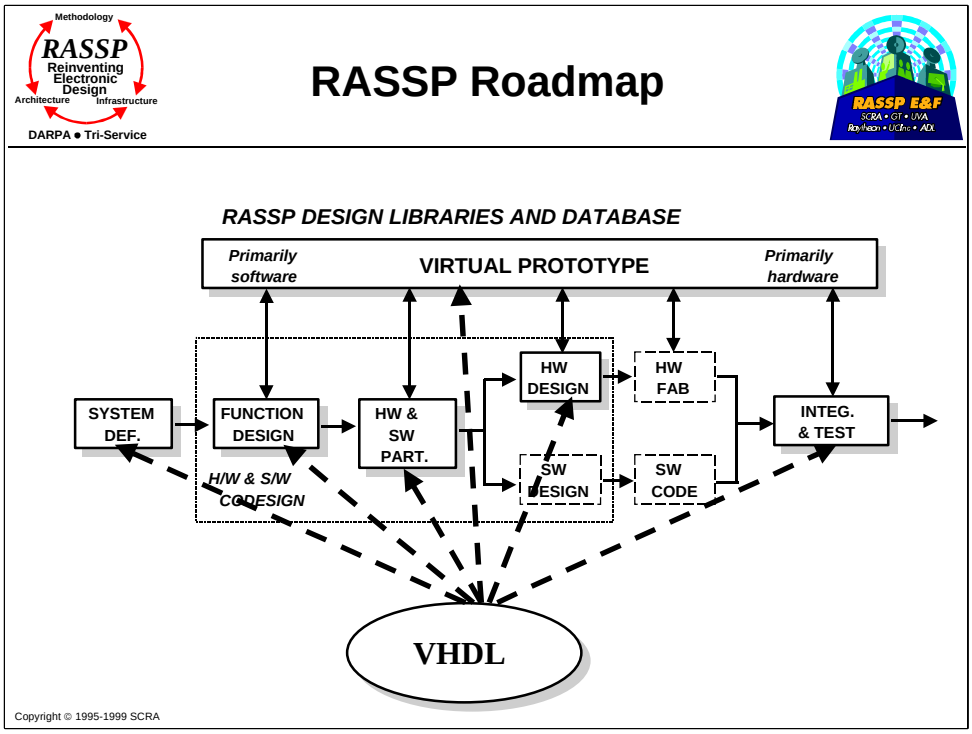
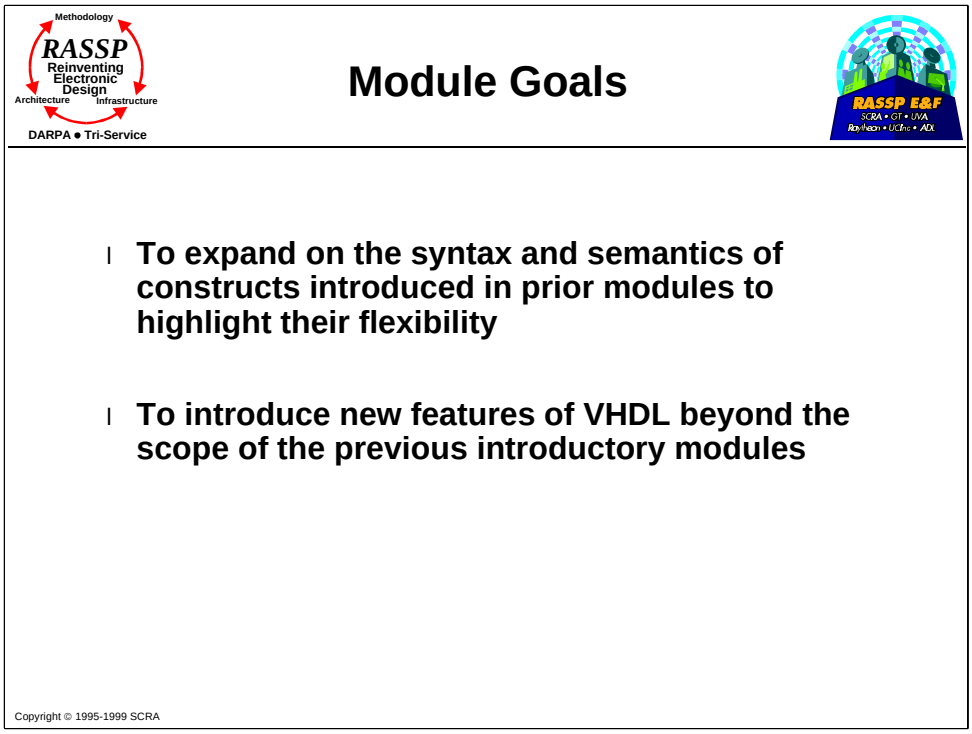
Advanced Concepts in VHDL

RASSP Education & Facilitation

Module 13

Version 3.00





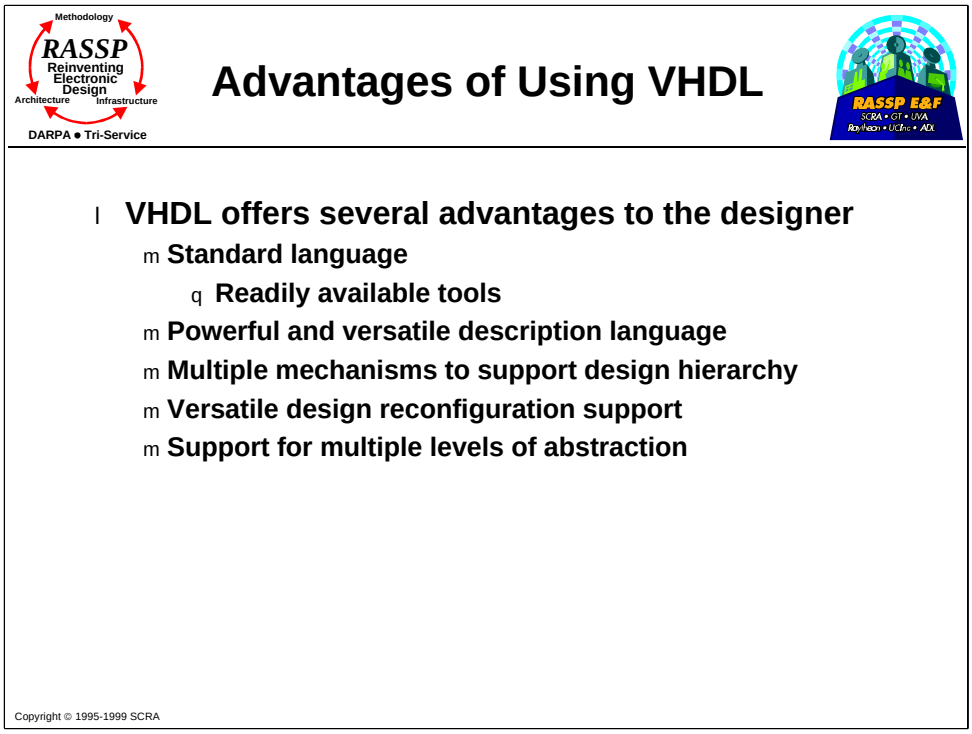
This is the fourth in the series of VHDL instructional modules prepared by the Rapid Prototyping of Application Specific Signal Processors (RASSP) Education & Facilitation team.

Outline

- **Introduction**
- **Revisiting some VHDL constructs**
  - **Aliases**
  - **Foreign interfaces**
  - **Files**
  - **Textio**
  - **Assert statements**
  - **Processes**
  - **Signal assignment statements**
  - **Shared variables**

Outline (Cont.)

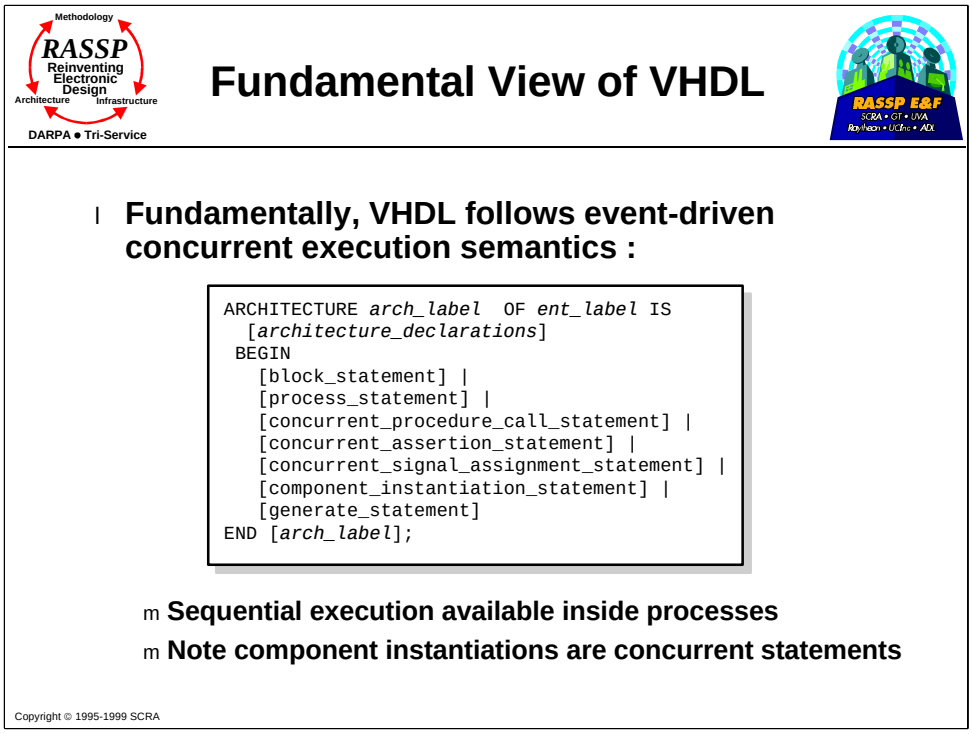
- **Examples**
  - **Abstract data type example**
  - **Example from UVA ADEPT**
- **Summary**



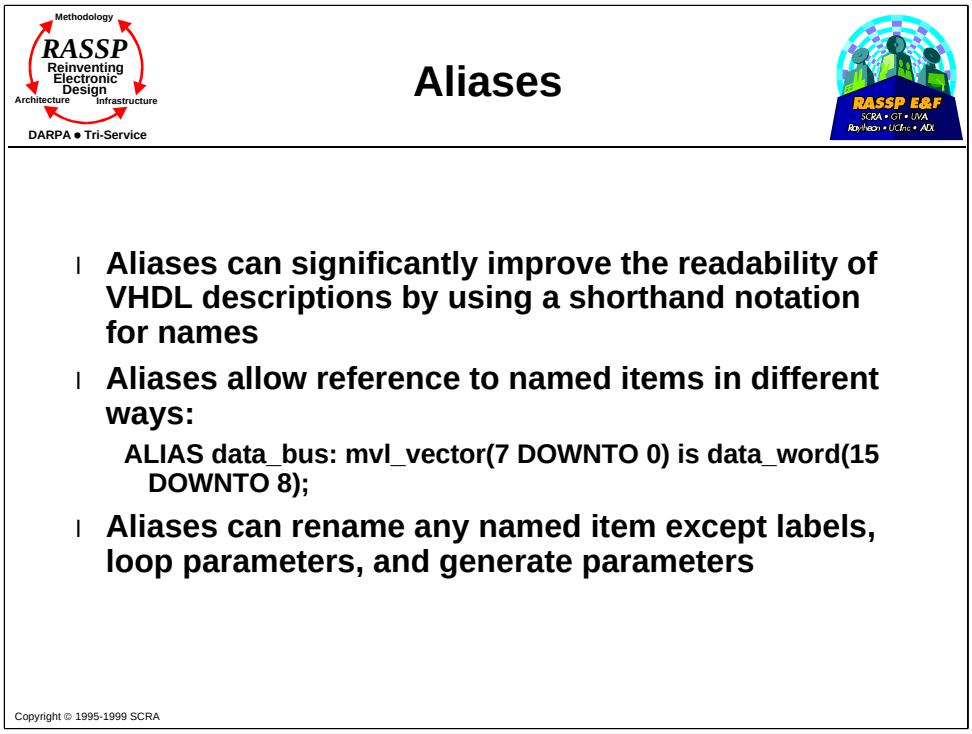
Some of the advantages in using VHDL as a description language include its versatility and the fact that it is an accepted standard with broad support from both government and industry.

Outline

- **Introduction**
- **Revisiting some VHDL constructs**
  - **Aliases**
  - **Foreign interfaces**
  - **Textio**
  - **Assert statements**
  - **Processes**
  - **Signal assignment statements**
  - **Shared variables**
- **Examples**
- **Summary**



By this time, the student should recognize that VHDL is actually a concurrent language in which consistent and predictable behavior is enforced by the underlying timing model. Sequential behavior is available within processes to facilitate the description of complex functionality that is more easily implemented with sequential statements, but each process is then itself a concurrent statement within VHDL.

