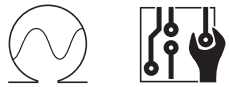
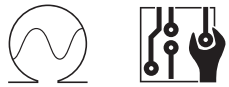


Technische Universität München
Department of Electrical Engineering and Information Technology
Institute for Electronic Design Automation

Design Space Exploration for Embedded Vision Applications on Zynq FPGA using HLS

Master Thesis

M. Umair Razzaq



Technische Universität München
Department of Electrical Engineering and Information Technology
Institute for Electronic Design Automation

Design Space Exploration for Embedded Vision Applications on Zynq FPGA using HLS

Master Thesis

M. Umair Razzaq

Supervisor :	M.Sc. Munish Jassi
Supervising Professor :	Prof. Dr. Ulf Schlichtmann
Topic issued :	15.04.2013
Date of submission :	05.11.2013

M. Umair Razzaq
Felsennelkenanger 13
80937 Munich

Abstract

Today's high-performance SoCs can host hundreds of IP blocks in a single design. With increasing demands on design productivity, a major shift towards IP reuse and efficient IP integration techniques is expected. Rapid development of application-specific system using standard IP libraries requires Design Space Exploration (DSE) at higher level of abstraction. In this thesis, a design methodology for efficient IP integration and Design Space Exploration on Zynq FPGA is presented. Development for both the Operating System based and standalone embedded system was evaluated. The thesis presents the design and implementation of standalone system setup for real-time embedded vision applications. The setup includes the HDL modules for camera interface, bus interconnects to ARM processor and display interface. First, the application software for video processing algorithms was developed for ARM processor. Later, the compute intensive tasks were migrated from ARM processor onto the Programmable Logic to benefit from FPGA performance. The desired accelerator IPs were generated and integrated with ARM based SoC according to our presented design methodology. Methodology was developed to synthesize the IPs from C-description, to prepare IP- library. Using this IP-library and IP-integration approach, DSE for video processing SoC architectures could be done iteratively. Finally, DSE is performed for selected video processing operations with various task mappings in HW and SW. The presented results provide an insight into selection of optimal architecture for application-specific systems based on computation, communication and area performance parameters. The results show progressive decrease in the CPU load with the integration of hardware accelerators.

Contents

List of Figures

List of Tables

1. Introduction

Advances in System-on-Chip (SoC) technology have enabled increased number of system functionality to be integrated on a single chip. SoC technology is a leading market driver for semiconductor industry (ITRS 2011). Today's SoCs host high-performance systems with hundreds of IP blocks. The integration of multiple applications on a single device have resulted in increasing demand for design complexity and design performance. On the other hand, there has been an ever growing gap between design productivity and design complexity. According to estimates from ITRS, at this increasing rate for logic circuits, a 10x design productivity would be required till 2019. To solve this productivity challenge, several approaches need to be combined. The level of design abstraction must be raised and the degree of automation in design verification and design implementation, must also be increased. The challenge also emphasizes on the need to increase the reuse of IPs in new designs. New design methodologies for IP integration in SoCs are needed to handle the design complexity and increase designer's productivity.

A major shift towards IP reuse and IP integration is expected in next generation of SoCs. This calls for new methods to encode and integrate the IPs. Future SoCs will require the designers to add appropriate HW accelerator IPs for application specific systems from HW IP library. These IPs can be integrated either manually or based on metamodels like IP-Xact(IP-XACT 2010) and Pcores. As a part of ongoing research at institute of Electronic Design Automation at TU Munich, a new model-driven IP integration methodology is presented in form of Grammar-based IP (GRIP) integration platform to tackle the challenge of future IP integration. It presents a concept to encode the design space of an IP library in form of rules based on generative graph grammars. These rules describe the IP integration steps to formalize the design space exploration(DSE). The platform takes RTL implementation of IP blocks and their SW access functions to generate a prototype implementation. The SoC architect starts from an initial SoC implementation and explores the design space applying grammar rules. Based on different implementations a final SoC implementation is generated.

There are two inputs to the GRIP platform from the IP supplier. First, a set of rules, known as graph grammar rules, that may be provided by IP supplier to SoC architect. These rules encode the changes applicable to the given IP. Second, it requires an IP-Xact metamodel for the HW accelerator and corresponding SW application programming interfaces (APIs). These metamodels help in identifying the design bottlenecks during the design space exploration phase. The HW and SW metamodels can be generated from CAD tools. These tools use High Level Synthesis (HLS) techniques to generate ready to integrate HW accelerators along with corresponding SW APIs. Vivado HLS from Xilinx is one such tool which helps in generation of synthesized IPs. Figure 1.1 gives an overview of a complete implementation flow using GRIP platform. In scope of this thesis, IP integration implementation for application specific

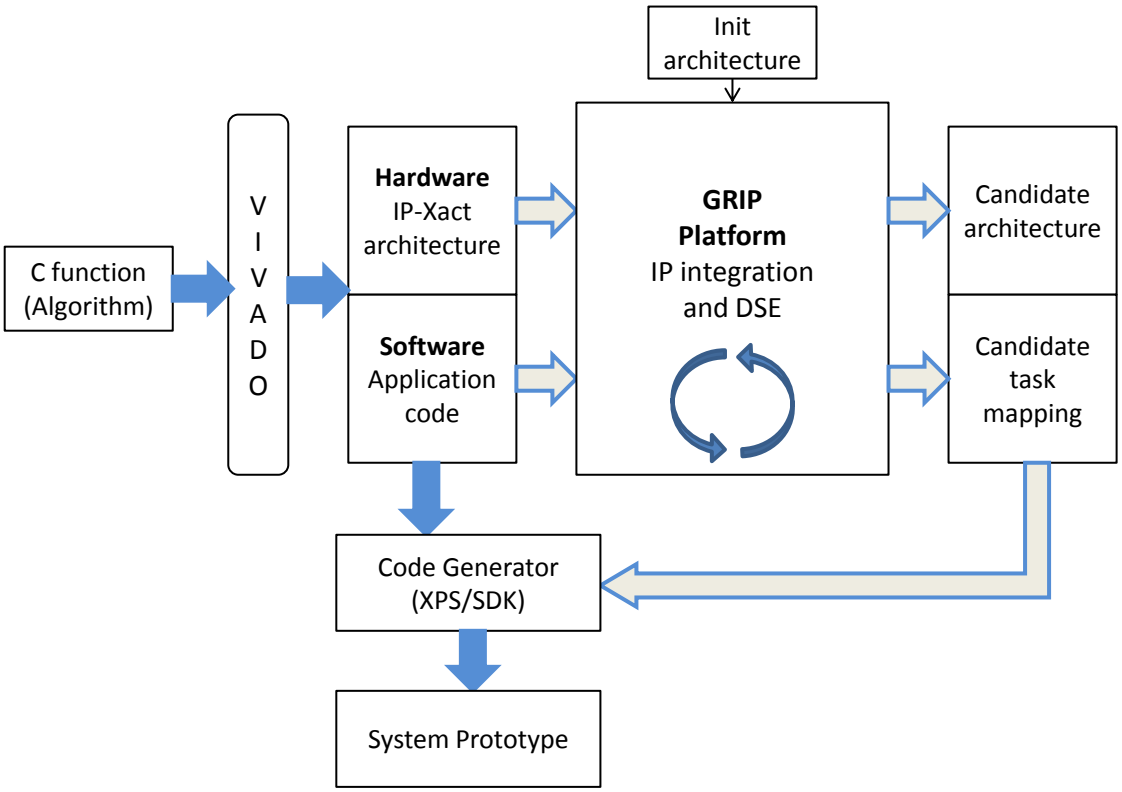


Figure 1.1.: Overview of GRIP platform

systems is presented. The application implementation elaborates a design methodology at high level of abstraction and provides basis for design space exploration. This work focuses on generating the HW architecture and SW access functions. While the GRIP platform automates the IP integration and DSE, these HW accelerators can be used for design implementation and performance evaluation on an FPGA platform. This implementation flow in the scope of this thesis is shown with solid lines in the above figure. The SoC designer takes a set of HW accelerators, integrates it with the system and generates the final implementation of the design.

The work carried out in this thesis covers the independent implementation of application specific SoC. Figure 1.2 shows the design flow that is followed in this work. A C/C++ implementation of algorithms is first converted to HW accelerators using Vivado tool. This is integrated with the rest of the embedded system consisting of a processor and peripherals. A methodology was developed to include the newly generated HW IP into Xilinx Embedded IP library, instantiate the required module and build the SoC with desired HW accelerator. The design is synthesized and implemented on Zynq FPGA.

To evaluate the impact of using different architectures on the performance of a real world application, a demanding application domain must be chosen. Embedded vision is one such application domain that requires high-performance in real-time. “Embedded Vision” refers to

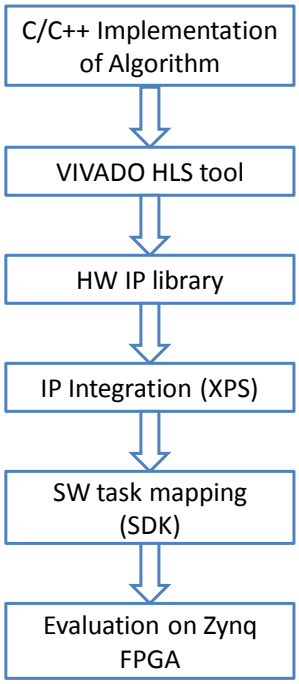


Figure 1.2.: Implementation flow from algorithm to realization

the embedded systems that not only see but also understand the meaning from visual input. High volume embedded vision markets include automotive, consumer electronics, gaming and surveillance. Video and imaging applications are becoming a greater part of our lives. From automated car parking to gaming console that understand the player's actions, from medical imaging to quality control in large production lines, from driver assistance systems to smart surveillance, all rely in one way or the other on the embedded vision technology. The computer vision tasks intensively utilize the processor resources and therefore are very suitable candidates for hardware acceleration.

Implementation of embedded vision systems is a challenging task, due to the fact that computer vision algorithms are computationally demanding and embedded systems are often highly constrained by cost, computational power and size. These vision algorithms typically require high-performance along with flexibility. Dedicated logic implementation offers high performance at lower cost but very little flexibility, whereas processors offer better flexibility at the cost of lower performance. A best compromise for demanding embedded vision application must use a combination of processing elements: Processor for decision making and flow control, and dedicated hardware logic for computationally intensive parallel processing. With the introduction of low-cost, powerful processors in conjunction with programmable logic, it is becoming possible to bring the advanced vision algorithms into reality. Zynq^{TM} family of all programmable SoCs from Xilinx provides an excellent platform for the demanding embedded vision applications.

For our work, we will present the IP integration based implementation of computer vision applications on SoC platform. Different video processing operations namely; Color conversion, image erosion and Sobel edge detection, were chosen as test applications to demonstrate the impact of various design architectures. Design space exploration is performed via different hardware and software mapping of these tasks. To evaluate the application it is required to design a setup that provides a video input to the system and displays the output on monitor. The basic setup needed to carry out experiments is shown in Figure 1.3. Other than basic system components it includes an input interface to camera and an output interface to display monitor.

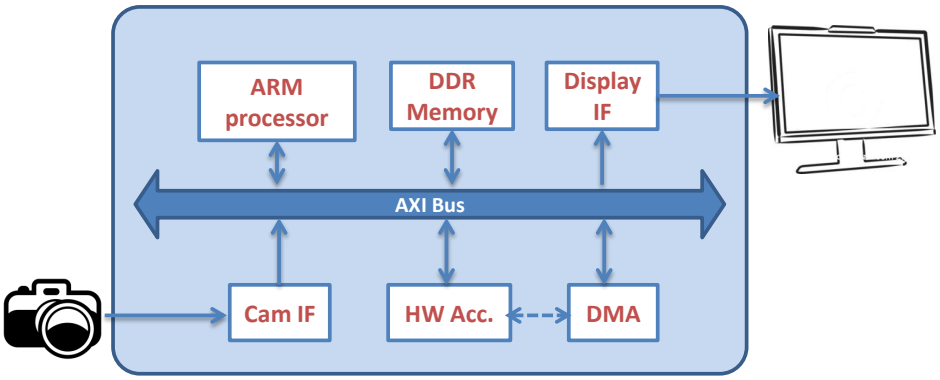


Figure 1.3.: System setup for embedded vision application

The Zedboard (ZedBoard 2013) SoC platform can be used for design implementation as it provides a single chip solution for hardware-software co-designed application. The design should be optimized based on exploration through various possible implementation schemes. These variations in implementation may result from different HW and SW partitioning of the design. A high level synthesis tool, Vivado HLS^{TM} can be used to generate IPs for subsequent parts of mentioned vision algorithm. Vivado HLS provides us with a methodology to migrate the vision algorithm from processor to FPGA logic. The design part to be implemented in hardware represents the computational bottleneck of the algorithm and should be discovered through code profiling. The thesis would provide an insight into selection of optimal implementation based upon timing, area and communication performance parameters.

To clearly present the work carried out in the thesis, this document is divided into six chapters. After brief introduction of the work in this chapter, necessary background information is detailed in second chapter. Third chapter looks into the possible solution approaches for implementing IP integration efficiently on Zedboard, while fourth chapter presents our design methodology and implementation. Fifth chapter presents the outcomes of this work followed by concluding remarks.

2. Background

This chapter provides overview of the topics that are helpful to understand the concepts and tools used in this thesis. This includes introduction to Embedded Computer Vision, overview of the ZedBoard and Zynq processors, video signaling and lastly a brief description of the OpenCV software library.

2.1. Embedded Computer Vision

Embedded computer vision, as defined by growing industry group "Embedded Vision Alliance" (EVA 2013), is a combination of two technologies: embedded systems and computer vision. Cutting-edge embedded vision systems are not only able to enhance and analyze the images but also able to trigger actions as a result of its analysis. Embedded vision can alert to a child struggling in a swimming pool and to an intruder attempting to enter a residence. It can warn the drivers of impending hazards on the roadway and even prevent them from executing lane-change, acceleration, and other maneuvers that could be hazardous to them and others. It can equip a military drone or other robot with electronic "eyes" that can enable limited or even fully autonomous operation. It can assist a physician in diagnosing a patient's illness. It can uniquely identify a face in front of the image sensor and subsequently initiate a variety of actions: automatically logging into a user account, for example, or displaying relevant news and other information. Over the past decades computer vision has been an academic research field, but with the introduction of powerful and low-cost processors it has become possible to integrate the performance of intelligent vision system with state-of-the-art embedded systems.

Generally, embedded systems can be defined as the devices used to control, monitor and assist the operation of a bigger system, machinery or plant. The term "Embedded" represents the fact that they are tightly integrated into the system. In some cases, their "Embeddedness" may be such that their existence can not be observed by a normal user of the system. Block diagram of a typical embedded system is shown in Figure 2.1. Embedded systems are processor based electronic systems other than general purpose computer with a specific task. As opposed to general purpose computer systems that perform a number of different tasks, embedded systems are usually designated to carry a single task, or small number of tasks. They are often required to fulfill the task in real-time. Embedded systems can be found in consumer electronics, automobiles, digital cameras, kitchen appliances, medical equipment, and numerous other places.

Computer vision is a research field that aims at replicating the cognitive abilities of human

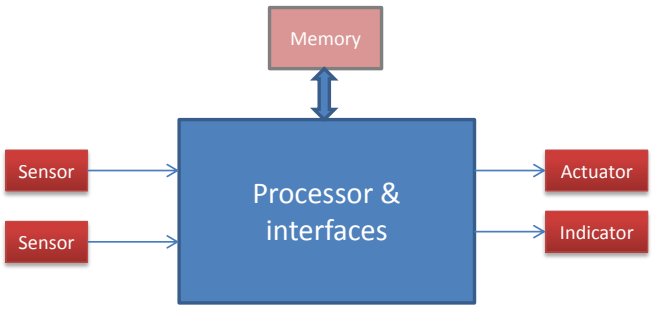


Figure 2.1.: A basic embedded system

vision into computing systems. It utilizes the digital processing and intelligent algorithms to extract information from image or video. On one hand computer vision algorithms provide the capability to enhance images by improving lighting conditions, focus and color adjustment, and on the other hand they can be used to analyze and extract information from these images. Computer vision finds its application in a wide variety of real-world applications today, including Medical imaging, Optical character recognition (OCR), Automotive safety, surveillance and so on. Although vision seems to be an effortless skill for humans it is extremely hard to automate this skill into machines. For machines it is quite hard to recognize a group of people in a frame or understand their facial expression as compared to the effort needed by a five year old kid to do the same task.

Implementing embedded vision applications is a challenging task. The complexity of vision algorithms, computationally demanding processing and limited processing power of embedded platforms contribute to these challenges. Complex embedded vision applications will most often use a combination of processing elements depending on the task at hand. A processor is normally used for decision-making, user interface, memory management and flow control, whereas a high performance DSP/FPGA can be utilized for real-time complex algorithms and parallel pixel processing. In this work we are using Xilinx's Zynq processors which provide a combination of powerful ARM processor along with Xilinx's 7 series FPGA fabric. This provides a convenient embedded platform to evaluate computer vision applications.

2.2. ZedBoard and Zynq Processors

In order to implement our Embedded Vision application, a community based development kit "ZedBoard" is used (ZedBoard 2013). The ZedBoard, as shown in Figure 2.2, is an evaluation and development board based on the Xilinx Zynq-7000 All Programmable SoC (AP SoC). The Zynq AP SoC combines the dual Cortex-A9 processing system (PS) with Xilinx's 7 series FPGA programmable logic (PL) and provides a suitable platform for architecture exploration of applications running both in hardware and software. A good mix of IO peripherals, powerful processor and multiple expansion capabilities makes it an ideal platform for rapid prototyping and proof-of-concept development for embedded vision application .

