2017 年全国大学生 FPGA 创新设计邀请赛

基于 ZYNQ 的运动目标检测系统

参赛组别：　　研究生组

队伍名称：　FPGA_CLUB1

参赛队员：赵博　郭海　李鹏飞

完成时间：　　2017.12

摘要

近年来，智能交通系统以及智能监控等人工智能领域受到广泛关注，其中运动目标检测系统作为监控系统中不可或缺的组成部分，研究小体积、低功耗、速度快的运动目标检测系统显然具有重要的意义。

本系统采用的是 Xilinx 公司 Zynq-7000 系列全面可编程片上系统平台，片内集成了双核 ARM Cortex-A9 处理器和 Xilinx 7 系列 FPGA，采用软硬件协同设计方法，将图像采集、图像处理、运动目标检测、视频显示集合到一个嵌入式系统中。图像采集方面采用 OV7670 摄像头进行采集，图像处理和运动物体检测方面，利用 Xilinx 高级开发工具 Vivado HLS （High-Level Synthesis)完成图像处理算法，并利用此工具把三帧差分法的运动目标检测算法成功封装成硬件 IP 移植到 FPGA 中，图像显示方面采用 VGA 接口连接显示器进行显示。整套系统通过各模块之间相互连接配合能够完成对运动目标实时有效的检测。本设计完成的主要工作包括:

（1） 利用 Vivado HLS 工具，完成了相关的图像处理和运动目标检测算法硬件加速 IP 核的设计，其中包括灰度图转换、高斯滤波、形态学处理、三帧差分法运动目标检测等算法，实现了运动物体的检测。为了验证其中的一些图像算法的正确性，在 HLS 中，编写了测试代码，验证了图像处理 IP 的效果。

（2） 利用 Vivado 工具，以图像化界面的方式，在 ZYNQ 的 PL 部分完成硬件工程的搭建，主要包括硬件加速图像处理 IP 核模块的移植、VDMA 模块配置、AXI 互联、VGA 显示模块和 ZYNQ 处理器相互之间的连接

（3） 在 ARM 部分实现了系统的软件设计，对 VDMA IP 核进行了配置。使用 OV7670 摄像头实现视频图像采集，以及利用自定义硬件加速 IP 核实现了视频图像处理和运动物体的检测。

本系统用软硬件协同的方法成功实现了基于 ZYNQ 的运动物体检测系统设计，并对系统进行了效果测试。对运动物体检测算法进行了硬件加速和综合结果分析，并对不同的运动物体进行了测试。经测试验证，系统在运行速度和运动物体检测上均达到了预期的效果。

ABSTRACT

In recent years, the field of artificial intelligence, such as intelligent transportation system and intelligent monitoring, has drawn much attention. Moving object detection system is an indispensable part of the monitoring system. It is obvious that researching small size, low power consumption and fast moving object detection system Significance.

The system uses Xilinx's Zynq-7000 series of fully programmable system-on-chip platform, dual-core ARM Cortex-A9 processor and Xilinx 7 Series FPGAs integrated on-chip, the use of hardware and software co-design method, the image acquisition, image processing, exercise Target detection, video display set to an embedded system. Image Acquisition Using OV7670 camera for image acquisition, image processing and detection of moving objects, the use of Xilinx advanced development tools Vivado HLS (High-Level Synthesis) to complete the image processing algorithms and use of this tool to three-frame difference method of moving target detection algorithm is successful Packaged into hardware IP ported to the FPGA, the image display using VGA interface to connect the display. The main work completed in this design includes:

(1) Using the Vivado HLS tool, we completed the design of the hardware-accelerated IP core for image processing and moving object detection algorithms, including algorithms such as grayscale transform, Gaussian filter, morphological processing, three-frame difference detection, Achieve the detection of moving objects. In order to verify the correctness of some of the image algorithms, a test code was written in HLS to verify the effect of image processing IP.

(2) The use of Vivado tools, graphical interface to the PL part of the ZYNQ to complete the hardware construction, including hardware accelerated image processing IP core module porting, VDMA module configuration, AXI interconnect, VGA display module and ZYNQ processing The connection between the devices

(3) In the ARM part of the system to achieve the software design, including VDMA IP core and custom hardware acceleration IP core driver design. With OV7670 camera for video image capture, and use of custom hardware to accelerate the IP core to achieve a video image Processing and detection of moving objects.

The system uses software and hardware synergetic method to successfully realize the design of the moving object detection system based on ZYNQ, and carries on the effect test to the system. The hardware acceleration and comprehensive result analysis of the moving object detection algorithm was carried out, and different moving objects were tested. After testing and verifying, the system has achieved the expected effect on the running speed and moving object detection.

目录

第一章 设计概述 1
1.1 设计目的 1
1.2 应用领域 2
1.2.1 智能交通系统 2
1.2.2 智能监控领域 2
1.3 运动物体检测系统难点 3
1.4 论文主要内容及结构 3
第二章 基于 ZYNQ 的运动目标检测系统总体结构 5
2.1 系统总体结构 5
2.2 开发平台 6
2.2.1 Zynq 芯片简介 6
2.2.2 EES-288 开发板 8
2.2.3 DDR 存储器 10
2.3 视频的采集与显示 10
2.3.1 OV7670 图像采集 10
2.3.2 VGA 显示 12
第三章 基于 Vivado HLS 的图像处理实现 15
3.1 Vivado HLS 简介 15
3.1.1 HLS 介绍 15
3.1.2 HLS 加速 FPGA 的视觉开发 15
3.1.3 HLS 设计流程 17
3.2 HLS 图像处理 17
3.2.1 灰度化 17
3.2.2 高斯滤波 18
3.2.3 形态学处理 19
3.2.4 运动检测算法 21

第四章 运动目标检测系统设计 25
4.1 开发流程概述 25
4.2 硬件工程设计 25
4.2.1 OV7670 寄存器配置 25
4.2.2 图像采集模块 27
4.2.3 AXI 接口 28
4.2.4 AXI VDMA 配置 29
4.2.5 三帧差分法设计运动目标检测 IP 31
4.2.6 硬件工程 34
4.3 软件工程设计 37
4.3.1 SDK 程序编写 37
4.3.2 制作启动镜像 38
第五章 系统调试与结果分析 41
5.1 系统平台搭建 41
5.2 器件资源利用率 42
5.3 遇到的主要问题及解决方法 42
第六章 总结与展望 45
6.1 成果总结及创新性分析 45
6.2 展望 46
6.3 心得体会 46

第一章 设计概述

1.1 设计目的

随着时代的发展，个人和公共的安全越来越受大家的关注。为了应对各种潜在未知的威胁，全世界在保护性行为和设施上的投入逐年递增。视频监控作为一个能直观、及时、准确提供丰富的信息的安全工具也倍受大家关注，现已经广泛应用于监控、安防、家居生活以及智能交通等诸多方面。尤其是近几十年来，机器视觉、人工智能及模式识别技术等理论研究获得日新月异的发展，视频监控技术也随之出现大的变革，逐渐朝着数字化、智能化、网络化等方向发展，从而使得视频监控成为了人类生活中不可或缺的一部分，为个人和公共的安全的保证提供极大的大助力。而运动目标检测与跟踪作为智能视频监控系统中的最基本、最核心的问题，成为众多科研学者密切关注的一个前沿课题。与过去基于无线电信号、声学信号的传感器系统，例如雷达比较，计算机视觉所采集到的图像具有更多更有用的信息，相比而言，一个很小的摄像头检测的范围更广更全面，同时视觉系统几乎不受四周环境的影响，是被动感知系统。从对计算机视觉的研究理论来讲，计算机视觉具有像人类视觉系统一样的观测能力，但不会因人长久的使用而感到疲劳，利用摄像头来采集图像对特定场景进行监控不但价格低廉，而且还不会占用太大空间。随着进一步的图像处理技术的研究，计算机视觉不单单可以应用在安防监控上，还可以应用在智能交通等其他领域。

虽然现在对运动目标检测算法的研究已经比较成熟，但是至今能够硬件上实现并能实时运行的运动目标检测算法并不多，在 PC 机上实现的运动物体检测，具有体积大、功耗高等缺陷，而基于嵌入式系统的运动物体检测具有更灵活的优势，因此加大对基于硬件的运动目标检测与跟踪系统的研究，仍然具有非常高的实用价值。目前的处理器件一般基于 ARM、FPGA 等单一芯片，能够利用 ARM 搭建嵌入式系统，但处理计算量大的算法时，其处理能力有限，达不到理想的处理效果。FPGA 拥有强大的并行处理能力，丰富的逻辑资源，但不能独立运行操作系统的能力。对于运动车辆检测与跟踪这类算法复杂，计算量相对庞大，建立一种实时性良好的系统，以上单独平台很难达到理想效果。因此，基于 FPGA+ARM 的嵌入式系统的研究，可以很好的弥补以上单一平台的不足。本系统所用的 Zynq 平台就是基于 ARM+F P GA 架构，充分利用二者的结构优势，对系统进行软硬划分，设计了一个体积小、性能强的实时操作处理系统，来对运动物体进行检测，旨在为智能监控领域提出可行的解决方案。

1.2 应用领域

1.2.1 智能交通系统

随着国家经济的飞速发展和交通运输的不断扩大，交通拥堵已然成为很普通的现象，这不仅阻碍了城市的发展而且对环境的污染甚是严重。然而，面对与日俱增的拥堵交通道路，仅依靠传统的技术如修建道路、增加新的交通设施，这对于现如今的交通发展趋势而言，远远达不到要求。因此，智能交通系统(ITS)已经成为了一个十分重要的发展方向。通过合理且科学运用人工智能、信息通信、图像处理、计算机网络、自动控制等技术，将它们有效的集成，来加强车辆、道路和使用者之间的联系，建立一种实时、准确、高效的综合交通运输管理系统。计算机视觉技术对车速、车流量、车辆密度等内容的采集，在智能交通中占据着十分重要的作用，而运动物体检测又作为智能交通系统的重要研究内容，近年来，国内外许多研究部门和高效都相继在这一领域进行研究探索，并取得了一定研究成果。 在我国的高速公路道路上方固定一部摄像机以获取路况视频，通过对视频进行实时分析，从而检测出当前场景内的车辆数目、车辆行驶速度以及某时间段内的车流量。此系统可以帮助交管部门在高峰出行时断，掌握各个路段的实时车流情况，便于快速发现拥堵路段，为交警做出灵活决策提供帮助，同时为广大出行的人们避免了不必要的等待时间。过去长时间，运动车辆检测的研究与应用都是依托 PC 机平台，但 PC 机本身体积大、功耗高等缺陷，对有的特定场景不太合适，因此类似本设计的嵌入式系统开发应用是必然趋势。

1.2.2 智能监控领域

近年来，在习主席全面建成小康社会的带动下，我国的国民经济取得了令人骄傲的成就，居民收入水平不断提高，安防民用监控成为行业热门话题，国家加大力度建设“平安城市”、“平安校园”，个人对人身安全、财产安全等越来越重视，防盗以及家庭老人、儿童、宠物的看护，慢慢也受到大家的重视，机器视觉是人类眼睛的延伸，正所谓“眼见为实”，安防民用监控市场需求愈加凸显。运动物体检测这种智能监控类技术直接关乎人身财产安全，毫无疑问将成为一种家庭必须品。

从国家安防层面来看，国家安防对智能安防的需求有增无减，其中视频监控在安防行业所占市场份额最大，将近百分之五十，已然成为安防系统的核心。而且由于 2016 年世界恐怖主义袭击频现，均会增加各个国家对于安防升级的需求，智能安防可以将事后追查升级到事前预防，并且可以 24 小时无死角监控。运动物体检测作为智能安防系统重要的一部分系统组

成，可以有效地将监控视频中的运动物体提取出来，发现入侵目标，大幅度减轻人工视频监控的强度，做到有的放矢地第一时间联动报警。

1.3 运动物体检测系统难点

近年来，许多学者或企业对视觉的运动物体检测做了许多研究，也取得了一定的成果，但并没有完全的解决实际中存在的问题，主要包括：

（1） 运动物体的检测缺乏自适应性

现实场景中的运动目标检测不应受环境影响，比如阴天、雨天、雪天，但在这样的环境下，图像的对比度较低，会直接影响运动物体检测。

（2） 体积和功耗问题

很据我们的应用场景分析，作为监控等使用，对于检测系统在体积和功耗上要求也要比其他方面的应用更为严苛。

（3） 系统平台的选取

在图像处理领域常用的嵌入式处理器包括 ARM、DSP、ASIC、FPGA 等，运动物体检测算法复杂且数据量大，因此系统平台的性能对运动目标的检测效果是一个很重要的影响因素。鉴于此，本系统选择了 ZYNQ 平台，合理地划分软硬件部分，以此充分利用 FPGA 与 ARM 各自的强项，建立一种运行速度快，检测失误率低的系统。本项目设计的平台以及系统搭建会在后续章节详细介绍。

1.4 论文主要内容及结构

本文的主要研究内容是基于 Xilinx 公司 ZYNQ 平台的运动物体检测系统设计。系统主要分为图像采集和显示，以及图像处理模块，然后根据模块的性能和资源需求，进行软硬件的划分。在 FPGA 部分，利用 Vivado 工具创建硬件工程，各模块之间连接及相应配置，完成系统的搭建，包括添加自定义图像处理 IP 核，视频图像数据的 VDMA 连接设置以及 VGA 显示等，其中自定义图像处理 IP 核由高级综合工具 Vivado HLS 设计完成，用来实现图像预处理的硬件加速。在 ARM 部分实现对 VDMA 的传输配置，从而来实现系统软硬件的调配。

本文主要分为六章，各章的内容简要说明如下：

第一章主要介绍对作品设计进行了概述，说明了我们设计的目的以及主要的应用领域，并说明了运动物体检测系统设计存在的难点。

第二章对设计作品的系统总体结构进行了介绍，包括系统总体结构的划分，开发平台以及对相应视频采集与显示模块的介绍。

第三章主要介绍了基于 Vivado HLS 的图像处理实现，包括对 HLS 工具的介绍、设计流程以及相关图像处理算法的介绍。

第四章介绍了基于 ZYNQ 运动物体检测系统的具体设计，包括对开发流程的概述，硬件工程设计和软件工程设计几部分，包括用 HLS 完成了改进的三帧差分实现运动目标检测的 IP 核设计，并用 Vivado 创建硬件工程，对图像传输通道 VDMA 作相应的配置，最后搭建成完整的视频通道，并在 SDK 验证了其正确性。

第五章介绍了系统调试与实验结果分析。包括系统平台的搭建，对器件资源利用率的分析，最后简述了系统设计过程中遇到的主要问题与解决方法。

第六章是对所设计作品的总结及创新性分析，并对设计作品的一个展望。

第二章 基于 ZYNQ 的运动目标检测系统总体结构

2.1 系统总体结构

本系统采用的是依元素公司的 ES-288 开发板，其核心是可以通过软硬件协同设计的方法来实现运动物体的检测，其中，视频的处理是通过 FPGA(Programmable Logic,PL)来完成的，ARM(Processing System,PS)端控制图像的存储和 VDMA 的配置。系统结构框图如图 2-1 所示。

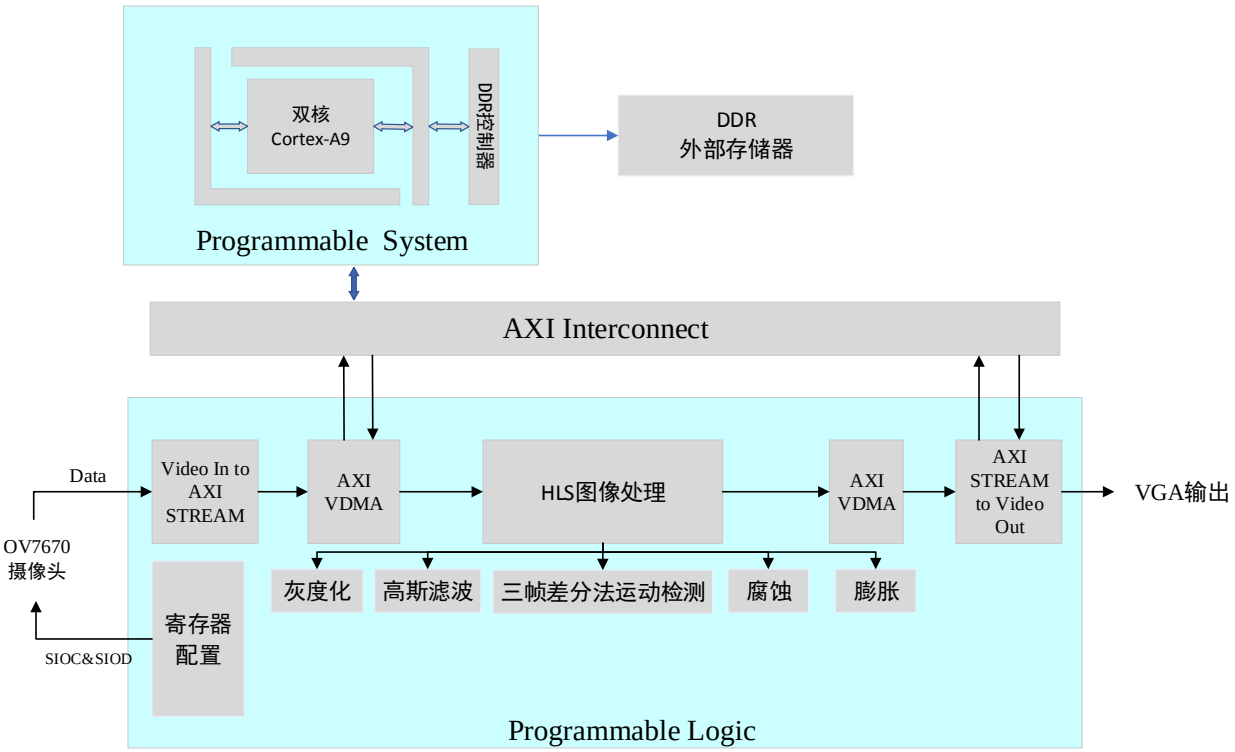


图 2-1 运动检测系统结构框图

本系统采用硬件加速的方式对图像进行处理，实现 FPGA 的运动目标检测。从图 2.1 可以看出，ZYNQ 中 PL 部分实现图像的传输以及图像处理的运动目标检测算法，PS 部分对相关模块进行驱动编写以及各模块的控制，系统总的流程为：通过 OV7076 摄像头采集图像，把源图像分三路缓冲在外部 DDR 存储器中，利用图像传输通道（VDMA）读取 DDR 中的连续三帧图像数据并传输到硬件加速的图像处理 IP 核，完成图像处理加速部分，处理后的图像再由 VDMA 传输到 DDR 中，最后用 VGA 显示模块完成运动检测效果的显示。

