Hardware Modeling and Top-Down Design Using VHDL

by

Dennis P. Morton

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in
Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 1991

Signature of Author _____
Department of Electrical Engineering and Computer Science
May 10, 1991

Certified by _____
Jonathan A. Allen
Thesis Supervisor

Certified by _____
Bryan P. Butler
Charles Stark Draper Laboratory

Accepted by _____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Hardware Modeling and Top-Down Design Using VHDL

by

Dennis P. Morton

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 1991 in partial fulfillment of the requirements for the degree of

Master of Science

Abstract

As digital designs grow more and more complex, some method of controlling this complexity must be used in order to reduce the number of errors and the time spent on a design. VHDL (Very High Speed Integrated Circuit Hardware Description Language) promises to ease the design and verification of complex digital circuits by encouraging the use of top-down design.

This thesis demonstrates how VHDL, combined with a top-down design methodology, enables the designer to specify and verify a digital design faster and with fewer errors. The scoreboard, a section of hardware in the Charles Stark Draper Laboratory's Fault Tolerant Parallel Processor, is used as an example to demonstrate the utility of VHDL. The scoreboard is responsible for message processing within the FTPP and thus has a critical effect on performance. It also represents the most significant risk of any component in the FTPP. The use of VHDL has the potential for ensuring an optimal scoreboard design with minimal errors and an improved design time.

Thesis Supervisor: Jonathan A. Allen
Title: Professor of Electrical Engineering

Acknowledgments

I owe a great debt to Bryan Butler and R. Edwin Harper, who decided to let me give VHDL a try. Without their support, my VHDL research would have never gotten off the ground. I would also like to thank the rest of the Fault-Tolerant Systems Division for providing such a swell working environment. Finally, to Nicole, who, despite not being very interested in VHDL, still listened to my ravings about it.

This work was done at the Charles Stark Draper Laboratory under NASA contract NAS-1-18565.

Table of Contents

List of Figures

1. Introduction

1.1. Problem Statement

In order to meet future requirements of extremely-reliable computers with high throughput, the Charles Stark Draper Laboratory (CSDL) initiated the Fault-tolerant Parallel Processor (FTPP) project. The FTPP achieved these requirements by combining Byzantine resilience[1] with parallelism via multiple, concurrently executing processors. Cluster 1 (C1), the laboratory prototype, was completed in 1988. While an excellent proof of concept, the design possessed design and implementation flaws which were difficult and tedious to find and rectify. The next FTPP, Cluster 3 (C3)[2], was conceived as a third-generation FTPP suitable for use in field applications.

The scoreboard is a section of hardware responsible for message processing in the FTPP. In C1, it was implemented with many PALs and RAMs and was very difficult to debug. Furthermore, flaws were found in the fundamental algorithm. These reasons, as well as a significant increase in the scoreboard functionality, necessitated a complete redesign of the scoreboard for C3.

This new design presented many challenges. First, the algorithm required extensive reworking to achieve the enhanced functionality. Second, the scoreboard's complexity mandated the use of good design techniques. Finally, the design had extensive testing problems which had to be solved. An effective methodology was needed to address these challenges and come up with the best possible design. VHDL (Very High Speed Integrated Circuit Hardware Description Language) encourages the use of such a methodology and has other advantages that made it an effective tool in designing the scoreboard: 1) it allows design tradeoffs to be investigated quickly and easily; 2) it simplifies testing and validation of the scoreboard by allowing one test bench to be used throughout the design process; and 3) it provides implementation-independence for much of the design cycle.

---

1 Byzantine resilience is a degree of fault tolerance allowing toleration of arbitrary faults. See section 2.1 for a more detailed explanation.

2 C2 was a minimum Byzantine resilient system designed to demonstrate high-speed fiber optics for inter-FCR communication.

1.2. Objective

The objective of this thesis is to design and document a fully functional behavioral scoreboard model in VHDL, both to demonstrate the advantages of VHDL and to accelerate the design process of the scoreboard. This thesis attempts to show that top-down design using VHDL yields better designs with fewer iterations. It also presents some guidelines and techniques to enhance the modeling process itself.

Since the entire design process, from concept to working hardware, cannot be completed in the amount of time allotted a thesis, the VHDL model is also used to document the work completed to date. The VHDL model, along with this thesis, will serve to completely document the work which has been done on the scoreboard.

1.3. Audience

The intended audience for this thesis, besides my advisors, is any engineer interested in modeling using VHDL, especially at the behavioral level. I have attempted to structure my writing such that little knowledge of VHDL or fault-tolerance is required. However, chapter 7 will have more meaning if the reader has at least a working knowledge of VHDL.

This thesis is also aimed at anyone who is skeptical of the utility of VHDL, especially those who are wary of any language which is a Department of Defense standard. Hopefully, the following exercise will persuade these people of the merits of using VHDL to design digital hardware.

1.4. Approach

This thesis is a VHDL design example. As such, it is structured to specify the scoreboard's design, explain the motivations for using VHDL, enumerate the advantages of VHDL, and interpret/analyze the VHDL model of the scoreboard with some suggestions for hardware implementations.

Chapter 2 familiarizes the reader with the concepts of fault-tolerance and Byzantine Resilience as applied to the FTPP in general and the scoreboard in particular. Chapter 3 introduces VHDL and motivates the subsequent chapter on the advantages of VHDL modeling. Chapter 5 covers modeling issues such as state-machines, timing, and synchronous designs. Chapter 6 provides a functional description of the scoreboard. Chapter

7 covers the behavioral model of the scoreboard. The final two chapters discuss various implementation methods and topics for further research.

2. Background Information

Since the dawn of the computer age, computer architects have been interested in constructing fault-tolerant computers to decrease down-time and increase reliability. One application for fault-tolerance is in transaction-processing systems, which should be fault-tolerant to make errors unlikely and to allow them to remain "up" while being repaired. One example is the Tandem line of NonStop® computers. Stratus also makes a line of fault-tolerant computers, notable for their ability to phone the factory when a part fails. These computers all have at least one thing in common — they take the approach of estimating and covering expected failure modes by replicating critical components and voting outputs. For example, alternate boards of a self-checking pair are powered by separate supplies so that if one power supply fails, only half of each pair is affected.

The Achille's heel of most fault-tolerant computers, including those mentioned above, is malicious faults. If a component fails in such a way that it produces conflicting outputs, these computers will probably not be able to reach an agreement. Figure 2-1 shows such a situation using three independent computers connected with bi-directional communication links. Computer **A** has failed maliciously and is transmitting conflicting information to the other two computers. Computers **B** and **C** must act off what Computer **A** has said[3]. However, due to Computer **A**'s malicious failure, no consensus is possible, since no clear majority exists [Lamp 82].

---

3 Computer **A** might have a sensor attached to it whose data Computers **B** and **C** need also.

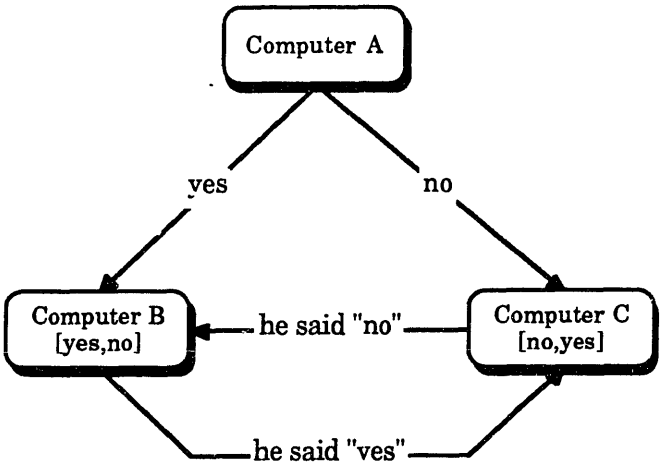


Figure 2-1, Malicious Failure with Three Computers

The ability to tolerate malicious failures is desirable for applications which require extremely high reliability. Some examples are flight system control, where a failure could cause the plane to crash, and jet engine control, where a failure could cause loss of the engine. The Byzantine Resilience [Lamp82] algorithm discussed below guarantees consensus even in the presence of malicious failures.

2.1. Fundamentals of Byzantine Resilience

A computer which is able to tolerate any arbitrary, single random fault is said to be 1-Byzantine resilient. Arbitrary means that there are no constraints on what fault modes are covered; any one fault, no matter how unlikely, may occur with 100% coverage. Byzantine resilient algorithms exist to cover any number (f) of arbitrary faults.

A fault-tolerant computer is designed with a number of interconnected fault-containment regions (FCR), each region being incapable of propagating an internal fault to other FCRs [Butler89]. This is achieved by physical and electrical isolation of the FCRs. Byzantine Resilience places four formal requirements on a fault-tolerant computer to achieve 100% coverage of a single arbitrary fault. These requirements are:

1. There must be at least 3f+1 FCRs [Lamp82].

2. Each FCR must be connected to at least 2f+1 other FCRs through unique communication links [Dolev82].

3. The protocol must consist of at least f+1 rounds of communication among FCRs. This is known as the source congruency requirement [Harper87].

4. The FCRs must be synchronized to within a known and bounded skew [Harper87].

A minimal 1-Byzantine Resilient configuration is shown in Figure 2-2. It contains four FCRs, each connected to the three other FCRs through bi-directional communication links.

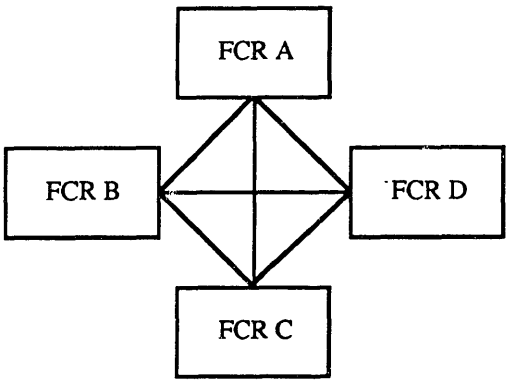


Figure 2-2, Minimal 1-Byzantine Resilient System

2.2. Exchanges

There are two fundamental methods of exchanging messages in a 1-Byzantine Resilient system to arrive at consistent data. These two exchange methods are known as class 1 and class 2 exchanges [Harper87]. Each type of exchange will be explained using Figure 2-2 as a reference.

A class 1 exchange is performed when all FCRs have a message which must be consistent across the system. This exchange has one phase wherein each FCR sends its message to the other three FCRs. Each FCR then votes the original message plus the three copies of the message it received to arrive at a consistent message. A class 1 exchange guarantees validity of the exchanged data.

A class 2, or source congruency, exchange is performed when one FCR has a message which must be distributed to all other FCRs. All non-faulty FCRs must agree on this message. This exchange has two phases. In the first phase, the source FCR sends its message to the three other FCRs. In the second phase, each FCR sends a copy of the message it received to the three other FCRs. Each FCR then votes the copies (the original is not included in the

voting) of the message to arrive at a consistent result. A class 2 exchange guarantees validity if the source is non-faulty and agreement if the source is faulty.

2.3. The FTPP

The Fault Tolerant Parallel Processor was designed to fill the need for an ultra-high reliability, high-performance computer. The first prototype FTPP, known as C1, is a 1-Byzantine Resilient system consisting of four FCRs interconnected by high-speed communication links [Harper87]. Each FCR contains one Network Element (NE) and four Processing Elements (PE). The physical configuration is shown in Figure 2-3. The PEs are single-board computers, while the NEs are custom hardware which perform the Byzantine resilience exchanges.

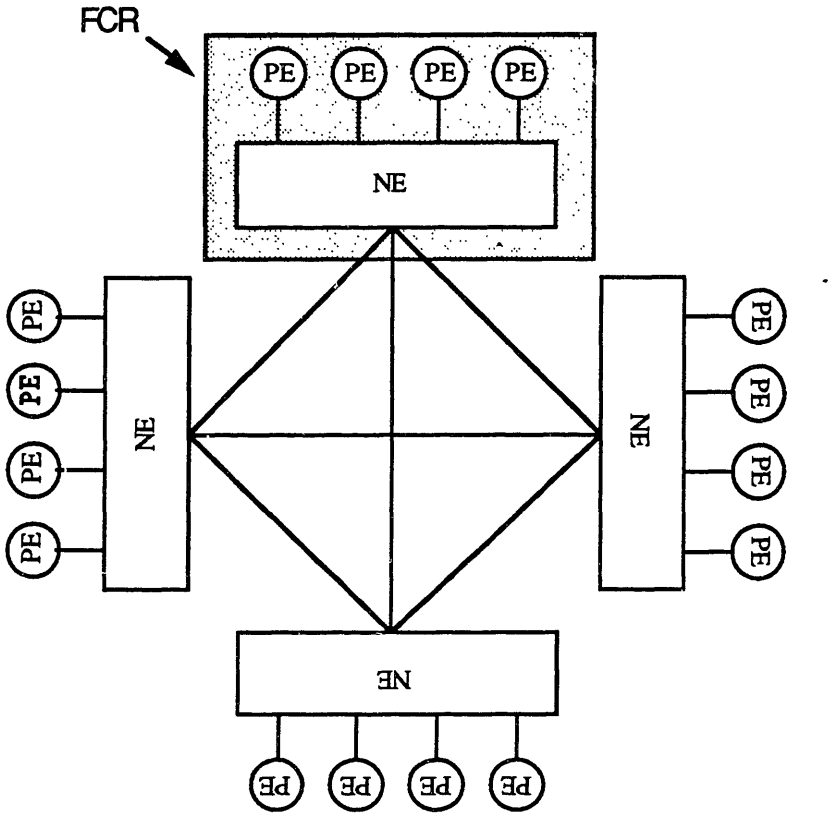


Figure 2-3, The FTPP

C1 provides the capability to logically group PEs together into fault masking groups (FMG) of two, three, or four processors to enhance the reliability of critical tasks. The members of a FMG run the same code and periodically exchange messages to ensure that they are operating on the same inputs and producing the same outputs. Each FMG is treated

as a single entity, or virtual group, for purposes of sending and receiving messages. When a FMG is sent a message, all PEs in the group receive a copy.

All messages in C1 are exchanged between virtual groups. A virtual group can be either a FMG or a single PE. Thus every PE in the system has two "addresses," its physical ID, which indicates its {NE,PE} location, and its virtual ID, which other virtual groups use to send it messages. The NE maintains a data structure called the configuration table (CT) which translates virtual IDs to physical IDs. Passing messages in this manner allows healthy PEs to transparently assume the tasks of faulty PEs.

When a virtual group wishes to send a message, it writes an exchange request into a FIFO (First In-First Out memory) in the NE. Periodically, each NE assembles this information into a Local Exchange Request Pattern (LERP) [Harper87]. Four source congruency exchanges are performed on the LERPs to ensure that all the NEs have consistent copies of the four LERPs. The aggregate of the four LERPs is called the System Exchange Request Pattern (SERP) [Harper87]. The SERP is delivered to the scoreboard, a section of hardware internal to the NE. The scoreboard processes the exchange requests in the SERP to decide which messages to exchange.

2.4. C3 Scoreboard Concepts

The C1 scoreboard possessed design and implementation flaws which were difficult to find and rectify. Similar flaws are intolerable in the fieldable C3, so it was decided to completely redesign the scoreboard from scratch. The rest of this chapter introduces the functions of the C3 scoreboard, which in many ways closely resembles that of C1. It describes the SERP and configuration table and provides an overview of timeouts. Chapter 6 contains the complete functional description of the scoreboard.

2.4.1. SERP

Periodically, each NE polls its own PEs to determine four pieces of information :

1. Does the PE have a message to send (is its Output Buffer Not Empty, OBNE)?
2. To whom will it be sent (destination virtual ID) ?
3. What type of message is it (i.e. class 1, class 2, other)?
4. Can the PE receive a message (is its Input Buffer Not Full, IBNF) ?

Once this information has been gathered for each PE, the combined information inside each NE is assembled into the Local Exchange Request Pattern (LERP). The NEs then execute a source congruency exchange to arrive at a consistent aggregate of the four LERPs, the System Exchange Request Pattern (SERP). The SERP is then passed on to the scoreboard for processing.

SERP entries are indexed by processor ID (PID) and network element ID (NEID). In other words, the first PE in the system (NE 0, PE 0) has the first entry in the SERP and so on traversing the NEs and PEs. The scoreboard, however, must read all the SERP entries corresponding to a VID in order to vote them. Therefore, some method of mapping PIDs to VIDs must exist. The data structure which implements this mapping is called the configuration table (CT).

2.4.2. Configuration Table

All PEs are combined into virtual groups composed of one, three, or four members known as simplexes, triplexes and quadruplexes (quads). The members of a virtual group are addressed in aggregate through the VID number. Unlike C1, C3 does not support virtual groups with two members since such a virtual group provides no fault masking capability. Each member must reside on a different NE to satisfy the isolation between FCRs necessary for Byzantine resilience.

Because PEs deal with virtual addresses and NEs deal with physical addresses, there must be a way of mapping PEs to VIDs. The data structure which performs this function is the configuration cable (CT). Each entry in the CT corresponds to one VID[4] and contains the redundancy level of the VID, a bit field denoting the NE locations of the members of that VID, the PIDs of all the VID's members, and a value to use when performing timeouts on the VID.

2.4.3. Timeouts

Timeouts are required because PEs are functionally synchronized. They arise from the need to be able to detect the absence of a message [Lamp82]. They allow the scoreboard to ignore faulty PEs who disagree with the other members of the virtual group for a given period of time. For example, if one member of a VID had its power turned off, its OBNE and IBNF

---

4 From this point on, the term VID denotes a virtual group.

bits would never get asserted. Without timeouts, that virtual group could not send or receive messages because agreement between members would never be achieved.

A timeout is begun on a virtual group whenever a majority, but not a unanimity, of its members have their OBNE or IBNF bits set. If the timeout expires before unanimity is observed, the OBNE or IBNF bit for the virtual group will be set. The OBNE and IBNF timeouts are handled independently. The exact protocol and rules for starting and checking timeouts will be discussed in section 6.4.1.4.

2.5. Design Goals

The C3 scoreboard presented many design challenges. First, the algorithm used required detailed specification to ensure that all tenets of Byzantine resilience were followed. Second, because of the complexity of this algorithm, the hardware to implement it had to be carefully designed and optimized. This process by its very nature would involve many design iterations. Finally, the testing strategy of the design required complex test-generation algorithms itself.

Because of these challenges, the top-down methodology was thought to be the wisest method to use for designing the scoreboard. It was also felt that VHDL would allow the design to be tested and optimized with the least amount of effort.

3. Hardware Description Languages

This chapter provides an overview of the major features of hardware description languages in general and VHDL in particular. An in-depth discussion is beyond the scope of this thesis. However, the following discussion presents the features which are important for understanding the scoreboard VHDL model. I recommend reading this chapter even if the reader has worked with VHDL before since it presents my view of the language (which is very likely different from other views).

The chapter begins with a brief history of hardware description languages (HDL) and the impetus behind their development. It then covers desirable features of an HDL. The rest of the chapter is devoted to the development and features of VHDL.

3.1. History

Hardware description languages were originally developed in the early 1970's to simplify the design of computer hardware. With the advent of large scale integration, schematics alone became less able to convey sufficient information about a design. Furthermore, there was an increasing need to describe and document designs at a higher level of abstraction. The new logic simulators of the time also required a means to describe a design [Lip 77].

HDL research caught fire with the promise of simplifying the design of increasingly complex computer circuits. It wasn't long before many HDLs existed, each exhibiting different strengths but none of which could be used for all levels of the design process [Lip 77]. In 1973, the ACM and the IEEE formed a combined, ad-hoc committee with the goal of attempting to standardize HDLs [Lip 77]. The committee was interested in creating standard features which all HDLs should incorporate. However, they did not wish to stifle HDL research so they proposed only a baseline feature set [Lip 77].

3.1.1. Desirable Features

An HDL must possess certain features in order to be effective and useful. First, and most importantly, it must support concurrency since pieces of hardware by nature operate in parallel. Most general purpose programming languages do not support parallelism, thus

making them poor choices for hardware modeling[5] [Lip 77]. A good HDL should also support differing levels of abstraction, even within the same model, but not force the modeler into using any particular style. Some common levels of abstraction, from most abstract to least, are algorithmic, dataflow, and structural. Finally, an HDL should provide a built-in model of time to make it useful for timing checks.

3.1.2. VHDL Impetus and Development

In 1980 the U.S. Government launched the Very High Speed Integrated Circuits (VHSIC) program with the goal of significantly increasing the performance and density of integrated circuits. Very soon afterwards, however, the Government realized that to help different contractors work together efficiently and ensure the reusability and maintainability of designs, a standard method to communicate design data was needed. Furthermore, the densities of VHSIC chips and the complexity of the resulting systems exposed the need for a method to smooth the design process and manage the huge amounts of design data [Wax89]. The VHDL program was born from these needs.

The VHDL program was officially begun in 1981 with an initial meeting of people from government, academia, and industry [Wax89]. The original language contracts were awarded to Intermetrics, IBM, and TI, with Intermetrics being the prime contractor. The IEEE, also recognizing the need for a standard hardware description language, began a standardization effort in 1986, the same year in which Intermetrics released the first VHDL toolset. IEEE Standard 1076, passed in December, 1987, standardized the VHDL language [Wax89].

At the time of standardization in December, 1987, only one crude VHDL toolset existed. By 1991, at least half a dozen commercial VHDL toolsets and a number of free university VHDL toolsets were available. Nearly every major CAE vendor has announced or is shipping a VHDL product, which indicates how well VHDL has become accepted both inside and outside government circles. VHDL promises to become nearly as pervasive as schematic entry systems, with an even greater impact on design productivity and automation.

---

5 Though many papers exist on such a subject.

3.2. VHDL Overview

VHDL is a concurrently executed language with a intrinsic sense of time. This means that parts of a given model will appear to execute concurrently. All VHDL statements are scheduled to execute at a given point in time and are executed sequentially[6] within a single delta – an infinitesimally small, but non-zero, unit of time. The simulation time is then advanced to the next set of scheduled statements which are executed in the next delta. Each such execute-update cycle is known as a simulation cycle [RL 89]. The time aspect of VHDL is complex and full of pitfalls. The author suggests reading Lipsett, Schaefer, and Ussery's excellent book "VHDL : Hardware Description and Design" for a more detailed description of timing in VHDL (especially Chapter 5).

The general model on which VHDL is based is composed of three distinct, interrelated models : behavior, time, and structure [RL 89]. The model of behavior allows the designer to specify the function of an object without regards to its internal structure. The structural model allows the designer to describe an object's function using simpler, interconnected objects. The model of time, perhaps the most important aspect of VHDL, allows the designer to embed timing information in the model. The following sections explain how VHDL implements these models.

3.2.1. The Design Entity

The principal hardware abstraction in VHDL is the design entity [RL 86]. A design entity is composed of two fundamental parts : the interface and the design body. An important feature of the language is that more than one design body can exist for a given interface. Different bodies can focus on different levels of hardware abstraction, for example. A VHDL model can be composed of any number of design entities connected together.

The design entity's interface is described by an entity declaration. This declaration contains an arbitrary number of ports and generics (though neither is syntactically necessary) which are used to pass information into and out of the design entity. The interface represents the only portion of the design visible outside the entity. The design body

---

6 Sequentially because VHDL platforms (at least as of this writing) all run on uniprocessor systems. Simulation speed would be greatly enhanced if VHDL ran on a parallel processor, though.

is composed of an architecture declaration and an optional configuration specification. The function of the entity is implemented inside of the architecture. The following sections describe the design entity in more detail.

3.2.1.1. Interface Declaration

The design entity interface is contained in an entity declaration. A sample declaration is shown in Figure 3-1. This declaration describes the interface to a 2-to-1 multiplexor.

```
entity multiplexor is
port      (  a,b : in bit;
             select_line : in bit;
             output : out bit
          );
end multiplexor;
```

Figure 3-1, Sample Entity Declaration

The ports of an entity are its communication channels with the outside world [RL 86]. A port declaration consists of a mode and a type. The mode specifies the direction of information flow through the port. A mode of **in**[7] specifies input only, a mode of **out** specifies output only, and a mode of **inout** specifies bidirectional flow. The other two modes, **buffer** and **linkage**, are special. Their meanings can be found in the IEEE VHDL Language Reference Manual (LRM) [IEEE88].

A port type specifies the data values which the port can assume [RL 86]. For example, a standard data type is bit, which can assume the values '0' or '1'. Another common type is integer. VHDL also supports composite types such as arrays and records. It is important to note that VHDL is a strongly typed language.

An entity declaration may also contain generics. Generics are constants used to increase the generality of an entity. A common use for generics is to pass timing information into an entity. This way, components of the same family can be substituted into a design without writing separate design units for each one. For example, if a design requires three NAND gates each with different timing, only one design unit need be written if generics are used to pass in the timing information.

---

7 For the remainder of this thesis, all VHDL keywords will be placed in **bold** letters.

3.2.1.2. Design Body

The function of the design entity is specified in an architecture. A sample architecture for the multiplexor is shown in Figure 3-2. VHDL supports three basic styles of functional description: behavioral, dataflow, and structural. Behavioral descriptions are the most abstract. They specify the output response to the inputs in algorithmic terms. Usually, little structure is implied. A dataflow description describes a function in terms of concurrently executing register transfer level (RTL) statements (Figure 3-2 is a dataflow description). It is less abstract than a behavioral description. The least abstract description style is structural. This style consists of interconnected components. Each component is instantiated in the architecture and wired together using signals. An architectural body is not limited to any one description style. Any mixture of the aforementioned styles may be utilized within the same architecture body.

```
architecture dataflow of multiplexor is
begin
   output <= a when select_line = '0' else
             b;
end dataflow;
```

Figure 3-2, Sample Architecture

The optional configuration specification binds a design body to an instantiated component. It provides the capability to bind an architectural body's components to similar, but not identical, design entities. For example, if three components are required which are functionally identical but differ in their timing, a configuration can be used to specify the timing data for each component.

3.2.2. The Testbench

A VHDL testbench is the highest level entity in a given simulation. It instantiates a design and drives the inputs in some prescribed manner. It also can perform sophisticated error checking because it has the power of a general purpose programming language at its disposal (see section 4.6). Using a testbench to test a design eliminates simulator dependence since no proprietary simulator command language is required.

3.2.3. Packages

Packages provide the VHDL modeler with a convenient method to group constants, types, signals, and subprograms so as to make them visible to multiple design units. A

package is composed of two parts, the package declaration and an optional package body. The package declaration contains the constant, type, and subprogram declarations, while the package body assigns values to the constants and fleshes out the subprograms. VHDL does not require every package to have a body. However, changes to the package body do not require design units referencing the package to be recompiled, whereas changes to the package declaration require recompilation of all design units referencing the package (see section 5.4).

3.2.4. Data Objects

To fulfill its function as a modeling language, VHDL contains three standard data objects: signals, variables, and constants. The fundamental data object in VHDL is the signal. Each signal is represented conceptually as a set of time-value pairs. The signal assumes the value in the pair at the simulation time specified. Each time a signal is assigned a value, a new time-value pair is added to the list[8] . Signals can be scheduled to take on a value after a given amount of time, such as in the statement

```
signal example 1:       signal clock : bit;
                        clock <= not clock after 10 ns;
```

They can also be assigned conditionally, such as in the following example

```
signal example 1:       signal output : bit;
                        output <= '1' when clock = '0' else '0';
```

Signals are also used to wire together components in a structural description.

As with most programming languages, VHDL provides the capability to declare and use variables. However, their use is much more restricted than that of signals because of the concurrent nature of VHDL . Variables can only be used within subprograms or processes to prevent them from being visible to multiple, concurrently executing processes.

```
variable example :      variable index : integer;
                        index := index + 1;
```

VHDL provides the ability to define and use constants. The constant must be assigned a value when it is declared, except for deferred constants in a package declaration.

---

8 This is not always true. The addition of a new time-value pair depends on what timing model, transport or inertial, was used and on the present time-value list. The meanings of each model can be found in Chapter 5 [Lip89].

```
constant example :      constant clock_period : time := 100 ns;
```

3.3. Description Styles Revisited

This section discusses some of the constructs within VHDL which facilitate the different modeling styles (i.e. behavioral, dataflow, and structural).

3.3.1. Processes

The process is VHDL's fundamental behavioral modeling construct as well as the fundamental unit of concurrency. All VHDL expressions have a corresponding process. A process is composed of three parts : the process declaration, an optional sensitivity list, and the process body. Note that if the sensitivity list is omitted, then at least one **wait** statement must be included[9]. Without either, the process will never suspend execution and thereby tie up the simulation. A single process may not have both a sensitivity list and a **wait** statement. The statements within a process execute sequentially within one unit of delta time, while all the processes in a given simulation execute concurrently.

A process is executed when one of three conditions is met. First, all processes are executed once up to the first wait statement (or entirely if a sensitivity list is included) when the simulation begins. Secondly, a process is executed when a signal in its sensitivity list changes. Finally, process execution resumes when the condition attached to a **wait** statement is met.

The following example shows a 2 to 1 synchronous multiplexor modeled using two different styles of processes, one with a sensitivity list and one with a **wait** statement.

```
package mux_package is
  subtype mux_type is bit;
  subtype control_type is boolean;

  constant clock_active : control_type;
  constant select_a : control_type;
end mux_package;

package body mux_package is
  constant clock_active : control_type := true;
  constant select_a : control_type := true;
end mux_package;

use work.mux_package.all;
entity two_to_one_mux is
```

---

9 This is not syntactically required. The analyzer will only give a warning that the process has neither a sensitivity list nor a **wait** statement.

```
  generic ( output_delay : TIME := 10 ns);
  port ( a,b : in mux_type;
         select_line : in control_type;
         clock : in control_type;
         output : out mux_type
       );
end two_to_one_mux;
```

Architecture 1	Architecture 2

`architecture two_to_one_mux_behavior of two_to_one_mux is`
`begin`
`  behavior : process (a,b,select_line,clock)`
`  begin`
`    if clock = clock_active and clock'event then`
`      if select_line = select_a then`
`        output <= a after output_delay;`
`      else`
`        output <= b after output_delay;`
`      end if;`
`    end if;`
`  end process;`
`end two_to_one_mux_behavior;`
	`architecture two_to_one_mux_behavior of two_to_one_mux is`
	`begin`
	`  behavior : process`
	`    begin`
	`      wait until clock = clock_active and clock'event;`
	`      if select_line = select_a then`
	`        output <= a after output_delay;`
	`      else`
	`        output <= b after output_delay;`
	`      end if;`
	`  end process;`
	`end two_to_one_mux_behavior;`

The package declaration contains two subtypes which abstract away the mux's input, output and select line types. This allows smooth conversion from high level modeling, where booleans and integers reign, to low level modeling where bits are prevalent. The two architectures given represent the two basic process styles : sensitivity lists and wait statement. Both architectures assign a new value to the output only on a rising clock edge. In architecture 1, the first **if** statement is executed each time signals a, b, select, or clock are updated but doesn't become true until a rising clock edge. The edge is detected using the predefined attribute **'event**. This attribute returns True when an event has just occurred on the attributed signal and False otherwise. In architecture 2, the process suspends at the **wait** statement until a clock rising edge. It then executes and suspends again at the **wait** statement. These two examples demonstrate two different methods of achieving the same result, a situation which occurs often in VHDL.

3.3.2. Signal Assignments

Signals may also be assigned values outside of processes[10]. All forms of signal assignment outside of processes are concurrent in nature. For example, the following two assignments occur simultaneously :

---

10 A duality, discussed in section 3.3.4, exists between signal assignments inside and outside of processes.

```
architecture example of example is
begin
  a <= b + 1;
  c <= b + 3;
end example;
```

VHDL also offers the modeler selected signal assignments, which are essentially case statements, and conditional signal assignments, which are essentially cascaded **if-then-else** statements (section 3.2.1.2 contains an example).

3.3.3. Blocks

Blocks are used in VHDL to organize groups of concurrent statements within an architectural body. The main advantage of blocks is that the block declaration can include a guard expression. This expression can be used to control signal assignments within the block[11]. Including a guard expression has the effect of creating an implicit Boolean signal within that block called "guard" which is True when the guard expression evaluates to True and False otherwise. Signal assignments within the block can be made conditional on the guard signal by using the reserved word **guard**. Figure 3-3 illustrates this technique. The signal test is only assigned the value '0' on the rising edge of the clock because it is a guarded assignment. As we will see in chapter 5, blocks can be used to model state machines.

```
architecture block_example of block_example is
  signal test : bit;
begin
  example : block (clock = '1' and clock'event)
  begin
    test <= guarded '0' after 100 ns;
  end block;
end block_example;
```

Figure 3-3, Block Example

3.3.4. Duality

VHDL has a very strong duality between concurrent and sequentially executed statements. If an action is implemented using concurrent signal assignments, there is an equivalent way to do the same thing using a process statement. The following example demonstrates this duality:

---

11 Blocks can also have ports and generics just like entities. This is to ensure duality, a concept discussed in the next section.

Concurrent	Sequential

output <= true **when** input = '1' **else** false;	**process** (input)
	begin
	  if input = '1' **then**
	  output <= true;
	  else
	  output <= false;
	  end if;
	end process;

VHDL also allows concurrent as well as sequential subprogram calls. Subprogram calls appearing within a process are sequential while those outside of any process are concurrent.

An exact duality also exists between component instantiations and blocks. An instance of a component inside an architecture can be replaced with a block statement with the same ports and generics as the component.

4. Motivation for Modeling

With all the hoopla surrounding VHDL, many people are asking "Why should I use it?" This chapter attempts to answer that question. It shows how VHDL can be used to shorten the design cycle and improve the quality of designs. In a few short years, VHDL will become ubiquitous in the digital design realm.

It is important to note that at the present time VHDL is not universally applicable to digital designs. Most of the tools are still too immature and standard model availability is still too limited for VHDL to be used for board-level design. However, tools are immediately available for ASIC design.

The first section discusses how to apply top-down design with VHDL. Subsequent sections cover the advantages VHDL has over gate-level design. These relate to design and description, concurrent design, complexity, and verification. The final two sections discuss when not to use VHDL and caveats for its use.

4.1. Top-Down Design

One of the great powers of VHDL is that it encourages true top-down design[12]. This methodology specifies that a design begins at a very abstract, behavioral level and is gradually worked down to a structural (gate) level. Each successive abstraction level is tested against the previous higher level for equivalence. Figure 4-1 displays the top-down method symbolically [Pain91]. Each lower level in the pyramid represents a more complex, less abstract step in the design cycle.

---

12 This is not to say that low-level design decisions can be completely deferred until the end. True top-down design typically means designing from the middle-out.

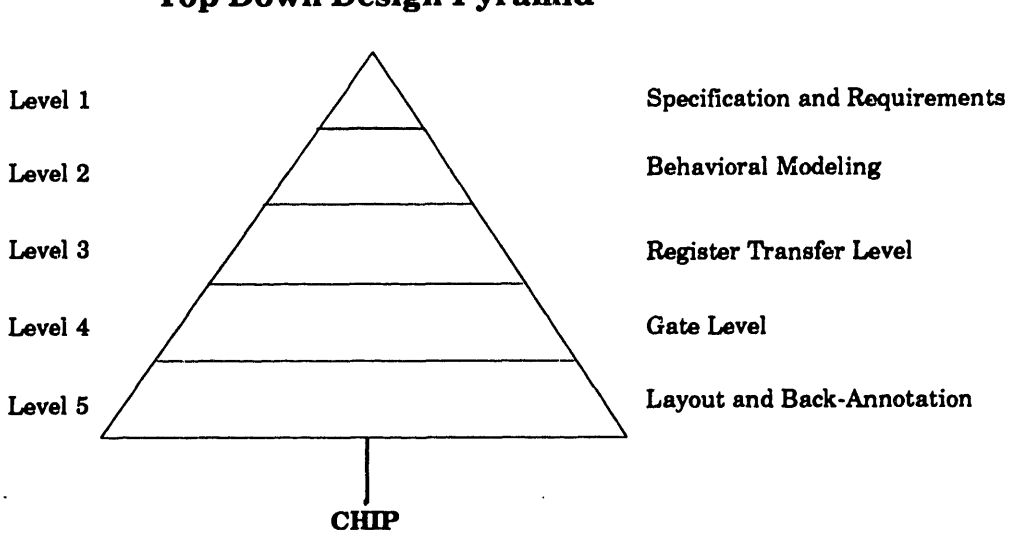


Figure 4-1, Top-Down Design Methodology

A true top-down design methodology can greatly simplify ASIC design. To see this, let's examine a typical design cycle with and without VHDL. Figure 4-2 shows the traditional design cycle (no VHDL). In general, the first task is to functionally specify the system – what inputs the design has and the functions it performs on those inputs to produce the output. A testing strategy is also developed at this point. The design is then parceled out to the members of the design team who begin drawing schematics, writing Boolean equations, and performing various other low-level design tasks. This is equivalent to skipping the second level of the pyramid. Concurrently, test vectors are generated. As each partition is completed, it is tested and revised, if necessary. When all partitions are complete, they are assembled and tested. At this point, the design, unless it is very small, will probably not work. Several (possibly extensive) revisions must be made before the design is complete and ready for placement, routing, and final simulation. If major architectural changes must be made the design will require extensive modifications. Thus, this method requires that a system be well-specified before actual design takes place to avoid compromising performance and/or functionality later in the design cycle.

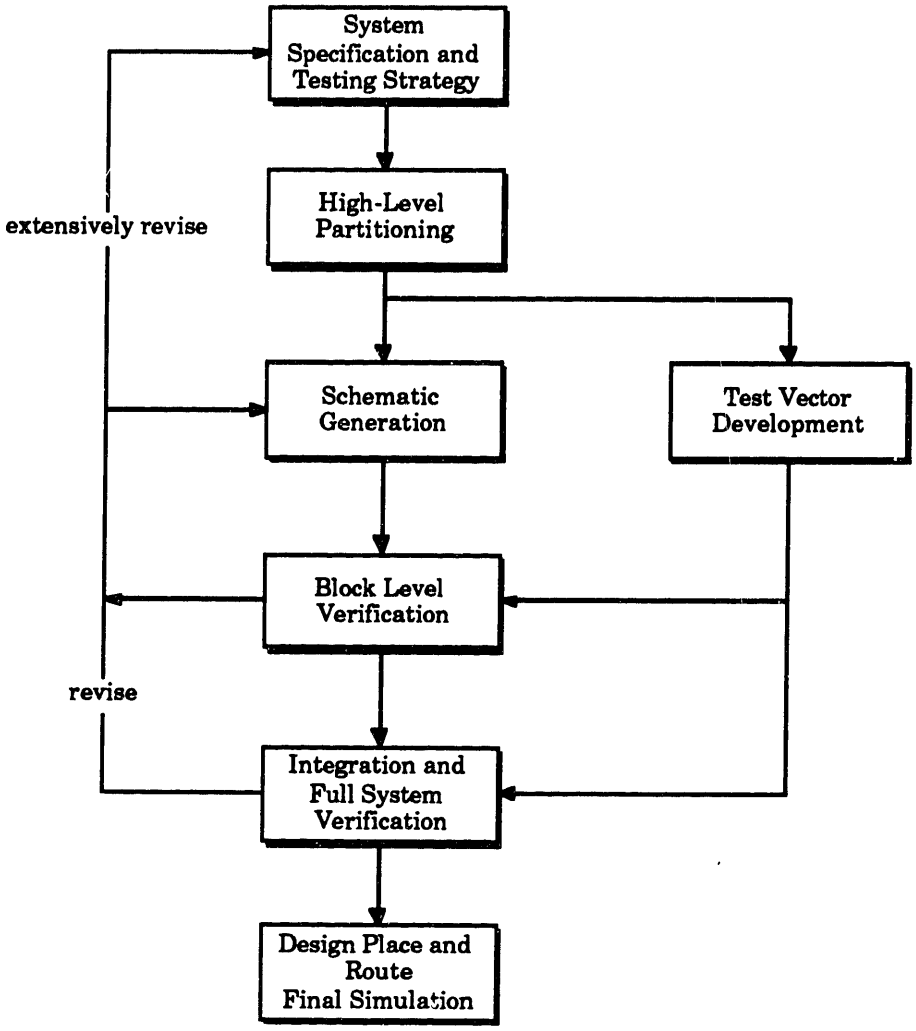


Figure 4-2, Conventional Design

Figure 4-3 shows the same design process using VHDL. This process follows the top-down methodology much more closely than does the conventional approach. The first two steps are again system specification– testing strategy and high-level partitioning. The third step when using VHDL is to develop behavioral models of the high-level components. These models are verified separately, "assembled" and tested with the VHDL testbench. Revisions at this level are inexpensive since even a complete redesign involves rewriting a relatively small amount of code. High-level architectural trade-offs can be made at this point. Once this initial model is complete, it can serve as the reference for subsequent, more structural models. Using VHDL at this level has the further advantage of allowing incremental testing of components. For example, if one team finishes their section before the others, their section can be substituted into the model in place of the behavioral model and tested by changing

configurations. The use of VHDL causes only minor revisions with respect to the level of abstraction at which they exist. In other words, the most radical modifications to the design, such as architectural tradeoffs, are made at the higher levels of abstraction where they are more tractable.

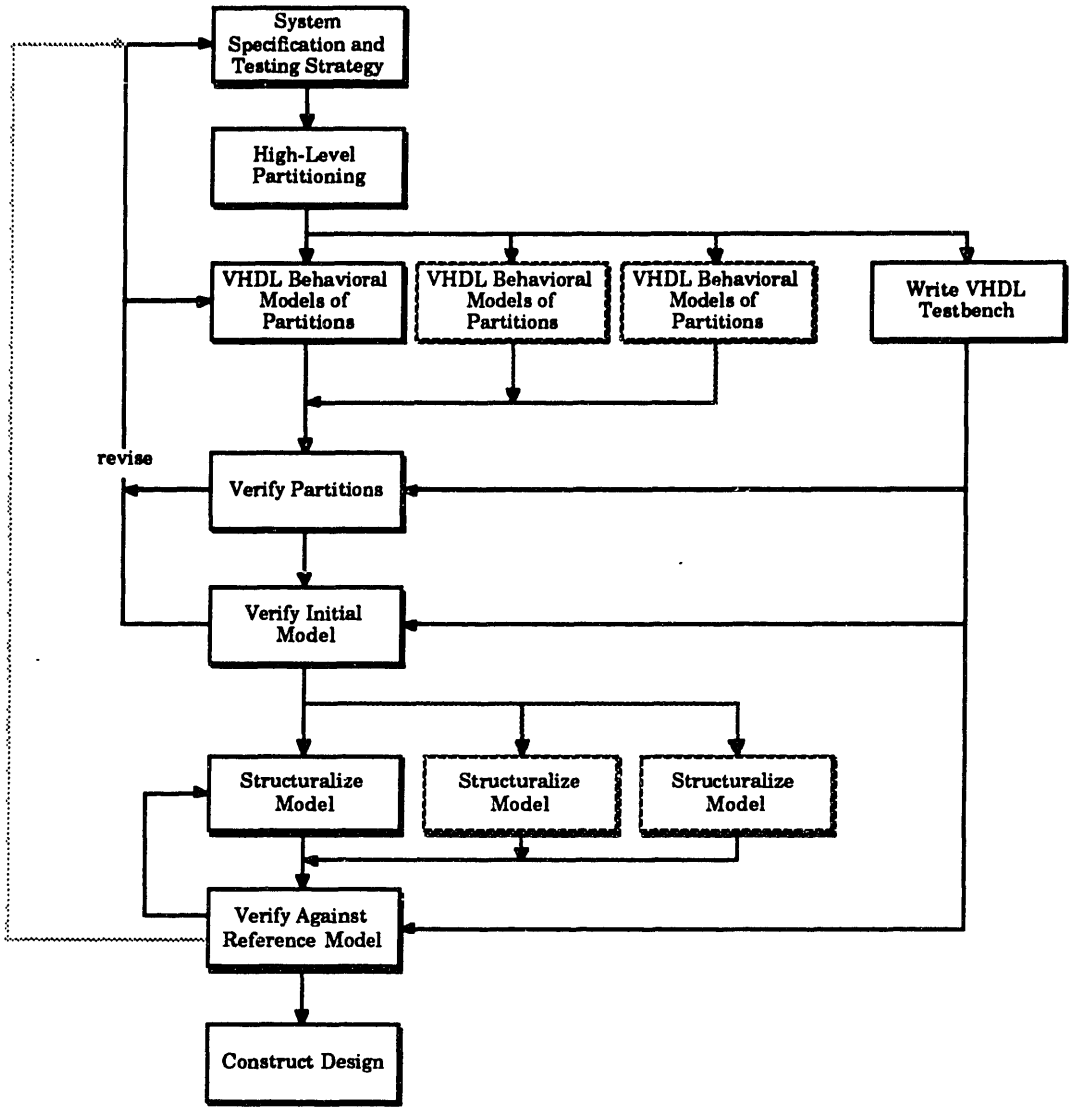


Figure 4-3, Top-Down Design with VHDL

As Figure 4-3 further indicates, test development can occur simultaneously with model development. A VHDL testbench (see Section 4.6) is a very flexible and powerful means to test a component. This testbench, once completed, can be used to test models at every level of abstraction with little modification.

The light grey line from the bottom to the top in Figure 4-3 represents respecification of the system after the gate-level models have been completed. This could happen if non-synthesizeable behavioral models are written (such those using access types). However, writing realistic behavioral models avoids costly redesign.

4.2. Synthesis

VHDL synthesis is perhaps the most exciting aspect of VHDL. Current synthesis tools are able to directly synthesize register-transfer level VHDL into a gate-level netlist. Synthesis can be an enormous time-saver. At the time of this writing, the best synthesis tools are equivalent to a digital designer with 14-15 years experience [Bohm91]. This makes them suitable for nearly any design.

VHDL synthesis also provides the ability to use VHDL throughout the entire design process, from concept to silicon. Even without synthesis capability, though, VHDL would still be very useful. Its use would cease at the RTL level where conventional design techniques would be more efficient. Synthesis avoids the need for this break in the top-down design methodology.

4.3. Design and Description in One

Another feature of VHDL is that it is actually two languages in one: a design language and a description language [Wax89]. Thus, it is useful for designing, testing, and documentation. A well written VHDL model is, in fact, self-documenting. Unlike a schematic diagram, which is practically useless in determining a system's overall function, a VHDL model (with accompanying testbench) can be read by humans and simulated by a machine, thereby forming a bridge between function and representation.

As a description language, VHDL is impressive in its ability to convey a designers intent. The downside to this is that VHDL is very verbose (like Ada). However, using VHDL becomes second nature after a couple of months and many tools are available which help the designer cope with the verbosity[13].

---

13 For example, entity-architecture-configuration templates can be used to avoid retyping.

4.4. Concurrent Design

VHDL modeling, due to the properties of the design entity, can proceed concurrently with testing. Furthermore, different designers can work on separate subsections independently and expect their various components to work together. All that is required is agreement on interfaces and function.

4.5. Complexity

VHDL provides many features for managing design complexity. First, VHDL supports design at all levels of abstraction, from the algorithmic level to the gate level[14]. Thus, when a design is in its infancy, its function can be specified as an algorithm which operates on abstract data types such as records, arrays, and integers. As the design matures, the abstract components can be replaced with models which operate on bits and bit vectors and specify their behavior using concurrent signal assignments.

