

版本信息：
版本
REV2018
时间
12/16/2017

ZYNQ 修炼秘籍

基于米联客系列开发板

第五季 基于 ZYNQ 的 HLS 图像算法设计

电子版自学资料

常州一二三电子科技有限公司
溧阳米联电子科技有限公司
版权所有

米联客学院 03QQ 群： 543731097
米联客学院 02QQ 群： 86730608
米联客学院 01QQ 群： 34215299

版本	时间	描述
Rev2016	2015-07-25	第一版初稿，大部分采用 zedboard 资料
Rev2017	2017-01-31	做了重大改进，自己编写里批处理命令，方便移植
Rev2018	2017-12-16	对 2017 版本改进，修改教程 bug 同时增加更多学习课程

感谢您使用米联客开发板团队开发的 ZYNQ 开发板，以及配套教程。本教程将对之前编写的《ZYNQ 修炼秘籍》-LINUX 部分内容做出改进，并且增加新的课程内容。本教程不仅仅适合用于米联客开发板，而且可以用于其他的 ZYNQ 开发。

软件版本：VIVADO2015.4 (linux 部分安装主要用到里面的交叉编译环境)

软件版本：VIVADO2016.4 (首期代码用 2016.4,读者可以自行升级到高版本)

软件版本：VIVADO2017.4 (2017.4 预计在 2018 年 1 月官方发布软件)

版权声明：

本手册版权归常州一二三电子科技有限公司/溧阳联电子科技有限公司所有，并保留一切权利，未经我司书面授权，擅自摘录或者修改本手册部分或者全部内容，我司有权追究其法律责任。

技术支持：

版主大神们都等着大家去提问--电子资源论坛 www.osrc.cn

微信公众平台：电子资源论坛



目录

ZYNQ 修炼秘籍	1
目录	4
S05_CH01_搭建 Modelsim 和 Vivado 联合调试环境	7
1.1 概述	7
1.2 使用 GUI 编译仿真库	7
1.3 使用命令行编译仿真库	9
1.4 HLS 简单介绍	10
1.4.1 OpenCV 和 HLS 视频库	11
1.4.2 AXI4 流和视频接口	12
1.4.3 OpenCV 到 RTL 代码转换的流程	13
1.5 本章小结	13
S05_CH02_shift_led 实验	14
2.1 概述	14
2.2 工程创建、仿真及优化	14
2.2.1 工程创建	14
2.2.2 代码综合	19
2.2.3 代码优化	22
2.2.4 仿真实现	24
2.3 HLS 代码封装	30
2.4 硬件平台实现	31
2.5 本章小结	36
S05_CH03_ImageLoad 实验	37
3.1 概述	37
3.2 图片数据的获取	37
3.3 视频流文件的载入	39
3.4 外部摄像头的调用	41
3.5 工程创建与验证	42
3.6 本章小结	50
S05_CH04_Skin_Dection 实验	51
4.1 肤色检测原理及应用	51
4.2 检测算法实现	51
4.2.1 工程创建	51
4.2.2 代码综合	55
4.2.3 代码优化	58
4.3 仿真测试	64
4.4 本章小结	65
S05_CH05_Sobel 算子硬件实现(一)_HLS 实现	66
5.1 Sobel 原理介绍	66

5.2	Sobel 算子在 HLS 上的实现.....	67
5.2.1	工程创建.....	67
5.2.2	代码优化及仿真.....	71
5.2.3	工程封装.....	78
5.3	代码详解.....	79
5.3	本章小结.....	84
S05_CH06_Sobel	算子硬件实现(二)_硬件验证.....	85
6.1	系统硬件设计.....	85
6.2	硬件工程创建.....	86
6.3	导入到 SDK.....	89
6.4	程序分析.....	96
S05_CH07_基于 Hough 变换的圆检测.....	99	
7.1	Hough 变换原理介绍.....	99
7.1.1	Hough 变换直线检测.....	99
7.1.2	Hough 变换圆检测.....	100
7.1.3	Hough 变换圆检测算法实现流程.....	101
7.2	Hough 在 HLS 上的实现.....	101
7.2.1	工程创建.....	102
7.2.2	仿真及优化.....	110
7.3	程序分析.....	114
7.4	本章小结.....	118
S05_CH08_傅里叶变换的 HLS 实现.....	119	
8.1	FFT 原理介绍.....	119
8.2	HLS 实现.....	122
8.3	Vivado 模块例化及 IP 封包.....	126
8.4	本章小结.....	136
S05_CH09_OTSU 自适应二值化.....	137	
9.1	OTSU 自适应二值化原理简介.....	137
9.2	HLS 实现.....	137
9.2.1	工程创建.....	137
9.2.2	仿真及优化.....	145
9.3	硬件工程实现.....	148
9.3.1	硬件平台搭建.....	148
9.3.2	导入到 SDK.....	149
9.4	程序分析.....	150
9.5	本章小结.....	153
S05_CH10_音频滤波(MZ7XB 不支持)	154	
10.1	ADAU1761 简介.....	154
10.1.1	ADAU1761 收发时序	154
10.1.2	ADAU1761 时钟和采样率	155
10.2	硬件平台的搭建.....	156
10.3	导入到 SDK.....	170
10.4	程序分析.....	175

10.5 本章小结.....	182
S05_CH11_快速角点检测的 HLS 实现.....	183
11.1 角点定义.....	183
11.2 角点检测算法.....	184
11.2.1 Moravec 角点检测算法	184
11.2.2 Harris 角点检测	184
11.2.3 FAST 角点检测算法	185
11.3 HLS 实现.....	187
11.3.1 工程创建.....	187
11.3.2 仿真及优化.....	189
11.4 硬件工程创建.....	195
11.4.1 硬件平台搭建	195
11.4.2 导入到 SDK	196
11.5 程序分析.....	203
11.6 本章小结.....	203
附录(常见问题汇总).....	204
一、工具篇.....	204
1.1 HLS 中文注释乱码问题解决方案	204
1.2 代码字体大小修改	205
二、设计篇.....	206
2.1 hls::stream 仿真警告	206
2.2 仿真时使用 cvShowImage () 函数但是没有任何错误提示仿真界面直接关闭	206

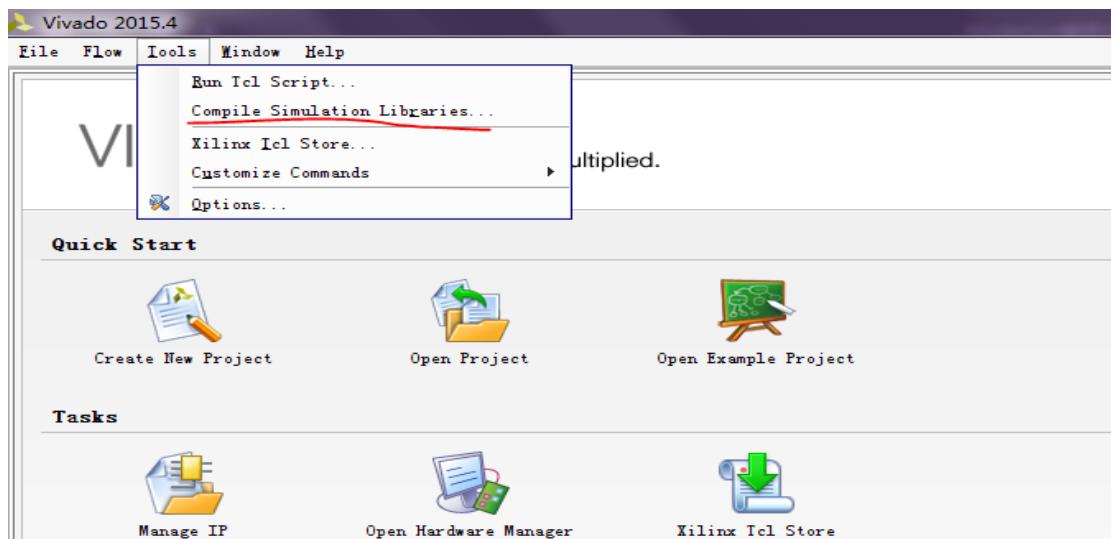
S05_CH01_搭建 Modelsim 和 Vivado 联合调试环境

1.1 概述

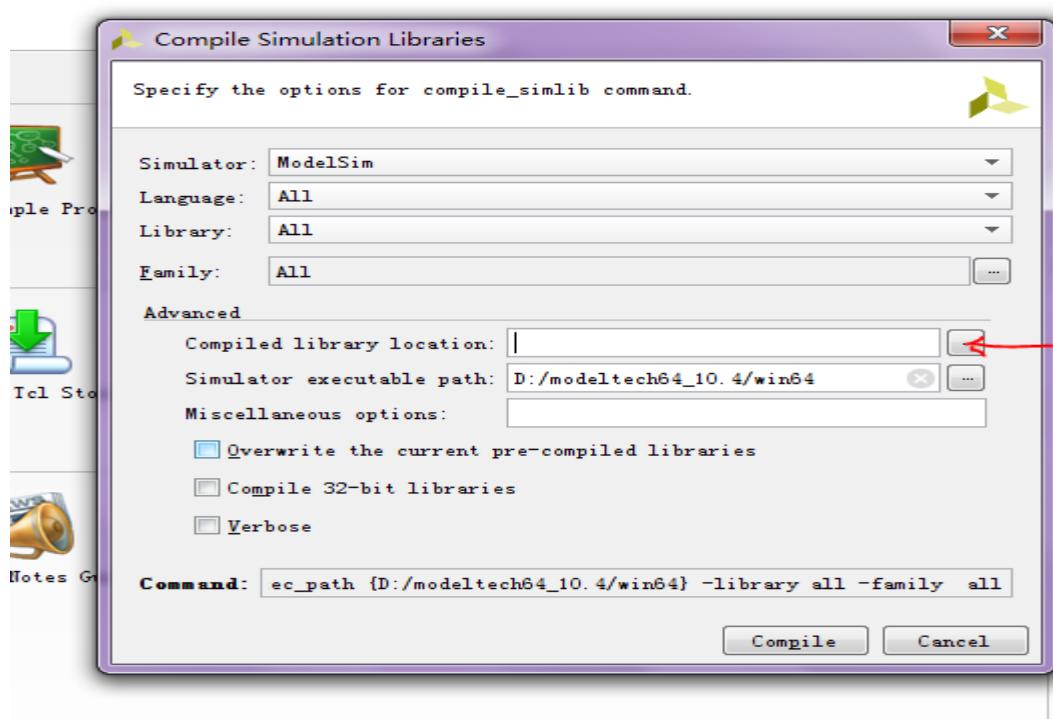
在进行 Vivado HLS 的学习之前，我们先把相应的准备工作做好，所谓工欲善其事，必先利其器，那么开发工具的安装必然是首要的，这里就不在赘述，这一节我们来讲讲如何搭建 Modelsim 与 Vivado 联合仿真环境。

1.2 使用 GUI 编译仿真库

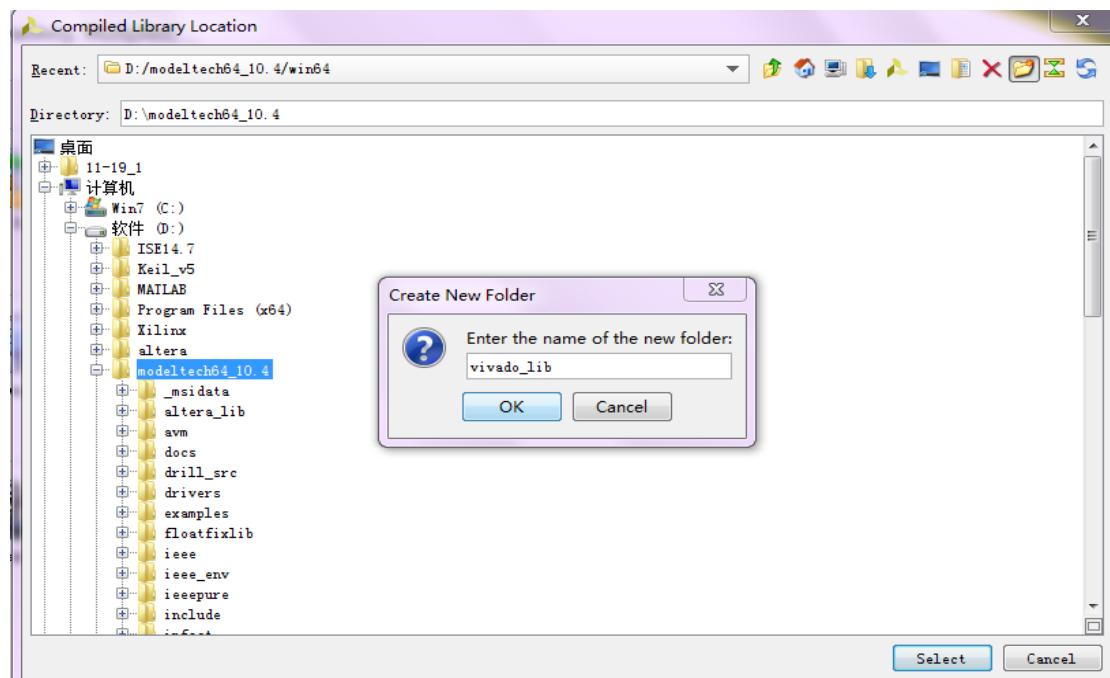
Step1:启动 Vivado，在菜单栏中选择 Tools → Compile Simulation Libraries，



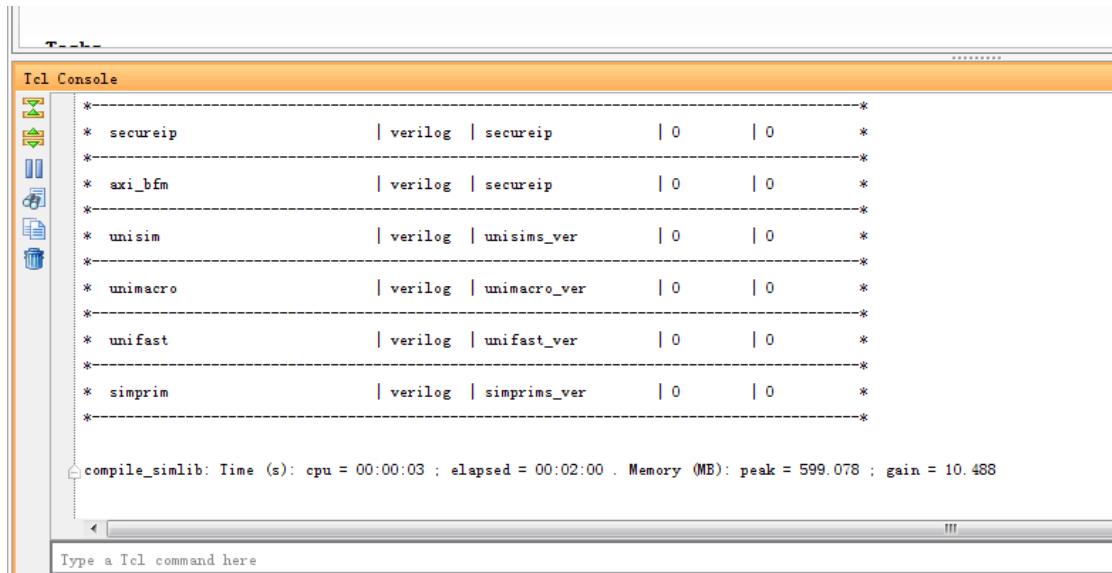
Step2:启动编译界面如下，在红色箭头指向的位置选择默认的仿真库的编译路径，其他的默认



Step3: 新建一个文件夹存放编译库的文件，这里我命名为 vivado_lib，单击 OK 完成操作。



Step4: 点击 Compile 静静等待仿真库文件的生成。



1.3 使用命令行编译仿真库

使用 TCL 脚本：compile_simlib 编译仿真库，下面的命令就可以实现（更多内容参考 ug835）

compile_simlib -directory <library_output_directory>-simulator <agr>
 simulator_exec_path<sim_install_location> 例 如： a) 仿 真 库 编 译 到
 D:/modeltech64_10.4/vivado_lib；仿 真 工 具 使 用 Modelsim；ModelSim 安 装 在
 D:/modeltech64_10.4/win64；那 么 完 整 的 tcl 命 令 就 是：

compile_simlib -directory D:/modeltech64_10.4/vivado_lib -simulator
 modelsim -simulator_exec_path D:/modeltech64_10.4/win64

```

***** Vivado v2015.4 (64-bit)
***** SW Build 1412921 on Wed Nov 18 09:43:45 MST 2015
***** IP Build 1412160 on Tue Nov 17 13:47:24 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Vivado% compile_simlib -directory D:/modeltech64_10.4/vivado_lib -simulator modelsim -simulator_exec_path D:/modeltech64_10.4/win64
Compiling libraries for 'modelsim' simulator in 'D:/modeltech64_10.4/vivado_lib'

Creating modelsim.ini file...
Model Technology ModelSim SE-64 vmap 10.4 Lib Mapping Utility 2014.12 Dec 3 2014
vmap -
Copying D:/modeltech64_10.4/win64../modelsim.ini to modelsim.ini
--> Compiling 'verilog.secureip' library...
  > Source Library = 'D:\Xilinx\Vivado\2015.4\data\secureip'
  > Compiled Path = 'D:\modeltech64_10.4\vivado_lib\secureip'
  > Log File     = 'D:\modeltech64_10.4\vivado_lib\secureip/.cxl.verilog.secureip.secureip.nt64.log'
** Warning: <vlib-34> Library already exists at "D:/modeltech64_10.4/vivado_lib/secureip".
--> Compiling 'verilog.secureip:verilog.axi_bfm' library...
  > Source Library = 'D:\Xilinx\Vivado\2015.4\data\secureip\axi_bfm'
  > Compiled Path = 'D:\modeltech64_10.4\vivado_lib\secureip'
```

当等待一段时间出现如下编译界面时，表明编译无误：

```

c:\ Vivado 2015.4 Tcl Shell - D:\Xilinx\Vivado\2015.4\bin\vivado.bat -mode tcl
* unifast          : vhdl      : unifast           : 0      :
@      *                                         :
*                                         :
* unisim           : verilog   : unisims_ver       : 0      :
@      *                                         :
*                                         :
* unimacro         : verilog   : unimacro_ver     : 0      :
@      *                                         :
*                                         :
* unifast           : verilog   : unifast_ver      : 0      :
@      *                                         :
*                                         :
* simprim          : verilog   : simprims_ver     : 0      :
@      *                                         :
*                                         :

compile_simlib: Time <s>: cpu = 00:00:01 ; elapsed = 00:04:02 . Memory <MB>: peak = 191.508 ; gain = 2.648
Vivado%

```

1.4 HLS 简单介绍

Vivado HLS 是 Xilinx 推出的高层次综合工具，采用 C/C++语言进行 FPGA 设计，HLS 提供了一些 example 样例方便大家熟悉基本的开发流程，另外关于 HLS 的使用介绍，Xilinx 官方有两个非常重要的开发文档，ug871-vivado-high-level-synthesis-tutorial.pdf 和 ug902-vivado-high-level-synthesis.pdf，里面详细介绍了包括怎样建立 HLS 工程，怎么编写 testbench，怎么进行优化等问题。关于优化，上面提到的两篇 PDF 文档里介绍的比较详细，在 HLS 软件界面，点击程序所在的文件，在右侧边栏有个 Directive，里面列出了程序中所有用到的变量、函数和循环结构，点击右键可以给其配置。对循环结构，一般选择 unroll（即展开循环），可以自己设定展开因子 factor。为提高程序的并行化处理，可以给函数选择 PIPELINE。对应数组，可以设置为 ARRAY_PARTITION，数组维数可以自己设定。HLS 软件其实很智能的，简单的结构，一般软件自己会优化好。每一个优化方案都保存在一个 Solution 里，HLS 可以创建多个 Solution，用于比较不用的优化效果。

从工业检测系统到自动驾驶系统，计算机视觉拥有广泛的应用领域。而 OpenCV 包含 2500 多个优化的视频函数的函数库，并且专门针对台式机处理器和图形处理器（Graphic Processing Unit, GPU）进行优化。利用逻辑硬件来加速 OpenCV 的性能在 HLS 上得以实现。

Xilinx 提供的 Vivado HLS 高层次综合工具能够通过 C/C++编写的代码直接创建 RTL 硬件，显著提高设计效率；同时，Xilinx ZynqSOC 系列器件嵌入双核 ARM Cortex-A9 处理器将软件可编程能

力与 FPGA 的硬件可编程能力实现完美结合，以低功耗和低成本等系统优势实现单芯片无以伦比的系统性能、灵活性、可扩展性，加速图形处理产品设计上市时间。

1.4.1 OpenCV 和 HLS 视频库

如下图所示，OpenCV 在视频处理系统中可以有四种不同的应用方式。第一种方式中，算法的设计和实现完全依赖于 OpenCV 的函数调用，利用文件的访问功能进行图片的输入、输出和处理；第二种方式中，算法可以在嵌入式系统（例如 Zynq Base TRD）中实现，利用特定平台的函数调用访问输入输出图像，但是，视频处理的实现依赖于嵌入式系统中处理器（例如 Cortex™-A9）对 OpenCV 功能函数的调用；第三种方式中，处理算法的 OpenCV 功能函数被 Xilinx Vivado HLS 视频库函数替换，而 OpenCV 函数则用于访问输入和输出图像，提供视频处理算法实现的设计原型。Vivado HLS 提供的视频库函数可以被综合，在对这些函数综合后，可以将处理程序模块整合到诸如 Zynq 的可编程逻辑中。这样，这些程序逻辑块就可以处理由处理器生产的视频流、从文件中读取的数据、外部输入的实时视频流。

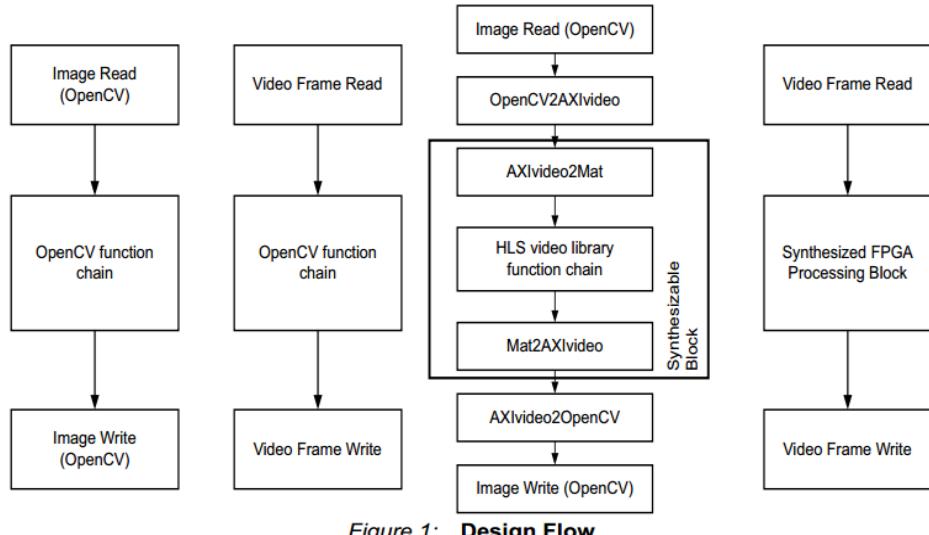


Figure 1: Design Flow

Vivado HLS 包含大量的视频库函数，方便于构建各种各样的视频处理程序。通过可综合的 C++ 代码，实现这些视频库函数。在视频处理功能和数据结构方面，这些综合后的代码与 OpenCV 基本对应。许多视频概念与抽象和 OpenCV 非常相似，很多图像处理模块函数和 OpenCV 库函数一致。

例如，OpenCV 中用于代表图片的很重要的一个类便是 cv::Mat 类，cv::Mat 对象定义如下：

```
cv::Mat image(1080, 1920, CV_8UC3);
```

该行代码声明了一个 1080×1920 像素，每一个像素都是由 3 个 8 位无符号数表示的变量 `image`。对应的 HLS 视频库模板类 `hls::Mat<>` 声明如下：

```
hls::Mat<2047, 2047, HLS_8UC3> image(1080, 1920);
```

这两行代码的参数形式、图像尺寸最大值、语法规则不同，生成的对象是相似的。如果图像规定的最大尺寸和图像的实际尺寸相同的话，也可以用下面的代码替代：

```
hls::Mat<1080, 1920, HLS_8UC3> image();
```

1.4.2 AXI4 流和视频接口

通过 AXI4 流协议，Xilinx 提供的视频处理模块实现像素数据通信。尽管底层 AXI4 流媒体协议不需要限制图片尺寸，但是，若图片尺寸相同，则将会大大简化大部分的复杂视频处理计算。对于遵循 AXI4 流协议的输入接口，可以保证每一帧都包含 `ROWS * COLS` 的像素。在保证前面视频帧保持完整性和矩形性的情况下，后续模块实现对视频帧有效地处理。

如图所示，Vivado HLS 包含 2 个可综合的视频接口转换函数。

Table 2: Vivado HLS Synthesizable Video Functions

Video Library Function	Description
<code>hls::AXIvideo2Mat</code>	Converts data from an AXI4 video stream representation to <code>hls::Mat</code> format.
<code>hls::Mat2AXIvideo</code>	Converts data stored as <code>hls::Mat</code> format to an AXI4 video stream.

视频库还提供了其他不可综合的视频接口函数，这些函数用于在基于 OpenCV 测试平台与综合后的函数结合。下图给出了不可综合的接口函数。

Table 3: Vivado HLS Non-Synthesizable Video Functions

Video Library Functions	
<code>hls::cvMat2AXIvideo</code>	<code>hls::AXIvideo2cvMat</code>
<code>hls::IplImage2AXIvideo</code>	<code>hls::AXIvideo2IplImage</code>
<code>hls::CvMat2AXIvideo</code>	<code>hls::AXIvideo2CvMat</code>

VivadoHLS 视频处理函数库使用 `hls::Mat<>` 数据类型，这种类型用于模型化视频像素流处理，实质等同于 `hls::steam<>` 流的类型，而不是 OpenCV 中在外部 memory 中存储的 matrix 矩阵类型。因此，在用 Vivado HLS 实现 OpenCV 的设计中，需要将输入和输出 HLS 可综合的视频设计接口，修

改为 Video stream 接口，也就是采用 HLS 提供的 video 接口可综合函数，实现 AXI4 video stream 到 VivadoHLS 中 hls::Mat<>类型的转换。

1.4.3 OpenCV 到 RTL 代码转换的流程

OpenCV 图像处理是基于存储器帧缓存而构建的，它总是假设视频帧数据存放在外部 DDR 存储器中。由于处理器的小容量高速缓存性能的限制，因此，OpenCV 访问局部图像性能较差。并且，从性能的角度来说，基于 OpenCV 设计的架构比较复杂，功耗更高。在对分辨率或帧速率要求低，或者在更大的图像中对需要的特征或区域进行处理时，OpenCV 似乎足以满足很多应用的要求；但是，对于高分辨率和高帧率实时处理的应用中，OpenCV 很难满足高性能和低功耗的需求。

基于视频流的架构能提供高性能和低功耗，链条化的图像处理函数减少了外部存储器访问。针对视频优化的行缓存和窗口缓存比处理器高速缓存更简单高效，更易于使用 Xilinx 提供的 Vivado HLS 在 FPGA 器件中采用数据流优化来实现。

Vivado HLS 对 OpenCV 的支持，不是指可以将 OpenCV 的函数库直接综合成 RTL 代码，而是需要将代码转换为可综合的代码。这些可综合的视频库称为 Vivado HLS 视频库，它们由 Vivado HLS 工具提供。

由于 OpenCV 函数一般都包含动态的内存分配、浮点以及假设图像在外部存储器中存放或者修改，所以不能直接通过 Vivado HLS 对 OpenCV 函数进行综合。

Vivado HLS 视频库用于替换很多基本的 OpenCV 函数，它与 OpenCV 具有相似的接口和算法，它主要针对在 FPGA 架构中实现的图像处理函数，其中包含了专门面向 FPGA 的优化，比如定点运算而非浮点运算（不必精确到比特位），片上的行缓存（line buffer）和窗口缓存（window buffer）。

1.5 本章小结

本章是 HLS 部分的第一章，为大家简单的介绍了一些关于 HLS 的基础知识，为下面正式开始学习 HLS 打下基础。HLS 的优势在于，它可以把 C 语言转化为我们的硬件描述语言，这意味着，因此，它可以让更多不懂硬件描述语言的开发者投入到 FPGA 的开发中来。一些优秀的 C 算法，我们也可以使用 HLS 将其转换为硬件描述语言，而且其质量也要比人工设计的更好，节省了人力和物力。

S05_CH02_shift_led 实验

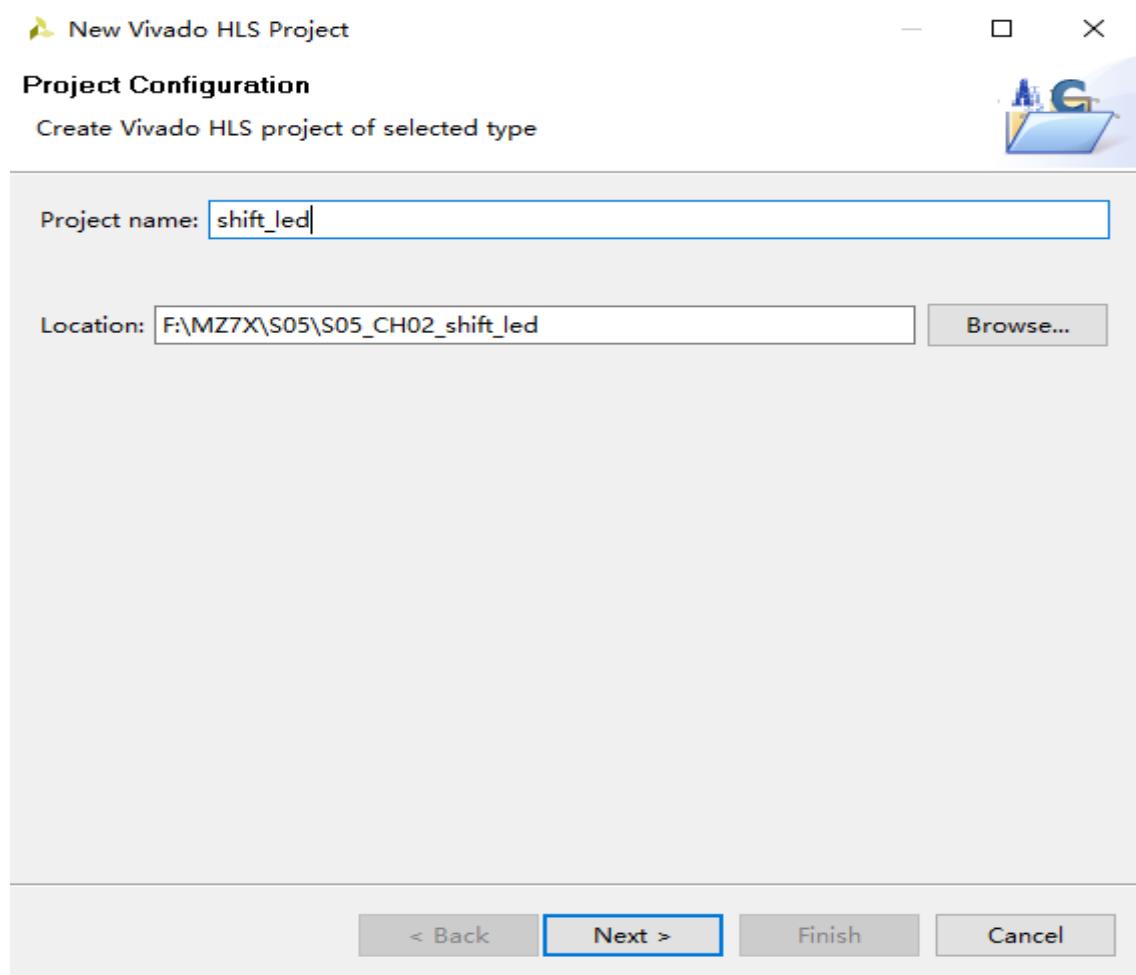
2.1 概述

如同软件开发都是从“Hello World!”进入编程的大门一样，这一章我们就通过 HLS 封装一个移位流水灯的程序熟悉 HLS 的开发流程，包括工程的创建，仿真，综合，封装，以及在硬件平台上的实现。

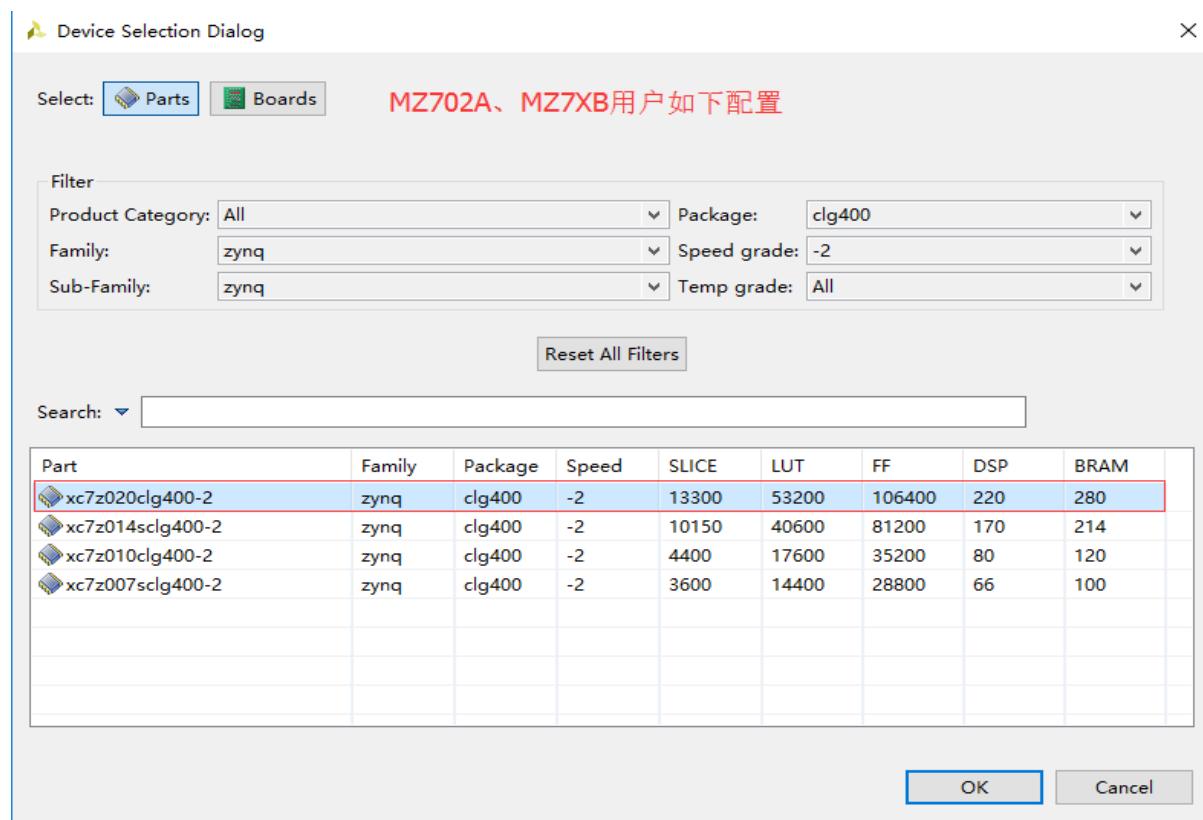
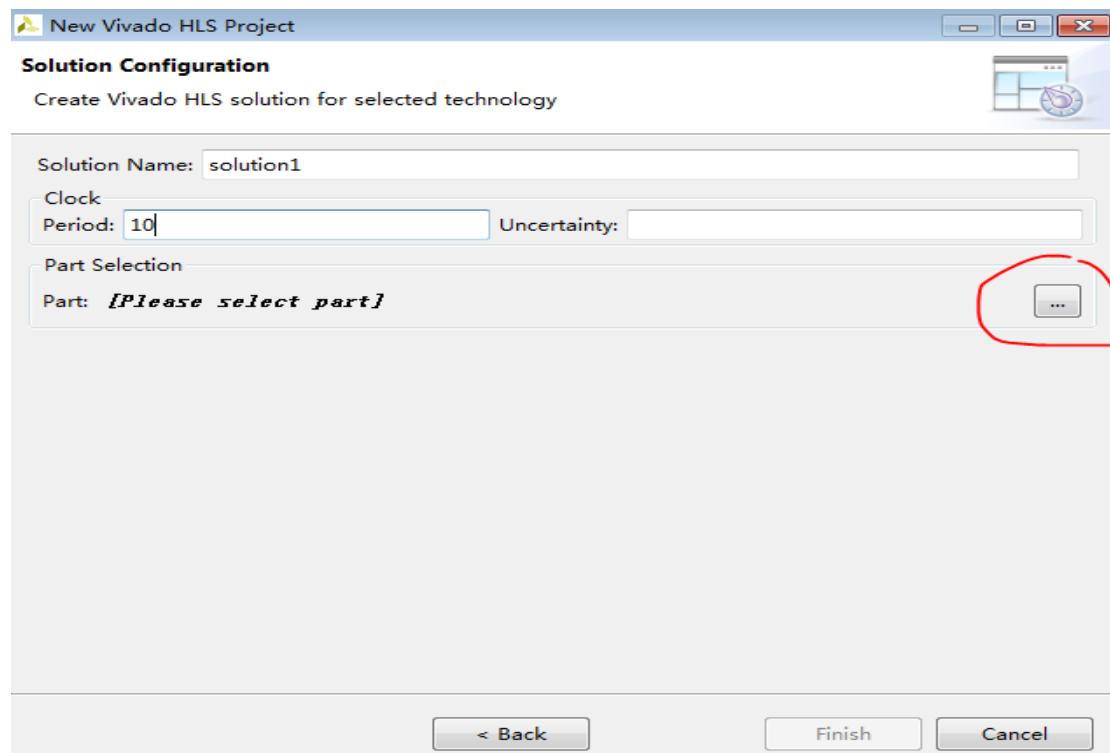
2.2 工程创建、仿真及优化

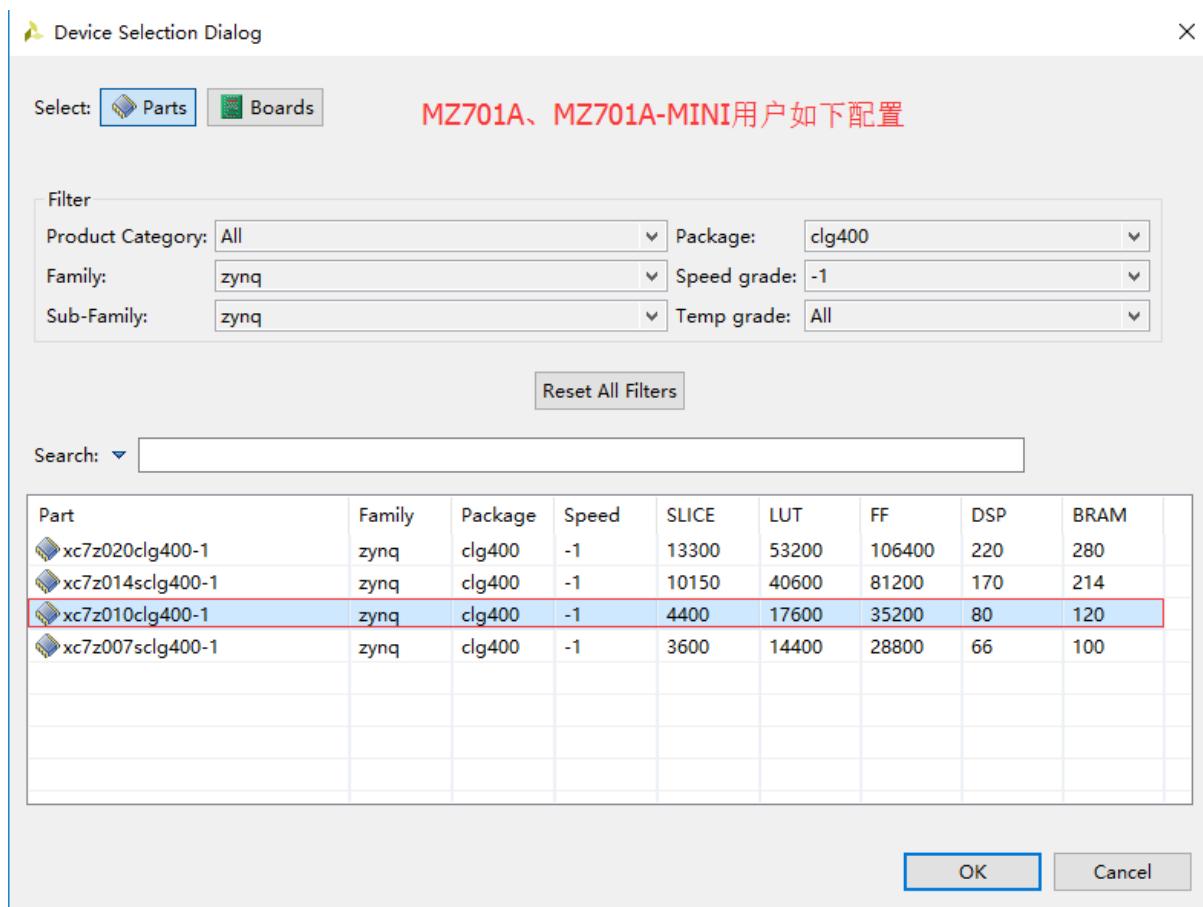
2.2.1 工程创建

Step1: 打开 Vivado HLS 开发工具，单击 Create New Project 创建一个新工程，设置好工程路径和工程名，一直点击 Next 按照默认设置，

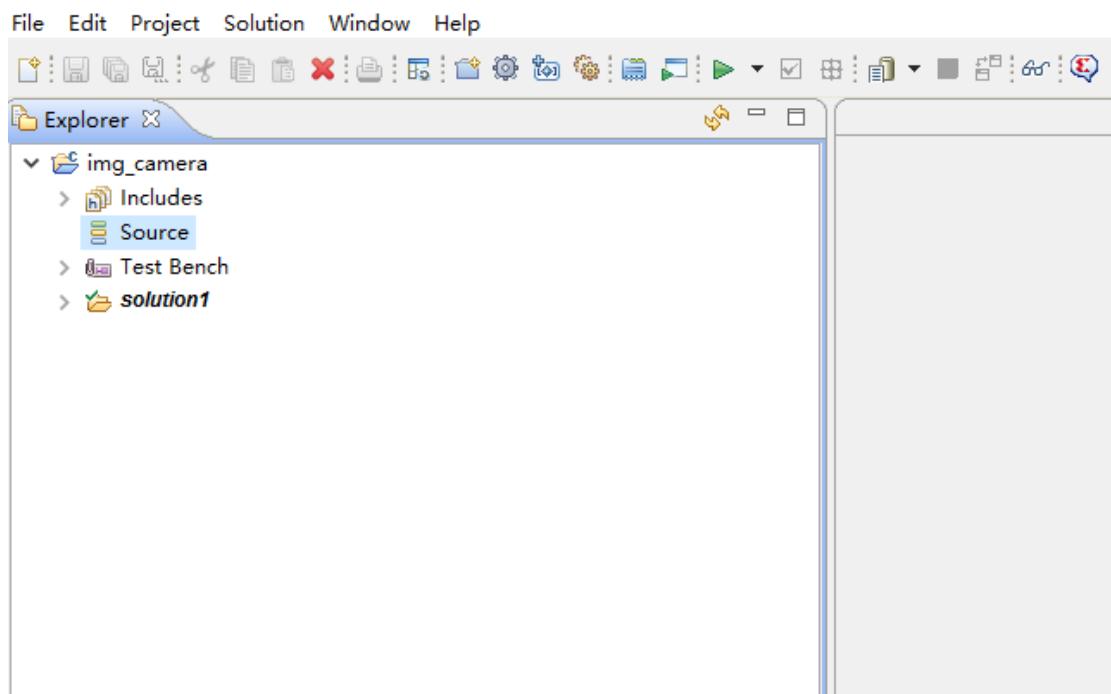


Step2: 出现如下图所示界面, 时钟周期 Clock Period 按照默认 10ns, Uncertainty 和 Solution Name 均按照默认设置, 点击红色圆圈部分选择芯片类型, 然后点击 OK。

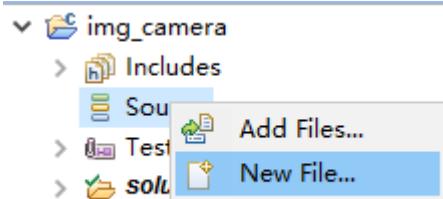




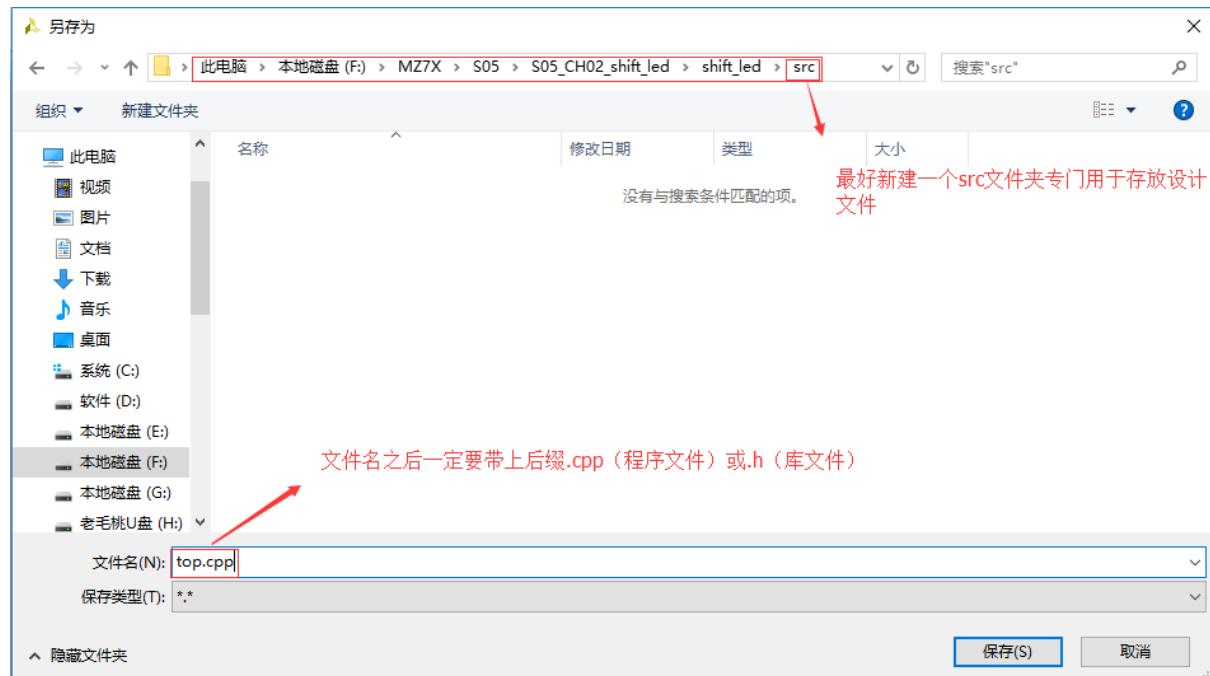
Step3: 点击 Finish, 出现如下界面:



Step4: 右单击 Source, 选择 New file, 添加一个设计源文件。



Step5：输入文件名和选择保存的路径，然后单击保存。



Step6：双击刚才添加的文件，添加如下程序。

```
#include "shift_led.h"

void shift_led(led_t *led_o, led_t led_i)
{
#pragma HLS INTERFACE ap_ovld port=led_o
#pragma HLS INTERFACE ap_ovld port=led_o
#pragma HLS INTERFACE ap_vld port=led_i

    led_t tmp_led;
    cnt32_t i;//for循环延时量
    tmp_led = led_i;
    for(i = 0; i < MAX_CNT; i++)
    {
        if(i==SHIFT_FLAG)
        {
            tmp_led = ((tmp_led>>7)&0x01) + ((tmp_led<<1)&0xFE); //左移
            *led_o = tmp_led;
    }
}
```

```
    }
}
}
```

Step7：按照同样的方法添加一个 shift_led.h 文件，添加下面的程序。

```
#ifndef _SHIFT_LED_H_
#define _SHIFT_LED_H_
//加入设置int自定义位宽的头文件
#include "ap_int.h"

//设置灯半秒动一次，开发板时钟频率是100M
//#define MAX_CNT 1000/2 //仅用于仿真，不然时间较长
#define MAX_CNT 100000000/2
#define SHIFT_FLAG MAX_CNT-2

//typedef int led_t;
//typedef int cnt32_t;//计数器
typedef ap_fixed<4, 4> led_t;
//第一个4代表总位宽，第二个4代表整数部分的位宽是8，则小数部分位宽=4-4=0
typedef ap_fixed<32, 32> cnt32_t;
void shift_led(led_t *led_o, led_t led_i);
#endif
```

Step8：单击 Test Bench，添加一个名为 Test_shift_led.cpp 的测试文件，并添加如下程序。

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

using namespace std;

int main()
{
    led_t led_o;
    led_t led_i = 0xFE;//1111_1110
    const int SHIFT_TIME = 8;
    int i;
```

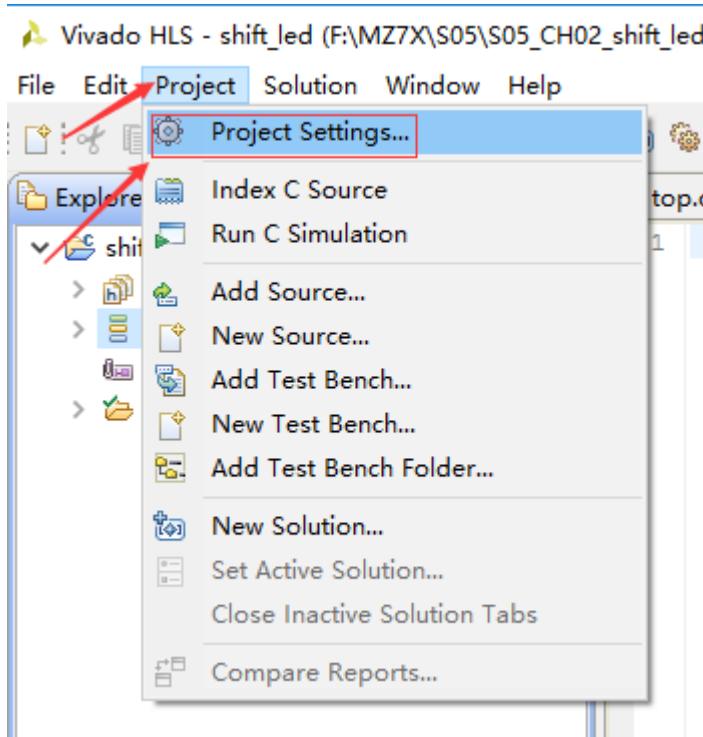
```

for(i = 0;i < SHIFT_TIME;i++)
{
    shift_led(&led_o,led_i);
    led_i = led_o;
    char string[25];
    itoa((unsigned
int)led_o&0xFF,string,2); //&0xFF是为取led_o的位 转为进制数
    if(i == 6)
        fprintf(stdout,"shift_out= 0%s\n",string); //数据挤高位零
    else
        fprintf(stdout,"shift_out= %s\n",string);
}
}

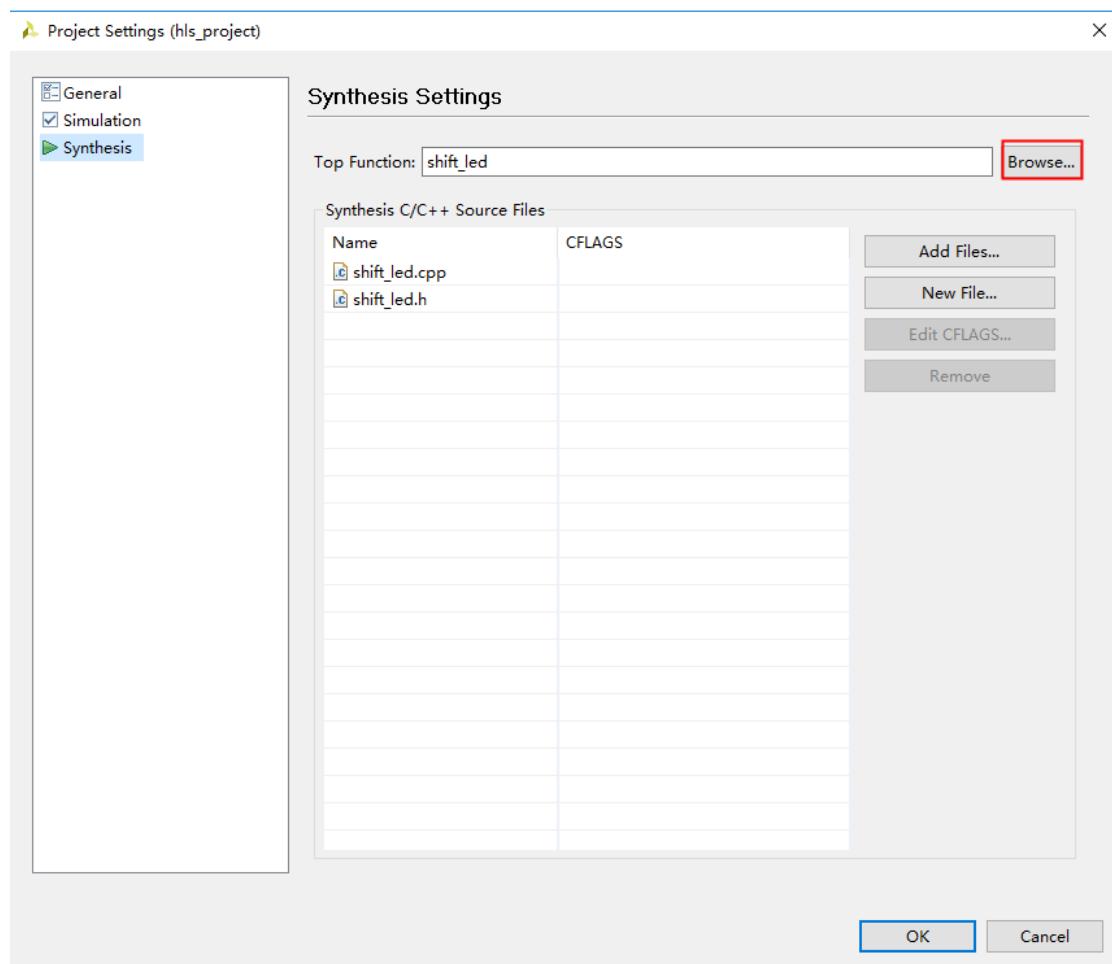
```

2.2.2 代码综合

Step1:点击 Project -> Project Settings 出现下图，在 Synthesis 界面下选择综合的顶层函数名。



Step2: 单击 Browse 指定工程的顶层文件，最后单击 OK 完成修改。



Step3: 因为当前工程中只存在一个 Solution (解决方案)，我们选择 Solution -> Run C Synthesis -> Active Solutions 或者直接点击 ➡ 进行综合，等待一段时间，在未经优化的情况下综合报告如图所示，我们可以看到 FF 和 LUT 分别使用了 93 和 186。

Performance Estimates

- Timing (ns)**
 - Summary**

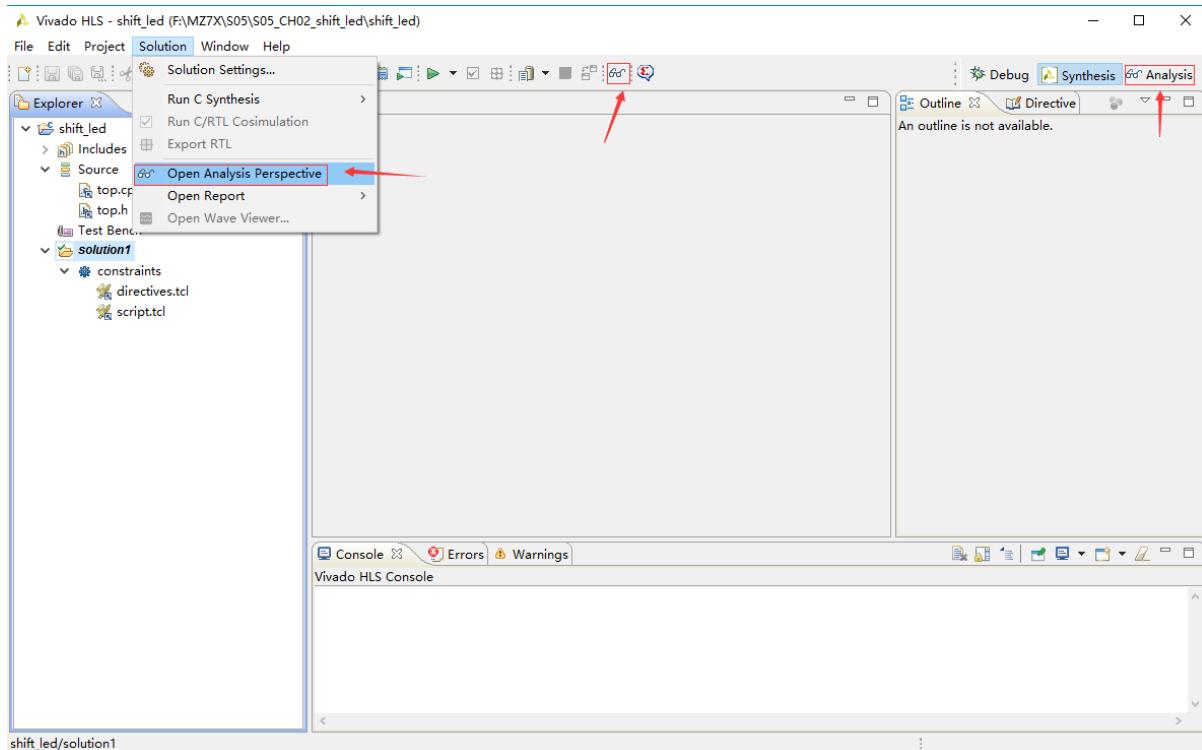
Clock	Target	Estimated	Uncertainty
default	10.00	2.43	1.25
- Latency (clock cycles)**

Utilization Estimates

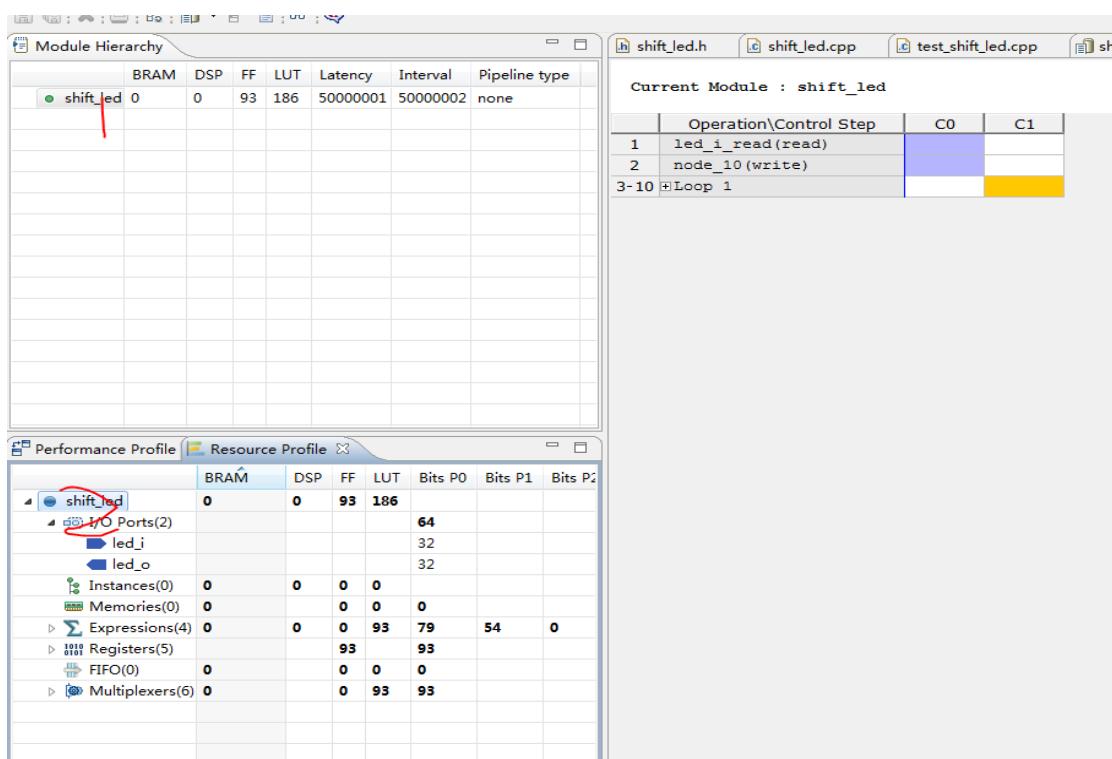
- Summary**

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	93
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	93
Register	-	-	93	-
Total	0	0	93	186
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0
- Detail**
 - Instance**

Step4: 我们单击方框选中的地方点击选择打开分析报告,

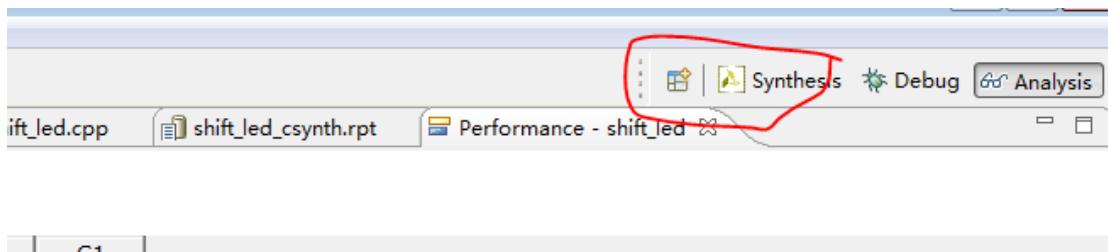


Step5: 在下图所示 1 的地方点击出现 2 所示的 shift_led, 点击展开后可以看到 LED 输入和输出位宽均为 32 位, 我们板载是 4 个 LED, 那么该怎么去进行优化得到我们想要的结果呢?



2.2.3 代码优化

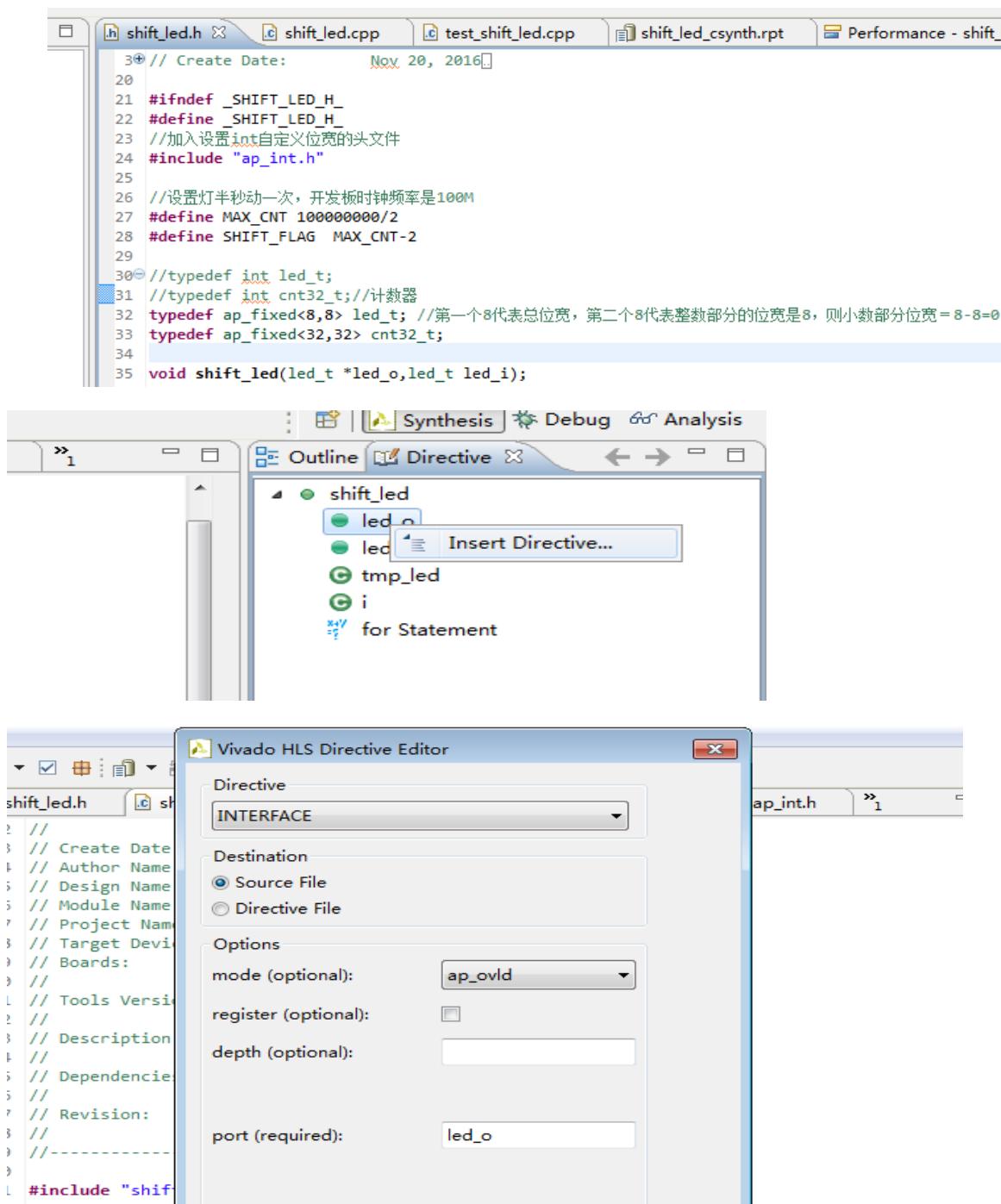
点击 Synthesis 切换到工作空间主界面。



在 shift_led.h 文件中我们包含一个设置 int 自定义位宽的头文件"ap_int.h"，我们使用 ap_fixed() 函数自定义 int 型的 bit 数，其函数原型为：

`class ap_fixed: public ap_fixed_base<_AP_W, _AP_I, true, _AP_Q, _AP_O, _AP_N>,` 那么这个函数怎么使用呢？它的定义如下：

ap_fixed<M, N>, 第一个 M 代表数据总位宽, N 代表数据整数部分的位宽, 那么小数部分的位宽即 M - N; 在下面的代码的第 30 和 31 行我们将它替换为 32 和 33 行所示的代码, 我们即可实现对端口数据位宽的约束, 另外我们再进行端口约束, 约束方法如下, 在需综合的 shift_led.cpp 文件中的 Directive 目录下的 led_o 上右键选择 Insert Directive



因为 led_o 是接口，所以 Directive 我们选择为 INTERFACE, Destination 选择为 Source File，那么有的会问了，这两个有什么区别吗？区别就是 Source File 是针对所有的 Solution 采用同一个优化手段，而 Directive File 是对当前的 Solution 有效，mode (optional) 我们选为 ap_ovld，即输出使能。对 led_i 进行同样的约束，最终效果如下，我们再次综合，对比下进行约束以后的资源利用情况。

```

15 // dependencies:
16 //
17 // Revision: Nov 20, 2016: 1.00 Initial version
18 //
19 //-----
20
21 #include "shift_led.h"
22
23 void shift_led(led_t *led_o, led_t led_i){
24 #pragma HLS INTERFACE ap_ovld port=led_o
25 #pragma HLS INTERFACE ap_vld port=led_i
26

```

Performance Estimates			
<input type="checkbox"/> Timing (ns)			
<input type="checkbox"/> Summary			
Clock	Target	Estimated	Uncertainty
default	10.00	2.43	1.25

Performance Estimates			
<input type="checkbox"/> Timing (ns)			
<input type="checkbox"/> Summary			
Clock	Target	Estimated	Uncertainty
default	10.00	2.43	1.25

Utilization Estimates				
<input type="checkbox"/> Summary				
Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	93
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	93
Register	-	-	93	-
Total	0	0	93	186
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Utilization Estimates				
<input type="checkbox"/> Summary				
Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	93
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	45
Register	-	-	45	-
Total	0	0	45	138
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

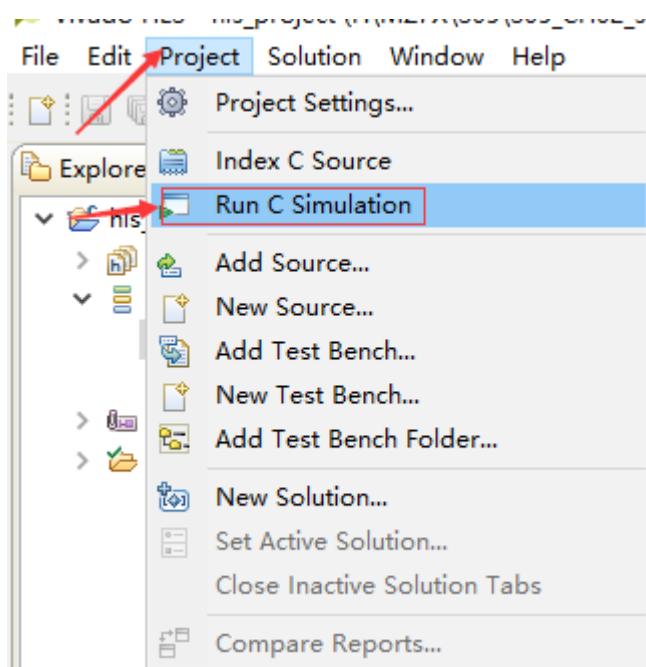
图左：优化前

图右：优化后

对比发现，经过优化以后，资源利用率显著降低，这为以后我们对资源进行优化提供了一个参考思路。

2.2.4 仿真实现

Step1:单击 Project 下的 Run C Simulation 或直接单击  开始 C 仿真。

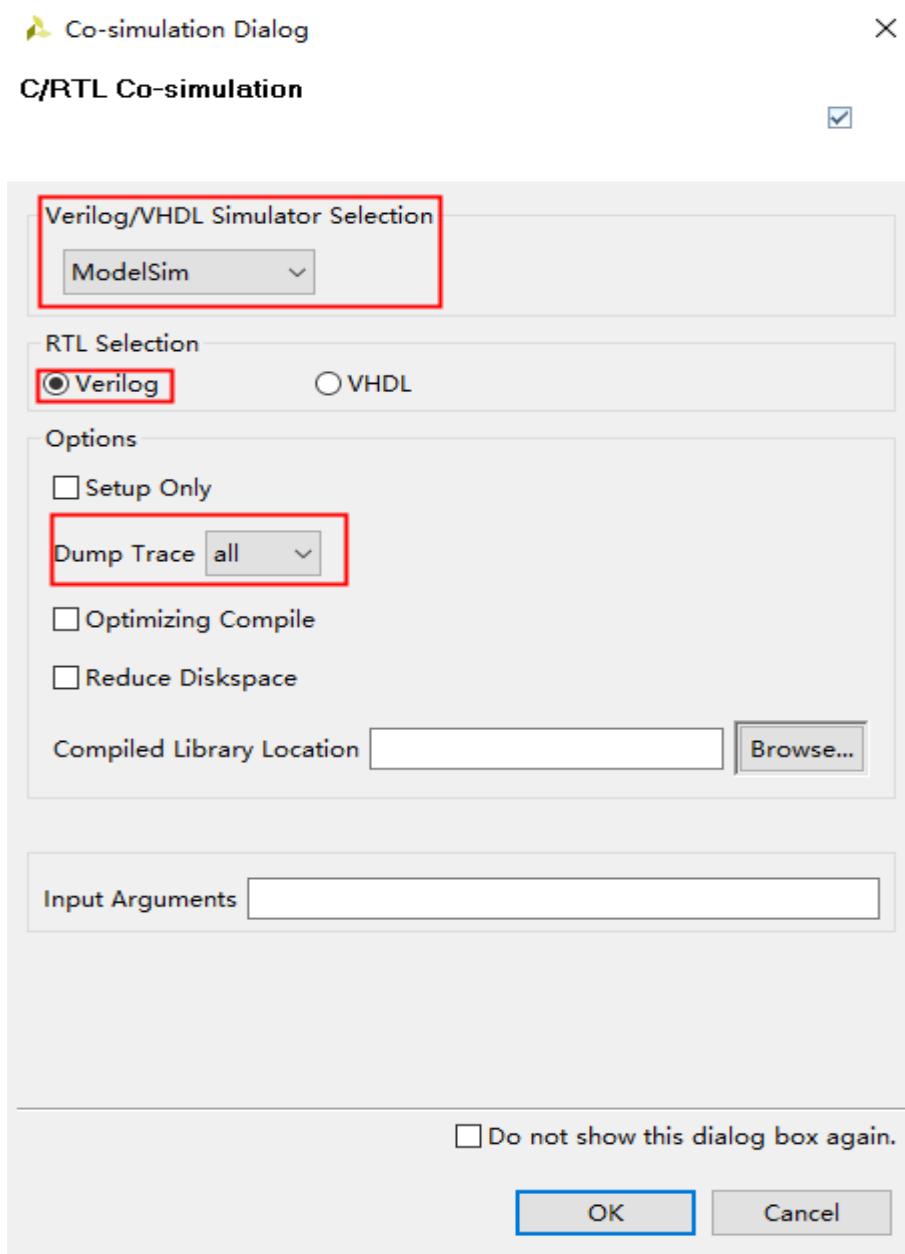


Step2: 等待一段时间，仿真结果如下，我们可以看到数据循环左移了一位，达到了我们想要的实验效果。

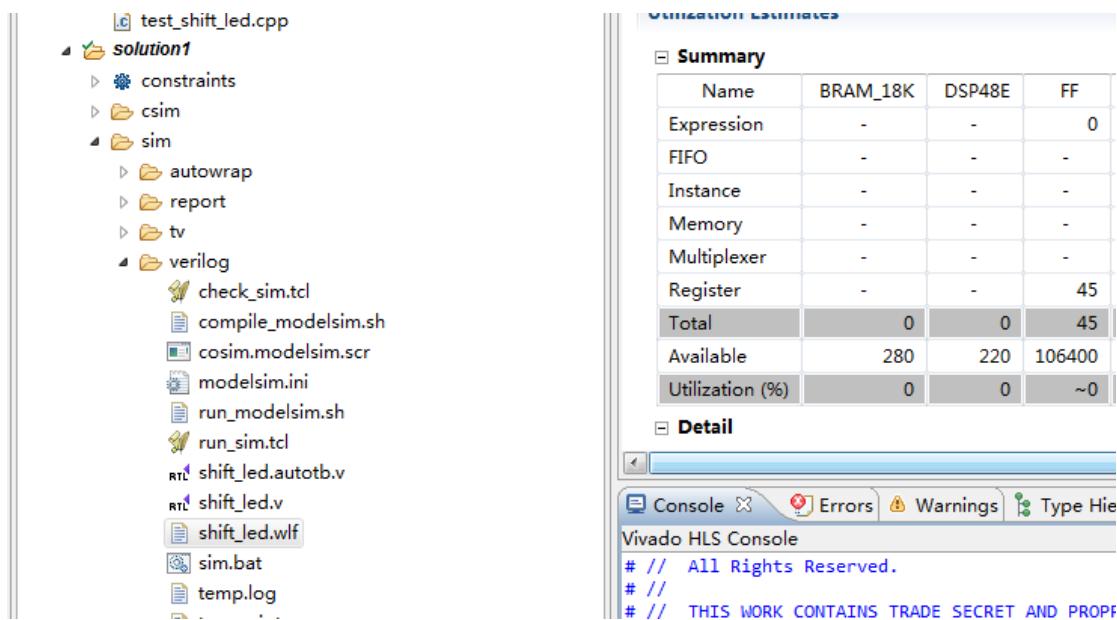
```
Vivado HLS Console
@I [HLS-10] Setting target device to XC7Z020CLG484-1
Compiling ../../src/test_shift_led.cpp in debug mode
Compiling ../../src/shift_led.cpp in debug mode
Generating csim.exe
shift_out= 11111101
shift_out= 11111011
shift_out= 11110111
shift_out= 11101111
shift_out= 11011111
shift_out= 10111111
shift_out= 01111111
shift_out= 11111110
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

The screenshot shows the Vivado HLS Console window. It displays the command-line interface used to run the simulation. The output shows the simulation results for the 'shift_out' variable, which is performing a left shift operation. The first few lines of output are: 'shift_out= 11111101', 'shift_out= 11111011', 'shift_out= 11110111', 'shift_out= 11101111', 'shift_out= 11011111', 'shift_out= 10111111', 'shift_out= 01111111', and 'shift_out= 11111110'. The final lines indicate that the simulation completed successfully with 0 errors and checked in the feature [VIVADO_HLS].

