

# Z7-Lite 系列 开发教程

## Xilinx ZYNQ7000 FPGA

Rev. 1.0



## 版本记录

版本	时间	描述
Rev. 1.0	2020.7.28	第一版发布

## 版权声明：

XME/IME 系列核心模块、评估板及其相关知识产权由微相科技有限公司所有。

本文档由微相科技有限公司版权所有，并保留一切权利。在未经微相公司书面许可的情况下，不得以任何方式或形式来修改、分发或复制本文档的任何部分。

## 免责声明：

产品中所提供的程序源代码、软件、资料文档等，微相科技不提供任何类型的担保；不论是明确的，还是隐含的，包括但不限于合适特定用途的保证，全部的风险，由使用者来承担。



## 目录

版本记录.....	2
版权声明： .....	3
免责声明： .....	3
第 1 章 开发板简介与开箱检测.....	9
第 2 章 ZYNQ 介绍.....	10
2.1 ZYNQ 结构介绍 .....	10
2.2 PS-PL 互联： AXI 总线.....	12
2.3 ZYNQ 开发流程 .....	13
第 3 章 LED 实验及开发环境体验.....	16
3.1 实验简介 .....	16
3.2 实验原理.....	16
3.3 创建工程.....	16
3.4 程序设计 .....	23
3.5 添加管脚约束.....	29
3.6 下载验证 .....	35
3.7 实验总结 .....	37
第 4 章 Verilog 语法基础.....	38
4.1 模块声明 .....	38
4.2 变量类型 .....	38
4.3 赋值语句 .....	41
4.4 运算符 .....	43
4.5 条件判断语句 .....	46
4.6 总结 .....	48
第 5 章 按键控制 LED 灯 .....	49
5.1 实验简介 .....	49

5.2 创建工程 .....	49
5.3 程序设计 .....	53
5.4 下载验证 .....	57
5.5 实验总结 .....	58
第 6 章 按键检测与消抖 .....	59
6.1 实验简介 .....	59
6.2 实验原理 .....	59
6.3 创建工程 .....	60
6.4 程序设计 .....	61
6.5 添加管脚约束 .....	64
6.6 下载验证 .....	66
6.7 实验总结 .....	68
第 7 章 PWM 呼吸灯 .....	69
7.1 实验简介 .....	69
7.2 实验原理 .....	69
7.3 创建工程 .....	70
7.4 程序设计 .....	70
7.5 仿真验证 .....	74
7.6 下载验证 .....	78
7.7 实验总结 .....	79
第 8 章 片上资源之 PLL .....	80
8.1 实验简介 .....	80
8.2 创建工程 .....	80
8.3 生成 IP 核步骤以及参数设置 .....	81
8.4 程序设计及调用 IP 核 .....	85
8.5 对 IP 核及顶层进行仿真 .....	89

8.6 添加管脚约束.....	100
8.7 下载验证.....	102
8.8 实验总结.....	102
第 9 章 片上资源之 RAM .....	104
9.1 实验背景.....	104
9.2 创建工程.....	104
9.3 生成 IP 核及参数配置.....	105
9.4 程序设计以及 IP 核的调用.....	109
9.5 仿真验证.....	114
9.6 实验总结.....	120
第 10 章 片上资源之 FIFO .....	121
10.1 实验简介.....	121
10.2 实验原理.....	121
10.3 创建工程.....	123
10.4 配置 FIFO IP 核.....	123
10.5 程序设计.....	127
10.6 仿真验证.....	130
10.7 实验总结.....	133
第 11 章 LCD 显示实验.....	133
11.1 VGA 原理和时序.....	133
11.2 创建工程.....	135
11.3 程序设计.....	135
11.4 添加约束文件 .....	138
11.5 下载验证.....	140
11.6 实验总结.....	141
第 12 章 HDMI 输出彩条实验 .....	142

12.1 实验简介 .....	142
12.2 实验原理 .....	142
12.3 创建工程 .....	144
12.4 整体架构 .....	144
12.5 添加引脚约束 .....	152
12.6 下载验证 .....	153
12.7 实验总结 .....	154
第 13 章 HDMI 输出图像实验 .....	155
13.1 生成图像文件 .....	155
13.2 创建工程 .....	157
13.3 创建 ROM IP .....	157
13.4 整体结构设计 .....	160
13.5 下载验证 .....	162
13.6 实验总结 .....	163
第 14 章 AD_DA 转换模块 .....	164
14.1 实验简介 .....	164
14.2 创建工程 .....	168
14.3 使用 matlab 生成 coe 文件 .....	168
14.4 初始化 ram .....	170
14.5 程序设计 .....	175
14.6 添加约束文件 .....	176
14.7 下载验证 .....	178
14.8 实验总结 .....	180
第 15 章 ADDA 转换与 HDMI 显示 .....	181
15.1 实验简介 .....	181
15.2 创建工程 .....	181

15.3 功能设计.....	181
15.4 添加约束文件 .....	194
15.5 下载验证.....	195
15.6 实验总结.....	196
第 16 章 使用第三方 IP 完成 HDMI 输出.....	197
16.1 实验背景.....	197
16.2 下载第三方 IP .....	197
16.3 创建工程.....	199
16.4 添加第三方 IP 到 IP 仓库中.....	200
16.5 生成第三方 IP .....	201
16.6 程序设计.....	203
16.7 添加约束文件 .....	204
16.8 实验现象.....	205
16.9 实验总结.....	205

## 第1章 开发板简介与开箱检测

具体内容，请查看 01\_开发教程与开箱检测文件夹下的测试教程。

## 第2章 ZYNQ 介绍

Zynq-7000 系列芯片是基于 Xilinx 的可扩展处理器平台架构(Extensible Processing Platform, EPP)，将双核 ARM Cortex-A9 处理器和 FPGA 可编程逻辑单元集成在一颗单芯片中，从而构成了 PS(Processing System)加 PL(Programmable Logic)的单芯片 SoC 解决方案。

使用 Zynq 的初学者会经常有个疑问，Zynq 是一个嵌入 ARM 核的 FPGA，还是一个嵌入 FPGA 逻辑的 ARM？从它的启动引导来看，是 ARM 作为主导整个系统配置，这个和原来先 FPGA 启动去引导软核 MicroBlaze 启动完全不一样，基于这点多数人认为是 Zynq 是嵌入 FPGA 逻辑的 ARM；与此 Intel FPGA 的 Cyclone/Arria（集成 ARM）的 SoC 启动方式又和 Zynq 稍有差异，Intel SoC 中的 FPGA 部分可以自行配置，也可以通过 ARM 配置，那么从这点又不好说谁主谁从。从芯片设计者的角度，他们更关心的芯片的使用方便性以及芯片满足各种应用需求的灵活性；从使用者的角度，更关心的是如何实现自己的需求，至于谁主谁从，仁者见仁，智者见智。如果非要分出个谁主谁从，我个人觉得还是要从系统应用来分。

### 2.1 ZYNQ 结构介绍

Zynq 就是两大功能块，PS 部分和 PL 部分，说白了，就是 ARM(PS)和 FPGA(PL)部分。其中，PS 集成了两个 ARM Cortex-A9 处理器，AMBA 互连，内部存储器，外部存储器接口和外设。这些外设主要包括 USB 总线接口，以太网接口，SD/SDIO 接口，I2C 总线接口，CAN 总线接口，UART 接口，GPIO 等。PL 的部分就是 7 系列的 FPGA 部分，型号不同，资源也会有所不同，比如逻辑单元，内部存储容量，乘法器，高速接口等

以下是 Zynq 的结构框图，

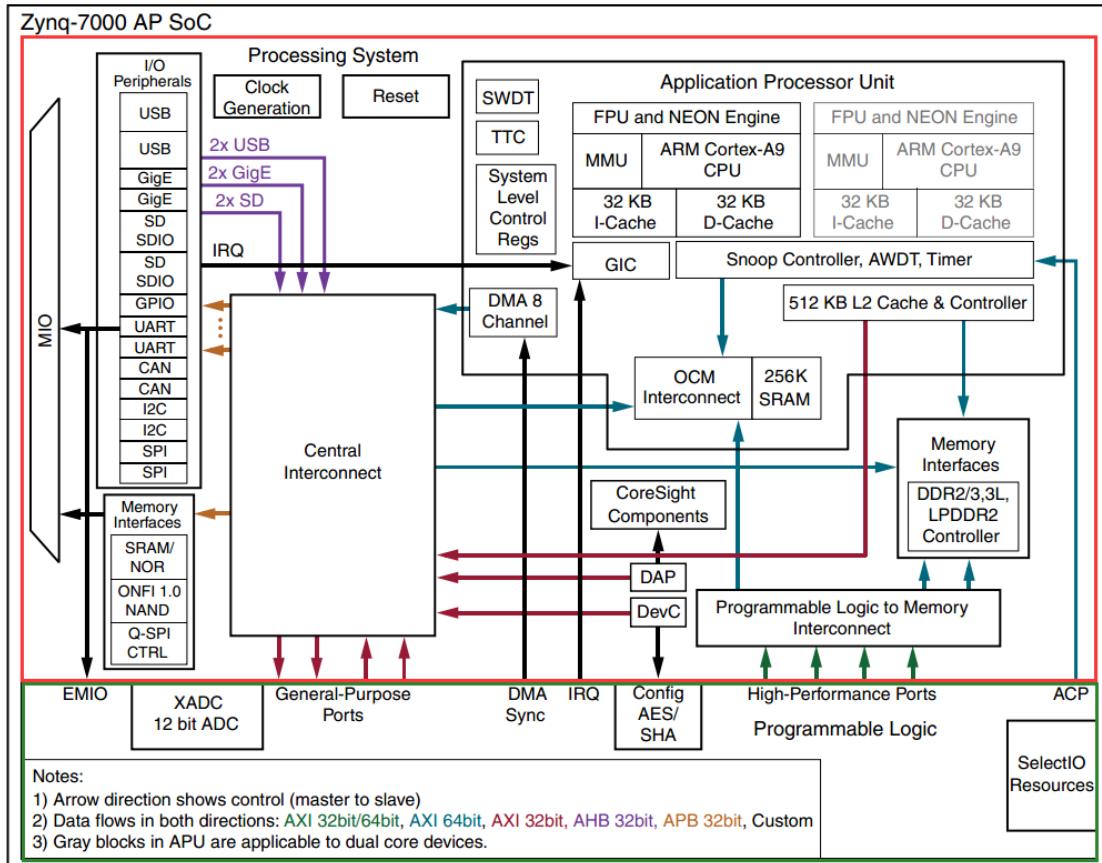


图 2-1

框图上面红色方框为 PS 部分，即 ARM 部分

框图下面蓝色方框为 PL 部分，即 FPGA 部分

这里需要介绍几个概念，后面会用到

**MIO** (Multiplexing IO, PS 可复用的 IO, 因为此 IO 管脚的配置在一定范围内有灵活性);

**EMIO** (Extensible MIO, 对 MIO 的扩展, 将 PL 的 IO 口扩展到 PS, 为 PS 直接所用);

**GIC** (General Interrupt Controller, 通用的中断控制器);

**IRQ** (Interrupt Request, 中断请求);

**OCM** (On Chip Memory, 片上存储);

**DMA** (Direct Memory Access, 直接存储访问)

**AXI** (Advanced eXtensible Interface) 是一种总线协议, Zynq 的 PS 和 PL 互联使用的是 AXI 总线

## 2.2 PS-PL 互联: AXI 总线

AXI 总线是连通 PS(programmable system)和 PL (programmable logic)的一个总线协议, 最终的形式就相当于 PL 块作为一个 ip core 挂载在 AXI 总线上, 然后由 PS 调用。Xilinx 从 6 系列的 FPGA 开始对 AXI 总线提供支持, 目前使用 AXI4 版本。

**ZYNQ 有三种 AXI 总线:**

**AXI4:** (For high-performance memory-mapped requirements.) 主要面向高性能地址映射通信的需求, 是面向地址映射的接口, 允许最大 256 轮的数据突发传输;

**AXI4-Lite:** (For simple, low-throughput memory-mapped communication ) 是一个轻量级的地址映射单次传输接口, 占用很少的逻辑单元。

**AXI4-Stream:** (For high-speed streaming data.) 面向高速流数据传输; 去掉了地址项, 允许无限制的数据突发传输规模。

**AXI 有三种接口:**

**AXI-GP 接口 (4 个):** 是通用的 AXI 接口, 包括两个 32 位主设备接口和两个 32 位从设备接口, 用过该接口可以访问 PS 中的片内外设。

**AXI-HP 接口 (4 个):** 是高性能/带宽的标准的接口, PL 模块作为主设备连接 (从下图中箭头可以看出)。主要用于 PL 访问 PS 上的存储器 (DDR 和 On-Chip RAM

AXI-ACP 接口（1 个）：是 ARM 多核架构下定义的一种接口，中文翻译为加速器一致性端口，用来管理 DMA 之类的不带缓存的 AXI 外设，PS 端是 Slave 接口。

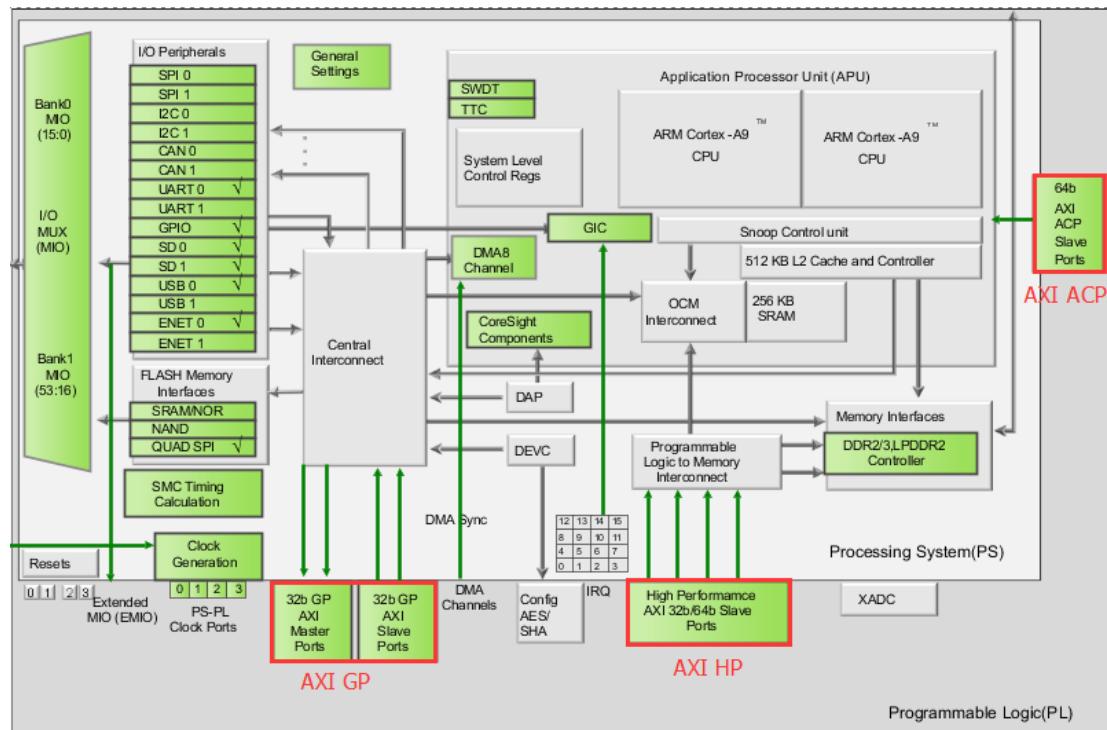


图 2-2

这三者的吞吐量比较如下（下图将 AXI GP 分了主从）：

Table 22-2: Theoretical Bandwidth of PS-PL and PS Memory Interfaces

Interface	Type	Bus Width (bits)	IF Clock (MHz)	Read Bandwidth (MB/s)	Write Bandwidth (MB/s)	R+W Bandwidth (MB/s)	Number of Interfaces	Total Bandwidth (MB/s)
General Purpose AXI	PS Slave	32	150	600	600	1,200	2	2,400
General Purpose AXI	PS Master	32	150	600	600	1,200	2	2,400
High Performance (AFI) AXI_H	PS Slave	64	150	1,200	1,200	2,400	4	9,600
AXI_ACP	PS Slave	64	150	1,200	1,200	2,400	1	2,400

图 2-3

## 2.3 ZYNQ 开发流程

ZYNQ 将 CPU 和 FPGA 集成到了一起，那么 ZYNQ 开发人员需要具备技能：

1. ARM 操作系统应用程序
2. 设备驱动程序
3. FPGA 硬件逻辑设计

就是说作为 ZYNQ 开发者，需要了解 Linux 操作系统，系统架构，FPGA 和 ARM 系统之间的硬件设计平台，即软硬协同设计，完成上面工作的可以是一个人，也可以是一个团队。

开发过程中所需的工具：

1. **Xilinx Vivado:** FPGA 部分的设计和开发，管脚和时序的约束，编译和仿真，实现 RTL 到比特流的设计。
2. **Xilinx SDK:** 在 Vivado 硬件设备的基础上，配置工具和库路径，编译器选项，JTAG 和闪存设置，调试器连接，裸板支持包，提供所支持的 Xilinx IP 硬核提供了驱动程序，SDK 支持 IP 硬核和处理器软件协同调试。可以使用高级 C 或者 C++ 语言开发调试 ARM 和 FPGA 系统。
3. **交叉编译工具作用:** 编译器，连接器，解析器组成的综合开发环境，由 binutils、gcc 和 glibcs3 个部分组成。Xilinx ZYNQ 平台的交叉编译工具基于 GNU 工具的编译和调试在虚拟机的 Ubuntu 操作系统环境下完成。
4. **软硬件调试工具:** FPGA-Xilinx 提供的 ChipScope Pro 作用：在线观察 FPGA 内部的信号 操作系统软件调试工具有 UNIX 下的程序调试工具 GDB 和远程调试工具 GDBserver.

开发流程也一定是先硬后软的流程：

1. Vivado 中创建项目，添加一个嵌入式的源文件
2. 硬件系统搭建，包括添加 PS 系统和外设，添加 PL 接口，IP
3. Vivado 创建顶层 HDL，编写必要的逻辑程序。
4. 添加约束，综合和编译生成.bit 文件
5. 导出硬件到 SDK

6. 在 SDK 里创建 FSBL 文件
7. 在虚拟机中生成 uboot.elf, bootloader 镜像
8. 由 FSBL, System.bit, uboot 生成 boot.bin
9. 生成 Ubuntu 的内核镜像文件 Zimage 和 Unbuntu 的文件系统 ramdisk,  
有时候还需要对接口驱动进行编写。
10. 将 Boot, 设备树, 内核镜像文件, 根文件系统放入 SD 卡启动板卡

以上是 Zynq 的通常使用开发流程, 实际使用中可能 有些步骤是用不上, 比如纯 PL 开发, 就不涉及 PS 以及 Linux 部分, 纯 ARM 裸机跑不太涉 PL 以及及 Linux 部分, 总之用户根据自己的实际情况来做开发。

## 第3章 LED 实验及开发环境体验

### 3.1 实验简介

在本章，将通过一个简单的 led 灯闪烁实验，介绍 Vivado 软件开发 Verilog 程序的基本流程，包括器件的选型，代码的编写，综合和编译，管脚分配，程序下载。

### 3.2 实验原理

在 Z7-Lite 开发板上有 3 个可供用户使用的 LED 灯，其中有一个 PS 端的 LED 灯，还有两个 PL 端的 LED 灯，如下图所示：

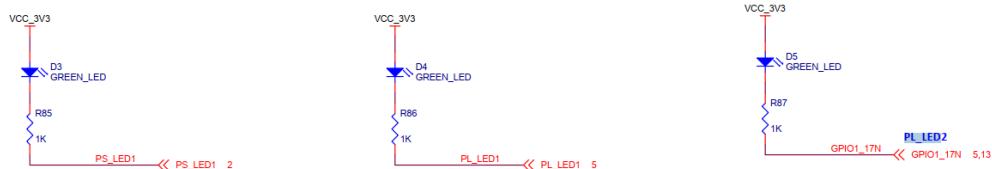


图 5-1

从原理图中可以看出，底板上的 led 灯为低电平有效，及当输出到 led 灯的电平为低时，led 灯点亮，高电平时 led 灯熄灭；

### 3.3 创建工程

- 1) 在开始菜单里选择 Xilinx Design Tools-->Vivado2018.3-->Vivado 2018.3 或双击桌面 Vivado 2018.3 图标，打开 Vivado 开发环境。

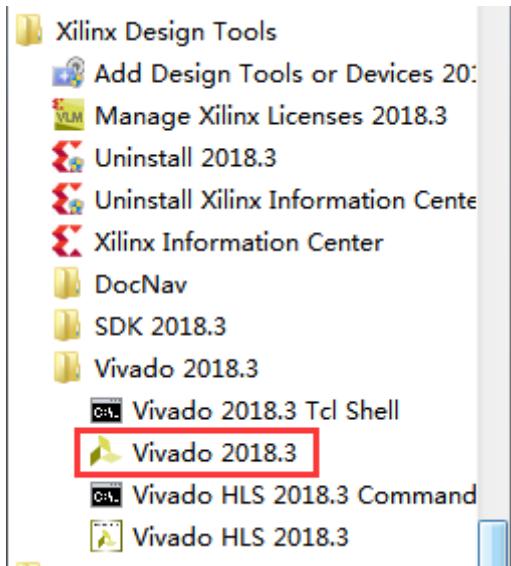


图 5- 2

2) 在开发环境里点击 Create New Project

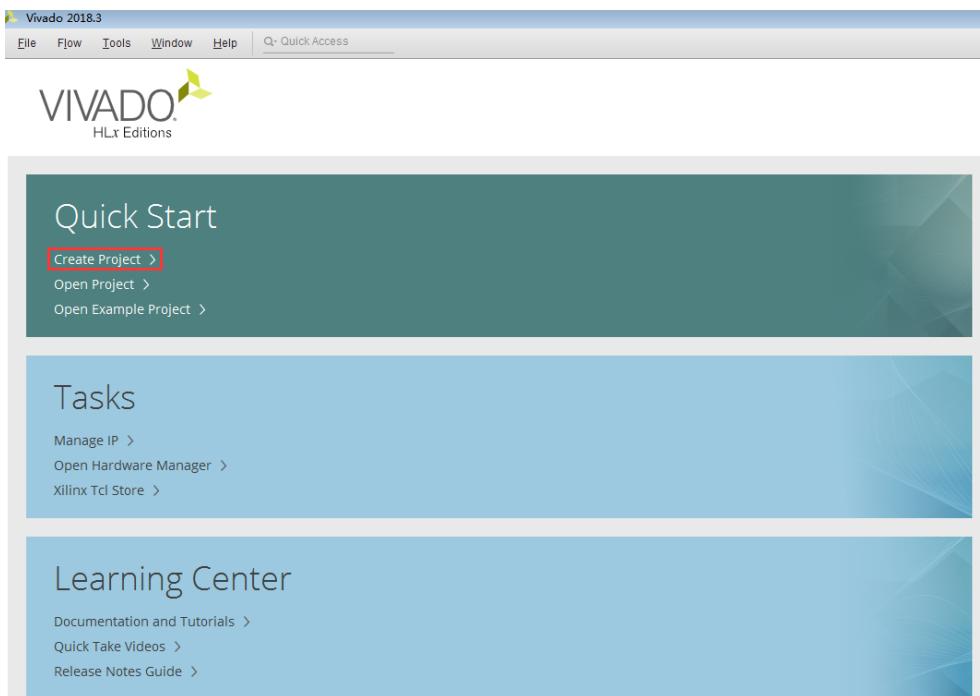


图 5- 3

3) 点击 Next

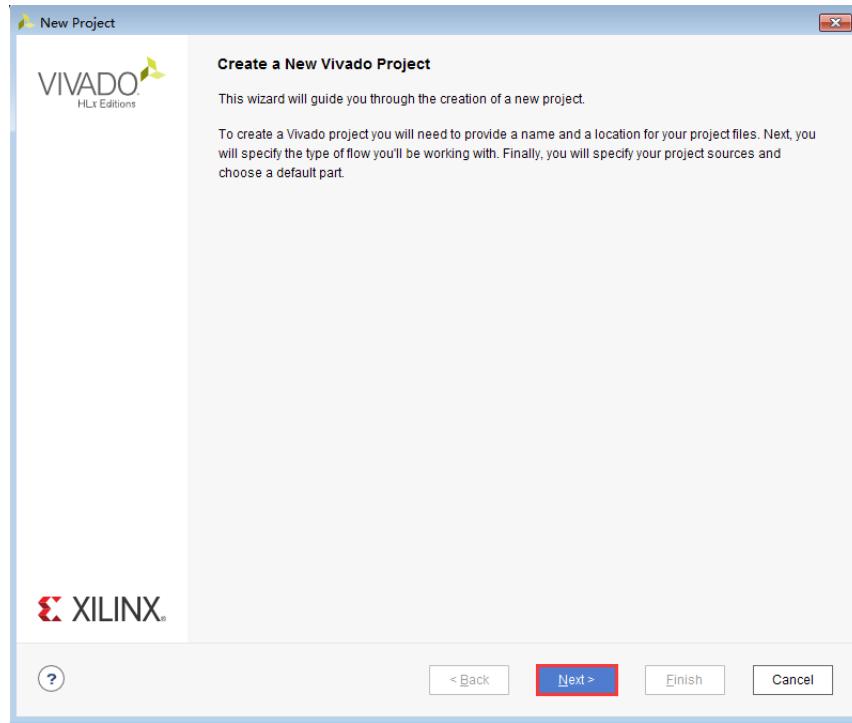


图 5-4

4) 输入项目名称，选择存放路径，然后点击 Next

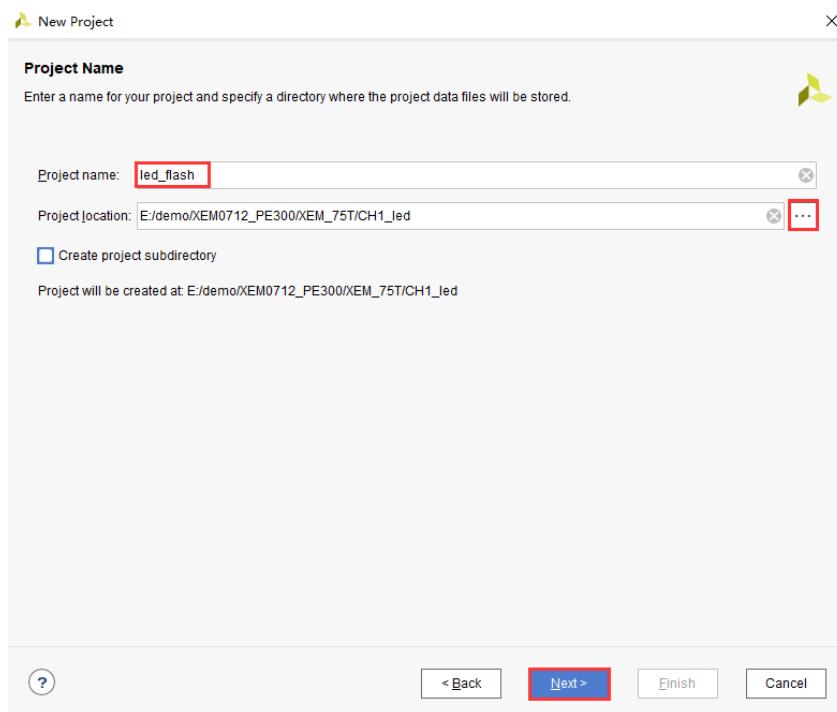


图 5-5

5) 选择 RTL Project, 然后点击 Next

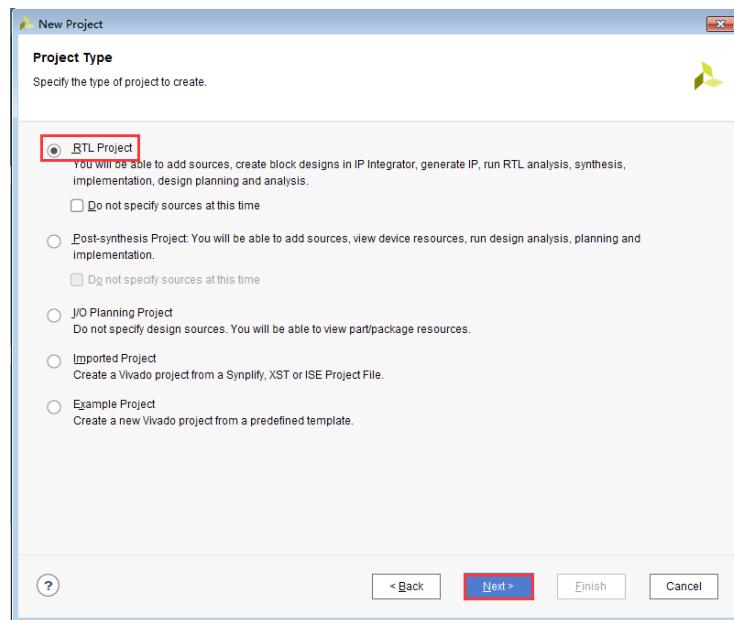


图 5-6

6) 这一步是添加 Source 文件, 这里先不添加, 点击 Next

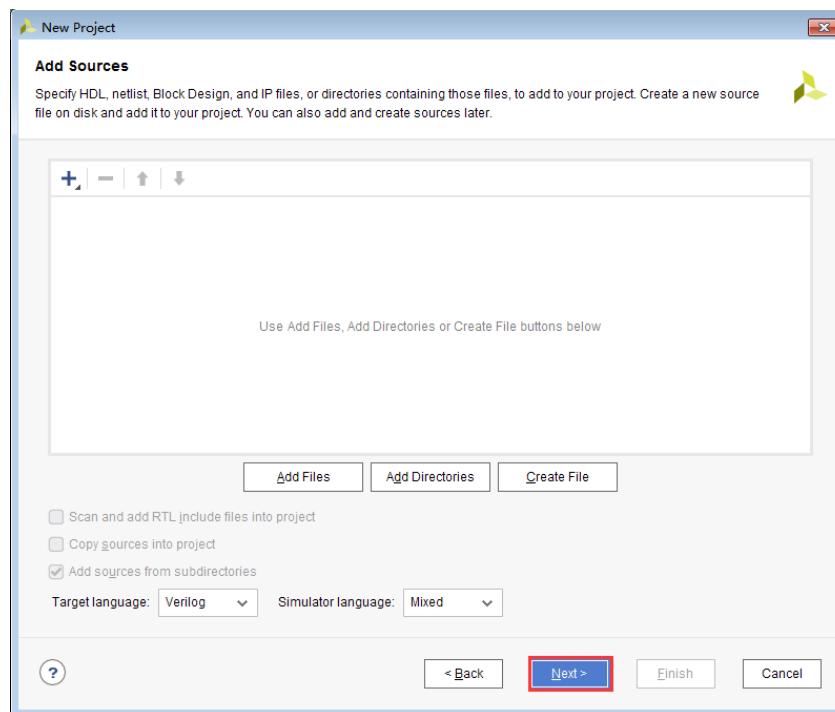


图 5-7

7) 这里是添加约束文件，同样暂时不添加，点击 Next

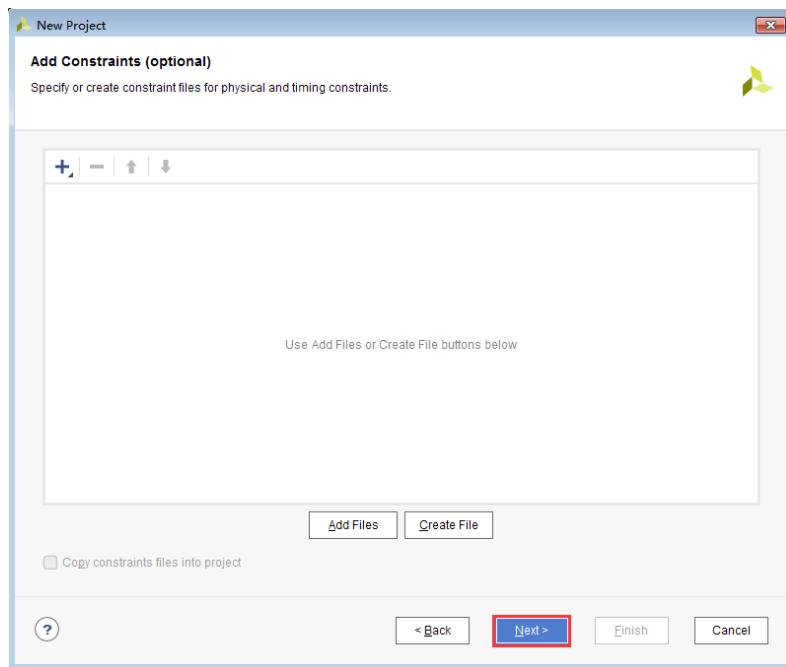


图 5-8

8) 选择器件，可以根据开发板上的芯片型号来做选择，由于芯片型号比较多，可以做一些筛选。

购买 Z7-Lite7020 的用户在 Family 栏里选择 Zynq-7000, Package 栏选择 clg400, 速度等级为 2，然后在下面的类表里选择相应的器件。

购买 Z7-Lite7010 的用户在 Family 栏里选择 Zynq-7000, Package 栏选择 clg400, 速度等级为 1，然后在下面的类表里选择相应的器件。

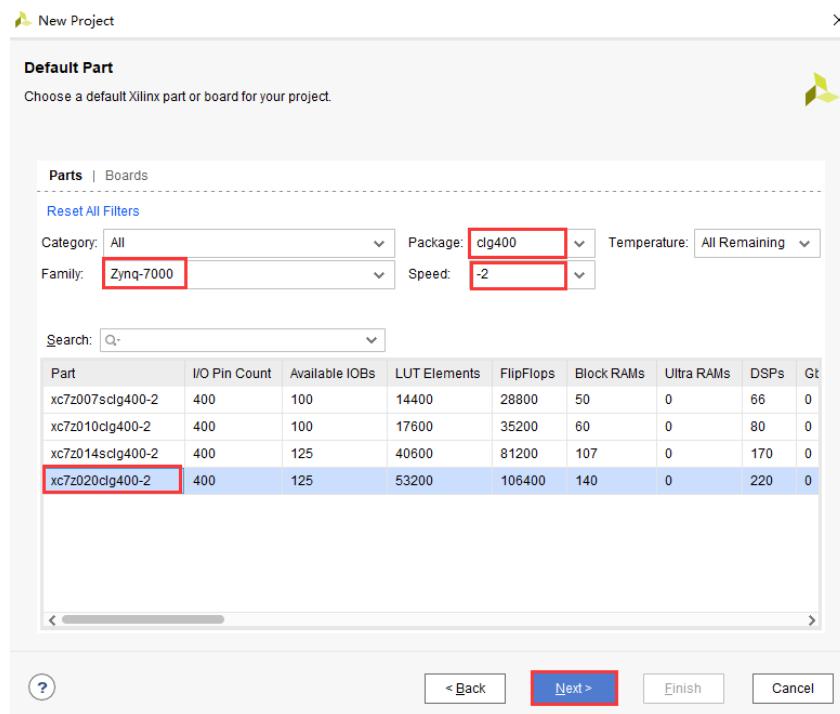


图 5-9

或者用户客根据自己 FPGA 芯片型号直接收索，**购买 Z7-Lite 的用户选中 xc7z020clg400-2**，**购买 Z7-Lite 的用户选中 Xc7z010clg400-1**

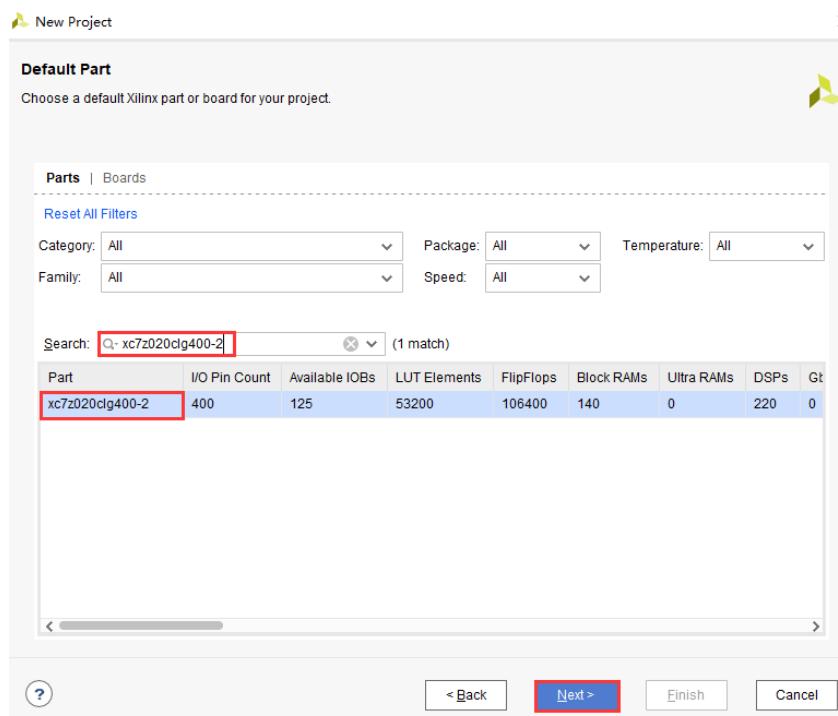


图 5-10

9) 再次确认一下型号是否选对，没有的话点击 Finish，完成项目创建

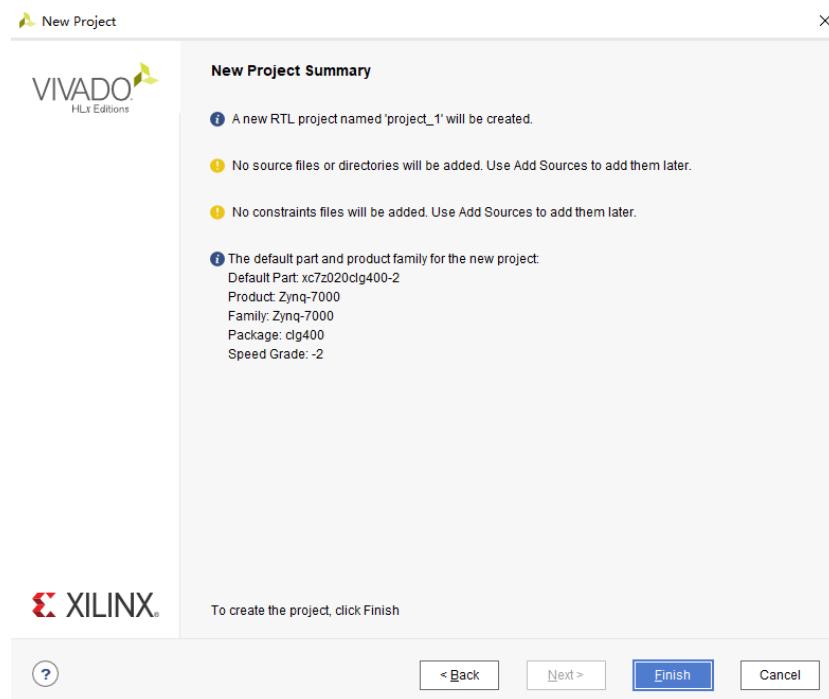


图 5-11

10) 创建后的项目

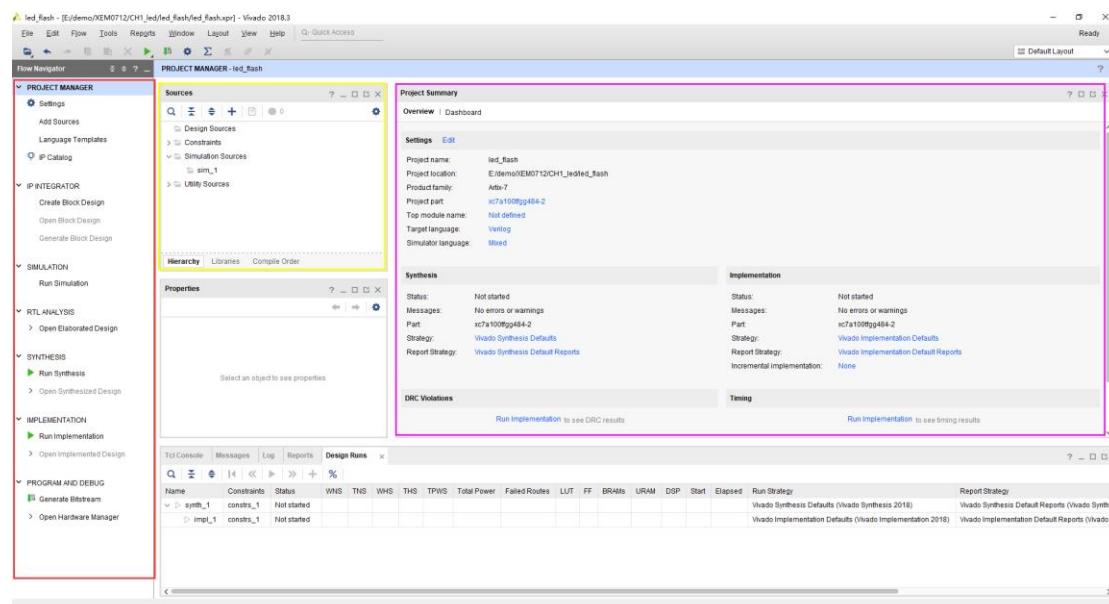


图 5-12

其中，红色框为设计流程导航窗，从上到下是 FPGA 开发的各个环节。黄色

框为工程数据文件窗口，可以查看工程的层次结构，编辑工程文件属性。紫色框为主工作窗，根据不同的 layout 有不同的显示内容。蓝色框为结果显示窗，显示各个环节的执行结果。黑色框可在不同的 layout 之间切换。

### 3.4 程序设计

- 1) 在做程序设计前，我们需要先创建一个.v 的程序文件，左侧边栏 Project Manager 下，点击 Add Source 图标

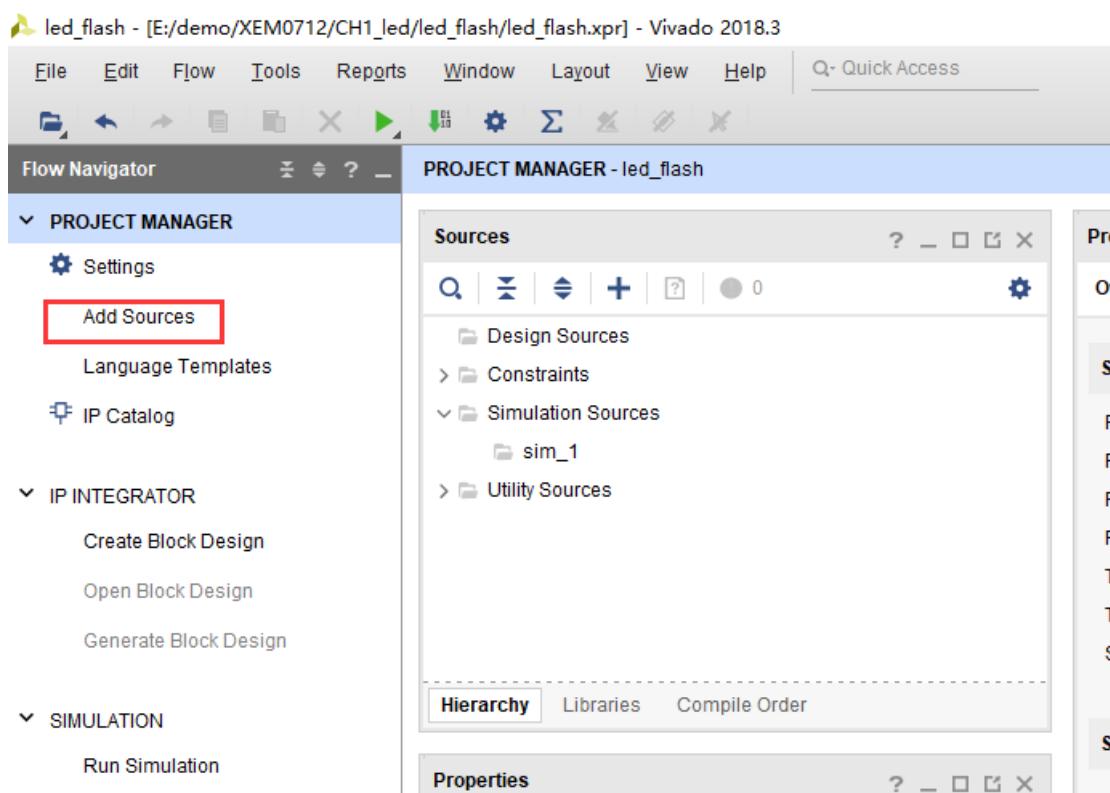


图 5- 13

- 2) 选择 Add or Create Design Source，然后点击 Next

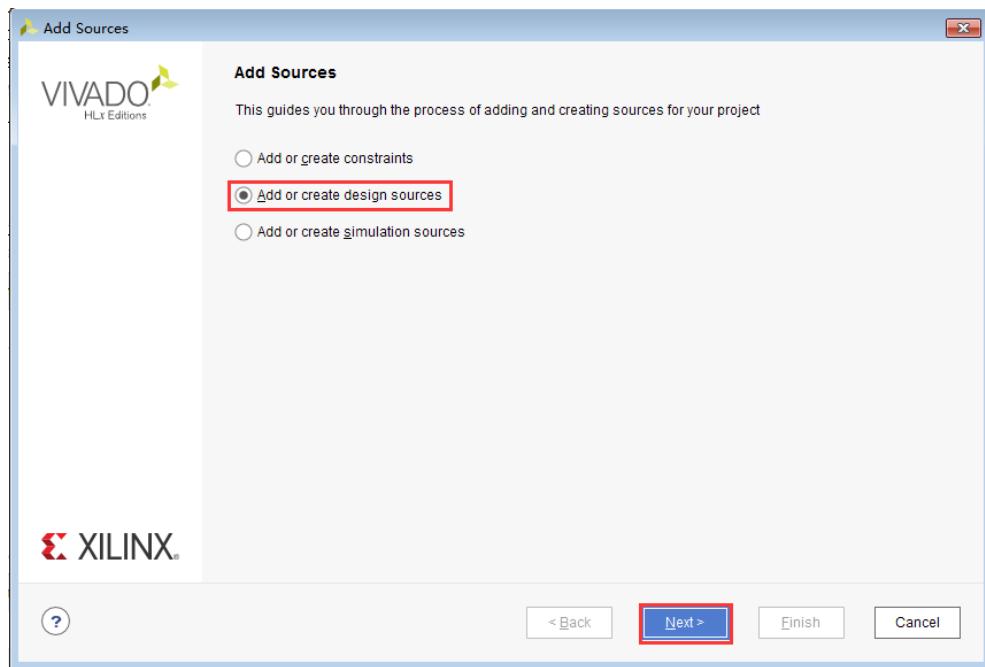


图 5- 14

3) 用户可以添加已有的程序，这里我们还有程序，选择 Create File

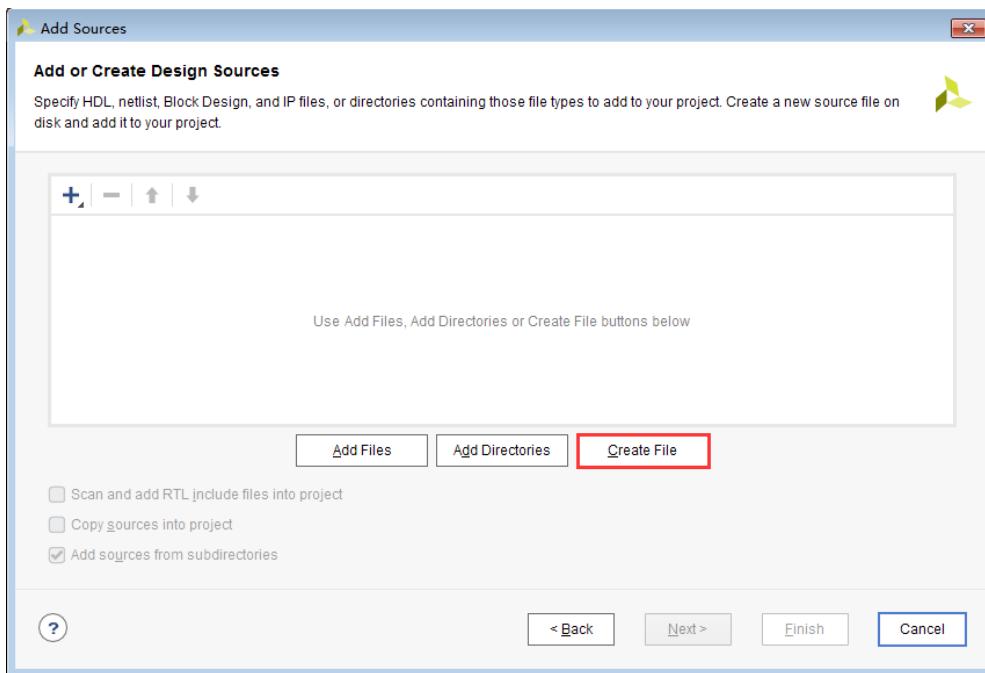


图 5- 15

弹出对话框，输入文件名 led，点击 OK

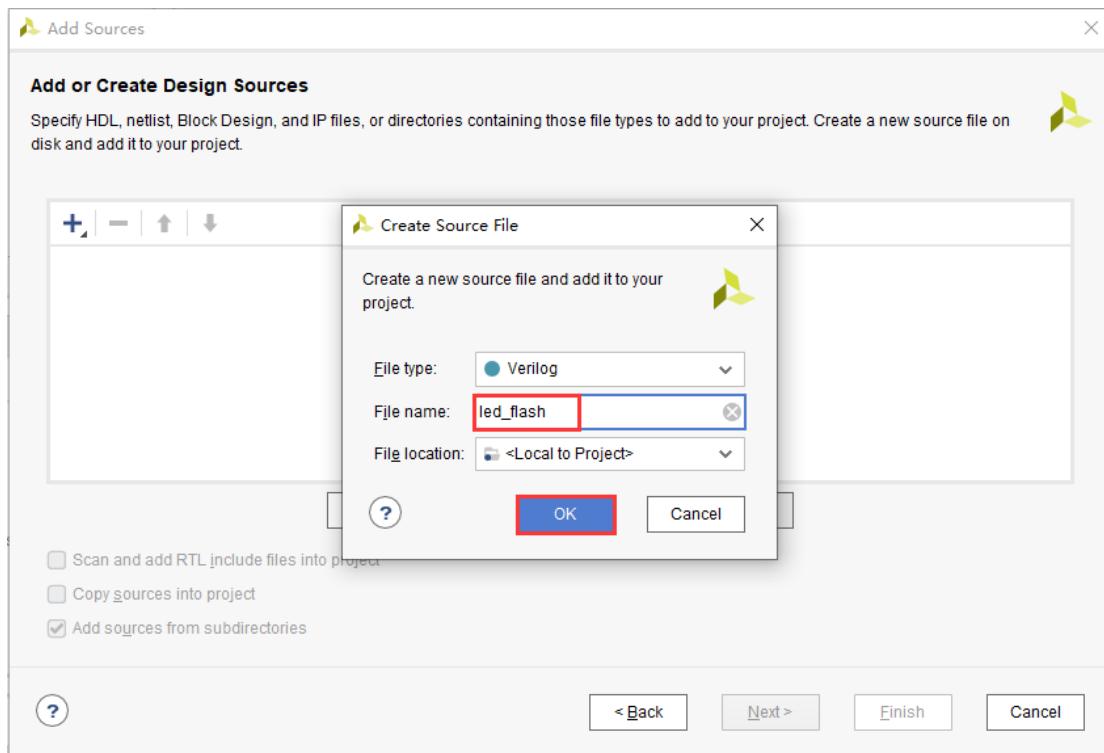


图 5-16

4) 点击 Finish 完成

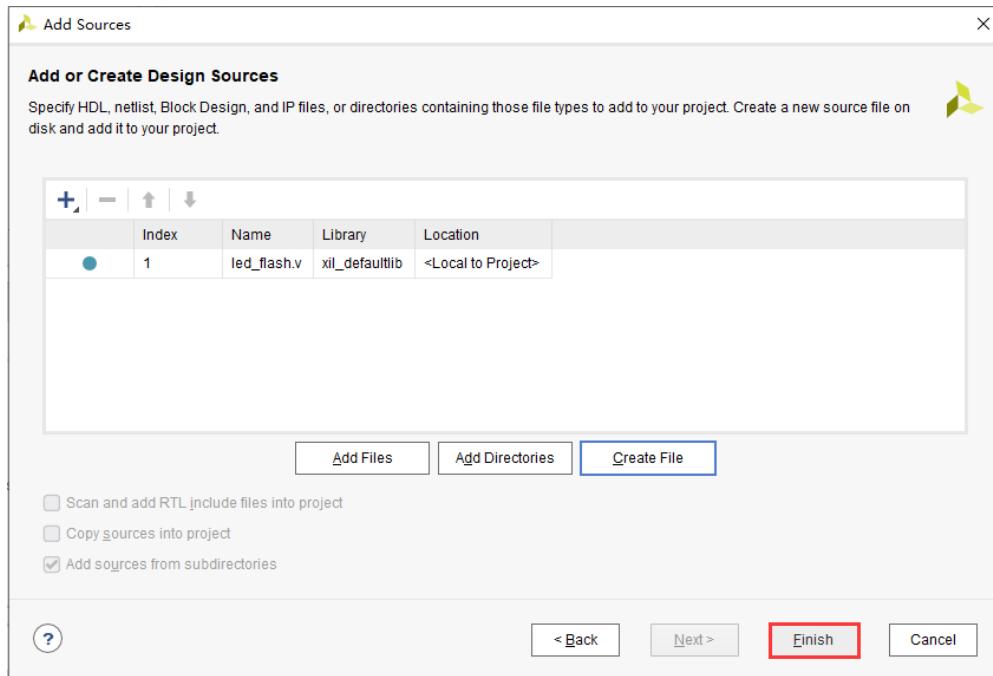


图 5-17

- 5) 向导会提示您定义 I/O 的端口，这里不定义，后面程序中编写即可

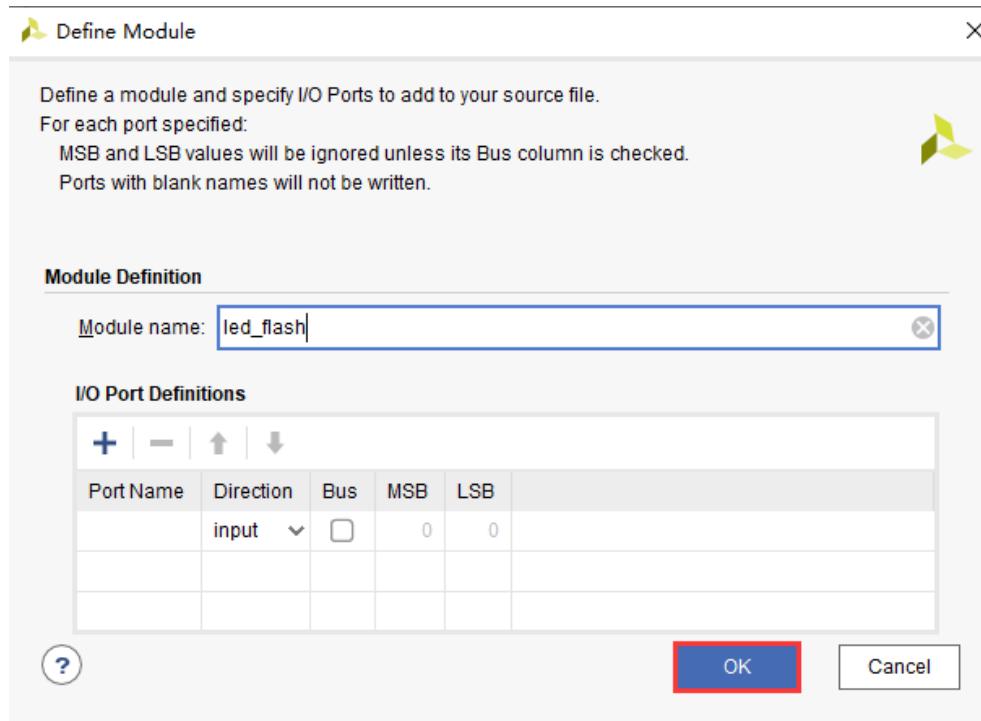


图 5-18

点击 Yes

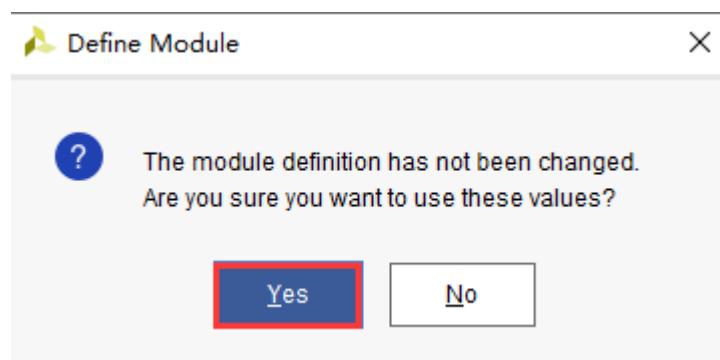


图 5-19

- 6) 此时在 Project Manager 界面下面的 Design Source 下有 led\_flash.v 文件，并且自动成为项目的 top 模块

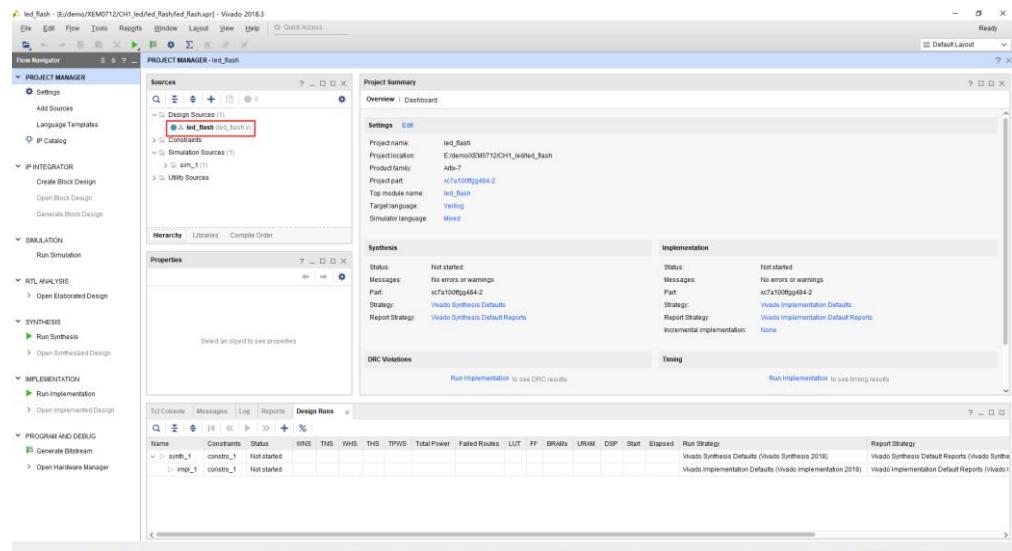


图 5- 20

7) 接下来就是编写本次实验主要的部分，完成流水灯的部分。代码如下：

Z7-Lite 系列代码如下：

```

1 module led_flash(
2     input  wire          clk      ,//System differential clock
3     input  wire          rst_n    ,//system reset
4     output reg [1:0]     led      //output led
5 );
6
7 //=====
8 //internal singals
9 //=====
10
11
12 reg   [27:0]  cnt      ;//0.25s counter
13 wire           add_cnt   ;
14 wire           end_cnt   ;
15
16
17 //-----cnt-----
18 always @(posedge clk or negedge rst_n)begin
19     if(rst_n==1'b0)begin
20         cnt <= 'd0;
21     end
22     else if(add_cnt)begin

