



# Advanced Concepts in VHDL

## RASSP Education & Facilitation

### Module 13

Version 3.00

Copyright 1995-1999 SCRA

All rights reserved. This information is copyrighted by the SCRA, through its Advanced Technology Institute (ATI), and may only be used for non-commercial educational purposes. Any other use of this information without the express written permission of the ATI is prohibited. Certain parts of this work belong to other copyright holders and are used with their permission. All information contained, may be duplicated for non-commercial educational use only provided this copyright notice and the copyright acknowledgements herein are included. No warranty of any kind is provided or implied, nor is any liability accepted regardless of use.

The United States Government holds "Unlimited Rights" in all data contained herein under Contract F33615-94-C-1457. Such data may be liberally reproduced and disseminated by the Government, in whole or in part, without restriction except as follows: Certain parts of this work to other copyright holders and are used with their permission; This information contained herein may be duplicated only for non-commercial educational use. Any vehicle, in which part or all of this data is incorporated into, shall carry this notice .

Copyright © 1995-1999 SCRA





## Module Goals



- | **To expand on the syntax and semantics of constructs introduced in prior modules to highlight their flexibility**
- | **To introduce new features of VHDL beyond the scope of the previous introductory modules**

Copyright © 1995-1999 SCRA

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**
  - m **Aliases**
  - m **Foreign interfaces**
  - m **Files**
  - m **Textio**
  - m **Assert statements**
  - m **Processes**
  - m **Signal assignment statements**
  - m **Shared variables**

Copyright © 1995-1999 SCRA



## Outline (Cont.)



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

Copyright © 1995-1999 SCRA



## Advantages of Using VHDL



- | **VHDL offers several advantages to the designer**
  - m **Standard language**
    - q **Readily available tools**
  - m **Powerful and versatile description language**
  - m **Multiple mechanisms to support design hierarchy**
  - m **Versatile design reconfiguration support**
  - m **Support for multiple levels of abstraction**

Copyright © 1995-1999 SCRA

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**

- m Aliases
- m Foreign interfaces
- m Textio
- m Assert statements
- m Processes
- m Signal assignment statements
- m Shared variables

- | Examples

- | Summary

Copyright © 1995-1999 SCRA

## Fundamental View of VHDL

- | **Fundamentally, VHDL follows event-driven concurrent execution semantics :**

```
ARCHITECTURE arch_label OF ent_label IS  
  [architecture_declarations]  
BEGIN  
  [block_statement] |  
  [process_statement] |  
  [concurrent_procedure_call_statement] |  
  [concurrent_assertion_statement] |  
  [concurrent_signal_assignment_statement] |  
  [component_instantiation_statement] |  
  [generate_statement]  
END [arch_label];
```

- m **Sequential execution available inside processes**
- m **Note component instantiations are concurrent statements**

Copyright © 1995-1999 SCRA

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.





# Aliases



- | **Aliases can significantly improve the readability of VHDL descriptions by using a shorthand notation for names**
- | **Aliases allow reference to named items in different ways:**
  - ALIAS data\_bus: mvl\_vector(7 DOWNT0 0) is data\_word(15 DOWNT0 8);**
- | **Aliases can rename any named item except labels, loop parameters, and generate parameters**

Copyright © 1995-1999 SCRA

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]

## Alias An Example

- | **An alias of an overloaded subprogram or literal requires a signature to determine the correct value to return**

```
TYPE mvl IS ('U', '0', '1', 'Z');  
TYPE trinary IS ('0', '1', 'Z');  
  
ALIAS mvl0 IS '0' [RETURN mvl];  
ALIAS tri0 IS '0' [RETURN trinary];
```

```
PROCEDURE preset_clear(SIGNAL drv: mvl_vector;  
    pc_value: INTEGER);  
PROCEDURE preset_clear(SIGNAL drv: BIT_VECTOR;  
    pc_value: INTEGER);  
  
ALIAS pcmvl IS preset_clear(mvl_vector, INTEGER);  
ALIAS pcbit IS preset_clear(BIT_VECTOR, INTEGER);
```

