An Efficiency Rating Tool for Process-Level VHDL Behavioral Models

by

John A. Wicks, Jr.

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

APPROVED:

Dr. J. R. Armstrong, Chairman

Dr. W. R. Cyre

Dr. S. F. Midkiff

Dr. F. G. Gray

Dr. C. J. Parry

December 1996

Blacksburg, Virginia

Key Words: VHDL, Static Rating, Dynamic Rating, Efficiency, Simulation Performance

An Efficiency Rating Tool for Process-Level VHDL Behavioral Models

by

John A. Wicks, Jr.

Dr. James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

Due to the great complexity of VHDL models that are created today, the amount of processing time required to simulate these models and the amount of labor required to develop these models have become critical issues. The amount of processing time required to simulate a model can be directly influenced by the efficient use of VHDL concepts in creating the model. This dissertation presents an approach to aiding the modeler in the development of more efficient VHDL models. This is done by measuring the simulation efficiency of process-level VHDL behavioral models. Research in the determination of what VHDL constructs and modeling styles are most efficient is presented. The development and use of a tool that parses VHDL behavioral models and reveals the efficiency of the code in the form of a numerical efficiency rating is also presented.

Dedication

This dissertation is dedicated to my father, John Sr., my mother, Shirley, and my sister, Treneice. Their love, encouragement, patience, support, and advice made this effort possible. They have been a constant source of inspiration all of my life.

Acknowledgments

I would like to express my greatest thanks to my advisor, Dr. James R. Armstrong, for his guidance and encouragement throughout this research. I would also like to thank Dr. W. R. Cyre, Dr. F. G. Gray, Dr. S. F. Midkiff, and Dr. C. J. Parry for serving as members of my committee.

I wish to express sincere gratitude to all of my friends and church family for their inspiration and constant support. Finally, I would like to thank all of my fellow graduate students in the Virginia Tech Information Systems Center who helped me immensely throughout my matriculation at Virginia Tech.

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

1.1 Motivation

In today's VLSI (Very Large Scale Integration) era, the complexity of integrated circuits continues to grow at an astonishing rate. Individual chips now contain hundreds of thousands of gates and millions of transistors. Understandably, it has become increasingly more difficult to manage the complexity of designing these chips [1].

Much effort has been made to simplify the aforementioned design process. Many computer tools were developed to be used in computer simulation for system testing and verification. As many simulation systems were being offered, there came a need for a versatile means of accurately describing the hardware systems to be simulated. Netlists were considered inadequate because of their incompatibility with

higher-level abstractions. Programming languages such as C and Pascal were ruled out because they lacked the necessary constructs to model concurrent hardware. Out of these shortcomings arose several specialized hardware description languages such as AHPL, CDL, and ISP. These specialized languages soon evolved into VHDL (**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage), which has since been endorsed by IEEE as a standard [2]. VHDL has rapidly become a very popular hardware description language for the documentation and modeling of complex systems to ensure design accuracy and portability. VHDL has a universal timing model that allows for this unique capability of designing with a high degree of accuracy [3]. VHDL is the first language to allow the designer to capture all the nuances of design complexity and to effectively manage data and the design process [4].

VHDL has indeed been a powerful aid in the design of complex systems. However, as the size and complexity of these systems continue to grow, the size and complexity of the VHDL models used in designing these systems increase as well. Due to the great complexity of VHDL models that are created today, the amount of CPU time required to simulate these models and the amount of labor required to develop these models have become very critical issues. The amount of processing time required to simulate a model can be directly influenced by the efficient use of VHDL constructs in creating the model. While there are many tools available for VHDL code simulation and logic synthesis, there has been very little work done toward the development of tools that meet the ever-growing demand for the quality evaluation and improvement of VHDL descriptions. Therefore, creating a tool that measures the efficiency of VHDL source code would be a significant contribution to the VHDL design community.

1.2 Contributions

The principle objective of this dissertation is to develop a system that measures the simulation efficiency of VHDL behavioral models in the form of a numerical rating. The first step in accomplishing this goal is to determine what VHDL modeling styles are efficient or inefficient in terms of simulation performance. Several VHDL researchers have done studies in this area. A few of note are as follows: Armstrong and Roig [5] studied the simulation performance of a set of VHDL coding styles on a group of models. Pick [6] offered a tutorial that presented efficient VHDL models, unexpected compilation errors, unexpected simulation results, and modeling recommendations. Finally, Paulsen and Levia [7] offered a tutorial that presented techniques for writing high performance and high quality VHDL models.

This dissertation presents an extended study of what VHDL constructs and modeling styles are most efficient. When modeling in VHDL, under many circumstances, there are many different ways to accomplish the same goal. For example, in many cases either signals or variables can be used to represent the same object. In this case, the use of a variable would be more efficient because all signals require scheduling which causes the process of assigning new values to be slower, but variables require no scheduling and when a variable is assigned a new value the change happens immediately. Many models were created to compare the simulation performance of signals versus variables. Many additional constructs were studied such as static sensitivity lists versus dynamic sensitivity lists, the use of resolved signals, the 'EVENT attribute versus the 'STABLE attribute, and much more.

After discussion of various VHDL modeling styles, techniques for measuring the overall efficiency of a VHDL model is discussed. Measurement parameters have

been drawn from the results of the experiments discussed above. More specifically, the percent differences between the simulation performance of different modeling styles that are semantically equivalent have been used in the development of a method to derive static and dynamic efficiency ratings for VHDL behavioral models. Static ratings are based on model content only, but dynamic ratings also consider the way the model is simulated. The tool that provides these ratings measures the efficiency of individual nodes of the Control Flow Graph (CFG) of the VHDL model. This tool has been developed using the VHDL Tool Integration Platform (VTIP), the C programming language, and the Synopsys simulator. Examples demonstrating how well the tool works on a set of VHDL models is included in this dissertation.

Another major contribution of this dissertation is the methodology used in measuring efficiency. Although this dissertation focuses on the efficiency of VHDL constructs, the methodology presented could be applied in the study of other languages and their inherent constructs.

1.3 Contents

This dissertation is organized as follows. Chapter 2, "The Simulation Performance of VHDL Modeling Styles," discusses in detail the experiments done on the simulation performance of various VHDL constructs. Chapter 3, "A Technique for Rating the Efficiency of VHDL Behavioral Models," describes the methodology that is used in rating the efficiency of VHDL models. Chapter 4, "Development of a Tool that Implements the Efficiency Rating Technique," explains the make-up of the computer program that serves as the efficiency rating tool. Chapter 5, "Examples of the Rating of VHDL Models," presents a set of example models that have been evaluated by the

efficiency rating tool. Finally, Chapter 6, "Conclusions," presents conclusions drawn from the research effort and example results. Future research possibilities are also discussed.

Chapter 2. The Simulation Performance of VHDL Modeling Styles

2.1 Introduction

As stated earlier, the first step in developing a tool that measures the efficiency of VHDL models is to determine what modeling styles are most efficient. VHDL is a complex language that allows for the use of many different coding styles to accomplish the same goal. However, when systems that are to be modeled become increasingly larger it is important that designers use the most efficient modeling styles. One of the most important reasons that efficient modeling styles must be used is to optimize simulation performance. The author conducted numerous experiments to identify some of the most

efficient modeling styles. In each experiment, a VHDL model was created to perform a certain function using one particular style. Another VHDL model was created to perform the same function using an alternate style. The same test bench was used to drive both models. The simulation performance of both models was then measured in terms of processing (CPU) time using a program written in C (time1.c - written by Prabhakar). Two different simulators were used in these comparisons. The simulators used were Vantage and Synopsys. Vantage is a compiled code simulator while Synopsys is an interpreted code simulator. In compiled simulators, no run time interpretation is necessary because the entire model is compiled into machine code before simulation. With interpreted simulators, the system interconnect is stored in tables, and the models of the individual devices are called at run time as specified by the inter device signal flow [8]. Generally, compiled code simulators analyze slower and simulate faster, and interpreted code simulators analyze faster and simulate slower. It is important to consider more than one simulator due to the fact that some modeling styles are more efficient in one simulator than in another simulator due to the manner in which the various simulators are coded [9]. Therefore, one modeling style is considered to be more efficient than another modeling style only when it consistently outperforms the alternate style when tested with different kinds of simulators.

In most cases, the test benches that drive the experimental models included for-loops. These for-loops were used to replicate the events, that drive the models, in the test bench enough times to insure that the experimental models would simulate long enough to see a difference between their performances. Because simulation time was extended significantly in some cases, these for-loops also served to demonstrate how these styles might perform in large scale VHDL models that are commonplace today. In all experiments, Vantage simulated models much faster than Synopsys, as expected. Because

of this, in many cases the percent difference between the simulation performance of alternate styles was much greater with Vantage than it was with Synopsys. An explanation of the experiments, their results, and conclusions drawn can be found in the following sections.

2.2 Modeling Style Experiments

2.2.1 Signals Versus Variables

Signals are objects that have a time dimension and whose values may be changed. Signal assignment statements are used to represent real circuit phenomena; therefore if there is no time delay specified for the assignment statement, it is assumed that the signal takes on its new value *delta* time later. Delta is an arbitrarily small time greater than zero [3].

Variables are objects whose value can be changed, and variable assignment statements have no time dimension associated with them which means that the effect is felt immediately. Hence, variables are useful for the representation of algorithms [3].

A **process** is a major modeling element in VHDL. Processes contain sequential statements that execute in order in a programming-like manner. Signals can be used to transfer data between processes, but variables can only be used for local storage inside a process [10]. Therefore the only time signals or variables can be used to perform the same function is when the signal in question is used in statements within a process and not as communication between processes. Signals require scheduling which causes the process of assigning new values to be slower. In fact, with 0 delay, a signal that gets assigned a new value assumes that value 1 simulation cycle later. However, variables require no

scheduling, and, as stated earlier, when a variable is assigned a new value the change happens immediately [7].

The facts discussed above suggest that variables will simulate better than signals. Several different kinds of experiments were done to verify this. In particular, one experiment was done that compared the simulation performance of a model that consisted of only signal assignments to the simulation performance of a model that consisted of only variable assignments. Both models contained 33 assignment statements within a single process with no specified delay. Each signal and variable was of type **bit_vector** (array of binary digits) of length six bits. The variable model simulated 20% faster than the signal model with the Vantage simulator. The variable model simulated approximately 5% faster than the signal model with the Synopsys simulator.

Figures 2.1 and 2.2 illustrate another example of the difference between using a variable as opposed to using a signal. The model to be evaluated is a parallel-to-serial converter. In the efficient model PAR-TO-SERe, (Figure 2.1), a variable is used to represent the object COUNT (COUNT is used to count the number of bits of the input vector that must be fed to the output serially). In Figure 2.1, the first line that is highlighted (bold text) is the declaration of variable COUNT. The other two highlighted lines are assignments statements where COUNT is the destination. In VHDL, variables are declared inside processes and variable assignment statements are represented by ":=". In the inefficient model PAR-TO-SERi, (Figure 2.2), a signal is used to represent the object COUNT. In Figure 2.2, the first line that is highlighted is the declaration of signal COUNT. The other two highlighted lines are assignments statements where COUNT is the destination. In VHDL, signals are declared external to the process and signal assignment statements are represented by "<=". With Synopsys, the efficient model simulated 3% faster than the inefficient model, and with Vantage, the efficient model

simulated 12% faster than the inefficient model. The results of all other experiments that were performed comparing the performance of signals versus variables were consistent with the examples described above. Therefore, it was concluded that it is more efficient, in terms of simulation performance, to use variables instead of signals whenever possible, and this conclusion is supported by data in the literature [5, 6, 7].

```
use work.all;
entity PAR_TO_SERe is
   port (RESET, CLK: in BIT;
         PARIN: in BIT_VECTOR(8 downto 1);
         SO: out BIT);
end PAR_TO_SERe;

architecture BEHAVIORAL of PAR_TO_SERe is

begin

   PARSERe: process(RESET, CLK, PARIN)

     variable COUNT: INTEGER range 0 to 8;

    begin

      if (RESET = '1') then
        COUNT := 8;
      elsif (CLK'EVENT and CLK = '1') then
        if (COUNT > 0) then
          SO <= PARIN(COUNT);
          COUNT := COUNT - 1;
        end if;
      end if;
   end process PARSERe;
end BEHAVIORAL;
```

Figure 2.1 VHDL Model of the Efficient Converter

```
use work.all;
entity PAR_TO_SERi is
   port (RESET, CLK: in BIT;
         PARIN: in BIT_VECTOR(8 downto 1);
         SO: out BIT);
end PAR_TO_SERi;

architecture BEHAVIORAL of PAR_TO_SERi is
 signal COUNT: INTEGER range 0 to 8;
begin
   PARSERi: process(RESET, CLK, PARIN)
     begin
      if (RESET = '1') then
        COUNT <= 8;
      elsif (CLK'EVENT and CLK = '1') then
        if (COUNT > 0) then
          SO <= PARIN(COUNT);
          COUNT <= COUNT - 1;
        end if;
      end if;
   end process PARSERi;
end BEHAVIORAL;
```

Figure 2.2 VHDL Model of the Inefficient Converter

2.2.2 Case Versus If - endif

Case statements are convenient when describing the decoding of registers and other codes. Case statements select one of a number of alternative sequences of statements for execution. Case statements contain multiple *when* clauses. The alternative statements are selected based upon the values in the when clauses that precede them [7]. If - endif statements perform the function normal to all programming languages.

A series of if - endif statements can be created to perform the same operation as any case statement. The number of ifs in the if - endif series would be equivalent to the number of whens in the case statement.

Several experiments were performed that compared the simulation efficiency of models that contained case statements to the simulation performance of models that contained an equivalent series of if - endif statements. In one experiment, both models contained 33 tests (case - 33 whens, if-endif - 33 ifs) with no specific delays. The case model simulated approximately 6% faster than the if - endif model with the Vantage simulator. The case model simulated approximately 3% faster than the if - endif model with the Synopsys simulator. Another example illustrating the difference between case and if - endif is given in Figures 2.3 and 2.4. The model to be evaluated is a simple

---

```
use work.all;

entity DECODERe is
   port (I: in BIT_VECTOR(2 downto 0); EN: in BIT; O: out INTEGER);
end DECODERe;

architecture BEHAVIORAL of DECODERe is

begin

   process(I, EN)
     begin
      if (EN = '1') then
        case I is
          when "000" => O <= 0;
          when "001" => O <= 1;
          when "010" => O <= 2;
          when "011" => O <= 3;
          when "100" => O <= 4;
          when "101" => O <= 5;
          when "110" => O <= 6;
          when "111" => O <= 7;
        end case;
```

```
    end if;
  end process DECODERe;
end BEHAVIORAL;
```

Figure 2.3 VHDL Model of the Efficient Decoder

decoder. The decoder takes a 3-bit input and outputs the equivalent integer. In the efficient model (Figure 2.3 - DECODERe) a case statement is used, and in the inefficient model (Figure 2.4 - DECODERi), a series of if-endif statements is used. With Synopsys,

```
use work.all;
entity DECODERi is
  port (I: in BIT_VECTOR(2 downto 0); EN: in BIT; O: out INTEGER);
end DECODERi;

architecture BEHAVIORAL of DECODERi is

begin

  process(I, EN)
    begin
     if (EN = '1') then
       if (I = "000") then
         O <= 0;
       end if;
       if (I = "001") then
         O <= 1;
       end if;
       if (I = "010") then
         O <= 2;
       end if;
       if (I = "011") then
         O <= 3;
       end if;
       if (I = "100") then
         O <= 4;
       end if;
       if (I = "101") then
         O <= 5;
       end if;
```

```
        if (I = "110") then
          O <= 6;
        end if;
        if (I = "111") then
          O <= 7;
        end if;
      end if;
    end process DECODERi;
end BEHAVIORAL;
```

Figure 2.4 VHDL Model of the Inefficient Decoder

the efficient model simulated 4% faster than the inefficient model, and with Vantage, the efficient model simulated 13% faster than the inefficient model. The results of other experiments comparing case to if were consistent with the ones described above, with percent differences using Vantage ranging from 5% - 13% and Synopsys ranging from 1% - 4%. Based upon these results, it was concluded that it is more efficient to use the case statement instead of an equivalent set of if - endif statements.

2.2.3 Integer Types Versus Logic Types

Several experiments were performed that compared the simulation performance of models that used objects of type integer to the simulation performance of models that used objects of type bit_vector (the binary equivalent to the integers in the first model). One experiment in particular compared one model that contained 33 signal assignments, where all signals were input ports of type integer, to an equivalent model that contained 33 signal assignments, where all signals where input ports of type bit_vector. The integer model simulated approximately 20% faster than the bit_vector model using the Vantage simulator. The integer model simulated approximately 16% faster than the bit_vector

model with the Synopsys simulator. All other experiments that were done comparing integers to bit_vectors yielded results similar to the example described here. However, models that compared the simulation performance of internal objects of type integer versus bit_vector had a slightly smaller percent difference than models comparing input ports of type integer versus bit_vector. Overall, with Vantage, integer models were faster than bit_vector models with a percent difference ranging from 20% - 30%. Using Synopsys, integer models were faster than bit_vector models with a percent difference ranging from 5% - 16%. Based upon these results, it was concluded that it is more efficient, in terms of simulation performance, to use integers instead of equivalent logic types.

2.2.4 Attributes: 'EVENT Versus 'STABLE

Attributes are values associated with a named entity in VHDL. Two signal attributes that are particularly useful in detecting signal changes and detailed timing modeling are 'EVENT and 'STABLE [3]. S'**EVENT** is a function call that returns true if the signal S has changed value during the current simulation cycle. S'**STABLE** is a Boolean signal whose value is true if there has been no event on the signal during the current cycle [7].

In some instances, 'EVENT and 'STABLE can be used to perform the same function. Specifically, S'EVENT is equivalent to not S'STABLE. In this case, 'EVENT is expected to perform better than 'STABLE during simulation because, as discussed above, 'EVENT is a function call and 'STABLE is a boolean signal (the delay caused by signals was discussed in Section 2.2.1 - Signals versus Variables).

Many experiments were conducted to test the simulation performances of 'EVENT and 'STABLE. In one experiment the simulation efficiency of a model that contained 21 if - then statements which each contained three 'EVENT tests was compared to the simulation efficiency of a model with an equivalent set of 21 if - then statements that each contained three not 'STABLE tests. All if - then statements contained 'or' operators (to combine the attributes into one expression), and each if - then statement was followed by a single assignment statement. The 'EVENT model simulated 6% faster than the not 'STABLE model with the Vantage simulator. The 'EVENT model simulated approximately 2% faster than the not 'STABLE model using Synopsys.

Another example illustrating the difference between the simulation performance of 'EVENT and 'STABLE is given in Figures 2.5 and 2.6. The model used in this test is a counter. Figure 2.5 contains the efficient model of the counter (COUNTERe). In this

---

```
use WORK.USER_TYPES.all;
entity COUNTERe is
   generic (DEL: TIME);
   port (RESET, EN, CLK, UP, SYNC, LOAD: in BIT;
         DATA: in BIT_VECTOR(3 downto 0);
         CNT: inout BIT_VECTOR(3 downto 0));
end COUNTERe;

architecture BEHAVIORAL of COUNTERe is
begin
   process(RESET, CLK)
     begin
--
-- RESET
--
      if ((CLK'EVENT and CLK = '1' and RESET = '1' and SYNC = '1') or
         (RESET'EVENT and RESET = '1' and SYNC = '0')) then
        CNT <= "0000" after DEL;
--
-- LOAD
--
      elsif (CLK'EVENT and CLK = '1' and LOAD = '1') then
        CNT <= DATA after DEL;
```

```
--
-- COUNT
--
      elsif (CLK'EVENT and CLK = '1' and EN = '1') then
        if (UP = '1') then
          CNT <= INC(CNT) after DEL;
        else
          CNT <= DEC(CNT) after DEL;
        end if;
      end if;
   end process COUNTERe;
end BEHAVIORAL;
```

Figure 2.5 VHDL Model of the Efficient Counter

model 'EVENT is used to check the status of the clock (CLK) and reset (RESET) input ports. Figure 2.6 contains the inefficient model of the counter (COUNTERi). In this

```
use WORK.USER_TYPES.all;
entity COUNTERi is
  generic (DEL: TIME);
  port (RESET, EN, CLK, UP, SYNC, LOAD: in BIT;
        DATA: in BIT_VECTOR(3 downto 0);
        CNT: inout BIT_VECTOR(3 downto 0));
end COUNTERi;

architecture BEHAVIORAL of COUNTERi is

begin

  process(RESET, CLK)
    begin
--
-- RESET
--
      if ((not CLK'STABLE and CLK = '1' and RESET = '1' and SYNC = '1') or
         (not RESET'STABLE and RESET = '1' and SYNC = '0')) then
        CNT <= "0000" after DEL;
--
-- LOAD
--
```

```
      elsif (not CLK'STABLE and CLK = '1' and LOAD = '1') then
        CNT <= DATA after DEL;
--

-- COUNT
--
      elsif (not CLK'STABLE and CLK = '1' and EN = '1') then
        if (UP = '1') then
          CNT <= INC(CNT) after DEL;
        else
          CNT <= DEC(CNT) after DEL;
        end if;
      end if;
   end process COUNTERi;
end BEHAVIORAL;
```

Figure 2.6 VHDL Model of the Inefficient Counter

model not 'STABLE is used to check the status of the clock and reset input ports. The efficient model simulated 7% faster than the inefficient model with Vantage, and with Synopsys, the efficient model simulated 3% faster than the inefficient model. All other experiments that were conducted to compare the performance of 'EVENT to 'STABLE were consistent with the results detailed here using Synopsys. However with Vantage, there were a few models done that showed either no speedup or a slight decrease in performmance using 'EVENT. Therefore, it is concluded that in terms of simulation performance, it is generally more efficient to use 'EVENT than 'STABLE, however at times some simulators, such as Vantage, may be unpredictable when considering these attributes due to the way the code interpreting them is written. Paulsen and Levia [7] also reached a similar conclusion.

2.2.5 Static Sensitivity List Versus Dynamic Sensitivity List

The **sensitivity list** of a process is the list of signals whose change in value activates the process and causes the statements within the process to be executed [8]. A **static** sensitivity list is specified in the process declaration statement at the beginning of the model (e.g. process (x, y, z)). A **dynamic** sensitivity list is specified at the end of a process in a wait statement (e.g. wait on x, y, z) [11]. **Wait** statements suspend processes until the specified event has occured, in this particular case, changes in the value of x, y, or z). Static sensitivity lists can be used to perform the same function as dynamic sensitivity lists if there are no other wait statements in the process.

