Z7-Lite 系列教程
之 HLS 篇

Xilinx ZYNQ-7000 SoC

Rev. 1.0

第 I 条 版本记录

版本	时间	描述
Rev. 1.0	2020.7.28	第一版发布

第 II 条版权声明：

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。

第 III 条　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。

第 I 条　　版本记录 2
第 II 条　　版权声明： 3
第 III 条　　免责声明： 3
第 1 章 HLS 简介 7
1.1 HLS 简介 7
1.2 Vivado HLS 开发流程简介 8
第 2 章 HLS 开发流程体验 11
2.1 使用 HLS 完成 led 灯闪烁 11
2.2 创建 Vivado 工程并导入 HLS 生成的 IP 核 33
2.3 将 HLS 产生的 IP 添加到工程当中 36
2.4 下载验证 40
2.5 实验总结 41
第 3 章 搭建 HLS 图像仿真平台 43
3.1 OpenCv 与 HLS 中的 OpenCv 库简介 43
3.2 常用的 OpenCv 中的结构与函数 44
3.3 HLS 中可综合的图像处理模块 49
3.4 灰度转换实例 51
3.5 实验总结 66
第 4 章 HLS 中的浮点运算与 BRAM 接口 68
4.1 HLS 中实现浮点运算算法核的实现 68
4.2 搭建测试硬件平台 75
4.3 创建 SDK 工程 84
4.4 实验总结 91
第 5 章 HLS 中实现 AXI-Stream 接口 92
5.1 HLS 中实现 AXI-stream 的简单接口 92
5.2 搭建硬件测试平台 96

5.3 创建 SDK 工程 100
5.4 实验总结 104
第 6 章 图像直方图 105
6.1 实验简介 105
6.2 图像直方图的 HLS 实现 107
6.3 搭建测试硬件平台 113
6.4 创建 SDK 工程 114
6.5 实验总结 120
第 7 章 图像直方图拉伸 121
7.1 实验简介 121
7.2 直方图拉伸的 HLS 实现 123
7.3 搭建硬件平台 127
7.4 创建 SDK 工程 130
7.5 实验总结 131
第 8 章 HLS 实现 2D Convolution(二维卷积) 132
8.1 实验简介: 132
8.2 图像腐蚀、膨胀的简单介绍 134
8.3 Sobel 边缘检测 136
8.4 HLS 视频库中的 Linebuffer 和 Window 类 138
8.5 2D convolution 的 HLS 实现 141
8.6 实验总结 151
第 9 章 图像腐蚀、膨胀、边缘检测综合实验 152
9.1 实验简介 152
9.2 搭建 Block Design 152
9.3 创建 SDK 工程 163
9.4 实验总结 169

第 10 章 OV5640 的 sobel 边缘检测 170
10.1 实验简介 170
10.2 OV5640 硬件介绍 170
10.3 HLS 中完成二维卷积的设计 179
10.4 搭建 Block Design 183
10.5 创建 SDK 工程 186
10.6 实验总结 192
第 11 章 快速角点检测 193
11.1 实验背景 193
11.2 HLS 实现角点检测 195
11.3 HLS 仿真与综合 201
11.4 搭建 Block Design 203
11.5 下载验证 204
11.6 实验总结 206

第1章　HLS 简介

1.1 HLS 简介

Vivado HLS(High Level Synthesis)是赛灵思提供给开发者的一款高层次综合工具，使用 HLS 能够加快我们开发的效率。开发者通过使用 HLS 能够将 C/C++ 或者 system C 等高级语言转化为 RTL 电路，这在处理某些较为复杂的算法的时候，能够为开发者提供极大的便利。

Vivado HLS 给开发者提供了常见的库，例如图像处理相关的 OpenCv 库和其它的数学库，开发者通过调用这些库中的函数，能够加快开发的进度。

Vivado 的设计是以 IP 为核心的，HLS 就相当于一个可以用于创建 IP 的工具，通过 HLS 创建的 IP 可以应用到 Vivado 中，通过例化或者使用 BlockDesign 的方式，将通过 HLS 创建的 IP 应用到自己的项目中去。

Xilinx 官方也为我们提供了 HLS 使用教程文档，感兴趣的可以自己去下载，在我们的开发包中也会提供这些文档。用户可以到赛灵思官网下载 UG902，UG871，UG1233。

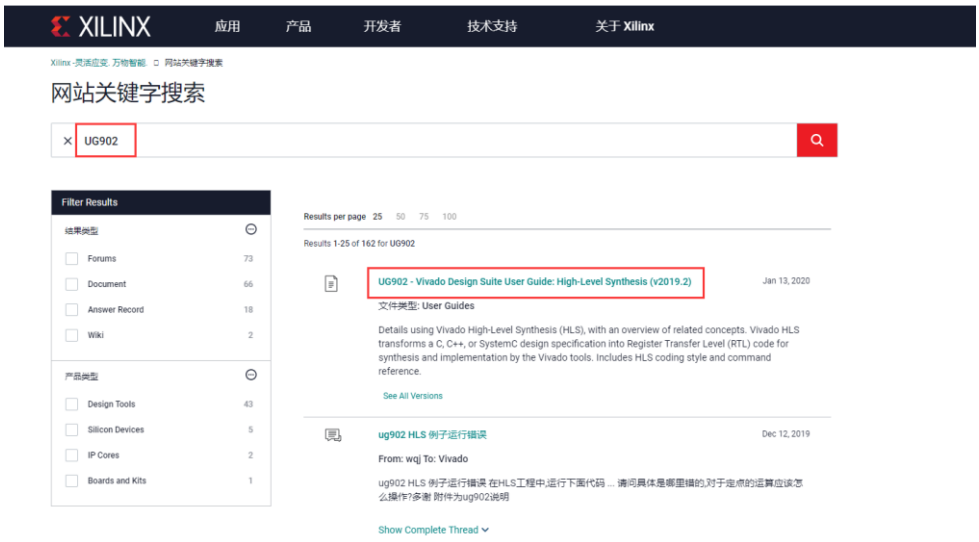


图 1- 1

在这里我们提供的 HLS2016.1 的版本的下载链接，赛灵思最新的 UG902 版本已经将其中的 HLS video 相关章节移动至新的文档当中，较 2016.1 的版本有了较大的变化，这里还是推荐使用这个 2016 版本的 user guide，里面有关于 HLS 图像处理函数的较为详细的说明，并且不会对我们使用的 Vivado2018.3 的版本有太大影响。

1) UG902

https://china.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug902-vivado-high-level-synthesis.pdf

最新版的有关 HLS 图像处理的参考文档 UG1233 连接如下

2) UG1233

https://china.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1233-xilinx-opencv-user-guide.pdf

关于 Vivado HLS 的使用方法的参考文档 UG871 的链接如下

3) UG871

https://china.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug871-vivado-high-level-synthesis-tutorial.pdf

1.2 Vivado HLS 开发流程简介

使用 HLS 开发的流程如下图所示：

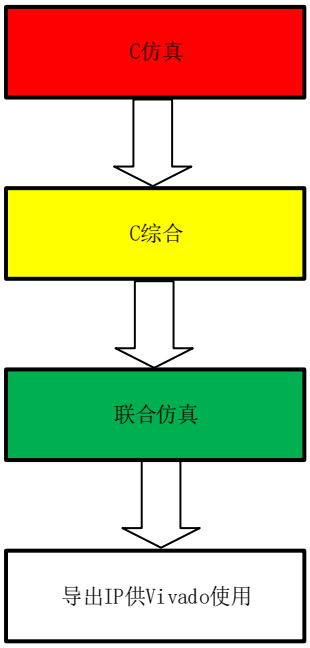


图 1- 2

首先需要进行 C/C++层面的仿真。做过 FPGA 开发的都知道，在 FPGA 开发过程中，会使用到 Testbench 来验证我们设计的模块的功能。在 HLS 的开发中，也同样需要进行仿真。在 C 仿真阶段，我们需要在源文件中，添加一个顶层函数，这个函数就是我们想要将来映射到 RTL 电路中的函数，有了这个函数后，我们需要一个 C Testbench 来对这个函数功能进行验证，在算法层面，检验我们的函数是否能够正常工作。算法层面的仿真，能够很快地就得出结果，有助于提高我们的开发效率。

在接下来我们需要对 C 代码进行综合，综合后会根据我们的功能函数，产生相应的电路。在 C 综合阶段，HLS 会根据我们对功能函数中的一些约束，来生成不同的接口，这些约束在将来我们会讲到，在 HLS 这些约束被称为 Directive。

C 综合完成之后，需要进行 C/RTL 的联合仿真，在这一阶段，HLS 会根据我们的 C Testbench 来生成我们的 RTL 的 Testbench 并且根据我们所选择的仿真工具来进行 RTL 级的仿真。仿真完成后我们可以观察联合仿真所产生的波形。若 RTL 仿真的结果正确，那么就可以进行下一步的操作。

前面有提到过 HLS 相当于一个 IP 生成器，它能够将我们的高级语言的代码

映射为一个 IP，我们可以根据需要将这些 IP 导出到 Vivado 的集成开发环境中，将这些算法的 IP 应用到实际的工程当中。

以上是对 HLS 开发的整个开发流程的概述，其中还设计到关于函数接口和优化内容，在之后的章节中进行介绍。下面我们以一个 led 灯闪烁的例子，来演示整个 HLS 开发的流程。

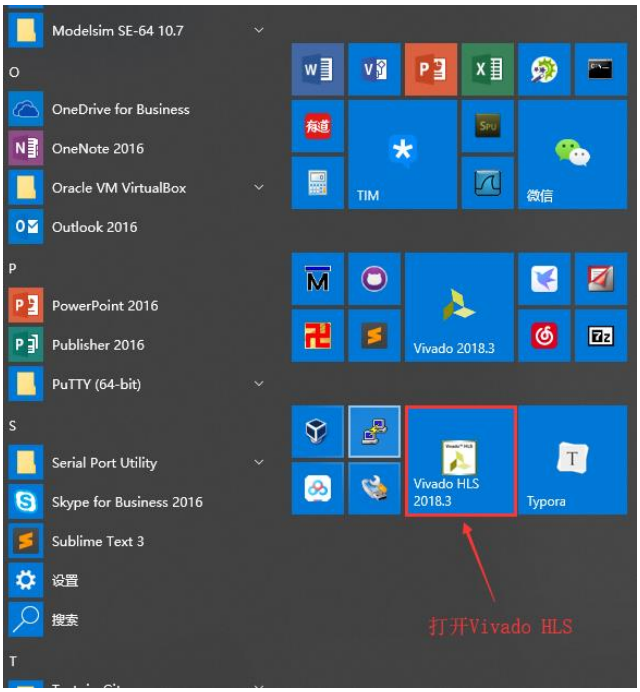
第2章　HLS 开发流程体验

2.1 使用 HLS 完成 led 灯闪烁

就像在软件开发中一样，学习一门新的语言，新的开发环境，总免不了写一个输出 Hello world！的程序一样，在 FPGA 开发中，适应一个新的开发环境，也免不了从控制 LED 灯开始。下面我们开始进行本个工程的创建。

2.1.1 新建一个 HLS 工程

安装好 Vivado 套件后，已经自动安装好了 HLS，我们只需打开 Vivado HLS 即可，如所示。



错误!未找到引用源。

图 2- 1

打开 HLS 后如下图所示，点击 Create New Project，创建一个新的工程

图 2- 2

接下来输入工程的名称，选择工程的保存路径，点击 Next；

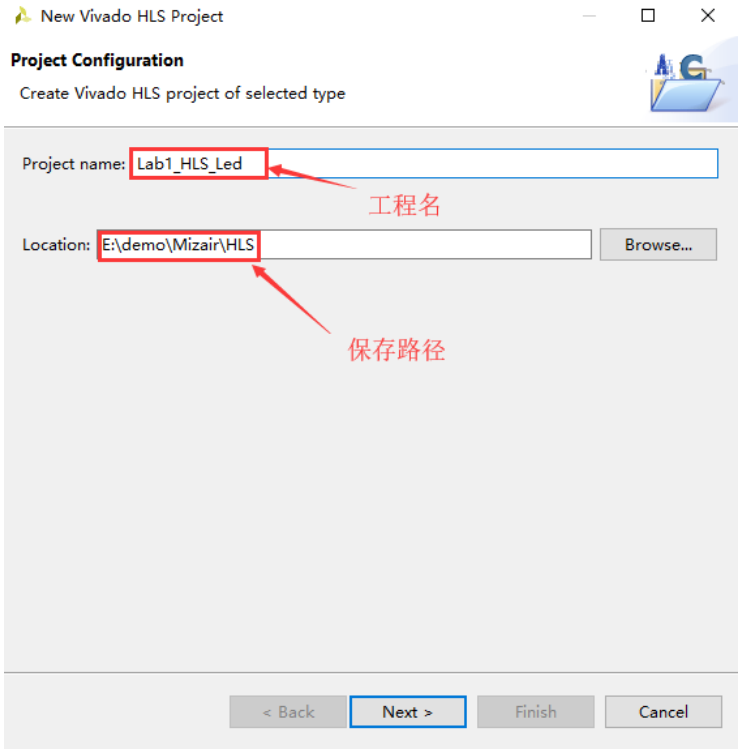


图 2- 3

接下来选择定成函数，我们在这里暂不不添加，直接点击 Next；

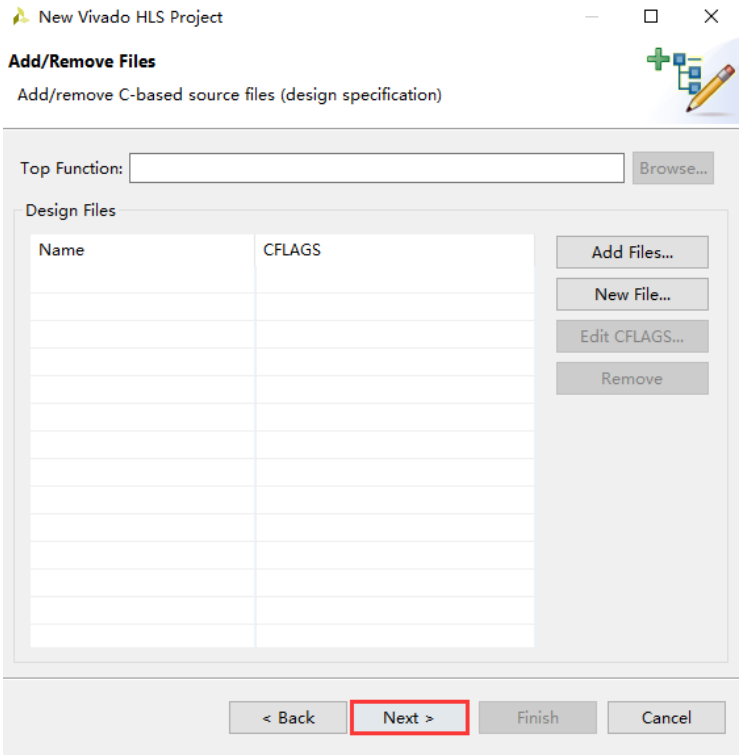


图 2- 4

接下来是添加 C 仿真文件，我们同样先不添加，点击 Next；

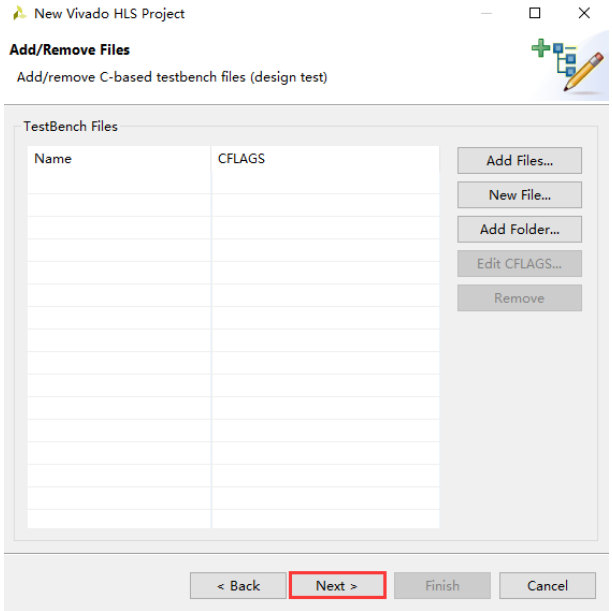


图 2- 5

接下来进行器件的选择，在首先我们点击图中 1 所示位置选择元器件，在选

期间选择窗口，购买 Z7-Lite7020 的用户选中我们开发板上的器件 xc7z020clg400-2，购买 Z7-Lite7010 的用户选中我们开发板上的器件 xc7z010clg400-1,点击 OK，Finish，完成工程的创建。

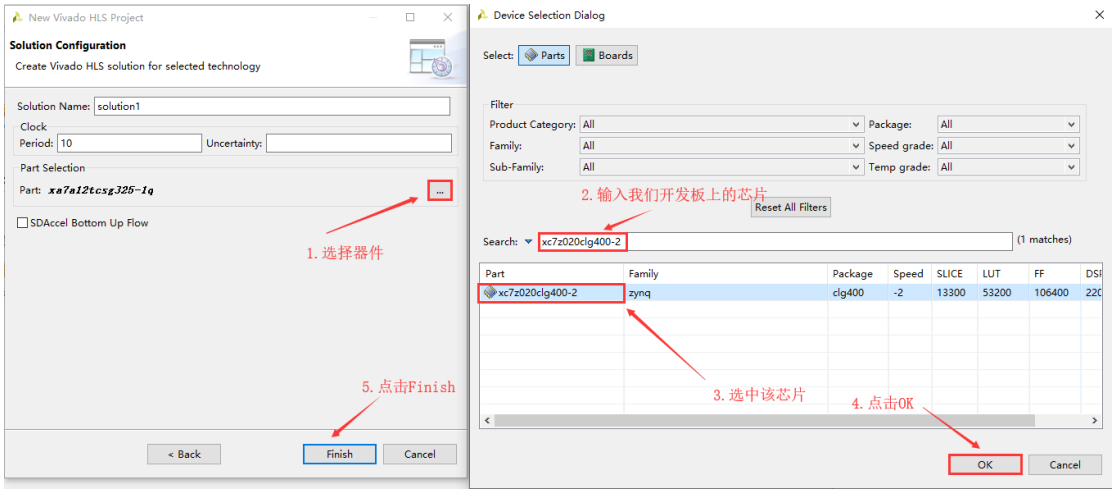


图 2- 6

完成创建后的界面所示，其中图中标注红色区域 1 是导航栏，绿色区域 2 是代码编辑区，蓝色区域 3 是所用到的变量函数和 Directive 区，紫色区域 4 是输出信息窗口。

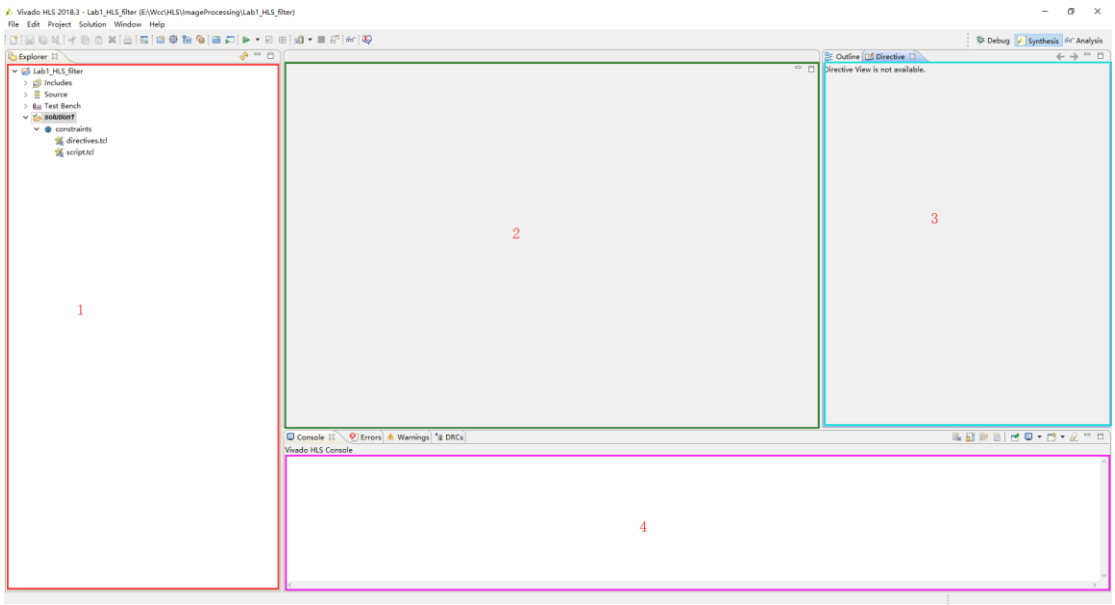


图 2- 7

在导航栏中，文件列表所示：

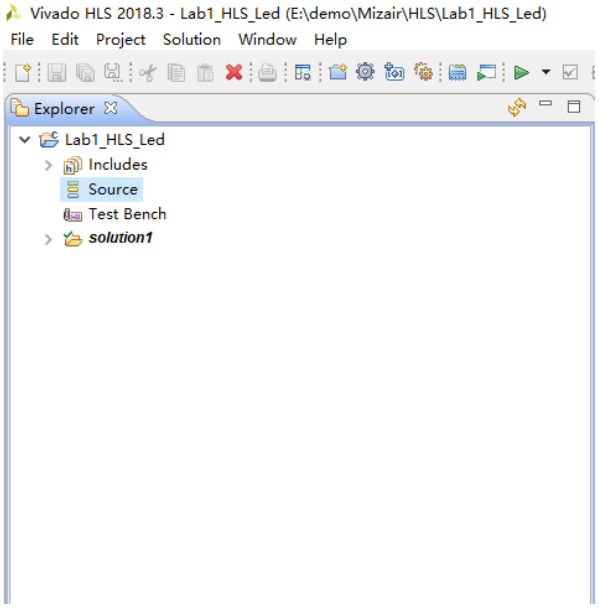


图 2- 8

其中 source 栏用来存放功能函数的源码，Test Bench 用来存放 C 仿真文件，solution 中包含着本次工程运行中和运行完成后的输出文件。

2.1.2 添加源文件

首先，我们添加本次工程的源码，所示，右键 Source，点击 New file，在弹出的窗口中，选中我们存放源码的目录后。新建一个 led.cpp 文件：

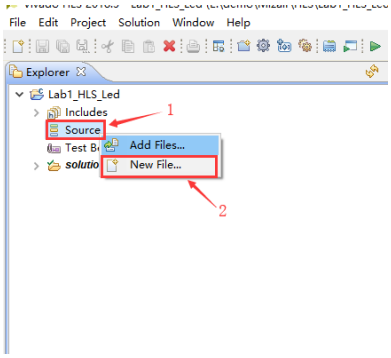


图 2- 9

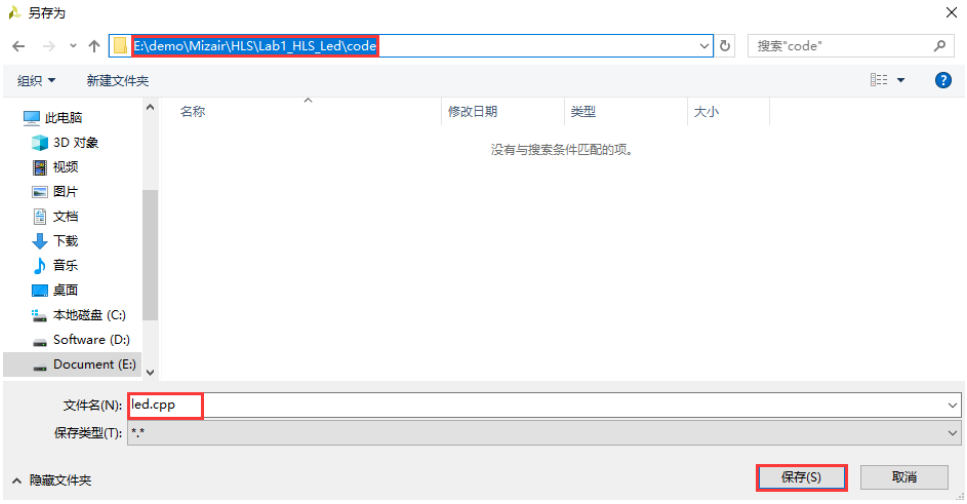


图 2- 10

回到保存后 HLS 会自动打开我们刚创建的文件，我们可以在该文件中编写，我们要实现的函数的代码。

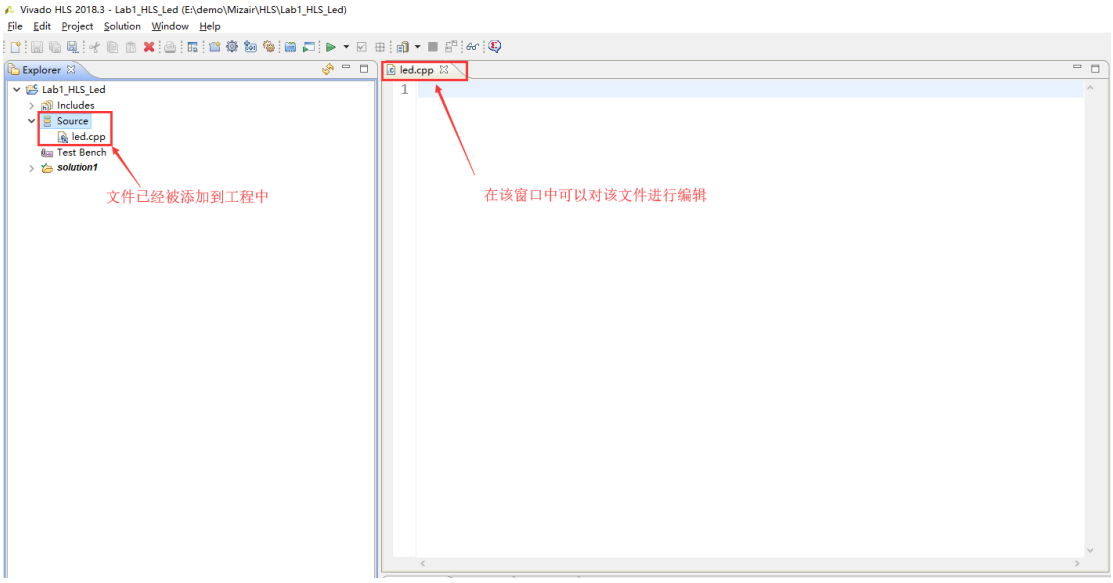


图 2- 11

按照同样的方法，我们可以为该 cpp 文件添加一个头文件；

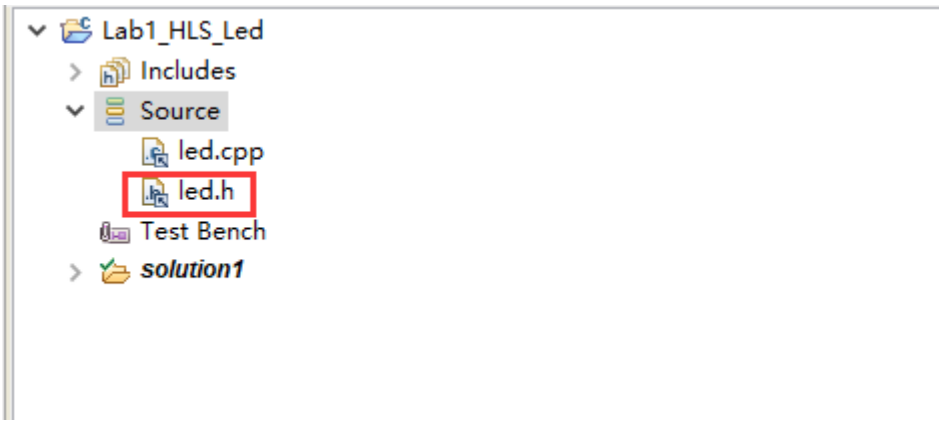


图 2- 12

接下来我们来完成本次实验的代码设计，本次实验的代码设计比较简单，其中 Source 文件中的头文件的代码如下：

```
#ifndef _SHIFT_LED_H_
#define _SHIFT_LED_H_
#define CNT_MAX 100000000
//#define CNT_MAX 100
#define FLASH_FLAG CNT_MAX-2
typedef        int led_t;
typedef        int cnt_t;
void flash_led(led_t *led_o , led_t led_i);
#endif
```

其中计数最大值 CNT_MAX 100000000 是在 100M 时钟频率下计数一秒钟所需要的计数次数，在仿真的时候，我们可以将其注释掉，采用下一个最大值定义，这样能够加快我们仿真的速度，FLASH_FLAG 是 LED 闪烁的标志，当计数到该值时，LED 发生变化。

接下来时对 int 类型变量的从定义。最后声明了一个函数，这也是我们咋子本次工程中所需要设计的定成函数。

Source 文件中的源文件设计如下：

首先引入头文件，然后完成该函数，使用了一个 for 循环来计数，当变量 i 计数到 FLASH_FLAG 时 led_o 的状态发生变化。可以看到该函数十分简单。

```
#include "led.h"

void flash_led(led_t *led_o , led_t led_i){
    cnt_t i;
    for(i=0;i<CNT_MAX;i++){
        if(i==FLASH_FLAG){
            *led_o = ~led_i;
        }
    }
}
```

2.1.3 添加 C 仿真文件

添加仿真文件的步骤与添加源文件类似，右键 Test Bench，选择添加新建文件，在弹出窗口新建一个 test_led.cpp。；

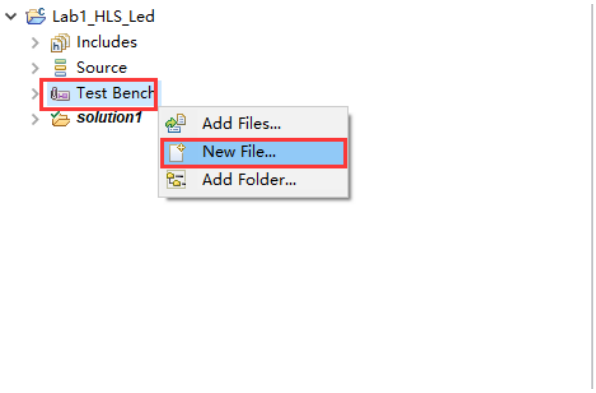


图 2- 13

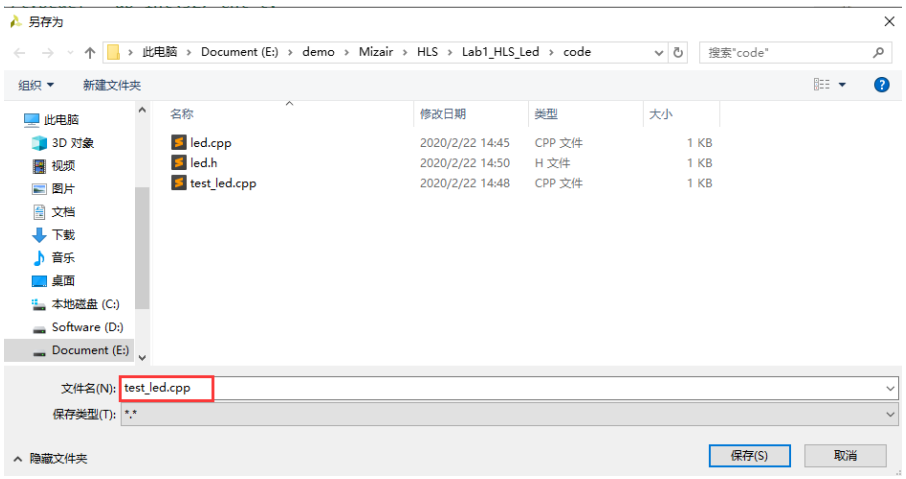


图 2- 14

C 仿真文件中，我们写入如下代码

```
#include "led.h"
#include <stdio.h>

int main(){

    led_t led_i=0x01;
    led_t led_o;
    const int SHIFT_TIME = 4;
    int i;
    for(i=0;i<SHIFT_TIME;i++){
        flash_led(&led_o , led_i);
        led_i = led_o;
        printf("shift_out is %d \n",(int)(led_o&0x01));
    }
}
```

在仿真文件中，我们将引入了 led.h 这个头文件，并且调用了 flash_led 这个函数，在仿真文件中，我们将这个输出从新给到输入，最终来输出 led_o 观察 led_o 的变化。完成 C 仿真文件后，我们可以进行 C 仿真和 C 综合映射到 RTL 电路。

2.1.4 进行 C 仿真与 C 综合

进行仿真之前，我们首先选中我们要仿真的顶层函数，点击 project →选择 project seethings→选择 synthesis→browser→选择 flash_led 作为顶层函数。

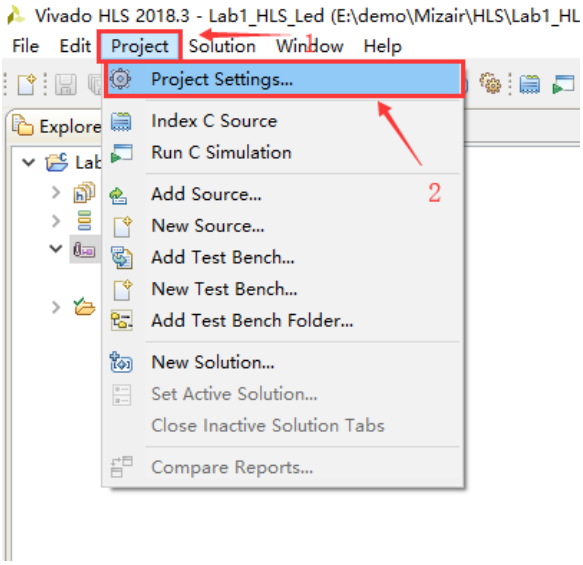


图 2- 15

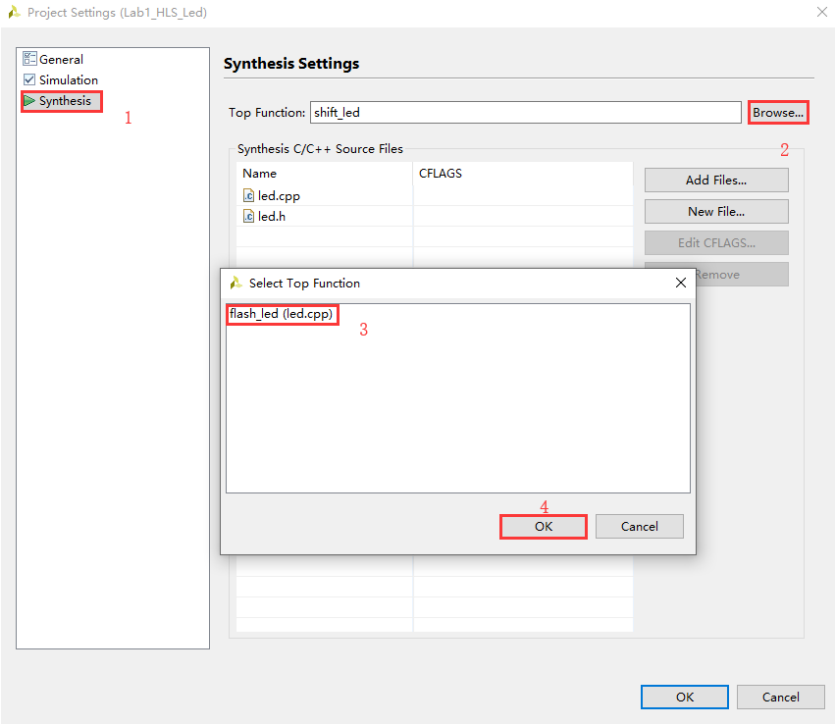


图 2- 16

然后点击 project → Run C simulation

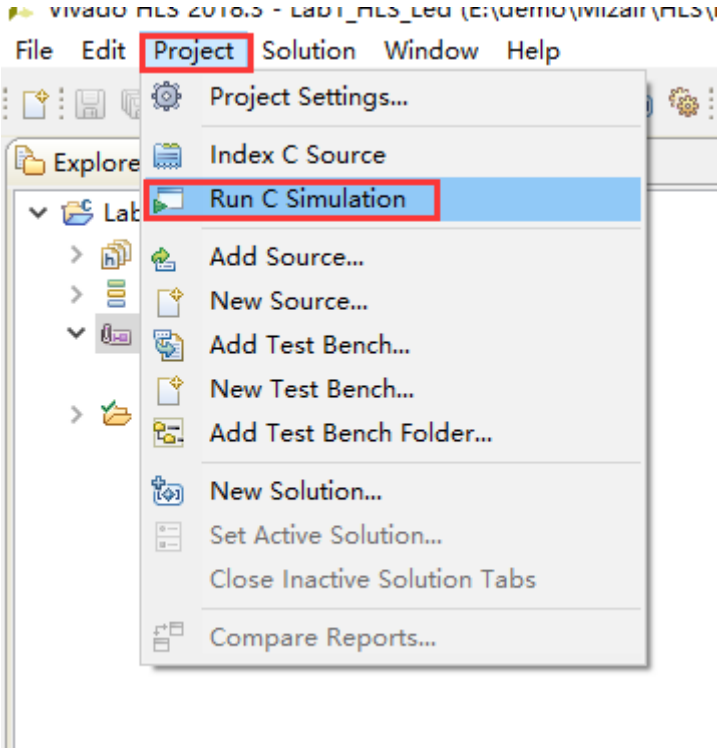


图 2- 17

在 Console 窗口中，我们可以看到输出的结果时 01 交替变化，证明 C 仿真的结果正确。

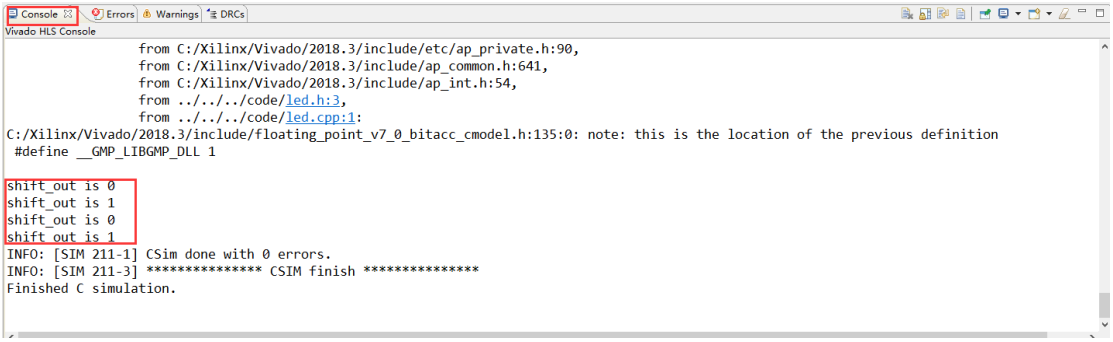


图 2- 18

接下来，进行 C 综合，编译器会将 C++代码映射到 RTL 电路，点击 Solution → Run C Synthesis → Active Solution。等待一段时间后，编译器会提示综合后的结果。

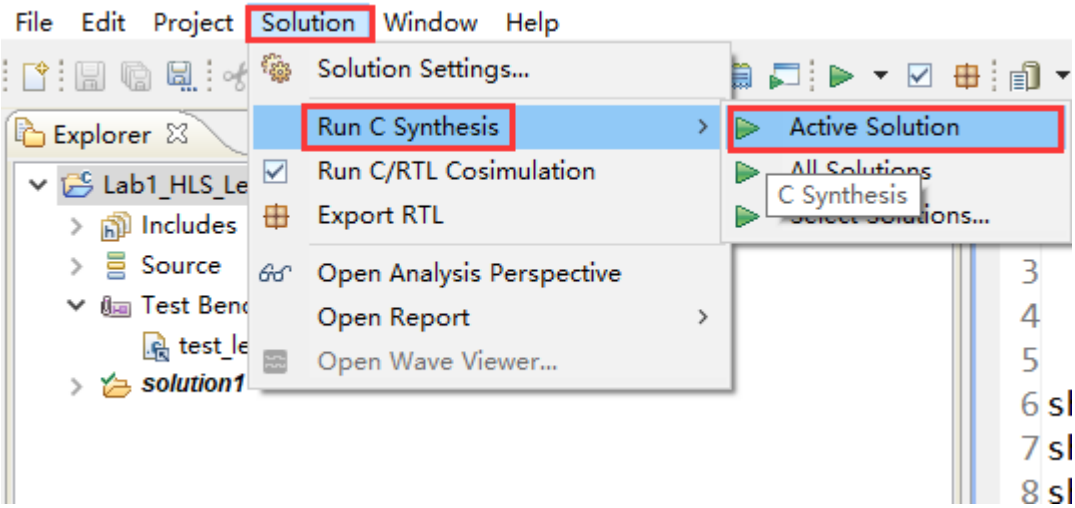


图 2- 19

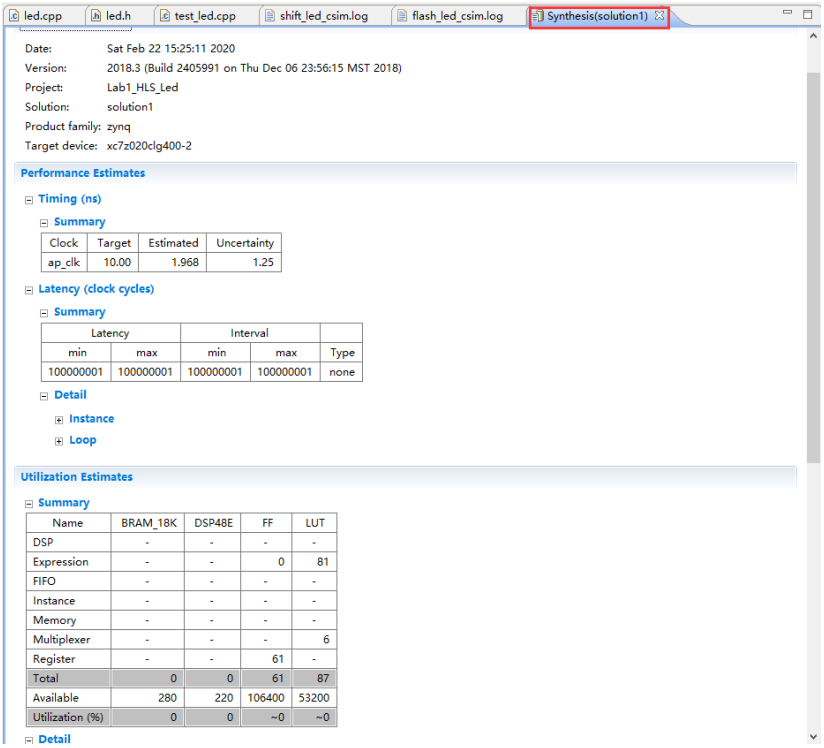


图 2- 20

在 C 综合后的结果中，我们可以查看所占用的资源，设计所需的 Latency，和接口的类型等等。

Timing 和 Latency 报告：其中 Latency 指的是，设计电路完成一次任务需要的时间，Interval 指的是两次任务之间的时间间隔

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	1.968	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
100000001	100000001	100000001	100000001	none

图 2- 21

占用的逻辑资源：可以看到，本次设计占用了 61 个触发器和 87 个查找表；

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	81
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	6
Register	-	-	61	-
Total	0	0	61	87
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

图 2- 22

综合出来的模块的接口如：我们可以看到，综合后的信号接口列表如下图所示，可以看到，综合后还生成了一些其他信号，这些信号的生成，这些信号可以用来标志本模块的工作状态。值得注意的是，这些信号与所采取的接口协议（Protocol）有关，中的 Protocol。相关的接口 Protocol 我们将在之后的章节中进行介绍，在这里我们只需要知道，这些接口的 Protocol 与我们采取的 Directive 相关。

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	flash_led	return value
ap_rst	in	1	ap_ctrl_hs	flash_led	return value
ap_start	in	1	ap_ctrl_hs	flash_led	return value
ap_done	out	1	ap_ctrl_hs	flash_led	return value
ap_idle	out	1	ap_ctrl_hs	flash_led	return value
ap_ready	out	1	ap_ctrl_hs	flash_led	return value
led_o	out	32	ap_vld	led_o	pointer
led_o_ap_vld	out	1	ap_vld	led_o	pointer
led_i	in	32	ap_none	led_i	scalar

图 2- 23

我们也可以查看 C 综合后生成的 RTL 代码，值得说明的是，由 HLS 生成的代码不具有可读性。在 Solution 的 Syn 中可以找到生成的 Verilog 代码。

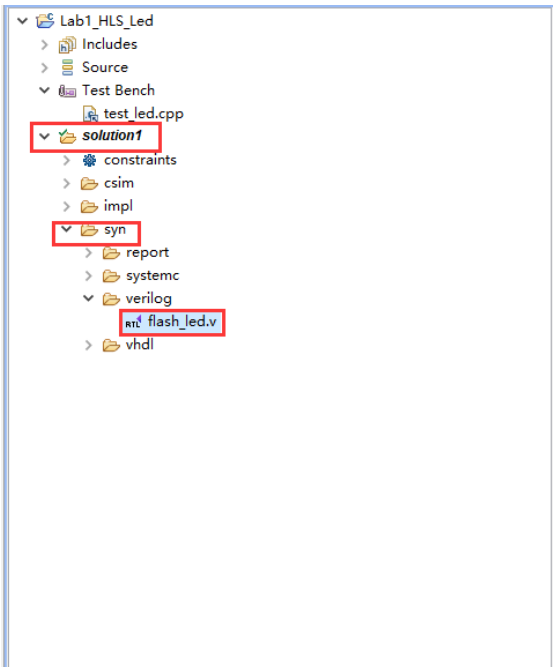


图 2- 24

2.1.5 联合仿真

接下来，我们进行 C/RTL 联合仿真，来验证映射出来的 RTL 电路是否正确，值得说明的是，Vivado HLS 会利用我们的 C Testbench 自动生成 Verilog 的 Testbench，同时，联合仿真结束过后，我们可以通过使用 Vivado 或者 Modelsim 来查看仿真

波形。

点击 Solution → Run C/RTL Cosimulation

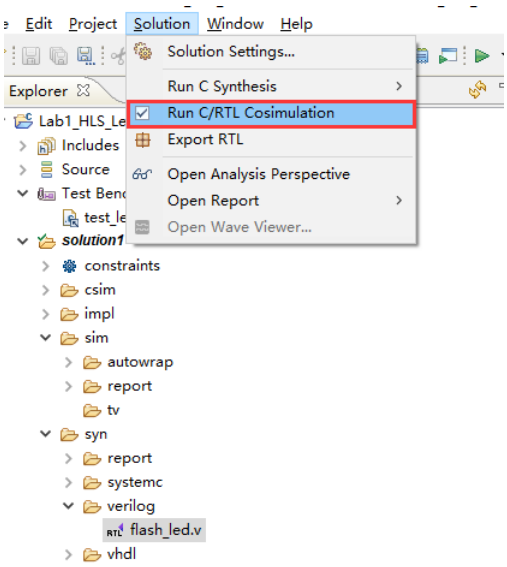


图 2- 25

在弹出窗口中首先选择仿真工具，我再这里选择了 Modelsim，也可以选择其他仿真工具，只要用户的 PC 中安装了该仿真工具，并且在环境变量 Path 中，有该仿真工具的路径，HLS 会自动定位到那个路径，然后开始仿真。

然后选择 Verilog 仿真，Dump Trace All 即可以开始仿真。同样可以在 Console 窗口中，观察打印信息。

由于前面在头文件中的最大值设置较大，会导致仿真时间很长，这与我们在 FPGA 开发中，进行仿真的时候，是一致的，从这里可以看出，HLS 本质上还是对电路的开发，只是使用了高级语言，通过工具将高级语言映射到 RTL 电路中。

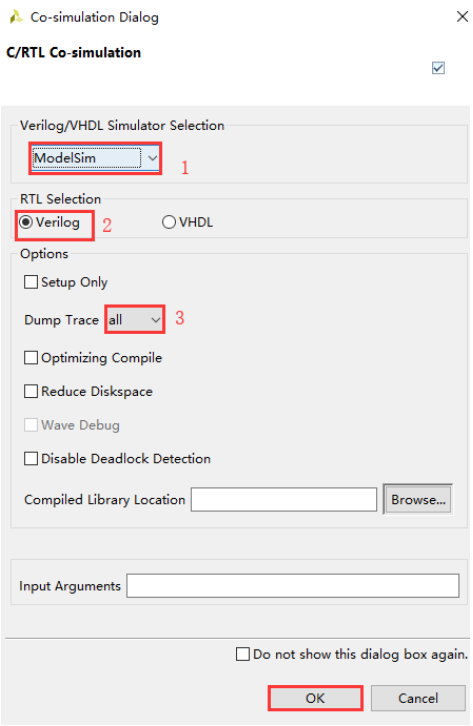


图 2- 26

在联合仿真时我们也可以观察 Console 打印出来的结果，：可以看到通过 C/RTL 联合仿真得到的结果与在 C 仿真时得到的结果一致。

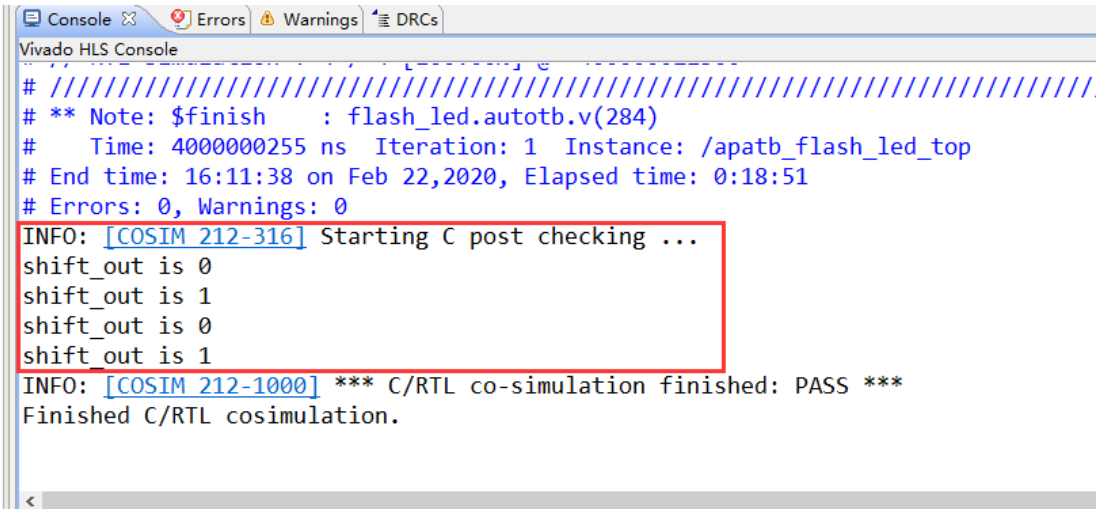


图 2- 27

至此时，我们已经可以说完成了整个设计，但是，我们可以考虑一下我们在设计中还存在的缺陷。对于这个 led 灯闪烁实验，我们可以看到，只需要 1bit 数据位宽就能表示这个变量，而在本次是实验中，我们还是采用的 int 类型来定义

的这个数据，从 C 综合 的结果来看，输入的 led_i 和输出的 led_o 仍然是 32 位位宽的。那么 HLS 中有没有办法实现像 FPGA 开发中那样，实现自定义的位宽呢？答案是可以的，接下来，就来对设计进行更改，完成自定以位宽的输入输出。

首先我们在源文件的头文件中，引入 ap_int.h 这个头文件，这个头文件是由 HLS 的库所提供的，通过引入这个头文件我们就可以调用其中的函数和关键字来声明一个任意位宽的数据了。

更改后的头文件如下：

```
1 #ifndef _SHIFT_LED_H_
2 #define _SHIFT_LED_H_
3
4 #include "ap_int.h"
5 #define CNT_MAX 100000000
6 //#define CNT_MAX 100
7 #define FLASH_FLAG CNT_MAX-2
8 // typedef        int led_t;
9 // typedef        int cnt_t;
10 typedef     ap_int<1> led_t;
11 typedef     ap_int<32> cnt_t;
12 void flash_led(led_t *led_o , led_t led_i);
13
14 #endif
```

可以看到我们在引入了头文件后，对于数据类型的从定义也发生了改变，在这里我们使用了 ap_int 的模板来从定义了 let_t 这个数据位宽为 1bit 的变量类型 led_t，和数据位宽为 32bit 的变量类型 cnt_t。

接下来我们再对源文件中的函数添加 Directive，来确定接口的类型。首先找到 Directive 的窗口，可以看到该窗口下有我们声明的函数，函数的形参和其中的 for 循环。在本章中，我们只进行添加 Directive，并不会对 Directive 做进一步说明，在后续的章节中，我们还会经常用到这些约束。

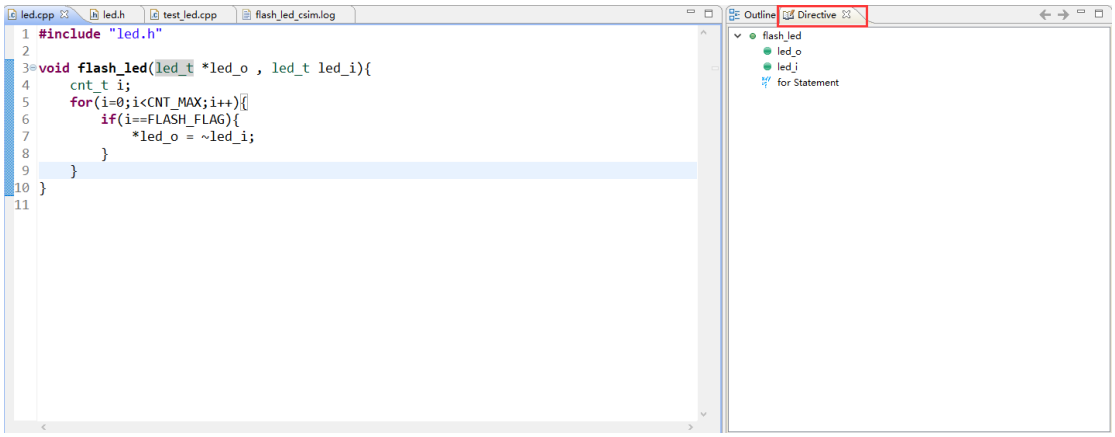


图 2- 28

首先对输出信号 led_o 进行约束，在 Directive 窗口中右键选中 led_o，选中插入 Directive；

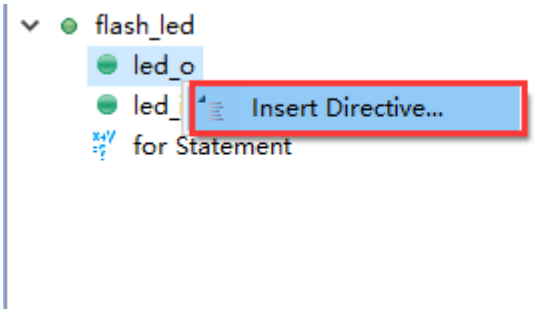


图 2- 29

在弹出窗口中，选择 Directive 为接口类型 INTERFACE；Destination 选择 Source file,这样工具会将约束语句添加到源文件中，若选择 Directive file 中会将 Directive 写入到一个 Tcl 脚本中，在这里选择添加到源文件中。Option 选项中，选择 ap_ovld 这会给输出信号一个输出有效指示信号。对于 led_i 采用类似方法，在 option 选项中选中 ap_vld 信号，这会给输入信号一个输入有效信号。