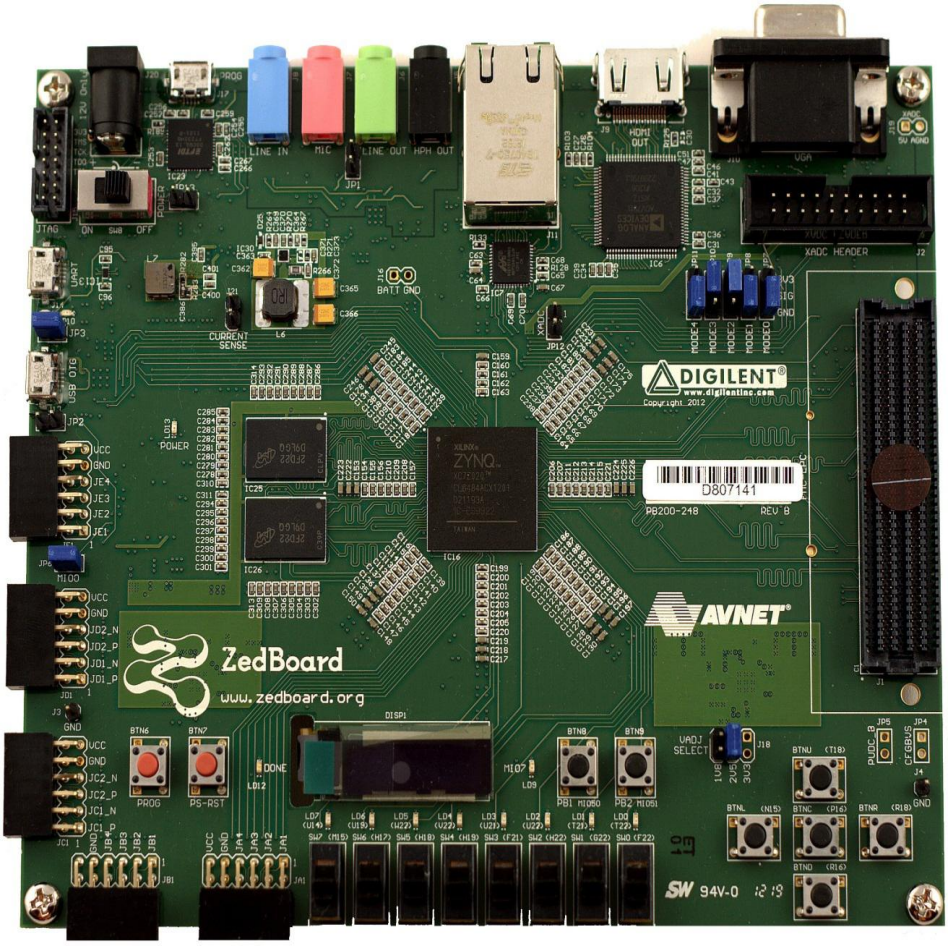


Figure 2.2.: The ZedBoard-**Z**ynq **e**valuation and **d**evelopment board

The key features provided by ZedBoard are listed below:

- Processor
  - Zynq^{TM} -7000 AP SoC XC7Z020-CLG484-1
- Memory
  - 512 MB DDR3
  - 256 Mb Quad-SPI Flash
  - 4 GB SD card
- Communication
  - Onboard USB-JTAG Programming

  - 10/100/1000 Ethernet
  - USB OTG 2.0 and USB-UART

- Expansion connectors

  - FMC-LPC connector (68 single-ended or 34 differential I/Os)
  - 5 Pmod compatible headers (2x6)

- Clocking

  - 33.33333 MHz clock source for PS
  - 100 MHz oscillator for PL

- Display

  - HDMI output supporting 1080p60 with 16-bit
  - VGA output (12-bit resolution color)
  - 128x32 OLED display

- General Purpose I/O

  - 8 user LEDs
  - 7 push buttons
  - 8 DIP switches

In this work, as we are concerned with embedded vision applications, the image capturing and display interfaces would be of primary interest to us. For image acquisition the USB OTG interface or Pmod expansion header are suitable candidates whereas one of the VGA or HDMI interface can be used for image/video display. DDR3 memory can be utilized to store the image frames. The USB-UART interface is used for user input through serial terminal.

At the heart of the ZedBoard sits its most important component-Zynq AP SoC. The Zynq chip integrates a dual-core ARM® cortex-A9 processing system(PS) and Xilinx Programmable logic(PL) in a single device. The Zynq-7000 family offers the flexibility and scalability of an FPGA, along with providing performance, power, and ease of use typically associated with ASIC and ASSPs. Thus, it provides an ideal platform for vision based embedded applications by enabling the designer to implement custom logic in the PL and custom software in the PS. Figure 2.3 gives a detailed insight into the offerings from a Zynq AP SoC platform.

The Processing System (PS), comprising of ARM core, would be used to host the software part of the application. The PS section consists of four major components

- Application processor unit (APU): It consists of dual-core ARM Cortex-A9 MPCores with operating frequency of upto 800 MHz, 512 KB Level 2 cache, 8-channel DMA support,

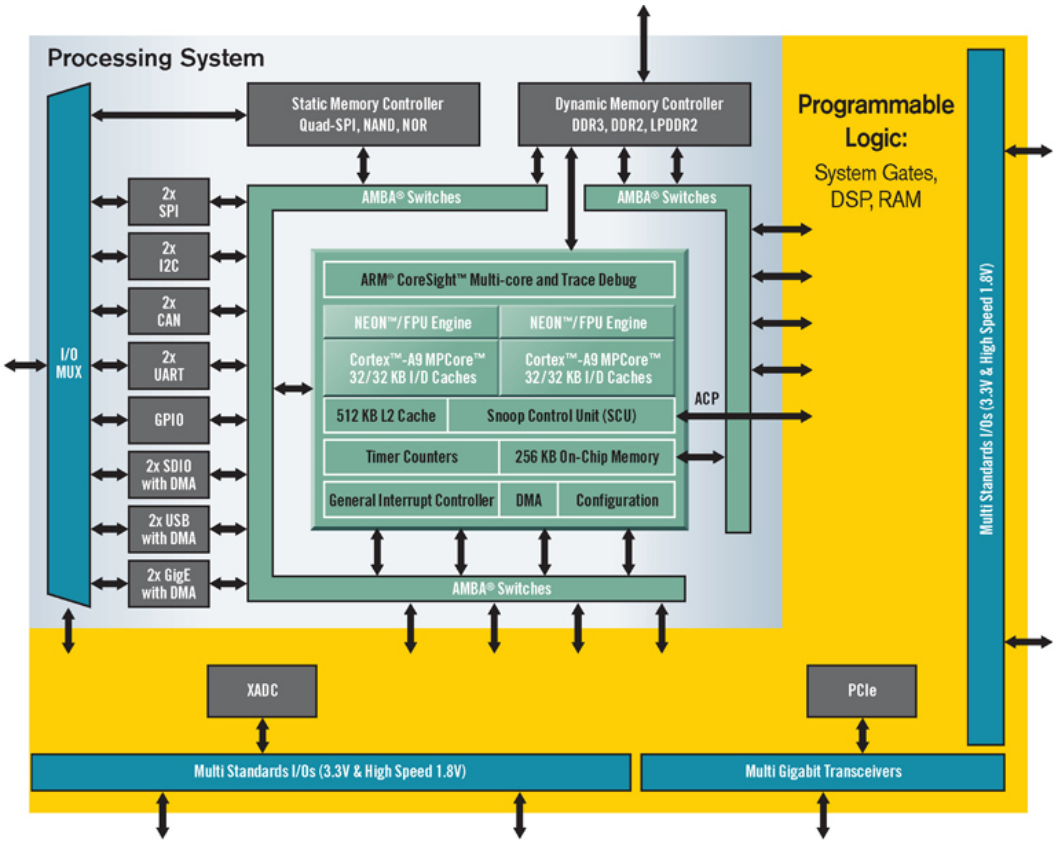


Figure 2.3.: A look inside Zynq AP SoC

interrupts and timers.

- Memory Interfaces: The memory interface unit comprises of a dynamic memory controller and static memory interface modules. The DDR memory has a multi-ported controller which enables the programmable logic and the processing system to have shared access to a common memory.

- I/O Peripherals (IOP) : The I/O peripheral unit contains the data communication peripherals. Some of the important peripherals include Ethernet, USB 2.0, CAN bus interface controllers.

- Interconnect: A multi-layered ARM AXI interconnect binds all the components of the complete system to each other and to the PL. These components include the APU, memory interface unit, and the IOP.

The PL section of the Zynq chip available on Zedboard is made up of Xilinx'x Artix-7 FPGA fabric and consists of 85K logic cells. The PL fabric also contains configurable logic blocks (CLB), 36Kb block RAM, DSP slices and programmable I/O Blocks. The hardware accelerators also known as user peripheral IPs would reside on the PL subsection.

The Zynq system provides multiple high-speed and high-performance interfaces between the PS and PL subsystems. The PS-PL interfaces include

- AXI interfaces for primary data communication,
  PS and PL can communicate via general purpose AXI ports which include two 32-bit AXI master interfaces and two 32-bit AXI slave interfaces. Along with this there are four buffered AXI slave interfaces that can be configured to 32 or 63-bit wide interface. These interfaces, referred to as high-performance AXI ports, have direct access to DDR memory and On-Chip Memory (OCM). Lastly, there is also one 64-bit AXI slave interface (ACP port) which provides coherent access to CPU memory.

- DMA, interrupts, events signals,
  There are 16 peripheral interrupts from PL to PS that PL peripheral and accelerator IPs can use to send interrupts to the PS Generic interrupt controller (GIC). Also there are four DMA channel signals for the PL to access memory.

- Extendable multiplexed I/O (EMIO,)
  EMIO allows unmapped PS peripherals to access PL I/O.

- Clocks and resets,
  There are four PS clock outputs to the PL, as well as four PS reset outputs to the PL.

2.3. Environment Setup

Embedded systems consist of hardware and software portions, which are projects in themselves, making it a complex system. Combining these two systems seamlessly, to make them function as one system, can be a challenging task. The Zynq Extended Processing Platform (EPP) solution reduces the design complexity by offering an ARM Cortex-A9 dual core as Hard IP and programmable logic along with it on a single device. For the design process, several set of tools offered by Xilinx would be utilized at different design phases.

The Embedded Development Kit (EDK) is a combination of Xilinx Platform Studio (XPS) and the Software Development Kit (SDK). It offers hardware and software application design, debug, and execution functionality. It helps to build the design on FPGA boards for verification and prototyping.

Xilinx Platform Studio (XPS) tool provides the development environment for designing the hardware portion of the embedded processor system. XPS can be used in batch mode or in the GUI mode. It can be used to add desired IPs to the existing design and create connection between the processor and peripherals. The microprocessors, peripherals, interconnection of these components and their address maps can be specified in XPS.

The Software Development Kit (SDK) is an integrated development environment, complementary to XPS, that is used for C/C++ embedded software application creation and verification.

SDK may also be used to debug the software applications. SDK is based on the Eclipse open-source framework.

PlanAhead software provides a central cockpit for design entry in RTL, synthesis and verification. PlanAhead offers integration with XPS for embedded processor design (including access to the Xilinx IP catalog), and SDK to complete the embedded processor software design. Implementation is achieved through integration with the ISE toolflow. The implementation flow of the design may be centrally launched and completed from PlanAhead. (ESTRM 2012) .

Xilinx provides the Vivado^{TM} Design Suite, which is an IP and system-centric design environment. It allows the system designer to exploit all the features available with the Zynq SoC. It increases the designer's productivity with fast IP integration and design cycles. One highlight of this design suite is its High Level Synthesis tool, Vivado HLS. The Vivado HLS takes the algorithms developed in C/C++ and converts them into RTL design specification that could be directly implemented on FPGA logic. The Vivado HLS tool is particularly well-suited to embedded vision designs. In this flow, One can create or write vision algorithms in C/C++ and compile the algorithm or parts of the algorithm into RTL using Vivado HLS. The designer can then determine which functions are better suited to run in FPGA logic and which are better suited to run on the ARM processor. It should be noted that HLS tool can only be used at component level not at system level.

2.4. Image Representation and Video Formats

Before proceeding with this thesis, it is necessary to understand how images and video are represented in digital formats.

2.4.1. Image Representation

An image can be represented in the digital format in the form of a 2D array containing M rows and N columns, where each element of the array consists of a value in a certain range. Each element of the image array is called picture elements or pixel. Pixel values typically represent gray levels, colours, heights, opacities etc. A single pixel when stored in digital format may be represented by any number of bits. The number of bits per pixel may depend on image format and image acquisition process. Common image formats include

- 1 sample per pixel (B&W or Grayscale)
- 3 samples per pixel (Red, Green, and Blue)
- 4 samples per pixel (Red, Green, Blue, and "Alpha", a.k.a. Opacity)

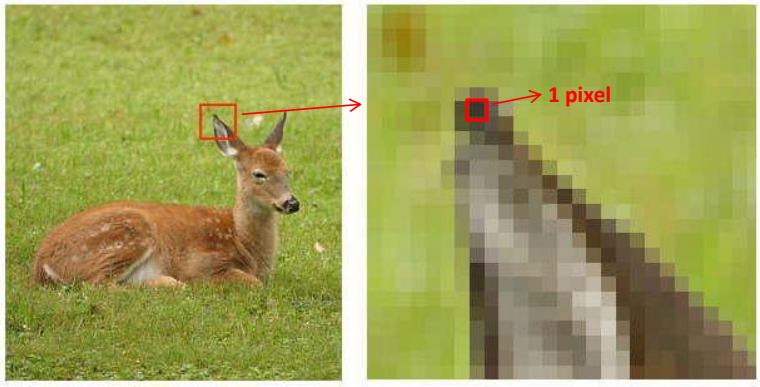


Figure 2.4.: An RGB image made of $M * N$ pixel

Each sample may consists of k bits depending on the level of intensities, L to be stored:

$$L = 2^k \tag{2.1}$$

In this project, a small camera module OV7670 is used to acquire the video. Please refer to section 4.2 for a detailed description of this camera module OV7670. As discussed in the previous section there are multiple image representations, here only the most relevant formats for the OV7670 camera are discussed.

A grayscale image consists of 8-bit pixel values in the range 0-255. Here, 0 is black, 255 is white and the intermediate values are scales of gray.

Using RGB model, each pixel must be stored as three intensities of red, green and blue lights. The most common format is RGB888, where each color pixel consists of 24-bits with 8-bits each for Red, Green and Blue components. The formats used by the OV7670 are the RGB565, RGB555 and RGB444. The difference with the RGB888 format, is the number of bits assigned to each channel. For example, in the RGB565 format, the red channel is stored as 5 bits, the green channel as 6 bits and the blue channel as 5 bits. These formats take less memory when stored but in exchange sacrifice the number of colors available.

YCbCr is a format in which a RGB color can be encoded. The Y or luminance component is the amount of white light of a color, and the Cb and Cr are the chroma components, which respectively encode the blue and red levels relative to the luminance component. The YCbCr stores each channel as 8 bits (from 0 to 255) and it can be converted into RGB using standard expressions. YCbCr444 format stores one value of Y, Cb and Cr for each pixel whereas YCbCr422 shares the same Cb and Cr value with two consecutive pixels.

A normal grayscale image has 8 bit color depth which corresponds 256 scales of gray. A “true color” image has 24 bit color depth with 8 bit each for R,G and B. this corresponds to 8 x 8 x 8 bits = 256 x 256 x 256 colors = 16 million colors. The amount of memory required to store

an image can be calculated as

$$b = M * N * k(bits) \tag{2.2}$$

For processing the images, each image is considered as a matrix. There are multiple possible ways of processing an image, which may involve single pixel processing or neighborhood operations. Single pixel operation alter the value of current pixel e.g. intensity value, while neighborhood operation use the neighboring pixels to evaluate the new value of the center pixel.

2.4.2. Video Signaling

A video is a succession of frames, a frame is a still image taken at an instant of time. A frame comprises of lines, and a line is comprised of pixels. In video processing the first concern is how to handle the synchronization signals or signal timings. Basically all video frames have three regions:

- Vertical blanking
- Horizontal blanking
- Active Image

The active region is the location of the frame where all the image pixels reside. All the processing algorithm act on the data in the active region and this is the data that is displayed

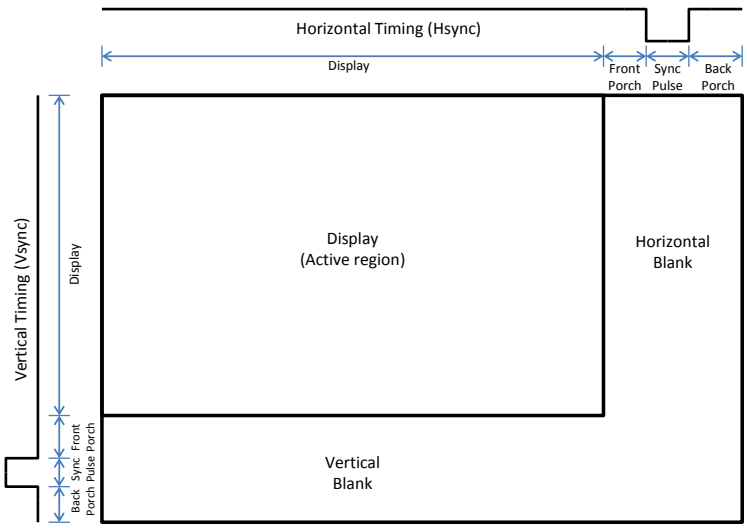


Figure 2.5.: Basic video frame format and synchronization signals

to user. The blanking regions are used to synchronize the video frames and provide space for signaling with video equipment to be able to detect and display the video properly. The video regions and synchronization signal are shown in Figure 2.5. These signals must be properly handled in order to process the video frames correctly. The values of front porch and back porch for both horizontal and vertical signals varies depending on the display resolution chosen.

The OV7670 camera module sends the video frame data along with synchronization signals as shown in Figure 2.6. These synchronization signals can be used to acquire the frame data properly. Vsync is utilized to synchronize frames whereas either of the HREF or HSYNC is used for line synchronization. This camera module can be configured for various video input formats and resolutions. This figure shows the timing for 640x480 RGB image transfer. For a detailed description of these signals please refer to OV7670 data-sheet (OV7670 2006).