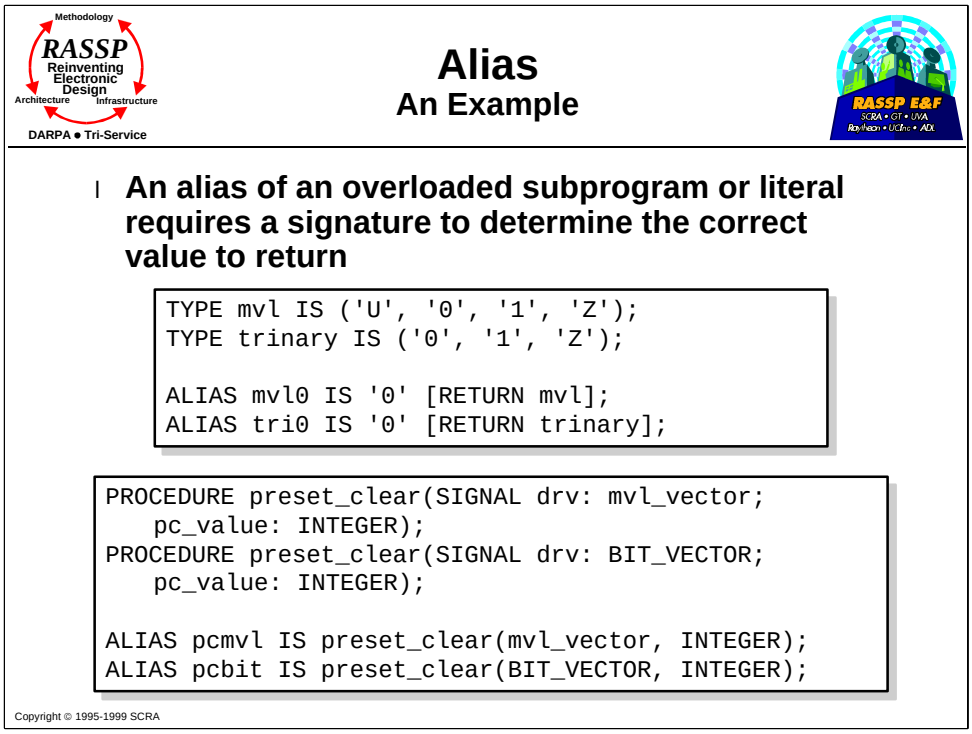


VHDL provides the alias construct to enhance readability in VHDL descriptions. Aliases are available in two varieties:

1. Object aliases rename objects
    a. constant
    b. signal
    c. variable
    d. file

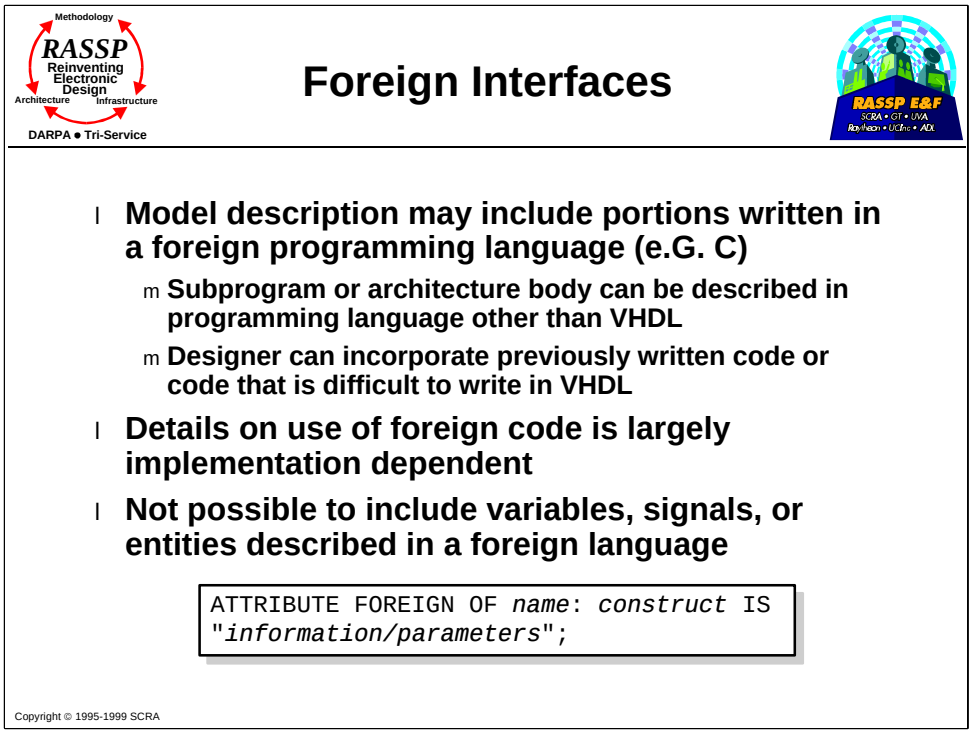
2. Non-object aliases rename items that are not objects
    a. function names
    b. literals
    c. type names
    d. attribute names

[Bhasker95]



A signature is required for an alias of a subprogram or an enumeration literal. A signature is also used to disambiguate overloaded subprograms and overloaded enumeration literals in which the signature indicates the parameter types and result type. A set of outer brackets “[“ and “]” is used to identify a signature.

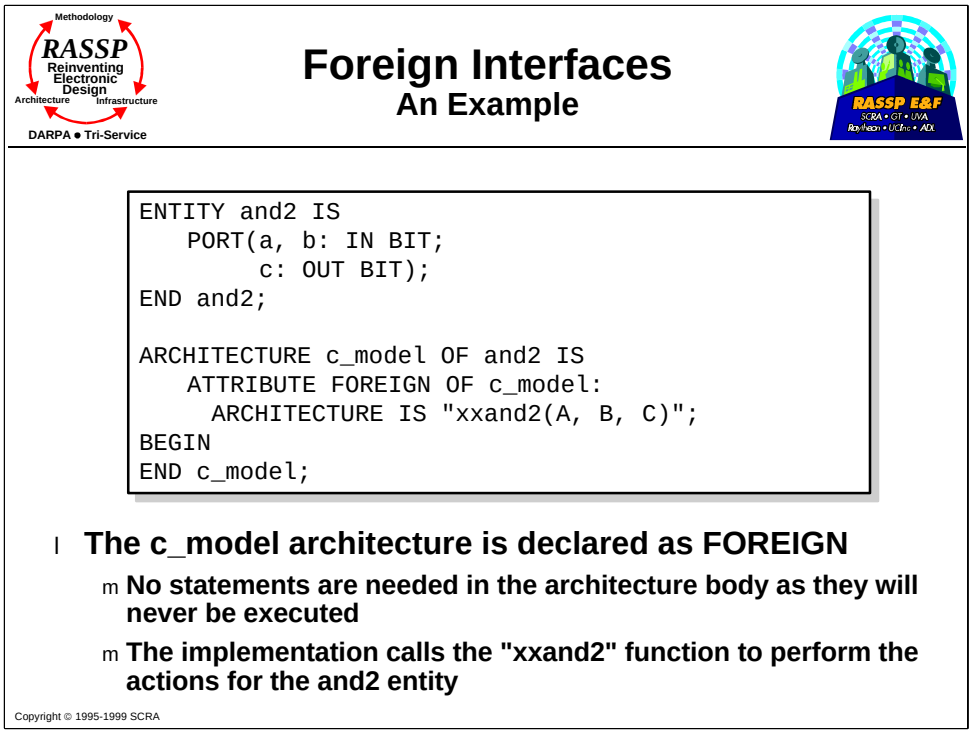
[Bhasker95]



VHDL allows the functionality of architecture bodies and subprograms to be described in a foreign language (e.g. C) and interfaced to a VHDL model. For example, foreign code may be used when it is difficult to implement the same functionality in VHDL, such as in cases where complex arithmetic functions not directly available in VHDL are required.

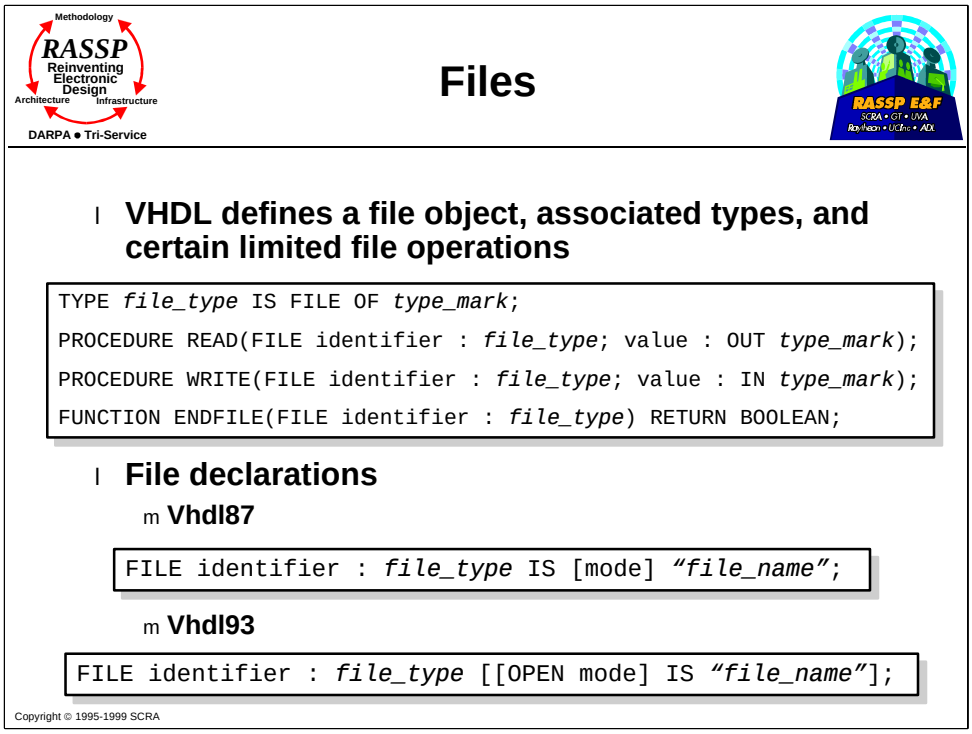
The interface between VHDL and foreign code is simulator implementation dependent. VHDL passes the parameters to the foreign code but has no further information about the foreign code. The use and structure of foreign code is largely up to the particular simulator implementation.

[Bergé93]



The c_model code for xxand2 exists in some form that is implementation dependent. This code could be in a library of other models written in C that may be similarly accessed.

[Bhasker95]



VHDL defines the file object and includes some basic file IO procedures implicitly after a file type is defined. A file type must be defined for each VHDL type that is to be input from or output to a file.

- Example:

  TYPE bit_file IS FILE of bit;

In VHDL87, there are no routines to open or close a file, so both the mode of the file and its name must be specified in the file declaration. The mode defaults to *read* if none is specified.

- Examples:

  FILE in_file:bit_file IS "my_file.dat" -- opens a file for reading

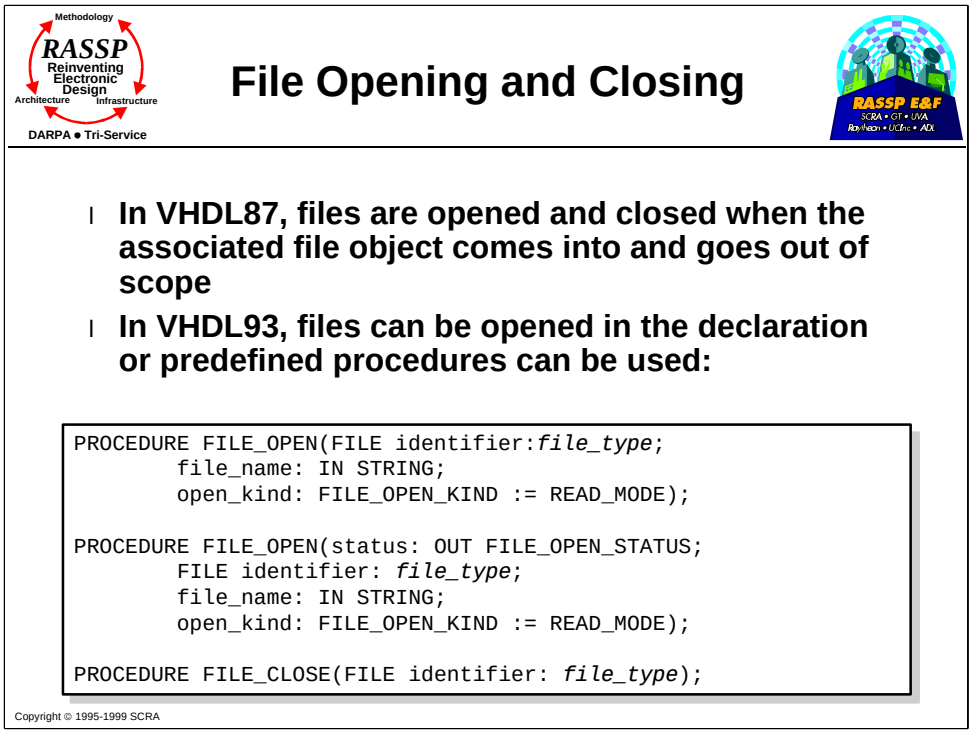
  FILE out_file:bit_file IS OUT "my_other_file.dat"; -- opens a file for writing

In VHDL93, a file can be named and opened in the declaration:

FILE in_file:bit_file OPEN READ_MODE IS "my_file.dat"; -- opens a file for reading

Or simply declared (and named and opened later):

FILE out_file:bit_file;



In VHDL87, the file is opened and closed when it come into and goes out of scope.

In VHDL93, there are two FILE_OPEN procedures, one of which returns a value of the status (success) for opening the file, and one which doesn't. There is also a FILE_CLOSE procedure.

The values for FILE_OPEN_KIND are:

READ_MODE,

WRITE_MODE, and

APPEND_MODE.

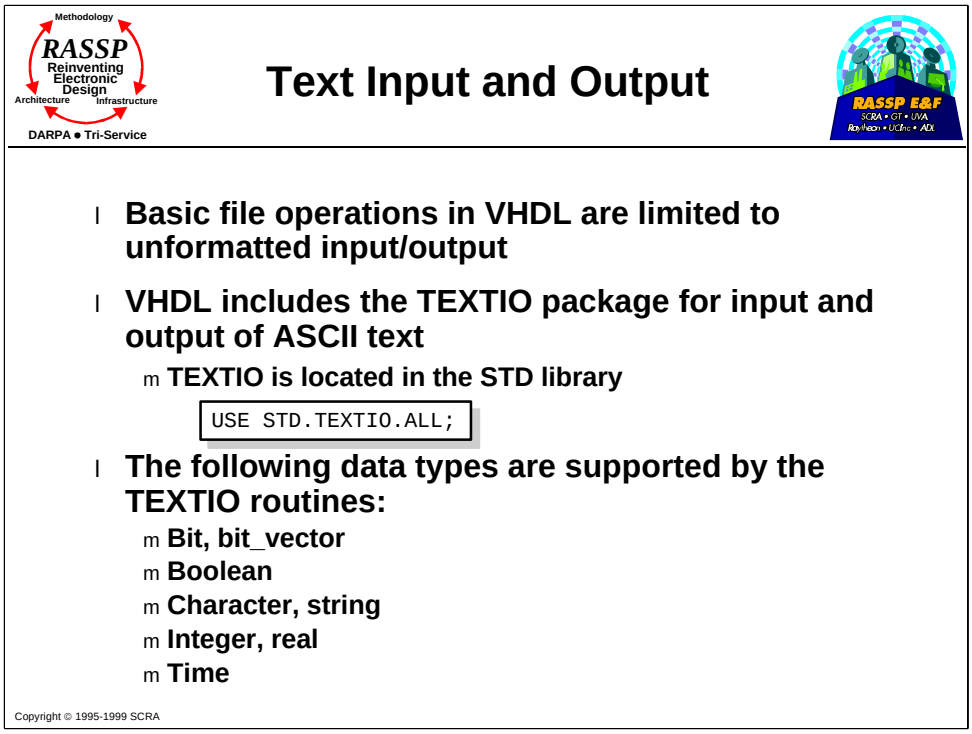
The values for FILE_OPEN_STATUS are:

OPEN_OK,

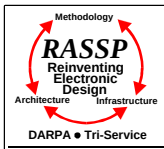
STATUS_ERROR,

NAME_ERROR, and

MODE_ERROR.



The TEXTIO package provides additional declarations and subprograms for handling text (ASCII) files in VHDL. For example, the basic READ and WRITE operations of the FILE type are not very useful because they work with binary files. Therefore, the TEXTIO package provides subprograms for manipulating text more easily and efficiently.



