2012

An Evaluation of the application of partial evaluation on color lookup table implementations

Jordan Hibbits

An Evaluation of the Application of Partial Evaluation on Color Lookup Table Implementations

by

Jordan A. Hibbits

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

Supervised by

Assistant Professor Dr. Dorin Patru
Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
January 2012

Approved by:

Dr. Dorin Patru, Assistant Professor
Thesis Advisor, Department of Electrical and Microelectronic Engineering

---

Dr. Eli Saber, Professor
Committee Member, Department of Electrical and Microelectronic Engineering

Dr. Sohail Dianat, Professor
Committee Member, Department of Electrical and Microelectronic Engineering

Dr. Sohail Dianat, Department Head
Department Head, Electrical and Microelectronic Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

An Evaluation of the Application of Partial Evaluation on Color Lookup Table Implementations

I, Jordan A. Hibbits, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Jordan A. Hibbits

Date

Abstract

An Evaluation of the Application of Partial Evaluation on Color Lookup Table Implementations

Jordan A. Hibbits

Supervising Professor: Dr. Dorin Patru

A number of SRAM-based *field-programmable gate arrays* (FPGAs) allow for *partial reconfiguration*, allowing a part of the device to be reconfigured while the rest of the device continues operating. *Partial evaluation*, or instance-specific design, allows a design to be optimized to a specific set of inputs. When combined with partial reconfiguration, the reconfigurable module can be reinstantiated based on the inputs to be processed and improve the performance of the design.

This thesis explores the effects, particularly the performance vs flexibility tradeoff, of using partial evaluation on the color look-up tables (CLUTs) of a color-space conversion module implemented on an FPGA. This thesis examines the impact of implementing the CLUTs as distributed RAMs, distributed ROMs, and block ROMs, as well as examining the effects of initializing block RAMs.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Color space conversion modules are used in a variety of commercial applications, including printers, scanners, and digital cameras. These devices typically implement color space conversion on *application-specific integrated circuits* (ASICs), gaining performance compared to software at the cost of flexibility. By utilizing *partial reconfiguration* (PR) that SRAM-based FPGAs allow, it is possible to retain the flexibility of software while achieving performance much nearer that of an ASIC implementation.

This thesis explores the effects of partial evaluation on an FPGA implementation of a color space conversion (CSC) engine using color lookup tables (CLUTs) as the element to test. The purpose of these tests was to improve the performance of a partially reconfigurable CSC engine. In the current PR CSC design, the *partially reconfigurable region* (PRR), or *reconfigurable partition* (RP) in the new PR design flow, is reconfigured, followed by the configuration of the CLUTs. In this thesis we evaluated

a number of partial evaluation variations to the CLUT design, including distributed RAMs, distributed ROMs, block RAMs with initial values, and block ROMs.

Chapter 2 reviews the background of this project. Chapter 3 describes the test cases studied in this thesis. Chapter 4 presents the results obtained. Chapter 5 presents closing remarks and future work.

Chapter 2

Background

Typically, FPGAs are programmed in their entirety, causing the system to wait while reconfiguration occurs. With *partial reconfiguration* (PR), a region of the device is reconfigured while the rest of the system can continue processing data. Manet *et al.* have reported on the advantages of dynamic partial reconfiguration in software defined radios and professional electronics [6]. Blodget *et al.* described a *self reconfigurable platform* (SRP), where specific circuits on the FPGA are used to reconfigure other regions of the device [1].

Partial evaluation, or *instance-specific design*, is an area of research in reconfigurable computing in which multiple FPGA implementations of a circuit are synthesized, each specialized to a specific instance of the problem. By specializing the circuit around the data to be processed, the hardware typically becomes simpler, smaller, and faster. For example, instead of implementing a filter with programmable coefficients, one can implement

multiple filters, each with a specific set of coefficients [4]. Previous uses of partial evaluation on FPGAs have used dynamic run-time synthesis of the circuit [7]. In the study of the CSC, the implementation of specific hardware for each set of CLUTs is an example of partial evaluation.

The application focused on in this thesis is a *color space conversion* (CSC) engine. A *color space* is a method of describing color in a standard way [3]. Several standardized color spaces exist, for example RGB, CMYK, CIE LAB, etc., as well as color spaces specific to individual devices such as printers or scanners. A typical application of a CSC engine is to convert from the color space of the device to a standardized color space, in a scanner or camera, or from a standardized color space to a device-specific color space, as in a printer. The conversion calculations are usually non-linear and complex in multiple dimensions [5].

The method of conversion of interest is using color look-up tables to perform arbitrary conversions. This is typically done in an *application-specific integrated circuit* (ASIC), however, prior work [2, 8] has taken an ASIC implementation of a CSC engine provided by Hewlett Packard (HP) and implemented it on a Virtex-II Pro XC2VP30 FPGA. The ASIC implementation has two major conversion units, one for converting a 3D input color space (*i.e.*, RGB) to another 3D or 4D color space, and one for converting a 4D input color space (*i.e.*, CMYK) to another 3D or 4D color space, as seen

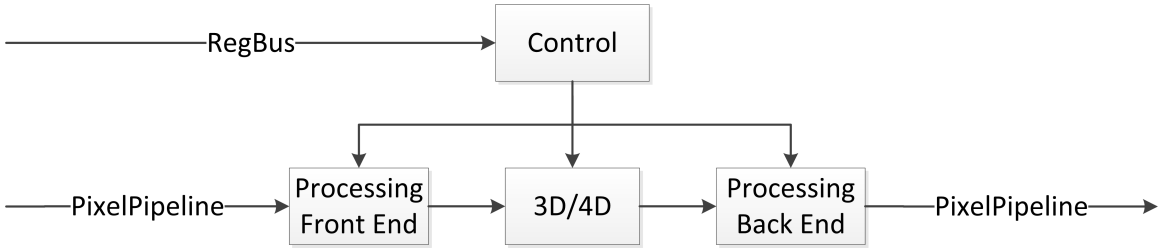

Figure 2.1: Core of the CSC engine.

in Figure 2.1. The ASIC also provides multiple write paths to the CLUTs. Due to resource limitations of the XC2VP30, and the availability of partial reconfiguration, the 3D and 4D modules were replaced with a single module which, through PR, can be configured for either 3D or 4D conversions. The FPGA implementation of [2] used block RAM resources to store the CLUTs.

SRAM-based FPGAs are the driving force of partial reconfiguration, with the Virtex family of Xilinx FPGAs the center of much research. This is due to the configuration architecture, which consists of three layers. The routing layer programs the interconnects related to the global resources, namely clocks, to ensure proper distribution in the FPGA. The user logic layer forms the hardware components of the FPGA design. The configuration layer configures the interconnects present to connect the user hardware components. In addition, the *internal configuration access port* (ICAP) present on these devices allows for user logic on the FPGA to access configuration memory on the device, enabling self-reconfiguration. The ICAP

has an 8 bit interface on the Virtex-II Pro, with a 32 bit interface available on Virtex 4, 5, and 6 devices, and a maximum frequency of 50 MHz to 100 MHz depending on the device.

This CSC application, due to its extensive use of CLUTs, is restricted by the available memory resources on the target FPGA. Table 2.1 shows the available memory resources of the various FPGAs used in the tests of this thesis. Of particular interest is the relative increase in available BRAM resources compared with the available distributed memory resources; between the Virtex 5 and Virtex 6, the available BRAM resources doubled, while the available LUT-based memory increased by more than a factor of 3.6.

Testing of the CSC was done across four Xilinx devices, one Spartan and three Virtex devices. In particular, the Spartan 3E XC3S1600E, Virtex-II Pro XC2VP30, Virtex 5 XC5VLX110T, and Virtex 6 XC6VLX240T were used. The Virtex-II Pro was originally chosen as the target device when the CSC was implemented on an FPGA, as the Xilinx University Program boards use the Virtex-II Pro. The Spartan 3E was chosen to investigate the implementation of the CSC engine on a low-cost FPGA. The Virtex 5 was explored as a potential upgrade route for newer testing. The Virtex 6 was chosen when finally upgrading, to migrate to the latest PR-enabled FPGA.

The Virtex-II Pro, now obsolete, was the first FPGA family to feature

the ICAP. This family of devices, in addition to the ICAP, featured several multi-gigabit transceivers, hardware multipliers and block memory, and larger devices in the family contained one or two IBM PowerPC 405 processor cores. ICAP is limited to 50 MHz on the Virtex-II Pro devices.

The Spartan 3E, on the other hand, was the first low-cost FPGA family to feature the ICAP module; however, functionality of the ICAP was not directly supported by the Xilinx design tools. The design tools supported by the Xilinx tools (specifically PlanAhead) were oriented around module-based partial reconfiguration. Due to the design of the Virtex family of FPGAs, the rest of the reconfigurable fabric-including unused bits in a PR frame-is capable of continuing to operate glitch-free during partial reconfiguration. However, when undergoing PR on the Spartan 3E, unused bits in a frame are temporarily reset; a method of handling these glitches must be added to any PR design. Both Virtex-II Pro and Spartan 3E frame sizes are one column of CLBs or block memory.

In addition to increasing the available resources, the Virtex 5 and 6 also featured enhancements to the ICAP module. Limited to 50 MHz and 8 bits in the Virtex-II Pro and Spartan 3E, the ICAP in Virtex 5 and 6 can be configured for data widths of 8, 16, or 32 bits. On Virtex 5 and 6 devices, ICAP is limited to 100 MHz.

