



Structural VHDL

RASSP Education & Facilitation

Module 11

Version 3.00

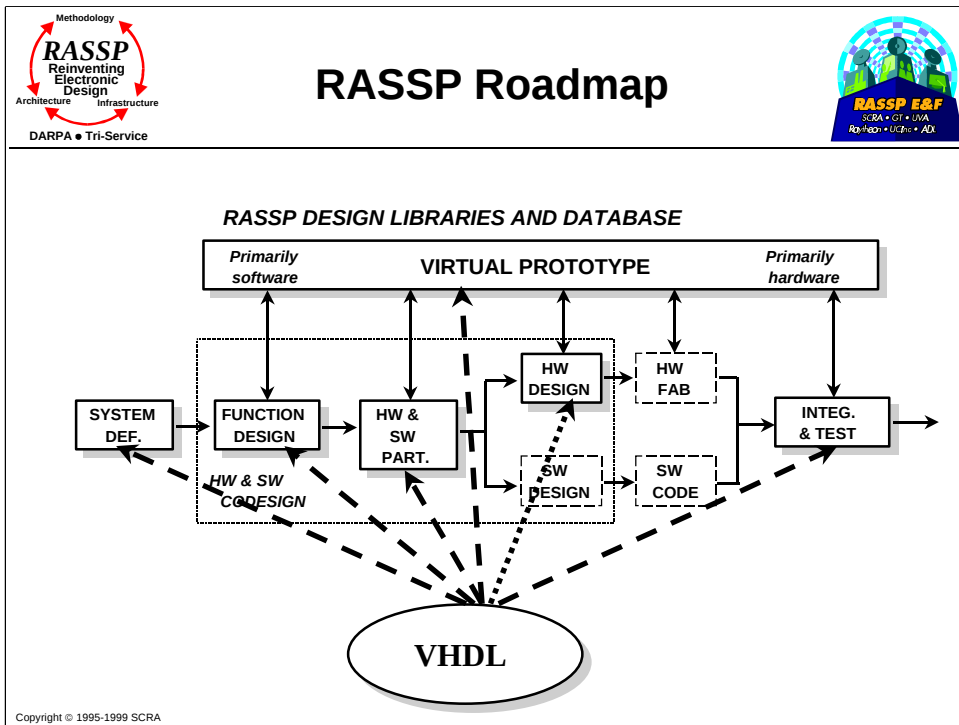
Copyright © 1995-1999 SCRA

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

Structural VHDL allows the designer to represent a system in terms of components and their interconnections. This module discusses the constructs available in VHDL to facilitate structural descriptions of designs.



This diagram emphasizes the role of VHDL in the RASSP program. VHDL can be used for system definition, functional design, hardware-software partitioning, hardware design and hardware-software integration and test. In RASSP, the concept of virtual prototyping uses VHDL as the binding language of choice for all design paradigms.

The most common usage of VHDL prior to RASSP was in the area of hardware design. The RASSP program has extended VHDL's use to include executable requirements, performance modeling/system level design as well as system integration and test.



Module Goals



I Introduce structural VHDL constructs

- m Use of *components*
- m Component binding indications
- m Use of configuration declarations
- m GENERATE statements

Copyright © 1995-1999 SCRA

The goals of this module are to introduce the concepts and constructs supporting structural modeling using VHDL. These include the mechanisms for incorporating other VHDL design objects into an architecture description. In addition, some powerful VHDL utilities that facilitate the design of systems with regular structures and constructs to support configuration control will be presented. The goal of this module is to bring the student to the point where she/he will be able to write code using the concepts of structural design in VHDL.

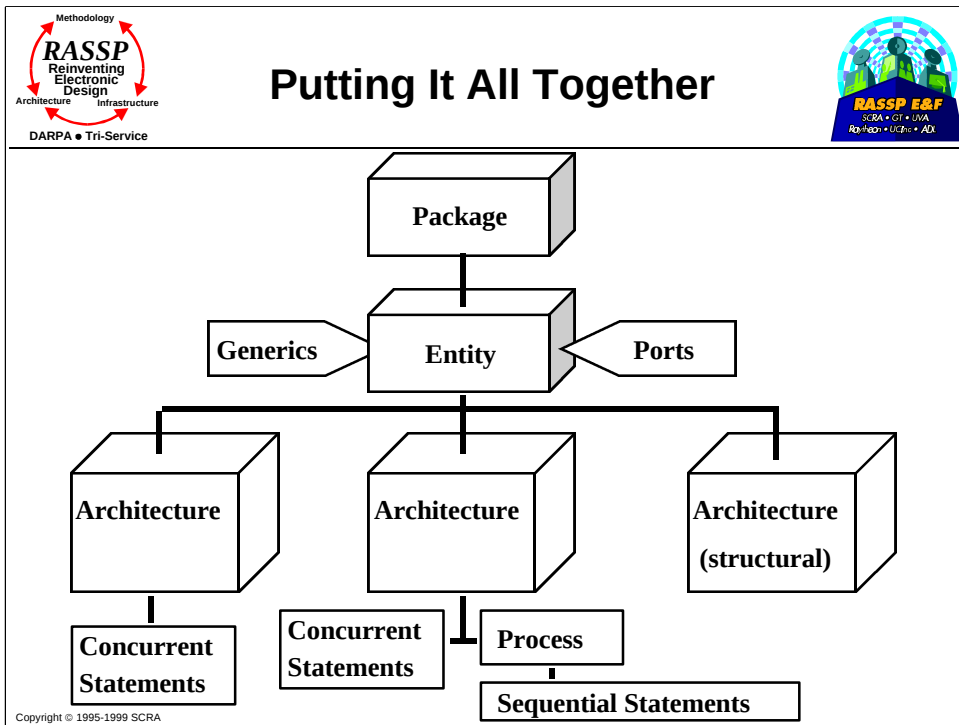


Module Outline



- | **Introduction**
- | **Incorporating VHDL Design Objects**
- | **Generate Statement**
- | **Examples**
- | **Summary**

Copyright © 1995-1999 SCRA



This figure captures the main features of a complete VHDL model. A single component model is composed of one entity and one or many architectures. The entity represents the interface specification (I/O) of the component. It defines the components external view, sometimes referred to as its "pins".

The architecture(s) describe the function or composition of an entity. There are three general types of architectures. One type of architecture describes the structure of the design (right hand side) in terms of its sub-components and their interconnections. A key item of a structural VHDL architecture is the "*binding* statement" which associates the entity of a sub-component to one of the possible several alternative architectures for that component.

A second type of architecture, containing only concurrent statements, is commonly referred to as a dataflow description (left hand side). Concurrent statements execute when data is available on their inputs. These statements can occur in any order within the architecture.

The third type of architecture is the behavioral description in which the functional and possibly timing characteristics are described using VHDL concurrent statements and processes. The process is a concurrent statement of an architecture. All statements contained within a process execute in a sequential order until it gets suspended by a wait statement.

Packages are used to provide a collection of common declarations, constants, and/or subprograms to entities and architectures.

Generics provide a method to communicate static information to an architecture from the external environment. They are passed through the entity construct.