Numerous experiments were performed that compared the simulation efficiency of models containing processes with static sensitivity lists to the simulation efficiency of equivalent models containing processes that have dynamic sensitivity lists. In all examples, there was minimal difference between the two modeling styles, with static sensitivity lists appearing to be slightly more efficient at times. It has been documented that static sensitivity lists are more efficient than dynamic sensitivity lists, and this may indeed be the case with some simulators [11]. For this reason, it is suggested that dynamic sensitivity lists be avoided when possible, although the simulation performance difference may be negligible at times.

2.2.6 Use of Aliases

An **alias** is an alternate name assigned to part of an object, which allows for simple access [10].

Several tests were done to compare the simulation efficiency of models using aliases to equivalent models that did not use aliases. In one particular test, both models had 50 input signals of type bit_vector with a length of 8 bits. In the alias model, each signal was separated into two aliases of type bit_vector with a length of 4 bits. In both models, the leftmost 4 bits of each signal was anded with the rightmost 4 bits of the same signal and assigned to some output. The alias model simulated approximately 6% faster than the model with no aliases using Vantage. With Synopsys, the alias model simulated equally as fast as the model with no aliases. The results of other experiments were generally consistent with these results. Overall, the use of aliases seemed to speed up simulation performance with Vantage slightly at times, with no difference evident with Synopsys. Based upon these results, it was concluded that using aliases when possible is an efficient VHDL style with simulation speed-up being evident with some simulators. In cases where there may not be a difference in simulation performance, aliases may be useful in making code more readable.

2.2.7 Use of Records

Records group objects of different types into a single object. These elements can be of **scalar** (single values) or **composite** types (multiple values) [3]. Records can be useful, for example, in representing a complex number rather than using an array of two numbers representing the real and imaginary parts of the complex number [12].

Several tests were done to compare the simulation efficiency of models using records to equivalent models that used arrays. In one specific example, the record model consisted of one record with 16 elements of type real and 16 elements of type integer. The array model consisted of two arrays. One array had 16 elements of type real, and the

other array had 16 elements of type integer. Using Synopsys, the record model simulated 8% faster than the array model. With Vantage, the record model simulated equally as fast as the array model. The results of other experiments were in agreement with these results. Therefore it was concluded that with some simulators records are at times more efficient than arrays when representing certain types of objects in VHDL, such as complex numbers.

2.2.8 Constrained Arrays Versus Unconstrained Arrays

Constrained arrays are array types that are declared with constant values for the ranges. **Unconstrained arrays** declare a type that has a variable number of elements to be determined when a variable or signal of that type is actually declared.

Several tests were done to compare the simulation efficiency of models containing constrained arrays to models containing unconstrained arrays. In one specific example, the constained array model and the unconstrained array model consisted of the declaration of 20 arrays whose elements were of type bit_vector with a length of 4 bits. Both models also contained 40 assignment statements using the arrays. The unconstrained arrays were defined when the signals of its type were declared. The constrained array model simulated only slightly faster than the unconstrained array model using Vantage. With Synopsys, the simulation performance of the constrained array model was essentially equal to that of the unconstrained model. Since the results of all experiments were in agreement with these results, it was concluded that the difference between the simulation performance of constrained arrays and unconstrained arrays is negligible.

2.2.9 Use of Resolved Signals

VHDL supports busing and wiring operations in the form of **resolution functions**. Resolution functions can solve bus contention with signals. However, there are times when resolution functions can be avoided. For example, when a port of mode **inout** (Objects of mode inout can serve as both inputs and outputs) is being driven in the process model and in the test bench, resolved signals can be used to solve contention. This method can be avoided by using an extra signal to separate one port of mode inout into two ports, one for input and the other for output with an extra internal signal (not a port) maintaining intermediate values.

Many tests were done to compare the simulation efficiency of models that employed the resolved signal approach to the simulation efficiency of models that used the extra signals as described above. In one specific test, the resolved signal model was 16% slower than the other model using Vantage and it was 23% slower using Synopsys. All other example results were consistent with these results. Therefore it was concluded that the use of resolved signals is very inefficient in terms of simulation performance and should be avoided whenever possible.

2.2.10 Use of Assertions

Assertions can be used in VHDL to identify timing violations. Tests were performed to check the effect of assertions on the simulation performance of a model. In one example, a controlled counter was created that used assertions to check for setup time and pulse width timing violations. The same model was simulated without assertions. Using Synopsys and Vantage, the time to simulate was not significantly increased in the

model with assertion checks. Based upon the example results, it was concluded that in models with a small number of assertions, the time to simulate will not increase significantly.

2.2.11 Use of File I/O

VHDL provides for **file I/O** capability so that external files can be read from and written to during simulation. The two types of files that have this capability are text and formatted files. **Text I/O** allows for the reading of text files (which can be created by a text editor or some other system program) during simulation. In **formatted I/O**, formatted files must be written by a VHDL simulation and file format is host dependent and is not human readable in the host environment.

Several experiments were performed to test the simulation performance of models using file I/O. One example is illustrated in Figure 2.7 [8]. It is a simple model that employs text I/O to read input test vectors. In this model, the process is activated once and all input vectors are read in a while - loop where each iteration of the loop is delayed for a period of time defined by the generic value PER. An alternative model was developed that performed the same function but employed signal assignments in a test

---

```
use work.all;
entity FILEIO is
   generic (PER: TIME);
   port (EN: in BIT; BVOUT: out BIT_VECTOR(7 downto 0));
end FILEIO;
use STD. TEXTIO.all;
architecture TIO of FILEIO is
begin
   process
       variable VLINE: LINE;
       variable V: BIT_VECTOR(7 downto 0);
```

```
    file INVECT: TEXT is "TVECT.TXT";
  begin
    wait on EN until EN = '1';
    while not (ENDFILE(INVECT)) loop
      READLINE(INVECT, VLINE);
      READ(VLINE, V);
      BVOUT <= V;
      wait for PER;
    end loop;
  end process;
end TIO;
```

Figure 2.7 VHDL Model that Employs TEXT I/O.

bench instead of using a text file. With Synopsys, the text I/O model was 26% faster than the test bench model, and with Vantage, the text I/O model was 17% faster than the test bench model. The results of all other experiments with models like this (where the models consisted of a process that was activated once and vectors are read periodically in a loop) were in agreement with the results in this experiment. However, in some models where processes are activated many times in order to receive input vectors (e.g. an external signal is used to indicate when the next vector should be read), text I/O only provided minimal speed-up. Examples were also done using formatted I/O and the results of these experiments were consistent with the results of the file I/O examples. Based upon all of these results, it was concluded that file I/O is an efficient modeling technique with very significant speed-up in simulation performance with certain kinds of models.

2.2.12 Use of Generics

Generics provide a way for static information to be fed to a model from its environment. Generics are most often used to define the timing characteristics of a model.

It has been suggested that generic parameters should be avoided because they can be expensive in terms of memory requirements during simulation run-time [11].

Several experiments were performed to test the simulation performance of models that employ generics. In one specific example, the simulation performance of a model that contained 33 generic names which represented the timing delays of 33 different signal assignment statements was compared to the simulation performance of a functionally equivalent model that contained the same assignment statments, but the actual delay was specified in the model. Using Synopsys and Vantage, the simulation times of both models were essentially the same. Other experiments were in agreement with this one. For large models where memory requirements are critical, it is suggested that generics be avoided although, as the experimental results suggest, they frequently do not hinder simulation performance.

2.2.13 Use of Functions

A **function** is a type of subprogram in a VHDL model. When a function is called, values are passed in through parameters before execution. The function then executes and returns only one value [10].

Several experiments were done to compare the simulation efficiency of models that contained functions to the simulation efficiency of equivalent models that did not use functions. In one particular experiment, one model used a function to convert bit_vectors into integers, and in the other model the type conversion was performed in the main process. In both models, the converted value was used in a simple addition operation. Using Synopsys and Vantage, the function model simulated equally as fast as the other model. Other experiments performed were consistent with this. Therefore it was

concluded that at times basic functions do not significantly hinder simulation performance.

2.2.14 Use of Procedures

A **procedure** is a type of subprogram in a VHDL model similar to the VHDL function but more flexible. Unlike functions, procedures can return more than one value, using parameters. The parameters are of mode IN, OUT, and INOUT. IN brings a value in, OUT sends a value back through an argument list, and INOUT brings a value in and sends it back. Parameters can be signals or variables [10].

Several experiments were done to test the simulation performance of procedures. In one particular experiment, a model used a procedure that converted bit_vectors into integers. In the model without the procedure, the type conversion was performed in the main process. In both models, the converted value was used in a simple addition operation. With Synopsys and Vantage, the procedure simulated 10% faster than the other model. Other examples showed minimal difference in simulation performance. Therefore, it was concluded that the simulation performance of models that employ procedures is highly dependent upon the operations that the procedures performs and the objects that are being used by the procedure.

2.2.15 Scalars Versus Arrays

A **scalar** is a literal made up of one element or value. **Arrays** are made up of multiple values.

Tests were done to compare the simulation efficiency of models using scalars to the simulation efficiency of equivalent models using arrays. In one specific example, the scalar model used 10 inputs of type bit in 10 logical 'and' operations, and the array model used one input of type bit_vector with length 10 bits in 10 logical 'and' operations. Using Synopsys and Vantage, the simulation times of both models was found to be essentially equivalent. Other example results were consistent with this, therefore it was concluded that the use of arrays does not significantly increase simulation time.

2.3 Summary

As stated earlier, all of the experiments described above were performed in order to gather more information concerning the simulation performance of certain VHDL modeling techniques. These particular VHDL constructs were chosen for tests because of information gathered in literature reviews, suggestions from VHDL experts, etc. Other tests that may not have been widely suggested and are not documented in this chapter were done including access types, for-loops, if-elsif, and conditional assignment statements. In all experiments that are not detailed in this chapter, no significant discoveries concerning simulation performance were made.

The results of the experiments described in the previous section could provide any VHDL modeler with useful information. After observing the differences between the simulation performance of models containing alternate, but functionally equivalent modeling styles, it is apparent that the strategic use of certain VHDL modeling styles could significantly reduce simulation time and thus make a model more efficient. As stated before, a tool that provides a numerical efficiency rating for VHDL models would be very useful today. Chapter 3 of this dissertation will include a demonstration of how

the experimental results discussed in this chapter can be used in the development of an efficiency rating tool.

Chapter 3. A Technique for Rating the Efficiency of VHDL Behavioral Models

3.1 The Rating Scheme

While there has been some work done in the determination of which VHDL modeling styles are most efficient, there has been little work done towards developing systems that examine VHDL models and produce a numerical efficiency rating [7]. Therefore, an efficiency rating system has been developed that accurately measures the efficiency of process-level VHDL models in terms of simulation performance.

In this system the contents of a process in a VHDL model are represented by a *control flow graph*. The **Control Flow Graph** (CFG) is a graphical representation of a program in the form of a directed graph, G(V, T), where V is the set of nodes and T is the set of directed edges. Each node V_i represents a distinct VHDL statement (terminated by a semicolon). Each directed edge T_{ij} establishes a directed connection between nodes V_i and V_j. This directed connection indicates that the node V_j is executed directly after the execution of node V_i, and T_{ij} represents the transfer of control. Therefore, the CFG can be used to define the execution order of statements within a program.

A system of penalties was developed for certain VHDL modeling styles based on an exhaustive set of experiments which were discussed in Chapter 2. The penalty for using a particular inefficient modeling style is some numeric value that is close to the percent differences in simulation performance found between models containing the inefficient modeling style and equivalent efficient models. For example, as discussed in Chapter 2, Section 2.2.1, experimental models that used objects that were signals simulated slower than equivalent models that used variable objects with a difference ranging from 12% to 23% using Vantange. The penalty given for inefficiently using signals with Vantage was chosen to be 20, which is within the 12 to 23 range.

The penalties are applied to the individual nodes of the control flow graph. Each node of the control flow graph of a model is initially given a numerical weight of 100. Then each node is evaluated in terms of any inefficient characteristics it may contain. The necessary penalties are deducted from the initial weight of 100. The overall **static** efficiency rating of the model is then calculated by taking the average of the final weights of each individual node. This rating is called the static rating because it is derived by considering only the makeup of the model which does not change. The

formula for the static rating is as follows:

$$E_s = \left(\sum_{i=0}^{n-1} N_{Ei} \right) \Big/ n$$

In the formula, E_s represents the overall static efficiency rating. N_{Ei} is the node efficiency rating at node i and n is the total number of nodes in the CFG.

An example of this system is given in Figures 3.1 to 3.4. The model to be evaluated is a simple RAM [3]. The efficient model of the RAM (RAMe) is given in Figure 3.1. The inefficient model of the RAM (RAMi) is given in Figure 3.2 with the node numbers from the corresponding CFG placed at the left of each VHDL statement. Figure 3.3 contains the CFG of the inefficient model with the final weight, N_{Ei}, of each node i. In this inefficient model a signal is used to represent memory (MEM), but in this case a variable could have been used. With Vantage, as stated earlier, the penalty

---

```
use work.all, work.USER_TYPES.all;
entity RAMe is
  generic (RDEL, DISDEL: TIME);
  port (DATA: inout MVL4_VECTOR(7 downto 0) := "ZZZZZZZZ";
        ADDR: in BIT_VECTOR(3 downto 0);
        RD, WR, NCS, INIT: in BIT);
end RAMe;
architecture BEHAVIORAL of RAMe is
    type MEMORY is array (0 to 15) of MVL4_VECTOR(7 downto 0);
begin
     RAM_E: process(NCS, RD, WR, INIT)
                variable MEM: MEMORY;
        begin
        if (INIT'EVENT and INIT = '1') then
          MEM(0) := "00000000";
          MEM(1) := "00000001";
          MEM(2) := "00000010";
          MEM(3) := "00000011";
```

```
        MEM(4) := "00000100";
        MEM(5) := "00000101";
        MEM(6) := "00000110";
        MEM(7) := "00000111";
        MEM(8) := "00001000";
        MEM(9) := "00001001";
        MEM(10) := "00001010";
        MEM(11) := "00001011";
        MEM(12) := "00001100";
        MEM(13) := "00001101";
        MEM(14) := "00001110";
        MEM(15) := "00001111";
      end if;
      if (NCS = '0') then
       if (RD'EVENT) then
        if (RD = '1') then
         DATA <= MEM(INTVAL(ADDR)) after RDEL;
        else
         DATA <= "ZZZZZZZZ" after DISDEL;
        end if;
       elsif (WR'EVENT and WR = '1') then
        MEM(INTVAL(ADDR)) := DATA;
       end if;
      else
       DATA <= "ZZZZZZZZ" after DISDEL;
    end if;
  end process;
endBEHAVIORAL;
```

Figure 3.1 Efficient VHDL Model of the RAM

for using signals instead of variables is 20, therefore, each signal assignment statement that has signal MEM as its destination will be penalized 20 points (Fig. 3.3, nodes 1 to 16, 24). Using the static efficiency rating formula, E_s for this model is 87.

```
use work.all, work.USER_TYPES.all;
entity RAMi is
  generic (RDEL, DISDEL: TIME);
  port (DATA: inout MVL4_VECTOR(7 downto 0) := "ZZZZZZZZ";
        ADDR: in BIT_VECTOR(3 downto 0);
        RD, WR, NCS, INIT: in BIT);
end RAMi;
architecture BEHAVIORAL of RAMi is
```

```
    type MEMORY is array (0 to 15) of MVL4_VECTOR(7 downto 0);
    signal MEM: MEMORY;
begin
    RAM_I: process(NCS, RD, WR, INIT)
      begin
 0    if (INIT'EVENT and INIT = '1') then
 1      MEM(0) <= "00000000";
 2      MEM(1) <= "00000001";
 3      MEM(2) <= "00000010";
 4      MEM(3) <= "00000011";
 5      MEM(4) <= "00000100";
 6      MEM(5) <= "00000101";
 7      MEM(6) <= "00000110";
 8      MEM(7) <= "00000111";
 9      MEM(8) <= "00001000";
10      MEM(9) <= "00001001";
11      MEM(10) <= "00001010";
12      MEM(11) <= "00001011";
13      MEM(12) <= "00001100";
14      MEM(13) <= "00001101";
15      MEM(14) <= "00001110";
16      MEM(15) <= "00001111";
      end if;
17    if (NCS = '0') then
18      if (RD'EVENT) then
19        if (RD = '1') then
20          DATA <= MEM(INTVAL(ADDR)) after RDEL;
21        else
22          DATA <= "ZZZZZZZZ" after DISDEL;
          end if;
23      elsif (WR'EVENT and WR = '1') then
24        MEM(INTVAL(ADDR)) <= DATA;
        end if;
25    else
26      DATA <= "ZZZZZZZZ" after DISDEL;
    end if;
  end process;
end BEHAVIORAL;
```

Figure 3.2 Inefficient VHDL Model of the RAM

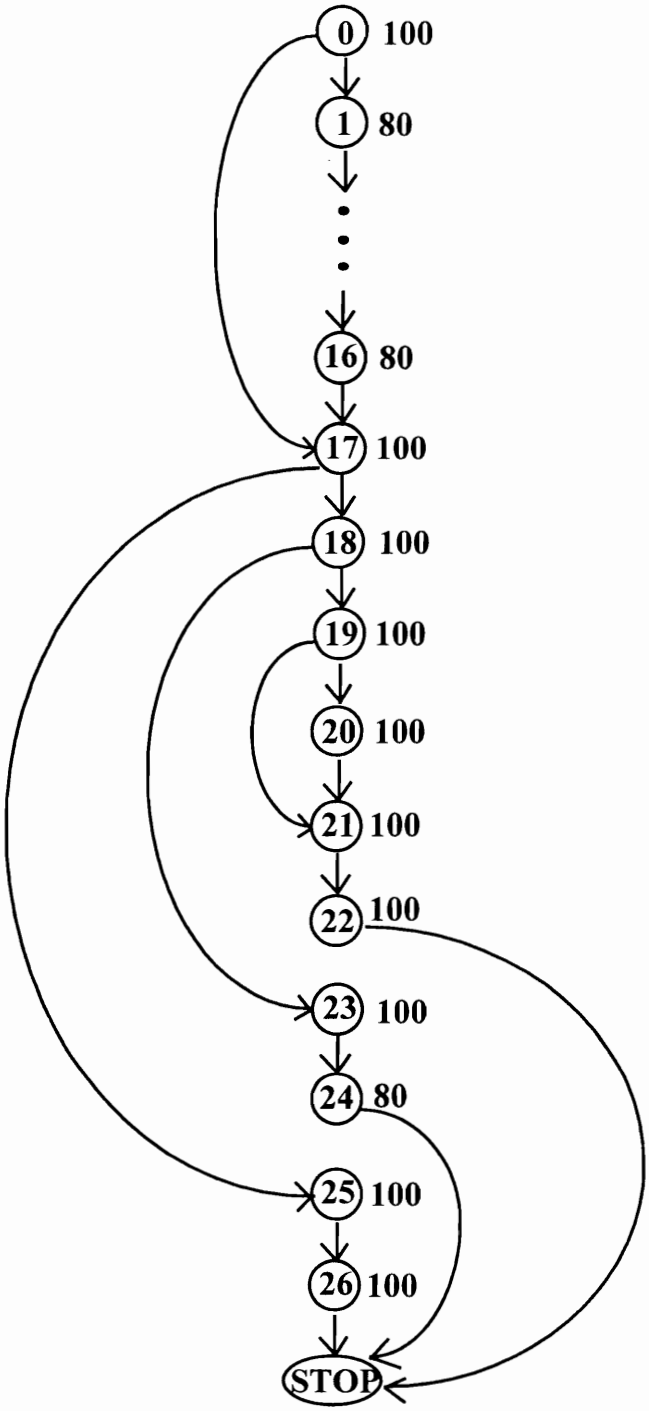

Figure 3.3 CFG of the RAM Model with Node Weights

The static rating is very useful, but it does not take into account the fact that certain paths of the CFG will be traversed many more times than other paths. Therefore, a more accurate **dynamic** rating system has been developed that considers the frequency of path traversals as a factor. To obtain this rating the test bench of the model is needed so that the **coverage** of each node can be retrieved. The coverage of a node is the number of times a node is executed during simulation of the model. The Synopsys coverage utility is used to retrieve node coverages. This coverage utility can be invoked using a coverage command in the simulation **control** file. The control file is normally used to specify what objects are to be reported in the output listing after simulation. However, in this case the node coverages is the only concern, therefore no object output values are requested and simulation to get only node coverages is much faster than normal simulations. The formulas used in calculating the dynamic efficiency rating are as follows:

$$C_T = \sum_{i=0}^{n-1} C_i$$

and

$$E_D = \left[\sum_{i=0}^{n-1} \left(C_i \bullet N_{Ei} \right) \right] \Big/ C_T$$

The coverage at each node i is C_i, and the sum of all node coverages is C_T. The node coverage, C_i, is multiplied by the node weight, N_{Ei}, to form a modified weight at each node i. The overall dynamic rating, E_D, is then calculated by taking the sum of all modified node weights and dividing by the sum of the node coverages. The dynamic rating is useful because it allows the modeler to pinpoint inefficient nodes in the model that are being traversed a high percentage of time and are, therefore, hindering simulation performance.

After simulation of the RAM model, the node coverages for the CFG in Figure 3.3 were as follows:

Nodes 0 and 17 -- coverage = 5,051

Nodes 1 - 16 -- coverage = 101

Node 18 -- coverage = 4,850

Node 19 -- coverage = 3,232

Node 20 - 22, 24 -- coverage = 1,616

Node 23 -- coverage = 1,618

Nodes 25 and 26 -- coverage = 201

Using these node coverages and the node weights shown in Figure 3.3, the overall dynamic efficiency rating, E_D, is 97.

