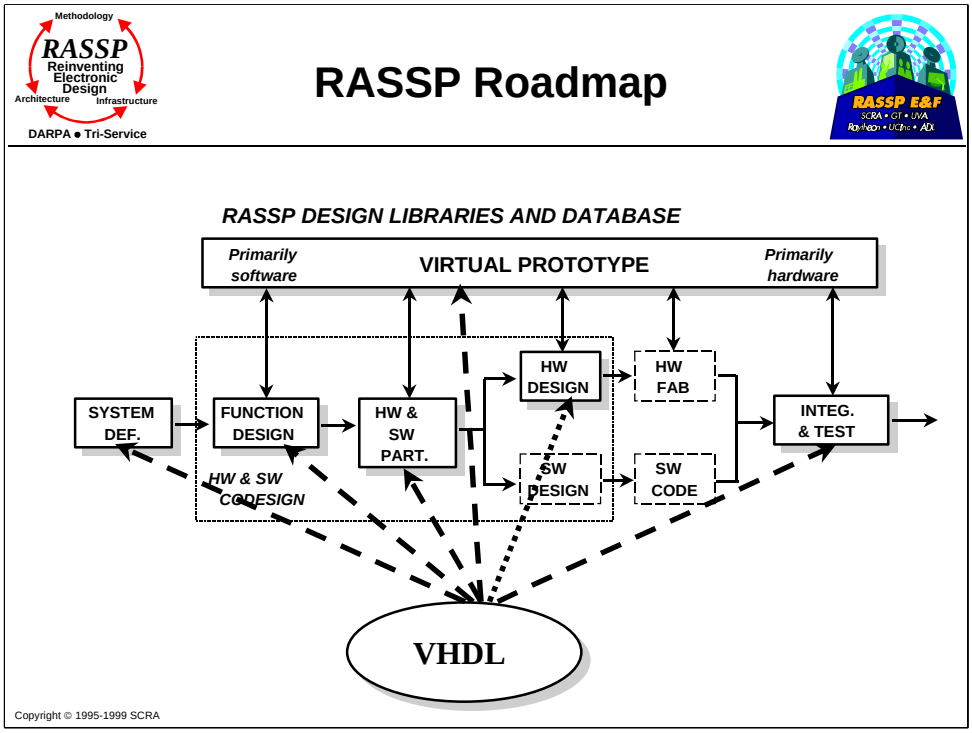
Structural VHDL

RASSP Education & Facilitation

Module 11

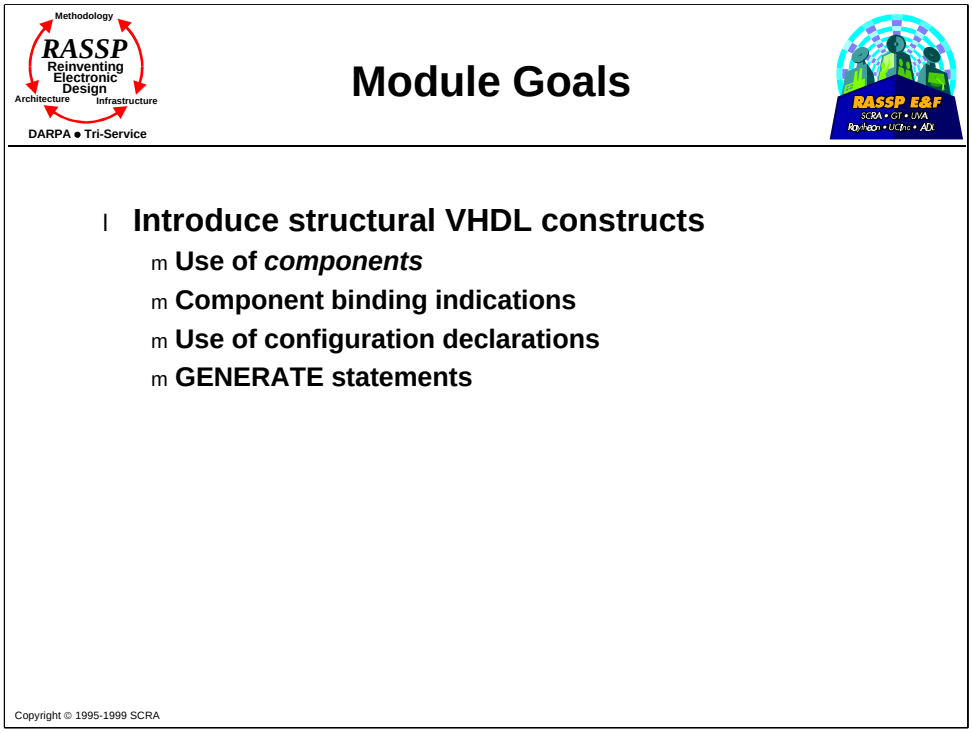
Version 3.00

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.

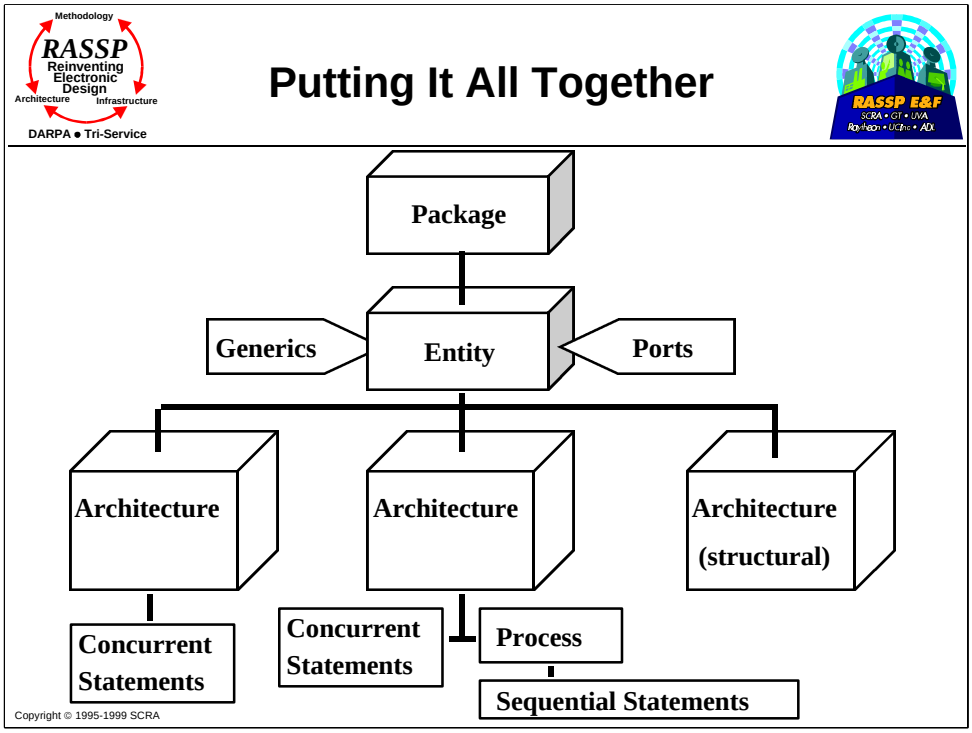
The most common usage of VHDL prior to RASSP was in the area of hardware design. The RASSP program has extended VHDL's use to include executable requirements, performance modeling/system level design as well as system integration and test.



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



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.

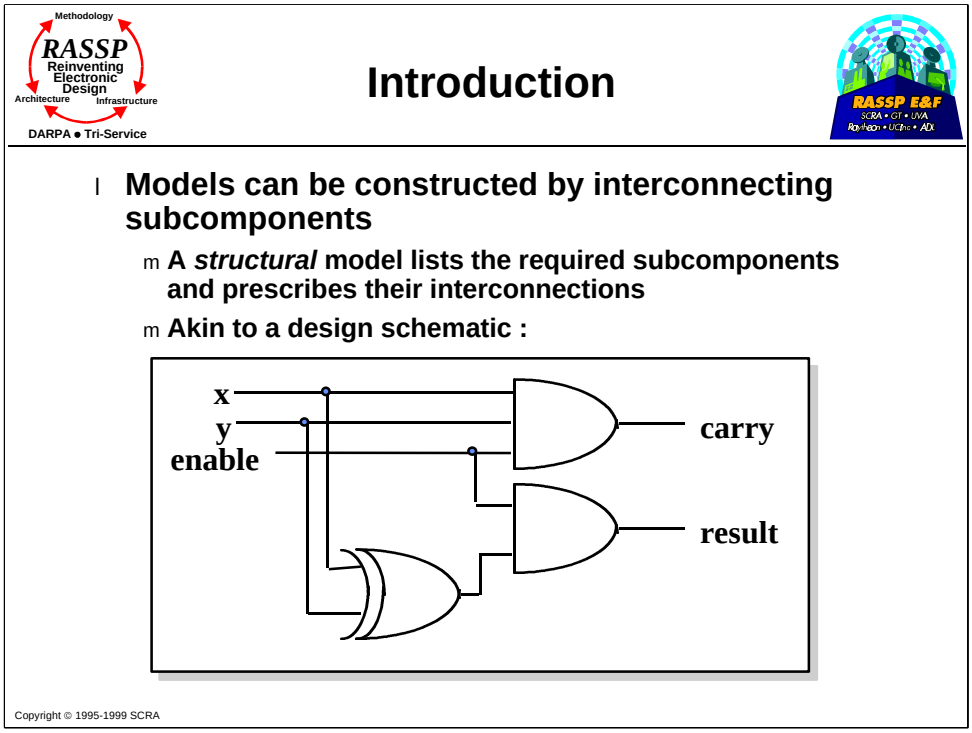
Packages are used to provide a collection of common declarations, constants, and/or subprograms to entities and architectures.

