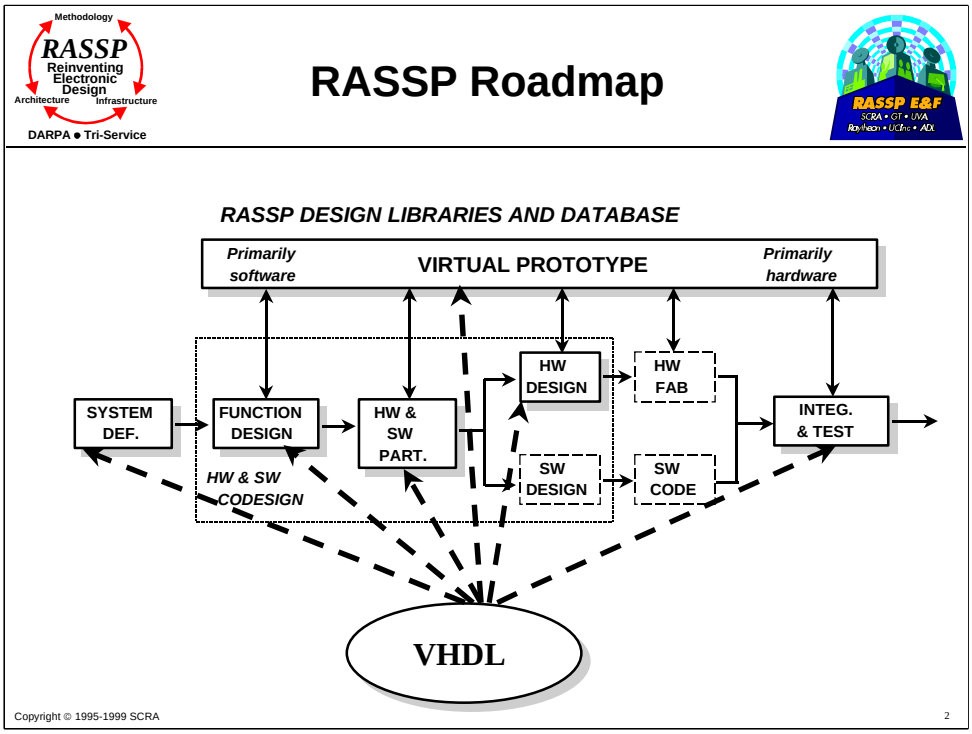
Basic VHDL

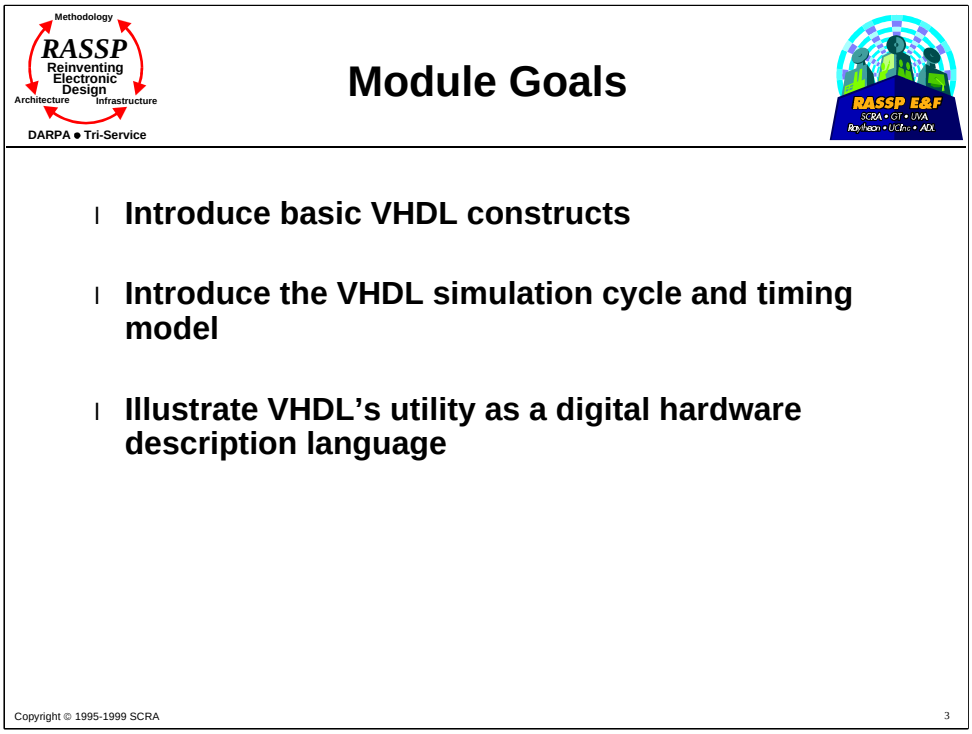
RASSP Education & Facilitation
Module 10

Version 3.00



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, 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 provide an introduction to the basic concepts and constructs of VHDL. VHDL is a versatile hardware description language which is useful for modeling electronic systems at various levels of design abstraction. Although most of the language will be touched on in this module, subsequent modules will cover specific areas of VHDL more thoroughly.

Specifically, areas to be covered in this module include:

-- the VHDL timing model

-- VHDL entities, architectures, and packages

-- Concurrent and sequential modes of execution

The goal of this module is to provide a basic understanding of VHDL fundamentals in preparation for the material to be covered in the subsequent VHDL modules.

Module Outline

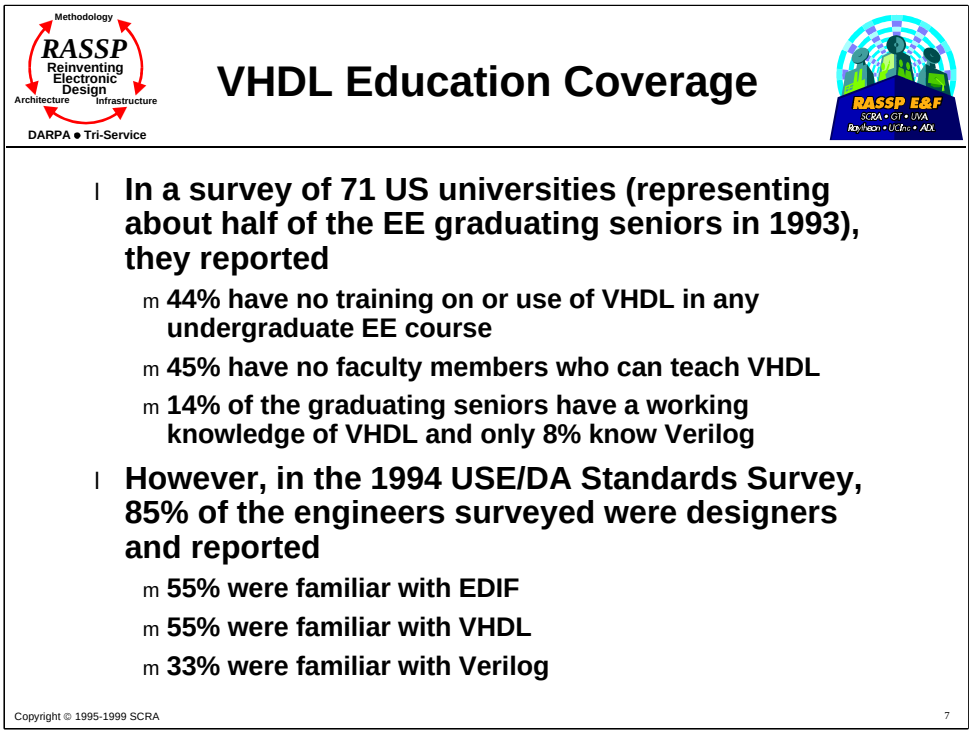
- Introduction
- VHDL Design Example
- VHDL Model Components
  - Entity Declarations
  - Architecture Descriptions
  - Timing Model

Module Outline (Cont.)