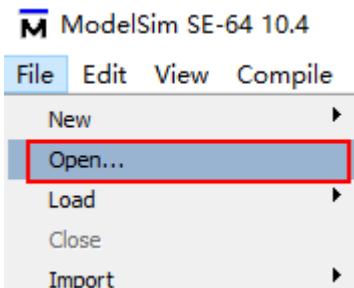
Step3: 单击 Solution 下的 Run C/RTL cosimulation 运行协同仿真。



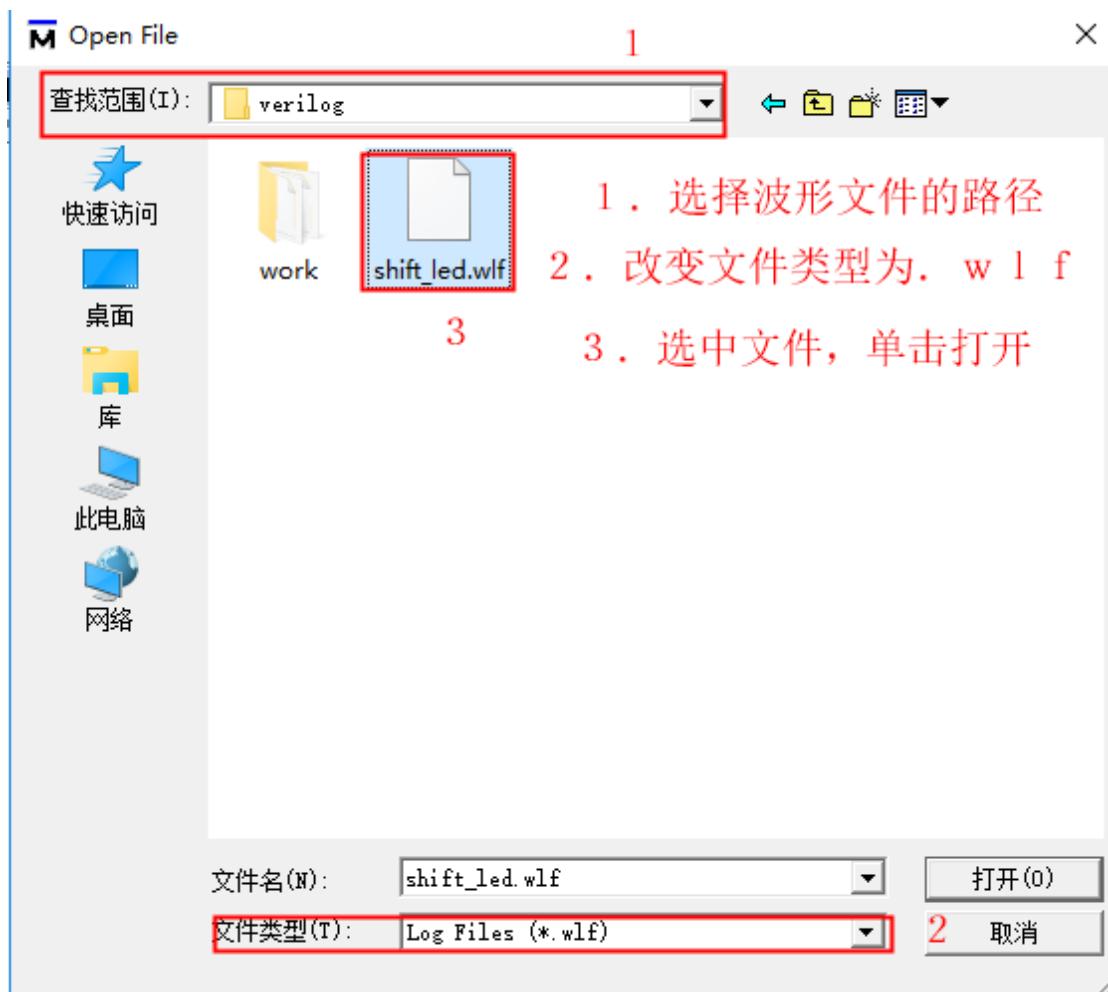
Step4：运行协同仿真一段时间后，我们可以发现在 solution1 目录下多了一个 sim 文件夹，在其 verilog 文件夹下我们可以看到生成的波形文件 shift_led.wlf。



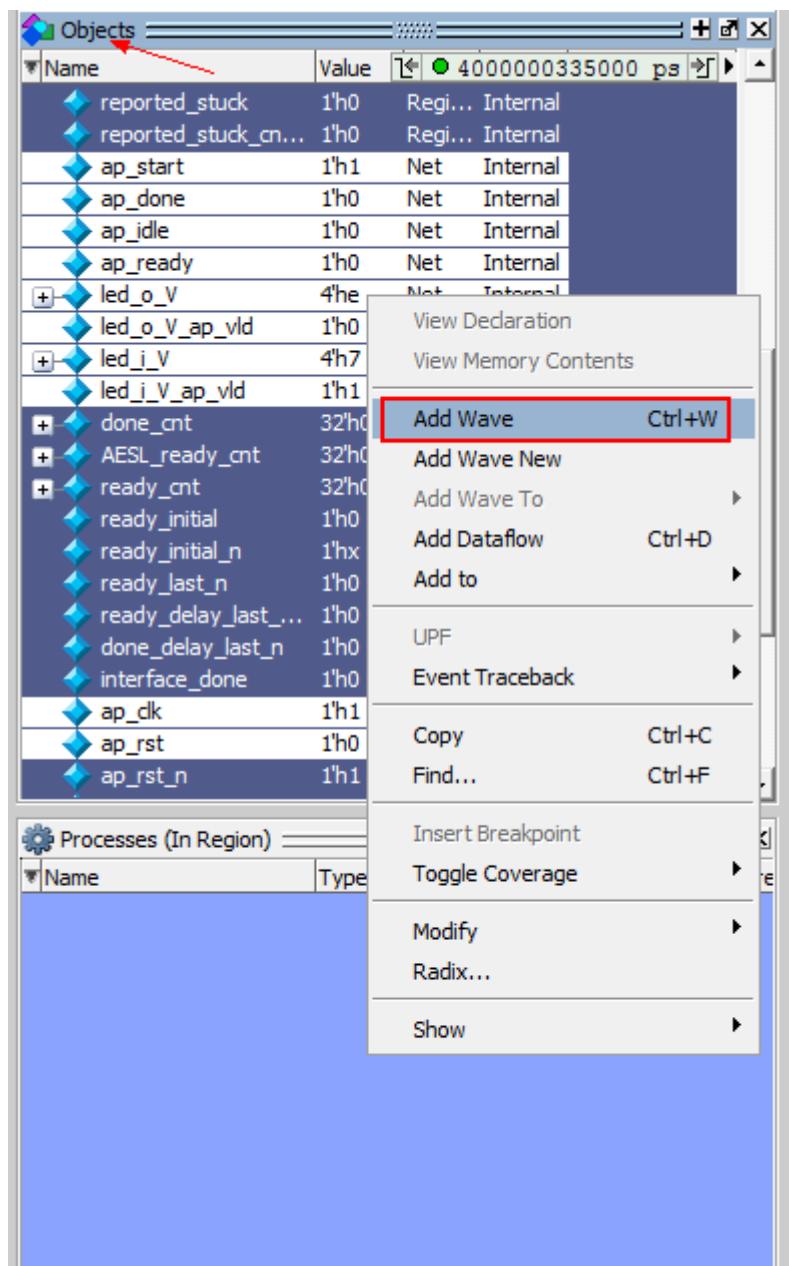
Step5: 通过 modelsim 打开该文件，我们看一下关键接口的时序。首先打开 modelsim, 然后单击 File 菜单下的 open 命令。



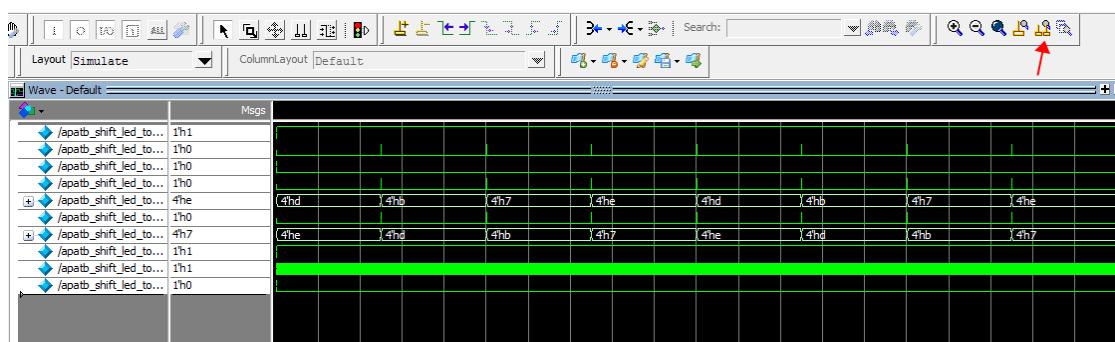
Step6: 打开刚才生成的波形文件 shift_led.wlf 的路径，将文件类型改为*.wlf，然后选中 shift_led.wlf，单击打开按钮。



Step7: 在 objects 设置区，按住 ctrl 键选中要查看波形的信号，然后右单击选择 Add wave 命令。



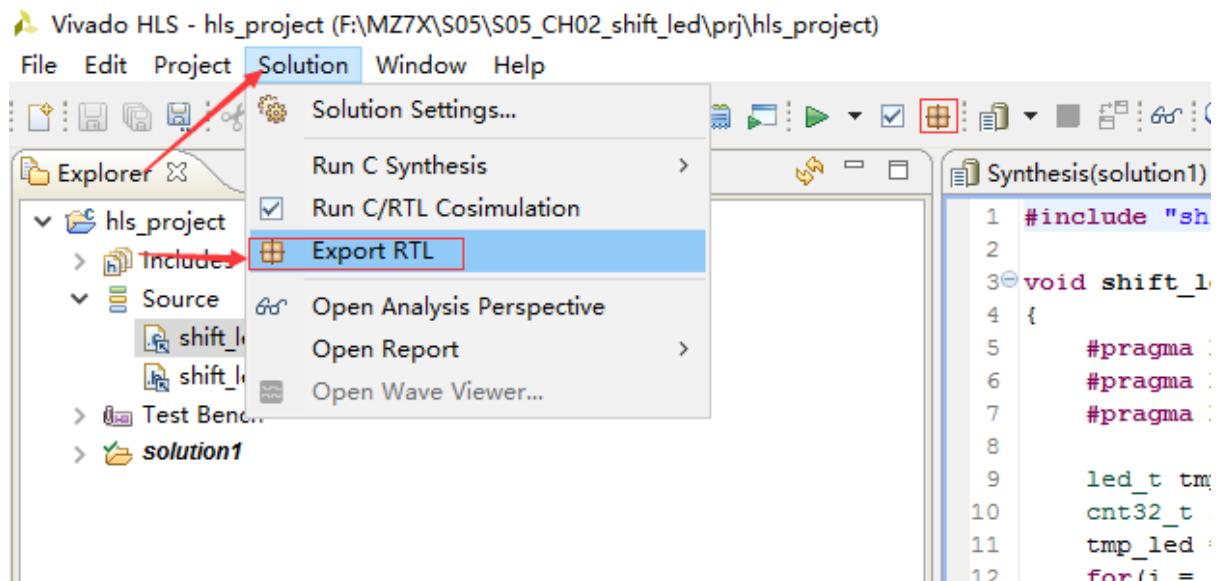
Step8：选中波形显示区，然后单击调整波形，方便查看功能是否正确。



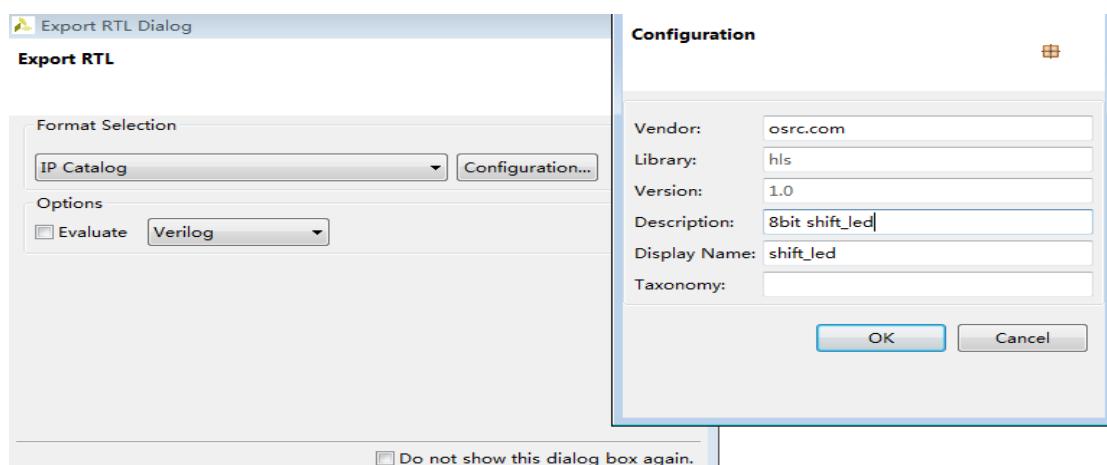
2.3 HLS 代码封装

通过前面的学习，我们了解了 HLS 基本的工程创建，仿真及优化技巧，但是 HLS 只是把你的算法实现从 C 到 RTL 的转化，而不能在硬件平台上进行测试，这一节我们就讲解一下如何把 HLS 工程打包成一个 IP 以便于 Vivado 进行调用。

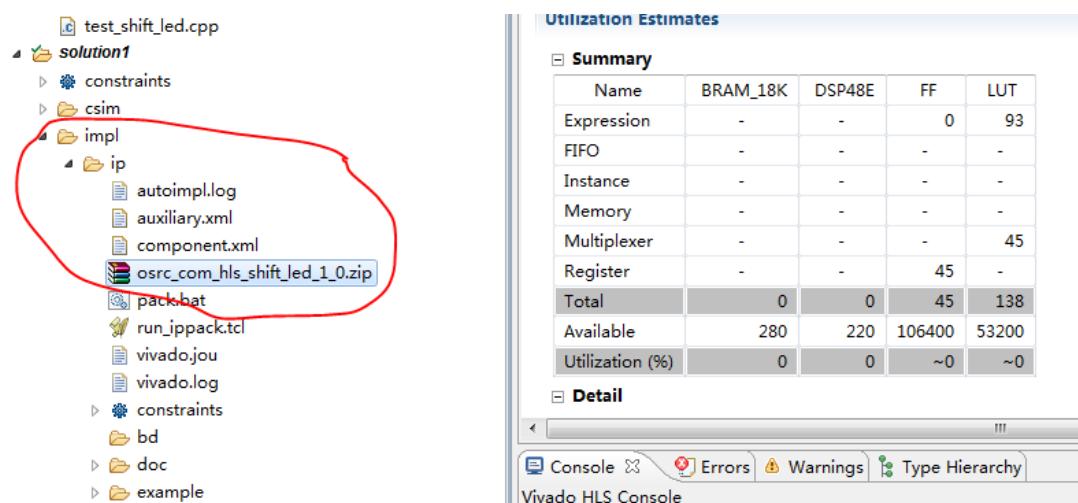
Step1：单击 Solution 菜单下的 Export RTL 或直接单击 导出 RTL 级。



Step2：在弹出的窗口中，点击 configuration 对一些参数进行补充，单击两次 OK。



Step3：等待一段时间后在 solution1 目录下多了一个 impl 文件夹，并且在 ip 文件夹生成了一个压缩包，这就是打包好的 IP，后续我们可以在 Vivado 中进行使用。

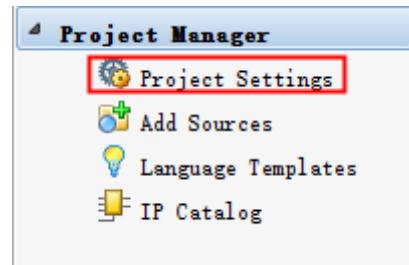


2.4 硬件平台实现

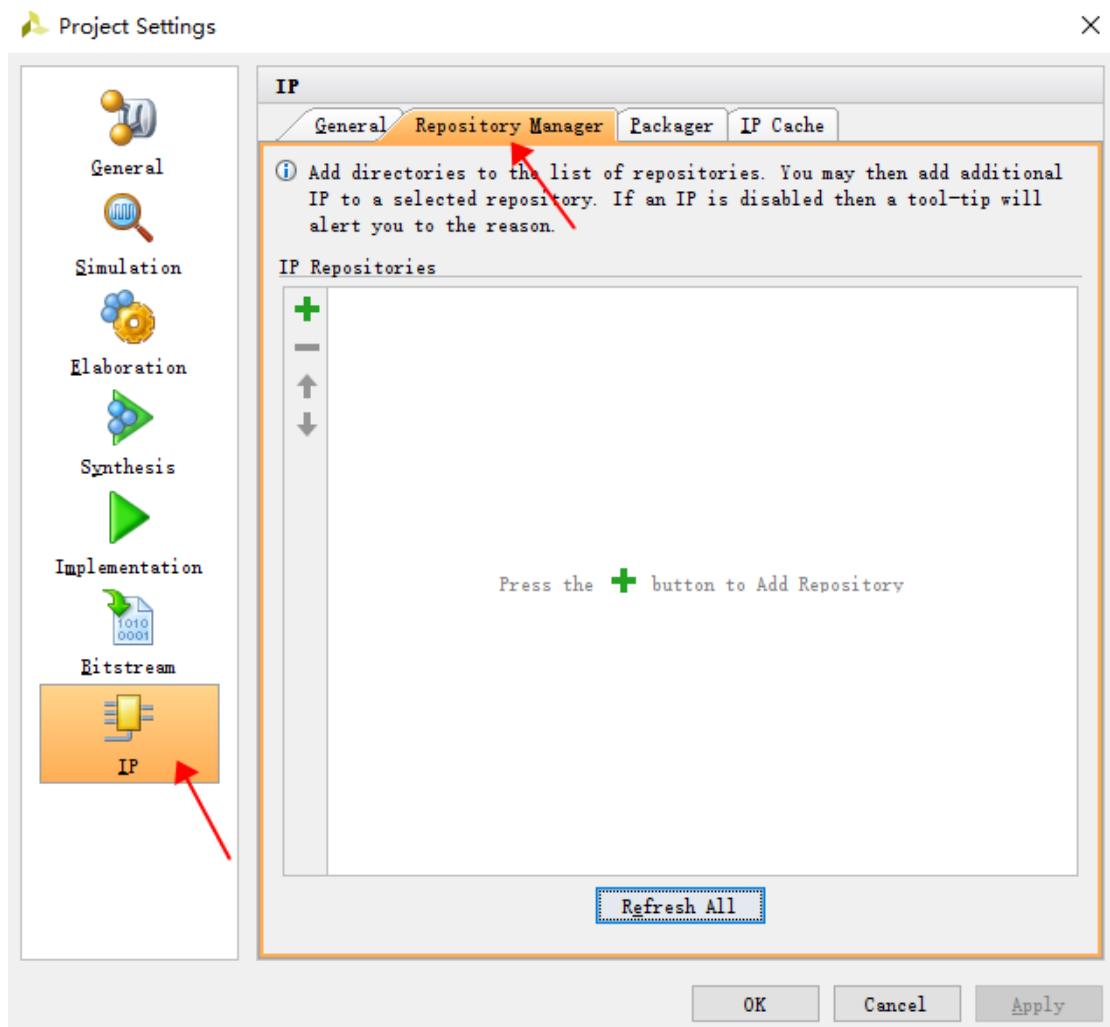
HLS 生成了 IP 之后，接下来的任务就是验证了。将刚才生成的 IP 解压，然后创建一个新的文件夹用来存放 VIVADO 工程。

Step1： 创建一个新的 VIVADO 工程。

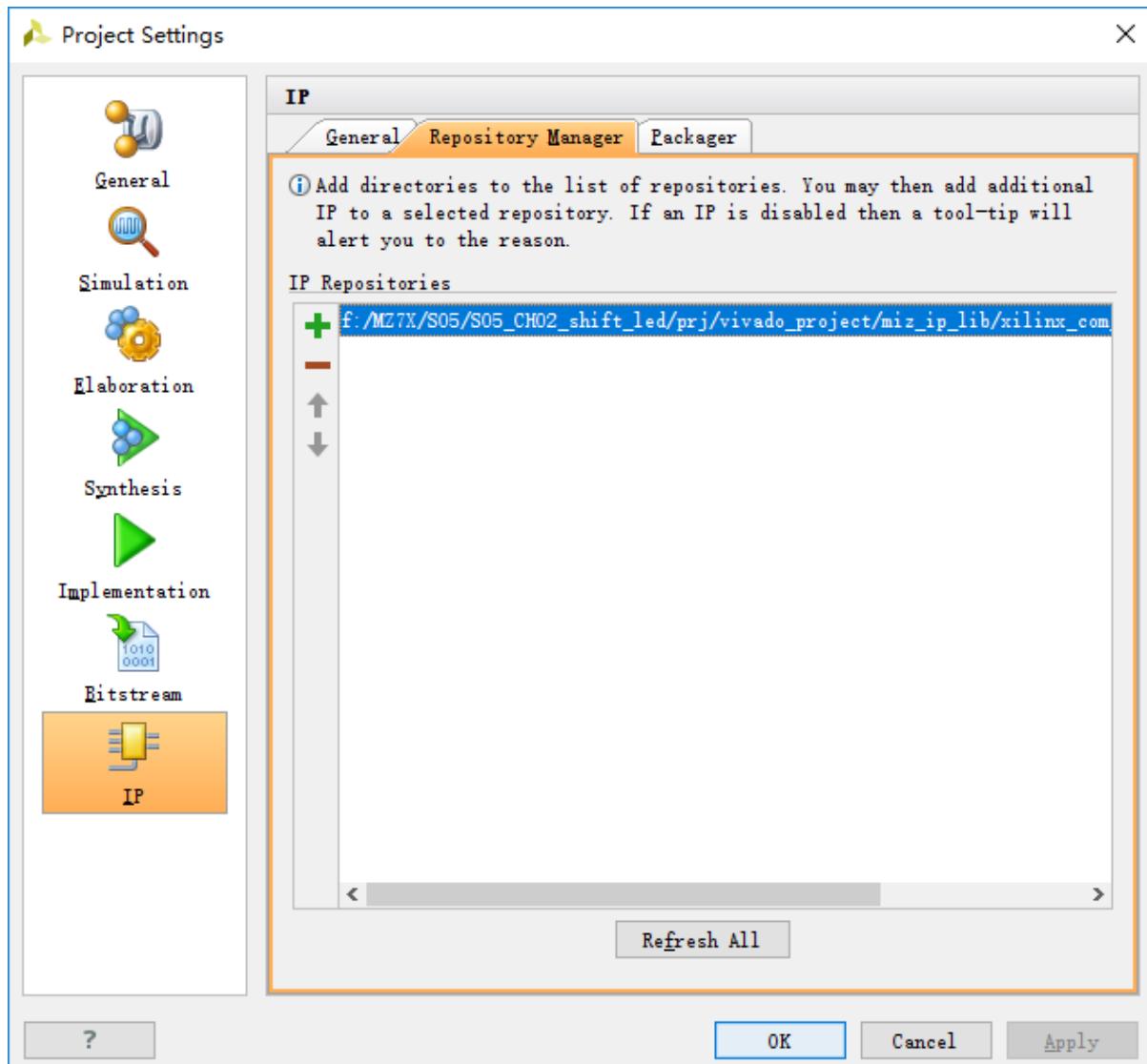
Step2： 在 project Manager 中单击 Project Setting。



Step3： 选择 IP 设置区的 Repository Manager 设置区。



Step4: 单击 将 IP 的路径添加进去，最后单击 OK 完成设置。



Step5: 创建一个名为 shift_led 的 Verilog 文件，并添加如下程序。

```
module shift_led
#(
    parameter DATA_WIDTH = 4 //鑑版堪縮夸縵淪?
)
(
    input                  i_clk,
    input                  i_RST_N,
    output reg [DATA_WIDTH-1:0] led
);

reg [1:0] cnt      ;
reg [DATA_WIDTH-1:0] led_i_V ;
wire ap_start ;
wire led_i_vld;
wire [DATA_WIDTH-1:0] led_o_V ;
```

```
always@(posedge i_clk or negedge i_rst_n)begin
    if(i_rst_n == 1'b0)
        cnt <= 2'd0;
    else if(cnt[1]==1'b0)
        cnt <= cnt + 1'b1;
end

always@(posedge i_clk or negedge i_rst_n)begin
    if(i_rst_n == 1'b0)
        led_i_V <= 2'd0;
    else if(cnt[0]==1'b1)
        led_i_V <= 4'h1;//LED鍊漬 鑄?
    else if(led_o_vld == 1'b1)
        led_i_V <= led_o_V;
end

always@(posedge i_clk or negedge i_rst_n)begin
    if(i_rst_n == 1'b0)
        led <= 1'b0;
    else if(led_o_vld == 1'b1)
        led <= led_o_V;
end

assign ap_start  = cnt[1];
assign led_i_vld = cnt[1];

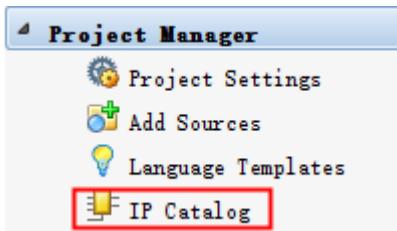
shift_led_0 u_shift_led_0(
    .led_o_V_ap_vld      (led_o_vld),//    output    wire
    led_o_vld
    .led_i_V_ap_vld      (led_i_vld),// input wire led_i_vld
    .ap_clk               (i_clk      ),// input wire ap_clk
    .ap_rst               (^i_rst_n ),// input wire ap_rst
    .ap_start              (ap_start   ),// input wire ap_start
    .ap_done               (          ),// output wire ap_done
    .ap_idle               (          ),// output wire ap_idle
    .ap_ready              (          ),// output wire ap_ready
    .led_i_V               (led_i_V  ),// output wire [7 : 0]
    led_o_V
    .led_o_V               (led_o_V  ) // input wire [7 : 0]
    led_i_V
);

```

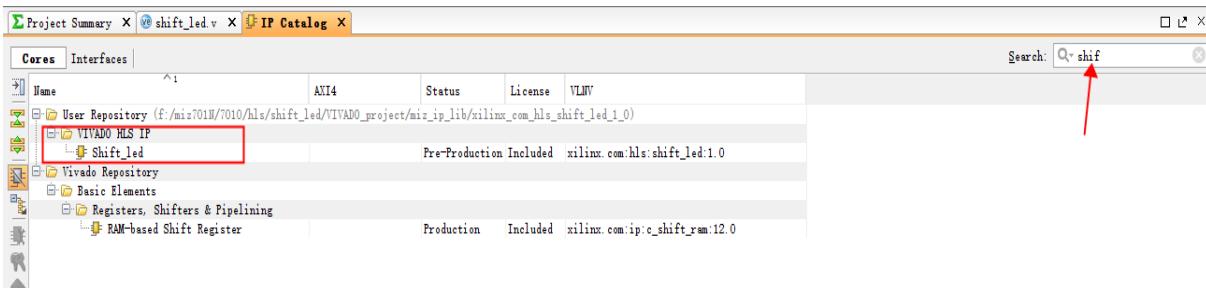
```
endmodule
```

这里例化的 shift_led_0 就是我们刚刚用 HLS 创建的 IP，这个程序的编写也是依据其时序来编写的。

Step6: 单击 Project Manager 中的 IP catalog。



Step7: 输入生成的 IP 的名字，将其添加到工程中来。

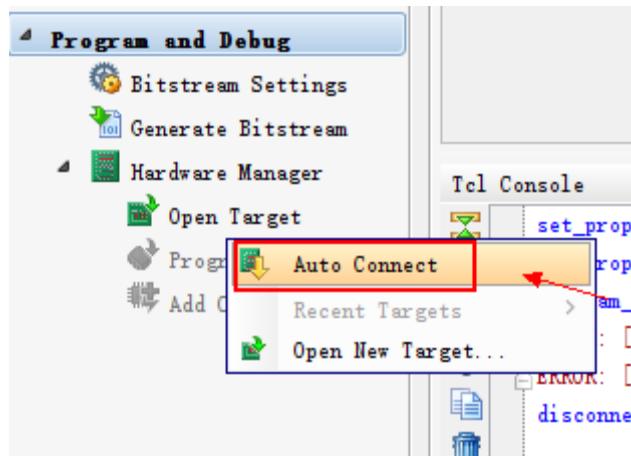


Step8: 在我们提供的本章源程序文件夹下的 DOC 文件夹下找到 XDC 文件夹，将其中的约束文件添加到 vivado 工程当中来。

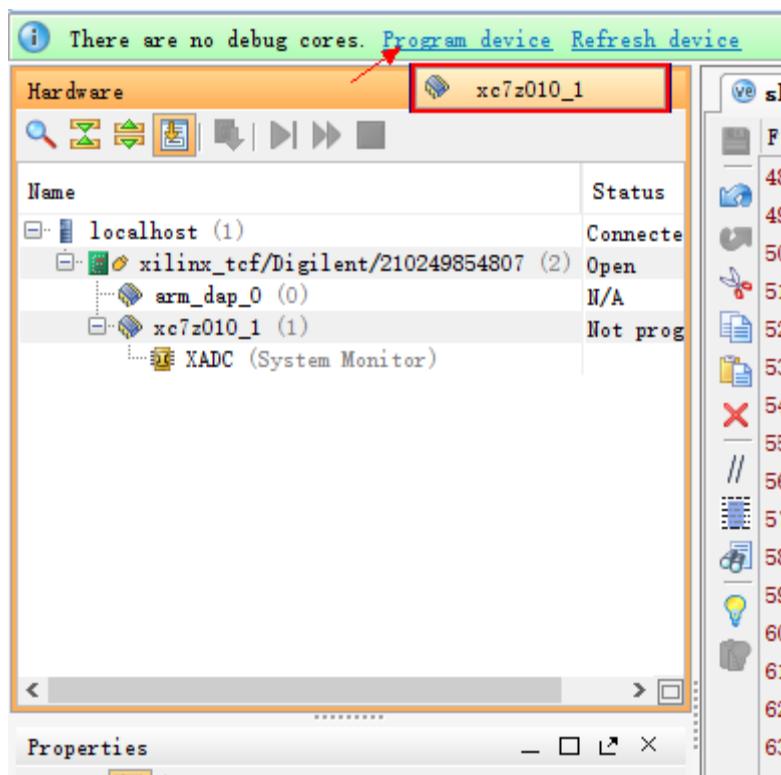
Step9: 单击 生成 bit 文件。



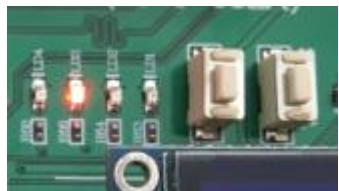
Step10: bit 文件生成完后，打开开发板电源，在 Program and Debug 设置区中单击 Hardware Manager 左边的三角形，选择 Open Target。



Step11: 下载程序到开发板中。



将程序进行编译下载后，我们可以看到板载 LED 每隔半秒向左移一位循环往复，结果如下图：



2.5 本章小结

本章通过一个简单的流水灯程序，向大家介绍了 HLS 的工程创建过程，以及代码的仿真方法，代码的简单优化等等。通过本章，大家需要掌握 HLS 的基本开发流程，为下面的课程打下基础。

S05_CH03_ImageLoad 实验

3.1 概述

通过前面的学习，我们已经基本熟悉了 HLS 的开发流程，那么在 HLS 上进行算法开发，算法实现综合后我们需要对算法的正确性进行验证，那么我们如何得到我们想要的视频图像数据流，对算法模块进行测试呢？在这一章我们就这些问题讨论一下如何搭建验证平台，主要包括图像、视频数据流以及外部摄像头的调用，方便对算法模块进行测试。

3.2 图片数据的获取

当我们进行前期算法验证的时候，需要读取图片进行仿真，那么关键的一步就是如何加载图片进行测试，在 HLS 中比较基础的两种加载图片的方法如下：

通过 cvLoadImage 函数加载图片，其格式如下：

```
IplImage* src = cvLoadImage(INPUT_IMAGE);
```

```
cvShowImage("src", src);
```

函数原型： IplImage* cvLoadImage(const char*filename, int iscolor

CV_DEFAULT(CV_LOAD_IMAGE_COLOR));

filename : 要被读入的文件的文件名(包括后缀)；

iscolor: 指定读入图像的颜色和深度；

指定的颜色可以将输入的图片转为 3 信道(CV_LOAD_IMAGE_COLOR)，单信道

(CV_LOAD_IMAGE_GRAYSCALE)，或者保持不变(CV_LOAD_IMAGEANYCOLOR)。

cvLoadImage 函数使用方法有下面几种：

```
cvLoadImage( filename, -1 ); 默认读取图像的原通道数
```

```
cvLoadImage( filename, 0 ); 强制转化读取图像为灰度图
```

```
cvLoadImage( filename, 1 ); 读取彩色图
```

我们在这里演示一个通过 cvLoadImage 函数读取图片显示的例子，源码及结果如下图：

```
IplImage* src = cvLoadImage(INPUT_IMAGE);
```

```
IplImage* dst = cvCreateImage(cvGetSize(src), src->depth, src->nChannels); // 获取原始图像大小
```

```
AXI_STREAM src_axi, dst_axi;
```

```
IplImage2AXIVideo(src, src_axi);
AXIVideo2IplImage(src_axi, dst);
cvSaveImage(OUTPUT_IMAGE, dst);
cvShowImage( "result_1080p", dst);
```



通过imread函数读取图片，格式如下：

```
Mat src_rgb = imread(INPUT_IMAGE);

IplImage src = src_rgb;

cvShowImage("src", &src);
```

首先，我们看 imread 函数，可以在官方文档中查到其原型如下：

```
CV_EXPORTS_W Mat imread( const string& filename, int flags=1 );
```

第一个参数，const string&类型的 filename，填我们需要载入的图片路径名。

第二个参数，int 类型的 flags，为载入标识，它指定一个加载图像的颜色类型。可以看到它自带缺省值 1. 所以有时候这个参数在调用时我们可以忽略，在看了下面的讲解之后，我们就会发现，如果在调用时忽略这个参数，就表示载入三通道的彩色图像。通过转到定义，我们可以在 highgui_c.h (192-204 行) 中发现这个枚举的定义是这样的：

```
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_IMAGEANYCOLOR   =4  
};
```

我们通过 imread 函数读取一副图片并且灰度显示的代码如下：

```
Mat src_rgb = imread(INPUT_IMAGE, CV_LOAD_IMAGE_GRAYSCALE); //加载图片并灰度显示  
IplImage src = src_rgb;  
cvSaveImage(OUTPUT_IMAGE, &src);  
cvShowImage("src", &src);
```



3.3 视频流文件的载入

cvCaptureFromAVI 函数进行视频文件的载入，

格式： cvCaptureFromAVI("AVI 文件名称");

功能：用来播放 AVI 文件视频；我们在 [highgui_c.h](#) 文件可以看到有如下定义：

```
#define cvCaptureFromFile cvCreateFileCapture
```

```
#define cvCaptureFromAVI cvCaptureFromFile
```

说明：所以用 cvCaptureFromAVI() 跟 cvCaptureFromFile(), cvCreateFileCapture() 都是一样的作用；文件的类型不一定必须是 AVI 格式，只要文件符合 OpenCV 支持的格式就能播放。

格式： int cvGrabFrame(CvCapture 结构体)；

功能：将 capture 抓下來的相片放在 OpenCV 中；其与 cvQueryFrame() 是相同的步骤；

cvGrabFrame() 返回值为 0 或 1；0 是失败，1 是成功。

格式：cvRetrieveFrame(CvCapture 结构)；

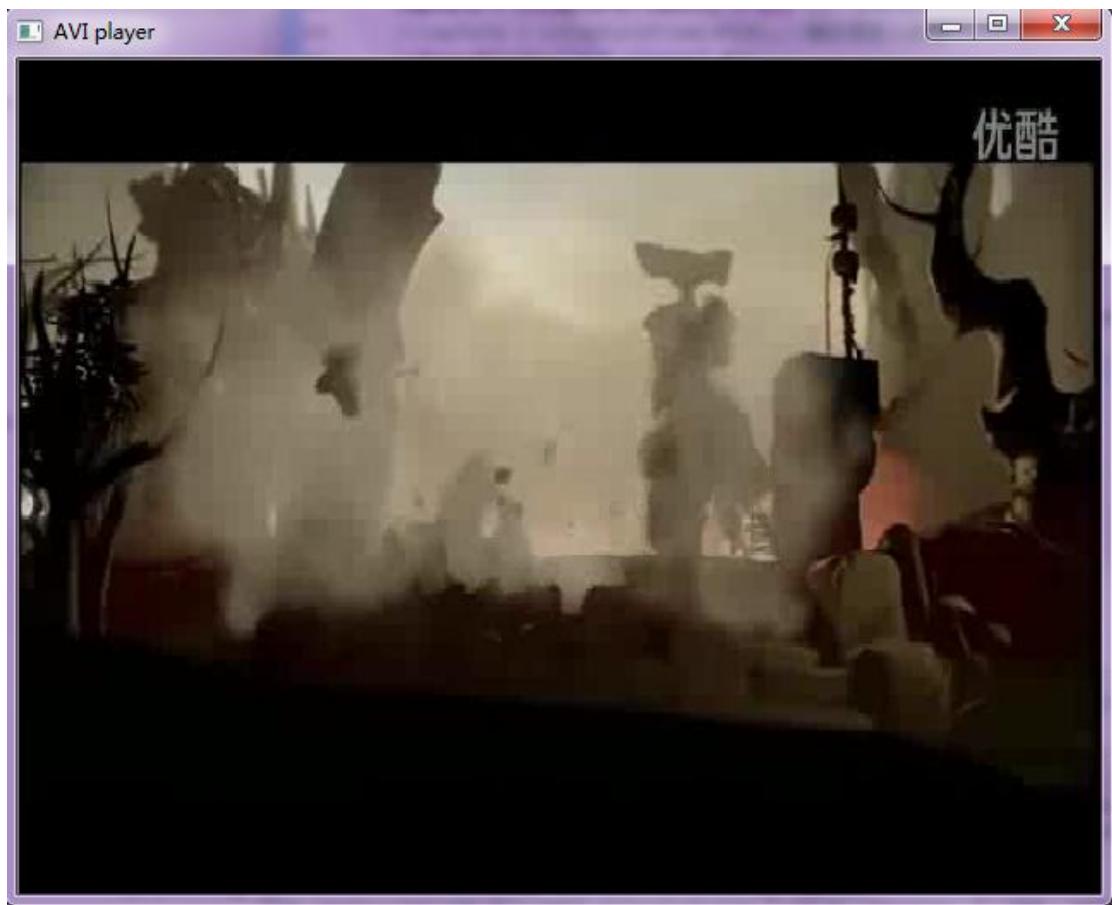
功能：从 OpenCV 快取中得到 Frame，并配置给 IplImage 结构体；其中：

cvQueryFrame()=cvGrabFrame() +cvRetrieveFrame().

我们通过一个实例让大家理解这几个函数的使用

```
IplImage *frame;  
CvCapture *capture = cvCaptureFromAVI("1.avi");//获取视频数据  
cvNamedWindow("AVI player", 0);  
while(true)  
{  
    if(cvGrabFrame(capture))  
    {  
        frame = cvRetrieveFrame(capture);  
        cvShowImage("AVI player", frame);  
        if(cvWaitKey(10)>=0) break;  
    }  
}
```

通过运行仿真，我们截取的视频画面如下：



3.4 外部摄像头的调用

```
CvCapture*cvCaptureFromCAM( int index );
```

参数: index

要使用的摄像头索引。释放这个结构, 使用函数 cvReleaseCapture。

要将视频写入文件中, 使用 cvWriteFrame 写入一帧到一个视频文件中

```
int cvWriteFrame( CvVideoWriter* writer, const IplImage* image );
```

通过摄像头捕获视频数据的关键代码如下, 并且通过调用外部 USB 摄像头成功采集到视频数据, 为后期算法验证提供了测试依据。

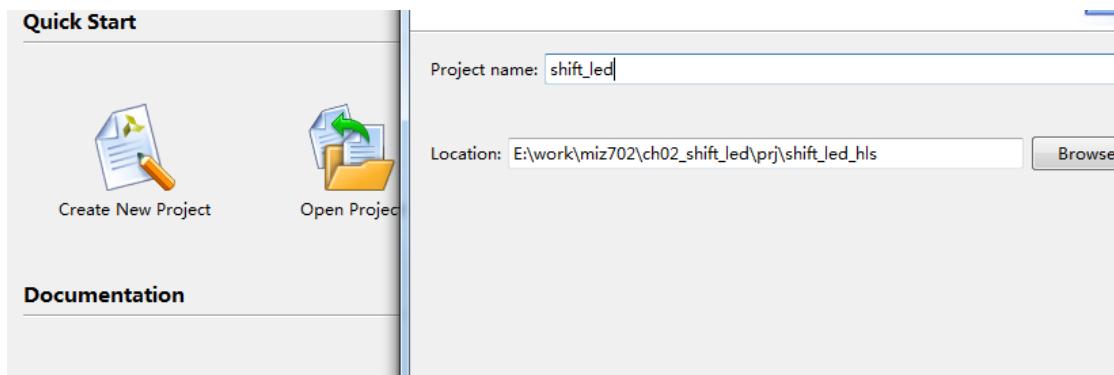
```
IplImage *frame;  
CvCapture *capture = cvCaptureFromCAM(1); //捕获摄像头数据0--笔记本自带摄像头 1--外部摄像头
```



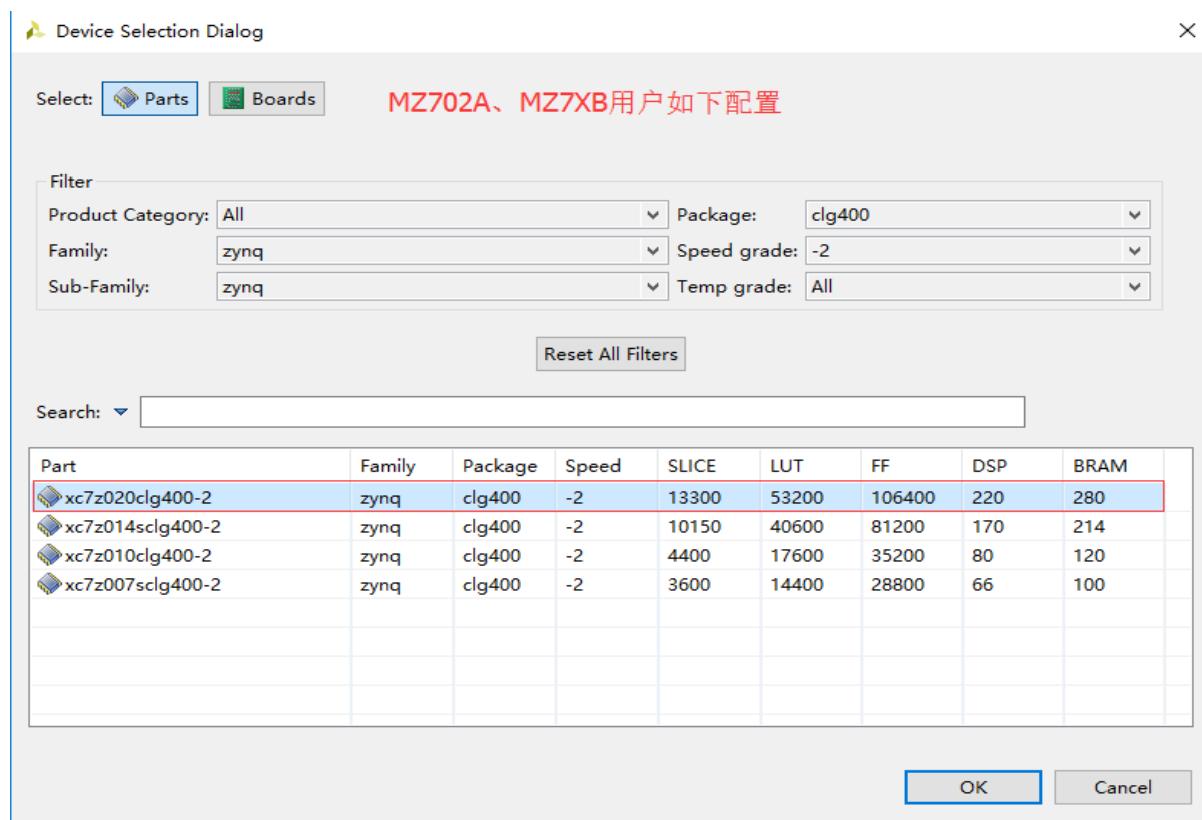
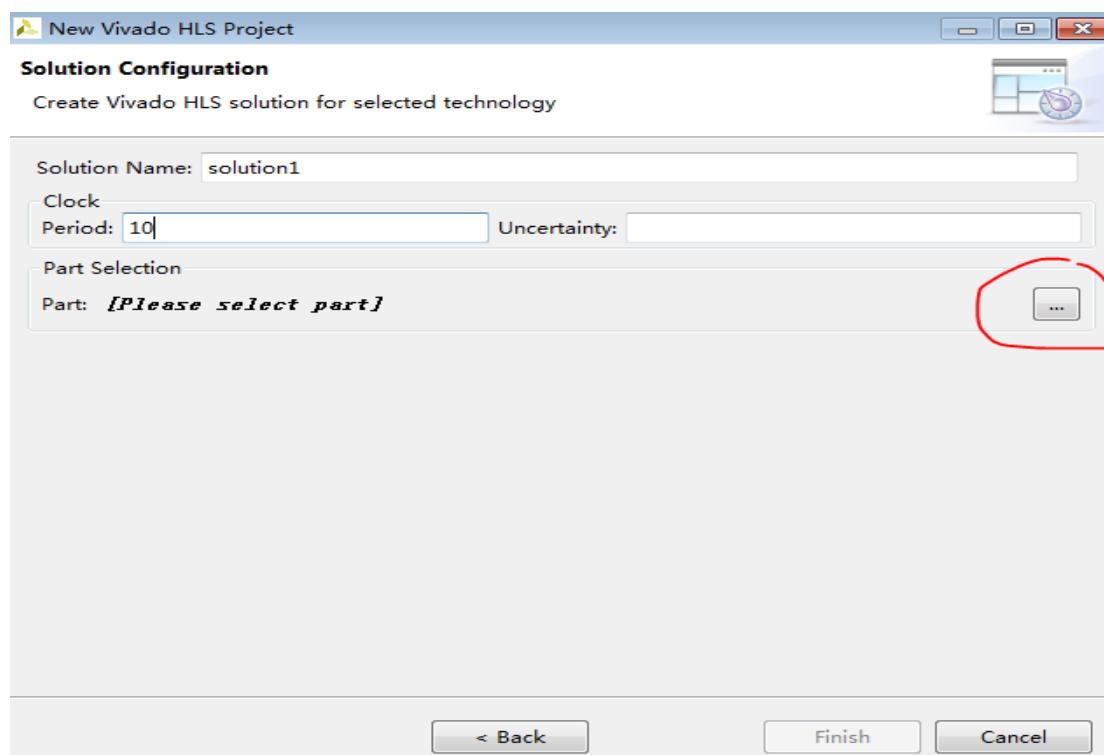
3.5 工程创建与验证

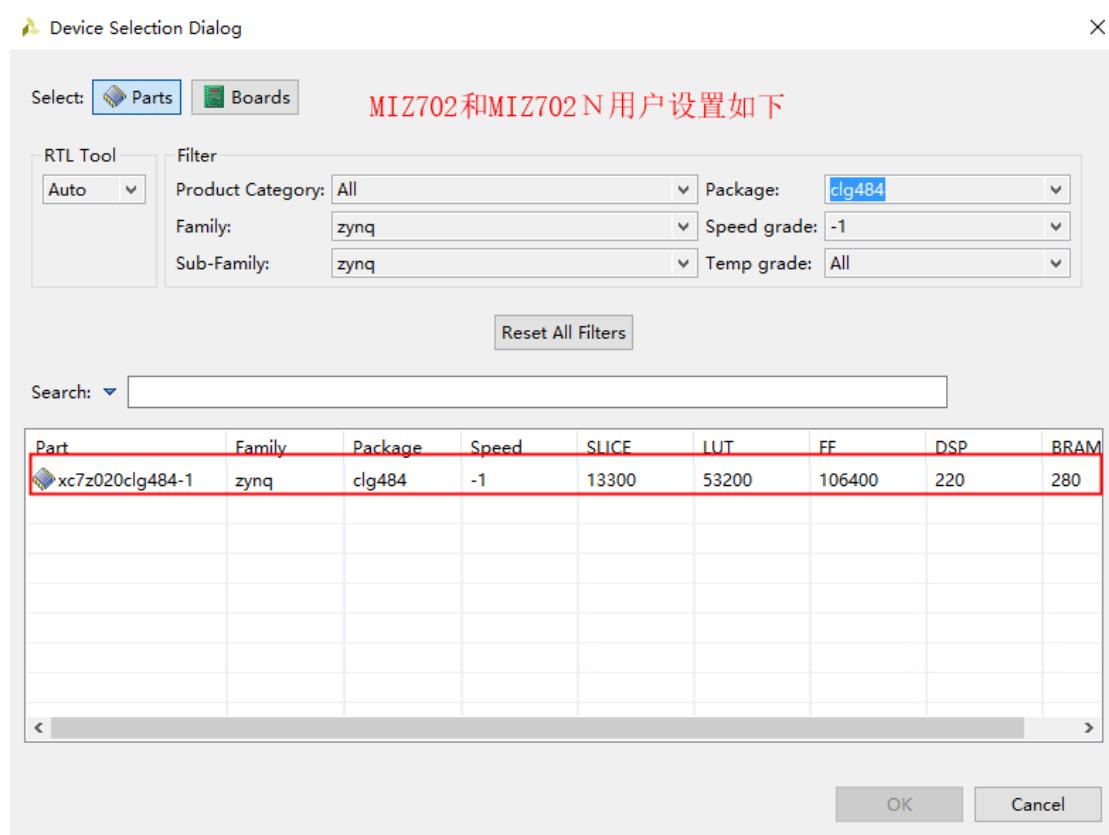
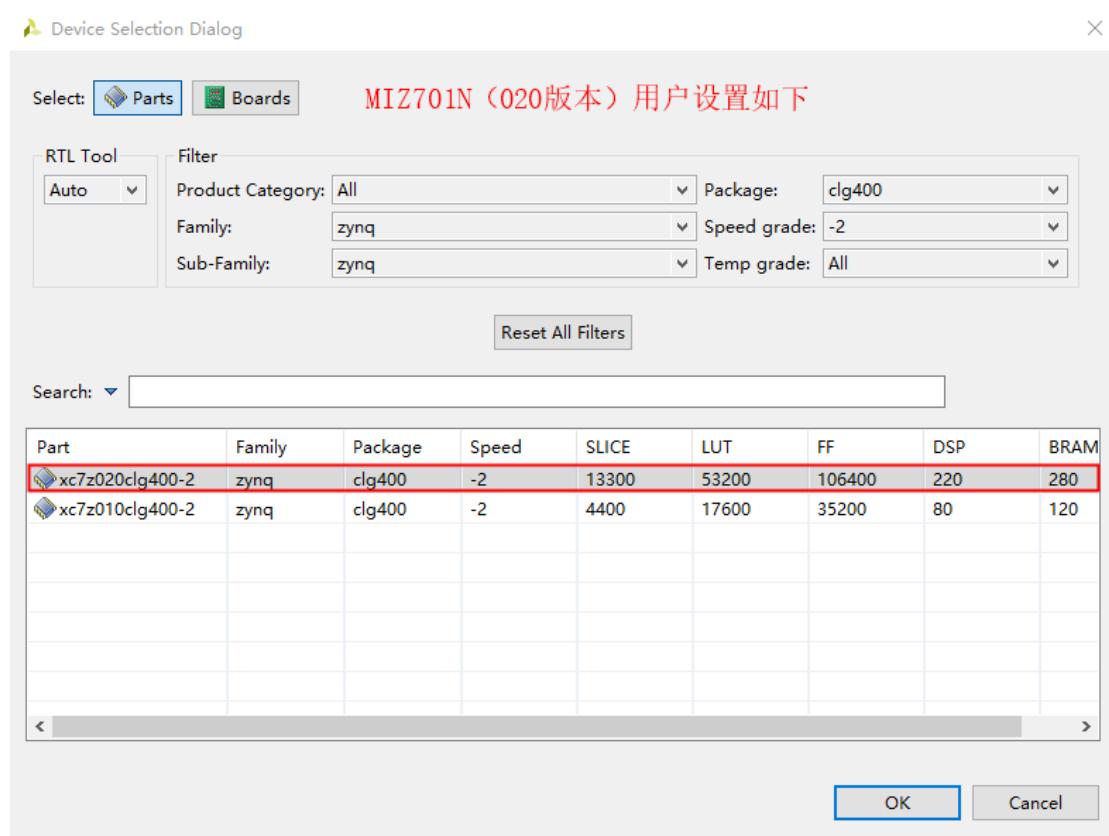
上面介绍完了仿真数据流的获取方法之后，接下来我们创建一个工程对其进行验证。

Step1: 打开 Vivado HLS 开发工具，单击 Create New Project 创建一个新工程，设置好工程路径和工程名，一直点击 Next 按照默认设置。

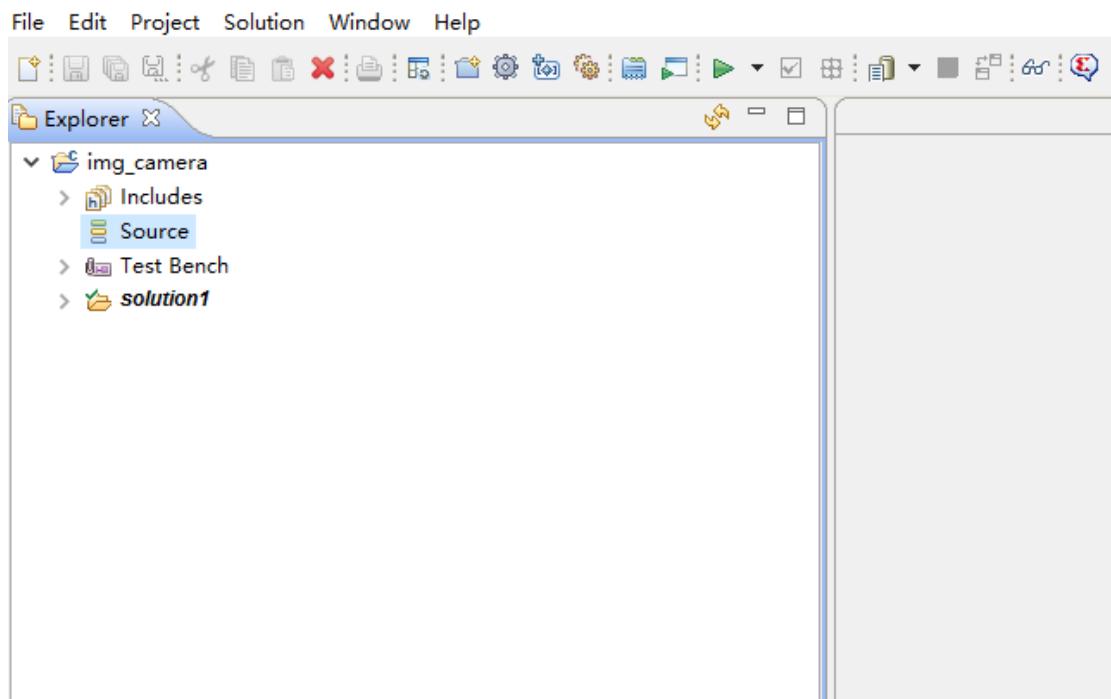


Step2: 出现如下图所示界面，时钟周期 Clock Period 按照默认 10ns，Uncertainty 和 Solution Name 均按照默认设置，点击红色圆圈部分选择芯片类型，然后点击 OK。





Step3: 点击 Finish, 出现如下界面:



Step4：与上一章不同的是，本章只要进行仿真，因此直接单击 Test Bench，添加一个名为 Test.cpp 的测试文件，并添加如下程序。

```
#include "hls_opencv.h"

using namespace cv;

#define INPUT_IMAGE           "test_1080p.bmp"
#define OUTPUT_IMAGE          "result_1080p.bmp"

int main (int argc, char** argv) {

    //方法 cvLoadImage函数载图片
    IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels); //获得原图像大小

    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIvideo(src, src_axi);
    //image_filter(src_axi, dst_axi, src->height, src->width);
    AXIvideo2IplImage(src_axi, dst);

    cvSaveImage(OUTPUT_IMAGE, dst);
    cvShowImage( "result_1080p",dst);
}
```

```
cvReleaseImage(&src);

cvWaitKey();

/*
//方法 cvLoadImage函数截图片
Mat src_rgb =
imread(INPUT_IMAGE,CV_LOAD_IMAGE_GRAYSCALE); //加载图片预览显示

IplImage src = src_rgb;
cvSaveImage(OUTPUT_IMAGE, &src);
cvShowImage("src",&src);
waitKey(0);
return 0;
*/



/*
//读视频文件
IplImage *frame;
CvCapture *capture = cvCaptureFromAVI("1.avi"); //读取数据
cvNamedWindow("AVI player",0);
while(true)
{
    if(cvGrabFrame(capture))
    {
        frame = cvRetrieveFrame(capture);
        cvShowImage("AVI player",frame);
        if(cvWaitKey(10)>=0) break;
    }
    else
    {
        break;
    }
}
cvReleaseCapture(&capture);
cvDestroyWindow("AVI player");

return 0;
*/



/*
//摄影操作
*/
```

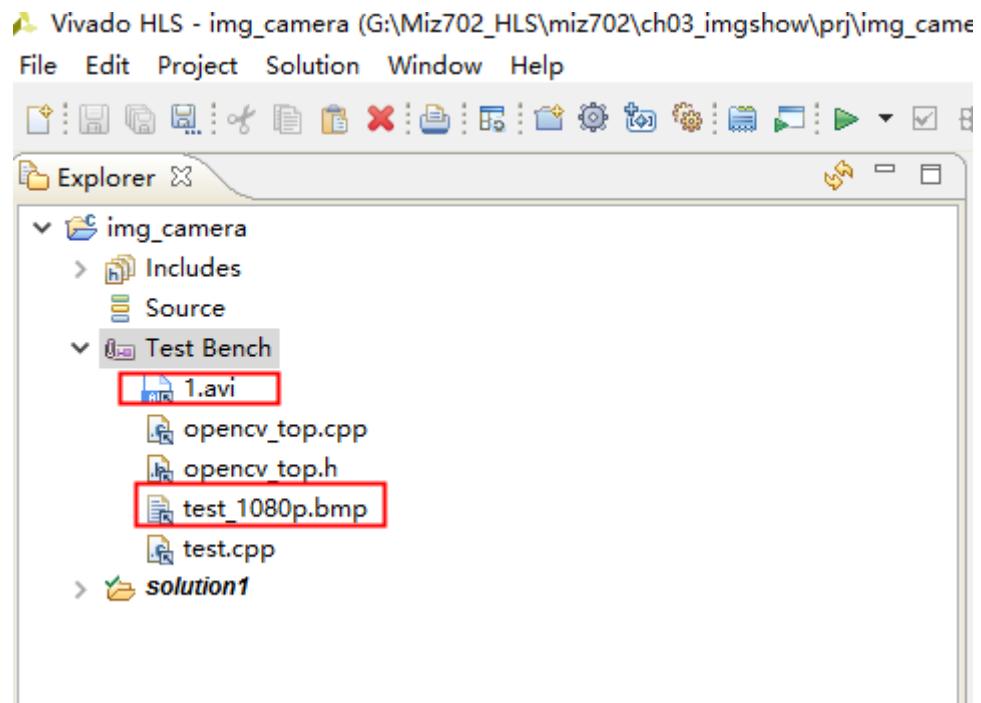
```
IplImage *frame;
CvCapture *capture = cvCaptureFromCAM(1); //捕获数据--笔记本摄像头
1--外接摄像头
cvNamedWindow("AVI player", 0);
while(true)
{
    if(cvGrabFrame(capture))
    {
        frame = cvRetrieveFrame(capture);

        cvShowImage("AVI player", frame);
        if(cvWaitKey(10)>=0) break;
    }
    else
    {
        break; //没有采集到数据退出
    }
}
cvReleaseCapture(&capture);
cvDestroyWindow("AVI player");

return 0;
*/
```

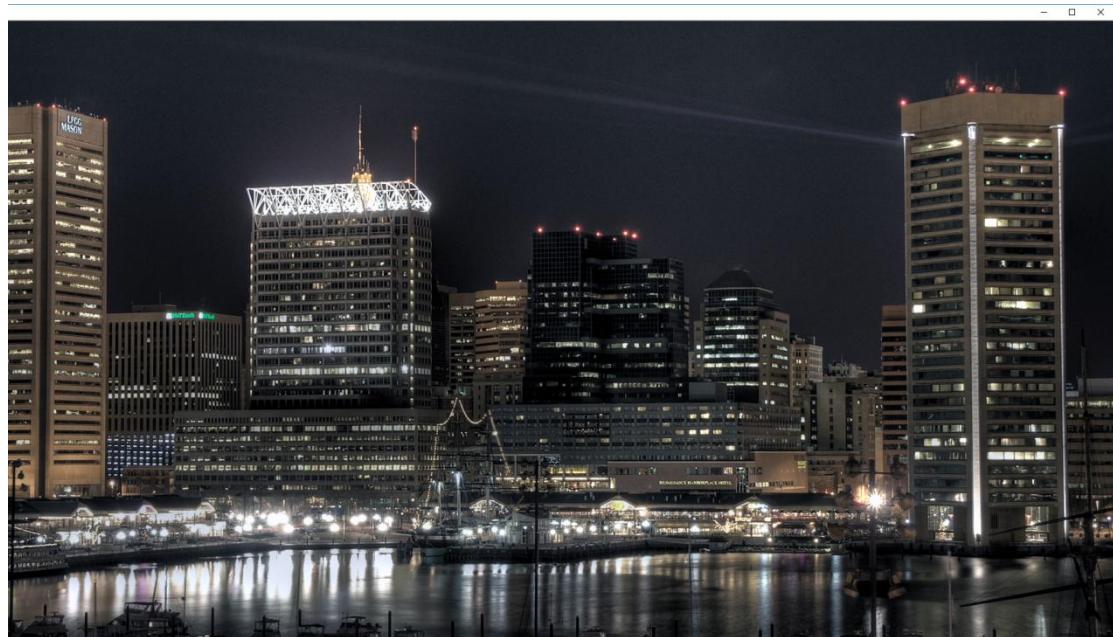
在这个程序中，我们把前 3 节的程序都写到了同一个函数当中，在程序中已经给出了注释，在仿真过程中可以将不必要的代码通过注释的方法屏蔽，分别对这三个部分进行测试。

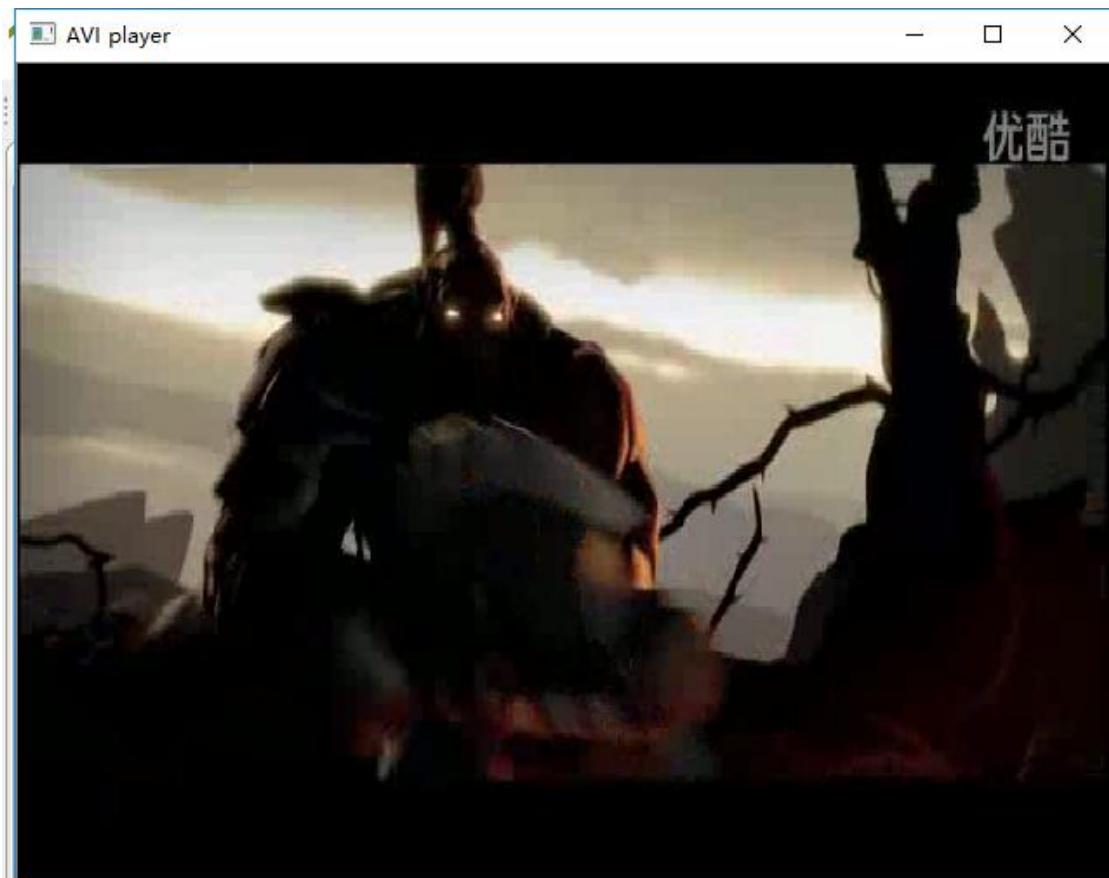
Step5：将要测试的文件添加进 Test Bench 中（文件在我们提供的源程序文件夹中的 image 文件夹中可以找到），添加方法是选中 Test Bench 右单击然后选择 Add File 命令。

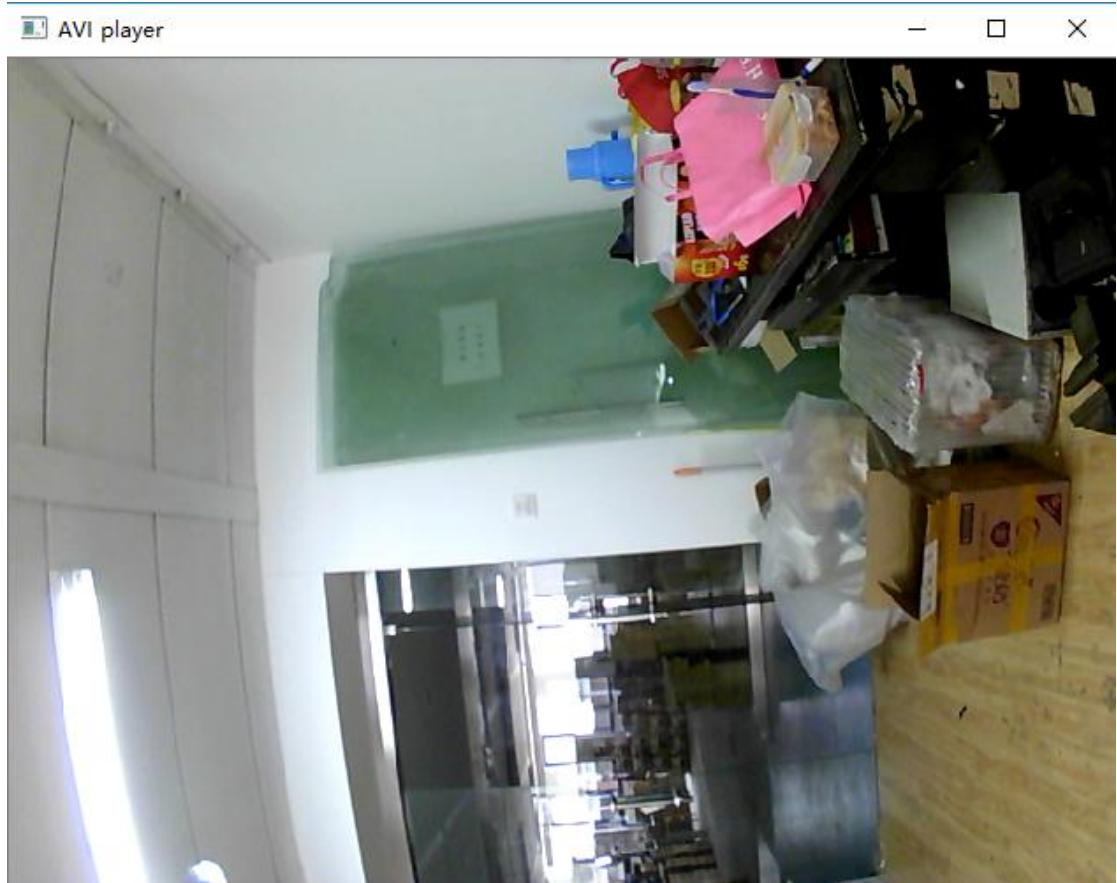


Step6: 单击 开始仿真。

Step7: 运行一段时间后, 运行结果如下所示:







3.6 本章小结

本章介绍了 3 种 HLS 仿真中最常用的获取仿真数据流的方法，并对其进行了验证。在接下来的算法设计中，我们就即将用到本章介绍的这 3 种方法，对设计的功能进行仿真，掌握好这一章的内容以便完成接下来的算法设计。