本系统主要分为四大模块：图像采集模块、HLS 图像加速处理模块、视频图像传输模块（VDMA）、VGA 显示模块。各部分功能如下：

（1） 图像采集模块

图像采集部分通过 OV7670 摄像头采集图像，并将数据通过 VDMA 存储到外部存储器 DDR 中，DDR 中的数据可进一步由 VDMA 进行读取传输。

（2） HLS 图像加速处理模块

HLS 图像加速处理模块是由 Vivado HLS 工具封装成的 IP 模块，在 FPGA 中实现图像的处理，其中包括灰度化、高斯滤波、三帧差分运动检测、二值化、形态学等处理操作，完成运动目标的检测。

（3） 视频图像传输模块（VDMA）

VDMA 主要完成读取 DDR 中的连续三帧图像给图像处理模块进行处理以及将处理完的图像输出给 DDR 供 VGA 显示，其中包括 VDMA 的接口配置和驱动程序编写等

（4） VGA 显示模块

VGA 显示模块主要是对运动物体检测的效果显示。

2.2 开发平台

2.2.1 Zynq 芯片简介

ZYNQ 是 Xilinx 公司推出的高性能和低功耗的可扩展处理器平台，该芯片结构采用 28nm 的高 K 金属栅极工艺。每个 ZYNQ 系列的单芯片内都集成了 ARM Cortex-A9 系列处理器系统（Processing System，PS）和 Xilinx 可编程逻辑（Programmable Logic，PL），并且 PS 中集成了内存控制器，外部存储器 DDR 控制模块以及大量的外设，使得 ARM Cortex-A9 硬核处理器可以在 ZYNQ 中不需要可编程逻辑（PL）部分的介入进行独立运行。而之前在一个只有 FPGA 结构的芯片中，工程师如果想要实现一些硬件语言很难描述的复杂算法的话，只能通过在 FPGA 中构造软核处理器来解决。ZYNQ 平台则通过嵌入 ARM 硬核处理器成功的解决了上面的难题。该平台兼具 ARM 处理器的灵活性和 FPGA 的可编程等特点。

基于 ZYNQ 的设计方式较传统嵌入式系统设计有以下优点：

（1） 个性化设计

基于 ZYNQ 平台的嵌入式设计可以利用其 PL 部分设计出开发板上没有的外设接口以及其它的非标准外设。

（2） 硬件加速

与传统的嵌入式系统相比较，Xilinx 公司的 ZYNQ 平台因为嵌入了可编程逻辑部分，因此具有强大的并行运算处理能力。开发者们可以通过利用 ZYNQ 中 PL 部分的强大并行运算处理能力，解决信号处理方面中的大量复杂数据处理问题，让处理器系统可W完成更多复杂的大数据处理任务。在多线程处理非常流行的当下，并行运算处理能为对于提高系统运行速度具有举足轻重的作用。

（3） 高速 AXI 接口

ZYNQ 平台中 PS 与化之间的数据传输任务是由 AXI 接口完成，AXI 接口能够以非常低的功耗去支持千兆位级的高速数据传输，因此能够解决数据通信中速度瓶颈问题。ZYNQ 中的 AXI 总线是 Xilinx 公司专口针对该拓展平台进行的优化，更具兼容性的同时也具有更低的延时。

（4） 成本和功耗

此次推出的 ZYNQ-7000 的四个系列产品（ZYNQ-7010，7020，7030，7040），都使用的是 800Mhz 的主频，配合单/双精度浮点运算，在相同的时钟频率下可以实现比单独的 Cortex-A9 处理器更好的性能，更具拓展性。ZYNQ 中的 PL 部分非常灵活，可设计出非标准外设接口和大部分数字逻辑电路，降低了开发成本。

ZYNQ 架构如图 2-2 所示。ARM 与 FPGA 之间的数据通信可通过 AXI 接口和 EMIO 接口实现。PS 部分包含 Cortex-A9 硬核处理器，外部 DDR 存储器接口和普通接口外设。Cortex-A9 硬核处理器中有外部存储器控制器和串行外设接口控制器；PL 部分包含可编程逻辑模块、数字信号处理模块、模数转换器等。

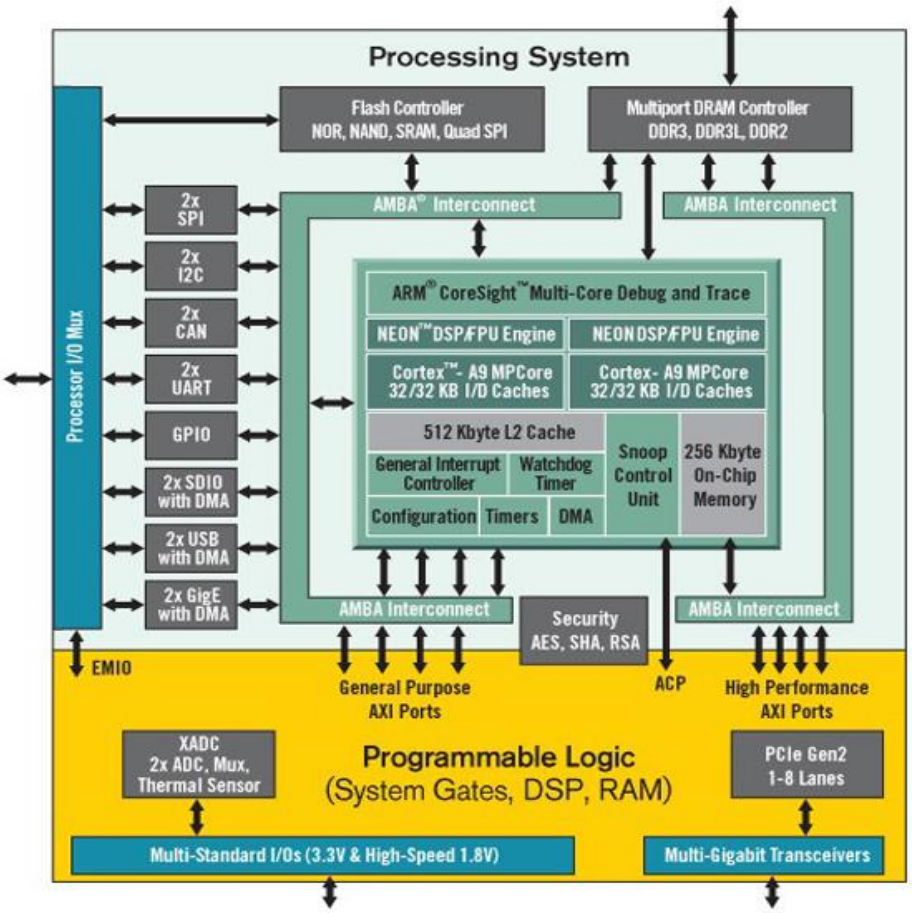


图 2-2 ZYNQ 架构

2.2.2 EES-288 开发板

EES-288 是基于 Xilinx Zynq-7000 系列 FPGA 的最小系统板。主芯片 XC7Z010CLG400-1 内部含有集成处理系统（PS）和可编程逻辑资源（PL）。PS 部分集成了功能丰富的两颗 Cortex-A9 核心，同时还包含片上存储器、外部存储接口和一组丰富的 I/O 外设，可以灵活应用于具有 FX8 接口的 FPGA 母板。EES-288 提供 SD 接口、UART 接口以及 USB2.0 接口，可以与大量外设进行通信，使其广泛应用在便携式、低功耗应用。另外还提供了一个 Ethernet 接口，该接口兼容 IEEE 802.3 标准，支持 10/100.1000M 传输速率，可以满足不同的网络应用开发。搭配 Xlinx 的 Vivado 软件，可灵活的进行系统设计和开发，它为所有可编程 SOC 提供了一个硬件设计开发环境。

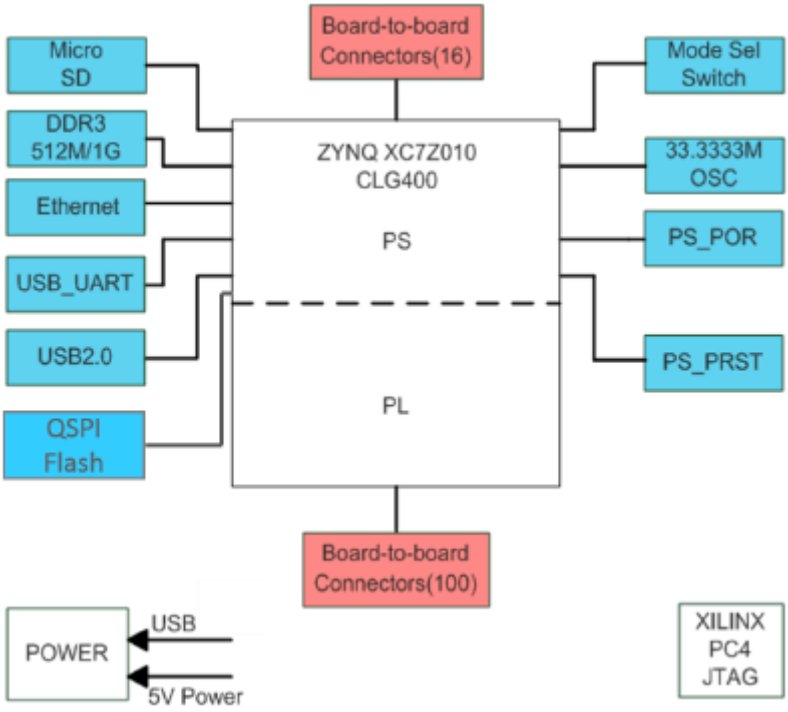


图 2-3 EES-288 开发板系统结构

硬件规格：

PS

- Zynq-7 XC7Z010-CLG400-1
- 1GB DDR3 Memory SODIMM
- 128Mb Quad SPI Flash Memory
- Secure Digital(SD) connector
- USB-to-UART bridge
- 10/100/1000 Ethernet PHY
- USB2.0 PHY

PL

- 2Bit User LED
- 2 个 2*8PMOD 输出接口
- 2 个 80pin 的 FX8 连接器

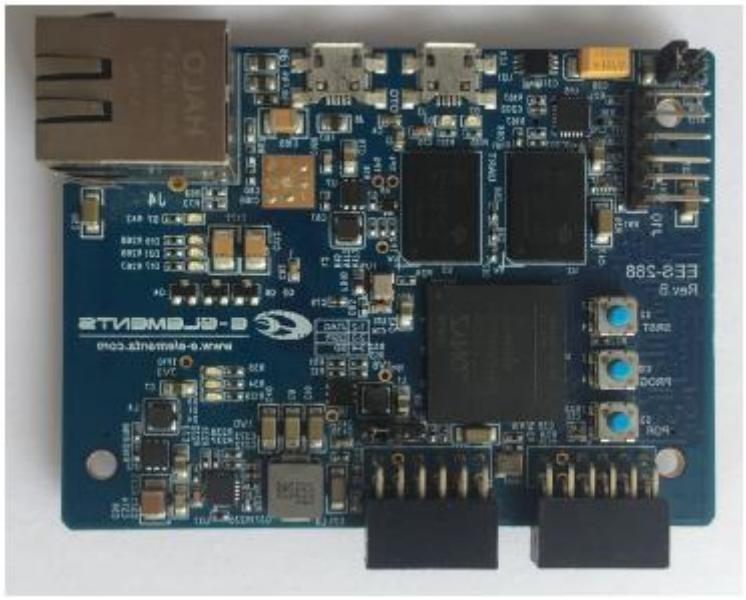


图 2-4 EES-288 开发板实物图

2.2.3 DDR 存储器

ZYNQ 中 PS 部分的 DDR 接口控制器可以支持 DDR2，DDR3 和 LPDDR2 设备。DDR 控制器由 AXI 存储器接口、负责调度和传输功能的控制器以及数字 PHY 控制器三部分构成。

EES-288 开发板中配备了容量为 1G 的 DDR 存储器。系统每次都是通过 4 个 64 位 AXI 端口来访问 DDR 外部存储器中的数据。其中一个 AXI 端口连接到 ARM 处理器系统中的 L2-Cache 和 ACP 接口，另外两个专用 AXI 端口连接到 AXI-HP 接口，最后一个端口是由其他的设备与 AXI 端口共享。

2.3 视频的采集与显示

2.3.1 OV7670 图像采集

本设计采用 OV7670 摄像头进行图像的采集，OV7670 是 OV（OmniVision）公司生产的一颗 1/6 寸的 CMOS VGA 图像传感器，如图 2-5 所示。该传感器体积小、工作电压低，提供单片 VGA 摄像头和影像处理器的所有功能。通过 SCCB 总线控制，可以输出整帧、子采样、取窗口等方式的各种分辨率 8 位影像数据。该产品 VGA 图像最高达到 30 帧/秒。用户可以完全控制图像质量、数据格式和传输方式。所有图像处理功能过程包括伽玛曲线、白平衡、度、色度等都可以通过 SCCB 接口编程。OmmiVision 图像传感器应用独有的传感器技术，通过减少或消除光学或电子缺陷如固定图案噪声、托尾、浮散等，提高图像质量，得到清晰的稳定的彩色图像。

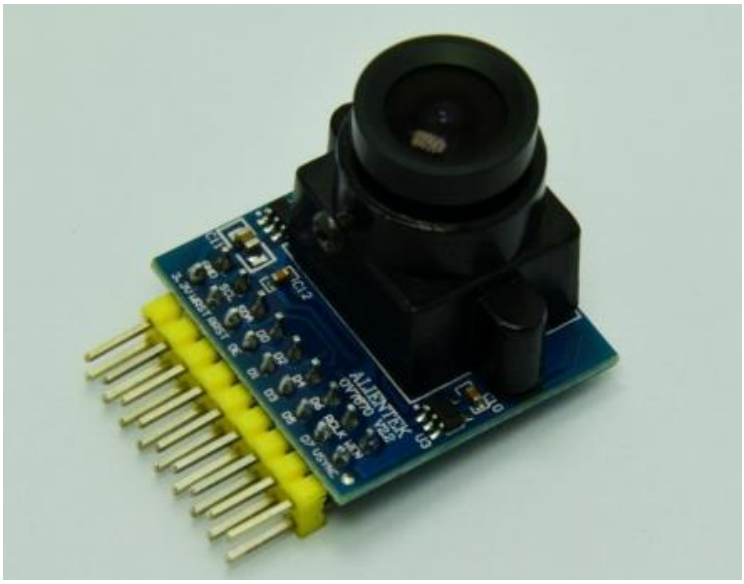


图 2-5 OV7670 摄像头实物图

OV7670 的特点有：

- 高灵敏度、低电压适合嵌入式应用
- 标准的 SCCB 接口，兼容 IIC 接口
- 支持 RawRGB、RGB(GBR4:2:2，RGB565/RGB555/RGB444)，YUV(4:2:2)和 YCbCr（4:2:2）输出格式
- 支持 VGA、CIF，和从 CIF 到 40*30 的各种尺寸输出
- 支持自动曝光控制、自动增益控制、自动白平衡、自动消除灯光条纹、自动黑电平校准等自动控制功能。同时支持色饱和度、色相、伽马、锐度等设置。
- 支持闪光灯
- 支持图像缩放

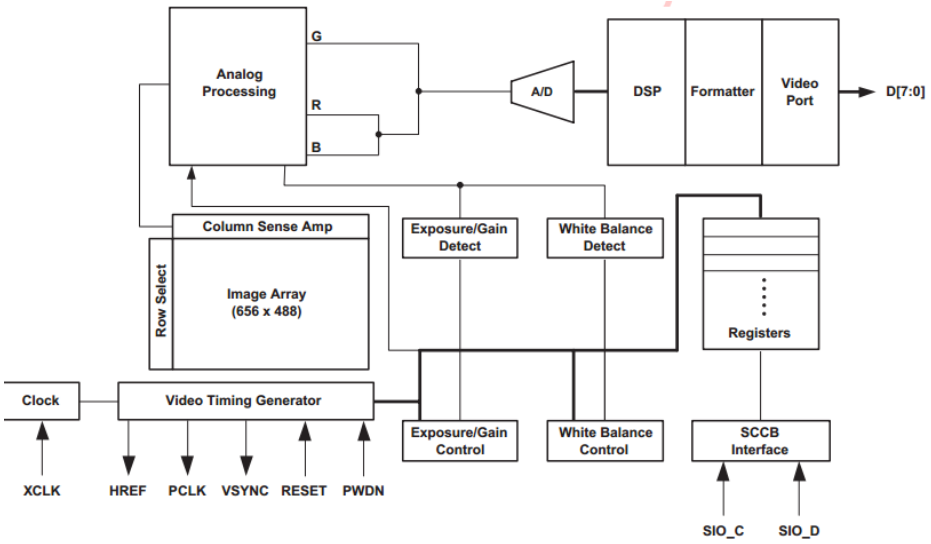


图 2-6 OV7670 内部结构

2.3.2 VGA 显示

VGA（Video Graphics Array）是 IBM 于 1987 年提出的一个使用模拟信号的电脑显示标准， VGA 接口即电脑采用 VGA 标准输出数据的专用接口，如图 2-7 所示。VGA 接口共有 15 针，分成 3 排，每排 5 个孔，显卡上应用最为广泛的接口类型，绝大多数显卡都带有此种接口。它传输红、绿、蓝模拟信号以及同步信号(水平和垂直信号)。

