Fall 1-31-2015

An embedded system supporting dynamic partial reconfiguration of hardware resources for morphological image processing

Gyana Ranjan Sahu

New Jersey Institute of Technology

ABSTRACT

AN EMBEDDED SYSTEM SUPPORTING DYNAMIC PARTIAL RECONFIGURATION OF HARDWARE RESOURCES FOR MORPHOLOGICAL IMAGE PROCESSING

by

Gyana Ranjan Sahu

Processors for high-performance computing applications are generally designed with a focus on high clock rates, parallelism of operations and high communication bandwidth, often at the expense of large power consumption. However, the emphasis of many embedded systems and untethered devices is on minimal hardware requirements and reduced power consumption. With the incessant growth of computational needs for embedded applications, which contradict chip power and area needs, the burden is put on the hardware designers to come up with designs that optimize power and area requirements.

This thesis investigates the efficient design of an embedded system for morphological image processing applications on Xilinx FPGAs (Field Programmable Gate Array) by optimizing both area and power usage while delivering high performance. The design leverages a unique capability of FPGAs called dynamic partial reconfiguration (DPR) which allows changing the hardware configuration of silicon pieces at runtime. DPR allows regions of the FPGA to be reprogrammed with new functionality while applications are still running in the remainder of the device.

The main aim of this thesis is to design an embedded system for morphological image processing by accounting for real time and area constraints as compared to a statically configured FPGA. IP (Intellectual Property) cores are synthesized for both static and dynamic time. DPR enables instantiation of more hardware logic over a period of time on an existing device by time-multiplexing the hardware realization of functions. A comparison of power consumption is presented for the statically and dynamically reconfigured designs. Finally, a performance comparison is included for the implementation of the respective algorithms on a hardwired ARM processor as well as on another general-purpose processor. The results prove the viability of DPR for morphological image processing applications.

AN EMBEDDED SYSTEM SUPPORTING DYNAMIC PARTIAL RECONFIGURATION OF HARDWARE RESOURCES FOR MORPHOLOGICAL IMAGE PROCESSING

by
Gyana Ranjan Sahu

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

January 2015

APPROVAL PAGE

AN EMBEDDED SYSTEM SUPPORTING DYNAMIC PARTIAL RECONFIGURATION OF HARDWARE RESOURCES FOR MORPHOLOGICAL IMAGE PROCESSING

Gyana Ranjan Sahu

Dr. Sotirios Ziavras, Thesis Advisor Date
Professor of Electrical and Computer Engineering & Associate Provost for Graduate Studies, NJIT

Dr. Durga Madhab Misra, Committee Member Date
Professor of Electrical and Computer Engineering, NJIT

Dr. Roberto Rojas Cessa, Committee Member Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author:	Gyana Ranjan Sahu
Degree:	Master of Science
Date:	January 2015

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, U.S.A, 2015

- Bachelor of Technology in Electrical Engineering,
  National Institute of Technology, Rourkela, India, 2010

Major: Computer Engineering

Dedicated to my late grandfather, my first teacher and friend in the journey of life, my father who has always supported me and has been a never ending source of inspiration for me and my family.

Finally, this thesis is dedicated to all those who believe in the richness of learning.

ACKNOWLEDGMENT

First and foremost, I would like to thank my advisor, Dr. Sotirios Ziavras, for all his guidance, support, and encouragement throughout my thesis work. I have learned a tremendous amount under his supervision, for which I am deeply grateful. His knowledge and assistance have been invaluable. I would also like to thank my committee members Dr. Durga Madhab Misra and Dr. Roberto Rojas Cessa for their feedback.

I would also like to express my gratitude to my friends Dario Gomez and Yaojie Lu, also members of the Computer Architecture and Parallel Processing Laboratory (CAPPL), who have assisted and guided me in the initial stage of the thesis.

Lastly, I would like to thank my family and my friends for their loving support, encouragement and confidence during my thesis work.

TABLE OF CONTENTS

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF SYMBOLS

⊕	Dilation
⊖	Erosion
€	Belongs to
∩	Intersection
●	Closing
○	Opening

LIST OF TABLES

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

1.1 Motivations and Objectives

As per [1], FPGAs are known to outperform General-Purpose Processors (GPPs) when they are used to implement a specific function in a hardware design. The main aim of this thesis is to present an efficient design of an embedded system targeting at morphological image processing applications. The said embedded system was implemented on the Xilinx AP-SoC FPGA Zedboard. Usually, the FPGA is treated as a slave component in such a reconfigurable system, when required; the complete FPGA is configured to offload the main processor, in this case an ARM processor. With the development of dynamic partial reconfiguration (DPR) features for some FPGAs, only part of the FPGA can be partially reconfigured at runtime when needed. By doing this, such a partially reconfigurable system can provide hardware adaptation for real time functionality.

In designing the embedded system, the image processing cores were built in IP and were stored in the configuration memory. Depending upon the algorithm to be executed, DPR allows the FPGAs to be reconfigured dynamically at runtime to match up changes in application behavior. By leveraging dynamic partial reconfiguration, not only do we achieve a significant reduction in the required floor area but can also gain in terms of power reduction. As for all chips, the power dissipation of FPGAs embodies static and dynamic ingredients. A study released by Xilinx reports that the static power rises substantially below the .25 micron feature size [16].

The major contributors to the dynamic energy consumption of an embedded platform FPGA are the processor and IP cores, and the auxiliary and I/O blocks. Any

FPGA asset consumes both static and dynamic power in the active state, and only static power in standby when its clock signal is disabled. Any power consumption could be eliminated by shutting down the power supply to an asset to put it into the sleep state. However, current FPGAs do not support this feature for specific assets. By configuring only parts of the FPGA to actually employ in computations, we can significantly cut down the dynamic power consumption.

1.2 Overview

The thesis is organized in seven chapters. Chapter 2 discusses the theory of reconfigurable computing. The philosophy of GPP designs, which is based on the von-Neumann computing paradigm, is presented. It also discusses the Application-Specific Integrated Circuit (ASIC) method of implementing designs on silicon. ASICs are the fastest and most optimized designs when it comes to match up application needs. But the fixed structure of ASICs renders them inflexible since they cannot be changed once fabricated. However, when there is a need for flexibility and high performance computational power, DPR has major advantages. Reconfigurable computing, also known as spatial computing, is generally implemented on FPGAs. The same chapter discusses at length the architecture of FPGAs and their ability to reconfigure the design by changing their SRAM configuration. We also present an entire design flow using FPGA-based logic synthesis tools and hardware/software codesign strategies. The chapter concludes by presenting various types of runtime DPR reconfiguration for FPGAs.

Chapter 3 briefly discusses the design flow for logic synthesis on Xilinx FPGAs supporting DPR. It assumes that IP cores used as hardware accelerators are already synthesized and have been converted into partial bitstreams. These bitstreams are available as binary files to be stored in the memory of the host processor.

Chapter 4 discusses the theory of mathematical morphology and various operators which will be implemented on the user programmable host. These different operators are programmed in MATLAB. Only basic operators are discussed as the main aim of this thesis is to design the embedded system for image processing applications. Complex applications can be run for high performance once the core algorithms are synthesized in hardware.

Chapter 5 summarizes the Vivado High-Level synthesis tool that was used to synthesize our design. The optimization effort for our application is included as well.

Chapter 6 presents performance and power analysis results, while Chapter 7 contains conclusions.

CHAPTER 2

RECONFIGURABLE COMPUTING

2.1 von-Neumann Computing Paradigm

The Hungarian mathematician John von-Neumann (VN) showed that a computing machine could have a simple, fixed hardware structure for the execution of any kind of computation, when given properly programmed control. Since then, the VN machine paradigm has been universally accepted as the standard computing architecture for conventional processing. The general architecture of a VN machine is shown in Figure 2.1 and consists of the following major components:

- A memory for storing data and programs which is sequentially accessed for operands and instructions. To improve performance, Harvard architectures contain two parallel accessible memories for storing program and data separately.
- A control unit featuring a program counter that holds the address of the next instruction to be executed.
- An arithmetic and logic unit (ALU) and CPU registers along with a data path for instruction execution.

A VN machine executes programs sequentially, instruction by instruction. At each step of program execution, the next instruction is fetched from the memory at the specified address stored in the program counter. The fetched instruction is then decoded and executed; a result may be written back into the memory or register.

The main advantage of the VN computing paradigm is its flexibility, because it can be used to program any given algorithm. Each algorithm to be run on a VN machine has to be coded sequentially according to VN rules. That is, ‘the algorithm must adapt itself to the hardware'. Because of the VN rules, that imply sequential execution, VN computation is also referred to as 'temporal computation'.

Generally, algorithms have some form of inherent instruction-level parallelism (ILP). They can be executed faster by taking advantage of ILP and often data-level parallelism (DLP). Since all algorithms executed on a pure VN machine are run sequentially; many algorithms cannot be executed to their best possible performance. Modern GPPs exploit ILP and/or some DLP to speed-up execution. Additionally, modern uni-core or multi-core architectures also exploit thread-level parallelism (TLP) to further enhance the performance.

However, if the class of algorithms to be executed is known in advance, then an adaptive processor could be modified to better match the computation paradigm of that class of applications.

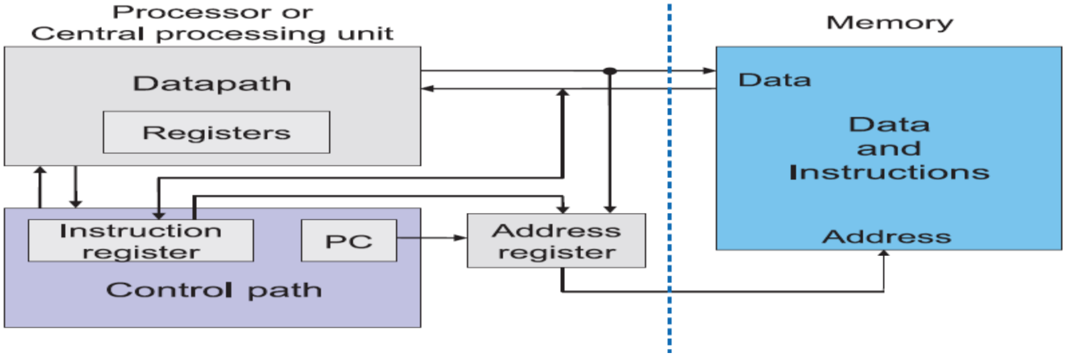


Figure 2.1 General structure of a von-Neumann Computer.
Source: [2]

Individual processor performance doubled about every 18 months until 2003, following the doubling of transistors as per Moore's law. However, memory improvements did not keep up. As processor speeds increased, the processors spend more time in the idle mode, waiting for data to be fetched from memory. No matter how fast a given processor can work, in effect it is limited by the rate of memory transfers.

Some of the approaches to overcoming this von-Neumann memory bottleneck are:

- Caching: The storage of frequently used data in a special area (usually SRAM), so that it is more readily accessible than if it were stored in the main memory.
- Prefetching: Moving some data into the cache before it is requested to speed up access in the event of a request.
- Multithreading: Managing multiple requests simultaneously in separate threads.
- New types of RAM: For example, DDR SDRAM, which provides an output on both the rising and falling edges of the system clock, rather than on just the rising edge. This doubles the transfer rate.
- Out of order execution: Instructions are dispatched to instruction queues called reservation stations and allowed to complete out of program order. But the instructions are committed to memory or the register file as per the order they appear in the original sequential program order.

2.2 Application Specific Processors and ASICs

If an algorithm or computation is fixed for an application, it can be optimized for the best possible performance by designing specialized hardware. In this case, we say that the hardware design matches the application. The hardware or the processing unit which is designed for the specific application is called an Application Specific Processor (ASIP). These, ASIPs are further classified into Domain Specific Processors (DSPs) and ASICs. In an ASIC, the entire application or computation is executed directly on the hardware and offers the best possible performance. In ASICs, the instruction fetch-based cycles are eliminated and hence the need for sequential execution is overcome. Moreover, in ASICs the computation is executed faster by taking advantage of the inherent parallelism in the program. ASICs use a spatial approach to execute an application since all the functional blocks needed for the computation are implemented in hardware. Hence, this kind of computation is also called ‘Spatial Computing’. ASICs are generally implemented with CMOS technology in a single chip. For example, in cell phones certain communication protocol algorithms are implemented on ASICs to provide real-time functionality.

The main drawback of ASICs is that, after fabrication the circuit design cannot be altered. This is in contrast to GPPs where, by changing the software instructions, the functionality of the system is altered on demand without changing the hardware. However, the downside of this flexibility is that the performance suffers, and is far below that of an ASIC. However, due to higher engineering costs resulting from longer design cycles and the increasing costs of design tools, the overall cost of ASICs may be prohibitively high.

2.3 Reconfigurable Computing

There exist two main attributes to characterize processors: flexibility and performance. Since, VN computers can execute any kind of algorithm, they are considered highly flexible. However, they do not execute the algorithm in ways that achieve the highest performance.

ASICs, on the other hand, allow the functional units to be fabricated on a chip. High performance is possible because ‘the hardware is adapted to the application’. If we consider two scales, one for the performance and the other one for the flexibility, then the VN computers can be placed at one end and ASICs at the other end.

Ideally, system designers desire the flexibility of GPPs and the performance of ASICs for the same device. To this extent, a hardware device is needed that can provide spatial computing for the application while simultaneously adapting to the algorithm. Such a hardware device is called a reconfigurable processing unit (RPU).

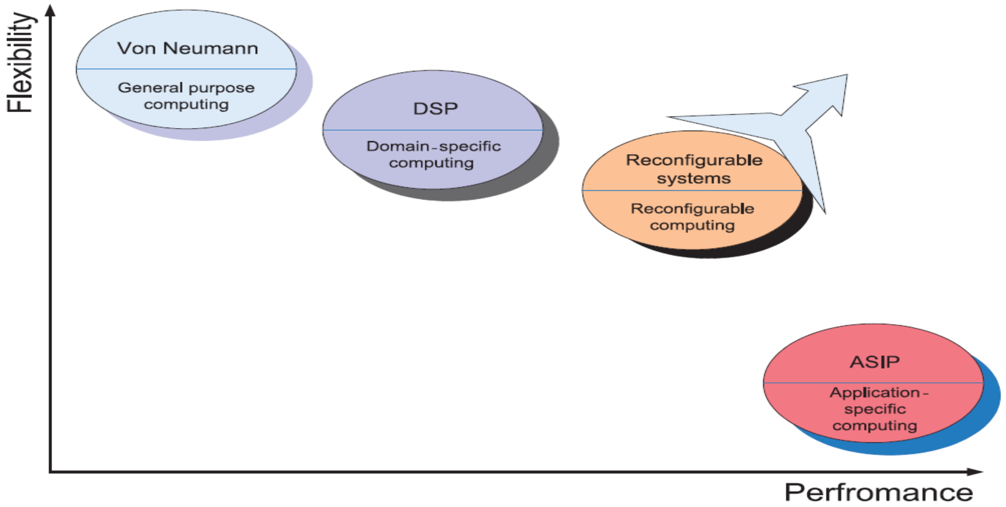


Figure 2.2 Comparison of various types of computers in terms of performance and flexibility.
Source: [2].

The main idea behind a reconfigurable device is to take advantage of the application inherent parallelism to achieve the best possible speedup. The structure of reconfigurable devices is changed by modifying all or part of the hardware at compile or run time, by downloading a configuration bitstream into the device. The most popular devices that support this type of reconfiguration are FPGAs.

2.4 FPGA Architecture

FPGAs were first introduced in the 1985s by Xilinx Inc. as programmable logic devices. The FPGA architecture consists of three main parts: a set of programmable logic cells (CLBs), a programmable interconnect network for routing information between input/output blocks, and a set of input and output cells around the device. About 90% of the FPGA area is made of programmable interconnects; the rest of the FPGA is made of logic blocks [3,6]. Figure 2.3 shows a generic FPGA architecture. Additional, resources such as memory, DSP blocks, embedded processors, etc., may be available on a FPGA depending on the vendor.

2.4.1 CLB (Configurable Logic Block)

The CLB is the basic logic unit in a FPGA. Exact numbers and features vary from device to device, but every CLB commonly consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry (MUX, etc.), and flip-flops [2]. The switch matrix is highly flexible and can be configured to handle combinatorial logic, shift registers or RAM. Altera Corp. refers to these logic blocks as Logic Array Blocks (LABs). In addition, each CLB contains several look-up tables (LUTs).

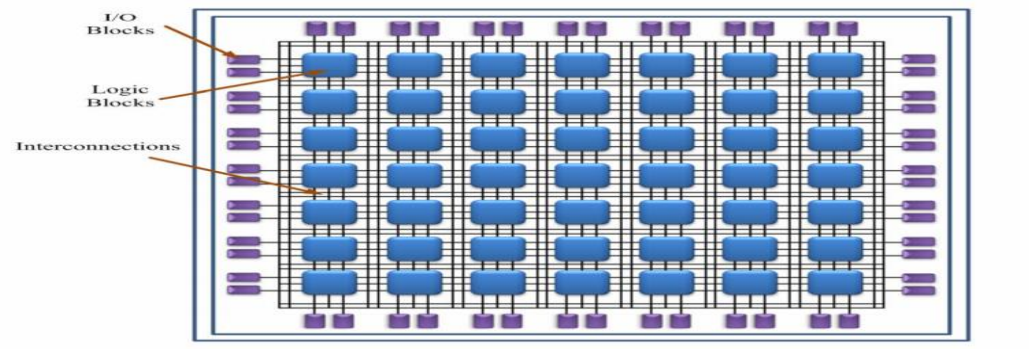


Figure 2.3 Matrix structure of a FPGA consisting of CLBs and interconnects.
Source: [2].

The LUT is used by most of the FPGA vendors mainly because an n-input LUT is capable of implementing of any n-input logic function. In other words, a LUT is a small memory that directly stores the truth table to realize a fundamental digital circuit. The number of CLBs and their configuration depend on the size and type of the FPGA. Figure 2.4 illustrates Xilinx CLB components.

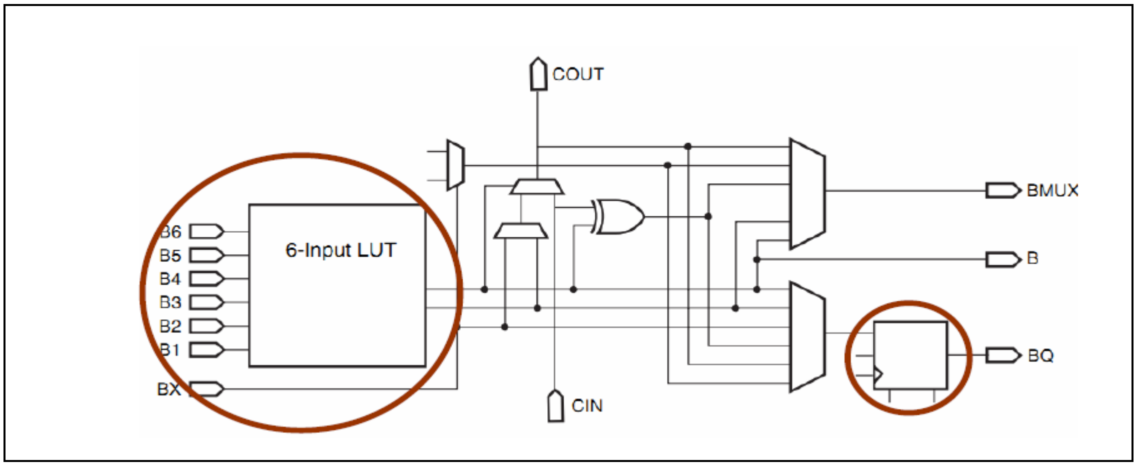


Figure 2.4 Structure of a CLB consisting of LUT and multiplexers.
Source:[2].

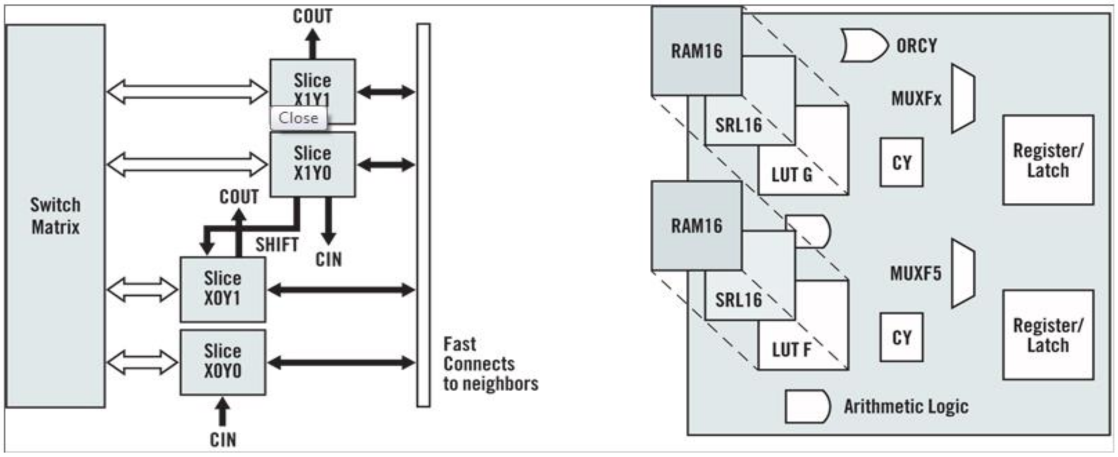


Figure 2.5 Structure of SLICE cells and the switch matrix for a Xilinx FPGA.
Source: [4].

2.4.2 Interconnects

While the CLBs provide logic capability, flexible interconnect routing routes [2] the signals between CLBs and to/from I/O ports. Routing comes in several flavors, from interconnecting CLBs, to fast horizontal and vertical long lines spanning the device, to global low-skew routing for clocking and other global signals. The design software hides interconnect routing from the user, unless specified otherwise.

2.4.3 Select IO (IOB)

Input/Output blocks provided on FPGAs are generally programmable for input or output. They are used to propagate signals from one logic element to another. Xilinx FPGAs provide support for various I/O standards [4]. In Xilinx 7 series FPGAs, each I/O bank contains fifty SelectIO pins which can support different I/O standards (single ended and differential ended I/O standards) [5].

2.4.4 Memory

Embedded BRAM (Block RAM) memory, which is available in most FPGAs, allows for on-chip memory to be instantiated in the design. These on-chip memories allow the designer to store coefficient and other buffer data.

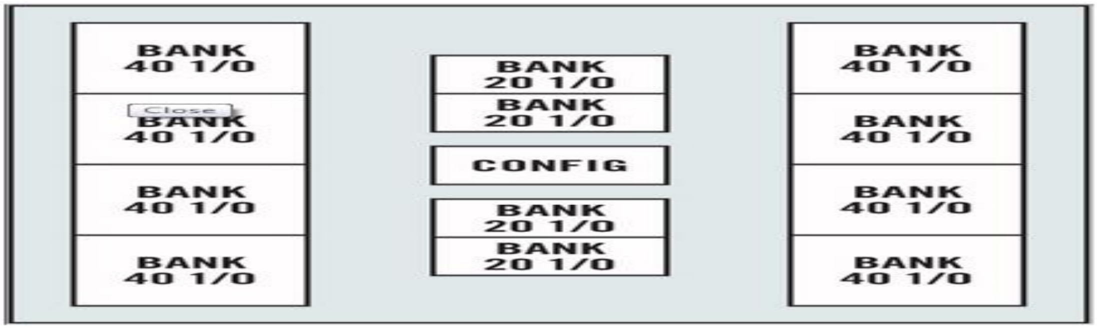


Figure 2.6 On-chip memory arranged in memory banks and available as BRAMs on Xilinx FPGAs.
Source:[4].