Copyright © 1995-1999 SCRA

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]

## Foreign Interfaces

- | **Model description may include portions written in a foreign programming language (e.G. C)**
  - m Subprogram or architecture body can be described in programming language other than VHDL
  - m Designer can incorporate previously written code or code that is difficult to write in VHDL
- | **Details on use of foreign code is largely implementation dependent**
- | **Not possible to include variables, signals, or entities described in a foreign language**

```
ATTRIBUTE FOREIGN OF name: construct IS  
"information/parameters";
```

Copyright © 1995-1999 SCRA

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]

## Foreign Interfaces An Example

```
ENTITY and2 IS
  PORT(a, b: IN BIT;
        c: OUT BIT);
END and2;

ARCHITECTURE c_model OF and2 IS
  ATTRIBUTE FOREIGN OF c_model:
    ARCHITECTURE IS "xxand2(A, B, C)";
BEGIN
END c_model;
```

l **The c\_model architecture is declared as FOREIGN**

- m **No statements are needed in the architecture body as they will never be executed**
- m **The implementation calls the "xxand2" function to perform the actions for the and2 entity**

Copyright © 1995-1999 SCRA

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 a file object, associated types, and certain limited file operations

```
TYPE file_type IS FILE OF type_mark;
PROCEDURE READ(FILE identifier : file_type; value : OUT type_mark);
PROCEDURE WRITE(FILE identifier : file_type; value : IN type_mark);
FUNCTION ENDFILE(FILE identifier : file_type) RETURN BOOLEAN;
```

## | File declarations

m Vhdl87

```
FILE identifier : file_type IS [mode] "file_name";
```

m Vhdl93

```
FILE identifier : file_type [[OPEN mode] IS "file_name"];
```

Copyright © 1995-1999 SCRA

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;
```

## File Opening and Closing

- | In VHDL87, files are opened and closed when the associated file object comes into and goes out of scope
- | In VHDL93, files can be opened in the declaration or predefined procedures can be used:

```
PROCEDURE FILE_OPEN(FILE identifier:file_type;  
    file_name: IN STRING;  
    open_kind: FILE_OPEN_KIND := READ_MODE);  
  
PROCEDURE FILE_OPEN(status: OUT FILE_OPEN_STATUS;  
    FILE identifier: file_type;  
    file_name: IN STRING;  
    open_kind: FILE_OPEN_KIND := READ_MODE);  
  
PROCEDURE FILE_CLOSE(FILE identifier: file_type);
```

Copyright © 1995-1999 SCRA

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.

## Text Input and Output

- | **Basic file operations in VHDL are limited to unformatted input/output**
- | **VHDL includes the TEXTIO package for input and output of ASCII text**
  - m **TEXTIO is located in the STD library**
- | **The following data types are supported by the TEXTIO routines:**
  - m **Bit, bit\_vector**
  - m **Boolean**
  - m **Character, string**
  - m **Integer, real**
  - m **Time**

```
USE STD.TEXTIO.ALL;
```

Copyright © 1995-1999 SCRA

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**
  - m 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:**
  - m **Readline(f,k)**
    - q Reads a line of file f and places it in buffer k
  - m **Read(k,v,...)**
    - q Reads a value of v of its type from k
  - m **Write(k,v,...)**
    - q Writes value v to LINE k
  - m **Writeline(f,k)**
    - q Writes k to file f
  - m **Endfile(f)** returns TRUE at the end of FILE

Copyright © 1995-1999 SCRA

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.





## Using TEXTIO



- | **Reading from a file**
  - m READLINE reads a line from the file into a LINE buffer
  - m READ gets data from the buffer
- | **Writing to a file**
  - m WRITE puts data into a LINE buffer
  - m WRITELINE writes the data in the LINE buffer to file
- | **READ and WRITE have several formatting parameters**
  - m Right or left justification
  - m Field width
  - m Unit displayed (for time)

Copyright © 1995-1999 SCRA

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.

## TEXTIO An Example

### | This procedure displays the current state of a FSM