```

```
23      if(end_cnt)
24          cnt <= 'd0;
25      else
26          cnt <= cnt + 1'b1;
27  end
28 end
29
30 assign add_cnt = 1;
31 assign end_cnt = add_cnt && cnt== 10_000_000 - 1;
32
33
34 //-----led-----
35 always  @(posedge clk or negedge rst_n)begin
36     if(rst_n==1'b0)begin
37         led <= 2'b10;//reset state
38     end
39     else if(end_cnt)begin
40         led <= {led[0], led[1]};
41     end
42     else begin
43         led <= led; //keep the state
44     end
45 end
46
47
48 endmodule
```

就代码中的几个注意点做以下说明：12 行声明了一个计数器，用来计数时间，通过该计数器来控制其他逻辑。18 到 31 行是计数器的实现代码，每个复位时计数器保持为 0，复位计数后，每个时钟周期计数器加一，当计数到最大值时清零计数器并且重新开始计数。

35 到 45 行是控制 led 灯流水的操作。在复位的时候，led 的最低位有效，每当计数器计数到最大值的时候，led 完成移位，这里使用了位拼接的方式来完成 led 的移位操作。可以看到代码十分的简单。

- 8) 编写好后按 Ctrl+S 保存代码。

### 3.5 添加管脚约束

和之前 ISE 软件不同， Vivado 使用的约束文件格式是 **.xdc** 文件。**.xdc** 文件里主要是定义管脚约束,时钟约束，以及其他时序约束。这里我们需要对 **led.v** 程序中的输入输出端口分配到 **FPGA** 的管脚上，这就需要将管脚分配的**.xdc** 文件添加到工程中

- 1) 我们需要先创建一个**.xdc** 的约束文件，左侧边栏 Project Manager 下，点击 Add Source 图标

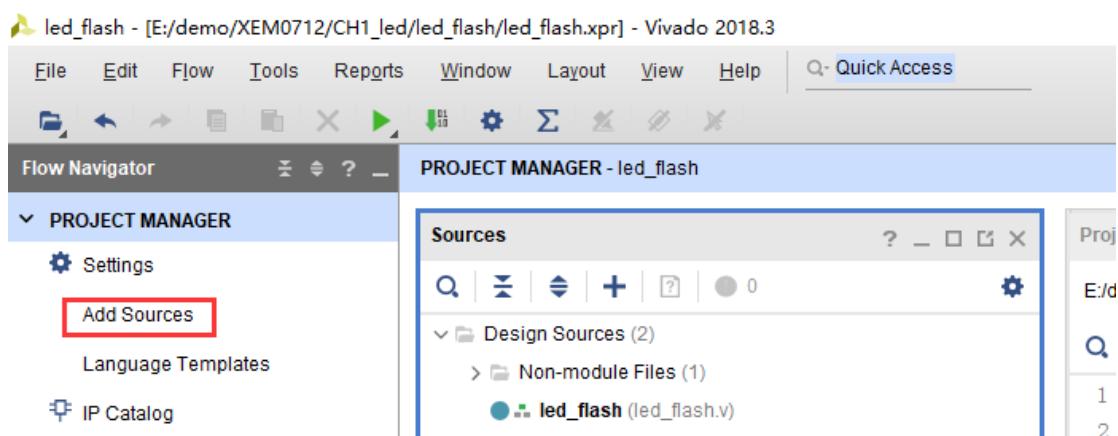


图 5- 21

- 2) 选择 Add or create constraints，点击 Next

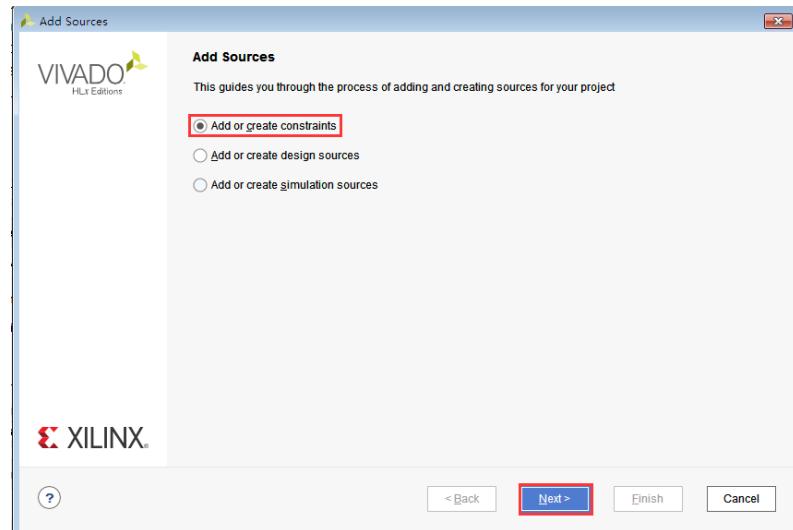


图 5- 22

3) 点击 Create File 按钮，在 file name 输入 top\_pin 文件名，点击 OK

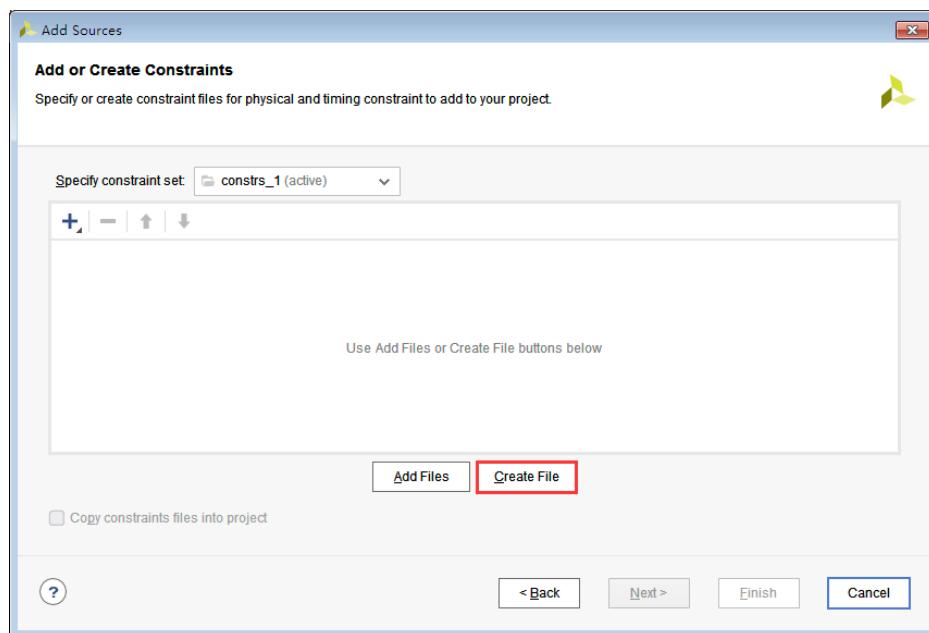


图 5- 23

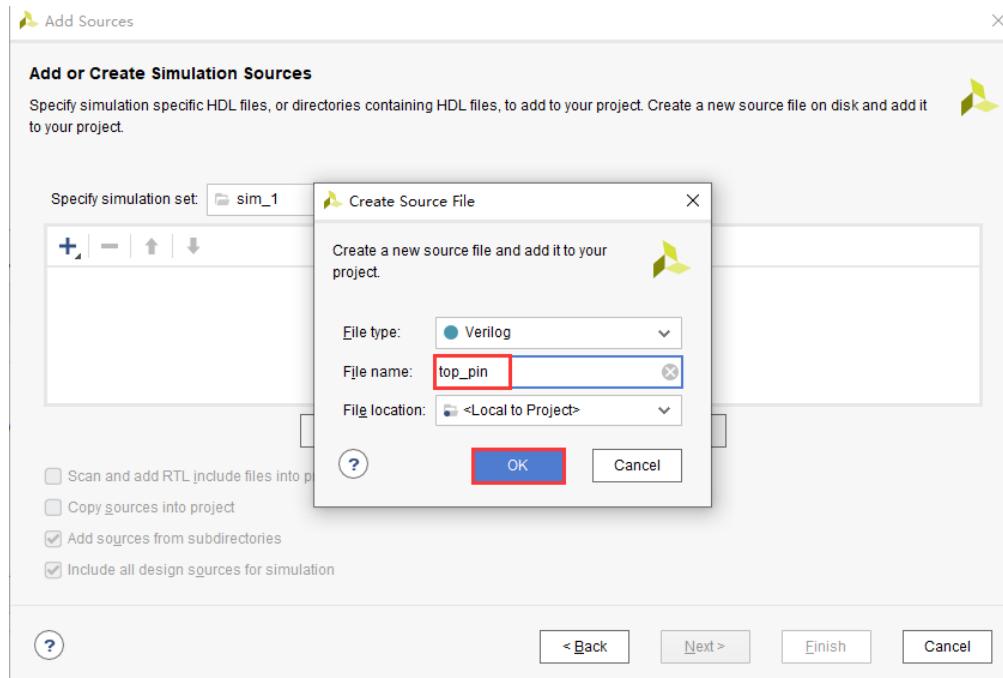


图 5- 24

4) 点击 Finish 按钮，完成约束文件的创建

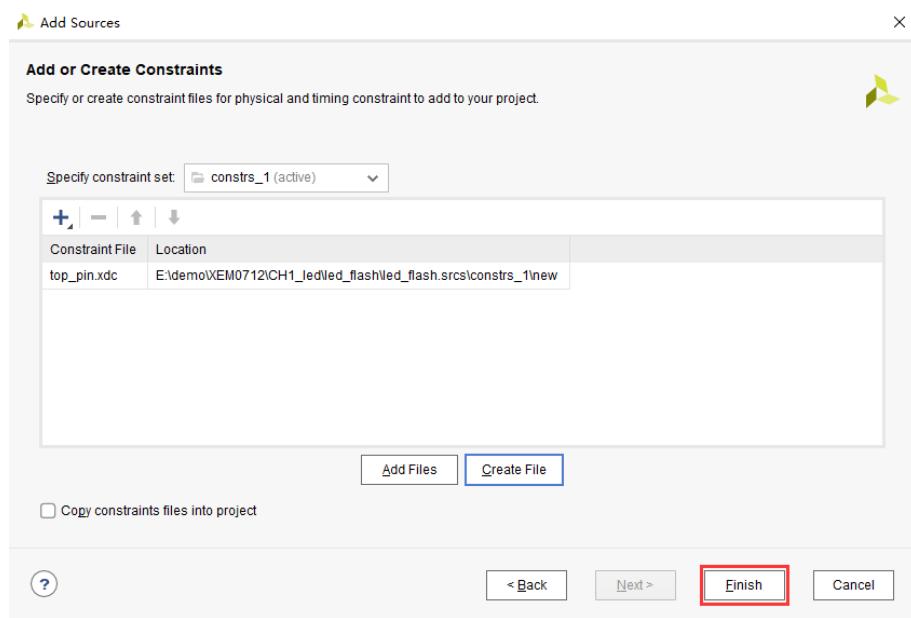


图 5- 25

5) 建好之后，在约束文件组下有个 top\_pin.xdc

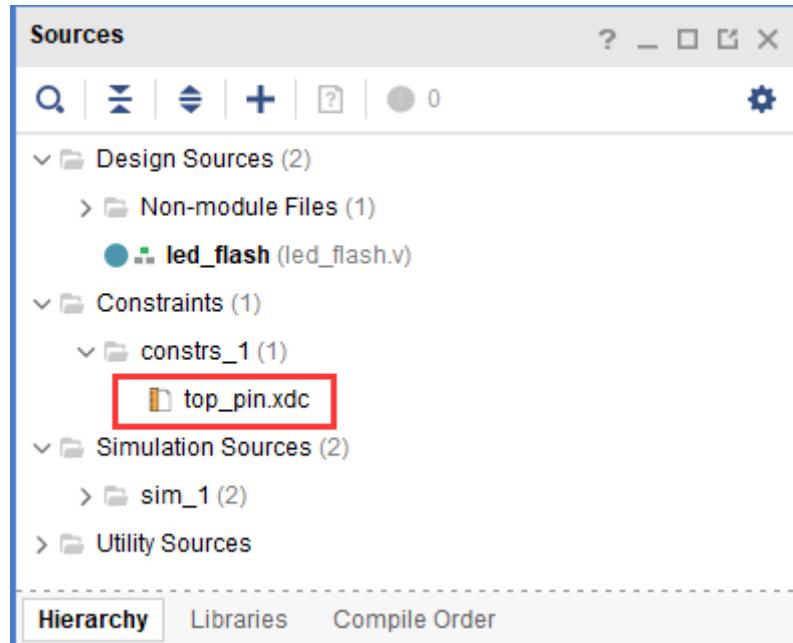


图 5-26

- 6) 打开 top\_pin.xdc 文件，添加如下约束，约束文件如何写，可以参考 xilinx 的相关文档，下面来介绍一下最基本的 XDC 编写的语法，普通 IO 口只需约束引脚号和电压，管脚约束如下：

```
set_property PACKAGE_PIN "引脚编号" [get_ports "端口名称"]
```

电平信号的约束如下：

```
set_property IOSTANDARD "电压" [get_ports "端口名称"]
```

这里需要注意文字的大小写，端口名称是数组的话用{}刮起来，端口名称必须和源代码中的名字一致，且端口名字不能和关键字一样。以 led[0]的约束为例：

set\_property PACKAGE\_PIN G14 [get\_ports {led[0]}] 是为了绑定 led[0]的管脚

set\_property IOSTANDARD LVCMOS33 [get\_ports {led[0]}]是为了设置 led[0]的电平约束。

Z7-Lite 系列的约束文件如下：

```
#####
# clock define#####
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports clk]

#####
# key define #####
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
#####

#####LED define#####
set_property PACKAGE_PIN P15 [get_ports {led[0]}]
set_property PACKAGE_PIN U12 [get_ports {led[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports {led[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
```

- 7) 写好后，点击保存。
- 8) 依次点击 1 综合，2 实现，3 生成.bit 文件，或者直接双击 3，其效果是一样的。

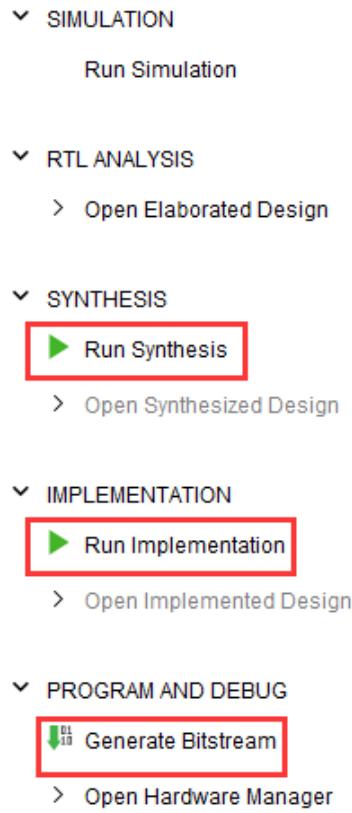


图 5- 27

生成.bit 文件后，可以在 Project summary 里看到相关信息，如整个项目使用资源的情况，功耗，Timing 等信息。

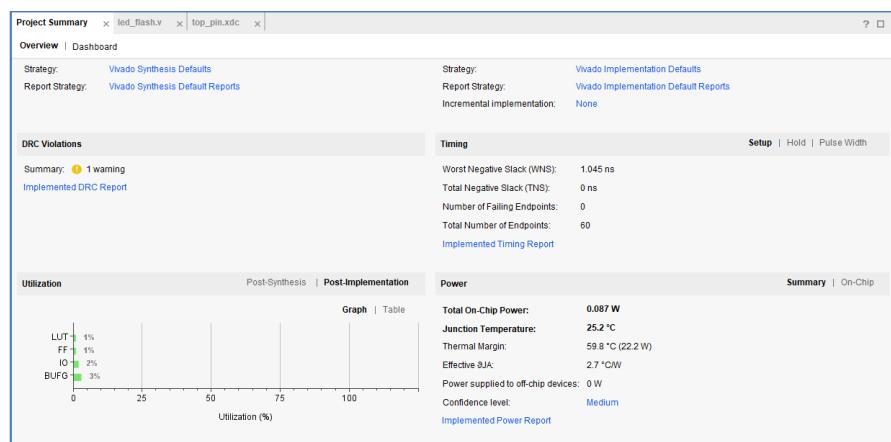


图 5- 28

### 3.6 下载验证

- 生成完.bit 文件就可以下载到 FPGA 芯片中，看一下 LED 的运行结果。将下载器连接到 JTAG 口，给板子上电，点击 Open Hardware Manager，在 Hardware Manager 窗口里点击 Open target，然后点击 Auto Connect.



图 5-29 Z7-Lite 连接方式

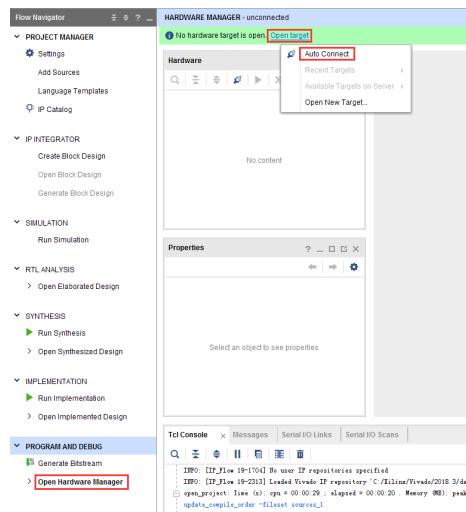


图 5-30

- 连接成功后，在 Hardware 窗口下会显示当前扫描到的设备，如下图扫描到的设备是 xc7z020 与我们在一开始的时候选择的器件型号相吻合。

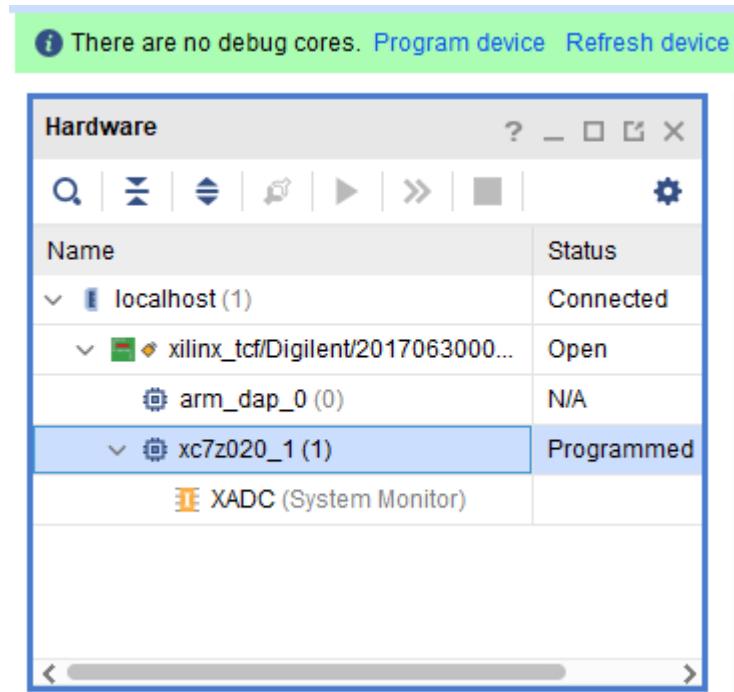


图 5- 31

3) 点击 Program device，下载程序，弹出对话框，直接点 Program 按钮。

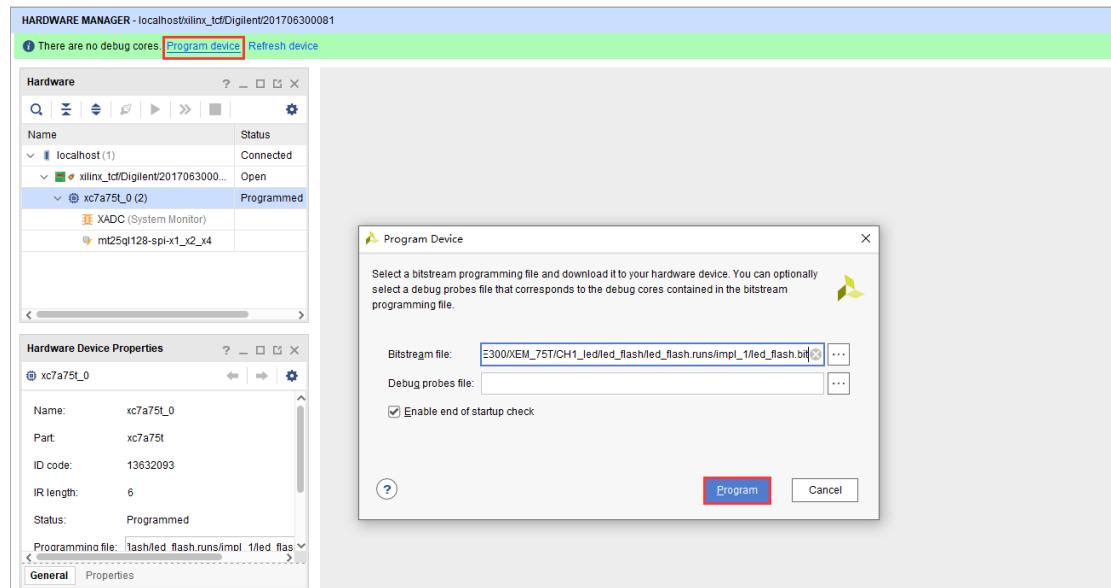


图 5- 32

4) 下载完成后可以看到 2 个 led 灯开始进行流水操作。

### 3.7 实验总结

本次实验初次使用 FPGA 完成了一个简单的流水灯实验，通过本次实验，了解了在 Vivado 中如何来进行开发，整个流程还是有许多注意事项的，在后续的实验中，经过多次练习，最终都能很熟练的掌握软件的使用。

## 第4章 Verilog 语法基础

### 4.1 模块声明

在 FPGA 开发中,是以模块为基础的,每一个可综合的.v 文件都是一个模块,模块由 module——endmodule 来声明,在这两个关键字内部,完成模块功能的实现。下面以一个简单的与门来演示如何声明模块。

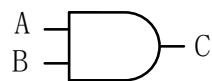


图 6-1

```

1 module and_gate(
2     input wire a ,
3     input wire b ,
4     output wire c
5 );
6
7 assign c = a & b;
8
9 endmodule

```

可以看到,在第一行声明了一个与门模块,模块的实现在 module 和 endmodule 之间。Module 后跟着的是模块名,模块名后面是模块的接口的信号,其中输入信号用 input 指示,输出信号用 output 指示。声明的信号之间用逗号隔开。模块声明结束后,需要在括号后跟上分号。

### 4.2 变量类型

在 Verilog 开发中有两种数据类型,一种是 wire(线),另一种是 reg(寄存器)。在数字电路中信号只有两种形态,一种是传输,一种是存储,传输是通过连接线,存储是用寄

存器，因此也就清楚了在 Verilog HDL 中常用 wire 和 reg 变量了。wire 和 reg 变量模型如下图所示：

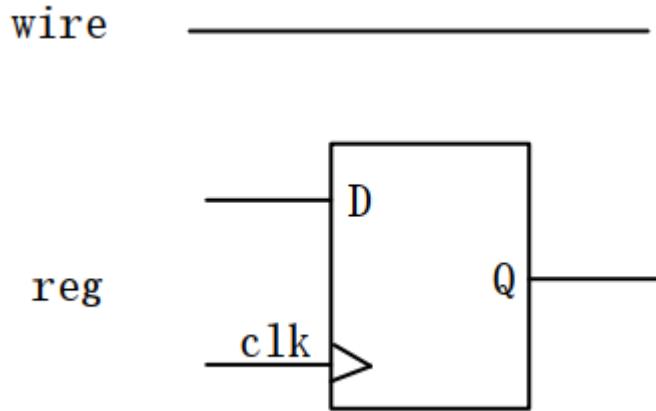


图 6-2

wire 型变量在物理结构上只是一根线，在 Verilog HDL 描述时，对线型变量赋值用 assign 即可，相对比较简单。

reg 型变量左端有一个输入端口 D，右端有一个输出端口 Q，并且 reg 型存储数据需要在 clk(时钟)沿的控制下完成。clk 也即是常说的方波，它是由晶振产生，是我们描述数字电路时最基本的时间单元，它的周期固定，占空比一般为 50%（即高电平占整个周期的比例）。clk 的低电平用数字 0 表示，高电平用 1 表示，从低电平转变到高电平的过程叫做上升沿，从高电平转变到低电平的过程叫做下降沿。

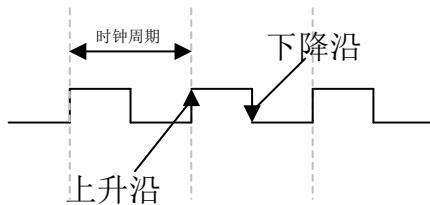


图 6-3

在对 reg 型变量进行赋值时，必须在 always 块内完成，可以选择用时钟上升沿，也可以选择时钟下降沿，具体用上升沿还是用下降沿可以根据需要所定。我们可以试着将前面模块声明中的输出改为 reg 型变量，完成两个输入相与之后的结果传输给寄存器变量 c，RTL 电路图如下图所示。

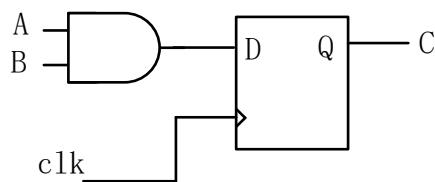


图 6-4

其 verilog 代码如下：

```

1 module and_gate(
2     input    wire    clk ,
3     input    wire    a   ,
4     input    wire    b   ,
5     output   reg     c
6 );
7
8 always @(posedge clk) begin
9     c <= a & b;
10 end
11
12 endmodule

```

第二行为输入的时钟，由晶振提供

由于输出变量 c 在 always 中赋值，所以其必须是 reg 类型

第 8 到 10 行使用 always 块对 reg 类型变量进行赋值，@(posedge clk)表示的是，每当遇到 clk 上升沿时执行 always 块中的语句。

第 9 行的赋值符号( $<=$ )是非阻塞赋值，在 Verilog 中由两种赋值方式，一种是非阻塞赋值( $<=$ )，另一种是阻塞赋值( $=$ )，在时序逻辑中使用非阻塞赋值，组合逻辑中使用阻塞赋值。

阻塞赋值时，输入改变输出也同时改变，在非阻塞赋值中，只有在时钟变化的时候，输出才会发生变化。

在数字电路中，所有的数据最终都是以二进制形式呈现的，二进制数据和十进制一样，都是数据的一种形式，数据既能用二进制表示也能用十进制表示，对

于同一个数据无论是表示为二进制还是十进制，大小都是相同的，十进制数据是 0 到 9 组成，二进制是 0, 1 组成，二进制的每个 0 或者 1 叫做 1 位，若想求得一个数据由多少 bit，首先需要将该数据转换成二进制数据，根据二进制数据有多少位来确定数据有多少 bit。

在之前定义的数据中，由于没有指定位宽，所以每个变量只能传输或者存储 1bit 数据，这些变量都只能表达 0 或者 1，若想表达的数据不止 0 或者 1 需要声明其位宽。对前面的代码做以下改变。

```

1 module and_gate(
2     input wire          clk,
3     input wire [1:0]    a   ,
4     input wire [1:0]    b   ,
5     output reg [1:0]   c
6 );
7
8 always @(posedge clk) begin
9     c <= a & b;
10 end
11
12 endmodule

```

第 3~5 行为定义带位宽的 wire、reg 型变量格式，变量类型关键字后面的中括号内写上数据的最高位和最低位，其中以冒号隔开。假设定义一个八位宽的线型变量 a，可以写成 wire[7:0] a，也可以写成 wire[8:1] a，建议最低位从 0 开始；

### 4.3 赋值语句

在 Verilog HDL 常见的赋值语句有三种，在前面已经接触到了两种，assign 和 always 这两种赋值语句可以最终被综合生成电路。

其中 assign 用于组合逻辑的赋值；

always 用于时序逻辑赋值；

还有一种赋值一句是 `initial`, 其常用于仿真时给变量赋值, 一般情况下, `initial` 赋值语句, 只使用在仿真文件当中。

### 1) `assign` 赋值

`assign` 语句在赋值的时候, 只能对组合逻辑进行赋值, 在赋值的时候无法加如时钟, 只能由线和运算符组成, 使用 `assign` 赋值的变量必须是 `wire` 型。

```

1 wire      [2:0]    a,
2 reg       [2:0]    b,
3 wire      [2:0]    c
4
5 assign c = a | b;
```

上面所示的代码中, 第 1 到 3 行定义了三个 3bit 的变量, 采用 `assign` 方式对 `wire` 变量 `c` 进行赋值, `c` 的结果是 `a` 与 `b` 取或, 可以看到, 等号右边的变量既可以是 `wire` 型也可以实 `reg` 型。

### 2) `always` 进行赋值

使用 `always` 进行赋值, 即可以实现组合逻辑, 也可以实现时序逻辑, 但是其赋值的对象只能是寄存器类型的数据。

`always` 赋值语句实现时序逻辑, 在前面已经有过介绍, 下面介绍下, 如何使用 `always` 语句实现时序逻辑。

```

1 wire      [2:0]    a,
2 reg       [2:0]    b,
3 reg       [2:0]    c
4
5 always @(a or b) begin
6     c = a & b;
7 end
```

第三行定义一个 `reg` 类型变量 `c`, `c` 在 `always` 块中进行赋值, 所以其只能是寄存器类型, 但由于该 `always` 块是对组合逻辑进行赋值, 所以其在综合器综合

的时候将被综合为 `wire` 类型。

第五行，`always` 后面是有`@()`形式的另外一种写法，此处括号内没有写`posedge` 或者 `negedge`，表明此语句不是由沿触发，而是由电平触发，所以该电路实现的是跟沿无关的组合逻辑。此处括号内的 `or` 可改为 “，”；

### 3) initial 语句

`initial` 语句是初始化语句，会在上电（电路刚刚运行时）执行一次，不会循环执行。在电路中只有可以存储数据的寄存器才有初始化的必要，所以 `initial` 语句中被赋值的变量也必须为 `reg`。某些第三方综合软件认为 `initial` 是不可以被综合的，也就是说不可以被写到功能文件中的，所以为了代码的兼容性，我们尽量只在测试文件中写 `initial` 语句。当 `initial` 和 `always` 同用时可以很好的描述出时钟，示例代码如下所示。

```

1 reg      clk;
2 initial begin
3   clk = 0;
4   always #(5) clk = ~clk;
5 end

```

上述代码，将会产生一个周期是 `10ns` 的时钟。第一行定义了一个 `reg` 类型的变量，第三行对其进行了初始化，第 4 行每隔 `5ns` 对 `clk` 进行赋值一次。

## 4.4 运算符

Verilog HDL 中的运算符与 C 语言中的运算符基本相同，本小结中不会对其做过多介绍，只介绍几种比较常用的操作符。

### 1) 关系运算符 (<、<=、==、>=、>、!=)

关系运算符也与 C 语言中的基本类似，需要注意的是，在 Verilog 中，与 C 语言还是有一点区别，

例如在 C 语言中对一个数据是否处于某个范围可使用: `if(2<a<6)`

但在 Verilog 中需要使用 `if(a>2 && a<6)` 来判断数据是否处于这个范围值中。

分析如下:

在 Verilog 中, 若使用 `if(2<a<6)` 来进行判断,

当  $a=1$  时,  $2<1$  为假结果为 0, 运算后的结果为  $0<6$  为真。当  $a=8$  时,  $2<8$  为真结果为 1, 预算后  $1<6$  结果为真, 可以看到 if 后的条件, 无论 a 取任何值, 都将保持为真。

而采用第二种写法, 只有当两边条件都满足时, if 后面的条件才为真。

## 2) 逻辑运算符 (&&、 ||、 !)

逻辑运算符与 C 语言中一致: 对于 `&&` 只有参与运算的两个数据都为真时, 结果才为真, 对于 `||` 只要参与运算的两个数据由一个为真, 结果就为真。`(!)` 进行的时逻辑取反。

## 3) 位运算符 (&、 |、 ~)

时按照参与运算的数据的每一位进行操作的。位运算是对二进制的运算, 运算时, 需要将数据都转换位二进制后才能进行计算。

```

1 wire[1:0] a=2'b10;
2 wire[2:0] b=3'd3;
3
4 assign c = a&b;
5 assign c = a|b;
6 assign c = ~a;
7 assign c = &a;

```

第 1、2 行定义两个具有初始值的变量;

第 4 行实现  $a \& b$ , 此时需要 a 和 b 都转换成二进制,  $a=2'b10$ ,  $b=3'b011$ , 此时 a 和 b 的位宽不一样, 位宽少的在高位补 0, 最后运算为

`3'b010 & 3'b011=3'b010;`

第 5 行实现  $a|b$ , 此时需要将  $a$  和  $b$  都转换成二进制,  $a=2'b10$ ,  $b=3'b011$ , 此时  $a$  和  $b$  的位宽不一样, 位宽少的在高位补 0, 最后运算为

```
3'b010|3'b011=3'b011;
```

第 6 行实现 $\sim a$ ,按位取反, 每一位都取其相反数, 运算结果为  $\sim 2'b10 = 2'b01$ ;

第 7 行实现按位取与, 运算结果为  $a[1]\&a[0] = 1\&0 = 0$ ;

#### 4) 条件运算符

格式:  $(x)?xxx:XXX$ ;使用 `assign` 进行赋值的时候, 有时候在条件不一样时对某一变量赋值, 可以用条件运算符进行赋值:

如: `assign a = (b>6)?1'b1:1'b0;`

当  $b$  大于 6 时,  $a$  等于 1, 否则  $a$  等于 0;

#### 5) 赋值运算符

前面已经介绍过 Verilog 中的赋值运算符, `<=` 和 `=`, 其中`<=` 是非阻塞赋值, 用于时序逻辑, `=`是阻塞赋值, 用于组合逻辑。

#### 6) 移位运算符(<<、>>)

`>>`为向右移位, 每次右移一位, 数据高位补零, 向下溢出的数据丢弃。`<<`与之相反。

#### 7) 位拼接运算符 ({})

位拼接运算符, 可以将不同数据的位拼接成一个新的数据。

```
1 reg[3:0] a=4'b0110;
2 reg[4:0] b=5'b10110;
3
4 always@(posedge clk)
5 c <= {b[1],a[1:0],b[3],b[1]};
```

拼接完成后，`c = 5'b11001;`

## 4.5 条件判断语句

同一个变量可以在不同的情况下获得不同的值，不同的情况需要判断语句来描述。Verilog HDL 中经常使用的判断语句有 `if else` 语句和 `case endcase` 语句，两种判断语句必须写在 `always` 语句中，不能写在 `assign` 中。

### 1) If-else

`if-else` 语句与 C 语言当中的使用方法类似。

```
1 always@(posedge clk)begin
2     if(a==1)
3         b <= 1;
4     else if(a==2)
5         b <= 2;
6     else if(a==3)
7         b <= 3;
8     else if(a==4)
9         b <= 4;
10    else if(a==5)
11        b <= 5;
12 end
```

`if else` 叠加不易过多，不然可能造成线路的延时过多。每一个 `if else` 语句都会生成一选择器，当 `if else` 过多时，选择器链路就会很长，而每两级选择器之间都会有线路的延时，当链路过多时，造成的延时就会很多，这样对于描述的电路的时序影响会很大，时序出问题时，就算是功能仿真正确，下板后电路也是不正确的。

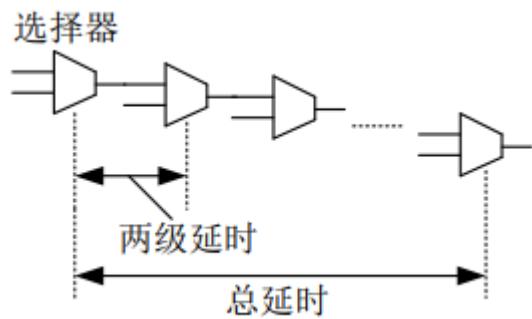


图 6-5

使用 if-else 语句时需要考虑优先级的顺序，优先执行的条件放在上方，优先级较低的条件放在后方。

## 2) Case 语句

在 if-else 级数过多的情况下，也可以使用 case 语句，case 语句生成的时多路器。

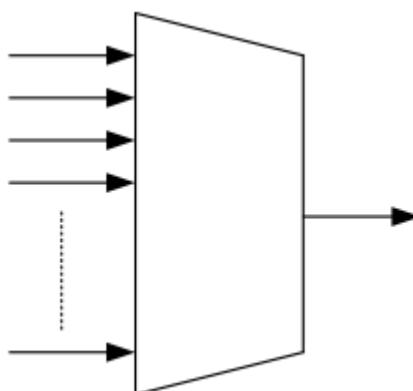


图 6-6

```

1 always@(posedge clk)
2 case(a)
3   0: b <= 0;
4   1: b <= 1;
5   2: b <= 2;
6   default:b <= 0;
7 endcase

```

Case 语句以 case 开始，以 endcase 结尾，在其之间，列出要判断的条件，

根据条件的值，执行对应的代码。

## 4.6 总结

在本章中，对 Verilog HDL 的语法进行了简单的介绍，关于 Verilog 语法，不必去深究，因为在 FPGA 开发，本质上是对电路进行开发，不必拘泥于语法，在之后的联系中，可以发现常用的 Verilog 语法就只有几句，掌握好以上的 Verilog 语法可以胜任大多是 Verilog 的开发。Verilog 开发主要关注的是时序和结构，代码仅仅只是实现方式。

## 第5章 按键控制 LED 灯

### 5.1 实验简介

在前面的章节中，介绍了 Verilog 的基本语法，在本章中用一个简单的实验，完成按键按下 led 灯点亮的实验，进一步掌握 FPGA 开发的流程。

在 Z7-Lite 开发板上板载有 2 个 PL 侧的按键和 LED 灯。通过按下按键点亮对应的 LED 灯，本次实验使用到其中的一个 LED 灯，两个按键，其中 PL\_KEY1 用于系统复位按键。

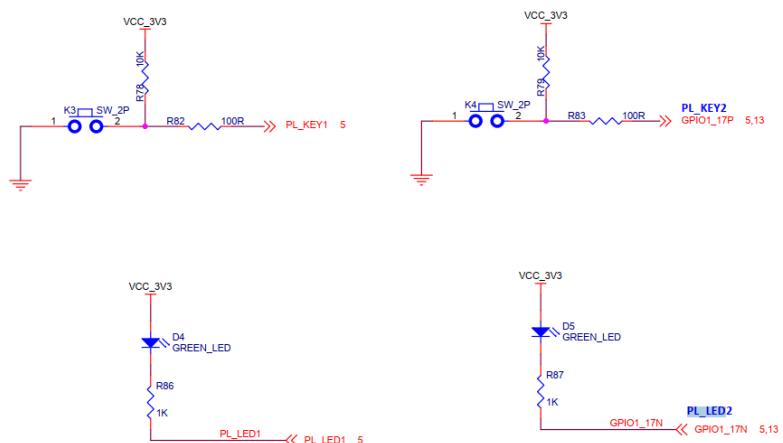


图 7-1

### 5.2 创建工程

- 1) 打开 Vivado 开发环境

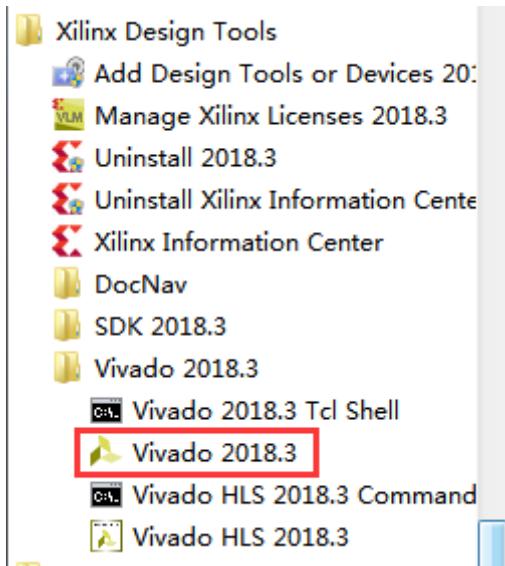


图 7- 2

2) 新建工程，输入工程名称，确定工程路径

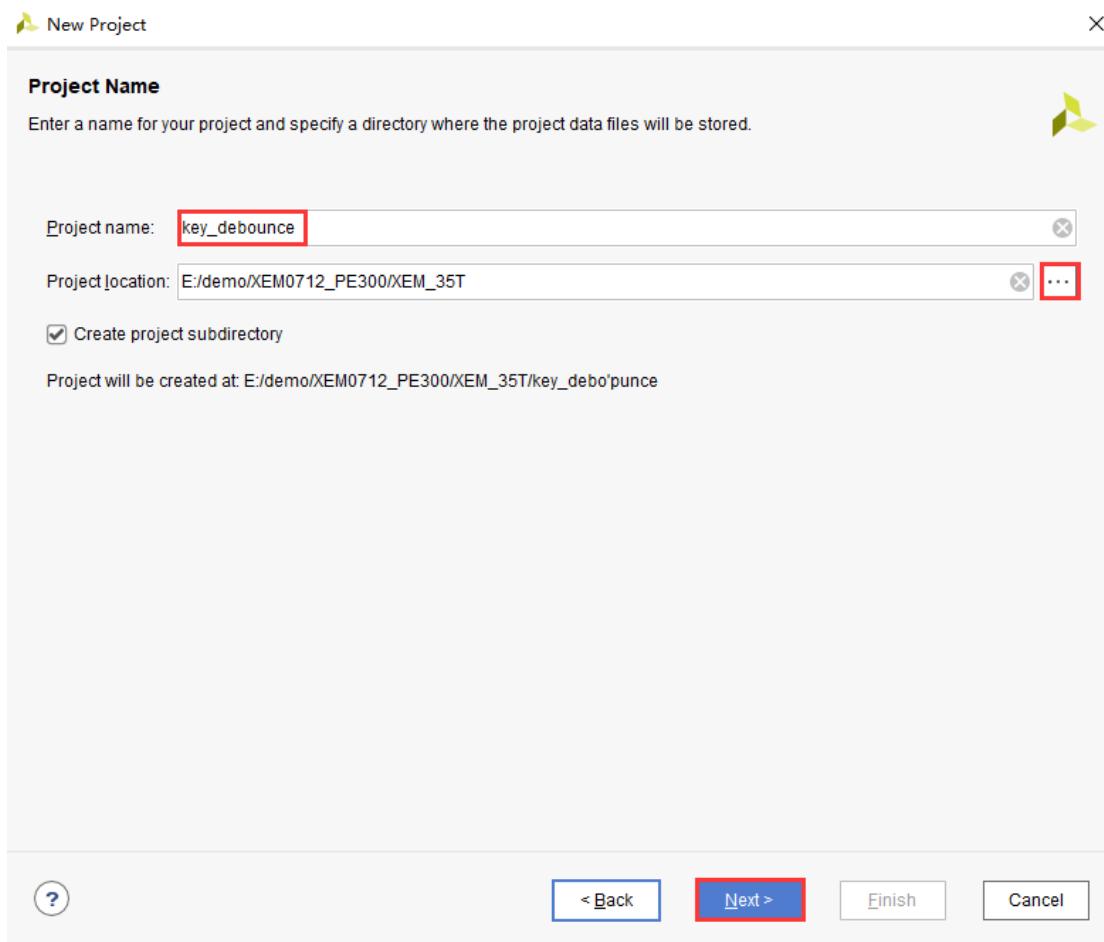


图 7- 3

选择器件，器件选择同上一章一样，可通过筛选也可以通过直接搜索的方法获取器件，这里以 xc7z020clg400-2 为例

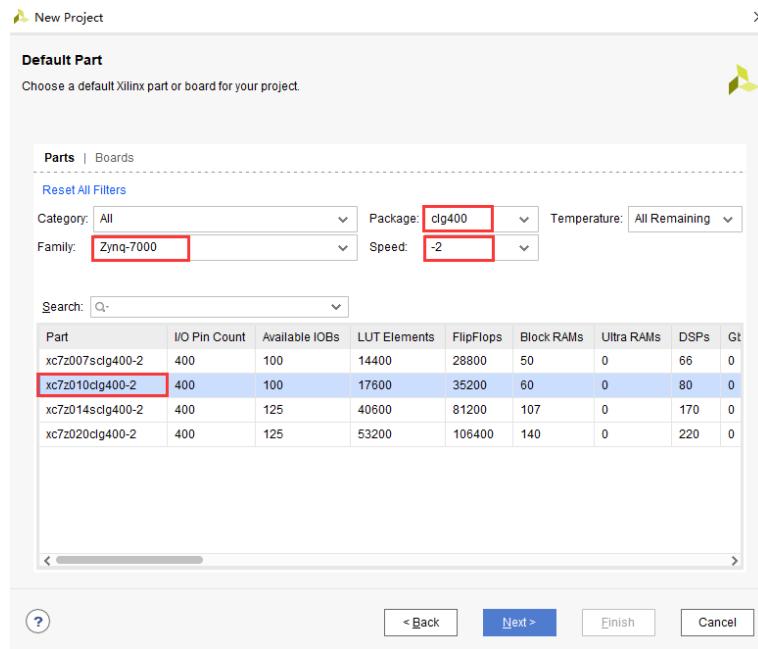


图 7-4

通过直接搜索得到：

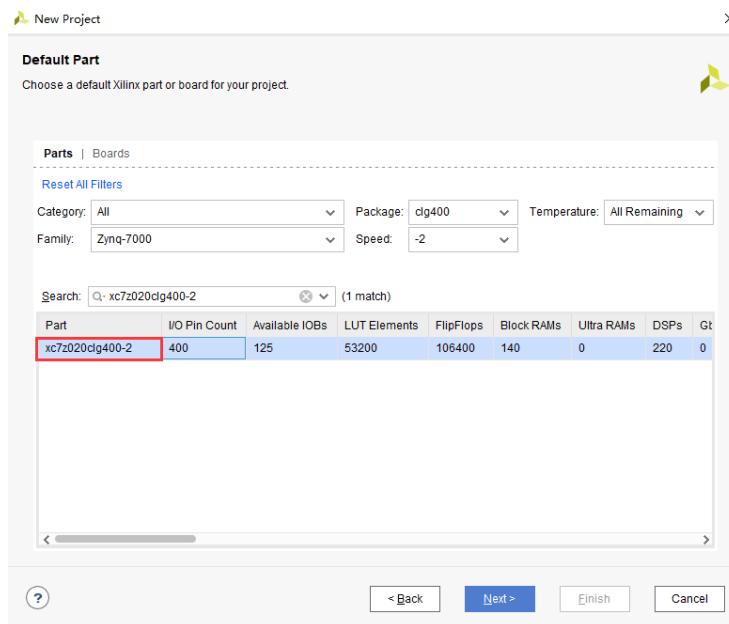


图 7-5

3) 创建设计文件，可以通过导航栏的 Add Source 或者通过文件管理栏的加号添

## 加文件

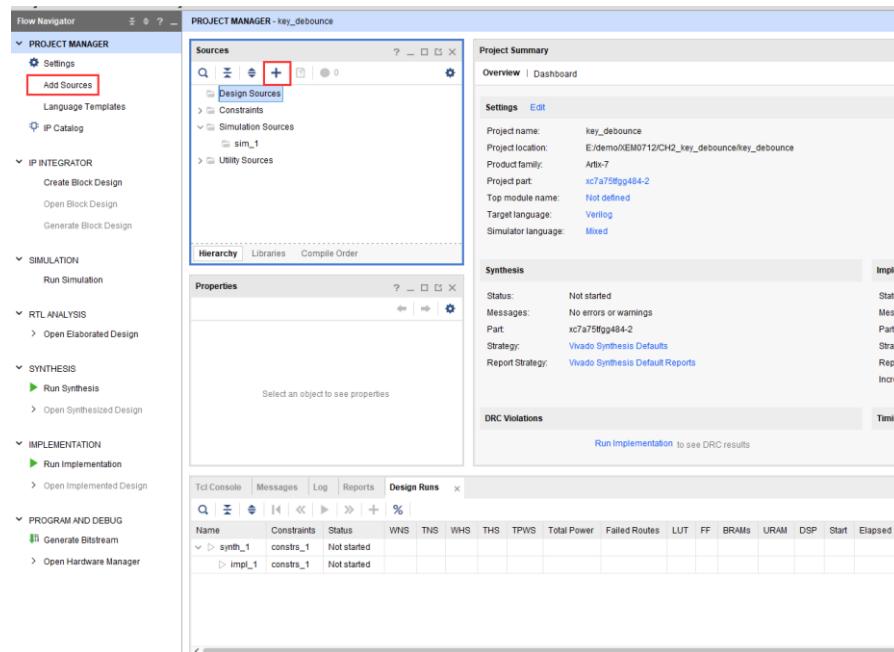


图 7-6

首先添加设计文件，命名为 key\_led

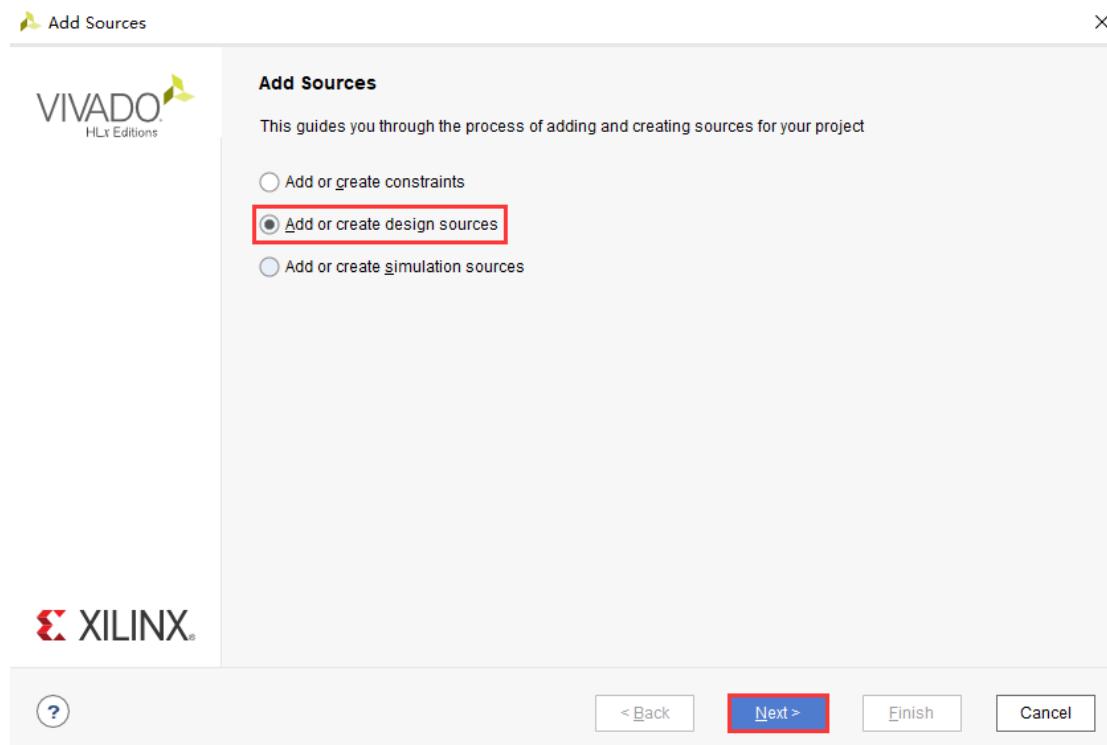


图 7-7

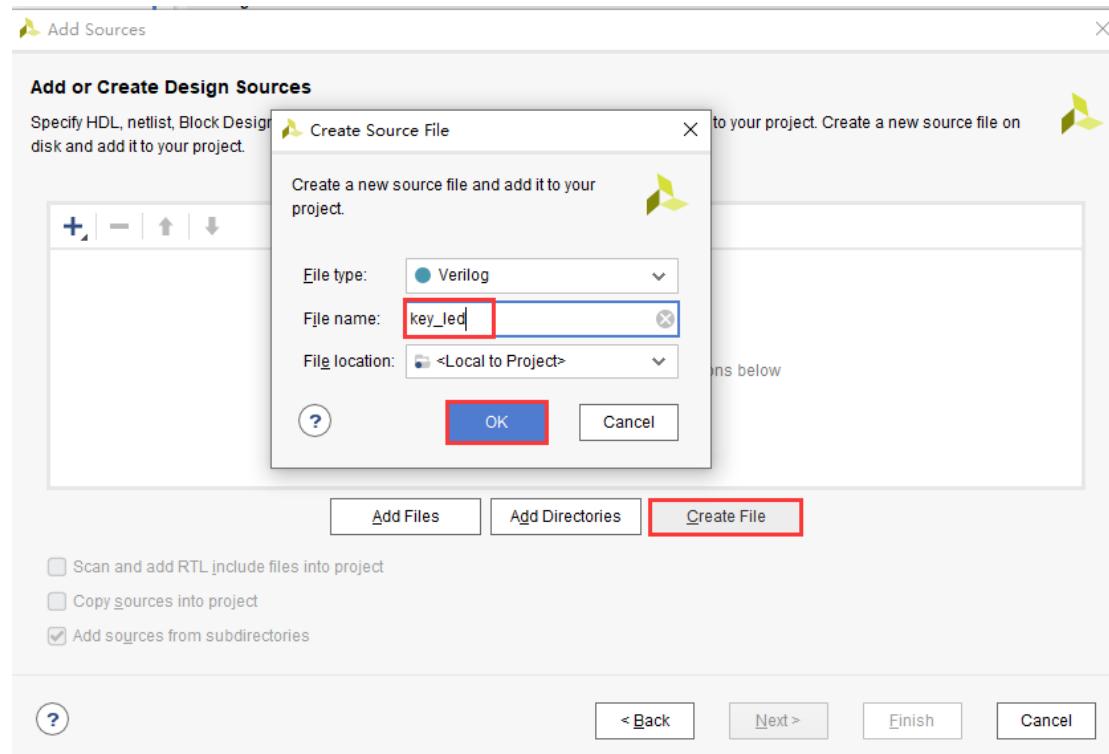


图 7-8

### 5.3 程序设计

```

1 `timescale 1ns / 1ps
2 module key_led(
3   input  wire        clk  ,
4   input  wire        rst_n  ,
5   input  wire        key  ,
6   output wire        led
7 );
8
9 // 对输入的信号进行打拍, 消除亚稳态
10 reg  [1:0]key_dd  ;
11
12 always @(posedge clk or negedge rst_n) begin
13   if (rst_n==1'b0) begin
14     key_dd <= 2'b11;
15   end
16   else begin

```

```
17     key_dd <= {key_dd[0],key};  
18 end  
19 end  
20  
21 // 输出的led为打拍过后的结果  
22 assign led = key_dd[1];  
23  
24 endmodule
```

然后点击综合，可以查看综合过后的结果：

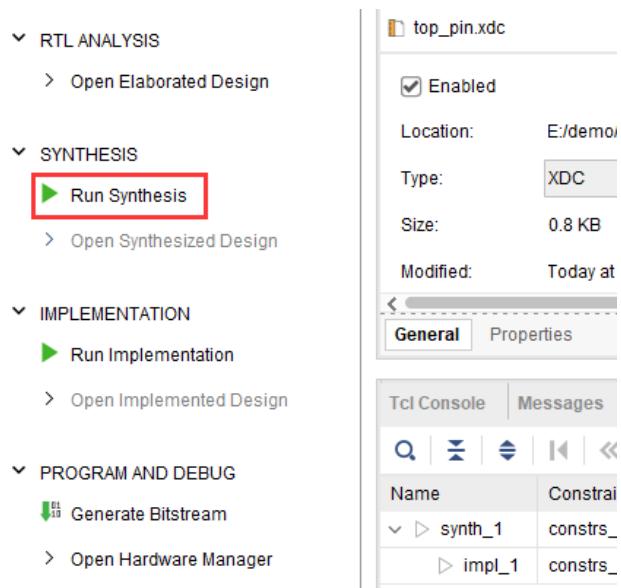


图 7-9

等到中和完成过后，可以打开综合后的结果，查看综合器将代码综合成了什么电路结构：

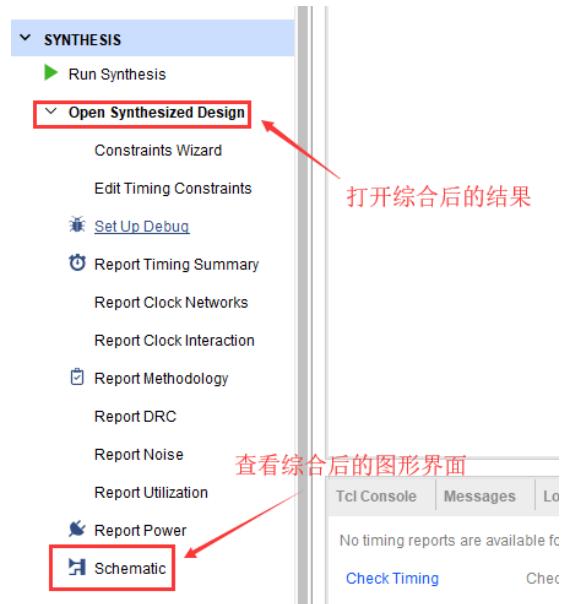


图 7-10

可以看到综合后，综合器生成了 2 个 D 触发器，这个正好对应了其中的 key\_dd 的 2 位位宽，由于采用了位拼接的方式来完成移位，通过查看 D 触发器之间的连线可以看到移位的数据流向。