Figure 3.4 demonstrates the accuracy of the rating schemes (Vantage). Using test_bench1 (the test bench used above), the time to simulate the efficient RAM model

RAM MODEL		Simulation Time	Actual Rating	Dynamic Rating	Static Rating
Test_bench1	RAMe	17.33 sec	100	100	100
	RAMi	17.46 sec	99	97	87
Test_bench2	RAMe	530 msec	100	100	100
	RAMi	600 msec	88	86	87

Figure 3.4 Efficiency Predictions and Actual Results

(RAMe) was 17.33 s. The time to simulate RAMi (inefficient) was 17.46 s. These times yield a 1% difference in simulation time. This means that the **actual** rating of RAMi is 99 (100 - 1). As stated above, the dynamic rating given by the rating system is 97 and the static rating is 87. The dynamic rating is much closer to the actual rating than the static rating because the node coverages are taken into account. More specifically, test_bench1 causes the initialization of memory (nodes 1 to 16), which are penalized for inefficiency, to be traversed a low percentage of the time, but as stated earlier, the static rating does not consider the percentages of node traversals and places equal weight on each node rating. However, in test_bench2, the static rating is closer to the actual rating because nodes 1 to 16 are traversed at a higher percentage of time than in test_bench1.

The static and dynamic ratings can obviously be useful in aiding designers in the development of highly efficient VHDL models. Chapter 5 includes discussions of many more examples that were done demonstrating the effectiveness of the rating system. The next section in this chapter details the various VHDL constructs that are penalized in models and what the specific penalties are.

3.2 Modeling Style Penalty Weights

As stated earlier, the penalty for using a particular inefficient construct is derived from the percent difference in the simulation performance of the experimental models that compared efficient constructs to their inefficient counterparts (Chapter 2). The following sections detail the inefficient constructs that are identified and penalized. Experiments revealed that it was not necessary to penalize a node more than once for the same violation in most instances. For example, if an elsif statement had more than

one occurrence of not Signal'Stable it would still be penalized only once. This can be assumed in all cases listed unless otherwise indicated. In most instances, the penalties are applied to the nodes of the CFG of a model, however, there are a couple of exceptions to this rule which are explained in (3), (4), and (9).

(1) If-Statement Penalties

If-statement CFG nodes are penalized in the following instances:

- Use of not Signal'STABLE (should be Signal'EVENT)
  Penalty: Synopsys - warning only
  Only a warning is given for Synopsys because of the small amount of speedup. No penalty or warning is given for Vantage because of slightly variant performance results (Chapter 2, Section 2.4). This is true for all instances of not 'STABLE found below.
- Use of a port of type bit_vector that could be an integer
  Penalty: Vantage - 10 points, Synopsys - 10 points
- Use of a function call that performs an arithmetic operation on a bit_vector that could be an integer (If the integer is used the function could then be avoided)
  Penalty: Vantage - 20 points, Synopsys - 20 points
- Use of a series of If-then-endif statements that could be a case statement
  Penalty: Vantage - 10 points, Synopsys - 2 points

Most of the same penalties were given for elsif statements:

- Use of not Signal'STABLE (should be Signal'EVENT)

  Penalty: Vantage - 10 points, Synopsys - 2 points
- Use of a port of type bit_vector that could be an integer

  Penalty: Vantage - 10 points, Synopsys - 10 points
- Use of a function call that performs an arithmetic operation on a bit_vector that could be an integer (If the integer is used the function could then be avoided)

  Penalty: Vantage - 20 points, Synopsys - 20 points

(2) Case Statement Penalties

Case statement CFG nodes are penalized in the following instances:

- Use of not Signal'STABLE in the control expression (should be Signal'EVENT)

  Penalty: Synopsys - warning only
- Use of a bit_vector port that could be an integer in the control expression

  Penalty: Vantage - 20 points, Synopsys - 20 points

(3) Variable Assignment Statement Penalties

Variable assignment statement CFG nodes are penalized in the following cases:

- Use of a bit_vector port that could be an integer

  Penalty: Vantage - 20 points, Synopsys - 10 points
- Use of a an internal object (not a port) of type bit_vector

that could be an integer

Penalty: Vantage - 20 points, Synopsys - 10 points

- Use of a function call that performs an arithmetic operation on a bit_vector port that could be an integer

  Penalty: Vantage - 30 points, Synopsys - 20 points

- Use of a function call that performs an arithmetic operation on an internal object of type bit_vector that could be an integer

  Penalty: Vantage - 30 points, Synopsys - 15 points

- Use of an unnecessary resolved signal

  In this case, the points are not deducted from the node. Instead they are deducted from the overall rating. This is done because of the enormous effect inefficient resolved signals have on simulation performance.

  Penalty: Vantage - 15 points, Synopsys - 10 points

- Use of not Signal'STABLE (should use Signal'EVENT)

  Penalty: Synopsys - warning only

(4) Signal Assignment Statement Penalties

Signal assignment statement CFG nodes have all of the penalties that variable assignment statements have, and there are a few additional penalties as well. All of the penalties are as follows:

- Use of a bit_vector port that could be an integer

  Penalty: Vantage - 20 points, Synopsys - 10 points

- Use of a an internal object (not a port) of type bit_vector

that could be an integer

Penalty: Vantage - 20 points, Synopsys - 10 points

- Use of a function call that performs an arithmetic operation on a bit_vector port that could be an integer

  Penalty: Vantage - 30 points, Synopsys - 20 points

- Use of a function call that performs an arithmetic operation on an internal object of type bit_vector that could be an integer

  Penalty: Vantage - 30 points, Synopsys - 15 points

- Use of an unnecessary resolved signal

  In this case, the points are not deducted from the node, but from the overall rating as explained before in (4).

  Penalty: Vantage - 15 points, Synopsys - 10 points

- Use of not Signal'STABLE (should use Signal'EVENT)

  Penalty: Synopsys - warning only

- Destination signal could be a variable

  Penalty: Vantage - 20 points, Synopsys - 2 points

(5) Wait Statement Penalties

Wait statement CFG nodes are penalized in the following instance:

- Use of not Signal'STABLE (should use Signal'EVENT)

  Penalty: Synopsys - warning only

- Use of a dynamic sensitivity list that could be static

  Only a warning is given. No penalty is given because experimental results revealed that with many simulators

simulator performance difference is minimal.

(6) Procedure Call Penalties

Procedure call CFG nodes are penalized in the following case:

- Use of a bit_vector as a parameter that could be an integer in a procedure call for a procedure that performs an arithmetic operation on the bit_vector.
  Penalty: Vantage - 20 points, Synospys - 20 points

(7) Assertion Statement Penalties

Assertion Statement CFG nodes are penalized in the following case:

- Use of not Signal'STABLE (could use Signal'EVENT)
  Penalty: Synopsys - warning only

(8) While-loop Statement Penalties

While-loop CFG nodes are penalized in the following instances:

- Use of bit_vector ports that could be integers
  Penalty: Vantage - 10 points, Synopsys - 10 points
- Use of a function call that performs an arithmetic operation on a bit_vector that could be an integer.
  Penalty: Vantage - 20 points, Synopsys - 20 points

(9) Models with an Inefficient Method of reading Input Vectors

As described in Section 2.2.11, certain types of models simulate more efficiently when input vectors are read using file I/O rather than signal assignments in a test bench. In this case, as with resolved signals, the points are not deducted from the node but from the overall rating. As stated earlier, this is done because of the greater than normal effect the use of file I/O has on the simulation performance of certain types of models. Penalties are:

Vantage - 15 points, Synopsys - 20 points

There are conditions that must be met in order to enforce many of the penalties that are listed. These conditions are described in detail in Chapter 4. Chapter 4 also contains an explanation of the algorithms that parse the VHDL code to check for these conditions that allow different nodes to be penalized. As stated earlier, Chapter 5 contains examples of how these penalties are employed.

Chapter 4. Development of a Tool that Implements the Efficiency Rating Technique

4.1 Tools Used in Program Development

The development of a tool that implements the efficiency rating scheme is a very involved process. It includes the use of the C programming language, the VHDL Tool Integration Platform (*VTIP*) VHDL analyzer, the VTIP Design Library System (*DLS*), the DLS *Browser* and the Software Procedural Interface (*SPI*) package.

VTIP is a tool that can be used to evaluate the structure and characteristics of VHDL models. The first step in using VTIP is to analyze the vhdl model to be studied. The analyzer detects all language and some design errors and gives helpful hints as to

how the errors could be corrected. The VTIP VHDL analyzer is invoked using the following command:

```
>       vhdl filename[.ext][-qualifiers]
```

The .ext is the extension of the vhdl filename (e.g. .vhd or .vhdl). The -qualifiers option allows the user to specify various switches to obtain the desired output. For example, the -list qualifier produces a numbered source listing in the file filename.lis. If no errors are present in the model, it is then stored in a design library.

The **VTIP Design Library System (DLS)** provides for the intermediate storage of analyzed VHDL design information. It is made up of Design Libraries and Design Library Units. The Design Library Units are the basic unit of storage for design data in the DLS. The Design Libraries are implemented as system directories and the Design Library Units are implemented as files stored in these directories.

Once the design library units have been established with the analyzed information, the DLS **Browser** can be used to examine them. The Browser is a screen-oriented utility that enables a user to open the library units, and traverse *nodes*, *lists*, and other data structures within a library unit. A **node** may represent a concept, an object, or simply a locus of information within a structure. An example of a node would be a VHDL process. A **list** is an ordered collection of *items*. Each **item** in list represents a node that resides in the list at the position of the item. An example of a list would be the list of highest-level statements that are contained within that process, and each individual statement would then be an item in the list [13].

The **Software Procedural Interface (SPI)** consists of callable routines and data types that implement the operations and data types of the DLS. It is used in accessing

design data stored within the DLS. Using SPI, a tool can open a library unit (i.e. analyzed VHDL model) traverse its internal structure and examine data or objects within the structure [14].

4.2 The Efficiency Rating Generator

The first step in implementing the efficiency rating scheme was to develop an automatic control flow graph generator. The CFG generator takes any VHDL model, that contains a single process, as input and textually displays the control flow information. This CFG generator has been developed using the aforementioned tools. Specifically, it is written in C (618 lines of code) and uses SPI to gather the neccessary information about the VHDL model (that has been analyzed using VTIP) in order to determine the flow of control in the model.

Once the CFG generator was completed, the next step was to develop the efficiency rating generator (ERG). The ERG is a C program (rating.c - 4,800 lines of code) that contains the CFG generator along with code that uses SPI and VTIP to examine the nodes of the CFG and provide an efficiency rating of the model based upon the characteristics of each node.

The remainder of this chapter contains details concerning the makeup of the ERG. This includes descriptions of the algorithms that are used in determining when nodes should be penalized.

4.2.1 Program Input

Before an efficiency rating for a model can be obtained, the model must be analyzed using the VTIP VHDL analyzer. The efficiency rating generator can then be invoked by typing the Unix executable filename **rating.** After the program has been invoked the user is prompted for several responses as illustrated in Figure 4.1. The user

```
> rating

Type "s" for Synopsys or "v" for Vantage:                          v

Type "s" (static rating) or "d" (dynamic rating):                  d

Enter the Library name: (default=Work)
Enter the Entity name:                                             PAR_TO_SERi
Enter the Architecture name: (default=BEHAVIORAL)

Enter the model name with extension: (default=PAR_TO_SERi.vhd)
Enter the test bench name with extension:                          par_to_ser_test.vhd
Enter the architecture name of the test bench:                     PAR_SER_TEST
```

Figure 4.1 Invoking the Efficiency Rating Generator

must indicate if the rating should be for a Synopsys or Vantage simulation, static or dynamic rating, the library name where the analyzed model is stored, entity name, and architecture name of the model. If a dynamic rating is requested the user is prompted for more information, which includes: The name of the file that contains the model, the name of the file that contains the test bench, the entity name in the test bench, and the architecture name in the test bench. The additional information, including the test bench, is needed for the dynamic rating because the coverage for each node must be obtained using Synopsys. In Figure 4.1, when no response is given for a particular

prompt, the default was taken. After providing the responses requested above, the user may have to answer more questions regarding the makeup of the model so that the penalties can be applied appropriately. This is discussed in more detail later in this chapter.

4.2.2 The Control Flow Graph Generator

As stated earlier, the CFG generator is contained within the efficiency rating generator. The control flow information is determined for each node immediately before the node is evaluated for modeling style violations. Each distinct VHDL statement contained within the process is placed in a DLS list, named STMTS, in order of occurrence in the process. Therefore each VHDL statement in the list STMTS is a DLS node that can be traversed to determine the necessary control flow information.

Each node of a CFG is considered to have a true and false edge, but the false edge of non-conditional VHDL statements is undefined. Determining the true edges of some statements can be a very complicated task. For example, the true edge of a top-level (not nested) assignment statement simply points to the next statement in the process, however, if the assignment statement is the last statement in a group of nested (e.g. inside an if-then statement or for-loop) statements then the process of finding the true edge is much more complicated. In this case the various nodes of the DLS must be traversed in order to find the parent statement of the assignment statement and the successor of the parent statement at its level in the DLS hierarchy. If there is no successor to the parent DLS node then the DLS structure must be checked for a parent statement of the parent statement (e.g. nested ifs) or grandparent DLS node. The

successor then must be found for the grandparent DLS node. The programming technique used in finding the true and false edges in cases such as this is recursion.

There are varying degrees of complexity involved in finding the control flow information for all of the various kinds of VHDL statements. However, the use of recursion and many subroutines makes the process more manageable. Figures 4.2 and 4.3 illustrate the control flow information that the tool produces for each model. The model is the parallel-to-serial converter used in chapter 2. The control flow information

```
1   use work.all, work.USER_TYPES.all;
2   entity PAR_TO_SERi is
3     port (RESET, CLK: in BIT;
4           PARIN: in BIT_VECTOR(8 downto 1);
5           SO: out BIT);
6   end PAR_TO_SERi;
7
8   architecture BEHAVIORAL of PAR_TO_SERi is
9     signal COUNT: INTEGER range 0 to 8;
10    begin
11      P: process(RESET, CLK, PARIN)
12        begin
13         if (RESET = '1') then
14           COUNT <= 0;
15         elsif (CLK'EVENT and CLK = '1') then
16           if (COUNT >= 0) then
17             SO <= PARIN(COUNT);
18             COUNT <= COUNT - 1;
19           end if;
20         end if;
21      end process P;
22  end BEHAVIORAL;
```

Figure 4.2 The Inefficient Parallel-to-Serial Converter Model

for each VHDL statement is given in terms of the line number in the model (Figure 4.2) and the CFG node number that is assigned to each statement in the program. The false edges of non-conditional statements are considered to be undefined.

CFG Node 0 (Line #13)
- True edge points to CFG Node 1 (Line #14).
- False edge points to CFG Node 2 (Line #15).

CFG Node 1 (Line #14)
- True edge points to the end of process code.
- False edge is undefined.

CFG Node 2 (Line #15)
- True edge points to CFG Node 3 (Line #16).
- False edge points to the end of process code.

CFG Node 3 (Line #16)
- True edge points to CFG Node 4 (Line #17).
- False edge points to the end of process code.

CFG Node 4 (Line #17)
- True edge points to CFG Node 5 (Line #18).
- False edge is undefined.

CFG Node 5 (Line #18)
- True edge points to the end of process code.
- False edge is undefined.

Figure 4.3 The CFG Generator Output

4.2.3 Identifying Modeling Style Violations

4.2.3.1 Inefficient Signal Assignment Statements

Before a signal assignment statement can be penalized because a more efficient variable assignment statement could be used, several conditions must be met. In the ERG, all of these conditions are checked for. They are as follows:

(1) The list of all statements must be checked for signal assignment statements that have the same target signal as the assignment statement in question. If any such statement has some specified delay associated with it, the statement being checked for a penalty cannot be penalized because the target must be a signal since variables have no delay.

(2) The process sensitivity list must be checked to see if the target signal of the assignment statement is indeed in the list. If this is the case, the signal assignment statement cannot be penalized because variables cannot be used in process sensitivity lists.

(3) The list of all statements must be checked to see if the target signal of the signal assignment statement in question is being used in a signal attribute. If this is the case, the statement cannot be penalized because variables cannot be used with signal attributes.

(4) The list of all statements must be checked to see if the target signal of the signal assignment statement is being used as a parameter in some function. If this is found, the user is prompted for a yes or no answer to the question of whether or not the *formal* parameter (declared in the function) is defined as a constant. If the answer is no then the ERG determines that the formal must be defined as a signal and the statement cannot be penalized because a function would not accept a variable as an *actual* parameter (declared in the model) when the formal parameter is a signal. If the answer is yes then the actual can be a variable or a signal and the ERG continues to search for conditions that may prevent changing the signal to a variable.

(5) The list of all statements must be checked to see if the target signal of the signal assignment statement is being used as a parameter in some procedure. If this is found, the user is prompted for a response to the same question described above in (4) because a procedure, just like a function, does not accept a variable as an actual parameter when the formal parameter is a signal and in this case the signal assignment statement could no be penalized.

4.2.3.2 Inefficient Use of Signal Attribute 'STABLE

As indicated in Chapter 2, using the signal attribute 'STABLE in the form *not Signal'STABLE* is generally considered to be inefficient. This inefficient style is often used in many different VHDL statements. In the ERG, the DLS is used to search for not Signal'STABLE in if-statements, elsif statements, case statements, signal assignment statements, variable assignment statements, wait statements, and assertion

statements. The algorithm used to search for not Signal'STABLE varies according to the kind of statement being searched. However, after the first couple of steps in each different algorithm, the same basic steps can be used. In this algorithm (common steps only - listed below) names that begin with q (e.g. **qFunctor** = operation such as "not") represent nodes or lists in the DLS that can be traversed to obtain more information about the statement in question (This can be seen using the DLS Browser). For example, qArgs provides a list of arguments for any node such as an if-statement or an "and" operation. Also, in the algorithm, *lstnum* represents the current number of lists that have been traversed, and *itmnum* is the current number of items that have been travesed in the current list. Much experience with VTIP is required to develop and understand all of the very complicated algorithms used in the ERG. The algorithm used for searching statements for instances of not Signal'STABLE is less complicated than others. It is presented here to give a very general idea of the algorithm makeup. The algorithm is given in Figure 4.4. Step (1) begins at a DLS Operation node. An Operation in DLS is a node in the DLS tree structure that represents any VHDL operation such as "and", "nand", or "xor".

---

(1) if **qFunctor** is "not" then
    if **qArgs** only has a boolean **Identifier** in the list then
        select **qParent** and go to (3).
    else go to (5).
else if qArgs does not contain an **Operation** then
    if qParent is an Operation then
        select qParent and go to (3).
    else go to (2).
else go to (2).

(2) Select qArgs, increment lstnum, itmnum(lstnum) = 1.
if the current item in the list is an Operation then
select the item and go to (1).
else go to (3).

(3) if the current item is not the last item in the list then
choose the next item.
if the current item is an Operation then
increment itmnum(lstnum), select item, and go to (1).
else increment itmnum(lstnum) and go to (3).
else select an item that is an Operation and go to (4).

(4) Select qParent(qParent) of current node.
if it is an Operation then
decrement lstnum, select qArgs and go to (3)
else **STOP**.

(5) Select qArgs.
if the current item in the list is an Operation then
select and go to (1).
else go to (6).

(6) if the current item in the list is an **AttributeOp** then
select the item and go to (7).
else select qParent, decrement lstnum, select qArgs, and go to (3).

(7) Select **qData**.
if **qText** = "STABLE" for the second item then
go to (8).
else select qParent, decrement lstnum, select qArgs, and go to (3).

(8) Select "**Measurement**" in the list.
if **qIntval4(qQuantity)** = 0 then ***WARNING !*** and **STOP**.
else go to (3).

Figure 4.4 The Algorithm that Searches for not Signal'STABLE

In step (1), if qFunctor of the Operation node is "not" and the qArgs list (list of items that come after the "not" operation) contains something other than just a boolean identifier then the algorithm branches to step (5) where the process of searching for the 'STABLE attribute can begin since "not" has already been identified. In step (2) since "not" wasn't found, the algorithm transfers control to another node on the DLS tree by selecting the qArgs list. If the current item in this list is an Operation, control is passed back to step (1) to see if this new Operation is "not". In step (3) the next item (after the current item) is chosen and if it is an Operation, control is transferred back to step (1) to see if this new Operation is "not". Step (4) transfers control back up the DLS tree by selecting qParent twice. This is done because control had been passed to a bottom branch of the DLS tree structure and not Signal'STABLE was not found. Step (5) receives control from Step (1) where "not" was found. If the current item in the qArgs list of the current node is not an Operation, control is passed to step (6) where the search for 'STABLE continues. In (6), if the current item in the list is an AttributeOp (identifier for all attributes in the DLS), it is selected and control is passed to step (7). In step (7), the AttributeOp is checked to see if it is 'STABLE by checking qText. In (8), if 'STABLE was found, then the "MEASUREMENT" item in the list is selected to see if there is a time specified with the 'STABLE attribute (e.g. not Signal'STABLE(10 ns) - meaning Signal was not stable for the duration of the last 10 ns). If a time is specified, then no warning can be given, because 'EVENT cannot be used to perform the same function. However, if the time is zero, a warning can be issued for using not Signal'STABLE instead of Signal'EVENT.

4.2.3.3 Inefficient Use of Objects of Type Bit-Vector

As discussed in Chapter 2, using integers in models is more efficient than using the binary equivalent in the form of bit-vectors. The algorithm in the ERG that searches for the inefficient use of bit-vectors penalizes if statements, elsif statements, case statements, variable assignment statements, signal assignment statements, procedure calls, and while loops for this violation. The algorithm considers all ports, internal signals, and variables in the model that are of type bit-vector in the search for inefficiency. In the algorithm, the current CFG node that is being examined for inefficiency is probed for all possible bit-vectors. If a bit-vector is found, several conditions must be met before the bit-vector can be considered inefficient and the CFG node can be penalized. The conditions are as follows:

(1) All statements in the model are checked to see if the bit-vector in question (bv(m to n)) is used in a statement where the range of bits is less than the defined range (e.g. bv((m + 1) to n)). In this case no penalty can be given because the modeler is using selected bits of the bit-vector and integers would not be applicable.

(2) All variable assignment statements, signal assignment statements, if statements, and elsif statements in the model are checked to see if the bit-vector is being assigned or compared to other objects of type bit-vector. In this case, no penalty can be given, because an object of type integer cannot be assigned to or compared to an object of type bit-vector. Only when the bit-vector is

assigned or compared to some literal (e.g. bv <= "1000") can the node possibly be penalized for not using an integer.