Xilinx FPGAs provide up to 10Mbits of on-chip memory in 36kbit blocks that can support true dual-port operation. Figure 2.6 shows the on-chip memory arranged as memory banks in the FPGA chip.

2.4.5 Clock Management

Digital clock management is provided by most FPGAs in the industry (all Xilinx FPGAs have this feature). The most advanced FPGAs from Xilinx offer both digital clock management and phase-looped locking that provide precision clock synthesis combined with jitter reduction and filtering [4]. This lets the designer operate the hardware module at different clock frequencies. Sometimes different IP cores instantiated in the design need to operate in a particular fixed clock domain and then synchronize the system with other modules running in a different clock domain. For example, in building a HDMI video pipeline, the design needs to operate at 148.5MHz while letting the other modules operate at 200MHz.

2.4.6 SRAM-based Configuration

Static memory is the most widely used method of configuring FPGAs. FPGAs can be configured any number of times by changing the bits stored in the static SRAM memory cells. The output of a memory cell is directly connected to another circuit and the state of the memory cell continuously controls the circuit being configured. Although using volatile SRAM cells has some disadvantages, the advantages far out-weight the disadvantages. Some disadvantages of SRAM-cell FPGAs are:

- The SRAM configuration memory consumes a noticeable amount of power, even when the program is not changed.
- The bits in the SRAM configuration may be susceptible to flipping.
- A large number of bits must be set in order to program an FPGA. Each combinational logic element requires many programming bits and each programmable interconnection point requires its own bit.

An example of using SRAM-controlled switches is illustrated in Figure 2.7. It shows two applications of SRAM cells for controlling the gate nodes of pass-transistor switches, and for control lines of multiplexers that drive logic block inputs. When both SRAM cells store one, the output of one logic block is connected to the other through the multiplexer. Whether an FPGA uses pass-transistors or multiplexers, or both, depends on the particular product.

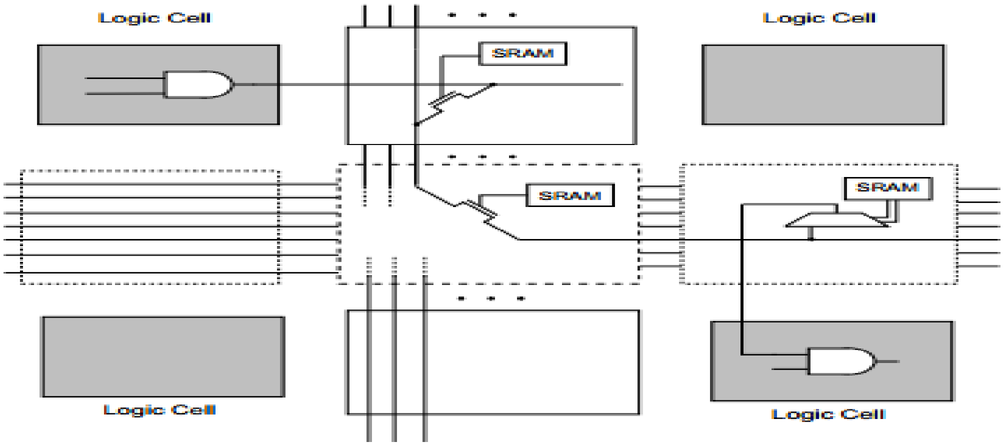


Figure 2.7 Dynamic configuration of logic cells by changing the SRAM configuration. *Source:* [4].

2.4.7 Logic Implementation on FPGAs

The steps required to implement an application or algorithm on an FPGA are commonly referred to as design flow. Programming an FPGA is significantly different from programming a GPP. Rather than generating sequences of instructions, we generate the hardware components that will be mapped at different times to the available resources. As per algorithm needs, the hardware resources will be generated for spatial computation. The generation of such components is called logic synthesis. It is an optimization process whose goal is to minimize some cost function aimed at producing, for instance, the fastest hardware with the smallest amount of resources and the smallest power consumption.

The algorithm or function is generated using either a schematic editor, a hardware description language (HDL), or a finite state machine (FSM) editor. HDLs, such as VHDL, Verilog and System C, are the most commonly used tools to generate the RTL of the hardware. Other than VHDL and Verilog, there are other hardware synthesis tools provided by different vendors which allow the synthesis of hardware from high level languages like C/C++. High-level synthesis or C-based design flows are relatively simpler and provide a straightforward approach for hardware synthesis from functions that are run in software on a GPP. Moreover, these high-level synthesis tools provide a faster approach for simulation and debugging of hardware before synthesis. For example, the high-level synthesis tool provided by Xilinx is called Vivado HLS; Altera uses OpenCL.

2.4.8 Place and Route

After functional simulation, the design can be compiled and optimized. It is first translated into a set of Boolean equations. Technology mapping is then used to implement the functions with the available modules in the function library of the target architecture. In case of FPGAs, this step is called LUT-based technology mapping, because LUTs are the modules used in the FPGA to implement the Boolean operators. The result of logic synthesis is called netlist. A netlist describes the modules used to implement the functions as well as their interconnections. For the netlist generated in the logic synthesis process, operators (LUTs, Flip-Flops, Multiplexers, etc.) should be placed on the FPGA and connected together through routing. These two steps are normally achieved by CAD tools provided by the FPGA vendors. After the placement and routing of a netlist, the CAD tools generate a file called a bitstream. A bitstream provides the description of all the bits used to configure the LUTs, the interconnect matrices, the state of the multiplexer and the I/O of the FPGA. Full and partial bitstreams (for partial FPGA reconfiguration) can now be stored in a memory to be downloaded into the FPGA fabric.

This thesis uses Vivado HLS to synthesize most of the morphological functions; they were implemented on the FPGA fabric by a different tool the Vivado Design Suite. The entire process of the FPGA design flow is shown in Figure 2.8.

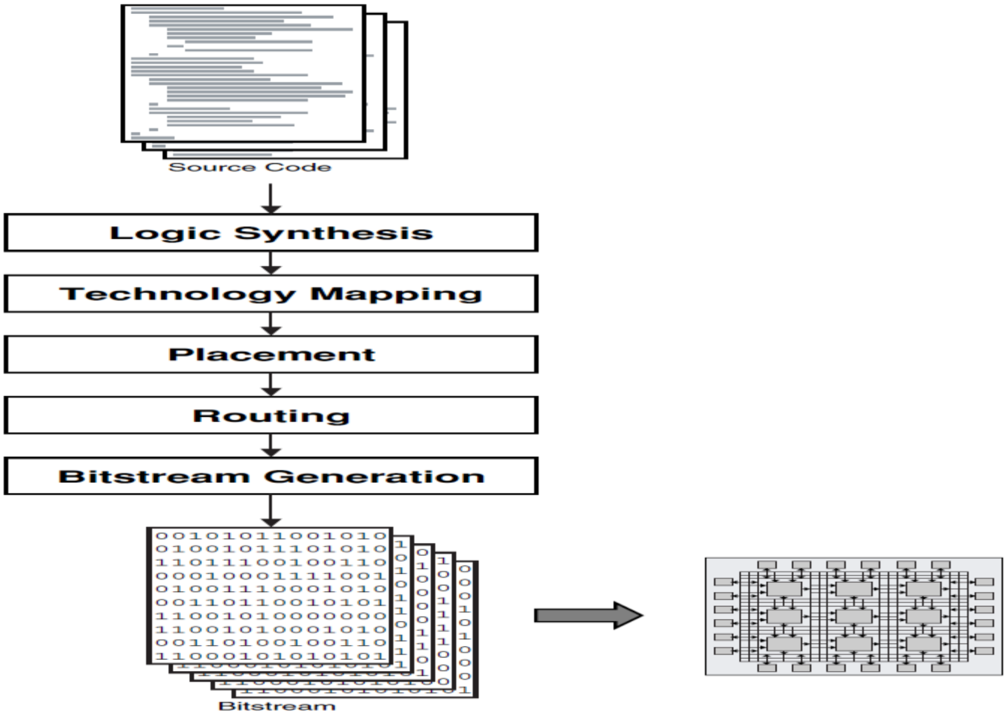


Figure 2.8 Entire design flow steps, such as logic synthesis, place, route and bitstream generation to configure a FPGA.
Source [2].

2.5 Hardware/Software Codesign for Platform FPGA

Currently, many FPGA vendors provide FPGA platforms in the form of a System-on-a-Programmable Chip (SoPC). Xilinx refers to its design architecture as “All Programmable-System on a Chip (SoC)” whereas Altera refers to its design architecture as “System on a Programmable Chip”. These devices, apart from containing an FPGA fabric, also include other system level components such as memory, ADC converters, USB ports, HDMI ports, etc. These SoCs enable extensive system level differentiation, integration, and flexibility through hardware, software, and I/O programmability.

System-on-a-chip FPGAs include embedded processors (hard or soft), bus protocols, memory and other IPs which provide an opportunity for system designers to develop high performance systems. Hard processors are microprocessors that have been diffused in the silicon that contains an FPGA. For example, the Virtex II chip from Xilinx includes a PowerPC processor whereas the Zedboard APSoC includes a dual-core ARM processor. Soft processors are microprocessors that are created out of the FPGA gate array and can be configured to suite a particular application. An example is the MicroBlaze 32-bit RISC processor available from Xilinx as an IP.

HW/SW codesign meets system level objectives by exploiting the synergism of hardware and software through a concurrent design. It attempts to manage the simultaneous development of hardware and software, it requires the use of multiple discrete design flows at the implementation level (the system is specified and analyzed, and then the hardware specification is passed to the hardware designers while the software specification is passed to the software designers).

In this thesis, the implementation is done on a Xilinx Zynq-7000 series SoC contained in the Zedboard. The Zedboard includes a dual-core ARM Cortex-A9 MPCore hard Processing System (PS) that interacts with tightly coupled 7 series 85K Programmable Logic (PL) cells. Other key peripherals on the board include:

- 512 MB DDR3
- 256 Mb Quad-SPI Flash
- 4 GB SD card

- USB-JTAG port
- 10/100/1000 Ethernet
- USB OTG 2.0 and USB-UART
- HDMI output supporting 1080p60 with 16-bit, YCbCr, 4:2:2 mode color
- VGA output (12-bit resolution color)
- 128x32 OLED display
- 8 user LEDs, 7 push buttons and 8 DIP switches.

2.6 Types of Reconfiguration and Granularity

Depending upon the granularity of the device, the reconfiguration can be classified as fine-grain or coarse-grain. Fine-grain reconfigurable devices are modified at a very low level of granularity. For instance, the device can be modified to add or remove a single inverter or a single two-input NAND gate. Fine-grain reconfigurable devices are mostly programmable logic devices (PLDs). However, fine-grain architectures may not be efficient because of large routing areas and poor routablity. Most of the reconfigurable architectures are built as coarse-grain reconfigurable arrays with large path widths. Computational data paths generally have widths greater than one bit and, hence, more area efficient designs are possible by using custom reconfigurable data paths of width greater than one. Several basic computing blocks are grouped into a coarse-grain CLB.

CHAPTER 3

DESIGN FLOW FOR PARTIAL RECONFIGURATION

3.1 Introduction

This chapter presents a detailed description of the design flow for generating partial configuration bitstreams targeting at Zynq AP-SoC devices. The final implementation of the modules is done on a Zedboard which contains a Zynq 7 series device. The process of generating a static configuration was explained in the preceding chapter. In partial reconfiguration, the design flow consists of generating both the static configuration and the partial reconfiguration bitstreams. In our case, the static configuration consists of generating a morphological image processing pipeline with functionality that remains fixed. Different morphological operations and algorithms are explained in the next chapter, and the synthesis of the algorithms is discussed in Chapter 5. For now, we assume that we have the synthesized algorithm and generate the RTL model. The next section describes the rest of the design flow for partial reconfiguration.

3.2 System Overview

This section describes our system for implementing two morphological image processing accelerators in Programmable Logic (PL) by using partial reconfiguration to load the desired functionality on demand. The Zynq-7000 AP-SoC integrates a dual-core ARM Cortex-A9 based processing system (PS) and programmable logic (PL) in a single device. The proposed design makes use of both the PS and PL portions and demonstrates how it is best to separate control (mapped onto the PS) and data path (mapped onto the PL). The PL implements a powerful, high-performance image processing pipeline that consists of input, core processing, and output stages (shown in Figure 3.1).

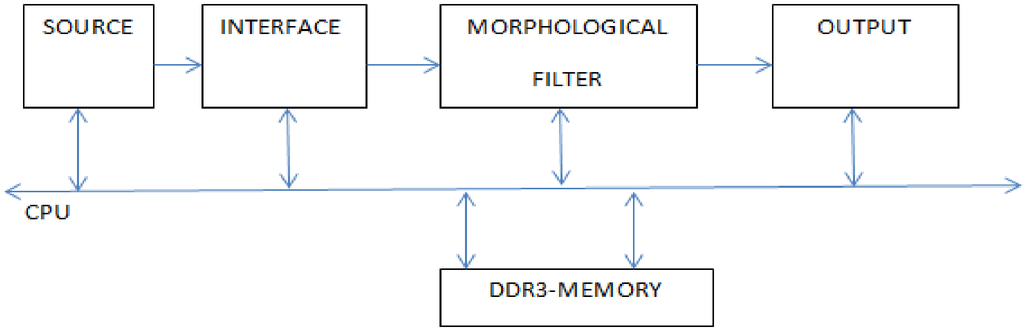


Figure 3.1 Proposed design for the morphological image processing pipeline.

The PS is used to configure the individual IP cores inside the PL and to control the data flow. The first morphological IP core is a dilation operation that increases the grayscale intensity of the image as per a structural element (SE) used in this process; the second morphological IP core is an erosion operator that does the opposite.

Figure 3.2 shows a comparison of original, dilation-processed, and erosion-processed images. The RTL for both morphological IP cores is generated from a C-algorithm description using the High-Level Synthesis tool Vivado HLS. The system overview of the PS section is shown in Figure 3.3. The PS is configured with the following I/O peripherals enabled: USB0, SD0, UART1, and GPIO. All I/O peripheral interfaces are configured for multiplexed I/O. Interrupt signals are connected to the DMA channel and the morphological IP cores.

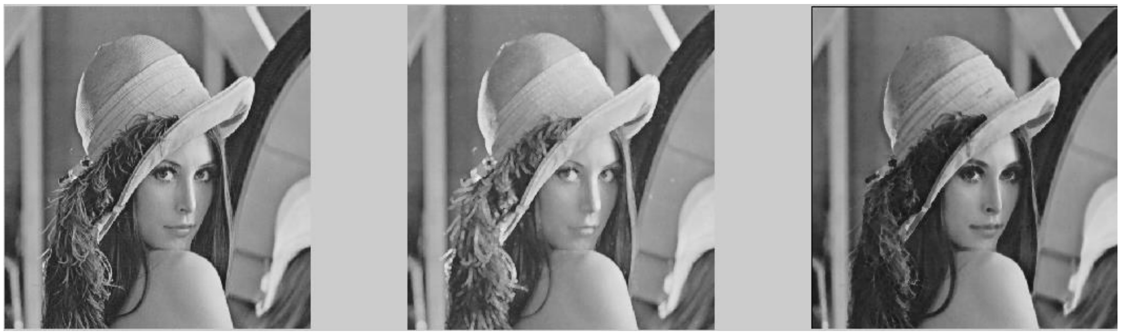


Figure 3.2 Comparison of original, dilated and eroded Lena images.

The General Purpose Master Port GP0 is used to configure and control memory-mapped IP cores via the AXI4-lite interface. The HP0 High Performance Ports are connected via the AXI4: the core processing pipeline is connected to the HP0 read/write channels; the input pipeline is connected to the HP0 write channel; the output pipeline is connected to the HP0 read channel. A PS internal clock generator provides a 100 MHz clock to the PL which sets the clock domain for the rest of the modules.

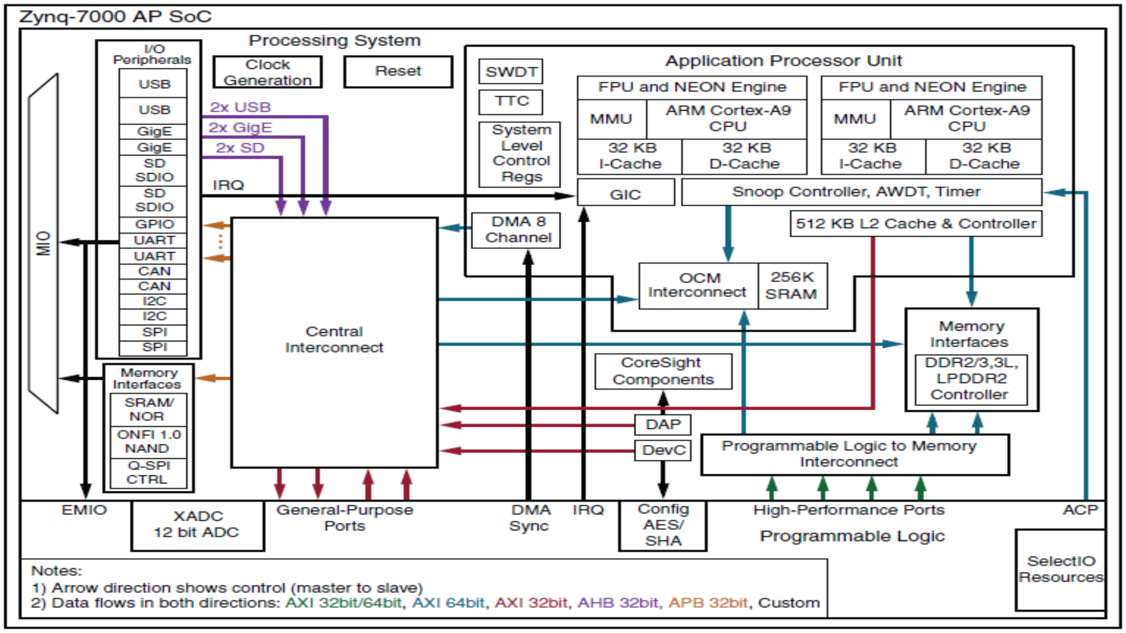


Figure 3.3 Zynq 7000 AP SoC system overview.
Source:[7].

3.3 Device Configuration Interface

The device configuration interface (DevC) includes the methods and procedures for initializing and configuring the PL under PS software control [7]. The DevC consists of a set of control/status registers and three main functional modules. The PS accesses the APB registers to control the three independent modules in the DevC which are the AXI-PCAP bridge, security management module and XADC interface. If an AES encrypted bitstream is used for configuration, then the AXI-PCAP bridge with the DMA is used by the PS to decrypt the bitstream in order to configure the FPGA. In this thesis, the PCAP (Processor Configuration Access Port) mode of reconfiguration is chosen as the interface for DPR. Xilinx provides access to all the registers in the DevC using a header file, Devcfg.h, and necessary function calls to specify the mode of configuration and DMA transfer calls. In this thesis the lower level functions have been further abstracted through a function call to the "XDcfg_TransferBitfile" function which takes in a pointer to the Devcfg, the starting address of the bitfile, and the word length of the bitfile. The starting address can be obtained by downloading the bitstream to a fixed location using the XMD tool or by using a FAT file system to download the bitstream from the SD card to a fixed location in memory. Figure 3.4 shows the block diagram of the DevC interface.

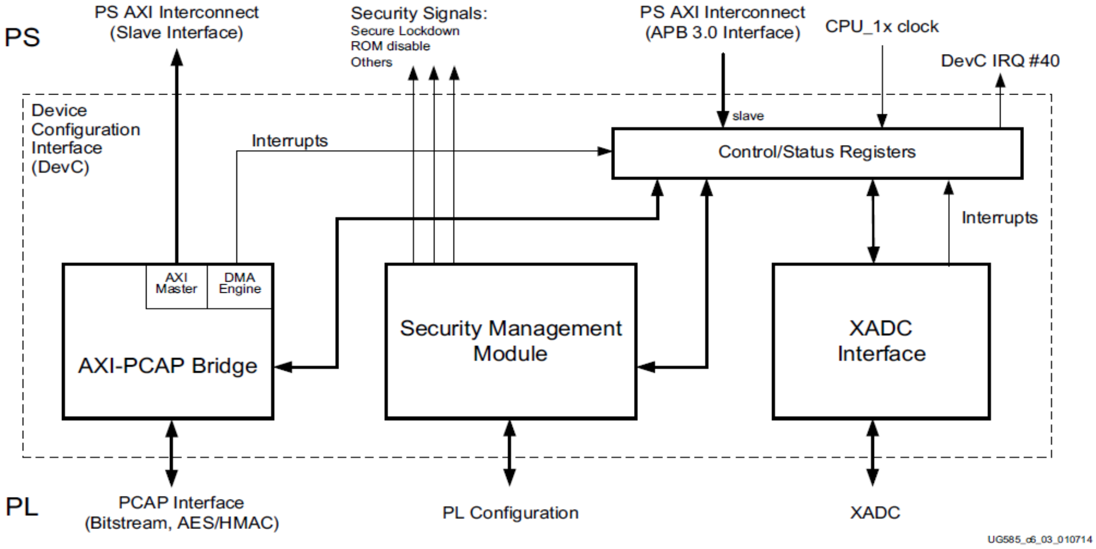


Figure 3.4 DevC Block diagram.
Source: [7].

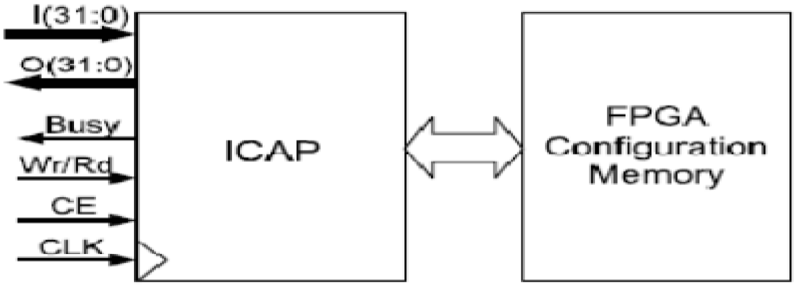


Figure 3.5 ICAP primitive on Xilinx FPGA.
Source: [8].

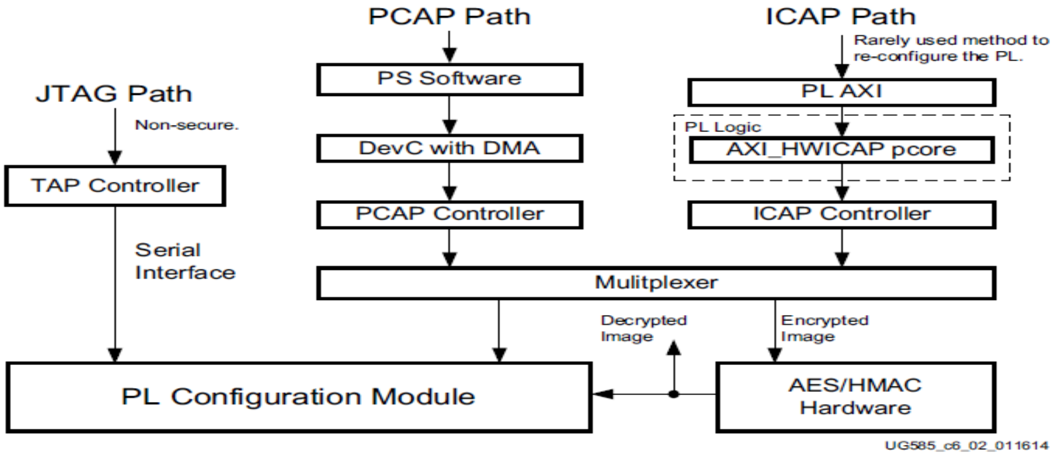


Figure 3.6 PL programming paths on Zynq 7000 devices.
Source: [7].