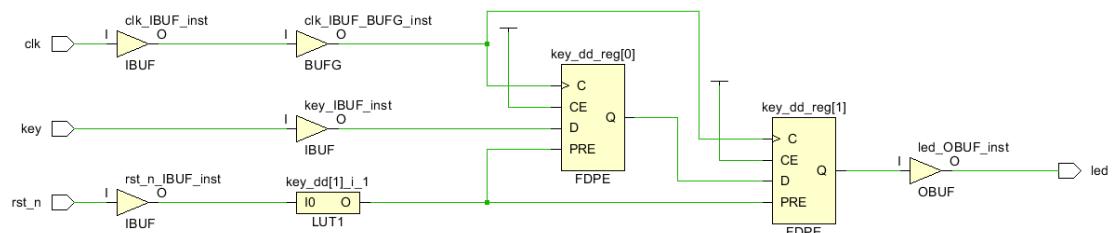


图 7-11 Z7-Lite 系列综合后结果

在该页面下还可以进行引脚分配，如下图所示：

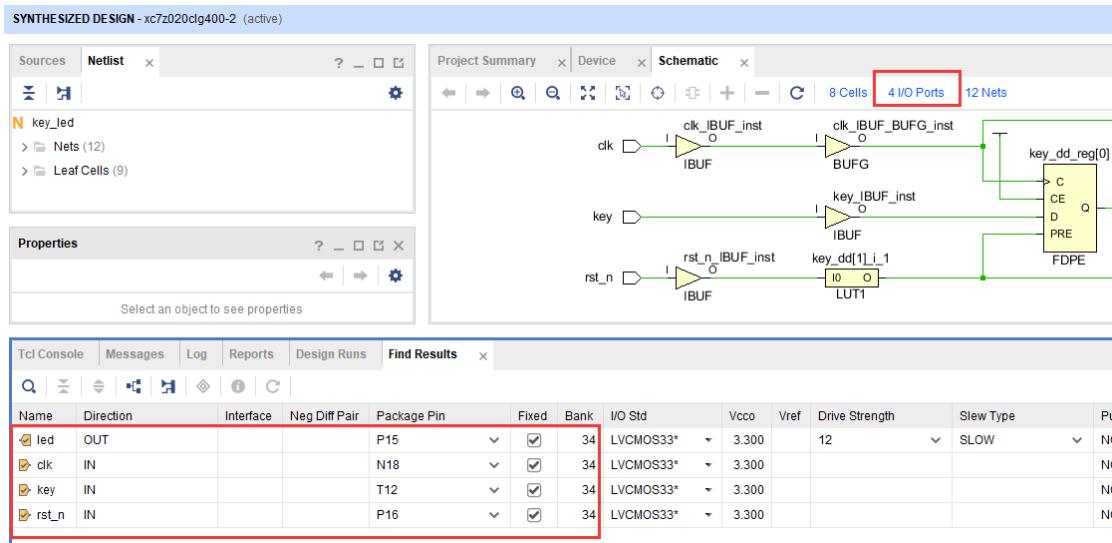


图 7- 12

分配好引脚后，**ctrl+s** 保存，Vivado 会生成一个约束文件，点击即可查看，这里的约束文件。

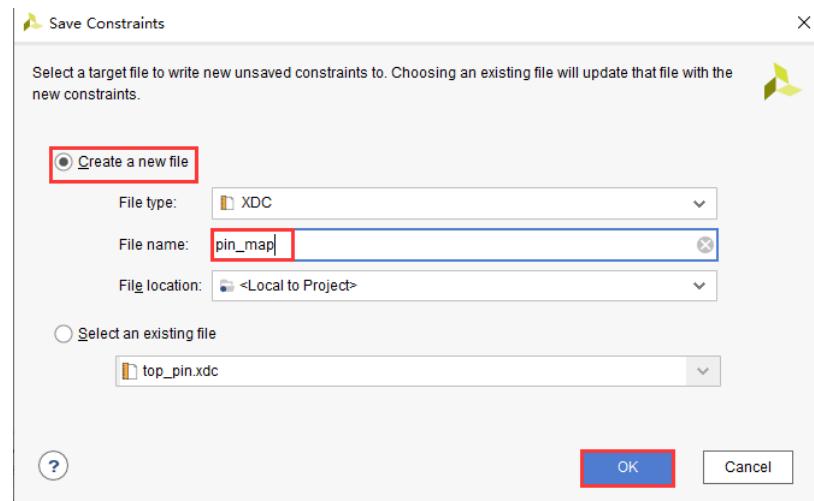


图 7- 13

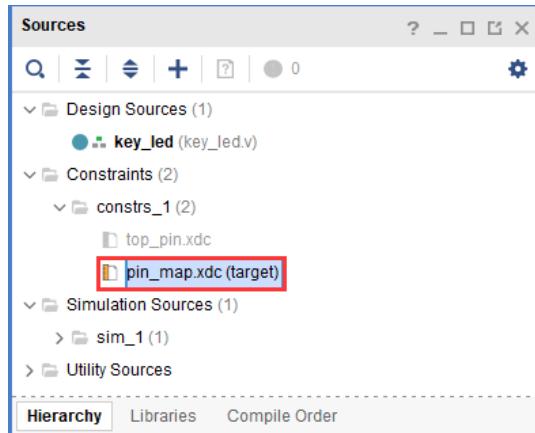


图 7-14

分配好引脚过后，可以继续后面的步骤，布局布线，生成 bit 文件。

## 5.4 下载验证

等待 bit 文件生成完成后，将 bit 文件下载进 FPGA，然后按下按键，即可看到对应的 led 灯点亮。

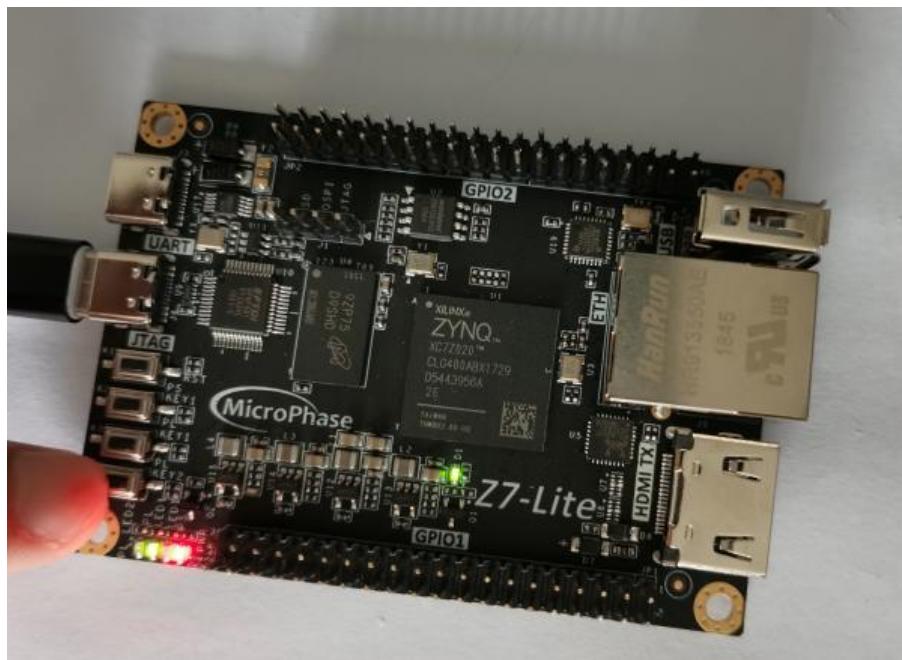


图 7-15 Z7-Lite 实验结果

## 5.5 实验总结

本次实验通过按键点亮 led，对前面的 Verilog 语法有了进一步的认识，并且了解到了 Vivado 中综合工具的作用，并通过系统的引脚分配工具，完成了引脚的约束文件的创建。

## 第6章 按键检测与消抖

### 6.1 实验简介

机械按键是常用的器件，但在机械按键使用的过程中，总伴随着抖动。在本次实验中，我们将完成按键的去抖，并且通过按键控制 led 灯的亮灭。

### 6.2 实验原理

在 Z7-Lite 开发板上，带有 2 个 PL 端按键，如原理图所示。其中有一个复位按键，两个 PL 端按键。本次实验使用 PL\_KEY，完成对 PL\_KEY 按下时的去抖和控制 led 灯。

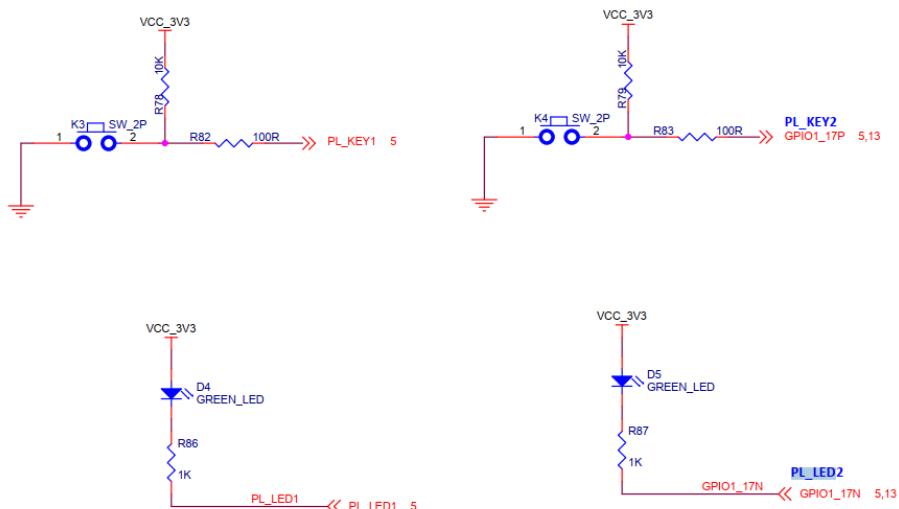


图 8-1

抖动产生的示意：

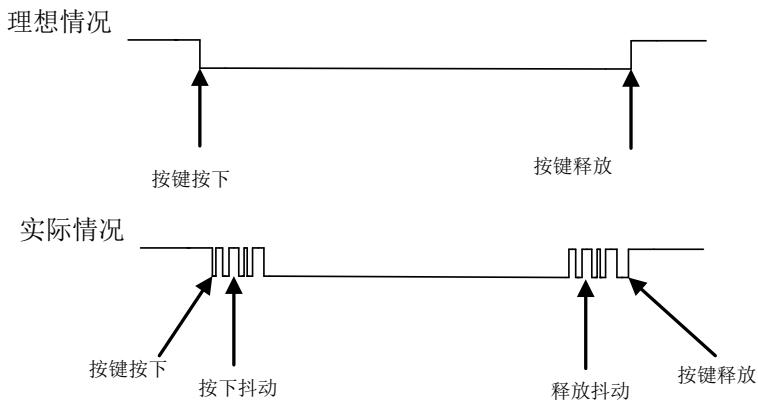


图 8-2

理想情况下，按键按下过后，按键处为低电平，当按键释放过后恢复为高电平，但在实际情况下，由于按键采用的机械结构，当按键按下过后，电平不能立即保持为低，会有一段时间的跳变后才稳定，保持为低。同理，当按键释放的时候，电平不能立即保持为高，也有一段时间的不稳定状态。

在本次实验中，通过按键输入电平保持为低的达到 20ms 后认为按键已经稳定，通过计数器对按键输入为低电平时间进行计数，消除按键抖动。

### 6.3 创建工程

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 ram_pp, 工程路径 E:\demo\FPGA\ram_pp 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件：购买 Z7-Lite7020 用户选中 xc7z020clg400-2，购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后，点击确认

## 6.4 程序设计

这里提供一种按键去抖的方法，在程序中使用了一个计数器，当计数器检测到按键输入为低的时候开始累加，当检测到高电平时清零。当计数器计数到一定时间后，认位此时按键已经稳定，产生一个按键稳定信号 key\_flag。

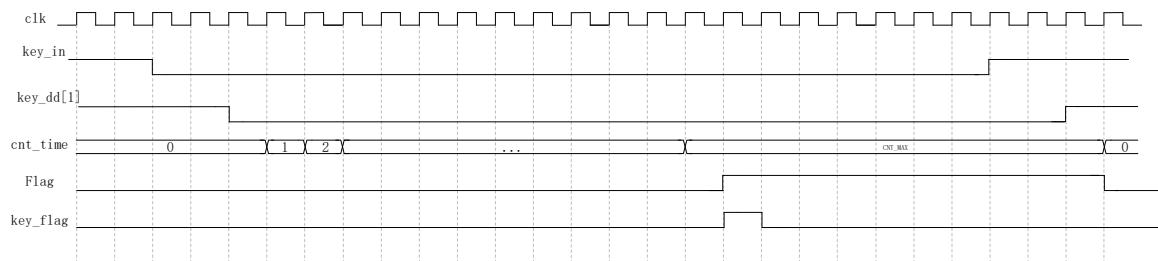


图 8-3

```

1 module key_debounce(
2     input wire          clk      ,//system clock input 50M
3     input wire          rst_n   ,//system reset
4     input wire          key_in  ,//key input
5     output reg [1:0]    led
6 );
7
8 //=====
9 //parameter define
10 //=====
11 parameter CNT_MAX = 4000_000 - 1; //20ms counter max data
12
13 //=====
14 //internal signals
15 //=====
16 reg [21:0] cnt_time ;//timer counter
17 wire       add_cnt_time;//add condition
18 wire       end_cnt_time;//end condition
19
20 reg [1:0]  key_dd   ;//aim to Eliminate metastable state
21 reg        key_flag  ;//key has been push down flag
22 reg        flag      ;
23 reg [1:0]  led_r    ;//led control
24
25

```

```
26 //-----key_dd-----
27 always @ (posedge clk or negedge rst_n)begin
28     if(rst_n==1'b0)begin
29         key_dd <= 'd0;
30     end
31     else begin
32         key_dd <= {key_dd[0],key_in}//key_in delay 2 beats
33     end
34 end
35
36 //-----cnt_time-----
37 always @ (posedge clk or negedge rst_n)begin
38     if(rst_n==1'b0)begin
39         cnt_time <= 'd0;
40     end
41     else if(add_cnt_time)begin
42         if(end_cnt_time)
43             cnt_time <= cnt_time;
44         else
45             cnt_time <= cnt_time + 1'b1;
46     end
47     else begin
48         cnt_time <= 'd0;
49     end
50 end
51
52 assign add_cnt_time = (key_dd[1]==1'b0);
53 assign end_cnt_time = add_cnt_time && cnt_time==CNT_MAX;
54
55 //-----flag-----
56 always @ (posedge clk or negedge rst_n)begin
57     if(rst_n==1'b0)begin
58         flag <= 1'b0;
59     end
60     else if(cnt_time==CNT_MAX)begin
61         flag <= 1'b1;
62     end
63     else if(key_dd[1]==1'b1)begin
64         flag <= 1'b0;
65     end
66     else begin
```

```
67      flag <= flag;
68  end
69 end
70
71 //-----key_flag-----
72 always  @(posedge clk or negedge rst_n)begin
73     if(rst_n==1'b0)begin
74         key_flag <= 1'b0;
75     end
76     else if(cnt_time == CNT_MAX && flag==1'b0)begin
77         key_flag <= 1'b1;
78     end
79     else begin
80         key_flag <= 1'b0;
81     end
82 end
83
84 //-----led_r-----
85 always  @(posedge clk or negedge rst_n)begin
86     if(rst_n==1'b0)begin
87         led_r <= 2'b01;
88     end
89     else if(key_flag==1'b1)begin
90         led_r <= {led_r[0],led_r[1]};
91     end
92     else begin
93         led_r <= led_r;
94     end
95 end
96
97 //-----led-----
98 always  @(posedge clk or negedge rst_n)begin
99     if(rst_n==1'b0)begin
100        led <= 2'b11;
101    end
102    else begin
103        led <= ~led_r;
104    end
105 end
106
107 endmodule
```

在本程序设计中，使用了一个移位寄存器 `key_dd`，目的是对输入的按键信号进行亚稳态处理，对输入信号采取延时两拍进行操作，能够有效降低亚稳态。

使用了一个计数器 `cnt_time`，用来计数按键输入为低电平的时间。当按键输入为高电平时，计数器清零。当达到一个 `10ms` 所需要的计数值时，计数器保持不变。

使用了一个计数指示信号 `flag`，用来定位与指示当前 `cnt_time` 是否达到了计数所需最大值。

使用了一个按键按下指示信号 `key_flag`，用来指示本次按键成功按下。

使用 `led_r` 来控制 `led` 灯移动，通过对 `key_flag` 进行检测来控制 `led` 灯的亮灭。

## 6.5 添加管脚约束

1) 首先创建一个管脚约束文件，点击导航栏 `Add Source`，在弹出界面选择第一项，添加约束文件。

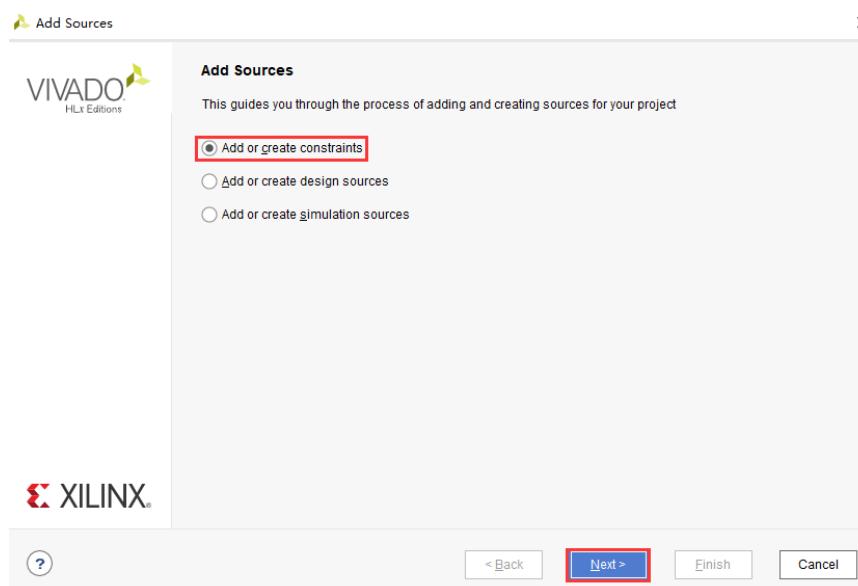


图 8- 4

2) 在弹出界面，选择创建文件，并给文件命名为 **top\_pin**

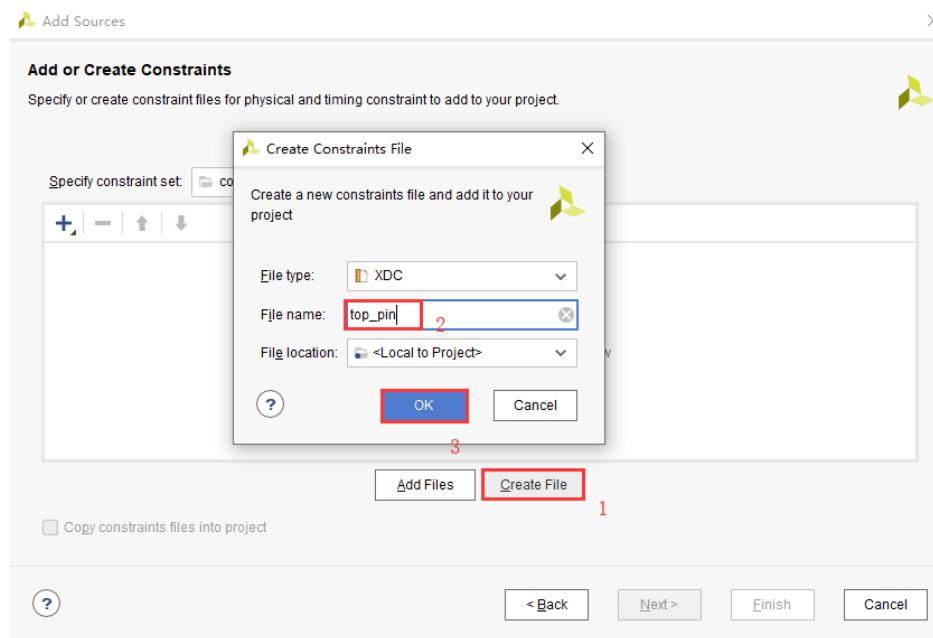


图 8-5

3) 添加约束信息

对输入差分时钟进行时钟周期，管脚和电平约束。

对 led 和按键分别进行管脚和电平约束。

Z7-Lite 系列约束文件：

```
#####
# clock define#####
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
#####
# key define#####
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property PACKAGE_PIN T12 [get_ports key_in]
#####
#LED define#####
set_property PACKAGE_PIN P15 [get_ports {led[0]}]
set_property PACKAGE_PIN U12 [get_ports {led[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports {led[*]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports key_in]
```

4) 保存管脚约束，并依次点击综合，实现，生成比特文件。或者直接点击生成比特文件。

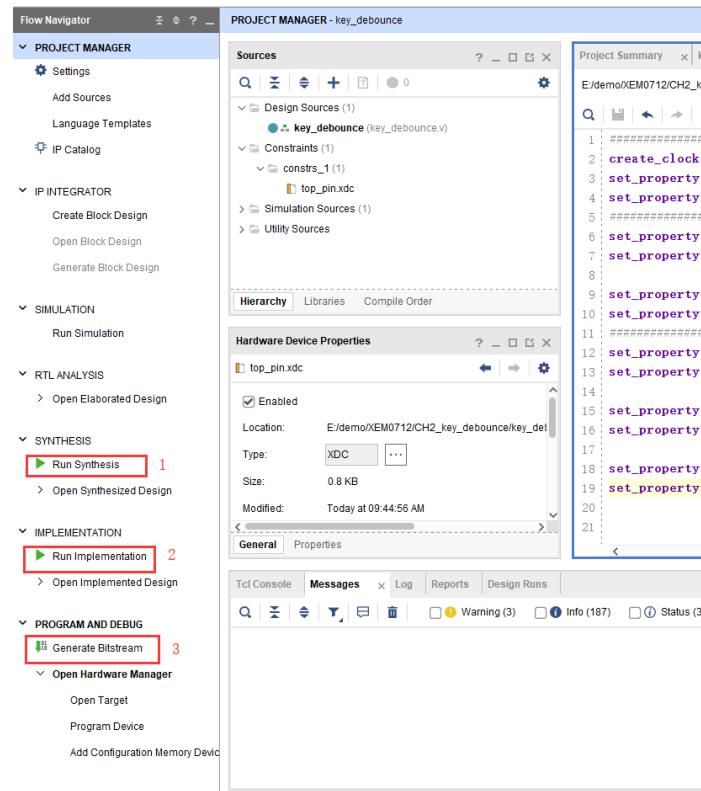


图 8-6

## 6.6 下载验证

1) 生成完 bit 文件过后，可以连接开发板进行 bit 文件的下载。将下载器连接到 JTAG 口，给板子上电，点击 Open Hardware Manager，在 Hardware Manager 窗口里点击 Open target，然后点击 Auto Connect.

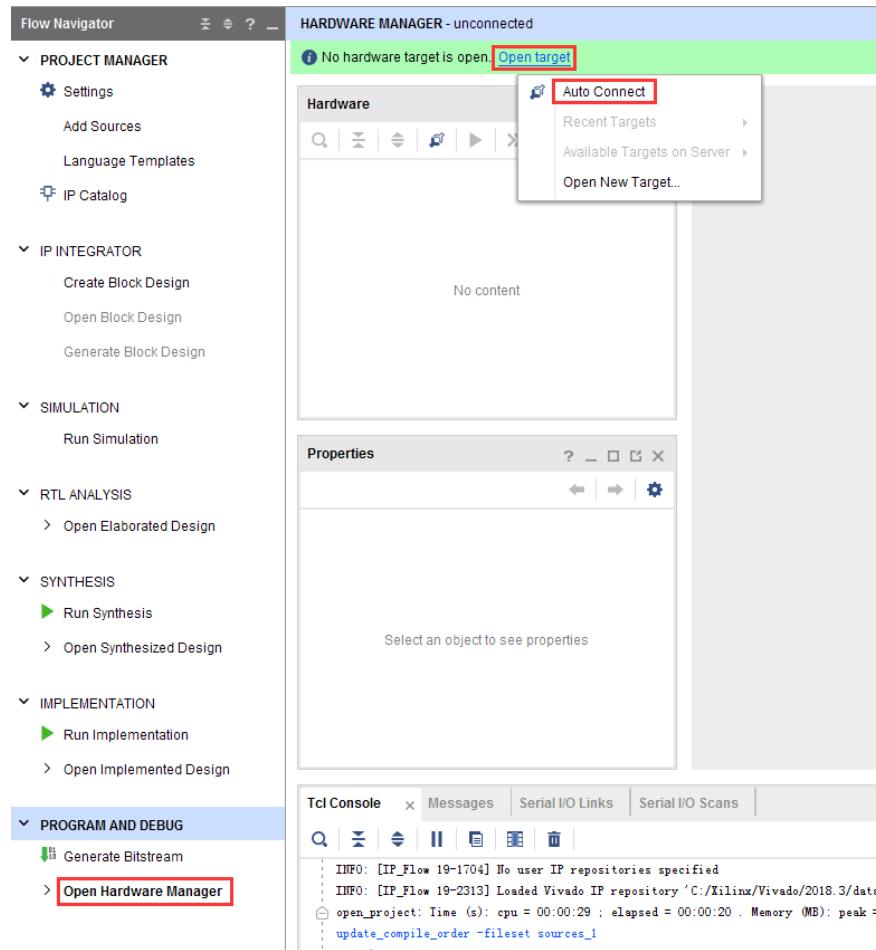


图 8-7

2) 选择 program device，将程序下载到开发板中，观察实验现象。

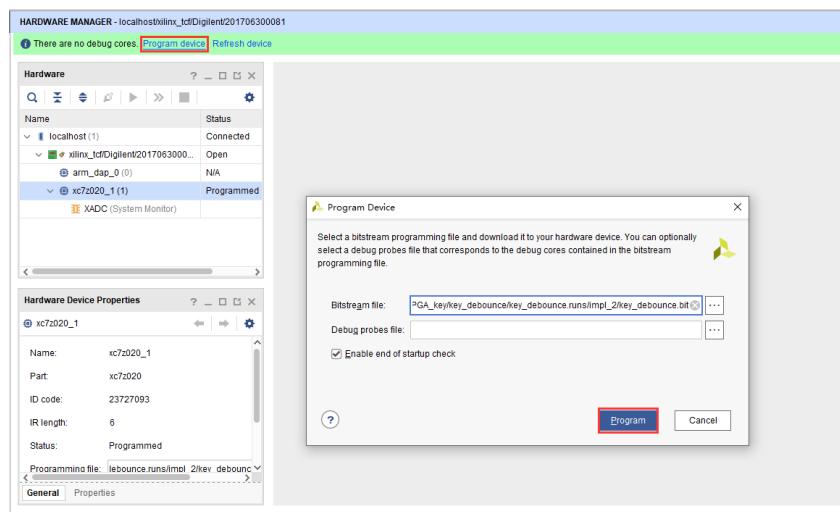


图 8-8

### 3) 观察现象

Bit 文件下载完毕后，按下的 PL\_KEY2 可以观察到 led 灯交替亮灭，每按每一次， led 灯移动一位。且按下一次按键 led 灯不会发生多次跳动。

## 6.7 实验总结

通过本次实验，完成了按键的去抖，本次实验主要是为了加深读者对计数器这一在 **FPGA** 开发中经常使用到的结构，并且又一次熟悉了 **FPGA** 的开发流程。通过本次实验，相信读读者能够对 Vivado 的开发 Verilog 代码有了更深刻的体会。

## 第7章 PWM 呼吸灯

### 7.1 实验简介

脉宽调制（Pulse Width Modulation，PWM），是利用微处理器/FPGA 的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用在测量、通信、功率控制与变换等许多领域。PWM 数字信号从处理器到被控系统都是数字形式，无需数模转换。

PWM 是一种对模拟信号电平进行数字编码的方法。通过高分辨率计数器的使用，方波的占空比被调制用来对一个具体模拟信号的电平进行编码。PWM 信号仍然是数字的，因为在给定的任何时刻，满幅值的直流供电要么完全有(ON)，要么完全无(OFF)。电压或电流源是以一种通(ON)或断(OFF)的重复脉冲序列被加到模拟负载上去的。通的时候即是直流供电被加到负载上的时候，断的时候即是供电被断开的时候。只要带宽足够，任何模拟值都可以使用 PWM 进行编码。

通俗来说，PWM 就是连续的、一定比例占空比的脉冲信号。通过控制占空比来实现不同的控制。简单地，我们可以认为 PWM 就是一种方波。

本次实验将采用 PWM 来完成一个呼吸灯，呼吸灯在 2 秒内由灭到亮，然后再熄灭的过程。

### 7.2 实验原理

输出电平时控制 LED 灯，当输出低电平时，灯亮，当输出高电平时，灯灭。如果一直输出低电平，则灯一直亮；如果一直输出高电平，则灯一直暗；如果 50% 时间输出低电平，50% 时间输出高电平，则灯会暗一些。所以占空比会影响到 LED 灯的明暗程度。

另一个影响 LED 亮度的是 PWM 波形的周期。试想一下，如果 PWM 的周

期是 2 秒，占空比为 50%，那我们看到的是将是 LED 暗 1 秒、亮 1 秒，而不是半亮的状态。只是我们提高 PWM 的周期，才能看到半亮的状态。根据经验值，PWM 的周期是 10 毫秒为宜。也就是说，控制高低电平的时间，也就是占空比（高电平占周期的百分比），以及控制 PWM 的周期，就可以控制灯的亮暗程度。

### 7.3 创建工程

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 pcie_test, 工程路径 E:\demo\pll 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件，购买 Z7-Lite7020 用户选中 xc7z020clg400-2，购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后，点击确认

### 7.4 程序设计

本次实验使用到两个计数器，一个用于计数 10ms，这是 PWM 方波的周期，一个用于计数 2s，为了来确定再 2s 内已经产生了多少个 PWM 波。

使用 `work_flag` 来指示当前方波的高电平的占空比是增加还是减少，当 `work_flag == 1` 时，此时控制 `led` 熄灭，方波的占空比应该降低，当 `work_flag=0` 时，此时控制 `led` 灯点亮，方波的占空比应该增加，使用 `duty_cycle` 来指示在当前 PWM 波中，高电平占整个方波的比例。

程序设计可以参考如下时序设计图：

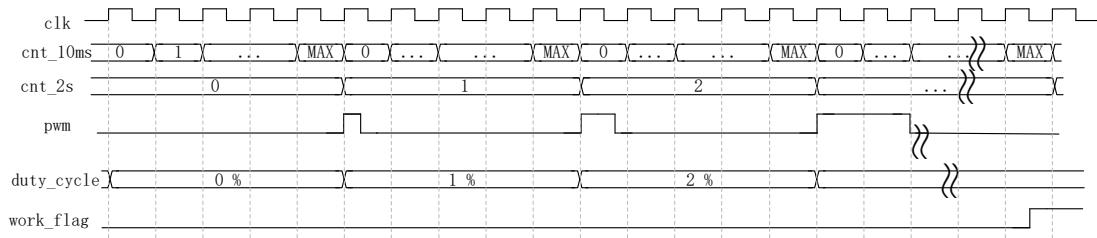


图 9-1

下面是本次实验的代码

```

1 `timescale 1ns / 1ps
2 module breath_led(
3   input wire          clk      ,
4   input    wire        rst_n    ,
5   output   wire        led
6 );
7
8
9 //=====
10 //parameter define
11 //=====
12 parameter CNT_10MS = 500000 - 1;      // 10ms 所需计数最大值
13 parameter CNT_2S = 200 - 1;           // 2s 所需计数最大值
14 parameter CHANGE_TIME = 100 - 1; // 1s 计数最大值
15 parameter PWM_OFFSET = 5000; // 每次脉冲调制的改变值
16
17 reg    [19:0]    cnt_10ms ;// 10ms 计数器
18 reg    [7:0]     cnt_2s   ;// 两秒计数器
19 reg    [19:0]    pwm      ;// 调制方波
20 reg    [19:0]    duty_cycle ;// 占空比
21 reg    [1:0]     work_flag ;// 工作状态, 0 由灭到亮, 1 由亮到灭
22
23 assign led = ~pwm;
24
25
26 //-----cnt_10ms-----
27 always @ (posedge clk or negedge rst_n) begin
28   if (rst_n == 1'b0) begin
29     cnt_10ms <= 'd0;
30   end
31 // 10ms 计数已到, 清零

```

```
32 else if (cnt_10ms == CNT_10MS) begin
33     cnt_10ms <= 'd0;
34 end
35 // 其他时间一致进行 10ms 计数
36 else begin
37     cnt_10ms <= cnt_10ms + 1'b1;
38 end
39 end
40
41 //-----cnt_2s-----
42 always @(posedge clk or negedge rst_n) begin
43     if (rst_n==1'b0) begin
44         cnt_2s <= 'd0;
45     end
46     // 10ms 计数完成时加一
47     else if (cnt_10ms == CNT_10MS)begin
48         // 2s 已经计数完成
49         if (cnt_2s == CNT_2S) begin
50             cnt_2s <= 'd0;
51         end
52         else begin
53             cnt_2s <= cnt_2s + 1'b1;
54         end
55     end
56 end
57
58 //-----work_flag-----
59 always @(posedge clk or negedge rst_n) begin
60     if (rst_n==1'b0) begin
61         work_flag <= 1'b0;
62     end
63     // 0~1 秒, 逐渐变亮, 计数到 1s 后, 状态发生变化
64     else if (cnt_2s == CHANGE_TIME && cnt_10ms == CNT_10MS) begin
65         work_flag <= 1'b1;
66     end
67     // 1~2 秒, 逐渐熄灭, 计数到 2s 后, 状态发生变化
68     else if (cnt_2s == CNT_2S && cnt_10ms == CNT_10MS) begin
69         work_flag <= 1'b0;
70     end
71 end
72
```

```
73 //-----duty_cycle-----
74 always @(posedge clk or negedge rst_n) begin
75   if (rst_n==1'b0) begin
76     duty_cycle <= 'd0;
77   end
78 // 由灭到亮时, pwm 占空比依次增加
79 else if (work_flag == 1'b0) begin
80   if (cnt_10ms == CNT_10MS) begin
81     duty_cycle <= duty_cycle + PWM_OFFSET;
82   end
83   else begin
84     duty_cycle <= duty_cycle;
85   end
86 end
87 // 由亮到灭, 占空比依次减少
88 else if (work_flag == 1'b1) begin
89   if (cnt_10ms == CNT_10MS) begin
90     duty_cycle <= duty_cycle - PWM_OFFSET;
91   end
92   else begin
93     duty_cycle <= duty_cycle;
94   end
95 end
96 end
97
98 //-----pwm-----
99 always @(posedge clk or negedge rst_n) begin
100  if (rst_n==1'b0) begin
101    pwm <= 1'b0;
102  end
103 else if (cnt_10ms < duty_cycle) begin
104   pwm <= 1'b1;
105 end
106 else begin
107   pwm <= 1'b0;
108 end
109 end
110
111 endmodule
```

## 7.5 仿真验证

设计好源代码后，可以进行仿真验证，仿真在 FPGA 开发中是很重要的一个步骤，通过仿真能够找出代码中存在的问题，然后可以回过头去修改代码，使之能够完成正确的功能。

### 1) 新建仿真文件

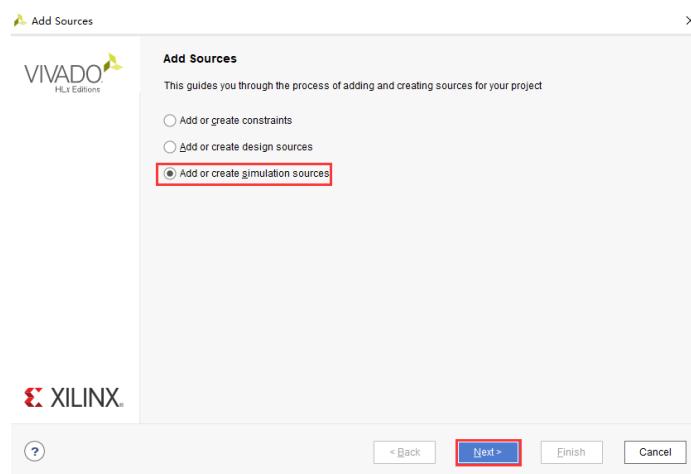


图 9-2

仿真文件的代码设计如下

```
1 `timescale 1ns/1ps
3 module tb_breath_led(); /* this is automatically generated */
4
5 reg rst_n;
6 reg clk;
7 wire led;
8
9 //(*NOTE*) replace reset, clock, others
10
11 parameter CNT_10MS = 500000 - 1;
12 parameter CNT_2S = 200 - 1;
13 parameter CHANGE_TIME = 100 - 1;
14 parameter PWM_OFFSET = 5000;
15
16 breath_led #(
17     .CNT_10MS(CNT_10MS),
```

```

18      .CNT_2S(CNT_2S),
19      .CHANGE_TIME(CHANGE_TIME),
20      .PWM_OFFSET(PWM_OFFSET)
21  ) inst_breath_led (
22      .clk    (clk),
23      .rst_n (rst_n),
24      .led    (led)
25  );
26
27 // clock
28 initial begin
29     clk = 0;
30     forever #(10) clk = ~clk;
31 end
32
33 initial begin
34     rst_n = 0;
35     repeat(10) @(posedge clk);
36     rst_n = 1;
37
38 end
39
40 endmodule

```

仿真文件第 5 到 7 行是输入输出的信号的定义，在访问文件中，激励信号用 `reg` 类型表示，中间模块间的信号和输出信号用 `wire` 类型。

11 到 14 行是本次实验的参数定义，通过修改参数，能够提高仿真效率，16 到 25 行是待仿真模块的例化，其中对带仿真的模块进行了参数的传递。

28 到 38 行是对输入信号的赋值。

仿真文件设计好后，即可进行仿真，Vivado 是一个集成式的开发环境，其带有仿真工具，可以直接使用 Vivado 自带的仿真工具来完成仿真，也可以使用第三方仿真工具，例如 Modelsim 来完成仿真。

## 2) 使用 Vivado 自带仿真工具完成仿真

选择 settings → Simulation → Vivado simulator → Apply → OK;

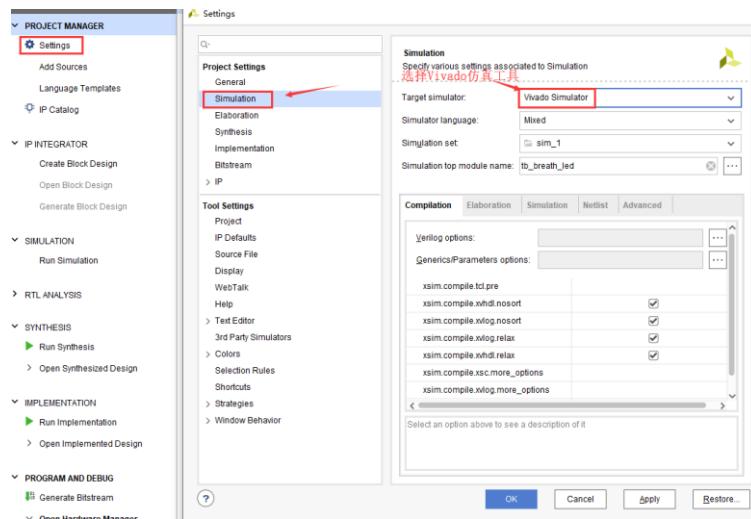


图 9-3

设置完成后，点击 Run simulation 即可开始仿真，接着 Vivado 会跳出如下图所示的仿真界面，其中红色窗口可以查看各个模块和 hierarchy，紫色窗口可以查看对应模块内部的信号，青色窗口可以查看对应信号的波形；

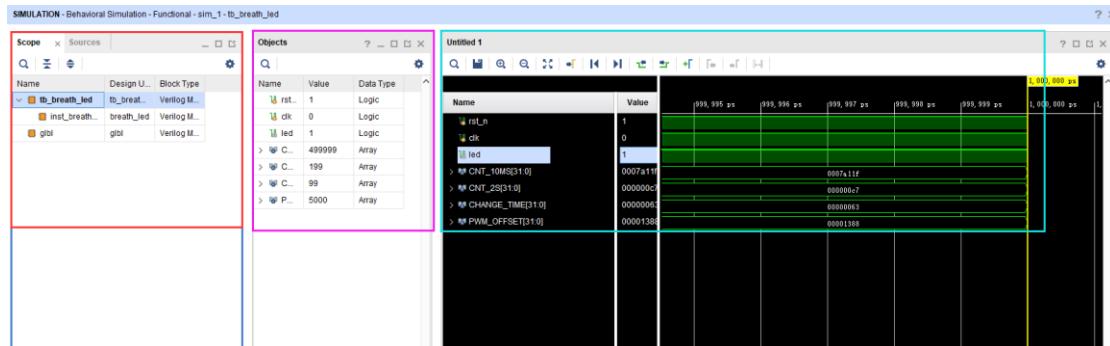


图 9-4

有想要观察的信号，只需将其选中，然后右键，添加到波形窗口中即可，如下图：

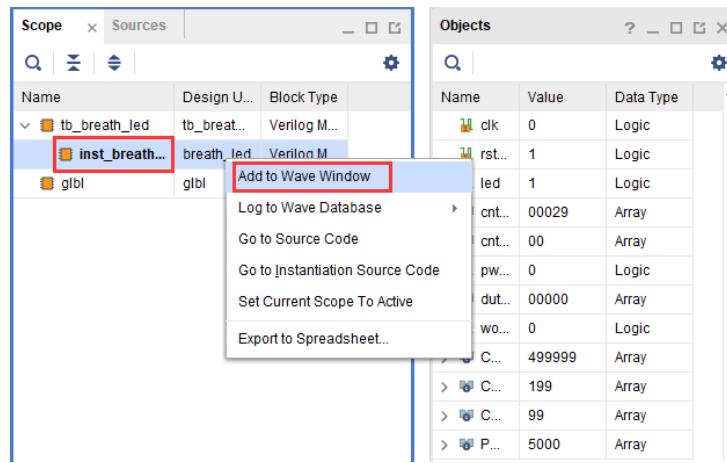


图 9-5

在仿真界面上方，有控制仿真开始选项，可以对仿真进行相关设置，如仿真时间等，在这里点击运行仿真，即可在波形窗口中，观察到想要显示的波形。

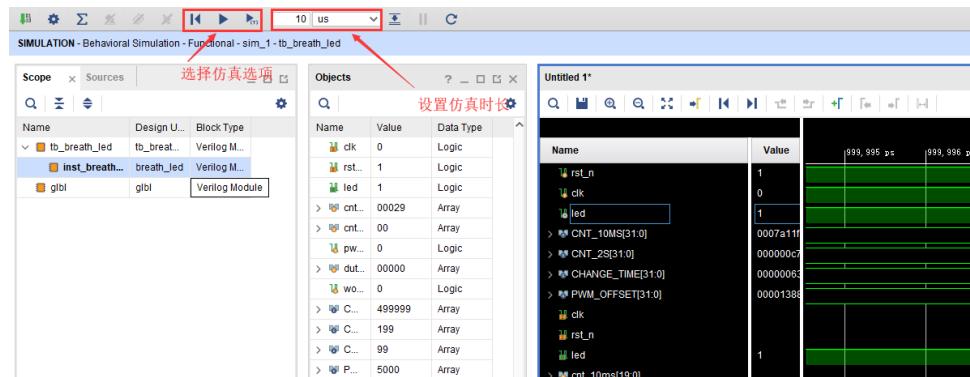


图 9-6

通过仿真可以发现，PWM 方波在两秒内，占空比从零增加至 100%，然后又逐渐减小到 0，通过仿真可以判断，源程序的功能正确，接下来可以进行生成 bit 文件下载验证了。

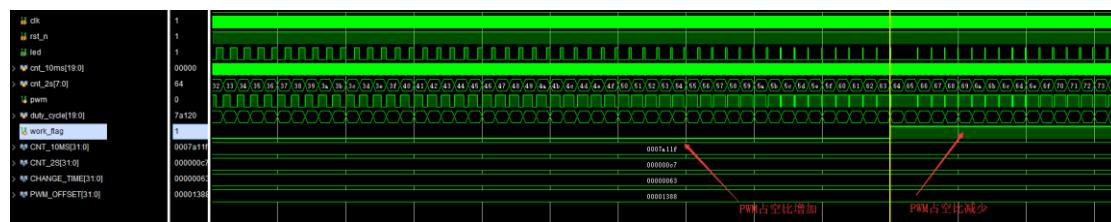


图 9-7

## 7.6 下载验证

### 1) 添加约束文件

本次实验的约束文件如下：

Z7-Lite 系列约束文件：

```
##### clock define#####
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
##### key define#####
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
#####LED define#####
set_property PACKAGE_PIN P15 [get_ports led]

set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports led]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
```

### 2) 下载验证

生成 bit 文件后，将 bit 文件下载进 FPGA，然后观察现象可以看到，LED1 从熄灭到点亮然后又熄灭，完成了一个呼吸灯的实验：

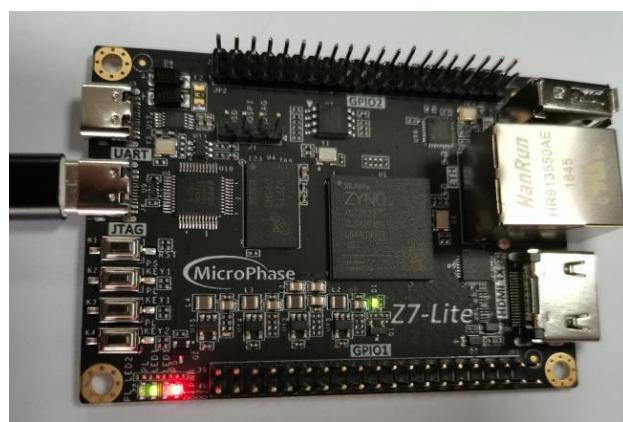


图 9-8

## 7.7 实验总结

通过本次实验，完成了一个呼吸灯的实验，本次实验了解到了 PWM 进行脉冲调制的方法。除此之外，本次实验还设计到了 FPGA 开发中非常重要的一个结构，计数器结构，在进行 FPGA 开发时，经常会使用到计数器来对某些信号进行计数，计数器在之后的开发中，经常会使用到，可能是在 FPGA 开发中，出现频率最高的一个结构。

关于计数器，需要弄清楚它的加一条件，即在什么情况下计数器加一；还有清零条件，即在什么情况下需要对计数器清零。弄清楚了这两个条件，计数器的结构就十分简单了。

## 第8章 片上资源之 PLL

在本章中，将向读者介绍 Vivado 使用赛灵思提供的 IP 进行开发和设计。本次实验将使用 modelsim 进行仿真。如果读者没有安装 modelsim 建议读者先安装好 modelsim 软件。当然读者也可以使用 Vivado 自带的仿真器对本次实验进行仿真。

### 8.1 实验简介

开发板给我们提供的时钟有晶振震荡之后，产生固定频率的信号，作为我们的时钟，但是这个时钟是相对比较固定的，但是在多样化的实验当中，可能需要不同频率的时钟，这时我们可以计数器分频对传入来的时钟对系统时钟进行分频，这是可以得到相比系统时钟频率更低的时钟，那如果需要得到比系统时钟频率更快的时钟的话，我们用户用逻辑代码来实现可能就没那么简单，或者说就算得到了对应频率快的时钟，也不一定能用，因为可能会产生一些毛刺。

此时，我们需要用 Vivado 工具里面自带的功能来进行时钟的倍频和分频。倍频的意思是由一个频率低的时钟产生一个频率更高的时钟，这里需要用到一个新的知识点，即 IP 核，IP 核意为知识产权，实际上就是一个封装好的模块，集成在了 Vivado 里面，如果需要此模块应用到实际项目中，就需要对其进行调取，调取过程中需要设置对应的参数具体详细的步骤下面会截图说明。

### 8.2 创建工程

打开 Vivado 后按照表格步骤进行操作。

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 ram_pp, 工程路径 E:\demo\FPGA\pll 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件: Z7-Lite7020 用户选中 xc7z020clg400-2, 购买 Z7-Lite7020 用户选中 xc7z010clg400-1
New Project summary	确认无误后, 点击确认

### 8.3 生成 IP 核步骤以及参数设置

1) 在导航栏中选择 IP Catalog, 会弹出 IP Catalog 的界面

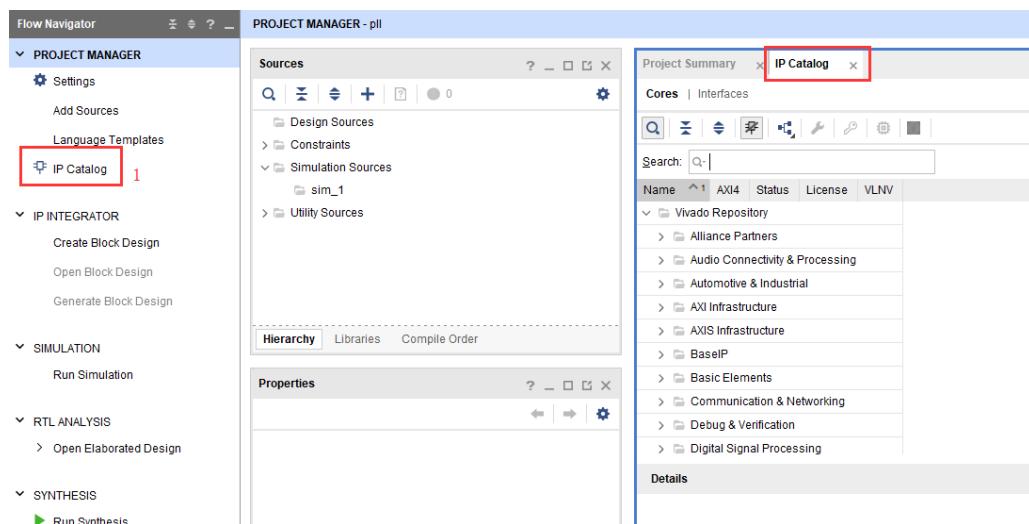


图 10- 1

2) 在搜索对栏中输入 clock, 选择 clocking wizard, 并双击弹出 clock IP 的配置界面

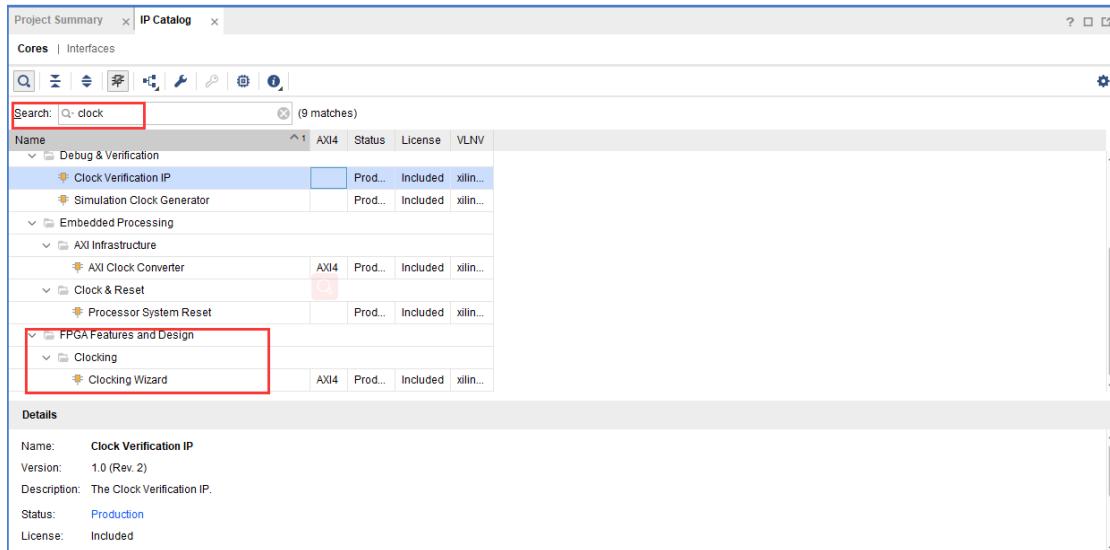


图 10- 2

3) 在配置界面的第一页，可以对 IP 名称进行修改，可以确定输入时钟的频率和类型，本实验中，将 IP 核名称修改为 clock，输入时钟为 200M，source 选项中选择 differential clock 差分时钟。因为板载的时钟管理芯片的输出是 200M 的差分时钟。设置完后选择下一页，output clocks。

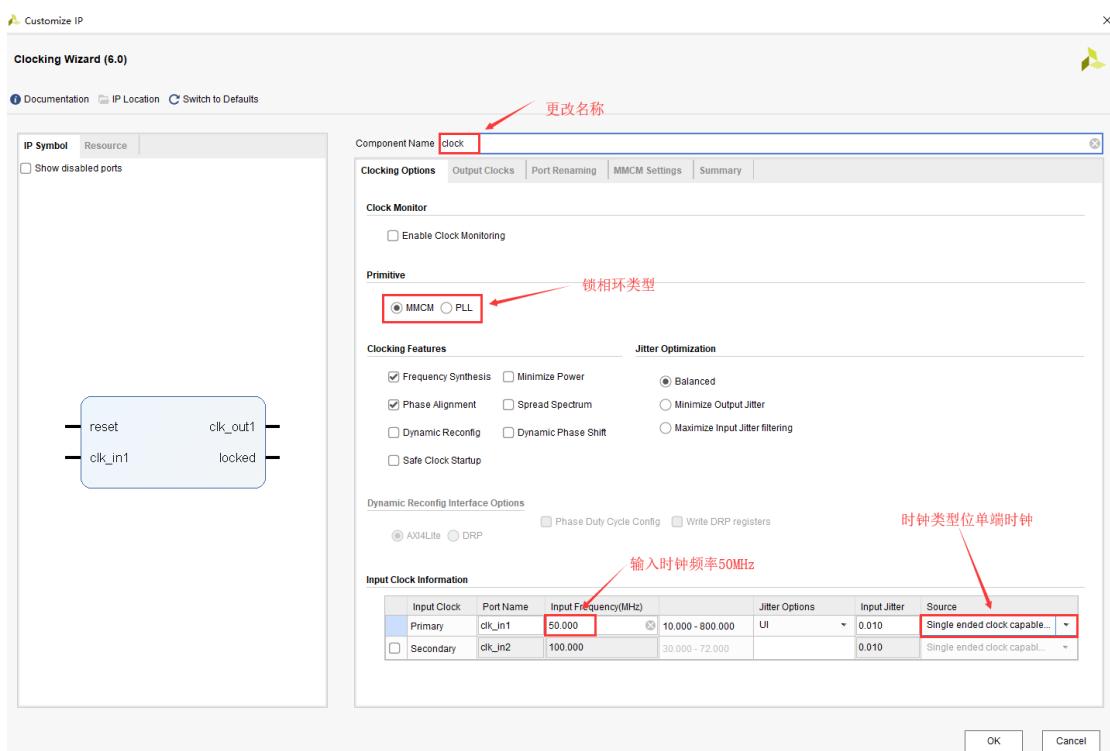


图 10- 3

4) 在 **output clocks** 界面，可以选择输出时钟的个数，输出时钟的频率，相位，占空比等，完成的输出时钟波形的控制，这本实验中，采用三个输出时钟，频率分别为 50M，100M。相位分别为 0，90。占空比保持默认。同样在本页面可以选择 IP 核的复位信号核锁定信号。在本实验中复位类型为高电平复位。**Locked** 信号是当所有输出时钟稳定后会拉高的一个信号，用来指示所有的信号已经稳定。

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drives
		Requested	Actual	Requested	Actual	Requested	Actual	
<input checked="" type="checkbox"/> clk_out1	clk_out1	50.000	<input type="button" value="X"/>	50.000	<input type="button" value="X"/>	0.000	<input type="button" value="X"/>	BUFG
<input checked="" type="checkbox"/> clk_out2	clk_out2	100.000	<input type="button" value="X"/>	100.000	<input type="button" value="X"/>	90.000	<input type="button" value="X"/>	BUFG
<input type="checkbox"/> clk_out3	clk_out3	100.000	<input type="button" value="X"/>	N/A	<input type="button" value="X"/>	0.000	<input type="button" value="X"/>	BUFG
<input type="checkbox"/> clk_out4	clk_out4	100.000	<input type="button" value="X"/>	N/A	<input type="button" value="X"/>	0.000	<input type="button" value="X"/>	BUFG
<input type="checkbox"/> clk_out5	clk_out5	100.000	<input type="button" value="X"/>	N/A	<input type="button" value="X"/>	0.000	<input type="button" value="X"/>	BUFG
<input type="checkbox"/> clk_out6	clk_out6	100.000	<input type="button" value="X"/>	N/A	<input type="button" value="X"/>	0.000	<input type="button" value="X"/>	BUFG
<input type="checkbox"/> clk_out7	clk_out7	100.000	<input type="button" value="X"/>	N/A	<input type="button" value="X"/>	0.000	<input type="button" value="X"/>	BUFG

输出时钟勾选即选中 时钟名称可修改

设置输出的频率

输出时钟相位

占空比

USE CLOCK SEQUENCING

Clocking Feedback

Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1
clk_out7	1

Source	Signaling
<input checked="" type="radio"/> Automatic Control On-Chip	<input checked="" type="radio"/> Single-ended
<input type="radio"/> Automatic Control Off-Chip	<input type="radio"/> Differential
<input type="radio"/> User-Controlled On-Chip	
<input type="radio"/> User-Controlled Off-Chip	