Table 2.1: Device Summary

Family	BRAM (Kbits)			Distributed RAM (Kbits)			Multipliers/DSP Slices		
	Min	Tested	Max	Min	Tested	Max	Min	Tested	Max
Spartan 3E	72	648	648	15	231	231	4	36	36
Virtex II Pro	216	2448	7992	44	428	1378	12	136	444
Virtex 5	936	5328	18576	320	1120	2280	24	64	1056
Virtex 6	5616	14976	38304	1045	3650	8280	288	768	2016

Chapter 3

Design of Experiment

We have studied the effects of partial evaluation on the color look-up table values in the HP color space conversion modules on Xilinx FPGAs. Four implementations of storing the CLUT data are investigated in this thesis.

1. Initial values to (block) RAMs
2. Specifying memories as (block) ROMs - no write interface needs implementing
3. Distributed RAMs
4. Distributed ROMs

Our initial investigation into initial values to BRAMs is based on the idea that since BRAMs are being used already, initializing the cells should give us initial values without any additional overhead. With that premise, if there is no additional overhead from initializing BRAMs, converting to Block

ROMs would allow for the removal of hardware associated with writing to the CLUTs, reducing resource utilization and potentially increasing the clock rate.

3.1 Tested Implementation Variants

Since not every FPGA has an abundance of BRAMs, though as seen in Table 2.1 the amount of available BRAMs is increasing with newer generations of FPGAs, we also looked into using the LUT resources of an FPGA to implement the CLUTs, known in Xilinx tools as *distributed RAMs* and *distributed ROMs*. A potential application of distributed memories could allow for a combination of distributed and block memories based on the available resources of the specific FPGA.

3.1.1 Initialized Block RAMs

The simplest change tested was initializing the BRAMs to store initial CLUT data. This testing led to the creation of tools and scripts to aid in the implementation of the design. Without initial values, a single BRAM Xilinx CORE Generator module of each memory size was created to simplify the design. However, initialized BRAMs require individual BRAM modules to specify unique coefficient (COE) files. A tool (hex2coe) was created to

convert from the transactional based CLUT data containing 32 bit values, used to fill the CLUTs one location at a time at runtime, to the COE format containing 40 or 48 bit values. This tool reads in the CLUT hex data and uses bit shifting to first separate each channel then writes them, in hex, as a coefficient file. Scripts to generate the BRAM modules, as well as a makefile to create the entire design, were created as well.

3.1.2 Distributed RAMs

The largest obstacle encountered while designing the distributed RAM implementation was the resource limitation. Initial testing on the XC2VP30 led to the discovery that, even considering just the resource requirements of the individual 3D or 4D conversion module of the PRR, the XC2VP30 did not have sufficient resources for accurate testing of such a design. This led to an evaluation of other devices and device families for a comparison of the resources available on the FPGAs.

3.1.3 Distributed ROMs

To test distributed ROMs, an individual distributed ROM module needed to be created for each CLUT and each test. As distributed ROMs store data in the LUT resources of the FPGA as an optimized function. The required resources depend on the test case; a set of identity CLUTs simplify down to

wires, while complex CSC conversions use resources comparable to those used by distributed RAMs, as little if any simplification can be done. With this design the write interface to the CLUTs in the PRR was removed.

3.2 Block ROMs

Similarly, block ROMs also used specific block ROM instantiations for each CLUT. While initial testing of block ROMs was done on the Virtex-II Pro XC2VP30, the block ROM design was fully tested in simulation and hardware on the Virtex 6 XC6VLX240T. The block ROM design removes the entire write mechanism to CLUTs in the CSC, including the pre- and post-conversion modules (Figure 3.1). This allows for the removal of all hardware involved in writing to the CLUTs, in particular the Autoload module, responsible for loading the CLUTs from the pixel pipeline. While removing all write access in the design is not entirely practical, as the pre- and post-conversion CLUTs need to be writable, this implementation allows for the greatest reduction in hardware.

3.3 Test Bench Design

During the course of this project we have revised the test bench several times. The initial test bench from [2], implemented on two Virtex-II Pro

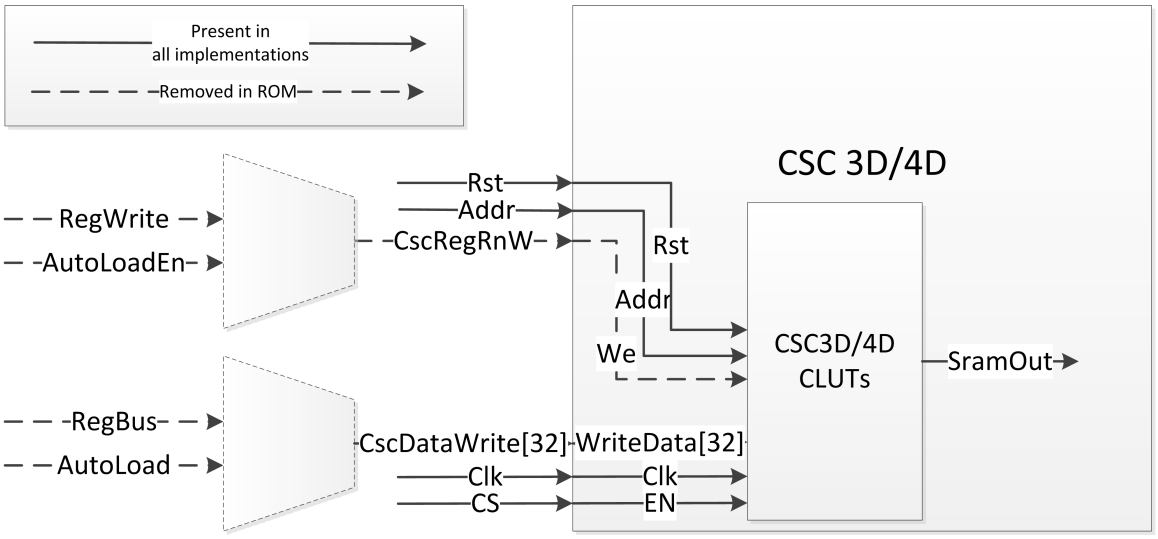

Figure 3.1: CLUT interface changes.

XC2VP30 Xilinx University Program development boards, proved limiting. The resources available on the XC2VP30 was insufficient for multiple tests. Additionally, the test bench takes on the order of 30 minutes to load test data, leading to evaluating alternative test setups. After initial results pointed to the resource limitations of the XC2VP30, a VHDL test bench was used to verify functionality of a non-PR implementation. Initial synthesis and VHDL testing for distributed ROM and distributed RAM implementations were performed, but not completed due to limitations in resources, on the Virtex-II Pro XC2VP30. The results on the Virtex-II Pro led to the testing of distributed ROM implementations of the PRR on the Virtex 5 XC5VLX110T and Spartan 3E XC3S1600E.

Initial testing of initialized block RAM and block ROM implementations were performed on the Virtex II Pro XC2VP30 and completed on the ML605 evaluation board, using a Virtex 6 XC6VLX240T. Initially, all block ROM tests on the Virtex 6 were completed in non-PR VHDL simulation. A PR-enabled version was implemented on the ML605 using a *built-in self test* (BIST) to verify functionality. Figure 3.2 shows the tool and data flow from configuration files, through design and building of the CSC, to testing the CSC design. Starting from configuration and lookup table data provided by HP, files containing the CLUT and register values for a conversion are created using an HP-provided tool. For testing versions which use initialized memories, the CLUT data is passed into hex2coe to obtain coefficient files for the memories. The steps here vary depending on the type of testing to be done. For hardware testing using partial reconfiguration, the partial bitstreams for the 3D and 4D modules are converted to test vectors using mrprdata [2]. For testing without initialized memories, the CLUT data is converted to the test vector format and added to the test vector. Source TIFF images are converted to text test vectors using a custom MATLAB script tiff2txt, making up the pixel data of the test vector. For initial testing on the XC2VP30, the test vectors are loaded onto a Compact Flash card to be processed. Later simulations use the same test vector format, without the partial bitstream data.