Another complexity management feature is the **configuration** statement. Using the power of the **configuration** statement, one of several models can be selected for testing. Also, configurations allow the same component to be wired in with different generics. An example of this is a design which requires many NAND gates, some of which have different timing than the others. Configurations also allow the re-wiring of a components ports. This could be used in fault-testing, for example, to determine the affect of wiring a pin to ground.

4.6. Verification

VHDL makes the power of a complete, general purpose programming language available to test a design. It allows the designer to use complicated dynamic test structures and sophisticated hardware handshaking to test a design. VHDL provides these capabilities in a simulator independent manner. While some simulator command languages have the previously mentioned capabilities, none are portable across simulators. A VHDL testbench is guaranteed[15] to run on any simulator which supports VHDL since VHDL is non-proprietary.

---

14 Transistor-level modeling can be done [RW 89], but it is much more difficult because VHDL has no direct constructs to support analog behavior.

15 The TEXTIO package is an exception to this rule since it is ambiguously defined in the LRM. Portability may not be perfect if this package is used.

Another VHDL verification advantage is that functional test vectors can be derived directly from the testbench with little or no modification. VHDL practically eliminates the problem with ASICs passing foundry vectors but failing to work in the system.

Because VHDL is a DoD and IEEE standard, designs produced today using VHDL can still be simulated five years in the future, even if the original tool is no longer available. Thus, effort expended in VHDL modeling is never wasted because the standard insulates the modeler from company failures and tool obsolescence. Furthermore, VHDL models will accelerate re-implementation of a design at a later time.

4.7. When not to Use VHDL

Though VHDL can be extraordinarily useful, there are situations where it should not be used (at least currently). One such instance is in designs with mixed analog and digital hardware. VHDL simply cannot adequately model the analog section, though some people claim it can. VHDL could be used to model the digital section (with all the benefits thereof). However, a full system simulation would be very difficult since a link between the analog simulator and VHDL would have to be established (via files, pipes, etc.). Mixed analog-digital designs complicate simulation and verification.

Another situation in which the advantages of using VHDL are diminished is in incrementally updating a previous design which didn't use VHDL. In this case, a VHDL model of the entire design would have to be written and verified from scratch. If the fix is minor, VHDL probably shouldn't be used. However, if the design might be re-targeted to a different technology at a later date, developing a VHDL model and testbench for it would not be in vain.

4.8. Caveats

Throughout this chapter I have attempted to explain the overwhelming advantages of using VHDL for digital design. This final section lists some warnings about using VHDL.

- The strongest caveat deals with transitioning a design from the behavioral level to the structural level. The smoothness of this process depends in part on the port types of the entities in the original behavioral model. Types which are easily translated to bit representations make for a smoother transition. Another danger lies with non-buildable constructs. Writing unbuildable code in VHDL is easy,

so care must be taken to avoid constructs with no hardware analog, such as access types (access types are pointers).

- Though VHDL possesses tremendous power for testing, it cannot test all designs. For example, if a design requires a large number of test vectors which can only be produced using a random algorithm, VHDL cannot do the job. In this case, a C language program could be used to write test vectors to a text file which the VHDL testbench could then read in and apply[16].

- Some people may look upon VHDL as the death blow to hardware engineers. They may point out that developing VHDL models is the task of software engineers. They could not be more wrong. Though software engineering principles are necessary in managing model development, coding in VHDL is much like designing hardware.

- The need for careful planning at the inception of a design has not been eliminated. With excellent synthesis tools available, it is tempting to begin "playing" with the tool immediately after a design has begun. This is unproductive. A thorough understanding of aggregate hardware requirements is necessary before synthesis should be attempted.

---

16 This is what I had to do with the scoreboard VHDL model.

5. Behavioral Modeling Considerations

The purpose of this chapter is to present issues related to abstract behavioral modeling in VHDL. An abstract behavioral model is one in which very little implementation specific information is used. For example, records and integers are used instead of bits and bit_vectors. The advantage of abstract modeling is that such things as data path width and address spaces are deferred until the implementation phase of the design. The disadvantage is that, before the model can be converted to a physical behavioral model, the entity declarations must be changed. This can be difficult and time-consuming in a concurrent design environment. Yet, for an algorithm as complicated as the scoreboard's, the advantage of quicker and easier functional verification outweighs the need to rewrite entity declarations.

The first section discusses the various methods for modeling state machines. State machine modeling is important because the scoreboard model contains many state machines, as will most VHDL models. The second section covers high-level modeling considerations such as synchronous designs and timing. The third section discusses how to use VHDL's compilation dependencies. The fourth section discusses the use of resolution functions in behavioral modeling, while the final section talks about the use of subprograms.

5.1. State Machine Modeling

The state machine is one of the basic components in almost every digital design. As such, the need will often arise to model state machines in VHDL. This section discusses four methods for modeling state machines. The unifying example is of the simple state machine shown in Figure 5-1.

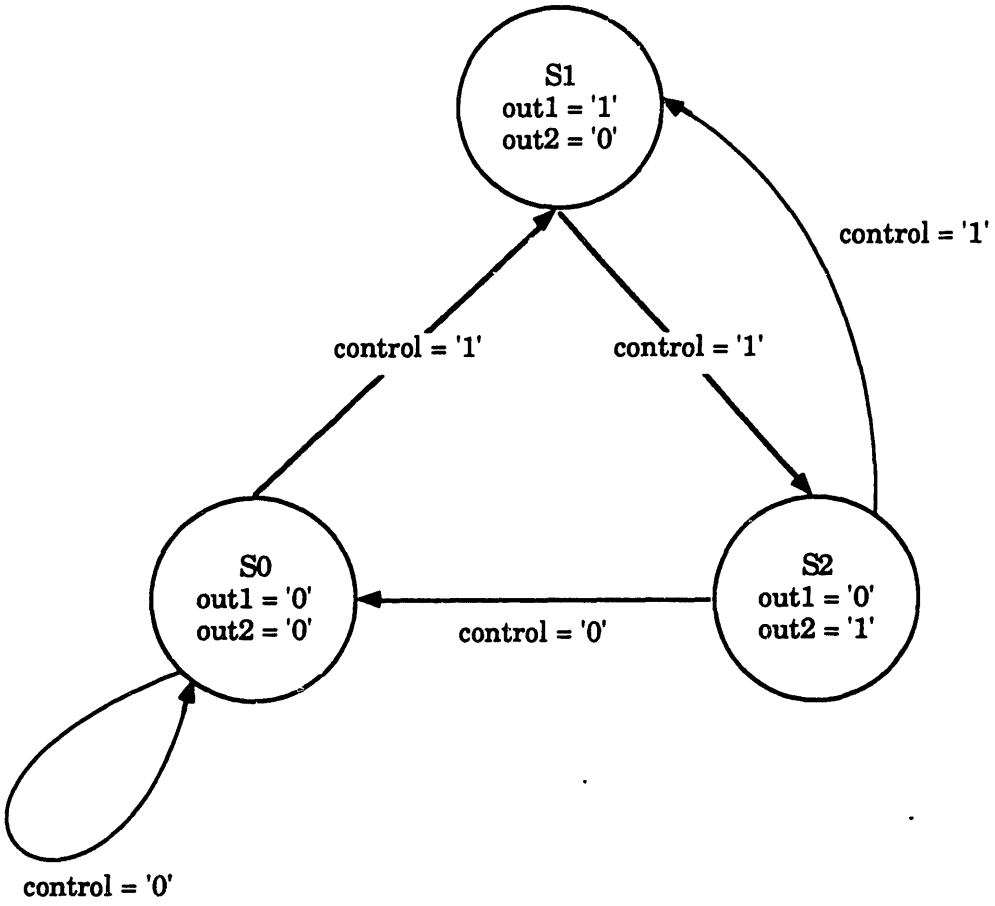


Figure 5-1, Example State Diagram

Figure 5-2 contains the package used by all the examples. It abstracts away the input and output types and provides constants for their active states. With this method, it is very easy to change from, for example, active high to active low logic. Notice that the state type includes four states but the example state machine only requires three states. This disparity will be used later on to demonstrate how the four methods handle trap states.

```
PACKAGE state_machine_package IS
  TYPE state_type IS (s0,s1,s2,s3);
  SUBTYPE control_type IS BIT;
  SUBTYPE output_type IS BIT;

  CONSTANT clock_active : control_type;
  CONSTANT control_active : control_type;
  CONSTANT output_active : output_type;

END state_machine_package;


PACKAGE BODY state_machine_package IS

  CONSTANT clock_active : control_type := '1';
  CONSTANT control_active : control_type := '1';
  CONSTANT output_active : output_type := '1';

END state_machine_package;
```

Figure 5-2, State Machine Package

The entity declaration for the state machine is contained in Figure 5-3 below. The two generics, output_delay and state_delay, are used in adding delay to signal assignments. The control port is used to control the state transitions, while the reset port sets the state to s0 when active. The clock controls the transitions, and out1 and out2 are the two outputs.

```
ENTITY state_machine IS
  GENERIC (
            output_delay : TIME := 1 ns;
            state_delay : TIME := 1 ns
            );
  PORT (
          control,reset : IN control_type;
          clock : IN control_type;
          out1,out2 : OUT output_type
          );
END state_machine;
```

Figure 5-3, State Machine Entity Declaration

Perhaps the best method for modeling state machines in VHDL is with a **CASE** statement on a state signal within a process. A variable, typically called next_state, is assigned based on the current state and the inputs. The state signal is assigned at the end of the process. Figure 5-4 shows an example of a state machine with a **CASE** statement.

```
ARCHITECTURE best OF state_machine IS
   SIGNAL state : state_type;
BEGIN
  machine : PROCESS (clock,reset)
   VARIABLE next_state : state_type;
  BEGIN
   IF reset = control_active THEN
     next_state := s0;
   ELSIF clock = clock_active AND clock'EVENT THEN
     CASE state IS

       WHEN s0 =>
         out1 <= NOT output_active AFTER output_delay;
         out2 <= NOT output_active AFTER output_delay;
         IF control = control_active THEN
            next_state := s1;
            out1 <= output_active AFTER output_delay;
            out2 <= NOT output_active AFTER output_delay;
         END IF;

       WHEN s1 =>
         IF control = control_active THEN
            next_state := s2;
            out1 <= NOT output_active AFTER output_delay;
            out2 <= output_active AFTER output_delay;
         END IF;

       WHEN s2 =>
         IF control = control_active THEN
            next_state := s1;
            out1 <= output_active AFTER output_delay;
            out2 <= NOT output_active AFTER output_delay;
         ELSE
            next_state := s0;
            out1 <= NOT output_active AFTER output_delay;
            out2 <= NOT output_active AFTER output_delay;
         END IF;

       WHEN OTHERS =>
         next_state := s0;
     END CASE;

   state <= next_state AFTER state_delay;

   END IF;
  END PROCESS;
END best;
```

Figure 5-4, **CASE** Statement Example

The first action of the process is to check for reset. As written, the reset is asynchronous. It can be made synchronous by making the first **IF** clause sensitive to the clock and then checking for reset. If reset is '1', the next state is set to s0, otherwise the normal state transition checks are performed. An **IF** clause within each **WHEN** clause controls transition to the next state. If a state transition condition is met, the next state is assigned. Nothing needs to be done if no transition occurs because the outputs retain the last value assigned to them. The output values (shown within the circles in Figure 5-1) are also set

within this **IF** clause. A **WHEN** clause is included for all valid states. The final **WHEN** clause ensures that any invalid states (trap states) cause the next state to be s0. The final signal assignment assigns the next_state variable to the state signal.

A variation on the CASE method is contained in Figure 5-5. It splits the state machine into synchronous and asynchronous parts. The synchronous process assigns the next_state signal to the state signal on a rising clock edge. The asynchronous process takes care of output and next state assignments. The main difference between Figure 5-4 and Figure 5-5 is that the output assignments occur asynchronously in Figure 5-5. The functionality is exactly the same otherwise.

```
ARCHITECTURE also_good OF state_machine IS

   SIGNAL state,next_state : state_type;

BEGIN

  asynchronous : PROCESS (state,control)
  BEGIN
   CASE state IS

     WHEN s0 =>
       out1 <= NOT output_active AFTER output_delay;
       out2 <= NOT output_active AFTER output_delay;

       IF control = control_active THEN
         next_state <= s1;
       END IF;

     WHEN s1 =>
       out1 <= output_active AFTER output_delay;
       out2 <= NOT output_active AFTER output_delay;

       IF control = control_active THEN
         next_state <= s2;
       END IF;

     WHEN s2 =>
       out1 <= NOT output_active AFTER output_delay;
       out2 <= output_active AFTER output_delay;

       IF control = control_active THEN
         next_state <= s1;
       ELSE
         next_state <= s0;
       END IF;

     WHEN OTHERS =>
       next_state <= s0;
   END CASE;
  END PROCESS;

  synchronous : PROCESS (clock,reset)
  BEGIN
   IF reset = control_active THEN
     state <= s0 AFTER state_delay;
   ELSIF clock = control_active AND clock'EVENT THEN
     state <= next_state AFTER state_delay;
   END IF;
  END PROCESS;
END also_good;
```

Figure 5-5, **CASE** Variation

There are many advantages to the **CASE** method (both variations). First, the **CASE** method is very clear. It is not difficult to recognize the correspondence between the VHDL in Figures 5-4 and 5-5 and the state diagram in Figure 5-1. A second advantage is that it is not limited to simple state machines. It can handle any number of states (though the **CASE** statement becomes unwieldy with too many states) and any number of inputs and outputs. The later methods do not share this advantage. Also, with the **CASE** method it is very easy to

add either synchronous or asynchronous reset capability and trap state handling. Finally, the **CASE** method is directly synthesizeable by VHDL synthesis tools such as the Synopsys Design Compiler© [Syn90].

A second method for modeling state machines was presented by Armstrong in his book [Arm87]. It uses nested **BLOCK**s and guarded signal assignments. Figure 5-6 contains an example. The state signal is declared as a **REGISTER**. Registered signals retain the last value assigned to them when all their drivers have been disconnected (through **BLOCK** guards evaluating to False)[17]. The outer **BLOCK** is guarded on the rising edge of the clock, while all inner **BLOCK**s are guarded by the Boolean **AND** of the outer guard and their respective states. The outputs are assigned based on the value of the state signal.

---

17 For more information, see Chapter 5 of VHDL : Hardware Description and Design.

```
ARCHITECTURE block_state_machine OF state_machine IS

  TYPE state_array_type IS ARRAY (NATURAL RANGE <>) OF state_type;

  FUNCTION state_resolver (state_array : IN state_array_type)
  RETURN state_type IS
   VARIABLE resolved_value : state_type;
  BEGIN
   FOR i IN state_array'RANGE LOOP
     resolved_value := state_array(i);
   END LOOP;
   RETURN resolved_value;
  END;

  SIGNAL state_register : state_resolver state_type REGISTER;

BEGIN

  synchronous : BLOCK (clock = clock_active AND clock'EVENT)
  BEGIN

   state0 : BLOCK (((state_register = s0) AND guard)
            OR (reset = control_active))
   BEGIN
     state_register <= guarded s1 AFTER state_delay
         WHEN control = control_active ELSE s0;
   END BLOCK state0;

   state1 : BLOCK ((state_register = s1) AND guard)
   BEGIN
     state_register <= guarded s2 AFTER state_delay
         WHEN control = control_active ELSE s1;
   END BLOCK state1;

   state2 : BLOCK ((state_register = s2) AND guard)
   BEGIN
     state_register <= guarded s1 AFTER state_delay
      WHEN control = control_active ELSE s0 AFTER state_delay;
   END BLOCK state2;

   out1 <= output_active AFTER output_delay
       WHEN state_register = s1 ELSE NOT output_active;

   out2 <= output_active AFTER output_delay
       WHEN state_register = s2 ELSE NOT output_active;

  END BLOCK synchronous;
END block_state_machine;
```

Figure 5-6, Nested Block Example

The advantages of this method are that, once again, the correspondence between the VHDL and the state diagram is good. Adding a synchronous reset is also simple. The disadvantages of this method are that a **BLOCK** must be written for each possible state and each output must have its own selected or conditional assignment. Thus, this method becomes unwieldy for even medium size state machines. One final disadvantage is that the author was unable to get it to work. In sum, this method should be avoided.

One final method for implementing state machines is through a conditional signal assignment statement. Figure 5-7 contains an example. The state transitions and output assignments are placed within **WHEN** clauses. Trap states are handled by the final **WHEN** clause. Sate assignment is done inside a synchronous **PROCESS.**

```
ARCHITECTURE ugly OF state_machine IS
  SIGNAL state,next_state : state_type;
BEGIN
  next_state <= s0 WHEN ((reset = control_active) OR
          ((control = NOT control_active)
          AND (state = s2))) ELSE
       s1 WHEN (((state = s0) OR (state = s2))
          AND (control = control_active)) ELSE
       s2 WHEN ((state = s1)
          AND (control = control_active)) ELSE
       s0;

  out1 <= output_active AFTER output_delay
       WHEN state = s1 ELSE NOT output_active;

  out2 <= output_active AFTER output_delay
       WHEN state = s2 ELSE NOT output_active;

  synchronous : PROCESS(clock)
  BEGIN
   IF clock = control_active AND clock'EVENT THEN
     state <= next_state AFTER state_delay;
   END IF;
  END PROCESS;

END ugly;
```

Figure 5-7, Conditional Signal Assignment Example

The conditional signal assignment method suffers from a lack of clarity. By duality, Figure 5-7 could be replaced with an equivalent **PROCESS** closely resembling that of the **CASE** method in Figure 5-4. Thus, in general this method should be avoided and the **CASE** method used instead.

5.2. Synchronous designs

When constructing abstract behavioral models, it is critical that fundamental assumptions about the underlying hardware not be violated. One such assumption is synchronicity. Just because the model is abstract doesn't mean that this basic notion of

digital hardware can be violated. Thus, it is good practice to make all processes sensitive to a clock edge[18] unless that process is modeling a section of combinational logic.

A side effect of constructing synchronous models is that reasonably accurate performance estimates can be obtained very early in the design process. The performance information is derived from the number of clock cycles the model requires to perform its functions. The information is accurate only if reasonable assumptions about underlying hardware operations are made. For example, it is unreasonable to assume that an integer multiply will be completed within 40 ns, but it is reasonable to assume that a memory read will require two clock cycles. This knowledge can be built into the model via a state machine which asserts an address and then waits one clock cycle, for example. Such performance estimates can then be used to assist in the specification of related components.

5.3. Timing

Realistic timing does not belong in an abstract model since such information cannot be extracted from the design at this stage. However, dummy delays are useful for making signal transitions more visible. Without them, all transitions occur one delta delay after an assignment, thereby making waveform displays more difficult to read. Readability is thus enhanced by adding a delay to all signal assignments. Such a delay can be an integral division of the clock period, for example.

5.4. Compilation Dependencies

VHDL's compilation dependencies, illustrated in Figure 5-8, can be used to the designer's advantage. Modifications made to higher level design units require recompilation of all lower level design units which reference it. VHDL allows separate compilation of package headers and bodies. Modifications made to the package body only require that the body be recompiled. Modification of the package header requires that all design units referencing that package be recompiled. Thus, it is good practice to separate the package body and header. The one exception is that modifying types or subtypes will always cause the package header to need recompilation since these declarations cannot be deferred to the package body.

---

18 Of course, if asynchronous hardware is being modeled this should not be done.

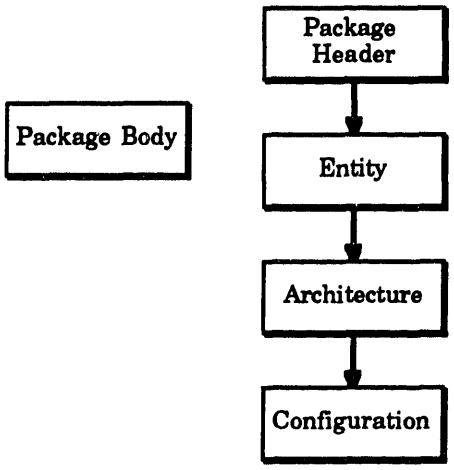


Figure 5-8, Compilation Dependencies [VHDL90]

The organization of packages can drastically affect recompilation. For example, if one package is used to define all types, constants, and subprograms, then nearly every design unit will have to reference the package. It is better to organize packages by function, i.e. place related constants, types, and subprograms together in the same package. This way, only a few design units will reference each package, thereby reducing recompilation if the package headers need to be modified.

5.5. Resolution functions

Resolution functions can be very useful in abstract behavioral modeling. However, their use is fundamentally limited by their physical interpretation of resolving the value of signals with more than one driver. In the case of the logic resolution function included with all VHDL simulators, this interpretation is intuitive since all digital designers know that, for example, a high-impedance, or 'Z', and a '1' result in a '1', and a '1' and a '0' result in an unknown condition, or 'X'. What if, however, a record type signal requires multiple drivers? A resolution function can be written for a record type, since any type may be resolved. This can be done by adding a Boolean field to the record called "high_z" (or similar). The resolution function can then be written to ignore all drivers whose high_z field equals True. Similarly, a high_z value can be added to enumerated types and resolution functions written to ignore all such values. The only problem is deciding what value to assign when more than one non-high_z driver exists (this is usually an error which could be flagged by an **ASSERT** statement).

The multiple driver problem usually arises in abstract behavioral modeling of data buses. An example should make things clearer. When constructing a model for a memory, it is natural to make the data port of mode **inout**. If this is done, then the port type must be resolved since both the memory and whatever is trying to write to the memory will be driving the port. If the memory stores abstract data such as records or enumerated types, a resolution function must be written in accordance with the guidelines mentioned in the previous paragraph. Another, somewhat simpler solution is to add explicit input and output ports to the memory. No resolution function is required, but the model is less clean.

But what about the address input to the memory ? Oftentimes, more than one signal will need to assert an address. Should a port be added for every such signal? To avoid port proliferation in this case, a resolution function should be written. This solution makes sense since, in general, all addresses in a given model will be of the same base type (i.e. integer). A high-impedance address can be chosen (such as -1 or integer'**right**) and the resolution function designed to ignore all drivers of that value. An example of such a function is given in section 7.2.1.2.

5.6. Subprograms

Subprograms are a very useful abstraction mechanism in behavioral modeling. They can be used to perform complex functions for which a hardware method does not yet exist. For example, in the initial scoreboard model, voting is done via multiple procedures and functions. Constructing the voter in this manner allowed the modeling to proceed quicker. Furthermore, if the subprograms are placed in a package, modifying them will require a smaller recompilation penalty than modifying architectures.

6. Scoreboard Functional Description

This chapter contains the complete functional description for the scoreboard. The first section covers the SERP and CT formats. The second section discusses the overall design goal, which is to optimize the common case. The third section discusses in depth the two major SERP processing phases. The final two sections cover fault conditions and the other functions the scoreboard must perform.

6.1. SERP Format

Each SERP entry has the form given in Figure 6-1. Each field is 8 bits wide, making each entry 32 bits in length.

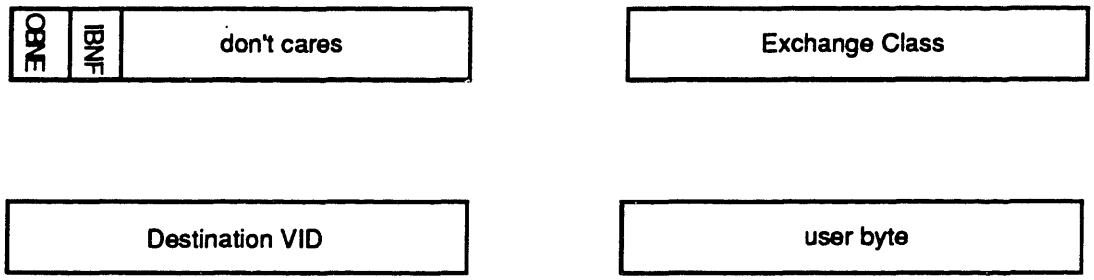


Figure 6-1, SERP Entry

The first byte contains the OBNE and IBNF bits for that PE. The rest of the bits are unused. The second byte contains the exchange class. A breakout of the bits in this field can be found in section 6.3.3.2. The third byte contains the destination VID for the message (if any), while the fourth byte is user defined. The exchange class, destination VID, and user byte are considered invalid by the scoreboard unless the OBNE bit is set.

6.2. CT Format

The form for a CT entry is shown in Figure 6-2. Each field is eight bits wide and the entire entry consumes 8 bytes.

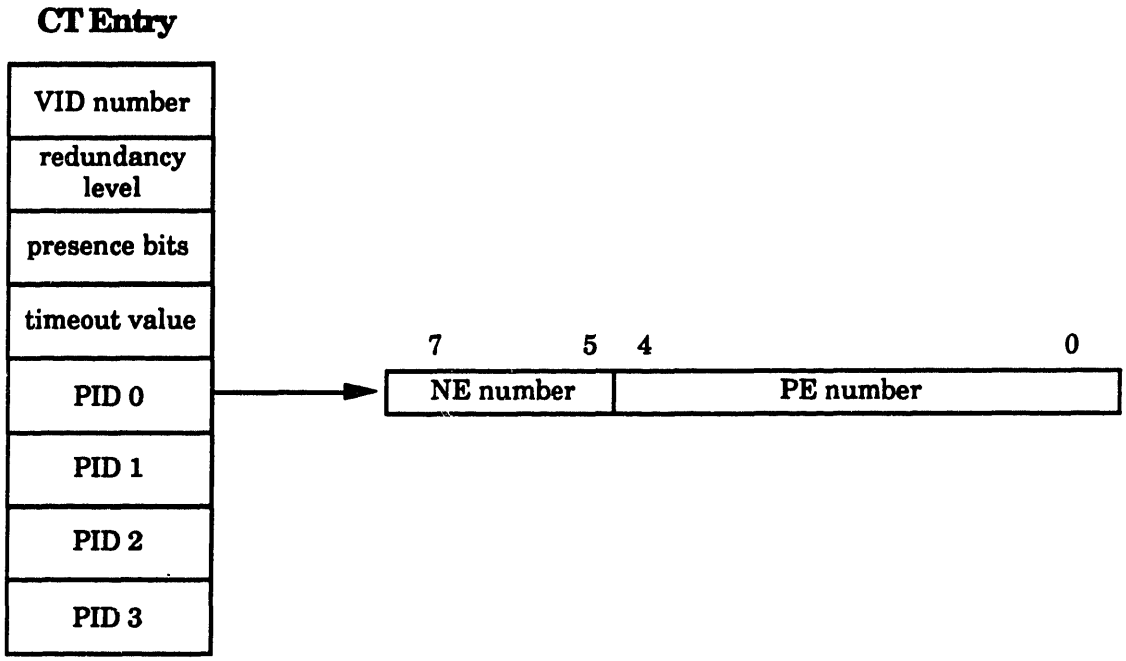


Figure 6-2, Configuration Table Entry

The first entry is the VID number, which can take on any value between 0 and 255[19]. The second field contains the redundancy level. The redundancy level can be either zero (for a non-active VID), one, three, or four. The third entry contains the presence bits. The form for the presence bits is shown in Figure 6-3. A presence bit is set when the VID has a member on the corresponding NE. The fourth entry is the value to use when calculating timeouts on the VID (how this entry is used is explained in section 6.3.1.4). The next one to four bytes contain the PIDs of all the VID members. The form of the PID is shown to the right of Figure 6-2. It is a simple encoding of an {NE,PE} pair – three bits for the NE and five for the PE. This allows a theoretical maximum of 8 NEs with 32 PEs each, more than enough for any realistic configuration.

x	x	x	NE 4	NE 3	NE 2	NE 1	NE 0

Figure 6-3, Format of the Presence Bits

19 Currently, the entire CT is composed of 256 entries (one for each possible VID). This could easily be reduced to save memory.

An example CT entry for a triplex is shown in Figure 6-4. The redundancy level field is Binary"011", or three, and the presence bits reflect the fact that the VID has members on NE's zero, two, and four. The fourth entry is the timeout value, while the last three entries are the PIDs of the members.

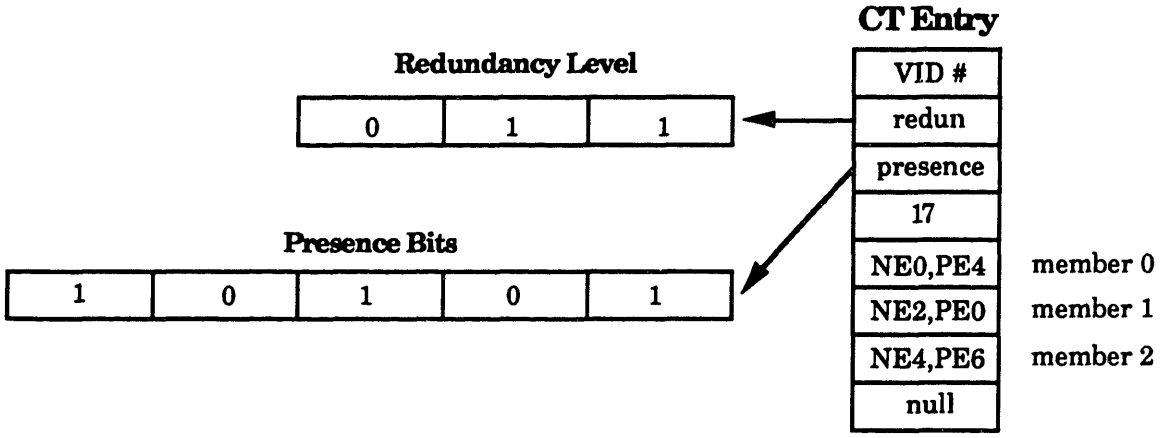


Figure 6-4, Example Configuration Table Entry

The entire CT, composed of 256 CT entries, is stored as a single block in memory. The entry for a given VID can be retrieved by multiplying the VID number by eight (or shifting it three places to the left) and using offsets 0 -7 to retrieve specific fields.

6.3. Goal : Optimize The Common Case

With most digital hardware, speed is of the essence. The scoreboard is no exception. In this case, speed can mean the difference between a viable real-time fault-tolerant parallel processor and a nifty laboratory prototype. This is because message passing latency is critical to real-time systems such as the FTPP. The scoreboard has a dominant affect on the latency of inter-processor messages on the FTPP.

The scoreboard SERP processing latency is the time span between the NE's global controller signalling the scoreboard to begin processing a new SERP and detection of the first ready message in the current SERP (assuming a valid message exists within that SERP). This latency limits the iteration rate of periodic tasks on the FTPP.

The scoreboard is designed to optimize SERP processing as much as possible. The two most common actions in SERP processing are voting and checking timeouts. Voting is a

common operation since all information in the SERP must be voted before it can be used by the scoreboard. However, the commonality of checking timeouts is not so obvious.

Consider a triplex which desires to send a message. Because the PE's composing the triplex are only loosely synchronized, one of them will be slightly behind the other two (but still within a bounded skew, as per the synchronization requirement of Byzantine Resilience). The lagging PE will set its OBNE bit after the other two. It is probable that, since a SERP cycle is shorter than the maximum skew, the scoreboard will see two asserted OBNE bits and one unasserted one. Though the lagging PE is not faulty, a timeout will have to be set because unanimity does not exist. If the PE responds before the timeout expires, the PE remains non-faulty and synchronized with the remaining PEs. If the timeout expires, the faulty PE is ignored. IBNF timeouts are handled in a similar fashion.

6.4. SERP Processing

In processing the SERP, the scoreboard passes through two phases. First each VID's SERP entries are voted and written into an intermediate storage called the voted SERP memory. Then the voted SERP is scanned for valid messages. This is the parallelization of a sequential algorithm presented in the next paragraph.

A sequential algorithm for processing the SERP is shown in Figure 6-5 [Mor91]. The algorithm assumes the existence of a look-up table which translates a VID number to its corresponding PIDs within the SERP. This table can be generated from the CT. The algorithm begins by reading the OBNE bits of the first VID and voting them. If the result is unanimous (or majority plus timeout), it votes the destination VID field[20], pulls its entries out of the SERP, and votes their IBNF bits. If the IBNF result is unanimous (or majority plus timeout), the exchange class and user bytes are voted and the message sent.

---

20 For the sake of brevity, I'll ignore the special case of broadcasts. The algorithm is essentially the same without them.

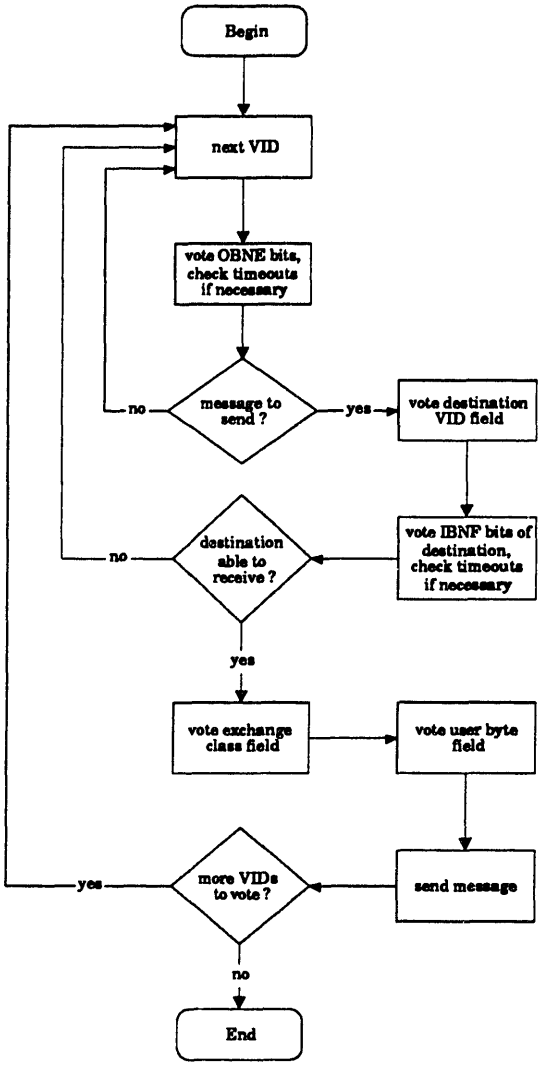


Figure 6-5, Sequential Scoreboard Algorithm

This algorithm, while ideal for a computer program, has problems when translated to hardware. The first problem is voter utilization. In the algorithm as presented, a hardware voter would be idle while timeouts were being checked. It makes sense to pipeline the voting such that the OBNE and IBNF bits are voted first and the other SERP fields are voted while timeouts are being checked.

The existing scoreboard algorithm reflects the pipelining idea. It uses on-board look-up tables extensively for efficient indirection. The penalty, besides additional memory, is longer reset and CT update times since the look-up tables must be regenerated after these operations. The remainder of this chapter functionally describes the scoreboard. Chapter 7

discusses the execution of these functions in greater detail. A flowchart representation of the entire algorithm can be found in Appendix 10.1.

6.4.1. Voting

Because of the special nature of the scoreboard, it cannot use a generic, four way, bit for bit majority voter. This is unfortunate since many such designs already exist. The scoreboard voter varies from conventional designs in the following ways:

① The voter has different rules for determining the majority result based on the redundancy level of the input and the data being voted.

② The voter is not masked in the normal sense.

③ Syndrome generation changes based on the redundancy level of the input and on the location of the VID's members.

6.4.1.1. Majority Rules

The scoreboard uses the following rules for determining if a majority of the inputs agree :

Redundancy Level	Type of Data	Majority
simplex	OBNE	1 of 1
simplex	IBNF	0 of 1
simplex	data bit	1 of 1
triplex	OBNE	2 of 3
triplex	IBNF	2 of 3
triplex	data bit	2 of 3
quad	OBNE	3 of 4
quad	IBNF	3 of 4
quad	data bit	2 of 4*

* 3 of 4 is also a valid majority for a quad.

The first column specifies the redundancy level of the input. The second column specifies the type of data being voted, whether OBNE, IBNF, or data. Data includes the destination VID, exchange class, and user byte. The final column indicates the number of inputs which must be asserted, or '1', for the majority condition to exist. Two cases are of special note. The first is the case of voting the IBNF bit of a simplex. The 0 of 1 majority condition specifies that a timeout should be set on a simplex whenever its IBNF bit is not set. This prevents a faulty simplex from holding up VIDs trying to send it messages. The second

notable case is that of voting quads. For the OBNE and IBNF bits, an unambiguous majority is required (3 of 4), while data requires only 2 of 4 to agree. This ensures that if a two-two split occurs (two members say one thing, two another) when voting either the OBNE or IBNF bits the "safe" option is taken. It is better to not send a message or risk overwriting a PE's input buffers until a clear manifestation of faulty conduct is seen. Since there is no clear "safe" option for data, either 2 of 4 or 3 of 4 may be used.

6.4.1.2. Masking

The scoreboard's voter is not maskable in the normal sense of "masking out" some input bits to prevent them from contributing to the voted result. Instead, inputs are masked automatically based on the redundancy level. Conceptually, the voting logic contains four pipeline registers, numbered 0 to 3, which feed the voter (reference Figure 6-6). When voting a simplex, the value to vote will always reside in register 0. When voting a triplex, the values will always reside in registers 0 to 2. Values for a quad will occupy all four registers. Inputs are voted based on the redundancy level and the data type as described in section 6.4.1.1. Because of this arrangement, no inputs need to be masked out. Figure 6-6 shows the conceptual representation of the scoreboard voter.

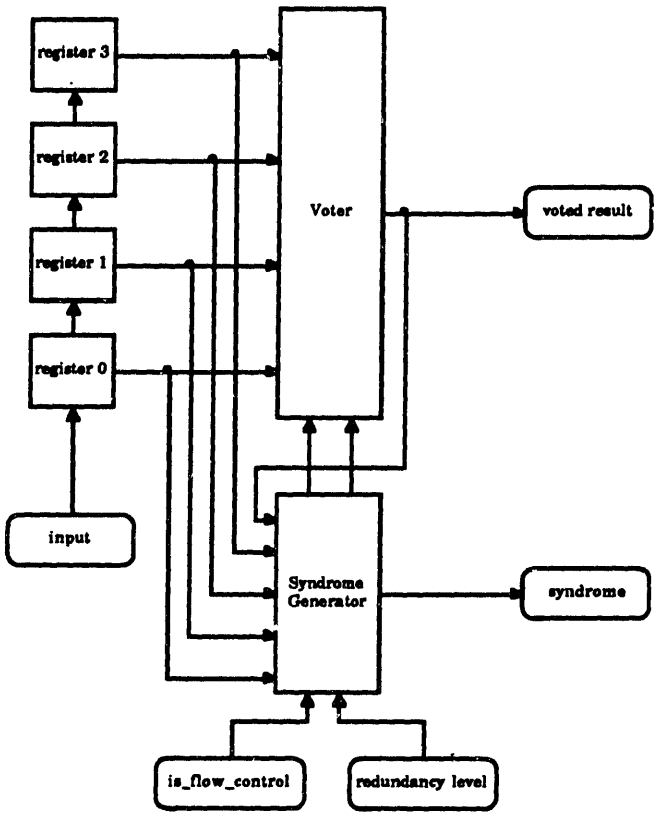


Figure 6-6, Conceptual View of the Voter

An implication of this is that, for example, a triplex always has three values to vote, even if one of those values originates from a faulty PE. In general, the data from a known faulty PE would be masked out to prevent it from contributing to the voted result. However, if reintegration of the faulty PE is desired, it must be allowed to participate in the voting so that it can be checked for continued faulty conduct. If it was masked out, there would be no way to check if the fault was transient or permanent. The disadvantage of this method is that, for permanently faulty PEs, the full timeout period must be paid each time a message is sent or is received by the VID until a CT update can be performed (see section 6.6 for a further explanation of CT updates).

6.4.1.3. Syndromes

The scoreboard voter must produce three different syndromes – OBNE, IBNF, and data syndromes. The syndromes are in CT-absolute format (see Figure 6-7), which means that each bit corresponds to a PID in the VID's CT entry, with the least significant bit corresponding to the first member in the VID's CT entry. A bit is set in the OBNE and IBNF

syndromes when a member of the VID times out. Bits corresponding to non-existing members are guaranteed to be '0'. For example, if the OBNE bits for the triplex presented in section 6.1 are :

member 1 : **set**
member 2 : **not set**
member 3 : **set,**

then the voted result is **OBNE set** and the OBNE syndrome will be as shown in Figure 6-8. The syndrome bit for member 2, which is the second least significant bit, is set. The syndrome bits for members 1 and 3 are unset. The other syndrome bit, which corresponds to the non-existent fourth member, is a '0'.

member 4	member 3	member 2	member 1

Figure 6-7, Syndrome Format

0	0	1	0

Figure 6-8, Example OBNE syndrome

The third syndrome produced by the scoreboard, the data syndrome, is generated differently then the other two syndromes. It is generated by OR'ing the syndromes from the destination VID, exchange class, and user byte. A set bit in this syndrome indicates non-agreement in one or more bits in one or more of those fields. As with the other two syndromes, bits corresponding to non-existent PEs are '0'.

6.4.1.4. Timeout Procedure

A timeout is set on a VID when a majority, but not a unanimity, of its members set their OBNE/IBNF bits. It is important to realize that majority conditions on these two SERP fields are the only conditions under which timeouts are set. The destination VID, exchange class, and user byte fields have no affect on timeouts.

The timeout algorithm is as follows (for the remainder of the paragraph, IBNF can be substituted for OBNE with no change in meaning). When a majority condition is seen on the OBNE bits, the scoreboard checks to see if a timeout has already been set on the VID. If no timeout is in progress, the current value of an internal free-running timer is read and

placed in a timeout storage area. If a timeout has been set, the stored value is subtracted from the current value of the timer. If the difference is less than or equal to the value stored in the VID's timeout field in the CT then the timeout has not yet expired, in which case the scoreboard clears the OBNE bit in the voted SERP memory (because the message cannot be sent yet). If the result is greater than the VID's timeout field, then the timeout has expired and the scoreboard does nothing to the voted OBNE bit. At this point, the VID can send the message if the other conditions are met (see the following sections for the explicit rules on valid messages).

The one exception occurs when the timeout field in the CT is zero. If this is the case, the VID will never time out. This case is useful for debugging purposes and in a situation where a VID should never time out.

Notice that IBNF timeouts are set on a VID regardless of whether another VID is waiting to send a message to that VID. This prevents the input buffers of an FMG from being overwritten. No messages will be sent to an FMG until a unanimity or a majority+timeout of its members assert their IBNF bits.

6.4.2. Finding Messages

Once the scoreboard has finished voting the SERP and checking timeouts, it searches the voted SERP for valid messages. The following sections cover message related functions.

6.4.2.1. Valid Exchange Classes

The exchange class byte of the SERP contains three sub-fields: the class, packet type, and mode. Figure 6-9 below shows their locations.

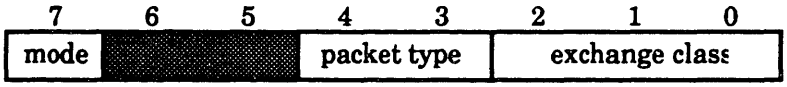


Figure 6-9, Exchange Class Byte Fields

The class defines the protocol to be used when exchanging the packet. Currently, the following values are valid (for an in-depth explanation of exchange classes see [Sak91]):

0-class 0 (no data)

1-class 1 (one round exchange)

2-class 2 (source congruency) from member on network element A

3-class 2 (source congruency) from member on network element B

4-class 2 (source congruency) from member on network element C

5-class 2 (source congruency) from member on network element D

6-class 2 (source congruency) from member on network element E

The packet type defines what the data in the packet represents. Data packets are the normal mode of communication between virtual groups. Data packets are treated as a contiguous stream of 64 bytes. There is no structure enforced by the NE on data packets. The other packet types, however, have specific formats that must be adhered to. The following are the current valid packet types:

0-data

1-configuration table update

2-isync

3-voted reset

The mode determines how the packet is to be distributed. Currently, the only modes supported are normal (bit 7 is cleared) and broadcast (bit 7 is set). In the normal mode, the packet is sent to the virtual group specified in the destination VID field. In broadcast mode, all active virtual groups will receive a copy of the packet (including the sender) and the destination VID field is ignored.

6.4.2.2. Invalid Messages

The following is a list of all the conditions under which an otherwise valid message can be declared invalid. In all cases, the message is "sent", but the destination is changed to the null destination (a PID of 0x1F). This PID value tells the NE to flush the packet after it has been received and not deliver it to any PE. The four cases boil down to two different conditions: invalid destination VID and/or invalid exchange class. A fourth, two-bit syndrome is generated by the scoreboard to represent the result of these two conditions.

- If the Higher Life Form (HLF) bit[21] is set, simplexes are not allowed to send CT update, broadcast, or voted reset packets.
- If the HLF bit is set, all CT update and voted reset packets must be exchanged using a class 1 exchange.
- If the exchange class is class 1 the source VID must be a FMG.
- The destination VID field must correspond to a valid VID[22].

6.4.2.3. Searching for Valid Messages

When the voting and timeout process is complete, the scoreboard scans the voted SERP for valid messages. The Boolean equation for a valid message is :

valid_message = source_VID(OBNE) **AND** destination_VID(IBNF)
AND message_info_valid;
OBNE = unanimous **OR** (majority **AND** timeout-expired)
IBNF = unanimous **OR** (majority **AND** timeout-expired)

A valid message exists when the sender either has a unanimous OBNE or a majority plus timeout-expired, the destination has either a unanimous IBNF or a majority plus timeout-expired, and all message information is valid. Once the condition is met but before the message is sent, the scoreboard clears the destination's IBNF bit since the act of sending a message to the VID may render it invalid. Thus, a given VID can send a maximum of one message and receive a maximum of one message per SERP cycle.

When a valid message is found, the scoreboard provides the NE with the following information:

---

21 The HLF bit exists to prevent, in a fielded FTPP, a simplex from doing things it is not normally allowed to do. In the laboratory, though, allowing simplexes to do such things is useful for debugging.

22 This condition must be flagged internally to prevent scoreboards residing on different NEs from reaching different results. For example, if the IBNF bit of the voted SERP memory location corresponding to the non-existent VID was '1' in one scoreboard but '0' in another (this could happen since that particular memory location is never written to by the scoreboard), they would reach different conclusions, thereby introducing a fault which would probably bring the system down.

1. OBNE,IBNF,and data syndromes
2. invalid data syndrome (explained in section 6.4.2.2)
3. Presence bits (used by the NE as a vote mask)
4. NE mask
5. the exchange class
6. the PID on the local NE where the message can be found
7. the PID on the local NE where the message is to be sent
8. A timestamp

Items 6 and 7 are determined from the source and destination VID's respective CT entries and by the local NE number. Using the triplex example from sections 6.1 and 6.3.1.3, if the triplex has a valid message to send to itself, then the scoreboards in the system would provide the following source and destination PIDs to their respective NEs :

Scoreboard's NE number	Source and Destination PIDs
0	NE0,PE4
1	0x3F
2	NE2,PE0
3	0x7F
4	NE4,PE6

When an NE does not have a VID member located on it, the scoreboard passes it a null PID of 0x1F (all 1's) so that the NE will participate in the exchange but will not deliver the message to any processor.

6.4.2.4. Message Sending Protocol

After a valid message condition exists and the scoreboard has gathered all the information necessary to send the message, it writes the data into a special area of memory and signals the NE to send the message. The scoreboard then continues to search for valid messages. If another message is found before the NE is finished sending the previous message, the scoreboard waits until the NE is finished. It then enqueues the new message and once again looks for more valid messages. When the entire voted SERP has been scanned, the scoreboard signals the NE that it is finished processing the SERP.

