2009

VERIFICATION AND DEBUG TECHNIQUES FOR INTEGRATED CIRCUIT DESIGNS

David Allen Crutchfield
University of Kentucky, crutch@lexmark.com

ABSTRACT OF THESIS

VERIFICATION AND DEBUG TECHNIQUES FOR INTEGRATED CIRCUIT DESIGNS

Verification and debug of integrated circuits for embedded applications has grown in importance as the complexity in function has increased dramatically over time. Various modeling and debugging techniques have been developed to overcome the overwhelming challenge. This thesis attempts to address verification and debug methods by presenting an accurate C model at the bit and algorithm level coupled with an implemented Hardware Description Language (HDL). Key concepts such as common signal and variable naming conventions are incorporated as well as a stepping function within the implemented HDL. Additionally, a common interface between low-level drivers and C models is presented for early firmware development and system debug. Finally, self-checking verification is discussed for delivering multiple test cases along with testbench portability.

KEYWORDS: C Modeling, Hardware Description Language, Verification, Debug, Testbench Portability

David Crutchfield

November 20, 2009

VERIFICATION AND DEBUG TECHNIQUES FOR INTEGRATED CIRCUIT DESIGNS

By

David Allen Crutchfield

Dr. Bruce Walcott
Director of Thesis

Dr. Stephen Gedney
Director of Graduate Studies

November 20, 2009

RULES FOR THE USE OF THESIS

Unpublished thesis submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name	Date

THESIS

David Allen Crutchfield

The Graduate School

University of Kentucky

2009

VERIFICATION AND DEBUG TECHNIQUES FOR INTEGRATED CIRCUIT DESIGNS

THESIS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering at the University of Kentucky

By

David Allen Crutchfield

Georgetown, Kentucky

Director: Dr. Bruce Walcott, Professor of Electrical Engineering

Lexington, Kentucky

2009

ACKNOWLEDGEMENTS

I want to thank Dr. Bruce Walcott, my advisor and friend, for his support throughout this process. Despite my ineffectiveness in choosing a path and completing the project in a timely fashion, he continued to motivate and encourage me. Even when I waited until literally the last hour, he helped me through several administrative hoops in the end. I am deeply indebted to Dr. Walcott for getting me to this point in both my graduate and undergraduate studies. His influence can be seen not only in my life, but also in the lives of others. I want to thank Dr. J. Robert Heath for serving on my committee and for all his support during my undergraduate studies. His influence has helped shape my desire for logic design and embedded microcontroller applications. I want to also thank Dr. YuMing Zhang for serving on my committee.

Next, I would like to thank several of my colleagues at Lexmark for their support at the final hour. I would like to thank Chris Case, James Sharpe and Zach Fister for taking extra time from their busy lives to read an unpolished version of this thesis and give critical and constructive feedback. I respect their talents and look forward to many more years as their co-worker.

I am grateful for my family and their support. My mother, Margaret Gilbert, instilled the desire to overachieve and pursue my dreams. I am thankful to her for that and for her love. Obviously, I could not have done this without my beautiful wife, Brandy. I thank her for being patient and loving after so many years and taking the boys away for that pivotal week. Without the solitude I could have never pulled this off.

Lastly, I want to thank my Lord and Savior Jesus Christ in whom all things are possible.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Verification and debug of integrated circuits for embedded applications has grown in importance as the complexity in function has increased dramatically over time. Various modeling and debugging techniques have been developed to overcome this overwhelming challenge. This thesis attempts to address verification and debug methods, by presenting an accurate C model at the algorithm and bit levels coupled with an implemented Hardware Description Language (HDL). Key concepts, such as common signal and variable naming conventions, are incorporated, as well as, a stepping function within the implemented HDL. Additionally, a common interface between low-level drivers and C models is presented for early firmware development and system debug. Finally, self-checking verification is discussed for delivering multiple test cases along with testbench portability.

In Chapter 2, the reader will find background material in the area of verification and debug of integrated circuits implementing complex algorithms. It should become obvious that the key concepts mentioned above are required for success in industry. While many of these concepts have been considered individually, merging them together into one model has been overlooked. Chapter 3 attempts to link the background material of Chapter 2 and point out the key concepts mentioned above.

Chapter 4 will discuss an implementation of the key concepts. This will be presented in an example using error-diffusion, which is a complex image processing task, especially needed in the printing industry, in particular with thermal ink-jets. Examples

of C code, Very High Speed Integrated Circuit Hardware Description Language (VHSICHDL or VHDL), register specifications of implemented HDL, and bus modeling will be used.

Chapter 5 will discuss error-diffusion further and explain how the key concepts lead to improvements in verification and debug through real examples. A Linear Feedback Shift Register (LFSR) design will be used to discuss the need for accurate modeling during verification and debug. After pointing out the need for accuracy, the topics of algorithm proofing and self-checking verification will be addressed.

Finally, Chapter 6 will present the next step in verification using an accurate self-checking C model. Here, it will be revealed that device independent portability can be achieved using C code generation. In continuation, constrained random verification using System Verilog will be discussed.

Chapter 2

BACKGROUND RESEARCH

To address the growing complexity of integrated circuits for embedded applications, and the verification and debug challenge associated with them, various modeling and debugging techniques have been developed. For instance, F. Wotawa presented a model-based diagnosis tool, VHDLDIAG, designed to automatically locate bugs in VHDL programs [1]. VHDLDIAG utilizes model based diagnosis for fault localization which requires a logic model of the VHDL program to be debugged [1]. While fault localization is useful, Wotawa admitted that unique identification of a fault is not guaranteed [1]. Thus, a C model that could help pinpoint failures within hardware would be more advantageous than fault localization.

The Open SystemC Initiative (OSCI) is an independent, not-for-profit association dedicated to defining and advancing SystemC™ as an open industry standard for system-level modeling, design and verification [2]. The IEEE Standards Association approved the standard for the SystemC library as IEEE Standard 1666-2005 [2]. The first version was released on February 2, 2002 [2]. OSCI was launched in 1999 [2]. Sanguinetti and Pursley proposed that SystemC™ would bridge the gap between high-level modeling and top-down design methodology [3]. While it is common practice to begin design with a high-level model, general-purpose programming languages, such as C/C++, are not acceptable languages for Register Transfer Level (RTL) modeling and synthesis [3]. SystemC™ adds synthesis and hardware architecture constructs to a general-purpose programming language [3]. This thesis shares the argument that system-level modeling is needed but accomplishes the same task without additional libraries or language

learning. It is not intended to bridge the gap with modeling and synthesis using C modeling. Both Verilog and VHDL are preferred languages by most designers in industry for RTL representation.

In most cases, system-on-chip (SOC) design is a sequential process, where supporting software, often referred to as firmware, cannot be developed until after the HDL implementation is complete [4]. Moreover, it is desirable to explore different design alternatives for complex systems which can be unreasonable using slow simulations of the HDL implementation [4]. Kruijtzer, Reyes and Gehrke argued the use of System C to provide a high level model developed with System C for early firmware development and algorithm proofing [4]. While it is true that a high level model is useful for most tasks, a C model with more low level debugging features would support additional needs such as direct simulation verification, emulation, and final system debug.

Designers spend up to 70% of their time developing and running tests attempting to verify their designs [5] [6]. Regression simulations, alone, can take several years of CPU time for completion [6]. As complexity of designs increase, the effort to verify those designs is increasing at a faster rate than Moore's Law [5] [6] [7]. One assertion suggests that the time required for verification increases as the square of the size of the design [6]. There is a great deal of interest in increasing verification capabilities and efficiencies today within industry and academia [6]. Much work has been applied to coverage tools both practically and theoretically and most HDL logic simulators have coverage tools either as optional or standard features [6]. Additionally, different methods of generating effective stimulus automatically have been developed [6]. For instance, constrained random stimulus has matured with many EDA companies offering some sort

of stimulus generation capabilities [6] [8] [9]. To aid in stimulus generation, Klein and Piekarz argue that using software is an effective means to help with the growing possibilities [6]. Using software would stimulate the design identically as it would be used in the final project [6]. This technique assumes that a processor functional model is available very early in the design stage. As an alternative, a device independent bus stimulus model capable of executing simple instructions would be more portable and accessible for these early design stages. It is also apparent that an accurate C model with an expected interface could be used to develop software which would test hardware functional expectations.

In 2008, Ng presented the usefulness of high-level models written in C and C++ focusing on RTL hardware model verification [10]. Figure 2.1 was presented as the simulation based RTL verification flow using system level models [10]. A C model enabling the verification flow described by Ng is desired. Simulation input vectors are needed as well as output vectors or expected results for file comparison. This will lead to a pass/fail structure eliminating the need for manual human interaction.

In July 2007, Brier and Mitra presented the importance of C/C++ models for architecture exploration and verification of DSP designs [11]. In this presentation important aspects of this C modeling technique are represented. It was imperative that the C model be as block accurate as possible giving the following benefits [11]:

- C model matches the functional block diagram [11].
- C model modules match RTL module partitioning [11].
- When considering re-use, modifications may be accomplished by substituting the same blocks in the RTL and C models [11].

- Observation points are more exposed and correlate [11].

These benefits would enhance debuggability and lead to higher quality design re-use [11]. Additionally, they offered that bit real representations of the RTL would enable faster debug of the RTL implementation for arithmetic structures, as well as, rounding and data concatenation [11]. Finally, Brier and Mitra suggested that cycle accurate modeling along with the model's functionality allowed for flagging errors in simulation much easier [11]. The C model presented in this thesis incorporates several of the aspects Brier and Mitra included as being imperative. It is important to have a model that matches RTL module partitioning as well as signal level naming conventions. Additionally, re-use within an emulation setting where a C model can easily replace hardware drivers, aids in debugging both firmware and hardware. Finally, a C model and implemented HDL that exposes observation points for correlation is needed.

In discussing the typical design flow for digital implementation Bertacco provided Figure 2.2 [12]. A C model that can be used at each verification stage of this flow is imperative. In most cases, when traditional signal forcing is used in verification for stimulus, the test cases have to be modified when moving from RTL to gate level verification. This type of modification is not needed in a system level verification environment where stimulus is generated via register interaction. Bertacco goes on to reinforce the difficulty of verification, particularly the verification infrastructure [12]. Distinct verification practices often change with subsequent designs due to the insufficient "correctness confidence-level" that most current approaches provide [12]. It becomes clear that more than 70% of design time and engineering resources are spent in

verification [12]. To address resource constraints and minimize modification of test cases at each verification level, a C model promoting portability is desired.

In his publication, Wilcox stated that finding 98% of the bugs in a design is just a matter of using methodical process [13]. However, finding the last 2% takes the most time and effort [13]. Tough bugs are hard to locate because of misconceptions. These obstacles involve a large amount of design time and they require many simulation cycles to find and recreate [13]. In many cases, verification engineers do not understand the operation of the design, therefore finding it difficult to focus on the root cause [13]. Long debug cycles are unacceptable with limited time and resources. It is very important in today's industry that verification teams reach their goals in less time with the least amount of resources possible [13]. A mantra of many good code developers is "write once and use often" [13]. Therefore, a C model which can address several of the issues as recognized by Wilcox would be beneficial. As an example, it would be advantageous for a C model to closely represent the implemented HDL as to expose several internal signals for observation. Additionally, if the implemented HDL exposes similar signals then debug time can be shortened drastically.

Dearth, Meeth and Whittemore presented a network driven system using C++ and Verilog Bus Functional Models to create a co-simulation environment between software and hardware engineers [14]. Figure 2.3 depicts a simplified model of the Object Based Interface they discuss [14]. An accurate C model is beneficial for developing the tests of this system. The tests would then be applied through the Object Based Interface to drive the device under test (DUT).

Li and Nagarajan recognized the importance of testing techniques which utilize C/C++ models that match bit for bit the VHDL/Verilog RTL model [15]. Additionally, they placed within their models probing nodes for comparison to facilitate early problem identification and resolution [15]. Li and Nagarajan also stressed that the image processing driver layers should be decoupled separating out the parameters from the code [15]. It has been reported that 61% of all new ASICs in industry require at least one respin [15] [16]. Their attempts were to present techniques that help reduce this number [15]. It is agreed that a C model incorporating philosophies that Li and Nagarajan recognized in 2005 is necessary for efficient verification and debug techniques.

Recognizing the importance of faster verification needs in 2004, Bernstein, Burton and Ghenassia introduced Transaction Level Modeling (TLM) to bridge the abstraction gap in system level modeling and design [17]. TLM addresses a number of practical system level design problems [17]. The OSCI Transaction Level Working Group (TLMWG) has identified a number of abstraction levels at which modeling can take place [17] [18] [19]. These include:

- *Algorithmic*: At the algorithmic level, there is no distinction made between hardware and software [17].
- *Software View*: At this level, there is a division made between hardware and software[17]. The model is at least suitable for programmers to develop their software [17]. This level is also referred to as the *Architectural* view [17].
- *Hardware View*: This level has enough information for hardware engineers to develop both the device itself and/or the devices surrounding the device being modeled [17]. It may not have the fidelity of the RTL, but enough for the

hardware designer [17]. This level is also referred to as the *Micro-Architectural* view [17].

TLM enables a representation of the hardware for software development at much faster rates than RTL modeling interfaced with software [17]. Modeling is used to reduce time to market in three principle areas: early embedded software development, performance analysis and functional verification [17]. It is desirable to have a C model which incorporates the hardware view but with more detail than a TLM. However, one modeling aspect of a TLM, which this thesis does not address, is system timing. The focus of this thesis is on functional detail rather than timing, and is intended to aid early embedded software development and functional verification with the addition of more enhanced debugging capabilities.