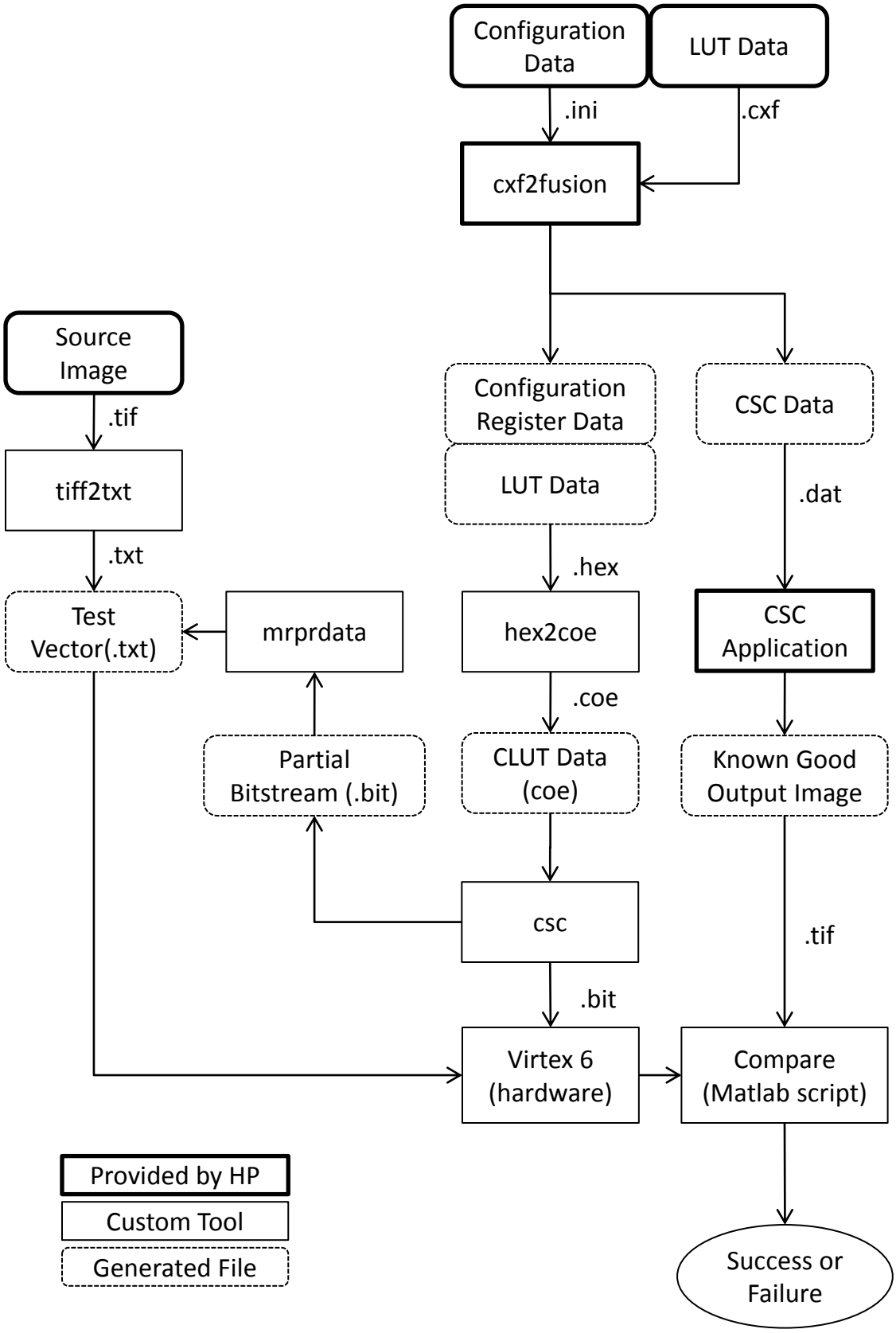

Figure 3.2: Tool flow for testing the CSC.

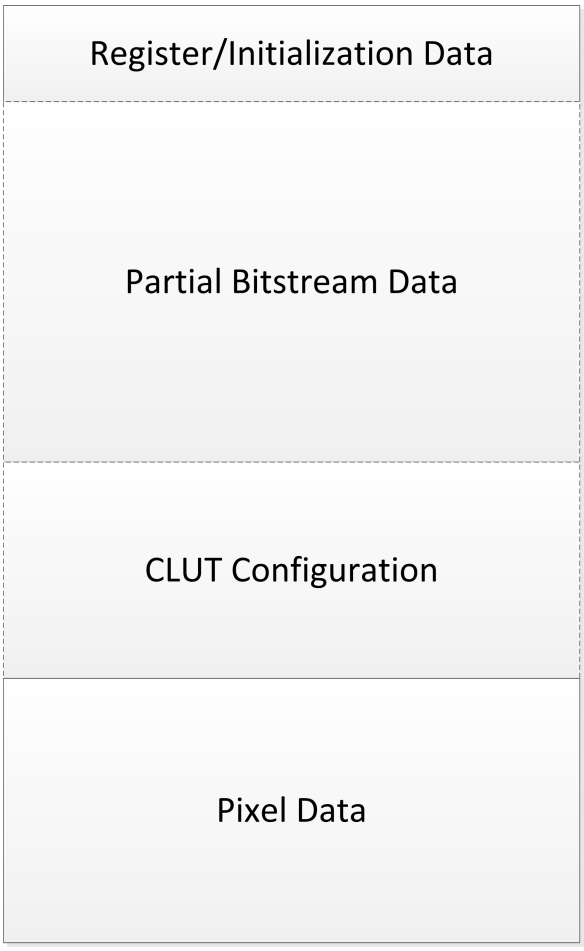

Figure 3.3: Sections of a test vector.

Chapter 4

Results

As seen in Table 2.1, there is a wide range of available resources, even within a given family of devices. In the driving example, the CSC, there is a fairly large usage of memory, used for storing CLUTs. Given that SRAM-based FPGAs store configuration data in SRAM memory cells, it's possible to use those same FPGA LUTs to serve as CLUTs in distributed memory, either RAM or ROM. Table 4.1 shows the initial resource requirements of the PR-enabled CSC on the XC2VP30. Of particular note, the partially reconfigurable region (PRR) uses 26.34% of the LUT resources in the block RAM implementation.

4.1 Distributed RAM

Implementing the distributed RAM design on the XC2VP30 led to a closer look at the memory requirements of the CSC, in Table 4.2. Focusing on

Table 4.1: CSC engine implementation results (XC2VP30).

Feature	PRR	Static Region and PRR	Available Resources
Slices	4,407	10,035	13,696
BRAMs	60	92	136
Slice Flip Flops	1,399	4,313	27,392
4 input LUTs	7,214	16,732	27,392
Max. clock rate	50MHz		

just the PRR, 383.8 Kbits are needed just for memory in the 3D module, or 89.7% of the available distributed memory on the XC2VP30. As seen in Table 4.3, the 3D PRR uses 18% of the available LUT resources, combining for over 100% of the device for the PR alone, preventing the design from synthesizing. While the 4D PRR requires fewer distributed RAM resources, the PRR size is determined by the largest amount of resources required between the modules that can occupy it.

Table 4.2: RAM resource requirements

Module	Size (Kb)
3D	383.8
4D	256.3
1D	48.75

4.2 Distributed ROM

Distributed ROM allows the synthesis tools to analyze the stored data for places to optimize in synthesis; this can lead to smaller designs for any collection of CLUTs that have a pattern in them. Also, by being a ROM, write hardware can be removed, allowing further hardware reductions. In a PR design, however, the distributed ROM in practicality will have limited hardware reduction due to patterns, as the PRR needs to be large enough to encompass any set of memories, CLUTs in the CSC. As Table 4.3 shows, distributed ROM implementations, or more likely balanced hybrid implementations, can be useful in devices such as the Spartan 3E, where there is a low amount of available BRAM and a surplus in LUT resources compared to those needed by logic. However, the trend in newer families is to increase the available BRAM resources, albeit at a lower rate than logic resources.

4.3 Initialized Block RAM

Due to the structure of the Virtex-II Pro, where configuration is done on a frame by frame basis, BRAMs have their own frames separate from slice configuration frames. As a result, adding initial values to the BRAMs caused the size of the full and partial bitstreams to grow from the initial data. However, as Table 4.4 shows, the size of the bitstream increases by

Table 4.3: Distributed ROM PRR synthesis results

Device	Method	Engine	LUT %	Slice %	BRAM	Synthesis Speed (MHz)
Virtex II Pro XC2VP30	BRAM	3D	18%	19%	44%	75
		4D	26%	27%	25%	75
	Dist ROM	3D	89%	97%	0%	66
		4D	87%	94%	0%	61
Virtex 5 XC5VLX110T	BRAM	3D	6%	6 %	20%	95
		4D	7%	7%	11%	78
	Dist ROM	3D	34%	34%	0%	89
		4D	32%	32%	0%	76
Spartan 3E XC3S1600E	BRAM	3D	16%	18%	176%	57
		4D	24%	26%	94%	55
	Dist ROM	3D	83%	90%	0%	48
		4D	81%	87%	0%	43

more than the BRAM size. This appears to be due to the additional BRAMs in the frames used by the PRR, though the discrepancy between the 3D and 4D changes in bitstream sizes is not explained.

Table 4.4: Initialized BRAM results

Engine	Partial Bitstream Size (Bytes)		Change in bitstream size (bytes)	CLUT Size (Bytes)
	No Initial Data	Initialized BRAMs		
3D	476116	628156	152040	49130
4D	473684	678828	205144	32805

4.4 Block ROM

The final implementation analyzed was using block ROMs in place of block RAMs. While the block ROM implementation suffers from the same noticeable increase in bitstream size as block RAMs, this implementation allows for the removal of write hardware from the design, potentially a significant improvement in both resource utilization and clock rate.

As Table 4.5 shows, there is a maximum reduction of 8.52% in the static logic resource utilization, with only a maximum of 2.24% reduction in the RP resources. The most significant change was the estimated clock rate, which increases by 11.35%. At the same time, the partial bitstream size increases by 13.75%. Table 4.6 shows, using the estimated clock rate and bitstream sizes, the approximate reconfiguration time of the RP, as well as the configuration time of the CLUTs for the BRAM implementation. The clock rate was estimated based on Xilinx ISE post-place and route static timing analysis for the non-PR implementations, with the lowest between 3D and 4D designs for each memory configuration being considered. Although partial reconfiguration might affect the actual clock frequency slightly, it should have a similar effect on both RAM and ROM implementations, and was ignored for this analysis. While the partial reconfiguration time of the

module increases by converting to the BROM implementation, during compact mode configuration of the 3D CLUTs the total configuration times have less than 0.1% variance, while under normal mode the BROM implementation proves to be faster. The BROM implementation, however, forces the entire RP reconfiguration even if just the CLUTs change, a less flexible system.