3.4 Configuration Ports and Interfaces

The PL can be configured by the PS in the secure or non-secure mode. The PL can also be configured by the TAP controller on the JTAG chain in the non-secure mode. The interfaces which allow the configuration of the PL are:

1. PS AXI-PCAP Interface [7]: The PL can be configured by downloading the bitstream to the PCAP (Processor Configuration Access Port) through the DevC interface in secure and non-secure modes. In PCAP mode of configuration the partial bitstreams are stored in a SD card and transferred to the DDR memory by calling a FAT file system function. From the memory the PL is configured by the PS section by downloading the bitstream to the PCAP through the DevC interface. Using this interface, the device can be configured at run-time to support DPR. The AXI-PCAP bridge converts 32-bit AXI formatted data to the 32-bit PCAP protocol and vice-versa. A transmit and receive FIFO buffers the data between the AXI and the PCAP interface. The 32-bit PCAP interface is clocked at 100 MHz and supports 400 MB/s download throughput for non-secure PL configuration and 100 MB/s for secure PL configuration where data is sent only every 4th clock cycle. To transfer data across the PCAP interface a DevC driver function needs to be called. The driver takes care of setting the correct PCAP mode and initiating the DMA transfer. The function call returns only after both the AXI and the PCAP transfers are complete.

2. JTAG TAP Controller: The PL section can also be configured by the JTAG interface through the TAP controller. This can be only done in the non-secure mode.

3. ICAP interface [8]: In this mode of partial reconfiguration, the bitstream is stored in a logic module instantiated in the PL section and dynamically loaded into the configuration memory. As shown in Figure 3.5, ICAP (Internal Configuration Access Port), which is the Xilinx provided hardware interface for partial reconfiguration, interfaces to the configuration memory and furthermore provides parallel access ports to programmable resources. During run-time, a master device (normally an embedded microprocessor) can transmit the partial reconfiguration bitstream from the storage devices to the ICAP to accomplish the reconfiguration process. The complete design, in which the ICAP primitive is instantiated, interfaces the system interconnect fabric in order to communicate with the processor and memories.

3.5 Image Processing Pipeline in the Vivado Design Suite

The morphological image processing pipeline is designed and built using needed IP core blocks. First, the primary morphological operators are built into IP core blocks using the Vivado HLS tool. Algorithm synthesis and optimizations are explained in subsequent chapters. For now, we assume the primary image processing operators are available as the Dilate IP core and the Erode IP cores. The processing pipeline is first designed with static configuration using the Erode and Dilate IP cores. The corresponding reconfigurable image processing pipeline design is shown in Figure 3.7. In the reconfigurable design the Morph IP core is time multiplexed to execute at will the functionality of Erode IP core or the Dilate IP core.

3.5.1 Zynq Configuration

The design consists of a Zynq processor instantiated with the following configuration:

- UART 1 is enabled at the baud rate of 1152Hz.
- All the four DMA channels are enabled.
- SD0 is enabled.
- GPIO is enabled.
- The ARM host processor frequency is set at 666.66MHz.
- The DDR frequency is set at 533MHz.
- Two fabric clocks FLCK0 and FCLK1 are enabled at 100MHz and 150MHz respectively. Two clock frequencies are selected because if system synthesis cannot meet the static timing requirements needed we switch to a slower clock.
- Fabric interrupts from the PL to PS sections used to select up to 16 peripheral interrupts from the PL section.
- Master AXI GPIO 0 is enabled to interface the low throughput data path from the ARM PS to PL.
- The slave high performance port HP0 is enabled to transfer data between the DDR memory and the PL section using a peripheral DMA transaction. The data width is set at 64 bits.

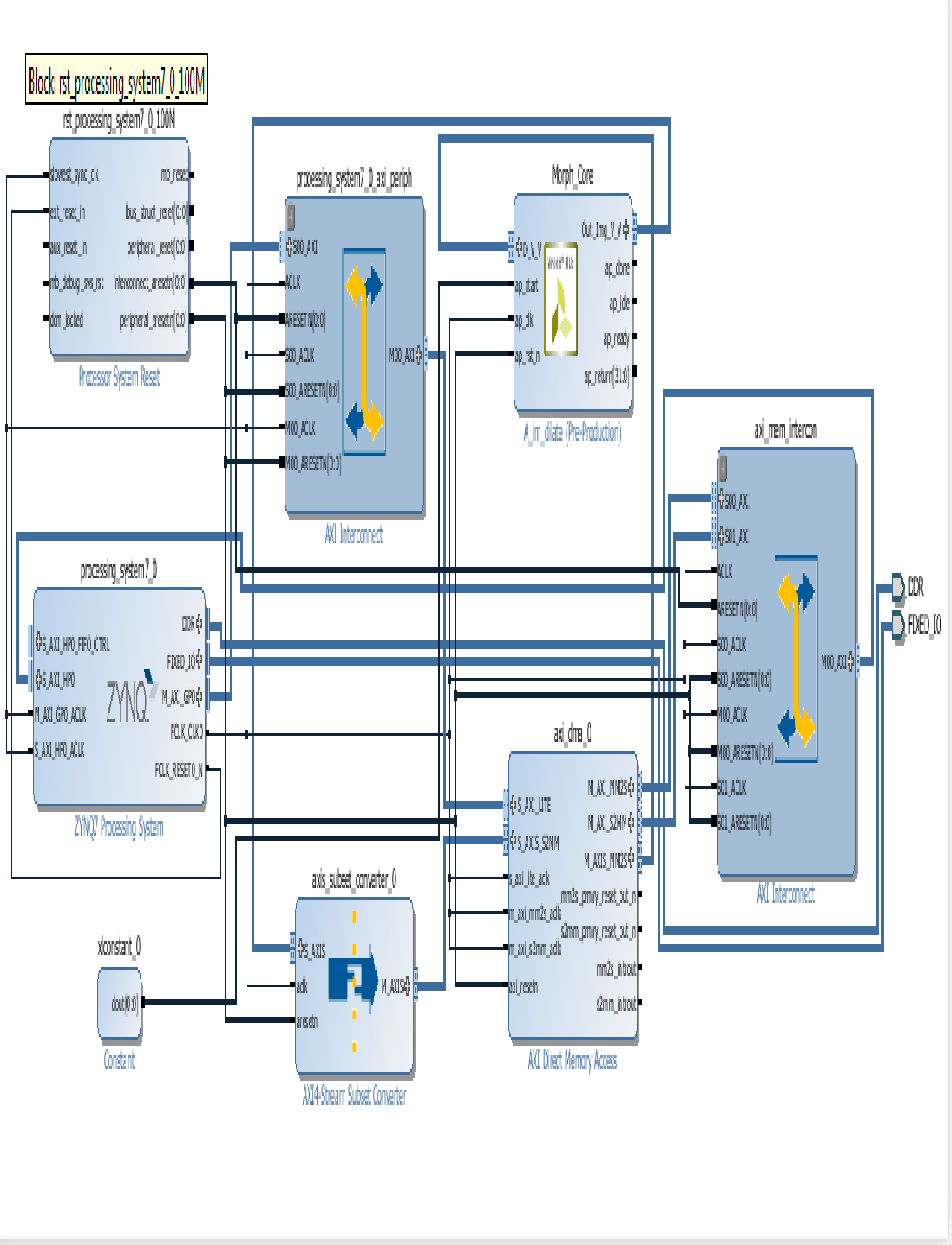


Figure 3.7 Final block diagram of the reconfigurable image processing pipeline.

3.5.2 AXI Interconnect

The IP AXI Interconnect core connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. The AXI interfaces conform to the AMBA AXI4 specification from ARM, including the AXI4-Lite control register interface subset [9]. It is used in this design to connect the Zynq processor with the Morph IP core through the DMA. When connecting one master to one slave, the AXI Interconnect core can optionally perform address range checking. It can also perform data-width conversions, clock-rate conversions, protocol conversions, register pipelining, and data path buffering functions. Each master and slave connection of the AXI interconnect can independently use data widths of 32, 64, 128, 256, 512, or 1024 bits. The AXI interconnect is very robust for our design because it is capable of addressing different AXI bus protocols such as the AXI Lite and the AXI MM (Memory mapped). In our design, the Morph core implements both the AXI streaming and the AXI4lite interface protocols.

3.5.3 DMA Core

The AXI DMA core is a soft Xilinx IP core for use with the Vivado Design Suite. The AXI DMA engine provides high-bandwidth DMA between the memory and the AXI4-Stream-type Morph peripherals. Its optional scatter and gather capabilities can off-load data movement tasks from the host CPU [10]. Initialization, status, and management registers are accessed through an AXI4-Lite slave interface which is connected to the ARM Master GPIO interface via the AXI interconnect. Primary high-speed DMA data movement between the system memory and the stream interface is through the AXI4 Memory Map Read Master to the AXI MM2S Stream Master, and the AXI S2MM

Stream Slave to the AXI4 Memory Map Write Master. The MM2S and S2MM channels operate independently and in the full-duplex mode. Furthermore, the AXI DMA provides byte-level data realignment allowing memory reads and writes to any byte offset location.

The MM2S channel supports an AXI Control stream for sending application data to the Morph IP. For the S2MM channel, an AXI Status stream is provided for receiving user application data from the Morph IP. All these features of the DMA IP core enable high performance data transfer rates for image pixel data between the Morph core and the Zynq PS.

3.5.4 AXI Stream Subset Converter

The AXI4-Stream Subset Converter provides a solution for connecting together slightly incompatible AXI4-Stream signal sets. The IP has configurable AXI4-Stream signals for each interface that allows converting one signal set to another in a consistent manner. All signals can be configured to be removed or added, and additionally the TDATA/TUSER signals can be remapped [11]. Vivado HLS generates the Morph IP cores which are slightly incompatible with the AXI DMA cores. Hence, to ensure proper transfers of data from memory to the Morph cores and back to memory, we need to use the AXI4 Stream subset converter.

3.6 Bottom Up Synthesis and Partial Bitstream Generation

Bottom-Up Synthesis is a design strategy to synthesize by modules. Bottom-Up Synthesis requires that a separate netlist is written for each partition, and no optimizations are done across these boundaries, ensuring that each portion of the design is synthesized independently. Since, we have built the Morph cores separately, we can implement the IP cores using the bottom up synthesis process. Top-level logic for the Morph core is synthesized assuming black boxes for the partitions. Here a partition refers to a logical section of the design, defined by the user at a hierarchical boundary to be considered for design reuse. A partition is either implemented as new or preserved from a previous implementation. A partition that is preserved maintains not only identical functionality but also identical implementation. Partition pins are the logical and physical connection between static logic and reconfigurable logic. Partition pins are automatically created for all reconfigurable partition ports. Here the static logic consists of the portion of the design that remains fixed and the reconfigurable portion refers to the synthesized Morph cores which keep on changing depending upon the higher level algorithms. The tool flow for generating the full and partial reconfiguration can be explained as follows:

- We synthesize the static and reconfigurable modules separately. The files for the static and reconfigurable designs are synthesized and stored as checkpoint files. If the reconfigurable logic consists of more than one reconfigurable module, we should have all the reconfigurable modules synthesized as checkpoint files for that configuration.

- The static design is opened in the netlist pane and consists of reconfigurable modules instantiated as black boxes. Following this, all the constraints for the static design are loaded into the memory.

- In the floor plan of the design, a partition called pblock is created which assigns the pins and ports for the reconfigurable partition. For creating a reconfigurable partition, a property HD.RECONFIGURABLE has to be set on the pblock. It is required that all the reconfigurable modules have the same ports and interfaces.

- The first reconfigurable module is now opened inside the partition assigned for the reconfigurable design. The entire design is reconfigurable, which is now ready for synthesis. Since partial reconfiguration is a licensed feature, the design can only be synthesized if the license is available.

- The design is now placed and routed, and two bitstreams are generated. The first bitstream is the static bitstream which can be used to configure the FPGA at power-on time. The second file is called the partial reconfiguration bitstream which is used for DPR. The file has to be in a binary form so as to be uploaded to the PL section of the FPGA at runtime.

- The same procedure is repeated for other reconfigurable modules and partial binary bitstreams are generated accordingly.

- All the binary bitstreams are uploaded into fixed locations in the DDR memory and function calls are made to transfer the bitstreams to the PL section through the DevC interface.

CHAPTER 4

MORPHOLOGICAL IMAGE PROCESSING ALGORITHMS

This chapter gives a detailed overview of the theory and application of Morphological image processing. All the image processing operators and functions described in this chapter have been implemented in MATLAB, on the ARM-processor (without DPR and PL acceleration), and on the Zynq device with both static and dynamic PL configuration. For simplicity, only MATLAB processed images are presented in this chapter.

Mathematical morphology is a set of tools for extracting image components that are useful in the representation and description of region and shape, such as boundaries, skeletons, etc. Morphological processing is constructed with operations on sets of pixels. Furthermore, morphological operations can be used for filtering, thinning and pruning. Originally, morphological operations were defined for binary images but they can be easily extended for gray-scale images.

Binary images contains pixels having only two values (0 and 1, or black and white). Instead of referring to the colors black and white, we can refer to them as foreground and background. In binary morphological image processing, operations are typically performed on the foreground or background only, and not on all the pixels in the image. Therefore, at any time either the set of black pixels or the set of white pixels will comprise the set of interest because any operation which affects the set of black pixels will also affect the set of white pixels.

.

4.1 Erosion and Dilation

The two basic operations for the construction of morphological operators are dilation and erosion. In the basic Minkowski set operations, the Minkowski addition of two sets F and B is defined as the piecewise vector sum of the elements of F and B:

$$F \oplus B = \{ f+b \mid f € F \text{ and } b € B\} \quad (4.1)$$

The Minkowski subtraction of a set B from a set F is defined as:

$$F \ominus B = \cap F\text{-}b \quad (4.2)$$

where –b represents the reflection of set b and the subtraction is obtained by taking the intersection (∩) with the set F.

In mathematical morphology, the Minkowski addition and subtraction are called dilation and erosion, respectively. Although both operands F and B are sets of the same type, the first operand is commonly interpreted as the image on which the operation is applied, and the second operand is usually a much smaller set called the structuring element (SE). SEs can be either non-flat (continuous variation of intensity is rarely used) or flat [12]. Unless mentioned otherwise, SEs are flat and symmetrical with the origin at the center. Two-dimensional, or flat, structuring elements consist of a matrix of 0's and

1's, typically much smaller than the image being processed. The center pixel of the SE, called the origin, identifies the pixel of interest that is the pixel being processed. The pixels in the SE containing 1's define the neighborhood of the structuring element. Some of the flat structuring elements available in the MATLAB library are:

- Arbitrary.
- Diamond.
- Pair.
- Disk.
- Line.
- Octagon.
- Periodic line.
- Rectangle.
- Square.

In most of the algorithms we consider an SE of disk type with a radius of two units and use the short hand notation SE(2). For the SE(2) of size 5x5, all the co-ordinates which are at a distance of 2 are marked as 1 and the others as 0. For gray-scale images, erosion and dilation are defined as follows:

4.1.1 Erosion

The erosion of an image f by a flat structuring element b at any location (x, y) is defined as the minimum value of the image in the region coincident with b when the origin of b is at (x, y) [12]. Therefore, the erosion at (x, y) of an image f by a structuring element b is given by:

$$[f \ominus b](x,y) = \min_{(s,t)\in b}\{(x+s,y+t)\} \qquad (4.3)$$

Here, the ordered pair (x,y) denotes the co-ordinates in the image f and (s,t) denotes the co-ordinates of the structuring element b.

The image co-ordinates x and y are incremented through all values required so that the origin of b visits every pixel in f. To find the erosion of image f by b, we place the origin of the structuring element at every pixel location in the image. The erosion is the minimum value of f from all values of f in the region of f coincident with b. Since, erosion replaces the current pixel with the minimum value from the neighborhood, the resultant image is darker than the original one.

4.1.2 Dilation

The dilation of an image f by a flat structuring element b at any location (x, y) is defined as the maximum value of the image in the window outlined by b^ = b(-x,-y) with the origin of b^ being at (x,y) [12]. That is

$$[f \oplus b](x,y) = \max_{(s,t)\in b}\{(x-s,y-t)\} \qquad (4.4)$$

where (s,t) are the co-ordinates of structuring element b.

The algorithm is similar to the one for erosion except for using maximum instead of minimum, also, the structuring element is reflected about the origin. Since dilation replaces the current pixel with the maximum value from the neighborhood, the effects are opposite to that of the dilation algorithm. That is, the resultant image after dilation with a flat SE is brighter than the original one.

In developing the algorithms for dilation and erosion, the SE of disk type and radius 2, i.e., SE (2), is used on images of size 512x512. The morphological operators of erosion and dilation were run on an Intel i5 quad-core processor which is configured to run at a maximum frequency of 2.50GHz.

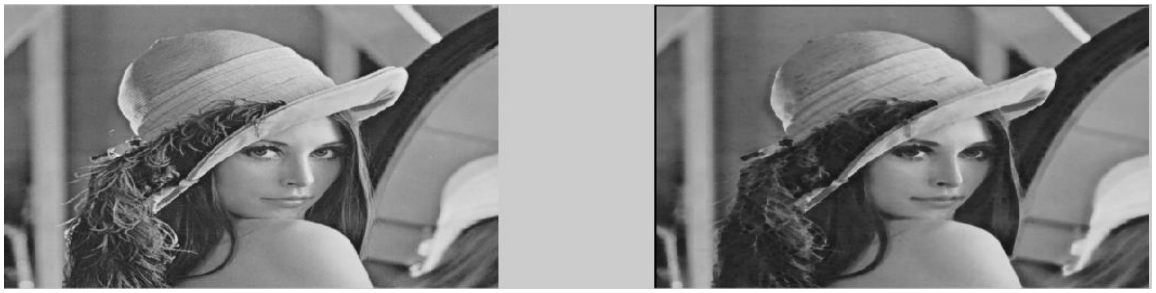
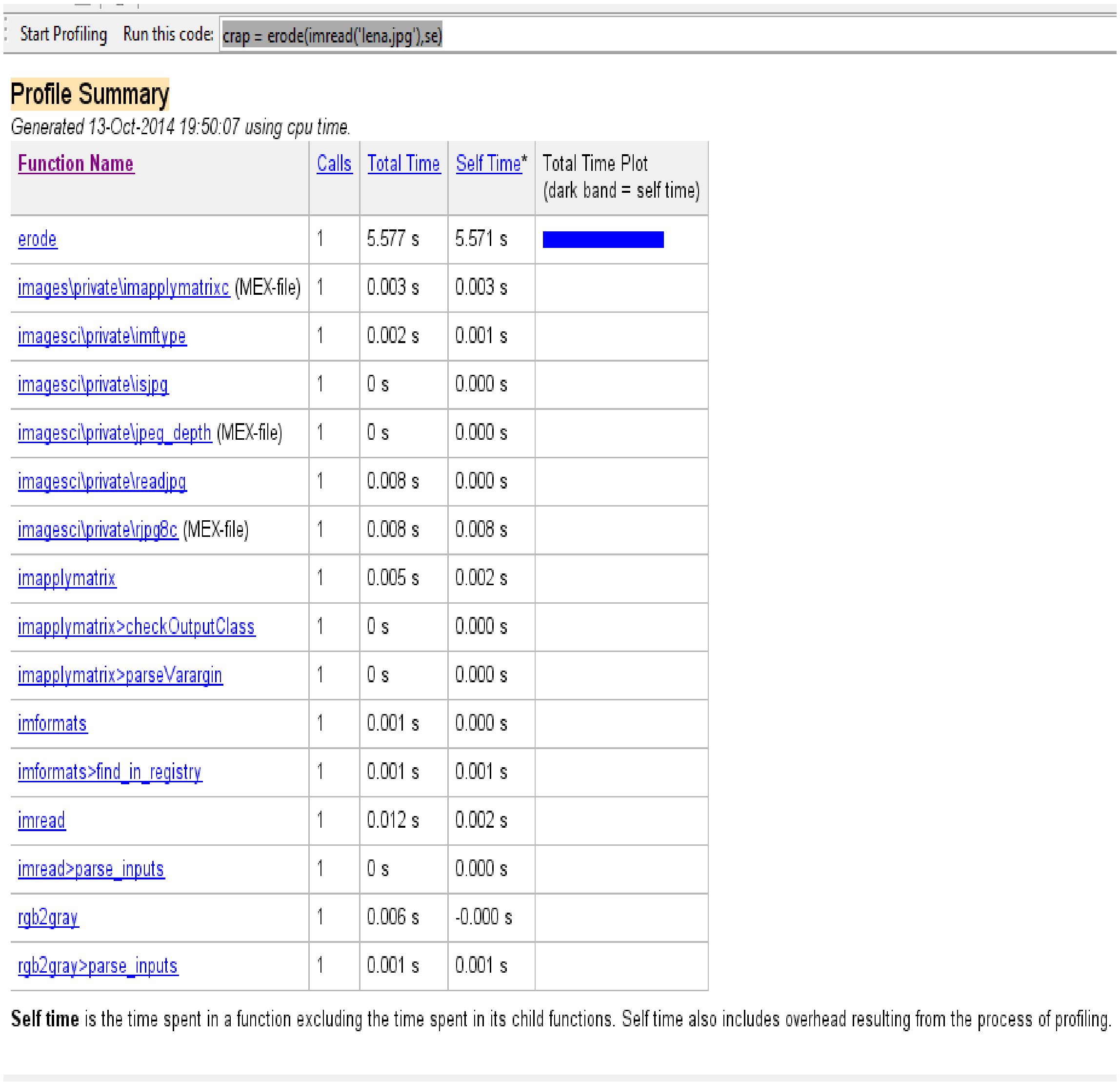


Figure 4.1 Original Lena image and the eroded image for SE(2).

The morphological operator Erode was first implemented as a MATLAB function and its run-time was profiled through the MATLAB profiler. The results are shown in Figure 4.2. We can see that for the 512x512 Lena image the erosion function takes about 5.571 seconds.



Start Profiling Run this code: crap = erode(imread('lena.jpg'),se)

Profile Summary

Generated 13-Oct-2014 19:50:07 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
erode	1	5.577 s	5.571 s	█
images\private\imapplymatrixc (MEX-file)	1	0.003 s	0.003 s	
imagesci\private\imftype	1	0.002 s	0.001 s	
imagesci\private\isjpg	1	0 s	0.000 s	
imagesci\private\jpeg_depth (MEX-file)	1	0 s	0.000 s	
imagesci\private\readjpg	1	0.008 s	0.000 s	
imagesci\private\rjpg8c (MEX-file)	1	0.008 s	0.008 s	
imapplymatrix	1	0.005 s	0.002 s	
imapplymatrix>checkOutputClass	1	0 s	0.000 s	
imapplymatrix>parseVarargin	1	0 s	0.000 s	
imformats	1	0.001 s	0.000 s	
imformats>find_in_registry	1	0.001 s	0.001 s	
imread	1	0.012 s	0.002 s	
imread>parse_inputs	1	0 s	0.000 s	
rgb2gray	1	0.006 s	-0.000 s	
rgb2gray>parse_inputs	1	0.001 s	0.001 s	

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 4.2 Matlab runtime profile of the Erosion function.