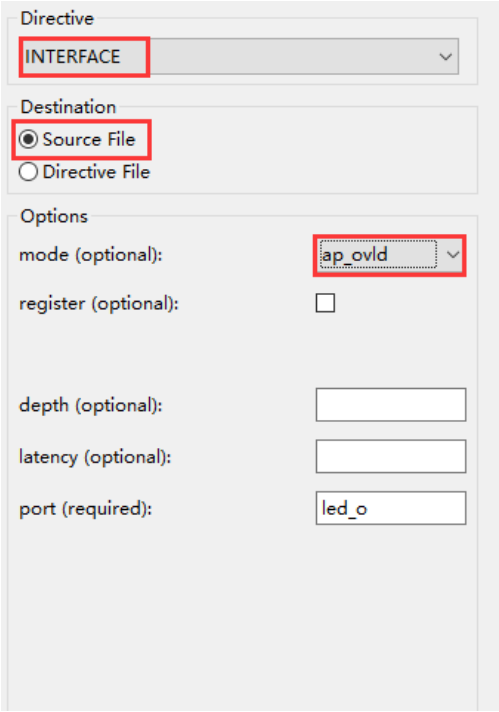


图 2- 30

添加完约束后的源文件如下：可以看到，Directive 已经被添加到了源文件中，接着我们再次进行 C 仿真。

```
#include "led.h"
void flash_led(led_t *led_o , led_t led_i){
#pragma HLS INTERFACE ap_vld port=led_i
#pragma HLS INTERFACE ap_ovld port=led_o
    cnt_t i;
    for(i=0;i<CNT_MAX;i++){
        if(i==FLASH_FLAG){
            *led_o = ~led_i;
        }
    }
}
```

仿真后的结果，可以看到，仿真结果依然正确，接下来进行 C 综合，我们通过观察添加 Directive 后的综合结果与之前的结果有何不同。

led.cpp | led.h | test_led.cpp | flash_led_csim.log

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../code/led.cpp in debug mode
   Generating csim.exe
shift_out is 0
shift_out is 1
shift_out is 0
shift_out is 1
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

图 2- 31

可以看到，改变将数据类型改变为自定义位宽的数据类型后，综合后生成所需要的触发器减少为 32 个，查找表减少为 70 个。

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	52
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	18
Register	-	-	32	-
Total	0	0	32	70
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

图 2- 32

接下来我们再查看添加 directive 后综合的接口信号，可以看到 HLS 已经为输入输出信号添加上了输入输出的有效标志。这样，我们再将来调用这个 IP 的时候，就能够控制它的时序。

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	flash_led	return value
ap_rst	in	1	ap_ctrl_hs	flash_led	return value
ap_start	in	1	ap_ctrl_hs	flash_led	return value
ap_done	out	1	ap_ctrl_hs	flash_led	return value
ap_idle	out	1	ap_ctrl_hs	flash_led	return value
ap_ready	out	1	ap_ctrl_hs	flash_led	return value
led_o_V	out	1	ap_ovld	led_o_V	pointer
led_o_V_ap_vld	out	1	ap_ovld	led_o_V	pointer
led_i_V	in	1	ap_vld	led_i_V	scalar
led_i_V_ap_vld	in	1	ap_vld	led_i_V	scalar

图 2- 33

最后，再次执行 Cosimulation。本次执行完联合仿真后，我们可以使用 Modelsim 或者 Vivado 软件打开联合仿真所产生的波形。

首先打开 Modelsim 软件，在 Modelsim 中，定位到 Solution 中，然后打开.wdf 文件。具体的路径是 Solution/sim/verilog/flash_led.wdf。使用 Modelsim 打开该文件即可观察波形。

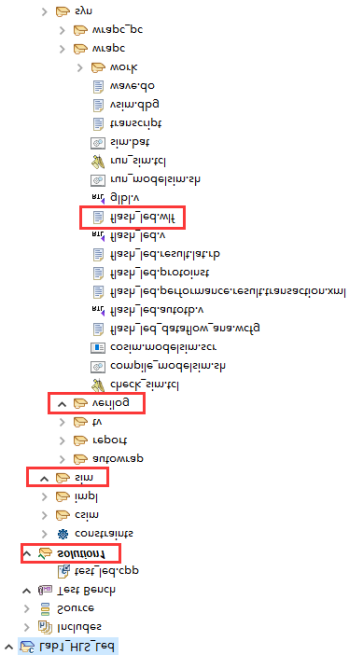


图 2- 34

打开该文件后，将定成模块的信号全部添加到波形窗口中。

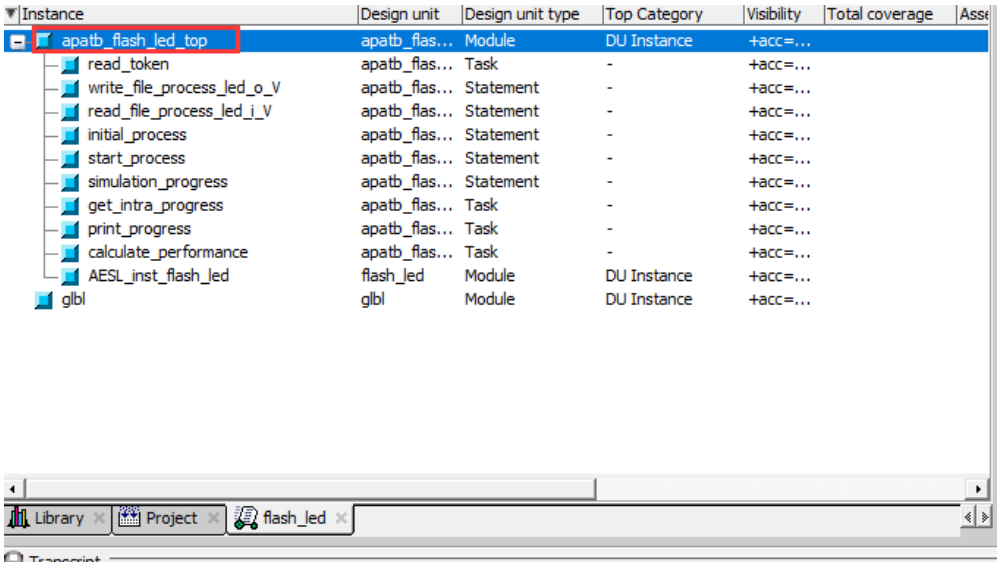


图 2- 35

，我们可以观察到输出输入输入输出信号，和其他一些，信号，其中需要重点关注的信号是上面 C 综合后生成的那些信号。

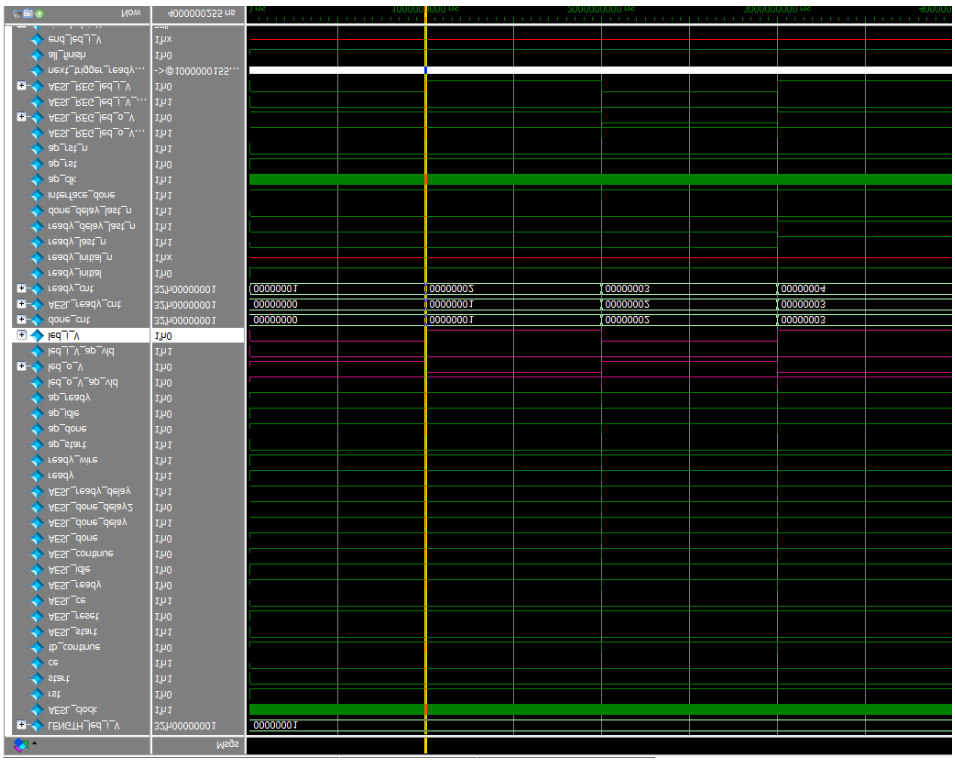


图 2- 36

1) 导出 HLS 工程生成的 IP 核

经过上面的步骤，已经完成了 HLS 验证的阶段，并且得知功能函数的结果无误，接下来就可以将生成的 RTL 电路导出，并在 Vivado 中调用。

点击 solution 选择 Export RTL 即可以将 HLS 所生成的 RTL 电路导出为 IP 核。

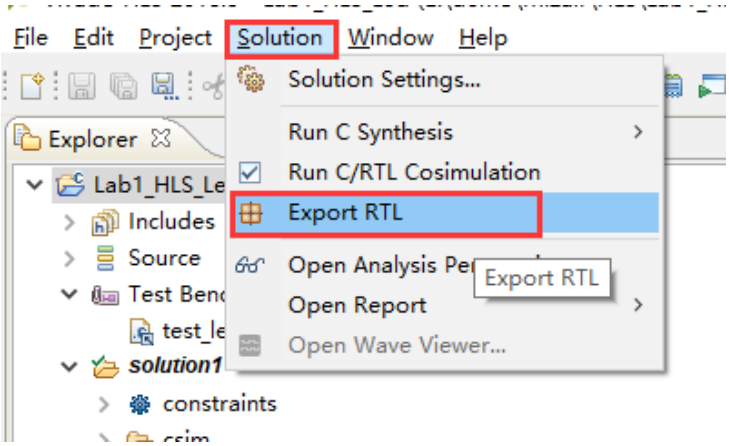


图 2- 37

导出的 IP 核将在 Solution 这个文件夹中可以找到。

2.2 创建 Vivado 工程并导入 HLS 生成的 IP 核

接下来将 HLS 生成的 IP 核导入到 Vivado 中。首先打开 Vivado，创建一个新的 vivado 工程。在前面的 FPGA 基础开发部分已经有介绍如何去创建一个 Vivado 工程，在这里不再赘述。这里建议创建的 Vivado 工程与 HLS 工程处于同一文件夹，便于我们管理文件。

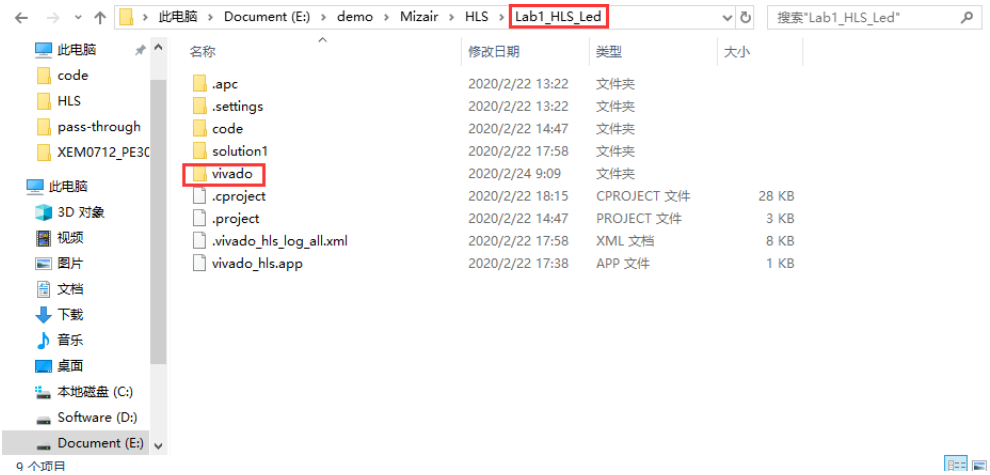


图 2- 38

创建完工程后，我们将由 HLS 生成的 IP 添加到我们的 IP 仓库中。点击 settings

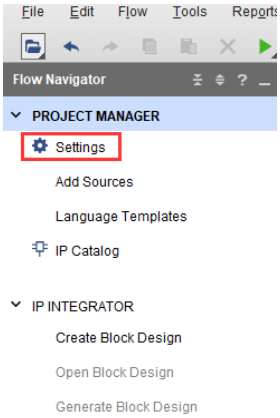


图 2- 39

然后点击 IP，选中仓库，再点击+号进行添加由 HLS 生成的 IP。

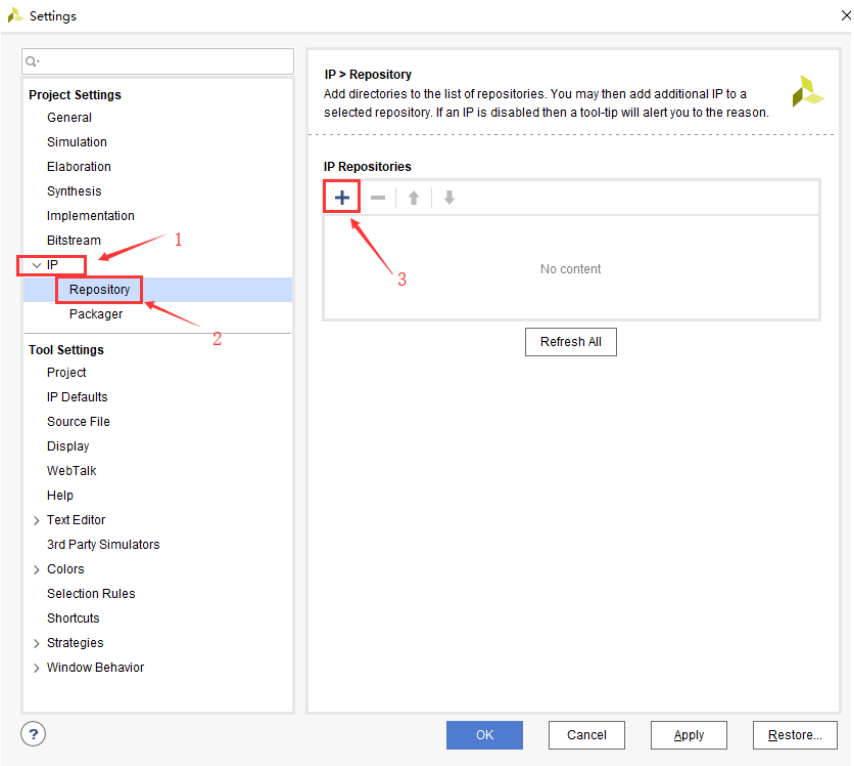


图 2- 40

在弹出的界面中，定位到 Solution，点击选择系统将自动识别到 IP;

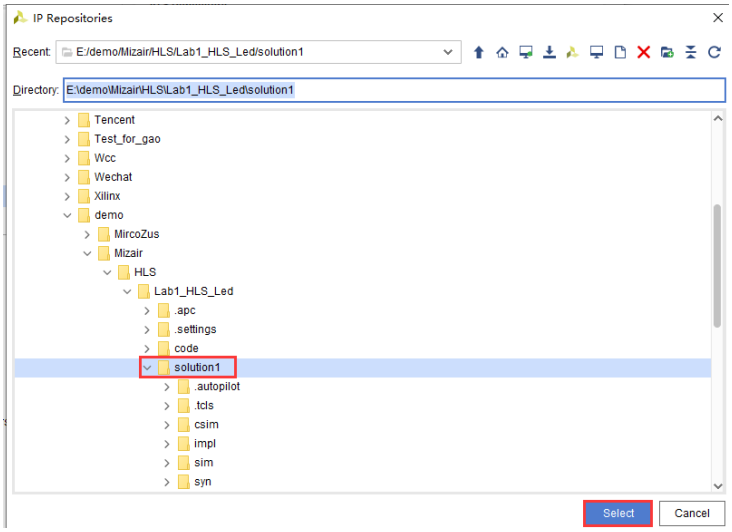


图 2- 41

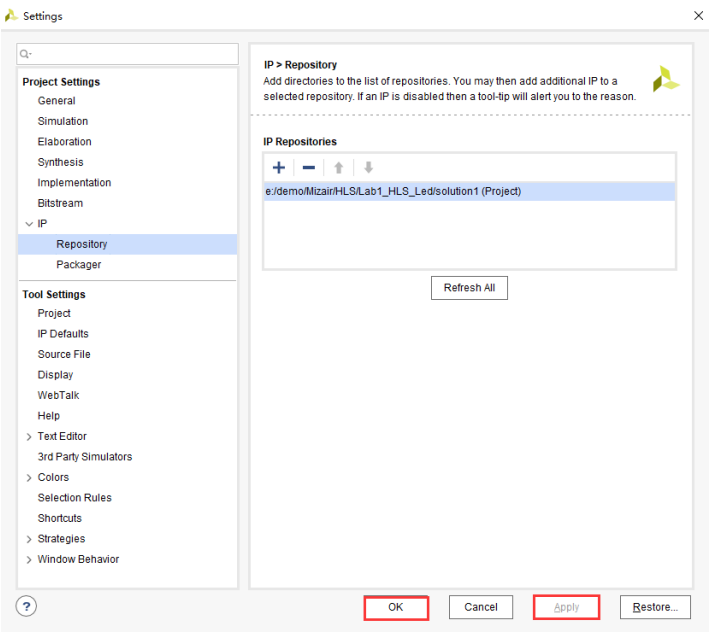


图 2- 42

点击 apply → OK,即可将 IP 添加到仓库中。

接下来我们来验证 HLS 生成的 IP 是否被导入到 Vivado IP 仓库。点击 IP Catalog 即可发现 HLS 已经被添加到了 IP 仓库中；

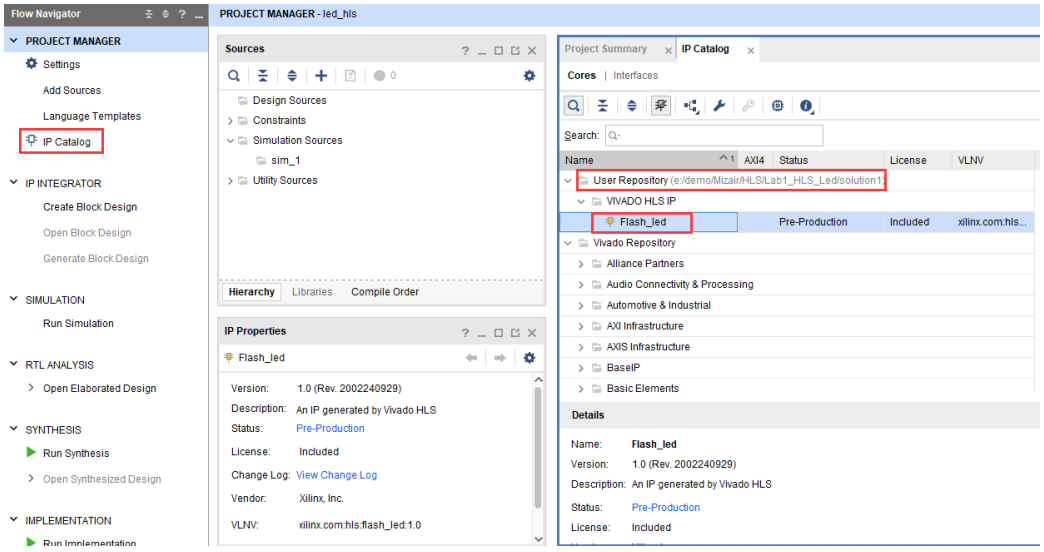


图 2- 43

2.3 将 HLS 产生的 IP 添加到工程当中

在 IP Catalog 中选中由 HLS 生成的 IP，双击并生成该 IP；

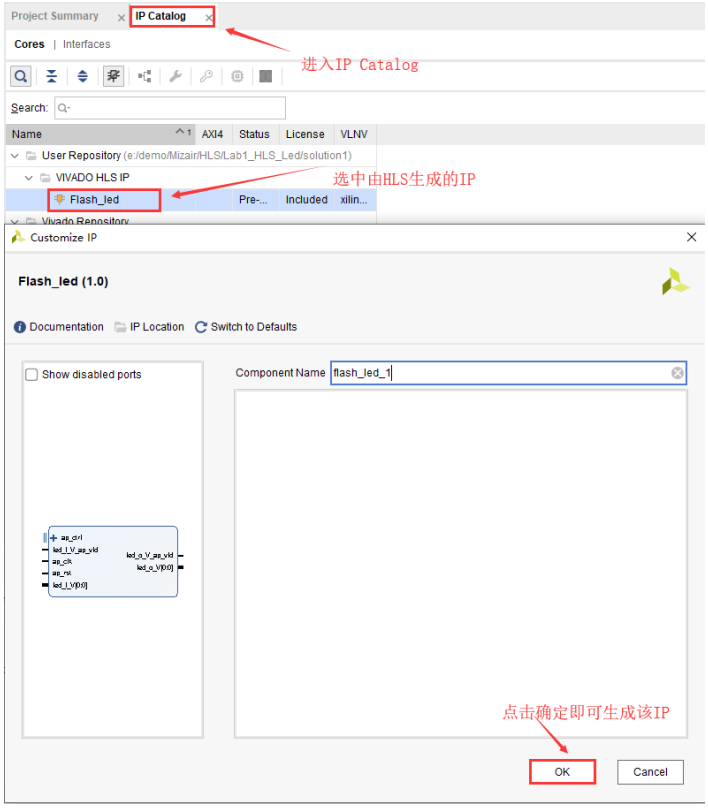


图 2- 44

向工程中添加一个新的文件，用于完成本次实验，点击 Add file

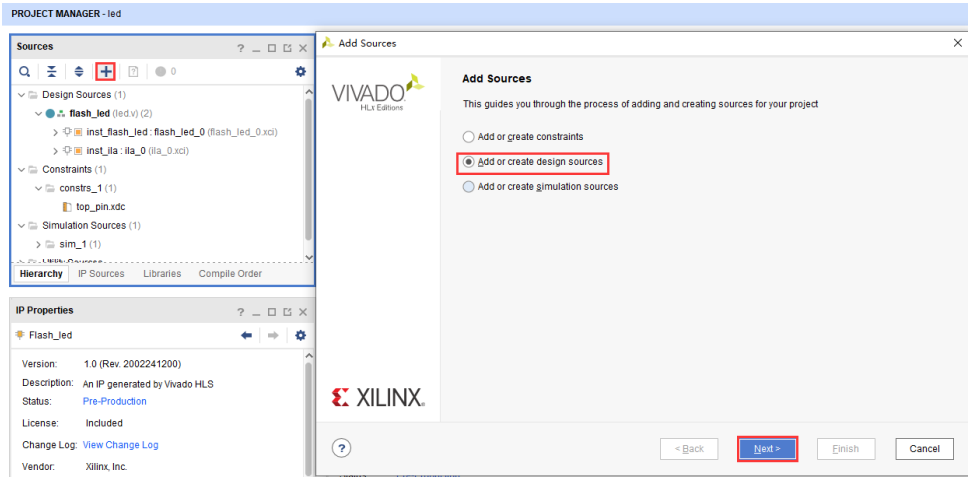


图 2- 45

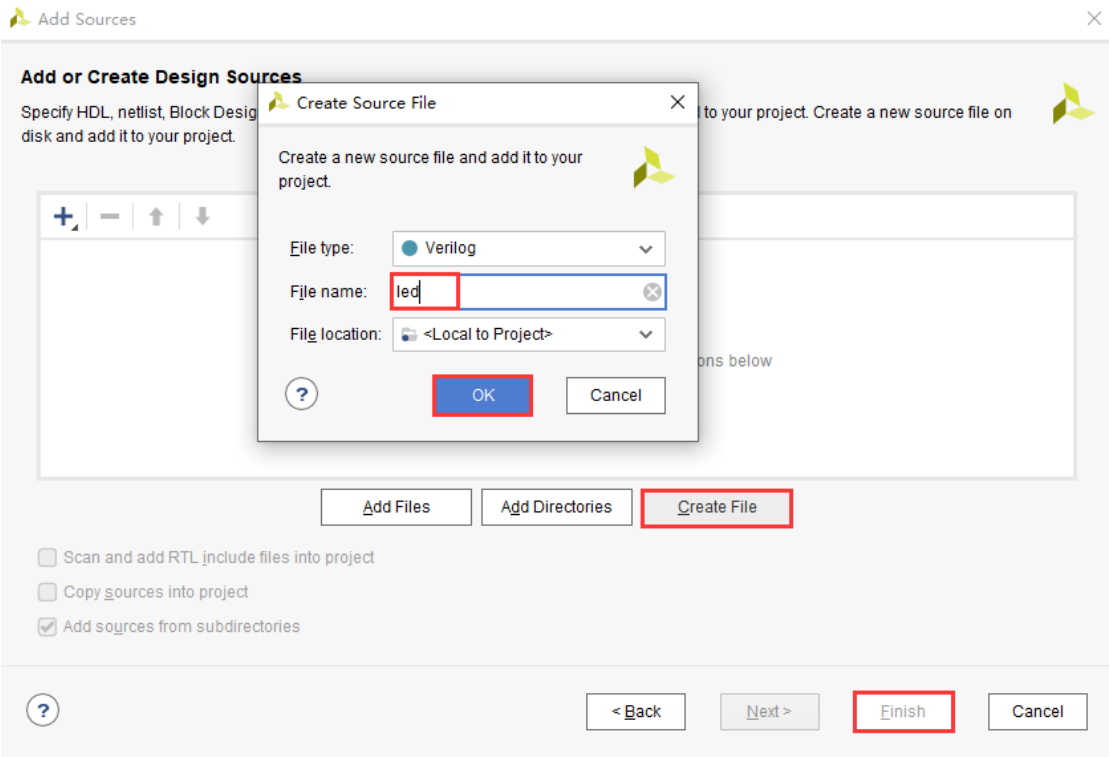


图 2- 46

添加完文件后，可以进行代码的编写，本次实验的代码比较简单，这里不做过多的讲解，只需将生成的 HLS IP 例化进工程当中即可。

```
`timescale 1ns / 1ps
module flash_led(
 input    wire          clk  ,
 input    wire          rst_n    ,
 output   wire          led_o
     );

wire        rst         ;//同步复位
wire        ap_ready        ;//当前可以接收下一次数据
reg         ap_start    ;//IP 开始工作
reg         led_i_vld       ;//输入数据有效
wire            led_o_vld       ;
reg         led_i           ;//输入的 led 信号
wire            led_o_r     ;
wire            ap_done         ;
wire            ap_idle     ;
reg     [1:0]   delay_cnt       ;
```

```
18
19  assign rst = ~rst_n ;
20  assign led_o = led_o_r   ;
21
22  //----------------delay_cnt------------------
23  always @(posedge clk) begin
24      if (rst==1'b1) begin
25          delay_cnt <= 'd0;
26      end
27      else if(delay_cnt[1]==1'b0) begin
28          delay_cnt <= delay_cnt + 1'b1;
29      end
30  end
31
32  //----------------ap_start------------------
33  always @(posedge clk) begin
34      if (rst==1'b1) begin
35          ap_start <= 1'b0;
36      end
37      else if(delay_cnt[1]==1'b1)begin
38          ap_start <= 1'b1;
39      end
40  end
41
42  //----------------led_i_vld------------------
43  always @(posedge clk) begin
44      if (rst==1'b1) begin
45          led_i_vld <= 1'b0;
46      end
47      else if(delay_cnt[1]==1'b1)begin
48          led_i_vld <= 1'b1;
49      end
50  end
51
52  //----------------ap_i------------------
53  always @(posedge clk) begin
54      if (rst==1'b1) begin
55          led_i <= 1'b0;
56      end
57      else if(led_o_vld==1'b1)begin
58          led_i <= led_o_r ;
```

```
        end
    end


    flash_led_0 inst_flash_led (
        .led_o_V_ap_vld(led_o_vld),    // output wire led_o_V_ap_vld
        .led_i_V_ap_vld(led_i_vld),    // input wire led_i_V_ap_vld
        .ap_clk(clk),                  // input wire ap_clk
        .ap_rst(rst),                  // input wire ap_rst
        .ap_start(ap_start),           // input wire ap_start
        .ap_done(ap_done),             // output wire ap_done
        .ap_idle(ap_idle),             // output wire ap_idle
        .ap_ready(ap_ready),           // output wire ap_ready
        .led_o_V(led_o_r),             // output wire [0 : 0] led_o_V
        .led_i_V(led_i)                // input wire [0 : 0] led_i_V
    );

    endmodule
```

接下来就可以添加约束文件和进行最终的板级验证了。

添加的约束文件如下：

```
##############LED define##################
set_property PACKAGE_PIN P15 [get_ports {led_o}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_o}]

##############Reset define##################
set_property PACKAGE_PIN P16 [get_ports {rst_n}]
set_property IOSTANDARD LVCMOS33 [get_ports {rst_n}]

##############50M CLK define##################
create_clock -period 20.000 -name clk -waveform {0.000 10.000} [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports {clk}]
set_property IOSTANDARD LVCMOS33 [get_ports {clk}]
```

2.4 下载验证

添加完约束文件后，就可以进行最终的板级验证了，首先生成 bit 流文件；

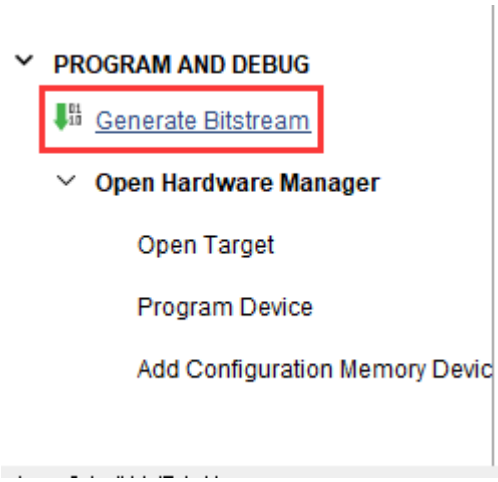


图 2- 47

等待 bit 文件生成完成之后，链接按照如下方式连接开发板

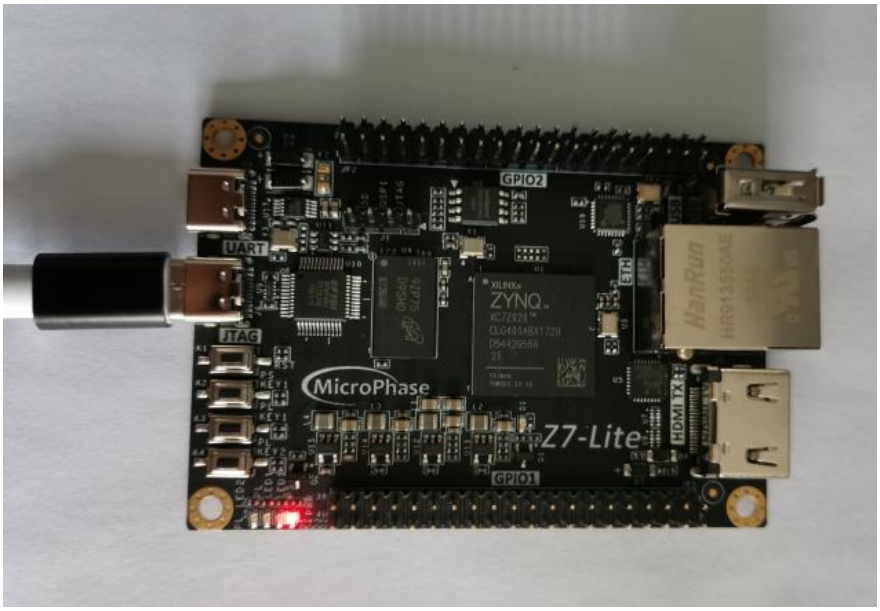


图 2- 48

然后将 bit 文件下载入 FPGA 中，观察实验现象，最终可以发现，开发板上的 led 灯开始持续亮灭。

2.5 实验总结

通过本次实验，完成了通过 HLS 来封装一个 IP 并将这个 IP 应用到 Vivado 工

程当中，可以看到，整个流程还是比较复杂的，本次实验，十分简单，仅仅指示展示了 HLS 开发的流程，并没有体现出 HLS 的优势，在之后的章节中，将会进一步加深对 HLS 软件的使用。

总结一下 HLS 开发的基本步骤就是，先要有 C/C++源文件和 C/C++仿真文件，进行 C 仿真，然后进行 C 综合，将 C/C++语言描述的函数，映射到 RTL 电路，最后进行 C/RTL 的联合仿真，通过 C 仿真的 Testbench 生成 Verliog 仿真的 Testbench，验证 RTL 电路的正确性。

第3章　搭建 HLS 图像仿真平台

在这一章节的实验中，将对 HLS 中带有的 OpenCv 库做一个简单的介绍，理解 HLS 进行图像算法开发的流程，分清楚 HLS 中可综合的代码和 C 仿真代码的之间的联系和区别。

3.1 OpenCv 与 HLS 中的 OpenCv 库简介

OpenCV 是一个函数库，它包含许多针对桌面处理器和 gpu 的常见计算机视觉功能的优化实现。尽管 OpenCV 库中的许多功能已经进行了大量优化，以使许多计算机视觉应用程序能够接近实时地运行，但优化的嵌入式实现通常更可取。

Vivado HLS 包含许多视频库，能够让用户更容易地构建各种视频处理系统。这些库由可综合的 C++代码实现，这些 C++代码的功能大致相当于 OpenCV 中实现的视频处理函数和数据结构。许多视频概念和抽象与 OpenCV 中的概念和抽象非常相似。特别是 OpenCV 图像处理模块中的许多函数都有相应的 Vivado HLS 库函数。

在视频处理系统的设计过程中，OpenCV 可以在多个位置使用。，可以使用 OpenCV 函数调用来设计和实现一个完整的算法，既可以使用文件访问函数来输入和输出图像，也可以对图像进行处理。该算法可以在嵌入式系统(如 Zynq Base TRD)中实现，使用特定于平台的函数调用来访问输入和输出图像。在这种情况下，视频处理仍然使用在处理器上执行的 OpenCV 函数调用来实现(例如 Zynq 处理器系统中的 Cortex -A9 处理器内核)。

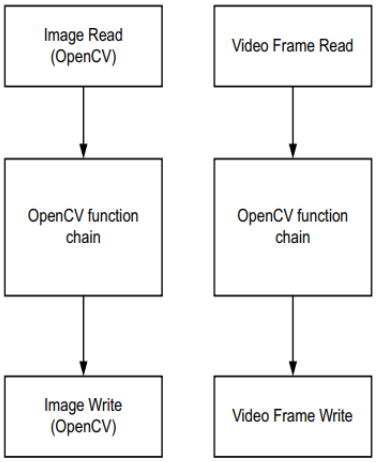


图 3- 1

OpenCV 函数调用也可以由 Xilinx Vivado HLS 视频库中相应的合成器函数替换。OpenCV 函数调用可用于访问输入和输出图像，并提供视频处理算法的黄金参考实现。合成后的处理块可集成到 Zynq 可编程逻辑中。根据在可编程逻辑中实现的设计，集成块可以处理处理器创建的视频流，比如从文件中读取的数据，或者从外部输入的实时视频流。

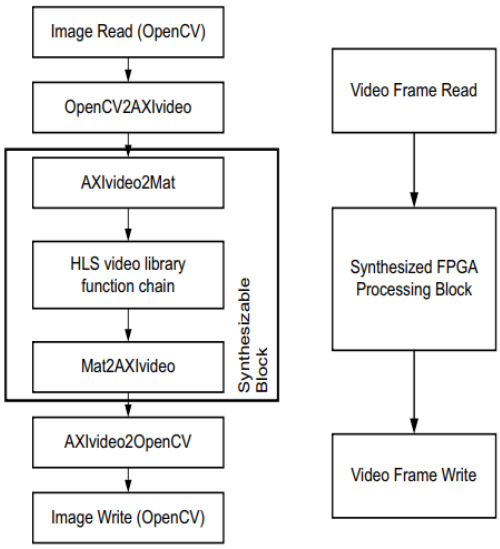


图 3- 2

3.2 常用的 OpenCv 中的结构与函数

在本章中，我们先介绍常用的 OpenCv 函数，了解使用 OpenCV 处理图像的过程。这些函数，将在搭建 C 仿真平台时提供读取输入图像、保存生成的图像，提供黄金参考对比图像等，通过这些函数，完成与我们可综合的 HLS 库中的相应函数的对比，来判断函数功能是否正确。想要使用 HLS 中的 OpenCv 库函数和变量，需引入头文件 hls_opencv.h 和 opencv2/opencv.hpp。

3.2.1 IplImage 结构体

IplImage 是 OpenCV 中基础的数据结构，用来表示图像，其中 Ipl 是 Intel Image Processing Library 的简写。IplImage 是一个结构体，其中结构的定义：

```
typedef struct _IplImage
{
    int  nSize;             /* sizeof(IplImage) */
    int  ID;                /* version (=0)*/
    int  nChannels;         /* Most of OpenCV functions support 1,2,3 or 4 channels */
    int  alphaChannel;      /* Ignored by OpenCV */
    int  depth;             /* Pixel depth in bits: IPL_DEPTH_8U, IPL_DEPTH_8S, IPL_DEPTH_16S,
                               IPL_DEPTH_32S, IPL_DEPTH_32F and IPL_DEPTH_64F are supported.  */
    char colorModel[4];     /* Ignored by OpenCV */
    char channelSeq[4];     /* ditto */
    int  dataOrder;         /* 0 - interleaved color channels, 1 - separate color channels.
                               cvCreateImage can only create interleaved images */
    int  origin;            /* 0 - top-left origin,
                               1 - bottom-left origin (Windows bitmaps style).  */
    int  align;             /* Alignment of image rows (4 or 8).
                               OpenCV ignores it and uses widthStep instead.    */
    int  width;             /* Image width in pixels.                           */
    int  height;            /* Image height in pixels.                          */
    struct _IplROI *roi;    /* Image ROI. If NULL, the whole image is selected. */
    struct _IplImage *maskROI;      /* Must be NULL. */
    void  *imageId;                 /* "           " */
    struct _IplTileInfo *tileInfo;  /* "           " */
    int  imageSize;         /* Image data size in bytes
                               (==image->height*image->widthStep
                               in case of interleaved data)*/
    char *imageData;        /* Pointer to aligned image data.         */
    int  widthStep;         /* Size of aligned image row in bytes.    */
    int  BorderMode[4];     /* Ignored by OpenCV.                     */
    int  BorderConst[4];    /* Ditto.                                 */
    char *imageDataOrigin;  /* Pointer to very origin of image data
                               (not necessarily aligned) -
                               needed for correct deallocation */
}
IplImage;
```

图 3- 3

根据该结构体的定义，我们可以看到，该结构类型的数据中，包含由图像的大小，通道数，深度等信息。IplImage 这个结构体中存储图像数据的部分是 char * imageData，这个指针指向图像数组的首地址。

1) 加载图像 cvLoadImage

在图像开发中，经常需要从外部读取图像文件到应用，在 OpenCv 中，可以使用 OpenCv 中的 cvLoadImage 这个函数，将图像数据读取进来。

cvLoadImage 的定义如下：从文件中加载文件，可以在加载文件的时候，通过参数，来改变或不该图片的状态。

```
enum
{
/* 8bit, color or not */
    CV_LOAD_IMAGE_UNCHANGED   =-1,
/* 8bit, gray */
    CV_LOAD_IMAGE_GRAYSCALE   =0,
/* ?, color */
    CV_LOAD_IMAGE_COLOR       =1,
/* any depth, ? */
    CV_LOAD_IMAGE_ANYDEPTH    =2,
/* ?, any color */
    CV_LOAD_IMAGE_ANYCOLOR    =4
};
/* load image from file
  iscolor can be a combination of above flags where CV_LOAD_IMAGE_UNCHANGED
  overrides the other flags
  using CV_LOAD_IMAGE_ANYCOLOR alone is equivalent to CV_LOAD_IMAGE_UNCHANGED
  unless CV_LOAD_IMAGE_ANYDEPTH is specified images are converted to 8bit
*/
CVAPI(IplImage*) cvLoadImage( const char* filename, int iscolor CV_DEFAULT(CV_LOAD_IMAGE_COLOR));
```

具体的使用方法如下：

```
IplImage * img_0;
img_0 = cvLoadImage("fileName.bmp");
```

img_0 将指向加载进来的图片的地址，通过这种方法，即可将图像导入到应用程序中来。

2) 创建图像 cvCreateImage

函数 cvCreateImage 创建图像首地址，并分配存储空间。

```
/* Creates IPL image (header and data) */
CVAPI(IplImage*)    cvCreateImage( CvSize size, int depth, int channels);
```

使用方法如下，其中 size 是要创建的图像的大小，depth 是要创建的图像的深度，channels 是要创建图像的通道数。

```
IplImage * img_0;
img_0 = cvCreateImage( size, depth,channels);
```

3) 保存图像 cvSaveImage

其功能是将图像保存到文件中，函数定义如下：

```
enum
{
    CV_IMWRITE_JPEG_QUALITY =1,
    CV_IMWRITE_PNG_COMPRESSION =16,
    CV_IMWRITE_PNG_STRATEGY =17,
    CV_IMWRITE_PNG_BILEVEL =18,
    CV_IMWRITE_PNG_STRATEGY_DEFAULT =0,
    CV_IMWRITE_PNG_STRATEGY_FILTERED =1,
    CV_IMWRITE_PNG_STRATEGY_HUFFMAN_ONLY =2,
    CV_IMWRITE_PNG_STRATEGY_RLE =3,
    CV_IMWRITE_PNG_STRATEGY_FIXED =4,
    CV_IMWRITE_PXM_BINARY =32
};

/* save image to file */
CVAPI(int) cvSaveImage( const char* filename, const CvArr* image,
                                    const int* params CV_DEFAULT(0) );
```

4) 释放图像 cvReleaseImage

该函数的功能是，释放图像所暂用的空间。

函数定义如下：

```
/* Releases IPL image header and data */
CVAPI(void)    cvReleaseImage( IplImage** image );
```

使用方法如下：

```
IplImage * img_0;
cvReleaseImage(&img_0);
```

3.2.2 OpenCv 中的 Mat 类

在早期的 OpenCV1.x 版本中，图像的处理是通过 IplImage 结构来实现的。早期的 OpenCV 是用 C 语言编写，因此提供的借口也是 C 语言接口，其源代码完全是 C 的编程风格。IplImage 结构是 OpenCV 矩阵运算的基本数据结构。到 OpenCV2.x 版本，OpenCV 开源库引入了面向对象编程思想，大量源代码用 C++重写，Mat 类（Matrix 的缩写）是 OpenCV 用于处理图像而引入的一个封装类。从功能上讲，Mat 类在 IplImage 结构的基础上进一步增强，并且，由于引入 C++高级编程特性，Mat 类的扩展性大大提高，Mat 类的内容在后期的版本中不断丰富，如果你查看 Mat 类的定义的话，会发现其设计实现十分全面而具体，基本覆盖计算机视觉对于图像处理的基本要求。

对于 Mat 类，功能十分强大，在本章中，只介绍其对象简单的初始化，有兴趣可以进入其中的库文件查看源码。

1) 读取图像 imread

其中 imread 的定义如下，通过读取外部文件，

```
enum
{
    // 8bit, color or not
    IMREAD_UNCHANGED    =-1,
    // 8bit, gray
    IMREAD_GRAYSCALE    =0,
    // ?, color
    IMREAD_COLOR        =1,
```

```
    // any depth, ?
    IMREAD_ANYDEPTH     =2,
    // ?, any color
    IMREAD_ANYCOLOR     =4
};
CV_EXPORTS_W Mat imread( const string& filename, int flags=1 );
```

通过 imread 可以对 Mat 类的对象进行初始化，具体使用方法如下：

```
Mat src_rgb = imread("fileName.bmp");
IplImage src = src_rgb;
cvShowImage("src",&src);
```

上述代码，能够实现将图像读入，并且显示出来。

3.3 HLS 中可综合的图像处理模块

在上一小节中，介绍的 OpenCv 函数，是可以供 HLS 软件直接调用的，但是却不是 HLS 所支持的可综合，对于可综合映射成 RTL 电路的图像处理函数还需要做进一步的处理。可按照**错误!未找到引用源。**所提示的流程来进行处理。

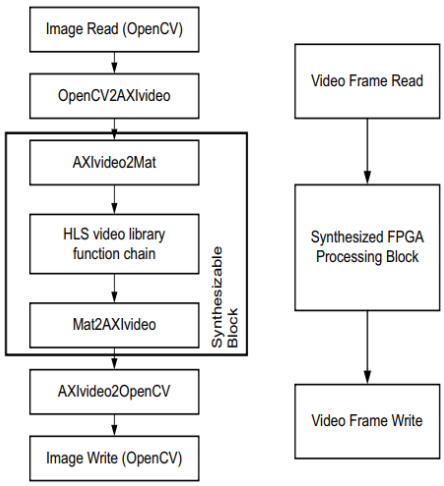


图 3- 4 HLS 处理可综合模块的流程

从**错误!未找到引用源。错误!未找到引用源。**中可以看出，可综合的模块的输入输出流都是 AXI-Stream 格式的。Xilinx 的视频处理组件通常使用一个通用的

AXI4 流协议来传输像素数据。该协议将视频像素的每一行描述为一个 AXI4 包，每一行的最后一个像素用 TLAST 信号断言。对于 AXI-Stream 协议可以参考 UG761。

https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf

从上面的流程图中，可以看出，使用 HLS 的可综合的模块时，首先需要将图像转换成编译器支持的 AXI-stream 格式的数据。在输出图像时，也需要将 AXI-stream 格式的数据转换成 HLS 软件支持的与 OpenCv 相同的数据格式。

其中，用于转换这两种数据格式的函数分别为：

IplImage 到 AXIVideo：

```
Template<int W>
void IplImage2AXIvideo(IplImage* img, hls::stream<ap_axiu<W,1,1,1> >& AXI_video_strm);
```

AXIVideo 到 IplImage：

```
template<int W>
void AXIvideo2IplImage(hls::stream<ap_axiu<W,1,1,1> >& AXI_video_strm, IplImage* img);
```

其中 hls::stream 时一个对象，可以将其理解成一水流（注：个人理解不一定准确），其中包含着 ap_axiu 这个类型的数据，ap_axiu 一个结构体，其数据成员组成了 AXI-stream 协议所需要的信号。

有了 AXi-stream 的数据流后，还需要将数据流转换到 HLS 支持的图像处理的数据结构才行。在 hls 中，也通 OpenCv 中类似，也采用 Mat 这个图像矩阵来表示图像。

通过以下两个函数，可以将 AXI-stream 格式的数据转换位 HLS 可综合的 Mat 类型。

AXIvideo 到 Mat

```
template<int W, int ROWS, int COLS, int T>
```

```
int AXIvideo2Mat(stream<ap_axiu<W,1,1,1> >& AXI_video_strm,Mat<ROWS, COLS, T>& img);
```

Mat 到 AXIvideo

```
template<int W, int ROWS, int COLS, int T>
int Mat2AXIvideo(Mat<ROWS, COLS, T>& img,stream<ap_axiu<W,1,1,1> >& AXI_video_strm);
```

在 HLS 较为简单的方法来区分可综合和不可综合的函数，就是看函数前的作用域，若是以 hls::function() 开头以 hls 那么该函数一般是可综合的，若是以 cv 开头，那么该函数一般不可综合。

3.4 灰度转换实例

接下来以一个图片的灰度转换，来对上面所展示的 HLS 开发流程做以个简单的演练，搭建一个图像算法的仿真平台，本章实验只做仿真。

3.4.1 创建 HLS 工程

打开 HLS 软件，按照上一次实验的步骤新建一个工程；

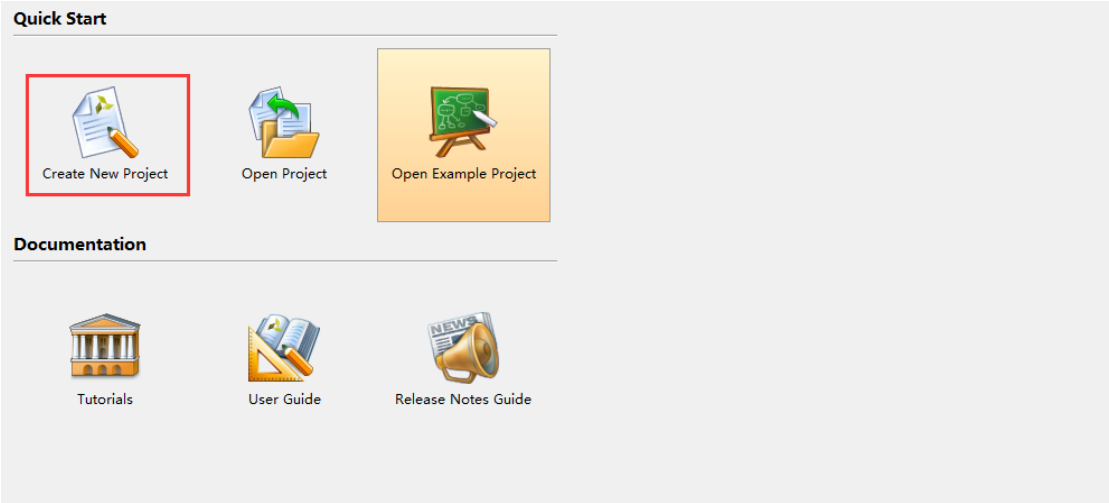


图 3- 5

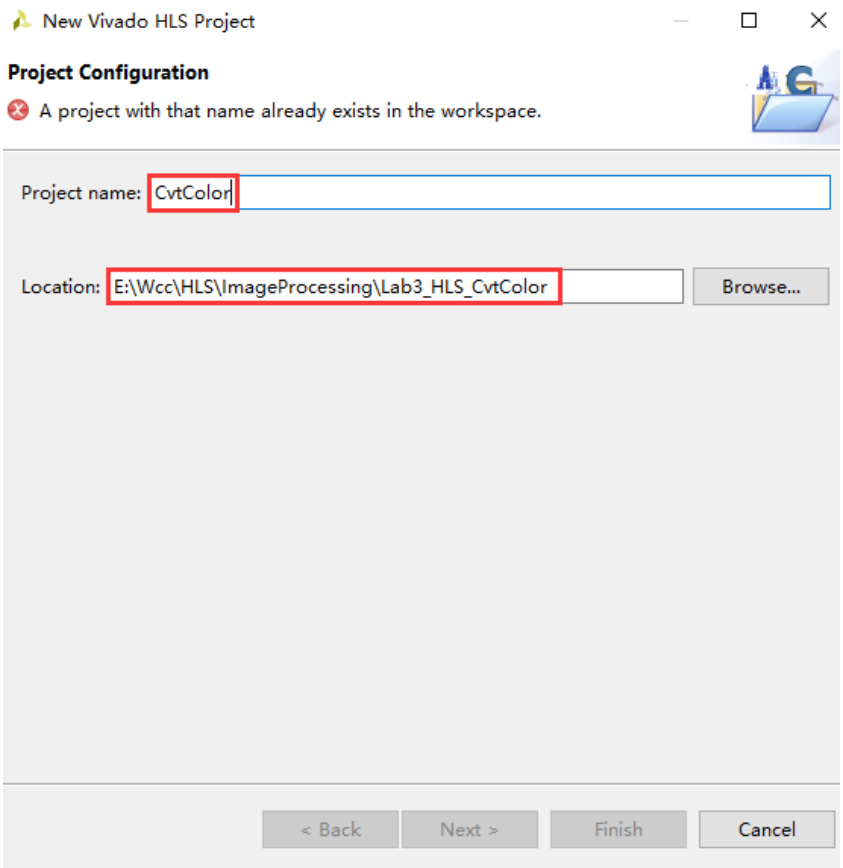


图 3- 6 选择路径与创建工程

选择对应的元器件：购买 Z7-Lite7020 的用户选中我们开发板上的器件 xc7z020clg400-2，购买 Z7-Lite7010 的用户选中我们开发板上的器件 xc7z010clg400-1

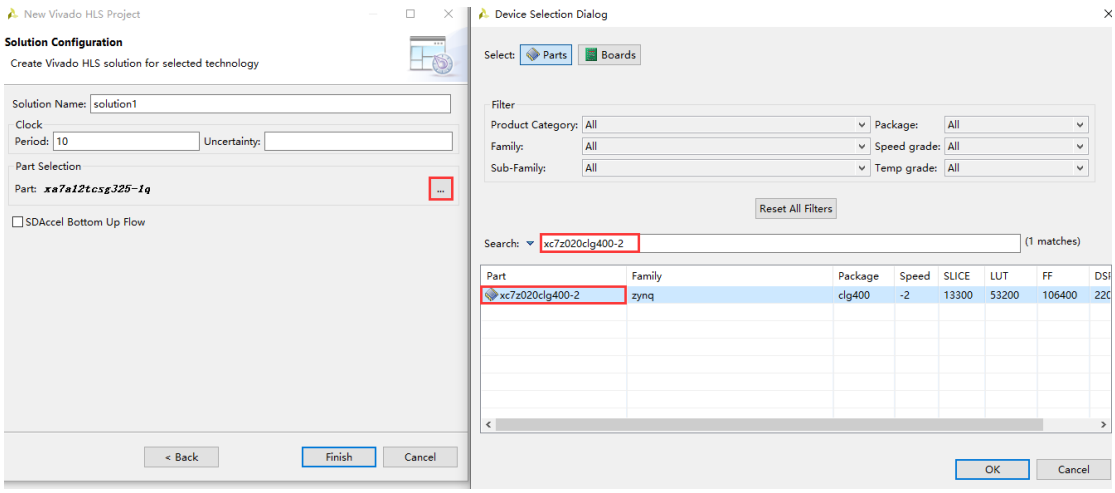


图 3- 7 选择元器件

3.4.2 向工程中添加文件

右键 Source 新建源文件和头文件

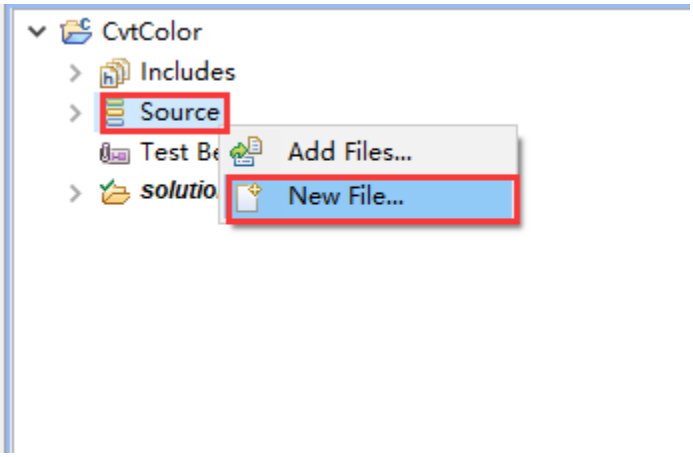


图 3- 8

首先添加头文件，头文件中，定义了 AXI-stream 的流格式，图像矩阵，声明了可综合的函数。

```
#ifndef __CVT_COLOR_HEAD_H_
#define __CVT_COLOR_HEAD_H_

#include "hls_video.h"
#include "ap_axi_sdata.h"

//定义图像的大小
#define MAX_COLS 1920
#define MAX_ROWS 1080

//定义输入输出图像
#define SRC_IMAGE "SrcImage.bmp"//输入图像路径
#define DST_IMAGE "DstImage.bmp"//输出图像路径
#define GOLD_IMAGE "GoldImage.bmp"//参考图像路径

//定义图像数据类型

//定义 AXI-Stream 数据流
```

```
19 typedef hls::stream<ap_axiu<32,1,1,1> > AXI_STREAM ;
20
21 //定义图像矩阵，其中图像矩阵中的像素格式为3 通道8 位无符号数，RGB 图像
22 typedef hls::Mat<MAX_ROWS,MAX_COLS,HLS_8UC3> IMAGE_RGB;
23
24 //定义图像矩阵的每个像素格式，3 通道8 位无符号型
25 typedef hls::Scalar<3,unsigned char> PIXEL_RGB;
26
27 //定义图像矩阵，其中图像矩阵中的像素格式为1 通道8 位无符号数，灰度图像
28 typedef hls::Mat<MAX_ROWS,MAX_COLS,HLS_8UC1> IMAGE_GRAY;
29
30 //定义图像矩阵的每个像素格式，1 通道8 位无符号型
31 typedef hls::Scalar<1,unsigned char> PIXEL_GRAY;
32
33
34 //top function
35 void image_filter(AXI_STREAM & inStream,AXI_STREAM & outStream,   int rows,int cols);
36 #endif
```