- **Basic VHDL Constructs**
  - **Data types**
  - **Objects**
  - **Sequential and concurrent statements**
  - **Packages and libraries**
  - **Attributes**
  - **Predefined operators**
- **Examples**
- **Summary**

Module Outline

- **Introduction**
- **VHDL Design Example**
- **VHDL Model Components**
- **Basic VHDL Constructs**
- **Examples**
- **Summary**

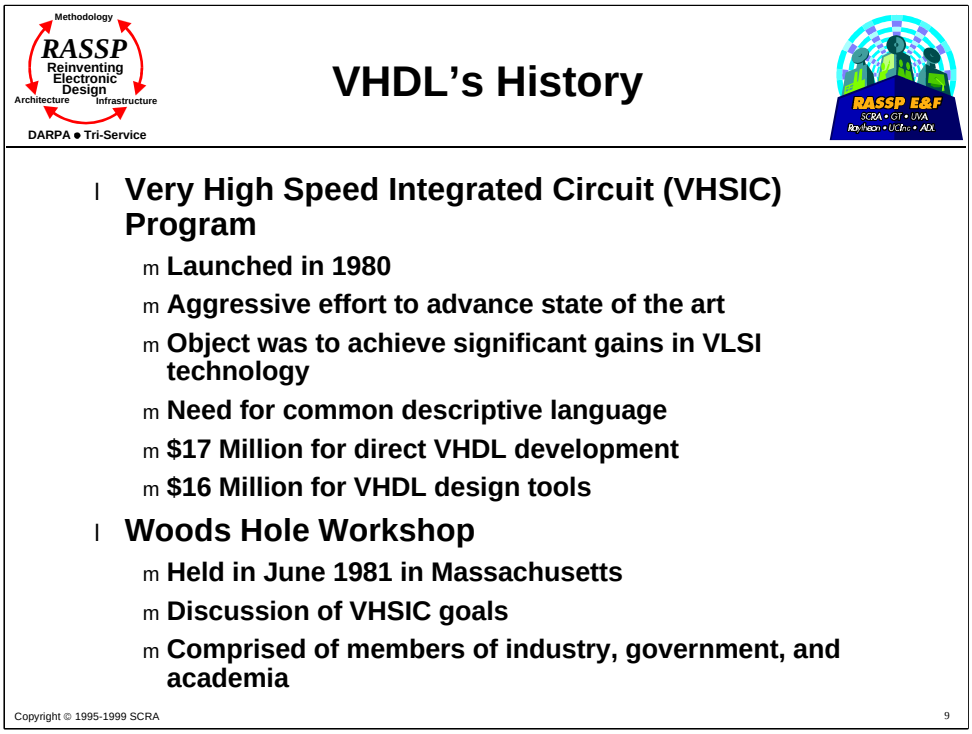


VHDL is an IEEE and U.S. Department of Defense standard for electronic system descriptions. It is also becoming increasingly popular in private industry as experience with the language grows and supporting tools become more widely available. Therefore, to facilitate the transfer of system description information, an understanding of VHDL will become increasingly important. This module provides a first step towards developing a basic comprehension of VHDL.

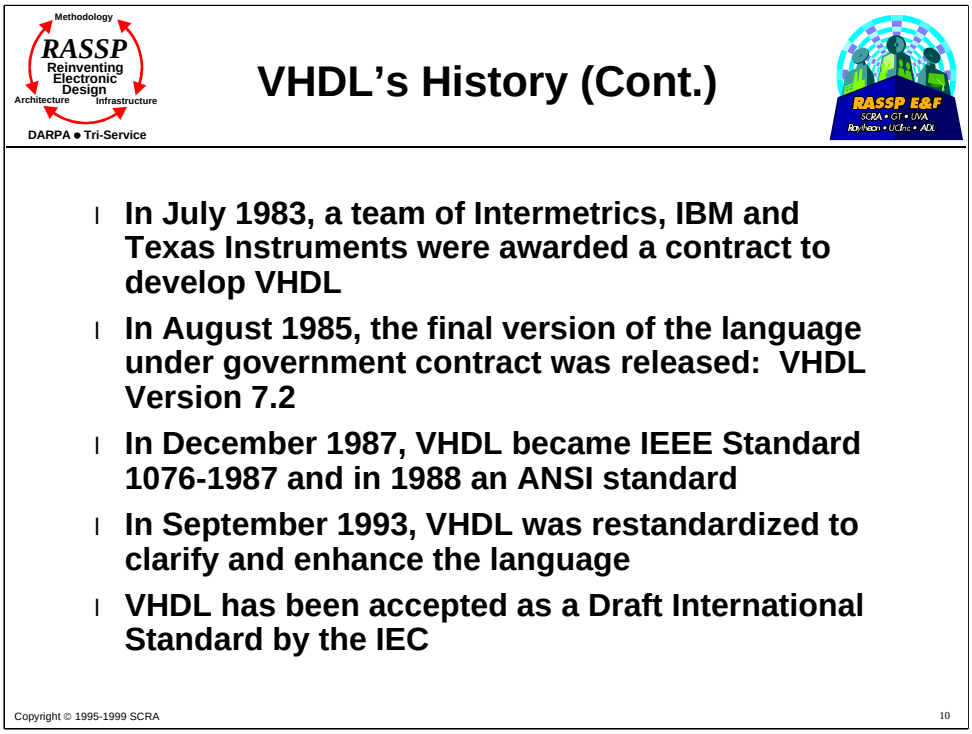
[VI93, USE/DA94]

Reasons for Using VHDL

- VHDL is an international IEEE standard specification language (IEEE 1076-1993) for describing digital hardware used by industry worldwide
- VHDL is an acronym for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
- VHDL enables hardware modeling from the gate to system level
- VHDL provides a mechanism for digital design and reusable design documentation

