5-1-2008

Reconfigurable hardware for color space conversion

Sreenivas Patil

Reconfigurable Hardware for Color Space Conversion

by

Sreenivas Patil

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science
in
Electrical Engineering

Approved By:

Dr. Eric Peskin
Thesis Advisor

Dr. Eli Saber
Thesis Committee

Dr. Sohail A. Dianat
Thesis Committee

Dr. Vincent Amuso
Department Head

Department of Electrical Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2008

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: Reconfigurable Hardware for Color Space Conversion

I, Sreenivas Patil, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Sreenivas Patil

Date

Dedication

To my parents and family, Dr. K. V. Reddy and family, and friends.

For making it possible for me to complete my studies in graduate school.

For the continued support and patience.

For helping me to be a better person.

For inspiring me to reach higher.

For the unconditional love.

For always being there for me.

I dedicate this thesis to you.

Acknowledgments

I would like to thank Dr. Eric Peskin for giving me the opportunity to be a part of this research project. He has significantly improved my knowledge about digital design and its application in the industry. He has always supported me in my research work, discussed and critiqued my ideas, and also provided different perspectives to consider in my work. He helped provide structure and clarity to my paper and thesis through numerous proofreading sessions. Dr. Peskin was so committed to my success on this project that he gave up time with his family on late nights and weekends so he could provide me with additional support.

I would also like to thank Dr. Eli Saber, Dr. Vincent Amuso, and Dr. Sohail Dianat for their reviews and suggestions during our weekly discussions. They took time from their schedules to provide their expert opinions and insights into my project work. Thank you to the professors for taking the time to be a part of thesis committee.

A special thanks goes to Hewlett-Packard, especially Dr. Kenneth Lindblom, Mr. Brad Larson, and Mr. Gene Roylance for supporting this research work and for the technical guidance they have provided with tools, documentation, and information on design and implementation. I would also like to thank Xilinx, Inc. for donating the software that I have used in this thesis.

Thank you to the Rochester Institute of Technology Electrical Engineering Department for providing me with the software, hardware, and technical support for my research work.

I would also like to thank my colleagues: Mr. Mustafa Jaber, Mr. Luis Garcia, Mr. Harsha Narne, Mr. Prudhvi Gurram, Mr. Kartheek Chandu, Mr. Manoj Reddy, Mr. Guru Balasubramanian, Mr. Bhargava Chinni and Mr. Juan Galindo for their valuable advice on image processing concepts and MATLAB usage.

Abstract

Color space conversion (CSC) is an important application in image and video processing systems. CSC has been implemented in software and various kinds of hardware. Hardware implementations can achieve a higher performance compared to software-only solutions. *Application specific integrated circuits* (ASICs) are efficient and have good performance. However, they lack the programmability of devices such as *field programmable gate arrays* (FPGAs).

This thesis studies the performance *vs.* flexibility tradeoffs in the migration of an existing CSC design from an ASIC to an FPGA. The existing ASIC is used within a commercial color-printing pipeline. Performance is critical in this application. However, the flexibility of FPGAs is desirable for faster time to market and also the ability to reuse one physical device across multiple functions. This thesis investigates whether the reprogrammability of FPGAs can be used to reallocate idle resources and studies the suitability of FPGAs for image processing applications. In the ASIC design, two major conversion units that are never used at the same time are identified. The FPGA-based implementation instantiates only one of these two units at a time, thus saving area. Reconfiguring the FPGA switches which of the two units is instantiated.

The goal is to configure the device and process an entire page within one second. The FPGA implementation is approximately a factor of three slower than the ASIC design, but fast enough to process one page per second. In the current setup, the configuration time is very high. It exceeds the total time allotted for both configuration and processing. However, other methods of configuration seem promising to reduce the time. Evaluation of the performance of the implementation and the reconfiguration time is presented. Methods to improve the performance and reduce the time and area for reconfiguration are discussed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Commercial printers typically use *application-specific integrated circuits* (ASICs) in their color-processing pipelines. ASICs are efficient and can achieve higher performance compared to software implementations. However, they are inflexible, and incorporating new features requires designing and fabricating a new ASIC. This incurs considerable cost and lead time. Furthermore, a given ASIC may need to support multiple features that are never used at the same time. In an ASIC, unused units sit idle. In contrast, reconfigurable devices such as *field-programmable gate arrays* (FPGAs) can redeploy silicon to the task at hand. FPGAs have established an attractive point on the tradeoff spectrum between the flexibility and low cost of software and the performance of hardware. The large number of logic elements available for use is well suited for processing large streams of image data and processing many streams in parallel. They offer good performance and design flexibility. Modern FPGAs feature embedded multipliers, on-chip memory, high speed transceivers and sometimes embedded processors as a part of the device. This combination of features enables FPGAs to be used in high performance image and video processing solutions.

The work presented in this thesis is a part of an ongoing research project to develop a *dynamic reconfiguration* [1] system for hardware features. It investigates how well FPGAs are suited to functions in typical color-processing pipelines within printing applications and whether they can replace ASICs in the future. Of particular interest are the tradeoffs involved in migrating existing functions from ASICs to FPGAs. In hardware implementations of color-processing pipelines, different resources are used to implement different

functions. It is a common occurrence that the functions are mutually exclusive in time (or all of the hardware resources are not active at the same time). These idle areas of the chip can be reprogrammed to perform another function, or the same function as another area, effectively realizing parallel processing. It would be possible to swap in and out different functions based on the resource availability.

As an initial case study, this thesis considers the *color space conversion* (CSC) unit from a Hewlett Packard (HP) color processing pipeline. The CSC unit is responsible for converting images represented in one color space to another color space. The HP CSC engine is chosen as the driving example for three main reasons. First, it is a commercial ASIC design with realistic size and speed requirements. Second, it performs CSC using *color look-up tables* (CLUTs) followed by interpolation [2]. This advanced technique requires both significant storage and arithmetic processing capabilities. Third, at the core of the pipeline, it contains two main conversion units. A *3D module* is used when the input space has three channels. A separate *4D module* is used when the input space has four channels. In most of the ASIC's target applications, these two units are never used at the same time. This presents an opportunity to take advantage of the ability of FPGAs to reallocate resources to the task at hand. This thesis investigates whether the reprogrammability of FPGAs can be used to redeploy any unused resources.

Most prior work on FPGA implementations of CSC [3, 4, 5] is based on linear matrix-based methods. These are simpler and only apply to conversions that are linear across the entire color space. Han [6, 7] does present a color-gamut-mapping architecture that uses CLUTs and bi-linear interpolation. This is implemented on both an FPGA and an ASIC. However, the FPGA is used for prototyping rather than as the final implementation. Furthermore, all these works deal with input color spaces that have three channels. Hence, they do not use a separate 4D module.

This thesis presents an FPGA-based implementation of the HP CSC design. The FPGA implementation consists of two separate configurations. One instantiates the 3D module.

The other instantiates the 4D module. Switching between use of the 3D or the 4D module can be accomplished by reconfiguring the FPGA, depending on the target application. The implementation described in this thesis also sets the stage for implementing dynamic reconfiguration. This will allow for individual modules to be replaced without reprogramming the entire FPGA and unused resources to be redeployed for other purposes. This also introduces the possibility of adding new functions into the existing pipeline, without redesigning the pipeline. The FPGA implementation has been tested on two different hardware platforms. One is a *Xilinx University Program* (XUP) Virtex-II Pro Development System featuring a Xilinx Virtex-II Pro series (XC2VP30-7FF896) FPGA [8]. The other is an Annapolis Micro Systems WILDCARD-4 with a Xilinx Virtex-4 series (XC4VSX35-10FF668) FPGA [9].

The following contributions are made in this thesis:

- Implemented an existing, commercial CSC ASIC design on an FPGA.
- Obtained design speed capable of processing one page per second.
- Tested design in various modes of operation and verified correct implementation of the design on the FPGA.
- Evaluated reconfiguration time and obtained results indicating that reconfiguration can be done once per job.
- Initial analysis of methods to improve reconfiguration time to allow reconfiguration on a per-page basis.

The rest of this thesis is structured as follows. Chapter 2 reviews the background material and related work in hardware implementations of CSC. Chapter 3 describes the existing CSC ASIC design and presents the FPGA-based implementation and the design changes made to implement the two versions. Chapter 4 describes the test methodology used to ensure correct implementation. It also presents the results obtained and evaluates the performance. Chapter 5 presents concluding remarks and future work.

Chapter 2

Background

Color is a visual sensation resulting from visible light falling on the retina. The human retina has three types of color photoreceptor cone cells, each responsive to a different region in the color spectrum. So, any given color can be described using three components, provided that an appropriate weight of each component is used [10]. A color image can be represented as an array of picture elements or *pixels*, each containing a combination of the components that describe a color.

A *color space* is a method of describing and representing colors in a standard way [11]. There are many color spaces in use, such as CIE XYZ, RGB, YUV, CMY, HSV, CIE LAB, *etc.*, most of which have three components. So, a color can also be defined as a point in the three-dimensional coordinate system of a color space. However, in printing applications, a fourth component, black (K) is also used to produce more accurate images and save toner, hence the color space CMYK. There are three popular groups of color spaces used to define colors in electronic devices, mainly RGB (used in display devices), YCrCb, YIQ and YUV (used in video systems) and CMYK (used in color printing).

Color space conversion (CSC) is the process of converting the representation of a given color or image from one color space to another. Different devices such as *cathode ray tube* (CRT) displays, digital cameras, scanners and printers use different color spaces. These devices are used in conjunction, and there arises a need for a conversion between the color spaces in use. A typical application is to convert from the color space of an image sensor of a camera to the color space of a CRT display or a printer. The computations involved

in these transformations are usually nonlinear and complex in multiple dimensions [12, 2]. They are computationally intense to be implemented in software. Other applications of CSC include *joint photographic experts group* (JPEG) image compression [4], *moving picture experts group* (MPEG) decoding [5], face detection [13, 14, 15], display device modeling, device independent color reproduction and colorimetry instrumentation [16].

Conversion between color spaces presents many interesting challenges. The transformation from one space to another space is not trivial because the relationship between the spaces is generally nonlinear. It is important to preserve the color information between the original image in a source device (like a image sensor, color scanner, CRT display, digital/video camera, computer software, *etc.*) and a translated copy in a target device (like a CRT display, color printer, *etc.*). The computations involved in these transformations are usually nonlinear and complex in multiple dimensions and are computationally intense. There are a few different approaches in common use.

2.1 Model-based Transforms

Pixels in one color space can be converted to any other color space using mathematical equations [11]. A straightforward approach to create a hardware design is to implement these mathematical equations. Depending on the conversion, the mathematical models range from simple and linear to complex and nonlinear. Conversions from the color space of one physical device to another physical device are usually nonlinear [2, 11]. It is a design challenge to implement the complex equations involved in the conversions and obtain accurate results with considerable speed.

A simple and widely used numerical method is matrix transformation. This is used when the underlying color transformation can be approximated by a linear conversion. A matrix consisting of conversion constants is derived from the mathematical equations used for the conversion. Each pixel of the input image is then multiplied with the conversion matrix to generate the pixel values in the output color space.

The matrix-based approach has been implemented in different kinds of hardware. Bensaali *et al.* [3] present an FPGA-based architecture for RGB to YCrCb color space conversion that offers a speedup of 100 compared to software. Agostini *et al.* [4] present parallel and pipelined architectures for the conversion from RGB to YCbCr. The FPGA implementation has operating frequencies of 40 MHz and can be used in real-time applications such as JPEG compression. Sima *et al.* [5] conduct a case study on Y′CbCr to R′G′B′ CSC for MPEG decoding and present an FPGA-based implementation that has a 40% speedup over the original design. Bilal and Masud [17] discuss a new architecture for color space conversion from RGB to YCbCr, using the instruction set of OpenRISC 32-bit *reduced instruction set computer* (RISC) processor. Andreadis [16] presents an ASIC design that is capable of conversion from RGB to CIE L*u*v* color space in real-time, operating at speeds of 20 MHz. Andreadis *et al.* [18] present a similar ASIC design that performs conversion from XYZ to CIE L*u*v* color space in real-time, with a maximum operation speed of 20 MHz. Nsour and Abdel-Aty-Zohdy [19] discuss an ASIC design that is capable of conversion from RGB to CIE L*a*b* color in real-time. CSC *intellectual property* (IP) cores are available for use from various vendors like Alma Technologies [20], CAST Inc. [21], Athena Group Inc. [22], Xilinx [23], Altera [24], and a semi-custom implementation from Triad Semiconductor [25].