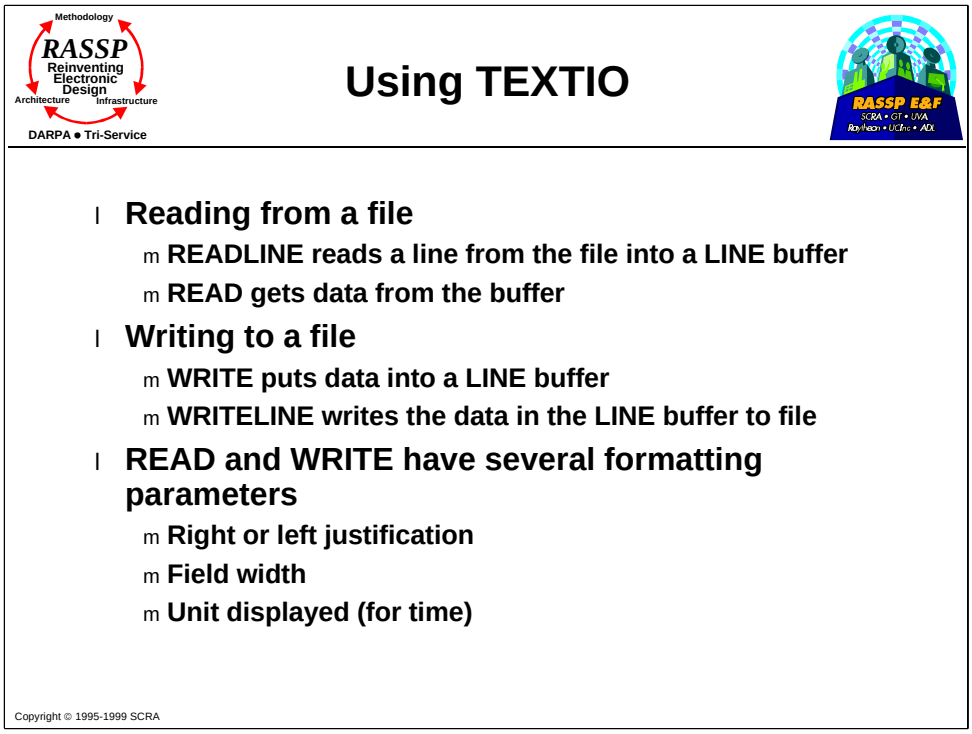
TEXTIO Procedures

- **TEXTIO defines a LINE data type**
  - **All read and write operations use the LINE type**
- **TEXTIO also defines a FILE type of TEXT for use with ASCII text**
- **Procedures defined by TEXTIO are:**
  - **Readline(f,k)**
    - **Reads a line of file f and places it in buffer k**
  - **Read(k,v,...)**
    - **Reads a value of v of its type from k**
  - **Write(k,v,...)**
    - **Writes value v to LINE k**
  - **Writeline(f,k)**
    - **Writes k to file f**
  - **Endfile(f) returns TRUE at the end of FILE**

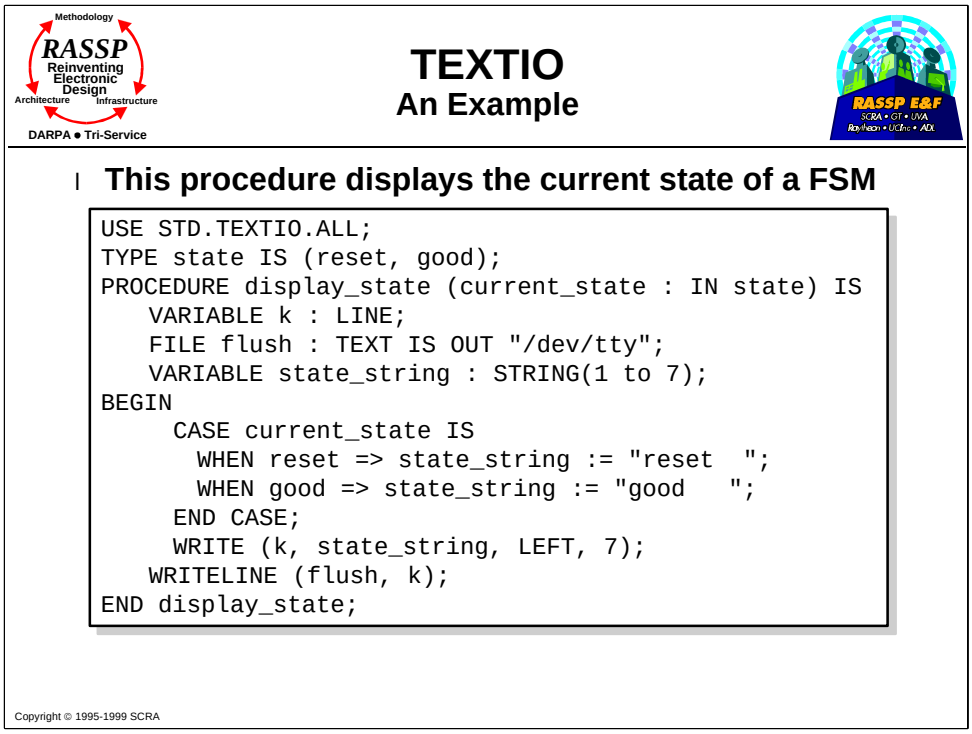
TEXTIO defines two new data types to assist in text handling. The first is the LINE data type. The LINE type is a text buffer used to interface VHDL I/O and the file. Only the LINE type may read from or written to a file.

A new FILE type of TEXT is also defined. A file of type TEXT may only contain ASCII characters.

Several of the procedures provided by TEXTIO for handling text input/output are also listed in this slide.



TEXTIO requires that all disk access go through a buffer of type LINE. In addition, the READ and WRITE procedures can further format the text. The field width of the text is the length of the text if not otherwise specified. If the text is of type TIME, the unit of time can be specified.



This example displays the current state of a finite state machine model execution. First, the USE clause makes the contents of the TEXTIO package available. The enumerated type STATE is then locally declared. The procedure display_state requires only one input value, the current state of the FSM.

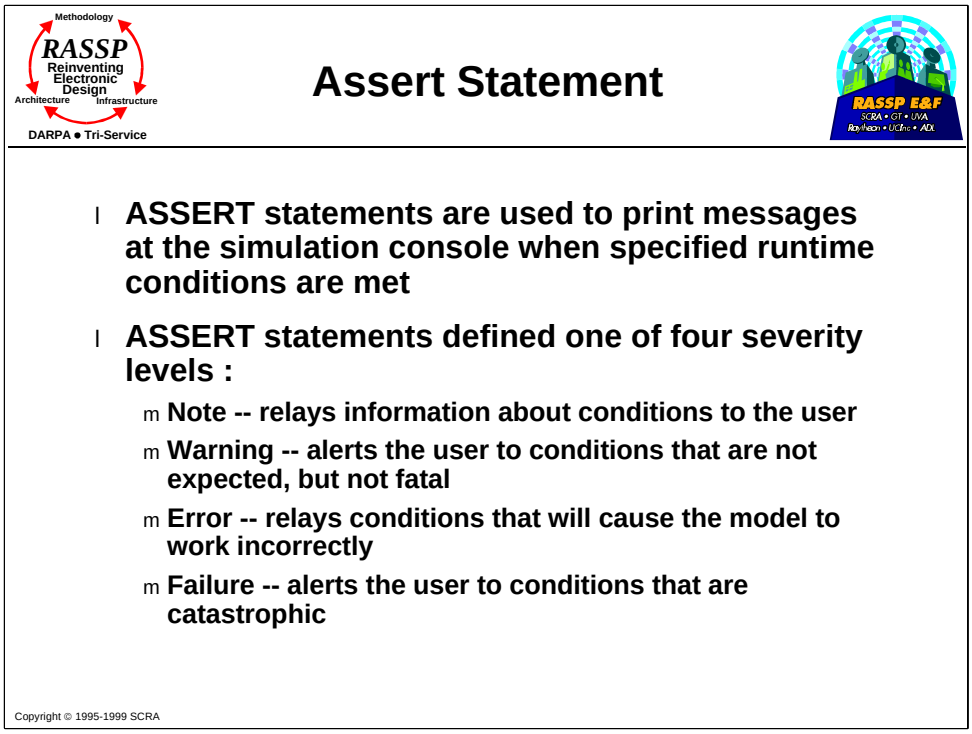
Several local variables are declared within the procedure. The buffer *k* of type LINE will be used for WRITE storage. The FILE *flush* is of type TEXT and will output to a file named /dev/tty (i.e. the system console in UNIX; that is, the procedure will write to the screen). The variable *state_string* holds the text value of the state.

The CASE statement is used to assign the appropriate string value to the variable *state_string* in preparation for outputting the information to a file. The WRITE statement then writes the value of *state_string* to the buffer *k*. The WRITE statement further specifies that the string should be left justified and be 7 spaces wide.

Finally, the WRITELINE sends the buffer *k* to the file *flush*. The text is then written to the screen.

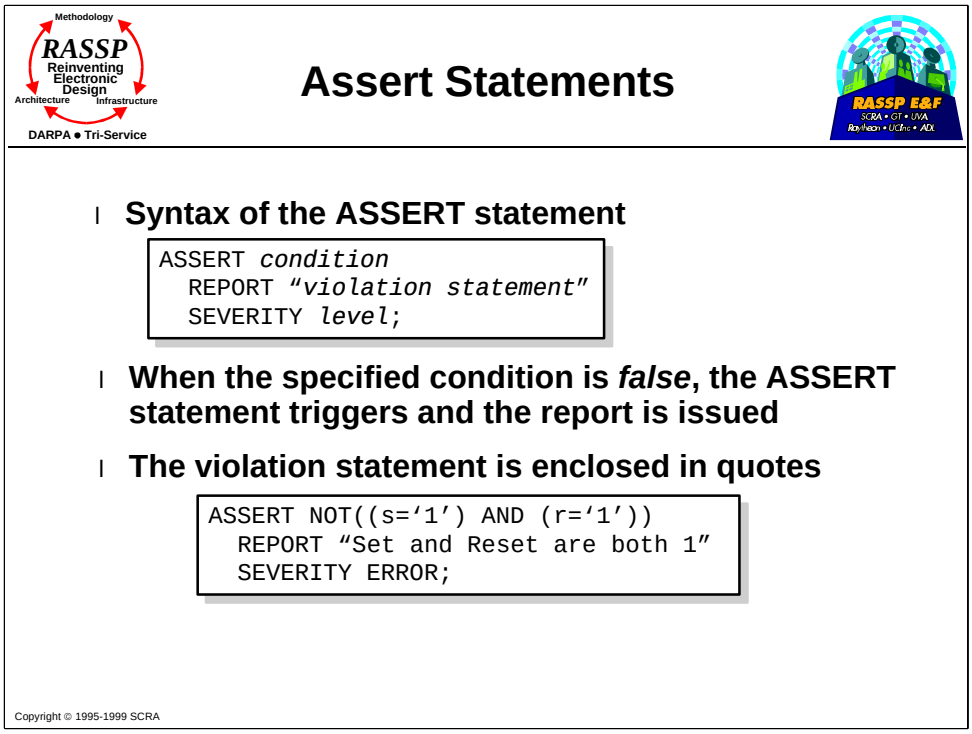
Note that this particular procedure would not work well for writing to a file since the file is re-initialized every time the procedure is used, and thus the text would always be written to the beginning of the file. However, using TEXTIO to write to a file may be accomplished by passing the file to the procedure as a parameter, or by using a process that implements the same functionality, for example.

Based on [Navabi93]



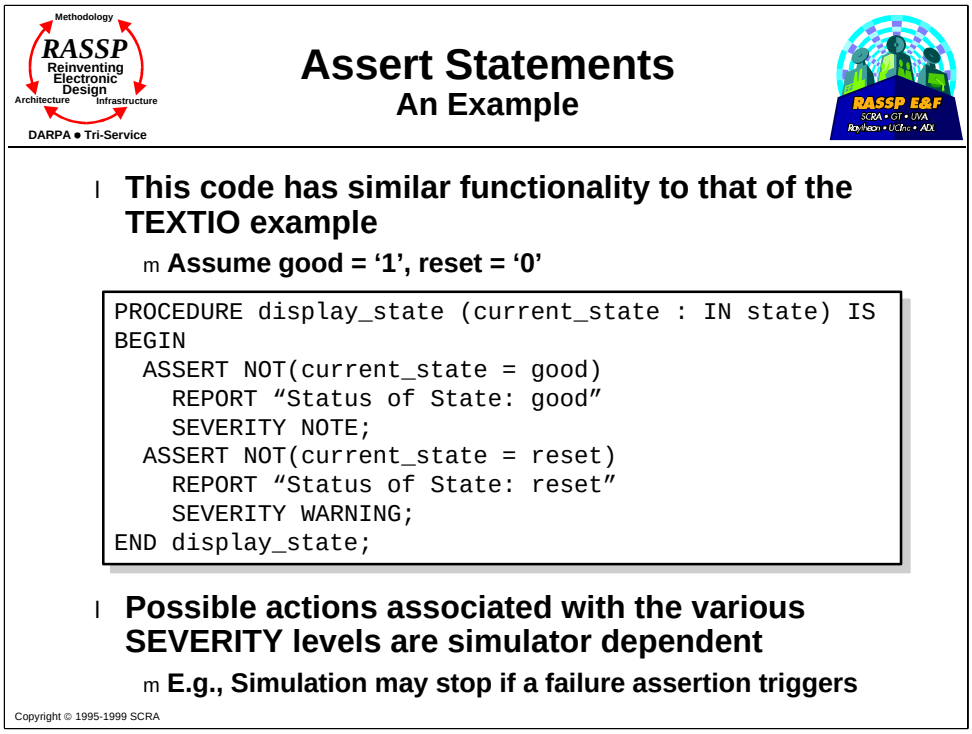
The ASSERT statement is used to alert the user of some condition inside the model. When the expression in the ASSERT statement evaluates to FALSE, the associated text message is displayed on the simulator console. Additionally, an evaluation of FALSE may be used to halt the simulation, depending on the severity level of the associated ASSERT statement.

The four severity levels, in increasing severity, are listed in this slide. However, the simulator actions associated with each severity level are simulator dependent. For example, a simulator implementation be use the *Failure* condition to halt a simulation but continue a simulation under the other assertion conditions.