6.4.2.5. Broadcasts

Broadcasts are a special form of message which cause the scoreboard to process the voted SERP differently. As soon as a valid broadcast message is encountered (exchange class = broadcast **AND** voted OBNE bit set), the scoreboard ceases to search for any more messages until the broadcast is sent. It first cycles through the entire voted SERP to check if all the IBNF bits are set. If they are (which is unlikely), then the message is sent immediately. Otherwise, the scoreboard votes SERPs until all IBNF bits are set, after which the broadcast is sent. Once a broadcast is noted, no other packets are exchanged until the broadcast is exchanged.

The broadcast is very useful for bringing the entire system into synchronization because no other messages can be exchanged until the broadcast is sent. As a result, the sender is assured that all PEs in the system receive the broadcast at the same time.

6.4.2.6. Priority

The scoreboard has two different methods for determining message priority – VID-ordered and round-robin. A VID based system assigns priority to the lowest VID number. This is done by always beginning the scan for valid messages at the start of the voted SERP memory. The round-robin system attempts to distribute priority so that no VID is favored over any other. On each SERP cycle, the scoreboard begins looking for valid messages one voted SERP entry later than in the previous cycle. This system ensures, for instance, that a babbling simplex cannot effectively shut out a VID by constantly sending messages to it.

6.5. Faulty Conditions

The scoreboard does not perform any fault diagnosis itself. It simply notes an anomaly and sets the appropriate syndrome bits. In the case of an invalid destination VID or an invalid exchange class, the scoreboard sets the corresponding invalid data syndrome bit. The following conditions indicate a fault has occurred:

- A VID times-out from either OBNE or IBNF, or both.
- The OBNE bit for a VID is unanimous, but one member doesn't agree on the destination, exchange class, or user byte (or any combination thereof).
- The destination VID does not correspond to a valid VID in the system.

- The exchange class field has an invalid value.

6.6. Other Operations

The scoreboard also performs the following initialization operations:

Synchronize Timer - the synchronize timer operation is used to bring the timers inside each scoreboard into synchronization. This ensures that all timestamps are congruent. This operation also causes the scoreboard to delete all pending timeouts.

CT Update - A CT update causes three actions to occur. First, it forces the scoreboard to regenerate all internal tables (explained in section 7.1) which are used to index SERP entries during SERP processing. Second, it deletes all timeouts, since a VID with a pending timeout may no longer exist after the update. Finally, a CT update causes the voted SERP processing section of the scoreboard to reset its internal priority pointer to the first voted SERP entry (if VID-ordered priority is implemented).

Reset - A scoreboard reset is performed when the NE is first powered-up and whenever the NE itself is reset. This operation performs a CT update and initializes the timer to zero.

7. Scoreboard Behavioral Model

This chapter describes in detail the operation of the behavioral level model of the scoreboard. The first section discusses the overall design of the model by explaining important sections of the code, beginning with the packages and ending with the entities. The functional description of the model is presented by describing each major operation.

7.1. Overall Design

The scoreboard behavioral model represents the third step, behavioral models of the high-level partitions, in the design process presented in section 5.1. Chapter 6 contains the first step, system specification. The second step, high-level partitioning, is presented in the following paragraph.

Figure 7-1 below shows the high-level partitioning of the scoreboard. The partitioning is based on the algorithm discussed in section 6.4. The Voting and Timeout Hardware uses the Lookup Table to cycle through the SERP, reading a VID's SERP entry each cycle, feeding them to the voter and checking timeouts. The voted result is written into the Voted SERP memory. When voting is complete, the Sender, using the VIDS-in-system table, cycles through the Voted SERP memory looking for valid messages. When a valid message is found, the message information is written into the Message Info RAM and the NE is told to send the message. The structure of the tables and how they are generated will be presented in later sections.

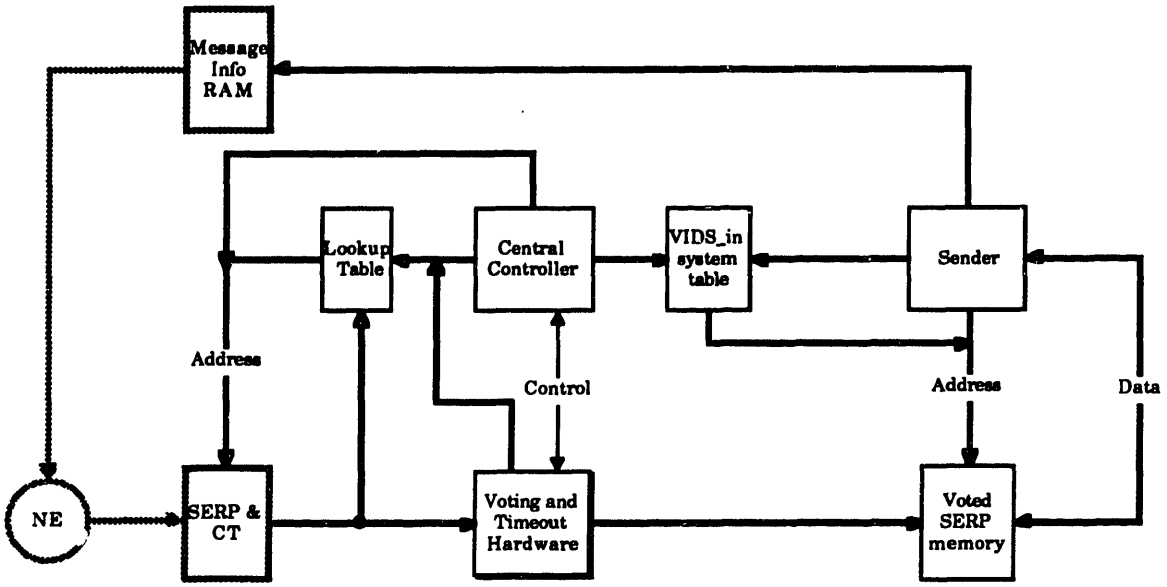


Figure 7-1, High-level Partitions of the Scoreboard

Once the partitioning had been accomplished, the VHDL models for each partition were written. The remainder of this chapter covers these models.

7.2. Explanation of Important Sections of Code

The following sections discuss the major sections of the behavioral VHDL model. The packages are covered first since they abstract away much of the detail and are critical for understanding the model. The function and interface of each entity is then described.

7.2.1. Packages

The behavioral model relies heavily on packages to allow multiple entities to share types, constants, and subprograms. The packages are organized by purpose and are usually associated with only a few entities.

7.2.1.1. Scoreboard package

The scoreboard_package is the global package and thus contains items common to the entire design. These items include the SERP, CT, and message item data types, configuration information, and two conversion functions.

The package begins with the following declarations which serve to abstract away the configuration of the FTPP :

```
CONSTANT num_ne : INTEGER := 5;
CONSTANT pe_per_ne : INTEGER := 8;
CONSTANT max_vid : INTEGER := 255;
CONSTANT max_redun_level : INTEGER := 4;
```

Because of these definitions, the overall configuration of the FTPP can be changed very easily. An additional set of constants is used to define the locations in the dual-port RAM of the SERP, CT, and message queue.

```
CONSTANT dpram_size : INTEGER := 300;
CONSTANT mem_base : address_type := -1;
CONSTANT serp_base : address_type := 0;
CONSTANT ct_base : address_type := dpram_size + 1;
CONSTANT msg_base : address_type := 2*dpram_size + 1;
```

The "meat" of the package is the type declarations. The initial declarations abstract away the types of the sub-fields of the SERP to aid visibility in the simulator. Flow_control_type is used to represent the OBNE and IBNF bits. Vid_type holds the destination VID number. Broadcast_type, packet_type, and ex_class_type represent the three fields within the exchange class SERP entry. Class_type represents the exchange class field of the SERP. Note the resemblance between its three fields and Figure 6-9.

```
SUBTYPE flow_control_type IS BOOLEAN;
SUBTYPE vid_type IS INTEGER RANGE 0 TO max_vid;
SUBTYPE broadcast_type IS BOOLEAN;
SUBTYPE packet_type IS INTEGER RANGE 0 TO 3;
SUBTYPE ex_class_type IS INTEGER RANGE 0 TO 7;

TYPE class_type IS RECORD
   broadcast : BOOLEAN;
   packet : packet_type;
   ex_class : ex_class_type;
END RECORD;
```

The second set of types is used to define CT entry fields. The redundancy level of a VID is represented by an enumerated type. A zero redundancy level indicates an inactive VID. Presence_type contains the presence bits field of a CT entry, while timeout_type contains the timeout field. Members_type holds the addresses of a VID's SERP entries.

```
TYPE redun_level_type IS (zero,simplex,triplex,quad);
TYPE presence_type IS ARRAY(0 TO (num_ne - 1)) OF BOOLEAN;
SUBTYPE timeout_type IS INTEGER RANGE 0 TO 255;
TYPE members_type IS ARRAY(0 TO (max_redun_level - 1)) OF pe_loc_type;
```

The last three types are used to represent an entry in the SERP, CT, or message queue. They are all records to facilitate encapsulation and clarity. Notice the direct correspondence

between serp_type and Figure 6-1 and between ct_type and Figure 6-2. Msg_type contains all the information necessary to send a message.

```
TYPE serp_type IS RECORD
   obne,ibnf : flow_control_type;
   dest_vid : vid_type;
   class : class_type;
END RECORD;

TYPE ct_type IS RECORD
   vid_number : vid_type;
   redun_level : redun_level_type;
   presence : presence_type;
   members : members_type;
   timeout : timeout_type;
END RECORD;

TYPE msg_type IS RECORD
   source_vid,dest_vid : vid_type;
   class : class_type;
   timestamp : TIME;
   obne_syndrome,ibnf_syndrome,vote_syndrome : presence_type;
   size : NATURAL;
END RECORD;
```

The final constant declarations are used to assign default values to the previously defined composite types. This is syntactically necessary if a port of mode in needs to remain open.

```
CONSTANT def_class : class_type := (FALSE,0,0);
CONSTANT def_presence: presence_type := (FALSE,FALSE,FALSE,FALSE,FALSE);
CONSTANT def_members : members_type := (0,0,0,0);
CONSTANT def_serp : serp_type := (FALSE,FALSE,0,def_class);
CONSTANT def_ct : ct_type := (0,zero,def_presence,def_members,0);
CONSTANT def_msg : msg_type := (0,0,def_class,0 ns,def_presence,
            def_presence,def_presence,0);
```

7.2.1.2. Address Package

The address_package contains the items necessary to implement a resolved address type for use in memory addressing. The scoreboard model contains many small memories, some of which need to be addressed by more than one entity. This creates a need for a common resolved type for use in addressing. A logic type could not be used since this would make the behavioral model less clear. It was desired to use a subtype of integer for addressing yet retain the ability to have multiple drivers. Address_package is the result.

The key to resolving an integer type is to define a high impedance value and write a resolution function which ignores all drivers with this value, just like a logic resolution function ignores drivers with high-impedance values. If an address has more than one driver with non-high-impedance values, then the resolution function should return a high-

impedance address. Address_type is defined as ranging from -1 to integer'**right** and the high_z_address is defined as -1. An array of addresses is also declared for use as input to the resolution function.

```
SUBTYPE address_type IS INTEGER RANGE -1 TO INTEGER'RIGHT;

CONSTANT high_z_address : address_type := -1;

TYPE address_array IS ARRAY (NATURAL RANGE <>) OF address_type;
```

The resolution function takes an array of addresses as input and returns the one non-high_z_address value. If more than one driver is not equal to -1, an error condition is asserted. Finally, the resolved address type is declared.

```
FUNCTION resolve_address (addresses: IN address_array)
RETURN address_type IS

   VARIABLE result : address_type;
   VARIABLE temp_i : INTEGER;
   VARIABLE found_one,more_than_one : BOOLEAN := FALSE;

BEGIN
   result := high_z_address;

   -- If no inputs then default to high_z_address
   IF (addresses'LENGTH = 0) THEN
      RETURN result;
   ELSIF (addresses'LENGTH = 1) THEN
      RETURN addresses(addresses'LOW);
   -- Calculate value based on inputs
   ELSE

   -- Iterate through all inputs
      FOR i IN addresses'RANGE LOOP
         IF ( addresses(i) = high_z_address  ) THEN
            NEXT;
         ELSIF NOT found_one THEN
            result := addresses(i);
            found_one := TRUE;
         ELSE
            more_than_one := TRUE;
         END IF;
      END LOOP;
      IF more_than_one THEN
         result := high_z_address;
         ASSERT FALSE
         REPORT "Address line has more than one driver"
         SEVERITY ERROR;
         END IF;

      -- Return the resultant value
      RETURN result;
   END IF;
END;

SUBTYPE resolved_address IS resolve_address address_type;
```

7.2.1.3. Voter Package

The voter package contains subprograms for converting the types used in the SERP to and from bits and bit_vectors and subprograms for performing rudimentary voting. It is important to note that this initial model does not perform majority voting; instead, it merely chooses the last parameter passed to it. This is not a major flaw since the current test vector generator does not generate faults. Instead, all SERP entries within a VID are the same. A structural voter currently being designed performs bitwise majority voting.

This package also defines types for use in the internal timer. The constant timer_resolution specifies the number of bits of resolution in the internal timer. Subtype timer_range is used to constrain the possible values of the timer. Init_timer_value is the value the timer assumes immediately after rollover, while max_timer_value is the rollover value. Timer_type represents a single entry in the internal timeout memory. Each entry has a flag to signal whether a timeout has been set and a variable to hold the value of the timeout. The declarations are given below.

```
CONSTANT timer_resolution : INTEGER := 16;
SUBTYPE timer_range IS INTEGER RANGE 0 TO (2**timer_resolution - 1);
CONSTANT init_timer_value : timer_range;
CONSTANT max_timer_value : timer_range;

TYPE timer_type IS RECORD
   timeout_set : BOOLEAN;
   value : timer_range;
END RECORD;

TYPE timeout_memory_type IS ARRAY(INTEGER RANGE <>) OF timer_type;
```

7.2.1.4. Other Packages

The behavioral model contains five more packages. The testbench package contains functions to read and write test vector files generated by a C program (see section 7.5.1 for further details). The file format is simple and can be gleaned from the C source code in Appendix 10.8.

The main control package contains two important enumerated type declarations. The first one, shown below, is used by the testbench to control the scoreboard. The unknown operation is included as an error check. A concurrent assertion statement warns the user if an unknown state is ever reached. The other operations will be explained in section 7.3.

```
TYPE operation_type IS (unknown,idle,reset_state,update_ct,
                        clear_timeouts,process_new_serp,continue);
```

The second declaration is used by the scoreboard to inform the testbench what operation it is performing and when it has completed a given operation.

```
TYPE return_operation_type IS (unknown,idle,busy,reset_complete,
                        ct_update_complete,clear_complete,
                              message_to_send,processing_complete);
```

The voted SERP package encapsulates types used by the entities which deal with the voted SERP memory. Each voted SERP entry is a record with the fields shown. The vid_is_simplex flag is used to flag illegal simplex messages since the redundancy level is not included in a voted SERP entry. Illegal message checking is performed by a subprogram also located in the voted serp package.

```
TYPE voted_serp_type IS RECORD
   obne,ibnf : flow_control_type;
   vid_is_simplex : BOOLEAN;
   source_vid,dest_vid : vid_type;
   class : class_type;
   obne_syndrome,ibnf_syndrome,sb_vote_syndrome : presence_type;
END RECORD;
```

The PID to VID package contains types used by the two internal translation tables. The first table allows the scoreboard to read SERP entries in VID order (they are in PID order inside the dual port RAM) and the second table allows the sender to cycle through the voted SERP memory efficiently (see section 7.2.2.1 for a more in depth explanation). Each PID to VID table entry is essentially the same as a CT entry. In an actual scoreboard, the members in the CT would be {NE,PE} encodings while each member in the PID-to-VID translation table would be the address of the PID's SERP entries.

```
TYPE pid_to_vid_entry_type IS RECORD
   vid : vid_type;
   redun_level : redun_level_type;
   presence : presence_type;
   members : members_type; -- these are really addresses
   timeout : timeout_type;
END RECORD;
```

The vids-in-system translation table does not require a composite type. Instead, each entry is merely an address into the voted SERP memory.

The final package in the model is the dual port ram package. It contains three array declarations, one each to hold the CT, the SERP, and the message queue.

7.2.2. Entities

This section provides an overview of the entities in the behavioral model. It gives the reader an insight into the structure of the scoreboard and how the entities are organized. Note that all the entities in the design are synchronous, meaning that all processes are sensitive to the clock and have the following basic form[23] :

```
example : PROCESS (clock)
BEGIN
   IF clock = f1 AND clock'EVENT THEN
   {
      body of process
   }
   END IF;
END PROCESS;
```

The advantages of a fully synchronous design were discussed in section 5.2. Additionally, all state machines within the design have the following basic form (this form is a simplification of the **CASE** variation of Figure 5-5):

```
state_machine : PROCESS (clock,activating_signal)
   TYPE state_type IS (s0,s1,s2);
BEGIN
   IF clock = f1 AND clock'EVENT THEN
      CASE state_signal IS
         WHEN s0 =>
            IF activating_signal = active THEN
                  state_signal <= s1;
            ELSE
            {
                  default assignments
            }
            END IF;

         WHEN s1 =>

         etc.
      END CASE;
   END IF;
END PROCESS;
```

The state machine remains in the initial state until the activating signal is brought to an active value. Otherwise, default assignments, which usually assign high-impedance values to shared signals, occur.

---

23 This chapter uses the Vantage 46 state logic system for all control-like signals. All that really needs to be known is that 'f1' and 'f0' are equivalent to '1' and '0', respectively.

Memory accesses within the model are assumed to take two clock cycles from the time the address is asserted to data valid. This results from the synchronous nature of the basic memory model, which is given below:

```
ENTITY example_memory IS
   GENERIC
      (
         read_delay: TIME := 10 ns
      );
   PORT
      (
         memory_output : OUT  memory_entry_type;
         memory_input: IN  memory_entry_type;
         read_write: IN  t_wlogic;
         address: IN  resolved_address;
         clock: IN  t_wlogic
      );
END example_memory;

ARCHITECTURE example_memory OF example_memory IS
   TYPE memory_type IS ARRAY(natural RANGE <>) OF memory_entry_type;
BEGIN
   memory_behavior : PROCESS (clock)
      VARIABLE memory : memory_type(mem_base TO mem_top);
   BEGIN
      IF clock = f1 AND clock'EVENT THEN
         IF read_write = f0 THEN
            memory(address) := memory_input;
         ELSE
            memory_output <= memory(address) AFTER read_delay;
         END IF;
      END IF;
   END PROCESS;
END example_memory;
```

All memories are built on the basic process model discussed previously. They generally have one port as an input into the memory and one port for the output (some memories are dual-ported). This has to be done to avoid writing resolution functions for inout ports[24]. The memory itself is simply an array whose index is the memory address. The generic read_delay is used to introduce an assignment delay. The purpose of this was discussed in section 5.3.

The entity declaration for the entire scoreboard is contained below. Operation_in is used by the NE to control the scoreboard, while operation_out is used by the scoreboard to tell the NE what it's doing. The higher life form (HLF) signal indicates whether a fault-

---

24 While many types can be resolved (like logic types), resolving a composite type doesn't make much sense. Since abstract behavioral models incorporate many such types, memories must have explicit **in** and **out** ports.

masking group is present in the system. Message_to_send tells the NE that a message is waiting to be sent. Sb_address and read_write are used to extract CT and SERP entries and write message entries. The data for these entries appears on the signals ct_data, serp_data, and msg_data, respectively. The system-wide clock, generated by the testbench, appears on the clock signal and is distributed to all entities with clock signals.

```
ENTITY scoreboard IS
   PORT
      (
         operation_in: IN  operation_type;
         operation_out: OUT  return_operation_type;
         hlf: IN  BOOLEAN;
         message_to_send: OUT  BOOLEAN;
         sb_address: OUT  resolved_address
         read_write: OUT  t_wlogic;
         ct_data: IN  ct_type;
         serp_data: IN  serp_type;
         msg_data: OUT  msg_type;
         clock: IN  t_wlogic;
      );
END scoreboard;
```

7.2.2.1. Dual Port RAM

The dual port RAM entity holds the CT and SERP. Message information is written into it by the sender. The dual port RAM and the scoreboard represent the top level architecture, which the testbench instantiates and tests.

Below is the entity declaration for the dual port RAM. Address0, RW0 (read/write), Act_in, Aserp_in, and Amsg_out control the NE side of the RAM. Address1, RW1, Bct_out, Bserp_out, and Bmsg_in control the scoreboard side of the RAM. The modes of the data ports represent the needs of the system. In other words, the unused ports, such as an Amsg_in, have been removed.

```
ENTITY dpram IS
   GENERIC
      (
         read_delay: TIME := 10 ns
      );
   PORT
      (
         address0: IN  address_type;
         RW0: IN  t_wlogic
         Act_in: IN  ct_type;
         Aserp_in: IN  serp_type;
         Amsg_out: OUT  msg_type;
         address1: IN  address_type;
         RW1: IN  t_wlogic;
         Bct_out: OUT  ct_type;
         Bserp_out: OUT  serp_type;
         Bmsg_in: IN  msg_type := def_msg;
         clock: IN  t_wlogic;
      );
```

```
END dpram;
```

7.2.2.2. Voted SERP

The voted SERP memory is organized as 256 [25] locations of one entry apiece. Any given VID's entry can be found by using its VID number as an address. Since even the largest system will only contain a maximum of 40 VIDs, the voted SERP memory will be sparsely populated. Memory is traded for speed in this case, since storing voted SERP entries in a packed format would require a table-lookup or a content-addressable memory.

The voted SERP memory entity declaration is shown below. It is a dual ported memory, except that port1 has no input port. Port0_in is used by the voting and timeout hardware to write voted SERP entries, while port1_out is used by the sender to read voted SERP entries. Port0_out is presently unused.

```
ENTITY voted_serp_memory IS
   GENERIC
      (
         read_delay: TIME := 10 ns
      );
   PORT
      (
         port1_rw: IN  t_wlogic := fl;
         port0_rw: IN  t_wlogic;
         clock: IN  t_.logic;
         port1_out: OUT  voted_serp_type;
         port1_address: IN  resolved_address;
         port0_address: IN  address_type;
         port0_out: OUT  voted_serp_type;
         port0_in: IN  voted_serp_type
      );

END voted_serp_memory;
```

7.2.2.3. Pid-to-vid Table

The scoreboard uses two internal tables to assist it in processing the SERP. The pid-to-vid table allows the scoreboard to read the SERP in VID order. This is important since that is how SERP entries must be voted. Each table entry contains the source VID, redundancy level, presence bits, and timeout value. The members array contains the dual port RAM addresses of each of the VID members SERP entries. The pid-to-vid table and the vids-in-system table (discussed in the next section) are regenerated when the scoreboard is reset and whenever a CT update is performed.

---

25 This number is dependent on the maximum VID number.

The pid-to-vid entity declaration is given below. It precisely follows the standard memory model.

```
ENTITY pid_to_vid IS
   GENERIC
      (
         read_delay: TIME := 10 ns
      );
   PORT
      (
         ptov_out: OUT  pid_to_vid_entry_type;
         ptov_in: IN  pid_to_vid_entry_type;
         read_write: IN  t_wlogic := f1;
         address: IN  resolved_address
         clock: IN  t_wlogic;
      );
END pid_to_vid;
```

7.2.2.4. Vids-in-system Table

The vids-in-system table allows the sender to cycle through the voted SERP memory efficiently by making a continuous traversal through the SERP. The vids-in-system table contains the addresses in the voted SERP memory of all the active VIDs in the system.

The entity declaration for the vids-in-system table is given below. It also precisely follows the standard memory model.

```
ENTITY vids_in_system IS
   GENERIC
      (
         read_delay: TIME := 10 ns
      );
   PORT
      (
         data_out: OUT  address_type;
         data_in: IN  address_type;
         read_write: IN  t_wlogic;
         address: IN  resolved_address;
         clock: IN  t_wlogic
      );
END vids_in_system;
```

7.2.2.5. Voting and Timeout Subsection

The voting and timeout subsystem performs the voting and timeout functions of SERP processing and writes the voted SERP entries into the voted SERP memory. It is organized as three processes. One process reads a VID's SERP entries, triggers the voter, and writes the voted result into the voted SERP memory. A second process performs the voting and timeout checking, and a third implements the scoreboard's internal timer. The first two

processes are state machines of the form discussed in section 7.2.2. The voting is done via a subprogram call. Section 7.2.1.3 discussed how the behavioral model performs voting.

The entity declaration for the voting and timeout hardware is shown below. Start_voting signals the voting and timeout hardware to begin voting the SERP. It asserts done_voting when SERP voting is complete. Num_vids is an integer which represents the number of vids in the system. The voting and timeout hardware uses this value to tell when all VIDS have been voted. Start_clear tells the voting and timeout hardware to start clearing timeouts. When timeouts are cleared, it signals clear_done. Ptov_address, ptov_rw (rw stands for read/write), and ptov_data are used to read entries from the pid-to-vid table. Dpram_address, dpram_rw, and serp_data are used to read SERP entries from the dual-port RAM. Voted_serp_address, voted_serp_rw, and voted_serp_data are used to write entries into the voted SERP memory. The clock is the system clock from the top-level entity.

```
ENTITY vote_timeout IS
   PORT
      (
         start_voting: IN  BOOLEAN;
         done_voting: OUT  BOOLEAN;
         num_vids: IN  INTEGER;
         start_clear: IN  BOOLEAN;
         clear_done: OUT  BOOLEAN;
         ptov_address: OUT  resolved_address := high_z_address;
         ptov_rw: OUT  t_wlogic;
         ptov_data: IN  pid_to_vid_entry_type;
         dpram_address: OUT  resolved_address := high_z_address;
         dpram_rw: OUT  t_wlogic;
         serp_data: IN  serp_type;
         voted_serp_address: OUT  address_type;
         voted_serp_rw: OUT  t_wlogic;
         voted_serp_data: OUT  voted_serp_type;
         clock: IN  t_wlogic
      );
END vote_timeout;
```

7.2.2.6. Sender

The sender entity cycles through the voted SERP memory using the vids-in-system table to check for valid messages. If the OBNE bit in a voted SERP entry is set, the potential message is checked for validity. If the message is valid, the sender reads the voted SERP entry corresponding to the destination VID. If the destination VID's IBNF bit is set, the message is enqueued. The sender has a priority pointer which is incremented after each SERP cycle so that it begins looking for valid messages one vids-in-system table entry later. In the initial model, the timestamp field of a message is generated from a signal internal to the sender instead of from the timer used for timeouts.

The entity declaration for the sender is shown below. Start_processing tells the sender to begin looking for valid messages. When the sender is completely finished processing the current SERP, it signals done. Message_to_send is asserted by the sender after it has found a valid message and written the message record into the dual-port RAM. The NE asserts continue after it has sent the message. The hlf signal affects valid messages as discussed in section 6.4.2.2. The ct_update signal tells the sender to reset its priority pointer to the beginning of the vids-in-system table. Num_vids is used by the sender to tell when all voted SERP entries have been checked for messages. Pass_through feeds a tri-state buffer which lets either the sender or the output from the vids-in-system table serve as the address into the voted SERP memory. The output from the vids-in-system table is used when OBNE bits are being checked, while the sender asserts the destination VID address after a set OBNE bit is encountered. Vis_address, vis_rw, and vis_data are used to read entries from the vids-in-system table. Voted_serp_address, vs_rw, and voted_serp_data are used to read voted SERP entries. Dpram_address, dpram_rw, and msg_data are used to write message records into the dual port RAM.

```
ENTITY sender IS
   PORT
      (
         start_processing: IN  BOOLEAN;
         done: OUT  BOOLEAN;
         message_to_send: OUT  BOOLEAN;
         continue: IN  BOOLEAN;
         hlf: IN  BOOLEAN;
         ct_update: IN  BOOLEAN;
         num_vids: IN  INTEGER;
         pass_through: OUT  BOOLEAN;
         vis_address: OUT  resolved_address;
         vis_rw: OUT  t_wlogic;
         vis_data: IN  address_type;
         voted_serp_address: OUT  address_type;
         vs_rw: OUT  t_wlogic;
         voted_serp_data: IN  voted_serp_type;
         dpram_address: OUT  resolved_address;
         dpram_rw: OUT  t_wlogic;
         msg_data: OUT  msg_type;
         clock: IN  t_wlogic
      );
END sender;
```

7.2.2.7. Main Controller

The main controller receives commands from the NE and asserts internal control signals to perform the correct actions in the proper order. The main controller is also responsible for informing the NE when requested actions have been completed. The valid commands were listed in section 7.2.1.4. The main controller also regenerates the pid-to-vid

and vids-in-system lookup tables during reset and CT update operations. The controller is composed of three process statements, one for processing commands, one to handle SERP processing, and one to generate the translation tables.

The grisly entity declaration for the main controller is shown below. Operation_in is used by the testbench to control the operation of the scoreboard, while operation_out is used by the scoreboard to inform the NE of what it is doing (see section 7.2.1.4 for the type declarations). Message_to_send is asserted by the sender when it has found a message. The main_controller asserts continue_processing after it receives a continue operation from the NE. Start_voting is used to start the voting and timeout hardware, which asserts done_voting when voting has been completed. Start_sender is then asserted to start the sender looking for messages. The sender asserts sender_done when it has completed message searching operations. The main controller asserts start_clear when it receives a clear_timeouts message. The voting and timeout hardware asserts clear_done when the clear has been completed. Ct_update is asserted by the main controller in response to a update_ct message. This signal tells the sender to reset its internal priority pointer. The num_vids signal tells the rest of the scoreboard how many active VIDs are in the CT. The remaining signals are used only when the vids-in-system and pid-to-vid tables need regeneration. Dpram_address, dpram_rw, and ct_data_in are used to read CT entries from the dual port RAM. Ptov_address, ptov_rw, and ptov_data are used to write pid-to-vid entries into the pid-to-vid table while vis_address,vis_rw, and vis_data are used to write vids-in-system entries into the vids-in-system table.

```
ENTITY main_controller IS
   PORT
      (
         operation_in: IN  operation_type;
         operation_out: OUT  return_operation_type;
         message_to_send: IN  BOOLEAN;
         continue_processing: OUT  BOOLEAN;
         start_voting: OUT  BOOLEAN;
         done_voting: IN  BOOLEAN;
         start_sender: OUT  BOOLEAN;
         sender_done: IN  BOOLEAN;
         start_clear: OUT  BOOLEAN;
         clear_done: IN  BOOLEAN;
         ct_update: OUT  BOOLEAN;
         num_vids: OUT  INTEGER;
         dpram_address: OUT  resolved_address := high_z_address;
         dpram_rw: OUT  t_wlogic
         ct_data_in: IN  ct_type;
         ptov_address: OUT  resolved_address := high_z_address;
         ptov_rw: OUT  t_wlogic;
         ptov_data: OUT  pid_to_vid_entry_type;
         vis_address: OUT  resolved_address;
         vis_data: OUT  address_type;
         vis_rw: OUT  t_wlogic;
```

```
        clock: IN  t_wlogic;
    );
END main_controller;
```

7.2.2.8. Address_buffer

The address buffer entity is essentially a tri-state buffer for tri-stating addresses. It is used by the sender to tri-state the output from the vids-in-system table so that it can assert the address into the voted SERP memory. It assigns the output to the input when pass_through is True and assigns high_z_address to the output when pass_through is False.

```
ENTITY address_buffer IS
   PORT
      (
         pass_through: IN  BOOLEAN;
         clock: IN  t_wlogic;
         output: OUT  resolved_address;
         input: IN  resolved_address
      );

END address_buffer;

ARCHITECTURE address_buffer_behavior OF address_buffer IS
BEGIN
   output <= input WHEN pass_through ELSE
             high_z_address;
END address_buffer_behavior;
```

7.3. Functional Description

Following is the functional description of the behavioral model. The effects of each major operation are explained in sequence, beginning with reset and ending with process_new_SERP. The order roughly corresponds to the events surrounding power-up to processing of the first SERP.

7.3.1. Reset

Before the scoreboard can perform any other action it must be reset. The reset causes two separate actions to occur : update_CT and clear_timeouts. When both actions are completed, the main controller signals reset_complete. Until then, it signals busy.

7.3.2. Clear_Timeouts

The clear_timeouts operation deletes all pending timeouts. A process within the voting and timeout architecture cycles through the timeout memories, setting the timeout_set field of each timeout entry to FALSE. Note that this function is not implemented in the initial behavioral model.

7.3.3. Update_CT

The update_CT operation causes two separate initializations to occur. First, a process within the main controller cycles through the CT looking for valid VIDs. When a VID with a non-zero redundancy level is found, its CT entry is converted into a pid-to-vid table entry and the VID number is added to the vids-in-system table. In the behavioral model, this conversion means the CT entry is copied into the pid-to-vid table. In the structural models, the (NE,PE) pairs within the CT entry will be converted to addresses into the SERP memory. The second intitialization simply performs a clear_timeouts operation.

7.3.4. Process_new_SERP

When the main controller receives a process_new_serp operation, it activates a process which handles all the necessary actions for SERP processing. The first action of this process is to activate the voting and timeout hardware which then votes the SERP. When the entire SERP has been voted and the result stored in the voted SERP memory, the sender is signalled to begin scanning for valid messages. When a message is found, the sender signals the SERP-processing process, which handles the message sending protocol. The message queue is not implemented in the initial model. Instead, the scoreboard idles until the NE sends a continue operation. When the sender has scanned the entire voted SERP, it signals the controlling process which in turn informs the NE that processing is complete. The following paragraphs provide more detail.

When the voting and timeout hardware is signaled to do so, it reads SERP entries, votes them, checks timeouts if necessary, and writes the result into the voted SERP memory. First, the addresses in a pid-to-vid translation table entry are used to read out the corresponding SERP entries. When all the SERP entries have been read, they are passed to a process which handles the voting and timeout checking. The voting is done by a subprogram which converts the SERP entries to bits, votes them, and converts them back to their original types. After timeouts are checked, the voted SERP entry is assembled and written to the voted SERP memory using the source VID as the address.

Once voting is completed, the sender is activated. The sender asserts an address into the vids-in-system table, each entry of which is an address into the voted SERP memory. When the resulting voted SERP entry appears on the data lines, the sender latches it and checks the OBNE bit. If the OBNE bit is not set, the next address in the vids-in-system table is asserted. If the OBNE bit is set, the potential message is checked for validity according to the

rules presented in section 6.4.2.2. The behavioral model does not flag invalid messages. The sender then asserts the destination VID as an address into the voted SERP memory. If the destination VID's IBNF bit is not set, the next address in the vids-in-system table entry is asserted. If it is set, the sender assembles a message record and informs the SERP processing controller that a message needs to be sent. The sender repeats this cycle until all the voted SERP entries have been processed.

7.4. Performance

The model estimates the following performance figures using a 25 MHz (40 ns) clock:

Operation	Time (μs)
reset	31.6
CT update	31.6
process_new_SERP to first message	14.5
process SERP and send all messages*	19.7

* Half of the VIDs source a message

7.5. Verification and Testing

The informal verification of the model involved two basic steps. The first was to write a test vector[26] generator in C based on the functional description of the scoreboard. The second was to write a VHDL testbench to read in these vectors from an external file, apply them to the behavioral model, and check the resulting outputs for validity. The following two sections discuss the algorithm used by the C program to generate test vectors and the testbench which reads and applies them.

7.5.1. C Program

Test vector generation begins by generating a CT. This is done by randomly generating a redundancy level (either 1,3 or 4) and attempting to fill it by cycling through the NE's and assigning a free PE from each until the VID is filled. If not enough free PEs are available to fill a VID, a new redundancy level is generated and the process repeats. Successful population of a VID allows the program to enter it in the CT[27].

---

26 Scoreboard test vectors consist of a CT and a number of SERPs generated from that CT.

27 The presence bits are fabricated and a VID number and timeout value are randomly generated first.

After all PEs have been assigned to a VID, the program begins to randomly generate messages. A source VID is chosen from a pool of free source VIDs and a destination VID is randomly generated (the destination VID number must correspond to a valid VID). Once the source and destination are selected the SERP entries are produced. The IBNF bits of the destination VID's members and the OBNE bits of the source VID's members are asserted. The exchange class and destination VID fields are also written. In the current version of the program (version 2.2), the exchange class is fixed and no user byte is written. The program produces messages until all VIDs have been used as sources.

The program writes each CT and SERP to an external file. All output is printable ASCII and entirely numeric. This ensures that the VHDL testbench has no problems reading and interpreting the file using the TEXTIO package.

7.5.2. Testbench

The scoreboard testbench simply instantiates the top level entity, which in the initial model encompasses both the scoreboard and the dual-port RAM, and feeds the model CTs and SERPs. The testbench first reads a CT from the test vector file, writes it into the dual-port RAM, and then tells the scoreboard to reset. After the reset is complete, the testbench reads the first SERP from the file, writes it into the dual-port RAM, and tells the scoreboard to process it. Each message the scoreboard sends is acknowledged by the testbench but currently only manual checks on message correctness are performed. When the scoreboard signals that SERP processing is complete, the testbench reads the next SERP from the file, writes it into the dual-port RAM, and tells the scoreboard to process it. This cycle is repeated as often as desired. The testbench is also responsible for generating the system-wide clock.

7.6. Limitations

The purpose of this section is to make explicit all the deficiencies of the VHDL model of the scoreboard. Many of these deficiencies were designed to limit complexity or resulted from changes in the algorithm. There was insufficient time to solve them.

- The C program needs rewriting to implement faults. This is the most serious limitation, for without the ability to generate faults or turn on OBNE bits over multiple SERP cycles, for example, much of the scoreboard is untested (i.e. the voter and the timeout mechanism).

- The main controller is really gross and kludgy since it was written incrementally. Most of its code is unnecessary, a fact discovered only after it was written.
- Timeout expiration is calculated incorrectly.
- The message sending protocol presented in the algorithm is not implemented. Instead, the scoreboard waits after each message for clearance to continue.
- IBNF bits are not cleared after sending a message. Thus, a VID could receive more than one message in a cycle.
- Invalid destination VIDs are not flagged.
- The model does not support a load timer operation.
- Full message validity checking is not implemented.
- The structural voter has not been tested.
- The testbench needs to perform complete message checking.

8. Discussions on Implementation

This chapter discusses a number of scoreboard implementation possibilities, shown in tree form in Figure 8-1. The conclusion of this chapter is that, in order to ensure a working implementation which meets throughput goals, an ASIC must be built, preferably using VHDL synthesis. If for some reason (such as cost) an ASIC cannot be constructed, the next best implementation would be to use a CPU. The sections following explain the advantages and disadvantages of each possibility in detail.

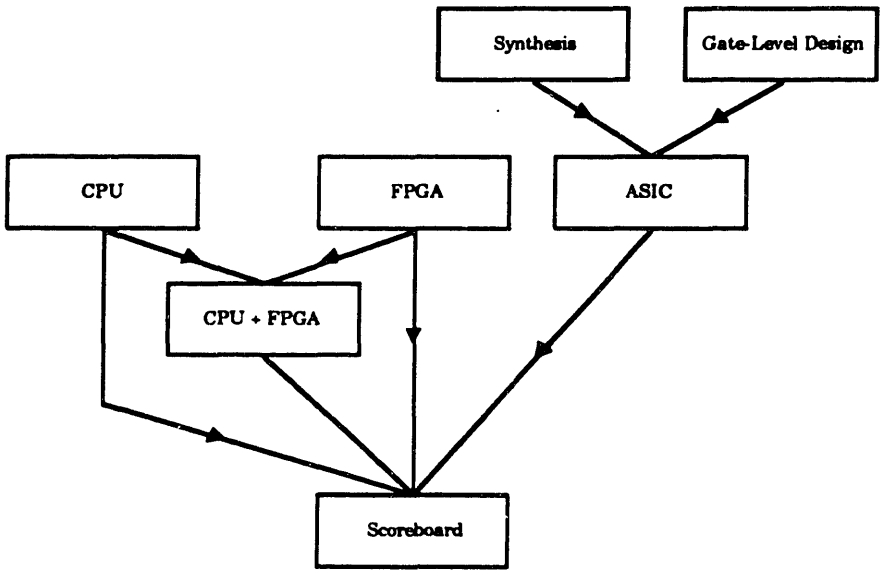


Figure 8-1, Implementation Tree

8.1. General Purpose Microprocessor

The simplest and most cost-efficient method for implementing the scoreboard is with a general purpose microprocessor. Since speed is the main goal, the preferred processor would be a RISC model, such as a SPARC or Motorola 88000. Figure 8-2 below shows the basic block diagram of such a design. It would consist of the processor, some dedicated memory, an external timer for timeouts[28], glue logic,and a dual-port RAM for communication between it and the rest of the NE. The dual-port RAM would hold the SERP, CT, and the

---

28 Though certain RISC processors (like the AMD 29000) have built-in timers, they could not be used because the timers in each scoreboard instance must be kept synchronized.

messages the scoreboard finds, while the private memory would hold any lookup tables used to speed SERP processing.

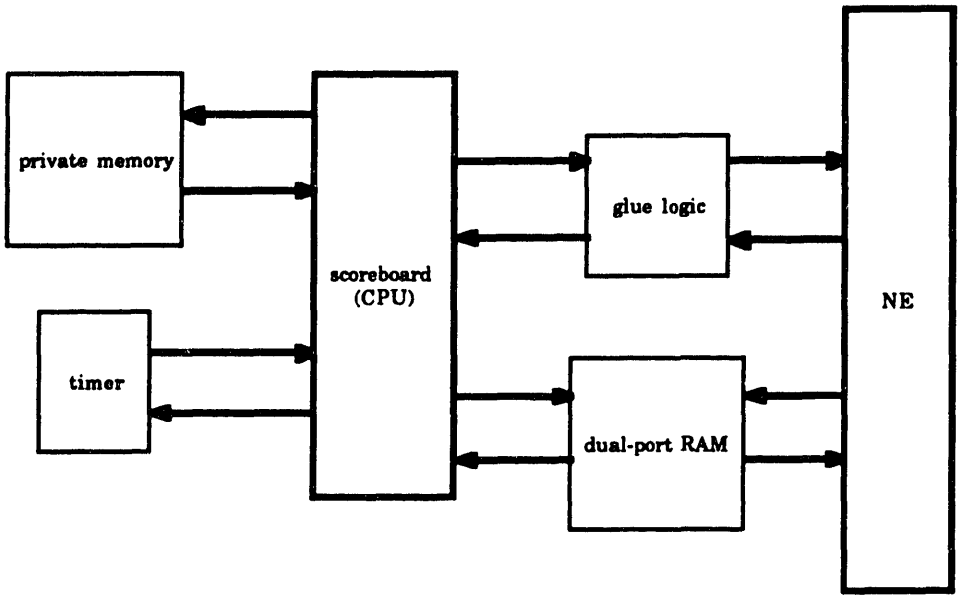


Figure 8-2, RISC scoreboard

The chief advantage to a RISC design is ease of design. Designing such a scoreboard would be simple since the only tricky part would be the glue logic, whose function it would be to interface to the NE's controller. The rest of the design is a simple matter of wiring pins together. The software design would be more complex, but still not too difficult since the scoreboard algorithm is easily expressed in C [29]. Example code for such a design can be found in Appendix 10.3. Because of its simplicity, a RISC scoreboard design is also easily changed.

The advantages of this design are compelling, so much so that it would be difficult to justify any other implementation save for two crippling disadvantages — performance and area. A feel for the performance can be obtained by examining some example scoreboard C code. The code which gets executed most often is the voting code, shown in Figure 8-3. Using fully optimized assembly language, 62 instructions are required to vote the OBNE bits of a triplex. Assuming an all triplex configuration (13 VIDS), 806 instructions would be executed to reach the conclusion that the SERP contains no messages. Using a 25 MHz processor (40

---

29 For a deliverable system, hand optimized assembly language would yield the best performance.

ns/instruction), this minimum case will require 32.2 μs to complete. When the overhead of performing timeouts and voting the rest of the SERP information is added, the scoreboard will be too slow to support real-time tasks with iteration rates of 100 Hertz.

```
/*****************************************************************
*/
/* vote is a generic vote function which will vote up to 4
*/
/* items passed to it.
/*****************************************************************
*/

int vote (a,b,c,d,redun_level,is_flow_control,unan)
int a,b,c,d,is_flow_control,redun_level,*unan;
{
  int result;

  switch (redun_level)
  {
  case 4:
    *unan = ((a == b) && (b == c) && (c == d)) ? TRUE : FALSE;
    if (is_flow_control)
      result = (a&b&c) | (a&c&d) | (b&c&d) | (a&b&d);
    else
      result = (a&b) | (b&c) | (c&d) | (a&c) | (a&d) | (b&d);
    break;
  case 3:
    *unan = ((a == b) && (b == c)) ? TRUE : FALSE;
    result = (a & b) | (b & c) | (a & c);
    break;
  case 1:
    *unan = TRUE;
    result = a;
    break;
  others:
    break;
  }
  return(result);
}
/* end vote */
```

Figure 8-3, C Voting Code

The second disadvantage of a RISC scoreboard is area. RISC chips alone are very large (approximately 200 pins is typical). The addition of support chips would cause the design to consume a large percentage of available board area.Thus, even though a RISC scoreboard is attractive from a design standpoint, it is unable to meet the design goals of C3.

8.2. FPGA

A second alternative for implementing the scoreboard is with Field Programmable Gate Arrays (FPGA). FPGAs have the advantages of relatively high-density, low cost, and reprogrammability. Most of them also have good design systems. Furthermore, an FPGA

implementation would probably be able to meet performance goals. An FPGA implementation has three major disadvantages, though.

First, the design task would be long and complex. A student at CSDL recently completed two FPGA designs for his MS Thesis [Sak91], one of which was a voter. The voter alone consumed an entire FPGA and could barely run at 12.5 MHz (a 25 MHz scoreboard is the goal). The scoreboard must contain a voter along with an abundance of additional hardware. Partitioning the design into multiple FPGAs would be a nightmare.

A second disadvantage is that the existing VHDL scoreboard models would be useless for designing the FPGAs. Although some companies have promised VHDL support for their FPGA design systems, such a capability is not currently available. With all the effort put forth into VHDL modeling (and the concomitant advantages), it would be undesirable to throw it all away.

The final disadvantage is verification. With VHDL, verification would proceed concurrently with transformation of the design to the gate level. Each step would be verified to ensure that the new model is correct and that design goals are being met. With an FPGA implementation, however, verification of the design would be much more difficult because it would be spread over multiple FPGAs. Verifying each FPGA would also be difficult because it would only perform a subsection of the full algorithm[30].

8.3. Combination

Another implementation strategy is to combine the CPU and FPGA. An FPGA could perform the speed critical task of voting while the CPU could take care of everything else including feeding the voter. This method would yield a fast enough design. However, it is probable that the overhead of reading SERP entries, writing them to the voter, and reading the result would incur the same overhead as software voting, since loads and stores are usually multiple cycle instructions. A solution would be to add address generation hardware to the voter so that it could read SERP entries on its own. If this is done, then why use a CPU at all?

---

30 It could be possible to generate test-vectors for the FPGA with a VHDL model, but the VHDL model would have to reflect the organization and gate-structure of the FPGAs. This would entail two complete designs of the scoreboard, one in VHDL and one in FPGAs, thus making this solution prohibitive.

Why not throw on the additional hardware to perform the rest of the scoreboard function? In short, a full FPGA implementation would be preferable to this option.