Table 4.5: Virtex 6 Block ROM Resource Utilization

Engine	Resource	RP Usage			Static Usage		
		RAM	ROM	% Delta	RAM	ROM	% Delta
3D	Registers	1010	1026	1.58%	3618	3621	0.08%
	LUTs	3730	3679	-1.37%	7787	7630	-2.02%
	Slices	1025	1002	-2.24%	2968	2830	-4.65%
	LUT-FF Pairs	3767	3727	-1.06%	8598	8422	-2.05%
	RAMB36s	28	28	0%	32	32	0%
	RAMB18s	4	4	0%	4	4	0%
	DSP48s	16	16	0%	0	0	0%
	Bitsteam Size (KB)	531	604	13.75%	9017	9017	0%
4D	Registers	1383	1383	0%	3618	3621	0.08%
	LUTs	4617	4564	-1.15%	7787	7630	-2.02%
	Slices	1233	1243	0.81%	2968	2715	-8.52%
	LUT-FF Pairs	4718	4646	-1.53%	8607	8351	-2.97%
	RAMB36s	2	2	0%	32	32	0%
	RAMB18s	30	30	0%	0	0	0%
	DSP48s	24	24	0%	0	0	0%
	Bitsteam Size (KB)	531	604	13.75%	9017	9017	0%
Estimated max. clock rate (MHz)					61.8	68.8	11.35%

Table 4.6: Estimated partial reconfiguration timings (XC6VLX240)

Implementation	Bitstream Size (Kbytes)	Speed (MHz)	Config Time (ms)[a]
BRAM	531	61.8	8.80
CLUT Data (3D)	48	61.8	0.20 (0.40)
CLUT Data (4D)	32	61.8	0.13 (0.27)
BROM	604	68.8	8.99

[a]CLUT Data compact (normal) mode configuration time

Chapter 5

Conclusions

We have presented the evaluation of four variations of storage as they apply to a partially reconfigurable FPGA implementation of a color space conversion module. With the initial design, when the CSC PRR is reconfigured, the CLUT RAMs are loaded with the appropriate CLUT data. The four variations of CLUT storage evaluated make trade-offs between resource utilization and performance. For the size of storage needed for the CLUTs in this design, distributed memories used too many LUT resources; however, such an implementation could be used for smaller memories, or in a mixed design with block and distributed memories, if the available block memory resources of the desired device is too limiting. Block ROMs can provide the greatest reduction in resources and increase in performance; in this design, there are multiple write paths to the CLUTs, with the critical path of the original design the write enable signal to one of the block RAM modules.

Additional application of partial evaluation could be applied to other

modules of the CSC; however, with the critical path being on a write enable signal to the CLUTs, the only likely gain would be in resource utilization. Future work to improve the performance of the CSC can look at overlapping reconfiguration of one region of the FPGA while processing data, so that the reconfiguration time overhead can be mitigated. Note that such a design would require a reevaluation of the design presented in [2], as that design utilized the pixel pipeline to feed bitstream data.

Bibliography

[1] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. A self-reconfiguring platform. In Cheung and George Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, chapter 55, pages 565–574. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2003.

[2] J. Galindo, E. Peskin, B. Larson, and G. Roylance. Leveraging Firmware in Multichip Systems to Maximize FPGA Resources: An Application of Self-Partial Reconfiguration. *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, pages 139–144, December 2008.

[3] P. Green and L.W. MacDonald. *Colour engineering: achieving device independent colour*. Wiley SID series in display technology. Wiley, 2002.

[4] Scott Hauck and André DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, November 2007.

[5] James M. Kasson, Sigfredo I. Nin, Wil Plouffe, and James Lee Hafner. Performing color space conversions with three-dimensional linear interpolation. *J. Electronic Imaging*, pages 226–250, 1995.

[6] Philippe Manet, Daniel Maufroid, Leonardo Tosi, Gregory Gailliard, Olivier Mulertt, Marco Di Ciano, Jean-Didier Legat, Denis Aulagnier, Christian Gamrat, Raffaele Liberati, Vincenzo La Barba, Pol Cuvelier, Bertrand Rousseau, and Paul Gelineau. An Evaluation of Dynamic Partial Reconfiguration for Signal and Image Processing in Professional Electronics Applications. *EURASIP Journal on Embedded Systems*, 2008, November 2008.

[7] N. McKay, T. Melham, and Kong W. Susanto. Dynamic specialisation of XC6200 FPGAs by partial evaluation. pages 308–309, April 1998.

[8] Sreenivas Patil. Reconfigurable hardware for color space conversion. Master's thesis, Rochester Institute of Technology, May 2008.