Using these designs, other conversions may be performed by changing the coefficients in the conversion matrix. However, all of the above mentioned implementations are limited to conversions in which the color transformation can be approximated by a linear conversion. They are also limited to input and output spaces with three channels. The CSC IP cores offer a small selection of conversions, but they are also limited to color spaces with three channels.

Matrix transformations can be efficiently implemented in hardware. However, such methods are only applicable to conversions that are linear throughout the entire space. Not all important CSCs fit this profile. In particular, conversions from the color space of one physical device to that of another physical device are usually nonlinear. This is because

of other physical dependencies such as the dye and toner properties, the color response of different types of media for the same colorant [12] and monitor component properties. Accurately modeling such transformations with mathematical equations poses difficulties. Simple equations do not provide enough accuracy. However, complex equations are too slow in software and too expensive in hardware.

2.2 Color Look-Up Tables with Interpolation

When the two color spaces are not trivially related, the arbitrary transformation function can be implemented using *color look-up tables* (CLUTs). In the extreme, the CLUT has an entry for every possible position in the input color space. Each entry in the CLUT stores the coordinates of the corresponding color in the output color space. With this approach, any arbitrary conversion can be implemented and no arithmetic hardware is required. While this method is more accurate than a matrix transformation, it requires significant memory resources for storing the output values. In a typical case of a eight bits per input component of a three-channel input, the look-up table would have 2^{24} or 16,777,216 entries, each containing three bytes for a three-component output or four bytes for a four-component output. The memory requirements of the pure CLUT approach quickly become impractical [2, 12].

An efficient way of reducing the memory requirements is to use CLUTs with interpolation [2]. In this approach, the CLUTs only have entries for a subset of the possible positions in the input color space. In its simplest form, the entire input color space can considered as a single cube. For example, consider a three channel input space like RGB, as shown in Figure 2.1. The primary colors (red, green and blue) and their combinations (cyan, magenta, yellow, black and white) are the vertices. There is a CLUT entry for each vertex. The output value is then calculated by interpolation between the vertices, using the distance from the vertices as weights.

In this extreme case where the entire input space is a single cube, this method reduces to the approach of Section 2.1. In the other extreme, there is a vertex at every possible

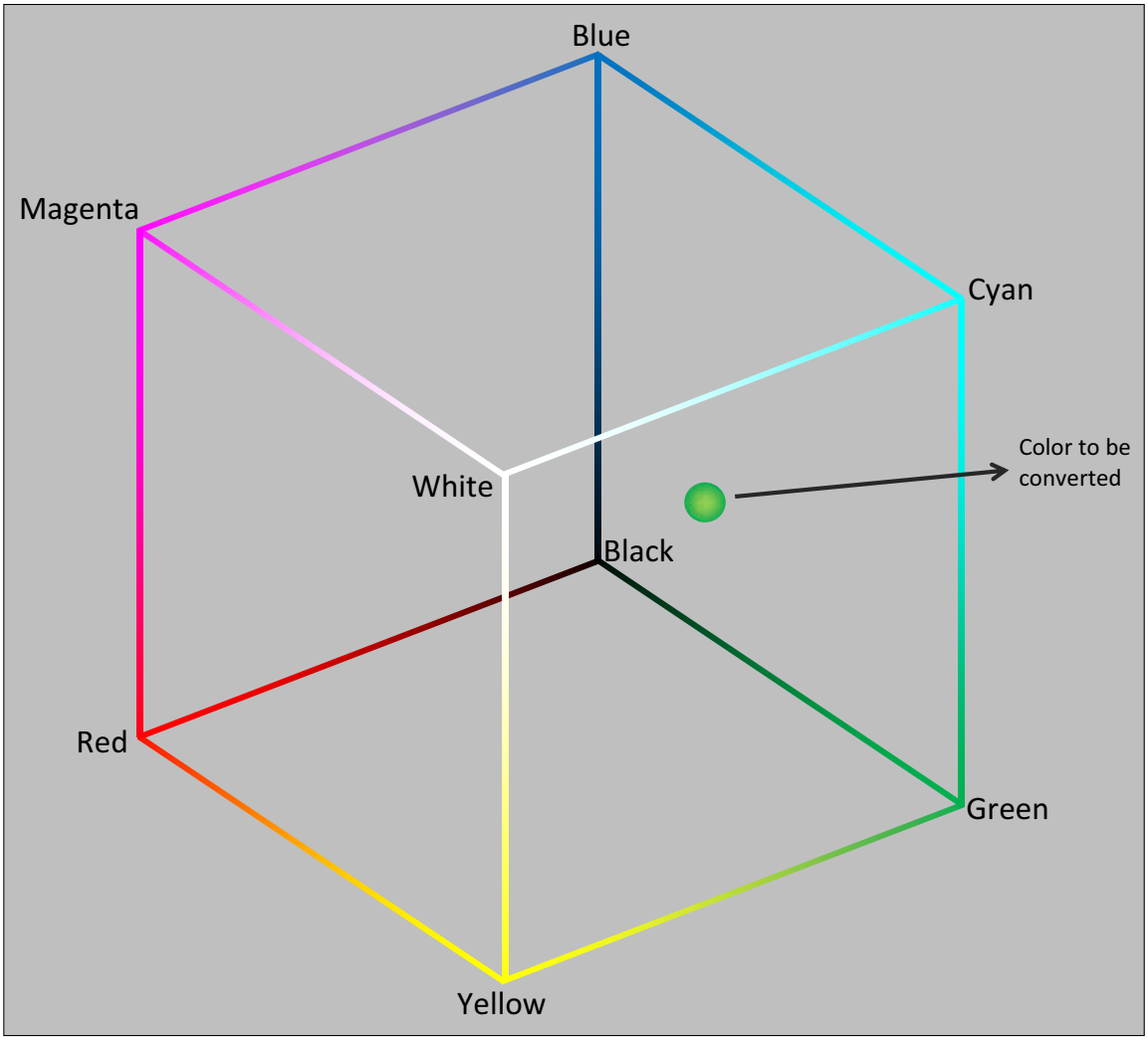

Figure 2.1: Simplest form of interpolation using eight vertices.

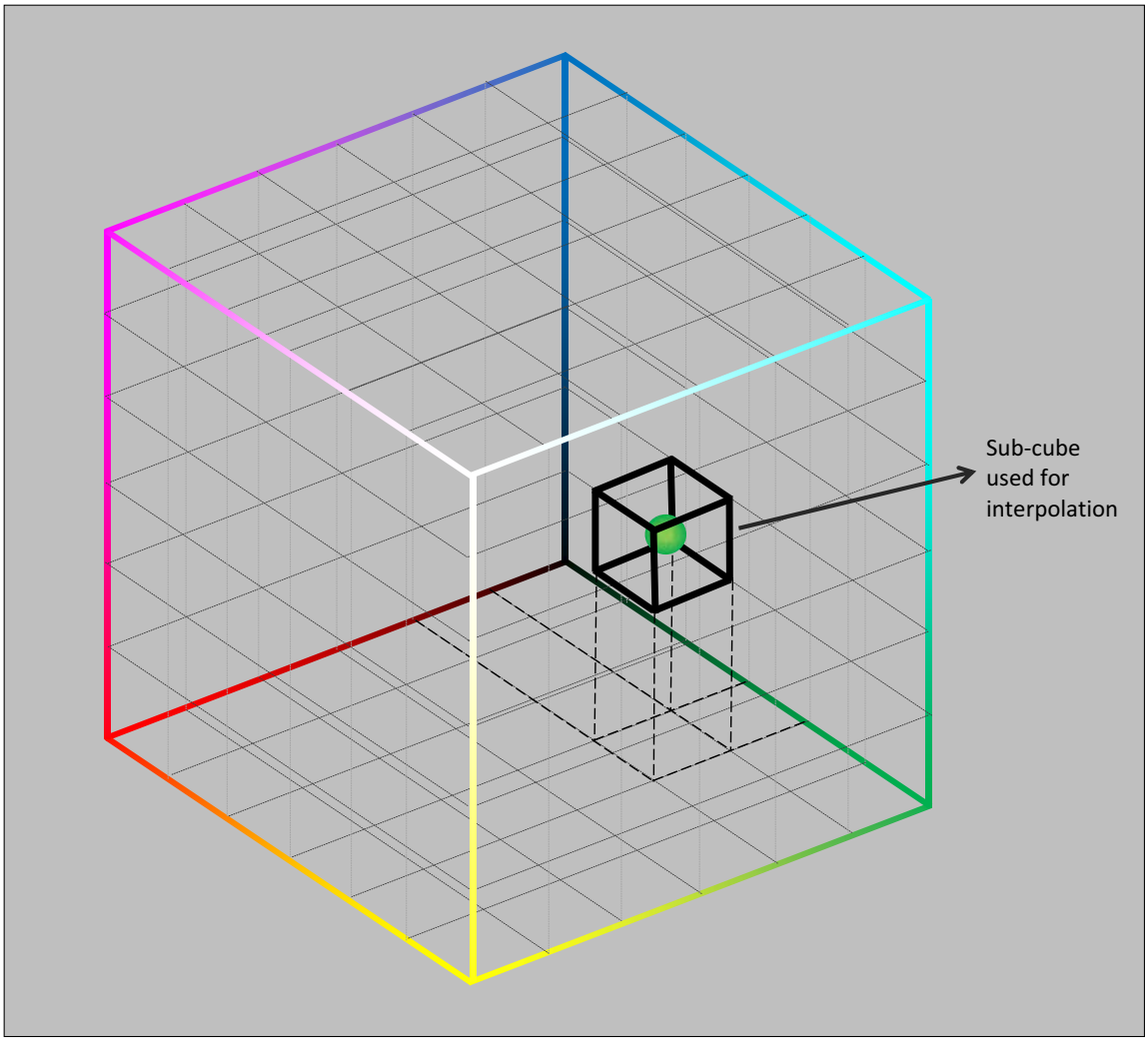

Figure 2.2: Interpolation using sub-cubes in the color space.

position in the input space and this method reduces to the pure CLUT method. In the intermediate case, the input space can be divided into sub-cubes as shown in Figure 2.2. CLUT entries are allocated for the vertices of the sub-cubes. Given an input color that lies within a particular sub-cube, the algorithm extracts the output colors stored at each of the vertices of that sub-cube. Interpolation between these values is used to determine the final output color. The size of the cubes must be chosen to balance memory requirements, arithmetic logic requirements, speed and accuracy of the conversion.

There are a number of interpolation methods used, for example: bilinear, trilinear,

PRISM, tetrahedral, *etc.* [2, 6]. The CLUTs-with-interpolation method has a balanced use of both memory and computing resources. However, the computational complexity of the methods and accuracy of the results varies, depending on which interpolation method is used. By using CLUTs with interpolation, a practical design with a wide range of conversion choices and accurate CSC results can be obtained.

A survey of current literature shows that very few implementations of the CLUT with interpolation method are available in the public domain. My hypothesis is that the implementations used in various devices are proprietary. The CSC design described in this thesis is one example. There are a few FPGA implementations of this method. Han [6,7] presents a color-gamut-mapping architecture that uses CLUTs and bilinear interpolation. The input and output color spaces each have three channels. The design is implemented on both an FPGA and an ASIC. The FPGA is used for prototyping rather than as the final implementation. In contrast, this thesis studies the suitability of FPGAs and the use of dynamic reconfiguration for related applications.

Chapter 3

Implementation

3.1 Existing ASIC Implementation

The starting point for the current research work is a design that was provided to us by HP. This design is used in an ASIC and is a part of the color pipeline. It supports a wide variety of modes and input methods. The design consists of many modules, each with a specific purpose. The configuration of the modules and the behavior of the CSC design are controlled by a set of configuration registers. The core part of the pipeline consists of a pre-processing unit, two main conversion units and a post-processing unit as shown in Figure 3.1. The 3D module handles conversions in which the input color space has three channels (such as RGB or L*a*b*). The 4D module handles conversions in which the input color space has four channels (such as CMYK). Both these modules convert the input space to a four-channel output space.

This design uses CLUTs with interpolation to perform the color space conversion. The CLUTs can be loaded with the different values corresponding to the conversion requirements. The CLUTs are accessed by the higher order bits of each input channel. These bits

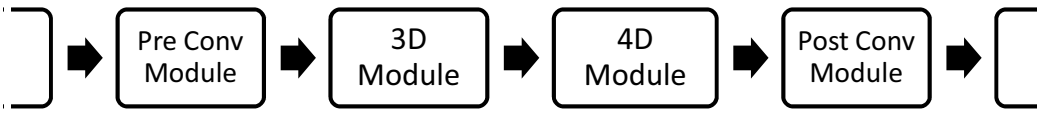

Figure 3.1: Core of the CSC engine.

identify the sub-cube that contains the input color. The output colors for the vertices of this sub-cube are extracted from the CLUT. The output value is obtained by interpolating between these values. The lower order bits of each input channel determine the relative position of the input color within the sub-cube. This is used by the interpolation algorithm to calculate the output value.