8.4. ASIC

The final implementation strategy is to use an ASIC. An ASIC has the advantages of speed, size, and verifiability but the disadvantage of high cost and high risk relative to the other implementation strategies. Two different paths exist for creating an ASIC – VHDL synthesis and gate level design. As previous sections have shown, using VHDL with synthesis is the preferred path.

There is little question that the fastest implementation is an ASIC. A single chip would also consume the least area of all the choices. Verifiability would also be the smoothest since the VHDL testbench could be used for all the functional test vectors. Additionally, good synthesis systems automatically insert additional hardware to aid final testing (i.e. scan-path). The problem with the ASIC approach is cost and risk. However, an ASIC is the best option for optimizing scoreboard performance.

As a sidenote, no matter which implementation method is chosen an emulator can be used to allow development of the rest of the NE while the scoreboard is being designed. This emulator would consist of a C (or similar) program running on a single-board computer. The NE could be set up to temporarily write SERPs and CTs into memory on board the emulator. The emulator would then process them and write messages back to the NE. To the NE, the emulator would simply appear as a slow scoreboard.

9. Conclusions and Recommendations

This thesis has discussed the advantages of using VHDL to design digital hardware. It also discussed modeling issues and applied them to the specification and modeling of the FTPP scoreboard. Finally, implementation options were discussed.

The main conclusion of this thesis is that VHDL, combined with the top-down design methodology, is a viable and useful digital hardware design method. The use of VHDL shortens the design cycle by facilitating the specification and verification of designs early in their life. Furthermore, abstract behavioral modeling, though requiring the rewriting of entity declarations, has been shown to be useful when little is known about implementation. The discussion on implementation concluded that an ASIC scoreboard would yield the best cost/performance, followed by a RISC-based scoreboard.

A great deal of work must be accomplished before a working scoreboard can be constructed. The author took the first step by structuralizing the voting and timeout hardware. As of this writing, though, it had not been tested. The same process of structuralization must be performed for all the entities in the behavioral VHDL model. After this has been accomplished, a VHDL synthesis tool could be used to produce a gate-level netlist. The test vector generator also requires an extensive rewrite to accommodate fault generation.

One implementation issue that was not discussed and should be further researched is that of using content-addressable memory for the voted SERP. It has the capability to reduce the memory demands of the scoreboard, both by reducing the size of the voted SERP memory and by eliminating the need for the vids-in-system table.

There is no text material missing here.
Pages have been incorrectly numbered.

10. Appendices

10.1. Glossary of Terms

CHDL - Computer Hardware Description Language.

CSDL - Charles Stark Draper Laboratory

FCR - Fault Containment Region : A circuit incapable of propagating internal hardware faults past its borders. This is achieved (usually) through physical and electrical isolation.

FTPP - Fault Tolerant Parallel Processor : A prototype fault-tolerant computer constructed to achieve high performance and high reliability for critical computing applications. See Figure 1 for a diagram.

HLF - Higher Life Form bit. This bit, used internally by the scoreboard when processing messages, indicates that at least one triplex or quad exists in the system.

LERP - Local Exchange Request Pattern : A data structure generated by each NE which contains message data for each PE in that NE. Spec ifically, the LERP contains whether the PE has a message to send and to whom and if the PE is able to receive a packet.

LRM - Language Reference Manual. This refers to the standard IEEE document on the VHDL language.

NE - Network Element : The part of the FTPP responsible for sending and receiving packets on behalf of the PE's.

NEFTP - Network Element Fault Tolerant Processor. A minimum Byzantine Resilient computer system used to demonstrate the utility of high-speed, fiber-optic data links.

Packet - The 64 byte block of data exchanged by the NE. Each inter-PE message is packetized by the NE before it is sent.

PE - Processing Element : The part of the FTPP which performs the computations. Usually a single-board computer.

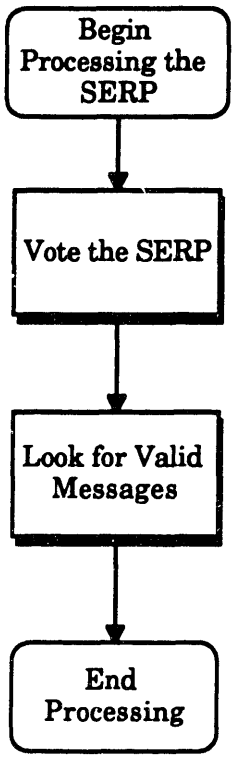
SERP - System Exchange Request Pattern : A data structure composed of the concatenation of the LERP from each FCR.

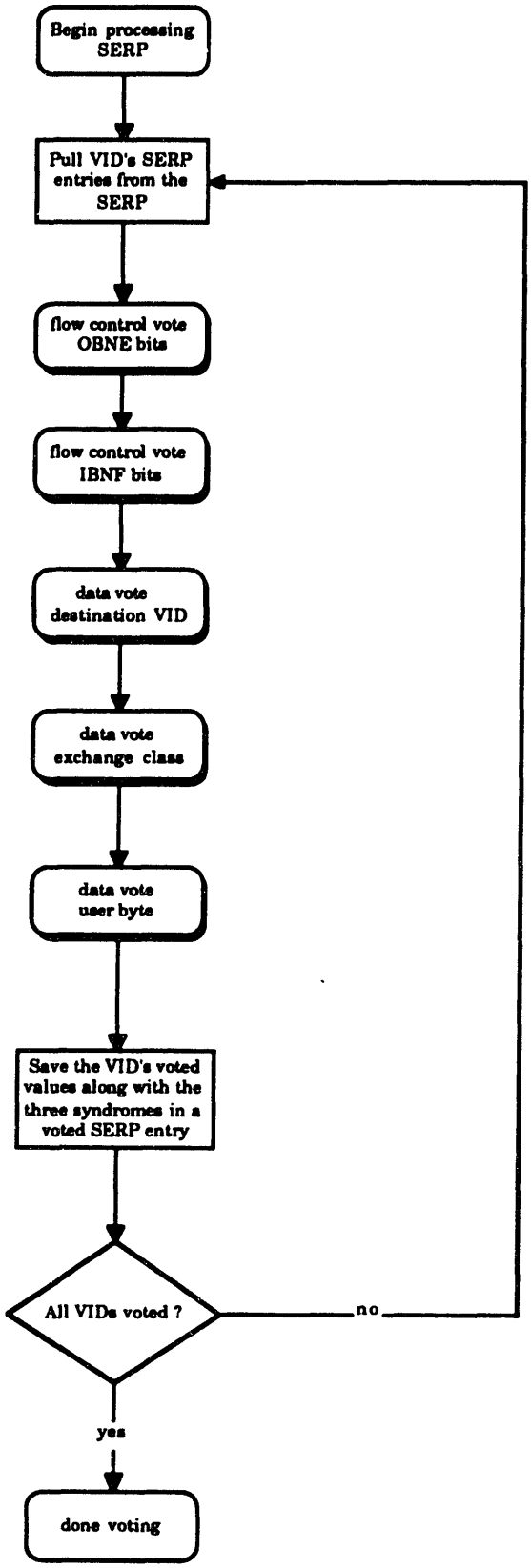
VHDL - VHSIC Hardware Description Language

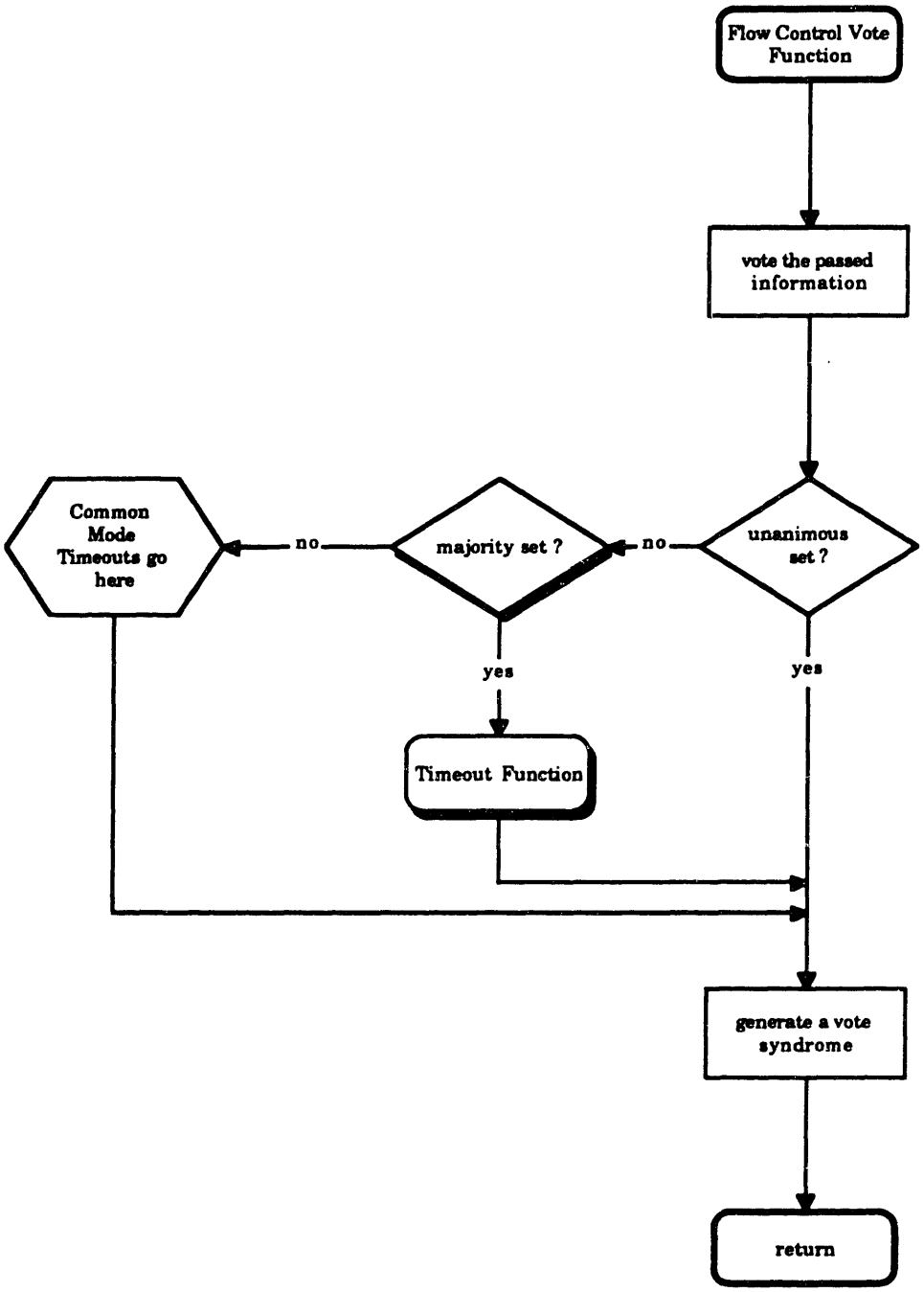
VHSIC - Very High Speed Integrated Circuit

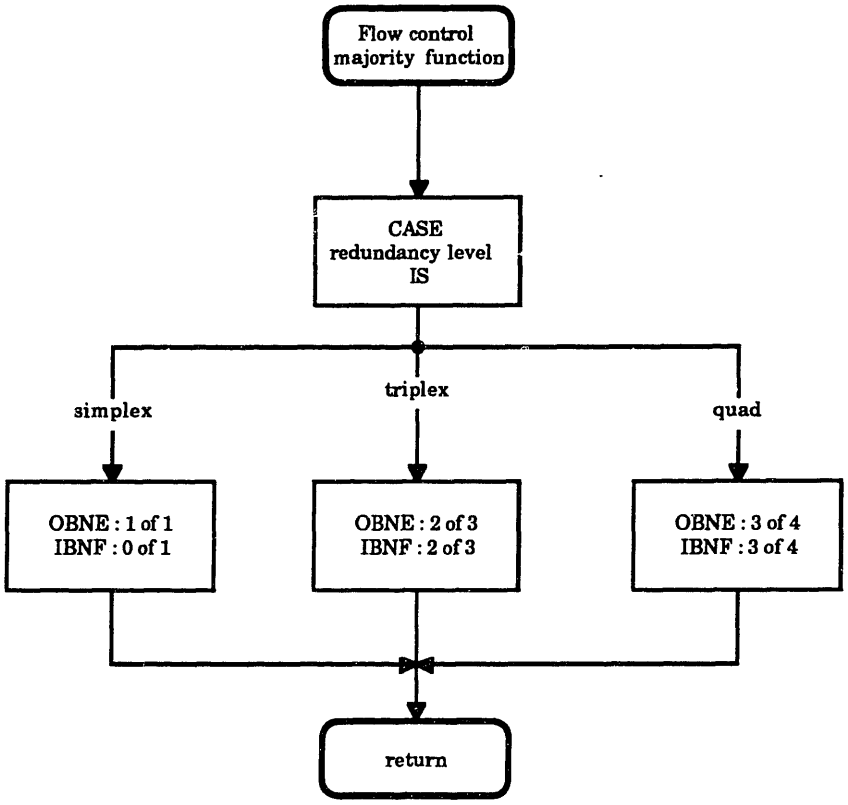
10.2. Scoreboard Algorithm

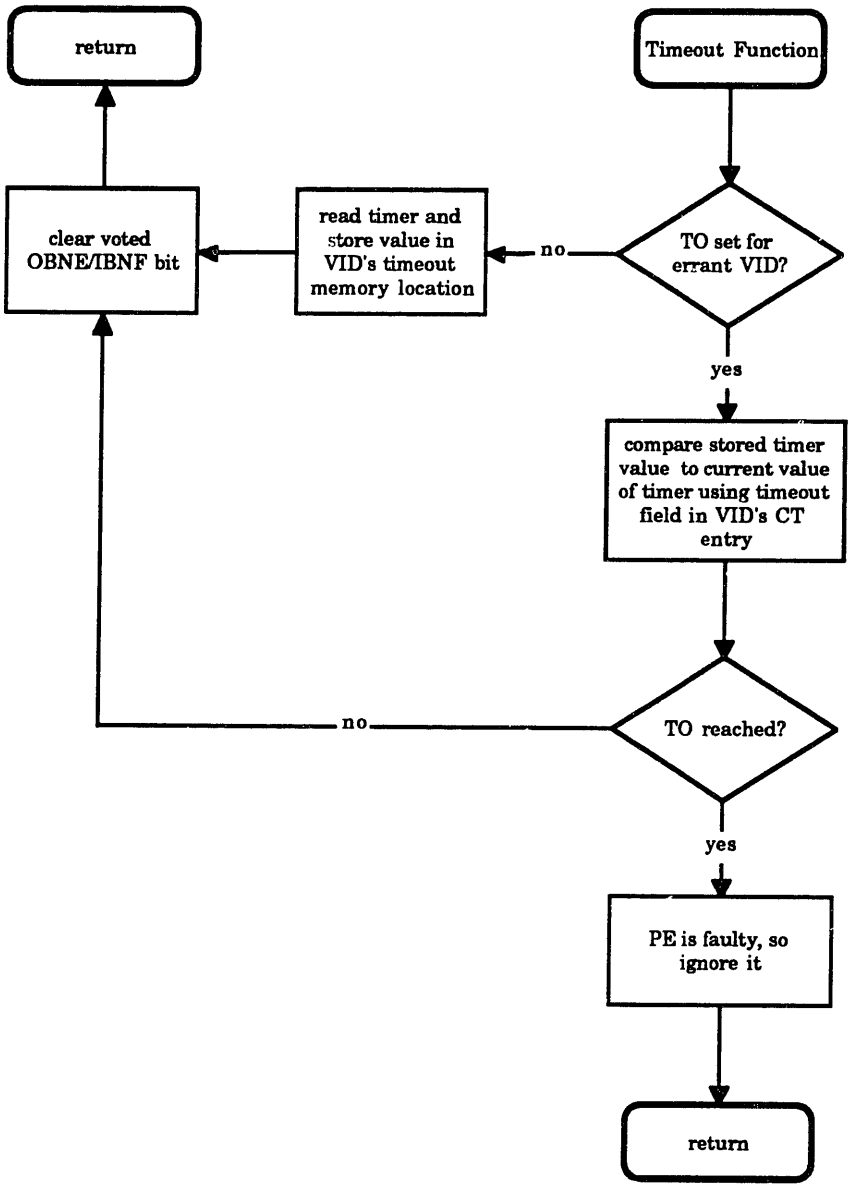
This appendix contains the scoreboard algorithm flowchart. Shadowed boxes refer to different pages.

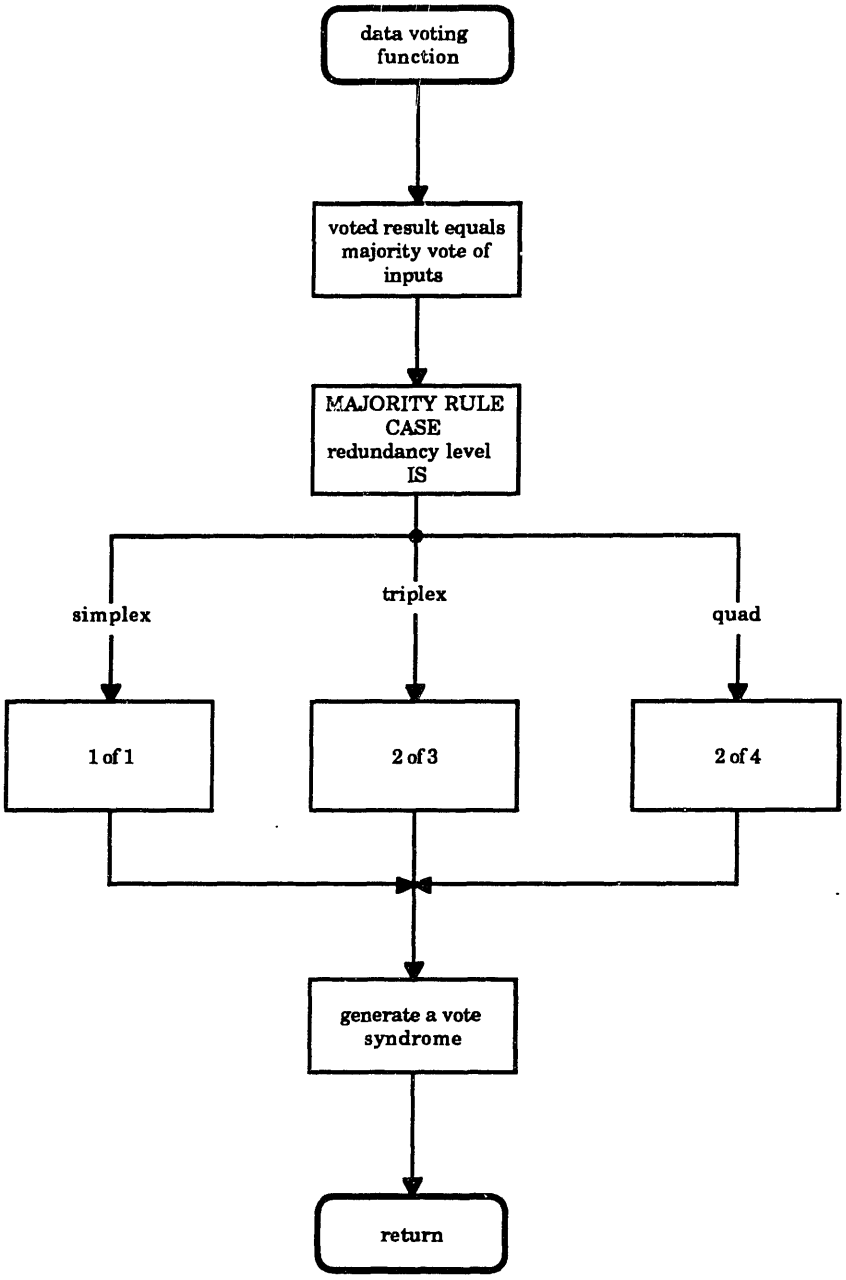


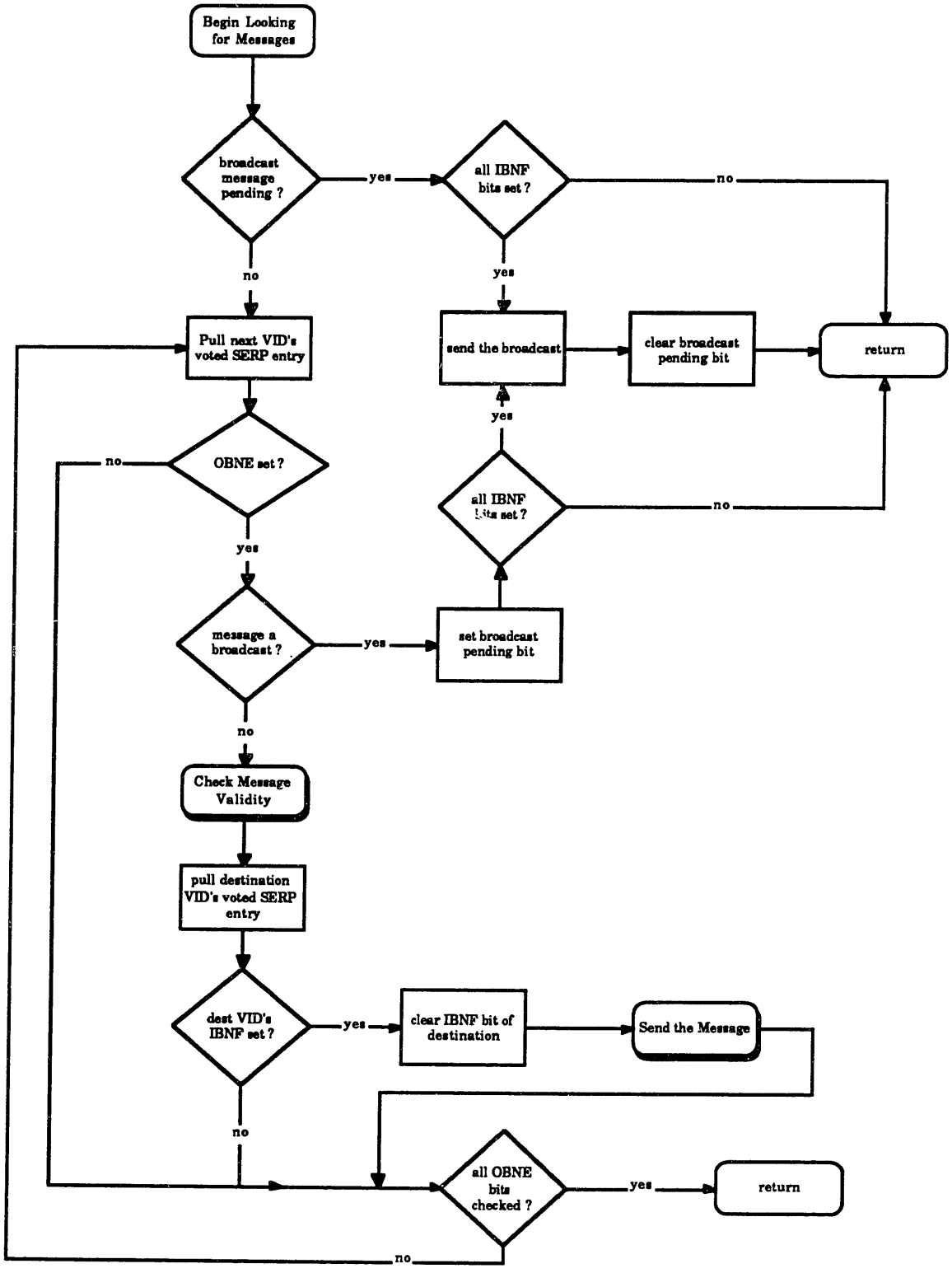

Begin processing SERP
Pull VID's SERP entries from the SERP
flow control vote OBNE bits
flow control vote IBNF bits
data vote destination VID
data vote exchange class
data vote user byte
Save the VID's voted values along with the three syndromes in a voted SERP entry
All VIDs voted ?
no
yes
done voting

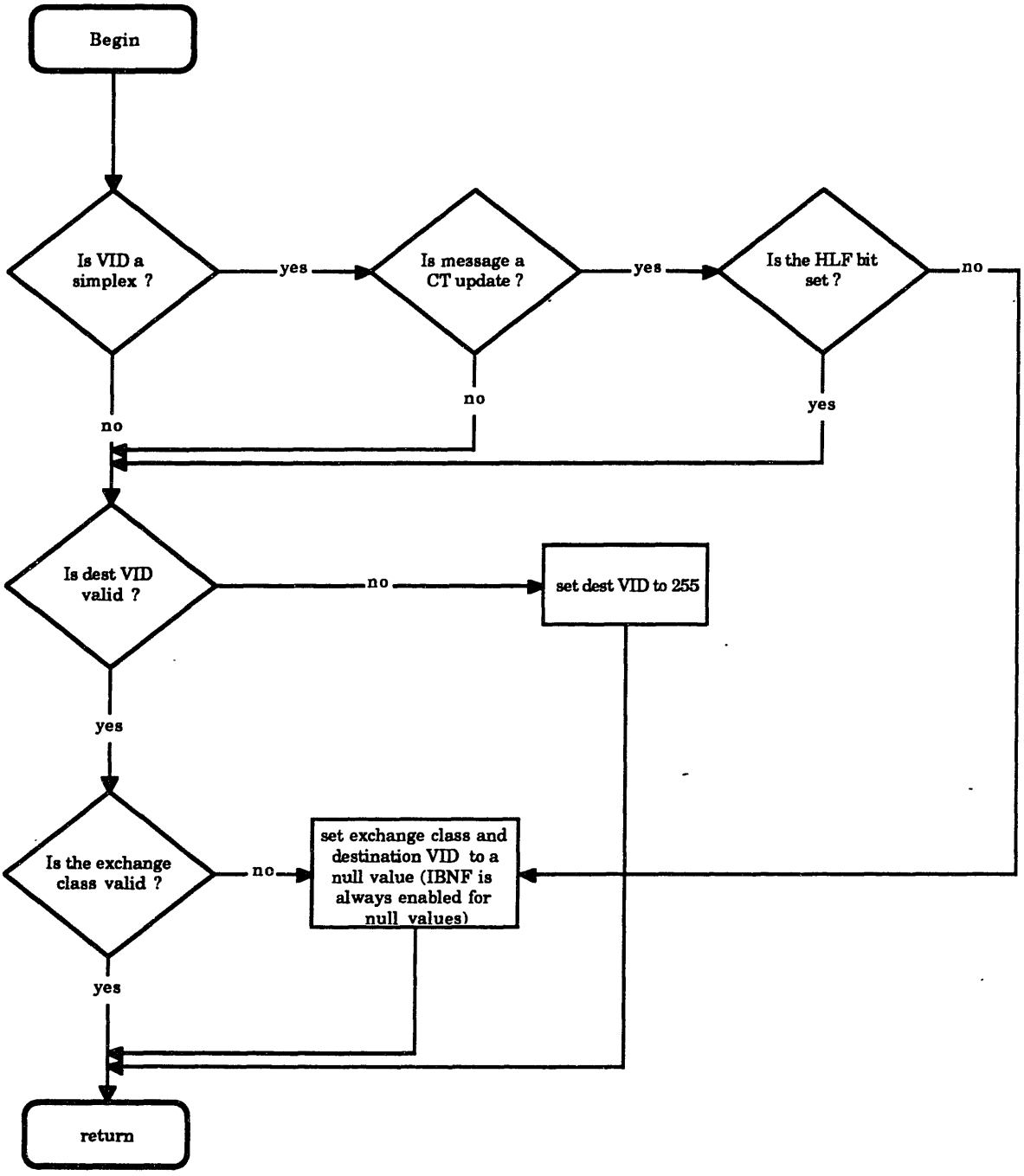

Flow Control Vote Function
vote the passed information
unanimous set ?
no
majority set ?
no
Common Mode Timeouts go here
yes
yes
Timeout Function
generate a vote syndrome
return

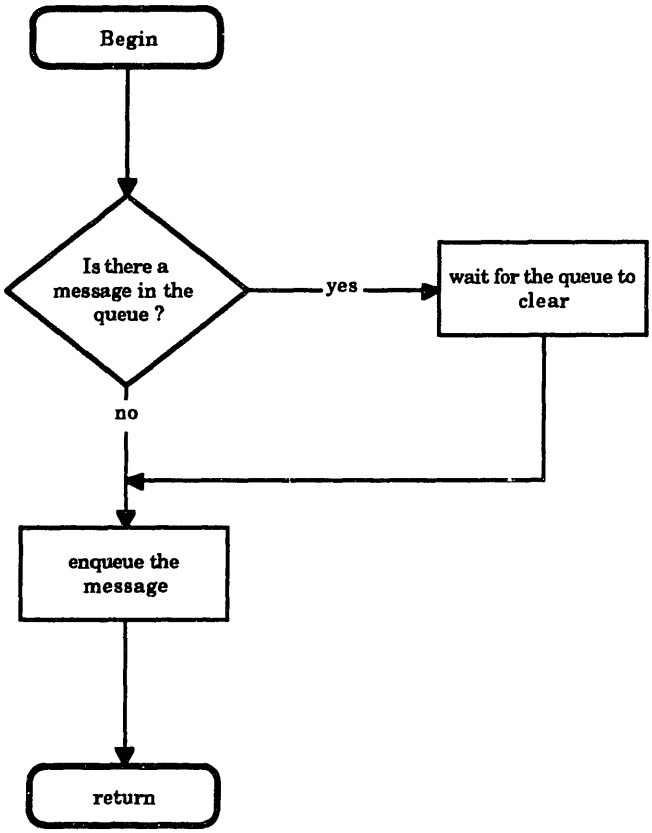

Flow control
majority function
CASE
redundancy level
IS
simplex
triplex
quad
OBNE : 1 of 1
IBNF : 0 of 1
OBNE : 2 of 3
IBNF : 2 of 3
OBNE : 3 of 4
IBNF : 3 of 4
return


return
Timeout Function
clear voted OBNE/IBNF bit
read timer and store value in VID's timeout memory location
no
TO set for errant VID?
yes
compare stored timer value to current value of timer using timeout field in VID's CT entry
no
TO reached?
yes
PE is faulty, so ignore it
return


data voting function
voted result equals majority vote of inputs
MAJORITY RULE CASE redundancy level IS
simplex
triplex
quad
1 of 1
2 of 3
2 of 4
generate a vote syndrome
return


Begin Looking for Messages
broadcast message pending ?
yes
all IBNF bits set ?
no
no
yes
Pull next VID's voted SERP entry
send the broadcast
clear broadcast pending bit
return
yes
OBNE set ?
no
all IBNF bits set ?
no
yes
message a broadcast ?
yes
set broadcast pending bit
no
Check Message Validity
pull destination VID's voted SERP entry
dest VID's IBNF set ?
yes
clear IBNF bit of destination
Send the Message
no
all OBNE bits checked ?
yes
return
no


Begin
Is VID a simplex ?
yes
Is message a CT update ?
yes
Is the HLF bit set ?
no
no
no
yes
Is dest VID valid ?
no
set dest VID to 255
yes
Is the exchange class valid ?
no
set exchange class and destination VID to a null value (IBNF is always enabled for null values)
yes
return


Begin
Is there a message in the queue ?
yes
wait for the queue to clear
no
enqueue the message
return

10.3. Sample Scoreboard Code

This appendix contains a C program which implements (most) of the scoreboard algorithm. It is intended as an example of code for a RISC scoreboard.

```
/* Scoreboard simulation program
   by
   Dennis Morton
   5 May 1991

   embedded scoreboard implementation
*/

/*********************************************************************/
/* This header contains globals used by the simulation               */
/*********************************************************************/

#define TRUE 1
#define FALSE 0

#define NUM_VIDS 256
#define MAX_VIDS_IN_SYSTEM
#define PE_PER_NE 8
#define NUM_NE 5
#define MAX_REDUN_LEVEL 4

#define OBNE_MASK 0x80000000
#define IBNF_MASK 0x40000000
#define DATA_MASK 0x00ffffff
#define DEST_VID_MASK 0x00ff0000
#define CLASS_MASK 0x0000ff00
#define BROADCAST_MASK 0x00008000
#define USER_BYTE_MASK 0x000000ff

#define UNAN_SYNDROME 0

typedef struct ct_entry_type
{
  int vid;
  int redun_level;
  int presence[NUM_NE];
  int timeout_value;
  int pids[MAX_REDUN_LEVEL];
};

typedef struct message_type
{
   int obne_syndrome,ibnf_syndrome,data_syndrome;
  int source_vid,dest_vid;
  int timestamp;
};

struct ct_entry_type ct[NUM_VIDS],ct_entry;
struct ct_entry_type translation_table[MAX_VIDS_IN_SYSTEM],translation_entry;
struct message_type message;

int serp[PE_PER_NE * NUM_NE];
int timeouts[NUM_VIDS];
int num_vids_in_system;

/*********************************************************************/
```

```
/*     vote is a generic vote function which will vote up to 4     */
/* items passed to it.                                             */
/*******************************************************************/

int vote (a,b,c,d,redun_level,is_flow_control,unan)
int a,b,c,d,is_flow_control,redun_level,*unan;
{
  int result;

  switch (redun_level)
  {
  case 4:
    *unan = ((a == b) && (b == c) && (c == d)) ? TRUE : FALSE;
    if (is_flow_control)
      result = (a&b&c) | (a&c&d) | (b&c&d) | (a&b&d);
    else
      result = (a&b) | (b&c) | (c&d) | (a&c) | (a&d) | (b&d);
    break;
  case 3:
    *unan = ((a == b) && (b == c)) ? TRUE : FALSE;
    result = (a & b) | (b & c) | (a & c);
    break;
  case 1:
    *unan = TRUE;
    result = a;
    break;
  others:
    break;
  }
  return(result);
}
/* end vote */

/*******************************************************************/
/*     check_to checks to see if the timeout value (to_value) has been */
/*  reached. If it has, then it returns a true value for to_reached.  */
/*******************************************************************/

int check_to (vid,to_value)
int vid,to_value;
{
  int to_reached = FALSE;
  int timer_value,timeout_value;

  timer_value = read_timer();
  timeout_value = timeouts[vid];
  if (timeout_value == 0)
    timeouts[vid] = timer_value;    /* TO set? then set a timeout */
  else if ((timer_value - timeout_value) > to_value)
  {
    to_reached = TRUE;
    timeouts[vid] = 0;
  }
  return (to_reached);
}
/* end check_to */

/*******************************************************************/
/* fc_vote performs the flow control vote function (i.e. OBNE      */
/* and IBNF).                                                      */
/*******************************************************************/

int fc_vote (vid,a,b,c,d,redun_level,to_value,syndrome)
int vid,a,b,c,d,redun_level,to_value,*syndrome;
```

```
{
   int i,unan,result;

   result = vote (a,b,c,d,redun_level,TRUE,&unan);

   if (unan)
      *syndrome = UNAN_SYNDROME;
   else
   {
      /* generate syndrome here */
   }

   if (!(unan) && (result != 0))    /* check for timeouts */
      if (!(check_to (vid,to_value)))
        /* timeout has not expired */
        result = FALSE;
   return (result);
}
/* end fc_vote */

/*****************************************************************/
/*     vote_other is the function which votes the destination    */
/* VID and exchange class fields of the SERP.                    */
/*****************************************************************/

int vote_data (a,b,c,d,redun_level,syndrome)
int a,b,c,d,redun_level,*syndrome;
{
   int result,unan;

   result = vote (a,b,c,d,redun_level,&unan);
   if (unan)
      *syndrome = UNAN_SYNDROME;
   else
   {
      /* generate syndrome here */
   }

   return (result);
}
/* end vote_other */

/************************ vote_serp ****************************/
/* vote_serp votes the SERP using the translation table to read */
/* entries out in VID order. It sends all messages it finds.    */
/*****************************************************************/

void vote_serp ()
{
   int obne_unan,ibnf_unan;
   int obne_syndrome,ibnf_syndrome,data_syndrome;
   int obne,ibnf,data,ex_class,dest_vid,user_byte;
   int a,b,c,d,i;
   static int broadcast_pending = FALSE;

   if (!(broadcast_pending))
     for (i = 0; ((i <= num_vids_in_system) && (!broadcast_pending)); i++)
     {
        translation_entry = translation_table[i];
        a = serp[translation_entry.pids[0]] & OBNE_MASK;
        b = serp[translation_entry.pids[1]] & OBNE_MASK;
        c = serp[translation_entry.pids[2]] & OBNE_MASK;
        d = serp[translation_entry.pids[3]] & OBNE_MASK;
        obne = fc_vote(translation_entry.vid,a,b,c,d,
```

```
                translation_entry.redun_level,
                translation_entry.timeout_value,&obne_syndrome);
      if (obne)
      {
       a = serp[translation_entry.pids[0]] & DATA_MASK;
       b = serp[translation_entry.pids[1]] & DATA_MASK;
       c = serp[translation_entry.pids[2]] & DATA_MASK;
       d = serp[translation_entry.pids[3]] & DATA_MASK;

       /* vote the exchange class, destination VID, and user byte */

       data = vote_data (translation_entry.vid,a,b,c,d,
                      translation_entry.redun_level,&data_syndrome);

       /* if message is a broadcast, processing is complete */
       if (data & BROADCAST_MASK)
         broadcast_pending = TRUE;
       else
       {
         dest_vid = data & DEST_VID_MASK;

         /* check ibnf bit of destination vid */
         ct_entry = ct[dest_vid];
         a = serp[ct_entry.pids[0]] & IBNF_MASK;
         b = serp[ct_entry.pids[1]] & IBNF_MASK;
         c = serp[ct_entry.pids[2]] & IBNF_MASK;
         d = serp[ct_entry.pids[3]] & IBNF_MASK;
         ibnf = fc_vote(ct_entry.vid,a,b,c,d,ct_entry.redun_level,
                      ct_entry.timeout_value,&ibnf_syndrome);
         if (ibnf)
         {
           /* send a message */
           message.obne_syndrome = obne_syndrome;
           message.ibnf_syndrome = ibnf_syndrome;
           message.data_syndrome = data_syndrome;
           message.source_vid = translation_entry.vid;
           message.dest_vid = ct_entry.vid;
           message.timestamp = 0xff;
         }
       }
      }
    }
  else
  {
       /* do broadcast stuff */
  }
}
```

10.4. Recommended Style Guide

I recommend adhering to the following style guide when modifying the scoreboard VHDL code in order to keep it uniform.

1. separate out the keywords by putting them in all capital letters.

2. use liberal indentation

3. follow this naming guide for constructs:

   - entities : descriptive name
   - architectures : entity_name_(behavioral,rtl,structural)
   - configuration : c(architecture_name)
   - packages : (descriptive name)_package
   - types : (descriptive_name)_type;

4. Model state machines using the method I describe in section 5.1.

10.5. Pitfalls to Avoid

The following is a list of pitfalls to avoid when using the Vantage Spreadsheet VHDL tool.

- once you change the grid, keep it consistent (I use a 5 point grid). Otherwise, signals will not connect to ports of entities created with a smaller grid if the port falls between grid points in the instantiating architecture.
- do not make port names visible to avoid unsightly clutter.
- when creating an entity, always draw the box larger then necessary since resizing it later is a pain.
- If a change is made to an component which is instantiated in an architecture, that component must be re-instantiated for the update to be reflected in the architecture. However, do **not** simply delete the old component, since any dangling signals will have to be redrawn. Instead, add a second component directly on top of the old component. Then, select them both and do an inform to find out the new components name (It'll be something like COMP 000025). The choose "unselect by name" and then delete. Do a screen update to see the new component. Be sure to rename it if the old component had a special name.

10.6. VHDL Behavioral Description

This appendix contains the complete VHDL source code for the behavioral model of the scoreboard. The files are in the same order as presented in section 7.2.

10.6.1. Scoreboard Package

```
--*************************************************************************
-- Scoreboard package declaration
--
-- This package contains data types and constants used throughout the
-- scoreboard entity. It is visible throughout the entire design
--------------------------------------------------------------------------

LIBRARY score;
USE  score.address_package.ALL;
USE  std.std_logic.ALL;
USE  std.std_ttl.ALL;

-- Note that deferred constants cannot be used very often in this
-- section because their values are needed later on in the package
-- declaration

PACKAGE scoreboard_package IS

-- define the clock_period

  CONSTANT clock_period : TIME;
  CONSTANT control_delay : TIME;

-- declare configuration type data (global in scope)

  CONSTANT num_ne : INTEGER := 5;
  CONSTANT pe_per_ne : INTEGER := 16;
  CONSTANT max_vid : INTEGER := 255;
  CONSTANT max_redun_level : INTEGER := 4;

  SUBTYPE pe_loc_type IS INTEGER RANGE 0 TO (pe_per_ne * num_ne - 1);

-- starting locations in the dual port ram
  CONSTANT dpram_size : INTEGER;
  CONSTANT mem_base : address_type;
  CONSTANT serp_base : address_type;
  CONSTANT ct_base : address_type;
  CONSTANT msg_base : address_type;

-- declare SERP related items
  SUBTYPE flow_control_type IS BOOLEAN;
  SUBTYPE vid_type IS INTEGER RANGE 0 TO max_vid;
  SUBTYPE broadcast_type IS BOOLEAN;
  SUBTYPE packet_type IS INTEGER RANGE 0 TO 3;
  SUBTYPE ex_class_type IS INTEGER RANGE 0 TO 7;

  TYPE class_type IS RECORD
    broadcast : BOOLEAN;
    packet : packet_type;
    ex_class : ex_class_type;
  END RECORD;

--
```

```
-- NOTICE that in this simulation no user byte is included. I'm still
-- debating whether to include it. The hooks will be there no matter
-- what, though.
--

  TYPE serp_type IS RECORD
    obne,ibnf : flow_control_type;
    dest_vid : vid_type;
    class : class_type;
  END RECORD;

-- declare configuration table related items

  TYPE redun_level_type IS (zero,simplex,triplex,quad);
  TYPE presence_type IS ARRAY(0 TO (num_ne - 1)) OF BOOLEAN;
  TYPE members_type IS ARRAY(0 TO (max_redun_level - 1)) OF pe_loc_type;
  SUBTYPE timeout_type IS INTEGER RANGE 0 TO 255;

  TYPE ct_type IS RECORD
    vid_number : vid_type;
    redun_level : redun_level_type;
    presence : presence_type;
    members : members_type;
    timeout : timeout_type;
  END RECORD;

-- the msg_data type is used to pass message data outside the scoreboard

  TYPE msg_type IS RECORD
    source_vid,dest_vid : vid_type;
    class : class_type;
    timestamp : TIME;
    obne_syndrome,ibnf_syndrome,vote_syndrome : presence_type;
    size : NATURAL;
  END RECORD;

--
-- define default constants for all the types in case an IN port of
-- these types wants to remain OPEN (won't work otherwise)
--

  CONSTANT def_class : class_type;
  CONSTANT def_presence : presence_type;
  CONSTANT def_members : members_type;
  CONSTANT def_serp : serp_type;
  CONSTANT def_ct : ct_type;
  CONSTANT def_msg : msg_type;

--************************************************************************
-- These next two functions are used to convert redun_level_type to and
-- from an INTEGER
--

  FUNCTION redun_to_int (redun : IN redun_level_type)
  RETURN INTEGER;

  FUNCTION int_to_redun (int : IN INTEGER)
  RETURN redun_level_type;

END scoreboard_package;


--************************************************************************
-- scoreboard_package body
```

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

PACKAGE BODY scoreboard_package IS

  CONSTANT clock_period : TIME := 40 ns;
  CONSTANT control_delay : TIME := clock_period/4;

--
-- starting locations in the dual port ram
--

  CONSTANT dpram_size : INTEGER := 300;
  CONSTANT mem_base : address_type := -1;
  CONSTANT serp_base : address_type := 0;
  CONSTANT ct_base : address_type := dpram_size + 1;
  CONSTANT msg_base : address_type := 2*dpram_size + 1;

--
-- Give values to the default constants
--

  CONSTANT def_class : class_type := (FALSE,0,0);
  CONSTANT def_presence : presence_type := (FALSE,FALSE,FALSE,FALSE,FALSE);
  CONSTANT def_members : members_type := (0,0,0,0);
  CONSTANT def_serp : serp_type := (FALSE,FALSE,0,def_class);
  CONSTANT def_ct : ct_type := (0,zero,def_presence,def_members,0);
  CONSTANT def_msg : msg_type := (0,0,def_class,0 ns,def_presence,
                def_presence,def_presence,0);

--************************************************************************
-- Elaborate the two conversion functions
--

  FUNCTION redun_to_int (redun : IN redun_level_type)
  RETURN INTEGER IS
  BEGIN
   CASE redun IS
     WHEN zero =>
       RETURN 0;

     WHEN simplex =>
       RETURN 1;

     WHEN triplex =>
       RETURN 3;

     WHEN quad =>
       RETURN 4;
   END CASE;
  END;

  FUNCTION int_to_redun (int : IN INTEGER)
  RETURN redun_level_type IS
  BEGIN
   CASE int IS
     WHEN 0 =>
       RETURN zero;

     WHEN 1 =>
       RETURN simplex;

     WHEN 3 =>
       RETURN triplex;
```

```
      WHEN 4 =>
        RETURN quad;

      WHEN OTHERS =>
        ASSERT FALSE REPORT "Integer Does Not Convert to redun";
        RETURN zero;
    END CASE;
  END;

END scoreboard_package;
```

10.6.2. Address Package

```
--*************************************************************************
-- Address Package Declaration
--
-- This package contains data types and a resolution function for
-- memory addresses. This package is included in all memory entities
-- and those which access them.
-------------------------------------------------------------------------

LIBRARY score;
USE std.std_logic.ALL;
USE std.std_ttl.ALL;

PACKAGE address_package IS

  SUBTYPE address_type IS INTEGER RANGE -1 TO INTEGER'RIGHT;

--*************************************************************************
-- Define a resolved address type. Somewhat kludgy, but it'll work.
--

  CONSTANT high_z_address : address_type;

  TYPE address_array IS ARRAY (NATURAL RANGE <>) OF address_type;

  FUNCTION resolve_address (addresses: IN address_array)
  RETURN address_type;

  SUBTYPE resolved_address IS resolve_address address_type;

--*************************************************************************

END address_package;


PACKAGE BODY address_package IS

--*************************************************************************
-- Address_type is resolved by checking for address_type'RIGHT. This
-- value is analogous to the 'Z' state of tri-state logic. In other
-- words, a value of dpram_size*3 does not have an effect
--

  CONSTANT high_z_address : address_type := -1;

  FUNCTION resolve_address (addresses: IN address_array)
  RETURN address_type IS

    VARIABLE result : address_type;
    VARIABLE temp_i : INTEGER;
    VARIABLE found_one,more_than_one : BOOLEAN := FALSE;

  BEGIN
    result := high_z_address;

    -- If no inputs then default to address'RIGHT
        IF (addresses'LENGTH = 0) THEN
            RETURN result;
        ELSIF (addresses'LENGTH = 1) THEN
```

```
            RETURN addresses(addresses'LOW);
          -- Calculate value based on inputs
          ELSE

              -- Iterate through all inputs
              FOR i IN addresses'LOW TO addresses'HIGH LOOP
                  IF ( addresses(i) = high_z_address  ) THEN
                    NEXT;
        ELSIF NOT found_one THEN
          result := addresses(i);
          found_one := TRUE;
        ELSE
          more_than_one := TRUE;
                  END IF;
              END LOOP;
         IF more_than_one THEN
           result := high_z_address;
--         ASSERT FALSE
--         REPORT "Address line has more than one driver"
--         SEVERITY ERROR;
         END IF;