Appendix A

Makefile

```
# Makefile for CSC PR
#  Initialize environment
#  Generate ngc files for each module and copy to desired location
#  Convert bit streams to test files
#  Test files: CSC/CXF2FUSION/Merging regions of test file with
#   header, PRM bit stream, CLUT data, image

RMDIR = $(RM) -R

STATIC_DIR = synth/Static
MOD3D_DIR = synth/Mod_3d
MOD4D_DIR = synth/Mod_4d
TOP_DIR = synth/Top
STATIC_PROJECTS = csc_auto_load/csc_auto_load.ise csc_control/csc_control.ise \
        csc_isolation_stage/csc_isolation_stage_69.ise \
        csc_isolation_stage/csc_isolation_stage_100.ise \
        csc_k_plane_mag/csc_k_plane_mag.ise csc_lut1d/csc_lut1d_pre.ise \
        csc_lut1d/csc_lut1d_post.ise csc_pre_match/csc_pre_match.ise \
        csc_reg/csc_reg.ise csc_wrapper/csc_wrapper.ise icap_eai/icap_eai.ise \
        lutchk_top/lutchk_top.ise \
        pipeline_handshake_enable/pipeline_handshake_enable.ise \
        testrig2dutinterface/testrig2dutinterface.ise
STATIC_HDL = $(notdir $(STATIC_PROJECTS:.ise=.v))
SOURCES_1D_SRAM := $(wildcard $(STATIC_DIR)/sram*wrapper_coregen2.v) \
        $(patsubst %.xco,%.ngc,$(wildcard $(MOD4D_DIR)/sram_*.xco))
SOURCES_STATIC = $(addprefix $(STATIC_DIR)/, $(STATIC_PROJECTS) $(STATIC_HDL) \
        $(STATIC_PROJECTS:.ise=.xst))
TARGETS_STATIC = $(addprefix $(STATIC_DIR)/, $(STATIC_PROJECTS:.ise=.ngc))
PR_3D_PROJECT = $(MOD3D_DIR)/Mod_3d.ise
PR_4D_PROJECT = $(MOD4D_DIR)/Mod_4d.ise
PR_PROJECTS = $(PR_3D_PROJECT) $(PR_4D_PROJECT)
SOURCES_PR_3D_SRAM := $(wildcard $(MOD3D_DIR)/sram_0*.xco) $(wildcard
    $(MOD3D_DIR)/sram_1*.xco)
SOURCES_PR_4D_SRAM := $(wildcard $(MOD4D_DIR)/sram_*.xco)
SOURCES_PR_3D_SRAM_COE := $(wildcard $(MOD3D_DIR)/*.coe)
SOURCES_PR_4D_SRAM_COE := $(wildcard $(MOD4D_DIR)/*.coe)
TARGETS_PR_3D_SRAM := $(SOURCES_PR_3D_SRAM:.xco=.edn)
TARGETS_PR_4D_SRAM := $(SOURCES_PR_4D_SRAM:.xco=.edn)
SOURCES_PR_3D = $(addprefix $(MOD3D_DIR)/, csc_3d_4d.v csc_3d.v csc_phase1_3d.v \
        csc_lut_wrappers_3d.v csc_phase2_3d.v csc_phase3_3d.v csc_phase3_3d_channel.v \
        csc_defs.vh rbist_defs.vh csc_3d_4d.lso) \
        $(wildcard $(MOD3D_DIR)/sram*wrapper_coregen2.v)
SOURCES_PR_4D = $(addprefix $(MOD4D_DIR)/, csc_3d_4d.v csc_4d.v csc_phase1_4d.v \
```

```
        csc_lut_wrappers_4d.v csc_phase2_4d.v csc_phase3_4d.v csc_phase3_4d_channel.v \
        csc_defs.vh rbist_defs.vh csc_3d_4d.lso) \
        $(wildcard $(MOD4D_DIR)/sram*wrapper_coregen2.v)
TARGETS_PR_3D = $(MOD3D_DIR)/csc_3d_4d.ngc
TARGETS_PR_4D = $(MOD4D_DIR)/csc_3d_4d.ngc
TARGETS_PR = $(TARGETS_PR_3D) $(TARGETS_PR_4D)
TOP_PROJECT = $(TOP_DIR)/Top.ise
SOURCES_TOP = $(TOP_DIR)/csc.v $(TOP_DIR)/csc.lso
TARGETS_TOP = $(TOP_DIR)/csc.ngc
TARGETS = $(TARGETS_PR) $(TARGETS_STATIC)
COPIES_3D =     $(addprefix $(NETLIST_DIR)/Mod_3d/, $(notdir $(TARGETS_PR_3D))) \
        $(addprefix $(NETLIST_DIR)/Mod_3d/, $(notdir $(TARGETS_PR_3D_SRAM)))
COPIES_4D = $(addprefix $(NETLIST_DIR)/Mod_4d/, $(notdir $(TARGETS_PR_4D))) \
        $(addprefix $(NETLIST_DIR)/Mod_4d/, $(notdir $(TARGETS_PR_4D_SRAM)))
COPIES_STATIC = $(addprefix $(NETLIST_DIR)/Static/, $(notdir $(TARGETS_STATIC)))
COPIES_TOP = $(addprefix $(NETLIST_DIR)/Top/, $(notdir $(TARGETS_TOP)) \
        busmacro_xc2vp_l2r_async_narrow.nmc busmacro_xc2vp_r2l_async_narrow.nmc)
INTERMEDIATES = $(wildcard synth/*/sram*ch*readme.txt) \
        $(wildcard synth/*/sram*ch*flist.txt) \ $(wildcard synth/*/sram*ch*.asy) \
        $(wildcard synth/*/sram*ch*.sym) $(wildcard synth/*/sram*ch*.v*) \
        $(wildcard synth/*/*.edn) $(wildcard synth/*/*log) \
        $(wildcard synth/*/*.ngo) $(wildcard synth/*/*.stx) \
        $(wildcard synth/*/*.syr) $(wildcard synth/*/*.ngr) \
        $(wildcard synth/*/*.html) $(wildcard synth/Static/*/*.html)
CVSIGNORED := $(find -name .cvsignore)

TEMPDIRS = $(wildcard $(STATIC_DIR)/*/_xmsgs) $(wildcard $(STATIC_DIR)/*/xst) \
        $(wildcard synth/*/_cg) $(wildcard synth/*/_xmsgs) $(wildcard synth/*/templates) \
        $(wildcard synth/*/tmp) $(TOP_DIR)/xst $(TOP_DIR)/xst
TOOLSDIR = tools
MRPRDATA = $(TOOLSDIR)/mrprdata/mrprdata
HEX2COE = $(TOOLSDIR)/hex2coe/hex2coe
PRDATA2PG = $(TOOLSDIR)/prdata2pg.pl
TOOLS = $(MRPRDATA) $(HEX2COE)
SCRIPTSDIR = scripts
SCRIPT_SETPR = $(SCRIPTSDIR)\\set_to_xilinx_pr.bat
EXPORT_3D_DIR = PA_Projects/csc_pr_v2p_3d_export
EXPORT_4D_DIR = PA_Projects/csc_pr_v2p_4d_export
NETLIST_DIR = netlists

all: tools coes synth netlists planahead hexfiles prdata testfiles
tools: $(TOOLS)
coes: $(HEX2COE) mkcoe
        ./mkcoe
synth: static pr
static: $(TARGETS_STATIC)
pr: $(TARGETS_PR)

# xst places files in xst/projnav.tmp and won't run if it doesn't exist
# xst: -ise project file (ise/xise depending on version) for gui mode
#      -intstyle: output message format
#      -ifn: input file name (xst)
#      -ofn: output (log) file name

# 3D module
$(TARGETS_PR_3D): $(PR_3D_PROJECT) $(SOURCES_PR_3D) coregen3d
        mkdir -p $(@D)/xst/projnav.tmp
        cd $(@D) && xst -ise $(<F) -intstyle xflow -ifn csc_3d_4d.xst -ofn csc_3d_4d.syr
```

```
# 4D module
$(TARGETS_PR_4D): $(PR_4D_PROJECT) $(SOURCES_PR_4D) coregen4d
	mkdir -p $(@D)/xst/projnav.tmp
	cd $(@D) && xst -ise $(<F) -intstyle xflow -ifn csc_3d_4d.xst -ofn csc_3d_4d.syr

# Top module
$(TARGETS_TOP): $(TOP_PROJECT) $(SOURCES_TOP)
	mkdir -p $(@D)/xst/projnav.tmp
	cd $(@D) && xst -ise $(<F) -intstyle xflow -ifn csc.xst -ofn csc.syr

coregen3d: $(TARGETS_PR_3D_SRAM) $(SOURCES_PR_3D_SRAM_COE)
coregen4d: $(TARGETS_PR_4D_SRAM) $(SOURCES_PR_4D_SRAM_COE)

# srams
%.edn: %.xco
	# backup SRAM xco's since coregen rewrites them with same data
	# after edn is created; helps preserve functionality of make.
	# Run coregen in batch (command line) mode
	# Restore after edn generation, preserving the original modified date
	cd $(@D) && cp $(<F) $(<F).bak
	cd $(@D) && coregen -b $(<F)
	cd $(@D) && mv $(<F).bak $(<F)

# static netlists
%.ngc: %.ise %.lso %.xst
	mkdir -p $(@D)/xst/projnav.tmp
	cd $(@D) && xst -ise $(<F) -intstyle xflow -ifn $(*F).xst -ofn $(*F).syr

# lso file defines how to search libraries, if not present then
# create with DEFAULT_SEARCH_ORDER keyword
%.lso:
	echo DEFAULT_SEARCH_ORDER > $@

# copies of netlists in netlists/ directory
# this is where the planahead project checks for netlists
netlists: $(COPIES_3D) $(COPIES_4D) $(COPIES_STATIC) $(COPIES_TOP)

$(NETLIST_DIR)/Static/%.ngc : $(STATIC_DIR)/%/%.ngc