In order to convert one pixel per clock cycle, it is necessary to access all values required for interpolation in a single clock cycle. This is achieved by implementing the CLUTs using multiple memories that can be accessed in parallel. The CLUT entries are distributed such that for any color, each vertex of the selected sub-cube is guaranteed to be in a different memory [12]. Thus, all the vertices of the sub-cube can be extracted in parallel. This enables a throughput of one pixel per clock cycle.

The CLUTs also feature high and low resolution modes. The design is able to process 16-bit color data as well as 8-bit color data. The CLUTs are typically loaded through a processor register interface, but the module also provides a means to load the CLUTs via a *direct memory access* (DMA) controller. The ASIC implementation has a throughput of one pixel per clock cycle with a maximum clock frequency of 167 MHz.

3.2 Proposed FPGA Implementation

The ASIC used for CSC has the advantage of achieving higher performance compared to a software-only solution. However, it has two key disadvantages. Firstly, incorporating new features into the algorithm requires the design and fabrication of a new ASIC. This involves a considerable amount of time and cost. Secondly, even if the design remains constant, the conversion requirements or types may change from job to job, or in some cases, from page to page. This requires multiple modules, each suited for a particular CSC requirement. In most products in which the ASIC is deployed, the two conversion units are never used at the same time. At any given time, one of the two modules is idle. This can be considered an inefficient use of available device resources. In contrast, on programmable

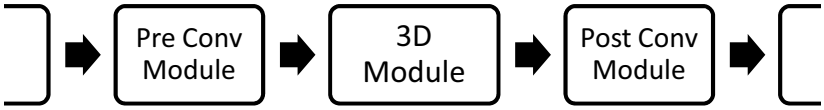

Figure 3.2: FPGA version including 3D module.

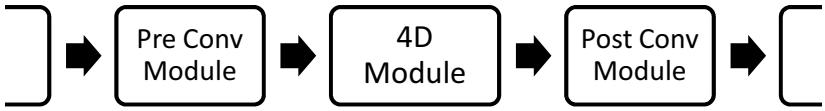

Figure 3.3: FPGA version including 4D module.

hardware such as an FPGA, these resources can be either eliminated or redeployed to the task at hand. The motivation for this thesis is to study how well are FPGAs suited for the type of CSC currently used and to use the reprogrammability of FPGAs to re-use the available hardware resources to the task at hand.

In a typical print application, only one of the two main conversion modules is used at any given time. Since the other module is idle, the design is modified such that only one of the modules is present at any given time. Thus, there are two versions of the design, each with one of the modules as shown in Figure 3.2 and Figure 3.3. The first step is to implement the existing ASIC design in an FPGA. In the process, some changes are made to the design that allow for implementation on an FPGA.

The existing ASIC design can make use of *static random access memory* (SRAM) from various vendors. Each kind of SRAM is enclosed in memory wrapper modules to keep the interface to the design constant. In the FPGA implementation, Xilinx Coregenerator is used to create these memory structures, which are implemented in *block random-access memories* (BRAMs) on the FPGA. Custom wrappers are created for each of the memory modules used.

The current ASIC design has two versions. One includes just the 3D module. The other includes both the 3D and 4D module. The chip designer has to choose between the two versions before fabricating the ASIC. The FPGA implementation also has two versions, one with just the 3D module and the other with just the 4D module. The two versions can

be swapped at runtime, by reconfiguring the FPGA with different bit streams.

As a result of the above change, the length of the pipeline also changes. In order to prevent loss of pixel information, the control module for the pipeline is also modified to compensate for the change in the pipeline length.

Another change to the design is the addition of a clock enable port. In the hardware test bench, Xilinx System Generator is used. This requires the use of a clock enable port for the imported *hardware description language* (HDL) module [26].

The ASIC design has a *built-in self test* (BIST) interface connected to the SRAMs. The BIST interface is very large and requires a large number of I/O ports. If BIST were included, there would be a shortage of I/O ports to implement the FPGA design. Also, the SRAM modules created do not include a BIST port. So, the BIST interface is not supported and is removed from the FPGA implementation.

Once these changes are made to the design, it can be tested in simulation or synthesized for *hardware co-simulation* and implementation. The FPGA can be programmed with one of the two configurations. The decision about what type of configuration and when to configure the FPGA is currently done manually. This could be automated by using a controller to manage the configurations and the programming of the FPGA.

Chapter 4

Results

4.1 Implementation

The FPGA versions of the design are synthesized for the Xilinx Virtex-II-Pro and Virtex-4 FPGAs. The synthesis tool used is Xilinx ISE 8.2.03i. The resource usage of each of the implementations is shown in Table 4.1 and Table 4.2. The values for resource usage are obtained after post place and route implementation. The values for design speed are obtained from the post-place-and-route static timing analysis.

The FPGA implementations occupy a large number of resources, especially the block RAMs. CSC using CLUTs with interpolation is an inherently memory-intensive application. Furthermore, the CLUTs in this design are not of the standard sizes of memory available as BRAMs on-chip. Hence, each instantiation of a memory block uses more than the required memory size.

Table 4.1: Implementation results for Virtex-II Pro (XC2VP30-7FF896).

Feature	CSC Design with 3D Mod.	CSC Design with 4D Mod.	Available Resources
Slice Flip Flops	3,817	4,268	27,392
4 input LUTs	13,409	15,153	27,392
Slices	7,737	9,292	13,696
Block RAMs	92	66	136
MULT18x18s	32	40	136
Max. clock rate	50.56 MHz	50.39 MHz	

Table 4.2: Implementation results for Virtex-4 (XC4VSX35-10FF668).

Feature	CSC Design with 3D Mod.	CSC Design with 4D Mod.	Available Resources
Slice Flip Flops	3,832	4,236	30,720
4 input LUTs	13,725	15,205	30,720
Slices	7,930	9,297	15,360
RAMB16s	92	66	192
DSP48s	16	24	192
Max. clock rate	50.33 MHz	50.95 MHz	

The goal of the implementation is to be able to configure the FPGA and then process an entire page within one second. One page is 8.5 by 11 inches at 600 *dots per inch* (DPI), which is about 33 million pixels. The FPGA versions maintain the throughput of one pixel per clock cycle (as in the ASIC version). Based on the post-place-and-route static timing analysis, the maximum operating speed of the FPGA implementations is about 50MHz. This is approximately a factor of three slower than the ASIC. However, at this rate, processing one page would take approximately 0.7 seconds.

4.2 Testing

Each type of conversion has a specific CLUT data file. These custom data files are processed in a HP software executable to generate the configuration register information, CLUT values and a CSC data file. The CSC data file is used to process the source image using another HP software executable that simulates the ASIC design. The result is a reference output image that is used for comparison with the simulation results. The configuration register information and CLUT values are used along with the image information to create an input text file containing test vectors. (The structure of the text file containing the test vectors is available in Appendix A.) This text file is used as an input to the Xilinx ISE simulator to perform the software simulation. Image information is extracted from the simulation output and compared against a reference output image obtained from the software executable. Figure 4.1 details the test method.

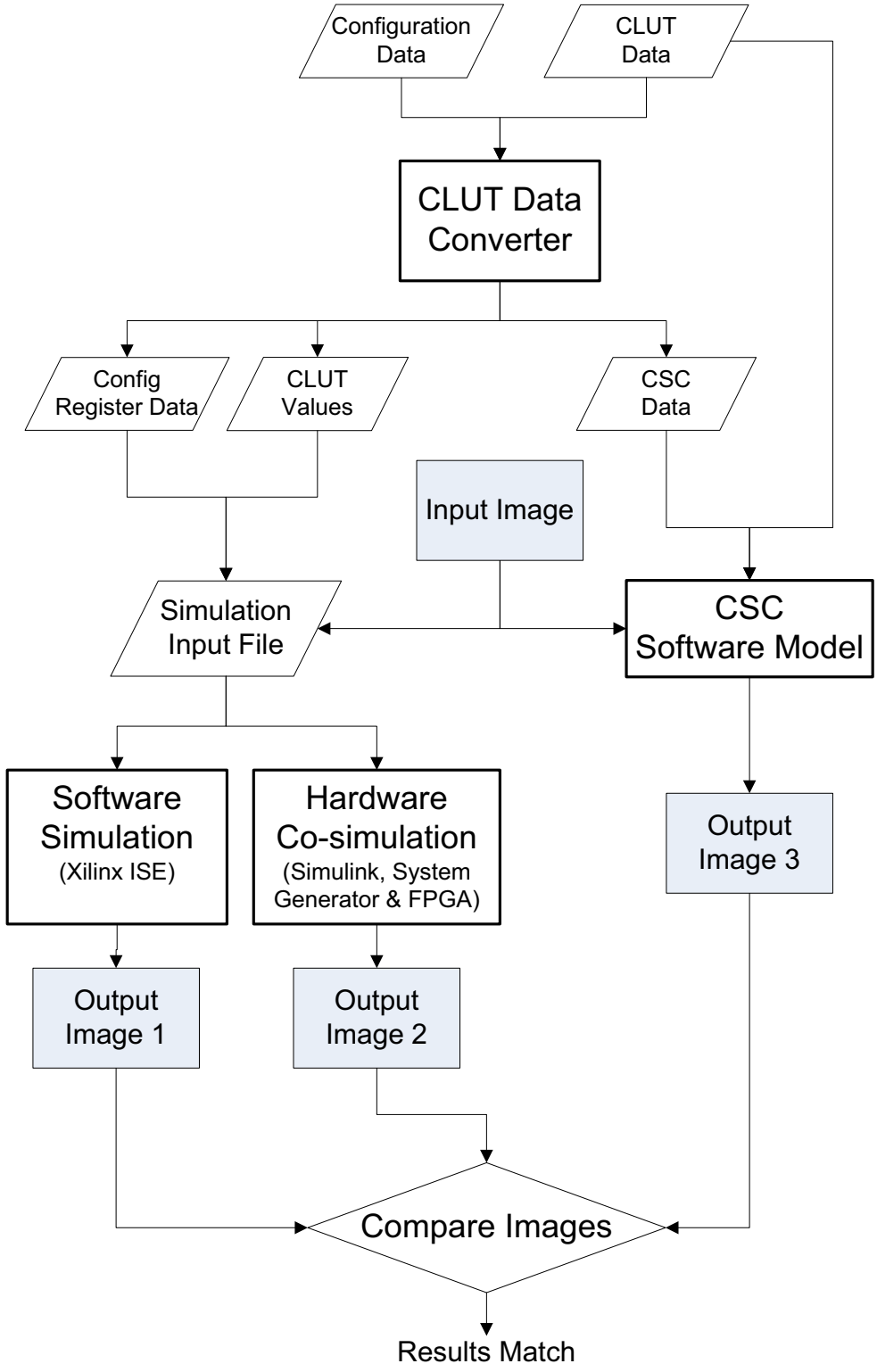

Figure 4.1: Test methodology.

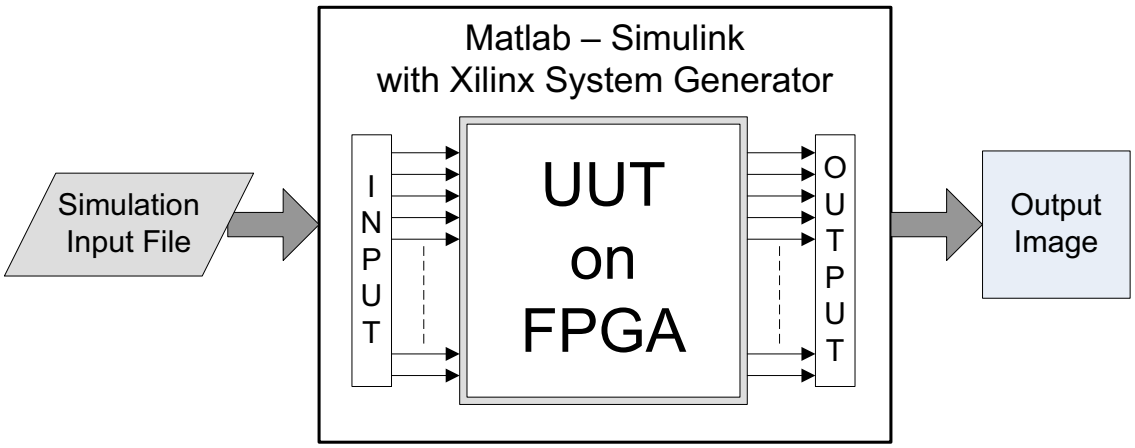

Figure 4.2: Hardware-in-the-loop testing.