图 10- 4

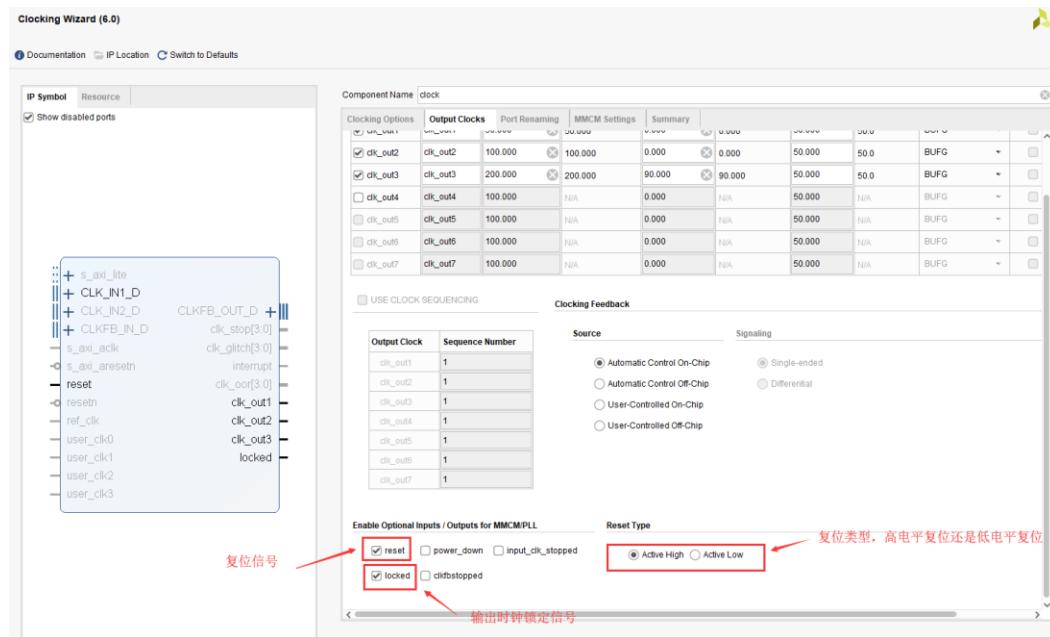


图 10-5

5) 其他选型保持默认即可，在 summary 中可以查看 IP 核配置信息，点击 OK，在弹出界面选择 generate，等待一段时间即可生成 IP 核。

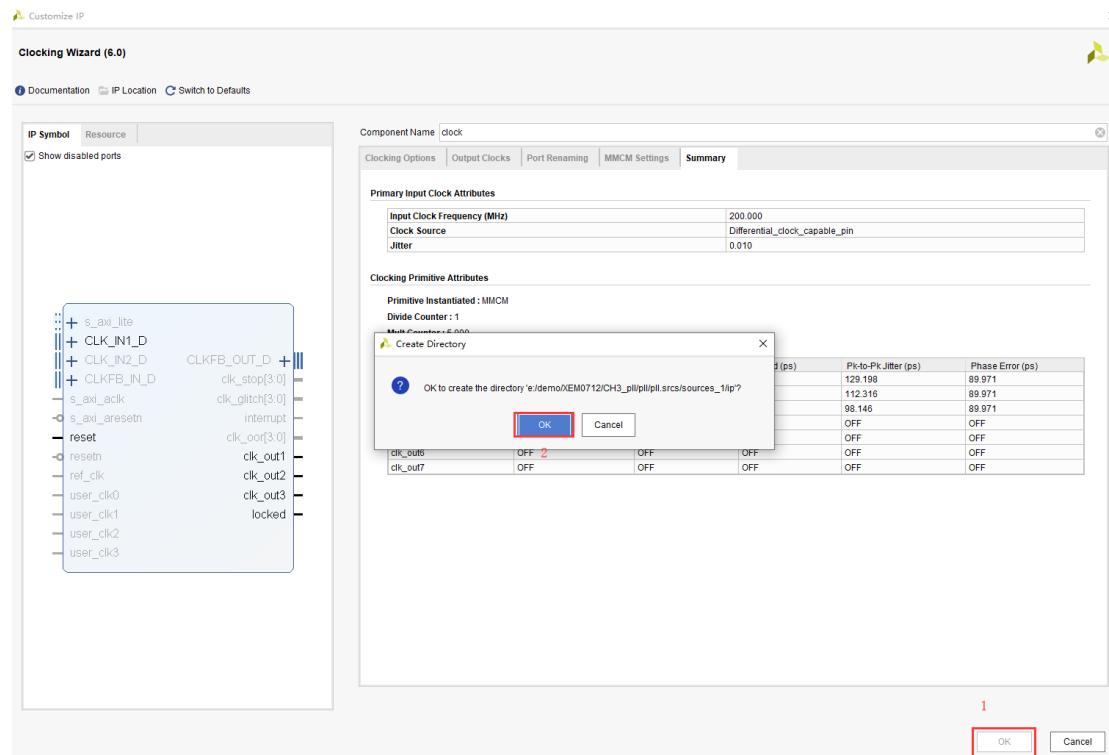


图 10-6

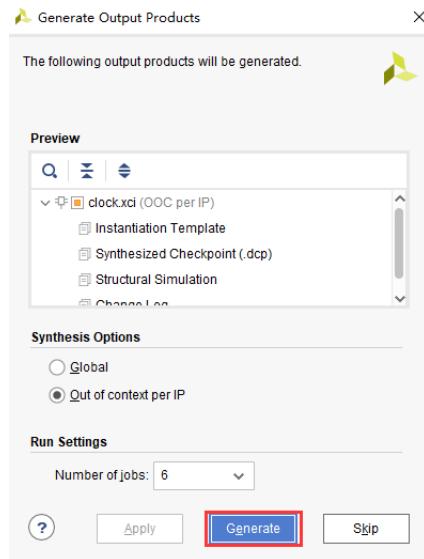


图 10-7

6) IP 生成完后可以在资源框中看到所生成的 IP

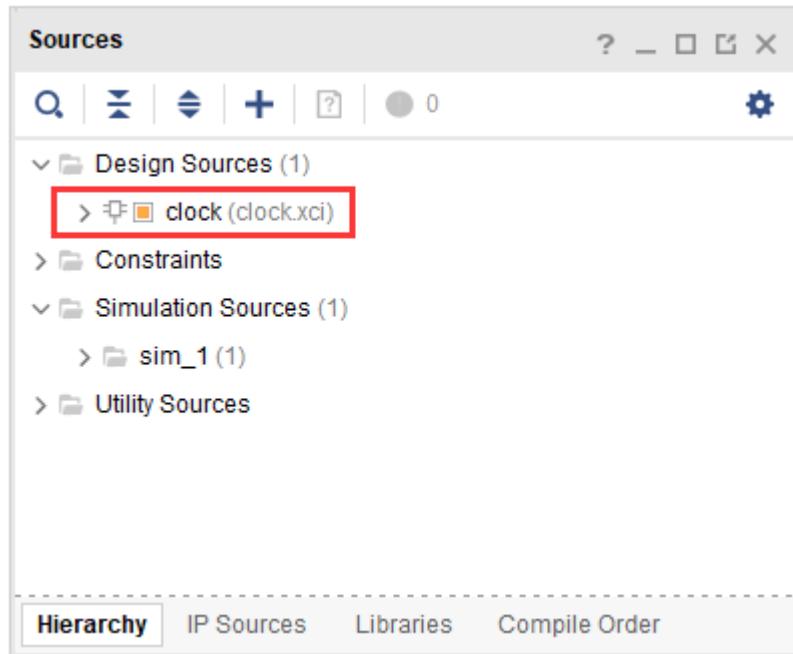


图 10-8

## 8.4 程序设计及调用 IP 核

IP 核生成完后，需要在其他文件中调用，才能发挥 IP 的作用。本次实验我

们将先对 IP 核进行仿真，然后利用 IP 核生成的时钟去驱动 Led 灯。

- 1) 添加一个 `pll_led.v` 文件，并编写程序。本次实验，通过调用所生成的 IP 核，来驱动 led 灯，在程序中，使用了 2 个相同的计数器，三个计数器的计数最大值相同，但 2 个计数器分别对所生成的 2 个不同的时钟进行计数。这样能够明显观测到 2 个 led 灯的亮灭是不一样的，证明实际生成了 2 个不同频率的时钟。
- 2) 在本次实验的程序编写过程中涉及到 IP 核的例化，例化 IP 核可以在使用 Vivado 提供的 IP 核例化模板。打开 source 窗口下的 IP source 找到 clock 的 Instantiation Template 下的 `clock.veo`，双击打开即可看到 `clock` IP 的例化模板，赋值该模板添加到设计程序中即可。

```

65
70 //----- Begin Cut here for INSTANTIATION Template ---// INST_TAG
71
72   clock instance_name
73   (
74     // Clock out ports
75     .clk_out1(clk_out1),      // output clk_out1
76     .clk_out2(clk_out2),      // output clk_out2
77     // Status and control signals
78     .reset(reset),           // input reset
79     .locked(locked),         // output locked
80     // Clock in ports
81     .clk_in1(clk_in1));      // input clk_in1
82 // INST_TAG_END ----- End INSTANTIATION Template -----
83

```

图 10-9

```

1 module  pll_led(
2   input   wire          clk      ,
3   input   wire          rst_n   ,
4   output  reg [1:0]     led
5
6 );
7
8 //=====
9 //parameter define
10 //=====
11 parameter  CNT_MAX    =  50_000_000 - 1;
12
13 //===

```

```
14 //internal signals
15 //=====
16 reg      [27:0]    timer1;//counter for clk_out1
17 reg      [27:0]    timer2;//counter for clk_out2
18 wire          locked;
19 wire          clk_out1;
20 wire          clk_out2;
21
22
23 //-----timer1-----
24 always  @(posedge clk_out1 or negedge rst_n)begin
25     if(rst_n==1'b0)begin
26         timer1 <= 'd0;
27     end
28     else if(locked==1'b1)begin
29         if(timer1==CNT_MAX)
30             timer1 <= 'd0;
31         else
32             timer1 <= timer1 + 1'b1;
33     end
34     else begin
35         timer1 <='d0;
36     end
37 end
38
39 //-----timer2-----
40 always  @(posedge clk_out2 or negedge rst_n)begin
41     if(rst_n==1'b0)begin
42         timer2 <= 'd0;
43     end
44     else if(locked==1'b1)begin
45         if(timer2==CNT_MAX)
46             timer2 <= 'd0;
47         else
48             timer2 <= timer2 + 1'b1;
49     end
50     else begin
51         timer2 <='d0;
52     end
53 end
54
```

```
55
56 //-----led1-----
57 always  @(posedge clk_out1 or negedge rst_n)begin
58     if(rst_n==1'b0)begin
59         led[0] <= 1'b1;
60     end
61     else if(timer1 == CNT_MAX)begin
62         led[0] <= ~led[0];
63     end
64     else begin
65         led[0] <= led[0];
66     end
67 end
68
69 //-----led2-----
70 always  @(posedge clk_out2 or negedge rst_n)begin
71     if(rst_n==1'b0)begin
72         led[1] <= 1'b1;
73     end
74     else if(timer2 == CNT_MAX)begin
75         led[1] <= ~led[1];
76     end
77     else begin
78         led[1] <= led[1];
79     end
80 end
81
82
83 clock clock_inst
84 (
85     // Clock out ports
86     .clk_out1(clk_out1),      // output clk_out1
87     .clk_out2(clk_out2),      // output clk_out2
88     // Status and control signals
89     .reset(~rst_n), // input reset
90     .locked(locked),        // output locked
91     // Clock in ports
92     .clk_in1(clk)           // input clk_in1
93 );
94
95 endmodule
```

## 8.5 对 IP 核及顶层进行仿真

在对设计进行仿真的时候，需要先进行仿真文件的编写，仿真文件也是一个.v 文件，也同样采用 Verilog 进行编写。

1) 首先点击导航栏的 Add Source，在弹出界面选择添加 Simulation Sources

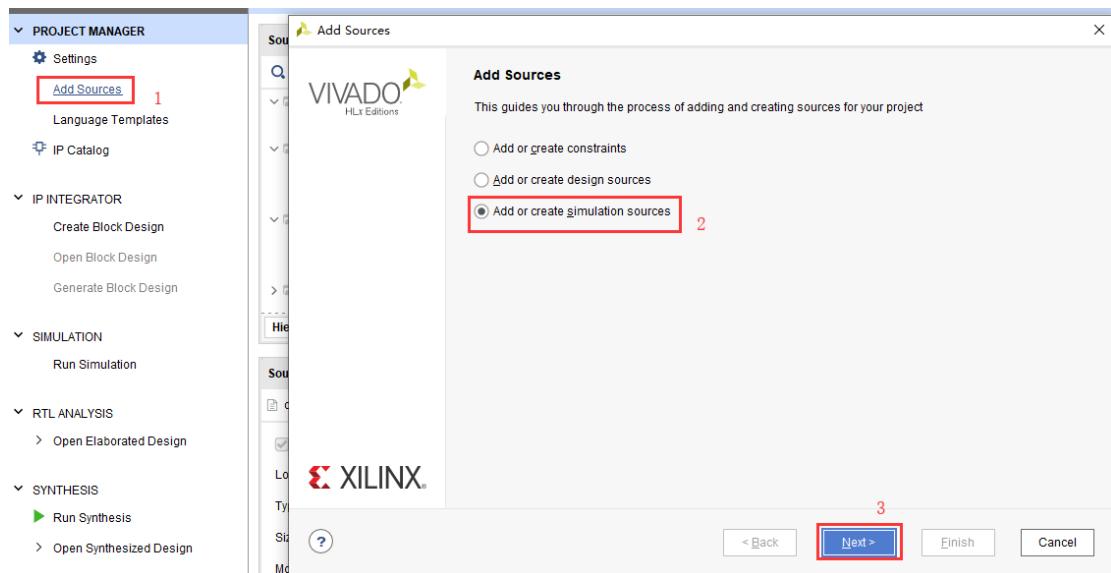


图 10- 10

2) 选择 create file，将文件名命名为 tb\_pll\_led 点击确定。

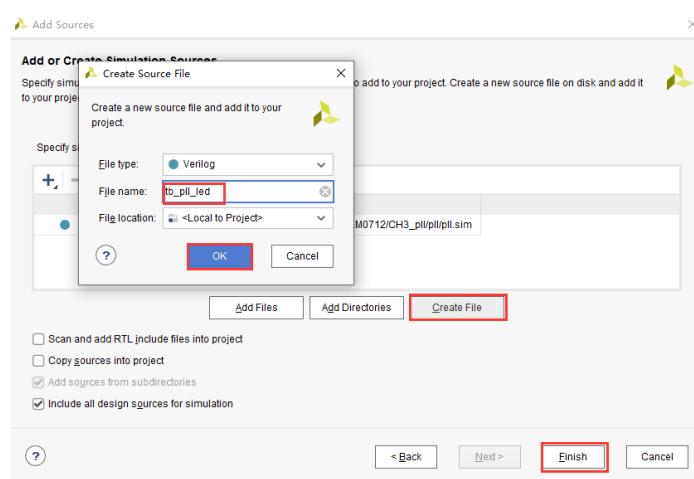


图 10- 11

3) 添加完文件后，可以在 source 栏查中的 simulation source 中查看仿真文件的层次结构

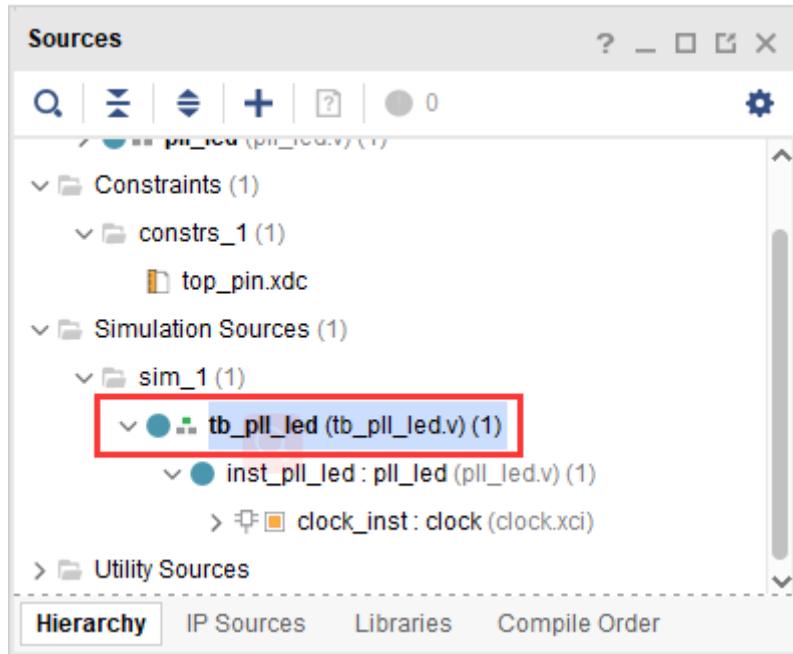


图 10-12

4) 接下来介绍仿真文件的编写

首先需要仿真步长和仿真精度 即第一行的`timescale 1ns/1ps 其中，1ns 指仿真步长，1ps 指仿真精度。

然后是仿真模块的定义，这个定义与定义设计文件的模块类似。

然后是信号类型的定义，`reg` 类型的变量是激励信号，由外部输入或者本模块内部产生。`Wire` 类型的变量是用于观测输出信号或者模块之间的信号连接。

接下来是例化要仿真的模块，将我们前面设计的模块例化到仿真文件中。

最后是激励信号的编写，通过激励信号，模拟我们想要对模块进行的操作，来验证我们的设计是否正确，从而对设计进行 debug。

值得说明的是，在仿真时使用参数化设计，能够加快仿真，提高仿真效率。例如，本次实验中，可以对 `CNT_MAX` 这个参数进行修改，从而改变计数器的计

数最大值，减少仿真时间。

```
'timescale 1ns/1ps

module tb_pll_led(); /* this is automatically generated */

parameter CNT_MAX = 100_000 - 1;

reg      clk;
wire [1:0] led;
reg      rst_n;

pll_led #( 
    .CNT_MAX(CNT_MAX)
) inst_pll_led (
    .clk      (clk),
    .rst_n    (rst_n),
    .led      (led)
);

initial begin
    clk = 0;
    forever #(10) clk = ~clk;
end

initial begin
    rst_n = 0;
    #500;
    rst_n = 1;
end
endmodule
```

## 5) 仿真准备

在这里分别使用 Vivado 自带的仿真器和第三方仿真器对仿真文件进行仿真，用户可根据自己的喜好选择仿真方式。

首先介绍 Vivado 自带的仿真器，Vivado 是一个全功能的设计平台，其中包括了仿真平台，我们可以从导航栏中，直接看到仿真这一选项。

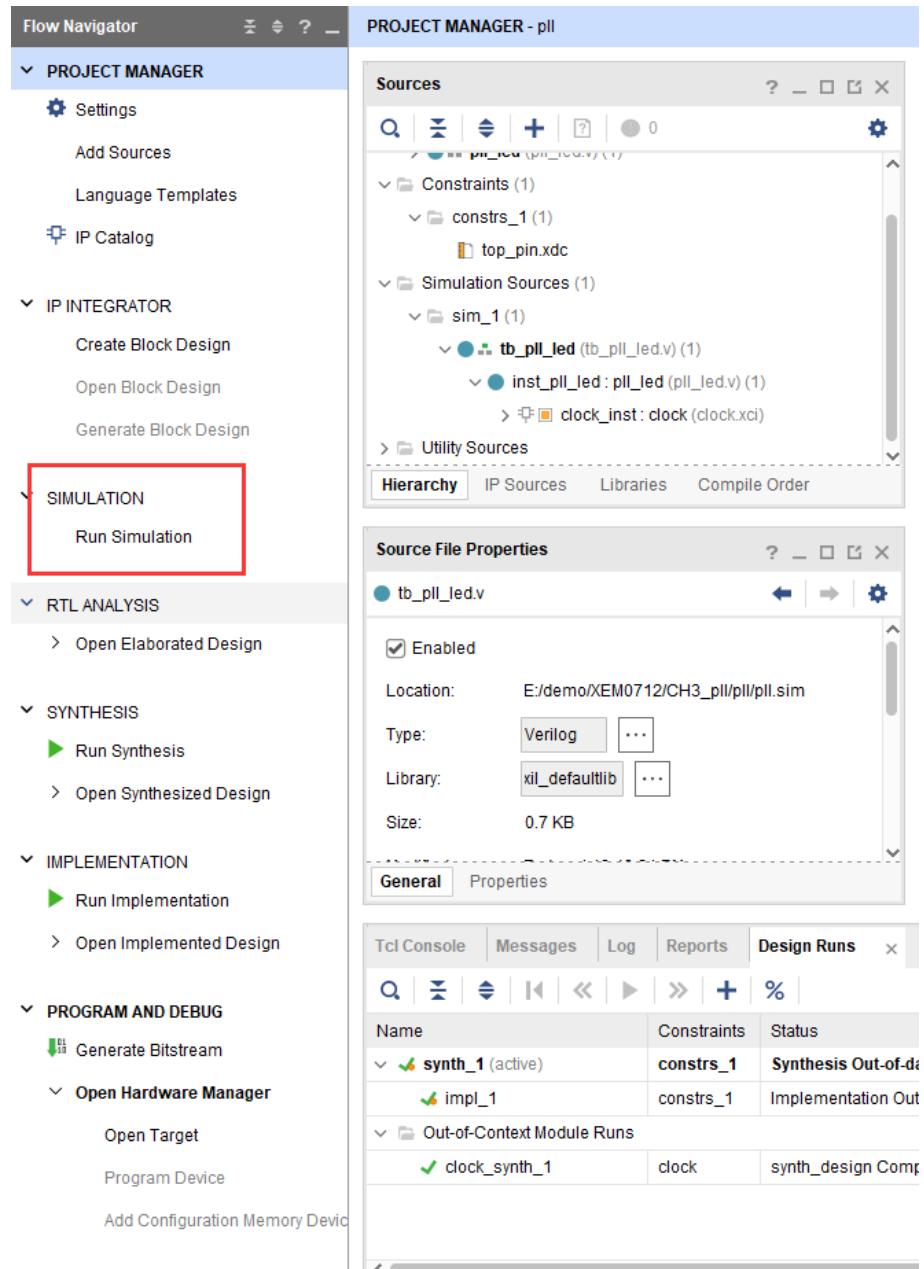


图 10-13

我们可以直接点击 run simulation 进行仿真，在这里选择行为仿真，等待一段时间后，Vivado 会弹出仿真界面。

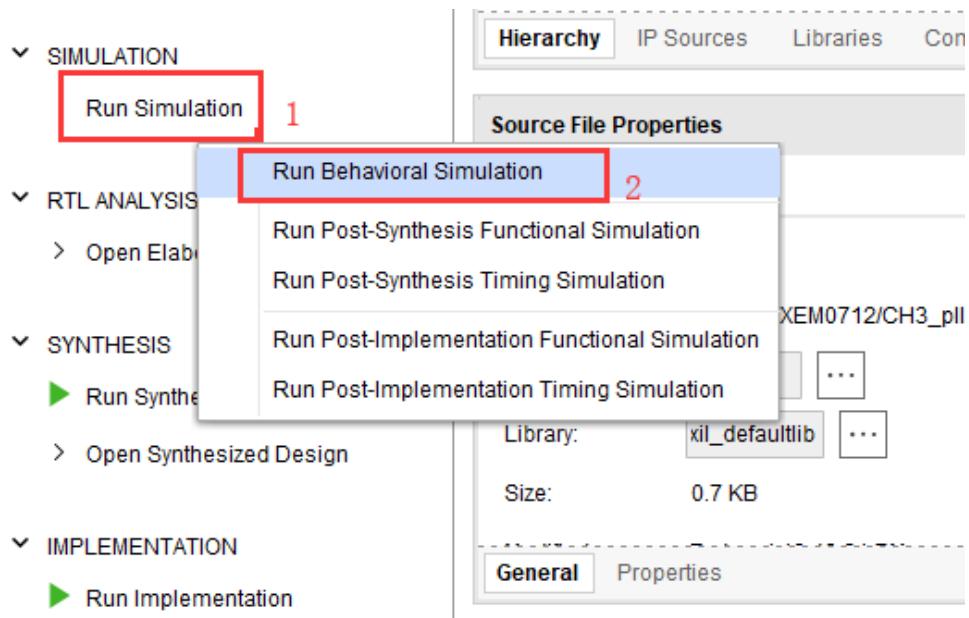


图 10-14

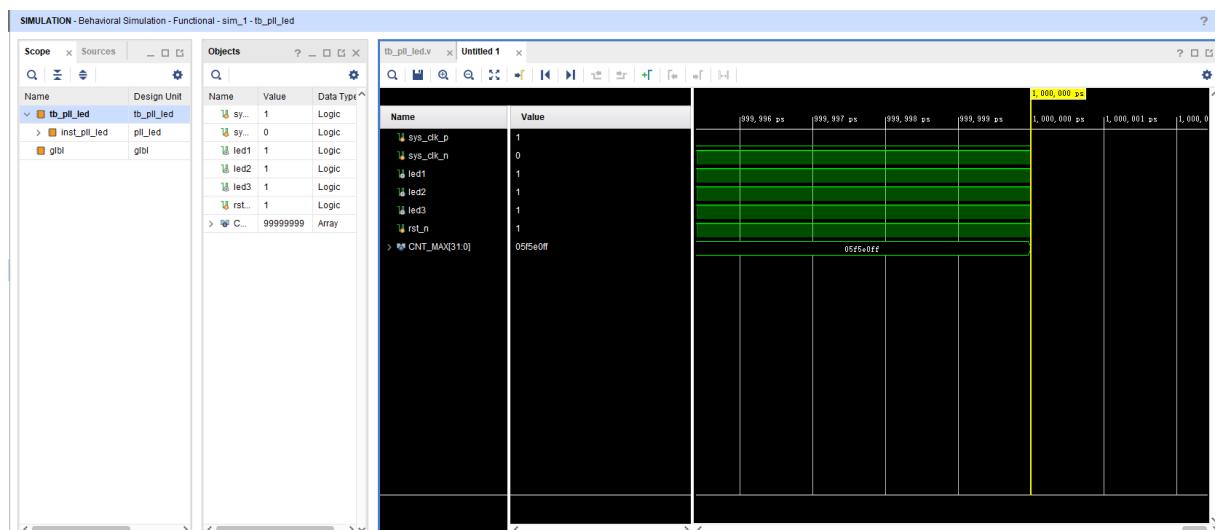


图 10-15

我们将模块中的其他信号添加到波形窗口中，右键 `inst_pll_led`，选择 `Add to Wave window`

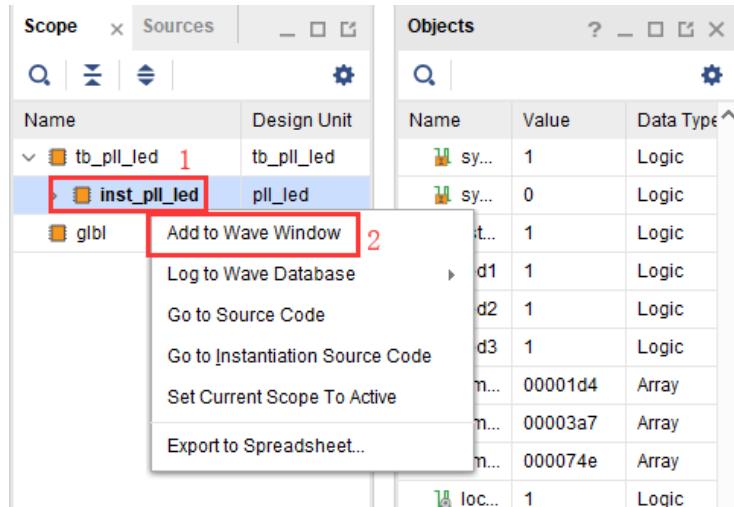


图 10-16

通过波形窗口，我们可以观察到 `clk_out1`, `clk_out2`, `clk_out3` 相对于输入的差分时钟 `sys_clk` 之间的关系。从波形图中可以看出，`clk_out1` 是频率为 50MHz 的时钟，`clk_out2` 为 100MHz 的时钟

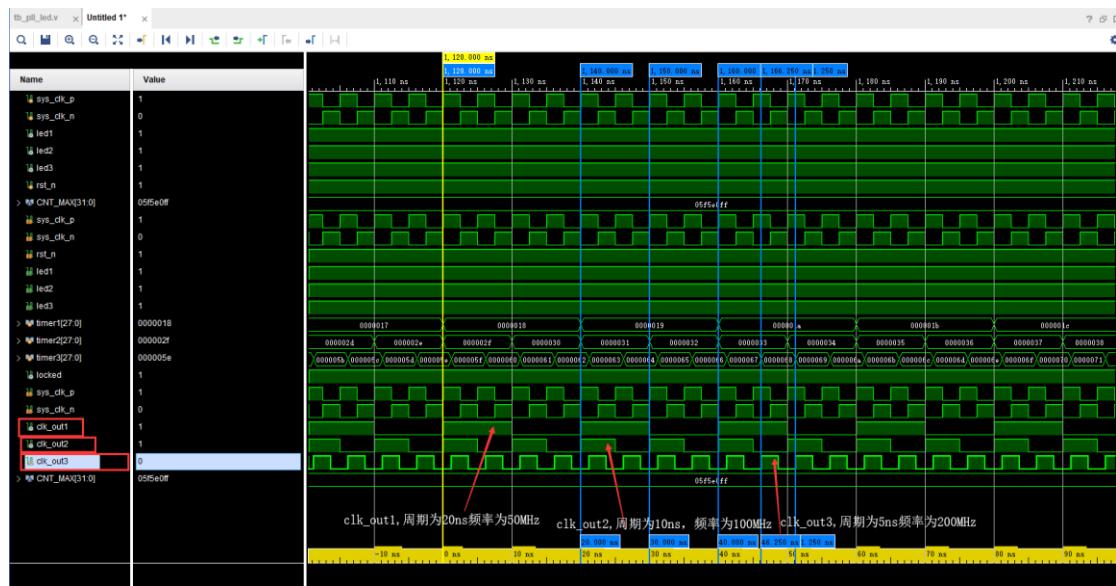


图 10-17

6) 接下来介绍使用第三方仿真软件 modelsim 对设计进行仿真

点击 tools -> settings

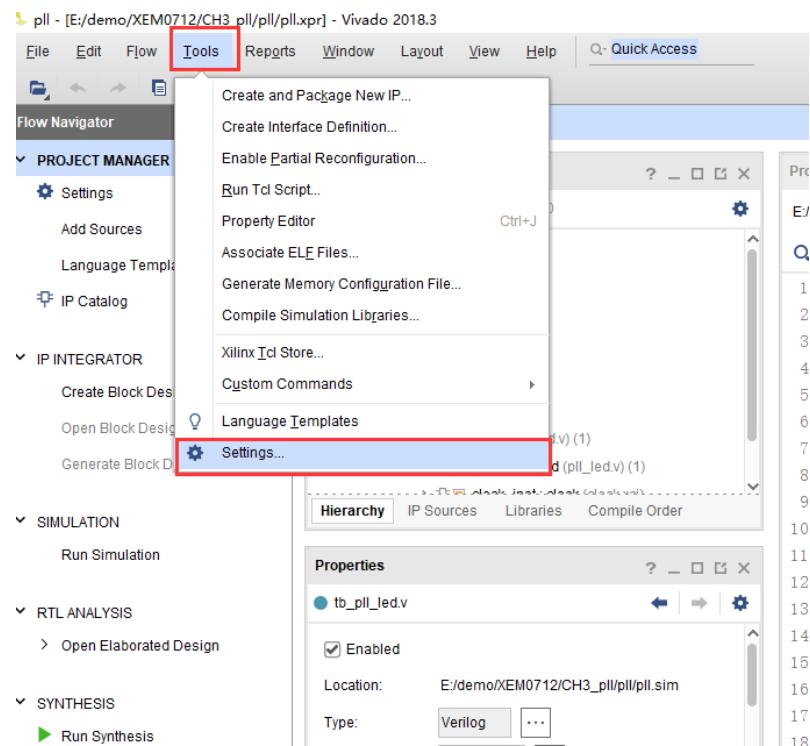


图 10-18

在 simulation 选项中选择 Midelsim simulator

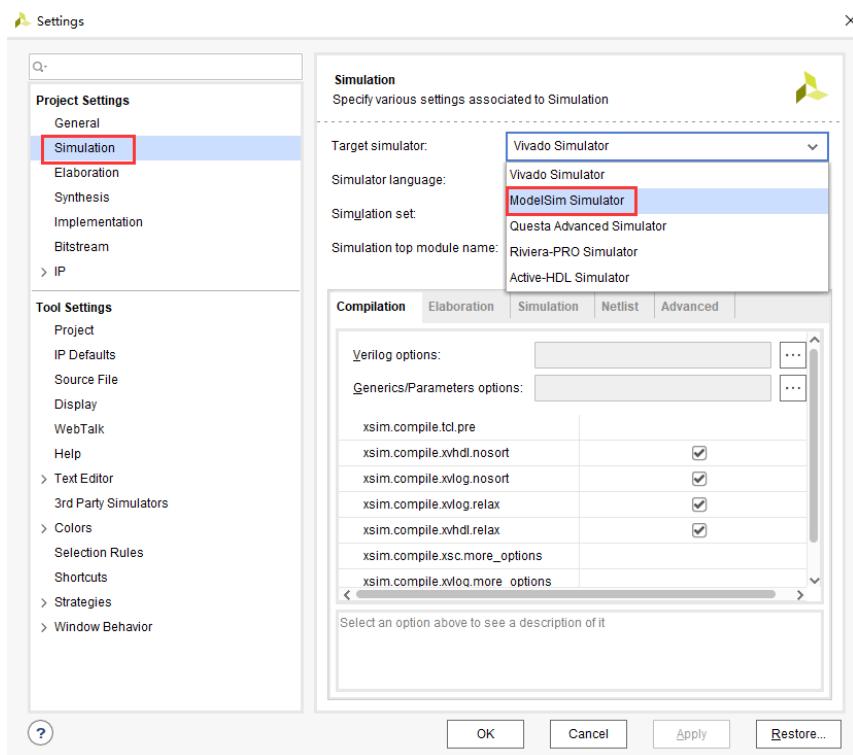


图 10-19

在弹出界面选择 yes

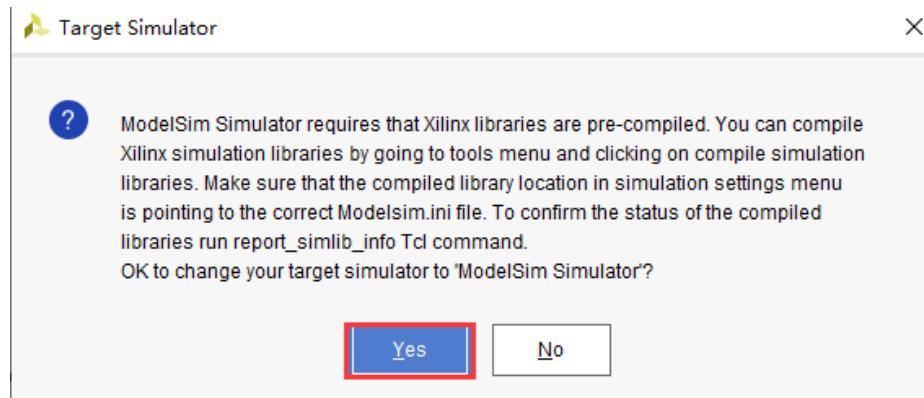


图 10- 20

接着点击第三方仿真工具，在 Modelsim 后选择安装路径，这里是用户安装 modelsim 软件的位置，注意要选择到 win64 这个目录。然后选择库文件存放的目录，用户可以自己创建一个库文件目录，用来存放仿真过程中的库文件。接下来点击 apply - > OK。

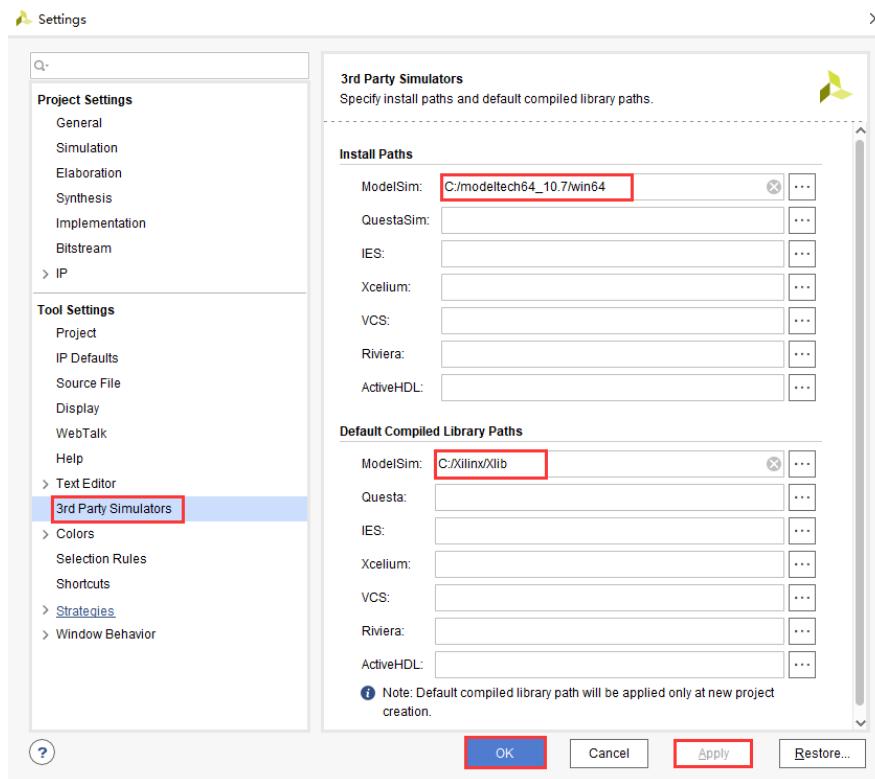


图 10- 21

设置完毕后，点击 run simulation 即可打开 modelsim 软件进行仿真。

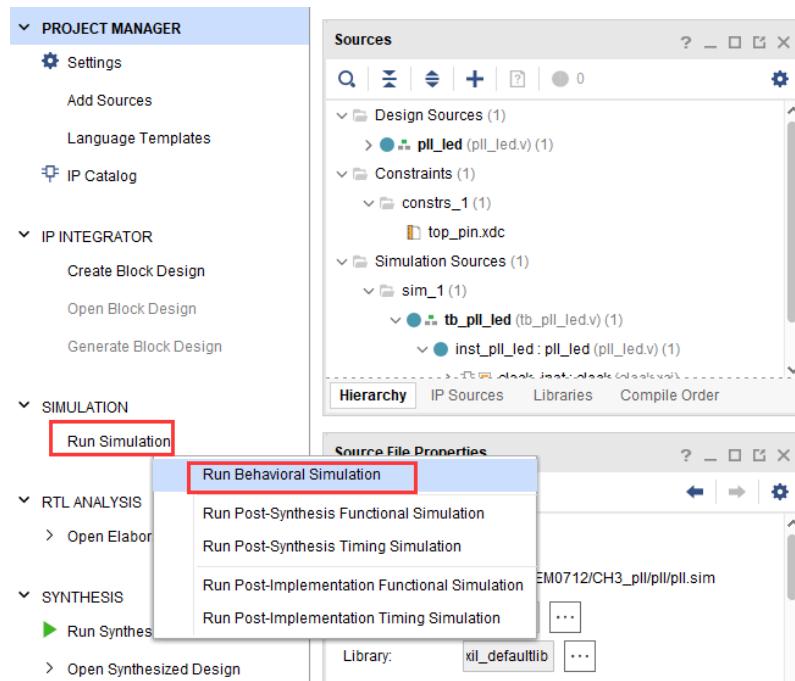


图 10- 22

打开 modelsim 后界面如图所示，其中红色部分可以查看各个文件，编译文件，和添加想要观察的信号。

紫色部分是打印的脚本信息，用户也可以使用命令行来对 modelsim 进行操作，实现仿真的控制。

黄色部分是模块中的信号及其状态。

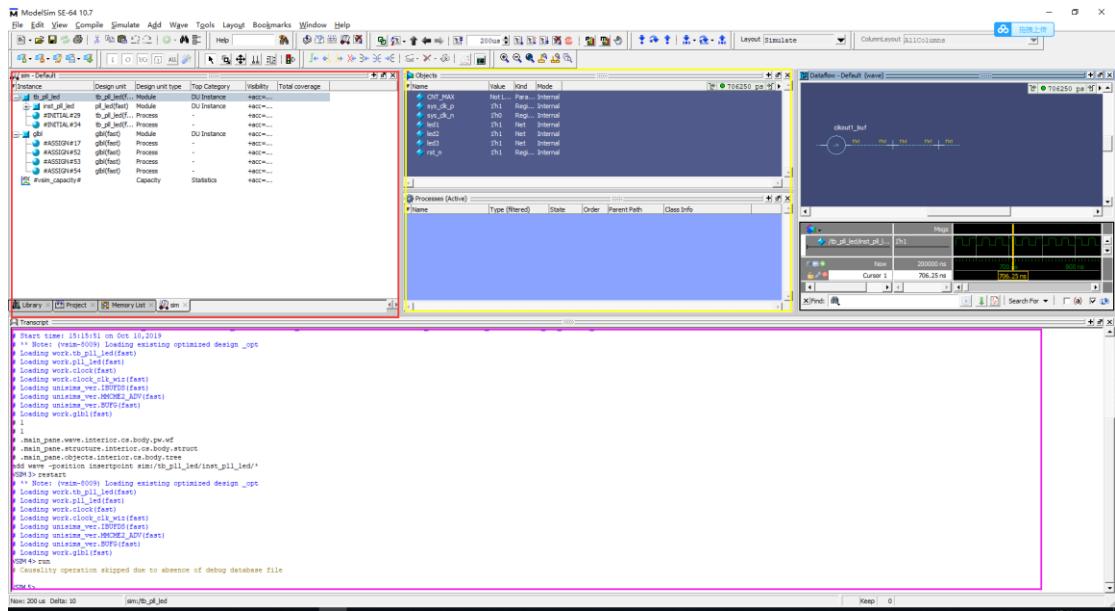


图 10-23

接下来选择将要观察的信号添加到波形窗口中，在红色区域，点击想要观察的信号，右键->add wave 或者使用快捷键 **Ctrl + w** 即可将波形添加到波形窗口。进入波形窗口后，可以使用快捷键 **Ctrl+A**，选中所有信号，**Ctrl + G** 对所有信号进行分组。通过这个操作，可以将不同模块中的信号进行分组，便于观察。

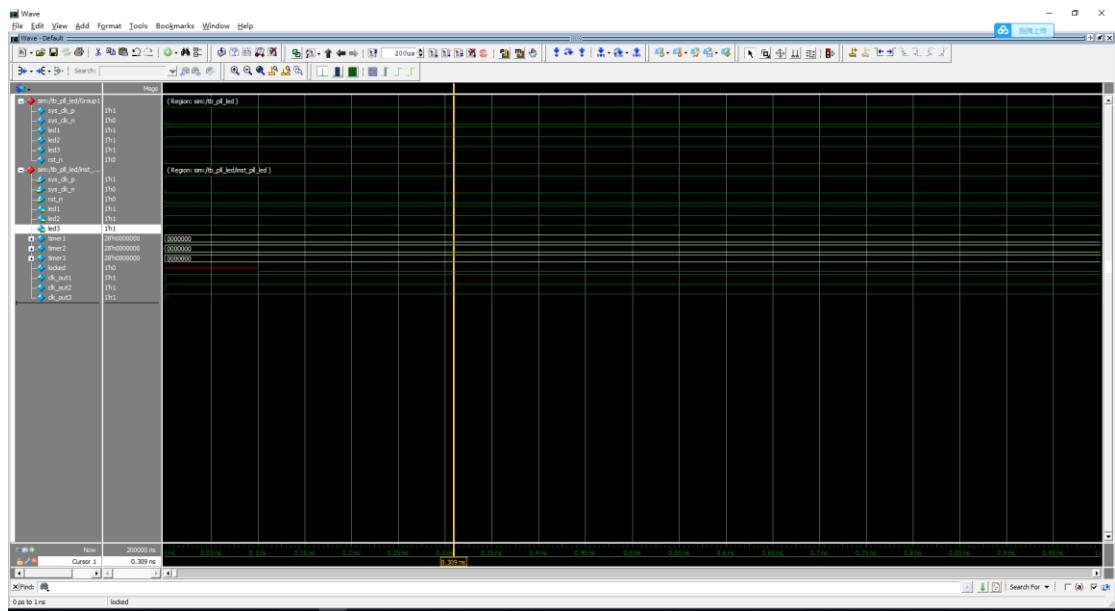


图 10-24

可以看到，此时仿真时间太短，看不到我们想要的波形，我们可以通过调整仿真时间，得到更多的波形信息。在菜单栏更改仿真时间，然后点击 run。

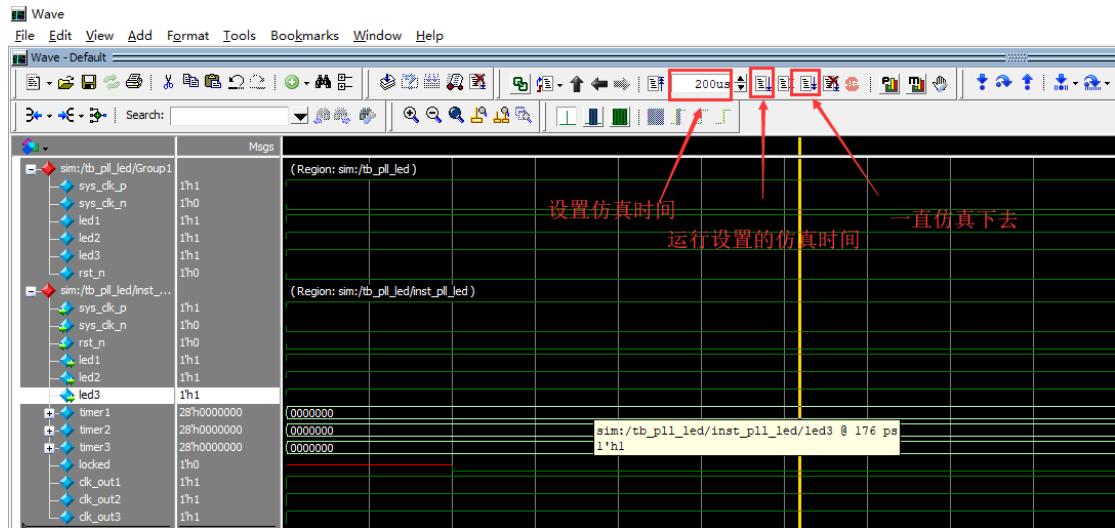


图 10- 25

在菜单栏，选择放大镜一样的图标，可以观察全部波形信息。按住 Ctrl 键让后滚动鼠标滚轮可以对局部进行放大或者缩小，便于观察信号。

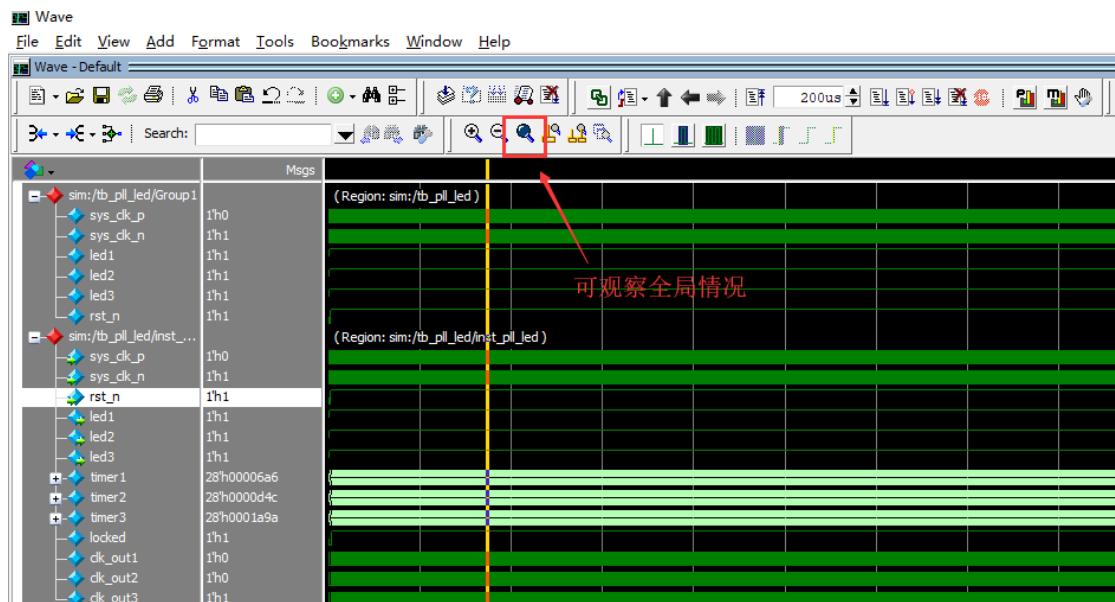


图 10- 26

接下来，我们重点由 IP 核输出的 2 个时钟信号和输入的系统时钟之间的关系。如下图所示：

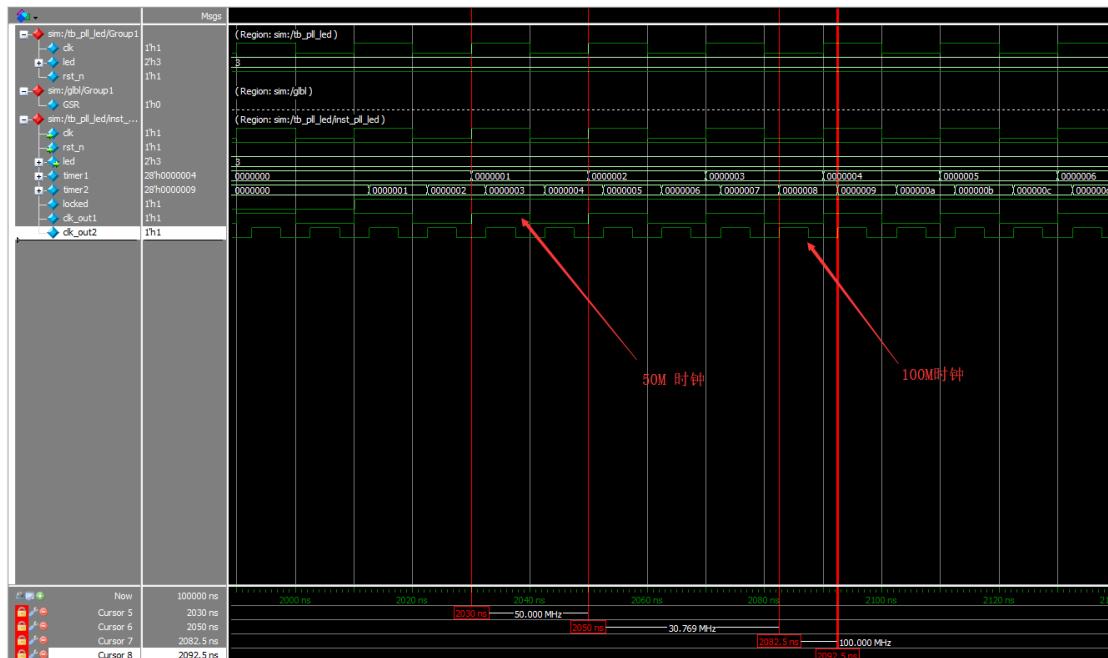


图 10-27

通过添加光标，可以确定每一个输出时钟的周期。可以看到，`clk_out1`, `clk_out2` 的周期分别为 20ns, 10ns。即说明 `clk_out1`, `clk_out2` 的频率分别为 50M, 100M。此外还能发现，`clk_out2` 与输入的时钟并没有对其，而是偏移了 90°，这与我们在配置 IP 核时所设置的一样，相对系统时钟偏移了 90°。

7) 至此对于测试模块仿真就完成了

## 8.6 添加管脚约束

1) 首先添加一个约束文件点击 Add Source -> Add or Create Constraints -> ok;

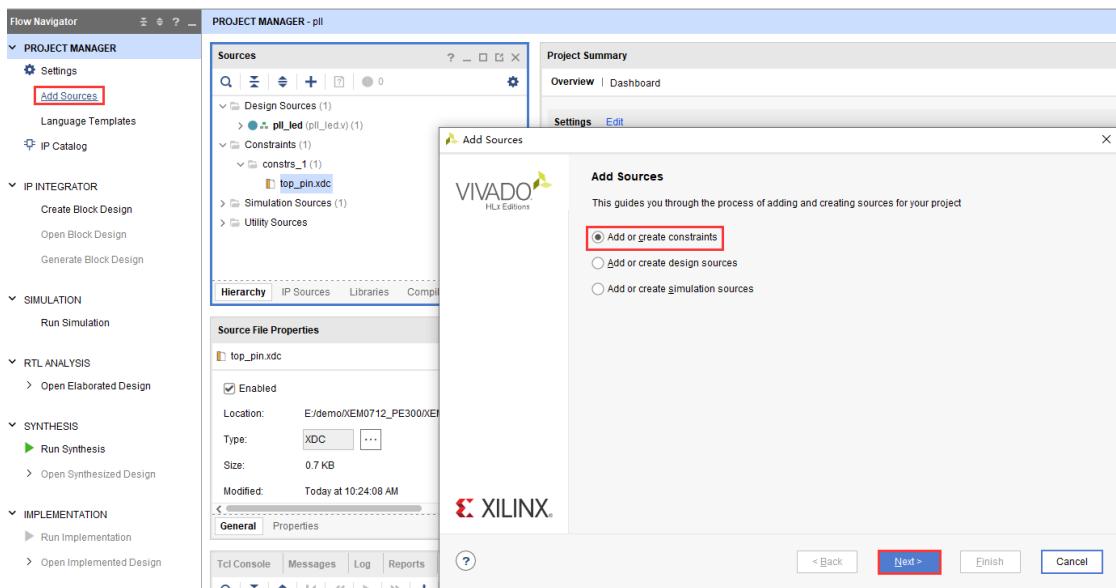


图 10- 28

2) 添加一个名为 top\_pin 的.xdc 文件

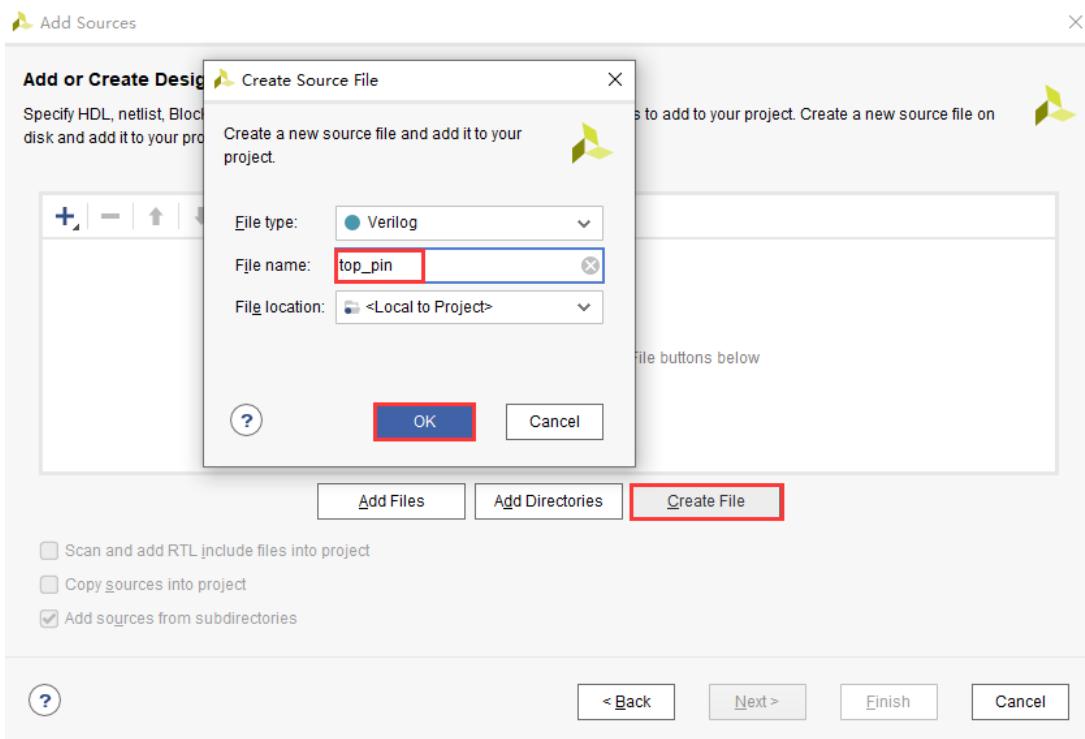


图 10- 29

3) 编写约束文件，约束文件与前面几次实验的大体相同，这里不做过多解释

```
##### clock define#####
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
##### key define#####
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]

#####LED define#####
set_property PACKAGE_PIN P15 [get_ports {led[0]}]
set_property PACKAGE_PIN U12 [get_ports {led[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports {led[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
```

## 8.7 下载验证

- 1) 接下来将到开发板上验证本次实验的最终结果。点击生成 bit 文件，等 bit 文件生成完毕后，将开发板连接好下载器。
- 2) 将 bit 文件下载到开发板中开始观察现象
- 3) 可以看到 2 个 led 等以不同的频率在闪烁。并且 2 个灯之间的闪烁刚好能够构成一个二级制加一的现象。

## 8.8 实验总结

通过本次实验，我们初步了解了 Vivado 下使用 IP 核进行开发的步骤，其实使用 IP 核开发 FPGA 程序的时候，都是采用这个步骤。在将来的章节中，我们还会使用到其他的 IP 核，也是使用这种方法来进行开发。

除此之外，我们还了解了，FPGA 开发中的仿真流程，可以说，仿真时 FPGA

开发流程中最重要的一个步骤，通过仿真我们能够快速定位到我们的设计中存在的问题，所以在这里希望大家要重视仿真。

我们为大家提供了两种仿真的方式，一种时使用 Vivado 自带的仿真工具进行仿真，一种是使用 modelsim 进行仿真，用户可以根据自身习惯的仿真工具进行仿真。我们推荐使用 modelsim 进行仿真，modelsim 是业内专业的仿真工具，相较于 Vivado 自带的仿真工具还是具有一定的优势。

## 第9章 片上资源之 RAM

### 9.1 实验背景

说到 FPGA，不得不提的是存储器，当我们做相关项目时，经常会遇到存储数据的问题，数据量过大时，我们可以将其存储在 FPGA 芯片的外设存储器上，比如 sdram、ddr sdram、ddr3 sdram 等，然而访问外设存储器相对比较麻烦，因此当数据量较小时，我们可以直接使用 FPGA 芯片内部自带的 ram 的 IP 核。

RAM 是 random access memory 的简称，即随机存储器的意思，ram 可以按照所需进行随机读/写。我们可以通过调用 FPGA 内部的 IP 核生成一个 ram，并通过编写 Verilog HDL 代码控制该 ram。

本章熟悉 RAM IP 的生成及相关设置，并编写操作两个简单双口 RAM 进行乒乓操作的程序，并对其进行仿真。

### 9.2 创建工程

双击桌面的 Vivado 2018.3 图标，打开开发环境，并按如下表格进行操作

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 ram_pp, 工程路径 E:\demo\Z7-Lite\ram_pp 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件，购买 Z7-Lite7020 用户选中 xc7z020clg400-2，购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后，点击确认

## 9.3 生成 IP 核及参数配置

本次实验配置一个简单双口 RAM，所谓双口 RAM，就是既可以读，又可以写的 RAM。在写端口，给出想要写入的数据和需要写入的地址，在写使能有效的情况下，在写时钟的上升沿即可将数据写入到 RAM 中；在读端口，给出想要读取的地址，在读使能有效的情况下，即可将数据从 RAM 中读出。接下来，我们来配置 RAM 的 IP 核。

单击 IP Catalog，在右边窗口 Search 位置输入 ram，在 Memories & Storage Elements 下可以看到有两个与 RAM 创建的入口

一个是 Distributed MemoryGenerator，另一个是 Block Memory Generator，两者最主要的区别是生成的 Core 所占用的 FPGA 资源不一样，从 Distributed Memory Generator 生成的 ROM/RAM Core 占用的资源是 LUT（查找表，查找表本质就是一个小的 RAM）；从 Block Memory Generator 生成的 ROM/RAM Core 占用的资源是 Block Memory（嵌入式的硬件 RAM）。用户根据实际资源使用需求选择创建的入口。这里主要讲解 Block RAM IP 的创建过程。双击 Block Memory Generator，弹出配置界面。

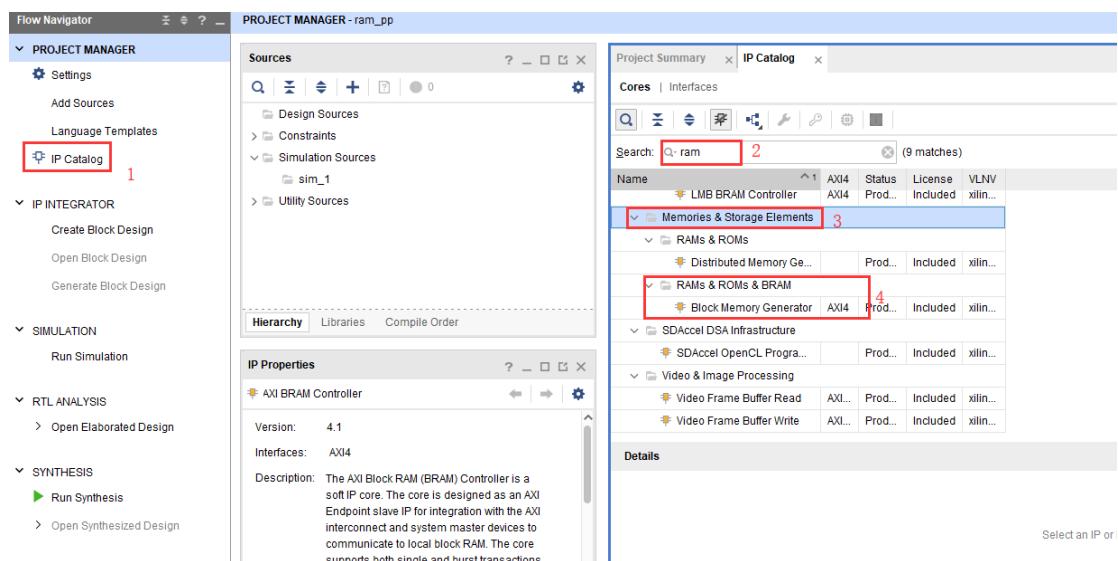


图 11-1

### 1) RAM 的基本配置

可以在本配置界面中，可以修改生成 IP 的名称，可以选择接口类型，这里选择 Native 类型，选择 RAM 的类型为 Simple Dual Port RAM；