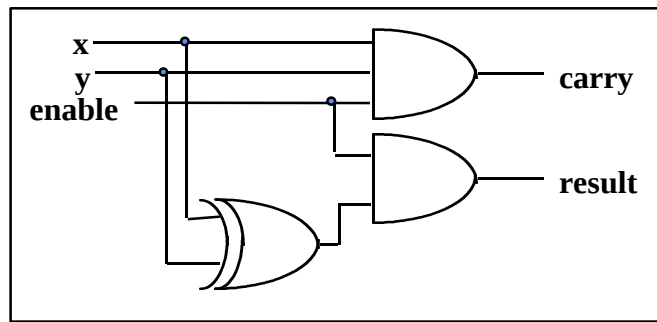
Ports provide the mechanism for a device to communication with its environment. A port declaration defines the names, types, directions, and possible default values for the signals in a component's interface.

Implicit in this figure is the testbench which is the top level of a self-contained simulatable model. The testbench is a special VHDL object for which the entity has no signals in its port declaration. Its architecture often contains construct from all three of the types described above. Structural VHDL concepts are used to connect the model's various components together, Dataflow and behavior concepts are often used to provide the simulation's start stop conditions, or other desired modeling directives.

[MG93]

Introduction

- | **Models can be constructed by interconnecting subcomponents**
 - m **A *structural* model lists the required subcomponents and prescribes their interconnections**
 - m **Akin to a design schematic :**



Copyright © 1995-1999 SCRA

Structural VHDL allows a designer to describe a model in terms of sub-components and their interconnections. In this figure, simple logic elements are used to design a full adder. A structural description views the hardware as a netlist or schematic of the device; the components and interconnections are visible, but the internal functions are hidden.



Module Outline



- | Introduction
- | **Incorporating VHDL Design Objects**
- | Generate Statement
- | Examples
- | Summary

Copyright © 1995-1999 SCRA



Mechanisms for Incorporating VHDL Design Objects



- | **VHDL mechanisms to incorporate design objects**
 - m Using direct instantiation (not available prior to VHDL-93)
 - m Using *component* declarations and instantiations
 - q Create idealized local *components* (i.e. declarations) and connect them to local signals (i.e. instantiations)
 - q *Component* instantiations are then bound to VHDL design objects either :
 - è Locally -- within the architecture declaring the component
 - è At higher levels of design hierarchy, via *configurations*
- | **Consider structural descriptions for the following entity :**

```
USE work.resources.all;  
  
ENTITY reg4 IS -- 4-bit register with no enable  
  GENERIC(tprop : delay := 8 ns;  
          tsu   : delay := 2 ns);  
  PORT(d0,d1,d2,d3 : IN level;  
        clk : IN level;  
        q0,q1,q2,q3 : OUT level);  
END reg4;
```

Copyright © 1995-1999 SCRA

There are a number of constructs available in VHDL to incorporate design objects into architecture descriptions. The simplest, but least versatile, is the direct instantiation of a VHDL *entity*. With this method, the details of the entity's interface must be known and cannot be customized at instantiation.

The other two mechanisms use locally declared *components* to define idealized element interfaces. A *component* is then plugged into the architecture description by connecting signals visible in the architecture to the interface of the component in an *instantiation* statement. The two mechanisms here differ in how a component is bound to an existing VHDL design object. One mechanism binds the component to an existing VHDL entity/architecture object within the architecture description in which the component was instantiated. The second mechanism defers the binding until higher levels in the design hierarchy via the use of *configurations* which are introduced in this section.



4-Bit Register as Running Example



I First, need to find the building block(s)

m Reusing an object from examples in Module 10

```
USE work.resources.all;

ENTITY dff IS

    GENERIC(tprop : delay := 8 ns;
            tsu   : delay := 2 ns);

    PORT(d      : IN level;
         clk    : IN level;
         enable : IN level;
         q      : OUT level;
         qn     : OUT level);

END dff;
```

```
ARCHITECTURE behav OF dff IS
BEGIN
    one : PROCESS (clk)
    BEGIN
        -- first, check for rising clock edge
        -- and check that ff is enabled
        IF ((clk = '1' AND clk'LAST_VALUE = '0')
            AND enable = '1') THEN
            -- now check setup requirement is met
            IF (d'STABLE(tsu)) THEN
                -- now check for valid input data
                IF (d = '0') THEN
                    q <= '0' AFTER tprop;
                    qn <= '1' AFTER tprop;
                ELSIF (d = '1') THEN
                    q <= '1' AFTER tprop;
                    qn <= '0' AFTER tprop;
                ELSE -- else invalid data
                    q <= 'X';
                    qn <= 'X';
                END IF;
            ELSE -- else setup not met
                q <= 'X';
                qn <= 'X';
            END IF;
        END IF;
    END PROCESS one;
END behav;
```

Copyright © 1995-1999 SCRA

As a running example, we will build a 4-bit register using the D flip-flop model presented in Module 10, the Basic VHDL Module.



General Steps to Incorporate VHDL Design Objects



- | A VHDL design object to be incorporated into an architecture must *generally* be :
 - m declared -- where a local interface is defined
 - m instantiated -- where local signals are connected to the local interface
 - q Regular structures can be created easily using *GENERATE* statements in component instantiations
 - m bound -- where an entity/architecture object which implements it is selected for the instantiated object

Copyright © 1995-1999 SCRA

The three steps shown above illustrate the general requirements for incorporating design objects. It is important to note that the *direct instantiation* method illustrated in slide 18 skips the declaration of a local *component* and combines the instantiation and binding into a single statement.

Also note that binding may be postponed to higher levels in the design hierarchy to provide flexibility in the selection of design objects to be incorporated.

Using Component Declarations and Local Bindings

- | Component declarations define interfaces for idealized local objects
 - m Component declarations may be placed in architecture declarations or in package declarations
- | Component instantiations connect local signals to component interface signals

```
USE work.resources.all;

ARCHITECTURE struct_2 OF reg4 IS
  COMPONENT reg1 IS
    PORT (d, clk : IN level;
          q : OUT level);
  END COMPONENT reg1;
  CONSTANT enabled : level := '1';
  FOR ALL : reg1 USE work.dff(behav)
    PORT MAP(d=>d, clk=>clk, enable=>enabled, q=>q, qn=>OPEN);
  BEGIN
    r0 : reg1 PORT MAP (d=>d0, clk=>clk, q=>q0);
    r1 : reg1 PORT MAP (d=>d1, clk=>clk, q=>q1);
    r2 : reg1 PORT MAP (d=>d2, clk=>clk, q=>q2);
    r3 : reg1 PORT MAP (d=>d3, clk=>clk, q=>q3);
  END struct_2;
```

Copyright © 1995-1999 SCRA

This example shows the three steps listed earlier:

- 1) A *component declaration* defines the interface to an idealized local component. Note that the component declaration may be placed in a package declaration and made visible to the architecture via a USE clause.
- 2) A binding indication assigns a VHDL entity/architecture object to component instances. In this case, all *reg1* components will use the *behav* architecture description for the *dff* entity in the *work* library.

*Note: The idealized component *reg1* uses a subset of the port signals of the work library element *DFF*. The port signal *enable* is tied to the locally declared constant *enabled*, which is set to the value of '1'.

- 3) Instantiation statements create copies of the component to be plugged into the architecture description by connecting local signals to signals in the component interface.