第 1、2 行，防止头文件重复调用；4、5 行引用了 HLS 图像处理的库和定义 stream 流中结构体的头文件；8、9 行定义了图像的大小；11 到 14 行定义了输入输出图像和黄金参考图像的路径；19 从定义了 AXI-Stream 的类型；21~30 行重定义了 RGB 图像矩阵，灰度图像矩阵，RGB 图像像素，灰度图像像素；35 行声明了顶层函数。

在源文件中，定义了对声明的函数进行了实现，具体代码入下表所示：

```
1   #include "ChangeColor.h"
2
3   void image_filter(AXI_STREAM & inStream,AXI_STREAM & outStream,   int rows,int cols){
4   #pragma HLS INTERFACE s_axilite port=cols bundle=CTRL_BUS
5   #pragma HLS INTERFACE s_axilite port=rows bundle=CTRL_BUS
6   #pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
7   #pragma HLS INTERFACE ap_stable port=rows
8   #pragma HLS INTERFACE ap_stable port=cols
9   #pragma HLS INTERFACE axis register port=outStream
10 #pragma HLS INTERFACE axis register both port=inStream
11   IMAGE_RGB img_0(rows,cols);
12   IMAGE_GRAY img_1(rows,cols);
```

```
13    IMAGE_RGB img_2(rows,cols);
14 #pragma HLS dataflow
15    hls::AXIvideo2Mat(inStream, img_0);
16    hls::CvtColor<HLS_BGR2GRAY>(img_0,img_1);
17    hls::CvtColor<HLS_GRAY2BGR>(img_1,img_2);
18    hls::Mat2AXIvideo(img_2,outStream);
19 }
```

第一行，引入了刚才创建的头文件，3~19 行是对我们声明的函数的实现。其中 4~10 行是添加的接口(INTERFACE)类型的 Directive，在上一章的内容中已经介绍了如何添加 Directive，在这一章中，将介绍如何添加上述 Directives；11 到 13 行定义了 3 个变量，其中 img_0 和 img_2 是 RGB 类型的图像矩阵，img_1 是 GRAY 类型的图像矩阵。15，18 行的作用类似，目的是进行图像矩阵和 AXI-stream 之间的转换，是 hls 中的可综合的函数能够处理传输进来的数据。16，17 行完成的是图像色彩的转换，通过 hls::CvtClolor 来进行图像的转换，函数的定义和用法可以参考 UG902 中的介绍，主要是将一种类型的图像转换成另一种类型的图像。

接下来介绍，如何添加这些 Directives，首先添加 AXI-Stream 类型接口的 Directive，已经知道输入输出的图像是 AXI-Stream 类型，在 Directive 窗口中，选中函数接口中的变量，然后右键选择添加 Directive，这里以 inStream 位例：

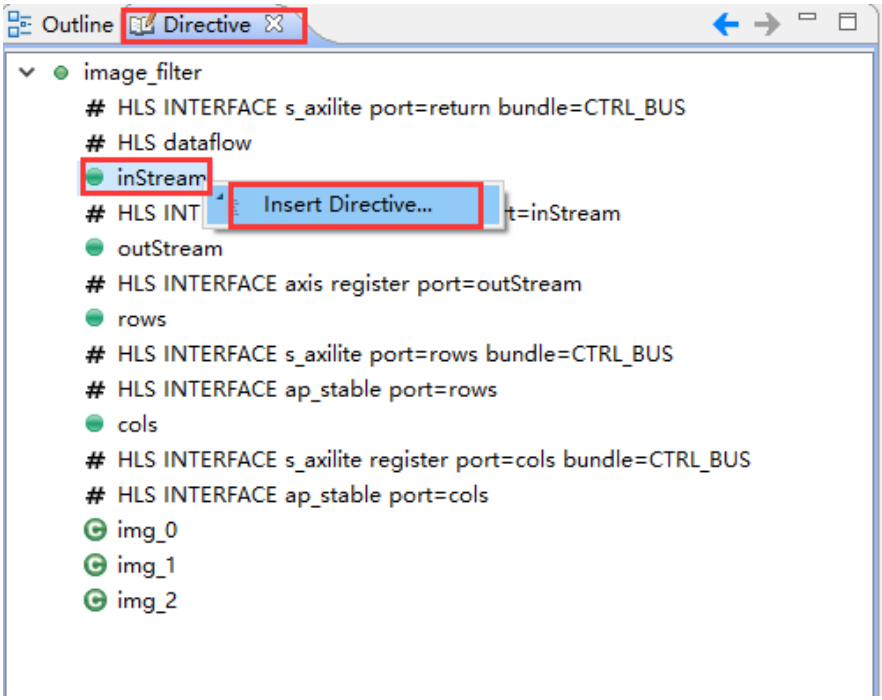


图 3- 9

在弹出界面中按照，Directive 类型选择接口类型，添加到源文件中，模式悬着 axis(即 AXI4-stream)然后点击 OK，就可以在源文件中，看到已经添加了 AXI-Stream 类型的接口约束。

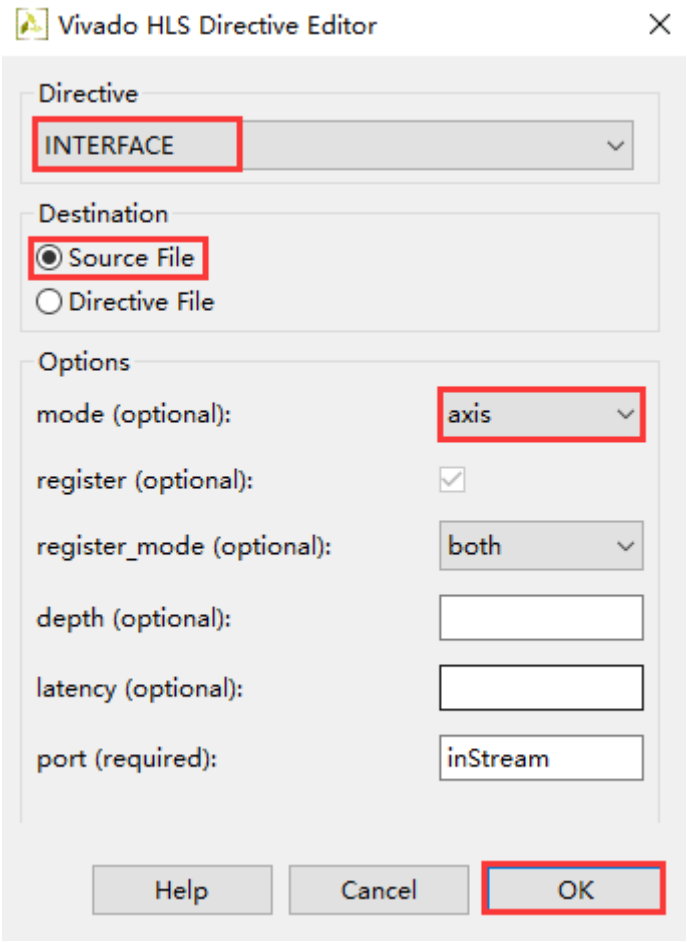


图 3- 10 添加 Directive

然后对基本类型进行约束，这些变量一般作为控制信号，在 HLS 常常将其作为 s_axilite 接口，该接口支持的协议是 AXI-Lite 协议，每次能够传输的数据量较小，场用于控制模块。我们按照先前介绍的方法，选中 int 类型的变量来添加 Directive。Option 中选择 s_axilite，在 bundle 中填入 CTRL_BUS 表明该信号是一个控制信号。对于输入的 rows 和 cols。还希望在模块工作时其保持稳定，因此还需添加约束 ap_stable 的约束，让其保持稳定。

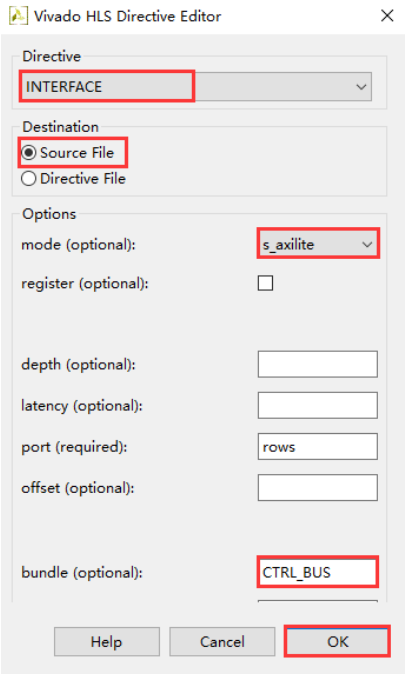


图 3- 11

可以看到，在接口处，还有一个对 return 的 Directive，但是函数时 void 类型的，没有返回值，这里添加 return 的 Directive 有什么作用呢？这是因为对 return 添加 s_axilite 的 Directive 后，在 SDK 中，能够为生成的 IP 提供配置信息和控制信号。就如同在 SDK 开发阶段需要对外设进行初始化，对 return 添加 Directive 后，SDK 中就能提供这些初始化和配置信号的驱动，便于我们在 SDK 进行开发。对 Return 的 Directive 添加方式：

最后，还可以看到一个 Directive #pragma HLS dataflow，这一个 Directive 不是接口类型的 Directive，这是一个用于优化函数的 Directive，添加了条语句，函数内部的将会并行执行，减小 Latency，提高运行速度；这个 Directive 按照添加：选中函数，右键选中 Directive 为 DATAFLOW。

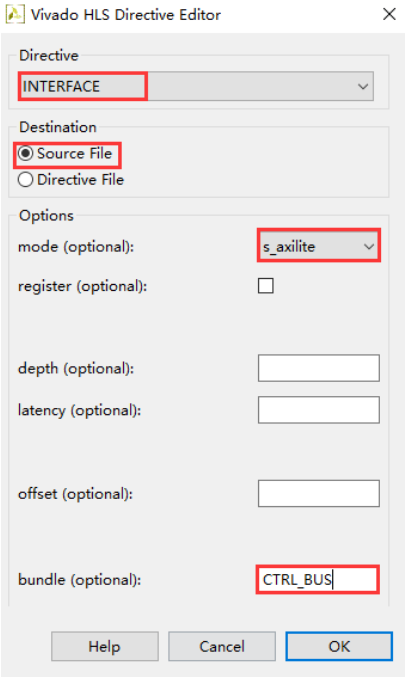


图 3- 12

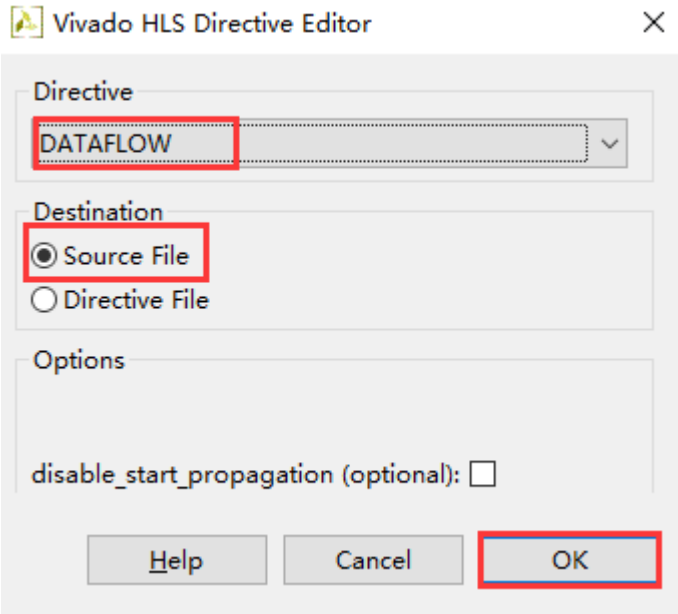


图 3- 13 添加 DataFlow

3.4.3 添加仿真文件

接下来添加仿真文件，在 Test Bench 栏右键新建文件：

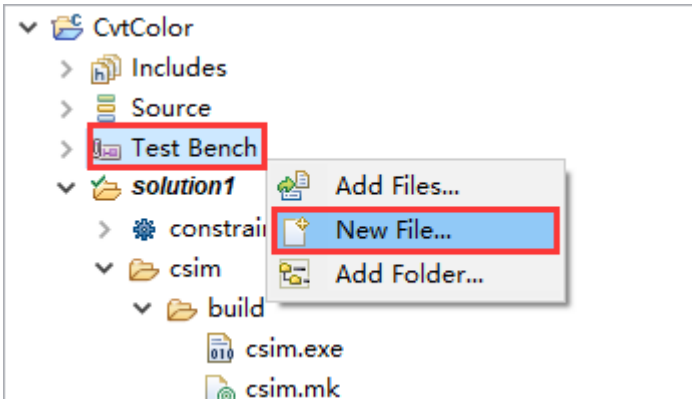


图 3- 14 添加仿真文件

仿真文件内容如下表：

```
1  #include <stdio.h>
2  #include <opencv2/opencv.hpp>
3  #include "ChangeColor.h"
4  #include "hls_opencv.h"
5  using namespace cv ;
6
7
8  //图片比对
9  int image_compare(const char* output_image, const char* golden_image) {
10     if (!(output_image) || !(golden_image)) {
11         printf("Failed to open images...exiting.\n");
12         return -1;
13     } else {
14         Mat o = imread(output_image);
15         Mat g = imread(golden_image);
16         assert(o.rows == g.rows && o.cols == g.cols);//assert 如果正确，则继续运行后面的程序，否则
报错
17         assert(o.channels() == g.channels() && o.depth() == g.depth());
18           printf("rows = %d, cols = %d, channels = %d, depth = %d\n", o.rows, o.cols, o.channels(),
o.depth());
19         int flag = 0;
20         for (int i = 0; i < o.rows && flag == 0; i++) {
21             for (int j = 0; j < o.cols && flag == 0; j++) {
22                 for (int k = 0; k < o.channels(); k++) {
23                     unsigned char p_o = (unsigned char)*(o.data + o.step[0]*i + o.step[1]*j + k);
24                     unsigned char p_g = (unsigned char)*(g.data + g.step[0]*i + g.step[1]*j + k);
25                     if (p_o != p_g) {
```

```
26                                  printf("First mismatch found at row = %d, col = %d\n", i, j);
27                                  printf("(channel%2d) output:%5d, golden:%5d\n", k, p_o, p_g);
28                                  flag = 1;
29                                  break;
30                              }
31                          }
32                      }
33                  }
34                  if (flag)
35                      printf("Test Failed!\n");
36                  else
37                      printf("Test Passed!\n");
38
39                  return flag;
40              }
41 }
42
43 int main(){
44     IplImage * SrcImage;//输入的图像
45     IplImage * DstImage;//输出的图像
46     IplImage * GoldImage;//参考图像
47     IplImage * GoldImageTmp;//参考图像
48     SrcImage = cvLoadImage(SRC_IMAGE,-1);//导入图像
49
50     //创建 opencv 支持的输出的图像
51     DstImage = cvCreateImage(cvGetSize(SrcImage),SrcImage->depth, SrcImage->nChannels);
52     GoldImage = cvCreateImage(cvGetSize(SrcImage),SrcImage->depth, SrcImage->nChannels);
53     GoldImageTmp = cvCreateImage(cvGetSize(SrcImage),SrcImage->depth, 1);
54     //AXI-Stream 数据流
55     AXI_STREAM inStream;
56     AXI_STREAM outStream;
57     IplImage2AXIvideo(SrcImage,inStream);
58
59     //可综合的函数
60     image_filter(inStream,outStream,SrcImage->height, SrcImage->width);
61     //保存输出结果
62     AXIvideo2IplImage(outStream,DstImage);
63     cvSaveImage(DST_IMAGE,DstImage);
64
65     //参考图像输出
66     cvCvtColor(SrcImage, GoldImageTmp, CV_BGR2GRAY);
```

```
67    cvCvtColor(GoldImageTmp, GoldImage, CV_GRAY2BGR);
68    cvReleaseImage(&GoldImageTmp);
69    cvSaveImage(GOLD_IMAGE, GoldImage);
70    cvShowImage(DST_IMAGE, DstImage);
71    cvWaitKey(0);
72
73    //比较黄金参考图片和 HLS 可综合执行结果
74    image_compare(DST_IMAGE,GOLD_IMAGE);
75    //释放图片
76    cvReleaseImage(&SrcImage);
77    cvReleaseImage(&DstImage);
78    cvReleaseImage(&GoldImage);
79    return 0;
80 }
```

1~4 行引入所需要用到的头文件，5 行使用 cv 这个命名空间。9~41 行定义了一个函数，该函数的作用是用来比较用通过使用 HLS 输出的图像和使用 OpenCv 生成的黄金参考图像是否完全一致。

43~80 行是本次方针的 main 函数，可以看到 Testbench 中的仿真流程，就是按照图中的类容进行的，首先从外界导入图片，导入图片是 IplImage 这个类型的：48 行；然后将 IplImage 类型的数据转换为 HLS 可综合模块支持的 AXI-stream 格式的数据：57 行；再调用可综合模块的顶层函数并给出数据流：60 行，函数调用完毕后，再将 AXI-Stream 格式的数据转换为 IplImage 格式，将生成的图片保存：62~63 行；除了调用可综合模块进行验证外，还使用 OpenCv 里的函数生成了一个黄金参考图片：66~69 行；比较黄金参考图像和可综合模块生成的图像有何不同，如果不同，打印出第一个出现不同的像素点位置。最终释放掉占用的内存。

3.4.4 仿真验证

选择一张图片，将其命名为 SrcImage.bmp 存放再与源代码路径相同的文件夹中。

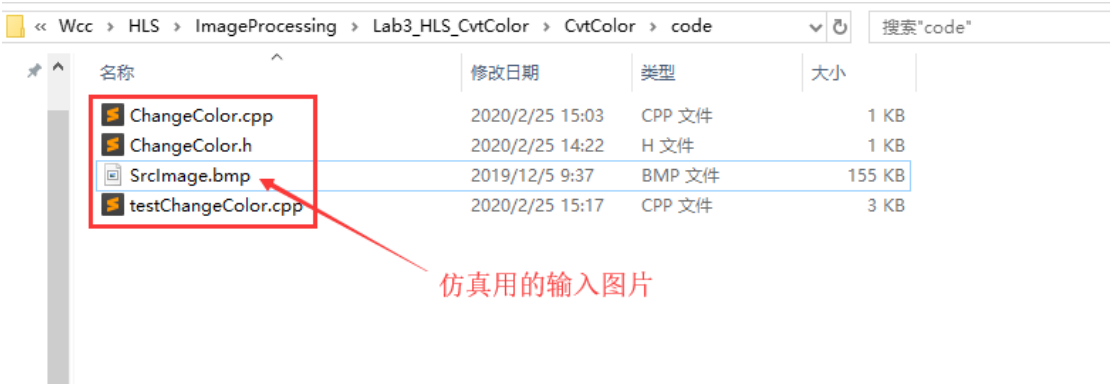


图 3- 15

点击 C 仿真开始运行仿真，仿真结束过后，可以看到 csim→bulid 文件夹中生成了两幅图像，可以打开查看这两幅图像。

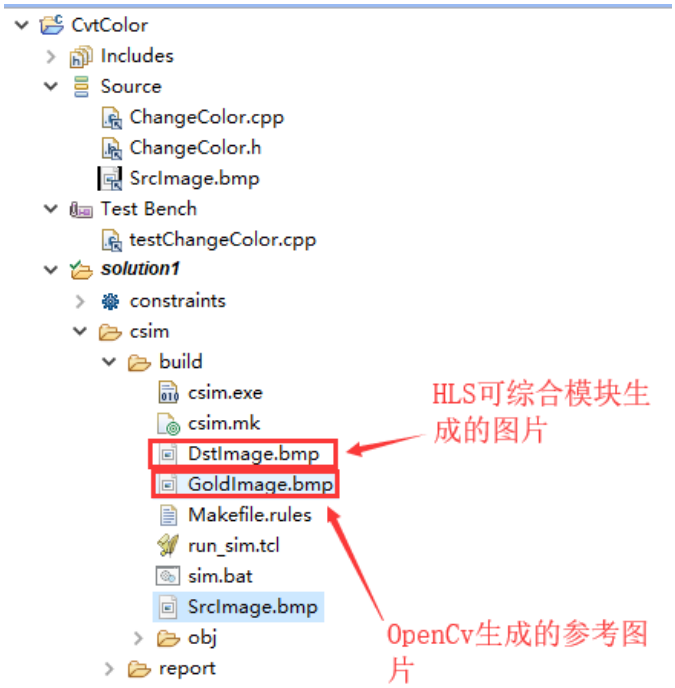


图 3- 16

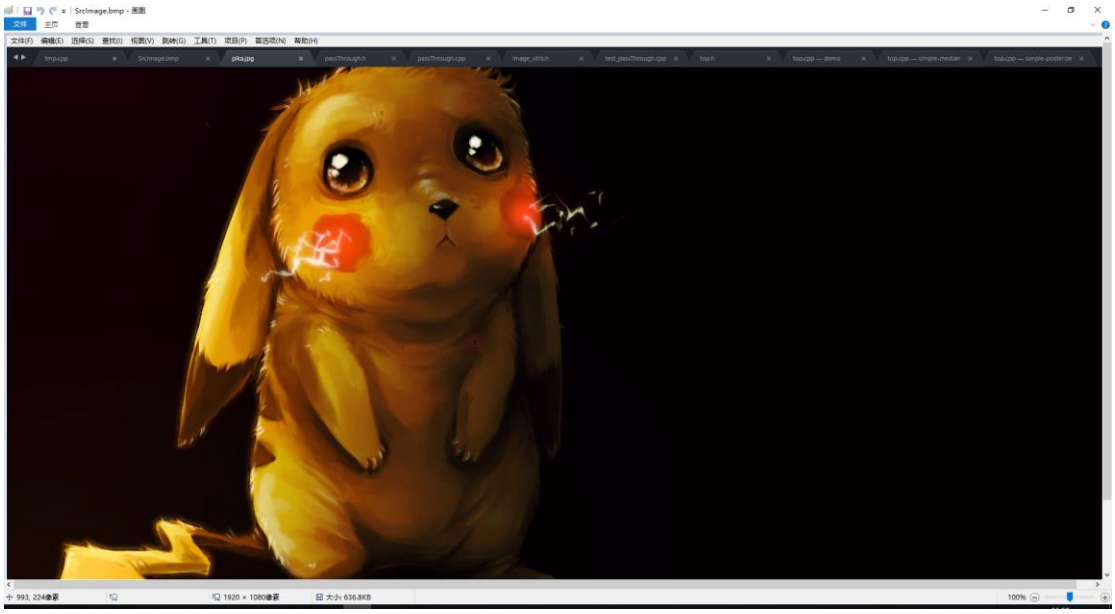


图 3- 17 皮卡丘原图

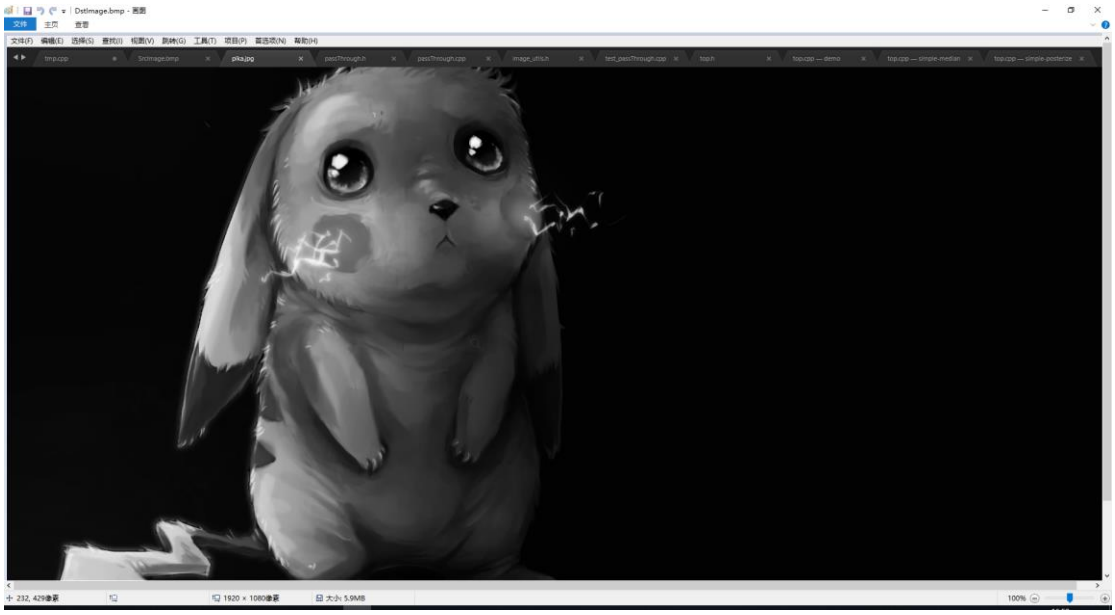


图 3- 18 通过 HLS 可综合模块生成的图像

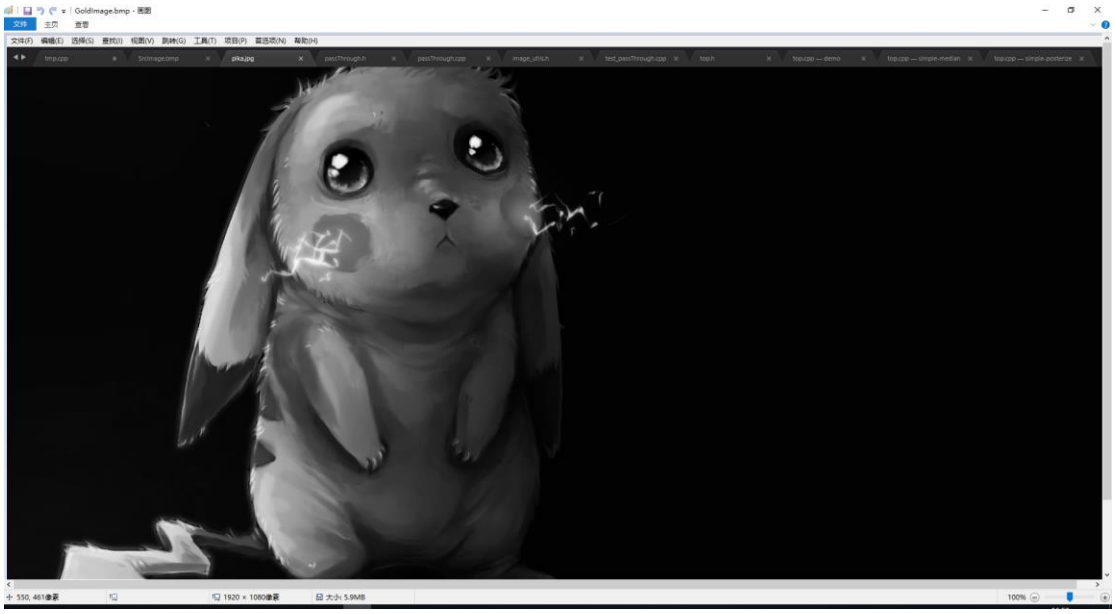


图 3- 19 黄金参考图片

通过观察两幅生成的图片，可以看到两者几乎没有差距，但是观察 Console 打印出来的信息，却可以发现两幅图片并不一样：

ChangeColor.cpp | ChangeColor.h | testChangeColor.cpp | _csim.log

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../code/testChangeColor.cpp in debug mode
   Compiling ../../../code/ChangeColor.cpp in debug mode
   Generating csim.exe
rows = 1080, cols = 1920, channels = 3, depth = 0
First mismatch found at row = 66, col = 498
(channel 0) output:    4, golden:    5
Test Failed!
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

图 3- 20

出现这种原因，是因为 HLS 的支持的可综合模块的图像处理函数与 OpenCv 库中的函数，精度不一致造成的。通过 beyond compare 可以观察到两幅图像之间的差异，观察报告可以得出，只有 35 个像素点之间有差异。

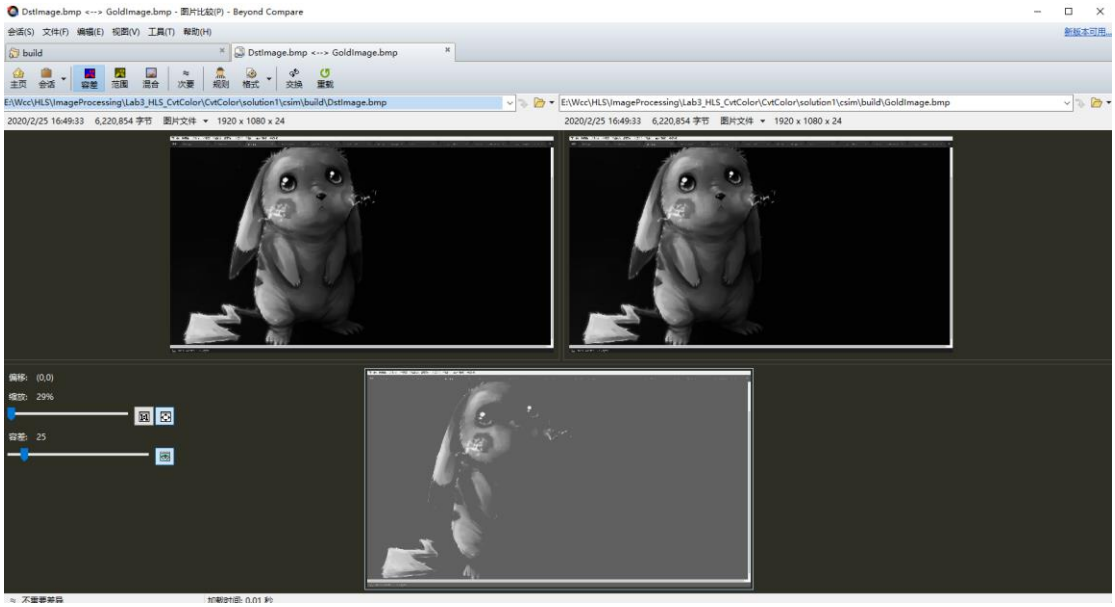


图 3- 21 使用 beyond compare 比较的图片

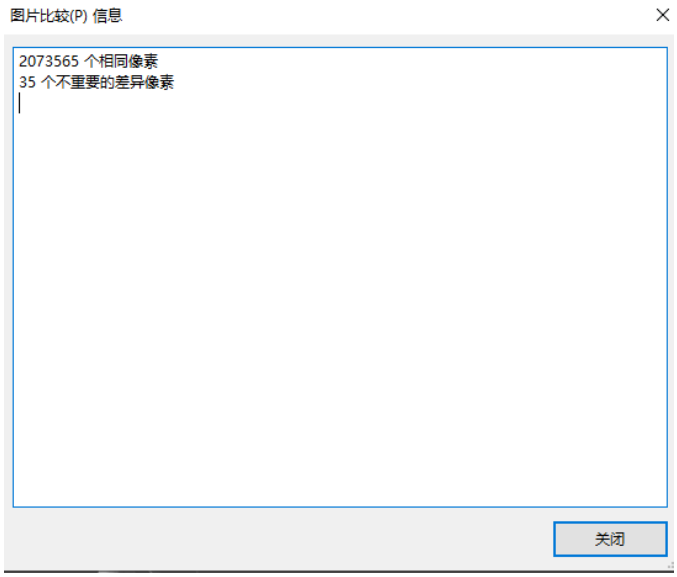


图 3- 22 比较结果

3.5 实验总结

本次实验主要介绍了 HLS 开发图像处理算法的流程，OpenCv 中基本函数的操作。通过本次是呀，搭建起了一个图像算法处理的验证平台，这个验证平台，在之后的章节中还会多次使用到。

本章中，需要注意可综合的 HLS 代码，与仿真时用到的代码的区别，其实思路就跟 FPGA 开发中的仿真是一样的。做完这章实验，其实最重要的就是掌握一个流程，根据**错误!未找到引用源。**所展示的那样，按照标准的路线去进行算法的开发。

第4章　HLS 中的浮点运算与 BRAM 接口

定点比浮点算法的 FPGA 实现要更快且效率更高，但是往往有时也需要浮点数来实现，这是因为定点数只有有限的数据动态范围，需要深入的分析来决定整个设计的中间数据位宽的变化模式，为了达到优化 QoR，需要引入很多不同类型的定点数作为中间变量，而浮点数具有更大的动态数据范围，从而在很多算法中只需要一种数据类型即可。

在本次实验中，调用函数库的数学函数，来展示如何在 HLS 进行浮点运算，并将生成的 IP 设计成为 BRAM 接口，生成一个可以用于浮点运算的 IP 可以在 Vivado 中进行调用。

完成以下公式的 HLS 运算，并且生成 BRAM 接口，在 Vivado 继承工具中调用，并在 SDK 中对生成的 IP 核进行验证。该公式并无实际意义，只是作为展示如何使用 HLS 中的数学库和浮点运算做一个提示作用。

$$f(x,y) = sinx + \sqrt{exp(y)}/2$$

4.1 HLS 中实现浮点运算算法核的实现

4.1.1 添加源文件

创建 HLS 工程后，新建一个文件用于存放源文件，在源文件中，添加如下内容：

```
#include <hls_math.h>

void crazyFunction(float x[100],float y,float result[100]){
#pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
#pragma HLS INTERFACE s_axilite port=y bundle=CRTL_BUS
#pragma HLS INTERFACE bram port=x
#pragma HLS INTERFACE bram port=result
```

```
8     int idx;
9     for(idx=0;idx<100;idx++){
10 #pragma HLS PIPELINE
11          result[idx] = (hls::sin(x[idx]) + hls::sqrtf(hls::expf(y)))/2;
12    }
13 }
```

第一行引入 HLS 中支持的数学函数库，通过引入这个库，可以和 C 语言类似地去调用数学函数。第 3 行声明了本次实验的顶层函数，可以看到接口处有两个浮点型的数组和一个浮点型的数据。

第 4 到第 7 行为 Directive，其中第 4 行是为这个函数在生成 IP 时能够提供可供 SDK 中调用的接口函数，并生成该 IP 的驱动函数。

第 5 行表示将 Y 这个变量声名成 AXI_Lite 接口，可以作为参数传递进来，改变函数。

第 6、7 行将两个数组声明为 BRAM 接口，其接口处连接这 BRAM，在 Vivado 集成环境中，可用以连接到 BRAM。

第 10 行是添加流水线的 Directive，能够对 for 循环进行优化，减小 Latency。

第 11 行调用库函数完成本次实验函数的主要功能。

以其中一个 Directive 来演示如何添加 BRAM 接口类型的 Directive：

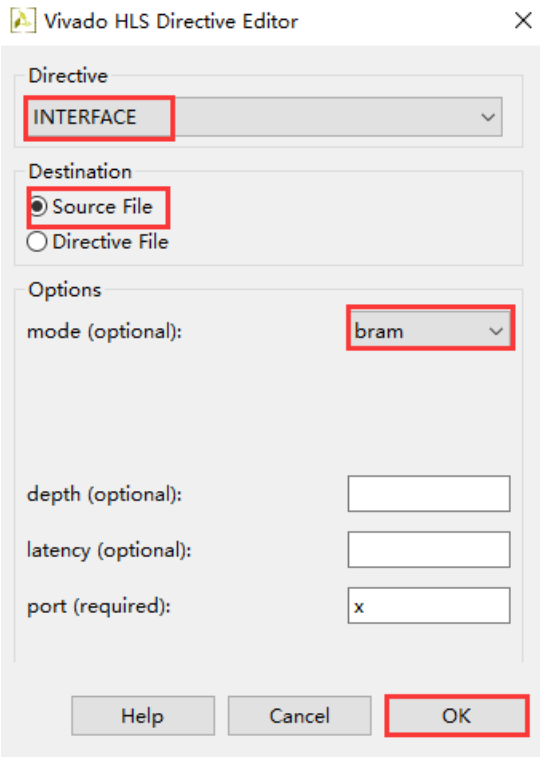


图 4- 23

4.1.2 添加 C 仿真文件

接下来添加 C 仿真文件用于验证顶层函数是否能够正确工作。

```
#include <stdio.h>
void crazyFunction(float x[100],float y,float result[100]);
int main(){
  float xVect[100];
  int idx ;
  float y = 2.0f;
  float result[100];
  for(idx=0;idx<100;idx++){
        xVect[idx] = idx;
  }
  crazyFunction( xVect,y,result);

  for(idx=0;idx<100;idx++){
        printf("result[%d]=%f\n",idx,result[idx]);
  }
  return 0;
}
```

