Interfacing VHDL Performance Models to Algorithm Partitioning Tools

by

Priya Balasubramanian

Thesis submitted to the faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Electrical Engineering

APPROVED:

Dr. F. G. Gray, Chairman

Dr. J. R. Armstrong

Dr. N. J. Davis, IV

July 1996
Blacksburg, Virginia

Key words : Performance models, algorithm partitioning tools

Interfacing VHDL Performance Models to Algorithm Partitioning Tools

by

Priya Balasubramanian

Dr. F. G. Gray

Electrical Engineering

(ABSTRACT)

Performance modeling is widely used to efficiently and rapidly assess the ability of multiprocessor architectures to effectively execute a given algorithm. In a typical design environment, VHDL performance models of hardware components are interconnected to form structural models of the target multiprocessor architectures. Algorithm features are described in application specific tools. Other automated tools partition the software among the various processors. Performance models evaluate the system performance. Since several iterations may be needed before a suitable configuration is obtained, a set of tools that directly interfaces the VHDL performance models to the algorithm partitioning tools will significantly reduce the time and effort needed to prepare the various models.

In order to develope the interface tools, it is essential to determine the information that needs to be interchanged between the two systems. The primary goals of this thesis are to study the various models, determine the information that needs to be exchanged, and to develop tools to automatically extract the desired information from each model.

This research work was funded by the RASSP (Rapid Prototyping of Application Specific Signal Processors) program, an ARPA/Tri-services program. A methodology for interfacing the VHDL performance models to algorithm partitioning tools is developed,

and proof of concept is provided by implementing three interface tools. The interface tools
developed include the Processor Characteristic Extraction Tool (PCET), the Architecture Characteristic Extraction Tool (ACET) and the Connectivity Extraction Tool (CONET). The PCET interfaces the processor characterization library to the partitioning tool, where as the ACET and CONET interface the VHDL structural models to the partitioning tool.

A set of VHDL procedures have also been developed to facilitate the writing of the application software for these performance models. These procedures are used to map the partitioned algorithm from the algorithm partitioning tool to the performance models.

For amma and appa

Acknowledgments

Numerous people were responsible for the success of this research work. I take this opportunity to thank them.

I would like to thank my advisor **Dr. F. G. Gray** for giving me the opportunity to work on this interesting research topic. Moreover his constant encouragement, advice, patience and support were invaluable to me.

I would like to thank my co-advisor **Dr. J. R. Armstrong** for his valuable ideas and suggestions.

I appreciate the efforts of **Dr. N. J. Davis, IV** for consenting to be on my graduate committee on a very short notice and for reviewing my thesis.

Special thanks to **Dr. Geoffrey Frank** and **Mr. Bud Clark** of the Research Triangle Institute for their invaluable ideas, suggestions, encouragement and motivation without which the work documented in this thesis would not have been completed.

Thanks are also due to my **colleagues and team members** on the RASSP project.

I would like to express my gratitude to the **students of the Computer Research Laboratory** at Virginia Tech for their company, help and friendship.

A very heartfelt thanks to all my **friends** in Blacksburg and elsewhere, whose constant help and encouragement kept me going.

I would like to thank my friend **Prakash** and my brother **Bharath** for cheering me on constantly.

This thesis is dedicated to my **parents** to whom I will forever be grateful for everything in my life.

Table of Contents

List of Figures

Chapter 1. Introduction

1.1 Motivation

Determining the optimum computer architecture configuration for a specific application or a generic algorithm is a difficult task. Performance modeling (see Chapter 2) is a widely used technique to efficiently and rapidly assess the ability of multiprocessor architectures to effectively execute a given algorithm. Since VHDL is increasing in popularity as a modeling language, there is significant interest in developing performance models in VHDL.

In a typical design environment, VHDL models of hardware components are interconnected to form structural models of the target multiprocessor architectures. Algorithm features are described in application specific tools. For example, SPW might be used to describe signal processing algorithms. An automated algorithm partitioning

tool might select a set of partitions to evaluate. For example APT developed at the Research Triangle Institute (RTI), Research Triangle Park, NC might be used. Finally, the hardware and software information needs to be combined to create an executable performance model. Figure 1.1 shows a block diagram of the environment.

Since several iterations may be needed before a suitable configuration is obtained, a set of tools that directly interfaces the VHDL processor models, the VHDL structural model, the SPW algorithm model and the VHDL performance model with the algorithm partitioning tool will significantly reduce the time and effort needed to prepare the various models. The primary goals of this thesis are to study the various models, determine the information that needs to be exchanged, and to develop tools to automatically extract the desired information from each model.

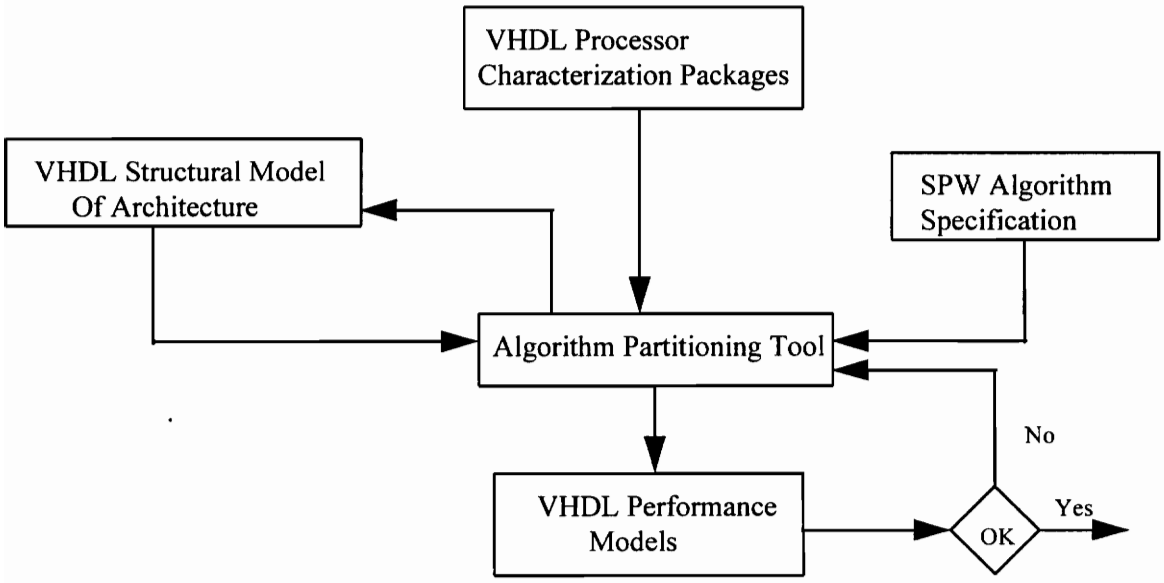


Figure 1.1 Design Environment

1.2 Task Description

This research work was done as part of the RASSP (Rapid Prototyping of Application Specific Signal Processors) project funded by an ARPA/TriServices contract. The objective is to interface VHDL performance models to an algorithm partitioning tool (APT) by a two-part process. First, a set of extraction tools is developed to interface the VHDL processor models and the VHDL structural models to the APT. These tools will be used to extract important characteristic information from the VHDL models which is required by the APT in order to determine candidate partitions of the algorithm for various multiprocessor architectures. Once an algorithm partition is developed by APT, the partitioning information is used to create a performance model. This model is executed to evaluate the effectiveness of the partition on that architecture. The second part of this thesis develops a set of target VHDL procedures for use in performance models.

1.3 Contributions

The following specific contributions were made to the state of art.

1.3.1 Developed a methodology for interfacing VHDL hardware models to algorithm partitioning tools.

A methodology was developed for interfacing the various VHDL models with an algorithm partitioning tool. This is described in great detail in Chapter 3. The structural models used to implement this methodology were developed by other persons working on the project. This methodology is not particular to a specific structural model or to a particular algorithm partitioning tool, and can be applied to the design of varied interfacing tools.

1.3.2 Constructed sample interface tools.

Three different interface tools were constructed and tested to demonstrate the proof of concept of the methodology proposed in Chapter 3. These were tested using two different example architectures. Results of these sample extractions have been included and documented. The tools are as follows.

1. Processor Characteristic Extraction Tool (PCET)

   This tool is used to extract processor characterization data from the Honeywell/Omniview Performance Model Library (PML)(see Chapter 2). The output is in the form of a processor characterization file suitable for use by the APT to determine system parameters such as number of processors needed and memory size requirements.
2. Architecture Characteristic Extraction Tool (ACET)

   This tool is used to extract architecture characterization data from a VHDL structural model with PML components. The output is in the form of a script file suitable for use by APT for determining candidate software partitions.
3. Connectivity Extraction Tool (CONET)

   This tool is used to extract connectivity information from a VHDL structural model built from PML components. This information is needed to construct the final performance model.

1.3.3 Developed VHDL subroutines for interfacing APT to VHDL performance models.

Five VHDL subroutines were developed to simplify the interface between APT and the VHDL performance models. These subroutines were tested by inserting them into existing working models. Identical outputs validated the subroutines. This was the first step to automate the labor intensive process of mapping the software tasks from the APT to the VHDL performance models. It significantly reduced the volume of VHDL code

involved in the performance models. These subroutines are described and documented in Chapter 4 of the thesis.

1.4 Research Approach

The customized primitives used in the VHDL structural models were derived from an earlier research project [3]. They were modified and further customized to meet the current research needs.

An approach for the development of tools to interface structural models to an APT follows.

- Perform a thorough study of the various components available in the libraries. The purpose of this step is to become familiar with the available components and resources and to recognize and understand the function of each parameter in each component. This is the most time consuming process in this thesis approach, but it is extremely necessary. This is because it has been noted time and again, that knowledge of details regarding working of the library components can greatly reduce the number of iterations required to produce satisfactory results.
- Determine the parametric information required to be extracted by the interface tools. A major contribution of this thesis is the study needed to define the interface parameters.
- Modify the VHDL components as necessary. The VHDL Tool Integration Platform's (VTIP) (see Chapter 2) VHDL analyzer does not accept all of the VHDL constructs. Some changes may need to be made to the library components to make them analyzable by VTIP. VTIP is a VHDL parser that is used to identify the various VHDL components and classify them in order to facilitate easy extraction of information from VHDL code. An important contribution of this thesis is also to identify the restrictions on VHDL code required for its compatibility with VTIP.
- Analyze the components and the structural model using the VTIP VHDL analyzer.

- Develop the extraction tools to meet design and format specifications. This is done using the DLS Browser (see Chapter 2) and the SPI (see Chapter 2) function calls.
- Test the output of the tools for accuracy. A structured system of testing has been developed to test these tools, and it is described in Chapter 3.
- If the results are unsatisfactory due to inappropriate parameters extracted or formatting errors, modify the extraction tool as required. Several iterations may need to be performed before satisfactory results are achieved.

This thesis uses the approach described earlier to construct extraction tools in a systematic manner. The PCET extracts the processor characteristics required by the APT to conduct the sizing analysis (see Chapter 2). This analysis is employed in early trade-off studies when architectures have not yet been selected. In a static analysis, it addresses issues such as how many processors of a given type and how much memory are required to execute a section of an algorithm or the entire algorithm within an allotted time. The ACET and CONET extract parameters necessary for the partitioning analysis (see Chapter 2). This analysis produces results which are then used to create a performance model that can be executed to identify major bottlenecks in the architecture.

The outputs produced by these tools are in the form of spreadsheet generating script files, which are formatted carefully to suit very specific requirements (see Chapter 3). These files can be invoked to automatically generate the spreadsheet and conduct the sizing and partitioning analyses. Earlier, these script files had to be generated manually. This was an extremely labor intensive task, especially since there usually needed to be several iterations for each architecture. Also, the special formatting required for these script files justified their automation. By invoking the files generated by these tools this process is made extremely simple, thereby saving significant time and labor for each iteration and speeding up the design process considerably.

1.5 Thesis Organization

Chapter 2. "Background and Literature Review" gives an introduction to the concept of performance modeling using VHDL. The VHDL Tool Integration Platform (VTIP) and the Algorithmic Partitioning Tool (APT) are introduced and a detailed description is provided. A section documenting the library components of the Performance Model Library (PML)[5], which is a library of hardware components developed by Honeywell, Inc. is also included.

Chapter 3. "Methodology of Interfacing VHDL to the APT" demonstrates the methodology used in this thesis to develop the extraction tools to interface the VHDL models to the APT.

Chapter 4. "VHDL procedures" describes the procedures that were developed to help interface the APT to the VHDL performance models.

Chapter 5. "Conclusions and Future Work" is the concluding chapter of the thesis and it explains the contributions made to the state of the art by the work documented in this thesis. The section on future work suggests areas that could be explored for further refinements and additions to the present work.

Chapter 2. Background and Literature review

2.1 About VHDL

One of the most widely used hardware description languages is the VHSIC Hardware Description Language (VHDL) [1]. VHDL was standardized by IEEE in 1987 as its 1076 standard [2]. The VHDL language enjoys growing popularity because it includes a powerful set of constructs for modeling, and because it provides a wide range of descriptive capability. VHDL supports behavioral descriptions of hardware at various degrees of abstraction from the digital system level to the gate level [3]. In the topmost level of abstraction, the system level, the digital system is defined by just its functionality, and bears little or no resemblance to actual implementation. The intermediate level of abstraction, the functional level, defines in slightly greater detail, the function of each component of the parent system, but still does not give the detailed

physical implementation. As the level of abstraction decreases, the level of detail and the resemblance to actual silicon implementation increases proportionally. Figure 2.1 shows the abstraction hierarchy of digital systems.

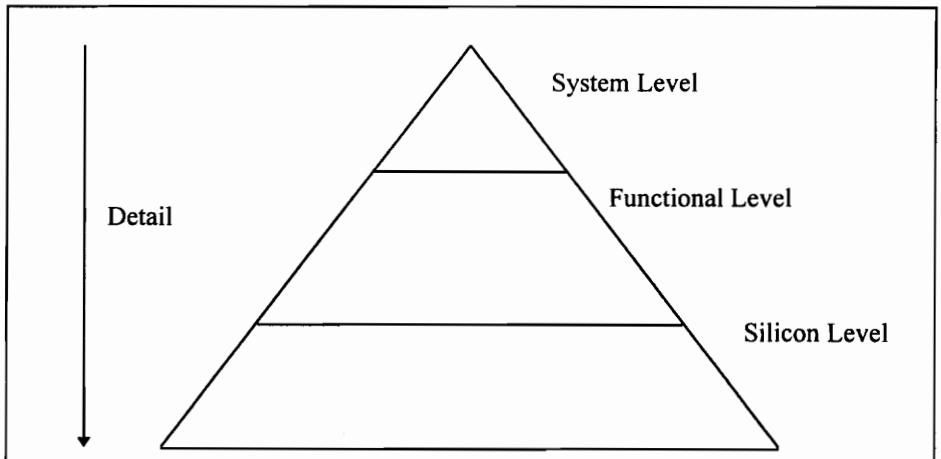


Figure 2.1 Abstraction Hierarchy

Work with VHDL has successfully been done in the areas of behavioral modeling, structural modeling and modeling for logic synthesis. The VHDL models are generally defined in two domains in the abstraction hierarchy: the structural domain and the behavioral domain. In the structural domain the model is represented as an interconnection of lower level primitives. In the behavioral domain, the model is described by defining its input and output response [1].

2.1.1 Structural Modeling

A structural form of hardware modeling implies a design decomposition process [1]. This is because, at any chosen level, the system model is composed of an interconnection of the primitives defined for that level. These primitives are frequently

defined with more primitives in the next lower level of the hierarchy. At the lowest level of this structure is the primitive that has been specified in terms of its behavior. Thus, a structural design can be compared to a tree as shown in Figure 2.3, with the different levels of the tree corresponding to the different levels of the hierarchy. The leaves of the tree at the lowest level are behavioral models of the lowest-level design components. An example of a structural model is shown in Figure 2.2.

```
entity TWO_FIVE is
  port(F: in BIT_VECTOR(4 downto 0);
       D: out BIT);
end TWO_FIVE;

entity AND5 is
  port(I1,I2,I3,I4,I5: in BIT; O: out BIT);
end AND5;

architecture BEHAVIOR of AND5 is
begin
  O <= (I1 and I2 and I3 and I4 and I5) after 1 ns;
end BEHAVIOR;


entity INV is
  port(I: in BIT; O: out BIT);
end INV;

architecture BEHAVIOR of INV is
begin
  O <= (not I) after 1 ns;
end BEHAVIOR;


entity OR10 is
  port(I: in BIT_VECTOR(0 to 9); O: out BIT);
end OR10;

architecture BEHAVIOR of OR10 is
```

```
begin
  O <= (I(9) or I(8) or I(7) or I(6) or I(5) or I(4) or I(3) or
        I(2) or I(1) or I(0)) after 1 ns;
end BEHAVIOR;

architecture STRUCTURAL of TWO_FIVE is

signal INT: BIT_VECTOR(0 to 9);
signal NOTF: BIT_VECTOR(4 downto 0);

component ANDFIVE
  port(I1,I2,I3,I4,I5: in BIT; O: out BIT);
end component;
component INVERT
  port(I: in BIT; O: out BIT);
end component;
component ORTEN
  port(I: in BIT_VECTOR(0 to 9); O: out BIT);
end component;

for all: ANDFIVE use entity AND5(BEHAVIOR);
for all: INVERT use entity INV(BEHAVIOR);
for all: ORTEN use entity OR10(BEHAVIOR);

begin
  I1:INVERT
    port map(F(4),NOTF(4));
  I2:INVERT
    port map(F(3),NOTF(3));
  I3:INVERT
    port map(F(2),NOTF(2));
  I4:INVERT
    port map(F(1),NOTF(1));
  I5:INVERT
    port map(F(0),NOTF(0));
  A0:ANDFIVE
    port map(NOTF(4),NOTF(3),NOTF(2),F(1),F(0),INT(0));
  A1:ANDFIVE
    port map(NOTF(4),NOTF(3),NOTF(1),F(2),F(0),INT(1));
  A2:ANDFIVE
    port map(NOTF(4),NOTF(3),NOTF(0),F(2),F(1),INT(2));
  A3:ANDFIVE
```

```
    port map(NOTF(4),NOTF(2),NOTF(1),F(3),F(0),INT(3));
  A4:ANDFIVE
    port map(NOTF(4),NOTF(2),NOTF(0),F(3),F(1),INT(4));
  A5:ANDFIVE
    port map(NOTF(4),NOTF(1),NOTF(0),F(3),F(2),INT(5));
  A6:ANDFIVE
    port map(NOTF(3),NOTF(2),NOTF(1),F(4),F(0),INT(6));
  A7:ANDFIVE
    port map(NOTF(3),NOTF(2),NOTF(0),F(4),F(1),INT(7));
  A8:ANDFIVE
    port map(NOTF(3),NOTF(1),NOTF(0),F(4),F(2),INT(8));
  A9:ANDFIVE
    port map(NOTF(2),NOTF(1),NOTF(0),F(4),F(3),INT(9));
  O1:ORTEN
    port map(INT,D);
end STRUCTURAL;
```

Figure 2.2 Example of Structural Model

Figure 2.2 shows that the entity TWO_FIVE is defined using the lower level entities INV, OR10 and AND5. These entities, which are the leaves of the tree are described in terms of their behavior. This is shown in Figure 2.3.

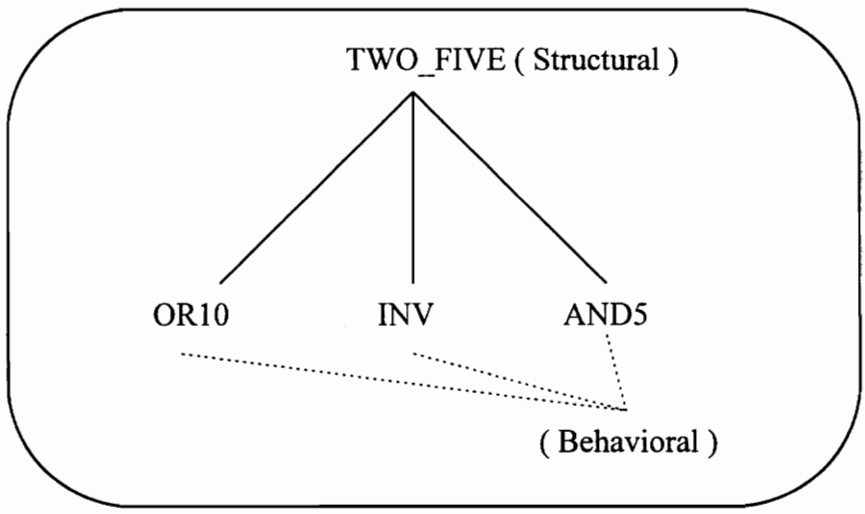


Figure 2.3 Tree structure of structural model

2.1.2 Behavior Modeling

The behavioral model differs from the structural model in the manner in which the architectural body is represented. In this domain the component is described by defining its input/output response. It is not structural and the responses are defined in the form of procedures and functions. The behavioral descriptions of systems are frequently divided into two types: algorithmic and data flow.

Algorithmic: "A behavioral description in which the procedure defining the input/output response is not meant to imply any particular physical implementation" [1]. An algorithmic model of the entity defined in Figure 2.2, is shown in Figure 2.4.

```
architecture ALGORITHMIC of TWO_FIVE is
begin
 P1:process(F)
 variable COUNT: INTEGER range 0 to 5;
 begin
  COUNT:=0;
  for I in 0 to 4 loop
   if F(I)='1' then
    COUNT:=COUNT+1;
   end if;
  end loop;
  if COUNT=2 then D<='1' after 3 ns;
        else D<='0' after 3 ns;
  end if;
 end process P1;
end ALGORITHMIC;
```

Figure 2.4 Example for Algorithmic model

Data flow: "A behavioral description in which the data dependencies in the description match those in a real implementation" [1].The Data Flow model of the entity defined in Figure 2.2, is shown in Figure 2.5.

```
architecture DATA_FLOW of TWO_FIVE is
begin
  D<=(not F(4) and not F(3) and not F(2) and F(1) and F(0)) or
     (not F(4) and not F(3) and F(2) and not F(1) and F(0)) or
     (not F(4) and not F(3) and F(2) and F(1) and not F(0)) or
     (not F(4) and F(3) and not F(2) and not F(1) and F(0)) or
     (not F(4) and F(3) and not F(2) and F(1) and not F(0)) or
     (not F(4) and F(3) and F(2) and not F(1) and not F(0)) or
     (F(4) and not F(3) and not F(2) and not F(1) and F(0)) or
     (F(4) and not F(3) and not F(2) and F(1) and not F(0)) or
     (F(4) and not F(3) and F(2) and not F(1) and not F(0)) or
     (F(4) and F(3) and not F(2) and not F(1) and not F(0))
     after 3 ns;
end DATA_FLOW;
```

Figure 2.5 Example of Data Flow model

When the designer creates a behavioral model of the system, the inputs and outputs to the system, called ports [2] are defined along with the generics in the entity declaration. The architectural body for a behavioral model consists of one or more processes that run concurrently. These processes contain signal assignments, loops and other constructs that characterize the input-output relationship of the system.

2.2 Performance Modeling using VHDL

The design and development of high performance computing systems is becoming increasingly complex. Performance modeling is often used to characterize the system level of abstraction in a digital system. A performance model is applied in the early stages

of system development, where it supports early trade-off studies when the actual detailed design is not yet available. "Performance modeling can aid evaluation of design alternatives, capture design decisions and assumptions, examine system behavior at boundary conditions and help determine bottlenecks and overdesign." [4]. A performance model expressed in VHDL serves as a simulatable specification and supports performance validation. It can be used for capturing and documenting architectural level designs, and as a testbed for architectural performance analysis studies.

Performance modeling simulates system performance without the unnecessary details. It is a means of speeding up simulations by ignoring the superfluous details unnecessary to the system performance, while in no way compromising fidelity to system response. System performance can be measured very effectively in this manner, especially in large and complex systems where the details usually bog down simulation speeds, thereby resulting in wastage of time and resources.

There are three basic statistics collected as part of the system performance evaluation: latency, utilization and throughput. These are called performance metrics.

- **Latency** - The time it takes for a token to get from point A to point B. This includes the time delays encountered by the token due to traffic, bus contentions etc.
- **Utilization** - Time busy / total simulation time
- **Throughput** - Data processed / time period

Performance and trade-off studies will be interpreted with respect to these metrics.

2.3 Performance Model Library (PML)

The VHDL Performance Model Library has been developed by Honeywell, Inc. and Omniview[5]. The PML_02a version of the library has been used for the present research work. The Abstract VHDL model library is composed of the following sub-libraries:

- **The GEN Library:** Contains all the models (entities and architectures) and packages other than those related to the processor model.
- **The PROC Library:** Contains all the models (entities and architectures) and packages related to the processor model.
- **Miscellaneous Tools and Resources:** These include the postprocessing tools to generate the activity and latency plots, UNIX shell scripts to run and monitor simulations and important documentation of the library. The structure is shown in Figure 2.6.

2.3.1 The GEN Library

The GEN library has two sub-libraries: *Packages* library and *Components* library. The *Packages* library contains important packages which define the basic means of communication in a model, routines which track simulation statistics, etc. The *Components* library contains primitive components suitable for erecting performance models at the system level of abstraction, like I/O devices, memory devices, etc.

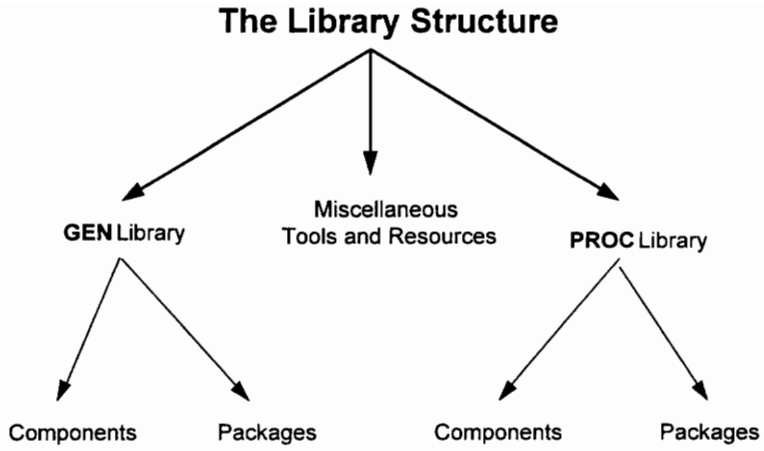


Figure 2.6 The Abstract VHDL Architecture Library Structure

2.3.1.1 GEN Library Packages

Important GEN Packages

- Base_Types Package
- Statistics Package
- Utility Package
- Math Package
- Strings Package
- GDL Package

2.3.1.1.1 Base-Types Package

This package contains the fundamental type declarations and definitions that are used throughout the library. This includes the token structure shown in Figure 2.7 and some token manipulation functions. Communication between all devices and components is achieved using these tokens. The record structure for this token is shown in the figure below.

```
            -- utoken : the basic unresolved token type.
TYPE utoken IS
  RECORD
            --  user fields
    parm1_real      : REAL;
    parm2_real      : REAL;
    parm1_int       : INTEGER;
    parm2_int       : INTEGER;
            --  control flow
    destination     : name_type;
            -- the name of the destination device
    source          : name_type;
            -- the name of the source device
    t_type          : token_type;
            -- token_type: DATATOKEN, READTOKEN
            -- WRITETOKEN, CONTROLTOKEN
            -- performance fields
    size            : data_size;
            --  Size or number of this token. For
            --  instance,if this were a `memory read'
            --  token, size would be construed to mean
            --  the number of words to read from the
            --  memory.
    value           : INTEGER;
            --  The token might have a particular value.
            --  For instance, a memory write token to a
            --  local address might actually place a
            --  value there.

            --  token tracking or statistics fields
    id              : uGIDType;
            -- Unique token identification number
    start_time      : TIME;
            -- time when token was created

            --  communication fields
    priority        : INTEGER;
            -- The message might have a priority
            -- associated with it in order to resolve
            -- contention.
    state           : State_Type;
            -- Indicates state of signal (busy, idle,
```

```
        -- etc)Used by brf.
protocol        : Protocol_Type;
        -- Bus Protocol to use for this token. Used
        -- by bus resolution function (brf).

        --  user communication tracking and control
        --  fields
collisions      : INTEGER;
        --  There might be a collision counter to
        --  track how many times the token contended
        --  with other tokens for the same resource.
retries         : INTEGER;
        --  This track the number of times the token
        --  tried to obtain a particular resource
route           : name_type;
        -- In case the token traverses multiple
        -- resources to get to its destination,this
        -- field contains the route string.

END RECORD;
```

Figure 2.7 The Basic Token Structure

2.3.1.2 GEN Library Component Devices (Leaf Cells)

All the device entities in the GEN *components* library have the following common characteristics (only the most commonly occurring parameters are explained now, a detailed explanation of the other parameters is given while explaining the GEN library components):

- They have an "INST" (instance) generic which is the name of the device. Each device has a unique name. If the token originates from that device then the source field of the token will contain the instance name.
- The TRACE generic is another common generic among all the component devices. With TRACE set to TRUE, the device will track statistics (latency and utilization) and will write the statistics to Standard Output (STD_OUTPUT) trace file. If

TRACE is set to FALSE then the device does not write its statistics to the Standard Output (STD_OUTPUT) trace file.

- Most devices have a queue attached to the output port of the device. The depth of the queue, i.e. how many tokens it can store before it overflows, is decided by the QUEUE_DEPTH generic.
- Most devices have latency and throughput distribution information associated with them by means of the LATENCY_INFO and THROUGHPUT_INFO generics. The throughput is given in hertz and is multiplied by the UNIT generic to compute data rates.
- Most devices have ports which are of inout mode and of type token (the basic token structure type).