Using Component Declarations and Configurations



```

USE work.resources.all;

ARCHITECTURE struct_3 OF reg4 IS
  COMPONENT reg1 IS
    PORT (d, clk : IN level;
          q : OUT level);
  END COMPONENT reg1;
  BEGIN
    r0 : reg1 PORT MAP (d=>d0,clk=>clk,q=>q0);
    r1 : reg1 PORT MAP (d=>d1,clk=>clk,q=>q1);
    r2 : reg1 PORT MAP (d=>d2,clk=>clk,q=>q2);
    r3 : reg1 PORT MAP (d=>d3,clk=>clk,q=>q3);
  END struct_3;

```

```

USE work.resources.all;

CONFIGURATION reg4_conf_1 OF reg4 IS
  CONSTANT enabled : level := '1';
  FOR struct_3
    FOR all : reg1 USE work.dff(behav)
      PORT MAP(d=>d,clk=>clk,enable=>enabled,q=>q,qn=>OPEN);
    END FOR;
  END FOR;
END reg4_conf_1;

```

```

-- Architecture in which a COMPONENT for reg4 is declared
...
FOR ALL : reg4_comp USE CONFIGURATION work.reg4_conf_1;
...

```

Copyright © 1995-1999 SCRA

Note that in this example, three separate VHDL files are used. The first file above shows the architecture description in which the *reg1* component is declared and instantiated.

The second file shows a configuration declaration in which the *reg1* components in the *struct_3* architecture of entity *reg4* are bound to *dff(behav)*.

The third example shows a small excerpt from an architecture description in which a locally visible component named *reg4_comp* is bound to a VHDL design object via the configuration declaration *reg4_conf_1* found in the *work* library (i.e. the configuration declaration shown in the middle section of this slide).

Note that the use of configurations to defer the binding of components adds flexibility to structural architecture descriptions by allowing alternative architecture to be plugged in to the design easily.



Power of Configuration Declarations



- | **Reasons to use *configuration declarations* :**
 - m Large design may span multiple levels of hierarchy
 - m When the architecture is developed, only the component interface may be available
 - m Mechanism to put the pieces of the design together
- | **Configurations can be used to customize the use of VHDL design objects interfaces as needed :**
 - m Entity name can be different than the component name
 - m Entity of incorporated design object may have more ports than the component declaration
 - m Ports on the entity declaration of the incorporated design object may have different names than the component declaration

Copyright © 1995-1999 SCRA

As indicated in the previous slide, configuration declarations provide a mechanism for replacing design objects in structural descriptions easily.

Similarly, they allow for structural descriptions to be developed before the entity/architecture building blocks have been finalized. This is particularly useful in a large system which may have been partitioned among several designers.

In addition, idealized components can be made to accommodate actual entity/architecture design object interface requirements in the configuration declaration. This may, for example, be used to assign generics and/or unused signals fixed values (e.g. enable signals to a constant ON value).

Generic Map

- | **Generics allow the component to be customized upon instantiation**
 - m Entity declaration of design object being incorporated provides default values
- | **The GENERIC MAP is similar to the PORT MAP in that it maps specific values to the generics of the component**

```
USE Work.my_stuff.ALL
ARCHITECTURE test OF test_entity
    SIGNAL S1, S2, S3 : BIT;
BEGIN
    Gate1 : my_stuff.and_gate -- component found in package
        GENERIC MAP (tph=>2 ns, tphl=>3 ns)
        PORT MAP (S1, S2, S3);
END test;
```

Copyright © 1995-1999 SCRA

Generics are mapped in a fashion similar to ports. If no default values are assigned in the design object's ENTITY declaration, a GENERIC MAP must be provided in the component's declaration, instantiation, or binding.

As in PORT MAP signal associations, associations may be made by position or by name.



Component Binding Specifications



| A component binding specification provides binding information for instantiated components

m Single component

```
FOR A1 : and_gate USE binding_indication;
```

m Multiple components

```
FOR A1, A2 : and_gate USE binding_indication;
```

m All components

```
FOR ALL : and_gate USE binding_indication;
```

-- All components of this type are affected

m Other components

```
FOR OTHERS : and_gate USE binding_indication;
```

-- i.e. for components that are not otherwise specified

Copyright © 1995-1999 SCRA

Unfortunately, component binding specifications are referred to as “configuration specifications” in the VHDL Language Reference Manual, but the term is avoided here to prevent confusion with *configuration* descriptions.

The component specification can be of several forms, and this slide shows examples for various types. The component specification identifies those components to be configured by name or by the keyword ALL. The keyword OTHERS selects all components not yet configured.

Binding Indication

- | The ***binding indication*** identifies the design object to be used for the component

- | Two mechanisms available :

- m VHDL entity/architecture design object

```
FOR ALL : reg1 USE work.dff(behav);
```

- m VHDL configuration

```
FOR reg4_inst : reg4_comp USE CONFIGURATION work.reg4_conf_1;
```

- | Binding indication may also include a PORT MAP and/or GENERIC MAP to customize the component(s)

Copyright © 1995-1999 SCRA

The binding indication identifies the design entity (i.e. entity/architecture object or configuration declaration) to bind with the component and maps the two interfaces together. That is, binding indication associates component instances with a particular design entity. The binding indication may include a PORT MAP and GENERIC MAP to adapt the interfaces of the entity and the component.

Using Direct Instantiation

- | Provides one-step mechanism for plugging in previously defined VHDL design objects
- | Only available one level up in hierarchy from level of incorporated building block(s)

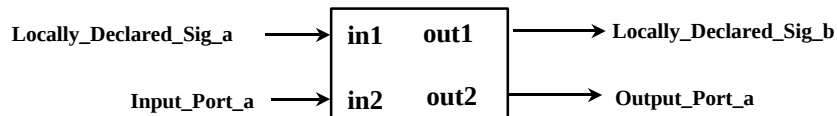
```
USE work.resources.all;  
  
ARCHITECTURE struct_1 OF reg4 IS  
  CONSTANT enabled : level := '1';  
  BEGIN  
    r0 : ENTITY work.dff(behav)  
      PORT MAP (d0,clk,enabled,q0,OPEN);  
    r1 : ENTITY work.dff(behav)  
      PORT MAP (d1,clk,enabled,q1,OPEN);  
    r2 : ENTITY work.dff(behav)  
      PORT MAP (d2,clk,enabled,q2,OPEN);  
    r3 : ENTITY work.dff(behav)  
      PORT MAP (d3,clk,enabled,q3,OPEN);  
  END struct_1;
```

Copyright © 1995-1999 SCRA

The *direct instantiation* method was introduced in VHDL-93. It allows a VHDL design object to be plugged in directly to an architecture's description by connecting local signals to its interface. This mechanism does not require the use of an idealized component to be declared, instantiated, and bound. Rather, a VHDL entity/architecture object may be inserted into an architecture description in one step.

Rules for Actuals and Locals

- | **An *actual* is either a signal declared within the architecture or a port in the entity declaration**
 - m A port on a *component* is known as a *local* and must be matched with a compatible *actual*
- | **VHDL has two main restrictions on the association of *locals* with *actuals***
 - m Local and actual must be of same data type
 - m Local and actual must be of compatible modes
 - q Locally declared signals do not have an associated mode and can connect to a local port of any mode



Copyright © 1995-1999 SCRA

Locals are defined as the ports of the *component*, and actuals are the signals visible within an architecture. VHDL has two restrictions on the association of locals with actuals.