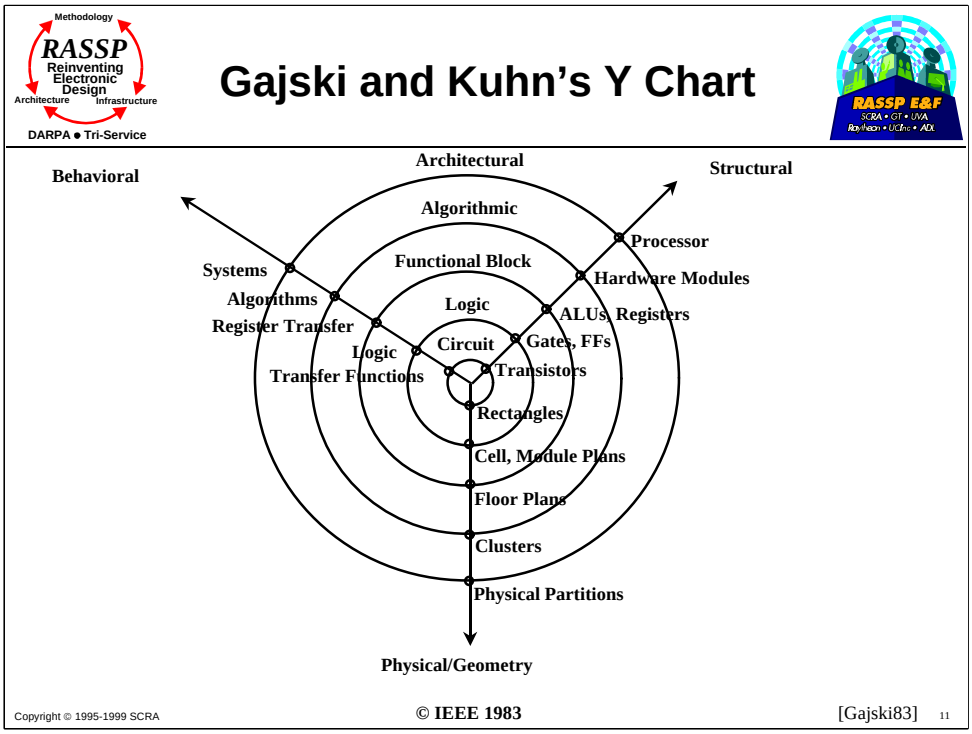


The Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) is the product of a US Government request for a new means of describing digital hardware. The Very High Speed Integrated Circuit (VHSIC) Program was an initiative of the Defense Department to push the state of the art in VLSI technology, and VHDL was proposed as a versatile hardware description language.



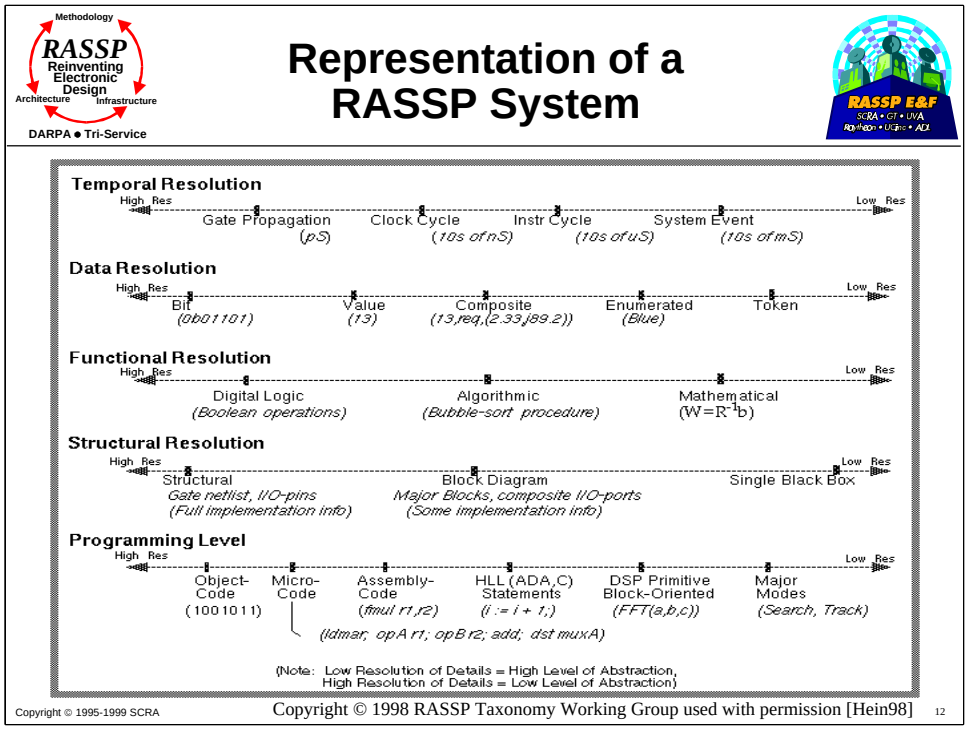
The contract for the first VHDL implementation was awarded to the team of Intermetrics, IBM, and Texas Instruments in July 1983. However, development of the language was not a closed process and was subjected to public review throughout the process (accounting for Versions 1 through 7.1). The final version of the language, developed under government contract, was released as VHDL Version 7.2.

In March 1986, IEEE proposed a new standard VHDL to extend and modify the language to fix identified problems. In December 1987, VHDL became IEEE Standard 1076-1987. VHDL was again modified in September 1993 to further refine the language. These refinements both clarified and enhanced the language. The major changes included much improved file handling and a more consistent syntax and resulted in VHDL Standard 1076-1993.



VHDL allows the designer to work at various levels of abstraction. Many of the levels are shown pictorially in the Gajski/Kuhn chart. Although VHDL does not support system description at the physical/geometry level of abstraction, many design tools can take behavioral or structural VHDL and generate chip layouts.

As an illustrative example, the next few slides will show a sample VHDL design process to demonstrate how a designer can move from an algorithmic behavioral description, to a register transfer (or dataflow) description, to a gate level description. See [Gajski83], [Walker85] and [Smith88].