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.

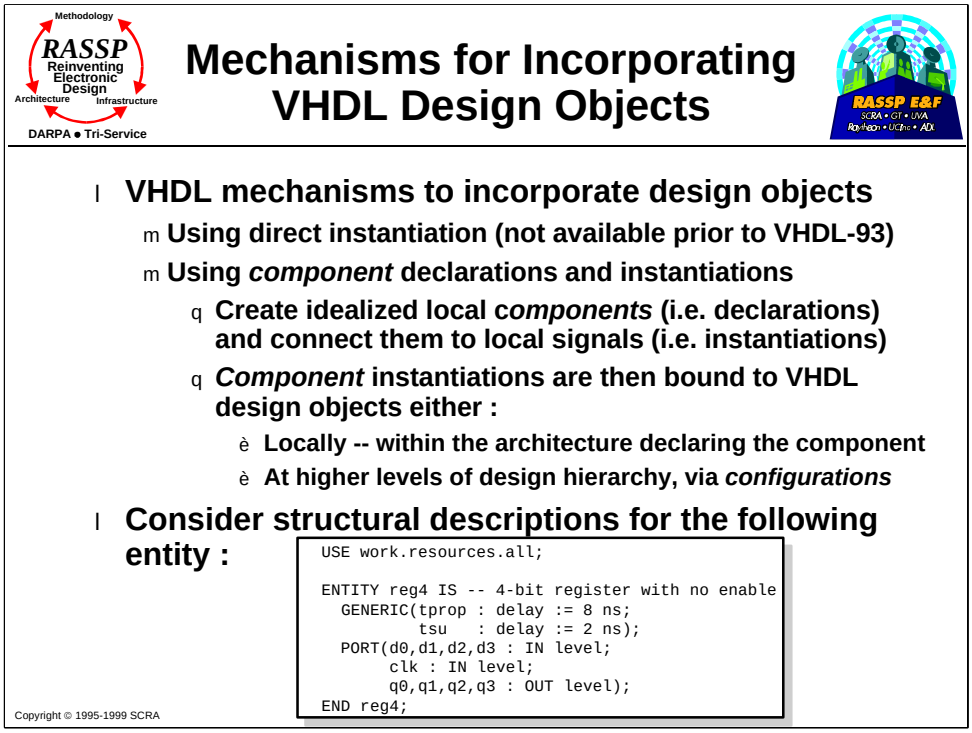
[MG93]



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.

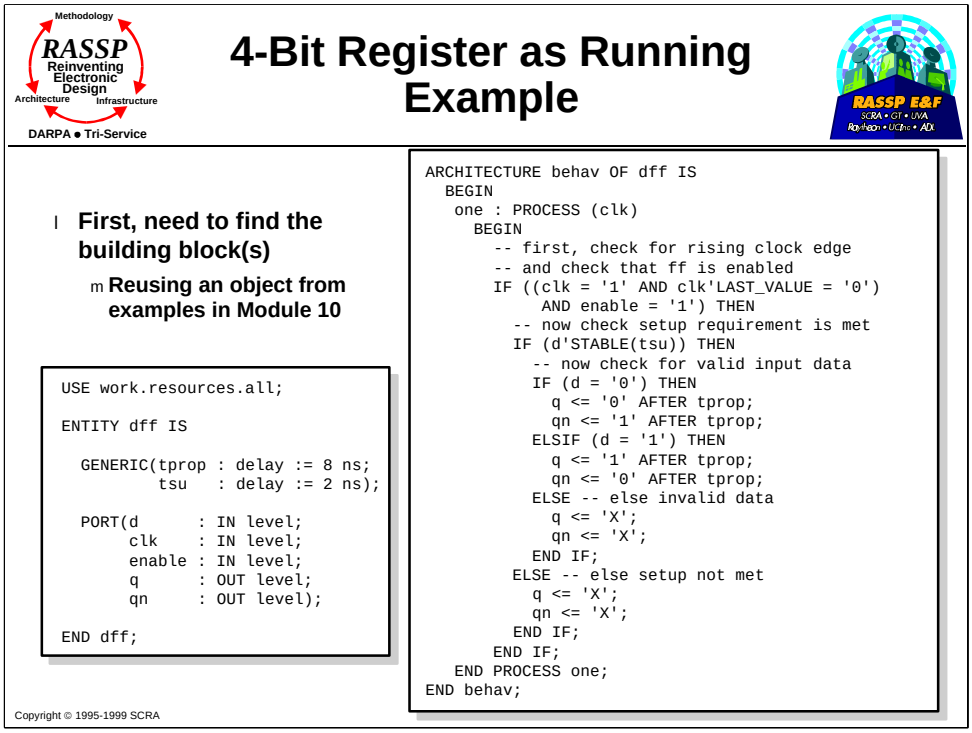
Module Outline

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

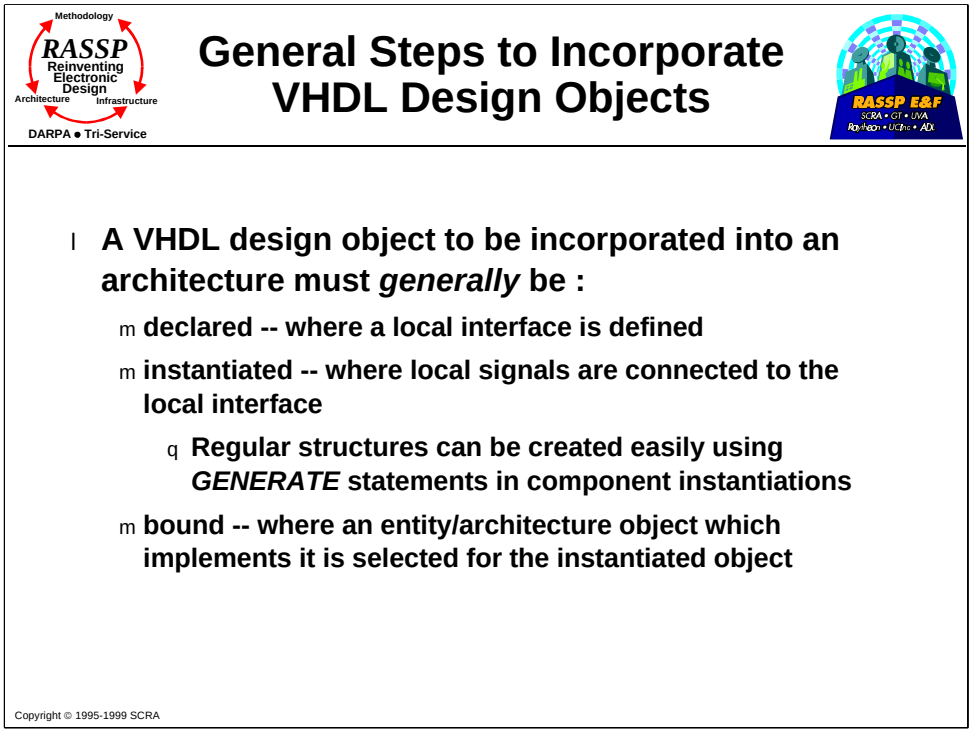


There are a number of constructs available in VHDL to incorporate design objects into architecture descriptions. The simplest, but least versatile, is the direct instantiation of a VHDL *entity.* With this method, the details of the entity's interface must be known and cannot be customized at instantiation.

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.