	cp $< $@

$(NETLIST_DIR)/Mod_3d/% : $(MOD3D_DIR)/%
	cp $< $@

$(NETLIST_DIR)/Mod_4d/% : $(MOD4D_DIR)/%
	cp $< $@

$(NETLIST_DIR)/Static/csc_isolation_stage/% : $(STATIC_DIR)/csc_isolation_stage/%
	cp $< $@

$(NETLIST_DIR)/Static/csc_lut1d/% : $(STATIC_DIR)/csc_isolation_stage/%
	cp $< $@

$(NETLIST_DIR)/Top/%.ngc : $(TOP_DIR)/%.ngc
	cp $< $@

planahead: build3d build4d
```

```
# Generates the bitstreams for 3d module (full and partial)
build3d: $(EXPORT_3D_DIR)/merge/static_full.ncd
    $(EXPORT_3D_DIR)/merge/ReconfigModules_CV_routed_partial.ncd

$(EXPORT_3D_DIR)/merge/static_full.ncd
    $(EXPORT_3D_DIR)/merge/ReconfigModules_CV_routed.ncd \
$(EXPORT_3D_DIR)/merge/reconfigmodules_cv_routed_partial.bit: \
$(COPIES_3D) $(COPIES_STATIC) $(COPIES_TOP)
        if [ -e $(EXPORT_3D_DIR) ]; then $(RM) -R $(EXPORT_3D_DIR)_old; \
                mv $(EXPORT_3D_DIR) $(EXPORT_3D_DIR)_old; fi
        $(RM) PA_projects/csc_pr_v2p_3d/csc_pr_v2p_3d.data/netlist/*.ngc
        if [ ! -e PA_projects/csc_pr_v2p_3d/csc_pr_v2p_3d.data/netlist/csc.ngc ]; then \
                mkdir -p PA_projects/csc_pr_v2p_3d/csc_pr_v2p_3d.data/netlist/; \
                cp PA_Projects/csc_pr_v2p_3d/csc_pr_v2p_3d.data/floorplan_1/csc.edf \
                PA_projects/csc_pr_v2p_3d/csc_pr_v2p_3d.data/netlist/;\
                fi
        mkdir $(EXPORT_3D_DIR)
        cmd /c "$(SCRIPT_SETPR) && planAhead -source $(SCRIPTSDIR)/csc_3d_pr_PA.tcl"
        cmd /c "$(SCRIPT_SETPR) && cd $(EXPORT_3D_DIR) && initModular.bat"
        cmd /c "$(SCRIPT_SETPR) && cd $(EXPORT_3D_DIR)/static && staticLogicImpl.bat"
        cmd /c "$(SCRIPT_SETPR) && cd $(EXPORT_3D_DIR) && processPblocks.bat"
        cmd /c "$(SCRIPT_SETPR) && cd $(EXPORT_3D_DIR)/merge && assemblePCfg.bat"

# Generates the bitstreams for 4d module (full and partial)
build4d: $(EXPORT_4D_DIR)/merge/static_full.ncd
    $(EXPORT_4D_DIR)/merge/ReconfigModules_CV_routed_partial.ncd

# Can't execute the first stages of the PA flow or you'll screw up the static region
# So copy the 3D directory, then copy the 4D netlists into the new directory
# Then rerun the last 2 steps with the 4D modules
$(EXPORT_4D_DIR)/merge/static_full.ncd \
$(EXPORT_4D_DIR)/merge/reconfigmodules_cv_routed_partial.ncd \
$(EXPORT_4D_DIR)/merge/reconfigmodules_cv_routed_partial.bit: \
$(COPIES_4D) $(COPIES_STATIC) $(COPIES_TOP)
        if [ -e $(EXPORT_4D_DIR) ]; then $(RM) -R $(EXPORT_4D_DIR)_old; \
                mv $(EXPORT_4D_DIR) $(EXPORT_4D_DIR)_old; fi
        cp -R $(EXPORT_3D_DIR)/ $(EXPORT_4D_DIR)/
        cp `ls -d $(NETLIST_DIR)/Mod_4d/* | grep -v CVS`
            $(EXPORT_4D_DIR)/ReconfigModules_CV/
        cmd /c "$(SCRIPT_SETPR) && cd $(EXPORT_4D_DIR) && processPblocks.bat"
        cmd /c "$(SCRIPT_SETPR) && cd $(EXPORT_4D_DIR)/merge && assemblePCfg.bat"

hexfiles: $(EXPORT_3D_DIR)/merge/csc_3d.hex $(EXPORT_4D_DIR)/merge/csc_4d.hex

$(EXPORT_3D_DIR)/merge/csc_3d.hex :
    $(EXPORT_3D_DIR)/merge/reconfigmodules_cv_routed_partial.bit
        cd $(EXPORT_3D_DIR)/merge/ && promgen -w -p hex -u 0
            reconfigmodules_cv_routed_partial.bit -o csc_3d.hex

$(EXPORT_4D_DIR)/merge/csc_4d.hex :
    $(EXPORT_4D_DIR)/merge/reconfigmodules_cv_routed_partial.bit
        cd $(EXPORT_4D_DIR)/merge/ && promgen -w -p hex -u 0
            reconfigmodules_cv_routed_partial.bit -o csc_4d.hex

# Convert partial bitstreams into test vectors using MRPRDATA
prdata: $(EXPORT_3D_DIR)/merge/csc3dpr.txt $(EXPORT_4D_DIR)/merge/csc4dpr.txt

$(EXPORT_3D_DIR)/merge/csc3dpr.txt : $(EXPORT_3D_DIR)/merge/csc_3d.hex
```

```
        $(MRPRDATA) $(EXPORT_3D_DIR)/merge/csc_3d.hex $(EXPORT_3D_DIR)/merge/csc3dpr.txt

$(EXPORT_4D_DIR)/merge/csc4dpr.txt : $(EXPORT_4D_DIR)/merge/csc_4d.hex
        $(MRPRDATA) $(EXPORT_4D_DIR)/merge/csc_4d.hex $(EXPORT_4D_DIR)/merge/csc4dpr.txt

testfiles: prplus3d.txt prplus4d.txt

# Combine premade header and image with partial bitstream vectors to make complete test
    vector
prplus3d.txt: testrig/testvectors/header3d.txt $(EXPORT_3D_DIR)/merge/csc3dpr.txt \
testrig/testvectors/clut3d_pre1d_post1d.txt testrig/testvectors/img3d.txt
        @echo $?
        cat $^ > $@
        $(PRDATA2PG) prplus3d.txt pg3d.txt

# Combine premade header and image with partial bitstream vectors to make complete test
    vector
prplus4d.txt: testrig/testvectors/header4d.txt $(EXPORT_4D_DIR)/merge/csc4dpr.txt \
testrig/testvectors/clut4d_pre1d_post1d.txt testrig/testvectors/img4d.txt
        cat $^ > $@
        $(PRDATA2PG) prplus4d.txt pg4d.txt

clean:
        $(RM) $(TARGETS) $(COPIES) $(INTERMEDIATES)
        $(RMDIR) $(TEMPDIRS)

# Verilog dependency files for each static project
$(TARGETS_STATIC): $(STATIC_DIR)/csc_defs.vh
$(STATIC_DIR)/csc_auto_load/csc_auto_load.ngc : $(STATIC_DIR)/csc_auto_load.v
$(STATIC_DIR)/csc_control/csc_control.ngc : $(STATIC_DIR)/csc_control.v
$(STATIC_DIR)/csc_isolation_stage/csc_isolation_stage_69.ngc :
    $(STATIC_DIR)/csc_isolation_stage_69.v
$(STATIC_DIR)/csc_isolation_stage/csc_isolation_stage_100.ngc :
    $(STATIC_DIR)/csc_isolation_stage_100.v
$(STATIC_DIR)/csc_k_plane_mag/csc_k_plane_mag.ngc : $(STATIC_DIR)/csc_k_plane_mag.v
$(STATIC_DIR)/csc_lut1d/csc_lut1d_post.ngc : $(STATIC_DIR)/csc_lut1d_post.v \
        $(STATIC_DIR)/csc_phase1_1d.v $(STATIC_DIR)/csc_phase2_1d.v \
        $(STATIC_DIR)/csc_lut_wrappers_1d.v $(STATIC_DIR)/rbist_defs.vh
$(STATIC_DIR)/csc_lut1d/csc_lut1d_pre.ngc : $(STATIC_DIR)/csc_lut1d_pre.v \
        $(STATIC_DIR)/csc_phase1_1d.v $(STATIC_DIR)/csc_phase2_1d.v \
        $(STATIC_DIR)/csc_lut_wrappers_1d.v $(STATIC_DIR)/rbist_defs.vh
$(STATIC_DIR)/csc_pre_match/csc_pre_match.ngc : $(STATIC_DIR)/csc_pre_match.v
$(STATIC_DIR)/csc_reg/csc_reg.ngc : $(STATIC_DIR)/csc_reg.v
$(STATIC_DIR)/csc_wrapper/csc_wrapper.ngc : $(STATIC_DIR)/CSC_wrapper.vhd
$(STATIC_DIR)/icap_eai/icap_eai.ngc : $(STATIC_DIR)/icap_eai.v
$(STATIC_DIR)/lutchk_top/lutchk_top.ngc : $(STATIC_DIR)/lutchk_ctrl.v \
        $(STATIC_DIR)/lutchk_regs.v $(STATIC_DIR)/lutchk_timer.v
            $(STATIC_DIR)/lutchk_top.v \
        $(STATIC_DIR)/csc_lutchk_crc16_parallel_n.v
$(STATIC_DIR)/pipeline_handshake_enable/pipeline_handshake_enable.ngc : \
        $(STATIC_DIR)/pipeline_handshake_enable.v
$(STATIC_DIR)/TestRig2DUTInterface/TestRig2DUTInterface.ngc : \
        $(STATIC_DIR)/TestRig2DUTInterface.v

.PHONY: coregen3d coregen4d
```