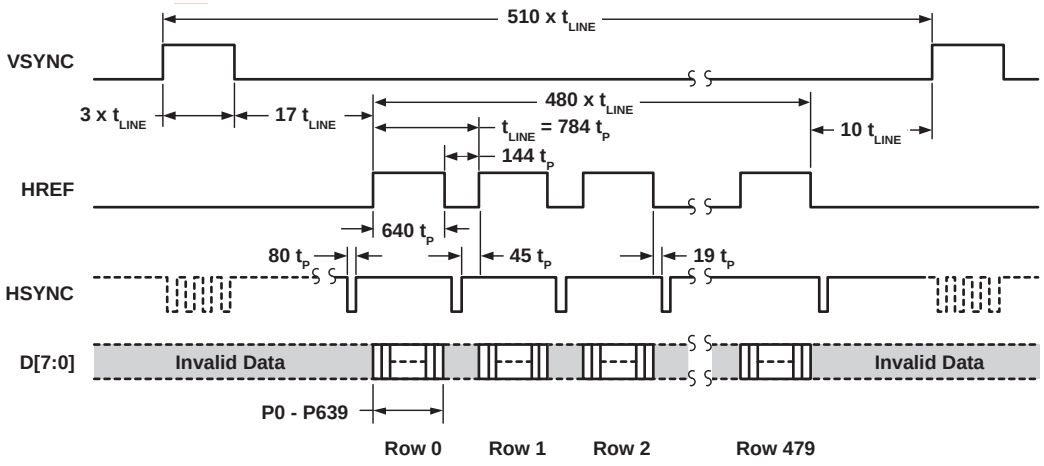


Figure 2.6.: Timing specification for video signal of OV7670 camera (OV7670 2006)

2.5. OpenCV

In this section, an open source software library will be introduced to help understand how the use of standard libraries can help increase the designers productivity. Open Source Computer Vision (OpenCV) Library is a set of computer vision software IP functions available in C, C++ and python programming languages (OpenCV n.d.). OpenCV is designed taking computational efficiency into consideration, and a strong focus on real-time applications. The OpenCV library consists of over 2500 functions that cover many areas in computer vision, including medical imaging, factory product inspection, surveillance, camera calibration, stereo vision, and robotics. OpenCV has a large user community including major companies and research centers.

OpenCV increases the productivity of software programmer by providing optimized IP functions that can be used without worrying about the implementation details. These functions

range from simple operations like color space conversion, brightness adjustment and image smoothing to complex operations like motion tracking and object detection. The programmer can simply needs use these functions into his code and tweak some configuration parameters to get the desired processed result.

With the work carried out in this thesis, we aim to provide such ease of design to the system designer. With a set of available hardware IP libraries for computer vision application a system designer should be able to use these IPs for a quick solution. The hardware IP library may comprise of open IPs, user IPs as well as third party IPs.

3. Possible Solution Approaches

The Zynq AP SoC offers multiple design options to develop an embedded system. It consists of an ARM Cortex-A9 multicore programmable system (PS) along with various dedicated peripherals as well as a configurable programmable logic (PL). The system can be designed to operate in three different modes:

- The Zynq PS can be used independently without the PL.
- The user logic design can be independently implemented on PL.
- IP cores may be added to the PL in conjunction with the PS to extend its functionality. This combination can be used to develop complex and efficient SoC designs.

As this work aims at accelerating the computer vision applications on embedded systems by use of custom hardware logic, the third configuration of PS along with PL is the most suitable option. Furthermore, this configuration allows for experimenting with different HW and SW mapping of the vision tasks and provides a platform for Design Space Exploration. Working forward in this direction, one can choose either to run an OS on the PS or set up the PS without the OS, also referred as "Bare Metal" or "standalone system". The decision to choose the PS operation mode depends on the targeted application and its requirements. In the following sections we look briefly into the factors affecting this decision.

3.1. Operating System based Application

Operating systems provide a set of services to the programmer which relieves him from worrying about underlying hardware details. They also provide software services including task prioritization, memory management, IO device drivers, inter-process communication and file systems. The OS based configuration is suitable for the systems that need a lot of flexibility and are somewhat less deterministic. Powerful processors have enabled the complex OS systems to be ported on these microprocessors. For complex processors like multicore ARM Cortex family, developers need to invest a lot of time and effort to provide a complete functional setup. This leads to use of off-the-shelf operating system. OS systems seem to be the right choice for an application running on complex processors with thousands of lines of code.

In order to evaluate the OS based implementation of a hardware accelerated embedded vision application, a complete OS was ported onto the Zynq PS. A custom hardware peripheral was later attached to the OS based system to understand the HW/SW Co-design mechanism in

this configuration. A pure software application based on OpenCV library was implemented on this system. As mentioned earlier, the Zynq platform supports a complete OS running on the ARM PS, including the embedded Linux. There are various components that are required to build, develop and boot a Linux system successfully on Zynq. The information about the PS configuration, the board layout, peripherals attached to the system and the custom hardware must be combined to make the system function properly. There are number of files that contain this information in a structured form. In this section, first, the steps required to generate and use these files to boot a Linux system on the Zedboard are discussed. Then methodology to attach a custom peripheral is presented. Lastly, GUI based OS for Zynq devices and its interface with the PL is explained.

Embedded Linux can be booted on the Zynq PS from an SD card attached to the Zedboard. The Zynq boot process begins with running code from the boot ROM. The boot ROM manages the early boot process by selecting the boot medium. Boot medium can be configured by setting appropriate jumpers on the Zedboard (ZedHW 2013). Table 3.1 lists Zynq PS boot mode selection. Afterwards, the Zynq system loads First Stage Boot Loader (FSBL) from boot medium. This requires four important items to be placed on the boot medium, which in our case is SD card. These items include boot.bin file, Linux file system, a Linux kernel image and a device tree. The boot.bin file in itself contains the initialization and configuration data for PS and PL respectively. Figure 3.1 details the design flow to build Linux based system on Zedboard. Each of the important items are discussed here briefly:

- BOOT.bin: This is a container file that consists of three important Xilinx specific files namely; FSBL.elf, uboot and bitstream.
  - First Stage Boot Loader (FSBL) file does the important early PS initialization. It initializes the DDR controller, configures the clock PLLs, configures the FPGA with the hardware bitstream (if it exists), loads and executes the Linux U-Boot image. The FSBL is created by Xilinx tools using information from the hardware project.
  - Bitstream: It contains the configuration information for the PL section of the Zynq.

	MIO[6]	**MIO[5]**	**MIO[4]**	**MIO[3]**	**MIO[2]**
JTAG Mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad SPI		1	0	0	
SD card		1	1	0	
PLL Mode					
PLL used	0				
PLL bybassed	1				

Table 3.1.: ZedBoard Configuration Modes

The bit file can be generated from the Xilinx Platform Studio (XPS).

– The U-Boot is also sometimes called second stage boot loader. It loads and starts the execution of the Linux kernel image, the Linux file system and the device tree.

- Linux kernel image: The image file contains the compressed Linux kernel. The Linux kernel can be obtained from git resources and cross compiled for targeted ARM processor.

- Device tree: The device tree file (.dtb) contains data structure that describes all the hardware present in a system to Linux kernel. This structure represents each hardware item in form of "nodes". These nodes contain necessary information for the corresponding driver to operate properly. The kernel passes through each of theses nodes during boot up process and initializes the drivers for them. It is important to note that the device tree must be updated by the designer to reflect any changes made in the hardware design.

- Linux file system: A root filesystem defines the directory hierarchy used to organize files on the system. The Zedboard supports different Linux file systems including Busybox ramdisk, Linaro and Xillinux. Last two are complete Ubuntu based Linux distributions.

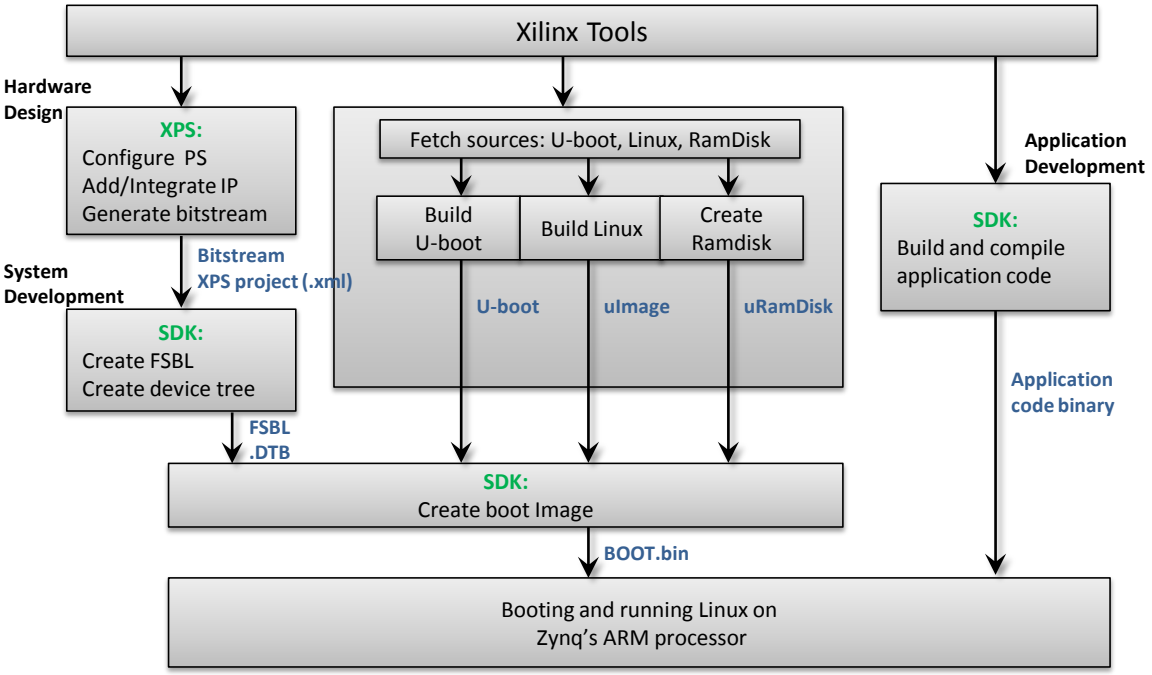


Figure 3.1.: High level block diagram of design flow for Zynq AP SoC

3.1.1. Attaching custom peripheral

Resources from "ZedBoard Linux Hardware Design" available from Digilent were used to port Embedded Linux onto the Zedboard (Digilent 2013). After having a complete OS running on the Zedboard, a peripheral in the form of an IP core in the PL can be connected to it. To assess this methodology, a simple LED controller was designed and added to the PL. A driver, was written to manage this new hardware and a software application was developed to control the LEDs on board.

The user can create an IP core using *Create or Import Peripheral* Wizard in Xilinx Platform Studio (XPS). This wizard leads the user through the settings to customize the IP core. Among others, these include defining the peripheral's bus interface, number of software accessible registers and selection of bus master support. Our IP "myled" simply shows the status of SW accessible registers onto the led. Once created the IP core can be seen in the project's repository. The wizard will generate two HDL files corresponding to IP core: `ip_name` and `user_logic`. The `user_logic` file can be updated to include the custom logic in the IP core. This IP core can be added to the hardware and interfaced with the PS through AXI bus interconnects. The XPS allows to connect the peripheral to the external ports in the Zynq PL. Any changes must also be updated in the respective .ucf file. Finally a bitstream is generated.

To boot the Linux Operating System on the ZedBoard, a boot.bin, a Linux kernel image (uImage), a device tree blob (DTB file), and a file system(uRamDisk) is needed. U-Boot can be obtained from the Digilent git repository and cross compiled for ARM by using Xilinx tools. The first stage boot loader (FSBL) is generated using Xilinx SDK. A new boot.bin file, with recently generated bitstream and FSBL, is created by Xilinx SDK. Other files namely the Linux kernel image and ramdisk image representing the filesystem stay the same. In this example pre-compiled Linux distribution provided by Digilent was used. Next step is to make this new peripheral available to the OS. For this purpose a driver code for Zedboard from Digilent was used (Digilent 2013). Additionally, the new peripheral must be included in the device tree as node. This will guide the OS to recognize that a new peripheral is attached to it and it will initialize the driver for it. The *compatibility* string of the node is the same as defined in the driver source code. The *reg* property defines the physical address space of the node and should match with the address of the IP core in the address tab of the XPS design, as shown in Figure 3.2.

After assembling all the files on the SD card as described in previous section, the system can be booted on the Zedboard. Figure 3.3 presents the architectural overview of the system. PS runnning a standalone application is interfaced with the user logic through AXI interconnect. "myled" is our custom peripheral core, "HDMI" controls the display interface operation while "I2C" core will allow the processor to configure the ADV7511 HDMI transmitter chip. After the driver is installed, an entry named "myled" will be created under the `/proc` file system. Writing `0xFF` to `/proc/myled` will light up the LEDs or as defined by the user. Linux commands `echo` and `cat` can be used to write and read on the user register defined in the peripheral and the resultant changes can be seen through LEDs. Finally, a user application was written that makes the LEDs blink by writing to `/proc/myled` utilizing the drivers.

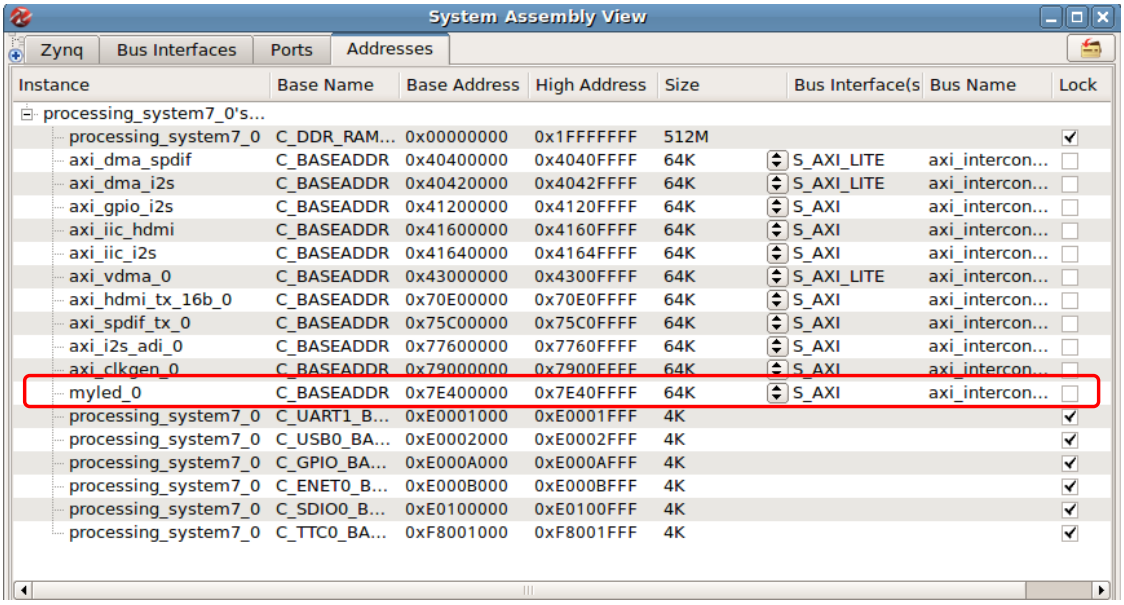


Figure 3.2.: Instantiating custom IP core in the device tree with correct physical address

This experiment demonstrates that IP integration of OS based application could be a viable approach. The methodology provides us with the IP cores that have standard interfaces which prove helpful for exploration of system architecture. Writing the driver to provide access function for the custom IP can be a cumbersome task. However, Vivado HLS tool can be used for custom hardware IP creation, which generates the drivers for synthesized IPs for Linux as well as for standalone application. This could substantially accelerate the system development time. This methodology provides interesting insights into the IP integration and could be explored further to assess its potential for design space exploration.

3.1.2. Running Xillinux OS on ZedBoard

Xillinux (Xillybus n.d.) is a complete Ubuntu based graphical Linux distribution for Zedboard. It is intended for rapid development of mixed 'software and programmable logic' applications. It provides a development platform for integration between the device's FPGA logic and user applications running on the ARM processors. Xillinux was ported on the Zedboard for evaluating the OS based application development. It boots from an SD card which must contain all the necessary items explained in the previous section. To prepare the SD card for booting, the Xillinux distribution can be downloaded from Xillybus site's download page (`http://xillybus.com/xillinux`). The distribution contains a raw image of the file system for Linux and a set of files for implementation with the Xilinx tools to produce a boot im-

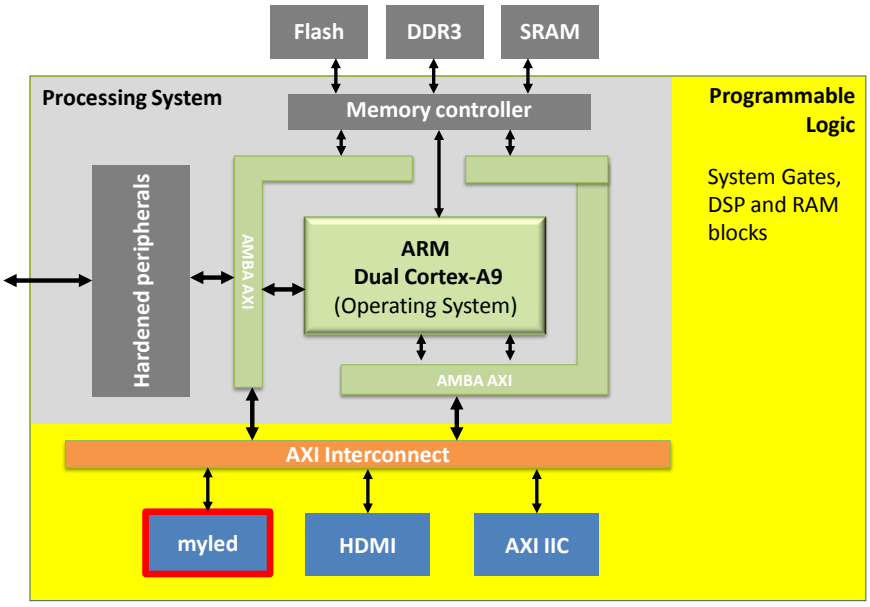


Figure 3.3.: System design to test a peripheral in PL with OS running on PS

age. Following the Xillinux guide (XillybusGS n.d.), the SD card was prepared to run Linux environment on Zedboard. To boot from SD card, jumpers on Zedboard need to be set as mentioned in Table 3.1. Before booting up the Zedboard, a computer monitor should be attached to the VGA port, a mouse and keyboard to the USB OTG connector, and an Ethernet cable can be connected to the respective port (optional). After performing all the necessary settings, Xillinux was successfully booted on the Zedboard. The Zedboard now behaves like a general purpose computer running Linux.