S05_CH04_Skin_Dection 实验

4.1 肤色检测原理及应用

肤色作为人的体表显著特征之一,尽管人的肤色因为人种的不同有差异,呈现出不同的颜色,但是在排除了亮度和视觉环境等对肤色的影响后,皮肤的色调基本一致,这就为利用颜色信息来做肤色分割提供了理论的依据。

在肤色识别中,常用的颜色空间为 YCbCr 颜色空间。在 YCbCr 颜色空间中, Y 代表亮度, Cb 和 Cr 分别代表蓝色分量和红色分量,两者合称为色彩分量。YCbCr 颜色空间具有将色度与亮度分离的特点,在 YCbCr 色彩空间中,肤色的聚类特性比较好,而且是二维独立分布,能够比较好地限制肤色的分布区域,并且受人种的影响不大。对比 RGB 颜色空间和 YCbCr 颜色空间,当光强发生变化时,RGB 颜色空间中 (R, G, B) 会同时发生变化,而 YCbCr 颜色空间中受光强相对独立,色彩分量受光强度影响不大,因此 YCbCr 颜色空间更适合用于肤色识别。

由于肤色在 YCbCr 空间受亮度信息的影响较小,本算法直接考虑 YCbCr 空间的 CbCr 分量,映射为二维独立分布的 CbCr 空间。在 CbCr 空间下,肤色类聚性好,利用人工阈值法将肤色与非肤色区域分开,形成二值图像。

RGB 转 YCbCr 的公式为:

$$Y = 0.257*R + 0.564*G + 0.098*B + 16$$

$$Cb = -0.148*R - 0.291*G + 0.439*B + 128$$

$$Cr = 0.439*R - 0.368*G - 0.071*B + 128$$

对肤色进行判定的条件常使用如下判定条件:

$$Cb > 77 \quad \&\& \quad Cb < 127$$

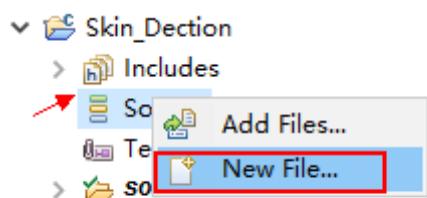
$$Cr > 133 \quad \&\& \quad Cr < 173$$

4.2 检测算法实现

4.2.1 工程创建

Step1: 打开 HLS, 按照之前介绍的方法, 创建一个新的工程, 命名为 Skin_Dection。

Step2: 右单击 Source 选项, 选择 New File, 创建一个名为 Top.cpp 的文件。



Step3: 在打开的编辑区中，把下面的程序拷贝进去：

```
#include "top.h"
#include <string.h>

void hls::hls_skin_dection(RGB_IMAGE& src, RGB_IMAGE& dst, int rows,
int cols,
    int y_lower, int y_upper, int cb_lower, int cb_upper, int
cr_lower, int cr_upper)
{

    LOOP_ROWS:for(int row = 0; row < rows ; row++)
    {

        LOOP_COLS:for(int col = 0; col < cols; col++)
        {

            //变量定义
            RGB_PIXEL src_data;
            RGB_PIXEL pix;
            RGB_PIXEL dst_data;
            bool skin_region;

            if(row < rows && col < cols) {
                src >> src_data;
            }

            //读RGB通道数据
            uchar B = src_data.val[0];
            uchar G = src_data.val[1];
            uchar R = src_data.val[2];

            //RGB-->YCbCr颜色转换
            uchar y = (76 * R + 150 * G + 29 * B) >> 8;
            uchar cb = ((128*B - 43*R - 85*G)>>8) + 128 ;
            uchar cr = ((128*R - 107*G - 21 * B)>>8)+ 128 ;
        }
    }
}
```

```
//肤色检测
if (y > y_lower && y < y_upper && cb > cb_lower && cb <
cb_upper && cr > cr_lower && cr < cr_upper)
    skin_region = 1;
else
    skin_region = 0;

uchar temp0= (skin_region == 1)? (uchar)255: B;
uchar temp1= (skin_region == 1)? (uchar)255: G;
uchar temp2= (skin_region == 1)? (uchar)255: R;

dst_data.val[0] = temp0;
dst_data.val[1] = temp1;
dst_data.val[2] = temp2;

dst << dst_data;
}

}

void ImgProcess_Top(AXI_STREAM& input, AXI_STREAM& output, int rows, int cols,
int y_lower, int y_upper, int cb_lower, int cb_upper, int
cr_lower, int cr_upper)
{
    RGB_IMAGE img_0(rows, cols);
    RGB_IMAGE img_1(rows, cols);

#pragma HLS dataflow

    hls::AXIvideo2Mat(input, img_0);
    hls::hls_skin_dection(img_0, img_1, rows, cols, y_lower, y_upper, cb_lower, cb_upper,
cr_lower, cr_upper);
    hls::Mat2AXIvideo(img_1, output);
}
```

Step4：在 Test Bench 中，用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码：

```
#include "top.h"
#include "hls_opencv.h"
#include "iostream"
#include <time.h>

using namespace std;
using namespace cv;

int main (int argc, char** argv)
{
    //IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* src = cvLoadImage("test.jpg");
    //IplImage* src = cvLoadImage("test_img1.jpg");
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth, src->nChannels);

    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIvideo(src, src_axi);

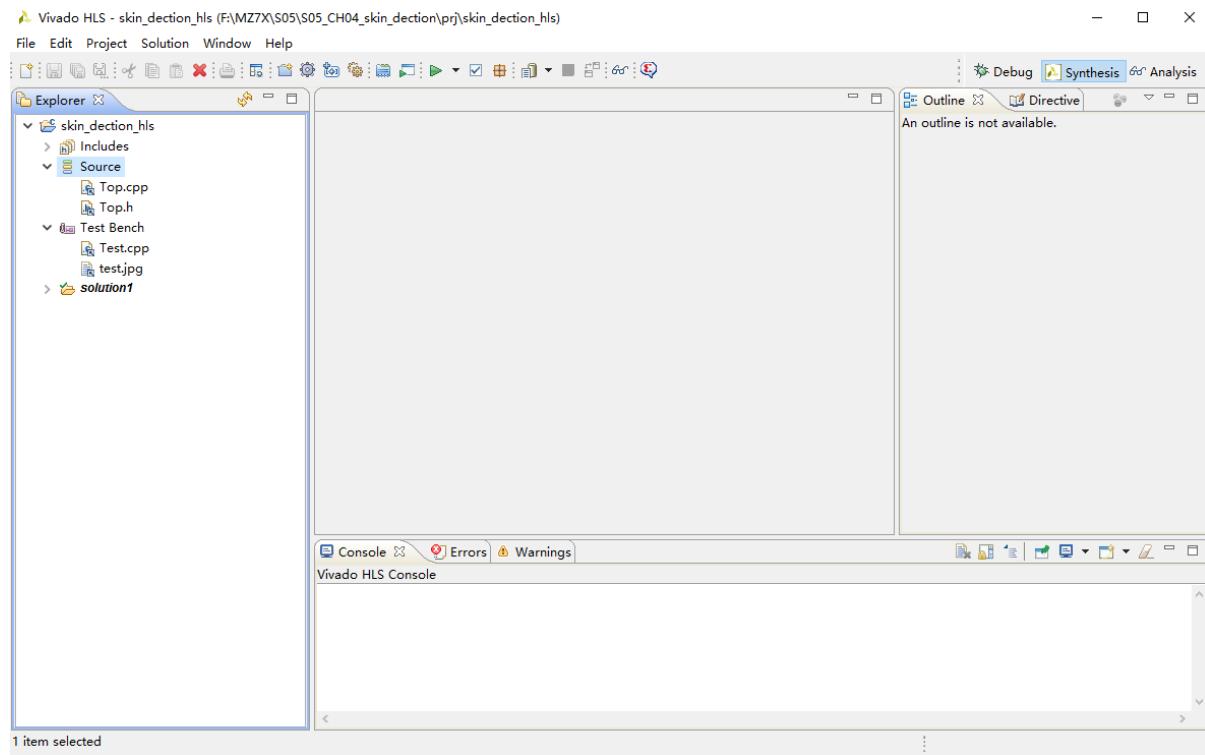
    ImgProcess_Top(src_axi, dst_axi, src->height,
src->width, 0, 255, 75, 125, 131, 185);
    AXIvideo2IplImage(dst_axi, dst);

    cvShowImage("src", src);
    cvShowImage("dst_hls", dst);
    waitKey(0);

    return 0;
}
```

Step6: 在 Test Bench 中添加一张名为 test.jpg 的测试图片，图片可以在我们提供的源程序中的

Image 文件夹中找到。完整的工程如下图所示：



4.2.2 代码综合

了解了肤色检测的原理以后我们进行算法实现可以发现，算法的关键就是进行 RGB 到 YCbCr 颜色空间的转换，其关键算法如下，大家可以发现其实算法只需要几十行就可以很容易的实现检测算法，不过需要大家注意的是因为我们使用 C++ 进行开发，那么为了程序的通用性以及方便团队合作开发，建议大家使用 namespace 命名空间，这样可以很好的解决重名问题，而且可以增加程序的可读性。

```
//声明hls命名空间
namespace hls
{
    void hls_skin_dectection(RGB_IMAGE& src, RGB_IMAGE& dst,int rows, int cols,
                                int y_lower,int y_upper,int cb_lower,int cb_upper,int cr_lower,int cr_upper);
}
```

```
void hls::hls_skin_dection(RGB_IMAGE& src, RGB_IMAGE& dst, int rows, int cols,
    int y_lower, int y_upper, int cb_lower, int cb_upper, int cr_lower, int cr_upper)
{
    Loop_ROWS:for(int row = 0; row < rows ; row++)
    {
        #pragma HLS loop_flatten off
        Loop_COLS:for(int col = 0; col < cols; col++)
        {
            #pragma HLS pipeline II=1
            //变量定义
            RGB_PIXEL src_data;
            RGB_PIXEL pix;
            RGB_PIXEL dst_data;
            bool skin_region;
            if(row < rows && col < cols) {
                src >> src_data;
            }

            //获取RGB通道数据
            uchar B = src_data.val[0];
            uchar G = src_data.val[1];
            uchar R = src_data.val[2];

            //RGB-->YCbCr颜色空间转换
            uchar y = (76 * R + 150 * G + 29 * B) >> 8;
            uchar cb = ((128*B - 43*R - 85*G)>>8) + 128 ;
            uchar cr = ((128*R - 107*G - 21 * B)>>8)+ 128 ;

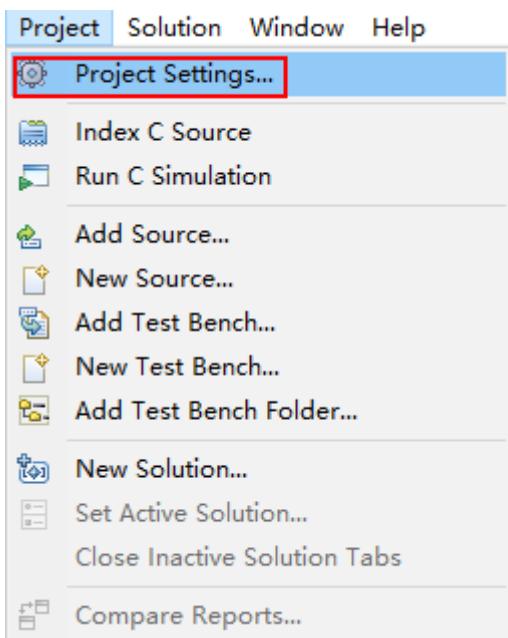
            //肤色区域判定
            if (y > y_lower && y < y_upper && cb > cb_lower && cb < cb_upper && cr > cr_lower && cr < cr_upper)
                skin_region = 1;
            else
                skin_region = 0;

            uchar temp0= (skin_region == 1)? (uchar)255: B;
            uchar temp1= (skin_region == 1)? (uchar)255: G;
            uchar temp2= (skin_region == 1)? (uchar)255: R;

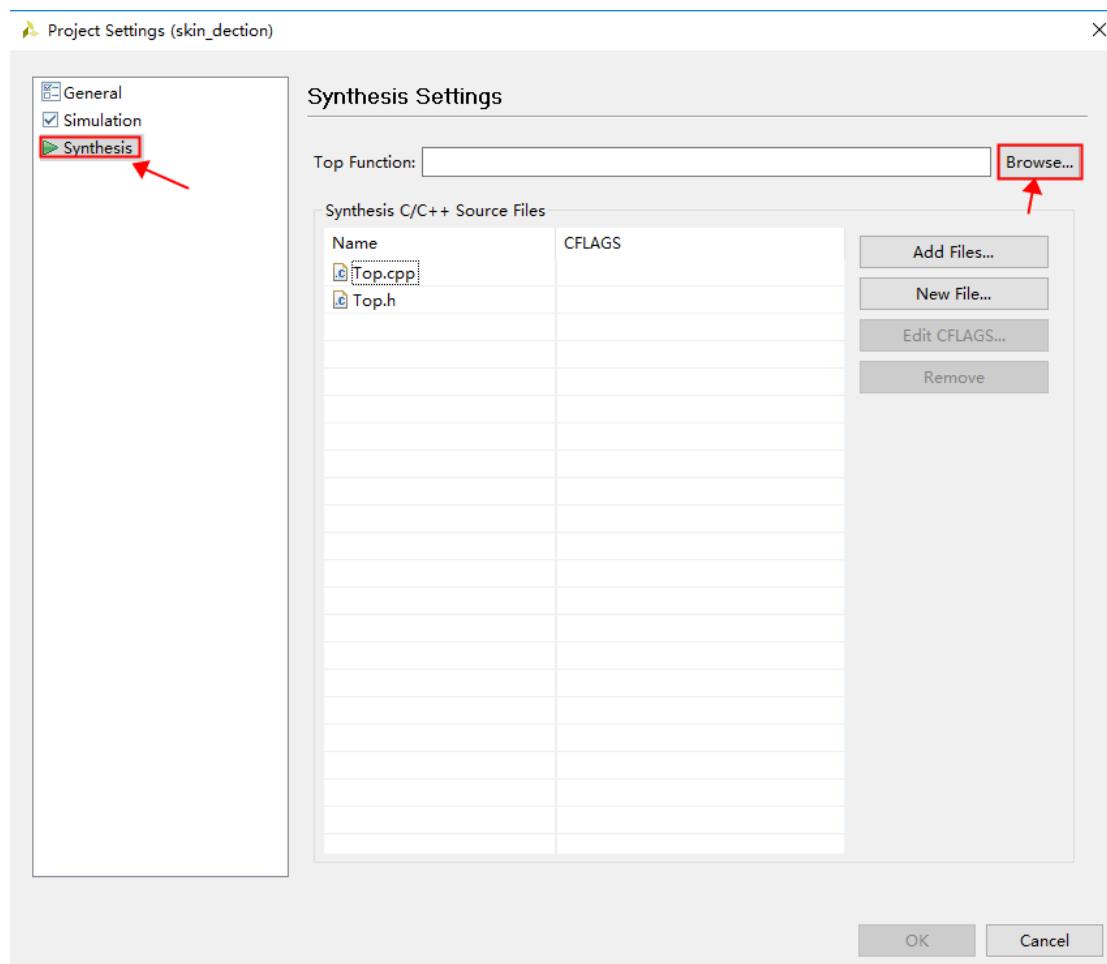
            dst_data.val[0] = temp0;
            dst_data.val[1] = temp1;
            dst_data.val[2] = temp2;
            dst << dst_data;
        }
    }
}
```

接下来，我们对项目进行代码综合。

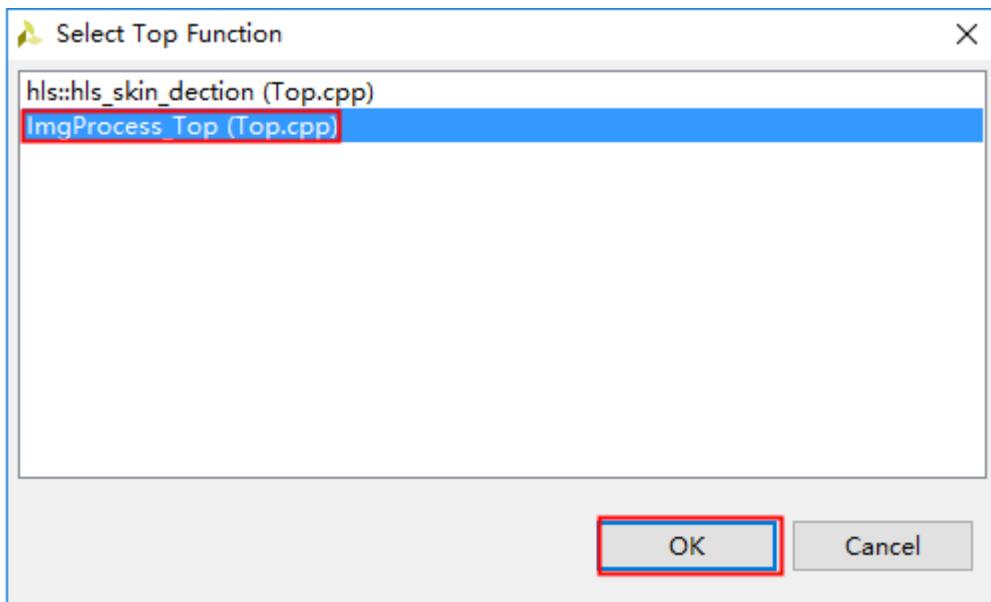
Step1：单击 Project—project settings 命令。



Step2: 在弹出的窗口中选择综合选项，然后在顶层函数设置区中单击右边的 Browse...按钮。



Step3: 选择 ImgProcess_Top 为顶层函数，然后单击两次 OK，完成工程的设置。



Step4: 单击综合快捷按钮开始代码综合。



等待一段时间后，在未经代码优化及约束的条件下生成的综合报告如图：

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	0	-	90	568
Instance	-	7	687	552
Memory	-	-	-	-
Multiplexer	-	-	-	6
Register	-	-	11	-
Total	0	7	788	1127
Available	280	220	106400	53200
Utilization (%)	0	3	~0	2

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
ImgProcess_Top_AXIvideo2Mat_U0	ImgProcess_Top_AXIvideo2Mat	0	0	198	220
ImgProcess_Top_Block_Mat_exit45_proc36_U0	ImgProcess_Top_Block_Mat_exit45_proc36	0	0	26	25
ImgProcess_Top_Mat2AXIvideo_U0	ImgProcess_Top_Mat2AXIvideo	0	0	79	75
ImgProcess_Top_hls_skin_dection_U0	ImgProcess_Top_hls_skin_dection	0	7	384	232
Total		4	0	7	687

DSP48

Memory

FIFO

4.2.3 代码优化

根据官方手册 how_to_accelerate_opencv_applications_using_vivado_hls.pdf 中的描述：

➤ Assign ‘input’ to be an AXI4 stream named “INPUT_STREAM”

```
#pragma HLS RESOURCE variable=input core=AXIS metadata="-bus_bundle INPUT_STREAM"
```

➤ Assign control interface to an AXI4-Lite interface

```
#pragma HLS RESOURCE variable=return core=AXI_SLAVE metadata="-bus_bundle CONTROL_BUS"
```

➤ Assign ‘rows’ to be accessible through the AXI4-Lite interface

```
#pragma HLS RESOURCE variable=rows core=AXI_SLAVE metadata="-bus_bundle CONTROL_BUS"
```

➤ Declare that ‘rows’ will not be changed during the execution of the function

```
#pragma HLS INTERFACE ap_stable port=rows
```

➤ Enable streaming dataflow optimizations

```
#pragma HLS dataflow
```

我们对视频接口的约束如下: 将 src 和 dst 指定为以“INPUT_STREAM”命名的 AXI4 Stream, 将控制接口分配到 AXI4 Lite 接口, 指定“rows”可通过 AXI4-Lite 接口进行访问并且声明在函数执行过程中“rows”不会改变, 其实这几句优化语句中最关键的一条指令为启用数据流优化:

#pragma HLS dataflow , 它使得任务之间以流水线的方式执行, 即
hls::AXIVideo2Mat(src, img_0);hls::skin_detect(img_0, img_1, rows, cols, cb_lower, cb_upper,
cr_lower, cr_upper);hls::Mat2AXIVideo(img_1, dst), 这三个函数之间为流水线方式。

```
#pragma HLS RESOURCE variable=input core=AXIS metadata="-bus_bundle INPUT_STREAM"
#pragma HLS RESOURCE variable=output core=AXIS metadata="-bus_bundle OUTPUT_STREAM"

#pragma HLS RESOURCE core=AXI_SLAVE variable=rows metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cols metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=y_lower metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=y_upper metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cb_lower metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cb_upper metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cr_lower metadata="-bus_bundle CONTROL_BUS"
```

```

#pragma HLS RESOURCE core=AXI_SLAVE variable=cr_upper
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=return
metadata="-bus_bundle CONTROL_BUS"

#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols
#pragma HLS INTERFACE ap_stable port=y_lower
#pragma HLS INTERFACE ap_stable port=y_upper
#pragma HLS INTERFACE ap_stable port=cb_lower
#pragma HLS INTERFACE ap_stable port=cb_upper
#pragma HLS INTERFACE ap_stable port=cr_lower
#pragma HLS INTERFACE ap_stable port=cr_upper

RGB_IMAGE img_0(rows, cols);
RGB_IMAGE img_1(rows, cols);

#pragma HLS dataflow

```

```

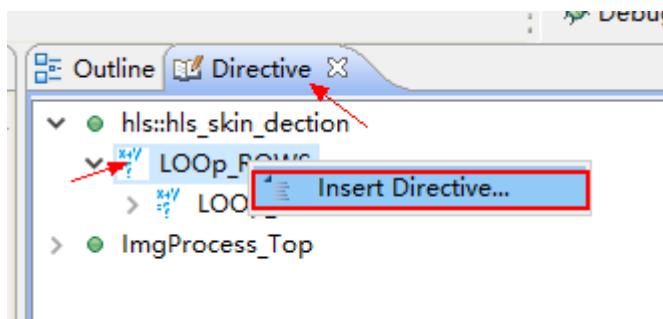
55 void ImgProcess_Top(AXI_STREAM& input, AXI_STREAM& output,int rows, int cols,
56                     int y_lower,int y_upper,int cb_lower,int cb_upper,int cr_lower,int cr_upper)
57 {
58
59     #pragma HLS RESOURCE variable=input core=AXIS metadata="-bus_bundle INPUT_STREAM"
60     #pragma HLS RESOURCE variable=output core=AXIS metadata="-bus_bundle OUTPUT_STREAM"
61
62     #pragma HLS RESOURCE core=AXI_SLAVE variable=rows metadata="-bus_bundle CONTROL_BUS"
63     #pragma HLS RESOURCE core=AXI_SLAVE variable=cols metadata="-bus_bundle CONTROL_BUS"
64     #pragma HLS RESOURCE core=AXI_SLAVE variable=y_lower metadata="-bus_bundle CONTROL_BUS"
65     #pragma HLS RESOURCE core=AXI_SLAVE variable=y_upper metadata="-bus_bundle CONTROL_BUS"
66     #pragma HLS RESOURCE core=AXI_SLAVE variable=cb_lower metadata="-bus_bundle CONTROL_BUS"
67     #pragma HLS RESOURCE core=AXI_SLAVE variable=cb_upper metadata="-bus_bundle CONTROL_BUS"
68     #pragma HLS RESOURCE core=AXI_SLAVE variable=cr_lower metadata="-bus_bundle CONTROL_BUS"
69     #pragma HLS RESOURCE core=AXI_SLAVE variable=cr_upper metadata="-bus_bundle CONTROL_BUS"
70     #pragma HLS RESOURCE core=AXI_SLAVE variable=return metadata="-bus_bundle CONTROL_BUS"
71
72     #pragma HLS INTERFACE ap_stable port=rows
73     #pragma HLS INTERFACE ap_stable port=cols
74     #pragma HLS INTERFACE ap_stable port=y_lower
75     #pragma HLS INTERFACE ap_stable port=y_upper
76     #pragma HLS INTERFACE ap_stable port=cb_lower
77     #pragma HLS INTERFACE ap_stable port=cb_upper
78     #pragma HLS INTERFACE ap_stable port=cr_lower
79     #pragma HLS INTERFACE ap_stable port=cr_upper
80
81     RGB_IMAGE img_0(rows, cols);
82     RGB_IMAGE img_1(rows, cols);
83
84     #pragma HLS dataflow
85     hls::AXIVideo2Mat(input,img_0);
86     hls::hls_skin_decton(img_0,img_1,rows,cols,y_lower,y_upper,cb_lower,cb_upper,cr_lower,cr_upper);
87     hls::Mat2AXIVideo(img_1, output);
88 }
89

```

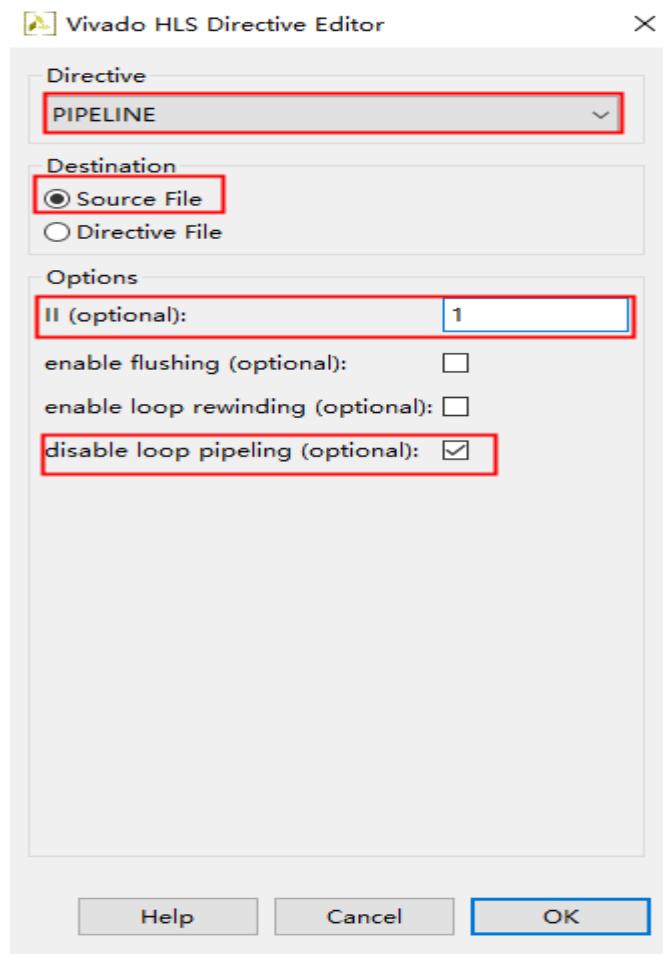
并且在肤色检测函数中添加如下的流水线:

```
4④ void hls::hls_skin_decton(RGB_IMAGE& src, RGB_IMAGE& dst, int rows, int cols,
5    int y_lower, int y_upper, int cb_lower, int cb_upper, int cr_lower, int cr_upper)
6 {
7
8     LOOP_ROWS:for(int row = 0; row < rows ; row++)
9     {
10        #pragma HLS PIPELINE II=1 off
11
12        LOOP_COLS:for(int col = 0; col < cols; col++)
13        {
14
15            //变量定义
16            RGB_PIXEL src_data;
17            RGB_PIXEL pix;
18            RGB_PIXEL dst_data;
19            bool skin_region;
20
21            if(row < rows && col < cols) {
22                src >> src_data;
23            }
24        }
25    }
26
27    dst >> dst_data;
28
29    if(y_lower < y_upper)
30    {
31        for(int i = y_lower; i < y_upper; i++)
32        {
33            for(int j = cb_lower; j < cb_upper; j++)
34            {
35                for(int k = cr_lower; k < cr_upper; k++)
36                {
37                    dst_data = src_data;
38
39                    if(skin_region)
40                    {
41                        dst_data = skin_color;
42                    }
43
44                    dst >> dst_data;
45                }
46            }
47        }
48    }
49
50    return;
51}
```

那么这条指令是如何添加的呢？在下图位置右键点击 Insert Directive...



配置如下即可生成#pragma HLS PIPELINE II=1 off 的优化指令，



进行优化以后我们再次进行综合，综合的结果如下，我们对比发现

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	0	-	30	120
Instance	-	7	360	466
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	4	-
Total	0	7	394	587
Available	280	220	106400	53200
Utilization (%)	0	3	~0	1

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
ImgProcess_Top_AXIvideo2Mat_U0	ImgProcess_Top_AXIvideo2Mat	0	0	147	163
ImgProcess_Top_Mat2AXIvideo_U0	ImgProcess_Top_Mat2AXIvideo	0	0	43	71
ImgProcess_Top_hls_skin_dection_U0	ImgProcess_Top_hls_skin_dection	0	7	170	232
Total		3	0	7	360

DSP48

Memory

FIFO

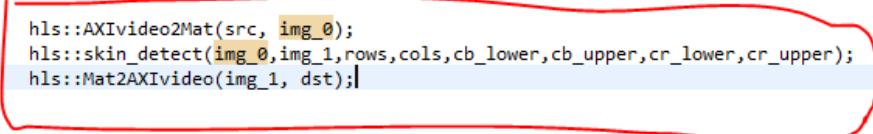
Name	BRAM_18K	FF	LUT	Depth	Bits	Size:D*B
img_0_data_stream_0_V_U	0	5	20	1	8	8
img_0_data_stream_1_V_U	0	5	20	1	8	8
img_0_data_stream_2_V_U	0	5	20	1	8	8
img_1_data_stream_0_V_U	0	5	20	1	8	8
img_1_data_stream_1_V_U	0	5	20	1	8	8
img_1_data_stream_2_V_U	0	5	20	1	8	8
Total	0	30	120	6	48	48

我们可以发现使用了 7 个 DSP48E, 以及 394 个触发器(FF)和 587 个查找表(LUT), 并且在 Instance 里面可以看到例化了四个模块, 其实对应的就是下面红框内的三个函数,

```

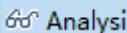
104     RGB_IMAGE img_0(rows, cols);
105     RGB_IMAGE img_1(rows, cols);
106
107     hls::AXIvideo2Mat(src, img_0);
108     hls::skin_detect(img_0,img_1,rows,cols,cb_lower,cb_upper,cr_lower,cr_upper);
109     hls::Mat2AXIvideo(img_1, dst);
110 }
111

```

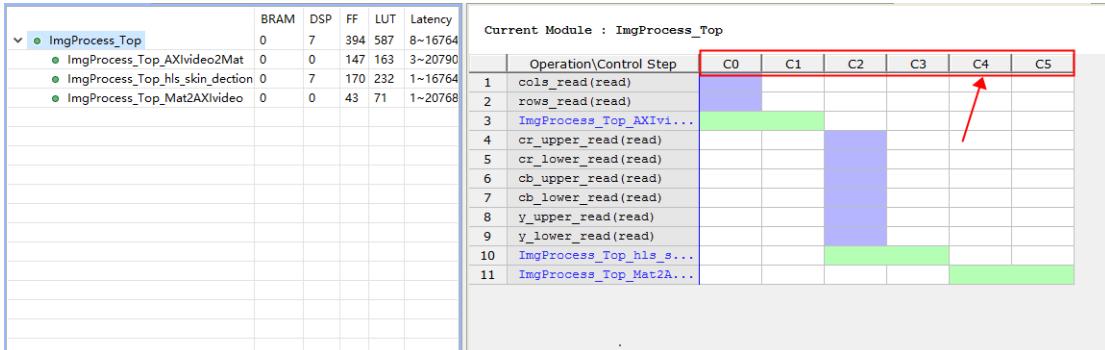


并且我们可以看到在 FIFO 中例化了 6 个不同名称的 FIFO, 那么对比名称我们能发现什么呢? 对了, 它就是声明中定义的 img_0 和 img_1 的数据的存储区, 那么为什么是 3 个 8bit 的呢? 因为对于 RGB_IMAGE 我们的定义如下, 其中 HLS_8UC3 即 3 通道的 8bit unsigned char 型数据, 因此, 综合生成了 3 个 8bit 位宽的 FIFO。

```
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> RGB_IMAGE;
```

单击  打开综合分析窗口, 我们再看一下我们的肤色检测的核心函数所消耗的时间,

将 LOOP 展开，我们可以发现函数消耗 6 个时钟周期才处理完毕。



查看代码我们可以发现我们使用了 for 循环 (LOOP)，下面对循环 LOOP 控制的优化指令进行说明：

Unrolling：展开循环，用于创建多个独立的操作，而不是对单个操作进行整合。

Merging：合并连续的循环，降低总延迟，提高共享和优化。

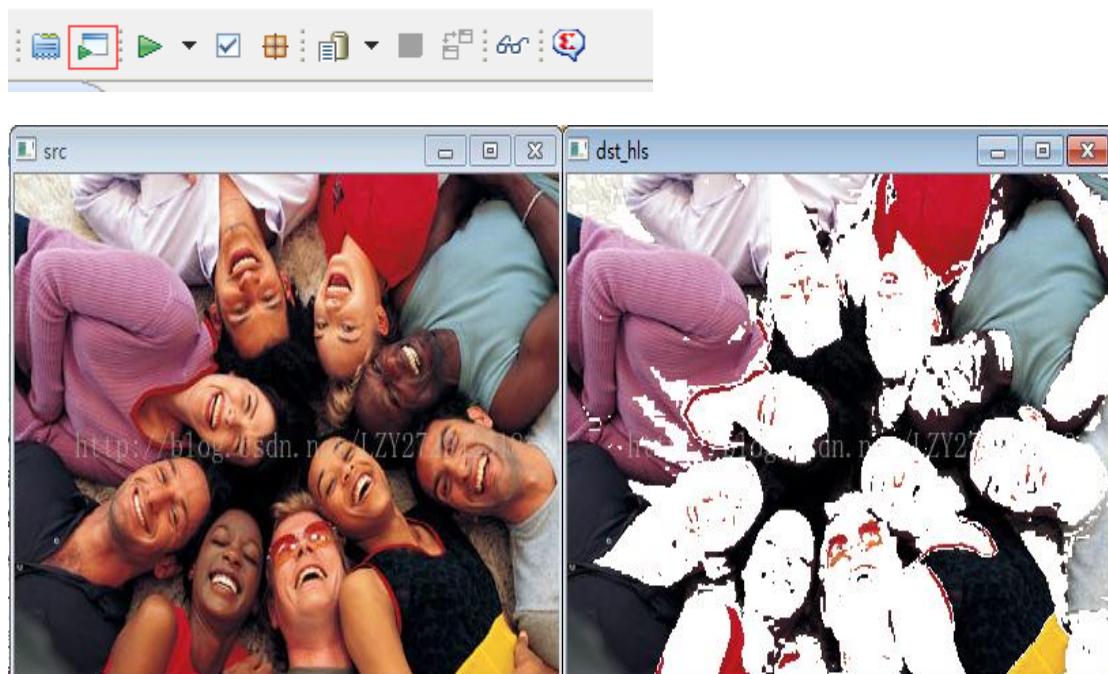
Flatting：允许将带有改善延迟和逻辑优化的嵌套循环，整理成一个单个的循环。

Dataflow：允许顺序循环，并发的操作。

Pipelining：通过执行并发的操作，提高吞吐量。

4.3 仿真测试

点击如下图方框圈出的快捷仿真按钮，我们对 Test.jpg 图片进行肤色检测，检测结果如下图所示。



通过对比发现基于颜色空间转换加阈值进行判断能在一定程度上进行检测，但是存在的检测误差较大的，在实际的项目应用中，需要对算法本身进行优化，那么在这里就不在进行介绍。

4.4 本章小结

本章节通过设计一个肤色检测的算法对前面几章的内容进行了巩固和验证。最后通过一组仿真验证了整个算法的有效性，虽然最后发现基于颜色空间转换加阈值进行肤色检测的结果又一点误差，但我们完全可以对算法进行优化，减小误差，本章重点介绍的是如何设计算法，因此对算法的优化就不再介绍，感兴趣的可以尝试优化一下算法。

S05_CH05_Sobel 算子硬件实现(一)_HLS 实现

5.1 Sobel 原理介绍

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 卷积因子为：

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

G_x

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

G_y

该算子包含两组 3x3 的矩阵，分别为横向及纵向，将之与图像作平面卷积，即可分别得出横向及纵向的亮度差分近似值。如果以 A 代表原始图像， G_x 及 G_y 分别代表经横向及纵向边缘检测的图像灰度值，其公式如下：

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

具体计算如下：

$$\begin{aligned} G_x &= (-1)*f(x-1, y-1) + 0*f(x, y-1) + 1*f(x+1, y-1) \\ &\quad + (-2)*f(x-1, y) + 0*f(x, y) + 2*f(x+1, y) \\ &\quad + (-1)*f(x-1, y+1) + 0*f(x, y+1) + 1*f(x+1, y+1) \\ &= [f(x+1, y-1) + 2*f(x+1, y) + f(x+1, y+1)] - [f(x-1, y-1) + 2*f(x-1, y) + f(x-1, y+1)] \\ G_y &= 1*f(x-1, y-1) + 2*f(x, y-1) + 1*f(x+1, y-1) \\ &\quad + 0*f(x-1, y) + 0*f(x, y) + 0*f(x+1, y) \\ &\quad + (-1)*f(x-1, y+1) + (-2)*f(x, y+1) + (-1)*f(x+1, y+1) \end{aligned}$$

$$= [f(x-1, y-1) + 2f(x, y-1) + f(x+1, y-1)] - [f(x-1, y+1) + 2*f(x, y+1) + f(x+1, y+1)]$$

其中 $f(a, b)$, 表示图像 (a, b) 点的灰度值;

图像的每一个像素的横向及纵向灰度值通过以下公式结合, 来计算该点灰度的大小:

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

通常, 为了提高效率 使用不开平方的近似值:

$$|G| = |G_x| + |G_y|$$

如果梯度 G 大于某一阀值 则认为该点 (x, y) 为边缘点。

然后可用以下公式计算梯度方向:

$$\Theta = \arctan\left(\frac{G_y}{G_x}\right)$$

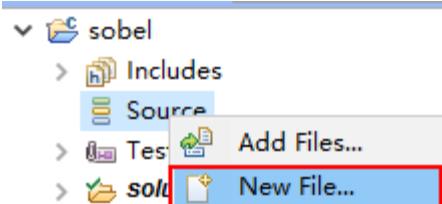
Sobel 算子根据像素点上下、左右邻点灰度加权差, 在边缘处达到极值这一现象检测边缘。对噪声具有平滑作用, 提供较为精确的边缘方向信息, 边缘定位精度不够高。当对精度要求不是很高时, 是一种较为常用的边缘检测方法。

5.2 Sobel 算子在 HLS 上的实现

5.2.1 工程创建

Step1: 打开 HLS, 按照之前介绍的方法, 创建一个新的工程, 命名为 sobel。

Step2: 右单击 Source 选项, 选择 New File, 创建一个名为 Top.cpp 的文件。



Step3: 在打开的编辑区中, 把下面的程序拷贝进去:

```
#include "top.h"

void hls_sobel(AXI_STREAM& INPUT_STREAM, AXI_STREAM&
OUTPUT_STREAM, int rows, int cols)
{
    //Create AXI streaming interfaces for the core
    #pragma HLS INTERFACE axis port=INPUT_STREAM
```

```

#pragma HLS INTERFACE axis port=OUTPUT_STREAM

#pragma HLS RESOURCE core=AXI_SLAVE variable=rows
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cols
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=return
metadata="-bus_bundle CONTROL_BUS"

#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols

RGB_IMAGE img_0(rows, cols);
RGB_IMAGE img_1(rows, cols);
RGB_IMAGE img_2(rows, cols);
RGB_IMAGE img_3(rows, cols);
RGB_IMAGE img_4(rows, cols);
RGB_IMAGE img_5(rows, cols);
RGB_PIXEL pix(50, 50, 50);

#pragma HLS dataflow
hls::AXIVideo2Mat(INPUT_STREAM, img_0);
hls::Sobel<1,0,3>(img_0, img_1);
hls::SubS(img_1, pix, img_2);
hls::Scale(img_2, img_3, 2, 0);
hls::Erode(img_3, img_4);
hls::Dilate(img_4, img_5);
hls::Mat2AXIVideo(img_5, OUTPUT_STREAM);
}

```

Step4: 再在 Source 中添加一个名为 Top.h 的库函数，并添加如下程序:

```

#ifndef _TOP_H_
#define _TOP_H_

#include "hls_video.h"

// maximum image size
#define MAX_WIDTH 1920
#define MAX_HEIGHT 1080

// I/O Image Settings
#define INPUT_IMAGE      "test_1080p.bmp"
#define OUTPUT_IMAGE     "result_1080p.bmp"

```

```

#define OUTPUT_IMAGE_GOLDEN "result_1080p_golden.bmp"

// typedef video library core structures
typedef hls::stream<ap_axiu<32,1,1,1>> AXI_STREAM;
typedef hls::Scalar<3, unsigned char> RGB_PIXEL;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> RGB_IMAGE;

// top level function for HW synthesis
void hls_sobel(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int rows,
int cols);

#endif

```

Step5: 在 Test Bench 中, 用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码:

```

#include "top.h"
#include "opencv_top.h"

using namespace std;
using namespace cv;

int main (int argc, char** argv)
{
    //读取图像数据
    IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels);

    //使用HLS库进行处理
    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIVideo(src, src_axi);
    hls_sobel(src_axi, dst_axi, src->height, src->width);
    AXIVideo2IplImage(dst_axi, dst);
    cvSaveImage(OUTPUT_IMAGE,dst);
    cvShowImage("hls_dst", dst);

    //使用OPENCV库进行处理
    opencv_image_filter(src, dst);
    cvShowImage("cv_dst", dst);
    cvSaveImage(OUTPUT_IMAGE_GOLDEN,dst);
    waitKey(0);

    //释放内存
    cvReleaseImage(&src);
}

```

```

    cvReleaseImage(&dst);
}

```

Step6: 用同样的方法，再在 Test Bench 中创建一个 opencv_top.cpp 和 OpenCV_top.h 文件，添加如下程序：

OpenCV_top.cpp 代码如下：

```

#include "opencv_top.h"
#include "top.h"

void opencv_image_filter(IplImage* src, IplImage* dst)
{
    IplImage* tmp = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels);
    cvCopy(src, tmp);
    cv::Sobel((cv::Mat)tmp, (cv::Mat)dst, -1, 1, 0);
    cvSubS(dst, cvScalar(50,50,50), tmp);
    cvScale(tmp, dst, 2, 0);
    cvErode(dst, tmp);
    cvDilate(tmp, dst);
    cvReleaseImage(&tmp);
}

void sw_image_filter(IplImage* src, IplImage* dst)
{
    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIVideo(src, src_axi);
    hls_sobel(src_axi, dst_axi, src->height, src->width);
    AXIVideo2IplImage(dst_axi, dst);
}

```

OpenCV_top.h 代码如下：

```

#ifndef __OPENCV_TOP_H__
#define __OPENCV_TOP_H__

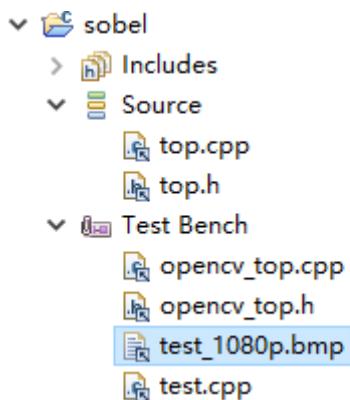
#include "hls_opencv.h"

void opencv_image_filter(IplImage* src, IplImage* dst);
void sw_image_filter(IplImage* src, IplImage* dst);

#endif

```

Step7: 在 Test Bench 中添加一张名为 test_1080p.bmp 的测试图片，图片可以在我们提供的源程序中的 Image 文件夹中找到。完整的工程如下图所示：



5.2.2 代码优化及仿真

在前面几个章节中我们已经讲过代码优化的方法和策略，这里在工程源文件中提供已经优化好的代码文件，本章节的代码优化部分如下所示：

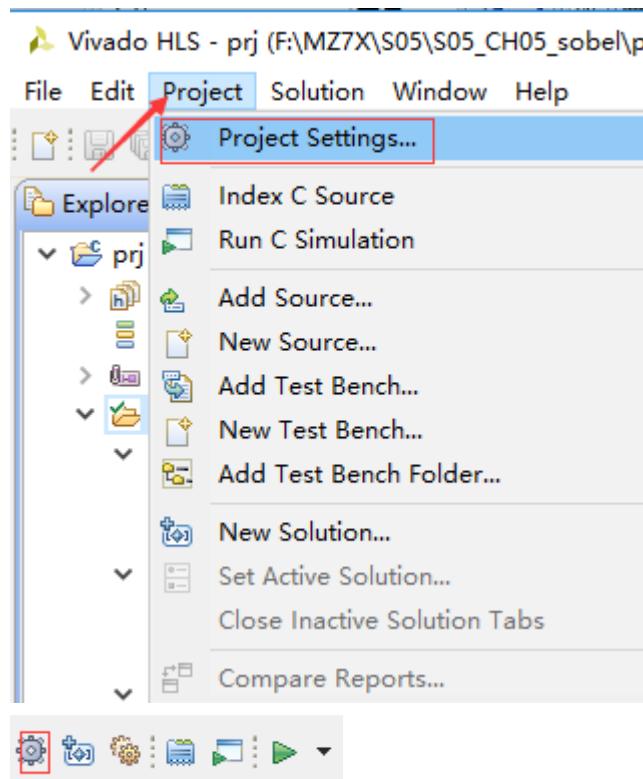
```
#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM

#pragma      HLS      RESOURCE      core=AXI_SLAVE      variable=rows
metadata="-bus_bundle CONTROL_BUS"
#pragma      HLS      RESOURCE      core=AXI_SLAVE      variable=cols
metadata="-bus_bundle CONTROL_BUS"
#pragma      HLS      RESOURCE      core=AXI_SLAVE      variable=return
metadata="-bus_bundle CONTROL_BUS"

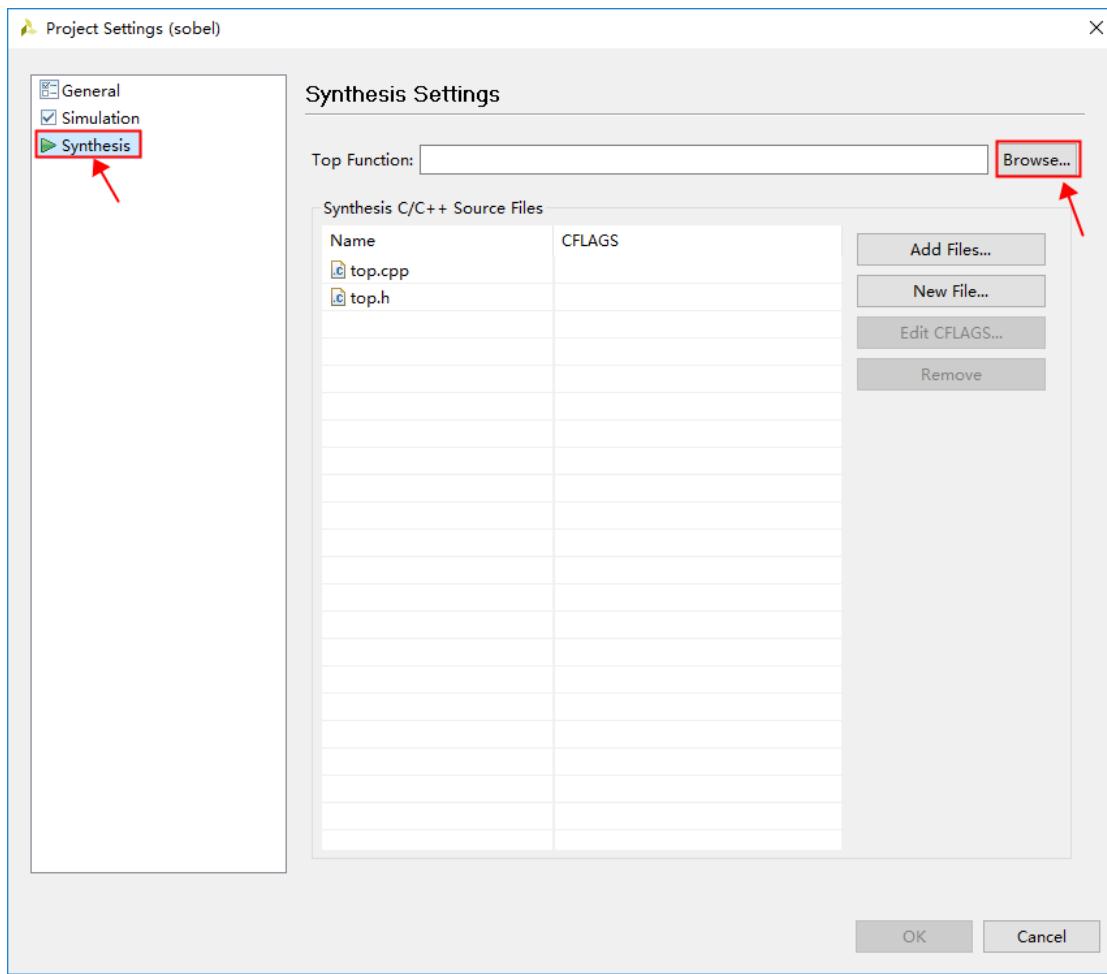
#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols
```

接下来对工程进行编译和仿真。

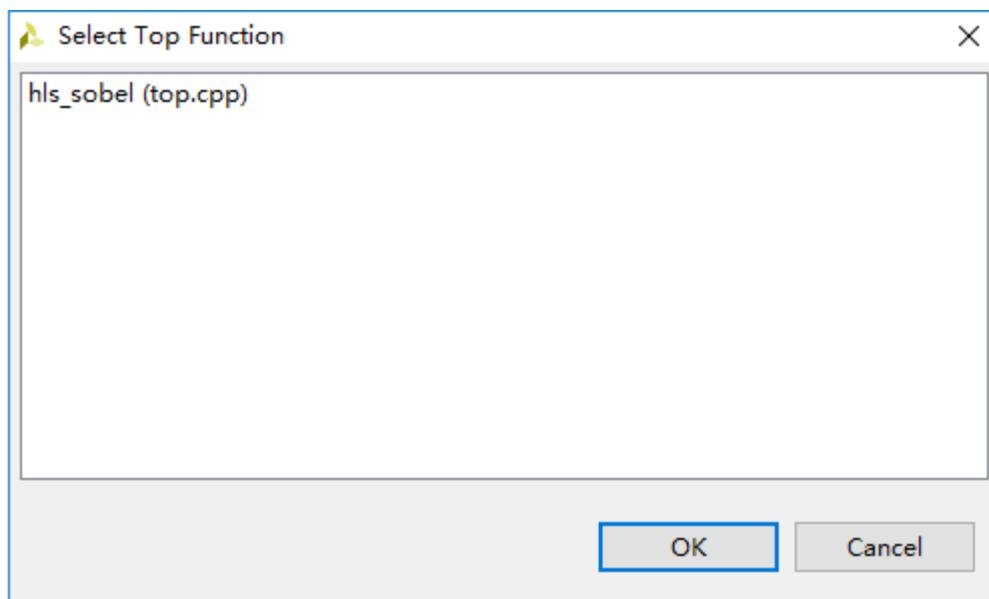
Step1：单击 Project→Project settings 或直接单击快捷按钮。



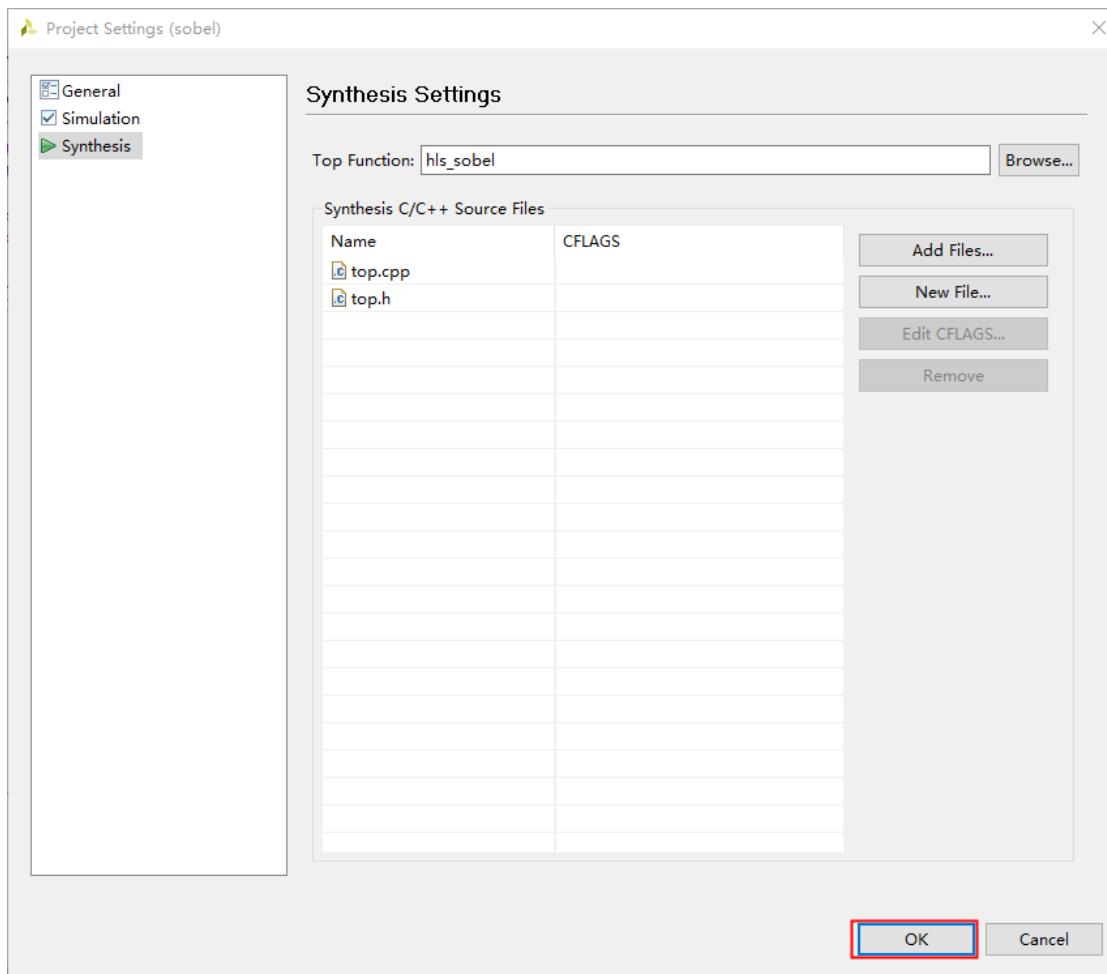
Step2: 选择 Synthesis 选项，然后点击 Browse.. 指定一个顶层函数。



Step3: 选定 hls_sobel 为顶层函数，然后点击 O K。



Step4: 再次单击 OK，完成工程的修改。



Step5: 单击 开始综合。



综合报告如下：

□ Timing (ns)**□ Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.73	1.25

□ Latency (clock cycles)**□ Summary**

Latency		Interval		
min	max	min	max	Type
197	2088277	185	2088265	dataflow

□ Detail**+ Instance****+ Loop****Utilization Estimates****□ Summary**

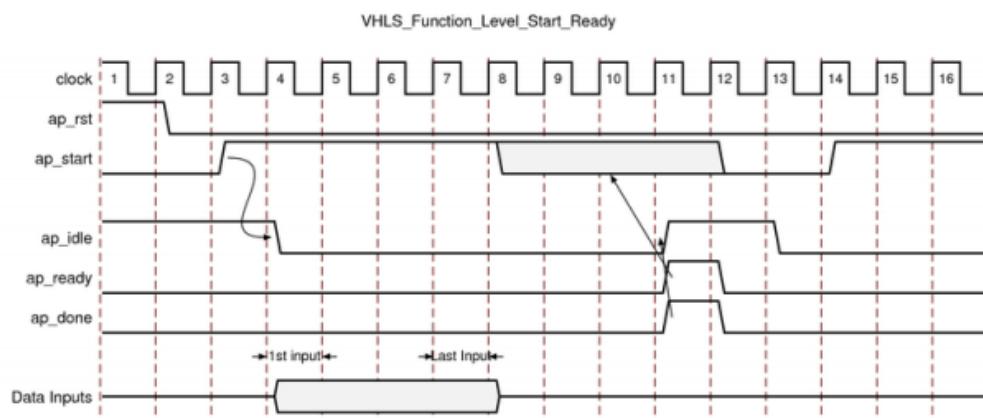
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	0	-	90	360
Instance	27	-	2099	3332
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	8	-
Total	27	0	2197	3693
Available	120	80	35200	17600
Utilization (%)	22	0	6	20

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
INPUT_STREAM_TDATA	in	32	axis	INPUT_STREAM_V_data_V	pointer
INPUT_STREAM_TKEEP	in	4	axis	INPUT_STREAM_V_keep_V	pointer
INPUT_STREAM_TSTRB	in	4	axis	INPUT_STREAM_V_strb_V	pointer
INPUT_STREAM_TUSER	in	1	axis	INPUT_STREAM_V_user_V	pointer
INPUT_STREAM_TLAST	in	1	axis	INPUT_STREAM_V_last_V	pointer
INPUT_STREAM_TID	in	1	axis	INPUT_STREAM_V_id_V	pointer
INPUT_STREAM_TDEST	in	1	axis	INPUT_STREAM_V_dest_V	pointer
INPUT_STREAM_TVALID	in	1	axis	INPUT_STREAM_V_dest_V	pointer
INPUT_STREAM_TREADY	out	1	axis	INPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TDATA	out	32	axis	OUTPUT_STREAM_V_data_V	pointer
OUTPUT_STREAM_TKEEP	out	4	axis	OUTPUT_STREAM_V_keep_V	pointer
OUTPUT_STREAM_TSTRB	out	4	axis	OUTPUT_STREAM_V_strb_V	pointer
OUTPUT_STREAM_TUSER	out	1	axis	OUTPUT_STREAM_V_user_V	pointer
OUTPUT_STREAM_TLAST	out	1	axis	OUTPUT_STREAM_V_last_V	pointer
OUTPUT_STREAM_TID	out	1	axis	OUTPUT_STREAM_V_id_V	pointer
OUTPUT_STREAM_TDEST	out	1	axis	OUTPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TVALID	out	1	axis	OUTPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TREADY	in	1	axis	OUTPUT_STREAM_V_dest_V	pointer
rows	in	32	ap_stable	rows	scalar
cols	in	32	ap_stable	cols	scalar
ap_clk	in	1	ap_ctrl_hs	hls_sobel	return value
ap_rst_n	in	1	ap_ctrl_hs	hls_sobel	return value
ap_start	in	1	ap_ctrl_hs	hls_sobel	return value
ap_done	out	1	ap_ctrl_hs	hls_sobel	return value
ap_idle	out	1	ap_ctrl_hs	hls_sobel	return value
ap_ready	out	1	ap_ctrl_hs	hls_sobel	return value

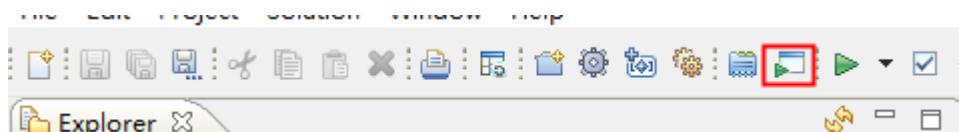
在协议类型里面我们可以看到我们主要使用了三种协议，分别是 axis、ap_stable 和 ap_ctrl_hs 三种，这些协议的详细解释我们均可以在官方手册 ug902 中找到，其中 ap_ctrl_hs 的时序操作如下图所示，说简单点就是复位完成等待 ap_start 信号开始进行操作。

ap_ctrl_hs

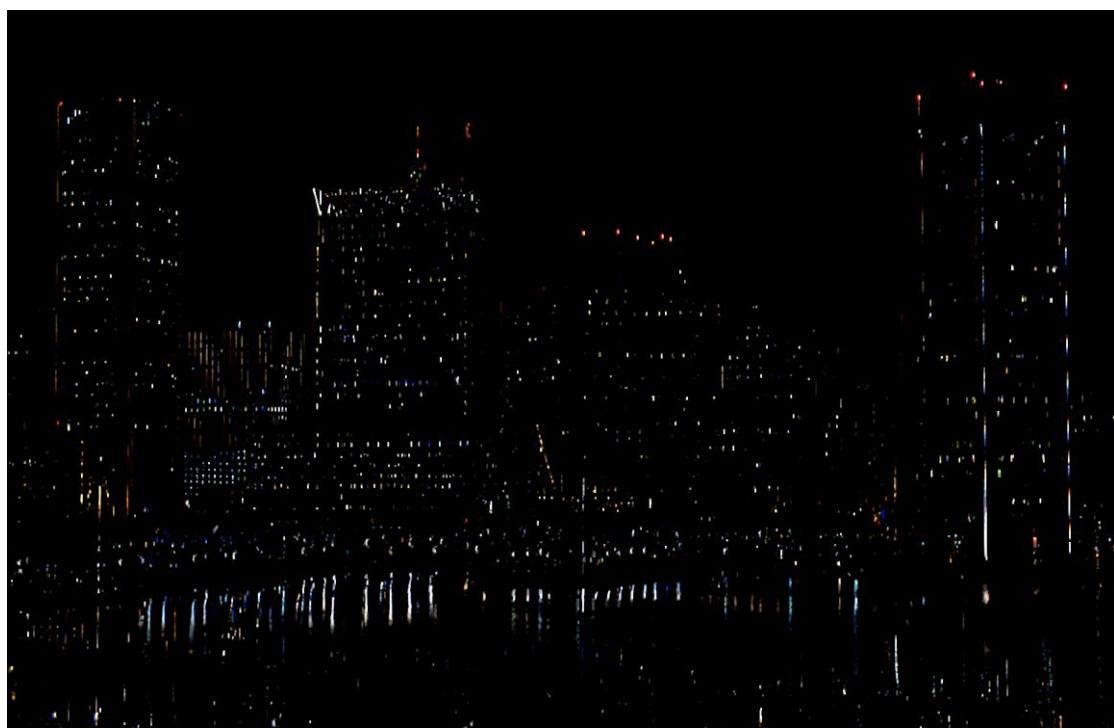
The behavior of the block level handshake signals created by interface mode ap_ctrl_hs are shown in the following figure and summarized below.



Step6：综合完毕，我们对代码进行仿真测试，单击 开始仿真。

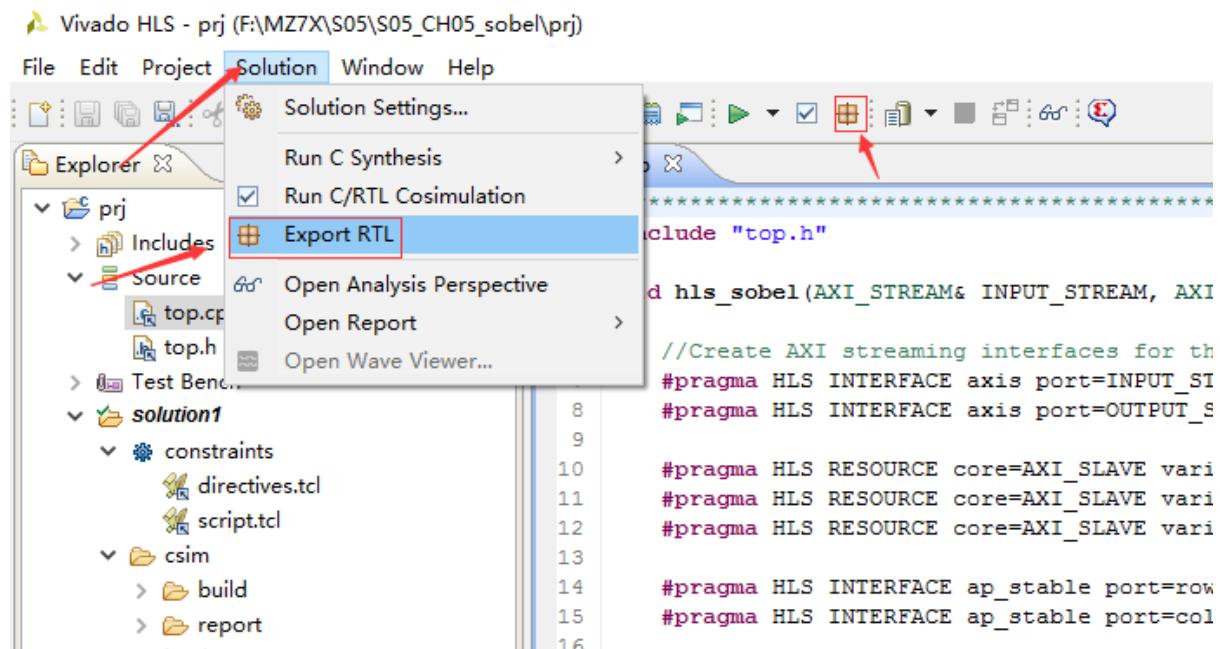


仿真结果如下，与通过 OPENCV 实现的 Sobel 检测结果基本一致。

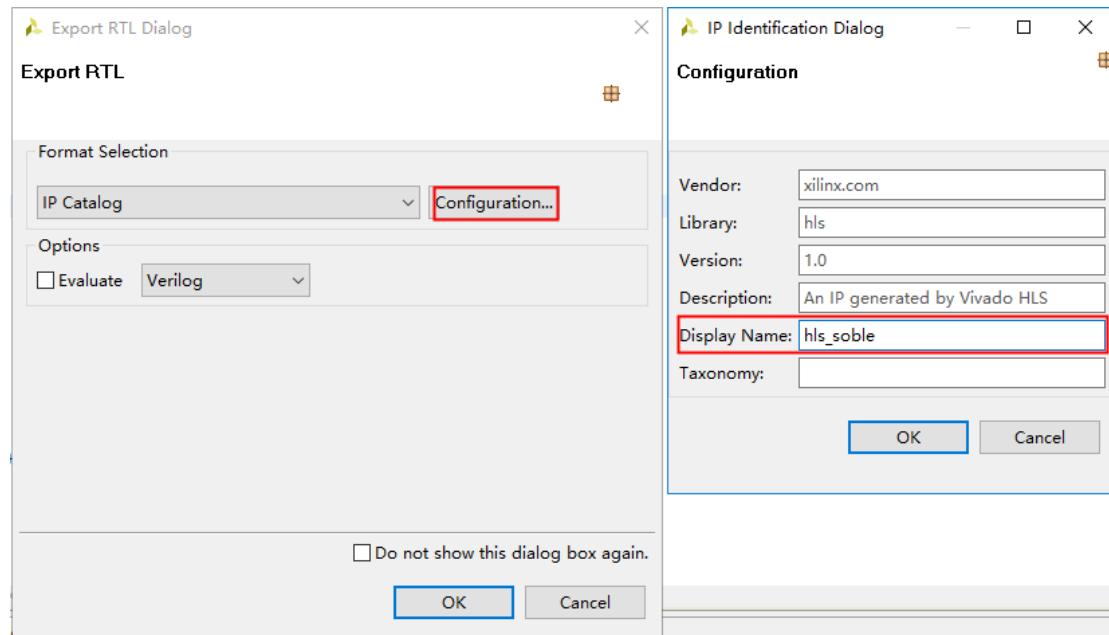


5.2.3 工程封装

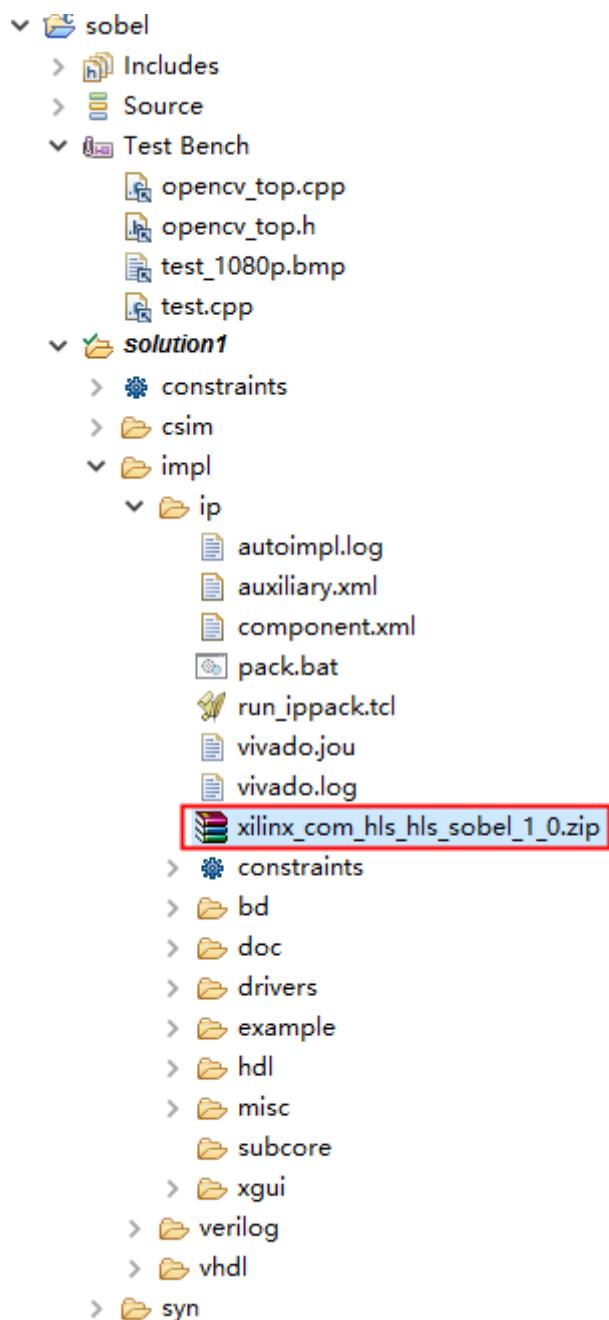
Step1：工程验证无误后，我们就可以开始进行代码封装了。单击 Solution-Export RTL 或直接单击 。



Step2：在弹出来的窗口中，点击 configuration，然后在弹出的窗口中做如下设置。



Step3：单击两次 OK，开始代码封装，最终在 Solution 里面生成了一个 impl 的文件夹，并且在 ip 里面多出了一个压缩包，这就是我们最终生成的 ip。



5.3 代码详解

在这一章我们通过调用 HLS 封装的视频处理库函数进行 Sobel 检测算子的实现，了解自带的处理函数的使用方法。在 HLS 中实现 Sobel 边缘检测的核心函数就下面几行，我们就对这几行代码进行介绍，使大家熟悉这几个函数的使用，为后续开发做好铺垫。

```

RGB_IMAGE img_0(rows, cols);
RGB_IMAGE img_1(rows, cols);
RGB_IMAGE img_2(rows, cols);
RGB_IMAGE img_3(rows, cols);
RGB_IMAGE img_4(rows, cols);
RGB_IMAGE img_5(rows, cols);
RGB_PIXEL pix(50, 50, 50);
```

```

#pragma HLS dataflow
hls::AXIVideo2Mat(INPUT_STREAM, img_0);
hls::Sobel<1,0,3>(img_0, img_1);
hls::SubS(img_1, pix, img_2);
hls::Scale(img_2, img_3, 2, 0);
hls::Erode(img_3, img_4);
hls::Dilate(img_4, img_5);
hls::Mat2AXIVideo(img_5, OUTPUT_STREAM);
```

我们首先用 hls 自带的视频处理库函数 hls::Sobel 进行 X 方向的边缘检测，为什么是 X 方向的？在官方给出的 ug902 技术手册中找到这个函数的介绍，如下图所示：

hls::Sobel

Synopsis

```

template<int XORDER, int YORDER, int SIZE, typename BORDERMODE, int SRC_T, int DST_T,
int ROWS,int COLS,int DROWS,int DCOLS>
void Sobel (
    Mat<ROWS, COLS, SRC_T>& _src,
    Mat<DROWS, DCOLS, DST_T>& _dst)
```



```

template<int XORDER, int YORDER, int SIZE, int SRC_T, int DST_T, int ROWS,int
COLS,int DROWS,int DCOLS>
void Sobel (
    Mat<ROWS, COLS, SRC_T>& _src,
    Mat<DROWS, DCOLS, DST_T>& _dst)
```

Parameters

Table 4-56: Parameters

Parameter	Description
src	Input image
dst	Output image

Description

- Computes a horizontal or vertical Sobel filter, returning an estimate of the horizontal or vertical derivative, using a filter such as:

[-1, 0, 1]

[-2, 0, 2]

[-1, 0, 1]

- SIZE=3, 5, or 7 is supported. This is the same as the equivalent OpenCV function.

- Only XORDER=1 and YORDER=0 (corresponding to horizontal derivative) or XORDER=0 and YORDER=1 (corresponding to a vertical derivative) are supported.