Appendix B

Hex2Coe

```
// This utility converts hex files to the coe format
// Input: Hex file
// Output: 2 coe files (for upper, lower)

// Usage: hex2coe [-c|-m] [-1|-3|-4] [-N|-C] -i <input> [-o <output stem>]
//      -c: specifies the output should be coe format (default)
//      -m: specifies the output should be mif format (not yet implemented)
//      -1: specifies the file is for a 1D memory (48 bits, 12 bits per channel)
//  -3 -4: specifies the file is for a 3D/4D memory (40 bits, 10 bits per channel)
//      -N: specifies the input is in normal mode (12 bits/channel for 1D, 10
    bits/channel for 3D/4D)
//      -C: specifies the input is in compact mode (8 bits/channel for 1D, 8 bits/channel
    for 3D/4D)
//  -f: Outputs the full 40 bit coe file in addition to the divided channels (3D/4D only)
//  -h: display this help and exit
// If neither -N nor -C is given, will attempt to determine the format based on the
    contents.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define CH01 "_ch01"
#define CH23 "_ch23"
#define COEEXT ".coe"
#define MIFEXT ".mif"
#define EXTSIZE 4
#define COEHEADERCOMMENTCH01 "; Initialization file for ch01 of %s\n"
#define COEHEADERCOMMENTCH23 "; Initialization file for ch23 of %s\n"
#define COEHEADERCOMMENTFULL "; Initialization file for %s\n"
#define COERADIX "memory_initialization_radix=16;\n"
#define COEVECTOR "memory_initialization_vector=\n"
#define COEEND ";"

typedef enum OUTFORMAT {COE, MIF} e_outformat;
typedef enum MEMTYPE {D1, D3, D4} e_memtype;
typedef enum MODE {NORMAL, COMPACT} e_mode;

typedef struct MEM3D4D {
  unsigned int ch0:10;
  unsigned int ch1:10;
  unsigned int ch2:10;
  unsigned int ch3:10;
```

```
} s_mem3d4d;
typedef struct MEM1D {
  unsigned int ch0:12;
  unsigned int ch1:12;
  unsigned int ch2:12;
  unsigned int ch3:12;
} s_mem1d;

void usage();
unsigned int atoh(char);
char* dec2bin(int val, char* str, int len);

int main(int argc, char *argv[]) {
  char *infilename;
  char *outfilenamebase;
  char *outfilenamech01;
  char *outfilenamech23;
  char *outfilenamefull;
  FILE *infile;
  FILE *outfilech01;
  FILE *outfilech23;
  FILE *outfilefull;
  int mode_set = 0;        // was mode specified by options
  int mode_determined = 0;         // if mode not specified, try to determine
  int type_set = 0;        // was type specified by options
  int oformat_set = 0;     // was output format specified by options
  int inname_set = 0;      // was input name provided
  int full_output = 0; // specifies for 3d/4d if the full 40 bit coe should be written
  int i;
  e_outformat oformat = COE;
  e_memtype type = D3;
  e_mode mode = NORMAL;
  int outputnameprovided = 0;
  s_mem3d4d mem3d4d_out;
  s_mem1d mem1d_out;
  char buffer[10];         // hex files have 8 characters per line (+1 for /n and +1 for
      /0)
  char mif_1d[13];         // 12 bits + /n
  char mif_3d4d[11];       // 10 bits + /n
  char* ext = COEEXT; // default
  mif_1d[12]='\0';
  mif_3d4d[10]='\0';

  if (argc == 1) { // no arguments
    usage();
    exit(1);
  }
  else
  {
    // parse command line
    for(i=1; i<argc; i++)
    {
      if (argv[i][0]=='-'){
        switch (argv[i][1]){
          case 'c':
            if(oformat_set == 0){
              oformat = COE;
              oformat_set = 1;
```

```
    }
    else{
      if (oformat != COE)
      {
        fprintf(stderr, "-c option conflicts with previous -m option. "
            Exiting.\n");
        exit(1);
      }
    }
    break;
  case 'm':
    if(oformat_set == 0){
      oformat = MIF;
      oformat_set = 1;
    }
    else{
      if (oformat != MIF)
      {
        fprintf(stderr, "-m option conflicts with previous -c option. "
            Exiting.\n");
        exit(1);
      }
    }
    break;
  case '1':
    if(type_set == 0)
    {
      type = D1;
      type_set = 1;
    }
    else{
      if (type != D1)
      {
        fprintf(stderr, "-1 option conflicts with previous -3 or -4 option. "
            Exiting.\n");
        exit(1);
      }
    }
    break;
  case '3':
    if(type_set == 0)
    {
      type = D3;
      type_set = 1;
    }
    else{
      if (type != D3)
      {
        fprintf(stderr, "-3 option conflicts with previous -1 or -4 option. "
            Exiting.\n");
        exit(1);
      }
    }
    break;
  case '4':
    if(type_set == 0)
    {
      type = D4;
```

```
      type_set = 1;
    }
    else{
      if (type != D4)
      {
        fprintf(stderr, "-4 option conflicts with previous -1 or -3 option. 
            Exiting.\n");
        exit(1);
      }
    }
    break;
  case 'N':
    if(mode_set == 0)
    {
      mode = NORMAL;
      mode_set = 1;
    }
    else{
      if (mode != NORMAL)
      {
        fprintf(stderr, "-N option conflicts with previous -C option. 
            Exiting.\n");
        exit(1);
      }
    }
    break;
  case 'C':
    if(mode_set == 0)
    {
      mode = COMPACT;
      mode_set = 1;
    }
    else{
      if (mode != COMPACT)
      {
        fprintf(stderr, "-C option conflicts with previous -N option. 
            Exiting.\n");
        exit(1);
      }
    }
    break;
  case 'i':
    if(inname_set == 0)
    {
      infilename = argv[i+1];
      i++;
      inname_set = 1;
    }
    else{
      fprintf(stderr, "Multiple sources provided. Exiting.\n");
      exit(1);
    }
    break;
  case 'o':
    if (outputnameprovided == 0)
    {
      outfilenamebase = argv[i+1];
      i++;
```

```
            outputnameprovided = 1;
          }
          else{
            fprintf(stderr, "Multiple targets provided. Exiting.\n");
            exit(1);
          }
          break;
        case 'f':
          full_output = 1;
          break;
        case 'h':
          usage();
          exit(0);
          break;
        default:
          fprintf(stderr,"Invalid option %s\n",argv[i]);
          usage();
          exit(1);
          break;
      }
    }
    else
    {
      usage();
      exit(1);
    }
  }
}
printf("Done reading options...\n");
// Generate the output base name if none was provided
if(outputnameprovided==0){
  outfilenamebase = (char *)malloc(strlen(infilename));
  if (outfilenamebase==NULL) {
    fprintf(stderr,"Could not allocate memory for output file name!");
    exit(1);
  }
  strcpy(outfilenamebase, infilename);
  outfilenamebase = strtok(outfilenamebase,"."); // this fails if the input file name
      is of the sort x.y.hex
}

// DEBUG
if(oformat_set==0){
  printf("No output format set. Using default\n");
}
if(type_set==0){
  printf("No memory type set. Using default\n");
}
if(mode_set==0){
  printf("No mode set. Will try to determine by contents\n");
}
if(outputnameprovided==0){
  printf("No output name provided. Using default\n");
}
printf("\n");
printf("Output Format: %s\n", (oformat==COE)?"COE":"MIF");
printf("Memory type: %s\n",(type==D1)?"1D":((type==D3)?"3D":"4D"));
printf("Memory Mode: %s\n",(mode==NORMAL)?"Normal":"Compact");
```

```
printf("Input file: %s\n", infilename);
printf("Output file: %s\n", outfilenamebase);
printf("\n");
// END DEBUG

if(oformat==MIF)
  ext=MIFEXT;

// open hex file
if (infilename != NULL){
  infile = fopen(infilename, "r");
  if(infile==NULL){
    fprintf(stderr,"Can't open file %s for reading!\n", infilename);
    exit(1);
  }
}

// open output files
if (outfilenamebase != NULL){
  outfilenamech01 = (char*)malloc(strlen(outfilenamebase)+strlen(CH01)+EXTSIZE);
  strcpy(outfilenamech01,outfilenamebase);
  strcat(outfilenamech01,CH01);
  strcat(outfilenamech01,ext);
  outfilech01 = fopen(outfilenamech01, "w");
  if(outfilech01 == NULL){
    fprintf(stderr,"Can't open file %s for writing!\n",outfilenamech01);
    exit(1);
  }
  outfilenamech23 = (char*)malloc(strlen(outfilenamebase)+strlen(CH23)+EXTSIZE);
  strcpy(outfilenamech23,outfilenamebase);
  strcat(outfilenamech23,CH23);
  strcat(outfilenamech23,ext);
  outfilech23 = fopen(outfilenamech23, "w");
  if(outfilech23 == NULL){
    fprintf(stderr,"Can't open file %s for writing!\n",outfilenamech23);
    exit(1);
  }
  outfilenamefull = (char*)malloc(strlen(outfilenamebase)+EXTSIZE);
  strcpy(outfilenamefull,outfilenamebase);
  strcat(outfilenamefull,ext);
  outfilefull = fopen(outfilenamefull, "w");
  if(outfilefull == NULL){
    fprintf(stderr,"Can't open file %s for writing!\n",outfilenamefull);
    exit(1);
  }
}

// write headers to COE files
if( oformat==COE ){
  fprintf(outfilech01, COEHEADERCOMMENTCH01, infilename);
  fprintf(outfilech01, COERADIX);
  fprintf(outfilech01, COEVECTOR);

  fprintf(outfilech23, COEHEADERCOMMENTCH23, infilename);
  fprintf(outfilech23, COERADIX);
  fprintf(outfilech23, COEVECTOR);

  fprintf(outfilefull, COEHEADERCOMMENTFULL, infilename);
```

```
  fprintf(outfilefull, COERADIX);
  fprintf(outfilefull, COEVECTOR);
}

// buffer size is 10 for 8 characters + newline + linefeed (change if system changes)
while(fgets(buffer, 10, infile)!=NULL){ // Read a 32 bit line and divide into channels
  if(mode_set==1 && mode==NORMAL){
    // process the input file as if it's a Normal mode (ie, 12/10 bits for 1/3/4D)
    printf("Normal mode specified\n");
    if(type == D1){ // 1d, 12 bits per channel
      // buffer: 0WWW0XXX\n\0
      printf("1-D Memory\n");
      printf("HEX: %s",buffer);
      mem1d_out.ch0 = atoh( buffer[1] );
      mem1d_out.ch0 = mem1d_out.ch0 << 4;
      mem1d_out.ch0 += atoh( buffer[2] );