图 2-7 VGA 接口

表 2-1 VGA 接口主要引脚介绍

信号线	定义
HS	行同步信号（3.3V 电平）
VS	场同步信号（3.3V 电平）
R	红基色（0~0.714V 模拟信号）
G	绿基色（0~0.714V 模拟信号）
B	蓝基色（0~0.714V 模拟信号）

VGA 显示器采用的是逐行扫描的方式进行扫描，在 VGA 屏幕上受水平同步信号控制自左向右扫描，受垂直同步信号 VGYNC 控制自上而下扫描。现在以 640*480 的图像为例，则扫描是从(0,0)点开始扫描，扫描到(639,479)截止，它的扫描顺序如图 2-8 所示。

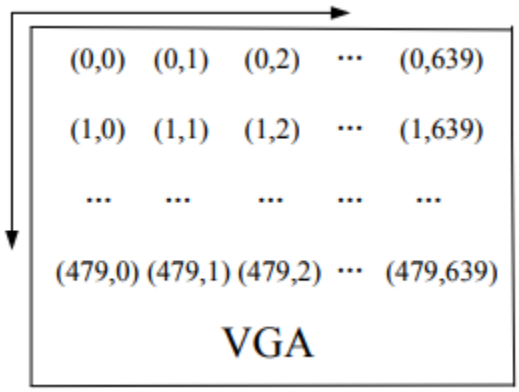


图 2-8 VGA 扫描顺序

扫描从屏幕的左上角的(0,0)点开始扫描，向右逐点进行扫描，知道扫描到第 639 个像素，到达整行的最右端后，又从下一行的最左边开始扫描下一行，再扫到最右边那个像素点，重复以上过程直到扫描到最后一行的最右边那个像素点截止，完成一帧图像的扫描。完成一帧图像的扫描后，又会回到起点继续上述过程，开始下一帧图像的扫描。

VGA 显示的时序如图 2-9 所示。

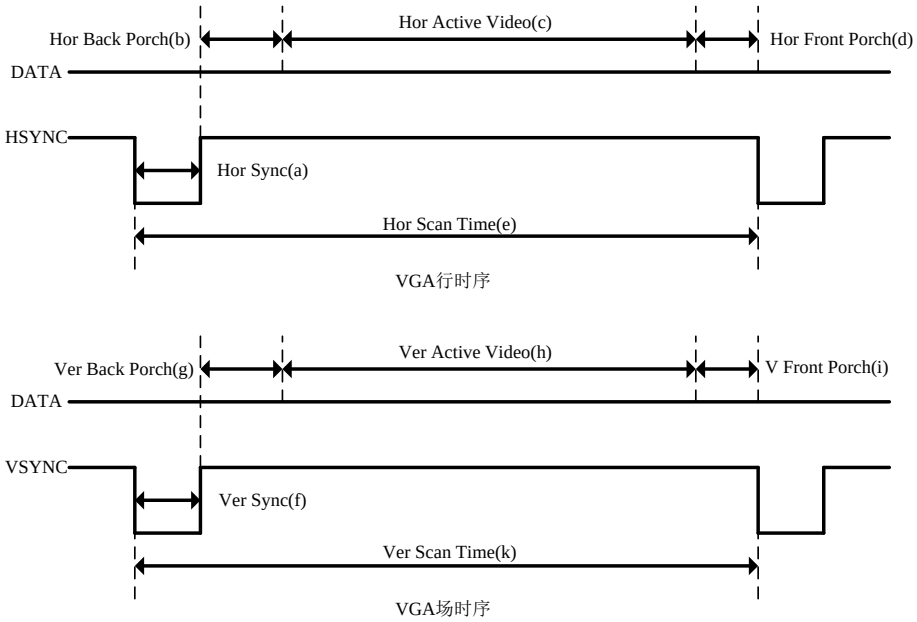


图 2-9 VGA 的行时序与场时序

由 VGA 的行时序可知：每一行都有一个负极性行同步脉冲，是数据行的结束标志，同时也是下一行的开始标志。在同步脉冲之后为显示后沿（Back porch b），在显示时序段（Display interval c）显示器为亮的过程，RGB 数据驱动一行上的每一个像素点，从而显示一行。在一行的最后为显示前沿（Front porch d）。在显示时间段（Display interval c）之外没有图像投射到屏幕是插入消隐信号。同步脉冲(Sync a)、显示后沿（Back porch b）和显示前沿（Front porch d）都是在行消隐间隔内(Horizontal Blanking Interval)，当消隐有效时，RGB 信号无效，屏幕不显示数据。

VGA 的场时序与行时序基本一样，每一帧的负极性脉冲（Sync a）是一帧的结束标志，同时也是下一帧的开始标志。而显示数据是一帧的所有行数据。

第三章 基于 Vivado HLS 的图像处理实现

3.1 Vivado HLS 简介

3.1.1 HLS 介绍

Vivado HLS 是嵌入在 Vivado 设计套件中，是 Xilinx 公司发布面向新一代可编程逻辑器件的高层次设计工具，是以后 FPGA 的重要设计工具。在 HLS 工具中，，不用受 HDL 语言的限制，能够选择不同的高级语言(C ，C++， System C)进行 FPGA 设计，经仿真、优化和综合等步骤后即可生成 RTL(Register Transfer Level)级代码，大幅降低了产品的研发周期。HLS 更注重系统级建模，是一种全新的设计方法，能够将设计效率大幅提高。用 HLS 进行相关的设计具有如下优势:

（1） 高效设计。

用 HLS 设计，对算法描述编写，数据类型规格(整数、定点或浮点)定义以及接口(FIFO s AXI4 } AXI4-Lite } AXI4-Stream)约束，可以快速对特定功能的实现;

（2） 灵活的设计方法。

使用者能够用 MATLAB 或 C 语言进行建模并对其进行相关测试，无误后可对整个设计进行优化，然后再综合，最后完成整个设计;

（3） 验证简单。

使用 C/C++测试平台仿真，加速了对 IP 核的验证。

3.1.2 HLS 加速 FPGA 的视觉开发

Vivado HLS 工具中有若干图像处理的视频库函数，能完成 FPGA 对图像处理的加速，大约为 44 个，满足一般情况的使用，如表 3-1 所示。在视频处理中专门设计了 AXI-Stream 的接口函数，添加约束指令将编写的函数算法的参数定义成 AXI 流接口，视频流作为数据传输实现在 FPGA 中加速。HLS 视频库中将传统的图像数据缓存改写了 FPGA 上可以实现的行缓存(Line Buffer)和窗口缓存(Window Buffer)类。

表 3-1 HLS 图像处理视频库函

视频数据建模		AXI4-Stream IO 函数	
Linebuffer Class	Window Class	AXIvideo2Mat	Mat2AXIvideo
OpenCV 接口函数			
cvMat2AXIvideo	AXIvideo2cvMat	cvMat2hlsMat	hlsMat2cvMat
IplImage2AXIvideo	AXIvideo2IplImage	IplImage2hlsMat	hlsMat2IplImage
CvMat2AXIvideo	AXIvideo2CvMat	CvMat2hlsMat	hlsMat2CvMat
视频函数			
AbsDiff	Duplicate	MaxS	Remap
AddS	EqualizeHist	Mean	Resize
AddWeighted	Erode	Merge	Scale
And	FASTX	Min	Set
Avg	Filter2D	MinMaxLoc	Sobel
AvgSdv	GaussianBlur	MinS	Split
Cmp	Harris	Mul	SubRS
CmpS	HoughLines2	Not	SubS
CornerHarris	Integral	PaintMask	Sum
CvtColor	InitUndistortRectifyMap	Range	Threshold
Dilate	Max	Reduce	Zero

OpenCV 在实时视频处理系统中可以有不同的应用方式，设计流程如图 3-1 所示

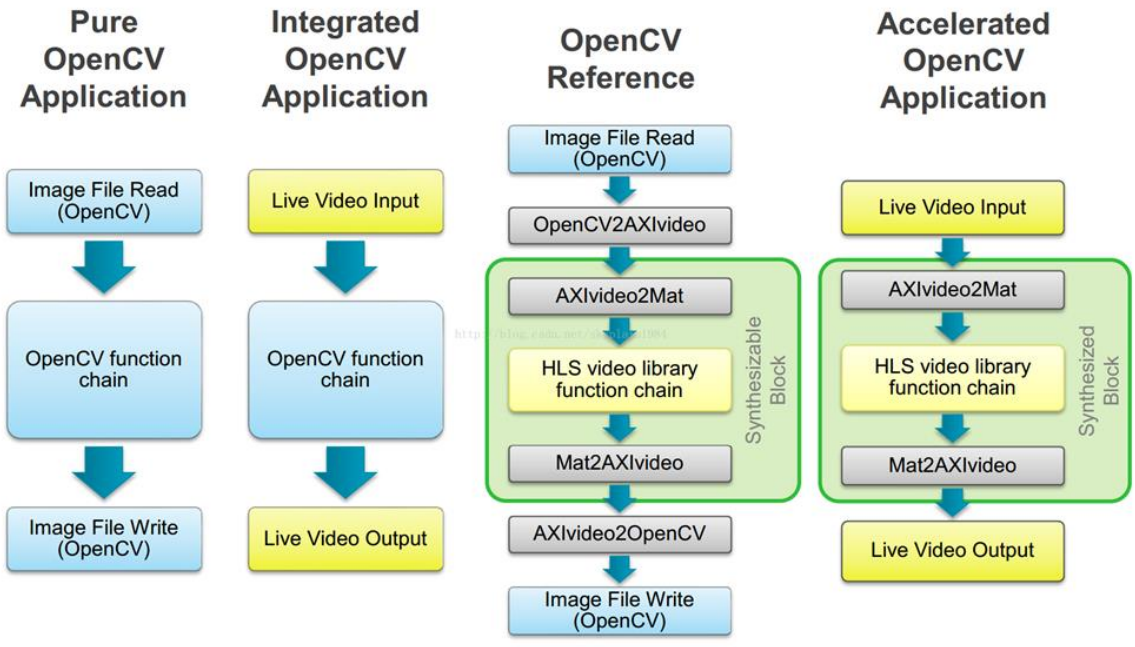


图 3-1 OpenCV 在实时视频处理系统中的应用方式

在图 3-1 的第二张图片中，在 C++的环境下开源的 OpenCV 函数，对其进行编译和仿真，进而生成可执行文件。这些设计可在 Zynq 处理器 ARM 上运行应用;在图 3-1 第四张图中，加速的 OpenCV 应用是利用 Vivado HLS 中相应的图像处理函数来替代 OpenCV 中的函数，并

使用 HLS 中相应的接口约束，最终生成 RTL 级代码并封装成 IP 核，将这模块整合到 Zynq 的可编程逻辑中，实现加速 OpenCV 的性能。

3.1.3 HLS 设计流程

Vivado HLS 的设计流程如图 3-2 所示。

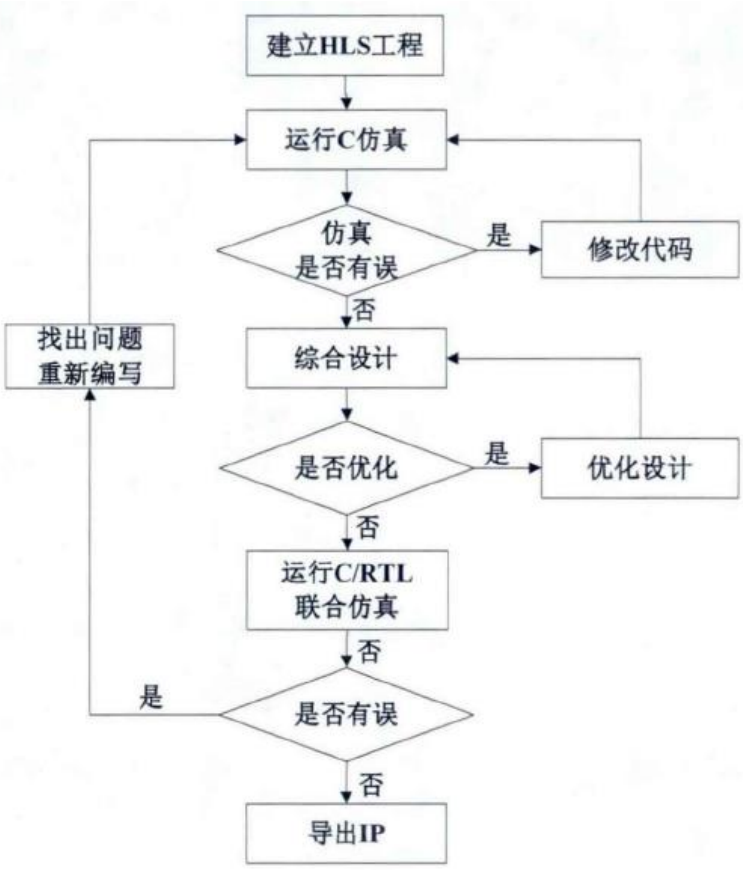


图 3-2 HLS 设计流程图

3.2 HLS 图像处理

3.2.1 灰度化

普通的摄像头采集的图像一般是 RGB（Red，Green，Blue）三通道彩图的结构，即每个像素点的颜色都是由红、绿、蓝 H 种颜色分量构成。本设计中采用的帧间差分法并不需要颜色信息，而且彩色图像包含的信息量要比灰度图像的大很多，所以会增加系统计算量，因而在进行帧间差分法前需要对每帧图像进行灰度化。图像灰度化处理算法是指将彩色图像的 R、G、B 三个分量按照不同的权值进行加权平均处理后变为灰度图像。

根据人体生物学信息原理可知，人眼的视网膜感知神经对绿色信息最为敏感，而对蓝色信息最不敏感，根据这个生物学原理，可对 R、G 、B 三分量做加权平均，获得灰度图像。效果如图 3-3 所示.

$$Gray = R\times 0.3 + G\times 0.59 + B\times 0.11$$

（a）灰度前原图　　（b）灰度后

图 3-3 灰度化效果图

3.2.2 高斯滤波

在采集，传输及处理图像的过程中往往会存在一定程度的噪声干扰，噪声恶化了图像的质量，使得图像模糊，淹没了特征，给图片分析带来困难。图像平滑是一种实用的图像处理技术，能消除图像采集，传输及处理过程中的噪声，高斯平滑处理是一种常用的平滑处理方法。平滑要使用滤波器一般使用线性滤波器，其统一形式如下

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l)$$

其中 h 称为滤波器的核函数也就是权值。

使用 3*3 的高斯核

$$\frac{1}{16}\times\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$

则计算公式如下

$$g(x,y) = \{f(x-1,y-1) + f(x-1,y+1) + f(x+1,y-1) + [f(x-1,y) + f(x,y-1) + f(x+1,y) + f(x,y+1)]*2 + f(x,y)*4\}/16$$

其中，f(x,y)为图像中(x,y)点的灰度值，g(x,y)为该点经过高斯滤波后的值。

在 HLS 中实现用窗口大小为 3*3，标准差采用自动计算策略，效果如图 3-4 所示。

（a）高斯滤波前 （b）高斯滤波后

图 3-4 高斯滤波处理

3.2.3 形态学处理

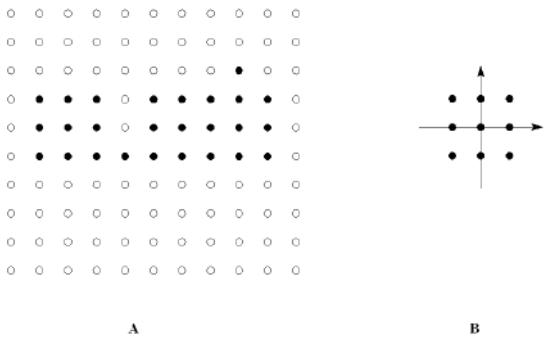
完成图像帧间的差分后，设置阂值，对图像进行分割，这种得到的目的图像中会有空的区域以及非目标区域的一些噪点。形态学对上述存在的问题能有一定的解决能力，其处理方法是把一定形态的结构元素来提取图像中所对应的形状，来实现对图像识别和分析的作用。形态学的操作对图像数据进行简化并保持图像基本形状，而且去除不相干结构。它由一组代数算子组成，基本的形态运算有:膨胀、腐蚀、开运算和闭运算。

(1)腐蚀

腐蚀是一种消除边界点，使边界向内部收缩的过程，可以用来消除小的且异常的目标。用集合论的观点定义为：对于集合 A，把结构元素 B 平移 a 后得到 B_a，若 B_a 依旧包含于 A，则记下 a 点，满足上述条件的 a 点组成的集合称作 A 被 B 腐蚀后的结果，用 $A\Theta B$ 表示

$$A\Theta B=\{a\,|\,B_a\subseteq A\}$$

图 3-5 就是二值化图像的腐蚀过程。



A B

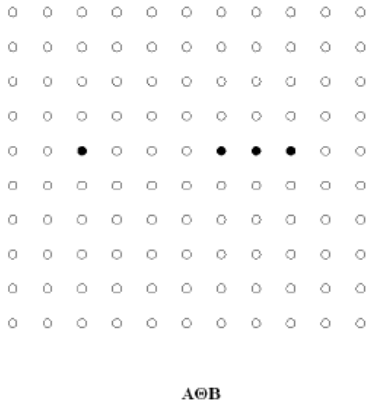


图 3-5 二值化图像的腐蚀过程

(2)膨胀

集合 B 对 A 膨胀，用 A.B 表示，公式表示如下:

$$X=A\oplus B=\{x|B(x)\}\cap A\neq\varnothing\}$$

式中，B 为结构元素，A 为输入的源图像，X 为膨胀后的结果图像。膨胀完成对目标物体内部空区域的填充，使物体的边界对外扩张，将距离近的不同连通区域合并为一个连通区域。图 3-6 就是二值化图像的膨胀过程