Cheema and Hammami also pointed out, in 2006, the usefulness of TLM. They identified the following benefits:

- Faster system design space exploration [20].
- Easier system modeling [20].
- Faster simulation speeds [20].
- Early start of software development [20].
- Avoidance of late bugs in a project [20].
- Hence: Shorter Times to Market for a product [20].

A C model addressing the benefits identified here is required. Additionally, quicker debug can be achieved with more detailed modeling of the internal structure.

In 2003, Kim, et al. described a verification scheme in the implementation of an MPEG-4 Video Codec in which C modeling techniques were used [21]. Their

verification environment included both HDL and C test bench models [21]. In 2000, S. M. Park, et al. used test bench models which including HDL-models and C-models to verify a Video/Audio Codec [21] [22]. The intent of the C models in these environments was to generate test vectors for comparison of actual hardware output [21] [22]. There was no documentation of utilizing the C models for early firmware development. Additionally, there was no indication that the models were detailed enough for faster debugging of issues. A C model is needed that can be used in early firmware development. Furthermore, intricate detail of the implemented HDL is necessary within the C model for faster debug of issues.

Bennour, Abid and Tourki presented the benefits of hardware-software co-verification in the simulation and emulation environments [23]. For register level interaction, it is assumed that either a processor model is available for simulation, or hardware has been implemented for emulation [23]. For high level interaction, a high level hardware model, possibly C, is assumed [23]. The intent is to shorten the development cycle by getting earlier software-hardware interaction [23]. A detailed C model for early firmware development, but at the register interaction level, is desirable to reduce the development cycle. This would alleviate the need for a hardware simulator during early firmware development, thus eliminating an additional learning curve for a firmware engineer.

Bergeron reviewed verification techniques in the industry in 2000 and released "Writing Testbenches: Functional Verification of HDL Models". In his second edition released in 2003, Bergeron pointed out typical design-for-verification techniques that include well-defined interfaces, clear separation of functions in relatively independent

units, providing additional software-accessible registers to control and observe internal locations, and providing programmable multiplexers to isolate or bypass functional units [24]. It is agreed that implemented HDL should incorporate software-accessible registers for observation points. Additionally, an intricately designed C model containing detail to work in concert with the HDL logic for debug comparison between variables and internal signaling is desired. Bergeron also introduced the method of designing a self-checking testbench where stimulus is applied and the results are verified by abstracting the physical-level transactions into high-level procedures using bus-functional models [24]. A C model capable of translating input requirements into high-level transactions for bus model interpretation is ideal. This would lead to constrained random verification and platform independence.

In 2006, Bombieri, Fummi and Pravadelli attempted to capture the state of the industry with regards to hardware design and simulation for verification [25]. In this document they described a transactor based system that provided stimulus to an RTL design through control and data inputs and acquired the results through control and data outputs [25]. The results would be checked through a result checker, and if successful, the next transaction would be initiated [25]. Figure 2.4 represents a pictorial view of the role of the transactor in testbench verification [25]. Once again a C model that can be used to generate the simple read and write transactions for control, as well as the input stimulus for data, is desirable. Additionally, generating expected results would be required for comparison in a checker.

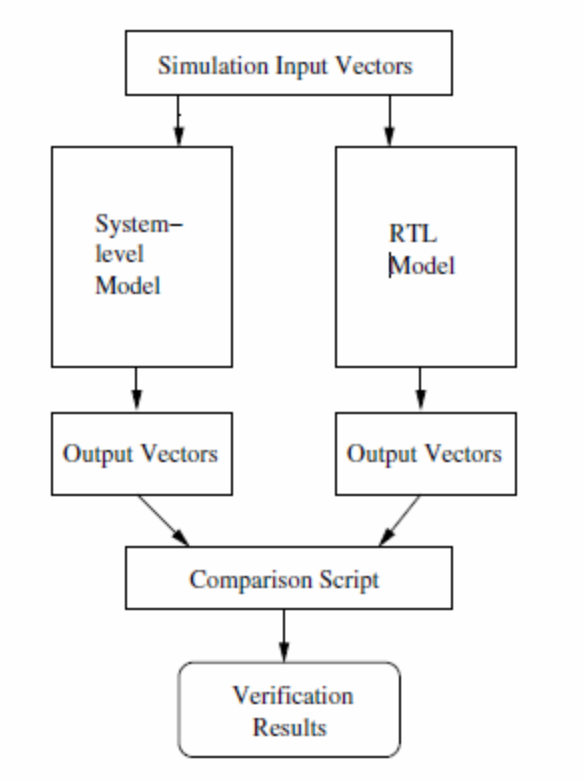

Figure 2.1: Simulation-based RTL Verification Flow Using System-level Models [10].

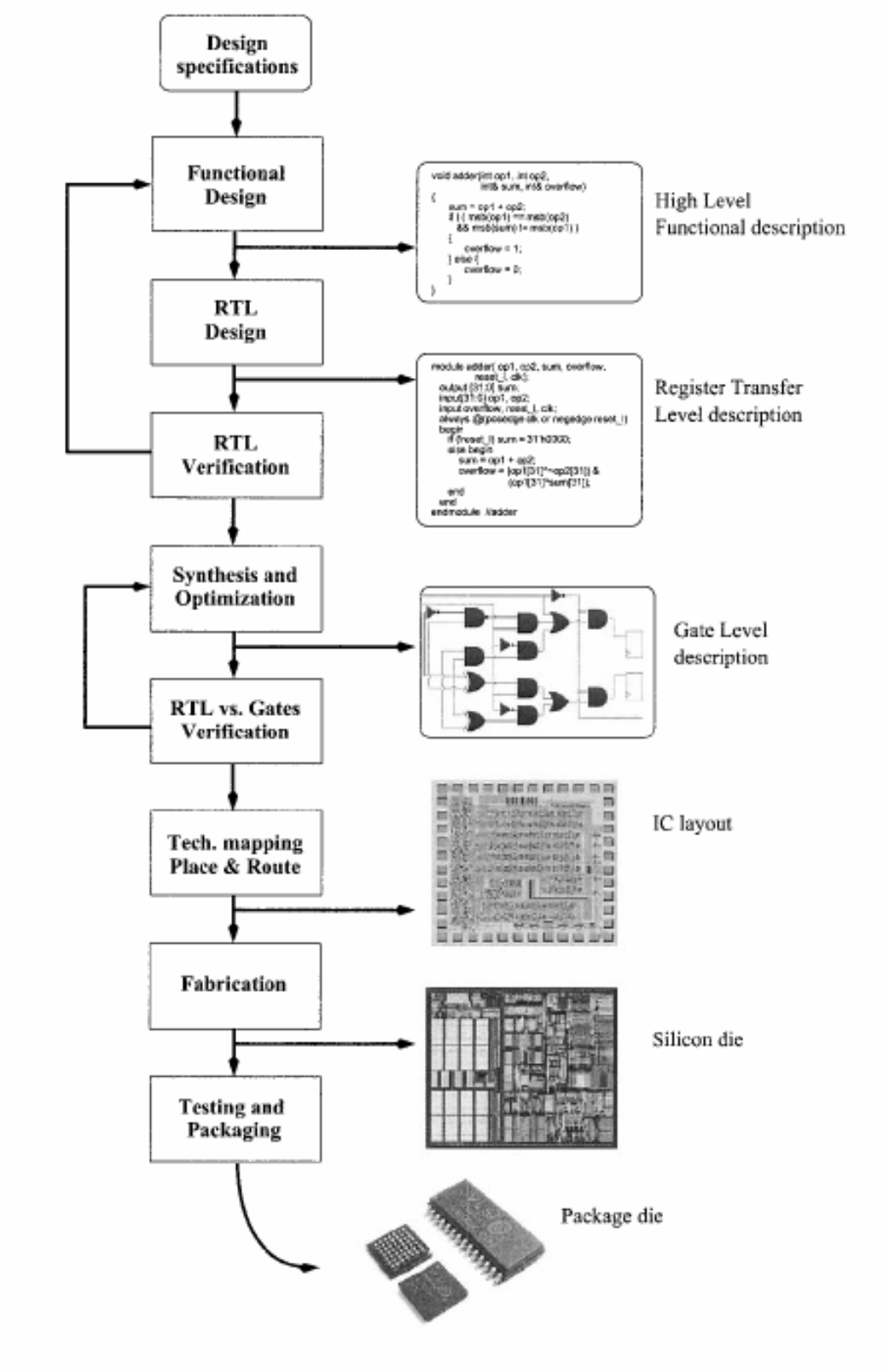

Figure 2.2: Conceptual Design Flow of a Digital System [12].

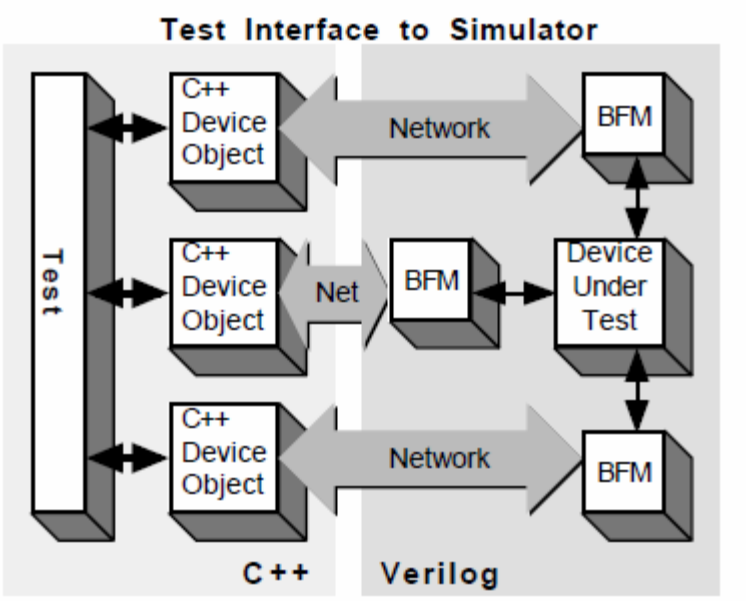

Figure 2.3: Simplified Model of the Object Based Interface [14].

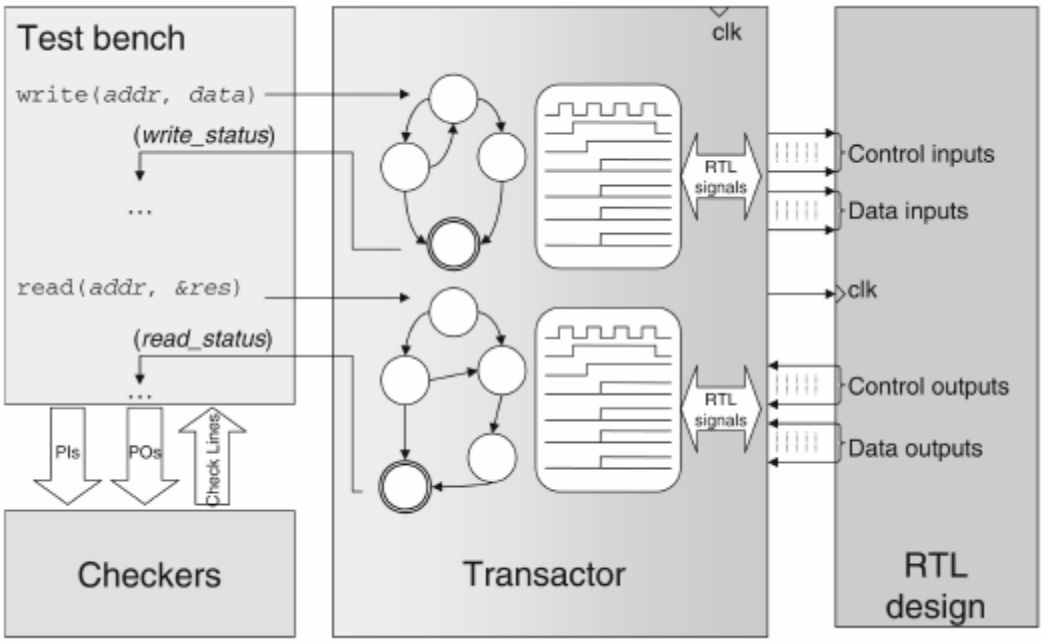

Figure 2.4: The Role of the Transactor in Testbench Verification [25].

Chapter 3

PROBLEM STATEMENT

This thesis intends to coordinate the ideals presented thus far and enhance them to develop a more complete verification and debug strategy. The research for this thesis began in April 2001 with the first version of the C model being completed in May 2001. The C model, including all aspects of this thesis, was completed in October 2002.

Over time, requirements for embedded applications have increased in function and complexity. In particular, imaging needs are increasing in complexity and magnitude. While some imaging applications can and have been implemented within an embedded microprocessor, it is advantageous to implement high performance functions within dedicated integrated circuits. This could be accomplished using a Hardware Description Language (HDL) targeted for an Application Specific Integrated Circuit (ASIC), Application Specific Standard Product (ASSP) or Field Programmable Gate Array (FPGA). Considering the complexity required for imaging applications, a more complex verification system is required.

Though modeling techniques are not new to the industry, the detail, portability, debug ability, and level of usefulness has been debated for some time. This thesis outlines a verification technique utilizing a C model that incorporates several important aspects of verification. First, an accurate C model is imperative to represent, at the bit level, the requested algorithms and modes of a given image processing task. This allows for early validation of the intended function before its availability. Kim, et al. recognized the need for comparison C models when they implemented verification for an MPEG-4

Video Codec in 2003 [21]. S.M. Park also recognized this verification need for a Video/Audio Codec [22] in 2000. It is unclear how well matched the detail of these models is to the hardware under verification.