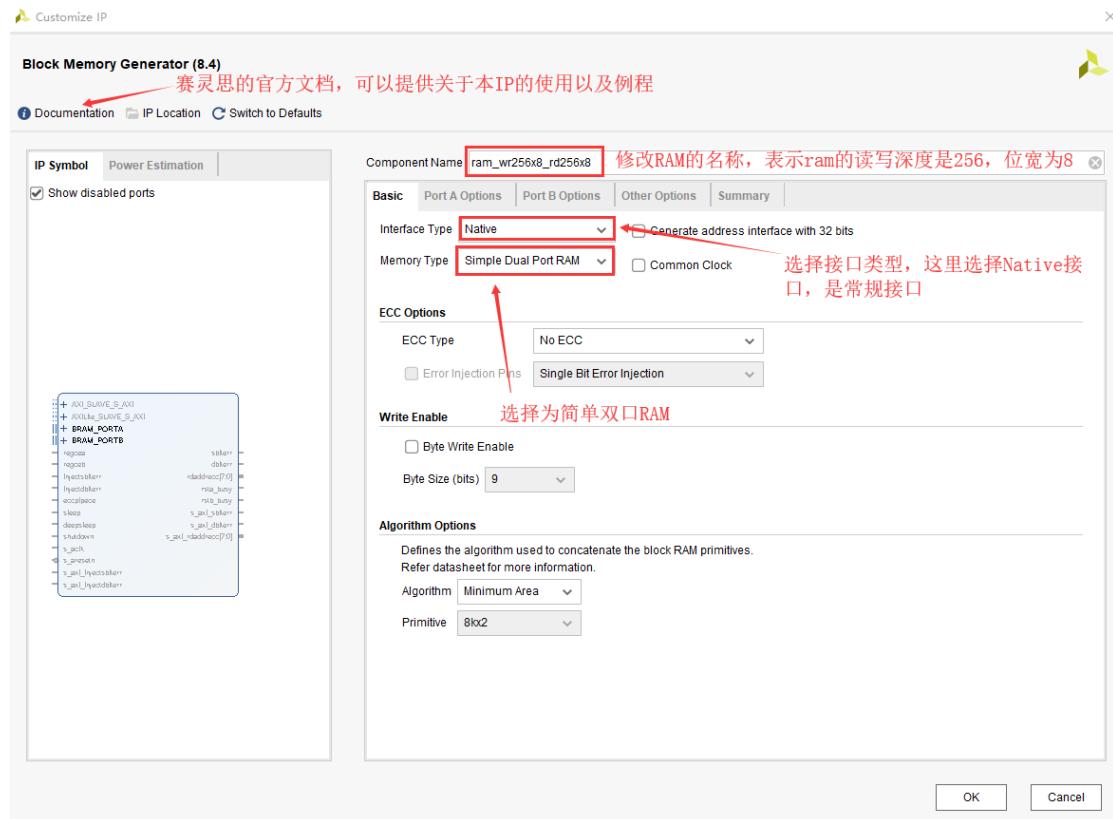


图 11- 2

## 2) 接下来配置输入端口

在输入端口可以选择配置输入端口的位宽，可以选择配置输入端口的深度。在本实验中，输入端口的位宽选择为 8，输入的深度为 256，Enable Port Type: use ENA Pin（使用使能控制信号）。

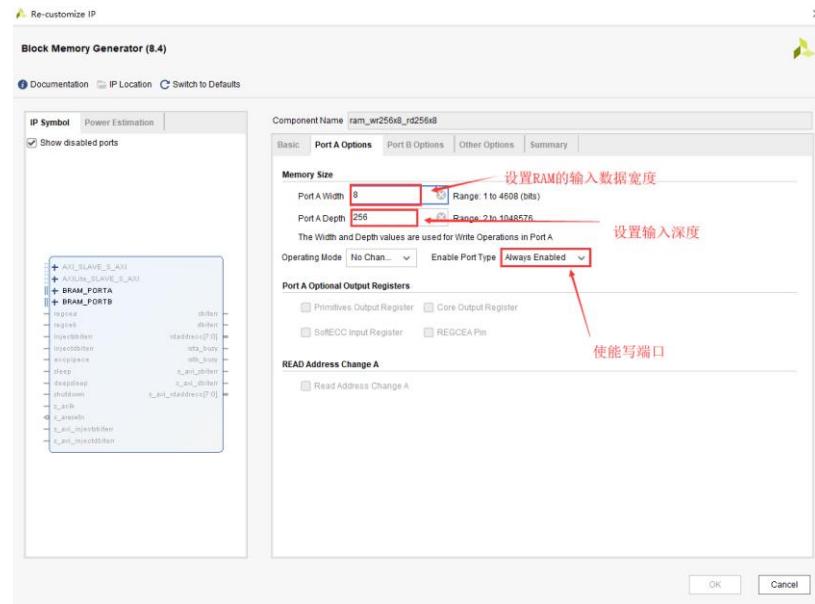


图 11- 3

### 3) 接下来是输出端口配置

输出端口依然可以对输出位宽进行选择，这里保持为 8bit，输出端口的深度是根据输入的深度，输入的位宽核输出的位宽确定的。

输出端口 B 不添加寄存器。使用寄存器会使输出数据多一拍延时，这个是否选择，要根据具体的设计来定，在这里不选择。

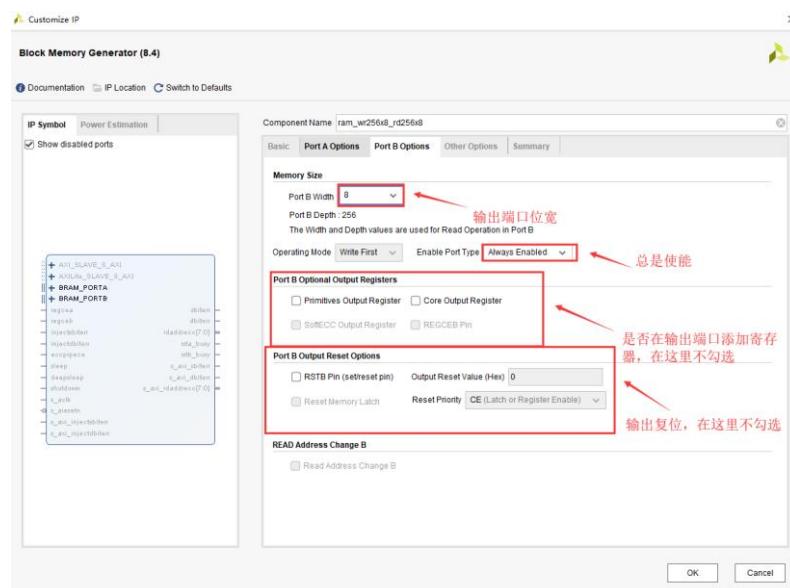


图 11- 4

4) 在其他选项中可以选择导入初始化 RAM 的文件，在本次实验中不导入。

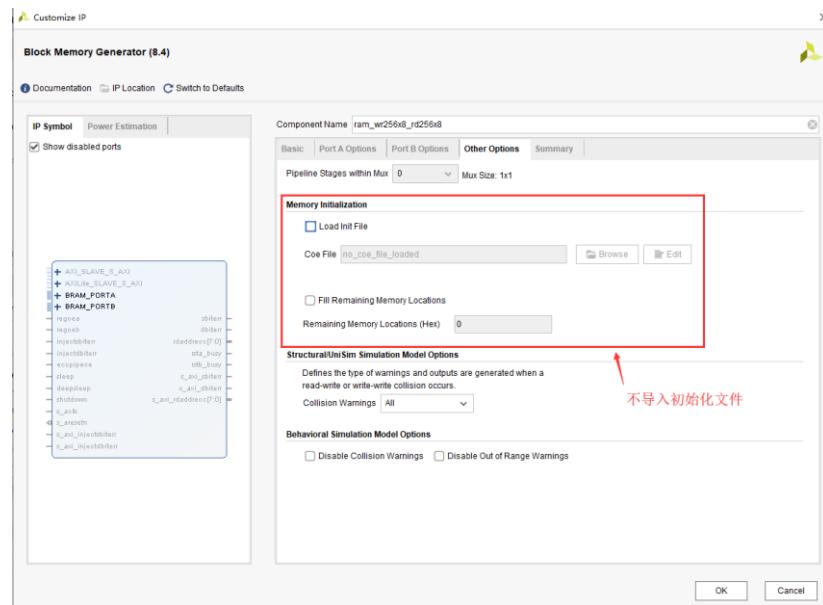


图 11- 5

5) 在最后的 Summary 中，可以查看关于配置的详细信息

注意到本次配置，从 B 端口读出的数据，具有 1 个 clock 的 Latency，即一个时钟周期的延时。

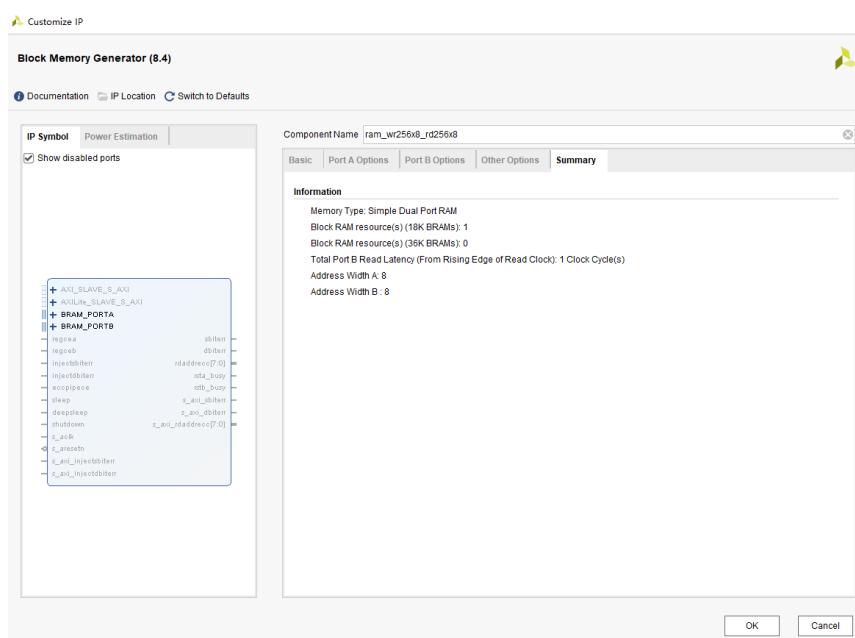


图 11- 6

6) 点击 OK 即完成了 RAM IP 的配置。

## 9.4 程序设计以及 IP 核的调用

本次实验需要使用生成的 IP，例化两个 RAM 以完成乒乓操作。两个 RAM 的交替读写操作我们称之为乒乓操作。为了使该操作更加明显，我们绘制出如下的时序图，来帮助我们理解本次实验的操作时序。该绘图提供在工程目录下的 img 目录下，若想看清晰的图形，可使用 Visio 打开。

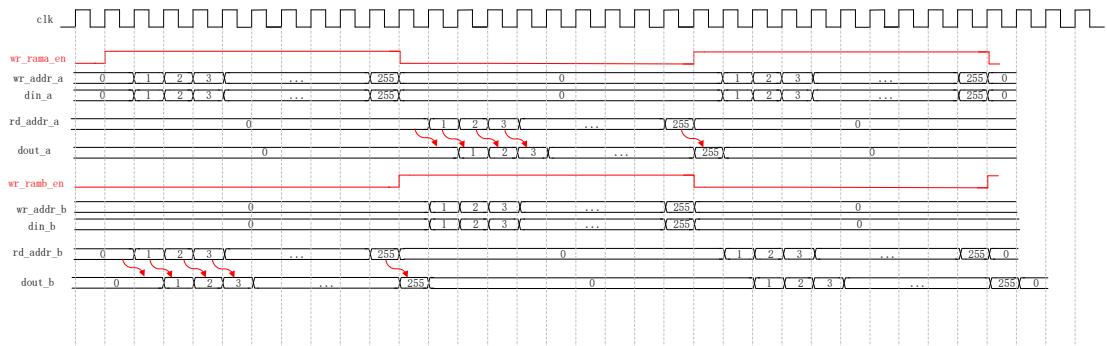


图 11-7

在本次实验中，使用两个指示信号 `wr_rama_en` 和 `wr_ramb_en` 来指示当前对哪一个 RAM 进行读写操作。

当 `wr_rama_en` 为高、`wr_ramb_en` 为低时，对 `ram_a` 进行写操作，从 `ram_a` 的地址 0 到地址 255，依次写入 0 到 255 的数据，与此同时，对 `ram_b` 进行读操作，将 `ram_b` 中的数据，依次从地址 0 到地址 255 读出，根据 IP 核的设置，读出的数据，相较于读地址，有一拍的延时。

当 `wr_rama_en` 为低、`wr_ramb_en` 为高的时候，对 `ram_a` 进行读操作，依次从 `ram_a` 的地址 0 到地址 255 中读出数据，可以看到读出数据，相较于读地址，也有一拍的延时。于此同时，对 `ram_b` 进行写操作，依次将 0 到 255 写入 `ram_b` 的地址 0 到地址 255。

通过对 ram\_a 和 ram\_b 的不同时读写，完成了乒乓操作。

本次实验的程序设计主要就是实现上面给出的那张波形图的时序，在 FPGA 开发中，通过画波形图来设计时序是经常使用的一种方法，使用这种方法，能够提升开发的效率，建议多采用这种方法。

### 1) 创建一个 ram\_pp.v 文件

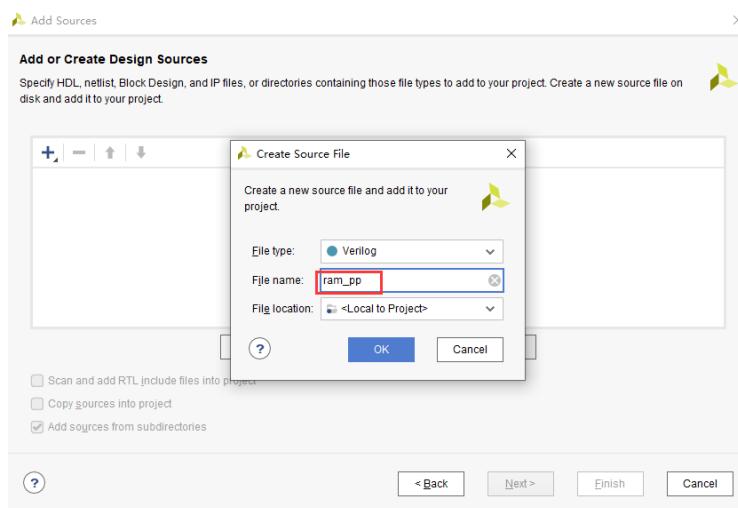


图 11-8

### 2) 编写实验代码

具体的代码提供在 demo 目录下的文件夹下。就代码里的关键部分做几点说明：

44 行到 60 行对 wr\_rama\_en 进行控制

```

44 //-----wr_rama_en-----
45 always @(posedge clk or negedge rst_n)begin
46   if(rst_n==1'b0)begin
47     wr_rama_en <= 1'b0;
48   end
49   //当 ram_b 写到最后一个地址时，wr_rama_en 拉高
50   else if((wr_addr_b==MAX) && (wr_rama_en==1'b0))begin
51     wr_rama_en <= 1'b1;
52   end
53   //当 ram_a 写道最后一个地址的时候，wr_rama_en 拉低

```

```

54     else if((wr_addr_a==MAX) && (wr_rama_en==1'b1))begin
55         wr_rama_en <= 1'b0;
56     end
57     else begin
58         wr_rama_en <= wr_rama_en;
59     end

```

62 行到 65 行采用组合逻辑，完成对 wr\_ramb\_en 的控制

```

62 //-----wr_ramb_en-----
63 always @(*)begin
64     wr_ramb_en = ~wr_rama_en;
65 end

```

68 行到 86 行产生写 ram\_a 的地址，和写入 ram\_a 的数据；

```

68 //-----wr_addr_a-----
69 always @(posedge clk or negedge rst_n)begin
70     if(rst_n==1'b0)begin
71         wr_addr_a <= 'd0;
72     end
73     //当 wr_rama_en 为高时， ram_a 写地址增加
74     else if(wr_rama_en == 1'b1)begin
75         if(wr_addr_a == MAX)
76             wr_addr_a <= 'd0;
77         else
78             wr_addr_a <= wr_addr_a + 1'b1;
79     end
80     else begin
81         wr_addr_a <= 'd0;
82     end
83 end
84
85 //-----din_a-----
86 assign din_a = wr_addr_a;

```

89 到 104 行产生读 ram\_a 的地址

```

89 //-----rd_addr_a-----
90 always @(posedge clk or negedge rst_n)begin
91     if(rst_n==1'b0)begin
92         rd_addr_a <= 'd0;
93     end
94     //当 wr_rama_en 为低时, ram_a 读地址增加
95     else if(wr_rama_en == 1'b0)begin
96         if(rd_addr_a == MAX)
97             rd_addr_a <= 'd0;
98         else
99             rd_addr_a <= rd_addr_a + 1'b1;
100    end
101   else begin
102       rd_addr_a <= 'd0;
103   end
104 end

```

对于 ram\_b 采用大致相同的操作，这里不再赘述。

144 行到 162 行例化了两个 ram 的 IP，ram\_a 和 ram\_b, 例化 IP 时可以使用 Vivado 提供的例化模板，注意选择模板时，要选择后缀名为.veo 的文件。

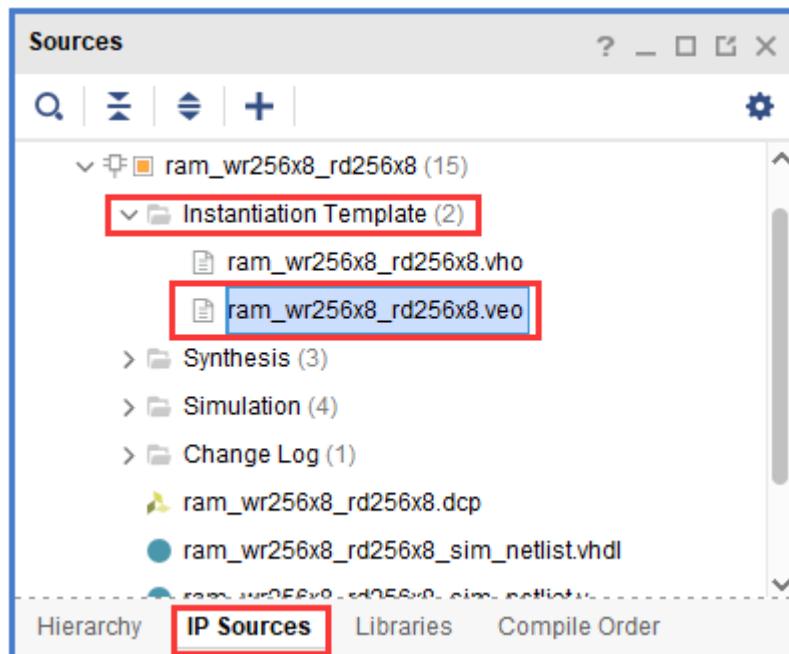


图 11-9

**144** ram\_wr256x8\_rd256x8 ram\_a (

```

145 .clk_a(clk), // input wire clk_a
146 .wea(wr_rama_en), // input wire [0:0] wea
147 .addr_a(wr_addr_a), // input wire [7:0] addr_a
148 .dina(din_a), // input wire [7:0] dina
149 .clk_b(clk), // input wire clk_b
150 .addr_b(rd_addr_a), // input wire [7:0] addr_b
151 .doutb(dout_a) // output wire [7:0] dout_b
152 );
153
154 ram_wr256x8_rd256x8 ram_b(
155 .clk_a(clk), // input wire clk_a
156 .wea(wr_ramb_en), // input wire [0:0] wea
157 .addr_a(wr_addr_b), // input wire [7:0] addr_a
158 .dina(din_b), // input wire [7:0] dina
159 .clk_b(clk), // input wire clk_b
160 .addr_b(rd_addr_b), // input wire [7:0] addr_b
161 .doutb(dout_b) // output wire [7:0] dout_b
162 );

```

164 到 182 行为了输出从 ram 中读出的数据，其中 wr\_rama\_dd 是 wr\_rama\_en 的延时一拍，目的是为了与从 ram 中读出的数据在时序上对齐。读出数据采用组合逻辑赋值给 rd\_data；当 wr\_rama\_dd 为一的时候，读出的是 ram\_b 中的数据，当 wr\_rama\_dd 为 0 时读出的是 ram\_a 中的数据。

```

164 //-----wr_rama_dd-----
165 always @ (posedge clk or negedge rst_n) begin
166   if(rst_n == 1'b0) begin
167     wr_rama_dd <= 1'b0;
168   end
169   else begin
170     wr_rama_dd <= wr_rama_en;
171   end
172 end
173
174 //-----rd_data-----
175 always @ (*) begin
176   // 读出的数据为 ram_a 中的数据
177   if(wr_rama_dd == 1'b1)
178     rd_data = dout_b;
179   // 读出的数据为 ram_b 中的数据

```

```

180      else
181          rd_data = dout_a;
182 end

```

## 9.5 仿真验证

接下来采用仿真的方式，验证设计的正确性。在本次实验中采用 modelsim 对设计进行仿真。

### 1) 添加仿真文件

Add Source => Add or Create Simulation Sources, 添加名为 tb\_ram\_pp 的仿真文件

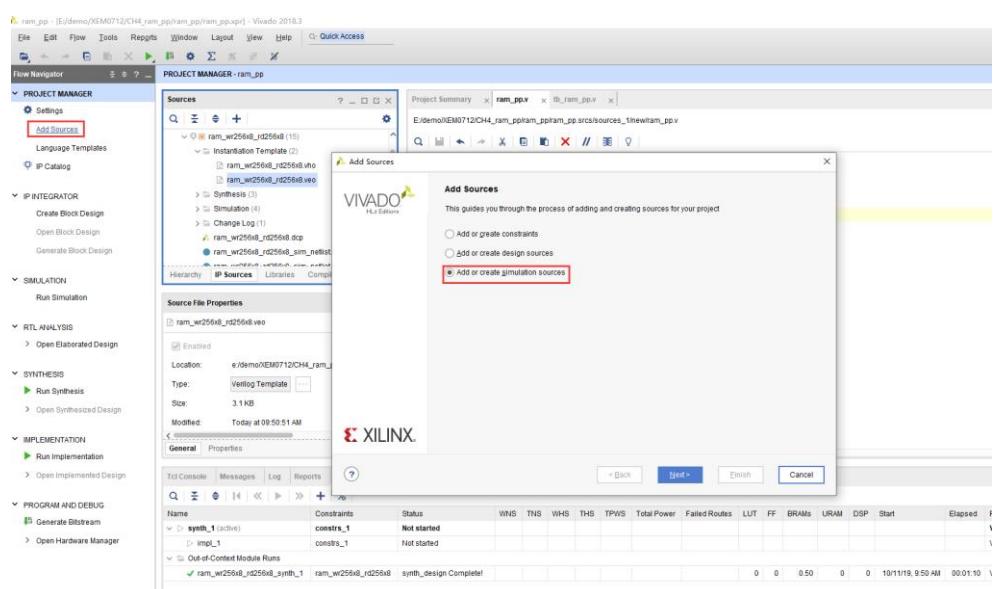


图 11-10

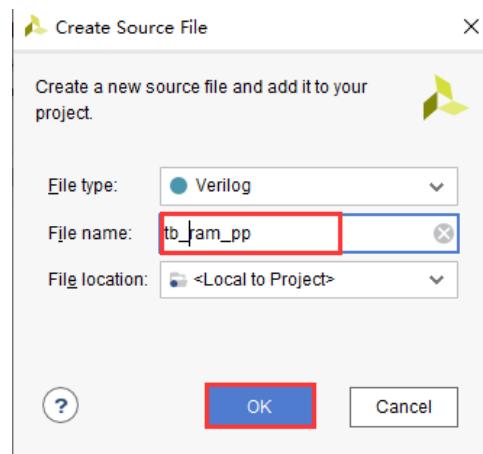


图 11-11

## 2) 编写仿真文件

仿真文件可以在工程目录下的 ram\_pp.sim 文件夹中找到，对仿真文件内容做几点说明：

6 到 8 行是激励和输出信号定义

10 到 16 行是待仿真模块的例化

```
'timescale 1ns/1ps

module tb_ram_pp(); /* this is automatically generated */

// (*NOTE*) replace reset, clock, others

parameter MAX = 256 - 1;

reg      clk ;
reg      rst_n ;
wire [7:0] rd_data ;

ram_pp #(
    .MAX(MAX)
) inst_ram_pp (
    .clk (clk),
    .rst_n (rst_n),
    .rd_data (rd_data)
```

```

);

// clock
initial begin
    clk = 1;
    forever #(10) clk = ~clk;
end

// reset
initial begin
    rst_n <= 0;
#200;
    rst_n <= 1;
end

endmodule

```

- 3) 保存仿真文件，准备开始仿真，按照上一章所演示的将 modelsim 和 Vivado 关联的方法，关联 modelsim 和 Vivado

点击 Tools => Settings

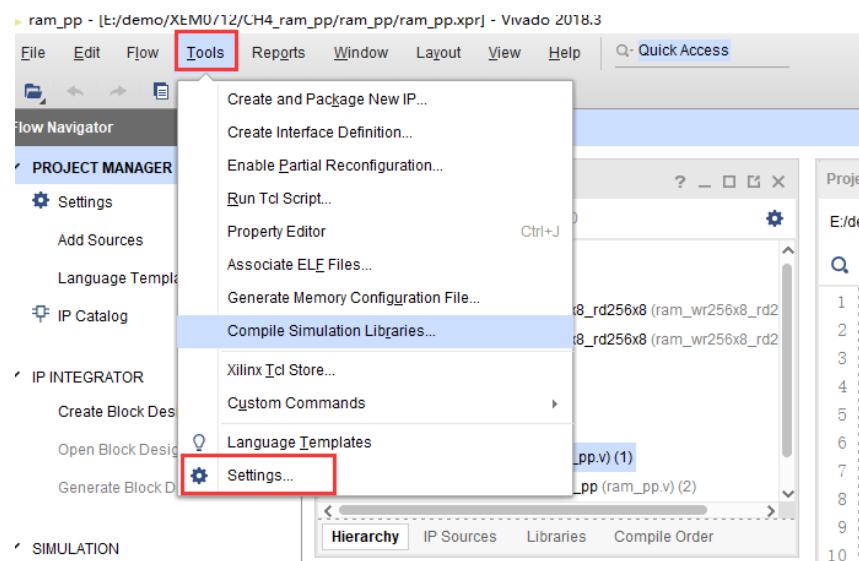


图 11-12

在 Project setting 中选择 Simulation 选项，选择 Modelsim simulator，在 Tools

setting 中选择 Modelsim 安装的路径和库文件所在的位置。

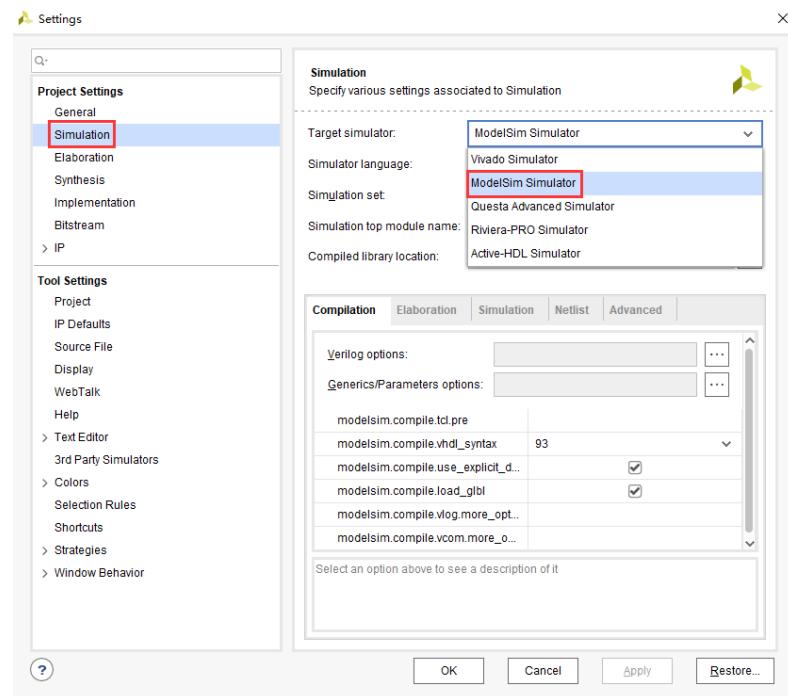


图 11- 13

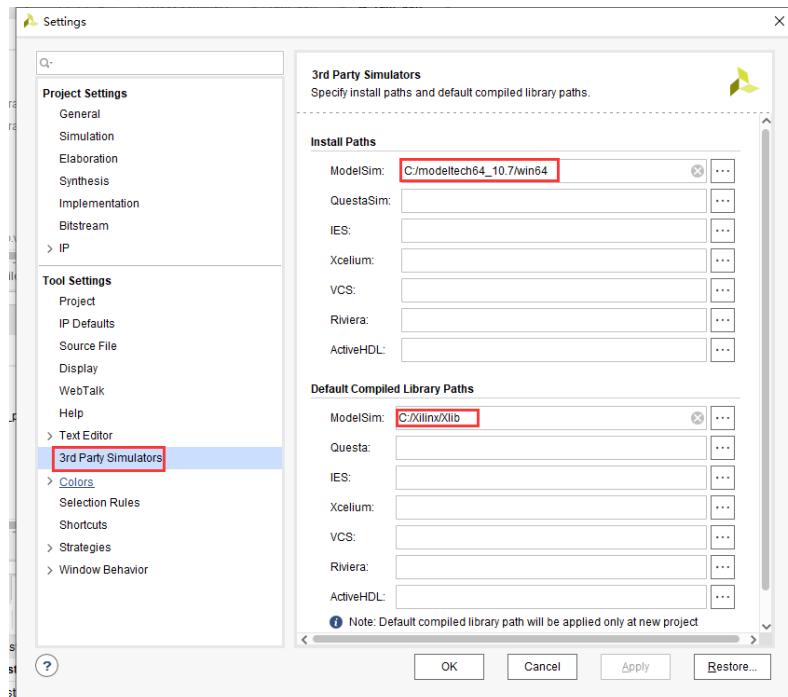


图 11- 14

4) 打开仿真界面

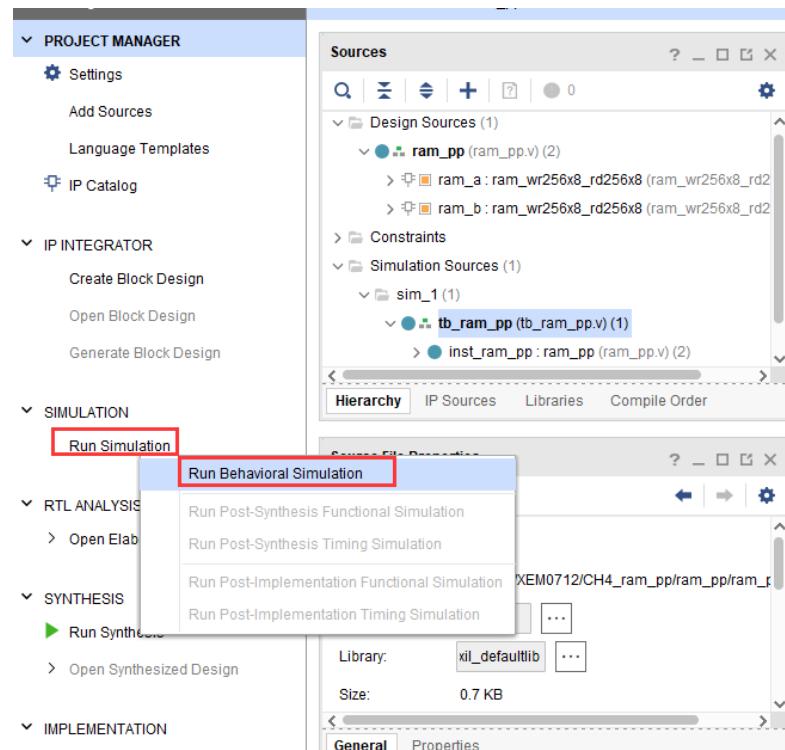


图 11- 15

## 5) 仿真观察波形

将 inst\_ram\_pp 添加到波形窗口

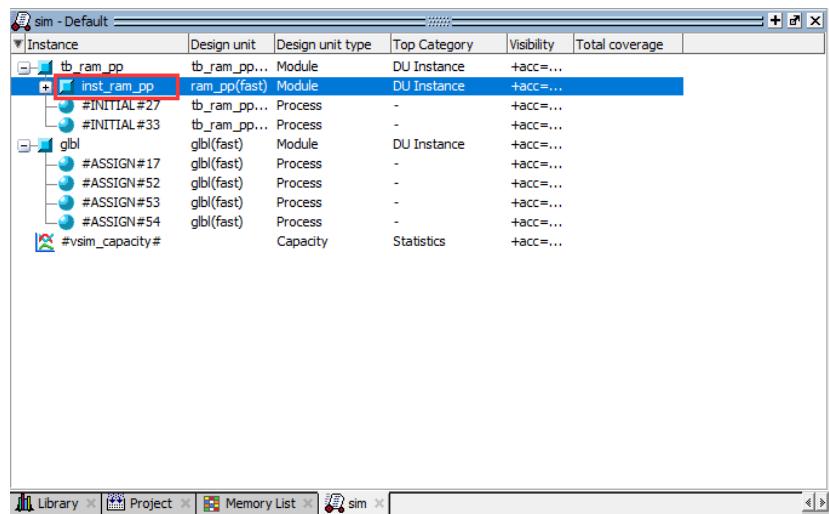


图 11- 16

设置仿真时间为 10us => restart => run 可以观察波形

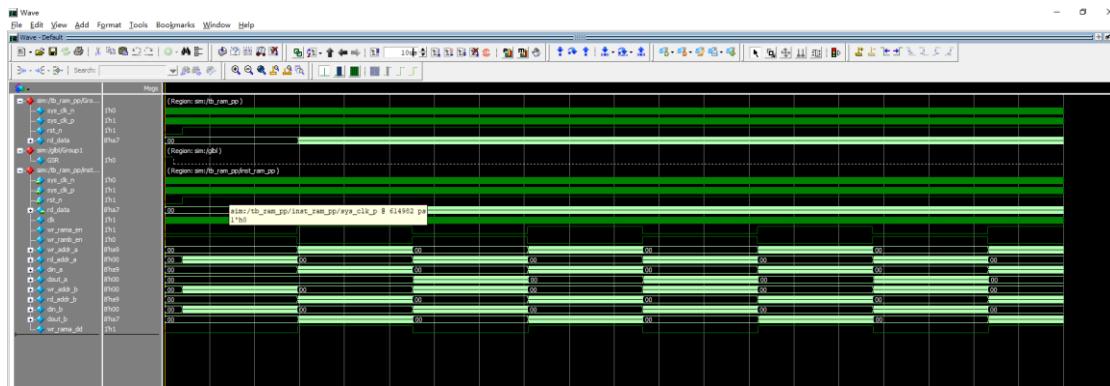


图 11- 17

观察波形可以看到两个 ram 进行了乒乓操作。

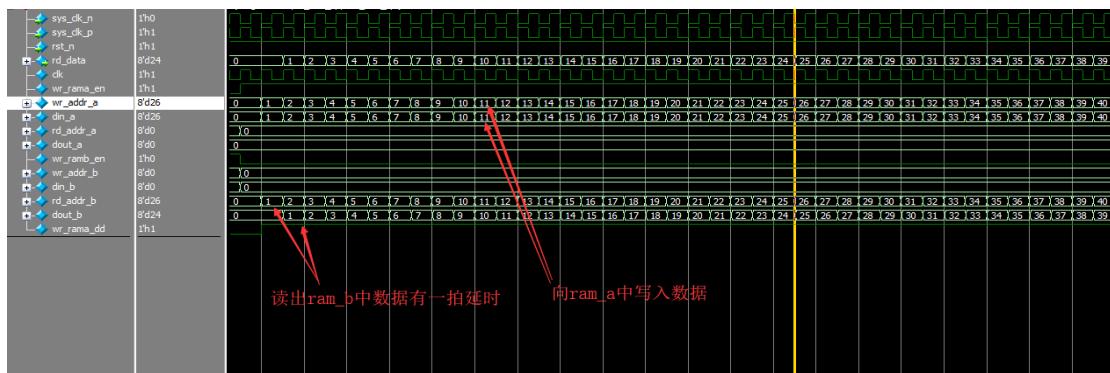


图 11- 18

通过上面这张截图可以看到，当 `wr_rama_en` 为高的时候，向 ram 中从地址 0 到地址 255，写入递增数据。于此同时，从 ram\_b 中读出地址 0 到地址 255 的数据。

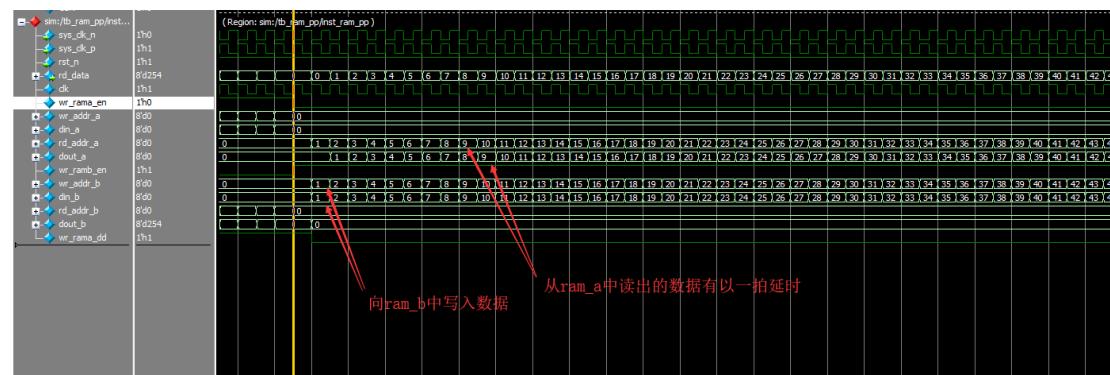


图 11- 19

当 `wr_rama_en` 为低时，从 `ram_a` 中读出数据，而此时 `ram_b` 正在向地址 0 到地址 255 写入递增数据。

除此之外，我们还能看到，输出信号 `rd_data` 输出的时连续数据，而这些数据，与我们写入 `ram` 中的数据想吻合。

到这里我们完成了对 RAM 乒乓操作的仿真。

## 9.6 实验总结

本次实验，介绍了 Vivado 总 RAM IP 核的配置，并以一个最为简单的，也是最常用的 IP 核配置，完成了一个 RAM 的乒乓操作的仿真。进一步加深了对于仿真对于设计的重要性。除此之外，还提供了一个在设计之前绘制的时序图，通过时序图来进行设计，这在实际开发中是经常用到的。乒乓操作，在处理高速数据流的时候是一种经常使用的手段，通过本章掌握对 `ram` 的乒乓操作，将来在使用其他类型的片上存储资源时，也能对其完成乒乓操作。

## 第10章 片上资源之 FIFO

### 10.1 实验简介

本次实验，将介绍 Vivado 中 FIFO 的使用，FIFO IP 核的配置以及对 FIFO IP 的仿真。

### 10.2 实验原理

#### 1) FIFO 的概念

1、 FIFO(Frist Input Frist Output)，即先入先出，也是一种存储器，一般做数据缓冲。

2、 FIFO 和 RAM 的共同点在于都能存储数据、都有控制写和读的信号；不同点在于 FIFO 没有地址，所以不能任意指定读取某一个数据，数据只能按照数据输入的顺序输出，即先入先出，并且读写可以同时进行。

3、如果数据把 FIFO 的深度写满了，数据将不能再进去，也不会覆盖原有的数据；读 FIFO 的数据也只能读一遍，读完一遍 FIFO 就空了，需要再往 FIFO 写数据才能读出新数据，否则读出的数据一直是最后一次读 FIFO 时的数据。

下图所示的是一个简单的异步 fifo 的示意图。在写端口有写时钟、写使能和待写入的数据。在读端口有读时钟，读使能和读出的数据。此外还有 fifo 的空满信号，当 fifo 中写满数据的时候，full 信号拉高，当 fifo 中数据全部读出后 empty 信号拉高。

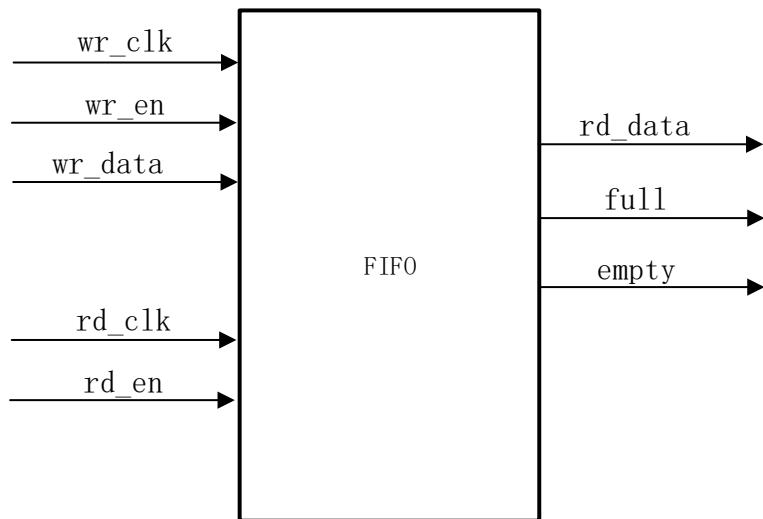


图 12-1

#### 4、FIFO 处理的关键时序

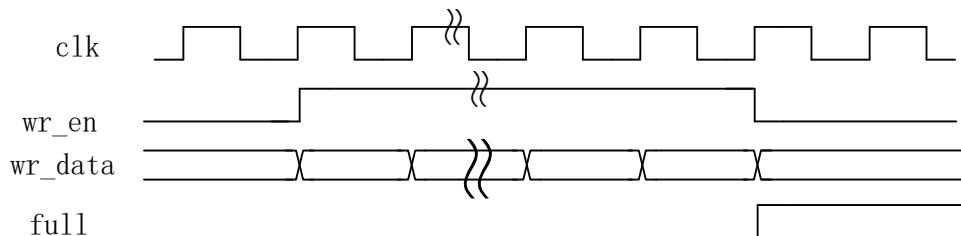


图 12-2 FIFO 写时序

FIFO 写时序如上图所示，写端口数据在写使能拉高的情况下，每个时钟周期将待写入的数据，写进 FIFO 中，当 FIFO 写满时，full 信号拉高。

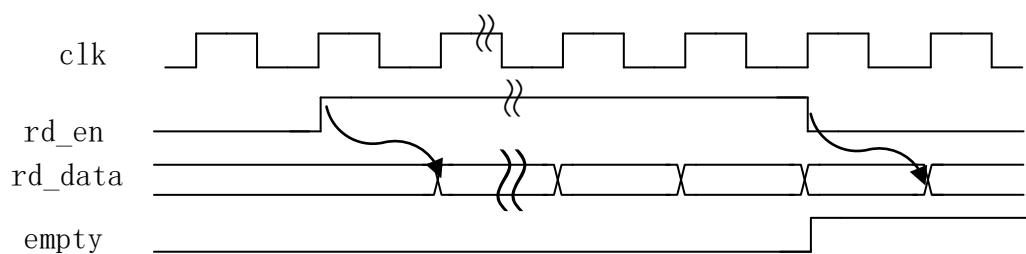


图 12-3 standard 读模式

Standard 的读时序如上图所示，在 rd\_en 拉高后，开始从 fifo 中读出数据，但是此时读出的数据具有一拍延时，Standard 模式用于输出端时序特性要求比较

高的场合，也可以设置多拍延迟这样经过寄存器大拍提高输出寄存器间的时序特性。

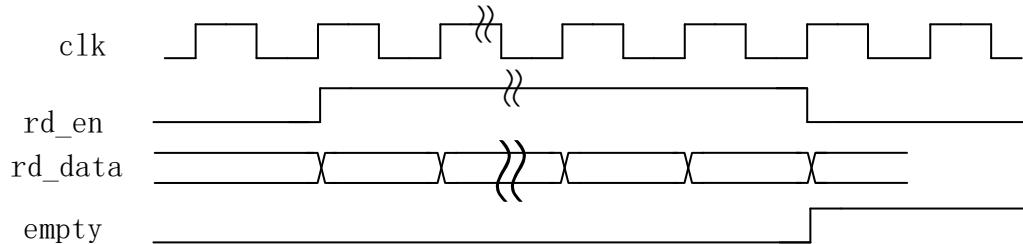


图 12- 4 First-word Fall-Through 模式

First-Word Fall-Through 模式广泛用于数据缓冲应用，特点相对读使能 0 延时输出读出数据，其原理是第一个数据永远不需要读使能就自动呈现在读端口上。

### 10.3 创建工程

双击桌面 Vivado2018.3 图标按如下表格操作

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 fifo_test, 工程路径 E:\demo\XEM0712\CH5_fifo 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件，购买 Z7-Lite7020 用户选中 xc7z020clg400-2，购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后，点击确认

### 10.4 配置 FIFO IP 核

#### 1) 添加 IP 核

点击左侧 IP Catalog，在搜索栏搜索 fifo，找到 Memories & storage Elements

下面的 FIFO Generate 并双击弹出配置界面

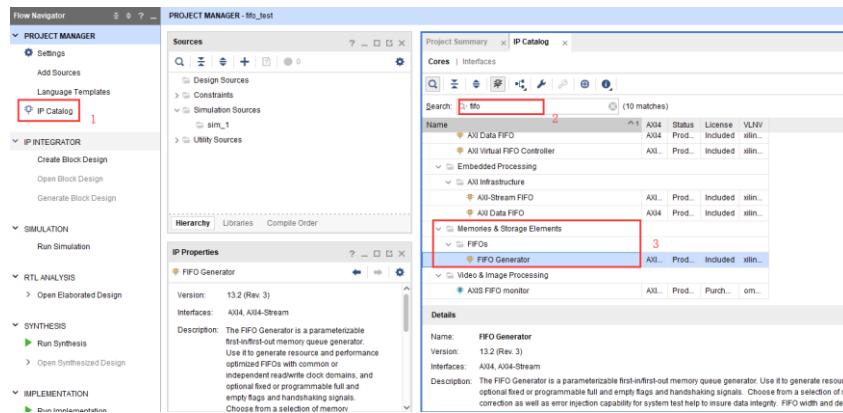


图 12-5

## 2) IP 核基础设置

在本页可以先选择修改 fifo 的名称。Fifo 接口类型选择 Native 类型，Fifo implementation 选择 Independent clock Block RAM（独立时钟块 RAM）

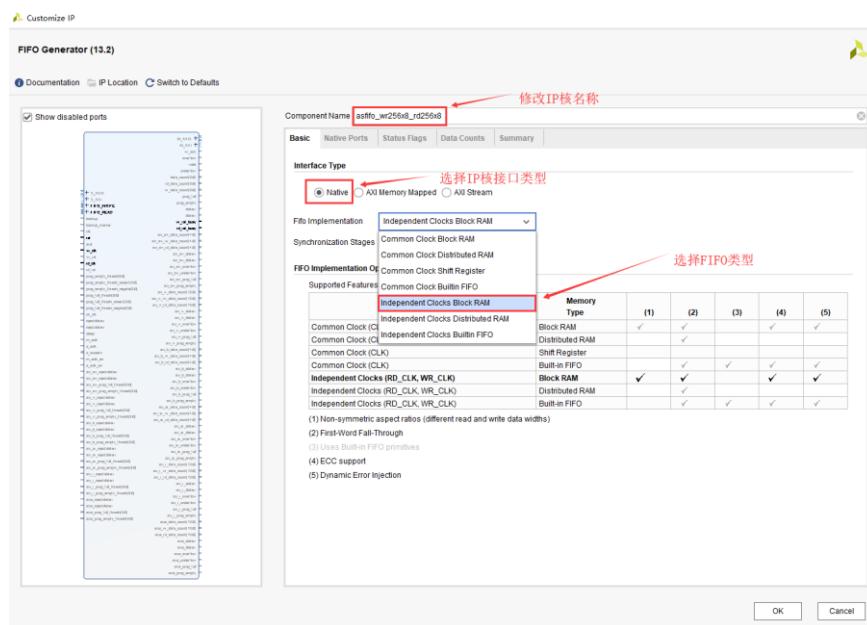


图 12-6

### 3) FIFO 输入输出设置

Read Mode 选择 First Word Fall Through 模式;

输入数据位宽设置为 8;

Fifo 深度设置为 256，其中 FIFO 的深度必须是 2 的幂次方

读 FIFO 位宽设置为 8;

不勾选 Reset Pin;

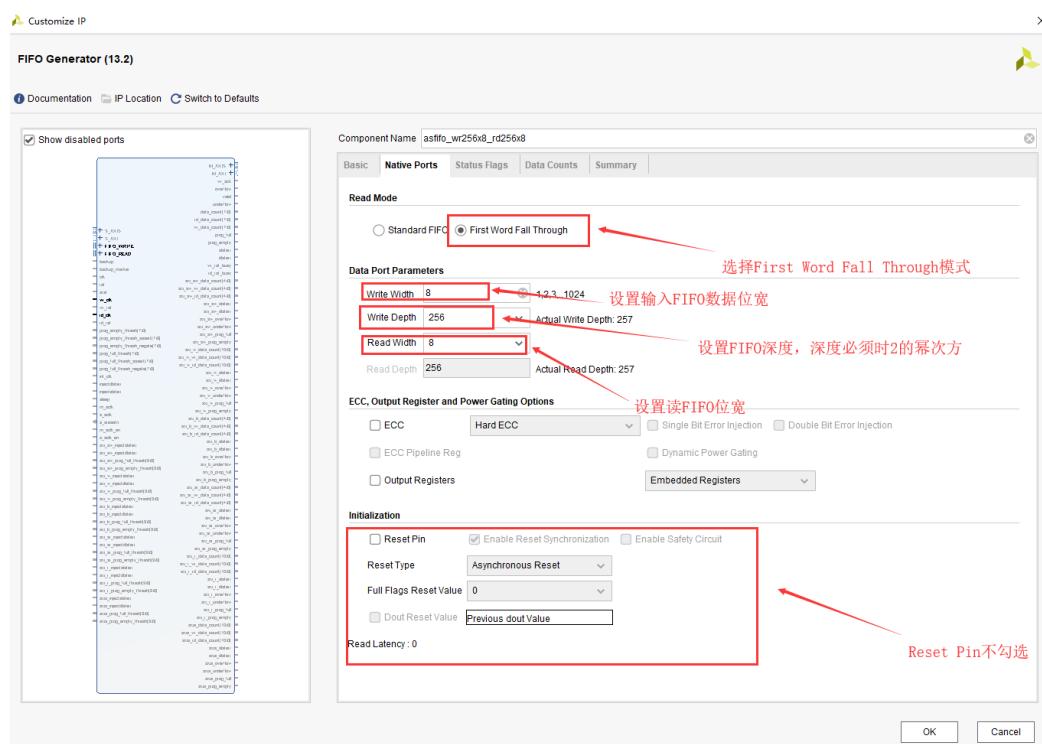


图 12- 7

### 4) Status Flag 页保持默认; Data count 页保持默认。

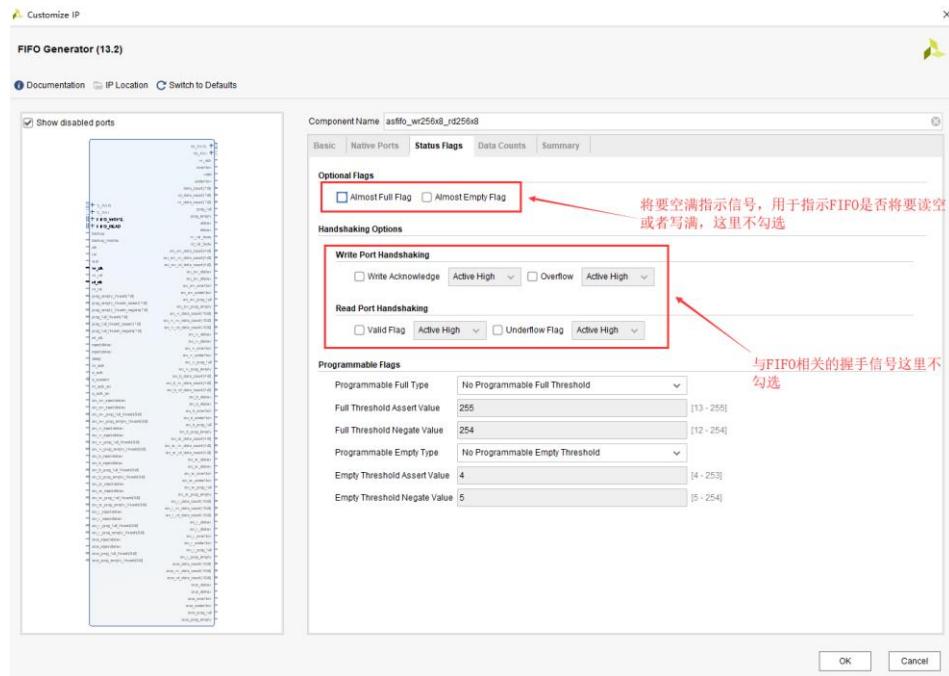


图 12-8

5) 在 Summary 页面可以看到我们将要生成的 FIFO 的信息，点击 OK 生成 FIFO IP 核；

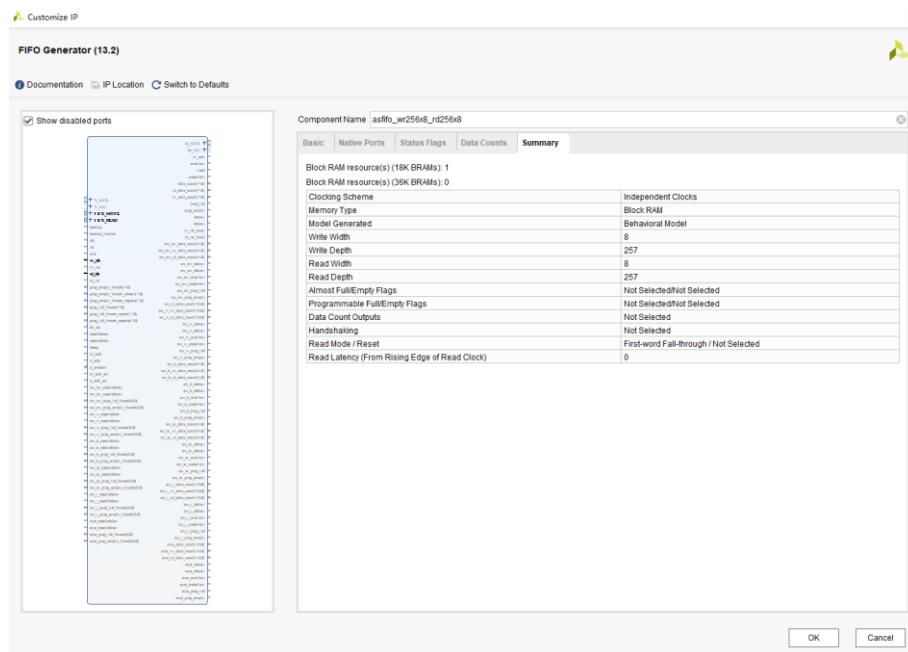


图 12-9

## 10.5 程序设计

### 1) 设计时序

首先，我们先设计一个时序图，并根据这个时序图，写出对应的代码。

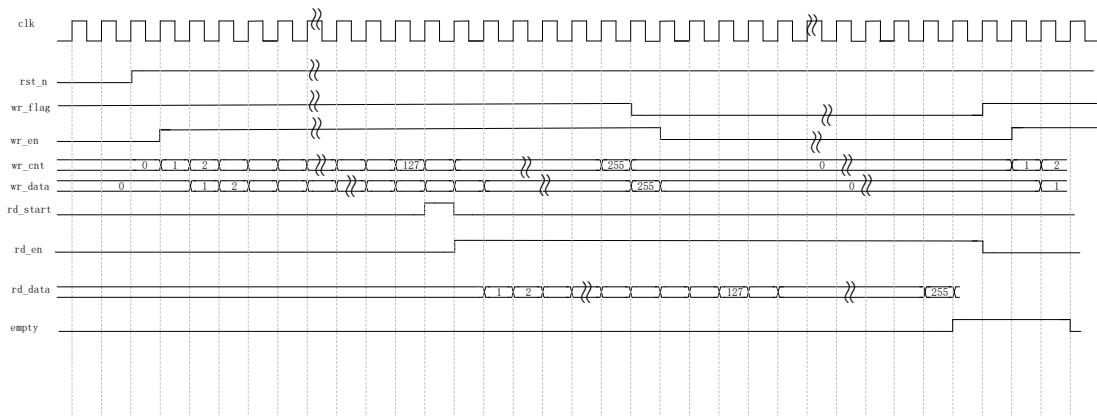


图 12-10

从这个时序图中可以看到，`wr_flag` 用于指示当前是否为写 FIFO 的状态，`wr_en` 为写 FIFO 使能，当其为高的时候，向 FIFO 中写入数据；`wr_data` 为写入 FIFO 的数据；`rd_start` 信号为读 FIFO 其实信号，当该信号拉高时开始准备读 FIFO；`rd_en` 信号为读 FIFO 使能信号，当该信号拉高时，FIFO 将数据送出到输出数据总线上；`empty` 信号在 FIFO 中所有数据都从 FIFO 中读出来时拉高。

根据这张时序图可以很轻松的设计出代码。

### 2) 代码设计

根据上面所提供的时序设计图，开始设计本次实验的代码，由于本次实验只做仿真来验证 FIFO 的功能，所以代码比较简单。

本次实验的工程代码可以在对应的工程中找到源码。

```
module fifo_test(
    input    wire        clk      //系统输入差分时钟
    input    wire        rst_n
);
```

```
parameter MAX      = 256 - 1;
parameter RD_START = 128 - 1;
reg           wr_en       ;//写FIFO指示信号
reg           wr_flag     ;//写数据指示信号
reg [8:0]     wr_cnt     ;//写FIFO计数器
reg [7:0]     wr_data    ;//写FIFO数据
wire          full,empty ;//空满信号
reg           rd_en       ;//读FIFO使能
reg           rd_start    ;//读FIFO开始信号
wire          [7:0]      rd_data    ;//读出的数据

//-----wr_flag-----
//写FIFO指示信号
always @ (posedge clk or negedge rst_n)begin
  if(rst_n==1'b0)begin
    wr_flag <= 1'b1;
  end
  else if(wr_cnt==MAX && wr_flag==1'b1)begin
    wr_flag <= 1'b0;
  end
  else if(empty==1'b1) begin
    wr_flag <= 1'b1;
  end
  else begin
    wr_flag <= wr_flag;
  end
end

//-----wr_en-----
always @ (posedge clk or negedge rst_n)begin
  if(rst_n==1'b0)begin
    wr_en <= 1'b0;//复位时为低
  end
  else begin
    wr_en <= wr_flag;
  end
end

//-----wr_cnt-----
```

```
always  @(posedge clk or negedge rst_n)begin
    if(rst_n==1'b0)begin
        wr_cnt <= 'd0;
    end
    else if(wr_flag==1'b1)begin//计数当前写了多少个数据
        if(wr_cnt == MAX)
            wr_cnt <= 'd0;
        else
            wr_cnt <= wr_cnt + 1'b1;
    end
    else begin
        wr_cnt <= 'd0;
    end
end

//-----wr_data-----
always  @(posedge clk or negedge rst_n)begin
    if(rst_n==1'b0)begin
        wr_data <= 'd0;
    end
    else begin
        wr_data <= wr_cnt;//写入FIFO的数据
    end
end

//-----rd_start-----
//开始进行读FIFO信号
always  @(posedge clk or negedge rst_n)begin
    if(rst_n==1'b0)begin
        rd_start <= 1'b0;
    end
    else if(wr_cnt == RD_START)begin//当写入128个数据后，开始从FIFO中读出数据
        rd_start <= 1'b1;
    end
    else begin
        rd_start <= 1'b0;
    end
end

//-----rd_en-----
always  @(posedge clk or negedge rst_n)begin
```

```
if(rst_n==1'b0)begin
    rd_en <= 1'b0;
end
else if(rd_start==1'b1)begin//读开始信号拉高后，读使能信号拉高
    rd_en <= 1'b1;
end
else if(empty == 1'b1)begin//FIFO读空后停止读
    rd_en <= 1'b0;
end
else begin
    rd_en <= rd_en;
end
end

asfifo_wr256x8_rd256x8 fifo_inst (
    .wr_clk(clk), // input wr_clk
    .rd_clk(clk), // input rd_clk
    .din(wr_data), // input [7 : 0] din
    .wr_en(wr_en), // input wr_en
    .rd_en(rd_en), // input rd_en
    .dout(rd_data), // output [7 : 0] dout
    .full(full), // output full
    .empty(empty) // output empty
);
endmodule
```