(3) All variable assignment statements, signal assignment statements, if statements, and elsif statements in the model are checked to see if the bit-vector is used in association with some operator other than "=" that would prevent the use of an integer. For example in the expression, bv and "1000", an integer cannot be used because of the 'and' operator.

(4) The entire model is probed for function calls that have the bit-vector in question as a parameter. In this case, the CFG node can be penalized only if the sole purpose of the function is to perform an arithmetic operation on the bit-vector and the bit-vector is the only parameter. In this instance an integer could have been used and the function call would have been avoided. The same rules hold true for procedure calls. This is another case where the user is prompted for a yes or no answer. The question is whether or not the function or procedure performs only an arithmetic operation on the bit_vector.

4.2.3.4 Inefficient Use of a Series of If-endif Statements

As discussed in Chapter 2, case statements are more efficient than an equivalent series of if-endif statements. In the ERG, there exists an algorithm that searches for if-then statements that are a part of a series of if-then statements that could be represented in the form of a case statement. The CFG node that represents an if-then statement that fits this category is penalized based upon certain conditions. The algorithm is given in

Figure 4.5. The algorithm begins at any if statement node in the DSL tree. Step (1) is begun by a check for only one statement list associated with the if statement. This is done because each elsif or else associated with an original if statement are listed as statement lists in the DLS. Penalties are not given when elsif or else are present, therefore the only statement list allowed is the statement list that represents the if statement. Secondly, in this step a list of all statements that are at the same level as the

---

(1) If there exists one and only one **statement list** for the if statement then
select the if statement.
Create a list (named STMTS) of all statements on the level of the if statement.
If the if statement contains the "=" operator in its control expression then
store the contents of the control expression of the if statement in memory and go to (2).
else **STOP**.
else **STOP**.

(2) Check the previous statement (if fwd = 0) or the next statement (if fwd = 1) in STMTS for another if statement.
If search is NULL then
if search was backwards then
set fwd = 1, go to (2).
else **STOP**.
else if the statement is a Wait statement then
if search was backwards then
set fwd = 1, go to (2).
else **STOP**.
else if the statement is an if statement then
if there exists only one statement list for the if statement then
store the contents of the control expression in memory.
compare the control expressions of if-statement1 (stored in step 1) and if-statement2.
if a match is found then
PENALTY! and **STOP**.

else go to (2).
else go to (2).
else go to (2).

Figure 4.5 The Algorithm that Searches for an Inefficient If-endif Series

if statement is created for the search for other if statements. This is done because two if statements that are at different levels (e.g. nested within different loops) cannot be combined to form a case statement. Finally, in this step, if the necessary conditions are met the contents of the control expression of the if statement is stored in memory for later comparison to other if statements. In step 2, the list named STMTS is probed for other if statements that can be combined with the original if statement to form an equivalent case statement. If another if statement is found that has the same control expression except for values being checked for on one side of equal operators, then a penalty is given. However, if a wait statement is found between the two if statements, no penalty can be given because the wait statement causes the two if statements to be executed at different times and in this situation a case statement cannot be used. The appendix contains the actual programming code that was used to implement this algorithm.

4.2.3.5 Use of a Dynamic Sensitivity List

Dynamic sensitivity lists are located at the end of processes in the form of wait statements. As mentioned in Chapter 2, the ERG does not penalize for using dynamic sensitivity lists but gives a warning that static sensitivity lists should be used instead

when possible. The algorithm that searches for dynamic sensitivity lists that could be static sensitivity lists issues a warning based upon the following conditions:

(1) The only wait statement in the model is the last statement in the list of all statements. If there are other wait statements in the process, a static sensitivity list cannot be used.

(2) If the wait statement is in the form "wait on" with only a list of signals following, then a warning message can be given for unnecessary dynamic sensitivity list.

(3) If the wait statement expression is in a form that can be placed in a static sensitivity list then the warning message can be given. For example, "wait until A'EVENT or B'EVENT" could be executed by simply placing A and B in a static sensitivity list. However, the statement "wait until A'EVENT and B'EVENT" cannot be expressed in a static sensitivity list (operators are not allowed in static sensitivity lists). The algorithm checks for all conditions that are similar to those discussed here.

4.2.3.6 File I/O

As explained in Chapter 2, there are certain types of models that simulate much faster when using file I/O as opposed to using signal assignments in a test bench. Specifically, it was discovered that models whose sensitivity list is only made up of a signal whose purpose is to enable or disable the process simulate much faster using file

I/O. In the ERG, there exists an algorithm that probes a model to see if it fits the category of a model that would simulate much faster using file I/O. The algorithm begins by checking to see if the model employs file I/O by searching for procedure calls that are used in file I/O to read input vectors. If the model does not use file I/O, then the sensitivity list of the model is examined to determine if the necessary conditions for deducting points in the rating have been met. When a single signal is detected in the sensitivity list, the user is prompted for a yes or no answer to the question of whether or not the purpose of the signal is simply to enable or disable the process. If the answer is yes, file I/O would be more efficient as described above.

4.2.3.7 Inefficient Use of Resolved Signals

As discussed in Chapter 2, using resolved signals is an inefficient coding style. There exists an algorithm in the ERG that searches for resolution function calls in variable and signal assignment statements and points are deducted from the rating of the model if they are found.

4.2.4 Program Output

The output of the ERG includes the control flow information discussed in section 4.2.2, a numerical rating for each node, detailed explanation of the problems at each inefficient node, suggestions for improving each inefficient node, node coverages (dynamic rating only), a static rating, and a dynamic rating. The output of the ERG in the rating of the parallel-to-serial converter model in Figure 4.2 is given in Figure 4.6. In this example, the input options chosen were the Vantage simulator and a dynamic

rating. As can be seen in the figure, nodes 1 and 5 are penalized for inefficiency and the overall dynamic rating is 97. A block diagram of the ERG is given in Figure 4.7. Chapter 5 contains many examples that demonstrate the effectiveness of the tool.

CFG Node 0 (Line #13)

- True edge points to CFG Node 1 (Line #14).
- False edge points to CFG Node 2 (Line #15).

- node coverage = 1837

- node efficiency rating = 100

CFG Node 1 (Line #14)

- True edge points to the end of process code.
- False edge is undefined.

Inefficient signal used as destination in the assignment statement - could use a variable instead

- node coverage = 102

- node efficiency rating = 80

CFG Node 2 (Line #15)

- True edge points to CFG Node 3 (Line #16).
- False edge points to the end of process code.

- node coverage = 1735

- node efficiency rating = 100

CFG Node 3 (Line #16)

- True edge points to CFG Node 4 (Line #17).
- False edge points to the end of process code.

- node coverage = 816

- node efficiency rating = 100

CFG Node 4 (Line #17)

- True edge points to CFG Node 5 (Line #18).
- False edge is undefined.

- node coverage = 816

- node efficiency rating = 100

CFG Node 5 (Line #18)

- True edge points to the end of process code.
- False edge is undefined.

Inefficient signal used as destination in the assignment statement - could use a variable instead

- node coverage = 816

- node efficiency rating = 80

MODEL DYNAMIC EFFICIENCY RATING = 97

Figure 4.6 The ERG Output - Rating the Parallel-to-Serial Converter

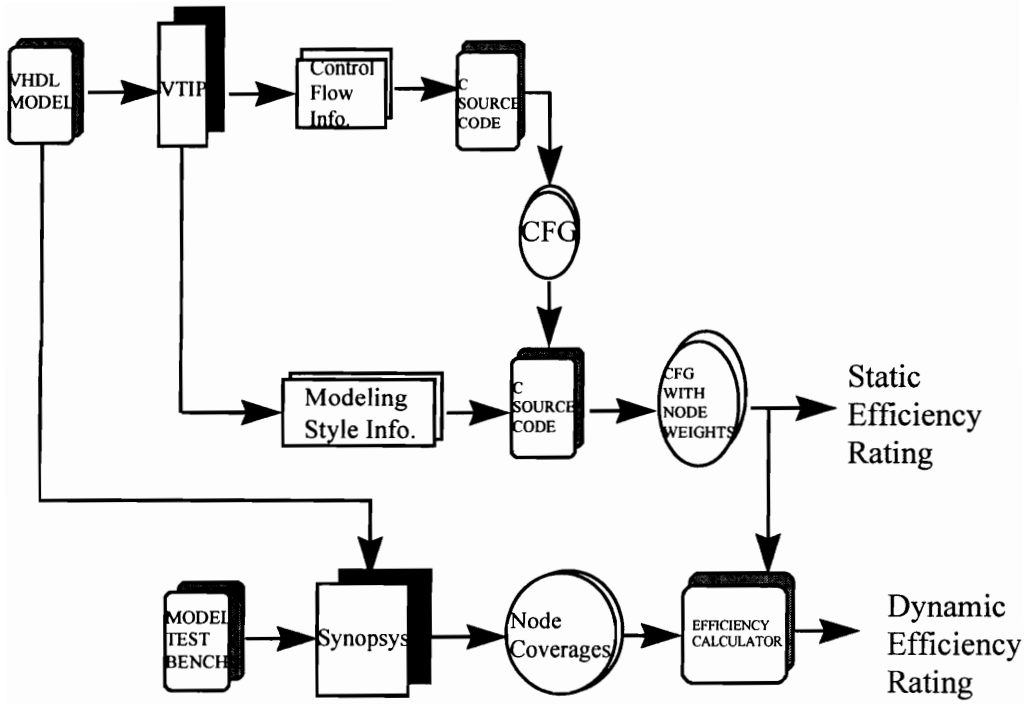

Figure 4.7 Block Diagram of the Efficiency Rating Generator (ERG)

Chapter 5. Rating the Efficiency of Example Models

This chapter contains an explanation of five models that demonstrate the effectiveness of the tool. A description of each model is given along with portions of the model code and portions of the corresponding ERG output.

5.1 MODELD

MODELD is a basic VHDL model that demonstrates the effectiveness of the ERG. The inefficient version of this model, MODELDi, is shown in Figure 5.1. The

```
1 use work.all;
2 entity MODELDi is
3   port(I: in BIT_VECTOR(2 downto 0); EN: in BIT; Q: out INTEGER);
4 end MODELDi;
5
6 architecture BEHAVIORAL of MODELDi is
7
8  signal O: INTEGER;
9
10  begin
11
12    P:process
13     begin
14      if EN = '1' then
15       if (I = "000") then
16        O <= 7;
17       end if;
18       if (I = "001") then
19        O <= 6;
20       end if;
21       if (I = "010") then
22        O <= 5;
23       end if;
24       if (I = "011") then
25        O <= 4;
26       end if;
27       if (I = "100") then
28        O <= 3;
29       end if;
30       if (I = "101") then
31        O <= 2;
32       end if;
33       if (I = "110") then
34        O <= 1;
35       end if;
36       if (I = "111") then
37        O <= 0;
38       end if;
39      end if;
40      wait for 5 ns;
```

```
41        Q <= O;
42      wait on I, EN;
43    end process;
44 end BEHAVIORAL;
```

Figure 5.1 The Inefficient Version of MODELD

inefficient characteristics of this model are as follows:

(1) Input port I satisfies the conditions of an object of type bit_vector that could be an integer.

(2) Internal signal O satisfies the conditions of a signal that could be a variable.

(3) The series of if-endif statements found in lines 15 to 36 satisfies the conditions of an if-endif series that could be a case statement.

As discussed earlier, the ERG can be used to gather information about the inefficient characteristics of each node of the model along with suggestions for improvement. Figure 5.2 displays the output when a dynamic efficiency rating (Vantage simulator) is requested for MODELDi. The CFG of the model has 20 nodes total (CFG nodes 0 - 19), and they are displayed in the figure along with the final overall dynamic efficiency rating. For each VHDL statement, the CFG node number, line number, CFG true and false edges, inefficient characteristics, node coverage, and node efficiency rating is given. Nodes 1, 3, 5, 7, 9, 11, 13, and 15 of the figure represent the if statements that are penalized for

inefficiency because a case statement could have been used. These nodes are also penalized for using a bit_vector port that could have been an integer. Nodes 2, 4, 6, 8, 10,

---

CFG Node 0 (Line #14)

- True edge points to CFG Node 1 (Line #15).
- False edge points to CFG Node 17 (Line #40).

- node coverage = 409

- node efficiency rating = 100

CFG Node 1 (Line #15)

- True edge points to CFG Node 2 (Line #16).
- False edge points to CFG Node 3 (Line #18).

Inefficient use of port of type bit_vector - could use integer instead

Inefficient use of if-endif series - could use case

- node coverage = 408

- node efficiency rating = 80

CFG Node 2 (Line #16)

- True edge points to CFG Node 3 (Line #18).
- False edge is undefined.

Inefficient signal used as destination in the assignment statement - could use a variable instead

- node coverage = 51

- node efficiency rating = 80

CFG Node 3 (Line #18)

- True edge points to CFG Node 4 (Line #19).
- False edge points to CFG Node 5 (Line #21).

Inefficient use of port of type bit_vector - could use integer instead

Inefficient use of if-endif series - could use case

- node coverage = 408

- node efficiency rating = 80

CFG Node 4 (Line #19)

- True edge points to CFG Node 5 (Line #21).
- False edge is undefined.

Inefficient signal used as destination in the assignment statement - could use a variable instead

- node coverage = 51

- node efficiency rating = 80

CFG Node 5 (Line #21)

- True edge points to CFG Node 6 (Line #22).
- False edge points to CFG Node 7 (Line #24).

Inefficient use of port of type bit_vector - could use integer instead

Inefficient use of if-endif series - could use case

- node coverage = 408

- node efficiency rating = 80

CFG Node 6 (Line #22)

- True edge points to CFG Node 7 (Line #24).
- False edge is undefined.

Inefficient signal used as destination in the assignment statement - could use a variable instead

- node coverage = 51

- node efficiency rating = 80

CFG Node 7 (Line #24)

- True edge points to CFG Node 8 (Line #25).
- False edge points to CFG Node 9 (Line #27).

Inefficient use of port of type bit_vector - could use integer instead

Inefficient use of if-endif series - could use case

- node coverage = 408

- node efficiency rating = 80

CFG Node 8 (Line #25)

- True edge points to CFG Node 9 (Line #27).
- False edge is undefined.

Inefficient signal used as destination in the assignment statement - could use a variable instead

- node coverage = 51

- node efficiency rating = 80

CFG Node 9 (Line #27)

- True edge points to CFG Node 10 (Line #28).
- False edge points to CFG Node 11 (Line #30).

Inefficient use of port of type bit_vector - could use integer instead

Inefficient use of if-endif series - could use case

- node coverage = 408

- node efficiency rating = 80

CFG Node 10 (Line #28)

- True edge points to CFG Node 11 (Line #30).
- False edge is undefined.

Inefficient signal used as destination in the assignment

statement - could use a variable instead

- node coverage = 51

- node efficiency rating = 80

CFG Node 11 (Line #30)

- True edge points to CFG Node 12 (Line #31).
- False edge points to CFG Node 13 (Line #33).

Inefficient use of port of type bit_vector - could use integer instead

Inefficient use of if-endif series - could use case

- node coverage = 408

- node efficiency rating = 80

CFG Node 12 (Line #31)

- True edge points to CFG Node 13 (Line #33).
- False edge is undefined.

Inefficient signal used as destination in the assignment statement - could use a variable instead

- node coverage = 51

- node efficiency rating = 80

CFG Node 13 (Line #33)

- True edge points to CFG Node 14 (Line #34).
- False edge points to CFG Node 15 (Line #36).

Inefficient use of port of type bit_vector - could use integer instead

Inefficient use of if-endif series - could use case

- node coverage = 408

- node efficiency rating = 80

CFG Node 14 (Line #34)

- True edge points to CFG Node 15 (Line #36).
- False edge is undefined.

Inefficient signal used as destination in the assignment statement - could use a variable instead

- node coverage = 51

- node efficiency rating = 80

CFG Node 15 (Line #36)

- True edge points to CFG Node 16 (Line #37).
- False edge points to CFG Node 17 (Line #40).

Inefficient use of port of type bit_vector - could use integer instead

Inefficient use of if-endif series - could use case

- node coverage = 408

- node efficiency rating = 80

CFG Node 16 (Line #37)

- True edge points to CFG Node 17 (Line #40).
- False edge is undefined.

Inefficient signal used as destination in the assignment statement - could use a variable instead

- node coverage = 51

- node efficiency rating = 80

CFG Node 17 (Line #40)

- True edge points to CFG Node 18 (Line #41).
- False edge is undefined.

- node coverage = 409

- node efficiency rating = 100

CFG Node 18 (Line #41)

- True edge points to CFG Node 19 (Line #42).
- False edge is undefined.

- node coverage = 409

- node efficiency rating = 100

CFG Node 19 (Line #42)

- True edge points to the end of process code.
- False edge is undefined.

- node coverage = 409

- node efficiency rating = 100

MODEL DYNAMIC EFFICIENCY RATING = 86

Figure 5.2 The ERG Output for the Rating of MODELDi

12, 14, and 16 are the signal assignment statements that are penalized because they could have been variable assignment statements. The efficient version of the model, MODELDe, is given in Figure 5.3. As shown in Figure 5.2, the overall dynamic efficiency rating for MODELDi is 86. This means that when using the Vantage simulator the efficient model should simulate approximately 14% (100 - 86) faster than the inefficient model. Both models were simulated 10 times (this was done for all examples models discussed in this chapter and Chapter 6) to see what the actual results would be. Also, as discussed in Chapter 2 the test bench was created in a manner to insure that the models were simulated long enough to see a difference in performance (this was done for

all example models discussed in this chapter and chapter 6). The average time of the 10 simulations in seconds for MODELDe was 2.40 s and for MODELDi it was 2.80 s. This

```
1 use work.all;
2 entity MODELDe is
3   port(I: in INTEGER; EN: in BIT; Q: out INTEGER);
4 end MODELDe;
5
6 architecture BEHAVIORAL of MODELDe is
7
8  begin
9
10    P:process
11      variable O: INTEGER;
12     begin
13      if EN = '1' then
14        case I is
15          when 0 => O := 7;
16          when 1 => O := 6;
17          when 2 => O := 5;
18          when 3 => O := 4;
19          when 4 => O := 3;
20          when 5 => O := 2;
21          when 6 => O := 1;
22          when 7 => O := 0;
23          when others => NULL;
24        end case
25       end if;
26      wait for 5 ns;
27      Q <= O;
28     wait on I, EN;
29    end process;
30 end BEHAVIORAL;
```

Figure 5.3 The Efficient Version of MODELD

result yields a 14.29% actual speedup for the efficient model using Vantage. The predicted speedup was 14. With Synopsys, the average simulation time for MODELDe was 4.61s and for MODELDi it was 5.05 s. This result indicates that there is a 8.71% actual speedup with the efficient model. The dynamic rating with Synopsys was 92 which means that the predicted speedup is 8%.

5.2 An Adder Model

A VHDL model of an adder was created to demonstrate how the ERG evaluates models that could have employed file I/O. As mentioned in Chapter 2 (Section 2.11), models that receive input vectors periodically in a loop simulate faster using file I/O as opposed to using signal assignments in a test bench. The inefficient adder model, ADDERi, is shown in Figure 5.4. Ports A, B, and CIN (carry-in bit) are added and produce outputs SUM and COUT (carry-out bit). In this model, the output is calculated periodically in a loop based upon the value of the generic time delay PER. The input

---

```
1 use work.all;
2 entity ADDERi is
3   generic (ADD_DEL, PER: TIME);
4   port (EN, CIN: in BIT; A, B: in BIT_VECTOR(3 downto 0);
5         SUM: out BIT_VECTOR(3 downto 0); COUT: out BIT);
6 end ADDERi;
7
```

```
8 architecture BEHAVIORAL of ADDERi is
9
10 begin
11  process
12    variable CARRY: BIT := '0';
13   begin
14    wait on EN until EN = '1';
15    while EN = '1' loop
16     CARRY := CIN;
17     for I in 0 to 3 loop
18      SUM(I) <= A(I) xor B(I) xor CARRY after ADD_DEL;
19      CARRY := ((A(I) and B(I)) or (A(I) and CARRY) or
20                    (B(I) and CARRY));
21     end loop;
22     COUT <= CARRY after ADD_DEL;
23     wait for PER;
24    end loop;
25  end process;
26 end BEHAVIORAL;
```

Figure 5.4 The Inefficient Adder Model

values are defined in the test bench of the model. This model is inefficient because the input ports could be driven using file I/O. In this case the model would simulate faster. The ERG output for this model using the Vantage simulator is given in Figure 5.5. Before the output is generated the user is prompted for a yes or no answer concerning input signal EN, since the process must wait on the signal. The question is whether or not this signal serves as only an enable signal for the process. In this case, the answer is yes. This must be determined because if this signal serves some other purpose then the conditions for using file I/O may not apply here (explained in Chapter 2, Section 2.11). At CFG node 0 in the figure, it is determined that a loop is being used to periodically read input

vectors from a test bench when file I/O could be used. No penalty is given at this particular node because it is not the node that is inefficient but the entire model as a whole. Therefore the penalty is deducted from the overall rating of the model. The rating of each node in the model is 100, but the penalty for using a test bench for driving ports when file I/O could be used is 15 using Vantage. Therefore the overall dynamic

---

CFG Node 0 (Line #14)

- True edge points to CFG Node 1 (Line #15).
- False edge is undefined.

Inefficient model that must be driven by a test_bench - could use file I/O (points will be deducted from overall rating)

- node coverage = 2

- node efficiency rating = 100

CFG Node 1 (Line #15)

- True edge points to CFG Node 2 (Line #16).
- False edge points to the end of process code.

- node coverage = 601

- node efficiency rating = 100

CFG Node 2 (Line #16)

- True edge points to CFG Node 3 (Line #17).
- False edge is undefined.

- node coverage = 600

- node efficiency rating = 100

CFG Node 3 (Line #17)
- True edge points to CFG Node 4 (Line #18).
- False edge points to CFG Node 6 (Line #22).

- node coverage = 3000

- node efficiency rating = 100

CFG Node 4 (Line #18)

- True edge points to CFG Node 5 (Line #19).
- False edge is undefined.

- node coverage = 2400

- node efficiency rating = 100

CFG Node 5 (Line #19)

- True edge points to CFG Node 3 (Line #17).
- False edge is undefined.