Secondly, a C model that contains variables matching internal registers and signals within the intended HDL is also desired. F. Wotawa presented a tool for fault localization but appeared to fall short on pinpointing root cause [1]. Wilcox understood the need for pinpointing errors faster given shorter design cycles and more complex designs [13]. Brier and Mitra presented the need for observation points for exposure and correlation in C models for pinpointing root cause [11]. Li and Nagarajan called out models that matched bit for bit the VHDL/Verilog RTL models that they created, as well as well placed probing nodes for comparison to facilitate early problem identification and resolution [15]. Well placed probing points in hardware coincident with internal signals, along with matching variable names for internal registers, would foster side-by-side comparison debugging between the expected output from the C model and the HDL, once implemented. The C model suggested by this thesis incorporates these ideals.

Thirdly, a C model incorporating input variables within a structure that represents control register function of the HDL once it is implemented would be advantageous. This will allow for firmware development without requiring the implemented HDL. Additionally, during system bring-up, this expected interaction allows for debug between the firmware function written to configure the implemented HDL control registers, or the C model itself. With earlier software development in mind, Bernstein, Burton and Ghenassia looked to TLM with the intent of bridging the abstraction gap in system level modeling and design [17]. Additionally, Bennour, Abid and Tourki presented the

benefits of hardware-software co-verification in the simulation and emulation environments using a processor model in early verification, thus, enabling earlier software development [23].

Also, the implemented HDL should contain a cycle stepping function that will effectively pause processing between pixels. This allows for state and register retrieval between pixels for comparison to C model variables much like comparison debugging between the C model and HDL simulation. Enabling an easier method for pinpointing of issues addresses Wilcox concern in [13]. Bergeron identified software-accessible registers for observation points as key enablers to debugging [24].

Finally, a C model that is capable of parsing an input parameters file and generating some sort of test stimulus based on these parameters is considered necessary. This allows for self checking and less human interaction while developing multiple tests automatically with little effort. Additionally, portability of design test cases from one level of design to the next, as discussed by Bertacco in Figure 2.2 [12], should be addressed. Portability to future designs should also be considered. In 2008 Ng presented a verification flow in Figure 2.1 that incorporated generating stimulus and comparing results between RTL and a system level model [10]. Bergeron also introduced the method of designing a self-checking testbench to aid in better test generation [24]. In 2006, Bombieri, Fummi and Pravadelli pointed out the use of a transactor based system in Figure 2.4 that would stage transactions based on success [25].

Implementing a C model with these aspects will aid in early firmware development, multiple test case generation within an encapsulated test environment, system emulation debugging, simulation debugging, and the capability to prove out

algorithms before HDL implementation. Additionally, a constrained random test environment can be achieved. These aspects will be demonstrated in the following chapters.

Chapter 4

MODELING AND VERIFICATION OF ERROR DIFFUSION USING C MODEL

There are many different examples of imaging tasks or systems. These include: image filtering, image compression, error diffusion halftoning, and several others. For the purposes of this document, error diffusion halftoning will be considered.

Error diffusion halftoning effectively reduces the bit depth of an image while giving the appearance of a constant gradient of colors with respect to the human eye. In most applications, error diffusion halftoning is used to convert a multi-level image into a binary image. The quantization residual is distributed to neighboring pixels to be processed at a later time. Neighboring input into the error diffusion function for a specific pixel location can subtly change the binary value of the pixel being affected. It is virtually impossible to visually track the correct value of all pixels within a halftoned image.

To properly address the verification challenge for an error diffusion halftoning function implemented in an HDL, a C modeling approach was incorporated. The C model contained the following characteristics. First, the C model was implemented to be bit level accurate for the requested algorithms and modes. This was important to allow color science validation of the implemented HDL before its availability, as well as proving out new inventions or requests before implementation. Second, the C model contained variables that matched internal registers and signals within the HDL. This was also important for fostering side-by-side comparison debugging between the expected output from the C model and the actual HDL implementation. Third, the C model

incorporated input variables within a structure that represented the control register function within the HDL implementation. This allowed for firmware development without requiring the implemented HDL. Additionally, during system bring-up and debugging, the firmware can effectively switch between a function written to configure the implemented HDL control registers, or the C model itself, for comparison purposes. Also, included within the implemented HDL, is a cycle stepping function that will effectively pause processing between pixels. This allowed for state and register retrieval between pixels for comparison to C model variables, much like comparison debugging between the C model and HDL simulation. Finally, in addition to the function of producing the expected result, the C model was expanded to parse an input parameters file and generate a stimulus script, thus, enabling an encapsulated verification approach. Enabling an encapsulated verification approach reduces the reliance on human interaction and thus the potential for human error. Due to the intricate nature of a complete C model design, only portions of the C model and the implemented HDL will be discussed in this document to illustrate the aforementioned points.

The Floyd-Steinberg error diffusion method was used for the basis of the algorithm being implemented and tested. This method consists of applying a threshold to a running pixel value and distributing the quantized error to neighboring pixels. For further consideration, Figure 4.1 depicts a pixel location and its effect on neighboring pixels. Also depicted is a pixel and how it is affected by its neighbors. Several variations and enhancements have been implemented over time to improve the quality of binary image produced over that of the Floyd-Steinberg method. For instance, attempts have been made to randomize the threshold used. This technique is used to break up artifacts,

sometimes referred to as worms, to give a better distribution of dots. For the implemented HDL, a Linear Feedback Shift Register (LFSR) was implemented to augment a desired threshold, giving a pseudo-random effect. This complicates bit accurate modeling, because the value of an LFSR depends on the number of cycles it has been iterated. For the C model to be bit accurate at a given pixel location during processing, the same number of cycles through the LFSR will have to be duplicated. Consider the excerpt of code in Figure 4.2 that highlights the LFSR function within the C model. Additionally Figure 4.3 highlights cycle duplicating of this LFSR function.

Note that the RandGen function models the LFSR behavior within the implemented HDL at the gate level. RandGen was called multiple times before processing the image to ensure that the value returned by this function exactly matched the LFSR implemented within hardware for processing the first pixel. Additionally, the RandGen function was called after processing each line to accomplish alignment between lines. The RandGen function is called during processing as well, when the LFSR value is actually used. While this is a trivial matter, considerations similar to this had to be made throughout the C model to force an exact match of running values and results to the implemented HDL. Given an exact matched C model, a color science or algorithm development team working in parallel with an ASIC team can ensure the output of a product before having a fabricated ASIC.

The running pixel mentioned previously is an addition of several components consisting of the current pixel to be quantized, the quantized error of neighboring pixels, and any remainder leftover from the by 16 division. The C model excerpt in Figure 4.4

represents an implemented HDL for the running pixel calculation. Figure 4.5 contains actual implemented HDL for running pixel calculation.

While the variables and signals are not identically matched names, they are easily identifiable. This is important for direct comparison between a C debug window and a simulation wave window during debug of the system. The running pixel value, or Dotval (DOT_VALUE) as it is referred to here, can be thought of as the signature of the error diffusion system. The point at which Dotval does not match between the C model and the implemented HDL is usually the first place to investigate for an error. Note that differences in Dotval may not cause an immediate difference in the binary output image, depending on several factors, including the threshold. Therefore, comparing the resultant binary output images from the C model and implemented HDL is not a complete verification of the system. The error passed to the next line, a derivative of Dotval, is the best place to complete verification. Because of this, it is essential that the error passed between lines match between the C model and implemented HDL. As is indicated in the comments of the LFSR C code excerpt, the implemented HDL has several pipeline stages. Because of the nature of a pipelined design, each variable into the Dotval equation is staged in different ways. This had to be accounted for in the C model to ensure exact comparison.

For early firmware development, or for in system debugging, the C model utilizes a struct variable named DiffParams to give a common interface between the C model and register level firmware. This struct has variables within it that represent the control register function of the implemented HDL. By providing this struct to a firmware programmer, and the associated C model before an implemented HDL is available,

development can proceed very early. Additionally, when unexpected results arise while using the implemented HDL, the C model can easily be interposed to verify whether the issue lies within the algorithm or the implementation. Figure 4.6 presents the DiffParams struct C model excerpt. Figure 4.7 highlights two control registers of the LFSR implemented HDL firmware specification for comparison. The firmware specification excerpt is the description for interfacing to the LFSR function within the implemented HDL. Note that the variable names within struct ErrorDiffParams are easily identifiable with the specification. This is the case with all other variables of ErrorDiffParams and the corresponding HDL specification.

The next aspect of this verification technique lies solely within the implemented HDL. For hardware debugging, a stepping function was implemented to force an execution pause of the controlling state machine for the implemented HDL. Figure 4.8 is an excerpt from the main control register firmware specification which contains this function's appropriate bits. When bits 1 and 7 are asserted for the *Err Diff Control Register*, the logic is configured to execute to a certain point within the controlling state machine and wait for clearance to execute the next pixel. Once in the waiting (paused) condition, the logic can be configured to process the next pixel by asserting bits 1, 7, and 8. Figure 4.9 presents an excerpt from the implemented HDL controlling state machine that highlights a signal named WAIT_ST enabling execution. If WAIT_ST is asserted, then the controlling state machine cannot continue execution. However, once WAIT_ST is de-asserted, execution can continue. Figure 4.10 contains a portion of the implemented HDL for the WAIT_ST assignment. Along with other signals, STEP_FUNC_EN and not STEP_ED are included to assert WAIT_ST_INT under the right conditions.

STEP_FUNC_EN and STEP_ED are the register bits 7 and 8 indicated above in the *Err Diff Control Register* specification. When STEP_FUNC_EN is asserted, WAIT_ST_INT will be asserted as well, until STEP_ED is provided.

While waiting for an asserted STEP_ED signal, the system will remain stable. During this time, debug registers can be read to gather information about the current state of several registers and signals. For instance, in Figure 4.9 there is a signal named STATE_SIGS. This signal is accessible via addressable registers. Figure 4.11 contains the firmware specification outlining access of STATE_SIGS, its encoded values, and what they represent. As is indicated, several states of the controlling state machine are encoded and can be viewed during execution. If, for instance, the implemented HDL fails to finish execution, then the STATE_SIGS variable can be viewed to understand during what state execution failed. Note that LXK_THRESH_RAND can be observed through bits 27 down to 16 in the same register mapping identified here. This signal is a combination of a threshold lookup table and the same LFSR discussed earlier. During design of the implemented HDL, it was important to choose several observation points such as the LFSR, to aid debug of the system. By choosing a good range of observation points, one can avoid having to reconfigure an FPGA to route out more signals. This can greatly reduce development time, as FPGA synthesis can consume more than a day depending on complexity.

Thus far, the discussion has centered on an accurate imaging C model and the benefits it brings to proving out color science algorithms, firmware development, debugging implemented HDL, and verification. To aid in more robust and complete simulation and emulation verification techniques, subtle but powerful additions were

made to the C model. First, the C model was expanded to read in and parse a parameters file that contained information on register settings. Second, the C model was expanded to include the capability for creating a simple scripting language with basic read and write instructions to memory locations targeting the DUT and system memory, as well as a waiting instruction for system feedback. These simple reads and writes were used for writing DUT registers, loading main memory with a desired image, and dumping main memory to compare an image processed by the DUT with an expected result after execution. Finally, to complete the system, a bus stimulus model was created to interpret the simple instructions of read, write, wait, and compare. The read and write instructions were to be applied to the system bus using the bus stimulus model. These additions allowed for an encapsulated verification environment where multiple simulations could be generated and verified, without the traditional approach of wave investigation. Furthermore, less human interaction was needed once an accurate encapsulated C model environment was incorporated.

Consider Figure 4.12, which depicts a verification system utilizing an accurate C model for comparison with implemented HDL or DUT. From this, it can be seen that the same parameters are used to stimulate both the implemented HDL and accurate C model algorithm. The C model parsing function reads in the parameters and applies them to the C model algorithm to generate expected results. Furthermore, the C model scripting function takes the parsed parameters and generates a unique stimulus script using the simple scripting language of reads, writes, wait, and compare.

See Figure 4.13 which presents an architecture utilizing a bus model attached to the internal system bus. In this architecture, the bus model acts as the microprocessor

and enacts read and write accesses on the internal system bus based on the unique script provided before simulation. As described above, this unique script is provided by the C model along with an expected resultant image. Figure 4.14 includes a sample of what a simple script looks like for this type of environment. The addresses provided in this sample are actual error diffusion hardware assist control registers within one implementation. In this script, the wrchk command is used to instruct the bus model to write the data in the second field to the address indicated in that of the first. After executing the write, the bus model reads back the address indicated to compare with what was written. The sense command instructs the bus model to check a signal bus going into the model for certain values. The first field indicates which bit of the bus to check. The second field indicates what value to look for. In this case, the bus model should check to see if bit 0 is asserted. Finally, the check command is used to read large amounts of data from a provided address, and place the data within a file indicated within the first field. Additionally, the check command instructs the bus model to compare the data as it is read with the data from the file indicated by the second field. The third and fourth fields of check indicate the starting address and how many bus words to read. The basic function of this script is to enable the error diffusion logic, wait for execution, and then compare the results. What is not shown here is configuring direct memory access (DMA), enabling the error diffusion logic to read a given image from main memory for diffusion, and then write the result back into memory. This is easily extensible to this script.

It should be noted that this verification environment is independent of the microprocessor chosen for the system. Therefore, the verification environment and tests developed around it are directly portable to future designs given that a bus model capable

of accepting identical commands exists. This becomes a very powerful argument as function requirements for higher end image processing chips continue to grow beyond resources to support them.

