导出硬件，如图 1-38 所示：

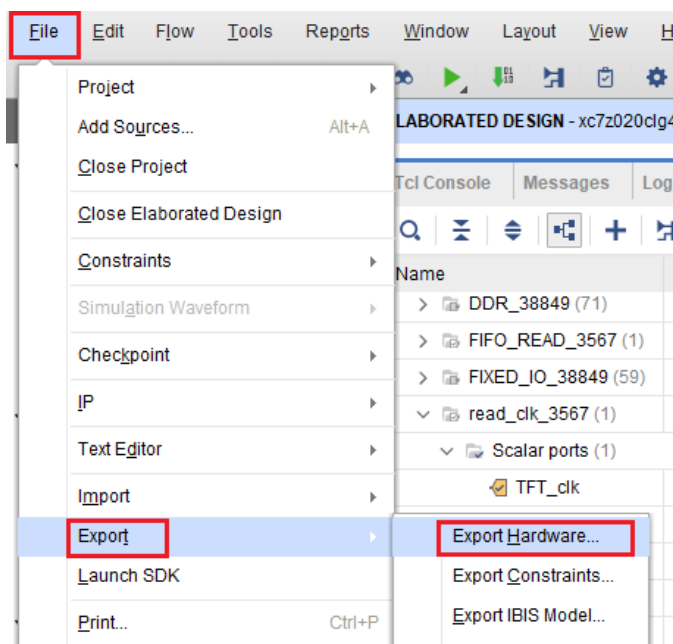


图 1-38 导出硬件

软件会弹出一个选项界面，这里我们需要勾选包含比特流，如图 1-39 所示：

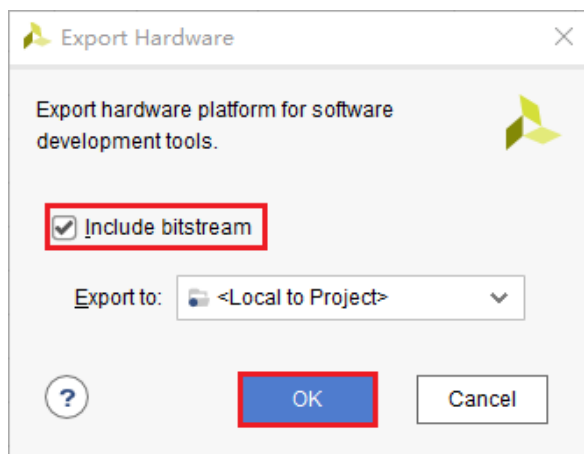


图 1-39 包含比特流

此时如果在资源管理器中打开工程路径，便会发现多出来了一个后缀为.sdk 的文件夹，该文件夹用来存放 SDK 相关文件。进入文件夹后便能看到我们刚刚导出的后缀为.hdf的硬件资源描述文件，如图 1-40 所示：