Once Linux is running on the PS, it can be interface to the custom logic in the PL. Xillinux has included a Xillybus IP core and its driver that can be used for data transfer between FPGA and a host running an OS. The Xillybus IP cores eliminates the need to deal with the low-level internals of kernel programming and interface with the processor. Figure 3.4 presents the block level design of Xillybus interfacing with an ARM processor. As shown in the figure, the application logic on the FPGA only needs to interact with standard FIFOs to transfer data to the host OS. The application on the computer interacts with device files that behave like named pipes. They are opened, read from and written to just like any file, but behave much like pipes between processes. The difference is that the other side of the stream is not another process, but a FIFO in the FPGA. The Xillybus driver on the host, streams the data efficiently between the FIFOs in the FPGAs and the device files on the host.

The hardware FIFOs are available in the form of a Xilinx project along with the distribution. To add custom logic to the PL, this project can be enhanced with additional HDL design files. The new bitstream and boot.bin files need to be regenerated and added to the SD card. A Xillydemo module that loops back the data from source and sink FIFOs was tested with the

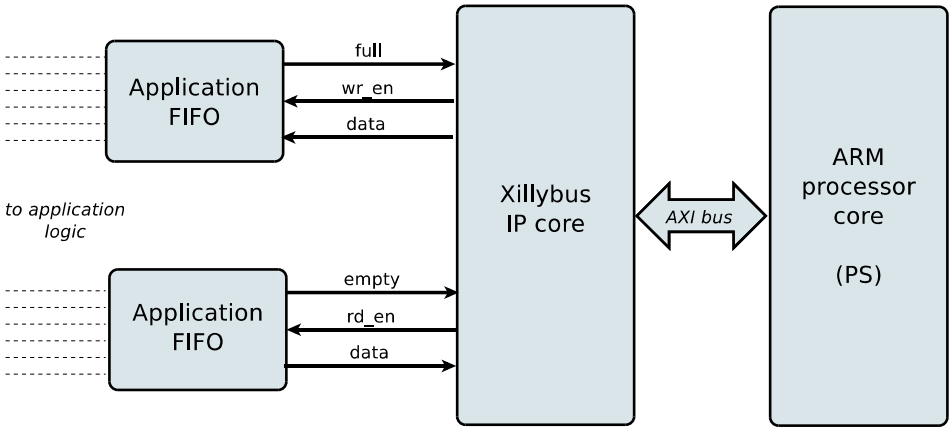


Figure 3.4.: Simplified FPGA block diagram of Xillybus using AXI

Xillinux in order to try out the basic functionality of the IP core. After installing the Xillybus driver, the loop back can be tested using Linux utility textttcat, which reads from and writes to device files. In a more practical approach, the source FIFO can be connected with the user logic that performs some processing on the input data. While sink FIFO can be connected to send IP's processed data back to the host.

The XillyBus based approach is although simple and provides quick solution for system development, but it does not offer a standard interface for IP integration. Xillybus IP is designed for simplicity rather than performance, and offers a maximum bandwidth of 200 MB/s for Zedboard. On the other hand, high quality video streaming applications require significantly higher data bandwidths. Due to lack of standard Interface and bandwidth limitations this design approach was not investigated further.

3.2. Standalone Application

In standalone or bare metal configuration, there is no supervising program (OS) between the actual application running on the processor and the embedded platform's hardware. This configuration is suitable for complicated applications that need to have the maximum control over the system resources like memory and I/O units. In this method, programmer can access the underlying core and peripherals, perform the direct data manipulations and operate on memories. There is nothing between the programmer and hardware. Timers, Interrupts and IOs can be handled directly. The standalone system is also suitable for the systems that have fewer lines of code and a defined set of tasks with minimum user intervention. In case of video processing, use of standalone system also makes sense, as the system has to perform a dedicated task instead of general purpose operation.

3.2.1. Attaching Custom Peripheral

To evaluate the standalone configuration, a custom peripheral in the Programmable Logic (PL) of the Zynq device was created, and interfaced with the ARM core in Programmable System (PS). In this configuration the ARM core only runs the user application. Once an embedded source in form of ARM is instantiated in XPS, and configured for the ZedBoard, a custom peripheral can be created. In Xilinx design flow these peripheral cores are referred as pcores. A basic peripheral core that allows writing to few registers and reads the processed data from another register was created. The peripheral can be created using *Create or Import Peripheral* Wizard. The wizard guides the user through various configuration options. The peripheral has an AXI interface and user logic software registers to allow for register access via the software on PS. It is important to select the"Generate template drive files to help you implement software interface" option during the core generation in the wizard. It will generate some sample code to help the ARM core to interact with the peripheral. When the wizard completes the core generation, the new core can be seen under the "Project Local Pcores" hierarchy in the IP catalog. To change or amend the functionality of the core, one of the files generated by wizard `user_logic.vhd`, can be modified by the user. The core can be integrated to the system by connecting it to the ARM processor in the XPS. XPS generates the attached peripheral's address map which is used by ARM processor to directly access the registers inside the core defined during core generation.

Next step is to write a software code to control this peripheral. For this purpose the generated hardware must be exported to the Xilinx SDK. In SDK, a new standalone Board Support Package (BSP) needs to be created that will be used by the tools to interface to hardware. Afterwards, a new C project is created for standalone application development. The driver file generated by *Create or Import Peripheral* wizard should be added to this project. It will simplify the interaction with the custom peripheral. This file can be located in the XPS project hierarchy under the `drivers` folder. Accessing the custom IP can be as simple as calling the hardware access functions from the driver header file, as shown in the code snippet below:

```
1  #include "led_2reg.h" //driver file
2  #define LED_2REGIP_BASEADDR 0x6E400000
3  int main()
4  {
5          //initialization code..
6          // write to reg 0 and reg 1
7      printf("Writing %i to the hardware register 0 ...\r\n ", write_value);
8      LED_2REG_mWriteReg(LED_2REGIP_BASEADDR, REGISTER_0, write_value);
9      write_value++;
10     printf("Writing %i to the hardware register 1...\r\n ", write_value);
11     LED_2REG_mWriteReg(LED_2REGIP_BASEADDR, REGISTER_1, write_value);
12
13     // now read the value of reg 2 (should be sum of above two regs)
14     current_value = LED_2REG_mReadReg(LED_2REGIP_BASEADDR, REGISTER_2 );
15     printf("Register value 2 = %i\r\n",current_value);
16     //.....
17 }
```

This design approach demonstrates a standalone application running on the Zynq ARM processor without any OS between the application and embedded hardware peripherals. This methodology not only provides an IP core with standard interfaces but also provides the SW access functions. For implementing a dedicated application like computer vision on the embedded platform this approach could prove more suitable. The vision related operation are compute intensive and need minimum user intervention. The Vivado HLS, which we aim to use for generating synthesized IP from C-based vision functions, also generates the SW access function for standalone system. Thus, this approach was selected to carry out further design experiments.

3.2.2. Standalone Video Processing System

To demonstrate a design flow for enabling acceleration of computer vision applications a video processing architecture needs to be developed. The architecture should leverage the hardware accelerators to process the video stream in real time. Video processing designs can follow one of the two common architectures. In "direct streaming" architecture data is received in the Programmable Logic part, is transmitted directly to the video processing elements in the PL and the processed data is sent directly to the video output. In second architecture, referred as "frame buffer" architecture, the frame data is first stored in external memory. It is processed from there and then stored again in the external memory, from where it can be forwarded to video output. In this work, we focus on "frame buffer" architecture as it provides more flexibility and can demonstrate how video processing application running on processor can be accelerated.

For design of a video processing architecture, essential components are shown in Figure 3.5. A video input unit captures the incoming frame and sends it to the external memory. Whereas, a video output unit is responsible to fetch the frame data from external memory and prepares it to be displayed properly. Various options to implement these units with regard to Zedboard are discussed with detail in chapter 4. Afterwards, a computer vision application can be implemented on this architecture.

A pure software based embedded implementation of vision algorithm can execute on Zynq's ARM processors. The vision algorithm may utilize OpenCV function calls to efficiently process the images. Alternatively, the algorithm can be re-factored to incorporate the acceleration by retargeting the computationally intensive part of the algorithm onto the Zynq PL. Here designer can take advantage from High Level Synthesis (HLS) technology in Vivado design suite. The Vivado HLS synthesizes the C based application-specific functions into RTL implementation. The Vivado also provides some useful synthesizable video libraries, that can be used directly to build a customized accelerator for a particular application. These video libraries also include some synthesizable openCV functions for Zynq devices implementation with both high performance and low power. Generally high data rate pixel processing tasks are moved to the PL part while low data rate frame processing tasks are handled on ARM cores. In order to evaluate the impact of migrating different functions from processor onto the programmable logic multiple hardware accelerators IPs may be generated. Each of these IP cores perform a

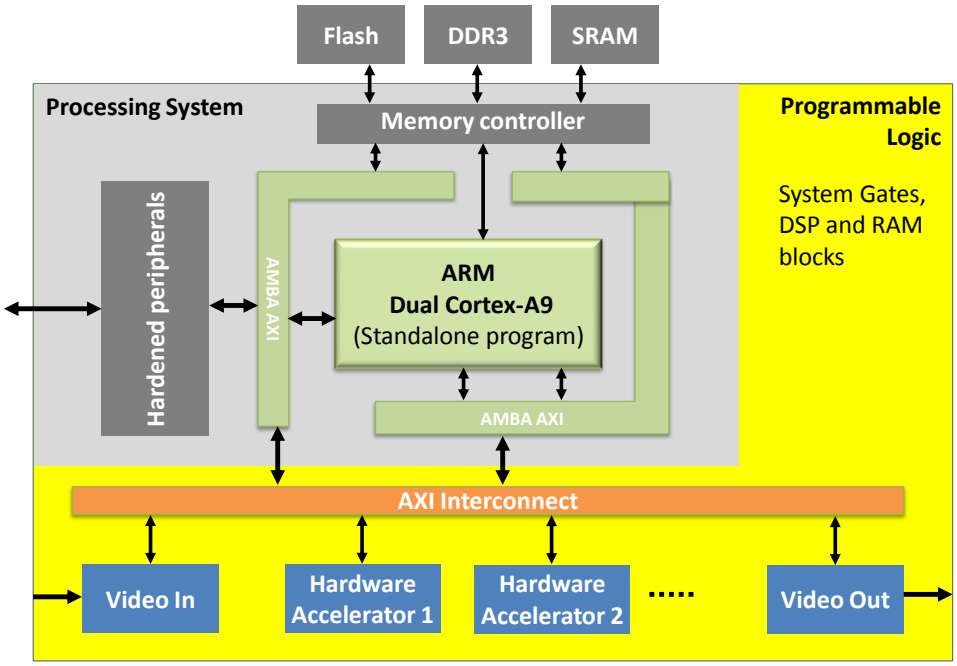


Figure 3.5.: Simplified video processing architecture

certain operation on the frame data and and store them back into the external memory.

3.3. Summary

In this chapter, we reviewed two schemes to implement embedded vision application when using PS in conjunction with PL. The OS based implementation scheme provides the developers with a set of services, based on which they can start developing their applications. The choice to run an OS or standalone software on the ARM processor highly depends on the targeted application. The applications having non determinant behavior and running thousands of lines of code are suited for OS based implementation. The applications with fewer lines of code and defined set of tasks are well suited for standalone implementation. Another important factor is the compatibility of IP and software access functions for the IP integration. Vivado HLS generates the IP cores in standardized formats with drivers for both Linux and standalone system. With two approaches discussed above, it is seen that driver files for OS based implementation need to be written by the designer, while standalone implementation can directly use the SW access functions to drive the HW accelerators. Considering these factors, a standalone embedded system would be implemented for vision algorithms. In the following chapter, the implementation scheme for video I/O interfaces along with design methodology to accelerate the user application by using HLS technology is discussed in detail.

4. Accelerating Vision Applications via IP Integration

As discussed earlier in chapter 1, we aim to accelerate the computer vision applications on an embedded platform with the help of synthesized custom IP cores. Building on the knowledge we gained through various experimental setups discussed in previous chapter, we will work forward with a standalone application development. In this chapter, the design methodology to accelerate vision applications is presented. A complete embedded system design that can be used to verify this methodology was developed. This system consists of a video input interface, a display interface and custom hardware accelerators other than basic components shown in figure 1.3 in chapter 1. In this chapter, the design components, video processing algorithms and their implementation scheme is discussed in detail.

4.1. Display Interface

A display medium is a must in any vision related application. Zedboard comes with two possibilities for display interface; Video Graphics Array (VGA) and High-Definition Multimedia Interface (HDMI) output. VGA connector on Zedboard allows for 12-bit color video output. A 4-bit video channel for each R, G and B components can be connected to respective PL pin on Zynq device (ZedHW 2013). For HDMI display, an Analog Devices ADV7511 HDMI Transmitter chip provides a digital video and audio interface to the ZedBoard. ADV7511 is 225 MHz transmitter that uses a 16bit YCbCr 4:2:2 video interface. The HMDI Transmitter has 25 connections to the Zynq device on Zedboard (ZedHW 2013).

Analog Devices offers drivers and reference designs illustrating how to interface to the HDMI transmitter device. For setting up the display for the system, the reference design from Analog devices was used (ADV7511 n.d.). This reference design provides the video and audio interface between the Zynq FPGA and ADV7511 transmitter chip on Zedboard. The reference design consists of two independent peripheral core (Pcore) modules interfaced with ARM processor. One is meant for video part while the other deals with the audio part.

The video part consists of Xilinx VDMA IP and the HDMI video interface core. AXI VDMA implements a high-performance, video-optimized DMA engine with frame buffering. It transfers video data streams to and from memory. In this design the VDMA streams frame data to the HDMI Pcore. The HDMI Pcore for video part reads 24-bits of RGB data from DDR memory. It performs the color space conversion (RGB to YCbCr) followed by subsampling (444 to 422), thus preparing the data for ADV7511 transmitter. The video core is capable of supporting any video format through a set of parameter registers. The reference design is supplemented with a *clock generator* module that generates the correct pixel clock for various

display resolution. Using this reference design the ADV7511 transmitter chip can be configured over I2C interface. There is a set of registers that must be programmed to select the right configuration for the video input and output mode. These settings are described in the ADV7511 programming guide (PG 2012).

To configure the HDMI core for various display formats and resolution through SW running on ARM processor the reference design uses the *ADV7511 Transmitter Library*. This library is a collection of APIs that provide a simple interface to ADV7511 transmitter chip. The library sits between the user application and the hardware cores and provides the user with a set of APIs that can be used to configure HDMI transmitter hardware without the need for low-level register access.

The hardware architecture for this reference design in shown in Figure 4.1. Audio related module in the PL are not shown in this block diagram. The AXI4-Lite interface is connected

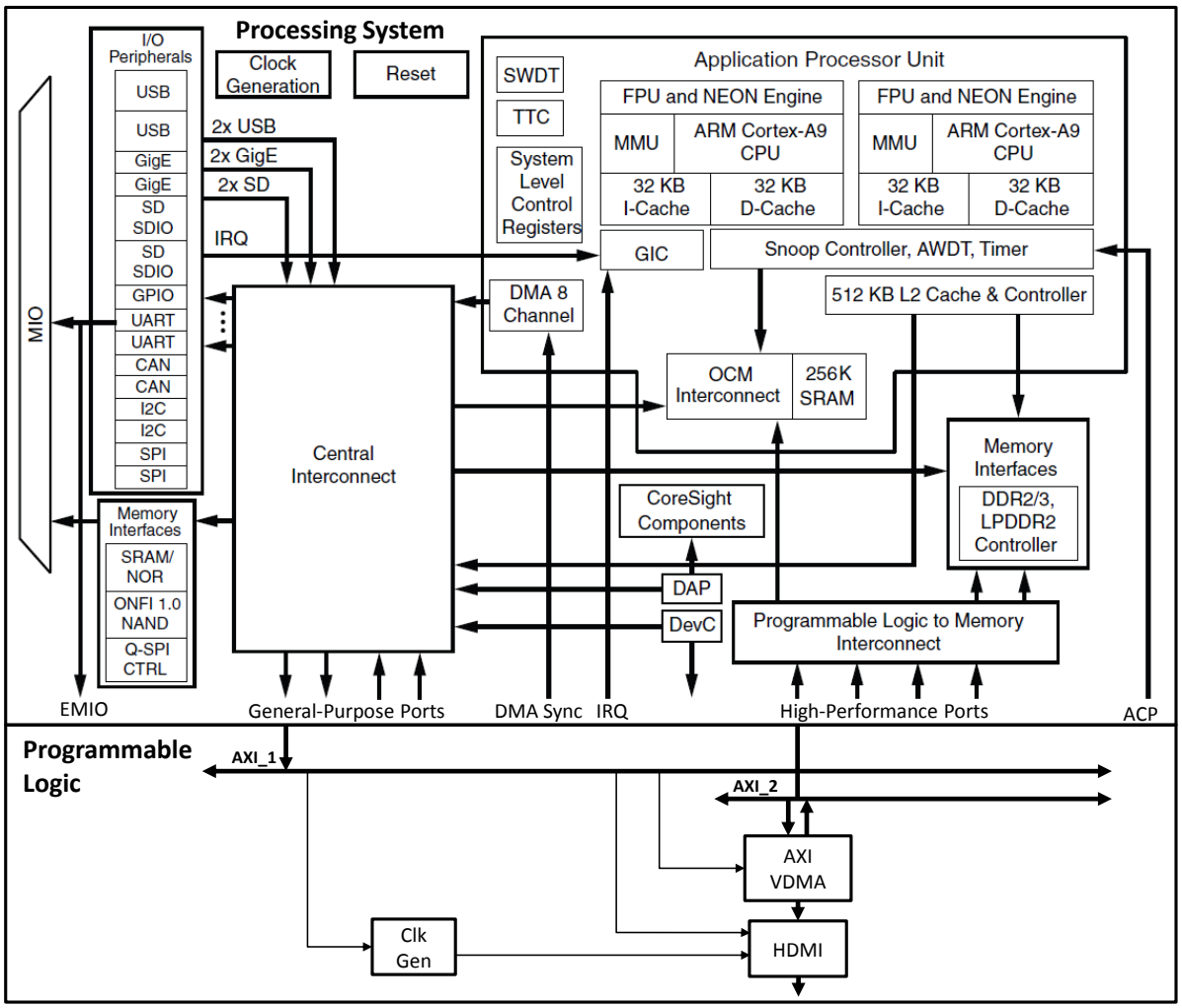


Figure 4.1.: Block diagram of hardware architecture for display Interface

to the general purpose port 'GP0' of the PS through an AXI4 Interconnect 'AXI_1'. It is used by the CPU to configure core's parameters at run time. Second AXI interface is connected to high performance port 'HP0' of the PS through 'AXI_2'. The video frame to be processed is read from memory by the VDMA through this port. This frame is forwarded to HDMI core which prepares the frame for display and sends it to the ADV7511 transmitter. In this setup, an image stored in the DDR memory is displayed repeatedly as there is no video input interfaced to the system, which will be the next design step.