## 10.6 仿真验证

接下来对 FIFO 进行仿真验证，本次实验依旧采用 modelsim 对 FIFO 进行仿真。

在开始仿真前，需先创建并完成 testbench 仿真文件。

### 1) 设计仿真文件

从本次实验的设计文件中可以看出，本次实验，只有三个输入，没有输出。所以在设计仿真文件时，只需给出相应的简单几个激励，通过观察模块内部的信

号来判断 FIFO 是否如预期的那样工作。

仿真文件如下：

```
'timescale 1ns/1ps

module tb_fifo_test(); /* this is automatically generated */

//(*NOTE*) replace reset, clock, others

parameter MAX      = 256 - 1;
parameter RD_START = 128 - 1;

reg   clk      ;
reg   rst_n    ;

fifo_test #(
    .MAX(MAX),
    .RD_START(RD_START)
) inst_fifo_test (
    .clk (clk),
    .rst_n (rst_n)
);

initial begin
    clk = 0;
    forever #(10) clk = ~clk;
end

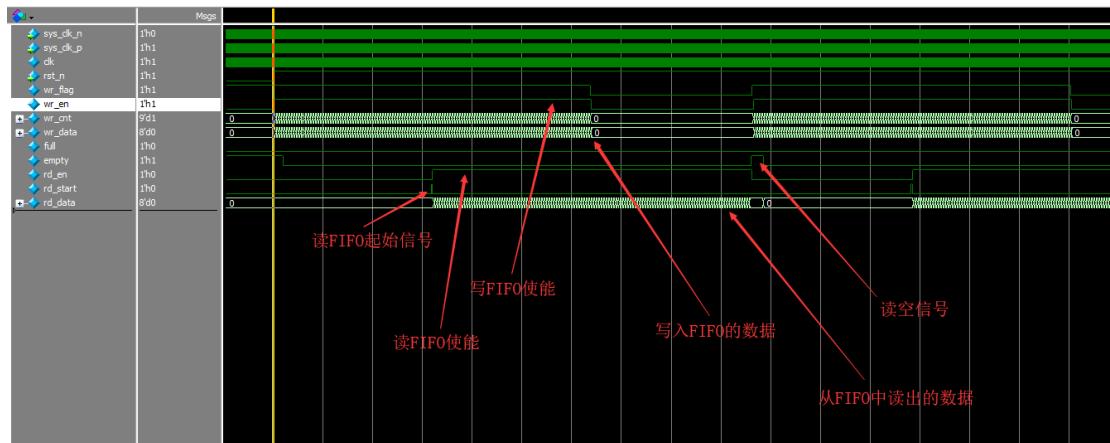
// reset
initial begin
    rst_n <= 0;
    #200
    rst_n <= 1;
end
endmodule
```

2) Vivado 关联 modelsim 进行仿真

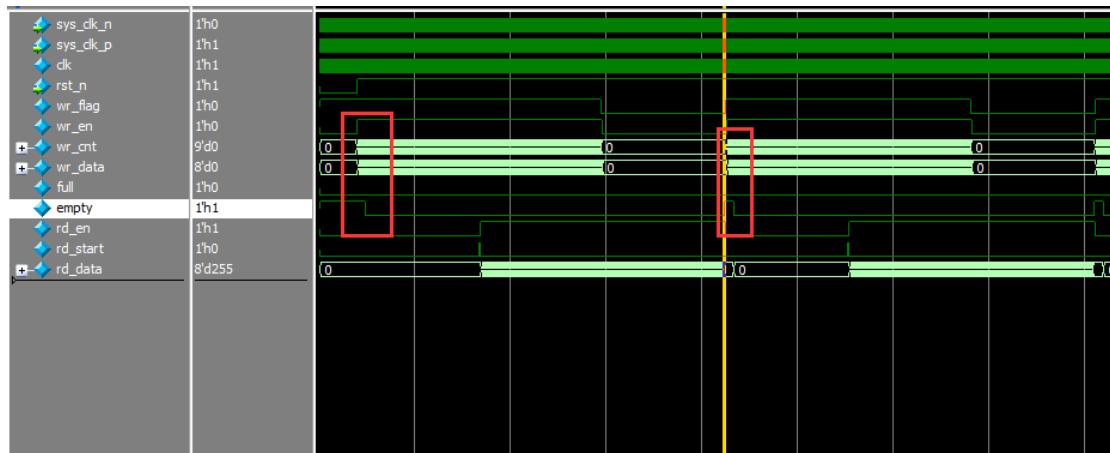
使用 Vivado 关联 modelsim 进行仿真的方法和上面几次实验的方法相同，在这里不进行赘述。

### 3) 观察结果

将带观测信号添加到波形窗口中进行观察



通过观察波形，我们可以发现通过仿真得到的波形与我们预先设计的时序图吻合。值得注意的是 FIFO 的空满信号。空满信号不是在 FIFO 中写满或读空时立即拉高或拉低。比如在本次实验中，当 FIFO 中已经写入了一定数据的时候，empty 信号依旧为高。



而这与 FIFO 的结构有关，FIFO 本质上也是一个特殊的 RAM，感兴趣的可以自己去找资料了解了解 FIFO 的结构。

## 10.7 实验总结

通过本次实验，展示了如何配置一个最常用的简单的异步 FIFO。并对这个简单的 FIFO 进行了仿真。FIFO 在数据缓冲，时钟域的切换中起着十分重要的作用。本章只是一个简单的示例，更多的使用方法和形式是要根据具体的设计来决定的。

# 第11章 LCD 显示实验

## 11.1 VGA 原理和时序

计算机显示区的显示有许多标准，常见的有 VGA、SVGA 等。在这里我们用 VGA 接口来控制显示器，VGA 即 Video Graphics Array 的缩写，也就是视频图形阵列。作为一种标准的显示接口得到广泛的应用。

常见的彩色显示器一般由 CRT（阴极射线管）构成，色彩是由 R、G、B（红、黄、蓝）三基色组成。显示是用逐行扫描的方式解决，阴极射线枪发出电子束打在涂有荧光粉的荧光屏上，产生 RGB 三基色，合成一个彩色像素。扫描从屏幕的左上方开始，从左到右，从上到下，进行扫描，每扫完一行，电子束回到屏幕的左边下一行的起始位置，在这其间 CRT 对电子束进行消隐。每行结束时，用行同步信号进行同步；扫描完所有行，用场同步信号进行同步，并使扫描回到屏幕左上方，同时进行场消隐，预备下一场的扫描。

对于普通的 VGA 显示器，共有 5 个信号：R、G、B 三基色；HS（行同步信号）；VS（场同步信号）。对于时序驱动，VGA 显示器要严格遵循“VGA”工业标准，我们选取分辨率为 1920x1080@60Hz 模式。通常我们用的显示器都满足工业标准，因此我们设计 VGA 控制器时要参考显示器的技术规格。下图是 VGA 行、场扫描的时序图：

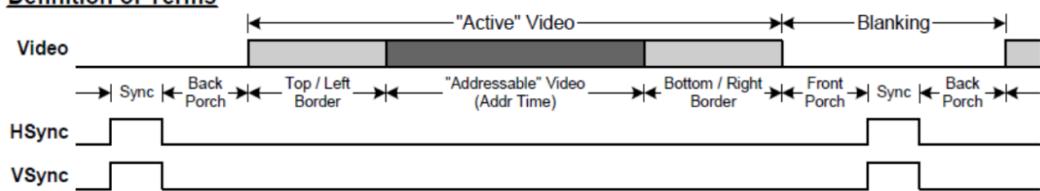
Definition of Terms

图 13- 1

完成一行扫描的时间称为水平扫描时间，其倒数称为行频率；完成一帧（整屏）扫描的时间称为垂直扫描时间，其倒数称为场频率，即刷新一屏的频率，常见的有 60Hz, 75Hz 等等。标准的显示的场频 60Hz。

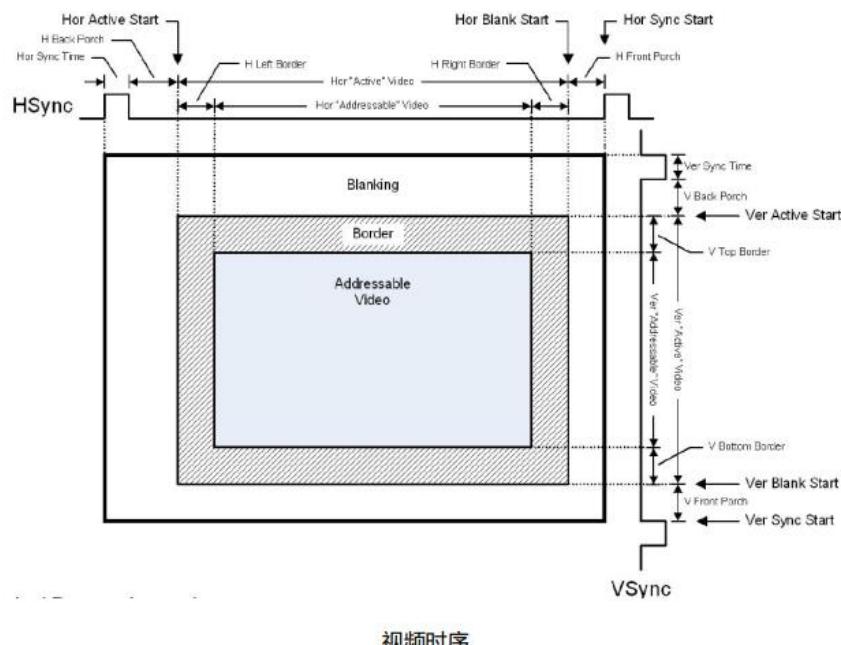


图 13- 2

时钟频率：以 1024x768@59.94Hz(60Hz)为例，每场对应 806 个行周期,其中 768 为显示行。每显示行包括 1344 点时钟,其中 1024 点为有效显示区。由此可知：需要点时钟频率：  $806 \times 1344 \times 60$  约 65MHz

VGA 扫描，基本元素是行扫描，多行组成一帧，下图显示一行的时序，其中“Active” Video 是一行视频的有效像素，大部分分辨率时钟中 Top/Left Border 和 Bottom / Right

Border 都是 0。“Blanking”是一行的同步时间，“Blanking”时间加上 Active” Video 时间就是一行的时间。“Blanking”又分为“Front Porch”、“Sync”、“Back Porch”三段。

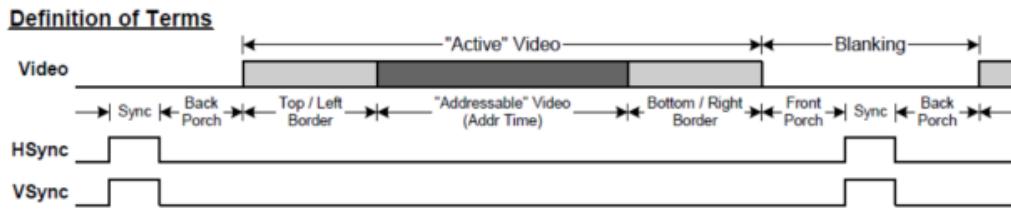


图 1 VGA 行扫描、场扫描时序示意图

图 13- 3

## 11.2 创建工程

了解 VGA 的时序后可以着手创建工程，

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 ram_pp, 工程路径 E:\demo\FPGA\lcdc 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件: 购买 Z7-Lite7020 用户选中 xc7z020clg400-2, 购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后, 点击确认

## 11.3 程序设计

本次实验的整体结果如下图所示：

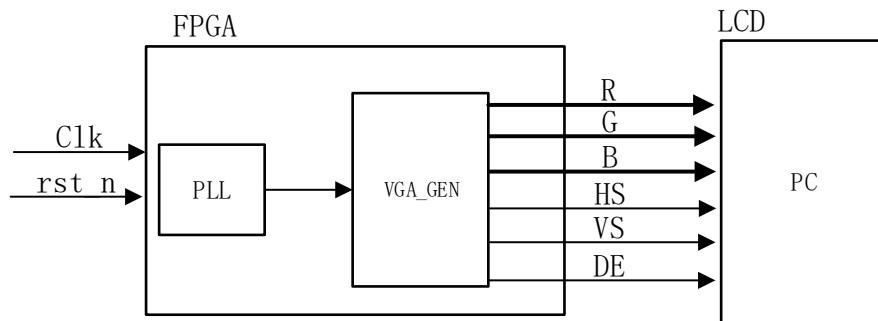


图 13-4

外部输入时钟和复位信号，外部输入的时钟经过 PLL 产生用于 VGA 图像输出所需要的时钟，并以 PLL 产生的时钟用来驱动 VGA\_GEN 模块开始工作，来产生行场同步信号，输出图像数据等，最终在 LCD 显示器上显示一个移动的方块。

### 1) 图形产生模块 VGA\_GEN

在本模块中，最重要的就是对行和场进行计数，其他所有信号都是根据行场信号的计数器的值来进行变化的。

下面对程序中的代码，进行简单的解释：

第 12 到 23 行是参数进行设置：

<b>12 parameter</b>	H_TOTAL = 525 - 1 ;//一行总共需要计数的值
<b>13 parameter</b>	H_SYNC = 41 - 1 ;//行同步计数值
<b>14 parameter</b>	H_START = 43 - 1 ;//行图像数据有效开始计数值
<b>15 parameter</b>	H_END = 523 - 1 ;//行图像数据有效结束计数值
<b>16 parameter</b>	V_TOTAL = 286 - 1 ;//场总共需要计数的值
<b>17 parameter</b>	V_SYNC = 10 - 1 ;//场同步计数值
<b>18 parameter</b>	V_START = 12 - 1 ;//场图像数据有效开始计数值
<b>19 parameter</b>	V_END = 284 - 1 ;//场图像数据有效结束计数值
<b>20 parameter</b>	SQUARE_X = 150; //方块的宽度
<b>21 parameter</b>	SQUARE_Y = 150; //方块的长度
<b>22 parameter</b>	SCREEN_X = 480; //屏幕水平长度
<b>23 parameter</b>	SCREEN_Y = 272; //屏幕垂直长度

第 42 到 49 行是调用 IP 核生成输出的像素时钟

<b>42</b>	clk_wiz_0 clock(
<b>43</b>	// Clock out ports

```

44     .clk_out1(vpg_pclk),      // output clk_out1
45     // Status and control signals
46     .reset(rst), // input reset
47     .locked(locked1),        // output locked
48     // Clock in ports
49     .clk_in1(clk));        // input clk_in1

```

52 到 76 行是行计数器和场计数器

```

52 //行计数器
53 always @(posedge vpg_pclk ) begin
54     if (rst==1'b1) begin
55         cnt_h <= 'd0;
56     end
57     else if (cnt_h == H_TOTAL) begin//计数到最大值, 清零
58         cnt_h <= 'd0;
59     end
60     else if(cnt_h != H_TOTAL) begin//还没有计数到最大值, 每个时钟周期加一
61         cnt_h <= cnt_h + 1'b1;
62     end
63 end
64
65 //场计数器
66 always @(posedge vpg_pclk ) begin
67     if (rst==1'b1) begin
68         cnt_v <='d0;
69     end
70     else if (cnt_v == V_TOTAL && cnt_h == H_TOTAL) begin//场计数器计数到最大值,
清零 (一帧结束)
71         cnt_v <= 'd0;
72     end
73     else if(cnt_h == H_TOTAL) begin//一行扫描结束, 场计数器加一
74         cnt_v <= cnt_v + 1'b1;
75     end
76 end

```

第 156 到 176 行根据计数器的值确定输出图像的值, 移动的方块是灰色的, 其余情况下, 显示背景色红绿蓝。

```
156 //rgb
```

```

157 always @(posedge vpg_pclk ) begin
158     if (rst==1'b1) begin
159         rgb <='d0;
160     end
161     else if(cnt_h    >=H_START+x    &&    cnt_h    <=H_START+SQUARE_X+x    &&
cnt_v >=V_START+y && cnt_v <=V_START+SQUARE_Y+y)begin
162         rgb <= 24'hFFB6C1;//输出方块图像
163     end
164     else if (cnt_h >=H_START && cnt_h <H_END && cnt_v >=V_START && cnt_v <V-END
&& cnt_h[4:0]>='d20) begin
165         rgb <=24'h00FF00;//green
166     end
167     else if (cnt_h >=H_START && cnt_h <H_END && cnt_v >=V_START && cnt_v <V-END
&& (cnt_h[4:0]>='d10 && cnt_h[2:0]<'d20)) begin
168         rgb <=24'h0000FF;//blue
169     end
170     else if (cnt_h >=H_START && cnt_h <H_END && cnt_v >=V_START && cnt_v <V-END
&& cnt_h[4:0]<'d10) begin
171         rgb <=24'hFF0000;//red
172     end
173     else begin
174         rgb <='d0;
175     end
176 end

```

## 11.4 添加约束文件

设计完成后，可以添加约束文件，本次实验，需要使用到微相科技的扩展子卡，在本次实验中，将其插入到 GPIO1 的插槽中，引脚分配如下：

```

#####
# clock define#####
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

#####
# key define#####
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]

```

```
##### vga clock ,de,hs,vs#####
set_property PACKAGE_PIN T10 [get_ports vpg_pclk]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_pclk]

set_property PACKAGE_PIN W15 [get_ports vpg_hs]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_hs]

set_property PACKAGE_PIN V15 [get_ports vpg_vs]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_vs]

set_property PACKAGE_PIN R14 [get_ports vpg_de]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_de]

set_property PACKAGE_PIN T11 [get_ports vpg_disp]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_disp]

set_property PACKAGE_PIN W16 [get_ports {rgb[0]}]
set_property PACKAGE_PIN V16 [get_ports {rgb[1]}]
set_property PACKAGE_PIN R18 [get_ports {rgb[2]}]
set_property PACKAGE_PIN T17 [get_ports {rgb[3]}]
set_property PACKAGE_PIN W13 [get_ports {rgb[4]}]
set_property PACKAGE_PIN V12 [get_ports {rgb[5]}]
set_property PACKAGE_PIN T15 [get_ports {rgb[6]}]
set_property PACKAGE_PIN T14 [get_ports {rgb[7]}]
set_property PACKAGE_PIN P18 [get_ports {rgb[8]}]
set_property PACKAGE_PIN Y17 [get_ports {rgb[9]}]
set_property PACKAGE_PIN Y16 [get_ports {rgb[10]}]
set_property PACKAGE_PIN V18 [get_ports {rgb[11]}]
set_property PACKAGE_PIN N17 [get_ports {rgb[12]}]
set_property PACKAGE_PIN V17 [get_ports {rgb[13]}]
set_property PACKAGE_PIN Y14 [get_ports {rgb[14]}]
set_property PACKAGE_PIN W14 [get_ports {rgb[15]}]
set_property PACKAGE_PIN Y19 [get_ports {rgb[16]}]
set_property PACKAGE_PIN Y18 [get_ports {rgb[17]}]
set_property PACKAGE_PIN W19 [get_ports {rgb[18]}]
set_property PACKAGE_PIN W18 [get_ports {rgb[19]}]
set_property PACKAGE_PIN U17 [get_ports {rgb[20]}]
set_property PACKAGE_PIN T16 [get_ports {rgb[21]}]
set_property PACKAGE_PIN R17 [get_ports {rgb[22]}]
```

```
set_property PACKAGE_PIN R16 [get_ports {rgb[23]}]
set_property IOSTANDARD LVCMOS33 [get_ports {rgb[*]}]
```

## 11.5 下载验证

等待 bit 文件生成后，将开发板按照如下方式于 LCD 扩展子卡相连接。

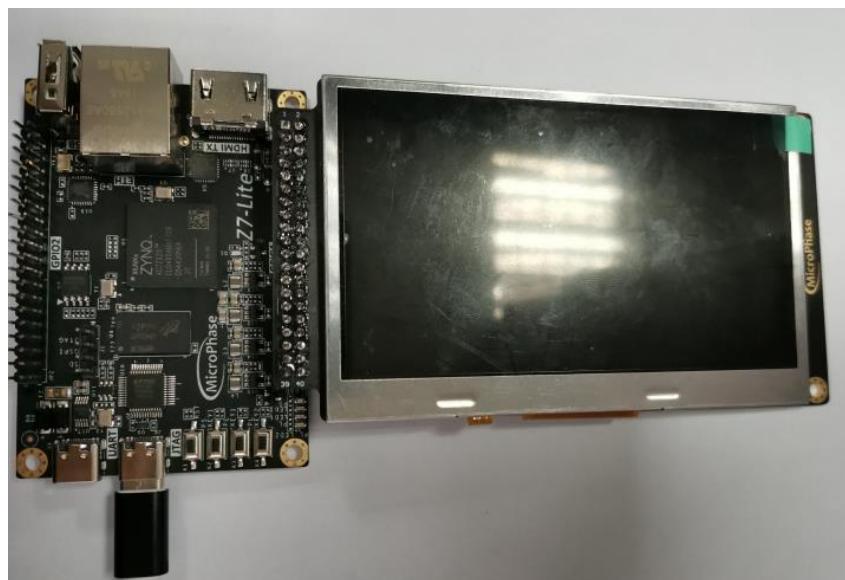


图 13-5

连接好 LCD 将 bit 文件下载到开发板中，即可看到屏幕中出现了移动的方块。

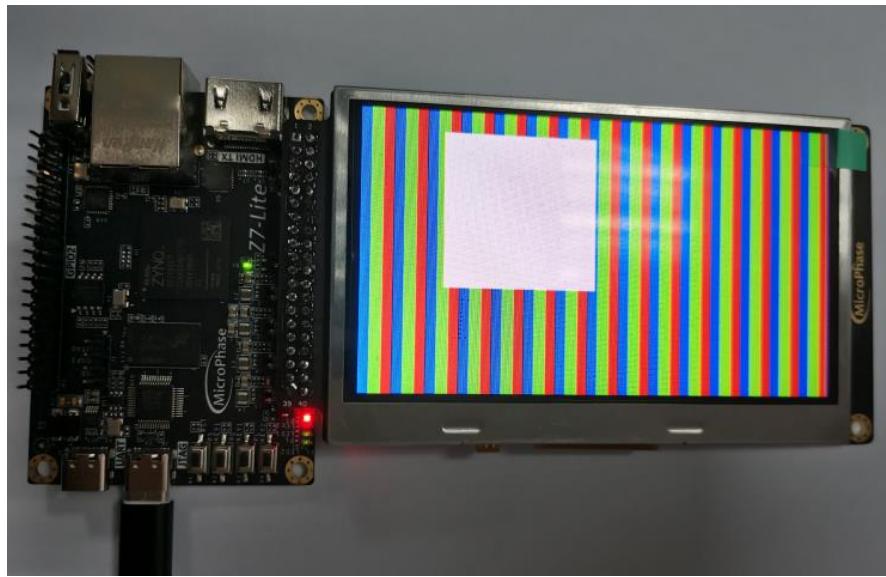


图 13-6

## 11.6 实验总结

在本次实验中，完成了使用 VGA 时序来驱动一个 LCD 显示的例子，本次实验中最重要的就是关于对 VGA 时序的理解，理解了相关的时序，在后面编写驱动程序的时候就显得十分简单了。

本次实验中由于时序的参数是可以重配置的，在之后对 VGA 模块进行移植的时候将会带来便利。

## 第12章 HDMI 输出彩条实验

### 12.1 实验简介

前面的章节中，已经了解了什么是 VGA 时序，并且已经成功地驱动了 LCD 显示器显示了简单地移动的方块，在 Z7-Lite 的开发板上，有两个 HDMI 接口，一个用于输入，一个用于输出，通过 FPGA 的管脚可以直接进行音视频的信号传输。在本次实验中，将会使用 FPGA 来完成 HDMI 到的输出实验。在上一章的基础上，将图像输出在支持 HDMI 的显示器上。首先需要了解以下 HDMI 的结构。

### 12.2 实验原理

#### 1) 概述

HDMI 系统架构由信源端和接收端组成。某个设备可能有一个或多个 HDMI 输入，一个或多个 HDMI 输出。这些设备上，每个 HDMI 输入都应该遵循 HDMI 接收端规则，每个 HDMI 输出都应该遵循 HDMI 信源端规则。如图所示，HDMI 线缆和连接器提供四个差分线对，组成 TMDS 数据和时钟通道。这些通道用于传递视频，音频和辅助数据。另外，HDMI 提供一个 VESA DDC 通道。DDC 是用于配置和在一个单独的信源端和一个单独的接收端交换状态。可选择的 CEC 在用户的各种不同的音视频产品中，提供高水平的控制功能。可选择的 HDMI 以太网和音频返回（HEAC），在连接的设备中提供以太网兼容的网络数据和一个和 TMDS 相对方向的音频回返通道。

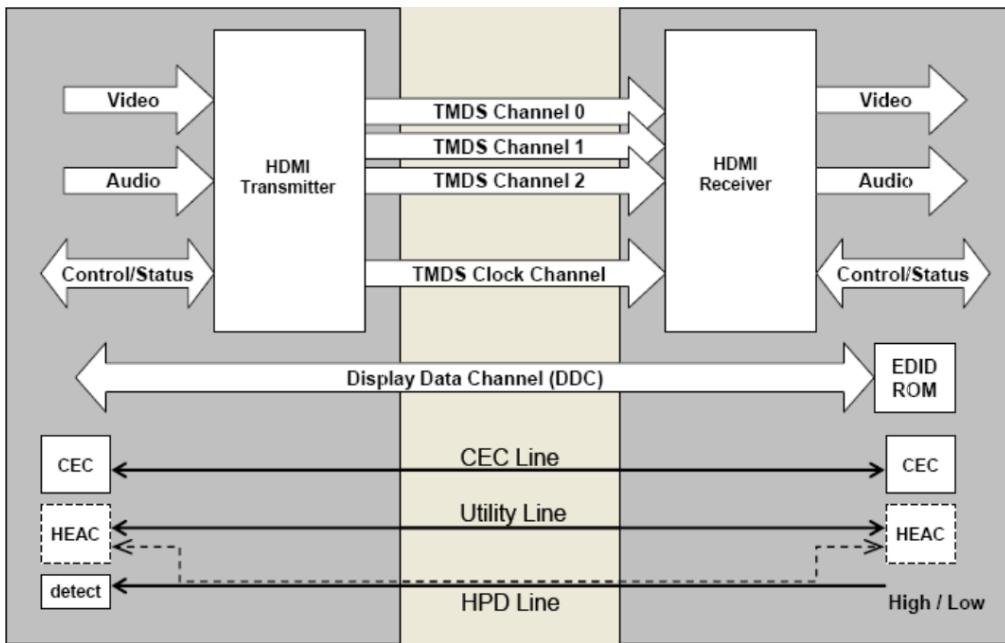


图 14-1

音频，视频和辅助数据在三个 TMDS 数据通道中传输。一个 TMDS 时钟，典型地是以视频像素速率，在 TMDS 时钟通道中传输，它被接收端做为一个频率参考，用于对三个 TMDS 数据通道的数据复原。在信源端，TMDS 编码将每个 TMDS 数据的 8 比特数据转换成 10 位的 DC - 平衡的最小变换序列，串行地，以每个 TMDS 时钟周期 10 位地，在差分线对上发送。视频数据，一个像素可以是 24, 30, 36, 48 比特。视频的默认 24 比特色深，在等于像素时钟的 TMDS 时钟上传递。更高的色深使用相应的更高的 TMDS 时钟率。视频格式 TMDS 时钟率低于 25M（比如 13.5M 的 480i/NTSC）可以使用重复像素发送的策略。视频像素可以用 RGB, YCbCr4:4:4, YCbCr4:2:2 格式编码。

## 2) 信号和编码

一个 HDMI 连接包括三个 TMDS 数据通道，一个 TMDS 时钟通道。TMDS 时钟通道以某种定常的速率运行，该速率和视频的像素率成比例。在每个 TMDS 时钟通道周期中，三个 TMDS 数据通道每个都发送 10 比特数据。这个 10 位的字被编码，采用某种不同的编码技术。输入到信源端的输入流，包含视频像素，数据包，和控制数据。数据包包括音频数据和辅助以及相关的纠错码。这些数据项被

不同地处理，在每个 TMDS 通道的 TMDS 编码器中表示为或者 2 比特控制数据，或者 4 比特报文数据，或者 8 比特视频数据。信源端在每个时钟周期内，对这些数据类型进行编码或者对某个边界字符编码。

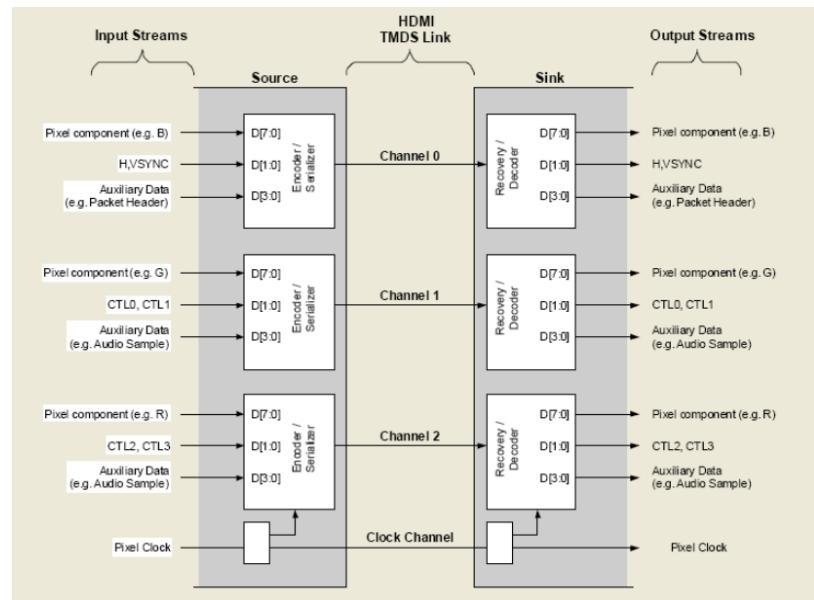


图 14-2

## 12.3 创建工程

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 ram_pp, 工程路径 E:\demo\FPGA\hdmi 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件：购买 Z7-Lite7020 用户选中 xc7z020clg400-2，购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后，点击确认

## 12.4 整体架构

本次实验的整体架构如下图所示，在 FPGA 内部例化有一个时钟模块，用于提供像素时钟和频率五倍于像素时钟的时钟。例化有一个 VGA 驱动模块，用于产生输出的图像。在 HDMI 发送模块内部分别由三个重要的部分，分别是 8b/10b 编码模块，串并转换模块和单端转差分模块。本次实验就主要围绕着这几个模块来完成。

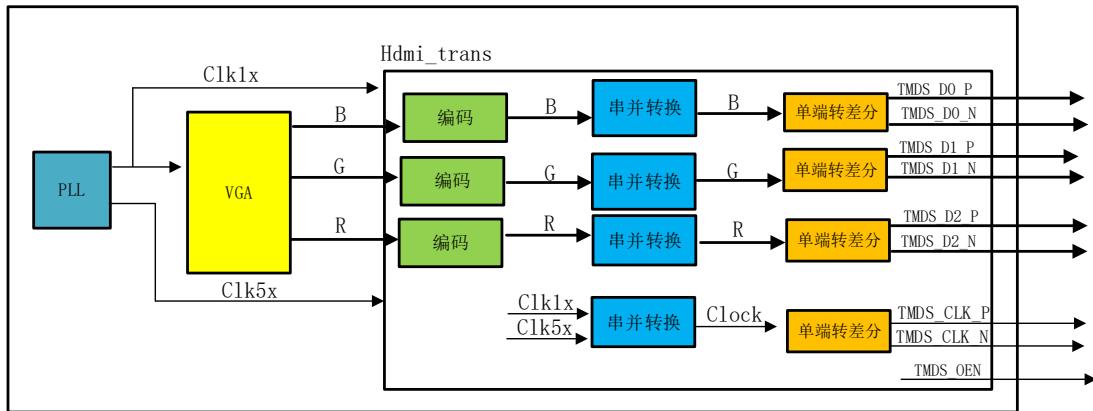


图 14-3

## 1) 编码模块

编码模块完成的是将 8bit 数据转换为 10bit 数据，RGB 每个颜色分量各占 8 位，那么每个通道上的颜色数据将通过一个 8B/10B 的编码器（Encoder）来转换成一个 10 位的像素字符。

三个通道的 DVI 编码器示意图。对于像素数据的 RGB 三个颜色通道，编码器的逻辑是完全相同的。VDE 用于各个通道选择输出视频像素数据还是控制数据，当 VDE 有效的时候表示当前传输的是视频数据，HSYNC 和 VSYNC 信号在蓝色通道进行编码得到 10 位字符，然后在视频消隐期传输。绿色和红色通道的控制信号 C0 和 C1 同样需要进行编码，并在消隐期输出。但是 DVI 规范中这两个通道的控制信号是预留的（未用到），因此将其置为 2'b00。

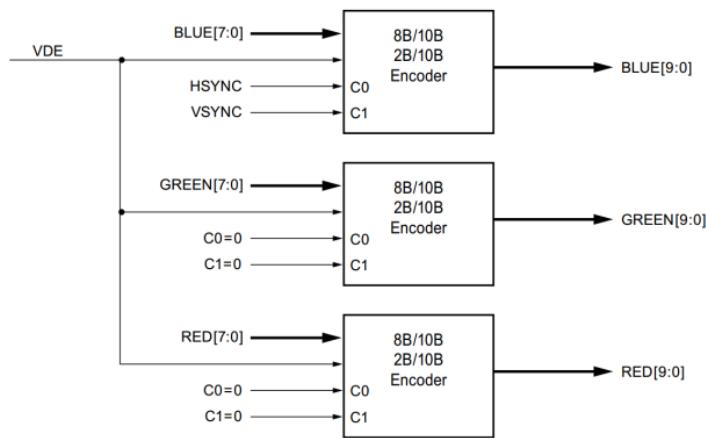


图 14- 4

本次实验的编码模块不需要自己编写，在 Xilinx 的官网上可以下载 8b/10b 的转换模块，在我们的工程中也已经提供。

## 2) 串并转换模块

经过编码模块后的 10 位的字符通过并串转换器 (**Serializer**) 转换成串行数据，最后由 TMDS 数据通道发送出去。这个 10:1 的并转串过程所生成的串行数据速率是实际像素时钟速率的 10 倍。由于 hdmi 在时钟的上升沿和下降沿都可以进行数据的传输，因此需要一个 5 倍于像素时钟的串行时钟即整体架构图中的 clk5x。

该模块中将会使用 Xilinx 提供的原语 OSERDESE2 进行串并转化，这个原语均可在 Xilinx 官方提供的 **UG768** 和 **UG471** 手册中查看。其原语的模型如下：可以看，一个该原语的模型，只支持 8bit 数据的串行转化，在本次实验中需要进行 10bit 数据的转化，因此需要使用到两个 OSERDESE2 原语级联来进行 10bit 数据的串行转化。

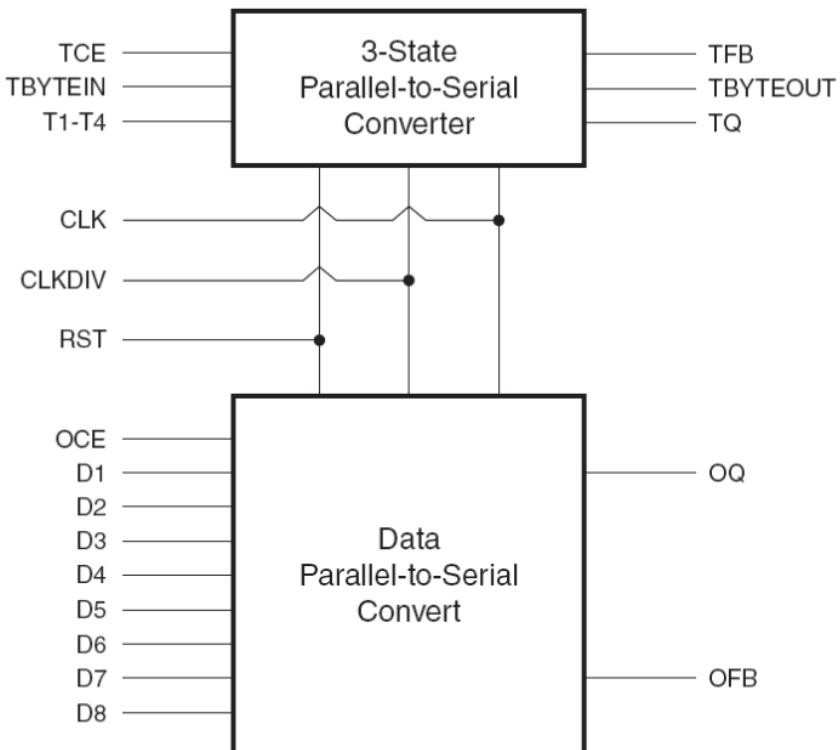


图 14-5

级联过后的原语模型如下: OSERDESE2 原语的 Master 端承接 10bit 数据的低 8bit, OSERDESE2 原语的 Slave 端的 D3 和 D4 承接 10bit 数据的高 2bit, 并将 Slave 的输出的 SHIFTOUT1、2 连接到 Master 的 SHIFTIN1、2 端, 输入的 10bit 数据将会按照由低位到高位的顺序从 Data Out 端输出。

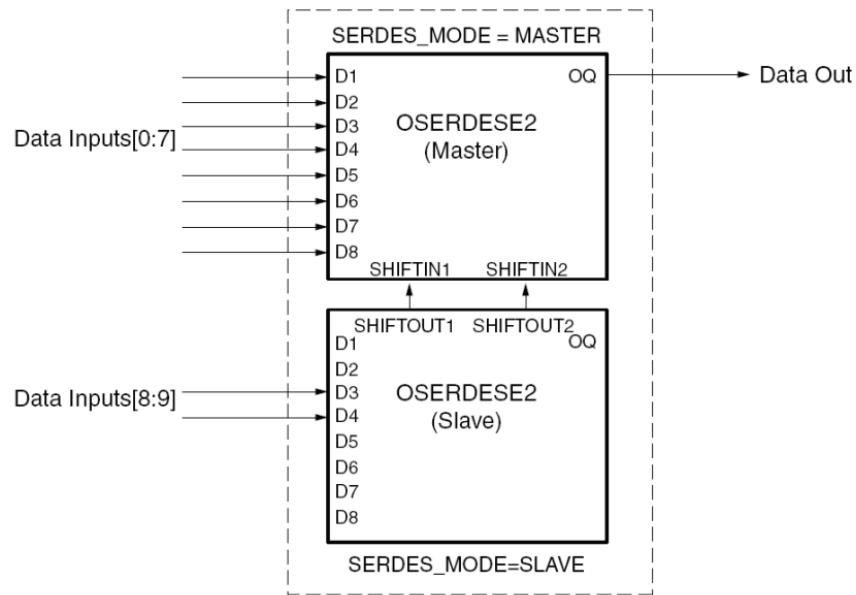


图 14- 6

### 3) 单端转差分模块

在完成串行转并行后，数据还是单端数据，此时还需要经过一个单端转差分的原语，才能使 FPGA 内部的信号转化成差分信号输出到 FPGA 外部。OBUFDS 的模型如下，OBUFDS 的使用同样可以参考 [UG768](#) 和 [UG471](#)。

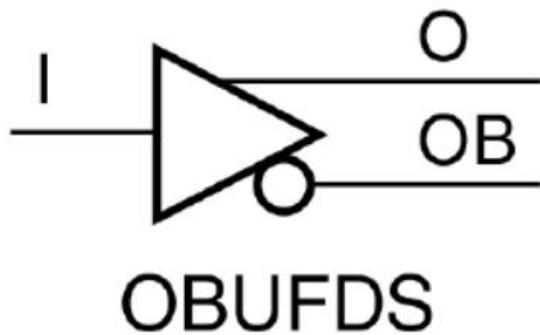


图 14- 7

最终完成的串并转化的代码如下：

```

1 module parallel_to_serial(
2     input    wire        clk1x      ,
3     input    wire        clk5x      ,
4     input    wire        rst       ,
5     input    wire [9:0] din   ,
6     output   wire       dout_p    ,
7     output   wire       dout_n
8 );
9
10
11 wire      dout      ;
12 wire      shift_in1 ;
13 wire      shift_in2 ;
14
15
16
17 OBUFDS #(
18     .IOSTANDARD("DEFAULT"), // Specify the output I/O standard
19     .SLEW("SLOW")          // Specify the output slew rate
20 ) OBUFDS_inst (
21     .O(dout_p),           // Diff_p output (connect directly to top-level port)
22     .OB(dout_n),          // Diff_n output (connect directly to top-level port)
23     .I(dout)              // Buffer input
24 );
25 OSERDESE2 #(
26     .DATA_RATE_OQ("DDR"), // DDR, SDR
27     .DATA_RATE_TQ("SDR"), // DDR, BUF, SDR
28     .DATA_WIDTH(10),      // Parallel data width (2-8,10,14)
29     .INIT_OQ(1'b0),       // Initial value of OQ output (1'b0,1'b1)
30     .INIT_TQ(1'b0),       // Initial value of TQ output (1'b0,1'b1)
31     .SERDES_MODE("MASTER"), // MASTER, SLAVE
32     .SRVAL_OQ(1'b0),      // OQ output value when SR is used (1'b0,1'b1)
33     .SRVAL_TQ(1'b0),      // TQ output value when SR is used (1'b0,1'b1)
34     .TBYTE_CTL("FALSE"),   // Enable tristate byte operation (FALSE, TRUE)
35     .TBYTE_SRC("FALSE"),   // Tristate byte source (FALSE, TRUE)
36     .TRISTATE_WIDTH(1)     // 3-state converter width (1,4)
37 )
38 OSERDESE2_inst_master (
39     .OFB(),                // 1-bit output: Feedback path for data
40     .OQ(dout),              // 1-bit output: Data path output
41     // SHIFTOUT1 / SHIFTOUT2: 1-bit (each) output: Data output expansion (1-bit each)

```

```

42      .SHIFTOUT1(),
43      .SHIFTOUT2(),
44      .TBYTEOUT(),    // 1-bit output: Byte group tristate
45      .TFB(),          // 1-bit output: 3-state control
46      .TQ(),           // 1-bit output: 3-state control
47      .CLK(clk5x),     // 1-bit input: High speed clock
48      .CLKDIV(clk1x),   // 1-bit input: Divided clock
49      // D1 - D8: 1-bit (each) input: Parallel data inputs (1-bit each)
50      .D1(din[0]),
51      .D2(din[1]),
52      .D3(din[2]),
53      .D4(din[3]),
54      .D5(din[4]),
55      .D6(din[5]),
56      .D7(din[6]),
57      .D8(din[7]),
58      .OCE(1'b1),        // 1-bit input: Output data clock enable
59      .RST(rst),         // 1-bit input: Reset
60      // SHIFTIN1 / SHIFTIN2: 1-bit (each) input: Data input expansion (1-bit each)
61      .SHIFTIN1(shift_in1),
62      .SHIFTIN2(shift_in2),
63      // T1 - T4: 1-bit (each) input: Parallel 3-state inputs
64      .T1(1'b0),
65      .T2(1'b0),
66      .T3(1'b0),
67      .T4(1'b0),
68      .TBYTEIN(1'b0),     // 1-bit input: Byte group tristate
69      .TCE(1'b0)          // 1-bit input: 3-state clock enable
70  );
71
72
73 OSERDESE2 #(
74     .DATA_RATE_OQ("DDR"),    // DDR, SDR
75     .DATA_RATE_TQ("SDR"),    // DDR, BUF, SDR
76     .DATA_WIDTH(10),         // Parallel data width (2-8,10,14)
77     .INIT_OQ(1'b0),          // Initial value of OQ output (1'b0,1'b1)
78     .INIT_TQ(1'b0),          // Initial value of TQ output (1'b0,1'b1)
79     .SERDES_MODE("SLAVE"), // MASTER, SLAVE
80     .SRVAL_OQ(1'b0),         // OQ output value when SR is used (1'b0,1'b1)
81     .SRVAL_TQ(1'b0),         // TQ output value when SR is used (1'b0,1'b1)
82     .TBYTE_CTL("FALSE"),     // Enable tristate byte operation (FALSE, TRUE)

```

```

83      .TBYTE_SRC("FALSE"),    // Tristate byte source (FALSE, TRUE)
84      .TRISTATE_WIDTH(1)     // 3-state converter width (1,4)
85  )
86  OSERDESE2_inst_slave (
87      .OFB(),                // 1-bit output: Feedback path for data
88      .OQ(),                 // 1-bit output: Data path output
89      // SHIFTOUT1 / SHIFTOUT2: 1-bit (each) output: Data output expansion (1-bit each)
90      .SHIFTOUT1(shift_in1),
91      .SHIFTOUT2(shift_in2),
92      .TBYTEOUT(),           // 1-bit output: Byte group tristate
93      .TFB(),                // 1-bit output: 3-state control
94      .TQ(),                 // 1-bit output: 3-state control
95      .CLK(clk5x),           // 1-bit input: High speed clock
96      .CLKDIV(clk1x),         // 1-bit input: Divided clock
97      // D1 - D8: 1-bit (each) input: Parallel data inputs (1-bit each)
98      .D1(),
99      .D2(),
100     .D3(din[8]),
101     .D4(din[9]),
102     .D5(),
103     .D6(),
104     .D7(),
105     .D8(),
106     .OCE(1'b1),             // 1-bit input: Output data clock enable
107     .RST(rst),              // 1-bit input: Reset
108     // SHIFTIN1 / SHIFTIN2: 1-bit (each) input: Data input expansion (1-bit each)
109     .SHIFTIN1(),
110     .SHIFTIN2(),
111     // T1 - T4: 1-bit (each) input: Parallel 3-state inputs
112     .T1(1'b0),
113     .T2(1'b0),
114     .T3(1'b0),
115     .T4(1'b0),
116     .TBYTEIN(1'b0),          // 1-bit input: Byte group tristate
117     .TCE(1'b0)               // 1-bit input: 3-state clock enable
118 );
119
120
121
122 endmodule

```

#### 4) VGA 时序发生模块

在前面的章节中，已经介绍了 VGA 模块，在本次实验中同样使用到 VGA 模块，在本次实验中，使用 VGA 模块并将其配置成 1920\*1080@60Hz 的时序，能够在 HDMI 显示器上显示处高清的图像。由于在前面章节中对时序进行了参数化的处理，在本次实验中只需更改其参数即可将 VGA 模块移植到本次工程中。

本次实验的时序参数如下：不同分辨率下的时序参数，在提供的 VESA 时序中可以进行查找。

parameter	H_TOTAL = 2200 - 1	//一行总共需要计数的值
parameter	H_SYNC = 44 - 1	//行同步计数值
parameter	H_START = 190 - 1	//行图像数据有效开始计数值
parameter	H_END = 2110 - 1	//行图像数据有效结束计数值
parameter	V_TOTAL = 1125 - 1	//场总共需要计数的值
parameter	V_SYNC = 5 - 1	//场同步计数值
parameter	V_START = 41 - 1	//场图像数据有效开始计数值
parameter	V_END = 1121 - 1	//场图像数据有效结束计数值
parameter	SQUARE_X = 500	//方块的宽度
parameter	SQUARE_Y = 500	//方块的长度
parameter	SCREEN_X = 1920	//屏幕水平长度
parameter	SCREEN_Y = 1080	//屏幕垂直长度

## 5) 时钟模块

在前面已经提到了本次实验所需要的两个时钟，一个是像素时钟，在本次实验中在 1080P 模式下的像素时钟是 148.5MHz，还需要一个 5 倍于像素时钟的串行时钟，用于在串并转换中。只需调用一个 clock IP 即可生成这两个时钟。

## 12.5 添加引脚约束

本次实验需要添加的引脚约束如下：分别对应于 TMDS 所需要的各个信号。其中 hdmi\_oen 是输出使能信号。

```
#####
# clock define#####
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
```

```
##### key define#####
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]

set_property PACKAGE_PIN U18 [get_ports hdmi_tx_clk_p]
set_property PACKAGE_PIN N20 [get_ports {hdmi_tx_chn_b_p}]
set_property PACKAGE_PIN T20 [get_ports {hdmi_tx_chn_g_p}]
set_property PACKAGE_PIN V20 [get_ports {hdmi_tx_chn_r_p}]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_r_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_g_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_b_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_clk_p]
```

## 12.6 下载验证

生成完 bit 文件过后即可进行下载验证，最终我们可以在 HDMI 显示器上看到已经出现了一个移动的方块。

首先连接好开发板后，将 bit 文件下载进 FPGA 中，然后插上 HDMI 线，观察在显示器上出现的画面。



图 14- 8

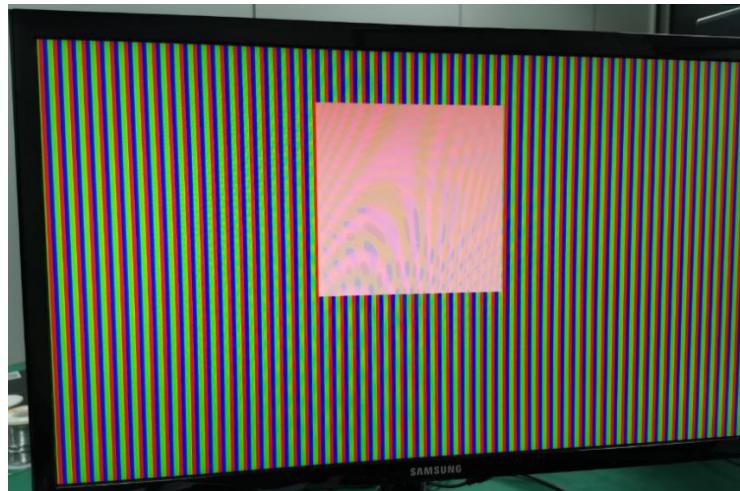


图 14- 9

## 12.7 实验总结

在本次实验中完成了 HDMI 显示器上显示移动方块的例子，本次实验主要内容是关于 HDMI 的原理和 HDMI 发送模块的实现，在 HDMI 发送实现模块中，需要关注的是 8b/10b 编码模块和串并转换模块，其中在串并转换模块中需要使用原语来进行串并转换，并将单端数据转换为差分数据，可以根据手册中提供的信息来使用这两个原语。

## 第13章 HDMI 输出图像实验

在上一次实验中，已经完成了在 HDMI 显示器上显示一个移动方块的，在本次实验中，将会显示一小幅图像在显示器上，由于开发板上的 DDR 芯片连接在 PS 端，在 PL 端开发时不利于直接访问，因此本次实验采用片内的存储空间来存储一幅图像。需要注意的是，由于 FPGA 内部的存储资源有限，因此一般不会使用到内部的存储资源来做大量数据的存储，本次实验，所要显示的图像大小是 256\*256。

### 13.1 生成图像文件

在进行本次实验之前，先使用 matlab 生成 16 进制格式的数据的图像。本次使用到的图片如下，大小为 256\*256。在将该图像与 matlab 脚本放置于同一文件夹下，然后新建 matlab 脚本，在 matlab 脚本中输入以下内容。



图 15- 1

```
% 实现了将 png 格式的图片转换为数字图像处理中的常用 RGB 格式
clear
rgb1=imread('image.png');%读取图片数据
imtool(rgb1)
r=rgb1(1:256,1:256,1);%取其中的 R 分量
```

```

g=rgb1(1:256,1:256,2);
b=rgb1(1:256,1:256,3);
uint32 rgb2[256,256];%定义一个 256*256, 深度为 32 位的矩阵
rgb2=bitor(bitshift(uint32(r),16),bitshift(uint32(g),8));
rgb2=bitor(rgb2,uint32(b));%进行或运算, 将像素值填充入矩阵中
fid0 = fopen('image.txt', 'wt');%打开一个文件
fprintf(fid0, '%06x,\n', rgb2);%将矩阵按照 6 位 16 进制数输出, 填充 0 占位
fid0 = fclose(fid0);%关闭这个文件

zero=zeros(256,256);%清空显示区域
R=cat(3,r,zero);%R 分量图像
G=cat(3,zero,g,zero);
B=cat(3,zero,zero,b);
RGB=cat(3,r,g,b);%原始图像
%显示图像
subplot(2,2,1),imshow(R),title('红色分量');
subplot(2,2,2),imshow(G),title('绿色分量');
subplot(2,2,3),imshow(B),title('蓝色分量');
subplot(2,2,4),imshow(RGB);

```

运行结束后将会生成一个 `image.txt` 的文本文件，在该文件的首部加入以下两行，使 Vivado 在初始化 ROM 时能够识别初始化文件

```
MEMORY_INITIALIZATION_RADIX=16;
```

```
MEMORY_INITIALIZATION_VECTOR=
```

```

1 MEMORY_INITIALIZATION_RADIX=16;
2 MEMORY_INITIALIZATION_VECTOR=
3
4 ffffff,
5 ffffff,
6 ffffff,
7 ffffff,
8 ffffff,
9 ffffff,
10 ffffff,
11 ffffff,
12 ffffff,
13 ffffff,
14 ffffff,
15 ffffff,
16 ffffff,
17 ffffff,
18 ffffff,
19 ffffff,
20 ffffff,
21 ffffff,

```

点击保存后，将该 txt 的文本数据的后缀名更改为.coe，这样就能使 Vivado 工具在初始化 RAM，ROM 时能够正确的对其进行初始化。

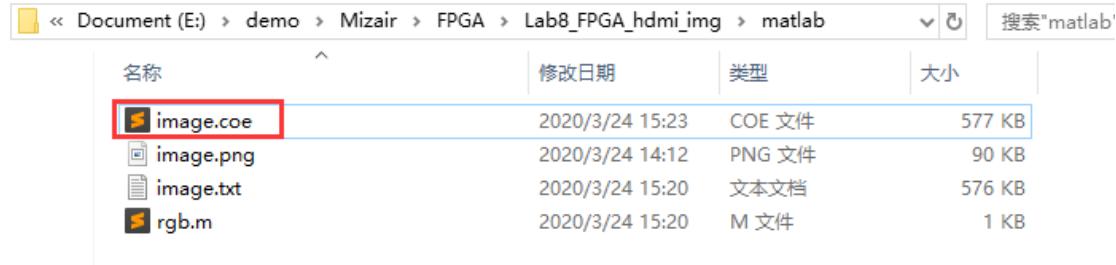


图 15-2

## 13.2 创建工程

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 ram_pp, 工程路径 E:\demo\FPGA\hdmi_img 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件：购买 Z7-Lite7020 用户选中 xc7z020clg400-2，购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后，点击确认

## 13.3 创建 ROM IP

在前面的章节中，介绍了如何使用 RAM，在本章中，介绍另一种片内存储资源 ROM。ROM 从字面意思理解就是 Read Only Memory 只读存储器，在实际使用过程中，可以使用 ROM 来保存一些少量的固定不变的内容。

点击 IP Catalog，输入 rom，选择 Block Memory Generator，进入 IP 的配置界

面。

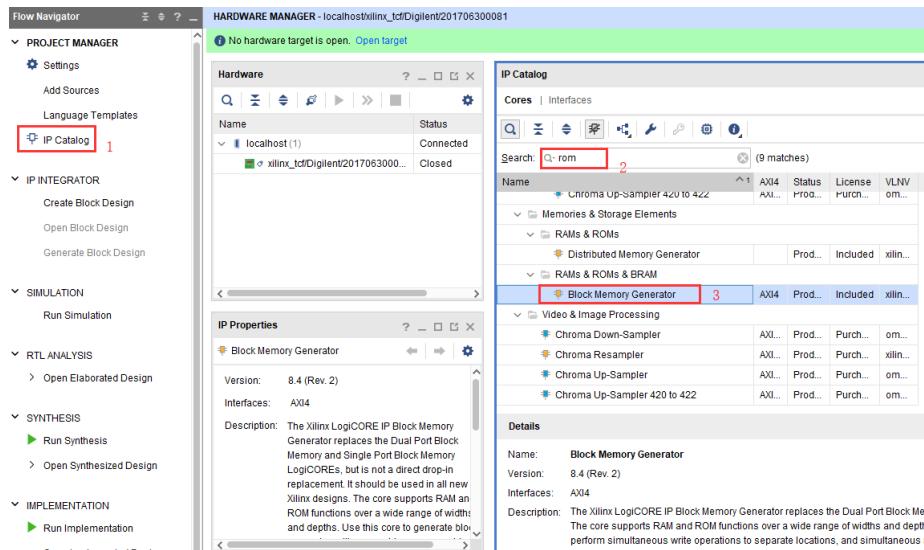


图 15- 3

在 Basic 页面，选择 Memory 类型为单端口 ROM；

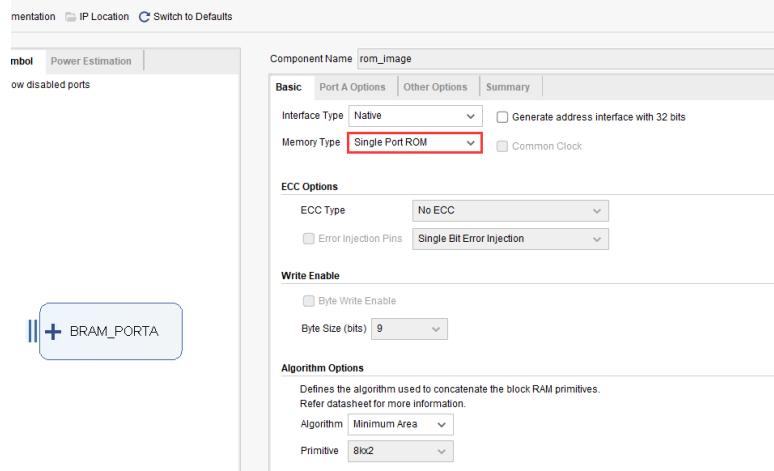


图 15- 4

设置数据位宽为 24bit，深度为 65536，端口使能类型选择一直使能；

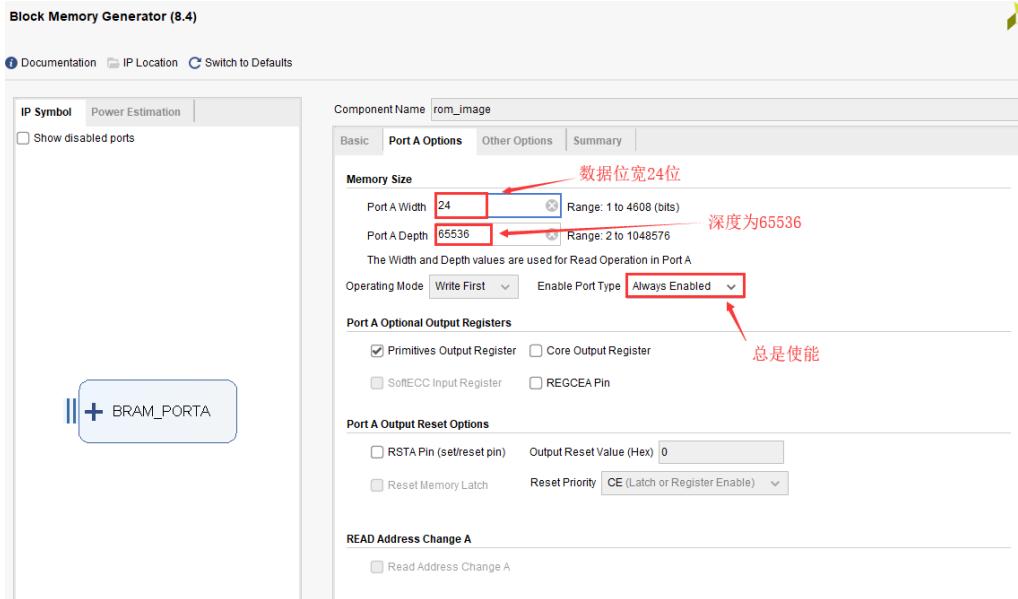


图 15- 5

在其他设置中，选择加载初始化文件，然后选择先前有 matlab 生成的 coe 文件，若正确无误，将成下图所示状态，若有错误，将会呈红色。

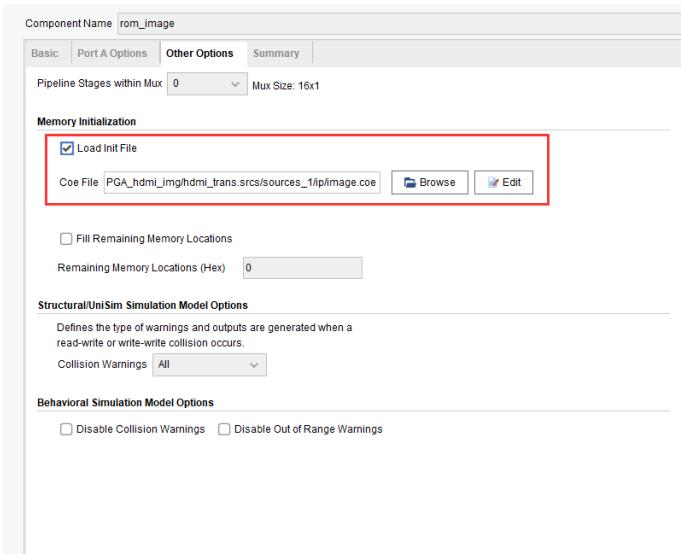


图 15- 6

配置完后，查看配置信息，从 summary 中可以看出，一共使用了 44 个 36K BRAM，Latency 为 2，级说明，从 ROM 中读出的数据相较于读取该数据的时钟周期上升沿由两个时钟周期的延时。