从上图可以得知，此函数的原函数使用了一个模板函数，在我们的程序中只使用了模板函数的前3个参数。Hls::sobel()函数中的两个参数src和dst分别是输入的图像和处理后的图像。

至于为什么是X方向的边缘检测，在图中方框圈出的地方给出了解释。方框中给出的意思为XORDER和YORDER控制是X方向还是Y方向检测。Hls::sobel()函数中这两个参数为1和0，是水平检测，也就是X方向的边缘检测。

模板函数中的第三个参数决定了窗口矩阵的大小，我们可以从图中看到可以是3种窗口，分别是3*3、5*5和7*7。所以整句函数的意思就是在3X3的窗口进行X方向的Sobel检测。当然在很多函数中我们能看到template<typename _T>的身影，这就是C++中模板的声明，建议大家在以后的代码编写中多使用模板，这样便于优化程序，关于这方面的知识自己查阅一下对应的资料，了解这块的开发过程。

进行边缘检测后我们使用hls::Subs进行数组的减法运算，函数原型如下：

```
template<int ROWS, int COLS, int SRC_T, typename _T, int DST_T>
void hls::SubRS (
    hls::Mat<ROWS, COLS, SRC_T>& src,
    hls::Scalar<HLS_MAT_CN(SRC_T), _T>& scl,
    hls::Mat<ROWS, COLS, DST_T>& dst);
```

Parameters

Table 4-59: Parameters

Parameter	Description
src	Input image
scl	Input scalar
dst	Output image
mask	Operation mask, an 8-bit single channel image that specifies elements of the dst image to be computed
dst_ref	Reference image that stores the elements for output image when mask(I) = 0

Description

- Computes the differences between elements of image src and scalar value scl.
- Saves the result in dst.

If computed with mask:

$$\text{dst}(I) = \begin{cases} \text{src}(I)-\text{scl} & \text{if } \text{mask}(I) \neq 0 \\ \text{dst_ref}(I) & \text{if } \text{mask}(I) = 0 \end{cases}$$

- Image data must be stored in src.
- If computed with mask, mask and dst_ref must have data stored.
- The image data of dst must be empty before invoking.
- Invoking this function consumes the data in src and fills the image data of dst.
- If computed with mask, the data of mask and dst_ref are also consumed.
- src and scl must have the same number of channels.
- dst and dst_ref must have the same size and number of channels as src

上图是官方的 hls 指导手册 ug902 对这个函数的描述，从圈出部分可以得知，这个函数是将输入的图像数据 src 与标称值 scl 进行比较，然后将得出的结果给 dst。也就是第二个方框中圈出的公式。在这个公式中可以注意到有一个 mask，这个值一般都不为 0。

因为后面我们需要对图像进行腐蚀膨胀操作，为了避免图像失真，我们先通过 hls::Scalar() 这个函数对图像做线性变换，如果不转换的话，可以发现显示的图像显示偏色。

Description

- Converts an input image src with optional linear transformation.
- Saves the result as image dst.

$$\text{dst}(I) = \text{src}(I) * \text{scale} + \text{shift}$$

- Image data must be stored in src.
- The image data of dst must be empty before invoking.
- Invoking this function consumes the data in src and fills the image data of dst.
- src and dst must have the same size and number of channels. scale and shift must have the same data types.



失真图像

最后我们通过 `hls::Erode()` 和 `hls::Dilate()` 函数分别进行腐蚀和膨胀操作，腐蚀和膨胀是一对相反的操作，腐蚀是对局部求最小值的操作，那么膨胀就是求局部最大值的操作，在数学上解释就是对图像或部分区域对核进行卷积的一个运算。官方对这两个函数的描述及原型如下：

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

- Erodes the image `src` using the specified structuring element constructed within kernel.
- Saves the result in `dst`.
- The erosion determines the shape of a pixel neighborhood over which the maximum is taken, each channel of image `src` is processed independently:

$$\text{dst}(x, y) = \min_{(x', y') : \text{element}(x', y') \neq 0} \text{src}(x + x', y + y')$$

- Image data must be stored in `src`.
- The image data of `dst` must be empty before invoking.
- Invoking this function consumes the data in `src` and fills the image data of `dst`.
- `src` and `dst` must have the same size and number of channels.

```
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);
```

- Each channel of image `src` is processed independently.

$$\text{dst}(x, y) = \max_{(x', y') : \text{element}(x', y') \neq 0} \text{src}(x + x', y + y')$$

- Image data must be stored in `src`.
- The image data of `dst` must be empty before invoking.
- Invoking this function consumes the data in `src` and fills the image data of `dst`.
- `src` and `dst` must have the same size and number of channels.

5.3 本章小结

本章介绍了一种使用 HLS 实现 Sobel 检测的方法，最后通过软件封装得到了一个可以在 VIVADO 上使用的硬件 IP。下一章我们将使用这个 IP，对此 IP 进行功能的验证。

S05_CH06_Sobel 算子硬件实现(二)_硬件验证

6.1 系统硬件设计

这一章我们通过调用 HLS 生成的 Sobel 算子的 IP 搭建一个 Sobel 边缘检测的硬件实现平台，解决大家一直疑惑和关心的如何使用 HLS 生成的 IP。

在搭建硬件系统之前，我们先用 Image2LCD 对一幅图像进行取模，后期我们通过 SDK 来对取模的图像送 SOBEL 处理，然后将处理后的结果和源图像一起显示。首先，我们打开 Image2LCD 软件，如下图所示对 lena.jpg 进行取模（lena.jpg 可在上一章工程文件夹中的 image 文件夹中找到）：



- 1、打开一幅图像
- 2、选择 C 语言数组
- 3、选择 32 位真彩色
- 4、输入宽度和高度（输入完成后，记得一定要点击旁边的三角形按钮，再看底部的提示信息是否正确，如果点击之后仍然底部的输出图像不对，则超过了软件的取模范围，请裁切图片再次取模）。
- 5、选择 32 位彩色
- 6、拖动颜色块，将其如上图所示排列

7、保存取模输出的 C 语言数组

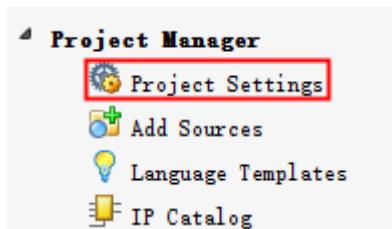
6.2 硬件工程创建

本章节的硬件工程可直接用第四季第一章 Linux 移植的图像显示系统进行修改，节省开发时间。

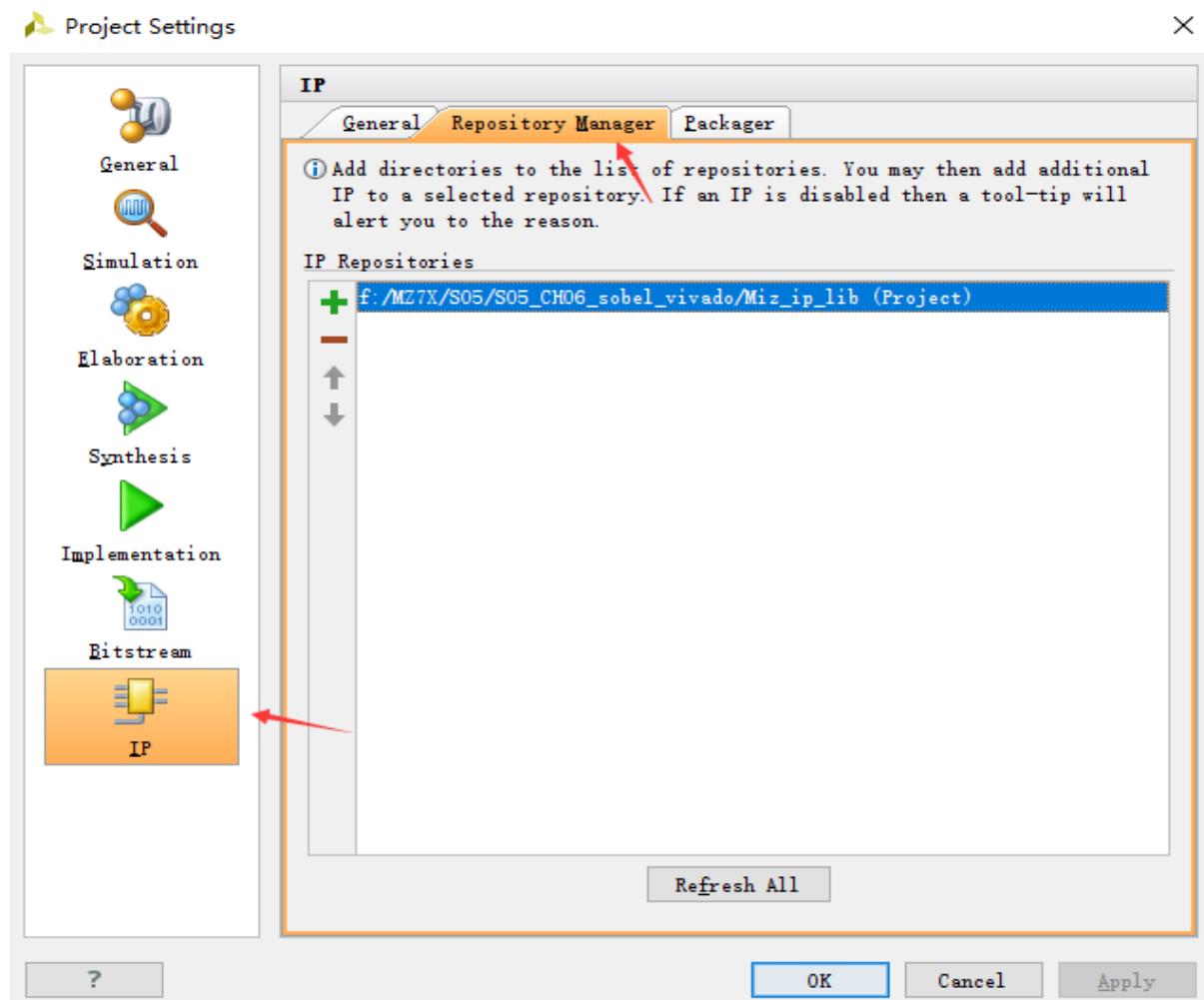
Step1：打开第四季第一章的工程（选择对应自己硬件的工程）。

Step2：双击 BD 文件对其进行一些修改。

Step3：在 Project Manager 设置区单击 Project settings。

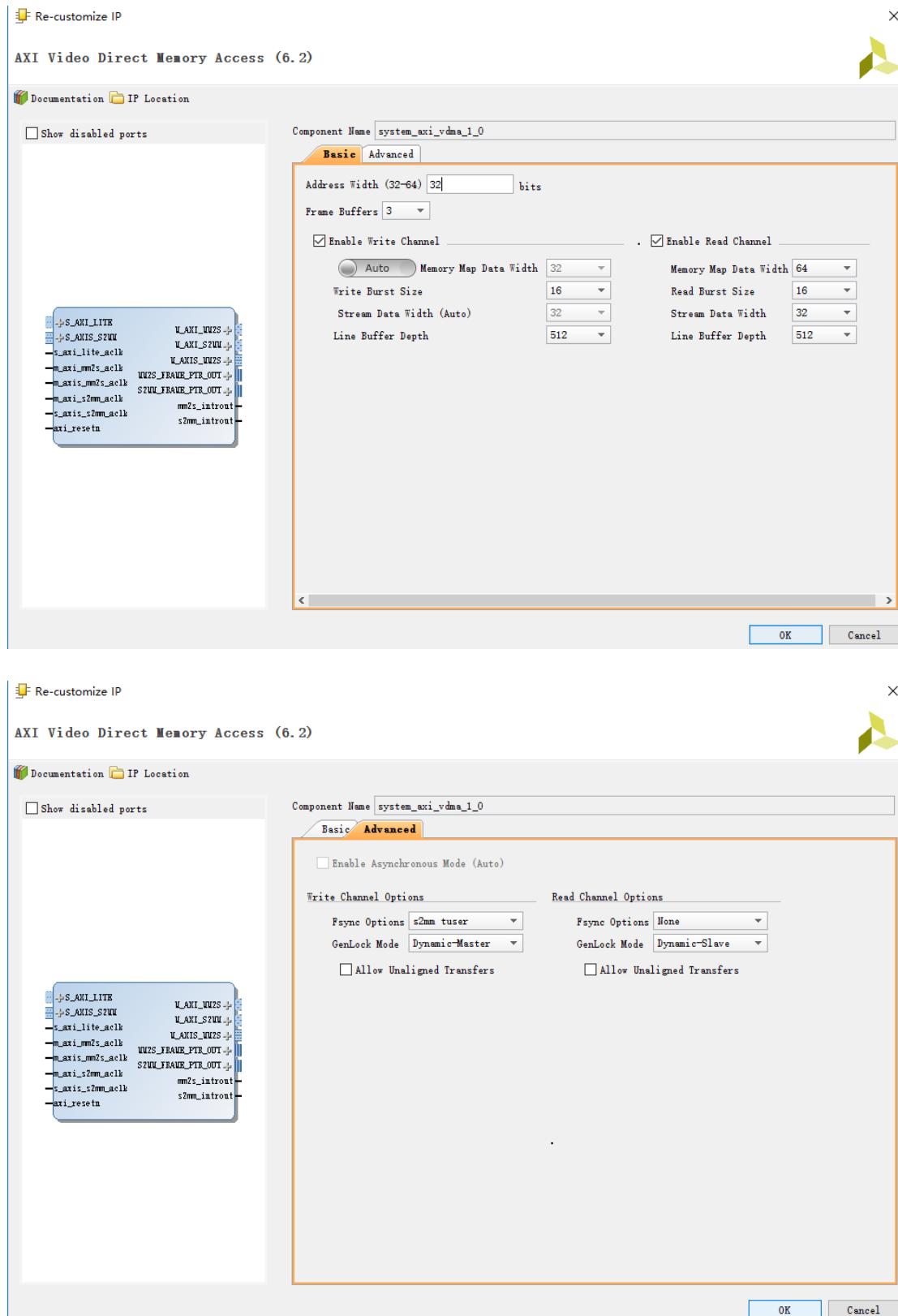


Step4：在弹出的窗口中，如下图设置：



单击图中的加号图标将我们上一章的 SOBEL 的 IP 添加到工程当中。

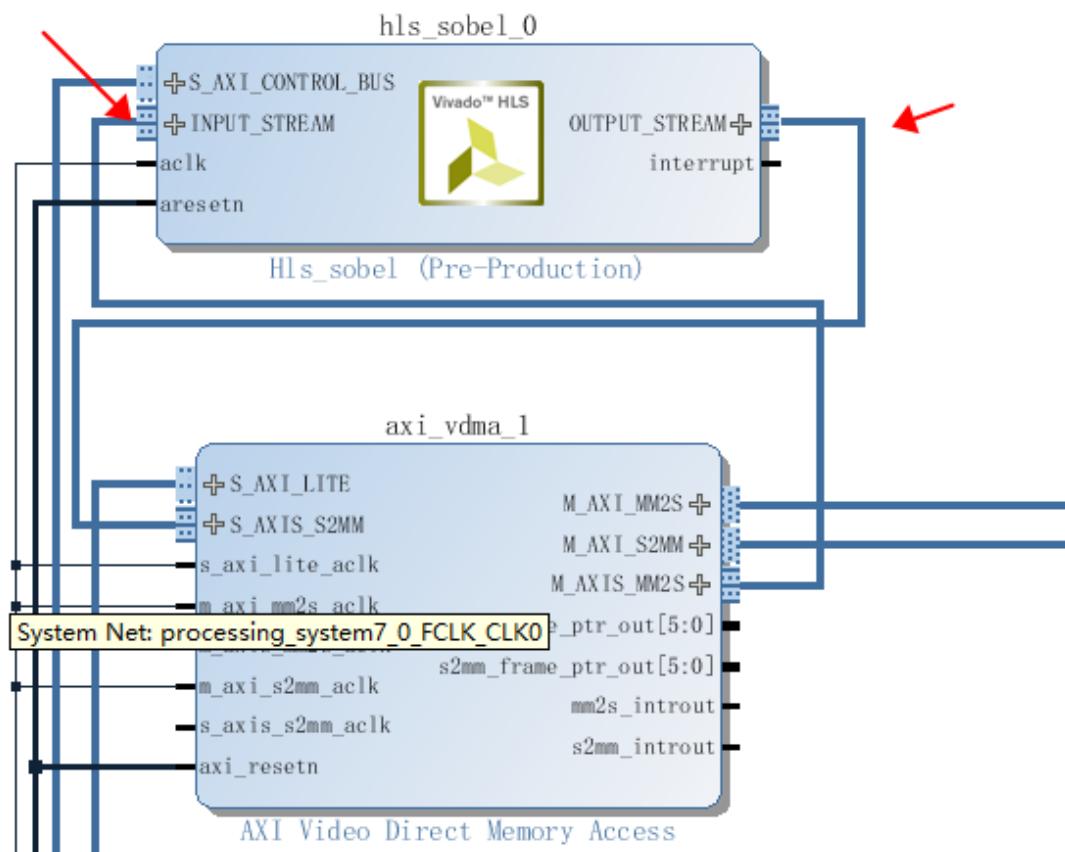
Step5：添加一个V D M A，并如下图所示配置。



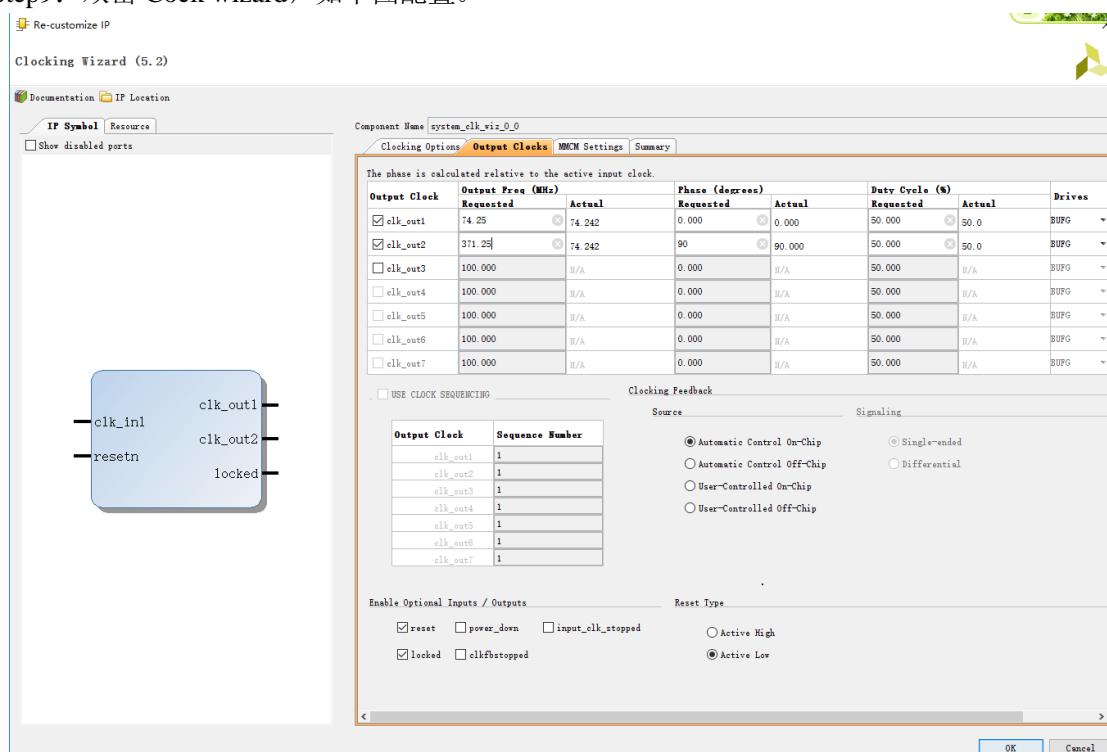
Step6：单击 Run connection Automation 进行自动布线。

Step7：添加 SOBEL IP，然后单击 Run connection Automation。

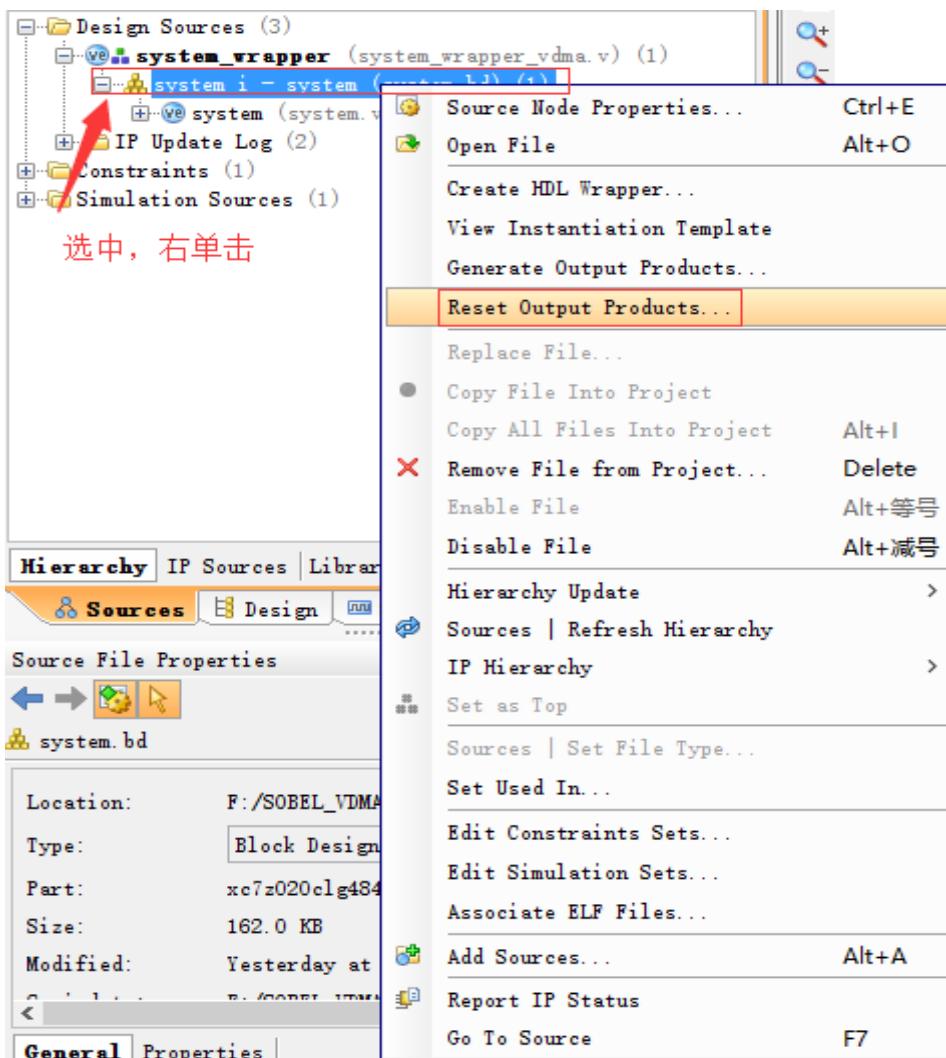
Step8: 按下图所示，完善连线。



Step9: 双击 Clock wizard, 如下图配置。



Step10: 选中 BD 文件，然后右单击，选择 Reset Output Products...。



Step11: 再次选中 BD 文件，选择 Generate Output Products..。

Step12: 选中 BD 文件，然后选择 Create HDL Wrapper。

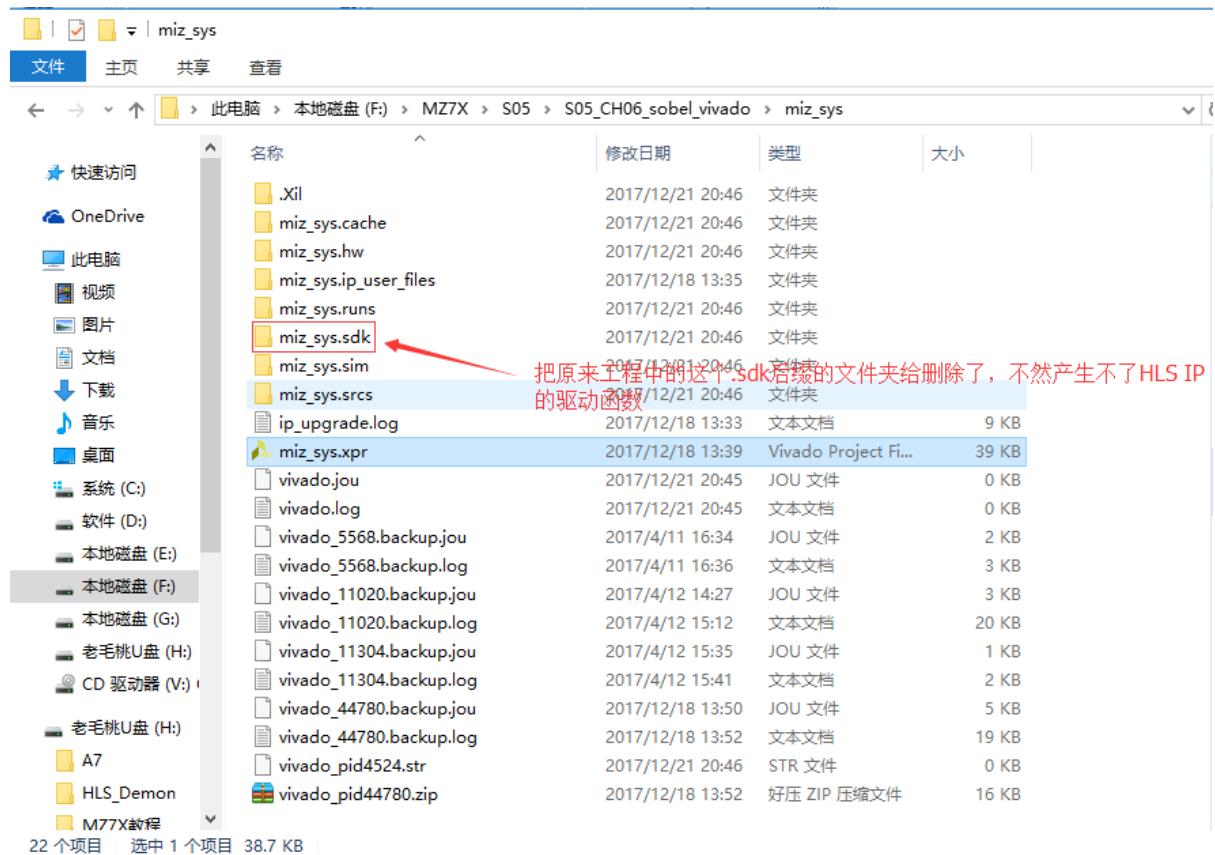
Step13: 单击 生成 Bit 文件。



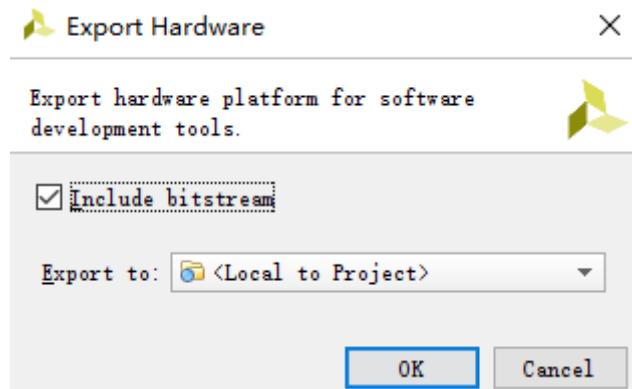
6.3 导入到 SDK

Bit 文件之后，接下来就是修改 SDK，添加相关的 SOBEL 的驱动。

Step1: 在工程文件夹中删除原来的工程的文件夹，这是因为如果不删除原来的 SDK 工程的话，系统就不会自动生成相关 HLS 生成的 IP 自带的驱动，笔者就曾经遇到过这种问题。



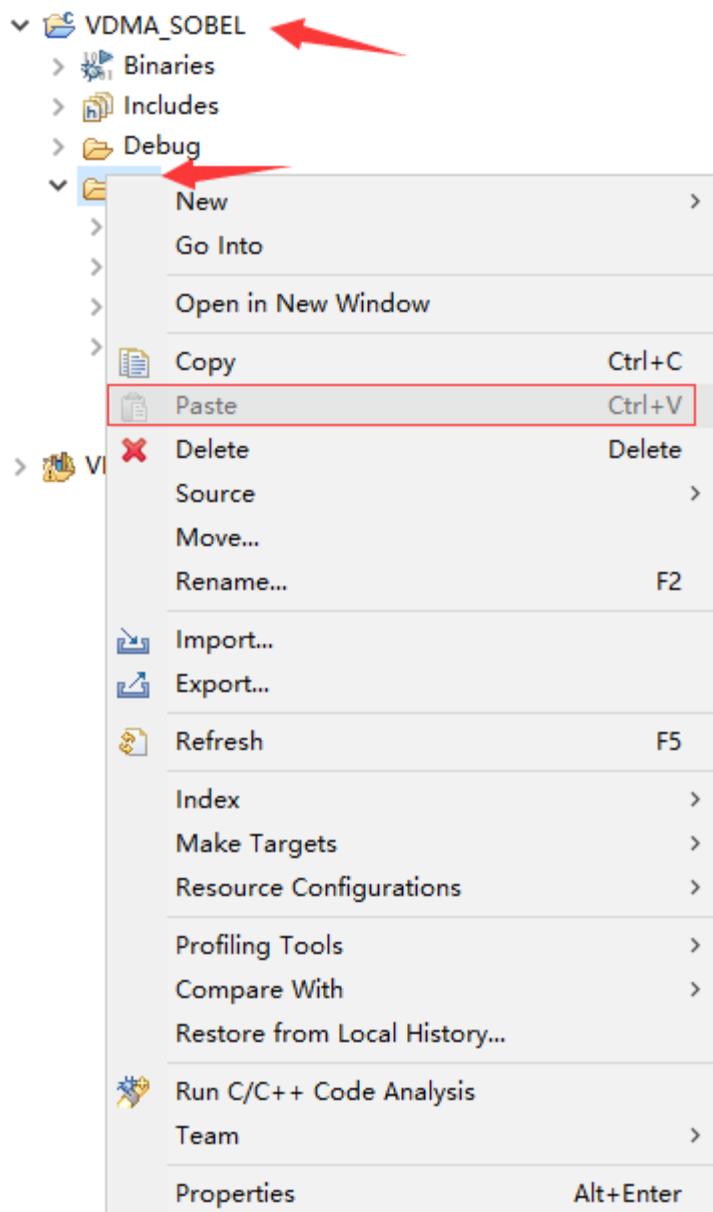
Step2：单击 File-Export-Export Hardware...。



Step2：单击 File-Launch SDK。

Step3：新建一个名为 VDMA_SOBEL 的空的工程。

Step4：复制第一节中生成的图片 C 数组，然后在 SDK 中，选中工程，在 Src 下直接按 Ctrl +V 将其复制到工程中来。



Step5: 双击 main.c，用如下程序进行替换。

```
#include "xaxivdma.h"
#include "xaxivdma_i.h"
#include "xhls_sobel.h"
#include "sleep.h"

#define DDR_BASEADDR          0x00000000
#define DISPLAY_VDMA          XPAR_AXI_VDMA_0_BASEADDR + 0
#define SOBEL_VDMA             XPAR_AXI_VDMA_1_BASEADDR + 0
#define DIS_X                  1280
#define DIS_Y                  720
#define SOBEL_ROW               512
#define SOBEL_COL               512
```

```

#define pi           3.14159265358
#define COUNTS_PER_SECOND (XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ)/64

#define SOBEL_S2MM    0x08000000
#define SOBEL_MM2S    0x0A000000
#define DISPLAY_MM2S  0x0C000000

u32 *BufferPtr[3];
static XHls_sobel sobel;

//函数声明
void Xil_DCacheFlush(void);
// 所有数据格式为RGBA, 低的通道在高位
extern const unsigned char gImage_lena[1048584];

void SOBEL_VDMA_setting(unsigned int width,unsigned int height,unsigned
int s2mm_addr,unsigned int mm2s_addr)
{
    //S2MM
    Xil_Out32(SOBEL_VDMA + 0x30, 0x4); //reset S2MM VDMA Control
Register
    usleep(10);
    Xil_Out32(SOBEL_VDMA + 0x30, 0x0); //genlock
    Xil_Out32(SOBEL_VDMA + 0xAC, s2mm_addr); //S2MM Start Addresses
    Xil_Out32(SOBEL_VDMA + 0xAC+4, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xAC+8, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xA4, width*4); //S2MM Horizontal Size
    Xil_Out32(SOBEL_VDMA + 0xA8, width*4); //S2MM Frame Delay and
Stride
    Xil_Out32(SOBEL_VDMA + 0x30, 0x3); //S2MM VDMA Control Register
    Xil_Out32(SOBEL_VDMA + 0xA0, height); //S2MM Vertical Size start
an S2M
    // Xil_DCacheFlush();

    //MM2S
    Xil_Out32(SOBEL_VDMA + 0x00,0x00000003); // enable circular mode
    Xil_Out32(SOBEL_VDMA + 0x5c,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x60,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x64,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x58,(width*4)); // h offset
    Xil_Out32(SOBEL_VDMA + 0x54,(width*4)); // h size
    Xil_Out32(SOBEL_VDMA + 0x50,height); // v size
    //Xil_DCacheFlush();
}

```

```
}
```

```
void DISPLAY_VDMA_setting(unsigned int width,unsigned height,unsigned int mm2s_addr)
{
    Xil_Out32((DISPLAY_VDMA + 0x000), 0x00000003); // enable
circular mode
    Xil_Out32((DISPLAY_VDMA + 0x05c), mm2s_addr); // start address
    Xil_Out32((DISPLAY_VDMA + 0x060), mm2s_addr); // start address
    Xil_Out32((DISPLAY_VDMA + 0x064), mm2s_addr); // start address
    Xil_Out32((DISPLAY_VDMA + 0x058), (width*4)); // h offset
(640 * 4) bytes
    Xil_Out32((DISPLAY_VDMA + 0x054), (width*4)); // h size
(640 * 4) bytes
    Xil_Out32((DISPLAY_VDMA + 0x050), height); // v size
(480)
}

void SOBEL_DDRWR(unsigned int addr,unsigned int cols,unsigned int lows)
{
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    for(i=0;i<cols;i++)
    {
        for(j=0;j<lows;j++)
        {
            b= gImage_lena[(j+i*cols)*4+1];
            g= gImage_lena[(j+i*cols)*4+2];
            r= gImage_lena[(j+i*cols)*4+3];

            Xil_Out32((addr+(j+i*cols)*4),((r<<24)|(g<<16)|(b<<8)|0x0));
        }
    }
    Xil_DCacheFlush();
}

void SOBEL_Setup()
{
//const int cols = 512;
//const int rows = 512;
XHls_sobel_SetRows(&sobel, SOBEL_COL);
XHls_sobel_SetCols(&sobel, SOBEL_ROW);
```

```
XHls_sobel_DisableAutoRestart(&sobel);
XHls_sobel_InterruptGlobalDisable(&sobel);
SOBEL_VDMA_setting(SOBEL_ROW, SOBEL_COL, SOBEL_S2MM, SOBEL_MM2S);
SOBEL_DDRWR(SOBEL_MM2S, SOBEL_ROW, SOBEL_COL);
XHls_sobel_Start(&sobel);
}

void show_img(u32 x, u32 y, u32 disp_base_addr, const unsigned char * addr,
u32 size_x, u32 size_y, u32 type)
{
    //读图片左上角标
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    u32 start_addr=disp_base_addr;
    start_addr = disp_base_addr + 4*x + y*4*DIS_X;
    for(j=0;j<size_y;j++)
    {
        for(i=0;i<size_x;i++)
        {
            if(type==0)
            {
                b = *(addr+(i+j*size_x)*4+2); //08
                g = *(addr+(i+j*size_x)*4+1); //60
                r = *(addr+(i+j*size_x)*4); //01
            }
            else
            {
                b = *(addr+(i+j*size_x)*4+1); //08
                g = *(addr+(i+j*size_x)*4+2); //60
                r = *(addr+(i+j*size_x)*4+3); //01
            }

            Xil_Out32((start_addr+(i+j*DIS_X)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
        }
    }
    Xil_DCacheFlush();
}

int main(void)
{
```

```

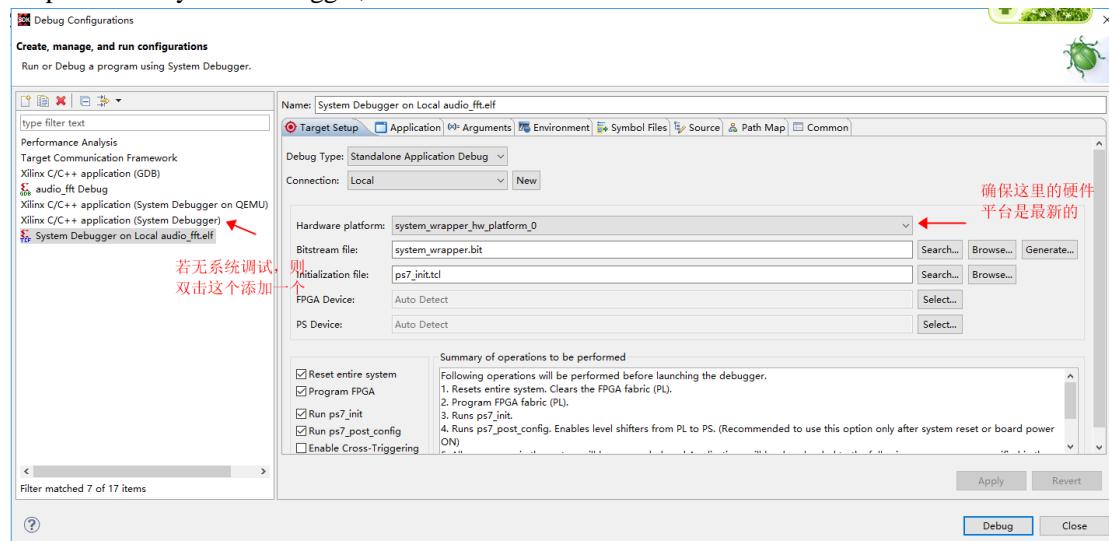
//Xil_DCacheFlush();
xil_printf("Starting the first VDMA \n\r");
int status = XHls_sobel_Initialize(&sobel,
XPAR_HLS_SOBEL_0_S_AXI_CONTROL_BUS_BASEADDR);
if(0 != status)
{
    xil_printf("XHls_Sobel_Initialize failed \n");
}
SOBEL_Setup();
DISPLAY_VDMA_setting(DIS_X,DIS_Y,DISPLAY_MM2S);

while(1)
{
    show_img(0,0,DISPLAY_MM2S,(void*)SOBEL_S2MM,512,512,0);
    show_img(522,0,DISPLAY_MM2S,(void*)SOBEL_MM2S,512,512,1);
}
return 0;
}

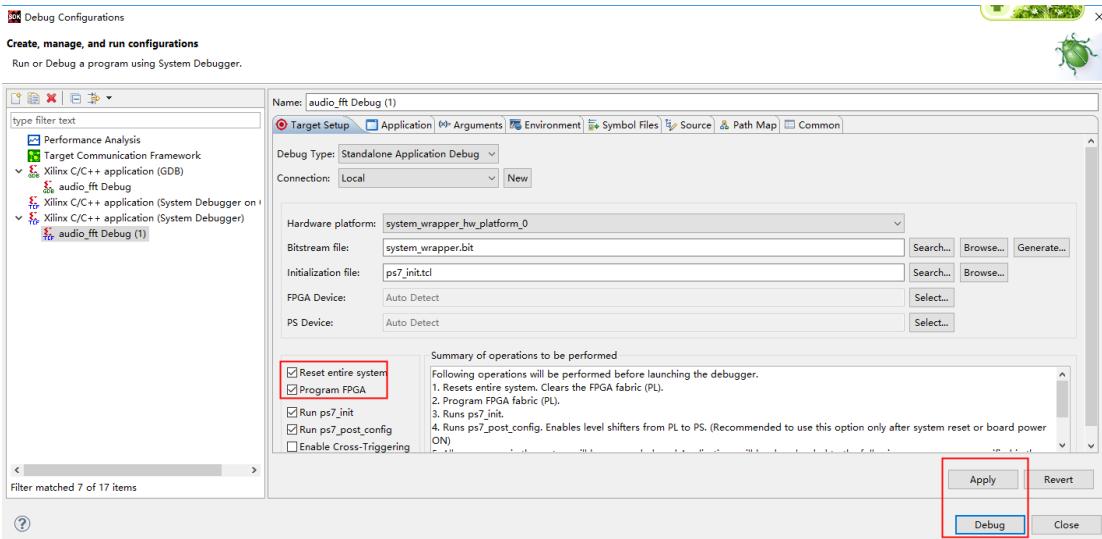
```

Step6: 右击工程，选择 Debug as ->Debug configuration。

Step7: 选中 system Debugger,双击创建一个系统调试。



Step8: 设置系统调试。



打开系统自带的窗口调试助手，点击运行按钮开始运行程序。



程序运行之后，运行结果如下图所示：



在图中，左边部分是 SOBEL 处理之后的结果，右边是未经处理的源图像，在上图中我们看到经过 SOBEL 之后，图像的边缘被检测出来了。

6.4 程序分析

本章的程序比较简单，HLS 的驱动使用的都是官方的驱动，大家在使用的时候照葫芦画瓢即可。这里重点介绍一下本章的 VDMA 配置。在本章的硬件工程中，使用了两个 VDMA，其中 VDMA 只有一个

写通道，负责完成显示的缓存功能，VDMA1 有两个通道，负责将我们取模的数组送入内存和 SOBEL 处理后的数据缓存的作用。因此，在本章中一共使用了三个 VDMA 的通道，因此在本章中我们设置了三块内存，分别存放这三个通道的数据，如下图所示：

```
#define SOBEL_S2MM      0x08000000
#define SOBEL_MM2S      0x0A000000
#define DISPLAY_MM2S    0x0C000000
```

接下来就是对这两个 VDMA 进行配置，这里就比如说配置 VDMA，其配置如下：

```
void SOBEL_VDMA_setting(unsigned int width,unsigned int height,unsigned int s2mm_addr,unsigned int mm2s_addr)
{
    //S2MM
    Xil_Out32(SOBEL_VDMA + 0x30, 0x4); //reset S2MM VDMA Control Register
    usleep(10);
    Xil_Out32(SOBEL_VDMA + 0x30, 0x0); //genlock
    Xil_Out32(SOBEL_VDMA + 0xAC, s2mm_addr); //S2MM Start Addresses
    Xil_Out32(SOBEL_VDMA + 0xAC+4, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xAC+8, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xA4, width*4); //S2MM Horizontal Size
    Xil_Out32(SOBEL_VDMA + 0xA8, width*4); //S2MM Frame Delay and Stride
    Xil_Out32(SOBEL_VDMA + 0x30, 0x3); //S2MM VDMA Control Register
    Xil_Out32(SOBEL_VDMA + 0xA0, height); //S2MM Vertical Size start an S2M
    // Xil_DCacheFlush();

    //MM2S
    Xil_Out32(SOBEL_VDMA + 0x00, 0x00000003); // enable circular mode
    Xil_Out32(SOBEL_VDMA + 0x5c, mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x60, mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x64, mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x58, (width*4)); // h offset
    Xil_Out32(SOBEL_VDMA + 0x54, (width*4)); // h size
    Xil_Out32(SOBEL_VDMA + 0x50, height); // v size
    //Xil_DCacheFlush();
}
```

这里的配置比较简单，就是往 VDMA 的寄存器中写入相应的值即可，在程序中都给出了对应的注释，分析的方法也在前面的教程中进行了介绍，这里不再反复的去讲解。直接看到以下程序部分：

```
show_img(0,0,DISPLAY_MM2S,(void*)SOBEL_S2MM,512,512);
show img(512,0,DISPLAY_MM2S,(void*)SOBEL_MM2S,512,512);
```

这个程序就是我们本章程序的描点程序，我们看看具体的函数定义：

```
void show_img(u32 x, u32 y, u32 disp_base_addr, const unsigned char * addr, u32 size_x, u32 size_y)
{
    //计算图片左上角坐标
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    u32 start_addr=disp_base_addr;
    start_addr = disp_base_addr + 4*x + y*4*DIS_X;
    for(j=0;j<size_y;j++)
    {
        for(i=0;i<size_x;i++)
        {
            //if(type==0)
            //{
                //b = *(addr+(i+j*size_x)*4+2); //08
                //g = *(addr+(i+j*size_x)*4+1); //60
                //r = *(addr+(i+j*size_x)*4); //01
            //}
            //else
            //{
                b = *(addr+(i+j*size_x)*4); //08
                g = *(addr+(i+j*size_x)*4+1); //60
                r = *(addr+(i+j*size_x)*4+2); //01
            //}
            Xil_Out32((start_addr+(i+j*DIS_X)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
        }
        Xil_DCacheFlush();
    }
}
```

在这里，前两个参数是描点的起始位置，`disp_base_addr` 是要写入的位置，`addr` 是写入的数据的来源，之后两个是图像的尺寸大小，这个程序比较简单，就是从 `addr` 中提取数据，然后抽取出三原色数据，再重新排列，最后写入到对应的内存当中。`Xil_DcacheFlush()` 函数是一个刷内存函数，

负责将 SDK 中写入的数据刷入内存当中。本章中还有一个与之类似的程序，如下图所示：

```
void SOBEL_DDRWR(unsigned int addr,unsigned int cols,unsigned int lows)
{
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    for(i=0;i<cols;i++)
    {
        for(j=0;j<lows;j++)
        {
            b= gImage_lena[(j+i*cols)*4+1];
            g= gImage_lena[(j+i*cols)*4+2];
            r= gImage_lena[(j+i*cols)*4+3];
            Xil_Out32((addr+(j+i*cols)*4),((r<<24)|(g<<16)|(b<<8)|0x0));
        }
    }
    Xil_DCacheFlush();
}
```

这个函数顾名思义，也是一个 SOBEL 往 DDR 中写入寄存器的函数，写入的数据就是取模的图片的数据。回到描点函数的分析中，接下来我们来看看描点函数是怎么被调用的。

```
show_img(0,0,DISPLAY_MM2S,(void*)SOBEL_S2MM,512,512);
show_img(522,0,DISPLAY_MM2S,(void*)SOBEL_MM2S,512,512);
```

结合上面的讲解，可以得知，这里写的数据来源来自 SOBEL_S2MM（也就是 VDMA1 的读通道），要写入的地址是 DISPLAY_MM2S（也就是 VDMA0 的写通道），整个框架下来就是 VDMA1 将取模的数组刷入内存，通过 VDMA1 的写通道送给 SOBEL 处理，然后将处理后的结果放在 VDMA1 的读通道中，然后再操作 VDMA0 的写通道将 VDMA1 S2MM 里 SOBEL 的结果显示在屏幕上，思想比较简单易懂，这里的划分内存操作的方式是大家在 VDMA 使用中一种很常用的方法，大家可以深入的了解一下。

6.5 本章小结

本章介绍了上一章生成的 SOBEL IP 如何进行调用，如何正确的驱动 HLS 生成的 IP，另外本章还介绍了一种全新的 VDMA 的操作方式，通过划分内存的方式对图片进行显示，这种方法可以延伸到以后的算法处理当中，是一种很方便的操作方式。

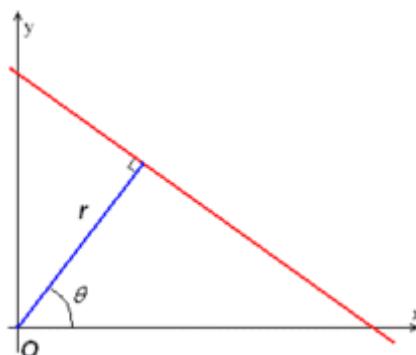
S05_CH07_基于 Hough 变换的圆检测

7.1 Hough 变换原理介绍

霍夫变换 (Hough Transform) 是图像处理中的一种特征提取技术，该过程在一个参数空间中通过计算累计结果的局部最大值得到一个符合该特定形状的集合作为霍夫变换结果。霍夫变换于 1962 年由 Paul Hough 首次提出，后于 1972 年由 Richard Duda 和 Peter Hart 推广使用，经典霍夫变换用来检测图像中的直线，后来霍夫变换扩展到任意形状物体的识别，多为圆和椭圆。霍夫变换运用两个坐标空间之间的变换将在一个空间中具有相同形状的曲线或直线映射到另一个坐标空间的一个点上形成峰值，从而把检测任意形状的问题转化为统计峰值问题。

7.1.1 Hough 变换直线检测

我们知道，一条直线在直角坐标系下可以用 $y=kx+b$ 表示，霍夫变换的主要思想是将该方程的参数和变量交换，即用 x, y 作为已知量 k, b 作为变量坐标，所以直角坐标系下的直线 $y=kx+b$ 在参数空间表示为点 (k, b) ，而一个点 (x_1, y_1) 在直角坐标系下表示为一条直线 $y_1=x_1 \cdot k+b$ ，其中 (k, b) 是该直线上的任意点。为了计算方便，我们将参数空间的坐标表示为极坐标下的 γ 和 θ 。因为同一条直线上的点对应的 (γ, θ) 是相同的，因此可以先将图片进行边缘检测，然后对图像上每一个非零像素点，在参数坐标下变换为一条直线，那么在直角坐标下属于同一条直线的点便在参数空间形成多条直线并内交于一点。因此可用该原理进行直线检测。



参数空间变换结果

如图所示，对于原图内任一点 (x, y) 都可以在参数空间形成一条直线，以图中一条直线为例有参数 $(\gamma, \theta)=(69.641, 30^\circ)$ ，所有属于同一条直线上的点会在参数空间交于一点，该点即为对应直线的参数。

7.1.2 Hough 变换圆检测

继使用 hough 变换检测出直线之后，顺着坐标变换的思路，提出了一种检测圆的方法。

1 如何表示一个圆？

与使用 (r, θ) 来表示一条直线相似，使用 (a, b, r) 来确定一个圆心为 (a, b) 半径为 r 的圆。

2 如何表示过某个点的所有圆？

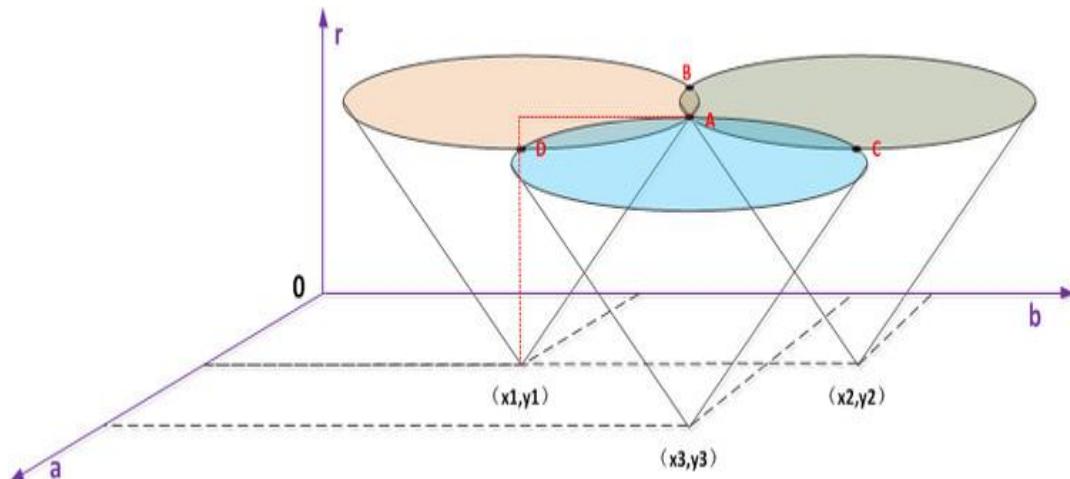
某个圆过点 (x_1, y_1) ，则有： $(x_1-a)^2 + (y_1-b)^2 = r^2$ 。

那么过点 (x_1, y_1) 的所有圆可以表示为 $(a_1(i), b_1(i), r_1(i))$ ，其中 $r_1 \in (0, \infty)$ ，每一个 i 值都对应一个不同的圆， $(a_1(i), b_1(i), r_1(i))$ 表示了无穷多个过点 (x_1, y_1) 的圆。

3 如何确定多个点在同一个圆上？

如(2)中说明，过点 (x_1, y_1) 的所有圆可以表示为 $(a_1(i), b_1(i), r_1(i))$ ，过点 (x_2, y_2) 的所有圆可以表示为 $(a_2(i), b_2(i), r_2(i))$ ，过点 (x_3, y_3) 的所有圆可以表示为 $(a_3(i), b_3(i), r_3(i))$ ，如果这三个点在同一个圆上，那么存在一个值 (a_0, b_0, r_0) ，使得 $a_0 = a_1(k) = a_2(k) = a_3(k)$ 且 $b_0 = b_1(k) = b_2(k) = b_3(k)$ 且 $r_0 = r_1(k) = r_2(k) = r_3(k)$ ，即这三个点同时在圆 (a_0, b_0, r_0) 上。

从下图可以形象的看出：



首先，分析过点 (x_1, y_1) 的所有圆 $(a_1(i), b_1(i), r_1(i))$ ，当确定 $r_1(i)$ 时， $(a_1(i), b_1(i))$ 的轨迹是一个以 $(x_1, y_1, r_1(i))$ 为圆心半径为 $r_1(i)$ 的圆。那么，所有圆 $(a_1(i), b_1(i), r_1(i))$ 的组成了一个以 $(x_1, y_1, 0)$ 为顶点，锥角为 90 度的圆锥面。

三个圆锥面的交点 A 既是同时过这三个点的圆。

7.1.3 Hough 变换圆检测算法实现流程

Hough 变换时一种利用图像的全局特征将特定形状边缘链接起来。它通过点线的对偶性，将源图像上的点影射到用于累加的参数空间，把原始图像中给定曲线的检测问题转化为寻找参数空间中的峰值问题。由于利用全局特征，所以受噪声和边界间断的影响较小，比较鲁棒。

Hough 变换思想为：在原始图像坐标系下的一个点对应了参数坐标系中的一条直线，同样参数坐标系的一条直线对应了原始坐标系下的一个点，然后，原始坐标系下呈现直线的所有点，它们的斜率和截距是相同的，所以它们在参数坐标系下对应于同一个点。这样在将原始坐标系下的各个点投影到参数坐标系下之后，看参数坐标系下有没有聚集点，这样的聚集点就对应了原始坐标系下的直线。

因此采用 hough 变换主要有以下几个步骤：

1) Detect the edge

检测得到图像的边缘

2) Create accumulator

采用二维向量描述图像上每一条直线区域，将图像上的直线区域计数器映射到参数空间中的存储单元， ρ 为直线区域到原点的距离，所以对于对角线长度为 n 的图像， ρ 的取值范围为 $(0, n)$ ， θ 值得取值范围为 $(0, 360)$ ，定义为二维数组 $\text{HoughBuf}[n][360]$ 为存储单元。

对所有像素点 (x, y) 在所有 θ 角的时候，求出 ρ 。从而累加 ρ 值出现的次数。高于某个阈值的 ρ 就是一个直线。这个过程就类似于横坐标是 θ 角， ρ 就是到直线的最短距离。横坐标 θ 不断变换，根据直线方程公司， $\rho = x\cos\theta + y\sin\theta$ 对于所有的不为 0 的像素点，计算出 ρ ，找到 ρ 在坐标 (θ, ρ) 的位置累加 1。

3) Detect the peaks, maximal in the accumulator

通过统计特性，假如图像平面上有两条直线，那么最终会出现 2 个峰值，累加得到最高的数组的值为所求直线参数。

7.2 Hough 在 HLS 上的实现

我们看下面一个实际问题：我们要从一副图像中检测出半径以知的圆形来。我们可以取和图像平面一样的参数平面，以图像上每一个前景点为圆心，以已知的半径在参数平面上画圆，并把结果进行累加。最后找出参数平面上的峰值点，这个位置就对应了图像上的圆心。在这个问题里，图像

平面上的每一点对应到参数平面上的一个圆。

把上面的问题改一下，假如我们不知道半径的值，而要找出图像上的圆来。这样，一个办法是把参数平面扩大称为三维空间。就是说，参数空间变为 $x - y - R$ 三维，对应圆的圆心和半径。图像平面上的每一点就对应于参数空间中每个半径下的一个圆，这实际上是一个圆锥。最后当然还是找参数空间中的峰值点。不过，这个方法显然需要大量的存储空间，运行速度也会是很大问题。

那么有什么比较好的解决方法么？我们前面假定的图像都是黑白图像（二值图像），实际上这些二值图像多是彩色或灰度图像通过边缘提取来的。我们前面提到过，图像边缘除了位置信息，还有方向信息也很重要，这里就用上了。根据圆的性质，圆的半径一定在垂直于圆的切线的直线上，也就是说，在圆上任意一点的法线上。这样，解决上面的问题，我们仍采用 2 维的参数空间，对于图像上的每一前景点，加上它的方向信息，都可以确定出一条直线，圆的圆心就在这条直线上。这样一来，问题就会简单了许多。

接下来我们来设计利用 Hough 变换来进行圆检测。

7.2.1 工程创建

Step1：打开 HLS，按照之前介绍的方法，创建一个新的工程，命名为 Hough。

Step2：右单击 Source 选项，选择 New File，创建一个名为 Top.cpp 的文件。



Step3：在打开的编辑区中，把下面的程序拷贝进去：

```
#include "top.h"
#include <stdio.h>
#include <iostream>

using namespace std;

void hls::hls_hough_line(GRAY_IMAGE &src,GRAY_IMAGE &dst,int
rows,int cols)
{
    GRAY_PIXEL result;
    int row,col,k;
```

```
//参数间的依赖(a,b)半径radius
int a = 0,b = 0,radius = 0;

//累加器
int A0 = rows;
int B0 = cols;

//注意HLS不支持数组 所以这里直接将数据读入
const int Size = 1089900;//Size = rows*cols*(120-110);

#ifndef __SYNTHESIS__
    int _count[Size];
    int *count = &_count[0];
#else
    int *count =(int *) malloc(Size * sizeof(int));
#endif

//偏移
int index ;

//为累加器赋值
for (row = 0;row < Size;row++)
{
    count[row] = 0;
}

GRAY_PIXEL src_data;
uchar temp0;
for (row = 0; row< rows;row++)
{
    for (col = 0; col< cols;col++)
    {
        src >> src_data;
        uchar temp = src_data.val[0];
        //检测黑线
        if (temp == 0)
        {
            //遍历 ,b 为 累加器值
            for (a = 0;a < A0;a++)
            {
                for (b = 0;b < B0;b++)
                {
                    radius = (int)(sqrt(pow((double)(row-a),2) +
```

```
pow((double)(col - b), 2));
    if(radius > 110 && radius < 120)
    {
        index = A0 * B0 *(radius-110) + A0*b + a;
        count[index]++;
    }
}
}
}
}

//遍历数组 找出所有圆
for (a = 0 ; a < A0 ; a++)
{
    for (b = 0 ; b < B0; b++)
    {
        for (radius = 110 ; radius < 120; radius++)
        {
            index = A0 * B0 *(radius-110) + A0*b + a;
            if (count[index] > 210)
            {
                //在image2中绘制圆
                for(k = 0; k < rows;k++)
                {
                    for (col = 0 ; col< cols;col++)
                    {
                        //x有两值 根据公式(x-a)^2+(y-b)^2=r^2得到
                        int temp =
(int) (sqrt(pow((double)radius,2)- pow((double)(col-b),2)));
                        int x1 = a + temp;
                        int x2 = a - temp;
                        if ( (k == x1) || (k == x2) ){
                            result.val[0] = (uchar)255;
                        }
                        else{
                            result.val[0] = (uchar)0;
                        }
                        dst << result;
                    }
                }
            }
        }
    }
}
```

```

    }
}

void hls_hough(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int rows,
int cols)
{
    GRAY_IMAGE img_src(rows, cols);
    GRAY_IMAGE img_dst(rows, cols);

    hls::AXIVideo2Mat(src_axi, img_src);
    hls::hls_hough_line(img_src, img_dst, rows, cols);
    hls::Mat2AXIVideo(img_dst, dst_axi);
}

```

代码中有一段非常需要注意的如下几行代码，这段代码中使用了动态分配内存的函数malloc，另外HLS中不支持C++中使用new关键字分配内存的方法。该函数只能用于在综合文件中进行仿真，主要是解决大数组所造成的的编译出问题，详细介绍可参考ug902关于Array的章节。

```

#ifdef __SYNTHESIS__
int _count[Size];
int *count = &_count[0];
else
int *count =(int *) malloc(Size * sizeof(int));
#endif

```

另外需要注意的是malloc 函数返回的是 void * 类型。对于C++，如果你写成：

int *count = **malloc**(Size * **sizeof(int)**);则程序无法通过编译，报错：“不能将 void* 赋值给 int * 类型变量”。所以必须通过 (**int** *) 来将强制转换。而对于C，没有这个要求，但为了使C程序更方便的移植到C++中来，建议养成强制转换的习惯。

Step4：再在 Source 中添加一个名为 Top.h 的库函数，并添加如下程序：

```

#ifndef _TOP_H_
#define _TOP_H_

//#include "iostream"
#include "hls_video.h"
#include "math.h"

//maximum image size
#define MAX_WIDTH 1920
#define MAX_HEIGHT 1080

```

```

//I/O Image Settings
#define INPUT_IMAGE           "circle2.bmp"
#define OUTPUT_IMAGE          "result.jpg"
#define OUTPUT_IMAGE_GOLDEN   "result_golden.jpg"

//typedef video library core structures
typedef hls::stream<ap_axiu<32,1,1,1> >
AXI_STREAM;
typedef hls::Scalar<3, unsigned char>           RGB_PIXEL;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> RGB_IMAGE;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1> GRAY_IMAGE;
typedef hls::Scalar<1, unsigned char>           GRAY_PIXEL;
typedef unsigned char uchar;

//定义点的结构体
template<typename T>
struct hls_Point
{
    T x;
    T y;
};

//定义命名空间hls并进行函数声明
namespace hls
{
    double hls_round(double x, int n);
    void hls_hough_line(GRAY_IMAGE &src,GRAY_IMAGE &dst,int
rows,int cols);
}

//top level function for HW synthesis
void hls_hough(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int rows,
int cols);

#endif

```

Step5: 在 Test Bench 中, 用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码:

```

#include <iostream>
#include "hls_opencv.h"
#include "top.h"
#include "opencv_top.h"

using namespace cv;

```

```

using namespace std;

int main()
{
    //加载图像
    IplImage* src =
cvLoadImage(INPUT_IMAGE, CV_LOAD_IMAGE_GRAYSCALE); //加载真图片并转换为灰度
    图像

    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels);
    cvShowImage("src", src);

    //使用HLS库进行处理
    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIVideo(src, src_axi);
    hls_hough(src_axi, dst_axi, src->height, src->width);
    AXIVideo2IplImage(dst_axi, dst);
    cvShowImage("hls_dst", dst);

    //使用OPENCV库进行处理
    opencv_hough(src, dst);
    cvShowImage("cv_dst", dst);

    waitKey(0);

    //释放内存
    cvReleaseImage(&dst);
    return 0;
}

```

Step6：用同样的方法，再在 Test Bench 中创建一个 opencv_top.cpp 和 Opencv_top.h 文件，添加如下程序：

Opencv_top.cpp 代码如下：

```

#include "opencv_top.h"
#include "top.h"

using namespace cv;
using namespace std;

void opencv_hough(IplImage* src, IplImage* dst)
{
    int i, j;

```

```
unsigned char *ptr, *dst_data;
//参数的参数圆心(a,b)半径radius
int a = 0, b = 0, radius = 0;
//累加器
int A0 = src->height;
int B0 = src->width;
int R0 = (src->width > src->height)? 2*src->width :
2*src->height;//R0取两者最大值的倍数

int countLength = A0*B0*R0;
int *count = new int[countLength];
//偏移
int index = A0 * B0 *radius + A0*b + a;
//累加器值
for (i= 0;i<countLength;i++)
{
    count[i]=0;
}

for (i = 0 ; i< src->height; i++)
{
    for (j = 0 ; j< src->width ; j++)
    {
        ptr = (unsigned char *)src->imageData +
i*src->widthStep + j;
        //检测越界
        if (*ptr == 0 )
        {
            //遍历a,b为累加器值
            for (a = 0 ; a< A0;a++)
            {
                for (b = 0; b< B0;b++)
                {
                    radius =
(int)(sqrt(pow((double)(i-a),2) + pow((double)(j - b),2)));
                    index = A0 * B0 *radius + A0*b + a;
                    count[index]++;
                }
            }
        }
    }
}
//image2全部赋值为
```

```
for (i= 0; i<dst->height; i++)
{
    for (j = 0 ; j< dst->width;j++)
    {
        dst_data = (unsigned char *)dst->imageData +
i*dst->widthStep + j;
        *dst_data = 0;
    }
}

//遍历数组 找出所有圆
for (a = 0 ; a < A0 ; a++)
{
    for (b = 0 ; b< B0; b++)
    {
        for (radius = 0 ; radius < R0; radius++)
        {
            index = A0 * B0 *radius + A0*b + a;
            if (count[index] > 200) //值不同会跳过该数据
            {
                //在image2中绘制圆
                for (j = 0 ; j< src->width;j++)
                {
                    i = (int)(sqrt(pow((double)radius,2)-
pow((double)(j-b),2)) + a);
                    if ((i< dst->height) && (i >= 0))
                    {
                        dst_data = (unsigned
char*)(dst->imageData + i * dst->widthStep + j);
                        *dst_data = 255;
                    }
                    //i有两个值
                    i = a -(int)(sqrt(pow((double)radius,2)-
pow((double)(j-b),2))) ;
                    if ((i< dst->height) && (i >= 0))
                    {
                        dst_data = (unsigned
char*)(dst->imageData + i * dst->widthStep + j);
                        *dst_data = 255;
                    }
                }
            }
        }
    }
}
```

```

    }
}
}
```

OpenCV_top.h 代码如下：

```

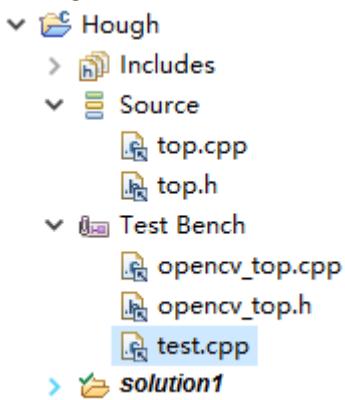
#ifndef __OPENCV_TOP_H__
#define __OPENCV_TOP_H__

#include "hls_opencv.h"

void opencv_hough(IplImage* src, IplImage* dst);

#endif
```

Step7：在 Test Bench 中添加一张名为 circle2.bmp 的测试图片，图片可以在我们提供的源程序中的 Image 文件夹中找到。完整的工程如下图所示：

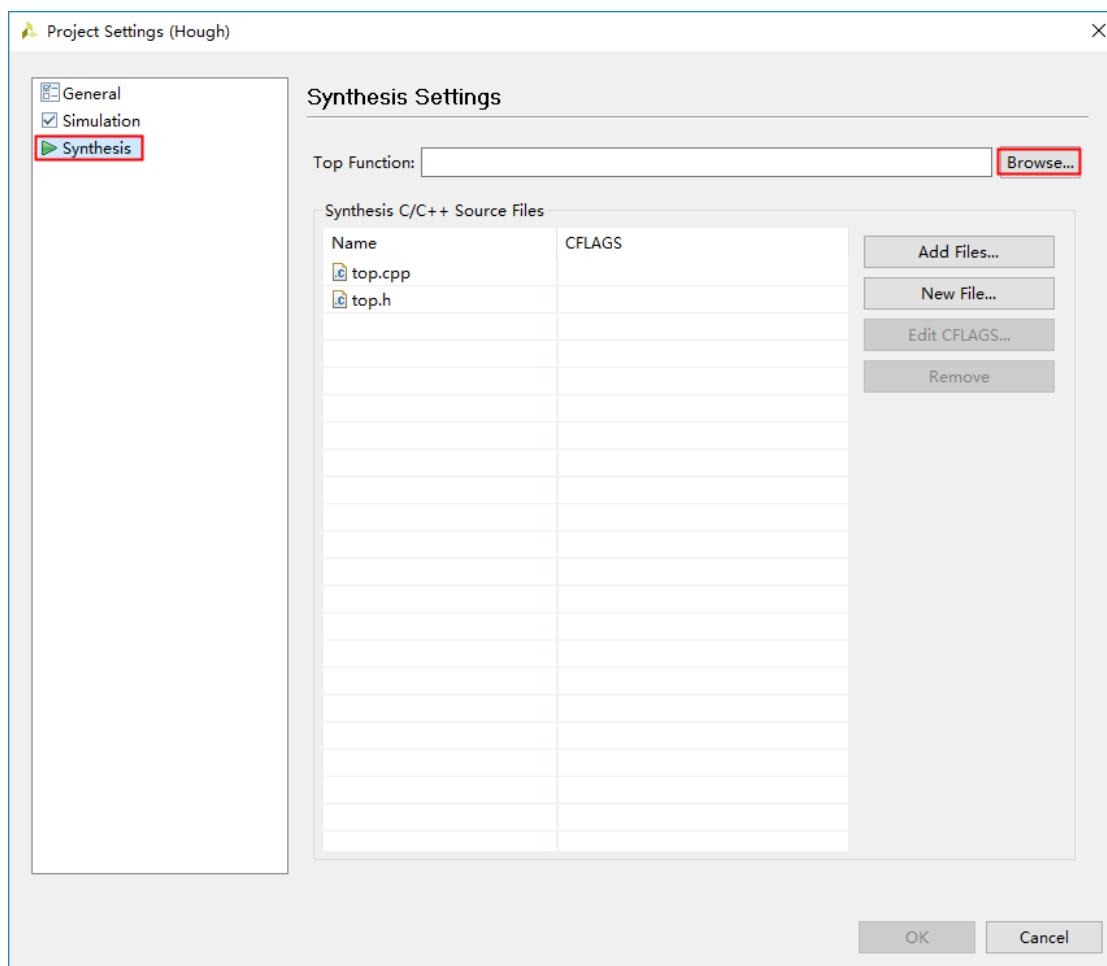


7.2.2 仿真及优化

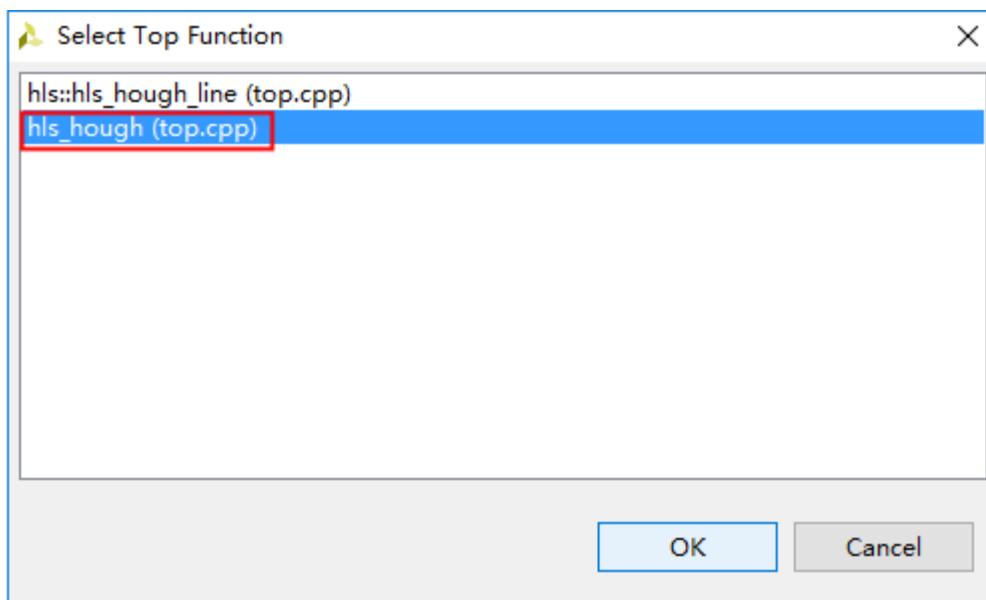
Step1：单击 Project settings 快捷键。



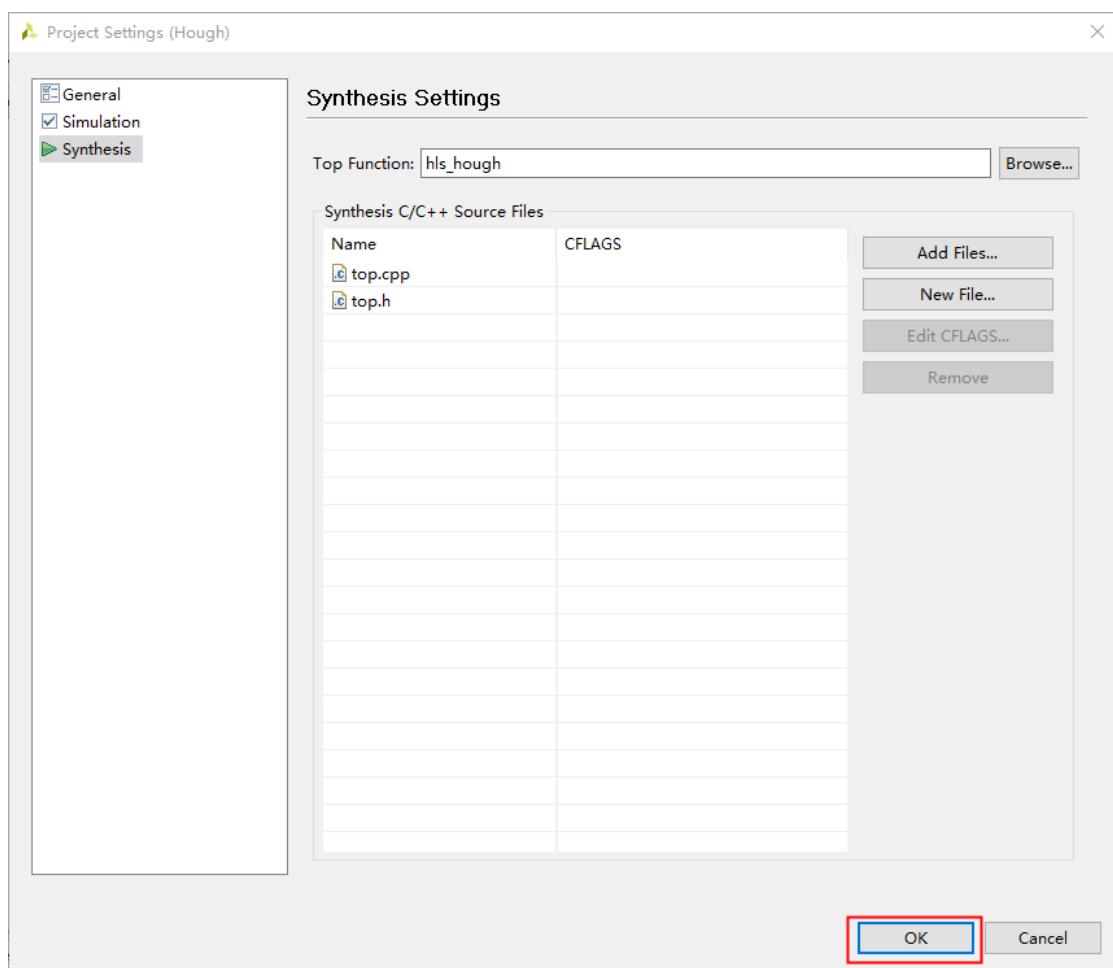
Step2：选择 Synthesis 选项，然后点击 Browse.. 指定一个顶层函数。