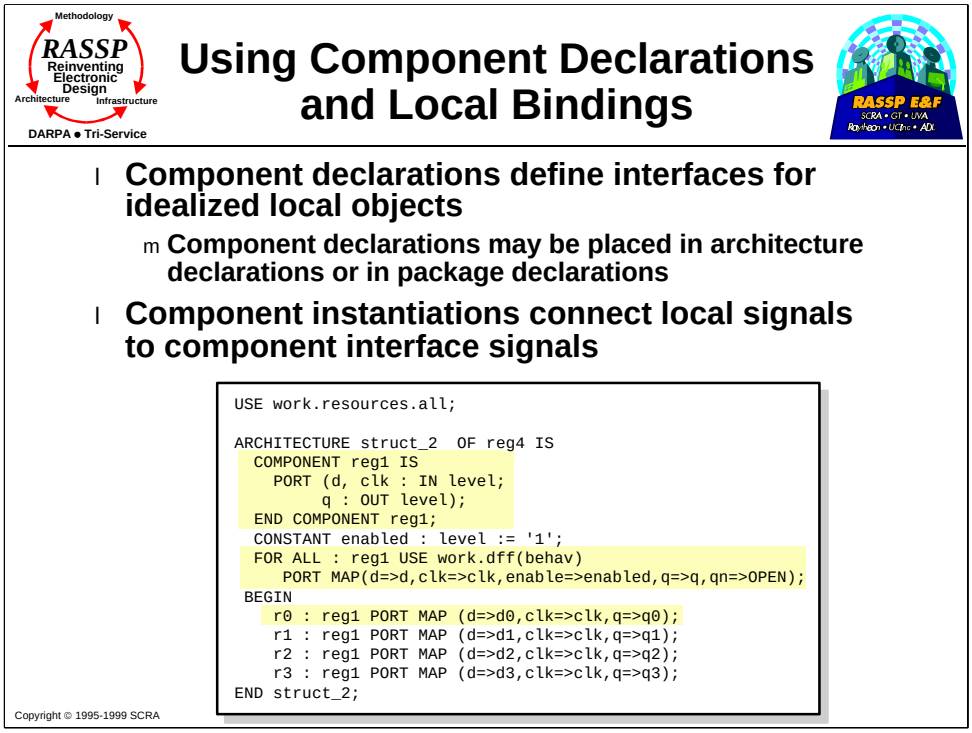


As a running example, we will build a 4-bit register using the D flip-flop model presented in Module 10, the Basic VHDL Module.



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.



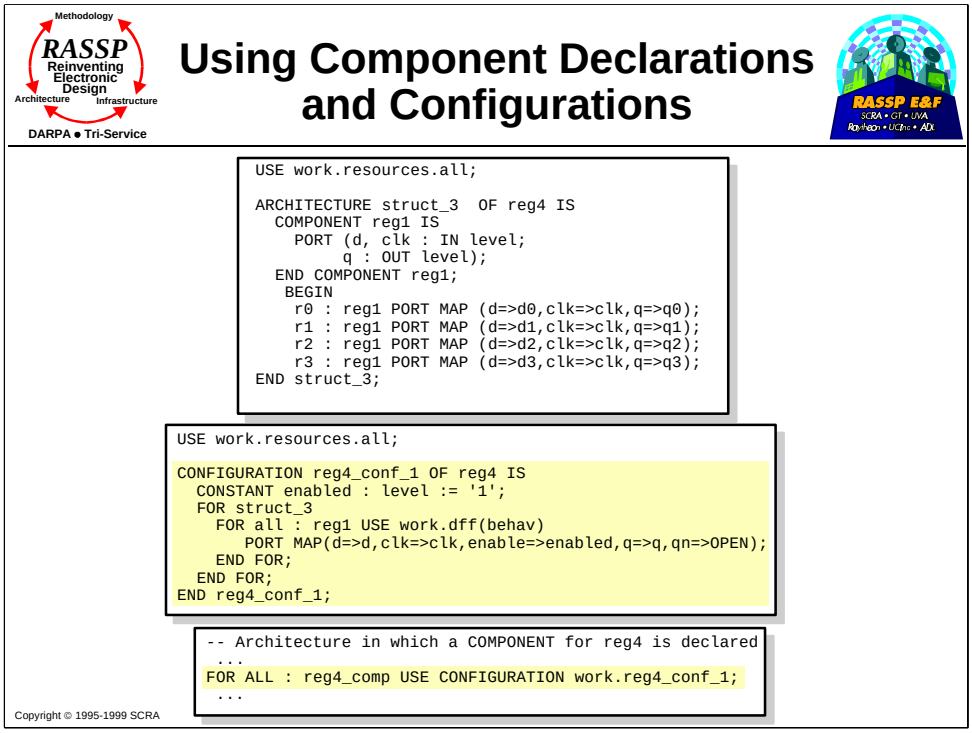
This example shows the three steps listed earlier:

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.

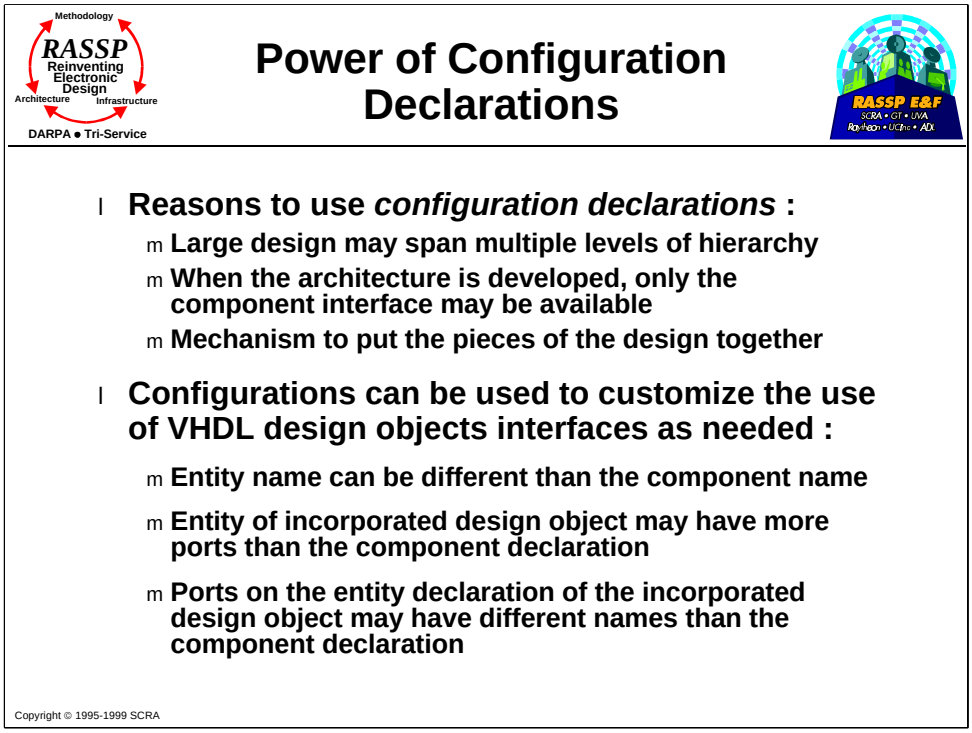


Note that in this example, three separate VHDL files are used. The first file above shows the architecture description in which the *reg1* component is declared and instantiated.

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).

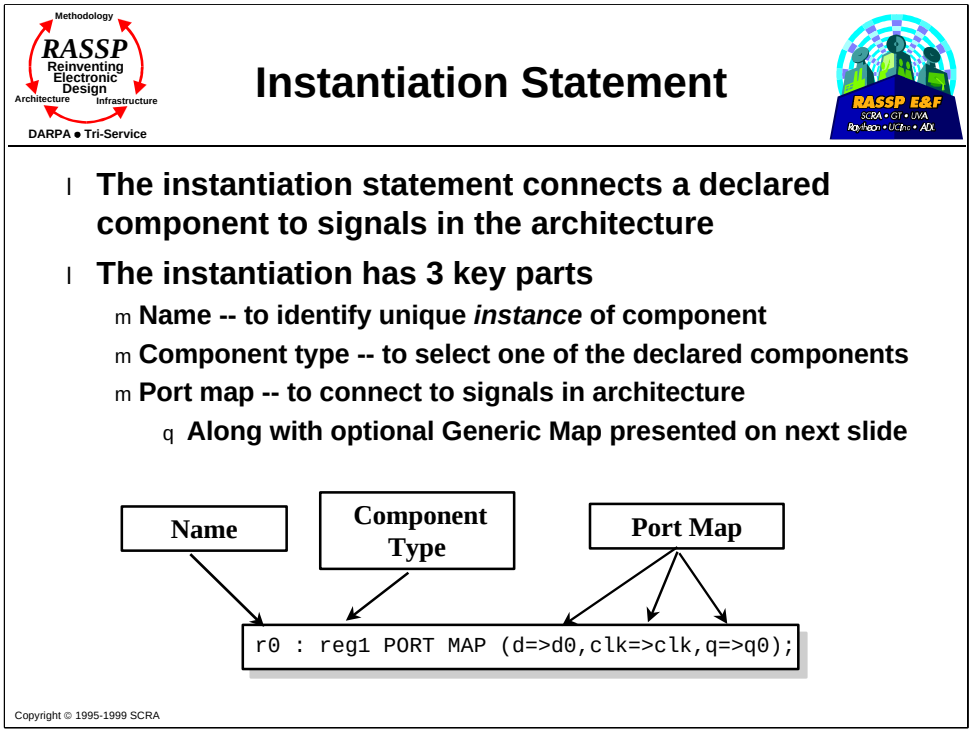
Note that the use of configurations to defer the binding of components adds flexibility to structural architecture descriptions by allowing alternative architecture to be plugged in to the design easily.