- 1) The local and actual must be of the same data type.
- 2) The local and actual must be of compatible modes. An actual of mode IN (i.e. a PORT of mode IN since locally declared signals do not have a mode) can only be associated with a local of mode IN, and an actual of mode OUT (i.e. a PORT of mode OUT) can only be associated with a local of mode OUT. A local INOUT port is generally associated with an INOUT or OUT actual. Locally declared signals can be connected to locals of any mode, but care must be exercised to avoid illegal connections (e.g. a single actual connected to two mode OUT locals).



Summary of Concepts of Structural VHDL



- | **Various levels of abstraction supported in description of VHDL structural models**
 - m **Direct instantiation** requires detailed knowledge of building blocks when they are incorporated
 - m **Use of *components*** allows definition and use of idealized local building blocks
 - q **Can define local interface** for component to be connected to local signals
 - q **Declared components bound to VHDL design objects (i.e. *entity/architecture* descriptions)**
 - è **Binding done either locally or deferred to higher levels in design hierarchy via use of configurations**
- | ***Actuals* and *locals* must be of compatible types and modes**

Copyright © 1995-1999 SCRA

In summary, structural VHDL is concerned with the interconnection and arrangement of components describing the contents of a design. The behavior of the underlying design objects, therefore, is not explicitly indicated. A structural description can be thought of as a physical netlist describing a hierarchical representation of a VHDL model.



Module Outline



- | Introduction
- | Incorporating VHDL Design Objects
- | **Generate Statement**
- | Examples
- | Summary

Copyright © 1995-1999 SCRA



Generate Statement



- | **VHDL provides the GENERATE statement to create well-patterned structures easily**
 - m Some structures in digital hardware are repetitive in nature (e.g. RAMs, adders)
- | **Any VHDL concurrent statement may be included in a GENERATE statement, including another GENERATE statement**
 - m Specifically, component instantiations may be made within GENERATE bodies

Copyright © 1995-1999 SCRA

Structural descriptions of large, but highly regular, structures can be tedious. A VHDL GENERATE statement can be used to include as many concurrent VHDL statements (e.g. component instantiation statements) as needed to describe a regular structure easily. In fact, a GENERATE statement may even include other GENERATE statements for more complex devices.. Some common examples include the instantiation and connection of multiple identical components such as half adders to make up a full adder, or exclusive or gates to create a parity tree.



Generate Statement FOR-Scheme



- | All objects created are similar
- | The **GENERATE** parameter must be discrete and is undefined outside the **GENERATE** statement
- | Loop cannot be terminated early

```
name : FOR parameter_specification GENERATE  
      [Declaration_statements  
BEGIN]  
      {concurrent_statements}  
END GENERATE [name];
```

Copyright © 1995-1999 SCRA

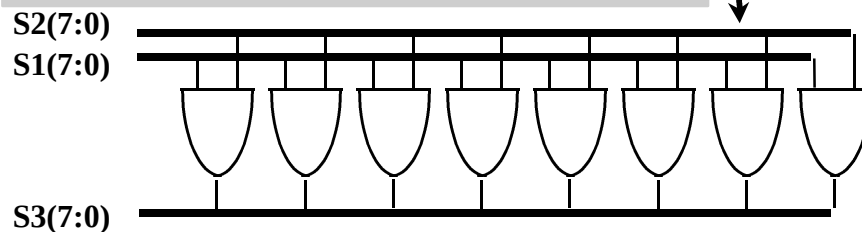
VHDL provides two different schemes of the **GENERATE** statement, the **FOR**-scheme and the **IF**-scheme. This slide shows the syntax for the **FOR**-scheme.

The **FOR**-scheme is reminiscent of a **FOR** loop used for sequence control in many programming languages. The **FOR**-scheme generates the included concurrent statements the assigned number of times. In the **FOR**-scheme, all of the generated concurrent statements must be the same. The loop variable is created in the **GENERATE** statement and is undefined outside that statement (i.e. it is not a variable or signal visible elsewhere in the architecture).

The syntax for the **FOR**-scheme **GENERATE** statement is shown in the slide. The loop variable in this case is **N**. The range can be any valid discrete range. After the **GENERATE** keyword, the concurrent statements to be generated are stated, and the **GENERATE** statement is closed with **END GENERATE**.

FOR-Scheme Example

```
-- this uses the and_gate component from before
ARCHITECTURE test_generate OF test_entity IS
    SIGNAL S1, S2, S3: BIT_VECTOR(7 DOWNTO 0);
BEGIN
    G1 : FOR N IN 7 DOWNTO 0 GENERATE
        and_array : and_gate
            GENERIC MAP (2 ns, 3 ns)
            PORT MAP (S1(N), S2(N), S3(N));
    END GENERATE G1;
END test_generate;
```



Copyright © 1995-1999 SCRA

This slide shows an example of the FOR-scheme. The code generates an array of AND gates. In this case, the GENERATE statement has been named G1 and instantiates an array of 8 and_gate components. The PORT MAP statement maps the interfaces of each of the 8 gates to specific elements of the S1, S2, and S3 vectors by using the FOR loop variable as an index.



Generate Statement IF-Scheme



- | **Allows for conditional creation of components**
- | **Cannot use ELSE or ELSIF clauses with the IF-scheme**

```
name : IF boolean_expression GENERATE  
      [Declaration_statements  
BEGIN]  
      {concurrent_statements}  
END GENERATE [name];
```

Copyright © 1995-1999 SCRA

The second form of the GENERATE statement is the IF-scheme. This scheme allows for conditional generation of concurrent statements. One obvious difference between this scheme and the FOR-scheme is that all the concurrent statements generated do not have to be the same. While this IF statement may seem reminiscent to the IF-THEN-ELSE constructs in programming languages, note that the GENERATE IF-scheme does not provide ELSE or ELSIF clauses.

The syntax of the IF-scheme GENERATE statement is shown in this slide. The boolean expression of the IF statement can be any valid boolean expression.

IF-Scheme Example

```

ARCHITECTURE test_generate OF test_entity
    SIGNAL S1, S2, S3: BIT_VECTOR(7 DOWNT0 0);
BEGIN
    G1 : FOR N IN 7 DOWNT0 0 GENERATE

        G2 : IF (N = 7) GENERATE
            or1 : or_gate
                GENERIC MAP (3 ns, 3 ns)
                PORT MAP (S1(N), S2(N), S3(N));
            END GENERATE G2;

        G3 : IF (N < 7) GENERATE
            and_array : and_gate
                GENERIC MAP (2 ns, 3 ns)
                PORT MAP (S1(N), S2(N), S3(N));
            END GENERATE G3;

    END GENERATE G1;
END test_generate;

```

Copyright © 1995-1999 SCRA

The example here uses the IF-scheme GENERATE statement to make a modification to the and_gate array such that the seventh gate of the array will be an or_gate.