4.2. Camera Interface

To acquire images from a video source, Zedboard comes with a USB On-The-Go (OTG) interface as well as Digilent Pmod headers. The USB OTG is connected to the 'USB0' hardened peripheral, connecting it to the ARM processor in Zynq PS. However, USB camera drivers are required to be developed for using this scheme. The other option is to use Pmod headers for camera interface. Zedboard has five Pmod headers (2x6), four of which are connected to the PL. A low cost image sensor OV7670 (OV7670 2006) was attached to the PL section of the Zynq AP SoC via two Pmod headers, that works as a video source.

The OV7670 is a low voltage CMOS image sensor that provides full functionality of a single chip VGA camera. The OV7670 camera module can operate at a maximum of 30 fps for 640 x 480 (VGA) resolution. It gives complete control to the user to select between various output formats, image quality and frame resolutions. It can be configured via the Serial Camera Control Bus (SCCB) at the start up. The supported output formats include Raw RGB, RGB (RGB565/555/444), YUV 422 and YCbCr 422. The camera module comes with a 9x2 header which is connected to the two Pmod headers, JA1 and JA2, on the Zedboard. Table 4.1 shows the functionality each pin as well as their connections with the Zedboard and subsequently with the Zynq FPGA.

4.2.1. Camera Configuration

To setup the OV7670 camera module, a configuration register set is to be programmed at system start up. This is done via SCCB interface which is compatible with I2C interface. For camera configuration the I2C interface transfers the register settings over the SCCB bus. The camera module is clocked by the XCLK, which is connected to a 25MHz clock source from Zynq PL. This clock is used to generate frame rate timing as directed by the respective register settings. The internal clock frequency F_{INT} of OV7670 module can be calculated from the input clock frequency F_{CLK} by following equation.

$$F_{INT} = F_{CLK} * PLL\_Multiplier/(2 * (CLKRC[5:0] + 1)) \tag{4.1}$$

Here 'PLL_multiplier register' selects the input clock multiplication factor and can be either by-passed or set to a value of 4, 6 and 8. The 'CLKRC' register is the clock pre-scalar and can

Name	Type	Description	Pmod connection	Zynq Pin connection
SDIOC	Input	SCCB clock	JB7	V8
SDIOD	Input/Output	SCCB data	JB3	W8
VSYNC	Output	Vertical synchronization	JB6	V9
HREF	Output	Horizontal synchronization	JB2	V10
PCLK	Output	Pixel clock	JB5	W10
XCLK	Input	System clock	JB1	W11
D_0	Output	Video data	JA1	AA11
D_1	Output	Video data	JA5	AB10
D_2	Output	Video data	JA2	Y10
D_3	Output	Video data	JA6	AB9
D_4	Output	Video data	JA3	AA9
D_5	Output	Video data	JA7	AA8
D_6	Output	Video data	JB0	W12
D_7	Output	Video data	JB4	V12
RESET	Input	Reset (Active low)	JA4	AB11
PWDN	Input	Power down (Active high)	JA0	Y11

Table 4.1.: Pin connections between OV7670 and Zedboard

be used to adjust the frame rate. Refer to Table 4.2 for these register settings. These register are set such that the camera module provides us with RGB 565 image format at VGA 640x480 resolution. The frames are received at 15fps. A color matrix compensates for lighting and temperature effects. Hue, color saturation and color space conversion can also be controlled by this matrix. Other registers configure the settings for image calibration and pixel correction. The detailed description of these register settings can be found in OV7670 implementation guide (OV7670 2005).

Register	Address	Default Value	Description
CLKRC	0x11	0x80	Bit[6]: 0: Apply prescaler on input clock 1: Use external clock directly Bit[5:0]: Clock prescaler F(internal clock) = F(input clock)/(Bit[5:0] + 1) Range [0 0000] to [1 1111]
DBLV	0x6B	0x0A	Bit[7-6]: PLL control 00: Bypass PLL 01: Input clock x 4 10: Input clock x 6 11: Input clock x 8

Table 4.2.: Internal clocking of OV7670

4.2.2. Camera Interface Peripheral Core

As shown in the Table 4.1 the OV7670 camera module supplies two output synchronization signals with the image data. The vertical sync signal (Vsync) and the horizontal reference signal (HREF) are provided along with Pixel clock (PCLK). By default, the PCLK will have the same frequency as XCLK, however prescalers and PLLs can be configured through the SCCB, to generate a PCLK of different frequency. Both 'Vsync' and 'HREF' are continuous signals. The 'HREF' is only valid when there is some data available at the output. When there is no data, the 'HREF' signal would stay inactive. The 'Vsync' signal is asserted for a specific duration after a complete frame has been transmitted. See Figure 2.6 for sync signal timing specifications. As we have set the camera module to provide data in RGB565 format, one complete pixel (consisting of 16-bits) is received in 2 successive clock cycles. The data format and timing is shown in Figure 4.2. The pixel data is captured at the rising edge of PCLK when 'HREF' is high. It is important to note that the rising edge of 'HREF' signals the start of a line, and the falling edge of 'HREF' signals the end of the line.

To configure the camera module and acquire the images from it, a custom peripheral core was designed and integrated with the Zynq SoC. Figure 4.3 shows the internal architecture of this custom peripheral core. The *Controller* module is responsible for programming the OV7670's configuration register set, as discussed in previous section. The *Register Data* module stores

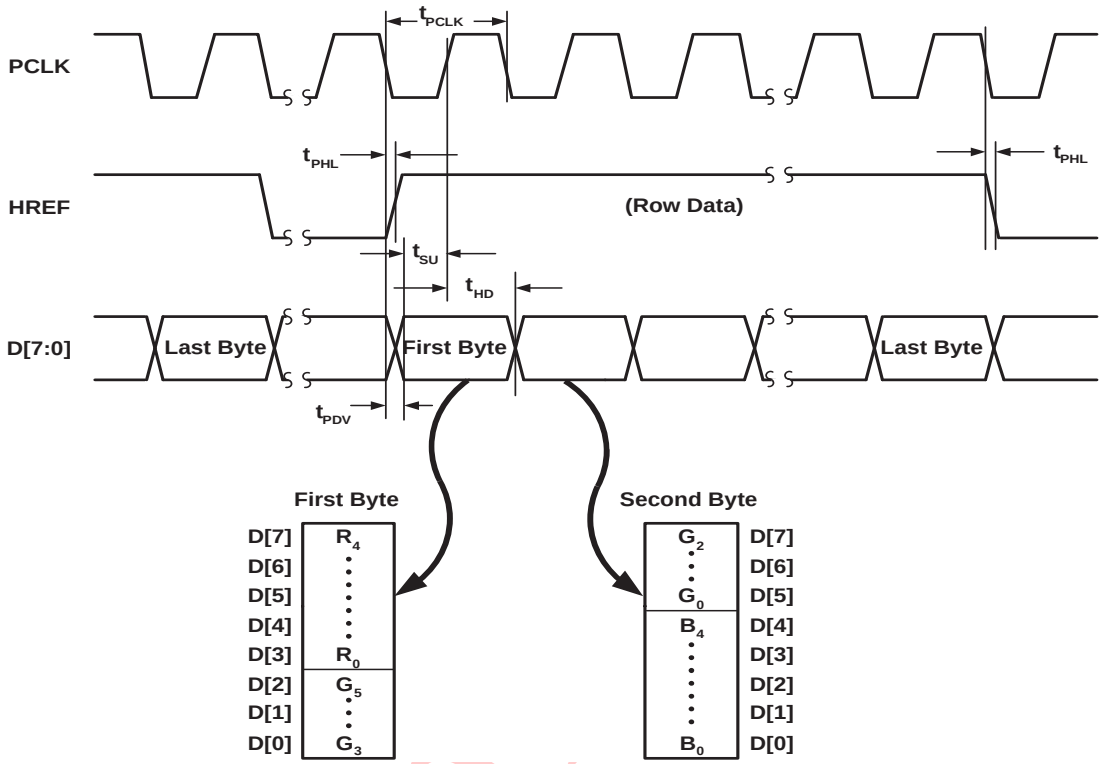


Figure 4.2.: RGB565 output timing diagram (OV7670 2006)

the initialization data which is transferred to the OV7670 camera via the *I2C interface* module. The *Capture Logic* module is responsible for acquiring the image data coming from the camera. This module samples the pixel data at rising edge of PCLK and creates an RGB pixel in two data cycles. In order to make this data available to the ARM processor, we need to interface this logic with the rest of the system. For this purpose an AXI4 slave memory peripheral was generated using 'Create and Import Peripheral' wizard in XPS tool. The peripheral consists of two user specific memory regions, called *Line Buffers*. These buffers store one line of the frame data i.e 640 pixels. The peripheral can be connected to the AXI4 interconnect through AXI IP interface (IPIF). IPIF provides a quick way to implement the interface between the AXI interconnect and user logic. The *Line Buffers* are connected to the AXI bus as slave peripherals on one side whereas they are written alternatively by the *Capture Logic* on the other side. This data stored on the line buffers can be read by an AXI bus master, in our case this happens to be the ARM processor. The synchronization signals, 'Vsync' and 'HREF', are transferred through clock domain crossing (CDC) and made external to the core. These signals are used as interrupts for the ARM processor.

The *Camera Interface* peripheral core (Pcore) module is connected to the rest of the Zynq SoC design in XPS. The camera IF peripheral. It is attached to the AXI interconnect as slave memory peripheral. This Pcore 'cam_mem' appears to the processing system as two addressable memory regions. Figure 4.4 shows the interconnections of the Pcore with the Zynq SoC as well as their address ranges. The sync signals are connected as interrupts to the Generic

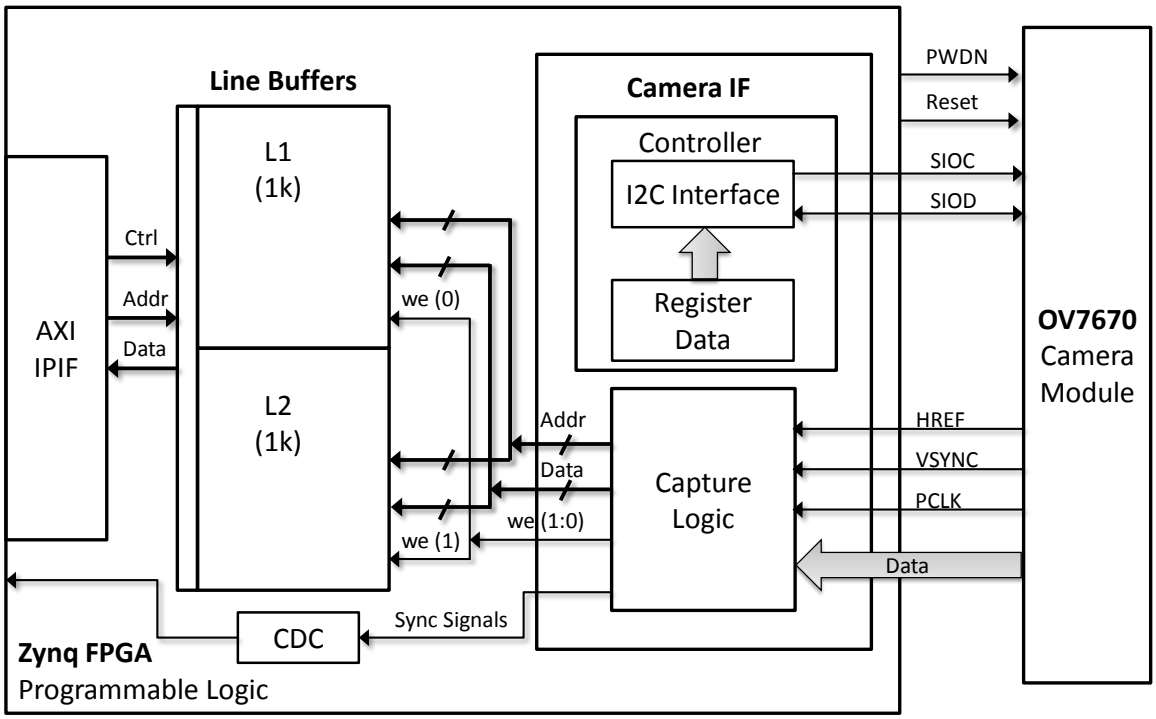
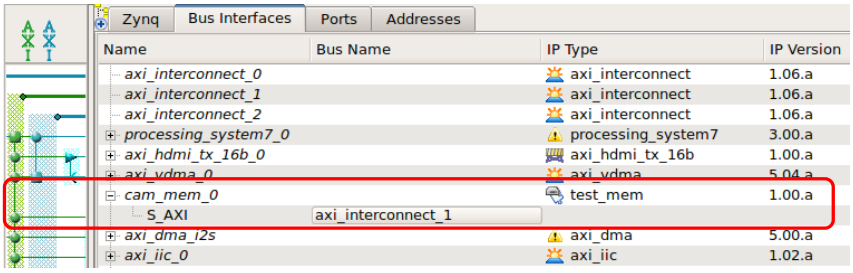
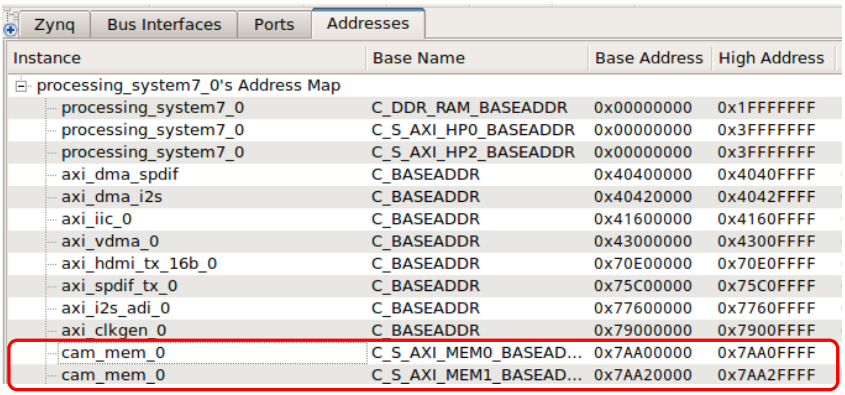


Figure 4.3.: Architectural block diagram of 'Camera Interface' peripheral



(a) Connecting pcore on the AXI interconnect



(b) Address map of the two line buffers

Figure 4.4.: Connecting Camera IF core with Zynq SoC

Interrupt Controller (GIC) of the ARM. The software running on the processor utilizes these interrupts for frame synchronization. Figure 4.5 shows the interrupt connections of the Pcore with processing system. The number shown along the connected interrupts is their Interrupt ID. It is used by the processor to identify the interrupt and call the respective Interrupt Service Routine (ISR).

Integrating the camera interface with Zynq AP SoC design completes the necessary setup required to perform further design exploration experiments. Figure 4.6 shows the hardware architecture of the programmable logic part of the design at this stage. In this setup we acquire data through camera interface, the ARM processor in the PS reads this data through the AXI interconnect and writes it onto the frame location in the DDR memory. The display interface puts this image stream onto the HDMI monitor.