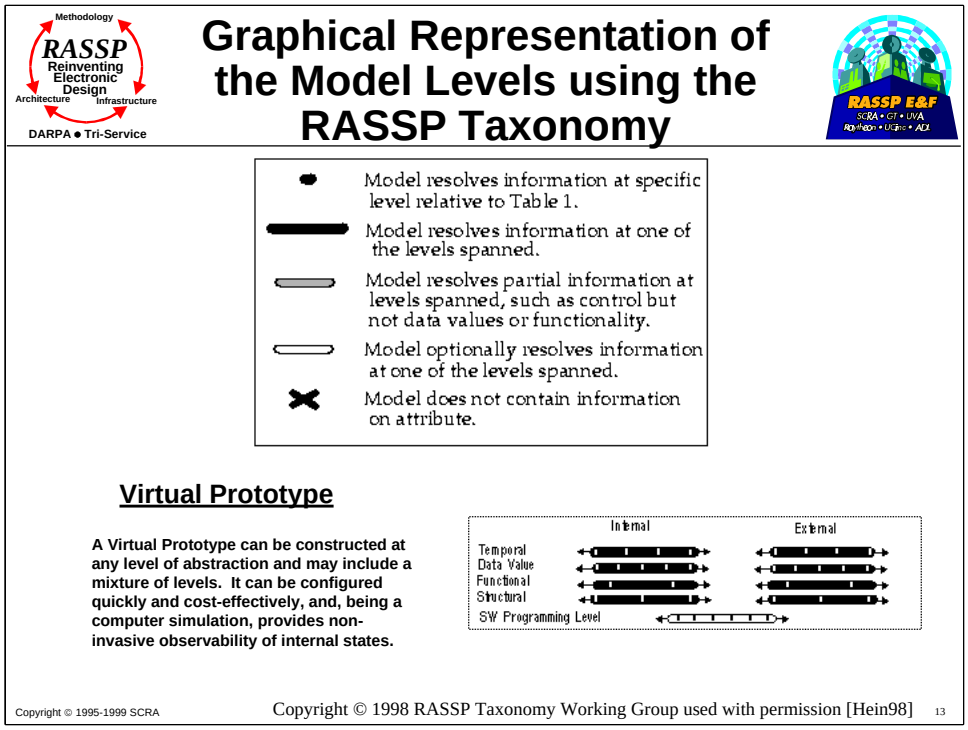


Alternatively, the various forms of system representation could may be described using the taxonomy developed under the RASSP project. This taxonomy uses five axes which represent:

- Time
- Data value
- Function
- Structure
- Programmability

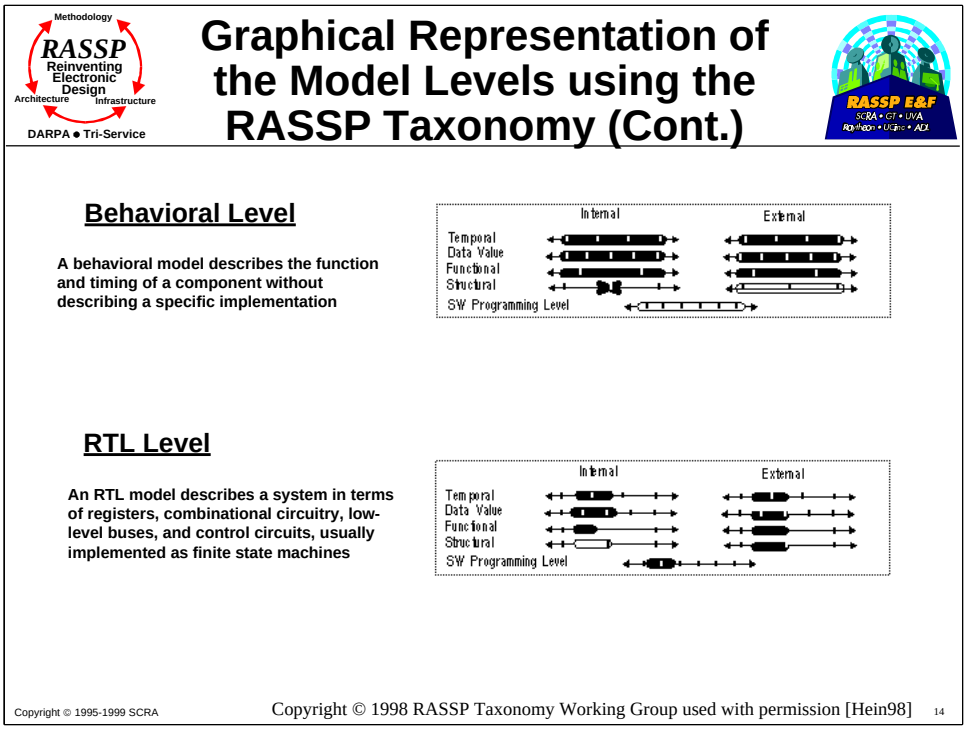
Note that both internal and external behavior is represented.

For example, time can be represented from low to high resolution. At the high level, we represent gates and at the low level purely functions (i.e. no timing).



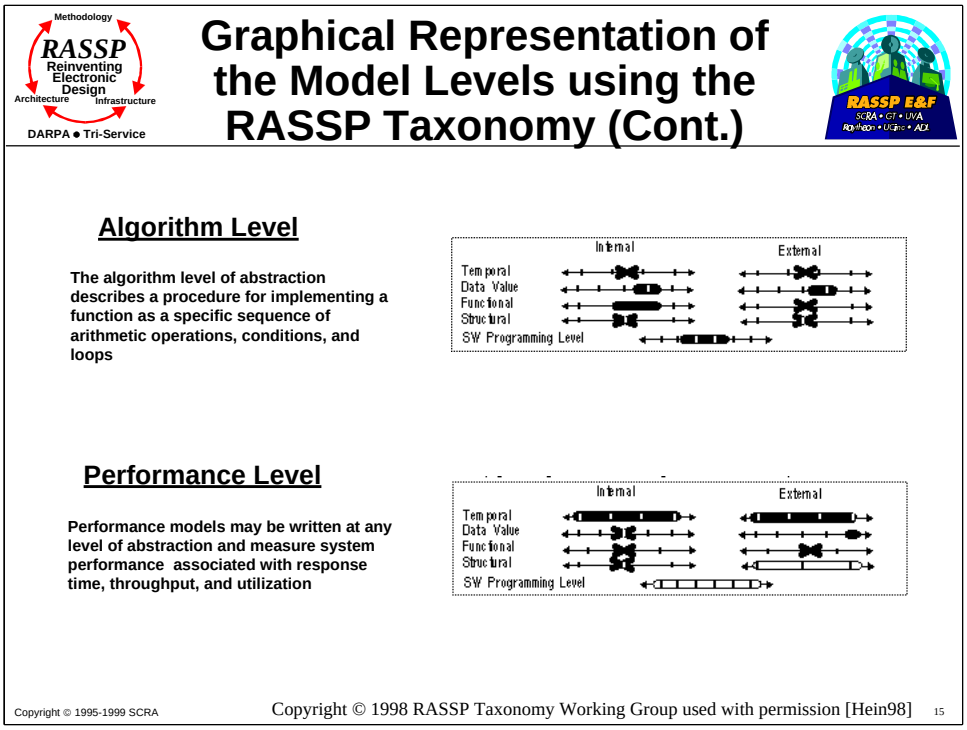
Using the above key code to utilize the RASSP taxonomy, a number of example modeling levels are presented in this and subsequent few slides.

A virtual prototype, for example, can be developed at any temporal, data value, functional, or structural level.