2.3.1.2.1 Input Device [5]

This device generates tokens with a specified distribution, which stimulate the whole model under study. It is purely a data source, and can produce data with different statistical distributions like UNIFORM, POISSON, GUASSIAN, etc. Refer to Appendix A for the entity declaration.

2.3.1.2.2 Outdevice[5]

One way to decompose a model is into the stimulus or inputs, the unit under study or design, and the outputs or its response. This device consumes or sinks the tokens from the model. This outdevice component is intended to accept the responses of the system under study. For instance, hard disk, display, etc are examples of outdevices.

2.3.1.2.3 Memory [5]

The memory device (also called an iodevice) element is basically a programmable delay element with the ability to return tokens to the sender. This memory component is intended to manipulate stimulus as part of the system under study or design, and provide some response to the environment. For instance, if a processor requests data from the memory, this module waits for a period of time equal to the read access time for that amount of data and sends an acknowledgment. Refer to Appendix A for the entity declaration.

2.3.1.2.4 Bus Interface Unit (BIU) [5]

The Bus Interface Unit is a programmable delay and a token filter. Figure 2.8 shows the schematic of the BIU.

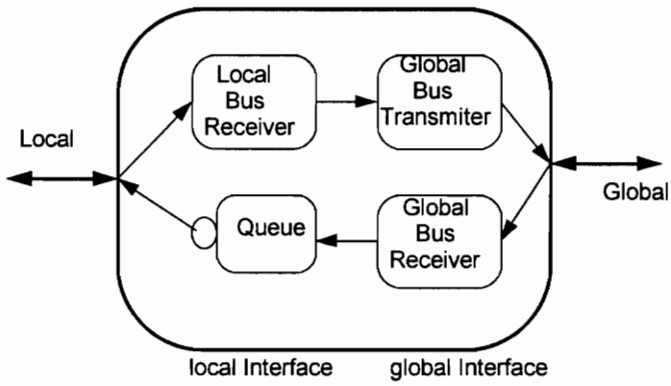


Figure 2.8 Internal Schematic of the BIU

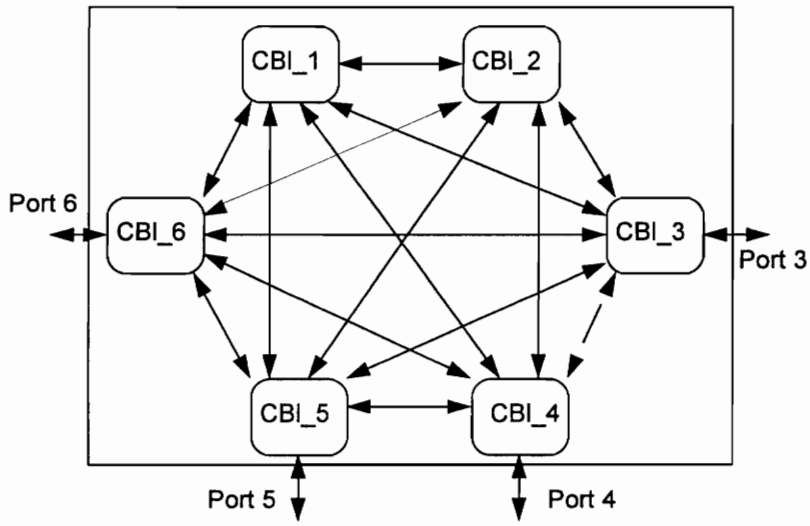
This device is designed in two parts. One is the "local" side, and is intended to interface with the local components like processor, memory etc. The other

side is the "global" side with which the different BIUs are connected to a global bus. The local interface has two logical components. The local bus receiver handles outgoing messages. The queue handles tokens which have been received from the global side. The global interface also has two logical components. The global bus transmitter handles outgoing messages, the global bus receiver receives tokens from the global bus. Both incoming and outgoing messages have latency delays associated with them. Refer to Appendix A for the entity declaration.

2.3.1.2.5 Crossbar Interconnection Device [3]

This device is used to provide routing of messages between devices. It has six ports of type "token" to which external devices are connected. The crossbar is a structural model of 6 fully inter-connected individual routers called crossblocks. This structure is shown in Figure 2.9. Each router has one external bi-directional port and 6 internal bi-directional ports as shown in Figure 2.10.

Each crossblock has two processes. The Receiver (Rx_Proc) receives tokens from the external port and directs them to the appropriate internal port using the route field of the token and a built-in routing table. The Transmitter(Tx_Proc) receives tokens from the internal signals and sends them to the external port. Refer to Appendix A for the entity declaration of the crossbar module and the crossblock module.



•The Crossbar is modelled as a fully inter-connected structural model of individual routers called CrossBlocks

Figure 2.9 Crossbar Interconnection Device

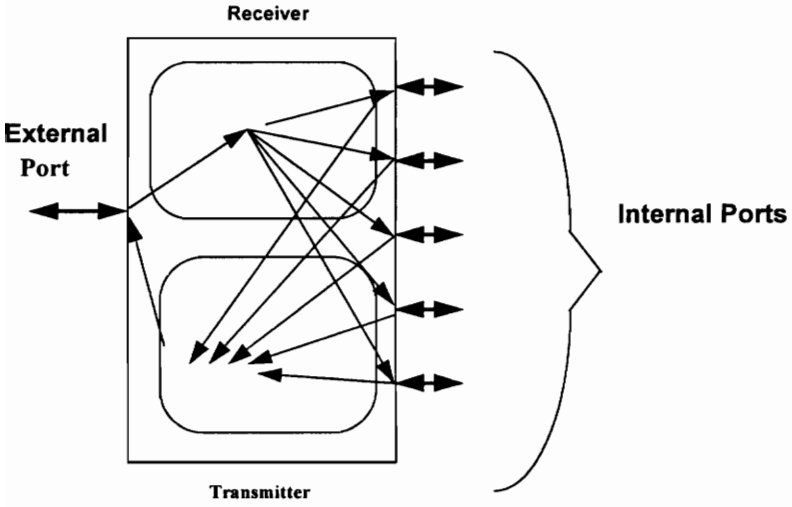


Figure 2.10 Internal Schematic of Crossbar Component

2.3.1.2.6 Biu_four component [3]

Figure 2.11 shows the schematic for a biu_four component. The biu_four was developed to connect four different devices to a common bus. This component is a structural interconnection of four BIUs with four external ports labeled TOP_Local, RIGHT_Local, LEFT_Local and BOTTOM_Local. The external ports are connected to the local ports of each of four BIU components labeled BIU_TOP, BIU_RIGHT, BIU_LEFT and BIU_DOWN respectively. The global parts of all four BIUs are connected to a common bus labeled BUS. The architecture for this component is described in Appendix A.

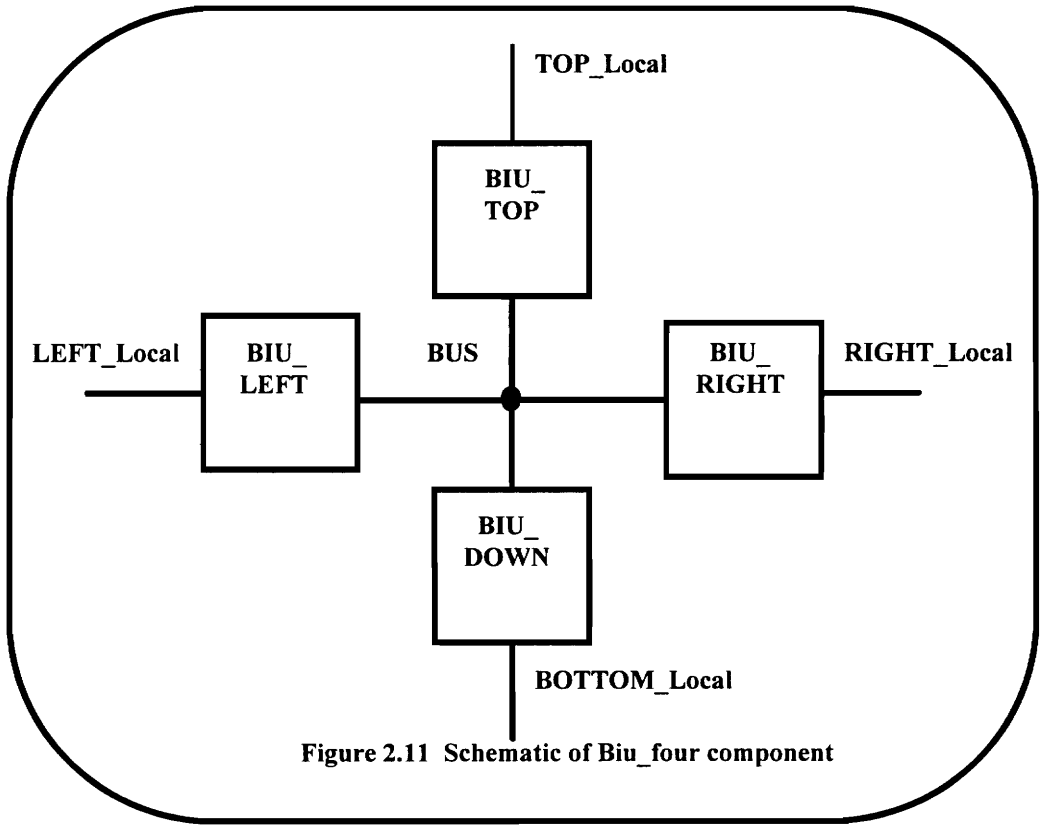


Figure 2.11 Schematic of Biu_four component

2.3.1.2.7 Biu_star component

The biu_star component is similar to the biu_four component but is a structural interconnection of *three* individual BIUs. All the generics are similar to those of the biu_four component.

2.4 CLSI VTIP, DLS and SPI [6,7,8]

The CLSI-VTIP cad tool is used to extract details from VHDL models and store them in the form of data structures. In this application, the input to the CLSI VTIP is the VHDL code obtained from the PML library and the structural model developed by the user. The analyzer processes the VHDL description, checks for syntactic and semantic errors and stores the data structure in an internal format in the Design Library System (DLS) [6,7].

The DLS consists of two fundamental components _ one abstract, the other concrete. The abstract component is the DLS Data Definition, which specifies the data elements, objects and structures that are supported by the DLS, together with the abstract operations that can be performed upon them. The concrete component is the Software Procedural Interface (SPI), which is a software implementation of the DLS Data Definition that supports the development of applications.

One can visualize the VHDL code that was parsed and stored in the DLS as a tree-like structure, in which the root is a string that specifies the name of the file containing the VHDL code. This string is called the primary name of the library unit. Refer to Figure 2.12. The entity and the architecture declarations are stored in the two nodes below the

library unit. The input and output ports and the generic values are all stored in the entity declaration branch. In the architectural declaration branch the different component instantiations are stored for structural models, and the different processes, procedures, function declarations or statements are stored for behavioral models. The branches keep on spreading until all the syntactic elements are covered. Figure 2.12 shows the DLS structure for a typical structural architecture, a special case being entity TWO_FIVE from Figure 2.2. It also shows the tree structure for a typical algorithmic architecture, a special case being entity TWO_FIVE from Figure 2.4. Note that the structure is not a tree because branches converge and intertwine. But it is tree-like at the top levels.

For the library unit two_five (structural), the node "identifiers" contains all the labels, signal names and component names in the architecture. These include I1-I5, A0-A9, O1, 'NOTF', 'INT', 'ANDFIVE', 'INVERT' and 'ORTEN'. The node signal declarations contains information about the two signals 'NOTF' and 'INT'. This information includes the type of signal and mode. The type of signal for signal 'INT' would be BIT_VECTOR and the mode is IN. The node "component declarations" contains information about each component name, and the IN and OUT signals associated with it. For the component 'ANDFIVE', the signals associated with it would be I1-I5, and 'O'. The "label declarations" and the "component instantiations" nodes both contain information about each component that is instantiated in the architecture. For instance, the label name of the first component would be 'I1', and the corresponding component name 'INVERT'.

For the library unit two_five (algorithmic), the node "signals" contains information about the two signals 'F' and 'D' which are used in the body of the

procedure. The node "process statements" has information about the sensitivity list and the statement list. The "sensitivity list" contains information about 'COUNT'. The "statement list" has FOR, IF and Variable assignments in it. The FOR node has the following information about the loop variable 'I'. It is of type INTEGER, and its lower and upper limits are 1 and 4 respectively. The Variable assignment again contains information about the value assigned to 'COUNT'.

The SPI is the fundamental part of the DLS [8]. The SPI consists of data types and callable routines that implement the data types and operations of the DLS. The SPI provides layers of data types and routines to support the layers (primitive and generic data defined by the DLS model) of the DLS Data Definition. The SPI is implemented in the C programming language. These routines provide high level functions that can be implemented in terms of lower-level functions. They support addition of extra information to nodes, evaluation of expressions, symbol searches and generalized queries of DLS structures. These SPI routines are made accessible by compiling the C source files with the SPI library functions.

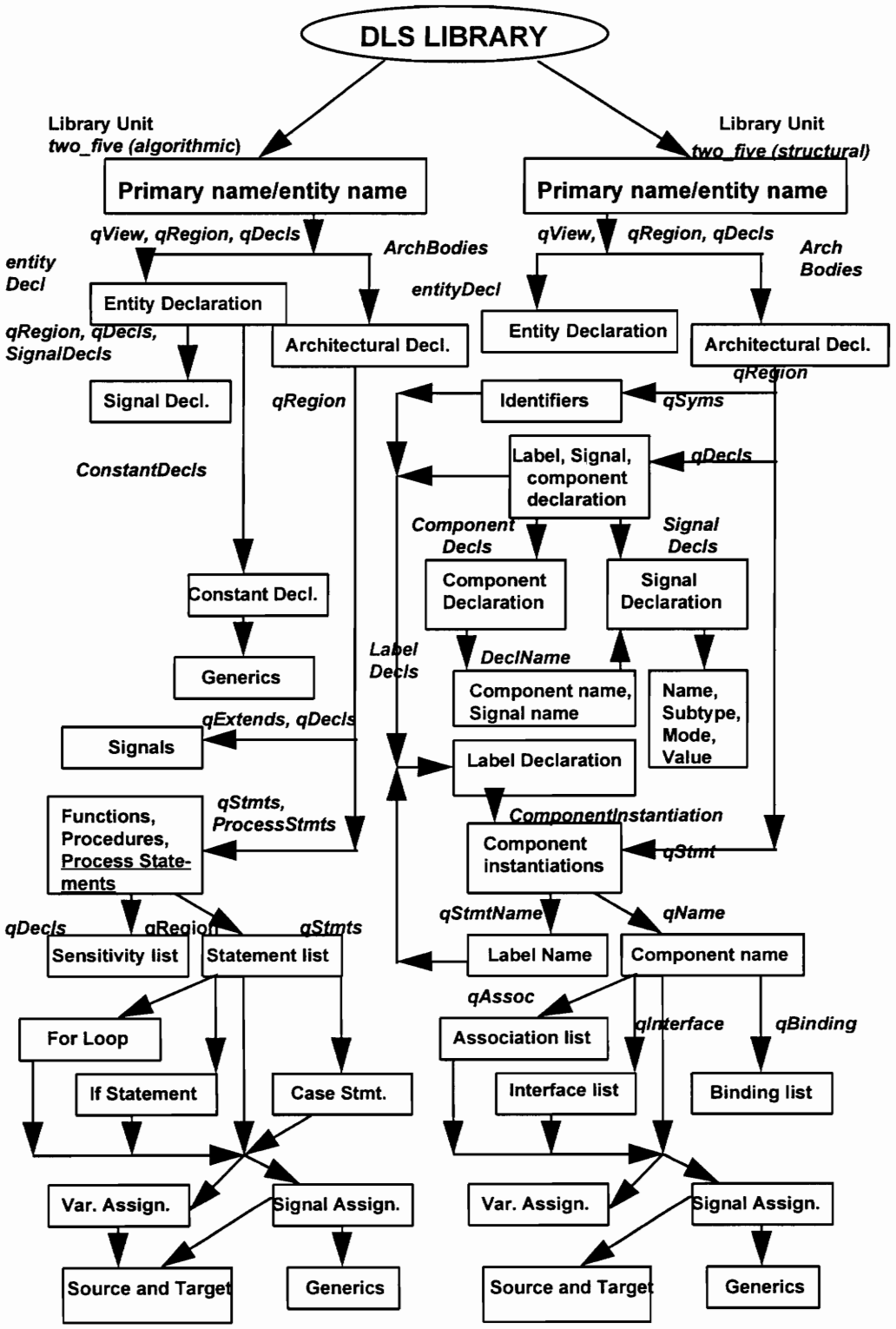


Figure 2.12 The VTIP Design Library System (DLS)

The functions in SPI match the data structure names in DLS. These can be viewed using *dlsbrowse* and are easy to use. The labels on the arcs in Figure 2.12 denote the SPI function call names that are required to reach that node or branch in the tree structure. Figure 2.13 contains some example C source code that uses some of the SPI functions. Most of the routines used to access different ObjectTypes have the same names as the ObjectTypes themselves. The first routine that should be called before any other routine is called is the *OpenDLS*. This opens and initializes the Library System and the SPI. The value of a parameter LibNamea is assigned to be equal to *dls_design_library*. A variable "Liba" is then given a value by calling an SPI routine *NewLibrarySymbol*, with LibNamea supplied as a parameter. The function *NewLibrarySymbol* creates a Library symbol node that serves as a parameter to procedure *OpenLibrary*, which makes available the library units within it. An appropriate library unit is opened using the function *OpenUnit*. The parameters passed to this function are **Library** (The library containing the unit to open, e.g. Liba), **UnitName** [8] (entity-name), **UnitQual** [8](arch-name), **UnitView** [8](The view type of the unit, e.g. VHDLView) and **Mode** [8](The type of access allowed, e.g. AugmentMode). The **Mode** specifies how to open the unit.It may be CreateMode to create a new unit, AugmentMode to modify an existing unit, ReadMode to read an existing unit, or TempMode to temporarily modify an existing unit. Using the function call *qView*, the VHDLView node is obtained and from here *qRegion* is used to access the library-level Region Node. We are now at the root of the tree for this library unit.

```
/**********************OpenDLS*****************/

OpenDLS();
SetReportLevel(Warning);
SetAbortLevel(Error);
strcpy(LibNamea,"dls_design_library");

/*******************Open  library "LibName"*****************/

 Liba = NewLibrarySymbol(LibNamea);
 OpenLibrary(Liba);


/*******************Open the Unit *******************/

 Unit = OpenUnit(Liba, entity_name, arch_name, VHDLView, AugmentMode);

/*** Locate the region corresponding to the design unit declaration ***/

 region = qRegion(qView(Unit));    /* get the library-level region */
```

Figure 2.13 A portion of C code to illustrate the use of SPI functions

The DLS Browser, as mentioned above, is a screen-oriented utility that allows the user to examine the DLS library units. The Browser enables a user to open library units and to traverse and examine the nodes, lists and data structures within a library unit. The node, list-item or unit to be traversed can be chosen by using the up and down arrows. The right arrow is used to move one level down the data structure of the dls_design_library, whereas the left arrow is used to move one level up the structure. Figure 2.14 shows the DLS Browser screen for the architectural declarations node of the structural model in Figure 2.2. It can be seen from the corresponding DLS data structure in Figure 2.12, that the three important nodes contained in this Architectural Declaration

node are *qSyms*, *qDecls* and *qStmts*. These three nodes are visible on the DLS Browser screen as separate nodes, and can be entered by using the arrows. The node **qLibUnit** has information about the library unit being opened. This will include the entity name, architecture name, VHDL filename etc. The node **qParent** has information about the parent node from which this node was derived. The node **qSyms** contains all the labels, signal names and component names in the architecture. These include I1-I5, A0-A9, O1, 'NOTF', 'INT', 'ANDFIVE', 'INVERT' and 'ORTEN'. The node **qDecls** contains information about the signal, label and component declarations. This includes the two signals 'NOTF' and 'INT', their type and mode. The type of signal for signal 'INT' would be BIT_VECTOR and the mode is IN. **qDecls** also contains information about each component name, and the IN and OUT signals associated with it. For the component 'ANDFIVE', the signals associated with it would be I1-I5, and 'O'. The node **qStmts** contains information about each component that is instantiated in the architecture. For instance, the label name of the first component would be 'I1', and the corresponding component name 'INVERT'.

The number in paranthesis following each attribute is the number of items in that list, and the numbers following each of these list numbers are internal pointers for the DLS Browser, and are not used by the user.

```
        DLS Browser 2.9.4.0

Node: 1:10        Block                    dls_design_lib:two_five-structura-
Attribute    Value    Kind        Miscellaneous

   qLibUnit:     1:1      LibraryUnit    0:1, "VHDL 1076-1987 Anal-", ...
   qParent:      1:7      ArchitectureBo- 72, 1:6, 1:10*, [1](-)
   qAttrs:      [0]()
   qToolInfo:    0
-->qSyms:       [21](1:11*, 1:13*, 1:15*, 1:17*, 1:19*, 1:21*, 1:23*, -)
   qExtends:     2:9      Block          [2](2:10*, 2:12*), Null, ...
   qDecls:      [21](1:12*, 1:14*, 1:16*, 1:18*, 1:20*, 1:22*, 1:24*, -)
   qControl:     Null
   qStmts:      [16](1:53*, 1:54*, 1:55*, 1:56*, 1:57*, 1:58*, 1:59*, -)
```

Figure 2.14 DLS Browser screen format

With the help of the SPI functions the generics from the VHDL models are obtained, along with the connectivity information for the various transfer components. Recall that the arc labels in Figure 2.12 denote the SPI function call names that are required to reach that node or branch in the tree structure. In order to reach a particular name or generic in the tree structure, all that is required is to simply trace the arc labels from the root to the item desired. For example, if the component name 'andfive' needs to be extracted, the following steps need to be followed. The name is stored in the node declaration name under the node Component Declaration. The path to the node needs to be traced with the DLS Browser using the up, down, left and right arrows as described earlier. Once this is done, the DLS library is opened using the *Openlibrary* function call,

as described in Figure 2.13. Then the required entity node is opened using the OpenUnit call. The SPI code for this step is

Unit = OpenUnit(Liba, two_five, structural, VHDLView, AugmentMode);

The Architecture body can be reached using the following section of SPI code.

ArchNode = ArchBodies(qDecls(qRegion(qView(Unit))));

After doing this, the required component name can be reached by simply tracing the tree structure from the architecture body node down to the Component Declaration node. Assuming that the component name needs to be copied into a string variable called **CompName**, the SPI code to do this is shown below.

strcpy(CompName, DeclName(ComponentDecls(qDecls(qRegion(ArchNode)))));

Similarly any Signal, name or generic can be extracted by using the DLS Browser and SPI function calls as described above.

SPI functions are called by the application program to extract information. In our application, the information extracted using these functions consists of names of component instantiations, the names of generics associated with each component along with the values mapped onto them, and the names of signals mapped onto the ports along with the components to which they are connected.

2.5 Algorithm Partitioning Tool (APT) [9]

"The Algorithm Partitioning Tool (APT) is designed to facilitate the transition from functional algorithm description and HDL or schematic hardware architecture and component description to a dynamic performance model with the algorithm mapped onto the architecture."[9] In the prototype used in the current application, the algorithm

characteristics are captured by the Signal Processing Worksystem (SPW) of the Cadence ALTA Group; component characteristics are captured from VHDL characteristic files and Architecture connectivity is captured either from a VHDL structural model or a network II.5 (CACI Products Company) generated schematic. This prototype employs Access Technology's 2020 spreadsheet.

Refering to Figure 2.15, the main steps in employing the APT are :

- Extract the processor characterization data from the PML processor library using the Processor Characteristic Extraction Tool (PCET).
- Determine the required numbers of processors to run a given application from the spreadsheet analysis.
- Build a VHDL structural model of the architecture.
- Automatically create the analysis spreadsheet using the Architecture Characteristic Extraction Tool (ACET).
- Use the spreadsheet to obtain the partitioning of the software application algorithm.
- Given the partitioned software application specifications, convert the software to PML code using the Connectivity Extraction Tool (CONET) and VHDL procedures, and map it onto the specific processors in the hardware architecture model.
- Evaluate the performance of the model. If the results are not satisfactory, change the partitioning of the software or the hardware model and repeat the steps.

The spreadsheet is employed in two modes which are referred to as "sizing" and "mapping". Sizing mode is employed in early trade studies when architectures have not yet been defined. It is a static analysis, and it addresses issues such as how many processors of a given type and how much memory are required to execute a primitive, a

section of the algorithm, or the entire algorithm within the allotted time. The Mapping mode is employed for static analysis of architectures already defined and to analyze alternate mappings of an algorithm onto the architecture. The spreadsheet output that is produced in this mode is used to construct a dynamic simulation model that can be used to assess additional issues such as latency and resource contention.

Figure 2.15 shows the Architecture trade off environment of the APT.

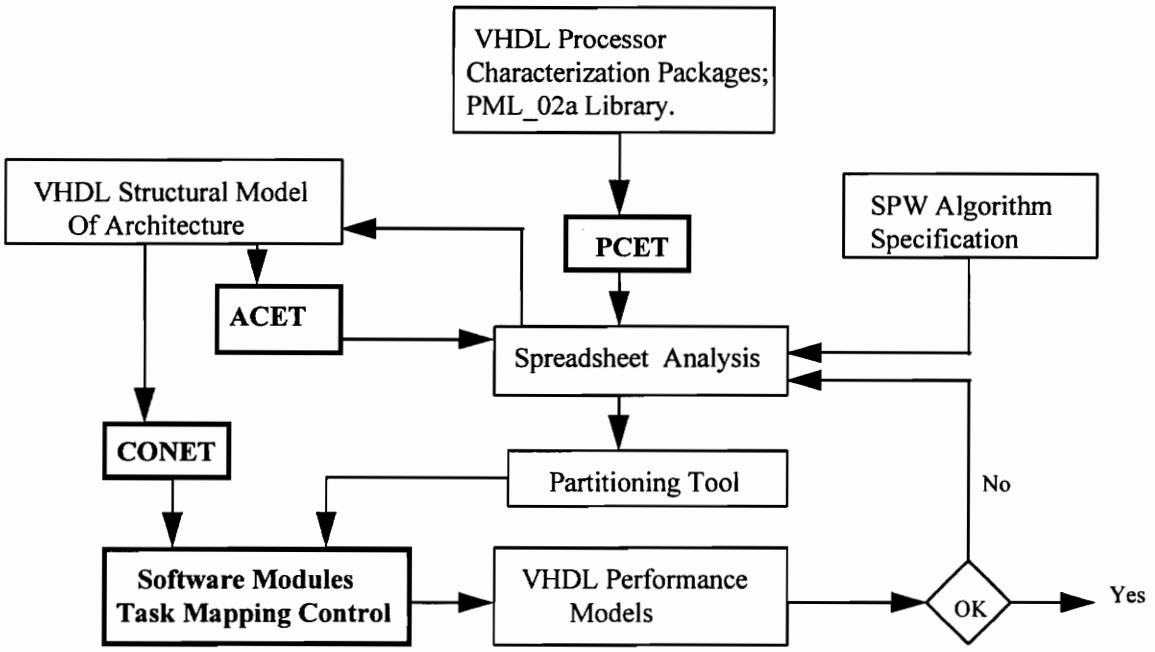


Figure 2.15 Architecture Trade-off Environment

Chapters 3 and 4 describe the PCET, ACET and CONET tools and the Software Modules Task Mapping Control development respectively. These modules form the main body of research work conducted in this thesis.

Chapter 3. Interfacing VHDL to APT

3.1 Methodology for Developing Interfaces between VHDL code and Partitioning Tools

This chapter deals with a methodology for developing interfaces between VHDL code and an algorithm partitioning tool. The process of designing the interface tools-Processor Characteristic Extraction Tool (PCET), Architecture Characteristic Extraction Tool (ACET) and Connectivity Extraction Tool (CONET) all followed a common methodology. This is outlined in Figure 3.1.

1. Study the VHDL source code and Target file format.
2. Analyze the VHDL source code using the VTIP VHDL analyzer.
    - Set up VTIP in the work environment and create directories for the VTIP design analyzer output.

- Make modifications to the VHDL code. VTIP does not recognize all valid VHDL constructs.
- Analyze the VHDL source code.

3. Determine any restrictions on source code and determine parameters to be extracted.
4. Find the path to the required parameters using the VTIP DLS browser.
5. Extract the parameters using SPI.
6. Format the output.
7. Test the output of the tools for accuracy.