4.3. IP integration with High Level Synthesis (HLS)

High Level Synthesis (HLS) tools provide a methodology to quickly integrate the customized IP into application specific systems. HLS tools can be used for efficient design at higher level of abstraction, that could prove helpful for Design Space Exploration. In this flow, algorithms are created in C/C++; complete algorithm or part of it is synthesized into RTL using Vivado

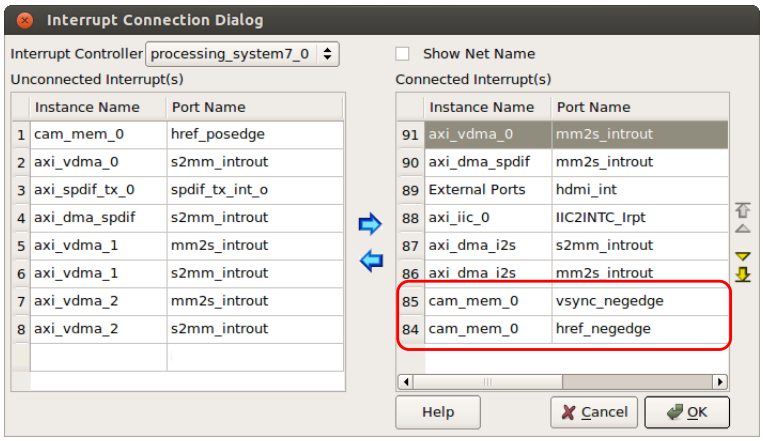


Figure 4.5.: Interrupt connections for camera IF core

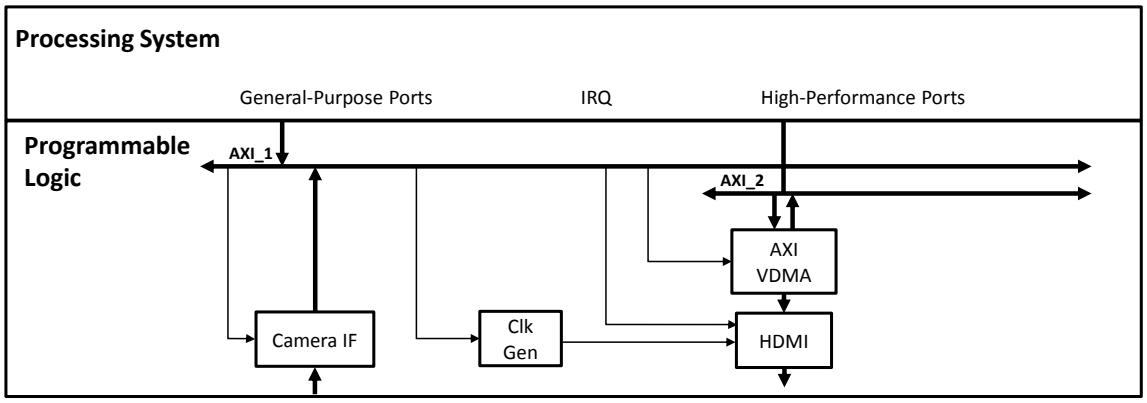


Figure 4.6.: PL architecture with Camera and Display interfaces

HLS and it is determined which functions are better suited to run on FPGA logic and which are better suited for ARM processor. In the scope of this work, we aim to generate customized IP core to accelerate the computer vision applications. The Vivado HLS tool from Xilinx is used to generated IP cores that migrate the application processing from processor onto FPGA logic. In case of Zynq device, it means moving the code from ARM Cortex-A9 in the PS to the FPGA logic in the PL. The part of processing moved from processor to the hardware should represent the computational bottleneck of the algorithm. This bottleneck is discovered through code profiling on the processor.

The procedure to take C implementation of algorithm, generate RTL with HLS tool and integrate it with Zynq embedded design is explained with the help of three different video processing functions that play important role in many computer vision applications. The functions 'Grayscale conversion', 'Erosion' and 'Edge detection' are used to evaluate the effect of various architectures on the computational load of the processor. Each of these functions will be discussed in detail in later sections. Before moving on with their hardware implementation it is necessary to touch upon some essential design topics. These topics provide primary insight

into the design methodology of video processing systems on Zynq Device.

AXI4 Streaming Protocol

Video processing components from Xilinx generally use the AXI4 streaming protocol to transfer the image data. This protocol describes each line of video pixels as an AXI4 packet. It has special signals to mark start and end of line as well as complete frame. Each AXI4 stream channel acts a unidirectional data flow channel. AXI4-Stream interfaces and transfers do not have address phases and allow unlimited burst size. The HLS tool synthesized IP cores for video processing usually have AXI4 stream interfaces. Video systems can be build by combining AXI4 streams with AXI memory mapped IP. A DMA engine can be used to move data between memory and AXI4 stream IP. The signals and timing details for this protocol can be found in AXI reference guide (AXIRG 2011).

AXI Video DMA IP

AXI Video Direct Memory Access (VDMA) core from Xilinx is designed to provide efficient high bandwidth data access between AXI4-Stream video data and AXI4 Memory Mapped data. On one end it has memory mapped interfaces which can be attached to an AXI interconnect as master. This interface can access memory directly, typically connected to external DDR memory. With the memory map interfaces there are two associated AXI4-Stream interfaces: AXI Memory Map to Stream (MM2S) master and AXI4-Stream to Memory Map (S2MM) slave. These stream interfaces can be connected to the user specific video processing hardware accelerator synthesized by Vivado HLS. The primary data movement between system memory and the target IP is through the AXI4 Memory Map read master to AXI MM2S stream master and AXI S2MM stream slave to AXI4 Memory Map write master.

One of the operating mode of the AXI VDMA is 'Register Direct Mode', which allows the processor to control the operation of core. In this mode video parameters like frame height and width, start addresses of memory can be configured by the processor through the slave AXI4-Lite interface. AXI VDMA is incorporated into the system via AXI interconnect. The AXI VDMA core supports multiple data bus width for AXI4 Memory Map and AXI4-Stream. For details on register space and design parameter configuration please refer to the product guide for the VDMA core (AXIVDMA 2012).

Memory Structures for Video Processing

Video processing algorithm, regardless of the exact computation being performed by them, are memory intensive operation. The memory architecture for these algorithm has a direct impact on the overall system performance and resource utilization. The Vivado HLS enables the generation of accelerators for different stages of processing and different video formats.

With regards to HLS tool their are various ways to define the memory structure in C. The video processing can be implemented at different level of granularity. A function may operate on a single pixel, a memory window or complete frame. A key aspect in video processing is instantiating and handling memory buffers. The memory buffers allow the designer to have temporal and spatial data for a function to operate on.

The three memory styles are shift registers, memory windows and line buffers. Shift registers are one of the simplest and most commonly used memory structures in DSP processing. They provide a one-dimensional temporary data buffer to store the incoming samples from a streaming interface. In the HLS tool, an array is the simplest and most straightforward way of declaring a shift register. A memory window defines a neighborhood of 'N' pixels centered on one pixel which form a 2D data storage element. Only the pixels required to compute some characteristic of center pixel are stored. An example of this is the 3 x 3 memory windows used in edge detection. In the HLS tool, a 2D array is the basic definition of declaring a memory window. An array can be decomposed into independent registers at the RTL level using the directives for array partitioning. A line buffer is a multi-dimensional shift register capable of storing several lines of pixel data. The line buffers are implemented as block RAMs in RTL. A line buffer is declared in C/C++ as a multi-dimensional array. Although a memory window is a subset of a line buffer, a line buffer cannot be used directly in most video and image processing algorithms. By coding memory architectures as part of the algorithm, the user has full control over the type of hardware being created by the Vivado HLS tool.

High Level Synthesis

Vivado HLS provides us with an efficient way to synthesize C/C++ implementation of the algorithms into RTL and integrate it with the SoC design. Vivado HLS transforms the C/C++ functions into optimized RTL for user-defined clock frequency and device. It enables rapid migration from desktop development to FPGA implementation. The design flow for HLS is shown in Figure 4.7. In the first stage, the algorithm that is targeted for FPGA logic is specified and developed in software. The functional verification can be carried out at this level thus elevating the verification abstraction level for RTL to C. In second stage Vivado HLS transforms this algorithm from C/C++ implementation into an optimized FPGA implementation. Finally the RTL implementation can be packaged into a standard IP format like IP-XACT or Pcore, that can be used by other Xilinx tools for IP integration into larger systems.

The high level synthesis performs two types of synthesis on the design:

- 'Algorithm synthesis' takes the content of the C function and synthesizes it into RTL statements.
- 'Interface synthesis' transforms the function parameters into RTL ports with specific protocol.

Other than these two synthesis, HLS can perform optimization on the design as per user

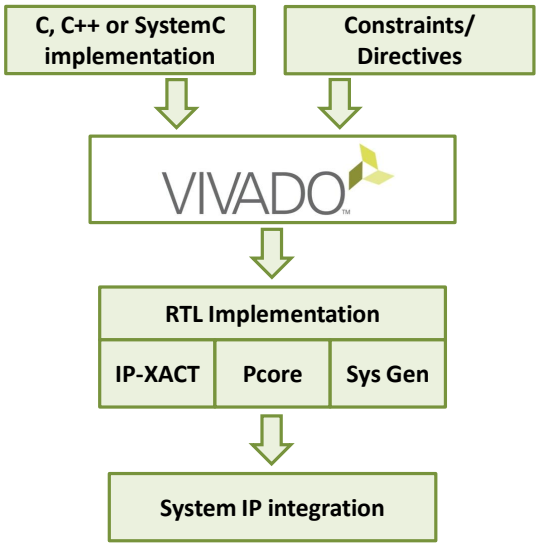


Figure 4.7.: High Level Synthesis design flow with Vivado HLS from C-Algorithm to IP integration

directives and constraints. With these optimization designer can direct the design towards the desired high performance implementation. When writing a function in high level language that is to be targeted for FPGA implementation, some important things must be considered. For compilation into an FPGA using Vivado HLS, the code cannot include any run-time dynamic memory allocations. The designer must explicitly describe all the memory that an algorithm consumes. In C based design, all input and output operations are performed, in zero time, through function arguments. In an RTL design, these same input and output operations must be performed via the design interface and must follow a specific IO protocol. User can explicitly describe the interface IO protocol in the input source code. Details of various options for these IO protocols for function arguments can be found under the "Interface Management" section of High-Level Synthesis user guide (HLSUG 2013).

For video processing it is common to use AXI4 stream interface. This interface can be connected to any function argument synthesized into RTL port using **ap_fifo** IO protocol. Vivado HLS provides C++ class `hls::stream<>` for modeling streaming data structures. Output interfaces are implemented as AXI4 stream masters while input interfaces as AXI4 stream slaves. As final goal is to interface this RTL implementation as IP in the XPS, the stream protocol should match the stream protocol in the block to which it will be connected. This can be modified by opening the configuration window of Vivado HLS core in XPS.

Processor Control of HLS designs

RTL implementation of the algorithm generated by Vivado HLS can be exported as standardized IP. For effective interfacing and control of these IP blocks by processor Vivado HLS also

provides software access functions. Only the scalar IO ports are accessible from the processor over the AXI4-Lite interface. These complimentary APIs enable the software development for the processor. When exporting the HLS design as Pcore these APIs are located inside the `include` directory of the Pcore. The Vivado HLS provides a header file which contains all the necessary driver functions for standalone software development. These functions assist the software with IP block initialization, status monitoring, I/O read and write operation. IP blocks created with Vivado also generate interrupt at the end of the function call. The API functions provided by the Vivado HLS tool have this naming convention:

`X{top level function name}_{operation}`

If we synthesize a top level function called `example`, then `XExample_Start()` API would serve to start the operation of this IP block. `XExample_IsDone()` monitors the status of IP block and the driver function `XExample_SetA` writes a value from the processor to port A of the IP block. Likewise, there are driver functions for interrupt control with regard to the IP block. Generally, any program making use of IP blocks generated by the Vivado HLS tool needs to execute these tasks:

- Initialize the IP block in the processor space
- A basic interrupt service routine for the IP block
- Register the IP block ISR in processor exception table
- Write data to the IP ports as per IO protocol
- Start the IP block
- Read the returned/processed data from IP

These SW access functions are necessary elements of a standalone application on the Zynq AP SoC which makes use of the Vivado HLS IP blocks. Below is a code snippet for a simple IP block generated from Vivado HLS to verify successful integration of the IP block into SoC design. Lines 18-20 set the parameters for the 'example' function, line 23 starts the execution of this function while line 28 reads the returned value.

```
1  #include "xexample.h" /* Generated from Vivado HLS for core 'example' along with the
       .c file */
2
3  #define N 16
4
5  XExample example_inst;
6  int main()
7  {
8          int i;
9          int *mem_ptr = (int *) XPS_TARGET_ADDRESS;
10     init_platform();
```

```
    Xil_DCacheDisable();

    example_inst.Lite_BaseAddress = XPAR_EXAMPLE_TOP_0_S_AXI_LITE_BASEADDR;
    example_inst.IsReady = XIL_COMPONENT_IS_READY;
    /* Initialize contiguous memory region with a pattern*/
        for(i=0;i<N;i++)
               mem_ptr[i] = i;
    XExample_SetByte_rdoffset(&example_inst, 0x0);
    XExample_SetByte_wroffset(&example_inst, 0x40);//4xno_of_words
    XExample_SetValueadded(&example_inst, 0x10);

    if (XExample_IsIdle (&example_inst))
        XExample_Start(&example_inst);

    while( !XExample_IsDone(&example_inst));

    for(i=0;i<N*2;i++)
        xil_printf("%d \n\r", *(volatile u32 *) (XPS_TARGET_ADDRESS + i*4));
    return 0;
}
```

4.4. Design Space Exploration for Computer Vision

Computer vision applications range from industrial monitoring to automotive systems. Many of these systems are implemented using various image processing operations to extract information from the video. In this work, a methodology to retarget these applications on Zynq devices is presented. The methodology leverages High-Level Synthesis (HLS) technology from Vivado, as well as from Xilinx synthesizable video libraries. This allows for quick development of embedded vision applications with optimized performance. With HLS the high data rate pixel processing tasks can be targeted onto Programmable Logic, while lower data rate processing and control tasks stay with the ARM core in the Processing System.

Three different image processing algorithms, “Grayscale Conversion”, “Erosion” and “Sobel Edge Detection”, that form essential part of many computer vision application are used to explore different architecture implementations on Zynq FPGAs. The architectures are based on different configuration of hardware and software mapping of these algorithms. Grayscale conversion is the simplest with respect to computational load, whereas Sobel edge detection is the most complex of the three. An algorithm may first be designed and implemented as C/C++ function running on Zynq AP SoC. In this case video processing is executed on the ARM Cortex-A9 processor. These video processing functions can be synthesized into standard IP blocks, implemented onto the Zynq PL and integrated with the AP SoC. The new video processing design flow now utilizes these synthesized IP blocks for performance optimized implementation.

‘Grayscale conversion’ of colored image is a single pixel operation and the simplest of these

algorithms. A colored image consists of R, G and B values, whereas a grayscale image only carries image intensity, composed of scales of gray. A straight forward way to convert a color pixel is to take an average value of three color channels. The *luminosity* method is a more sophisticated version of the average method. It calculates the weighted average to account for human perception. As human eye is more sensitive to green than other colors, so green is weighted most heavily. Equation 4.2 and 4.3 represent the formulas for *average* and *luminosity* methods respectively.

$$Y = (R + G + B)/3 \tag{4.2}$$

$$Y = .212R + .715G + .072B \tag{4.3}$$

'Erosion' is a morphological image processing operation which operates on the neighborhood of N pixels centered on pixel P. The kernel or mask is a 2D operator consisting of 3x3 pixels, centered on pixel P. As the kernel is scanned over the image, the minimal pixel value in the neighborhood is computed and the pixel P is replaced with this value. This causes bright regions in the image to shrink.

'Sobel edge detection' is a classical algorithm for extracting the object edges in an image. It is an image transformation operator which changes the image representation. Sobel operation calculates the gradients in 'x' and 'y' directions to find the edges. The Sobel operator kernel is shown in Figure 4.8. This operator also uses two 3x3 kernels which are scanned over the whole image to find gradients in 'x' and 'y' direction. Absolute values of x and y gradients are combined to get the new value of the center pixel.

We start with a pure software implementation of these algorithms based on the "frame buffer" architecture, explained in section 3.2.2. The algorithms are implemented as user code running on the ARM processors in the PS. Each of the algorithm reads a frame from DDR memory modifies it and stores the processed frame back to external memory. At this stage, the whole application consisting of three different operations with distinct computational complexities can be time profiled. See section 5.1 for details on profiling the software on ARM processors. Based on computational complexity a complete algorithm or a part thereof can be retargeted onto FPGA. The design methodology to retarget the algorithm from PS onto the PL leverages from HLS technology provided by Vivado. The algorithms may be implemented as synthesizable

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

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

Figure 4.8.: Kernel for Sobel operator

C/C++ code using HLS video library combined with application specific user code to build customized IPs. This design methodology is presented in the next section. HLS generated IPs are integrated with the Zynq system to accelerate the application processing time. The application can once more be profiled to see the impact of various architecture mappings. The camera interface acquires the video stream and ARM processor stores it onto the external DDR memory. Depending on the task mapping, these IP blocks may process a video stream from external DDR memory. The processed frames from IP cores are stored back into the DDR memory and sent to display by HDMI interface.

4.5. IP Integration Design Methodology

The design methodology for IP integration relies on the backbone architecture developed earlier, that provides 'Video In' and 'Video Out' interfaces for camera and display respectively . The methodology follows the below mentioned steps to accelerate the embedded vision applications :

1. Develop and execute the application on the ARM processor in Zynq AP SoC.
2. Refactor the application to figure out the compute intensive functions.
3. Use Vivado HLS for these identified functions to generate accelerator IP cores and their software APIs.
4. Integrate the HW accelerators to the Zynq AP SoC in the Programmable logic.
5. Replace the calls to the SW functions with the calls to the accelerator APIs.

This design methodology is elaborated with an example of color to Grayscale conversion algorithm. As a first step, the algorithm is targeted for implementation on ARM processor. The color conversion can be implemented either by user code or with help of OpenCV library functions. Now, in order to accelerate this algorithm, a synthesizable C/C++ implementation of the algorithm is required. Vivado HLS contains a number of video libraries, implemented as synthesizable C/C++ code, that make it easier to build such designs. Many of the video concepts and data structures in this video library are similar to the OpenCV. An important structure to represent the images is `hls::mat<>` in HLS library that corresponds to `cv::Mat` class of OpenCV. The synthesizable video library functions are listed in High-Level Synthesis user guide (HLSUG 2013). The Vivado HLS video library implements some of the important OpenCV functions like `cvtColor` for color conversion. To perform color conversion, Vivado HLS provides `hls::CvtColor` library function. It converts a color image from/to a Grayscale image. The type of conversion is defined by the value of code:

- HLS_RGB2GRAY convert a RGB color image to a grayscale image.

- HLS_BGR2GRAY convert a BGR color image to a grayscale image.
- HLS_GRAY2RGB convert a grayscale image to a RGB image

The `hls::CvtColor` is converted to RTL implementation by 'Algorithm synthesis'. As mentioned earlier, the Xilinx blocks use AXI4-Stream format to transfer videos. For 'Interface synthesis', proper interface formats and IO protocols must be defined. To abstract programmer from these interfacing issues a set of synthesizable video interface functions are included in Vivado HLS. In particular, the `AXIvideo2Mat` function receives a sequence of images using the AXI4 streaming video and produces an `hls::Mat` representation. Similarly, the `Mat2AXIvideo` function receives an `hls::Mat` representation of a sequence of images and encodes it correctly into the AXI4 streaming video protocol.