      mem1d_out.ch0 = mem1d_out.ch0 << 4;
      mem1d_out.ch0 += atoh( buffer[3] );

      mem1d_out.ch1 = atoh( buffer[5] );
      mem1d_out.ch1 = mem1d_out.ch1 << 4;
      mem1d_out.ch1 += atoh( buffer[6] );
      mem1d_out.ch1 = mem1d_out.ch1 << 4;
      mem1d_out.ch1 += atoh( buffer[7] );
      if(fgets(buffer, 10, infile)!=NULL){
        printf("HEX: %s",buffer);
        mem1d_out.ch2 = atoh( buffer[1] );
        mem1d_out.ch2 = mem1d_out.ch2 << 4;
        mem1d_out.ch2 += atoh( buffer[2] );
        mem1d_out.ch2 = mem1d_out.ch2 << 4;
        mem1d_out.ch2 += atoh( buffer[3] );

        mem1d_out.ch3 = atoh( buffer[5] );
        mem1d_out.ch3 = mem1d_out.ch3 << 4;
        mem1d_out.ch3 += atoh( buffer[6] );
        mem1d_out.ch3 = mem1d_out.ch3 << 4;
        mem1d_out.ch3 += atoh( buffer[7] );
      }
      printf("Channels: %x %x %x %x\n", mem1d_out.ch0,
          mem1d_out.ch1,mem1d_out.ch2,mem1d_out.ch3);
      printf("Channels(merged): %x %x\n\n", (mem1d_out.ch0<<12)+mem1d_out.ch1,
          (mem1d_out.ch2<<12)+mem1d_out.ch3);
      // write channels to files (COE or MIF)
      if( oformat == COE ){
        fprintf(outfilech01, "%x\n", (mem1d_out.ch0<<12)+mem1d_out.ch1);
        fprintf(outfilech23, "%x\n", (mem1d_out.ch2<<12)+mem1d_out.ch3);
      }
      else{
        dec2bin(mem1d_out.ch0, mif_1d, 12);
        fprintf(outfilech01, "%s",mif_1d);
        dec2bin(mem1d_out.ch1, mif_1d, 12);
        fprintf(outfilech01, "%s\n",mif_1d);
        dec2bin(mem1d_out.ch2, mif_1d, 12);
        fprintf(outfilech23, "%s",mif_1d);
        dec2bin(mem1d_out.ch3, mif_1d, 12);
        fprintf(outfilech23, "%s\n",mif_1d);
      }
    }
```

```
  else{ // 3d or 4d, same format, 10 bits per channel
    // buffer: 0[00ww]WW0[00xx]XX
    printf("3D/4D Memory\n");
    printf("HEX: %s",buffer);
    mem3d4d_out.ch0 = atoh( buffer[1] ); // ch0=ww
    mem3d4d_out.ch0 = mem3d4d_out.ch0 << 4;
    mem3d4d_out.ch0 += atoh( buffer[2] );  // ch0=wwW
    mem3d4d_out.ch0 = mem3d4d_out.ch0 << 4;
    mem3d4d_out.ch0 += atoh( buffer[3] );  // ch0=wwWW

    mem3d4d_out.ch1 = atoh( buffer[5] );
    mem3d4d_out.ch1 = mem3d4d_out.ch1 << 4;
    mem3d4d_out.ch1 += atoh( buffer[6] );
    mem3d4d_out.ch1 = mem3d4d_out.ch1 << 4;
    mem3d4d_out.ch1 += atoh( buffer[7] );
    if(fgets(buffer, 10, infile)!=NULL){
      printf("HEX: %s",buffer);
      mem3d4d_out.ch2 = atoh( buffer[1] );
      mem3d4d_out.ch2 = mem3d4d_out.ch2 << 4;
      mem3d4d_out.ch2 += atoh( buffer[2] );
      mem3d4d_out.ch2 = mem3d4d_out.ch2 << 4;
      mem3d4d_out.ch2 += atoh( buffer[3] );

      mem3d4d_out.ch3 = atoh( buffer[5] );
      mem3d4d_out.ch3 = mem3d4d_out.ch3 << 4;
      mem3d4d_out.ch3 += atoh( buffer[6] );
      mem3d4d_out.ch3 = mem3d4d_out.ch3 << 4;
      mem3d4d_out.ch3 += atoh( buffer[7] );
    }
    // debug
    printf("Channels: %x %x %x %x\n", mem3d4d_out.ch0,
        mem3d4d_out.ch1,mem3d4d_out.ch2,mem3d4d_out.ch3);
    printf("Channels(merged): %x %x\n\n", (mem3d4d_out.ch0<<10)+mem3d4d_out.ch1,
        (mem3d4d_out.ch2<<10)+mem3d4d_out.ch3);
    // add output code here
    if( oformat == COE ){
      fprintf(outfilech01, "%x\n", (mem3d4d_out.ch0<<10)+mem3d4d_out.ch1);
      fprintf(outfilech23, "%x\n", (mem3d4d_out.ch2<<10)+mem3d4d_out.ch3);
      fprintf(outfilefull, "%x%x\n", (mem3d4d_out.ch0<<10)+mem3d4d_out.ch1,
          (mem3d4d_out.ch2<<10)+mem3d4d_out.ch3);
    }
    else{
      // add code for MIF output (binary)
      dec2bin(mem3d4d_out.ch0, mif_3d4d, 10);
      fprintf(outfilech01, "%s",mif_3d4d);
      dec2bin(mem3d4d_out.ch1, mif_3d4d, 10);
      fprintf(outfilech01, "%s\n",mif_3d4d);
      dec2bin(mem3d4d_out.ch2, mif_3d4d, 10);
      fprintf(outfilech23, "%s",mif_3d4d);
      dec2bin(mem3d4d_out.ch3, mif_3d4d, 10);
      fprintf(outfilech23, "%s\n",mif_3d4d);
    }
  }

}
else if (mode_set==1 && mode==COMPACT)  {        // Compact mode (8 bits per channel
    in source)
  printf("Compact mode specified\n");
```

```
if(type == D1){ // 1d, 12 bits per channel
  // buffer: WWXXYYZZ\n\0
  printf("1D memory\n");
  printf("HEX: %s",buffer);
  mem1d_out.ch0 = atoh( buffer[0] );
  mem1d_out.ch0 = mem1d_out.ch0 << 4;
  mem1d_out.ch0 += atoh( buffer[1] );
  mem1d_out.ch0 = mem1d_out.ch0 << 4;

  mem1d_out.ch1 = atoh( buffer[2] );
  mem1d_out.ch1 = mem1d_out.ch1 << 4;
  mem1d_out.ch1 += atoh( buffer[3] );
  mem1d_out.ch1 = mem1d_out.ch1 << 4;

  mem1d_out.ch2 = atoh( buffer[4] );
  mem1d_out.ch2 = mem1d_out.ch2 << 4;
  mem1d_out.ch2 += atoh( buffer[5] );
  mem1d_out.ch2 = mem1d_out.ch2 << 4;

  mem1d_out.ch3 = atoh( buffer[6] );
  mem1d_out.ch3 = mem1d_out.ch3 << 4;
  mem1d_out.ch3 += atoh( buffer[7] );
  mem1d_out.ch3 = mem1d_out.ch3 << 4;

  printf("Channels: %x %x %x %x\n", mem1d_out.ch0,
      mem1d_out.ch1,mem1d_out.ch2,mem1d_out.ch3);
  printf("Channels(merged): %x %x\n\n", (mem1d_out.ch0<<12)+mem1d_out.ch1,
      (mem1d_out.ch2<<12)+mem1d_out.ch3);
  // add output code here
  if( oformat == COE ){
    fprintf(outfilech01, "%x\n", (mem1d_out.ch0<<12)+mem1d_out.ch1);
    fprintf(outfilech23, "%x\n", (mem1d_out.ch2<<12)+mem1d_out.ch3);
  }
  else{
    // add code for MIF output (binary)
    dec2bin(mem1d_out.ch0, mif_1d, 12);
    fprintf(outfilech01, "%s",mif_1d);
    dec2bin(mem1d_out.ch1, mif_1d, 12);
    fprintf(outfilech01, "%s\n",mif_1d);
    dec2bin(mem1d_out.ch2, mif_1d, 12);
    fprintf(outfilech23, "%s",mif_1d);
    dec2bin(mem1d_out.ch3, mif_1d, 12);
    fprintf(outfilech23, "%s\n",mif_1d);
  }
}
else{ // 3d or 4d, same format, 10 bits per channel
  // buffer: WWXXYYZZ\n\0
  printf("3D or 4D memory\n");
  printf("HEX: %s",buffer);
  mem3d4d_out.ch0 = atoh( buffer[0] ); // ch0=ww
  mem3d4d_out.ch0 = mem3d4d_out.ch0 << 4;
  mem3d4d_out.ch0 += atoh( buffer[1] );  // ch0=wwW
  mem3d4d_out.ch0 = mem3d4d_out.ch0 << 2;

  mem3d4d_out.ch1 = atoh( buffer[2] );
  mem3d4d_out.ch1 = mem3d4d_out.ch1 << 4;
  mem3d4d_out.ch1 += atoh( buffer[3] );
  mem3d4d_out.ch1 = mem3d4d_out.ch1 << 2;
```

```
        mem3d4d_out.ch2 = atoh( buffer[4] );
        mem3d4d_out.ch2 = mem3d4d_out.ch2 << 4;
        mem3d4d_out.ch2 += atoh( buffer[5] );
        mem3d4d_out.ch2 = mem3d4d_out.ch2 << 2;

        mem3d4d_out.ch3 = atoh( buffer[6] );
        mem3d4d_out.ch3 = mem3d4d_out.ch3 << 4;
        mem3d4d_out.ch3 += atoh( buffer[7] );
        mem3d4d_out.ch3 = mem3d4d_out.ch3 << 2;
        // debug
        printf("Channels: %x %x %x %x\n", mem3d4d_out.ch0,
            mem3d4d_out.ch1,mem3d4d_out.ch2,mem3d4d_out.ch3);
        printf("Channels(merged): %x %x\n\n", (mem3d4d_out.ch0<<10)+mem3d4d_out.ch1,
            (mem3d4d_out.ch2<<10)+mem3d4d_out.ch3);
        // add output code here
        if( oformat == COE ){
          fprintf(outfilech01, "%x\n", (mem3d4d_out.ch0<<10)+mem3d4d_out.ch1);
          fprintf(outfilech23, "%x\n", (mem3d4d_out.ch2<<10)+mem3d4d_out.ch3);
          fprintf(outfilefull, "%x%x\n", (mem3d4d_out.ch0<<10)+mem3d4d_out.ch1,
              (mem3d4d_out.ch2<<10)+mem3d4d_out.ch3);
        }
        else{
          // add code for MIF output (binary)
          dec2bin(mem3d4d_out.ch0, mif_3d4d, 10);
          fprintf(outfilech01, "%s",mif_3d4d);
          dec2bin(mem3d4d_out.ch1, mif_3d4d, 10);
          fprintf(outfilech01, "%s\n",mif_3d4d);
          dec2bin(mem3d4d_out.ch2, mif_3d4d, 10);
          fprintf(outfilech23, "%s",mif_3d4d);
          dec2bin(mem3d4d_out.ch3, mif_3d4d, 10);
          fprintf(outfilech23, "%s\n",mif_3d4d);
        }
      }
    }
  }

  if( oformat==COE ){
    fprintf(outfilech01, COEEND);
    fprintf(outfilech23, COEEND);
  }

  if(infile != NULL)
    fclose(infile);
  if(outfilech01 != NULL)
    fclose(outfilech01);
  if(outfilech23 != NULL)
    fclose(outfilech23);
  return 0;
}

void usage(){
  printf("Usage: hex2coe [-c|-m] [-1|-3|-4] [-N|-C] -i <input> [-o <output stem>]\n");
  printf("    -c: specifies the output should be coe format (default)\n");
  printf("    -m: specifies the output should be mif format (not yet implemented)\n");
  printf("    -1: specifies the file is for a 1D memory (48 bits, 12 bits per channel)\n");
  printf("    -3 -4: specifies the file is for a 3D/4D memory (40 bits, 10 bits per
      channel)\n");
```

```
  printf("   -N: specifies the input is in normal mode (12 bits/channel for 1D, 10
      bits/channel for 3D/4D)\n");
  printf("   -C: specifies the input is in compact mode (8 bits/channel for 1D, 8
      bits/channel for 3D/4D)\n");
  printf("   -f: Outputs the full 40 bit coe file in addition to the divided channels
      (3D/4D only)\n");
  printf("   -h: display this help and exit\n");
  printf("If neither -N nor -C is given, will attempt to determine the format based on
      the contents.\n");
  return;
}

// Converts an ascii hex string 0-9A-F to an integer
unsigned int atoh(char str)
{
  unsigned int Value = 0, Digit;
  char c;
    c=str;
    if (c >= '0' && c <= '9'){
      Digit = (unsigned int) (c - '0');
    }
    else if (c >= 'a' && c <= 'f'){
      Digit = (unsigned int) (c - 'a') + 10;
    }
    else if (c >= 'A' && c <= 'F'){
      Digit = (unsigned int) (c - 'A') + 10;
    }
    else{
    }
    Value = (Value << 4) + Digit;
  return Value;
}

// Converts an integer to a binary string
char* dec2bin(int val, char* str, int len)
{
  int i;
  for (i = len-1; i>=0; i--){
    str[i]=(val%2==0)?'0':'1';
    val = val>>1;
  }
  return str;
}
```