Once the design is verified in software, it can be tested in hardware. Xilinx System Generator is used for hardware-in-the-loop testing. This gives the ability to use MATLAB via Simulink as a hardware test platform. This interface is very convenient because it allows for testing the design on the FPGA and also allows for sending test vectors and receiving output values from within MATLAB. The design to be tested is imported as a *black box* component in Simulink. *Gateway-in* and *gateway-out* components are used to connect the input and output ports of the design to the MATLAB workspace. This Simulink model is synthesized for the target device and a hardware co-simulation block is created. (More information about the Simulink model and screen shots are available in Appendix B.) Test vectors, including both configuration data and an input image are setup in MATLAB using the input text file and applied to the inputs of the design on the FPGA. The outputs are collected, the image information is extracted and compared with the results of the software model and a match can be validated. This is illustrated in Figure 4.2. The test results are expected to have a bit-for-bit match with the reference output image.

Even though Xilinx System Generator provides a convenient interface, the speed of testing is limited by the slow link between the host and the FPGA. Vectors stream through the host computer, then MATLAB and System Generator and finally the FPGA. The test vectors are shifted serially from the host computer and then applied to the design. Table 4.3

Table 4.3: Hardware-in-the-loop testing – XUP Development System.

Num. of Pixels		Num. of Vectors	Config Time	Proc. Time	Time per Vector	Vector Rate
Image 1	9600	17234	7.7s	21.1s	1.6×10^{-3}s	636 Hz
Image 2	19200	26834	7.9s	41.8s	1.6×10^{-3}s	642 Hz
Image 3	38400	46034	8.1s	71.3s	1.5×10^{-3}s	646 Hz
Image 4	57600	65234	8.2s	100.2s	1.5×10^{-3}s	641 Hz

Table 4.4: Hardware-in-the-loop testing – Annapolis WILDCARD-4.

Num. of Pixels		Num. of Vectors	Config Time	Proc. Time	Time per Vector	Vector Rate
Image 1	9600	17234	2.5s	3.8s	2.2×10^{-4}s	4.5 KHz
Image 2	19200	26834	2.5s	5.8s	2.2×10^{-4}s	4.6 KHz
Image 3	38400	46034	2.4s	9.7s	2.1×10^{-4}s	4.7 KHz
Image 4	57600	65234	2.5s	14.3s	2.2×10^{-4}s	4.6 KHz

and Table 4.4 show the time for processing four different test images. The test vectors contain the CSC configuration, CLUT values and the image pixels. The time required to configure the FPGA and to process all the test vectors is measured. In the tests performed, the effective clock rate is approximately 4.6 KHz for the Virtex-4 FPGA and approximately 641 Hz for the Virtex-II Pro FPGA. Even in case of the Virtex-4, the clock rate is four orders of magnitude slower than the maximum operating speed of the FPGA implementation. Thus, although System Generator is a convenient interface to verify correctness on the hardware, it does not allow for at-speed testing of this design.

The two FPGA versions of the CSC design have been tested in several different modes of operation. Table 4.5 shows the 24 tests performed and the different combinations of parameters used. The columns in Table 4.5 are defined as follows:

- Pipeline - indicates whether the pipeline includes the 3D module or the 4D module.
- CLUT Load Method - shows the method used to load the CLUTs.
- CLUT Resolution - shows whether the CLUTs use high resolution or low resolution.
- Conversion - indicates the specific conversion performed.

Table 4.5: Tests in the different modes of operation.

Test #	Pipeline	CLUT Load Method	CLUT Resolution	Conversion	Image Resolution	Results Match
1	PreConv-3D-PostConv	Register	High	3D4D Type 1	8-bit	Yes
2					16-bit	Yes
3				3D4D Type 2	8-bit	Yes
4					16-bit	Yes
5			Low	3D4D Type 3	8-bit	Yes
6					16-bit	Yes
7		DMA	High	3D4D Type 1	8-bit	Yes
8					16-bit	Yes
9				3D4D Type 2	8-bit	Yes
10					16-bit	Yes
11			Low	3D4D Type 3	8-bit	Yes
12					16-bit	Yes
13	PreConv-4D-PostConv	Register	Low	4D4D Type 1	8-bit	Yes
14					16-bit	Yes
15			High	4D4D Type 2	8-bit	Yes
16					16-bit	Yes
17				4D4D Type 3	8-bit	Yes
18					16-bit	Yes
19		DMA	Low	4D4D Type 1	8-bit	Yes
20					16-bit	Yes
21			High	4D4D Type 2	8-bit	Yes
22					16-bit	Yes
23				4D4D Type 3	8-bit	Yes
24					16-bit	Yes

- Image Resolution - shows the resolution of the source image.
- Results Match - shows the result of the comparison between the images obtained from hardware co-simulation and the software model.

In all the test cases, both the output from software simulation and also that from the actual FPGA exactly match (bit-for-bit) the desired result from the HP software model. Figure 4.3 shows the result of one of the tests. The conversion performed is RGB to CMYK. Figure 4.3(a) is the source RGB image. The result of the HP CSC software executable is shown in Figure 4.3(b) and the result of the hardware co-simulation is shown in Figure 4.3(c).

(a) Source Image. (b) CSC Executable. (c) Hardware Co-simulation.

Figure 4.3: Test image results.

In the current test setup, the FPGA configuration process is a part of the test routine. In the case of the XUP board, the FPGA can be programmed independently to measure the time for configuration. Xilinx Impact can configure the FPGA in 3.5 seconds. Using the test setup, configuration time is approximately 8 seconds (as shown in Table 4.3). In case of the WILDCARD-4, the time to configure the FPGA is 2.5 seconds (as shown in Table 4.4). The configuration time alone is over the one-second budget in either case. Section 4.3 presents potential solutions to this problem.

4.3 Faster Configuration: Preliminary Results

Section 4.2 shows that when using hardware co-simulation, the time to configure the FPGA externally exceeds the budget of one second to both reconfigure the FPGA and process one page. The configuration time can be reduced in two ways. One is to reduce the size of the configuration bit-stream. The other is to increase the throughput at which the bit-stream is loaded.

The FPGA versions of the CSC differ only by the main conversion module. All other modules are common to both the designs. Hence, there is no need to reprogram the FPGA with the entire CSC design. If the 3D module can be replaced by the 4D module and *vice-versa* without disturbing the other modules, then switching between the two designs could take place quickly. This can be achieved by implementing *partial reconfiguration* [27]. This allows part of the FPGA to be reconfigured, while the configuration in other areas of

Table 4.6: Physical resource usage within PRR in Virtex-II Pro (XC2VP30-7FF896).

Feature	PRR with PRM-3D	PRR with PRM-4D	Available resources within PRR
LUT	4729	7074	11200
FF	974	1399	11200
SLICE	2885	4315	5600
MULT18X18	0	0	60
RAMB16	60	34	60

Table 4.7: Physical resource usage within PRR in Virtex-4 (XC4VSX35-10FF668).

Feature	PRR with PRM-3D	PRR with PRM-4D	Available resources within PRR
LUT	4001	5725	9600
FF	924	1370	9600
SLICEL	1220	1746	2400
SLICEM	1220	1746	2400
DSP48	16	24	80
FIFO48	0	0	80
RAMB16	60	34	80

the FPGA remains unchanged. Furthermore, a *partial bit-stream* contains only the information required to reconfigure the changing region. A suitable area on the FPGA, called the *partially reconfigurable region* (PRR) [27] that can serve as either the 3D or the 4D module is reserved. The rest of the CSC design is built around the PRR. This design is synthesized and implemented to yield full and partial bit streams, which can be used to program the FPGA. Figure 4.4 and Figure 4.5 show the floor plan of the CSC design in Xilinx Plan Ahead 8.2.10. The PRR is indicated by the rectangular shaped region surrounded by the magenta border. The area surrounding the PRR is the *static region*. It contains the base design that remains unchanged during partial reconfiguration. *Partially Reconfigurable Modules* (PRMs) [27] are connected to the base design using *bus macros* [28]. The resource usage for the different PRMs in the PRR is shown in Table 4.6 and Table 4.7.

The size and shape of the PRR is determined by the resources required by the different PRMs. In the Virtex-II Pro and Virtex-4 FPGAs, BRAMs are arranged in columns. Since

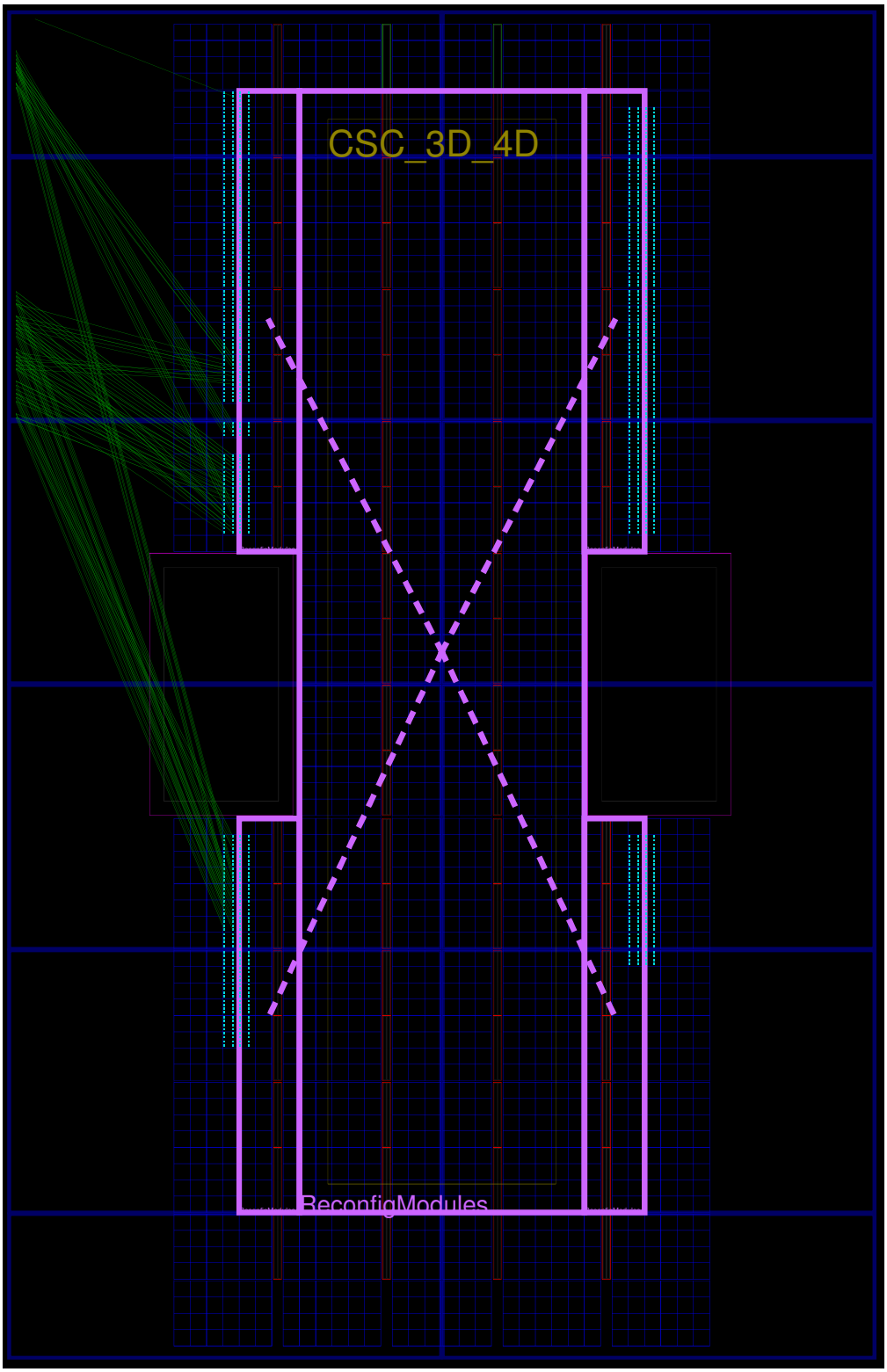

Figure 4.4: Floor plan showing PRR in Virtex-II Pro (XC2VP30-7FF896).