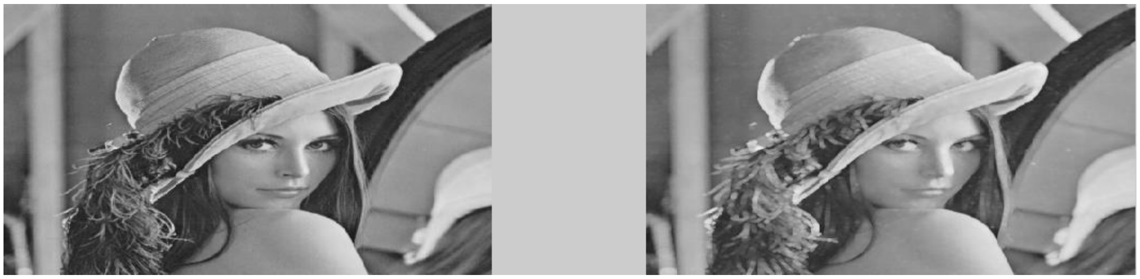
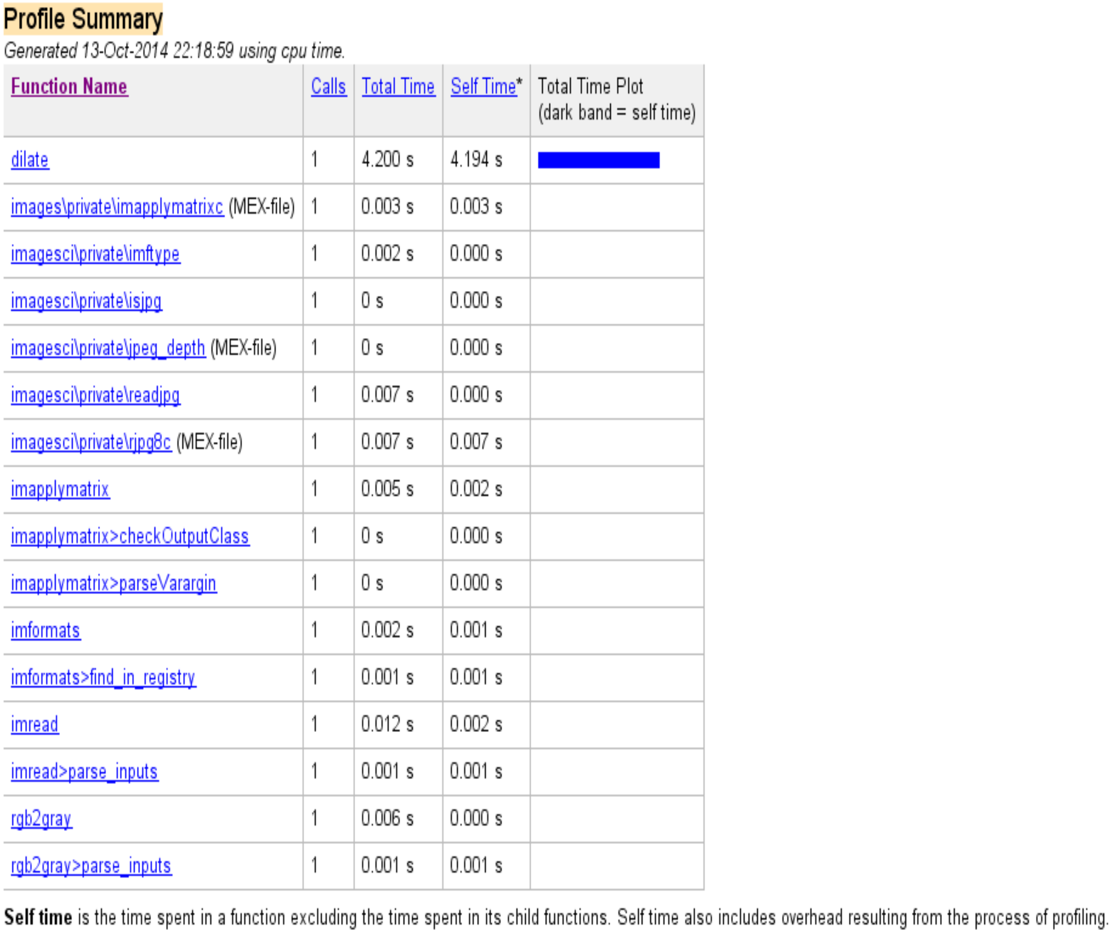


Figure 4.3 Original Lena image and the dilated image for SE(2).



Profile Summary

Generated 13-Oct-2014 22:18:59 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
dilate	1	4.200 s	4.194 s	▬
images\private\imapplymatrixc (MEX-file)	1	0.003 s	0.003 s	
imagesci\private\imftype	1	0.002 s	0.000 s	
imagesci\private\isjpg	1	0 s	0.000 s	
imagesci\private\jpeg_depth (MEX-file)	1	0 s	0.000 s	
imagesci\private\readjpg	1	0.007 s	0.000 s	
imagesci\private\rjpg8c (MEX-file)	1	0.007 s	0.007 s	
imapplymatrix	1	0.005 s	0.002 s	
imapplymatrix>checkOutputClass	1	0 s	0.000 s	
imapplymatrix>parseVarargin	1	0 s	0.000 s	
imformats	1	0.002 s	0.001 s	
imformats>find_in_registry	1	0.001 s	0.001 s	
imread	1	0.012 s	0.002 s	
imread>parse_inputs	1	0.001 s	0.001 s	
rgb2gray	1	0.006 s	0.000 s	
rgb2gray>parse_inputs	1	0.001 s	0.001 s	

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 4.4 Matlab runtime profile of the Dilation function.

Erosion and dilation by themselves are not very useful in gray-scale image processing. These operations become powerful when used in combination to develop high-level algorithms. Some of the properties of Erosion and Dilation are [13]:

- Erosion is in general not commutative: $A \ominus B \neq B \ominus A$.
- Dilation is associative: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$, for any sets A, B, and C.
- Translation is invariant: $Ax \oplus B = (A \oplus B)x$ and $Ax \ominus B = (A \ominus B)x$.
- Dilation and erosion are in a sense dual operators. Dilation can be defined as the erosion of the complement of a set. If A^c denotes the complement of the set A (i.e., a € Ac implies a does not belong to A), then the dilation of a set A by a set B is equivalent to the complement of eroding A^c by set B. In other words,

$A \oplus B = (A^c \ominus B)^c$.

- Dilation and erosion are not inverses of each other.
- Dilation distributes over union.
- Erosion distributes over intersection.
- The erosion of a set A by the union of two sets B and C is the same as the intersection of the erosion of A by B and the erosion of A by C.
- Repeated erosion of a set A by sets B0, . . . ,Bn, is the same as the erosion of A by the dilation of the sets B0, . . . ,Bn.

The last four properties are called decomposition theorems. These properties of dilation and erosion can be utilized in the parallel implementation of morphological functions. That is, a large SE can be decomposed into smaller subsets allowing efficient and parallel implementations.

For example, if we have to dilate a set A by a set B, then we need to visit every possible neighborhood of A as defined by the size of set B. This makes the algorithm unsuitable for parallel implementation. However, using the above mentioned property, if we can decompose the set B into smaller subsets, then the computations can be performed in parallel. For FPGAs, we can take advantage of the concurrency of hardware to spawn such parallel functions which can complete the computation in a small number of clock cycles. In modern multi-core digital computers, such parallel computations are performed through thread-based implementations where a function is split over several threads toward parallel computation.

4.2 Opening

For binary and gray-scale images, the opening of image f by SE b is defined as the erosion of f by b followed by the dilation of the result by b. The opening operation is shown in Equation 4.5.

$$f \circ b = (f \ominus b) \oplus b \tag{4.5}$$

4.3 Closing

For binary and gray-scale images, the closing of image f by SE b is defined as the dilation of f by b followed by the erosion of the result by b. The closing operation is shown in Equation 4.6 .

$$f \bullet b = (f \oplus b) \ominus b \tag{4.6}$$

The opening and closing for gray-scale images are duals with respect to complementation and SE reflectioner, as per equation 4.7.

$$\begin{gathered}
(f \bullet b)^c = f^c \circ b^\wedge \\
(f \circ b)^c = f^c \bullet b^\wedge \\
f^c = -f(x,y) \qquad (4.7) \\
-(f \bullet b) = (-f \circ b^\wedge)
\end{gathered}$$

$$-(f \circ b) = (-f \bullet b^\wedge)$$

The opening and closing of images has a simple geometrical interpretation. The Image f(x,y) can be viewed as a 3D surface, where the intensity values of pixels are interpreted as heights over the xy-plane [12]. Then, the opening of f by b can be interpreted as “pushing” SE b up from below against the under surface of f. At each location of the origin of b, the opening is the highest value reached by any part of b as it pushes against the under surface of f. The complete opening is then the set of all such values obtained by having the origin of b visit every (x, y) coordinate of f. See Figure 4.5.

.

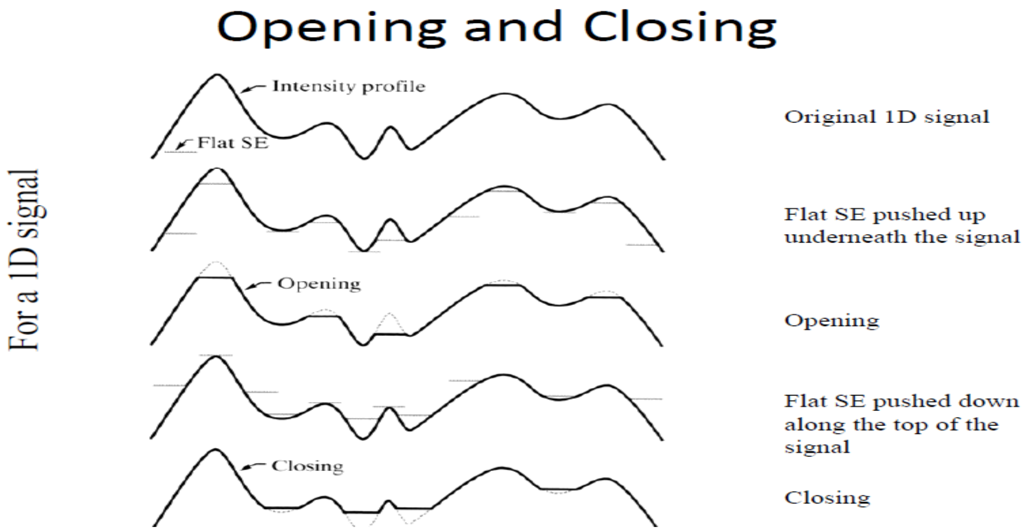


Figure 4.5 Geometrical interpretation of the Opening and Closing operations.
Source: [12].

Since the opening operation first erodes the image before dilation, the overall effect of the opening operation is that the intensity of all bright features decreases, depending on the sizes of the features compared to the SE. In the opening operation, erosion has a negligible effect on dark features and, hence, the effect on the background is negligible. The overall effect of the closing operation is that the dark features get attenuated, with the background unaffected. In developing the algorithms for opening and closing, SE(2) is used for images of size 512x512. The results of the opening operations are shown in Figures 4.6 and 4.8 . Performance results are shown in Figure 4.7.

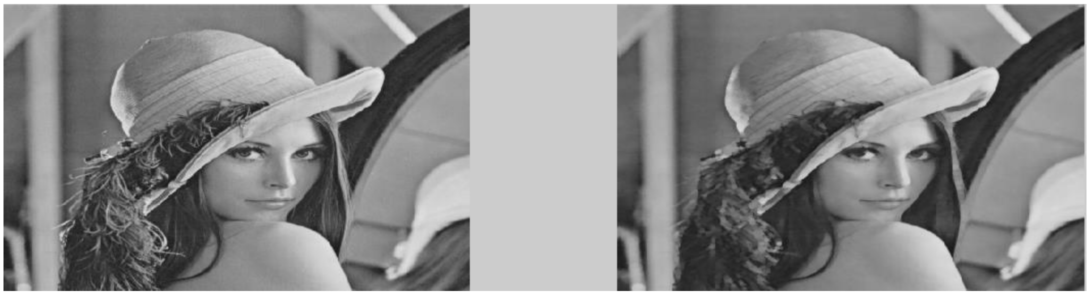
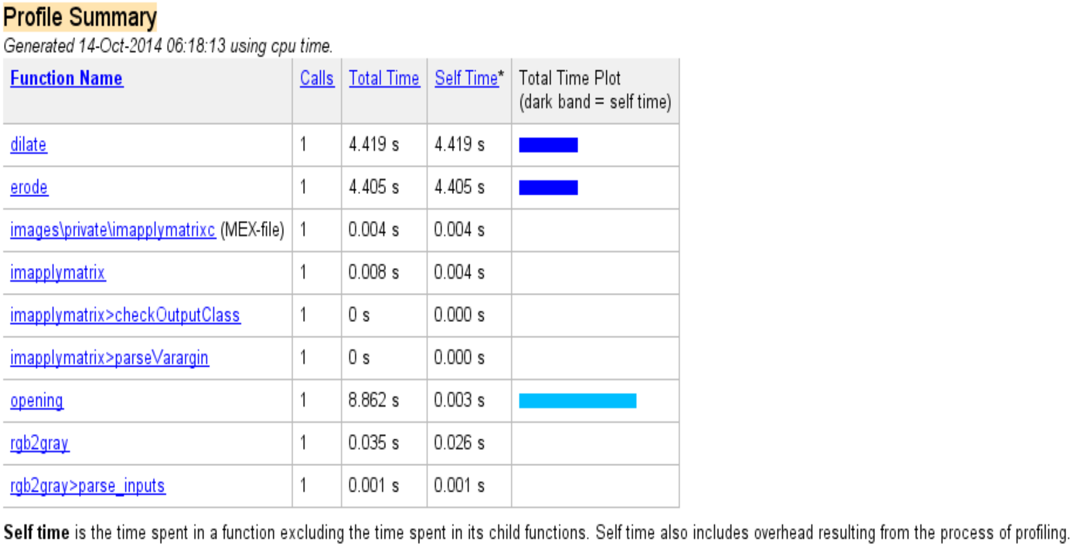


Figure 4.6 Original Lena image and the results of the Opening operation for SE(2).



Profile Summary

Generated 14-Oct-2014 06:18:13 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
dilate	1	4.419 s	4.419 s	
erode	1	4.405 s	4.405 s	
images\private\imapplymatrixc (MEX-file)	1	0.004 s	0.004 s	
imapplymatrix	1	0.008 s	0.004 s	
imapplymatrix>checkOutputClass	1	0 s	0.000 s	
imapplymatrix>parseVarargin	1	0 s	0.000 s	
opening	1	8.862 s	0.003 s	
rgb2gray	1	0.035 s	0.026 s	
rgb2gray>parse_inputs	1	0.001 s	0.001 s	

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 4.7 Matlab runtime profile of the Opening function.

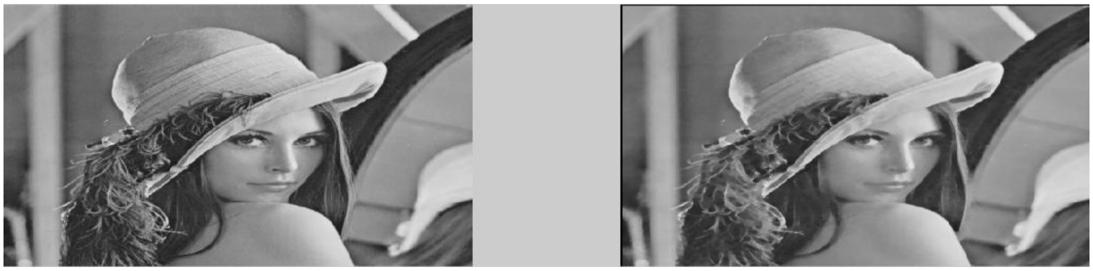


Figure 4.8 Original Lena image and the result of the Closing operation for SE(2).

4.4 Morphological Smoothing

Noise is often the most intractable problem in the field of image processing. There are two ways to work around it: either design particularly robust algorithms that can work in noisy environments, or try to eliminate the noise in a first step while losing as little relevant information as possible and consequently use a normally robust algorithm. There are many algorithms that have been cited in the literature aiming at reducing the amount of noise in images. In mathematical morphology, alternating sequential filtering, which is the application of an opening operation after a closing operation on the image, is often used to remove noise to a certain extent from the images. Since opening suppresses bright details smaller than the specified SE and closing suppresses dark details, they are used in combination as morphological filters for image smoothing and noise removal. Run-time profiling of the Morphological Smoothing operation is shown in Figure 4.10.

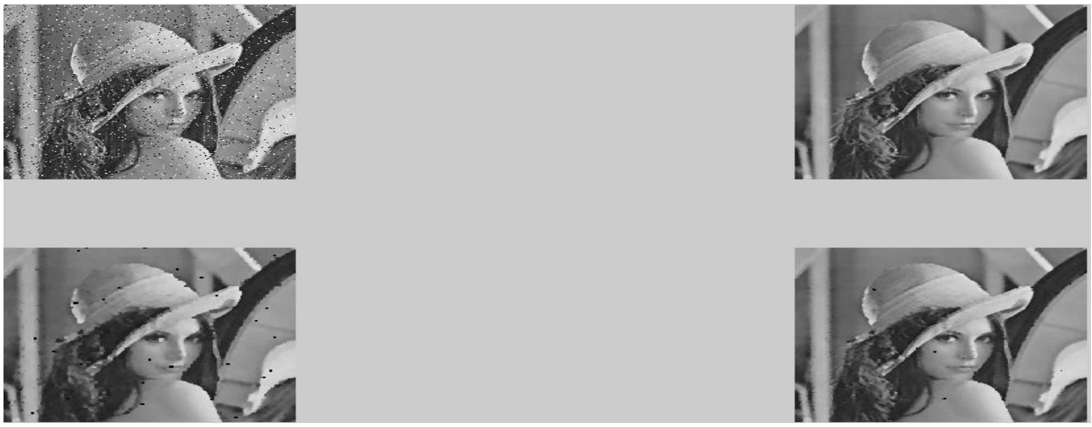
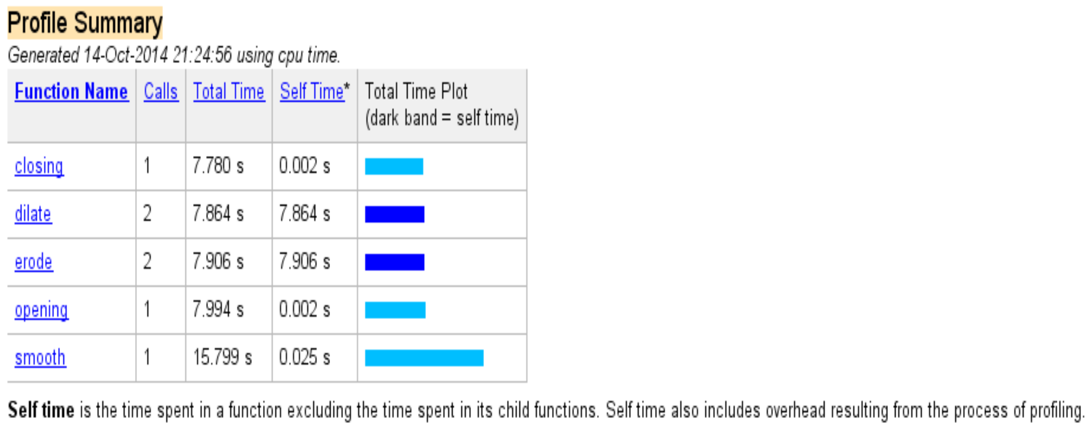


Figure 4.9 Salt and Pepper added to the Lena image (top-left), Morphological Smoothing with SE (disk type, radius 1) (bottom left); Morphological Smoothing with SE(2) (top-right); Morphological Smoothing with SE (disk type, radius 1) and SE (disk type, radius 3) (bottom-right).



Profile Summary

Generated 14-Oct-2014 21:24:56 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
closing	1	7.780 s	0.002 s	▬
dilate	2	7.864 s	7.864 s	▬
erode	2	7.906 s	7.906 s	▬
opening	1	7.994 s	0.002 s	▬
smooth	1	15.799 s	0.025 s	▬

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 4.10 Matlab runtime profile of the Morphological Smoothing function.

4.5 Morphological Gradient

Determining the gradient of an image is a fundamental image processing operation that is often used as a precursor to other, more advanced operations such as feature extraction and segmentation. The morphological gradient operator provides a simple approach to find the gradient of an image by combining the dilation and erosion operators. Generally these gradients are used in segmentation applications with edge searches, thresholding or the water shed transformation. The morphological image gradient operator g is defined in Equation 4.8.

$$g = (f \oplus b) - (f \ominus b) \tag{4.8}$$

The dilation thickens regions in an image and the erosion shrinks them. Therefore, their difference emphasizes the boundaries between regions. If the SE is relatively small, homogeneous areas will not be affected by dilation and erosion, so the subtraction tends to eliminate them. The net result is an image with the gradient-like effect. The effect of morphological gradient operation is shown in Figure 4.11 and its run-time profiling is shown in Figure 4.12.

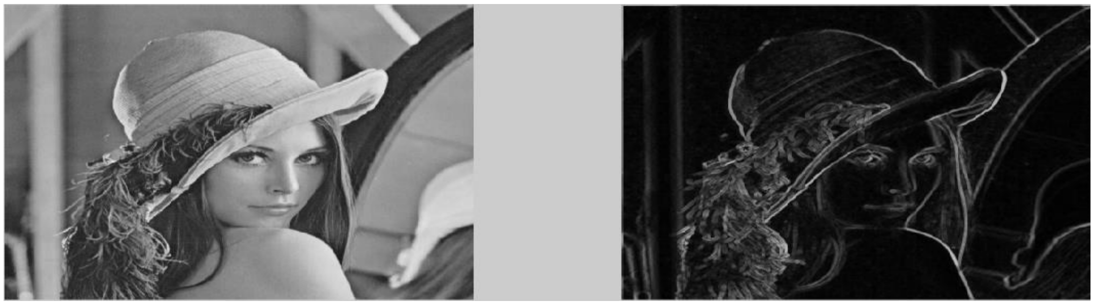
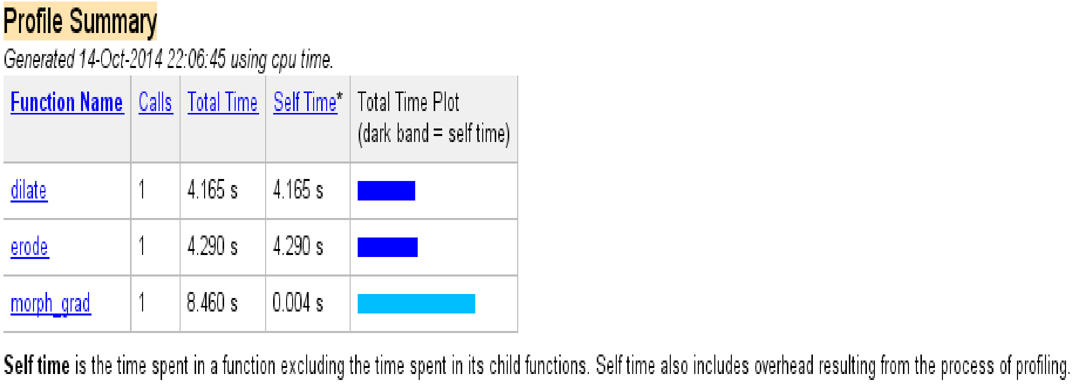


Figure 4.11 Original Lena image and the morphological gradient operated image.