In summary, verification of image processing tasks within hardware has become a very complex and overwhelming task. Complex tools and techniques are required for success. Within this verification implementation, a C model exactly matching the algorithm desired and the implemented HDL of that algorithm was incorporated. Features of this C model included variables that matched internal registers and signals within implemented HDL, and variables within a structure that represented the control register function within the HDL implementation. Additionally, the implemented HDL contained a cycle stepping function that effectively paused processing between pixels. Finally, the C model was expanded to parse an input parameters file, generate a stimulus script, and generate an expected result to enable encapsulated verification.

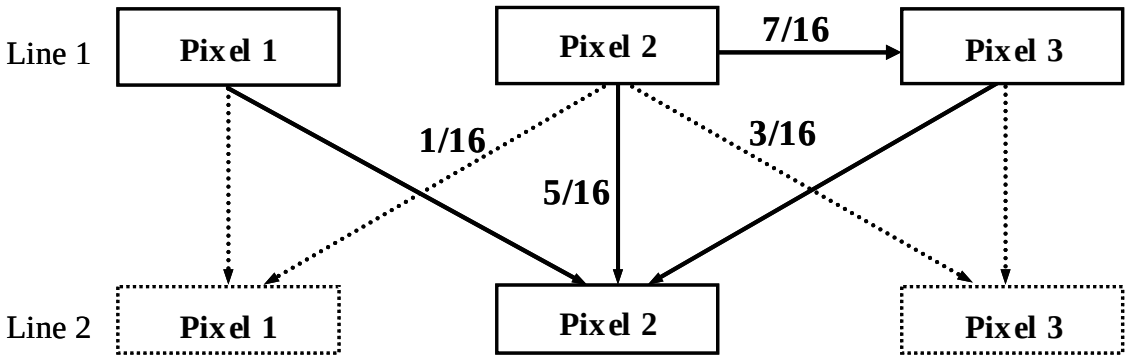

Figure 4.1: Effect of Error Spreading on Neighboring Pixels.

```
/**********Pseudo-Random Number Generator for Diffusion (LFSR)**************/
void RandGen (ErrorDiffParams *DiffParams)
{
        /*RandNum is a global variable*/
        short int   X1_in1, X1_in2, X2_in1, X2_in2, X3_in1, X3_in2;
        short int   X1, X2, X3;
        unsigned short int BitSel0, BitSel1, BitSel2, BitSel3;
        unsigned short int EnableXor0, EnableXor1, EnableXor2, EnableXor3;
        unsigned char ReSeed, Feedback;

        /*Peel off the register settings*/
        BitSel0 = DiffParams->RandomBitSel0;
        BitSel1 = DiffParams->RandomBitSel1;
        BitSel2 = DiffParams->RandomBitSel2;
        BitSel3 = DiffParams->RandomBitSel3;
        EnableXor0 = DiffParams->RandomEnableXor0;
        EnableXor1 = DiffParams->RandomEnableXor1;
        EnableXor2 = DiffParams->RandomEnableXor2;
        EnableXor3 = DiffParams->RandomEnableXor3;
        ReSeed = DiffParams->Seed;
        Feedback = DiffParams->RandomFeedback;

        if (ReSeed)
        {
                RandNum = DiffParams->SeedVal;
        }
        else
        {  /*If not a ReSeed then generate a new random value.*/
                /*Get the first bit of the first xor.*/
                X1_in1 = (RandNum >> BitSel0)&0x1&EnableXor0;
                /*Get the second bit of the first xor.*/
                X1_in2 = (RandNum >> BitSel1)&0x1&EnableXor1;
                X1 = X1_in1^X1_in2;

                /*Get the first bit of the second xor.*/
                X2_in1 = X1;
                /*Get the second bit of the second xor.*/
                X2_in2 = (RandNum >> BitSel2)&0x1&EnableXor2;
                X2 = X2_in1^X2_in2;

                /*Get the first bit of the third xor.*/
                X3_in1 = X2;
                /*Get the second bit of the third xor.*/
                X3_in2 = (RandNum >> BitSel3)&0x1&EnableXor3;
                X3 = X3_in1^X3_in2;
                RandNum = (((RandNum>>1)&0x7FFF) & (~(0x1<<Feedback))) | ((X3)<<Feedback);
            /*Shift RandNum right by one with the top bit being the xor.*/
        }
}
```

Figure 4.2: C Model Excerpt of LFSR Function.

```
/* This starts the Error-Diffusion part. */
while (LineNum < DiffParams.NumLinesToProcess)
{
        /*Line the LFSR up with the HW at the beginning.*/
        if (DiffParams.Enable)
        {
                if (!LineNum)
                {
                        /*First Line. Time to reseed*/
                        RandGen(&DiffParams);
                }
                /*Pipeline Stage: SOURCE_IN_REG1*/
                RandGen(&DiffParams);
                /*Pipeline Stage: SOURCE_IN_REG2*/
                RandGen(&DiffParams);
                /*Pipeline Stage: SOURCE_IN_REG3*/
                RandGen(&DiffParams);
                /*Pipeline Stage: SOURCE_IN_REG4*/
                RandGen(&DiffParams);
                /*Pipeline Stage: SOURCE_IN_REG5*/
                RandGen(&DiffParams);
                if (!DiffParams.SpreadPassEnable)
                {
                        /*Two extra stages are needed in the pipeline when the Spread
                           passing is not enabled*/
                        /*Pipeline Stage: SOURCE_IN_REG6*/
                        RandGen(&DiffParams);
                        /*Pipeline Stage: SOURCE_IN_REG7*/
                        RandGenA(&DiffParams);
                }/*End if DiffParams.SpreadPassEnable*/
        }/*End if DiffParams.Enable*/
                *
                *
                *
                *
                *
        if (DiffParams.Enable)
        {
                /*Line the LFSR up with the HW at the end of the line*/
                RandGen(&DiffParams);
        }
}/*End of LineNum loop*/
```

Figure 4.3: C Model Excerpt for Duplication of LFSR Function.

```
/* This is the most important equation in this file!!!
   Explanation:
   We want to take the dot value from the raster line and add in errors from the previous
   line and from the previous dot. Here is what everything means:
         ErrorFW - Error from the previous dot.
         SourceIn - Current raster byte.
         DiffParams.Shift – Weight control of source passed via DiffParams struct.
         Error.Rem - Remainder from division by 16 of previous Error.Whole.
         PrevLineErrorTotal - Total of ErrorDB, ErrorDN, and ErrorFW
            from the dots on the previous line that contribute to the current raster byte.
         DotVal - The adjusted raster byte with error added and multiplied by 16.
*/
DotVal = ErrorFW + (((SourceIn<<DiffParams.Shift)&0xFFF) + Error.Rem) + PrevLineErrorTotal;
```

Figure 4.4: C Model Excerpt of Implemented Running Pixel Calculation.

```
SYNC: process(ASIC_CLK, RESET_N)
begin
         if (RESET_N = '0') then
                  DOT_VALUE <= (others => '0');
         elsif (ASIC_CLK'event and ASIC_CLK = '1') then
                  if (CLEAR_ACC = '1') then
                           DOT_VALUE <= (others => '0');
                  elsif (ENABLE_CALC1 = '1') then -- Green light
                  -- stage 1 calculation
                  -- Add source, line-to-line error, remainder and forward error and latch
                  -- them to DOT_VALUE.
                  -- Extend the sign for the ERRORs.
                  --
                  -- DOT_VALUE is made up of 16 bits.  This allows 4 bits of remainder,
                  -- 10 bits of whole number, one bit for sign and one bit for overflow.
                           DOT_VALUE <=  ERROR_FW + ('0'&SOURCE_SHIFTED) +
                                         REMNDR_INT + ERROR_IN_INT;
                  end if;
         end if;
end process SYNC;
```

Figure 4.5: HDL Excerpt of Implemented Running Pixel Calculation.

```
#ifndef CHKDIFF_H
#define CHKDIFF_H

typedef struct
{
        /*Register variables*/
        Byte1Type       Enable;
        Byte1Type       InitDir;
        Byte2Type       NumLinesToProcess;
        Byte4Type       NumBytesToProcess;
        Byte1Type       BitsPerPix;
        Byte1Type       Shift;
        Byte1Type       NumRandomBits;
        Byte2Type       Seed;
        Byte2Type       InitSeed;
        Byte1Type       RandomBitSel0;
        Byte1Type       RandomBitSel1;
        Byte1Type       RandomBitSel2;
        Byte1Type       RandomBitSel3;
        Byte1Type       RandomEnableXor0;
        Byte1Type       RandomEnableXor1;
        Byte1Type       RandomEnableXor2;
        Byte1Type       RandomEnableXor3;
        Byte1Type       RandomFeedback;

        /*DMA variables*/
        Byte4Type       DMAErrBuffStart;        /*DMA Start of the Internal Error Buffer*/
        Byte4Type       DMAErrBuffEnd;          /*DMA End of the Internal Error Buffer*/
        Byte4Type       DMAPrintBuff;           /*DMA Start of Print Buffer*/
        Byte4Type       DMAMarkBuff;            /*DMA Start of Mark Buffer*/
        Byte4Type       DMAPrintIndex;          /*DMA Print Step Index*/

}ErrorDiffParams;
```

Figure 4.6: C Model Excerpt of DiffParamsStruct.

Err Diff Random Control Register **rb4ErrdiffRandCntrl** **0x1000138C**

This register allows firmware to control the pseudo-random number generator. Also, this register contains the selection bits for each feedback bit into the XOR logic. Up to four terms can be fed back through XOR's into the LFSR. An explanation of the LFSR and its settings occurs later in this document.

Bit	POR	Description	Operation
15:0	0	*Not Used*	Read Only
19:16	0	Random selection bits for term 1. Represents which one of 16 bits will be used from the pseudo-random number generator for first feedback bit.	Read/Write
23:20	1	Random selection bits for term 2. Represents which one of 16 bits will be used from the pseudo-random number generator for second feedback bit.	Read/Write
27:24	0	Random selection bits for term 3. Represents which one of 16 bits will be used from the pseudo-random number generator for third feedback bit.	Read/Write
31:28	0	Random selection bits for term 4. Represents which one of 16 bits will be used from the pseudo-random number generator for fourth feedback bit.	Read/Write

Err Diff Seed Register **rb4ErrdiffSeed** **0x10001390**

This register allows firmware to input a new seed into the pseudo-random number generator. When reading the Seed/LFSR bits of this register the current LFSR state is returned. Also, this register contains the disable bits of each feedback entry of the XOR's in the LFSR polynomial. When set to 1 the input into the XOR for the particular entry is set to 0. This essentially forces the XOR into pass-through mode. Finally, this register contains the feedback entry selection bits.

Bit	**POR**	**Description**	**Operation**
15:0	0	Seed/LFSR bits.	Read/Write
16	0	Disables Random feedback entry 1.	Read/Write
17	0	Disables Random feedback entry 2.	Read/Write
18	1	Disables Random feedback entry 3.	Read/Write
19	1	Disables Random feedback entry 4.	Read/Write
23:20	F	Random feedback entry selection bits. Represents which one of 16 bits will be the feedback point of the LFSR. For instance, if set to 4 then bit 4 of the LFSR will be where the feedback from the final XOR term enters the LFSR.	Read/Write
31:24	0	*Not used*	Read Only

Figure 4.7: Implemented HDL Firmware Specification Excerpt for LFSR Function.

Err Diff Control Register **rb4ErrdiffControl** **0x10001380**

This register holds the control bits for the Error Diffusion Hardware Assist.

Bit	POR	Description	Operation
0	*0*	*Left out for this illustration*	*Read Only*
1	0	Run operation bit. Writing a 0 tells the hardware to stop operation and return to an initial state. Writing a 1enables operation. This bit will be automatically cleared when the process completes.	Read/Write
6:0	*0*	*Left out for this illustration*	*Read Only*
7	0	This bit, when set to a 1, enables the Error Diffusion logic to run in the step mode.	Read/Write
8	0	By setting this bit to a 1 the Error Diffusion logic is told to step when in the step mode. This bit will be cleared automatically by the hardware.	Read/Write
31:9	*0*	*Left out for this illustration*	*Read Only*

Figure 4.8: Main Control Register Firmware Specification Excerpt.