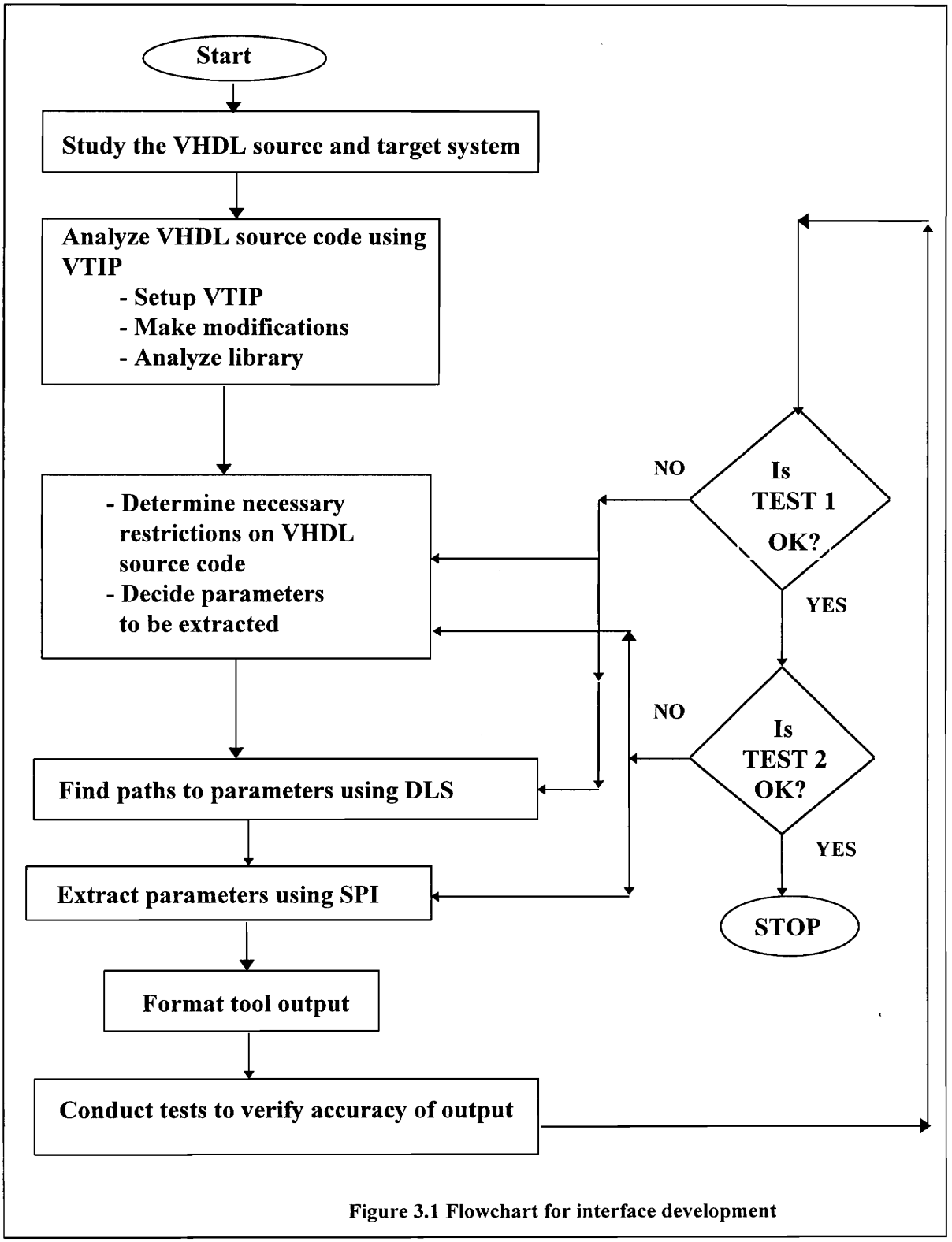


Figure 3.1 Flowchart for interface development

3.1.1 Study of the VHDL source code and target system

The VHDL source code needs to be studied in detail. Here, the source code is the PML library and it needs to be studied in detail in order to understand the working of the various components described in it, like the memory, processor, processor scheduler etc. The target system needs to be studied in order to gain an understanding of the interface required to be developed.

3.1.2 Analysis of the library using the VTIP VHDL analyzer

This process can be decomposed into the following three steps.

- Set up VTIP in the work environment and creating directories for the VTIP design analyzer.
- Make modifications in the VHDL source code.
- Analyze the VHDL source code.

3.1.2.1 Setting up VTIP in the work environment and creating directories for the VTIP design analyzer

VTIP needs to be setup in the work environment and some physical directories need to be created to contain the PML library. In order to setup VTIP in the work environment, some environment variables need to be added in the startup shell script (see Appendix B). The PML library is organized into three different directories which need to be created. Also, a directory for the VTIP analyzer needs to be created (see Appendix B).

3.1.2.2 Making modifications in the VHDL source code

The VTIP VHDL analyzer does not recognize the full syntax of the PML models. Hence, a few modifications need to be made to the PML library for the VTIP analyzer to successfully analyze the library.

3.1.2.2.1 Modifications to Processor Models

1. Processor-c.vhdl

All statements in the file "processor-c.vhdl" in the PROC library that mention the modifier "threads" need to be commented out. Each such line is a FOR statement, so the corresponding END FOR statements also need to be removed. Also, the label 'sw' needs to be changed to 'sw1'. The original code appears as follows.

```
FOR behavior
        FOR threads(TASK_1)
            FOR cmp
                FOR sw:application_sw
                  USE ENTITY proc.application_sw(receive_interrupt_sw);
                END FOR;
            END FOR;
        END FOR;
END FOR;
```

After the modifications described above, the code appears as follows.

```
FOR behavior
      --*   FOR threads(TASK_1)
            --*   FOR cmp
                  FOR sw1:application_sw
                    USE ENTITY proc.application_sw(receive_interrupt_sw);
                  END FOR;
            --*END FOR;
      --*   END FOR;
END FOR;
```

2. procapplication-a.vhdl

The statements in the file "procapplication-a.vhdl" in the PROC library that mention the modifiers "threads" and "generate" need to be commented out. These statements generate threads in a loop. These loops now need to be unraveled by defining the labels separately. For example, if the loop generates two threads, the label for the application_sw needs to be changed from "sw" to "sw1" and a similar mapping needs to be written with another label "sw2". The original code appears as follows.

```
 --setup processes for each of the individual tasks of the software model
threads:
FOR task IN user_tasks GENERATE
 cmp: IF the_tasks(task).port_id > 0 GENERATE
   sw: application_sw
    GENERIC MAP ( task_data => the_tasks(task),
                DEBUG => DEBUG )
    PORT MAP (send    => FromTask,
              receive => ToTask(task_data(task).port_id+NUM_DEFAULT_TASKS));
  END GENERATE;
END GENERATE;
```

After the modifications described above, the code appears as follows.

```
-- setup processes for each of the individual tasks of the software model
-- threads:
 FOR task IN user_tasks GENERATE
--  cmp: IF the_tasks(task).port_id > 0 GENERATE
    sw1: application_sw
```

```
    GENERIC MAP ( task_data => the_tasks(task),
                  DEBUG => DEBUG )
    PORT MAP (send    => FromTask,
              receive => ToTask(task_data(task).port_id+NUM_DEFAULT_TASKS));
-- END GENERATE;
-- END GENERATE;

sw2: application_sw
    GENERIC MAP ( task_data => the_tasks(task),
                  DEBUG => DEBUG )
    PORT MAP (send    => FromTask,
              receive => ToTask(task_data(task).port_id+NUM_DEFAULT_TASKS));
```

It should be noted, that for other applications that use more than two threads, the user will need to define similar labels for each individual thread.

3.1.2.2.2 Modifications to Memory Modules

None of the PML memory modules contain information about the capacity of the memory. A specification for capacity needs to be added to the generic declaration section of each memory module. An example follows.

```
ENTITY memoryH IS
    GENERIC ( INST: STRING := "_IO";
                    .
                    .
                    .
              CAPACITY : DATA_SIZE := 1024 bit_size;
                    .
```

```
.
.
                                    )
    PORT ( IO: INOUT  token );
END memoryH;
```

3.1.2.3 Analyzing the VHDL source code

In this step, the VTIP VHDL analyzer is used to parse the VHDL code in the PML libraries. The VHDL files need to be analyzed in a carefully controlled order. Every externally declared entity name in the file currently being analyzed must have already been parsed by the VTIP analyzer. The VTIP command to analyze a VHDL file is "VHDL". Hence, the command to analyze a file called proccore-c.vhdl will look like

```
VHDL proccore-c.vhdl
```

A shell script is written for each subdirectory that contains a sequence of "VHDL" commands in the proper order. This shell script resides in the subdirectory along with the VHDL files to be analyzed, and needs to be invoked from there. The following are the first few lines in the script file that analyzes the files in the packages subdirectory.

```
#!/bin/sh

# Properly sequenced compilation shell script for VTIP. --* Bp 3/15
# *IMPORTANT* These packages get analyzed first ! before anything else.

VHDL   base_types-p.vhdl
VHDL   base_types-b.vhdl
```

VTIP creates files in the dls_design_library for each file analyzed with the "VHDL" command. The new files have names similar to the entity analyzed along with the architecture, but with the extension ".vhdlview". For example, the script file above will create two files in the dls_design_library with the following names:

base_types.vhdlview
base_types-body.vhdlview

To maintain the required analysis order, the subdirectories need to be analyzed in the order: packages, leafcells, processor. Following all of these analyses, the dls_design_library contains the required set of files.

3.1.3 Determination of Restrictions on source code and parameters to be extracted

3.1.3.1 Determining Restrictions on Source code

Some restrictions need to be applied on the source code in order to facilitate the development of the interface tools. These restrictions may need to be defined in advance to make the interface possible, or they may be arrived at after the development as a result of the inherent properties of the tools being developed.

3.1.3.2 Determining the parameters to be extracted

In order for the extraction tools to be developed, the parameters that need to be extracted must be specified. Depending on what the final format of the tool output needs to be, some component characteristics are obtained directly from the DLS Library after analysis by the VHDL VTIP analyzer. But some characteristics are obtained only after some post processing is done on one or more parameters extracted from the library. These

parameters need to be identified, and the post processing specified before the tools required to extract them can be designed.

3.1.4 Finding the path to the required parameters using the VTIP DLS browser

When a VHDL file is analyzed by the VHDL analyzer, the analyzed information is stored in the dls_design_library in a file with the same name as the VHDL entity and architecture analyzed and with an extension ".vhdlview". The parameters are stored in a list of parameter values in a tree structure within this file. The path to this list of parameters is obtained using the VTIP DLS browser. The VHDLView file corresponding to the entity is opened using the DLS Browser. Then the path to the parameters can be found by tracing the SPI function calls while traversing down the tree structure using the right, left, up and down arrows. This process is described in detail earlier in Chapter 2.

3.1.5 Extraction of the parameters

The parameters required by the extraction tool can be extracted using the SPI. The entity and architecture names are received as command line parameters in the main program, and the library is designated as the dls_design_library.

```
main(argc,argv)
int argc;
char *argv[];
  {
LibName = "dls_design_library";
 EntityName = argv[1];
ArchName = argv[2];
```

The variable argc is an integer, and stores the number of command line parameters. The variable argv is an array of strings, and stores the value of each command line parameter

as successive strings in their order of occurrence. For example, if a command line reads as follows

ACET arch24t structural

then argc = 3, argv[0] = "ACET", argv[1] = "arch24t" and argv[2] = "structural". So, the line

EntityName = argv[1];

assigns "arch24t" to a variable called EntityName. Also,

ArchName = argv[2];

assigns the word "structural" to a variable called ArchName.

These names are then passed as parameters to the extraction subroutine.

NumberSignals(LibName, EntityName,ArchName);

In the extraction subroutine *NumberSignals*, the DLS is opened, along with the dls_design_library.

OpenDLS();

Liba = NewLibrarySymbol(LibName);

OpenLibrary(Liba);

The *OpenDLS* opens and initializes the Library System and the SPI. A variable "Liba" is then given a value by calling an SPI routine *NewLibrarySymbol*, with the LibName supplied as parameter. The function *NewLibrarySymbol* creates a Library symbol node that serves as a parameter to procedure *OpenLibrary*, which makes available the library units within it. Once the library is opened, the VHDLView node corresponding to the entity and architecture names is opened using the function *OpenUnit*.

Unit = OpenUnit(Lib, EntityName, ArchName, VHDLView, AugmentMode);

The parameters passed to this function are **Library** (The library containing the unit to open, e.g. Liba), **UnitName** (The primary name of the unit), **UnitQual** (The secondary name of the unit), **UnitView** (The view type of the unit, e.g. VHDLView) and **Mode** (The type of access allowed, e.g. AugmentMode). Here, the parameters Lib, EntityName and ArchName are passed to the subroutine from the main program.

Using the function call *qView*, the VHDLView node is obtained and from here *qRegion* is used to access the library-level RegionNode.

region = qRegion(qView(Unit));

Now the parameters can be reached by traversing down the tree structure according to the path found by using the DLS browser. Once these parameters have been reached, they are copied into variables. See next section for details relating to individual tools.

When all the parameters required have been copied to variables, the VHDLView node, the dls_design_library and the DLS are closed.

CloseUnit(Unit,Save);

CloseLibrary(Lib);

CloseDLS();

3.1.6 Formatting the output

The output needs to be formatted according to the requirements of the APT. This includes deriving information from already extracted parameters, converting the parameters into the units required by the output (Mbytes to Bytes etc.), and printing the information to the standard output or to a file in the format desired. See the following section for details relating to specific tools.

3.1.7 Testing of the output of the tools for accuracy

Once the tools have been developed, the outputs are tested for accuracy. There are two kinds of tests involved.

1. TEST 1
2. TEST 2

1. **TEST 1**

The parameters that are extracted from the model or the library are changed, and then the model is re-analyzed. The tool is run to produce the new output, and the output is checked to verify that the parameter is indeed changed in the output. If this is not the case, it might be due to two reasons.

- The path to the parameter is wrong. In this case, the correct path is found using the DLS browser, and the parameters extracted again.
- The wrong parameter is being extracted. In this case, the correct parameter to be extracted is decided upon, and the extraction completed.

2. **TEST 2**

The output files from the extraction tools are used for spreadsheet generation and checked for satisfactory performance. Here too, failure could be due to two reasons.

- Formatting errors. Since the output files are script files, every character needs to be formatted exactly right. Sometimes, the output file has unwanted characters like carriage returns, commas, special characters etc. These can cause a failure in the spreadsheet generation. If this is the case, the output needs to be reformatted to suit requirements.
- The wrong parameters are being extracted. In this case, the correct parameters need to be carefully decided upon, and then extracted again.

In all of these cases, the iteration of testing and modifying the code to suit the specifications is done until satisfactory results are produced.

The results of the PCET, ACET and CONET have been tested using the following two structural models. These models were developed as a part of the ongoing RASSP research project at Virginia Tech.

Four Processor Structural Model

The schematic of this structural model is shown in Figure 3.2.

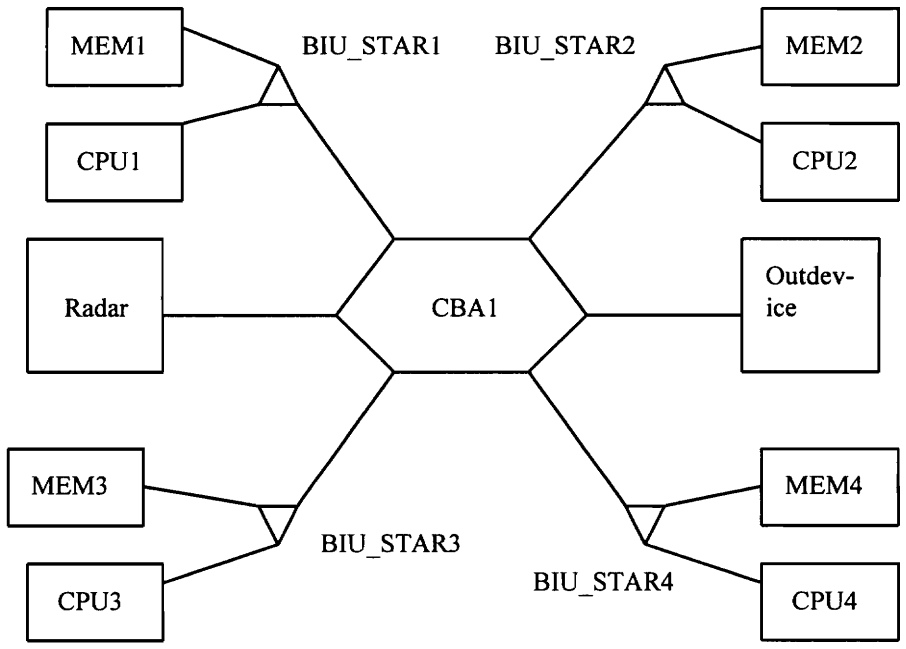


Figure 3.2 The Four Processor Architecture

This architecture consists of four biu_star clusters connected to a single crossbar. Each biu_star in turn has a processor and a local memory attached to it. This architecture

has an outdevice component which models a slow mass storage media, typically a disk drive. The indevice models the input radar.

Seventeen Processor Architecture Example

The schematic of this structural model is shown in Figure 3.3.

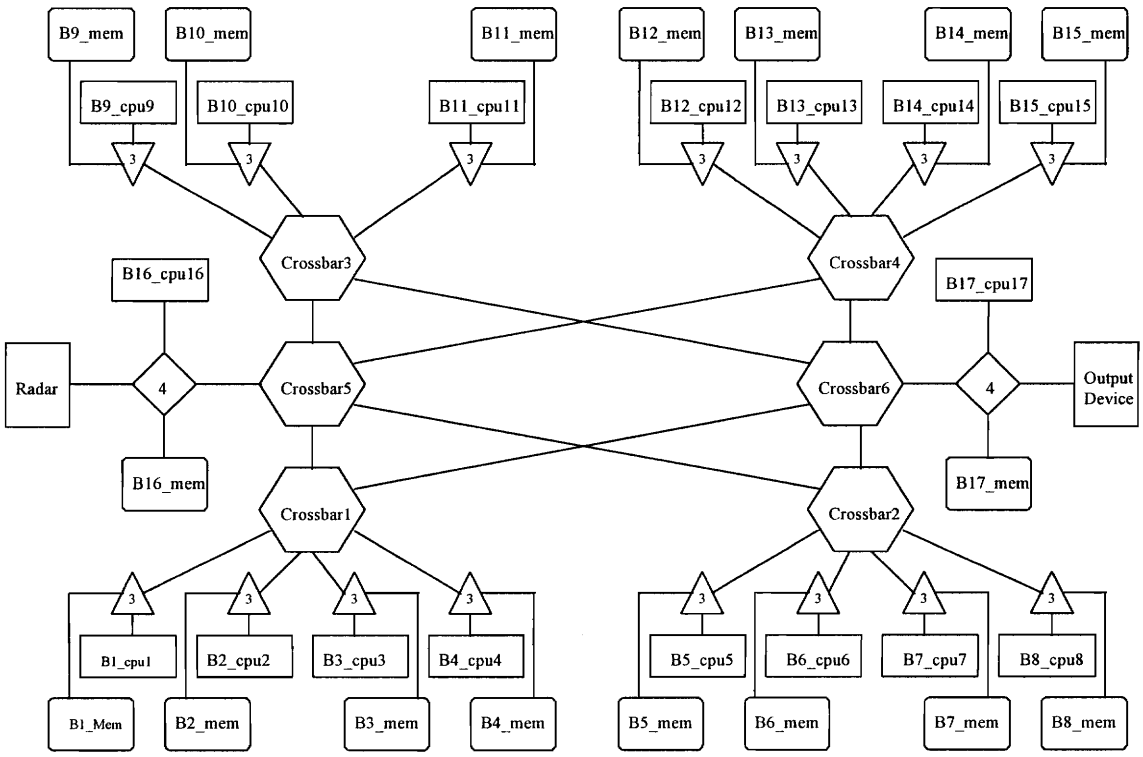


Figure 3.3 Seventeen Processor Raceway Architecture

There are seventeen processors in this architecture. In the figure, these processors are represented as B*i*_cpu*i*(*i* = 1 to 17). Each processor has a local memory module connected to it. The memory is shown as B*i*_mem*i*(*i* = 1 to 17). There are six crossbar modules which connect the seventeen processors. They are shown by hexagonal blocks in

modules which connect the seventeen processors. They are shown by hexagonal blocks in the figure. There are fifteen biu_star clusters in this architecture. The biu_star component is used to connect a processor, its local memory and one of the ports of the crossbar to a common bus. These biu_star components are represented by triangular blocks in the figure. There are two biu_four components in the seventeen processor architecture. In the figure, these components are represented as diamonds. The first biu_four connects the radar, B16_cpu16, B16_mem and crossbar5 to a common bus. The second biu_four connects the output device, B17_cpu17, B17_mem and crossbar6 to a common bus.

3.2 Specific Applications of the General Methodology

Three specific applications were developed as proof of concept of the methodology proposed in the previous section. These interface tools are the Processor Characteristic Extraction Tool (PCET), Architecture Characteristic Extraction Tool (ACET) and Connectivity Extraction Tool (CONET). These tools were tested using the 2 structural models described in section 3.1.7. The development process is described in the following section and the test results are included.

3.2.1 Processor Characteristic Extraction Tool (PCET)

In the PML library, each processor is characterized by its own instruction set and clockspeed. Each instruction in the instruction set has an instruction count associated with it, which refers to the number of clock cycles needed to perform one such instruction. These processors are characterized in the file proccore-c.vhdl, which is the configuration file for the different processor cores. This information is required by the sizing spreadsheet.

3.2.1.1 Study of Source and Target

The source of the PCET is the VHDL processor characterization packages in the PML library. These packages are carefully studied to gain a complete understanding of the various parameters of the components. The PCET is used to generate a processor characteristic file for each processor of interest. This target processor characteristic file has a filename equal to the processor entity name, with a c20 extension (e.g. pentium.c20). This file contains all of the parameters needed by the sizing spreadsheet. The output format for a PCET file follows.

! *processor_name*.c20

#LET([*,clock] = *Clock Rate*)
#LET([*,*instruction_name1*] = *Cycles1*)
#LET([*,*instruction_name2*] = *Cycles2*)
.
.
>[*,1]#/ "*processor_name*#/

where the words in italics represent the parameters extracted by the PCET. Each line in the above file is a spreadsheet macro, and hence needs to have a special format.

3.2.1.2 Analysis of the VHDL source code

The VHDL source code, which is the PML library is modified to suit the VTIP syntax. Then it is analyzed using the VTIP VHDL analyzer as described in section 3.1.2.

3.2.1.3 Determination of Restrictions and Parameters

The restrictions or naming conventions that need to be applied to the source code, and the parameters that need to be extracted by the PCET are determined. Since the source code in this case is the PML library, there are no restrictions to be applied to it. The parameters that are required to be extracted by the PCET follow.

1. Name of the processor entity (*processor_name*). This is extracted from the following line of code in the file proccore-c.vhdl
"CONFIGURATION proccore_960mx of PROCCORE IS"

2. The *Clock Rate* in MHz. This is extracted from the following line of code in the file proccore-c.vhdl, for the processor entity proccore_960mx.
CYCLE_COST => 40 Ns, -- (25 Mhz)
Note that the clock period is provided. Therefore, the extraction tool must compute the clock rate and adjust for units.

3. Each Instruction (*intsruction_namei*) along with its cycle cost (*cyclesi*).
This information can be extracted from the following section of the code.

```
SOFT_INSTRUCTION_DATA => (
    FLADD => (
      memory     => NO_MEM_ACCESS, -- Register to register only
      interrupt  => DISABLE_INT,   -- 960MX FPU not pipelined
      instr_cost => 3), -- in clocks

    FLDIV => (
      memory     => NO_MEM_ACCESS,
      interrupt  => DISABLE_INT,
      instr_cost => 14), -- in clocks
```

.

.

.

A list of the parameters to be extracted, along with the processing required, and their output formats is shown in Figure 3.4.

Parameter name	Processing	Output Format	Example
Processor entity-name	1. Truncate first 9 characters 2. Add '.c20' at end	! processor-name.c20	! i960mx.c20
CYCLE_COST	Clock Rate = 1000 / CYCLE_COST	#LET([*,clock] = Clock Rate)	#LET([*,clock] = 25)
SOFT_INSTRUCTION_DATA (name and cycles)	None	#LET([*,instruction name] = cycles)	#LET([*,fladd] = 3) #LET([*,fldiv] = 14)

Figure 3.4 List of Specifications for PCET parameters and their processing

3.2.1.4 Finding paths to the parameters

The paths to the parameters can be found as described in section 3.1.4 by using the DLS Browser.

3.2.1.5 Extraction of parameters using SPI

The parameters required by the PCET are extracted using SPI function calls corresponding to the paths found by using the DLS Browser.

1. The following piece of code extracts the *processor_name*, and copies it into the string variable **Name**.

```
Decl = Value(LastItem(qDecls(region))); /* Traverse tree to reach processor_name */
I = strlen((char *)qText(qDeclName(Decl)));
```

```
Name = malloc(I);
strcpy(Name,(char *)qText(qDeclName(Decl))); /* Copy processor_name into variable*/
```

2. The following piece of code extracts the *Cycle_cost*, and copies it into the string variable **Clock**.

```
region1 = qRegion(Decl); /* Traverse tree to reach clock_cycle */
Temp = qStmts(qRegion(Value(FirstItem(qStmts(region1)))));
Component = Value(NthItem(3,Temp));
Decl1 = qData(Value(LastItem(qData(qBinding(Component)))));
Gen = Value(NthItem(2,Decl1));
Clock = qIntVal4(qQuantity(Value(LastItem(qData(Gen))))); /* Copy clock_cycle to
variable Clock */
```

3. The following piece of code extracts the *instruction_name* and *cycles* and copies them into the variables **Operation** and **InstCost**.

```
Generic = Value(NthItem(11,Decl1)); /*Traverse tree to reach Cycles */
MainList = qData(Value(LastItem(qData(Generic))));
SomeItem = FirstItem(MainList);
while(!NullItem(SomeItem))
{ Decl2 = Value(SomeItem);
  if(IsA(Kind(Value(FirstItem(qData(Decl2)))),AscendingRange))
{Temp1 = qData(Value(LastItem(qData(Decl2))));
Tempr = Value(NthItem(3,Temp1));
InstCost = qIntVal4(Value(LastItem(qData(Tempr)))); /* Copy Cycles into variable*/
ThisItem = FirstItem(qConstraint(Value(FirstItem(qData(Decl2))))));/*Traverse tree to
reach Instruction_name*/
while(!NullItem(ThisItem))
{J = strlen((char *)qText(Value(ThisItem)));
Operation = malloc(J);
strcpy(Operation,(char *)qText(Value(ThisItem)));/*Copy Instruction_name to variable*/
```

3.2.1.6 Formatting of output

The parameters extracted using the SPI code described above are formatted to suit output specifications.

1. The processor entity-name that is extracted is proccore_960mx. The first 9 characters of this parameter need to be stripped off, and a "c20" appended to the result thus giving the name 960mx.c20 which is the name of the characteristic file for the processor entity proccore_960mx. The following section of code performs the required formatting.
The required name is stored in the variable **Name1**.

```
Name1 = malloc(I-9);
for (z=9; z <= I; z=z+1)
{
Name1[z-9] = Name[z];   /* First 9 characters are stripped off */
}
strcat(Name1,".c20");  /* "c.20" is appended to Name1 */
```

2. The extracted parameter CYCLE_COST, gives us the clock period in Ns. This parameter needs to be post-processed by converting it to the clock frequency in MHz. This is achieved by the following conversion:
clock frequency (MHz) = 1000 / CYCLE_COST /* in NS */
The following piece of code does the conversion desired. The Clock Rate is stored in the variable **CS**.

```
CS = 1000/Clock;
```

3. The names of the instructions such as FLADD, FLDIV etc. along with their instruction costs given by the parameter instr_cost can be extracted directly from the dls_design_library. No postprocessing is required.

The parameters thus extracted are printed into files using the following lines of code.

```
fprintf(ptr1,"! %s \n\n",Name1);
fprintf(ptr1,"#LET([*,clock] =  %3.0f)\n",CS);
fprintf(ptr1,"#LET([*,%s] =  %d)\n",Operation,InstCost);
```

The complete code for PCET is in Appendix E.

3.2.1.7 Test Results

The outputs of the PCET have been subjected to the two tests described in section 3.1.7 and verified for accuracy. An example of a processor characteristic file generated by the PCET for the i960mx processor follows.

```
! i960mx.c20

#LET([*,clock] =  25)
#LET([*,fladd] =  3)
#LET([*,fldiv] =  14)
#LET([*,flmlt] =  3)
#LET([*,multadd] =  3)
#LET([*,cmult] =  12)
#LET([*,btfy] =  18)
#LET([*,flsin] =  150)
#LET([*,flop] =  3)
#LET([*,iadd] =  1)
#LET([*,idiv] =  12)
#LET([*,imlt] =  4)
#LET([*,isqr] =  4)
#LET([*,iop] =  1)
#LET([*,read] = 1)
#LET([*,write] = 1)
>[*,1]#/ "i960mx#/
```

where clock = clock frequency in MHz;

fladd = floating point addition;

fldiv = floating point division;

flmlt = floating point multiplication;

multadd = floating point multiply and add;

cmult = complex multiply;

btfy = butterfly (rad 2 complex multiply);

flsin = floating point sine;

flop = floating point operation;

iadd = integer addition;

idiv = integer division;

imlt = integer multiplication;

isqr = integer squaring;

iop = integer operation;

read = read;

write = write;

In the above characteristic file, clock represents the instruction cycle in MHz. Each of the instruction numbers is the number of cycles necessary to execute one of the instructions. If the cycle time is not explicitly specified, the time for multadd is generated as equal to flmlt; the time for cmult equal to 4 * flmlt; and the time for btfy equal to 6 * flmlt. Only basic instructions needed in DSP applications are included.

3.2.2 Architecture Characteristic Extraction Tool (ACET)

The ACET interfaces the VHDL structural model with the APT. It extracts the architecture characteristics of the structural model, such as names of the components and their types, and produces the output in the form of a spreadsheet building script file. This script file is used to conduct mapping analyses of the architecture.

3.2.2.1 Study of Source and Target

The VHDL source code and target format of the ACET need to be studied in order to understand the source parameters and to determine the parameters to be extracted. The source code for the ACET is the VHDL structural model, which in turn derives its primitives from the PML library. The output format for the ACET is shown below.

#! /bin/csh -f

hw_build *Device-Type* (store) *Device-Name* *Read access time* *Write access time* *Unit* *Capacity* *Device-Type* (process) *Device-Name* *Processor-Type* ..... where the words in italics represent the parameters extracted by the ACET. The output of the ACET is a shell script that is used to call another shell script, hw_build. Thus, the first line is

#! /bin/csh -f

This is followed by a single (no carriage returns) line consisting of the call hw_build, followed by the parameters for the processors and memories comprising the architecture.