仿真文件也比较简单，就是将顶层函数例化进来，并且给定输入数组，来观察输出数组打印的结果。

4.1.3 运行 C 仿真

点击运行 C 仿真后，可以观察到 Console 中将结果输出了出来。

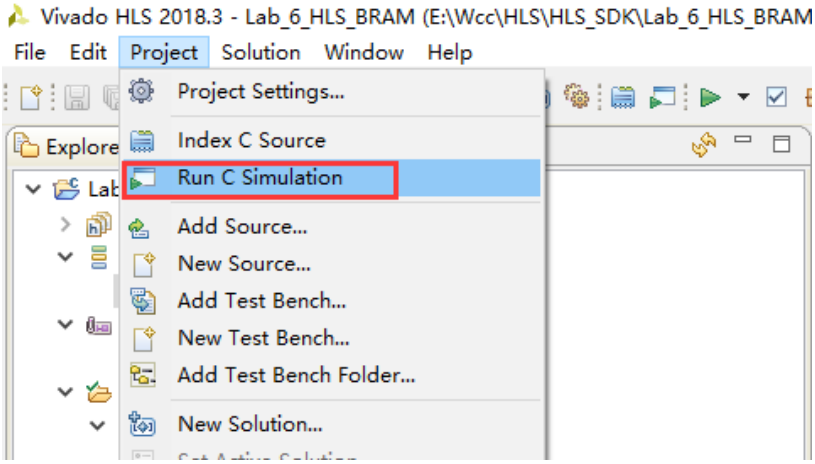


图 4- 24

运行 C 仿真的结果：

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../code/test_crazyFunction.cpp in debug mode
   Generating csim.exe
result[0]=1.359141
result[1]=1.779876
result[2]=1.813790
result[3]=1.429701
result[4]=0.980740
result[5]=0.879679
result[6]=1.219433
result[7]=1.687634
result[8]=1.853820
result[9]=1.565200
result[10]=1.087130
result[11]=0.859146
```

图 4- 25

4.1.4 进行 C 综合

首先选中要综合的顶层函数，在 Project 中选中 Project Settings，在弹出界面中选择 Synthesis 然后选择顶层函数即可；

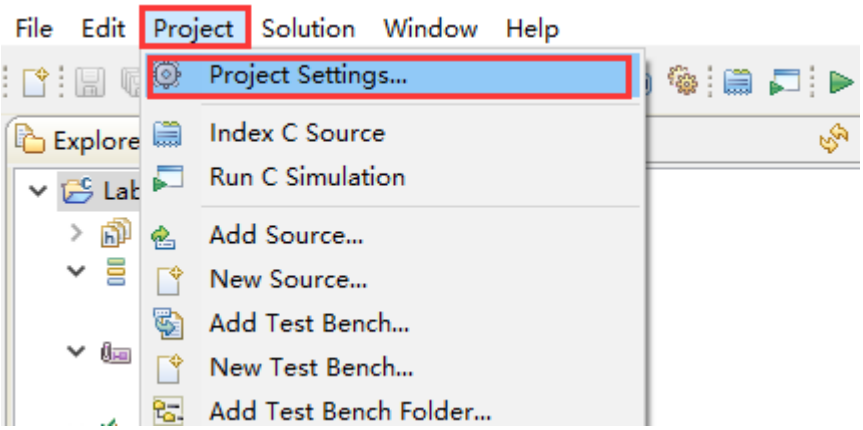


图 4- 26

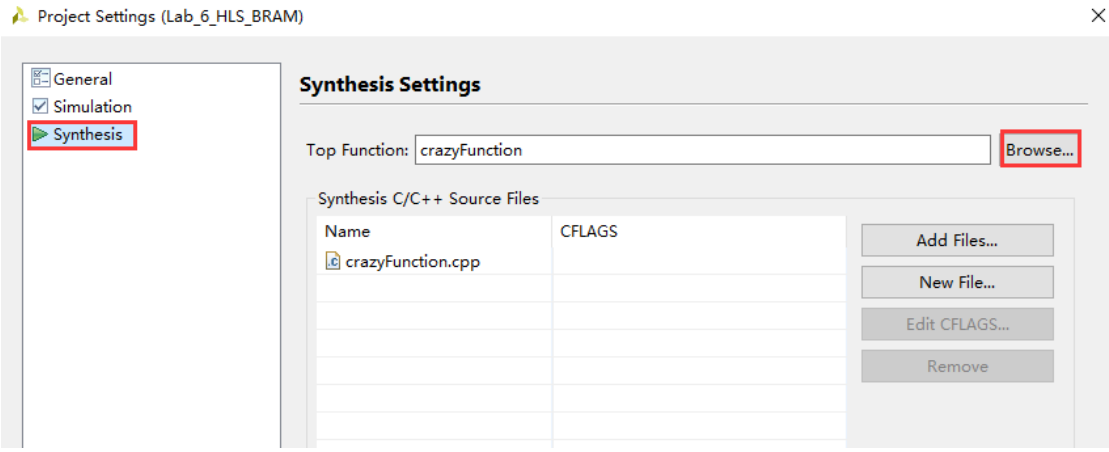


图 4- 27

设置完毕后，点击 C 综合，即可开始综合。综合完毕后，可以查看占用资源和时序总结，可以看到这里占用的延时较小。若用户感兴趣，可以将源文件中，添加流水线的 Directive 去掉对比以下两者之间在时序上的差别。使用流水线对 for 循环优化后能够明显减小 Latency。

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.682	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
126	126	126	126	none

Detail

Instance

Loop

图 4- 28 时序报告

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	14
FIFO	-	-	-	-
Instance	0	25	2937	4873
Memory	-	-	-	-
Multiplexer	-	-	-	18
Register	0	-	436	96
Total	0	25	3373	5001
Available	280	220	106400	53200
Utilization (%)	0	11	3	9

Detail

图 4- 29 资源占用

主要需要关注的是综合后的接口，：

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_CRTL_BUS_AWVALID	in	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_AWREADY	out	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_AWADDR	in	5	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_WVALID	in	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_WREADY	out	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_WDATA	in	32	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_WSTRB	in	4	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_ARVALID	in	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_ARREADY	out	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_ARADDR	in	5	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_RVALID	out	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_RREADY	in	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_RDATA	out	32	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_RRESP	out	2	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_BVALID	out	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_BREADY	in	1	s_axi	CRTL_BUS	scalar
s_axi_CRTL_BUS_BRESP	out	2	s_axi	CRTL_BUS	scalar

图 4- 30AXI_Lite 接口

x_Addr_A	out	32	bram	x	array
x_EN_A	out	1	bram	x	array
x_WEN_A	out	4	bram	x	array
x_Din_A	out	32	bram	x	array
x_Dout_A	in	32	bram	x	array
x_Clk_A	out	1	bram	x	array
x_Rst_A	out	1	bram	x	array
result_Addr_A	out	32	bram	result	array
result_EN_A	out	1	bram	result	array
result_WEN_A	out	4	bram	result	array
result_Din_A	out	32	bram	result	array
result_Dout_A	in	32	bram	result	array
result_Clk_A	out	1	bram	result	array
result_Rst_A	out	1	bram	result	array

图 4- 31 BRAM 接口

可以看到，在源文件添加的 INTERFACE 类型的 Directive 已经起到了作用，分别生成了 AXI_Lite 和 BRAM 接口类型的数据。

其中 AXI_Lite 符合 AXI 总线协议规定的信号格式，对于参数 Y 和函数的返回值都生成了对应的 AXI_Lite 接口，在将来调用时可以用来控制 IP 工作。

对于 x，和 result 接口，分别对应生成了各自的 BRAM 接口，可以将其数据存储在 BRAM 中。

4.1.5 进行联合仿真

综合完成之后，进行联合仿真，点击 solution→Run C/RTL Cosimulation

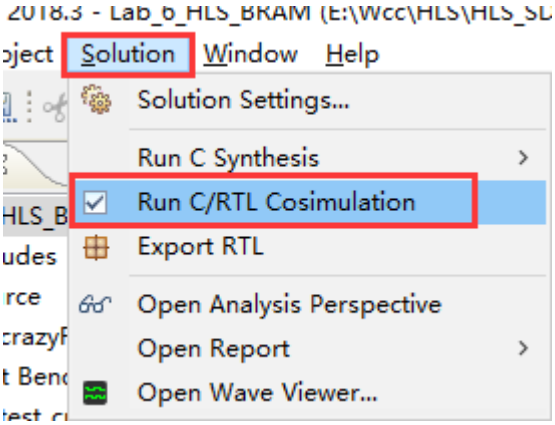


图 4- 32

HLS 编译器将会根据 C 仿真文件生成 RTL 仿真文件，并进行 RTL 级的仿真，通过 C 仿真得到的结果和 RTL 仿真的得到的结果将会展示在 Console 中。

4.1.6 导出 HLS IP

Co-simulation 完成后即可将 IP 导出，供 Vivado 集成环境调用。

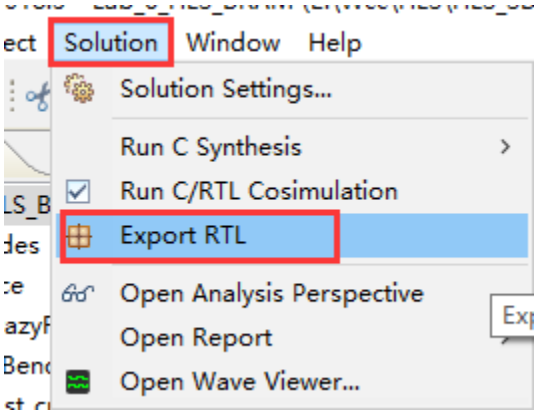


图 4- 33

4.2 搭建测试硬件平台

4.2.1 添加 HLS IP

在 Vivado IP 仓库中，将 HLS 生成的 IP，添加到 VivadoIP 仓库里，点击 Setting→IP→Repository →点击加号，然后定位到 HLS 的 solution 路径下，Vivado 将会识别出该 IP。

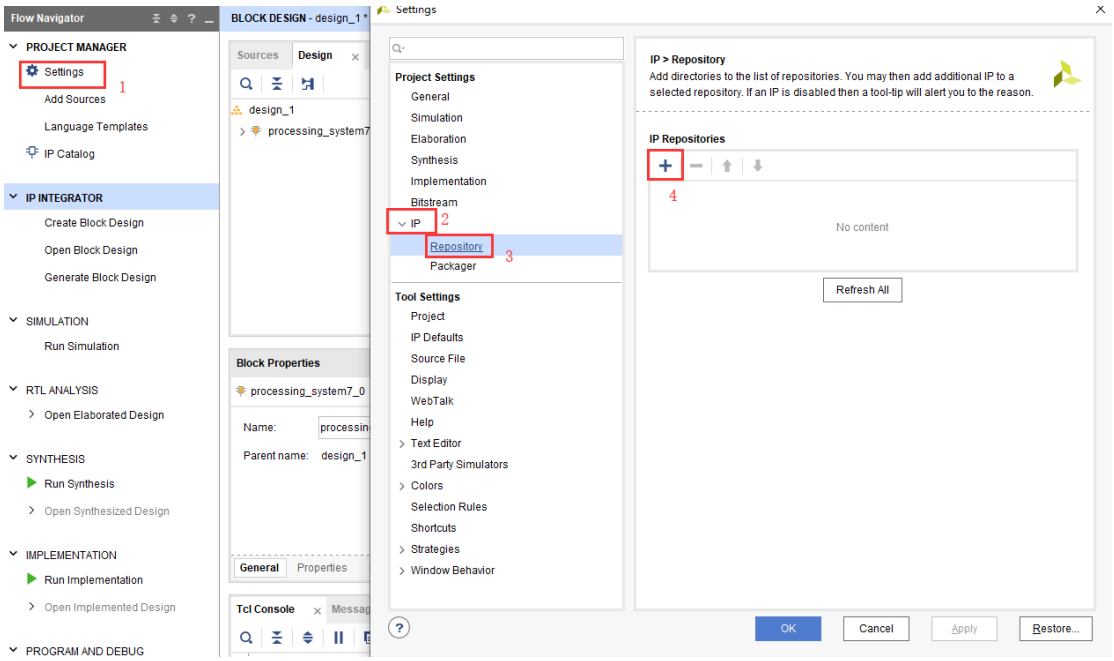


图 4- 34

4.2.2 创建 BlockDesign

点击创建 Block Design 来搭建本次实验的硬件环境。

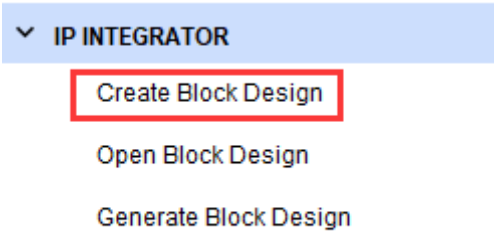


图 4- 35

添加 ZYNQ IP 核到 Block design 中双击 ZYNQ 进行配置，在 MIO Configuration 中设置 Bank1 的电压为 LVCMOS 1.8V，选中 SD0 和 UART1。

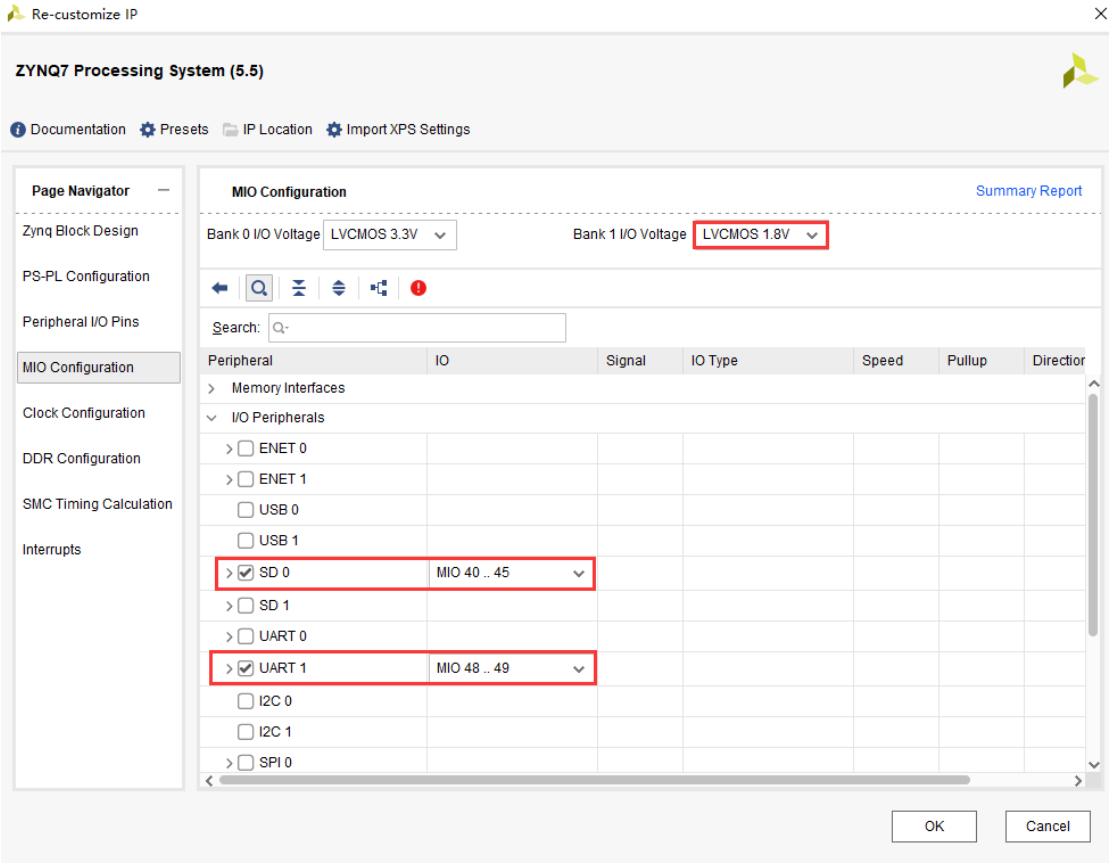


图 4- 36

在 DDR Configuration 中，选则型号：

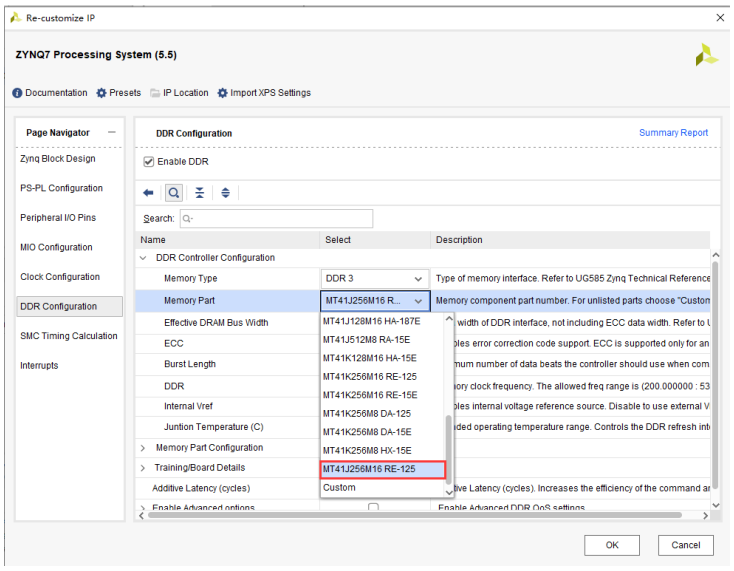
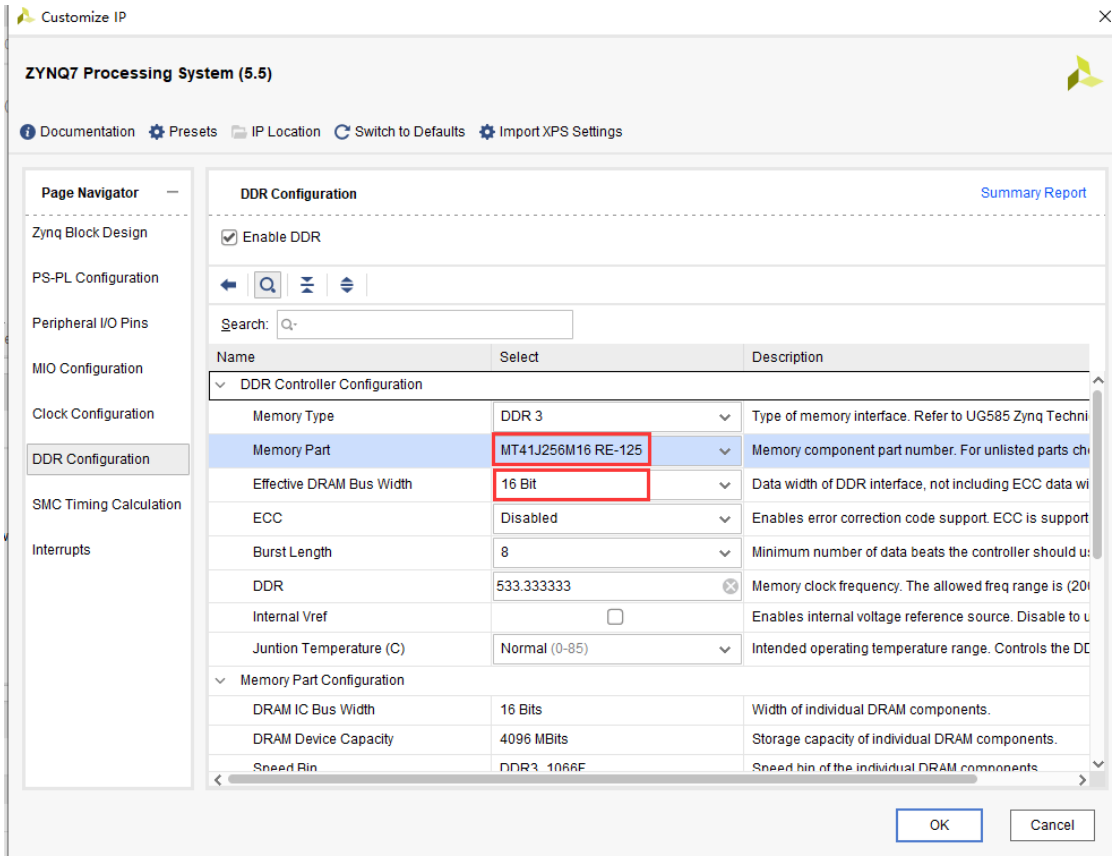


图 4- 37



接下来，添加 HLS 生成的浮点运算 IP，搜索 crazyFunction

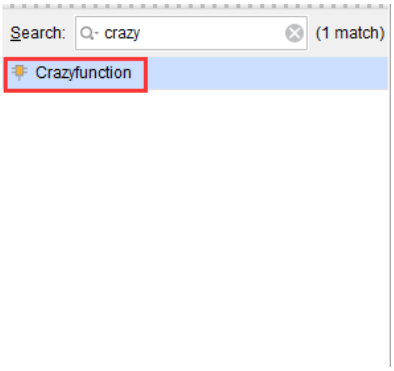


图 4- 38

点击 Run connection Automation，首先选中控制总线，点击连接，连接后如下图所示：

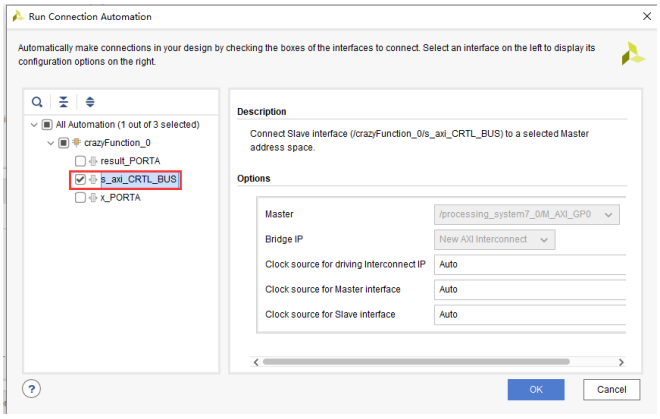


图 4- 39

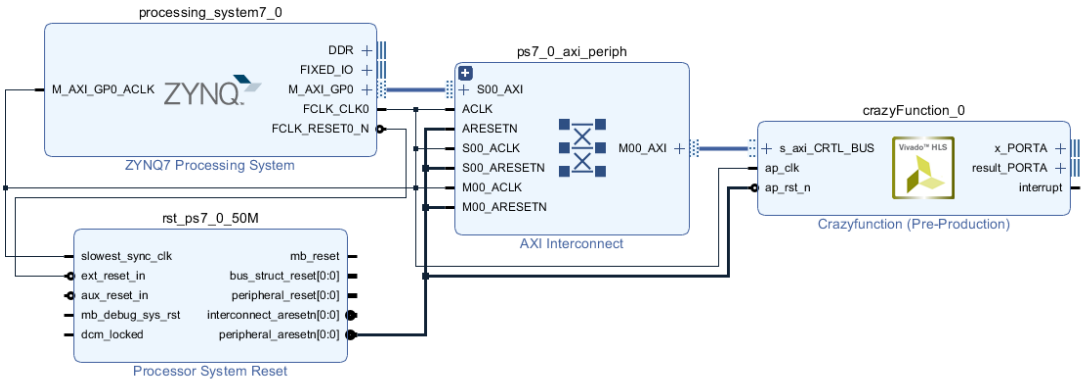


图 4- 40

再次点击 Run connection Automation，徇众其中一个接口，如下图所示：

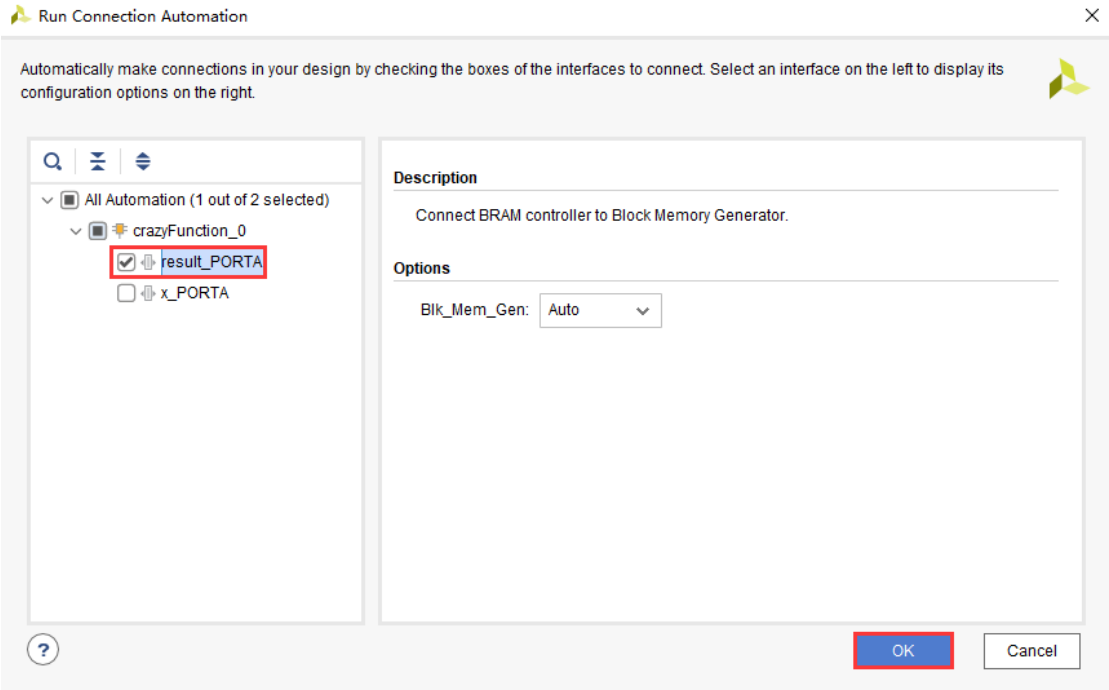


图 4- 41

可以看到自动生成了一个 BRAM，如下图所示：

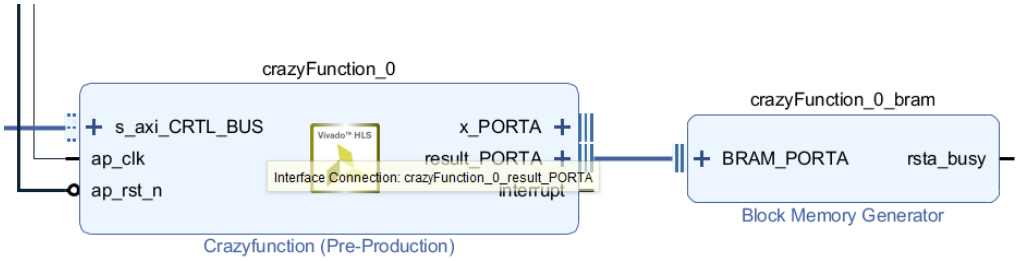


图 4- 42

点击生成的 BRAM 将其更改为双端口 RAM，如下图所示：

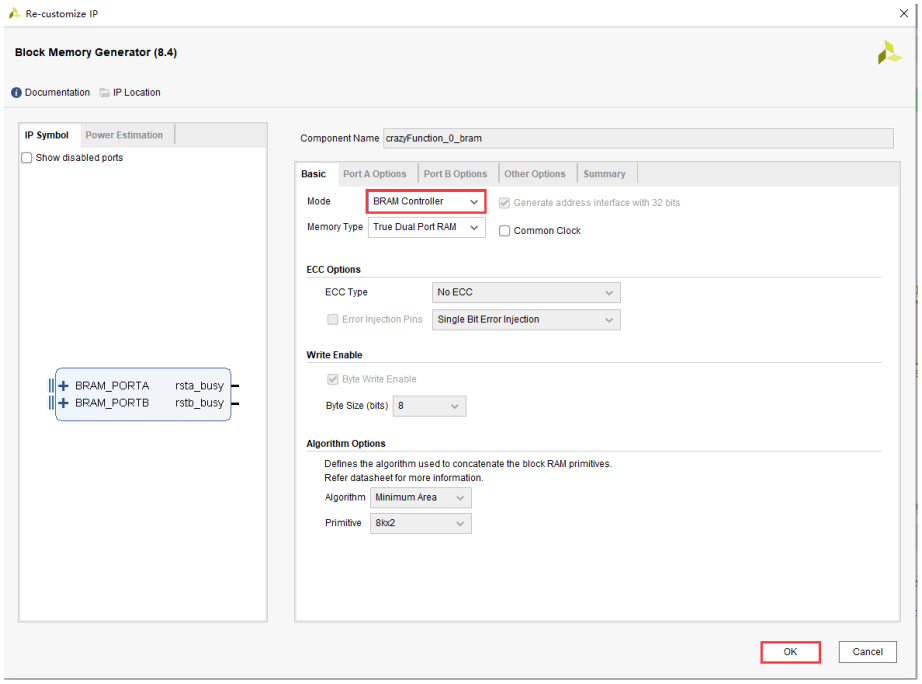
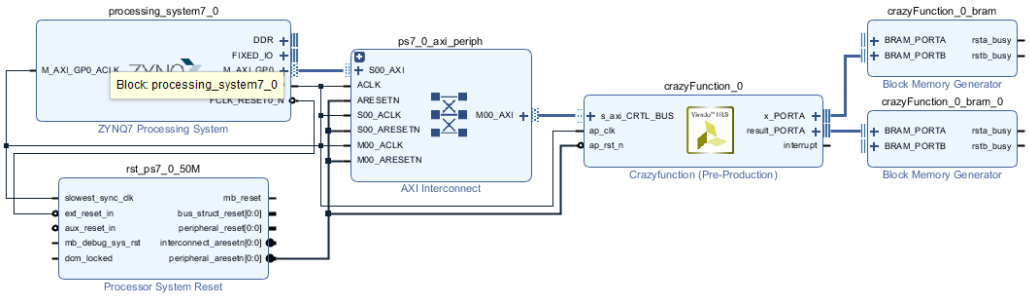


图 4- 43 双端口 RAM

对于 X-port 采用同样的方式操作，运行完后结果如下：



添加 axi_bram_controller，点击+号，搜索 bram 即可看到 axi_bram_controller

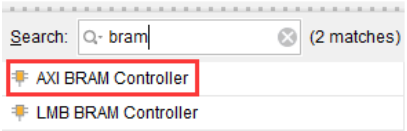


图 3- 44

然后将 axi_bram_controller 设置为一个端口

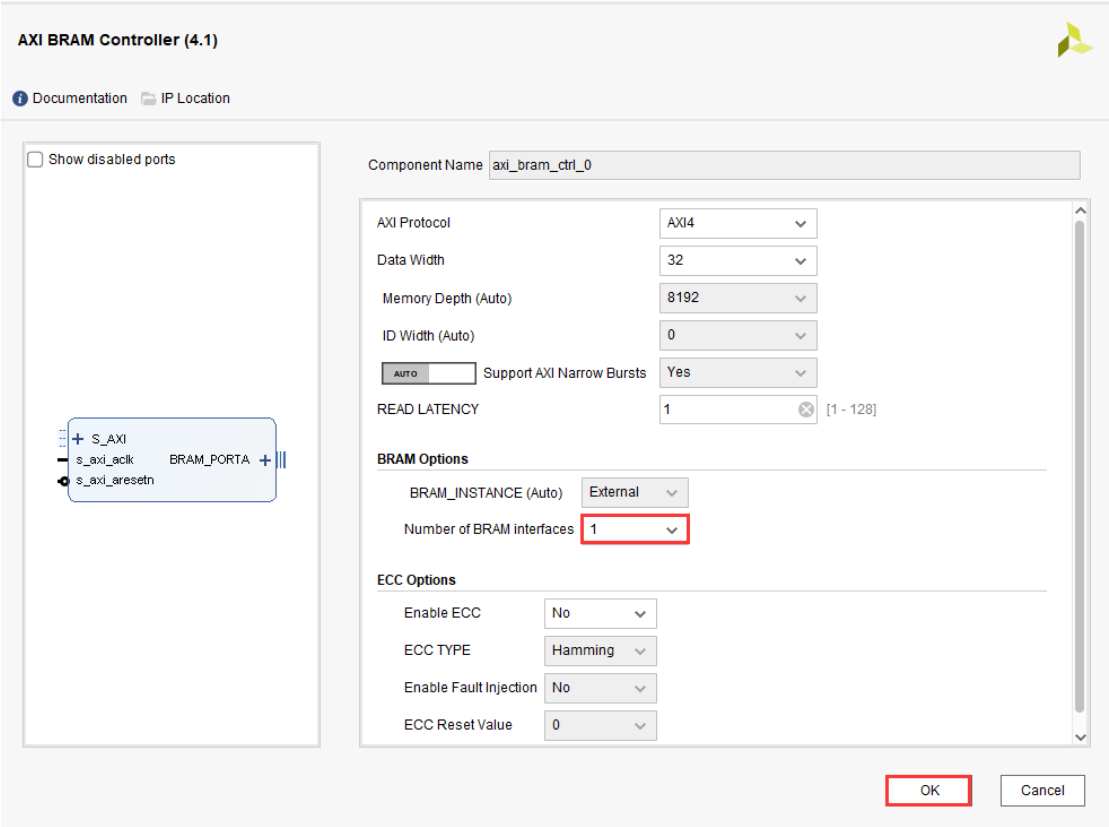


图 4- 45

添加完成后，按照如下图所示方式连接：

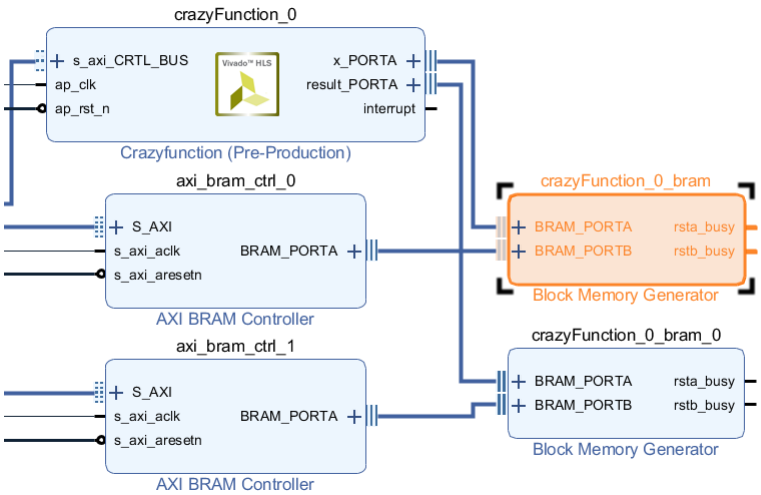


图 4- 46

然后点击 Run connection Automation，选中所有：

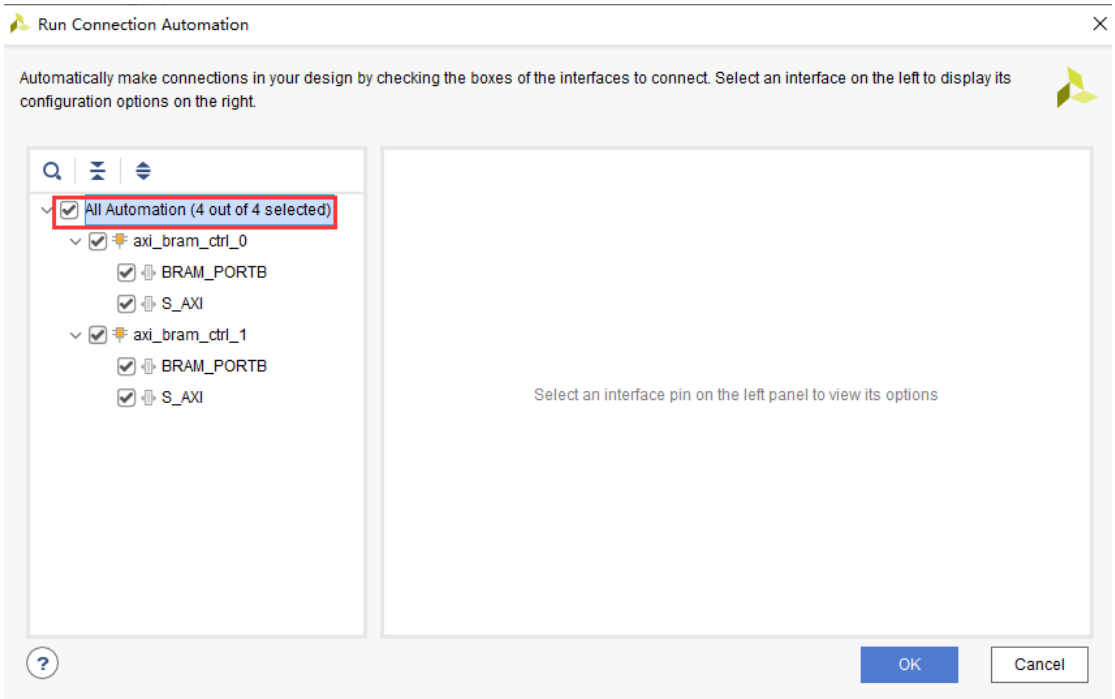


图 4- 47

最后点击 Run Block Automation，完成硬件模块的搭建：

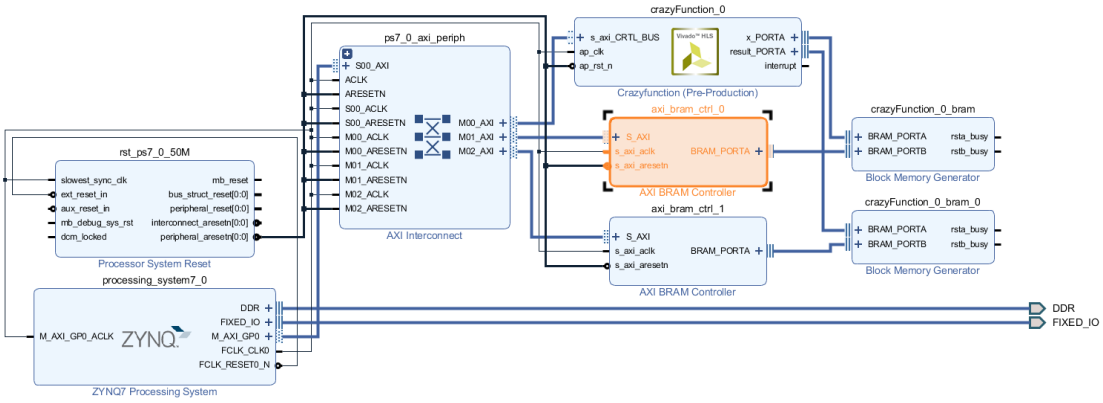


图 4- 48

生成顶层 wrapper 后，综合生成 bit 文件；

4.3 创建 SDK 工程

Bit 文件生成完毕后，导出 bit 文件，点击 export hardware→勾选 include bitstream。

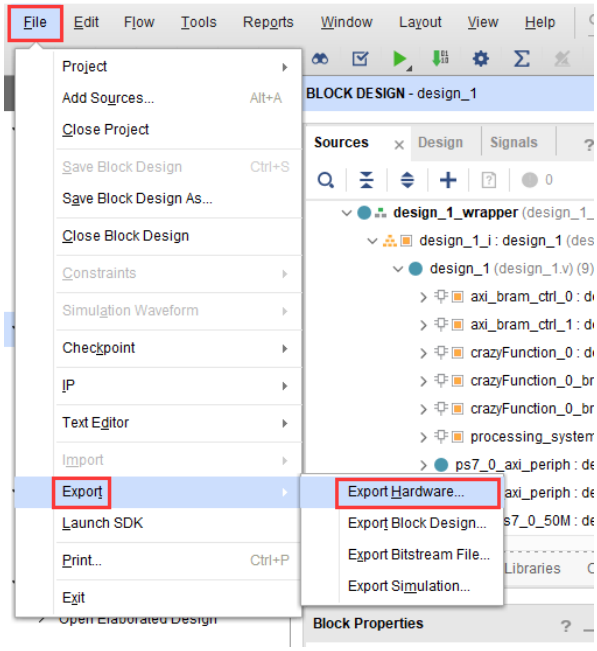


图 4- 49

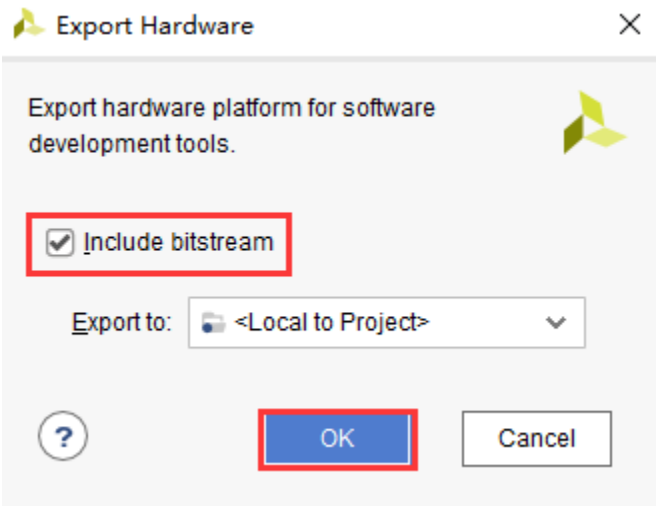


图 4- 50

然后点击 Launch SDK，打开 SDK。

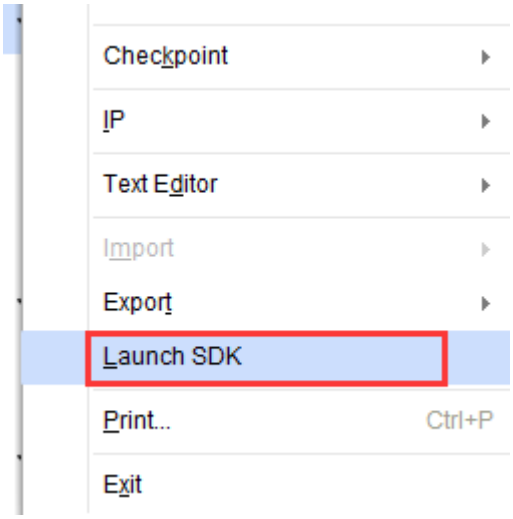


图 4- 51

进入 SDK 中，创建一个空的 C++工程：

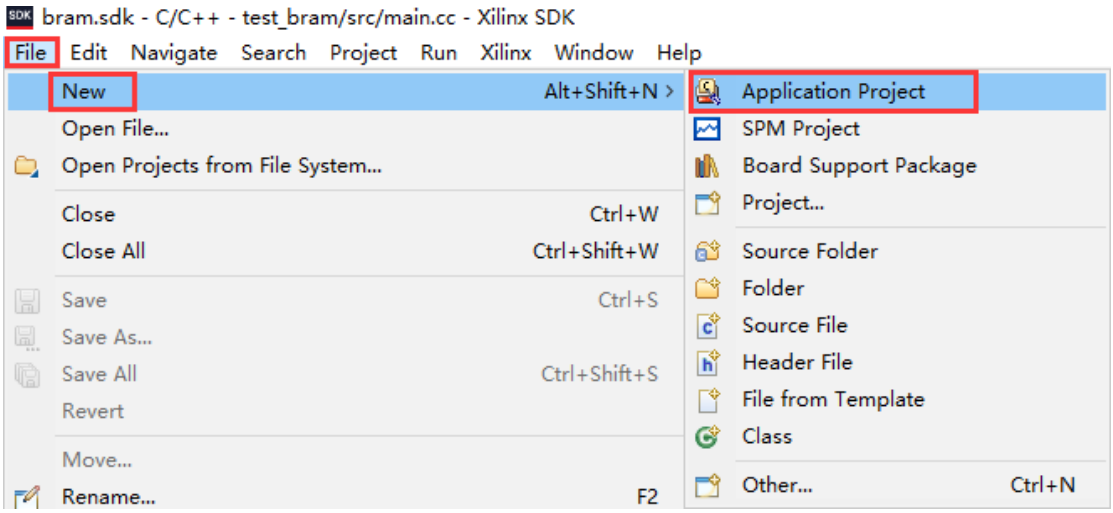


图 4- 52

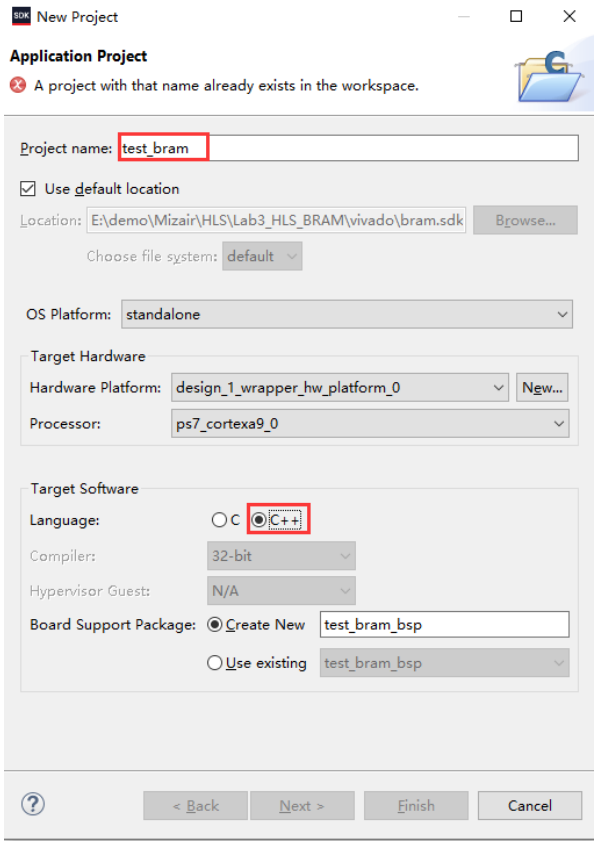


图 4- 53

按照如所示连接好开发板：

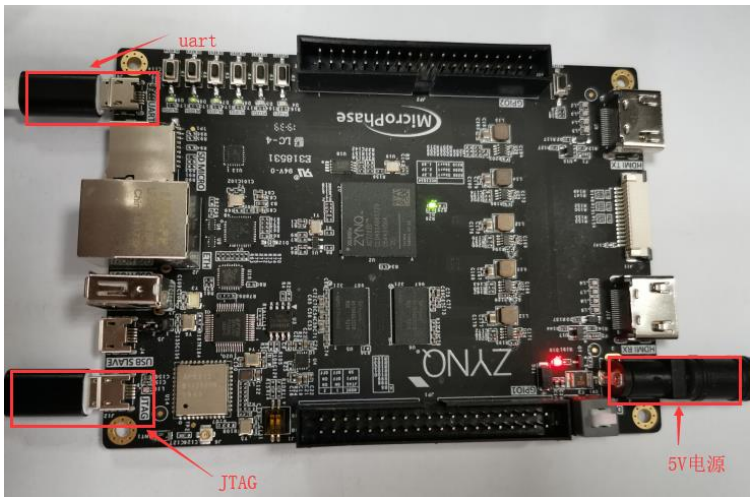


图 4- 54

创建完工程后添加如下代码：

```
/*
 * Empty C++ Application
 */
#include <stdio.h>
#include "xil_printf.h"
#include "xparameters.h"
#include "math.h"
#include "xcrazyfunction.h"


#define FUNC_DEV_ID XPAR_CRAZYFUNCTION_0_DEVICE_ID


float * xVect_Hw = (float *) 0x40000000;
float * recVect_Hw = (float *) 0x42000000;

static XCrazyfunction Crazyfunction;
static XCrazyfunction_Config *Crazyfunction_Cfg;


int initCrazyfunction(){
  int status;
  Crazyfunction_Cfg = XCrazyfunction_LookupConfig(FUNC_DEV_ID);
  status = XCrazyfunction_CfgInitialize(&Crazyfunction,Crazyfunction_Cfg);
  if(status!=XST_SUCCESS){
        printf("initialize function failed\n");
        return status;
  }
  return XST_SUCCESS;
}



void crazyFunction(float x[100],float y,float result[100]){
  int idx;
  for(idx=0;idx<100;idx++){
        result[idx] = (sin(x[idx]) + sqrtf(expf(y)))/2;
  }
}
```

```
40
41 u32 float_to_u32(float value){
42   u32 result;
43   union float_byte{
44         float v;
45         u8 byte[4];
46   }data;
47   data.v = value;
48   result = (data.byte[3]<<24) + (data.byte[2]<<16)+(data.byte[1]<<8) + (data.byte[0]);
49   return result;
50
51 }
52
53
54 int main()
55 {
56
57   int idx;
58   float xVect_Sw[100];
59   float recVect_Sw[100];
60
61   int status;
62   status = initCrazyfunction();
63   if(status!=XST_SUCCESS){
64         printf("initialize function failed\n");
65         return status;
66   }
67
68   for(idx=0;idx<100;idx++){
69         xVect_Sw[idx] = idx;
70         xVect_Hw[idx] = idx;
71   }
72   printf("test in software mode\n");
73
74   crazyFunction(xVect_Sw,1.0f,recVect_Sw);
75
76
77   printf("test in hardware mode\n");
78   XCrazyfunction_Set_y(&Crazyfunction,float_to_u32(1.0f));
79
80   XCrazyfunction_Start(&Crazyfunction);
```

```
while(!XCrazyfunction_IsDone(&Crazyfunction)){

};

//输出计算结果
for(idx=0;idx<100;idx++)
{
        printf("resVect_Hw[%d]                =                %f,recVect_Sw[%d]=                %f
\n",idx,recVect_Hw[idx],idx,recVect_Sw[idx]);
}

return 0;
}
```

该代码完成的功能是通过调用 HLS IP 来计算函数运行的结果，和通过软件调用函数直接计算函数运行的结果，通过比较二者的值，可以判断 IP 是否正常工作。

选中工程进行调试，如下图所示：

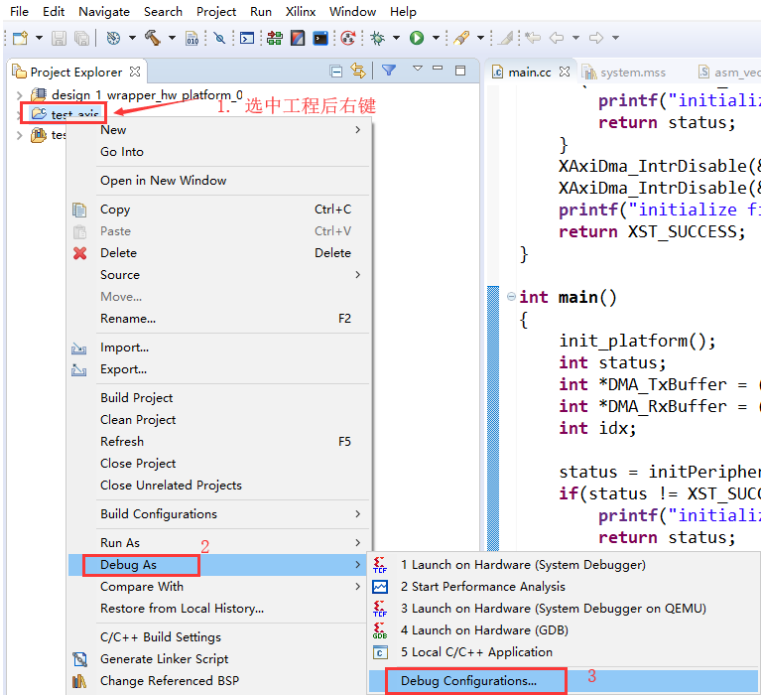


图 4- 55

在弹出界面中，按照如下图所示方式进行配置后即可运行进入 SDK 调试

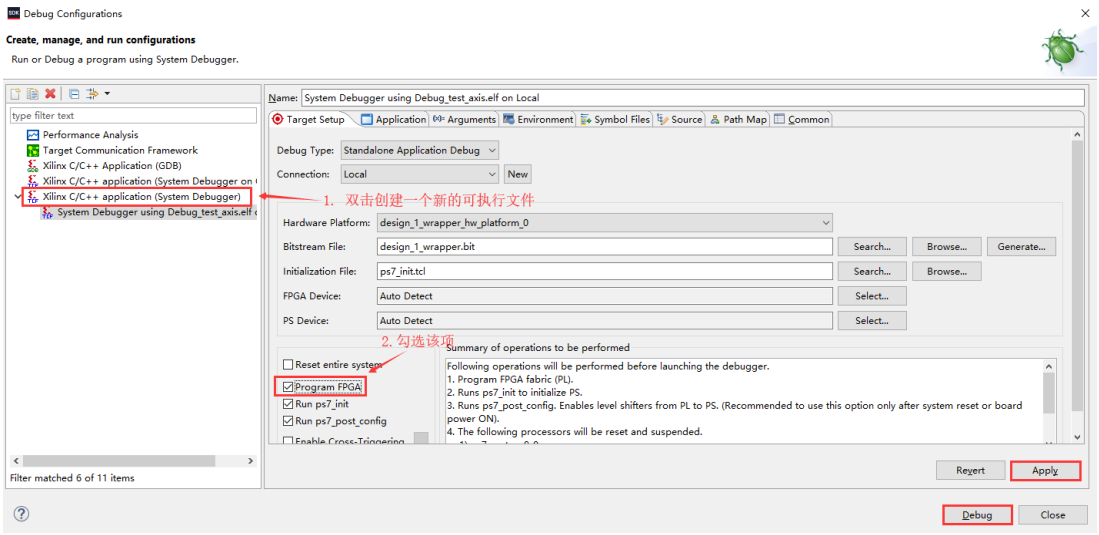


图 4- 56

进入调试界面后，首先连接串口，方便观察打印结果：

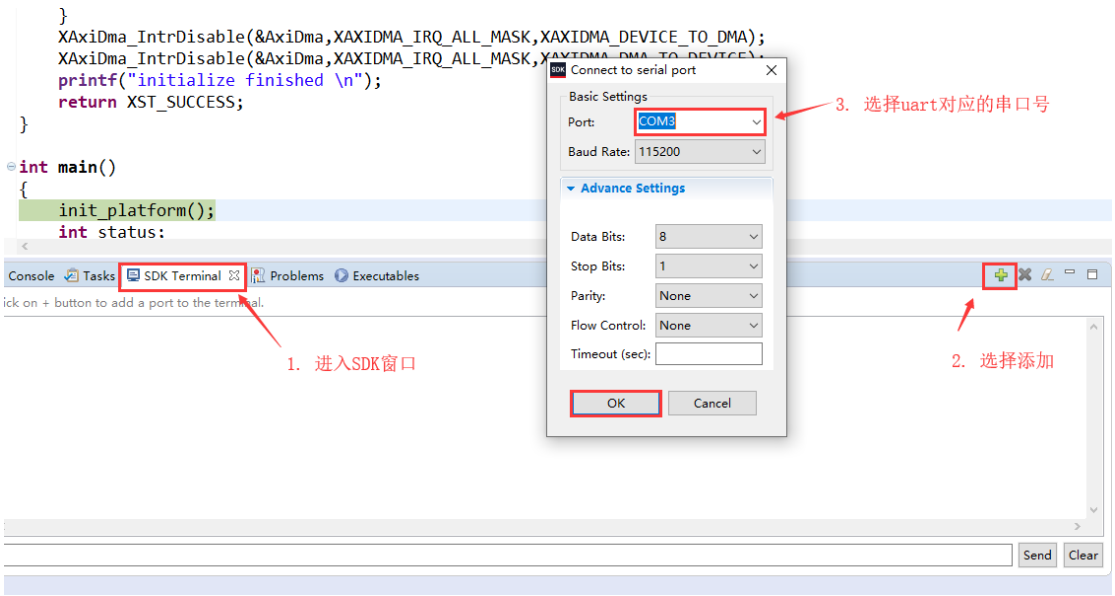


图 4- 57

点击 Debug 后，运行完整的结果如图所示：

```
test in software mode
test in hardware mode

resVect_Hw[0] = 0.824361,recVect_Sw[0]= 0.824361
resVect_Hw[1] = 1.245096,recVect_Sw[1]= 1.245096
resVect_Hw[2] = 1.279009,recVect_Sw[2]= 1.279009
resVect_Hw[3] = 0.894921,recVect_Sw[3]= 0.894921
resVect_Hw[4] = 0.445959,recVect_Sw[4]= 0.445959
resVect_Hw[5] = 0.344898,recVect_Sw[5]= 0.344898
resVect_Hw[6] = 0.684653,recVect_Sw[6]= 0.684653
resVect_Hw[7] = 1.152854,recVect_Sw[7]= 1.152854
resVect_Hw[8] = 1.319040,recVect_Sw[8]= 1.319040
resVect_Hw[9] = 1.030420,recVect_Sw[9]= 1.030420
resVect_Hw[10] = 0.552350,recVect_Sw[10]= 0.552350
resVect_Hw[11] = 0.324366,recVect_Sw[11]= 0.324365
resVect_Hw[12] = 0.556074,recVect_Sw[12]= 0.556074

resVect_Hw[13] = 1.034444,recVect_Sw[13]= 1.034444
resVect_Hw[14] = 1.319664,recVect_Sw[14]= 1.319664
resVect_Hw[15] = 1.149505,recVect_Sw[15]= 1.149505
resVect_Hw[16] = 0.680409,recVect_Sw[16]= 0.680409
resVect_Hw[17] = 0.343662,recVect_Sw[17]= 0.343662
resVect_Hw[18] = 0.448867,recVect_Sw[18]= 0.448867
resVect_Hw[19] = 0.899299,recVect_Sw[19]= 0.899299
resVect_Hw[20] = 1.280833,recVect_Sw[20]= 1.280833
resVect_Hw[21] = 1.242688,recVect_Sw[21]= 1.242688
resVect_Hw[22] = 0.819935,recVect_Sw[22]= 0.819935
resVect_Hw[23] = 0.401250,recVect_Sw[23]= 0.401250
resVect_Hw[24] = 0.371571,recVect_Sw[24]= 0.371571
```

图 4- 58

可以看出，通过直接调用软件中的函数，与通过调用 IP 的结果是一致的，由此可以说明，IP 当前能够正常工作。

4.4 实验总结

在本次实验中，通过在 HLS 中调用 hls 所支持的数学函数库，来完成浮点数的计算，并且生成了 BRAM 类型的接口，通过在 Vivado 中搭建起来的硬件环境，完成 PS 端与 PL 端的数据交互。并且在 SDK 环境中，通过简单的验证代码，对 HLS 生成的 IP 进行了验证。

本次实验比较简单，主要是为了展示，如何使用这一套完整的软件链，可以看出来 Xilinx 的工具之间关联性还是很强的。

第5章　HLS 中实现 AXI-Stream 接口

在上一章中，介绍了常用的接口 BRAM，BRAM 能够很好的用于数据连续且数据量不大的情况下，在本次实验中，介绍使用 HLS 来生成 AXI-stream 接口的 HLS IP，本次实验依旧比较简单，通过一个简单的实验，来完成 AXI-stream 的接口，这种接口在将来还会经常用到。

5.1 HLS 中实现 AXI-stream 的简单接口

5.1.1 添加源文件

创建好 HLS 工程后，新建源文件。首先新建一个头文件

```
#ifndef __DO_GANI_H__
#define __DO_GAIN_H__
#include "hls_stream.h"
#include "ap_axi_sdata.h"

//定义一个 AXI stream 的类型的数据，伴随着有 sideChannel
typedef ap_axis<32,1,1,1> intSideChannel;
typedef hls::stream<intSideChannel> AXI_STREAM;

//声明函数
void doGain(AXI_STREAM & inStream,AXI_STREAM & outStream, int gain);
#endif
```

头文件中引入了 hls_stream.h 和 ap_axi_sdata.h 这两个头文件，其中 hls_stream 包含有 AXI_Stream 流的相关变量，引入该头文件后，可以用来声明 AXI_Stream 流格式的数据。Ap_axi_sdata.h 这个头文件可以用来定义 AXI_Stream 流中的数据格式，使其满足 AXI_stream 流的协议，如 keep，id，dst 等信息，包含在 axi_stream 中。

5.1.2 添加定成函数

本次实验的顶层函数比较简单，仅仅只是将输入流的数据乘以 5 给到输出数

据流的数据，其他信息保持不变。

源文件如下：

```
1  #include "doGain.h"
2
3  //AXI stream 的输入输出流
4  void doGain(AXI_STREAM & inStream,AXI_STREAM & outStream, int gain){
5  #pragma HLS INTERFACE axis register both port=outStream
6  #pragma HLS INTERFACE axis register both port=inStream
7  #pragma HLS INTERFACE s_axilite port=gain bundle=CRTL_BUS
8  #pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
9    int idx;
10   for(idx=0;idx<1000;idx++){
11 #pragma HLS PIPELINE
12         intSideChannel ValueIn ;
13         intSideChannel ValueOut ;
14         ValueIn = inStream.read();
15         ValueOut.data = ValueIn.data * gain;
16         ValueOut.keep = ValueIn.keep;
17         ValueOut.id = ValueIn.id;
18         ValueOut.last = ValueIn.last;
19         ValueOut.user = ValueIn.user;
20         ValueOut.dest = ValueIn.dest;
21         ValueOut.strb = ValueIn.strb;
22         outStream.write(ValueOut);
23   }
24 }
```

5 到 8 行为接口的 Directive，添加这些接口后，可以将 AXI_STREAM 类型的数据的接口转换为 AXI_STREAM 接口类型。

AXI_STREAM 接口类型的 Directive 的添加方式与 BRAM 接口类型的 Directive 添加方式类似，如图**错误!未找到引用源。**所示：

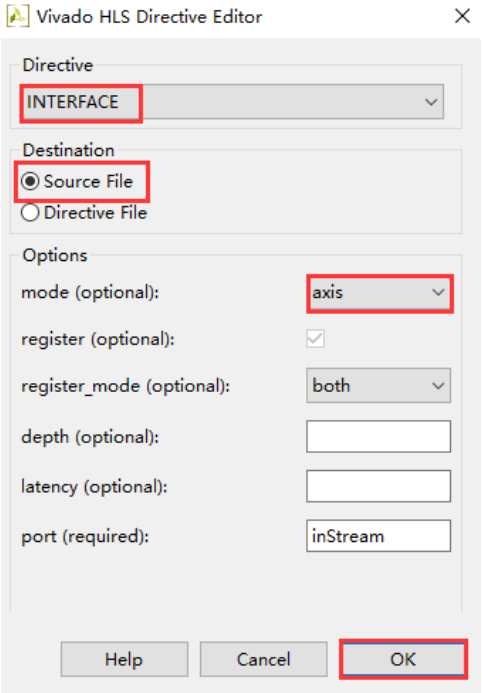


图 5- 1

在定成函数中，完成的功能仅仅只是将输入流中的数据乘以一个整数给到输出数据流。

5.1.3 添加 C 仿真文件

C 仿真文件中，完成的就是给待测试函数，提供输入流，通过观察结果，来判断顶层函数是否正常工作，仿真文件中的<< 和>> 是操作符的重载，相当于把数据推进数据流管道或者接收来自数据流管道中的数据。

```
#include "stdio.h"
#include "doGain.h"

int main(){
  //声明输入输出流
  AXI_STREAM inStream;
  AXI_STREAM outStream;
  int idx;
  //将数据填入输入流
  for(idx=0;idx<1000;idx++){
        intSideChannel valIn;
        valIn.data = idx;
```

```
        valIn.dest = 0;
        valIn.id = 0;
        valIn.keep = 1;
        valIn.last = 1;
        valIn.strb = 1;
        inStream << valIn;
    }
    //调用顶层函数
    doGain(inStream, outStream,5);

    //将输出流的结果打印出来
    for(idx=0;idx<1000;idx++){
        intSideChannel valOut;
        outStream.read(valOut);
        printf("value out is %d\n",(int)valOut.data);
    }

    return 0;
}
```

添加完 C 仿真文件后，进行 C 仿真，C 仿真的结果如图所示：

```
INFO: [SIM 211-2] *************** CSIM start ***************
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
make: 'csim.exe' is up to date.
value out is 0
value out is 5
value out is 10
value out is 15
value out is 20
value out is 25
value out is 30
value out is 35
value out is 40
value out is 45
value out is 50
value out is 55
value out is 60
value out is 65
value out is 70
value out is 75
value out is 80
value out is 85
value out is 90
```

图 5- 2

可以看到，输出数据流的结果是输入数据流的 5 倍。

5.1.4 进行 C 综合

C 仿真结束后，进行 C 综合，映射出 RTL 电路，C 综合完毕后，查看接口列表，可以看到 AXI_Stream 的接口已经生成，如图所示

inStream_TDATA	in	32	axis	inStream_V_data_V	pointer
inStream_TVALID	in	1	axis	inStream_V_dest_V	pointer
inStream_TREADY	out	1	axis	inStream_V_dest_V	pointer
inStream_TDEST	in	1	axis	inStream_V_dest_V	pointer
inStream_TKEEP	in	4	axis	inStream_V_keep_V	pointer
inStream_TSTRB	in	4	axis	inStream_V_strb_V	pointer
inStream_TUSER	in	1	axis	inStream_V_user_V	pointer
inStream_TLAST	in	1	axis	inStream_V_last_V	pointer
inStream_TID	in	1	axis	inStream_V_id_V	pointer
outStream_TDATA	out	32	axis	outStream_V_data_V	pointer
outStream_TREADY	in	1	axis	outStream_V_data_V	pointer
outStream_TVALID	out	1	axis	outStream_V_dest_V	pointer
outStream_TDEST	out	1	axis	outStream_V_dest_V	pointer
outStream_TKEEP	out	4	axis	outStream_V_keep_V	pointer
outStream_TSTRB	out	4	axis	outStream_V_strb_V	pointer
outStream_TUSER	out	1	axis	outStream_V_user_V	pointer
outStream_TLAST	out	1	axis	outStream_V_last_V	pointer
outStream_TID	out	1	axis	outStream_V_id_V	pointer

图 5- 3

紧接着进行联合仿真和导出 IP，这里不再赘述。

5.2 搭建硬件测试平台

在 Vivado IP Repository 中添加由 HLS 生成的 IP，找到 HLS 工程目录下的 solution 即可。

然后创建 Block Design，在 Block Design 中首先添加 ZYNQ 的 IP，并且配置 ZYNQ IP 核，如图所示：

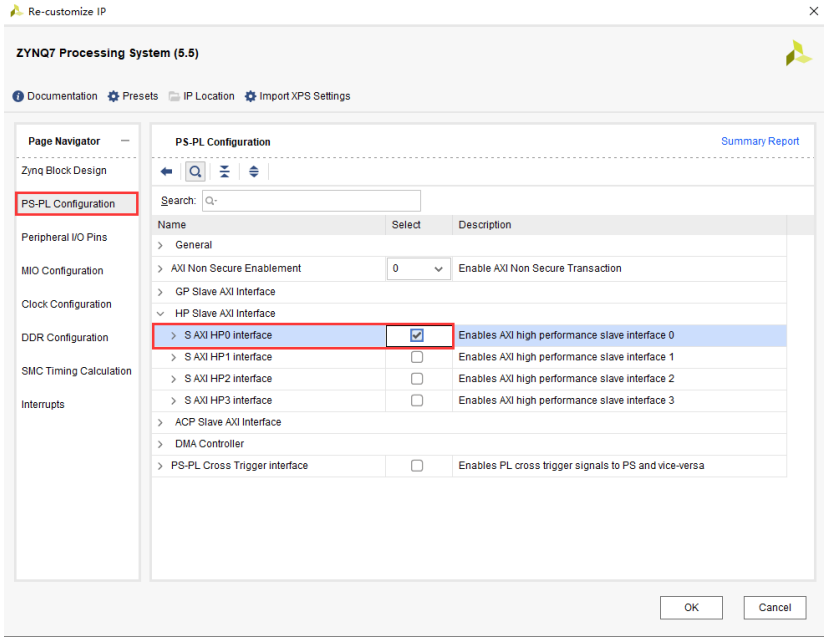

Re-customize IP
ZYNQ7 Processing System (5.5)
Documentation
Presets
IP Location
Import XPS Settings
Page Navigator
Zynq Block Design
PS-PL Configuration
Peripheral I/O Pins
MIO Configuration
Clock Configuration
DDR Configuration
SMC Timing Calculation
Interrupts
PS-PL Configuration
Summary Report
Search:
Name
Select
Description
General
AXI Non Secure Enablement
Enable AXI Non Secure Transaction
GP Slave AXI Interface
HP Slave AXI Interface
S AXI HP0 interface
Enables AXI high performance slave interface 0
S AXI HP1 interface
Enables AXI high performance slave interface 1
S AXI HP2 interface
Enables AXI high performance slave interface 2
S AXI HP3 interface
Enables AXI high performance slave interface 3
ACP Slave AXI Interface
DMA Controller
PS-PL Cross Trigger interface
Enables PL cross trigger signals to PS and vice-versa
OK
Cancel

图 5- 4

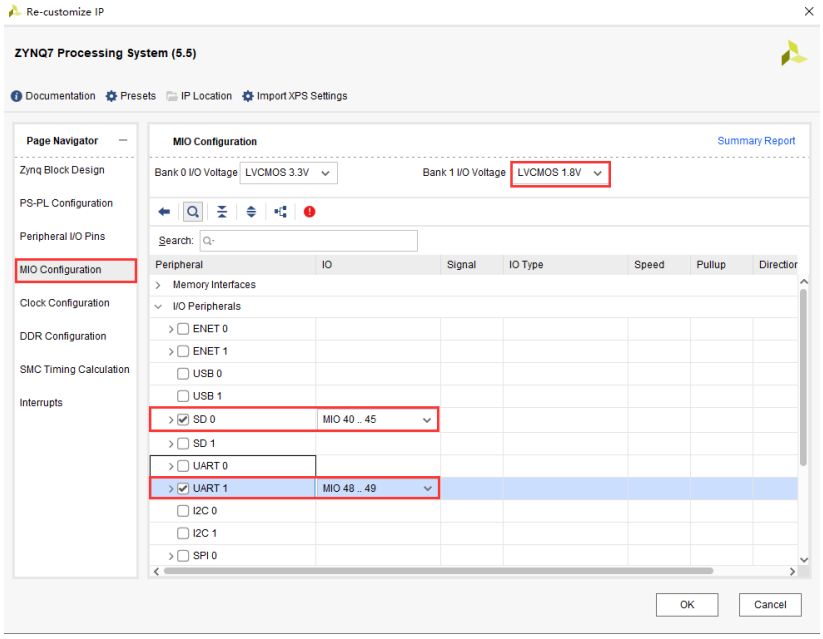

Re-customize IP
ZYNQ7 Processing System (5.5)
Documentation
Presets
IP Location
Import XPS Settings
Page Navigator
Zynq Block Design
PS-PL Configuration
Peripheral I/O Pins
MIO Configuration
Clock Configuration
DDR Configuration
SMC Timing Calculation
Interrupts
MIO Configuration
Summary Report
Bank 0 I/O Voltage
LVCMOS 3.3V
Bank 1 I/O Voltage
LVCMOS 1.8V
Search:
Peripheral
IO
Signal
IO Type
Speed
Pullup
Directior
Memory Interfaces
I/O Peripherals
ENET 0
ENET 1
USB 0
USB 1
SD 0
MIO 40 .. 45
SD 1
UART 0
UART 1
MIO 48 .. 49
I2C 0
I2C 1
SPI 0
OK
Cancel

图 5- 5

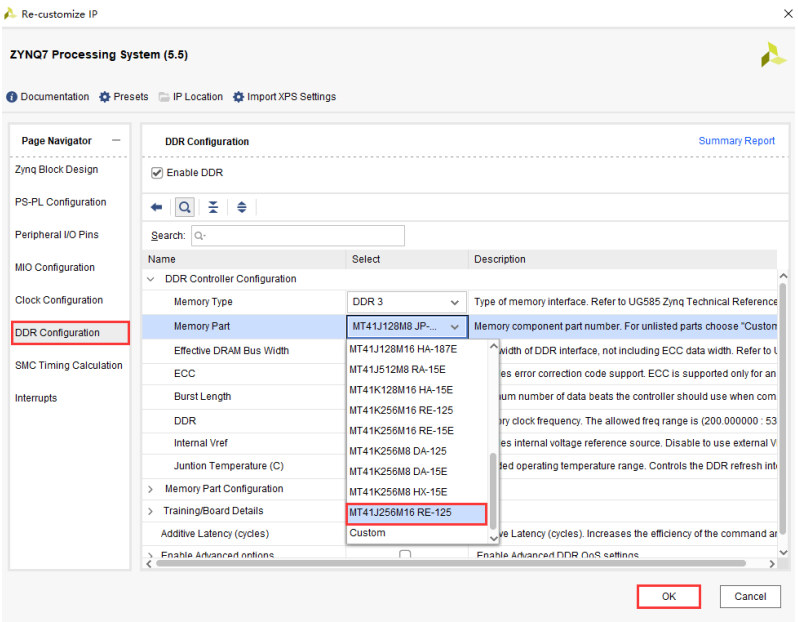


图 5- 6

接下来添加 AXI DMA IP，并对其进行配置

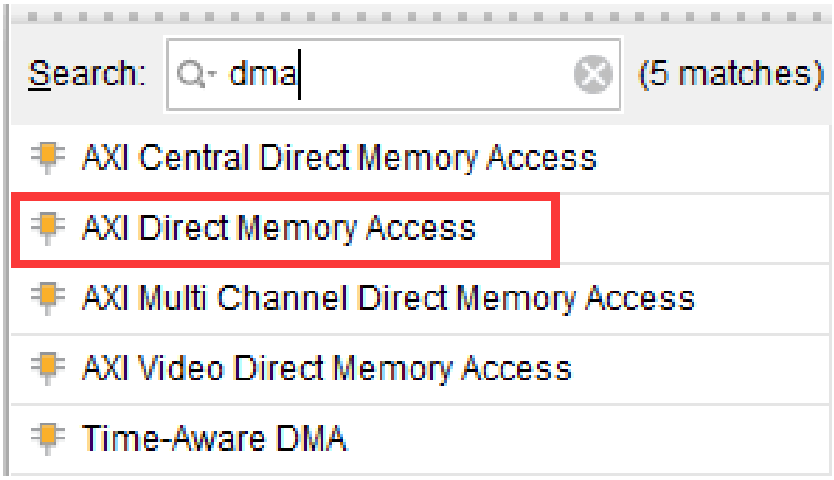


图 5- 7 添加 DMA IP

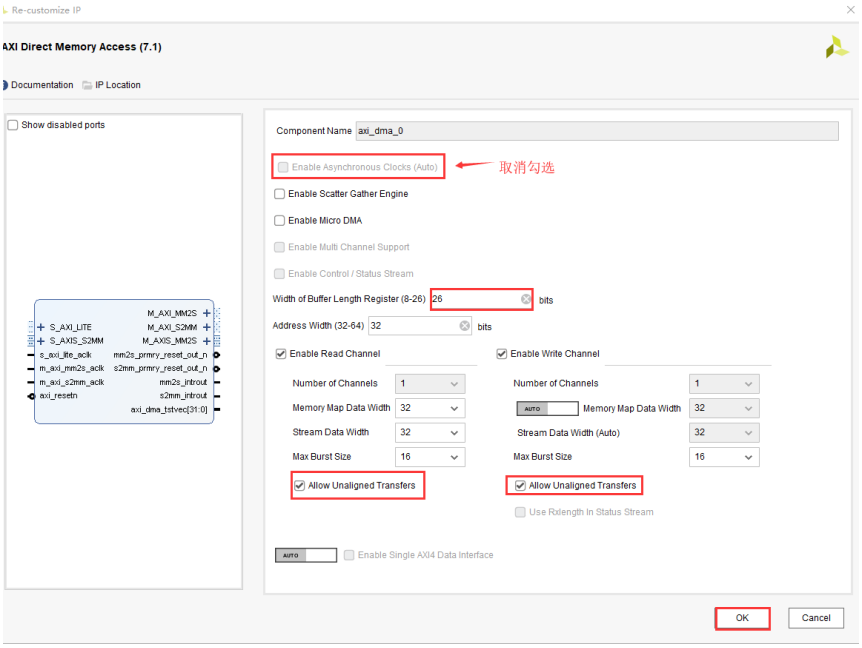


图 5- 8 配置 DMA

接下来添加 HLS 生成的带 AXI_Stream 接口的 IP，添加 IP 后将会 HLS IP 的 AXI_Stream 接口与 DMA 上的 AXI_Stream 接口相连接并连接好时钟线，如图所示：

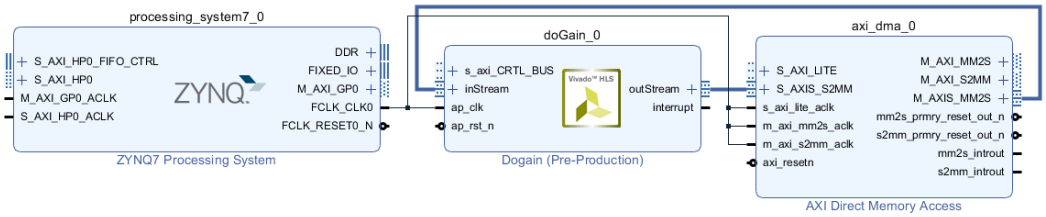


图 5- 9

然后点击自动连线完成 Block Design 的搭建，最终的 Block Design 如图所示：

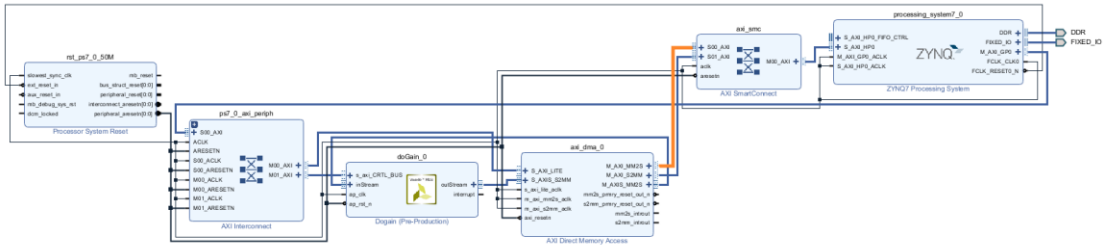


图 5- 10

然后创建 wrapper 生成 bit 流文件，导出 bit 流后，Launch SDK，在 SDK 中进行开发。

5.3 创建 SDK 工程

导出 bitstream 后，Launch SDK 创建一个新的工程。在工程的 main.c 文件中，完成的代码如下：

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xaxidma.h"
#include "xparameters.h"
#include "xdogain.h"


#define GAIN_DEV_ID XPAR_DOGAIN_0_DEVICE_ID
#define DMA_DEV_ID XPAR_AXI_DMA_0_DEVICE_ID

 #define MEM_BASE_ADDR 0x01000000
 #define TX_BUFFER_ADDR (MEM_BASE_ADDR + 0x00100000)
 #define RX_BUFFER_ADDR (MEM_BASE_ADDR + 0x00300000)

 #define ARR_SIZE 1000