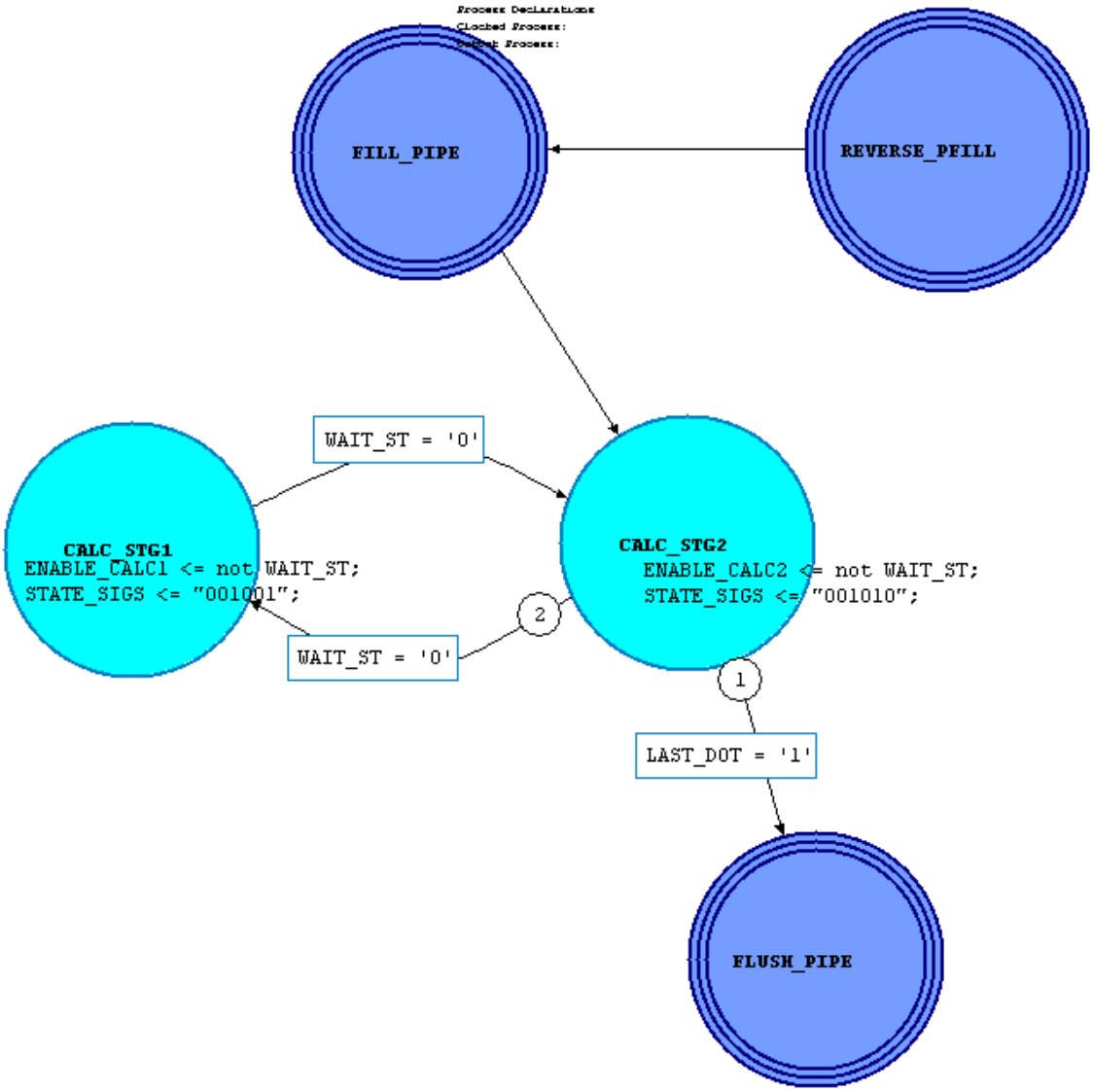

Figure 4.9: Portion of Controlling State Machine Driven by Signal WAIT_ST.

```
-- Generate the signal that moves the ERRDIFF_CTRL_SM into the wait state
WAIT_ST_INT <= (STEP_FUNC_EN and not STEP_ED) or
               PRINT_OUT_FULL or
               (SMBUFF_OUT_ENABLE and ERROR_OUT_FULL) or
               (SPREAD_PASS_EN_REG and PACKED_SPREAD_FULL) or
               (SMBUFF_IN_ENABLE and ERROR_IN_EMPTY) or
               not SOURCE_RDY;
WAIT_ST <= WAIT_ST_INT;
```

Figure 4.10: Excerpt of Implemented HDL for WAIT_ST Assignment.

Err Diff Dot Map Reg **rb4ErrdiffDotMap** **0x100013A0**

This register allows firmware to control the dots that will be produced when a given threshold is met. The STATE_SIGS_REG and LXK_THRESH are for debugging purposes only.

Bit	POR	Description	Operation
7:0	0	Dot map. Each pair of bits represents each combination using Lexmark and 2bpp thresholds. Bits 1:0 represent no dot placement and no threshold met. Bits 3:2 represent above the 2bpp-threshold only (low). Bits 5:4 represent above Lexmark threshold only (mid). Bits 7:6 represent above both thresholds (top).	Read/Write
13:8	0	STATE_SIGS_REG	Read Only
15:14	0	*Not used*	Read Only
27:16	0	LXK_THRESH_RAND observation. Output of the LXK threshold table plus the selected RAND_NUM.	Read Only
31:28	0	*Not used*	Read Only

State values:

SOURCE_IN_REG1:	000001
SOURCE_IN_REG2:	000010
SOURCE_IN_REG3:	000011
SOURCE_IN_REG4:	000100
SOURCE_IN_REG5:	000101
DOT_VALUE:	000110
SOURCE_IN_REG6:	000111
SOURCE_IN_REG7:	001000
CALC_STG1:	001001
CALC_STG2:	001010
FS1_STG2:	001011
FS2_STG2:	001100
FS3_STG2:	001101
FS4_STG2:	001110
FS5_STG2:	001111
FS6_STG2:	010000
FS7_STG2:	010001
FDV_STG2:	010010
WAIT_ON_DMA:	010101
WAIT_SPREAD_FIFO:	011100
FLUSH_LAST_ERROR1:	011111
END_OF_LINE:	100000
WAIT_ON_PRINT:	100001
WRITE_LAST_ERROR:	100100
WRITE_LAST_SHIFT:	100110

Figure 4.11: Firmware Specification Excerpt Showing STATE_SIGS Access.

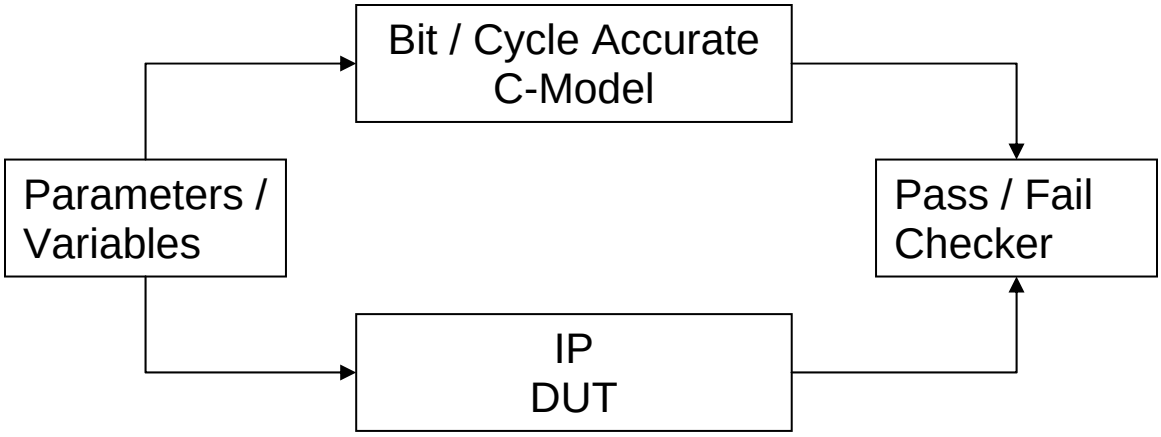

Figure 4.12: Encapsulated Verification System Utilizing an Accurate C Model.

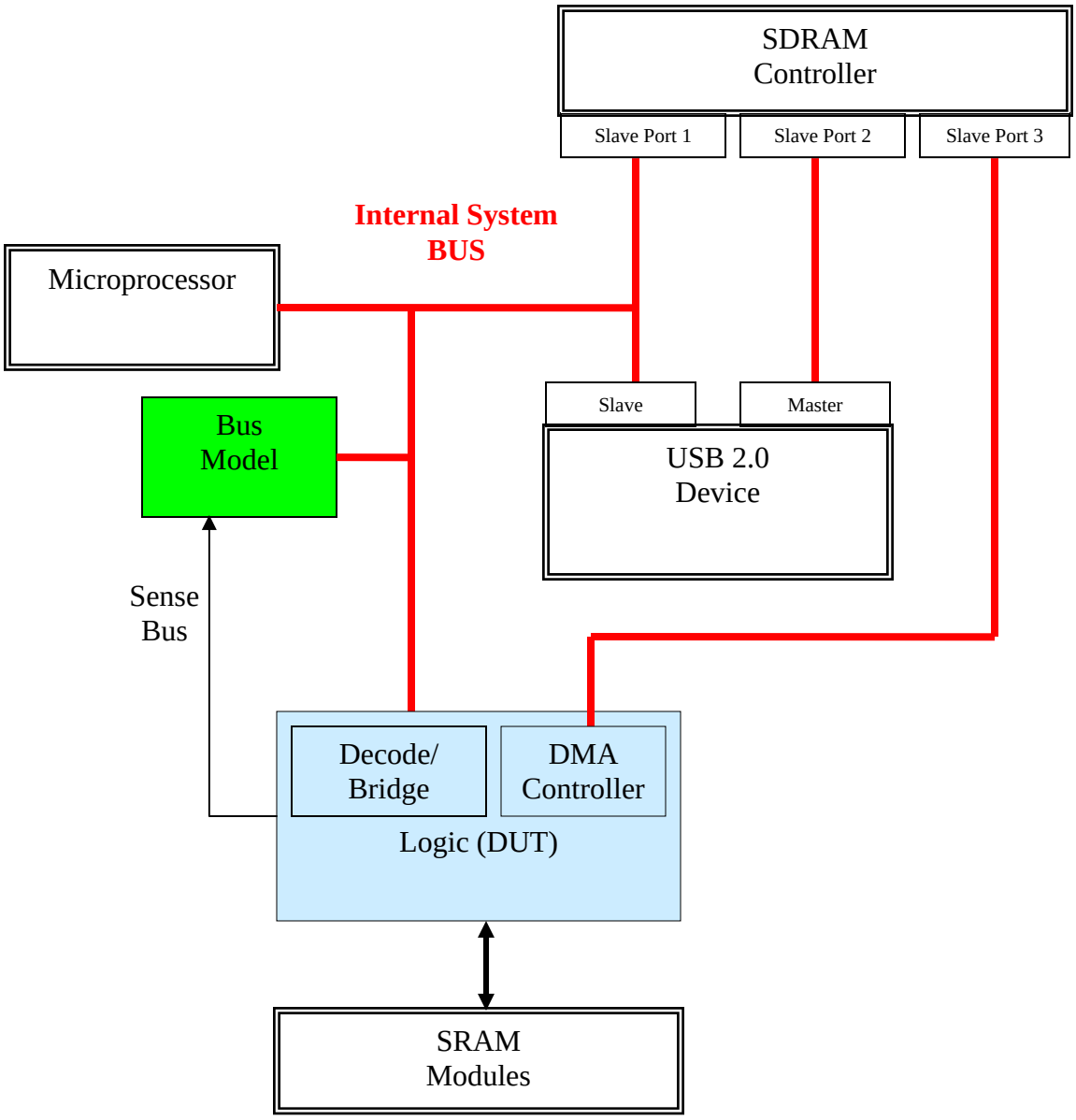

Figure 4.13: System Architecture Incorporating Bus Model.

```
-----Write ERRDIFF registers----
wrchk 10001384 00010010                 --ErrDiff Bytes/Lines Register
wrchk 1000138C 0010E853                 --ED Random Control Register
wrchk 10001390 00001711                 --ED Seed Register
wrchk 12001394 0000637F                 --ED Top Threshold register
wrchk 12001398 00007F7F                 --ED Mid Threshold register
wrchk 1200139C 00007F7F                 --ED Low Threshold register
wrchk 12001380 0003D63A                 --ED Control Register
--------------------------------
-----Wait for the interrupt-----
sense 01 01
--------------------------------
-------Clear the interrupt------
wrchk 12001128 00600020                 --ED_A interrupt
--------------------------------
----Now read back the output----
check cyerrout cyerrin 00000400 40      --Store the cyan errors while checking them
check cydotsout cydotsin 00010000 1430  --Store the cyan print data while checking it
```

Figure 4.14: Sample Simple Script for Bus Model Stimulus.

Chapter 5

USING THE ERROR DIFFUSION C MODEL

To better understand the importance of having an accurate C model, let us consider the LFSR previously discussed. The LFSR designed in this implementation of error diffusion is designed to break up unpleasant patterns that may result from distributing quantized error to neighboring pixels. It is intended to randomize the threshold step of the system. One could argue that the value of an LFSR at any given time is not important and does not need to be predicted. Regardless of the LFSR value, randomization should be achieved assuming the designed full cycle range is targeted. However, please consider the following two images. Figure 5.1 represents a diffused image using a C model that exactly matches the implemented HDL. Figure 5.2 represents a diffused image using an altered version of the C model. The altered C model contains one less function call to RandGen, thus, misaligning the LFSR cycle of the C model to that of the implemented HDL. Note that the two images appear very similar and are acceptable. Now consider Figures 5.3 and 5.4. Figure 5.3 is the same as Figure 5.1, but focusing only on the left corner of the image. Thus, Figure 5.3 is a focused zoom on the left corner of the image generated using a C model that exactly matches the implemented HDL. Similarly, Figure 5.4 is a focused zoom on the left corner of the image generated using the altered C model. Note that the two images are not the same and would be very hard to detect without focusing on a unique area of pixels. Without knowing what the intended result was, or having an accurate C model to point out this error, it would be virtually impossible to find. Additionally, without an exact C model it

would be hard to predict results of marginal algorithm improvements over model implementation deficiencies.