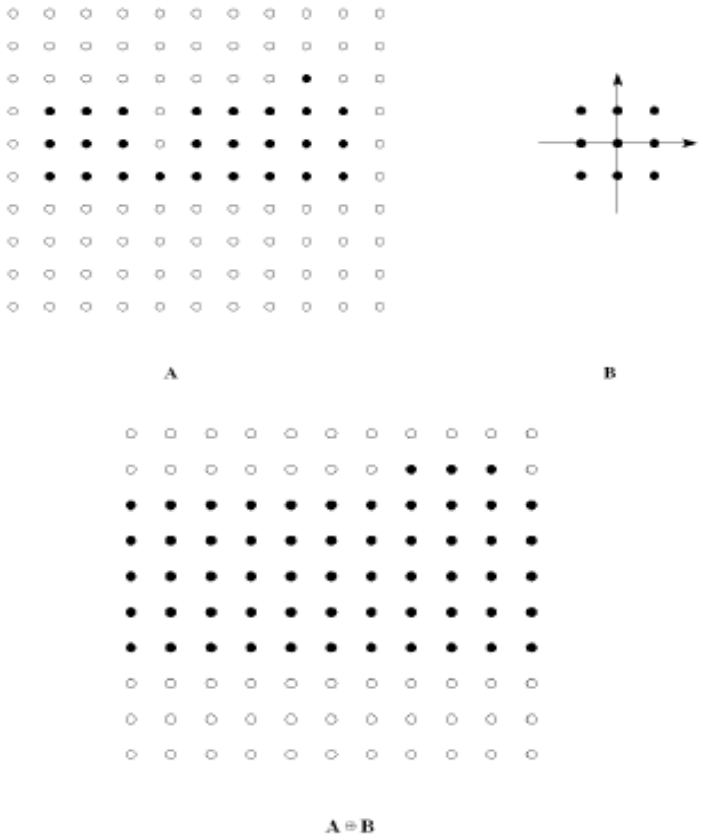


图 3-6 二值化图像的膨胀过程

(3)开运算

集合 B 对集合 A 进行开运算，表示为 A B，公式表示如下:

$AB=(A\ominus B)\oplus B$

开运算就是先做腐蚀操作后膨胀操作的过程。开运算把目标物体之外的毛刺小点消除掉，使物体轮廓变得光滑并保持了总的位置和形状不变。

(4)闭运算

集合 B 对集合 A 进行闭运算，表示为 AB，公式表示如下:

$AB=(A\oplus B)\ominus B$

闭运算是先做膨胀操作后做腐蚀操作的过程，通常是弥合小裂缝，使物体轮廓变得光滑并保持了总的位置和形状不变。

在 HLS 中对图像单独进行腐蚀和膨胀运算，如图 3-7 和 3-8

图 3-7 腐蚀处理效果

图 3-8 膨胀处理效果

根据本系统设计需求，最终采用了形态学处理的开运算对检测到的运动目标进行了处理。

3.2.4 运动检测算法

运动物体检测简单来说就是把视频序列图像中变化的区域给提取出来。但是运动目标的检测也有其相对的难点，例如易受如光照、天气、阴影及障碍物干扰等外界因素影响，使得在对目标检测上存在一定的困难。当前对运动目标检测方法的研究主要可分为帧差法、背景减除法，团块匹配法、光流法和运动能量法等。帧差法、背景差分法和光流法是常用的目标检测算法，其中帧差法和背景差分法来实现的目标检测计算量相对较小，光流法设计难度大而且计算量也大。具体对比分析如下：

（1） 团块匹配法

团块匹配主要应用于运动估计，在视频处理中，图像块匹配的主要是通过在前后两帧视频序列中，找到图像中最相关的部位，并建立他们的联系。这样就可以通过其中一幅图片及

关系信息，还原出另一张图片的信息。这种方法可以在复杂环境或目标跟踪等情况使用，针对遮挡情况有一定优势，但在运动物体检测方面不经常使用。

（2） 光流法

光流法检测运动目标，其基本思想是赋予图像中的每一个像素点一个速度矢量，从而形成了该图像的运动场。图像上的点和三维物体上的点在某一特定的运动时刻是一一对应的，根据各像素点的速度矢量特征对图像进行动态的分析。若图像中不存在运动目标，那么光流矢量在整个图像区域则是连续变化的，而当物体和图像背景中存在相对运动时，运动物体所形成的速度矢量则必然不同于邻域背景的速度矢量，从而将运动物体的位置检测出来。通过计算光流场得到的像素运动向量是由目标和摄像机之间的相对运动产生的。因此该类检测方法可以适用于摄像机静止和运动两种场合。但是光流场的计算过于复杂，迭代时间长，运算开销大，而且在实际情况中，由于光线等因素的影响，目标在运动时，其表面的亮度并不是保持不变的，这就不满足光流基本约束方程的假设前提，导致计算会出现很大的误差，这也使得这种方法的实时性和实用性都比较差，很难满足实时运动目标检测的要求。

（3） 运动能量法

运动能量法适合于复杂变化的环境，能消除背景中振动的像素，使按某一方向运动的对象更加突出地显现出来，但运动能量法不能精确地分割出对象。

（4） 背景差分法

背景差分法是一种有效的运动对象检测算法，基本思想是利用背景的参数模型来近似背景图像的像素值，将当前帧与背景图像进行差分比较实现对运动区域的检测，其中区别较大的像素区域被认为是运动区域，而区别较小的像素区域被认为是背景区域。背景差分法必须要有背景图像，并且背景图像必须是随着光照或外部环境的变化而实时更新的，因此背景减除法的关键是背景建模及其更新。在运用背景差分法时需要有一定的限制：要求前景(运动物体)象素的灰度值和背景象素的灰度值存在一定的差别。

（5） 帧间差分法

帧差法是最为常用的运动目标检测和分割方法之一，基本原理就是在图像序列相邻两帧或三帧间采用基于像素的时间差分通过闭值化来提取出图像中的运动区域。首先，将相邻帧图像对应像素值相减得到差分图像，然后对差分图像二值化，在环境亮度变化不大的情况下，如果对应像素值变化小于事先确定的阂值时，可以认为此处为背景像素:如果图像区域的像素值变化很大，可以认为这是由于图像中运动物体引起的，将这些区域标记为前景像素，利用标记的像素区域可以确定运动目标在图像中的位置。由于相邻两帧间的时间间隔非常短，用

前一帧图像作为当前帧的背景模型具有较好的实时性，其背景不积累，且更新速度快、算法简单、计算量小。算法的不足在于对环境噪声较为敏感，阈值的选择相当关键，选择过低不足以抑制图像中的噪声，过高则忽略了图像中有用的变化。对于比较大的、颜色一致的运动目标，有可能在目标内部产生空洞，无法完整地提取运动目标。

相邻帧间差分法法算法实现简单，程序设计复杂度低，运行速度快；动态环境自适应强，对场景光线变化不敏感。但有时容易出现“双影”现象。三帧差法可在一定程度上消除相邻帧间差分法的“双影”现象，因此本系统在运动物体检测算法上使用三帧差分法来实现。

第四章 运动目标检测系统设计

4.1 开发流程概述

本系统基于 Zynq 平台的 ARM+FPGA 硬件结构采用软硬件协同设计的方法来实现运动物体的检测。其中，FPGA 部分完成的图像采集和预处理，采用三帧差分法进行运动目标的检测，并实现 VGA 显示，ARM 部分对 VDMA IP 核进行配置，并控制 DDR 存储器中图像的存储。硬件工程使用 Vivado 2017.2 工具设计完成，其中运动目标检测 IP 由 Vivado HLS 2017.2 设计实现，并由 SDK 制作系统启动文件 BOOT.BIN，拷贝到 SD 卡中，即可完成整个系统流程。开发流程如图 4-1 所示。

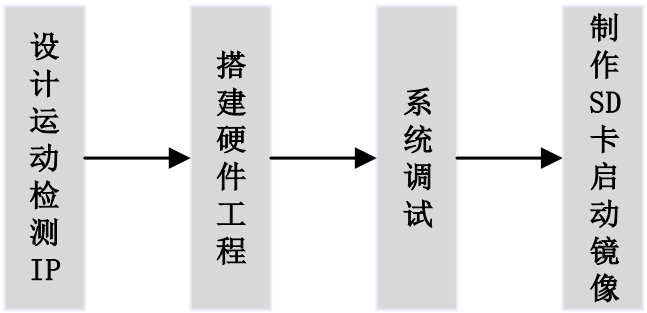


图 4-1 系统开发流程

4.2 硬件工程设计

4.2.1 OV7670 寄存器配置

OV7670 图像传感器在工作之前需要进行初始化配置，其配置接口为标准的 SCCB(Serial Camera Control Bus)，通过该总线的配置是摄像头工作在希望的状态模式下。OV7670 的 SCCB 总线，实际上就是常用的 I2C 通信总线，只不过 Omni Vision 针对 Camera 的控制，专门提供了一个术语 SCCB 总线，如 I2C 总线一样，在使用时都必须按照严格的时序规范，它由 SIO_D 和 SIO_C 两条总线构成，一个负责传输硬件地址和数据，一个负责为传输提供时钟，两条总线都是串行的。在总线启动之前，SIO_C 会一直保持高电平，当检测到 SIO_D 出现一个下降沿，就会启动传输；在传输阶段，每当 SIO_C 产生一个正脉冲，SIO_D 总线上就会一次性传输 8 位数据；传输停止是由 SIO_C 高电平和 SIO_D 上升沿共同决定的。SCCB 的整体时序如图 4-2 所示。

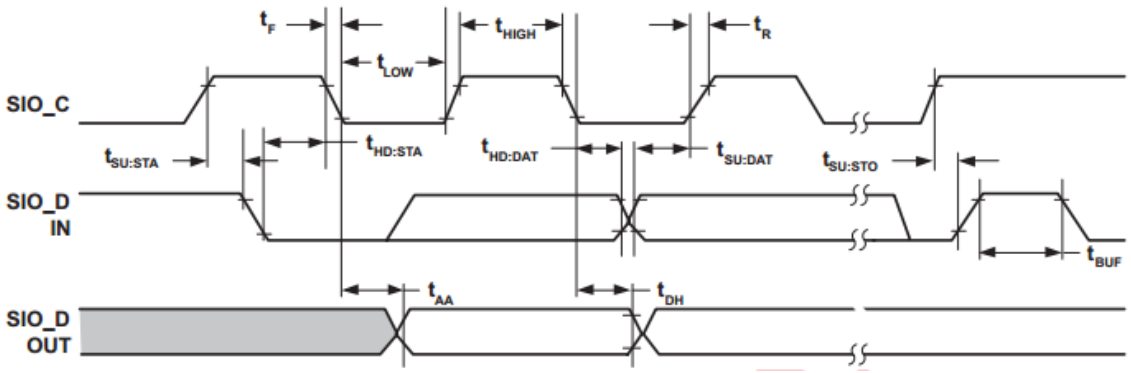


图 4-2 SCCB 总线的整体时序

由于主要是通过 SCCB 总线对摄像头寄存器进行写操作，因此这里主要研究写时序，写时序如图 4-3 所示，其中 IP address 如果进行写操作则为 0x42，sub-address 是具体的寄存器地址，write data 为要写入的寄存器数据。写入数据格式为每组 9 位，前 8 位是数据，最后“X”是从设备的回应。

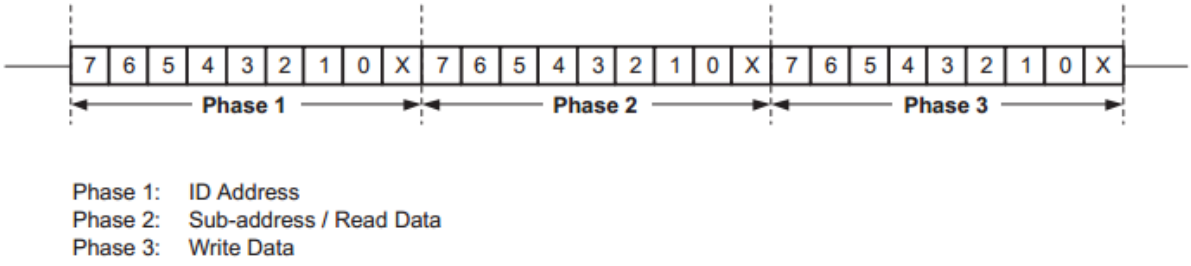


图 4-3 SCCB 写时序

在 FPGA 中对 OV7670 的初始化配置，其实就是实现 SCCB 总线控制器。通过对写时序图的分析，使用 Verilog 生成对应的波形，从而实现 SCCB 的写操作，将实现代码封装成 IP，其示意图如图 4-4 所示。

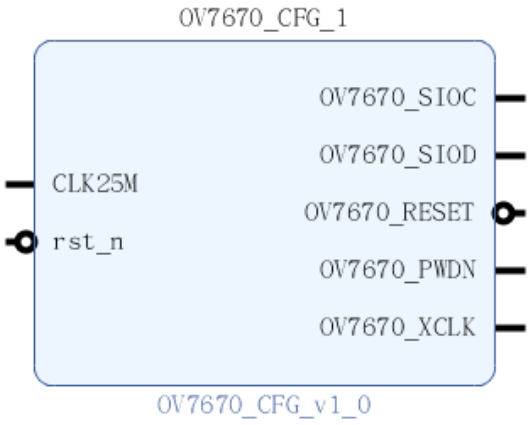


图 4-4 OV7670 配置 IP

OV7670 配置模块端口说明如表 4-1 所示：

表 4-1 OV7670 配置模块端口说明

端口	端口类型	说明
OV7670_SIOC	output	串行时钟线
OV7670_SIOD	inout	串行数据线
OV7670_RESET	output	输出 0，正常模式
OV7670_PWDN	output	输出 0，正常模式
OV7670_XCLK	output	CMOS 输入的时钟信号
rst_n	input	复位信号，低电平有效
CLK25M	input	配置模块时钟信号

4.2.2 图像采集模块

在初始化 OV7670 时，将 CMOS 摄像头配置成了 RGB565 模式，RGB565 的时序图如图 4-5 所示。

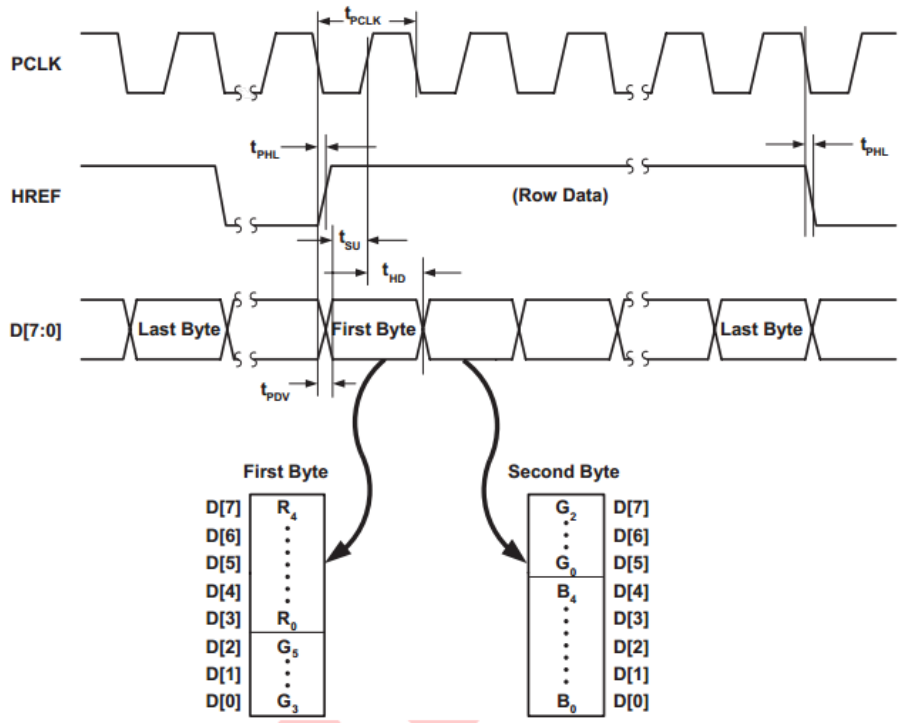


图 4-5 RGB565 时序

当帧同步信号 VSYNC 出现有效边沿后，在 HREF 为高电平时，第一个 PCLK 上升沿读取第一个字节数据（D7~D0）。但是这个字节并不代表第一个像素，而是第一个像素的 R[4:0]以及 G[5:3]，第二个 PCLK 上升沿读取的字节则是读取的第一个像素的 G[2:0]以及 B[4:0]。当第二个 PCLK 上升沿到来时，将这两个字节组成一个完整的像素，就得到了第一个像素。以此类推，采集一行数据（640×2），就得到 640 个像素值，当采集完 480 行的时候，就得到了一帧数据的采集。

由于摄像头传入的信号使用的协议并非 AXI4-Stream，而 VDMA 却需要接收 AXI4-Stream 类型的数据，所以需要实现两者之间的转换。在这里我们采用的依元素公司提供的 IP，实现二者之间的转换。输入只需要提供四路信号：行同步、场同步、像素时钟、像素数据。视频数据采集模块的顶层结构图如下所示。

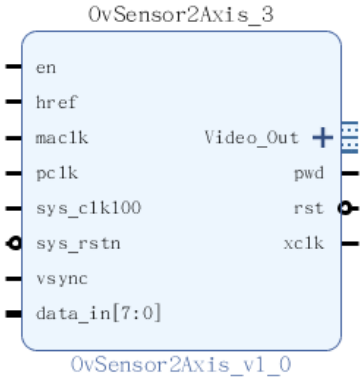


图 4-6 视频数据采集模块的顶层结构

4.2.3 AXI 接口

Zynq 平台是新一代全面可编程片上系统，它是 Xilinx 公司推出的首款组合了一个双核 ARM Cortex-A9 处理器和一个传统的 FPGA 逻辑部件在单芯片内紧密结合的架构体系。高效的片内高性能处理器 ARM 与 FPGA 数据交互通路是设计的关键部分，而 Zynq 中 PS 与 PL 之间数据交互接口是 AXI 接口。