- node coverage = 2400

- node efficiency rating = 100

CFG Node 6 (Line #22)

- True edge points to CFG Node 7 (Line #23).
- False edge is undefined.

- node coverage = 600

- node efficiency rating = 100

CFG Node 7 (Line #23)

- True edge points to CFG Node 1 (Line #15).
- False edge is undefined.

- node coverage = 600

- node efficiency rating = 100

MODEL DYNAMIC EFFICIENCY RATING = 85

Figure 5.5 The ERG Output for the Rating of ADDERi

efficiency rating for the model is 85. The efficient model of the adder, ADDERe, is given in Figure 5.6. This model employs file I/O to provide the input vectors for ports A, B, and

---

```
1 use STD.TEXTIO.all, work.USER_TYPES.all, work.all;
2 entity ADDERe is
3   generic (ADD_DEL, PER: TIME);
4   port (EN: in BIT;
5         SUM: out BIT_VECTOR(3 downto 0); COUT: out BIT);
6 end ADDERe;
7
8 architecture TIO of ADDERe is
9
10 begin
11   process
12     variable VLINE: LINE;
13     variable A, B: BIT_VECTOR(3 downto 0);
14     variable CIN: BIT;
15     variable CARRY: BIT := '0';
16     file INVECT: TEXT is "ADDRVEC.TXT";
17    begin
18      wait on EN until EN = '1';
19     while not (ENDFILE(INVECT)) loop
20       READLINE(INVECT, VLINE);
21       READ(VLINE, A);
22       READ(VLINE, B);
23       READ(VLINE, CIN);
```

```
24        CARRY := CIN;
25        for I in 0 to 3 loop
26          SUM(I) <= A(I) xor B(I) xor CARRY after ADD_DEL;
27          CARRY := ((A(I) and B(I)) or (A(I) and CARRY) or
28                (B(I) and CARRY));
29        end loop;
30        COUT <= CARRY after ADD_DEL;
31        wait for PER;
32      end loop;
33  end process;
34 end TIO;
```

Figure 5.6 The Efficient Adder Model

CIN. Both models were simulated to determine what the actual speedup would be with file I/O. The efficient model had an average simulation time of .70 s, and the inefficient model had an average simulation time of .89 s which yields a 21.35% speedup. The predicted speedup was 15%. With Synopsys, the average simulation time for the efficient model was 6.52 s, and the average simulation time for the inefficient model was 7.77 s. This means there was 16.09% speedup and the rating was 80 which means the predicted speedup was 20%.

5.3 UART Model

A model of a UART (Universal Asynchronous Receiver/Transmitter) was developed to further test the ERG. This UART functions as a parallel-serial/serial-parallel converter. The inefficient version of the UART, UARTi, is shown in Figure 5.7 [3].

```
1 use work.USER_TYPES.all, work.all;
2 entity UARTi is
3  generic (CLK_PER, ODEL, INDEL, INTDEL: TIME);
4  port (I: in MVL4; LOAD, READ: in BIT;
5       DATA: inout BUS1(7 downto 0) := "ZZZZZZZZ";
6       NINTO, NINTI: out BIT; O: out MVL4);
7 end UARTi;
8
9 architecture BEHAVIORAL of UARTi is
10
11  signal ICLK, OCLK, ISTRT: BIT;
12  signal NINTI1, NINTI2: BIT := '1';
13  signal I_FLAG: BOOLEAN := TRUE;
14  signal IREG, OREG: MVL4_VECTOR(7 downto 0);
15  signal ICNTR, OCNTR: INTEGER;
16
17
18 begin
19
20  P: process(LOAD, OCLK, I, ISTRT, ICLK, READ)
21
22     begin
23
24 --
25 -- OUTPUT
26 --
27    if (LOAD'EVENT or OCLK'EVENT) then
28      if (LOAD'EVENT and LOAD = '1') then
29        OREG <= SENSE(DATA, '1');
30        OCNTR <= 7;
31        NINTO <= '1' after INTDEL;
32        O <= '0' after ODEL;
33        OCLK <= not OCLK after CLK_PER;
34      end if;
35
36      if (OCLK'EVENT) then
37        if (OCNTR /= -1) then
38          O <= OREG(OCNTR) after ODEL;
39          OCNTR <= OCNTR - 1;
40          OCLK <= not OCLK after CLK_PER;
41        else
42          O <= '1' after ODEL;
43          NINTO <= '0' after INTDEL;
44        end if;
45      end if;
46    end if;
```

```
47 --
48 -- INPUT
49 --
50    if (I'EVENT or ISTRT'EVENT or ICLK'EVENT) then
51      if (I_FLAG) then
52        if (I'EVENT and I = '0') then
53          ISTRT <= '1' after CLK_PER/2;
54        end if;
55      end if;
56
57      if (ISTRT'EVENT and I = '0') then
58        I_FLAG <= FALSE;
59        ISTRT <= '0';
60        ICNTR <= 7;
61        ICLK <= not ICLK after CLK_PER;
62      end if;
63
64      if (ICLK'EVENT) then
65        if (ICNTR /= -1) then
66          IREG(ICNTR) <= I;
67          ICNTR <= ICNTR - 1;
68          ICLK <= not ICLK after CLK_PER;
69        else
70          NINTI1 <= '0' after INTDEL;
71          I_FLAG <= TRUE;
72        end if;
73      end if;
74    end if;
75 --
76 -- READ
77 --
78    if (READ'EVENT) then
79      if (READ = '1') then
80        DATA <= DRIVE(IREG) after INDEL;
81        NINTI2 <= '1' after INTDEL;
82      else
83        DATA <= "ZZZZZZZZ" after INDEL;
84      end if;
85    end if;
86 --
87 -- INTERRUPTS
88 --
89    if (NINTI1'EVENT or NINTI2'EVENT) then
90      if (NINTI1'EVENT) then
91        NINTI <= NINTI1;
92      elsif (NINTI2'EVENT) then
93        NINTI <= NINTI2;
94      end if;
95    end if;
```

```
96  end process;
97 end BEHAVIORAL;
```

Figure 5.7 The Inefficient UART Model

LOAD is the input control signal, DATA is used to pass input and output register values, and OREG is the output register. O is the serial output, OCLK is the internal output shift clock, NINTO is the interrupt output signal, and I is the serial input. In UARTi, objects OREG, OCNTR, I_FLAG, I_CNTR, and I_FLAG are declared as signals but could have been declared as variables. Input port DATA is a signal that must be resolved (Line 80, Figure 5.7). This signal must be resolved because it is being driven in the model and in the test bench since it is used to pass input and output register values. This could be avoided by using multiple unidirectional signals to perform the same function as the resolved signal. A portion of the output of the ERG (CFG nodes 31 - 42) when a dynamic rating is requested for UARTi using Vantage is given in Figure 5.8. CFG Nodes 2, 3, 10,

```
---------------------------------------------------------------
CFG Node 31  (Line #71)

 - True edge points to CFG Node 32  (Line #78).
 - False edge is undefined.

Inefficient signal used as destination in the assignment
statement - could use a variable instead

 - node coverage = 202

 - node efficiency rating = 80

---------------------------------------------------------------
```

CFG Node 32 (Line #78)

- True edge points to CFG Node 33 (Line #79).
- False edge points to CFG Node 38 (Line #89).

- node coverage = 4950

- node efficiency rating = 100

CFG Node 33 (Line #79)

- True edge points to CFG Node 34 (Line #80).
- False edge points to CFG Node 36 (Line #82).

- node coverage = 404

- node efficiency rating = 100

CFG Node 34 (Line #80)

- True edge points to CFG Node 35 (Line #81).
- False edge is undefined.

Inefficient use of resolved signal - could avoid by using extra signals (points will be deducted from overall rating)

- node coverage = 202

- node efficiency rating = 100

CFG Node 35 (Line #81)

- True edge points to CFG Node 38 (Line #89).
- False edge is undefined.

- node coverage = 202

- node efficiency rating = 100

CFG Node 36 (Line #82)

- True edge points to CFG Node 37 (Line #83).
- False edge is undefined.

- node coverage = 202

- node efficiency rating = 100

CFG Node 37 (Line #83)

- True edge points to CFG Node 38 (Line #89).
- False edge is undefined.

- node coverage = 202

- node efficiency rating = 100

CFG Node 38 (Line #89)

- True edge points to CFG Node 39 (Line #90).
- False edge points to the end of process code.

- node coverage = 4950

- node efficiency rating = 100

CFG Node 39 (Line #90)

- True edge points to CFG Node 40 (Line #91).
- False edge points to CFG Node 41 (Line #92).

- node coverage = 1

- node efficiency rating = 100

CFG Node 40 (Line #91)

- True edge points to the end of process code.
- False edge is undefined.

- node coverage = 1

- node efficiency rating = 100

CFG Node 41 (Line #92)

- True edge points to CFG Node 42 (Line #93).
- False edge points to the end of process code.

- node coverage = 0

- node efficiency rating = 100

```
---------------------------------------------------------------
CFG Node 42  (Line #93)

- True edge points to the end of process code.
- False edge is undefined.

- node coverage = 0

_____________________________________________________________

MODEL DYNAMIC EFFICIENCY RATING = 88

_____________________________________________________________
```

Figure 5.8 The ERG Output for the Rating of UARTi (Nodes 31 - 42)

20, 22, 27, and 31 are all signal assignment statements that could have been variable assignment statements and are penalized accordingly. Node 31 is shown in the figure. At Node 34, the ERG detects a resolved signal that could have been avoided. The penalty is applied to the overall rating instead of the node because of the greater than normal effect resolved signals have on simulation performance. As can be seen in the figure, the overall efficiency rating of the model is 88. In the efficient model, UARTe, the suggested corrections were made, and both models were simulated to see what the actual speedup is. The average simulation time of the efficient model was 2.28 s. The average simulation time of the inefficient model was 2.52 s which means that there is a 9.52% speedup. The predicted speedup was 12%. With Synopsys, the dynamic efficiency rating was 84. The efficient model had an average simulation time of 31.51 s, and the inefficient model had an average simulation time of 36.43 s. This yields a speedup of 13.51% and the prediction was 16%.

5.4 The MARK2 Processor Model

The MARK2 is a processor that has a small instruction set but still contains the essential elements of most general-purpose processors [3]. The word length of the MARK2 is 8 bits, where 5 bits are used for the address and 3 bits are used for the instruction opcode. The instructions are listed in Figure 5.9. An inefficient model of the MARK2 was developed (MARK2i) and a portion of it (Lines 1 to 26, 87 to 109) is shown in Figure 5.10. The model has a total of 250 lines of code. The portion of code shown in the figure is the declaration section (lines 1 to 26) and a part of the execute cycle section (lines 87 to 109) of the model. In the figure, the instruction register (IR) is being checked

Instruction	Opcode	Description
JMP	000	absolute jump
TCA	001	twos complement of the accumulator
LDA	010	load the accumulator
STA	011	store the accumulator contents
ADD	100	add the addressed operand to the accum.
INT	101	interrupt control
JPN	110	jump if accumulator negative
STP	111	stop

Figure 5.9 Instructions for the MARK2

for the load (010) and store (011) instructions. During this execution cycle section, inefficient if statements are being used to check the instruction register for all different

instruction values. In this case, a case statement would be more efficient. More inefficiency can be found in the model with objects INTE (interrupt signal) and IR which are declared as signals but could be variables. Therefore, every signal assignment statement in the model that has either of these signals as the destination is inefficient and could be a variable assignment statement. The final inefficient characteristic of the model is the use of resolved signal DATA. DATA is used to pass information to input and

---

```
1 use work.fncMARK2.all, work.USER_TYPES.all, work.all;
2 entity MARK2i is
3  generic (RDEL, WDEL, ODEL, MADEL, INTDEL, PER: TIME);
4  port (RUN, INT: in BIT; FETCH, EXECUTE: in BOOLEAN;
5       DATA: inout BUS1(7 downto 0) := "ZZZZZZZZ";
6       MA: out BIT_VECTOR(4 downto 0);
7       RD, WRITE, IO, RDS, RDP, WTS, WTP, INTA: out BIT);
8 end MARK2i;
9
10 architecture BEHAVIORAL of MARK2i is
11
12  signal STOP, STOPR, STOPE: BIT;
13  signal INTE, INTER, INTEE, INTEI: BIT;
14  signal TEMP_RD, RDF, RDE: BIT;
15  signal WRITEF, WRITEE: BIT;
16  signal IOWAIT, IOWAITF, IOWAITE, IOWAITI: BIT;
17  signal TEMP_MA, MAF, MAE: BIT_VECTOR(4 downto 0);
18  signal CLK, INTERUPT: BOOLEAN;
19  signal PCF, PCI, PCE, PC: BIT_VECTOR(4 downto 0);
20  signal IR: BIT_VECTOR(7 downto 0);
21  signal ACC: MVL4_VECTOR(7 downto 0);
22
23 begin
24
25  P: process
26      begin
          .
          .
          .
87          if (IR(7 downto 5) = "010") then
88            MAE <= IR(4 downto 0);                    --lda
89            RDE <= '1' after ODEL;
90            WRITEE <= '0' after ODEL;
91            IOWAITE <= '1';
```

```
92          wait for RDEL;
93          RDE <= '0' after ODEL;
94          IOWAITE <= '0';
95          ACC <= SENSE(DATA, '1');
96        end if;
97
98        if (IR(7 downto 5) = "011") then
99          DATA <= transport DRIVE(ACC) after ODEL;          --sta
100         DATA <= transport "ZZZZZZZZ" after 3 * ODEL;
101         DATA <= transport "ZZZZZZZZ" after (3 * ODEL) + WDEL;
102         MAE <= IR(4 downto 0) after MADEL;
103         RDE <= '0' after ODEL;
104         WRITEE <= '1' after ODEL;
105         IOWAITE <= '1';
106         wait for WDEL;
107         WRITEE <= '0' after ODEL;
108         IOWAITE <= '0';
109       end if;
```

Figure 5.10 The Inefficient MARK2 Model (Lines 1 to 26, 87 to 109)

output registers. It is defined as a resolved signal so it can be driven in the test bench and in the model. However, this could be avoided by using extra signals. The ERG detects all of the inefficiency and provides the appropriate rating. A portion of the ERG output for the rating of MARK2i CFG nodes (30 to 40) with the Synopsys option is shown in Figure 5.11. The CFG of MARK2i has a total of 136 nodes. Nodes 30 and 31 correspond to the

---

```
-------------------------------------------------------------
CFG Node 30  (Line #87)

- True edge points to CFG Node 31  (Line #88).
- False edge points to CFG Node 39  (Line #98).

Inefficient use of if-endif series - could use case

- node coverage = 18

- node efficiency rating = 98
```

CFG Node 31 (Line #88)

- True edge points to CFG Node 32 (Line #89).
- False edge is undefined.

- node coverage = 6

- node efficiency rating = 100

CFG Node 32 (Line #89)

- True edge points to CFG Node 33 (Line #90).
- False edge is undefined.

- node coverage = 6

- node efficiency rating = 100

CFG Node 33 (Line #90)

- True edge points to CFG Node 34 (Line #91).
- False edge is undefined.

- node coverage = 6

- node efficiency rating = 100

CFG Node 34 (Line #91)

- True edge points to CFG Node 35 (Line #92).
- False edge is undefined.

- node coverage = 6

- node efficiency rating = 100

CFG Node 35 (Line #92)

- True edge points to CFG Node 36 (Line #93).
- False edge is undefined.

- node coverage = 6

- node efficiency rating = 100

CFG Node 36 (Line #93)

- True edge points to CFG Node 37 (Line #94).
- False edge is undefined.

- node coverage = 6

- node efficiency rating = 100

CFG Node 37 (Line #94)

- True edge points to CFG Node 38 (Line #95).
- False edge is undefined.

- node coverage = 6

- node efficiency rating = 100

CFG Node 38 (Line #95)

- True edge points to CFG Node 39 (Line #98).
- False edge is undefined.

- node coverage = 6

- node efficiency rating = 100

CFG Node 39 (Line #98)

- True edge points to CFG Node 40 (Line #99).
- False edge points to CFG Node 50 (Line #111).

Inefficient use of if-endif series - could use case

- node coverage = 18

- node efficiency rating = 98

CFG Node 40 (Line #99)

- True edge points to CFG Node 41 (Line #100).
- False edge is undefined.

Inefficient use of resolved signal - could avoid by using
extra signals (points will be deducted from overall rating)

- node coverage = 6

- node efficiency rating = 100

Figure 5.11 The ERG Output for the Rating of MARK2i (Nodes 30 - 40)

if statements in Figure 5.10 and are penalized for not using a case statement. At node 40, signal DATA is identified as a resolved signal and the penalty is applied with the overall rating, as mentioned with earlier examples. The overall dynamic rating for MARK2i (Synopsys) is 84 which means predicted speedup is 16%. An efficient model was developed and simulated along with the inefficient model. The average simulation time for the efficient model was 1.65 s, and the average simulation time of the inefficient model was 2.17 s which indicates there is a 24% actual speedup. With Vantage, the dynamic rating was 89 which means predicted speedup is 11%. The average simulation time of the efficient model was .75 s and the average simulation time of the inefficient model was .94 s which means the actual speedup is 20%.

5.5 The AMD2910 Micro-controller Model

The AMD2910 is a micro-controller that generates control signals which control the actions of other system components [15]. The AMD2910 contains an instruction

decoder, a stack and its pointers, a multiplexer, a controlled up-counter, and a controlled down-counter. An inefficient VHDL model of the AMD2910 was developed (AMD2910i) and a portion of it (lines 1 - 74) is shown in Figure 5.12. The figure includes the declaration section of the model and the stack pointer assignment section of the

---

```
1 use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
2 -- ***********************************************************
3 entity AMD2910i is
4  port (G_RESET, CLKIN: in BIT; CARRYIN, LOAD, CCEN, CC: in MVL4;
5       INSTRUCTION: in BIT_VECTOR(0 to 3); DATAIN: in MVL4_VECTOR(1 to 12);
6       OUTADDR: inout MVL4_VECTOR(1 to 12); FULL, INTERENABLE, MAPPINENABLE,
7       PIPENABLE: out MVL4);
8 end AMD2910i;
9 -- ***********************************************************
10
11 architecture BEHAVIORAL of AMD2910i is
12
13  signal STACK_CNTL: BIT_VECTOR(0 to 1);
14  signal UPC_CNTL: BIT;
15  signal m8: MVL4_VECTOR(1 to 12);
16  signal m7: MVL4_VECTOR(1 to 12);
17  signal m6: MVL4_VECTOR(1 to 12);
18  signal m5: MVL4_VECTOR(1 to 12);
19  signal m4: MVL4_VECTOR(1 to 12);

20  signal m3: MVL4_VECTOR(1 to 12);
21  signal m2: MVL4_VECTOR(1 to 12);
22  signal m1: MVL4_VECTOR(1 to 12);
23  signal stk_ptr: BIT_VECTOR(2 downto 0);
24  signal GLB_RESET: BIT;
25  signal CLOCK: BIT;
26  signal STACK_OUT: MVL4_VECTOR(1 to 12);
27  signal UPC_OUT: MVL4_VECTOR(1 to 12);
28  signal DATA: MVL4_VECTOR(1 to 12);
29  signal REGCNT_ZERO: MVL4;
30  signal REGCNT_OUT: MVL4_VECTOR(1 to 12);
31  signal REGCNT_CNTL: BIT_VECTOR(0 to 1);
32  signal MUX_CNTL: BIT_VECTOR(0 to 2);
33
34 begin
35  P:process (GLB_RESET, CLOCK, G_RESET, DATAIN, CLKIN, REGCNT_OUT,
```

```
36        UPC_OUT, STACK_OUT, DATA, MUX_CNTL, LOAD, REGCNT_ZERO,
37        REGCNT_CNTL, CCEN, CC, INSTRUCTION)
38  begin
39
40  -- ------------------------------------------------------
41  -- STACK_PTR
42  -- ------------------------------------------------------
43
44    if (GLB_RESET'EVENT or CLOCK'EVENT) then
45     if (GLB_RESET = '1') then
46        stk_ptr <= "000";
47        FULL <= '0';
48     elsif (CLOCK'EVENT and CLOCK='1') then
49      if (STACK_CNTL = "01") then
50            stk_ptr <= "000";
51            FULL <= '0';
52      end if;
53      if (STACK_CNTL = "11") then
54            if (stk_ptr = "111") then
55              FULL <= '1';
56            else
57              FULL <= '0';
58              stk_ptr <= INC(stk_ptr);
59            end if;
60      end if;
61      if (STACK_CNTL = "10") then
62            if (stk_ptr = "000") then
63               FULL <= '0';
64            else
65               FULL <='0';
66               stk_ptr <= DEC(stk_ptr);
67            end if;
68      end if;
69      if (STACK_CNTL = "00") then
70            FULL <= '0';
71      end if;
72     end if;
73  end if;
```

Figure 5.12 The Inefficient AMD2910 Model (Lines 1 - 73)

model. The model has a total of 391 lines of code. The inefficiency in this model is as follows:

(1) REG_CNTL, STACK_CNTL, and stk_ptr are all objects of type bit_vector that could be integers. Input port INSTRUCTION is also a bit_vector type that could be an integer.

(2) The model contains many if statements that could be in the form of case statements. These if statements check the status of STACK_CNTL, stk_ptr, MUX_CNTL, and INSTRUCTION. The inefficient if statements that check the status of STACK_CNTL are shown in the figure at lines 49, 53, 61, and 69.

(3) UPC_CNTL, STACK_CNTL, MUX_CNTL, and the memory objects m1 - m8 are declared as signals but could be declared as variables.

(4) Since stk_ptr could be an integer, the functions (INC and DEC) used to increment and decrement the value of stk_ptr (shown in lines 58 and 66 of Figure 5.12) are inefficient. If type integer was used the addition and subtraction operators could be utilized.

The ERG was used to get a dynamic efficiency rating for the inefficient model. The output of the ERG for nodes 5 - 13 when the Vantage option was chosen is shown in Figure 5.13. Nodes 5 - 13 correspond to lines 49 - 51 and 53 - 58 of Figure 5.12. The CFG of the model contained a total of 211 nodes. When the ERG is invoked the user must give a yes or no answer to the questions of whether or not functions INC and DEC

CFG Node 5 (Line #49)

- True edge points to CFG Node 6 (Line #50).
- False edge points to CFG Node 8 (Line #53).