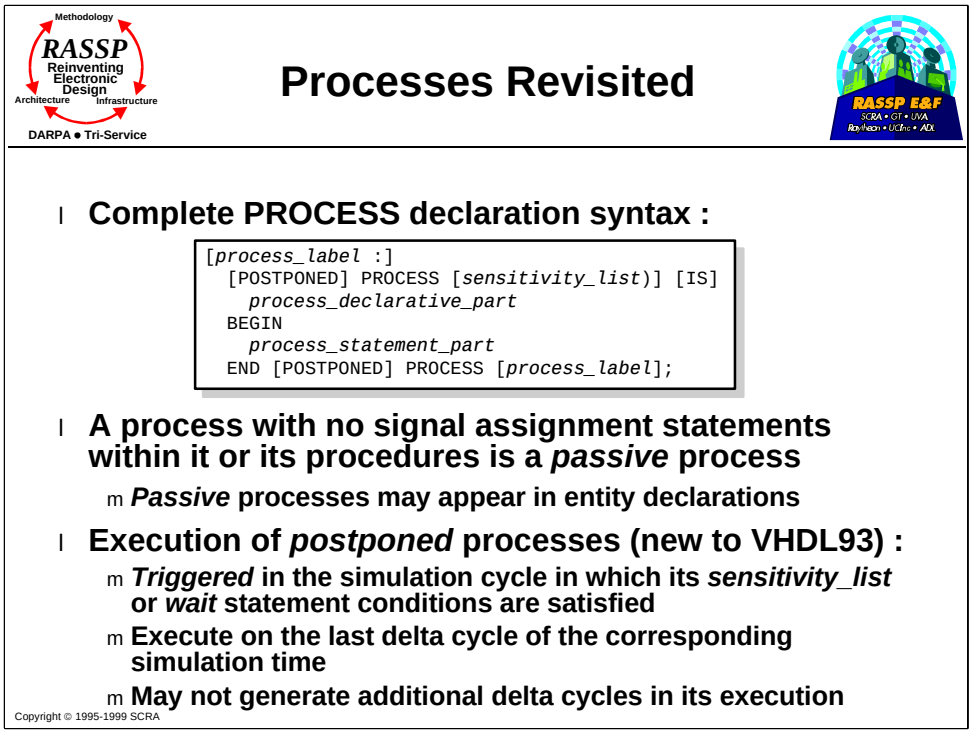
This slide shows the syntax of the ASSERT statement. The ASSERT statement will trigger when the condition is false. The REPORT statement to be displayed is enclosed in quotes.

The Set and Reset lines of the S-R flip-flop in this example cannot simultaneously equal one. Therefore, the ASSERT statement evaluates to FALSE (most easily described using the NOT function) if this situation is observed during simulation.



The example shown here provides a similar functionality to the TEXTIO example shown previously. The ASSERT statements are used to display the current state of a FSM. Note that these ASSERT statements are concurrent. ASSERTs can be concurrent or sequential depending on whether they appear outside or inside VHDL processes, respectively. ASSERTs can also be put in *entity* statements.

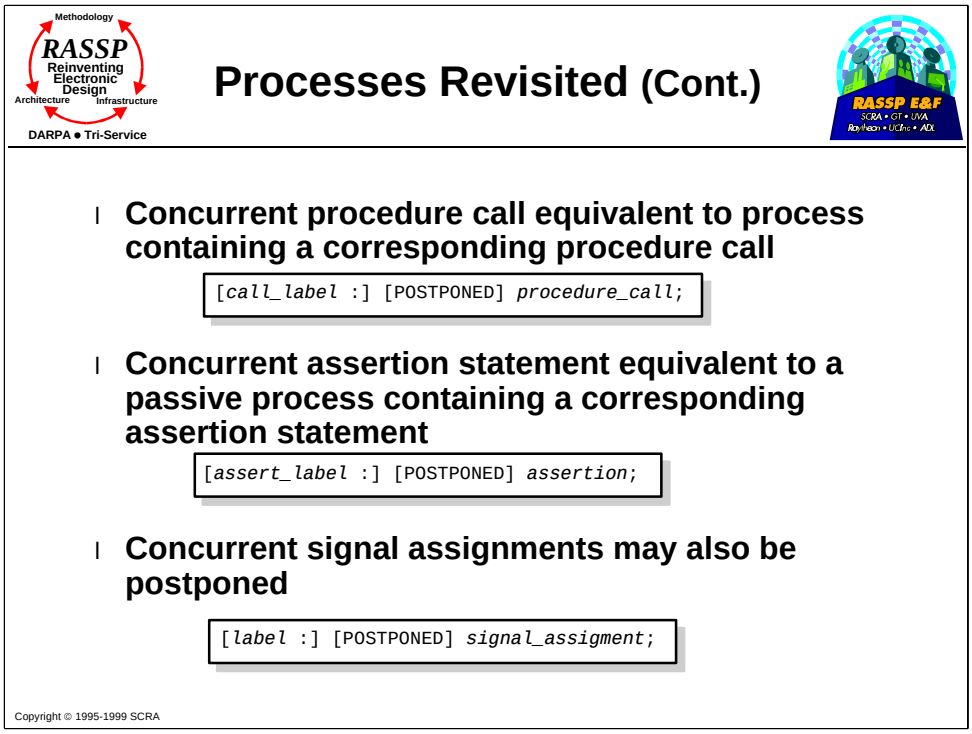
While this procedure does a similar job to the TEXTIO example, it can provide more information to the user and the simulator. The SEVERITY level may cause the simulator to pause or stop altogether. While these actions are implementation defined, they can be useful.



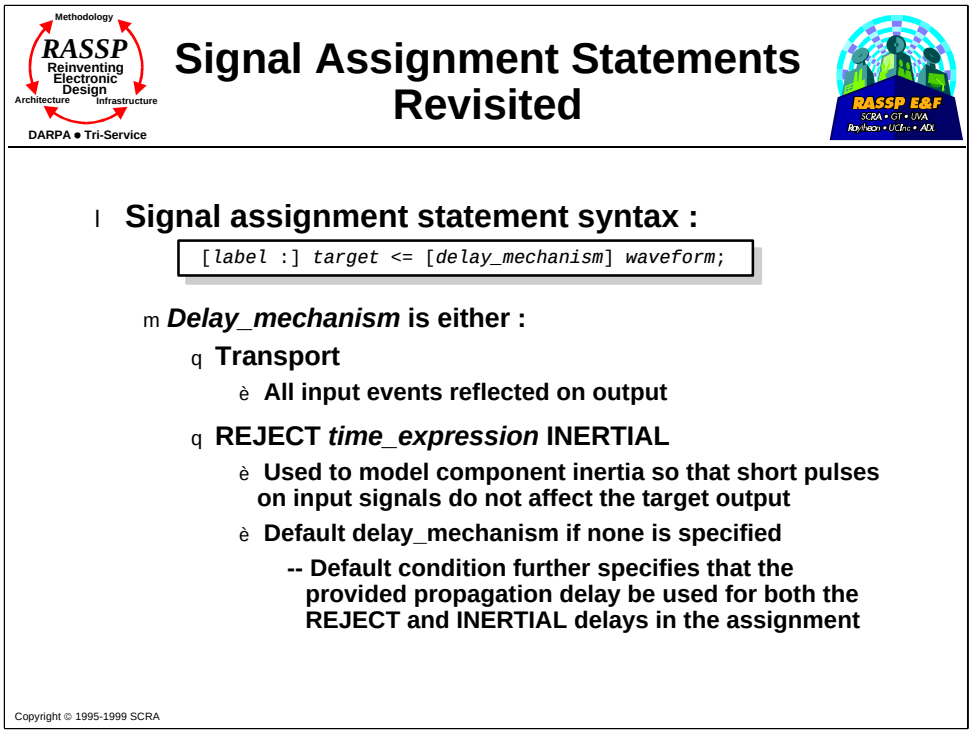
The full syntax of the VHDL process statement is shown here. Two important points are made in this slide.

First, the notion of a passive process is introduced. Because passive processes do not create events in the VHDL timing cycle (i.e. they do not make signal assignments), they may be included in VHDL *entity* declarations where they may be used with TEXTIO or assert statements to report on the state of a simulation, for example.

Second, the postponed process was introduced in VHDL93 to allow a modeler to implement processes that will not be executed until the last possible moment in the simulation cycle. Postponed processes may be used to allow transient conditions to settle out before a simulation state is examined or an assignment is made. Note that any signal assignment in a postponed process must include an assigned delay (i.e. cannot default to a delta cycle delay) to prevent the addition of further delta cycles within the simulation cycle such that the delta cycle in which the postponed process executed would no longer be the last of the simulation cycle.

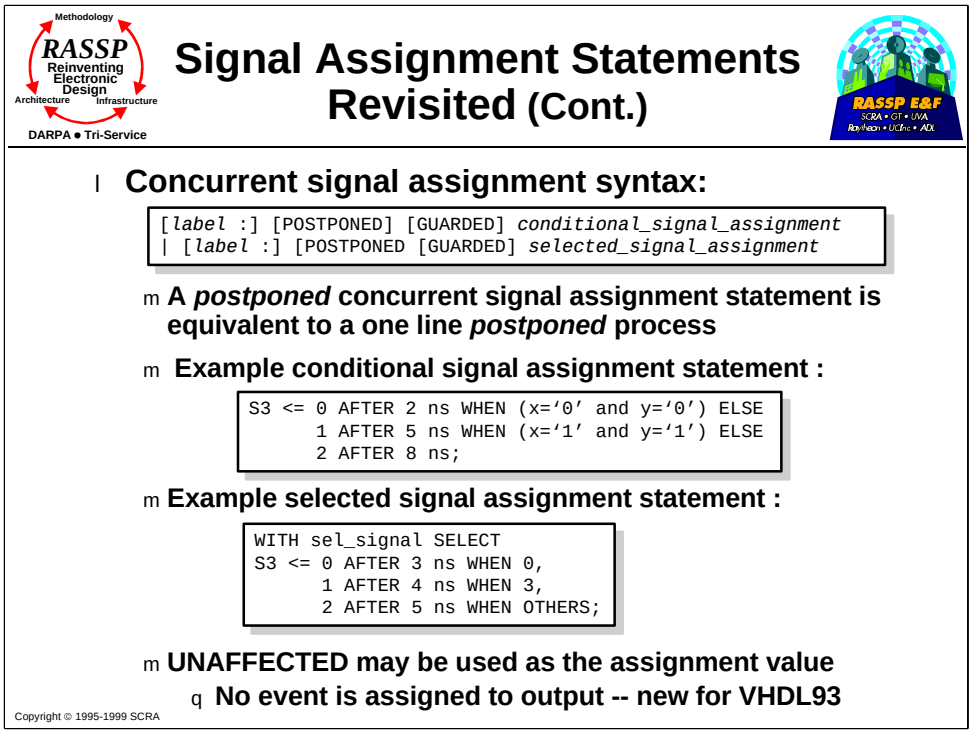


This slide reiterates the equivalence between processes and other concurrent statements. Note that many concurrent statements may be similarly postponed, for example, so that their executions will only occur in the final delta cycle of a simulation cycle.



In this section, signal assignment statements are revisited paying special attention to both the similarities and the differences between concurrent and sequential signal assignment statements.

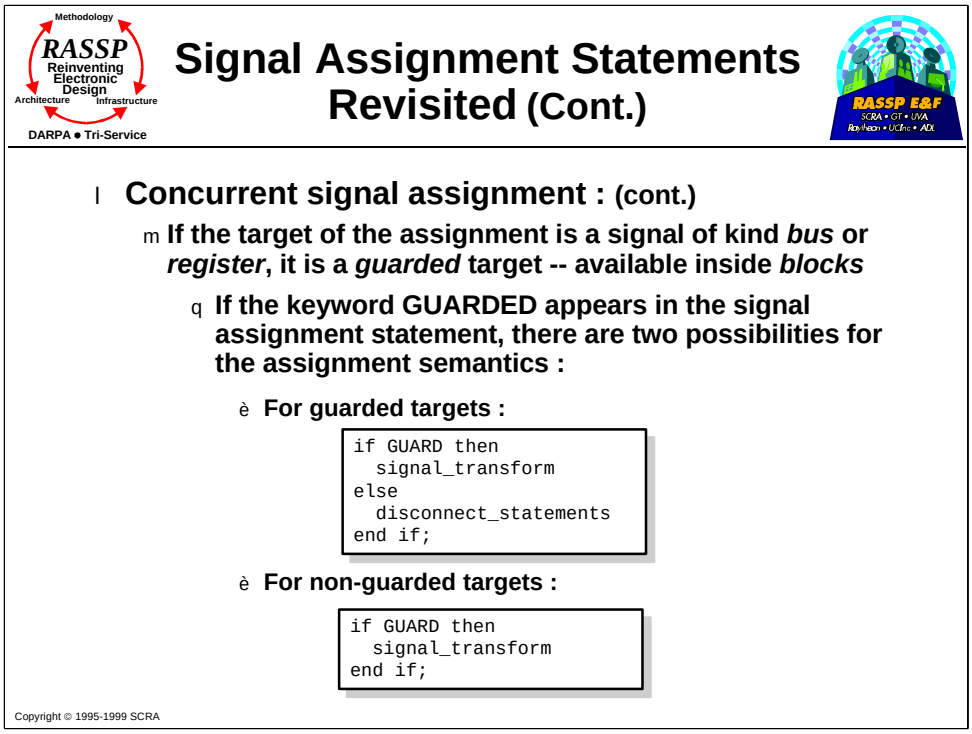
The delay_mechanism construct is common to both concurrent and sequential signal assignment statements. It provides flexibility in determining the response to changes to input signals.



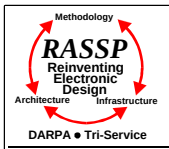
The syntax for concurrent signal assignment statements is shown here. Note that there are two types of concurrent signal assignment statements, conditional and selected. The conditional signal assignment statement is very general in that any readable signals or inputs may be tested to determine the value to be assigned to the target. Note that the simple concurrent signal assignment statement (e.g. *A* <= *B*;) is simply the degenerate case of a conditional signal assignment statement.

The selected signal assignment statement is reminiscent of a CASE statement in that the condition of a predetermined signal is examined to determine the value to be assigned to the target.

The keyword UNAFFECTED may be used as the assignment value so that the output can be left unchanged when the required conditions for such an (in)action are met.