```
USE STD.TEXTIO.ALL;
TYPE state IS (reset, good);
PROCEDURE display_state (current_state : IN state) IS
  VARIABLE k : LINE;
  FILE flush : TEXT IS OUT "/dev/tty";
  VARIABLE state_string : STRING(1 to 7);
BEGIN
  CASE current_state IS
    WHEN reset => state_string := "reset  ";
    WHEN good => state_string := "good   ";
  END CASE;
  WRITE (k, state_string, LEFT, 7);
  WRITELINE (flush, k);
END display_state;
```

Copyright © 1995-1999 SCRA

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]



## Assert Statement



- | **ASSERT statements are used to print messages at the simulation console when specified runtime conditions are met**
- | **ASSERT statements defined one of four severity levels :**
  - m **Note -- relays information about conditions to the user**
  - m **Warning -- alerts the user to conditions that are not expected, but not fatal**
  - m **Error -- relays conditions that will cause the model to work incorrectly**
  - m **Failure -- alerts the user to conditions that are catastrophic**

Copyright © 1995-1999 SCRA

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.

## Assert Statements

### | Syntax of the ASSERT statement

```
ASSERT condition  
REPORT "violation statement"  
SEVERITY level;
```

### | When the specified condition is *false*, the ASSERT statement triggers and the report is issued

### | The violation statement is enclosed in quotes

```
ASSERT NOT((s='1') AND (r='1'))  
REPORT "Set and Reset are both 1"  
SEVERITY ERROR;
```

Copyright © 1995-1999 SCRA

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.



## Assert Statements An Example



- | **This code has similar functionality to that of the TEXTIO example**

m **Assume good = '1', reset = '0'**

```
PROCEDURE display_state (current_state : IN state) IS
BEGIN
    ASSERT NOT(current_state = good)
        REPORT "Status of State: good"
        SEVERITY NOTE;
    ASSERT NOT(current_state = reset)
        REPORT "Status of State: reset"
        SEVERITY WARNING;
END display_state;
```

- | **Possible actions associated with the various SEVERITY levels are simulator dependent**

m **E.g., Simulation may stop if a failure assertion triggers**

Copyright © 1995-1999 SCRA

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.

## Processes Revisited

### | Complete PROCESS declaration syntax :

```
[process_label :]
[POSTPONED] PROCESS [sensitivity_list] [IS]
    process_declarative_part
BEGIN
    process_statement_part
END [POSTPONED] PROCESS [process_label];
```

### | A process with no signal assignment statements within it or its procedures is a *passive* process

- m *Passive* processes may appear in entity declarations

### | Execution of *postponed* processes (new to VHDL93) :

- m *Triggered* in the simulation cycle in which its *sensitivity\_list* or *wait* statement conditions are satisfied
- m Execute on the last delta cycle of the corresponding simulation time
- m May not generate additional delta cycles in its execution

Copyright © 1995-1999 SCRA

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.

## Processes Revisited (Cont.)

- | **Concurrent procedure call equivalent to process containing a corresponding procedure call**

```
[call_label :] [POSTPONED] procedure_call;
```

- | **Concurrent assertion statement equivalent to a passive process containing a corresponding assertion statement**

```
[assert_label :] [POSTPONED] assertion;
```

- | **Concurrent signal assignments may also be postponed**

```
[label :] [POSTPONED] signal_assignment;
```

Copyright © 1995-1999 SCRA

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.

# Signal Assignment Statements Revisited

## | Signal assignment statement syntax :

```
[label :] target <= [delay_mechanism] waveform;
```

m **Delay\_mechanism** is either :

q **Transport**

è All input events reflected on output

q **REJECT *time\_expression* INERTIAL**

è Used to model component inertia so that short pulses on input signals do not affect the target output

è Default delay\_mechanism if none is specified

-- Default condition further specifies that the provided propagation delay be used for both the REJECT and INERTIAL delays in the assignment

Copyright © 1995-1999 SCRA

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.



## Signal Assignment Statements Revisited (Cont.)

### | Concurrent signal assignment syntax:

```
[label :] [POSTPONED] [GUARDED] conditional_signal_assignment
| [label :] [POSTPONED] [GUARDED] selected_signal_assignment
```

m **A *postponed* concurrent signal assignment statement is equivalent to a one line *postponed* process**

m **Example conditional signal assignment statement :**

```
S3 <= 0 AFTER 2 ns WHEN (x='0' and y='0') ELSE
      1 AFTER 5 ns WHEN (x='1' and y='1') ELSE
      2 AFTER 8 ns;
```

m **Example selected signal assignment statement :**

```
WITH sel_signal SELECT
S3 <= 0 AFTER 3 ns WHEN 0,
      1 AFTER 4 ns WHEN 3,
      2 AFTER 5 ns WHEN OTHERS;
```

m **UNAFFFECTED may be used as the assignment value**

q **No event is assigned to output -- new for VHDL93**

Copyright © 1995-1999 SCRA

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.  $A \leq B$ ;) 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 UNAFFFECTED 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.

## Signal Assignment Statements Revisited (Cont.)

### | Concurrent signal assignment : (cont.)

m If the target of the assignment is a signal of kind *bus* or *register*, it is a *guarded* target -- available inside *blocks*

q If the keyword **GUARDED** appears in the signal assignment statement, there are two possibilities for the assignment semantics :

è For guarded targets :

```
if GUARD then
  signal_transform
else
  disconnect_statements
end if;
```

è For non-guarded targets :

```
if GUARD then
  signal_transform
end if;
```

Copyright © 1995-1999 SCRA

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

Copyright © 1995-1999 SCRA

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

## Named Associations

### | Any index or parameter can be associated by position or by name

m Assignments to elements in records in arrays can use “|” and “OTHERS” :

```
TYPE array_ex IS ARRAY (1 TO 3) OF INTEGER;

VARIABLE var_pos : array_ex := (12,34,5);
VARIABLE var_nam1 : array_ex := (3=>23,2=>14,1=>8);
VARIABLE var_nam2 : array_ex := (1|3=>11,OTHERS=>15);
```

m Port map and generic map associations can use “OPEN” :

```
ENTITY dff is
  PORT(d,clk,enable : IN level;
        qn,q : OUT level);
END dff;
```

```
r0 : ENTITY work.dff(behav)
  PORT MAP (d0,clk,q=>q0,qn=>OPEN,enable=>enabled);
```

Copyright © 1995-1999 SCRA

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.



## Shared Variables



- | **In VHDL87, the scope of a variable was limited to the process in which it was declared**
  - m **Signals were the only means of inter-process communication**
- | **VHDL93 introduced the *shared variable***
  - m **Available to many processes or procedures**
- | **Shared variables are useful for system level modeling or object-oriented programming**
  - m **Shared variables also introduce some non-determinism in VHDL, limiting the uses of this new construct**

Copyright © 1995-1999 SCRA

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 if two or more processes make assignments in the same simulation cycle.

[Bergé93]

## Shared Variables Non-determinism

```
ARCHITECTURE non_determinist OF example IS
  SHARED VARIABLE count : INTEGER;
BEGIN
  p1 : PROCESS
  BEGIN
    count := 1;
    WAIT;
  END PROCESS p1;

  p2 : PROCESS
  BEGIN
    count := 2;
    WAIT;
  END PROCESS p2;
END non_determinist;
```

| **The final value of count is unpredictable**

Copyright © 1995-1999 SCRA

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]

## Shared Variables Stack Example

- | A shared variable is best used for system level modeling and object-oriented programming
- | The following `stack_of_integer` package uses shared variables to make the stack available to more than one procedure

```
PACKAGE stack_of_integer IS  
  PROCEDURE push (what : IN INTEGER);  
  PROCEDURE pop (what : OUT INTEGER);  
END stack_of_integer;
```

Copyright © 1995-1999 SCRA

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]

## Shared Variables Stack Example

```
PACKAGE BODY stack_of_integer IS
  TYPE stack_type IS ARRAY (0 TO 100) OF INTEGER;
  SHARED VARIABLE stack : stack_type;
  SHARED VARIABLE index : NATURAL := 0;