To take this example a step further, consider the debug steps required when an image difference such as this occurs. Initially, it could be detected just by simple comparison. Unfortunately, this would not be enough to point out the root cause. With an accurate C model, a simulation can be generated and compared by stepping through the C code using a debugger. Now consider Figure 5.5 which represents a screen capture of a code debugger alongside a simulation wave window for variable and signal comparison. Knowing that the failure occurs during initial processing, the focus is on the first pixel. When the first DotVal calculation is made, both the simulation and C model match with a value of 0x00FC, however, there is a difference between rand_no_reg of the implemented HDL and RandNum of the C model during this cycle. The two should be aligned one cycle before DotVal. This is the case because rand_no_reg actually is added to other signals and captured by the threshold register on the next cycle to align with DotVal in the pipeline. Notice that RandNum has a value of 0x7171 while rand_no_reg has 0xB8B8. While RandNum and rand_no_reg only disagree by one cycle, and this difference did not cause an immediate problem, it will force an issue with DotVal at some later cycle. Figure 5.6 presents the code debugger state along with the simulation wave window after taking four steps through the model. Note that DotVal still matches with a value of 0x0238, and that rand_no_reg and RandNum are still misaligned with values of 0x8B8B and 0x1717 respectfully. The first DotVal misalignment occurs after the next step and is shown in figure 5.7. Here, the C model expects DotVal to be 0xFA48, while the implemented HDL is actually 0x0240. Note the value of PrintByte does not match

between the two as well. PrintByte represents the diffused dots as they are created. The C model placed a dot at pixel location 5, which explains a negative DotVal of 0xFA48. Surrounding pixels must be negatively affected by the dot placement. Because the LFSR is applied to the threshold logic, the error manifested itself as incorrect predicted dot placement versus actual dot placement. After determining that the LFSR cycle appears to be misaligned, additional RandNum function calls can be added to the C model. Figure 5.8 illustrates correct alignment at the first DotVal calculation or first pixel. Both systems match for DotVal and LFSR with 0x00FC and 0xB8B8 respectfully. Fortunately, in this example the incorrect LFSR cycle would not be considered an HDL implementation issue. In fact, given the nature of an LFSR, its current value is irrelevant. However, the C model would need to be corrected to match exactly the implemented HDL for accurate prediction.

A similar approach could be taken when debugging the actual hardware. Assume for demonstration purposes that the LFSR cycle error was implemented in the C model and had escaped HDL verification. The expected result would be that displayed in Figures 5.2 or 5.4 and the actual hardware result would be that displayed in Figures 5.1 or 5.3. Focus would quickly turn to processing the first few pixels. Utilizing the stepping function provided within the actual hardware, the system could be configured to stop processing just before the first pixel. At that point, the current LFSR value could be read and compared to that of the predicted result. Once identified as an issue, the scenario could be recreated in simulation to point out that the LFSR C model design was missing an extra access.

While the above LFSR examples were fabricated and would not necessarily point out a hardware design issue, they do point out a need for accurate modeling techniques. It could be that some simple design flaw existed in how to implement binary overflow conditions. Something like this would not happen often in images with light color, but could be problematic in those with dark shades. Figure 5.9 illustrates an image diffused with approximately 2% error overflow accruing. Figure 5.10 illustrates the same image diffused using better techniques to eliminate error overflow. As illustrated in the first few lines of the image in Figure 5.9, overflow within error can cause quantization loss and unsightly diffusion issues. By utilizing an accurate C model, running error could be checked between lines and debugged similarly to that of the LFSR example above.

Given that an exact C model has been developed, proving out new functions or enhancements can become achievable. For instance, consider United States Patent #7486834, System and Method for Dynamically Shifting Error Diffusion Data [26]. This patent entails compressing diffused error that passes between lines during processing. The passing of error between lines can be cumbersome in two ways. First, it may burden the system bandwidth while passing error into and out of main memory. Second, an internal SRAM is used to optimize system bandwidth. This technique can be large and expensive in area, depending on the number of colors being processed simultaneously, the width of the image, and the resolution of passed error. Before, it was stated that diffused error can be viewed as the system signature, and it is vital that its integrity is maintained. Any failure to accomplish this will result in a poor diffusion system.

Therefore, something as intrusive as error compression had to be proven. Additionally, the results had to be accepted by a panel of engineers skilled in the art of

color science and image manipulation. Using a C model that accurately represented what the implementation would be easily enabled the development of print samples for approval. Several print samples were generated using uncompressed error and two different variations of compression techniques. Given an objective survey between the different images, it was proven that the error compression technique in question was acceptable. This allowed for savings in both SRAM area and system bandwidth when required. Once the C model was proven, the implemented HDL was designed to produce identical and acceptable output.

In all cases discussed thus far that implied simulation, an encapsulated verification method was utilized. As was discussed before, the C model expects a parameters file to parse, which dictates the intended operation. Additionally, the C model expects an input file on which to execute the operation. Based on these two variables, a stimulus script for a targeted bus model will be generated, as well as the expected resultant image from the operation requested. The mode of operation to generate a new test case using this method is to change the parameters file and input image. Multiple parameters files and images were generated to test different problematic areas. For instance, the parameters file indicates LFSR setup information. By targeting and changing this information for different images, the LFSR function can be more completely verified. Consider Figure 5.11, which represents an excerpt from an actual parameters file that could be used for generating a test case. As it can be seen, the parameters did, in some cases, represent the register settings very closely. This gave tighter control between parameter variance and control register effect. As will be

explained later, this type of verification lends itself nicely to constrained random verification.

In summary, the importance of an accurate C model at the processing cycle level is proven when considering several examples. As presented here, a simple cycle misalignment with a predicted LFSR function caused subtle differences within the image. While it was subtle and probably acceptable, it would make system debug impossible without first correcting the C model. During the debug phase, it is vital that variables within the C model accurately represent signals and registers of the implemented HDL. Without this, correlation to root cause becomes a guessing exercise. Assuming an accurate C model is provided, new features or enhancements can be proven through presentation of expectations to experts in color science. Finally, by implementing the encapsulated simulation environment, multiple test cases can be generated by changing the input parameters file and input image with less reliance on human interaction.

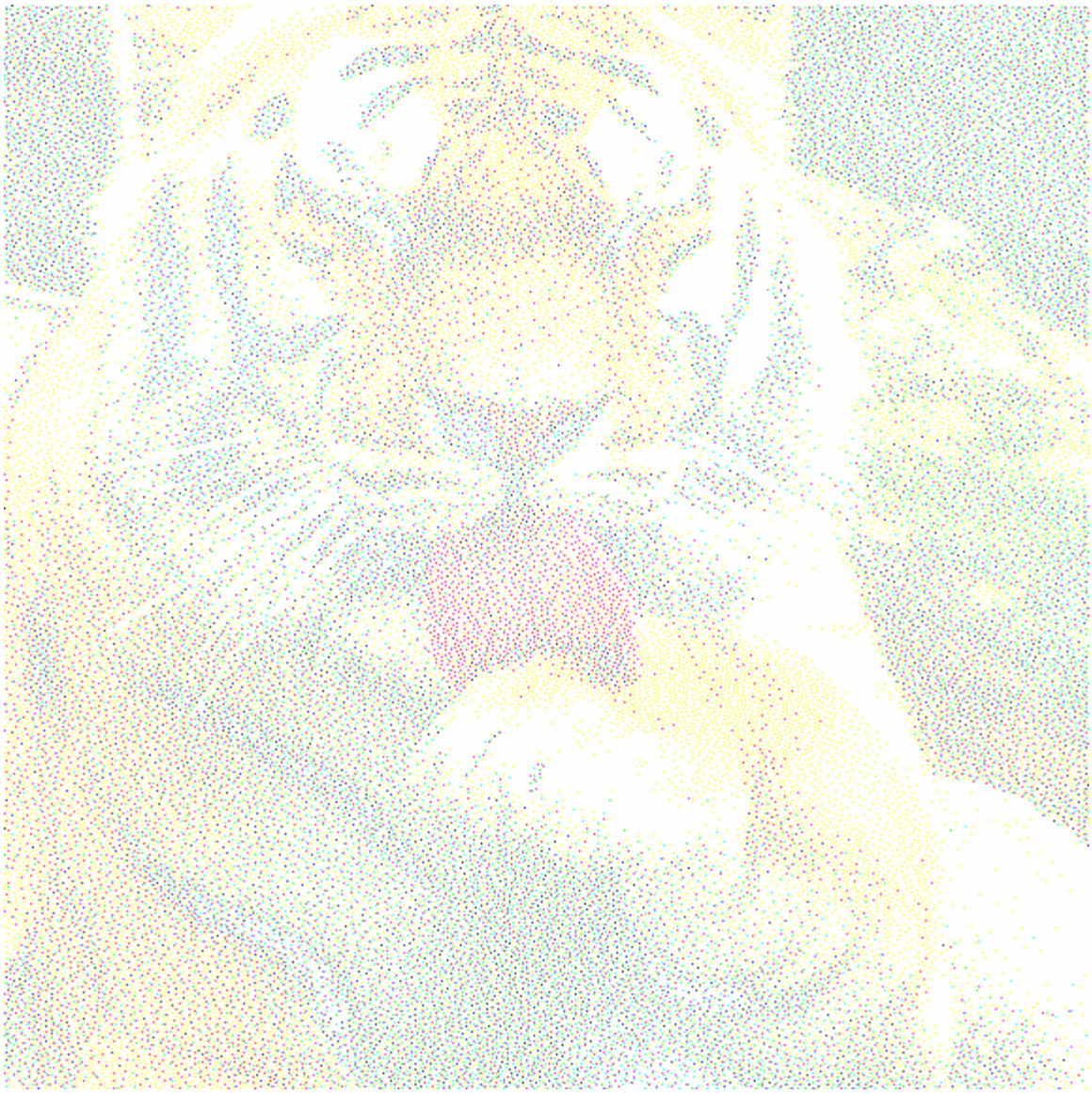

Figure 5.1: Diffused Image Using Accurate C Model.

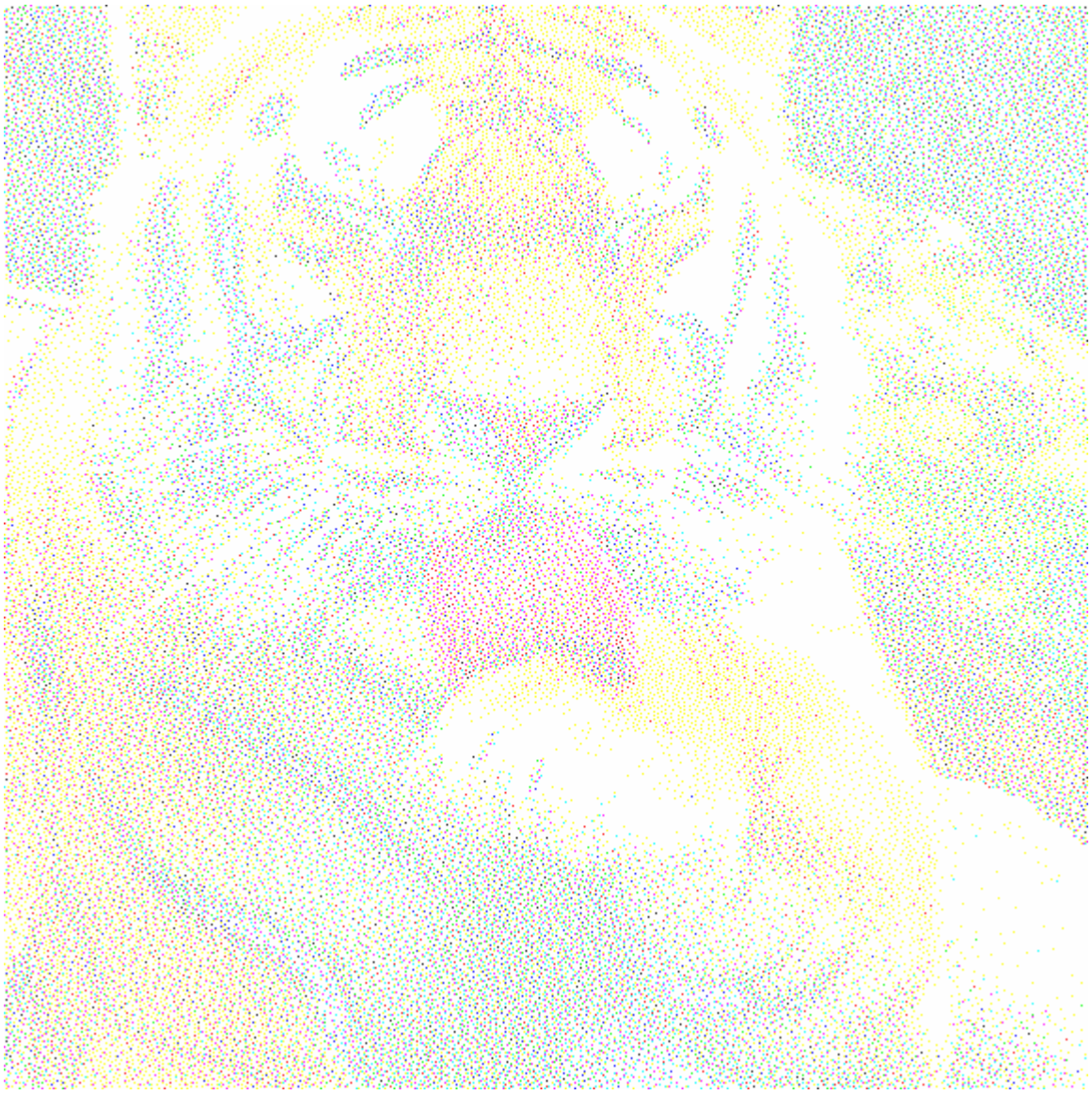

Figure 5.2: Diffused Image Using Altered C Model.

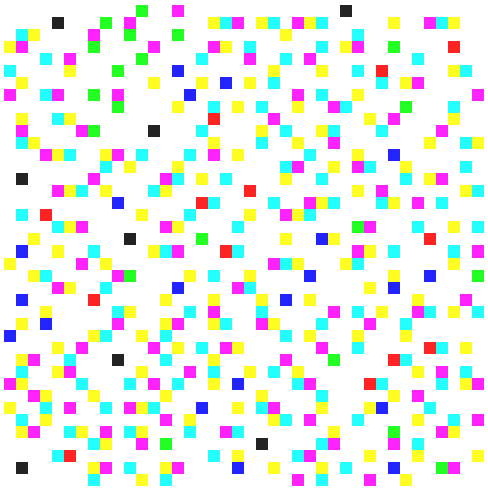

Figure 5.3: Diffused Image Using Accurate C Model.