图 1-40 导出的硬件资源描述文件

接下来我们运行 SDK 开始 PS 部分的设计。点击 File->Launch SDK，软件

会弹出路径设置界面，这里我们保持默认点击 OK，软件会为我们调用 SDK 软件，如图 1-41 所示：

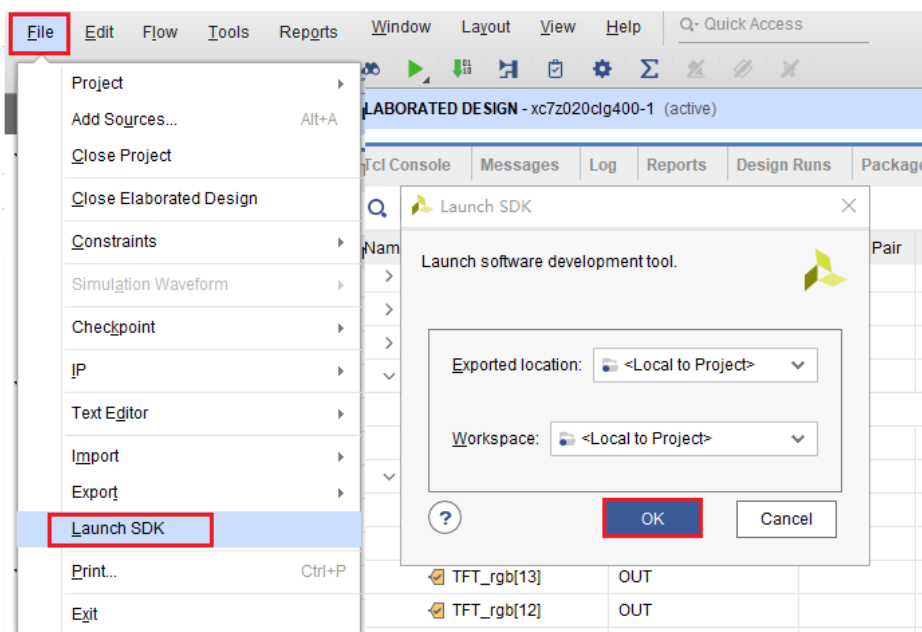


图 1-41 运行 SDK

1.2.7 运行 SDK

SDK 软件在打开后会创建工作空间，将导出的硬件资源描述文件解压到工作空间中，得到包含 bit 在内的一系列文件，如图 1-42 所示：

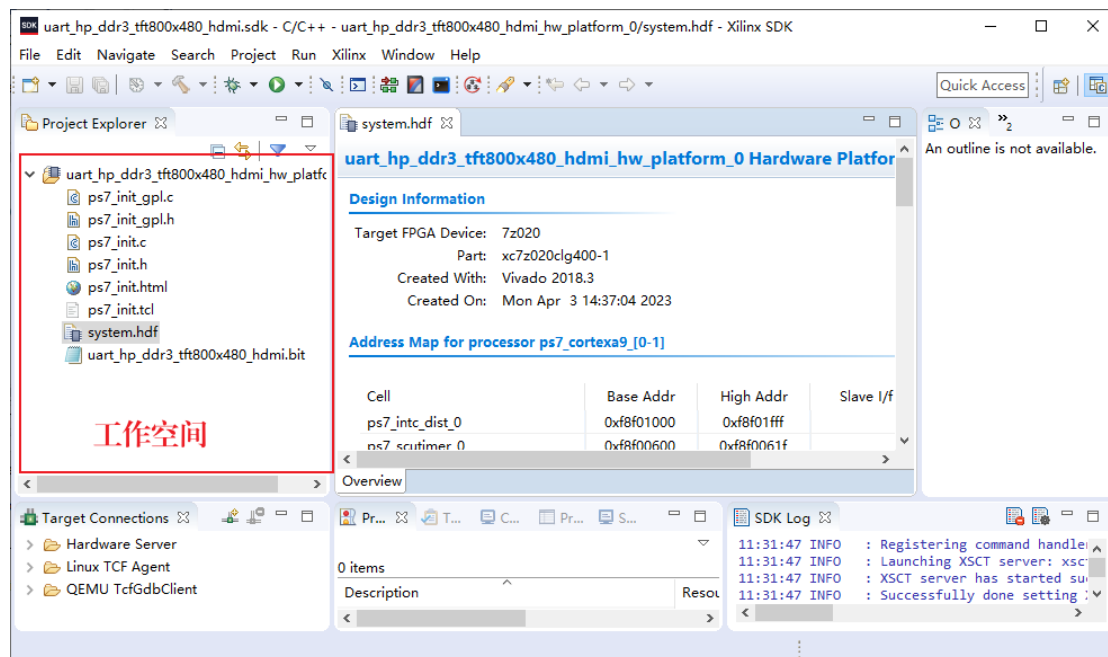


图 1-42 SDK 界面

其中的 ps7_init.tcl 和 bit 文件便是本次设计所需的文件，前者为 PS 端的初始化（配置）脚本，后者为 PL 端配置文件，如图 1-43 所示：

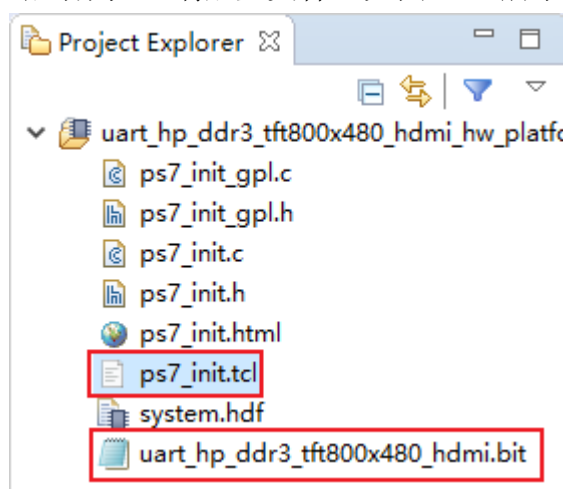


图 1-43 工作空间