Inefficient use of if-endif series - could use case

- node coverage = 6

- node efficiency rating = 90

CFG Node 6 (Line #50)

- True edge points to CFG Node 7 (Line #51).
- False edge is undefined.

Inefficient use of object of type bit_vector - could use integer instead

- node coverage = 2

- node efficiency rating = 80

CFG Node 7 (Line #51)

- True edge points to CFG Node 8 (Line #53).
- False edge is undefined.

- node coverage = 2

- node efficiency rating = 100

CFG Node 8 (Line #53)

- True edge points to CFG Node 9 (Line #54).
- False edge points to CFG Node 14 (Line #61).

Inefficient use of if-endif series - could use case

- node coverage = 6

- node efficiency rating = 90

CFG Node 9 (Line #54)

- True edge points to CFG Node 10 (Line #55).
- False edge points to CFG Node 11 (Line #56).

- node coverage = 2

- node efficiency rating = 100

CFG Node 10 (Line #55)

- True edge points to CFG Node 14 (Line #61).
- False edge is undefined.

- node coverage = 0

- node efficiency rating = 100

CFG Node 11 (Line #56)

- True edge points to CFG Node 12 (Line #57).
- False edge is undefined.

- node coverage = 2

- node efficiency rating = 100

CFG Node 12 (Line #57)

- True edge points to CFG Node 13 (Line #58).
- False edge is undefined.

- node coverage = 2

- node efficiency rating = 100

CFG Node 13 (Line #58)

- True edge points to CFG Node 14 (Line #61).
- False edge is undefined.

Inefficient use of object of type bit_vector - could use integer instead

Inefficient use of a function that performs an arithmetic operation on a bit_vector that could be an integer - function then would not be needed

- node coverage = 2

- node efficiency rating = 70

Figure 5.13 The ERG Output for the Rating of AMD2910i (Nodes 5 to 13)

perform arithmetic operations on bit_vector stk_ptr. This is done because the ERG detects the statements, stk_ptr <= INC(stk_ptr) and stk_ptr <= DEC(stk_ptr), which are statements where a function is used to perform an operation on a bit_vector and the output is assigned to the same bit_vector. This is shown in Figure 5.13, node 13 (line 58) and node 19 (line 66). Since the answer to both questions is yes (i.e. the functions are arithmetic, therefore, if stk_ptr was an integer they would not be needed), those nodes are penalized. The dynamic rating for the model with Vantage was 95, which means that the efficient model should simulate approximately 5% faster. Both models were simulated and the average speedup was 3.23% (efficient model - .60 s, inefficient model - .62 s). The dynamic rating for Synopsys was 97, which means that there should be approximately 3% speedup. The actual speedup was 4.41% (efficient model - 4.12 s, inefficient model - 4.31 s).

5.6 Summary

All of the examples given in sections 5.1 to 5.5 of this chapter demonstrate that the ERG does a good job of identifying inefficiency in a model and predicting the severity of the inefficiency. Chapter 6 contains tabular results of more examples and conclusions drawn from them.

Chapter 6. Conclusion

This chapter contains detailed information on the results of a set of tests that were performed on the ERG. Conclusions are drawn from statistical analysis of these results. A brief discussion of future research possibilities is also included.

6.1 Tabular Results of Example Models

Many example models were developed to test the effectiveness of the ERG. Some of these examples, including those described in Chapter 5, and their results are shown in Tables 6.1 to 6.4. Tables 6.1 and 6.2 list some of the models that were simulated using

Table 6.1 Results of the First Set of Example Models Simulated with Vantage

Vantage	Simulation Time	Actual Rating	Dynamic Rating	Static Rating
MODELAe	650 msec	100	100	100
MODELAi	890 msec	73	72	71
MODELBe	2.40 sec	100	100	100
MODELBi	2.80 sec	86	87	92
MODELCe	2.40 sec	100	100	100
MODELCi	2.68 sec	90	94	91
MODELDe	2.40 sec	100	100	100
MODELDi	2.80 sec	86	86	84
PAR_TO_SERe	530 msec	100	100	100
PAR_TO_SERi	550 msec	96	97	93
FLEIO	500 msec	100	100	100
NOFLEIO	600 msec	83	85	85
ADDERe	700 msec	100	100	100
ADDERi	890 msec	79	85	85
RAM2e	610 msec	100	100	100
RAM2i	730 msec	84	90	90

Table 6.2 Results of the Second Set of Example Models Simulated with Vantage

Vantage		Simulation Time	Actual Rating	Dynamic Rating	Static Rating
Test_bench1	RAMe	17.33 sec	100	100	100
	RAMi	17.46 sec	99	97	87
Test_bench2	RAMe	1.44 sec	100	100	100
	RAMi	1.54 sec	94	92	87
Test_bench3	RAMe	530 msec	100	100	100
	RAMi	600 msec	88	86	87
Test_bench1	UARTe	2.28 sec	100	100	100
	UARTi	2.52 sec	90	88	86
Test_bench2	UARTe	3.06 sec	100	100	100
	UARTi	3.43 sec	89	88	86
Test_bench1	MARK2e	750 msec	100	100	100
	MARK2i	940 msec	80	89	88
Test_bench2	MARK2e	.94 sec	100	100	100
	MARK2i	1.11 sec	85	89	88
Test_bench1	AMD2910e	60 msec	100	100	100
	AMD2910i	62 msec	97	95	90
Test_bench2	AMD2910e	62 msec	100	100	100
	AMD2910i	64 msec	97	95	90

Vantage, and Tables 6.3 and 6.4 list the same models simulated using Synopsys. Each model name that ends with 'e' is the efficient version of the model and the model name ending in 'i' is the inefficient version of the model. In Table 6.1, models A, B, C, and D are basic models that test the inefficiency of constructs such as inefficient if statements, inefficient objects of type bit_vector, and inefficient objects declared as signals. The model PAR_TO_SER is a parallel-to-serial converter that also measures the inefficiency of signals that could be variables. The models FLEIO (employs file I/O) and NOFLEIO (reads vectors from a test bench) tests the efficiency of using file I/O as opposed to a test bench. Model ADDER, presented in Chapter 5, also tests the efficiency of file I/O. RAM2 is a model that measures the inefficiency of resolved signals. In Table 6.2, The RAM model, as described in Chapter 3, demonstrates how the dynamic rating accurately changes with the test bench, unlike the static rating. Specifically, the dynamic rating for the RAM using Test_bench1 and Test_bench3 is accurate in both cases even though the actual rating changes significantly. However, as shown with these two test benches, the accuracy of the static rating is dependent upon the test bench. The other models shown in Table 6.2 were described in detail in Chapter 5. These models also show that the dynamic rating is accurate for multiple test benches. As stated earlier, Tables 6.3 and 6.4 give the same examples using the Synopsys simulator. These tables demonstrate that the ERG works with the Synopsys simulator also. Appendix B contains the VHDL code for all inefficient models that are not listed in entirety in the main body of the dissertation. Table 6.5 shows the percent error for each of the Vantage and Synopsys ratings.

Table 6.3 Results of the First Set of Example Models Simulated with Synopsys

Synopsys	Simulation Time	Actual Rating	Dynamic Rating	Static Rating
MODELAe	90 msec	100	100	100
MODELAi	92 msec	98	95	95
MODELBe	4.61 sec	100	100	100
MODELBi	5.03 sec	92	92	95
MODELCe	4.61 sec	100	100	100
MODELCi	4.92 sec	94	96	98
MODELDe	4.61 sec	100	100	100
MODELDi	5.05 sec	91	92	94
PAR_TO_SERe	3.53 sec	100	100	100
PAR_TO_SERi	3.93 sec	90	99	99
FLEIO	1.45 sec	100	100	100
NOFLEIO	1.95 sec	74	80	80
ADDERe	6.52 sec	100	100	100
ADDERi	7.77 sec	84	80	80
RAM2e	1.49 sec	100	100	100
RAM2i	1.93 sec	77	85	85

Table 6.4 Results of the Second Set of Example Models Simulated with Synopsys

Synopsys		Simulation Time	Actual Rating	Dynamic Rating	Static Rating
Test_bench1	RAMe	7.61 sec	100	100	100
	RAMi	7.65 sec	99	99	98
Test_bench2	RAMe	4.02 sec	100	100	100
	RAMi	4.06 sec	99	99	98
Test_bench3	RAMe	2.46 sec	100	100	100
	RAMi	2.53 sec	97	98	98
Test_bench1	UARTe	31.51 sec	100	100	100
	UARTi	36.43 sec	86	84	84
Test_bench2	UARTe	31.86 sec	100	100	100
	UARTi	36.92 sec	86	84	84
Test_bench1	MARK2e	1.65 sec	100	100	100
	MARK2i	2.17 sec	76	84	84
Test_bench2	MARK2e	1.73 sec	100	100	100
	MARK2i	2.27 sec	76	84	84
Test_bench1	AMD2910e	4.14 sec	100	100	100
	AMD2910i	4.26 sec	97	97	97
Test_bench2	AMD2910e	4.12 sec	100	100	100
	AMD2910i	4.31 sec	96	97	97

Table 6.5 Percent Error of the Example Model Ratings

Model Name	% Error Vantage	Synopsys
MODELAi	1.37%	3.06%
MODELBi	1.16%	0%
MODELCi	4.44%	2.13%
MODELDi	0%	1.10%
PAR_TO_SERi	1.04%	10%
NOFLEIO	2.41%	8.11%
ADDERi	7.59%	4.76%
RAM2i	7.14%	10.39%
RAMi (test_bench1)	2.02%	0%
RAMi (test_bench2)	2.13%	0%
RAMi (test_bench3)	2.27%	1.03%
UARTi (test_bench1)	2.22%	2.33%
UARTi (test_bench2)	1.12%	2.33%
MARK2i (test_bench1)	11.25%	10.53%
MARK2i (test_bench2)	4.71%	10.53%
AMD2910i (test_bench1)	2.06%	0%
AMD2910I (test_bench2)	2.06%	1.04%

The ERG output shown in the example models of Chapter 5 show that the ERG is an effective tool for identifying inefficiency and suggesting improvements at individual nodes. As can be seen in Tables 6.1 through 6.4, the dynamic rating is within 10 points of the actual rating for all models, much closer in most cases. Based upon the results shown,

the static rating is also good for measuring inefficiency in a model. In most cases, the static rating is close to the dynamic rating, however, as shown in Chapter 3, Section 3.1, the static rating is not as accurate when a large segment of inefficient code in a model is not executed frequently during simulation, e.g. Table 6.2 - RAMi (test_bench1). The results also show that the ERG works with large models, e.g. AMD2910I, as well as smaller models. However, in some cases when there is more inefficiency in a large model, e.g MARK2I, the ratings may be slightly more inaccurate.

There are several statistical methods that can be used to analyze the data in Tables 6.1 through 6.4. As stated earlier, Table 6.5 shows the percent error of the dynamic rating of each inefficient model. The *percent error* is calculated as follows:

$$\%error = 100 \times \frac{|actual\_rating - dynamic\_rating|}{actual\_rating}$$

The **mean**, often called average, of a set of values is the sum of the values divided by their number [16]. The formula for calculating the mean is as follows:

$$\bar{x} = \frac{\sum x}{n}$$

In the case of the percent errors (Table 6.5), x is each percent error, and n is the total number which is 17. The mean percent error for the Vantage ratings is 3.23%, and the mean percent error for the Synopsys ratings is 3.96%.

The **standard deviation** is generally considered to be the most useful measure of the extent to which data is dispersed, i.e. varies from the mean [16]. The formula for calculating the standard deviation is as follows:

$$s = \sqrt{\frac{\sum\left(x-\bar{x}\right)^2}{n-1}}$$

The standard deviation for the Vantage ratings is 2.93 and for the Synopsys ratings it is 4.27.

The fact that the mean percent error for the Vantage and Synopsys ratings is 3.23% and 3.96%, respectively, shows that the ERG is a good tool for predicting the effect that the inefficiency in a model has on simulation performance. These results also show that the ERG works slightly better when predicting the performance of models simulated using Vantage. The fact that the standard deviation for the percent error of the Vantage and Synopsys ratings is only 2.93 and 4.27 respectively, shows that there is not much variance in the percent error of the ratings which means that the tool is relatively consistent with its predictions.

All of the qualities discussed above combine to make the ERG a very effective tool. This tool can certainly aid designers in developing highly efficient models. As models become increasingly complex, this becomes more important.

6.2 Future Research

This project has a wide range of future research possibilities. One possible effort could be an alternative method of providing dynamic ratings. The dynamic rating presented in this dissertation requires a test bench so that node coverage can be retrieved. A possible alternative would be to predict node coverage based upon the characteristics of the model which means that the test bench would not be needed. Sample code that demonstrates how node coverage might be predicted is shown below.

```
if (RESET'EVENT and RESET = '1') then
  CNT <= "0000";
elsif (CLK'EVENT and CLK = '1') then
  if (UP = '1') then
    CNT <= INC(CNT) after DEL;
  else
    CNT <= DEC(CNT) after DEL;
  end if;
end if;
```

In models that contain reset and clock signals, the reset is activated at a much lower percentage of time than the clock. Therefore in the example above it is apparent that the

clock path will be traversed much more frequently than the reset path. Hence, when the effect of inefficiency in the model is being predicted, the inefficiency in the clock path should be weighted much higher than the inefficiency in the reset path. After much study of the characteristics of models including node traversal trends this method of predicting inefficiency could be beneficial.

Another future research area that could be useful is to rate models for many areas of performance in addition to simulation efficiency. For example, models could be rated for their ability to be synthesized efficiently. This would save the modeler time and effort by avoiding the process of coding and synthesizing repeatedly in an effort to create the most efficient gate-level circuit. This could lead to an interesting study in identifying modeling constructs that are good for both simulation and synthesis. Readability is another issue that could be considered in rating models, since designers today create models that other designers must use as components in large complex systems. In summary, a tool that gives the user the option to rate models in several different areas including simulation performance, synthesizability, and readability would be very useful today. The ERG is a significant first step toward this goal.

Bibliography

[1] Wicks, John A., Jr., "Design and Verification of a Fault-Tolerant RISC Microprocessor Using VHDL," Master's Thesis, North Carolina A & T State University, June 1991.

[2] Singh, Balraj, "A Parametrized CAD Tool for VHDL Model Development with X Windows," Master's Thesis, Virginia Polytechnic Institute and State University, July 1990.

[3] Armstrong, James R., *Chip-Level Modeling With VHDL*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[4] Lipsett, R., Schaeffer, C. F., and Ussery, C., *VHDL: Hardware Description and Design*, Klumer Academic Publishers, Norwell, MA, 1989.

[5] Roig, R. L., "VHDL Coding Style for an Improved Simulation Performance," Master's Project, Virginia Polytechnic Institute and State University, January 1996.

[6] Pick, J., "VHDL Simulation Techniques and Recommendations" (Tutorial), VHDL International User's Forum, Spring 1996.

[7] Paulsen, B. and Levia, O., "Techniques for Writing High Performance and High Quality VHDL Models," (Tutorial), EURO VHDL '92.

[8] Armstrong, J. R. and Gray, F. G., *Structured Logic Design With VHDL*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[9] Wicks, John A., Jr. and Armstrong, James R., "VHDL Model Efficiency," *The Third Asia Pacific Conference on Hardware Description Languages (APCHDL '96)*, pp. 150-154.

[10] Mazor, S. and Langstraat, P., *A Guide to VHDL*, Kluwer Academic Publishers, Boston, MA, 1992.

[11] Vantage Language System User's Guide, Vol. 1, 1992.

[12] Gummadi, Ram, "Methodology for Structured VHDL Model Development", Master's Thesis, Virginia Polytechnic Institute and State University, 1995.

[13] Kapoor, Shekhar, "Process Level Test Generation for VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, February 1994.

[14] Li, W., "A Test Generation System For Behaviorally Modeled Digital Circuits," Ph.D. Dissertation, Virginia Polytechnic Institute and State University, June 1996.

[15] Freund, J. E., *Modern Elementary Statistics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[16] Wicks, J. A, and Armstrong, J. R., "Rating the Efficiency of VHDL Behavioral Models," *VIUF Fall Conference Proceedings*, 1996, pp. 345-351.

[17] Bondy, J. A., and Murty, U. S. R., *Graph Theory With Applications*, Elseiver Science Publishing Co., Inc., New York, NY, 1976.

[18] Software Procedural Interface (SPI) User's Manual, CAD Language Systems, Inc., 1989.

[19] Bernstein, D. B., and Charness, D., "Challenges in the Analysis of VHDL," *European Design Automation Conference - EURO VHDL '92*, pp. 740 - 745.

[20] Balboni, A., Mastretti, and Stefanoni, M., "Static Analysis for VHDL model Evaluation," *EURO VHDL '94*, pp. 586-591.

[21] Coelho, D., *The VHDL Handbook,* Kluwer Academic Publishers, Netherlands, 1989.

[22] Perry, D., *VHDL*, McGraw-Hill, New York, 1991.

[23] *IEEE Standard VHDL Language Reference Manual*, IEEE, New York, 1988.

[24] *VHDL System Simulator Core Programs Manual*, Synopsys, Inc. 1991.

[25] Bhasker, J., *A VHDL Primer*, Prentice - Hall, Englewood Cliffs, NJ, 1995.

[26] Harr, R. and Stanculescu, A., *Applications of VHDL to Circuit Design*, Kluwer Publishers, Boston, MA, 1991.

[27] Scott, K., "*Modeling Guidelines for Efficient Simulation*," VHDL User's Group, Fall 1989.

[28] Levine D. and Waxman, R., "Criteria for the Evaluation of VHDL Simulators," Using VHDL in System Design, Test, and Manufacturing, *VIUF Spring Conference Proceedings*, 1992, pp. 197 - 206.

[29] Saunders, L. and Trivedi, Y., "VHDL Simulators," *Asic & EDA*, July 1994, pp. 13 - 37.

[30] Sissler, J., "Evaluating VHDL Simulation Performance," Managing System Design and Development with VHDL, *VIUF Fall Conference Proceedings*, 1992, pp. 196 - 203.

[31] Ashenden, P. J., *The Designer's Guide to VHDL*, Morgan Kaufman Publishers, Inc., San Francisco, CA, 1996.

[32] Bhasker, J., *A Guide to VHDL Syntax*, Prentice-Hall, Englewood Cliffs, NJ, 1995.

[33] Navabi, Z., *VHDL Analysis and Modeling of Digital Systems*, McGraw-Hill, New York.

[34] Cohen, B., *VHDL Coding Styles and Methodologies*, Kluwer Academic Publishers, Boston, MA, 1995.

[35] Kennedy, J. B., and Neville, A. M., *Basic Statistical Methods for Engineers and Scientists*, Harper & Row Publishers, New York, 1986.

Appendix A: Program Code that Implements the Inefficient If-endif Series Algorithm

```
/*
 *      Checking for inefficient use of if-endif
 */
      if (ListLength(qRegions(Stm)) == 1) {
       Stmts3 = qStmts(qParent(Stm));
       Ptr2 = FirstItem(qRegions(Stm));
       if (strcmp(qText(qFunctor(qControl(Value(Ptr2)))), "=") == 0) {
        Stm4 = qControl(Value(Ptr2));
        comp = 0;
        strcpy(Lfsd1, "");
        strcpy(Rtsd1, "");
        strcpy(Lfsd2, "");
        strcpy(Rtsd2, "");
        lsnm = 0;
        strd = StrExp(&Stm4, &bkwd, &comp, itnm, &lsnm, Lfsd1, Rtsd1, Lfsd2, Rtsd2);
        bkwd = 0;
        if (strd == 1) {
         Stm2 = Stm;
         Stm3 = Stm;
         fwd = 0;
         penalty = IfEnd(&Stm2, &Stm3, &Stm4, &Stmts3, &comp, &fwd, &bkwd,
                   Lfsd1, Rtsd1, Lfsd2, Rtsd2);
         if (penalty == 1) {
          NodeInfo[Nodenum][17] = 1;
          if (vntg == 1) {
           penalty = 10;
          }
          else {
           penalty = 2;
          }
         }
         bkwd = 0;
         Nodratng = Nodratng - penalty;
        }
       }
      }
```

```
                .
                .
IfEnd(Stm2, Stm3, Stm4, Stmts3, comp, fwd, bkwd, Lfsd1, Rtsd1, Lfsd2, Rtsd2)

  Node       *Stm2;
  Node       *Stm3;
  Node       *Stm4;
  List      *Stmts3;
  Integer4   *comp;
  Integer4   *fwd;
  Integer4   *bkwd;
  char      *Lfsd1;
  char      *Rtsd1;
  char      *Lfsd2;
  char      *Rtsd2;


{


  Integer4    subpen;
  Integer4    strd;
  Integer4    itmnum;
  Integer4    itnm[100];
  Integer4    lsnm;
  Item       Ptr2;

  strd = 0;
  lsnm = 0;
  subpen = 0;
  *bkwd = 0;
  if (*fwd == 0) {
   Ptr2 = PrevItem(FindItem(*Stmts3, *Stm3));
  }
  else {
   Ptr2 = NextItem(FindItem(*Stmts3, *Stm3));
  }
  if (!NullItem(Ptr2)) {
   if (IsA(Kind(Value(Ptr2)), WaitStatement)) {
    if (*fwd == 0) {
     *fwd = 1;
     *Stm3 = *Stm2;
     subpen = IfEnd(Stm2, Stm3, Stm4, Stmts3, comp, fwd, bkwd,
              Lfsd1, Rtsd1, Lfsd2, Rtsd2);
    }
   }
   else if (!(IsA(Kind(Value(Ptr2)), IfStatement))) {
    *Stm3 = Value(Ptr2);
    subpen = IfEnd(Stm2, Stm3, Stm4, Stmts3, comp, fwd, bkwd,
             Lfsd1, Rtsd1, Lfsd2, Rtsd2);
   }
```

```
    else {
     *Stm3 = Value(Ptr2);
     if (ListLength(qRegions(*Stm3)) == 1) {
      if (strcmp(qText(qFunctor(qControl(Value(FirstItem(qRegions(*Stm3)))))), "=")
        == 0) {
       strcpy(Lfsd2, "");
       strcpy(Rtsd2, "");
       *comp = 1;
       *Stm4 = qControl(Value(FirstItem(qRegions(*Stm3))));
       strd = StrExp(Stm4, bkwd, comp, itnm, &lsnm, Lfsd1, Rtsd1, Lfsd2, Rtsd2);
       if (strd == 0) {
        subpen = IfEnd(Stm2, Stm3, Stm4, Stmts3, comp, fwd, bkwd,
                  Lfsd1, Rtsd1, Lfsd2, Rtsd2);
       }
       else {
        if ((strcmp(Lfsd1, Lfsd2) == 0) && (!(strcmp(Rtsd1, Rtsd2) == 0))) {
         subpen = 1;
        }
        else {
         subpen = IfEnd(Stm2, Stm3, Stm4, Stmts3, comp, fwd, bkwd,
                   Lfsd1, Rtsd1, Lfsd2, Rtsd2);
        }
       }
      }
      else {
       subpen = IfEnd(Stm2, Stm3, Stm4, Stmts3, comp, fwd, bkwd,
                 Lfsd1, Rtsd1, Lfsd2, Rtsd2);
      }
     }
     else {
      subpen = IfEnd(Stm2, Stm3, Stm4, Stmts3, comp, fwd, bkwd,
                Lfsd1, Rtsd1, Lfsd2, Rtsd2);
     }
    }
   }
   else {
    if (*fwd == 0) {
     *fwd = 1;
     *Stm3 = *Stm2;
     subpen = IfEnd(Stm2, Stm3, Stm4, Stmts3, comp, fwd, bkwd,
               Lfsd1, Rtsd1, Lfsd2, Rtsd2);
    }
   }
   return subpen;
}
```