3.2.2.2 Analysis of VHDL Source Code

The PML library is modified to suit the VTIP requirements, and analyzed as described in section 3.1.2. The VHDL structural model created by the user needs to be analyzed using the VTIP VHDL analyzer. This is done by typing the VTIP command "VHDL" followed by the name of the VHDL file describing the structural model at the UNIX prompt. VTIP creates a file whose name is the same as the entity of the model analyzed, along with its architecture name and an extension ".vhdlview". This file is stored in the dls_design_library.

3.2.2.3 Determination of Restrictions and Parameters

The extraction tool needs to extract the instantiation names of the components in the structural model developed by the user. These instantiation names need to follow some naming conventions in order for the extraction tools to work on them, and also to satisfy the spreadsheet requirements. These naming conventions need to be established before the structural model can be developed.

The restrictions or naming conventions that need to be applied to the source code, and the parameters that need to be extracted by the ACET are determined. Since the source code in this case is the VHDL structural model, the following restrictions need to be applied to it.

The spreadsheet is designed so that columns and rows can be addressed by name (the character string in the first row of the column or the first column of the row, called a label). The component names are used for addressing in load calculations. Therefore all names need to be unique, need to start with an alpha character or an underscore (i.e. must be a string), and cannot contain a period or a blank (periods denote "ranges" in 2020 addresses). In addition names must not be able to be interpreted as cell addresses. For example, PE1 is not valid. Hence, an underscore is appended as the first character in all the instantiations by the ACET. In order to allow differentiation of fields in the output files, component names must be limited to a maximum of 10 characters including the underscore. If the names are longer than 10 characters, they get truncated by the ACET. Hence, the first 9 characters of a component name (excluding the underscore) have to be unique.

The parameters that are required to be extracted by the ACET follow.

1. Device-Name

For each component the first two parameters are Device Type (process or store) and a unique Device Name or the instantiation name for each component. The instantiation name "B1_Mem" is extracted from the following section of code.

B1_Mem: memoryh

2. Device-Type

This is the type of the component (processor, indeviceh, memoryh etc). It is the first parameter for each component in the ACET. The device type is the component name, memoryh, found in the following section of code.

B1_Mem: memoryh

3. Processor-Type

This is extracted from the following section of code. The words in lower case italics are the key words.

FOR CPU01:processor USE CONFIGURATION PROC.DINKY_PROCESSOR1;
FOR CONFIGURATION DINKY_PROCESSOR1 USE proccore_29050

4. THROUGHPUT_INFO

This is extracted from the following section of code:

THROUGHPUT_INFO => "CONSTANT 40",

5. Unit or wordsize

Wordsize is directly available as a generic, and can be extracted from the dls_design_library. The wordsize is extracted from the following section of the code.

UNIT => 64 Kbyte ;

6. Capacity

The *Capacity* is found in the following section of the code.

CAPACITY => 428000000 bytes;

A list of the parameters required to be extracted, along with the processing required, and their output formats is shown in Figure 3.5.

Parameter name	Processing	Output Format	Example
Device-Type	1. If processor, indeviceh, outdevice : process 2. If memoryh : store		
Device-Name	1. Add underscore '_' at the beginning 2. Truncate to 10 characters	Device-Type Device-Name Processor-Type	process _cpu01 i960mx
Processor-Type	1. If "386" or "960mx" : add 'I' 2. If "sharc" :"21062"		
THROUGHPUT_INFO (memoryh)	1. Convert to numeric 2. Read /Write Access Time = 1000000 / THROUGHPUT_INFO	Device-Type Device-Name Read-Access-Time Write-Access-Time UNIT CAPACITY	store _mem1 10000 10000 64 46080
UNIT(memoryh)	None		
CAPACITY (memoryh)	1. If in Mbytes : UNIT * 1048576 2. If in Kbytes : UNIT * 1024 3. If in bit_size : UNIT / 8		

Figure 3.5 List of Specifications for ACET parameters and their processing

3.2.2.4 Finding paths to the parameters

The paths to the parameters can be found as described in section 3.1.4 by using the DLS Browser.

3.2.2.5 Extraction of parameters using SPI

The parameters required by the ACET are extracted using SPI function calls corresponding to the paths found by using the DLS Browser.

1. The following section of code is used to extract the parameter Device-Name and store it in a string variable **Name4**.

```
Decl = Value(LastItem(qDecls(region))); /* Traverse tree to reach Device-Name */
MainList = qStmts(qRegion(Decl));
SomeItem = FirstItem(MainList);
ctr = 1;
while(!NullItem(SomeItem))
{ Decl2 = Value(SomeItem);
I = strlen((char *)qText(qStmtName(Decl2)));
Name4 = (char *)malloc(I);
strcpy(Name4,(char *)qText(qStmtName(Decl2))); /* Copy Device-name into variable*/
```

2. The following section of code is used to extract the parameter Device-Type and store it in a string variable **Name1**.

```
J = strlen((char *)qText(qName(Decl2)));
Name1 = (char *)malloc(J);
strcpy(Name1,(char *)qText(qName(Decl2))); /* Copy Device-Type to variable */
```

3. The parameter Processor-Type is extracted using the following section of code and stored in the variable **Name3**.

```
node1 = qDef(Value(FirstItem(qData(qBinding(Decl2))))); /* Traverse tree to reach
Processor-Type */
node2 = qRegion(Value(FirstItem(qStmts(qRegion(node1))))));
node3 = qData(qBinding(Value(FirstItem(qStmts(node2))))));

K = strlen((char *)qText(Value(FirstItem(node3))));
Name2 = (char *)malloc(K);
Name3 = (char *)malloc(K-9);
strcpy(Name2,(char *)qText(Value(FirstItem(node3))));
for (z=9; z <= K; z=z+1)
{
Name3[z-9] = Name2[z]; /* Copy Processor-Type into variable */
}
```

4. The section of code needed to extract throughput_info is shown below. The generic is stored in a string variable **Names**.

```
MainList1 = qAssocs(Decl2); /* Traverse tree to reach throughput_info */
SomeItem1 = FirstItem(MainList1);

{if(IsA((Kind(Value(SomeItem1))),GenericAssocOp))
{Decl3 = Value(SomeItem1);
I = strlen((char *)qText(Value(FirstItem(qData(Decl3)))));
Names4 = (char *)malloc(I);
strcpy(Names4,(char *)qText(Value(FirstItem(qData(Decl3)))));
if((strcmp(Names4,"throughput_info")) == 0)
        {Gen1 = Decl3;}
I = strlen((char *)qStrVal(Value(LastItem(qData(Gen1)))));
Names = (char *)malloc(I);
strcpy(Names,(char *)qStrVal(Value(LastItem(qData(Gen1))))); /* Copy throughput_info
onto variable */
```

5. The C code required to extract the parameter Unit is shown below. The *Unit* is stored in the variable **wordsize**.

```
if((strcmp(Names4,"unit")) == 0)
        {Gen3 = Decl3;}
wordsize = qIntVal4(qQuantity(Value(LastItem(qData(Gen3))))); /* Copy Unit into
variable */
```

6. The C code needed to extract the parameter Capacity is shown below. The *Capacity* is stored in the variable **capacity**.

```
if((strcmp(Names4,"capacity")) == 0)
        {Gen2 = Decl3;}
capacity1 = qIntVal4(qQuantity(Value(LastItem(qData(Gen2))))); /* Copy Capacity into
variable */
```

3.2.2.6 Formatting of output

The parameters extracted using the SPI code described above, are formatted to suit output specifications.

1. The extracted parameter Device-Name needs to be post-processed to meet output requirements. It has an underscore appended as the first character to it in every case, and the length of the name is limited to 10 characters, with the name getting truncated in case it is longer than 10 characters. If the first 10 characters of each instantiation name is not unique, then it will result in erroneous results, as that name will occur twice in the ACET output. The section of code to do this is shown below. The final result is stored in the string variable **Name**.

```
Name = (char *)malloc(10);
strcpy(Name,"_");
strncpy(tempstr,Name4,9);
strcat(Name,tempstr);
Name[10] = '\0';
```

2. The Device-Type can be extracted directly from the structural model. No post processing is required.

3. The parameter Processor-Type needs to be post-processed. If the type is "386" or "960mx", the name is changed to "i386" and "i960mx". If the type is "sharc" the name is changed to "21062". The following section of code performs the processing.

```
if((strcmp(Name3,"960mx")) == 0 || (strcmp(Name3,"386")) == 0)
{
Name5 = (char *)malloc(K-8);
strcpy(Name5,"i  ");
for (z=0; z <= K-7; z=z+1)
{
Name5[z+1] = Name3[z];
}
Name3 = (char *)malloc(K-8);
strcpy(Name3,Name5);
```

```
}
if((strcmp(Name3,"sharc")) == 0)
{
strcpy(Name3,"21062");
}
```

4. The parameter throughput_info needs to be post processed. It is a string variable, and it needs to be converted into a numeric. This is done by first stripping off the first 9 characters of 'constant 100000'. Then the remaining string variable is converted to a numeric. The following section of code does this. The numeric value of THROUGHPUT_INFO is stored in a numeric variable **throughput**.

```
Names1 = (char *)malloc(I-9);
for (z=9; z <= I; z=z+1)
{Names1[z-9] = Names[z];}  /*Stripping off the first 9 characters */
Names1[I-9] = '\0';
throughput = 0;
for (z=0; z < I-9; z=z+1)
{
switch(Names1[z])
   {
        case '0'  :  dummy = 0;
                        break;
        case '1'  :  dummy = 1;
                        break;
        case '2'  :  dummy = 2;
                        break;
        case '3'  :  dummy = 3;
                        break;
        case '4'  :  dummy = 4;
                        break;
        case '5'  :  dummy = 5;
                        break;
        case '6'  :  dummy = 6;
                        break;
        case '7'  :  dummy = 7;
                        break;
        case '8'  :  dummy = 8;
                        break;
        case '9'  :  dummy = 9;
```

```
}
throughput = (throughput*10) + dummy;   /* Converting string to numeric variable */
}
```

The Read/Write access times are not available directly, but are calculated from the throughput. The throughput gives the number of words read or written per second. So, in order to obtain the Read and Write Access Time in ms, the following conversion is done- Read/Write Access Time = 1000000/throughput. The following section of code does this. The final result is stored in the variable **CS1**.

```
CS1 = 1000000./throughput;
```

5. The extracted parameter wordsize or unit does not require any post processing.

6. The extracted parameter, capacity may need to have its unit converted from Mbytes, Kbytes or Bit-size to byte. This is done by multiplying the capacity by a factor corresponding to its unit. The following section of code does this conversion.

```
I = strlen((char *)qText(qQuantum(Value(LastItem(qData(Gen2))))));
name = (char *)malloc(I);
strcpy(name,(char *)qText(qQuantum(Value(LastItem(qData(Gen2))))));
if((strcmp(name,"kbyte")) == 0)
        {
        units = 1024;
        }
else if((strcmp(name,"mbyte")) == 0)
        {
        units = 1048576;
        }
else if((strcmp(name,"byte")) == 0)
        {
        units = 1;
        }
else if((strcmp(name,"bit_size")) == 0)
        {
units = 1./8;
        }
capacity = capacity1*units;
```

The parameters are printed out using the following section of code.

```
printf("#! /bin/csh -f\n");
printf(" hw_build ");
printf(" %s %s %s ",N1,Name,Name3);
printf(" %f %f %d %f ",CS1,CS1,wordsize,capacity);
```

The complete code for ACET is in Appendix E.

3.2.2.7 Test Results

The outputs of the ACET have been subjected to the two tests described in section 3.1.7 and verified for accuracy. The script file generated by the ACET for the architecture shown in Figure 3.2 follows.

```
#! /bin/csh -f
 hw_build store _mem1 10000.000000 10000.000000 64 46080.000000
process _cpu01 i960mx store _mem2 10000.000000 10000.000000 64
46080.000000 process _cpu02 i960mx store _mem3 10000.000000
10000.000000 64 46080.000000 process _cpu03 i960mx store _mem4
10000.000000 10000.000000 64 46080.000000 process _cpu04 i960mx
process _radar indevice process _dumb_devi outdevice
```

The script file generated by the ACET for the architecture shown in Figure 3.3 follows.

```
#! /bin/csh -f
 hw_build store _b1_mem 0.025000 0.025000 4 46080.000000 process
_b1_cpu1 21062 store _b2_mem 0.025000 0.025000 4 46080.000000 process
_b2_cpu2 21062 store _b3_mem 0.025000 0.025000 4 46080.000000 process
_b3_cpu3 21062 store _b4_mem 0.025000 0.025000 4 46080.000000 process
_b4_cpu4 21062 store _b5_mem 0.025000 0.025000 4 46080.000000 process
_b5_cpu5 21062 store _b6_mem 0.025000 0.025000 4 46080.000000 process
_b6_cpu6 21062 store _b7_mem 0.025000 0.025000 4 46080.000000 process
_b7_cpu7 21062 store _b8_mem 0.025000 0.025000 4 46080.000000 process
_b8_cpu8 21062 process _b16_radar indevice store _b16_mem 0.025000
0.025000 4 46080.000000 process _b16_cpu16 21062 store _b9_mem
0.025000 0.025000 4 46080.000000 process _b9_cpu9 21062 store
```

_b10_mem 0.025000 0.025000 4 46080.000000 process _b10_cpu10 21062 store _b11_mem 0.025000 0.025000 4 46080.000000 process _b11_cpu11 21062 store _b12_mem 0.025000 0.025000 4 46080.000000 process _b12_cpu12 21062 store _b13_mem 0.025000 0.025000 4 46080.000000 process _b13_cpu13 21062 store _b14_mem 0.025000 0.025000 4 46080.000000 process _b14_cpu14 21062 store _b15_mem 0.025000 0.025000 4 46080.000000 process _b15_cpu15 21062 process _b17_outde outdevice store _b17_mem 0.025000 0.025000 4 46080.000000 process _b17_cpu17 21062

3.2.3 Connectivity Extraction Tool (CONET)

The CONET tool extracts connectivity information and parametric values for transfer devices. The name of each of the transfer devices is extracted, along with its parametric values and a list of devices that are connected to its ports. This information is required to complete the mapping of partitioned software onto the candidate architecture.

3.2.3.1 Study of Source and Target

The VHDL source code, and target format of the CONET need to be studied in order to understand the working of the parameters, and to decide the parameters required to be extracted. The source code for CONET is the VHDL structural model, which in turn derives its primitives from the PML library. The output format for CONET is shown below.

Transfer Device Name

Parameter1 Value1
Parameter2 Value2
.
.

Signal name1 Connected component1

Signal name2 *Connected component2*

.

.

3.2.3.2 Analysis of VHDL Source Code

The PML library is modified to suit the VTIP requirements, and analyzed as described in section 3.1.2. The VHDL structural model created by the user needs to be analyzed using the VTIP VHDL analyzer. This is done by typing the VTIP command "VHDL" followed by the name of the VHDL file describing the structural model at the UNIX prompt. VTIP creates a file whose name is the same as the entity of the model analyzed, along with its architecture name and an extension ".vhdlview". This file is stored in the dls_design_library.

3.2.3.3 Determination of Restrictions and Parameters

The extraction tool needs to extract the instantiation names of the components in the structural model developed by the user. These instantiation names need to follow some naming conventions in order for the extraction tools to work on them, and also to satisfy the spreadsheet requirements. These naming conventions need to be established before the structural model can be developed.

The restrictions or naming conventions that need to be applied to the source code and the parameters that need to be extracted by CONET are determined. Since the source code in this case is the VHDL structural model, the following restrictions need to be applied to it.

The instantiation names of all the transfer devices, for which connectivity information is required, need to have the first three characters to be either "cba" or "biu".

The parameters that are required to be extracted by CONET follow.

1. Transfer Device Name

The name of the transfer device starting with "cba" or "biu". This can be found in the code as follows.

BIU1: biu_star

2. Parameters and their values

These are the list of generics associated with the transfer device and their values. In case the transfer device is a crossbar, this information is found in the following piece of code

```
GENERIC MAP(
        INST            => "_TRFCB2",
        NUM_PORTS       => 6,
        PROC_TIME_INFO          => "CONSTANT 1",
        TIMEOUT_INFO            => "CONSTANT 100",
        PRIORITY_INFO           => "CONSTANT 0", --* not used at present;
        QUEUE_DEPTH             => 10,
        PROTOCOL                => handshake,
        TX_LATENCY_INFO         => "CONSTANT 5",
        THROUGHPUT_INFO         => "CONSTANT 200",
        TRACE                   => TRUE,
        RX_LATENCY_INFO         => "CONSTANT 5",
        UNIT                    =>  64 Kbyte
       )
```

In case the transfer device is a biu interface, this information is found in the following piece of code.

```
GENERIC MAP( ACK_TIME_INFO     => "CONSTANT 100",
           BUS_TIMEOUT_INFO  => "CONSTANT 100",
           TOP_GLOBALID      => "B2_Mem",
           CENTER_GLOBALID   => "?*",
           BOTTOM_GLOBALID   => "B2_CPU2",
           SCREENID          => "B2_?*",
           GLOBAL_PROTOCOL   => PIBUS32,
           INST              => "B2_STAR",
           TOP_LOCALID       => "?*",
```

```
CENTER_LOCALID   => "?*",
BOTTOM_LOCALID   => "?*",
QUEUE_DEPTH      => 9,
RX_LATENCY_INFO  => "CONSTANT 100",
RX_PRIORITY_INFO => "CONSTANT 0",
THROUGHPUT_INFO  => "CONSTANT 200",
TOKEN_PROTOCOL   => HANDSHAKE,
TRACE            => TRUE,
TX_LATENCY_INFO  => "CONSTANT 100",
TX_PRIORITY_INFO => "CONSTANT 0",
UNIT             => 64 Kbyte )
```

3. Signal and connected device

These are the list of components connected to the transfer device, along with the name of the signal connecting them to the transfer device under consideration. This information is obtained by tracing each signal mapped to the port of the transfer device, to the port of another device. This signal connects the two devices. This information is found in the following piece of code.

```
PORT MAP (
      ExtPorts(1) =>  IntPort1,
      ExtPorts(2) =>  IntPort2,
      ExtPorts(3) =>  IntPort3,
      ExtPorts(4) =>  IntPort4,
      ExtPorts(5) =>  IntPort5,
      ExtPorts(6) =>  IntPort6);

PORT MAP (UP_local    => s11,
        DOWN_local  => s21,
        CENTER_local => IntPort1);
```

A list of the parameters required to be extracted, along with the processing required, and their output formats is shown in Figure 3.6.

Parameter name	Processing	Output Format	Example
Device Name	None	-	biu_star1
Generic Map	None	Generic Value	queue_depth 1 unit 64 kbyte
Port Map and connected device	Find connected device by tracing signal from Port Map	Signal Device name	s11 mem1 s31 cb1

Figure 3.6 List of Specifications for CONET parameters and their processing

3.2.3.4 Finding paths to the parameters

The paths to the parameters can be found as described in section 3.1.4 by using the DLS Browser.

3.2.3.5 Extraction of parameters using SPI

The parameters required by CONET are extracted using SPI function calls corresponding to the paths found by using the DLS Browser.

1. The following section of code is used to extract the parameter Device Name. The Device Name is copied into a variable called **Name**.

```
MainList = qStmts(qRegion(Decl)); /*Traverse tree to reach Device Name */
 SomeItem = FirstItem(MainList);
 SomeItem1 = FirstItem(MainList);
 ctr = 1;
 while(!NullItem(SomeItem))
  { Decl2 = Value(SomeItem);
    I = strlen((char *)qText(qStmtName(Decl2)));
```

```
Name = (char *)malloc(I);
    strcpy(Name,(char *)qText(qStmtName(Decl2)));  /* Copy Device Name into
variable */
```

2. The C code required to extract the transfer device parameters and their values is shown below. The parameter name is stored in the variable **Names4**, and its value in the variable **Names5**.

```
MapList = qAssocs(Decl2);  /* Traverse tree to reach parameter and value */
      MapItem = FirstItem(MapList);
    while(!NullItem(MapItem))
       {
        if(IsA(Kind(Value(MapItem)),GenericAssocOp))
           {
                Decl3 = Value(MapItem);
                I = strlen((char *)qText(Value(FirstItem(qData(Decl3)))));
                Names4 = (char *)malloc(I);
                strcpy(Names4,(char *)qText(Value(FirstItem(qData(Decl3))))); /*
Copy parameter name into variable */
                I = strlen((char *)qStrVal(Value(LastItem(qData(Decl3)))));
                Names5 = (char *)malloc(I);
                strcpy(Names5,(char  *)qStrVal(Value(LastItem(qData(Decl3)))));
/* Copy parameter value into variable */
```

3. The signal names and the names of the connected devices are extracted using the following section of C code. The signals are stored in the variable **Signal1**, and the connected device name is stored in the variable **Label**.

```
if(IsA(Kind(Value(MapItem)),PortAssocOp))
{
 List1 = qData(Value(MapItem));
 Item1 = LastItem(List1);
    I = strlen((char *)qText(Value(Item1)));
    Signal = (char *)malloc(I);
    strcpy(Signal,(char *)qText(Value(Item1)));

    SomeItem1 = FirstItem(MainList);
    while(!NullItem(SomeItem1))
```

```
{
 I = strlen((char *)qText(qStmtName(Value(SomeItem1))));
 Label = (char *)malloc(I);
 strcpy(Label,(char *)qText(qStmtName(Value(SomeItem1))));
 if(strcmp(Name,Label) != 0 )
  {
   Item2 = FirstItem(qAssocs(Value(SomeItem1)));

   while(!NullItem(Item2))
       {
        if(IsA((ObjectType)PortAssocOp,Kind(Value(Item2))))
         {
          Item3 = LastItem(qData(Value(Item2)));

          if(IsA((ObjectType)ListOp,Kind(Value(Item3))))
              {
               Itemlast = FirstItem(qData(Value(Item3)));
               while(!NullItem(Itemlast))
                {
                                                    I = strlen((char
*)qText(Value(LastItem(qData(Value(Itemlast))))));
                                  Signal1 = (char *)malloc(I);
                                                    strcpy(Signal1,(char
*)qText(Value(LastItem(qData(Value(Itemlast))))));
```

3.2.3.6 Formatting of output

The parameters extracted using the SPI code described above are readily available in the required format to suit output specifications. Hence no post processing is necessary. The parameters are printed out using the following section of code.

```
printf("\n %s \n",Name);
printf("\n    %s    %s ",Names4,Names5);
printf(" \n\n \t %s \t %s",Signal1,Label);
```

The complete code for the CONET is in Appendix E.

3.2.2.7 Test Results

The outputs of CONET have been subjected to the two tests described in section 3.1.7 and verified for accuracy. The output file generated by CONET for the architecture shown in Figure 3.2 follows. The values of the parameters ack_time_info, bus_timout_info, tx_latency_info and rx_latency_info are in micro seconds. The priority info refers to actual priority and not relative priority. The unit of throughput is words/second.

cba1

timeout_info CONSTANT 100
priority_info CONSTANT 0
queue_depth 1
protocol handshake
tx_latency_info CONSTANT 5
throughput_info CONSTANT 9000
rx_latency_info CONSTANT 5
unit 64 kbyte

s31 biu_star1
s32 biu_star2
s33 biu_star3
s34 biu_star4
s35 radar
s36 outdevice

biu_star1

ack_time_info CONSTANT 100
bus_timeout_info CONSTANT 100
queue_depth 1
rx_latency_info CONSTANT 100
rx_priority_info CONSTANT 0
throughput_info CONSTANT 1000
token_protocol handshake

```
    tx_latency_info    CONSTANT 100
    tx_priority_info    CONSTANT 0
    unit    64    kbyte


        s11      mem1
        s31      cba1
        s21      cpu1


biu_star2

    ack_time_info    CONSTANT 100
    bus_timeout_info    CONSTANT 100
    queue_depth    1
    rx_latency_info    CONSTANT 100
    rx_priority_info    CONSTANT 0
    throughput_info    CONSTANT 1000
    token_protocol    handshake
    tx_latency_info    CONSTANT 100
    tx_priority_info    CONSTANT 0
    unit    64    kbyte


        s12      mem2
        s32      cba1
        s22      cpu2


biu_star3

    ack_time_info    CONSTANT 100
    bus_timeout_info    CONSTANT 100
    queue_depth    1
    rx_latency_info    CONSTANT 100
    rx_priority_info    CONSTANT 0
    throughput_info    CONSTANT 1000
    token_protocol    handshake
    tx_latency_info    CONSTANT 100
    tx_priority_info    CONSTANT 0
    unit    64    kbyte
```

```
        s13     mem3
        s33     cba1
        s23     cpu3


biu_star4

    ack_time_info     CONSTANT 100
    bus_timeout_info     CONSTANT 100
    queue_depth     1
    rx_latency_info     CONSTANT 100
    rx_priority_info     CONSTANT 0
    throughput_info     CONSTANT 1000
    token_protocol     handshake
    tx_latency_info     CONSTANT 100
    tx_priority_info     CONSTANT 0
    unit     64     kbyte


        s14     mem4
        s34     cba1
        s24     cpu4
```

The output file generated by CONET for the architecture shown in Figure 3.3 is in Appendix C.

A design methodology has thus been developed to interface the VHDL performance models to the Algorithm Partitioning Tool. The interface tools PCET, ACET and CONET developed in this chapter, demonstrate proof of concept of the methodology proposed. These tools have been tested using two different sample architectures. These tools interface the VHDL library and structural model to the APT, thereby saving significant time and labor and speeding up the design process considerably.

Chapter 4. VHDL Interface Procedures

This chapter describes the VHDL procedures that are used in the development of the application software for the performance models. The spreadsheet analysis gives a candidate software partitioning, which defines the individual tasks that need to be performed by each component in the architecture. The procedures developed in this chapter significantly simplify the task descriptions in the application software code.

The simulation of an algorithm on a multiprocessor architecture involves the passing of data back and forth among the various processors and memories. All such communication between the components is effected through tokens. See Chapter 2 for details on syntax. There are three kinds of tokens used by the application software - *Readtoken*, *Writetoken* and *Controltoken*. A *Readtoken* is sent from a processor to a memory to simulate the reading of data from the memory. The *Readtoken* specifies the

amount of data to be read and the path from the processor to the memory. A *Writetoken* is sent from a processor to a memory to simulate the writing of data into the memory. The *Writetoken* specifies the amount of data to be written and the path from the processor to the memory. A *Controltoken* is sent from one processor to another processor to note the occurrence of an event. Tokens can be initiated only by processors. In essence, the application software consists of a series of read and write operations interspersed with processing delays associated with computations involving the data. Since these read and write operations take place very frequently, a procedure for these operations greatly reduces code complexity. This chapter describes the procedures written for the read and write operations, along with other common functions that help to facilitate the development of the application software.

4.1 The Processor Configuration[5]

The Processor is made up of three major sub-components: the Processor Application module, the Processor Scheduler and the Processor Core. Refer to Figure 4.1.

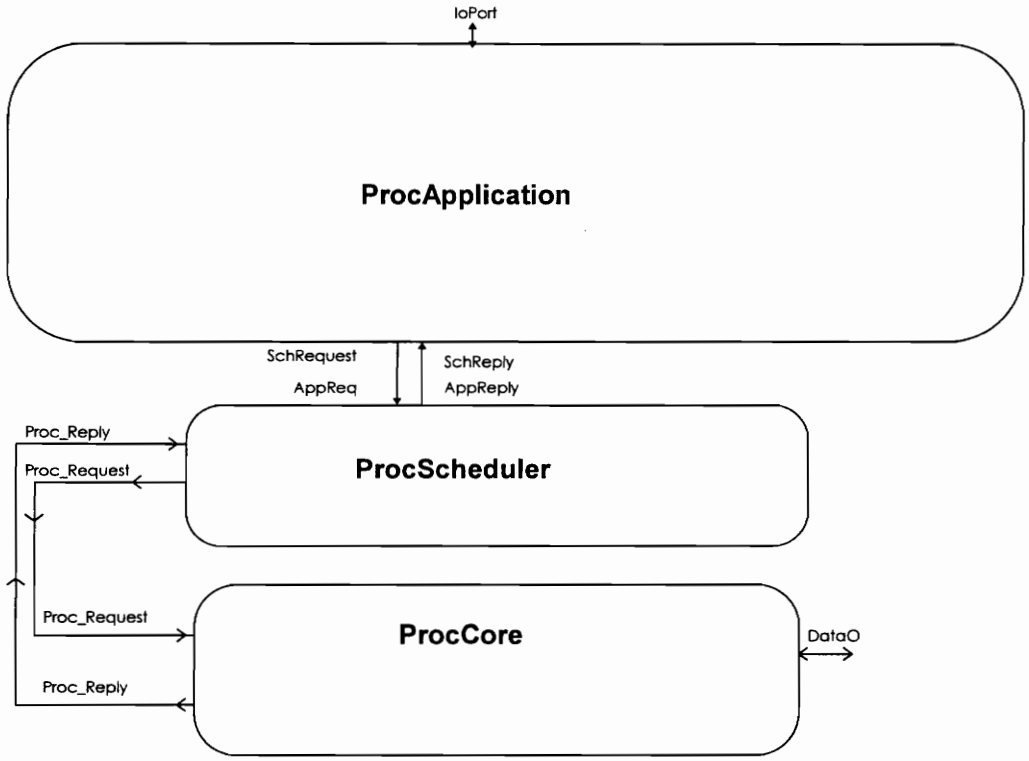


Figure 4.1 Structural Schematic of the Processor Model

4.1.1 Processor Application [5]

The processor application component instantiates the application-specific algorithmic behavior, and directs the communications between the software tasks and the scheduler. Requests to the scheduler are routed from the software tasks via a resolved software token bus. Replies back to the tasks are routed through an array of unresolved software token signals, which take as an index the destination task's ID number.