```

```
18
19
20  int inStream[ARR_SIZE];
21
22  static XDogain DoGian;
23  static XDogain_Config * Dogain_Cfg;
24  static XAxiDma AxiDma;
25  static XAxiDma_Config * AxiDma_Cfg;
26
27  int initPeripherals(){
28      int status;
29      printf("initialize doGain core\n");
30      Dogain_Cfg = XDogain_LookupConfig(GAIN_DEV_ID);
31      status = XDogain_CfgInitialize(&DoGian,Dogain_Cfg);
32      if(status != XST_SUCCESS){
33          printf("initialize failed\n");
34          return status;
35      }
36      printf("initialize AXIDMA core\n");
37      AxiDma_Cfg = XAxiDma_LookupConfig(DMA_DEV_ID);
38      status = XAxiDma_CfgInitialize(&AxiDma,AxiDma_Cfg);
39      if(status != XST_SUCCESS){
40          printf("initialize failed\n");
41          return status;
42      }
43
    XAxiDma_IntrDisable(&AxiDma,XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
44
    XAxiDma_IntrDisable(&AxiDma,XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);
45      printf("initialize finished \n");
46      return XST_SUCCESS;
47  }
48
49  int main()
50  {
51      init_platform();
52      int status;
53      int *DMA_TxBuffer = (int *) TX_BUFFER_ADDR;
54      int *DMA_RxBuffer = (int *) RX_BUFFER_ADDR;
55      int idx;
56
```

```
57        status = initPeripherals();
58        if(status != XST_SUCCESS){
59              printf("initialize failed\n");
60              return status;
61        }
62        //初始化发送 buffer
63        for(idx=0;idx<ARR_SIZE;idx++){
64              inStream[idx] = idx;
65        }
66
67        while(1){
68              int gain;
69              printf("Enter a gain:");
70              scanf("%d",&gain);
71              //设置 gain 的值
72              XDogain_Set_gain(&DoGian,gain);
73              //开始测试 IP 核
74              XDogain_Start(&DoGian);
75              //刷新 cache 以便能够观察到 cache 中的数据
76              Xil_DCacheFlushRange((u32)&inStream,(ARR_SIZE * sizeof(int)));
77              Xil_DCacheFlushRange((u32)DMA_RxBuffer,(ARR_SIZE * sizeof(int)));
78              printf("sending data to IP\n");
79              //从 DMA 发送数据到 IP 核
80              XAxiDma_SimpleTransfer(&AxiDma,(u32)inStream,ARR_SIZE                    *
sizeof(int),XAXIDMA_DMA_TO_DEVICE);
81
82              printf("getting data from IP \n");
83              //从 IP 核得到数据
84              XAxiDma_SimpleTransfer(&AxiDma,(u32)DMA_RxBuffer,ARR_SIZE                *
sizeof(int),XAXIDMA_DEVICE_TO_DMA);
85              while(XAxiDma_Busy(&AxiDma,XAXIDMA_DEVICE_TO_DMA)){
86
87              };
88
89              Xil_DCacheInvalidateRange((u32)DMA_RxBuffer,ARR_SIZE * sizeof(int));
90              while(!XDogain_IsDone(&DoGian)){
91
92              };
93              printf("calculate is done \n");
94              for(idx=0;idx<ARR_SIZE;idx++){
95                    printf("Rx_Buffer[%d]=%d\n",idx,DMA_RxBuffer[idx]);
```

```
            }
        }

        return 0;
}
```

该部分的代码完成的功能是，用户通过按键输入一个数字来为 IP 的 gain 输入赋值，数据通过 DMA 传递给 IP 核，IP 核将传入的数据流与 gain 相乘后将输出的数据流通多 DMA 传递到内存中。以此完成 AXI_Stream 形式的数据的流动。

连接好开发板后，点击 debug 开始运行；在 SDK Terminal 中会提示输入一个值，通过在键盘上输入，然后通过串口发送到 FPGA 中。

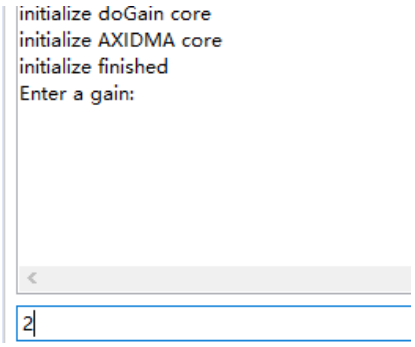


图 5- 11

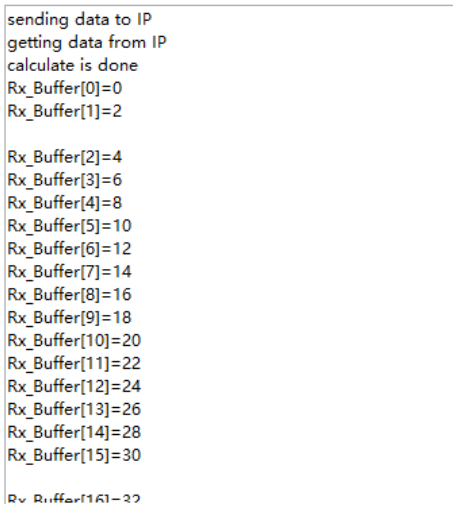


图 5- 12

可以观察到输出数据通过串口打印在控制台上。

5.4 实验总结

在本章实验中，通过简单的示例，完成了 HLS 生成一个带 AXI_Stream 接口的 IP 核的生成，该 IP 完成的任务比较简单。在 vivado 集成环境中，通过通过 DMA 将 HLS 生成的 IP 核通过 DMA 与 DDR 相连，使 ZYNQ 能够通过内存访问到 HLS 生成的 IP。

支持 AXI_Stream 协议的数据，在在使用 HLS 进行图像处理相关的开发时会经常使用到。在之后的实验中，还会大量使用到 AXI_Stream 这种类型的数据。

第6章　图像直方图

6.1 实验简介

对于一幅灰度图像，可以表示为 I(x,y)。I 是图像的强度大小。灰度直方图就是统计不同灰度的分布概率。比如对于一个像素 bit 位为 8 的图像，那么其灰度范围就是从 0~255.灰度直方图就是统计每个灰度级拥有的像素比例。通过直方图可以直观的反映出图像的明暗程度，比如 0 如果概率大，就说明图像偏暗，255 概率大，就说明图像偏亮。

下面通过一幅灰度图像来简单说明一下灰度直方图的含义：

在 matlab 中输入以下代码，可以将一幅彩色的 RGB 图片，转换为灰度图像，并将灰度图像的直方图展示出来。

```
img_rgb = imread('Lena.jpg');
imtool(img_rgb);
img_gray = rgb2gray(img_rgb);
imtool(img_gray);
imhist(img_gray);
```

图 6- 1 原始图像

图 6- 2 灰度图像

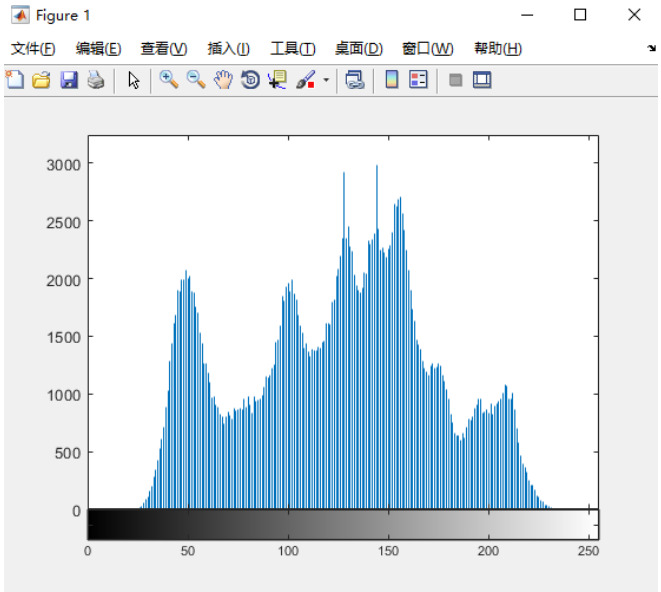


图 6- 3 灰度图像直方图

从上面的灰度图像的直方图可以看出，灰度图像直方图，就是对图像中每个像素点的灰度值进行统计。按照一定的灰度区间划分不同的 bin，如果图像的灰度值落在该 bin 的区间内，那么 bin 区间的统计值就加一。在上图中，bin 的区间的范围是 1，即对每个不同的灰度值都进行统计，从灰度图像可以看出，图像的灰度值分布适中，也可以说该幅图像的明暗程度是适中的。

6.2 图像直方图的 HLS 实现

在本次实验中，在 HLS 中实现对一幅图像的直方图的统计。创建好工程后，首先添加源文件和 C 仿真文件。

6.2.1 定义头文件

头文件的定义如下：

```
#ifndef __HISTO_HEAD_H__
#define __HISTO_HEAD_H__
//引入 stream 类
#include <hls_stream.h>
//使用 axi stream side channel(TLAST,TKEEP,TUSER,TID and so on)
#include <ap_axi_sdata.h>

//定义输入输出文件
#define INPUT_IMAGE "Lena.bmp"
#define HISTOGRAM_FILE "histogram.txt"
//8 位无符号 side_channel 数据类型
typedef ap_axiu<8,1,1,1> uint_8_side_channel;
typedef hls::stream<uint_8_side_channel> AXI_STREAM;
void Histogram(AXI_STREAM &inStream,int HistOut[256]);

#endif
```

头文件中，重定义了 AXI_Stream 流数据中，每个数据的格式；重定义了 AXI_Stream 的数据流的格式，声明了顶层函数，定义了输入输出文件。

6.2.2 添加源文件

在源文件中定义顶层函数，顶层函数的定义如下：

```
#include "histogram.h"

void Histogram(AXI_STREAM &inStream, int HistoOut[256]){
#pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
#pragma HLS INTERFACE axis register port=inStream
#pragma HLS INTERFACE bram port=HistOut

```

```
8     int idx;
9     uint_8_side_channel recPixel;
10    //每一帧图像来之前，清空 bram 中的内容
11    for(idx=0;idx<256;idx++){
12 #pragma HLS PIPELINE
13          HistoOut[idx]=0;
14    }
15    //接收图像的数据，并且计算直方图
16    for(idx=0;idx<(512*512);idx++){
17          recPixel = inStream.read();
18          HistoOut[recPixel.data] +=1;
19    }
20 }
```

第 4~6 行是添加接口的 Directives，其中将输入的 AXI_Stream 格式的数据定义为 AXI_Stream 类型的接口，将数组类型的数据定义为 BRAM 类型的接口，并对函数的返回值定义为 AXI_Lite 接口，方便再后续的 SDK 中对该 IP 进行调用。

11 到 14 行完成将数组中的数据清零，这样每次一幅图像到来的时候都能重新开始统计。

16 到 19 行通过读取输入数据流中的数据，将该图像的灰度值统计到直方图中。

6.2.3 添加仿真文件

在仿真文件中，添加如下内容：

```
1   #include "stdio.h"
2   #include "histogram.h"
3   #include <hls_opencv.h>
4   #include <opencv2/core/core.hpp>
5
6   int hsitogram[256];
7
8   void saveHistgram(const char * file, int *histgram){
9     FILE *pFile;
10    pFile = fopen(file,"w");
11    int idx;
12    if(pFile!=NULL){
```

```
13          for(idx=0;idx<256;idx++){
14              fprintf(pFile,"Bin[%d]=%d\n",idx,histgram[idx]);
15          }
16      }
17  }
18
19  int main(){
20      printf("Load Image %s \n",INPUT_IMAGE);
21      cv::Mat imageSrc;
22      imageSrc = cv::imread(INPUT_IMAGE);
23      //将图像转换为灰度图像
24      cv::cvtColor(imageSrc, imageSrc,CV_BGR2GRAY);
25      printf("image row=%d,image col= %d\n",imageSrc.rows,imageSrc.cols);
26      //定义输入输出的图像数据流
27      AXI_STREAM inStream;
28      for(int idxRows=0;idxRows<imageSrc.rows;idxRows++){
29          for(int idxCols=0;idxCols<imageSrc.cols;idxCols++){
30              uint_8_side_channel valIn;
31              valIn.data = imageSrc.at<unsigned char>(idxRows, idxCols);
32              valIn.dest = 1;
33              valIn.keep = 1;
34              valIn.last = 1;
35              valIn.strb = 1;
36              valIn.id = 0;
37              valIn.user = 1;
38              inStream << valIn;
39          }
40      }
41      Histogram(inStream, hsitogram);
42      saveHistgram(HISTOGRAM_FILE,hsitogram);
43      return 0;
44  }
```

在 main 函数中，通过一种与第二次实验中不同的方法将外部文件的图片导入到工程中，并将其转换为 HLS 支持的数据格式。

第 38 行的<< 是对运算符的重载，相当于把数据推入到管道中。最后将得到的灰度图像的指仿图输出到外部的 txt 文件中。

6.2.4 进行仿真与综合

添加好源文件后，以此进行 C 仿真，C 综合，C/RTL 联合仿真。

在 C 仿真结束后，可以看到，Console 没有报错，如图所示：

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../code/test_histogram.cpp in debug mode
   Compiling ../../../code/histogram.cpp in debug mode
   Generating csim.exe
Load Image Lena.bmp
image row=512,image col= 512
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************

```

图 6- 4

此时在 Solution 的 Csim 文件夹下，会出现一个输出的 txt 文本数据，里面记录了对本幅图像的直方图统计。

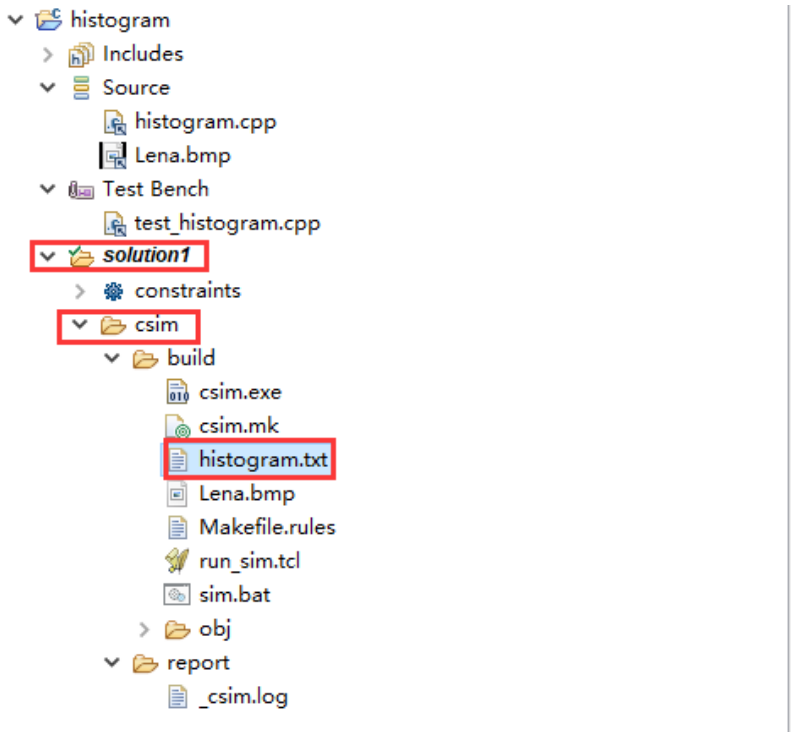


图 6- 5

打开查看即可看到，图像直方图的统计，如图所示：

```
Bin[20]=1
Bin[21]=0
Bin[22]=0
Bin[23]=2
Bin[24]=4
Bin[25]=8
Bin[26]=15
Bin[27]=24
Bin[28]=55
Bin[29]=85
Bin[30]=112
Bin[31]=159
Bin[32]=202
```

图 6- 6

C 综合完毕后，可以查看映射为 RTL 电路的 IP 的接口和资源占用情况。资源占用情况如图所示，可以看到直方图的统计对资源的占用是很小的。

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	77
FIFO	-	-	-	-
Instance	0	-	36	40
Memory	-	-	-	-
Multiplexer	-	-	-	31
Register	-	-	82	-
Total	0	0	118	148
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

图 6- 7 资源占用情况

综合后的接口如图所示：可以看到综合完成之后，分别生成了 AXI_Lite, AXI_Stream 和 BRAM 接口。

s_axi_CRTL_BUS_RDATA	out	32	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_RRESP	out	2	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_BVALID	out	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_BREADY	in	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_BRESP	out	2	s_axi	CRTL_BUS	return void
ap_clk	in	1	ap_ctrl_hs	Histogram	return value
ap_rst_n	in	1	ap_ctrl_hs	Histogram	return value
interrupt	out	1	ap_ctrl_hs	Histogram	return value
inStream_TDATA	in	8	axis	inStream_V_data_V	pointer
inStream_TVALID	in	1	axis	inStream_V_dest_V	pointer
inStream_TREADY	out	1	axis	inStream_V_dest_V	pointer
inStream_TDEST	in	1	axis	inStream_V_dest_V	pointer
inStream_TKEEP	in	1	axis	inStream_V_keep_V	pointer
inStream_TSTRB	in	1	axis	inStream_V_strb_V	pointer
inStream_TUSER	in	1	axis	inStream_V_user_V	pointer
inStream_TLAST	in	1	axis	inStream_V_last_V	pointer
inStream_TID	in	1	axis	inStream_V_id_V	pointer
HistoOut_Addr_A	out	32	bram	HistoOut	array
HistoOut_EN_A	out	1	bram	HistoOut	array
HistoOut_WEN_A	out	4	bram	HistoOut	array
HistoOut_Din_A	out	32	bram	HistoOut	array
HistoOut_Dout_A	in	32	bram	HistoOut	array
HistoOut_Clk_A	out	1	bram	HistoOut	array
HistoOut_Rst_A	out	1	bram	HistoOut	array

图 6- 8 综合后接口

进行完 C/RTL 联合仿真后，我们可以在 Solution 下的 Sim 文件夹下发现 RTL 级仿真后输出的结果。

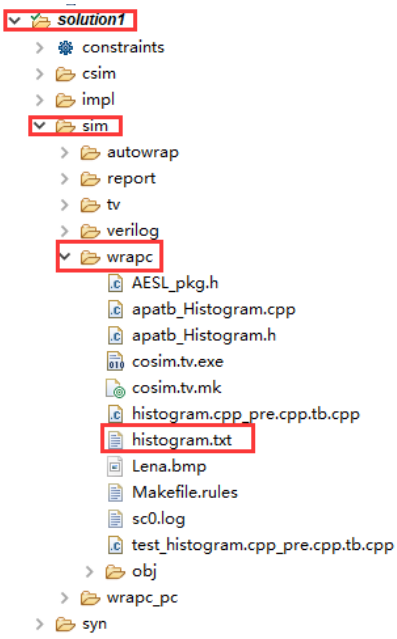


图 6- 9

打开查看该文件夹下的 histogram.txt 文件，可以得到由 RTL 仿真得到的结果。可以看出，由 RTL 和 C 仿真得到的结果一致。

接下来即可导出 IP 核，在 Vivado 集成开发环境中进行调用。

6.3 搭建测试硬件平台

导出 IP 后，创建 Vivado 工程，将 HLS 生成的 IP 添加到 Vivado 的 IP Repository 中。

在 Block Design 中首先添加 ZYNQ 处理器，配置方法与上一章实验中类似。然后添加 Histogram IP，添加后点击 Run Connection Automation，首先进行 BRAM 端口侧的配置，选中 HistoOut_PORTA 然后点击 OK，即自动生成一个 BRAM。然后按前面第三章的实验中，添加 axi_bram_controller，使得 PS 端能够访问该 BRAM。如图所示：

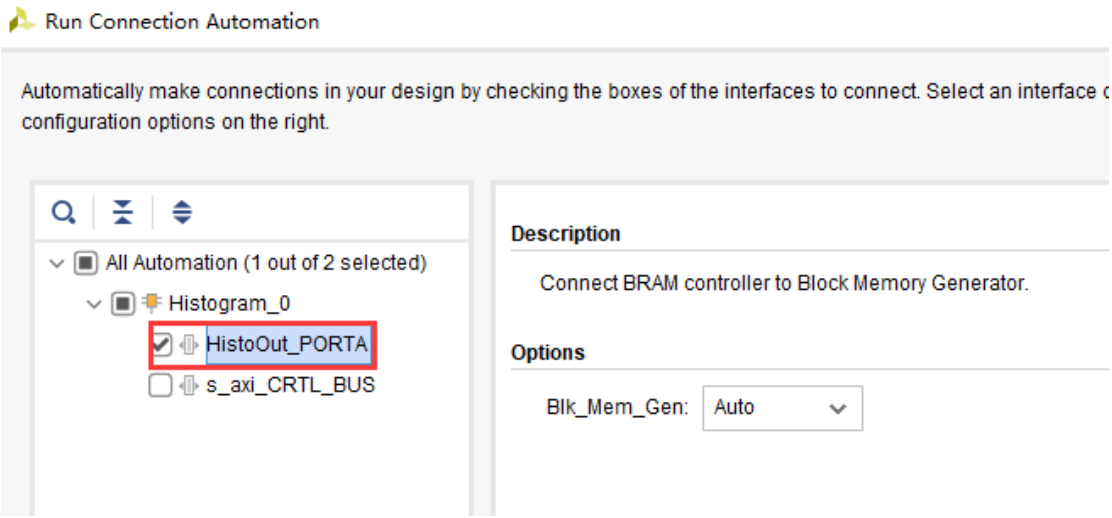


图 6- 10

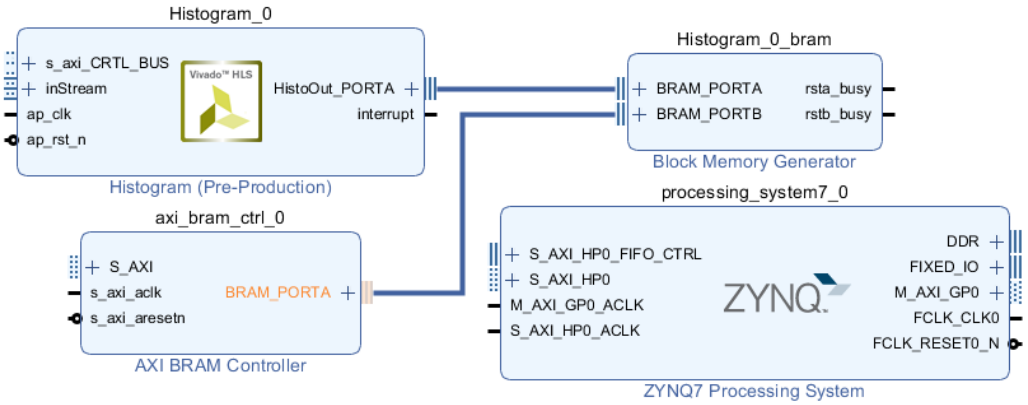


图 6- 11

接下来添加 DMA 使得 IP 能够访问 DDR 中的内容。DMA 的配置方法与第 4 章实验当中的配置方法类似，只需使用到读通道即可。按**错误!未找到引用源。**所示连接好 IP。然后点击 Run Connection Automation 完成 IP 间的互联。

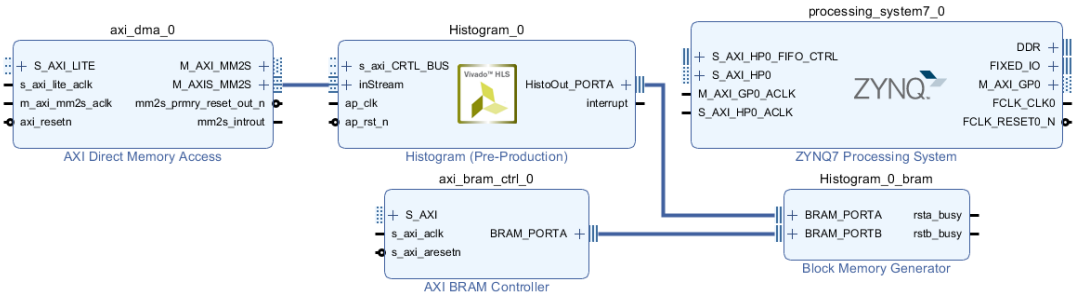


图 6- 12

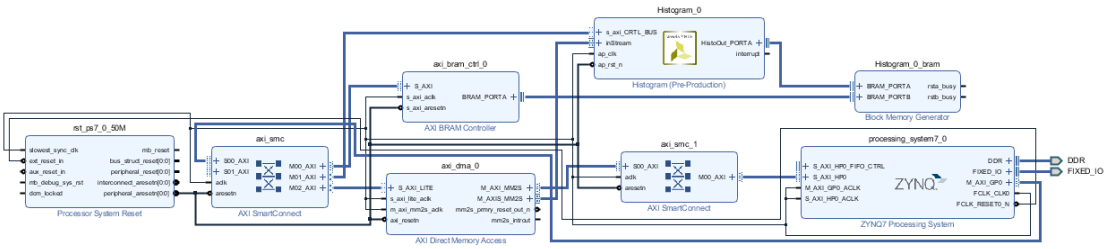


图 6- 13

全部连接完毕后，生成 wrapper 然后生成 bitstream，等到 bitstream 生成完毕后即可到 SDK 中进行开发。

6.4 创建 SDK 工程

在创建 Vivado 工程前，先使用 matlab 生成一幅灰度图像的十六进制的格式的 txt 文本文件。

打开 matlab，在工程路径下，运行以下脚本。

```
img_rgb = imread('Lena.bmp');
img_gray = rgb2gray(img_rgb);
fid0 = fopen('Lena.txt', 'wt');
fprintf(fid0, '0x%02x, ', img_gray');
fid0 = fclose(fid0);
```

运行完成后，即可生成一个名为 Lena.txt 的文本文件，将这个文本文件的内容更改为一个数组，然后将这个数组添加到 SDK 工程中。

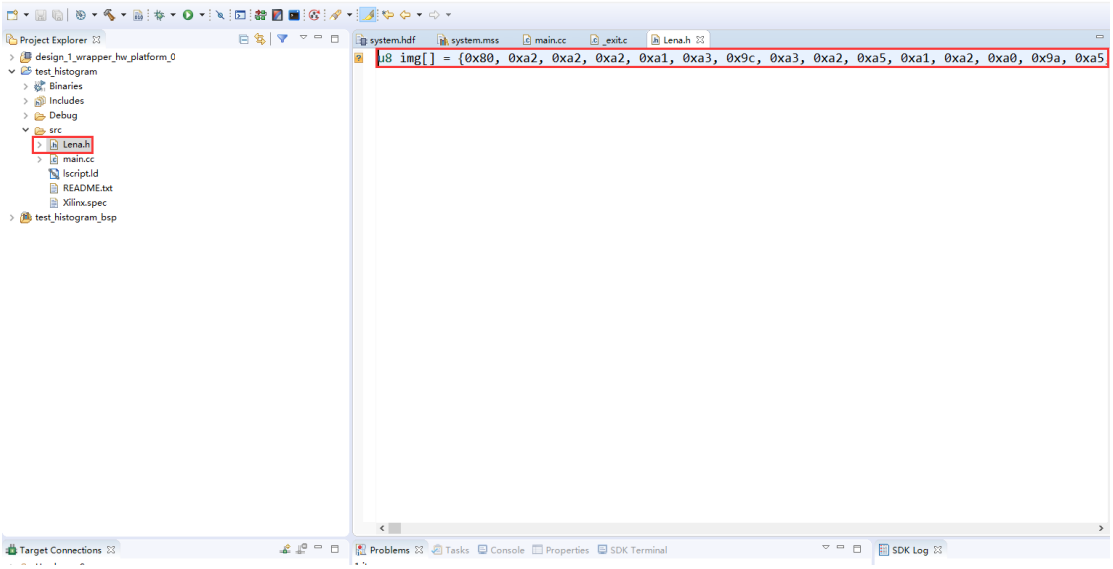


图 6- 14

在 SDK 中，进行如下设计，分别通过软件和调用 IP 核来完成对图像直方图的统计，并将找出直方图中的最大值和最小值，最后分别比较图像的直方图统计结果是否一致。

代码设计如下：

```
#include <stdio.h>
#include <xaxidma.h>
#include <xhistogram.h>
#include "Lena.h"
#include "xparameters.h"

//IP 的设备地址
#define HISTO_DEV_ID XPAR_HISTOGRAM_0_DEVICE_ID
#define AXIDMA_DEV_ID XPAR_AXIDMA_0_DEVICE_ID

```

```
//图像大小
#define ARR_SIZE 512*512

//DMA 使用到的内存地址
#define MEM_BASE_ADDR    0x01000000
#define memRxBuffer_Base (MEM_BASE_ADDR + 0x00300000)

//BRAM  的地址
#define BRAM_BASE_ADDR 0x40000000

//定义指向输入输出 buffer 的指针

;unsigned char * RxBuffer = (unsigned char *) memRxBuffer_Base;

//通过软件计算出的直方图的数组
unsigned int hist_Sw[256];

unsigned int    * hist_Hw = (unsigned int *)BRAM_BASE_ADDR;
//软件输出图像的矩阵
unsigned char imageOut_Sw[ARR_SIZE];
//输入如图像的像素矩阵
unsigned char imageIn_Hw[ARR_SIZE];

//定义 IP 的配置和初始化结构体
static XAxiDma AxiDma;
static XAxiDma_Config * AxiDma_Cfg;
static XHistogram Histo;
static XHistogram_Config *Histo_Cfg;

int initperipheral(){
        int status;

        //初始化 Histogram IP
        printf("initialize Histogram...\n");
        Histo_Cfg = XHistogram_LookupConfig(HISTO_DEV_ID);
        status = XHistogram_CfgInitialize(&Histo,Histo_Cfg);
        if(status !=XST_SUCCESS){
                printf("initialize failed\n");
        }


```

```
52        //初始化 AXI DMA
53
54        printf("initialize AXIDMA core...\n");
55        AxiDma_Cfg = XAxiDma_LookupConfig(AXIDMA_DEV_ID);
56        status = XAxiDma_CfgInitialize(&AxiDma,AxiDma_Cfg);
57        if(status != XST_SUCCESS){
58              printf("initialize failed\n");
59              return status;
60        }
61
    XAxiDma_IntrDisable(&AxiDma,XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
62
    XAxiDma_IntrDisable(&AxiDma,XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);
63        printf("initialize finished \n");
64        return XST_SUCCESS;
65  }
66
67
68  //通过软件的方法计算出图像直方图
69  void doHistSw(unsigned char *img,unsigned int *Array){
70        //清空直方图数组中的数据
71        for(int idx=0;idx<256;idx++){
72              Array[idx] = 0;
73        }
74
75        for(int idx=0;idx<ARR_SIZE;idx++){
76              //通过判断图像中各个像素的值来填充直方图数组的数据
77              Array[img[idx]] +=    1;
78        }
79  }
80
81  int main()
82  {
83
84        int status;
85        unsigned char min;
86        unsigned char max;
87        //初始化外设
88        status = initperipheral();
89        if(status!=XST_SUCCESS){
90              printf("initialize timer failed\n");
```

```
91              return status;
92          }
93          //使用软件方法计算图像直方图
94          printf("Doing histogram on software...\n");
95          doHistSw(img,hist_Sw);
96          //得到软件计算的灰度最大值和最小值
97          for(int idx=0;idx<256;idx++){
98              min = idx;
99              if(hist_Sw[idx])
100                 break;
101         }
102
103         //得到软件计算得到的最大值
104         for(int idx=255;idx>=0;idx--){
105             max = idx;
106             if(hist_Sw[idx])
107                 break;
108         }
109         printf("The histogram in Software is between %d~%d  !   \n",min,max);
110
111         //通过 FPGA 完成直方图的处理
112         printf("Doing Histogram in hardware...\n");
113         for(int idx=0;idx<ARR_SIZE;idx++){
114             imageIn_Hw[idx] = img[idx];
115         }
116
117         XHistogram_Start(&Histo);
118         //通过硬件计算完成 Histogram
119
120         //刷新 cache 以便能够观察到内存中数据的变化
121         Xil_DCacheFlushRange((u32)imageIn_Hw,(ARR_SIZE * sizeof(unsigned char)));
122         Xil_DCacheFlushRange((u32)RxBuffer,(ARR_SIZE*sizeof(unsigned char)));
123
124         //DMA 进行数据传递
125
126         //将图像数据传输到 IP
127         XAxiDma_SimpleTransfer(&AxiDma,(u32)imageIn_Hw,(ARR_SIZE*sizeof(unsigned
char)),XAXIDMA_DMA_TO_DEVICE);
128
129         //等待数据传输完毕
130         while(XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE)){
```

```
    };

    //得到软件计算的灰度最大值和最小值
    for(int idx=0;idx<256;idx++){
        min = idx;
        if(hist_Hw[idx])
            break;
    }

    //得到软件计算得到的最大值
    for(int idx=255;idx>=0;idx--){
        max = idx;
        if(hist_Hw[idx])
            break;
    }
    printf("The histogram in Hardware is between %d~%d \n",min,max);

    //比较通过软件和硬件计算出来的结果
    printf("compare the result in Software and Hardware...\n");
    int error=0;
    for(int idx=0;idx<256;idx++){
        if(hist_Hw[idx]!=hist_Sw[idx]){
            error=1;
        }
    }
    if(error == 1){
        printf("The result do not match !\n");
    }
    else{
        printf("The result do matches well!\n");
    }

    return 0;
}
```

点击 Debug 后进行运行工程，运行结果如图所示，从运行结果可以看出来，图像的直方图的范围为 20~245，其中通过软件方法和通过调用 IP 核运算出来的结果是一致的。

```
initialize Histogram...
initialize AXIDMA core...
initialize finished
Doing histogram on software...
The histogram in Software is between 20~245 !
Doing Histogram in hardware...
The histogram in Hardware is between 20~245

compare the result in Software and Hardware...
The result do matches well!
```

图 6- 15

6.5 实验总结

在本次实验中，了解了如何去处理一幅灰度图像的直方图，并通过 HLS 生成了一个计算直方图的 IP 核，在 Vivado 中进行调用，并通过 SDK 对 IP 核的功能进行了验证。

直方图在图像处理中经常会使用到，在做类似于图像分割的应用中，直方图有着很大的作用。通过这一实验，只能说简单的了解一下什么是直方图，具体的应用还需要用户自己去摸索。

第7章　图像直方图拉伸

7.1 实验简介

在上一章实验中，简单进行了直方图的统计，在本章中，对直方图的做一个简单的扩展使用，用来将直方图进行伸展，调节图像的对比对。

首先用 matlab 来对图像拉伸进行一个简单且直观的说明。下面有两幅图像如图所示，其中一幅图像是没有进行直方图拉伸时的原图，另外一幅图像时进行直方图拉伸之后的结果。从结果可以看出，经过拉伸后的图像比原来的图像要更加明亮，对比度也比原来要高。使用 matlab 来查看一下两者的直方图分布，直方图分布如图所示：

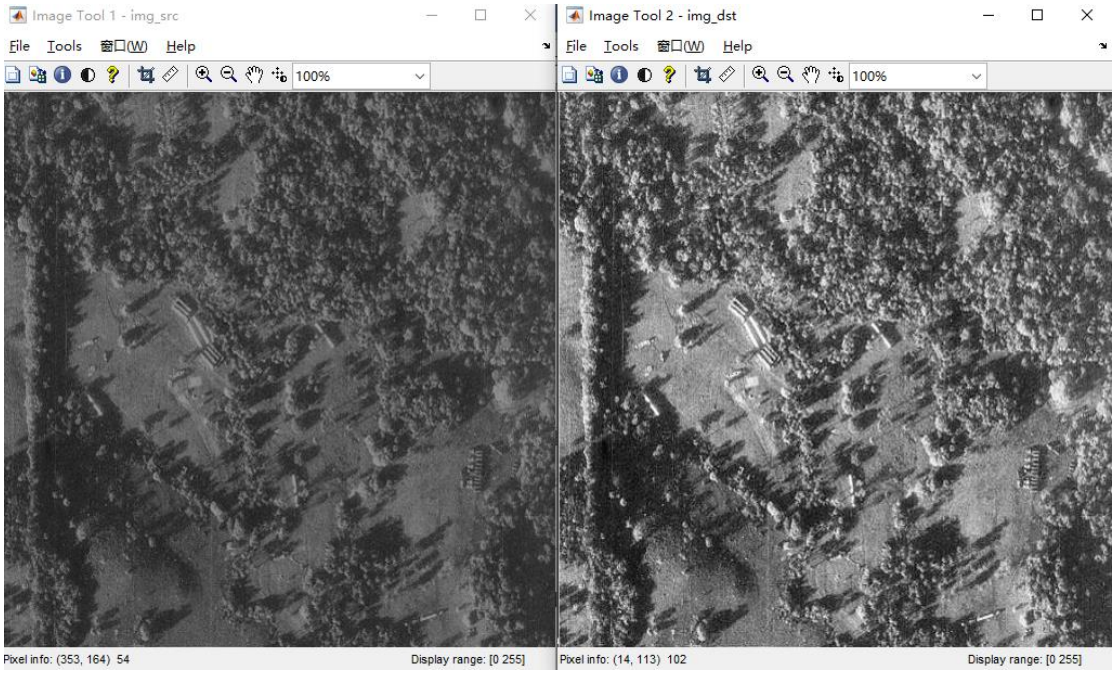


图 6- 1 图像对比

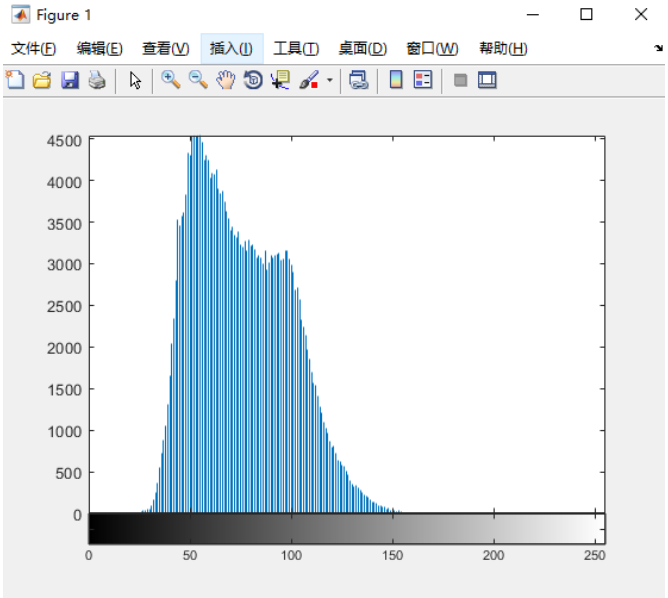


图 7- 2 原图的直方图分布

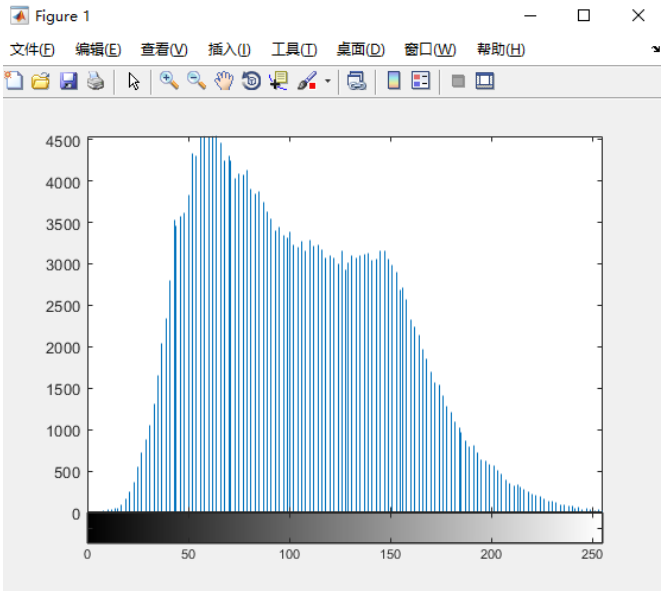


图 7- 3 拉伸过后的直方图分布

通过直方图的对比可以发现，原图像中图像的灰度主要集中在灰度值较小的区域，因此显得比较暗，经过拉伸后，图像的灰度分布更加均衡，因此图像也显得比原来更加明亮。

在本次实验的 matlab 脚本如下：首先自定义一个用于伸展直方图的函数

```
function [imgOut] = change(imgIn)
yMax = single(255);
xMin = single(min(min(imgIn)));
xMax = single(max(max(imgIn)));
imgOut = imgIn;
for idx=1:numel(imgIn)
    x = single(imgIn(idx));
    imgOut(idx)=((x-xMin))/(xMax-xMin) * yMax;
end
end
```

然后在输入图像进行直方图的拉伸

```
img_src = imread('tree.png');
img_src = rgb2gray(img_src);
imtool(img_src);
img_dst = change(img_src);
imtool(img_dst);
```

本次实验使用到的简单的直方图拉伸计算公式如下：其中，将 Ymax 设置为最大的灰度值 255,X 为当前像素的灰度值，Xmin 为图像最小的灰度值，Xmax 为图像最大的灰度值。

$$y = \frac{(X - Xmin) * Ymax}{(Xmax - Xmin)}$$

7.2 直方图拉伸的 HLS 实现

创建好工程后，以此添加好源文件和 C 仿真文件。其中头文件的定义如下：与上一章实验的内容几乎一致。

```
#ifndef __STRETCH_HEAD_H__
#define __STRETCH_HEAD_H__
//引入 stream 类
#include <hls_stream.h>
//使用 axi stream side channel(TLAST,TKEEP,TUSER,TID and so on)
#include <ap_axi_sdata.h>
```

```
7
8  //定义输入输出文件
9  #define INPUT_IMAGE     "Tree.png"
10 #define OUTPUT_IMAGE   "TreeWithAdjust.bmp"
11
12 //8 位无符号 side_channel  数据类型
13 typedef ap_axiu<8,1,1,1> uint_8_side_channel;
14 typedef hls::stream<uint_8_side_channel> AXI_STREAM;
15
16  void  Stretch(AXI_STREAM  &inStream,AXI_STREAM  &outStream,  unsigned  char  min,
unsigned char max);
17
18 #endif
```

接下来是顶层函数的源文件，如下所示：

```
1   #include "HistgramStretch.h"
2
3    void  Stretch(AXI_STREAM  &  inStream,AXI_STREAM  &  outStream,  unsigned  char  min,
unsigned char max){
4   #pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
5   #pragma HLS INTERFACE s_axilite port=max bundle=CRTL_BUS
6   #pragma HLS INTERFACE s_axilite port=min bundle=CRTL_BUS
7   #pragma HLS INTERFACE axis port=outStream
8   #pragma HLS INTERFACE axis    port=inStream
9
10   //计算 histogram 的宽度
11   float xWidth = max - min;
12   //开始处理传入进来的数据
13   uint_8_side_channel recPixel;
14   for(int idx=0;idx <(512*512);idx++){
15 #pragma HLS PIPELINE
16          //读取数据
17          uint_8_side_channel currentPixel = inStream.read();
18
19          uint_8_side_channel dataOut;
20          //取得 histogram 中的值
21          unsigned char x_t = currentPixel.data;
22          //伸展 histogram
23          float y_t_float = ((x_t-min)/xWidth) * 255;
24          unsigned char y_t = y_t_float;
25          //将数据通过 AXI stream 进行输出
```

```
26          dataOut.data = y_t;
27          dataOut.dest = currentPixel.dest;
28          dataOut.id = currentPixel.id;
29          dataOut.keep = currentPixel.keep;
30          dataOut.last = currentPixel.last;
31          dataOut.strb = currentPixel.strb;
32          dataOut.user = currentPixel.user;
33          //send the stream
34          outStream.write(dataOut);
35    }
36 }
```

4 到 8 行是接口类型的 Directive 声明，使之在映射为 RTL 电路后，能够生成对应的接口。

在函数内部，完成的即使对图像直方图的伸展，通过传入的灰度的最大值和最小值来计前面提到的计算公式的原始区间，然后通过计算得到当前像素新的灰度值。最终将结果给到输出数据流。

C 仿真文件如下：

```
1   #include "HistgramStretch.h"
2   #include <hls_opencv.h>
3   #include <stdio.h>
4   #include <opencv2/core/core.hpp>
5
6   char outImage[512][512];
7
8   void saveImages(std::string path, cv::InputArray inArr){
9     double min;
10    double max;
11    cv::minMaxIdx(inArr, &min, &max);
12    cv::Mat adjMap;
13    cv::convertScaleAbs(inArr, adjMap,255/max);
14    cv::imwrite(path,adjMap);
15 }
16
17
18 int main(){
19    printf("Load Image %s",INPUT_IMAGE);
```

```
20  cv::Mat imageSrc;
21  imageSrc = cv::imread(INPUT_IMAGE);
22  //将图像转换为灰度图像
23  cv::cvtColor(imageSrc, imageSrc,CV_BGR2GRAY);
24  printf("image row=%d,image col= %d\n",imageSrc.rows,imageSrc.cols);
25  //定义输入输出的图像数据流
26  AXI_STREAM inStream;
27  AXI_STREAM outStream;
28  //定义一个 CV 的图像矩阵
29  cv::Mat
imgCvOut(cv::Size(imageSrc.rows,imageSrc.cols),CV_8UC1,outImage,cv::Mat::AUTO_STEP);
30  //将输入的图像数据填充到 input stream 中
31  for(int idxRows=0;idxRows<imageSrc.rows;idxRows++){
32              for(int idxCols=0;idxCols<imageSrc.cols;idxCols++){
33                      uint_8_side_channel valIn;
34                      valIn.data = imageSrc.at<unsigned char>(idxRows, idxCols);
35                      valIn.dest = 1;
36                      valIn.keep = 1;
37                      valIn.last = 1;
38                      valIn.strb = 1;
39                      valIn.id = 0;
40                      valIn.user = 1;
41                      inStream << valIn;
42              }
43      }
44  //伸展 histogram
45  Stretch(inStream, outStream, (unsigned char)22, (unsigned char)154);
46  //将伸展后的图像填充到图像矩阵中
47  for(int idxRows=0;idxRows<imageSrc.rows;idxRows++){
48      for(int idxCols=0;idxCols<imageSrc.cols;idxCols++){
49              uint_8_side_channel valOut;
50              valOut = outStream.read();
51              outImage[idxRows][idxCols]=valOut.data;
52      }
53  }
54  saveImages(std::string(OUTPUT_IMAGE), imgCvOut);
55  return 0;
56 }
```

仿真文件完成的就是导入外部输入图片，然后将图像转换为数据流给到待仿真的函数，经过函数处理后，将结果输出，并保存图片。

然后依次进行 C 仿真，C 综合和 C/RTL 联合仿真。在 C 仿真结束之后，可以在 Csim 目录下，看到已经生成了经过转换后的图片。

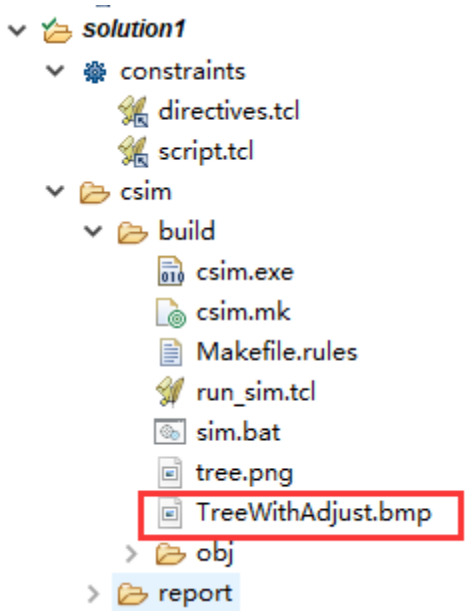


图 7- 4

依次进行上面的步骤，若没有错误即可导出 IP，然后再 Vivado 中搭建本次实验的硬件平台。

7.3 搭建硬件平台

本次实验，将使用到上一次实验中的生成的 Histogram IP，同样需要将该 IP 导入到本次工程中，再 Vivado 工程下，新建一个 ip_repo 文件夹，用来保存用户自己的 IP。

首先需要注意 HLS 生成的 IP 的保存路径。在 HLS 的 Solution 界面下，可以看到 Solution 文件夹下的 impl 目录下有一个 IP 文件夹，其中就存放着有由 HLS 生成的 IP，若在其他项目中，想要使用到这个 IP，只需复制这个文件夹到其他项目中即可。在本次实验中，我们将上一章实验和本章实验由 HLS 生成的 IP 都添加到 vivado 工程的 ip_repo 目录下。

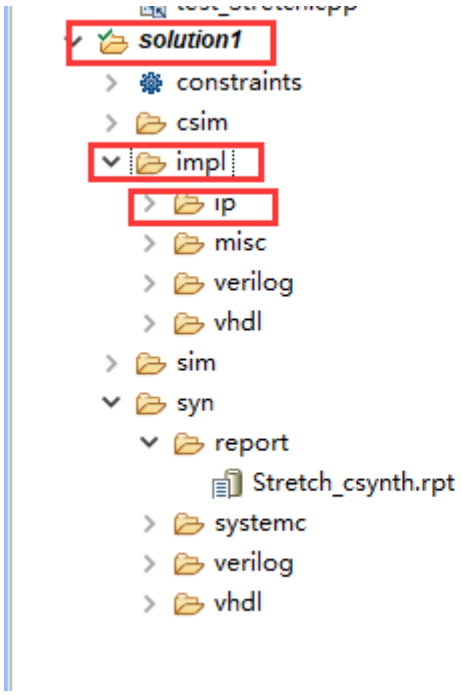


图 7- 5

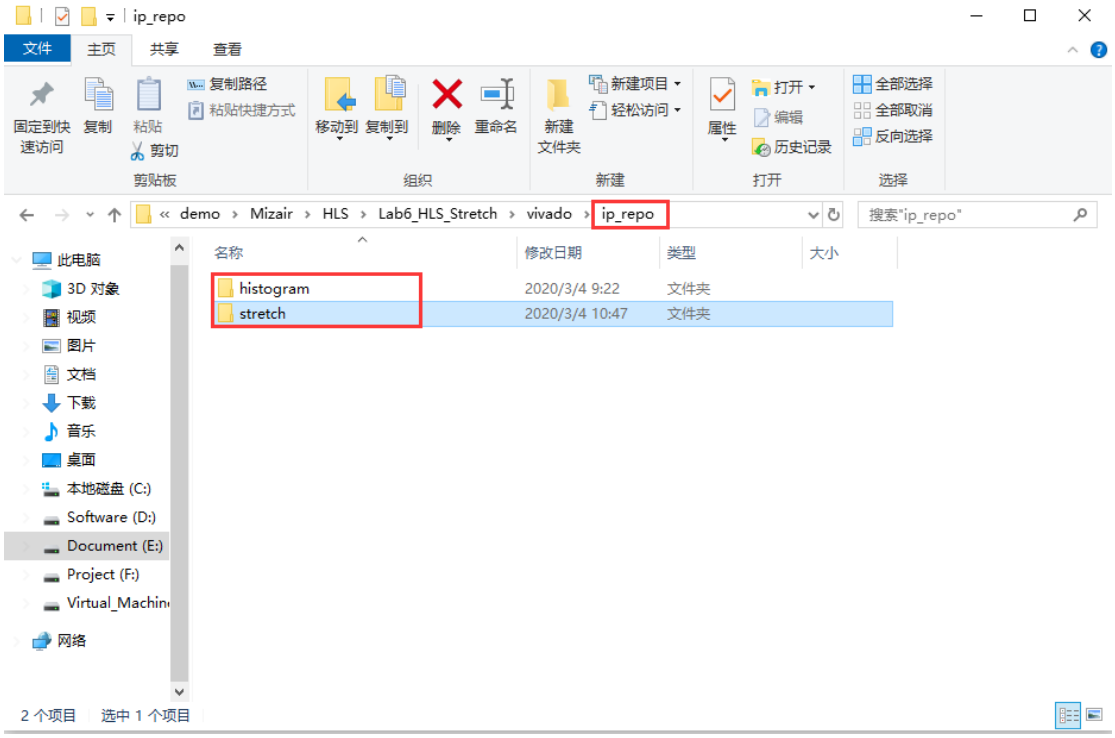


图 7- 6

创建好 vivado 工程后，进行 Block design 的搭建，具体的 Block Design 的搭建方法，与前面章节的类似，这里只对其中几个 IP 的连接做简单讲解。

在 DMA 的配置中，需要使能读写通道，如图所示：

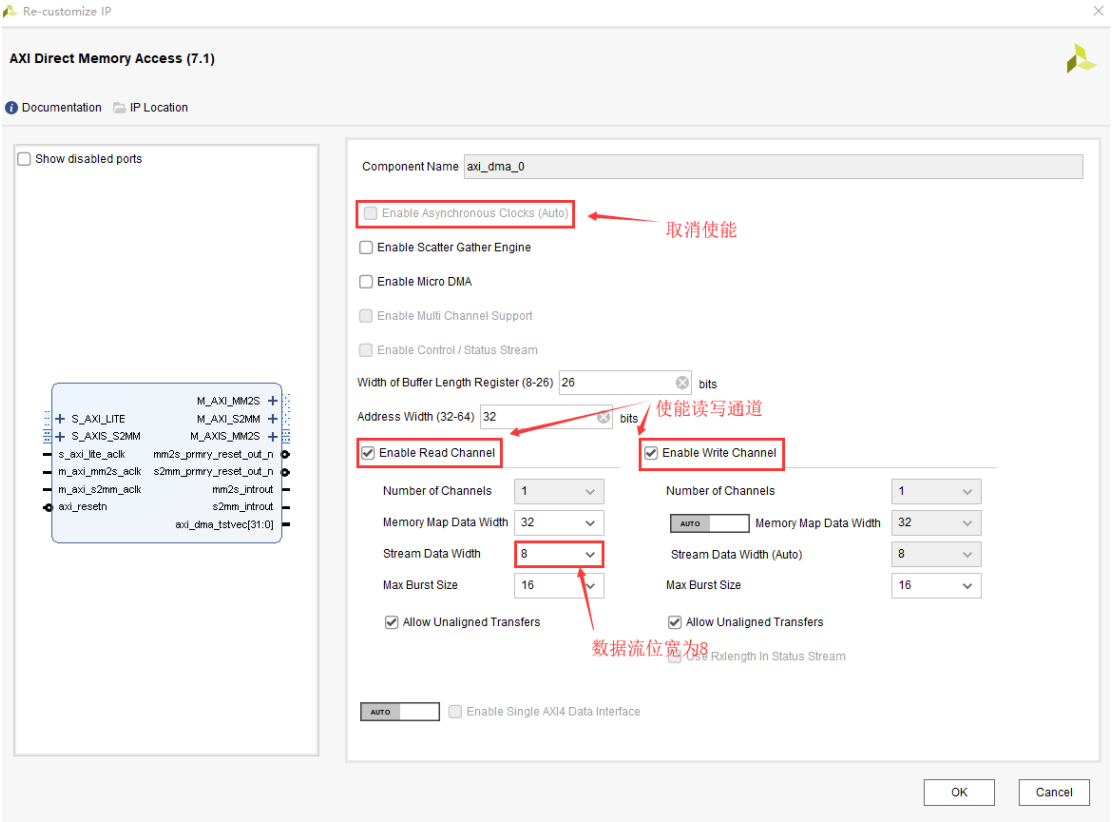


图 7- 7

本次实验将使用到 axi_broadcaster 这个 IP，这个 IP 的作用就是将 AXI_Stream 流能分成两个完全一致的流信号，类似于广播，将 AXI_Stream 数据流输出给到两个 HLS 的 IP，如图所示。

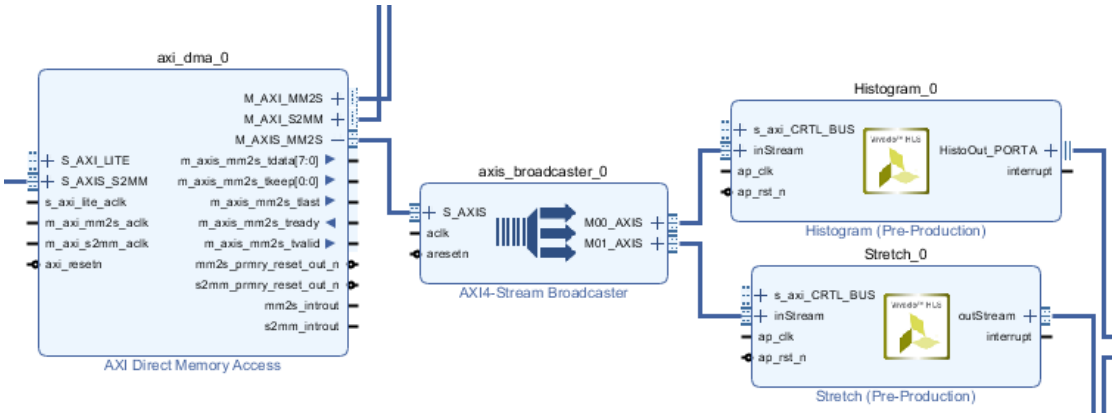


图 7- 8

此外本次实验再添加一个 axi_timmer IP 用来计算时间，用于比较通过软件和调用 IP 计算图像直方图所需要的时间。全部连接完成后如图所示：

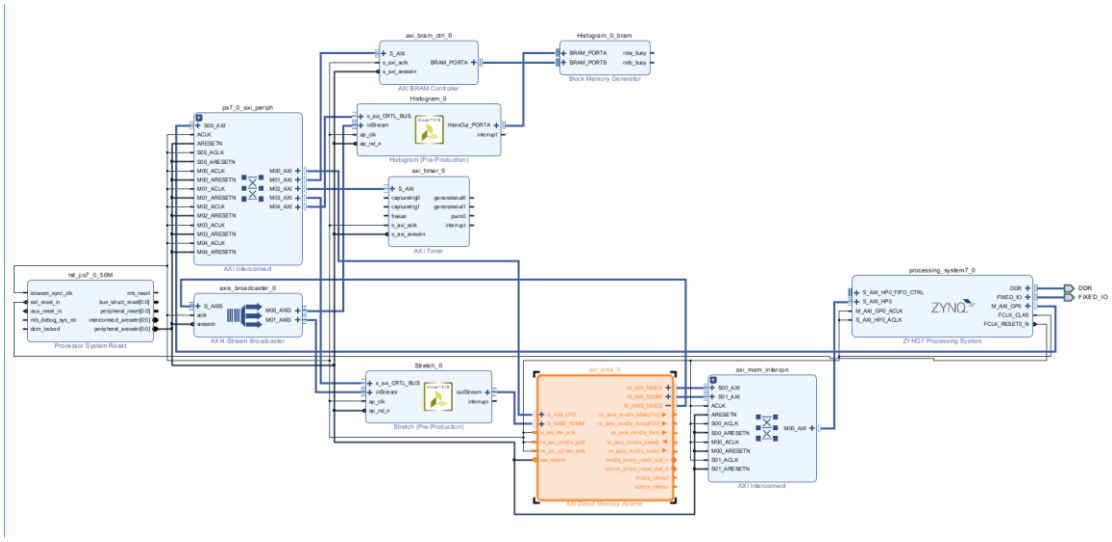


图 7- 9

7.4 创建 SDK 工程

在 SDK 设计中，首先通过软件方式求出其直方图，并找出灰度的最大值和最小值，然后通过软件方法对直方图进行拉伸。计算软件完成这些运算需要的时间。在这之后，调用 IP 完成运算，来比较硬件和软件的运行时间。

运行结果如图所示：从结果中，可以看出，软件和调用 IP 计算出来的直方图区间一致，伸展后的图像各个像素也一致，但是相较于软件，调用 HLS 生成的 IP 完成直方图的拉伸的速度却远远快于通过软件方法来完成。