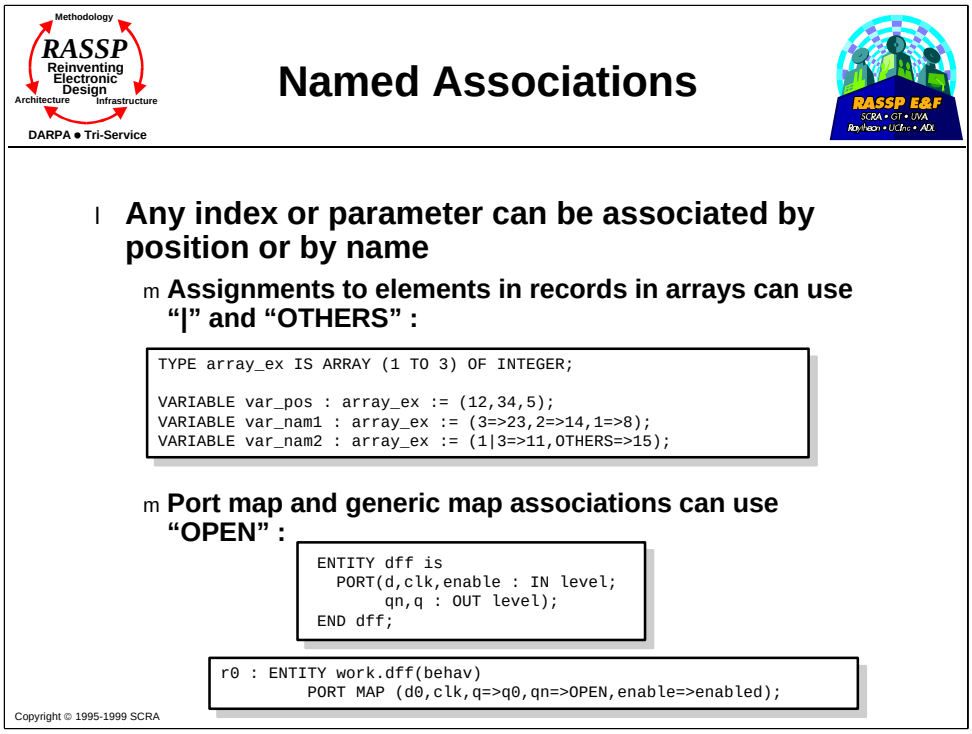
Recalling the previous presentation of VHDL BLOCKs and GUARDs, the target of a concurrent signal assignment statement containing the keyword GUARDED and appearing within a BLOCK statement is a *guarded* target. The use of BLOCKs and GUARDs allows *guarded* targets to have their signal drivers disconnected (i.e. turned off) so that another concurrent signal assignment statement to the same target signal can determine the signal's value without the use of a VHDL Bus Resolution Function. This mechanism is analogous to the use of tri-state bus drivers in digital hardware designs.



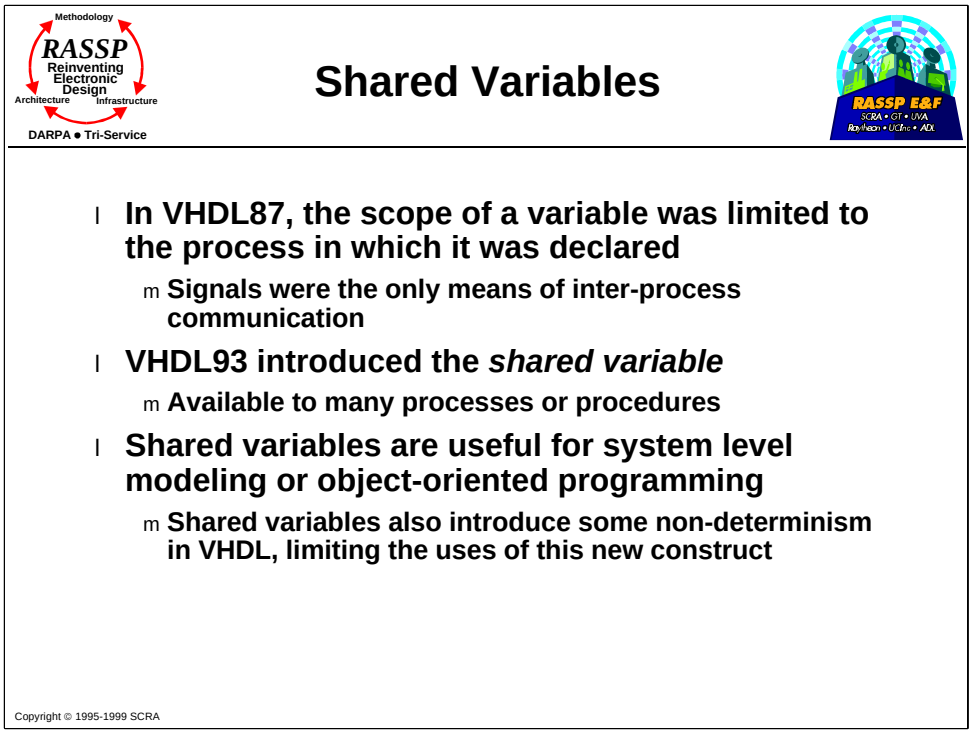
Signal Assignment Statements Revisited (Cont.)

- **Sequential signal assignment statement :**
  - **No mechanisms for guarded, postponed, conditional, or selected signal assignments**
    - **No *guarded* statements because blocks are concurrent rather than sequential statements**
    - **No *postponed* statements because sequential signal assignment statements are NOT equivalent to one line processes**
    - **No conditional or selected signal assignment statements because their function is provided by other means in sequential statements**
      - **E.G. IF-THEN_ELSE and CASE statements**
  - **UNAFFECTED not allowed as an assignment value**
    - **Not needed since no conditional or selected assignment statements are available**

Additional differences in functionality between sequential and concurrent signal assignment statements are shown here.



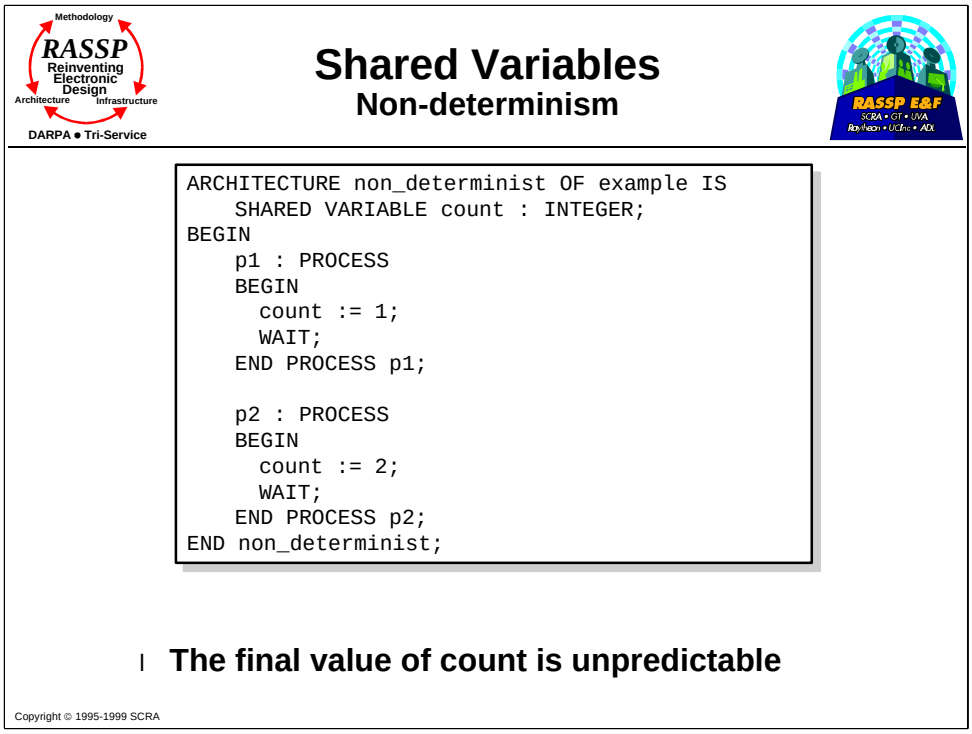
At any VHDL assignment to objects with parameters or indices, associations may be made by position, name, or by a combination of the two as long as the association is not then made ambiguous. Named associations are highlighted here for two reasons. First, the use of OTHERS can be very useful when assigning the values to an object with many indices. Second, it can be confusing to see an assignment as the one in the declaration of the variable *var_nam1* above in which a constant seems to be assigned to another constant when in actuality it is a constant being assigned to the location referenced by a constant index.



VHDL87 limited the scope of the variable to the process in which it was declared. Signals were the only means of communication between processes, but signal assignments require an advance in either delta time or simulation time.

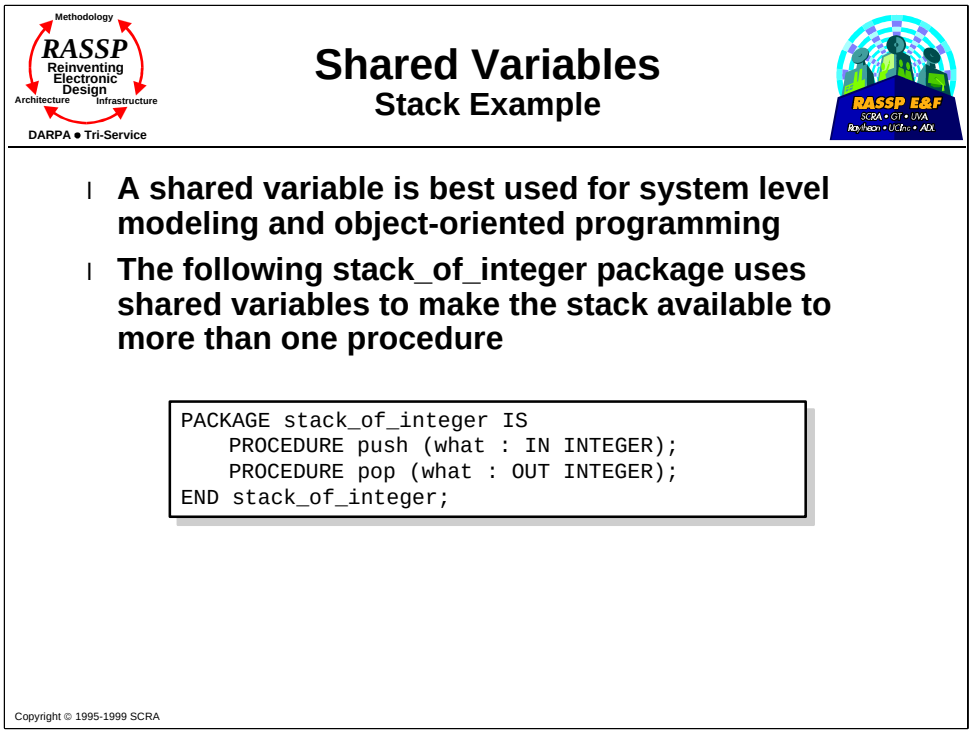
VHDL '93 introduced shared variables which are available to more than one process. Like ordinary VHDL variables, their assignments take effect immediately. However, caution must be exercised when using shared variables because multiple processes making assignments to the same shared variable can lead to unpredictable behavior if the assignments are made concurrently. The VHDL '93 standard does not define the value of a shared variable it two or more processes make assignments in the same simulation cycle.

[Bergé93]



The syntax of the shared variable is similar to that of the normal variable. However, the keyword SHARED is placed in front of VARIABLE in the declaration

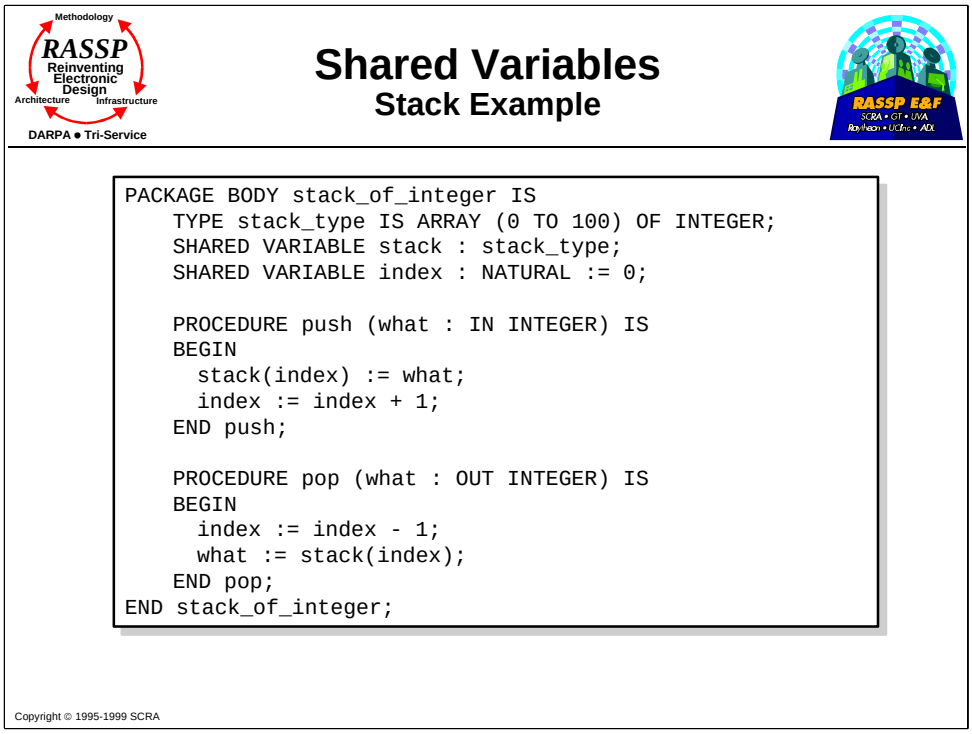
[Bergé93]



As an example of where shared variables are useful, the *stack_of_integer* package in this and the next slide uses a shared variable in two procedures used to maintain a stack. The designer is responsible for ensuring that no two processes call these two procedures at any one time.

The package declarative region is shown here declaring two procedures, push and pop.