PS 与 PL 主要包括下面两种接口类型：

（1） 功能接口

包括 AXI 接口、EMIO 接口、DMA 流控制、中断、时钟和调试接口。在实际工程应用中，完成 FPGA 模块设计时，依据需求进行选择功能接口，实现 PL 与 PS 的数据交互。

（2） 配置接口

包括 SEU、PCAP、配置状态信号以及 Program/Done/Init 信号。这些配置接口固定在可编程逻辑部分的特定逻辑中，给 PL 部分提供了对 PS 部分的控制能力。

本文主要介绍 AXI 接口，Zynq 中的 AXI 接口一共有九个，包括如下类型：

- AXI_GP 接口（4 个）：通用 AXI 接口，包括两个 32 位主设备接口与两个 32 位从设备接口，用该接口可以访问 PS 中的片内外设。
- AXI_HP 接口（4 个）：是高性能/带宽的 AXI 接口，PL 模块作为主设备连接。主要用于 PL 访问 PS 上的存储器（DDR 和 On-Chip RAM）。

- AXI_ACP 接口（4 个）：ARM 多核架构下定义的一种接口，为加速器一致性端口，用来管理 DMA 之类的不带缓存的 AXI 外设，PS 端是从接口。

AXI(Advanced eXtensible Interface)协议采用的是一种 READY，VALID 握手通信机制，简单来说主从双方进行数据通信前，有一个握手的过程。传输源产生 VALID 信号来指明何时数据或控制信息有效，而目地源产生 READY 信号来指明已经准备好接受数据或控制信息。数据传输发生在 READY 和 VALID 信号同时为高的时候。

AXI 协议将读地址通道，读数据通道，写地址通道，写数据通道，写响应通道分开，各自通道都有自己的握手协议，每个通道互不干扰却又彼此依赖。Zynq 内部有非常多的设备需要连接。如 CPU 要访问 DDR 和 OCM，PL 也要访问 DDR 和 OCM，还有 DMA 也是需要访问内存的，这些内部设备都有 AXI 接口，并且都使用的是 AXI 协议，这使数据具有完全互连通信的能力。AXI_HP 接口为 PL 访问 PS 上的存储器（DDR 和 OCM）提供了高带宽的数据通道，一般完成视频处理时，高清图像数据的传输。

4.2.4 AXI VDMA 配置

AXI VDMA(AXI Video Direct Memory)IP 核是 Xilinx 公司为解决类似视频图像这样的大数据在外部存储器与内部 FPGA 之间的高速数据传输问题而专口设计的，用以提供 DDR 与 AXI4 视频流类型目标外设之间的直接内存访问，在本设计中发挥着重要的作用，因此对其进行研究非常有必要。

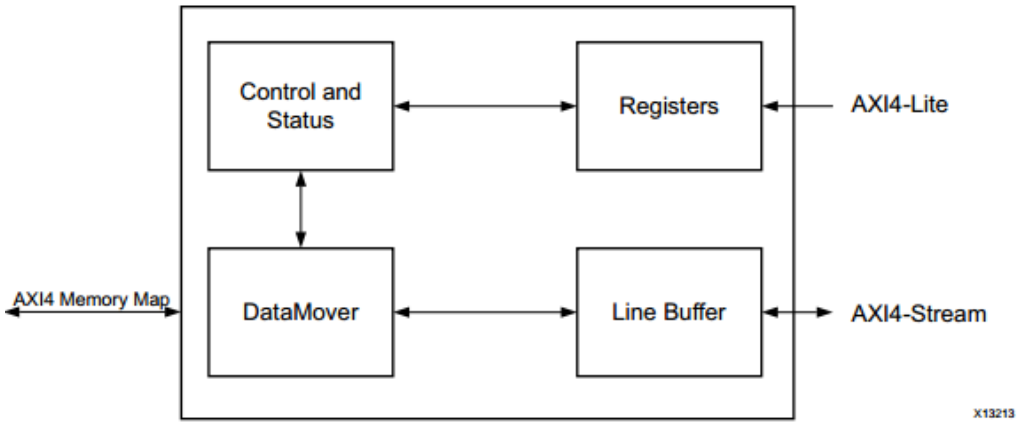


图 4-7 VDMA 内部结构图

AXI VDMA 的主要接口有三个，分别为 AXI4 Memory Map、AXI4-Lite、AXI-Stream。其中 AXI4-Lite 接口用于读写 VDMA 内部寄存器，从而实现对 VDMA 的状态的控制，包括 VDMA 的注册，访问和配置是通过 AXI4-Lite 从接口来实现的。AXI-Stream 接口分为两个：一个是 AXI MM2S 读通道接口，另一个是 AXI S2MM 写通道接口。

下面以垂直大小为 5 行，每行 16 字节，跨度为 32 字节的条件为例，简析一下 m_axi_mm2s 接口的时序，由于 m_axi_s2mm 接口类似，不做详述。

MM2S 接口的时序图如图 4-8 所示。

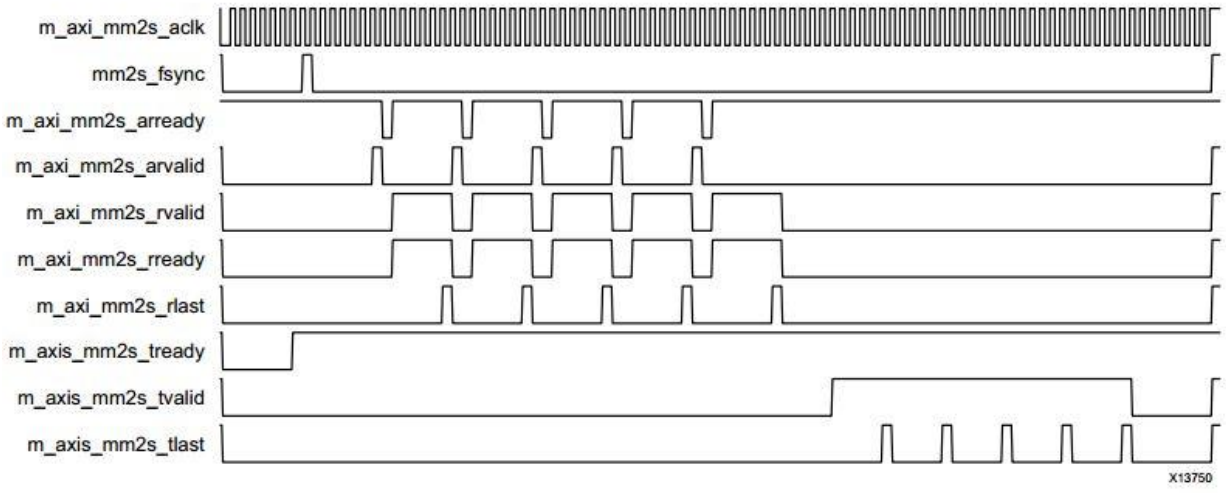


图 4-8 MM2S 接口时序图

在 mm2s_fsync 接口接收到信号之后，AXI VDMA IP 核就会立即发出 m_axi_mm2s_arvalid 信号，与此同时在 m_axi_mm2s_araddr 信号线上设置起始地址。其中 m_axi_mm2s_arvalid 信号需要被发送五次，这是为了接收 Vertical Size 的五行数据。在数据发送端，被读取的数据存储在行缓存区中，然后被传送到视频流端口，同时发送 m_axis_mm2s_tvalid 信号。每一行传输结束后，都需要发送 m_axis_mm2s_tlast 信号。

AXI VDMA 要被映射在不可缓存的内存空间，本系统所涉及到的几个主要寄存器的地址映射如下表所示。

表 4-2 VDMA 主要寄存器地址映射

地址空间偏移	名称	注释
00h	MM2S_VDMACR	MM2S VDMA 注册控制器
5Ch to 98h	MM2S_START_ADDRESS	MM2S 起始地址
54h	MM2S_HSIZE	MM2S 水平尺寸寄存器
50h	MM2S_VSIZE	MM2S 垂直尺寸寄存器
30h	S2MM_VDMACR	S2MM VDMA 注册控制器
ACh to E8h	S2MM_START_ADDRESS	S2MM 起始地址
A4h	S2MM_HSIZE	S2MM 水平尺寸寄存器
A0h	S2MM_VSIZE	S2MM 垂直尺寸寄存器

在 Vivado 中连接添加 VDMA IP，添加完成后还要对它的一些参数进行设置。其中 Address Width 的设置为 32 位，Frame Buffer 的设置最少为 3 。使能读通道 MM2S 的 Stream Data Width 参数时，需要设置成 32bits，这就是为了完成 VDMA 传输 32 位的图像数据到自定义 IP 核中。配置界面如图 4-9 所示。

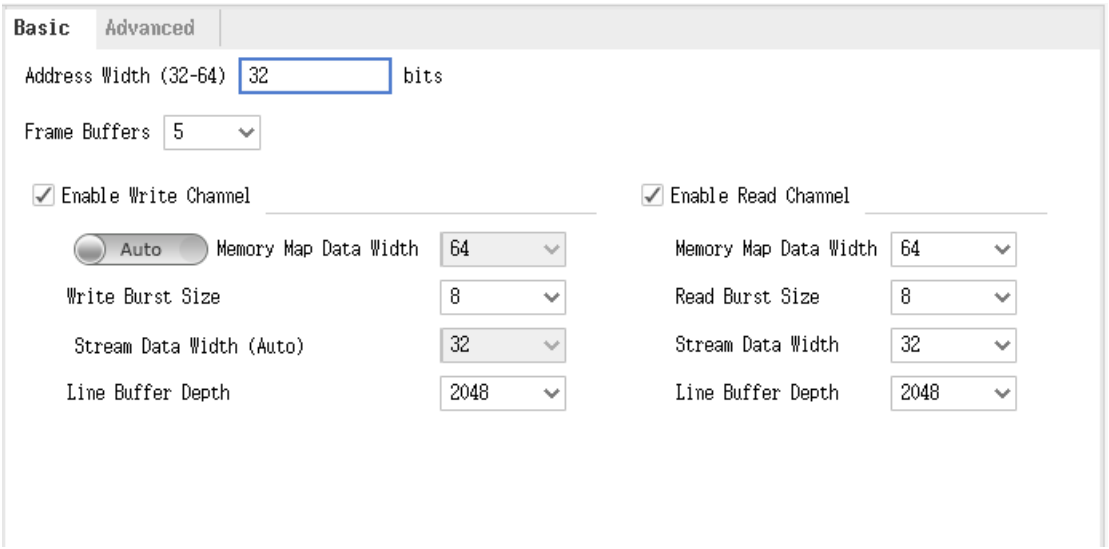


图 4-9 VDMA 配置（a）

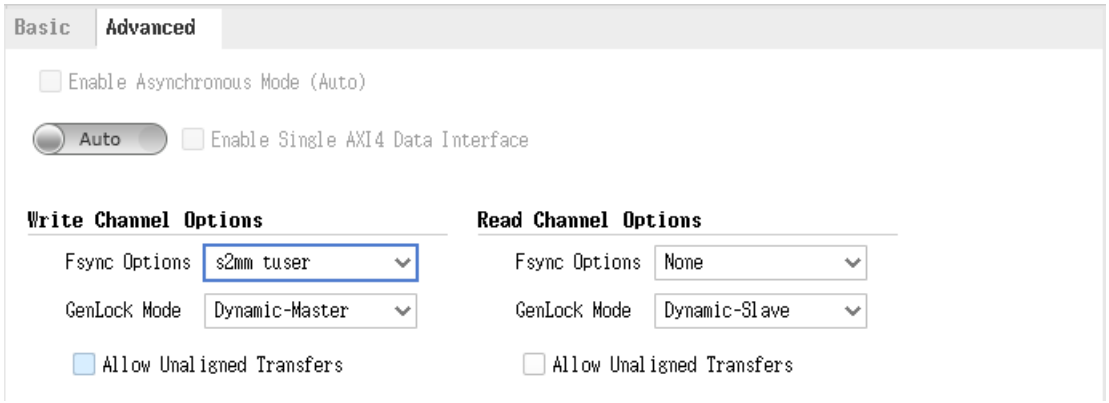


图 4-9 VDMA 配置（b）

4.2.5 三帧差分法设计运动目标检测 IP

本系统中采用三帧差分法来完成运动目标的检测，处理流程如图 4-10 所示。

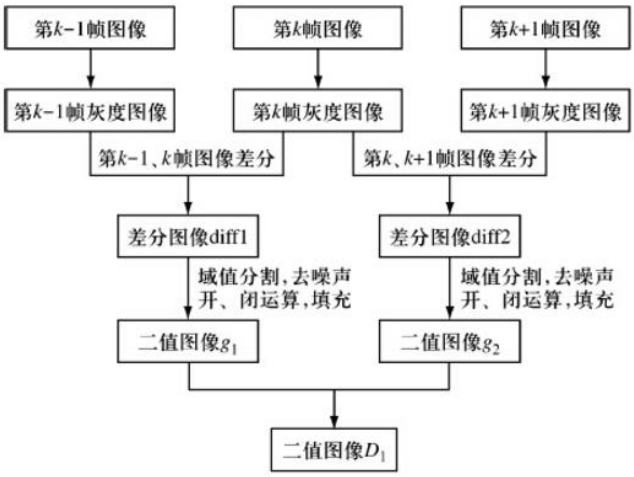


图 4-10 三帧差分法流程图

Vivado HLS 完成上述的算法流程需要经过特定的步骤。第一步先创建 HLS 工程，选择配置所需的时钟频率 100Mhz，以及与 ES-288 平台相对应的器件。在完成工程创建后，需要正式的添加算法程序了，其中关键的算法编写包括如下几个部分:

（1） 数据类型的声明

本系统共涉及 4 种图像格式，分别声明为：

```
typedef hls::stream<ap_axiu<32,1,1,1> > AXI_STREAM_IN;
typedef hls::stream<ap_axiu<8,1,1,1> > AXI_STREAM_OUT;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3>  RGB_IMAGE;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1>  GRAY_IMAGE;
```

AXI_STREAM_IN 是对输入图像数据类型的声明，AXI_STREAM_OUT 是对输出图像数据类型的声明，最后会综合成视频流接口；RGB_IMAGE 声明的是三通道整型数据的图像；GRAY_IMAGE 声明的是单通道整型数据的图像；

（2） 主函数的声明

```
void movedetection(AXI_STREAM_IN& input0_axi,AXI_STREAM_IN& input1_axi,
AXI_STREAM_OUT& output_axi,int rows,int cols,int TH);
```

主函数声明完后，在主体函数中编写上述的算法，按照 HLS 的规则严格进行程序的设计，以及添加相应的约束语句，最后将上述代码中的 input0_axi、input1_axi、output_axi、rows、cols 和 TH 综合生成 IP 核的 I/O 口。

（3） 仿真测试

在 HLS 的 TestBench 中，调用顶层主函数来对运动目标进行处理，测试采用两张像素进行了位移的图片来代替我们运动物体的两帧视频图像，处理效果如图 4-11 所示。

（a）输入图像 1

（b）输入图像 2（图像 1 下移两行像素）

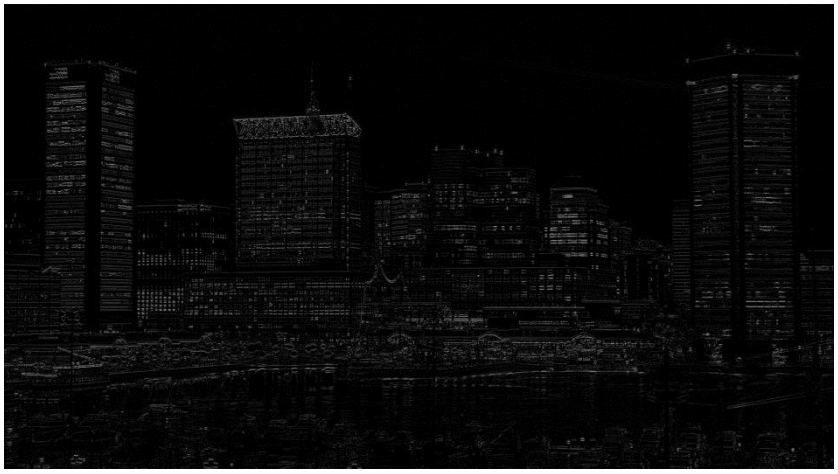


图 4-11 运动检测 IP 仿真结果图

通过仿真结果可以看出，运动物体检测算法能够检测出运动物体，并保留运动物体的轮廓边缘信息。

（4） IP 核的综合封装

仿真无误后，对整个工程进行编译并综合，使用者能够在 Console 控制台看到整个过程中综合的相关信息。最后 HLS 会打印出相关的综合报告，其中包含时序信息和器件资源利用率估计，如图 4-12 和图 4-13 所示。综合成功后，即可将设计封装为 IP 核。选择 Solution/Export RTL 命令，然后在选择导出的方式为 IP Catalog，就能得到 IP 核封装文件，这样就能被 Vivado 识别。到此完成了运动检测 IP 核的所有设计过程，下一步将此 IP 核文件添加到硬件工程中。

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.73	1.25

图 4-12 运动目标检测 IP 核时序信息

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	17
FIFO	0	-	385	1756
Instance	12	9	3629	5227
Memory	-	-	-	-
Multiplexer	-	-	-	108
Register	-	-	55	-
Total	12	9	4069	7108
Available	120	80	35200	17600
Utilization (%)	10	11	11	40

图 4-13 运动目标检测 IP 资源利用率

在表中可看出，时钟周期设置为 10ns，时钟周期的不确定性是 1.25ns，时钟周期不确定性是 FPGA 中逻辑器件布线造成的，估计时钟周期（最坏延迟）是 8.73ns，其要求条件为 Target>Estimated+Uncertainty。在表中可以看到，完成这些图像的操作，需要大约 4000 个 flip-flops,7100 个 LUT 等一些单元，最后一列也可以看到这些单元的利用率。