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

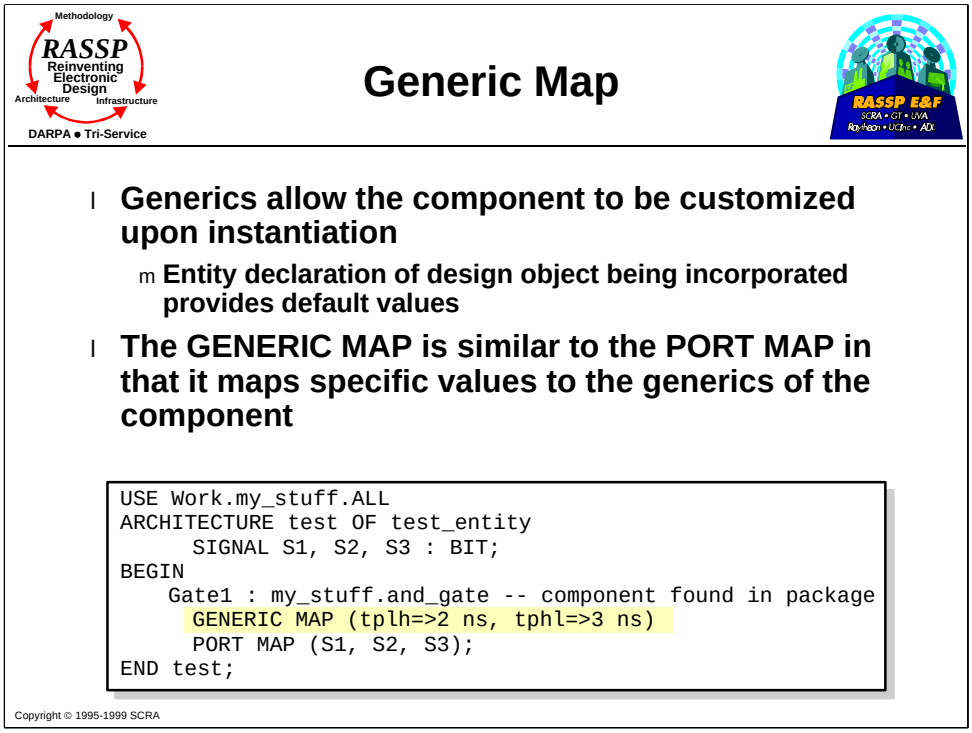
Similarly, they allow for structural descriptions to be developed before the entity/architecture building blocks have been finalized. This is particularly useful in a large system which may have been partitioned among several designers.

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).



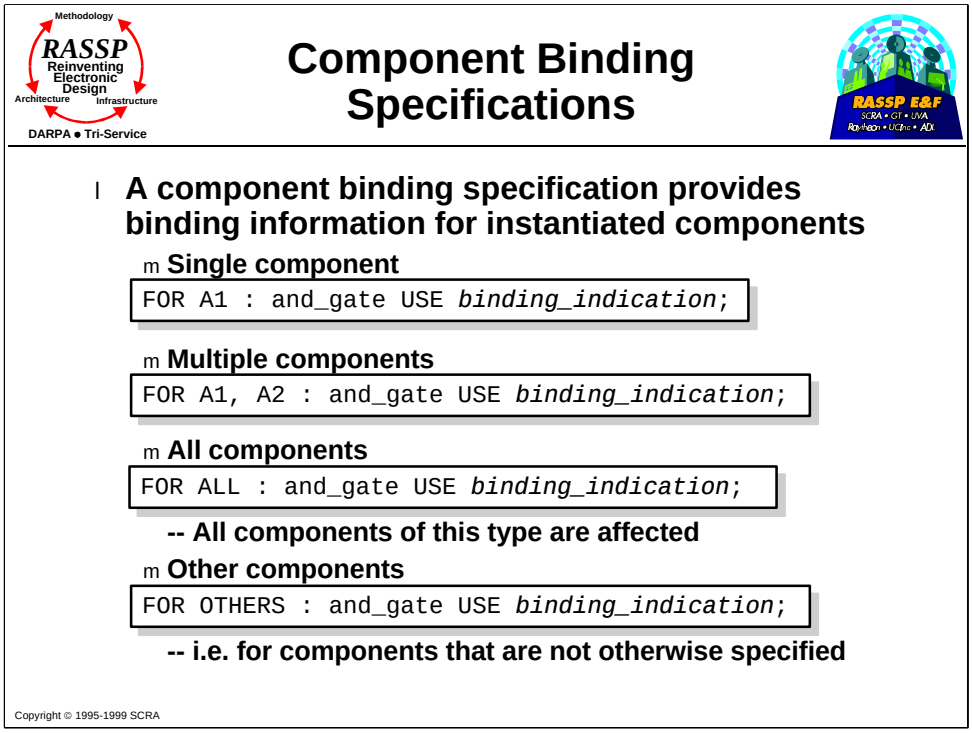
The above example shows an instance of the *reg1* component which has been given the name *r0*. The PORT MAP section of the instantiation indicates how the signals in the interface of the component are assigned to local signals.

Note that in this example, we associated each signal to a port on the component by naming the PORT signals explicitly. VHDL also allows for positional association, and the two styles may be used together as long as the association is not then made ambiguous.



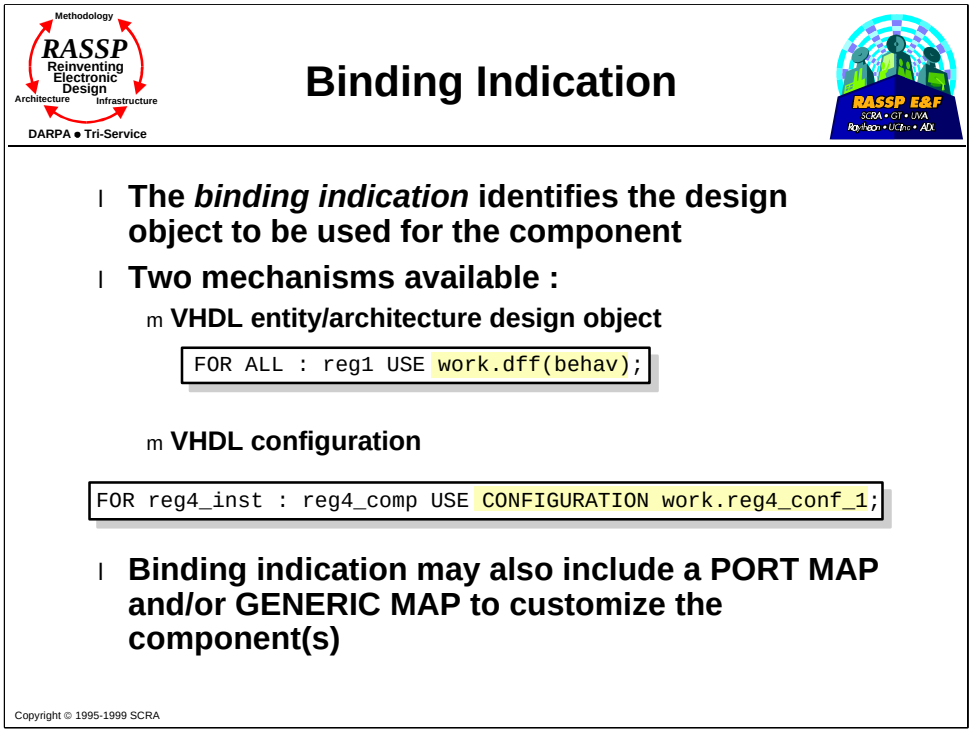
Generics are mapped in a fashion similar to ports. If no default values are assigned in the design object's ENTITY declaration, a GENERIC MAP must be provided in the component's declaration, instantiation, or binding.