由于本次设计不涉及 PS 端的用户程序，所以至此，我们便完成了整个工程设计，接下来便可以连接硬件，进行烧录验证了。

1.3 板级验证

本节将进行基于 ACZ702 开发板的串口传图设计的板级验证，设计需要使用到 PL 端的串口，因此需要通过 40pin 拓展接口外接 EDA 拓展板。

1.3.1 系统所需硬件

1. ACZ702 开发板
2. 电源线一根（可选）
3. Type-c 线两根
4. EDA 拓展板一个
5. HDMI 线缆一根
6. 支持 HDMI 接口的液晶显示器一台

1.3.2 板级验证需求

串口传图的板级验证，主要验证以下三个方面：

1. 能够正确地全屏点亮 TFT 屏和 HDMI 显示器，显示稳定。

2. 能否正确地显示颜色，即按照需求制定需要显示的颜色。
3. 能否正确地定位坐标，即实现在指定的位置显示对应的数据。

1.3.3 硬件连接

本次硬件连接如图 1-44 和图 1-45 所示：

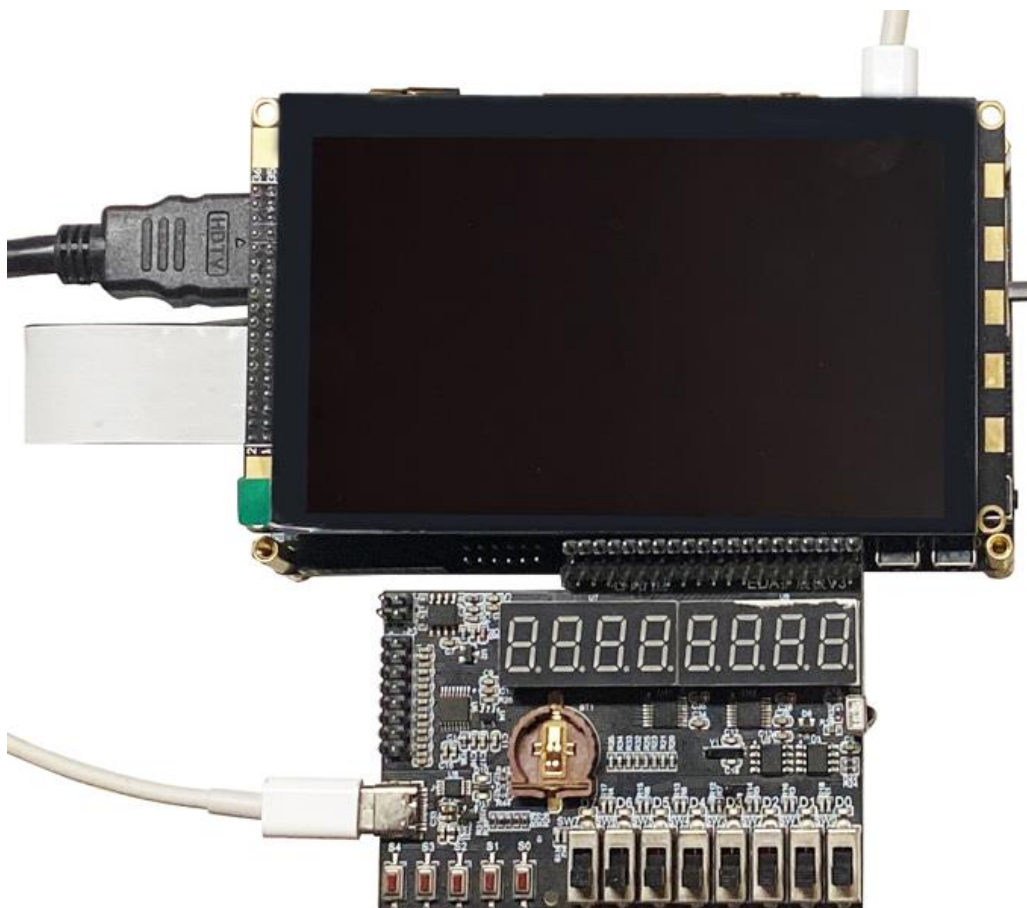