4.2.6 硬件工程

Vivado HLS 工具完成了三帧差分法提取运动目标，封装成可以移植的 IP 核，并在 Vivado 中找到相应路径下的 IP 核文件，然后调用，如图 4-14 所示。

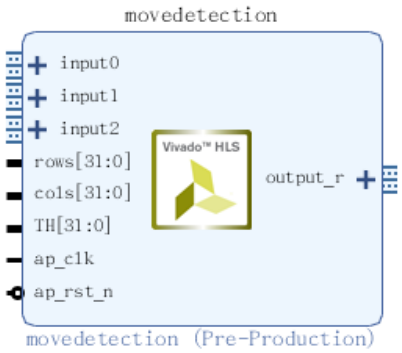


图 4-14 运动目标检测 IP 核结构

该运动目标检测 IP 中，input0，input1，input2 表示连续输入的三帧，rows 和 cols 分别表示图像的行像素和列像素个数，TH 表示三帧差分法的阈值，output 表示输出的流信号，ap_clk 和 ap_rst_n 分别为时钟和复位信号。

在 Vivado 中搭建顶层系统，可根据图形界面化，把运动目标检测 IP 核、VDMA、OV7670 配置 IP 核、视频采集 IP、VGA 显示 IP 核以及 AXI Interconnect 连接成完整的视频通路，对需要配置的模块进行配置，最终 Vivado 建立起的硬件工程系统框图如图 4-15 所示。

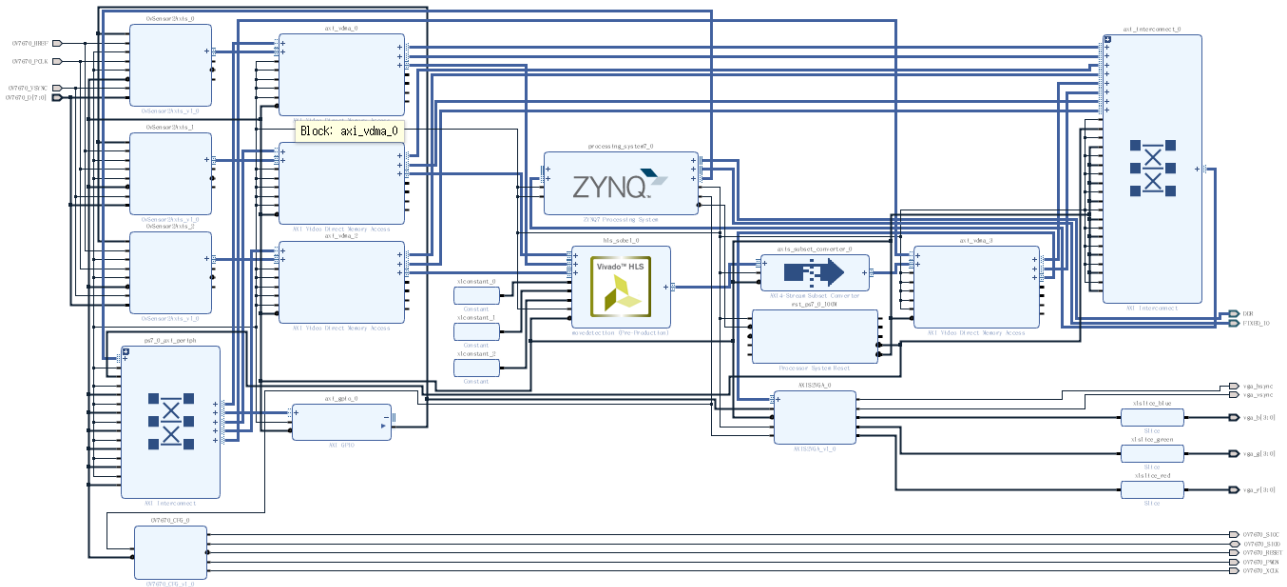


图 4-15 硬件工程总体结构框图

如上图所示，vdma_0 控制采集的视频数据存储到 DDR 中，然后 vdma_0，vdma_1，vdma_2，三个 VDMA 完成图像数据传输运动目标检测 IP 核，即通过 AXI Interconnect 来实现高带宽数据通路 S_AXI_HP0 接口，读取存储器 DDR 中的图像数据，三个 VDMA 读通道 M_AXIS_MM2S 读取 DDR 中的三帧图像数据，传输给运动目标检测 IP 核，在完成图像处理后，经 IP 核的 output 端口将数据传输给 vdma_4 的写通道 M_AXIS_S2MM，vdma_4 再利用 AXI Interconnect 连接到高带宽数据通路 S_AXI_HP0 接口，把经过处理后的图像数据传回到存储器 DDR 中，这完成的是 vdma_4 的写操作。运动目标检测 IP 核与 VDMA 的连接图如下所示。

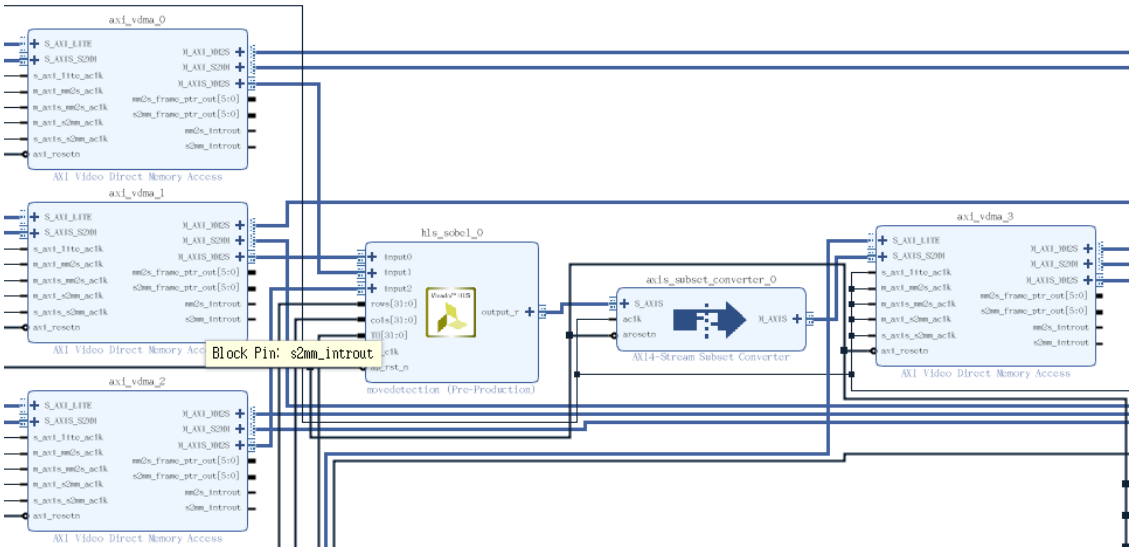


图 4-16 运动目标检测 IP 核与 VDMA 的连接关系

在软件工程中需要对硬件工程里的 IP 核进行控制操作，牵涉到内存中的地址空间，因此要对各 IP 核分配有效地址空间。在 Vivado 中选择自动分配地址，会对每个模块分配地址以及相应的空间大小，如图 4-17 所示。

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_vdma_0	S_AXI_LITE	Reg	0x4300_0000	64K	0x4300_FFFF
axi_vdma_1	S_AXI_LITE	Reg	0x4301_0000	64K	0x4301_FFFF
axi_vdma_2	S_AXI_LITE	Reg	0x4302_0000	64K	0x4302_FFFF
axi_vdma_3	S_AXI_LITE	Reg	0x4303_0000	64K	0x4303_FFFF
axi_vdma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
axi_vdma_1					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
axi_vdma_2					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
axi_vdma_3					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF

图 4-17 IP 核地址分配

4.3 软件工程设计

4.3.1 SDK 程序编写

VDMA 的用途是完成 FPGA 与 DDR 内存之间的高速 DMA 操作的。VDMA 的配置就是对 VDMA 的端口寄存器进行读写，进而完成 DMA 传输的启动与停止等功能。其中 VDMA 有 M2SS（读通道）和 S2MM（写通道）两个通道，分别用于控制 DDR 到 FPGA 和 FPGA 到 DDR 的双向通路，并对读写通道进行配置。

在 SDK 中对 VDMA 的操作可以分为以下几步：

（1） 复位 VDMA

（2） 设置 VDMA，主要是针对寄存器进行操作，包括开辟的帧缓存的数量、起始地址、大小等，使 VDMA 工作在需要的模式下

（3） 开启 VDMA，开启读写通道。

初始化 VDMA 的部分代码如下所示：

```
/***********************S2MM*****************************
Xil_Out32((VDMA_BASEADDR0 + 0x030), 0x00000004);
Xil_Out32((VDMA_BASEADDR0 + 0x030), 0x0000008b);
Xil_Out32((VDMA_BASEADDR0 + 0x0AC), VIDEO_BASEADDR0);
Xil_Out32((VDMA_BASEADDR0 + 0x0B0), VIDEO_BASEADDR1);
Xil_Out32((VDMA_BASEADDR0 + 0x0B4), VIDEO_BASEADDR2);
Xil_Out32((VDMA_BASEADDR0 + 0x0A8), (HL*4));
Xil_Out32((VDMA_BASEADDR0 + 0x0A4), (HL*4));
Xil_Out32((VDMA_BASEADDR0 + 0x0A0), (VL));
/***********************M2SS*****************************
Xil_Out32((VDMA_BASEADDR0 + 0x000), 0x0000004);
Xil_Out32((VDMA_BASEADDR0 + 0x000), 0x000108b);
Xil_Out32((VDMA_BASEADDR0 + 0x05c), VIDEO_BASEADDR0);
Xil_Out32((VDMA_BASEADDR0 + 0x060), VIDEO_BASEADDR1);
Xil_Out32((VDMA_BASEADDR0 + 0x064), VIDEO_BASEADDR2);
Xil_Out32((VDMA_BASEADDR0 + 0x058), (HL*4));
Xil_Out32((VDMA_BASEADDR0 + 0x054), (HL*4));
Xil_Out32((VDMA_BASEADDR0 + 0x050), VL);
```

因为本设计中摄像头采集的图像大小均是 640×480 个像素，所以 HL 取 640，VL 取 480，又因为 MM2S_HSIZE 与 S2MM_HSIZE 的单位为 Bytes，而传输图像的格式占用是 32bits，所的将 MM2S_HSIZE 和 S2MM_HSIZE 的值应设定为 HL*4。

4.3.2 制作启动镜像

Zynq 有两种启动模式，一个是 JTAG 来启动,另一个是从 BootROM 启动， JTAG 启动模式主要是应用在在线调试中，一旦掉电，程序也就丢失了。在没有外部 JTAG 的情况下，处理器系统（PS）与可编程逻辑（PL）都必须依靠 PS 来完成芯片的初始化配置，这里我们主要来讨论 BootROM 来进行启动。

ZYNQ 的 BootROM 启动配置分多级进行的，通常按如下三个阶段进行：

（1） 阶段 0：该阶段被称为 BootRom 过程，Zynq 中的一段 ROM 中固化了一个无法修改的程序，处理器在上电或者热启动时会自动的执行这部分代码。它将基本的外设进行初始化，可以将外设中的 Boot.bin 代码加载到 OCM 中。

（2） 阶段 1：启动加载 FSBL，配置完 PS 部分，再去配置 PL 部分，最后还可以加载阶段 2 的代码

（3） 阶段 2：这阶段通常可以是用户的 PS 端的设计代码，当然也可以是第二阶段的启动加载器（SSBL），这个阶段可以完全由用户控制，是可选的。

本系统采用了 SD 卡启动的方式，因此需要制作 BOOT.BIN 文件，BOOT.BIN 文件的组成如下图所示。

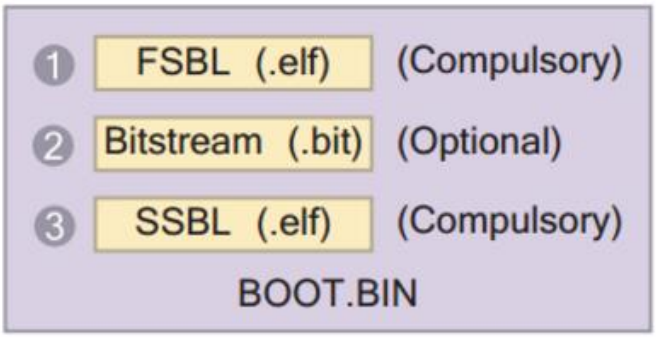


图 4-18 BOOT.BIN 文件组成

FSBL.elf 文件需要在 SDK 软件中生成，在 SDK 中点击菜单 File->New->Application Project。项目名 FSBL，OS 平台 standalone，语言为 C，点击 Next，Templates 选项选择 ZynqFSBL。点击 Finish，SDK 会自动编译 FSBL 代码，并在工程 Debug 下生成 FSBL.elf 目标文件。.bit 文件为 FPGA 的配置文件，.elf 文件为裸跑程序。

在得到FSBL.elf，.bit 文件和裸跑程序后就可以制作 BOOT.BIN 启动镜像。点击菜单 Xilinx Tools > Create Zynq Boot Image，首先设置 BIF 文件路径。BIF 文件用于指定当前 BOOT.BIN 制作过程中用到的三个文件的路径，相当于一个项目配置文件,设置好后按顺序添加三个文件,如图 4-19 所示，

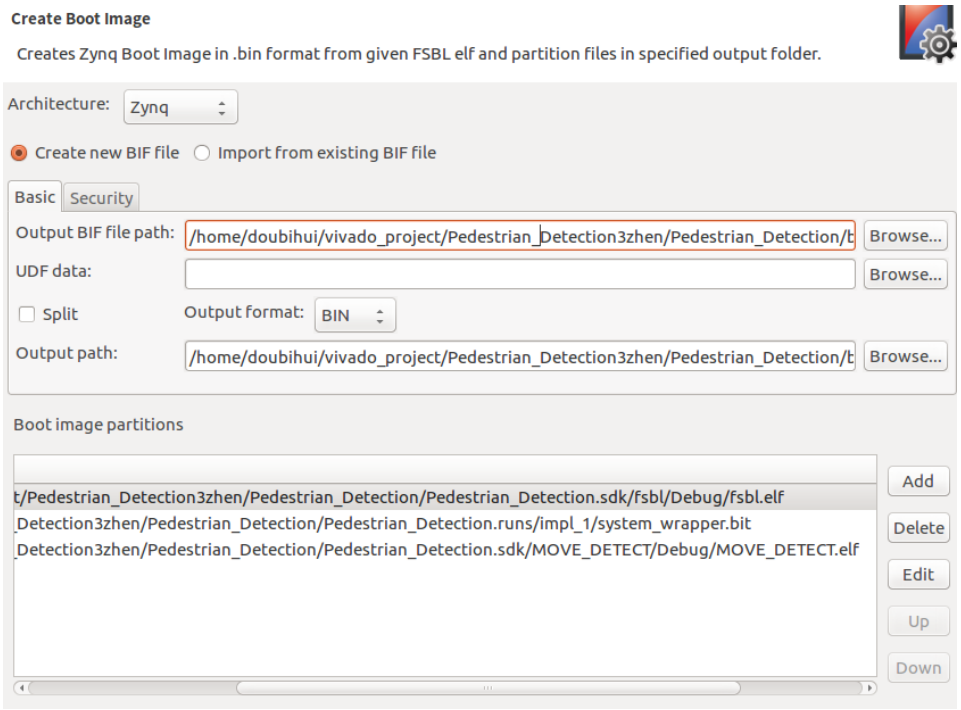


图 4-19 BOOT.BIN 文件制作过程

添加完毕后，点击 Create Image 创建 BOOT.BIN,会在输出路径生成相应文件，将 BOOT.BIN 文件拷贝到 SD 卡中，既可以启动运动检测目标系统。

第五章 系统调试与结果分析

5.1 系统平台搭建

将拷贝了 BOOT.BIN 文件的 SD 卡插入 ESS-288 开发板，并设置成 SD 卡启动，将 OV7670 摄像头和显示器连接至开发板，其中显示与开发板的连接采用了 Pmod VGA。连接完成后，整个硬件平台如图 5-1 所示。