             -- Return the resultant value
             RETURN result;
        END IF;
  END;

END address_package;
```

10.6.3. Voter Package

```
--*************************************************************************
-- Voter Package
--
-- This package contains subprograms to convert high level data types
-- to bit vectors so that they can be easily voted
-------------------------------------------------------------------------
LIBRARY score;
USE score.scoreboard_package.ALL;
USE score.voted_serp_package.ALL;
USE std.std_logic.ALL;
USE std.std_cmos.ALL;

PACKAGE voter_package IS

--*************************************************************************
-- Declare a type to hold an array of serp entries. This models the
-- registers at the input to the voter.
--
  TYPE serp_array IS ARRAY(NATURAL RANGE <>) OF serp_type;
-------------------------------------------------------------------------

--*************************************************************************
-- Declare timeout memory related stuff
--

  CONSTANT timer_resolution : INTEGER := 16;
  SUBTYPE timer_range IS INTEGER RANGE 0 TO (2**timer_resolution - 1);
  TYPE timer_type IS RECORD
    timeout_set : BOOLEAN;
    value : timer_range;
  END RECORD;

  TYPE timeout_memory_type IS ARRAY(INTEGER RANGE <>) OF timer_type;
  CONSTANT init_timer_value : timer_range;
  CONSTANT max_timer_value : timer_range;
-------------------------------------------------------------------------

--*************************************************************************
-- Declare the states for the voter controller
--
  TYPE vote_state_type IS (v0,v1,v2,v3,v4,v5,v6,v7,v8,v9,v10);
-------------------------------------------------------------------------

--*************************************************************************
-- Procedure vote_vid : this procedure takes in the SERP values for a
-- given vid and performs all the voting necessary to produce a
-- voted_serp entry.
-- NOTE : these procedures must be changed for a max_redun_level of
-- less than 4!!
-------------------------------------------------------------------------

  PROCEDURE vote_vid (    SIGNAL voted_serp_entry : INOUT voted_serp_type;
          SIGNAL vote_values : IN serp_array;
          SIGNAL current_vid : IN vid_type;
          SIGNAL presence : IN presence_type;
          obne_unan,ibnf_unan : INOUT BOOLEAN);

--*************************************************************************
-- Procedure vote_bits : this procedure simply votes a bit_vector and
```

```
-- returns both the result and a UNANIMOUS flag
-------------------------------------------------------------------------

  PROCEDURE vote_bits (   a,b,c,d : IN bit_vector;
          SIGNAL presence : IN presence_type;
          syndrome : OUT presence_type;
          unan : OUT BOOLEAN;
          result : INOUT bit_vector);

--************************************************************************
-- Procedure vote_bit : this procedure votes one bit (used for OBNE
-- and IBNF ) and returns a UNANIMOUS flag
-------------------------------------------------------------------------

  PROCEDURE vote_bit (    a,b,c,d : IN bit;
          SIGNAL presence : IN presence_type;
          syndrome : OUT presence_type;
          unan : OUT BOOLEAN;
          result : INOUT bit);

--************************************************************************
-- Below are the overloaded convert_to_bits procedures
--************************************************************************

  FUNCTION convert_to_bits ( a   : IN flow_control_type)
  RETURN bit;

  FUNCTION convert_to_bits ( a   : IN INTEGER)
  RETURN bit_vector;

  PROCEDURE convert_to_bits ( a,b,c,d : IN flow_control_type;
                ba,bb,bc,bd : OUT bit);

  PROCEDURE convert_to_bits ( a,b,c,d : IN INTEGER;
                ba,bb,bc,bd : OUT bit_vector);

  PROCEDURE convert_to_bits ( a,b,c,d : IN class_type;
                ba,bb,bc,bd : OUT bit_vector);

--************************************************************************
-- Below are the overloaded convert_back procedures which convert bits
-- back to abstract types
--************************************************************************

  FUNCTION convert_back ( a : IN BIT)
  RETURN BOOLEAN;

  PROCEDURE convert_back (       flow_control_bit : IN bit;
            SIGNAL flow_control : OUT flow_control_type);

  PROCEDURE convert_back (       bits : IN bit_vector;
            SIGNAL int : OUT INTEGER);

  PROCEDURE convert_back (       bits : IN bit_vector;
            SIGNAL class : OUT class_type);

  TYPE power_of_2_array IS ARRAY (NATURAL RANGE <>) OF NATURAL;
  CONSTANT power_of_2 : power_of_2_array(0 TO 7) := (1,2,4,8,16,32,64,128);

END voter_package;
```

```
--**************************************************************************
-- Voter Package Body
--
-- This is the body for the voter package
--------------------------------------------------------------------------

PACKAGE BODY voter_package IS

  CONSTANT init_timer_value : timer_range := 0;
  CONSTANT max_timer_value : timer_range := 2**timer_resolution - 1;

--**************************************************************************
-- PROCEDURE BODY vote_vid
--------------------------------------------------------------------------

  PROCEDURE vote_vid (    SIGNAL voted_serp_entry : INOUT voted_serp_type;
          SIGNAL vote_values : IN serp_array;
          SIGNAL current_vid : IN vid_type;
          SIGNAL presence : IN presence_type;
          obne_unan,ibnf_unan : INOUT BOOLEAN)
  IS
--
-- the b_ variables represent the SERP fields transformed into bits
--

    VARIABLE a,b,c,d : serp_type;
    VARIABLE ba,bb,bc,bd : bit; -- bit values of obne and ibnf
    VARIABLE voted_obne,voted_ibnf : bit;

    VARIABLE bva,bvb,bvc,bvd : bit_vector(7 DOWNTO 0);
    VARIABLE voted_dvid,voted_class : bit_vector(7 DOWNTO 0);

    VARIABLE dvid_syndrome,class_syndrome,obne_syndrome,ibnf_syndrome
                                : presence_type;
    VARIABLE dvid_unan,class_unan : BOOLEAN := FALSE;

    VARIABLE index : INTEGER := 0;

  BEGIN
    index := vote_values'LOW;
    a := vote_values(index);
    b := vote_values(index + 1);
    c := vote_values(index + 2);
    d := vote_values(index + 3);

--**************************************************************************
-- Vote obne

    convert_to_bits (a.obne,b.obne,c.obne,d.obne,ba,bc,bb,bd);
    vote_bit (ba,bb,bc,bd,presence,obne_syndrome,obne_unan,
            voted_obne);
    convert_back(voted_obne,voted_serp_entry.obne);

--**************************************************************************
-- Vote ibnf

    convert_to_bits (a.ibnf,b.ibnf,c.ibnf,d.ibnf,ba,bc,bb,bd);
    vote_bit (ba,bb,bc,bd,presence,ibnf_syndrome,ibnf_unan,
            voted_ibnf);
    convert_back(voted_ibnf,voted_serp_entry.ibnf);

--**************************************************************************
-- Vote destination VID
```

```
    convert_to_bits (a.dest_vid,b.dest_vid,c.dest_vid,d.dest_vid,
            bva,bvb,bvc,bvd);
    vote_bits (bva,bvb,bvc,bvd,presence,dvid_syndrome,dvid_unan,
             voted_dvid);
    convert_back(voted_dvid,voted_serp_entry.dest_vid);

--************************************************************************
-- Vote class

    convert_to_bits (a.class,b.class,c.class,d.class,
            bva,bvb,bvc,bvd);
    vote_bits (bva,bvb,bvc,bvd,presence,class_syndrome,class_unan,
             voted_class);
    convert_back(voted_class,voted_serp_entry.class);

    voted_serp_entry.source_vid <= current_vid;

  END;

--************************************************************************
-- PROCEDURE BODY vote_bits
--------------------------------------------------------------------------

  PROCEDURE vote_bits (   a,b,c,d : IN bit_vector;
          SIGNAL presence : IN PRESENCE_type;
          syndrome : OUT presence_type;
          unan : OUT BOOLEAN;
          result : INOUT bit_vector)
  IS

   VARIABLE ta,tb,tc,td : bit_vector(a'RANGE);

  BEGIN
    ta := a;
    tb := b;
    tc := c;
    td := d;

--
-- for now, the voter simply returns the last value which isn't masked out
-- this is ok because faults aren't being handled yet
--

    IF presence(0) THEN
      result := a;
    END IF;

    IF presence(1) THEN
      result := b;
    END IF;

    IF presence(2) THEN
      result := c;
    END IF;

    IF presence(3) THEN
      result := d;
    END IF;

-- result :=         (ta AND tb AND tc) OR (ta AND tc AND td) OR
--         (tb AND tc AND td) OR (ta AND tb AND td) OR
--         (ta AND tb) OR (tb AND tc) OR (tc AND td) OR
--         (ta AND tc) OR (ta AND td) OR (tb AND td) OR
--         ta OR tb OR tc OR td;
```

```
    unan := TRUE;

  END;

--************************************************************************
-- PROCEDURE BODY vote_bit
--------------------------------------------------------------------------

  PROCEDURE vote_bit (    a,b,c,d : IN bit;
          SIGNAL presence : IN presence_type;
          syndrome : OUT presence_type;
          unan : OUT BOOLEAN;
          result : INOUT bit)
  IS

    VARIABLE ta,tb,tc,td : bit;
  BEGIN
    ta := a;
    tb := b;
    tc := c;
    td := d;

    IF presence(0) THEN
      result := a;
    END IF;

    IF presence(1) THEN
      result := b;
    END IF;

    IF presence(2) THEN
      result := c;
    END IF;

    IF presence(3) THEN
      result := d;
    END IF;

    unan := TRUE;
  END;

--************************************************************************
-- Below are the PROCEDURE BODIES for the overloaded convert_to_bits
-- procedures
--------------------------------------------------------------------------

--************************************************************************
-- convert flow_control_type to bits
--------------------------------------------------------------------------

  FUNCTION convert_to_bits ( a   : IN flow_control_type)
  RETURN bit IS
  BEGIN

    IF a THEN
      RETURN '1';
    ELSE
      RETURN '0';
    END IF;
  END;


  PROCEDURE convert_to_bits ( a,b,c,d : IN flow_control_type;
```

```
                ba,bb,bc,bd : OUT bit)
  IS
  BEGIN

-- ASSERT (ba'LENGTH = bb'LENGTH = bc'LENGTH = bd'LENGTH)
-- REPORT "Yo!! Bit_vectors passed to convert_to_bits not the same length"
-- SEVERITY ERROR;


   IF a THEN
     ba := '1';
   ELSE
     ba := '0';
   END IF;

   IF b THEN
     bb := '1';
   ELSE
     bb := '0';
   END IF;

   IF c THEN
     bc := '1';
   ELSE
     bc := '0';
   END IF;

   IF d THEN
     bd := '1';
   ELSE
     bd := '0';
   END IF;
  END;

--*************************************************************************
-- convert subtypes of INTEGER to bits (limited to 8 bit resolution)
-------------------------------------------------------------------------

  FUNCTION convert_to_bits ( a   : IN INTEGER)
  RETURN bit_vector IS
   VARIABLE place,ta : INTEGER;
   VARIABLE temp : bit_vector(7 DOWNTO 0);
  BEGIN
   place := temp'RIGHT;
   ta := a;

   FOR i IN temp'RANGE LOOP
     IF (ta MOD 2) = 0 THEN
       temp(place) := '0';
     ELSE
       temp(place) := '1';
     END IF;

   ta := ta/2;
   place := place + 1;
   END LOOP;

   RETURN (temp);
  END;


  PROCEDURE convert_to_bits ( a,b,c,d : IN INTEGER;
                ba,bb,bc,bd : OUT bit_vector)
```

```
  IS

    VARIABLE place : INTEGER := 0;
    VARIABLE ta,tb,tc,td : INTEGER;

  BEGIN
--  ASSERT (ba'LENGTH = bb'LENGTH = bc'LENGTH = bd'LENGTH)
--  REPORT "Yo!! Bit_vectors passed to convert_to_bits not the same length"
--  SEVERITY ERROR;

    ta := a;
    tb := b;
    tc := c;
    td := d;

--
-- The choice of ba is arbitrary since all the bit_vectors must be the same size
--

    place := ba'RIGHT;
    FOR i IN ba'RANGE LOOP

      IF (ta MOD 2) = 0 THEN
        ba(place) := '0';
      ELSE
        ba(place) := '1';
      END IF;

      IF (tb MOD 2) = 0 THEN
        bb(place) := '0';
      ELSE
        bb(place) := '1';
      END IF;

      IF (tc MOD 2) = 0 THEN
        bc(place) := '0';
      ELSE
        bc(place) := '1';
      END IF;

      IF (td MOD 2) = 0 THEN
        bd(place) := '0';
      ELSE
        bd(place) := '1';
      END IF;

      ta := ta/2;
      tb := tb/2;
      tc := tc/2;
      td := td/2;
      place := place + 1;
    END LOOP;

  END;

--************************************************************************
-- convert class_type to bits
------------------------------------------------------------------------

  PROCEDURE convert_to_bits ( a,b,c,d : IN class_type;
                ba,bb,bc,bd : OUT bit_vector)
  IS

  BEGIN
```

```
-- ASSERT (ba'LENGTH = bb'LENGTH = bc'LENGTH = bd'LENGTH)
-- REPORT "Yo!! Bit_vectors passed to convert_to_bits not the same length"
-- SEVERITY ERROR;

    IF a.broadcast THEN
      ba(7) := '1';
    ELSE
      ba(7) := '0';
    END IF;

    IF b.broadcast THEN
      bb(7) := '1';
    ELSE
      bb(7) := '0';
    END IF;

    IF c.broadcast THEN
      bc(7) := '1';
    ELSE
      bc(7) := '0';
    END IF;

    IF d.broadcast THEN
      bd(7) := '1';
    ELSE
      bd(7) := '0';
    END IF;

    convert_to_bits(a.ex_class,b.ex_class,c.ex_class,d.ex_class,
           ba(2 DOWNTO 0),bb(2 DOWNTO 0),bc(2 DOWNTO 0),
           bd(2 DOWNTO 0));
    convert_to_bits(a.packet,b.packet,c.packet,d.packet,
           ba(4 DOWNTO 3),bb(4 DOWNTO 3),bc(4 DOWNTO 3),
           bd(4 DOWNTO 3));
--
-- don't care about the fifth and sixth bits, so don't bother assigning them
--

  END;

--***************************************************************************
-- Below are the PROCEDURE BODIES for the overloaded convert_back
-- procedures
-----------------------------------------------------------------------------

--***************************************************************************
-- convert bit to flow_control_type
-----------------------------------------------------------------------------

  FUNCTION convert_back ( a : IN BIT)
  RETURN BOOLEAN IS
  BEGIN
   IF a = '1' THEN
     RETURN(TRUE);
   ELSE
     RETURN(FALSE);
   END IF;
  END;


  PROCEDURE convert_back (       flow_control_bit : IN bit;
             SIGNAL flow_control : OUT flow_control_type)
  IS
```

```
  BEGIN
   IF flow_control_bit = '1' THEN
     flow_control <= TRUE;
   ELSE
     flow_control <= FALSE;
   END IF;
  END;

--************************************************************************
-- convert bit_vector to integer
--************************************************************************

  PROCEDURE convert_back (       bits : IN bit_vector;
             SIGNAL int : OUT  INTEGER)
  IS
   VARIABLE temp : INTEGER := 0;
   VARIABLE place : INTEGER := 0;
  BEGIN
   FOR i IN bits'REVERSE_RANGE LOOP
     IF bits(i) = '1' THEN
       temp := temp + power_of_2(place);
     END IF;
     place := place + 1;
   END LOOP;
   int <= temp;
  END;

--************************************************************************
-- convert a bit_vector to a class
--************************************************************************

  PROCEDURE convert_back (       bits : IN bit_vector;
             SIGNAL class : OUT  class_type)
  IS
  BEGIN
   IF bits(7) = '1' THEN
     class.broadcast <= TRUE;
   ELSE
     class.broadcast <= FALSE;
   END IF;

   convert_back(bits(2 DOWNTO 0),class.ex_class);
   convert_back(bits(4 DOWNTO 3),class.packet);

  END;


END voter_package;
```

10.6.4. Testbench Package

```
--*********************************************************************
-- This package contains subprograms and constants used by the
-- testbench. Its primary purpose is to abstract away the file reading
-- and writing from the testbench architecture
---------------------------------------------------------------------
LIBRARY score;
USE score.scoreboard_package.ALL;
USE std.textio.ALL;
USE std.std_logic.ALL;
USE std.std_cmos.ALL;

PACKAGE tb_package IS

--
-- These constants must be the same value as those in "config.h"
--
  CONSTANT bytes_per_CT_entry : INTEGER;
  CONSTANT bytes_per_SERP_entry : INTEGER;
  CONSTANT num_ne : INTEGER;
  CONSTANT pe_per_ne : INTEGER;

  TYPE int_array IS ARRAY (NATURAL RANGE <>) OF INTEGER;

--
-- read_serp_entry reads one SERP entry from an external file
-- the name of the file is contained in the following declaration,
-- which should be modified as needed.
--

  FILE test_data : TEXT IS IN "/usr/usr/ftpp/dennis/score/sbr2.2/test.i";

  PROCEDURE get_status (input_file : IN TEXT;
          regenerate_ct : OUT BOOLEAN;
          num_vids : OUT INTEGER;
          num_serp_entries : OUT INTEGER;
          num_messages : OUT INTEGER );

  PROCEDURE get_num_serp_entries (input_file : IN TEXT;
              num_entries : OUT INTEGER);

  PROCEDURE read_serp_entry (input_file : IN TEXT;
              serp_entry : OUT serp_type);

  PROCEDURE get_num_vids (input_file : IN TEXT;
            num_vids : OUT INTEGER);

  PROCEDURE regenerate_ct (input_file : IN TEXT;
             regenerate : OUT BOOLEAN);

  PROCEDURE read_ct_entry (input_file : IN TEXT;
             ct_entry : OUT ct_type);

  PROCEDURE get_msg_length (input_file : IN TEXT;
              msg_length : OUT INTEGER);

  PROCEDURE read_msg_entry (input_file : IN TEXT;
              msg_entry : OUT msg_type);

END tb_package;
```

```
--*****************************************************************************
-- Testbench Package Body
-------------------------------------------------------------------------------

PACKAGE BODY tb_package IS

--
-- These constants must be the same value as those in "config.h"
--

  CONSTANT bytes_per_CT_entry : INTEGER := 8;
  CONSTANT bytes_per_SERP_entry : INTEGER := 4;
  CONSTANT num_ne : INTEGER := 5;
  CONSTANT pe_per_ne : INTEGER := 8;

--*****************************************************************************
-- PROCEDURE get_status
--
-- This PROCEDURE reads the status line of the input file to determine
-- whether to perform a ct_update, and if so how many VID entries to
-- read. It also returns the number of SERP and message entries there
-- are before the next status line
-------------------------------------------------------------------------------

  PROCEDURE get_status (input_file : IN TEXT;
          regenerate_ct : OUT BOOLEAN;
          num_vids : OUT INTEGER;
          num_serp_entries : OUT INTEGER;
          num_messages : OUT INTEGER )
  IS

    VARIABLE l : line;
    VARIABLE good : BOOLEAN;
    VARIABLE temp : INTEGER;

  BEGIN
    readline(input_file,l);
    read(l,temp,good);
    IF temp = 0 THEN
      regenerate_ct := FALSE;
    ELSE
      regenerate_ct := TRUE;
    END IF;
    ASSERT good
    REPORT "Could not read number of SERP entries -- HALTING"
    SEVERITY FAILURE;

    read(l,num_vids,good);
    ASSERT good
    REPORT "Could not read number of SERP entries -- HALTING"
    SEVERITY FAILURE;

    read(l,num_serp_entries,good);
    ASSERT good
    REPORT "Could not read number of SERP entries -- HALTING"
    SEVERITY FAILURE;

    read(l,num_messages,good);
    ASSERT good
    REPORT "Could not read number of SERP entries -- HALTING"
    SEVERITY FAILURE;
  END;
```

```
--*************************************************************************
-- Get_num_serp_entries
--
-- This PROCEDURE determines how many serp entries should be read from
-- the input file
-------------------------------------------------------------------------

  PROCEDURE get_num_serp_entries (input_file : IN TEXT;
                num_entries : OUT INTEGER)
  IS
   VARIABLE l : LINE;
   VARIABLE good : BOOLEAN;
  BEGIN
   readline(input_file,l);
   read(l,num_entries,good);
   ASSERT good
   REPORT "Could not read number of SERP entries -- HALTING"
   SEVERITY FAILURE;

  END get_num_serp_entries;

--*************************************************************************
-- Read_serp_entry
--
-- This PROCEDURE reads the next serp entry from the input file.
-------------------------------------------------------------------------

  PROCEDURE read_serp_entry (input_file : IN TEXT;
              serp_entry : OUT serp_type)
  IS
   VARIABLE l : LINE;
   VARIABLE values : int_array(1 TO 6);
   VARIABLE good : BOOLEAN;
   VARIABLE temp : serp_type;
  BEGIN
   readline(input_file,l);

   -- extract out the various fields from the line just read
   FOR i IN values'RANGE LOOP
      read(l,values(i),good);
     ASSERT (good)
     REPORT "Problem with serp input file"
     SEVERITY failure;
   END LOOP;

   -- assign values to the record fields
   IF (values(1) = 0) THEN
     temp.obne := FALSE;
   ELSE
     temp.obne := TRUE;
   END IF;

   IF (values(2) = 0) THEN
     temp.ibnf := FALSE;
   ELSE
     temp.ibnf := TRUE;
   END IF;

   temp.dest_vid := values(3);

   IF (values(4) = 0) THEN
     temp.class.broadcast := FALSE;
   ELSE
```

```
      temp.class.broadcast := TRUE;
      ASSERT FALSE REPORT "Broadcast message has been read";
    END IF;

    temp.class.packet := values(5);
    temp.class.ex_class := values(6);

    serp_entry := temp;
  END read_serp_entry;

--****************************************************************************
-- Regenerate_ct
--
-- This PROCEDURE determines if a new ct must be read in prior to reading
-- another serp.
------------------------------------------------------------------------------

  PROCEDURE regenerate_ct (input_file : IN TEXT;
            regenerate : OUT BOOLEAN)
  IS
   VARIABLE l : LINE;
   VARIABLE temp : INTEGER;
  BEGIN
   readline(input_file,l);
   read(l,temp);
   IF temp = 1 THEN
     regenerate := TRUE;
   ELSE
     regenerate := FALSE;
   END IF;
  END;

--****************************************************************************
-- Get_num_vids
--
-- This PROCEDURE reads the first entry in the input file to determine
-- how many vids to read in
------------------------------------------------------------------------------

  PROCEDURE get_num_vids (input_file : IN TEXT;
           num_vids : OUT INTEGER)
  IS
   VARIABLE l : LINE;
   VARIABLE good : BOOLEAN;
  BEGIN
      readline(input_file,l);
   read(l,num_vids,good);
   ASSERT (good)
   REPORT "Problem with CT input file (bad number of vids)"
   SEVERITY failure;
  END;

--****************************************************************************
-- Read_ct_entry
--
-- This PROCEDURE reads the next ct entry from the input file
------------------------------------------------------------------------------

  PROCEDURE read_ct_entry (input_file : IN TEXT;
            ct_entry : OUT ct_type)
  IS
   VARIABLE l : LINE;
   VARIABLE values : int_array ( 1 TO (max_redun_level + 4));
   VARIABLE good : BOOLEAN;
```

```
    VARIABLE temp : ct_type;
    VARIABLE redun : INTEGER;
    VARIABLE temp_to : timeout_type;
  BEGIN
    -- the first part reads in the file entry for a vid
    readline(input_file,l);

    read(l,temp.vid_number,good);
    ASSERT (good)
    REPORT "Problem with CT input file (bad VID number)"
    SEVERITY failure;

    read(l,redun,good);
    ASSERT (good)
    REPORT "Problem with CT input file (bad redun)"
    SEVERITY failure;

    FOR i IN 1 TO (redun + 4) LOOP
       read(l,values(i),good);
      ASSERT (good)
      REPORT "Problem with CT input file (bad mask or pe location)"
      SEVERITY failure;
    END LOOP;

    read(l,temp_to,good);
    ASSERT (good)
    REPORT "Problem with CT input file (bad timeout value"
    SEVERITY failure;

    temp.timeout := temp_to;

    -- the second part does the decoding and assigning
    CASE redun IS
      WHEN 1 =>
         temp.redun_level := simplex;
      WHEN 3 =>
         temp.redun_level := triplex;
      WHEN 4 =>
         temp.redun_level := quad;
      WHEN OTHERS =>
        ASSERT FALSE
        REPORT "Bad redundancy level - assigning default"
        SEVERITY FAILURE;
         temp.redun_level := simplex;
    END CASE;

--
-- Change when C simulation has been updated !!!
--
    FOR i IN 1 TO (num_ne - 1) LOOP
      IF values(i) = 1 THEN
         temp.presence(i-1) := TRUE;
      ELSE
         temp.presence(i-1) := FALSE;
      END IF;
    END LOOP;
    FOR i IN 1 TO redun LOOP
      temp.members(i-1) := values(i + max_redun_level);
    END LOOP;

    ct_entry := temp;

  END read_ct_entry;
```

```
--*************************************************************************
-- Get_msg_length
--
-- This PROCEDURE gets the number of entries in the msg file
-------------------------------------------------------------------------

  PROCEDURE get_msg_length (input_file : IN TEXT;
              msg_length : OUT INTEGER)
  IS
  BEGIN
  END;

--*************************************************************************
-- Read_msg_entry
--
-- This PROCEDURE reads an entry in the msg input file
-------------------------------------------------------------------------

  PROCEDURE read_msg_entry (input_file : IN TEXT;
              msg_entry : OUT msg_type)
  IS
  BEGIN
  END read_msg_entry;

END tb_package;
```

10.6.5. Main Control Package

```
-------------------------------------------------------------------------
-- Main Control Package
--
-- This package contains types and constants used by the main
-- controller. Its main puprose is to abstract away the state
-- definitions for use with multiple architectures.
-------------------------------------------------------------------------

LIBRARY score;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE std.std_logic.ALL;

PACKAGE main_control_package IS

--*************************************************************************
-- This type is used by the NE to tell the scoreboerd what to do
--

  TYPE operation_type IS (unknown,idle,reset_state,update_ct,
            clear_timeouts,process_new_serp,continue);

--*************************************************************************

--*************************************************************************
-- This type is used by the scoreboard to inform the NE of what its doing
--

  TYPE return_operation_type IS (unknown,idle,busy,reset_complete,
              ct_update_complete,clear_complete,
              message_to_send,processing_complete);

--*************************************************************************

--*************************************************************************
-- The following two TYPES contain states for state machine PROCESSes
-- within the main controller.
--

  TYPE ptov_state_type IS (s0,s1,s2,s3,s4);

  TYPE serp_processor_state_type IS (unknown,idle,vote_serp,find_messages,
                  send_message,processing_complete);

--*************************************************************************

END main_control_package;
```

10.6.6. Voted SERP Package

```
--**************************************************************************
-- Voted Serp Package
--
-- This package contains types, subprograms, and constants used by the
-- sender and voter-timeout entities.
--------------------------------------------------------------------------

LIBRARY score;
USE score.scoreboard_package.ALL;
USE std.std_logic.ALL;
USE std.std_cmos.ALL;

PACKAGE voted_serp_package IS

  TYPE voted_serp_type IS RECORD
      obne,ibnf : flow_control_type;
    vid_is_simplex : BOOLEAN;
      source_vid,dest_vid : vid_type;
    class : class_type;
    obne_syndrome,ibnf_syndrome,sb_vote_syndrome : presence_type;
  END RECORD;

  TYPE voted_serp_memory_type IS ARRAY (INTEGER RANGE <>) OF
                    voted_serp_type;

  PROCEDURE message_is_legal ( VARIABLE vs_entry : INOUT voted_serp_type;
                SIGNAL hlf : IN BOOLEAN;
                VARIABLE valid : OUT BOOLEAN);


END voted_serp_package;


PACKAGE BODY voted_serp_package IS

  PROCEDURE message_is_legal ( VARIABLE vs_entry : INOUT voted_serp_type;
                SIGNAL hlf : IN BOOLEAN;
                VARIABLE valid : OUT BOOLEAN)
  IS
  BEGIN

    IF vs_entry.vid_is_simplex AND vs_entry.class.broadcast THEN
      valid := FALSE;
    ELSE
      valid := TRUE;
    END IF;

  END;

END voted_serp_package;
```

10.6.7. PID to VID Package

```
-------------------------------------------------------------------------
-- Pid_to_vid_package
--
-- This package contains a few declarations useful to the pid to
-- vid translation table.
-------------------------------------------------------------------------

LIBRARY score;
USE score.address_package.ALL;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE std.std_logic.ALL;

PACKAGE pid_to_vid_package IS

  CONSTANT table_size : INTEGER := num_ne * pe_per_ne + 2*max_vid;

-- For now, this type is exactly equivalent to ct_type. However, in the
-- future, the members part will be different since it will store an
-- address rather than an encoded location.

  TYPE pid_to_vid_entry_type IS RECORD
    vid : vid_type;
    redun_level : redun_level_type;
    presence : presence_type;
    members : members_type; -- these are really addresses
    timeout : timeout_type;
  END RECORD;

  TYPE pid_to_vid_table_type IS ARRAY(INTEGER RANGE <>) OF
                    pid_to_vid_entry_type;

--
-- TYPE vids_in_system is used to keep track of all the vids in the system.
-- The sender uses it to cycle through the voted serp memory looking for messages
--
  TYPE vids_in_system_memory_type IS ARRAY (INTEGER RANGE <>) OF address_type;

END pid_to_vid_package;
```

10.6.8. Dual Port RAM Package

```
--------------------------------------------------------------------------
-- Dual Port Ram Package
--
-- This package contains types and constants for the dual port ram
--------------------------------------------------------------------------

LIBRARY score;
USE score.scoreboard_package.ALL;
USE std.std_logic.ALL;

PACKAGE dpram_package IS

  CONSTANT write : t_wlogic;

  TYPE serp_memory_type IS ARRAY (INTEGER RANGE <>) OF serp_type;
  TYPE msg_memory_type IS ARRAY (INTEGER RANGE <>) OF msg_type;
  TYPE ct_memory_type IS ARRAY (INTEGER RANGE <>) OF ct_type;

END dpram_package;


PACKAGE BODY dpram_package IS

  CONSTANT write : t_wlogic := f0;

END dpram_package;
```

10.6.9. Scoreboard

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.main_control_package.ALL;
USE score.pid_to_vid_package.ALL;
USE score.voted_serp_package.ALL;
USE score.address_package.ALL;
ENTITY scoreboard IS

   PORT
      (
          message_to_send: OUT  BOOLEAN;
          operation_out: OUT  return_operation_type;
          operation_in: IN  operation_type;
          hlf: IN  BOOLEAN;
          ct_data: IN  ct_type;
          msg_data: OUT  msg_type;
          read_write: OUT  t_wlogic;
          clock: IN  t_wlogic;
          serp_data: IN  serp_type;
          sb_address: OUT  resolved_address
      );

END scoreboard;

LIBRARY SCORE;

USE std.std_logic.ALL;
ARCHITECTURE scoreboard OF scoreboard IS

   COMPONENT vote_timeout
      PORT
         (
             clock: IN  t_wlogic;
             serp_data: IN  serp_type;
             voted_serp_data: OUT  voted_serp_type;
             ptov_address: OUT  resolved_address := high_z_address;
             ptov_rw: OUT  t_wlogic;
             dpram_rw: OUT  t_wlogic;
             dpram_address: OUT  resolved_address := high_z_address;
             ptov_data: IN  pid_to_vid_entry_type;
             start_voting: IN  BOOLEAN;
             done_voting: OUT  BOOLEAN;
             num_vids: IN  INTEGER;
             voted_serp_rw: OUT  t_wlogic;
             clear_done: OUT  BOOLEAN;
             start_clear: IN  BOOLEAN;
             voted_serp_address: OUT  address_type
         );

   END COMPONENT;
   COMPONENT pid_to_vid
      GENERIC
         (
             read_delay: TIME := 10 ns
         );
      PORT
         (
```

```
        address: IN  resolved_address;
        ptov_out: OUT  pid_to_vid_entry_type;
        clock: IN  t_wlogic;
        read_write: IN  t_wlogic := f1;
        ptov_in: IN  pid_to_vid_entry_type
    );

END COMPONENT;
COMPONENT vids_in_system
   GENERIC
    (
        read_delay: TIME := 10 ns
    );
   PORT
    (
        clock: IN  t_wlogic;
        address: IN  resolved_address;
        read_write: IN  t_wlogic;
        data_in: IN  address_type;
        data_out: OUT  address_type
    );

END COMPONENT;
COMPONENT address_buffer
   PORT
    (
        input: IN  resolved_address;
        output: OUT  resolved_address;
        clock: IN  t_wlogic;
        pass_through: IN  BOOLEAN
    );

END COMPONENT;
COMPONENT sender
   PORT
    (
        clock: IN  t_wlogic;
        voted_serp_address: OUT  address_type;
        vs_rw: OUT  t_wlogic;
        broadcast_pending: OUT  BOOLEAN;
        dpram_address: OUT  resolved_address;
        dpram_rw: OUT  t_wlogic;
        msg_data: OUT  msg_type;
        voted_serp_data: IN  voted_serp_type;
        hlf: IN  BOOLEAN;
        message_to_send: OUT  BOOLEAN;
        start_processing: IN  BOOLEAN;
        done: OUT  BOOLEAN;
        num_vids: IN  INTEGER;
        continue: IN  BOOLEAN;
        ct_update: IN  BOOLEAN;
        vis_data: IN  address_type;
        vis_rw: OUT  t_wlogic;
        vis_address: OUT  resolved_address;
        pass_through: OUT  BOOLEAN
    );

END COMPONENT;
COMPONENT voted_serp_memory
   GENERIC
    (
        read_delay: TIME := 10 ns
    );
   PORT
```

```
      (
         port0_in: IN   voted_serp_type;
         port0_out: OUT   voted_serp_type;
         port0_address: IN   address_type;
         port1_address: IN   resolved_address;
         port1_out: OUT   voted_serp_type;
         clock: IN   t_wlogic;
         port0_rw: IN   t_wlogic;
         port1_rw: IN   t_wlogic := f1
      );

END COMPONENT;
COMPONENT main_controller
   PORT
      (
         dpram_rw: OUT   t_wlogic;
         ptov_rw: OUT   t_wlogic;
         dpram_address: OUT   resolved_address := high_z_address;
         clock: IN   t_wlogic;
         ct_data_in: IN   ct_type;
         operation_in: IN   operation_type;
         operation_out: OUT   return_operation_type;
         ptov_address: OUT   resolved_address := high_z_address;
         ptov_data: OUT   pid_to_vid_entry_type;
         start_voting: OUT   BOOLEAN;
         num_vids: OUT   INTEGER;
         start_clear: OUT   BOOLEAN;
         clear_done: IN   BOOLEAN;
         done_voting: IN   BOOLEAN;
         start_sender: OUT   BOOLEAN;
         sender_done: IN   BOOLEAN;
         message_to_send: IN   BOOLEAN;
         continue_processing: OUT   BOOLEAN;
         ct_update: OUT   BOOLEAN;
         vis_address: OUT   resolved_address;
         vis_rw: OUT   t_wlogic;
         vis_data: OUT   address_type
      );

END COMPONENT;

FOR translation_table:pid_to_vid
   USE CONFIGURATION SCORE.cpid_to_vid_arch;

FOR vis:vids_in_system
   USE CONFIGURATION SCORE.cvids_in_system_behavior;

FOR buff:address_buffer
   USE CONFIGURATION SCORE.caddress_buffer_behavior;

FOR sender_subsystem:sender
   USE CONFIGURATION SCORE.csender_behavior;

FOR voted_serp:voted_serp_memory
   USE  OPEN;

FOR controller:main_controller
   USE CONFIGURATION SCORE.cmain_control_behavior;
SIGNAL SGNL000079:   address_type;
SIGNAL SGNL000078:   t_wlogic;
SIGNAL SGNL000077:   resolved_address;
SIGNAL SGNL000075:   BOOLEAN;
SIGNAL SGNL000072:   BOOLEAN;
SIGNAL SGNL000071:   BOOLEAN;
```

```
    SIGNAL SGNL000070:   BOOLEAN;
    SIGNAL SGNL000044:   BOOLEAN;
    SIGNAL SGNL000043:   BOOLEAN;
    SIGNAL SGNL000042:   BOOLEAN;
    SIGNAL SGNL000040:   INTEGER;
    SIGNAL SGNL000026:   BOOLEAN;
    SIGNAL SGNL000018:   pid_to_vid_entry_type;
    SIGNAL SGNL000012:   resolved_address := high_z_address;
    SIGNAL ptovrw:   t_wlogic;
    SIGNAL SGNL000083:   t_wlogic := f1;
    SIGNAL SGNL000050:   t_wlogic;
    SIGNAL SGNL000099:   voted_serp_type;
    SIGNAL SGNL000084:   resolved_address;
    SIGNAL SGNL000051:   address_type;
    SIGNAL SGNL000048:   voted_serp_type;
    SIGNAL SGNL000096:   BOOLEAN;
    SIGNAL SGNL000098:   address_type;
    SIGNAL SGNL000032:   pid_to_vid_entry_type;
    SIGNAL feedback0:   BOOLEAN;


BEGIN
    message_to_send <= feedback0;


    voting_subsystem: vote_timeout
       PORT MAP (
           voted_serp_address => SGNL000051,
           start_clear => SGNL000042,
           clear_done => SGNL000043,
           voted_serp_rw => SGNL000050,
           num_vids => SGNL000040,
           done_voting => SGNL000044,
           start_voting => SGNL000026,
           ptov_data => SGNL000032,
           dpram_address => sb_address,
           dpram_rw => read_write,
           ptov_rw => ptovrw,
           ptov_address => SGNL000012,
           voted_serp_data => SGNL000048,
           serp_data => serp_data,
           clock => clock );

    translation_table: pid_to_vid
       PORT MAP (
           ptov_in => SGNL000018,
           read_write => ptovrw,
           clock => clock,
           ptov_out => SGNL000032,
           address => SGNL000012 );

    vis: vids_in_system
       PORT MAP (
           data_out => SGNL000098,
           data_in => SGNL000079,
           read_write => SGNL000078,
           address => SGNL000077,
           clock => clock );

    buff: address_buffer
       PORT MAP (
           pass_through => SGNL000096,
           clock => clock,
           output => SGNL000084,
```

```
        input => SGNL000098 );

    sender_subsystem: sender
       PORT MAP (
           pass_through => SGNL000096,
           vis_address => SGNL000077,
           vis_rw => SGNL000078,
           vis_data => SGNL000098,
           ct_update => SGNL000075,
           continue => SGNL000072,
           num_vids => SGNL000040,
           done => SGNL000071,
           start_processing => SGNL000070,
           message_to_send => feedback0,
           hlf => hlf,
           voted_serp_data => SGNL000099,
           msg_data => msg_data,
           dpram_rw => read_write,
           dpram_address => sb_address,
           broadcast_pending => OPEN,
           vs_rw => SGNL000083,
           voted_serp_address => SGNL000084,
           clock => clock );

    voted_serp: voted_serp_memory
       PORT MAP (
           port1_rw => SGNL000083,
           port0_rw => SGNL000050,
           clock => clock,
           port1_out => SGNL000099,
           port1_address => SGNL000084,
           port0_address => SGNL000051,
           port0_out => OPEN,
           port0_in => SGNL000048 );

    controller: main_controller
       PORT MAP (
           vis_data => SGNL000079,
           vis_rw => SGNL000078,
           vis_address => SGNL000077,
           ct_update => SGNL000075,
           continue_processing => SGNL000072,
           message_to_send => feedback0,
           sender_done => SGNL000071,
           start_sender => SGNL000070,
           done_voting => SGNL000044,
           clear_done => SGNL000043,
           start_clear => SGNL000042,
           num_vids => SGNL000040,
           start_voting => SGNL000026,
           ptov_data => SGNL000018,
           ptov_address => SGNL000012,
           operation_out => operation_out,
           operation_in => operation_in,
           ct_data_in => ct_data,
           clock => clock,
           dpram_address => sb_address,
           ptov_rw => ptovrw,
           dpram_rw => read_write );

END scoreboard;

CONFIGURATION cscoreboard_behav OF scoreboard IS
```

```
    FOR scoreboard

       FOR controller : main_controller
          USE CONFIGURATION score.cmain_control_behavior;
       END FOR;

       FOR translation_table : pid_to_vid
          USE CONFIGURATION score.cpid_to_vid_arch;
       END FOR;

       FOR vis : vids_in_system
      USE CONFIGURATION score.cvids_in_system_behavior;
       END FOR;

       FOR voting_subsystem : vote_timeout
      USE CONFIGURATION score.cvote_timeout_behav;
       END FOR;

       FOR voted_serp : voted_serp_memory
      USE CONFIGURATION score.cimproved_voted_serp_memory;
       END FOR;

       FOR sender_subsystem : sender
      USE CONFIGURATION score.csender_behavior;
       END FOR;

       FOR buff : address_buffer
      USE CONFIGURATION score.caddress_buffer_behavior;
       END FOR;

    END FOR;

END cscoreboard_behav;
```

10.6.10. Dual Port Ram

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.dpram_package.ALL;
USE score.active_package.ALL;
USE score.address_package.ALL;
ENTITY dpram IS

   GENERIC
      (
         read_delay: TIME := 10 ns
      );
   PORT
      (
         clock: IN  t_wlogic;
         Bct_out: OUT  ct_type;
         Bserp_out: OUT  serp_type;
         address1: IN  address_type;
         address0: IN  address_type;
         Bmsg_in: IN  msg_type := def_msg;
         Amsg_out: OUT  msg_type;
         Act_in: IN  ct_type;
         Aserp_in: IN  serp_type;
         RW1: IN  t_wlogic;
         RW0: IN  t_wlogic
      );

END dpram;

ARCHITECTURE dpram_behav OF dpram IS

BEGIN

  A0 : PROCESS(clock,address0,address1,rw0,rw1)

          VARIABLE serp_memory : serp_memory_type (mem_base TO 3*dpram_size);
          VARIABLE msg_memory : msg_memory_type (mem_base TO 3*dpram_size);
          VARIABLE ct_memory : ct_memory_type (mem_base TO 3*dpram_size);

   VARIABLE ran_process_once : BOOLEAN := FALSE;
   VARIABLE vid : vid_type := 0;

  BEGIN

--
-- This loop simply writes all the VID numbers into the
-- vid_number field of each CT entry. In the future, this will be
-- done by resetting the scoreboard. I do it here to save on simulator
-- time since this will be done by the time the simulator comes
-- up.
--
   IF NOT ran_process_once THEN
     FOR i IN 0 TO (max_vid - 1) LOOP
        ct_memory(ct_base + i).vid_number := vid;
       vid := vid + 1;
    END LOOP;
     ct_memory(ct_base + max_vid).vid_number := max_vid;
     ran_process_once := TRUE;
```

```
    END IF;

    IF clock = f1 AND clock'EVENT THEN

    -- take care of data port 1
      IF rw0 = write THEN
        serp_memory(address0) := Aserp_in;
        ct_memory(address0) := Act_in;
      ELSE
        Amsg_out <= msg_memory(address0) AFTER read_delay;
      END IF;

    -- take care of data port 2
      IF rw1 = write THEN
        msg_memory(address1) := Bmsg_in;
      ELSE
        Bserp_out <= serp_memory(address1) AFTER read_delay;
        Bct_out <= ct_memory(address1) AFTER read_delay;
      END IF;
    END IF;
  END PROCESS;

END dpram_behav;


CONFIGURATION cdpram_behav OF dpram IS
  FOR dpram_behav
  END FOR;
END cdpram_behav;
```

10.6.11. Voted SERP Memory

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.voted_serp_package.ALL;
USE std.std_cmos.ALL;
USE score.address_package.ALL;
ENTITY voted_serp_memory IS

   GENERIC
      (
         read_delay: TIME := 10 ns
      );
   PORT
      (
         port1_rw: IN  t_wlogic := f1;
         port0_rw: IN  t_wlogic;
         clock: IN  t_wlogic;
         port1_out: OUT  voted_serp_type;
         port1_address: IN  resolved_address;
         port0_address: IN  address_type;
         port0_out: OUT  voted_serp_type;
         port0_in: IN  voted_serp_type
      );

END voted_serp_memory;

--************************************************************************
-- Improved Voted Serp Memory Behavioral Architecture
--
-- This file contains an improved behavioral architecture of the voted
-- serp memory. It splits the memory operation into two parts, an
-- asynchronous part and a synchronous part.
------------------------------------------------------------------------

ARCHITECTURE improved_voted_serp_memory OF voted_serp_memory IS

   SIGNAL asynch_port0_out,asynch_port1_out : voted_serp_type;

BEGIN

  asynch : PROCESS (clock,port0_address,port1_address,port0_rw,port1_rw)

   VARIABLE voted_serp_mem :
           voted_serp_memory_type (mem_base TO dpram_size);

  BEGIN
   IF port0_rw = f0 THEN
     voted_serp_mem(port0_address) := port0_in;
   ELSE
     asynch_port0_out <= voted_serp_mem(port0_address);
   END IF;
   asynch_port1_out <= voted_serp_mem(port1_address);
  END PROCESS;

  synch : PROCESS(clock)
  BEGIN
   IF clock = f1 AND clock'EVENT THEN
     port1_out <= asynch_port1_out AFTER read_delay;
```

```
      port0_out <= asynch_port0_out AFTER read_delay;
    END IF;
  END PROCESS;

END improved_voted_serp_memory;


CONFIGURATION cimproved_voted_serp_memory OF voted_serp_memory IS
  FOR improved_voted_serp_memory
  END FOR;
END cimproved_voted_serp_memory;
```

10.6.12. PID to VID Table

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.pid_to_vid_package.ALL;
USE std.std_cmos.ALL;
USE score.address_package.ALL;
ENTITY pid_to_vid IS

   GENERIC
      (
         read_delay: TIME := 10 ns
      );
   PORT
      (
         ptov_in: IN  pid_to_vid_entry_type;
         read_write: IN  t_wlogic := f1;
         clock: IN  t_wlogic;
         ptov_out: OUT  pid_to_vid_entry_type;
         address: IN  resolved_address
      );

END pid_to_vid;

-------------------------------------------------------------------------
-- Pid_to_vid_arch
--
-- This is the behavioral architecture for the pid-to-vid translation
-- table. Its basically just a simple memory.
-------------------------------------------------------------------------

ARCHITECTURE pid_to_vid_arch OF pid_to_vid IS
BEGIN

  simple : PROCESS(clock,address,read_write)

--
-- NOTE that the table is MUCH larger than it has to be so that the
-- same resolution function can be used for ALL addresses. If the table
-- were smaller, a different high_z_address and resolution function
-- would need to be defined for each address range.
-- This way, when addresses are dropped to bits, no contortions will
-- result from inconsistencies

   VARIABLE pid_to_vid_table :
          pid_to_vid_table_type(mem_base TO dpram_size);

  BEGIN
   IF clock = f1 AND clock'EVENT THEN
     IF read_write = f0 THEN
        pid_to_vid_table(address) := ptov_in;
     ELSE
        ptov_out <= pid_to_vid_table(address) AFTER read_delay;
     END IF;
   END IF;

  END PROCESS;