Profile Summary

Generated 14-Oct-2014 22:06:45 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
dilate	1	4.165 s	4.165 s	▬
erode	1	4.290 s	4.290 s	▬
morph_grad	1	8.460 s	0.004 s	▬

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 4.12 Matlab runtime profile of the morphological gradient function.

4.6 Top-Hat and Bottom-Hat Transformation

The hat transforms represent an important class of morphological transforms used for detail extraction from signals or images. One principal application of these transforms is the removal of objects from an image by using an SE in the opening and closing that does not fit the objects to be removed. The difference then yields an image with only the removed objects.

4.6.1 Top-Hat Transformation

In mathematical morphology, top-hat transformation is an operation that extracts small elements and details from given images. The top-hat transform is defined as the difference between the input image and its opening by some SE. Top-hat transforms are used for various image processing tasks, such as feature extraction, background equalization, image enhancement, and others. An important use of the top-hat transformation is in correcting the effects of non-uniform illumination. The top-hat transform of f is given by Equation 4.9 and its results is shown in Figure 4.13. Its run-time profile is shown in Figure 4.14.

$$T_w(f) = f - (f \circ b) \qquad (4.9)$$

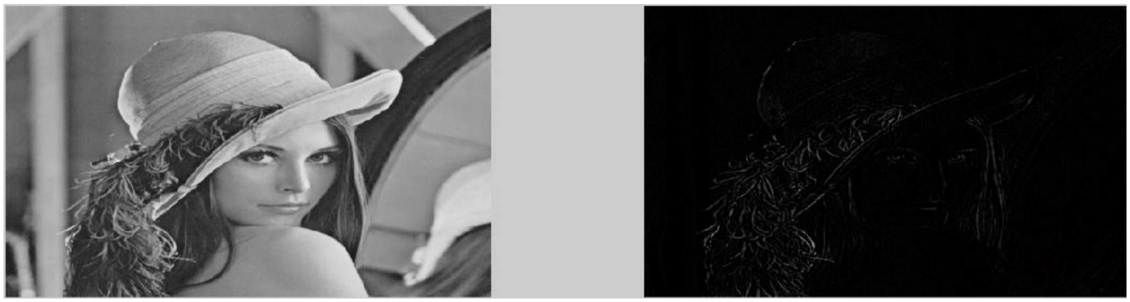
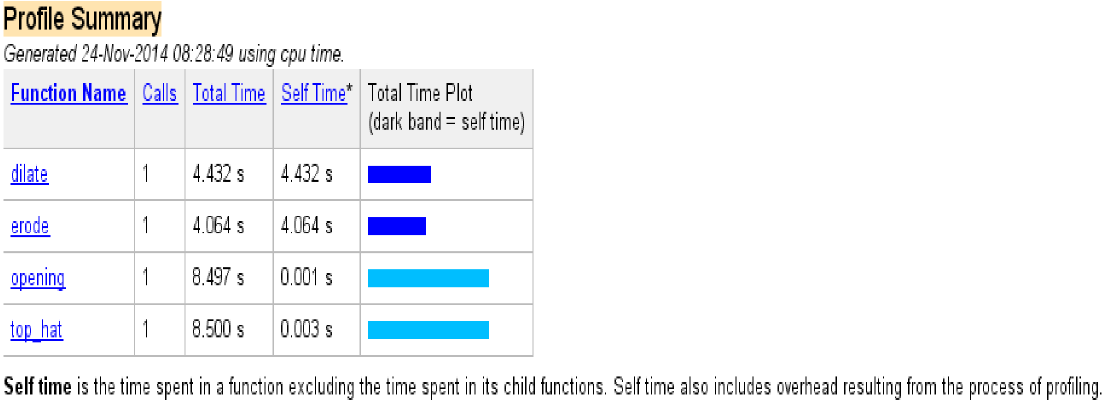


Figure 4.13 Original Lena image and the results of the top-hat transformed image.



Profile Summary

Generated 24-Nov-2014 08:28:49 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
dilate	1	4.432 s	4.432 s	▬
erode	1	4.064 s	4.064 s	▬
opening	1	8.497 s	0.001 s	▬
top_hat	1	8.500 s	0.003 s	▬

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 4.14 Matlab runtime profile of the top-hat transformation function.

4.6.2 Bottom-Hat Transformation

The bottom-hat morphological operator subtracts an input image from the result of morphological closing on the input image. Applied to a binary image, this transformation allows getting all the pixels that were added by the closing filter but were not removed afterwards due to formed connections. The bottom-hat transform of f is given by Equation 4.10 and is shown in Figure 4.15. Its run-time profile is shown in Figure 4.16.

$$T_b(f) = (f \bullet b) - f \tag{4.10}$$

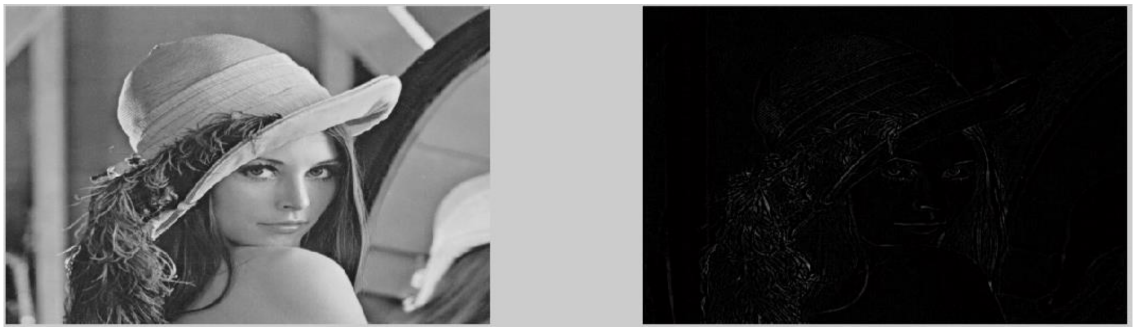
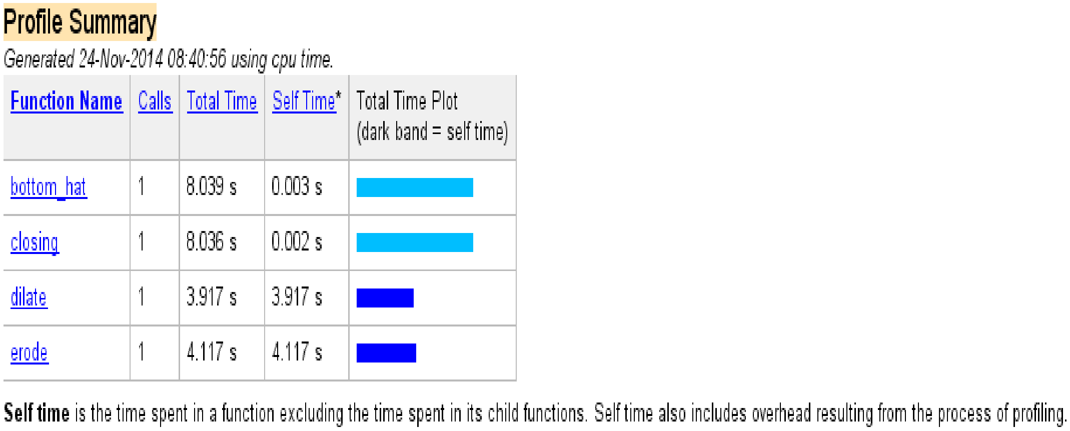


Figure 4.15 Original Lena image and the bottom-hat transformed image.



Profile Summary

Generated 24-Nov-2014 08:40:56 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
bottom_hat	1	8.039 s	0.003 s	▬
closing	1	8.036 s	0.002 s	▬
dilate	1	3.917 s	3.917 s	▬
erode	1	4.117 s	4.117 s	▬

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 4.16 Matlab runtime profile of the bottom-hat transformation function.

CHAPTER 5

VIVADO HIGH LEVEL SYNTHESIS (HLS) AND ALGORITHM SYNTHESIS

5.1 Introduction

This chapter describes the Xilinx Vivado HLS synthesis tool that was used to synthesize the core IP blocks for the morphological operators. The user guide for Vivado HLS is relatively vast and, hence, only the specific optimizations and directives that were implemented in the thesis will be discussed. Vivado HLS is a Xilinx tool that transforms a C specification into a Register Transfer Level (RTL) implementation that can be synthesized for a Xilinx FPGA by another Xilinx tool, the Vivado Design Suite. The C specifications can be either in C, C++ or SystemC. The C function or algorithm is then synthesized into an IP block which can be integrated into a hardware system. It provides comprehensive language support, a rich set of libraries and directives for creating the most optimal implementation for the specified C algorithm. The functionality inside Vivado HLS enables the following design flow:

- Compile, execute (simulate) and debug the C algorithm.
- Synthesize the C algorithm into an RTL implementation, with or without user optimization directives.
- Comprehensive reporting and analysis of resource usage and timing analysis.
- Automated verification of the RTL implementation.
- Package the RTL implementation into a selection of IP formats.

In HLS, a test bench in the form of a C program is supported and referred to as a C simulation. Executing the C test bench validates the algorithm’s functional integrity. The primary input to Vivado HLS is a top level C function written in C, C++ or SystemC.

This function may contain a hierarchy of sub-functions, loops and additional inputs as constraints and directives. The constraints are mandatory and include the clock period, the clock uncertainty (this defaults to 12.5% of the clock period if not specified) and the FPGA target device. The directives are optional and Vivado HLS uses them to direct the synthesis process to implement a specific behavior or implementation.

The primary output from Vivado is the implementation of the C-specification in RTL format. The RTL output is made available in various industry standard Hardware Description Language (HDL) formats of Verilog and VHDL. These HDLs can then be synthesized to gate-level implementation by logic synthesis. The Vivado Design Suite includes all the development tools required to create a bitstream file from the HDL specifications. Generally, the RTL is packaged into IP blocks for use within other tools in Xilinx design flows. The Vivado HLS tool supports different IP formats such as IP-Catalog, Pcore and System generator for DSP.

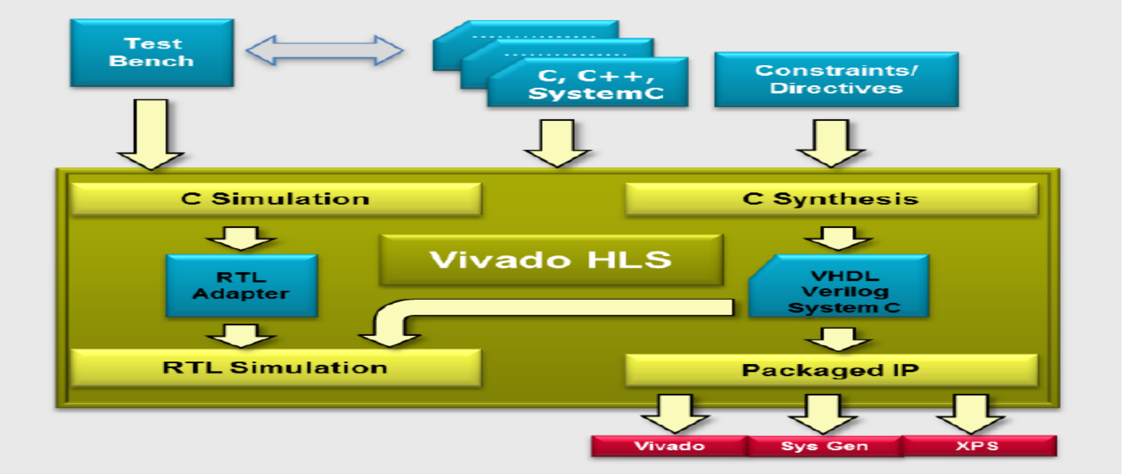


Figure 5.1 Vivado HLS design flow.
Source:[14].

5.2 Scheduling and Binding

Scheduling and binding are the processes at the heart of High-Level Synthesis. The scheduling process analyzes the C-specification and, thereafter, HLS determines the operations to be completed in any particular clock cycle. The scheduling process takes into account the clock frequency, timing information from the device technology library, and any user specified optimization directives. Let us take the following example: y= x*a+b+c. Figure 5.2 shows the process of scheduling. The multiplication and the first addition are scheduled to execute in the first clock cycle. The next clock cycle performs the second addition and the output is available at the end of the second clock cycle.

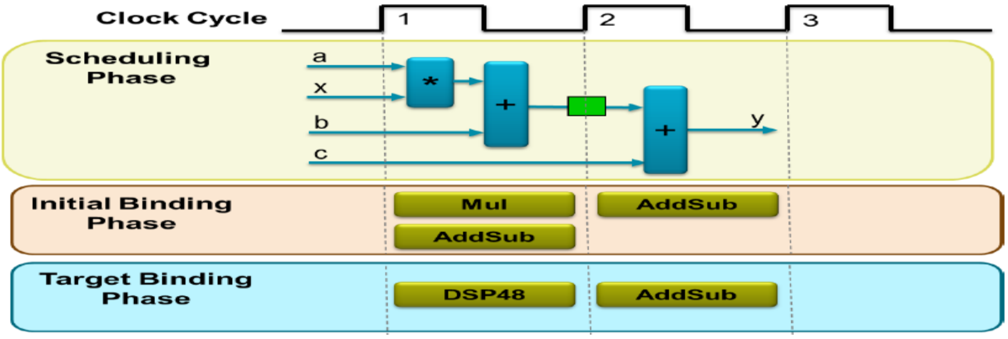


Figure 5.2 Scheduling and Binding in HLS.
Source: [14].

The scheduling process decides the number of clock cycles that can be allocated to an operation depending upon the length of the clock cycle on the target device. The target FPGA device defines the length of the clock cycle and the time to complete each operation. For faster FPGAs, the scheduling process assigns a larger number of operations per clock cycle; conversely, for slower FPGAs with a shorter clock length, a smaller number of operations are scheduled.

The Binding process maps the hardware resources to each of the scheduled operations. The hardware resource is mapped in two phases: initial binding and target binding phases. As Figure 5.2 shows, an initial binding for this example implements the multiplier operation using a Mul resource (a combinational multiplier) and both add operations using an AddSub resource (a combinational adder or subtractor). In the target binding phase, HLS uses device specific information to implement the operations in the most optimal way. As shown in Figure 5.2, the target binding phase implements the multiplication operation using a DSP48 resource and one of the additions is implemented using an AddSub resource. A DSP48 resource is a computational block available in the FPGA architecture that provides the ideal balance between high-performance and small area.

5.3 Interface Synthesis and IO Protocols

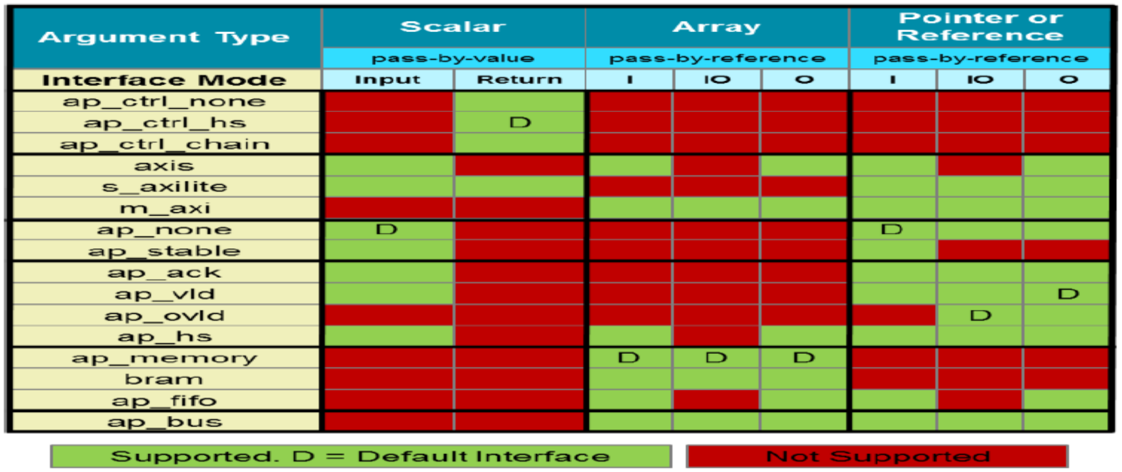
In general, all the inputs and outputs for C functions are passed through function arguments. However, in RTL designs the input and outputs for function are provided through a port in the design interface which operates by following specific input-output (I/O) protocols. When the top-level function is synthesized, the parameters to the function are synthesized into RTL ports. This process is called interface synthesis. Vivado HLS creates three types of ports in the RTL design:

- Clock and Reset ports: ap_clk and ap_rst.
- Block-Level interface protocols: ap_start, ap_done, ap_ready and ap_idle. By default, a block-level interface protocol is added to the design. The control port ap_start is held high when the block starts processing the data. Similarly ap_done indicates if the block has completed its execution. Other signals, such as ap_ready indicate if the block is ready to accept new data. ap_idle indicates if the block is in the idle state. These signals are shown in the simulation waveform of Figure 5.16.

- Port Level interface protocols: These signal ports are created for each argument in the top-level function and the return value of the block, if implemented. After the block-level protocol has been used to start the operation in the block, the port level protocols are used to transfer data in and out of the block. The I/O protocol created depends upon the type of argument used in the function parameter. Figure 5.3 shows the mapping of function argument type to the available I/O protocols. By default, input pass-by-value arguments and pointers are implemented as wire ports with no handshaking signal and output pointers are implemented using output valid signal. It is also possible to implement a function argument without any I/O protocol using the ap_none interface directive. In this case, the data must be held stable until read. It is generally advantageous to use handshaking protocols on function parameters so that the values can be probed later.

- Separate input and output ports are created for function arguments which are both read from and written into.

- If the function has a return value, an output port ap_return is implemented to provide the return value. Completing one transaction in RTL is equivalent to completing execution of one C function call. The block-level protocols indicate the function is complete with the ap_done signal. This also indicates the data on port ap_return is valid and can be read.



Argument Type	Scalar		Array			Pointer or Reference		
	pass-by-value		pass-by-reference			pass-by-reference		
Interface Mode	Input	Return	I	IO	O	I	IO	O
ap_ctrl_none	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_hs	🟥	🟩 D	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_chain	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
axis	🟩	🟥	🟩	🟥	🟩	🟩	🟥	🟩
s_axilite	🟩	🟩	🟥	🟥	🟥	🟩	🟩	🟩
m_axi	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩
ap_none	🟩 D	🟥	🟥	🟥	🟥	🟩 D	🟩	🟩
ap_stable	🟩	🟥	🟥	🟥	🟥	🟩	🟥	🟥
ap_ack	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩
ap_vld	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩 D
ap_ovld	🟥	🟥	🟥	🟥	🟥	🟥	🟩 D	🟩
ap_hs	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩
ap_memory	🟥	🟥	🟩 D	🟩 D	🟩 D	🟥	🟥	🟥
bram	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥
ap_fifo	🟥	🟥	🟩	🟥	🟩	🟩	🟥	🟩
ap_bus	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩

🟩 Supported. D = Default Interface 🟥 Not Supported

Figure 5.3 Interface Synthesis in Vivado HLS.
Source: [14].

5.3.1 Block-Level Interface Protocols

There are three kinds of block level interface protocols available: ap_ctrl_none,ap_ctrl_hs and ap_ctrl_chain. These interface protocols are specified in the function or on the function return. Even if the function does not use a return value, the block-level protocol is specified on function return. The ap_ctrl_hs mode is the default protocol. In this mode, ap_start is used to indicate the beginning of data processing in the IP block. The ap_start signal remains high until the ap_ready signal goes high, which indicates that the block is ready to accept new data. The ap_ctrl_chain protocol is similar to ap_ctrl_hs, and is used to cascade or chain the IP blocks. It has an additional input port ap_continue which provides back-pressure from blocks consuming its data. The ap_ctrl_none mode implements the design without any block-level I/O protocol.

5.3.2 Port-Level Memory Interface Protocols

For processing or storing large sets of data, arrays are used as function parameters. By default, HLS implements such arrays using the ap_memory interface. The memory is generally implemented as a standard BRAM with data, address, chip enable and write enable ports. BRAMs are instantiated either as single-port or dual-port BRAMs. A single port BRAM can read and write one unit of data per clock cycle whereas a dual-port BRAM can access either two read or two writes per clock cycle. However, during the scheduling process, if HLS determines that using a dual-port BRAM cannot optimize the design, it is converted to a single-port BRAM. The RESOURE directive can be used to specify the memory resource. We can further explicitly specify whether to use a single-port BRAM or a dual-port BRAM.

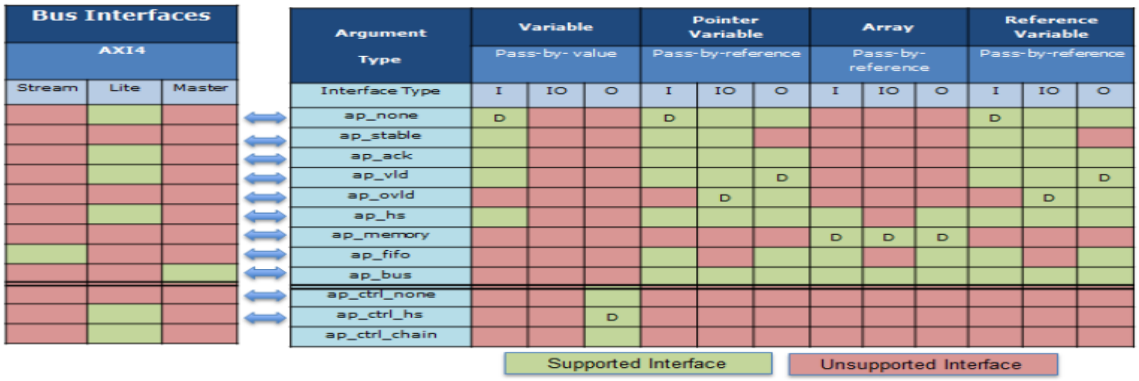
The next alternative for the memory interface is the BRAM interface mode which is functionally identical to the ap_memory interface. The only difference between the two is that ap_memory is implemented with multiple, separate ports whereas the BRAM interface is implemented with a single/grouped port which can be connected to a Xilinx BRAM. Finally, if the array is accessed in a sequential manner, an ap_fifo interface can be used. For the ap_fifo interface, the data access to the block has to be in sequential order. If HLS determines that the data are not accessed in a sequential order, the program execution is halted and an error message is reported. Another point to note for the ap_fifo interface is that it can only be used for reading or writing, not both.