  PROCEDURE push (what : IN INTEGER) IS
  BEGIN
    stack(index) := what;
    index := index + 1;
  END push;

  PROCEDURE pop (what : OUT INTEGER) IS
  BEGIN
    index := index - 1;
    what := stack(index);
  END pop;
END stack_of_integer;
```

Copyright © 1995-1999 SCRA

[Bergé93]





# Outline



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

Copyright © 1995-1999 SCRA



## Abstract Data Type Example



- | **A first example will be the implementation of an abstract data type (ADT) in VHDL**
- | **An abstract data type consists of two things**
  - m The custom VHDL data types and subtypes
  - m Operators that manipulate data of those custom types
- | **Examples of ADTs include :**
  - m Queue data type
  - m Finite state machine data type
  - m Floating and complex data type
  - m Vector and matrix data types

Copyright © 1995-1999 SCRA

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.

## Abstract Data Types

### An Example Package Declaration

```
PACKAGE complex_type IS

  CONSTANT re : INTEGER := 0;

  CONSTANT im : INTEGER := 1;

  TYPE complex IS ARRAY (NATURAL RANGE re TO im) OF REAL;

  FUNCTION "+" (a, b : complex) RETURN complex;
  FUNCTION "-" (a, b : complex) RETURN complex;
  FUNCTION "*" (a, b : complex) RETURN complex;
  FUNCTION "/" (a, b : complex) RETURN complex;

END complex_type;
```

Copyright © 1995-1999 SCRA

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.

## Abstract Data Types An Example Package Body

```
PACKAGE BODY complex_type IS

  FUNCTION "+" (a, b : complex)
    RETURN complex IS

    VARIABLE t : complex;
    Begin

      T(re) := a(re) + b(re);
      T(im) := a(im) + b(im);
      RETURN t;

    End "+";

  FUNCTION "-" (a, b : complex)
    RETURN complex IS

    VARIABLE t : complex;
    Begin

      T(re) := a(re) - b(re);
      T(im) := a(im) - b(im);
      RETURN t;

    End "-";
```

```
  FUNCTION "*" (a, b : complex) RETURN complex
    IS

    VARIABLE t : complex;
    BEGIN

      t(re) := a(re) * b(re) - a(im) * b(im);
      t(im) := a(re) * b(im) + b(re) * a(im);
      RETURN t;

    END "*";

  FUNCTION "/" (a, b : complex) RETURN
    complex IS

    VARIABLE i : real;
    VARIABLE t : complex;
    BEGIN

      t(re) := a(re) * b(re) + a(im) * b(im);
      t(im) := b(re) * a(im) - a(re) * b(im);
      i := b(re)**2 + b(im)**2;
      t(re) := t(re) / i;
      t(im) := t(im) / i;
      RETURN t;

    END "/";
END complex_type;
```

Copyright © 1995-1999 SCRA

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



## Example From UVA ADEPT



- | **The following example is based on the performance and reliability modeling tool, ADEPT, developed at the University of Virginia**
  - m **Note that the implementations of the modules shown here are greatly simplified subsets of those actually used in ADEPT**
- | **Some particularly useful features of this example :**
  - m **A complex bus resolution function is used to achieve an embedded fully interlocked handshake protocol between communicating components**
  - m **VHDL procedures and functions are used extensively to hide the implementation details of the underlying behavior**

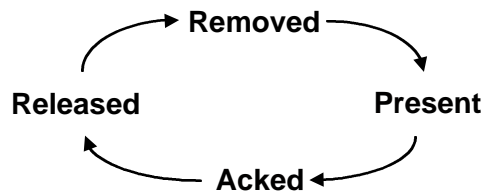
Copyright © 1995-1999 SCRA

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.

## Example From UVA ADEPT Bus Resolution Function

| The token status priority used in the protocol bus resolution function is illustrated below :

- m Note that the positions of the four arrows represent the four states in which the protocol token inputs may be
- m For each of the four input conditions, the token with the status at the head of the arrow is selected
- m Note that the cycle indicated by the illustration also shows the order of the status at the output of protocol



Copyright © 1995-1999 SCRA

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 *Ackerd* 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.



## Example From UVA ADEPT UVA Package Declaration