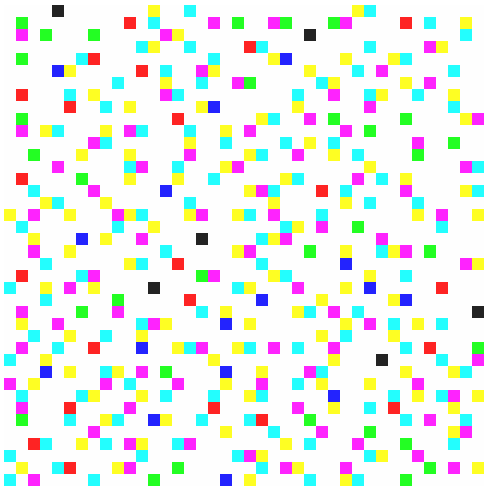

Figure 5.4: Diffused Image Using Altered C Model.

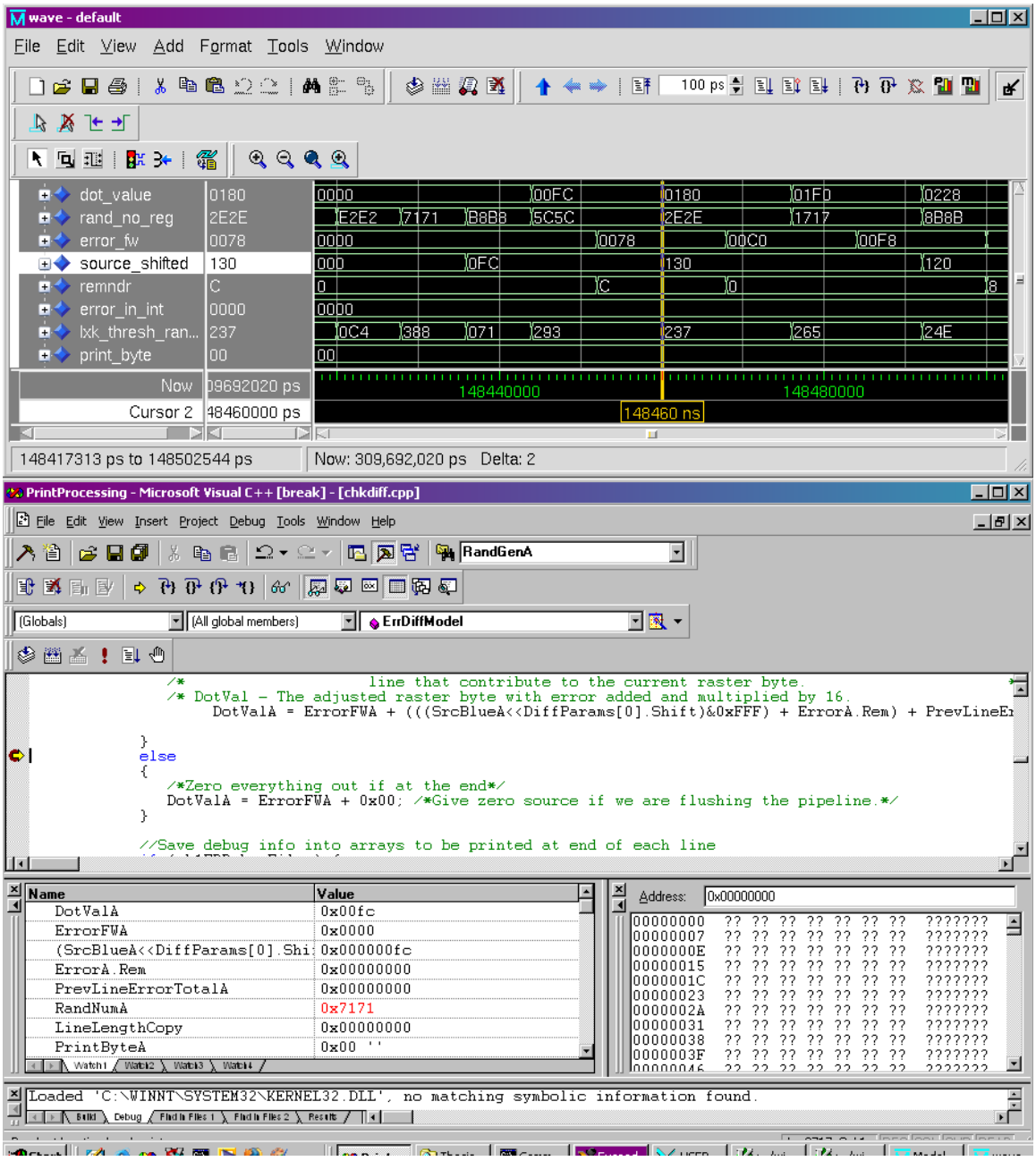

Figure 5.5: Simulation Wave vs. C Model Debugger at First Pixel.

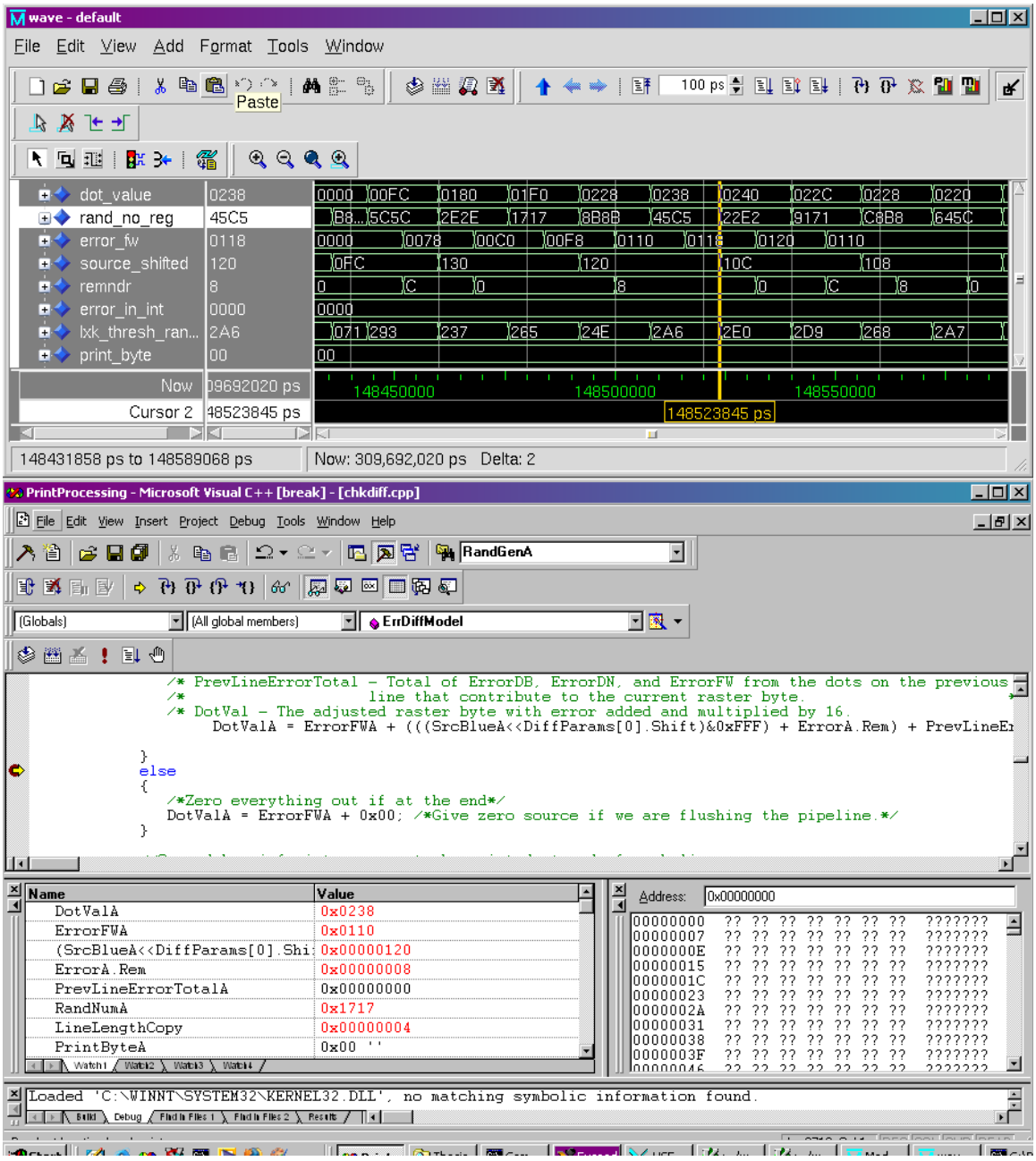

Figure 5.6: Simulation Wave vs. C Model Debugger at Fourth Pixel.

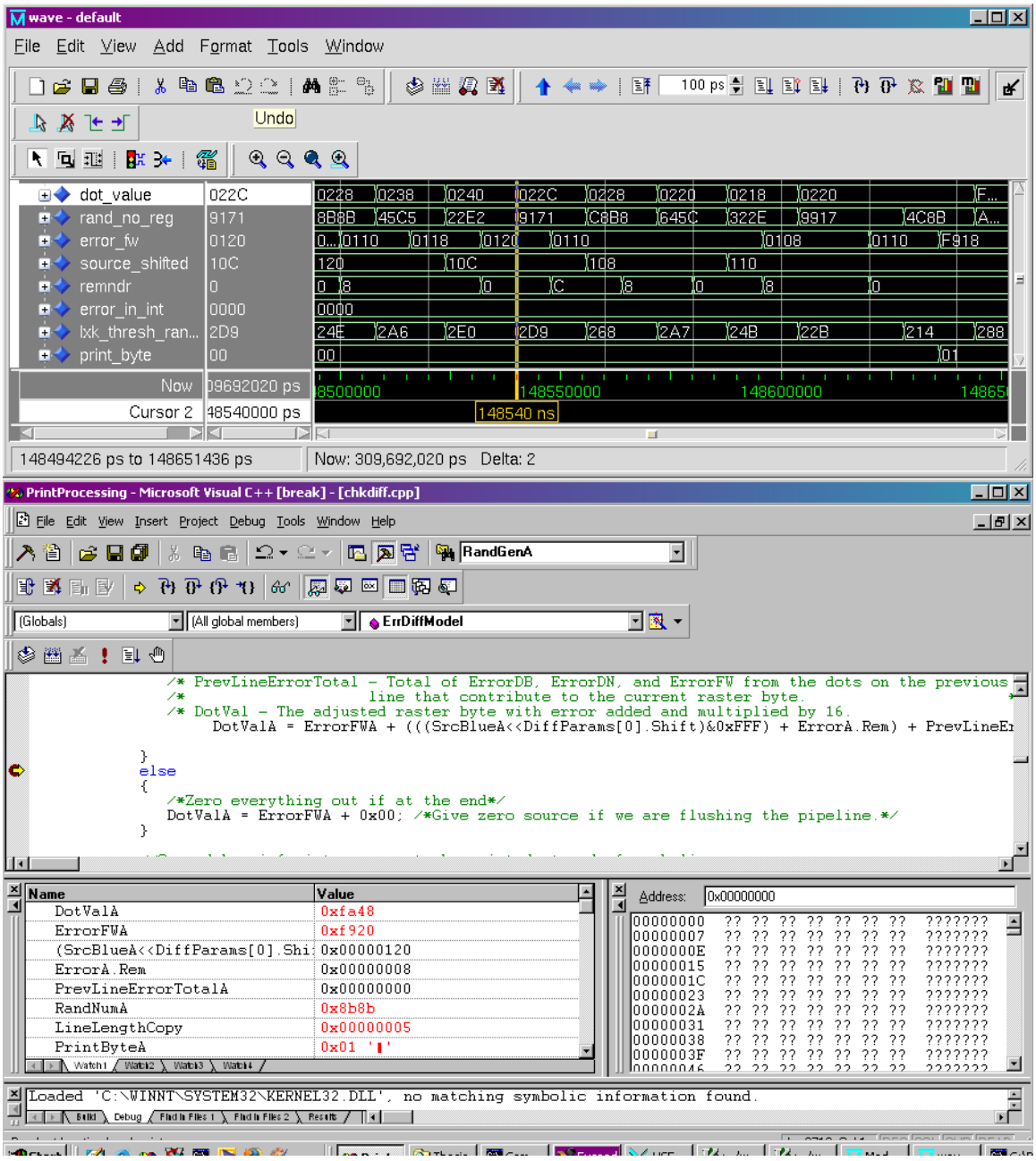

Figure 5.7: Simulation Wave vs. C Model Debugger at Fifth Pixel.