图 1-44 硬件连接

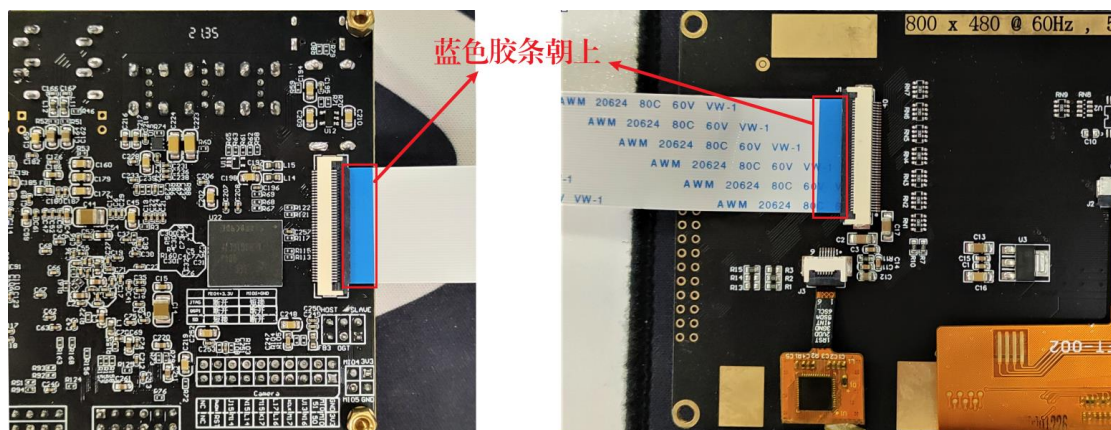


图 1-45 硬件连接（续）

使用 HDMI 线缆连接开发板与 HDMI 显示器，注意线缆一端连接开发板 HDMI 接口，一端连接显示器 HDMI 接口；使用软排线连接开发板与 TFT 屏，连接时注意软排线的蓝色胶条需要朝上；将 EDA 拓展板插接在开发板 40pin 接口上，正确连接时，接口应该一一接合；两个 type-c 线一根连接开发板 PS 侧调试接口和主机，一根连接 EDA 拓展板上的串口。

由于本次设计对供电要求不高，因此可以仅使用 type-c 线供电，在连接完硬件后将电源拨码开关拨动到对应启动侧，接下来便可以准备烧录了。

1.3.4 创建配置项

设计需要配置 PS，因此需要通过 SDK 进行烧录。右键单击工作空间中的 platform 文件夹，选择 Run As->Run Configurations...，创建配置任务，如图 1-46 所示：