```
initialize Histogram...

initialize Stretch...

initialize Timer...
initialize AXIDMA core...
initialize finished
Doing histogram on software...
Doing histogram on in software takes time :596451.000000 ns!
 The histogram in Software is between 22~154!
Doing Stretch in Software...
Doing Stretch on in software takes time : 136552.000000 ns!
 Doing Histogram in hardware...
Doing Histogram on in Hardware takes time :574468.000000 ns!
 The histogram in Hardware is between 22~154
Doing Stretch on in Hardware takes time :559661.000000 ns...
 compare the result in Software and Hardware
The result do matches well!
```

软件方法计算出来的直方图区间

软件完成拉伸所需时间

调用IP计算出来的区间

使用IP所需的时间

图 7- 10

7.5 实验总结

本次实验，完成了一个简单的直方图拉伸实验。通过拉伸直方图，提升了图像的对比度。本次实验，在上次实验的基础上完成。同时完成了对于直方图的统计。并且最终在 SDK 中完成了对于软件进行图像处理和硬件进行图像处理的速度比较，可以预见的时，对于更加大量的数据处理时，使用 FPGA 进行处理的效率能够大大提高，这也是 FPGA 的优势所在。

第8章　HLS 实现 2D Convolution(二维卷积)

8.1 实验简介:

二维卷积的使用非常广泛，常用于图像处理和深度学习中，有关二维卷积的数学原理和推导，感兴趣得用户可以去查阅相关的资料，了解二维卷积的来源，这对理解卷积的作用很有帮助。在本次实验中，主要以一种简单的方式，来介绍一下二维卷积的实现过程。

二维卷积的计算可以分为两大大步：一是预处理，将卷积核翻转 180°，然后扩充边界，一般在边界处补零；二是滑动卷积核进行卷积预算。

如图所示，是一个 sobel 边缘检测的 3x3 Kernel 的水平翻转示意图，该卷积核是卷积运算中十分重要的一部分。

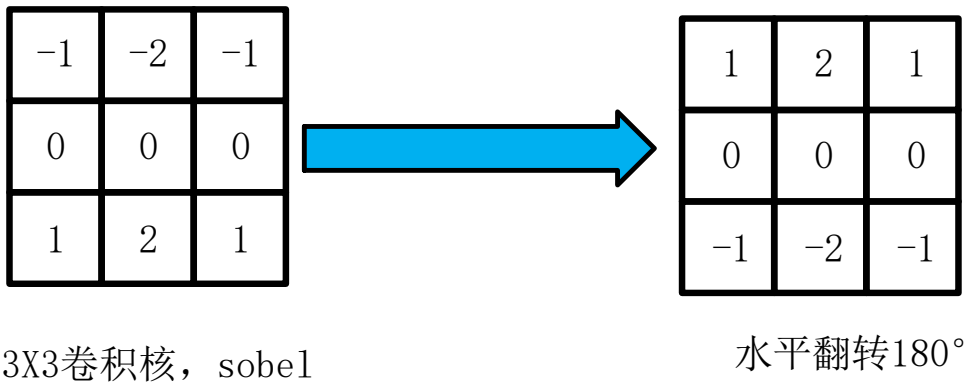


图 8- 1

如图所示的是对原始图像的补边操作，最简单的方法是在外围补上一圈 0，补零的目的是为了卷积运算完毕后的图像与原始图像的大小保持一致。如图所示，在一个 5x5 大小的图像外围填充一圈 0。

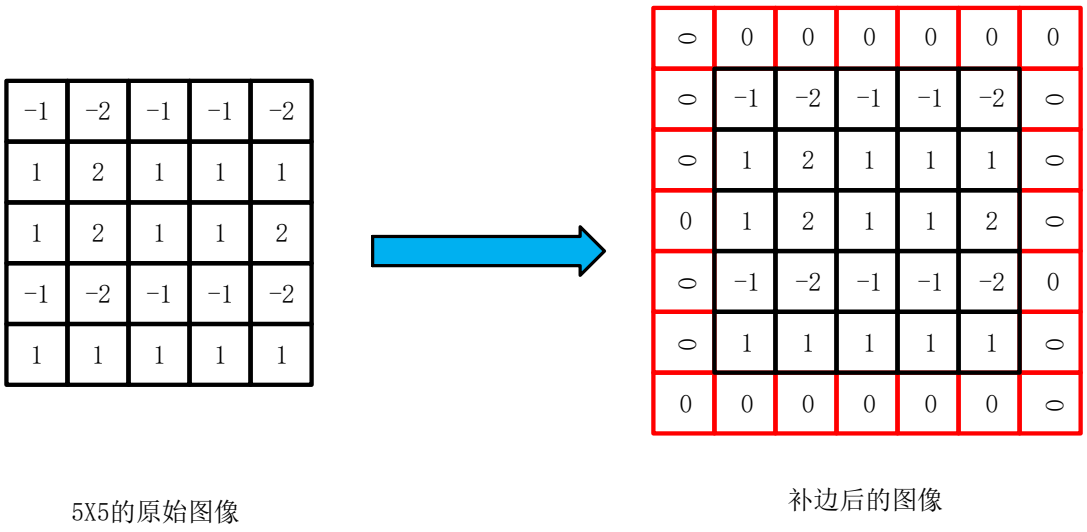


图 8- 2

滑动卷积运算其实就是将卷积核放在图像中，通过对卷积核与图像重叠的部分进行运算，并不断滑动卷积核，使得整幅图像的数据都能被卷积窗口作用。下下图所示的是进行 sobel 卷积的过程，卷积核（窗口）在图像上依次移动，对卷积核与图像重合的地方依次求得其卷积，然后将结果填充在中间像素位置。如图所示，因为在原图像外围填入了一圈 0，所以经过卷积后的图像与原来的图像保持为一样的大小，依旧是 5x5。

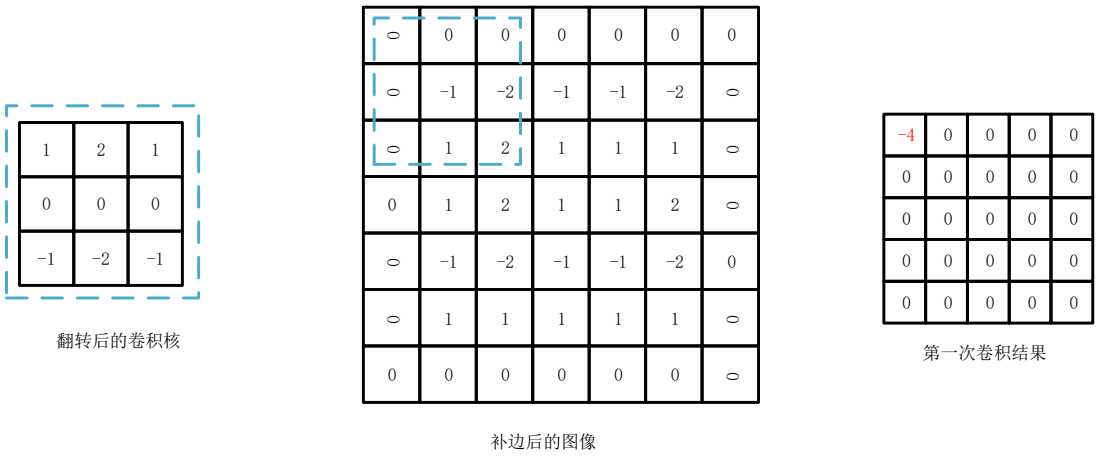


图 8- 3 第一次卷积运算

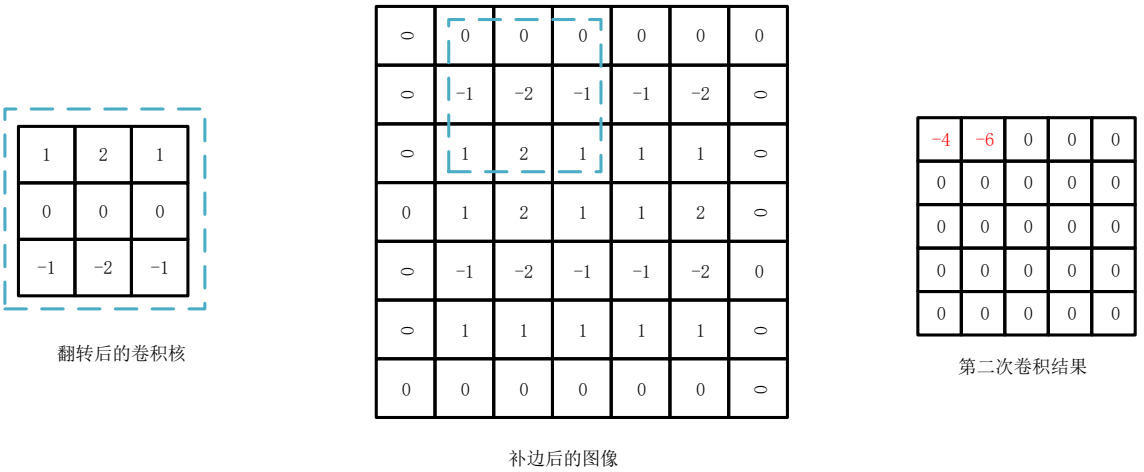


图 8- 4 第二次卷积运算

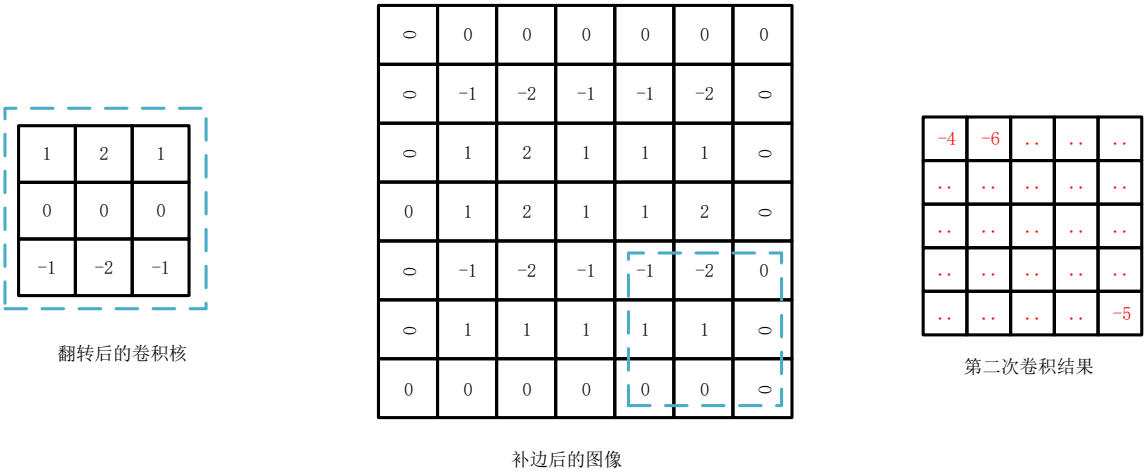


图 8- 5 最后一次卷积运算

8.2 图像腐蚀、膨胀的简单介绍

下面简单介绍一下有关图像的腐蚀和膨胀的相关知识，并通过 matlab 来直观的展示一下图像经过膨胀和腐蚀后的作用效果。膨胀与腐蚀的作用有：消除噪声、分割(isolate)独立的图像元素，以及连接(join)相邻的元素，寻找图像中的明显的极大值区域或极小值区域。

8.2.1 图像腐蚀

图像腐蚀简单来说就是找寻区域中最小的极小值，经过腐蚀后的图像亮度相对于没腐蚀之前降低。在 matlab 中使用如下脚本即可快速的观察到对图像腐蚀过后的结果。

```
img_rgb=imread('Lena.bmp');        %载入路径下的图像
img_gray=rgb2gray(img_rgb);
imtool(img_gray);
se=strel('disk',1);            %生成圆形结构元素
I2=imerode(img_gray,se);              %用生成的结构元素对图像进行腐蚀
imtool(I2);
```

运行后的结果如图所示，左边是原来的图像，右边是经过腐蚀过后的图像，还是能够看出来腐蚀过后的图像与原图像的区别的，相较于原图像，图像的噪点明显减少，且亮度降低。

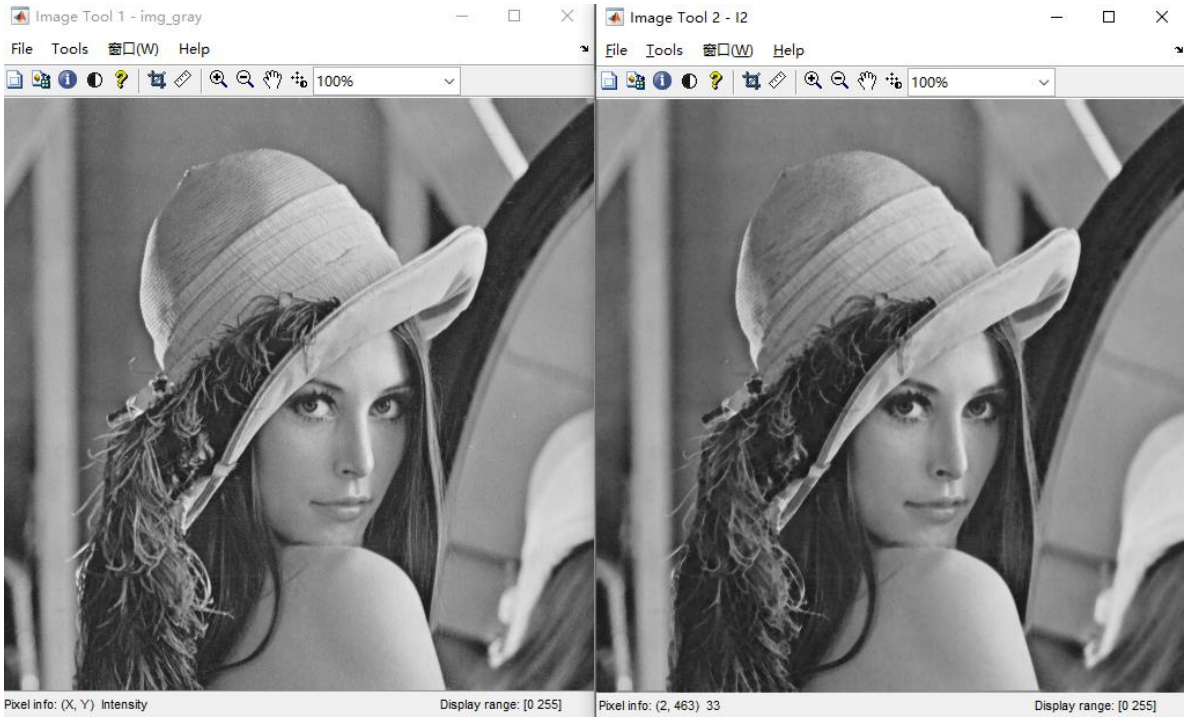


图 8- 6 图像腐蚀

在本次实验中，相当于一个 3x3 的卷积窗口滑动到图像上，返回这个卷积窗口中的最小值，作为中间像素的值。

8.2.2 图像膨胀

膨胀与腐蚀的作用相反，是寻找区域中的极大值，经过膨胀后的图像，亮度会比原图像增加。在 matlab 中运行一下脚本即可快速的得到膨胀后的结果。

```
img_rgb=imread('Lena.bmp');      %载入路径下的图像
img_gray=rgb2gray(img_rgb);
imtool(img_gray);
se=strel('disk',1);          %生成圆形结构元素
I2=imdilate(img_gray,se);            %用生成的结构元素对图像进行腐蚀
imtool(I2);
```

运行后的结果如图所示：可以看到图像的白色部分向外扩张，图像的亮度相较于之前有所提高。

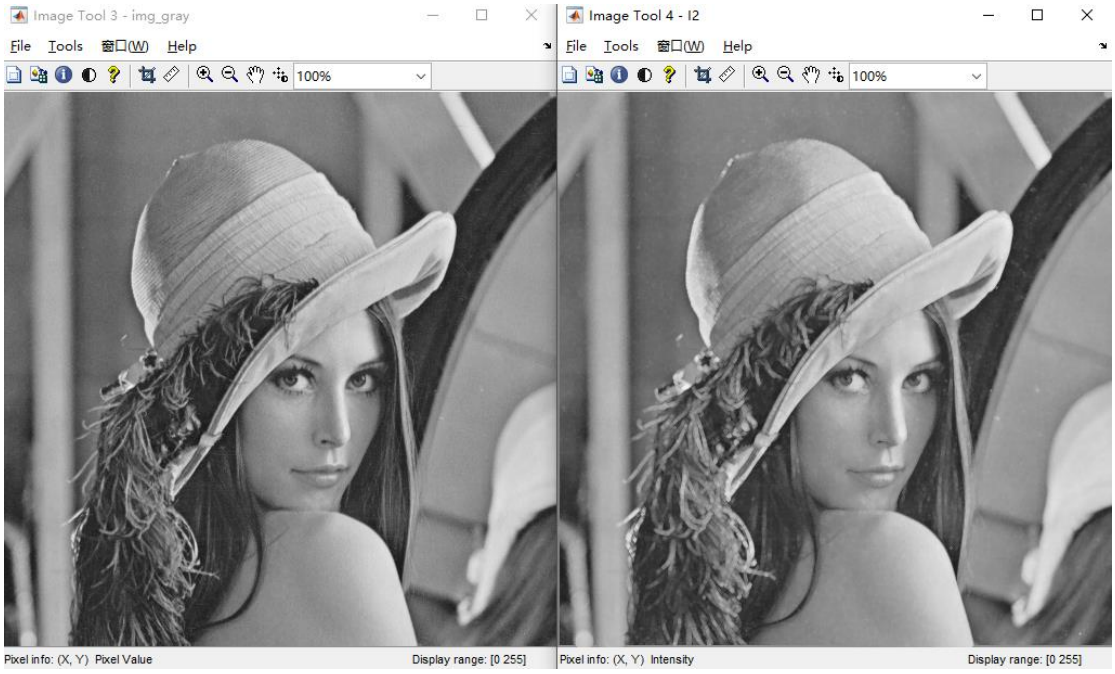


图 8- 7 图像膨胀

在本次实验中，相当于一个 3x3 的卷积窗口滑动到图像上，返回这个卷积窗口中的最小值，作为中间像素的值。

8.3 Sobel 边缘检测

边缘检测是在工业生产中，经常会使用到的一种手段，常用于定位，提取轮廓等场合。在本章中，将使用 FPGA 完成简单的边缘检测算法——sobel 边缘检测算法。首先需要了解以下 sobel 算子。

Sobel 算子

索贝尔算子（Sobel operator）主要用作边缘检测，在技术上，它是一离散性差分算子，用来运算图像亮度函数的灰度之近似值。在图像的任何一点使用此算子，将会产生对应的灰度矢量或是其法矢量

Sobel 算子的卷积因子如下图：

Sobel卷积因子为：

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

在使用该算子的时候，将该算子与图像中对应的像素点相乘后求和，能够得到图像在水平和垂直方向的梯度值。例如利用该算子与如下的 3X3 的像素窗口进行卷积

A1	A2	A3
B1	B2	B3
C1	C2	C3

可以得到水平和垂直方向的梯度计算结果如下：

Dx= (A3-A1) + 2*(B3-B1) + (C3-C1);

Dy= (A1-C1) + 2*(A2-C2) + (A3-C3);

图中图像的像素窗口的值表示的是图像的灰度值，使用该算子求得的梯度的最终结果还是比较简单，在求得该梯度后需要使用如下公式来判断该点的梯度：

$$\mathbf{G} = \sqrt{\mathbf{G_x}^2 + \mathbf{G_y}^2}$$

一般情况下，为了计算的方便，一般使用如下公式来计算图像的梯度：

$$|G| = |Gx| + |Gy|$$

若求得的最终梯度 G 大于某一阈值，则可以判定该点是图像的边缘。

8.4 HLS 视频库中的 Linebuffer 和 Window 类

在本次实验中，不会使用 HLS 自带的边缘检测，膨胀腐蚀函数，而是通过使用 HLS 视频库中的 Linebuffer 和 Window 类来完成相关的图像算法的开发。用户也可以直接调用 HLS 现成的函数，完成图像处理的相关操作。

8.4.1 Linebuffer 的简单介绍

通过查阅手册 UG902 可以得到 Xininx 对 Linebuffer 的介绍。

LineBuffer 类是一个 c++类，它允许用户在算法代码中轻松地声明和管理行缓冲区。该类提供了实例化和使用行缓冲区所需的所有方法。LineBuffer 类可以处理所有数据类型。

Linebuffer 主要有以下 5 个方法，分别是：

- shift_pixels_up()
- shift_pixels_down()
- insert_bottom_row()
- insert_top_row()
- getval(row,column)

图 8- 8 Linebuffer 常用方法

根据这些方法的名称很容易能够理解各个方法的作用。Shift_pixel_up()是将指定列的像素向上移动一个。Insert_bottom_row()是在指定列的最后一行添加一个数据。Getval(row,column)能够返回指定像素点的值。

下面以一个例子来简单说明，如何声明一个 Linebuffer 的对象并且使用以上方法。

hls::LineBuffer<rows, columns, type> variable;是 Linebuffer 类的模板，通过使用这个模板即可生成一个行缓冲区。

如 hls::LineBuffer<3,5,unsigned char> Buff_A;声明了一个 3 行 5 列，每个点中是无符号 char 类型的数据的行缓冲区。

下面是对该图像使用 shift_down 和 insert_top 方法后的简单示例

```
Buff_A.shift_pixels_down(2);

Buff_A.insert_top_row(100,2);
```

Table 2-8: **Data Set After Shift Down and Insert Top Classes Used**

Line	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	1	2	100	4	5
Row 1	6	7	3	9	10
Row 2	11	12	8	14	15

图 8- 9

如上图所示，使用 shift_pixels_down 和 insert_top_row 方法后，在第零行和

第二列添加了一个新的数 100。

有关 Linebuffer 的其他方法在手册里也有详细描述，用户可以查阅 UG902 中 Linebuffer 这以节的内容来了解其他方法。其实，通过方法函数的名称也能很容易理解函数的作用。

8.4.2 Window 类的简单介绍

HLS 中除了提供了行缓冲区外，也提供了一个窗缓冲区，便于用户来进行卷积运算。同 Linebuffer 类似，window 类也具有如下方法：

- shift_pixels_up()
- shift_pixels_down()
- shift_pixels_left()
- shift_pixels_right()
- insert_pixel(value,row,colum
- insert_row()
- insert_bottom_row()
- insert_top_row()
- insert_col()
- insert_left_col()
- insert_right_col()
- getval(row, column)

通过方法名称也能很容易知道该函数的作用，下面介绍以下一个比较常用的方法，insert_pixel(value,row,column);该方法的作用是在窗口指定的位置填入一个数。

首先使用如下模板定义一个 window 类的变量：

```
hls::Window<row, column, type> variable;
```

下面以一个 3x3 的 char 类型窗口为例：

```
hls::window<3,3,char>    window_B;
```

window_B 调用插入数据的方法后的结果如下：

```
Buff_B.insert_pixel(100,1,1);
```

Table 2-15: **Memory Window Data Set After Insertion Operation at Location 1,1**

Column 0	Column 1	Column 2	Row
1	2	3	Row 0
6	100	8	Row 1
11	12	13	Row 2

图 8- 10 在第 1 行第 1 列插入值 100

有关 window 类的其他方法就不做过多解释，也比较好理解，用户有不理解的地方只需查阅手册 UG902。

8.5 2D convolution 的 HLS 实现

8.5.1 添加头文件

接下来，在 HLS 中实现二维卷积，首先是头文件，头文件中对一些常用的数据机构类型进行重定义。

```
#ifndef __TEST_HEAD_H__
#define __TEST_HEAD_H__

#include <hls_video.h>
#include <ap_axi_sdata.h>

#define KERNEL_DIM 3
#define IMG_WIDTH 512
#define IMG_HEIGHT 512
#define INPUT_IMAGE "Lena.bmp"
#define OUTPUT_IMAGE "LenaWithAdjust.bmp"

//定义数据流中的数据
typedef ap_axiu<8,1,1,1>        uint8_side_channel;
typedef ap_axis<8,1,1,1>        int8_side_channel;
//定义 AXI—Stream 数据流
typedef hls::stream<uint8_side_channel>                AXI_STREAM;
//定义 char 类型的卷积窗口
typedef hls::Window<KERNEL_DIM,KERNEL_DIM,char>            CHAR_WINDOW;
```

```
20 //定义 short 类型的卷积窗口
21 typedef hls::Window<KERNEL_DIM,KERNEL_DIM,short>          SHORT_WINDOW;
22 //定义 LineBuffer 用于延时
23 typedef hls::LineBuffer<KERNEL_DIM,IMG_WIDTH,unsigned char>      BUFFER;
24
25 //参考函数
26 short convolved2d(CHAR_WINDOW * window,char kernel[KERNEL_DIM * KERNEL_DIM]);
27
28
29 //求卷积和
30 short sumWindow(SHORT_WINDOW *window);
31 //求卷积窗口中的最小值
32 short minWindow(SHORT_WINDOW *window);
33 //求卷积窗口中的最大值
34 short maxWindow(SHORT_WINDOW *window);
35
36 //顶层函数
37 void imagePrco(AXI_STREAM & inStream,AXI_STREAM & outStream, char
kernel[KERNEL_DIM*KERNEL_DIM], int operation);
38
39
40 #endif
```

第 7 行定义了卷积核的大小，第 8，9 行定义了要处理的图像的大小，第 10，11 行定义了输入输出图像的路径，14，15 行定义了流中的像素点，17 行定义了数据流的类型，19，21 行定义了两个 window 类的对象，第 23 行定义了 linebuffer 类的对象。29 到 34 行声明了对卷积核中的数据进行的操作，其中 sumWindow 用于求得卷积窗口中的和。minWindow 用于求得卷积窗口的最小值(Erode 腐蚀)，maxWindow 用于返回卷积窗口中的最大值(dilate 膨胀)。

37 行声明了本次实验的顶层函数，对顶层函数实现功能的改变只需改变输入给顶层函数的卷积核 kernel 和选择的操作即可。

8.5.2 添加源文件

在源文件中，添加如下内容，用于实现顶层函数。

```
1    #include "kernel.h"
2
```

```
void imagePrco(AXI_STREAM & inStream,AXI_STREAM & outStream, char kernel[KERNEL_DIM*KERNEL_DIM], int operation)
{
#pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=operation bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=kernel bundle=KERNEL_BUS
#pragma HLS INTERFACE axis register both port=outStream
#pragma HLS INTERFACE axis register both port=inStream

    BUFFER lineBuf;
    SHORT_WINDOW window;

    //用来标记行和列的索引值
    int idxCol = 0;
    int idxRow = 0;
    int pixConvolved = 0;

    //计算延时，填充 line_buffer
    int waitPixel = (IMG_WIDTH *(KERNEL_DIM-1) + KERNEL_DIM)/2;
    int cntWait = 0;
    int sentPixel = 0;

    uint8_side_channel dataOutChannel;
    uint8_side_channel currentChannel;

    //迭代所有输入的像素
    for(int idxPix=0;idxPix<(IMG_HEIGHT*IMG_WIDTH);idxPix++)
    {
#pragma HLS PIPELINE

        currentChannel = inStream.read();
        unsigned char inPix = currentChannel.data;

        //put data on the line buffer
        lineBuf.shift_pixels_up(idxCol);
        lineBuf.insert_bottom_row(inPix, idxCol);

        //put data on the window, and multiply by kernel
        for(int idxWinRow = 0; idxWinRow < KERNEL_DIM; idxWinRow++)
        {
            for(int idxWinCol=0; idxWinCol < KERNEL_DIM; idxWinCol++)
```

```
43                  {
44                      //idxWinCol and pixConvolved will slide the window
45                      short    val    =    (short)lineBuf.getval(idxWinRow,    idxWinCol    +
pixConvolved);
46                      //采样窗口中的数据与 Kernel(卷积算子)相乘
47                      val = (short)kernel[(idxWinRow * KERNEL_DIM) + idxWinCol] * val;
48                      window.insert(val,idxWinRow,idxWinCol);
49                  }
50              }
51
52
53              short valOut=0;
54
55              //避免计算图像的边界()
56              if((idxRow >= KERNEL_DIM-1) && (idxCol >= KERNEL_DIM -1))
57              {
58
59                  switch(operation)
60                  {
61                  case 0:
62                  {
63                      valOut = sumWindow(&window);
64                      if(valOut < 0 ){
65                          valOut = 0;
66                      }
67                      break;
68                  }
69
70                  case 1:
71                  {
72                      valOut = minWindow(&window);
73                      break;
74                  }
75
76                  case 2:
77                  {
78                      valOut = maxWindow(&window);
79                      break;
80                  }
81                  default:;
82                  }
```

```
83                  pixConvolved ++;
84              }
85
86              cntWait++;
87              if(cntWait > waitPixel){
88                  dataOutChannel.data = (unsigned char)valOut;
89                  dataOutChannel.dest = currentChannel.dest;
90                  dataOutChannel.id = currentChannel.id;
91                  dataOutChannel.keep = currentChannel.keep;
92                  dataOutChannel.strb = currentChannel.strb;
93                  dataOutChannel.user = currentChannel.user;
94                  dataOutChannel.last = 0;
95                  outStream.write(dataOutChannel);
96                  sentPixel++;
97              }
98              //计算行和列的索引值
99              if(idxCol < IMG_WIDTH - 1){
100                 idxCol ++;
101             }
102             else{
103                 idxCol = 0;
104                 idxRow++ ;
105                 pixConvolved = 0;
106             }
107             //send data to the stream
108
109         }
110         for(cntWait = 0; cntWait < waitPixel; cntWait++){
111         dataOutChannel.data = 0;
112         dataOutChannel.dest = currentChannel.dest;
113         dataOutChannel.id = currentChannel.id;
114         dataOutChannel.keep = currentChannel.keep;
115         dataOutChannel.strb = currentChannel.strb;
116         dataOutChannel.user = currentChannel.user;
117         if(cntWait < waitPixel - 1)
118             dataOutChannel.last = 0;
119         else
120             dataOutChannel.last = 1;
121         outStream.write(dataOutChannel);
122         sentPixel++;
123         }
```

```
}


//求卷积和
short sumWindow(SHORT_WINDOW *window){
	short sum=0;
	for(int idxRow = 0; idxRow < KERNEL_DIM; idxRow++){
		for(int idxCol = 0;idxCol < KERNEL_DIM;idxCol++){
			sum += (short)window->getval(idxRow, idxCol);
		}
	}
	return sum;

}
//求卷积窗口中的最小值
short minWindow(SHORT_WINDOW *window){
	unsigned char minVal = 255;
	for(int idxRow = 0; idxRow < KERNEL_DIM; idxRow++){
		for(int idxCol = 0; idxCol < KERNEL_DIM; idxCol++){
			unsigned char valInWindow;
			valInWindow = window->getval(idxRow, idxCol);
			if(valInWindow < minVal){
				minVal = valInWindow;
			}
		}
	}
	return minVal;
}
//求卷积窗口中的最大值
short maxWindow(SHORT_WINDOW *window){
	unsigned char maxVal = 0;
	for(int idxRow=0;idxRow<KERNEL_DIM;idxRow++){
		for(int idxCol=0;idxCol<KERNEL_DIM;idxCol++){
			unsigned char valInWindow;
			valInWindow = window->getval(idxRow, idxCol);
			if(valInWindow > maxVal){
				maxVal = valInWindow;
			}
		}
	}
```

```
165        return maxVal;
166 }
```

源文件中，5~9 行声明了定义了函数的接口类型，在综合生成 IP 的时候，能够将该 IP 综合成所期望的接口类型。第 28 行的 for 循环对所有输入的图像像素进行迭代，第 36 和 37 行调用 linebuffer 的方法，将输入的数据填充到 Linebuffer 的缓冲区中，第 40 到 50 行完成了卷积窗口中对应，kernel 和对应像素相乘，并将乘积填充在卷积窗口的对应位置。56 到 84 行正式进行卷积运算，防止计算图像的边缘。87 到 97 行将卷积计算后的结果给到输出数据流。

110 行到 125 行对缺失的图像进行补齐。因为本次实验中，没有在预处理阶段进行补零，所以最终得到的图像将会相较于原图像少了两行两列。若不将输出像素补充完整，将会给出如图所示警告，最终将会导致在 SDK 中调用生成的 IP 时会不能工作。出现如图所示警告，一般情况下是因为图像中的像素不匹配导致，输入输出数据流中需要有 tlast，tuser 等信号，若不指定，将会出现这个问题。