4.1.2 Processor Scheduler [5]

The Scheduler releases the specific "software" tasks for execution on the performance model of the processor. Task instruction requests come to the scheduler in

the form of software tokens. The scheduler manages the task requests from the software applications by assigning each new request to a thread. The threads are always handled in order of task priority. Instructions specifically addressed to the scheduler (schedule task, change priority, begin job, etc.) are handled by the scheduler, and the results are then passed back up to the applications through a new software token. Processor related instructions (interrupt operations, caching operations, hardware operations) are passed through the scheduler, down to the processor core. The results from the processor core are then passed back up through the scheduler.

4.1.3 Processor Core [5]

The Processor core simulates the hardware of the processor. This is where the processor is characterized, interrupts are handled, instructions are executed (both hardware and software), and inter-processor token communications takes place. Instructions (both hardware and software) are passed down from the processor scheduler. Completed instructions (or partially completed instructions) are passed back up to the scheduler. An instruction can be partially completed if it is interrupted. Hardware instructions that are destined for an off-board component are sent out via the hardware port.

4.2 Hardware Operations

4.2.1 Read

In order for a processor to read from a memory the following sequence of operations need to be performed. Figure 4.2 shows the sequence of operations for a READ operation from PROC 1 to MEM 2. The numbers on the arcs correspond to the sequence of operations in the list below.

1. The application software sends a request for READ to the processor scheduler.
2. The processor scheduler then grants permission for the operation to be performed by the processor.
3. The application software sends the token containing information about the READ operation to the scheduler, which in turn sends the token down to the processor core.
4. The processor sends a *Readtoken* to the destination memory. The memory is busy for a specific period of time to simulate the reading of data. After this time, it sends an acknowledge token to the processor indicating that the READ operation has been completed. It should be noted that this is a two-way process. If the memory is attached to some processor other than the one executing the READ operation, the processor to which the memory is attached does not participate in this exchange of data. In Figure 4.2, note that PROC 2 was not involved in the operation.
5. The results from the processor core are passed back to the applications through the scheduler.

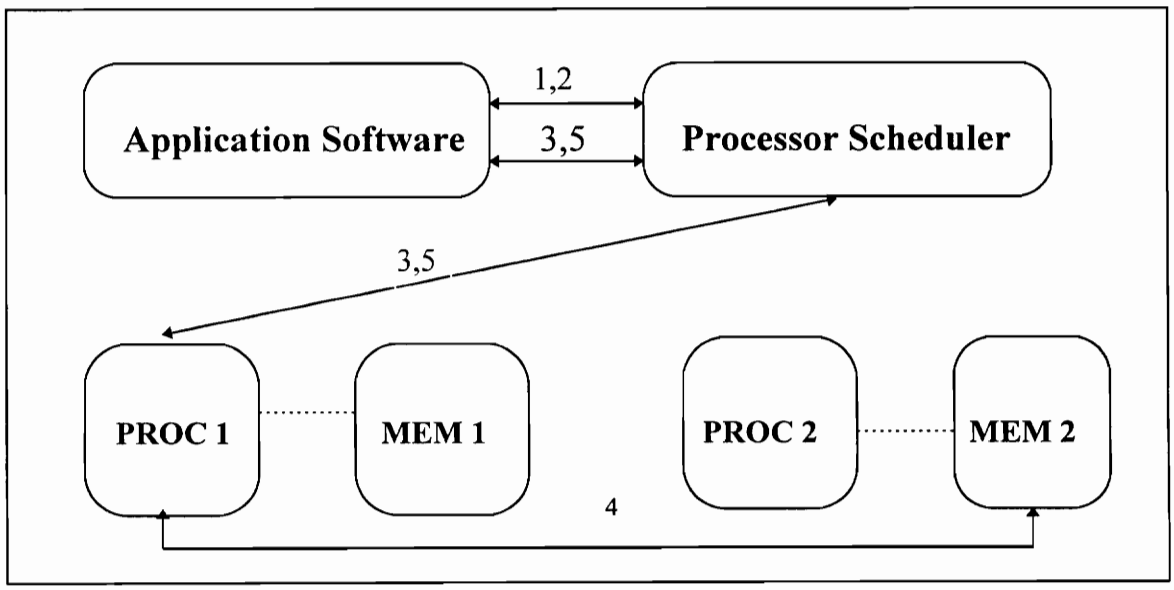


Figure 4.2 READ Operation

4.2.2 Write

In order for a processor to write data into the memory attached to another processor, the following sequence of operations needs to be performed. Figure 4.3 shows the sequence of operations for a WRITE operation from PROC 1 to MEM 2. The numbers on the arcs correspond to the sequence of operations in the list below.

1. The application software sends a request for WRITE to the processor scheduler.
2. The processor scheduler then grants permission for the operation to be performed by the processor.
3. The application software sends the token containing information about the WRITE operation to the scheduler, which in turn sends the token down to the processor core.
4. The processor sends a *Writetoken* to the destination memory. The memory is busy for a specific period of time to simulate the writing of data. After this time, it sends an acknowledge token to the processor indicating that the WRITE operation has been completed. It should be noted that this is a two-way process.
5. The processor to which the memory is attached is not aware of the data being written into its memory. Therefore, the initiating processor must send a *Controltoken* to the processor attached to the memory, notifying it of the data transfer. In Figure 4.3, note that event 5 is unidirectional, i.e. no acknowledge token is returned to PROC1 by PROC2.
6. The results from the processor core are passed back to the applications through the scheduler.

Since the memory cannot initiate a token, the notification to a processor that a WRITE operation has been performed to its memory must be done by the processor writing the data. The initiating processor sends a *Controltoken* to the receiving processor. It is therefore necessary to have three procedures for data transfer operations: READ, WRITE and CONTROL.

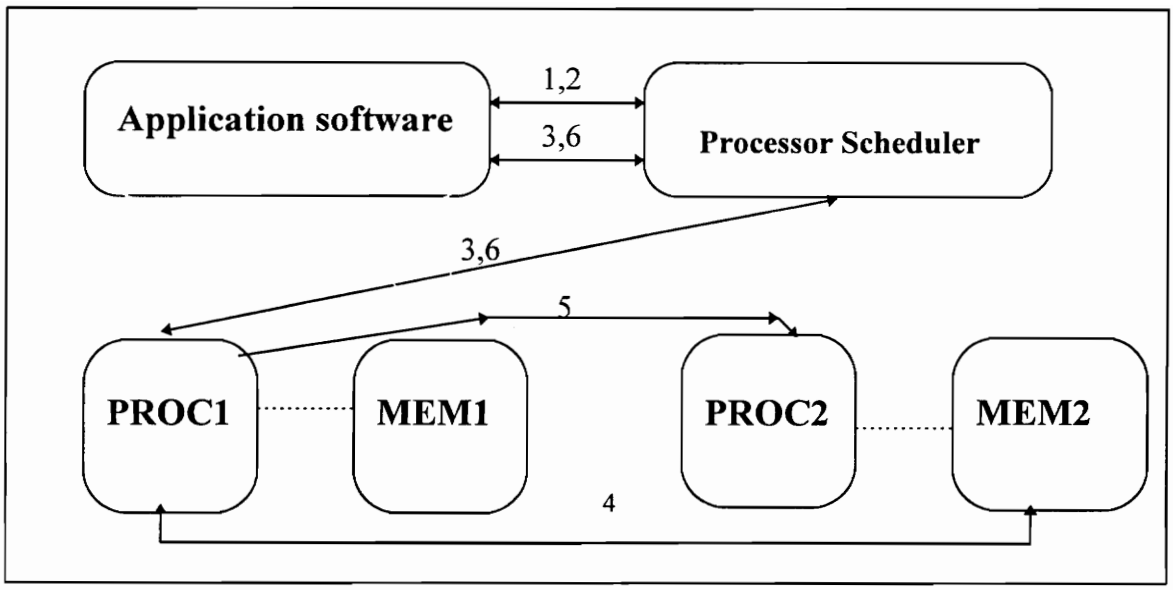


Figure 4.3 WRITE Operation

4.2.3 Procedure HwExecute

The PML library contains a Procedure HwExecute, that is used in the VHDL procedures developed in this Chapter. This procedure is used to send a Hardware token from a processor to another component in a network architecture. The procedure declaration is as follows.

```
PROCEDURE  HwExecute (me: INOUT task_id;
                      SIGNAL send: OUT task_message;
                      SIGNAL receive: utask_message;
                      msg : INOUT task_message;
                      hw_op        : HardwareOp;
                      blocked : BOOLEAN;
                      num_instr: natural;
```

comment: string := "");

PARAM **me** : INOUT task_id

Task ID Number. This number is assigned by the PML software, and should not be changed by the user.

PARAM SIGNAL **send** : OUT task_message

Outgoing signal to the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM SIGNAL **receive** : IN utask_message

Incoming signal from the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM **msg** : INOUT task_message

The hardware token sent down to the processor core. Once the hardware operation is complete, the processor core will send back a new hardware token.

PARAM **hw_op** : HardwareOp

The hardware operation to be performed by the processor core.

PARAM **blocked** : boolean

Boolean value indicating whether or not an acknowledge token has to be sent by the destination to the source.

PARAM **num_instr** : natural

The number of hardware instructions to be simulated on the processor.

PARAM **comment** : string

Textual comment to be added to the log record.

4.3 Constants Package

Some of the parameters that are defined in these procedures are constants. Hence a package of these constants, called Package Constants, was developed in the PROC library. The following constants were defined in the package.

```
CONSTANT cntl_msg_size : data_size := 16 bytes;
CONSTANT rqst_msg_size : data_size := 16 bytes;
```

The request to the processor scheduler to send the *Readtoken* and the *Controltoken* is of the same size for a given configuration irrespective of the size of data being read. Hence, this size was declared as a constant of size 16 bytes in each case. The constant was called rqst_msg_size in the case of the *Readtoken*, and cntl_msg_size in the case of *Controltoken.*

The actual size of the *Controltoken* needs to be a number corresponding to the event being sent by the token. There are 16 events defined in the PML library labeled from EVENT_1 to EVENT_16. Each event is assigned a token data value by the PML software from 19 to 34 respectively. These values are defined as constants in the package.

```
CONSTANT EVENT1 : INTEGER := 19;
CONSTANT EVENT2 : INTEGER := 20;
CONSTANT EVENT3 : INTEGER := 21;
CONSTANT EVENT4 : INTEGER := 22;
CONSTANT EVENT5 : INTEGER := 23;
CONSTANT EVENT6 : INTEGER := 24;
CONSTANT EVENT7 : INTEGER := 25;
CONSTANT EVENT8 : INTEGER := 26;
```

```
CONSTANT EVENT9  : INTEGER := 27;
CONSTANT EVENT10 : INTEGER := 28;
CONSTANT EVENT11 : INTEGER := 29;
CONSTANT EVENT12 : INTEGER := 30;
CONSTANT EVENT13 : INTEGER := 31;
CONSTANT EVENT14 : INTEGER := 32;
CONSTANT EVENT15 : INTEGER := 33;
CONSTANT EVENT16 : INTEGER := 34;
```

The actual size of the *Writetoken* can be any value larger than the values assigned to the *Controltoken*. Hence, this value is made a constant 100, and called write_req.

```
CONSTANT write_req : INTEGER := 100;
```

4.4 VHDL Procedures

The VHDL procedures are declared in the file called subroutines-p.vhdl, with the procedure bodies in the file subroutines-b.vhdl. The samples of code shown below to illustrate the following VHDL procedures were all tested on the 17 processor architecture described in chapter 3. The routing tags and the device names all correspond to the components in the 17 processor architecture.

4.4.1 READ procedure

The READ procedure declaration is as follows.

```
PROCEDURE read (me: INOUT task_id;
                SIGNAL send: OUT task_message;
                SIGNAL receive: utask_message;
```

msg : INOUT task_message;

size : data_size;

value : INTEGER;

destination : string;

route : string;

reroute : string;

blocked : BOOLEAN;

instr: natural;

comment: string);

PARAM **me** : INOUT task_id

Task ID Number. This number is assigned by the PML software, and should not be changed by the user.

PARAM SIGNAL **send** : OUT task_message

Outgoing signal to the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM SIGNAL **receive** : IN utask_message

Incoming signal from the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM **msg** : INOUT task_message

The hardware token sent down to the processor core. Once the hardware operation is complete, the processor core will send back a new hardware token.

PARAM **size** : data_size

The size of the request token sent to the scheduler

PARAM **value** : INTEGER

The size of the actual data to be read by the processor.

PARAM **destination** : string

The name of the memory from which data is to be read.

PARAM **route** : string

The route from the processor to the memory for the token.

PARAM **reroute** : string

The return route from the memory to the processor.

PARAM **blocked** : boolean

Boolean value indicating whether or not an acknowledge token has to be sent by the destination to the source.

PARAM **instr** : natural

The number of hardware instructions to be simulated on the processor.

PARAM **comment** : string

Textual comment to be added to the log record.

The READ procedure sends a signal **send** from the application software to the processor scheduler, requesting a read operation by the processor. The scheduler responds with a signal **receive** to the application software, granting permission for the operation. Then, a token **msg** of type task_message is sent to the processor core, along with the information required for the operation to be performed. The processor core sends an acknowledge token back to the application software once the operation has been performed. The **destination** is the memory from which the data has to be read. The **route**, as the name suggests, is the route the *Readtoken* needs to follow to reach the destination memory from the processor. The **reroute** is the return route that the acknowledge token needs to follow from the memory back to the processor. The **size** is the size of the request token for read operation, and is always assigned the value of rqst_msg_size from the package constants. The **value** is the actual amount of data in bytes that needs to be read from the destination memory. The parameter **me** is the task_id generated by the application software and remains constant throughout the operation. This appears in the log record, and is used to track the tokens pertaining to a single operation. This should

not be changed by the user. The parameter **blocked** is a boolean value indicating whether or not the destination of the token should send an acknowledge token back to the source. A value of TRUE indicates that at the end of the operation a token is returned to the host processor indicating the completion of the task. A value of FALSE indicates that no token is returned to the host processor on completion of task. Hence, a direct inference is that if the value of BLOCKED is FALSE, no return route needs to be provided to the destination memory or processor. The parameter **instr** denotes the number of instructions to be simulated on the processor, and the parameter **comment** appears in the log record, and indicates the operation being performed by the processor. This is included for documentation purposes.

The PML procedure HwExecute is called in the body of the procedure. This procedure executes a hardware operation (READ or WRITE) within the processor core. The token type is assigned to be *Readtoken*, and the hardware operation to be performed by the processor core is assigned as a READ.

The procedure body is as follows.

```
PROCEDURE read (me: INOUT task_id;
                SIGNAL send: OUT task_message;
                SIGNAL receive: utask_message;
                msg : INOUT task_message;
                size        : data_size;
                value       : INTEGER;
                destination : string;
                route       : string;
                reroute     : string;
                blocked : BOOLEAN;
                instr: natural;
```

```
                comment: string ) is
variable m,n,p : integer := 0;
variable temp_token : token;

BEGIN
    m := destination'length;
    n := route'length;
    temp_token.destination := (OTHERS => NUL);
    temp_token.route := (OTHERS => NUL);
    temp_token.parm1_real := (others =>NUL);
    temp_token.destination(1 TO m) := destination(1 TO m);
    temp_token.route(1 TO n)      := route(1 TO n);
            if ((m /= n) and (blocked = true)) then
            p := reroute'length;
            temp_token.parm1_real(1 TO p)      := reroute;
            end if;
    temp_token.size       := size;
    temp_token.value      := value;
    temp_token.t_TYPE     := READTOKEN;
    msg.proc_msg.token_msg := temp_token;
    HwExecute (me, send, receive, msg, HW_READ, blocked, instr, comment);
END read;
```

The code needed to accomplish a READ operation without using the READ procedure is as follows.

```
temp_token.destination := (OTHERS => NUL); -- clear the field to NUL
temp_token.destination(1 TO 7) := "B16_Mem";
```

```
temp_token.route := (OTHERS => NUL); -- clear the field to NUL
temp_token.route(1 TO 7)      := "B16_Mem";
temp_token.size       := 16 byte;    --  read request
temp_token.value      := 16256;   --  amount of data to be read in bytes
temp_token.t_TYPE     := READTOKEN;
msg.proc_msg.token_msg := temp_token;
HwExecute(me, send, receive, msg, HW_READ, TRUE, 1, "Pole Read");
```

This is a read operation of 16256 bytes from B16_Mem. The route is the same as the destination since it is attached to the processor doing the read, and hence there is no necessity for the token to pass through a network crossbar. The token type is initialized to *Readtoken* and the operation being performed is the Pole Read. The procedure call for this read operation looks like this.

read(me, send, receive, msg, rqst_msg_size, 16256, “B16_Mem", “B16_Mem", “”, TRUE, 1, “Pole Read”);

4.4.2 WRITE procedure

The WRITE procedure declaration is as follows.

```
PROCEDURE  write (me: INOUT task_id;
                  SIGNAL send: OUT task_message;
                  SIGNAL receive: utask_message;
                  msg : INOUT task_message;
                  size          : data_size;
                  value         : INTEGER;
                  destination   : string;
```

route : string;

reroute : string;

blocked : BOOLEAN;

instr: natural;

comment: string);

PARAM **me** : INOUT task_id

Task ID Number. This number is assigned by the PML software, and should not be changed by the user.

PARAM SIGNAL **send** : OUT task_message

Outgoing signal to the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM SIGNAL **receive** : IN utask_message

Incoming signal from the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM **msg** : INOUT task_message

The hardware token sent down to the processor core. Once the hardware operation is complete, the processor core will send back a new hardware token.

PARAM **size** : data_size

The size of the actual data to be written by the processor.

PARAM **value** : INTEGER

The size of the request token sent to the scheduler.

PARAM **destination** : string

The name of the memory from which data is to be written.

PARAM **route** : string

The route from the processor to the memory for the token.

PARAM **reroute** : string

The return route from the memory to the processor.

PARAM **blocked** : boolean

Boolean value indicating whether or not an acknowledge token has to be sent by the destination to the source.

PARAM **instr** : natural

The number of hardware instructions to be simulated on the processor.

PARAM **comment** : string

Textual comment to be added to the log record.

The WRITE procedure sends a signal **send** from the application software to the processor scheduler, requesting a write operation by the processor. The scheduler responds with a signal **receive** to the application software, granting permission for the operation. Then, a token **msg** of type task_message is sent to the processor core, along with the information required for the operation to be performed. The processor core sends an acknowledge token back to the application software once the operation has been performed. The **destination** is the memory to which the data has to be written. The **route**, as the name suggests, is the route the *Writetoken* needs to follow to reach the destination memory from the processor. The **reroute** is the return route that the acknowledge token needs to follow from the memory back to the processor. The **value** is the size of the request token for the write operation, and is always assigned the value of write_req from the package constants. Parameter **size** is the actual amount of data in bytes that needs to be written into the destination memory. **me**, **blocked**, **instr** and **comment** have the same functions as in the READ procedure.

In the body of the procedure, the PML procedure HwExecute is called. The token type is assigned to be *Writetoken*, and the hardware operation to be performed by the processor core is assigned as a WRITE.

The procedure body is as follows.

```
PROCEDURE  write (me: INOUT task_id;
                  SIGNAL send: OUT task_message;
                  SIGNAL receive: utask_message;
                  msg : INOUT task_message;
                  size        : data_size;
                  value       : INTEGER;
                  destination   : string;
                  route       : string;
                  reroute       : string;
                  blocked : BOOLEAN;
                  instr: natural;
                  comment: string ) is
variable m,n,p : integer := 0;
variable temp_token : token;

BEGIN
    m := destination'length;
    n := route'length;
    temp_token.destination := (OTHERS => NUL);
    temp_token.route := (OTHERS => NUL);
    temp_token.parm1_real := (others =>NUL);
    temp_token.destination(1 TO m) := destination(1 TO m);
    temp_token.route(1 TO n)      := route(1 TO n);
                  if ((m /= n) and (blocked = true)) then
                  p := reroute'length;
                  temp_token.parm1_real(1 TO p)      := reroute;
                  end if;
```

```
    temp_token.size      := size;
    temp_token.value     := value;
    temp_token.t_TYPE    := WRITETOKEN;
    msg.proc_msg.token_msg := temp_token;
    HwExecute (me, send, receive, msg, HW_WRITE, blocked, instr, comment);
END write;
```

The code needed to accomplish a WRITE operation without using the WRITE procedure is as follows. The Vantage simulator concatenates strings at the end of some component names, like the processor for instance. A '?*' is added at the end of each component name so that it acts as a wildcard character in order to achieve a match with the names concatenated by the simulator.

```
temp_token.destination := (OTHERS => NUL); -- clear the field to NUL
temp_token.destination(1 TO 6) := "B1_Mem";
temp_token.route := (OTHERS => NUL); -- clear the field to NUL
temp_token.route(1 TO 16)     := "_CBar1_?*.B1_Mem";
temp_token.parm1_real := (others =>NUL); -- clear the field to NUL
temp_token.parm1_real(1 to 22) :=  "_CBar5_?*.B16_CPU16_?*";
temp_token.size      := 8140 byte;    --  number of bytes to be written
temp_token.value     := 100;   --  write request
temp_token.t_TYPE    := WRITETOKEN;
msg.proc_msg.token_msg := temp_token;
HwExecute(me, send, receive, msg, HW_WRITE, TRUE, 1, "Even Write");
```

This is a write operation of 8140 bytes to B1_Mem. The route is assigned to the appropriate token field, and so is the return route. The description of the working of the routing tag follows. When a source is communicating with the destination, the path to be

taken by the token is specified in the **route** field of the token. In the example above, the source is connected to CBAR1 and the destination is also connected to CBAR1. Then the route field of the token is given by CBAR1.destination. As the token passes through the crossbar, that particular device name is stripped off and the token continues to its destination. The crossbar picks up any token that matches its globalid and rejects the tokens which match its screenid. This is done to make sure that the crossbar does not pick up any token from a port where the source and destination are connected to the same port. The following example makes the concept clear.

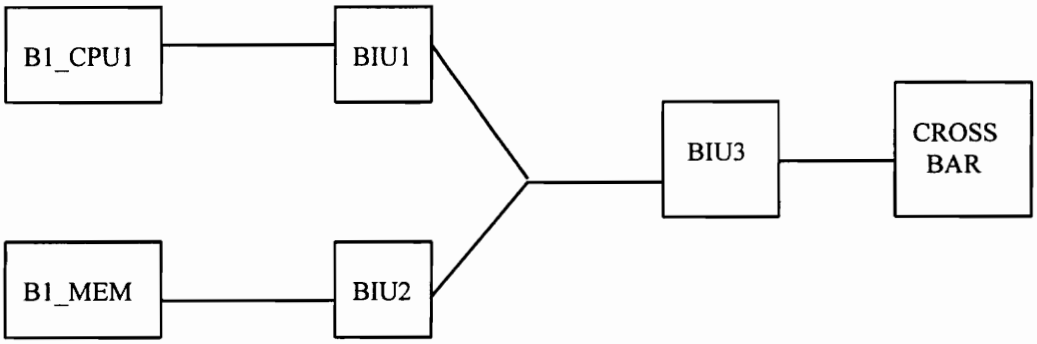


Figure 4.4 Example to illustrate routing scheme

In Figure 4.4, suppose a token is being sent from B1_CPU1 to B1_MEM. The generic globalid for BIU1 and BIU2 is set as "B1_CPU1" and "B1_MEM" respectively meaning, these BIUs are going to pick up any token which has a destination matching their globalid and filter all other tokens. The globalid for BIU3 should be given as "?*" indicating that this BIU is going to pick up all tokens irrespective of the destination. This is done because the crossbar has many components connected to it and it is not possible to give just a single instance name for the globalid. In the above example, the token being sent from B1_CPU1 to B1_MEM would also be picked up not only by BIU2 but also by BIU3 which is then routed to the crossbar. In order to avoid this, screenid for this BIU3 is set to "B1_?*" so that this BIU does not pick up the token being sent from B1_CPU1

which has a destination field as "B1_MEM" and hence is filtered before reaching the crossbar.

The procedure call for the write operation described earlier follows.

write(me, send, receive, msg, 8140, write_req, "B1_Mem", "_CBar1_?*.B1_Mem ", "_CBar5_?*.B16_CPU16_?*", TRUE, 1, "Even Write");

4.4.3 CONTROL procedure

The CONTROL procedure declaration is as follows.

```
PROCEDURE control (me: INOUT task_id;
                   SIGNAL send: OUT task_message;
                   SIGNAL receive: utask_message;
                   msg : INOUT task_message;
                   size        : data_size;
                   value       : INTEGER;
                   destination : string;
                   route       : string;
                   instr: natural;
                   comment: string );
```

PARAM **me** : INOUT task_id

Task ID Number. This number is assigned by the PML software, and should not be changed by the user.

PARAM SIGNAL **send** : OUT task_message

Outgoing signal to the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM SIGNAL **receive** : IN utask_message

Incoming signal from the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM **msg** : INOUT task_message

The hardware token sent down to the processor core. Once the hardware operation is complete, the processor core will send back a new hardware token.

PARAM **size** : data_size

The size of the request token sent to the scheduler.

PARAM **value** : INTEGER

The index corresponding to the event being sent.

PARAM **destination** : string

The name of the processor to which *Controltoken* is to be sent.

PARAM **route** : string

The route from the source processor to the destination processor for the token.

PARAM **instr** : natural

The number of hardware instructions to be simulated on the processor.

PARAM **comment** : string

Textual comment to be added to the log record.

This procedure notifies the processor about the data that has been written into the memory attached to it. The procedure follows the usual protocol for executing a hardware operation by the processor core. Hence, the parameters **me**, **send**, **receive**, **msg**, **route**, **destination**, **instr** and **comment** have the same functions as in the READ and WRITE operations. There is no acknowledge token ever sent, and so the parameter **blocked**

becomes redundant as it is always assigned a value of FALSE. This parameter is not included. Instead, the information is built into the procedure itself. Consequently, the parameter **reroute** is not included either. The parameter **size** is the size of the request token for the CONTROL operation, and is always assigned a value of cntl_msg_size from the package constants. The parameter **value** needs to have the index of the event assigned to it, and this is also done using the constants defined in the package constants.

In the body of the procedure, the PML procedure HwExecute is called. The token type is assigned to be *Controltoken*, and the hardware operation to be performed by the processor core is assigned as a WRITE.

The procedure body is as follows.

```
PROCEDURE control (me: INOUT task_id;
                   SIGNAL send: OUT task_message;
                   SIGNAL receive: utask_message;
                   msg : INOUT task_message;
                   size        : data_size;
                   value       : INTEGER;
                   destination : string;
                   route       : string;
                   instr: natural;
                   comment: string ) is
variable m,n,p : integer := 0;
variable temp_token : token;

BEGIN
    m := destination'length;
    n := route'length;
```

```
    temp_token.destination := (OTHERS => NUL);
    temp_token.route := (OTHERS => NUL);
    temp_token.destination(1 TO m) := destination(1 TO m);
    temp_token.route(1 TO n)      := route(1 TO n);
    temp_token.size       := size;
    temp_token.value      := value;
    temp_token.t_TYPE     := CONTROLTOKEN;
    msg.proc_msg.token_msg := temp_token;
    HwExecute (me, send, receive, msg, HW_WRITE, false, instr, comment);
END control;
```

The code needed to accomplish a CONTROL operation without using the CONTROL procedure is as follows:

```
    temp_token.destination := (OTHERS => NUL);
    temp_token.route := (OTHERS => NUL);

    temp_token.destination(1 TO 10) := "B1_CPU1_?*";
    temp_token.route(1 TO 20)      := "_CBar1_?*.B1_CPU1_?*";
    temp_token.size       := 1 byte;  --  don't care
    temp_token.value      := 20 ;     -- EVENT_2
    temp_token.t_type     := CONTROLTOKEN; -- interrupt_token
    msg.proc_msg.token_msg := temp_token;
      HwExecute(me, send, receive, msg, HW_WRITE, FALSE, 1, "Trying to send
EVENT_2..");
```

This is the control operation performed corresponding to the write operation described above. The token type is initialized to *Controltoken* and the operation being performed is Event 2. The procedure call for this control operation looks like this.

control(me, send, receive, msg, cntl_msg_size, EVENT2, "B1_Mem", "_CBar1_?*.B1_Mem ", 1, "Trying to send Event 2");

The Read, Write and Control procedures are used for conducting all basic intercomponent communications. But there are some other functions which occur frequently enough to justify writing a procedure for them. These functions are much more complicated than the Read, Write and Control procedures already defined. They are used to perform a complex function involving several reads and writes to different destinations in particular orders. There are two such procedures- Distribute and Broadcast.

4.4.4 Procedure DISTRIBUTE

The DISTRIBUTE procedure performs the following function. It has a host processor, along with a memory attached to it. It also has an array of target processors and the memories attached to these processors. Each time the host processor receives a token, it sends the token to one processor in the array. The next token is sent to the next processor in the array, and so on. In this fashion, consecutive tokens are sent to consecutive processors in a round robin fashion. Once all the processors have received a token each, the array of destinations wraps around. That is, the first processor in the array receives the next token again, and this process continues. It has to be noted here, that only one processor receives each token.

Figure 4.5 shows the working schematic of a Distribute operation.