A synthesizable C++ implementation of Grayscale conversion using video library is shown in the code below. Along with the video library functions, the synthesizable code also contains a number of `#pragma` directives that enable ease of interfacing. These directives specify the input and output streams as AXI4 streaming interfaces (line 3-4) and other inputs and outputs as AXI4 Slave interface (line 6-8). In addition *rows* and *cols* are specified to use `ap_stable` IO protocol (line 10-11). The directives can also be used for design optimization like *dataflow* directive (line 15) enables concurrent execution of processing functions.

```
1  void convert_to_gray(AXI_STREAM& input, AXI_STREAM& output, int rows, int cols) {
2      //Create AXI streaming interfaces for the core
3  #pragma HLS RESOURCE variable=input core=AXIS metadata="-bus_bundle INPUT_STREAM"
4  #pragma HLS RESOURCE variable=output core=AXIS metadata="-bus_bundle OUTPUT_STREAM"
5
6  #pragma HLS RESOURCE core=AXI_SLAVE variable=rows metadata="-bus_bundle CONTROL_BUS"
7  #pragma HLS RESOURCE core=AXI_SLAVE variable=cols metadata="-bus_bundle CONTROL_BUS"
8  #pragma HLS RESOURCE core=AXI_SLAVE variable=return metadata="-bus_bundle
       CONTROL_BUS"
9
10 #pragma HLS INTERFACE ap_stable port=rows
11 #pragma HLS INTERFACE ap_stable port=cols
12
13     RGB_IMAGE img_0(rows, cols);
14     RGB_IMAGE img_1(rows, cols);
15 #pragma HLS dataflow
16     hls::AXIvideo2Mat(input, img_0);
17     hls::CvtColor<HLS_RGB2GRAY>(img_0,img_1);
18     hls::Mat2AXIvideo(img_1, output);
19 }
```

Vivado HLS provides a complete development environment for software applications targeted for synthesis. The software functionality can be verified by simulating the design with input vectors. Next, the C design is synthesized into RTL design. Vivado HLS not only allows for simulation of the software functions in desktop environment, but also provides C/RTL cosimulation where result from RTL design is applied back to C testbench to verify the results.

The final step in high level synthesis flow is to export the RTL implementation as a standard IP that can be integrated with rest of the design. As Zynq based embedded system is developed in XPS, the IP needs to be exported as Pcore. The export process will not only package the RTL into IP block but also generate the drivers for standalone and Linux based systems development.

Next step in our design methodology requires this IP to be integrated to the "frame buffer" backbone architecture. The packaged IP and driver files can be found under the Vivado project hierarchy in `impl/pcores` and `impl/drivers` folders respectively. The Pcore representing the accelerator IP can be copied into the XPS project hierarchy or under central peripheral repository. In case an XPS project is already open, the `project > Rescan User Repositories` need to be selected to update the newly added accelerator IP in the *Project Local PCores*. The three accelerators for our algorithms generated with HLS design flow can be seen in the Figure 4.9 in the left window. Now these Pcore IPs can be added to the system and connected with the rest of the system. As these are video processing IP blocks operating on AXI4-Stream format video, an AXI VDMA IP from Xilinx IP catalog can be used for interfacing the IP with Zynq AP SoC. Example connections for "Sobel edge detection" hardware accelerator are highlighted in Figure 4.9. The Sobel IP is attached to one end of the `axi_vdma_4` through AXI4 stream interfaces. On the other end, `axi_vdma_4` is connected to the High Performance port via `axi_interconnect_3`. Through this interconnect the VDMA can access the external DDR

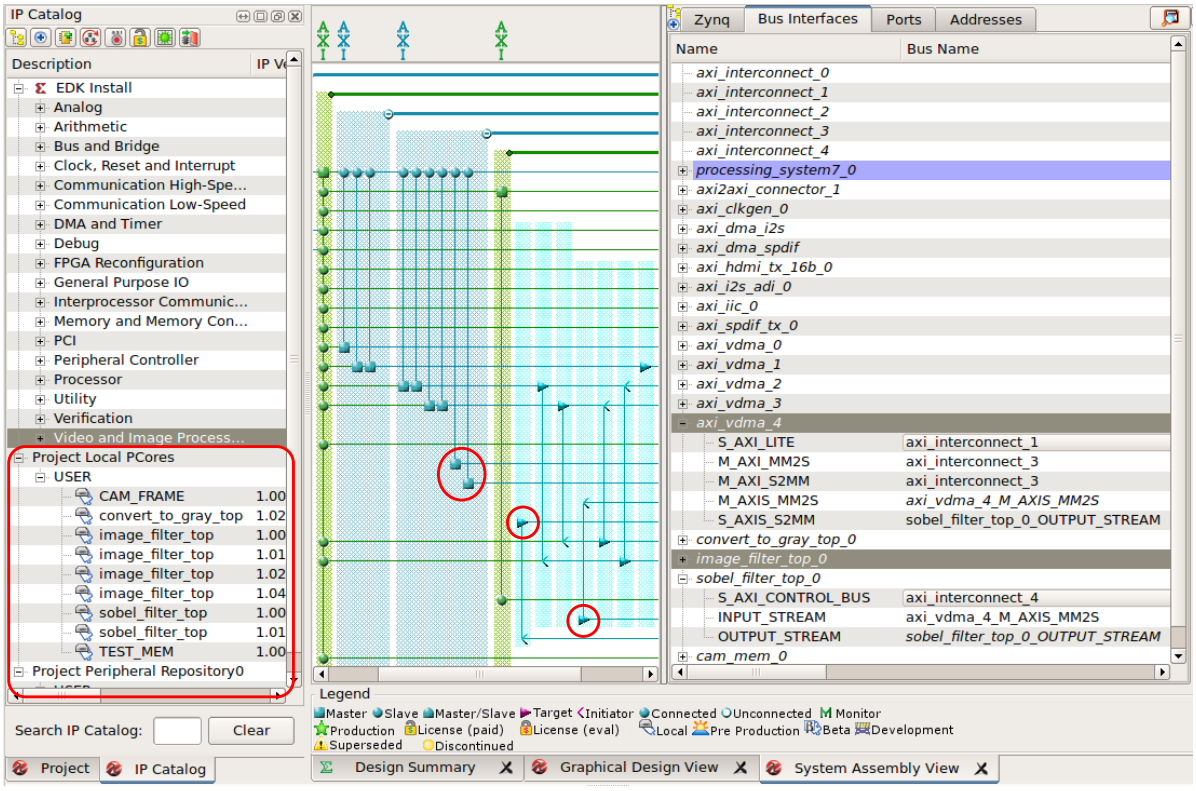


Figure 4.9.: User IPs in Pcore and their Bus Interconnects with Zynq embedded system

memory. In similar manner, integrating each of these IPs through VDMA and AXI interconnect we have a modified system that can utilize hardware IP blocks to accelerate the real time operation of embedded vision application.

The complete hardware architecture capable of executing all the video processing tasks in the Programmable Logic is shown in the Figure 4.10 below. All the IP blocks in the PL are configured and controlled via AXI-Lite interconnect. The video processing blocks are connected to the system through VDMAs. These VDMAs access the DDR memory through 64-bit High Performance ports. The VDMAs have Memory Mapped interface with the AXI interconnect and Stream interface with the video processing blocks. Direction of the arrow in AXI interfaces represents master to slave connection.

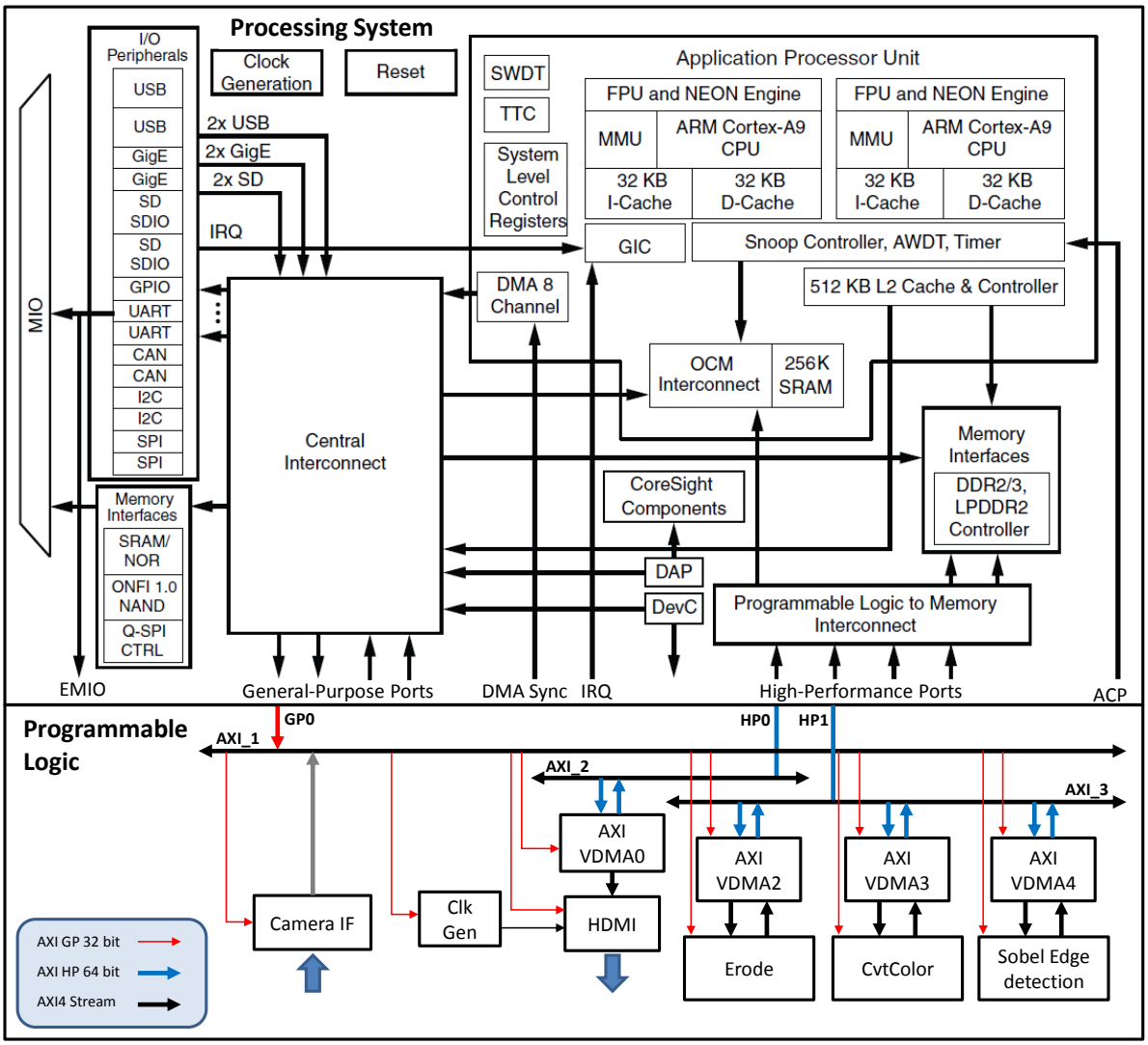


Figure 4.10.: Block diagram of the hardware architecture for video processing

Following the IP integration the software for standalone application can be developed in Xilinx SDK environment. The driver functions generated by Vivado HLS tool while packaging the IP are used to set the parameters for the video processing IPs. The files under path `/impl/pcores/<ip_name>/include` in the Vivado project hierarchy are used to provide SW access APIs. The VDMA blocks are set to operate in 'register direct mode'. The DDR 'frame buffer start address' for the VDMAs, video parameters and DMA control registers are set before starting the video transfer (AXIVDMA 2012). The video data is captured from camera module by the processor over the AXI Lite interface and stored into the DDR memory. The VDMAs read the video stream from DDR to the video processing IP blocks generated by HLS. The processed video stream is buffered in the DDR memory before it is picked up by the HDMI interface for display.

4.6. Summary

A complete embedded system capable of processing video was developed. The basic components for Video I/O i.e camera interface and display interface were integrated into the Zynq SoC as first step toward the embedded vision system development. Developing on the basic video I/O system, a design methodology was developed for IP integration and HW acceleration of vision applications. The design methodology relies on efficient C-to-IP algorithm flow, which benefits from High Level Synthesis technology. Vivado HLS enables quick integration of IP blocks into the system and provides the basis for design space exploration. The three image processing algorithms namely 'Grayscale conversion', 'Erosion' and 'Sobel edge detection' are discussed. They are used for HLS IP block generation and integrated with the Zynq SoC following the presented design methodology. This complete video processing system can be used to evaluate different performance parameters depending on HW and SW mapping of the tasks. Next chapter presents and investigates the results collected with different task mappings on the system presented in this chapter.

5. Results

Hardware IP integration for computationally demanding tasks in embedded vision systems enables the real time processing of high quality video. This advantage comes with a logic area and communication bandwidth trade-off. On one hand, CPU is off-loaded from performing these data intensive tasks. On the other hand, the hardware IP implemented in FPGA fabric consumes the programmable logic and requires the bus bandwidth for transferring data to/from the frame buffers in external memory. Thus, the HW and SW mapping of the sub-tasks is to be carefully determined. The impact of various architecture mappings on CPU load, bus load and FPGA area utilization needs to be qualitatively analyzed. In this chapter, first the schemes for code profiling and bus load monitoring are explained briefly, then the real-time data resulting from different architectures is presented and analyzed.

5.1. ARM Processor Code Profiling

To find the computational bottleneck in the vision algorithm running on the ARM processor, it is necessary to profile the code. This would help to figure out the part of the algorithm that heavily loads the processor. ARM cortex processors provide a set of *performance monitor registers* that can be used for benchmarking and cycle-accurate profiling of the software running on them. The ARM Cortex-A9 core consists of a cycle counter, which can be configured to increment either for every core cycle or for every 64 core cycles. It also has 6 configurable event counters, which can be set to count a chosen event e.g instruction count or branch miss. The performance counters can be configured and accessed through software calls only in privileged mode. To profile a code in software following steps must be followed

1. Reset performance counters
2. Enable performance counters
3. Call function to profile
4. Disable performance counters
5. Read out performance counters

Precautions were taken to cater for the profiling overhead and to check that the performance counters have not overflowed after profiling. In Cortex-A9 cores, the performance counters are configured via the CP15 Performance Monitoring Unit (PMU) registers. There are number of

PMU registers for benchmarking, but here only two relevant registers for processor cycle counters, namely 'Performance Monitor Control Register' and 'Cycle Count Register' are explained. These registers can be accessed using `MCR` or `MRC` commands.

Performance Monitor Control Register (PMCR)

The PMCR register controls the performance monitor including the cycle counter. It is a 32-bit register, as shown in Figure 5.1, and only accessible when user-mode access is enabled. The `D`, `C` and `E` bits are important for our use. The `D` bit is the clock divider bit and selects whether PMCCNTR counts every clock cycle or once every 64 clock cycles. The `C` bit resets the PMCCNTR to zero and the `E` bit enables all counters including PMCCNTR. To access the PMCR, use:

```
MRC p15, 0, <Rd>, c9, c12, 0 ; Read PMCR Register
MCR p15, 0, <Rd>, c9, c12, 0 ; Write PMCR Register
```

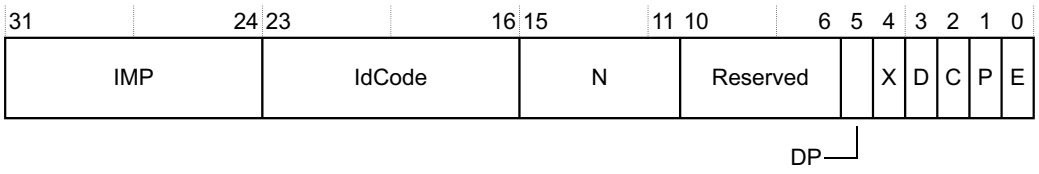


Figure 5.1.: Bit assignment of PMCR

Cycle Count Register (PMCCNTR)

PMCCNTR register counts processor clock cycles. Depending on `PMCR.D` bit it increments either on every clock cycle or on every 64th processor cycle. It is a 32-bit register and can count upto maximum $2^{32} * 64$ processor cycles depending on the configuration. To access the PMCCNTR, use:

```
MRC p15, 0, <Rd>, c9, c13, 0 ; Read PMCCNTR
MCR p15, 0, <Rd>, c9, c13, 0 ; Write PMCCNTR
```

5.2. AXI Communication Load Monitoring

As discussed earlier, when we move the algorithm from SW to a HW IP core, it loads the bus communication. The AXI interconnect to which the HW IPs are connected, can be monitored to estimate their impact on communication load. The 'AXI Performance Monitor' IP core (AXIPM 2012) from Xilinx can be used to measure major performance metrics of the AXI interconnects. It measures bus latency of a specific master/slave (AXI4/AXI4-Stream) in a system, the amount of memory traffic for specific duration and other performance metrics. The

'Performance Monitor Unit' provides hardware for counting events associated with the AXI bus transactions. One core can be connected with up to 8 monitored AXI agents, which may either be an AXI4 MM or AXI4-Stream interconnects. All the AXI signals of the agents are connected to the monitoring slot of the core. A core can have upto 10 metric counters each of which can be configured to monitor a specific performance parameters.

A Performance Monitor IP core is added to our system in PL for monitoring the AXI interconnect, with which the HW IPs are attached. The 'Slot0' of the IP core is connected to the AXI interconnect 'AXI_3' that accesses the memory for frame trasfer to HW IP blocks. The configuration registers for Performance Monitor IP core are configured via its AXI-Lite interface. This core is generated with four metric counters that could be configured to provide various performance parameters of 'AXI_3' interconnect. Following steps are followed to get the metric count for memory transfers:

1. Reset the global clock counter and metric counters by writing 0x00020002 onto the Control Register (0x300).

2. Configure the metric selection register (0x44) for the required slot metrics. A value of (0x03020100) would set the metric counters to provide write transaction count, read transaction count, write byte count and read byte count for 'Slot0' respectively.