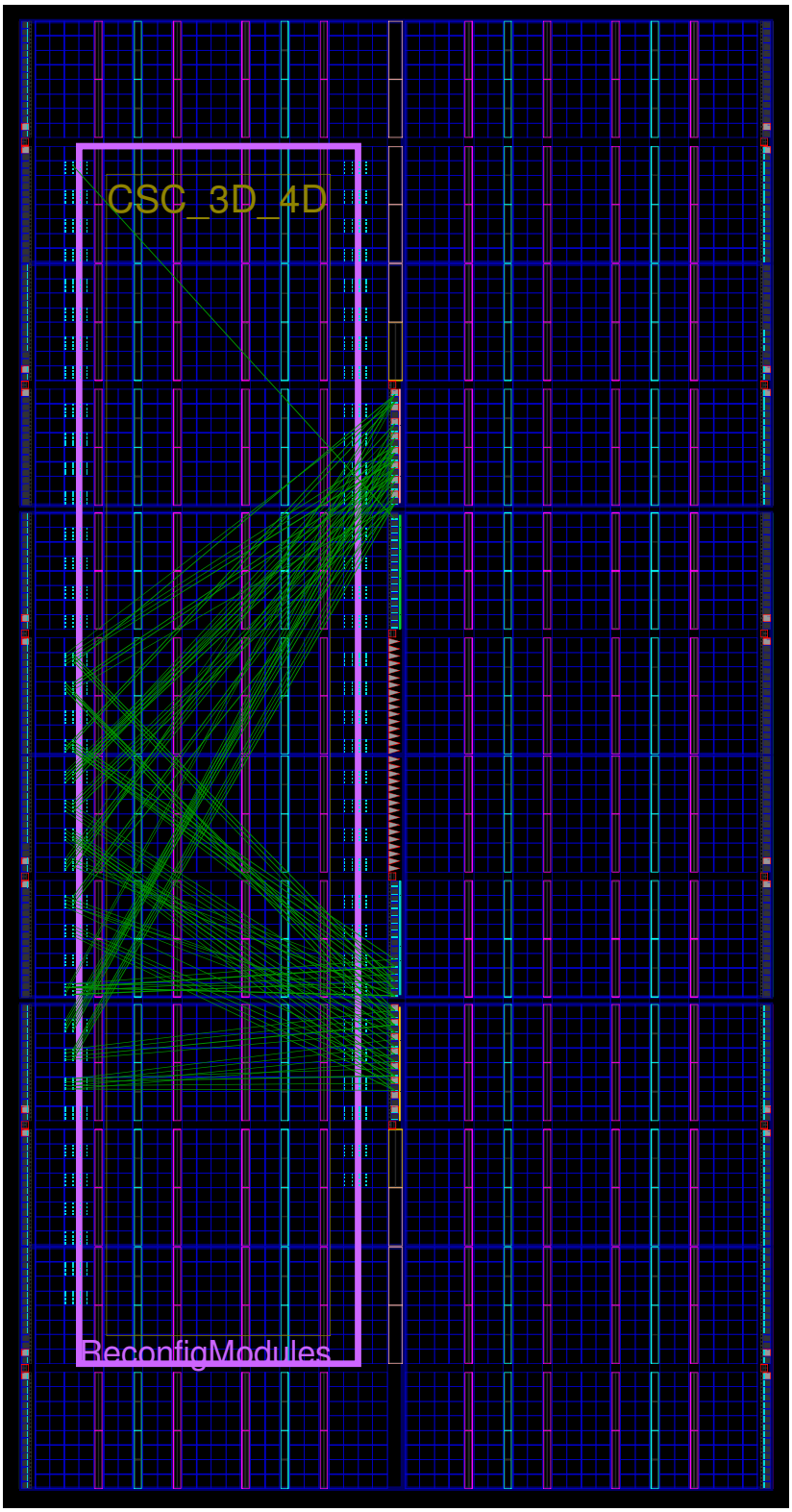

Figure 4.5: Floor plan showing PRR in Virtex-4 (XC4VSX35-10FF668).

the 3D and 4D modules make use of the BRAMs, the area covered by the PRR must include these columns. In the current implementation, the size of the PRR is chosen to meet the BRAM (RAMB16) requirements of the PRM. In the PR implementation using the Virtex-II Pro FPGA, the BRAM usage is 100% (as shown in Table 4.6). In case of the Virtex-4 FPGA, the size chosen for the PRR is slightly larger than the required size (as shown in Table 4.7). The area can be trimmed down to the exact size required.

The shape of the PRR is usually rectangular. In case of the PR implementation using the Virtex-II Pro FPGA, the PRR has a dumbbell-like shape. This is because the device contains two Power PC blocks (the two rectangular blocks in the middle of the device in Figure 4.4). In order to meet the BRAM requirements of the PRM, the area chosen for the PRR overlaps the Power PC blocks. This splits the PRR into five rectangular areas, arranged in a dumbbell-like pattern. Partial reconfiguration for PRRs of non-rectangular shapes can be performed using Xilinx Plan Ahead [29] and the software tools in [30].

In the Virtex-II Pro FPGA, the smallest unit of configuration is a *frame*, which spans the entire height of the device [27]. In the Virtex-4 FPGA, the configuration architecture is still frame-based, but a frame spans 16 rows of *configurable logic blocks* (CLBs) rather than the full device height [27]. However, using the software tools in [30] and the partial reconfiguration flow described in [28], PRR of any rectangular size can be implemented. The design flow for using Plan Ahead to generate the full and partial bit streams is described in [29, 31].

As indicated in Table 4.8, the size of the original bit stream in reduced by a factor of three in both the Virtex-II Pro and Virtex-4 FPGAs. In research experiments with the Virtex-II Pro device, it is observed that the configuration time scales linearly with the configuration bit stream size. In case of the Virtex-4, when using hardware co-simulation, the time to configure the entire FPGA externally is about 2.5 seconds. If partial reconfiguration were used to swap in a different module, the configuration time would be about 0.7 seconds. In case of the Virtex-II Pro, the time to configure the entire FPGA externally using Impact is about 3.5 seconds. If partial reconfiguration were used to swap in a different module, the

Table 4.8: Configuration time.

Device	Type	Bit-stream Size	Time (Goal $\ll$ 1s) Externally configured	Internally configured
Virtex-4	Full	1673 KB	2.5 s†	0.03 s*
	Partial	445 KB	0.7 s*	0.01 s*
Virtex-II Pro	Full	1415 KB	3.5 s‡	0.03 s*
	Partial	466 KB	1.1 s‡	0.01 s*

*Estimated values.

†Using hardware co-simulation.

‡Using Impact.

configuration time would be about 1.1 seconds. Even though this is a good reduction in the configuration time, the total time for configuring the FPGA and processing an entire page is still over the budget of one second.

Current research in improving the time for configuration involves using different methods of configuration. Virtex FPGAs can be programmed externally using SelectMAP. This interface could yield faster configuration times. Another method of configuration that holds promising results is *internal configuration access port* (ICAP) [32]. This can be used to program the FPGAs internally. This interface allows for reading and writing configuration bit streams. It has an 8-bit data port and can be run at a clock speed of 50 MHz [33, 34]. Table 4.8 shows that at this rate, reconfiguration time would be negligible compared to the time to process one full page of pixels. This would bring the total for configuration and processing within the goal of one second. Current implementation using these methods is still in its preliminary stages.

Chapter 5

Conclusions and Future Work

An existing, commercial CSC ASIC design has been successfully reimplemented in an FPGA. The FPGA implementation consists of two versions. The first version contains all units except the 4D conversion engine. The second version contains all units except the 3D conversion engine. In each case, the interface is the same as that of the original ASIC implementation of the CSC, except that BIST is not supported. Test results show an exact match between the output of the FPGA implementation and a software model of the original ASIC. The FPGA-based implementation is slower than the ASIC, but it can still process a full page of pixels in one second. This has been achieved with few FPGA-specific optimizations.

In the ASIC implementation, the pipeline contains two main conversion modules, only one of which is used. In the FPGA implementation, only one main conversion module is present in each version. This decreases the logic resources required. Since there is no need to include the bypass logic for the other module, there is also a reduction in the required routing resources.

It is important to note that an attempt has been made to keep the FPGA implementation as similar as possible to the original ASIC implementation. The CSC design is a good target for optimizations such as *partial evaluation* [35, 36]. This gives the possibility of specializing the design for a specific set of values or modes, which could potentially result in decreased resource usage and improved performance.

In order to support the various modes of the design, several multiplexers are implemented in the data path. This allows for existing features to be bypassed and alternate features to be included. If the design is implemented as-is on an FPGA, the multiplexers are also synthesized. However, if a conversion requires a specific configuration of the data path, some of the multiplexers need not be included. This would reduce the area and simplify *place and route* (PAR). However, this type of optimization requires that many bit streams be generated for the different configurations.

Another area where partial evaluation can be implemented is the CLUT configurations. The initial values of the CLUTs are zero (reset value). The values for the required conversion are loaded prior to conversion. However, if the type of conversion is known beforehand, CLUTs can be initialized with the conversion values. For other conversions, partial reconfiguration can be used to change the CLUT values. In the current implementation, the CLUTs are instantiated as *random access memories* (RAMs). If they are preloaded with the conversion values, they can be specified as *read only memories* (ROMs) instead. Thus, no interface for writing values into the CLUTs would be synthesized. This reduction in logic results in a lesser usage of resources and area on the chip.

Future work will consider methods to optimize the speed of the FPGA-based implementation. Of particular interest is the time required to switch between these two configurations. Results show that JTAG configuration is too slow for the target application. However, ongoing work investigates the use of partial configuration and internal configuration through ICAP. These methods promise to enable applications that require reconfiguring on a page-by-page basis. Future work will add a controller to automate device programming and to manage configuration bit streams.

References

[1] P. Lysaght and J. Dunlop, "Dynamic reconfiguration of FPGAs," in *More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications*, W. Moore and W. Luk, Eds., Oxford, England, 1993, pp. 82–94.

[2] J. M. Kasson, S. I. Nin, W. Plouffe, and J. L. Hafner, "Performing color space conversions with three-dimensional linear interpolation," *Journal of Electronic Imaging*, vol. 4, no. 3, pp. 226–250, 1995.

[3] F. Bensaali, A. Amira, and A. Bouridane, "Accelerating matrix product on reconfigurable hardware for image processing applications," *IEE Proceedings - Circuits, Devices and Systems*, vol. 152, no. 3, pp. 236–246, 2005.

[4] L. V. Agostini, I. S. Silva, and S. Bampi, "Parallel color space converters for JPEG image compression," *Microelectronics Reliability*, vol. 44, no. 4, pp. 697–703, April 2004.

[5] M. Sima, S. Vassiliadis, S. Cotofana, and J. T. J. van Eijndhoven, "Color space conversion for MPEG decoding on FPGA-augmented trimedia processor," in *Proceedings. IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, Jun. 2003, pp. 250–259.

[6] D. Han, "Real-time color gamut mapping method for digital tv display quality enhancement," *IEEE Transactions on Consumer Electronics*, vol. 50, no. 2, pp. 691–698, 2004.

[7] ——, "A cost effective color gamut mapping architecture for digital tv color reproduction enhancement," *IEEE Transactions on Consumer Electronics*, vol. 51, no. 1, pp. 168–174, 2005.

[8] Xilinx XUP Virtex II Pro Development System. [Online]. Available: http://www.xilinx.com/univ/xupv2p.html

[9] WILDCARD-4 from Annapolis Micro Systems, Inc. [Online]. Available: http://www.annapmicro.com/wc4.html

[10] C. Poynton, "A guided tour of color space," in *Proceedings of the SMPTE Advanced Television and Electronic Imaging Conference*, February 1995, pp. 167–180.

[11] P. Green and L. MacDonald, Eds., *Color Engineering, Achieving Device Independent Color*. John Wiley Sons Ltd, 2002.

[12] G. L. Vondran, Jr., "Apparatus for generating interpolator input data," U.S. Patent 5 717 507, Feb. 10, 1998.

[13] A. Albiol, L. Torres, and E. J. Delp, "An unsupervised color image segmentation algorithm for face detection applications," in *Proceedings. 2001 International Conference on Image Processing*, vol. 2, 2001, pp. 681–684 vol.2.

[14] P. Kuchi, P. Gabbur, P. S. Bhat, and S. David, "Human face detection and tracking using skin color modelling and connected component operators," *The IETE Journal of Research, Special issue on Visual Media Processing*, May 2002.

[15] B. Menser and M. Brunig, "Face detection and tracking for video coding applications," in *Conference Record of the Thirty-Fourth Asilomar Conference on Signals, Systems and Computers*, vol. 1, 2000, pp. 49–53.

[16] I. Andreadis, "A real-time color space converter for the measurement of appearance," *Pattern Recognition*, vol. 34, no. 6, pp. 1181–1187, June 2001.