```
PACKAGE uva IS

  TYPE Handshake IS (Removed, Acked, Released, Present);
  TYPE Token_Fields IS (Status, Tag1, Tag2, Tag3,
                        Index, Act_Time, Color);

  TYPE Color_Type IS
    ARRAY(Token_Fields RANGE Tag1 TO Act_Time) OF INTEGER;

  TYPE Token IS
    RECORD
      Status : Handshake;
      Color : Color_Type;
    END RECORD;

  TYPE Token_Vector IS ARRAY (Integer RANGE <> ) OF Token;
  FUNCTION Protocol (Input : Token_Vector) RETURN Token;
  SUBTYPE Token_Res IS Protocol Token;

  CONSTANT Def_Colors : Color_Type := (OTHERS=>0);
  CONSTANT Def_Source-Token : Token := (Released, Def_Colors);
  CONSTANT Def_Sink-Token : Token := (Removed, Def_Colors);

  -- Package declaration continued on next slide
```

Copyright © 1995-1999 SCRA

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]

## Example From UVA ADEPT UVA Package Declaration (Cont.)

```
-- Package declaration continued from previous slide

PROCEDURE Place-Token(SIGNAL T : INOUT Token);
PROCEDURE Place-Token(SIGNAL T : INOUT Token; Delay : TIME);
PROCEDURE Ack-Token(SIGNAL T : INOUT Token);
PROCEDURE Release-Token(SIGNAL T : INOUT Token);
PROCEDURE Remove-Token(SIGNAL T : INOUT Token);

function Token_Present(T : Token) RETURN BOOLEAN;
function Token_Acked(T : Token) RETURN BOOLEAN;
function Token_Released(T : Token) RETURN BOOLEAN;
function Token_Removed(T : Token) RETURN BOOLEAN;

END uva;
```

Copyright © 1995-1999 SCRA

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



## Example From UVA ADEPT UVA Package Body

```
PACKAGE BODY uva IS

FUNCTION Protocol (Input : Token_Vector) RETURN Token IS
    VARIABLE Source-Token : Token := Def_Source-Token;
    VARIABLE Sink-Token : Token := Def_Sink-Token;
    VARIABLE I : INTEGER;
BEGIN
    -- First, determine status of input tokens
    FOR I in Input'RANGE
        IF (Input(I).Status = Present) THEN Source-Token := Input(I);
        ELSIF (Input(I).Status = Acked) THEN Sink-Token := Input(I);
        END IF; -- else use default assignments from variable declarations
    END loop;

    -- Resolve based on status of tokens identified
    IF (Source-Token.Status=Present) THEN
        IF (Sink-Token.Status=Acked) THEN RETURN Sink-Token;
        ELSE RETURN Source-Token;
        END IF;
    ELSIF (Sink-Token.Status=Acked) THEN RETURN Source-Token;
    ELSE RETURN Sink-Token;
    END IF;

END Protocol;

-- Package body continued on next slide
```

Copyright © 1995-1999 SCRA

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
```

Copyright © 1995-1999 SCRA

## Example From UVA ADEPT UVA Package Body (Cont.)

```
-- Token body continued from previous slide

PROCEDURE Release-Token(SIGNAL T : INOUT Token) IS
VARIABLE Temp : Token;
BEGIN
    Temp := T;
    Temp.Status := Released;
    T <= Temp;
END Release-Token;

PROCEDURE Remove-Token(SIGNAL T : INOUT Token) IS
VARIABLE Temp : Token;
BEGIN
    Temp := T;
    Temp.Status := Removed;
    T <= Temp;
END Remove-Token;

-- Token body continued on next slide
```

Copyright © 1995-1999 SCRA

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;
```

Copyright © 1995-1999 SCRA

## Simple Module Examples Source Module

```

LIBRARY uvalib;
USE uvalib.uva.ALL;

ENTITY Source IS
  GENERIC (Step : TIME);
  PORT (Data_Output : INOUT Token;
END Source;