Step3: 选定 hls_hough 为顶层函数，然后点击 O K。



Step4: 再次单击 OK，完成工程的修改。



Step5：单击 开始综合。



综合报告如下：

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2179898	2481068676906370	2179899	2481068676906371	none

Detail

Instance

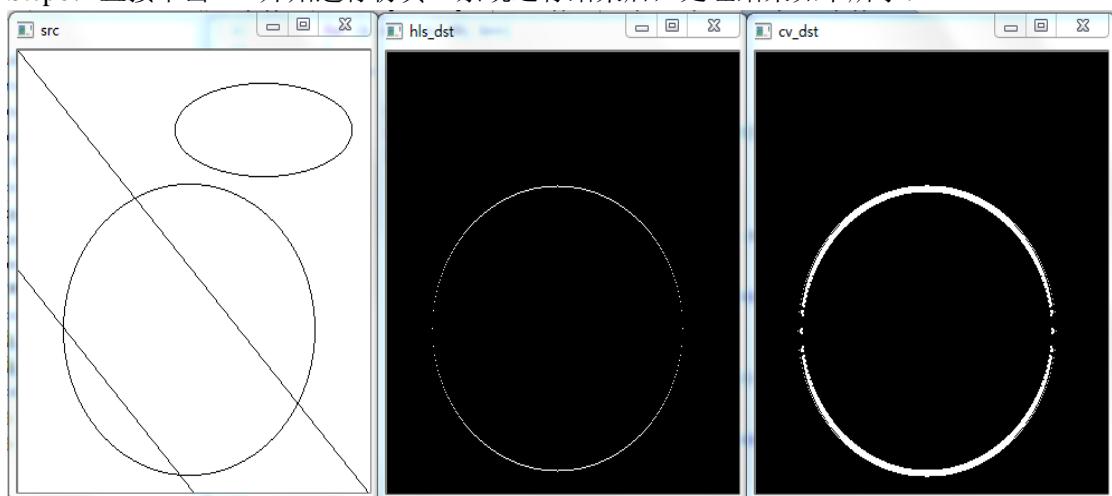
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	49
FIFO	0	-	10	40
Instance	4096	18	4301	7287
Memory	-	-	-	-
Multiplexer	-	-	-	34
Register	-	-	104	-
Total	4096	18	4415	7410
Available	120	80	35200	17600
Utilization (%)	3413	22	12	42

Step6: 直接单击  开始进行仿真。系统运行结束后，处理结果如下所示：



在这里我们给出的是 hough 变换的圆的检测，各位也可以根据 HLS 提供的视频库进行 hough 变换最基本的直线检测，这里给出函数的简单示例：

```

    hls::Scalar<3,unsigned char> color(50,50,50);
    const unsigned int linesMax = 3;
    hls::Polar_<float,float> Polar_line[linesMax];
    hls::HoughLines2<1,2>(src,Polar_line,150);
    //依次在图中绘制出每条线段
    for( int i = 0; i < linesMax; i++ )
    {
        float rho = Polar_line[i].rho, theta = Polar_line[i].angle;
        hls_Point<int> pt1, pt2;
        ...
        ...
        ...
    }
}

```

其中 hls_Point 的结构体原型为：

```

//定义点的结构体
template<typename T>
struct hls_Point
{
    T x;
    T y;
};

```

Hough 变换还可以解决许多类似的问题，如检测出椭圆，正方形，长方形，圆弧等等。这些方法大都类似，关键就是需要熟悉这些几何形状的数学性质。霍夫变换由于不受图像旋转的影响，所以很容易的可以用来进行定位。

霍夫变换有许多改进方法，一个比较重要的概念是广义霍夫变换，它是针对所有曲线的，用处也很大。就是针对直线的霍夫变换也有很多改进算法，比如前面的方法我们没有考虑图像上的这一直线上的点是否连续的问题，这些都要随着应用的不同而有优化的方法。

简单点概括就是：Hough 变换的基本原理在于利用点与线的对偶性，将原始图像空间的给定的曲线通过曲线表达形式变为参数空间的一个点。这样就把原始图像中给定曲线的检测问题转化为寻找参数空间中的峰值问题。也即把检测整体特性转化为检测局部特性。比如直线、椭圆、圆、弧线等。

算法如下：

- (1) 初始化一个变换域空间的数组，r 方向上的量化数目图像对角线方向像素，0 方向上的量化数目为 180。

- (2) 顺序搜索图像的所有黑点。对每一个黑点，在变换域的对应个点加一。

- (3) 求出变换域最大值点并记录。最大值点就是直线的所在了。

或者说：hough 变换利用图像空间和 hough 参数空间的点一线对偶性，把图像空间中的检测问题转换到参数空间。通过在参数空间里进行简单的累加统计，然后在 hough 参数空间寻找累加器峰值的方法检测直线。

7.3 程序分析

本章节的程序比较简单，主要的圆检测功能函数如下图所示：

```

28 void hls::hls_hough_line(GRAY_IMAGE &src,GRAY_IMAGE &dst,int rows,int cols)
29 {
30     GRAY_PIXEL result;
31     int row,col,k;
32     //参数空间的参数圆心O (a,b) 半径radius
33     int a = 0,b = 0,radius = 0;
34
35     //累加?
36     int A0 = rows;
37     int B0 = cols;
38
39     //注意HLS不支持变长数组，?以这里直接指定数据长?
40     const int Size = 1089900;//Size = rows*cols*(120-110);
41
42     #ifdef __SYNTHESIS__
43         int *_count[Size];
44         int *count = &_count[0];
45     #else
46         int *count =(int *) malloc(Size * sizeof(int));
47     #endif
48
49     //偏移
50     int index ;
51
52     //为累加器赋?
53     for (row = 0;row < Size;row++)
54     {
55         count[row] = 0;
56     }
57
58     GRAY_PIXEL src_data;
59     uchar temp0;
60     for (row = 0; row< rows;row++)
61     {
62         for (col = 0; col< cols;col++)
63         {
64             src >> src_data;
65             uchar temp = src_data.val[0];
66             //?测黑?
67             if (temp == 0)
68             {
69                 //遍历a ,b?:累加器赋?
70                 for (a = 0;a < A0;a++)
71                 {
72                     for (b = 0;b < B0;b++)
73                     {
74                         radius = (int)(sqrt(pow((double)(row-a),2) + pow((double)(col - b),2)));
75                         if(radius > 110 && radius < 120)
76                         {
77                             index = A0 * B0 *(radius-110) + A0*b + a;
78                             count[index]++;
79                         }
80                     }
81                 }
82             }
83         }
84     }
85
86     //遍历累加器数组，找出?有的?
87     for (a = 0 ; a < A0 ; a++)
88     {
89         for (b = 0 ; b < B0 ; b++)
90         {
91             for (radius = 110 ; radius < 120; radius++)
92             {
93                 index = A0 * B0 *(radius-110) + A0*b + a;
94                 if (count[index] > 210)
95                 {
96                     //在image2中绘制该?
97                     for(k = 0; k < rows;k++)
98                     {
99                         for (col = 0 ; col< cols;col++)
100                         {
101                             //x有两个?:根据圆公?(x-a)^2+(y-b)^2=r^2得到
102                             int temp = (int)(sqrt(pow((double)radius,2) - pow((double)(col-b),2)));
103                             int x1 = a + temp;
104                             int x2 = a - temp;
105                             if ( (k == x1)|| (k == x2) )
106                             {
107                                 result.val[0] = (uchar)255;
108                             }
109                             else{
110                                 result.val[0] = (uchar)0;
111                             }
112                         }
113                     }
114                 }
115             }
116         }
117     }
118 }

```

在程序中有一些必要的注释，大家可以看一看。之前的一些变量定义在此都给出了释义，在此

就直接跳过，先看到此函数中的这一部分，如下图：

```

52 //为累加器赋值
53 for (row = 0; row < Size; row++)
54 {
55     count[row] = 0;
56 }
57
58 GRAY_PIXEL src_data;
59 uchar temp0;
60 for (row = 0; row < rows; row++)
61 {
62     for (col = 0; col < cols; col++)
63     {
64         src >> src_data;
65         uchar temp = src_data.val[0];
66         //测黑点
67         if (temp == 0)
68         {
69             //遍历a,b;累加器赋值
70             for (a = 0; a < A0; a++)
71             {
72                 for (b = 0; b < B0; b++)
73                 {
74                     radius = (int)(sqrt(pow((double)(row-a), 2) + pow((double)(col - b), 2)));
75                     if (radius > 110 && radius < 120)
76                     {
77                         index = A0 * B0 * (radius-110) + A0*b + a;
78                         count[index]++;
79                     }
80                 }
81             }
82         }
83     }
84 }

```

一开始，程序中用了一个 for 循环，把 count 数组中的内容清零清理，这里需要提出的是 Size 的容量问题，在其定义时也给出了相关的注释，如下图所示：

```
//注意HLS不支持变长数组，以这里直接指定数据长
const int Size = 1089900; //Size = rows*cols*(120-110);
```

这里方便仿真，我们直接限定了其范围，将数组的范围定义成了一个固定的数值，但是需要注意的是如果在实际的项目当中，就应该对内存进行操作。在 size 公式中的 (120-110) 是表示圆的半径在 110-120 个像素点之间。回到函数的分析当中，接下来，函数定义了两个变量，我们注意到第一个变量的类型为一个名为 GRAY_PIXEL 的关键字，我们将鼠标停放在这个关键字上查看其定义，如下图所示：

```

39 typedef hls::Scalar<3, unsigned char> RGB_PIXEL;
40 typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> RGB_IMAGE;
41 typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1> GRAY_IMAGE;
42 typedef hls::Scalar<1, unsigned char> GRAY_PIXEL;
43 typedef unsigned char uchar;
44

```

从上图中我们注意到有一个与其非常相似的定义，通过二者的比较，我们可以猜测这必定是定义了一种色彩空间，其类型为无符号整型数据，因此这里的 src_data 应该就是一个灰度值数据。接下来看到这一句：

```

for (row = 0; row < rows; row++)
{
    for (col = 0; col < cols; col++)
    {
        src >> src_data;
        uchar temp = src_data.val[0];
        //♦?测黑♦?
        if (temp == 0)
    }
}

```

一开始，是个双重 for 语句，将行列的数据逐个的赋值给了 src_data,然后将 temp 指向了 src_data 的开始位，也就是说 temp 指向的是图像的起始处。然后是个 if 判断，这个 if 判断在这就是起了个检测图像的黑线的作用。然后看到接下来的这一部分：

```

//遍历a ,b♦?;累加器赋♦?
for (a = 0; a < A0; a++)
{
    for (b = 0; b < B0; b++)
    {
        radius = (int)(sqrt(pow((double)(row-a), 2) + pow((double)(col - b), 2)));
        if(radius > 110 && radius < 120)
        {
            index = A0 * B0 * (radius-110) + A0*b + a;
            count[index]++;
        }
    }
}

```

这部分就是遍历 a,b 为累加器赋值！这里的 radius 就是求出圆的半径，也就是我们第一节中讲到的

$$(x_1 - a_1)^2 + (y_1 - b_1)^2 = r_1^2$$
 检测过圆的点的公式：之后求出的

半径在于实际的圆的半径对比，如果符合条件，就记下这点的偏移量，然后将这个偏移量存放在 count 数组对应的偏移地址中。经过这一个过程，所有过圆的点就将全部都会记录在 count 数组当中。最后分析一下这段代码：

```
//遍历累加器数组，找出是否有圆点
for (a = 0 ; a < A0 ; a++)
{
    for (b = 0 ; b < B0; b++)
    {
        for (radius = 110 ; radius < 120; radius++)
        {
            index = A0 * B0 *(radius-110) + A0*b + a;
            if (count[index] > 210)
            {
                //在image2中绘制该点
                for(k = 0; k < rows;k++)
                {
                    for (col = 0 ; col< cols;col++)
                    {
                        //x有两个点，根据圆公式(x-a)^2+(y-b)^2=r^2得到
                        int temp = (int)(sqrt(pow((double)radius,2) - pow((double)(col-b),2)));
                        int x1 = a + temp;
                        int x2 = a - temp;
                        if ( (k == x1) || (k == x2) )
                        {
                            result.val[0] = (uchar)255;
                        }
                        else{
                            result.val[0] = (uchar)0;
                        }
                        dst << result;
                    }
                }
            }
        }
    }
}
```

在这部分程序当中也是先遍历 a, b, 然后在指定的半径中, 先求出了一个偏移量 index, 然后判断在 count 数组的这个偏移量位置中的数据是否大于 210, 这里的这个 210 是我们设置的一个阀值, 实际上也可以取其他的数值, 之后在指定范围 (rows,cols) 内, 通过特定的公式判断这点是否是圆上的一点, 若是则用 255 代替原来的数值, 否则用 0 代替原来的数值, 最后将结果赋值给了目标图像。

7.4 本章小结

本章介绍了如何使用 HLS 设计一个基于 Hough 变换的圆检测算法, 通过这种方法, 还可以对其进行拓展, 设计基于 Hough 变换的直线检测算法, 感兴趣的可以尝试一下, 在一些特征物体具有特定的形状时, 此算法可方便对目标进行识别, 方便定位, 具有一定得实用价值。

S05_CH08_傅里叶变换的 HLS 实现

8.1 FFT 原理介绍

FFT 是离散傅立叶变换的快速算法，可以将一个信号变换到频域。有些信号在时域上是很难看出什么特征的，但是如果变换到频域之后，就很容易看出特征了。这就是很多信号分析采用 FFT 变换的原因。另外，FFT 可以将一个信号的频谱提取出来，这在频谱分析方面也是经常用的。在笔者参与的几个项目中就有几个使用到了 FFT，因此在这里准备在 HLS 上实现这一算法。另外在后面的几个实验中我们都用到了这一算法，因此前面先对它进行讲解，方便大家在接下来的实验中进行使用。

FFT 结果的物理意义网上有一大神圈圈对此做了详细的描述，我们在这里摘录如下方便大家理解 FFT。

一个模拟信号，经过 ADC 采样之后，就变成了数字信号。根据采样定理，采样频率要大于信号频率的两倍。采样得到的数字信号，就可以做 FFT 变换了。N 个采样点，经过 FFT 之后，就可以得到 N 个点的 FFT 结果。为了方便进行 FFT 运算，通常 N 取 2 的整数次方。

假设采样频率为 F_s ，信号频率 F ，采样点数为 N 。那么 FFT 之后结果就是一个为 N 点的复数。每一个点就对应着一个频率点。这个点的模值，就是该频率值下的幅度特性。具体跟原始信号的幅度有什么关系呢？假设原始信号的峰值为 A ，那么 FFT 的结果的每个点（除了第一个点直流分量之外）的模值就是 A 的 $N/2$ 倍。而第一个点就是直流分量，它的模值就是直流分量的 N 倍。而每个点的相位就是在该频率下的信号的相位。第一个点表示直流分量（即 0Hz），而最后一个点 N 的再下一个点（实际上这个点是不存在的，这里是假设的第 $N+1$ 个点，也可以看做是将第一个点分做两半分，另一半移到最后）则表示采样频率 F_s ，中间被 $N-1$ 个点平均分成 N 等份，每个点的频率依次增加。例如某点 n 所表示的频率为： $F_n = (n-1) * F_s / N$ 。由上面的公式可以看出， F_n 所能分辨到频率为 F_s/N ，如果采样频率 F_s 为 1024Hz，采样点数为 1024 点，则可以分辨到 1Hz。1024Hz 的采样率采样 1024 点，刚好是 1 秒，也就是说，采样 1 秒时间的信号并做 FFT，则结果可以分析到 1Hz，如果采样 2 秒时间的信号并做 FFT，则结果可以分析到 0.5Hz。如果要提高频率分辨率，则必须增加采样点数，也即采样时间。频率分辨率和采样时间是倒数关系。

假设 FFT 之后某点 n 用复数 $a+bi$ 表示，那么这个复数的模就是 $A_n = \sqrt{a^2 + b^2}$ ，相位就是 $P_n = \text{atan2}(b, a)$ 。根据以上的结果，就可以计算出 n 点 ($n \neq 1$, 且 $n < N/2$) 对应的信号的表达式为：

$A_n / (N/2) * \cos(2\pi F_n t + P_n)$, 即 $2A_n/N * \cos(2\pi F_n t + P_n)$ 。

对于 $n=1$ 点的信号, 是直流分量, 幅度即为 A_1/N 。

由于 FFT 结果的对称性, 通常我们只使用前半部分的结果, 即小于采样频率一半的结果。

假设我们有一个信号, 它含有 2V 的直流分量, 频率为 50Hz、相位为 -30 度、幅度为 3V 的交流信号, 以及一个频率为 75Hz、相位为 90 度、幅度为 1.5V 的交流信号。用数学表达式就是如下:

$$S = 2 + 3 * \cos(2\pi * 50 * t - \pi * 30 / 180) + 1.5 * \cos(2\pi * 75 * t + \pi * 90 / 180)$$

式中 cos 参数为弧度, 所以 -30 度和 90 度要分别换算成弧度。我们以 256Hz 的采样率对这个信号进行采样, 总共采样 256 点。按照我们上面的分析, $F_n = (n-1) * F_s / N$, 我们可以知道, 每两个点之间的间距就是 1Hz, 第 n 个点的频率就是 $n-1$ 。我们的信号有 3 个频率: 0Hz、50Hz、75Hz, 应该分别在第 1 个点、第 51 个点、第 76 个点上出现峰值, 其它各点应该接近 0。实际情况如何呢?

我们来看看 FFT 的结果的模值如图所示。

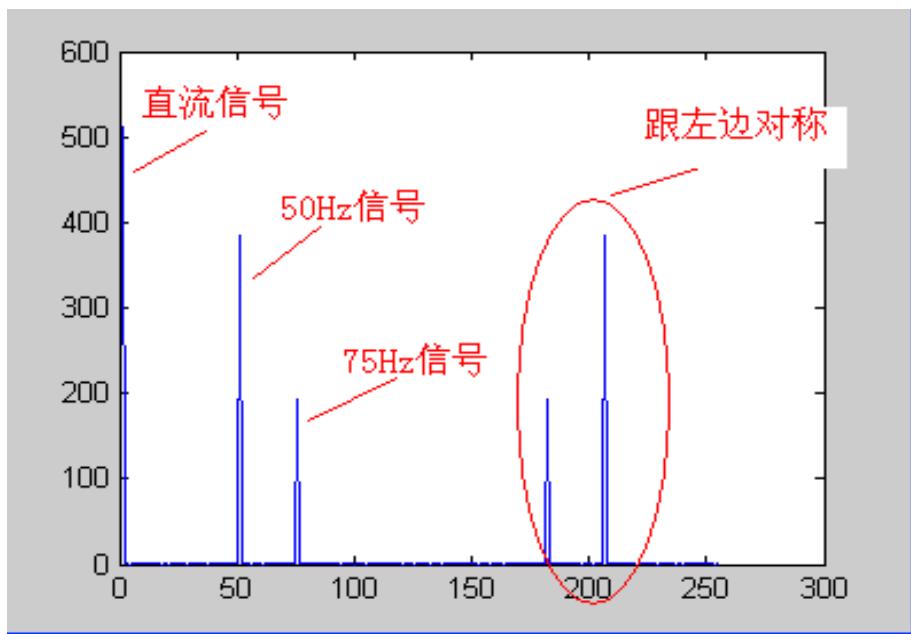


图 1 FFT 结果

从图中我们可以看到, 在第 1 点、第 51 点、和第 76 点附近有比较大的值。我们分别将这三个点附近的数据拿上来细看:

1 点: $512+0i$

2 点: $-2.6195E-14 - 1.4162E-13i$

3 点: $-2.8586E-14 - 1.1898E-13i$

50 点: $-6.2076E-13 - 2.1713E-12i$

51 点: 332.55 - 192i

52 点: -1.6707E-12 - 1.5241E-12i

75 点: -2.2199E-13 -1.0076E-12i

76 点: 3.4315E-12 + 192i

77 点: -3.0263E-14 +7.5609E-13i

很明显, 1 点、51 点、76 点的值都比较大, 它附近的点值都很小, 可以认为是 0, 即在那些频率点上的信号幅度为 0。接着, 我们来计算各点的幅度值。分别计算这三个点的模值, 结果如下:

1 点: 512

51 点: 384

76 点: 192

按照公式, 可以计算出直流分量为: $512/N=512/256=2$; 50Hz 信号的幅度为: $384/(N/2)=384/(256/2)=3$; 75Hz 信号的幅度为 $192/(N/2)=192/(256/2)=1.5$ 。可见, 从频谱分析出来的幅度是正确的。

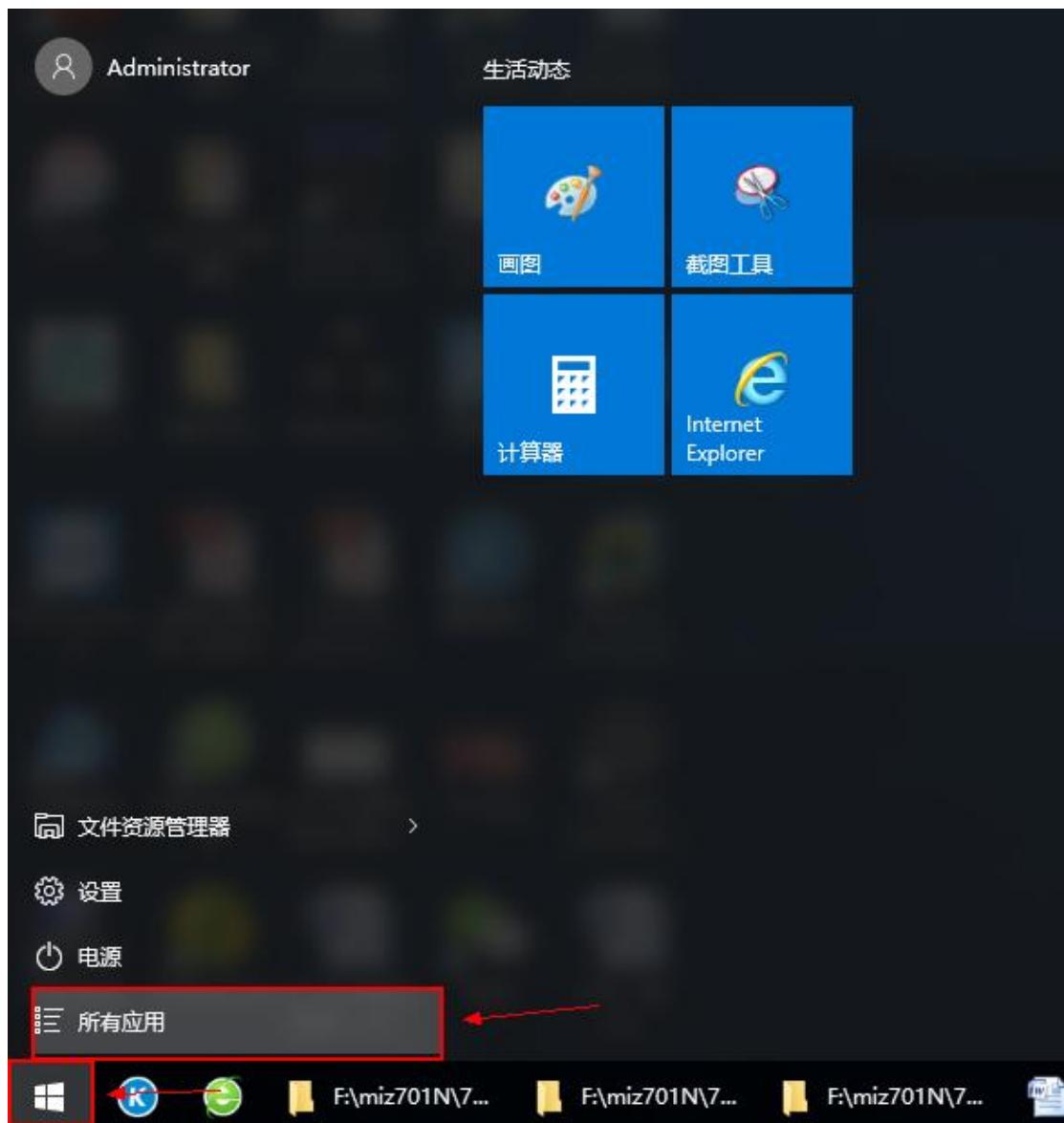
然后再来计算相位信息。直流信号没有相位可言, 不用管它。先计算 50Hz 信号的相位, $\text{atan2}(-192, 332.55)=-0.5236$, 结果是弧度, 换算为角度就是 $180*(-0.5236)/\pi=-30.0001$ 。再计算 75Hz 信号的相位, $\text{atan2}(192, 3.4315E-12)=1.5708$ 弧度, 换算成角度就是 $180*1.5708/\pi=90.0002$ 。可见, 相位也是对的。根据 FFT 结果以及上面的分析计算, 我们就可以写出信号的表达式了, 它就是我们开始提供的信号。

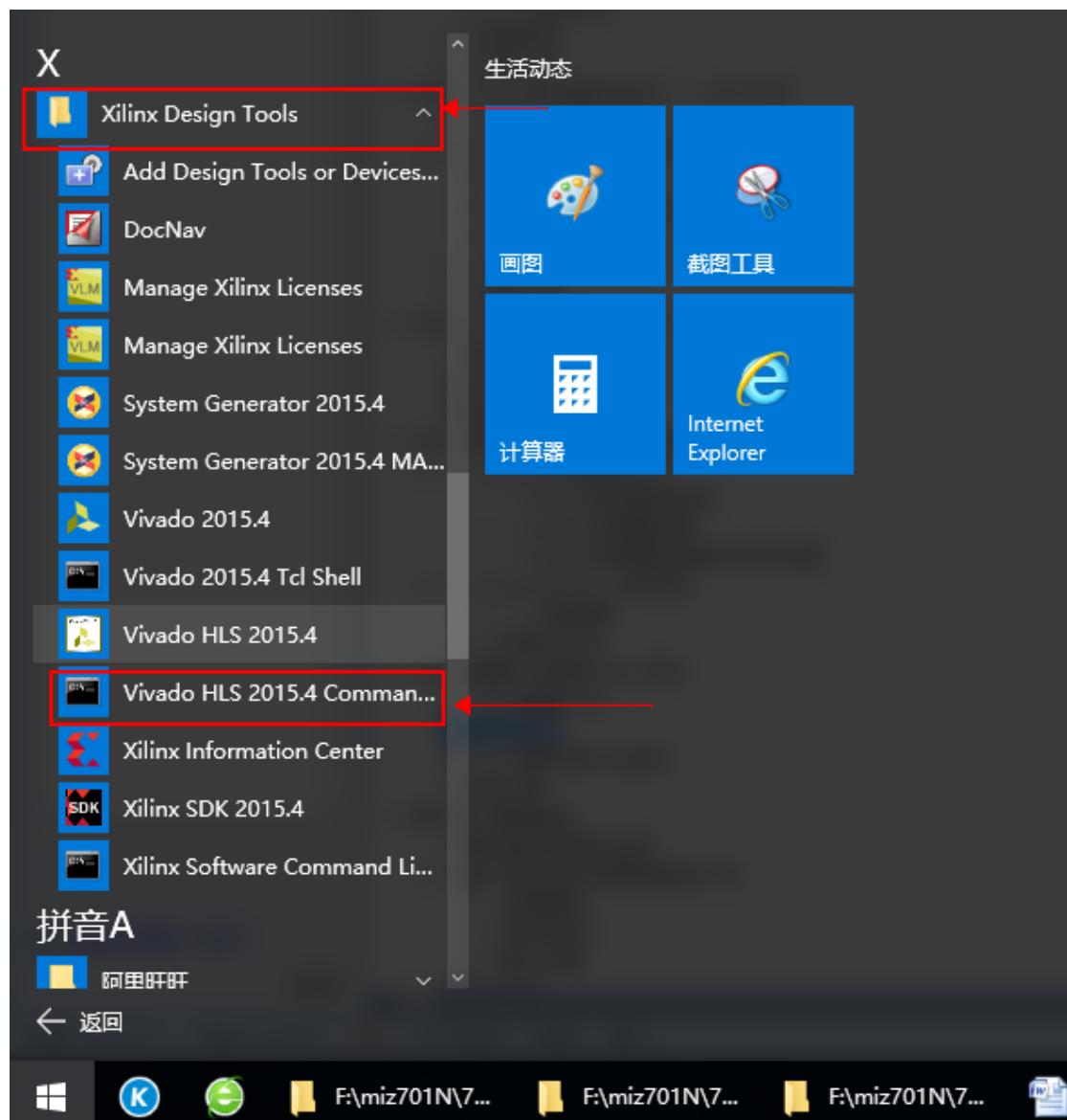
总结: 假设采样频率为 F_s , 采样点数为 N , 做 FFT 之后, 某一点 n (n 从 1 开始) 表示的频率为: $F_n=(n-1)*F_s/N$; 该点的模值除以 $N/2$ 就是对应该频率下的信号的幅度(对于直流信号是除以 N); 该点的相位即是对应该频率下的信号的相位。相位的计算可用函数 $\text{atan2}(b, a)$ 计算。 $\text{atan2}(b, a)$ 是求坐标为 (a, b) 点的角度值, 范围从 $-\pi$ 到 π 。要精确到 x Hz, 则需要采样长度为 $1/x$ 秒的信号, 并做 FFT。要提高频率分辨率, 就需要增加采样点数, 这在一些实际的应用中是不现实的, 需要在较短的时间内完成分析。解决这个问题的方法有频率细分法, 比较简单的方法是采样比较短时间的信号, 然后在后面补充一定数量的 0, 使其长度达到需要的点数, 再做 FFT, 这在一定程度上能够提高频率分辨率。具体的频率细分法可参考相关文献。

8.2 HLS 实现

在官方的参考手册 ug871-vivado-high-level-synthesis-tutorial.pdf 中对 FFT 有比较详细的介绍，并且官方提供的 Example 中也有对应的工程方便大家学习，这一节我们主要说一下如何使用 TCL 语言创建工程，并且进行对应的操作，源码直接使用官方提供的。

Step1: 在开始菜单中找到并打开 vivado hls 2015.4 command (此处以 win10 操作系统为例, win7 操作系统区别不是很大)。





Step2:一般通过 cd 路径即可进入你想进入的文件路径，比如我们想进入我们的工作路径，输入 cd F:\MZ7X\S05\S05_CH08_fft\tcl，我们看红色标注的地方会发现路径没有切换，那么这个问题我们应该怎么解决呢？输入 cd/?查看系统给出的解决方案，那么进入盘符我们通过使用 cd /d 工作文件夹路径的命令进行路径切换，

```

c:\ Vivado HLS 2016.4 Command Prompt
=====
== Vivado HLS Command Prompt
== Available commands:
== vivado_hls, apcc, gcc, g++, make
=====
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

d:\Xilinx\Vivado_HLS\2016.4>cd f:/MZ7X/S05/S05_CH08_fft/tcl
d:\Xilinx\Vivado_HLS\2016.4>

```

搜狗拼音输入法 全 :

```

c:\ Vivado HLS 2016.4 Command Prompt
d:\Xilinx\Vivado_HLS\2016.4>cd /?
显示当前目录名或改变当前目录。
CHDIR [/D] [drive:][path]
CHDIR [..]
CD [/D] [drive:][path]
CD [..]

... 指定要改成父目录。
键入 CD drive: 显示指定驱动器中的当前目录。
不带参数只键入 CD, 则显示当前驱动器和目录。
使用 /D 开关, 除了改变驱动器的当前目录之外,
还可改变当前驱动器。
如果命令扩展被启用, CHDIR 会如下改变:
当前的目录字符串会被转换成使用磁盘名上的大小写。所以,
如果磁盘上的大小写如此, CD C:\TEMP 会将当前目录设为
C:\Temp。
CHDIR 命令不把空格当作分隔符, 因此有可能将目录名改为一个
带有空格但不带有引号的子目录名。例如:
cd \winnt\profiles\username\programs\start menu
与下列相同:
请按任意键继续. . .

```

```

d:\Xilinx\Vivado_HLS\2016.4>cd /d f:/MZ7X/S05/S05_CH08_fft/tcl
f:\MZ7X\S05\S05_CH08_fft\tcl>

```

Step3:路径切换以后我们通过 vivado_hls -f run_hls.tcl 命令对 tcl 文件进行编译，注意路径及命令格式(所有的文件在我们提供的源程序包中的 src 文件夹中可以找到)：

```

d:\Xilinx\Vivado_HLS\2016.4>cd /d f:/MZ7X/S05/S05_CH08_fft/tcl
f:\MZ7X\S05\S05_CH08_fft\tcl>vivado_hls -f run_hls.tcl

```

Step4:运行命令后，系统就将自动的完成 tcl 脚本中的设定，运行信息和结果如下所示：

```
## run all
$finish called at time : 17574 ns : File "F:/miz701N/7010/hls/hls_fft/tcl/fe_vhls_prj/IPXACTExport/sim/verilog/hls_real2
xfft.autotb.v" Line 293
## quit
INFO: [Common 17-206] Exiting xsim at Mon Mar 13 16:40:38 2017...
WI [SIM-316] Starting C post checking ...


0: { -0.000061, 0.000000 }: mag = 0.000061
1: { 0.000000, 0.000000 }: mag = 0.000000
2: { 0.000000, 0.000000 }: mag = 0.000000
3: { 0.000000, 0.000000 }: mag = 0.000000
4: { -0.050415, 0.000000 }: mag = 0.050415
5: { 0.118408, 0.000000 }: mag = 0.118408
6: { -0.050415, 0.000000 }: mag = 0.050415
7: { 0.000000, 0.000000 }: mag = 0.000000
8: { 0.000000, 0.000000 }: mag = 0.000000
9: { 0.000000, 0.000000 }: mag = 0.000000
10: { 0.000000, 0.000000 }: mag = 0.000000
11: { 0.000000, 0.000000 }: mag = 0.000000
12: { 0.000000, 0.000000 }: mag = 0.000000
13: { 0.000000, 0.000000 }: mag = 0.000000
14: { -0.000031, -0.000031 }: mag = 0.000043
15: { -0.000031, -0.000031 }: mag = 0.000043
16: { 0.000000, -0.000031 }: mag = 0.000031
17: { 0.000000, 0.000000 }: mag = 0.000000
18: { 0.000000, 0.000000 }: mag = 0.000000
19: { 0.000000, 0.000000 }: mag = 0.000000
20: { 0.000000, -0.000031 }: mag = 0.000031
21: { 0.000000, -0.000031 }: mag = 0.000031
22: { 0.000000, 0.000000 }: mag = 0.000000
23: { 0.000000, 0.000000 }: mag = 0.000000
24: { 0.000000, 0.000000 }: mag = 0.000000
25: { 0.000000, 0.000000 }: mag = 0.000000
26: { 0.000000, 0.000000 }: mag = 0.000000
27: { 0.000000, 0.000000 }: mag = 0.000000
28: { 0.000000, 0.000000 }: mag = 0.000000
```

```

I [RTGEN-100] Finished creating RTL model for 'hls_xfft2real_Loop_realfft_be_buffer_proc'.
I [HLS-111] Elapsed time: 0.164 seconds; current memory usage: 796 MB.
I [HLS-10]
I [HLS-10] -- Generating RTL for module 'hls_xfft2real_Loop_realfft_be_descramble_pro'
I [HLS-10]
I [RTGEN-100] Generating core module 'hls_xfft2real_mac_muladd_16s_16s_31s_31' : 1 instance(s).
I [RTGEN-100] Generating core module 'hls_xfft2real_mac_mulsub_16s_16s_31s_31' : 1 instance(s).
I [RTGEN-100] Generating core module 'hls_xfft2real_mul_mul_16s_16s_31_3' : 2 instance(s).
I [RTGEN-100] Finished creating RTL model for 'hls_xfft2real_Loop_realfft_be_descramble_pro'.
I [HLS-111] Elapsed time: 0.227 seconds; current memory usage: 796 MB.
I [HLS-10]
I [HLS-10] -- Generating RTL for module 'hls_xfft2real'
I [HLS-10]
I [RTGEN-500] Setting interface mode on port 'hls_xfft2real/din_V_data' to 'axis'.
I [RTGEN-500] Setting interface mode on port 'hls_xfft2real/din_V_last_V' to 'axis'.
I [RTGEN-500] Setting interface mode on port 'hls_xfft2real/dout_V' to 'axis'.
I [RTGEN-500] Setting interface mode on function 'hls_xfft2real' to 'ap_ctrl_hs'.
I [RTGEN-100] Finished creating RTL model for 'hls_xfft2real'.
I [HLS-111] Elapsed time: 0.274 seconds; current memory usage: 796 MB.
I [RTMG-279] Implementing memory 'hls_xfft2real_Loop_realfft_be_descramble_pro_twid_rom_0_rom' using auto ROMs.
I [RTMG-279] Implementing memory 'hls_xfft2real_Loop_realfft_be_descramble_pro_twid_rom_1_rom' using auto ROMs.
I [RTMG-278] Implementing memory 'hls_xfft2real_descramble_buf_0_M_real_V_memcore_ram' using block RAMs.
I [HLS-10] Finished generating all RTL models.
I [WSYSC-301] Generating RTL SystemC for 'hls_xfft2real'.
I [WVHDL-304] Generating RTL VHDL for 'hls_xfft2real'.
I [WVLOG-307] Generating RTL Verilog for 'hls_xfft2real'.
I [IMPL-8] Exporting RTL as an IP in IP-XACT.

***** Vivado v2015.4 (64-bit)
**** SW Build 1412921 on Wed Nov 18 09:43:45 MST 2015
**** IP Build 1412160 on Tue Nov 17 13:47:24 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

source run_ipack.tcl -notrace
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository 'D:/Xilinx/Vivado/2015.4/data/ip'.
INFO: [Common 17-206] Exiting Vivado at Mon Mar 13 16:42:18 2017...
I [HLS-112] Total elapsed time: 233.481 seconds; peak memory usage: 796 MB.
F:\miz701N\7010\hls\hls_fft\tcl>

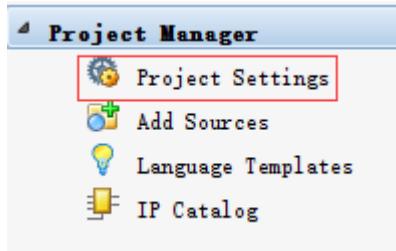
```

8.3 Vivado 模块例化及 IP 封包

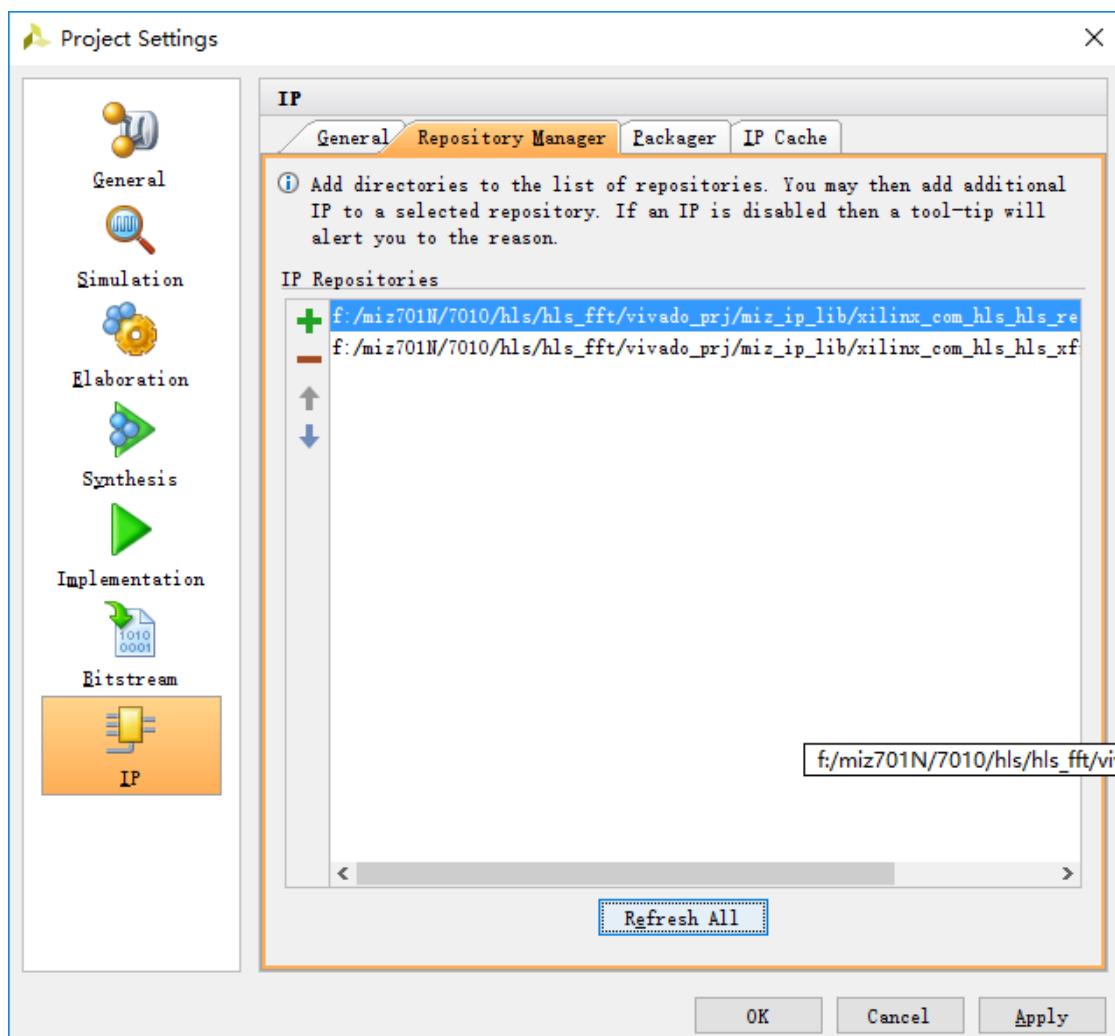
Step1:HLS 生成了 IP 之后，此时还不能直接拿来使用，还需要对其进行一些必要的配置，然后将上一节生成的两个 IP 封装成一个 IP 使用。首先我们创建一个名为 fft_ifft 的工程。

Step2：一直 next 按照默认设置配置下去，在芯片设置步骤时根据自身芯片类型选择，最后点击 Finish 完成工程的创建。

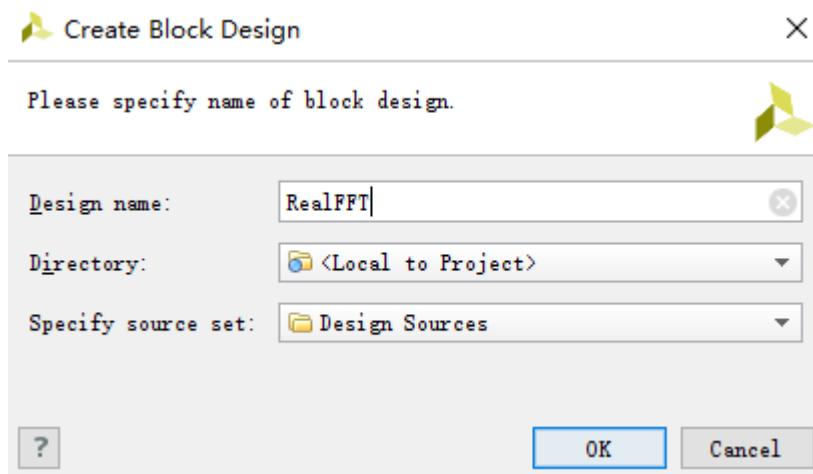
Step3: 工程创建好以后，我们需要把生成的 HLS IP 添加进当前的工程项目当中，单击 project manager 中的 project settings。



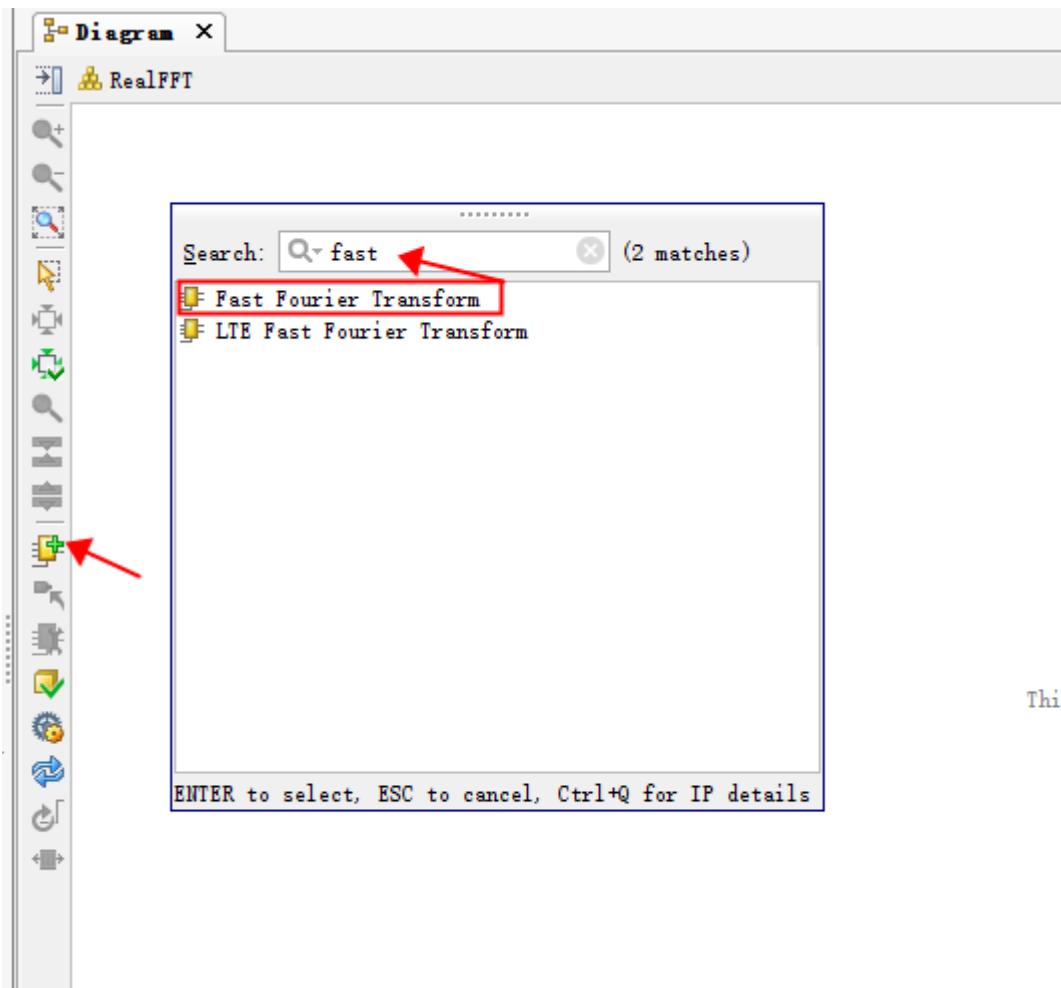
Step4：选择 IP 选项，再选择 Repository Manager 选项，单击加号图标将上一节生成的 IP 添加到工程当中，最后单击 OK。



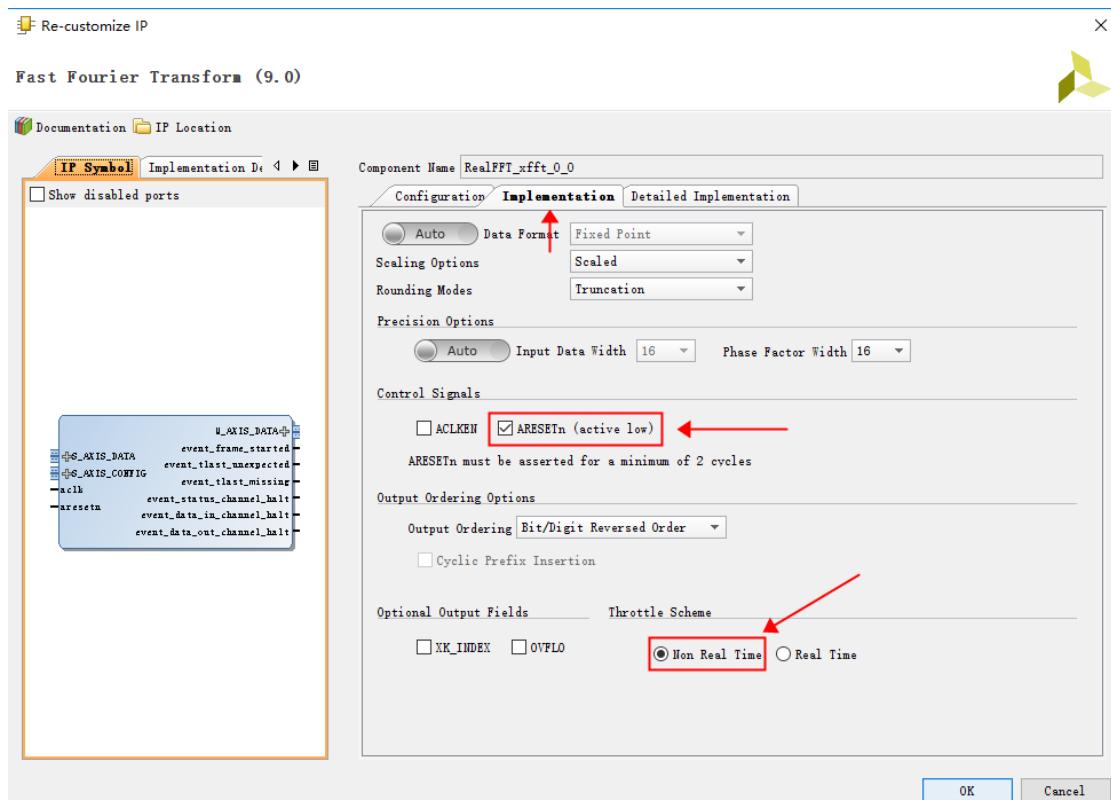
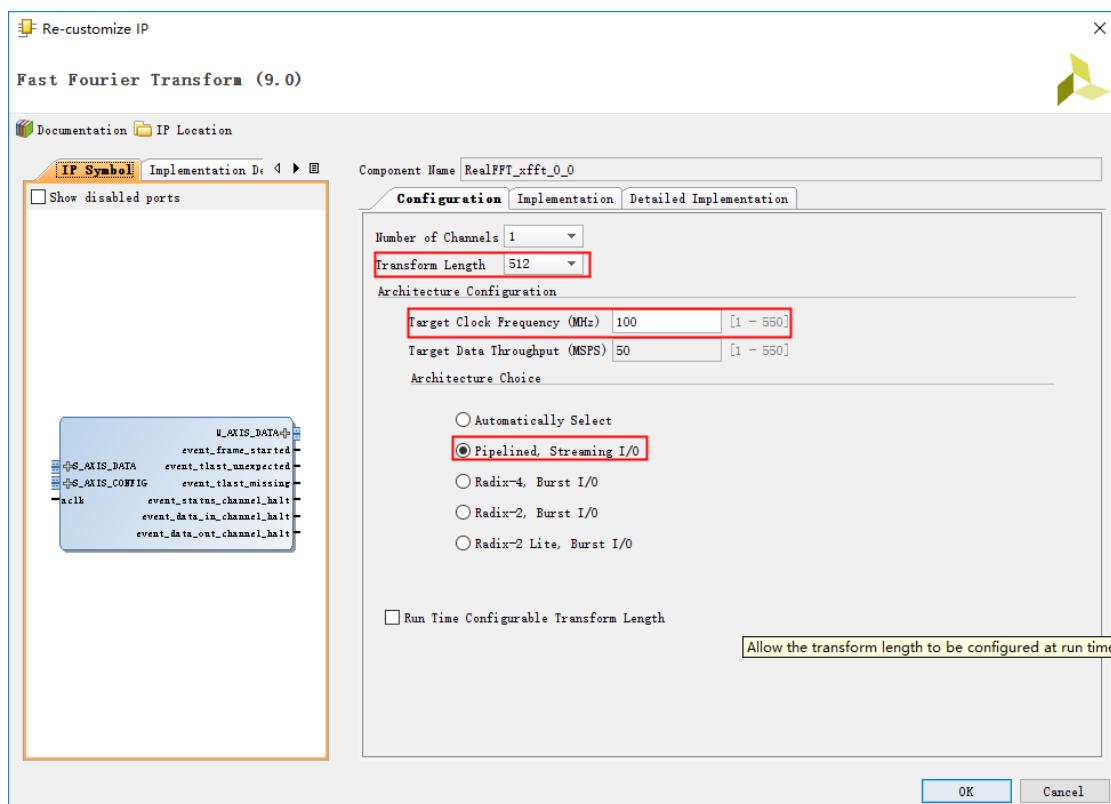
Step5: 单击 Create Block design, 创建一个名为 RealFFT 的 BD 文件。



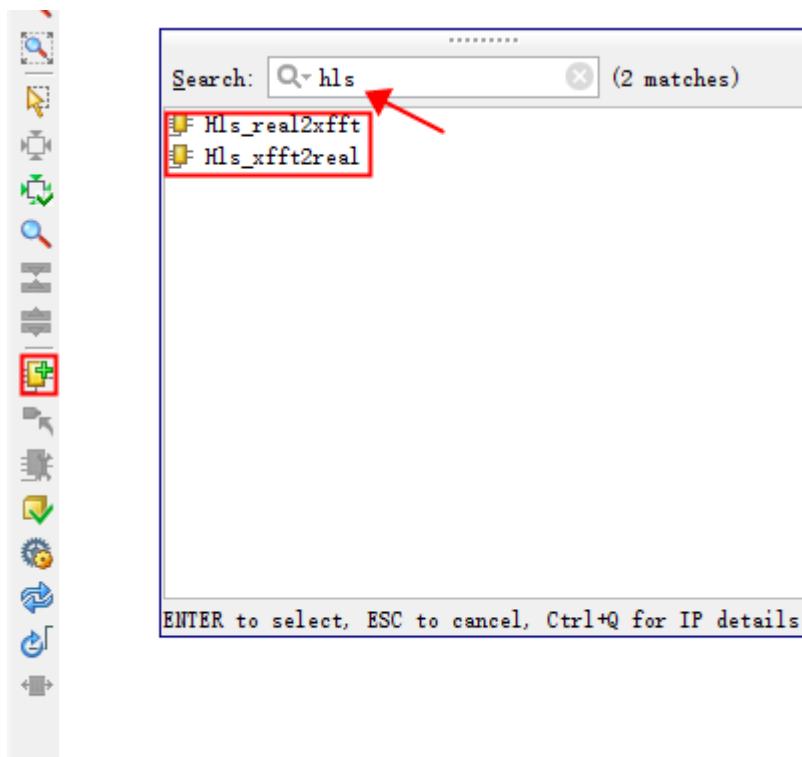
Step6: 单击 添加一个名为 Fast Fourier Transfer 的 IP。



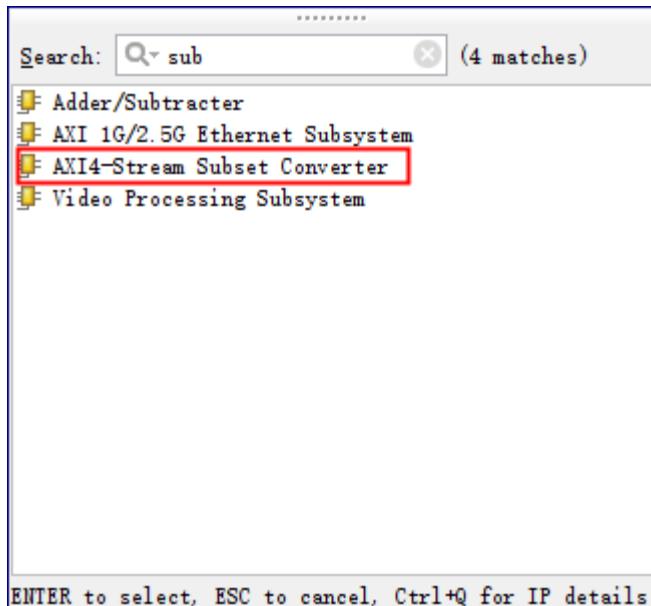
Step7: 双击这个IP，如下图所示配置此IP，然后单击OK。



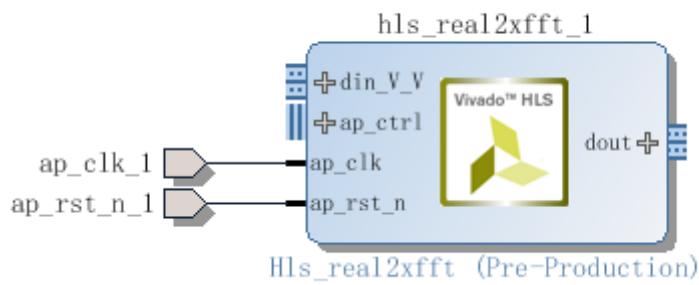
Step8: 单击 将上一节生成的两个 IP 添加到 BD 文件中来。



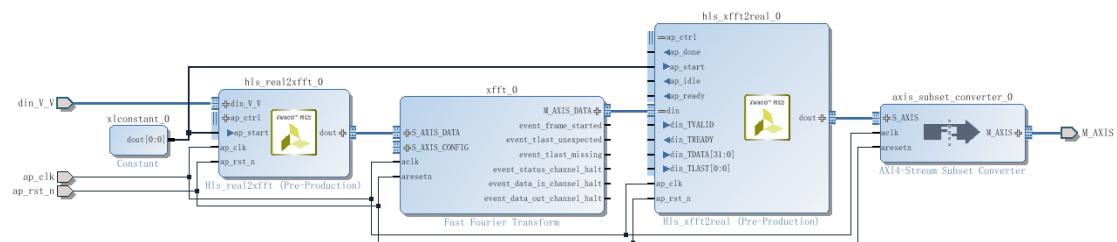
Step9: 添加一个 AXI4-Stream Subset Converter。



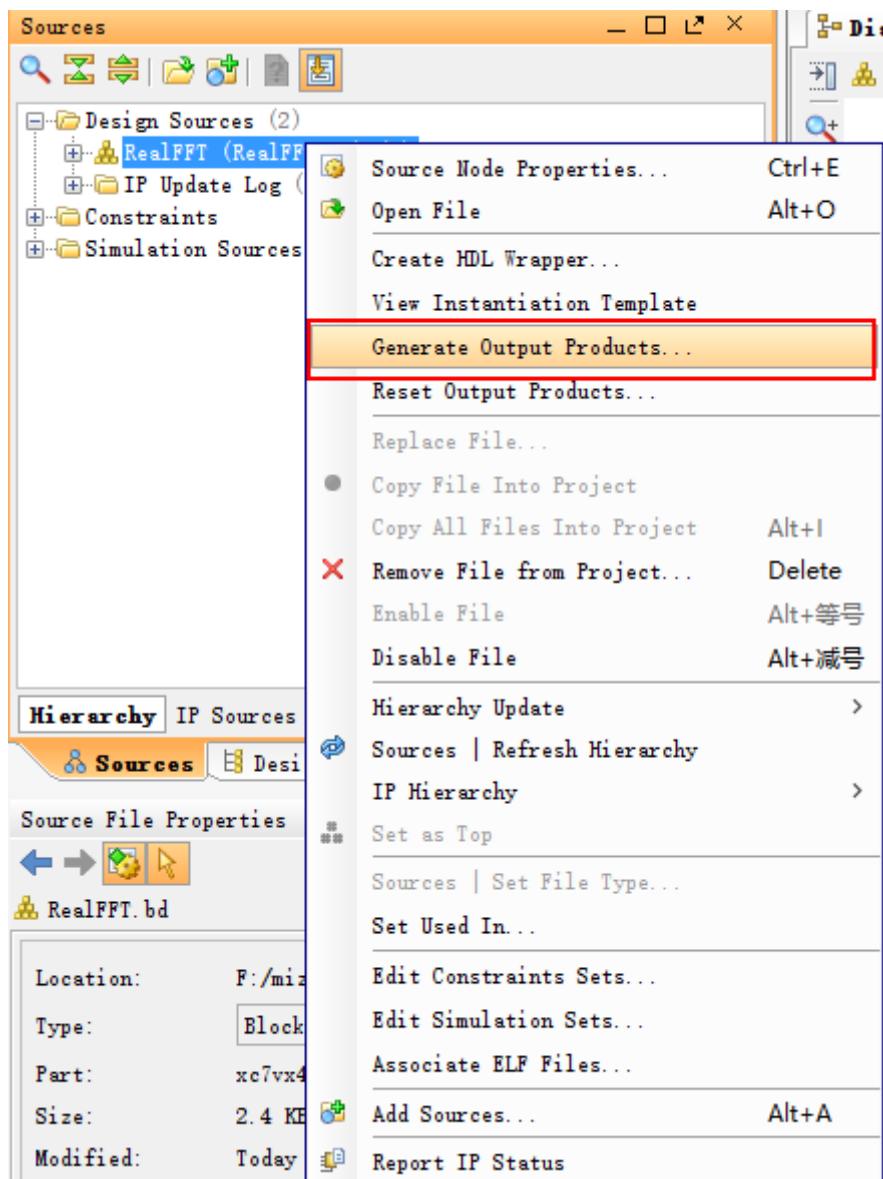
Step10: 为 ap_clk, ap_rst_n 引出一个端口, 如下图所示。



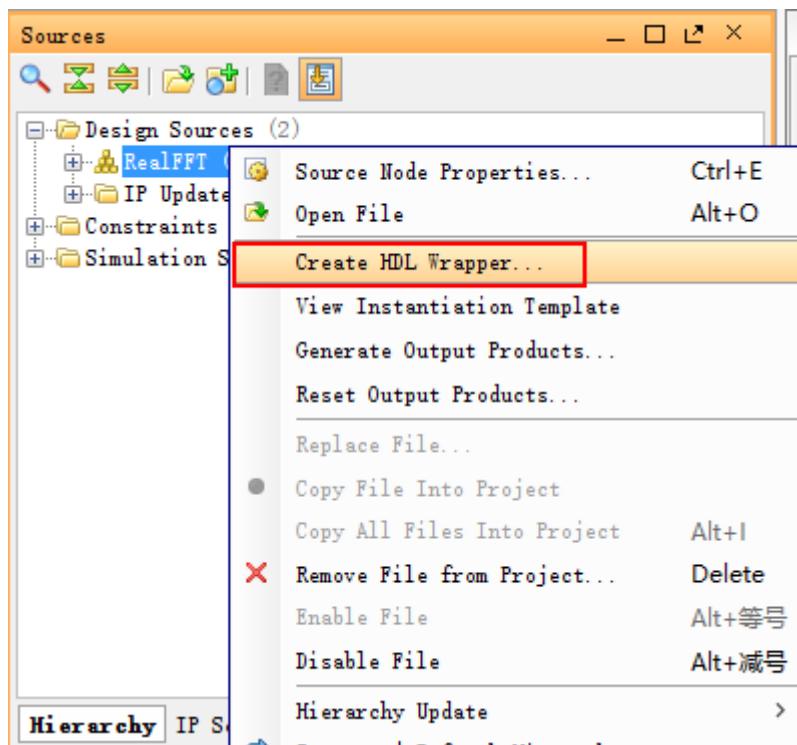
Step11: 单击 再添加一个 constant IP, 然后按下图完善硬件电路。



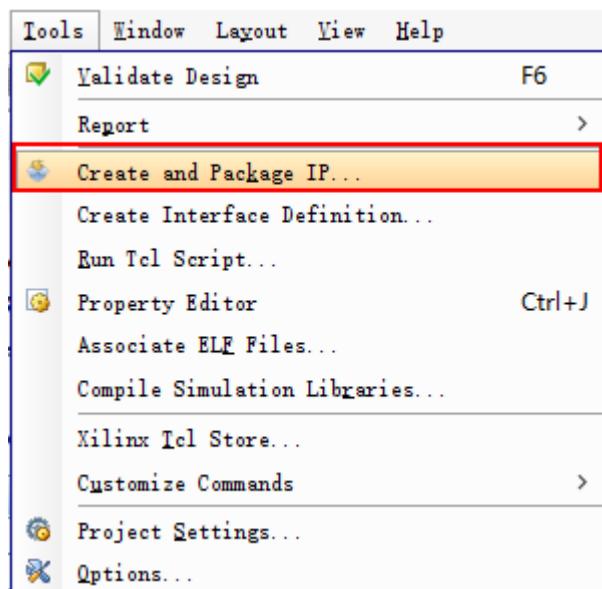
Step12: 选中 BD 文件, 右单击选择 Generate Output Products。



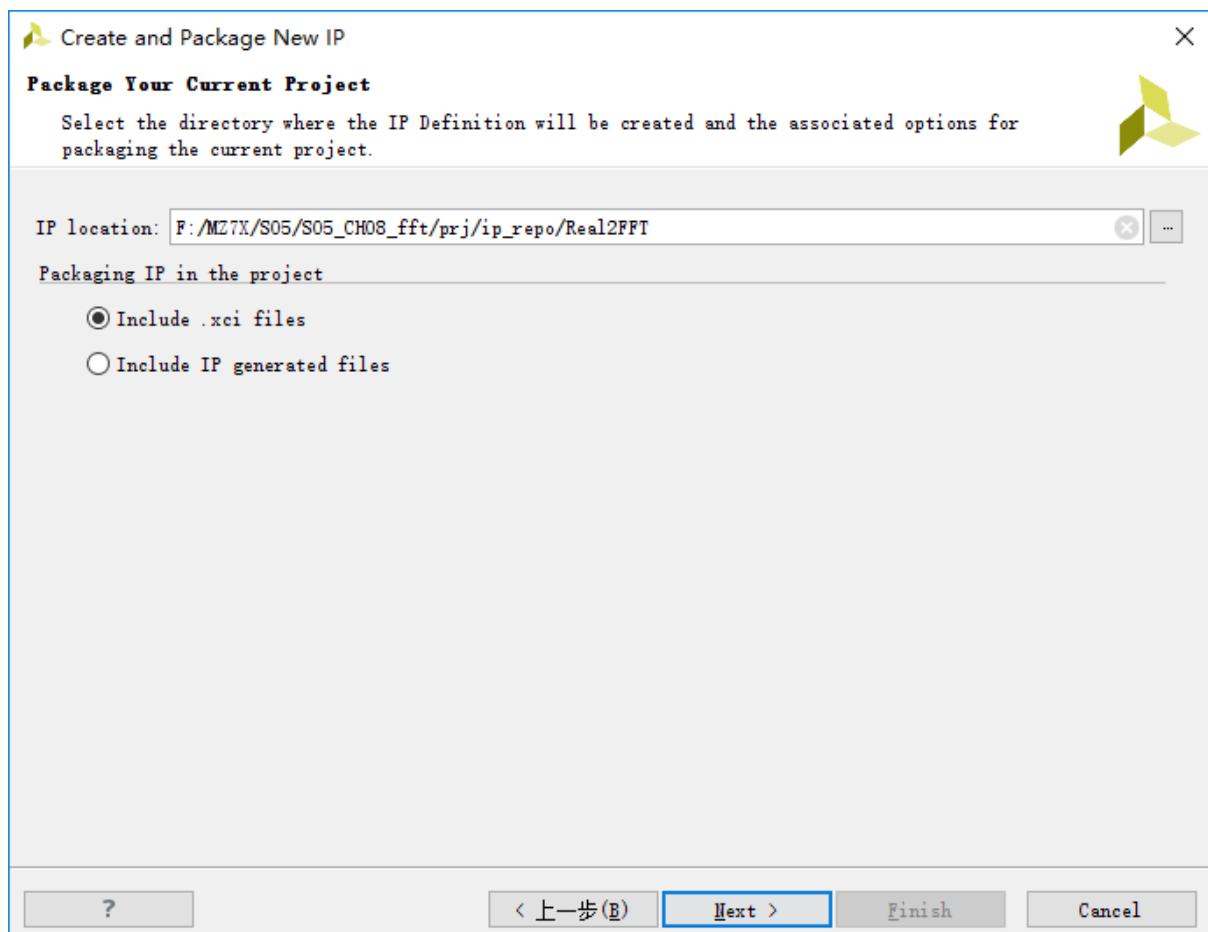
Step13: 单击 Create HDL Wrapper 生成顶层文件。



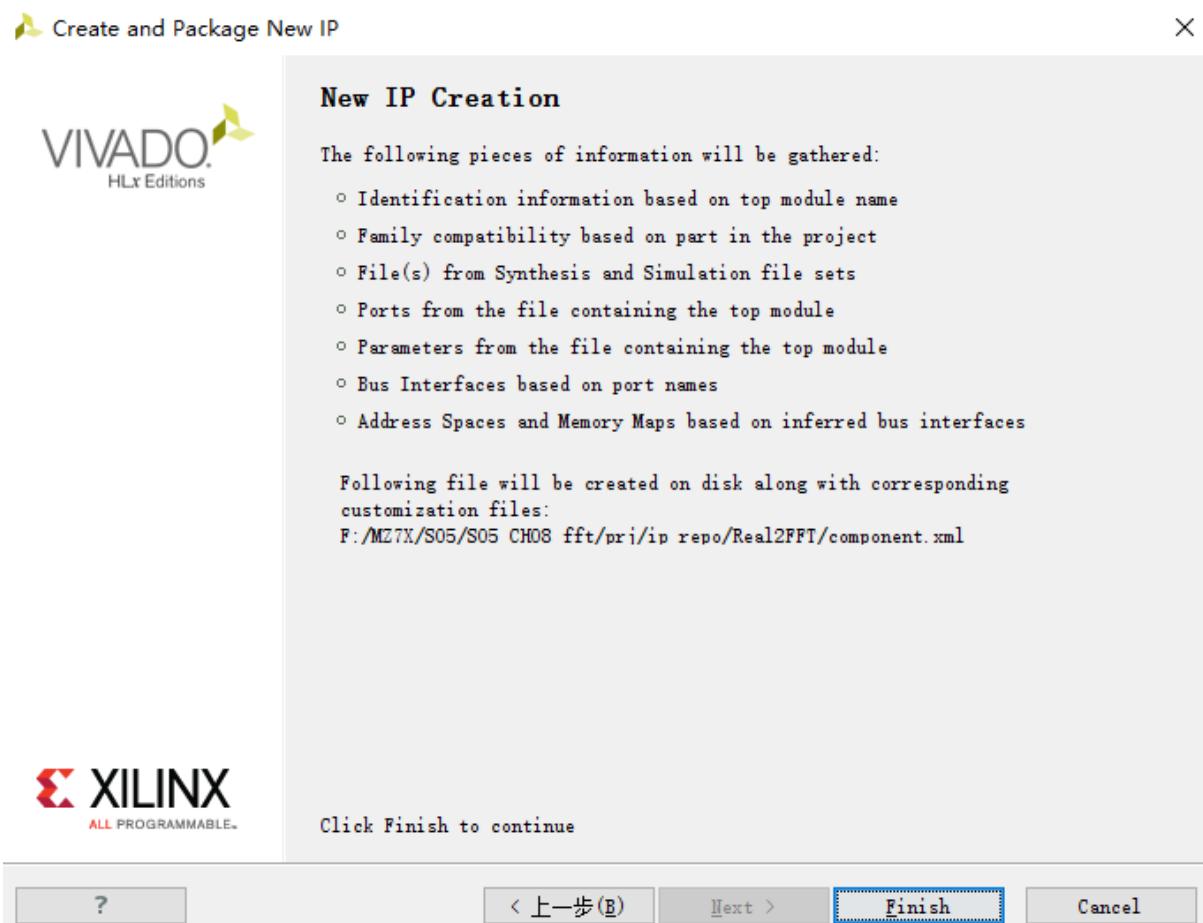
Step14: 接下来将其打包成一个 IP，单击 Tools 菜单下的 Create and package IP.. 命令。