3. Enable the metric counters and if required global clock counter by writing 0x00010001 in the Control Register(0x300).

4. Start DMA/Memory transfers.

5. After the transfers are complete, read the metric counters (0x100, 0x110, 0x130 and 0x140) to get the counter values.

5.3. Results and Analysis

To compare and contrast the effects of architecture mapping on high data-rate video processing application, we start with a complete SW application running on the ARM processor. This application mapping is shown in first data flow graph is Figure 5.2. The only hardware elements in this implementation are for acquiring and displaying the image. They are present as peripheral cores in the PL. The input side of the video subsystem captures the a video stream from camera and places the pixel data into DDR memory. The video out peripheral handles the transfer of pixel data from DDR memory onto a video display.

With no hardware acceleration i.e. if all the video processing is executed on ARM, this complete processing has a throughput of less than 3 frames per second for a 640x480 video stream. This frame processing rate is significantly reduced for HD video streams. Thes compute intensive SW video processing functions are ported on the FPGA fabric in form of HW accelerators. The second data flow graph in Figure 5.2 shows the new mapping of the algorithms after HLS

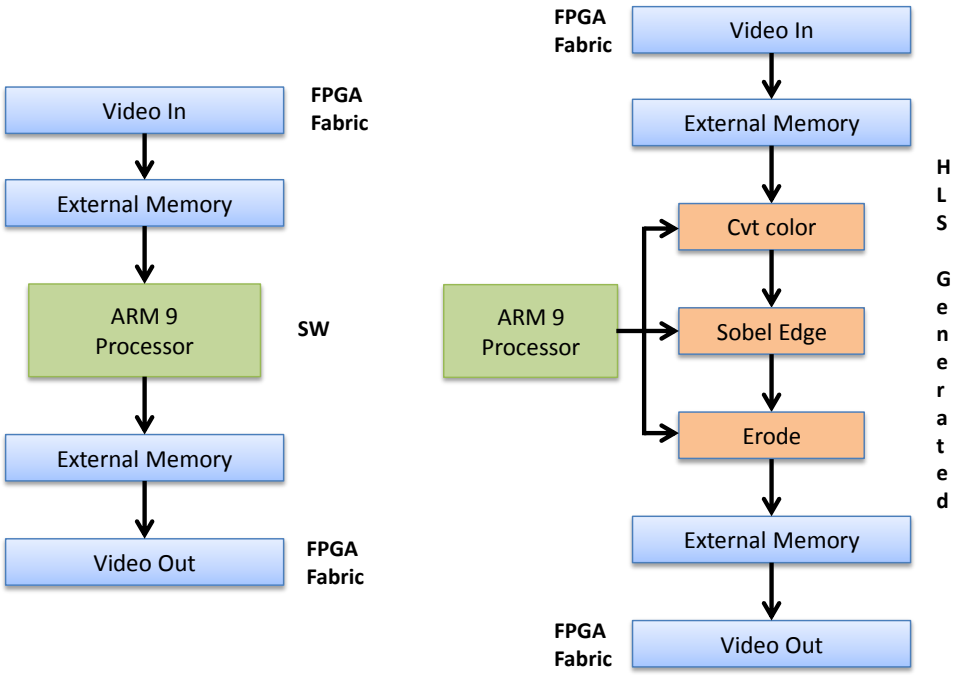


Figure 5.2.: Comparison of application flow on ARM processor vs FPGA fabric

IP generation. The core computations of the algorithm, which were previously executing on the ARM processor, are compiled into multiple Vivado HLS-generated IP blocks and mapped into FPGA fabric. This results in potential reduction of processing time and enhances the frame per second processing capability of the system.

Various configurations for design space exploration were used to benchmark the selected algorithms against computational load on ARM processor and communication load on AXI bus. Pure SW, mixed HW and SW with multiple accelerator IPs and Pure HW algorithm implementation were benchmarked. Six different configurations with distinct task mappings were assessed against our performance parameters. These configurations are listed in Table 5.1. The CPU processing cycles per frame are calculated with help of performance counters, as explained in previous section, to evaluate the CPU load. To find the AXI bus load, AXI

Config ID	**Configuration**
a	All algorithms in SW
b	HW Grayscale; SW Sobel, Erode
c	HW Grayscale, Sobel; SW Erode
d	HW Grayscale, Erode; SW Sobel
e	HW Sobel, Erode; SW Grayscale
f	HW Grayscale, Erode, Sobel; SW none

Table 5.1.: Configurations for different HW and SW mapping of algorithms

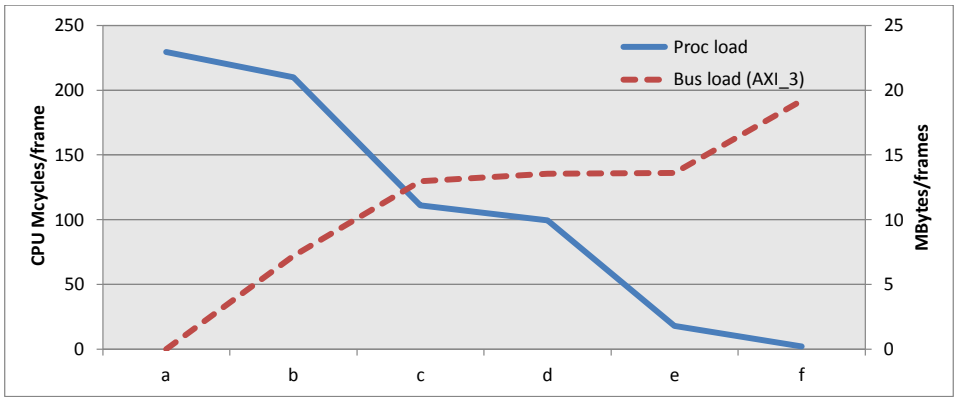


Figure 5.3.: Comparing the CPU load and bus load for different task mappings

performance monitor IP is used that calculates the AXI bus transactions data when different hardware processing IPs are attached to it. The processor load and bus load corresponding to each configuration are presented in the plot shown in Figure 5.3.

It can be seen that if complete algorithm is processed in software it requires around 230M processor cycles to process one complete frame. With our ARM processor running at 667MHz this corresponds to a frame rate of less than 3 fps, which is way less than acceptable as video is streamed at 15 fps. When processing a High Definition video technology, the processing time could fall below 1 fps. The code profiling reveals that Erode and Sobel operations heavily load the processor. The hardware IPs for each of these were generated as per the design methodology discussed in chapter 4. In the graph, it can be seen that as the video processing tasks are mapped into the HW, the ARM processor gets offloaded. One trade-off for this gain is that the communication load on the AXI bus as well as on the DDR bus starts rising. Here the bus load of the non-system AXI bus that connects the HW IPs with the SoC is presented as sum of write and read bytes being transferred over it.

It is important to note that as one video processing task is moved to the HW, it would equally load the bus regardless of the amount of processing done within this task. It is due to the fact that a task would require the VDMA to transfer the frames over the AXI bus. This asks for a careful evaluation of the computation versus communication load for each task. In the 'bus load' curve it can be noticed that 'Grayscale' conversion task offloads the ARM processor slightly but increases the bus load significantly (configuration a and b). On the other hand, moving the 'Sobel edge detection' task into the PL results in a significant reduction in processor load while adding the same amount of bus load as 'Grayscale'. A single processing task in one HW accelerator attached to one VDMA results in high bus bandwidth requirement. This approach was followed to evaluate the the impact of individual tasks on performance parameters. A more practical approach would be to perform all these tasks in single chain of operations inside one IP attached to one VDMA. This would reduce the requirements on memory bus bandwidth.

The plot in Figure 5.4 shows a comparison of the processing times of the three video processing operations. The processing time for SW as well as for HW implementation of these tasks is

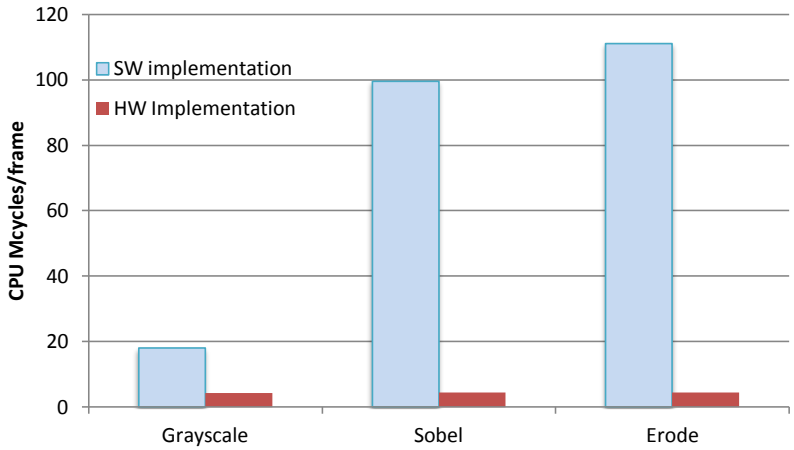


Figure 5.4.: Processing time of different video processing tasks in SW and estimated HW processing time with reference to CPU cycles per frame

presented with reference to ARM processing cycle count. The time taken by HW accelerator to complete one operation on one frame is calculated from the estimates provided by Vivado analysis tool. With a pipelined implementation in hardware, the effective time to process the frame is reduced to one cycle per pixel. Thus the time to process one frame depends on the frame resolution. The figure shows a drastic reduction in processing time of the frame operations when they are mapped to hardware IP blocks generated by Vivado HLS. This gain is achieved due to the fact that image operations can be highly parallelized and availability of dedicated resources in hardware. The overall processing time for one frame is reduced from 230M to around 7M processor cycles and the system now has the computational capacity to process up to 100 frames per second for VGA video stream. However, when processing video at high frame rates, the communication overhead must also be carefully considered.

Area is the another important metrics and is a measure of how many hardware resources are required to implement the design. The logic area consumed after hardware synthesis by 'Grayscale conversion', 'Erode' and 'Sobel edge detection' is shown in Figure 5.5. It is important to consider the area consumed by the hardware accelerator when mapping a task to hardware. From this figure we can see that chaining together the subsequent processing operation could save area in programmable logic. This savings comes from reduction in logic area due to IP interfacing overhead. The logic area consumed by VDMA in hardware is significant and represents the area overhead of attaching an independent HW IP. The Vivado HLS tool can also perform optimization on the design through directives to satisfy timing and area goals. It is important to note that using one hardware IP block, attached to one VDMA core, to perform a chain of operations would save significant amount of hardware resources in the PL.

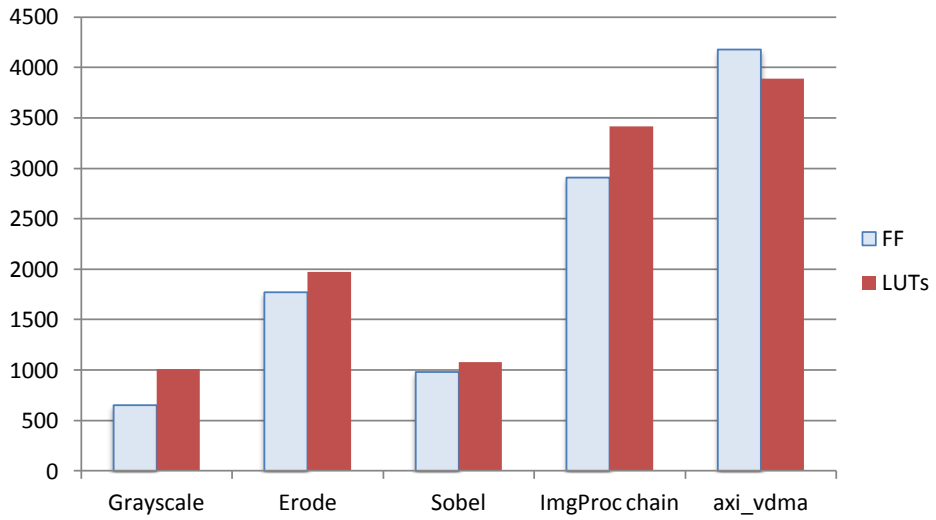


Figure 5.5.: The area utilization by HLS generated IP blocks after synthesis

5.4. Summary

This chapter presented the results of Design Space Exploration (DSE) for selected vision algorithms. It compares different architectures against processor computation load, bus communication and hardware area utilization. It shows that vivado HLS technology provides quick solution for IP integration and DSE. The results presented in this chapter show that HLS generated IP blocks not only accelerate the vision algorithms on embedded systems, but they can also be used for generating real-time performance metrics. This performance metrics guides the system designer to chose an optimized implementation for any user specific vision algorithm. The performance metrics depicts the trade-offs for different task mapping in terms of area, timing and communication. The mapping of these algorithms have to be carefully selected to get most favorable results for these performance parameters within user constraints.

6. Conclusion and Future Work

In this thesis the development of application specific SoC based on rapid IP integration was presented. It provides an insight into methodologies to deal with IP reuse challenges for future SoCs. The thesis presents a design methodology to efficiently generate and integrate the IP blocks into the embedded system. The methodology presented was demonstrated for embedded vision applications on Zynq FPGA.

In the beginning, different options to implement the vision applications on Zynq SoC were evaluated. ARM processor on the Zynq PS was configured for OS-based application processing as well as for standalone application processing. IP integration with each of these configurations was examined according to application requirements. Due to its appropriateness with regards to vision applications standalone system was chosen for developing IP integration methodology.

The Zynq All Programmable SoC available on Zedboard was used for hardware and software Co-design application development. The necessary video I/O subsystems were developed to provide the basic setup for Design Space Exploration. HLS technology enables rapid integration of customized IPs for application specific systems. The Vivado HLS from Xilinx provides a methodology to move computationally intensive applications from ARM processor into FPGA logic. Application was first written in software and computational bottleneck was extracted via code profiling. The part of algorithm to be mapped into hardware was written in high level language (C/C++) and synthesized into RTL implementation by Vivado HLS tool. The Vivado HLS provides the RTL implementation as standardized IP core with driver functions. These IPs were integrated into the HW using standard interconnects and configured in SW through provided driver functions. Thus, computer vision applications targeted at Zynq platform can take advantage of HLS technology. C, C++ code for ARM processing can be rapidly developed while compute intensive functions are accelerated using high performance FPGA fabric.

The implementation scheme and design methodology presented in this work provides a platform for further design exploration. A set of image processing algorithms were implemented on this platform using proposed design methodology. Different architectures were assessed against processor's computation load, interconnect's communication load and FPGA area utilization. The results show that compute intensive vision algorithms can be accelerated through HLS generated IP blocks with communication and FPGA area trade-offs. For our presented algorithms significant improvement in frame processing rate is achieved. Thus HLS proves to be an enabling technology for design space exploration at higher design abstraction level while providing actual performance estimates for optimal decision making.

The work presented in this thesis aims to support the model-driven IP integration platform based on generative graph grammar. This platform formalizes the DSE through generative

graph grammar rules. This thesis provides the inputs to model-driven platform in the form of HW accelerator and their SW access functions along with implementation results. These inputs would help in identifying the design bottlenecks during design space exploration.

In the future, OS based implementation scheme discussed in Chapter 3 could be further explored. The Vivado generated IP blocks can be attached to the ARM processor running OS and performance metrics can be compared to a standalone application. As CPU consumes huge amount of core cycles in transferring data from camera module to DDR, one improvement in the current design could be to use DMA to relieve the CPU of data movement task. Xilinx's 'Video In to AXI-4 Stream' IP core could prove helpful in this regard. The methodology presented in this thesis could also be extended to generate an optimized HW IP library for application specific systems. The IPs from the library could be utilized to build any application specific SoC.

Bibliography

ADV7511 (n.d.): ADV7511 Xilinx Evaluation Boards Reference design, Analog Devices, Inc.
http://wiki.analog.com/resources/fpga/xilinx/kc705/adv7511

AXIPM (2012): LogiCORE IP AXI Performance Monitor v3.00a Product Guide, Xilinx Inc.
http://www.xilinx.com/support/documentation/ip_documentation/axi_perf_mon/v3_00_a/pg037_axi_perf_mon.pdf

AXIRG (2011): UG 761 AXI Reference Guide, Xilinx Inc.
http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

AXIVDMA (2012): LogiCORE IP AXI Video Direct Memory Access v5.00.a Product Guide PG020, Xilinx Inc.
http://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v5_00_a/pg020_axi_vdma.pdf

Digilent (2013): Embedded Linux Hands-on Tutorial – ZedBoard, Digilient Inc.
https://www.digilentinc.com/Products/Detail.cfm?Prod=ZEDBOARD

ESTRM (2012): UG111 (v14.2) Embedded System Tools Reference Manual, Xilinx Inc.

EVA (2013): Embedded Vision Alliance.
www.embedded-vision.com

HLSUG (2013): Vivado Design Suite User Guide UG902, Xilinx Inc.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf

IP-XACT (2010): IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows, IEEE.
http://standards.ieee.org/getieee/1685/download/1685-2009.pdf

ITRS (2011): International Technology Roadmap for Semiconductors 2011 Edition System Drivers, http://www.itrs.net/links/2009itrs/.

OpenCV (n.d.): Open Source Computer Vision Library.
http://opensource.org

OV7670 (2005): OV7670 Implementation Guide, OmniVision Technologies Inc.

OV7670 (2006): OV7670/OV7171 CMOS VGA (640x480) CAMERACHIP^{TM} with OmniPixel Technology, OmniVision Technologies Inc.

PG, ADV7511 (2012): Low-Power HDMI Transmitter Programming Guide, Analog Devices, Inc.

Xillybus (n.d.): Xillinux: A Linux distribution for the Zedboard, Xillybus Ltd.
`http://xillybus.com/xillinux`

XillybusGS (n.d.): Getting started with Xillinux for Zynq-7000 EPP, Xillybus Ltd.

ZedBoard (2013): Zynq Evaluation and Development Board.
`www.zedboard.org`

ZedHW (2013): ZedBoard Hardware User's Guide, Avnet Electronics.