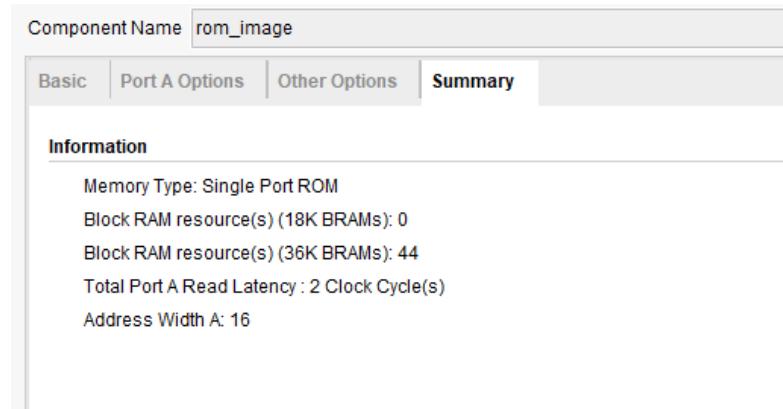


图 15- 7

## 13.4 整体结构设计

本次实验在上次实验的基础上，增加了一个 ROM 提供输出的图像数据，只需要对上次实验的工程做简单的增删改即可。整体结构如下：

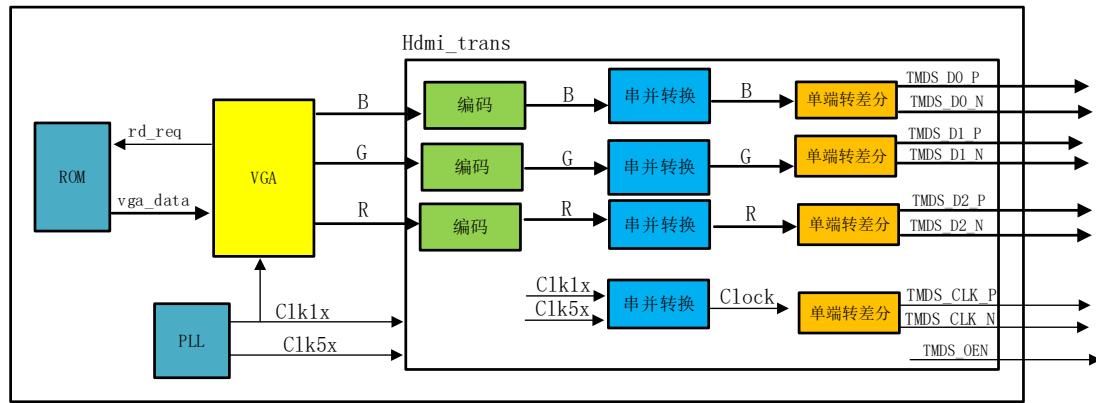


图 15- 8

### 1) ROM 读取模块

ROM 读取模块所实现的任务是，当接收到由 VGA 模块传递过来的读请求后，将数据从 ROM 中读出，并将读出的数据给到 VGA 模块输出到显示器上。本模块的代码如下：

```
1 `timescale 1ns / 1ps
```

```
2 module rd_image(
3   input  wire          clk ,
4   input  wire          rst ,
5   input  wire          rd_req ,
6   output wire[23:0]    rd_data
7 );
8
9
10 //=====
11 //parameter define
12 //=====
13 parameter      STOP_ADDR  = 256*256 - 1;
14
15 reg   [15:0]      rd_addr ;
16 wire  [23:0]      dout       ;
17
18 assign rd_data = dout;
19 //-----rd_addr-----
20 always @(posedge clk) begin
21   if (rst==1'b1) begin
22     rd_addr <= 'd0;
23   end
24   else if(rd_req==1'b1 )begin //读数据请求来临时，读地址加一
25     if(rd_addr == STOP_ADDR)begin
26       rd_addr <= 'd0;
27     end
28     else begin
29       rd_addr <= rd_addr + 1'b1;
30     end
31   end
32 end
33
34
35 rom_image inst_rom (
36   .clka(clk),    //input wire clka
37   .addr(a),      //input wire [15 : 0] addr
38   .dout(dout)   //output wire [23 : 0] dout
39 );
40 endmodule
```

## 2) VGA 模块修改

在 VGA 模块中，需要对原来的模块进行修改，其中需要新增一个读数据请求信号，利用该信号，从 ROM 中读出数据。前面注意到从 ROM 中读出的数据，相较于读处该数据的时钟由两个时钟周期的延时，因此想要在方块中显示该图像时，需要提前两拍给出读请求信号。

可以用下面的时序图来简单描述，行计数器计数到该显示图像数据的位置时，这时候输出的数据该是从 ROM 中读出的数据，但是由于 ROM 中的数据读出有两拍的延时，因此需要提前两拍发出读请求这样能够保证数据的对齐。

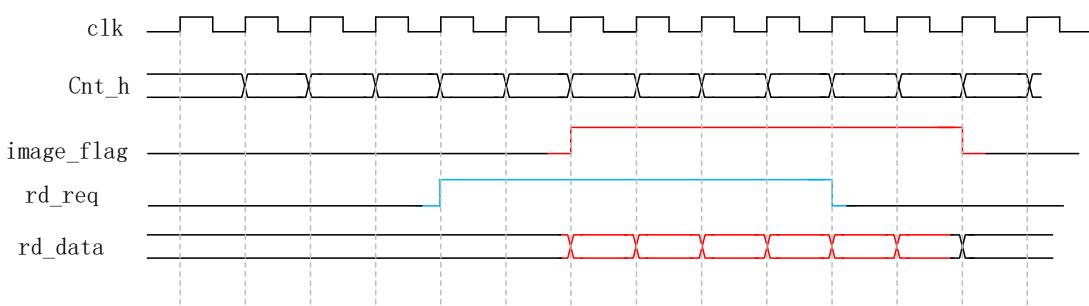


图 15- 9

### 13.5 下载验证

生成 bit 文件后，将 bit 文件进行下载过后，连接好 HDMI 线，可以在屏幕上看到最终输出的移动方块中的内容变成了输出的图像。



图 15- 10

## 13.6 实验总结

本次实验在上次实验的基础上，增加了一个 ROM 存储了一幅图像的数据，并最终将图像数据输出到显示器上。由于暂时还不便于使用 PS 端的 DDR 来进行图像的存储，这里使用的是片内存储资源，片内存储资源是比较紧缺的，可以看到，本次实验仅仅存储了 256\*256 大小的图像，都已经占用了 ZYNQ7000 系列 7020 芯片 1/3 的片内存储资源。

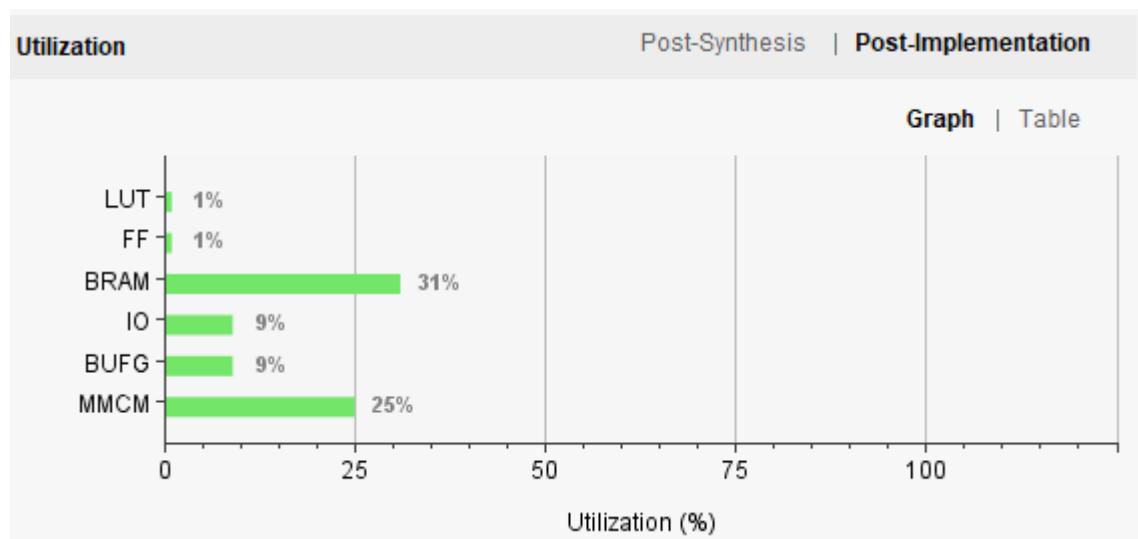


图 15- 11

在之后的部分中，会使用到 DDR 来进行数据缓冲，到时候，可以做一些简单的图像处理相关的内容。

## 第14章 AD\_DA 转换模块

### 14.1 实验简介

#### 1) 板卡简介

在本章中，我们使用微相科技的 ADDA 子卡模块完成一个简单的 AD\_DA 转换。高速 ADDA 子卡是一款高速 ADDA 数据转换卡，它包含一路高速 ADC 芯片（由 TI 公司生产的 ADC08060），一路高速 DA 转换芯片（由 TI 公司生产的 DAC908E）。该子卡可用于数据采集，信号产生，DSP 等应用。

该子卡与 GPIO 的连接示意图如下图所示：

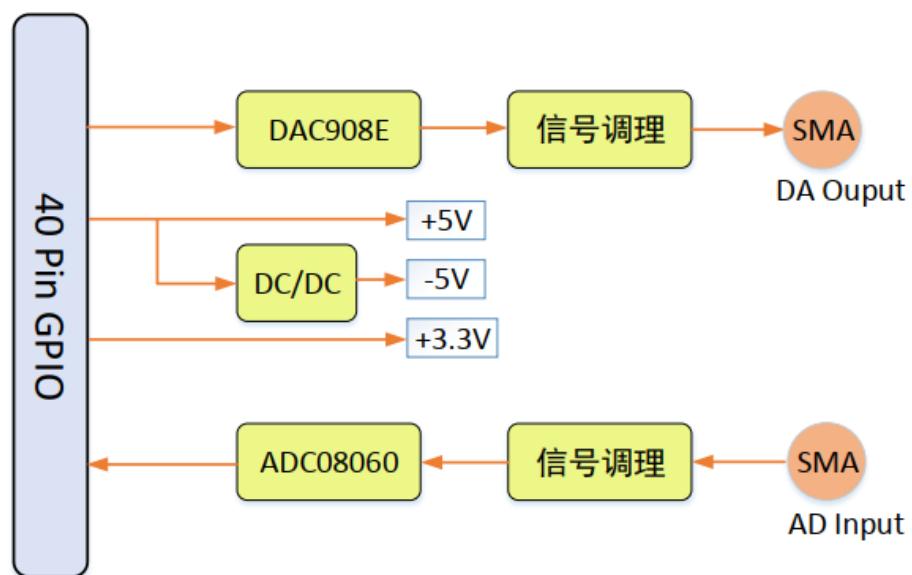


图 16-1

#### 2) 高速 DA 转换模块

高速 ADDA 子卡所采用的 DA 转换芯片的结构如下图所示：

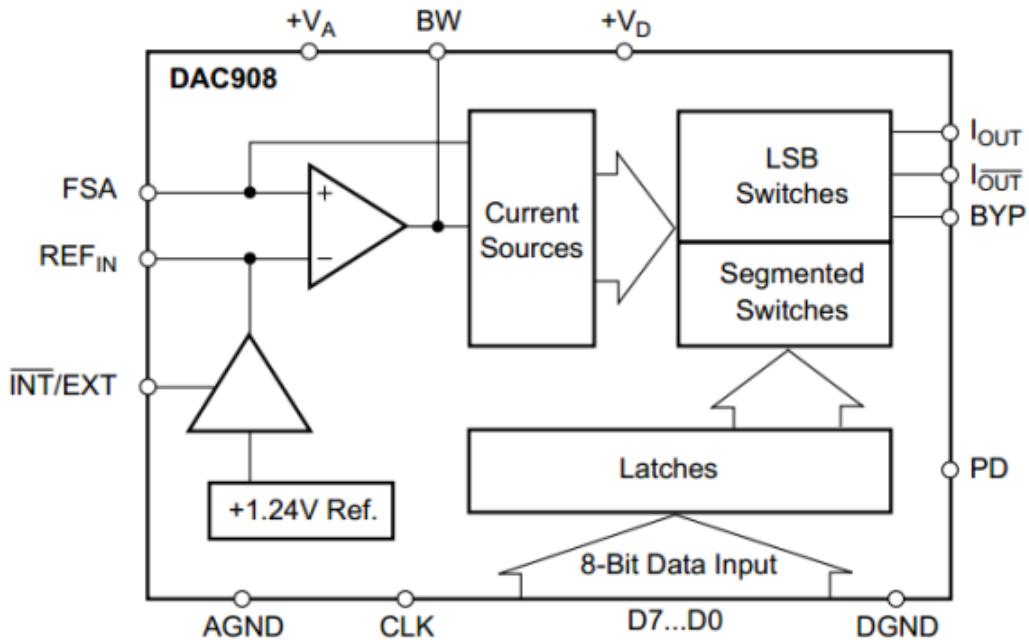


图 16- 2

从该芯片的 dataset 中我们可以看到它的参考设计如下图：

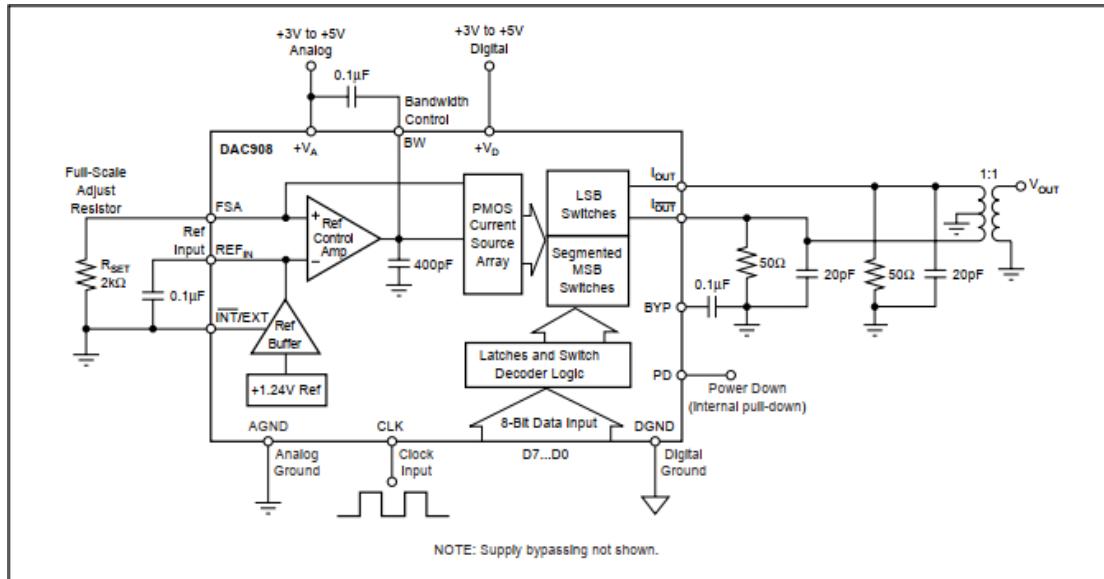


图 16- 3

从参考设计中，我们可以看到其中 CLK 信号是给到该芯片的输入信号，8bit 的数据是给到该芯片进行数模转换的。此外我们还需要关注，对该芯片操作的一些时序，具体需要注意的时序如下图：

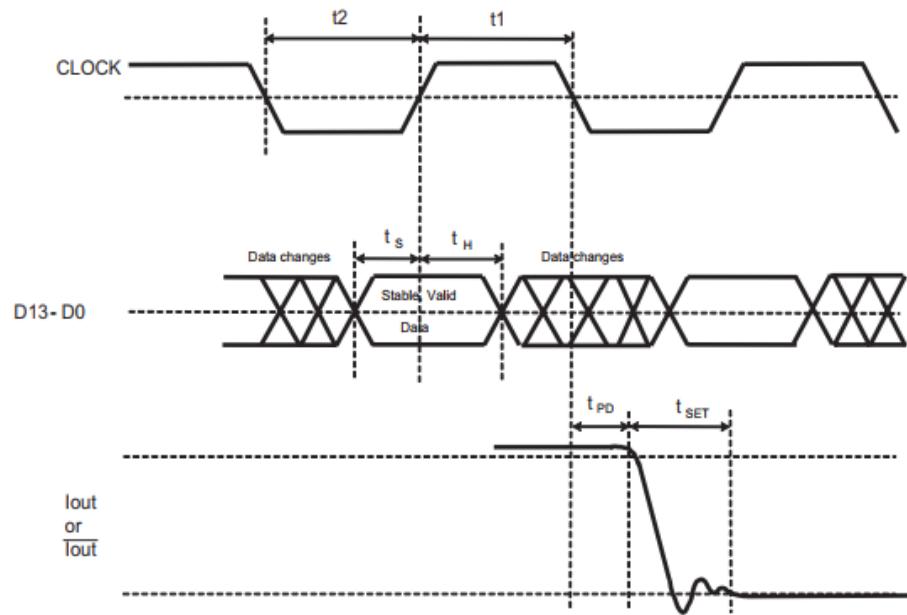


图 16-4

其中的时序参数列表如下图所示，我们在进行 FPGA 设计的时候，需要根据这些参数来设计输出数据和时钟之间的关系。

SYMBOL	DESCRIPTION	MIN	TYP	MAX	UNITS
$t_1$	Clock Pulse HIGH Time		3		ns
$t_2$	Clock Pulse LOW Time		3		ns
$t_s$	Data Setup Time		1.0		ns
$t_h$	Data Hold Time		1.5		ns
$t_{PD}$	Propagation Delay Time		1		ns
$t_{SET}$	Output Settling Time to 0.1%		30		ns

图 16-5

### 3) 高速 AD 转换模块

高速 AD 转换模块所使用芯片 ADC08060 的结构如下图：

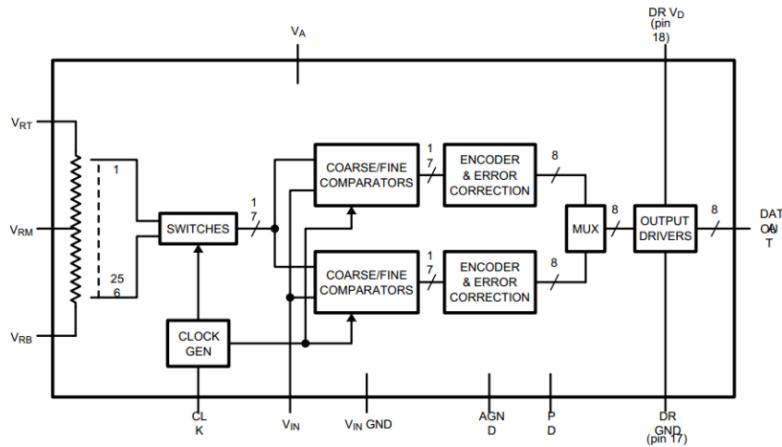


图 16- 6

从该结构中可以看出，该芯片的工作时钟也是由外部所提供，在芯片内部会通过一个类似 PLL 的结构对时钟进行转化供采样使用，最终输出 8 bit 的数字信号。

其时序图如下所示：

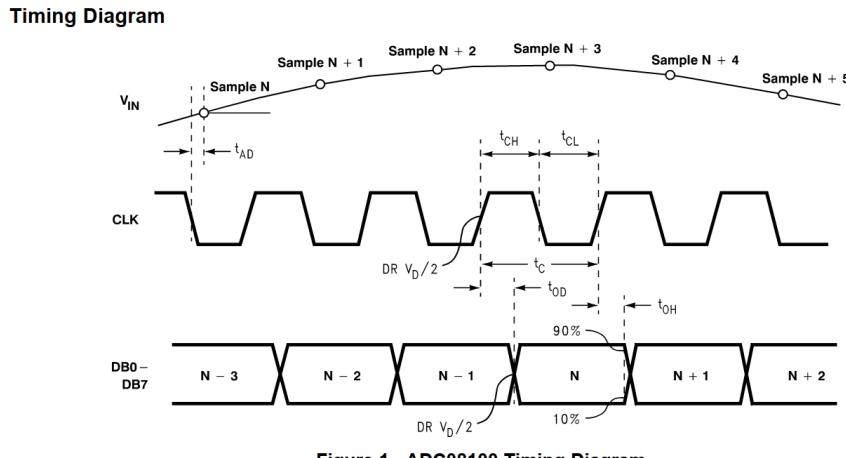


Figure 1. ADC08100 Timing Diagram

图 16- 7

时序参数如下图所示，在实际设计的时候，若想要有更高的速度，可以更具以下时序参数对时序进行设计和修改。

AC ELECTRICAL CHARACTERISTICS					
$f_{C1}$	Maximum Conversion Rate		125	100	MHz (min)
$f_{C2}$	Minimum Conversion Rate		20		MHz
$t_{CL}$	Minimum Clock Low Time			4.5	ns (min)
$t_{CH}$	Minimum Clock High Time			4.5	ns (min)
$t_{OH}$	Output Hold Time	CLK Rise to Data Invalid	4.4		ns
$t_{OD}$	Output Delay	CLK Rise to Data Valid	5.9	8.5	ns (max)
	Pipeline Delay (Latency)		2.5		Clock Cycles
$t_{AD}$	Sampling (Aperture) Delay	CLK Fall to Acquisition of Data	1.5		ns
$t_{AJ}$	Aperture Jitter		2		ps rms

图 16- 8

## 14.2 创建工程

按照表格所示步骤进行工程的创建：

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 pcie_test, 工程路径 E:\demo\adda_test 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件购买 Z7-Lite7020 用户选中 xc7z020clg400-2, 购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后, 点击确认

## 14.3 使用 matlab 生成 coe 文件

在进行本实验之前首先需要生成一个 rom 的初始化文件。在 matlab 下执行以下脚本。

```
X=linspace(0,2*pi,512);
Y=(255/2)*(sin(X)+1);
Y=uint8(Y);
plot(X,Y);
fid0 = fopen('sin512.txt', 'wt');%打开一个文件
```

```
fprintf(fid0, '%02x,\n', Y);%将矩阵按照2位16进制数输出，填充0占位
fid0 = fclose(fid0);
```

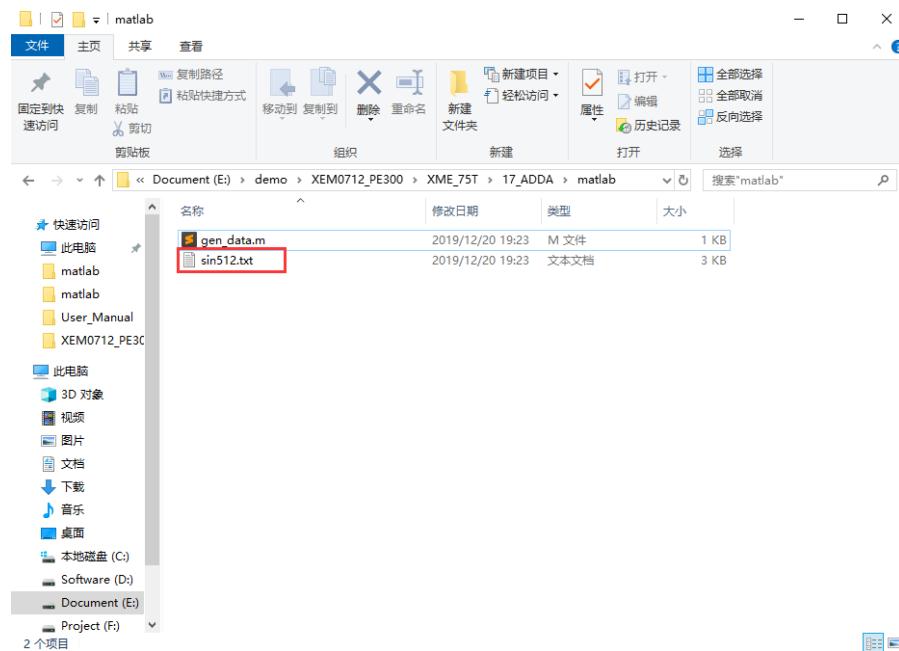


图 16-9

执行完成之后会生成一个 **TXT** 格式的文本数据，我们需要在该文本中添加以下内容：

**MEMORY\_INITIALIZATION\_RADIX=16;**

**MEMORY\_INITIALIZATION\_VECTOR=** 如下图所示：

```
1 MEMORY_INITIALIZATION_RADIX=16;
2 MEMORY_INITIALIZATION_VECTOR=
3 80,
4 81,
5 83,
6 84,
7 86,
8 87,
9 89,
10 8a,
11 8c,
12 8e,
```

图 16-10

更改完成后，我们需要将该 txt 格式的文件后缀名更改为.coe。

## 14.4 初始化 ram

### 1) 初始化 rom

接下来我们将该波形数据使用一个 rom 存储起来，这样就能通过对 rom 的读取，来产生波形。

点击 IP catalog 进入 IP 生成页面：

在搜索栏中输入 rom，在下方选中 block Memory Generator

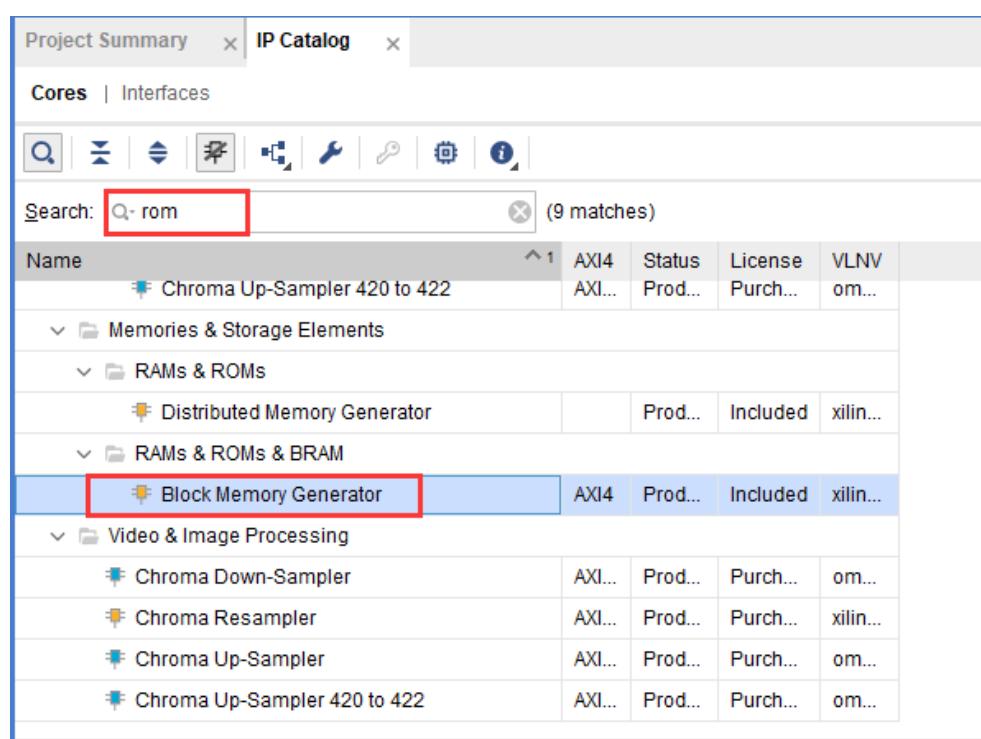


图 16- 11

在 basic 配置页面，对要生成的 ram 进行命名，选择单端口 rom；

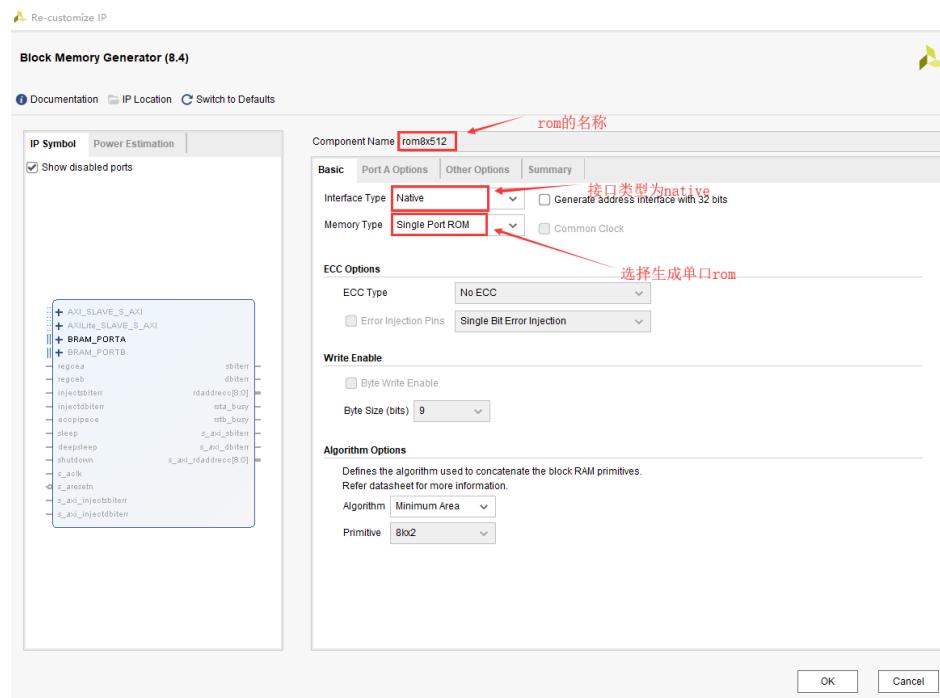


图 16- 12

在端口 A 的配置页面需要对 rom 的位宽和深度进行配置，配置成深度为 512 位宽为 8 的模式。

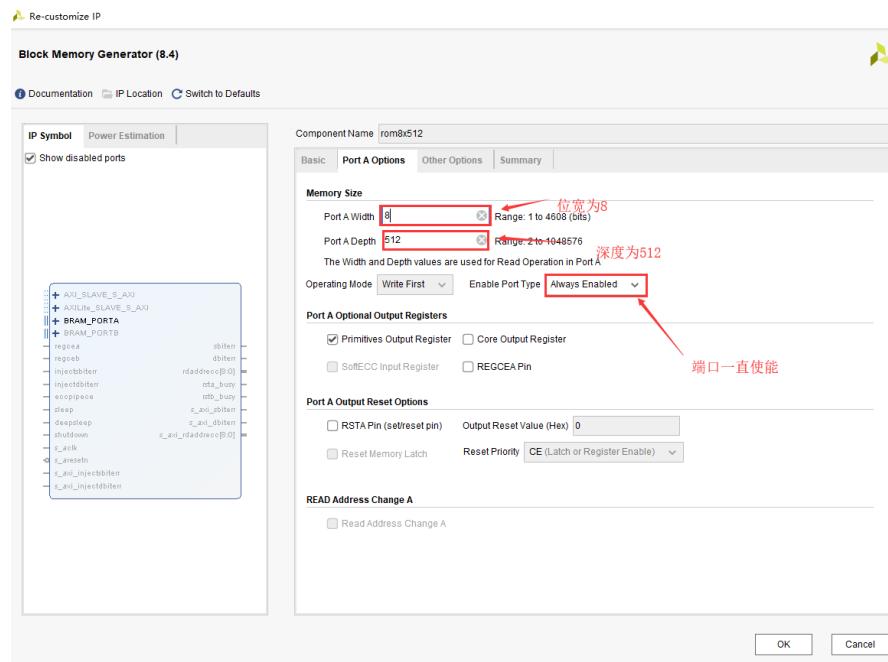


图 16- 13

接着添加导入初始化文件 coe 文件

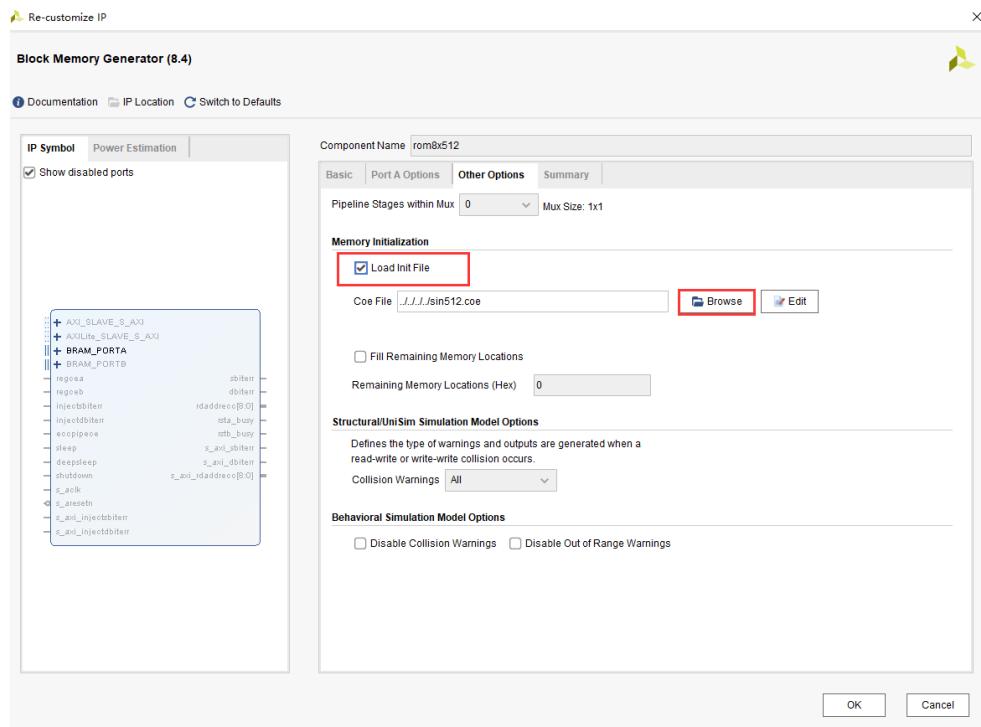


图 16-14

选择 matlab 文件夹下的刚刚生成的 coe 文件。

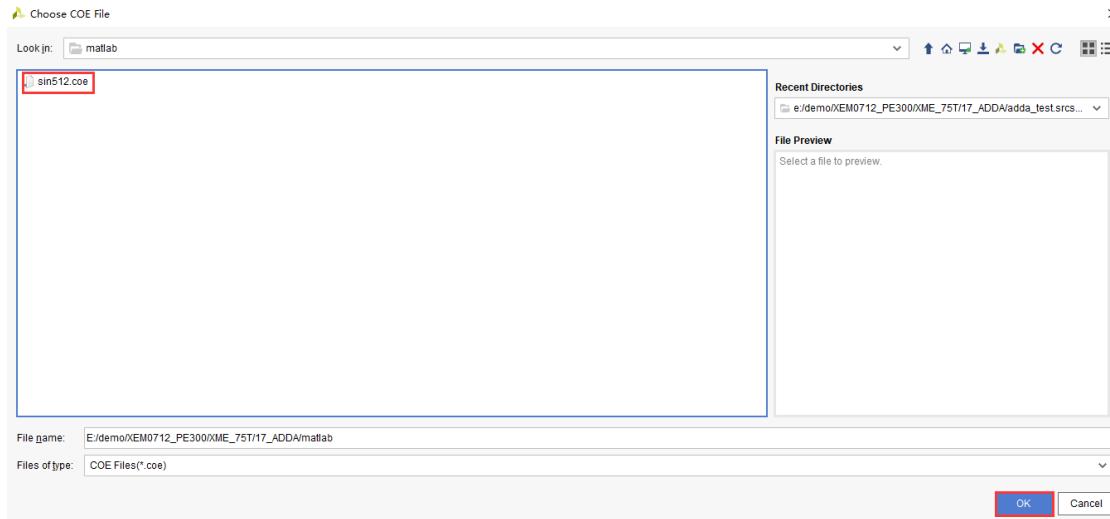


图 16-15

至此完成了对 rom 的配置，接下来需要生成一个 PLL。

## 2) 配置时钟 IP

其中一个输出时钟配置为 50MHz，另一个时钟配置为 20MHz。按如下步骤进行操作。

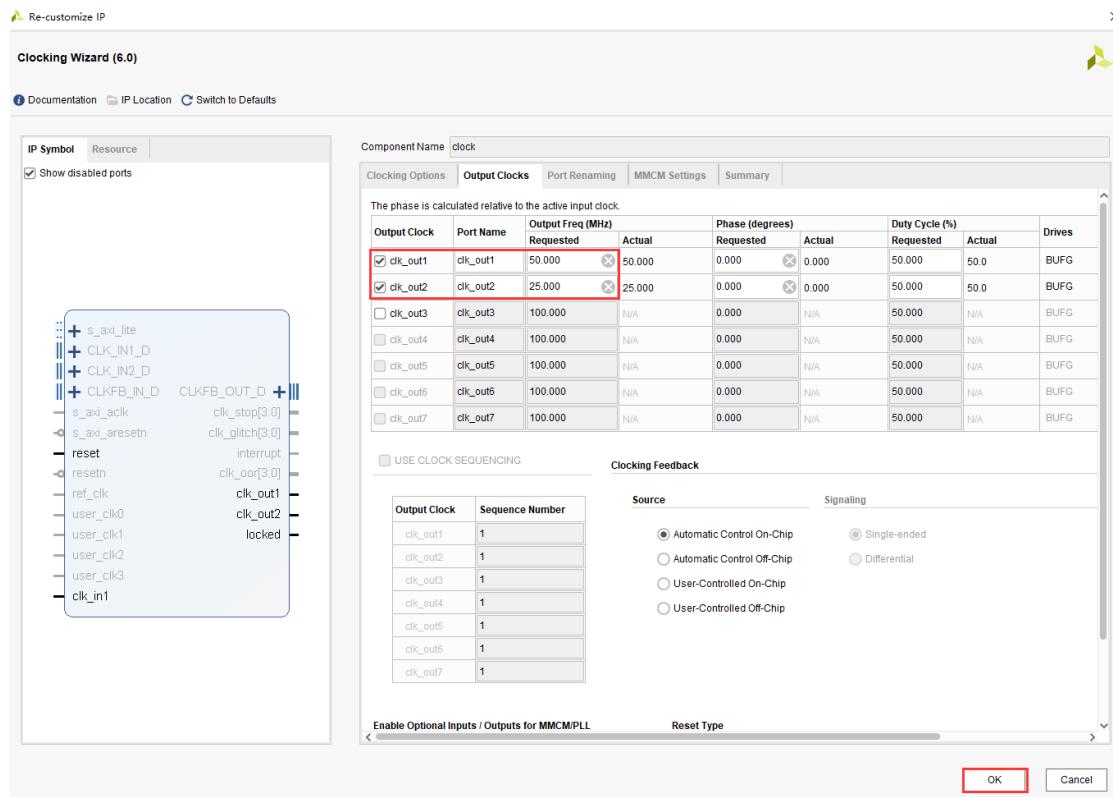


图 16- 16

### 3) 添加 ILA 用于观察结果

ILA 即嵌入式逻辑分析仪，使用 ila 能够通过使用 FPGA 内部的资源用于观察 FPGA 内部信号。这样在调试的时候会带来极大的便利。

本次实验使用到两个 ila，一个用于观察 DA 转换时，从 FPGA 输出的数据，一个用于 AD 转换后 FPGA 内部采集到的数据。

点击 IP Catalog → 收索 ila → 选中图中所示的 ila，进入配置界面。

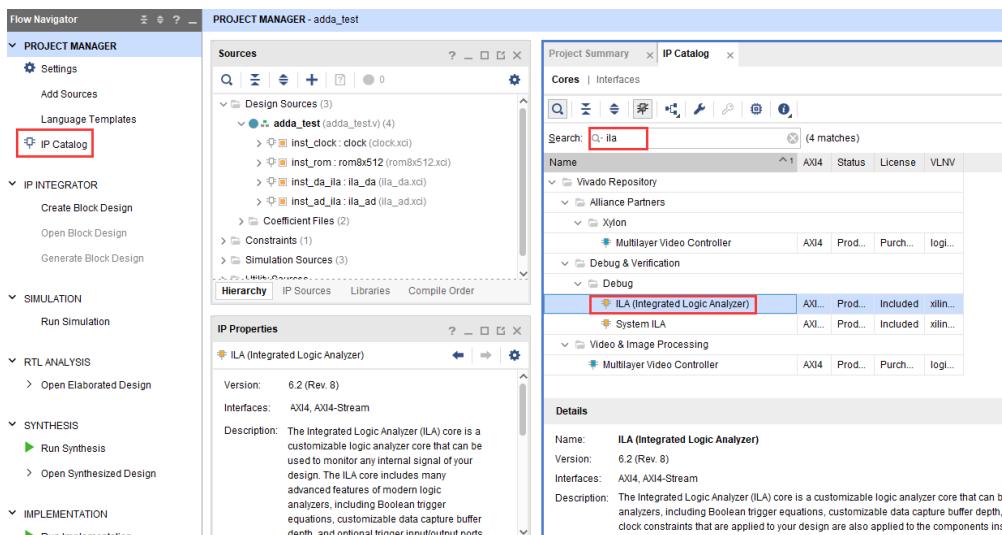


图 16-17

在 IP 配置界面，选择接口类型为 native 接口类型，探针数量设置为 1，采样深度设置为 1024；接下来设置探针的位宽为 8，即要观察的信号的位宽时是 8bit 的。

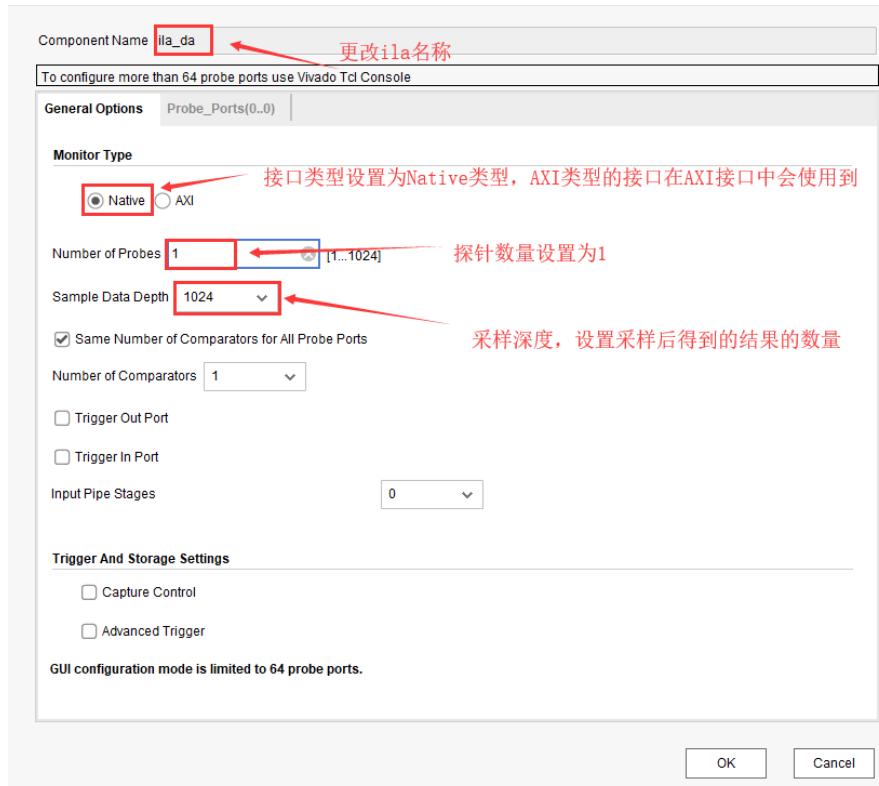


图 16-18

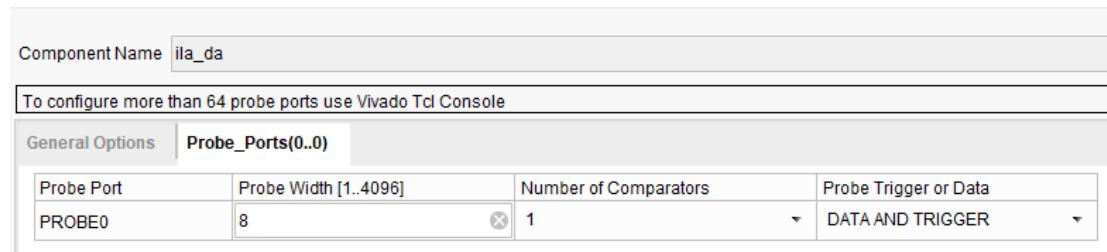


图 16-19

本次实验中的两个 ila 按照此方法进行配置即可。

## 14.5 程序设计

本次实验比较简单，主要是通过 ila 来观察输入输出的波形，本次实验的程序设计如下：

```

1 `timescale 1ns / 1ps
2 module adda_test(
3   input  wire      clk      ,
4   input  wire      rst_n   ,
5
6   output wire      ad_clk   ,//ad 模块时钟
7   input  wire[7:0]ad_data ,//ad 转换过后的数据
8
9   output wire      da_clk   ,//da 模块输出时钟
10  output wire[7:0]da_data   //da 模块输出数据
11 );
12
13 wire      clk_50m ;
14 wire      clk_25m ;
15 wire      sys_clk ;
16 wire      locked ;
17 reg   [8:0]rom_addr ;
18
19
20 assign    ad_clk = clk_25m ;
21 assign    da_clk = clk_50m ;
22
23 clock inst_clock(

```

```
24     .clk_out1(clk_50m),      // output clk_out1
25     .clk_out2(clk_25m),       // output clk_out2
26     .reset(~rst_n), // input reset
27     .locked(locked),        // output locked
28     .clk_in1(clk)          // input clk_in1
29   );
30 always @(posedge clk_50m or negedge rst_n) begin
31   if (rst_n==1'b0) begin
32     rom_addr <= 'd0;
33   end
34   else begin
35     rom_addr <= rom_addr + 1'b1;
36   end
37 end
38
39
40
41 rom8x512 inst_rom (
42   .clka(clk_50m),    // input wire clka
43   .addra(rom_addr), // input wire [8 : 0] addra
44   .douta(da_data)  // output wire [7 : 0] douta
45 );
46
47 ila_da inst_da_ila (
48   .clk(clk_50m), // input wire clk
49
50
51   .probe0(da_data) // input wire [7:0] probe0
52 );
53
54 ila_ad inst_ad_ila (
55   .clk(clk_25m), // input wire clk
56
57
58   .probe0(ad_data) // input wire [7:0] probe0
59 );
60 endmodule
```

## 14.6 添加约束文件

本次实验的约束文件如下：

```
##### clock define#####
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

##### key define#####
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]

set_property PACKAGE_PIN W19 [get_ports {da_data[0]}]
set_property PACKAGE_PIN W18 [get_ports {da_data[1]}]
set_property PACKAGE_PIN U17 [get_ports {da_data[2]}]
set_property PACKAGE_PIN T16 [get_ports {da_data[3]}]
set_property PACKAGE_PIN R17 [get_ports {da_data[4]}]
set_property PACKAGE_PIN R16 [get_ports {da_data[5]}]
set_property PACKAGE_PIN P18 [get_ports {da_data[6]}]
set_property PACKAGE_PIN N17 [get_ports {da_data[7]}]
set_property PACKAGE_PIN Y18 [get_ports da_clk]

set_property SLEW FAST [get_ports {da_data[*]}]
set_property SLEW FAST [get_ports da_clk]

set_property IOSTANDARD LVCMOS33 [get_ports {da_data[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports da_clk]

set_property PACKAGE_PIN U12 [get_ports ad_clk]
set_property PACKAGE_PIN V13 [get_ports {ad_data[0]}]
set_property PACKAGE_PIN U13 [get_ports {ad_data[1]}]
set_property PACKAGE_PIN U15 [get_ports {ad_data[2]}]
set_property PACKAGE_PIN U14 [get_ports {ad_data[3]}]
set_property PACKAGE_PIN R14 [get_ports {ad_data[4]}]
set_property PACKAGE_PIN P14 [get_ports {ad_data[5]}]
set_property PACKAGE_PIN W15 [get_ports {ad_data[6]}]
set_property PACKAGE_PIN V15 [get_ports {ad_data[7]}]

set_property SLEW FAST [get_ports ad_clk]

set_property IOSTANDARD LVCMOS33 [get_ports ad_clk]
set_property IOSTANDARD LVCMOS33 [get_ports {ad_data[*]}]
```

接下来可以综合生成 bit 文件。

## 14.7 下载验证

生成 bit 文件后我们将 ADDA 子卡与 Z7-Lite 开发板按照如下方式进行连接：

将 bit 文件下载至 FPGA 中，通过 ila 抓取数据，观察输出的波形。

进入波形窗口，将数据显示格式转换为模拟量的形式，然后观察波形，如下图：



图 16-20

将 bit 文件下载完成后，会在右侧弹出一个波形窗口，在该窗口中可以看到希望采样观察的信号。红色区域内，是在使用 ila 进行分析时可使用的用来采集和观察波形的按钮，黄色区域内时当前触发的状态跟触发点采样窗口个数的设置。

紫色区域可以设置触发条件，只要当条件满足后才会触发，采样得到波形。

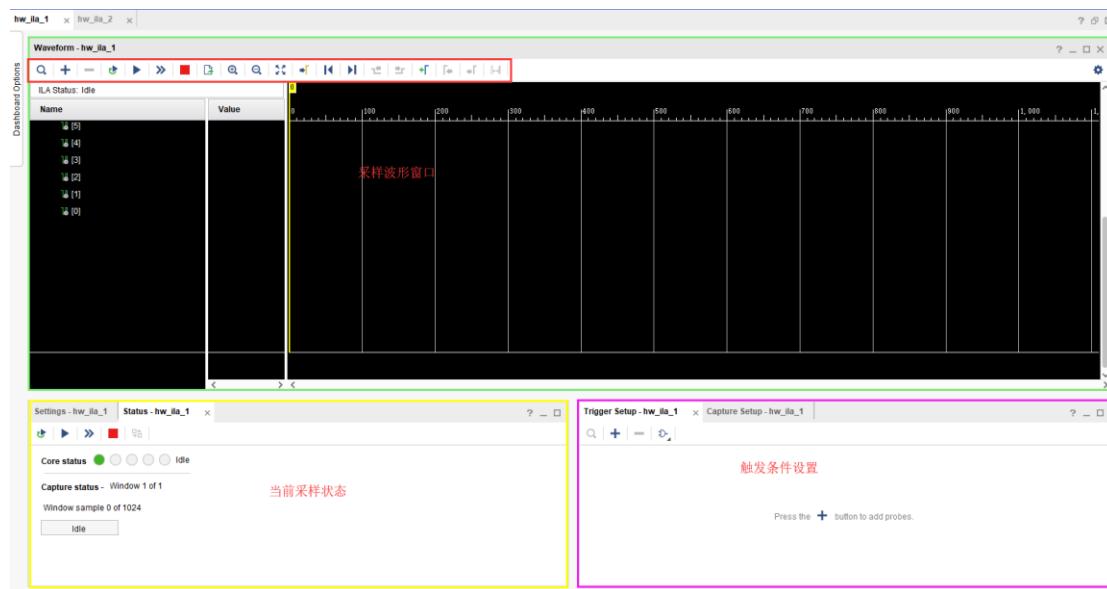


图 16- 21

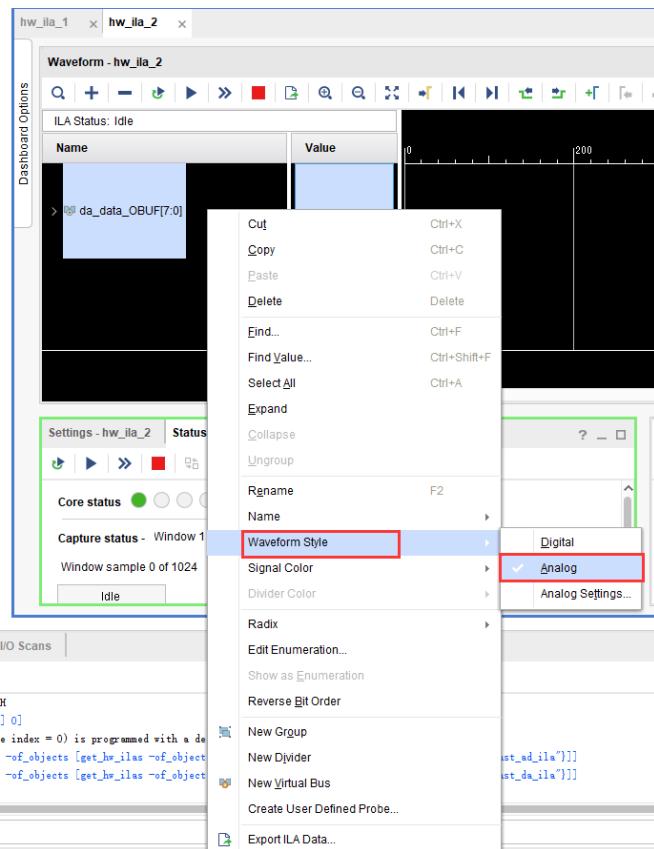
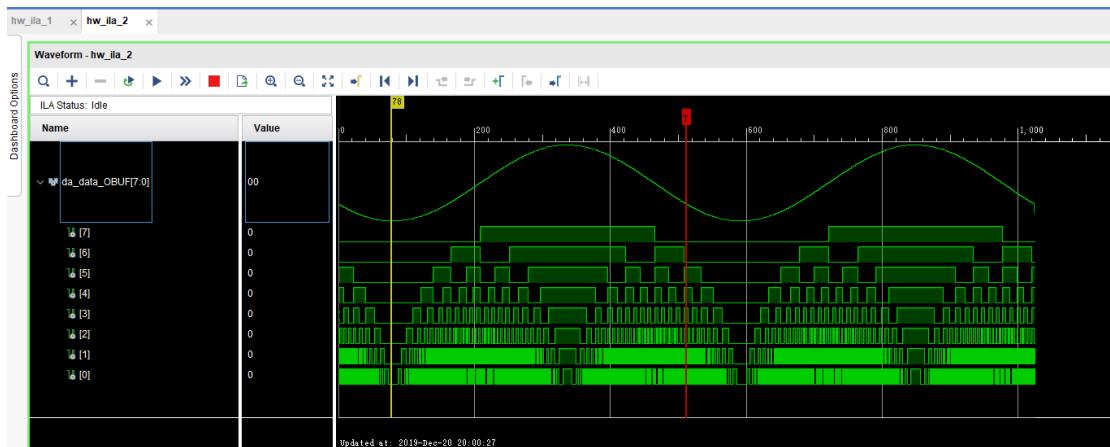
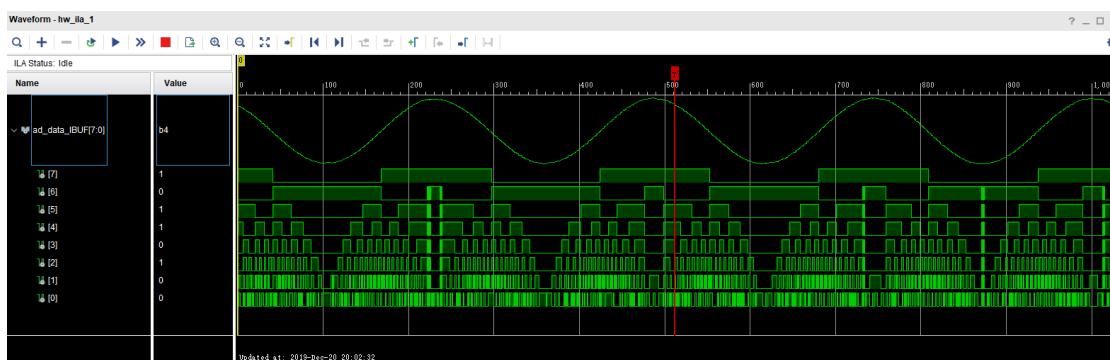


图 16- 22

点击触发，可以发现从 FPGA 输出 da\_data 的波形如下图所示，输出的波形为正弦波。



然后观察 ad 转换过后的 ad\_data,可以看到输入的模拟量已经被正确的接收。



至此完成了简单的 AD\_DA 转换。

## 14.8 实验总结

在本次实验中，使用 ADDA 子卡，完成了输出正弦波和对正弦波进行采样得到数字量的实验。

通过这次实验，粗略了解了以下 ADDA 子卡该如何使用，以及对 TI 的两款数模转换芯片有了一定的了解。

## 第15章 ADDA 转换与 HDMI 显示

### 15.1 实验简介

在前面的章节中，我们介绍了 ADDA 子卡的使用，并且通过 ILA 观察了 FPGA 内部输出和采样的到的数据，在本章中，我们将继续使用 ADDA 子卡，完成 ADDA 的转换，并将 AD 采样得到的数据，通过 HDMI 在显示器上显示出来。