[17] M. Bilal and S. Masud, "Efficient color space conversion using custom instruction in a risc processor," in *IEEE International Symposium on Circuits and Systems*, 2007, pp. 1109–1112.

[18] I. Andreadis, A. Gasteratos, and P. Tsalides, "A new asic for the measurement of appearance," in *Instrumentation and Measurement Technology Conference, 1996. IMTC-96. Conference Proceedings. 'Quality Measurements: The Indispensable Bridge between Theory and Reality'., IEEE*, vol. 1, 1996, pp. 545–548 vol.1.

[19] M. Nsour and H. S. Abdel-Aty-Zohdy, "An improved asic design and implementation for color space conversion applications," in *IEEE 39th Midwest symposium on Circuits and Systems*, vol. 2, 1996, pp. 609–612.

[20] CSC-PT core. [Online]. Available: http://www.alma-tech.com/products_index.php?item=/02_Image%20Processing&sid=0

[21] CSC Color Space Conversion Core. [Online]. Available: http://www.cast-inc.com/cores/csc/index.shtml

[22] Athena Color Space Converter. [Online]. Available: http://www.athena-group.com/pdf/AthenaCSCv1.pdf

[23] CSC Color Space Converter. [Online]. Available: http://www.xilinx.com/products/logicore/alliance/cast/cast_csc.pdf

[24] Altera Color Space Converter MegaCore. [Online]. Available: http://www.altera.com.cn/literature/ug/csc_ug.pdf

[25] Designing a Video Color Space Converter on the VCA-6160 Platform. [Online]. Available: http://www.triadsemi.com/page/TSA001

[26] System Generator for DSP User Guide. [Online]. Available: http://www.xilinx.com/support/sw_manuals/sysgen_user.pdf

[27] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," *IEE Proceedings - Computers and Digital Techniques*, vol. 153, no. 3, pp. 157–164, 2006.

[28] Early Access Partial Reconfiguration User Guide. [Online]. Available: http://www.xilinx.com/support/prealounge/protected/docs/ug208.pdf

[29] Partial Reconfiguration Design with PlanAhead. [Online]. Available: http://www.xilinx.com/support/prealounge/protected/docs/PR_User_Guide.pdf

[30] Partial Reconfiguration Early Access Software Tools. [Online]. Available: http://www.xilinx.com/support/prealounge/protected/index.htm

[31] Partial Reconfiguration Software Users Guide. [Online]. Available: http://www.xilinx.com/support/prealounge/protected/software/pa_pr_user_guide_81.pdf

[32] B. Blodget, P. James-Roxby, E. Keller, S. Mcmillan, and P. Sundararajan, "A self-reconfiguring platform," *Field-Programmable Logic and Applications*, pp. 565–574, 2003.

[33] R. J. Fong, S. J. Harper, and P. M. Athanas, "A versatile framework for FPGA field updates: an application of partial self-reconfiguration," in *Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping*, 9-11 June 2003, pp. 117–123.

[34] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine, "Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems," in *IEEE Symposium on Security and Privacy*, 20-23 May 2007, pp. 281–295.

[35] S. Singh, J. Hogg, and D. McAuley, "Expressing dynamic reconfiguration by partial evaluation," in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, Eds., Napa, CA, Apr. 1996, pp. 188–194.

[36] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton, "Design patterns for reconfigurable computing," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 13–23.

Appendix A

Generation of Test Vectors

Table A.1 shows the general structure of the test vector file. The size of the CLUT values and image data section varies based on the conversion required, the resolution of the CLUTs and the size of the input image.

Purpose	Reset	Enable	Addr	Read/Write	LUT Data	Handshaking	Data Flags	Pixel Data	Num of Clock Cycles
Reset	Active	Active	-	-	-	-	-	-	3 - 4
CSC Configuration	Inactive	Active	Config Reg	Write	-	-	-	-	6 - 8
Check Configuration*	Inactive	Active	Config Reg	Read	-	-	-	-	6 - 8
Load CLUT values	Inactive	Active	CLUT	Write	CLUT Data	-	-	-	5816 - 14162
Handshaking	-	-	-	-	-	Start CSC	-	-	1
Process Image	Inactive	Active	-	-	-	-	-	Image Data	9600 - 57600
Extra clock cycles (Latency)	Inactive	Active	-	-	-	-	-	-	24

* Optional

Table A.1: General structure of the test vector file.

Appendix B

Hardware Co-simulation

The following screen shots from Simulink show the hardware co-simulation model. Figure B.1 shows the model that uses the top level HDL module as a block. This model can be used for software simulation. Once the design is verified, a hardware co-simulation block can be generated. The hardware co-simulation block is used to program the FPGA to implement the CSC design. Figure B.2 shows the model with the hardware co-simulation block. More details about performing hardware co-simulation are available in [26].

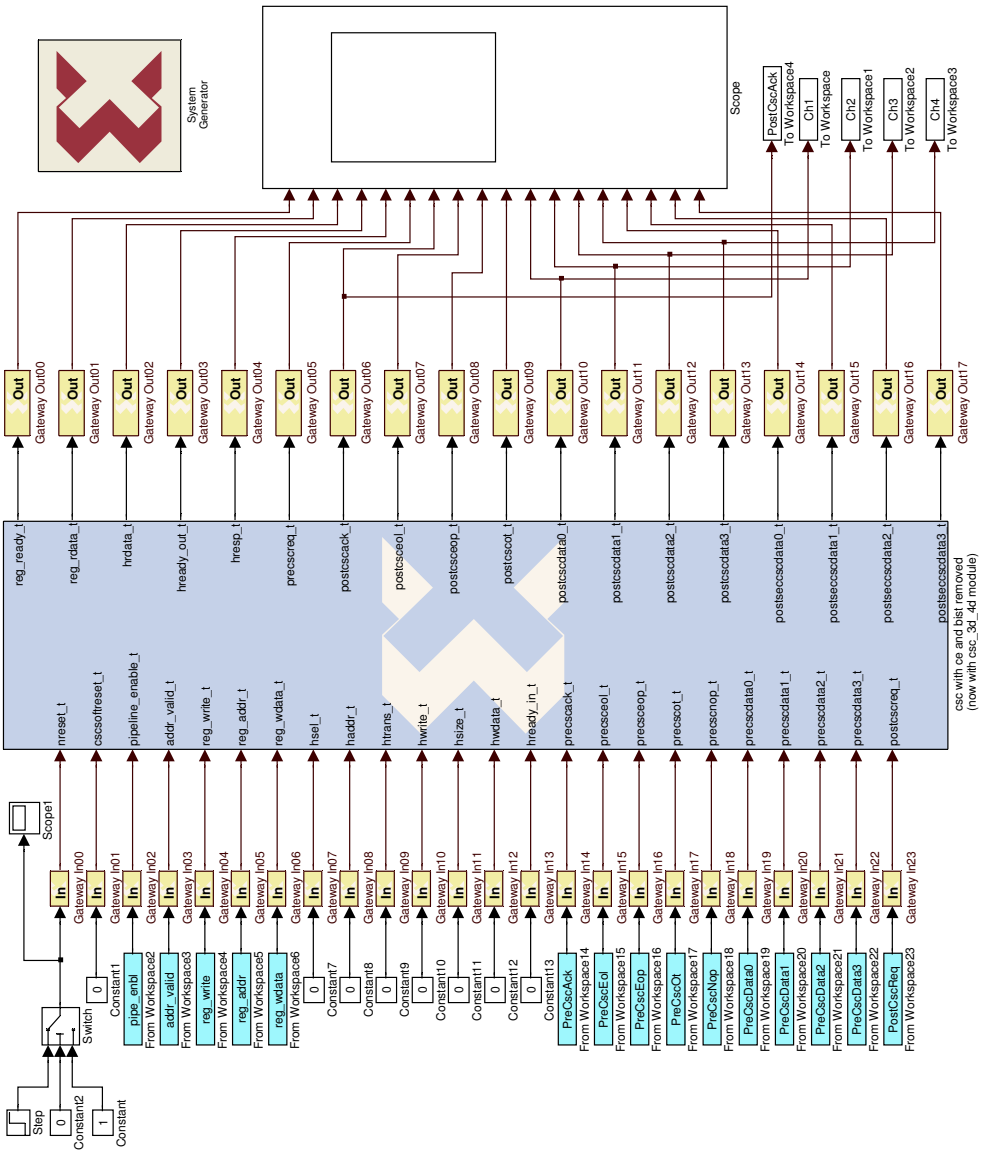

Figure B.1: System Generator project for simulation.

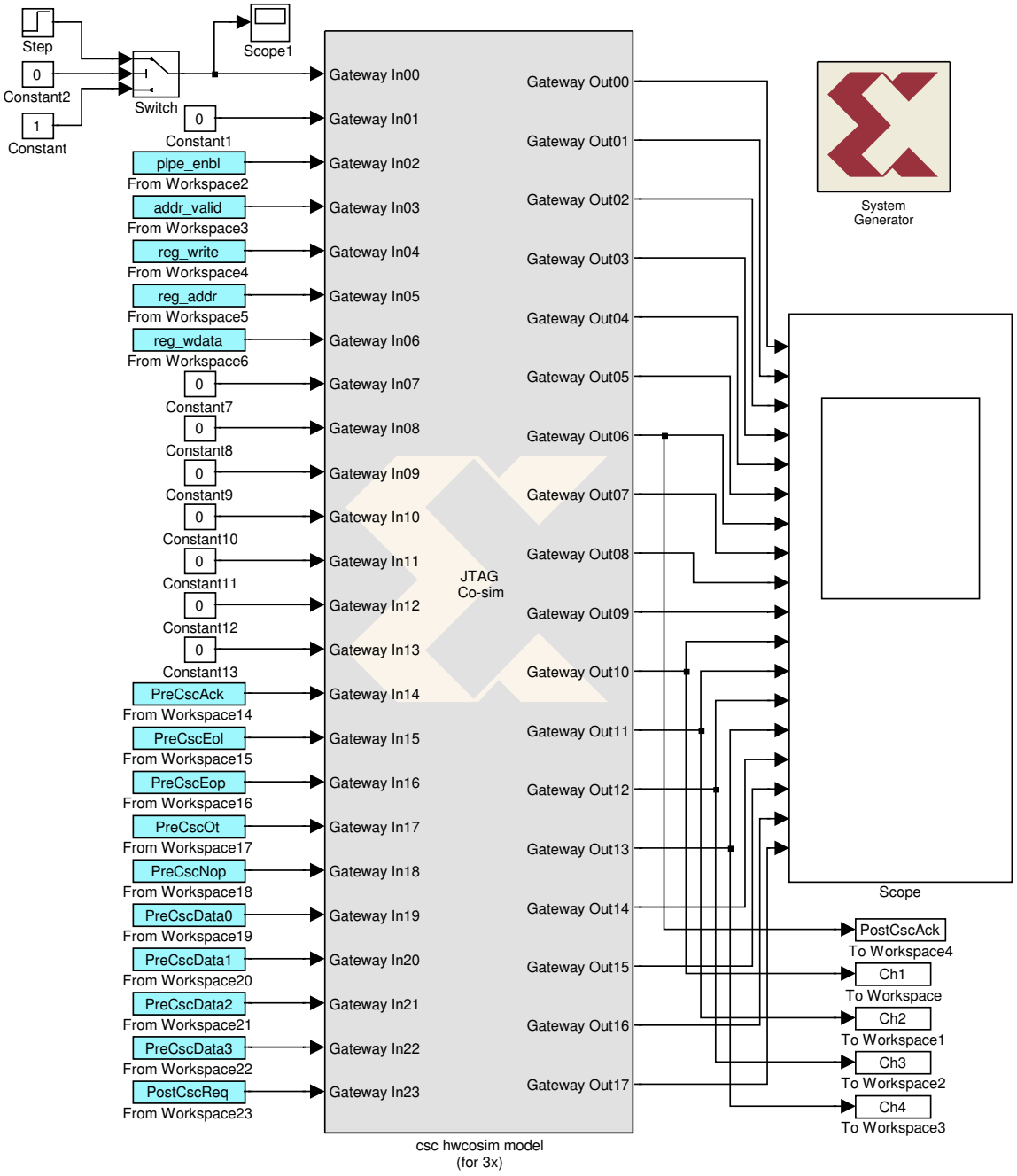

Figure B.2: System Generator project for hardware-in-the-loop testing.

Appendix C

Hardware and Software Used

Table C.1: List of hardware used for testing.

Development Board	FPGA
XUP Virtex-II Pro Development System [8]	XC2VP30-7FF896
Annapolis Micro Systems WILDCARD-4 [9]	XC4VSX35-10FF668

Table C.2: List of software used in implementation and testing.