Architecture Ar_Source OF Source IS
BEGIN
  PROCESS
  BEGIN
    Place-Token(Data_Output);
    WAIT UNTIL Token_Acked(Data_Output);

    Release-Token(Data_Output);
    WAIT UNTIL Token_Removed(Data_Output);

    WAIT FOR Step;
  END PROCESS;
END Ar_Source;

```

Copyright © 1995-1999 SCRA

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]

## Simple Module Examples

### Fixed\_delay Module

```

LIBRARY uvalib;
USE uvalib.uva.ALL;

ENTITY FD IS
  GENERIC (Delay : Time);
  PORT (Data_Input, Data_Output : INOUT Token;
END FD;

Architecture Ar_FD of FD IS
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL Token_Present(Data_Input) AND Token_Removed(Data_Output);

    Place_Token(Data_Output, Delay); -- Note use of overloaded procedure
    WAIT UNTIL Token_Acked(Data_Output);

    Ack_Token(Data_Input);
    Release_Token(Data_Output);
    WAIT UNTIL Token_Released(Data_Input);

    Remove_Token(Data_Input);
  END PROCESS;
END Ar_FD;

```

Copyright © 1995-1999 SCRA

A simplified version of the Fixed\_Delay module from 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]

## Simple Module Examples

### Sink Module

```
LIBRARY uvalib;  
USE uvalib.uva.ALL;  
  
ENTITY Sink IS  
  PORT (Data_Input : INOUT Token;  
END Source;  
  
ARCHITECTURE Ar_Sink OF Sink IS  
BEGIN  
  PROCESS  
  BEGIN  
    WAIT UNTIL Token_Present(Data_Input);  
  
    Ack_Token(Data_Input);  
    WAIT UNTIL Token_Released(Data_Input);  
  
    Remove_Token(Data_Input);  
  END PROCESS;  
END Ar_Sink;
```

Copyright © 1995-1999 SCRA

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]

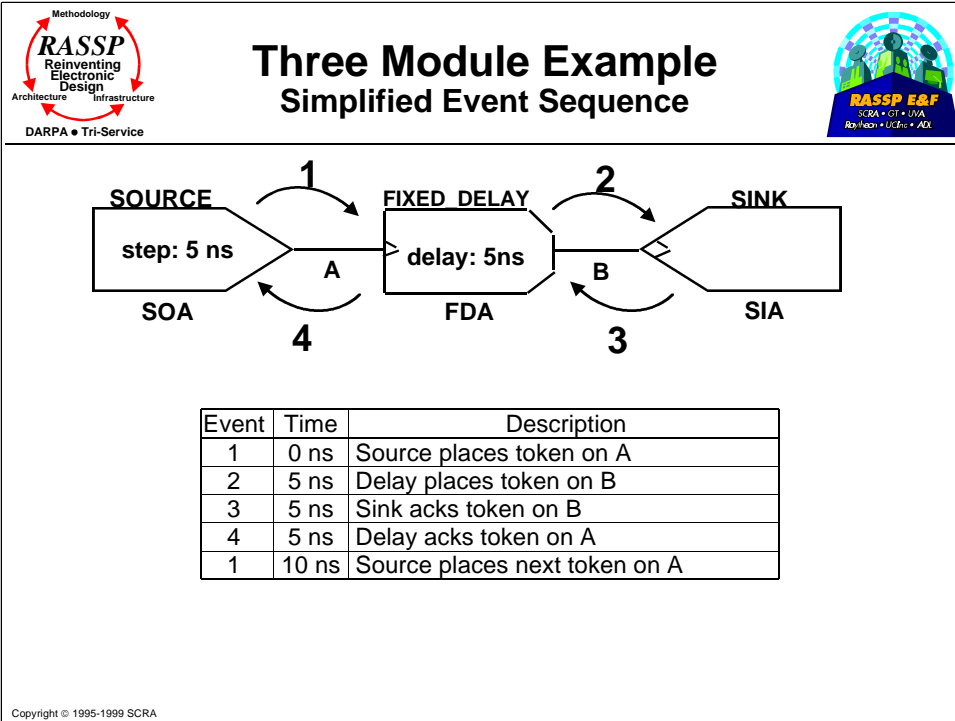
## Three Module Example Testbench Description