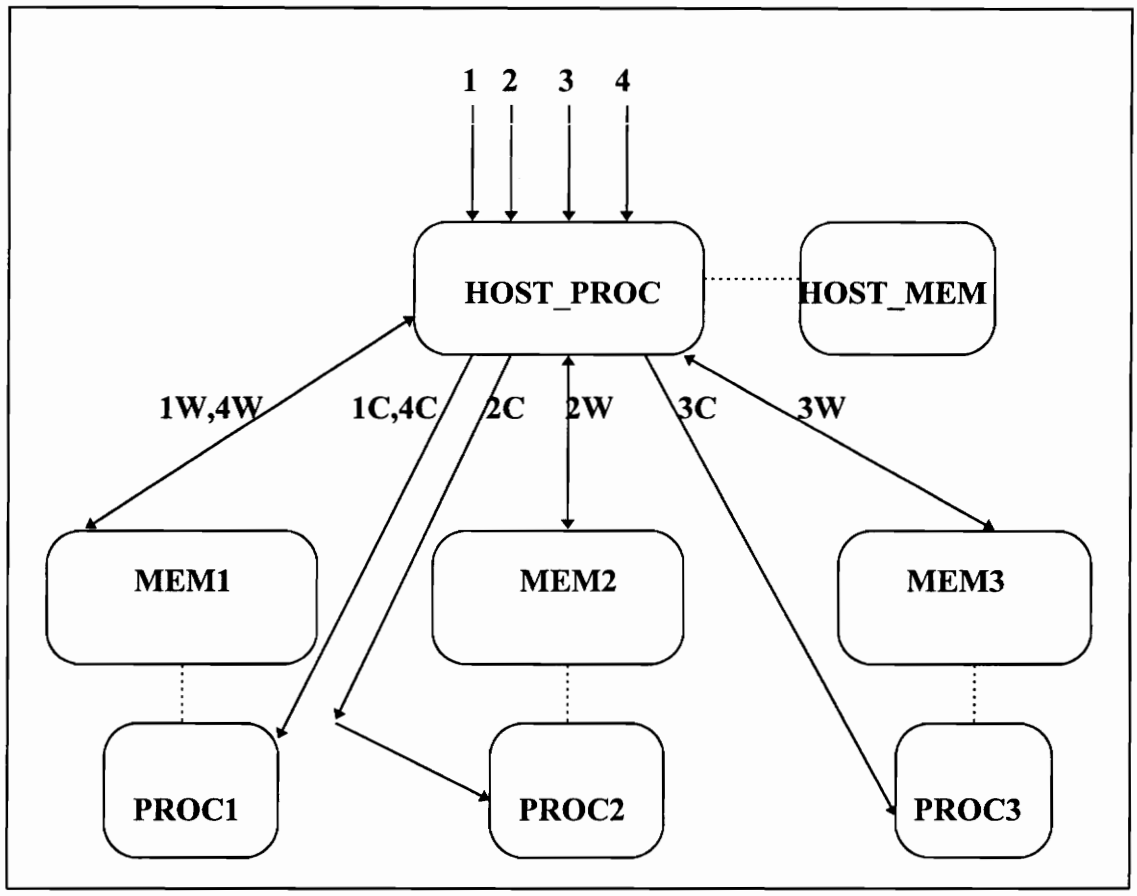


Figure 4.5 Distribute Operation

1,2,3 and **4** represent tokens in their order of occurrence. The host processor is HOST_PROC, and the memory attached to it is denoted by HOST_MEM. The destination processors are PROC1, PROC2 and PROC3, with the memories attached to them being MEM1, MEM2, and MEM3 respectively. The host processor receives tokens **1,2,3** and **4** in that order. It sends token 1 to the first processor PROC1. That is, it writes token 1 to the memory attached to the first processor: MEM1. This operation is done using a *Writetoken*, and is denoted by **1W**. Then, the host processor sends a notification to the processor attached to this memory about the data written into it. This operation is done using a *Controltoken*, and is denoted by **1C**. Similarly, the subsequent tokens are

written into the memories attached to the subsequent destination processors. These operations are denoted by **2W, 3W** and **4W** respectively. The *Controltokens* sent to these processors notifying them of the write operation, are denoted by **2C, 3C** and **4C** respectively. It should be noted that the fourth token is sent back to the first destination after a wrap around of the array occurs. Also, the write operations are bidirectional. That is, there is a token sent back from the memories to the processor. But the control operations are unidirectional. That is, the tokens are sent only from the host processor to the destination processor.

The DISTRIBUTE procedure declaration is as follows.

```
PROCEDURE distribute (me: INOUT task_id;
                      SIGNAL send: OUT task_message;
                      SIGNAL receive: utask_message;
                      msg : INOUT task_message;
                      dest_index    : INOUT INTEGER ;
                      dest_count    : INTEGER;
                      control_size        : data_size;
                      message_size        : data_size;
                      value        : INTEGER;
                      event_index        : INTEGER;
                      msg_destination    : table_typ;
                      event_destination    : table_typ;
                      msg_route        : table_typ;
                      event_route        : table_typ;
                      msg_reroute        : table_typ;
                      instr: natural;
                      hosts : resource_name;
```

msg_comment: string;

event_comment: string);

PARAM **me** : INOUT task_id

Task ID Number. This number is assigned by the PML software, and should not be changed by the user.

PARAM SIGNAL **send** : OUT task_message

Outgoing signal to the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM SIGNAL **receive** : IN utask_message

Incoming signal from the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM **msg** : INOUT task_message

The hardware token sent down to the processor core. Once the hardware operation is complete, the processor core will send back a new hardware token.

PARAM **dest_index** : INOUT INTEGER

The index of the latest destination.

PARAM **dest_count** : INTEGER

The total number of destinations.

PARAM **control_size** : data_size

The size of the request sent to the scheduler for the control operation.

PARAM **message_size** : data_size

The actual size of the data to be written by the host processor for the write operation.

PARAM **value** : INTEGER

The size of the request sent to the scheduler for the write operation.

PARAM **event_index** : INTEGER

The index corresponding to the event being sent for the control operation.

PARAM **msg_destination** : table_typ

An array of strings containing the names of the memories into which tokens are to be written, in the order desired.

PARAM **event_destination** : table_typ

An array of strings containing the names of the processors which need to be notified about data written into their memories. These are to be in the order corresponding to that of **msg_destination**.

PARAM **msg_ route** : table_typ

An array of strings containing the routes from the host processor to the destination memories for the write token. These are to be in the order corresponding to that of **msg_destination**.

PARAM **event_ route** : table_typ

An array of strings containing the routes from the host processor to the destination processors for the control token. These are to be in the order corresponding to that of **event_destination**.

PARAM **msg_ reroute** : table_typ

An array of strings containing the return routes from the destination memories to the host processor for the write token. These are to be in the order corresponding to that of **msg_route**.

PARAM **instr** : natural

The number of hardware instructions to be simulated on the processor.

PARAM **hosts** : resource_name

The name of the host processor.

PARAM **msg_comment** : string

Textual comment to be added to the log record for the write operations.

PARAM **event_comment** : string

Textual comment to be added to the log record for the control operations.

This procedure sends tokens to a predetermined set of destinations, on a round robin basis. The procedure follows the usual protocol for executing a hardware operation by the processor core. Hence, the parameters **me**, **send**, **receive**, **msg** and **instr** have the same function as in the READ, WRITE and CONTROL operations. The write operations always return an acknowledge token, and the control operations never return an acknowledge token. So the parameter **blocked** becomes redundant as it is always assigned a value of TRUE for the write operations, and FALSE for the control operations. This parameter is not included. Instead, the information is built into the procedure itself. The parameter **control_size** is the size of the request for the *Controltoken*, and is always assigned a value of cntl_msg_size from the package constants. The parameter **message_size** is the actual size of data that needs to be written into the destination memory. The parameter **value** is the size of the request for the write operation, and is always assigned the value of write_req from the package constants. The parameter **event_index** needs to have the index of the control event assigned to it, and this is also done using the constants defined in the package constants. The parameters **msg_destination**, **event_destination**, **msg_route**, **event_route**, **msg_reroute** are all arrays of strings, which contain in the same order, the names of the destination memories, destination processors, routes to the destination memories, routes to the destination processors, and return routes from the destination memories respectively. There is no token returning from the destination processors, and so there is no parameter for the return route from the destination processors. The parameter **hosts** has the name of the host processor assigned to it. The parameters **msg_comment** and **event_comment** appear in the log record, and indicate the operation being performed by the processor during the write and control operation respectively. These are included for documentation purposes.

The algorithm followed in the distribute procedure is as follows.

1. If DEST_INDEX = DEST_COUNT, then initialize DEST_INDEX = 0.
2. DEST_INDEX = DEST_INDEX + 1.
3. Initialize Write token fields destination, route and reroute with corresponding entries in MSG_DESTINATION, MSG_ROUTE and MSG_REROUTE according to DEST_INDEX.
4. Send Write token.
5. Initialize Control token fields destination and route with corresponding entries in EVENT_DESTINATION and EVENT_ROUTE according to DEST_INDEX.
6. Send Control token.

The procedure body is as follows.

```
PROCEDURE  distribute (me: INOUT task_id;
                       SIGNAL send: OUT task_message;
                       SIGNAL receive: utask_message;
                       msg : INOUT task_message;
                       dest_index    : INOUT INTEGER ;
                       dest_count    : INTEGER;
                       control_size        : data_size;
                       message_size        : data_size;
                       value       : INTEGER;
                       event_index       : INTEGER;
                       msg_destination   : table_typ;
                       event_destination   : table_typ;
                       msg_route       : table_typ;
                       event_route       : table_typ;
                       msg_reroute       : table_typ;
                       instr: natural;
```

```
                hosts : resource_name;
                msg_comment: string;
                event_comment: string ) is
variable m,n,p : integer := 0;
variable temp_token : token;

BEGIN
                IF (dest_index = dest_count) THEN
                dest_index := 0;
                END IF;
    dest_index := dest_index+1;
    temp_token.destination := (OTHERS => NUL);
    temp_token.route := (OTHERS => NUL);
    temp_token.parm1_real := (others =>NUL);
    temp_token.destination := msg_destination(dest_index);
    temp_token.route      := msg_route(dest_index);
    temp_token.parm1_real := msg_reroute(dest_index);
    temp_token.size       := message_size;
    temp_token.value      := value;
    temp_token.t_TYPE     := WRITETOKEN;
    msg.proc_msg.token_msg := temp_token;
    HwExecute(me, send, receive, msg, HW_WRITE, TRUE, instr, msg_comment);

    temp_token.destination := (OTHERS => NUL);
    temp_token.route := (OTHERS => NUL);
    temp_token.parm1_real := (others =>NUL);
    temp_token.destination := event_destination(dest_index);
    temp_token.route      := event_route(dest_index);
```

```
temp_token.size      := control_size;
temp_token.value     := event_index;
temp_token.t_TYPE    := CONTROLTOKEN;
msg.proc_msg.token_msg := temp_token;
HwExecute(me, send, receive, msg, HW_WRITE, FALSE, instr, event_comment);

        IF (me.host = hosts) THEN
        ASSERT FALSE REPORT "Event sent to " & image(hosts)
        SEVERITY NOTE;
        ELSE
        ASSERT FALSE REPORT "Illegal Host Name encountered"
        SEVERITY ERROR;
        END IF;

END distribute;
```

See Appendix D for VHDL code before and after using the DISTRIBUTE procedure.

4.4.5 Procedure BROADCAST

The BROADCAST procedure performs the following function. It has a host processor, along with a memory attached to it. It also has an array of target processors and the memories attached to these processors. Each time the host processor receives a token, it sends the token to all the processors in the array.

Figure 4.6 shows the working schematic of a Broadcast operation.

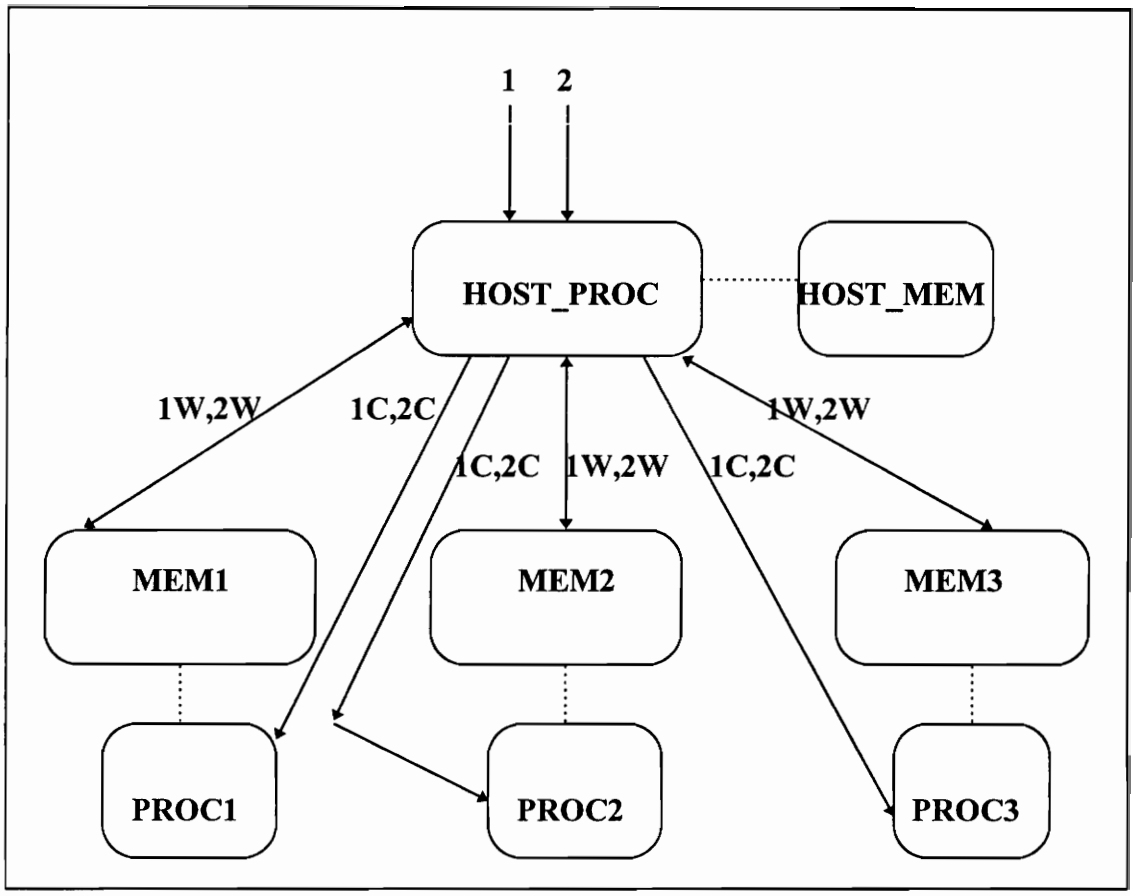


Figure 4.6 Broadcast Operation

1 and **2** represent tokens in their order of occurrence. The host processor is HOST_PROC, and the memory attached to it is denoted by HOST_MEM. The destination processors are PROC1, PROC2 and PROC3, with the memories attached to them being MEM1, MEM2, and MEM3 respectively. The host processor receives tokens **1** and **2** in that order. It sends token 1 to all the destination processors PROC1, PROC2 and PROC3. That is, it writes token 1 to the memories attached to every processor: MEM1, MEM2 and MEM3. This operation is done using a *Writetoken*, and is denoted by **1W**. Then, the host processor sends a notification to the processors attached to these memories about the data written into them. This operation is done using a *Controltoken*, and is denoted by **1C**. Similarly, the subsequent token **2** is also written into all the memories and the Control tokens are sent to these processors notifying them of the write

operation. It should be noted that the write operations are bidirectional. That is, there is a token sent back from the memories to the processor. But the control operations are unidirectional. That is, the tokens are sent only from the host processor to the destination processor.

The BROADCAST procedure declaration is as follows.

```
PROCEDURE  broadcast (me: INOUT task_id;
                      SIGNAL send: OUT task_message;
                      SIGNAL receive: utask_message;
                      msg : INOUT task_message;
                      dest_count    : INTEGER;
                      control_size          : data_size;
                      message_size          : data_size;
                      value         : INTEGER;
                      event_index           : INTEGER;
                      msg_destination     : table_typ;
                      event_destination     : table_typ;
                      msg_route         : table_typ;
                      event_route           : table_typ;
                      msg_reroute           : table_typ;
                      instr: natural;
                      hosts : resource_name;
                      msg_comment: string;
                      event_comment: string );

    PARAM     me : INOUT task_id
```

Task ID Number. This number is assigned by the PML software, and should not be changed by the user.

PARAM SIGNAL **send** : OUT task_message

Outgoing signal to the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM SIGNAL **receive** : IN utask_message

Incoming signal from the scheduler. This signal is defined by the PML software, and should not be changed by the user.

PARAM **msg** : INOUT task_message

The hardware token sent down to the processor core. Once the hardware operation is complete, the processor core will send back a new hardware token.

PARAM **dest_count** : INTEGER

The total number of destinations.

PARAM **control_size** : data_size

The size of the request sent to the scheduler for the control operation.

PARAM **message_size** : data_size

The actual size of the data to be written by the host processor for the write operation.

PARAM **value** : INTEGER

The size of the request sent to the scheduler for the write operation.

PARAM **event_index** : INTEGER

The index corresponding to the event being sent for the control operation.

PARAM **msg_destination** : table_typ

An array of strings containing the names of the memories into which tokens are to be written, in the order desired.

PARAM **event_destination** : table_typ

An array of strings containing the names of the processors which need to be notified about data written into their memories. These are to be in the order corresponding to that of **msg_destination**.

PARAM **msg_ route** : table_typ

An array of strings containing the routes from the host processor to the destination memories for the write token. These are to be in the order corresponding to that of **msg_destination**.

PARAM **event_ route** : table_typ

An array of strings containing the routes from the host processor to the destination processors for the control token. These are to be in the order corresponding to that of **event_destination**.

PARAM **msg_ reroute** : table_typ

An array of strings containing the return routes from the destination memories to the host processor for the write token. These are to be in the order corresponding to that of **msg_route**.

PARAM **instr** : natural

The number of hardware instructions to be simulated on the processor.

PARAM **hosts** : resource_name

The name of the host processor.

PARAM **msg_comment** : string

Textual comment to be added to the log record for the write operations.

PARAM **event_comment** : string

Textual comment to be added to the log record for the control operations.

This procedure sends every token received to a predetermined set of destinations. The procedure follows the usual protocol for executing a hardware operation by the processor core. Hence, the parameters **me**, **send**, **receive**, **msg** and **instr** have the same

function as in the READ, WRITE and CONTROL operations. The write operation always returns an acknowledge token, and the control operation never returns an acknowledge token. So the parameter **blocked** becomes redundant as it is always assigned a value of TRUE for the write operations, and FALSE for the control operations. This parameter is not included. Instead, the information is built into the procedure itself. The parameter **control_size** is the size of the request for the *Controltoken*, and is always assigned a value of cntl_msg_size from the package constants. The parameter **message_size** is the actual size of data that needs to be written into the destination memory. The parameter **value** is the size of the request for the write operation, and is always assigned the value of write_req from the package constants. The parameter **event_index** needs to have the index of the control event assigned to it, and this is also done using the constants defined in the package constants. The parameters **msg_destination**, **event_destination**, **msg_route**, **event_route**, **msg_reroute** are all arrays of strings, which contain in the same order, the names of the destination memories, destination processors, routes to the destination memories, routes to the destination processors, and return routes from the destination memories respectively. There is no token returning from the destination processors, and so there is no parameter for the return route from the destination processors. The parameter **hosts** is the name of the host processor assigned to it. The parameters **msg_comment** and **event_comment** appear in the log record, and indicate the operation being performed by the processor during the write and control operation respectively. These are included for documentation purpose.

The algorithm followed in the broadcast procedure is as follows.

1. Initialize I = 1. For I in 1 to DEST_COUNT loop
2. Initialize Write token fields of destination, route and reroute with corresponding entries in MSG_DESTINATION, MSG_ROUTE and MSG_REROUTE according to loop variable I.
3. Send Write token.

4. Initialize Control token fields destination and route with corresponding entries in EVENT_DESTINATION and EVENT_ROUTE according to loop variable I.
5. Send Control token.
6. If I < DEST_COUNT then I = I + 1. Go to step 2. Else, exit.

The procedure body is as follows.

```
PROCEDURE  broadcast (me: INOUT task_id;
                      SIGNAL send: OUT task_message;
                      SIGNAL receive: utask_message;
                      msg : INOUT task_message;
                      dest_count    : INTEGER;
                      control_size        : data_size;
                      message_size        : data_size;
                      value       : INTEGER;
                      event_index       : INTEGER;
                      msg_destination   : table_typ;
                      event_destination   : table_typ;
                      msg_route       : table_typ;
                      event_route       : table_typ;
                      msg_reroute       : table_typ;
                      instr: natural;
                      hosts : resource_name;
                      msg_comment: string;
                      event_comment: string ) is
variable m,n,p : integer := 0;
variable temp_token : token;

BEGIN
```

```
FOR I in 1 to dest_count LOOP
dest_index := dest_index+1;
temp_token.destination := (OTHERS => NUL);
temp_token.route := (OTHERS => NUL);
temp_token.parm1_real := (others =>NUL);
temp_token.destination := msg_destination(I);
temp_token.route      := msg_route(I);
temp_token.parm1_real := msg_reroute(I);
temp_token.size       := message_size;
temp_token.value      := value;
temp_token.t_TYPE     := WRITETOKEN;
msg.proc_msg.token_msg := temp_token;
HwExecute(me, send, receive, msg, HW_WRITE, TRUE, instr, msg_comment);

temp_token.destination := (OTHERS => NUL);
temp_token.route := (OTHERS => NUL);
temp_token.parm1_real := (others =>NUL);
temp_token.destination := event_destination(I);
temp_token.route      := event_route(I);
temp_token.size       := control_size;
temp_token.value      := event_index;
temp_token.t_TYPE     := CONTROLTOKEN;
msg.proc_msg.token_msg := temp_token;
HwExecute(me, send, receive, msg, HW_WRITE, FALSE, instr, event_comment);

        IF (me.host = hosts) THEN
        ASSERT FALSE REPORT "Event sent to " & image(hosts)
        SEVERITY NOTE;
```

ELSE

ASSERT FALSE REPORT "Illegal Host Name encountered"

SEVERITY ERROR;

END IF;

END broadcast;

See Appendix D for VHDL code before and after using the BROADCAST procedure.

The following table summarizes the results of the above VHDL procedure substitutions.

PROCEDURE	Lines of code before procedure substitution	Lines of code after procedure substitution
READ	11	1
WRITE	11	1
CONTROL	9	1
DISTRIBUTE	245	35
BROADCAST	490	57

The VHDL subroutines developed in this chapter have therefore been shown to greatly reduce code complexity. The use of these procedures helps to speed up the development of the application software.

Chapter 5. Conclusions and Future Work

5.1 Conclusions

This thesis has developed a design methodology to interface the VHDL Performance models to the Algorithm Partitioning Tool. Three different interface tools have been constructed and tested to demonstrate the proof of concept of the methodology proposed in chapter 3. These have been tested using two different sample architectures. Results of these sample extractions have been included and documented.

In the second part of this thesis, a set of VHDL procedures have been developed to help interface the APT to the VHDL Performance Models. This was the first step toward automating the labor intensive process of mapping the software tasks from the

APT back to the Performance Models. These subroutines have been described in Chapter 4 of the thesis, and have been documented.

It has been shown that by using the PCET, ACET and CONET for generating script files, the process of extracting parameters required for the APT and converting them into a form directly usable by the APT, is made extremely simple, thereby saving significant time and labor for each iteration and speeding up the design process considerably. The use of the VHDL subroutines developed, in the application software significantly reduces the volume of VHDL code involved in the Task Mapping Process, thereby greatly increasing speed and ease of automation. Overall, the results of this thesis provide a way to almost completely automate the process of finding an optimum computer architecture configuration for a specific application or a generic algorithm.

5.2 Future Work

The following areas could be explored for further refinements and additions to the present work.

1. **Automate Task Mapping Interface**

   The Task Mapping interface between the APT and the Performance Model could be completely automated. The results from the CONET extraction could be used to automatically generate a Routing between any two components, and the VHDL routines could be further customized to achieve easier automation.

2. **User-Friendliness**

   The user-friendliness of the library needs to be improved. Graphical user interfaces (GUI) would be extremely helpful in making the library easy to use for the

user/designer. This need is being met by Omniview, Inc. A fully operational package is scheduled for release in the near future.

3. Pre-customized Library

The customized library components that are analyzable by VTIP can be made readily available to the user in a separate library. A different library with the original components can also be made available to the user. This would give the user the option to use the appropriate library for VTIP or Vantage, thereby saving the user the need to customize the library components.

These two different libraries are being created at Virginia Tech. One library contains the original models for use in simulations of the performance models by the Vantage simulator. A separate library of customized components, which have already been analyzed using VTIP, are also made available to the user. The user is required to analyze only the top level structural model, thereby saving him the need to customize the library.

References

[1] J. R. Armstrong and F. G. Gray, *Structured Logic Design with VHDL*, Prentice Hall, 1993.

[2] *IEEE Standard VHDL Language Reference Manual*, 1988.

[3] H. P. Commissariat, "Performance Modeling of Single Processor and Multi-Processor Computer Architectures", *Master's Thesis*, Virginia Polytechnic Institute and State University, 1995.

[4] F. Rose, T. Steeves, T. Carpenter, Honeywell Technology Center, "VHDL Performance Modeling", *Proceedings of 1st Annual RASSP Conference*, 1994.

[5] Comments from the VHDL code provided by Honeywell inc. GOVERNMENT PURPOSE LICENSE RIGHTS LEGEND CONTRACT NUMBERS: F33615-94-C-1501 (Omniview PMW), DAAL01-93-C-3380 (Martin Marietta RASSP), F33615-94-C-1495 (VHDL Hybrid Models), F33615-92-C-3802 (CDG), CONTRACTOR: Honeywell inc. 27327.

[6] CAD Language Systems, *VHDL Analyzer Designer's Manual*, April 1993.

[7] CAD Language Systems, *Design Library System*, April 1991.

[8] CAD Language Systems, *DLS Application Development : The Software Procedural Interface*, March 1993.

[9] *Algorithm Partitioning Tool Users Manual*, Research Triangle Institute, 1996.

APPENDIX A

This Appendix contains the entity declarations of the components of the PML Library.

InDevice

```
ENTITY InDeviceH IS
 GENERIC ( INST: STRING := "_Sensor";
        DESTINATION_INFO: STRING := "ITEM ?*";
        DUTYCYCLE_INFO: STRING := "CONSTANT 25";
        LATENCY_INFO: STRING := "CONSTANT 1000";
        PRIORITY_INFO: STRING := "CONSTANT 0";
        QUEUE_DEPTH : POSITIVE := 2;
        REFERENCE: STRING := "Spec #";
        SIZE_INFO: STRING := "CONSTANT 1";
        SOURCE_INFO: STRING := "ITEM NULL";
        THROUGHPUT_INFO: STRING := "CONSTANT 100";
        TOKEN_PROTOCOL: protocol_type := handshake;
        TRACE: BOOLEAN := FALSE;
        TTYPE: TOKEN_TYPE:= DATATOKEN;
        UNIT: DATA_SIZE := 64 kbyte;
        VALUE_INFO: STRING := "CONSTANT 25";
        PULSES: NATURAL := 3 ;
        SOME_DELAY_INFO: STRING := "CONSTANT 25000");
 PORT ( DOut: INOUT  token );
END InDeviceH;
```

PORT DOut INOUT token

This is the output port from which all tokens ensue based on the distributions specified in the various generics.

GENERICS :

GEN INST: STRING

This is the name of the component instance. The inst value is the identifier that is placed in the output report files.

GEN DESTINATION_INFO: STRING

This is a GDL string representing the destination for the output tokens.

GEN DUTYCYCLE_INFO: STRING

This is a GDL string representing the percentage of a cycle that the device is active generating a particular output.

GEN LATENCY_INFO: STRING

This is a GDL string representing the length of time it takes the leading wavefront of a token to propagate through this instance.

GEN PRIORITY_INFO: STRING

This is a GDL string representing the change in priority this instance will inflict upon tokens.

GEN QUEUE_DEPTH: POSITIVE

This is the depth of the token output queue.

GEN REFERENCE: STRING

This is a nonfunctional string purely for documenting any references which might be appropriate or useful to the modeller.

GEN SIZE_INFO: STRING

This is a GDL string indicating how large the output token is.

GEN SOURCE_INFO: STRING

This is a GDL string indicating the source of the token.

GEN THROUGHPUT_INFO: STRING

This is a GDL string representing the rate at which tokens of size UNIT are processed through this instance. THROUGHPUT is expressed in Hertz.

GEN TOKEN_PROTOCOL: PROTOCOL_TYPE

This is the protocol of the output token.

GEN TRACE: BOOLEAN

This controls whether this node enters runtime performance information in the output trace file.

GEN TTYPE: TOKEN_TYPE

This specifies type of the token which will be generated.

GEN UNIT: DATA_SIZE

This represents the granularity of operation of this component. The intent is to allow specification in units appropriate or natural for each instance or component. This attribute along with THROUGHPUT is used to determine the amount of time a component instance is busy for each incident token. For instance, if a given data processing node which works at the rate of 100 Hz on 64 kB data packets receives an input of only 32 kB, it will be busy for 5 ms.

GEN VALUE_INFO: STRING

This is a GDL string representing the "value" field in the output token.

GEN PULSES: NATURAL

This gives the number of pulses(tokens) the indevice puts out before waiting for time duration SOME_DELAY_INFO (explained next).

GEN SOME_DELAY_INFO: STRING

This is linked to the above generic PULSES. After sending out the number of pulses specified by the generic PULSES the indevice stays dormant for a delay period given by SOME_DELAY.