Software	Version	Purpose
Xilinx ISE (for normal designs)	8.2.03i	Synthesis, Place and Route Software Simulation (stand alone and using System Generator)
Xilinx ISE (for PR designs)	8.2.01i_PR_07b	Synthesis of netlist for PRM and Synthesis of PR designs
Xilinx iMPACT	8.2.03i	Programming the FPGA
MATLAB	7.2.0.232 (R2006a)	Generating image information and extracting output from results
Simulink	6.4 (R2006a)	Create the hardware test bench
System Generator	8.2	Extends Simulink for use in FPGA hardware design.
Plan Ahead	8.2.10	Reserve PRR and Synthesis of PR designs
CSC Software Model (HP Executable)	5/23/2007 15:32:10	Convert source image into target color space
CLUT Data Converter (HP Executable)	0.810 Linux	Generate configuration register information, CLUT values and CSC data file.
Board support Packages for FPGA boards	–	Enables communication between MATLAB/Simulink and FPGA

Appendix D

MATLAB Source Code

This section presents the source code that is used in MATLAB/Simulink for hardware co-simulation.

1. Config File - csc_config.m

   This file is originally generated by Simulink when the top level HDL module is imported. It is then edited to include all the source files required for the compiling the design. It is used by the implementation tools to generate the block used for hardware co-simulation.

2. Preload File - im_preload.m

   This file reads the test vector input text file. The columns in the input file are separated and applied to the inputs of the HDL module.