END pid_to_vid_arch;
```

```
CONFIGURATION cpid_to_vid_arch OF pid_to_vid IS
  FOR pid_to_vid_arch
  END FOR;
END cpid_to_vid_arch;
```

10.6.13. VIDs in System Table

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE score.pid_to_vid_package.ALL;
USE score.address_package.ALL;
ENTITY vids_in_system IS

   GENERIC
      (
         read_delay: TIME := 10 ns
      );
   PORT
      (
         data_out: OUT  address_type;
         data_in: IN  address_type;
         read_write: IN  t_wlogic;
         address: IN  resolved_address;
         clock: IN  t_wlogic
      );

END vids_in_system;

--*****************************************************************************
-- This file contains the ARCHITECTURE for vids_in_system, a lookup
-- table which the sender uses to cycle through the voted_serp memory
-- looking for valid messages.
-------------------------------------------------------------------------------

ARCHITECTURE vids_in_system_behavior OF vids_in_system IS

BEGIN

  memory : PROCESS (clock,address,read_write,data_in)

   VARIABLE vis_memory :
           vids_in_system_memory_type (mem_base TO dpram_size);

  BEGIN
   IF clock = f1 AND clock'EVENT THEN
     IF read_write = f0 THEN
       vis_memory(address) := data_in;
     ELSE
       data_out <= vis_memory(address) AFTER read_delay;
     END IF;

   END IF;
  END PROCESS;

END;


CONFIGURATION cvids_in_system_behavior OF vids_in_system IS
  FOR vids_in_system_behavior
  END FOR;
END cvids_in_system_behavior;
```

10.6.14. Voting and Timeout Hardware

```
LIBRARY score;

LIBRARY voters;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE score.pid_to_vid_package.ALL;
USE score.voter_package.ALL;
USE score.voted_serp_package.ALL;
USE score.address_package.ALL;
ENTITY vote_timeout IS

   PORT
      (
          voted_serp_address: OUT  address_type;
          start_clear: IN  BOOLEAN;
          clear_done: OUT  BOOLEAN;
          voted_serp_rw: OUT  t_wlogic;
          num_vids: IN  INTEGER;
          done_voting: OUT  BOOLEAN;
          start_voting: IN  BOOLEAN;
          ptov_data: IN  pid_to_vid_entry_type;
          dpram_address: OUT  resolved_address := high_z_address;
          dpram_rw: OUT  t_wlogic;
          ptov_rw: OUT  t_wlogic;
          ptov_address: OUT  resolved_address := high_z_address;
          voted_serp_data: OUT  voted_serp_type;
          serp_data: IN  serp_type;
          clock: IN  t_wlogic
      );

END vote_timeout;

--*************************************************************************
-- Voter and Timeout Behavioral Architecture
--
-- This architecture conatins the (very) behavioral description of the
-- voting and timeout hardware. This first architecture is composed
-- entirely of process statements which communicate via signals.
-------------------------------------------------------------------------

ARCHITECTURE vote_timeout_behav OF vote_timeout IS

   TYPE temp_state_type IS (idle,vote,check_timeouts);
   SIGNAL temp_state : temp_state_type := idle;

   SIGNAL vote_state : vote_state_type;

--
-- timer_value holds the value of the timer.
--
   SIGNAL timer_value : timer_range;

--
-- Declare the signals that will be used to communicate between
-- processes
--
```

```
    SIGNAL voted_serp_entry : voted_serp_type;
    SIGNAL vote_values : serp_array(0 TO (max_redun_level - 1));
    SIGNAL current_vid : vid_type;
    SIGNAL presence : presence_type;
    SIGNAL timeout_value : timeout_type;
    SIGNAL read_obne_timeout,read_ibnf_timeout : t_wlogic;
    SIGNAL obne_timeout_address,ibnf_timeout_address : address_type;
    SIGNAL vid_is_simplex : BOOLEAN := FALSE;

--
-- vote_now tells the voter process to vote the vote_values
--
    SIGNAL vote_vid_now,done_voting_vid : BOOLEAN := FALSE;

--
-- Declare the signals which exit the voter
--
    SIGNAL voted_obne,voted_ibnf : flow_control_type;
    SIGNAL voted_dest_vid : vid_type;
    SIGNAL voted_class : class_type;

  BEGIN

--************************************************************************
-- controller PROCESS : implements the voter sub-controller. It reads in SERP
-- entries using the pid_to_vid_table, votes them, checks timeouts, collects
-- the three syndromes, and then collects all the voted data into a voted_serp
-- record and writes it into the voted_serp memory.
------------------------------------------------------------------------

  controller : PROCESS(clock,start_voting)

    VARIABLE temp_dpram_address : address_type;
    VARIABLE vids_voted : INTEGER := 0;

    VARIABLE current_ptov_entry : pid_to_vid_entry_type;
--
-- num_members is redun_level converted to a number, used as a loop control index
--
    VARIABLE num_members : INTEGER;

--
-- current_member is the index into the members array of the current_ptov_entry
--
    VARIABLE current_member : INTEGER;

  BEGIN
    IF clock = f1 AND clock'EVENT THEN
      CASE vote_state IS

--
-- State v0 is the idle state
--
        WHEN v0 =>
          done_voting <= FALSE;
          ptov_rw <= z0;
          dpram_rw <= z0;
          dpram_address <= high_z_address;
          voted_serp_rw <= z0;
          vids_voted := 0;
          vote_state <= v0;
          IF start_voting  THEN
        ptov_address <= vids_voted;
        ptov_rw <= f1;
```

```
        dpram_rw <= f1;
        vote_state <= v1;
           END IF;

--*************************************************************************
-- States v1 to v4 read in the ptov entry and then each SERP entry in
-- the VID
-- Be careful of the array bounds since current_member goes from 1 TO 4
-- while members_type goes from 0 TO 3.
-------------------------------------------------------------------------

--
-- wait for the ptov data to appear
--
         WHEN v1 =>
           vote_state <= v2;

         WHEN v2 =>
            current_ptov_entry := ptov_data;
            current_vid <= current_ptov_entry.vid;

            num_members := redun_to_int(current_ptov_entry.redun_level);
            timeout_value <= current_ptov_entry.timeout;
            presence <= current_ptov_entry.presence;
            vid_is_simplex <= (current_ptov_entry.redun_level = simplex);
           current_member := 0;
           vote_state <= v3;

--*************************************************************************
-- These next two states read each SERP entry in the VID into the
-- serp_array
-------------------------------------------------------------------------

         WHEN v3 =>
--
-- Check to see if redun_level serp entries have been read
--
           IF NOT (current_member = num_members) THEN
--
-- If not, then go on to read the serp entry from the dpram
--
              temp_dpram_address := serp_base +
            current_ptov_entry.members(current_member);
        dpram_address <= temp_dpram_address;
        vote_state <= v4;
           ELSE
--
-- If so, then go on to the voting state
--
        vote_state <= v6;
           END IF;

--
-- wait for the serp data to appear
--
         WHEN v4 =>
           vote_state <= v5;

--
-- v5 simply assigns the serp_data which appears on the data line to vote_values
--

         WHEN v5 =>
           vote_values(current_member) <= serp_data;
```

```
          current_member := current_member + 1;

          vote_state <= v3;

--*************************************************************************
-- End section to read in the serp entries
--------------------------------------------------------------------------

--
-- v6 tells the voter to vote and increments the vids_voted variable
--

        WHEN v6 =>
          vote_vid_now <= TRUE AFTER clock_period/2;
          vote_state <= v7;

--
-- v7 simply idles while the timeouts are being checked
--
        WHEN v7 =>
          IF done_voting_vid THEN
            vote_state <= v8;
          ELSE
       vote_state <= v7;
          END IF;

--
-- voting is done, so write the voted serp data into the voted serp memory
--
        WHEN v8 =>
          vote_vid_now <= FALSE AFTER clock_period/2;
          voted_serp_data <= voted_serp_entry;
          voted_serp_address <= current_ptov_entry.vid;
          voted_serp_rw <= f0 AFTER clock_period/2;
          vids_voted := vids_voted + 1;
          vote_state <= v9;

        WHEN v9 =>
          voted_serp_rw <= f1 AFTER clock_period/2;
          vote_state <= v10;

--
-- v10 checks to see if all vids have been voted. If so, then it signals that
-- voting is done and idles the voter subcontroller
--

        WHEN v10 =>
          voted_serp_rw <= f1;
          IF vids_voted = num_vids THEN
       done_voting <= TRUE;
       vote_state <= v0;
          ELSE
--
-- assert next ptov table address and start over
--
          ptov_address <= vids_voted;
       vote_state <= v1;
          END IF;
     END CASE;
   END IF;

  END PROCESS;

--*************************************************************************
```

```
-- voter PROCESS : this proces implements the voter. It uses overloaded
-- operators to convert the incoming data to t_wlogic_vectors, votes it
-- (bit for bit majority), and then converts it back to its original
-- high-level form.
-------------------------------------------------------------------------

  voter : PROCESS(clock,vote_vid_now)

--
-- For simplicities sake, the timeout memories are contained in this PROCESS
-- This allows faster verification while not sacrificing the readability
-- of the timreout rules
--
    VARIABLE obne_timeout_memory : timeout_memory_type(0 TO max_vid);
    VARIABLE ibnf_timeout_memory : timeout_memory_type(0 TO max_vid);

    VARIABLE to_address : address_type;
    VARIABLE to_set : BOOLEAN;
    VARIABLE diff : timeout_type;

    VARIABLE obne_unan,ibnf_unan : BOOLEAN;

  BEGIN
    IF clock = f1 AND clock'EVENT THEN
      CASE temp_state IS

        WHEN idle =>
          IF vote_vid_now THEN
           temp_state <= vote;
          END IF;
          done_voting_vid <= FALSE;

        WHEN vote =>
           vote_vid(voted_serp_entry,vote_values,current_vid,presence,
          obne_unan,ibnf_unan);
          voted_serp_entry.vid_is_simplex <= vid_is_simplex;
          temp_state <= check_timeouts;

        WHEN check_timeouts =>
--************************************************************************
-- Check obne timeout
--
           to_address := voted_serp_entry.source_vid;
           to_set := obne_timeout_memory(to_address).timeout_set;
--
-- Unanimous? If so, then clear any timeout set on the VID
--
          IF obne_unan THEN
             obne_timeout_memory(to_address).timeout_set := FALSE;
--
-- Majority? If so, then set a timeout if one hasn't been set or check for
-- timeout expiration if one has been set
--
          ELSIF voted_serp_entry.obne THEN
            IF obne_timeout_memory(to_address).timeout_set THEN
--
-- Check for timeout expiration
--
          diff := abs(timer_value-obne_timeout_memory(to_address).value);
--
-- no faults for now
--
          IF (diff > timeout_value) THEN
            NULL;
```

```
          ELSE
            voted_serp_entry.obne <= FALSE;
          END IF;
--
-- set a timeout
--
             ELSE
          obne_timeout_memory(to_address).value := timer_value;
          obne_timeout_memory(to_address).timeout_set := TRUE;
          voted_serp_entry.obne <= FALSE;
            END IF;
          END IF;
--
-- End obne timeout check
-------------------------------------------------------------------------

--*************************************************************************
-- Check ibnf timeout
--
           to_address := voted_serp_entry.source_vid;
           to_set := ibnf_timeout_memory(to_address).timeout_set;
--
-- Unanimous? If so, then clear any timeout set on the VID
--
          IF ibnf_unan THEN
             ibnf_timeout_memory(to_address).timeout_set := FALSE;
--
-- Majority? If so, then set a timeout if one hasn't been set or check for
-- timeout expiration if one has been set
--
          ELSIF voted_serp_entry.ibnf THEN
            IF ibnf_timeout_memory(to_address).timeout_set THEN
--
-- Check for timeout expiration
--
          diff := abs(timer_value-ibnf_timeout_memory(to_address).value);
--
-- no faults for now
--
         IF (diff > timeout_value) THEN
           NULL;
         ELSE
           voted_serp_entry.ibnf <= FALSE;
         END IF;
--
-- set a timeout
--
             ELSE
          ibnf_timeout_memory(to_address).value := timer_value;
          ibnf_timeout_memory(to_address).timeout_set := TRUE;
          voted_serp_entry.ibnf <= FALSE;
            END IF;
          END IF;
--
-- End ibnf timeout check
-------------------------------------------------------------------------
           done_voting_vid <= TRUE AFTER clock_period/2;
           temp_state <= idle;
      END CASE;
    END IF;

  END PROCESS;
```

```
--************************************************************************
-- obne_timeout_checker PROCESS : implements the obne timeout checker
--------------------------------------------------------------------------

--   obne_timeout_checker : PROCESS(clock,read_obne_timeout,
--              ibnf_timeout_address)

--       VARIABLE obne_timeout_memory : timeout_memory_type(0 TO max_vid);

--   BEGIN
--   END PROCESS;
--************************************************************************
-- ibnf_timeout_checker PROCESS : implements the ibnf timeout checker
--------------------------------------------------------------------------

--   ibnf_timeout_checker : PROCESS(clock,read_ibnf_timeout,
--              ibnf_timeout_address)

--       VARIABLE ibnf_timeout_memory : timeout_memory_type(0 TO max_vid);

--   BEGIN
--   END PROCESS;

--************************************************************************
-- timeout_clearer PROCESS : this process clears both the ibnf and obne
-- timeout memories
--------------------------------------------------------------------------

  timeout_clearer : PROCESS(clock,start_clear)
  BEGIN
    IF clock = f1 AND clock'EVENT THEN
      IF start_clear THEN
        clear_done <= TRUE;
      ELSE
        clear_done <= FALSE;
      END IF;
    END IF;
  END PROCESS;

--************************************************************************
-- timer PROCESS : this process implements the timeout timer. It counts from
-- 1 to max_timer_value and then wraps around
--------------------------------------------------------------------------

  timer : PROCESS(clock)

    VARIABLE temp_timer_value : timer_range := 0;

  BEGIN

    IF clock = f1 AND clock'EVENT THEN
      IF NOT (temp_timer_value = max_timer_value) THEN
        temp_timer_value := temp_timer_value + 1;
      ELSE
        temp_timer_value := init_timer_value;
      END IF;
      timer_value <= temp_timer_value;
    END IF;

  END PROCESS;

END vote_timeout_behav;
```

```
-- Provide a default configuration

CONFIGURATION cvote_timeout_behav OF vote_timeout IS
  FOR vote_timeout_behav
  END FOR;
END cvote_timeout_behav;
```

10.6.15. Sender

```
LIBRARY score;

USE std.std_logic.ALL;
USE std.std_cmos.ALL;
USE score.scoreboard_package.ALL;
USE score.voted_serp_package.ALL;
USE score.address_package.ALL;
ENTITY sender IS

   PORT
      (
          pass_through: OUT  BOOLEAN;
          vis_address: OUT  resolved_address;
          vis_rw: OUT  t_wlogic;
          vis_data: IN  address_type;
          ct_update: IN  BOOLEAN;
          continue: IN  BOOLEAN;
          num_vids: IN  INTEGER;
          done: OUT  BOOLEAN;
          start_processing: IN  BOOLEAN;
          message_to_send: OUT  BOOLEAN;
          hlf: IN  BOOLEAN;
          voted_serp_data: IN  voted_serp_type;
          msg_data: OUT  msg_type;
          dpram_rw: OUT  t_wlogic;
          dpram_address: OUT  resolved_address;
          broadcast_pending: OUT  BOOLEAN;
          vs_rw: OUT  t_wlogic;
          voted_serp_address: OUT  address_type;
          clock: IN  t_wlogic
      );

END sender;

--*************************************************************************
-- This file contains the behavioral architecture for the sender
-- entity. It's job is to cycle through the voted serp looking
-- for messages to send. When it finds a valid message, it gathers
-- all the information which the NE requires and then informs the
-- NE that a message needs to be sent. After it is sent, the sender
-- continues processing until either another message is found or all
-- voted serp entries have been processed.
--
-- NOTE: Broadcasts are not implemented yet.
-- FIXES REQUIRED : clearing of the IBNF bit after a message is sent
--          valid message checking (NULL dest_vid delivery)
--          invalid destination VID checking
-------------------------------------------------------------------------

ARCHITECTURE sender_behavior OF sender IS

   TYPE sender_state_type IS (send0,send1,send2,send3,send4,send5,
                send6,send7,send8);

   SIGNAL sender_state : sender_state_type;
   SIGNAL time_stamper_signal : TIME := 0 ns;

BEGIN
```

```
   sender_state_machine : PROCESS(clock,start_processing,continue)

--
-- Where_to_start tells the sender where to begin looking for valid messages
-- (its a pointer into the vids_in_system translation table) This ensures
-- fairness because the same VID cannot keep sending a message to the exclusion
-- of others.
--
    VARIABLE where_to_start : address_type := 0;
    VARIABLE temp_vis_address : address_type := 0;
    VARIABLE valid_message : BOOLEAN := TRUE;
    VARIABLE source_entry,dest_entry : voted_serp_type;
    VARIABLE message : msg_type;


  BEGIN
    IF clock = f1 AND clock'EVENT THEN
      CASE sender_state IS

        WHEN send0 =>
          IF start_processing THEN
        temp_vis_address := where_to_start;
        vis_address <= temp_vis_address;
        vis_rw <= f1;
        vs_rw <= f1;
        pass_through <= TRUE;
        sender_state <= send1;
        done <= FALSE AFTER clock_period/2;

        ASSERT FALSE REPORT "Beginning scan for valid messages";
           ELSE
        done <= FALSE;
        pass_through <= FALSE;
        dpram_rw <= z0;
        vis_rw <= z0;
        vs_rw <= z0;
        voted_serp_address <= high_z_address;
        vis_address <= high_z_address;
        dpram_address <= high_z_address;
           END IF;

--
-- send1 is a wait state for the vids_in_system memory
--
        WHEN send1 =>
          IF temp_vis_address = num_vids THEN
              temp_vis_address := 0;
          ELSE
             temp_vis_address := temp_vis_address + 1;
          END IF;
          sender_state <= send2;

--
-- send2 waits for the voted_serp_memory
--
        WHEN send2 =>
          sender_state <= send3;

--
-- send3 checks the OBNE of the voted serp entry. If its set, it asserts the
-- address of the destination vid to check its IBNF.
--

        WHEN send3 =>
```

```
          source_entry := voted_serp_data;
          IF source_entry.obne THEN
--
-- This VID wants to send a message (very badly, I may add), so check the IBNF
-- of the destination VID. Also check for illegal messages.
--
       IF source_entry.vid_is_simplex THEN
          message_is_legal(source_entry,hlf,valid_message);
       ELSE
          valid_message := TRUE;
       END IF;

       IF valid_message THEN
          pass_through <= FALSE;
          voted_serp_address <= source_entry.dest_vid;
          vs_rw <= f1;
          sender_state <= send4;
       ELSE
          sender_state <= send8;
       END IF;
          ELSE
       sender_state <= send8;
          END IF;
--
-- Wait for the data to appear on the data lines
--

        WHEN send4 =>
          sender_state <= send5;

        WHEN send5 =>
          dest_entry := voted_serp_data;
          IF (dest_entry.ibnf) THEN
--
-- A valid message exists, so assemble a message data structure and signal the
-- main controller. Also begin a write to the voted serp memory to set
-- the ibnf_processed field to TRUE. This prevents two messages from being
-- sent to the same VID in the same SERP round. (not implemented yet)
--

       sender_state <= send6;
         ELSE
       sender_state <= send8;
         END IF;

        WHEN send6 =>
           message.source_vid := source_entry.source_vid;
           message.dest_vid := source_entry.dest_vid;
           message.class := source_entry.class;
           message.vote_syndrome := source_entry.sb_vote_syndrome;
           message.obne_syndrome := source_entry.obne_syndrome;
           message.ibnf_syndrome := source_entry.ibnf_syndrome;
           message.timestamp := time_stamper_signal;
--
-- NOTE : sources and dests fields of msg_type are not implemented yet
--
          msg_data <= message;
          dpram_rw <= f0;
          dpram_address <= msg_base;
          message_to_send <= TRUE;
          sender_state <= send7;

       WHEN send7 =>
          dpram_rw <= f1;
```

```
          message_to_send <= FALSE;
          IF continue THEN
        sender_state <= send8;
          ELSE
        sender_state <= send7;
          END IF;

--
-- Check to see if entire voted serp has been processed. If not, then start
-- the cycle again.
--
       WHEN send8 =>
--
-- Have we processed the entire voted_serp?
--
          IF temp_vis_address = where_to_start THEN
--
-- IF yes, then signal DONE and go to the idle state
--
       IF where_to_start = num_vids THEN
         where_to_start := 0;
       ELSE
         where_to_start := where_to_start + 1;
       END IF;
       done <= TRUE AFTER clock_period/2;
       sender_state <= send0;
--
-- If no, then make sure temp_vis_address hasn't been incremented one too far,
-- assign the new vis_address and repeat the cycle.
--
          ELSE
       vis_address <= temp_vis_address;
       voted_serp_address <= high_z_address;
       vis_rw <= f1;
            pass_through <= TRUE;
            sender_state <= send1;
          END IF;
     END CASE;
   END IF;
  END PROCESS;

  time_stamper_signal <= (time_stamper_signal + clock_period) WHEN
            (clock = f1 AND clock'EVENT) ELSE time_stamper_signal;

END sender_behavior;


CONFIGURATION csender_behavior OF sender IS
  FOR sender_behavior
  END FOR;
END csender_behavior;
```

10.6.16. Main Controller

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.main_control_package.ALL;
USE std.std_cmos.ALL;
USE score.active_package.ALL;
USE score.pid_to_vid_package.ALL;
USE score.address_package.ALL;
ENTITY main_controller IS

   PORT
      (
         vis_data: OUT  address_type;
         vis_rw: OUT  t_wlogic;
         vis_address: OUT  resolved_address;
         ct_update: OUT  BOOLEAN;
         continue_processing: OUT  BOOLEAN;
         message_to_send: IN  BOOLEAN;
         sender_done: IN  BOOLEAN;
         start_sender: OUT  BOOLEAN;
         done_voting: IN  BOOLEAN;
         clear_done: IN  BOOLEAN;
         start_clear: OUT  BOOLEAN;
         num_vids: OUT  INTEGER;
         start_voting: OUT  BOOLEAN;
         ptov_data: OUT  pid_to_vid_entry_type;
         ptov_address: OUT  resolved_address := high_z_address;
         operation_out: OUT  return_operation_type;
         operation_in: IN  operation_type;
         ct_data_in: IN  ct_type;
         clock: IN  t_wlogic;
         dpram_address: OUT  resolved_address := high_z_address;
         ptov_rw: OUT  t_wlogic;
         dpram_rw: OUT  t_wlogic
      );

END main_controller;

--*************************************************************************
-- Main Controller Behavioral Architecture
--
-------------------------------------------------------------------------

ARCHITECTURE main_control_behavior OF main_controller IS

   SIGNAL ptov_state : ptov_state_type := s0;
   SIGNAL serp_processor_state : serp_processor_state_type := unknown;
   SIGNAL start_ct_update,ct_update_done : BOOLEAN := FALSE;
   SIGNAL start_processing,done_processing : BOOLEAN := FALSE;

BEGIN

  ASSERT NOT(operation_in = unknown)
  REPORT "port OPERATION in main_controller in unknown state"
  SEVERITY ERROR;

--
-- The ct_update port is used to inform the sender to reset its
```

```
-- where_to_start variable. If it didn't, it might end up pointing
-- to a vid which no longer exists
--
  ct_update <= start_ct_update;

-- This process simply serves as a dispatcher. The variable
-- temp_operation is used to dispatch in order to prevent two
-- operations to be pending simultaneously. This could occur if
-- dispatching was done off of the operation port itself.

  main_state_machine : PROCESS(clock,operation_in)

   VARIABLE temp_operation : operation_type := unknown;
--
-- op_out is used to read the value of the operation_out signal. A
-- BUFFER port should be used, but they aren't supported yet
--
   VARIABLE op_out : return_operation_type := idle;

  BEGIN
   IF clock = f1 AND clock'EVENT THEN
     CASE temp_operation IS
       -- provide a kick start out of unknown
      WHEN unknown =>
      temp_operation := operation_in;
      operation_out <= unknown;
      op_out := unknown;

      WHEN idle =>
      temp_operation := operation_in;
      operation_out <= idle;
      op_out := idle;

--
-- For now, a reset is defined as updating the CT and clearing all
-- timeouts. This is to avoid multiple drivers. In the future, a reset
-- must also copy all the vid numbers into the first byte of each
-- CT entry
--

      WHEN reset_state =>
      IF NOT ct_update_done THEN
--
-- The second IF is necessary to avoid continually performaing a ct_update
-- start_ct_update must be made FALSE at some point before ct_update_done
-- becomes TRUE
--
        IF NOT (op_out = busy) THEN
          start_ct_update <= TRUE;
        ELSE
          start_ct_update <= FALSE;
        END IF;

        operation_out <= busy;
        op_out := busy;
      ELSIF NOT clear_done THEN
--
-- temporary, remove when timeouts implemented
--
        start_clear <= TRUE;
      ELSE
        temp_operation := operation_in;
        operation_out <= reset_complete;
        op_out := reset_complete;
```

```
          start_clear <= FALSE;
        END IF;

        WHEN update_ct =>
        IF NOT ct_update_done THEN
          IF NOT (op_out = busy) THEN
            start_ct_update <= TRUE;
          ELSE
            start_ct_update <= FALSE;
          END IF;

          operation_out <= busy;
          op_out := busy;
        ELSE
          temp_operation := operation_in;
          operation_out <= ct_update_complete;
          op_out := ct_update_complete;
        END IF;

        WHEN clear_timeouts =>
        IF NOT clear_done THEN
          start_clear <= TRUE;
          operation_out <= busy;
          op_out := busy;
        ELSE
          temp_operation := operation_in;
          operation_out <= clear_complete;
          op_out := clear_complete;
          start_clear <= FALSE;
        END IF;

        WHEN process_new_serp =>
        IF NOT done_processing THEN
--
-- The second IF statement prevents the start_processing signal from
-- remaining TRUE for too long
--
          IF op_out = busy THEN
            start_processing <= FALSE AFTER clock_period/2;
          ELSE
            start_processing <= TRUE AFTER clock_period/2;
          END IF;
          operation_out <= busy;
          op_out := busy;
        ELSE
          temp_operation := operation_in;
          op_out := processing_complete;
          operation_out <= op_out;
        END IF;

        WHEN continue =>
        temp_operation := operation_in;
        operation_out <= busy;
        op_out := busy;
      END CASE;
    END IF;

  END PROCESS;

--*************************************************************************
-- This state machine implements the pid_to_vid translation table
-- generator
-------------------------------------------------------------------------
```

```
   ptov_state_machine : PROCESS (clock,start_ct_update)

    VARIABLE pid_to_vid_entry : pid_to_vid_entry_type;
    VARIABLE ct_address : address_type := ct_base;
    VARIABLE vids_in_system : INTEGER := 0;

   BEGIN
    IF clock = f1 AND clock'EVENT THEN
      CASE ptov_state IS

--
-- s0 is the idle state
--
       WHEN s0 =>
         IF start_ct_update THEN
           ptov_state <= s1;
       ct_update_done <= FALSE;
         ELSE
           ptov_state <= s0;
       dpram_address <= high_z_address;
       ptov_address <= high_z_address;
       vis_address <= high_z_address;
       vis_rw <= z0;
       ptov_rw <= z0;
       dpram_rw <= z0;
         END IF;
--
-- s1 asserts the ptov address and the ct_address into the dpram
--
       WHEN s1 =>
       ptov_address <= vids_in_system;
            dpram_address <= ct_address;
       ptov_rw <= f1;
       dpram_rw <= f1;
       ct_address := ct_address + 1;
       ptov_state <= s2;

--
-- s2 is a wait state
--
       WHEN s2 =>
       ptov_state <= s3;


--************************************************************************
-- s3 reads the ct entry at the address asserted by s1. If the redun level is
-- zero it skips to the next ct entry. Otherwise it constructs a pid_to_vid
-- table entry and writes it into the table. It also checks to see if its
-- reached the end of the ct. If so, it asserts ct_update_done, tri-states
-- the dpram address line, and goes to the idle state(s0)
--************************************************************************

       WHEN s3 =>
      IF NOT(ct_data_in.redun_level = zero) THEN
--
-- Found a new vid, so increment the counter
--
        vis_address <= vids_in_system;
        vis_data <= ct_data_in.vid_number;
        vids_in_system := vids_in_system + 1;

        pid_to_vid_entry.vid := ct_data_in.vid_number;
        pid_to_vid_entry.redun_level:= ct_data_in.redun_level;
        pid_to_vid_entry.presence := ct_data_in.presence;
```

```
          pid_to_vid_entry.members := ct_data_in.members;
          pid_to_vid_entry.timeout := ct_data_in.timeout;

          ptov_data <= pid_to_vid_entry;
--
-- Wait for a falling edge to assert the write signal
--
          ptov_rw <= f0 AFTER clock_period/2;
          vis_rw <= f0 AFTER clock_period/2;
          ptov_state <= s4;
        ELSE
--
-- else go to next ct entry
--
          ptov_state <= s1;
        END IF;

        IF ct_data_in.vid_number = max_vid THEN
          ASSERT FALSE REPORT "Done with translation table";
          ptov_state <= s0;
          ct_update_done <= TRUE;
--
-- this is a slight optimization to only assign a value to this port (num_vids)
-- once instead of over and over again
--
          num_vids <= vids_in_system;

        END IF;

--
-- state s4 simply gives enough time for the write signal to be taken
--
        WHEN s4 =>
        ptov_rw <= f1 AFTER clock_period/2;
        vis_rw <= f1 AFTER clock_period/2;
        ptov_state <= s1;

--
-- Appease the syntax deity by including this clause
--
        WHEN OTHERS =>
          ASSERT FALSE REPORT "Unimplemented state in ct_update controller";
          ptov_state <= s0;
      END CASE;
    END IF;
  END PROCESS;

--************************************************************************
-- This process takes care of all the control signals involved in
-- processing the SERP
------------------------------------------------------------------------

  serp_processor : PROCESS (clock,start_processing,done_voting,
                sender_done,message_to_send)

  BEGIN
    IF clock = f1 AND clock'EVENT THEN
      CASE serp_processor_state IS

        WHEN unknown =>
          serp_processor_state <= idle;

        WHEN idle =>
          done_processing <= FALSE;
```

```
          IF start_processing THEN
        start_voting <= TRUE;
        serp_processor_state <= vote_serp;
          ELSE
        start_voting <= FALSE;
        start_sender <= FALSE;
          END IF;

        WHEN vote_serp =>
          start_voting <= FALSE;
          IF done_voting THEN
            start_sender <= TRUE;
        serp_processor_state <= find_messages;
        ASSERT FALSE REPORT "SERP Voting Done";
          END IF;

        WHEN find_messages =>
          start_sender <= FALSE;
          continue_processing <= FALSE;
          IF message_to_send THEN
        serp_processor_state <= send_message;
          ELSIF sender_done       THEN
        serp_processor_state <= processing_complete;
          END IF;

        WHEN send_message =>
          IF operation_in = continue THEN
          ASSERT FALSE REPORT "Sent a message";
            continue_processing <= TRUE;
            serp_processor_state <= find_messages;
          ELSE
        serp_processor_state <= send_message;
          END IF;

        WHEN processing_complete =>
          done_processing <= TRUE AFTER clock_period/2;
          serp_processor_state <= idle;
          ASSERT FALSE REPORT "Processing is complete";

      END CASE;
    END IF;
  END PROCESS;

END main_control_behavior;


CONFIGURATION cmain_control_behavior OF main_controller IS
  FOR main_control_behavior
  END FOR;
END cmain_control_behavior;
```

10.6.17. Address Buffer

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.address_package.ALL;
ENTITY address_buffer IS

   PORT
      (
          pass_through: IN  BOOLEAN;
          clock: IN  t_wlogic;
          output: OUT  resolved_address;
          input: IN  resolved_address
      );

END address_buffer;

--*****************************************************************************
-- This component simply acts as a buffer to turn an address line on
-- and off (tri_state);
-------------------------------------------------------------------------------

ARCHITECTURE address_buffer_behavior OF address_buffer IS

BEGIN

    output <= input WHEN pass_through ELSE
          high_z_address;

END address_buffer_behavior;


CONFIGURATION caddress_buffer_behavior OF address_buffer IS
  FOR address_buffer_behavior
  END FOR;
END caddress_buffer_behavior;
```

10.6.18. Scoreboard Subsystem

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.address_package.ALL;
USE score.main_control_package.ALL;
ENTITY sb_subsystem IS

   PORT
      (
          message_to_send: OUT  BOOLEAN;
         hlf: IN  BOOLEAN;
          operation_out: OUT  return_operation_type;
          operation_in: IN  operation_type;
          msg_data_out: OUT  msg_type;
          read_write: IN  t_wlogic;
         done: OUT  t_wlogic;
          ct_data_in: IN  ct_type;
          serp_data_in: IN  serp_type;
          address: IN  address_type;
          clock: IN  t_wlogic
      );

END sb_subsystem;

LIBRARY SCORE;

USE std.std_logic.ALL;
ARCHITECTURE sb_subsystem OF sb_subsystem IS

   COMPONENT scoreboard
      PORT
         (
             sb_address: OUT  resolved_address;
             serp_data: IN  serp_type;
             clock: IN  t_wlogic;
             read_write: OUT  t_wlogic;
             msg_data: OUT  msg_type;
             ct_data: IN  ct_type;
             hlf: IN  BOOLEAN;
             operation_in: IN  operation_type;
             operation_out: OUT  return_operation_type;
             message_to_send: OUT  BOOLEAN
         );

   END COMPONENT;
   COMPONENT dpram
      GENERIC
         (
             read_delay: TIME := 10 ns
         );
      PORT
         (
            RW0: IN  t_wlogic;
            RW1: IN  t_wlogic;
            Aserp_in: IN  serp_type;
            Act_in: IN  ct_type;
            Amsg_out: OUT  msg_type;
            Bmsg_in: IN  msg_type := def_msg;
```

```
            address0: IN   address_type;
            address1: IN   address_type;
            Bserp_out: OUT   serp_type;
            Bct_out: OUT   ct_type;
            clock: IN   t_wlogic
         );

   END COMPONENT;

   FOR behav_sb:scoreboard
      USE CONFIGURATION SCORE.cscoreboard_behav;

   FOR dp_ram:dpram
      USE CONFIGURATION SCORE.cdpram_behav;
   SIGNAL sb_ct:   ct_type;
   SIGNAL sb_serp:   serp_type;
   SIGNAL sb_address:   address_type;
   SIGNAL sb_rw:   t_wlogic;


BEGIN

   behav_sb: scoreboard
      PORT MAP (
          message_to_send => message_to_send,
          operation_out => operation_out,
          operation_in => operation_in,
         hlf => hlf,
         ct_data => sb_ct,
         msg_data => msg_data_out,
         read_write => sb_rw,
         clock => clock,
         serp_data => sb_serp,
         sb_address => sb_address );

   dp_ram: dpram
      PORT MAP (
         clock => clock,
         Bct_out => sb_ct,
         Bserp_out => sb_serp,
         address1 => sb_address,
         address0 => address,
         Bmsg_in => OPEN,
         Amsg_out => OPEN,
         Act_in => ct_data_in,
         Aserp_in => serp_data_in,
         RW1 => sb_rw,
         RW0 => read_write );

END sb_subsystem;

-------------------------------------------------------------------------
-- Behavioral Scoreboard Subsystem Configuration
--
-- This is the configuration for the top-level scoreboard subsystem
-------------------------------------------------------------------------

CONFIGURATION csb_behav_subsystem OF sb_subsystem IS

   FOR sb_subsystem

      FOR dp_ram:dpram
         USE CONFIGURATION  score.cdpram_behav;
      END FOR;
```

```
      FOR behav_sb:scoreboard
         USE  CONFIGURATION  score.cscoreboard_behav;
       END FOR;

   END FOR;

END  csb_behav_subsystem;

CONFIGURATION  csb_behav_tb  OF  sb_testbench  IS
  FOR  sb_behav_tb
    FOR sbs  :  sb_subsystem
   USE CONFIGURATION  score.csb_behav_subsystem;
    END FOR;
  END FOR;
END  csb_behav_tb;
```

10.6.19. Testbench

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.tb_package.ALL;
USE score.dpram_package.ALL;
USE std.std_cmos.ALL;
USE score.active_package.ALL;
USE score.main_control_package.ALL;
USE score.address_package.ALL;
ENTITY sb_testbench IS


END sb_testbench;

--*****************************************************************************
-- Scoreboard Behavioral Testbench
--
-- This architecture constains code to test the highly behavioral
-- version of the scoreboard model
-- I have liberally used ASSERT statements throughout the design as
-- "signposts" to when critical actions have occurred. They allow easy
-- zooming to different areas inside of results display
-------------------------------------------------------------------------------

ARCHITECTURE sb_behav_tb OF sb_testbench IS

COMPONENT sb_subsystem
   PORT
      (
          message_to_send: OUT  BOOLEAN;
          hlf: IN  BOOLEAN;
          operation_out: OUT  return_operation_type;
          operation_in: IN  operation_type;
          msg_data_out: OUT  msg_type;
          read_write: IN  t_wlogic;
          done: OUT  t_wlogic;
          ct_data_in: IN  ct_type;
          serp_data_in: IN  serp_type;
          address: IN  address_type;
          clock: IN  t_wlogic
      );

  END COMPONENT;

  SIGNAL read_write : t_wlogic := f1;
  SIGNAL operation_in : operation_type := unknown;
  SIGNAL operation_out : return_operation_type := unknown;
  SIGNAL hlf : BOOLEAN := TRUE;
  SIGNAL message_to_send : BOOLEAN;
  SIGNAL clock : t_wlogic := clk_active;
  SIGNAL done : t_wlogic := f0;

  SIGNAL serp_data_in  : serp_type;
  SIGNAL ct_data_in  : ct_type;
  SIGNAL msg_data_out : msg_type;
  SIGNAL address : address_type;

BEGIN
```

```
   sbs : sb_subsystem
     PORT MAP (
    message_to_send => message_to_send,
    hlf => hlf,
    operation_in => operation_in,
    operation_out => operation_out,
    msg_data_out => msg_data_out,
    read_write => read_write,
    ct_data_in => ct_data_in,
    serp_data_in => serp_data_in,
    address => address,
    done => done,
    clock => clock );
   driver : PROCESS

--
-- declare temporary variables to hold signal values before assignment
-- 't' in front means temporary
--

    VARIABLE tsd : serp_memory_type (0 TO dpram_size);
    VARIABLE tmsg : msg_memory_type (0 TO dpram_size);
    VARIABLE tct : ct_memory_type (0 TO dpram_size);
    VARIABLE taddress : address_type;

    VARIABLE do_ct_update : BOOLEAN := FALSE;
    VARIABLE num_vids : INTEGER := 0;
    VARIABLE num_serp_entries : INTEGER := 0;
    VARIABLE num_messages,cnum_messages : INTEGER := 0;
   BEGIN

--
-- read in the ct
-- num_vids is the number of vids to read into the simulation
--
-- process 4 SERPs

    FOR i IN 1 TO 4 LOOP
       get_status(test_data,do_ct_update,num_vids,num_serp_entries,
           cnum_messages);
      IF do_ct_update THEN
        FOR i IN 0 TO (num_vids - 1) LOOP
           read_ct_entry(test_data,tct(i));
        END LOOP;

--
-- write the ct into memory and perform a reset
--

        FOR i IN 0 TO (num_vids - 1) LOOP
          WAIT UNTIL clock = f0 AND clock'EVENT;
          address <= ct_base + tct(i).vid_number;
          ct_data_in <= tct(i);
          read_write <= f0;
        END LOOP;
--
-- must WAIT so that the last ct entry is writen into the dpram
--
        WAIT UNTIL clock = f0 AND clock'EVENT;
        read_write <= f1;
        operation_in <= reset_state;
        ASSERT FALSE REPORT "Beginning Initial Reset";
```

```
        WAIT FOR clock_period;
        operation_in <= idle;

        WAIT UNTIL operation_out = reset_complete
          AND clock = f1 AND clock'EVENT;
        ASSERT FALSE REPORT "Initial Reset Complete";
      END IF;

--
-- read in the first SERP
--
      FOR i IN 0 TO (num_serp_entries - 1) LOOP
        read_serp_entry(test_data,tsd(i));
      END LOOP;

--
-- write the first SERP into memory and begin processing it
--
      FOR serp_loc IN 0 TO (num_serp_entries - 1) LOOP
        WAIT UNTIL clock = f0 AND clock'EVENT;
        address <= serp_base + serp_loc;
        serp_data_in <= tsd(serp_loc);
        read_write <= f0;
      END LOOP;
      WAIT UNTIL clock = f0 AND clock'EVENT;

      read_write <= f1;
      operation_in <= process_new_serp;
      ASSERT FALSE REPORT "Processing First SERP";
      WAIT FOR clock_period;
      operation_in <= idle;

      WHILE NOT (operation_out = processing_complete) LOOP
        IF message_to_send THEN
          num_messages := num_messages + 1;
          operation_in <= continue AFTER clock_period/2;
          WAIT UNTIL clock = f1 AND clock'EVENT;
        ELSE
          operation_in <= idle AFTER clock_period/2;
          WAIT UNTIL clock = f1 AND clock'EVENT;
        END IF;
        IF operation_out = processing_complete THEN
          EXIT;
        END IF;
      END LOOP;
    END LOOP;

    WAIT;
  END PROCESS;

  clock_driver : PROCESS
  BEGIN
    clock <= NOT clock;
    WAIT FOR clock_period/2;
  END PROCESS;

END sb_behav_tb;
```

10.7. Structural VHDL for theVoting and Timeout Hardware

This appendix contains the VHDL source code for the uncompleted structural architecture of the voting and timeout hardware.