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

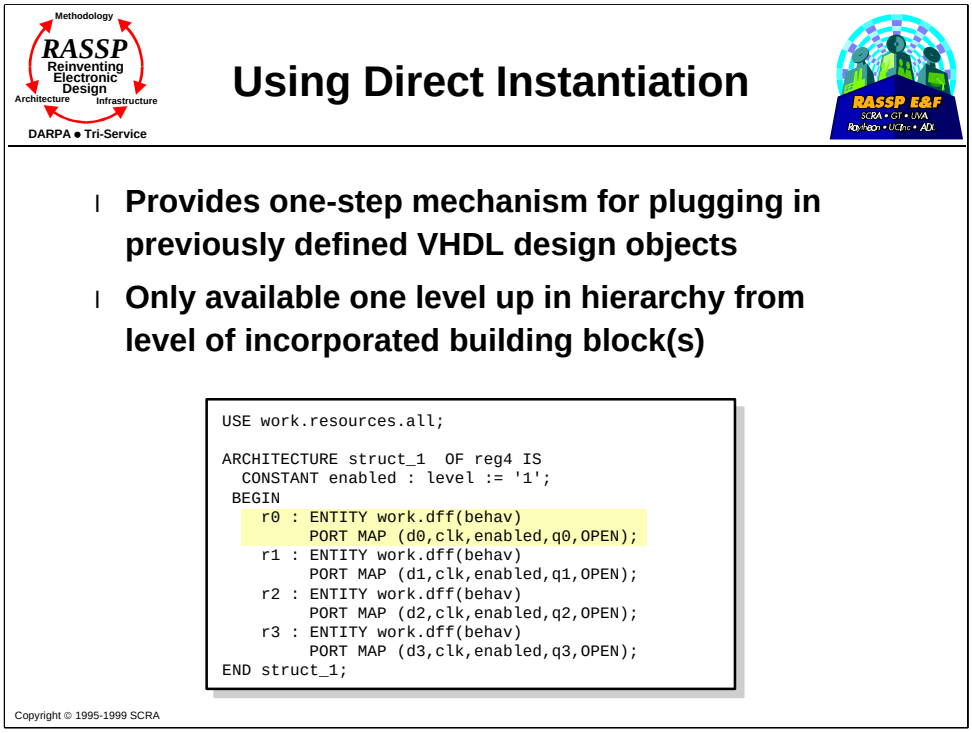


Unfortunately, component binding specifications are referred to as “configuration specifications” in the VHDL Language Reference Manual, but the term is avoided here to prevent confusion with *configuration* descriptions.

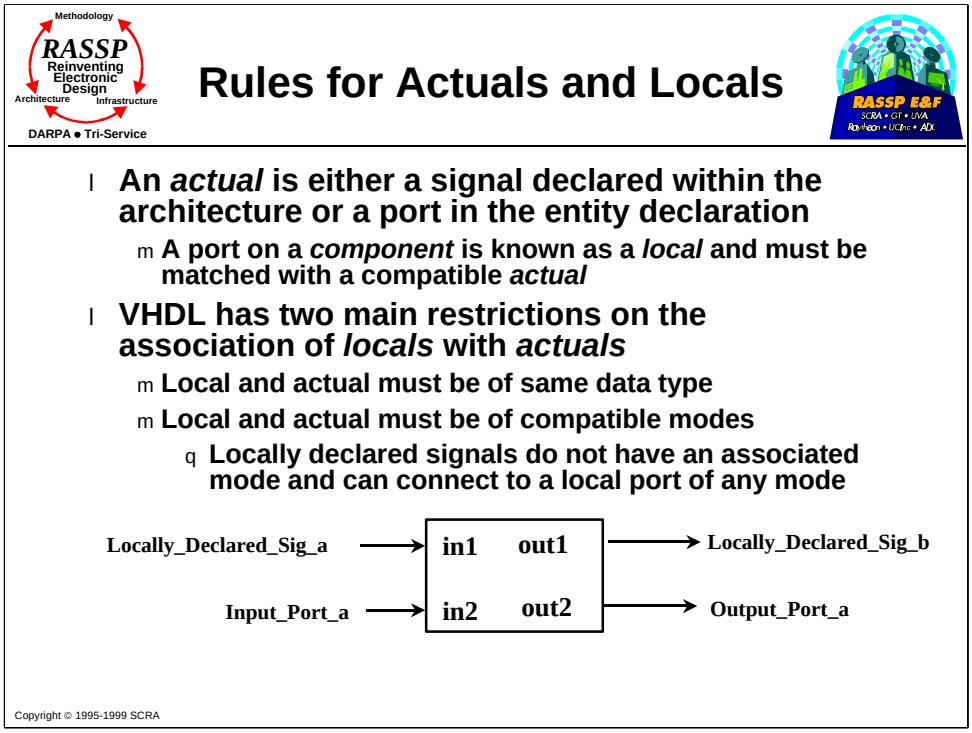
The component specification can be of several forms, and this slide shows examples for various types. The component specification identifies those components to be configured by name or by the keyword ALL. The keyword OTHERS selects all components not yet configured.



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.



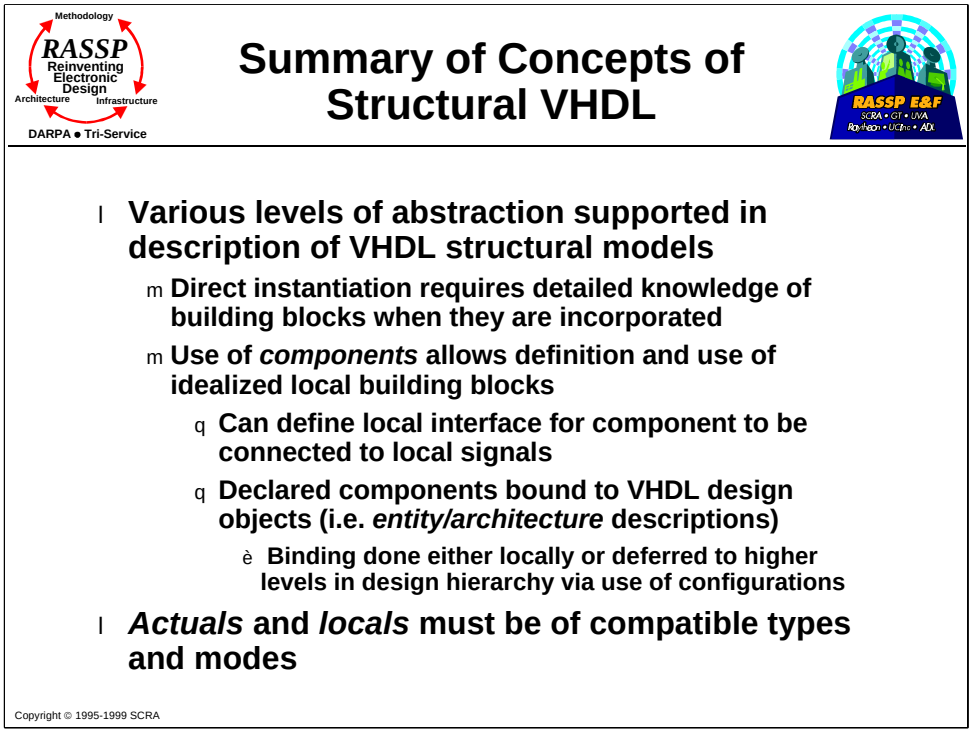
The *direct instantiation* method was introduced in VHDL-93. It allows a VHDL design object to be plugged in directly to an architecture's description by connecting local signals to its interface. This mechanism does not require the use of an idealized component to be declared, instantiated, and bound.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



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.