StrExp(Stm4, bkwd, comp, itnm, lsnm, Lfsd1, Rtsd1, Lfsd2, Rtsd2)

```
  Node       *Stm4;
  Integer4   *bkwd;
  Integer4   *comp;
  Integer4   itnm[100];
  Integer4   *lsnm;
  char       *Lfsd1;
  char       *Rtsd1;
  char       *Lfsd2;
  char       *Rtsd2;


{

  Integer4   subpen;
  Integer4   itmnum;
  Integer4   rtsd;
  Integer4   strd;

  strd = 0;
  subpen = 0;
  rtsd = 0;
  if ((*bkwd == 1) && (strcmp(qText(qFunctor(*Stm4)), "=") == 0)) {
   itmnum = 2;
   rtsd = 1;
   if (*comp == 0) {
    *Rtsd1 = StrItm(itmnum, Rtsd1, rtsd, Stm4);
    if (!(strcmp(Rtsd1, "\0")) == 0) {
     strd = 1;
    }
   }
   else {
    *Rtsd2 = StrItm(itmnum, Rtsd2, rtsd, Stm4);
    if (!(strcmp(Rtsd2, "\0")) == 0) {
     strd = 1;
    }
   }
  }
  else {
   if (!((IsA(Kind(Value(NthItem(1, qArgs(*Stm4)))), Operation)) ||
     (IsA(Kind(Value(NthItem(2, qArgs(*Stm4)))), Operation)))) {
    itmnum = 1;
    if (*comp == 0) {
     *Lfsd1 = StrItm(itmnum, Lfsd1, rtsd, Stm4);
     if (!(strcmp(Lfsd1, "\0")) == 0) {
      strd = 1;
     }
    }
    else {
     *Lfsd2 = StrItm(itmnum, Lfsd2, rtsd, Stm4);
     if (!(strcmp(Lfsd2, "\0")) == 0) {
```

```
    strd = 1;
   }
  }
  if (strcmp(qText(qFunctor(*Stm4)), "=") == 0) {
   itmnum = 2;
   rtsd = 1;
   if (*comp == 0) {
    *Rtsd1 = StrItm(itmnum, Rtsd1, rtsd, Stm4);
    if (!(strcmp(Rtsd1, "\0")) == 0) {
     strd = 1;
    }
   }
   else {
    *Rtsd2 = StrItm(itmnum, Rtsd2, rtsd, Stm4);
    if (!(strcmp(Rtsd2, "\0")) == 0) {
     strd = 1;
    }
   }
  }
  else {
   itmnum = 2;
   if (*comp == 0) {
    *Lfsd1 = StrItm(itmnum, Lfsd1, rtsd, Stm4);
    if (!(strcmp(Lfsd1, "\0")) == 0) {
     strd = 1;
    }
   }
   else {
    *Lfsd2 = StrItm(itmnum, Lfsd2, rtsd, Stm4);
    if (!(strcmp(Lfsd2, "\0")) == 0) {
     strd = 1;
    }
   }
   if (strd == 1) {
    if ((itnm[*lsnm] == 1) || (*lsnm == 1)) {
     *Stm4 = qParent(*Stm4);
     *bkwd = 1;
     itnm[*lsnm] = 2;
     *lsnm = *lsnm - 1;
     subpen = StrExp(Stm4, bkwd, comp, itnm, lsnm,
           Lfsd1, Rtsd1, Lfsd2, Rtsd2);
    }
    else {
     *Stm4 = qParent(qParent(*Stm4));
     *bkwd = 1;
     *lsnm = *lsnm - 1;
     itnm[*lsnm] = 2;
     subpen = StrExp(Stm4, bkwd, comp, itnm, lsnm,
               Lfsd1, Rtsd1, Lfsd2, Rtsd2);
    }
   }
```

```
    }
   }
   else {
    if (*bkwd == 0) {
     if (IsA(Kind(Value(NthItem(1, qArgs(*Stm4)))), Operation)) {
      if (!(IsA(Kind(qFunctor(Value(NthItem(1, qArgs(*Stm4))))), Identifier))) {
       itmnum = 1;
       if (*comp == 0) {
        *Lfsd1 = StrItm(itmnum, Lfsd1, rtsd, Stm4);
        strd = 1;
       }
       else {
        *Lfsd2 = StrItm(itmnum, Lfsd2, rtsd, Stm4);
        strd = 1;
       }
       if (strd == 1) {
        *Stm4 = Value(NthItem(1, qArgs(*Stm4)));
        *lsnm = *lsnm + 1;
        itnm[*lsnm] = 1;
        subpen = StrExp(Stm4, bkwd, comp, itnm, lsnm,
                  Lfsd1, Rtsd1, Lfsd2, Rtsd2);
       }
      }
     }
}
    else {
     if (IsA(Kind(Value(NthItem(2, qArgs(*Stm4)))), Operation)) {
      if (!(IsA(Kind(qFunctor(Value(NthItem(2, qArgs(*Stm4))))), Identifier))) {
       itmnum = 2;
       if (*comp == 0) {
        *Lfsd1 = StrItm(itmnum, Lfsd1, rtsd, Stm4);
        strd = 1;
       }
       else {
        *Lfsd2 = StrItm(itmnum, Lfsd2, rtsd, Stm4);
        strd = 1;
       }
       if (strd == 1) {
        *Stm4 = Value(NthItem(2, qArgs(*Stm4)));
        *lsnm = *lsnm + 1;
        *bkwd = 0;
        subpen = StrExp(Stm4, bkwd, comp, itnm, lsnm,
                  Lfsd1, Rtsd1, Lfsd2, Rtsd2);
       }
      }
     }
     else {
      itmnum = 2;
      if (*comp == 0) {
       *Lfsd1 = StrItm(itmnum, Lfsd1, rtsd, Stm4);
       if (!(strcmp(Lfsd1, "\0")) == 0) {
```

```
            strd = 1;
           }
          }
          else {
           *Lfsd2 = StrItm(itmnum, Lfsd2, rtsd, Stm4);
           if (!(strcmp(Lfsd2, "\0")) == 0) {
            strd = 1;
           }
          }
          if (strd == 1) {
           if ((itnm[*lsnm] == 1) || (*lsnm == 1)) {
            *Stm4 = qParent(*Stm4);
            itnm[*lsnm] = 2;
            *lsnm = *lsnm - 1;
            subpen = StrExp(Stm4, bkwd, comp, itnm, lsnm,
                      Lfsd1, Rtsd1, Lfsd2, Rtsd2);
           }
           else {
            *Stm4 = qParent(qParent(*Stm4));
            *lsnm = *lsnm - 1;
            itnm[*lsnm] = 2;
            subpen = StrExp(Stm4, bkwd, comp, itnm, lsnm,
                      Lfsd1, Rtsd1, Lfsd2, Rtsd2);
           }
          }
         }
        }
       }
   }
   return strd;
}



StrItm(itmnum, Tmpstrg, rtsd, Stm4)

   Node       *Stm4;
   char       *Tmpstrg;
   Integer4    rtsd;
   Integer4    itmnum;

{

   Integer4    subpen;
   Item       Ptr2;
   Node        Stm5;

   if ((IsA(Kind(Value(NthItem(itmnum, qArgs(*Stm4)))), CharacterLiteral)) ||
     (IsA(Kind(Value(NthItem(itmnum, qArgs(*Stm4)))), EnumeratedLiteral))) {
    strcat(Tmpstrg, qText(Value(NthItem(itmnum, qArgs(*Stm4)))));
```

```
    }
    else if (IsA(Kind(Value(NthItem(itmnum, qArgs(*Stm4)))), StringValue)) {
     strcat(Tmpstrg, qStrVal(Value(NthItem(itmnum, qArgs(*Stm4)))));
    }
    else if (IsA(Kind(Value(NthItem(itmnum, qArgs(*Stm4)))), IntegerValue4)) {
     sprintf(Tmpstrg, "%s%d", Tmpstrg, qIntVal4(Value(NthItem(itmnum, qArgs(*Stm4)))));
    }
    else if (rtsd == 0 && (IsA(Kind(Value(NthItem(itmnum, qArgs(*Stm4)))), Operation))) {
     strcat(Tmpstrg, qText(qFunctor(Value(NthItem(itmnum, qArgs(*Stm4))))));
    }
    else if (rtsd == 0 && (IsA(Kind(Value(NthItem(itmnum, qArgs(*Stm4)))), Identifier))) {
     strcat(Tmpstrg, qText(Value(NthItem(itmnum, qArgs(*Stm4)))));
    }
    else if (rtsd == 0 && (IsA(Kind(Value(NthItem(itmnum, qArgs(*Stm4)))), SliceOp))) {
     Stm5 = Value(NthItem(itmnum, qArgs(*Stm4)));
     if ((IsA(Kind(Value(NthItem(1, qData(Stm5)))), Identifier)) &&
       ((IsA(Kind(Value(NthItem(2, qData(Stm5)))), AscendingRange)) ||
       (IsA(Kind(Value(NthItem(2, qData(Stm5)))), DescendingRange)))) {
       strcat(Tmpstrg, qText(Value(NthItem(1, qData(Stm5)))));
       Stm5 = Value(NthItem(2, qData(Stm5)));
       sprintf(Tmpstrg, "%s%d", Tmpstrg, qIntVal4(Value(NthItem(1, qConstraint(Stm5)))));
       sprintf(Tmpstrg, "%s%d", Tmpstrg, qIntVal4(Value(NthItem(2, qConstraint(Stm5)))));
     }
    }
    else if (rtsd == 0 && (IsA(Kind(Value(NthItem(itmnum, qArgs(*Stm4)))), IndexOp))) {
     Stm5 = Value(NthItem(itmnum, qArgs(*Stm4)));
     if ((IsA(Kind(Value(NthItem(1, qData(Stm5)))), Identifier)) &&
       ((IsA(Kind(Value(NthItem(2, qData(Stm5)))), Identifier)) ||
       (IsA(Kind(Value(NthItem(2, qData(Stm5)))), IntegerValue4)))) {
       strcat(Tmpstrg, qText(Value(NthItem(1, qData(Stm5)))));
       if (IsA(Kind(Value(NthItem(2, qData(Stm5)))), Identifier)) {
        strcat(Tmpstrg, qText(Value(NthItem(2, qData(Stm5)))));
       }
       else {
        sprintf(Tmpstrg, "%s%d", Tmpstrg, qIntVal4(Value(NthItem(2, qData(Stm5)))));
       }
     }
    }
    return *Tmpstrg;
}
```