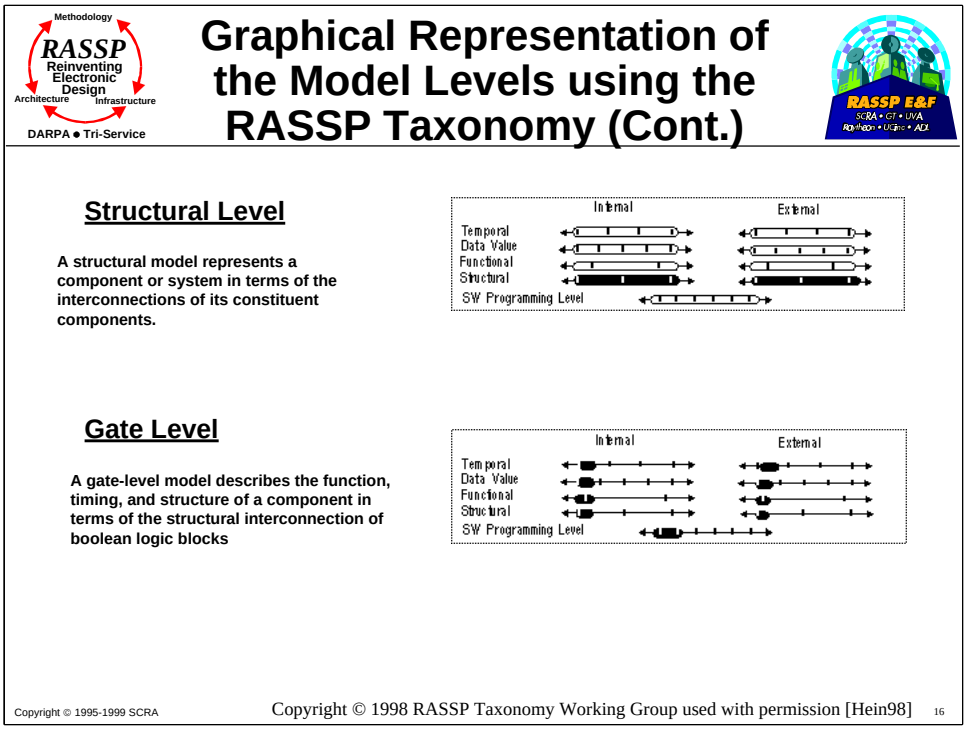
Behavior can be described as above. Note that structure is not maintained internally and that timing, function, and data values can be modeled at any level and is case dependent.

RTL and gate level models specify more of the details of a design and hence tend to be at the higher end of the resolution scales in all categories.

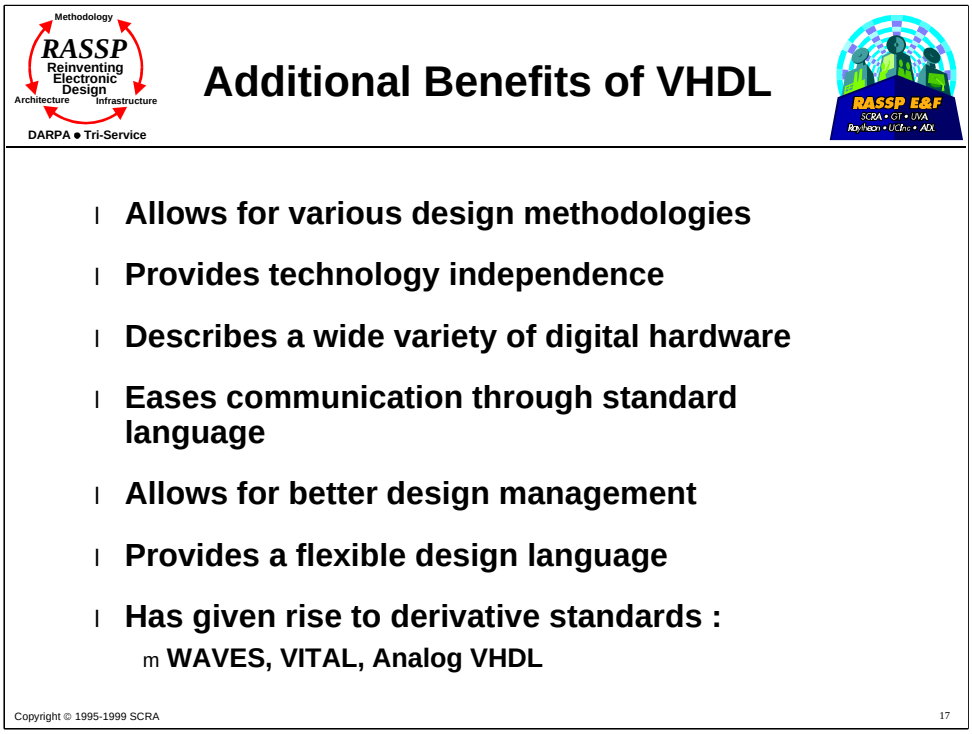


At the algorithm level, a model does not contain timing information, either internally or externally. At this level, data is represented at the functional level where function is captured internally, but no external interface is modeled. The structure is not defined, and SW programmability is at the DSP primitive level (i.e. FFT, FIR, etc..) e.g. using Matlab.

Performance models mainly measure the time effects of the system such as throughput, latency, and utilization



The gate level model includes structure at the higher resolution.



VHDL is a powerful and versatile language and offers numerous advantages:

1) Design Methodology: VHDL supports many different design methodologies (top-down, bottom-up, delay of detail) and is very flexible in its approach to describing hardware.

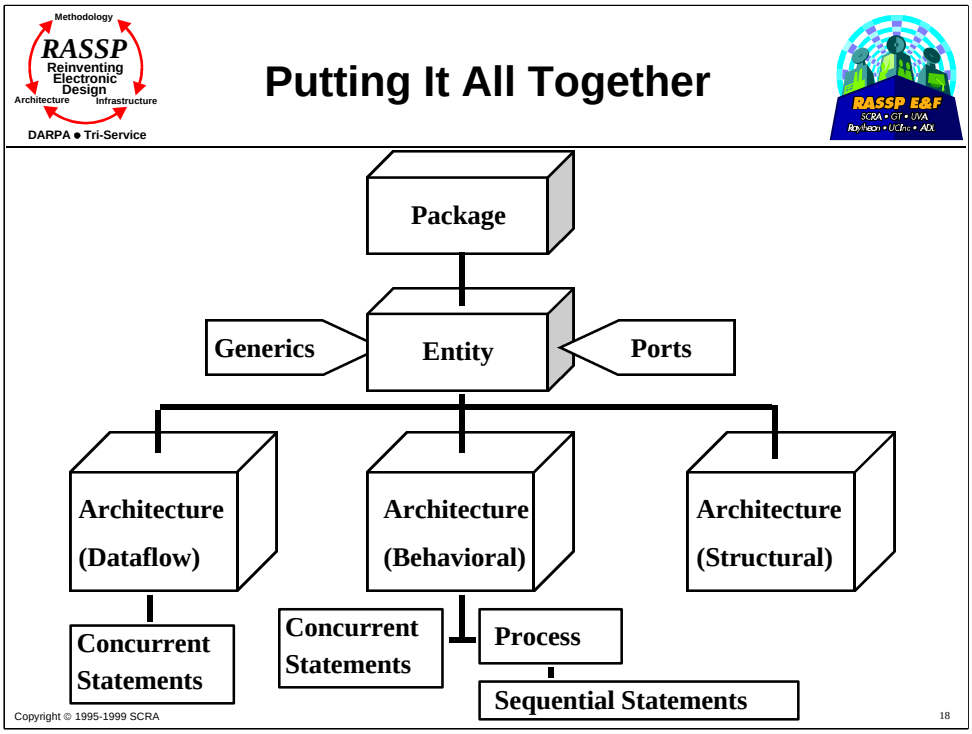
2) Technology Independence: VHDL is independent of any specific technology or process. However, VHDL code can be written and then targeted for many different technologies.