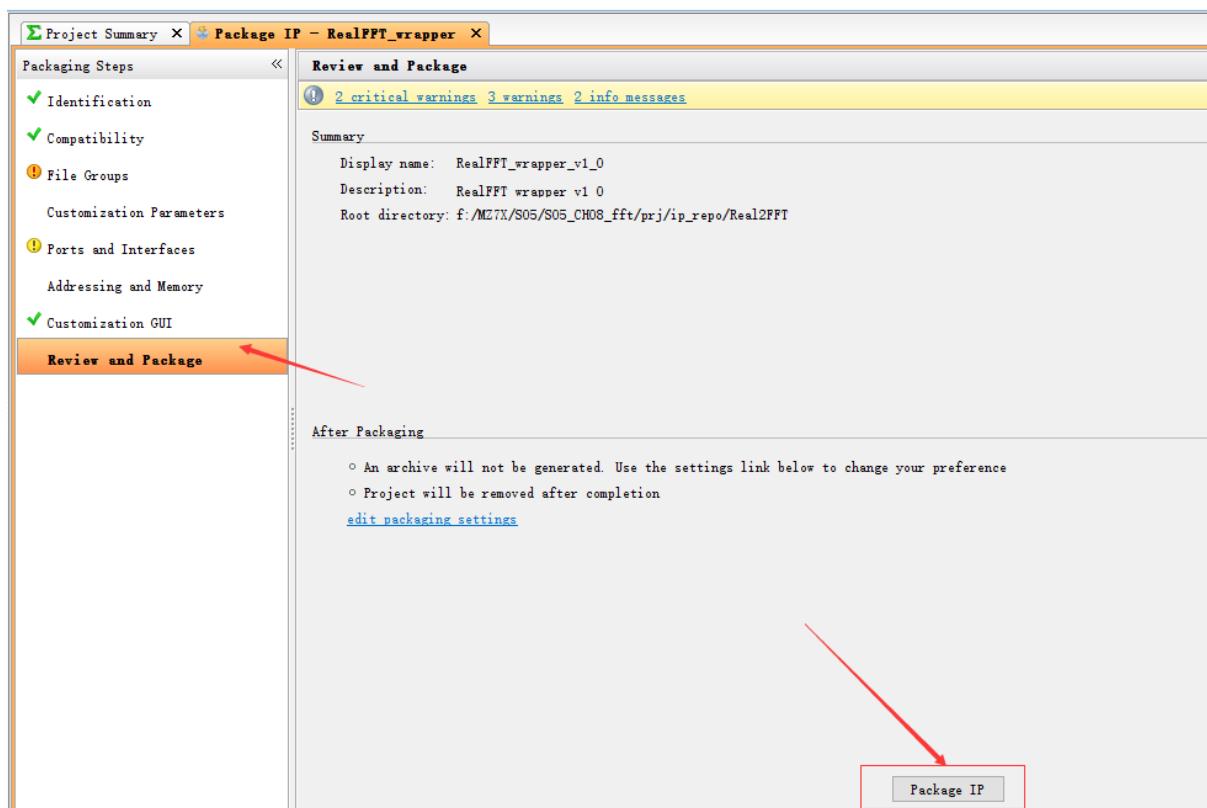
Step15: 单击 Next, 直到出现下图界面，按图中的设置方法设置。



Step16: 单击 next, 最后单击 Finish。



Step17: 在跳出来的页面当中，直接选择 Review and Package, 然后点击 Package IP 既完成了 IP 的封装。



8.4 本章小结

本章向大家介绍了 FFT 的原理，然后使用官方给出的代码使用 HLS 生成了两个 FFT 的 IP，之后在 VIVADO 中将其封装成了一个可供之后章节使用的 IP。本章需要重点掌握的是 FFT 的原理和熟练的掌握 HLS 的开发流程。本章参考了以下文献：

<http://www.xuebuyuan.com/539160.html>

<http://blog.csdn.net/sshcx/article/details/1651616>

S05_CH09_OTSU 自适应二值化

9.1 OTSU 自适应二值化原理简介

最大类间方差法是由日本学者大津于 1979 年提出的，是一种自适应的阈值确定的方法，又叫大津法，简称 OTSU。它是按图像的灰度特性，将图像分成背景和目标 2 部分。背景和目标之间的类间方差 σ^2 越大，说明构成图像的 2 部分的差别越大，当部分目标错分为背景或部分背景错分为目标都会导致 2 部分差别变小。因此，使类间方差最大的分割意味着错分概率最小。对于图像 $I(x, y)$ ，前景(即目标)和背景的分割阈值记作 T ，属于前景的像素点数占整幅图像的比例记为 ω_0 ，其平均灰度 μ_0 ；背景像素点数占整幅图像的比例为 ω_1 ，其平均灰度为 μ_1 。图像的总平均灰度记为 μ ，类间方差记为 σ^2 。假设图像的背景较暗，并且图像的大小为 $M \times N$ ，图像中像素的灰度值小于阈值 T 的像素个数记作 N_0 ，像素灰度大于阈值 T 的像素个数记作 N_1 ，则有：

$$\omega_0 = N_0 / M \times N \quad (1)$$

$$\omega_1 = N_1 / M \times N \quad (2)$$

$$N_0 + N_1 = M \times N \quad (3)$$

$$\omega_0 + \omega_1 = 1 \quad (4)$$

$$\mu = \omega_0 * \mu_0 + \omega_1 * \mu_1 \quad (5)$$

$$\sigma^2 = \omega_0 (\mu_0 - \mu)^2 + \omega_1 (\mu_1 - \mu)^2 \quad (6)$$

将式(5)代入式(6)，得到等价公式： $\sigma^2 = \omega_0 \omega_1 (\mu_0 - \mu_1)^2 \quad (7)$

采用遍历的方法得到使类间方差最大的阈值 T ，即为所求。

Otsu 算法步骤如下：

设图象包含 L 个灰度级 ($0, 1 \dots, L-1$)，灰度值为 i 的象素点数为 N_i ，图象总的象素点数为 $N = N_0 + N_1 + \dots + N_{L-1}$ 。灰度值为 i 的点的概率为：

$$P(i) = N(i) / N.$$

门限 t 将整幅图象分为暗区 c_1 和亮区 c_2 两类，则类间方差 σ^2 是 t 的函数：

$$\sigma^2 = a_1 * a_2 * (u_1 - u_2)^2 \quad (2)$$

式中， a_j 为类 c_j 的面积与图象总面积之比， $a_1 = \sum(P(i)) \quad i \rightarrow t$ ， $a_2 = 1 - a_1$ ； u_j 为类 c_j 的均值， $u_1 = \sum(i * P(i)) / a_1 \quad 0 \rightarrow t$ ，

$$u_2 = \sum(i * P(i)) / a_2, \quad t+1 \rightarrow L-1$$

该法选择最佳门限 t^* 使类间方差最大，即：令 $\Delta u = u_1 - u_2$ ， $\sigma^2 = \max\{a_1(t) * a_2(t) \Delta u^2\}$

9.2 HLS 实现

9.2.1 工程创建

Step1：打开 HLS，按照之前介绍的方法，创建一个新的工程，命名为 otsu_threshold。

Step2：右单击 Source 选项，选择 New File，创建一个名为 Top.cpp 的文件。

Step3:在打开的编辑区中，把下面的程序拷贝进去：

```
#include "top.h"
#include "hls_math.h"

#define Simulation 0
#if Simulation
    #include "iostream"
    using namespace std;
#endif

namespace hls
{
    template<int SRC_T, int DST_T,int ROW, int COL, int N>
    void threshold(Mat<ROW, COL, SRC_T> &_src,Mat<ROW, COL,
DST_T> &_dst,ap_uint<8> (&map) [N]) {
        const int NUM_STATES = 4; //必须为偶数
        Window<1,NUM_STATES,ap_uint<8> > addr_win; //声明1行4列，每个元素为无符号8bit的窗口

        //OSTU需要统计不同灰度值个数
        ap_uint<BitWidth<ROW*COL>::Value> hist_out[N]; //以行列乘积
        最大位宽为边界，如ROW*COL = 1920*1080 = 2073600 ,
        //尾位宽存储需21位，则ap_uint<BitWidth<ROW*COL>::Value> 等同于
        ap_uint<21>
        Window<1,NUM_STATES,ap_uint<BitWidth<ROW*COL>::Value> >
hist_win;
        ap_uint<BitWidth<ROW*COL>::Value> hist;
        ap_uint<8> addr;
        ap_uint<8> addr_last;
        ap_uint<BitWidth<ROW*COL>::Value> hist_last;
        ap_uint<8> addr_flag;
        ap_uint<BitWidth<ROW*COL>::Value> hist_flag;
        ap_uint<8> addr_w;
        ap_uint<BitWidth<ROW*COL>::Value> hist_w;
        ap_uint<8> threshold = 0;
        ap_uint<BitWidth<ROW*COL>::Value> tmp=0;
        float pixelPro[256];

        for(int i=0;i<NUM_STATES;i++) {
#pragma HLS UNROLL
            addr_win(0,i)=i;//NUM_STATES 初始化不同的地址
            hist_win(0,i)=0;
```

```
}

for(int i=0;i<N;i++) {
    hist_out[i] = 0;
    pixelPro[i] = 0.0f;
}

//轮询统计原图像中不同灰度值的个数
int cols=_src.cols;
int rows=_src.rows;
assert(rows <= ROW);
assert(cols <= COL);
loop_height: for(int i=0;i<rows;i++) {
    loop_width: for(int j=0;j<cols;j++) {

#pragma HLS PIPELINE
#pragma HLS LOOP_FLATTEN OFF
#pragma HLS DEPENDENCE array inter false
        ap_uint<4> flag=NUM_STATES;
        HLS_TNAME(SRC_T) tempsrc=0;
        HLS_TNAME(DST_T) tempdst=0;

        _src.data_stream[0] >> tempsrc;
        tempdst = map[tempsrc];
        _dst.data_stream[0] << tempdst;
        for (int m=0; m<NUM_STATES; m++) {
            if (tempsrc==addr_win(0,m)) {
                flag = m;
                break;
            }
        }

latency_region:{

#pragma HLS latency min=0 max=1
        addr_last = addr_win(0,NUM_STATES-1);
        hist_last = hist_win(0,NUM_STATES-1)+1;

        for (int m=NUM_STATES-1; m>0; m--) {
            addr = addr_win(0,m-1);
            hist = hist_win(0,m-1);
            if (m == NUM_STATES/2) {
                addr_w = addr;
                if (m == flag+1) {
                    hist_w = hist+1;
                }
            }
        }
    }
}
```

```
        } else {
            hist_w = hist;
        }
    }

    if (m==flag+1) {
        addr_flag = addr;
        hist_flag = hist+1;
        addr_win(0,m) = addr_flag;
        hist_win(0,m) = hist_flag;
    } else {
        addr_win(0,m) = addr;
        hist_win(0,m) = hist;
    }
}

if (flag==NUM_STATES) {
    hist_win(0,0) = hist_out[tempsrc]+1;
    addr_win(0,0) = tempsrc;
} else if (flag==NUM_STATES-1) {
    addr_win(0,0) = addr_last;
    hist_win(0,0) = hist_last;
} else if (flag>=NUM_STATES/2) {
    addr_win(0,0) = addr_flag;
    hist_win(0,0) = hist_flag;
} else {
    addr_win(0,0) = addr_w;
    hist_win(0,0) = hist_w;
}

hist_out[addr_w] = hist_w;
}
}

for (int m=0; m<NUM_STATES/2; m++) {
#pragma HLS PIPELINE
    hist_out[addr_win(0,m)]=hist_win(0,m);
}

//计算直方图映射表
float scale = 255.0f/(cols*rows);
ap_uint<BitWidth<ROW*COL>::Value> sum=0;
```

```
loop_normalize: for(int i=0;i<N;i++) {
#pragma HLS PIPELINE
    tmp = hist_out[i];
    pixelPro[i] = (float)(tmp) / (float)(cols*rows);
}

//经典ostu算法,得到前景和背景的分割
//遍历灰度级[0,255],计算出方差最大的灰度值,为最佳阈值
float w0, w1, u0tmp, ultmp, u0, u1, deltaTmp, deltaMax = 0.0f;
loop_forward:for(int i = 0; i < 256; i++) {
#pragma HLS loop_flatten off
    w0 = w1 = u0tmp = ultmp = u0 = u1 = deltaTmp = 0.0f;

    loop_front:for(int j = 0; j < i; j++) {
#pragma HLS PIPELINE II=5
        w0 += pixelPro[j];
        u0tmp += (float)j * pixelPro[j];
    }

    loop_back:for(int j = i; j < 256; j++) {
#pragma HLS PIPELINE II=5
        ultmp += (float)j * pixelPro[j];
    }

    w1 = 1 - w0;
    u0 = u0tmp / w0;
    u1 = ultmp / w1;

    //计算类间方差
    deltaTmp = w0*w1*(u0-u1)*(u0-u1);

    //找出最大类间方差以及对应的阈值
    if(deltaTmp > deltaMax) {
        deltaMax = deltaTmp;
        threshold = i;
    }
}
#endif Simulation
cout << "The Threshold is " << (int)(threshold) << endl;
#endif

loop_map:for(int i=0;i<N;i++) {
```

```
#pragma HLS PIPELINE
    tmp = hist_out[i];
    sum+=tmp;
    ap_uint<8> val=sr_cast< ap_uint<8> > (sum*scale);
    #if Simulation
        cout << "The data is " << (int)(threshold) << endl;
    #endif
    ap_uint<8> data_val = (val > threshold) ? 255 : 0;
    map[i]=data_val;
}
map[0]=0;

}

static ap_uint<8> array_data[256];
template<int SRC_T, int DST_T,int ROW, int COL>
void ostu_threshold(
    Mat<ROW, COL, SRC_T>      &_src,
    Mat<ROW, COL, DST_T>      &_dst)
{
#pragma HLS INLINE
    threshold(_src, _dst, array_data);
}
}

void hls_counter_color(AXI_STREAM_IN& INPUT_STREAM,
AXI_STREAM_OUT& OUTPUT_STREAM, int rows, int cols)
{
#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM

#pragma HLS RESOURCE core=AXI_SLAVE variable=rows
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cols
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=return
metadata="-bus_bundle CONTROL_BUS"

#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols

RGB_IMAGE img_0(rows, cols);
GRAY_IMAGE img_1(rows, cols);
```

```

GRAY_IMAGE img_2(rows, cols);
RGB_IMAGE img_3(rows, cols);

#pragma HLS dataflow
hls::AXIVideo2Mat(INPUT_STREAM, img_0);
hls::CvtColor<HLS_RGB2GRAY>(img_0, img_1);
hls::ostu_threshold(img_1, img_2);
hls::CvtColor<HLS_GRAY2RGB>(img_2, img_3); //将灰度图像转换为RGB
图像
hls::Mat2AXIVideo(img_3, OUTPUT_STREAM);
}

```

Step4：再在 Source 中添加一个名为 Top.h 的库函数，并添加如下程序：

```

#ifndef _TOP_H_
#define _TOP_H_

#include "hls_video.h"
#include "ap_int.h"
#include <math.h>

#define MAX_WIDTH 1920
#define MAX_HEIGHT 1080

#define INPUT_IMAGE           "car.bmp"
//#define INPUT_IMAGE          "test_1080p.bmp"
#define OUTPUT_IMAGE           "result_1080p.bmp"
#define OUTPUT_IMAGE_GOLDEN    "result_1080p_golden.bmp"

// typedef video library core structures
typedef hls::stream<ap_axiu<32,1,1,1> >           AXI_STREAM_IN;
typedef hls::stream<ap_axiu<32,1,1,1> >           AXI_STREAM_OUT;
typedef hls::Scalar<3, unsigned char>                 RGB_PIXEL;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3>     RGB_IMAGE;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1>     GRAY_IMAGE;
typedef hls::Scalar<1, unsigned char>                 GRAY_PIXEL;

typedef unsigned char        uchar;
typedef ap_uint<12>         int12_t; //自定?12位符整型

//顶层函数
void hls_counter_color(AXI_STREAM_IN& src_axi, AXI_STREAM_OUT&
dst_axi, int rows, int cols);

```

```
#endif
```

Step5: 在 Test Bench 中, 用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码:

```
#include "top.h"
#include "opencv_top.h"

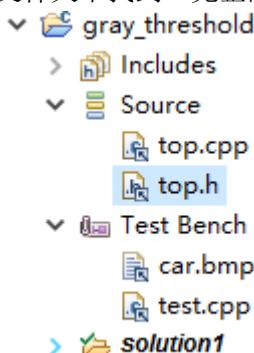
using namespace std;
using namespace cv;

int main (int argc, char** argv)
{
    //加载图像数据
    IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
1); //src->nChannels

    //使用HLS库进行处理
    AXI_STREAM_IN src_axi;
    AXI_STREAM_OUT dst_axi;
    IplImage2AXIVideo(src, src_axi);
    hls_counter_color(src_axi, dst_axi, src->height, src->width);
    AXIVideo2IplImage(dst_axi, dst);
    cvSaveImage(OUTPUT_IMAGE, dst);
    cvShowImage("hls_dst", dst);
    waitKey(0);

    //释放内存
    cvReleaseImage(&src);
    cvReleaseImage(&dst);
}
```

Step6: 在 Test Bench 中添加一张名为 car.bmp 的测试图片, 图片可以在我们提供的源程序中的 Image 文件夹中找到。完整的工程如下图所示:

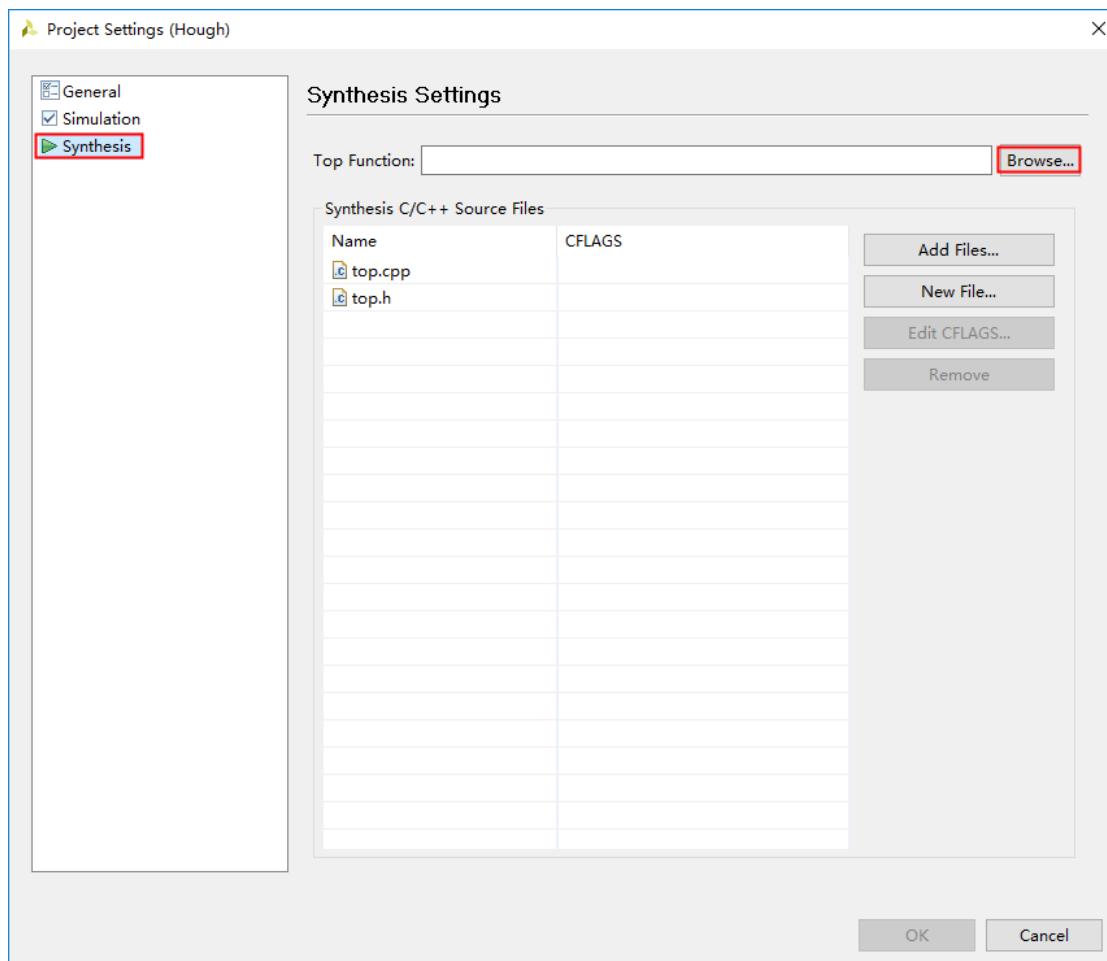


9.2.2 仿真及优化

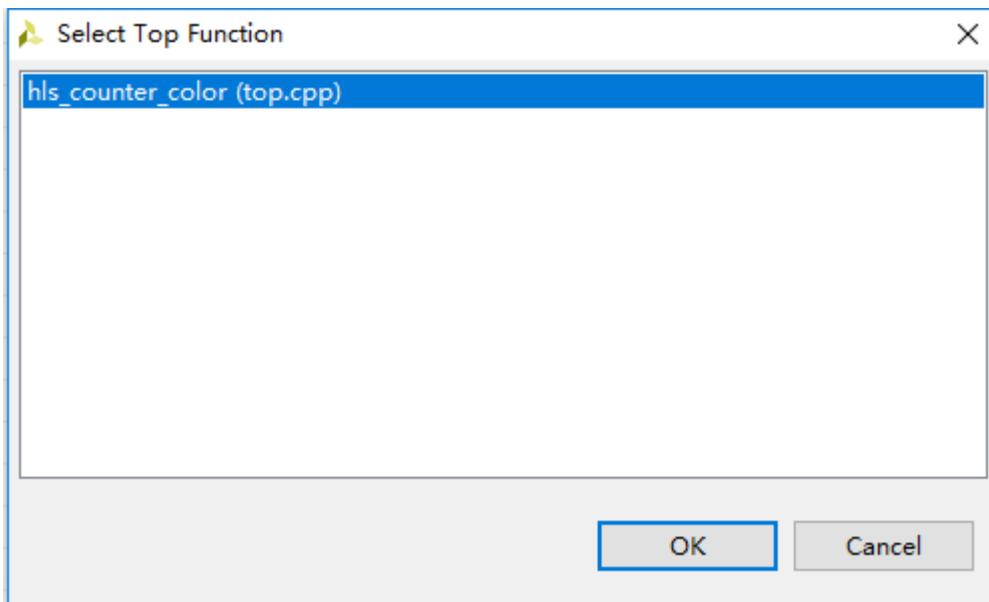
Step1：单击 Project settings 快捷键。



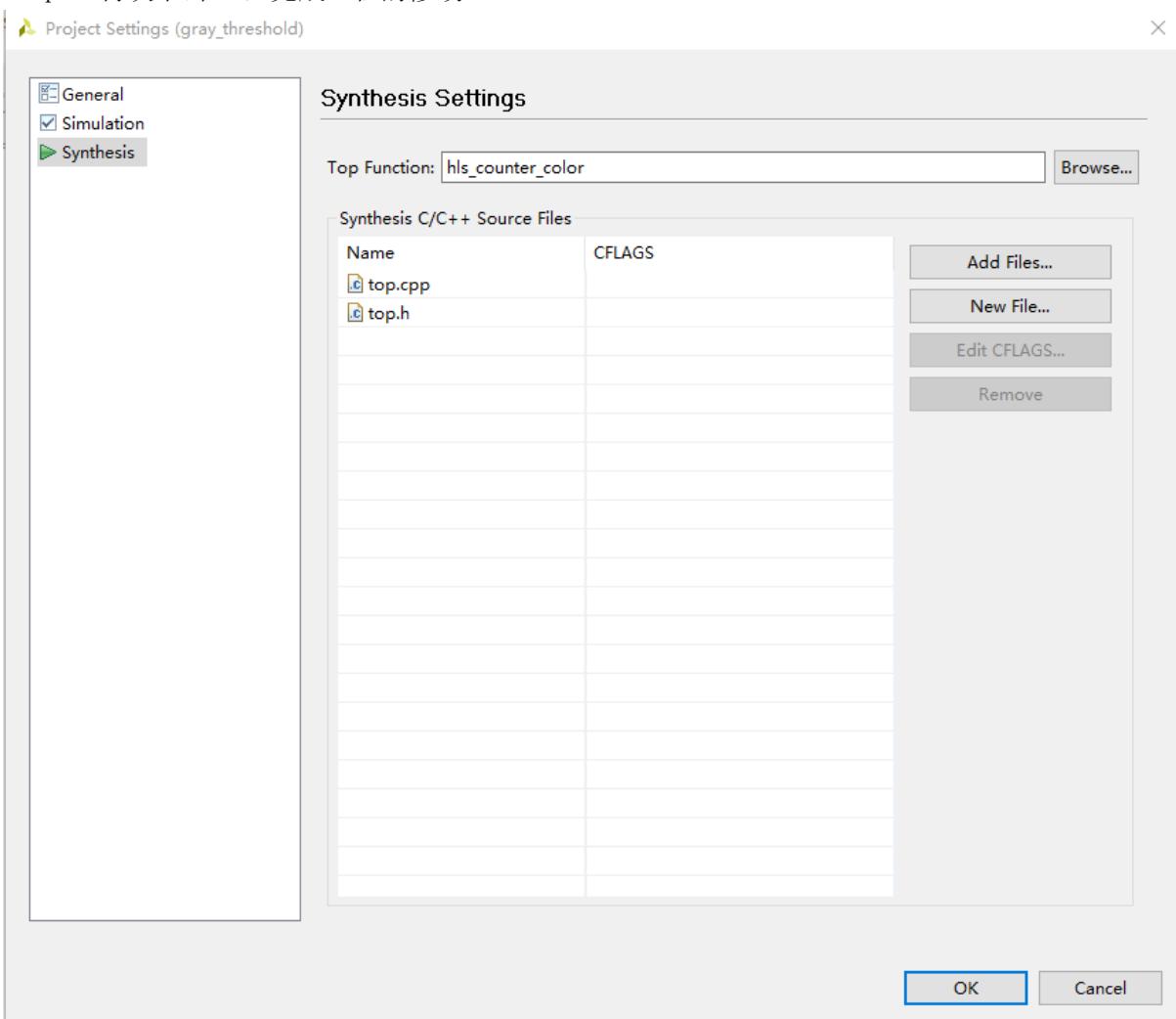
Step2：选择 Synthesis 选项，然后点击 Browse.. 指定一个顶层函数。



Step3：选定 hls_counter_color 为顶层函数，然后点击 O K。



Step4: 再次单击 OK，完成工程的修改。



Step5: 单击 开始综合。



综合报告如下：

□ Timing (ns)

□ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	12.91	1.25

□ Latency (clock cycles)

□ Summary

Latency		Interval		Type
min	max	min	max	
11325	3136341	11320	3136336	dataflow

□ Detail

+ Instance

+ Loop

Utilization Estimates

□ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	40	160
Instance	3	9	3713	7416
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	3	9	3753	7578
Available	280	220	106400	53200
Utilization (%)	1	4	3	14

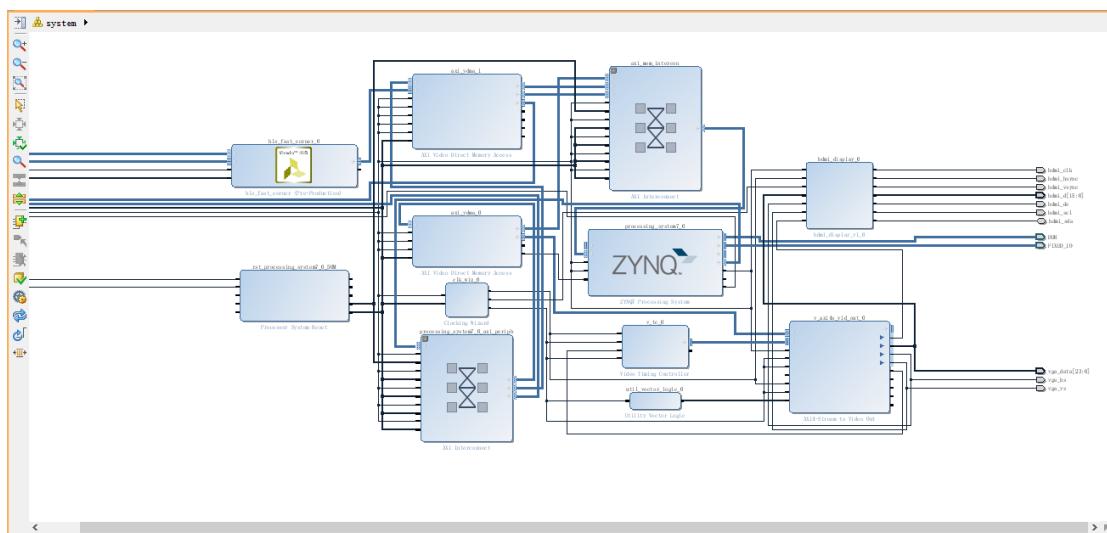
Step6：直接单击 开始进行仿真。系统运行结束后，处理结果如下所示：



9.3 硬件工程实现

9.3.1 硬件平台搭建

本章的硬件平台与 sobel 算子的实现几乎是一样的，只是在这里需要把 sobel 实现的 HLS IP 替换成本章 otsu 的 IP。系统整体硬件电路如下图所示：

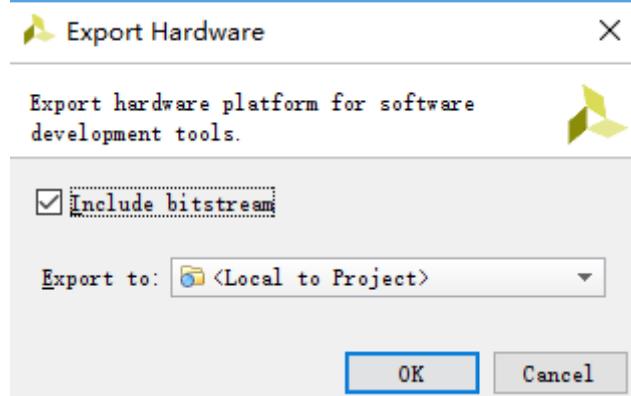


9.3.2 导入到 SDK

硬件平台搭建完成之后，让工程重新生成 Bit 文件，之后记得删除之前的 SDK 工程，以便 HLS IP 的驱动文件能正确的产生。

Step1：在工程文件夹中删除原来工程的 sdk 文件夹。

Step2：单击 File-Export-Export Hardware..。



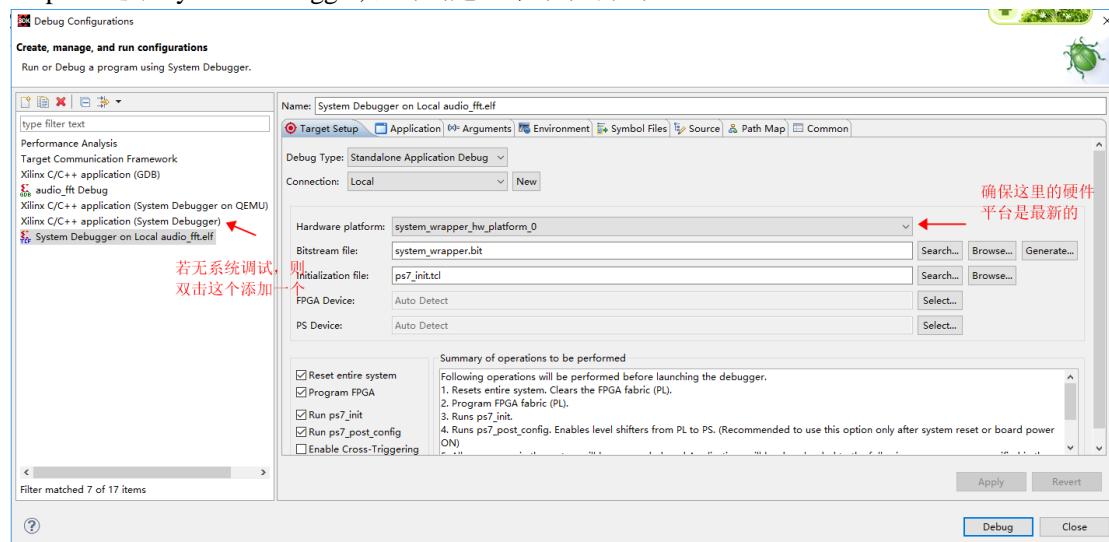
Step3：单击 File-Launch SDK。

Step4：新建一个名为 OTSU_Test 的空白工程。

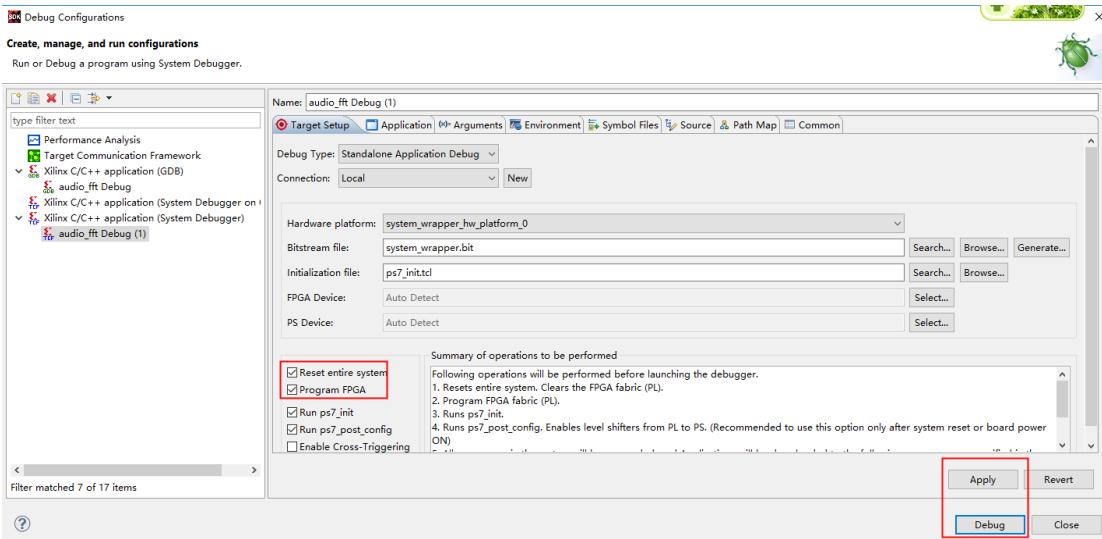
Step5：复制我们提供的 SDK 驱动文件，然后在 SDK 中，选中工程，在 Src 下直接按 Ctrl +V 将其复制到工程中来。

Step6：右击工程，选择 Debug as ->Debug configuration。

Step7：选中 system Debugger，双击创建一个系统调试。



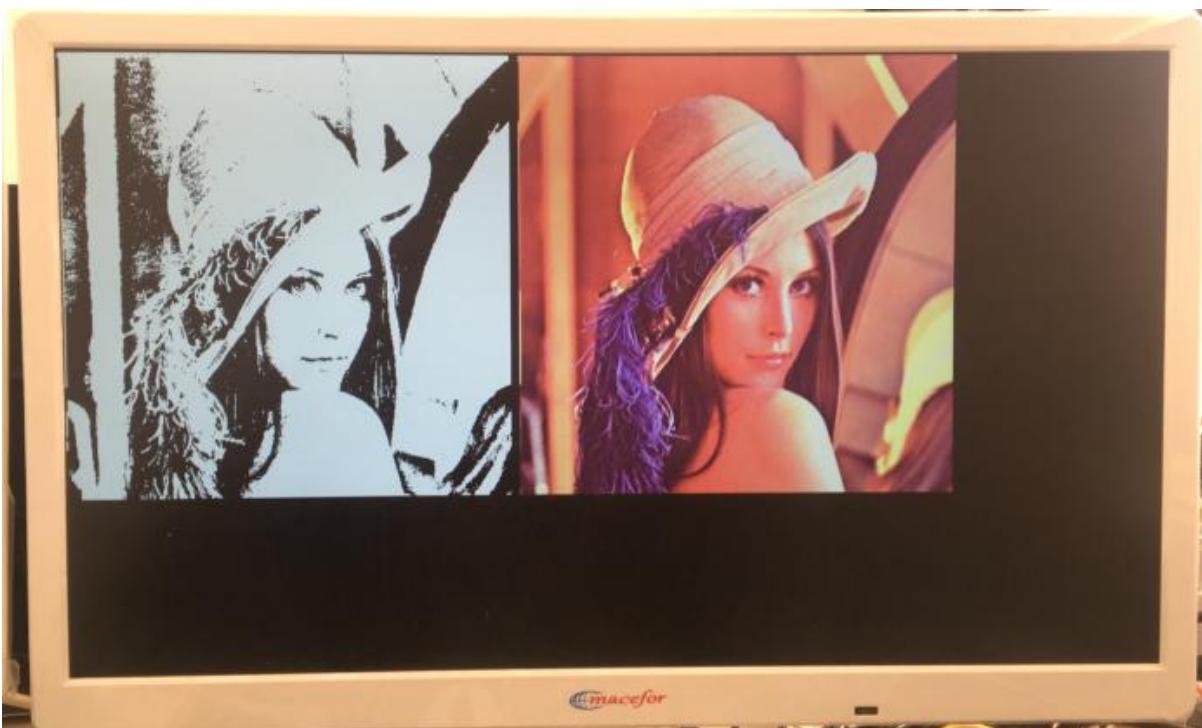
Step8：设置系统调试。



打开系统自带的窗口调试助手，点击运行按钮开始运行程序。



程序运行之后，运行结果如下图所示：



9.4 程序分析

本章的程序中 SDK 部分基本与第六章一致，这里我们重点讲解 HLS 部分的代码。与之前的代码不同的是，本章不是直接调用的官方视频库的函数，而是自己设计的一个函数。通过这个例子，大家以后在设计自身需要的程序功能的时候，就可以借鉴此程序了。本章内容 OTSU 的原理部分在前面已经进行过讲解，在此就直接切入主题，看看程序的实现。

首先我们来看看子程序的定义，其如下图所示：

```
namespace hls
{
    template<int SRC_T, int DST_T, int ROW, int COL, int N>
    void threshold(Mat<ROW, COL, SRC_T> &src, Mat<ROW, COL, DST_T> &dst, ap_uint<8> (&map)[N]) {
        const int NUM_STATES = 4; //必须为偶数?
        Window<1,NUM_STATES,ap_uint<8>> addr_win;
```

可以看到，这里用了一个 C++语言中的命令空间 namespace，空间名为 hls，其中包含了两个子函数（这里只截了一个，具体看程序），首先我们来看看 threshold 子函数，这里有三个参数，分别是 mat 型的输入图像和输出图像，以及 8 位无符号整型数据 N（具体意思程序内分析）。我们可以看到这三个参数都是从模板 中获取的参数。然后再看具体的程序实现。

```
    for(int i=0;i<NUM_STATES;i++) {
#pragma HLS UNROLL
        addr_win(0,i)=i;//NUM_STATES 初始化不同的地址
        hist_win(0,i)=0;
    }

    for(int i=0;i<N;i++){
        hist_out[i] = 0;
        pixelPro[i] = 0.0f;
    }
}
```

要理解这一段程序，则需要理解程序中变量的含义，其中 addr_win 为写入 hist_w 到 hist_out[N] 所对应的索引值，hist_win 写入的灰度数据，因此可知执行第一个 for 循环是为了初始化两个数组（索引窗口默认值为其地址，灰度窗口清零），紧接着来看第二个 for 循环，这个循环的意思就比较明了，对两个数组变量进行清零操作，其中 hist_out 是用来存放各个灰度值的像素个数，i 即为具体的灰度值（0-255），pixelpro 为各个灰度值出现的概率。

接下来看到下面这段程序：

```
int cols=_src.cols;
int rows=_src.rows;
assert(rows <= ROW);
assert(cols <= COL);
loop_height: for(int i=0;i<rows;i++) {
    loop_width: for(int j=0;j<cols;j++) {
#pragma HLS PIPELINE
#pragma HLS LOOP_FLATTEN OFF
#pragma HLS DEPENDENCE array inter false
        ap_uint<4> flag=NUM_STATES;
        HLS_TNAME(SRC_T) tempsrc=0;
        HLS_TNAME(DST_T) tempdst=0;

        _src.data_stream[0] >> tempsrc;
        tempdst = map[tempsrc];
        _dst.data_stream[0] << tempdst;
        for (int m=0; m<NUM_STATES; m++) {
            if (tempsrc==addr_win(0,m)) {
                flag = m;
                break;
            }
        }
    }
}
```

这段程序实现了将不同像素的灰度值写入到其对应的窗口中的功能，即是在原理处讲解的统计

不同灰度值的个数，其中重点需要理解的就是图中圈出部分，其中第一句是将输入图像数据赋值给 tempsrc，tempdst 是要在下一步赋值给输出图像数据的中间变量，输出图像数据是从映射表中获取对应的值。

```

latency_region: {
HLS latency min=0 max=1
    addr_last = addr_win(0,NUM_STATES-1); //统计的对应的灰度值的索引不变
    hist_last = hist_win(0,NUM_STATES-1)+1; //统计的不同灰度的像素值的个数加1

    for (int m=NUM_STATES-1; m>0; m--) {
        addr = addr_win(0,m-1);
        hist = hist_win(0,m-1);
        if (m == NUM_STATES/2)
            addr_w = addr;
        if (m == flag+1) {
            hist_w = hist+1;
        } else {
            hist_w = hist;
        }
    }
    if (m==flag+1) {
        addr_flag = addr;
        hist_flag = hist+1;
        addr_win(0,m) = addr_flag;
        hist_win(0,m) = hist_flag;
    } else {
        addr_win(0,m) = addr;
        hist_win(0,m) = hist;
    }
}
}

```

接着看到此处的 latency_regin，这是一个延时的处理模块，因为本程序中涉及到了数学运算并且需要统计一帧数据的灰度值的个数，所以必然不能实时处理一帧数据，需要考虑到延时，此程序参考至官方代码中的直方图均衡化的源代码，是官方提供的一种延时的处理方法，具体实现方法就是使用了一个 Num_States 来循环对四帧数据进行处理，让这四帧错开来处理，这样间接来实现延时（这是笔者自己的理解，如有错误，还请指正）。

接下来看到下面这一句：

```

for (int m=0; m<NUM_STATES/2; m++) {
HLS PIPELINE
    hist_out[addr_win(0,m)]=hist_win(0,m); //将更新后的灰度值的像素个数写入对应的灰度值的地址内
}
end

```

这一句的注释在图中已经给出来了，就是得出各个灰度值的像素个数然后写入对应的灰度值的地址内。

```

begin 累计归一化的直方图
float scale = 255.0f/(cols*rows);
ap_uint<BitWidth<ROW*COL>::Value> sum=0;
loop_normalize: for(int i=0;i<N;i++){
HLS PIPELINE
    tmp = hist_out[i];//获取不同灰度值的像素个数
    pixelPro[i] = (float)(tmp)/(float)(cols*rows);//计算不同灰度级的概率
}
end

```

之后在得出了上一步的结果之后，再由此求出各个灰度值的出现的概率。

```

//经典otsu算法,得到前景和背景的分割
//遍历灰度级[0,255],计算出方差最大的灰度值,为最佳阈值
float w0, w1, u0tmp, ultmp, u0, u1, deltaTmp, deltaMax = 0.0f;
loop_forward:for(int i = 0; i < 256; i++){
HLS loop_flatten off
    w0 = w1 = u0tmp = ultmp = u0 = u1 = deltaTmp = 0.0f;

    loop_front:for(int j = 0; j < i; j++){
HLS PIPELINE II=5
        w0 += pixelPro[j];
        u0tmp += (float)j * pixelPro[j];
    }

    loop_back:for(int j = i; j < 256; j++){
#pragma HLS PIPELINE II=5
        ultmp += (float)j * pixelPro[j];
    }

    w1 = 1 - w0;
    u0 = u0tmp / w0;
    u1 = ultmp / w1;

    //计算类间方差
    deltaTmp = w0*w1*(u0-u1)*(u0-u1);

    //找出最大类间方差以及对应的阈值
    if(deltaTmp > deltaMax){
        deltaMax = deltaTmp;
        threshold = i;
    }
}
}

```

接下来的这一段就是我们在原理部分介绍过的自适应二值化的计算方法，最后计算出的阈值为threshold。

```

//step3:begin 计算新的像素值
loop_map:for(int i=0;i<N;i++){
    #pragma HLS PIPELINE
    tmp = hist_out[i];
    sum+=tmp;//计算累积的灰度值
    ap_uint<8> val=sr_cast< ap_uint<8> > (sum*scale);//计算映射后的灰度值并取整
    #if Simulation
        cout << "The data is " << (int)(threshold) << endl;
    #endif
    ap_uint<8> data_val = (val > threshold) ? 255 : 0;//进行二值化处理
    map[i]=data_val;
}
map[0]=0;
}

```

最后，对图像数据进行二值化处理，得出新的像素数据，即为我们需要的二值化数据。

9.5 本章小结

本章为大家介绍了如何设计 OTSU 自适应二值化的功能，本章的功能是自主设计的一个程序功能，这对大家在设计自己的程序功能时能提供一定的帮助。另外本章中为大家介绍了一种时序延迟

的方法，希望大家可以在课后多加揣摩。

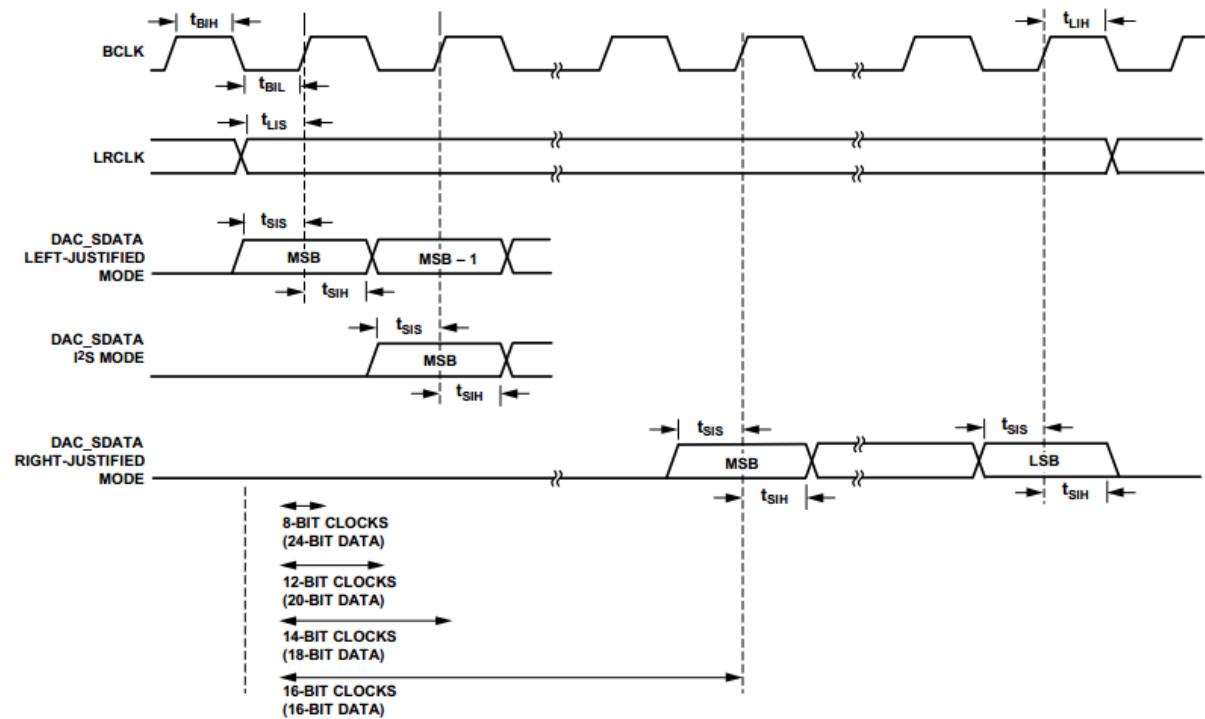
S05_CH10_音频滤波 (MZ7XB 不支持)

10.1 ADAU1761 简介

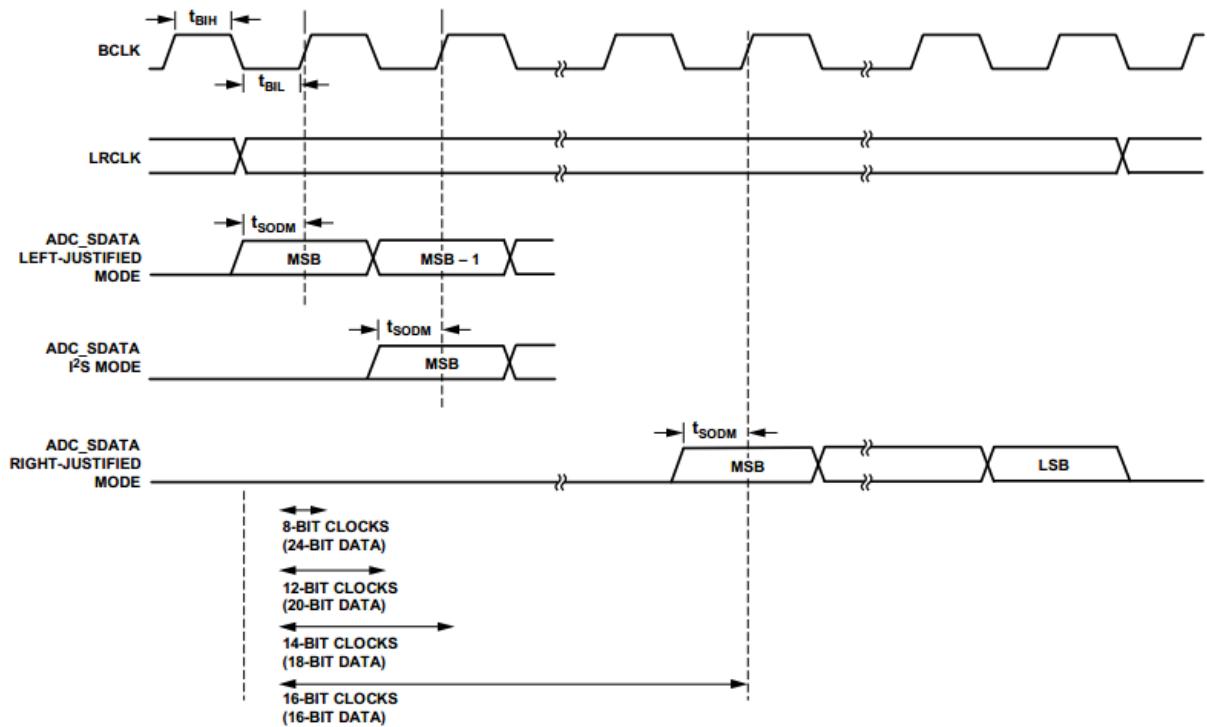
ADI 公司的 ADAU1761 是低功耗集成了数字音频处理的立体声 CODEC, 支持立体声 48kHz 录音, 1.8V 播放时的功耗为 14mW. 立体声 ADC 和 DAC 支持取样速率从 8kHz 到 96kHz 以及数字音量控制。音频处理 SigmaDSP 核具有 28 位处理能力, 能使设计者补偿麦克风, 扬声器, 放大器和听觉环境中现实世界的限制, 通过均衡, 多频带压缩, 限幅和第三方算法来极大地改善音频质量. 主要应用在智能手机/多媒体手机, 数码相机/数码摄像机, 手提媒体播放器/手提音频播放器等.

10.1.1 ADAU1761 收发时序

音频输入采样时序:



音频输出时序：



10.1.2 ADAU1761 时钟和采样率

ADAU1761 内部有两个时钟控制寄存器 R0 和 R1 来控制其时钟和采样率的大小。ADAU1761 时钟树原理图如下图所示：

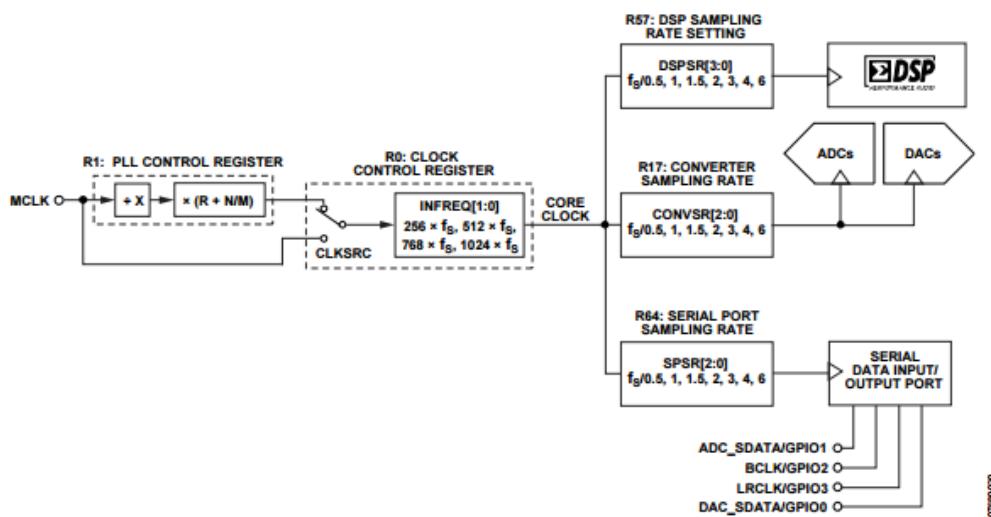


Figure 30. Clock Tree Diagram

在上图中 core clock 就是我们端口，转换器和 DSP 的时钟源。通过控制 R0 和 R1 可以选择 core clock 是通过 p11 生成还是直接通过 MCLK 得到。R0 和 R1 的寄存器如下图所示：

Table 12. Clock Control Register (Register R0, Address 0x4000)

Bits	Bit Name	Settings
3	CLKSRC	0: Direct from MCLK pin (default) 1: PLL clock
[2:1]	INFREQ[1:0]	00: $256 \times f_s$ (default) 01: $512 \times f_s$ 10: $768 \times f_s$ 11: $1024 \times f_s$
0	COREN	0: Core clock disabled (default) 1: Core clock enabled

Table 15. PLL Control Register (Register R1, Address 0x4002)

Bits	Bit Name	Description
[47:32]	M[15:0]	Denominator of the fractional PLL: 16-bit binary number 0x00FD: M = 253 (default)
[31:16]	N[15:0]	Numerator of the fractional PLL: 16-bit binary number 0x000C: N = 12 (default)
[14:11]	R[3:0]	Integer part of PLL: four bits, only values 2 to 8 are valid 0010: R = 2 (default) 0011: R = 3 0100: R = 4 0101: R = 5 0110: R = 6 0111: R = 7 1000: R = 8

Rev. C | Page 27 of 92

ADAU1761

Bits	Bit Name	Description
[10:9]	X[1:0]	PLL input clock divider 00: X = 1 (default) 01: X = 2 10: X = 3 11: X = 4
8	Type	PLL operation mode 0: Integer (default) 1: Fractional
1	Lock	PLL lock (read-only bit) 0: PLL unlocked (default) 1: PLL locked
0	PLLEN	PLL enable 0: PLL disabled (default) 1: PLL enabled

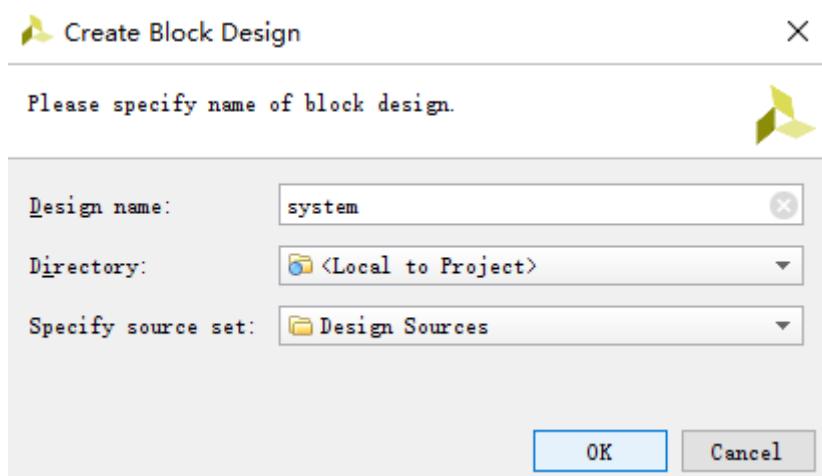
通过这两个寄存器的设置，我们就可以知道 ADAU1761 的采样率和时钟情况，具体的分析我们放在程序部分讲解。

10.2 硬件平台的搭建

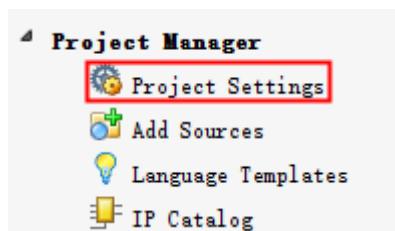
这一章可以算是对前面的实验的一个综合，涉及到 Audio 控制、OLED 显示控制以及 HLS 生成的 IP 核的使用，操作步骤如下：

Step1: 打开 VIVADO 软件, 创建一个名为 miz_sys 的工程, 根据自身的硬件平台正确配置芯片型号。

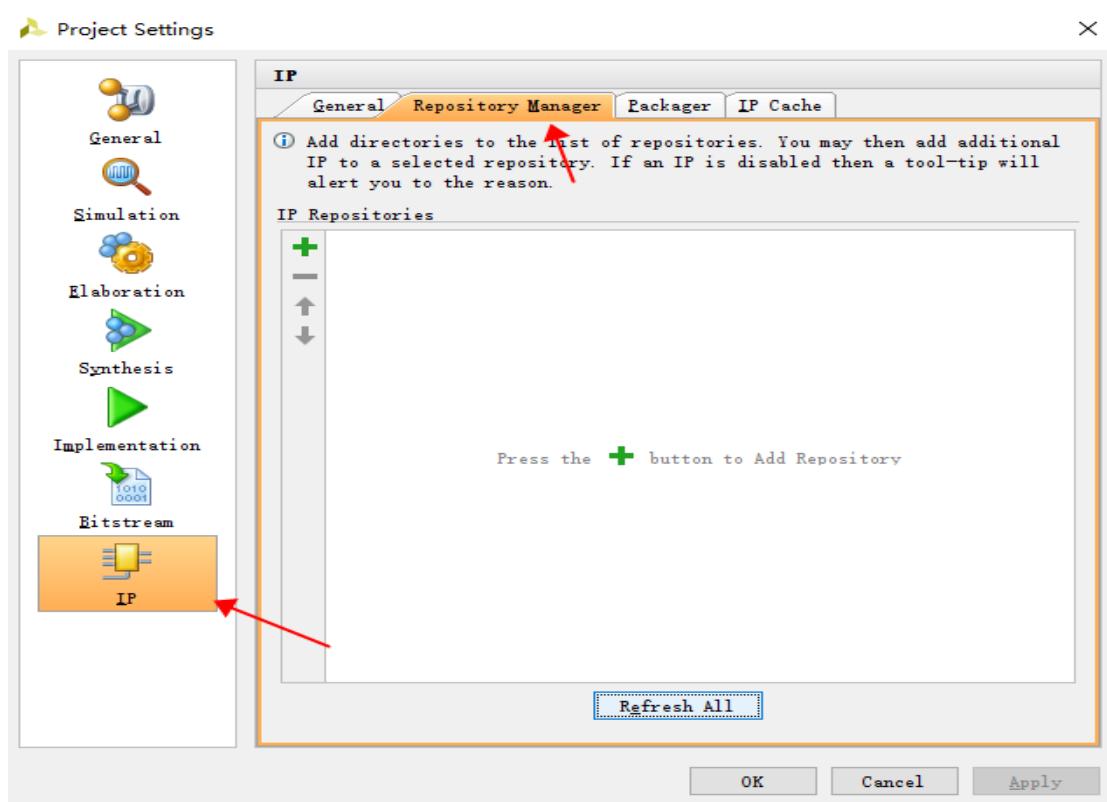
Step2: 双击 Create Block Design, 创建一个名为 system 的 BD 文件。



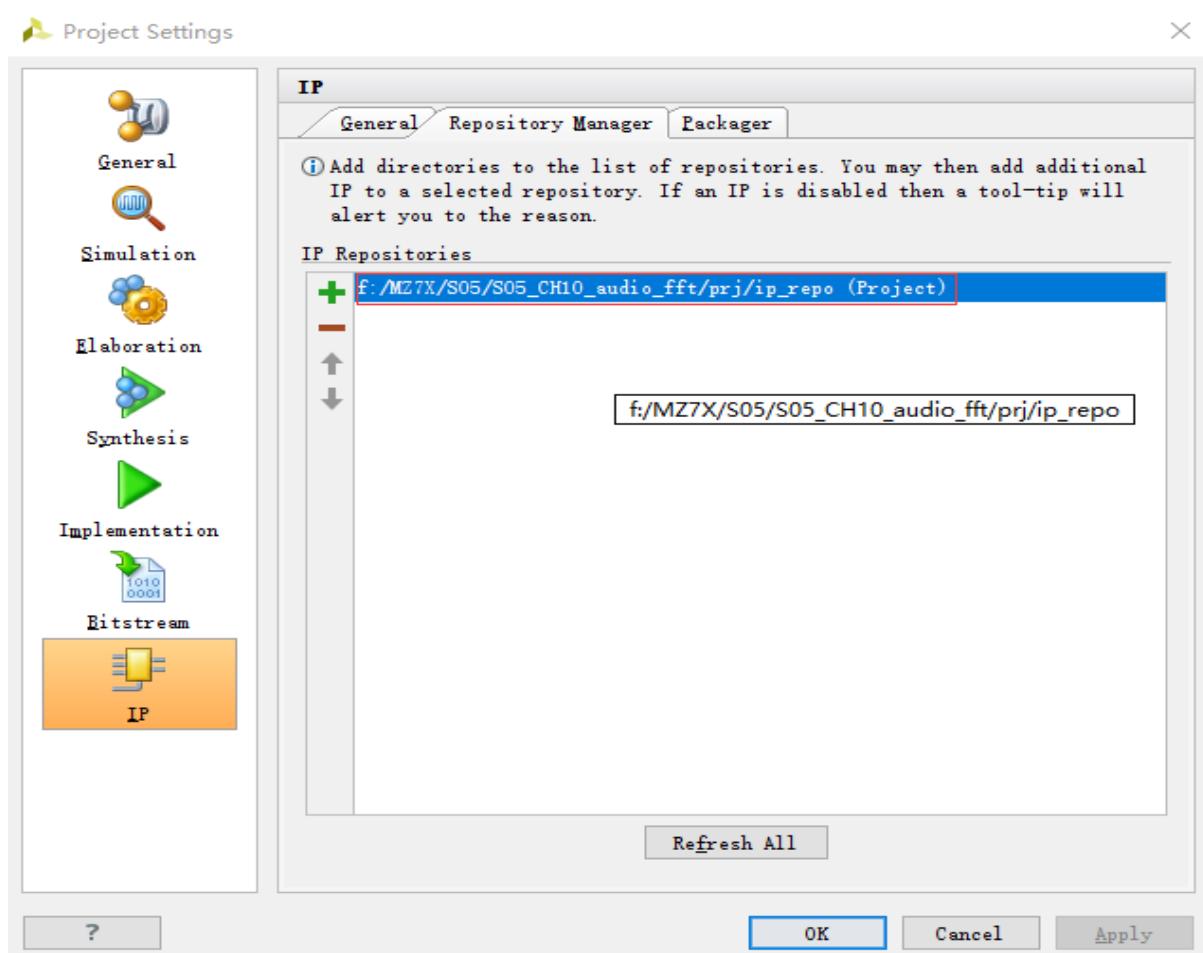
Step3: 在 Project Manager 设置区单击 Project settings。



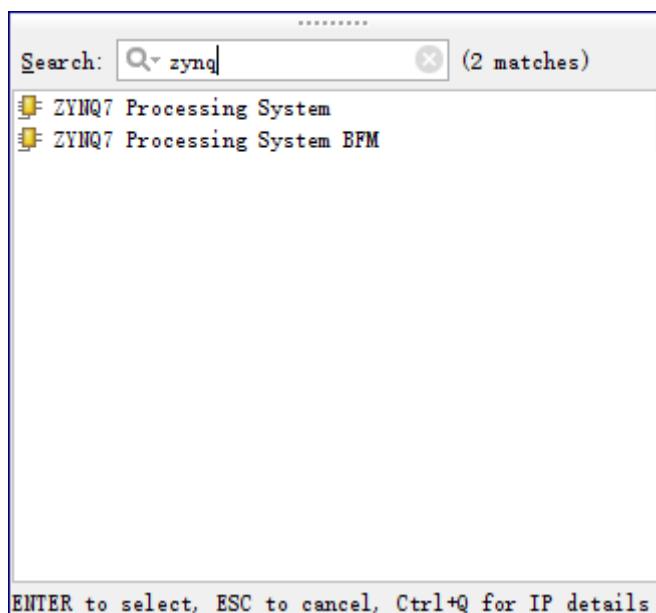
Step4: 在弹出的窗口中, 如下图设置:



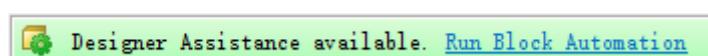
Step5：在我们提供的源程序对应章节的文件夹中找到 miz_ip_lib 文件夹，此文件夹存放了我们要用到的 IP，这其中就包括了我们第八章封装好了的 FFT IP 和我们之前用到过的 OLED IP，单击+图标，将 miz_ip_lib 文件夹添加到 IP 库当中，系统会自动扫描其中的 IP，之后单击 OK 完成修改。



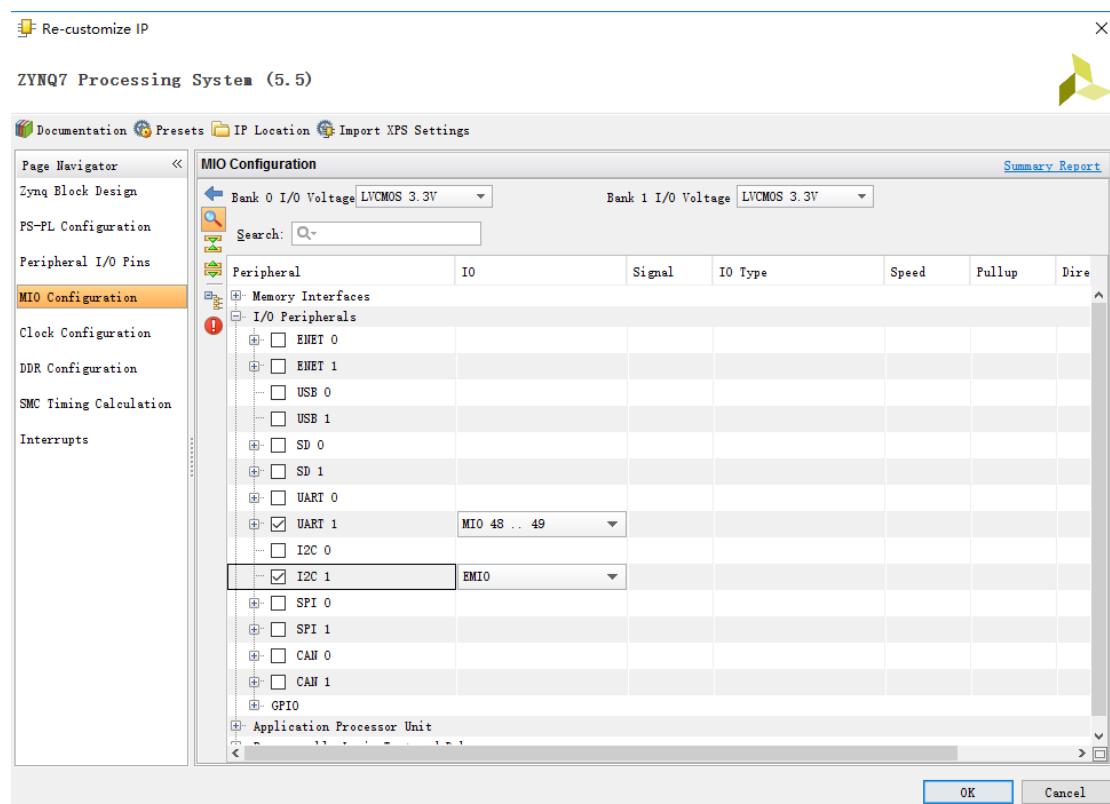
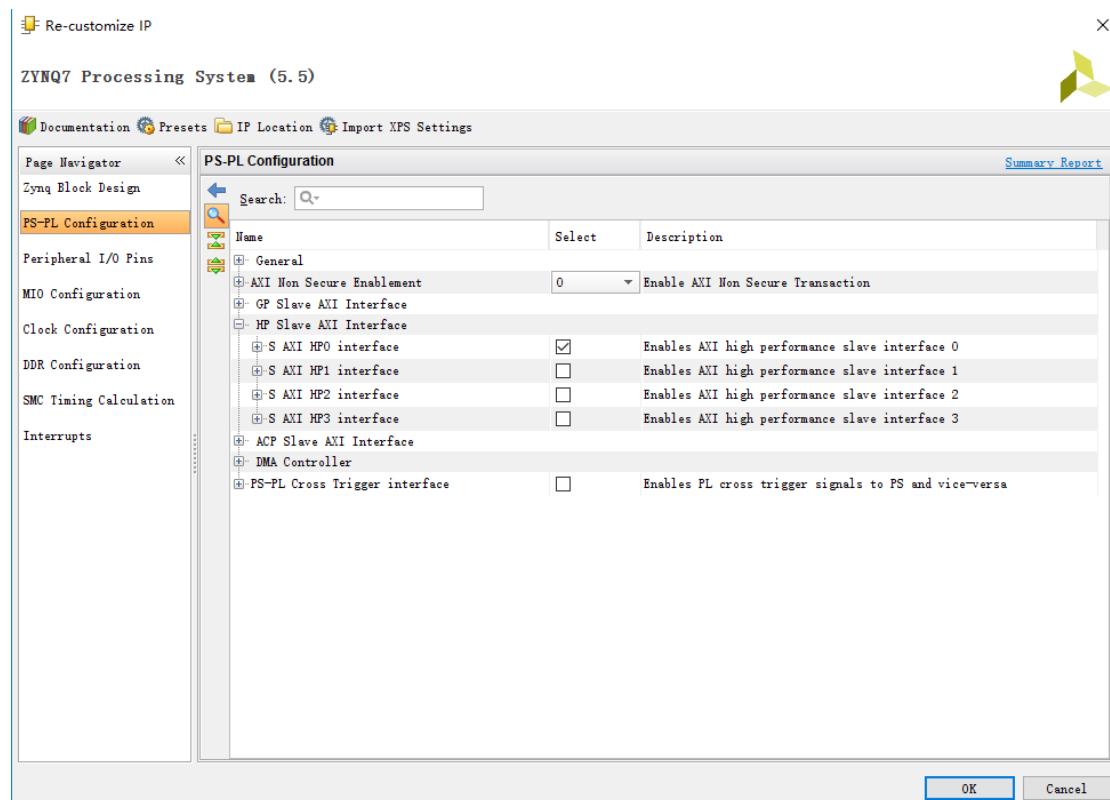
Step6: 单击 图标添加一个 ZYNQ Processing System IP。