```
WARNING: Hls::stream 'hls::stream<ap_axiu<8, 1, 1, 1> >.2' is read while empty, which may result in RTL simulation hanging.
WARNING: Hls::stream 'hls::stream<ap_axiu<8, 1, 1, 1> >.2' is read while empty, which may result in RTL simulation hanging.
WARNING: Hls::stream 'hls::stream<ap_axiu<8, 1, 1, 1> >.2' is read while empty, which may result in RTL simulation hanging.
WARNING: Hls::stream 'hls::stream<ap_axiu<8, 1, 1, 1> >.2' is read while empty, which may result in RTL simulation hanging.
WARNING: Hls::stream 'hls::stream<ap_axiu<8, 1, 1, 1> >.2' is read while empty, which may result in RTL simulation hanging.
WARNING: Hls::stream 'hls::stream<ap_axiu<8, 1, 1, 1> >.2' is read while empty, which may result in RTL simulation hanging.
WARNING: Hls::stream 'hls::stream<ap_axiu<8, 1, 1, 1> >.2' is read while empty, which may result in RTL simulation hanging.
WARNING: Hls::stream 'hls::stream<ap_axiu<8, 1, 1, 1> >.2' is read while empty, which may result in RTL simulation hanging.
```

图 7- 11

在 117 到 120 行中，完成的是对整个数据流的最后一个像素进行判断，标志着数据流的结束。这是为了与 AXI-stream 的接口协议吻合。

8.5.3 添加 C 仿真文件

本次实验的仿真文件与前面章节中的实验的仿真文件差不多，这里只对其中有使用到的几个卷积核 kernel 进行简单的讲解。

1) Impulse Kernel

0	0	0
0	1	0
0	0	0

该 kernel 不对图像进行任何处理，就将原来的输入图像进行输出即可。若选择该 Kernrl，并设置 operation 为 0，经过二维卷积后的图像与原图像的灰度图像如图所示：

图 8- 12 卷积后的图像

图 8- 13 灰度图像

仔细观察可以发现，C 仿真得到的图片的四周少了两行两列，这与前面在源代码出提到的相吻合，因为没有对原始图像进行补零操作，将会导致输出图像较原图像缺少两行两列。

2) Kernel for Erode or Dilate

1	1	1
1	1	1
1	1	1

若选择该 kernel 并将 operation 设置为 1 或 2,就可以对图像进行膨胀和腐蚀。该 Kernel 的作用就是用来便于寻找卷积域内的最小值和最大值，下面 C 仿真后的腐蚀后的图像与膨胀后的图像：

图 8- 14 腐蚀过后的图像

图 8- 15 膨胀过后的图像

3) Sobel kernel

−1	−2	−1
0	0	0
1	2	1

上面所示的是垂直方向的 sobel kernel，若选择该 kernel 并设置 operation 为 0，即求得卷积窗口中的和，一次来确定图像的边缘。

使用该 kernel 并设置 operation 为 0 后的 C 仿真结果如下：

图 8- 16 sobel kernel 的卷积结果

4) Edge detect Kernel

除了上面介绍的 sobel 边缘检测算子外，这里再介绍一种边缘检测的算子。

−1	−1	−1
−1	8	−1
1	−1	1

使用该算子，并设置 operation 为 0,进行 C 仿真后的结果如下：

图 8- 17 边缘检测

可以看到，相较于 sobel 边缘检测，该算子在检测边缘时效果更好。因为上面的 sobel 算子只是垂直方向上的算子。

8.6 实验总结

在本章中，完成了有关的简单的图像算法的 HLS 实现，包括图像的腐蚀膨胀，以及简单的边缘检测等。介绍了 HLS 图像处理中两个比较重要的类，Linebuffer 和 Window，通过使用这两个类，能够帮助用户完成有关二维卷积的算法的开发。

在进行完本次实验之后，应导出 IP，在下次实验中，我们将使用本次实验生成的 IP，在 Vivado 和 SDK 中调用，将图像通过 HDMI 显示在显示器上。

本次实验中，最终要的部分便是对一开始所介绍的二维卷积的实现过程的理解，理解了卷积实现的过程，在进行相关图像处理算法开发的过程中，就能提高效率。

本次实验中还存在一些优化的问题，笔者在之后的版本中将会去逐步完善，通过添加 Directives 来对，数组 for 循环进行优化，提高 IP 核的效率。当前生成的 IP，已经可以正常使用。

第9章　图像腐蚀、膨胀、边缘检测综合实验

9.1 实验简介

在上次实验中，已经在 HLS 完成了算法的开发，并生成了 IP 核，在本次实验中，来调用上次实验生成的 IP，让所进行的图像处理的操作能够直观地显示在 HDMI 显示器上。

在本次实验中，将会使用到 VDMA 来快速地将图像显示在显示器上。

9.2 搭建 Block Design

9.2.1 视频输出部分

1)　添加 VDMA

在搜索栏中搜索 VDMA，然后添加 VDMA 到 Block Design 中。

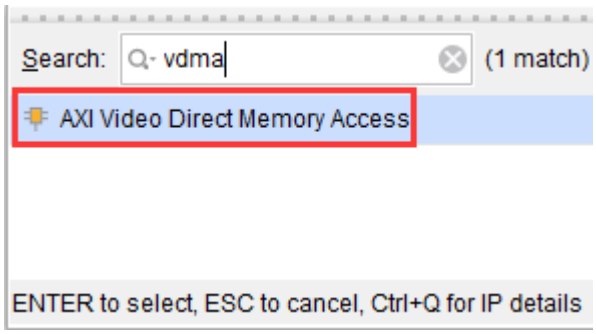


图 9- 1

添加 VDMA 后对 VDMA 进行如下配置：

取消写通道使能，设置读通道数据流位宽位 32 位，设置 linebuffer 深度位 2048。

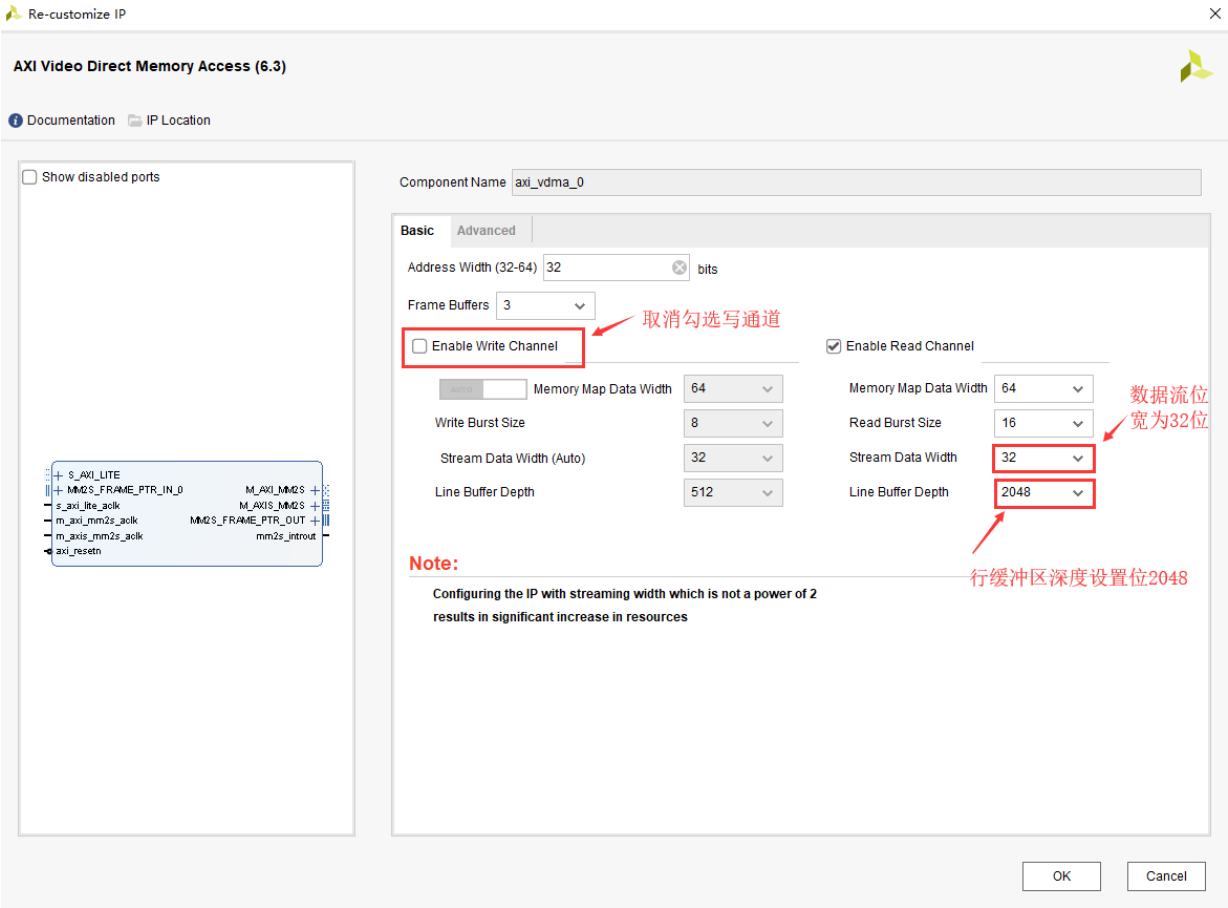


图 9- 2 VDMA 配置

2) Video out IP

添加 video out IP 通过该 IP 来提供输出的视频信号，将其设置位独立时钟模式。

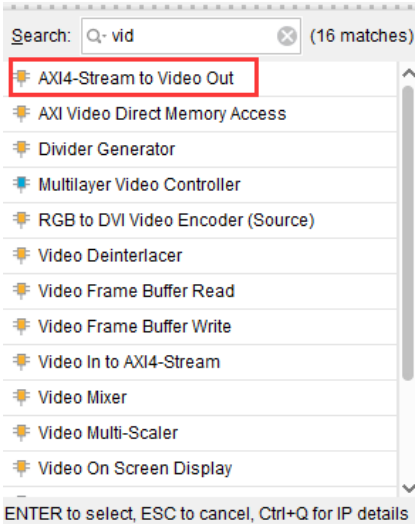


图 9- 3

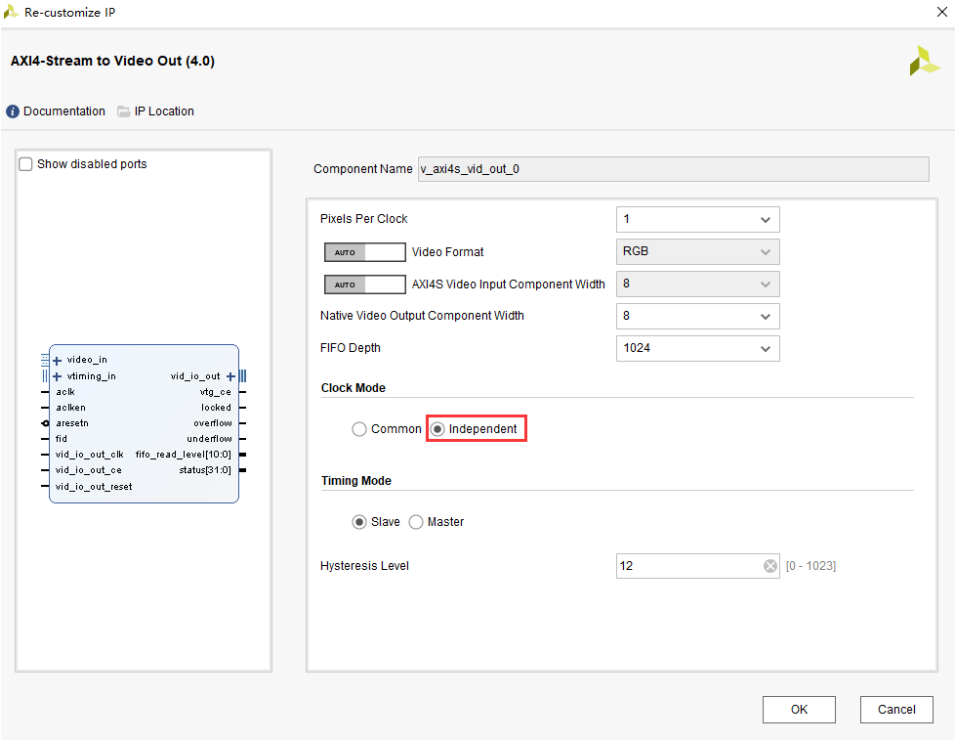


图 9- 4

然后将 VDMA 域 video out IP 相连接。

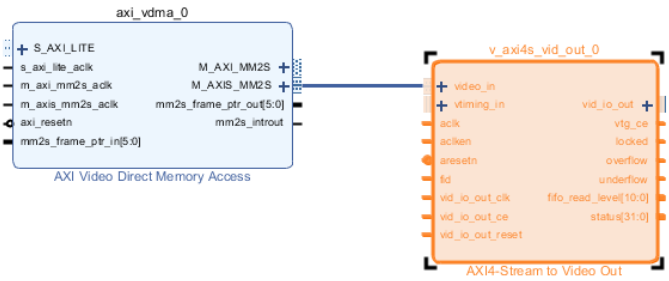


图 9- 5

3) Video timing IP

添加 Video Timing Controller IP，用来给输出视频输出提供同步信号。

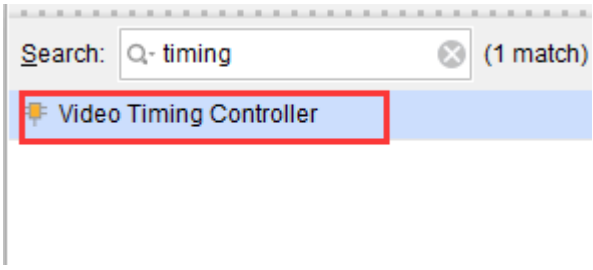


图 9- 6

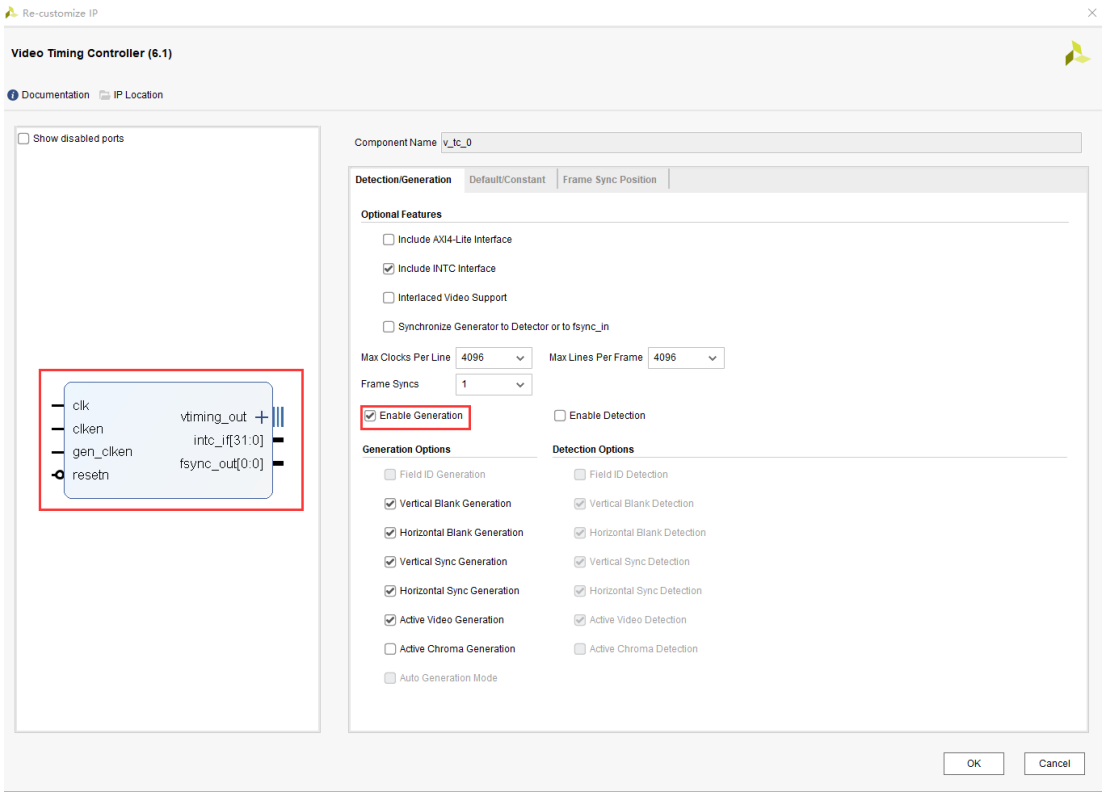


图 9- 7

然后将输出的 timing 信号连接到 Video Out IP 的 timing 接口上。

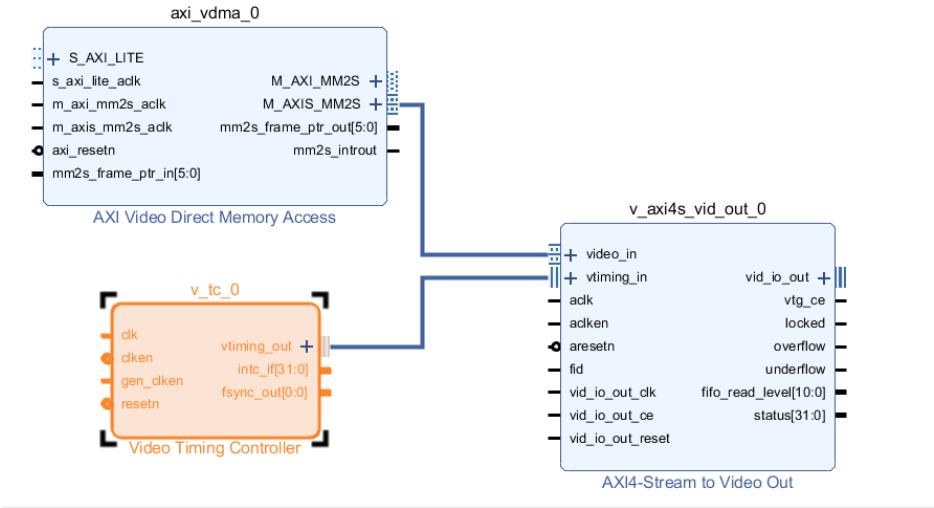


图 9- 8

4) 添加 HDMI 输出 IP

HDMI 输出 IP 完成将并行的数据，通过编码后以串行的方式发送到显示器端。这是一个自定义的 IP，在本次实验中，采用是一个已经封装好的 IP，在 ip_repo 的目录下，用户只需将 IP 添加到仓库中，即可使用该 IP。

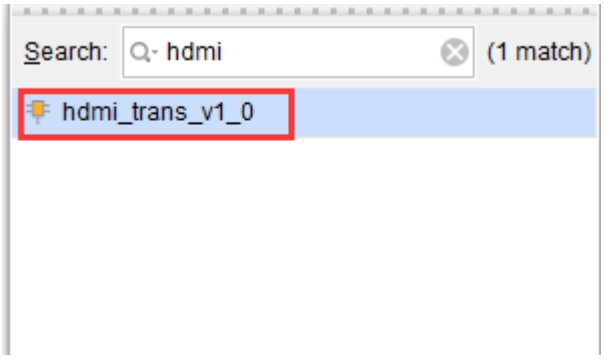


图 9- 9

然后将 video out IP 输出的信号给到 HDMI 输出模块。

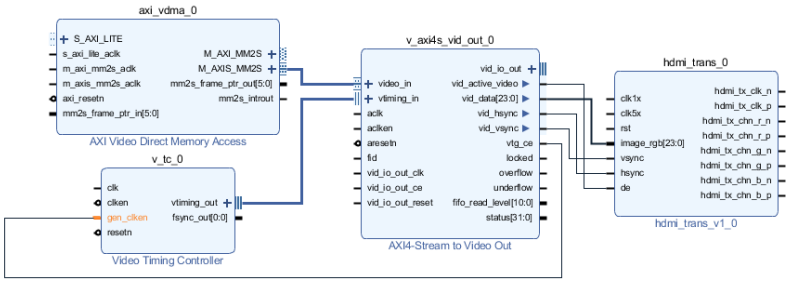


图 9- 10

5) 添加 PLL

由于 HDMI 输出模块需要的有两个时钟，一个是像素时钟，一个是串行输出所需要的时钟，所以需要使用一个 PLL 来产生对应频率的时钟。

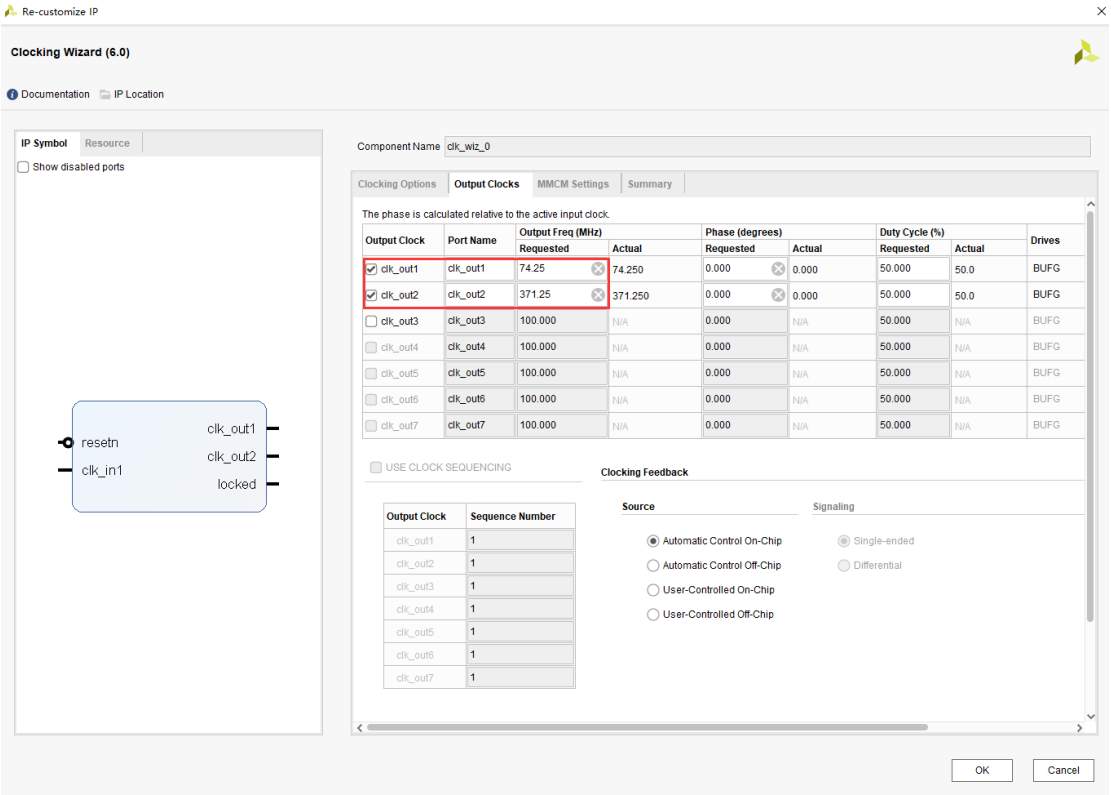


图 9- 11

接下来，连接与图像输出相关的 IP 的时钟和复位信号

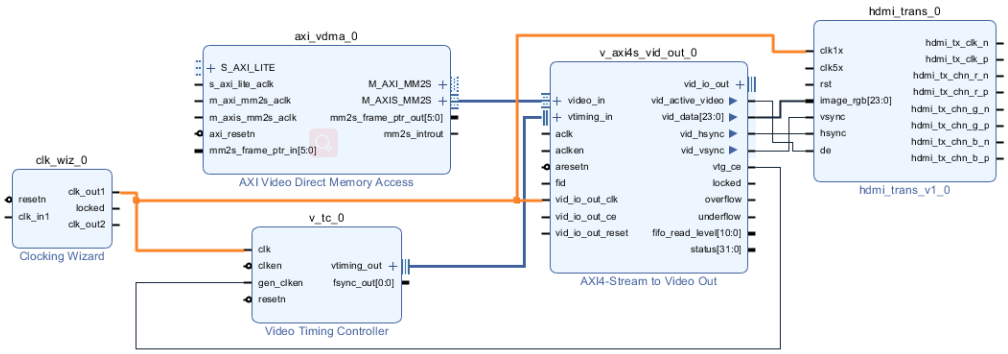


图 9- 12 像素时钟连接

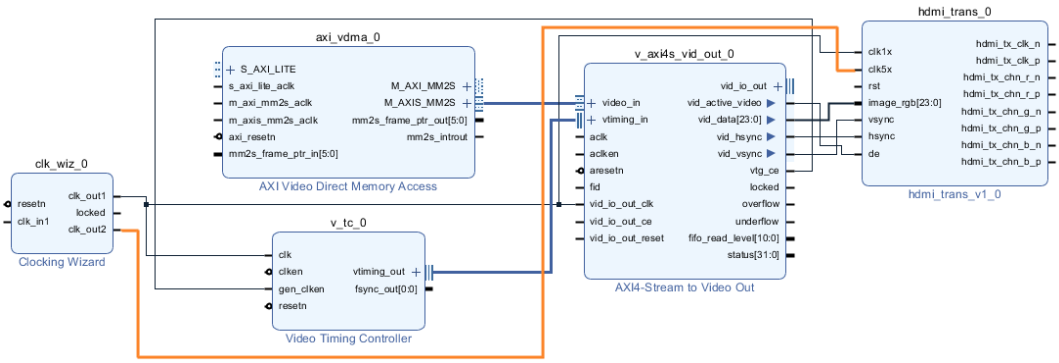


图 9- 13 串行时钟连接

各个与图像输出相关的复位信号，其中有些模块需要高复位，有些模块需要低复位，添加一个 vector IP 用来转换简单的电平信号。

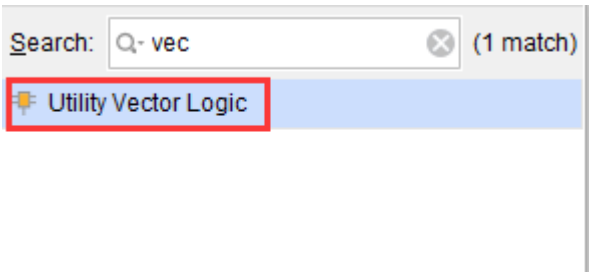


图 9- 14

将该 IP 设置为一位取反操作。

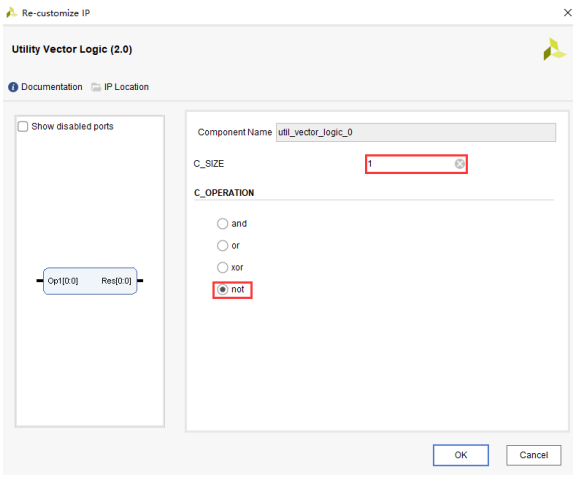


图 9- 15

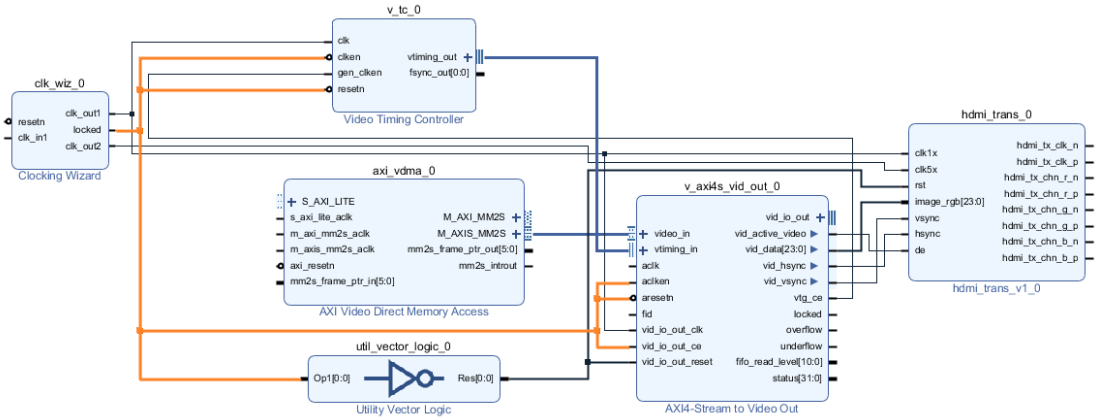


图 9- 16 低电平复位连接

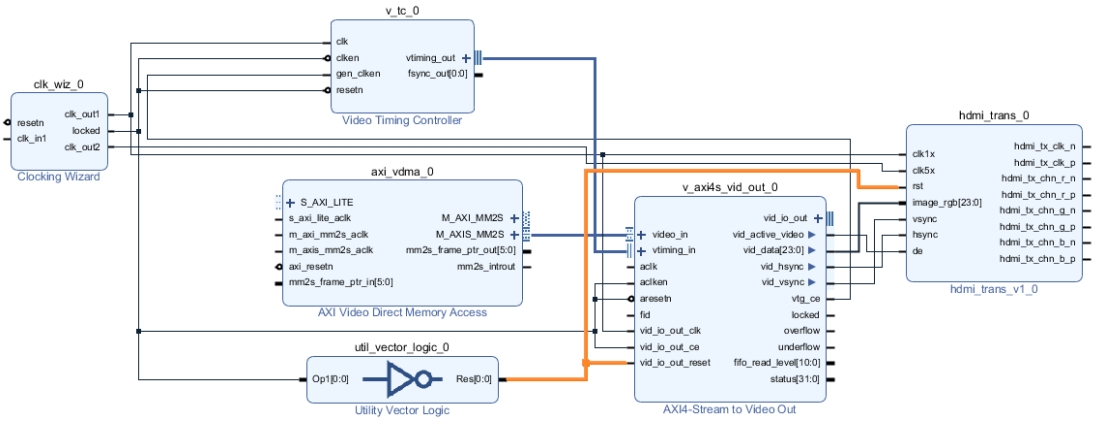


图 9- 17 高电平复位连接

9.2.2 ZYNQ 处理器

接下来添加ZYNQ处理器，并且对其进行配置，配置方法与前面的实验类似。

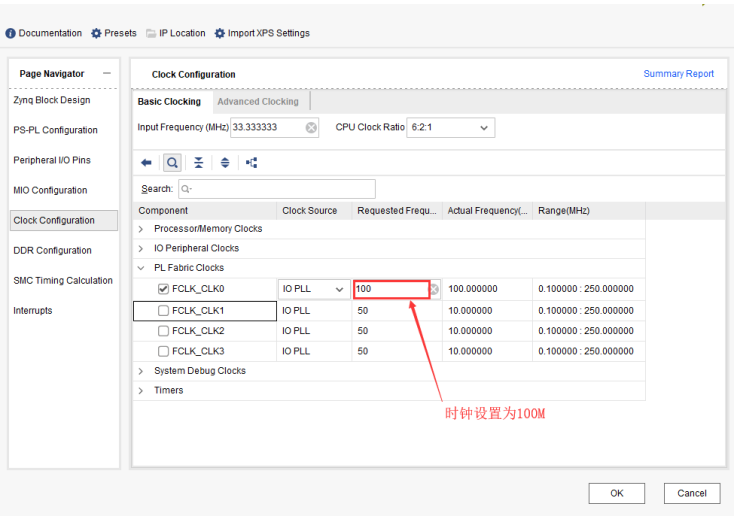


图 9- 18

然后将时钟给到 PLL，用于产生像素时钟。

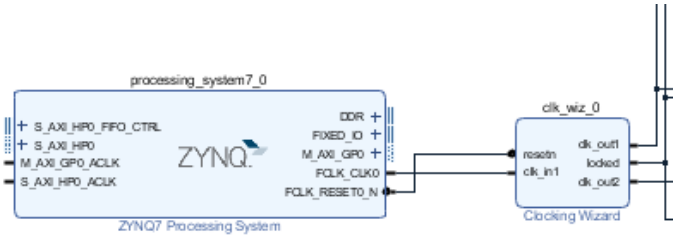


图 9- 19

9.2.3 HLS IP 部分

接下来添加 HLS 生成的二维卷积运算 IP，该 IP 也已经添加到 ip_repo 路径下，直接搜索即可使用该 IP。

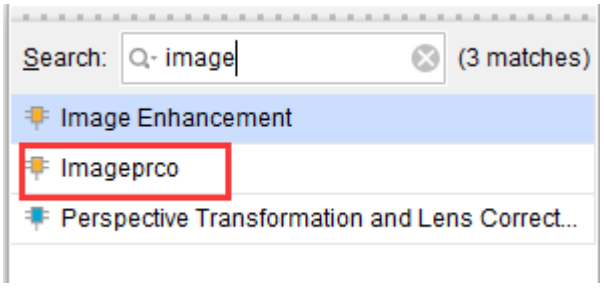


图 9- 20

对于该 IP 的使用与，前面章节的使用类似，添加 DMA 与其相连接。

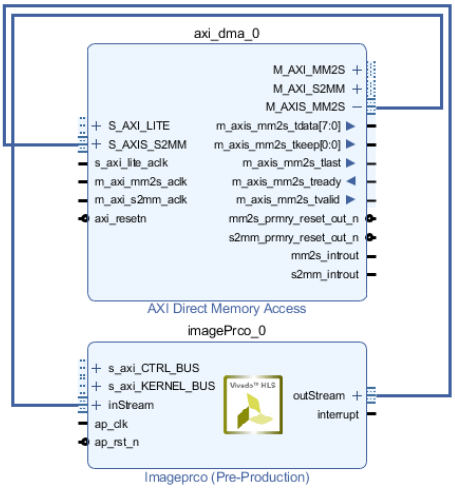


图 9- 21

9.2.4 连线创建 Block Design

以上 IP 全部添加完毕后，点击自动连线生成本次实验最终的 Block Design。连接完成后的 Block Design 如图所示：

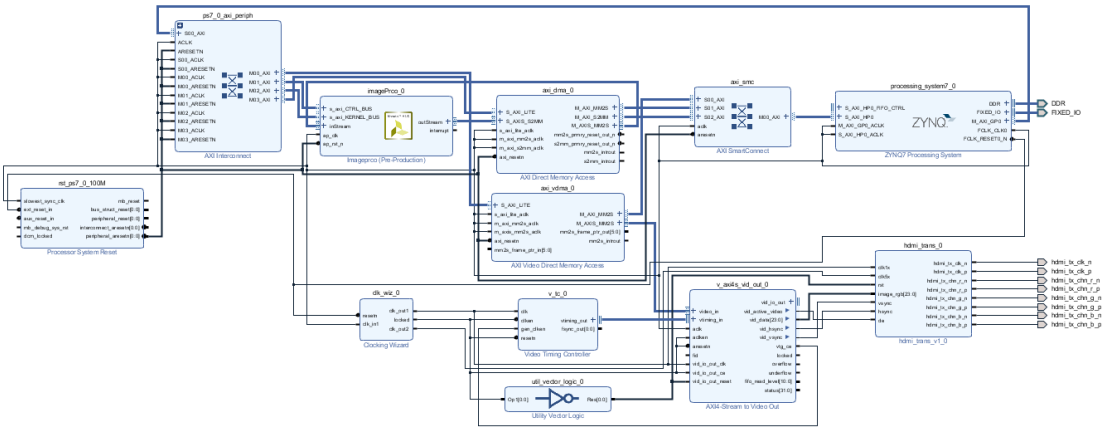


图 9- 22

添加约束文件，在 constrian 栏中添加本次实验的引脚约束：

```
set_property PACKAGE_PIN K17 [get_ports hdmi_tx_clk_p]
set_property PACKAGE_PIN G19 [get_ports {hdmi_tx_chn_r_p}]
set_property PACKAGE_PIN F19 [get_ports {hdmi_tx_chn_g_p}]
set_property PACKAGE_PIN D19 [get_ports {hdmi_tx_chn_b_p}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {hdmi_oen[0]}]
set_property PACKAGE_PIN M20 [get_ports {hdmi_oen[0]}]
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]
```

然后创建 wrapper，生成 bit 文件。

9.3 创建 SDK 工程

生成完 bit 文件后就可以在 SDK 中进行开发，本次实验完成通过按键输入 0、1、2、3 来选择要对图像进行的操作，并且通过显示器能够直观地在显示器上观察出对图像进行不同卷积操作后地的变化。

SDK 的代码在这里不做详细讲解。只介绍其中比较重要的几个代码的含义。

首先，对于 VDMA 的初始化和配置，是调用了 Xilinx 官方的 api 函数来完成的，官方的 API 函数，在我们的工程中也已经提供。在本次实验中，只需调用其中的一个 API 函数，即可完成对 VDMA 的配置。

```
run_vdma_frame_buffer(&Vdma,VDMA_DEV_ID,1280,720,VDMA_MM2S_ADDR,0,
0,ONLY_READ);
```

使用该函数，将 VDMA 设置图像缓冲区中缓存图像的大小为 1280*720，设置为 720P 格式。

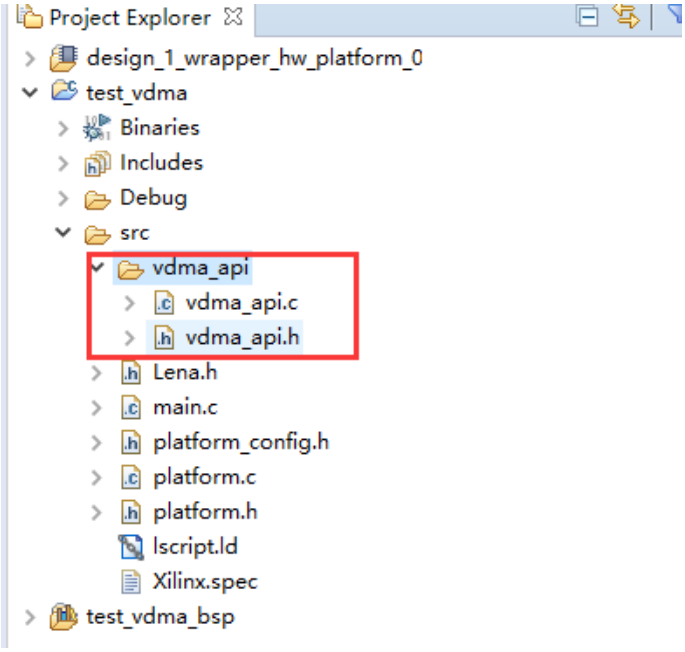


图 9- 23 VDMA API 函数

下面的表格中是不同的 kernel，这些 kernel 的作用不相同，其中 Erode_or_dilatekernel 是用于对图像的腐蚀和膨胀，impulse Kernel 只对图像进行输出，不做任何改变，目的是为了能够和进行操作后的图像进行对比。Sobel_kernel 用于进行 sobel 边缘检测，edge_kernel 也是一种边缘检测的算子。这些 kernel 通过与键盘输入的 operation 配合，完成最终的图像的显示。

```
char Erode_or_Dilate_kernel[9] = {
        1, 1, 1,
        1, 1, 1,
        1, 1, 1,
};

//impulse kernel (只将图像输入，不做任何改变)
char impulse_kernel[9] = {
        0, 0, 0,
        0, 1, 0,
        0, 0, 0,
};

//kernel for sobel
char sobel_kernel[9] = {
        -1, -2, -1,
```

```
            0,   0,   0,
            1,   2,   1,
};

//kernel for edge
char edge_kernel[9] = {
            -1, -1, -1,
            -1,   8, -1,
            -1, -1, -1,
};
```

点击 debug 后进入调试界面，先连接好串口，方便观察打印信息。在调试界面点击运行。

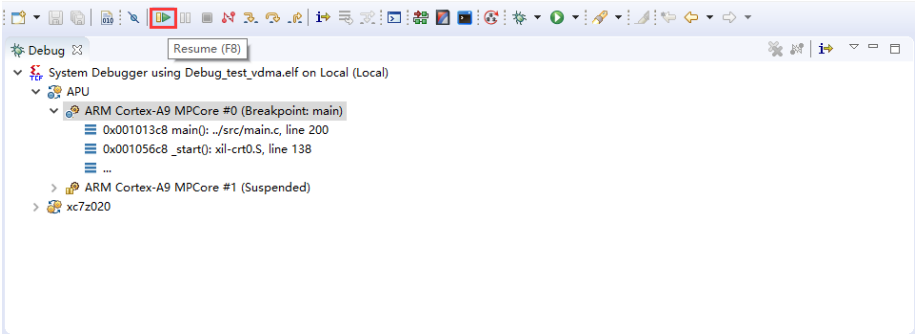


图 9- 24

等到串口输出提示信息后，先连接好 HDMI 线。

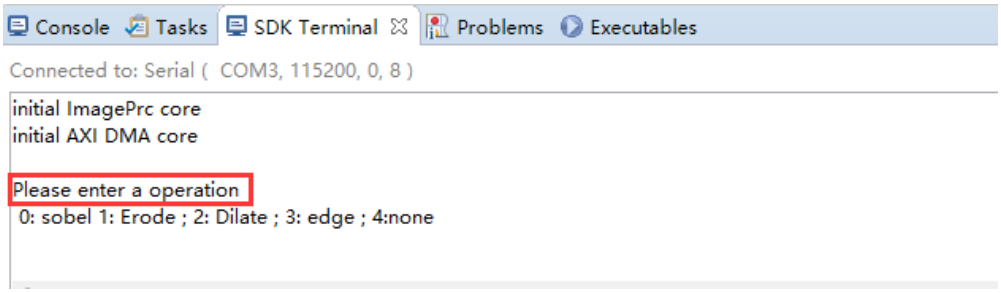


图 9- 25

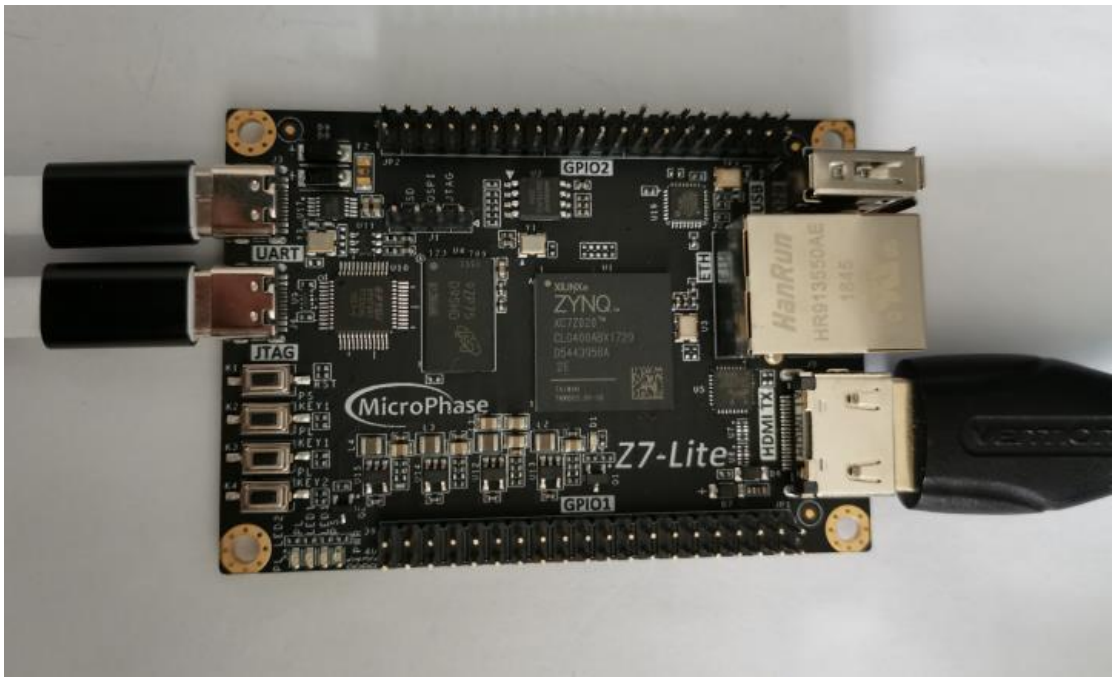


图 9- 26

此时在串口输入 0~4 可以观察到对图像进行的不同的处理。例如，此时输入 4，即不对图像做任何处理，显示在屏幕上的就是原图像。

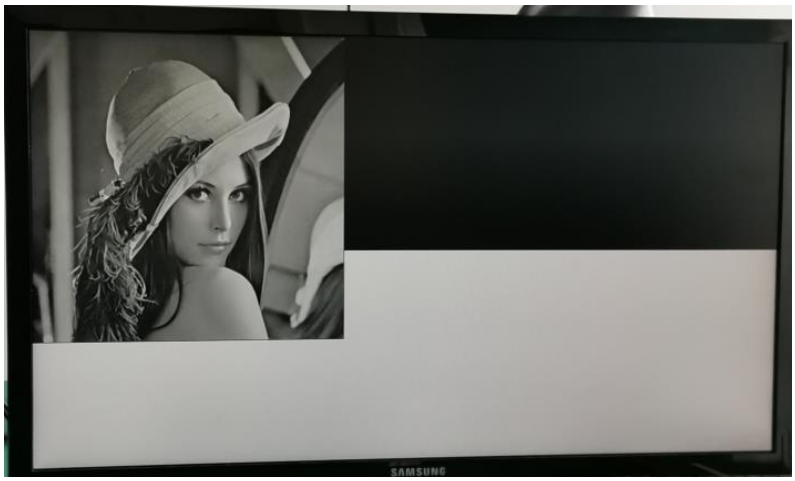


图 9- 27 在屏幕上显示原图像

此时输入 1，即可观察到腐蚀后的图像。

图 9- 28 腐蚀后的图像

此时输入 2，即可观察膨胀后的图像。

图 9- 29 膨胀后的图像

在文档中，不一定能够看出较明显的差距，若在实际中，快速切换腐蚀和膨胀，就能看到明显的差距。

此时输入 0，即可看到 sobel 边缘检测后的图像。

图 9- 30 sobel 边缘检测图像

此时输入 3 可以的到另一种边缘检测后的图像

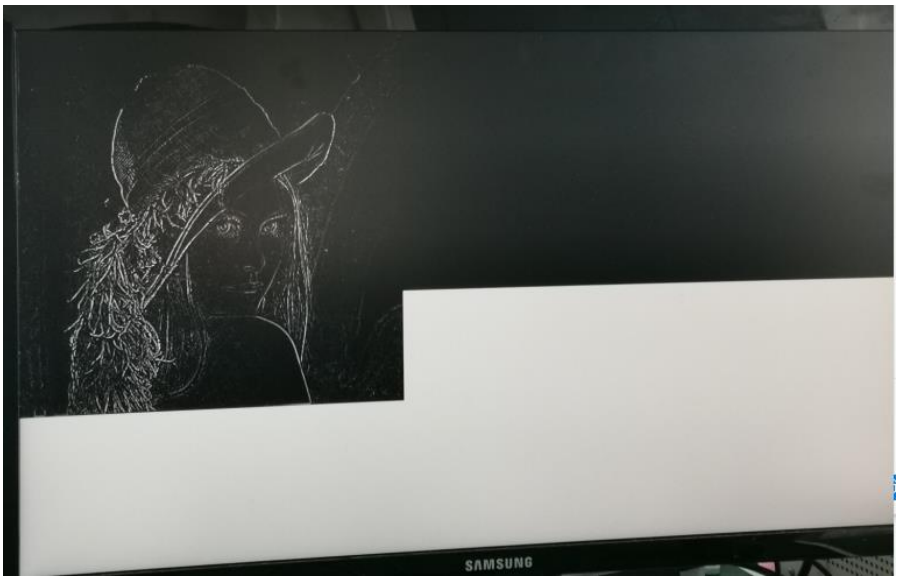


图 9- 31 边缘检测后的图像

自此，就完成了，本次实验的全部展示，在 SDK 中，通过不同的卷积核(Kernel)完成了对图像的不同的处理方法。通过调用 HLS 生成的 IP 核来完成卷积运算的效率相较于传统的方法，还是有很大的提升的。

9.4 实验总结

在本次实验中，通过调用上次实验中由 HLS 生成的 IP，在 Vivado 中搭建了 Block Design，使用 VDMA 来进行管理输出的图像，能够简化对 DDR 的操作。在 SDK 中，通过对按键输入对应的按键来选择对图像进行不同的处理，完成了图像简单的腐蚀、膨胀和边缘检测功能。

本次实验，需要注意的就是在 Vivado 搭建 block Design 是需要注意各个模块之间的连接，其中有些信号还是比较容易因为疏忽而漏掉。

第10章　OV5640 的 sobel 边缘检测

10.1 实验简介

在上一章的实验当中，介绍了如何使用 HLS 进行二维卷积，并且最终完成了在显示器上显示经过二维卷积处理过后的图像。在本章实验当中，将会使用微相科技的扩展子卡 OV5640 双目摄像头上的其中一个摄像头，在上一章实验的基础上完成对摄像头采集到的图像数据进行边缘检测。

10.2 OV5640 硬件介绍

OV5640 双目摄像头子卡如下图所示：

图 10- 1

OV5640 是一款 1/4 英寸单芯片图像传感器，其感光阵列达到 2592*1944（即 500W 像素），能实现最快 15fps QSXVGA（2592*1944）或者 90fps VGA（640*480）分辨率的图像采集。传感器采用 OmniVision 推出的 OmniBSI（背面照度）技术，使传感器达到更高的性能，如高灵敏度、低串扰和低噪声。传感器内部集成了图

像处理的功能，包括自动曝光控制（AEC）、自动白平衡（AWB）等。同时该传感器支持 LED 补光、MIPI（移动产业处理器接口）输出接口和 DVP（数字视频并行）输出接口选择、ISP（图像信号处理）以及 AFC（自动聚焦控制）等功能。

10.2.1 OV5640 整体架构

通过查阅 OV5640 的 datasheet 可以看到 OV5640 的整体架构，入下图所示：

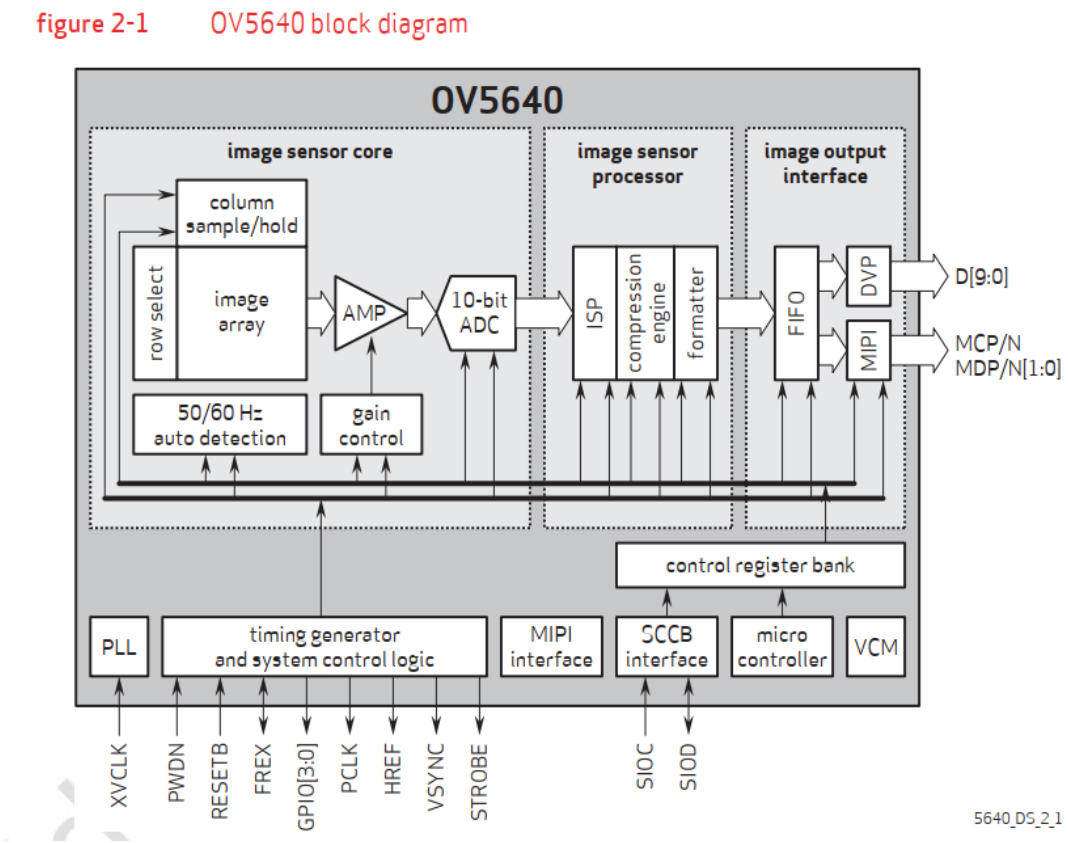


图 10- 2

阅读 datasheet 可以知道 OV5640 的大致工作流程是，在图像采集传感器 image sensor core 中，通过曝光和采样，在图像阵列中得到原始模拟图像数据，经过 AMP 对图形进行放大、校正，然后将校正后的图像通过 10 位的 ADC 芯片转换位数字信号，经过图像单元 image sensor processer 能够得到图像的数字信号，缓存到图像输出接口中，最终通过 DVP 或者 MIPI 接口将 10bit 数据流输出。

时序发生器（timing generator and system control logic）控制着感光阵列，AMP

和 ADC，并且通过该模块产生图像的时序信号 VSYNC、HREF、PCLK，外部时钟通过 XVCLK 经过 PLL 进入系统作为系统的工作时钟。

对于如何确定其中各个模块的工作状态和输出图像的格式和输出时序都是通过配置 OV5640 的寄存器实现的。配置 OV5640 的寄存器是通过 SCCB 接口来实现的。

10.2.2 SCCB 简介

SCCB（serial camera control bus，串行摄像头控制总线），是 OmniVision 公司定义和发展的三线式串行总线，通过该总线能够完成摄像头内部的寄存的配置，设计摄像头的功能，包括图像数据格式，分辨率和图像处理的参数。OV 公司为了减少传感器引脚的封装，现在 SCCB 总线大多采用两线式接口总线。

SCCB 从形式上与 I2C 协议十分类似，所以在本章中我们只是简单介绍一下 SCCB 协议。SCCB 主要由两条线 SCL 和 SDA，通过阔这着两条信号线的时序，完成数据的传输。

1) SCCB 写传输时序：

图 10- 3

与 I2C 协议传输类似，SCCB 在传输数据时，首先也要产生一个起始位，然后给出 7 位器件地址和要写操作，等待第一个八位数据传输完毕后会出现一个不用关心位，这里与 I2C 协议有区别，在 I2C 传输协议中，一字节数据发送完毕后会有一字节的响应位，而在 SCCB 协议中是不用关心的。

再然后传输一字节的寄存器地址，同样的传输完成后有一个不用关心位，在之后是传输一字节的写入寄存器的数据，然后是不用关心位，最后产生一个停止位结束本次传输。

可以看到 SCCB 传输数据与 I2C 是非常类似的，有了前面的 I2C 协议的基础，

这里的 SCCB 协议就不难理解。

2) SCCB 读传输时序

SCCB 读传输时序要经过两个阶段，这与 I2C 协议也是类似的，但是需要注意的是，SCCB 读时序中，第一个部分是写器件地址和写寄存器地址，然后要产生一个停止位。在第二个阶段又要产生一个起始位，然后是读寄存器和读出的数据，最后是停止位。SCCB 的读时序和 I2C 读时序有较大的区别，SCCB 传输协议中 START 信号和 STOP 总是成对出现的。而在 I2C 协议中，是不用的。

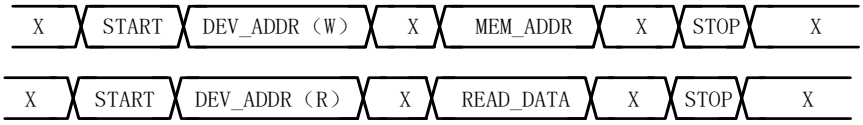


图 10- 4

除此之外，还需要注意的地方是，在 SCCB 传输协议中，是不能连续读写，也就是说，SCCB 一次只能对一个寄存器进行读写，而不能连续读写。本次实验还需要注意的事项是，本次实验使用的器件 OV5640 的急促请你地址是 16 位地址，在传输寄存器地址时要分为两个字节进行传输，第一个字节传输寄存器地址高八位，第二字节传输寄存器地址低八位。

10.2.3 配置寄存器

确定 OV5640 的工作模式，我们需要对 OV5640 的寄存器进行配置。具体的配置可以参照 OV5640 的用户手册和软件手册，我们在工程目录下的 src\doc 目录提供有有关 OV5640 的相关手册。

这里简单介绍一下其中比较重要的寄存器配置。

图像窗口配置：

图像窗口用来配置输出图像的大小和缩放比例。通过配置这几个寄存器能够改变窗口的起始位置，窗口的大小和窗口的相对于起始位置的偏移量，和缩放比，

设置这些参数能够完成输出图像的大小。例如在本次实验中，我们将设置窗口的有效大小 1024*768，就可以通过下面的这些寄存器来配置。

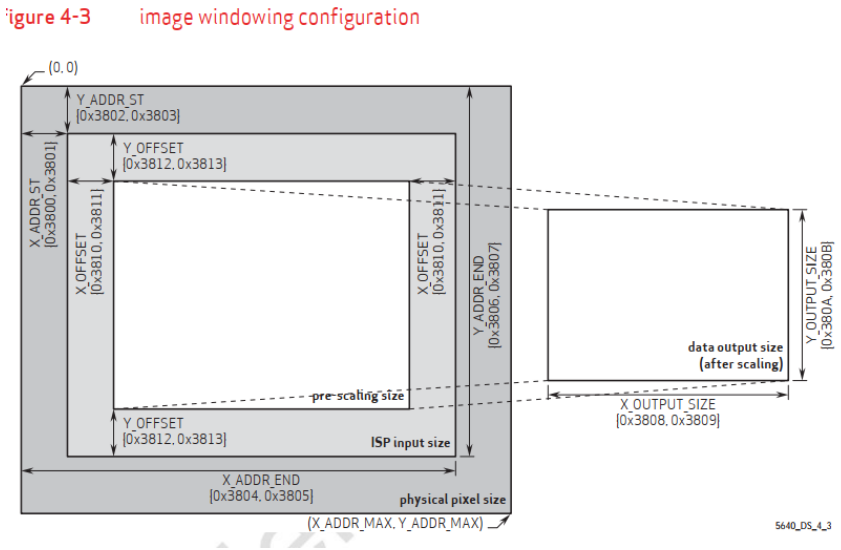


图 10- 5

table 4-2 image windowing registers (sheet 1 of 2)

address	register name	default value	R/W	description
0x3800	TIMING HS	0x00	RW	Bit[3:0]: X address start high byte[11:8] high byte
0x3801	TIMING HS	0x00	RW	Bit[7:0]: X address start low byte[7:0] low byte
0x3802	TIMING VS	0x00	RW	Bit[2:0]: Y address start high byte[10:8] high byte
0x3803	TIMING VS	0x00	RW	Bit[7:0]: Y address start low byte[7:0] low byte
0x3804	TIMING HW	0x0A	RW	Bit[3:0]: X address end high byte[11:8] high byte
0x3805	TIMING HW	0x3F	RW	Bit[7:0]: X address end low byte[7:0] low byte
0x3806	TIMING VH	0x07	RW	Bit[2:0]: Y address end high byte[10:8] high byte
0x3807	TIMING VH	0x9F	RW	Bit[7:0]: Y address end low byte[7:0] low byte
0x3808	TIMING DVPHO	0x0A	RW	Bit[3:0]: DVP output horizontal width[11:8] high byte

图 10- 6

table 4-2 image windowing registers (sheet 2 of 2)

address	register name	default value	R/W	description
0x3809	TIMING DVPHO	0x20	RW	Bit[7:0]: DVP output horizontal width[7:0] low byte
0x380A	TIMING DVPVO	0x07	RW	Bit[2:0]: DVP output vertical height[10:8] high byte
0x380B	TIMING DVPVO	0x98	RW	Bit[7:0]: DVP output vertical height[7:0] low byte
0x380C	TIMING HTS	0x0B	RW	Bit[3:0]: Total horizontal size[11:8] high byte
0x380D	TIMING HTS	0x1C	RW	Bit[7:0]: Total horizontal size[7:0] low byte
0x380E	TIMING VTS	0x07	RW	Bit[7:0]: Total vertical size[15:8] high byte
0x380F	TIMING VTS	0xB0	RW	Bit[7:0]: Total vertical size[7:0] low byte
0x3810	TIMING HOFFSET	0x00	RW	Bit[3:0]: ISP horizontal offset[11:8] high byte
0x3811	TIMING_HOFFSET	0x10	RW	Bit[7:0]: ISP horizontal offset[7:0] low byte
0x3812	TIMING VOFFSET	0x00	RW	Bit[2:0]: ISP vertical offset[10:8] high byte
0x3813	TIMING VOFFSET	0x04	RW	Bit[7:0]: ISP vertical offset[7:0] low byte

图 10- 7

1) 输出图像格式配置：

图像的输出格式，具体要去配置 0x4300 这个寄存器，在本实验中我们通过配置这个寄存器，最终让摄像头输出 rgb565 格式的数据。只需将 0x4300 这个寄存器的 bit[7:4]设置成 0x6 即可。当设置 0x4300 的输出格式位 rgb565 后，进行图像数据输出顺序，在这里我们设置 0x4300 的第四位 bit[3:0]为 0x1，这样就完成了输出的顺序时按照红绿蓝的顺序输出。

Format control converts the internal data format into the desired output format including YUV, RGB, or RAW.

table 6-5 FORMAT control registers (sheet 1 of 5)

address	register name	default value	R/W	description
0x4300	FORMAT CONTROL 00	0xF8	RW	Format Control 00 Bit[7:4]: Output format of formatter module 0x0: RAW Bit[3:0]: Output sequence 0x0: BGBG... / GRGR... 0x1: GBGB... / RGRG... 0x2: GRGR... / BGBG... 0x3: RGRG... / GBGB... 0x4~0xF: Not allowed 0x1: Y8 Bit[3:0]: Does not matter 0x2: YUV444/RGB888 (not available for full resolution) Bit[3:0]: Output sequence 0x0: YUVYUV..., or GBRGBR... 0x1: YVUYVU..., or GRBGRB... 0x2: UYVUYV..., or BGRBGR... 0x3: VYUVYU..., or RGBRGB... 0x4: UVYUVY..., or BRGBRG... 0x5: VUYVUY..., or RBGRBG... 0x6~0xE: Not allowed 0xF: UYVUYV..., or BGRBGR... 0x3: YUV422 Bit[3:0]: Output sequence 0x0: YUYV... 0x1: YVYU... 0x2: UYVY... 0x3: VYUY... 0x4~0xE: Not allowed 0xF: UYVY... 0x4: YUV420 Bit[3:0]: Output sequence 0x0: YYYY... / YUYV... 0x1: YYYY... / YVYU... 0x2: YYYY... / UYVY... 0x3: YYYY... / VYUY... 0x4: YUYV... / YYYY...

idential for
usheng

table 6-5 FORMAT control registers (sheet 2 of 5)

address	register name	default value	R/W	description
				0x5: YVYU... / YYYY... 0x6: UYVY... / YYYY... 0x7: VYUY... / YYYY... 0x8~0xE: Not allowed 0xF: YYYY... / UYVY... 0x5: YUV420 (for MIPI only) Bit[3:0]: Output sequence 0x0~0xD: Not allowed 0xE: VYYVYY... / UYYUYY... 0xF: UYYUYY... / VYYVYY... 0x6: RGB565 Bit[3:0]: Output sequence 0x0: {b[4:0],g[5:3]}, {g[2:0],r[4:0]} 0x1: {r[4:0],g[5:3]}, {g[2:0],b[4:0]} 0x2: {g[4:0],r[5:3]}, {r[2:0],b[4:0]} 0x3: {b[4:0],r[5:3]}, {r[2:0],g[4:0]} 0x4: {g[4:0],b[5:3]}, {b[2:0],r[4:0]} 0x5: {r[4:0],b[5:3]}, {b[2:0],g[4:0]} 0x6~0xE: Not allowed 0xF: {g[2:0],b[4:0]}, {r[4:0],g[5:3]} 0x7: RGB555 format 1 Bit[3:0]: Output sequence 0x0: {b[4:0],g[4:2]}, {g[1:0],1'b0,r[4:0]} 0x1: {r[4:0],g[4:2]}, {g[1:0],1'b0,b[4:0]} 0x2: {g[4:0],r[4:2]}, {r[1:0],1'b0,b[4:0]} 0x3: {b[4:0],r[4:2]}, {r[1:0],1'b0,g[4:0]} 0x4: {r[4:0],b[4:2]}, {b[1:0],1'b0,g[4:0]} 0x5: {g[4:0],b[4:2]}, {b[1:0],1'b0,r[4:0]} 0x6~0xE: Not allowed 0xF: {g[1:0],1'b0,b[4:0]}, {r[4:0],g[4:2]} 0x8: RGB555 format 2 Bit[3:0]: Output sequence 0x0: {1'b0,b[4:0],g[4:3]}, {g[2:0],r[4:0]}		

Confidential for Ruisipusheng

在这里要对这个输出格式做一个简单的说明，输出的图像是 rgb565 格式的数据，为两字节，但是在输出数据的时候，我们的 DVP 接口的输出数据位宽为 10 位即 D[9:0],为了能够正确传输数据，一个像素的数据将分为两个时钟周期发送，只去 D[9:2]作为数据的有效位，按照前面 0x4300 给出的配置，可以看到，在

第一个时钟周期输出的 D[9:2]={r[4:0],g[5:3]}；输出的 r 分量和 g 分量的高 3 位。

第二个时钟周期输出的 D[9:2]={g[2:0],b[4:0]};输出的 g 分量的低三位和 b 分量。这样就能将图像正确的传输了。

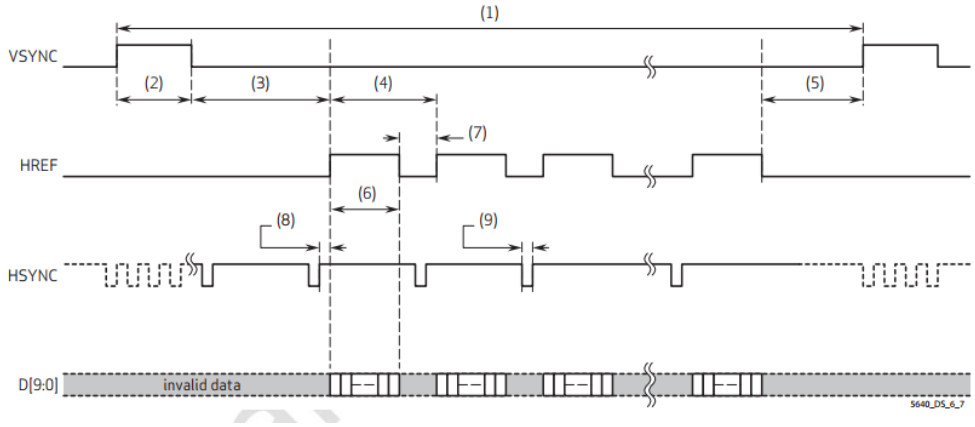
对于其他的寄存器配置，用户可以参考文档中的 OV5640_CSP3_DS_2.01 这个文档中的寄存器说明进行调整。

10.2.4 图像输出时序：

通过查阅用户手册 OV5640_CSP3_DS_2.01 可以看到输出图像的时序，如下图所示：

6.6.2 DVP timing

figure 6-7 DVP timing diagram



接下来简单介绍一下这个时序图，以 1024*768 分辨率的图像为例，下图是该分辨率下的时序参数：

XGA 1024x768	(1) 5596992 tp (2) 5688tp (3) 2825588 tp (4) 2844 tp (5) 583344tp (6) 1024 tp (7) 1820 tp (8) 0 tp (9) 1820 tp

对于 DVP 接口的输出时序模式，一般由两种模式可以选择，一种是 VSYNC 和 HSYNC 模式，这种模式与我们熟悉的 VGA 模式是类似的；另外一种模式是 VSYNC 和 HREF 模式，这种模式下 HREF 信号为高的时候代表数据有效，在使用 OV5640 采集图像的时候我们一般采用这种方式来输出图像，这样能够在用户接收图像数据的时候能够提供便利，当 HREF 信号有效的时候，就表示当前的数据有效。

在本时序图中值得注意的是，时序图中的时序参数 T p 并不一定是 OV5640 的系统时钟。这个要根据我们输出的像素格式来确定。例如输出 rgb565 格式的

数据是 tp 指的是像素 时钟，这个时钟是输出一个像素点的时钟，在前面讲到过，rgb565 格式下输出一个像素需要两个系统时钟，所以这路的 T p=2* T sys。

在本次实验中，有关与 OV5640 的简单介绍就到这里了，用户若想深入了解 OV5640 的工作原理，可以去查看用户手册和其对应的软硬件手册。

在本次实验中，我们会提供一个已经封装好的摄像头采集 IP，采集到的摄像头视频输出格式为 1024x768。

10.3 HLS 中完成二维卷积的设计

本次实验相较于上次实验在 HLS 部分需要做的改动很少，源代码如下：

```
#include "kernel.h"

void doKernel(AXI_STREAM & inStream,AXI_STREAM & outStream, char kernel[KERNEL_DIM*KERNEL_DIM], int operation)
{
#pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=operation bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=kernel bundle=KERNEL_BUS
#pragma HLS INTERFACE axis register both port=outStream
#pragma HLS INTERFACE axis register both port=inStream

 BUFFER lineBuf;
 SHORT_WINDOW window;

 //用来标记行和列的索引值
 int pixConvolved = 0;

 //计算延时，填充 line_buffer
 int waitPixel = IMG_WIDTH + 1;
 int cntWait = 0;
 int sentPixel = 0;
 unsigned char pix;

 uint_side_channel dataOutChannel;
```

```
uint_side_channel currentChannel;


for(int i = 0; i < (IMG_HEIGHT); i++ )
{
    for(int j = 0; j < (IMG_WIDTH); j++)
    {
#pragma HLS PIPELINE
        currentChannel = inStream.read();
        unsigned char R = currentChannel.data & 0xFF;
        unsigned char G = (currentChannel.data >> 8) & 0xFF;
        unsigned char B = (currentChannel.data >> 16) & 0xFF;
        pix = (R*76 + G*150 + B*30) >> 8;
        lineBuf.shift_pixels_up(j);
        lineBuf.insert_bottom_row(pix,j);
        //put data on the window, and multiply by kernel
        for(int idxWinRow = 0; idxWinRow < KERNEL_DIM; idxWinRow++)
        {
            for(int idxWinCol=0; idxWinCol < KERNEL_DIM; idxWinCol++)
            {
                //idxWinCol and pixConvolved will slide the window
                short val = (short)lineBuf.getval(idxWinRow, idxWinCol + pixConvolved);
                //采样窗口中的数据与 Kernel(卷积算子)相乘
                val = (short)kernel[(idxWinRow * KERNEL_DIM) + idxWinCol] * val;
                window.insert(val,idxWinRow,idxWinCol);
            }
        }
        short valOut;
        if((i >= KERNEL_DIM-1) && (j >= KERNEL_DIM -1))
        {
            valOut = sumWindow(&window);
            if(valOut < 0 )
            {
                valOut = 0;
            }
            pixConvolved++;
            dataOutChannel.data = (valOut << 16) | (valOut << 8) | valOut;
            dataOutChannel.dest = currentChannel.dest;
            dataOutChannel.id = currentChannel.id;
            dataOutChannel.keep = currentChannel.keep;
```

```
                dataOutChannel.strb = currentChannel.strb;
                dataOutChannel.user = currentChannel.user;
                dataOutChannel.last = currentChannel.last;
                outStream.write(dataOutChannel);
                sentPixel++;

            }
            else
            {
                dataOutChannel.data = 0;
                dataOutChannel.dest = currentChannel.dest;
                dataOutChannel.id = currentChannel.id;
                dataOutChannel.keep = currentChannel.keep;
                dataOutChannel.strb = currentChannel.strb;
                dataOutChannel.user = currentChannel.user;
                dataOutChannel.last = currentChannel.last;
                outStream.write(dataOutChannel);
                sentPixel++;
            }
        }
        pixConvolved = 0;
    }
}

//求卷积和
short sumWindow(SHORT_WINDOW *window){
    short sum=0;
    for(int idxRow = 0; idxRow < KERNEL_DIM; idxRow++){
        for(int idxCol = 0;idxCol < KERNEL_DIM;idxCol++){
            sum += (short)window->getval(idxRow, idxCol);
        }
    }
    return sum;

}
```

完成的工作是将输入的 rgb 图像转换为灰度图像，再将灰度图像进行二维卷积的结果输出，以此得到边缘检测后的图像。

C 仿真文件如下的 main 函数如下。