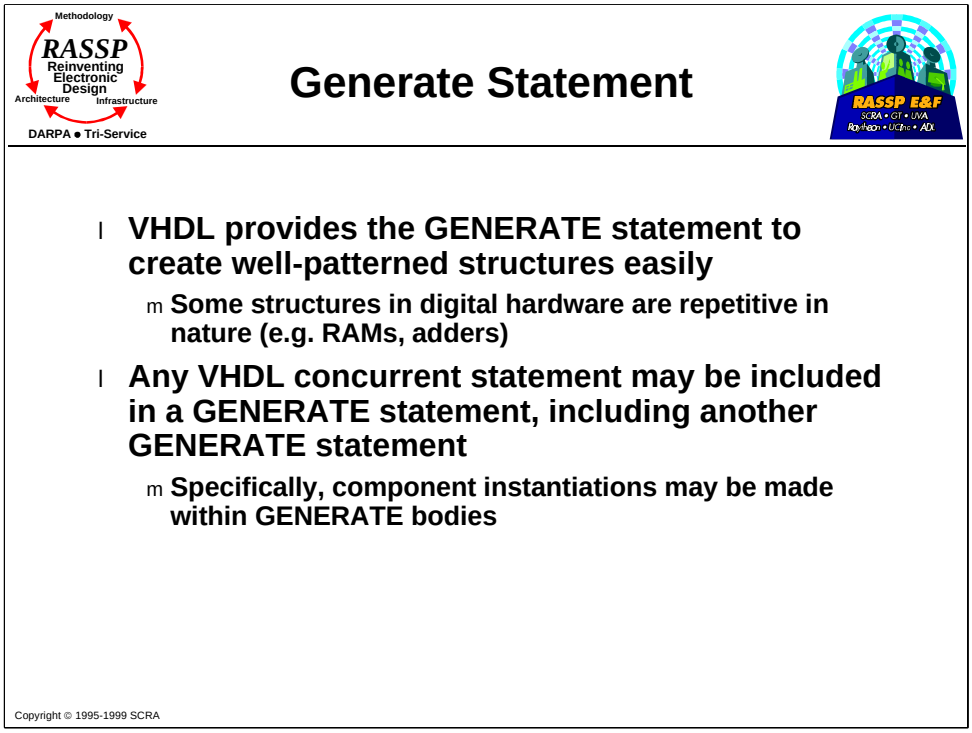
2) The local and actual must be of compatible modes. An actual of mode IN (i.e. a PORT of mode IN since locally declared signals do not have a mode) can only be associated with a local of mode IN, and an actual of mode OUT (i.e. a PORT of mode OUT) can only be associated with a local of mode OUT. A local 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).



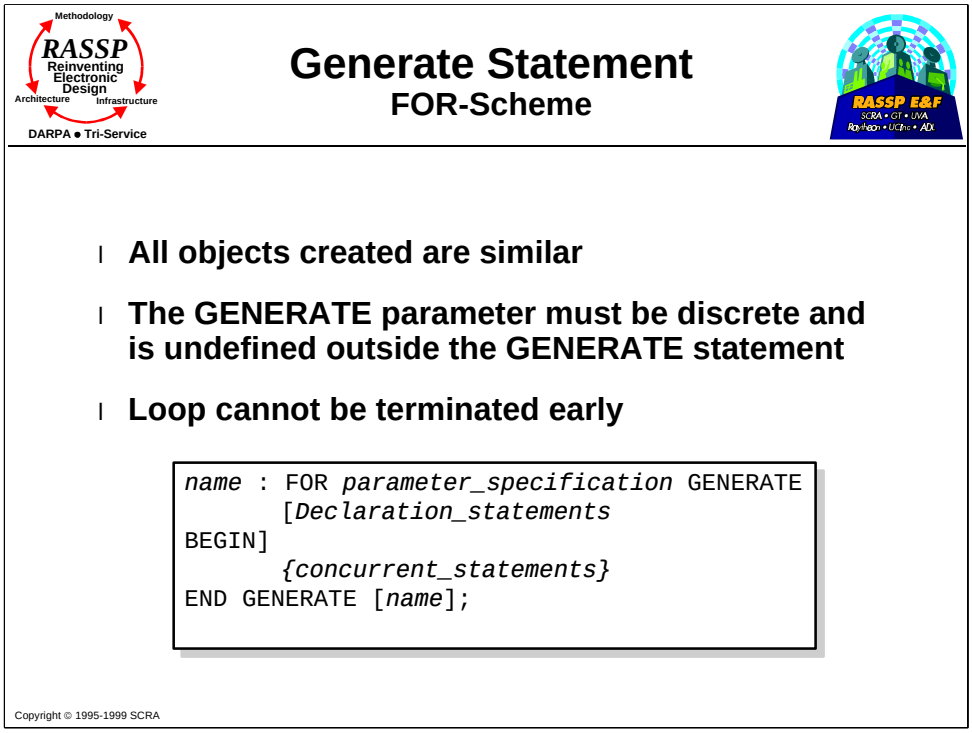
In summary, structural VHDL is concerned with the interconnection and arrangement of components describing the contents of a design. The behavior of the underlying design objects, therefore, is not explicitly indicated. A structural description can be thought of as a physical netlist describing a hierarchical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

Module Outline

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



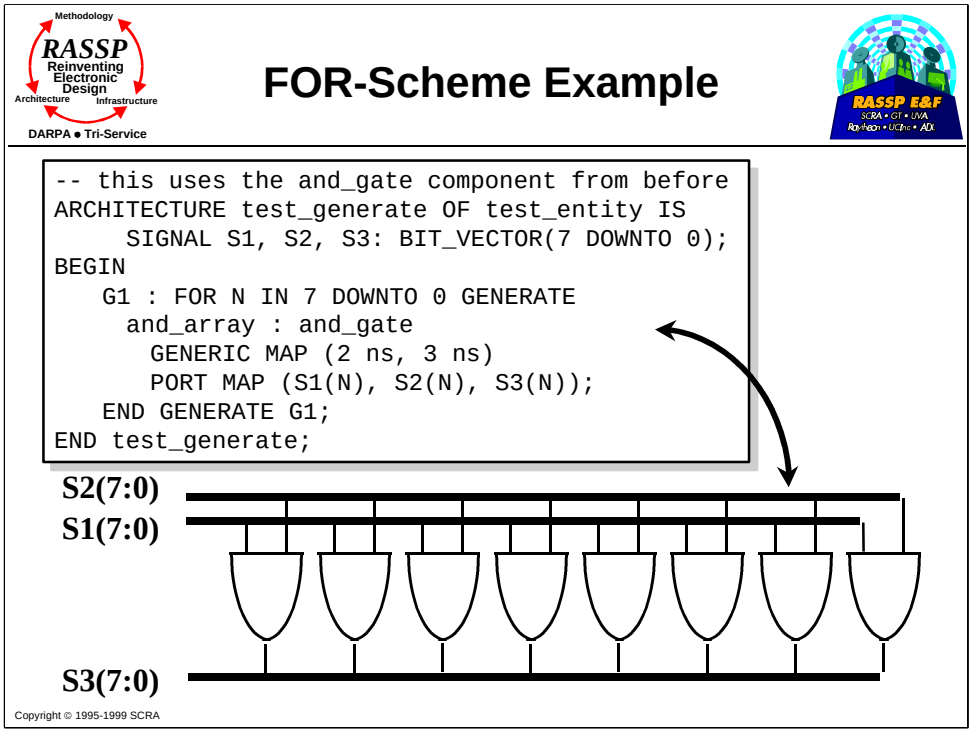
Structural descriptions of large, but highly regular, structures can be tedious. A VHDL GENERATE statement can be used to include as many concurrent VHDL statements (e.g. component instantiation statements) as needed to describe a regular structure easily. In fact, a GENERATE statement may even include 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.



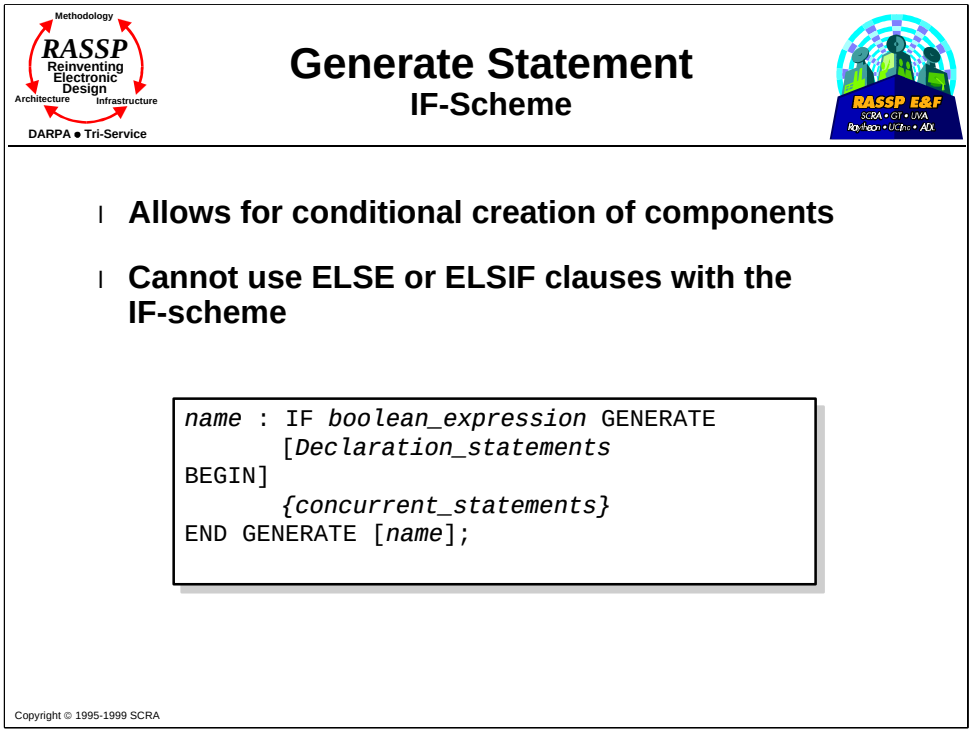
VHDL provides two different schemes of the GENERATE statement, the FOR-scheme and the IF-scheme. This slide shows the syntax for the FOR-scheme.