3. Post Execute File - post_exec.m

   This file extracts the output image from the simulation results and compares it with the output of the software model of the CSC. This file also displays the result of the comparison in a message box. The statistics (max, min and mean) of the two output images, the difference image and the time required to process the image are displayed in the MATLAB command window.

```
function csc_config(this_block)

  % Revision History:
  %
  %   18-Sep-2007  (04:14 hours):
  %     Original code was machine generated by Xilinx's System Generator after parsing
  %     C:\Work\PR\s_code_PR_test\csc.v
  %

  this_block.setTopLevelLanguage('Verilog');

  this_block.setEntityName('csc');

  % System Generator has to assume that your entity  has a combinational feed through;
  %   if it  doesn't, then comment out the following line:
  this_block.tagAsCombinational;

  this_block.addSimulinkInport('nreset_t');
  this_block.addSimulinkInport('cscsoftreset_t');
  this_block.addSimulinkInport('pipeline_enable_t');
  this_block.addSimulinkInport('addr_valid_t');
  this_block.addSimulinkInport('reg_write_t');
  this_block.addSimulinkInport('reg_addr_t');
  this_block.addSimulinkInport('reg_wdata_t');
  this_block.addSimulinkInport('hsel_t');
  this_block.addSimulinkInport('haddr_t');
  this_block.addSimulinkInport('htrans_t');
  this_block.addSimulinkInport('hwrite_t');
  this_block.addSimulinkInport('hsize_t');
  this_block.addSimulinkInport('hwdata_t');
  this_block.addSimulinkInport('hready_in_t');
  this_block.addSimulinkInport('precscack_t');
  this_block.addSimulinkInport('precsceol_t');
```

```
this_block.addSimulinkInport('precsceop_t');
this_block.addSimulinkInport('precscot_t');
this_block.addSimulinkInport('precscnop_t');
this_block.addSimulinkInport('precscdata0_t');
this_block.addSimulinkInport('precscdata1_t');
this_block.addSimulinkInport('precscdata2_t');
this_block.addSimulinkInport('precscdata3_t');
this_block.addSimulinkInport('postcscreq_t');

this_block.addSimulinkOutport('reg_ready_t');
this_block.addSimulinkOutport('reg_rdata_t');
this_block.addSimulinkOutport('hrdata_t');
this_block.addSimulinkOutport('hready_out_t');
this_block.addSimulinkOutport('hresp_t');
this_block.addSimulinkOutport('precscreq_t');
this_block.addSimulinkOutport('postcscack_t');
this_block.addSimulinkOutport('postcsceol_t');
this_block.addSimulinkOutport('postcsceop_t');
this_block.addSimulinkOutport('postcscot_t');
this_block.addSimulinkOutport('postcscdata0_t');
this_block.addSimulinkOutport('postcscdata1_t');
this_block.addSimulinkOutport('postcscdata2_t');
this_block.addSimulinkOutport('postcscdata3_t');
this_block.addSimulinkOutport('postseccscdata0_t');
this_block.addSimulinkOutport('postseccscdata1_t');
this_block.addSimulinkOutport('postseccscdata2_t');
this_block.addSimulinkOutport('postseccscdata3_t');

reg_ready_t_port = this_block.port('reg_ready_t');
reg_ready_t_port.setType('UFix_1_0');
reg_ready_t_port.useHDLVector(false);
reg_rdata_t_port = this_block.port('reg_rdata_t');
reg_rdata_t_port.setType('UFix_32_0');
```

```
hrdata_t_port = this_block.port('hrdata_t');
hrdata_t_port.setType('UFix_32_0');
hready_out_t_port = this_block.port('hready_out_t');
hready_out_t_port.setType('UFix_1_0');
hready_out_t_port.useHDLVector(false);
hresp_t_port = this_block.port('hresp_t');
hresp_t_port.setType('UFix_1_0');
hresp_t_port.useHDLVector(false);
precscreq_t_port = this_block.port('precscreq_t');
precscreq_t_port.setType('UFix_1_0');
precscreq_t_port.useHDLVector(false);
postcscack_t_port = this_block.port('postcscack_t');
postcscack_t_port.setType('UFix_1_0');
postcscack_t_port.useHDLVector(false);
postcsceol_t_port = this_block.port('postcsceol_t');
postcsceol_t_port.setType('UFix_1_0');
postcsceol_t_port.useHDLVector(false);
postcsceop_t_port = this_block.port('postcsceop_t');
postcsceop_t_port.setType('UFix_1_0');
postcsceop_t_port.useHDLVector(false);
postcscot_t_port = this_block.port('postcscot_t');
postcscot_t_port.setType('UFix_2_0');
postcscdata0_t_port = this_block.port('postcscdata0_t');
postcscdata0_t_port.setType('UFix_12_0');
postcscdata1_t_port = this_block.port('postcscdata1_t');
postcscdata1_t_port.setType('UFix_12_0');
postcscdata2_t_port = this_block.port('postcscdata2_t');
postcscdata2_t_port.setType('UFix_12_0');
postcscdata3_t_port = this_block.port('postcscdata3_t');
postcscdata3_t_port.setType('UFix_12_0');
postseccscdata0_t_port = this_block.port('postseccscdata0_t');
postseccscdata0_t_port.setType('UFix_12_0');
postseccscdata1_t_port = this_block.port('postseccscdata1_t');
```

```
postseccscdata1_t_port.setType('UFix_12_0');
postseccscdata2_t_port = this_block.port('postseccscdata2_t');
postseccscdata2_t_port.setType('UFix_12_0');
postseccscdata3_t_port = this_block.port('postseccscdata3_t');
postseccscdata3_t_port.setType('UFix_12_0');

% ------------------------------
if (this_block.inputTypesKnown)
  % do input type checking, dynamic output type and generic setup in this code block.

  if (this_block.port('nreset_t').width ~= 1);
    this_block.setError('Input data type for port "nreset_t" must have width=1.');
  end

  this_block.port('nreset_t')
```

```
if (this_block.port('reg_write_t').width ~= 1);
  this_block.setError('Input data type for port "reg_write_t" must have width=1.');
end

this_block.port('reg_write_t').useHDLVector(false);

if (this_block.port('reg_addr_t').width ~= 18);
  this_block.setError('Input data type for port "reg_addr_t" must have width=18.');
end

if (this_block.port('reg_wdata_t').width ~= 32);
  this_block.setError('Input data type for port "reg_wdata_t" must have width=32.');
end

if (this_block.port('hsel_t').width ~= 1);
  this_block.setError('Input data type for port "hsel_t" must have width=1.');
end

this_block.port('hsel_t').useHDLVector(false);

if (this_block.port('haddr_t').width ~= 18);
  this_block.setError('Input data type for port "haddr_t" must have width=18.');
end

if (this_block.port('htrans_t').width ~= 2);
  this_block.setError('Input data type for port "htrans_t" must have width=2.');
end

if (this_block.port('hwrite_t').width ~= 1);
  this_block.setError('Input data type for port "hwrite_t" must have width=1.');
end
```

```
this_block.port('hwrite_t').useHDLVector(false);

if (this_block.port('hsize_t').width ~= 3);
  this_block.setError('Input data type for port "hsize_t" must have width=3.');
end

if (this_block.port('hwdata_t').width ~= 32);
  this_block.setError('Input data type for port "hwdata_t" must have width=32.');
end

if (this_block.port('hready_in_t').width ~= 1);
  this_block.setError('Input data type for port "hready_in_t" must have width=1.');
end

this_block.port('hready_in_t').useHDLVector(false);

if (this_block.port('precscack_t').width ~= 1);
  this_block.setError('Input data type for port "precscack_t" must have width=1.');
end

this_block.port('precscack_t').useHDLVector(false);

if (this_block.port('precsceol_t').width ~= 1);
  this_block.setError('Input data type for port "precsceol_t" must have width=1.');
end

this_block.port('precsceol_t').useHDLVector(false);

if (this_block.port('precsceop_t').width ~= 1);
  this_block.setError('Input data type for port "precsceop_t" must have width=1.');
end

this_block.port('precsceop_t').useHDLVector(false);
```

```
if (this_block.port('precscot_t').width ~= 2);
  this_block.setError('Input data type for port "precscot_t" must have width=2.');
end

if (this_block.port('precscnop_t').width ~= 1);
  this_block.setError('Input data type for port "precscnop_t" must have width=1.');
end

this_block.port('precscnop_t').useHDLVector(false);

if (this_block.port('precscdata0_t').width ~= 16);
  this_block.setError('Input data type for port "precscdata0_t" must have width=16.');
end

if (this_block.port('precscdata1_t').width ~= 16);
  this_block.setError('Input data type for port "precscdata1_t" must have width=16.');
end

if (this_block.port('precscdata2_t').width ~= 16);
  this_block.setError('Input data type for port "precscdata2_t" must have width=16.');
end

if (this_block.port('precscdata3_t').width ~= 16);
  this_block.setError('Input data type for port "precscdata3_t" must have width=16.');
end

if (this_block.port('postcscreq_t').width ~= 1);
  this_block.setError('Input data type for port "postcscreq_t" must have width=1.');
end

this_block.port('postcscreq_t').useHDLVector(false);
```

```
end  % if(inputTypesKnown)
% ------------------------------

% ------------------------------
 if (this_block.inputRatesKnown)
   setup_as_single_rate(this_block,'clk_t','ce_t')
 end  % if(inputRatesKnown)
% ------------------------------

% Add addtional source files as needed.
%  |-------------
%  | Add files in the order in which they should be compiled.
%  | If two files "a.vhd" and "b.vhd" contain the entities
%  | entity_a and entity_b, and entity_a contains a
%  | component of type entity_b, the correct sequence of
%  | addFile() calls would be:
%  |    this_block.addFile('b.vhd');
%  |    this_block.addFile('a.vhd');
%  |-------------

%    this_block.addFile('');
%    this_block.addFile('');

  this_block.addFile('../csc_defs.vh');
  this_block.addFile('../rbist_defs.vh');

  this_block.addFile('../sram_65x24.edn');
  this_block.addFile('../sram_65x24.xco');
  this_block.addFile('../sram_65x24.v');

  this_block.addFile('../sram_dp_bw_65x48_wrapper_coregen2.v');

  this_block.addFile('../csc_lut_wrappers_1d.v');
```

```
this_block.addFile('../csc_phase2_1d_channel.v');
this_block.addFile('../csc_phase2_1d.v');
this_block.addFile('../csc_phase1_1d.v');
this_block.addFile('../csc_lut1d.v');

this_block.addFile('../sram_729x15.edn');
this_block.addFile('../sram_729x15.xco');
this_block.addFile('../sram_729x15.v');

this_block.addFile('../sram_729x20.edn');
this_block.addFile('../sram_729x20.xco');
this_block.addFile('../sram_729x20.v');

this_block.addFile('../sram_256x20.edn');
this_block.addFile('../sram_256x20.xco');
this_block.addFile('../sram_256x20.v');

this_block.addFile('../sram_320x20.edn');
this_block.addFile('../sram_320x20.xco');
this_block.addFile('../sram_320x20.v');

this_block.addFile('../sram_400x20.edn');
this_block.addFile('../sram_400x20.xco');
this_block.addFile('../sram_400x20.v');

this_block.addFile('../sram_500x20.edn');
this_block.addFile('../sram_500x20.xco');
this_block.addFile('../sram_500x20.v');

this_block.addFile('../sram_512x15.edn');
this_block.addFile('../sram_512x15.xco');
this_block.addFile('../sram_512x15.v');
```

```
this_block.addFile('../sram_512x20.edn');
this_block.addFile('../sram_512x20.xco');
this_block.addFile('../sram_512x20.v');

this_block.addFile('../sram_576x15.edn');
this_block.addFile('../sram_576x15.xco');
this_block.addFile('../sram_576x15.v');

this_block.addFile('../sram_576x20.edn');
this_block.addFile('../sram_576x20.xco');
this_block.addFile('../sram_576x20.v');

this_block.addFile('../sram_625x20.edn');
this_block.addFile('../sram_625x20.xco');
this_block.addFile('../sram_625x20.v');

this_block.addFile('../sram_648x15.edn');
this_block.addFile('../sram_648x15.xco');
this_block.addFile('../sram_648x15.v');

this_block.addFile('../sram_648x20.edn');
this_block.addFile('../sram_648x20.xco');
this_block.addFile('../sram_648x20.v');

this_block.addFile('../sram_sp_bw_256x40_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_320x40_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_400x40_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_500x40_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_512x30_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_512x40_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_576x30_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_576x40_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_625x40_wrapper_coregen2.v');
```

```
this_block.addFile('../sram_sp_bw_648x30_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_648x40_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_729x30_wrapper_coregen2.v');
this_block.addFile('../sram_sp_bw_729x40_wrapper_coregen2.v');
this_block.addFile('../csc_lut_wrappers_3d.v');
this_block.addFile('../csc_phase1_3d.v');
this_block.addFile('../csc_phase2_3d.v');
this_block.addFile('../csc_phase3_3d_channel.v');
this_block.addFile('../csc_phase3_3d.v');
this_block.addFile('../csc_3d.v');
this_block.addFile('../csc_4d.v');
this_block.addFile('../csc_lut_wrappers_4d.v');
this_block.addFile('../csc_phase1_4d.v');
this_block.addFile('../csc_phase2_4d.v');
this_block.addFile('../csc_phase3_4d_channel.v');
this_block.addFile('../csc_phase3_4d.v');
this_block.addFile('../csc_lutchk_crc16_parallel_n.v');
this_block.addFile('../lutchk_ctrl.v');
this_block.addFile('../lutchk_regs.v');
this_block.addFile('../lutchk_timer.v');
this_block.addFile('../lutchk_top.v');
this_block.addFile('../ahblite_to_regbus.v');
this_block.addFile('../csc_reg.v');
this_block.addFile('../busmacro_l2r_standin.v');
this_block.addFile('../csc_3d_4d.v');
this_block.addFile('../csc_control.v');
this_block.addFile('../pipeline_handshake_enable.v');
this_block.addFile('../csc_isolation_stage.v');
this_block.addFile('../csc_auto_load.v');
this_block.addFile('../csc_pre_match.v');
this_block.addFile('../csc_k_plane_mag.v');

this_block.addFile('../csc_target.v');
```

```
    this_block.addFile('../csc_toner_limit.v');

    this_block.addFile('../clk_w_ce.v');
    this_block.addFile('../csc.v');

return;

% ------------------------------------------------------------

function setup_as_single_rate(block,clkname,cename)
  inputRates = block.inputRates;
  uniqueInputRates = unique(inputRates);
  if (length(uniqueInputRates)==1 & uniqueInputRates(1)==Inf)
    block.setError('The inputs to this block cannot all be constant.');
    return;
  end
  if (uniqueInputRates(end) == Inf)
     hasConstantInput = true;
     uniqueInputRates = uniqueInputRates(1:end-1);
  end
  if (length(uniqueInputRates) ~= 1)
    block.setError('The inputs to this block must run at a single rate.');
    return;
  end
  theInputRate = uniqueInputRates(1);
  for i = 1:block.numSimulinkOutports
     block.outport(i).setRate(theInputRate);
  end
  block.addClkCEPair(clkname,cename,theInputRate);
  return;
% ------------------------------------------------------------
```

```
%% This file is used to read the input text file and load the vectors
% into the matlab workspace, for use in simulink.
clear all;
close all;

%% Read the Configuration data file.
[InFileName,PathName] = uigetfile( {'*.txt;','Text Files (*.txt)';'*.*', 'All Files (*.*)'}, 'File Selector
- Configuration Data','..\..\Sim_Inputs\');

if isequal(InFileName,0)
    disp('File not selected. Restart execution.');
    msgbox('File not selected. Restart simulation.','HW Simulation','help');

else
    fid = fopen([PathName InFileName]);
    confdata = textscan(fid, '%u8 %u8 %u8 %5c %8c %u8 %u8 %u8 %u8 %u8 %4c %4c %4c %4c %u8');
    fclose(fid);

    %% Generate the input vectors from the configuration file.
    pipe_enbl   = confdata{1,1};
    addr_valid  = confdata{1,2};
    reg_write   = confdata{1,3};
    reg_addr    = hex2dec(confdata{1,4});
    reg_wdata   = hex2dec(confdata{1,5});
    PreCscAck   = confdata{1,6};
    PreCscEol   = confdata{1,7};
    PreCscEop   = confdata{1,8};
    PreCscOt    = confdata{1,9};
    PreCscNop   = confdata{1,10};
    PreCscData0 = hex2dec(confdata{1,11});
    PreCscData1 = hex2dec(confdata{1,12});
    PreCscData2 = hex2dec(confdata{1,13});
    PreCscData3 = hex2dec(confdata{1,14});
```

```
    PostCscReq  = confdata{1,15};

    % Find the size of the number of rows in the input vectors.
    % I can use any variable. pipe_enbl is just a random one I chose.
    [row_count,col_count] = size(pipe_enbl);

    %% Add the 'time value' column to each of the input vectors.
    % Also the data value need to be in double form

    %% The 'from workspace' block in simulink requires the array to
    % be in the following format -> var = [TimeValues DataValues]
    pipe_enbl   = [ double(0:row_count-1)' double(pipe_enbl)];
    addr_valid  = [ double(0:row_count-1)' double(addr_valid)];
    reg_write   = [ double(0:row_count-1)' double(reg_write)];
    reg_addr    = [ double(0:row_count-1)' double(reg_addr)];
    reg_wdata   = [ double(0:row_count-1)' double(reg_wdata)];
    PreCscAck   = [ double(0:row_count-1)' double(PreCscAck)];
    PreCscEol   = [ double(0:row_count-1)' double(PreCscEol)];
    PreCscEop   = [ double(0:row_count-1)' double(PreCscEop)];
    PreCscOt    = [ double(0:row_count-1)' double(PreCscOt)];
    PreCscNop   = [ double(0:row_count-1)' double(PreCscNop)];
    PreCscData0 = [ double(0:row_count-1)' double(PreCscData0)];
    PreCscData1 = [ double(0:row_count-1)' double(PreCscData1)];
    PreCscData2 = [ double(0:row_count-1)' double(PreCscData2)];
    PreCscData3 = [ double(0:row_count-1)' double(PreCscData3)];
    PostCscReq  = [ double(0:row_count-1)' double(PostCscReq)];

end;
```

```
%% The program is used to compare the outputs of the Xilinx or Hardware
% co-simulation and the csc application without losing any bits (bit width = 16)

%% Read File : Result of the CSC Application.
[OutFileName,PathName,FilterIndex] = uigetfile( {'*.tiff;*.tif','TIFF Files (*.tiff,*.tif)'}, 'Chose result
of CSC Application','..\..\csc_v3\');

if isequal(OutFileName,0)
    disp ('File not selected. Restart simulation.');
    msgbox('File not selected. Restart simulation.','HW Simulation','help');
else
    app_img = imread([PathName OutFileName]);

    [num_row,num_col,dim]=size(app_img);
    num_pixels = num_row * num_col;

    % We can use the PostCscAck signal to find out where the image pixel starts
    i=find(PostCscAck == 1);
    startrow = i(1) + 1;
% The extra 1 is because of the blank first pixel. This can be removed if the config file is modified

    iCh1 = Ch1(startrow:startrow + num_pixels -1,1);
    iCh2 = Ch2(startrow:startrow + num_pixels -1,1);
    iCh3 = Ch3(startrow:startrow + num_pixels -1,1);
    iCh4 = Ch4(startrow:startrow + num_pixels -1,1);

    sim_C = uint16(reshape(iCh1,num_row,num_col));
    sim_M = uint16(reshape(iCh2,num_row,num_col));
    sim_Y = uint16(reshape(iCh3,num_row,num_col));
    sim_K = uint16(reshape(iCh4,num_row,num_col));

    sim_img (:,:,1) = sim_C;
    sim_img (:,:,2) = sim_M;
```

```
    sim_img (:,:,3) = sim_Y;
    sim_img (:,:,4) = sim_K;

    %% Display the statistics of the individual images and the comparison
    %clc
    disp ('Result Image from Simulation');
    disp ('?  bit     C          M          Y          K');
    maxval = sprintf('Max %6.3f %6.3f %6.3f %6.3f',max(max(sim_img(:,:,1))),max(max(sim_img(:,:,2))), max↙
(max(sim_img(:,:,3))),max(max(sim_img(:,:,4))));
    disp(maxval);
    meanval = sprintf('Mean %6.3f %6.3f %6.3f %6.3f',mean(mean(sim_img(:,:,1))),mean(mean(sim_img(:,:,2))),↙
mean(mean(sim_img(:,:,3))),mean(mean(sim_img(:,:,4))));
    disp(meanval);
    minval = sprintf('Min %6.3f %6.3f %6.3f %6.3f\n',min(min(sim_img(:,:,1))),min(min(sim_img(:,:,2))), min↙
(min(sim_img(:,:,3))),min(min(sim_img(:,:,4))));
    disp(minval);

    disp (OutFileName);
    disp ('?  bit     C          M          Y          K');
    maxval = sprintf('Max %6.3f %6.3f %6.3f %6.3f',max(max(app_img(:,:,1))),max(max(app_img(:,:,2))), max↙
(max(app_img(:,:,3))),max(max(app_img(:,:,4))));
    disp(maxval);
    meanval = sprintf('Mean %6.3f %6.3f %6.3f %6.3f',mean(mean(app_img(:,:,1))),mean(mean(app_img(:,:,2))),↙
mean(mean(app_img(:,:,3))),mean(mean(app_img(:,:,4))));
    disp(meanval);
    minval = sprintf('Min %6.3f %6.3f %6.3f %6.3f\n',min(min(app_img(:,:,1))),min(min(app_img(:,:,2))), min↙
(min(app_img(:,:,3))),min(min(app_img(:,:,4))));
    disp(minval);

    diff = double(app_img) - double(sim_img);
    disp ('Difference in CMYK Values (Application - Simulation)');
    disp ('?  bit     C          M          Y          K');
    maxval = sprintf('Max %6.3f %6.3f %6.3f  %6.3f',max(max(diff(:,:,1))),max(max(diff(:,:,2))), max(max↙
```

```
(diff(:,:,3))),max(max(diff(:,:,4))));
    disp(maxval);
    meanval = sprintf('Mean %6.3f %6.3f %6.3f %6.3f',mean(mean(diff(:,:,1))),mean(mean(diff(:,:,2))), mean↙
(mean(diff(:,:,3))),mean(mean(diff(:,:,4))));
    disp(meanval);
    minval = sprintf('Min %6.3f %6.3f %6.3f %6.3f\n',min(min(diff(:,:,1))),min(min(diff(:,:,2))), min(min↙
(diff(:,:,3))),min(min(diff(:,:,4))));
    disp(minval);
    % disp('Done');

    %% compare the two files using isequal
    resmatch = isequal(app_img,sim_img);
    if resmatch
        match_image = sprintf('The images match');
    else
        match_image = sprintf('The images do not match');
    end
    % Display the results in the Matlab console and a message box
    disp_info=sprintf('Image1 : Result Image from Simulation,\n (Input = %s) \nImage2 : %s \n %s',↙
InFileName,OutFileName,match_image);
    disp(disp_info);
    msgbox(disp_info,'HW Simulation Results','help');
end;

% Display processing time after impreload in the InitFcn, at start of
% StartFcn and after end of simulation - StopFcn

times = sprintf('\nSimulation times \n Load = %2.4f seconds \n Configuration = %2.4f seconds \n Simulation =↙
%2.4f seconds',toc1,toc2-toc1,toc3-toc1);
disp (times);
```