Another example use of the IF-scheme GENERATE is in the conditional execution of timing checks. Timing checks can be incorporated inside a GENERATE IF-scheme. For example, the following statement can be used:

Check_time : IF TimingChecksOn GENERATE

This allows the boolean variable TimingChecksOn to enable timing checks by generating the appropriate concurrent VHDL statements in the description. This parameter can be set in a package or passed as a generic and can improve simulation speed by shutting off this computational section.



Module Outline



- | Introduction
- | Component Instantiation
- | Generate Statement
- | **Examples**
- | Summary

Copyright © 1995-1999 SCRA



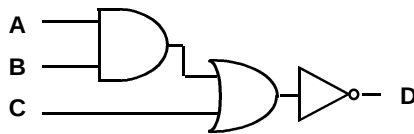
Examples



- | **Build higher level modules from the library of basic gates**
 - m AND-OR-Invert
 - m 8 Bit Register using DFFs
 - m 8 Bit Shift Register using Multiplexors and DFFs
- | **Use these modules to construct a datapath for unsigned 8 bit multiplication**

Copyright © 1995-1999 SCRA

Structural And-Or-Invert Gate Example (Entity)



```
LIBRARY gate_lib;
USE gate_lib.resources.all;

ENTITY aoi2_str IS

  GENERIC(trise : delay := 12 ns;
          tfall : delay := 9 ns);

  PORT(a : IN level;
        b : IN level;
        c : IN level;
        d : OUT level);

END aoi2_str;
```

Copyright © 1995-1999 SCRA

This is the schematic and the VHDL entity description of a simple and-or-invert gate.



Structural And-Or-Invert Gate Example (Architecture)



```

ARCHITECTURE structural OF aoi2_str IS

  -- COMPONENT DECLARATIONS
  COMPONENT and2
    GENERIC(trise : delay;
            tfall : delay);
    PORT(a : IN level;
          b : IN level;
          c : OUT level);
  END COMPONENT;

  COMPONENT or2
    GENERIC(trise : delay;
            tfall : delay);
    PORT(a : IN level;
          b : IN level;
          c : OUT level);
  END COMPONENT;

  COMPONENT inv
    GENERIC(trise : delay;
            tfall : delay);
    PORT(a : IN level;
          b : OUT level);
  END COMPONENT;

```

```

-- BINDING INDICATIONS
FOR ALL : and2 USE ENTITY gate_lib.and2(behav);
FOR ALL : or2  USE ENTITY gate_lib.or2(behav);
FOR ALL : inv  USE ENTITY gate_lib.inv(behav);

SIGNAL and_out : level; -- signal for output
                        -- of AND gate
SIGNAL or_out  : level; -- signal for output
                        -- of OR gate

BEGIN

  -- COMPONENT INSTANTIATIONS
  AND_1 : and2 GENERIC MAP(trise => trise,
                           tfall => tfall)
    PORT MAP(a => a, b => b,
              c => and_out);

  OR_1  : or2  GENERIC MAP(trise => trise,
                           tfall => tfall)
    PORT MAP(a => and_out, b => c,
              c => or_out);

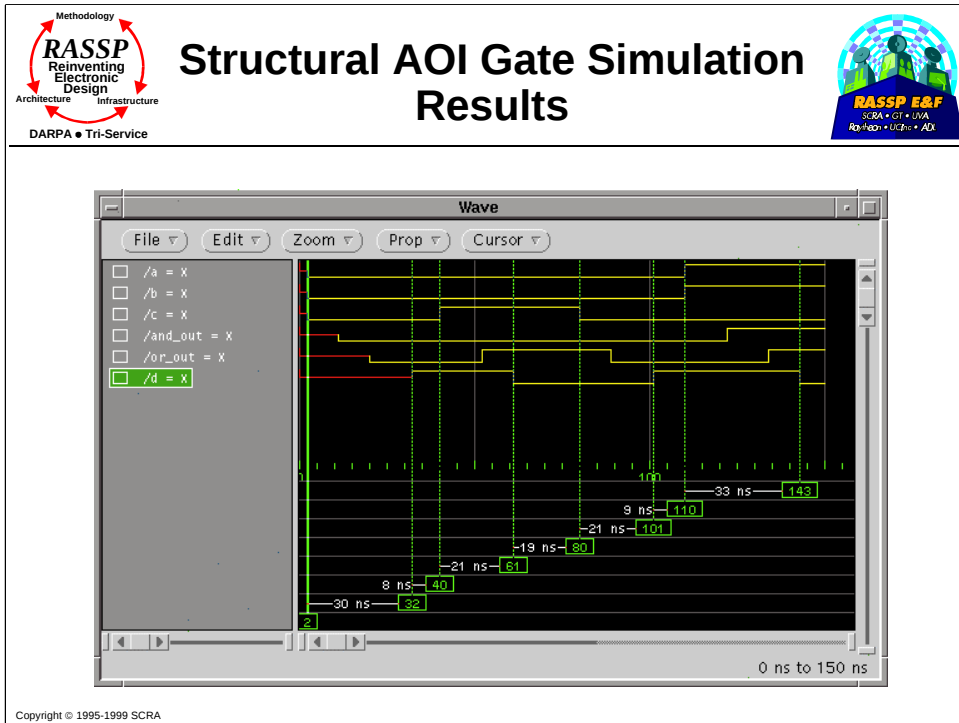
  INV_1 : inv  GENERIC MAP(trise => trise,
                           tfall => tfall)
    PORT MAP(a => or_out, b => d);

END structural;

```

Copyright © 1995-1999 SCRA

This is the structural architecture of the and-or-invert gate. It shows the three major elements of a structural description. The component declarations which list which components will be used in the structure are in yellow. The binding indications which tell what library the component comes from and which library component is to be used for each declared component are in blue. Finally, the green highlights the component instantiations where the individual components are “placed” in the structure and connected to the proper generics and ports or signals.



This is the QuickVHDL simulation results for the simple AOI gate. Note that the values on the internal signals are traced.

Structural 8 Bit Register Example Simple Generate Statement

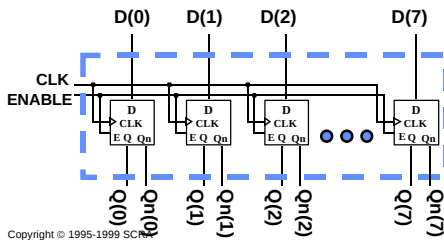
```
LIBRARY gate_lib;
USE gate_lib.resources.all;

ENTITY reg8_str IS

    GENERIC(tprop : delay := 8 ns;
            tsu : delay := 2 ns);

    PORT(d      : IN level_vector(7 DOWNT0 0);
         clk    : IN level;
         enable  : IN level;
         q       : OUT level_vector(7 DOWNT0 0);
         qn      : OUT level_vector(7 DOWNT0 0));

END reg8_str;
```



Copyright © 1995-1999 SCRA

ARCHITECTURE structural OF reg8_str IS

```
-- COMPONENT DECLARATION
COMPONENT dff
    GENERIC(tprop : delay;
            tsu : delay);
    PORT(d      : IN level;
         clk    : IN level;
         enable  : IN level;
         q       : OUT level;
         qn      : OUT level);
END COMPONENT;
```

```
-- BINDING INDICATIONS
FOR ALL : dff USE ENTITY gate_lib.dff(behav);
```