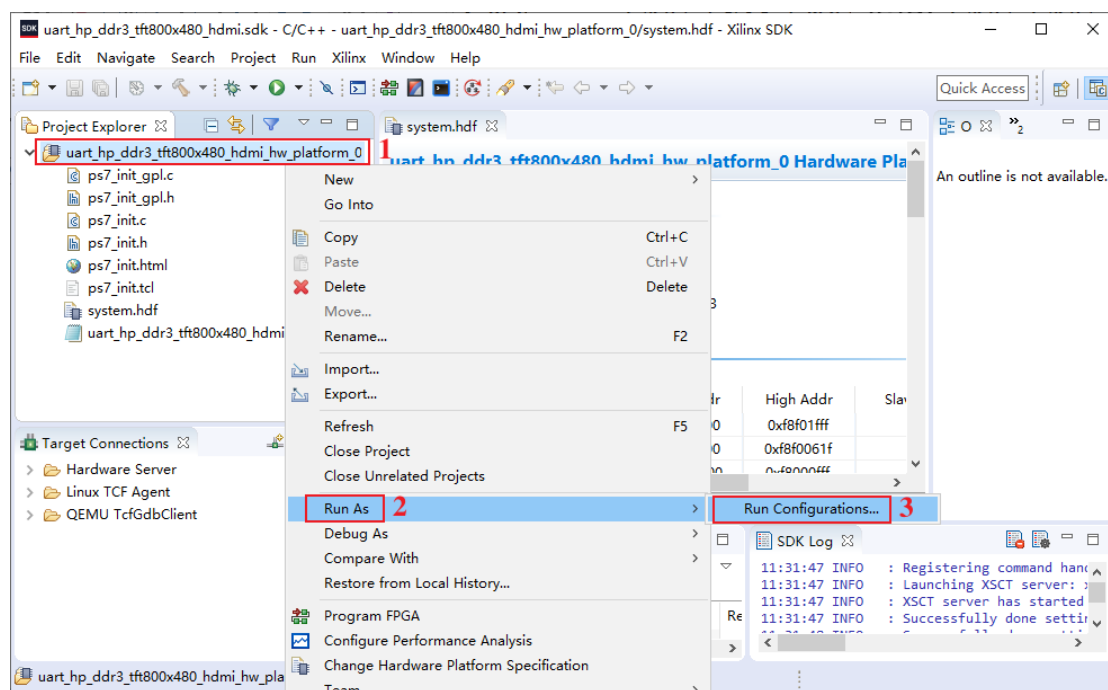


图 1-46 创建配置

在接下来弹出的界面中我们需要进行如下操作：

1. 双击左侧第三项创建一个 GDB 配置项
2. 点击 Search...添加 bit 文件和 ps7_init.tcl 脚本
3. 勾选 Reset entire system、Program FPGA、Run ps7_init、Run ps7_post_config
4. 点击 Apply，保存对配置任务的设置