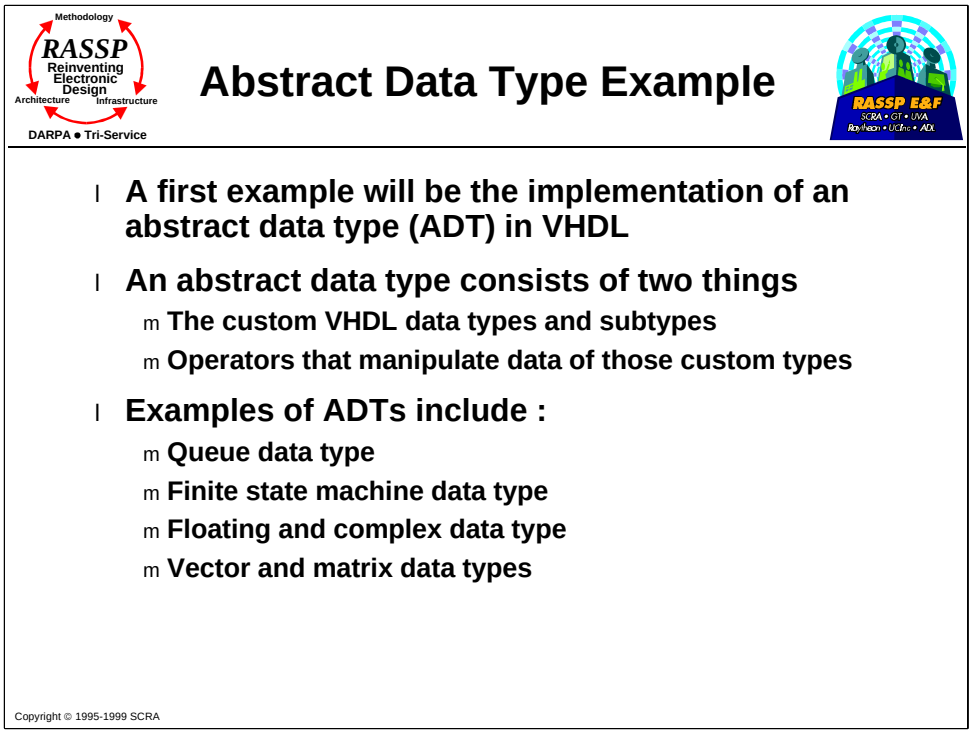
[Bergé93]



[Bergé93]

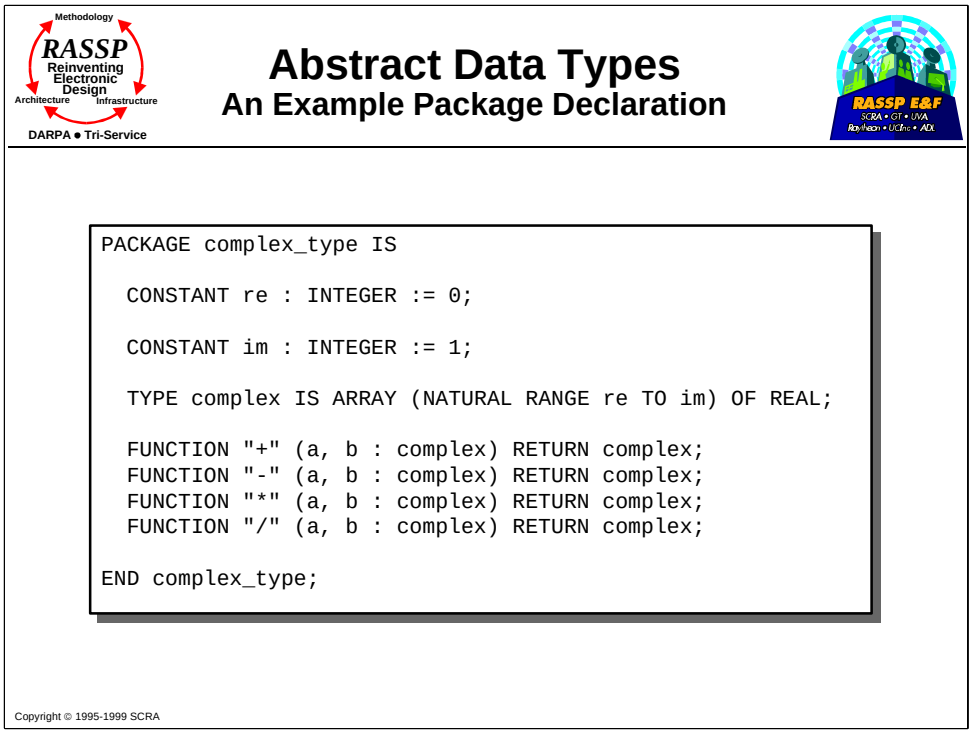
Outline

- Introduction
- Revisiting some VHDL constructs
- **Examples**
  - Abstract data type example
  - Example from UVA ADEPT
- Summary

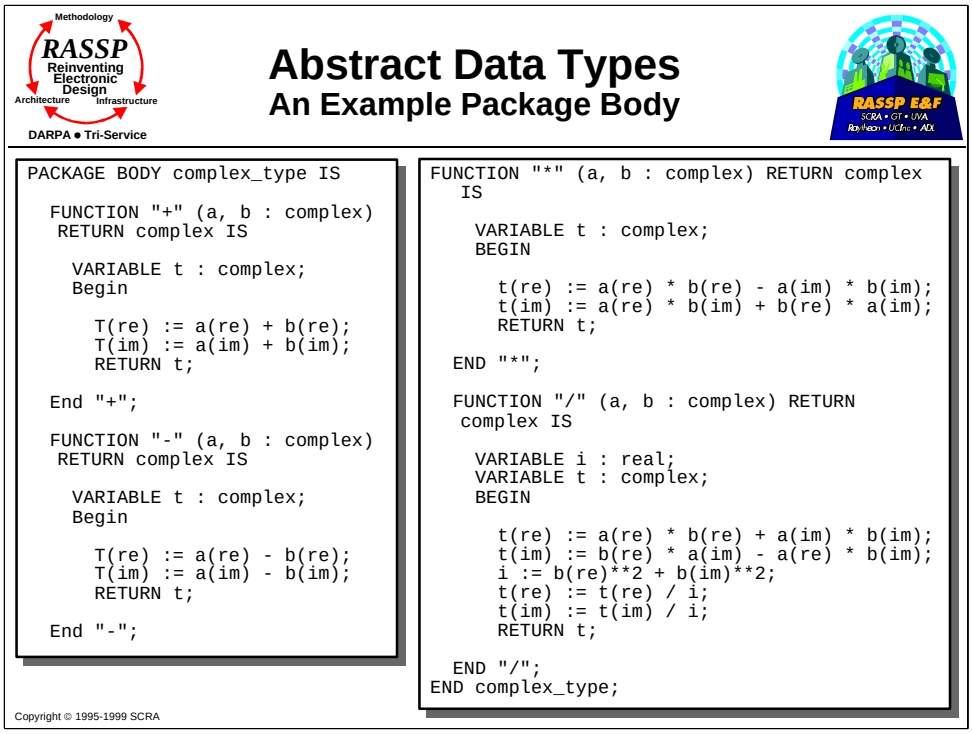


Abstract data types (ADTs) are objects which can be used to represent an activity or component in behavioral modeling. An ADT supports data hiding, encapsulation, and parameterized reuse. As such they give VHDL some object-oriented capability.

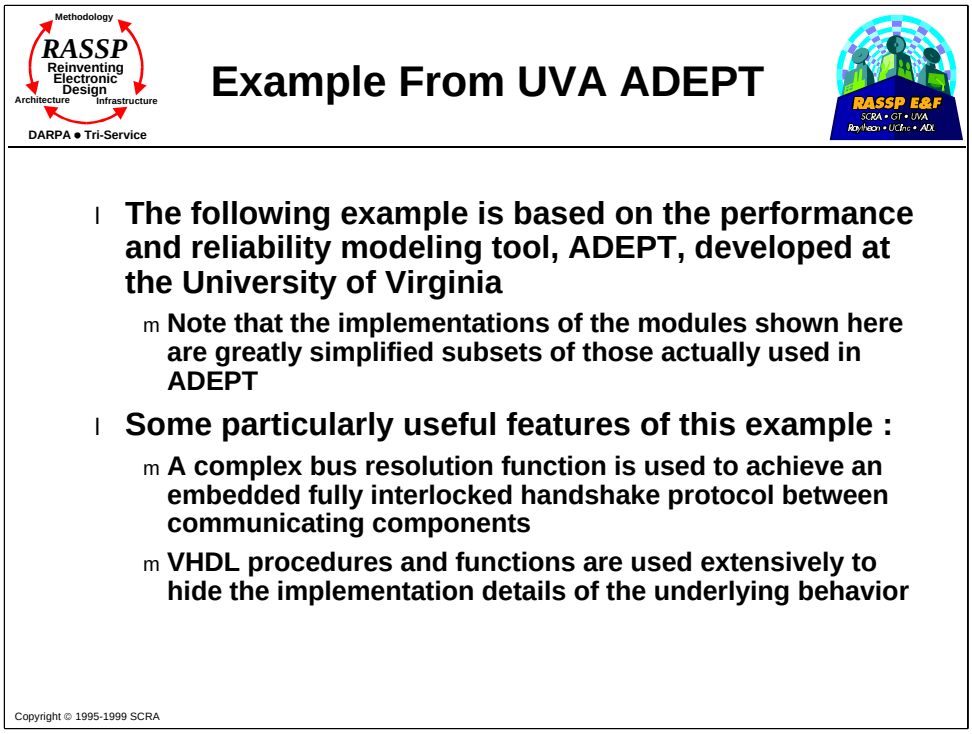
An ADT is both a data structure (such as a stack, queue, tree, etc.) and a set of functions (e.g. operators) that provide useful services of the data. For example, a stack ADT would have functions for pushing an element onto the stack, retrieving an item from the stack, and perhaps several user-accessible attributes such as whether the stack is full or empty.



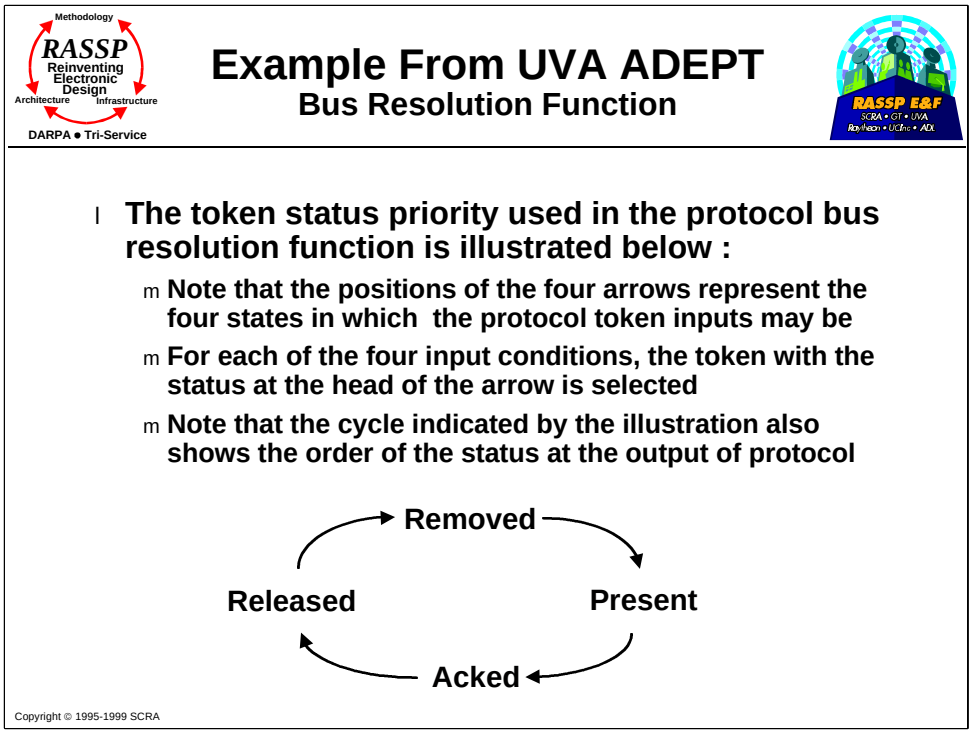
This is a package declaration for a package that implements a complex number data type. Note that the data type is given as well as some standard operators on that type.



This is the package body showing the implementation of the standard operators on the complex type.



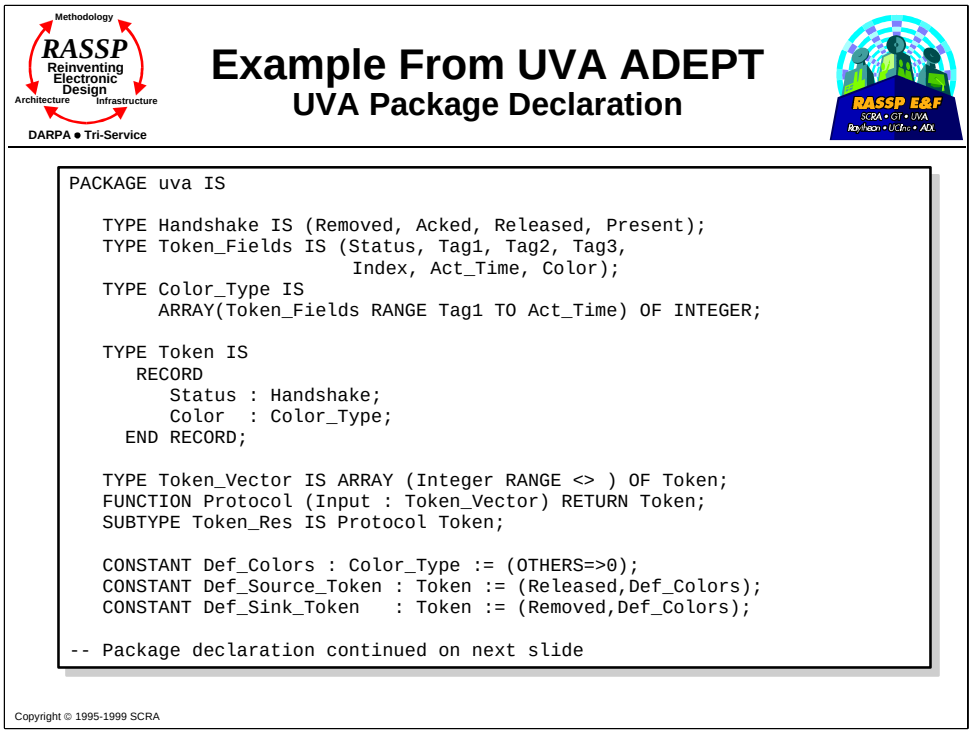
This section presents the description of some simple performance modeling elements that are based on elements from UVA's ADEPT tool. This example will illustrate the use of a Bus Resolution Function to implement an embedded communication protocol used to pass information between components. In addition, functions and procedures are used extensively throughout the example to enhance readability and reuse as well to abstract away implementation details.



This slide illustrates the priority implemented in the Bus Resolution Function Protocol. The simplest case to consider (and the only one that will be used in the following example) is for a point-to-point connection in which one element serves as the *token* source and the other serves as the *token* sink. In this case, the status of the output *token* for the source will be either *Present* or *Released*, and the status of the output *token* for the sink will be either *Acked* or *Removed*.

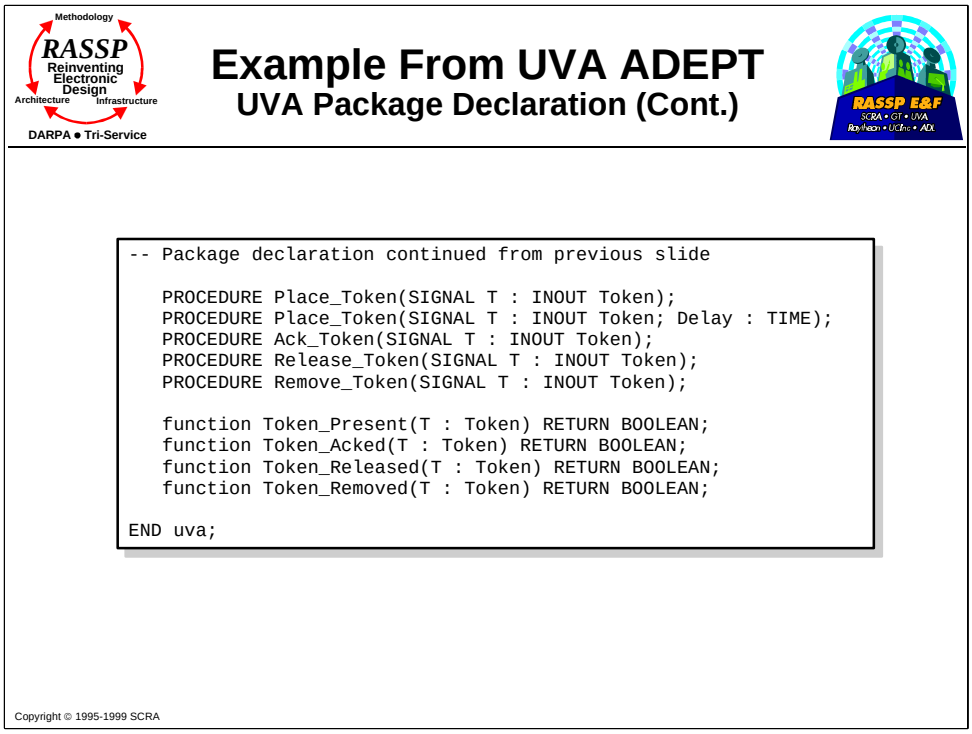
The circle in the slide above serves two related purposes. First, note that at any one time, the arrows at the “corners” indicate the four possible states in which the two *token* drivers can be. For any of these four conditions, *protocol* will select the *token* that is at the head of the arrow.