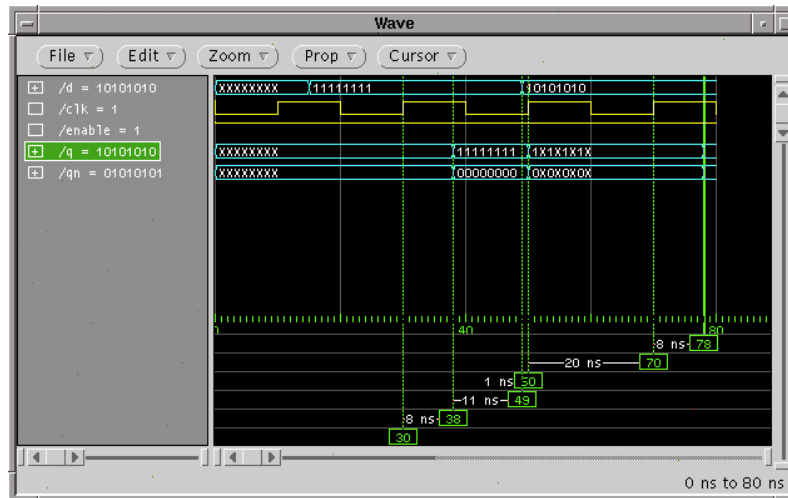
BEGIN

```
-- COMPONENT INSTANTIATION (GENERATE)
R1:FOR i IN 1 TO 8 GENERATE
    I1:dff GENERIC MAP(tprop => tprop,
                       tsu => tsu)
        PORT MAP(d => d(i-1), clk => clk,
                 enable => enable,
                 q => q(i-1), qn => qn(i-1));
END GENERATE R1;
```

END structural;

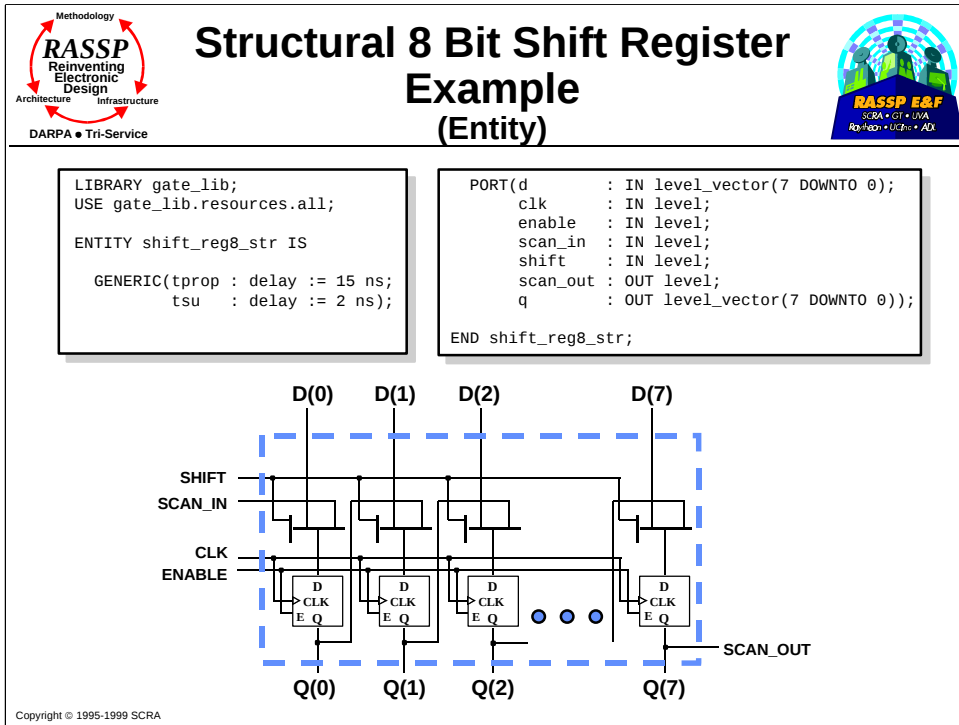
This is a structural description of an 8 bit register using DFFs from the library. A simple generate statement is used to instantiate the DFFs and connect them to the individual “bits” at the register’s input and output. The colors highlight the same parts of the structural description as before.

Structural 8 Bit Register Simulation Results



Copyright © 1995-1999 SCRA

Simulation results; it works!



This is the schematic and entity description for an 8 bit shift register.

Note that in the schematic, signals will be needed between the multiplexor output and the dff's input and the dff output and the shift register output (Q). The signal on the DFF outputs is needed because it feeds back to the input of the muxes, and that can't be done by connecting both directly to an output port (signals of mode OUT are not readable inside the architecture). Thus, some concurrent signal assignment statements will be necessary to connect the signal at the dff outputs to the Q outputs.



RASPP
Reinventing
Electronic
Design
Architecture Infrastructure
DARPA • Tri-Service

Structural 8 Bit Shift Register Example

(Architecture - Generate with If Scheme)



RASPP E&F
SCRA • GI • UVA
Rpi • Geo • UChicago • AD

```

ARCHITECTURE structural OF shift_reg8_str IS

  -- COMPONENT DECLARATION
  COMPONENT mux2
    GENERIC(tprop : delay);
    PORT(a : IN level;
          b : IN level;
          sel : IN level;
          c : OUT level);
  END COMPONENT;

  COMPONENT dff
    GENERIC(tprop : delay;
            tsu : delay);
    PORT(d : IN level;
          clk : IN level;
          enable : IN level;
          q : OUT level;
          qn : OUT level);
  END COMPONENT;

  -- BINDING INDICATIONS
  FOR ALL : mux2 USE ENTITY
    gate_lib.mux2(behav);
  FOR ALL : dff USE ENTITY
    gate_lib.dff(behav);

  SIGNAL mux_out : level_vector(7 DOWNTO 0);
  SIGNAL dff_out : level_vector(7 DOWNTO 0);

BEGIN