10.7.1. Voting and Timeout Hardware

```
LIBRARY score;

LIBRARY voters;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE score.pid_to_vid_package.ALL;
USE score.voter_package.ALL;
USE score.voted_serp_package.ALL;
USE score.address_package.ALL;
ENTITY vote_timeout IS

   PORT
      (
          voted_serp_address: OUT  address_type;
          start_clear: IN  BOOLEAN;
          clear_done: OUT  BOOLEAN;
          voted_serp_rw: OUT  t_wlogic;
          num_vids: IN  INTEGER;
          done_voting: OUT  BOOLEAN;
          start_voting: IN  BOOLEAN;
          ptov_data: IN  pid_to_vid_entry_type;
          dpram_address: OUT  resolved_address := high_z_address;
          dpram_rw: OUT  t_wlogic;
          ptov_rw: OUT  t_wlogic;
          ptov_address: OUT  resolved_address := high_z_address;
          voted_serp_data: OUT  voted_serp_type;
          serp_data: IN  serp_type;
          clock: IN  t_wlogic
      );

END vote_timeout;

--*************************************************************************
-- Voter and Timeout Structural Architecture
--
-- This ARCHITECTURE contains the structural implementation of the
-- voting and timeout subsection. Actually, its also partly dataflow.
-------------------------------------------------------------------------

ARCHITECTURE vote_timeout_struct OF vote_timeout IS

   CONSTANT control_delay : TIME := clock_period/4;
   CONSTANT obne_bit_pos : INTEGER := 7;
   CONSTANT ibnf_bit_pos : INTEGER := 6;

--
-- In this next TYPE, "rse" stands for "read SERP entry" and "v" stands
-- for "vote".
--
   TYPE struct_vote_state_type IS (rse0,rse1,rse2,rse3,rse4,rse5,v0,
                    v1,v2,v3,v4,v5,v6,v7,v8,v9,v10);
```

```
    SIGNAL vote_state : struct_vote_state_type;

--
-- The following signals are repositories for intermediate data
--
    SIGNAL voted_data : BIT_VECTOR(7 DOWNTO 0);
    SIGNAL unan : BIT_VECTOR(7 DOWNTO 0);
    SIGNAL a_syndrome : BIT_VECTOR(7 DOWNTO 0);
    SIGNAL b_syndrome : BIT_VECTOR(7 DOWNTO 0);
    SIGNAL c_syndrome : BIT_VECTOR(7 DOWNTO 0);
    SIGNAL d_syndrome : BIT_VECTOR(7 DOWNTO 0);
    SIGNAL overall_vote_syndrome : presence_type;

    SIGNAL voted_serp_entry : voted_serp_type;
    SIGNAL voted_obne,voted_ibnf : flow_control_type;
    SIGNAL voted_dest_vid : vid_type;
    SIGNAL voted_class : class_type;

    SIGNAL start_to_check,check_done : BOOLEAN;
    SIGNAL obne_syndrome,ibnf_syndrome : presence_type;
    SIGNAL clear_obne,clear_ibnf : BOOLEAN := FALSE;
    SIGNAL current_timer_value : timer_range;

--
-- These signals are "registers" for holding information
--
    SIGNAL source_vid : vid_type;
    SIGNAL presence : presence_type;
    SIGNAL redun_level : redun_level_type;
    SIGNAL timeout_value : timeout_type;
    SIGNAL vid_is_simplex : BOOLEAN;
    SIGNAL is_flow_control : BIT;

--
-- The following signals are used mainly by the controller. In this
-- ARCHITECTURE the controller is not a separate component. It seems
-- easier to debug, but it's slower to compile (where's my SPARC2, eh?).
--
    TYPE bv_array IS ARRAY (NATURAL RANGE <>) OF BIT_VECTOR(7 DOWNTO 0);

    SIGNAL vote_values : serp_array (0 TO (max_redun_level - 1));
    SIGNAL bit_vote_values : bv_array (0 TO (max_redun_level - 1));

--
-- These signals will become ports when I get around to it
--
    SIGNAL load_timer : t_wlogic := f1;
    SIGNAL new_timer_value : timer_range;

--
-- Here are the components used in the architecture
--

COMPONENT voting_subsystem

   GENERIC
      (
          voter_delay: TIME := 1 ns;
          unan_delay: TIME := 1 ns;
          syndrome_delay: TIME := 1 ns
      );
   PORT
      (
```

```
        is_flow_control: IN  BIT;
    presence : IN presence_type;
        redun_level: IN  redun_level_type;
        unan: OUT  BIT_VECTOR(7 DOWNTO 0);
        vote_result: OUT  BIT_VECTOR(7 DOWNTO 0);
        d: IN  BIT_VECTOR(7 DOWNTO 0);
        c: IN  BIT_VECTOR(7 DOWNTO 0);
        b: IN  BIT_VECTOR(7 DOWNTO 0);
        a: IN  BIT_VECTOR(7 DOWNTO 0);
        d_syndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
        c_syndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
        b_syndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
        a_syndrome: OUT  BIT_VECTOR(7 DOWNTO 0)
    );

END COMPONENT;

COMPONENT timeout_subsystem

   GENERIC
      (
         do_CMTO: BOOLEAN := FALSE
      );
   PORT
      (
         vid_is_simplex: IN  BOOLEAN;
         clear_timeouts: IN  BOOLEAN;
         check_done: OUT  BOOLEAN;
         ct_to_value: IN  timeout_type;
         timer_value: OUT  timer_range;
         load_timer: IN  t_wlogic;
         new_timer_value: IN  timer_range;
         ibnf_unan: IN  BOOLEAN;
         ibnf: IN  flow_control_type;
         ibnf_syndrome_out: OUT  presence_type;
         clear_ibnf: OUT  BOOLEAN;
         start_to_check: IN  BOOLEAN;
         obne_syndrome_out: OUT  presence_type;
         obne_unan: IN  BOOLEAN;
         clear_obne: OUT  BOOLEAN;
         obne: IN  flow_control_type;
         source_vid: IN  vid_type;
         clock: IN  t_wlogic
      );

END COMPONENT;

BEGIN

--*************************************************************************
-- controller PROCESS : implements the voter sub-controller. It reads in
-- SERP entries using the pid_to_vid_table, sends them to the voter,
-- collects the syndromes, and writes the voted results to the voted
-- SERP memory.
-------------------------------------------------------------------------

  controller : PROCESS(clock,start_voting)

   VARIABLE temp_dpram_address : address_type;
   VARIABLE vids_voted : INTEGER := 0;

   VARIABLE current_ptov_entry : pid_to_vid_entry_type;
--
-- num_members is redun_level converted to a number, used as a loop control index
```

```
--
    VARIABLE num_members : INTEGER;

--
-- current_member is the index into the members array of the current_ptov_entry
--
    VARIABLE current_member : INTEGER;

  BEGIN
    IF clock = f1 AND clock'EVENT THEN
      CASE vote_state IS

--
-- State rse0 is the idle state
--
        WHEN rse0 =>
          done_voting <= FALSE;
          ptov_rw <= z0;
          dpram_rw <= z0;
          dpram_address <= high_z_address;
          voted_serp_rw <= z0;
          vids_voted := 0;
          vote_state <= rse0;
          IF start_voting  THEN
       ptov_address <= vids_voted;
       ptov_rw <= f1;
       dpram_rw <= f1;
       vote_state <= rse1;
          END IF;

--************************************************************************
-- States rse1 to rse4 read in the ptov entry and then each SERP entry in
-- the VID
-- Be careful of the array bounds since current_member goes from 1 TO 4
-- while members_type goes from 0 TO 3.
------------------------------------------------------------------------

--
-- wait for the ptov data to appear
--
        WHEN rse1 =>
          vote_state <= rse2;

        WHEN rse2 =>
          current_ptov_entry := ptov_data;
          source_vid <= current_ptov_entry.vid;

          num_members := redun_to_int(current_ptov_entry.redun_level);
          timeout_value <= current_ptov_entry.timeout;
          presence <= current_ptov_entry.presence;
          vid_is_simplex <= (current_ptov_entry.redun_level = simplex);
         current_member := 0;
         vote_state <= rse3;

--************************************************************************
-- These next two states read each SERP entry in the VID into the
-- serp_array
------------------------------------------------------------------------

        WHEN rse3 =>
--
-- Check to see if redun_level serp entries have been read
--
          IF NOT (current_member = num_members) THEN
```

```
--
-- If not, then go on to read the serp entry from the dpram
--
            temp_dpram_address := serp_base +
           current_ptov_entry.members(current_member);
         dpram_address <= temp_dpram_address;
         vote_state <= rse4;
            ELSE
--
-- If so, then go on to the voting state
--
         vote_state <= v0;
            END IF;

--
-- wait for the serp data to appear
--
          WHEN rse4 =>
            vote_state <= rse5;

--
-- rse5 simply assigns the serp_data which appears on the data line to
-- vote_values
--

          WHEN rse5 =>
            vote_values(current_member) <= serp_data;
            current_member := current_member + 1;

            vote_state <= rse3;

--************************************************************************
-- End section to read in the serp entries
------------------------------------------------------------------------

--
-- v0 sends the OBNE and IBNF bits to the voter. Notice that the OBNE and
-- IBNF are the MSB and (MSB-1) bits of the byte sent to the voter.
-- NOTE : In the voter, ALL four entries are always converted and sent
-- to the voter. The redun_level signal tells the voter which inputs
-- to ignore.
--

         WHEN v0 =>
           FOR i IN 1 TO max_redun_level LOOP
            bit_vote_values(i - 1)(obne_bit_pos) <=
            convert_to_bits(vote_values(i - 1).obne);
            bit_vote_values(i - 1)(ibnf_bit_pos) <=
            convert_to_bits(vote_values(i - 1).ibnf);
           END LOOP;
           vote_state <= v1;

--
-- v1 assigns the voted obne and ibnf "registers" there values, starts
-- the timeout process, and sends the destination VID to the voter
--
         WHEN v1 =>
            convert_back(voted_data(obne_bit_pos),voted_obne);
            convert_back(voted_data(ibnf_bit_pos),voted_ibnf);
--
-- start timeout process
--

          start_to_check <= TRUE AFTER control_delay;
```

```
          FOR i IN 1 TO max_redun_level LOOP
        bit_vote_values(i-1) <=
            convert_to_bits(vote_values(i - 1).dest_vid);
          END LOOP;
          vote_state <= v2;

--
-- v2 converts the voted destination VID back and decides whether to continue
-- voting based on the results of the timeout calculation.
-- NOTE : I'm assuming timeout calculations take only one clock cycle.
-- This is unrealistic, but for now it will do. Change the states around so
-- that whenever the timeouts are done, the voter decides whether to go on
-- voting or not.
--
        WHEN v2 =>
           convert_back(voted_data,voted_dest_vid);
--
-- IF (majority + timeout) THEN continue voting ELSE stop
          IF voted_obne AND NOT(clear_obne) THEN
            vote_state <= v3; -- last state of the voter
          ELSE
        vote_state <= v7;
          END IF;

--
-- For now, don't vote the rest of the stuff
--
        WHEN v3 =>
          vote_state <= v7;


--
-- v7 assigns the intermediate values to the voted_serp_data port
--
        WHEN v7 =>
           voted_serp_data.obne <= voted_obne;
           voted_serp_data.ibnf <= voted_ibnf;
           voted_serp_data.vid_is_simplex <= vid_is_simplex;
           voted_serp_data.source_vid <= source_vid;
           voted_serp_data.dest_vid <= voted_dest_vid;
           voted_serp_data.class <= voted_class;
           voted_serp_data.obne_syndrome <= obne_syndrome;
           voted_serp_data.ibnf_syndrome <= ibnf_syndrome;
           voted_serp_data.sb_vote_syndrome <= overall_vote_syndrome;

           voted_serp_address <= current_ptov_entry.vid;
          voted_serp_rw <= f0 AFTER control_delay;
          vids_voted := vids_voted + 1;
          vote_state <= v8;

--
-- v9 checks to see if all vids have been voted. If so, then it signals that
-- voting is done and idles the voter subcontroller
--

        WHEN v8 =>
          voted_serp_rw <= f1 AFTER control_delay;
          IF vids_voted = num_vids THEN
        done_voting <= TRUE;
        vote_state <= rse0;
          ELSE
--
-- assert next ptov table address and start over
```

```
--
          ptov_address <= vids_voted;
       vote_state <= rsel;
          END IF;
--
-- Appease that damn syntax diety, again. He's a demanding bastage.
--
        WHEN OTHERS =>
          ASSERT FALSE REPORT "PUKE"; --  SEVERITY ANNOYANCE;


     END CASE;
   END IF;

  END PROCESS;

  voter : voting_subsystem
       PORT MAP
       (
          redun_level => redun_level,
    presence => presence,
          d => bit_vote_values(3),
          c => bit_vote_values(2),
          b => bit_vote_values(1),
          a => bit_vote_values(0),
    a_syndrome => a_syndrome,
    b_syndrome => b_syndrome,
    c_syndrome => c_syndrome,
    d_syndrome => d_syndrome,
          vote_result => voted_data,
          is_flow_control => is_flow_control,
    unan => unan
      );

  timeout : timeout_subsystem
   PORT MAP
   (
    ct_to_value => timeout_value,
    vid_is_simplex => vid_is_simplex,
    clear_timeouts => start_clear,
          load_timer => load_timer,
          new_timer_value => new_timer_value,
          ibnf_unan => convert_back(unan(ibnf_bit_pos)),
          ibnf => voted_ibnf,
          ibnf_syndrome_out => ibnf_syndrome,
          clear_ibnf => clear_ibnf,
          start_to_check => start_to_check,
    check_done => check_done,
          obne_syndrome_out => obne_syndrome,
          obne_unan => convert_back(unan(obne_bit_pos)),
          clear_obne => clear_obne,
          obne => voted_obne,
          source_vid => source_vid,
    timer_value => current_timer_value,
          clock => clock
   );

END vote_timeout_struct;

--*****************************************************************************
-- Voter and Timeout Structural Configuration
--
-- This file contains the CONFIGURATION for the structural vote_timeout
-- implementation.
```

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

CONFIGURATION cvote_timeout_struct OF vote_timeout IS

  FOR vote_timeout_struct

    FOR timeout : timeout_subsystem
    USE CONFIGURATION work.ctimeout_subsystem
    GENERIC MAP ( do_CMTO => FALSE);
    END FOR;

    FOR voter : voting_subsystem

       USE CONFIGURATION voters.cvoting_subsystem
       GENERIC MAP (       voter_delay => clock_period/4,
              syndrome_delay => clock_period/4,
              unan_delay => clock_period/4);
    END FOR;

  END FOR;

END cvote_timeout_struct;
```

10.7.2. Timeout Subsystem

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE score.voter_package.ALL;
USE score.address_package.ALL;
ENTITY timeout_subsystem IS

   GENERIC
      (
         do_CMTO: BOOLEAN := FALSE
      );
   PORT
      (
          vid_is_simplex: IN  BOOLEAN;
          clear_timeouts: IN  BOOLEAN;
          check_done: OUT  BOOLEAN;
          ct_to_value: IN  timeout_type;
          timer_value: OUT  timer_range;
          load_timer: IN  t_wlogic;
          new_timer_value: IN  timer_range;
          ibnf_unan: IN  BOOLEAN;
          ibnf: IN  flow_control_type;
          ibnf_syndrome_out: OUT  presence_type;
          clear_ibnf: OUT  BOOLEAN;
          start_to_check: IN  BOOLEAN;
          obne_syndrome_out: OUT  presence_type;
          obne_unan: IN  BOOLEAN;
          clear_obne: OUT  BOOLEAN;
          obne: IN  flow_control_type;
          source_vid: IN  vid_type;
          clock: IN  t_wlogic
      );

END timeout_subsystem;

LIBRARY SCORE;

USE std.std_logic.ALL;
ARCHITECTURE timeout_subsystem OF timeout_subsystem IS

   COMPONENT timer
      PORT
         (
            clock: IN  t_wlogic;
            input_value: IN  timer_range;
            timer_value: OUT  timer_range;
            load_timer: IN  t_wlogic
         );

   END COMPONENT;
   COMPONENT timeout_checker
      GENERIC
         (
            do_CMTO: BOOLEAN := FALSE;
            simplex_is_special: BOOLEAN := FALSE
         );
      PORT
```

```
      (
          clock: IN  t_wlogic;
          flow_control_bit: IN  flow_control_type;
          clear_flow_control: OUT  BOOLEAN;
          unan: IN  BOOLEAN;
          syndrome_out: OUT  presence_type;
          start_to_check: IN  BOOLEAN;
          to_memory_rw: OUT  t_wlogic;
          timeout_value: IN  timer_type;
          timer_value: IN  timer_range;
          check_done: OUT  BOOLEAN;
          ct_to_value: IN  timeout_type;
          vid_is_simplex: IN  BOOLEAN := FALSE;
          to_address: OUT  resolved_address;
          clear_timeouts: IN  BOOLEAN;
          timeout_out: OUT  timer_type
      );

END COMPONENT;
COMPONENT timeout_memory
   GENERIC
      (
          read_delay: TIME := clock_period/4
      );
   PORT
      (
          clock: IN  t_wlogic;
          address: IN  resolved_address;
          input: IN  timer_type;
          output: OUT  timer_type;
          read_write: IN  t_wlogic
      );

END COMPONENT;

FOR scoreboard_timer:timer
   USE CONFIGURATION SCORE.ctimer_behavior;

FOR obne_to_checker:timeout_checker
   USE CONFIGURATION SCORE.ctimeout_checker_behavior
   GENERIC MAP (
      do_CMTO => FALSE,
      simplex_is_special => FALSE );

FOR ibnf_to_checker:timeout_checker
   USE CONFIGURATION SCORE.ctimeout_checker_behavior
   GENERIC MAP (
      do_CMTO => FALSE,
      simplex_is_special => FALSE );

FOR obne_to_memory:timeout_memory
   USE CONFIGURATION SCORE.ctimeout_memory_behavior
   GENERIC MAP (
      read_delay => clock_period/4 );

FOR ibnf_to_memory:timeout_memory
   USE CONFIGURATION SCORE.ctimeout_memory_behavior
   GENERIC MAP (
      read_delay => clock_period/4 );
SIGNAL SGNL000021:  t_wlogic;
SIGNAL SGNL000051:  timer_type;
SIGNAL SGNL000057:  timer_type;
SIGNAL SGNL000017:  t_wlogic;
SIGNAL SGNL000049:  timer_type;
```

```
   SIGNAL SGNL000056:   timer_type;
   SIGNAL feedback0:    timer_range;


BEGIN
   timer_value <= feedback0;


   scoreboard_timer: timer
      PORT MAP (
          load_timer => load_timer,
          timer_value => feedback0,
          input_value => new_timer_value,
          clock => clock );

   obne_to_checker: timeout_checker
      PORT MAP (
          timeout_out => SGNL000056,
          clear_timeouts => clear_timeouts,
          to_address => OPEN,
          vid_is_simplex => vid_is_simplex,
          ct_to_value => ct_to_value,
          check_done => OPEN,
          timer_value => feedback0,
          timeout_value => SGNL000049,
          to_memory_rw => SGNL000017,
          start_to_check => start_to_check,
          syndrome_out => obne_syndrome_out,
          unan => obne_unan,
          clear_flow_control => clear_obne,
          flow_control_bit => obne,
          clock => clock );

   ibnf_to_checker: timeout_checker
      PORT MAP (
          timeout_out => SGNL000057,
          clear_timeouts => clear_timeouts,
          to_address => OPEN,
          vid_is_simplex => vid_is_simplex,
          ct_to_value => ct_to_value,
          check_done => check_done,
          timer_value => feedback0,
          timeout_value => SGNL000051,
          to_memory_rw => SGNL000021,
          start_to_check => start_to_check,
          syndrome_out => ibnf_syndrome_out,
          unan => ibnf_unan,
          clear_flow_control => clear_ibnf,
          flow_control_bit => ibnf,
          clock => clock );

   obne_to_memory: timeout_memory
      PORT MAP (
          read_write => SGNL000017,
          output => SGNL000049,
          input => SGNL000056,
          address => source_vid,
          clock => clock );

   ibnf_to_memory: timeout_memory
      PORT MAP (
          read_write => SGNL000021,
          output => SGNL000051,
          input => SGNL000057,
```

```
          address => source_vid,
          clock => clock );

END timeout_subsystem;

--*****************************************************************************
-- Timeout Subsystem Structural Configuration
--
-- This file contains the CONFIGURATION for the structural
-- timeout_subsystem implementation.
-------------------------------------------------------------------------------


CONFIGURATION ctimeout_subsystem OF timeout_subsystem IS

   FOR timeout_subsystem

      FOR scoreboard_timer:timer
         USE CONFIGURATION score.ctimer_behavior;
      END FOR;

      FOR obne_to_checker:timeout_checker
         USE CONFIGURATION score.ctimeout_checker_behavior
      GENERIC MAP (
         do_CMTO => FALSE,
         simplex_is_special => FALSE );
      END FOR;

--
-- A simplex is treated differently for IBNF than for OBNE, so make the
-- GENERIC the proper value
--
      FOR ibnf_to_checker:timeout_checker
         USE CONFIGURATION score.ctimeout_checker_behavior
      GENERIC MAP (
         do_CMTO => FALSE,
         simplex_is_special => TRUE );
      END FOR;

      FOR obne_to_memory:timeout_memory
         USE CONFIGURATION score.ctimeout_memory_behavior
      GENERIC MAP (
         read_delay => clock_period/4 );
      END FOR;

      FOR ibnf_to_memory:timeout_memory
         USE CONFIGURATION score.ctimeout_memory_behavior
      GENERIC MAP (
         read_delay => clock_period/4 );
      END FOR;

   END FOR;

END ctimeout_subsystem;
```

10.7.3. Timeout Checker

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE score.address_package.ALL;
USE score.voter_package.ALL;
ENTITY timeout_checker IS

   GENERIC
      (
         do_CMTO: BOOLEAN := FALSE;
          simplex_is_special: BOOLEAN := FALSE
      );
   PORT
      (
          timeout_out: OUT  timer_type;
          clear_timeouts: IN  BOOLEAN;
          to_address: OUT  resolved_address;
          vid_is_simplex: IN  BOOLEAN := FALSE;
          ct_to_value: IN  timeout_type;
          check_done: OUT  BOOLEAN;
          timer_value: IN  timer_range;
          timeout_value: IN  timer_type;
          to_memory_rw: OUT  t_wlogic;
          start_to_check: IN  BOOLEAN;
          syndrome_out: OUT  presence_type;
          unan: IN  BOOLEAN;
          clear_flow_control: OUT  BOOLEAN;
          flow_control_bit: IN  flow_control_type;
          clock: IN  t_wlogic
      );

END timeout_checker;

--*************************************************************************
-- Timeout Checker Behavioral Architecture
--
-- This ARCHITECTURE contains the behavioral implementation of the
-- timeout checker. The generic do_CMTO flags wehter to perform a
-- Common Mode Timeout. This feature is currently not implemented,
-- but the hook is there.
-------------------------------------------------------------------------

ARCHITECTURE timeout_checker_behavior OF timeout_checker IS

   TYPE checker_state_type IS (c0,c1,c2,c3,c4);
   SIGNAL checker_state : checker_state_type := c0;

   SIGNAL difference : timer_range;

BEGIN

  controller : PROCESS (clock,start_to_check)

  BEGIN
    IF clock = f1 AND clock'EVENT THEN
      CASE checker_state IS
```

```
    WHEN c0 =>
      IF start_to_check THEN
        checker_state <= c1;
      ELSE
        checker_state <= c0;
      END IF;

    WHEN c1 =>
      IF (flow_control_bit AND NOT(unan)) OR vid_is_simplex THEN
         difference <= abs(timer_value - timeout_value.value);
        checker_state <= c2;
      ELSE
--
-- Place Common Mode Timeout code here
--
        checker_state <= c4;
      END IF;

    WHEN c2 =>
      IF difference > ct_to_value THEN
--
-- The timeout period has expired
--
        clear_flow_control <= FALSE AFTER control_delay;
        checker_state <= c3;
      ELSE
        clear_flow_control <= TRUE AFTER control_delay;
        checker_state <= c4;
      END IF;

    WHEN c3 =>
      checker_state <= c4;
    WHEN c4 =>
      check_done <= TRUE AFTER control_delay;
      checker_state <= c0;

     END CASE;
    END IF;
  END PROCESS;

END timeout_checker_behavior;


CONFIGURATION ctimeout_checker_behavior OF timeout_checker IS
  FOR timeout_checker_behavior
  END FOR;
END ctimeout_checker_behavior;
```

10.7.4. Timeout Memory

```
LIBRARY score;

USE score.voter_package.ALL;
USE std.std_logic.ALL;
USE std.std_cmos.ALL;
USE score.address_package.ALL;
USE score.scoreboard_package.ALL;
ENTITY timeout_memory IS

   GENERIC
      (
          read_delay: TIME := clock_period/4
      );
   PORT
      (
          read_write: IN  t_wlogic;
          output: OUT  timer_type;
          input: IN  timer_type;
          address: IN  resolved_address;
          clock: IN  t_wlogic
      );

END timeout_memory;

--*****************************************************************************
-- Timeout Memory Behavioral ARCHITECTURE
--
-- This file contains the behavioral ARCHITECTURE for the timeout
-- memory.
-------------------------------------------------------------------------------

ARCHITECTURE timeout_memory_behavior OF timeout_memory IS

BEGIN

  behavior : PROCESS (clock,read_write,address)
   VARIABLE memory : timeout_memory_type(mem_base TO max_vid);
  BEGIN
   IF clock = f1 AND clock'EVENT THEN
     IF read_write = f0 THEN
       memory(address) := input;
     ELSE
       output <= memory(address) AFTER read_delay;
     END IF;
   END IF;
  END PROCESS;
END;


CONFIGURATION ctimeout_memory_behavior OF timeout_memory IS
  FOR timeout_memory_behavior
  END FOR;
END ctimeout_memory_behavior;
```

10.7.5. Timer

```
LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE score.voter_package.ALL;
ENTITY timer IS

   PORT
      (
         load_timer: IN  t_wlogic;
         timer_value: OUT  timer_range;
         input_value: IN  timer_range;
         clock: IN  t_wlogic
      );

END timer;

--*************************************************************************
-- Timer Behavioral Architecture
--
-- This ARCHITECTURE contains the behavioral implementation of the
-- scoreboard's internal timer. Ideally, a selected signal assignment
-- statement should be used. However, because the current version
-- does not support BUFFER ports, a PROCESS must be used instead.
-------------------------------------------------------------------------

ARCHITECTURE timer_behavior OF timer IS

BEGIN

  yo : PROCESS(clock)

--
-- this variable is required because the OUT port timer_value
-- cannot be read
--
    VARIABLE temp_timer_value : timer_range := 0;

  BEGIN

    IF clock = f1 AND clock'EVENT THEN
      IF NOT (temp_timer_value = max_timer_value) THEN
        temp_timer_value := temp_timer_value + 1;
      ELSE
        temp_timer_value := init_timer_value;
      END IF;

      IF load_timer = f1 THEN
        temp_timer_value := input_value;
      END IF;

      timer_value <= temp_timer_value;
    END IF;

  END PROCESS;
```

```
END timer_behavior;


CONFIGURATION ctimer_behavior OF timer IS
  FOR timer_behavior
  END FOR;
END ctimer_behavior;
```

10.7.6. Voting Subsystem

```
LIBRARY voters;

LIBRARY score;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.voter_package.ALL;
ENTITY voting_subsystem IS

   GENERIC
      (
         voter_delay: TIME := 1 ns;
         unan_delay: TIME := 1 ns;
         syndrome_delay: TIME := 1 ns
      );
   PORT
      (
         presence: IN  presence_type;
         d_syndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
         c_syndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
         b_syndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
         a_syndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
         is_flow_control: IN  BIT;
         redun_level: IN  redun_level_type;
         unan: OUT  BIT_VECTOR(7 DOWNTO 0);
         vote_result: OUT  BIT_VECTOR(7 DOWNTO 0);
         d: IN  BIT_VECTOR(7 DOWNTO 0);
         c: IN  BIT_VECTOR(7 DOWNTO 0);
         b: IN  BIT_VECTOR(7 DOWNTO 0);
         a: IN  BIT_VECTOR(7 DOWNTO 0)
      );

END voting_subsystem;

LIBRARY VOTERS;

USE std.std_logic.ALL;
ARCHITECTURE voting_subsystem OF voting_subsystem IS

   COMPONENT eight_bit_voter
      GENERIC
         (
            voter_delay: TIME := 1 ns
         );
      PORT
         (
            is_flow_control: IN  BIT;
            result: OUT  BIT_VECTOR(7 DOWNTO 0);
            a: IN  BIT_VECTOR(7 DOWNTO 0);
            b: IN  BIT_VECTOR(7 DOWNTO 0);
            c: IN  BIT_VECTOR(7 DOWNTO 0);
            d: IN  BIT_VECTOR(7 DOWNTO 0);
            redun_level: IN  redun_level_type
         );

   END COMPONENT;
   COMPONENT eight_bit_syndrome
      GENERIC
         (
```

```
            syndrome_delay: TIME := 1 ns
        );
    PORT
        (
            a: IN   BIT_VECTOR(7 DOWNTO 0);
            b: IN   BIT_VECTOR(7 DOWNTO 0);
            c: IN   BIT_VECTOR(7 DOWNTO 0);
            d: IN   BIT_VECTOR(7 DOWNTO 0);
            asyndrome: OUT   BIT_VECTOR(7 DOWNTO 0);
            bsyndrome: OUT   BIT_VECTOR(7 DOWNTO 0);
            csyndrome: OUT   BIT_VECTOR(7 DOWNTO 0);
            dsyndrome: OUT   BIT_VECTOR(7 DOWNTO 0);
            vote_result: IN   BIT_VECTOR(7 DOWNTO 0);
            presence: IN   presence_type
        );

END COMPONENT;
COMPONENT eight_bit_unan
    GENERIC
        (
            unan_delay: TIME := 1 ns
        );
    PORT
        (
            a: IN   BIT_VECTOR(7 DOWNTO 0);
            b: IN   BIT_VECTOR(7 DOWNTO 0);
            c: IN   BIT_VECTOR(7 DOWNTO 0);
            d: IN   BIT_VECTOR(7 DOWNTO 0);
            vote_result: IN   BIT_VECTOR(7 DOWNTO 0);
            unan: OUT   BIT_VECTOR(7 DOWNTO 0);
            redun_level: IN   redun_level_type
        );

END COMPONENT;

FOR voter:eight_bit_voter
    USE CONFIGURATION  VOTERS.cfull_voter_behavior
    GENERIC MAP (
        voter_delay => 1 ns );

FOR syndrome_generator:eight_bit_syndrome
    USE CONFIGURATION  VOTERS.cfull_syndrome_behavior
    GENERIC MAP (
        syndrome_delay => 1 ns );

FOR unan_generator:eight_bit_unan
    USE CONFIGURATION  VOTERS.ceight_bit_unan_behavior
    GENERIC MAP (
        unan_delay => 1 ns );
SIGNAL feedback0:   BIT_VECTOR(7 DOWNTO 0);


BEGIN
    vote_result <= feedback0;


    voter: eight_bit_voter
        PORT MAP (
            redun_level => redun_level,
            d => d,
            c => c,
            b => b,
            a => a,
            result => feedback0,
```

```
            is_flow_control => is_flow_control );

    syndrome_generator: eight_bit_syndrome
       PORT MAP (
          presence => presence,
          vote_result => feedback0,
          dsyndrome => d_syndrome,
          csyndrome => c_syndrome,
          bsyndrome => b_syndrome,
          asyndrome => a_syndrome,
          d => d,
          c => c,
          b => b,
          a => a );

    unan_generator: eight_bit_unan
       PORT MAP (
          redun_level => redun_level,
          unan => unan,
          vote_result => feedback0,
          d => d,
          c => c,
          b => b,
          a => a );

END voting_subsystem;

CONFIGURATION cvoting_subsystem OF voting_subsystem IS

   FOR voting_subsystem

      FOR voter:eight_bit_voter
         USE CONFIGURATION work.cfull_voter_behavior
      GENERIC MAP (
         voter_delay => 1 ns );
      END FOR;

      FOR syndrome_generator:eight_bit_syndrome
         USE CONFIGURATION work.cfull_syndrome_behavior
      GENERIC MAP (
         syndrome_delay => 1 ns );
      END FOR;

      FOR unan_generator:eight_bit_unan
         USE CONFIGURATION work.ceight_bit_unan_behavior
      GENERIC MAP (
         unan_delay => 1 ns );
      END FOR;

   END FOR;

END cvoting_subsystem;
```

10.7.7. One Bit Voter

```
LIBRARY voters;

LIBRARY score;

USE std.std_logic.ALL;
USE std.std_cmos.ALL;
USE score.scoreboard_package.ALL;
ENTITY one_bit_voter IS

   GENERIC
      (
          voter_delay: TIME := 1 ns
      );
   PORT
      (
          is_flow_control: IN  BIT;
          redun_level: IN  redun_level_type;
          result: OUT BIT;
          d: IN  BIT;
          c: IN  BIT;
          b: IN  BIT;
          a: IN  BIT
      );

END one_bit_voter;

--*************************************************************************
-- One Bit Voter Behavioral description
--    This file contains the behavioral description of a one bit voter.
-- It uses a selected signal assignment statement to vote based on
-- the redundancy level.
-------------------------------------------------------------------------

ARCHITECTURE one_bit_voter_behavior OF one_bit_voter IS

BEGIN

  voter_process : PROCESS(a,b,c,d,redun_level,is_flow_control)

   VARIABLE flow_quad_result,data_quad_result : BIT := '0';
   VARIABLE quad_result,triplex_result,simplex_result : BIT := '0';

  BEGIN

   ASSERT NOT (redun_level = zero)
   REPORT "redun_level in voter is zero!"
   SEVERITY ERROR;

   simplex_result := a;

--
-- For triplex voting, the 'd' input can be ignored since it will not have
-- valid data on it
--
   triplex_result := (a AND b) OR (a AND c) OR (b AND c);

   flow_quad_result :=
       (a AND b AND c) OR (a AND b AND d) OR
       (a AND c AND d) OR (b AND c AND d);
```

```
    data_quad_result :=
        (a AND b) OR (a AND c) OR (a AND d) OR
        (b AND c) OR (b AND d) OR (c AND d);

    IF is_flow_control = '1' THEN
      quad_result := flow_quad_result;
    ELSE
      quad_result := data_quad_result;
    END IF;

    CASE redun_level IS
      WHEN zero => result <= '0' AFTER voter_delay;
      WHEN simplex => result <= simplex_result AFTER voter_delay;
      WHEN triplex => result <= triplex_result AFTER voter_delay;
      WHEN quad => result <= quad_result AFTER voter_delay;
    END CASE;

  END PROCESS;

END one_bit_voter_behavior;


CONFIGURATION cone_bit_voter_behavior OF one_bit_voter IS
  FOR one_bit_voter_behavior
  END FOR;
END cone_bit_voter_behavior;
```

10.7.8. One Bit Unanimity Generator

```
LIBRARY score;

LIBRARY voters;

USE std.std_logic.ALL;
USE std.std_cmos.ALL;
USE score.scoreboard_package.ALL;
USE score.voter_package.ALL;
ENTITY one_bit_unan IS

   GENERIC
      (
         unan_delay: TIME := 1 ns
      );
   PORT
      (
         redun_level: IN  redun_level_type;
         vote_result: IN  BIT;
         unan: OUT  BIT;
         d: IN  BIT;
         c: IN  BIT;
         b: IN  BIT;
         a: IN  BIT
      );

END one_bit_unan;

--*************************************************************************
-- One bit unanimity checker
-- This file contains the behavioral description of a single bit
-- unanimity checker.
-------------------------------------------------------------------------

ARCHITECTURE one_bit_unan_behavior OF one_bit_unan IS

BEGIN

  unan_checker: PROCESS (a,b,c,d,vote_result,redun_level)

   VARIABLE quad_result,triplex_result,simplex_result : BOOLEAN := FALSE;

  BEGIN

   simplex_result :=       TRUE;

   triplex_result :=       (a   = vote_result) AND
             (b   = vote_result) AND
             (c   = vote_result);

   quad_result :=          (a   = vote_result) AND
             (b   = vote_result) AND
             (c   = vote_result) AND
             (d   = vote_result);

   CASE redun_level IS
     WHEN zero => unan <= '0' AFTER unan_delay;
     WHEN simplex =>
      unan <= convert_to_bits(simplex_result) AFTER unan_delay;
     WHEN triplex =>
```

```
      unan <= convert_to_bits(triplex_result) AFTER unan_delay;
    WHEN quad =>
      unan <= convert_to_bits(quad_result) AFTER unan_delay;
  END CASE;

  END PROCESS;


END one_bit_unan_behavior;


CONFIGURATION cone_bit_unan_behavior OF one_bit_unan IS
  FOR one_bit_unan_behavior
  END FOR;
END cone_bit_unan_behavior;
```

10.7.9. One Bit Syndrome Accumulator

```
LIBRARY score;

LIBRARY voters;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE score.voter_package.ALL;
ENTITY one_bit_syndrome IS

   GENERIC
      (
          syndrome_delay: TIME := 1 ns
      );
   PORT
      (
          presence: IN  presence_type;
          vote_result: IN  BIT;
          dsyndrome: OUT  BIT;
          csyndrome: OUT  BIT;
          bsyndrome: OUT  BIT;
          asyndrome: OUT  BIT;
          d: IN  BIT;
          c: IN  BIT;
          b: IN  BIT;
          a: IN  BIT
      );

END one_bit_syndrome;

--*************************************************************************
-- One Bit Voter Behavioral description
--   This file contains the behavioral description of a one bit voter.
-- It uses a selected signal assignment statement to vote based on
-- the redundancy level.
-------------------------------------------------------------------------

ARCHITECTURE one_bit_syndrome_behavior OF one_bit_syndrome IS

BEGIN


END one_bit_syndrome_behavior;


CONFIGURATION cone_bit_syndrome_behavior OF one_bit_syndrome IS
  FOR one_bit_syndrome_behavior
  END FOR;
END cone_bit_syndrome_behavior;
```

10.7.10. Eight Bit Voter

```
LIBRARY score;

LIBRARY voters;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE score.voter_package.ALL;
USE std.std_cmos.ALL;
ENTITY eight_bit_voter IS

   GENERIC
      (
         voter_delay: TIME := 1 ns
      );
   PORT
      (
         redun_level: IN  redun_level_type;
         d: IN  BIT_VECTOR(7 DOWNTO 0);
         c: IN  BIT_VECTOR(7 DOWNTO 0);
         b: IN  BIT_VECTOR(7 DOWNTO 0);
         a: IN  BIT_VECTOR(7 DOWNTO 0);
         result: OUT  BIT_VECTOR(7 DOWNTO 0);
         is_flow_control: IN  BIT
      );

END eight_bit_voter;

--*********************************************************************
-- Eight bit voter
-- This file contains the structural description for an eight bit
-- voter. A generate statement creates and maps the 8 one bit voters.
---------------------------------------------------------------------

ARCHITECTURE eight_bit_voter_behavior OF eight_bit_voter IS

  COMPONENT one_bit_voter
    PORT
      (
         is_flow_control: IN  BIT;
         redun_level: IN  redun_level_type;
         result: OUT  BIT;
         d: IN  BIT;
         c: IN  BIT;
         b: IN  BIT;
         a: IN  BIT
      );
  END COMPONENT;

BEGIN

  voter_gen : FOR i IN result'RANGE GENERATE

   slice : one_bit_voter PORT MAP
          (
             is_flow_control => is_flow_control,
             redun_level => redun_level,
             result => result(i),
             d => d(i),
             c => c(i),
```

```
            b => b(i),
            a => a(i)
          );

  END GENERATE voter_gen;

END eight_bit_voter_behavior;


CONFIGURATION cfull_voter_behavior OF eight_bit_voter IS
  FOR eight_bit_voter_behavior
    FOR voter_gen
    FOR slice : one_bit_voter
      USE CONFIGURATION voters.cone_bit_voter_behavior
      GENERIC MAP (voter_delay => voter_delay);
    END FOR;
     END FOR;
  END FOR;
END cfull_voter_behavior;
```

10.7.11. Eight Bit Unanimity Generator

```
LIBRARY score;

LIBRARY voters;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
ENTITY eight_bit_unan IS

   GENERIC
      (
         unan_delay: TIME := 1 ns
      );
   PORT
      (
         redun_level: IN  redun_level_type;
         unan: OUT  BIT_VECTOR(7 DOWNTO 0);
         vote_result: IN  BIT_VECTOR(7 DOWNTO 0);
         d: IN  BIT_VECTOR(7 DOWNTO 0);
         c: IN  BIT_VECTOR(7 DOWNTO 0);
         b: IN  BIT_VECTOR(7 DOWNTO 0);
         a: IN  BIT_VECTOR(7 DOWNTO 0)
      );

END eight_bit_unan;

--*************************************************************************
-- Eight unan checker
-- This file contains the structural description for an eight bit
-- unanimity checker.
-------------------------------------------------------------------------

ARCHITECTURE eight_bit_unan_behavior OF eight_bit_unan IS

  COMPONENT one_bit_unan
   PORT
      (
         redun_level: IN  redun_level_type;
         vote_result: IN  BIT;
         unan: OUT  BIT;
         d: IN  BIT;
         c: IN  BIT;
         b: IN  BIT;
         a: IN  BIT
      );
  END COMPONENT;

BEGIN

  unan_gen : FOR i IN unan'RANGE GENERATE

   slice : one_bit_unan PORT MAP
          (
          vote_result => vote_result(i),
             redun_level => redun_level,
             unan => unan(i),
             d => d(i),
             c => c(i),
             b => b(i),
             a => a(i)
```

```
        );

  END GENERATE unan_gen;

END eight_bit_unan_behavior;

CONFIGURATION ceight_bit_unan_behavior OF eight_bit_unan IS
  FOR eight_bit_unan_behavior
    FOR unan_gen
   FOR slice : one_bit_unan
     USE CONFIGURATION voters.cone_bit_unan_behavior
     GENERIC MAP (unan_delay => unan_delay);
   END FOR;
    END FOR;
  END FOR;
END ceight_bit_unan_behavior;
```

10.7.12. Eight Bit Syndrome Accumulator

```
LIBRARY score;

LIBRARY voters;

USE std.std_logic.ALL;
USE score.scoreboard_package.ALL;
USE std.std_cmos.ALL;
USE score.voter_package.ALL;
ENTITY eight_bit_syndrome IS

   GENERIC
      (
          syndrome_delay: TIME := 1 ns
      );
   PORT
      (
          presence: IN  presence_type;
          vote_result: IN  BIT_VECTOR(7 DOWNTO 0);
          dsyndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
          csyndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
          bsyndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
          asyndrome: OUT  BIT_VECTOR(7 DOWNTO 0);
         d: IN  BIT_VECTOR(7 DOWNTO 0);
         c: IN  BIT_VECTOR(7 DOWNTO 0);
         b: IN  BIT_VECTOR(7 DOWNTO 0);
         a: IN  BIT_VECTOR(7 DOWNTO 0)
      );

END eight_bit_syndrome;

--*************************************************************************
-- Eight bit syndrome
-- This file contains the structural description for an eight bit
-- syndrome. A generate statement creates and maps the 8 one bit syndromes.
-------------------------------------------------------------------------

ARCHITECTURE eight_bit_syndrome_behavior OF eight_bit_syndrome IS

  COMPONENT one_bit_syndrome
    PORT
      (
          presence: IN  presence_type;
          vote_result: IN  BIT;
          dsyndrome: OUT  BIT;
          csyndrome: OUT  BIT;
          bsyndrome: OUT  BIT;
          asyndrome: OUT  BIT;
         d: IN  BIT;
         c: IN  BIT;
         b: IN  BIT;
         a: IN  BIT
      );
  END COMPONENT;

BEGIN

   syndrome_gen : FOR i IN vote_result'RANGE GENERATE

    slice : one_bit_syndrome PORT MAP
```

```
            (
            presence => presence,
            vote_result => vote_result(i),
            dsyndrome => dsyndrome(i),
            csyndrome => csyndrome(i),
            bsyndrome => bsyndrome(i),
            asyndrome => asyndrome(i),
                d => d(i),
                c => c(i),
                b => b(i),
                a => a(i)
            );

  END GENERATE syndrome_gen;

END eight_bit_syndrome_behavior;

CONFIGURATION cfull_syndrome_behavior OF eight_bit_syndrome IS
  FOR eight_bit_syndrome_behavior
    FOR syndrome_gen
    FOR slice : one_bit_syndrome
      USE CONFIGURATION voters.cone_bit_syndrome_behavior
      GENERIC MAP (syndrome_delay => syndrome_delay);
    END FOR;
     END FOR;
  END FOR;
END cfull_syndrome_behavior;
```

10.8. C Test Vector Generator

This appendix contains the complete C source code for the scoreboard test vector generator.

10.8.1. File config.h

This file contains global configuration information such as the location within the SERP of the OBNE and IBNF bits.

```
/*********************************************************************/
/* Define configuration information                                  */
/* changing these defines will change such things as where the obne */
/* bit is located and how many bytes per serp_entry                  */
/*********************************************************************/

/* bytes per entry in the voted_serp array */
/* changing this also requires that the write_result procedure be    */
/* changed as well (it's in the vote.c source file                   */
#define VS 7
/* these definitions all affect the voted serp array. Be careful!!   */
#define VS_to_loc 0
#define VS_obne_loc 1
#define VS_ibnf_loc 1
#define VS_class_loc 1
#define VS_dvid_loc 2
#define VS_obne_syn_loc 3
#define VS_ibnf_syn_loc 4
#define VS_timer_loc 5
#define VS_init_timer_loc 6
#define VS_processed_bit_loc 1

/* define masks for the voted serp array */
#define VS_obne_mask 0x80
#define VS_ibnf_mask 0x40
#define VS_class_mask 0x1f
#define VS_obne_syn_mask 0x0f
#define VS_ibnf_syn_mask 0x0f
#define VS_processed_bit_mask 0x20

/* These #defines correspond to the locations of all the bytes in    */
/* the SERP and CT                                                   */
/* NOTE that the first entry is numbered 0                           */

/* CT related locations */
#define Bytes_per_CT_entry 8
#define Redun_loc 1
#define Presence_loc 1
#define Base_pe_loc 2  /* location of the first PE in a CT entry */
#define To_loc 6

/* SERP related locations */
#define Bytes_per_SERP_entry 4
#define Dvid_loc 1
#define Obne_loc 0
#define Ibnf_loc 0
#define Class_loc 0
```

```
/* System configuration definitions */
#define Max_redun_level 4
#define Num_ne 5
#define Pe_per_ne 8
#define Num_vids 256
#define Max_vid 254

/* These masks correspond to the location within a byte. They don't */
/* need to be changed unless the bit locations are changed.         */

#define obne_mask 0x80
#define ibnf_mask 0x40
#define class_mask 0x1f
#define redun_level_mask 0x7
#define presence_shift 3
#define ne_mask 0x07;
#define pe_shift 3

/* End configuration information */
/*********************************************************************/
```

10.8.2. File sbdefs.h

This file contains global definitions and variables.

```
/* Scoreboard simulation program
   by
   Dennis Morton
   3 Jan 1991

   revision 2.2 (everything parameterized, file output added)
*/

#include "config.h"

/*********************************************************************/
/* This header contains globals used by the simulation               */
/*********************************************************************/

#define True 1
#define False 0
#define Prob_fault 1

typedef int Boolean;
typedef int Byte;
typedef int Bit;
typedef Byte Serp_type[Pe_per_ne * Num_ne * Bytes_per_SERP_entry];

struct message_struct
{
  Byte source_vid;
  Byte dest_vid;
  Byte sources[Max_redun_level];
  Byte dests[Max_redun_level];
  Byte obne_to;
  Byte ibnf_to;
  Byte timer_value;
  Byte itv;            /* initial timer value */
  Byte ex_class;
  Byte timestamp;
  Byte vote_mask;
  Byte size;
};

/*****************************************************************/
/* this structure is used to generate serps. serps_to_do tells */
/* how many times to include that entry as a potential message */
/* in the serp. serps_done tells how many serps have been sent */
/* with that entry. When these two become equal, source_vid is */
/* added to the free_sources array. On the next serp, new      */
/* parameters will be generated for source_vid.                */
/*****************************************************************/

struct serp_source_struct
{
  Byte source_vid;
  Byte dest_vid;
  /* srd = source redundancy level, drd = destination redun level */
  Byte srd;
  Byte drd;
  Byte obne_to;
  Byte ibnf_to;
```

```
  Byte to_value;
  int serps_done;
} gserp_source[Num_vids];

/* correct message array */
struct message_struct gcmessage[Pe_per_ne * Num_ne];

/* messages found by simulation */
struct message_struct gmessage[Pe_per_ne * Num_ne];

Serp_type serp;
Byte gvoted_serp[Num_vids * VS];

/*****************************************************/
/* define lookup tables used by the various modules */
/*****************************************************/
int to[Num_vids];
Byte ct[Num_vids * Bytes_per_CT_entry];
Byte gvids_used[Num_vids];
Byte ptov_table[Pe_per_ne * Num_ne * Bytes_per_SERP_entry];
Boolean gfree_sources[Num_vids];
Boolean gfree_dests[Num_vids];
/*****************************************************/

/* to_clock is an integer which emulates a clock */
int to_clock;

/* num_vids contains the number of defined vids currently in the ct */
int gnum_vids;

/* this variable controls the verbosity of the output */
/* 1 = keep it simple */
/* 2 = intermediate but useful for bugs */
/* 3 = inundate me with info */
int debug_level;

/* This flag decides whether to generate faults or not */
Boolean generate_faults;

/* This variable determines how many total rounds to perform */
int num_rounds;

/* this array is used to generate the presence bits */
Byte gpmasks[Max_redun_level];
```

10.8.3. File ct.c

This file contains functions to generate and check the CT.