The second purpose of the circle is to illustrate the sequence of *token* status conditions that will be seen by an observer on the signal connecting the two elements during a communication.

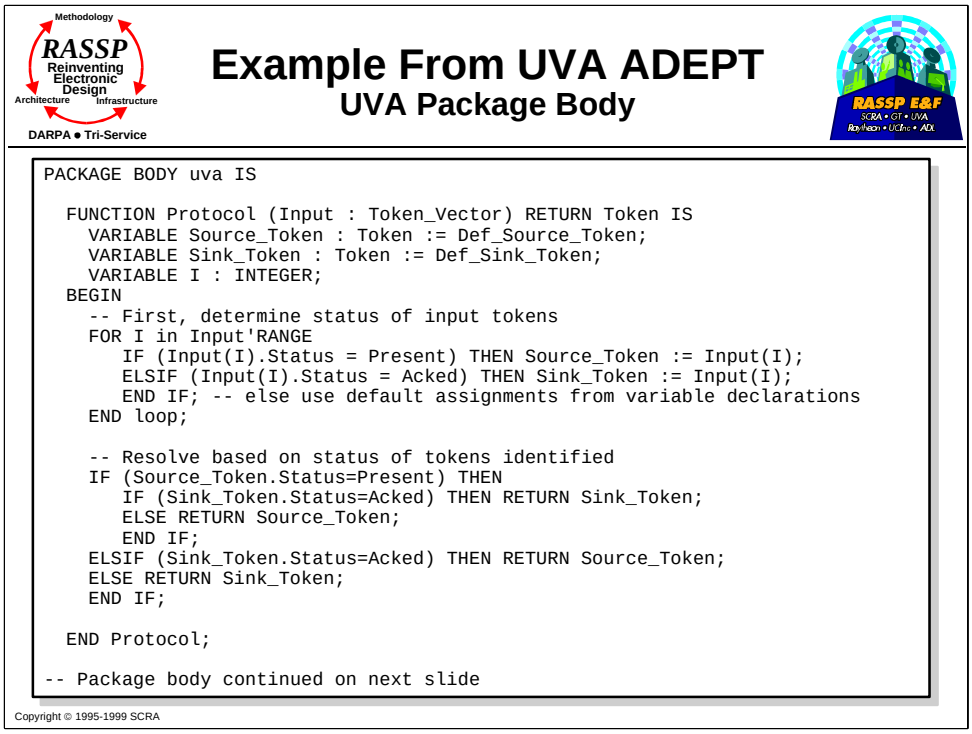


The package declaration required for this example is shown here. Several required data types and useful constants are declared. Note, for example, that the *token* type is a record that contains an enumerated type and an array of integers.

[UM93]



This continues the declaration section of the package with declarations of a number of useful procedures and functions.



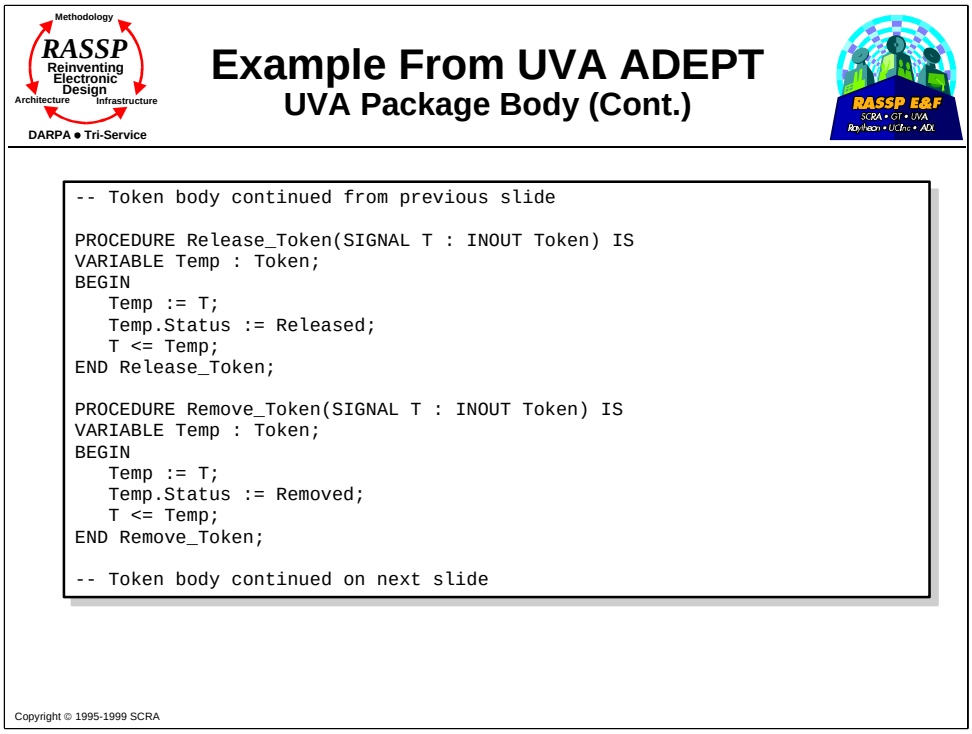
The body of the package used in the example is shown here. The complete VHDL file, which spans several slides, includes the implementation of the various functions and procedures declared in the declarative section of the package.

The implementation of the Bus Resolution Function *protocol* is shown in this particular slide. The first section of the function searches through the input *token_vector* to find a sink *token* of status *Acked* and/or a source *token* status *Present* to select a single source *token* and a single sink *token* between which to arbitrate. If no appropriate source or sink *token* is found for either of these, default status conditions of *Released* and *Removed* are used for the source and sink *tokens*, respectively. The function then picks the appropriate *token* from between the two tokens selected in the first section via the arbitration priority that was described earlier.

Example From UVA ADEPT

UVA Package Body (Cont.)


```
-- Package body continued from previous slide
  PROCEDURE Place_Token (SIGNAL T : INOUT Token) IS
    VARIABLE Temp : Token;
  BEGIN
    Temp := T;
    Temp.Status := Present;
    T <= Temp;
  END Place_Token;

  PROCEDURE Place_Token (SIGNAL T : INOUT Token; Delay : TIME) IS
    VARIABLE Temp : Token;
  BEGIN
    Temp := T;
    Temp.Status := Present;
    T <= Temp after delay;
  END Place_Token;

  PROCEDURE Ack_Token(SIGNAL T : INOUT Token) IS
    VARIABLE Temp : Token;
  BEGIN
    Temp := T;
    Temp.Status := Acked;
    T <= Temp;
  END Ack_Token;

-- Package body continued on next slide
```

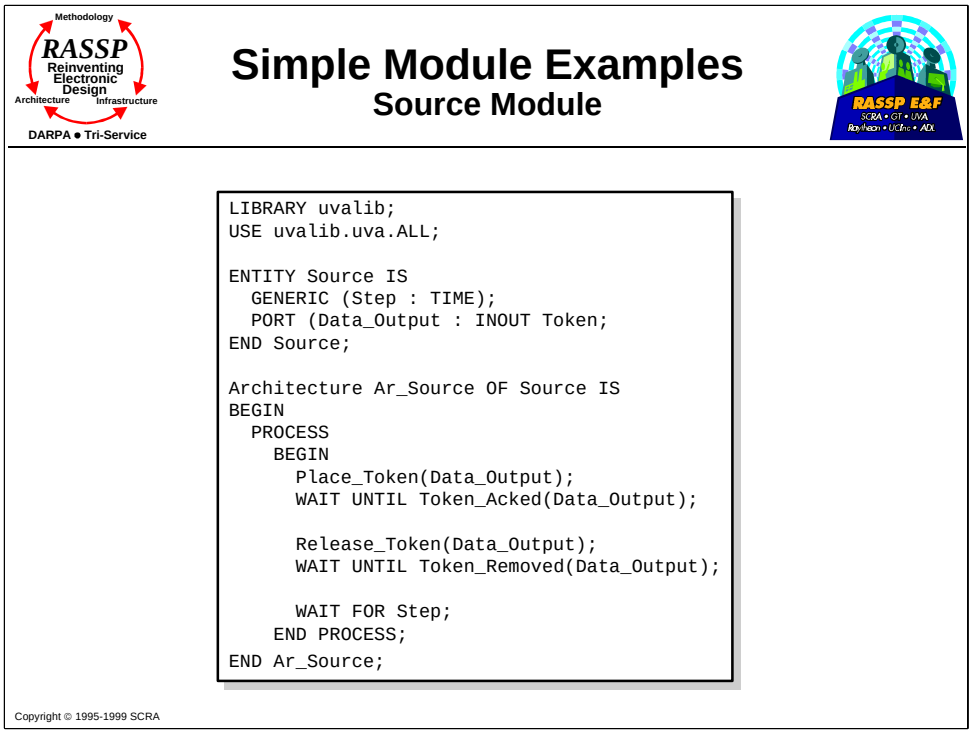


This and the following two slides show the implementation of the remaining procedures and functions in this package body.

Example From UVA ADEPT

UVA Package Body (Cont.)

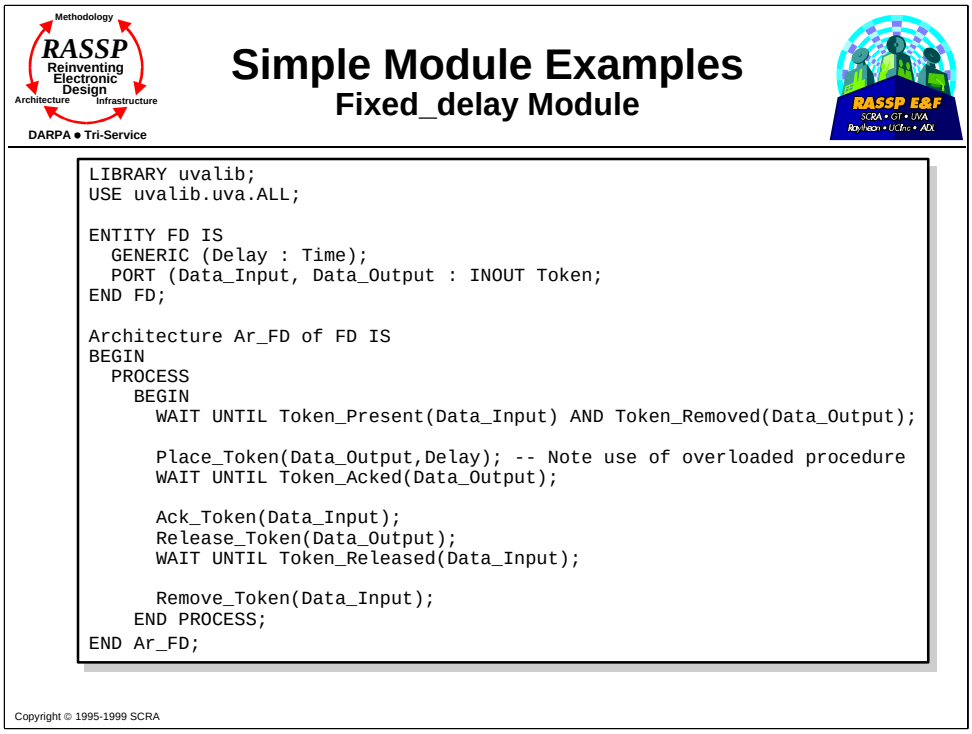

```
-- Package body continued from previous slide
FUNCTION Token_Present (T : Token) RETURN BOOLEAN IS
   BEGIN
      IF (T.Status = Present) THEN RETURN TRUE;
      ELSE RETURN FALSE;
      END IF;
   END Token_Present;

FUNCTION Token_Acked(T : Token) RETURN BOOLEAN IS
   BEGIN
      IF (T.Status = Acked) THEN RETURN TRUE;
      ELSE RETURN FALSE;
      END IF;
   END Token_Acked;

FUNCTION Token_Released(T : Token) RETURN BOOLEAN IS
   BEGIN
      IF (T.Status = Released) THEN RETURN TRUE;
      ELSE RETURN FALSE;
      END IF;
   END Token_Released;

FUNCTION Token_Removed(T : Token) RETURN BOOLEAN IS
   BEGIN
      IF (T.Status = Removed) THEN RETURN TRUE;
      ELSE RETURN FALSE;
      END IF;
   END Token_Removed;

END uva;
```



The description of the source module above is a greatly simplified version of the source module found in the ADEPT library. Note that this and the subsequent model descriptions assume that the package presented in this example will be compiled into the “uvalib” library.

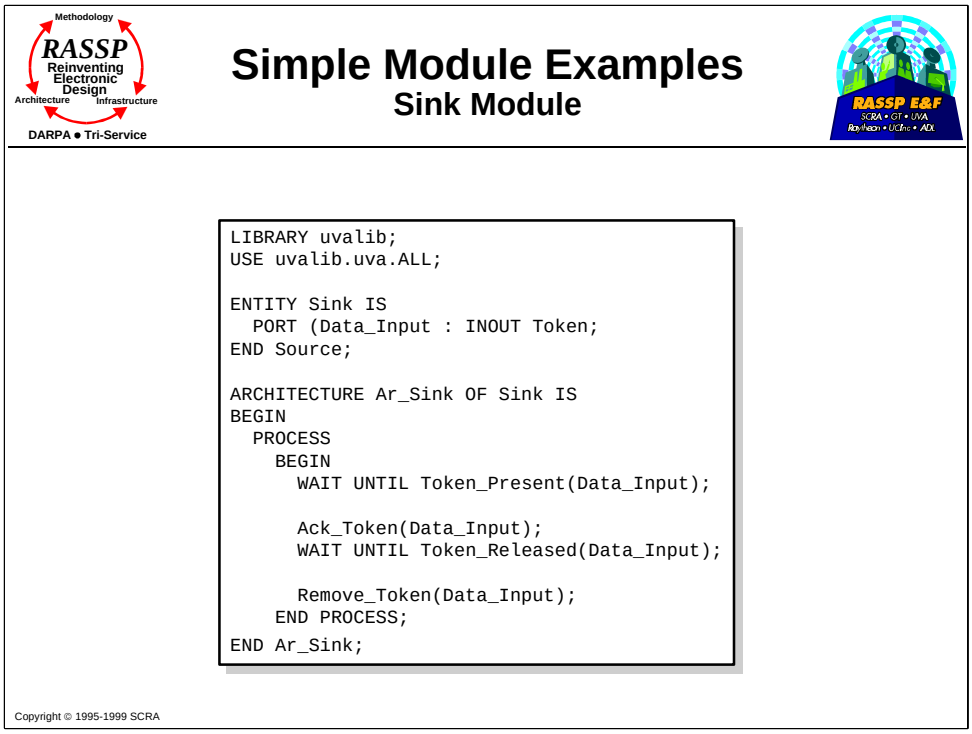
The description above is sequential in nature in that the source module activates its token driver (i.e., “places” a *token*), waits for the adjacent module to activate its driver (i.e., by it “acknowledging” the *token*), inactivates its driver (i.e. ,“releases” the *token*), waits for the adjacent module to inactivate its driver (i.e., by it “removing” the *token*), and finally waits for the specified delay before beginning the sequence again.