3) Wide Range of Descriptions: VHDL can model hardware at various levels of design abstraction. VHDL can describe hardware from the standpoint of a "black box" to the gate level. VHDL also allows for different abstraction-level descriptions of the same component and allows the designer to mix behavioral descriptions with gate level descriptions.

4) Standard Language: The use of a standard language allows for easier documentation and the ability to run the same code in a variety of environments. Additionally, communication among designers and among design tools is enhanced by a standard language.

5) Design Management: Use of VHDL constructs, such as packages and libraries, allows common elements to be shared among members of a design group.

6) Flexible Design: VHDL can be used to model digital hardware as well as many other types of systems, including analog devices.



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 component's external view, sometimes referred to as its "pins".

The architecture(s) describe(s) 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 "configuration statement" which binds the entity of a sub-component to one of 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 for communicating static information to an architecture from the external environment. They are passed through the entity construct.

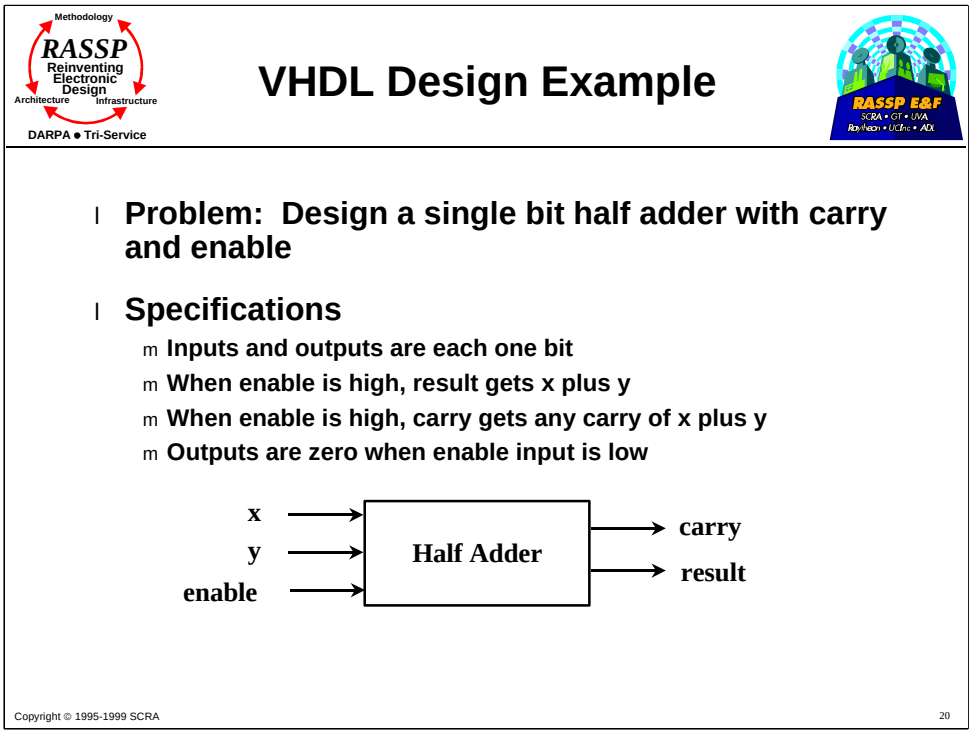
Ports provide the mechanism for a device to communicate with its environment. A port declaration defines the names, types, directions, and possible default values for the signals in a component's interface.

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 constructs from all three of the types described above. Structural VHDL concepts are used to connect the model's various components together. Dataflow and behavioral concepts are often used to provide the simulation's start/stop conditions, or other desired modeling directives.

This slide will be used again at the end of this module as a review.

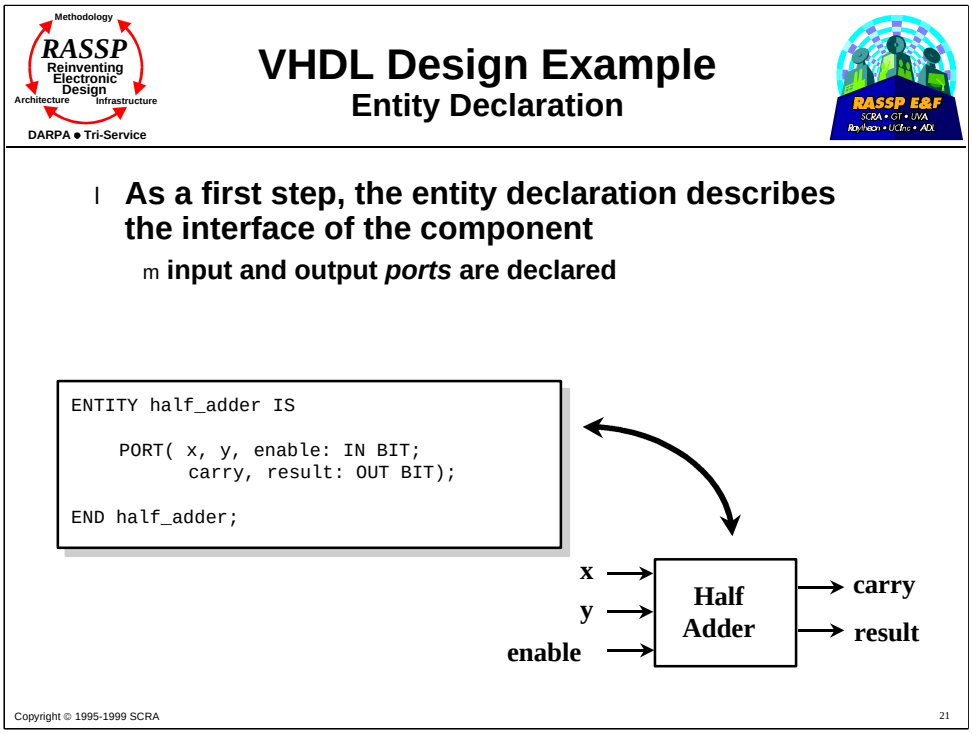
Module Outline

- Introduction
- **VHDL Design Example**
- VHDL Models of Hardware
- Basic VHDL Constructs
- Examples
- Summary



As a first example, we will consider the design a single bit adder with carry and enable functions. When the enable line is low, the adder is to place zeroes on its outputs.