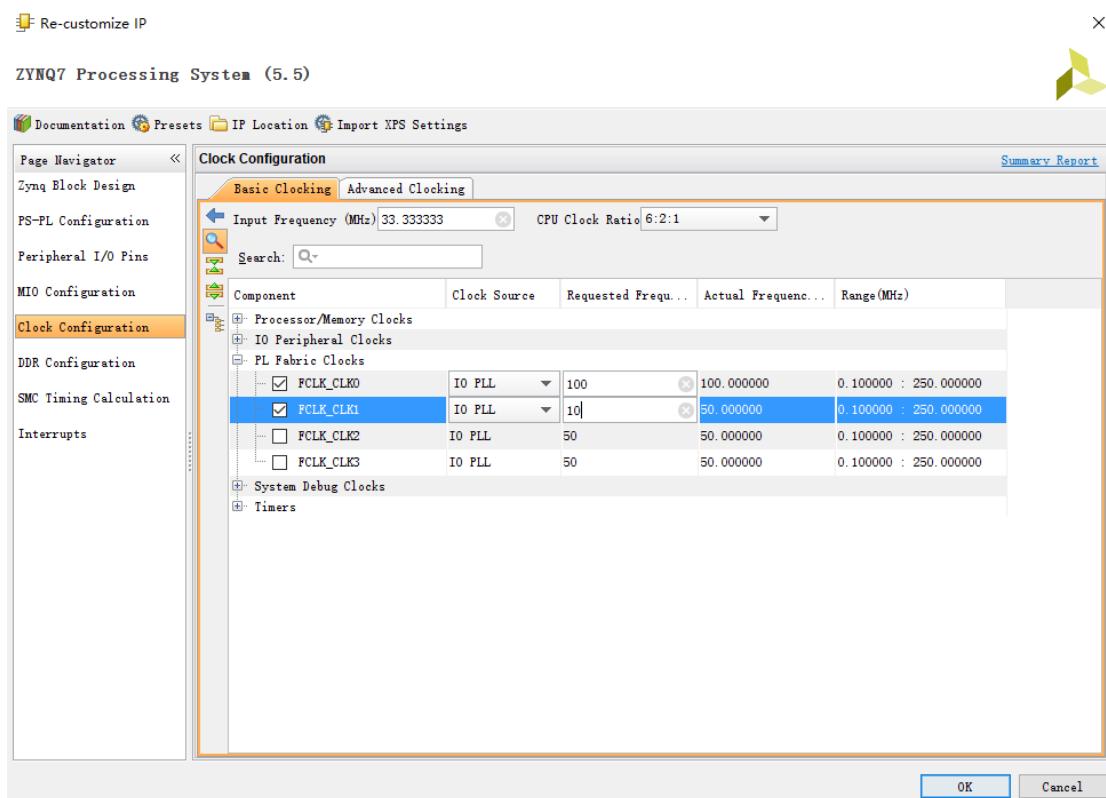


Step7: 单击 Run Block Automation, 在弹出来的窗口中直接单击 OK。

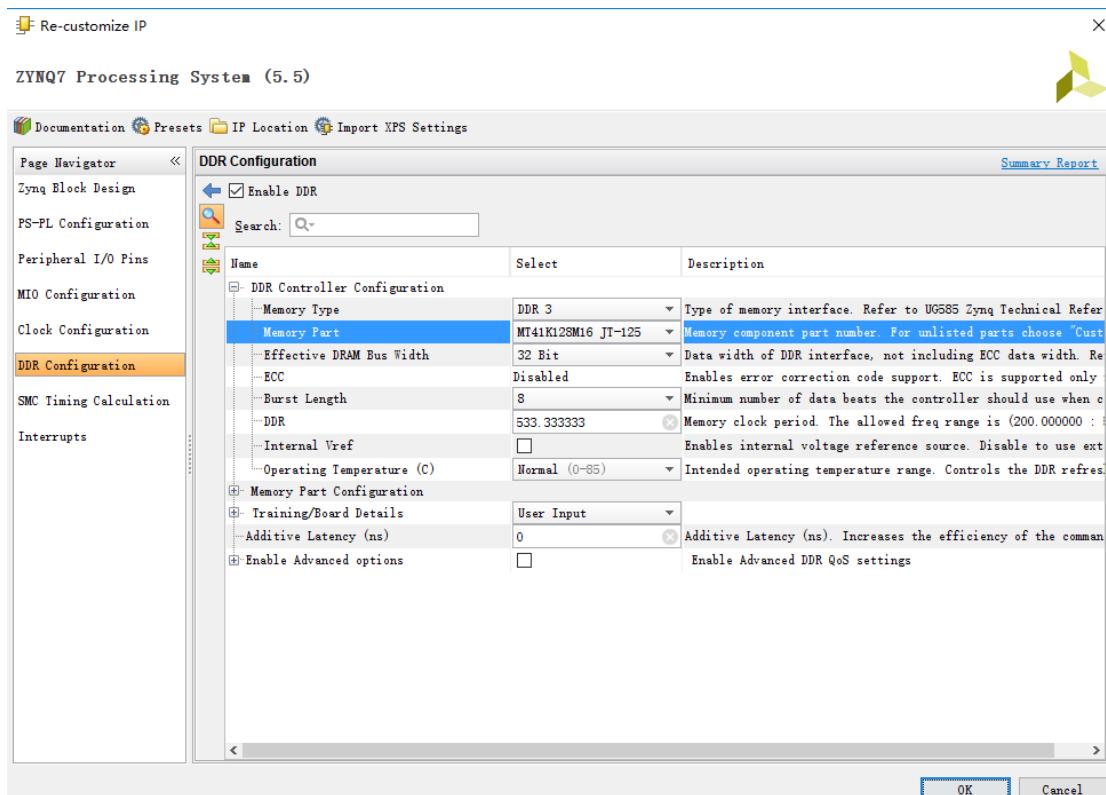


Step8：根据下图设置 ZYNQ Processing system。

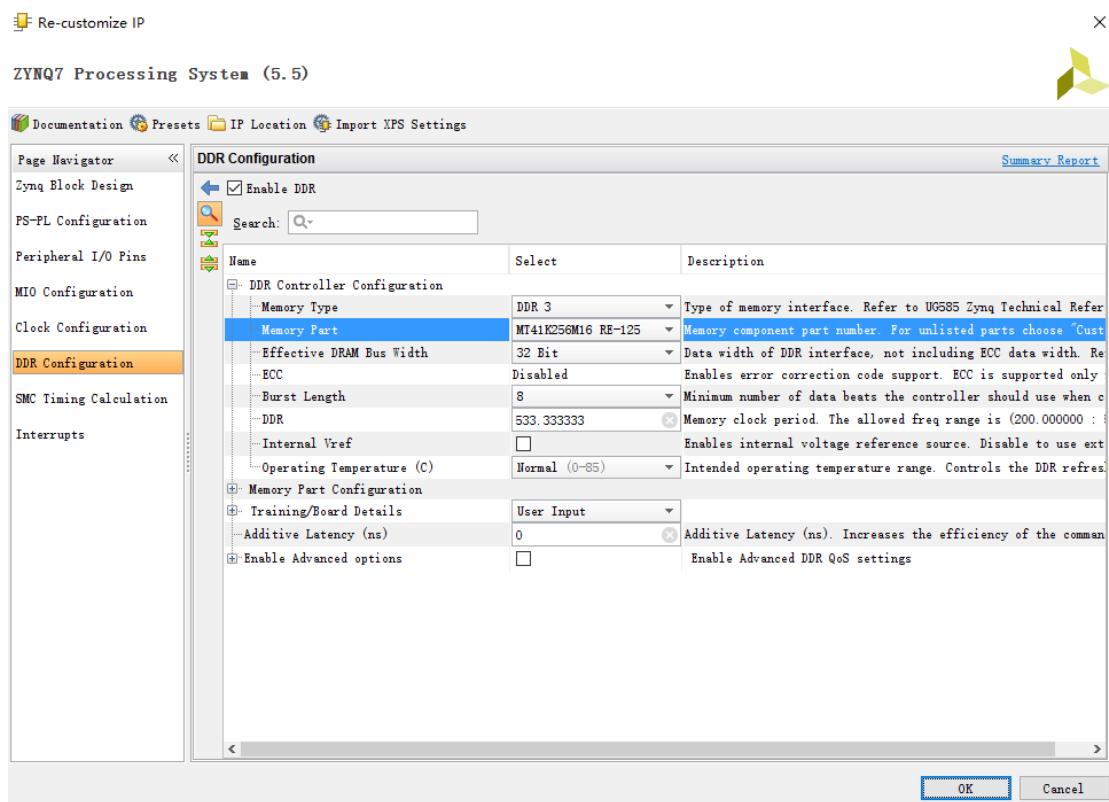




MZ701A-MINI 内存型号如下设置：

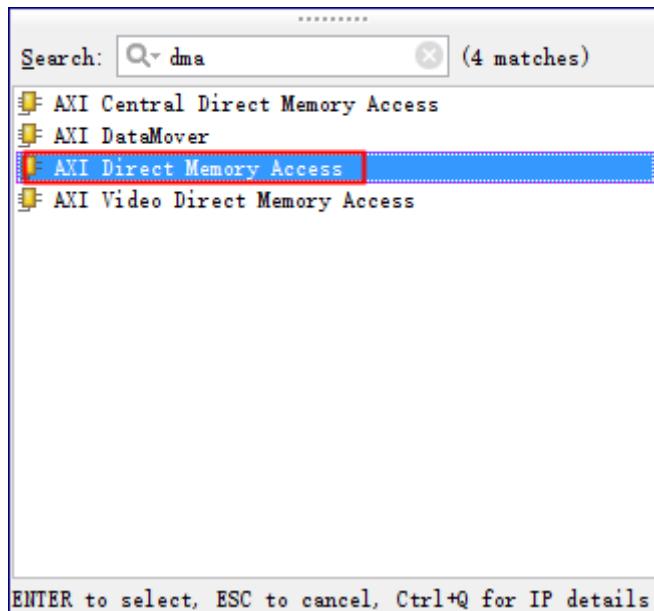


MZ701A、MZ702A 内存型号如下设置：

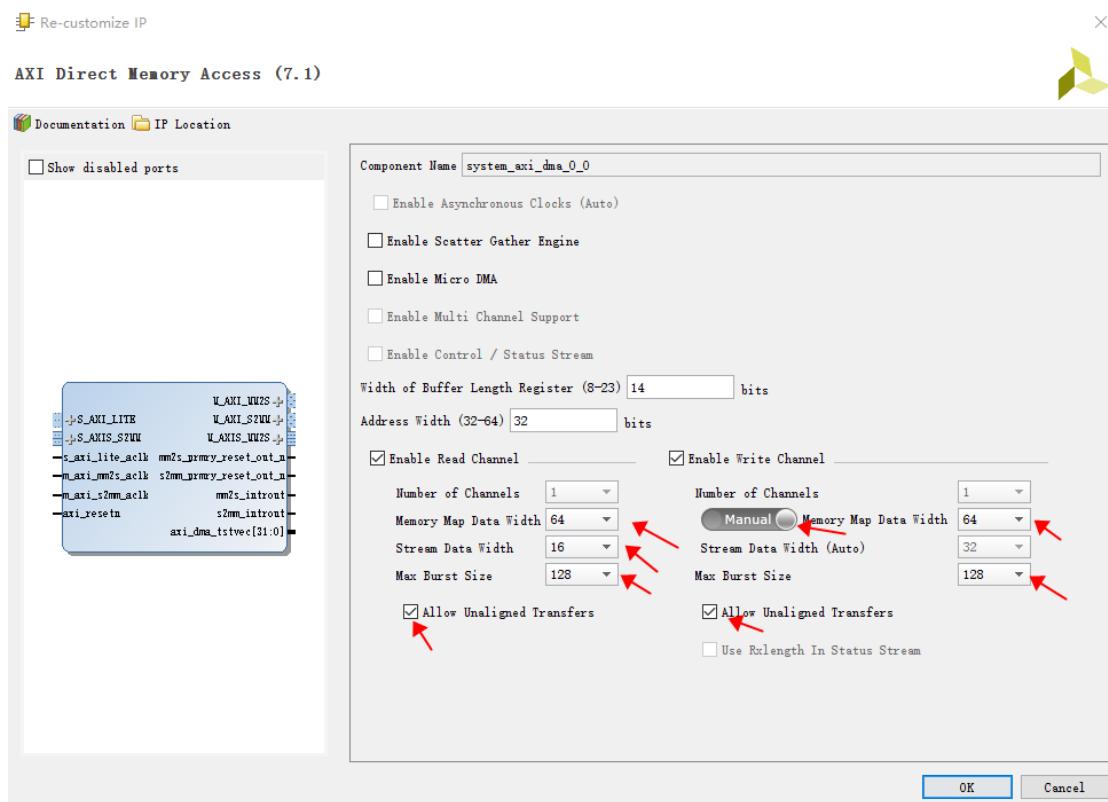


全部设置好了之后单击 OK 完成修改。

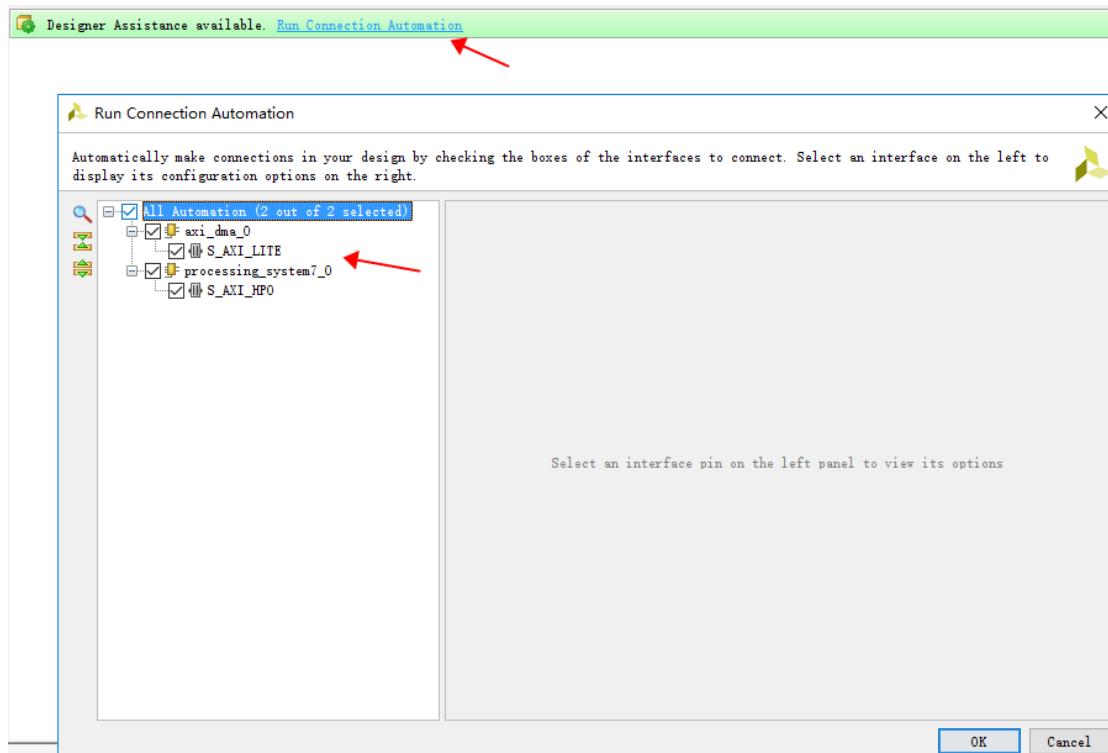
Step9：根据上面讲过的方法，添加一个 DMA 的 I P。



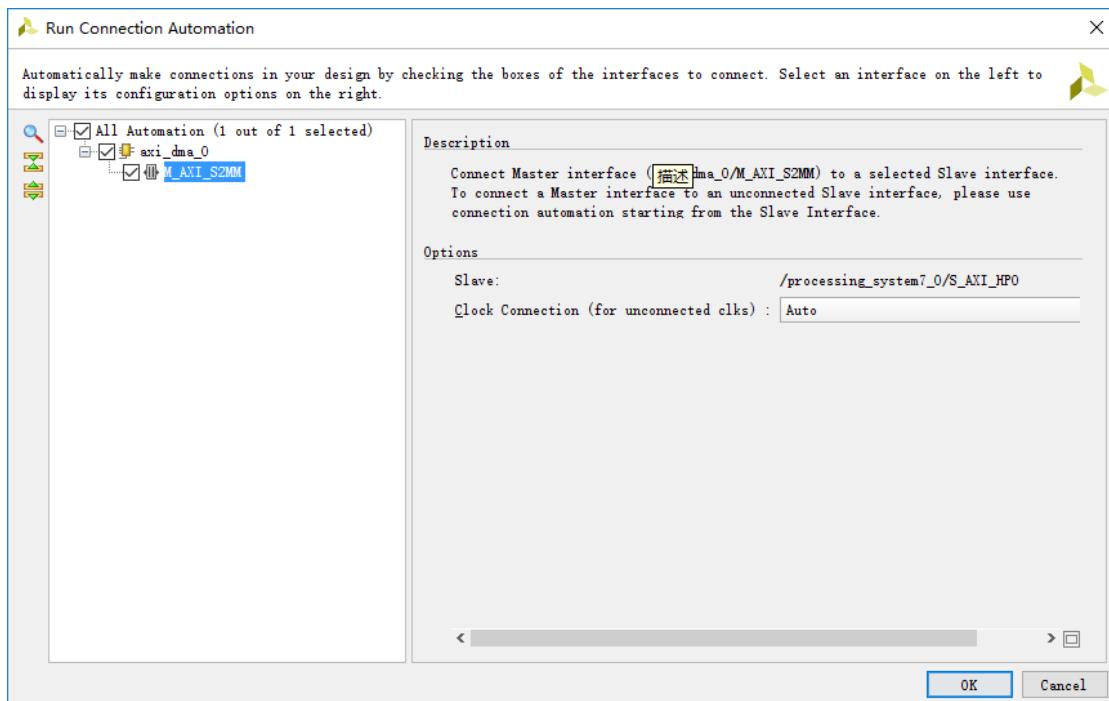
Step10：如下图所示配置 DMA，然后单击 OK。



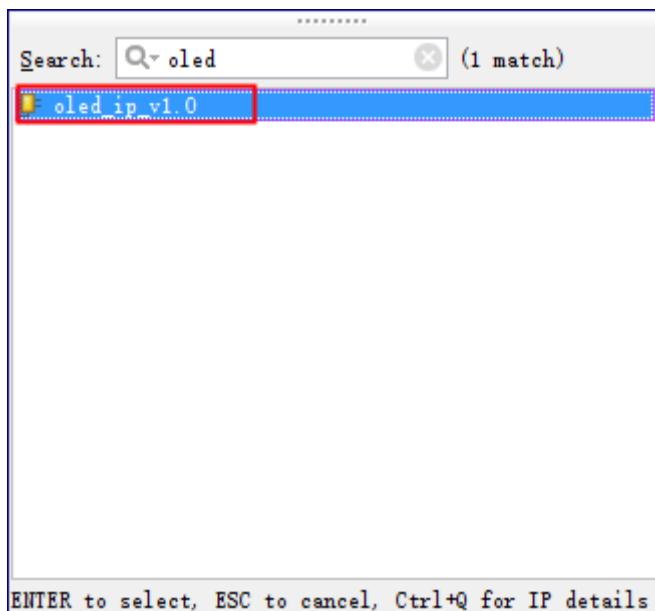
Step11: 单击 Run connection Automation。



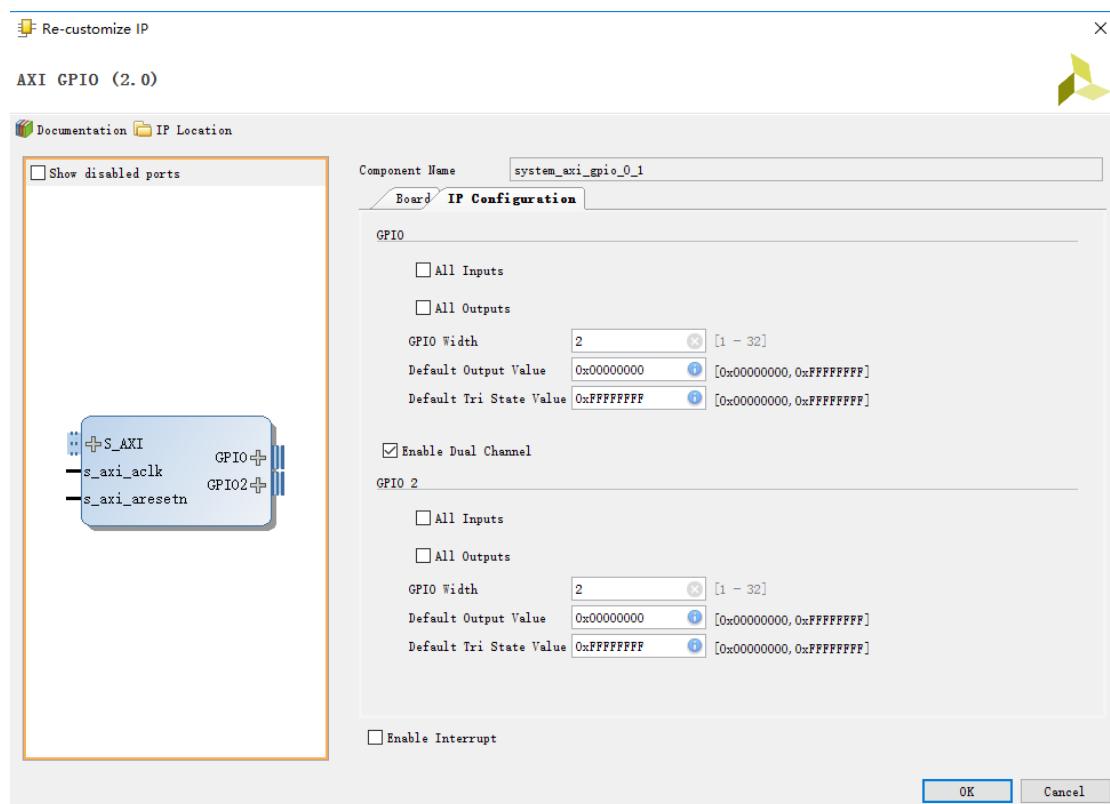
Step12: 再次单击 Run connection Automation。



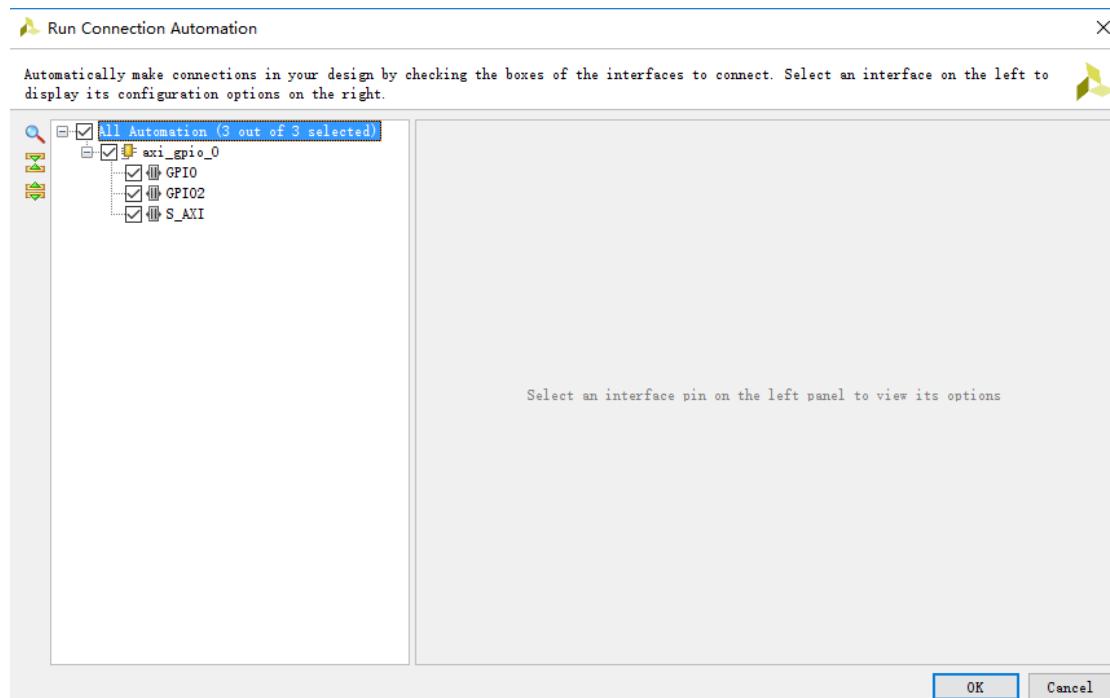
Step13: 添加一个oled IP，然后单击 run connection Automation。



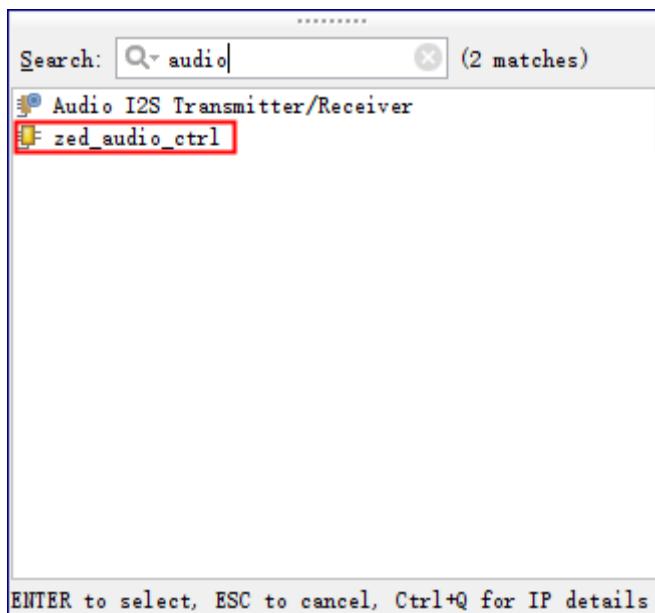
Step14: 添加一个AXI GPIO IP，并对其进行配置。



Step15: 单击 Run connection Automation, 然后按下图设置, 最后单击 OK。

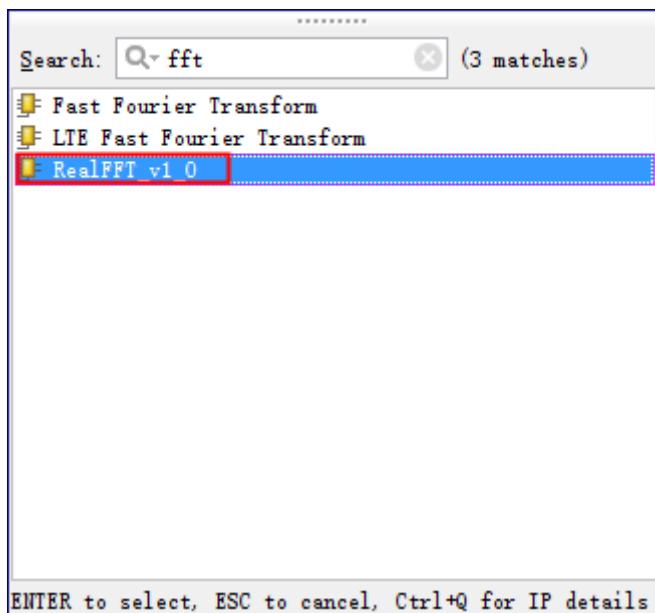


Step16: 添加一个 zed_audio_ctrl IP, 然后单击 Run connection Automation。

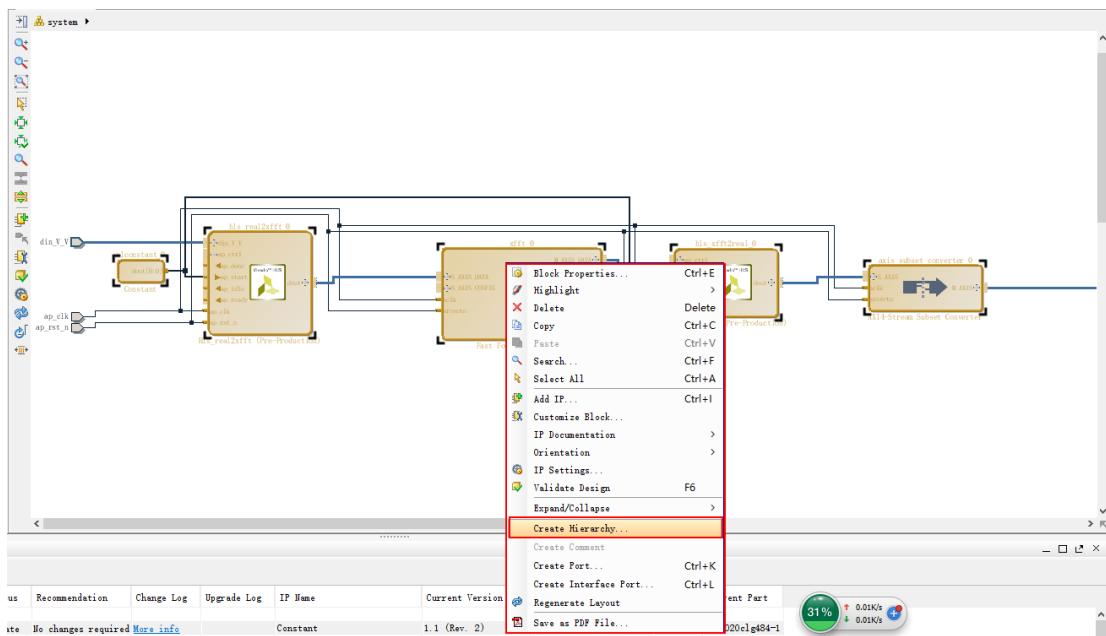


Step17: 依次为 SDATA_I, BCLK, LRCLK, SDATA_O, OLED[5:0], IIC1 引出端口，方法是选中这些信号，然后按 Ctrl+T。

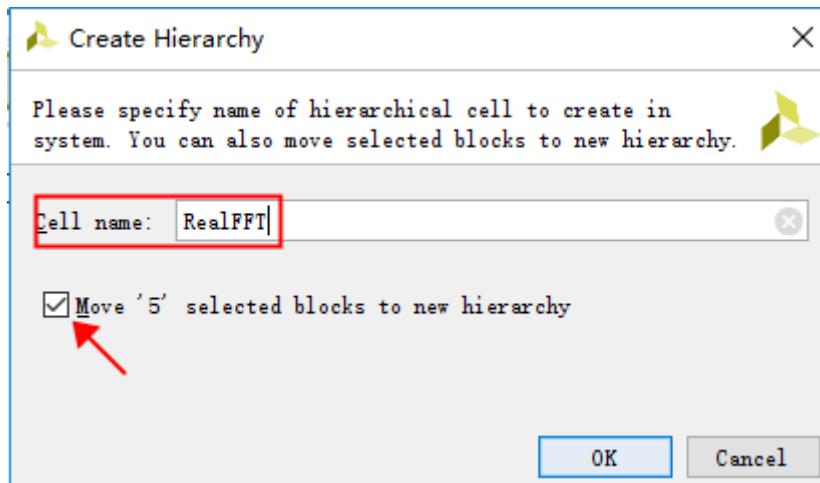
Step18: 将第八章中封装好的 IP 添加到 BD 文件当中。



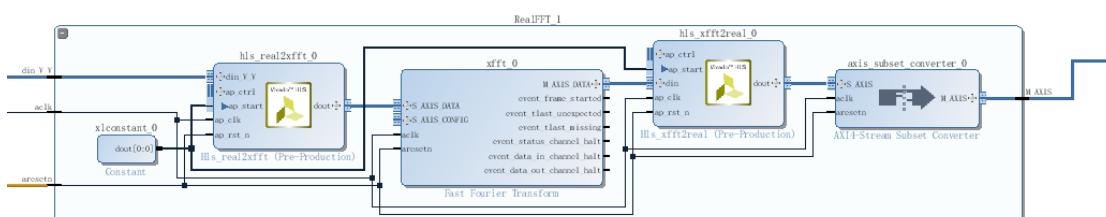
此处另外介绍一种方便多 IP 管理的方法：先按照第八章封装 IP 的硬件电路把要用到的 IP 添加到 BD 文件当中，所有配置与连线都与之前一样，完成之后选中这些 IP，然后右单击，选择 Create Hierarchy，如下图所示：



在跳出来的窗口中如下图设置，最后单击 OK。

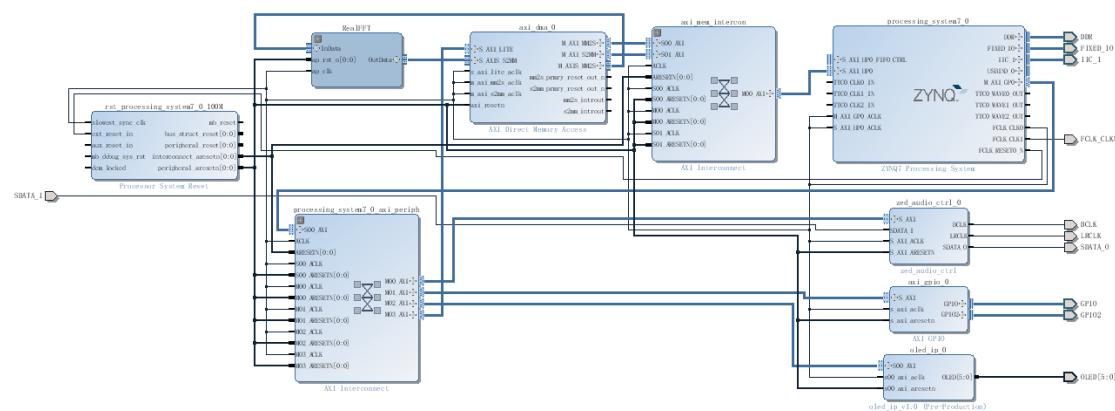
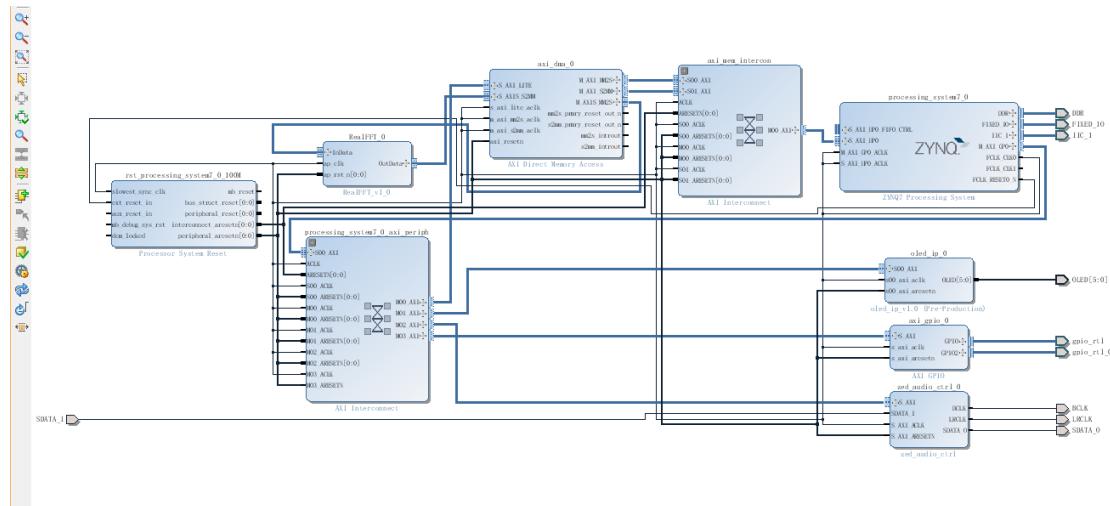


之后系统就会将这几个 IP 整合到一起，生成了一个新的 GUI，在这个 GUI 上有一个加号图标，将鼠标放到这个加号图标上，鼠标的图形会变成斜的展开的式样，这时单击即可查看各个 IP，这样就方便了 IP 的管理，如下图所示：

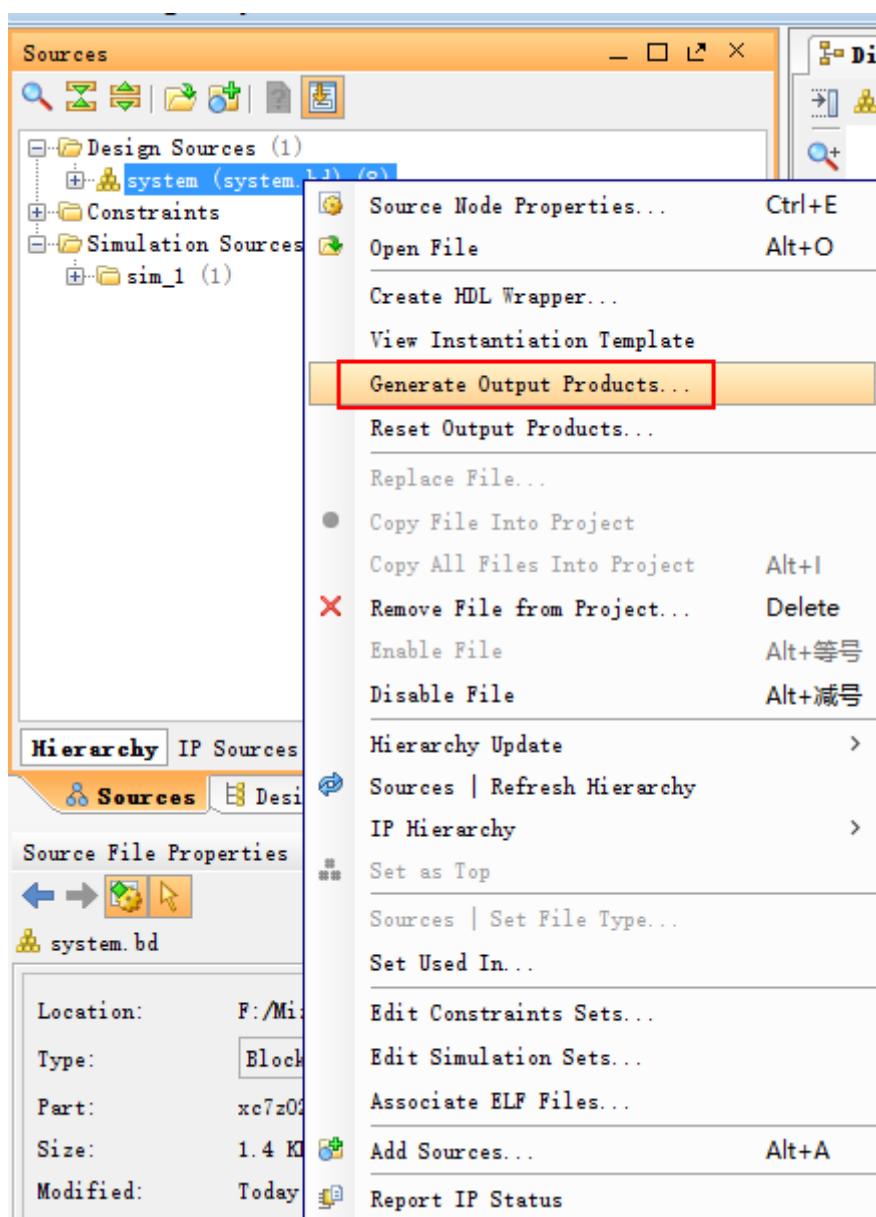


Step19：将以下线路连接起来： InData->M_AXIS_MM2S ， OutData->S_AXIS_S2MM ，

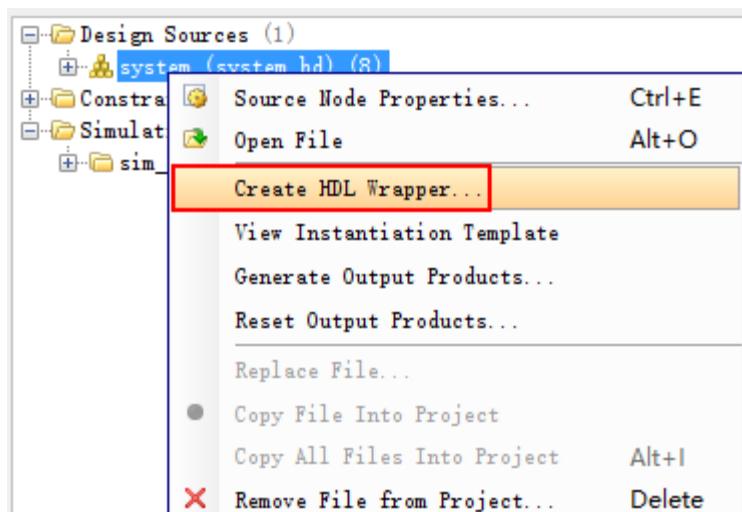
ap_clk→s_axi_lite_aclk, ap_rst_n→axi_resetn (Create Hierarchy之后要将其引出的端口删除再进行连接)。完成后的整体电路如下图所示:



Step20: 选中 BD 文件，右单击选择 Generator Output Products. . . 。



Step21: 右单击 BD 文件，选择 Create HDL Wrapper。



Step22: 把我们提供的约束文件添加到工程当中来。

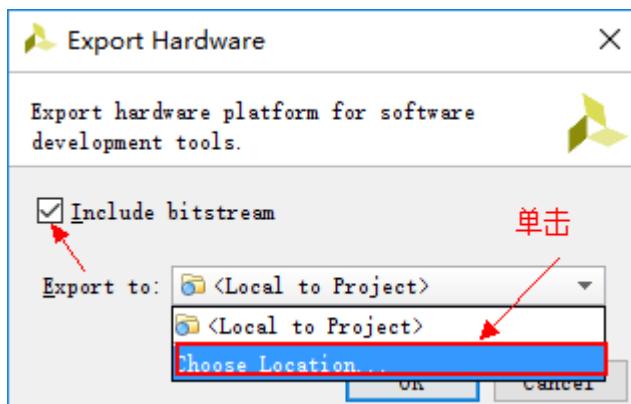
Step23: 单击 生成 bit 文件。

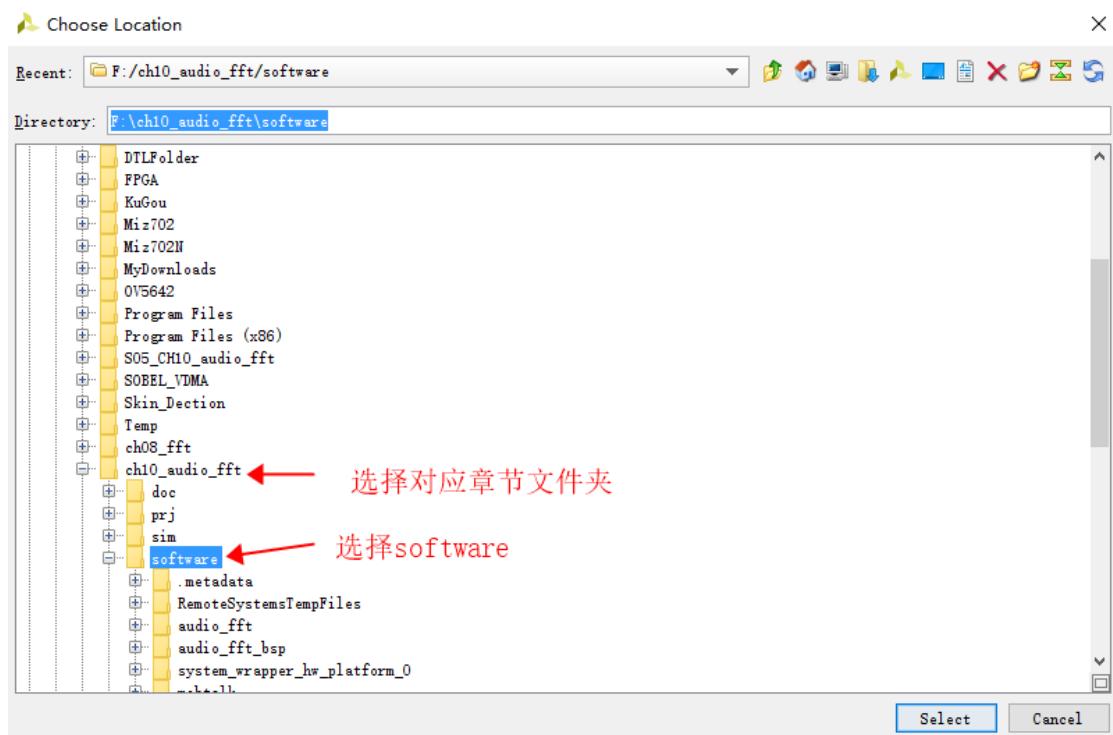
10.3 导入到 SDK

生成 Bit 文件之后，接下来就是 SDK 驱动的编写。在我们提供的源码中，已经提供了供 SDK 使用的工程，为了节省时间，可以直接使用我们提供的工程。

Step1: 单击 File-Export-Export Hardware..。

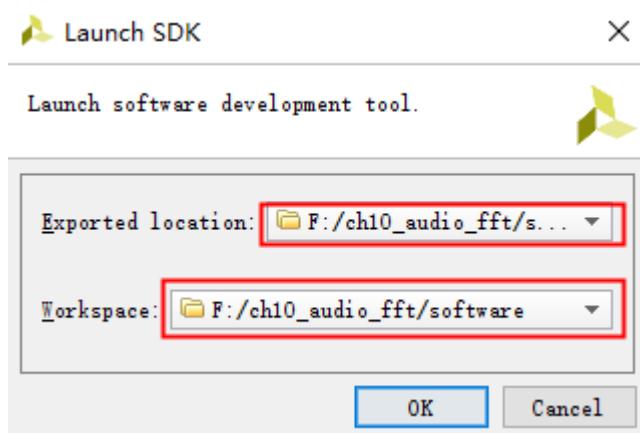
Step2: 选择导出的路径，这里将其指向我们提供的 SDK 驱动工程（在我们提供的源代码对应章节文件夹下的 software 文件夹），如下图所示。



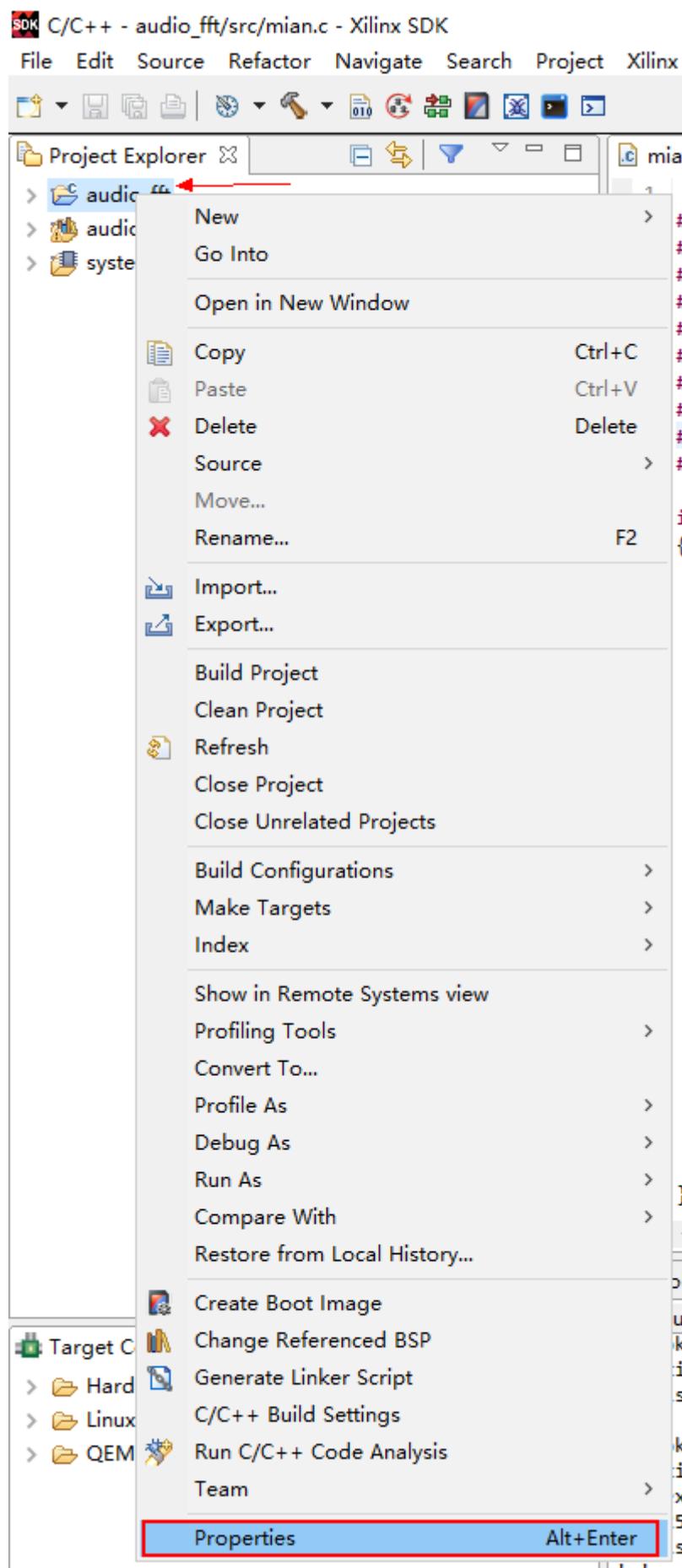


Step3: 单击 File-Launch SDK。

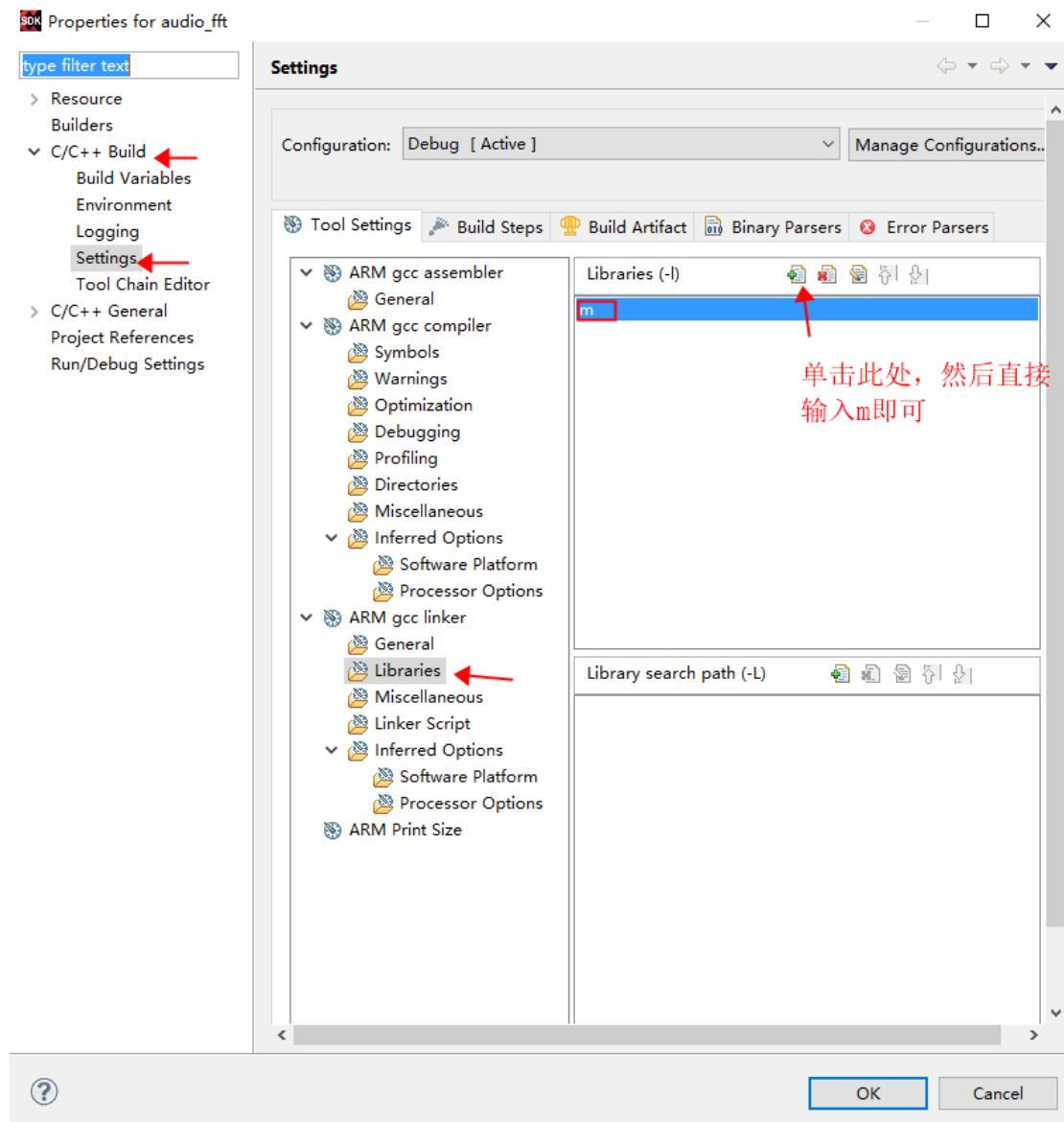
Step4: 在弹出的窗口中, 将下图方框中的路径都选择为 software 文件夹。



Step5: 右单击 audio fft, 选择 Properties。

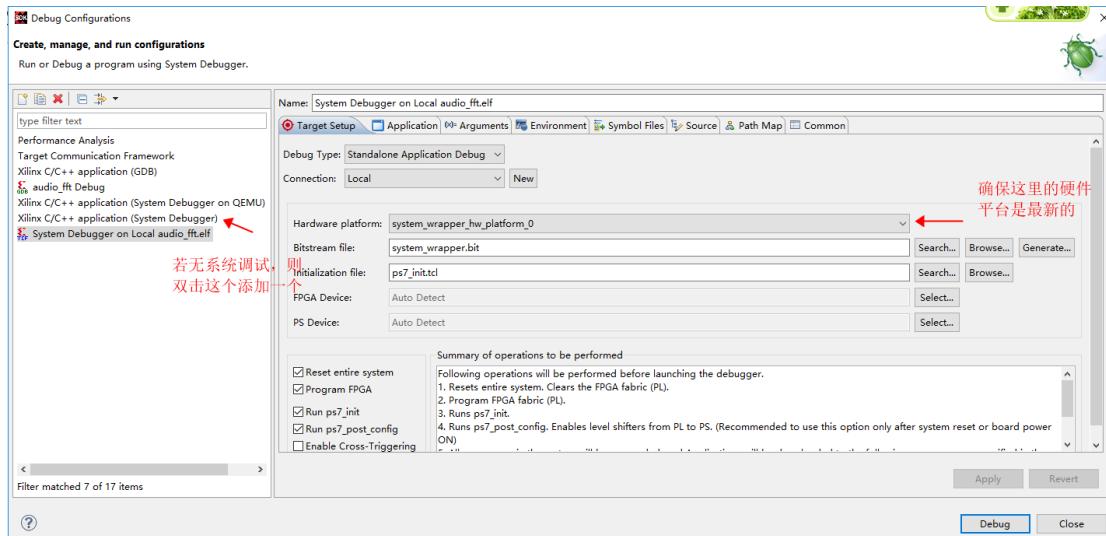


Step5：在下图所示界面将数学函数库关联到 SDK 当中来。

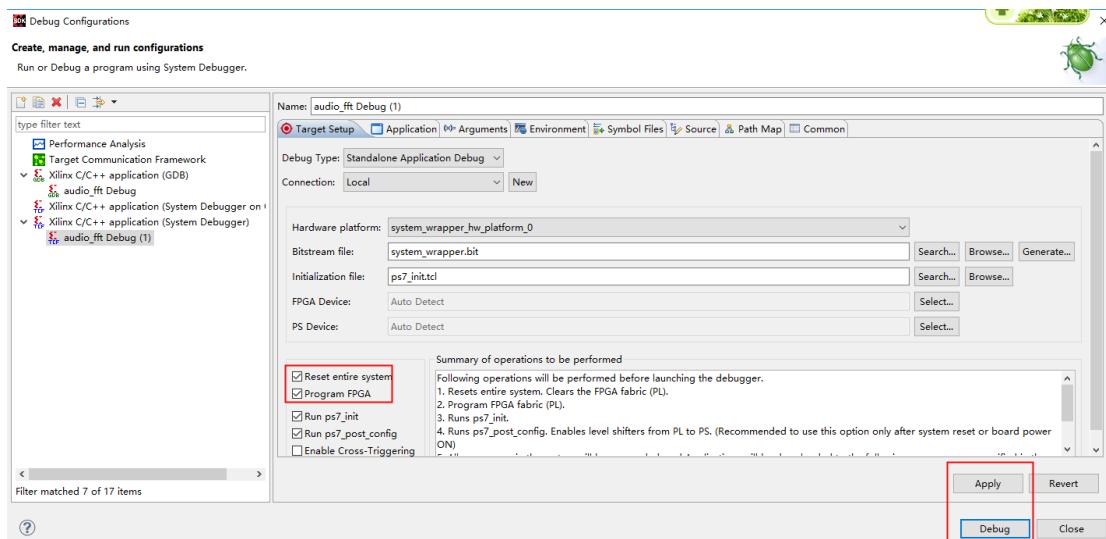


Step6：右击工程，选择 Debug as ->Debug configuration。

Step7：选中 system Debugger,双击创建一个系统调试。



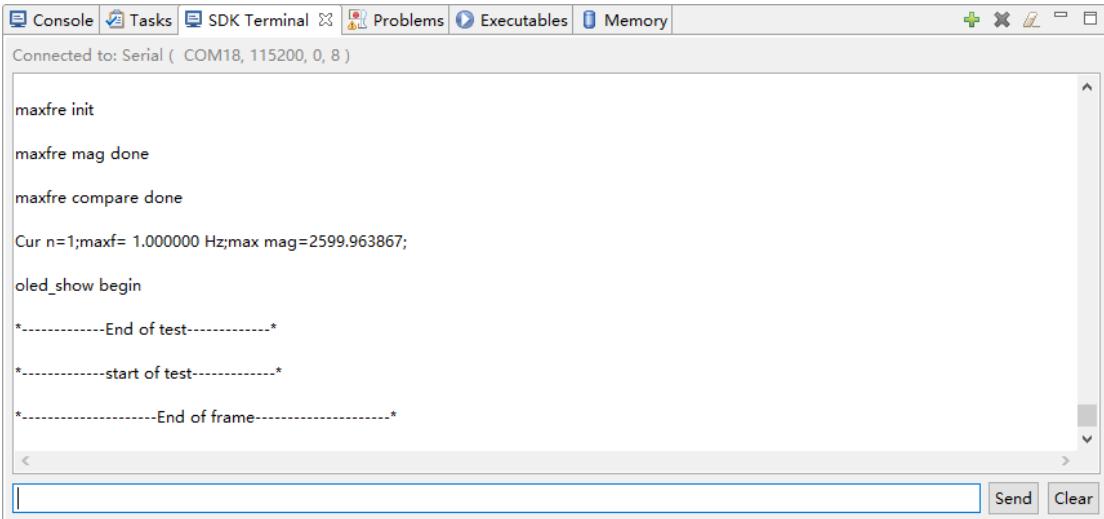
Step8：设置系统调试。



打开系统自带的窗口调试助手，点击运行按钮开始运行程序。



系统运行结果如下图所示：



```

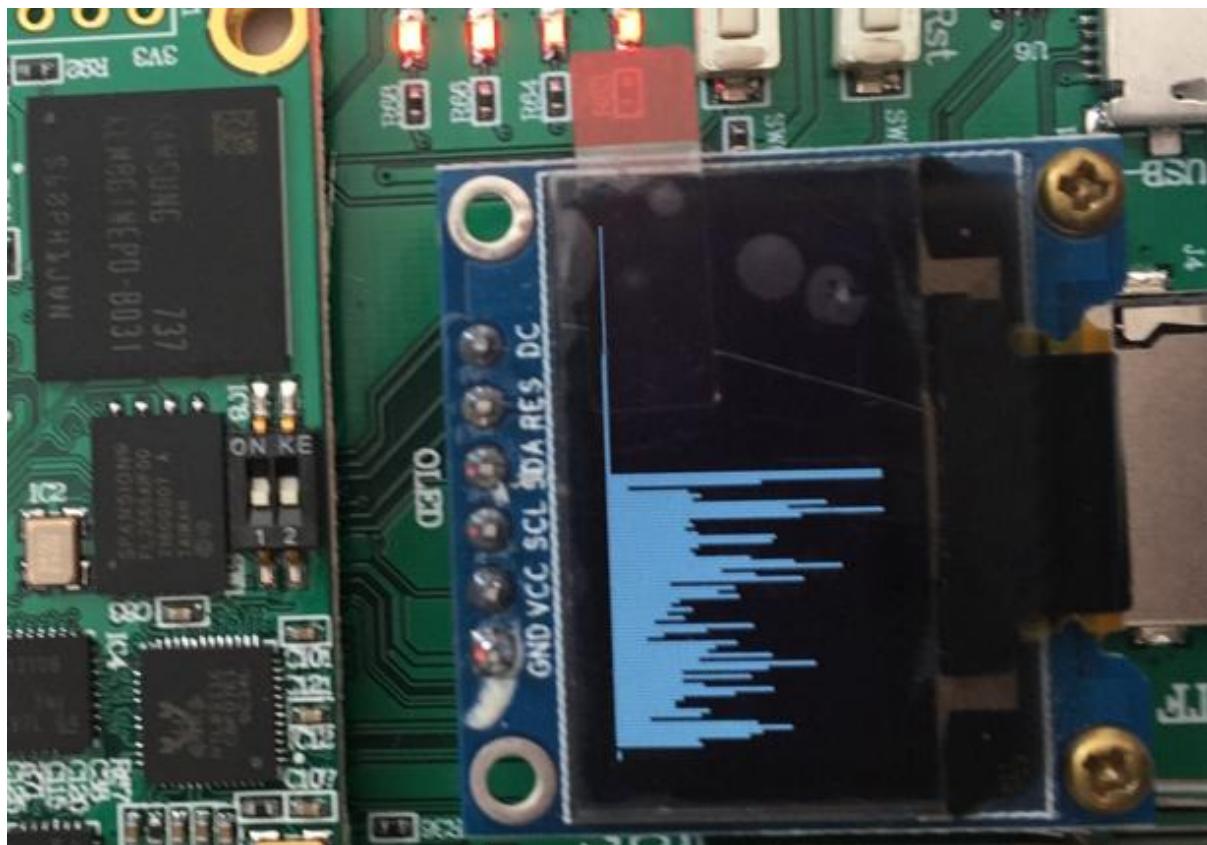
Console Tasks SDK Terminal Problems Executables Memory

Connected to: Serial ( COM18, 115200, 0, 8 )

maxfre init
maxfre mag done
maxfre compare done
Cur n=1;maxf= 1.000000 Hz;max mag=2599.963867;
oled_show begin
*-----End of test-----
*-----start of test-----
*-----End of frame-----

```

在音频输入接入音频子卡的 J4 口，可看到 oled 实时显示音频的频谱。



10.4 程序分析

本章的程序主要可以分为四大部分：DMA 驱动、OLED 驱动、FFT 配置、ADAU1761 驱动。其中前两个都在我们之前的教程中有过详细的讲解，这里着重讲解后两部分。先看到 ADAU1761 的驱动，ADAU1761.c 就是 ADAU1761 的驱动。AUAD1761.c 中有五个函数，其中 IicConfig() 与 AudioWriteToReg() 比较简单，其中的大部分函数在之前都有过讲解，在用户自己设计相关项目时，可以直接拿来稍微修改就可以使用。本节重点讲解另外三个函数，先看到下面这个函数：

```

void AudioPllConfig() {
    unsigned char u8TxData[8], u8RxData[6];
    int Status;

    Status = IicConfig(XPAR_XIICPS_0_DEVICE_ID); //, FMC_IIC_ID, HDMI_IIC_ID);
    if(Status != XST_SUCCESS) {
        xil_printf("\nError initializing IIC");
        //return XST_FAILURE;
    }

    AudioWriteToReg(R0_CLOCK_CONTROL, 0x0E);

    // Write 6 bytes to R1
    u8TxData[0] = 0x40;
    u8TxData[1] = 0x02;
    u8TxData[2] = 0x02; // byte 1
    u8TxData[3] = 0x71; // byte 2
    u8TxData[4] = 0x02; // byte 3
    u8TxData[5] = 0x3C; // byte 4
    u8TxData[6] = 0x21; // byte 5
    u8TxData[7] = 0x01; // byte 6

    XIicPs_MasterSendPolled(&Iic, u8TxData, 8, (IIC_SLAVE_ADDR >> 1));
    while(XIicPs_BusIsBusy(&Iic));

    // Poll PLL Lock bit
    u8TxData[0] = 0x40;
    u8TxData[1] = 0x02;

    do {
        XIicPs_MasterSendPolled(&Iic, u8TxData, 2, (IIC_SLAVE_ADDR >> 1));
        while(XIicPs_BusIsBusy(&Iic));
        XIicPs_MasterRecvPolled(&Iic, u8RxData, 6, (IIC_SLAVE_ADDR >> 1));
        while(XIicPs_BusIsBusy(&Iic));
    }
    while((u8RxData[5] & 0x02) == 0);

    AudioWriteToReg(R0_CLOCK_CONTROL, 0x0F); //COREN
}

```

从上图的函数命令和一些注释我们可以得知这是一个 pll 的配置，一开始是之前老套路的初始化程序，初始化之后，我们注意到其向 R0 写入了一个值 0E，我们可以通过前面的讲解来看看这是什么意思。

Table 12. Clock Control Register (Register R0, Address 0x4000)

Bits	Bit Name	Settings
3	CLKSRC	0: Direct from MCLK pin (default) 1: PLL clock
[2:1]	INFREQ[1:0]	00: 256 × f _s (default) 01: 512 × f _s 10: 768 × f _s E=1110 11: 1024 × f _s
0	COREN	0: Core clock disabled (default) 1: Core clock enabled

因此这里的意思就很明显了禁止 core clock。接下来看到这一段程序。

```

// Write 6 bytes to R1
u8TxData[0] = 0x40;
u8TxData[1] = 0x02;
u8TxData[2] = 0x00; // byte 1 //625
u8TxData[3] = 0x7D; // byte 2
u8TxData[4] = 0x00; // byte 3 //572
u8TxData[5] = 0x12; // byte 4
u8TxData[6] = 0x31; // byte 5
u8TxData[7] = 0x01; // byte 6

XIicPs_MasterSendPolled(&Iic, u8TxData, 8, (IIC_SLAVE_ADDR >> 1));
while(XIicPs_BusIsBusy(&Iic));

```

从上图的注释可以知道这部分就是一个向 R1 寄存器的写入操作，此时两个时钟控制寄存器都已配置好，根据 10.1.2 的时钟树原理图可以求出 core clock：

$$\text{Core clock} = \text{CLKSRC} * \text{INFREQ}[1:0]$$

其中 $\text{CLKSRC} = \text{MCLK} * (\text{R} + \text{N}/\text{M})/\text{X}$, 对照 R1 的寄存器定义, 我们得知 M 是写入的 byte1 和 byte2(也就是 (0x007D=125)), N 是写入的 byte3 和 byte4 (0x0012=18), R 和 X 分别是 byte5 的 Bit[14:11] 和 Bit[10:9]。此时算得 $\text{CLKSRC} = 8 * (6 + 18/125) = 49.152$ 。

求得了 CLKSRC 之后，就可以求出采样率了。具体怎么求，数据中已经给出了方法，如下图所示：

CORE CLOCK

Clocks for the converters, the serial ports, and the DSP are derived from the core clock. The core clock can be derived directly from MCLK or it can be generated by the PLL. The CLKSRC bit (Bit 3 in Register R0, Address 0x4000) determines the clock source.

The INFREQ[1:0] bits should be set according to the expected input clock rate selected by CLKSRC; this value also determines the core clock rate and the base sampling frequency, f_s .

For example, if the input to CLKSRC = 49.152 MHz (from PLL), then

$$\text{INFREQ}[1:0] = 1024 \times f_s$$

$$f_s = 49.152 \text{ MHz} / 1024 = 48 \text{ kHz}$$

The PLL output clock rate is always $1024 \times f_s$, and the clock control register automatically sets the INFREQ[1:0] bits to $1024 \times f_s$ when using the PLL. When using a direct clock, the INFREQ[1:0] frequency should be set according to the MCLK pin clock rate and the desired base sampling frequency.

在本节程序中，INFREQ=11 (E=1110)，CLKSRC 为 49.152，刚好和图中数据一致，因此和上图中的采样率 f_s 一致，都为 48KHZ。

接下来的两个函数是对 ADAU1761 的输入和输出进行配置，如下图所示：

To utilize the maximum amount of DSP instructions, the core clock should run at a rate of $1024 \times f_s$.