[UM93]



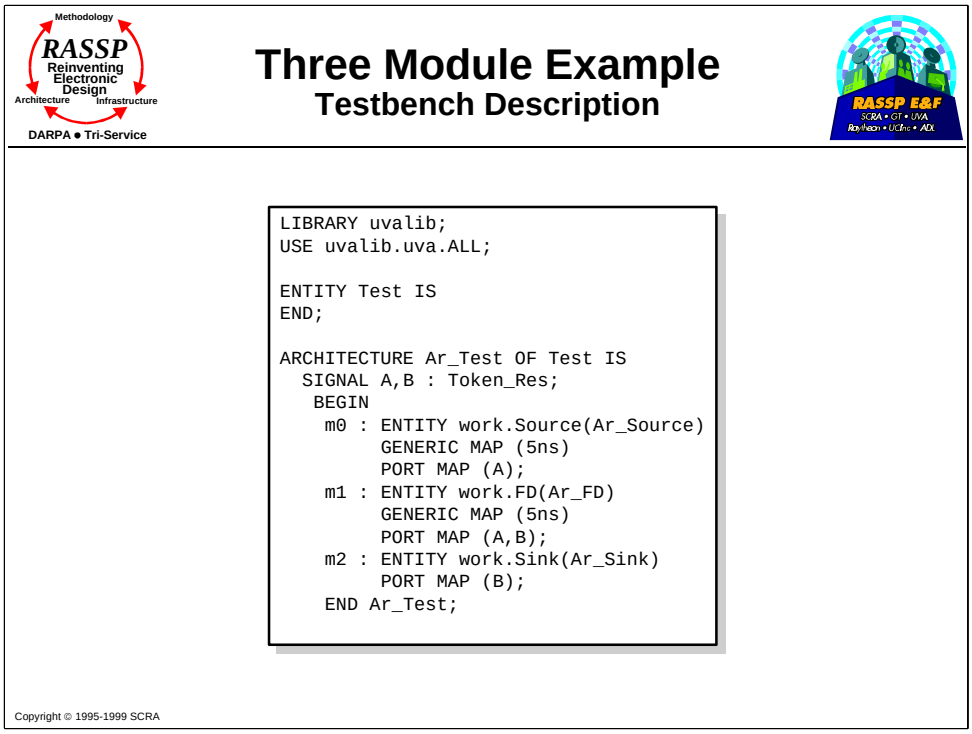
A simplified version of the Fixed_Delay module form the ADEPT library is shown above. In this case, the module waits for a *token* to arrive at its input and then places a *token* on its output using an overloaded version of the *place_token* procedure that includes a delay parameter. After the output *token* is acknowledged, the module acknowledges its input *token* and releases its output *token* as it begins to prepare for the arrival of the next *token* on its input by continuing with the *token* status sequence defined by *protocol* .

[UM93]

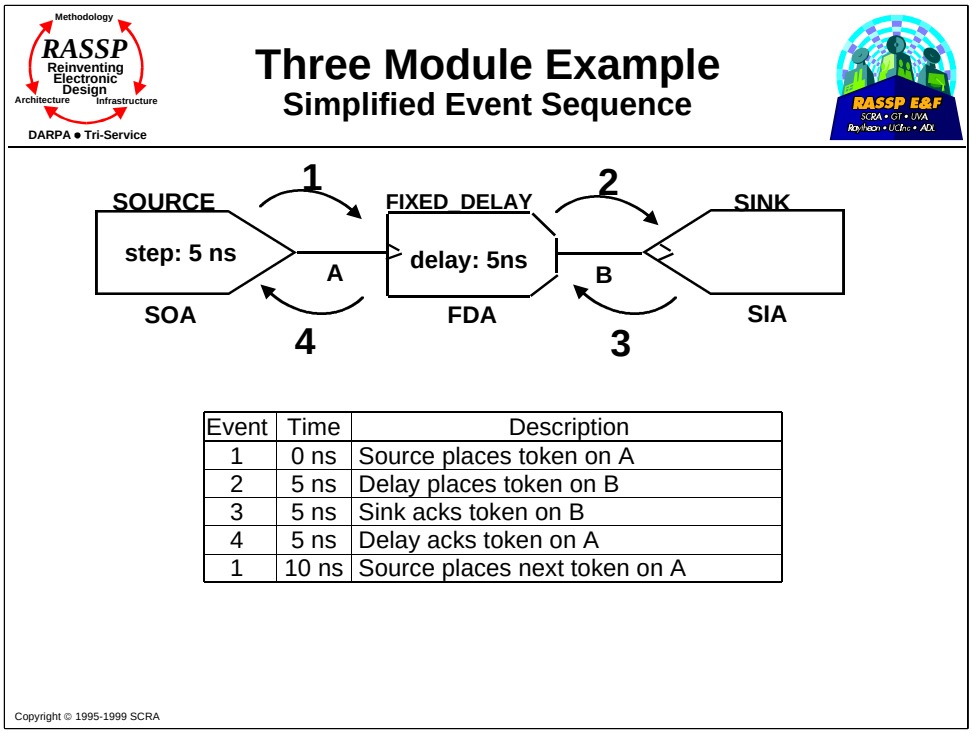


A simplified version of the Sink module from the ADEPT library is shown above. This module waits for an input *token* to arrive. It then acknowledges the input *token* and continues through the *token* status sequence defined by *protocol* to prepare it for the arrival of the next input *token*.

[UM93]



This slide shows the top level VHDL description in which the three modules just described are instantiated and connected to each other by their PORT MAP assignments.

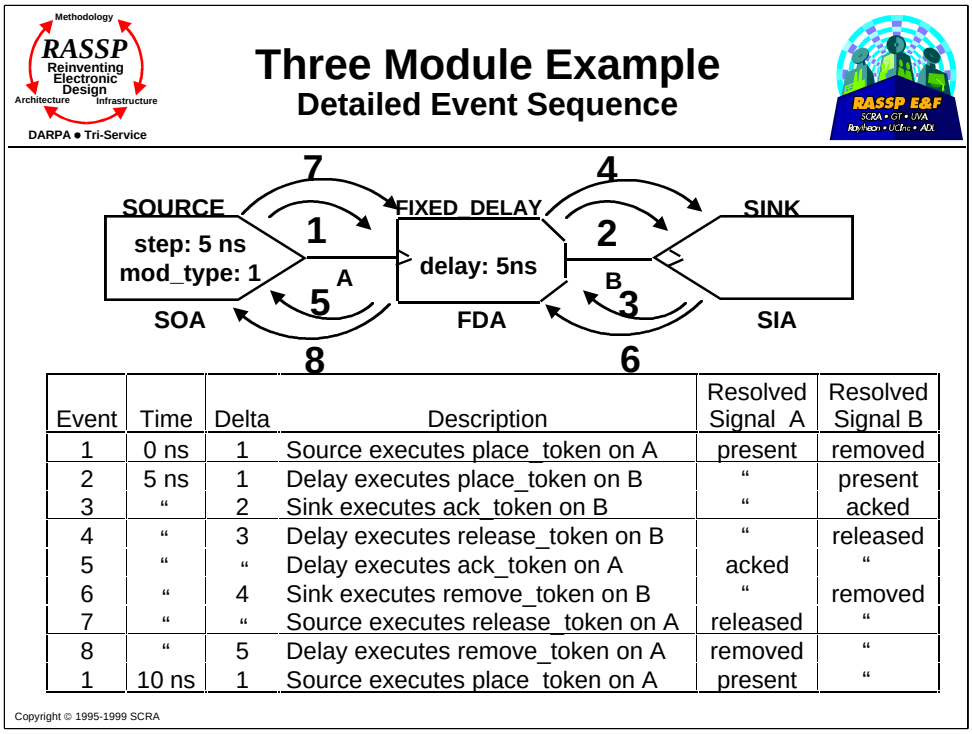


Event	Time	Description
1	0 ns	Source places token on A
2	5 ns	Delay places token on B
3	5 ns	Sink acks token on B
4	5 ns	Delay acks token on A
1	10 ns	Source places next token on A

The three-module example shown in this slide will be used to illustrate the important events in the passing of *tokens* from a *token* source to a *token* sink. As the table shows, the relevant events are the placing and acknowledging of *tokens*. The other two states in the four-cycle handshake, releasing and removing, are only required to effect the interlocked handshake protocol.

Note that the fixed_delay module does not acknowledge the source's *token* until its output has been acknowledged by the sink module (i.e., there is no buffering between inputs and outputs). This is an important characteristic of the communication standard used by ADEPT modules (unless explicitly stated otherwise, as in the BUFFER module).

[UM93]



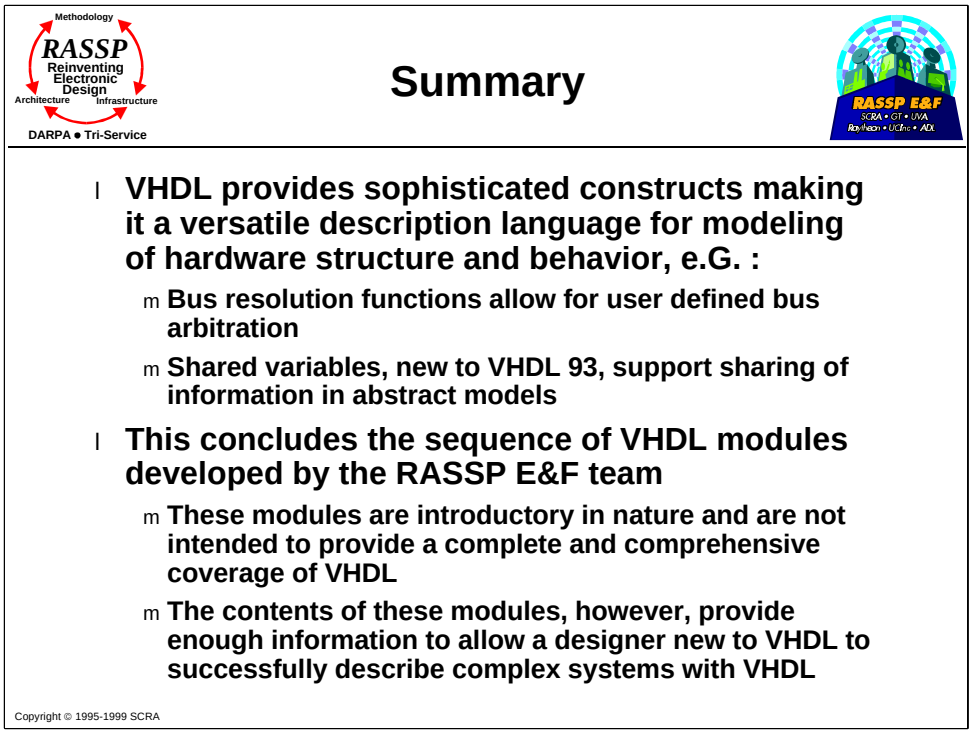
Event	Time	Delta	Description	Resolved Signal A	Resolved Signal B
1	0 ns	1	Source executes place_token on A	present	removed
2	5 ns	1	Delay executes place_token on B	"	present
3	"	2	Sink executes ack_token on B	"	acked
4	"	3	Delay executes release_token on B	"	released
5	"	"	Delay executes ack_token on A	acked	"
6	"	4	Sink executes remove_token on B	"	removed
7	"	"	Source executes release_token on A	released	"
8	"	5	Delay executes remove_token on A	removed	"
1	10 ns	1	Source executes place_token on A	present	"

This example shows the entire four-cycle sequence of token assignments made in the passing of *tokens* in the model. Note that after a *token* is acknowledged, the release and removal of that *token* take place in delta time (e.g., event 4 and 6 for B in the example).

[UM93]

Outline

- Introduction
- Revisiting some VHDL constructs
- Examples
- **Summary**



- **VHDL provides sophisticated constructs making it a versatile description language for modeling of hardware structure and behavior, e.G. :**
  - **Bus resolution functions allow for user defined bus arbitration**
  - **Shared variables, new to VHDL 93, support sharing of information in abstract models**
- **This concludes the sequence of VHDL modules developed by the RASSP E&F team**
  - **These modules are introductory in nature and are not intended to provide a complete and comprehensive coverage of VHDL**
  - **The contents of these modules, however, provide enough information to allow a designer new to VHDL to successfully describe complex systems with VHDL**

This instructional module has illustrated the versatility of VHDL in supporting abstraction and information encapsulation to facilitate the description of complex systems. Example system design and description methodologies based on VHDL were included primarily to illustrate the VHDL constructs used to support modeling at higher levels of design abstraction.

References

[Bergé93] Bergé, J-M., Fonkoua, A., Maginot, S., Rouillard, J., *VHDL'92: The New Features of the VHDL Hardware Description Language*, Kluwer Academic Publishers, 1993.

[Bhasker95] Bhasker, J. *A VHDL Primer*, Prentice Hall, 1995.

[Hein95] Hein, Karl, et al. "RASSP VHDL Modeling Terminology and Taxonomy-Revision 1.0", Proceedings of the 2nd Annual RASSP Conference, July 24-27, 1995.

[Honeywell94] Carpenter, T., Rose, F., Steeves, T., Performance Modeling with VHDL, Honeywell Systems & Research Center, 1994.

[Honeywell95] Honeywell Performance Modeling Library, 1995.

[IEEE] All referenced IEEE material is used with permission.

[LRM93] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, 1994.

[Navabi93] Navabi, Z., VHDL: Analysis and Modeling of Digital Systems, McGraw-Hill, 1993.

[UM93] Cutright, E.D., Rao, R., Johnson, B.W., Aylor, J.H., A Handbook on the Unified Modeling Methodology Building Block Set, CSIS, http://www.ee.virginia.edu/research/CSIS/, University of Virginia, 1993.

[Richards97] Richards, M., Gadient, A., Frank, G., eds. *Rapid Prototyping of Application Specific Signal Processors*, Kluwer Academic Publishers, Norwell, MA, 1997