### 15.2 创建工程

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 pcie_test, 工程路径 E:\demo\adda_hdmi 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件，购买 Z7-Lite7020 用户选中 xc7z020clg400-2，购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后，点击确认

### 15.3 功能设计

本次实验想要通过 HDMI 显示通过 AD 转换模块采集到的数据，主要的任务其实就转换为了，如何实现图像的叠加，首先我们先看以下本次实验的最终显示效果。

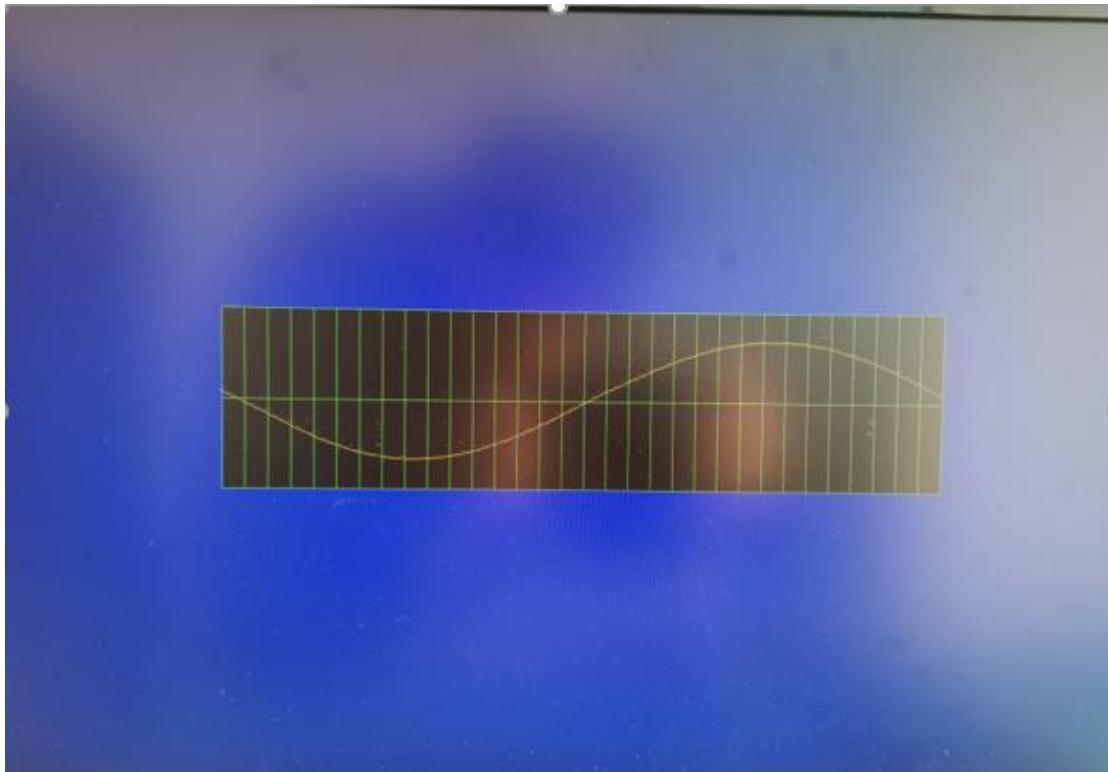


图 17- 1

本次实验将在 HDMI 的蓝色背景的中央区域显示一个 1024x256 大小的波形显示窗口，在该窗口中显示有栅格和黑色的背景，并且最终将我们采集得到的波形显示在该窗口中。

接下来我们来介绍以下，为了实现该功能，我们需要做哪些工作。

### 1) DA 波形发生模块

该模块的主要作用是将 FPGA 内部 rom 中的数据读出并将这些数据通过 DA 转换芯片输出到外部。该模块的示意图如下：

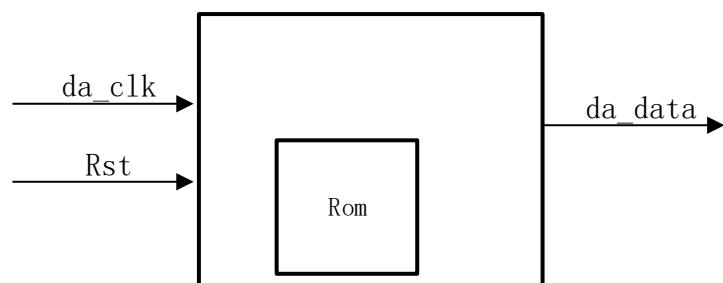


图 17- 2

在这里按照如下方法对 rom 进行初始化：

点击 IP catalog 进入 IP 生成页面：

在搜索栏中输入 rom，在下方选中 block Memory Generator

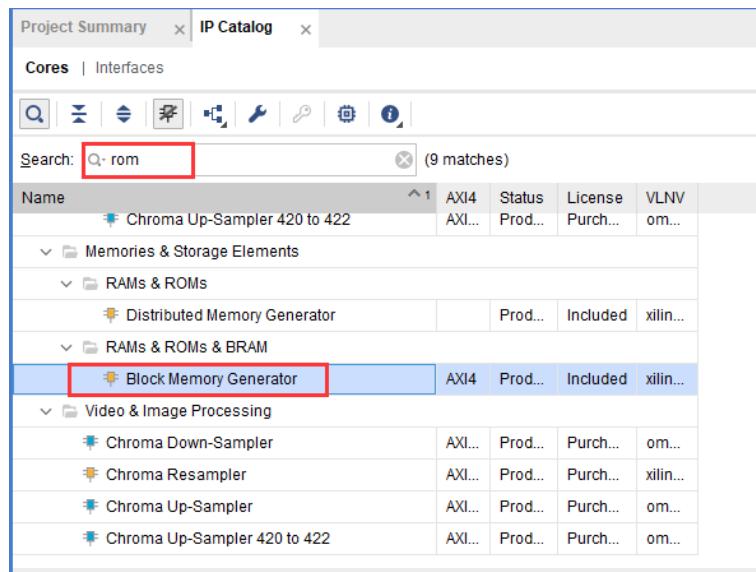


图 17- 3

在 basic 配置页面，对要生成的 ram 进行命名，选择单端口 rom；

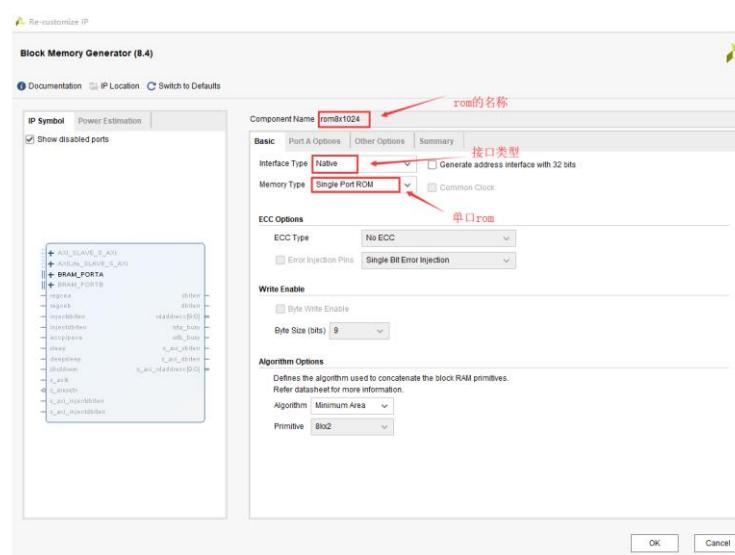


图 17- 4

在端口 A 的配置页面需要对 rom 的位宽和深度进行配置，配置成深度为 1024 位宽为 8 的模式。

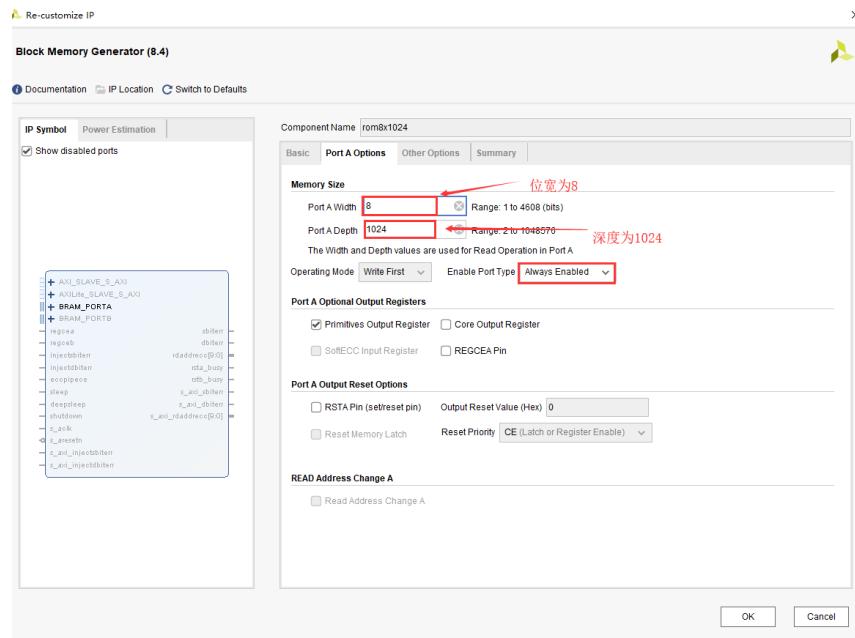


图 17- 5

接着添加导入初始化文件 coe 文件

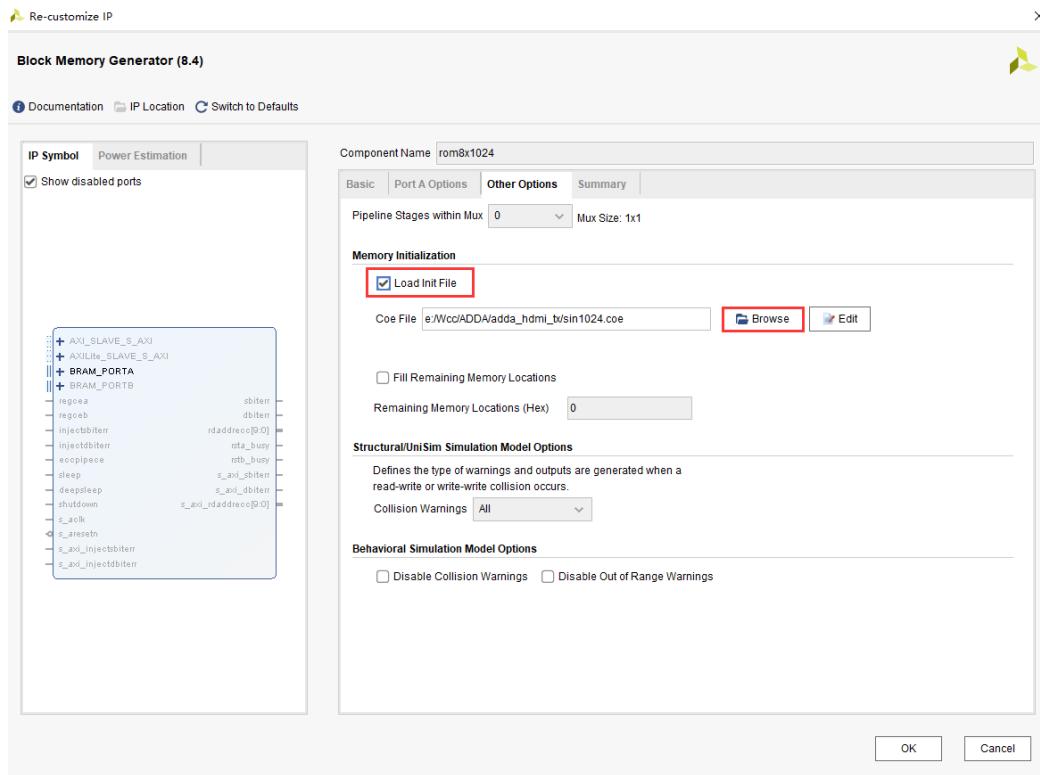


图 17- 6

选择工程目录下的 coe 文件。

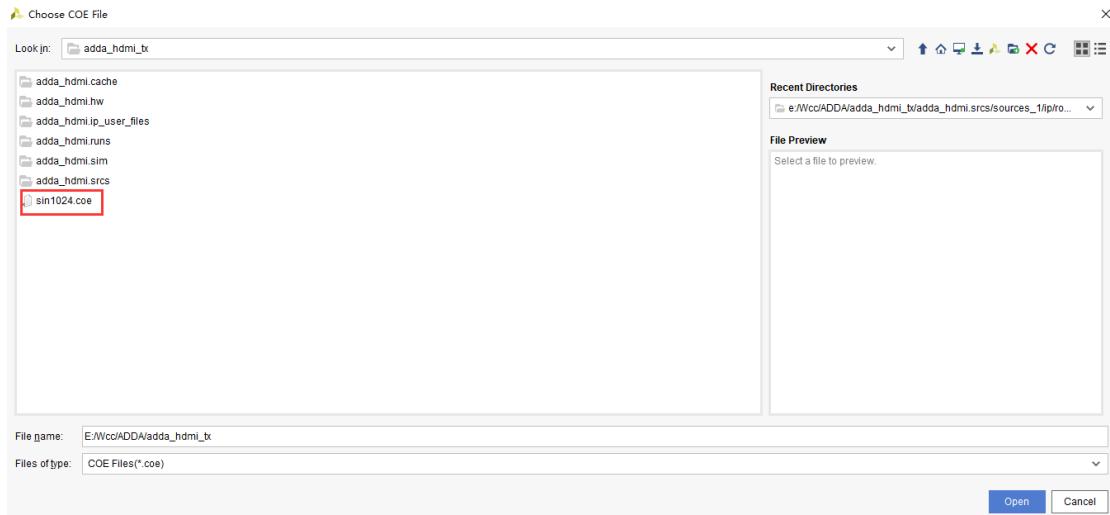


图 17- 7

## 2) 图像背景模块

该模块的主要作用是产生 HDMI 显示的图像背景，包括蓝色的背景和波形窗口的背景，边框和栅格等。该模块与前面在做 HDMI 输出方块那一章节中的内容十分相似。

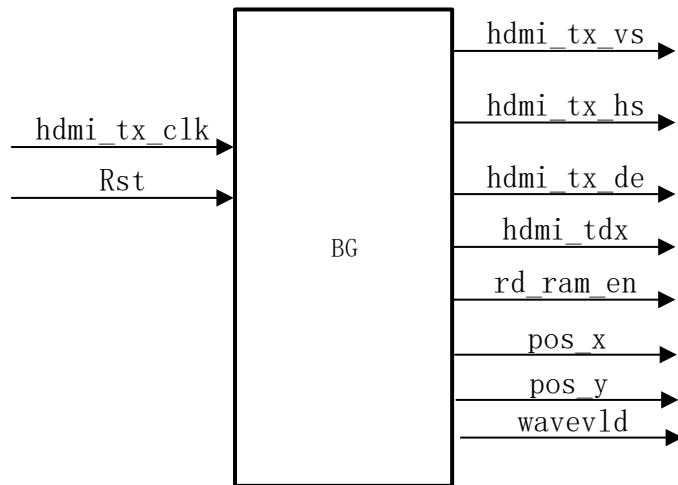


图 17-8

端口名称	I/O	位宽	备注
hdmi_tx_clk	I	1	像素时钟
rst	I	1	复位信号
hdmi_tx_vs	O	1	场同步信号
hdmi_tx_hs	O	1	行同步信号
hdmi_tx_de	O	1	输出数据有效信号
hdmi_tdx	O	24	输出的背景图像数据
rd_ram_en	O	1	读取 ram 中存储的 AD 数据使能
pos_x	O	13	波形窗口横坐标
pos_y	O	13	波形窗口纵坐标
wave_vld	O	1	波形窗口有效信号

本模块中，主要是通过行计数器和场计数器来控制要显示图像的窗口和窗口中要显示的栅格、边界等等。其时间简单来说就是调整图像输出的优先级，让图像能够出现在背景之上。如下边这段代码所示：

在波形窗口中即 `wave_vld` 有效时，若处于边界位置则输出绿色的边界，否则输出黑色的背景。在 `wave_vld` 有效区域之外的地方则显示蓝色的背景。可以看出来，实现这一部分的图像叠加还是比较简单的。

```
//hdmi_td
always @(posedge hdmi_tx_clk) begin
    if (rst==1'b1) begin
        hdmi_td <='d0;
    end
    else if(wave_vld)begin
        if(x==0 || y==0 || x==WINDOW_X-1 || y==WINDOW_Y-1 ||(y==(WINDOW_Y>>1)) || x[4:0]==5'h1f)
            hdmi_td <= 24'h00ff00;//输出图像
        else
            hdmi_td <='d0;
    end
    else if((cnt_h >= H_START) && (cnt_h < H_END) && (cnt_v >= V_START) && (cnt_v < V_END))begin
        hdmi_td <= 24'h0000ff;//输出图像
    end
    else begin
        hdmi_td <='d0;
    end
end
```

### 3) 波形叠加模块

在本模块中，主要功能是实现 AD 数据的采集，并将采集到的数据叠加到波形窗口中进行显示。可以看到该模块中有一部分用来完成 AD 数据的采集，采集到的数据存储到 RAM 中，例化的背景图像模块在本模块中会将 RAM 中的数据读出，并将读出的数据，与当前扫描到的波形窗口的点进行比较，以此来确定波形在窗口中显示的结果。

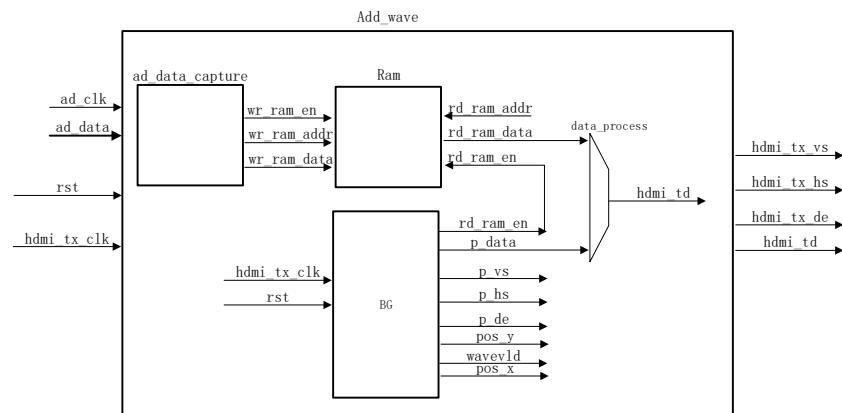


图 17-9

接口列表如下图所示：

端口名称	I/O	位宽	备注
hdmi_tx_clk	I	1	像素时钟
rst	I	1	复位信号
hdmi_tx_vs	O	1	场同步信号
hdmi_tx_hs	O	1	行同步信号
hdmi_tx_de	O	1	输出数据有效信号
hdmi_td	O	24	输出的背景图像数据
ad_clk	I	1	AD 采样时钟
ad_data	I	8	AD 采集到的数据

本模块中，为了使波形能够在窗口中显示，同样对要输出的图像做了优先级的处理。将背景图像模块中输出的数据，与本次模块中，是否使波形点的位置进行了叠加，最终能够将波形显示在前面设置的窗口中。

本模块的代码如下：

```

1 `timescale 1ns / 1ps
2 module wave_mix(
3     input      wire      ad_clk      ,
4     input      wire [7:0] ad_data      ,
5     input      wire      hdmi_tx_clk      ,
6     input      wire      rst      ,
7     output     wire      hdmi_tx_hs      ,
8     output     wire      hdmi_tx_vs      ,
9     output     wire      hdmi_tx_de      ,
10    output    wire [23:0]   hdmi_td
)

```

```
11      );
12 //=====
13 //parameter define
14 //=====
15 parameter IDLE = 3'b001;
16 parameter CAPTURE = 3'b010;
17 parameter WAIT = 3'b100;
18
19 parameter WAIT_MAX = 4000000;
20 parameter DATA_MAX = 1024 - 1;
21 parameter ADDR_MAX = 1024 - 1;
22 parameter AMP_DATA = 256 - 1;
23
24
25 //=====
26 //internal signals
27 //=====
28 reg [9:0] wr_ram_addr ;
29 wire wr_ram_en ;
30 wire rd_ram_en ;
31 reg rd_ram_en_dd ;
32 reg [9:0] rd_ram_addr ;
33 wire [12:0] pos_x ;//横坐标
34 wire [12:0] pos_y ;//纵坐标
35 wire wave_vld ;//波形窗口
36 reg [2:0] state ;
37
38 reg [11:0] cnt_data ;
39 wire add_cnt_data ;
40 wire end_cnt_data ;
41 wire [7:0] dout ;
42
43 reg [31:0] timer ;
44 reg [23:0] hdmi_td_r ;
45 wire [23:0] p_data ;
46 wire p_de ;
47 wire p_vs ;
48 wire p_hs ;
49
50
51
```

```
52 assign wr_ram_en = state==CAPTURE;
53 assign hdmi_td = hdmi_td_r ;
54 assign hdmi_tx_hs = p_hs;
55 assign hdmi_tx_de = p_de;
56 assign hdmi_tx_vs = p_vs;
57
58 //-----
59 //receive port
60 //-----
61
62 //-----state machine describe-----
63 always @(posedge ad_clk) begin
64   if (rst==1'b1) begin
65     state <= IDLE;
66   end
67   else begin
68     case(state)
69       IDLE:begin
70         state <= CAPTURE;
71       end
72
73       CAPTURE:begin
74         if(end_cnt_data)
75           state <= WAIT;
76         else
77           state <= CAPTURE;
78       end
79
80       WAIT:begin
81         if(timer==WAIT_MAX)
82           state <= CAPTURE;
83         else
84           state <= WAIT;
85       end
86
87       default:begin
88         state <= IDLE;
89       end
90     endcase
91   end
92 end
```

```
93
94 //-----cnt_data-----
95 always @(posedge ad_clk)begin
96     if(rst==1'b1)begin
97         cnt_data <= 0;
98     end
99     else if(add_cnt_data)begin
100        if(end_cnt_data)
101            cnt_data <= 0;
102        else
103            cnt_data <= cnt_data + 1'b1;
104    end
105    else begin
106        cnt_data <= 'd0;
107    end
108 end
109
110 assign add_cnt_data = state==CAPTURE;
111 assign end_cnt_data = add_cnt_data && cnt_data== DATA_MAX;
112
113 //-----timer-----
114 always @(posedge ad_clk)begin
115     if(rst == 1'b1)begin
116         timer <= 1'b0;
117     end
118     else if(state==WAIT)begin
119         if(timer==WAIT_MAX)
120             timer <= 'd0;
121         else
122             timer <= timer + 1'b1;
123     end
124     else begin
125         timer <= 'd0;
126     end
127 end
128
129 //-----wr_ram_addr-----
130 always @(posedge ad_clk)begin
131     if(rst == 1'b1)begin
132         wr_ram_addr <= 'd0;
133     end
```

```
134     else if(wr_ram_en==1'b1)begin
135         if(wr_ram_addr==ADDR_MAX)
136             wr_ram_addr <= 'd0;
137         else
138             wr_ram_addr <= wr_ram_addr + 1'b1;
139     end
140     else begin
141         wr_ram_addr <= 'd0;
142     end
143 end
144
145 dpram_8x1024 inst_ram (
146     .clka(ad_clk),           // input wire clka
147     .wea(wr_ram_en),        // input wire [0 : 0] wea
148     .addr(wr_ram_addr),// input wire [9 : 0] addra
149     .dina(ad_data),        // input wire [7 : 0] dina
150     .clkb(hdmi_tx_clk), // input wire clkb
151     .addrb(rd_ram_addr),// input wire [9 : 0] addrb
152     .doutb(dout)          // output wire [7 : 0] doutb
153 );
154
155 hdmi_background inst_hdmi_background (
156     .rst      (rst),
157     .hdmi_tx_clk (hdmi_tx_clk),
158     .hdmi_tx_de  (p_de),
159     .hdmi_tx_hs  (p_hs),
160     .hdmi_tx_vs  (p_vs),
161     .hdmi_td     (p_data),
162     .pos_x      (pos_x),
163     .pos_y      (pos_y),
164     .rd_ram_en   (rd_ram_en),
165     .wave_vld    (wave_vld)
166 );
167
168 //-----
169 //transfer port
170 //-----
171 //-----rd_ram_addr-----
172 always @ (posedge hdmi_tx_clk)begin
173     if(rst == 1'b1)begin
174         rd_ram_addr <= 'd0;
```

```
175    end
176    else if(rd_ram_en==1'b1)begin
177        if(rd_ram_addr==ADDR_MAX)
178            rd_ram_addr <= 'd0;
179        else
180            rd_ram_addr <= rd_ram_addr + 1'b1;
181    end
182    else begin
183        rd_ram_addr <= 'd0;
184    end
185 end
186
187 always @(posedge hdmi_tx_clk)begin
188     rd_ram_en_dd <= rd_ram_en;
189 end
190
191 always @(posedge hdmi_tx_clk)begin
192     if(rst == 1'b1)begin
193         hdmi_td_r <= 'd0;
194     end
195     else if(rd_ram_en_dd==1'b1)begin
196         if(AMP_DATA-pos_y == {4'd0,dout})//当前位置是否处于波形上
197             hdmi_td_r <= 24'hFFFF00;
198         else
199             hdmi_td_r <= p_data;
200     end
201     else begin
202         hdmi_td_r <= p_data;
203     end
204 end
205
206 endmodule
```

其中值得注意的代码片段是 191 行到 203 行，该段代码完成了将背景图像模块的数据和 AD 波形叠加显示的功能。

关于本次工程中的其他模块，还请查看本次工程的其他源码，这些模块在前的章节中已经有过介绍，在本章中不再赘述。

## 15.4 添加约束文件

本次实验的约束文件，就是在 ADDA 转换实验章节的基础上，添加上 HDMI 显示模块的约束即可。约束文件如下：

```
##### clock define#####
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

##### key define#####
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]

set_property PACKAGE_PIN U18 [get_ports hdmi_tx_clk_p]
set_property PACKAGE_PIN N20 [get_ports {hdmi_tx_chn_b_p}]
set_property PACKAGE_PIN T20 [get_ports {hdmi_tx_chn_g_p}]
set_property PACKAGE_PIN V20 [get_ports {hdmi_tx_chn_r_p}]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_r_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_g_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_b_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_clk_p]

set_property PACKAGE_PIN W19 [get_ports {da_data[0]}]
set_property PACKAGE_PIN W18 [get_ports {da_data[1]}]
set_property PACKAGE_PIN U17 [get_ports {da_data[2]}]
set_property PACKAGE_PIN T16 [get_ports {da_data[3]}]
set_property PACKAGE_PIN R17 [get_ports {da_data[4]}]
set_property PACKAGE_PIN R16 [get_ports {da_data[5]}]
set_property PACKAGE_PIN P18 [get_ports {da_data[6]}]
set_property PACKAGE_PIN N17 [get_ports {da_data[7]}]
set_property PACKAGE_PIN Y18 [get_ports da_clk]

set_property SLEW FAST [get_ports {da_data[*]}]
set_property SLEW FAST [get_ports da_clk]

set_property IOSTANDARD LVCMOS33 [get_ports {da_data[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports da_clk]
```

```
set_property PACKAGE_PIN U12 [get_ports ad_clk]
set_property PACKAGE_PIN V13 [get_ports {ad_data[0]}]
set_property PACKAGE_PIN U13 [get_ports {ad_data[1]}]
set_property PACKAGE_PIN U15 [get_ports {ad_data[2]}]
set_property PACKAGE_PIN U14 [get_ports {ad_data[3]}]
set_property PACKAGE_PIN R14 [get_ports {ad_data[4]}]
set_property PACKAGE_PIN P14 [get_ports {ad_data[5]}]
set_property PACKAGE_PIN W15 [get_ports {ad_data[6]}]
set_property PACKAGE_PIN V15 [get_ports {ad_data[7]}]

set_property SLEW FAST [get_ports ad_clk]

set_property IOSTANDARD LVCMOS33 [get_ports ad_clk]
set_property IOSTANDARD LVCMOS33 [get_ports {ad_data[*]}]
```

## 15.5 下载验证

接下来就可以生成 bit 文件进行下载验证了。生 bit 文件后按照如下图所示连接好开发板。

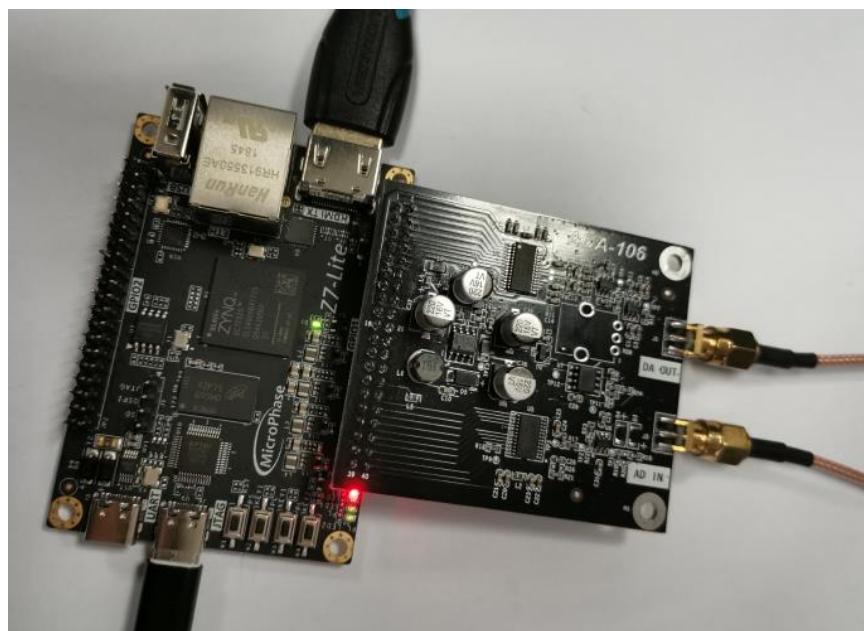


图 17- 10

接下来将 bit 文件下载到 FPGA 中，过一会即可看到在显示器上出现了采集到的波形。

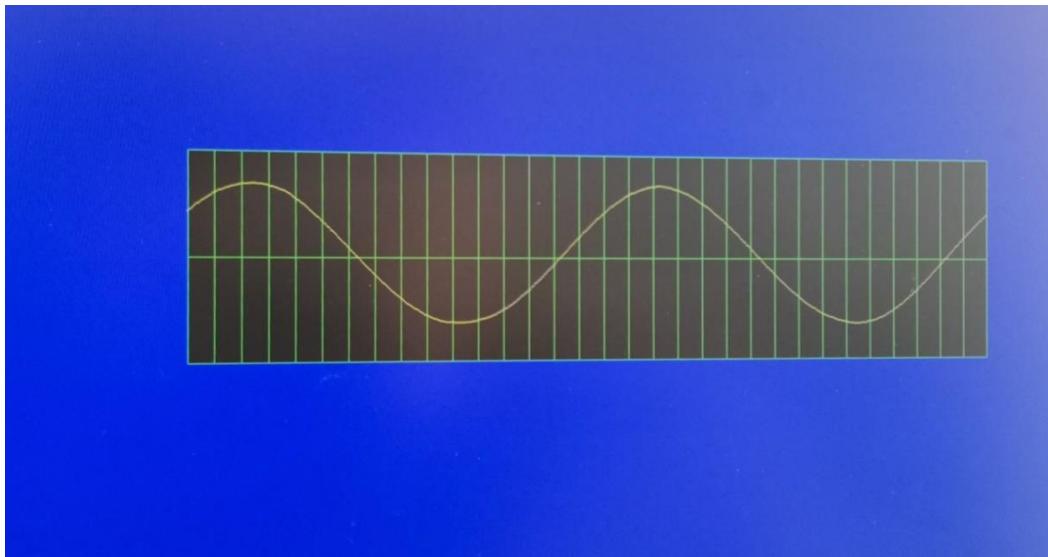


图 17-11

至此我们完成了，使用 ADDA 子卡，在屏幕上显示正弦波的实验，使用同样的方法，只需更改初始化文件 coe 文件即可更采样的波形，同样，调整我们采样的频率也能得到不同的波形。在本章中不再进行介绍，有兴趣的可以自己去设计几个简单的波形显示器。

## 15.6 实验总结

在本次实验中，通过 ADDA 子卡完成了在显示器上显示采集到的波形，本次实验主要就是完成图像的叠加，简单来说图像的叠加就是改变想要显示的图像的优先级，根据优先级将图像叠加到背景之中。

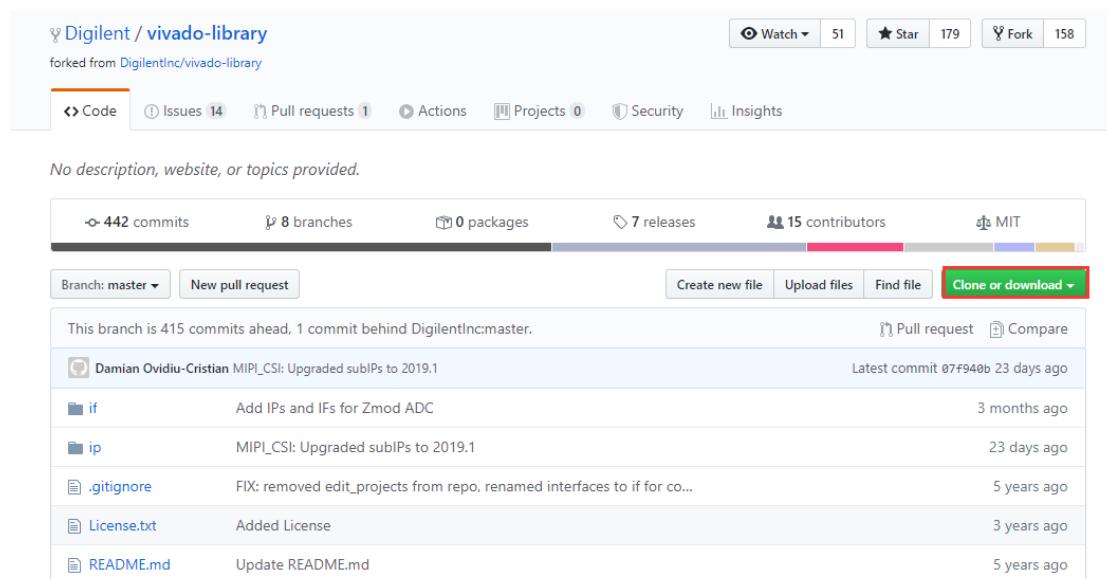
## 第16章 使用第三方 IP 完成 HDMI 输出

### 16.1 实验背景

在 Z7-Lite 开发板上有 1 个 HDMI 接口，都可以作为 HDMI 的输入或者输出，本次实验将使用第三方 IP 完成 HDMI 的输出。本次实验只需使用到开源的 IP 即可。

### 16.2 下载第三方 IP

本次实验需要使用到第三方的 IP，在 `github` 上可以找到，以下是 `github` 链接：<https://github.com/Digilent/vivado-library>



No description, website, or topics provided.

Branch: master	New pull request	Create new file	Upload files	Find file	Clone or download
This branch is 415 commits ahead, 1 commit behind DigilentInc:master.					
Damian Ovidiu-Cristian MIPI_CSI: Upgraded subIPs to 2019.1					
if	Add IPs and IFs for Zmod ADC	3 months ago			
ip	MIPI_CSI: Upgraded subIPs to 2019.1	23 days ago			
.gitignore	FIX: removed edit_projects from repo, renamed interfaces to if for co...	5 years ago			
License.txt	Added License	3 years ago			
README.md	Update README.md	5 years ago			

图 18- 1

点击 `clone` 即可将其他用户仓库中的文件下载到本地。下载到本地后，可以将第三方 IP 添加到 `Vivado` 工程中，然后用户就可以使用第三方的 IP 了。

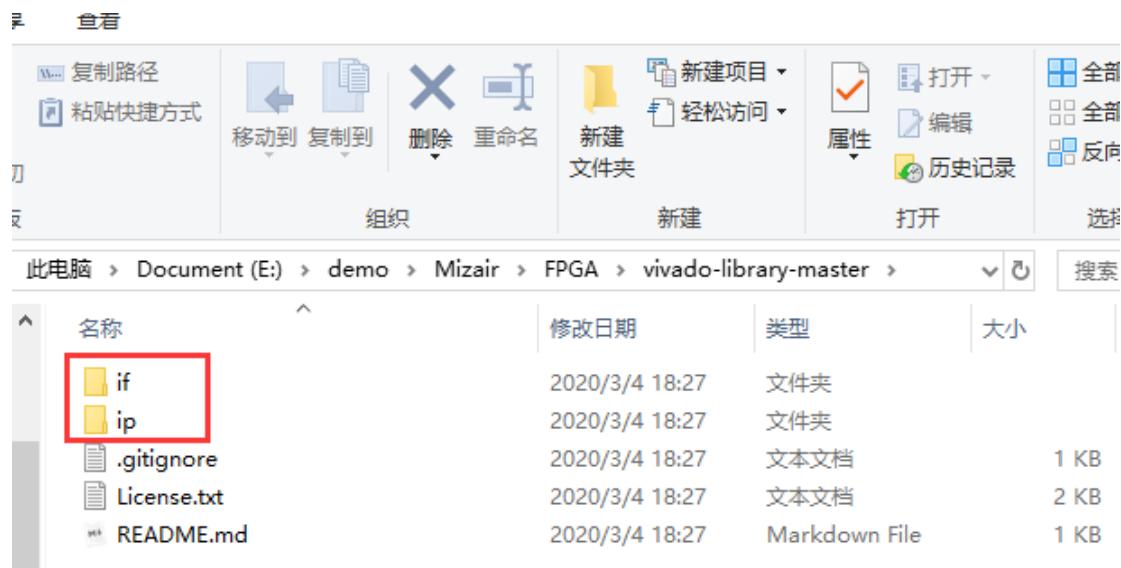


图 18- 2

下载完成后，可以看到其中有两个文件夹，其中 ip 文件夹用来存放 IP，if 文件夹用来存放 interface。本次实验需要使用到 ip 文件夹中的 dvi2rgb 和 rgb2dvi 这两个 IP，还需要使用到 if 文件夹中的 tmds 接口定义。

在工程目录下新建一个 ip\_repo 文件夹用来存放第三方 IP，将前面提到的所需要的 IP 赋值到该文件夹下。

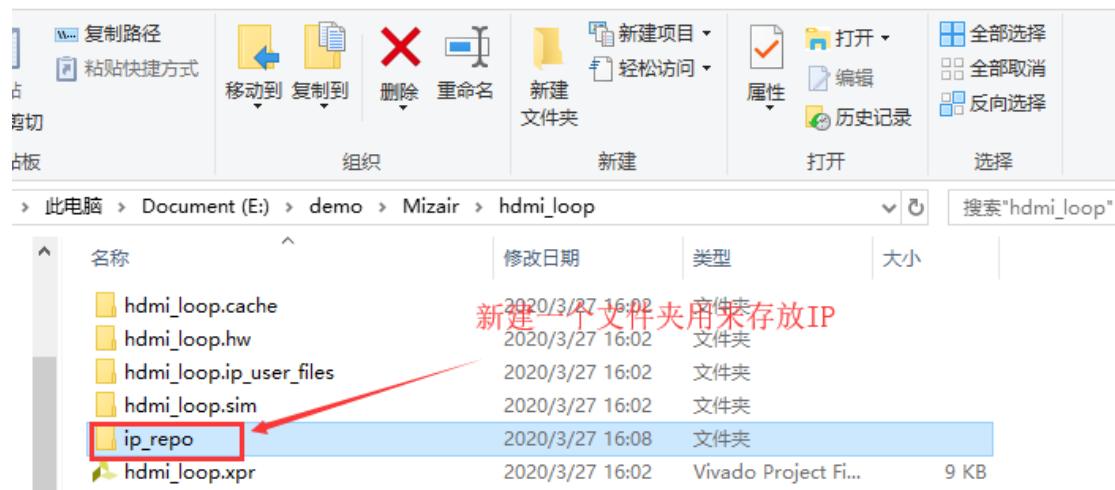


图 18- 3

复制所需的 IP 文件夹到 ip\_repo

名称	修改日期	类型
dvi2rgb	2020/3/4 18:27	文件夹
hls_contrast_stretch_1_0	2020/3/4 18:27	文件夹
hls_gamma_correction_1_0	2020/3/4 18:27	文件夹
hls_saturation_enhance_1_0	2020/3/4 18:27	文件夹
MIPI_CSI_2_RX	2020/3/4 18:27	文件夹
MIPI_D_PHY_RX	2020/3/4 18:27	文件夹
MotorFeedback_1_0	2020/3/4 18:27	文件夹
Pmods	2020/3/4 18:27	文件夹
PWM_1.0	2020/3/4 18:27	文件夹
PWM_2.0	2020/3/4 18:27	文件夹
PWM_Analyzer_1.0	2020/3/4 18:27	文件夹
rgb2dpvid_v1_0	2020/3/4 18:27	文件夹
rgb2dvi	2020/3/4 18:27	文件夹
rgb2vga_v1_0	2020/3/4 18:27	文件夹
Sync_v1_0	2020/3/4 18:27	文件夹
usb2device_v1_0	2020/3/4 18:27	文件夹
video_scaler	2020/3/4 18:27	文件夹
Zmods	2020/3/4 18:27	文件夹

复制IP文件夹

图 18- 4

复制所需的 if 文件夹到 ip\_repo

### 16.3 创建工程

起始页	Create New Project
向导起始页	Next
Project Name	工程名为 pcie_test, 工程路径 E:\demo\hdmi_loop 勾选 Createproject subdirectory, Next
Project Type	RTL Project, Next
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	选择器件, 购买 Z7-Lite7020 用户选中 xc7z020clg400-2, 购买 Z7-Lite7010 用户选中 xc7z010clg400-1
New Project summary	确认无误后, 点击确认

## 16.4 添加第三方 IP 到 IP 仓库中

创建完工程后，首先需要将第三方的 IP 添加到 IP 仓库中，点击 settings → 选择 IP repository → 点击“+”号选择 ip\_repo 的路径。

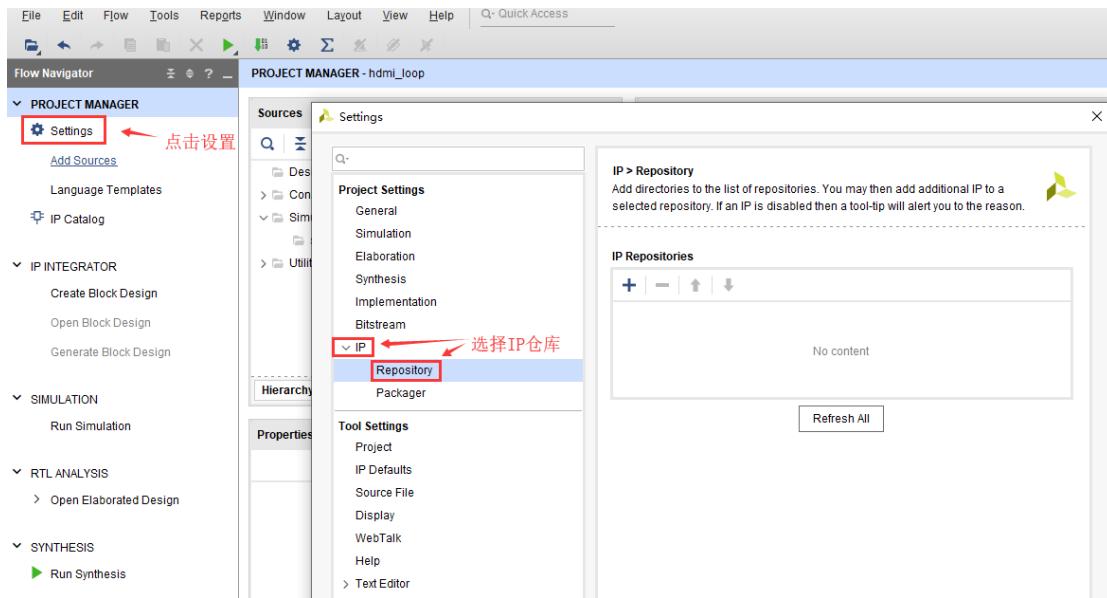


图 18-5

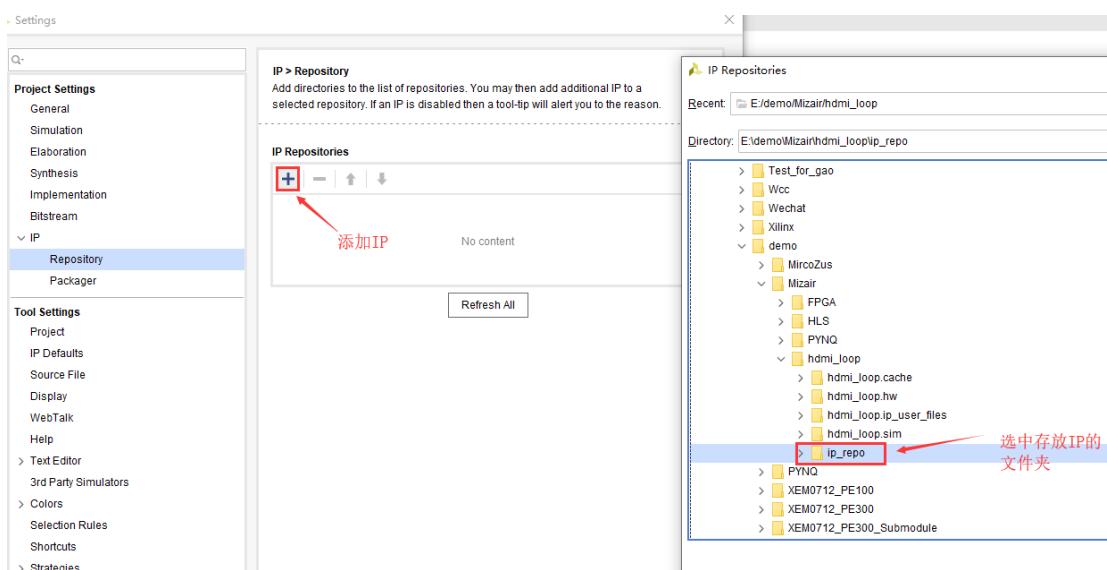


图 18-6

选中 IP 存放的路径后，Vivado 工具会识别处其中的自定义 IP 和自定义接口并将其添加到 IP 仓库中。最后点击 apply → OK 完成添加 IP 到 IP 仓库中。

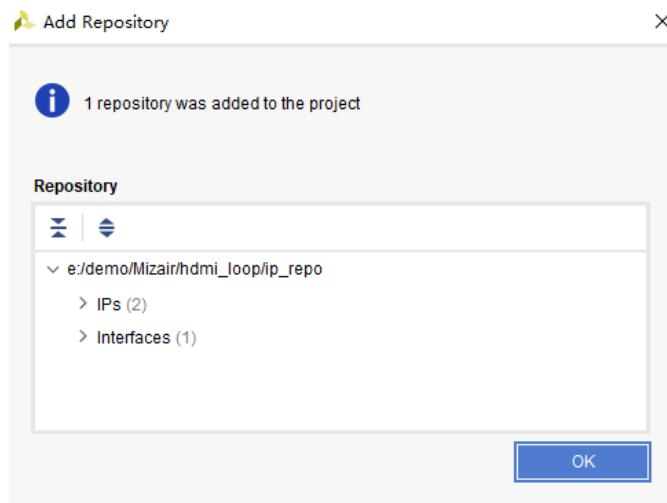


图 18- 7

## 16.5 生成第三方 IP

点击 IP Catalog → 选择 User Repository → RGB to DVI Encoder → 双击进行配置。

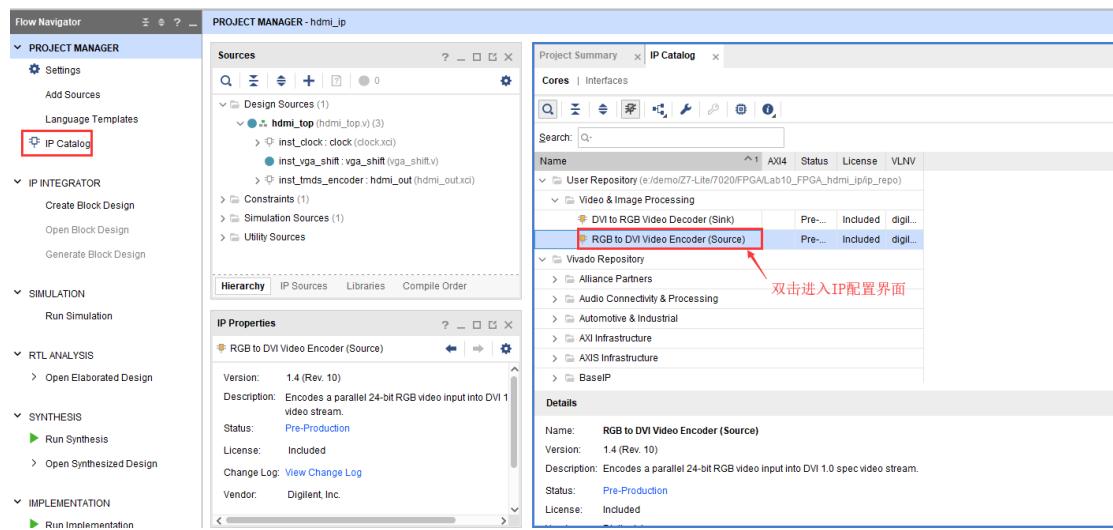


图 18- 8

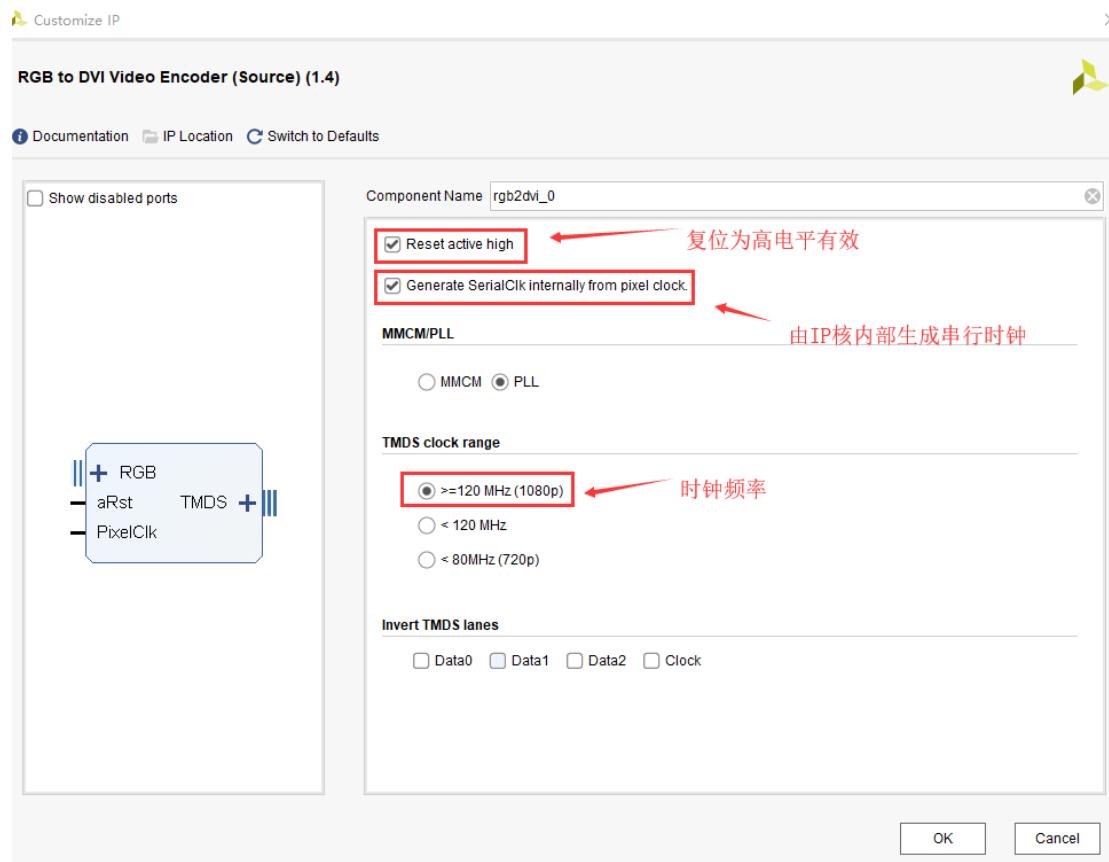


图 18- 9 IP 核配置

```

51 // The following must be inserted into your Verilog file for this
52 // core to be instantiated. Change the instance name and port connections
53 // (in parentheses) to your own signal names.
54
55
56 //----- Begin Cut here for INSTANTIATION Template ---// INST_TAG
57 hdmi_out your_instance_name (
58     .TMDS_Clk_p(TMDS_Clk_p),           // output wire TMDS_Clk_p
59     .TMDS_Clk_n(TMDS_Clk_n),           // output wire TMDS_Clk_n
60     .TMDS_Data_p(TMDS_Data_p),         // output wire [2 : 0] TMDS_Data_p
61     .TMDS_Data_n(TMDS_Data_n),         // output wire [2 : 0] TMDS_Data_n
62     .aRst(aRst),                     // input wire aRst
63     .vid_pData(vid_pData),            // input wire [23 : 0] vid_pData
64     .vid_pVDE(vid_pVDE),              // input wire vid_pVDE
65     .vid_pHSync(vid_pHSync),          // input wire vid_pHSync
66     .vid_pVSync(vid_pVSync),          // input wire vid_pVSync
67     .PixelClk(PixelClk)             // input wire PixelClk
68 );
69 // INST_TAG_END ----- End INSTANTIATION Template -----
70
71

```

图 18- 10 IP 例化模板

## 16.6 程序设计

本次实验的程序设计，在前面第 14，15 章中已经介绍过，本次实验中，只需将前面用户自己设计的 TMDS 编码的模块，替换成本次实验中的第三方 IP 即可。

其中底层模块如下：

只需要将 IP 例化到工程中，替换掉前面由用户自己实现的编码，串行转换模块。

```
1 `timescale 1ns / 1ps
2
3
4 module hdmi_top(
5   input   wire      clk      ,
6   input   wire      rst_n    ,
7   output  wire      hdmi_tx_clk_p  ,
8   output  wire      hdmi_tx_clk_n  ,
9   output  wire [2:0]hdmi_tx_data_p  ,
10  output  wire [2:0]hdmi_tx_data_n
11 );
12
13 wire      locked      ;
14 wire      pixel_clk   ;
15 wire      rst         ;
16
17 wire      vpg_hs     ;
18 wire      vpg_vs     ;
19 wire      vpg_de     ;
20 wire [23:0] rgb        ;
21
22 assign rst = ~locked;
23 clock inst_clock(
24   .clk_out1(pixel_clk),
25   .reset(~rst_n),
26   .locked(locked),
27   .clk_in1(clk)
28 );
```

```

29
30 vga_shift inst_vga_shift (
31     .rst      (rst),
32     .vpg_pclk (pixel_clk),
33     .vpg_de   (vpg_de),
34     .vpg_hs   (vpg_hs),
35     .vpg_vs   (vpg_vs),
36     .rgb      (rgb)
37 );
38
39 hdmi_out inst_tmds_encoder (
40     .TMDS_Clk_p(hdmi_tx_clk_p),    // output wire TMDS_Clk_p
41     .TMDS_Clk_n(hdmi_tx_clk_n),    // output wire TMDS_Clk_n
42     .TMDS_Data_p(hdmi_tx_data_p),  // output wire [2 : 0] TMDS_Data_p
43     .TMDS_Data_n(hdmi_tx_data_n),  // output wire [2 : 0] TMDS_Data_n
44     .aRst(rst),                  // input wire aRst
45     .vid_pData(rgb),            // input wire [23 : 0] vid_pData
46     .vid_pVDE(vpg_de),          // input wire vid_pVDE
47     .vid_pHSync(vpg_hs),        // input wire vid_pHSync
48     .vid_pVSync(vpg_vs),        // input wire vid_pVSync
49     .PixelClk(pixel_clk)       // input wire PixelClk
50 );
51
52
53 endmodule

```

## 16.7 添加约束文件

```

##### clock define#####
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

##### key define#####
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property PACKAGE_PIN U18 [get_ports hdmi_tx_clk_p]
set_property PACKAGE_PIN N20 [get_ports {hdmi_tx_data_p[0]}]
set_property PACKAGE_PIN T20 [get_ports {hdmi_tx_data_p[1]}]

```

```
set_property PACKAGE_PIN V20 [get_ports {hdmi_tx_data_p[2]}]
set_property IOSTANDARD TMDS_33 [get_ports {hdmi_tx_data_p[*]}]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_clk_p]
set_property CLOCK_DEDICATED_ROUTE_BACKBONE [get_nets inst_clock/inst/clk_in1_clock]
```

## 16.8 实验现象

本次实验的实验结果和前面第 14 章的实验结果是一致的，会在显示器上显示一个移动的方块。

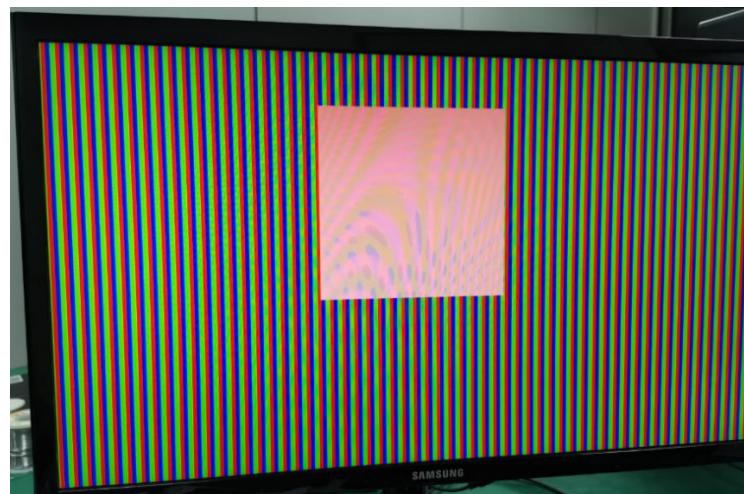


图 18-11

## 16.9 实验总结

在本次实验中，通过使用开源的 IP，来完成 HDMI 的显示，在本次实验中介绍了在 Vivado 中添加第三方的 IP，并且在工程中将其例化。用户可以通过这种方式来添加自定义的 IP 或者第三方 IP，能够加快开发的速率。

## 第一版本小结

感谢您选择微相科技（上海）有限公司 Z7-Lite ZYNQ7000 系列产品，通过前面的内容，相信您对我们的开发板已经比较熟悉了，也能够熟练的运用起板上 PL（可编程逻辑）部分的资源，PL 端的开发与传统的 FPGA 开发流程一致，在之后的内容中，将会去了解到 PS 端的开发，那时候才能真正体现 ZYNQ 系列的优势。由于本人水平有限，笔者不能保证文中的内容全都正确，若您发现了其中的错误，请您与我们联系，我们会尽快更改。暂时能够提供的例程还十分有限，但请给予我们多一点时间，我们会努力丰富和完善本开发板的相关资源。

本文档的内容，作为我的初次编写的内容，其中出现的错误，还请大家包涵与指正。我们会尽快修改，本文档会随着时间推移逐渐丰富，而不是一成不变的，还请您继续关注。

MircoPhase