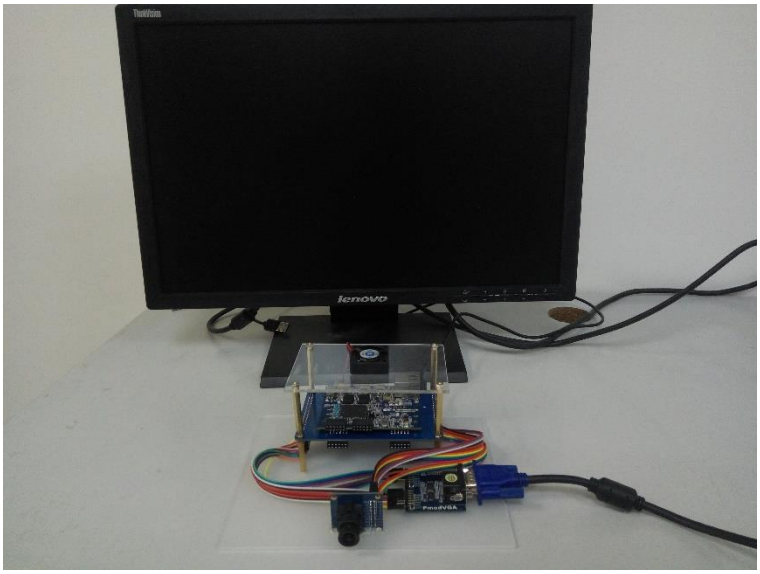


图 5-1 总体硬件系统平台

对基于 Zynq 的运动目标检测系统进行整体测试，通过对手势移动进行简单测试，测试结果如图 5-2 所示：

(a)手势运动识别

(b)行人运动识别

图 5-2 整体系统测试

5.2 器件资源利用率

硬件工程综合完成后，Vivado 给出了相关的报告信息，其器件资源利用率报告如图 5-3 所示。

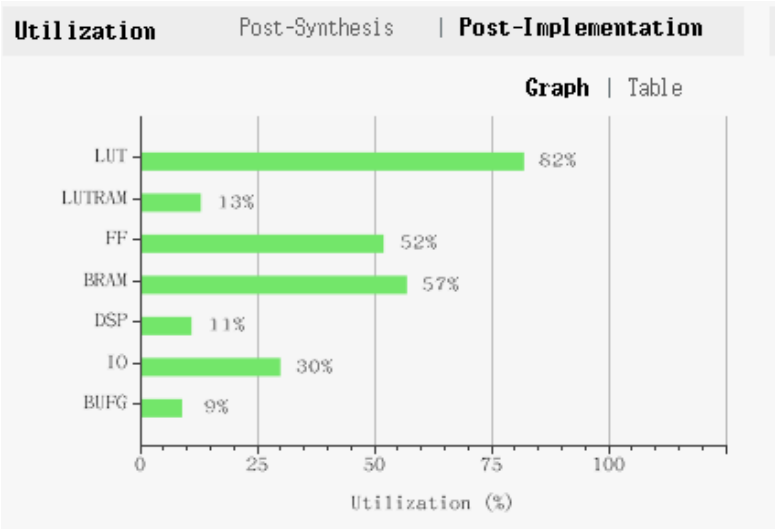


图 5-3 器件资源利用率

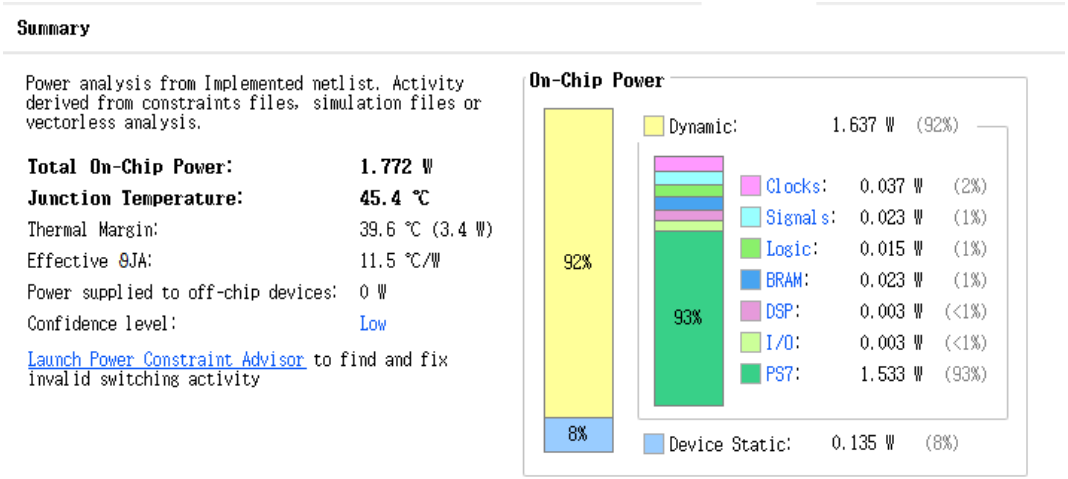


图 5-4 系统功耗综合评估

同时 Vivado 也给出了系统功耗综合评估，如图 5-4 所示，上图中显示运动检测系统的功耗仅为 1.772W，相比于其他处理器的功耗要低，由此可见 Zynq 这种结构的平台，不单单具备高性能的计算能力，还具备构建复杂系统的能力，对一些特殊的工程应用具有低成本、低功耗、体积小、灵活　性强的优势与特点。

5.3 遇到的主要问题及解决方法

在完成基于 Zynq 的运动目标检测系统设计过程中，遇到了一些问题，主要问题阐述如下：

（1） VGA 显示器输出杂乱无章的斑点

问题描述：在 Vivado 中搭建了以 VDMA 为桥梁来连接 PL 与 PS 的完整视频流通路，把图像数据传输到 IP 核后，处理后的图像显示成杂乱无章的图像斑点。

问题分析：将没有经过运动检测 IP 的图像来供显示，显示器可以正常显示采集的图像，出现杂乱无章的图像斑点以下几种可能，输入至运动检测 IP 核的两幅图像没有对齐，VDMA 配置问题。

解决办法：通过读该问题探究，发现输入到运动检测 IP 的三幅图像没有对齐，我们直接将 OvSensor2Axis IP 输出的视频流文件与 VDMA 读通道 MM2S 输出的视频流输入到了运动检测 IP，造成了输入数据不能对齐，解决办法是新增两个 VDMA，使得运动检测 IP 的三路输入视频流均来自 VDMA 的读通道，这样就解决了数据不能对齐的问题。

（2） VGA 显示器无法运动检测信息

问题描述：搭建完运动检测系统后，进行调试，VGA 显示器无运动信息，一直为黑色

问题分析：VGA 显示器一直为黑色，有可能是运动检测 IP 的输出的视频流位宽与 VDMA 的数据位宽不匹配，或者 SDK 中对 VDMA 的配置存在问题。

解决办法：通过对运动检测 IP 核的研究，发现输出的视频流位宽为 8 位，而 VDMA 的 SS2M 通道设置的为 32 位，存在数据位宽的不匹配，因此采用了 AXI-Stream Subset Converter IP，将 8 位的数据转换为 32 位，IP 的配置如下所示：

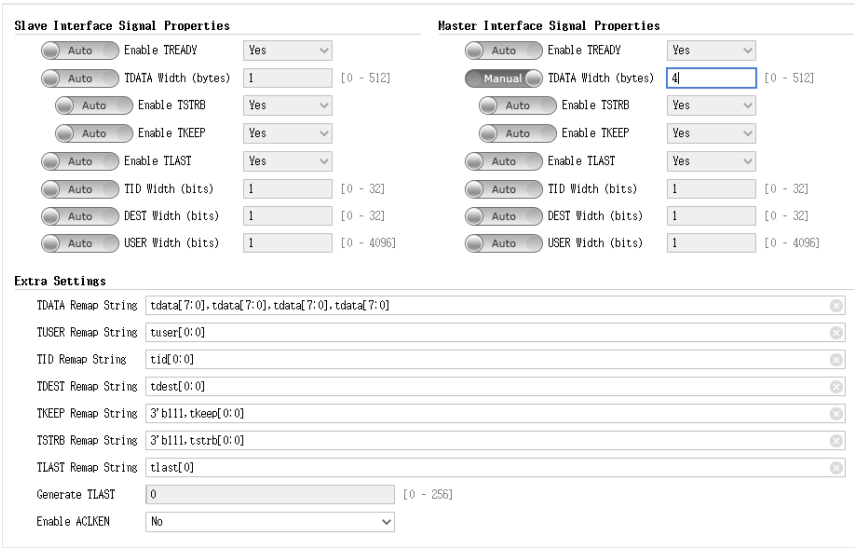


图 5-5 AXI-Stream Subset Converter IP 核配置

第六章 总结与展望

6.1 成果总结及创新性分析

本作品是以基于 ZYNQ 平台的运动物体检测系统作为设计目标。系统采用的是 EES-288 开发板作为硬件平台，利用 OV7670 摄像头模块完成了图像采集，利用 HLS 完成了图像处理与运动物体检测等硬件加速 IP,并且通过 VDMA 图像传输模块等的配置，最终将运动物体检测效果在 VGA 接口的显示器显示。本系统采用软硬件协同设计的思想，用 FPGA 来实现图像的加速，用 ARM 来完成一些控制操作，合理地利用 Zynq 中硬件资源，整个系统运行流畅，工作稳定，达到了预期效果。在设计过程中，我们将系统模块化，提高了移植性，这也方便以后的开发与维护。

本作品完成了基于 ZYNQ 平台的运动物体检测系统，设计过程中主要做了如下一些工作：

（1） 查找了运动物体检测相关的资料并进行了研究，也在 Xilinx 官方网站上对 Zynq 的一些技术文档进行学习和对例程做了一些实验，结合二者的学习经验总结，利用 Zynq 的硬件结构，对运动物体检测系统进行架构划分，分开模块进行设计，完成软硬设计方案。

（2） 深入研究了图像处理和运动检测等相关的算法后，利用 Vivado HLS 工具，进行相关的图像加速算法 IP 核的设计，其中包括灰度图转换、高斯滤波、形态学处理、三帧差分法等操作，实现了运动物体的检测。为了验证其中的一些图像算法的正确性，在 HLS 中，编写了测试代码，验证了图像处理 IP 的效果。

（3） 在 Zynq 的 PL 部分完成硬件工程的搭建，主要包括硬件加速图像处理 IP 核模块的移植、VDMA 模块配置以及 VGA 显示模块相互之间的接口信号连线;PS 部分实现系统软件设计，主要包括 VDMA 核和自定义硬件加速 IP 核的驱动程序设计，用 OV7670 摄像头实现视频图像采集，以及利用自定义硬件加速 IP 核实现视频图像处理和运动物体的检测。

（4） 用软硬件协同的方法成功实现了基于 Zynq 的运动物体检测系统设计，并对系统进行了效果测试。对运动物体检测算法进行了硬件加速和综合结果分析，并对不同的运动物体进行了测试。经测试验证，系统在运行速度和运动物体检测上均达到了预期的效果。

6.2 展望

本系统根据 ZYNQ 结构特点完成了软件与硬件部分的协同工作，但也存在一定的问题，需要在日后的设计中发现和改进。本系统虽能完成正常的运动物体检测，但 PS 部分还没有充分利用，后续的研究，希望能增加一些复杂且有实际应用价值的图像处理算法在 PS 部分，如运动物体分类识别等算法，使得本系统与其相关连的领域设计能配合使用，更大的提升作品的应用价值。

6.3 心得体会

充实而又匆忙的一个月很快就过去了，一个月前还遥不可及的作品截止提交日期，现在想来，竟然就是后天。从 11 月初，我们队三个人还是刚相处不到 50 天，参赛通知还是博哥（赵博）看到的。一般博哥看好的事情我们都相信，然后我们剩余两个人毫不犹豫就跟着博哥报了，这不分分钟就组成了我们的 fpgaclub1 队。接下来的日子就比较惨了，甚至我一度怀疑过博哥把我们坑了，三个曾经励志要做数字集成电路的男人，昨天还搞着什么 DC 综合、DFT、分析着那些单看都能认识的 NMOS 和 PMOS 管，但放在一起就看不懂的电路。现在发现选题竟然是人工智能，机器视觉！嗯，我听说过这些词，也只是听说过而已，现在竟然要玩真的了。只能告诉自己别怂，撸起袖子开始干。

在开发板还没回来之前，我们就开始收集运动检测相关的一些资料，了解运动检测常用的算法和需要运用的开发工具等，并且根据系统架构进行了分工，一个人负责了解和学习开发板，一个人负责学习图像处理的算法，一个人专门去研究运动检测相关的算法。十一月中旬，开发板到手了，我们好像也像那么回事儿。当然，中间省略了各种吐槽，网上的博客跑不通啊，还有什么算法怎么都不是 C 写的，要是有人共享个源码多好啊，这类吐槽后期竟然成了只有我们三个人能听懂的语言，其他人只觉得你们说的好高端，但一句也没听懂。就这样在各种 BUG 中摸爬滚打了半个月，十一月底，我们竟然每个人都能对自己负责的部分讲的头头是道，什么 Vivado、SDK 都能熟练使用，C++语言写起来也和 C 差不了多少吗，图像处理算法、运动检测算法也就那么几种，优缺点都如数家珍，正当我们还为自己懂了点皮毛洋洋得意的时候，桌上的台历瞧瞧的翻了一页。

距离作品提交日期还剩 15 天!!! 也是从此开始，室友除了上课都很难见到我了，每天我们三个可谓是披星戴月，处理算法开始在 HLS 中编写，各种综合报错，技术文档整天开着都快成桌面了，博哥那边硬件模块配置，隔几分钟就对着系统结构沉思。等图像处理和运动检

测算法的 IP 终于生成了第一版以后，我们就开始了第一次整体测试，结果不出所料，显示屏跟没开机一样，连个图像的影子都没有，更别说什么运动物体检测了。换个角度想，要是一次就成功，那也太低估我们作品的技术含量了。接下来就是各种调试，排错，排除各个地方的错误，尝试各种可能的实现方式，第六感都快用上了，显示效果还是不尽人意，就在我们都怀疑人生，想要放弃做最后挣扎的尝试调整系统架构后，显示效果尽然好了。当时三个人都激动的不行，我们的作品终于做出来，真的做出来了！

上面写了这么多的心得体会都难以抑制我们激动的心，其实我只想正经的说：没有什么做不出的竞赛作品，只要有一个踏实肯干的团队，每人都有坚持到底的毅力和决心，再加上时间的投入。人言“成功总在想要放弃的时候发生”，每次在想要放弃的时候，再坚持那么一下，也许下一步就真的成功了。