5.4 AXI Interfaces

In addition to the standard block-level and port-level interfaces explained in the interface synthesis section, Vivado HLS can also add bus interfaces to the RTL design. These bus interfaces are generally added to make them compatible with the AXI bus interfaces of the ARM peripherals and ports. The AXI bus interfaces are added to the design during the IP export process and, hence, are not reflected in the synthesis reports. The following bus interfaces are available:

- AXI4-Lite Slave
- AXI4 Master
- AXI4-Stream

The above mentioned AXI bus interfaces can be added only to certain block-level and port level RTL I/O protocols. Figure 5.4 shows a list of the RTL interface ports that Vivado HLS creates and bus interfaces that can be connected to them. For example, an AXI4-Stream bus interface can only be added to ports of type ap_fifo.



Bus Interfaces			Argument Type	Variable			Pointer Variable			Array			Reference Variable		
AXI4				Pass-by- value			Pass-by-reference			Pass-by- reference			Pass-by-reference		
Stream	Lite	Master	Interface Type	I	IO	O	I	IO	O	I	IO	O	I	IO	O
🟥	🟩	🟥	ap_none	🟩 D	🟥	🟥	🟩 D	🟩	🟩	🟥	🟥	🟥	🟩 D	🟩	🟩
🟥	🟥	🟥	ap_stable	🟩	🟥	🟥	🟩	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟥
🟥	🟩	🟥	ap_ack	🟩	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥	🟩	🟩	🟩
🟥	🟩	🟥	ap_vld	🟩	🟥	🟥	🟩	🟩	🟩 D	🟥	🟥	🟥	🟩	🟩	🟩 D
🟥	🟥	🟥	ap_ovld	🟥	🟥	🟥	🟥	🟩 D	🟩	🟥	🟥	🟥	🟥	🟩 D	🟩
🟥	🟩	🟥	ap_hs	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩
🟥	🟥	🟥	ap_memory	🟥	🟥	🟥	🟥	🟥	🟥	🟩 D	🟩 D	🟩 D	🟥	🟥	🟥
🟩	🟥	🟥	ap_fifo	🟥	🟥	🟥	🟩	🟥	🟩	🟩	🟥	🟩	🟩	🟥	🟩
🟥	🟥	🟩	ap_bus	🟥	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
🟥	🟥	🟥	ap_ctrl_none	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
🟥	🟩	🟥	ap_ctrl_hs	🟥	🟥	🟩 D	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
🟥	🟩	🟥	ap_ctrl_chain	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥

🟩 Supported Interface 🟥 Unsupported Interface

Figure 5.4 Bus Interface compatibility with different port and block level interfaces.
Source: [14].

5.4.1 AXI Lite Slave Interface

An AXI4 slave interface is typically used to access the function interface ports in the design and to control the IP block by some form of host processor or micro-controller. When multiple AXI Lite ports are used on an IP block, they can be grouped or bundled into a common AXI bus. These slave ports are available to the software world using device drivers. The header file of the IP block contains a simple function to access the port values.

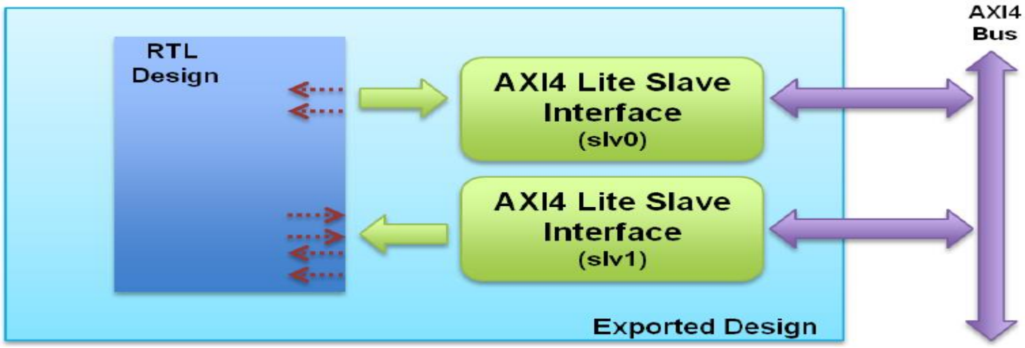

RTL Design
AXI4 Lite Slave Interface (slv0)
AXI4 Lite Slave Interface (slv1)
AXI4 Bus
Exported Design

Figure 5.5 AXI4 Lite Slave Interfaces with grouped RTL ports.
Source:[14].

In addition to providing support for accessing the port register values, these header files also contain functions to access the block level ports, such as ap_start, ap_idle, ap_return, etc, which can be used to control the designed IP block. Setting the control register for the ap_start signal to logic 1 causes the block to execute one transaction, it must be set to logic 1 again to start the next transaction. After the block starts, its operation, the ap_done port can be polled to check the completion of data processing.

5.4.2 AXI4 Master Interface

To create an AXI4 Master interface, the RTL port must have an ap_bus interface, as shown in Figure 5.6. This interface is used with any array or pointer/reference arguments in any of two modes:

- Individual data transfers.
- Burst mode data transfers using the C memcpy function.

In individual data transfers, data is transferred over the AXI4 Master interface in a simple read or write operation with one address and one data values at a time. In the burst transfer mode, data is transferred using a single base address followed by multiple sequential data values. A burst mode data transfer is used for high throughput.

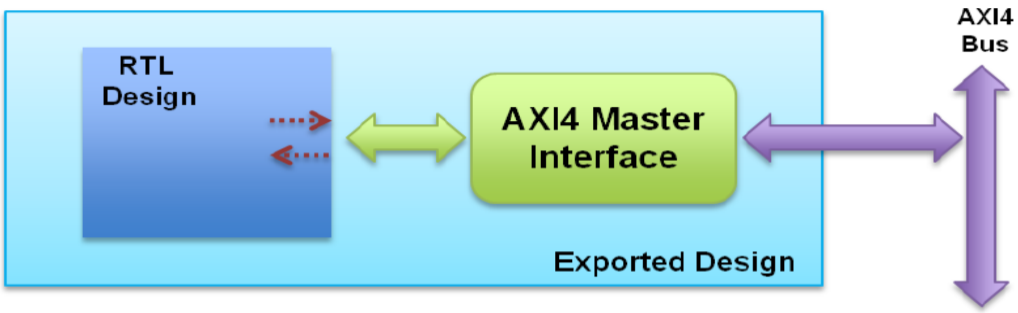


Figure 5.6 AXI4 Master Interface.
Source:[14].

5.4.3 AXI Stream Interface

An AXI4 interface can be applied to any ap_fifo RTL port. This interface can be either in the master or slave configuration. The AXI Stream interface is generally used for data stream input and output. Output interfaces are implemented as AXI4 Stream master interfaces and input interfaces as AXI4 Stream slave interfaces. Multiple RTL ports can be grouped into a single AXI4 stream interface in the same manner as for an AXI4 slave interface. The RTL ports grouped into an AXI4 stream interface, however, must be either all input ports or all output ports.

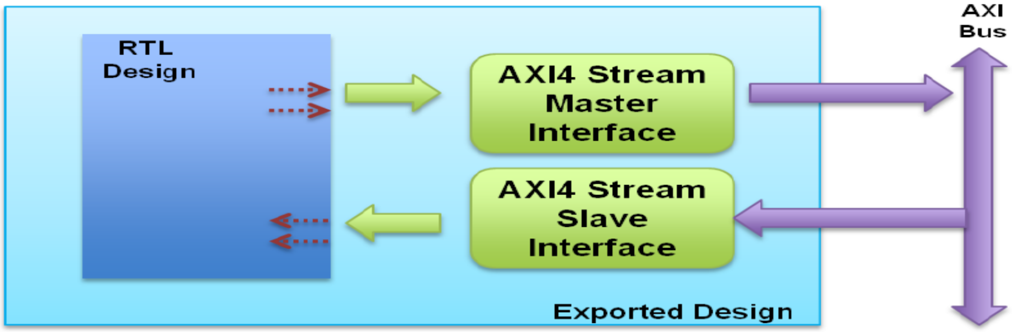


Figure 5.7 AXI4 Stream Interface.
Source: [14].

5.5 Optimizations

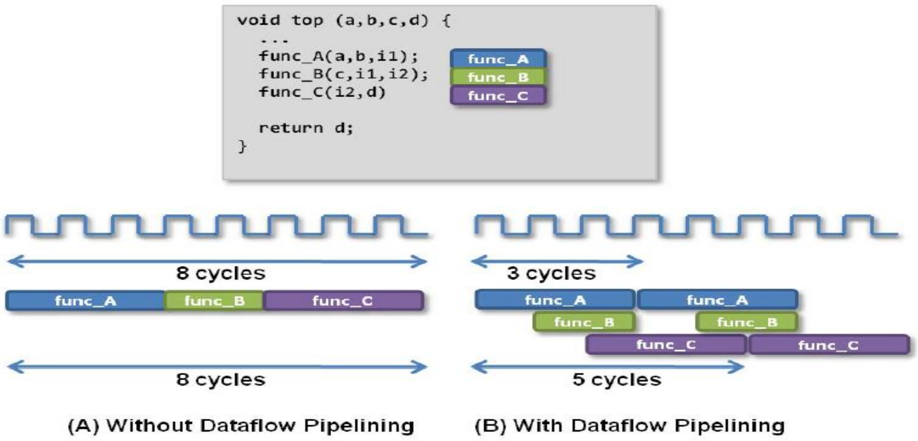
Vivado HLS can apply various optimization strategies on the C specification to produce a micro-architecture that meets the desired area and timing goals. These optimizations can be applied to the top level functions, sub-functions and memory resources as directives, and, when properly used, can reduce the overall latency and increase the throughput of the IP block. Moreover, different optimization strategies can be applied to the same function as tcl directives.

5.5.1 Function Inlining

Inlining a function can be used to remove the overhead of the clock cycles needed to enter and exit the function call. If a function is called over several times, for example 100 times, the overhead of entering and exiting the function can accrue to 200 clock cycles. And, hence, inlining the function can remove the extra clock cycles. However, inlining the function removes the hierarchy and increase the area of the design.

5.5.2 Function Dataflow Pipelining

Function Dataflow Pipelining allows the execution of different functions to overlap, which results in increasing the overall throughput and reducing the latency of the design. Vivado HLS takes a sequential functional description, consisting of various sub functions, and creates a parallel processing architecture from it.



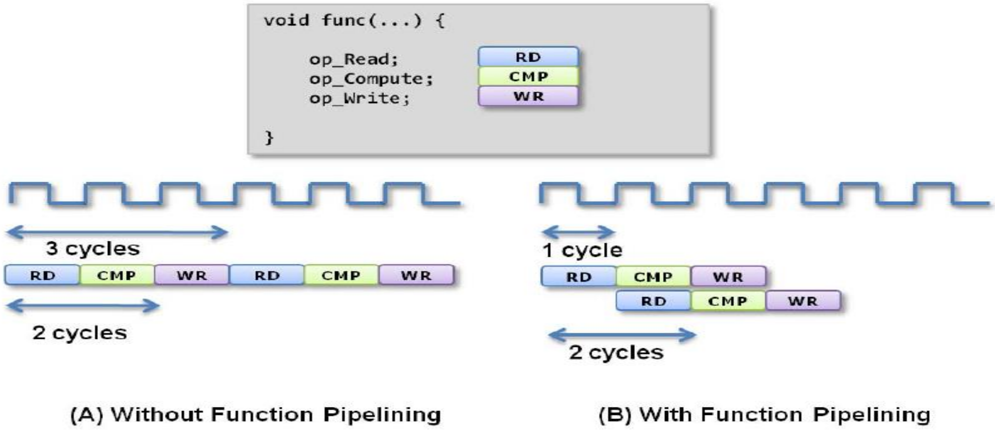
```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

Figure 5.8 Function dataflow pipelining for the top function.
Source:[14].

For example, if a top function consists of sub functions func_A, func_B and func_C, the sequential execution of the top function will take eight clock cycles. However, if we use the dataflow pipelining directive on the top function, it will try to execute the sub functions concurrently. From Figure 5.8 we can see that the overall latency of the top function has been reduced to five clock cycles from eight clock cycles.

5.5.3 Function Pipelining

Function pipelining is similar to function dataflow pipelining and, if specified as a directive, Vivado HLS tries to optimize the operations inside the function. This has the benefit of decreasing the latency and increasing the throughput of the individual function. The throughput improvements in function pipelining are shown in Figure 5.9.



```
void func(...) {

    op_Read;
    op_Compute;
    op_Write;

}
```

(A) Without Function Pipelining

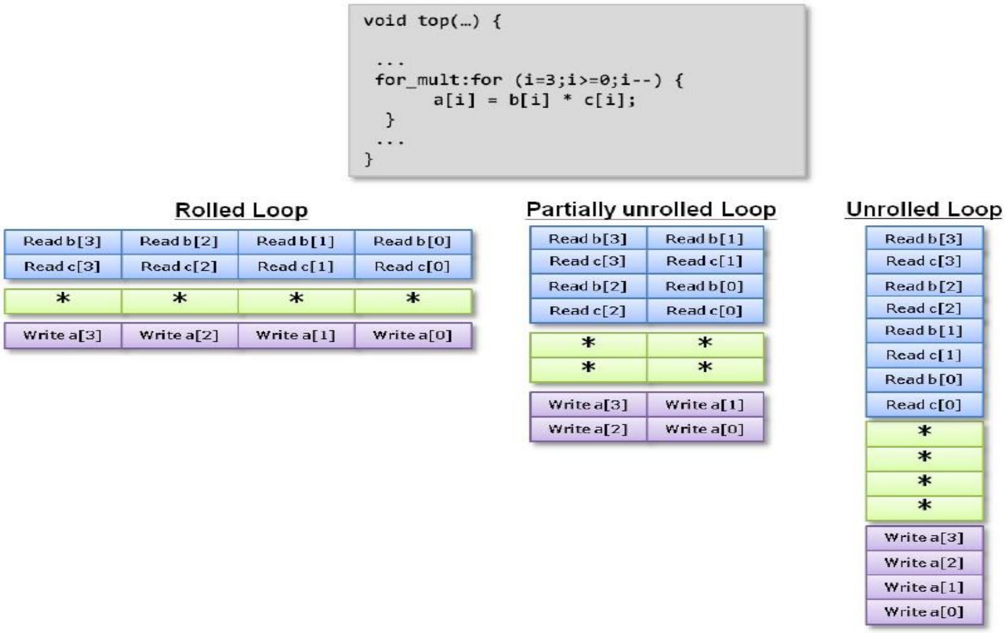
(B) With Function Pipelining

Figure 5.9 Function dataflow pipelining for the top function.
Source:[14].

As shown in Figure 5.9, the function consists of three operations read, compute and write. It takes three clock cycles to execute the function and, hence, the next read can take place after every three clock cycles. However, when the function pipelining directive is issued on the function, a new input is read in each clock cycle, with no change to the output latency or resources used. It is only possible as long as there is no resource contention or data dependency that prevents pipelining.

5.5.4 Loop Unrolling

Loops are the most common construct of any programming language and, hence, loop optimization is very critical for high performance. Vivado HLS provides the option of loop unrolling with three flavours: rolled loops, unrolled loops and partially unrolled loops. By default, loops are kept rolled and treated as a single entity. All the operations in a rolled loop are executed using the same hardware resources. Loops can be unrolled completely or partially by placing directives on the loops. Figure 5.10 shows the partial and complete unrolling for the loop contained in a top function.



```
void top(…) {

 ...
 for_mult:for (i=3;i>=0;i--) {
       a[i] = b[i] * c[i];
  }
 ...
}
```

Figure 5.10 Loop unrolling in Vivado HLS.
Source:[14].

When loops are completely unrolled, Vivado HLS tries to execute the entire operation in one clock cycle, which provides the best possible latency. However, it depends upon the array variables if they can read and write the data in one clock cycle. Generally, arrays are mapped to BRAMs which can provide maximum read/write capability if they have two ports. In such cases, the final loop delay will depend on the delay of the BRAM.

5.5.5 Loop Merging

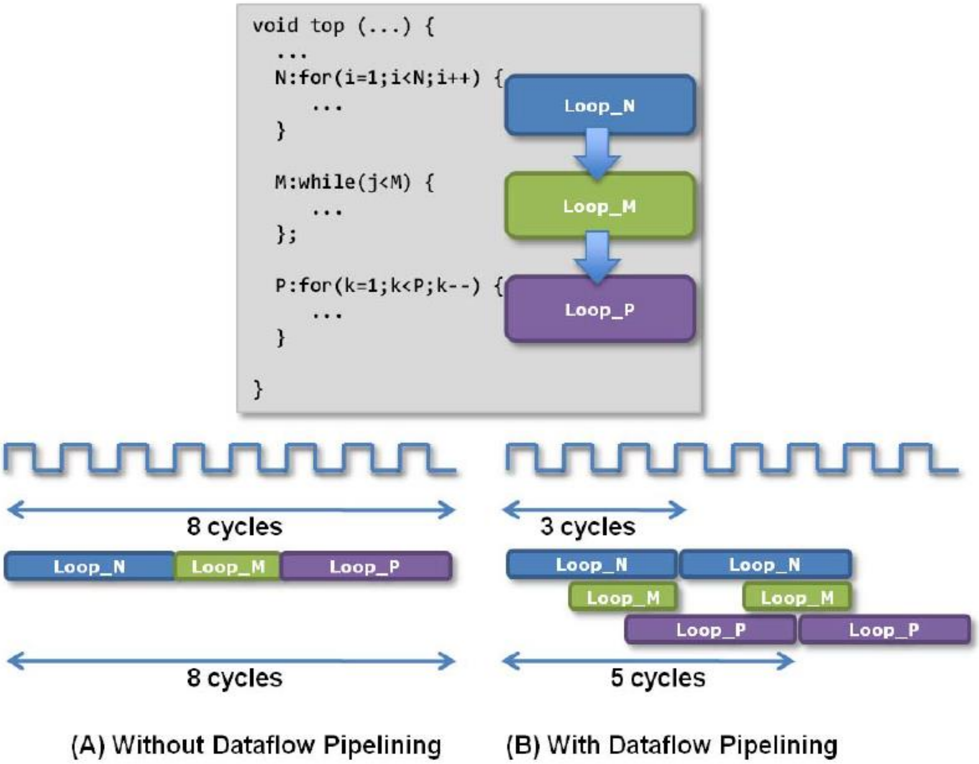
By placing the loop merging directive on two or mode loops, Vivado HLS can produce a control structure which optimizes both of the loops concurrently. However, there are a few restrictions which do not allow different loops to be merged. For example, if the loop bounds are variables, they must have the same value. If the loop bounds are constants, the maximum constant value is used as the bound of the merged loop.

5.5.6 Flattening Nested Loops

When the C specification contains nested loops, additional clock cycles are spent to move between the rolled loops. It takes one clock cycle to move from an outer loop to an inner loop, and from an inner loop to an outer loop. Additionally, nested loops prevent the outer loop from being pipelined. When the loop flattening directive is issued on the outer loop, Vivado HLS builds the entire loop into a single hierarchy and, hence, eliminates the overhead of extra clock cycles.

5.5.7 Loop Dataflow Pipelining

Loop dataflow pipelining is similar to function dataflow pipelining. When the loop pipelining directive is issued on loops, it converts the sequential operation to a concurrent operation in RTL. Dataflow pipelining should be applied to a function, loop or region that contains all functions or all loops Figure 5.11 shows the performance benefit we achieve from loop dataflow pipelining. As shown, without dataflow pipelining, loop N must execute and complete all iterations before loop M can begin. With dataflow pipelining, these loops are allowed to operate in parallel, accepting new inputs every three cycles and outputting a result every five cycles.



```
void top (...) {
  ...
  N:for(i=1;i<N;i++) {
    ...
  }

  M:while(j<M) {
    ...
  };