5. 点击 Run，开始烧录

相关步骤操作如图 1-47 所示：

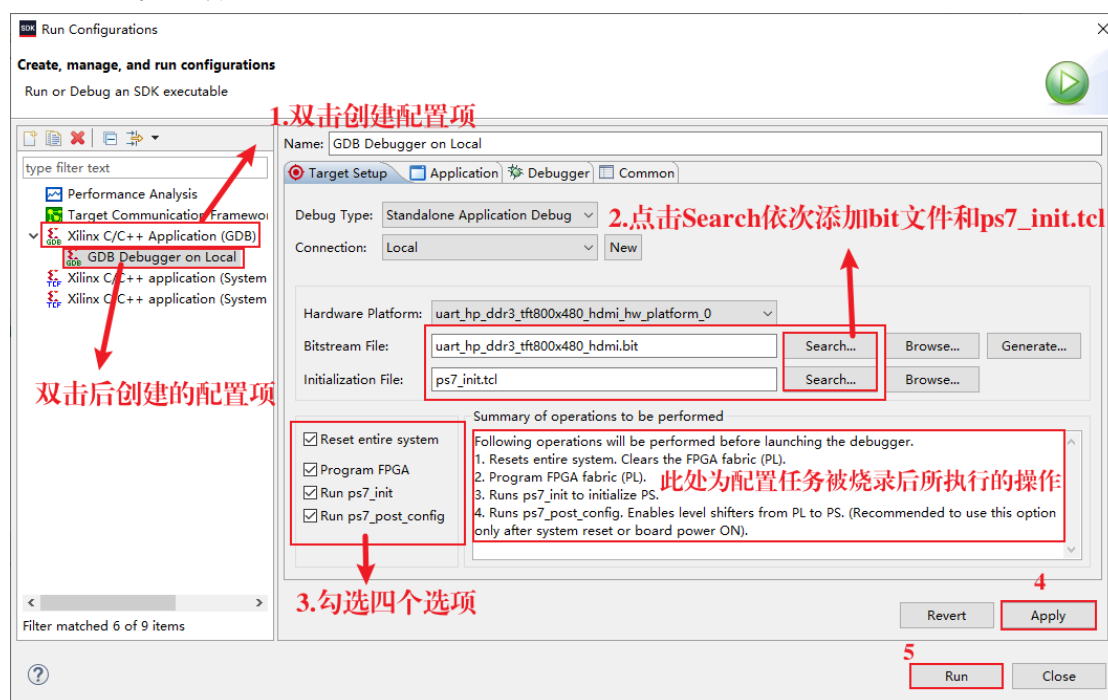


图 1-47 设置配置项

从图 1-47 中可以看到，软件给出了该配置任务被烧录后会执行的操作，依次是复位整个系统（PS 与 PL）、配置 PL、初始化 PS、使能 PL 到 PS 电平移位寄存器。如果设计包含 PS 端的用户设计，还会多出一个将用户程序（PS）烧录进指定处理器的操作。

在将配置任务烧录到开发板后，TFT 屏和 HDMI 显示器会呈现碎花屏，如图 1-48 所示：



图 1-48 显示效果（碎花屏）

这是正常现象，因为此时我们的串口并没有接收到数据，而显示模块已经开始工作。接下来我们就需要连接串口，通过串口传输图像数据。

1.3.5 串口传输图像数据

首先打开设备管理器，在端口中找到 EDA 拓展板上串口（CH340）的端口号，以笔者为例，端口为 COM11，如图 1-49 所示：

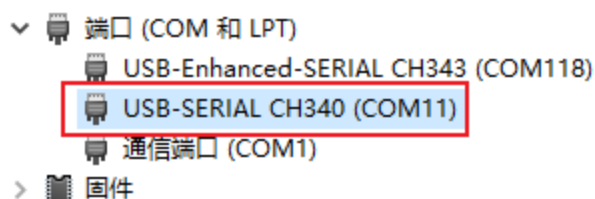


图 1-49 查询端口号

接下来打开小梅哥串口传图工具，该工具可以在开发板资料包的驱动和软件文件夹下的常用工具和附件文件夹中找到，如图 1-50 所示：

« 05_驱动和软件 » 02_常用工具和附件 » 小梅哥串口传图V3软件和图片素材 » 串口传图 »

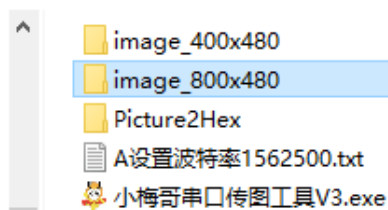


图 1-50 小梅哥串口传图软件和图片素材资料包

除了该小工具外，我们还为大家提供了一些测试用的图片，当然用户也可以使用 Picture2Hex 文件夹下的软件自行制作测试图片。

双击打开小梅哥串口传图工具，设置波特率为1562500，连接从设备管理器中查询到的端口号。随后设置图像大小为 800*480，点击打开图片选择一张 800*480 分辨率，RGB565 格式的照片，随后点击发送图片，如图 1-51 所示：