最后，希望我们努力的成果能得到各位专家评委老师的认可，在此表示衷心的感谢！

附录

1. OV7670 配置模块

```
//**********************************顶层**********************************//
 module OV7670_CFG(
        output      OV7670_SIOC,
        inout       OV7670_SIOD,
        output      OV7670_RESET,
        output      OV7670_PWDN,
        output      OV7670_XCLK,
        input       CLK25M ,rst_n);
    assign OV7670_RESET = 1'b1 ;
    assign OV7670_PWDN = 1'b0 ;
    pullup up (OV7670_SIOD);
    wire clk = CLK25M ;
    assign OV7670_XCLK = CLK25M ;
     parameter [7:0] par_camera_address = 8'h42;
    wire [7:0]camera_address = par_camera_address ;
    wire [15:0 ] cfg_d16 ;
    wire valid , ready ;
    cfg_reg I_CFG_REG    (
                .clk( clk ) ,
                .rst_n(rst_n) ,
                .dout( cfg_d16    ) ,
                .m_valid( valid ) ,
                .m_ready( ready )
            );
    SCCB_sender I_SCCB_sender(
                    .clk( clk ),
                    .siod( OV7670_SIOD ),
                    .sioc( OV7670_SIOC ),
                    .ready( ready ),
```

```
                    .valid( valid    ),
                    .id( camera_address    ),
                    .sub_addr( cfg_d16[15:8] ),
                    .value( cfg_d16[7:0]    )
                );
endmodule
//************************************cfg_reg*****************************//
module cfg_reg(
        input clk,    rst_n,
        output reg    [15:0] dout,
        output reg    m_valid,
        input m_ready
                );
    parameter LEN =122;
    reg [10:0] cntr = 0 ;
    always @ (posedge clk)
        if (!rst_n) cntr<=0;
        else if ( m_ready & m_valid    )
        begin
            cntr<=cntr + 1 ;
        end
always @ (posedge clk) m_valid <= ( cntr < LEN )     ;
always @ (posedge clk)
case (cntr)
    0 :     dout <= 'h3A_04 ;
    1 :     dout <= 'h67_C0 ;
    2 :     dout <= 'h68_80 ;
    3 :     dout <= 'h11_C0 ;
    4 :     dout <= 'h01_A0 ;
    5 :     dout <= 'h02_A0 ;
    6 :     dout <= 'h40_10 ;
    7 :     dout <= 'h12_04 ;
    8 :     dout <= 'h13_E7 ;
    9 :     dout <= 'h32_80 ;
```

```
10 :    dout <= 'h17_16 ;
11 :    dout <= 'h18_04 ;
12 :    dout <= 'h19_02 ;
13 :    dout <= 'h1A_7A ;
14 :    dout <= 'h03_05 ;
15 :    dout <= 'h0C_00 ;
16 :    dout <= 'h3E_00 ;
17 :    dout <= 'h70_00 ;
18 :    dout <= 'h71_01 ;
19 :    dout <= 'h72_11 ;
20 :    dout <= 'h73_00 ;
21 :    dout <= 'hA2_02 ;
22 :    dout <= 'h7A_20 ;
23 :    dout <= 'h7B_1C ;
24 :    dout <= 'h7C_28 ;
25 :    dout <= 'h7C_28 ;
26 :    dout <= 'h7D_3C ;
27 :    dout <= 'h7E_55 ;
28 :    dout <= 'h7F_68 ;
29 :    dout <= 'h80_76 ;
30 :    dout <= 'h81_80 ;
31 :    dout <= 'h82_88 ;
32 :    dout <= 'h83_8F ;
33 :    dout <= 'h84_96 ;
34 :    dout <= 'h85_A3 ;
35 :    dout <= 'h86_AF ;
36 :    dout <= 'h87_C4 ;
37 :    dout <= 'h88_D7 ;
38 :    dout <= 'h89_E8 ;
39 :    dout <= 'h09_00 ;
40 :    dout <= 'h10_00 ;
41 :    dout <= 'h0D_00 ;
42 :    dout <= 'h14_20 ;
43 :    dout <= 'hA5_05 ;
```

```
44 :    dout <= 'hAB_07 ;
45 :    dout <= 'h24_8A ;
46 :    dout <= 'h25_73 ;
47 :    dout <= 'h26_A5 ;
48 :    dout <= 'h9F_78 ;
49 :    dout <= 'hA0_68 ;
50 :    dout <= 'hA1_03 ;
51 :    dout <= 'hA6_DF ;
52 :    dout <= 'hA7_DF ;
53 :    dout <= 'hA8_F0 ;
54 :    dout <= 'hA9_90 ;
55 :    dout <= 'hAA_94 ;
56 :    dout <= 'h13_E7 ;
57 :    dout <= 'h0E_61 ;
58 :    dout <= 'h0F_4B ;
59 :    dout <= 'h16_02 ;
60 :    dout <= 'h1E_27 ;
61 :    dout <= 'h21_02 ;
62 :    dout <= 'h22_91 ;
63 :    dout <= 'h29_07 ;
64 :    dout <= 'h33_0B ;
65 :    dout <= 'h35_0B ;
66 :    dout <= 'h37_1D ;
67 :    dout <= 'h38_71 ;
68 :    dout <= 'h39_2A ;
69 :    dout <= 'h3C_78 ;
70 :    dout <= 'h4D_40 ;
71 :    dout <= 'h4E_20 ;
72 :    dout <= 'h69_00 ;
73 :    dout <= 'h6B_40 ;
74 :    dout <= 'h74_19 ;
75 :    dout <= 'h8D_4F ;
76 :    dout <= 'h8E_00 ;
77 :    dout <= 'h8F_00 ;
```

```
78 :    dout <= 'h90_00 ;
79 :    dout <= 'h91_00 ;
80 :    dout <= 'h92_00 ;
81 :    dout <= 'h96_00 ;
82 :    dout <= 'h9A_80 ;
83 :    dout <= 'hB0_84 ;
84 :    dout <= 'hB1_0C ;
85 :    dout <= 'hB2_0E ;
86 :    dout <= 'hB3_82 ;
87 :    dout <= 'hB8_0A ;
88 :    dout <= 'h43_14 ;
89 :    dout <= 'h44_F0 ;
90 :    dout <= 'h45_34 ;
91 :    dout <= 'h46_58 ;
92 :    dout <= 'h47_28 ;
93 :    dout <= 'h48_3A ;
94 :    dout <= 'h59_88 ;
95 :    dout <= 'h5A_88 ;
96 :    dout <= 'h5B_44 ;
97 :    dout <= 'h5C_67 ;
98 :    dout <= 'h5D_49 ;
99 :    dout <= 'h5E_0E ;
100 :   dout <= 'h64_04 ;
101 :   dout <= 'h65_20 ;
102 :   dout <= 'h66_05 ;
103 :   dout <= 'h94_04 ;
104 :   dout <= 'h95_08 ;
105 :   dout <= 'h6C_0A ;
106 :   dout <= 'h6D_55 ;
107 :   dout <= 'h4F_99 ;
108 :   dout <= 'h50_99 ;
109 :   dout <= 'h51_00 ;
110 :   dout <= 'h52_28 ;
111 :   dout <= 'h53_70 ;
```

```
        112 :    dout <= 'h54_99 ;
        113 :    dout <= 'h58_9E ;
        114 :    dout <= 'h76_E1 ;
        115 :    dout <= 'h11_01 ;
        116 :    dout <= 'h6B_40 ;
        117 :    dout <= 'h6E_11 ;
        118 :    dout <= 'h6F_9F ;
        119 :    dout <= 'h55_00 ;
        120 :    dout <= 'h56_5A ;
        121 :    dout <= 'h57_80 ;
    default    dout <= 16'h57_80;
    endcase
    endmodule
    //**********************************sccb_sender*****************************//
    module SCCB_sender(
                input            clk,
                inout            siod,
                output    reg     sioc,
                output    reg    ready = 0,
                input             valid,
                input [7:0] id,sub_addr,value );
          reg [20:0] cntr    = 0 ;
          always @ (posedge clk) // valid 有效时开始
                if (0==cntr)
                      cntr <= valid ;
                else
                begin
                      if ( 31 == cntr[20:11] ) cntr <= 0 ;    else cntr <= cntr + 1 ;
                end
          always @ (posedge clk) ready <=    (cntr == 0) && (valid ==1) ;
          reg [7:0]idr,addr,val;
          always @(posedge clk)if ( ( cntr == 0 ) && ( valid == 1 ) ) idr      <=    id ;
          always @(posedge clk)if ( ( cntr == 0 ) && ( valid == 1 ) ) addr    <=    sub_addr ;
          always @(posedge clk)if ( ( cntr == 0 ) && ( valid == 1 ) ) val     <=    value ;
```

```
reg SIOD_EN ;
always @ (posedge clk)
case   ( cntr[20:11] )
      0: sioc <= 1 ;
      1: case (cntr[10:9])
            2'b00:   sioc <= 1 ;
            2'b01:   sioc <= 1 ;
            2'b10:   sioc <= 1 ;
            2'b11:   sioc <= 0 ;
      endcase
      29:
      case (cntr[10:9])
            2'b00:   sioc <= 0 ;
            2'b01:   sioc <= 1 ;
            2'b10:   sioc <= 1 ;
            2'b11:   sioc <= 1 ;
      endcase
      30,31: sioc <= 1 ;
      default
      case (cntr[10:9])
            2'b00:   sioc <= 0 ;
            2'b01:   sioc <= 1 ;
            2'b10:   sioc <= 1 ;
            2'b11:   sioc <= 0 ;
      endcase
endcase
always @ (posedge clk)
case   (cntr[20:11])
      10,19,28 :SIOD_EN <=0 ;
      default SIOD_EN<=1;
endcase
reg SIOD_DAT ;//SIOD_DAT
always @(posedge clk)
case (cntr[20:11])
```

```
            0:SIOD_DAT<=1;
            1:SIOD_DAT<=0;
            2:SIOD_DAT<=idr[7] ;
            3:SIOD_DAT<=idr[6] ;
            4:SIOD_DAT<=idr[5] ;
            5:SIOD_DAT<=idr[4] ;
            6:SIOD_DAT<=idr[3] ;
            7:SIOD_DAT<=idr[2] ;
            8:SIOD_DAT<=idr[1] ;
            9:SIOD_DAT<=idr[0] ;
            11:SIOD_DAT<=addr[7] ;
            12:SIOD_DAT<=addr[6] ;
            13:SIOD_DAT<=addr[5];
            14:SIOD_DAT<=addr[4];
            15:SIOD_DAT<=addr[3];
            16:SIOD_DAT<=addr[2];
            17:SIOD_DAT<=addr[1];
            18:SIOD_DAT<=addr[0];
            20:SIOD_DAT<=val[7] ;
            21:SIOD_DAT<=val[6] ;
            22:SIOD_DAT<=val[5] ;
            23:SIOD_DAT<=val[4] ;
            24:SIOD_DAT<=val[3] ;
            25:SIOD_DAT<=val[2] ;
            26:SIOD_DAT<=val[1] ;
            27:SIOD_DAT<=val[0] ;
            29:SIOD_DAT<=0;
            30:SIOD_DAT<=1;
            default SIOD_DAT<=1;
        endcase
        assign siod = ( SIOD_EN ) ? SIOD_DAT : 'BZ ;
endmodule
```

2. HLS 运动检测 IP

头文件 top.h

```
#ifndef _TOP_H_
#define _TOP_H_
#include "hls_video.h"
#define MAX_WIDTH 1920
#define MAX_HEIGHT 1080

typedef hls::stream<ap_axiu<32,1,1,1> > AXI_STREAM_IN;
typedef hls::stream<ap_axiu<8,1,1,1> > AXI_STREAM_OUT;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3>  RGB_IMAGE;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1> GRAY_IMAGE;

void movedetection(AXI_STREAM_IN& input0_axi,AXI_STREAM_IN& input1_axi, AXI_STREAM_OUT& output_axi,int rows,int cols,int TH);

#endif
```

主程序 movedetection.cpp

```
#include "top.h"
Void movedetection(AXI_STREAM_IN& input0, AXI_STREAM_IN& input1,AXI_STREAM_OUT& output,int rows,int cols,int TH){

#pragma HLS INTERFACE axis port=input0
#pragma HLS INTERFACE axis port=input1
#pragma HLS INTERFACE axis port=output
#pragma HLS INTERFACE ap_none port=cols
#pragma HLS INTERFACE ap_none port=rows

#pragma HLS interface ap_ctrl_none port=return

RGB_IMAGE framePre(rows, cols);//上一帧
RGB_IMAGE frameNow(rows, cols);//当前帧
```

```
    RGB_IMAGE out(rows, cols);
    GRAY_IMAGE framePre_gray(rows, cols);
    GRAY_IMAGE frameNow_gray(rows, cols);

    GRAY_IMAGE Det1(rows, cols);
    GRAY_IMAGE Det2(rows, cols);
    GRAY_IMAGE Det3(rows, cols);
    GRAY_IMAGE Det4(rows, cols);
    GRAY_IMAGE temp2_pre(rows, cols);
    GRAY_IMAGE temp2_cur(rows, cols);

     #pragma HLS DATAFLOW // must use data flow to stream the data

    hls::AXIvideo2Mat(input0, framePre);
    hls::AXIvideo2Mat(input1, frameNow);

    hls::CvtColor<HLS_BGR2GRAY>(framePre,framePre_gray);//转灰度
    hls::CvtColor<HLS_BGR2GRAY>(frameNow,frameNow_gray);
    hls::GaussianBlur<3,3>( framePre_gray, temp2_pre );//高斯滤波
    hls::GaussianBlur<3,3>( frameNow_gray, temp2_cur );

    hls::AbsDiff(temp2_pre,temp2_cur,Det1);
    hls::Threshold(Det1,Det2,TH,255,HLS_THRESH_BINARY);//二值化
    hls::Erode(Det2,Det3);//腐蚀
    hls::Dilate(Det3,Det4);//膨胀
    hls::Mat2AXIvideo(Det4, output); //write the frames to video stream

}
```

测试程序

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

using namespace cv;
```

```
int main (int argc, char** argv)
{
    IplImage* src0 = cvLoadImage("input0.jpg");
    IplImage* src1 = cvLoadImage("input1.jpg");
    IplImage* dst= cvCreateImage(cvGetSize(src0), src0->depth, 3);
    AXI_STREAM_IN    input0_axi;
    AXI_STREAM_IN    input1_axi;
    AXI_STREAM_OUT output_axi;
     IplImage2AXIvideo(src0, input0_axi);
     IplImage2AXIvideo(src1, input1_axi);
     movedetection(input0_axi, input1_axi, output_axi,src0->height, src0->width);
     AXIvideo2IplImage(output_axi, dst);
     cvSaveImage(OUTPUT_IMAGE, dst);
     cvReleaseImage(&src0);
     cvReleaseImage(&src1);
      cvReleaseImage(&dst);
```

3. SDK 程序

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xil_io.h"
#include "xparameters.h"
#include "xscugic.h"
#include "sleep.h"
#define EN_ADDR 0x41200000
#define DDR_BASEADDR            0x00000000
#define VDMA_BASEADDR0          0x43000000
#define VDMA_BASEADDR1          0x43010000
#define VDMA_BASEADDR2          0x43020000
#define VDMA_BASEADDR3          0x43030000
#define VIDEO_BASEADDR0 DDR_BASEADDR + 0x2000000
#define VIDEO_BASEADDR1 DDR_BASEADDR + 0x3000000
#define VIDEO_BASEADDR2 DDR_BASEADDR + 0x4000000
```

```
#define VIDEO_BASEADDR3 DDR_BASEADDR + 0x5000000
#define VIDEO_BASEADDR4 DDR_BASEADDR + 0x6000000
#define VIDEO_BASEADDR5 DDR_BASEADDR + 0x7000000
#define VIDEO_BASEADDR6 DDR_BASEADDR + 0x8000000
#define VIDEO_BASEADDR7 DDR_BASEADDR + 0x9000000
#define HL 640
#define VL 480

int main()
{
init_platform();
print("Hello VDMA\n\r");
 Xil_Out8(EN_ADDR,0x00);
 usleep(100000);
//************************VDMA0 SS2M**********************************
 Xil_Out32((VDMA_BASEADDR0 + 0x030), 0x00000004);// enable circular mode
 Xil_Out32((VDMA_BASEADDR0 + 0x030), 0x0000008b);// enable circular mode
 Xil_Out32((VDMA_BASEADDR0 + 0x0AC), VIDEO_BASEADDR0);
 Xil_Out32((VDMA_BASEADDR0 + 0x0B0), VIDEO_BASEADDR1);
 Xil_Out32((VDMA_BASEADDR0 + 0x0B4), VIDEO_BASEADDR2);
 Xil_Out32((VDMA_BASEADDR0 + 0x0B8), VIDEO_BASEADDR3);
 Xil_Out32((VDMA_BASEADDR0 + 0x0BC), VIDEO_BASEADDR4);
 Xil_Out32((VDMA_BASEADDR0 + 0x0A8), (HL*4));
 Xil_Out32((VDMA_BASEADDR0 + 0x0A4), (HL*4));
 Xil_Out32((VDMA_BASEADDR0 + 0x0A0), (VL));
 usleep(330000);
 Xil_Out8(EN_ADDR,0xff);
 usleep(330000);
 //************************VDMA0 MM2S**********************************
 Xil_Out32((VDMA_BASEADDR0 + 0x000), 0x0000004);
 Xil_Out32((VDMA_BASEADDR0 + 0x000), 0x000108b);
 Xil_Out32((VDMA_BASEADDR0 + 0x05c), VIDEO_BASEADDR0);
 Xil_Out32((VDMA_BASEADDR0 + 0x060), VIDEO_BASEADDR1);
 Xil_Out32((VDMA_BASEADDR0 + 0x064), VIDEO_BASEADDR2);
```

```
Xil_Out32((VDMA_BASEADDR0 + 0x068), VIDEO_BASEADDR3);
Xil_Out32((VDMA_BASEADDR0 + 0x06C), VIDEO_BASEADDR4);
Xil_Out32((VDMA_BASEADDR0 + 0x058), (HL*4));          // h offset (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR0 + 0x054), (HL*4));          // h size (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR0 + 0x050), VL);
Xil_Out32((VDMA_BASEADDR1 + 0x030), 0x00000004);
Xil_Out32((VDMA_BASEADDR1 + 0x030), 0x0000008b);
Xil_Out32((VDMA_BASEADDR1 + 0x0AC), VIDEO_BASEADDR0);
Xil_Out32((VDMA_BASEADDR1 + 0x0B0), VIDEO_BASEADDR1);
Xil_Out32((VDMA_BASEADDR1 + 0x0B4), VIDEO_BASEADDR2);
Xil_Out32((VDMA_BASEADDR1 + 0x0B8), VIDEO_BASEADDR3);
Xil_Out32((VDMA_BASEADDR1 + 0x0A8), (HL*4));
Xil_Out32((VDMA_BASEADDR1 + 0x0A4), (HL*4));
Xil_Out32((VDMA_BASEADDR1 + 0x0A0), (VL));
Xil_Out32((VDMA_BASEADDR1 + 0x000), 0x0000004);
Xil_Out32((VDMA_BASEADDR1 + 0x000), 0x000108b);
Xil_Out32((VDMA_BASEADDR1 + 0x05c), VIDEO_BASEADDR0);
Xil_Out32((VDMA_BASEADDR1 + 0x060), VIDEO_BASEADDR1);
Xil_Out32((VDMA_BASEADDR1 + 0x064), VIDEO_BASEADDR2);
Xil_Out32((VDMA_BASEADDR1 + 0x068), VIDEO_BASEADDR3);
Xil_Out32((VDMA_BASEADDR1 + 0x058), (HL*4));
Xil_Out32((VDMA_BASEADDR1 + 0x054), (HL*4));
Xil_Out32((VDMA_BASEADDR1 + 0x050), VL);
Xil_Out32((VDMA_BASEADDR2 + 0x030), 0x00000004);
Xil_Out32((VDMA_BASEADDR2 + 0x030), 0x0000008b);
Xil_Out32((VDMA_BASEADDR2 + 0x0AC), VIDEO_BASEADDR0);
Xil_Out32((VDMA_BASEADDR2 + 0x0B0), VIDEO_BASEADDR1);
Xil_Out32((VDMA_BASEADDR2 + 0x0B4), VIDEO_BASEADDR2);
Xil_Out32((VDMA_BASEADDR2 + 0x0A8), (HL*4));
Xil_Out32((VDMA_BASEADDR2 + 0x0A4), (HL*4));
Xil_Out32((VDMA_BASEADDR2 + 0x0A0), (VL));
Xil_Out32((VDMA_BASEADDR2 + 0x000), 0x0000004);
Xil_Out32((VDMA_BASEADDR2 + 0x000), 0x000108b);
Xil_Out32((VDMA_BASEADDR2 + 0x05C), VIDEO_BASEADDR0);
```

```
 Xil_Out32((VDMA_BASEADDR2 + 0x060), VIDEO_BASEADDR1);
 Xil_Out32((VDMA_BASEADDR2 + 0x064), VIDEO_BASEADDR2);
 Xil_Out32((VDMA_BASEADDR2 + 0x058), (HL*4));
 Xil_Out32((VDMA_BASEADDR2 + 0x054), (HL*4));
 Xil_Out32((VDMA_BASEADDR2 + 0x050), VL);
 Xil_Out32((VDMA_BASEADDR3 + 0x030), 0x00000004);
 Xil_Out32((VDMA_BASEADDR3 + 0x030), 0x0000008b);
 Xil_Out32((VDMA_BASEADDR3 + 0x0AC), VIDEO_BASEADDR5);
 Xil_Out32((VDMA_BASEADDR3 + 0x0B0), VIDEO_BASEADDR6);
 Xil_Out32((VDMA_BASEADDR3 + 0x0B4), VIDEO_BASEADDR7);
 Xil_Out32((VDMA_BASEADDR3 + 0x0A8), (HL*4));
 Xil_Out32((VDMA_BASEADDR3 + 0x0A4), (HL*4));
 Xil_Out32((VDMA_BASEADDR3 + 0x0A0), (VL));
 Xil_Out32((VDMA_BASEADDR3 + 0x000), 0x0000004);
 Xil_Out32((VDMA_BASEADDR3 + 0x000), 0x000108b);
 Xil_Out32((VDMA_BASEADDR3 + 0x05c), VIDEO_BASEADDR5);
 Xil_Out32((VDMA_BASEADDR3 + 0x060), VIDEO_BASEADDR6);
 Xil_Out32((VDMA_BASEADDR3 + 0x064), VIDEO_BASEADDR7);
 Xil_Out32((VDMA_BASEADDR3 + 0x058), (HL*4));
 Xil_Out32((VDMA_BASEADDR3 + 0x054), (HL*4));
 Xil_Out32((VDMA_BASEADDR3 + 0x050), VL);
 cleanup_platform();
  return 0;
}
```