  P:for(k=1;k<P;k--) {
    ...
  }

}
```

Figure 5.11 Loop dataflow pipelining in Vivado HLS.
Source:[14].

5.5.8 Array Partitioning and Optimizations

Memory mapping in hardware designs plays a significant role in improving the performance and area. Arrays in the algorithm description can be implemented in different ways in the hardware design. They can be mapped to BRAMs or registers/LUTs or can be built as FIFOs using a stream interface. When arrays are used to store constant coefficient values, ROMs are instantiated to store these coefficients, at the RTL level.

Vivado HLS supports the initialization of coefficient arrays by including the values in the bitstream.

However, when there are many small arrays in the algorithm specification, mapping them to separate memory locations can lead to significant wastage of on-chip memory resources. In this case, it is desirable to bundle the different small arrays into a single large array. Vivado HLS provides options of horizontal and vertical mapping to combine arrays into single RAM blocks. Horizontal mapping is shown in Figure 5.12 and vertical mapping is shown is Figure 5.13.

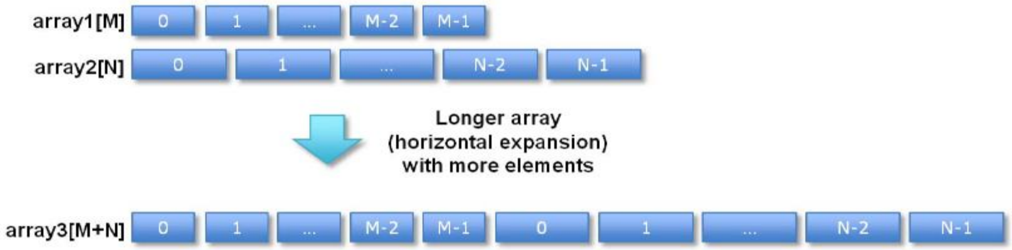


Figure 5.12 Horizontal mapping of arrays in Vivado HLS.
Source:[14].

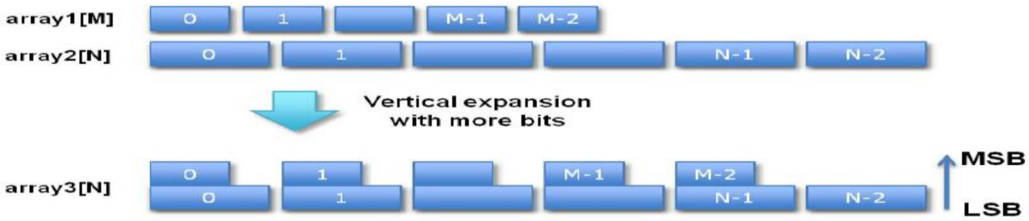


Figure 5.13 Vertical mapping of arrays in Vivado HLS.
Source:[14].

Finally, the last optimization we can apply on arrays is partitioning them into smaller arrays. Memories only have a limited amount of read ports and write ports that can limit the throughput of a load/store intensive algorithm. We can improve the bandwidth by splitting up the original array (a single memory resource) into multiple smaller arrays (multiple memories), effectively increasing the number of ports. Vivado HLS provides three kinds of array partitioning. These are as follows.

- Block: The original array is split into equally sized blocks of consecutive elements in the original array.
- Cyclic: The original array is spilt into equally sized blocks interleaving the elements in the original array.
- Complete: The original array is split into its individual elements. This corresponds to resolving a memory into registers.

Out of these, the complete partitioning of arrays provides the greatest throughput but consumes the highest number of memory resources. Partitioning of arrays is shown in Figure 5.14.

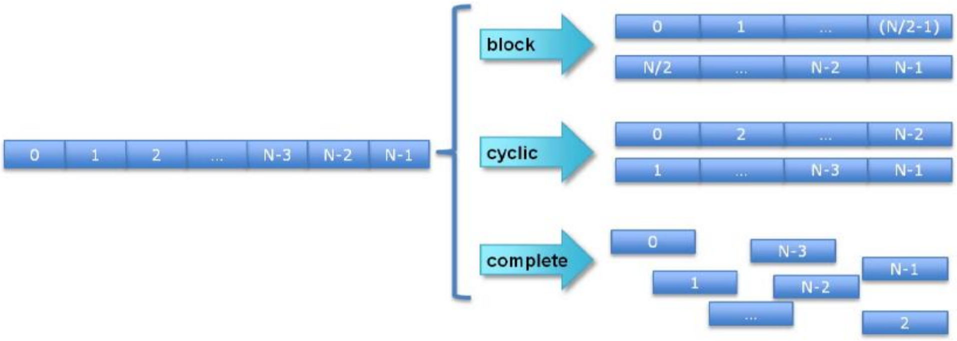


Figure 5.14 Partitioning of arrays in Vivado HLS.
Source:[14].

5.5.9 Arbitrary Precision Datatypes

Vivado HLS provides the ability to specify arbitrary precision data types for the C-specification. The advantage of arbitrary precision data types is that they allow the C code to be updated to use variables with smaller bit widths, thus consuming fewer hardware resources at the RTL level.

5.6 Morphological Algorithm Synthesis and Optimizations

This section describes the actual implementation of the morphological image processing algorithm using Vivado HLS for synthesis. All possible approaches were considered to reduce the latency and provide the maximum throughput for a single operation on the image pixel values.

5.6.1 Input and Output Interfaces

The inputs to the image processing blocks are the image pixel values, the width and length of the image, and the values of the SE matrix. Since the width and the length of the image do not change over a single transaction, the width and length parameters are implemented with the AXI-Lite resource and the ap_vld interface. The ap_vld interface is chosen as we can probe the port for the values of length and width. Alternatively, if we do not want to use these parameters, we can embed these values in the loop of the algorithm.

An SE consists of a set of binary values and can be implemented in a couple of ways. Here, we consider an SE to be a matrix of size 5x5. This matrix can be implemented as a twenty five bit integer and is input to the IP block using the AXI Lite Resource port. Once the value is read into the port, the individual matrix values can be extracted using the bit extraction algorithm. One other way of implementing an SE is by using twenty five separate AXI Lite Ports on the input interface. However, every time we have to change the SE, we have to write into all the twenty five different ports. Alternatively, we can use the AXI-Stream resource and the ap_fifo interface to read the SE value into the matrix. Finally, if the SE remains constant throughout the algorithm, we can internally map the matrix into a ROM. In the implemented algorithm, the SE is read into an internal matrix which allows the array to be partitioned completely and, hence, improve the throughput.

The final ports to consider are the input and output pixel ports. Since the algorithm was synthesized for a large number of dynamic data (512x512 pixels), the ports were mapped to an AXI-Streaming Resource with the ap_fifo interface. The input pixel port was mapped to an AXIS slave interface and the output pixel port was mapped to a master interface. These AXI Stream ports allow high bandwidth transfers of data to the DDR memory through a DMA peripheral. The DMA peripheral is, in turn, connected to the ARM processor on the FPGA board via the high performance slave ports AXI HP0 and HP1. While transferring the pixel data, the AXI Stream, restricts the transfer to a sequential access and no throughput optimization is possible. Also, data can be only read or written once to/from the ports. So, when working with the AXI Stream, we have to store the data into the memory if any subsequent access is necessary.

5.6.2 Memory Architecture and Image Buffers

Memory buffers are fundamental features of any image processing or video processing algorithm. The memory buffers provide temporal and spatial access to the pixel data for the algorithm to work. Generally, these memory structures are implemented in hardware as shift registers, line buffers and memory windows. All of these memory buffers have effect the latency, order of computation, and functional correctness of the hardware generated by the HLS tool. In this thesis, the memory buffers have been implemented as line buffers and memory windows.

5.6.2.1 Line Buffers

A line buffer can be considered as a two-dimensional shift register storing some lines of pixel data. These buffers are implemented as BRAMs to avoid the communication overhead with off-chip memories. The pixel values are accessed from the AXI-Stream port and are stored as rows of image data. After each computation in the memory window, the line buffer is refreshed with a new row of data. The line buffer in this thesis is implemented as a 5x512 2-dimensional matrix. So, at any time we will have access to five rows of pixel data. In working with such high dimensional data, the performance can easily be affected, especially with the shift and refresh operations. Shift operations allow the data pixels to shift from one row to the other. Refresh operations allow a row of pixels to be updated from the AXI Stream port. In sequential computing, shifting one row of data to the other row will take up to a number of clock cycles equal to the width of the image. That is, we have to spend 512 clock cycles in shifting one row of data to the other, which becomes the bottleneck of the operation. However, we have applied Vivado HLS array optimizations and loop optimizations. The line buffer is completely partitioned and the loop is completely unrolled. The above mentioned optimizations produce efficient shift processing where we can shift one row into the other in exactly one clock cycle. Similarly, we refresh the row data with the streaming interface but now we are restricted to the bandwidth of the interface as streaming interfaces are sequential.

5.6.2.2 Memory Window

The memory window is a subset of the line buffer and is used in the core computation of the algorithm. It contains the values of pixels which are in the neighbourhood of the current pixel. They are typically implemented in hardware as flip-flops. For the complete computation of the algorithm, the memory window is shifted pixel by pixel until we have covered every single pixel in the image. The memory window is refreshed with new data from the line buffer. The same loop optimization and array partitioning that were applied to the line buffer are applied to the memory window. The advantage of the memory window over the line buffer is that, refreshing the data takes the least time as both memory structures are completely partitioned and, hence, can be accessed independently in time. The entire computation algorithm, in terms of memory structures, is shown in Figure 5.15.

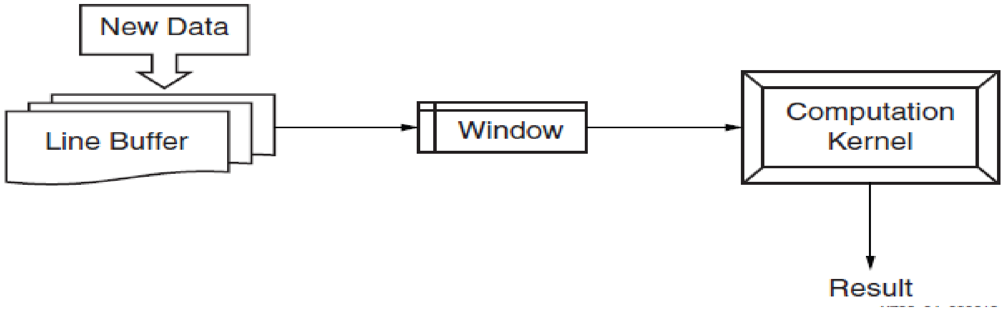


Figure 5.15 Processing of data with memory buffers.
Source:[15].

5.6.3 Arbitrary Precision Data types

To reduce hardware resource usage, various precision data types were used. Since the image pixels have 8-bit unsigned integer values, the ap_unit8 data type was used on the AXI Stream ports. The counters used to loop through the pixel values are assigned 9-bit data types since the width of the image is 512. The SE consists of logical values and, hence, for the dilation algorithm the SE data type was chosen as a 1-bit unsigned integer. Similarly, the counters used in the memory buffer and line buffers were optimized according to their range of usage.

5.6.4 Optimizations

As discussed in the preceding sections, Vivado HLS allows a number of optimization directives to be placed on the functions and loops to improve the throughput. In the implemented algorithm, all the function hierarchies were removed and most of the computation is in the form of loops. It takes one clock cycle to enter a function and one clock cycle to exit a function call. So, for a counter with a maximum value of 512, we incur an overhead of 1024 clock cycles. The intention of not using the functions is to remove this overhead. This is equivalent to making the function calls, and using the Vivado HLS optimization of function inlining and function dataflow pipelining. The top-level function is, however, pipelined to achieve the maximum possible concurrency. All the implemented loops are unrolled; arrays are partitioned completely and pipelined. This guarantees the maximum possible parallelism.

5.7 Simulation and Waveforms

Vivado HLS allows generating test benches to validate the functionality of the RTL design before synthesis by the C-Simulation. We can also use the RTL Co-Simulation tool to simulate the RTL and generate the waveforms. Test benches were generated for both the erosion and dilation IP cores, and random pixel values were fed to the RTL ports. All the waveforms were simulated in the SystemC RTL Co-Simulation mode and Value Change Dump (VCD) files were created. We have used the Synopsys custom wave viewer to generate the output waveform from the VCD files.

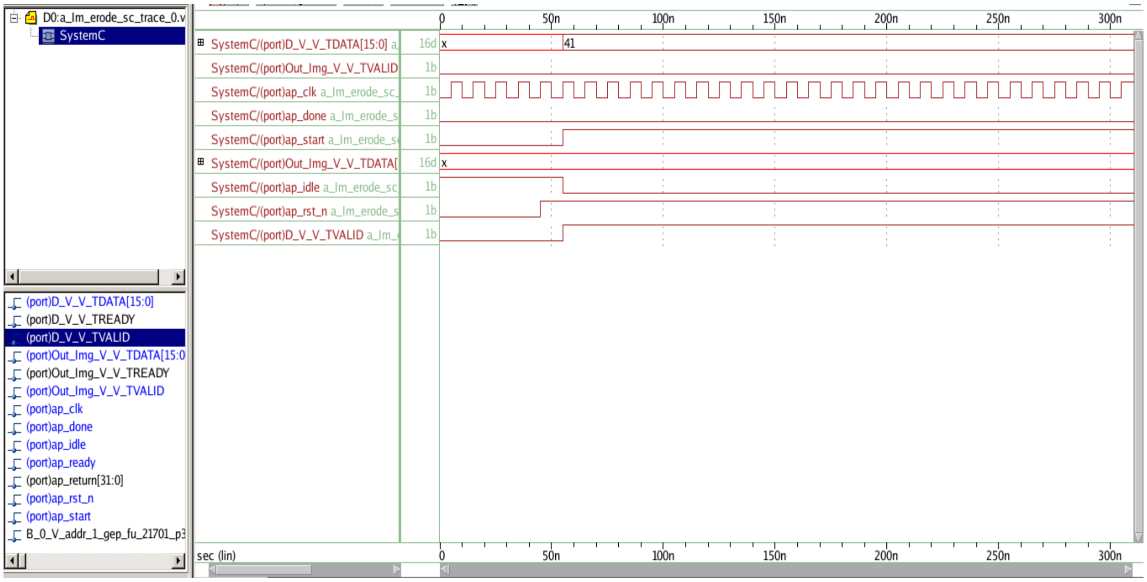

D0:a_Im_erode_sc_trace_0.v
SystemC
(port)D_V_V_TDATA[15:0]
(port)D_V_V_TREADY
(port)D_V_V_TVALID
(port)Out_Img_V_V_TDATA[15:0
(port)Out_Img_V_V_TREADY
(port)Out_Img_V_V_TVALID
(port)ap_clk
(port)ap_done
(port)ap_idle
(port)ap_ready
(port)ap_return[31:0]
(port)ap_rst_n
(port)ap_start
B_0_V_addr_1_gep_fu_21701_p3
SystemC/(port)D_V_V_TDATA[15:0] a_
SystemC/(port)Out_Img_V_V_TVALID
SystemC/(port)ap_clk a_Im_erode_sc_
SystemC/(port)ap_done a_Im_erode_s
SystemC/(port)ap_start a_Im_erode_s
SystemC/(port)Out_Img_V_V_TDATA[
SystemC/(port)ap_idle a_Im_erode_sc
SystemC/(port)ap_rst_n a_Im_erode_s
SystemC/(port)D_V_V_TVALID a_Im_
16d
1b
x
41
0
50n
100n
150n
200n
250n
300n
sec (lin)

Figure 5.16 Waveform generated for the erosion IP core showing the beginning of the transaction.

In the waveform of figure 5.16, we can see that in the first few clock cycles the ap_start block is zero and, hence, the input and output ports show uninitialized values. After the ap_start goes high, data transactions on the ports begin. As soon as ap_start goes high the ap_reset_n port goes low indicating the beginning of data processing in the IP core. The first pixel data is read into the streaming data port D with the corresponding TVALID port going high. As the first output occurs after three rows of input pixels are processed, the output streaming data port shows invalid data on port Out_Img. This validates that the RTL design is working as expected. Figure 5.17 shows the waveform for a streaming input.

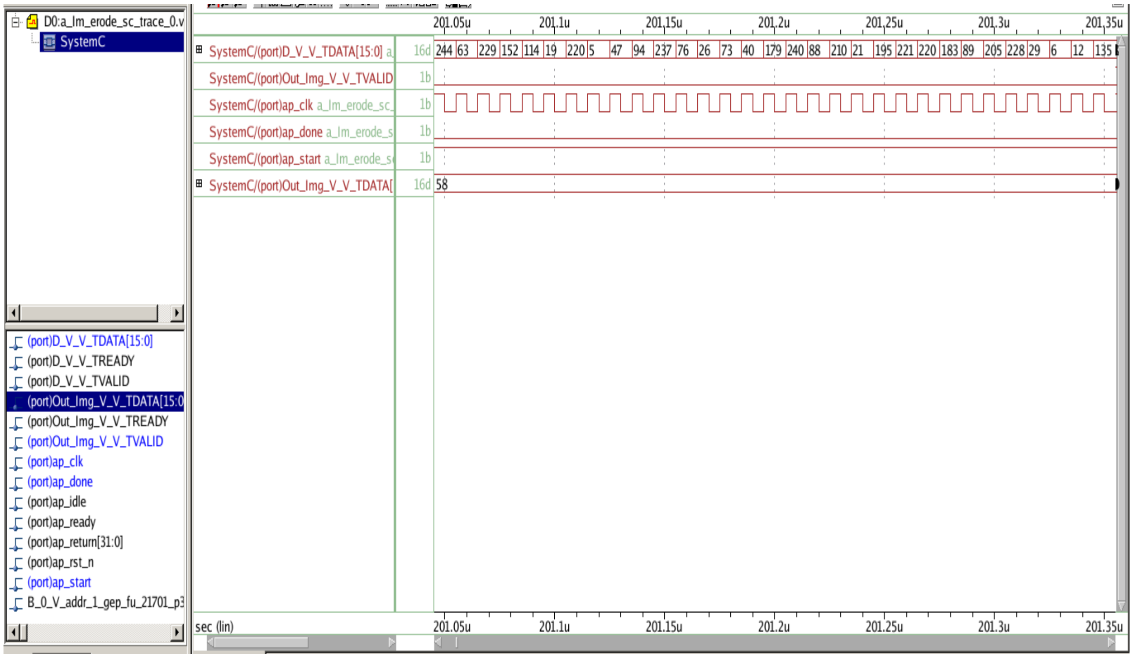


Figure 5.17 Waveform generated for the erosion IP core showing streaming input pixel values.

CHAPTER 6

PERFORMANCE AND POWER ANALYSIS OF PARTIAL RECONFIGURATION

6.1 Introduction