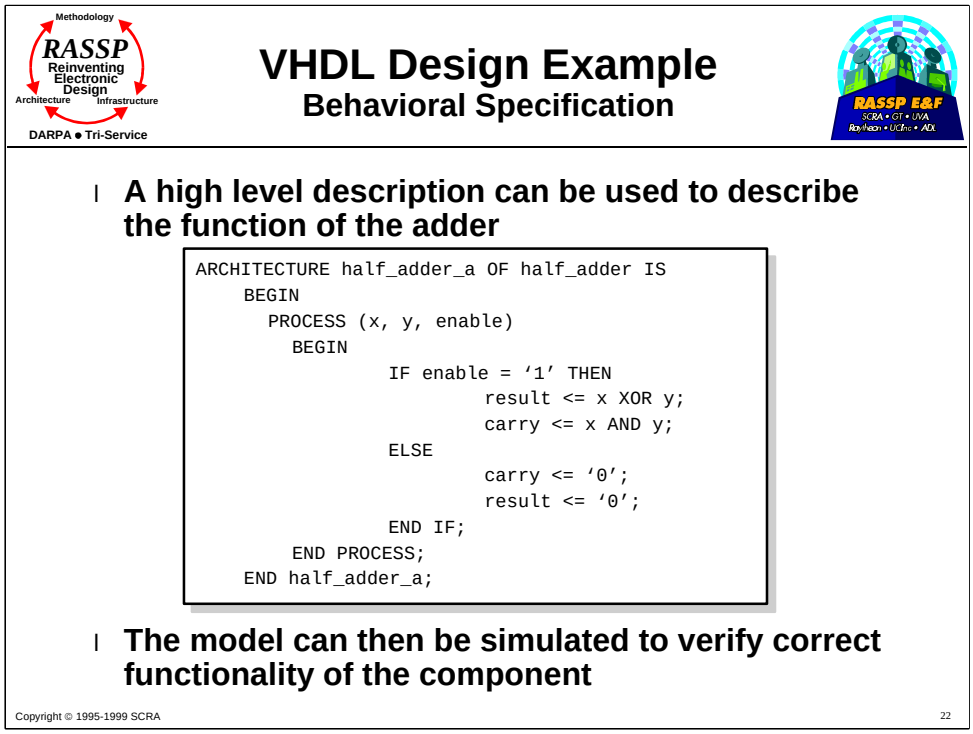
This sample design sequence is based on an example in [Navabi93].



In this slide the term *entity* refers to the VHDL construct in which a component's interface (which is visible to other components) is described. The first line in an entity declaration provides the name of the entity.

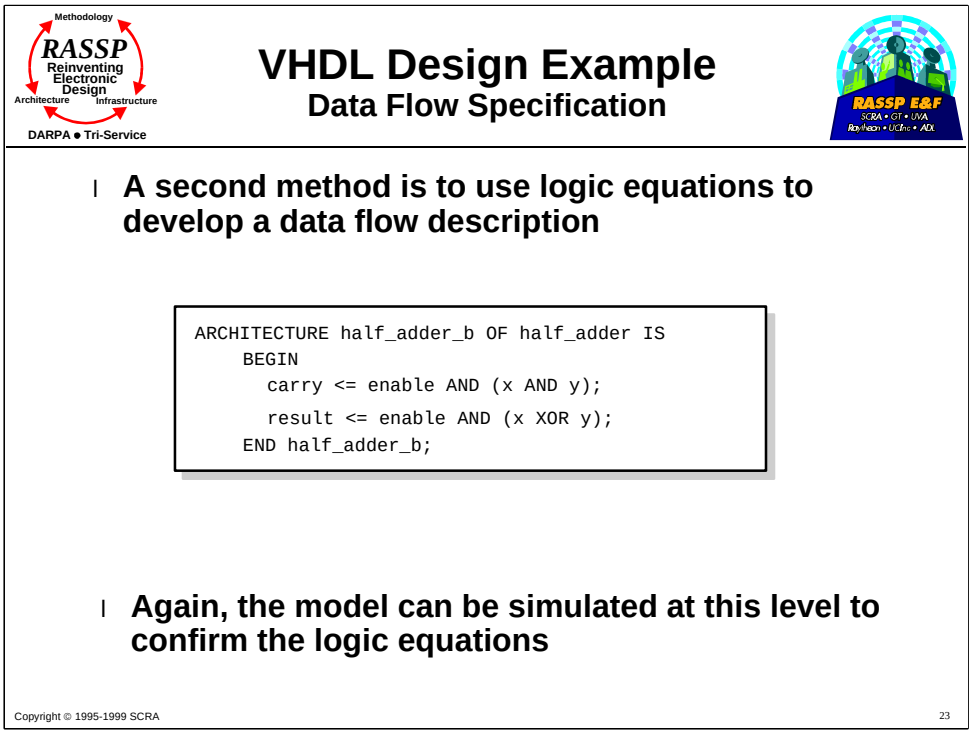
Next, the PORT statement indicates the actual interface of the entity. The port statement lists the signals in the component's interface, the direction of data flow for each signal listed, and type of each signal. In the above example, signals x, y, and enable are of direction IN (i.e. inputs to this component) and type bit, and carry and result are outputs also of type bit. Notice that if signals are of the same mode and type, they may be listed on the same line.

Particular attention should be paid to the syntax in that no semicolon is required before the closing parenthesis in the PORT declaration (or GENERIC declaration, for that matter, which is not shown here). The entity declaration statement is closed with the END keyword, and the name of the entity is optionally repeated.



In the first stage of the design process, a high level behavior of the adder is considered. This level uses abstract constructs (such as the IF-THEN-ELSE statement) to make the model more readable and understandable.

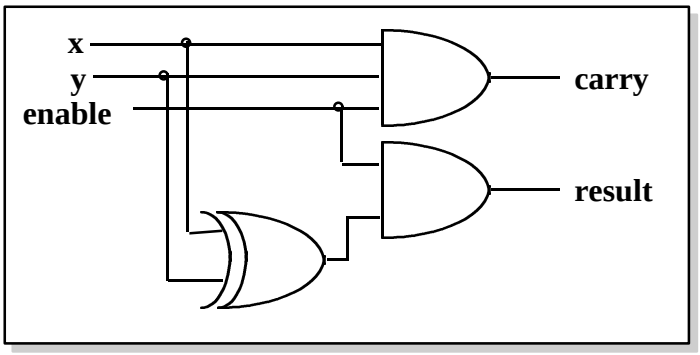
Simulation of the adder at this level shows correct understanding of the problem specifications of the adder. VHDL code for this adder will be shown later.



An alternative method for describing the functionality of the component is to use data flow specifications with concurrent signal assignment statements. An example of this is shown in the representation here which uses logic equations to describe functionality of the *carry* and *result* outputs. Notice that the sequential IF-THEN-ELSE statement cannot be used here (i.e. outside a *process*).