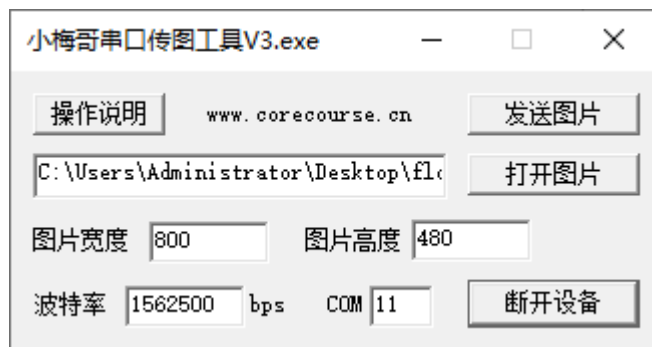


图 1-51 发送图片

这里选用了素材图片中的莲花照片作为实例，由于串口传输速率的限制，图像会有很明显的从上至下逐行扫描的过程，如图 1-52、图 1-53 和图 1-54 所示：



图 1-52 串口传图显示 1



图 1-53 串口传图显示 2

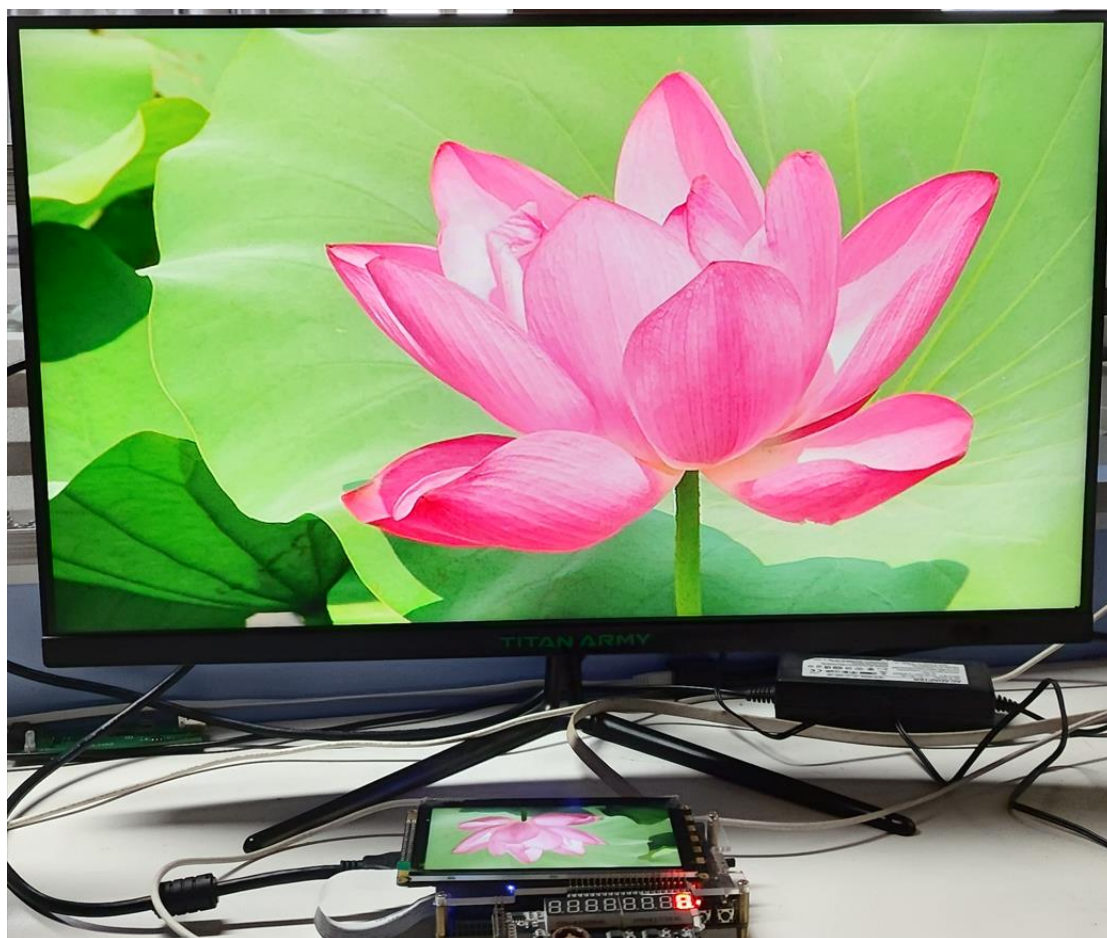


图 1-54 完整显示效果

可以看到，图像被完整显示到了 TFT 屏和显示器上，且画面正常，颜色层次清晰，证明本次设计成功。

1.4 总结

本章讲解了基于 DDR3 的串口传图帧缓存系统的设计与实现方法。对于 ACZ702 开发板 PL 端的设计来说，要想读写 DDR3，只能使用 AXI 总线通过 HP 接口对 PS 端读写数据，然后借由 PS 端的 DDR 控制器实现 DDR 读写操作。

作为 DDR 读写操作的第一个章节，本章中涉及到的许多知识和原理都将作为后续 DDR 相关实验的基础。建议读者跟随本章内容熟悉相关流程，理解并体会各个模块以及流程中各个步骤的功能和目的。