```
1  int main(){
2    IplImage * SrcImage;//输入的图像
3    IplImage * DstImage;//输出的图像
4    IplImage * GoldImage;//参考图像
5    IplImage * GoldImageTmp;//参考图像
6    SrcImage = cvLoadImage(INPUT_IMAGE,-1);//导入图像
7
8    //创建 opencv 支持的输出的图像
9    DstImage = cvCreateImage(cvGetSize(SrcImage),SrcImage->depth, SrcImage->nChannels);
10   GoldImage = cvCreateImage(cvGetSize(SrcImage),SrcImage->depth,
SrcImage->nChannels);
11   GoldImageTmp = cvCreateImage(cvGetSize(SrcImage),SrcImage->depth, 1);
12   //AXI-Stream 数据流
13   AXI_STREAM inStream;
14   AXI_STREAM outStream;
15   IplImage2AXIvideo(SrcImage,inStream);
16
17   //可综合的函数
18   doKernel(inStream,outStream,kernel,0);
19   //保存输出结果
20   AXIvideo2IplImage(outStream,DstImage);
21   cvSaveImage(OUTPUT_IMAGE,DstImage);
22
23   //参考图像输出
24   cvCvtColor(SrcImage, GoldImageTmp, CV_BGR2GRAY);
25   cvCvtColor(GoldImageTmp, GoldImage, CV_GRAY2BGR);
26   cvReleaseImage(&GoldImageTmp);
27   cvSaveImage(GOLD_IMAGE, GoldImage);
28   cvShowImage(OUTPUT_IMAGE, DstImage);
29   cvWaitKey(0);
30
31   //比较黄金参考图片和 HLS 可综合执行结果
32   image_compare(OUTPUT_IMAGE,GOLD_IMAGE);
33   //释放图片
34   cvReleaseImage(&SrcImage);
35   cvReleaseImage(&DstImage);
36   cvReleaseImage(&GoldImage);
37   return 0;
38
39
40 }
```

再 C 仿真的时候，用 impulse kernel 进行测试，得到的输出图像如下图所示：

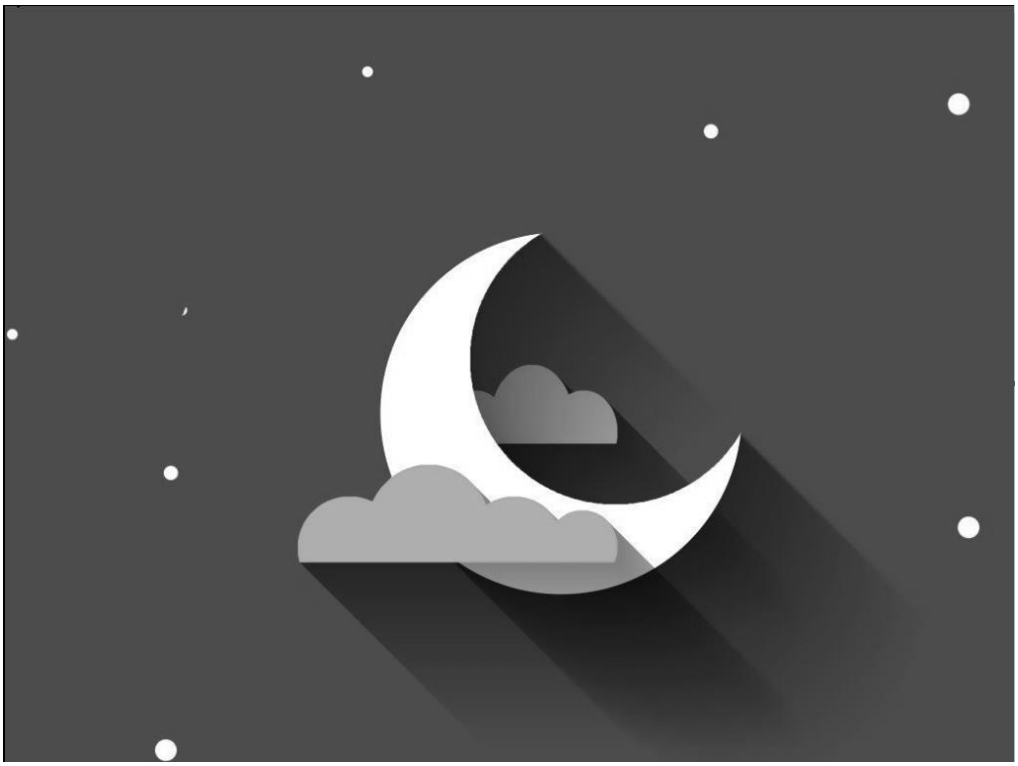


图 10- 8

可以看到，因为再源文件中使用到了 Linebuffer 以此整个图像有所偏移。在之后即可以进行 C 综合和 C 仿真，仿真无误后，即可导出 IP 再 Vivado 中调用。

10.4 搭建 Block Design

生成 HLS IP 后，首先将 IP 添加进 Vivado IP 仓库中。本次实验中，提供有一个 OV5640 采集的 IP，该 IP 是微相在其他开发中所使用到的一个 IP，这里直接将其封装好。该 IP 设置 ov5640 的输出图像是 1024x768@60Hz 的格式的。在前面的 HLS 中，生成的 IP 也是用于处理大小为 1024x768 的。本次实验，以 1024x768 为参考。

在用户的 IP 仓库中，可以看到本次实验 HLS 生成的和用户自定义的 IP。

图 10- 9

接下来搭建本次实验的 Block Design

10.4.1 摄像头采集部分

1) 添加用户自定义摄像头采集 IP

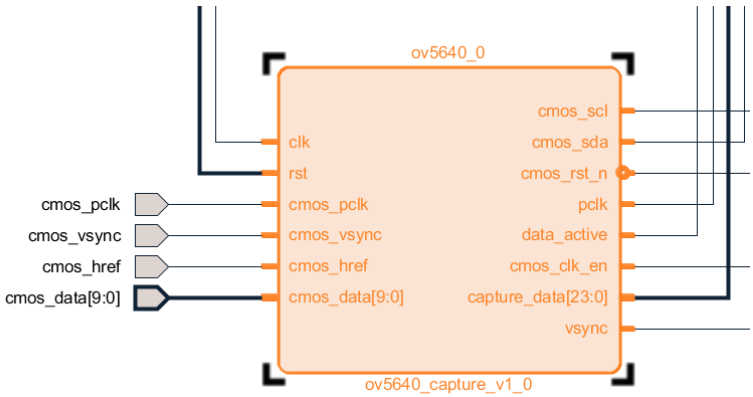


图 10- 10

2) 添加用于视频输入的 VDMA

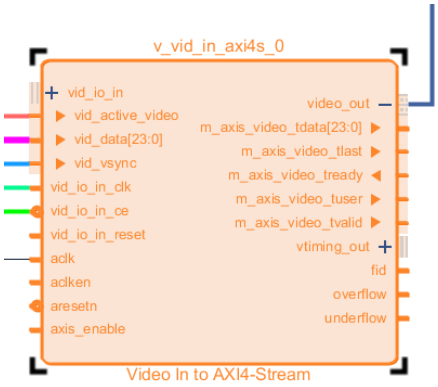


图 10- 11

对视频输入 VDMA 进行如下配置：使能读写通道，同时设置读通道的数据流位宽为 24 位。

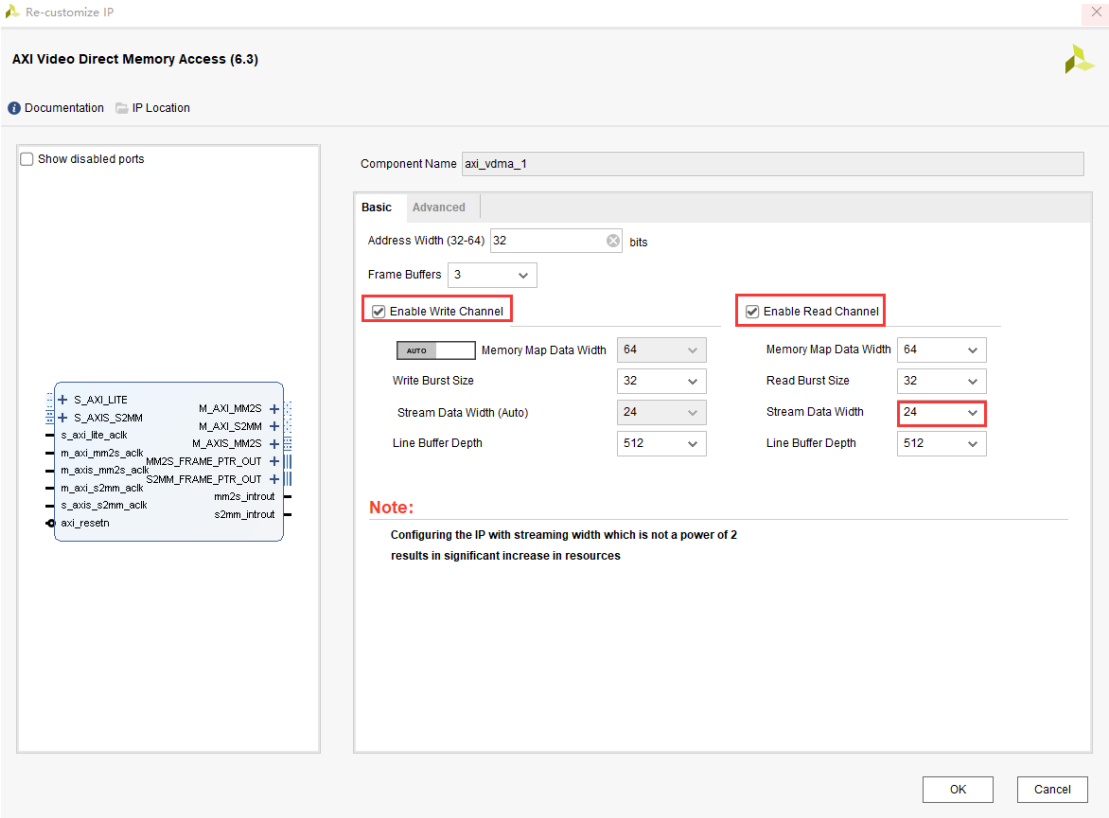


图 10- 12

将视频输入 VDMA 和摄像头采集模块按照如下方式连接：

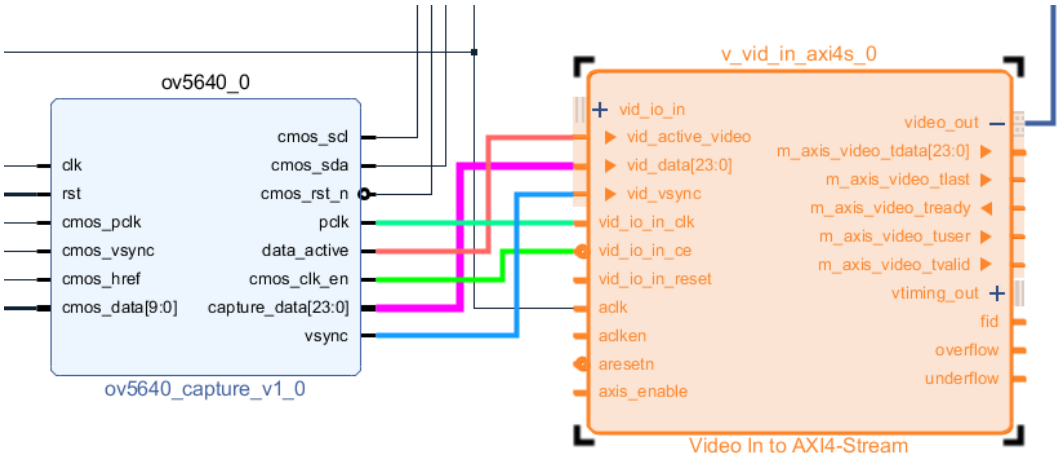


图 10- 13

在摄像头采集模块，pclk 是像素时钟，data_active 是数据有效信号，

cmos_clk_en 是时钟使能信号，capture_data 是采集到的图像数据。Vsynac 是场同步信号。连接好后，需要对摄像头采集 IP 引出其余引脚，用于采集数据和配置 OV5640。

10.4.2 视频输出

对于视频输出通路，与上次实验的搭建方式类似，用户只需在添加视频输出的 VDMA 按照和输入的 VDMA 同样配置即可。

10.4.3 添加图像处理 IP

接下来添加由 HLS 生成的用于图像处理的 IP。并将该 IP 与两个 VDMA 按照如下方式连接。

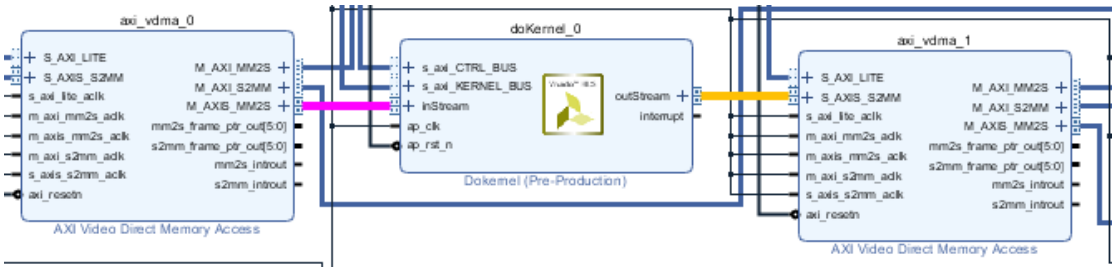


图 10- 14

接下来点击自动连接完成整个 block design 的搭建即可。

10.5 创建 SDK 工程

生成完 bit 文件后，就可以导出 bit 文件，在 SDK 中进行开发了。

```
1  #include <stdio.h>
2  #include "platform.h"
3  #include "xil_printf.h"
4  #include "vdma_api/vdma_api.h"
5  #include "xdokernel.h"
6  #include "xparameters.h"
7  #include "xil_cache.h"
8
```

```
9
10  #define DISPLAY_VDMA_DEV_ID   XPAR_AXI_VDMA_1_DEVICE_ID
11  #define HLS_VDMA_DEV_ID       XPAR_AXI_VDMA_0_DEVICE_ID
12  #define KERNEL_DEV_ID         XPAR_DOKERNEL_0_DEVICE_ID
13
14  #define DISP_BASE_ADDR        0x08000000
15  #define HLS_BASE_ADDR         0x03000000
16  #define SCREEN_X              1024
17  #define SCREEN_Y              768
18
19
20  static XAxiVdma               Vdma;
21  static XDokernel              doKernel    ;
22  static XDokernel_Config       *doKernel_Cfg;
23
24
25  //impulse kernel (只将图像输入，不做任何改变)
26  char impulse_kernel[9] = {
27          0, 0, 0,
28          0, 1, 0,
29          0, 0, 0,
30  };
31
32  //kernel for sobel
33  char sobel_kernel[9] = {
34          -1, -2, -1,
35            0,   0,   0,
36            1,   2,   1,
37  };
38
39  //kernel for edge
40  char edge_kernel[9] = {
41          -1, -1, -1,
42          -1,   8, -1,
43          -1, -1, -1,
44  };
45
46
47  int initdokernel(){
48   int status;
49   doKernel_Cfg = XDokernel_LookupConfig(KERNEL_DEV_ID);
```

```
50  status = XDokernel_CfgInitialize(&doKernel,doKernel_Cfg);
51  if(status != XST_SUCCESS){
52          printf("initialize failed! \n");
53          return status;
54  }
55
56
57  return status;
58 }
59
60
61
62
63 //设置显示器背景
64 void setBackground(){
65  u32 *memAddr;
66  int idxRow;
67  int idxCol;
68  memAddr = (u32 *) DISP_BASE_ADDR;
69  for(idxRow = 0;idxRow < SCREEN_Y; idxRow++){
70          for(idxCol = 0;idxCol < SCREEN_X; idxCol++){
71                  if(idxRow < SCREEN_Y/2){
72                          memAddr[idxCol + idxRow*SCREEN_X] = 0;
73                  }
74                  else{
75                          memAddr[idxCol + idxRow*SCREEN_X] = 0xFFFFFF;
76                  }
77          }
78  }
79  Xil_DCacheFlush();
80 }
81
82
83
84
85
86 int main()
87 {
88  int status;
89      //初始化VDMA并进行配置
90  run_vdma_frame_buffer(&Vdma, HLS_VDMA_DEV_ID, SCREEN_X, SCREEN_Y,HLS_BASE_ADDR,0,
```

```
0,BOTH);
91    run_vdma_frame_buffer(&Vdma, DISPLAY_VDMA_DEV_ID, SCREEN_X, SCREEN_Y,DISP_BASE_ADDR,0,
0,BOTH);
92    status = initdokernel();
93    if(status != XST_SUCCESS){
94            printf("initialize failed! \n");
95            return status;
96    }
97    setBackground();
98    //设置Kernel_bus,用于设置图像采用的卷积运算算子
99    XDokernel_Write_kernel_Bytes(&doKernel,0,edge_kernel,9);
100
101   //设置操作类型operation 0:求卷积和;
102   XDokernel_Set_operation(&doKernel,0);
103 //
104   while(1){
105           XDokernel_Start(&doKernel);
106           while(!XDokernel_IsDone(&doKernel)){
107
108           }
109   }
110         print("Hello World\n\r");
111
112         return 0;
113 }
```

SDK 的主要程序如下，完成了对其中 VDMA 的初始化和和对图像处理 IP 的初始化。

其中第 99 行用于设置要采取的卷积核(kernel)。第 102 行设置为求和操作。第 104 到 109 的循环用于使能图像处理 IP 开始工作。

准备完毕后按照下图所示连接好开发板，将 OV5640 模块插到 JP1 接口：

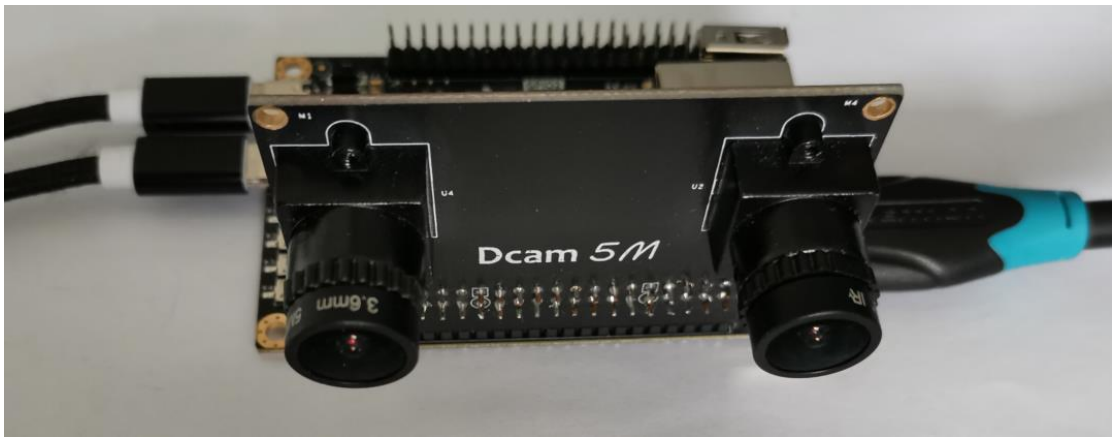


图 10- 15

编译完成后，点击运行，连接到显示器后可以观察输出结果。首先选择 impulse_kernel，观察显示器上出现了所采集到的图像的灰度图像，这是由于在 HLS 中，首先进行的就是将图像转换为灰度图像。

图 10- 16 摄像头采集到的经过灰度转换后的画面

接下来选择 sobel Kernel 观察输出的图像结果，可以看到输出的图像为边缘。

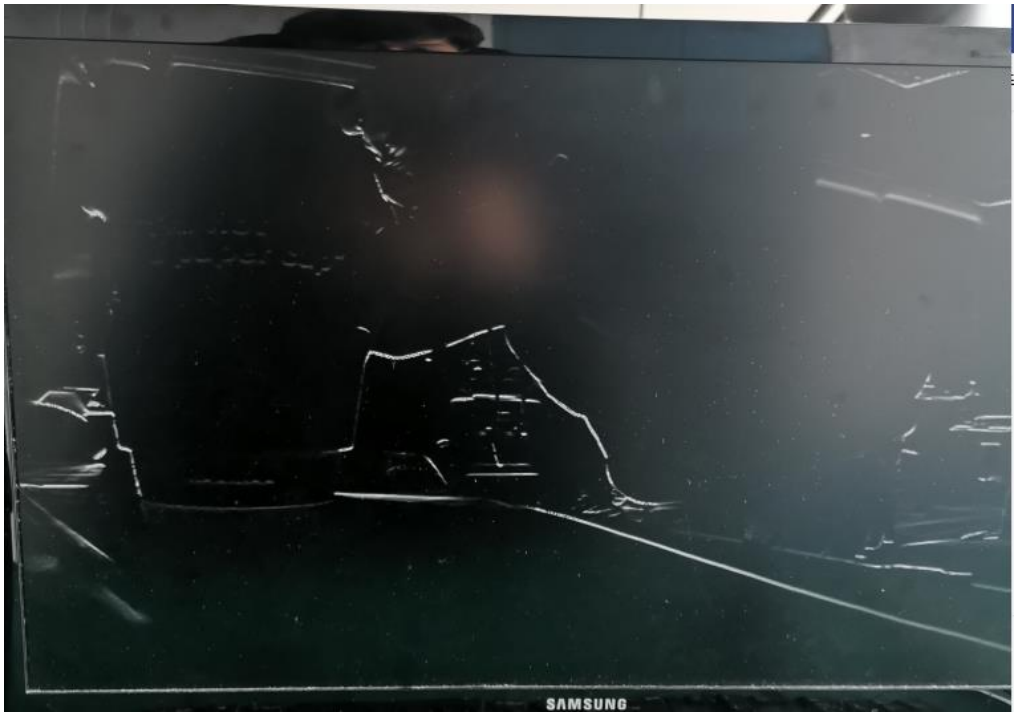


图 10- 17

接下来选择 edgeKernel 观察输出的图像结果，同样可以看到输出的图像为边缘，但这个边缘信息更为细腻。

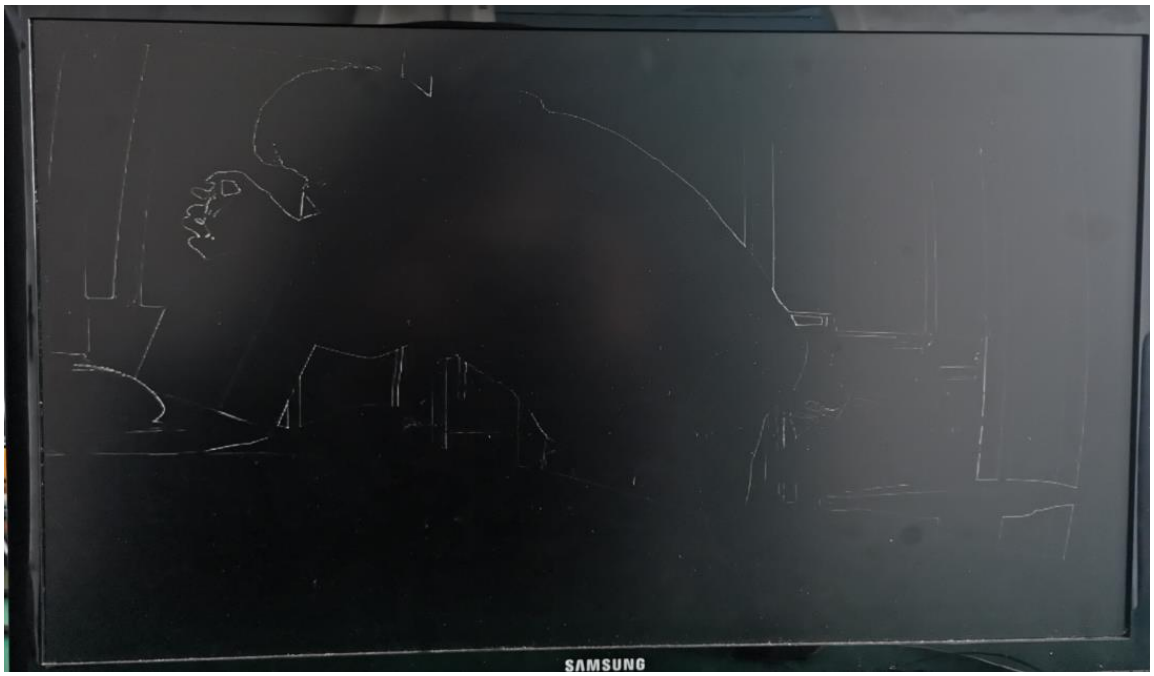


图 10- 18

至此完成了边缘检测功能。可以看到在 HLS 中还可以对 IP 进一步完善，是整个系统能够达到更好的显示效果。

10.6 实验总结

本次实验在上一章实验的基础上，更进一步使用到了 OV5640 来采集图像，并对采集到的图像进行处理，完成了简单的边缘检测。可以看到，使用 HLS 在将某些图像算法移植到硬件上 FPGA 实现还是比较简单和容易实现的。当然，本章实验中也还存在某些问题，比如由于 LInebuffer 导致的图像有两行两列的偏移，对摄像头采集到到的数据是 rgb565 格式转换为 rgb888 格式时带来的精度的损失等。

总的来说，本次实验是一个较为综合性的实验，从 HLS 到 Vivado 到 SDK 都需要掌握。

第11章　快速角点检测

11.1 实验背景

在前面的章节中，已经介绍过 sobel 边缘检测，sobel 边缘检测是用于检测边缘是用来判断水平或者竖直方向上梯度的变化，在水平或者竖直方向上梯度变化明显的地方即是边缘。

角点检测与 Soebl 边缘检测有相似的地方，对于边缘，只要在某两个方向上的梯度很大即可判定该处为边缘，对于角来说，需要满足其在各个方向上的梯度变化都很明显才能判定其为角点。

关于角点检测，在数学上有严格的推导，感兴趣的客户，可以试着去看看其推导过程。在本次实验中，将采用 Xilinx HLS 中提供的函数来完成角点检测的功能。

本次实验参照的是 Xilinx 提供的 demo XAPP1167，该 demo 可以在 Xilinx 官网下载，在我们提供的参考文件中，也可以找到其功能，其中 Xilinx 官方还提供了一些其他的 demo 供用户参考。

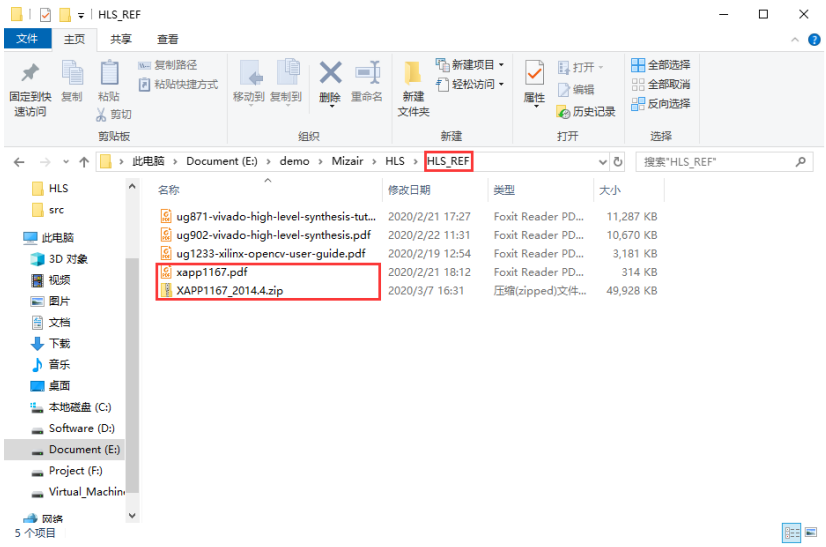


图 11- 1

本次实验想要实现的目标是将 OV5640 采集到的图像转换为灰度图像，并将在采集到的图像中，对角点位置做出标识，能够直观地在显示器上显示出来。

最终效果如下：可以看到，在显示器中，出现了 OV5640 采集到的图像，并将其中的角点位置做出了标注。

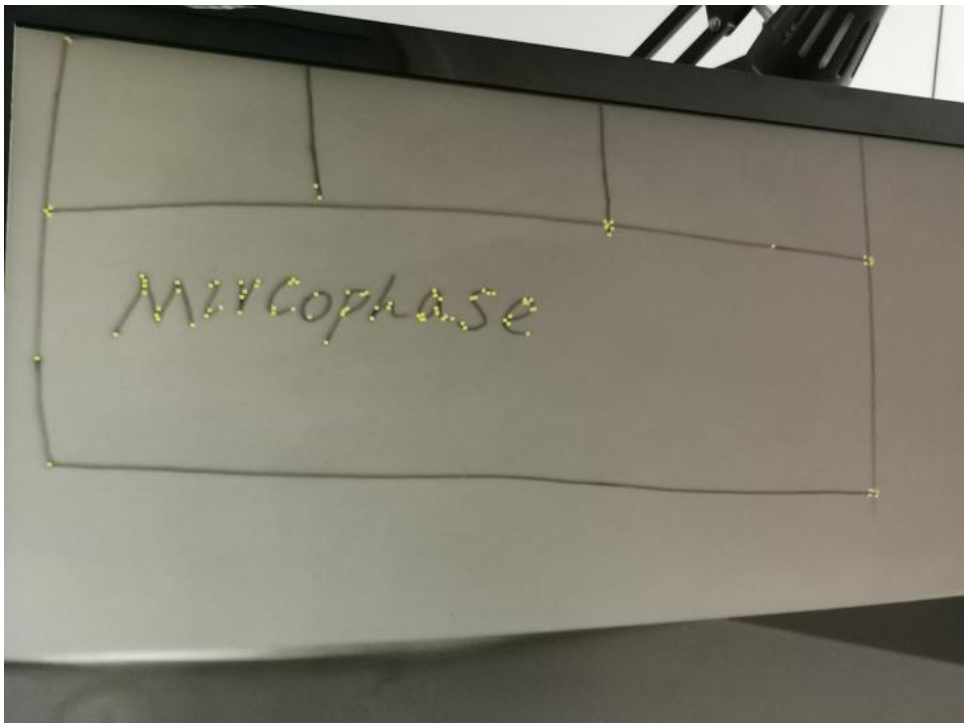


图 11- 2

11.2 HLS 实现角点检测

头文件的定义如下：头文件中定义了常用的图像格式和像素的格式。

```
#ifndef __CORNER_HEAD_H_
#define __CORNER_HEAD_H_

#include "hls_video.h"
#include "ap_axi_sdata.h"

//定义图像的大小
#define WIDTH 1024
#define HEIGHT 768

//定义输入输出图像
#define SRC_IMAGE "screen.bmp"//输入图像路径
#define DST_IMAGE "DstImage.bmp"//输出图像路径
#define GOLD_IMAGE "GoldImage.bmp"//参考图像路径

//定义图像数据类型

//定义 AXI-Stream 数据流
typedef ap_axiu<24,1,1,1> int_sideChannel;
typedef hls::stream<ap_axiu<24,1,1,1> > AXI_STREAM ;

//定义图像矩阵，其中图像矩阵中的像素格式为3 通道8 位无符号数，RGB 图像
typedef hls::Mat<HEIGHT,WIDTH,HLS_8UC3> IMAGE_RGB;

//定义图像矩阵的每个像素格式，3 通道8 位无符号型
typedef hls::Scalar<3,unsigned char> PIXEL_RGB;

//定义图像矩阵，其中图像矩阵中的像素格式为1 通道8 位无符号数，灰度图像
typedef hls::Mat<HEIGHT,WIDTH,HLS_8UC1> IMAGE_GRAY;

//定义图像矩阵的每个像素格式，1 通道8 位无符号型
typedef hls::Scalar<1,unsigned char> PIXEL_GRAY;


//top function
void rgb2gray(IMAGE_RGB & imgIn, IMAGE_RGB & imgOut_3C, IMAGE_GRAY &
```

```
imgOut_1C);
37 void doCorner(AXI_STREAM & inStream,AXI_STREAM & outStream);
38 #endif
```

在角点检测的顶层函数中，会调用一个灰度转换函数，该灰度转换函数负责将输入的 RGB 图像转换为三通道的灰度图像和单通道的灰度图像。其中三通道的灰度图形，其每一个通道上的值都是灰度值，这是根据灰度图像显示的原理得到的：灰度图像最终想要以 24 位显示在显示器上，需要先将 RGB 图形转换位灰度图像，再将每一个通道的值用灰度值替换，这样最终显示在屏幕上的就是转换后的灰度图像。

其中单通道的灰度图像输出，负责将 RGB 图片转换位灰度图像，便于下一步的处理。

值得说明的是，HLS 函数库中也提供有图像转换函数 hls::cvtColor，但是在本次实验中，自己动手写一个灰度转换函数的原因是因为，调用系统自带的函数进行图像的颜色转换，在综合映射成 rtl 电路时，会导致生成的 IP 在时序上不满足。因此自己动手完成一个灰度转换的函数，能够有效避免时序违例带来的问题。

```
void rgb2gray(IMAGE_RGB & imgIn, IMAGE_RGB & imgOut_3C, IMAGE_GRAY & imgOut_1C){
    PIXEL_RGB     pixIn;
    PIXEL_RGB     pixOut_3C;
    PIXEL_GRAY    pixOut_1C;

    for(int idxRow = 0; idxRow < HEIGHT; idxRow++){
        for(int idxCol = 0; idxCol < WIDTH; idxCol++){
            //输入图像是三通道的 RGB 图像
            imgIn >> pixIn;
            unsigned short R = pixIn.val[0];
            unsigned short G = pixIn.val[1];
            unsigned short B = pixIn.val[2];

            //输出三通道图像，各个通道都是灰度值，最终显示出来的图像是灰度图像
            pixOut_3C.val[0] = (unsigned char) ((R*76 + G*150 + B*30) >> 8);
            pixOut_3C.val[1] = (unsigned char) ((R*76 + G*150 + B*30) >> 8);
```

```
                pixOut_3C.val[2] = (unsigned char) ((R*76 + G*150 + B*30) >> 8);

                //输出单通道图像，只有一个通道，为灰度图像
                pixOut_1C.val[0] = (unsigned char) ((R*76 + G*150 + B*30) >> 8);
                imgOut_3C << pixOut_3C;
                imgOut_1C << pixOut_1C;
            }
        }
}
```

定义好该函数后，接下来可以来实现顶层函数，本次实验只需要调用 HLS 图像处理函数库中的函数即可。如下：

```
void doCorner(AXI_STREAM & inStream,AXI_STREAM & outStream){
#pragma HLS INTERFACE axis    port=outStream
#pragma HLS INTERFACE axis    port=inStream
#pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS


  IMAGE_RGB img_0;
  IMAGE_RGB img_1;
  IMAGE_GRAY img_2;
  IMAGE_RGB img_3;
  IMAGE_GRAY mask;
  IMAGE_GRAY dmask;

#pragma HLS dataflow
#pragma HLS stream depth=20000 variable=img_1_.data_stream
  hls::AXIvideo2Mat(inStream, img_0);
  PIXEL_RGB color(255,255,0);
  rgb2gray(img_0, img_1, img_2);
  hls::FASTX(img_2, mask, 20, true);//快速角点检测
  hls::Dilate(mask, dmask);//对检测出来的点进行膨胀
  hls::PaintMask(img_1, dmask, img_3, color);//将膨胀后的点添加到图像上
  hls::Mat2AXIvideo(img_3, outStream);
}
```

对于其中使用到的函数，在 UG902 中可以找到，下面对这些函数做简单介绍。笔者在做本次实验的时候，参考的是 Xapp1167 中 fast_corner 的示例 demo。

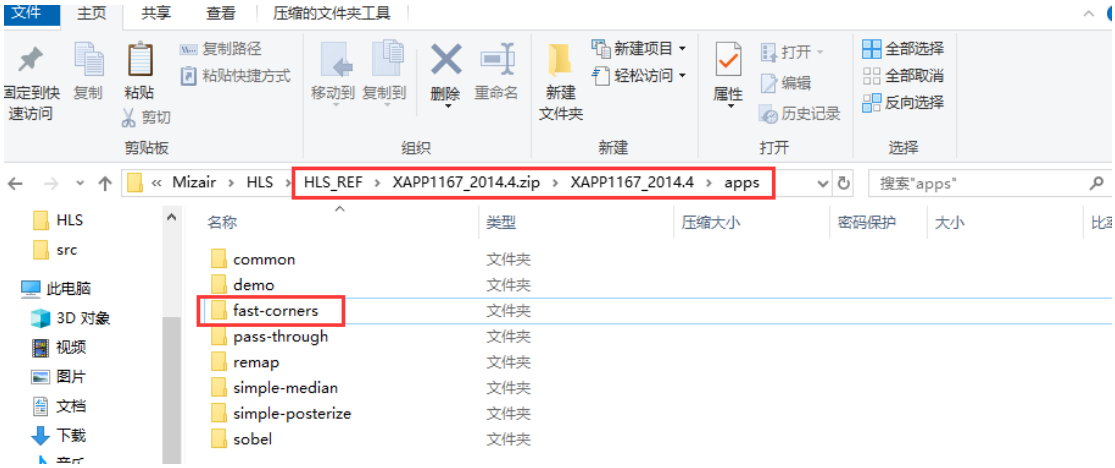


图 11- 3

11.2.1 快速角点检测函数

快事角点函数的作用：实现快速拐角检测器，生成拐角掩码或坐标数组

快速角点检测函数的模板如下：可以看到该函数有两种形式，在本次实验中使用到的是第一中形式。

hls::FASTX

Synopsis

```
template<int SRC_T,int ROWS,int COLS>
void FASTX(
        hls::Mat<ROWS,COLS,SRC_T>     &_src,
        hls::Mat<ROWS,COLS,HLS_8UC1> &_mask,
        int     _threshold,
        bool    _nomax_supression);

template<typename T, int N, int SRC_T,int ROWS,int COLS>
void FASTX(
        hls::Mat<ROWS,COLS,SRC_T>     &_src,
        Point_<T> (&_keypoints)[N],
        int     _threshold,
        bool    _nomax_supression);
```

图 11- 4 快速角点检测函数

关以形参的描述如下图所示：其中 src 是输入的原图像。

mask 将会在一幅与输入图像同样大小的灰度图像上角点所对应的地方以白

色标识出来。

Threshold 是用来判断是否位角点的阈值，该值越大角点检测出来的结果显示为角点的区域越小。

Nomax_supression：如果为真，则启用非极大值抑制，该参数是有角点检测的数学推导而来，在本次实验中保持为真。

Table 4-31: **Parameters**

Parameter	Description
src	Input image
mask	Output image with value 255 where corners are detected
keypoints	Array of the coordinates of detected corners
threshold	FAST detector threshold. If a pixel differs from the center pixel of the window by more than this threshold, then it is either a light or a dark pixel.
nomax_supression	If true, then enable suppression of non-maximal edges

图 11- 1 参数含义

11.2.2 膨胀函数

如代码中第 20 行所示 hls::Dilate(mask, dmask);是对图像进行膨胀操作，在前面的章节中，已经介绍过膨胀的概念，在本次实验中调用 HLS 自带的函数来对图像进行膨胀，mask 是输入的图像，dmask 是经过膨胀后的图像。膨胀的目的是为了让角点更加明显。

hls::Dilate

Synopsis

Default:

```
template<int ROWS, int COLS, int SRC_T, int DST_T>
void hls::Dilate (
     hls::Mat<ROWS, COLS, SRC_T>& src,
     hls::Mat<ROWS, COLS, DST_T>& dst);
```

Custom:

```
template<int ROWS, int COLS, int SRC_T, int DST_T, int K_ROWS, int K_COLS, typename
K_T, int Shape_type, int ITERATIONS>
void hls::Dilate (
     hls::Mat<ROWS, COLS, SRC_T>&      src,
     hls::Mat<ROWS, COLS, DST_T>&      dst,
     hls::Window<K_ROWS, K_COLS, K_T> & kernel);
```

图 11- 2

11.2.3 绘制掩膜

如第 21 行所示 hls::PaintMask(img_1, dmask, img_3, color) 完成的时候将掩膜 mask 与输入图形对应的位置用特殊颜色标记出来。Color 在 17 行已经定义。在第 17 行定义了一个彩色像素：PIXEL_RGB color(255,255,0);

其中关于 paint mask 函数的定义如下：

hls::PaintMask

Synopsis

```
template<int SRC_T,int MASK_T,int ROWS,int COLS>
void PaintMask(
        hls::Mat<ROWS,COLS,SRC_T>        &_src,
        hls::Mat<ROWS,COLS,MASK_T>       &_mask,
        hls::Mat<ROWS,COLS,SRC_T>        &_dst,
        hls::Scalar<HLS_MAT_CN(SRC_T),HLS_TNAME(SRC_T)> _color);
```

图 11- 3

Table 4-47: **Parameters**

Parameter	Description
src	Input image
mask	Input mask
dst	Output image
color	Color for marking

图 11- 4

最终完成在角点对应的位置，用特定的颜色将其标记出来。

对于本次使用到的函数，就简单介绍到这里，可以看到使用 HLS 提供的库，能够大大地加快开发效率。

11.2.4 添加 Directives

在本次实验中，需要添加的 directive 其中一条十分重要，笔者在没添加该 directive 时，在 C/RTL 联合仿真的 RTL 仿真阶段总是出错，在参考了 xapp1167 后添加了该 Directive，最终解决了问题。

```
#pragma HLS stream depth=20000 variable=img_1_.data_stream
```

大致是能够对数据流进行缓冲，相当于对数据流的延时，否则若暂时没有使用到该流的数据时进行了其他操作，就无法再访问该流中的数据了。与 axi_stream 格式的数据是单向流动的类似。

11.3 HLS 仿真与综合

原始图像如下所示：

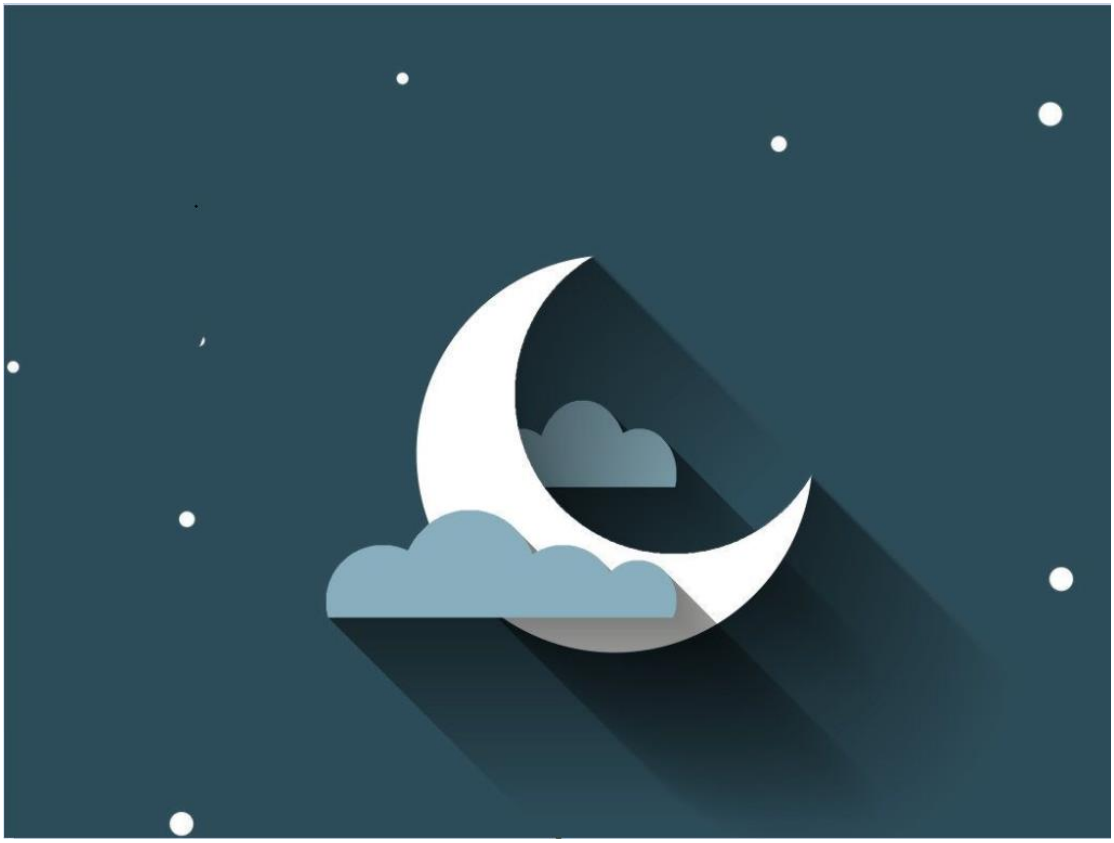


图 11- 5 原始图像

添加完 C 仿真文件后即可开始进行 C 仿真，仿真运行的结果如图所示：

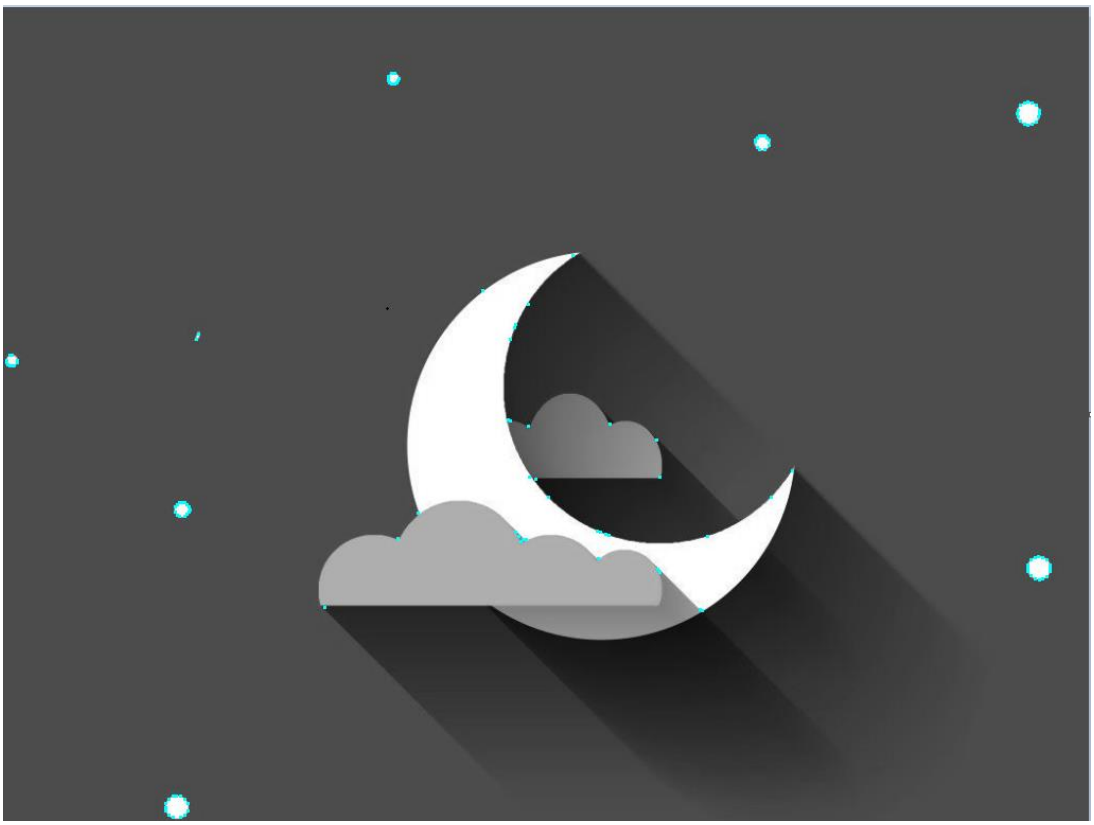


图 11- 6

可以看到 C 仿真的结果，已经将图像转换为灰度图像，并在角点位置处将其标出。

接下来可以进行 C 综合和 C 仿真。仿真结束后即可导出 IP 给到 Vivado 中和 SDK 中使用。

11.4 搭建 Block Design

本次实验的 Block design 与上次实验相比，只需要将 HLS 生成的 IP 进行更改即可。其他模块都可以不做更改。最终搭建的 Block Design 如下：

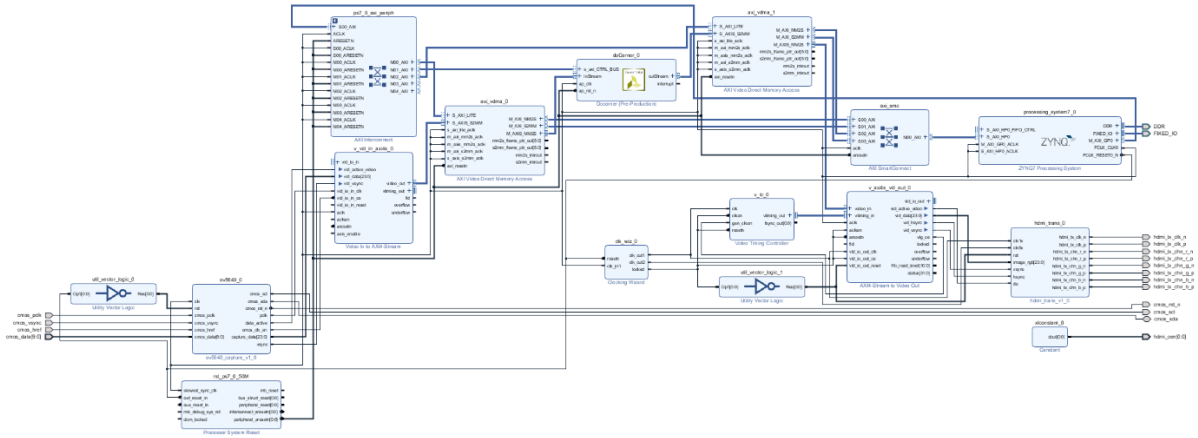


图 11- 11

生成 bit 文件后导入到 SDK 中进行 SDK 的开发。本次实验的 SDK 开发也很简单，再这里不再做过多说明。

11.5 下载验证

一切准备就绪后，按照下图所示连接好开发板：

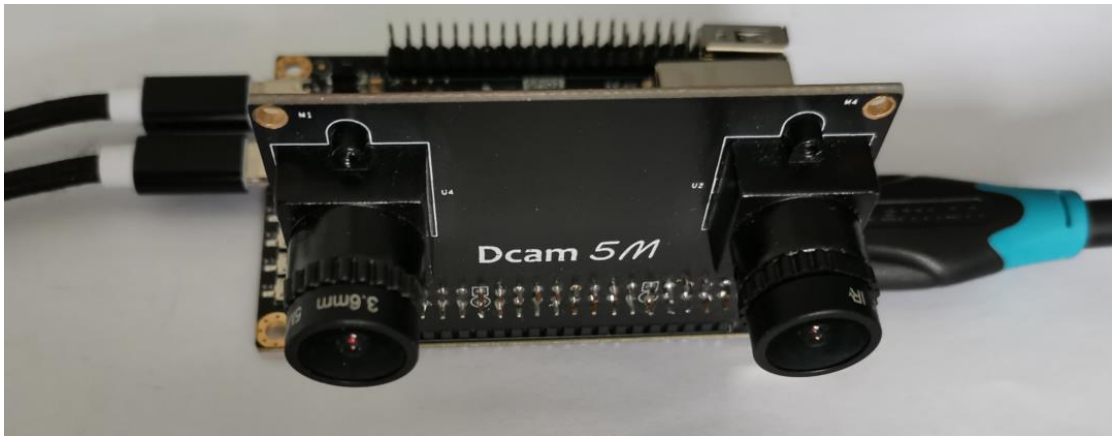


图 11- 12

然后通过 SDK 启动程序，并点击运行

可以在显示器上看到显示结果如下图：

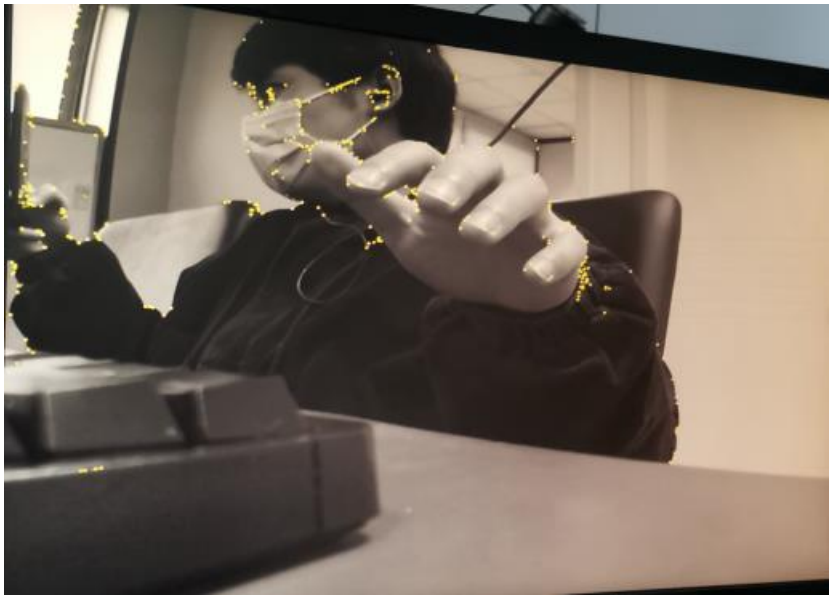


图 11- 13

可以看到，显示器显示的画面中，已经出现了标记点，接下来自己画一个方框，看能否检测到方框中的点。

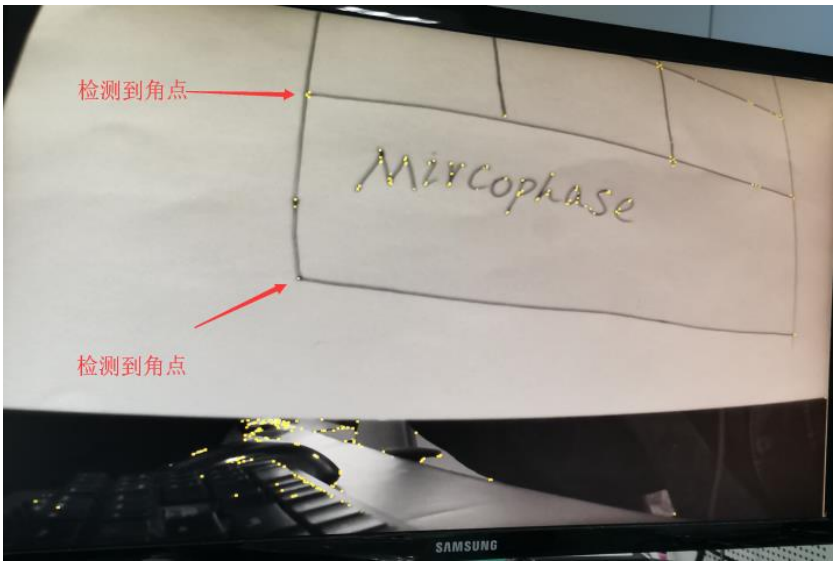


图 9- 19

可以看到，已经成功检测到了画面中角落的位置。

11.6 实验总结

本次实验通过使用 HLS 中的图像处理函数完成了快速角点检测的功能，通过使用自带的函数能够加快开发效率。同时也应该注意到，快速角点检测对噪声铭感，所以画面中存在许多检测不正确的点，可以通过采用高斯滤波或者使用其他方法来减小噪声，能够提高角点检测的准确度。

第一版总结

感谢您选择微相科技（上海）有限公司的产品，本文档旨在帮助用户使用 Z7-Lite ZYNQ7000 系列开发板进行 HLS 开发的快速入门，相信用户完整的浏览过本教程后，对使用 HLS 进行开发有了一个初步的认识。本文档中各章实验都提供有配套源码，用户可以根据源代码和工程来观察实验现象，作为本人初次编写的 HLS 技术文档，其中难免会有错误，请你发现后及时与我们联系，我在之后的版本中会及时对本文档进行更改。

本文档的内容不会是一成不变的，在将来还会更新更多关于 HLS 进行开发的工程示例，还请您持续关注，最后祝您工作顺利。