Table 12. Clock Control Register (Register R0, Address 0x4000)

Bits	Bit Name	Settings
3	CLKSRC	0: Direct from MCLK pin (default) 1: PLL clock
[2:1]	INFREQ[1:0]	00: $256 \times f_s$ (default) 01: $512 \times f_s$ 10: $768 \times f_s$ 11: $1024 \times f_s$
0	COREN	0: Core clock disabled (default) 1: Core clock enabled

```

1125 void AudioConfigureJacks()
1126 {
1127     AudioWriteToReg(R4_RECORD_MIXER_LEFT_CONTROL_0, 0x01); //enable mixer 1
1128     AudioWriteToReg(R5_RECORD_MIXER_LEFT_CONTROL_1, 0x07); //unmute Left channel of line in into mxr 1 and set gain to 6 dB
1129     AudioWriteToReg(R6_RECORD_MIXER_RIGHT_CONTROL_0, 0x01); //enable mixer 2
1130     AudioWriteToReg(R7_RECORD_MIXER_RIGHT_CONTROL_1, 0x07); //unmute Right channel of line in into mxr 2 and set gain to 6 dB
1131     AudioWriteToReg(R19_ADC_CONTROL, 0x13); //enable ADCs
1132
1133     AudioWriteToReg(R22_PLAYBACK_MIXER_LEFT_CONTROL_0, 0x21); //unmute Left DAC into Mxr 3; enable mxr 3
1134     AudioWriteToReg(R24_PLAYBACK_MIXER_RIGHT_CONTROL_0, 0x41); //unmute Right DAC into Mxr4; enable mxr 4
1135     AudioWriteToReg(R26_PLAYBACK_LR_MIXER_LEFT_LINE_OUTPUT_CONTROL, 0x05); //unmute Mxr3 into Mxr5 and set gain to 6dB; enable mxr 5
1136     AudioWriteToReg(R27_PLAYBACK_LR_MIXER_RIGHT_LINE_OUTPUT_CONTROL, 0x11); //unmute Mxr4 into Mxr6 and set gain to 6dB; enable mxr 6
1137     AudioWriteToReg(R29_PLAYBACK_HEADPHONE_LEFT_VOLUME_CONTROL, 0x00); //Mute Left channel of HP port (LHP)
1138     AudioWriteToReg(R30_PLAYBACK_HEADPHONE_RIGHT_VOLUME_CONTROL, 0x00); //Mute Right channel of HP port (RHP)
1139     AudioWriteToReg(R31_PLAYBACK_LINE_OUTPUT_LEFT_VOLUME_CONTROL, 0xE6); //set LOUT volume (0dB); unmute left channel of Line out port; set Line out port to line in
1140     AudioWriteToReg(R32_PLAYBACK_LINE_OUTPUT_RIGHT_VOLUME_CONTROL, 0xE6); //set ROUT volume (0dB); unmute right channel of Line out port; set Line out port to line in
1141     AudioWriteToReg(R35_PLAYBACK_POWER_MANAGEMENT, 0x03); //enable left and right channel playback (not sure exactly what this does...)
1142     AudioWriteToReg(R36_DAC_CONTROL_0, 0x03); //enable both DACs
1143
1144     AudioWriteToReg(R58_SERIAL_INPUT_ROUTE_CONTROL, 0x01); //Connect I2S serial port output (SDATA_O) to DACs
1145     AudioWriteToReg(R59_SERIAL_OUTPUT_ROUTE_CONTROL, 0x01); //connect I2S serial port input (SDATA_I) to ADCs
1146
1147     AudioWriteToReg(R17_CONVERTER_CONTROL_0, 0x06); //96 kHz
1148     AudioWriteToReg(R64_SERIAL_PORT_SAMPLING_RATE, 0x06); //96 kHz
1149     AudioWriteToReg(R19_ADC_CONTROL, 0x13);
1150     AudioWriteToReg(R36_DAC_CONTROL_0, 0x03);
1151     AudioWriteToReg(R35_PLAYBACK_POWER_MANAGEMENT, 0x03);
1152     AudioWriteToReg(R58_SERIAL_INPUT_ROUTE_CONTROL, 0x01);
1153     AudioWriteToReg(R59_SERIAL_OUTPUT_ROUTE_CONTROL, 0x01);
1154     AudioWriteToReg(R65_CLOCK_ENABLE_0, 0x7F); //Enable clocks
1155     AudioWriteToReg(R66_CLOCK_ENABLE_1, 0x03); //Enable rest of clocks
1156 }
1157
1158 void LineinLineoutConfig() {
1159
1160     AudioWriteToReg(R4_RECORD_MIXER_LEFT_CONTROL_0, 0x01);
1161     AudioWriteToReg(R5_RECORD_MIXER_LEFT_CONTROL_1, 0x05); //0 dB gain
1162     AudioWriteToReg(R6_RECORD_MIXER_RIGHT_CONTROL_0, 0x01);
1163     AudioWriteToReg(R7_RECORD_MIXER_RIGHT_CONTROL_1, 0x05); //0 dB gain
1164
1165     AudioWriteToReg(R22_PLAYBACK_MIXER_LEFT_CONTROL_0, 0x21);
1166     AudioWriteToReg(R24_PLAYBACK_MIXER_RIGHT_CONTROL_0, 0x41);
1167     AudioWriteToReg(R26_PLAYBACK_LR_MIXER_LEFT_LINE_OUTPUT_CONTROL, 0x03); //0 dB
1168     AudioWriteToReg(R27_PLAYBACK_LR_MIXER_RIGHT_LINE_OUTPUT_CONTROL, 0x09); //0 dB
1169     AudioWriteToReg(R29_PLAYBACK_HEADPHONE_LEFT_VOLUME_CONTROL, 0xE7); //0 dB
1170     AudioWriteToReg(R30_PLAYBACK_HEADPHONE_RIGHT_VOLUME_CONTROL, 0xE7); //0 dB
1171     AudioWriteToReg(R31_PLAYBACK_LINE_OUTPUT_LEFT_VOLUME_CONTROL, 0xE6); //0 dB
1172     AudioWriteToReg(R32_PLAYBACK_LINE_OUTPUT_RIGHT_VOLUME_CONTROL, 0xE6); //0 dB
1173 }

```

程序的意思已经在程序中进行了必要的注释，具体的寄存器意义还请读者自行查阅 ADAU1761 的官方数据手册。

接下来看到 FFT 的配置部分，FFT 的配置位于 fft_audio.c 当中。首先看到这一段函数：

```
5⑤void audio_fre()
6 {
7     int i,j;
8     short dataIn[FFT_SIZE*4];
9     compx out[FFT_SIZE/2];
10    float mag[FFT_SIZE/2];
11
12 //===== DMA =====
13 XAxiDma axiDma;
14 int status;
15 status = init_dma(&axiDma);
16 if (status != XST_SUCCESS)
17 {
18     exit(-1);
19 }
20
21 printf("*-----start of test-----*\n\r");
22
23 //-----audio value-----//
24 get_audioData(dataIn);
25 for (i = 1; i < 4; i++)
26 {
27     for(j=0;j<FFT_SIZE;j++)
28     {
29         dataIn[j + i * FFT_SIZE]=dataIn[j];
30     }
31 }
32
33 //-----fft-----//
34 RealFFT_DMA(&axiDma, dataIn, out);
35 maxfre(out,mag);
36 oled_show(mag);
37
38 //-----end-----//
39 printf("*-----End of test-----*\n\r");
40 }
```

这一段函数相当于 FFT 配置函数的 main 函数功能。首先程序进行了一些初始化，之后对 DMA 进行了初始化。之后在看到 get_audioData 函数。其函数定义如下图所示：

```

103 void get_audioData(short* audio_data)
104 {
105     unsigned long u32Temp;
106     Xint32 i;
107     short in_data;
108     i = 0;
109     while (i<FFT_SIZE)           //128
110     {
111         do //wait for RX data to become available
112         {
113             u32Temp = Xil_In32(I2S_STATUS_REG);
114         } while (u32Temp == 0);
115
116         in_data = Xil_In32(I2S_DATA_RX_L_REG);
117         Xil_Out32(I2S_DATA_TX_L_REG, in_data);
118
119         *audio_data= in_data;//((in_data<<8)>>16);
120
121         Xil_Out32(I2S_STATUS_REG, 0x00000001); //Clear data ready bit
122         audio_data++;
123         i++;
124     }
125 }
126 }
```

这一段程序完成的是从 ADAU1761 中获取音频数据，这里需要注意的是获取到的数据要在 FFT_SIZE 这个范围内，这段程序实际上就是一个反复的写寄存器，然后读寄存器中的数据的操作。这里的 I2S_DATA_RX_L_REG 和 I2S_DATA_TX_L_REG 就是 ADAU1761 存放音频数据的寄存器，我们任取一个将鼠标停放在它们上面，然后按 F3 可查看其定义，如下图所示：

```

enum i2s_regs {
    I2S_DATA_RX_L_REG          = 0x00 + AUDIO_BASE,
    I2S_DATA_RX_R_REG          = 0x04 + AUDIO_BASE,
    I2S_DATA_TX_L_REG          = 0x08 + AUDIO_BASE,
    I2S_DATA_TX_R_REG          = 0x0c + AUDIO_BASE,
    I2S_STATUS_REG              = 0x10 + AUDIO_BASE,
};
```

在获取了音频的数据之后，通过一个 for 循环，将其数据量放在了一个四倍于其容量的数组当中，为接下来的处理做准备。然后看到 RealFFT() 函数，其定义如下：

```

201 void RealFFT_DMA(XAxiDma *InstancePtr,short *dataIn,compx * dataOut)
202 {
203     int i=0;
204     int status;
205
206     Xil_DCacheFlushRange((unsigned)dataIn, 4 * FFT_SIZE * sizeof(short)); //Refresh cache
207     status = XAxidma_SimpleTransfer(InstancePtr, (u32)dataIn,4 * FFT_SIZE * sizeof(short), XAXIDMA_DMA_TO_DEVICE); //send enough data to DMA
208     for (i = 0; i < 2; i++)
209     {
210         // Setup DMA from PL to PS memory using AXI DMA's 'simple' transfer mode
211         status = XAxidma_SimpleTransfer(InstancePtr, (u32)dataOut,
212                                         FFT_SIZE / 2 * sizeof(compx), XAXIDMA_DEVICE_TO_DMA); //send result to PS
213         // Poll the AXI DMA core
214         do
215         {
216             status = XAxidma_Busy(InstancePtr, XAXIDMA_DEVICE_TO_DMA); //wait until transfer done
217         } while(status);
218     }
219     // Data cache must be invalidated for 'realspectrum' buffer after DMA
220     Xil_DCacheInvalidateRange((unsigned)dataOut,FFT_SIZE / 2 * sizeof(compx));
221
222     // DMA another frame of data to PL
223     if (!XAxidma_Busy(InstancePtr, XAXIDMA_DMA_TO_DEVICE)) //Continue sending data
224     status = XAxidma_SimpleTransfer(InstancePtr, (u32)dataIn,FFT_SIZE * sizeof(short), XAXIDMA_DMA_TO_DEVICE);
225
226     printf("-----End of frame-----\n\r");
227 }
228
229 fflush(stdout);
230 }
```

程序一开始，通过刷内存函数 Xil_DCacheFlushRange 将数据刷入了内存。紧接着启动了 DMA 传输，将一定量的数据传输给了 DMA。

```
// Setup DMA from PL to PS memory using AXI DMA's 'simple' transfer mode
status = XAxiDma_SimpleTransfer(InstancePtr, (u32)dataOut,
    FFT_SIZE / 2 * sizeof(compx), XAXIDMA_DEVICE_TO_DMA);           //send result to PS
// Poll the AXI DMA core
do
{
    status = XAxiDma_Busy(InstancePtr, XAXIDMA_DEVICE_TO_DMA);        //wait until transfer done
}
while(status);
```

接下来的这一句用了一个 do while 语句，其目的是为了确保 DMA 传输完成，这一段完成了一个向 PS 发送数据的功能，发送的数据也就是 dataOut，也就是 FFT 处理之后的结果。之后再看到这一句：

```
// Data cache must be invalidated for 'realspectrum' buffer after DMA
Xil_DCacheInvalidateRange((unsigned)dataOut,FFT_SIZE / 2 * sizeof(compx));
```

Xil_DcacheInvalidateRange() 函数将一段指定长度的高速数据缓冲器无效，这里即为在整段处理的结果传送完成之后，再释放这一段缓冲区，为下一次数据传输做准备。接下来的这一句函数就是启动传输下一段数据传输，如下图所示：

```
// DMA another frame of data to PL
if (!XAxiDma_Busy(InstancePtr, XAXIDMA_DMA_TO_DEVICE))//Continue sending data
status = XAxiDma_SimpleTransfer(InstancePtr, (u32)dataIn,FFT_SIZE * sizeof(short), XAXIDMA_DMA_TO_DEVICE);
```

这个函数整体下来就是一个数据传输过来，然后再将结果传到 PS，然后准备下一次传输的过程。回到 audio_fre() 函数的分析当中，接下来看到 maxfre 函数，其函数定义如下所示：

```
128 void maxfre(compx *x, float *mag)
129 {
130     printf("maxfre in \r\n");
131     int i, n=1;
132     float tmp, maxf;
133     printf("maxfre init \r\n");
134     for (i=0; i<FFT_SIZE/2; i++)          //get the magnitude
135     {
136         float real =(float) x[i].real;
137         float imag =(float) x[i].imag;
138         mag[i]=sqrtf(real * real + imag * imag);
139     }
140     printf("maxfre mag done \r\n");
141     tmp=mag[1];                         //first is DC component
142     for(i=1; i<FFT_SIZE/2; i++)          //get the max mag & its fre
143     {
144         if(tmp<mag[i])
145         {
146             tmp=mag[i];
147             n=i;
148         }
149     }
150     }
151     printf("maxfre compare done \r\n");
152     maxf=1.0*n*FFT_SIZE/FFT_SIZE;
153     printf("Cur n=%d;maxf= %f Hz;max mag=%f;\r\n",n,maxf,tmp);
154 }
```

这个程序完成的计算 FFT 频谱的频率的功能。这里面的计算是一种特定的计算方法，mag 是 FFT 的幅度值，它的值为 FFT 的实部和虚部的平方和。这里的第一个 for 循环也就是求出输入的所有数据的幅度，然后由第二个 for 循环找出其中最大的幅度值，并由这个幅度值求出其频率。

10.5 本章小结

本章向大家介绍了利用第八章生成的 FFT IP 设计了一个实时音乐频谱显示的例子，FFT 在我们的设计过程中非常常见，尤其是在无线通信领域应用非常广泛，传统的 FFT 实现方式十分的繁琐，而利用 HLS 设计则非常的节省时间而且不是十分复杂，其实用性不言而喻。

S05_CH11_快速角点检测的 HLS 实现

11.1 角点定义

角点检测(Corner Detection)是计算机视觉系统中用来获得图像特征的一种方法，广泛应用于运动检测、图像匹配、视频跟踪、三维建模和目标识别等领域中。也称为特征点检测。

角点通常被定义为两条边的交点，更严格的说，角点的局部邻域应该具有两个不同区域的不同方向的边界。而实际应用中，大多数所谓的角点检测方法检测的是拥有特定特征的图像点，而不仅仅是“角点”。这些特征点在图像中有具体的坐标，并具有某些数学特征，如局部最大或最小灰度、某些梯度特征等。

现有的角点检测[算法](#)并不是都十分的鲁棒。很多方法都要求有大量的训练集和冗余数据来防止或减少错误特征的出现。角点检测方法的一个很重要的评价标准是其对多幅图像中相同或相似特征的检测能力，并且能够应对光照变化、图像旋转等图像变化。

在我们解决问题时，往往希望找到特征点，“特征”顾名思义，指能描述物体本质的东西，还有一种解释就是这个特征微小的变化都会对物体的某一属性产生重大的影响。而角点就是这样的特征。

观察日常生活中的“角落”就会发现，“角落”可以视为所有平面的交汇处，或者说是所有表面的发起处。假设我们要改变一个墙角的位置，那么由它而出发的平面势必都要有很大的变化。所以，这就引出了图像角点的定义，“如果某一点在任意方向的一个微小变动都会引起灰度很大的变化，那么我们就把它称之为角点”

特征检测与匹配是 Computer Vision 应用中重要的一部分，这需要寻找图像之间的特征建立对应关系。点，也就是图像中的特殊位置，是很常用的一类特征，点的局部特征也可以叫做“关键特征点”(keypoint feature)，或“兴趣点”(interest point)，或“角点”(corner)。

关于角点的具体描述可以有几种：

1. 一阶导数(即灰度的梯度)的局部最大所对应的像素点；
2. 两条及两条以上边缘的交点；
3. 图像中梯度值和梯度方向的变化速率都很高的点；
4. 角点处的一阶导数最大，二阶导数为零，指示物体边缘变化不连续的方向。

11.2 角点检测算法

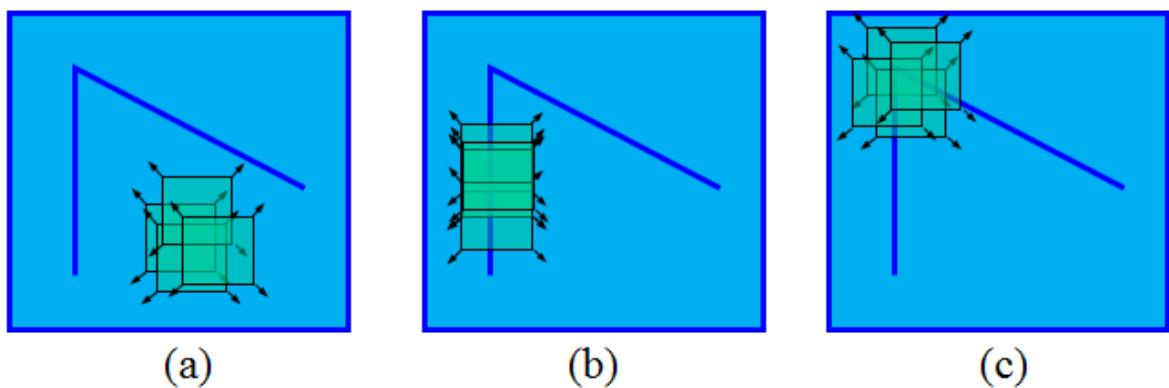
11.2.1 Moravec 角点检测算法

Moravec 角点检测算法是最早的角度检测算法之一。该算法将角点定义为具有低“自相关性”的点。算法会检测图像的每一个像素，将像素周边的一个邻域作为一个 patch，并检测这个 patch 和周围其他 patch 的相关性。这种相关性通过两个 patch 间的平方差之和 (SSD) 来衡量，SSD 值越小则相似性越高。

如果像素位于平滑图像区域内，周围的 patch 都会非常相似。如果像素在边缘上，则周围的 patch 在与边缘正交的方向上会有很大差异，在与边缘平行的方向上则较为相似。而如果像素是各个方向上都有变化的特征点，则周围所有的 patch 都不会很相似。Moravec 会计算每个像素 patch 和周围 patch 的 SSD 最小值作为强度值，取局部强度最大的点作为特征点。

11.2.2 Harris 角点检测

当一个窗口在图像上移动，在平滑区域如图(a)，窗口在各个方向上没有变化。在边缘上如图(b)，窗口在边缘的方向上没有变化。在角点处如图(c)，窗口在各个方向上具有变化。Harris 角点检测正是利用了这个直观的物理现象，通过窗口在各个方向上的变化程度，决定是否为角点。



将图像窗口平移 $[u, v]$ 产生灰度变化 $E(u, v)$

$$E(u, v) = \sum_{x,y} w(x, y) [I(x+u, y+v) - I(x, y)]^2$$

由： $I(x+u, y+v) = I(x, y) + I_x u + I_y v + O(u^2, v^2)$ ， 得到：

$$E(u, v) = [u, v] \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

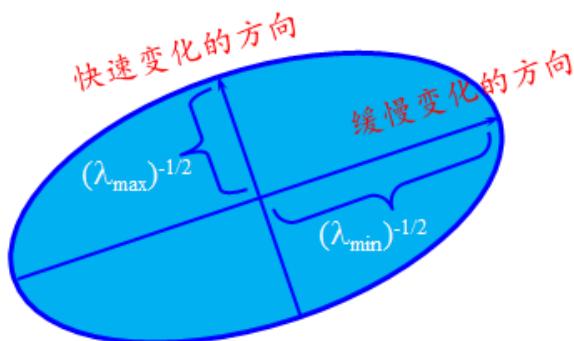
对于局部微小的移动量 $[u, v]$ ，近似表达为：

$$E(u, v) \approx [u, v] M \begin{bmatrix} u \\ v \end{bmatrix}$$

其中 M 是 2×2 矩阵，可由图像的导数求得：

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

$E(u, v)$ 的椭圆形式如下图：



定义角点响应函数 R 为：

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

Harris 角点检测算法就是对角点响应函数 R 进行阈值处理： $R > \text{threshold}$ ，即提取 R 的局部极大值。

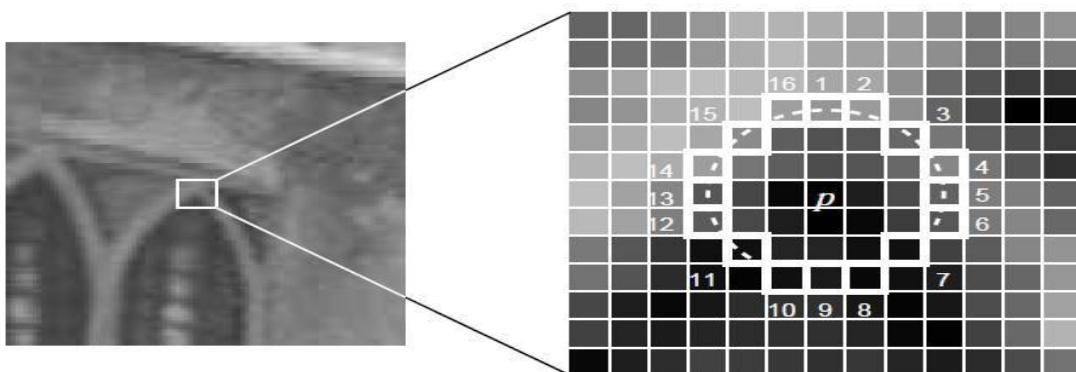
11.2.3 FAST 角点检测算法

Smith 和 Brady 在 1997 年提出了一种完全不同的角点提取方法，即“SUSAN (Smallest Univalued Segment Assimilating Nucleus)” 提取算子。SUSAN 提取算子的基本原理是，与每一图像点相关的局部区域具有相同的亮度。如果某一窗口区域内的每一像素亮度值与该窗口中心的像素亮度值相同或相似，这一窗口区域将被称之为“USAN”。计算图像每一像素的“USAN”，为我们提供了是否有边缘的方法。位于边缘上的像素的“USAN”较小，位于角点上的像素的“USAN”更小。因此，我们仅需寻找最小的“USAN”，就可确定角点。该方法由于不需要计算图像灰度差，因此，具

有很强的抗噪声的能力。

Edward Rosten and Tom Drummond 在 2006 年提出了一种简单快速的角点探测算法，该算法检测的角点定义为在像素点的周围邻域内有足够的像素点与该点处于不同的区域。应用到灰度图像中，即有足够的像素点的灰度值大于该点的灰度值或者小于该点的灰度值。

考虑下图中 p 点附近半径为 3 的圆环上的 16 个点，一个思路是若其中有连续的 12 个点的灰度值与 p 点的灰度值差别超过某一阈值，则可以认为 p 点为角点。



这一思路可以使用[机器学习](#)的方法进行加速。对同一类图像，例如同一场景的图像，可以在 16 个方向上进行训练，得到一棵决策树，从而在判定某一像素点是否为角点时，不再需要对所有方向进行检测，而只需要按照决策树指定的方向进行 2-3 次判定即可确定该点是否为角点。

在本章节的 HLS 实现中我们主要介绍 FAST 角点检测，很多传统的[算法](#)都很耗时，而且特征点检测算法只是很多复杂图像处理里中的第一步，得不偿失。FAST 特征点检测是公认的比较快速的特征点检测方法，只利用周围像素比较的信息就可以得到特征点，简单，有效。

FAST 特征检测算法来源于 corner 的定义，这个定义基于特征点周围的图像灰度值，检测候选特征点周围一圈的像素值，如果候选点周围领域内有足够的像素点与该候选点的灰度值差别够大，则认为该候选点为一个特征点。

$$N = \sum_{x \in \text{circle}(p)} |I(x) - I(p)| > \epsilon_d \quad (1)$$

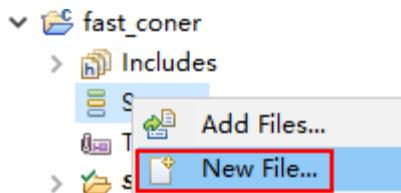
其中 $I(x)$ 为圆周上任意一点的灰度， $I(p)$ 为圆心的灰度， ϵ_d 为灰度值差得阈值，如果 N 大于给定阈值，一般为周围圆圈点的四分之三，则认为 p 是一个特征点。

11.3 HLS 实现

11.3.1 工程创建

Step1：打开 HLS，按照之前介绍的方法，创建一个新的工程，命名为 fast_corner。

Step2：右单击 Source 选项，选择 New File，创建一个名为 Top.cpp 的文件。



Step3：在打开的编辑区中，把下面的程序拷贝进去：

```
#include "top.h"

void hls_fast_corner(AXI_STREAM& INPUT_STREAM, AXI_STREAM&
OUTPUT_STREAM, int rows, int cols, int threshold)
{
    //Create AXI streaming interfaces for the core
#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM

#pragma HLS RESOURCE core=AXI_SLAVE variable=rows
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cols
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=threshold
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=return
metadata="-bus_bundle CONTROL_BUS"

#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols
#pragma HLS INTERFACE ap_stable port=threshold

RGB IMAGE      _src(rows,cols);
RGB IMAGE      _dst(rows,cols);
RGB IMAGE      src0(rows,cols);
RGB IMAGE      src1(rows,cols);
GRAY IMAGE     mask(rows,cols);
```

```

GRAY IMAGE      dmask(rows,cols);
GRAY IMAGE      gray(rows,cols);

#pragma HLS dataflow
#pragma HLS stream depth=20000 variable=src1.data_stream
hls::AXIvideo2Mat(INPUT_STREAM, _src);
hls::Scalar<3,unsigned char> color(255,0,0);
hls::Duplicate(_src,src0,src1);
hls::CvtColor<HLS_BGR2GRAY>(src0,gray);
hls::FASTX(gray,mask,threshold,true);
hls::Dilate(mask,dmask);
hls::PaintMask(src1,dmask,_dst,color);
hls::Mat2AXIvideo(_dst, OUTPUT_STREAM);
}

```

Step4: 再在 Source 中添加一个名为 Top.h 的库函数，并添加如下程序：

```

#ifndef _TOP_H_
#define _TOP_H_

#include "hls_video.h"

// maximum image size
#define MAX_WIDTH 1920
#define MAX_HEIGHT 1080

// I/O Image Settings
#define INPUT_IMAGE          "test_1080p.bmp"
#define OUTPUT_IMAGE          "result.bmp"
#define OUTPUT_IMAGE_GOLDEN   "result_golden.bmp"

// typedef video library core structures
typedef hls::stream<ap_axiu<32,1,1,1> >           AXI_STREAM;
typedef hls::Scalar<3, unsigned char>                 RGB_PIXEL;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3>    RGB_IMAGE;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1>    GRAY_IMAGE;

// top level function for HW synthesis
void hls_fast_corner(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int
rows, int cols, int threshold);

#endif

```

Step5: 在 Test Bench 中，用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码：

```

#include "hls_opencv.h"
#include "top.h"

using namespace cv;

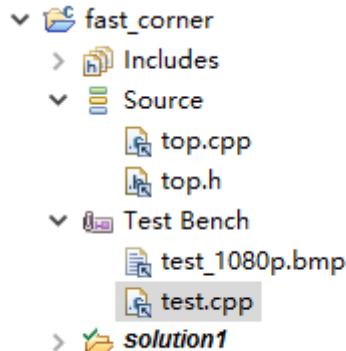
int main (int argc, char** argv) {
    IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels);
    cvShowImage("hls_src", src);

    //HLS视频处理
    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIVideo(src, src_axi);
    hls_fast_corner(src_axi, dst_axi, src->height, src->width, 20);
    AXIVideo2IplImage(dst_axi, dst);
    cvShowImage("hls_dst", dst);
    cvSaveImage(OUTPUT_IMAGE,dst);
    waitKey(0);

    return 0;
}

```

Step6: 在 Test Bench 中添加一张名为 test_1080p.bmp 的测试图片，图片可以在我们提供的源程序中的 Image 文件夹中找到。完整的工程如下图所示：

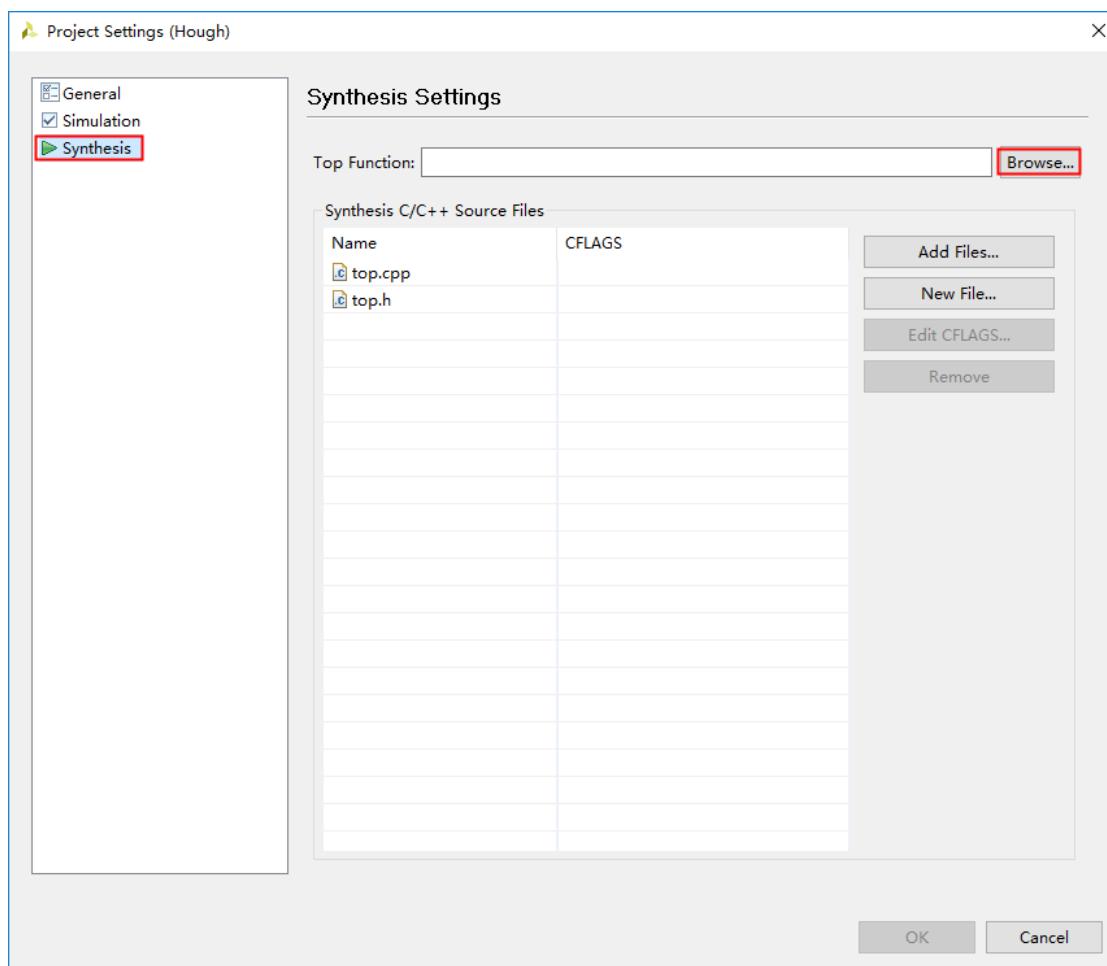


11.3.2 仿真及优化

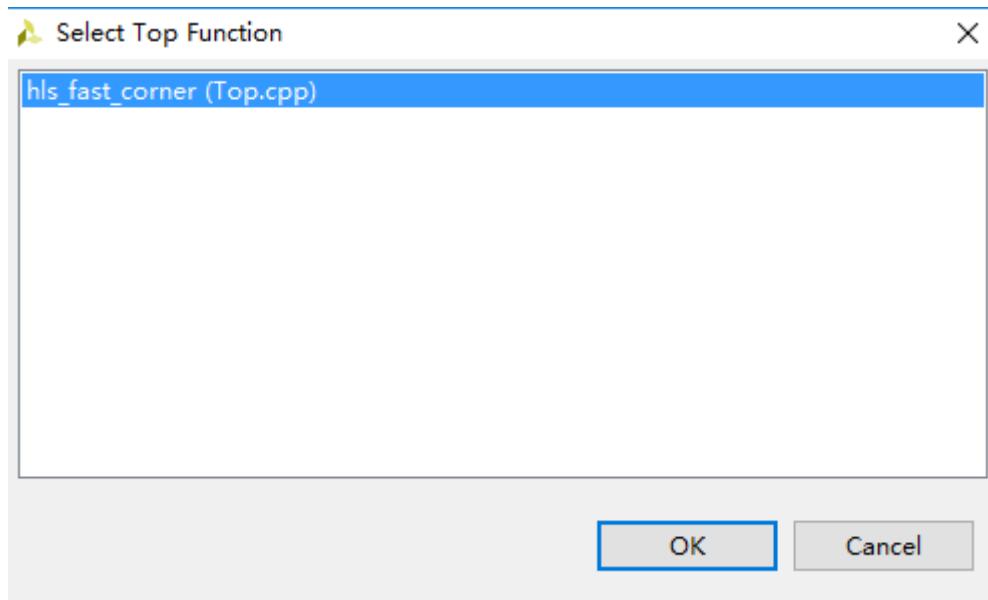
Step1: 单击 Project settings 快捷键。



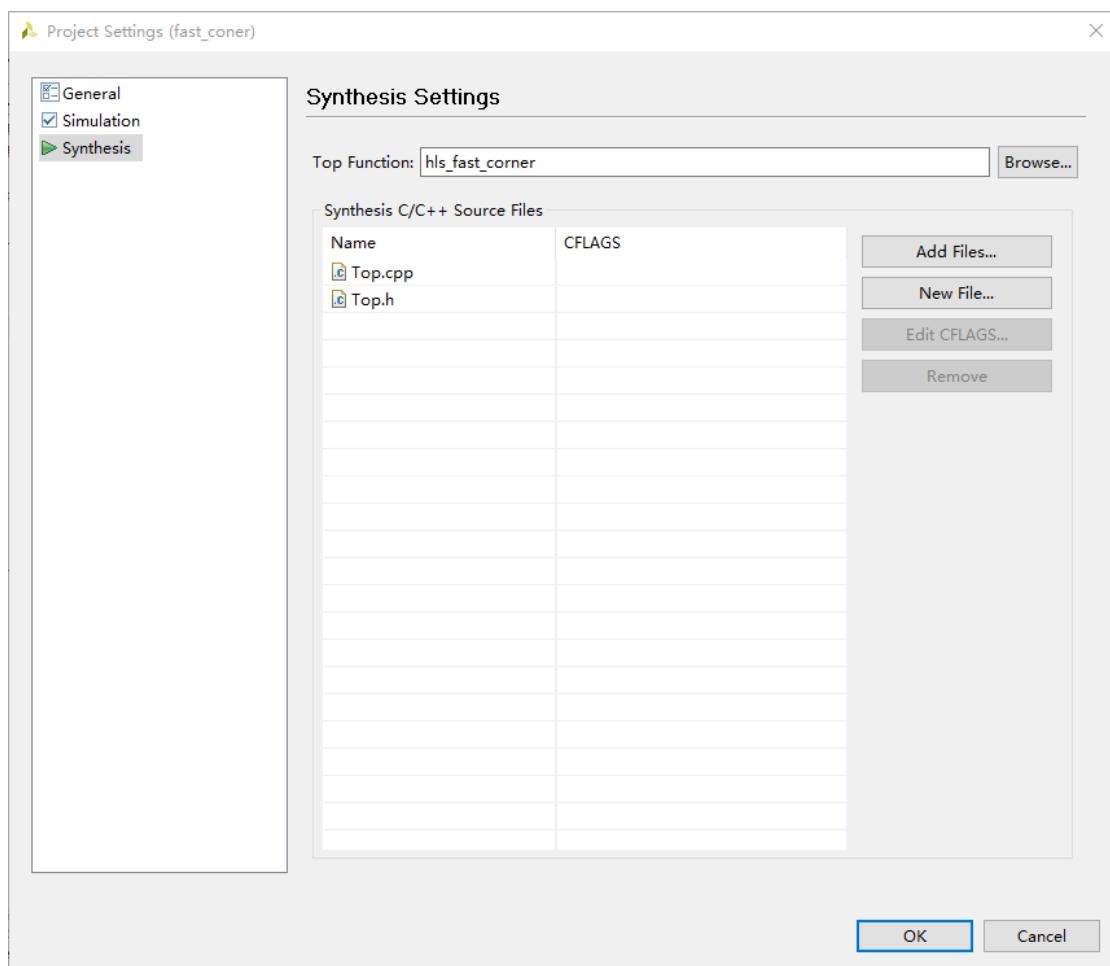
Step2: 选择 Synthesis 选项，然后点击 Browse.. 指定一个顶层函数。



Step3: 选定 hls_fast_corner 为顶层函数，然后点击 O K。



Step4: 再次单击 OK，完成工程的修改。



Step5：单击  开始综合。



综合报告如下：

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	11.12	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
210	2111641	183	2111634	dataflow

Detail

Instance

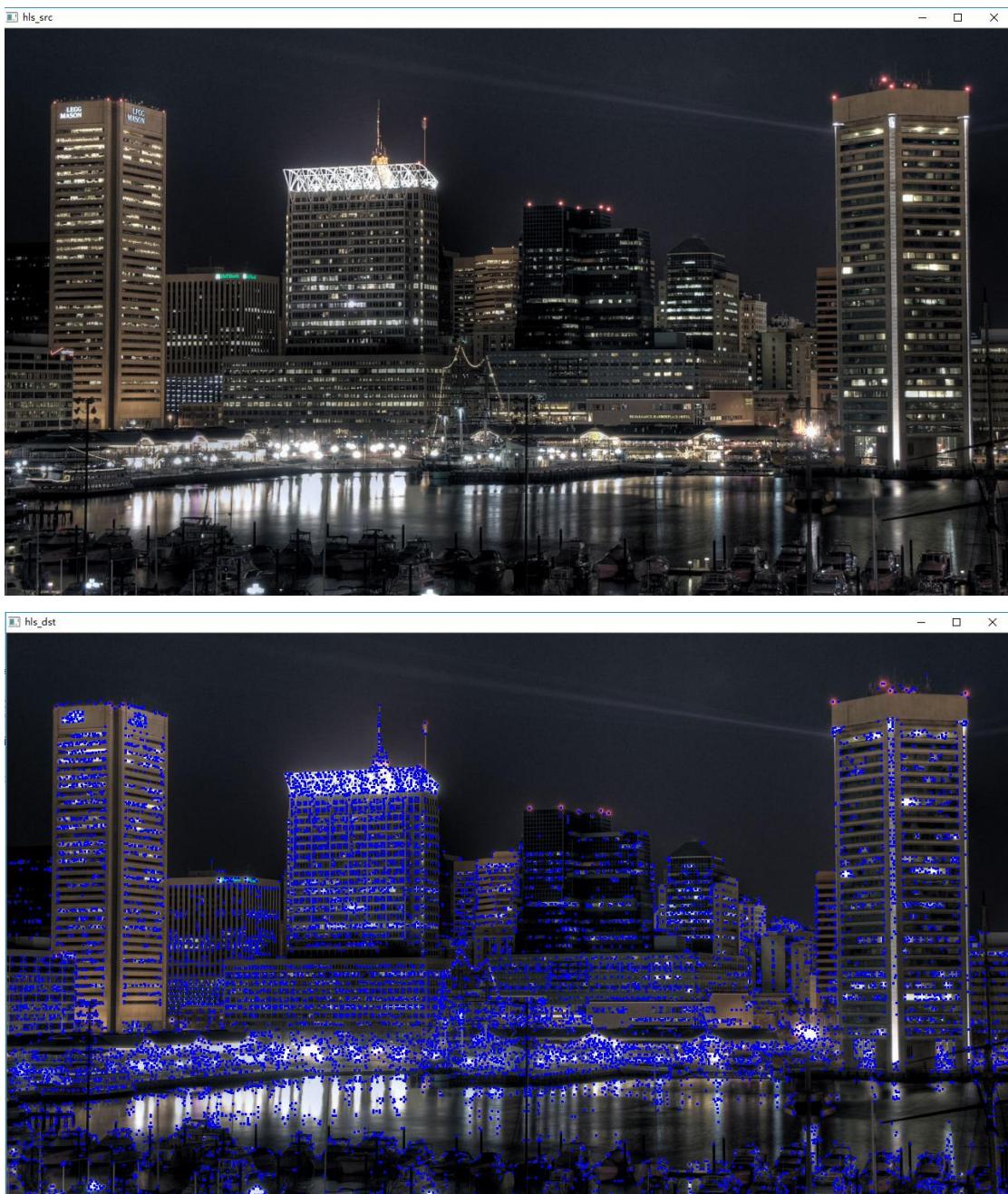
Loop

Utilization Estimates

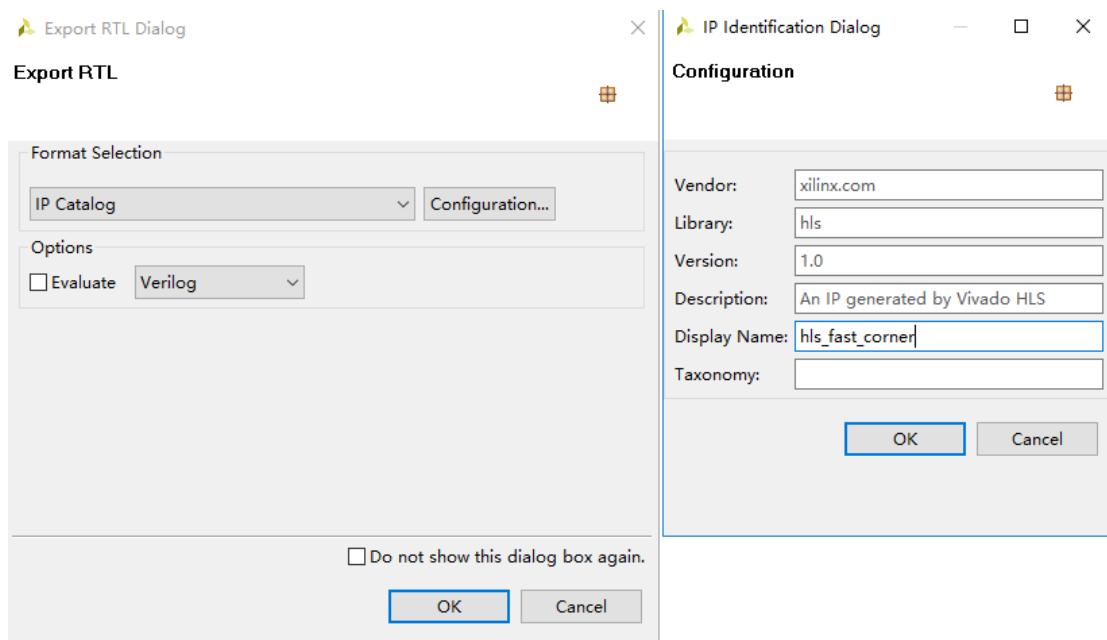
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	48	-	708	2334
Instance	11	3	5208	5867
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	8	-
Total	59	3	5924	8202
Available	120	80	35200	17600
Utilization (%)	49	3	16	46

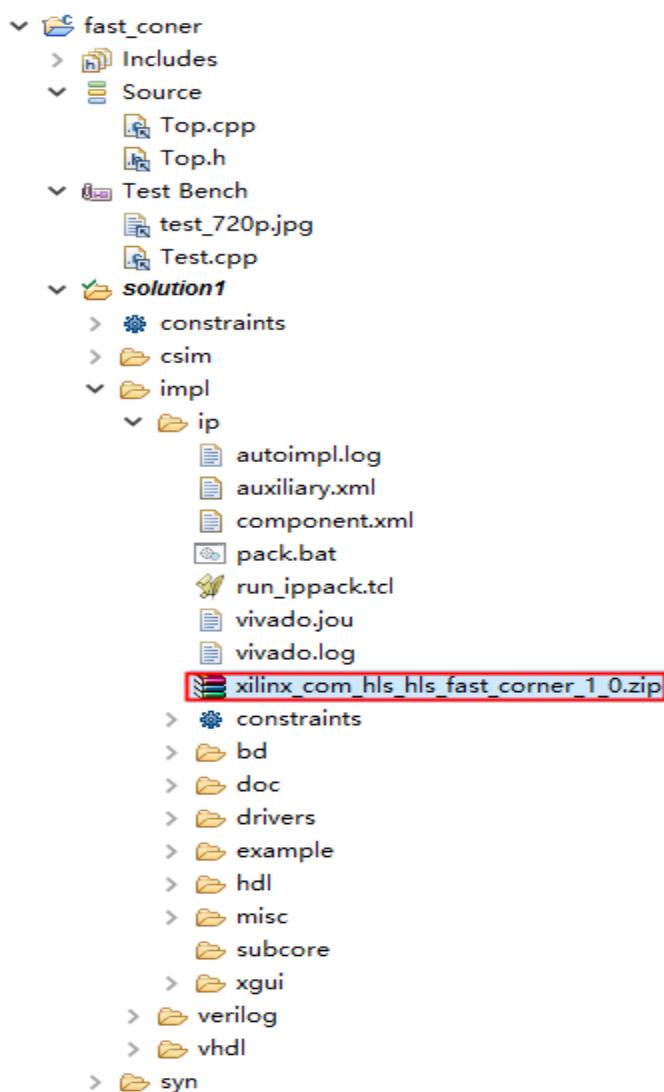
Step6：直接单击  开始进行仿真。系统运行结束后，处理结果如下所示：



Step7:单击 导出供 VIVADO 使用的 IP，然后在弹出来的新窗口中按下图设置。



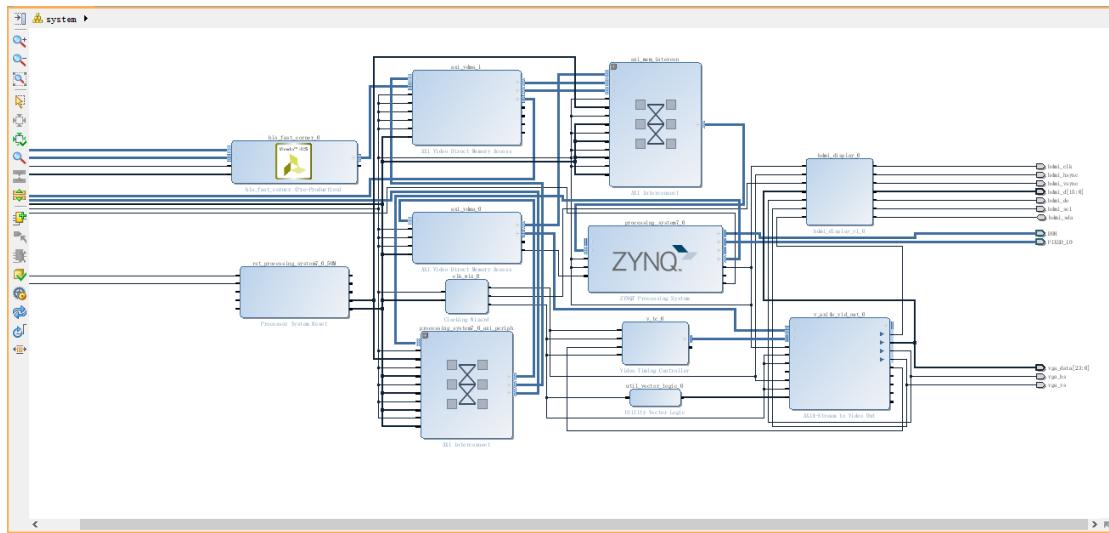
综合完成之后可以看到系统生成的 IP。



11.4 硬件工程创建

11.4.1 硬件平台搭建

本章的硬件平台与 sobel 算子的实现几乎是一样的，只是在这里需要把 sobel 实现的 HLS IP 替换成角点检测的 IP 。系统整体硬件电路如下图所示：

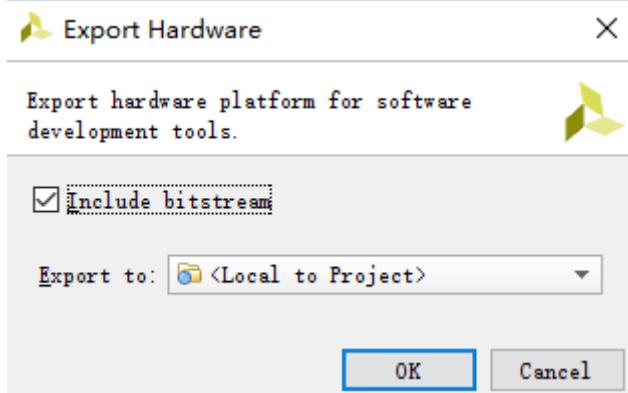


11.4.2 导入到 SDK

硬件平台搭建完成之后，让工程重新生成 Bit 文件，之后记得删除之前的 SDK 工程，以便 HLS IP 的驱动文件能正确的产生。

Step1：在工程文件夹中删除原来工程的 sdk 文件夹。

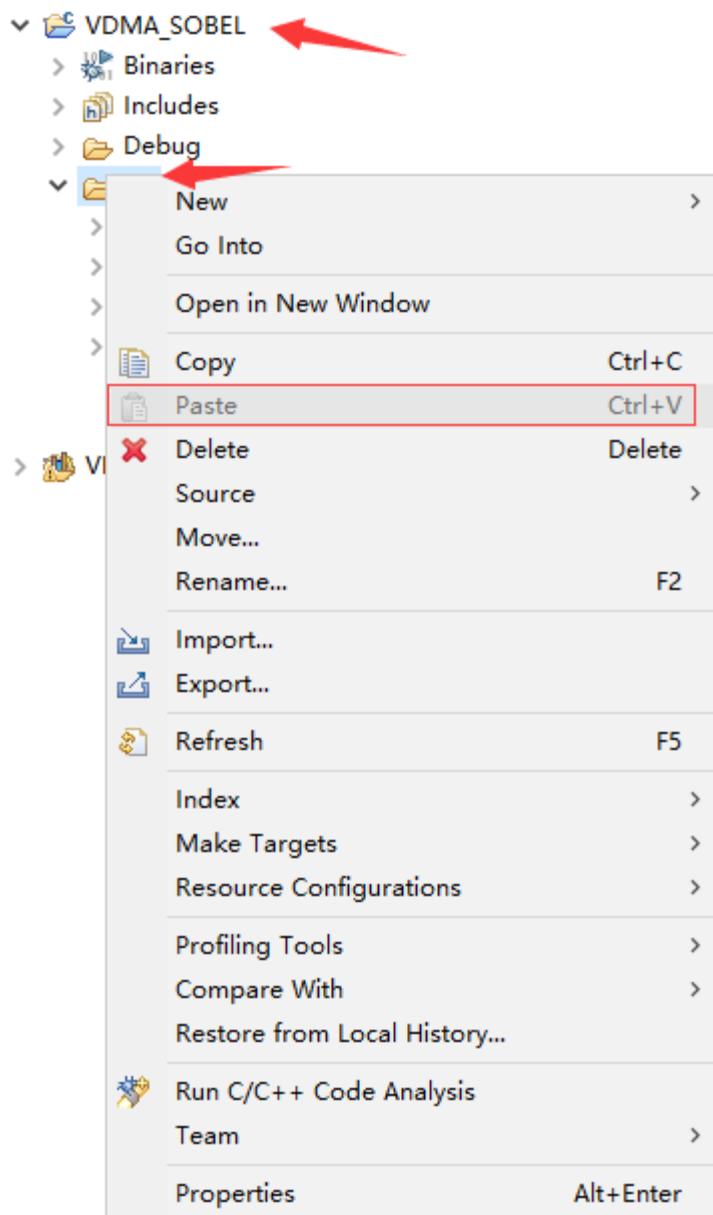
Step2：单击 File-Export-Export Hardware..。



Step3：单击 File-Launch SDK。

Step4：新建一个名为 Fast_Corner 的空白工程。

Step5：复制我们提供的 SDK 驱动文件，然后在 SDK 中，选中工程，在 Src 下直接按 Ctrl +V 将其复制到工程中来。



Step6: 双击 main.c，用如下程序进行替换。

```
#include "xaxivdma.h"
#include "xaxivdma_i.h"
#include "xhls_fast_corner.h"
#include "sleep.h"

#define DDR_BASEADDR          0x00000000
#define DISPLAY_VDMA          XPAR_AXI_VDMA_0_BASEADDR + 0
#define SOBEL_VDMA             XPAR_AXI_VDMA_1_BASEADDR + 0
#define DIS_X                  1280
#define DIS_Y                  720
#define SOBEL_ROW               512
#define SOBEL_COL               512
```

```

#define pi          3.14159265358
#define COUNTS_PER_SECOND (XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ)/64

#define SOBEL_S2MM    0x08000000
#define SOBEL_MM2S    0x0A000000
#define DISPLAY_MM2S  0x0C000000

u32 *BufferPtr[3];
u32 threshold=20;
static XHls_fast_corner fast;

//函数声明
void Xil_DCacheFlush(void);
// 所有数据格式为RGBA, 低拍透膜不适用
extern const unsigned char gImage_lena[1048584];
extern const unsigned char gImage_logo[284008];

void SOBEL_VDMA_setting(unsigned int width,unsigned int height,unsigned int s2mm_addr,unsigned int mm2s_addr)
{
    //S2MM
    Xil_Out32(SOBEL_VDMA + 0x30, 0x4); //reset S2MM VDMA Control Register
    usleep(10);
    Xil_Out32(SOBEL_VDMA + 0x30, 0x0); //genlock
    Xil_Out32(SOBEL_VDMA + 0xAC, s2mm_addr); //S2MM Start Addresses
    Xil_Out32(SOBEL_VDMA + 0xAC+4, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xAC+8, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xA4, width*4); //S2MM Horizontal Size
    Xil_Out32(SOBEL_VDMA + 0xA8, width*4); //S2MM Frame Delay and Stride
    Xil_Out32(SOBEL_VDMA + 0x30, 0x3); //S2MM VDMA Control Register
    Xil_Out32(SOBEL_VDMA + 0xA0, height); //S2MM Vertical Size start an S2M
    // Xil_DCacheFlush();

    //MM2S
    Xil_Out32(SOBEL_VDMA + 0x00,0x00000003); // enable circular mode
    Xil_Out32(SOBEL_VDMA + 0x5c,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x60,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x64,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x58,(width*4)); // h offset
    Xil_Out32(SOBEL_VDMA + 0x54,(width*4)); // h size
}

```

```
Xil_Out32(SOBEL_VDMA + 0x50,height); // v size
//Xil_DCacheFlush();
}

void DISPLAY_VDMA_setting(unsigned int width,unsigned height,unsigned
int mm2s_addr)
{
    Xil_Out32((DISPLAY_VDMA + 0x000), 0x00000003);      // enable
circular mode
    Xil_Out32((DISPLAY_VDMA + 0x05c), mm2s_addr);    // start address
    Xil_Out32((DISPLAY_VDMA + 0x060), mm2s_addr);    // start address
    Xil_Out32((DISPLAY_VDMA + 0x064), mm2s_addr);    // start address
    Xil_Out32((DISPLAY_VDMA + 0x058), (width*4));     // h offset
(640 * 4) bytes
    Xil_Out32((DISPLAY_VDMA + 0x054), (width*4));     // h size
(640 * 4) bytes
    Xil_Out32((DISPLAY_VDMA + 0x050), height);       // v size
(480)
}

void SOBEL_DDRWR(unsigned int addr,unsigned int cols,unsigned int lows)
{
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    for(i=0;i<cols;i++)
    {
        for(j=0;j<lows;j++)
        {
            b= gImage_lena[(j+i*cols)*4+2];
            g= gImage_lena[(j+i*cols)*4+1];
            r= gImage_lena[(j+i*cols)*4];
            Xil_Out32((addr+(j+i*cols)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
        }
    }
    Xil_DCacheFlush();
}

void SOBEL_Setup()
{
    //const int cols = 512;
    //const int rows = 512;
    XHls_fast_corner_SetRows(&fast, SOBEL_COL);
```

```
XHls_fast_corner_SetCols(&fast, SOBEL_ROW);
XHls_fast_corner_SetThrehold(&fast, threshold);
XHls_fast_corner_DisableAutoRestart(&fast);
XHls_fast_corner_InterruptGlobalDisable(&fast);
SOBEL_VDMA_setting(SOBEL_ROW, SOBEL_COL, SOBEL_S2MM, SOBEL_MM2S);
SOBEL_DDRWR(SOBEL_MM2S, SOBEL_ROW, SOBEL_COL);
//init_hls_sobel_dma(cols,rows, VIDEO_BASEADDR,
HLS_VDMA_MM2S_ADDR);
//DDRVideowr(HLS_VDMA_MM2S_ADDR, cols,rows);
XHls_fast_corner_Start(&fast);
}

void Set_background(u32 size_x,u32 size_y,u32 disp_addr)
{
    u32 i=0;
    u32 j=0;
    //u32 r,g,b;
    for(j=0;j<size_y;j++)
    {
        for(i=0;i<size_x;i++)
        {
            Xil_Out32((disp_addr+(i+j*size_x)*4),0);
        }
    }
    Xil_DCacheFlush();
}

void show_img(u32 x, u32 y, u32 disp_base_addr, const unsigned char * addr,
u32 size_x, u32 size_y)
{
    //计算图片左上角坐标
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    u32 start_addr=disp_base_addr;
    start_addr = disp_base_addr + 4*x + y*4*DIS_X;
    for(j=0;j<size_y;j++)
    {
        for(i=0;i<size_x;i++)
        {
            //if(type==0)
            //{
            //b = *(addr+(i+j*size_x)*4+2); //08
        }
    }
}
```

```
//g = *(addr+(i+j*size_x)*4+1); //60
//r = *(addr+(i+j*size_x)*4); //01
//}
//else
//{
    b = *(addr+(i+j*size_x)*4); //08
    g = *(addr+(i+j*size_x)*4+1); //60
    r = *(addr+(i+j*size_x)*4+2); //01
}

Xil_Out32((start_addr+(i+j*DIS_X)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
}
}

int main(void)
{
    //Xil_DCacheFlush();
    xil_printf("Starting the first VDMA \n\n");
    int status = XHls_fast_corner_Initialize(&fast,
XPAR_HLS_FAST_CORNER_0_S_AXI_CONTROL_BUS_BASEADDR);
    if(0 != status)
    {
        xil_printf("XHls_Sobel_Initialize failed \n");
    }
    SOBEL_Setup();
    DISPLAY_VDMA_setting(DIS_X,DIS_Y,DISPLAY_MM2S);
    Set_blackground(1280,720,DISPLAY_MM2S);
    ****
    for(i=0;i<614400;i++)
    {
        Xil_Out32(VIDEO_BASEADDR0+i,0);
    }
    ****
    while(1)
    {
        //show_img(0,0,VIDEO_BASEADDR0,&gImage_beauty[0],563,600);
        //sleep(5);

        //show_img(0,0,VIDEO_BASEADDR0,&gImage_miz702_rgba[0],375,400);
        //sleep(5);
    }
}
```

```

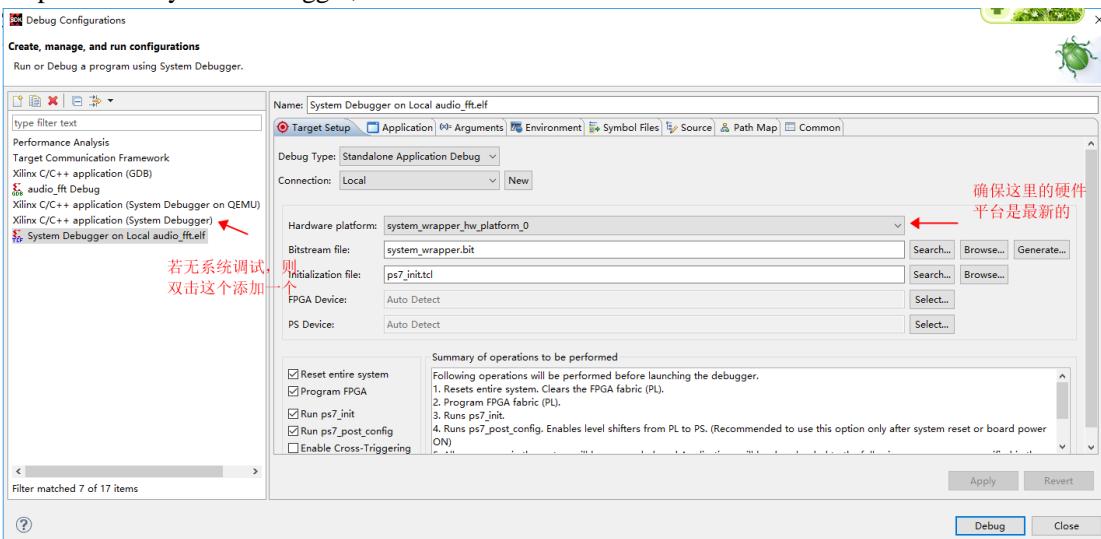
show_img(0,0,DISPLAY_MM2S,(void*)SOBEL_S2MM,512,512);
show_img(522,0,DISPLAY_MM2S,(void*)SOBEL_MM2S,512,512);
//show_img(0,513,DISPLAY_MM2S,&gImage_logo[0],355,200);
}

return 0;
}

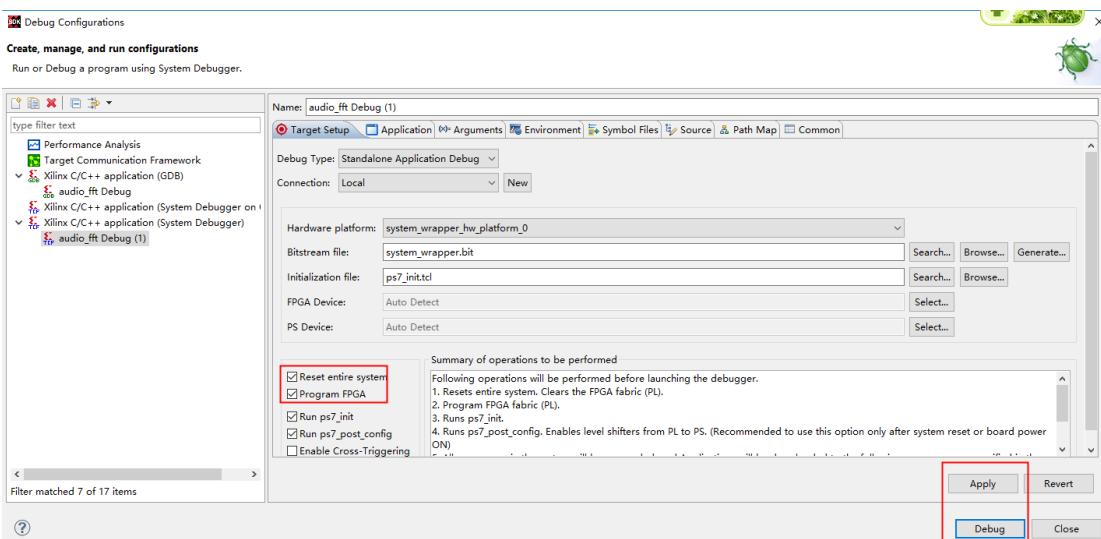
```

Step8: 右击工程，选择 Debug as ->Debug configuration。

Step9: 选中 system Debugger,双击创建一个系统调试。



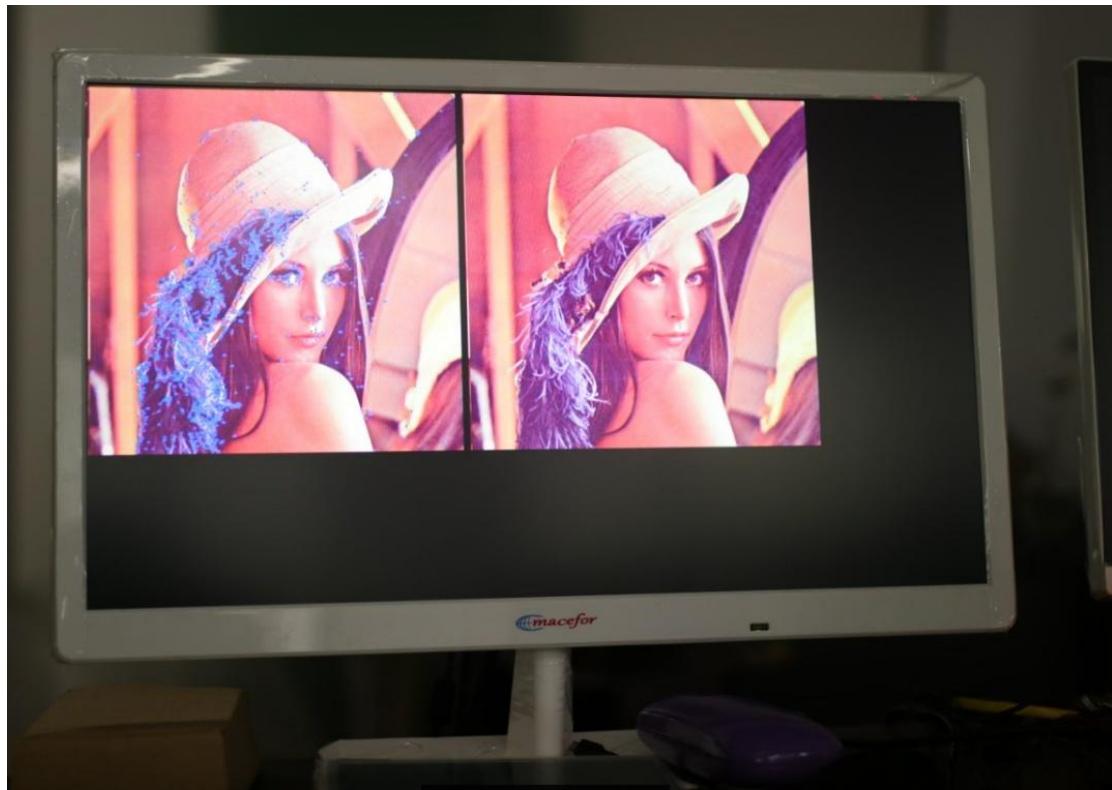
Step10: 设置系统调试。



打开系统自带的窗口调试助手，点击运行按钮开始运行程序。



程序运行之后，运行结果如下图所示：



11.5 程序分析

本章节的程序与第六章 SOBEL 的程序基本上是一致的，这是因为 HLS 产生的 IP 大多驱动方式是一样的，本章程序中快速角点检测 IP 的驱动函数如下图所示：

```
//const int cols = 512;
//const int rows = 512;
XHls_fast_corner_SetRows(&fast, SOBEL_COL);
XHls_fast_corner_SetCols(&fast, SOBEL_ROW);
XHls_fast_corner_SetThreshold(&fast, threshold);
XHls_fast_corner_DisableAutoRestart(&fast);
XHls_fast_corner_InterruptGlobalDisable(&fast);
SOBEL_VDMA_setting(SOBEL_ROW, SOBEL_COL, SOBEL_S2MM, SOBEL_MM2S);
SOBEL_DDRWR(SOBEL_MM2S, SOBEL_ROW, SOBEL_COL);
//init_hls_sobel_dma(cols,rows, VIDEO_BASEADDR, HLS_VDMA_MM2S_ADDR);
//DDRVideoWr(HLS_VDMA_MM2S_ADDR, cols,rows);
XHls_fast_corner_Start(&fast);
```

可以看出，这与第六章的程序相比较只是函数名改了个名字而已，驱动方式是一样的，大家在以后的设计中就可以仿照这里对 HLS 生成的 IP 进行驱动

11.6 本章小结

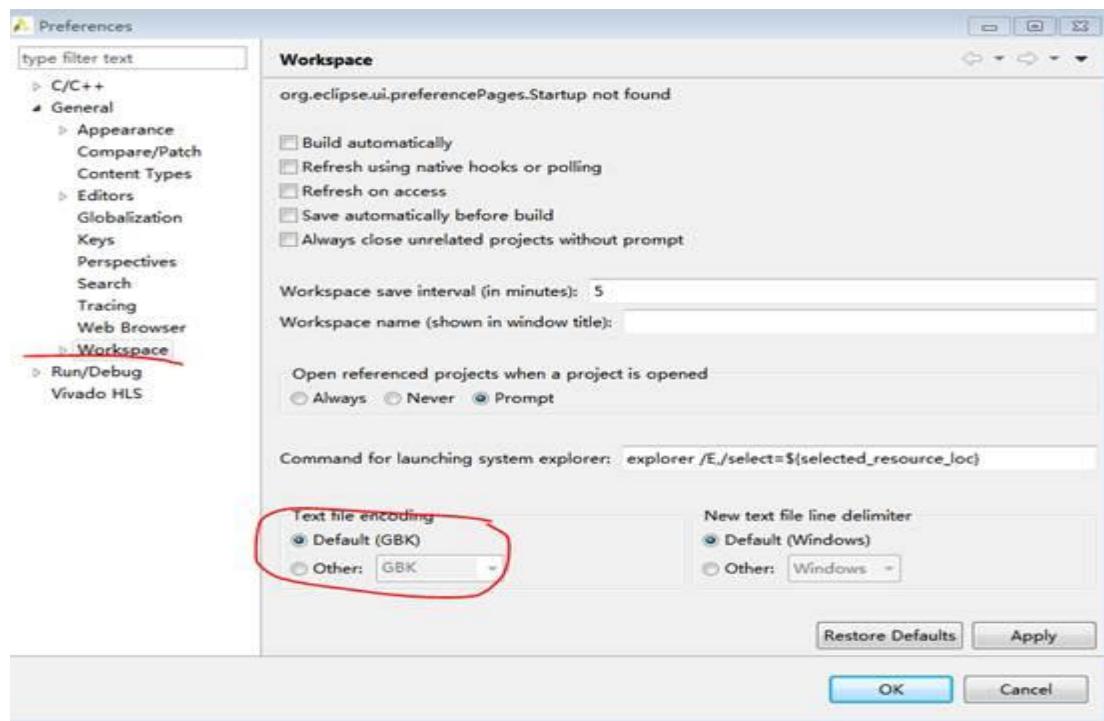
本章向大家介绍了如何利用 HLS 来实现快速角点检测算法，快速角点检测可以用于特征提取，因此具有很大的实用性，最后利用我们手头上的 MIZ 系列开发板对其进行了验证，通过本章，大家应该掌握利用 HLS 来进行基本的算法开发，这也大大节省我们的时间。

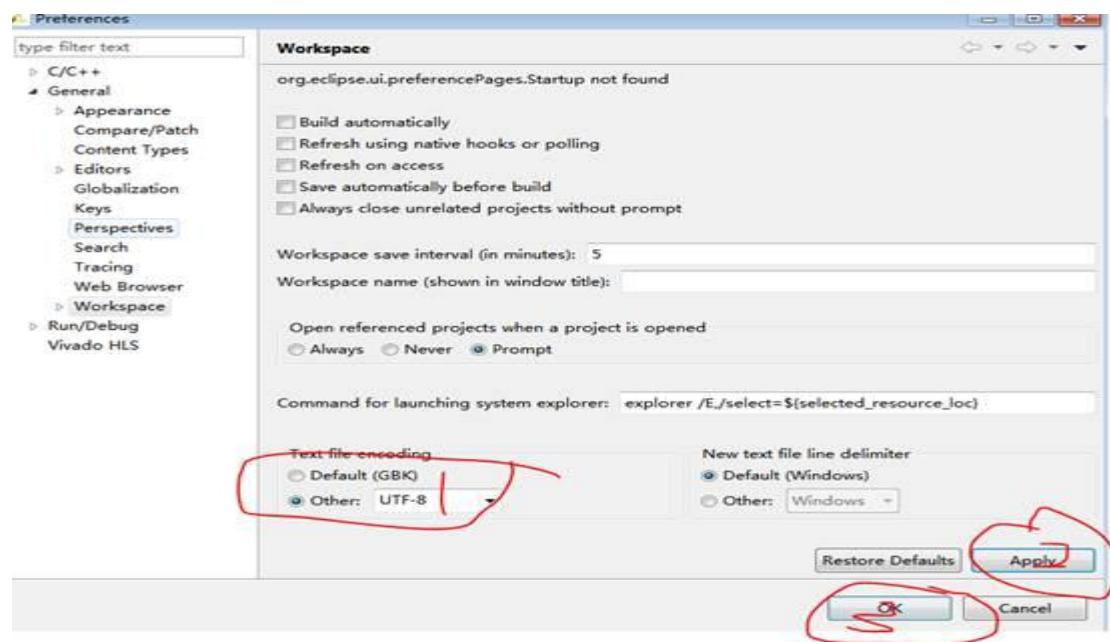
附录(常见问题汇总)

一、工具篇

1.1 HLS 中文注释乱码问题解决方案

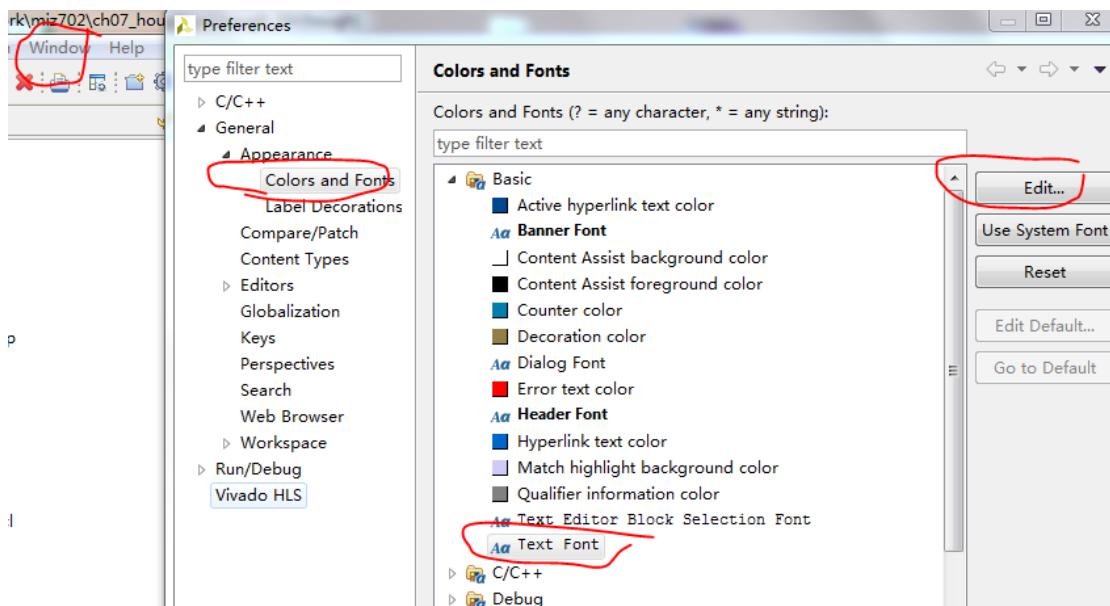
菜单栏选择 Window -> Preferences -> Workspace, 将红色标注的默认格式更改为 UTF-8

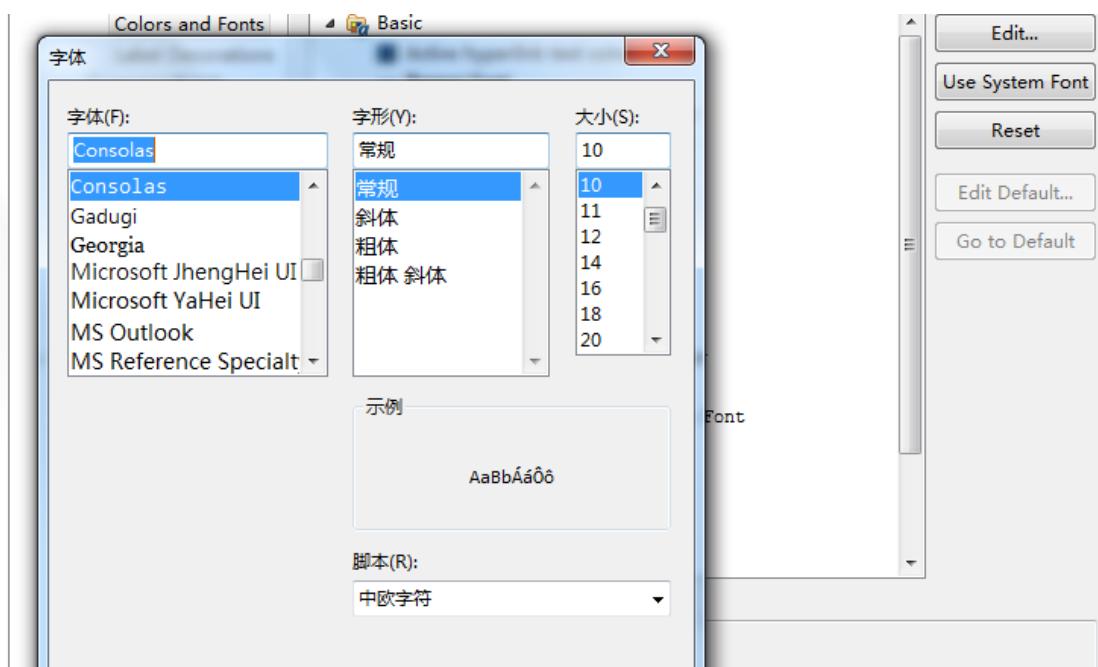




1.2 代码字体大小修改

在 HLS 代码编辑窗口中文注释字体代码偏小，解决办法如下，在字体里的脚本设置为中欧字符。





二、设计篇

2.1 hls::stream 仿真警告

```

6 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
7 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
8 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
9 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
10 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
11 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
12 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
13 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
14 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
15 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
16 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
17 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
18 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
19 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
20 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
21 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
22 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han

```

检查设计文件，查看是否是定义了 stream 类型的结构但是没有使用或者综合优化掉了。

2.2 仿真时使用 cvShowImage() 函数但是没有任何错误提示仿真界面直接关闭

- 1、检查测试代码中是否添加 waitKey(0) 或类似语句，笔者调试的时候就因为不小心把这句话屏蔽掉了，仿真界面直接关闭，导致查代码查了好久。
- 2、检查代码的正确性，因为 HLS 存在一些 Bug，可以尝试新建一个 Solution 再次仿真

一个完美的结束
意味着一个新的开始！

www.osrc.cn

米联客
技术论坛
秀出你的风采！