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.

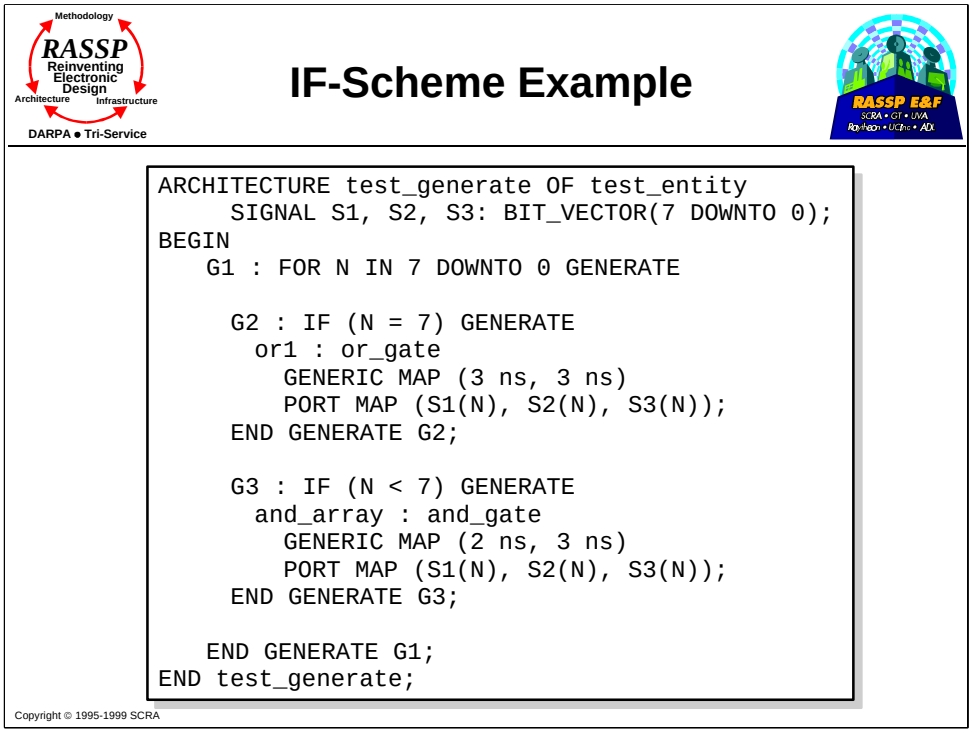


This slide shows an example of the FOR-scheme. The code generates an array of AND gates. In this case, the GENERATE statement has been named G1 and instantiates an array of 8 and_gate components. The PORT MAP statement maps the interfaces of each of the 8 gates to specific elements of the *S1*, *S2*,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



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.



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.

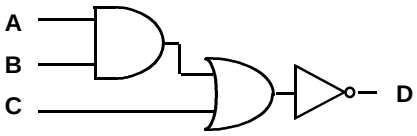
Module Outline

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

Examples

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

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;
```


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

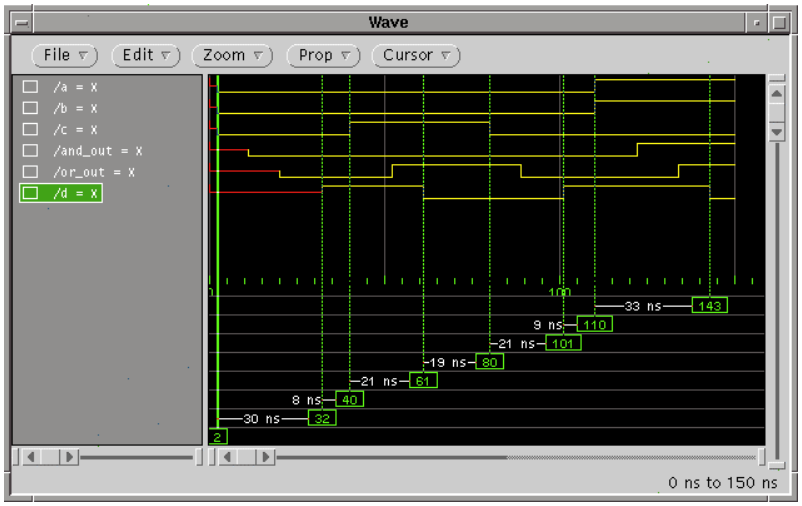
Structural And-Or-Invert Gate Example
(Architecture)

```
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;
```


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.

Structural AOI Gate Simulation Results

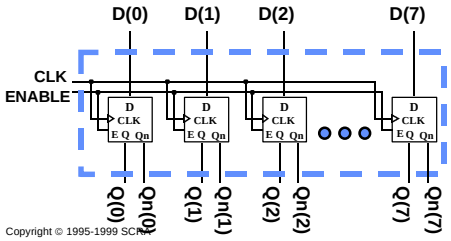


This is the QuickVHDL simulation results for the simple AOI gate. Note that the values on the internal signals are traced.

Structural 8 Bit Register Example

Simple Generate Statement

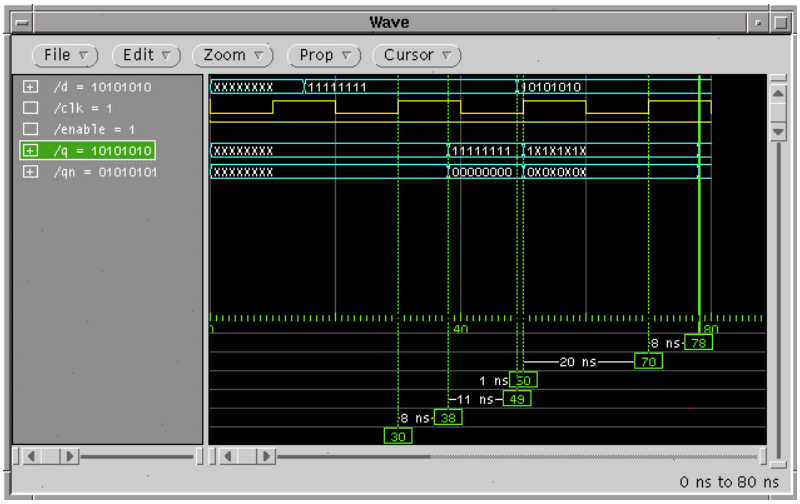
```
LIBRARY gate_lib;
USE gate_lib.resources.all;

ENTITY reg8_str IS

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

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

END reg8_str;
```




```
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.

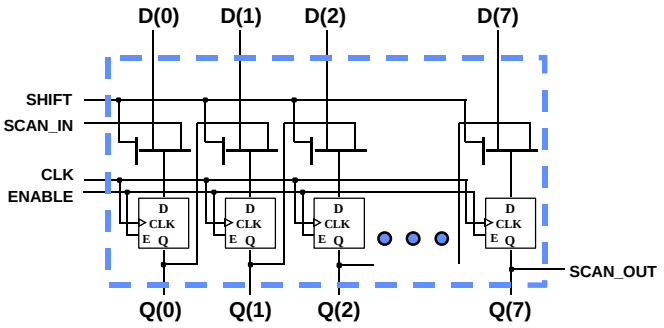
Structural 8 Bit Register Simulation Results



Simulation results; it works!

Structural 8 Bit Shift Register Example
(Entity)

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

ENTITY shift_reg8_str IS

  GENERIC(tprop : delay := 15 ns;
          tsu   : delay := 2 ns);

  PORT(d        : IN level_vector(7 DOWNTO 0);
       clk      : IN level;
       enable   : IN level;
       scan_in  : IN level;
       shift    : IN level;
       scan_out : OUT level;
       q        : OUT level_vector(7 DOWNTO 0));

END shift_reg8_str;
```



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.

Structural 8 Bit Shift Register Example
(Architecture - Generate with If Scheme)

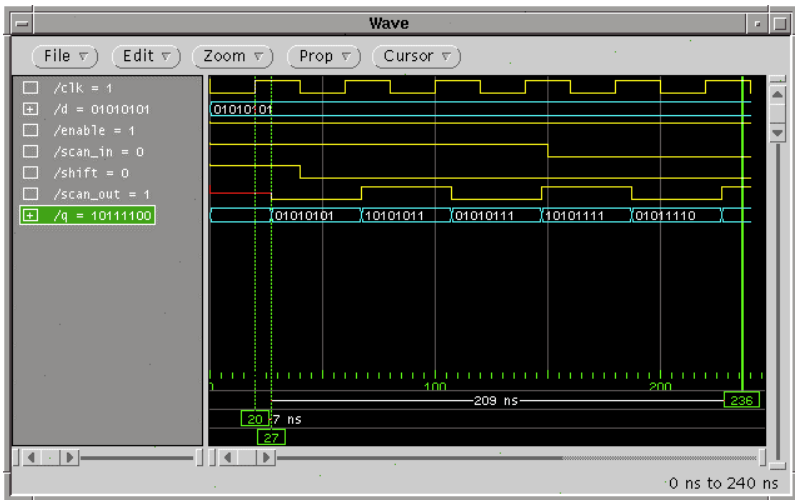
```
ARCHITECTURE structural OF shift_reg8_str IS

  -- 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.