Memory

```
ENTITY memoryH IS
        GENERIC ( INST: STRING := "_IO";
                  REPLY_TO : STRING := "_Proc";
                  DMA_DEVICE_INFO : STRING := "ITEM _Sensor";
                  HIT_INFO: STRING := "CONSTANT 1";
                  LATENCY_INFO: STRING := "CONSTANT 1000";
                  PRIORITY_INFO: STRING := "CONSTANT 1";
                  QUEUE_DEPTH : POSITIVE := 2;
```

```
                    REFERENCE: STRING := "Spec # -1";
                    SWITCH_TOKEN: BOOLEAN := FALSE;
                    THROUGHPUT_INFO: STRING := "CONSTANT 100";
                    TOKEN_PROTOCOL: protocol_type := handshake;
                    TRACE: BOOLEAN := FALSE;
                    TXFORM_INFO: STRING := "CONSTANT 1.0";
                    UNIT: DATA_SIZE := 64 kbyte );
  PORT ( IO: INOUT  token );
END memoryH;
```

Important details about the memory device entity, and peculiar to the memory device are explained below:

PORT IO: INOUT token

This is both the input and output port for this element.

GENERICS :

GEN REPLY_TO: STRING

An interrupt is sent to this instance after DMA is completed. This is normally the processor on the local bus of the memory.

GEN DMA_DEVICE_INFO: STRING

This is the name of the device which can perform a DMA operation on the memory. If tokens are received from this device then the acknowledgment token (which will be an interrupt) is sent to the device specified by the REPLY_TO generic string.

GEN HIT_INFO: STRING

This is a GDL string representing the probability that a particular memory request was in the memory.

GEN SWITCH_TOKEN: BOOLEAN

This indicates the iodevice is to switch the source and destination fields of the token and return it.

GEN TXFORM_INFO: STRING

This is a GDL string representing the factor by which incident data is changed (increased or decreased).

Bus Interface Unit (BIU)

Entity Declaration for the BIU

```
LIBRARY GEN;
USE gen.base_types.ALL;

ENTITY comm_intH IS
  GENERIC ( ACK_TIME_INFO:      STRING := "CONSTANT 100";
            BUS_TIMEOUT_INFO:   STRING := "CONSTANT 100";
            GLOBALID:           STRING := "B1_Mem1";
            SCREENID:           STRING := "B1_?*";
            GLOBAL_PROTOCOL:    PROTOCOL_TYPE := PIBUS32;
            INST:               STRING := "_CIF";
            LOCALID:            STRING := "?*";
            QUEUE_DEPTH:        INTEGER := 2;
            RX_LATENCY_INFO:    STRING := "CONSTANT 100";
            RX_PRIORITY_INFO:   STRING := "CONSTANT 0";
            THROUGHPUT_INFO:    STRING := "CONSTANT 1000";
            TOKEN_PROTOCOL:     PROTOCOL_TYPE := HANDSHAKE;
            TRACE:              BOOLEAN := TRUE;
            TX_LATENCY_INFO:    STRING := "CONSTANT 100";
            TX_PRIORITY_INFO:   STRING := "CONSTANT 0";
            UNIT:               DATA_SIZE := 64 KBYTE );
  PORT ( Local:  INOUT  token;
         Global: INOUT  token );
END comm_intH;
```

Important details peculiar to the Bus Interface Units are explained as follows:

PORT Local INOUT token

This is the local (non shared) port of the communication interface element.

PORT Global INOUT token

This is the global (shared) port of the communications interface.

GENERICS :

GEN ACK_TIME_INFO STRING

This is a GDL string representing the length of time it takes the intended receiver of a message to acknowledge a request for a connection. Units are of BASE_TIME.

GEN BUS_TIMEOUT_INFO: STRING

This is a GDL string representing the length of time a communications element will wait for an acknowledge of a request for a connection.

GEN GLOBALID: STRING

This is address of the global port of this communication element, which is normally the name of the device connected to the BIU's Local port.

GEN LOCALID: STRING

This is address of the local port of this communication element.

GEN GLOBAL_PROTOCOL: PROTOCOL_TYPE

This specifies the communications protocol for the global or shared side of this communication element.

GEN RX_LATENCY_INFO: STRING

This is a GDL string representing the length of time it will take a message incident upon a receiving communication element to propagate through the receiver.

GEN RX_PRIORITY_INFO: STRING

This is a GDL string specifying the priority which will be placed on the tokens which are received from the communications medium and placed on the local signal.

GEN THROUGHPUT_INFO: STRING

This is a GDL string representing the rate at which tokens of size UNIT are processed through this instance. THROUGHPUT is expressed in Hertz.

GEN TOKEN_PROTOCOL : PROTOCOL_TYPE

This is the protocol of the output token.

GEN TX_LATENCY_INFO: STRING

This is a GDL string representing the length of time it will take a message incident upon a transmitting communication element to propagate through the transmitter to appear on the global communications media.

GEN TX_PRIORITY_INFO: STRING

This is a GDL string representing the priority of this particular communication instance.

Crossbar

```
LIBRARY GEN;
use GEN.base_types.all;

ENTITY Crossbar IS

  GENERIC (
            INST: STRING := "_CBar";
            NUM_PORTS : INTEGER := 6;
            ROUTING_TABLE : TABLE_TYP; --:= (others => (others => NUL) );
            PROC_TIME_INFO: STRING := "CONSTANT 100";
            TIMEOUT_INFO: STRING := "CONSTANT 100";
            PRIORITY_INFO: STRING := "CONSTANT 0";
            QUEUE_DEPTH : POSITIVE := 2;
            PROTOCOL: PROTOCOL_TYPE := handshake;
            TX_LATENCY_INFO: STRING := "CONSTANT 100";
            THROUGHPUT_INFO: STRING := "CONSTANT 100";
            TRACE: BOOLEAN := TRUE;
            RX_LATENCY_INFO: STRING := "CONSTANT 100";
            UNIT: DATA_SIZE := 64 Kbyte
           );
  PORT ( ExtPorts : INOUT token_vector (1 to NUM_PORTS) );

END Crossbar;
```

Port declarations and generics peculiar to the crossbar are explained below.

PORT ExtPort: INOUT token_vector(1 to NUMPORTS)

These are the external ports of the crossbar onto which the external devices can be attached.

GEN NUM_PORTS: INTEGER

This generic is used to configure the number of ports on the crossbar.

GEN ROUTING_TABLE: TABLE_TYP

This is the routing table which has the names of the devices connected to the external ports. It is an array of strings, the strings contain the names of the devices.

Crossblock

```
LIBRARY GEN;
USE GEN.base_types.all;

ENTITY CrossBlock IS
  GENERIC (
             INST: STRING := "_CB1";
             PORT_ID : POSITIVE := 1;
             NUM_PORTS : INTEGER := 6;
             ROUTING_TABLE : TABLE_TYP;
             PROC_TIME_INFO: STRING := "CONSTANT 5";
             TIMEOUT_INFO: STRING := "CONSTANT 100";
             PRIORITY_INFO: STRING := "CONSTANT 0";
             QUEUE_DEPTH : POSITIVE := 1;
             PROTOCOL: PROTOCOL_TYPE := handshake;
             TX_LATENCY_INFO: STRING := "CONSTANT 10";
             THROUGHPUT_INFO: STRING := "CONSTANT 6000";
             TRACE: BOOLEAN := FALSE;
             RX_LATENCY_INFO: STRING := "CONSTANT 10";
             UNIT: DATA_SIZE := 64 Kbyte
            );
  PORT (
         ExtPort   : INOUT Token;
         IntSig    : INOUT token_vector(1 to NUM_PORTS)
        );

END CrossBlock;
```

Generics and port declarations peculiar to the crossblock are explained below:

GEN PORT_ID: POSITIVE

Each crossblock controls traffic to and from a particular external port, and the port number of that port is given by the PORT_ID generic.

PORT ExtPort: INOUT Token

This is the single external port which is component mapped onto one of the external ports of the crossbar device.

PORT IntSig: INOUT token_vector(1 to NUM_PORTS)

These are the internal bi-directional ports for routing the tokens internally.

Biu_Four Component

The architecture of the biu_four component is shown below.

ARCHITECTURE Structural of BIU_FOUR is

```
COMPONENT comm_int
GENERIC ( ACK_TIME_INFO:    STRING := "CONSTANT 100";
      BUS_TIMEOUT_INFO: STRING := "CONSTANT 100";
      GLOBALID:       STRING := "?*";
      GLOBAL_PROTOCOL:  PROTOCOL_TYPE := PIBUS32;
      INST:           STRING := "_CIF";
      LOCALID:        STRING := "?*";
      QUEUE_DEPTH:     INTEGER := 2;
      RX_LATENCY_INFO:  STRING := "CONSTANT 100";
      RX_PRIORITY_INFO: STRING := "CONSTANT 0";
      THROUGHPUT_INFO:  STRING := "CONSTANT 100";
      TOKEN_PROTOCOL:   PROTOCOL_TYPE := HANDSHAKE;
      TRACE:          BOOLEAN := FALSE;
      TX_LATENCY_INFO:  STRING := "CONSTANT 100";
      TX_PRIORITY_INFO: STRING := "CONSTANT 0";
      UNIT:           DATA_SIZE := 64 KBYTE );
PORT ( Local:  INOUT  token;
    Global: INOUT  token );
END COMPONENT;

COMPONENT comm_intH
GENERIC ( ACK_TIME_INFO:    STRING := "CONSTANT 100";
      BUS_TIMEOUT_INFO: STRING := "CONSTANT 100";
      GLOBALID:       STRING := "?*";
      SCREENID:       STRING := "XYZ";
      GLOBAL_PROTOCOL:  PROTOCOL_TYPE := PIBUS32;
      INST:           STRING := "_CIF";
      LOCALID:        STRING := "?*";
      QUEUE_DEPTH:     INTEGER := 2;
```

```
        RX_LATENCY_INFO:  STRING := "CONSTANT 100";
        RX_PRIORITY_INFO: STRING := "CONSTANT 0";
        THROUGHPUT_INFO:  STRING := "CONSTANT 100";
        TOKEN_PROTOCOL:   PROTOCOL_TYPE := HANDSHAKE;
        TRACE:          BOOLEAN := FALSE;
        TX_LATENCY_INFO:  STRING := "CONSTANT 100";
        TX_PRIORITY_INFO: STRING := "CONSTANT 0";
        UNIT:           DATA_SIZE := 64 KBYTE );
 PORT ( Local:  INOUT  token;
      Global: INOUT  token );
 END COMPONENT;

  FOR ALL: comm_intH USE ENTITY gen.comm_intH(system);
  FOR ALL: comm_int USE ENTITY gen.comm_int(system);

SIGNAL interconnect :token;
BEGIN

 BIU_TOP: comm_int
  GENERIC MAP(  ACK_TIME_INFO    => ACK_TIME_INFO,
           BUS_TIMEOUT_INFO  => BUS_TIMEOUT_INFO,
           GLOBALID        => TOP_GLOBALID,
           GLOBAL_PROTOCOL   => GLOBAL_PROTOCOL,
           INST            => inst&"_TOP_BIU",
           LOCALID         => TOP_LOCALID,
           QUEUE_DEPTH     => QUEUE_DEPTH,
           RX_LATENCY_INFO   => RX_LATENCY_INFO,
           RX_PRIORITY_INFO  => RX_PRIORITY_INFO,
           THROUGHPUT_INFO   => THROUGHPUT_INFO,
           TOKEN_PROTOCOL    => TOKEN_PROTOCOL,
           TRACE           => TRACE,
           TX_LATENCY_INFO   => TX_LATENCY_INFO,
           TX_PRIORITY_INFO  => TX_PRIORITY_INFO,
           UNIT            => UNIT )
   PORT MAP( Local => TOP_Local,
         Global=> interconnect);

 BIU_DOWN: comm_int
  GENERIC MAP(  ACK_TIME_INFO    => ACK_TIME_INFO,
           BUS_TIMEOUT_INFO  => BUS_TIMEOUT_INFO,
           GLOBALID        => BOTTOM_GLOBALID,
              -- SCREENID        => SCREENID,
```

```
        GLOBAL_PROTOCOL  => GLOBAL_PROTOCOL,
        INST             => inst&"_DOWN_BIU",
        LOCALID          => BOTTOM_LOCALID,
        QUEUE_DEPTH      => QUEUE_DEPTH,
        RX_LATENCY_INFO  => RX_LATENCY_INFO,
        RX_PRIORITY_INFO => RX_PRIORITY_INFO,
        THROUGHPUT_INFO  => THROUGHPUT_INFO,
        TOKEN_PROTOCOL   => TOKEN_PROTOCOL,
        TRACE            => TRACE,
        TX_LATENCY_INFO  => TX_LATENCY_INFO,
        TX_PRIORITY_INFO => TX_PRIORITY_INFO,
        UNIT             => UNIT )
  PORT MAP( Local => BOTTOM_Local,
       Global=> interconnect);


 BIU_LEFT: comm_int
  GENERIC MAP( ACK_TIME_INFO    => ACK_TIME_INFO,
        BUS_TIMEOUT_INFO => BUS_TIMEOUT_INFO,
        GLOBALID         => LEFT_GLOBALID,
    --  SCREENID         => SCREENID,
        GLOBAL_PROTOCOL  => GLOBAL_PROTOCOL,
        INST             => inst&"_LEFT_BIU",
        LOCALID          => LEFT_LOCALID,
        QUEUE_DEPTH      => QUEUE_DEPTH,
        RX_LATENCY_INFO  => RX_LATENCY_INFO,
        RX_PRIORITY_INFO => RX_PRIORITY_INFO,
        THROUGHPUT_INFO  => THROUGHPUT_INFO,
        TOKEN_PROTOCOL   => TOKEN_PROTOCOL,
        TRACE            => TRACE,
        TX_LATENCY_INFO  => TX_LATENCY_INFO,
        TX_PRIORITY_INFO => TX_PRIORITY_INFO,
        UNIT             => UNIT )
  PORT MAP( Local => LEFT_Local,
       Global=> interconnect);

 BIU_RIGHT: comm_inth
GENERIC MAP(    ACK_TIME_INFO    => ACK_TIME_INFO,
        BUS_TIMEOUT_INFO => BUS_TIMEOUT_INFO,
        GLOBALID         => RIGHT_GLOBALID,
        SCREENID         => SCREENID,
        GLOBAL_PROTOCOL  => GLOBAL_PROTOCOL,
```

```
        INST            => inst&"_RIGHT_BIU",
        LOCALID         => RIGHT_LOCALID,
        QUEUE_DEPTH     => QUEUE_DEPTH,
        RX_LATENCY_INFO  => RX_LATENCY_INFO,
        RX_PRIORITY_INFO => RX_PRIORITY_INFO,
        THROUGHPUT_INFO  => THROUGHPUT_INFO,
        TOKEN_PROTOCOL   => TOKEN_PROTOCOL,
        TRACE           => TRACE,
        TX_LATENCY_INFO  => TX_LATENCY_INFO,
        TX_PRIORITY_INFO => TX_PRIORITY_INFO,
        UNIT            => UNIT )

    PORT MAP( Local => RIGHT_Local,
        Global=> interconnect);


END Structural;
```

As seen from the architecture description above, the instantiation BIU_RIGHT is the BIU connected to one of the ports of the crossbar module and hence is different from the other three instantiations. The entity declaration for this component is shown below.

```
ENTITY BIU_FOUR  IS
 GENERIC ( ACK_TIME_INFO:      STRING := "CONSTANT 100";
        BUS_TIMEOUT_INFO:      STRING := "CONSTANT 100";
        TOP_GLOBALID:     STRING := "?*";
        RIGHT_GLOBALID:          STRING := "?*";
        LEFT_GLOBALID:   STRING := "?*";
          BOTTOM_GLOBALID:     STRING := "?*";
        SCREENID:             STRING := "XYZ";
        GLOBAL_PROTOCOL:      PROTOCOL_TYPE := PIBUS32;
        INST:             STRING := "_CIF";
        TOP_LOCALID:      STRING := "?*";
        LEFT_LOCALID:      STRING := "?*";
        RIGHT_LOCALID:   STRING := "?*";
          BOTTOM_LOCALID :     STRING := "?*";
        QUEUE_DEPTH:      INTEGER := 2;
        RX_LATENCY_INFO:        STRING := "CONSTANT 100";
        RX_PRIORITY_INFO:       STRING := "CONSTANT 0";
```

```
        THROUGHPUT_INFO:        STRING := "CONSTANT 100";
        TOKEN_PROTOCOL:         PROTOCOL_TYPE := HANDSHAKE;
        TRACE:          BOOLEAN := FALSE;
        TX_LATENCY_INFO:        STRING := "CONSTANT 100";
        TX_PRIORITY_INFO:       STRING := "CONSTANT 0";
        UNIT:           DATA_SIZE := 64 KBYTE );

 PORT ( LEFT_Local    : INOUT  token;
      TOP_Local     : INOUT  token;
      RIGHT_Local : INOUT token;
      Bottom_Local   : INOUT  token
     );

END BIU_FOUR;
```

The port declarations and the generics peculiar to this component are described below.

Ports LEFT_LOCAL, TOP_LOCAL and BOTTOM_LOCAL are the left, top and bottom ports of the biu_four component connected to any of the external devices except a crossbar.

Port RIGHT_LOCAL is the port connected to one of the ports of a crossbar.

TOP_GLOBALID, LEFT_GLOBALID, BOTTOM_GLOBALID are strings used to specify the names of the devices connected to the corresponding ports.

RIGHT_GLOBALID is a string which is usually set to "?*" since the right port is connected to the crossbar.

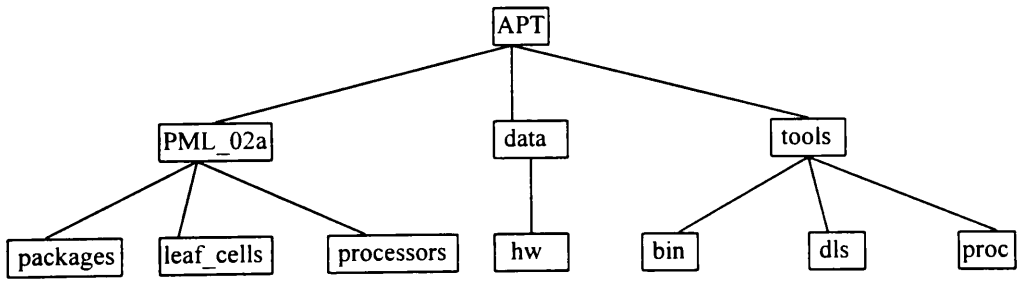
TOP_LOCALID, LEFT_LOCALID, BOTTOM_LOCALID, RIGHT_LOCALID are strings. This *LOCALID* screens traffic from the local bus onto the global bus.

APPENDIX B

User Manual for Extraction Tools

Before using these tools, install VHDL Tool Integration Platform (VTIP) from CAD Language Systems, Inc. Also, install the Performance Model Library (PML) from Honeywell (marketed by Omniview, Inc.).
The project directory should be organized into the following subdirectories.

APT/PML_02a/packages	Copy of package files from GEN library
APT/PML_02a/leaf_cells	Copy of leafcells from GEN library
APT/PML_02a/processors	Copy of processor files from PROC library
$aptsys/bin	All executable extraction tool files and shell scripts should be in this directory. The path to this directory must be specified by the user in the startup script file.
$aptsys/dls	VTIP generates this library
$aptsys/proc	This directory will contain the processor characteristic files generated by the PCET.
$aptdata/hw	This directory will contain the spreadsheet building script files generated by the ACET and CONET.



The environment variables $aptsys and $aptdata need to be set by the user to point to the following directories.

$aptsys	APT/tools
$aptdata	APT/data

Once that is accomplished, set the following variables.

setenv dls_design_library $aptsys/dls
setenv WORK dls_design_library

setenv GEN dls_design_library
setenv PROC dls_design_library

The first command assigns the physical directory of the dls to the environment variable dls_design_library. The other three commands map the virtual libraries WORK, GEN and PROC respectively, to the physical library.

The PML code for 3 files is not analyzable by VTIP. Changed files have been included in the PML_02a directory in order to make the library analyzable by VTIP. For ACET the memory modules must have a CAPACITY generic. Modified memory modules have been provided and analyzed into the dls_design_library. The directory $aptsys/dls contains a library of modified PML components already analyzed by VTIP and readily available. This library can be utilized by the user for the ACET and the CONET tools. Script files named **vtipc** have been included in each PML_02a subdirectory to be used to regenerate the $aptsys/dls library if necessary.

Processor Characterization File
Use the processor characteristic extraction tool (PCET) to generate a characteristic file for each processor of interest. The shell script which does this is called **get_proc.** This shell script in turn calls the PCET extraction tool. The command is

get_proc *argument1*

where *argument1* is the name of the processor entity that needs to be analyzed (e.g.: proccore_pentium). The entity names are currently all located in the PML file "proccore-c.vhdl" in the PROC library (subdirectory "processor"). This might change in future versions of PML.

This command invokes PCET, which generates a processor characteristic file with filename equal to the processor entity name and with a c20 extension (e.g.: pentium.c20). The shell script **get_proc** automatically directs this file into the $aptsys/proc directory. This file contains all of the parameters needed by the sizing spreadsheet. An example of a processor characteristic file follows:

```
! i960mx.c20

#LET([*,clock] =  25)
#LET([*,fladd] =  3)
#LET([*,fldiv] =  14)
#LET([*,flmlt] =  3)
#LET([*,multadd] =  3)
#LET([*,cmult] =  12)
```

```
#LET([*,btfy] = 18)
#LET([*,flsin] = 150)
#LET([*,flop] = 3)
#LET([*,iadd] = 1)
#LET([*,idiv] = 12)
#LET([*,imlt] = 4)
#LET([*,isqr] = 4)
#LET([*,iop] = 1)
#LET([*,read] = 1)
#LET([*,write] = 1)
>[*,1]#/ "i960mx#/
```

The processor characteristic file contains information about instruction execution times in the processor of interest. Only basic instructions needed in DSP applications are included.

Spreadsheet building script file

The ACET tool extracts information from the VHDL structural model that is needed to create the spreadsheet analysis tool. It generates an executable script file that automatically creates a 2020 spreadsheet with the required number of columns and rows. The connectivity information for the structural model developed, along with parameter information for the connectivity devices like throughput_info, protocol type etc need to be extracted automatically using the Connectivity Extraction Tool (CONET). Both these tools are invoked by running the shell script **get-vhdl**. This shell script needs to be invoked from the directory which contains the structural model. The shell script get-vhdl does the following things.

1. It analyzes the VHDL structural model, and stores the information in the $aptsys/dls directory.
2. It invokes ACET and directs the output to the $aptdata/hw directory.
3. It changes permission on the ACET output file and makes it executable.
4. It invokes CONET and directs the output to the $aptdata/hw directory.

The command for **get-vhdl** is

get-vhdl *argument1 argument2 argument3*

where *argument1* is the name of the entity of the structural model that needs to be analyzed, *argument2* is the name of the architecture of the model (structural, in this case), and *argument3* is the name of the VHDL file that contains the structural model to be analyzed. The shell script analyzes the VHDL file *argument3*.vhdl using VTIP. Then, the

shell script invokes ACET, and directs the output to the $aptdata/hw directory with the file name *argument3*.ssh. Then it makes the file *argument3*.ssh executable. After that is done, the shell script invokes CONET and directs the output to the directory $aptdata/hw with the name of the file being *argument3*.ssc.

Part of an example ACET output file follows.

```
#! /bin/csh -f
 hw_build    store   _b1_mem   0.025000   0.025000   4   46080.000000     process
_b1_cpu1  21062   store  _b2_mem  0.025000  0.025000  4  46080.000000   process
_b2_cpu2  21062   store  _b3_mem  0.025000  0.025000  4  46080.000000   process
_b3_cpu3  21062   store  _b4_mem  0.025000  0.025000  4  46080.....
```

Part of an example CONET output file follows. The values of the generics bus_timeout_info, tx_latency_info, rx_latency_info and ack_time_info are in micro seconds. The unit of throughput is words/second. The priority info refers to actual priority and not relative priority.

```
cbar1

    timeout_info    CONSTANT 100
    priority_info    CONSTANT 0
    queue_depth    10
    protocol    handshake
    tx_latency_info    CONSTANT 0
    throughput_info    CONSTANT 142000000
    rx_latency_info    CONSTANT 0
    unit    4    byte


        intport1        biu_star1

        intport2        biu_star2

        intport3        biu_star3

        intport4        biu_star4

        intport5        cbar5

        intport6        cbar6
```

```
biu_star1

   ack_time_info    CONSTANT 100
   bus_timeout_info    CONSTANT 100
   queue_depth    9
   rx_latency_info    CONSTANT 0
   rx_priority_info    CONSTANT 0
   throughput_info    CONSTANT 142000000
   token_protocol    handshake
   tx_latency_info    CONSTANT 0
   tx_priority_info    CONSTANT 0
   unit    4    byte


        s11      b1_mem

        intport1          cbar1

        s21      b1_cpu1
```

Trouble shooting

The following problems may arise causing a failure in the generation of PCET, ACET or CONET files.

1. There may be a sudden truncation of the ACET output file, before end of file is reached. This is mostly due to the absence of the CAPACITY generic in the memory module. Also, an improper naming of the transfer devices will result in the truncation of the CONET output file.

2. The $aptsys/dls library may get outdated. This can be fixed by running the shell script **vtipc** in sequence in the packages, leaf_cells and processors directories. This shell script analyzes the modified PML components using VTIP in a specific order, and recreates the VTIP library in the directory $aptsys/dls.

APPENDIX C

Results of CONET Extraction for the architecture in Figure 3.3

The values of the parameters ack_time_info, bus_timout_info, tx_latency_info and rx_latency_info are in micro seconds. The priority info refers to actual priority and not relative priority. The unit of throughput is words/second.

cbar1

```
timeout_info    CONSTANT 100
priority_info   CONSTANT 0
queue_depth     10
protocol    handshake
tx_latency_info    CONSTANT 0
throughput_info    CONSTANT 142000000
rx_latency_info    CONSTANT 0
unit    4    byte

        intport1        biu_star1

        intport2        biu_star2

        intport3        biu_star3

        intport4        biu_star4

        intport5        cbar5

        intport6        cbar6
```

biu_star1

```
ack_time_info    CONSTANT 100
bus_timeout_info    CONSTANT 100
```

```
  queue_depth    9
  rx_latency_info    CONSTANT 0
  rx_priority_info    CONSTANT 0
  throughput_info    CONSTANT 142000000
  token_protocol    handshake
  tx_latency_info    CONSTANT 0
  tx_priority_info    CONSTANT 0
  unit    4    byte


      s11      b1_mem

      intport1          cbar1

      s21      b1_cpu1


biu_star2

  ack_time_info    CONSTANT 100
  bus_timeout_info    CONSTANT 100
  queue_depth    9
  rx_latency_info    CONSTANT 0
  rx_priority_info    CONSTANT 0
  throughput_info    CONSTANT 142000000
  token_protocol    handshake
  tx_latency_info    CONSTANT 0
  tx_priority_info    CONSTANT 0
  unit    4    byte


      s12.    b2_mem

      intport2          cbar1

      s22      b2_cpu2


biu_star3

  ack_time_info    CONSTANT 100
  bus_timeout_info    CONSTANT 100
```

```
    queue_depth    9
    rx_latency_info    CONSTANT 0
    rx_priority_info    CONSTANT 0
    throughput_info    CONSTANT 142000000
    token_protocol    handshake
    tx_latency_info    CONSTANT 0
    tx_priority_info    CONSTANT 0
    unit    4    byte


        s13      b3_mem

        intport3          cbar1

        s23      b3_cpu3


biu_star4

    ack_time_info    CONSTANT 100
    bus_timeout_info    CONSTANT 100
    queue_depth    9
    rx_latency_info    CONSTANT 0
    rx_priority_info    CONSTANT 0
    throughput_info    CONSTANT 142000000
    token_protocol    handshake
    tx_latency_info    CONSTANT 0
    tx_priority_info    CONSTANT 0
    unit    4    byte


        s14      b4_mem

        intport4          cbar1

        s24      b4_cpu4


cbar2

    timeout_info    CONSTANT 100
    priority_info    CONSTANT 0
```

```
  queue_depth    10
  protocol    handshake
  tx_latency_info    CONSTANT 0
  throughput_info    CONSTANT 142000000
  rx_latency_info    CONSTANT 0
  unit    4    byte


        intport7          biu_star5

        intport8          biu_star6

        intport9          biu_star7

        intport10         biu_star8

        intport11         cbar5

        intport12         cbar6


biu_star5

  ack_time_info    CONSTANT 100
  bus_timeout_info    CONSTANT 100
  queue_depth    9
  rx_latency_info    CONSTANT 0
  rx_priority_info    CONSTANT 0
  throughput_info    CONSTANT 142000000
  token_protocol    handshake
  tx_latency_info    CONSTANT 0
  tx_priority_info    CONSTANT 0
  unit    4    byte


        s41      b5_mem

        intport7          cbar2

        s51      b5_cpu5
```

```
biu_star6

   ack_time_info    CONSTANT 100
   bus_timeout_info    CONSTANT 100
   queue_depth    9
   rx_latency_info    CONSTANT 0
   rx_priority_info    CONSTANT 0
   throughput_info    CONSTANT 142000000
   token_protocol    handshake
   tx_latency_info    CONSTANT 0
   tx_priority_info    CONSTANT 0
   unit    4    byte


        s42      b6_mem

        intport8          cbar2

        s52      b6_cpu6


biu_star7

   ack_time_info    CONSTANT 100
   bus_timeout_info    CONSTANT 100
   queue_depth    9
   rx_latency_info    CONSTANT 0
   rx_priority_info    CONSTANT 0
   throughput_info    CONSTANT 142000000
   token_protocol    handshake
   tx_latency_info    CONSTANT 0
   tx_priority_info    CONSTANT 0
   unit    4    byte


        s43      b7_mem

        intport9          cbar2

        s53      b7_cpu7
```

biu_star8

```
ack_time_info    CONSTANT 100
bus_timeout_info    CONSTANT 100
queue_depth    9
rx_latency_info    CONSTANT 0
rx_priority_info    CONSTANT 0
throughput_info    CONSTANT 142000000
token_protocol    handshake
tx_latency_info    CONSTANT 0
tx_priority_info    CONSTANT 0
unit    4    byte


        s44      b8_mem

        intport10        cbar2

        s54      b8_cpu8
```