```
LIBRARY uvalib;  
USE uvalib.uva.ALL;  
  
ENTITY Test IS  
END;  
  
ARCHITECTURE Ar_Test OF Test IS  
  SIGNAL A,B : Token_Res;  
  BEGIN  
    m0 : ENTITY work.Source(Ar_Source)  
      GENERIC MAP (5ns)  
      PORT MAP (A);  
    m1 : ENTITY work.FD(Ar_FD)  
      GENERIC MAP (5ns)  
      PORT MAP (A,B);  
    m2 : ENTITY work.Sink(Ar_Sink)  
      PORT MAP (B);  
  END Ar_Test;
```

Copyright © 1995-1999 SCRA

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.

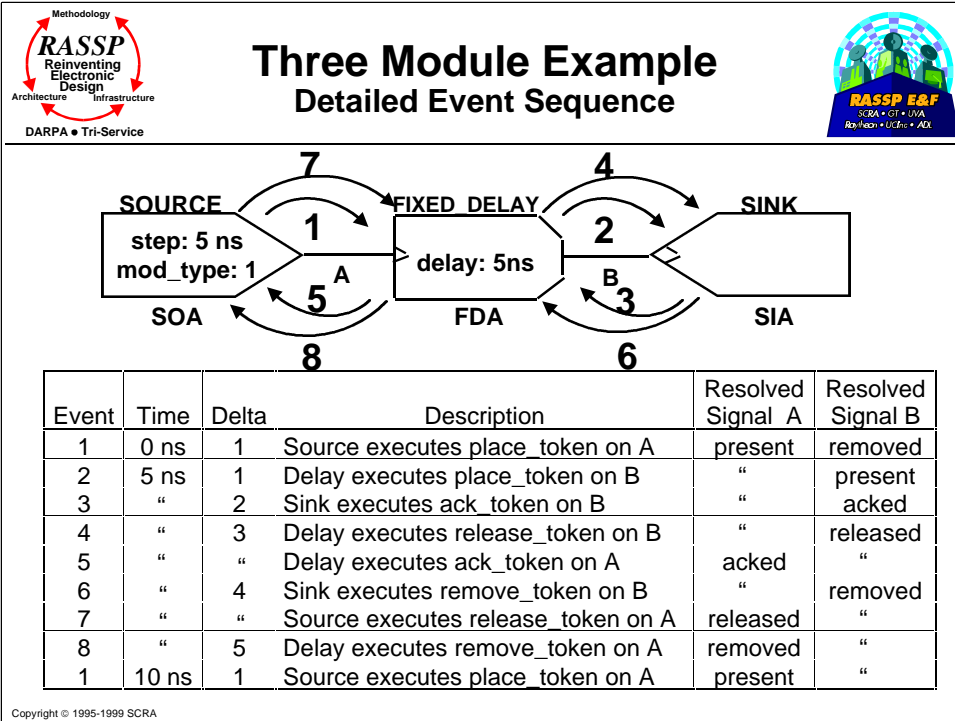




The three-module example shown in this slide will be used to illustrate the important events in the passing of *tokens* from a *token* source to a *token* sink. As the table shows, the relevant events are the placing and acknowledging of *tokens*. The other two states in the four-cycle handshake, releasing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



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**

Copyright © 1995-1999 SCRA



## Summary



- | **VHDL provides sophisticated constructs making it a versatile description language for modeling of hardware structure and behavior, e.G. :**
  - m **Bus resolution functions allow for user defined bus arbitration**
  - m **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**
  - m **These modules are introductory in nature and are not intended to provide a complete and comprehensive coverage of VHDL**
  - m **The contents of these modules, however, provide enough information to allow a designer new to VHDL to successfully describe complex systems with VHDL**

Copyright © 1995-1999 SCRA

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., Gadiant, A., Frank, G., eds. *Rapid Prototyping of Application Specific Signal Processors*, Kluwer Academic Publishers, Norwell, MA, 1997

Copyright © 1995-1999 SCRA