Chapter 3 discussed the complete design of our embedded system and the interfacing of the PL with the ARM processor. Also, the hardware accelerators which were implemented in the PL section were discussed in Chapter 3; the algorithms were implemented in MATLAB. The morphological operators of dilation and erosion were taken as the basic building blocks for the design and the synthesis of the hardware cores using Vivado HLS were discussed in Chapter 5. Furthermore, the performance morphological of the algorithms was profiled with MATLAB. In this chapter, we discuss the various benefits and trade-offs when using DPR as compared to using static configuration, of an FPGA. Also, a performance comparison is drawn with respect to executing the algorithms on the embedded system vs the MATLAB implementation.

6.2 Comparison of Hardware Resource Utilization

As you recall from Chapter 2, an algorithm run on a GPP is executed sequentially and, as such, many algorithms or applications which possess inherent parallelism cannot be executed to their best potential speed. And, hence, if an algorithm's parallelism can be exploited for faster execution, we could shift to the reconfigurable computing paradigm using FPGAs. In an effort to implement an algorithm at the highest possible speed, we often tend to consume an inordinate amount of hardware resources on the FPGA. Most of the FPGA vendors offer their products under various families which vary upon their hardware resources. For example, Xilinx offers different FPGAs in the Vertex, Zynq, Spartan and other families. Out of these, Virtex-Ultra Scale offers the highest number of hardware resources (4,432,680 logic cells, 2880 DSP Slices, etc.) whereas the Artix-7 series has the lowest number (215,360 logic cells, 740 DSP Slices, etc.). The designers can choose from this wide range of products depending upon their target application and resource consumption.

However, as an alternative the designers can choose to benefit from the DPR of FPGAs, assuming that if the design can be segregated into static and reconfigurable modules. Then, by dynamically time-multiplexing the hardware functions more logic can be accommodated. In our design, we have chosen to dynamically reconfigure the basic morphological operations of dilation and erosion. We can now compare the hardware resource utilization of the static configuration design and the reconfigurable design. Figures 6.1 and 6.3 show the static and reconfigurable designs whereas Tables 6.1 and 6.2 show their resource utilization as obtained from the Vivado design suite.

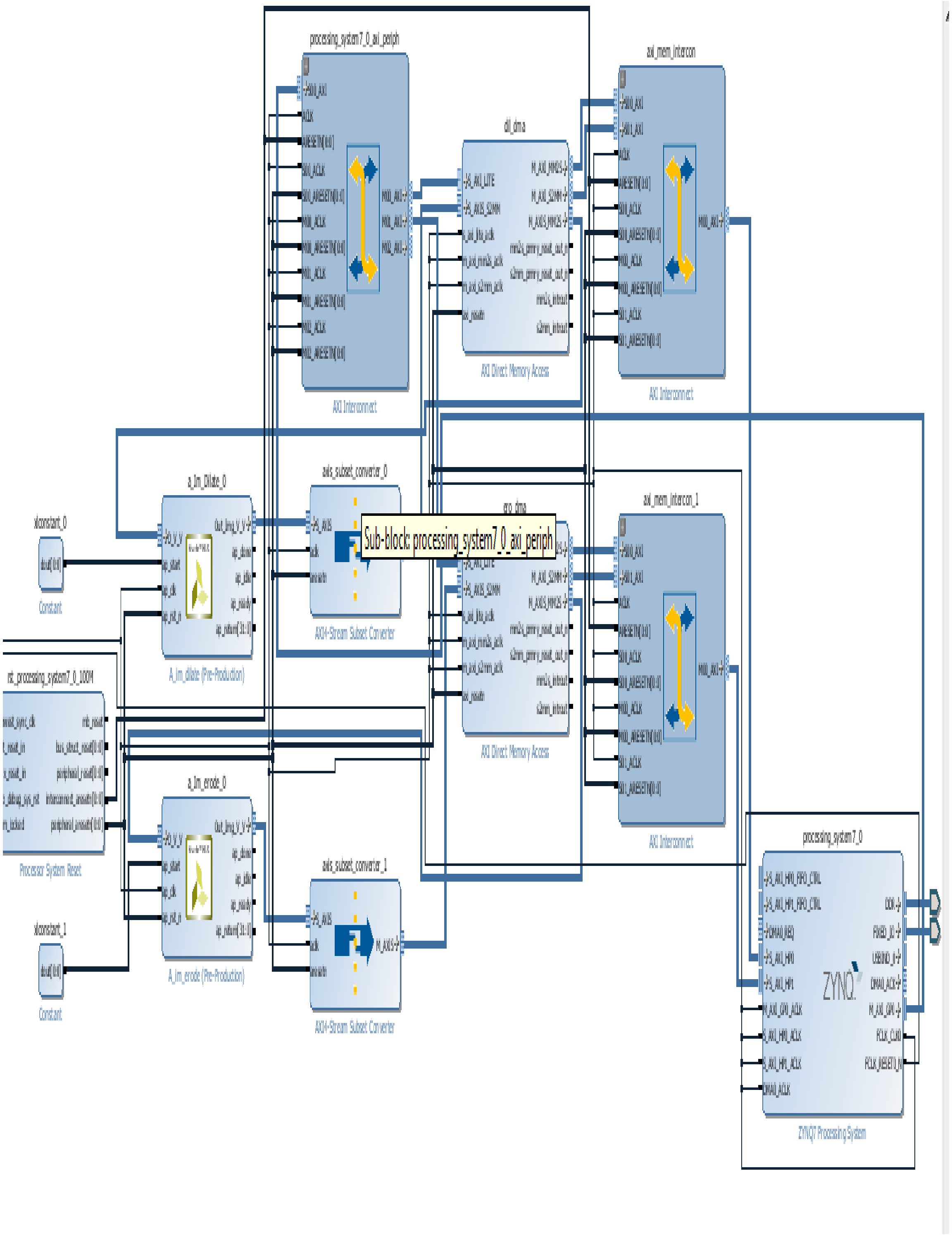


Figure 6.1 Static configuration design.

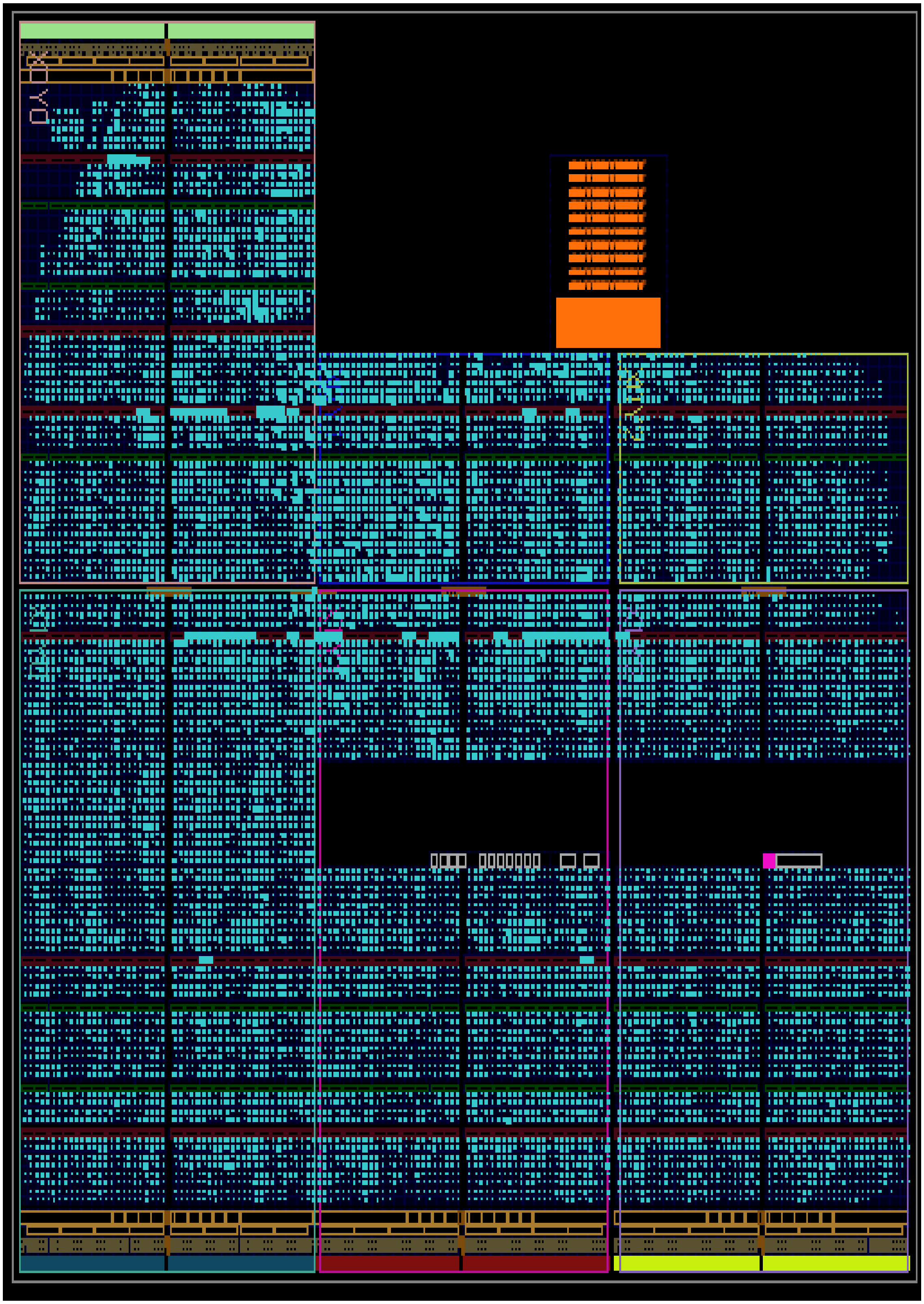


Figure 6.2 Floor design of the static configuration showing placed and routed cells.

Table 6.1 Hardware Resource Utilization for the Static Configuration.

Resources	Utilization	Available	Utilization%
Slice LUT's	21024	53200	39.52
Slice Registers	40661	106400	38.22
Memory	17	140	12.14
Clocking	1	32	3.12

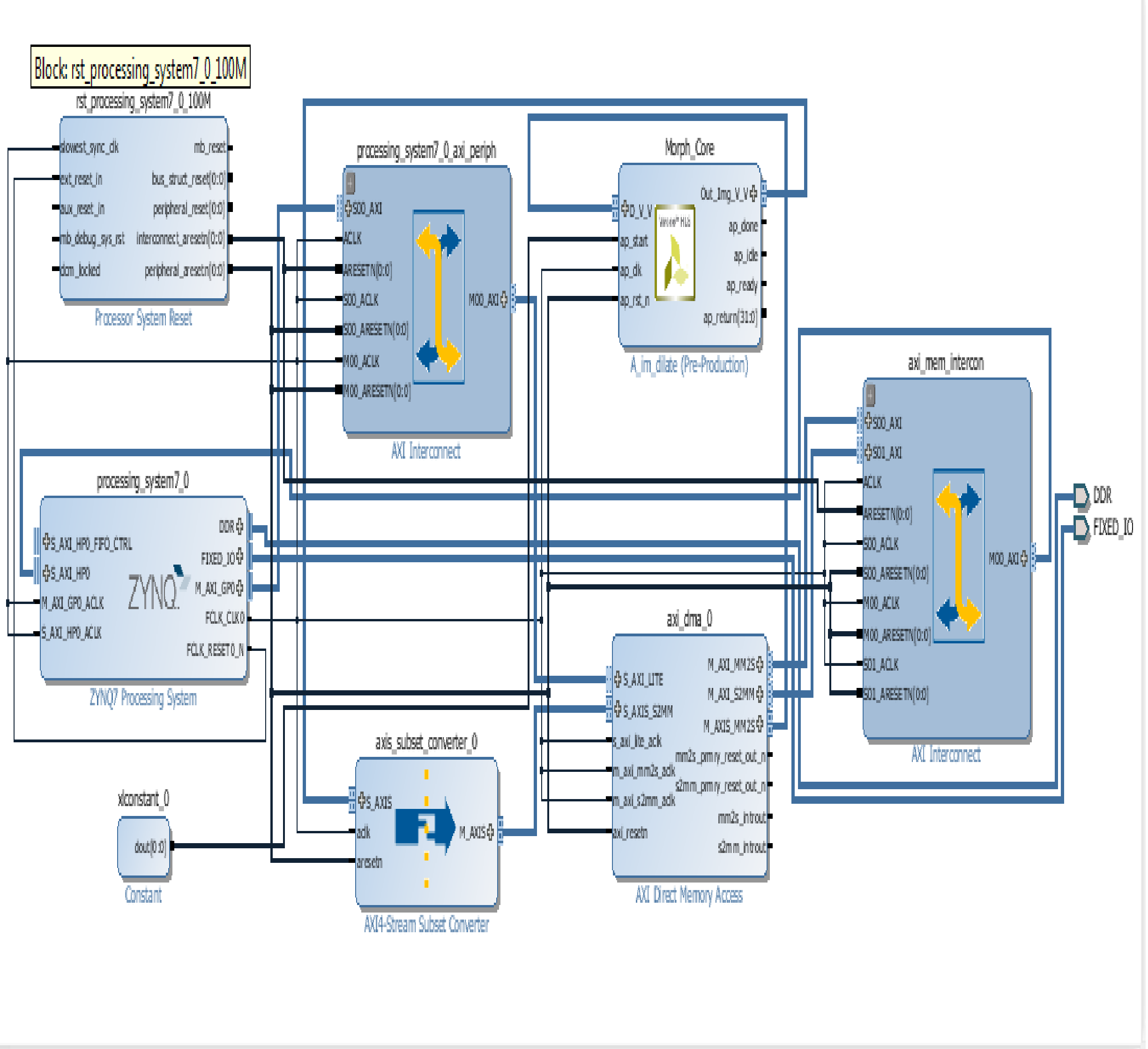


Figure 6.3 Reconfigurable design.

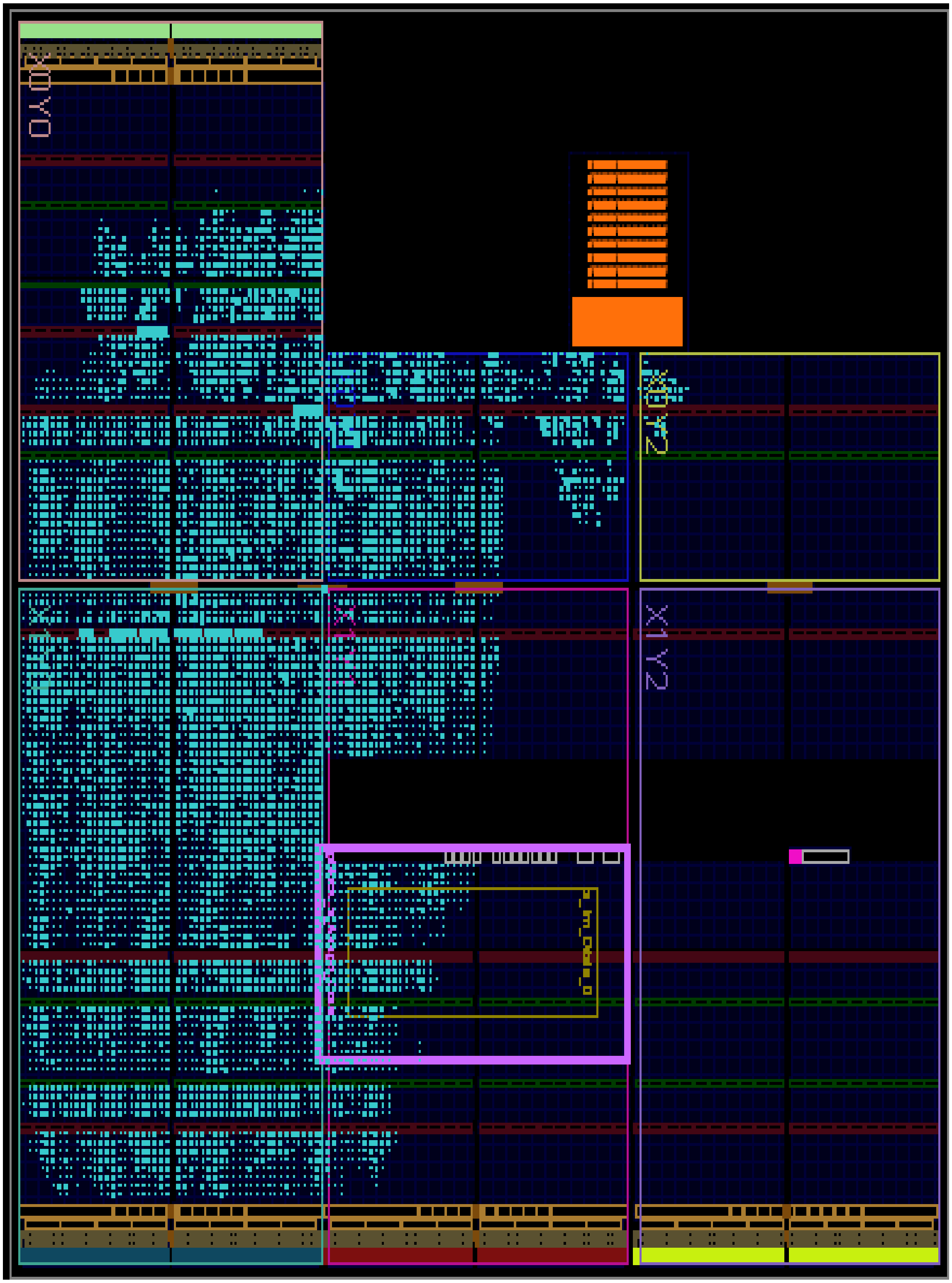


Figure 6.4 Floor design of the reconfigurable design, showing placed and routed cells.

Table 6.2 Hardware Resource Utilization for the Reconfigurable Design.

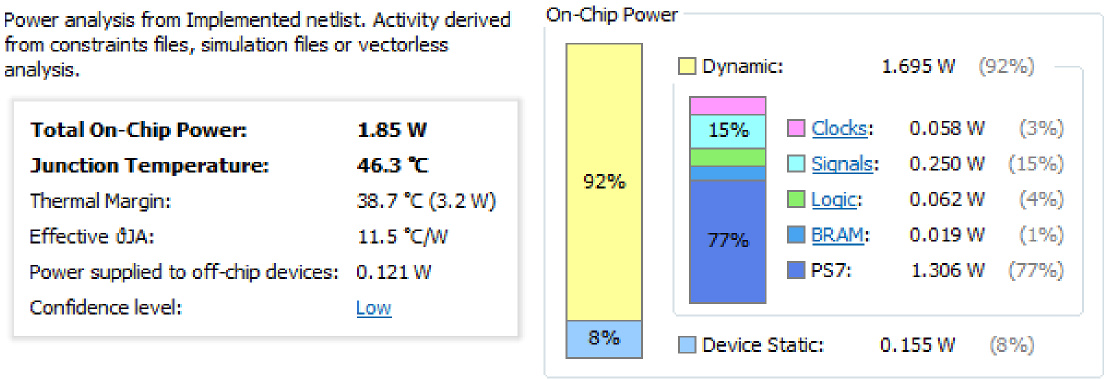
Resources	**Utilization**	**Available**	**Utilization%**
Slice LUT’s	3843	53200	6.55
Slice Registers	4000	106400	3.76
Memory	4	140	2.86
Clocking	1	32	3.12

We can clearly see from Figure 6.2, that in the static design a very high number of slice cells have been placed and routed. When compared to the reconfigurable design in Figure 6.4, we can see a much smaller number of placed and routed cells. Table 6.1 and Table 6.2 shows the exact number of slice cells and memories consumed by the respective designs. We can reduce the number of slice cells from 39.52 to 6.55 percent by using the reconfigurable design. Similarly, we see a reduction from 38.22 percent in the static design to 3.76 percent in the reconfigurable design for the configuration of slice registers.

6.3 Comparison of Power Usage

As discussed in Chapter 1, there are various methods to reduce the static power; partial reconfiguration is one of them. The idea behind power savings is that, when there are many hardware modules running on a system, not all of them are active simultaneously. But all those inactive modules do consume static power and, hence, lead to power inefficiency. Using partial reconfiguration, we can replace these circuits during their idle time with others that are actually needed in the execution. In our design, we are swapping in and out morphological IP cores on demand and, hence, save static power.

However, the process of partial reconfiguration consumes some power and ultimately its judicious use is needed to determine the overall power savings. In this thesis, the power consumption of the static and reconfigurable designs are presented as reported by the Vivado Design Suite “Report Power” utility. We can see from Figures 6.5 and 6.6 that the power consumption of the static design is higher than that of the reconfigurable design. The amount of power savings achieved by using dynamic partial configuration is 0.361 Watts or 19.5 percent over the static design. The power estimated for the static design is reported higher because more number of logic cells are active in the design and hence consume static power even if not being used at any given time. The dynamic power reported in static configuration design is estimated higher because it is assumed that all logic cells in the design with take part in active switching.



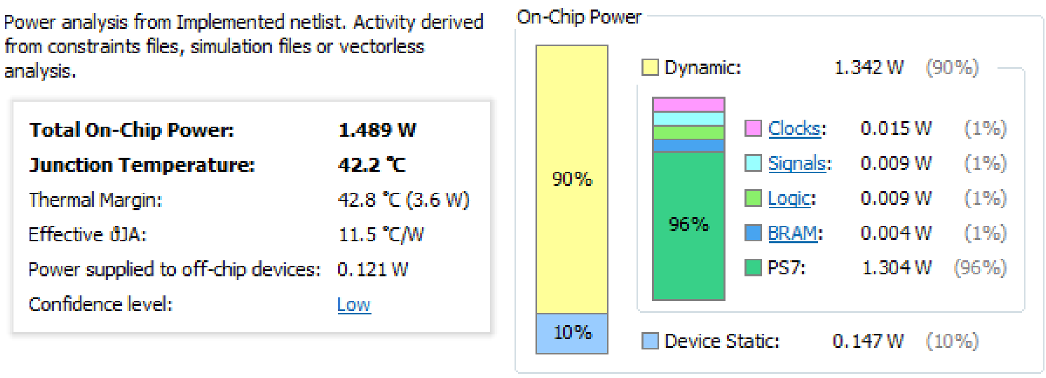
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.85 W**
Junction Temperature:	**46.3 ℃**
Thermal Margin:	38.7 ℃ (3.2 W)
Effective ϑJA:	11.5 ℃/W
Power supplied to off-chip devices:	0.121 W
Confidence level:	Low

On-Chip Power

□ Dynamic:	1.695 W	(92%)
□ Clocks:	0.058 W	(3%)
□ Signals:	0.250 W	(15%)
□ Logic:	0.062 W	(4%)
□ BRAM:	0.019 W	(1%)
□ PS7:	1.306 W	(77%)
□ Device Static:	0.155 W	(8%)

Figure 6.5 Power analysis of the static design as reported by the Vivado Design Suite.



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.489 W**
Junction Temperature:	**42.2 ℃**
Thermal Margin:	42.8 ℃ (3.6 W)
Effective ϑJA:	11.5 ℃/W
Power supplied to off-chip devices:	0.121 W
Confidence level:	Low

On-Chip Power

□ Dynamic:	1.342 W	(90%)
□ Clocks:	0.015 W	(1%)
□ Signals:	0.009 W	(1%)
□ Logic:	0.009 W	(1%)
□ BRAM:	0.004 W	(1%)
□ PS7:	1.304 W	(96%)
□ Device Static:	0.147 W	(10%)

Figure 6.6 Power analysis of the reconfigurable design as reported by the Vivado Design Suite.

6.4 Comparison of Performance on the GPP (MATLAB), Zynq (Static) and Zynq (Reconfigurable)

Various morphological algorithms were discussed in Chapter 3 and their time of execution was presented on the GPP. The same algorithms were realized on the Zynq AP-SoC with PL hardware acceleration and both static and dynamic configurations. The execution times were obtained from the hardware timer “XSCUTIMER”. The hardware timer is clocked at half the CPU clock frequency. Since the CPU clock frequency is set at 666.66MHz, the hardware timer runs at 333.33MHz. Since the timer can count down from a maximum preset value, which in this case is set to 0xFFFFFFFF, it needs to be connected to the SCUGIC interrupt and the ISR (Interrupt Service Routine) takes on the responsibility to reset the counter.

Table 6.3 Performance of the morphological algorithms on MATLAB, Zynq (Static configuration) and Zynq (Partial Reconfiguration).

Operator	**MATLAB** **(Seconds)**	**Static Design** **(Seconds/Clock Cycles)**	**DPR Design** **(Seconds/Clock Cycles)**
Dilation	5.5	0.770/256673340	0.772/257340000
Erosion	4.2	0.725/241784792	0.727/242451452
Opening	8.86	1.498/499437760	1.502/500771080
Closing	7.78	1.506/502100779	1.510/503434099
Top-Hat Transform	8.50	1.489/496467951	1.493/497801271
Bottom-Hat Transform	8.04	1.507/502345978	1.511/503679298
Morphological Gradient	8.46	1.543/514528466	1.547/515861786
Morphological Smoothing	15.79	3.796/1265428564	3.800/1266761884

The GPP executing the morphological functions in MATLAB is an Intel i5 quad-core processor which runs at a maximum frequency of 2.5GHz. From Table 6.3, we can find out that the overall speedup achieved by switching to the FPGA domain is more than 6 times. When comparing the execution time between DPR execution and static execution of the algorithms, we find out that the DPR executions are marginally slower than their static counterparts. This is because of the timing overhead accrued because of swapping in and out the different modules. However, when compared in terms of hardware resource utilization, the savings are substantial. Moreover, when multiple images need to be processed, we can further reduce the partial reconfiguration timing overhead. It is finally left to the designer to decide whether partial reconfiguration is required or not for high performance while requiring less area.

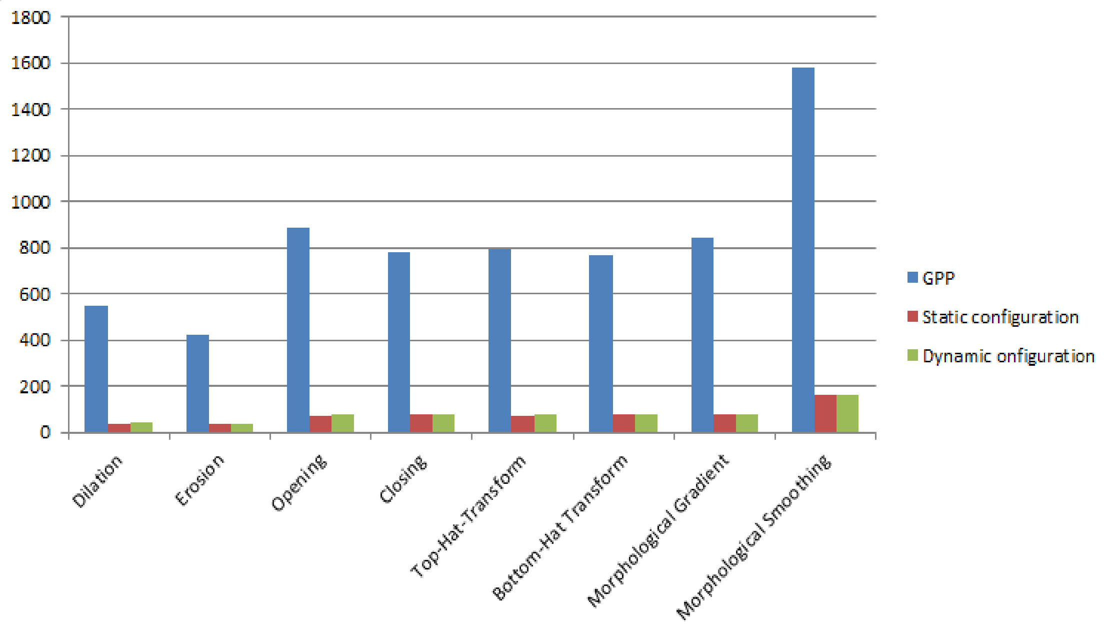


Figure 6.7 Scaled execution time on the GPP, and on the FPGA for static and dynamic configurations.

CHAPTER 7

CONCLUSIONS

In this thesis, an approach was studied to design an embedded system targeting at morphological image processing applications. The embedded system consists of an ARM host processor interfaced with a programmable logic section. To design the embedded system efficiently, a special feature of FPGAs, called dynamic partial reconfiguration (DPR), was used. DPR allows to time-multiplex the functionality of an FPGA area by dynamically swapping in/out hardware modules at run time.

The hardware modules needed by morphological image processing functions were designed in advance. Although these low level morphological functions are seldom used on their own, they are very useful in building other high-level image processing applications. The corresponding morphological functions were also deployed in MATLAB to study their run-time on GPPs. A static image processing pipeline was designed and then was reconfigured at run time for the FPGA. The execution time, power consumption, and hardware resource consumption of the static and reconfigurable designs were studied. It was determined that both the static and reconfigurable designs provide a speed up larger than six compared to a GPP for the implementation of the corresponding image processing applications. A comparison of static and reconfigurable designs shows that the DPR-based design uses less hardware resources and drains less power while providing high performance computational power. Such high performance computing power is typically expected from a real-time embedded system where time is of critical essence for the application. Therefore, DPR provides high performance while reducing the required area and power consumption.

REFERENCES

1 S. Suresh, S.F. Beldianu and S.G. Ziavras, “FPGA and ASIC Square Root Designs for High Performance and Power Efficiency,” 24th IEEE International Conference on Application-specific Systems, Architectures and Processors, June 2013.

2. C. Bobda, *Introduction to Reconfigurable Computing Architectures, Algorithms,and Applications*, Published by Springer: Dordrecht, The Netherlands, 2007.

3. Z. Huang, S. Malik, N. Moreano, and G. Aruajo “The Design of Dynamically Reconfigurable Datapath Coprocessor” ACM Transactions on Embedded Computing Systems, Vol. 3, No. 2, May 2004.

4. Xilinx homepage: http://www.xilinx.com/fpga/; Xilinx Inc. Accessed on 11/25/2014.

5. 7 Series FPGAs SelectIO Resources UG471 (v1.4), Xilinx Inc. May 13, 2014.

6. V. George, “Low Energy Field-Programmable Gate Array,” Ph.D. Dissertation, UC Berkeley, 2000.

7. Zynq-7000 All Programmable SoC Technical Reference Manual UG585 (v1.7), Xilinx Inc. February 11, 2014.

8. C.V. Borkute1, A.Y. Deshmukh “ Run time Dynamic Partial Reconfiguration using Microblaze Soft Core Processor for DSP applications.” IJRET: International Journal of Research in Engineering and Technology. eISSN: 2319-1163 | pISSN: 2321-7308.

9. LogiCORE IP AXI Interconnect v2.0 Product Guide for Vivado Design Suite PG059, Xilinx Inc. March 20, 2013.

10. LogiCORE IP AXI DMA v7.0 Product Guide for Vivado Design Suite PG021, Xilinx Inc. March 20, 2013.

11. LogiCORE IP AXI4-Stream Interconnect v1.1 Product Guide PG035, Xilinx Inc. December 18, 2012.

12. G.V. Tcheslavski “Image Processing: Gray-scale Morphology”. http://ee.lamar.edu/gleb/dip/index.htm. Accessed on 10/11/2014.

13. A. Meijster “Efficient Sequential and Parallel Algorithms for Morphological Image Processing.” Ph.D. Thesis, University of Groningen, The Netherlands, 2004, ISBN-90-367-1978-x.

14. Vivado Design Suite User Guide-High-Level Synthesis UG902 (v2014.1), Xilinx Inc. May 30, 2014.

15. F.M. Vallina. “Xilinx Application note Implementing Memory Structures for Video Processing in the Vivado HLS Tool.” XAPP793 (v1.0), Xilinx Inc. September 20, 2012.

16. A. Telikepalli, “Power vs. Performance: the 90nm Inflection Point,” Xilinx White Paper (WP223), April 2005.