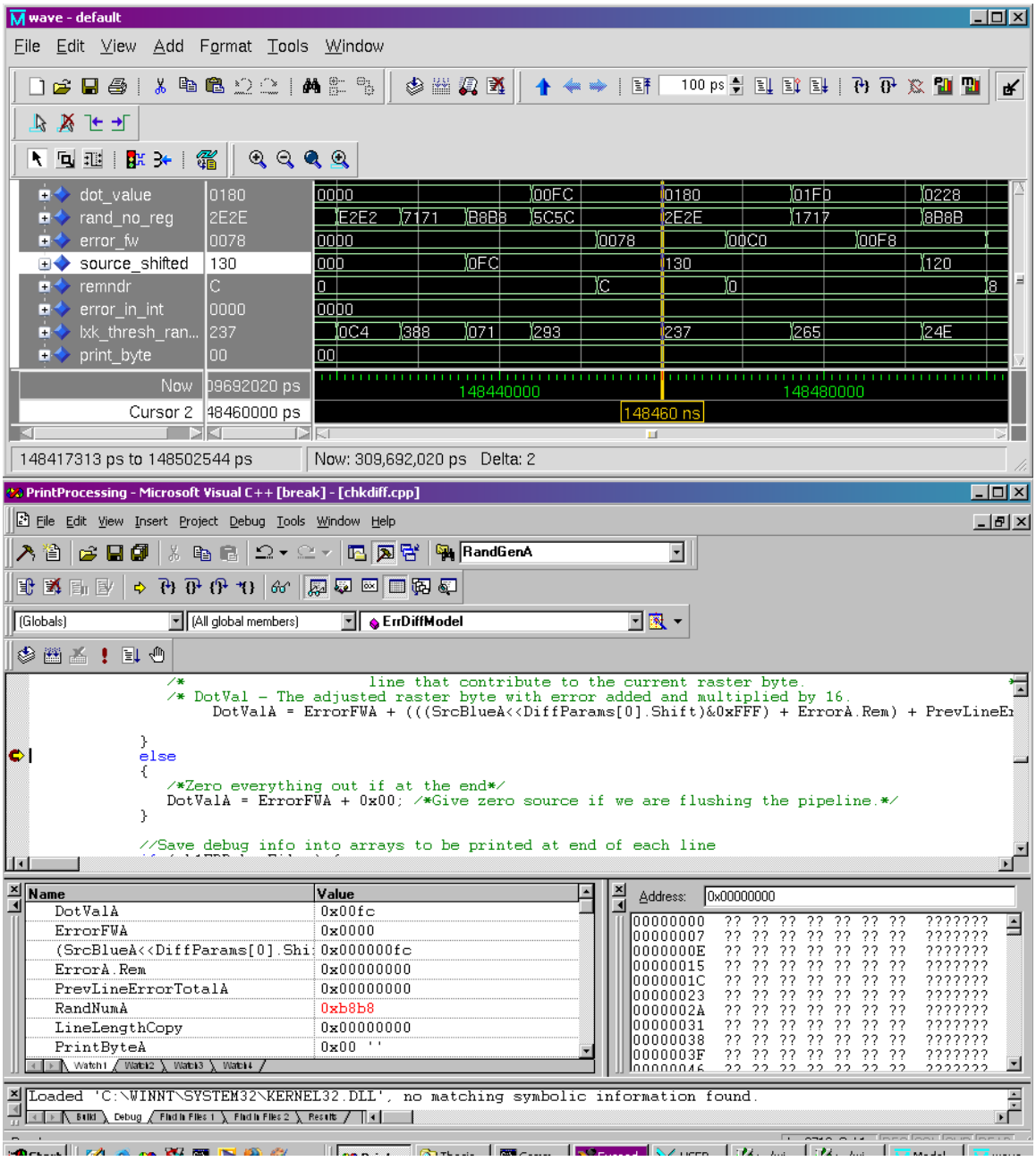

Figure 5.8: Simulation Wave vs. C Model Debugger with Corrected LFSR at First Pixel.

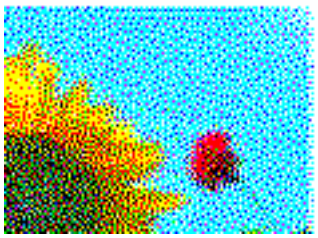

Figure 5.9: Image Containing ~2% Error Overflow.

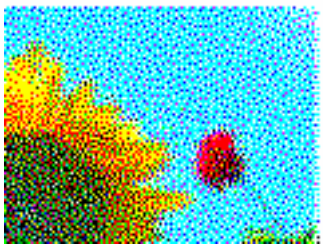

Figure 5.10: Image Diffused With No Error Overflow.

```
--Psuedo-Random Number Generator Seed (default=0x5905)
RandomSeed = 0000
--Selects first bit that goes into first XOR (default=0)
RandomBitSel0 = 0
--Selects second bit that goes into first XOR (default=11)
RandomBitSel1 = 11
--Selects second bit that goes into second XOR (first bit is output of first XOR) (default=13)
RandomBitSel2 = 13
--Selects second bit that goes into third XOR (first bit is output of second XOR) (default=14)
RandomBitSel3 = 14
--1=Enable first bit of first XOR, 0=Force 0 into first bit of first XOR (default=1)
RandomEnableXor0 = 1
--1=Enable second bit of first XOR, 0=Force 0 into second bit of first XOR (default=1)
RandomEnableXor1 = 1
--1=Enable second bit of second XOR, 0=Force 0 into second bit of second XOR (default=1)
RandomEnableXor2 = 1
--1=Enable second bit of third XOR, 0=Force 0 into second bit of third XOR (default=1)
RandomEnableXor3 = 1
--Selects the feedback bit entry point into the LFSR (0-15) (default=15)
RandomFeedback = 15
```

Figure 5.11: Parameters File Excerpt for LFSR Function Directives.

Chapter 6

FUTURE WORK

While implementing a simple scripting language for generating encapsulated test cases is powerful, this idea could be carried further. Using the bus model described above, adding the ability to branch to various lines of the script based on a check unlocks more possibilities. With a branch command, the role of the bus model could be extended to modeling a microprocessor utilizing a simple device independent scripting language. With more effort, a C compiler could be developed to convert simple C code into the simple commands described above. By designing the accurate C model to write simple C instructions instead of the device independent scripting, the C model is now able to write low level C code that can be used in various ways. It could be used to aid in firmware development and could be used at each level of hardware development, from simulation to actual FPGA or silicon fabrication regardless of the platform. During early development verification, the microprocessor model C compiler would compile the C model generated code for simulation verification. At later stages, when an actual microprocessor is available, the corresponding compiler would compile the same C code into the correct assembly language. Device independent and portable platform verification can thus be achieved.

Obtaining identifiable parameters with expected boundaries and an encapsulated environment leads to constrained random verification techniques. In industry today, verification engineers are utilizing tools such as System Verilog to randomize variables within constraints to achieve very high testing coverage. Additionally, code and functional coverage metrics are being applied. The C model described in this thesis is

ideal for utilization within a System Verilog environment. A System Verilog simulation could be developed to generate the parameters files and select input images for each new test case to be created. The System Verilog code would randomize the variables/parameters within specified constraints and be able to generate thousands of test cases. Furthermore, given that the test cases being generated utilize C code, the test cases are directly portable to an FPGA emulation or ASIC platform.

In summary, the C model concept can be further innovated to take advantage of an encapsulated verification environment. This yields several powerful possibilities. Among these possibilities is device independent verification from simulation to actual silicon. Additionally, constrained random verification can be achieved at each stage of verification.

References

[1] F. Wotawa, "Debugging VHDL designs using model-based reasoning", In *Artificial Intelligence in Engineering 14*, 2000. 331-351

[2] *http://www.systemc.org*.

[3] John Sanguinetti, David Pursley, "High-Level Modeling and Hardware Implementation with General-Purpose Languages and High-Level Synthesis", *Ninth IEEE/DATC Electronic Design Processes Workshop*, April 2002.

[4] Wido Kruijtzer, Victor Reyes and Winfried Gehrke, "Design, synthesis and verification of a smart imaging core using SystemC", *Springer Science+Business Media, LLC*, 2006.

[5] *2004 IC/ASIC Functional Verification Study*, Collette International Research, Jan 2005.

[6] Russell Klein, Tomascz Piekarz, "Accelerating Functional Simulation for Processor Based Designs", *Proceedings of the 9th International Database Engineering & Application Symposium*, 2005.

[7] G. Moore, “Cramming More Components onto Integrated Circuits” In *Electronics*, Vol. 38, Number 8, 1965.

[8] R. Pugh, N. Mullenger, J. Hopkins, “Attacking the Verification Challenges”, www.synopsys.com/products/designware/pdfs/dw_vip_wp.pdf.

[9] H. Faque, J. Micahelson, K. Khan, “The Art of Verification”, pp. 213-215 9/2001.

[10] Kelvin Ng, “Challenges in Using System-level Models for RTL Verification”, In *Proceedings-Design Automation Conference*, p. 812-815, 2008, Proceedings of the 45th Design Automation Conference, DAC.

[11] David Brier and Raj S. Mitra, “Use of C/C++ Models for Architecture Exploration and Verification of DSPs”, *ACM, DAC 2006*, July 24-28, 2006.

[12] “Design and Verification of Digital Systems”, In *Scalable Hardware Verification with Symbolic Simulation*, Springer Science+Business Media, Inc, 2006.

[13] Paul Wilcox, “Professional Verification a Guide to Advanced Functional Verification”, Kluwer Academic Publishers, 2004.

[14] Glenn Dearth, Scott Meeth, Paul Whittemore, "Networked Object Oriented Verification with C++ and Verilog", In *Verilog HDL Conference and VHDL International Users Forum*, 1998.

[15] Xing Li, Ramesh Nagarajan, Modeling for Image Processing System Validation, Verification and Testing, In *ACM SIGSOFT Software Engineering Notes*, 2005.

[16] Moretti, G, "Find and fix problems early in the design cycle", http://www.edn.com/article/CA330095.html.

[17] A. Bernstein, M. Burton, F. Ghenassia, "How to Bridge the Abstraction Gap in System Level Modeling and Design", IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004. 6-11 Nov. 2004.

[18] W. Muller, W. Rosenstiel, J. Ruf (Eds.), "SystemC – Methodologies and Applications", Kluwer Academic Publishers, 2003.

[19] "SystemC 2.0 Functional Specification" – Open SystemC Initiative, 2000.

[20] Muhammad Omer Cheema, Omar Hammami, "Introducing Energy and Area Estimation in HW/SW Design Flow Based on Transaction Level Modeling", *In Proc. International Conference on Microelectronics* (ICM), 2006, pp. 182-185.

[21] Seong-Min Kim et al., “Hardware-Software Implementation of MPEG-4 Video Codec”, *ETRI J.*, vol. 25, no. 6, Dec. 2003, pp. 489-502.

[22] S. M. Park et al., “A Single-Chip Video/Audio Codec for Low Bit Rate Application”, *ETRI J.*, vol. 22, no. 1, Mar. 2000, pp. 20-29.

[23] I. E. Bennour, M. Abid and R. Tourki, “Hardware/Software Co-Verification: Models and Methods”, In *SAMS*, 2002, Vol. 42, pp. 1391-1417.

[24] J. Bergeron, “Writing Testbenches: Functional Verification of HDL Models”, Kluwer Academic Publishers, Norwell Massachusetts, 2003.

[25] N. Bombieri, F. Fummi, and G. Pravadelli, “Hardware Design and Simulation for Verification”, *Lecture Notes in Computer Science*, 2006.

[26] J. Bailey, D. Crutchfield and S. Love, “System and Method for Dynamically Shifting Error Diffusion Data”, *United States Patent #7,486,834*, February 3, 2009.

VITA

Personal

Date of Birth: October 15, 1973

Place of Birth: Mayfield, Kentucky

Education

University of Kentucky

Bachelor of Science in Electrical Engineering

December 1997

Professional

Lexmark International

ASIC Engineer/Manager

January 1998 – present

Publications

D. Crutchfield, S. Cunnagin and T. Rademacher, “Method and apparatus for converting IEEE 1284 signals to or from IEEE 1394 signals”, *United States Patent #6,643,728*, November 4, 2003.

A. Billings, D. Crutchfield, D. Iorio and G. Rasche, “Dual tray printer with single drive shaft and dual media picks”, *United States Patent #6,688,590*, February 10, 2004.

D. Crutchfield, T. Rademacher and G. Rasche, “Relaxed-timing universal serial bus with a start of frame packet generator”, *United States Patent #6,801,959*, October 5, 2004.

J. Bates, D. Crutchfield and J. Ward, “Systems and methods for printhead architecture hardware formatting”, *United States Patent #6,817,697*, November 16, 2004.

C. Adkins and D. Crutchfield, “Method and apparatus for sampling digital data at a virtually constant rate, and transferring that data into a non-constant sampling rate device”, *United States Patent #6,865,241*, March 8, 2005.

A. Ahne, D. Crutchfield, M. Edwards and G. Rasche, “Quick edit and speed print capability for a stand-alone ink jet printer”, *United States Patent #7,068,387*, June 27, 2006.

D. Crutchfield, T. Rademacher and G. Rasche, “Method and apparatus for effecting synchronous pulse generation for use in serial communications”, *United States Patent #7,103,125*, September 5, 2006.

D. Crutchfield, T. Rademacher and G. Rasche, “Method and apparatus for effecting synchronous pulse generation for use in variable speed serial communications”, *United States Patent #7,139,344*, November 21, 2006.

D. Crutchfield, “Calculating error diffusion errors to minimize memory accesses”, *United States Patent #7,333,243*, February 19, 2008.

A. Ahne, D. Crutchfield, M. Edwards and G. Rasche, “Quick Edit and speed print capability for a stand-alone ink jet printer”, *United States Patent #7,385,715*, June 10, 2008.

D. Crutchfield, T. Rademacher and G. Rasche, “Method and apparatus for effecting synchronous pulse generation for use in serial communications”, *United States Patent #7,409,023*, August 5, 2008.

J. Bailey, D. Crutchfield and S. Love, “System and method for dynamically shifting error diffusion data”, *United States Patent #7,486,834*, February 3, 2009.

J. Bailey, C. Breswick, D. Crutchfield, R. Garnett and B. Pham, “Systems and methods for error diffusion”, *United States Patent #7,551,323*, June 23, 2009.

J. Bailey, C. Breswick, D. Crutchfield, T. Eade and Z. Fister, “Method of an image processor for transforming a n-bit data packet to a m-bit data packet using a lookup table”, *United States Patent #7,580,564,* August 25, 2009.