```
#include "sbdefs.h"

/************************ file ct.c ****************************
* This file contains the source code which generates and checks *
* the configuration table. It also clears all the important     *
* variables and arrays.                                         *
*****************************************************************/

/* These numbers affect how likely each redun level will be      */
/* generated                                                     */
#define Prob_simplex 5
#define Prob_triplex 11

/************************ get_table ****************************
* get_table generates the PID to VID translation table used to  *
* feed the voter one VID at a time                              *
*****************************************************************/

void get_table()
{
  int i,j,serp_place;
  Byte redun_level;
  /* calculate the pid to vid translation table which will allow */
  /*   the voter to be fed one vid at a time                     */

  serp_place = 0;
  for(i = 0; i <= (gnum_vids - 1); i++)
  {
    redun_level = ct[gvids_used[i] *
                  Bytes_per_CT_entry + Redun_loc] & redun_level_mask;
    ptov_table[serp_place] = redun_level;
    serp_place += 1;
    for(j = 0; j <= (redun_level-1); j++)
    {
      ptov_table[serp_place] = unpack(ct[gvids_used[i] *
                                  Bytes_per_CT_entry +
                                  Base_pe_loc + j]);
      serp_place += 1;
    }
  }
}

/************************ check_ct ****************************
* ensures that a single pe is not a member of more than 1 VID   *
*****************************************************************/

void check_ct()
{
  int i,j,vid_loc,redun_level,ne,pe,number_pes;
  int r1,r2,r3,r4,r5; /* redundancy level counters */
  int upe[Pe_per_ne * Num_ne]; /* pe's used in a VID */

  r1 = 0;
  r2 = 0;
  r3 = 0;
  r4 = 0;
  r5 = 0;
```

```
   for(i = 0; i <= (Pe_per_ne * Num_ne - 1); i++)
     upe[i] = 0;

   for(i = 0; i < gnum_vids; i++)
   {
      vid_loc = gvids_used[i] * Bytes_per_CT_entry;
      redun_level = ct[vid_loc + Redun_loc] & redun_level_mask;
      switch (redun_level)
      {
      case 1: printf("\nvid %i is a simplex\n",ct[vid_loc]);
        r1 += 1;
        break;
      case 2: printf("\nWARNING!! vid %i is a duplex\n",ct[vid_loc]);
        r2 += 1;
        break;
      case 3: printf("\nvid %i is a triplex\n",ct[vid_loc]);
        r3 += 1;
        break;
      case 4: printf("\nvid %i is a quad\n",ct[vid_loc]);
        r4 += 1;
        break;
      case 5: printf("\nvid %i is a quint\n",ct[vid_loc]);
        r5 += 1;
        break;

      }
      for (j = 0; j <= (redun_level-1); j++)
      {
        ne = ct[vid_loc + Base_pe_loc + j] & ne_mask;
        pe = ct[vid_loc + Base_pe_loc + j] >> pe_shift;
        printf("member %i: ne = %i, pe = %i\n",j,ne,pe);
        if (upe[(ne * Pe_per_ne) + pe])
        printf("WARNING!! ne = %i pe = %i used more than once in ct\n"
               ,ne,pe);
        else upe[(ne * Pe_per_ne) + pe] = 1;
      }
   }
   number_pes = (r5 * 5) + (r4 * 4) + (r3 * 3) + (r2 * 2) + r1;
   if (number_pes > (Pe_per_ne * Num_ne))
     printf("WARNING!! Used too many pe's\n\n");

   printf("\nThe redundant groups broke down like so:\n");
   printf("      simplex = %i\n",r1);
   printf("       duplex = %i\n",r2);
   printf("      triplex = %i\n",r3);
   printf("         quad = %i\n",r4);
   printf("        quint = %i\n\n\n",r5);

}

/***************************** get_ct ******************************
*   generates a new ct when called                                  *
*****************************************************************/

void calculate_ct()
{
   int i,j,place,starting_place,num_entries_found,ct_entry[5];
   int ne,pe,desired_redun,redun_level,presence;
   int remaining_pes,used_pes[Pe_per_ne * Num_ne];
   Byte vid;
   Boolean vid_filled,able_to_fill_vid,got_a_vid,once_around;

   for (i = 0; i <= (Pe_per_ne * Num_ne - 1); i++)
```

```
  used_pes[i] = 0;
remaining_pes = Pe_per_ne * Num_ne;
place = 0;
while (!(remaining_pes == 0))
{
  desired_redun = random() & 0xf;
  if (desired_redun <= Prob_simplex)
    desired_redun = 1;
  else if (desired_redun <= Prob_triplex)
    desired_redun = 3;
  else
    desired_redun = 4;
  if (desired_redun > Max_redun_level)
    desired_redun = Max_redun_level;

  vid_filled = False;
  able_to_fill_vid = True;
  once_around = False;
  num_entries_found = 0;
  starting_place = place;
  while (!(vid_filled) && able_to_fill_vid)
  {
    if (used_pes[place] == 0)
    {
     ct_entry[num_entries_found] = place;
     num_entries_found += 1;
     /* skip to next NE */
     place = ((place / Pe_per_ne) + 1) * Pe_per_ne;
    }
    else place += 1;

    if (place > (Pe_per_ne * Num_ne))
    {
     place = 0;
     once_around = True;
    }
    if (once_around && (place >= starting_place))
    able_to_fill_vid = False;
    if (num_entries_found == desired_redun)
    vid_filled = True;
  }
  if (vid_filled)
  {
    /* Find a vid number for the new virtual group */
    got_a_vid = False;
    while (!(got_a_vid))
    {
     vid = random() & 0xff;
     if (!(vid_found(vid)) && (vid <= Max_vid))
       got_a_vid = True;
    }
    presence = 0;
    ct[vid * Bytes_per_CT_entry + Redun_loc] = desired_redun;
    for (i = 0; i <= (desired_redun - 1); i++)
    {
     /* this fashions the proper number of presence bits */
     presence |= gpmasks[i];
     ne = ct_entry[i] / Pe_per_ne;
     pe = ct_entry[i] % Pe_per_ne;
     ct[vid * Bytes_per_CT_entry + i + Base_pe_loc] = pack(ne,pe);
     used_pes[ct_entry[i]] = 1;
    }
     ct[vid * Bytes_per_CT_entry + Presence_loc] |=
     (presence << presence_shift);
```

```
      /* set a timeout value for the vid */
      ct[vid * Bytes_per_CT_entry + To_loc] = random() & 0xff;
      gvids_used[gnum_vids] = vid;
      gnum_vids += 1;
      remaining_pes -= desired_redun;
    }
  }
}

/********************** init_arrays ********************************
* this routine clears the vids_used array,the ct, and the gnum_vids  *
* variable.                                                          *
*********************************************************************/

void init_arrays()
{
  int i,vid;

  gpmasks[0] = 1;
  for (i = 1; i <= (Max_redun_level - 1); i++)
    gpmasks[i] = gpmasks[i - 1] * 2;

  for (i = 0; i <= (Pe_per_ne * Num_ne * Bytes_per_SERP_entry - 1); i++)
    ptov_table[i] = -1;
  for (vid = 0; vid <= (Num_vids - 1); vid++)
    ct[vid * Bytes_per_CT_entry] = vid;
  for (i = 0; i <= (Num_vids - 1); i++)
  {
    gvids_used[i] = -1;
    gfree_sources[i] = True;
    gfree_dests[i] = True;
  }
  gnum_vids = 0;
}

/*********************************************************************/
/* Clear_voted_serp                                                  */
/* This function clears the processed bits in the voted serp         */
/* array.                                                            */
/*********************************************************************/

void clear_voted_serp ()
{
  int i;

  for (i = 0; i <= gnum_vids; i++)
    gvoted_serp[gvids_used[i] * VS + VS_processed_bit_loc]
      ^= VS_processed_bit_mask;
}
```

10.8.4. File nf_serp.c

This file contains functions to generate a SERP which does not contain any faults.

```
#include "sbdefs.h"

/****************** set_bytes ********************************
* this function sets the dest_vid field of svid to dvid, the *
* ibnf bits of svid, and the obne bits of dvid. It then      *
* builds a reference message for error checking.             *
*************************************************************/

void nf_set_bytes (svid,dvid,num_messages)
Byte svid,dvid;
int *num_messages;
{
   Byte ex_class = 1;
   Byte source_presence,dest_presence;
   int i,redun_level,pe;

   /* set to,dest buffer,obne,exchange class, and dest vid for source */
   redun_level = ct[svid * Bytes_per_CT_entry + Redun_loc]
     & redun_level_mask;
   for (i = 0; i <= (redun_level - 1); i++)
   {
     pe = unpack(ct[svid * Bytes_per_CT_entry + Base_pe_loc + i]);
     serp[(pe * Bytes_per_SERP_entry) + Obne_loc] |= obne_mask;
     serp[(pe * Bytes_per_SERP_entry) + Class_loc] |= ex_class;
     serp[(pe * Bytes_per_SERP_entry) + Dvid_loc] = dvid;
   }
   /* set ibnf for destination */
   redun_level = ct[dvid * Bytes_per_CT_entry + Redun_loc]
     & redun_level_mask;
   for (i = 0; i <= (redun_level - 1); i++)
   {
     pe = unpack(ct[dvid * Bytes_per_CT_entry + Base_pe_loc + i]);
     serp[(pe * Bytes_per_SERP_entry) + Ibnf_loc] |= ibnf_mask;
   }
   source_presence = ct[svid * Bytes_per_CT_entry + Presence_loc]
     >> presence_shift;
   dest_presence = ct[dvid * Bytes_per_CT_entry + Presence_loc]
     >> presence_shift;
    set_message(num_messages,svid,dvid,ex_class,source_presence,
              dest_presence);
}

/****************** generate_nf_serp ************************
* generates a new no-fault serp by randomly sending messages  *
* between VID's. It also builds the correct-message structure *
*************************************************************/

void generate_nf_serp(num_messages)
int *num_messages;
{
   Boolean got_source = False,got_dest= False,all_done = False;
   int i,vids_left,pot_svid,pot_dvid;
   int used_sources[Num_vids],used_dest[Num_vids];

   *num_messages = 0;
   vids_left = gnum_vids;
```

```
    /* initialize serp to 0 */
    for (i = 0; i <= (Pe_per_ne * Num_ne * Bytes_per_SERP_entry - 1); i++)
      serp[i] = 0;

    /* initialize "used" array's to zero */
    for (i = 0; i <= (Num_vids - 1); i ++)
    {
      used_sources[i] = 0;
      used_dest[i] = 0;
    }

    while (!(all_done))
    {
      while (!(got_source))
      {
        /* generate a random VID from 0 to 255 */
        /* pot_svid = potential source vid      */
        pot_svid = random() & 0xff;

        if (vid_found(pot_svid))
        if (!(used_sources[pot_svid]))
        {
         /* source VID found, invalidate this VID as a potential source */
          used_sources[pot_svid] = 1;
          got_source = True;
        }
      }

      while (!(got_dest))
      {
        /* pot_dvid = potential destination vid */
        pot_dvid = random() & 0xff;
        if (vid_found(pot_dvid))
        if (!(used_dest[pot_dvid]))
        {
          /* invalidate this VID as a potential dest */
          used_dest[pot_dvid] = 1;
          got_dest = True;
          nf_set_bytes (pot_svid,pot_dvid,num_messages);
          *num_messages += 1;
          vids_left -= 2;
        }
      }
      if (vids_left < 2)
        all_done = True;
      got_source = False;
      got_dest = False;
    }
}
```

10.8.5. File serp.c

This file contains functions to generate a SERP which contains faults embedded in it. This version does not work correctly.

```
#include "sbdefs.h"
int num_faults;

/*************************************************************************/
/************************* file serp.c ***********************************
* This file containd the source code which generates the serp. It      *
* updates the serp_source array, and then it creates a SERP from       *
* that array.                                                           *
*************************************************************************/
/*************************************************************************/


/****************************** pack   ***********************************
*   pack converts an (ne,pe) pair into its corresponding byte           *
* representation in the ct.                                             *
*************************************************************************/

Byte pack(ne,pe)
int ne,pe;
{
  return((pe << pe_shift) | ne);
}

/****************** unpack   *********************************
* this function "unpacks" the physical pe number encoded in   *
* the ct.                                                     *
***************************************************************/

Byte unpack(ct_entry)
Byte ct_entry;
{
  int pe,ne,serp_loc;

  ne = ct_entry & ne_mask;  /* ne number is last three bits */
  pe = ct_entry >> pe_shift;   /* shift out ne number for pe number */

  serp_loc = (ne * Pe_per_ne) + pe;
  return(serp_loc);
}

/********************** vid_found ***************************
* this function searches the vids_used array for the vid it *
* it is passed.                                             *
*************************************************************/

Boolean vid_found(vid)
Byte vid;
{
  int i;
  Boolean found = False;

  for (i = 0; i <= (gnum_vids - 1); i++)
    if (vid == gvids_used[i])
      found = True;
```

```
  return(found);
}

/***************** set_message ****************************
* sets the correct message information in the cmessage      *
* array. This is the information which the scoreboard must  *
* provide after it has processed the serp.                  *
************************************************************/

void set_message(num_messages,svid,dvid,ex_class,obne_syndrome,
                 ibnf_syndrome)
int *num_messages;
Byte svid,dvid,ex_class,obne_syndrome,ibnf_syndrome;
{
  int i,redun_level;
  struct message_struct *s;

  s = &(gcmessage[*num_messages]);

  s->source_vid = svid;
  s->dest_vid = dvid;
  s->ex_class = ex_class;
  s->obne_to = obne_syndrome;
  s->ibnf_to = ibnf_syndrome;

  for (i = 0; i <= (Max_redun_level - 1); i++)
  {
    s->sources[i] = 0;
    s->dests[i] = 0;
  }
  /* set sources array to pe's in source vid */
  redun_level = ct[svid * Bytes_per_CT_entry + Redun_loc]
    & redun_level_mask;
  for (i = 0; i <= (redun_level - 1); i++)
    s->sources[i] = ct[svid * Bytes_per_CT_entry + Base_pe_loc + i];

  /* set dests array to pe's in destination vid */
  redun_level = ct[dvid * Bytes_per_CT_entry + Redun_loc]
    & redun_level_mask;
  for (i = 0; i <= (redun_level - 1); i++)
    s->dests[i] = ct[dvid * Bytes_per_CT_entry + Base_pe_loc + i];
}

/****************** set_bytes *********************************
* this function sets the dest_vid field of svid to dvid, the   *
* ibnf bits of svid, and the obne bits of dvid. It then builds *
* a reference message for error checking.                      *
****************************************************************/

void set_bytes(svid,dvid,obne,ibnf,ex_class)
Byte svid,dvid;
Bit obne[Max_redun_level],ibnf[Max_redun_level];
Byte ex_class;
{
  int i,redun_level,pe;

/* set to,dest buffer,obne,exchange class, and dest vid for source */
  redun_level = ct[svid * Bytes_per_CT_entry + Redun_loc]
    & redun_level_mask;
  for (i = 0; i <= (redun_level - 1); i++)
  {
    pe = unpack(ct[svid * Bytes_per_CT_entry + Base_pe_loc + i]);
    serp[(pe * Bytes_per_SERP_entry) + Obne_loc] |= obne[i];
    serp[(pe * Bytes_per_SERP_entry) + Class_loc] |= ex_class;
```

```
      serp[(pe * Bytes_per_SERP_entry) + Dvid_loc] = dvid;
    }
  /* set ibnf for destination */
    redun_level = ct[dvid * Bytes_per_CT_entry + Redun_loc]
      & redun_level_mask;
    for (i = 0; i <= (redun_level - 1); i++)
    {
      pe = unpack(ct[dvid * Bytes_per_CT_entry + Base_pe_loc + i]);
      serp[(pe * Bytes_per_SERP_entry) + Ibnf_loc] |= ibnf[i];
    }
  }

  /********************* generate_serp ************************
  * this function actually produces the serp based on the      *
  * info contained in the serp_source array                    *
  ************************************************************/

  void generate_serp(num_messages)
  int *num_messages;
  {
    struct serp_source_struct *s;
    int i,j,redun_level;
    Boolean obne_unan,ibnf_unan,message_to_send;
    Byte obne[Max_redun_level],ibnf[Max_redun_level];
    Byte ex_class = 3;

    /* masks array is used to mask out all but one bit of the obne */
    /* and ibnf timeouts */

    *num_messages = 0;
    for (i = 0; i <= (gnum_vids - 1); i++)
    {
      s = &gserp_source[i];
      obne_unan = True;
      ibnf_unan = True;
      message_to_send = False;

      redun_level = s->srd;
      for (j = 0; j <= (redun_level - 1); j++)
      {
        if (((s->obne_to) & gpmasks[j]) > 0)
        obne[j] = obne_mask;
       else
       {
        obne[j] = 0;
        obne_unan = False;
       }
      }

      redun_level = s->drd;
      for (j = 0; j <= (redun_level - 1); j++)
      {
        if (((s->ibnf_to) & gpmasks[j]) > 0)
        ibnf[j] = ibnf_mask;
       else
       {
        ibnf[j] = 0;
        ibnf_unan = False;
       }
      }
      message_to_send = obne_unan && ibnf_unan;

       set_bytes(s->source_vid,s->dest_vid,obne,ibnf,ex_class);
```

```
    /* increment serps_done variable */
    s->serps_done += 1;

    if (message_to_send)
    {
        set_message(num_messages,s->source_vid,s->dest_vid,ex_class,
                s->obne_to,s->ibnf_to);
      *num_messages += 1;
      gfree_sources[s->source_vid] = True;
      gfree_dests[s->dest_vid] = True;
    }
  }
}

/******************** get_fault ****************************
* get_fault decides which member of a vid is too be faulty   *
* and generates the proper timeout syndrome.                 *
************************************************************/

void get_flow_control(vid,redun_level,fault,syndrome)
Byte vid,redun_level;
Boolean *fault;
Byte *syndrome;
{
   int i,rnumber,faulty_pe,position,non_faulty_syndrome;
  Boolean inject_fault = False;

  /* create the default syndrome */
  *syndrome = 0;
  for (i = 0; i <= (redun_level - 1); i++)
    *syndrome |= gpmasks[i];

  /* determines whether to inject a fault or not */
  rnumber = random() & 0xf;
  if (rnumber <= Prob_fault)
    inject_fault = True;

  /* make sure only one fault per message */
  if (inject_fault && !(*fault))
  {
    if (debug_level >= 3)
      printf("NOTICE! Fault injected in vid = %i\n",vid);

    *fault = True;
    num_faults += 1;
    switch(redun_level)
    {
    case 1: faulty_pe = 1;
      break;

    case 3:
      faulty_pe = random() & 0x03;
      if ((faulty_pe == 3) || (faulty_pe == 0)) faulty_pe = 4;
      break;

    case 4:
      faulty_pe = random() & 0x03;
      if (faulty_pe == 0) faulty_pe = 1;
      else if (faulty_pe == 1) faulty_pe = 2;
      else if (faulty_pe == 2) faulty_pe = 4;
      else if (faulty_pe == 3) faulty_pe = 8;
      break;
    }
    /* set the faulty pe's presence bit to zero */
```

```
      *syndrome ^= faulty_pe;
    }
  }

  /***************** set_serp_source_entry ********************
  * this function creates an entry in the serp_source array     *
  * for the source_vid it is passed.                            *
  ***************************************************************/

  void set_serp_source_entry (source_vid,dest_vid,location)
  Byte source_vid,dest_vid;
  int location;
  {
    struct serp_source_struct *s;
    Boolean *fault = False;

    s = &gserp_source[location];
    s->source_vid = source_vid;
    s->dest_vid = dest_vid;
    s->srd = ct[source_vid * Bytes_per_CT_entry + Redun_loc]
      & redun_level_mask;
    s->drd = ct[dest_vid * Bytes_per_CT_entry + Redun_loc]
      & redun_level_mask;

    get_flow_control(source_vid,s->srd,&fault,&(s->obne_to));
    get_flow_control(dest_vid,s->drd,&fault,&(s->ibnf_to));

    s->serps_done = 0;
  }

  /****************** generate_serp_source *********************
  * generates a new serp by randomly sending messages between   *
  * VID's. It builds the correct message-structure too.         *
  ***************************************************************/

  void generate_serp_source()
  {
    Boolean got_source,got_dest,all_done;
    int i,source_vid,dest_vid,pot_dvid,place;

    got_source = False;
    got_dest= False;
    all_done = False;
    place = 0;
    num_faults = 0;

    /* initialize serp to 0 */
    for (i = 0; i <= (Pe_per_ne * Num_ne * Bytes_per_SERP_entry); i++)
      serp[i] = 0;

    while (!(all_done))
    {
      while (!(got_source))
      {
        if (gfree_sources[gvids_used[place]])
        {
         source_vid = gvids_used[place];
         gfree_sources[source_vid] = False;
         got_source = True;
        }
        else place += 1;
        if (place == gnum_vids)
        {
         got_source = True;
```

```
        all_done = True;
       }
     }

     while (!(got_dest) && !(all_done))
     {
       pot_dvid = random() & 0xff;
       if (vid_found(pot_dvid))
       if (gfree_dests[pot_dvid])
       {
        dest_vid = pot_dvid;
        gfree_dests[dest_vid] = False;
          got_dest = True;
       }
     }
     if (!(all_done))
        set_serp_source_entry(source_vid,dest_vid,place);
     got_source = False;
     got_dest = False;
   }
}

/****************************** get_serp ******************************
*   generates a new serp when called                                    *
*********************************************************************/

void get_serp (num_messages)
int *num_messages;
{
   static Boolean generate_ct = True;

   if (generate_ct)    /* generate a new ct? */
   {
      init_arrays();
      calculate_ct();
      get_table();
      generate_ct = False;
      if (debug_level >= 1)
        check_ct();
   }
   if (generate_faults)
   {
      generate_serp_source();
      if (debug_level >= 1)
        printf("\nNOTICE! number of faults = %i\n",num_faults);
      generate_serp(num_messages);
   }
   else
   {
      generate_nf_serp(num_messages);
   }
}
```

10.8.6. File vote.c

This file contains functions to vote the SERP to arrive at correct messages. It is very similar to the code in Appendix 10.2.

```
#include "sbdefs.h"
#include <math.h>

/********************** file vote.c ******************************
* This file contains the source code for voting the SERP, keeping *
* track of timeouts, and writing the results into voted_serp.      *
*******************************************************************/

/*******************************************************************/
/*     vote is a generic vote function which will vote up to 5     */
/* items passed to it. It returns three flags and the result.      */
/* The simplex flag signals that the presence bits indicate a      */
/* simplex configuration.                                          */
/*******************************************************************/

Byte vote (vote_values,redun_level,unan)
Byte vote_values[Max_redun_level];
int redun_level;
Bit *unan;
{
  Byte a,b,c,d;
  Byte result,int_res1,int_res2; /* int=intermediate */
  Boolean AB,BC,CD;

  a = vote_values[0];
  b = vote_values[1];
  c = vote_values[2];
  d = vote_values[3];

  AB = (a == b) ? True : False;    /* used for flags */
  BC = (b == c) ? True : False;
  CD = (c == d) ? True : False;

  switch (redun_level)
  {
  case 4:
    *unan = (AB && BC && CD) ? True : False;
    result = (a&b&c) | (a&c&d) | (b&c&d) | (a&b&d);
    break;
  case 3:
    *unan = (AB && BC) ? True : False;
    result = (a & b) | (b & c) | (a & c);
    break;
  case 2:
    printf("ERROR! Voted a duplex\n");
    break;
  case 1:
    *unan = True;
    result = a;
    break;
  }
  return(result);
}
/* end vote */

/*******************************************************************/
```

```
/*     read_timer reads and returns the current timer value.            */
/*************************************************************************/

int read_timer()
{
   return (to_clock);
}

/*************************************************************************/
/*     check_to checks to see if the timeout value (to_value) has been */
/*  reached. If it has, then it returns a true value for to_reached.   */
/*************************************************************************/

Boolean check_to (vid,to_value,timer_value,init_timer_value)
Byte vid, to_value,*timer_value,*init_timer_value;
{
  Boolean to_reached = False;

  *timer_value = read_timer();
  if (to[vid] == 0)
    to[vid] = *timer_value;    /* TO set? then set a timeout */
  else if ((*timer_value - to[vid]) > to_value)
  {
    printf("NOTICE! timeout for vid %i reached\n",vid);
    to_reached = True;
    *init_timer_value = to[vid];
    to[vid] = 0;
  }
  return (to_reached);
}
/* end check_to */

/*************************************************************************/
/*     fc_vote performs the flow control vote function (i.e. OBNE      */
/* and IBNF). If a timeout is reached, it clears that pe's syndrome    */
/* bit and sets the result to true (ibnf or obne).                      */
/*************************************************************************/

void fc_vote (vid,vote_values,redun_level,to_value,result,fault,
              syndrome,timer_value,init_timer_value)

Byte vid,vote_values[Max_redun_level];
int redun_level;
Byte to_value, *result;
Boolean *fault;
Byte *syndrome,*timer_value,*init_timer_value;
{
  Boolean unan;
  int i;
  Byte old_result;

  *result = vote (vote_values,redun_level,&unan);

  /* set default syndrome */
  *syndrome = 0;
  for (i = 0; i <= (redun_level - 1); i++)
    *syndrome |= gpmasks[i];

  if (!(unan) && (*result != 0))    /* check for timeouts */
  {
    old_result = *result;
    *result = 0;
    *fault = check_to (vid,to_value,timer_value,init_timer_value);
    if (*fault)
```

```
    {
      /* reset obne (or ibnf) bit and zero proper presence bit */
      *result = old_result;

      /* clear the offending pe's syndrome bit */
      for (i = 0; i <= (redun_level - 1); i++)
      if (vote_values[i] != *result)
        *syndrome ^= gpmasks[i];
    }
  else                    /* check for illegal transitions */
    {
      /* to be determined */
    }
  }
}
/* end fc_vote */

/*****************************************************************/
/*     vote_other is the function which votes the destination    */
/* VID and exchange class fields of the SERP.                    */
/*****************************************************************/

void vote_other (vote_values,redun_level,result,obne,fault)
Byte vote_values[Max_redun_level];
int redun_level;
Byte *result;
Boolean *fault;
{
  Bit unan,maj;

  *result = vote (vote_values,redun_level,&unan);

  if (!unan && obne) *fault = True;
}
/* end vote_other */

/*****************************************************************/
/*   write_result writes the overall, voted SERP entry for       */
/* each VID into the voted_serp table.                           */
/*****************************************************************/

void write_result (vid,to_value,timer_value,init_timer_value,obne,ibnf,
                   ex_class,dest_vid,obne_syndrome,ibnf_syndrome)
Byte vid,to_value,timer_value,init_timer_value,obne,ibnf;
Byte ex_class,dest_vid,obne_syndrome,ibnf_syndrome;
{

  gvoted_serp[vid * VS + VS_to_loc] = to_value;
  if (obne)
    gvoted_serp[vid * VS + VS_obne_loc] |= VS_obne_mask;
  if (ibnf)
    gvoted_serp[vid * VS + VS_ibnf_loc] |= VS_ibnf_mask;
  gvoted_serp[vid * VS + VS_class_loc] |= ex_class;
  gvoted_serp[vid * VS + VS_dvid_loc] = dest_vid;
  gvoted_serp[vid * VS + VS_obne_syn_loc] |= obne_syndrome;
  gvoted_serp[vid * VS + VS_ibnf_syn_loc] |= ibnf_syndrome;
  gvoted_serp[vid * VS + VS_timer_loc] = timer_value;
  gvoted_serp[vid * VS + VS_init_timer_loc] = init_timer_value;
}

/************************ vote_serp ****************************
* vote_serp receives the serp entries from feed_voter and votes *
* them when told to. It writes the overall result for each VID  *
* into the voted_serp array.                                    *
```

```
*****************************************************************/

void vote_serp (vid,serp_values,redun_level,to_value,fault)
Byte vid,serp_values[Max_redun_level * Bytes_per_SERP_entry];
int redun_level;
Byte to_value;
Boolean *fault;
{
   Byte i,vote_values[Max_redun_level],obne_syndrome,ibnf_syndrome;
   Byte timer_value,init_timer_value,obne,ibnf;
   Byte ex_class,dest_vid;

   /*********************** get and vote OBNE bits */
   for (i = 0; i <= (Max_redun_level - 1); i++)
     vote_values[i] = serp_values[i * Bytes_per_SERP_entry + Obne_loc]
       & obne_mask;

   /* NOTE!! timer value will take on the timer value at the time */
   /* the ibnf is voted, NOT the obne */
   fc_vote (vid,vote_values,redun_level,to_value,&obne,fault,
          &obne_syndrome,&timer_value,&init_timer_value);
   if ((*fault) && debug_level >= 3)
     printf("FAULT in OBNE vote\n");
   *fault = False;
   /*************************************************************/

   /*********************** get and vote IBNF bits */
   for (i = 0; i <= (Max_redun_level - 1); i++)
     vote_values[i] = serp_values[i * Bytes_per_SERP_entry + Ibnf_loc]
       & ibnf_mask;
   fc_vote (vid,vote_values,redun_level,to_value,&ibnf,fault,
          &ibnf_syndrome,&timer_value,&init_timer_value);
   if ((*fault) && debug_level >= 3)
     printf("FAULT in IBNF vote\n");
   *fault = False;
   /*************************************************************/

    /*********************** get and vote exchange class */
   for (i = 0; i <= (Max_redun_level - 1); i++)
     vote_values[i] = serp_values[i * Bytes_per_SERP_entry + Class_loc]
       & class_mask;
   vote_other (vote_values,redun_level,&ex_class,obne,fault);
   if ((*fault) && debug_level >= 3)
     printf("FAULT in exchange class vote\n");
   *fault = False;
   /*************************************************************/

   /*********************** get and vote destination VID */
   for (i = 0; i <= (Max_redun_level - 1); i++)
     vote_values[i] = serp_values[i * Bytes_per_SERP_entry + Dvid_loc];
   vote_other (vote_values,redun_level,&dest_vid,obne,fault);
   if ((*fault) && debug_level >= 3)
     printf("FAULT in dest VID vote = %i\n",dest_vid);
   *fault = False;
   /*************************************************************/

   write_result (vid,to_value,timer_value,init_timer_value,obne,ibnf,
              ex_class,dest_vid,obne_syndrome,ibnf_syndrome);

}

/********************** feed_voter ********************************
* feeds the serp voting function one serp value at a time using  *
* a vid-order translation table                                  *
```

```
**********************************************************************/

void feed_voter()
{
  Boolean fault;
  int i,j,num_entries,serp_place,vid_place,current_vid,fault_mask;
  Byte redun_level,serp_values[Max_redun_level * Bytes_per_SERP_entry];
  Byte to_value;

  /* feed the voter one vid at a time */

  num_entries = gnum_vids + (Pe_per_ne * Num_ne);
  serp_place = 0;
  vid_place = 0;
  fault = False;
  while (serp_place <= num_entries)
  {
    current_vid = gvids_used[vid_place];
    redun_level = ptov_table[serp_place];
    vid_place += 1;
    serp_place += 1;

    /* iterate over the redundancy level */
    for(i = 0; i <= (redun_level - 1); i++)
    {
      /* accumumulate each PE's entry */
      for(j = 0; j <= (Bytes_per_SERP_entry - 1); j++)
      {
       serp_values[Bytes_per_SERP_entry * i + j] =
         serp[ptov_table[serp_place]   * Bytes_per_SERP_entry + j];
      }
      /* move to next pe */
      serp_place += 1;
    }
    to_value = ct[current_vid * Bytes_per_CT_entry + To_loc];
    vote_serp (current_vid,serp_values,redun_level,to_value,&fault);
  }
}
```

10.8.7. File send.c

This file contains functions to cycle through the voted SIFT memory and "send" all messages contained therein.

```
#include "sbdefs.h"

/******************** file send.c ********************************
* this file cycles through the voted serp, sending all valid    *
* messages                                                      *
*****************************************************************/

/***************** create_message ******************************
* this function creates the message packet when called by send *
*****************************************************************/

void create_message(source_vid,dest_vid,ex_class,message_number,
                    obne_syndrome,ibnf_syndrome,timer_value,
                    init_timer_value)
Byte source_vid,dest_vid,ex_class;
int message_number;
Byte obne_syndrome,ibnf_syndrome,timer_value,init_timer_value;
{
   struct message_struct *s;
   Byte redun_level;
   int i,j;

   s = &gmessage[message_number];

   s->source_vid = source_vid;
   redun_level = ct[source_vid * Bytes_per_CT_entry + Redun_loc]
     & redun_level_mask;
   for(i = 0; i <= (redun_level - 1); i++)
     s->sources[i] = ct[source_vid * Bytes_per_CT_entry + Base_pe_loc+i];

   s->dest_vid = dest_vid;
   redun_level = ct[dest_vid * Bytes_per_CT_entry + Redun_loc]
     & redun_level_mask;
   for(i = 0; i <= (redun_level - 1); i++)
     s->dests[i] = ct[dest_vid * Bytes_per_CT_entry + Base_pe_loc + i];

   s->ex_class = ex_class;
   s->obne_to = obne_syndrome;
   s->ibnf_to = ibnf_syndrome;
   s->timer_value = timer_value;
   s->itv = init_timer_value;
}

/*************************** send ********************************
* send cycles through the vote_serp array, sending all valid    *
* messages.                                                     *
*****************************************************************/

void send(num_messages)
int *num_messages;
{
   Boolean all_valid_sent = False;
   Byte source_vid,dest_vid,ex_class;
   Bit obne,ibnf,processed;
   Byte obne_syndrome,ibnf_syndrome,timer_value,init_timer_value;
   int processed_vids = 0;
```

```
    int current_vid_place = 0;

    *num_messages = 0;

    /* this is the vid where send begins to look for messages */
    source_vid = gvids_used[current_vid_place];

    while(!(all_valid_sent))
    {
       processed = gvoted_serp[source_vid * VS + VS_processed_bit_loc]
         & VS_processed_bit_mask;
       if (!(processed))
       {
         processed_vids += 1;
         gvoted_serp[source_vid * VS + VS_processed_bit_loc]
         |= VS_processed_bit_mask;
         obne = gvoted_serp[source_vid * VS + VS_obne_loc] & VS_obne_mask;
         if (obne)
         {
          dest_vid = gvoted_serp[source_vid * VS + VS_dvid_loc];
          ibnf = gvoted_serp[dest_vid * VS + VS_ibnf_loc] & VS_ibnf_mask;
          if (ibnf)
          {
            if (debug_level >= 3)
            {
               printf("Sending message %i\n",*num_messages);
               printf("      Source vid = %i\n",source_vid);
               printf("      Dest vid = %i\n\n",dest_vid);
            }
            ex_class = gvoted_serp[source_vid * VS + VS_class_loc]
              & VS_class_mask;
            obne_syndrome = gvoted_serp[source_vid * VS + VS_obne_syn_loc]
              & VS_obne_syn_mask;

            /* ibnf syndrome comes from dest_vid */
            ibnf_syndrome = gvoted_serp[dest_vid * VS + VS_ibnf_syn_loc]
              & VS_ibnf_syn_mask;
            timer_value = gvoted_serp[source_vid * VS + VS_timer_loc];
            init_timer_value = gvoted_serp[source_vid * VS +
                                       VS_init_timer_loc];
            create_message(source_vid,dest_vid,ex_class,*num_messages,
                        obne_syndrome,ibnf_syndrome,timer_value,
                        init_timer_value);
            *num_messages += 1;
          }
         }
       }
       if (processed_vids == gnum_vids)
         all_valid_sent = True;

       if (current_vid_place > (gnum_vids - 1))
         current_vid_place = 0;
       else current_vid_place += 1;
       source_vid = gvids_used[current_vid_place];
    }
}
```

10.8.8. File check.c

This file contains functions to check the messages found by send.c against those written by nf_serp.c or serp.c.

```
#include "sbdefs.h"

#define MAX(a,b) ((a >= b) ? (a) : (b))

/********************** get_vid_position ***************************
* this function returns the index in the vids used array for the  *
* vid passed to it.                                                *
*******************************************************************/

int get_vid_position(vid)
Byte vid;
{
  int i = 0;

  while (i < gnum_vids)
    if (gvids_used[i] == vid)
      return(i);
    else ++i;
}

/************************ check_others *****************************
* this function checks for correctness all the messages not found *
* in the cmessage array                                            *
*******************************************************************/

void check_others(num_messages,marked_messages)
int num_messages,marked_messages[50];
{
  struct message_struct *m;
  struct serp_source_struct *ss,*sd; /* ss = source pointer */
  int i;                             /* sv = dest pointer */
  Boolean error;

  for (i = 0; i <= (num_messages - 1); i++)
  {
    gfree_sources[gmessage[i].source_vid] = True;
    gfree_dests[gmessage[i].dest_vid] = True;
    if (!(marked_messages[i]))
    {
      error = False;
      m = &gmessage[i];
       ss = &gserp_source[get_vid_position(m->source_vid)];
       sd = &gserp_source[get_vid_position(m->dest_vid)];

      /* the first IF statement decides which to_value to use */
      /* in checking for premature messages.                  */

      if (m->obne_to != ss->obne_to)
      /* use source to_value */
      if ((m->timer_value - m->itv) < ss->to_value)
      {
        printf("ERROR! message %i sent prematurely\n",i);
        error = True;
      }
      else if (m->ibnf_to != ss->ibnf_to)
        /* use dest to_value */
```

```
          if ((m->timer_value - m->itv) < sd->to_value)
          {
            printf("ERROR! message %i sent prematurely\n",i);
            error = True;
          }

          if (m->obne_to != ss->obne_to)
          {
           printf("ERROR! message %i has an incorrect OBNE syndrome\n",i);
           error = True;
          }
          if (m->ibnf_to != ss->ibnf_to)
          {
           printf("ERROR! message %i has an incorrect IBNF syndrome\n",i);
           error = True;
          }
          if (error)
          {
           printf("         sv = %i\n",m->source_vid);
           printf("         dv = %i\n",m->dest_vid);
          }
        }
      }
    }

/*****************************************************************
* this function compares the unanimous message list with the one *
* generated by the scoreboard. It reports all inconsistencies.   *
* It then calls check_others to check any remaining messages for *
* correctness.                                                    *
*****************************************************************/

check_messages (cnum_messages,num_messages)
int cnum_messages,num_messages;
{
  struct message_struct *s,*cs;
  int i,diff,place;
  Bit marked_messages[50];
  Boolean found,message_not_found;
  Byte source_vid,csource_vid,cdest_vid;

  for (i = 0; i <= 50; i++)
    marked_messages[i] = 0;

  if (cnum_messages != num_messages)
  {
    if (cnum_messages > num_messages)
      printf("ERROR! Not enough messages found!\n\n");
    else
      printf("ERROR! Too many messages found!\n\n");
  }

  if (debug_level >= 2)
  {
    printf("Simulation found %i messages\n",num_messages);
    printf("There are %i necessary messages\n\n",cnum_messages);
  }

  /* check that all unanimous messages have been sent */
  for (i = 0; i <= (cnum_messages - 1); i++)
  {
    found = False;
    message_not_found = False;
    place = 0;
```

```
    cs = &gcmessage[i];
    csource_vid = cs->source_vid;
    cdest_vid = cs->dest_vid;
    while (!(found))
    {
      s = &gmessage[place];
      if ((s->source_vid == csource_vid) && (s->dest_vid == cdest_vid))
      {
       found = True;
       marked_messages[place] = 1;
       if ((s->obne_to) != (cs->obne_to))
         printf("ERROR! obne syndrome incorrect for message %i\n",i);
       if ((s->ibnf_to) != (cs->ibnf_to))
         printf("ERROR! ibnf syndrome incorrect for message %i\n",i);
      }
      else place += 1;

      /* check to see that array bounds haven't been reached */
      if (place > num_messages)
      {
       found = True; /* exit from loop */
       message_not_found = True; /* signal an error */
      }
    }
    if (message_not_found)
    {
      printf("WARNING! Message number %i not found\n",i);
      printf("       source vid = %i\n",csource_vid);
      printf("       dest vid = %i\n\n",cs->dest_vid);
    }
  }

  check_others(num_messages,marked_messages);
}
```

10.8.9. File io.c

This file contains input-ouput functions to write SERPs and CTs to an external file for reading into the VHDL model.

```
#include "sbdefs.h"
#include <stdio.h>


/*************************************************************/
/* write_status                                              */
/* This function writes the status line to the output file.  */
/* It is called after each serp-message cycle.               */
/*************************************************************/

void write_status(output_file,regenerate_ct,num_vids,num_serp_entries,
                  num_messages)
FILE *output_file;
Boolean regenerate_ct;
int num_vids,num_serp_entries,num_messages;
{
   fprintf(output_file,"%i %i %i %i",regenerate_ct,num_vids,num_serp_entries,
         num_messages);
  fprintf(output_file,"                        Status line\n");
}

/*************************************************************/
/* write_serp                                                */
/* This function simply writes the SERP to a file.           */
/*************************************************************/

void write_serp(output_file,num_serp_entries)
FILE *output_file;
int num_serp_entries;
{
  int place,i;
   int obne,ibnf,dvid,broadcast,packet_type,ex_class;

  for (i = 0; i < num_serp_entries; i++)
  {
    /* write out the complete SERP entry for each PE */
    place = i * Bytes_per_SERP_entry; /* a place holder */
    obne = (serp[place + Obne_loc] & obne_mask) ? True : False;
    ibnf = (serp[place + Ibnf_loc] & ibnf_mask) ? True : False;
    dvid = serp[place + Dvid_loc];
    broadcast = False; /* no broadcasts for now (30 Jan 91) */
    packet_type = 0;
    ex_class = 0;

    /* write the values to the file */
    fprintf(output_file,"%i %i %i %i %i %i\n",obne,ibnf,dvid,broadcast,
           packet_type,ex_class);

  }
}

/*************************************************************/
/* write_ct                                                  */
/* This function writes the ct to the output file in the     */
/* pre-determined format. (see documentation)                */
/*************************************************************/
```

```
void write_ct(output_file)
FILE *output_file;
{
  int i,j,loc;
  int vid_number,redun,presence,timeout;

  for (i = 0; i < Num_vids; i++)
  {
    /* calculate all the entries for a file line */
    vid_number = ct[i * Bytes_per_CT_entry];
    redun = ct[i * Bytes_per_CT_entry + Redun_loc] & redun_level_mask;
    timeout = ct[i * Bytes_per_CT_entry + To_loc];

    /* only print a VID's entry if the redun is non-zero */
    if (redun)
    {
       fprintf(output_file,"%i ",vid_number);
       fprintf(output_file,"%i ",redun);

       presence = ct[i * Bytes_per_CT_entry + Presence_loc]
       >> presence_shift;
       for (j = 0; j < Max_redun_level; j++)
       if (presence & gpmasks[j])
          fprintf(output_file,"1 ");
       else
          fprintf(output_file,"0 ");

      for (j = 0; j < redun; j++)
      {
       loc = unpack(ct[i * Bytes_per_CT_entry + Base_pe_loc + j]);
       fprintf(output_file,"%i ",loc);
      }
       fprintf(output_file,"%i ",timeout);
       fprintf(output_file,"\n");
    }
  }
}

/*************************************************************/
/* write_messages                                            */
/*************************************************************/

void write_messages(output_file,cnum_messages)
FILE *output_file;
int cnum_messages;
{

}
```

10.8.10. File main.c

This file contains the main() function which takes care of the command line switches.

```
#include "sbdefs.h"
#include <sys/time.h>
#include <stdio.h>

void increment_timer()
{
  to_clock += random() & 0x07;
}

/************************ parse_commands *** ********************
* This function parses the command line for default overrides of *
* num_rounds, generate_faults, and debug_level.                  *
*****************************************************************/

void parse_commands (argc,argv,seed,operation,file_name)
int argc;
char *argv[];
int *seed;
char *operation,file_name[10];
{
  char *str;

  while (--argc > 0)
  {
    str = argv[argc];
    if (!(str[0] == '-'))
      ;
    else
    {
      switch(str[1])
      {
      case 'd':
       if (sscanf(str,"%*c%*c%d",&debug_level) != 1)
       {
         printf("Bad debug level argument\n");
         exit(1);
       }
       break;
    case 'n':
       if (sscanf(str,"%*c%*c%d",&num_rounds) != 1)
       {
         printf("Bad number of rounds argument\n");
         exit(1);
       }
       break;
      case 'f':
       generate_faults = True;
       break;
      case 's':
       if (sscanf(str,"%*c%*c%d",seed) != 1)
       {
         printf("Bad seed argument\n");
         exit(1);
       }
       break;
      case 'o':
```

```
        /* tell the program to send the ct and serp to a file */
        *operation = 'o';
        /*if (sscanf(argv[--argc],"%s",file_name) != 1)
        {
          printf("Bad output file name\n");
          exit(1);
        }*/
        break;
       }
     }
   }
}

#define default_file "test.i"
#define seed_file_name "seed.last"

main (argc,argv)
int argc;
char *argv[];
{
  int i,seed = 0;
  int num_messages,cnum_messages;
  int num_serp_entries,serp_round,numbers;
  char operation,*file_name,*ptime_string,time_string[26];
  struct timeval time;
  struct timezone tzp;
  FILE *out_file,*seed_file;

  /* get the default seed for the random number generator */
  numbers = gettimeofday(&time,&tzp);
  /*ptime_string = time_string;
  ptime_string = asctime(time);*/
  seed = time.tv_sec & 0xff;

  /* set defaults and interpret the command line arguments */
  debug_level = 1;
  num_rounds = 50;
  generate_faults = False;
  operation = 's';
  file_name = default_file;

  parse_commands(argc,argv,&seed,&operation,file_name);

  /* seed the random number generator */
  for (i = 0; i <= seed; i++)
    numbers = random() & 0xff;

  /* write the seed to a file */
  if ((seed_file = fopen(seed_file_name,"a")) == NULL)
    printf("Error opening seed file -- continuing\n\n");
  else
  {
    fprintf(seed_file,"seed = %i\n",seed);
    fclose(seed_file);
  }

  switch(operation)
  {
  case 'o':
    /* 'o' for output to a file */
    printf("%s\n",file_name);
    if ((out_file = fopen(file_name,"w")) == NULL)
    {
      printf("Error in opening file -%s-\n",file_name);
```

```
      exit(1);
    }
    increment_timer();
    num_serp_entries = Pe_per_ne * Num_ne;
    get_serp(&cnum_messages);
     write_status(out_file,True,gnum_vids,num_serp_entries,cnum_messages);
    write_ct(out_file);
     write_serp(out_file,num_serp_entries);
     write_messages(out_file,cnum_messages);
    for (i = 0; i < num_rounds; i++)
    {
       get_serp(&cnum_messages);
        write_status(out_file,False,gnum_vids,num_serp_entries,cnum_messages);
       write_serp(out_file,num_serp_entries);
       write_messages(out_file,cnum_messages);
    }
    fclose(out_file);
   break;
  case 's':
    /* 's' for simulate */
    serp_round = 0;
    for (i = 0;i < num_rounds; i++)
    {
       increment_timer();
       get_serp(&cnum_messages);
       serp_round += 1;
       if (debug_level >= 1)
       {
        printf("*****************************************************\n");
        printf("Round = %i\n",serp_round);
        printf("*****************************************************\n");
       }
       feed_voter();
       send(&num_messages);
        check_messages(cnum_messages,num_messages);
       clear_voted_serp();
    }
  }
}
```

10.8.11. makefile

```
CFLAGS = -g

sb: io.o vote.o serp.o nf_serp.o ct.o send.o check.o main.o
	cc $(CFLAGS) -o sb io.o vote.o serp.o nf_serp.o ct.o send.o check.o main.o

io.o: io.c sbdefs.h config.h
	cc -c $(CFLAGS) io.c

vote.o: vote.c sbdefs.h config.h
	cc -c $(CFLAGS) vote.c

serp.o : serp.c sbdefs.h config.h
	cc -c $(CFLAGS) serp.c

nf_serp.o : nf_serp.c sbdefs.h config.h
	cc -c $(CFLAGS) nf_serp.c

ct.o : ct.c sbdefs.h config.h
	cc -c $(CFLAGS) ct.c

send.o : send.c sbdefs.h config.h
	cc -c $(CFLAGS) send.c

check.o : check.c sbdefs.h config.h
	cc -c $(CFLAGS) check.c

main.o: main.c sbdefs.h config.h
	cc -c $(CFLAGS) main.c

clean:
	rm *.*.~*
	rm *.~*
```

11. References

[Arm87] Armstrong, J. Chip-Level Modeling with VHDL, Prentice-Hall: Englewood Cliffs, 1989.

[Bohm91] Bohm, M. "Top-Down Design Using VHDL." Tutorial, VHDL User's Group Spring 1991 Conference, Apr. 8-10, 1991.

[Butler89] Butler, B. "A Fault-Tolerant Shared Memory System Architecture for a Byzantine Resilient Computer." Master of Science Thesis, MIT 1989.

[Dolev82] Dolev, D. "The Byzantine Generals Strike Again", Journal of Algorithms, Vol. 3, 1982, pp. 14-30.

[Harper87] Harper, Richard. "Critical Issues in Ultra-Reliable Parallel Processing." Doctor of Philosophy Thesis, MIT 1987.

[IEEE88] Institute of Electrical and Electronics Engineers Inc. 1988. "IEEE Standard VHDL Language Reference Manual," IEEE Standard 1076-1987.

[Jain91] Jain, P. "Architectural Models are Key to System Level Design", Electronic Design, Mar 28, 1991, pp. 57-70.

[Lamp82] Lamport, L. et. al. "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, Jul 1982, pp. 383-401.

[Mor91] Morton, D. "Hardware Modeling and Top-Down Design Using VHDL." Master of Science Thesis, MIT 1991.

[RL86] R. Lipsett, E. Marschner, M. Shahdad. "VHDL – The Language," IEEE Design and Test of Computers, Vol 3, No. 2, April, 1986, pp. 28-41.

[RL89] R. Lipsett, C. Schaefer, C. Ussery. VHDL : Hardware Description and Design, Kluwer : Boston, 1989.

[Sak91] Sakamaki, C. " The Design and Construction of a Data Path Chip Set for a Fault Tolerant Parallel Processor." Master of Science Thesis, MIT 1991.

[Syn90] Synopsys, Inc. "VHDL Compiler™ Reference," July, 1990.

[VHDL90] The VHDL Consulting Group. "VHDL System Design I," Seminar, Jul 8-9, 1990.

[Wax89] R. Waxman, L. Saunders. "The Evolution of VHDL," Information Processing, 1989, pp 735-742.