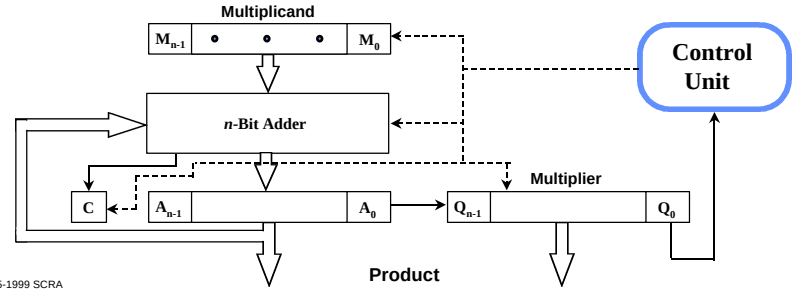
Structural 8 Bit Shift Register Simulation Results



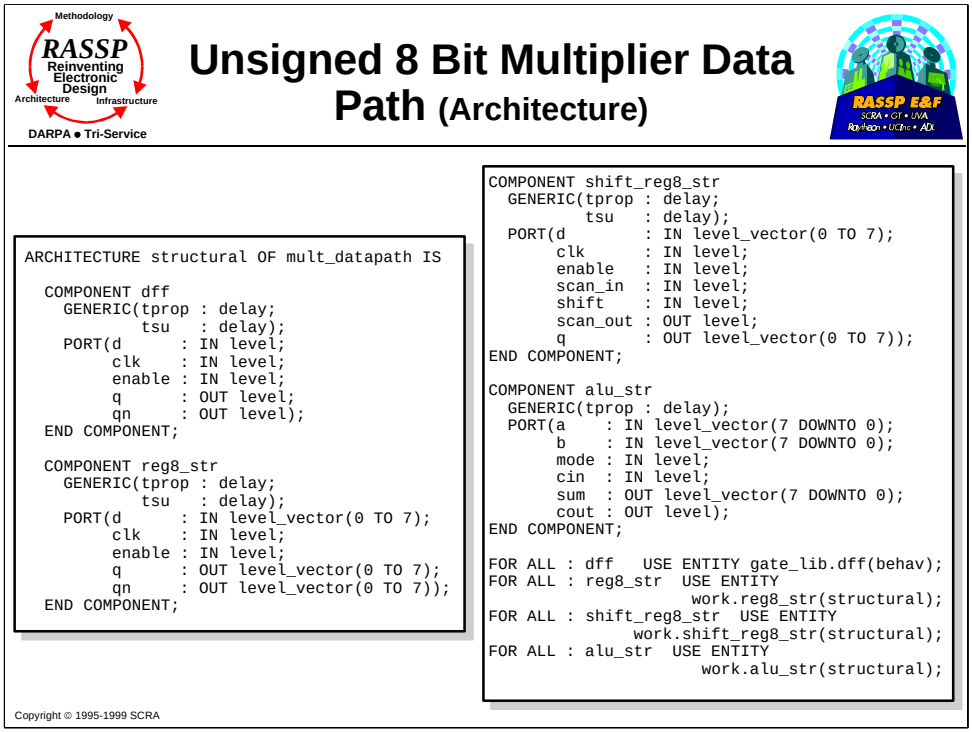
Here is the simulation results showing a parallel load followed by scanning the loaded data.

Unsigned 8 Bit Multiplier Data Path (Entity)

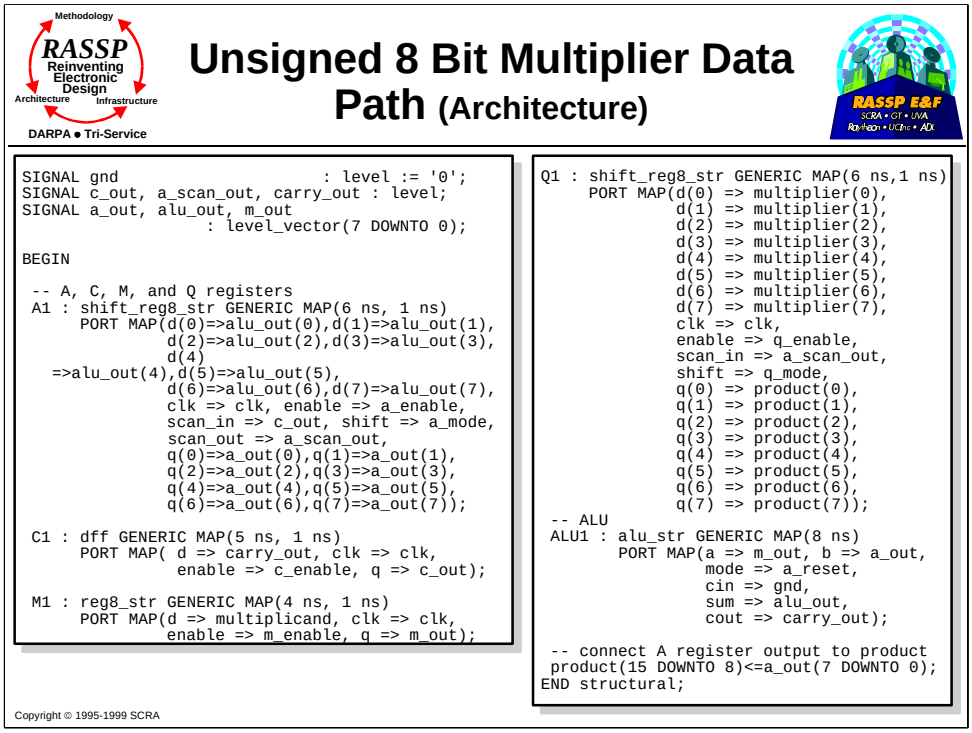
```
LIBRARY work;
LIBRARY gate_lib;
USE gate_lib.resources.all;
ENTITY mult_datapath IS
  PORT(multiplicand : IN level_vector(7 DOWNTO 0);
       multiplier   : IN level_vector(7 DOWNTO 0);
       a_enable     : IN level;  -- clock enable for A register
       a_reset      : IN level;  -- Reset control 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;
```



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.



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



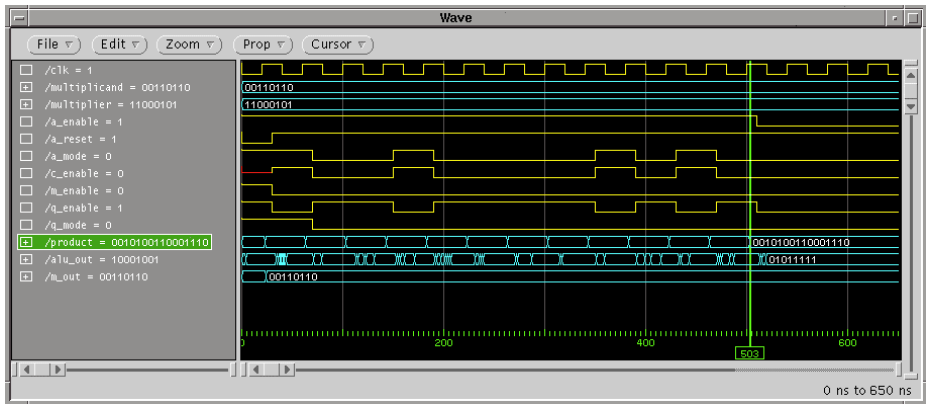
This is the component instantiations for the datapath. The mapping of individual bits of the *D* and *Q* input and output of the shift registers is necessary to reverse the inputs and outputs. Recall that the shift register was a “shift up” type where the *scan_in* input goes to *D*(0) and *D*(0) to *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.

Unsigned 8 Bit Multiplier Data Path Simulation Results



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**

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

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., Gadient, 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.