```

```

-- COMPONENT INSTANTIATION (GENERATE W/ IF)
G1:FOR i IN 0 TO 7 GENERATE

  G2 : IF (i = 0) GENERATE
    MUX1 : mux2 GENERIC MAP(tprop => tprop/2)
      PORT MAP(a => scan_in,
                b => d(i),
                sel => shift,
                c => mux_out(i));
    DFF1 : dff GENERIC MAP(tprop => tprop/2,
                          tsu => tsu)
      PORT MAP(d => mux_out(i),
                clk => clk,
                enable => enable,
                q => dff_out(i));
    q(i) <= dff_out(i);
  END GENERATE G2;
  G3 : IF (i > 0) GENERATE
    MUX1 : mux2 GENERIC MAP(tprop => tprop/2)
      PORT MAP(a => dff_out(i-1),
                b => d(i),
                sel => shift,
                c => mux_out(i));
    DFF1 : dff GENERIC MAP(tprop => tprop/2,
                          tsu => tsu)
      PORT MAP(d => mux_out(i),
                clk => clk,
                enable => enable,
                q => dff_out(i));
    q(i) <= dff_out(i);
  END GENERATE G3;
END GENERATE G1;

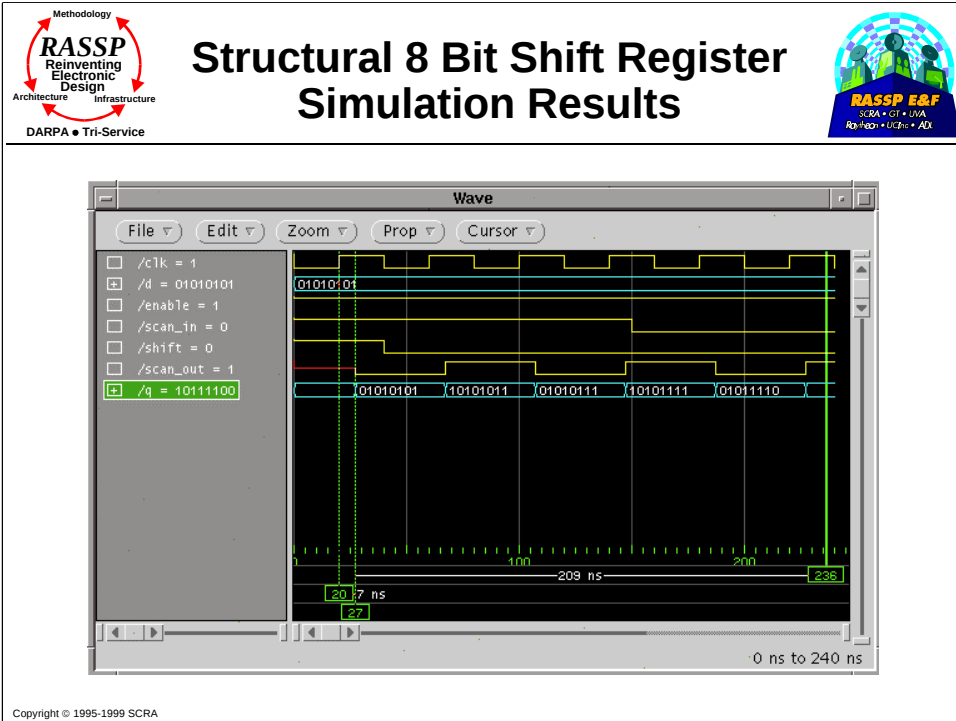
scan_out <= dff_out(7);

END structural;

```

This is the architecture which uses a complex generate statement. There is an IF statement within the generate statement to handle the fact that the mux that is connected to the 0th bit is connected to *scan_in* instead of the output of the DFF in the previous bit position. Here again, the colors highlight the three parts of the structural description.

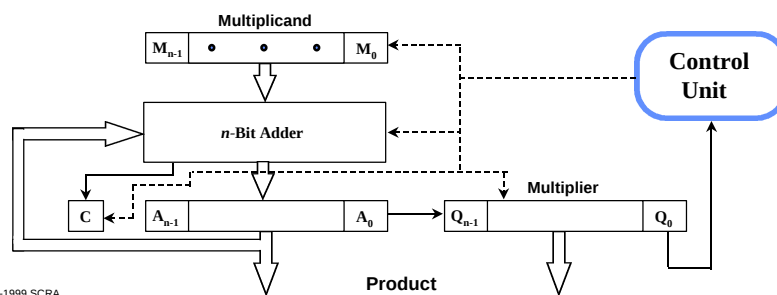
Note that the concurrent signal assignment statements to connect the *dff_out* signal to the Q outputs are inside the generate statements.



Here is the simulation results showing a parallel load followed by scanning the loaded data.

Unsigned 8 Bit Multiplier Data Path (Entity)

```
LIBRARY work;
LIBRARY gate_lib;
USE gate_lib.resources.all;
ENTITY mult_datapath IS
  PORT(multiplicand : IN level_vector(7 DOWNTO 0);
        multiplier   : IN level_vector(7 DOWNTO 0);
        a_enable     : IN level; -- clock enable for A register
        a_reset      : IN level; -- Reset control for A register
        a_mode       : IN level; -- Shift or load mode for A
        c_enable     : IN level; -- clock enable for c register
        m_enable     : IN level; -- clock enable for M register
        q_enable     : IN level; -- clock enable for Q register
        q_mode       : IN level; -- Shift or load mode for Q
        clk          : IN level;
        product      : OUT level_vector(15 DOWNTO 0));
END mult_datapath;
```



Copyright © 1995-1999 SCRA

The final example is of an RTL level datapath for an unsigned 8 bit multiplier. It illustrates the use of complex components in a structural description and the use of multiple levels of hierarchy in that the components are themselves structural descriptions of lower level components. This is the entity description for the datapath. The register controls (*enable*, *mode*) will go to the control unit when it is added.



RASSP
Reinventing
Electronic
Design

Architecture Infrastructure

DARPA • Tri-Service

Unsigned 8 Bit Multiplier Data Path (Architecture)



RASSP E&F
SCRA • GI • UVA
Rapidgo • UChicago • AD

ARCHITECTURE structural OF mult_datapath IS

```

COMPONENT dff
  GENERIC(tprop : delay;
           tsu   : delay);
  PORT(d       : IN level;
        clk    : IN level;
        enable : IN level;
        q      : OUT level;
        qn     : OUT level);
END COMPONENT;

COMPONENT reg8_str
  GENERIC(tprop : delay;
           tsu   : delay);
  PORT(d       : IN level_vector(0 TO 7);
        clk    : IN level;
        enable : IN level;
        q      : OUT level_vector(0 TO 7);
        qn     : OUT level_vector(0 TO 7));
END COMPONENT;

```

```

COMPONENT shift_reg8_str
  GENERIC(tprop : delay;
           tsu   : delay);
  PORT(d       : IN level_vector(0 TO 7);
        clk    : IN level;
        enable : IN level;
        scan_in : IN level;
        shift   : IN level;
        scan_out : OUT level;
        q       : OUT level_vector(0 TO 7));
END COMPONENT;

COMPONENT alu_str
  GENERIC(tprop : delay);
  PORT(a       : IN level_vector(7 DOWNTO 0);
        b       : IN level_vector(7 DOWNTO 0);
        mode    : IN level;
        cin     : IN level;
        sum     : OUT level_vector(7 DOWNTO 0);
        cout    : OUT level);
END COMPONENT;

FOR ALL : dff USE ENTITY gate_lib.dff(behav);
FOR ALL : reg8_str USE ENTITY
  work.reg8_str(structural);
FOR ALL : shift_reg8_str USE ENTITY
  work.shift_reg8_str(structural);
FOR ALL : alu_str USE ENTITY
  work.alu_str(structural);

```

Copyright © 1995-1999 SCRA

This shows the component declarations and binding indications for the datapath.