Appendix B: Inefficient VHDL Example Models

```
1   use work.all;
2   entity MODELAi is
3     port(EN: in BIT);
4   end MODELAi;
5
6   architecture BEHAVIORAL of MODELAi is
7
8     signal sig1, sig2, sig3, sig4, sig5,
9          sig6, sig7, sig8, sig9, sig10: NATURAL;
10
11    signal val1, val2, val3, val4, val5,
12         val6, val7, val8, val9, val10: BIT_VECTOR(3 downto 0);
13
14  begin
15
16    P:process(EN)
17      begin
18        if (EN = '1') then
19          sig1 <= sig1 + 1;
20          sig2 <= sig2 + 1;
21          sig3 <= sig3 + 1;
22          sig4 <= sig4 + 1;
23          sig5 <= sig5 + 1;
24          sig6 <= sig6 + 1;
25          sig7 <= sig7 + 1;
26          sig8 <= sig8 + 1;
27          sig9 <= sig9 + 1;
28          sig10 <= sig10 + 1;
29  --
30          val1 <= "0000";
31          val2 <= "0001";
32          val3 <= "0011";
33          val4 <= "0100";
34          val5 <= "0101";
35          val6 <= "0110";
36          val7 <= "0111";
37          val8 <= "1000";
38          val9 <= "1001";
```

```
39              val10 <= "1010";
40            end if;
41          end process;
42      end BEHAVIORAL;
```

```
1   use work.all;
2   entity MODELBi is
3     port(I: in BIT_VECTOR(2 downto 0); EN: in BIT; Q: out INTEGER);
4   end MODELBi;
5
6   architecture BEHAVIORAL of MODELBi is
7
8     begin
9
10      P:process
11        variable O: INTEGER;
12       begin
13        if EN = '1' then
14          if (I = "000") then
15            O := 7;
16          end if;
17          if (I = "001") then
18            O := 6;
19          end if;
20          if (I = "010") then
21            O := 5;
22          end if;
23          if (I = "011") then
24            O := 4;
25          end if;
26          if (I = "100") then
27            O := 3;
28          end if;
29          if (I = "101") then
30            O := 2;
31          end if;
32          if (I = "110") then
33            O := 1;
34          end if;
35          if (I = "111") then
36            O := 0;
37          end if;
38        end if;
39        wait for 5 ns;
40        Q <= O;
41       wait on I, EN;
42     end process;
43  end BEHAVIORAL;
```

```
1   use work.all;
2   entity MODELCi is
3     port(I: in BIT_VECTOR(2 downto 0); EN: in BIT; Q: out INTEGER);
4   end MODELCi;
5
6   architecture BEHAVIORAL of MODELCi is
7
8     signal O: INTEGER;
9
10    begin
11
12      P:process
13        begin
14          if EN = '1' then
15            case I is
16              when "000" => O <= 7;
17              when "001" => O <= 6;
18              when "010" => O <= 5;
19              when "011" => O <= 4;
20              when "100" => O <= 3;
21              when "101" => O <= 2;
22              when "110" => O <= 1;
23              when "111" => O <= 0;
24            end case;
25          end if;
26          wait for 5 ns;
27          Q <= O;
28        wait on I, EN;
29      end process;
30  end BEHAVIORAL;
```

```
1   use work.USER_TYPES.all;
2   entity NOFLEIO is
3     generic (PER: TIME);
4     port (EN: in BIT; BVIN: in BIT_VECTOR(7 downto 0);
5         BVOUT: out BIT_VECTOR(7 downto 0));
6   end NOFLEIO;
7
8   architecture BEHAVIORAL of NOFLEIO is
9     begin
10      process
11        begin
12          wait on EN until EN = '1';
13          while (EN = '1') loop
14            BVOUT <= BVIN;
15            wait for PER;
16          end loop;
17      end process;
18  end BEHAVIORAL;
```

```
1   use work.all, work.USER_TYPES.all;
2   entity RAM2i is
3     generic (RDEL, DISDEL: TIME);
4     port (DATA: inout BUS1(7 downto 0) := "ZZZZZZZZ";
5           ADDR: in BIT_VECTOR(3 downto 0);
6           RD, WR, NCS, INIT: in BIT);
7   end RAM2i;
8
9   architecture BEHAVIORAL of RAM2i is
10
11    type MEMORY is array (0 to 15) of MVL4_VECTOR(7 downto 0);
12
13  begin
14
15    P:process(NCS, RD, WR, INIT)
16
17      variable MEM: MEMORY;
18
19      begin
20
21        if (INIT'EVENT and INIT = '1') then
22          MEM(0) := "00000000";
23          MEM(1) := "00000001";
24          MEM(2) := "00000010";
25          MEM(3) := "00000011";
26          MEM(4) := "00000100";
27          MEM(5) := "00000101";
28          MEM(6) := "00000110";
29          MEM(7) := "00000111";
30          MEM(8) := "00001000";
31          MEM(9) := "00001001";
32          MEM(10) := "00001010";
33          MEM(11) := "00001011";
34          MEM(12) := "00001100";
35          MEM(13) := "00001101";
36          MEM(14) := "00001110";
37          MEM(15) := "00001111";
38        end if;
39        if (NCS = '0') then
40          if (RD'EVENT) then
41            if (RD = '1') then
42              DATA <= DRIVE(MEM(INTVAL(ADDR))) after RDEL;
43            else
44              DATA <= "ZZZZZZZZ" after DISDEL;
45            end if;
46          elsif (WR'EVENT and WR = '1') then
47            MEM(INTVAL(ADDR)) := SENSE(DATA, '1');
48          end if;
49        else
50          DATA <= "ZZZZZZZZ" after DISDEL;
```

```
51          end if;
52        end process;
53      end BEHAVIORAL;
```

```
1   use work.USER_TYPES.all, work.all;
2   entity UARTi is
3     generic (CLK_PER, ODEL, INDEL, INTDEL: TIME);
4     port (I: in MVL4; LOAD, READ: in BIT;
5         DATA: inout BUS1(7 downto 0) := "ZZZZZZZZ";
6         NINTO, NINTI: out BIT; O: out MVL4);
7   end UARTi;
8
9   architecture BEHAVIORAL of UARTi is
10
11    signal ICLK, OCLK, ISTRT: BIT;
12    signal NINTI1, NINTI2: BIT := '1';
13    signal I_FLAG: BOOLEAN := TRUE;
14    signal IREG, OREG: MVL4_VECTOR(7 downto 0);
15    signal ICNTR, OCNTR: INTEGER;
16
17
18  begin
19
20    P: process(LOAD, OCLK, I, ISTRT, ICLK, READ)
21
22       begin
23
24  --
25  -- OUTPUT
26  --
27      if (LOAD'EVENT or OCLK'EVENT) then
28        if (LOAD'EVENT and LOAD = '1') then
29          OREG <= SENSE(DATA, '1');
30          OCNTR <= 7;
31          NINTO <= '1' after INTDEL;
32          O <= '0' after ODEL;
33          OCLK <= not OCLK after CLK_PER;
34        end if;
35
36        if (OCLK'EVENT) then
37          if (OCNTR /= -1) then
38            O <= OREG(OCNTR) after ODEL;
39            OCNTR <= OCNTR - 1;
40            OCLK <= not OCLK after CLK_PER;
41          else
42            O <= '1' after ODEL;
43            NINTO <= '0' after INTDEL;
44          end if;
45        end if;
46      end if;
47  --
48  -- INPUT
49  --
50      if (I'EVENT or ISTRT'EVENT or ICLK'EVENT) then
```

```
51          if (I_FLAG) then
52            if (I'EVENT and I = '0') then
53              ISTRT <= '1' after CLK_PER/2;
54            end if;
55          end if;
56
57          if (ISTRT'EVENT and I = '0') then
58            I_FLAG <= FALSE;
59            ISTRT <= '0';
60            ICNTR <= 7;
61            ICLK <= not ICLK after CLK_PER;
62          end if;
63
64          if (ICLK'EVENT) then
65            if (ICNTR /= -1) then
66              IREG(ICNTR) <= I;
67              ICNTR <= ICNTR - 1;
68              ICLK <= not ICLK after CLK_PER;
69            else
70              NINTI1 <= '0' after INTDEL;
71              I_FLAG <= TRUE;
72            end if;
73          end if;
74        end if;
75    --
76    -- READ
77    --
78        if (READ'EVENT) then
79          if (READ = '1') then
80            DATA <= DRIVE(IREG) after INDEL;
81            NINTI2 <= '1' after INTDEL;
82          else
83            DATA <= "ZZZZZZZZ" after INDEL;
84          end if;
85        end if;
86    --
87    -- INTERRUPTS
88    --
89        if (NINTI1'EVENT or NINTI2'EVENT) then
90          if (NINTI1'EVENT) then
91            NINTI <= NINTI1;
92          elsif (NINTI2'EVENT) then
93            NINTI <= NINTI2;
94          end if;
95        end if;
96      end process;
97    end BEHAVIORAL;
```

```
1   use work.fncMARK2.all, work.USER_TYPES.all, work.all;
2   entity MARK2i is
3     generic (RDEL, WDEL, ODEL, MADEL, INTDEL, PER: TIME);
4     port (RUN, INT: in BIT; FETCH, EXECUTE: in BOOLEAN;
5         DATA: inout BUS1(7 downto 0) := "ZZZZZZZZ";
6         MA: out BIT_VECTOR(4 downto 0);
7         RD, WRITE, IO, RDS, RDP, WTS, WTP, INTA: out BIT);
8   end MARK2i;
9
10  architecture BEHAVIORAL of MARK2i is
11
12    signal STOP, STOPR, STOPE: BIT;
13    signal INTE, INTER, INTEE, INTEI: BIT;
14    signal TEMP_RD, RDF, RDE: BIT;
15    signal WRITEF, WRITEE: BIT;
16    signal IOWAIT, IOWAITF, IOWAITE, IOWAITI: BIT;
17    signal TEMP_MA, MAF, MAE: BIT_VECTOR(4 downto 0);
18    signal CLK, INTERUPT: BOOLEAN;
19    signal PCF, PCI, PCE, PC: BIT_VECTOR(4 downto 0);
20    signal IR: BIT_VECTOR(7 downto 0);
21    signal ACC: MVL4_VECTOR(7 downto 0);
22
23  begin
24
25    P: process
26       begin
27
28  --
29  -- System Clock
30  --
31          if (CLK'EVENT or RUN'EVENT) then
32            if (RUN = '1') then
33              CLK <= transport not CLK after PER;
34            end if;
35          end if;
36  --
37  -- RUN
38  --
39          if (RUN'EVENT) then
40            if (RUN = '1') then
41              STOPR <= '0';
42              INTER <= '0';
43            else
44              STOPR <= '1';
45            end if;
46          end if;
47  --
48  -- SET INTERRUPT
49  --
50          if (STOP'EVENT or IOWAIT'EVENT or CLK'EVENT or
```

```
51              FETCH'EVENT or EXECUTE'EVENT or INTERUPT'EVENT) then
52             if (RUN'EVENT and (STOP = '0') and (IOWAIT = '0') and CLK'EVENT) then
53              if (EXECUTE and (INT = '1' and INTE = '1')) then
54               INTERUPT <= TRUE;
55              else
56               INTERUPT <= FALSE;
57              end if;
58             end if;
59            end if;
60    --
61    -- FETCH_CYCLE
62    --
63            if (CLK'EVENT and CLK = TRUE and FETCH = TRUE) then
64             MAF <= PC after MADEL;
65             RDF <= '1' after ODEL;
66             WRITEF <= '0' after ODEL;
67             IOWAITF <= '1';
68             wait for RDEL;
69             IR <= MVL4VEC_TO_BITVEC(SENSE(DATA, '1'));
70             RDF <= '0' after ODEL;
71             IOWAITF <= '0';
72             PCF <= INC_ADDR(PC);
73            end if;
74    --
75    -- EXECUTE CYCLE
76    --
77            if (CLK'EVENT and CLK = TRUE and EXECUTE = TRUE) then
78
79             if (IR(7 downto 5) = "000") then
80              PCE <= IR(4 downto 0);                    --jmp
81             end if;
82
83             if (IR(7 downto 5) = "001") then
84              ACC <= INC_WORD(INVW(ACC));                --tca
85             end if;
86
87             if (IR(7 downto 5) = "010") then
88              MAE <= IR(4 downto 0);                    --lda
89              RDE <= '1' after ODEL;
90              WRITEE <= '0' after ODEL;
91              IOWAITE <= '1';
92              wait for RDEL;
93              RDE <= '0' after ODEL;
94              IOWAITE <= '0';
95              ACC <= SENSE(DATA, '1');
96             end if;
97
98             if (IR(7 downto 5) = "011") then
99              DATA <= transport DRIVE(ACC) after ODEL;          --sta
100             DATA <= transport "ZZZZZZZZ" after 3 * ODEL;
101             DATA <= transport "ZZZZZZZZ" after (3 * ODEL) + WDEL;
```

```
102             MAE <= IR(4 downto 0) after MADEL;
103             RDE <= '0' after ODEL;
104             WRITEE <= '1' after ODEL;
105             IOWAITE <= '1';
106             wait for WDEL;
107             WRITEE <= '0' after ODEL;
108             IOWAITE <= '0';
109           end if;
110
111           if (IR(7 downto 5) = "100") then
112             MAE <= IR(4 downto 0);                    --add
113             RDE <= '1' after ODEL;
114             WRITEE <= '0' after ODEL;
115             IOWAITE <= '1';
116             wait for RDEL;
117             RDE <= '0' after ODEL;
118             IOWAITE <= '0';
119             ACC <= ADDD8(ACC, SENSE(DATA, '1'));
120           end if;
121
122           if (IR(7 downto 5) = "101") then
123             if IR(4) = '1' then                       --eni
124               INTEE <= '1';
125             else
126               INTEE <= '0';
127             end if;
128           end if;
129
130           if (IR(7 downto 5) = "110") then
131             if ACC(7) = '1' then                      --jpn
132               PCE <= IR(4 downto 0);
133             end if;
134           end if;
135
136           if (IR(7 downto 5) = "111") then
137             STOPE <= '1';                             --stop
138           end if;
139         end if;
140   --
141   -- INTERRUPT CYCLE
142   --
143         if (INTERUPT'EVENT and INTERUPT = TRUE and INTE = '1') then
144           INTEI <= '0';
145           INTA <= '1' after ODEL;
146           IOWAITI <= '1';
147           wait for INTDEL;
148           PCI <= VDAD(SENSE(DATA, '1'));
149           INTA <= '0' after ODEL;
150           IOWAIT <= '0';
151         end if;
152   --
```

```
153   -- I/O Control Signal Decoding
154   --
155         if (TEMP_MA(4) = '1' and TEMP_MA(3) = '1') then
156           if (TEMP_MA'EVENT) then
157             RDP <= not TEMP_MA(2) and not TEMP_MA(1) and not TEMP_MA(0) after ODEL;
158             WTP <= not TEMP_MA(2) and not TEMP_MA(1) and TEMP_MA(0) after ODEL;
159             RDS <= not TEMP_MA(2) and TEMP_MA(1) and not TEMP_MA(0) after ODEL;
160             WTS <= not TEMP_MA(2) and TEMP_MA(1) and TEMP_MA(0) after ODEL;
161             RDP <= '0' after ODEL + RDEL;
162             WTP <= '0' after ODEL + WDEL;
163             RDS <= '0' after ODEL + RDEL;
164             WTS <= '0' after ODEL + WDEL;
165           end if;
166         end if;
167
168         if (TEMP_MA(4)'EVENT or TEMP_MA(3)'EVENT) then
169           IO <= TEMP_MA(4) and TEMP_MA(3) after ODEL;
170           IO <= '0' after ODEL + RDEL;
171         end if;
172   --
173   -- Process Output Multiplexing
174   --
175         if (PCE'EVENT or PCI'EVENT or PCF'EVENT) then
176           if (PCE'EVENT) then
177             PC <= PCE;
178           elsif (PCI'EVENT) then
179             PC <= PCI;
180           else
181             PC <= PCF;
182           end if;
183         end if;
184
185         if (STOPR'EVENT or STOPE'EVENT) then
186           if (STOPR'EVENT) then
187             STOP <= STOPR;
188           else
189             STOP <= STOPE;
190           end if;
191         end if;
192
193         if (IOWAITF'EVENT or IOWAITE'EVENT or IOWAITI'EVENT) then
194           if (IOWAITF'EVENT) then
195             IOWAIT <= IOWAITF;
196           elsif (IOWAITE'EVENT) then
197             IOWAIT <= IOWAITE;
198           else
199             IOWAIT <= IOWAITI;
200           end if;
201         end if;
202
203         if (INTER'EVENT or INTEE'EVENT or INTEI'EVENT) then
```

```
204             if (INTER'EVENT) then
205               INTE <= INTER;
206             elsif (INTEE'EVENT) then
207               INTE <= INTEE;
208             else
209               INTE <= INTEI;
210             end if;
211           end if;
212
213           if (RDF'EVENT or RDE'EVENT) then
214             if (RDF'EVENT) then
215               TEMP_RD <= RDF;
216             else
217               TEMP_RD <= RDE;
218             end if;
219           end if;
220
221           if (TEMP_RD'EVENT) then
222             RD <= TEMP_RD;
223           end if;
224
225           if (WRITEF'EVENT or WRITEE'EVENT) then
226             if (WRITEF'EVENT) then
227               WRITE <= WRITEF;
228             else
229               WRITE <= WRITEE;
230             end if;
231           end if;
232
233           if (MAF'EVENT or MAE'EVENT) then
234             if (MAF'EVENT) then
235               TEMP_MA <= MAF;
236             else
237               TEMP_MA <= MAE;
238             end if;
239           end if;
240
241           if (TEMP_MA'EVENT) then
242             MA <= TEMP_MA;
243           end if;
244
245       wait on CLK,RUN,STOP,IOWAIT,FETCH,EXECUTE,INTERUPT,STOPR,STOPE,IOWAITF,
246               IOWAITE,IOWAITI,INTER,INTEE,INTEI,RDF,RDE,WRITEF,WRITEE,MAF,MAE,
247               TEMP_RD,TEMP_MA,PCF,PCI,PCE;
248     end process;
249   end BEHAVIORAL;
```

```
1   use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
2   -- ********************************************************
3   entity AMD2910i is
4     port (G_RESET, CLKIN: in BIT; CARRYIN, LOAD, CCEN, CC: in MVL4;
5         INSTRUCTION: in BIT_VECTOR(0 to 3); DATAIN: in MVL4_VECTOR(1 to 12);
6         OUTADDR: inout MVL4_VECTOR(1 to 12); FULL, INTERENABLE, MAPPINENABLE,
7         PIPENABLE: out MVL4);
8   end AMD2910i;
9   -- ********************************************************
10
11  architecture BEHAVIORAL of AMD2910i is
12
13    signal STACK_CNTL: BIT_VECTOR(0 to 1);
14    signal UPC_CNTL: BIT;
15    signal m8: MVL4_VECTOR(1 to 12);
16    signal m7: MVL4_VECTOR(1 to 12);
17    signal m6: MVL4_VECTOR(1 to 12);
18    signal m5: MVL4_VECTOR(1 to 12);
19    signal m4: MVL4_VECTOR(1 to 12);
20    signal m3: MVL4_VECTOR(1 to 12);
21    signal m2: MVL4_VECTOR(1 to 12);
22    signal m1: MVL4_VECTOR(1 to 12);
23    signal stk_ptr: BIT_VECTOR(2 downto 0);
24    signal GLB_RESET: BIT;
25    signal CLOCK: BIT;
26    signal STACK_OUT: MVL4_VECTOR(1 to 12);
27    signal UPC_OUT: MVL4_VECTOR(1 to 12);
28    signal DATA: MVL4_VECTOR(1 to 12);
29    signal REGCNT_ZERO: MVL4;
30    signal REGCNT_OUT: MVL4_VECTOR(1 to 12);
31    signal REGCNT_CNTL: BIT_VECTOR(0 to 1);
32    signal MUX_CNTL: BIT_VECTOR(0 to 2);
33
34  begin
35    P:process (GLB_RESET, CLOCK, G_RESET, DATAIN, CLKIN, REGCNT_OUT,
36          UPC_OUT, STACK_OUT, DATA, MUX_CNTL, LOAD, REGCNT_ZERO,
37          REGCNT_CNTL, CCEN, CC, INSTRUCTION)
38    begin
39
40    -- -------------------------------------------------------
41    -- STACK_PTR
42    -- -------------------------------------------------------
43
44      if (GLB_RESET'EVENT or CLOCK'EVENT) then
45        if (GLB_RESET = '1') then
46          stk_ptr <= "000";
47          FULL <= '0';
48        elsif (CLOCK'EVENT and CLOCK='1') then
49          if (STACK_CNTL = "01") then
50                stk_ptr <= "000";
```

```
51                      FULL <= '0';
52              end if;
53              if (STACK_CNTL = "11") then
54                      if (stk_ptr = "111") then
55                        FULL <= '1';
56                      else
57                        FULL <= '0';
58                        stk_ptr <= INC(stk_ptr);
59                      end if;
60              end if;
61              if (STACK_CNTL = "10") then
62                      if (stk_ptr = "000") then
63                        FULL <= '0';
64                      else
65                        FULL <='0';
66                        stk_ptr <= DEC(stk_ptr);
67                      end if;
68              end if;
69              if (STACK_CNTL = "00") then
70                      FULL <= '0';
71              end if;
72            end if;
73          end if;
74
75      -- ------------------------------------------------------
76      -- STACK_OUT
77      -- ------------------------------------------------------
78
79          if (stk_ptr'EVENT) then
80            if (stk_ptr = "000") then
81                STACK_OUT <= m1;
82            end if;
83            if (stk_ptr = "001") then
84                STACK_OUT <= m2;
85            end if;
86            if (stk_ptr = "010") then
87                STACK_OUT <= m3;
88            end if;
89            if (stk_ptr = "011") then
90                STACK_OUT <= m4;
91            end if;
92            if (stk_ptr = "100") then
93                STACK_OUT <= m5;
94            end if;
95            if (stk_ptr = "101") then
96                STACK_OUT <= m6;
97            end if;
98            if (stk_ptr = "110") then
99                STACK_OUT <= m7;
100           end if;
101           if (stk_ptr = "111") then
```

```
102             STACK_OUT <= m8;
103           end if;
104         end if;
105
106       -- ------------------------------------------------------
107       -- STACK_MEM
108       -- ------------------------------------------------------
109
110         if (GLB_RESET'EVENT or CLOCK'EVENT) then
111           if (GLB_RESET = '1' ) then
112             m1 <= "000000000000";
113             m2 <= "000000000000";
114             m3 <= "000000000000";
115             m4 <= "000000000000";
116             m5 <= "000000000000";
117             m6 <= "000000000000";
118             m7 <= "000000000000";
119             m8 <= "000000000000";
120           elsif (CLOCK'EVENT and CLOCK = '1') then
121             if (STACK_CNTL = "11")  then
122               if (stk_ptr = "000") then
123                 m1 <= UPC_OUT;
124               end if;
125               if (stk_ptr = "001") then
126                 m2 <= UPC_OUT;
127               end if;
128               if (stk_ptr = "010") then
129                 m3 <= UPC_OUT;
130               end if;
131               if (stk_ptr = "011") then
132                 m4 <= UPC_OUT;
133               end if;
134               if (stk_ptr = "100") then
135                 m5 <= UPC_OUT;
136               end if;
137               if (stk_ptr = "101") then
138                 m6 <= UPC_OUT;
139               end if;
140               if (stk_ptr = "110") then
141                 m7 <= UPC_OUT;
142               end if;
143               if (stk_ptr = "111") then
144                 m8 <= UPC_OUT;
145               end if;
146             end if;
147           end if;
148         end if;
149
150       -- ------------------------------------------------------
151       -- GRETBUF
152       -- ------------------------------------------------------
```

```
153
154       if (G_RESET'EVENT) then
155         GLB_RESET <= G_RESET;
156       end if;
157
158     -- ------------------------------------------------------
159     -- DATABUF
160     -- ------------------------------------------------------
161
162       if (DATAIN'EVENT) then
163         DATA <= DATAIN;
164       end if;
165
166     -- ------------------------------------------------------
167     -- CLKBUF
168     -- ------------------------------------------------------
169
170       if (CLKIN'EVENT) then
171         CLOCK <= CLKIN;
172       end if;
173
174     -- ------------------------------------------------------
175     -- UPC
176     -- ------------------------------------------------------
177
178       if (CLOCK'EVENT or GLB_RESET'EVENT) then
179         if (GLB_RESET = '1') then
180               UPC_OUT <= "000000000000";
181         elsif (CLOCK'EVENT and CLOCK = '1' ) then
182               if (UPC_CNTL = '1') then
183                     if (CARRYIN = '1') then
184                           UPC_OUT <= BVtoMVL4V(INC(MVL4VtoBV(OUTADDR)));
185                     else
186                           UPC_OUT <= OUTADDR;
187                     end if;
188               else
189                     UPC_OUT <= "000000000000";
190               end if;
191         end if;
192       end if;
193
194     -- ------------------------------------------------------
195     -- REGCNT2
196     -- ------------------------------------------------------
197
198       if (REGCNT_OUT'EVENT) then
199         if (REGCNT_OUT = "000000000000") then
200               REGCNT_ZERO <= '1';
201         else
202               REGCNT_ZERO <= '0';
203         end if;
```

```
204      end if;
205
206     -- ------------------------------------------------------
207     -- REGCNT1
208     -- ------------------------------------------------------
209
210      if (REGCNT_CNTL'EVENT or CLOCK'EVENT or GLB_RESET'EVENT) then
211       if (GLB_RESET = '1') then
212            REGCNT_OUT <= "000000000000";
213       elsif (CLOCK'EVENT and CLOCK = '1') then
214            if (REGCNT_CNTL = "01") then
215                 REGCNT_OUT <= DATA;
216            elsif (REGCNT_CNTL = "10") then
217                 REGCNT_OUT <= BVtoMVL4V(DEC(MVL4VtoBV(REGCNT_OUT)));
218            elsif (REGCNT_CNTL = "00") then
219                 REGCNT_OUT <= REGCNT_OUT;
220            end if;
221       end if;
222      end if;
223
224     -- ------------------------------------------------------
225     -- MUX
226     -- ------------------------------------------------------
227
228      if (UPC_OUT'EVENT or STACK_OUT'EVENT or REGCNT_OUT'EVENT or
229        DATA'EVENT or MUX_CNTL'EVENT) then
230       if (MUX_CNTL = "000") then
231            OUTADDR <= DATA;
232       end if;
233       if (MUX_CNTL = "001") then
234            OUTADDR <= REGCNT_OUT;
235       end if;
236       if (MUX_CNTL = "011") then
237            OUTADDR <= STACK_OUT;
238       end if;
239       if (MUX_CNTL = "010") then
240            OUTADDR <= UPC_OUT;
241       end if;
242       if (MUX_CNTL = "100") then
243            OUTADDR <= "000000000000";
244       end if;
245       if (MUX_CNTL = "101") then
246            OUTADDR <= "000000000000";
247       end if;
248       if (MUX_CNTL = "110") then
249            OUTADDR <= "000000000000";
250       end if;
251       if (MUX_CNTL = "111") then
252            OUTADDR <= "000000000000";
253       end if;
254      end if;
```

```
255
256     -- ------------------------------------------------------
257     -- CONTROL
258     -- ------------------------------------------------------
259
260      if (REGCNT_ZERO'EVENT or LOAD'EVENT or CCEN'EVENT or
261        CC'EVENT or INSTRUCTION'EVENT) then
262       PIPENABLE <= '1';
263       MAPPINENABLE <= '1';
264       INTERENABLE <= '1';
265       UPC_CNTL <= '1';
266       STACK_CNTL <= "00";
267       MUX_CNTL <= "010";
268
269       if (LOAD = '1') then
270            REGCNT_CNTL <= "00";
271       else
272            REGCNT_CNTL <= "01";
273       end if;
274
275       if (INSTRUCTION = "0000") then
276                       UPC_CNTL <= '0';
277                       STACK_CNTL <= "01";
278                       MUX_CNTL <= "100";
279                       PIPENABLE <= '0';
280       end if;
281       if (INSTRUCTION = "0001") then
282                       PIPENABLE <= '0';
283                       if (CCEN = '1' or CC = '0') then
284                       STACK_CNTL <= "11";
285                               MUX_CNTL <= "000";
286                       end if;
287       end if;
288       if (INSTRUCTION = "0010") then
289                       MAPPINENABLE <= '0';
290                       MUX_CNTL <= "000";
291       end if;
292       if (INSTRUCTION = "0011") then
293                       PIPENABLE <= '0';
294                       if (CCEN = '1' or CC = '0') then
295                               MUX_CNTL <= "000";
296                       end if;
297       end if;
298       if (INSTRUCTION = "0100") then
299                       STACK_CNTL <= "11";
300                       PIPENABLE <= '0';
301                       if (CCEN = '1' or CC = '0') then
302                               REGCNT_CNTL <= "01";
303                       end if;
304       end if;
305       if (INSTRUCTION = "0101") then
```

```
306                     PIPENABLE <='0';
307                     STACK_CNTL <= "11";
308                     if (CCEN ='0' and CC = '1') then
309                             MUX_CNTL <= "001";
310                     else
311                             MUX_CNTL <= "000";
312                     end if;
313     end if;
314     if (INSTRUCTION = "0110") then
315                     INTERENABLE <= '0';
316                     if (CCEN = '1' or CC = '0') then
317                             MUX_CNTL <= "000";
318                     end if;
319     end if;
320     if (INSTRUCTION = "0111") then
321                     PIPENABLE <= '0';
322                     if (CCEN ='0' and CC ='1') then
323                             MUX_CNTL <= "001";
324                     else
325                             MUX_CNTL <= "000";
326                     end if;
327     end if;
328     if (INSTRUCTION = "1000") then
329                     PIPENABLE <= '0';
330                     if (REGCNT_ZERO = '0') then
331                             MUX_CNTL <= "011";
332                             REGCNT_CNTL <= "10";
333                     else
334                             STACK_CNTL <= "10";
335                     end if;
336     end if;
337     if (INSTRUCTION = "1001") then
338                     PIPENABLE <= '0';
339                     if (REGCNT_ZERO = '0') then
340                             MUX_CNTL <= "000";
341                             REGCNT_CNTL <= "10";
342                     end if;
343     end if;
344     if (INSTRUCTION = "1010") then
345                     PIPENABLE <='0';
346                     if (CCEN = '1' or CC ='0') then
347                             MUX_CNTL <= "011";
348                             STACK_CNTL <= "10";
349                     end if;
350     end if;
351     if (INSTRUCTION = "1011") then
352                     PIPENABLE <='0';
353                     if (CCEN = '1' or CC = '0') then
354                             MUX_CNTL <= "000";
355                             STACK_CNTL <= "10";
356                     end if;
```

```
357         end if;
358         if (INSTRUCTION = "1100") then
359                         PIPENABLE <= '0';
360                         REGCNT_CNTL <= "01";
361         end if;
362         if (INSTRUCTION = "1101") then
363                         PIPENABLE <= '0';
364                         if (CCEN = '0' and CC = '1') then
365                                 MUX_CNTL <= "011";
366                         else
367                                 STACK_CNTL <= "10";
368                         end if;
369         end if;
370         if (INSTRUCTION = "1111") then
371                         PIPENABLE <= '0';
372                         if (REGCNT_ZERO = '0') then
373                                 REGCNT_CNTL <= "10";
374                                 if (CCEN = '0' and CC = '1') then
375                                         MUX_CNTL <= "011";
376                                 else
377                                         STACK_CNTL <= "10";
378                                 end if;
379                         else
380                                 STACK_CNTL <= "10";
381                                 if (CCEN = '0' and CC = '1') then
382                                         MUX_CNTL <= "000";
383                                 end if;
384                         end if;
385         end if;
386         if (INSTRUCTION = "1110") then
387               PIPENABLE <= '0';
388         end if;
389       end if;
390     end process;
391   end BEHAVIORAL;
```

Vita

John A. Wicks, Jr. was born on October 15, 1966 in Jackson, Mississippi. He graduated from Murrah High School in Jackson in June of 1984. John then entered Alcorn State University in Lorman, Mississippi where he graduated with a bachelor's degree in Computer Science and Applied Mathematics in May of 1988. John entered graduate school at North Carolina A & T State University in Greensboro, NC where he graduated with a Master of Science degree in Electrical Engineering in June of 1991. Finally, John entered Virginia Polytechnic Institute and State University in Blacksburg, VA where he graduated with a Ph.D. degree in Electrical Engineering in December of 1996.