cbar5

```
timeout_info    CONSTANT 100
priority_info    CONSTANT 0
queue_depth    10
protocol    handshake
tx_latency_info    CONSTANT 0
throughput_info    CONSTANT 142000000
rx_latency_info    CONSTANT 0
unit    4    byte


        intport25        four16

        intport5         cbar1

        intport11        cbar2

        intport17        cbar3

        intport23        cbar4
```

cbar3

timeout_info CONSTANT 100
priority_info CONSTANT 0
queue_depth 10
protocol handshake
tx_latency_info CONSTANT 0
throughput_info CONSTANT 142000000
rx_latency_info CONSTANT 0
unit 4 byte

intport13 biu_star9

intport14 biu_star10

intport16 biu_star11

intport17 cbar5

intport18 cbar6

biu_star9

ack_time_info CONSTANT 100
bus_timeout_info CONSTANT 100
queue_depth 9
rx_latency_info CONSTANT 0
rx_priority_info CONSTANT 0
throughput_info CONSTANT 142000000
token_protocol handshake
tx_latency_info CONSTANT 0
tx_priority_info CONSTANT 0
unit 4 byte

s71 b9_mem

intport13 cbar3

```
        s81     b9_cpu9


biu_star10

    ack_time_info    CONSTANT 100
    bus_timeout_info    CONSTANT 100
    queue_depth    9
    rx_latency_info    CONSTANT 0
    rx_priority_info    CONSTANT 0
    throughput_info    CONSTANT 142000000
    token_protocol    handshake
    tx_latency_info    CONSTANT 0
    tx_priority_info    CONSTANT 0
    unit    4    byte


        s72     b10_mem

        intport14       cbar3

        s82     b10_cpu10


biu_star11

    ack_time_info    CONSTANT 100
    bus_timeout_info    CONSTANT 100
    queue_depth    9
    rx_latency_info    CONSTANT 0
    rx_priority_info    CONSTANT 0
    throughput_info    CONSTANT 142000000
    token_protocol    handshake
    tx_latency_info    CONSTANT 0
    tx_priority_info    CONSTANT 0
    unit    4    byte


        s73     b11_mem

        intport16       cbar3
```

```
        s83     b11_cpu11


cbar4

   timeout_info    CONSTANT 100
   priority_info    CONSTANT 0
   queue_depth    10
   protocol    handshake
   tx_latency_info    CONSTANT 0
   throughput_info    CONSTANT 142000000
   rx_latency_info    CONSTANT 0
   unit    4    byte


        intport19       biu_star12

        intport20       biu_star13

        intport21       biu_star14

        intport22       biu_star15

        intport23       cbar5

        intport24       cbar6


biu_star12

   ack_time_info    CONSTANT 100
   bus_timeout_info    CONSTANT 100
   queue_depth    9
   rx_latency_info    CONSTANT 0
   rx_priority_info    CONSTANT 0
   throughput_info    CONSTANT 142000000
   token_protocol    handshake
   tx_latency_info    CONSTANT 0
   tx_priority_info    CONSTANT 0
   unit    4    byte
```

```
        sa1     b12_mem

        intport19       cbar4

        sb1     b12_cpu12


biu_star13

   ack_time_info    CONSTANT 100
   bus_timeout_info    CONSTANT 100
   queue_depth    9
   rx_latency_info    CONSTANT 0
   rx_priority_info    CONSTANT 0
   throughput_info    CONSTANT 142000000
   token_protocol    handshake
   tx_latency_info    CONSTANT 0
   tx_priority_info    CONSTANT 0
   unit    4    byte


        sa2     b13_mem

        intport20       cbar4

        sb2     b13_cpu13


biu_star14

   ack_time_info    CONSTANT 100
   bus_timeout_info    CONSTANT 100
   queue_depth    9
   rx_latency_info    CONSTANT 0
   rx_priority_info    CONSTANT 0
   throughput_info    CONSTANT 142000000
   token_protocol    handshake
   tx_latency_info    CONSTANT 0
   tx_priority_info    CONSTANT 0
   unit    4    byte
```

```
        sa3     b14_mem

        intport21       cbar4

        sb3     b14_cpu14


biu_star15

   ack_time_info    CONSTANT 100
   bus_timeout_info    CONSTANT 100
   queue_depth    9
   rx_latency_info    CONSTANT 0
   rx_priority_info    CONSTANT 0
   throughput_info    CONSTANT 142000000
   token_protocol    handshake
   tx_latency_info    CONSTANT 0
   tx_priority_info    CONSTANT 0
   unit    4    byte



        sa4     b15_mem

        intport22       cbar4

        sb4     b15_cpu15


cbar6


   timeout_info    CONSTANT 100
   priority_info    CONSTANT 0
   queue_depth    10
   protocol    handshake
   tx_latency_info    CONSTANT 0
   throughput_info    CONSTANT 142000000
```

```
rx_latency_info    CONSTANT 0
unit    4    byte


      intport27        four17

      intport6         cbar1

      intport12        cbar2

      intport18        cbar3

      intport24        cbar4
```

APPENDIX D

This Appendix contains proprietary information. For further information, please contact Dr. F. G. Gray of the Bradley Department of Electrical Engineering at (540) 231-7059.

Appendix E

PCET

The C code for PCET follows.

```
#define MAIN
#include "SPI.h"
#include <stdio.h>
#include <malloc.h>

void NumberSignals(LibName, EntityName,ArchName)


    StringN   LibName;    /* the name of the library                    */
    StringN   EntityName; /* the name of the entity                     */
    StringN   ArchName;   /* the name of the architecture (possibly null) */



{ int    I,J,K,Clock,InstCost,InstCost1,InstCost2,ctr,z,len;
  float  CS;
  Node    Lib;
  Node    Unit;
  Node    Decl;
  Node    Generic;
  Node    Gen;
  Node    Tempr;
  Node    Component;
  Node    Decl2;
  Node    region;
  Node    region1;
  Item    SomeItem,ThisItem;
  List    Decl1,Temp,MainList,Temp1;
  char    *Name,*Name1,*Name2,*Operation,*Speed;
  FILE    *ptr1;

OpenDLS();
  SetReportLevel(Warning);
  SetAbortLevel(Error);


Lib = NewLibrarySymbol(LibName);
```

```
 OpenLibrary(Lib);


 Unit = OpenUnit(Lib, EntityName, ArchName, VHDLView, AugmentMode);

region = qRegion(qView(Unit));

 Decl = Value(LastItem(qDecls(region)));

region1 = qRegion(Decl);

I = strlen((char *)qText(qDeclName(Decl)));
Name = malloc(I);
Name1 = malloc(I-9);
Name2 = malloc(I-9);

strcpy(Name,(char *)qText(qDeclName(Decl)));
for (z=9; z <= I; z=z+1)
{
Name1[z-9] = Name[z];
}
strcpy(Name2,Name1);
strcat(Name1,".c20");
Temp = qStmts(qRegion(Value(FirstItem(qStmts(region1)))));
Component = Value(NthItem(3,Temp));
Decl1 = qData(Value(LastItem(qData(qBinding(Component)))));
Generic = Value(NthItem(11,Decl1));
Gen = Value(NthItem(2,Decl1));
Clock = qIntVal4(qQuantity(Value(LastItem(qData(Gen)))));
K = strlen((char *)qText(Value(FirstItem(qData(Gen)))));
Speed = malloc(K);
strcpy(Speed,(char *)qText(Value(FirstItem(qData(Gen)))));
CS = 1000/Clock;
ptr1 = fopen(Name1,"w");
fprintf(ptr1,"! %s \n\n",Name1);
fprintf(ptr1,"#LET([*,clock] =  %3.0f)\n",CS);

ctr = 1;

MainList = qData(Value(LastItem(qData(Generic))));
SomeItem = FirstItem(MainList);
while(!NullItem(SomeItem))
{  Decl2 = Value(SomeItem);
   if(IsA(Kind(Value(FirstItem(qData(Decl2)))),AscendingRange))
{

Temp1 = qData(Value(LastItem(qData(Decl2))));
Tempr = Value(NthItem(3,Temp1));
InstCost = qIntVal4(Value(LastItem(qData(Tempr))));
ThisItem = FirstItem(qConstraint(Value(FirstItem(qData(Decl2)))));
while(!NullItem(ThisItem))
```

```
{

J = strlen((char *)qText(Value(ThisItem)));
Operation = malloc(J);
strcpy(Operation,(char *)qText(Value(ThisItem)));
fprintf(ptr1,"#LET([*,%s] =  %d)\n",Operation,InstCost);


ThisItem = NextItem(ThisItem);
}

}
else {
J = strlen((char *)qText(Value(FirstItem(qData(Decl2)))));
Operation = malloc(J);
strcpy(Operation,(char *)qText(Value(FirstItem(qData(Decl2)))));
Temp1 = qData(Value(LastItem(qData(Decl2))));
Tempr = Value(NthItem(3,Temp1));
InstCost = qIntVal4(Value(LastItem(qData(Tempr))));
fprintf(ptr1,"#LET([*,%s] =  %d)\n",Operation,InstCost);
if((strcmp(Operation,"flmlt")) == 0)
{
fprintf(ptr1,"#LET([*,multadd] =  %d)\n",InstCost);
InstCost1 = 4*InstCost;
fprintf(ptr1,"#LET([*,cmult] =  %d)\n",InstCost1);
InstCost2 = InstCost*6;
fprintf(ptr1,"#LET([*,btfy] =  %d)\n",InstCost2);
}

}
SomeItem = NextItem(SomeItem);
ctr = ctr + 1;
 }
fprintf(ptr1,">[*,1]#/ '%s#/\n",Name2);


  fclose(ptr1);
 CloseUnit(Unit,Save);


 CloseLibrary(Lib);

 CloseDLS();
}

main(argc,argv)

int argc;
char *argv[];

{
```

```
  StringN   EntityName;
  StringN   ArchName;
  StringN   LibName;

  char      Buff1[128];
  char      Buff2[128];
  char      Buff3[128];


  LibName = "dls_design_lib";
  EntityName = argv[1];
  ArchName = "";


  NumberSignals(LibName, EntityName,ArchName);

}
```

ACET

The C code for ACET follows.

```
#define MAIN
#include "SPI.h"
#include <stdio.h>
#include <malloc.h>
#include <string.h>

  char    N1[128] = {"process"};
  char    N2[128] = {"store"};
  char    ArchName1[128];

void NumberSignals(LibName, EntityName,ArchName)


    StringN   LibName;
    StringN   EntityName;
    StringN   ArchName;
{ int     I,J,K,ctr,z,q;
  Node    Lib;
  Node    Unit;
  Node    Decl;
  Node    node1;
  Node    node2;
```

```
 Node    Component;
 Node    Decl2;
 Node    region;
 Item    SomeItem;
 List    MainList,MainList1,node3;
 char    *Name,*Name1,*Name2,*Name3,*Name4,*Name5, tempstr[8];
int     wordsize,dummy,throughput = 0;
 float   CS,CS1,capacity,capacity1,units;
 Node    Decl3;          /*a Decl. node rep. the a region  */
 Node    Gen1,Gen2,Gen3,Gen4;  /*a Decl. node rep. the a region      */
 char    *Names,*Names1,*Names4,*unit,*name;
 Item    SomeItem1;


OpenDLS();
 SetReportLevel(Warning);
 SetAbortLevel(Error);


Lib = NewLibrarySymbol(LibName);
 OpenLibrary(Lib);


 Unit = OpenUnit(Lib, EntityName, ArchName, VHDLView, AugmentMode);

region = qRegion(qView(Unit));

 Decl = Value(LastItem(qDecls(region)));

MainList = qStmts(qRegion(Decl));

SomeItem = FirstItem(MainList);
printf("#! /bin/csh -f\n");
printf(" hw_build  ");
ctr = 1;
while(!NullItem(SomeItem))
{  Decl2 = Value(SomeItem);

I = strlen((char *)qText(qStmtName(Decl2)));
Name4 = (char *)malloc(I);
strcpy(Name4,(char *)qText(qStmtName(Decl2)));
Name = (char *)malloc(9);
strcpy(Name,"_");
strncpy(tempstr,Name4,8);
strcat(Name,tempstr);
Name[9] = '\0';

J = strlen((char *)qText(qName(Decl2)));
Name1 = (char *)malloc(J);
strcpy(Name1,(char *)qText(qName(Decl2)));
```

```
if((strcmp(Name1,"processor")) == 0 || (strcmp(Name1,"indeviceh")) == 0|| (strcmp(Name1,"outdevice"))
== 0)
{
if((strcmp(Name1,"indeviceh")) == 0)
{
strcpy(Name1,"indevice");
}
if((strcmp(Name1,"processor")) == 0)
{
node1 = qDef(Value(FirstItem(qData(qBinding(Decl2)))));
node2 = qRegion(Value(FirstItem(qStmts(qRegion(node1)))));
node3 = qData(qBinding(Value(FirstItem(qStmts(node2)))));

K = strlen((char *)qText(Value(FirstItem(node3))));
Name2 = (char *)malloc(K);
Name3 = (char *)malloc(K-9);
strcpy(Name2,(char *)qText(Value(FirstItem(node3))));
for (z=9; z <= K; z=z+1)
{
Name3[z-9] = Name2[z];
}
if((strcmp(Name3,"960mx")) == 0 || (strcmp(Name3,"386")) == 0)
{
Name5 = (char *)malloc(K-8);
strcpy(Name5,"i ");
for (z=0; z <= K-7; z=z+1)
{
Name5[z+1] = Name3[z];
}
Name3 = (char *)malloc(K-8);
strcpy(Name3,Name5);
}
if((strcmp(Name3,"sharc")) == 0)
{
strcpy(Name3,"21062");
}
printf(" %s %s %s ",N1,Name,Name3);
}
else
{
printf(" %s %s %s",N1,Name,Name1);
}
}
else if(strcmp(Name1,"memoryh") == 0)
{
printf(" %s %s ",N2,Name);



MainList1 = qAssocs(Decl2);
SomeItem1 = FirstItem(MainList1);
```

```
while(!NullItem(SomeItem1))
{
if(IsA((Kind(Value(SomeItem1))),GenericAssocOp))
{
Decl3 = Value(SomeItem1);

I = strlen((char *)qText(Value(FirstItem(qData(Decl3)))));
Names4 = (char *)malloc(I);
strcpy(Names4,(char *)qText(Value(FirstItem(qData(Decl3)))));

if((strcmp(Names4,"throughput_info")) == 0)
        {
        Gen1 = Decl3;
        }
else if((strcmp(Names4,"capacity")) == 0)
        {
        Gen2 = Decl3;
        }
else if((strcmp(Names4,"unit")) == 0)
        {
        Gen3 = Decl3;
        }
}
SomeItem1 = NextItem(SomeItem1);
}
I = strlen((char *)qStrVal(Value(LastItem(qData(Gen1)))));
Names = (char *)malloc(I);
strcpy(Names,(char *)qStrVal(Value(LastItem(qData(Gen1)))));

Names1 = (char *)malloc(I-9);
for (z=9; z <= I; z=z+1)
{
Names1[z-9] = Names[z];
}
Names1[I-9] = '\0';
throughput = 0;

for (z=0; z < I-9; z=z+1)
{
switch(Names1[z])
   {
        case '0'  : dummy = 0;
                    break;
        case '1'  : dummy = 1;
                    break;
        case '2'  : dummy = 2;
                    break;
        case '3'  : dummy = 3;
                    break;
        case '4'  : dummy = 4;
```

```
                        break;
        case '5'  : dummy = 5;
                        break;
        case '6'  : dummy = 6;
                        break;
        case '7'  : dummy = 7;
                        break;
        case '8'  : dummy = 8;
                        break;
        case '9'  : dummy = 9;

   }

throughput = (throughput*10) + dummy;

}

wordsize = qIntVal4(qQuantity(Value(LastItem(qData(Gen3)))));

CS1 = 1000000./throughput;
capacity1 = qIntVal4(qQuantity(Value(LastItem(qData(Gen2)))));
I = strlen((char *)qText(qQuantum(Value(LastItem(qData(Gen2))))));
name = (char *)malloc(I);
strcpy(name,(char *)qText(qQuantum(Value(LastItem(qData(Gen2))))));

if((strcmp(name,"kbyte")) == 0)
        {
        units = 1024;
        }
else if((strcmp(name,"mbyte")) == 0)
        {
        units = 1048576;
        }
else if((strcmp(name,"byte")) == 0)
        {
        units = 1;
        }
else if((strcmp(name,"bit_size")) == 0)
        {
        units = 1./8;
        }

capacity = capacity1*units;

printf(" %f  %f  %d  %f ",CS1,CS1,wordsize,capacity);


}
ctr = ctr + 1;
SomeItem = NextItem(SomeItem);
 }
```

```
printf("\n\n\n");

  free(Name);
  free(Names);
  free(Name1);
  free(Names1);
  free(Name2);
  free(Name3);
  free(Name4);
  free(Names4);
  free(Name5);


  CloseUnit(Unit,Save);


  CloseLibrary(Lib);

  CloseDLS();
}


main(argc,argv)

int argc;
char *argv[];

{

  StringN    EntityName;
  StringN    ArchName;
  StringN    LibName;

  LibName = "dls_design_lib";
  EntityName = argv[1];
  ArchName = argv[2];


  NumberSignals(LibName, EntityName,ArchName);

}
```

CONET

The C code for CONET follows.

```
#define MAIN
#include "SPI.h"
#include <stdio.h>
#include <malloc.h>
#include <string.h>

char    N1[128] = {"process"};
char    N2[128] = {"transfer"};
char    N3[128] = {"store"};

void NumberSignals(LibName, EntityName,ArchName)


   StringN   LibName;    /* the name of the library                    */
   StringN   EntityName; /* the name of the entity                     */
   StringN   ArchName;   /* the name of the architecture (possibly null) */

{ int     I,J,K,ctr,z,x,P;
 Node     Lib;
 Node     Unit;
 Node     Decl;
 Node     Component;
 Node     Decl2,Decl3;
 Node     region;
 Item     SomeItem,MapItem,Item1,SomeItem1,Item2,Item3,Itemlast,Item4,Item5;
 List     MainList,MapList,List1,List2;
 char     *Name,*Name1,*Name2,*Names4,*Names5,*Signal,*Label,*Signal1;

 OpenDLS();
 SetReportLevel(Warning);
 SetAbortLevel(Error);


 Lib = NewLibrarySymbol(LibName);
 OpenLibrary(Lib);


 Unit = OpenUnit(Lib, EntityName, ArchName, VHDLView, AugmentMode);

 region = qRegion(qView(Unit));

 Decl = Value(LastItem(qDecls(region)));

 MainList = qStmts(qRegion(Decl));

 SomeItem = FirstItem(MainList);
 SomeItem1 = FirstItem(MainList);
 ctr = 1;
 while(!NullItem(SomeItem))
   {  Decl2 = Value(SomeItem);
```

```
    I = strlen((char *)qText(qStmtName(Decl2)));
    Name = (char *)malloc(I);
    strcpy(Name,(char *)qText(qStmtName(Decl2)));

        x = strlen("trf");
    Name2 = (char *)malloc(x);
                strncpy(Name2,Name,x);
Name2[3] = '\0';

    J = strlen((char *)qText(qName(Decl2)));
    Name1 = (char *)malloc(J);
    strcpy(Name1,(char *)qText(qName(Decl2)));

    if((strcmp(Name2,"cba")) == 0 || (strcmp(Name2,"biu")) == 0 )
        {
      printf("\n %s \n",Name);
        MapList = qAssocs(Decl2);
        MapItem = FirstItem(MapList);

        while(!NullItem(MapItem))
         {
          if(IsA(Kind(Value(MapItem)),GenericAssocOp))
                {
                        Decl3 = Value(MapItem);

                I = strlen((char *)qText(Value(FirstItem(qData(Decl3)))));
                        Names4 = (char *)malloc(I);
                strcpy(Names4,(char *)qText(Value(FirstItem(qData(Decl3)))));



                if((strcmp(Names4,"ack_time_info")) == 0)
                        {
                        I = strlen((char *)qStrVal(Value(LastItem(qData(Decl3)))));
                        Names5 = (char *)malloc(I);
                strcpy(Names5,(char *)qStrVal(Value(LastItem(qData(Decl3)))));
                        printf("\n    %s    %s ",Names4,Names5);
                        }
                else  if((strcmp(Names4,"bus_timeout_info")) == 0 || (strcmp(Names4,"timeout_info"))
== 0)
                        {
                        I = strlen((char *)qStrVal(Value(LastItem(qData(Decl3)))));
                        Names5 = (char *)malloc(I);
                strcpy(Names5,(char *)qStrVal(Value(LastItem(qData(Decl3)))));
                        printf("\n    %s    %s ",Names4,Names5);
                        }
                else if((strcmp(Names4,"queue_depth")) == 0)
                        {
                        I = qIntVal4(Value(LastItem(qData(Decl3))));
                        printf("\n    %s    %d ",Names4,I);
                        }
```

```
else if((strcmp(Names4,"rx_latency_info")) == 0)
        {
        I = strlen((char *)qStrVal(Value(LastItem(qData(Decl3)))));
        Names5 = (char *)malloc(I);
strcpy(Names5,(char *)qStrVal(Value(LastItem(qData(Decl3)))));
        printf("\n    %s    %s ",Names4,Names5);
        }
else if((strcmp(Names4,"rx_priority_info")) == 0 || (strcmp(Names4,"priority_info")) ==
0)
        {
        I = strlen((char *)qStrVal(Value(LastItem(qData(Decl3)))));
        Names5 = (char *)malloc(I);
strcpy(Names5,(char *)qStrVal(Value(LastItem(qData(Decl3)))));
        printf("\n    %s    %s ",Names4,Names5);
        }
else if((strcmp(Names4,"throughput_info")) == 0)
        {
        I = strlen((char *)qStrVal(Value(LastItem(qData(Decl3)))));
        Names5 = (char *)malloc(I);
strcpy(Names5,(char *)qStrVal(Value(LastItem(qData(Decl3)))));
        printf("\n    %s    %s ",Names4,Names5);
        }
else if((strcmp(Names4,"protocol")) == 0 || (strcmp(Names4,"token_protocol")) == 0)
        {
        I = strlen((char *)qText(Value(LastItem(qData(Decl3)))));
        Names5 = (char *)malloc(I);
strcpy(Names5,(char *)qText(Value(LastItem(qData(Decl3)))));
        printf("\n    %s    %s ",Names4,Names5);
        }
else if((strcmp(Names4,"tx_latency_info")) == 0)
        {
        I = strlen((char *)qStrVal(Value(LastItem(qData(Decl3)))));
        Names5 = (char *)malloc(I);
strcpy(Names5,(char *)qStrVal(Value(LastItem(qData(Decl3)))));
        printf("\n    %s    %s ",Names4,Names5);
        }
else if((strcmp(Names4,"tx_priority_info")) == 0)
        {
        I = strlen((char *)qStrVal(Value(LastItem(qData(Decl3)))));
        Names5 = (char *)malloc(I);
strcpy(Names5,(char *)qStrVal(Value(LastItem(qData(Decl3)))));
        printf("\n    %s    %s ",Names4,Names5);
        }
else if((strcmp(Names4,"unit")) == 0)
        {
        P = qIntVal4(qQuantity(Value(LastItem(qData(Decl3)))));
        I = strlen((char *)qText(qQuantum(Value(LastItem(qData(Decl3))))));
        Names5 = (char *)malloc(I);
strcpy(Names5,(char *)qText(qQuantum(Value(LastItem(qData(Decl3))))));
        printf("\n    %s    %d    %s \n",Names4,P,Names5);
        }
```

```
        }
    if(IsA(Kind(Value(MapItem)),PortAssocOp))
        {
         List1 = qData(Value(MapItem));
         Item1 = LastItem(List1);

         if(IsA((ObjectType)ListOp,Kind(Value(Item1))))
          {
           List2 = qData(Value(Item1));
                Item4 = FirstItem(List2);
                while(!NullItem(Item4))
                {
                 Item5 = LastItem(qData(Value(Item4)));

                if(IsA((ObjectType)Identifier,Kind(Value(Item5))))
           {
           I = strlen((char *)qText(Value(Item5)));
           Signal = (char *)malloc(I);
           strcpy(Signal,(char *)qText(Value(Item5)));

           SomeItem1 = FirstItem(MainList);
           while(!NullItem(SomeItem1))
                 {
                  I = strlen((char *)qText(qStmtName(Value(SomeItem1))));
                  Label = (char *)malloc(I);
                  strcpy(Label,(char *)qText(qStmtName(Value(SomeItem1))));
                  if(strcmp(Name,Label) != 0 )
                    {
                     Item2 = FirstItem(qAssocs(Value(SomeItem1)));

                     while(!NullItem(Item2))
                           {
                            if(IsA((ObjectType)PortAssocOp,Kind(Value(Item2))))
                              {
                               Item3 = LastItem(qData(Value(Item2)));

                               if(IsA((ObjectType)ListOp,Kind(Value(Item3))))
                                     {
                                      Itemlast = FirstItem(qData(Value(Item3)));
                                      while(!NullItem(Itemlast))
                                        {
                                                                                I     =     strlen((char
*)qText(Value(LastItem(qData(Value(Itemlast))))));
                                           Signal1 = (char *)malloc(I);
                                                                                strcpy(Signal1,(char
*)qText(Value(LastItem(qData(Value(Itemlast))))));
```

```
                                    if(strcmp(Signal,Signal1) == 0)
                                        {

                                         printf(" \n\n \t %s \t %s",Signal1,Label);

                                        }
                                    Itemlast = NextItem(Itemlast);
                                   }
                                 }
                              else if(IsA((ObjectType)Identifier,Kind(Value(Item3))))
                                    {
                                     I = strlen((char *)qText(Value(Item3)));
                                     Signal1 = (char *)malloc(I);
                                     strcpy(Signal1,(char *)qText(Value(Item3)));

                                     if(strcmp(Signal,Signal1) == 0)
                                      {

                                       printf(" \n\n \t %s \t %s",Signal1,Label);

                                      }
                                    }
                                  }
                                 Item2 = NextItem(Item2);
                                }
                              }
                             SomeItem1 = NextItem(SomeItem1);
                             }
                            }
                            Item4 = NextItem(Item4);

                           }
                          }
    else if(IsA((ObjectType)Identifier,Kind(Value(Item1))))
       {
       I = strlen((char *)qText(Value(Item1)));
       Signal = (char *)malloc(I);
       strcpy(Signal,(char *)qText(Value(Item1)));

       SomeItem1 = FirstItem(MainList);
       while(!NullItem(SomeItem1))
             {
              I = strlen((char *)qText(qStmtName(Value(SomeItem1))));
              Label = (char *)malloc(I);
              strcpy(Label,(char *)qText(qStmtName(Value(SomeItem1))));
              if(strcmp(Name,Label) != 0 )
                {
                 Item2 = FirstItem(qAssocs(Value(SomeItem1)));

                 while(!NullItem(Item2))
                       {
```

```
                                    if(IsA((ObjectType)PortAssocOp,Kind(Value(Item2))))
                                     {
                                      Item3 = LastItem(qData(Value(Item2)));

                                      if(IsA((ObjectType)ListOp,Kind(Value(Item3))))
                                            {
                                             Itemlast = FirstItem(qData(Value(Item3)));
                                             while(!NullItem(Itemlast))
                                              {.
                                                                                    I     =     strlen((char
*)qText(Value(LastItem(qData(Value(Itemlast))))));
                                                Signal1 = (char *)malloc(I);
                                                                                          strcpy(Signal1,(char
*)qText(Value(LastItem(qData(Value(Itemlast))))));

                                                if(strcmp(Signal,Signal1) == 0)
                                                      {

                                                       printf(" \n\n \t %s \t %s",Signal1,Label);

                                                      }
                                                Itemlast = NextItem(Itemlast);
                                              }
                                            }
                                      else if(IsA((ObjectType)Identifier,Kind(Value(Item3))))
                                            {
                                             I = strlen((char *)qText(Value(Item3)));
                                             Signal1 = (char *)malloc(I);
                                             strcpy(Signal1,(char *)qText(Value(Item3)));

                                             if(strcmp(Signal,Signal1) == 0)
                                              {

                                               printf(" \n\n \t %s \t %s",Signal1,Label);

                                              }
                                            }
                                     }
                                    Item2 = NextItem(Item2);
                                   }
                                 }
                                SomeItem1 = NextItem(SomeItem1);
                               }
                               }
                      }
              MapItem = NextItem(MapItem);
             }

            printf(" \n\n");
           }
    ctr = ctr + 1;
```

```
    SomeItem = NextItem(SomeItem);
   }

 free(Name);
 free(Name1);
 free(Name2);
 free(Signal);
 free(Signal1);
 free(Label);

 CloseUnit(Unit,Save);


 CloseLibrary(Lib);

 CloseDLS();
}

main(argc,argv)

int argc;
char *argv[];

{

 StringN   EntityName;
 StringN   ArchName;
 StringN   LibName;

 char     Buff1[128];
 char     Buff2[128];
 char     Buff3[128];


 LibName = "dls_design_lib";
 EntityName = argv[1];
 ArchName = argv[2];



 NumberSignals(LibName, EntityName,ArchName);

}
```

Vita

Priya Balasubramanian was born on the 15th of May, 1972, in the city of Madras, India. She passed out of R. S. Krishnan higher secondary school in Trichy. She received her Bachelor of Engineering (B.E.) in Electrical and Electronics Engineering in May 1994 from the University of Madras.

She decided to pursue her MS in the Computer Engineering Department at Virginia Polytechnic Institute and State University (Virginia Tech) beginning from the Fall '94 term.

After completing her MS CpE in the summer '96 term, she will be working with Intel Corporation, Portland, Oregon as Design Engineer.