RASSP
Reinventing
Electronic
System
Path

DARPA • Tri-Service

Unsigned 8 Bit Multiplier Data Path (Architecture)



RASSP E&F
SCRA • GI • UVA
Rapid • UChicago • AD

```

SIGNAL gnd                : level := '0';
SIGNAL c_out, a_scan_out, carry_out : level;
SIGNAL a_out, alu_out, m_out
                : level_vector(7 DOWNT0 0);

BEGIN

  -- A, C, M, and Q registers
  A1 : shift_reg8_str GENERIC MAP(6 ns, 1 ns)
    PORT MAP(d(0)=>alu_out(0),d(1)=>alu_out(1),
              d(2)=>alu_out(2),d(3)=>alu_out(3),
              d(4)=>alu_out(4),d(5)=>alu_out(5),
              d(6)=>alu_out(6),d(7)=>alu_out(7),
              clk => clk, enable => a_enable,
              scan_in => c_out, shift => a_mode,
              scan_out => a_scan_out,
              q(0)=>a_out(0),q(1)=>a_out(1),
              q(2)=>a_out(2),q(3)=>a_out(3),
              q(4)=>a_out(4),q(5)=>a_out(5),
              q(6)=>a_out(6),q(7)=>a_out(7));

  C1 : dff GENERIC MAP(5 ns, 1 ns)
    PORT MAP( d => carry_out, clk => clk,
              enable => c_enable, q => c_out);

  M1 : reg8_str GENERIC MAP(4 ns, 1 ns)
    PORT MAP(d => multiplicand, clk => clk,
              enable => m_enable, q => m_out);

```

```

Q1 : shift_reg8_str GENERIC MAP(6 ns, 1 ns)
  PORT MAP(d(0) => multiplier(0),
            d(1) => multiplier(1),
            d(2) => multiplier(2),
            d(3) => multiplier(3),
            d(4) => multiplier(4),
            d(5) => multiplier(5),
            d(6) => multiplier(6),
            d(7) => multiplier(7),
            clk => clk,
            enable => q_enable,
            scan_in => a_scan_out,
            shift => q_mode,
            q(0) => product(0),
            q(1) => product(1),
            q(2) => product(2),
            q(3) => product(3),
            q(4) => product(4),
            q(5) => product(5),
            q(6) => product(6),
            q(7) => product(7));

  -- ALU
  ALU1 : alu_str GENERIC MAP(8 ns)
    PORT MAP(a => m_out, b => a_out,
              mode => a_reset,
              cin => gnd,
              sum => alu_out,
              cout => carry_out);

  -- connect A register output to product
  product(15 DOWNT0 8)<=a_out(7 DOWNT0 0);
END structural;

```

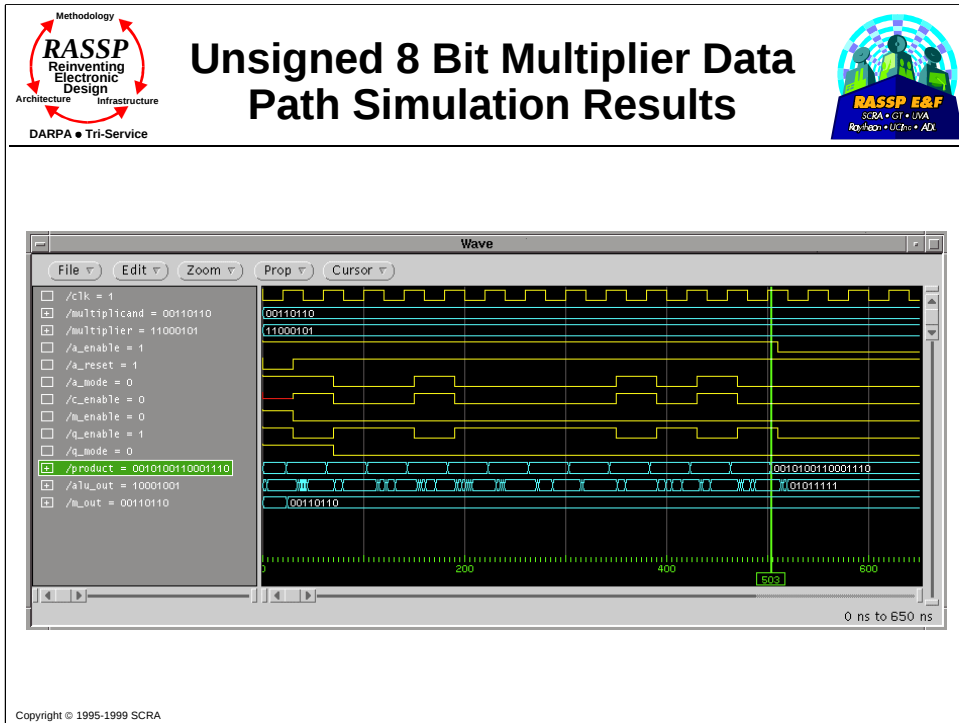
Copyright © 1995-1999 SCRA

This is the component instantiations for the datapath. The mapping of individual bits of the *D* and *Q* input and output of the shift registers is necessary to reverse the inputs and outputs. Recall that the shift register was a “shift up” type where the *scan_in* input goes to *D*(0) and *D*(0) to *D*(6) go to *D*(1) to *D*(7) when in shift mode. What is needed for the multiplier is a “shift down” type register where *scan_in* goes to *D*(7), etc.

It should have been possible (we believe) to use the syntax

d => multiplier(0 to 7)

in the *shift_reg8_str* PORT MAP to accomplish the same thing, but the QuickVHDL compiler gave a “warning” and the simulator crashed, so we don’t know if it really should work.



This shows the simulation results. The control points were manipulated by using forces in the simulation. Note the correct result “0010100110001110” on the *product* output.



Module Outline



- | Introduction
- | Component Instantiation
- | Generate Statement
- | Examples
- | **Summary**

Copyright © 1995-1999 SCRA



Summary



- | **Structural VHDL describes the arrangement and interconnection of components**
- | **Components can be at any level of abstraction -- low level gates or high level blocks of logic**
- | **Generics are inherited by every architecture or component of that entity**
- | **GENERATE statements create large, regular blocks of logic easily**
- | **Configurations give the designer control over the entity and architecture used for a component**

Copyright © 1995-1999 SCRA



References



- [Bhasker95] Bhasker, J. A VHDL Primer, Prentice Hall, 1995.
- [Calhoun95] Calhoun, J.S., Reese, B.,. "Class Notes for EE-4993/6993: Special Topics in Electrical Engineering (VHDL)", Mississippi State University, <http://www.erc.msstate.edu/>, 1995.
- [Coelho89] Coelho, D. R., The VHDL Handbook, Kluwer Academic Publishers, 1989.
- [IEEE] All referenced IEEE material is used with permission.
- [Lipsett89] Lipsett, R., C. Schaefer, C. Ussery, VHDL: Hardware Description and Design, Kluwer Academic Publishers, , 1989.
- [LRM93] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993.
- [Menchini94] Menchini, P., "Class Notes for Top Down Design with VHDL", 1994.
- [MG90] An Introduction to Modeling in VHDL, Mentor Graphics Corporation, 1990.
- [MG93] Introduction to VHDL, Mentor Graphics Corporation, 1993.
- [Navabi93] Navabi, Z., VHDL: Analysis and Modeling of Digital Systems, McGraw-Hill, 1993.
- [Perry94] Perry, D. L., VHDL, McGraw-Hill, 1994.
- [Richards97] Richards, M., Gadiant, A., Frank, G., eds. *Rapid Prototyping of Application Specific Signal Processors*, Kluwer Academic Publishers, Norwell, MA, 1997
- [Williams94] Williams, R. D., "Class Notes for EE 435: Computer Organization and Design", University of Virginia, <http://www.ee.virginia.edu/research/CSIS/>, 1994.

Copyright © 1995-1999 SCRA