VHDL Design Example
Structural Specification

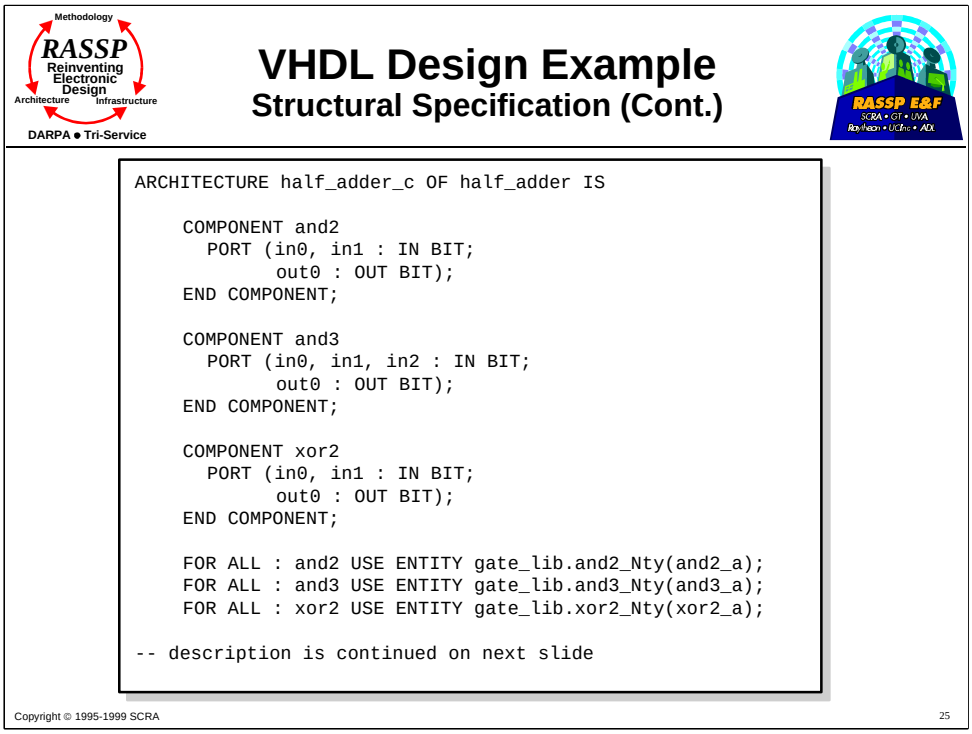
- **As a third method, a structural description can be created from predescribed components**



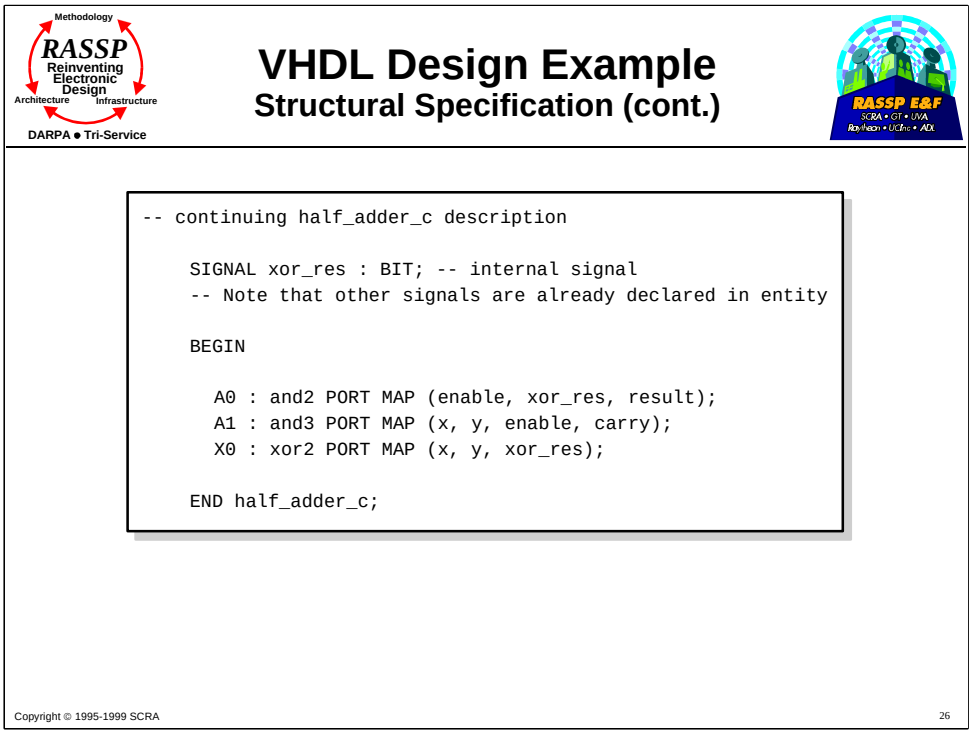
- **These gates can be pulled from a library of parts**

The previous data flow description maps easily to logic gates. The models for these gates can come from many different libraries, which may represent specific implementation technologies, for example.

VHDL structural descriptions may similarly be used to describe the interconnections of high level components (e.g. multiplexors, full adders, microprocessors).



The VHDL structural description for this example is shown in this and the next slide. A number of locally defined idealized *components* are declared in this slide. These components are then *bound* to VHDL entities found in a library called *gate_lib*.

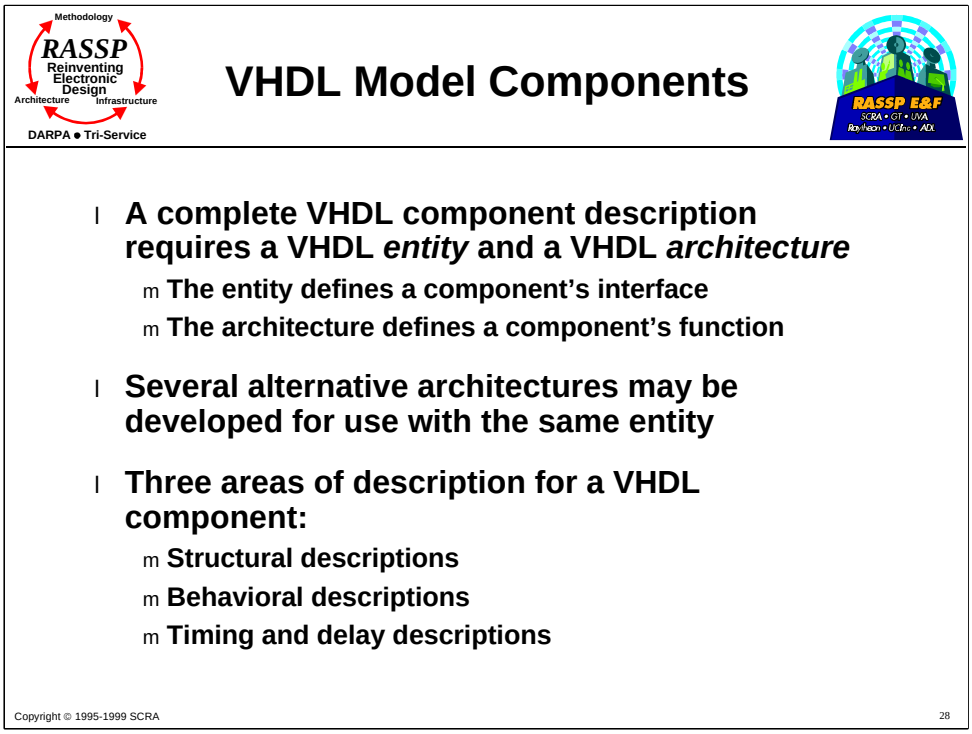


This second slide of the structural description shows the declaration of a local signal to be used in connecting the components together.

Finally, the body of the architecture shows the component *instantiations* and how they are interconnected to each other and the outside world via the attaching of signals in their PORT MAPs.

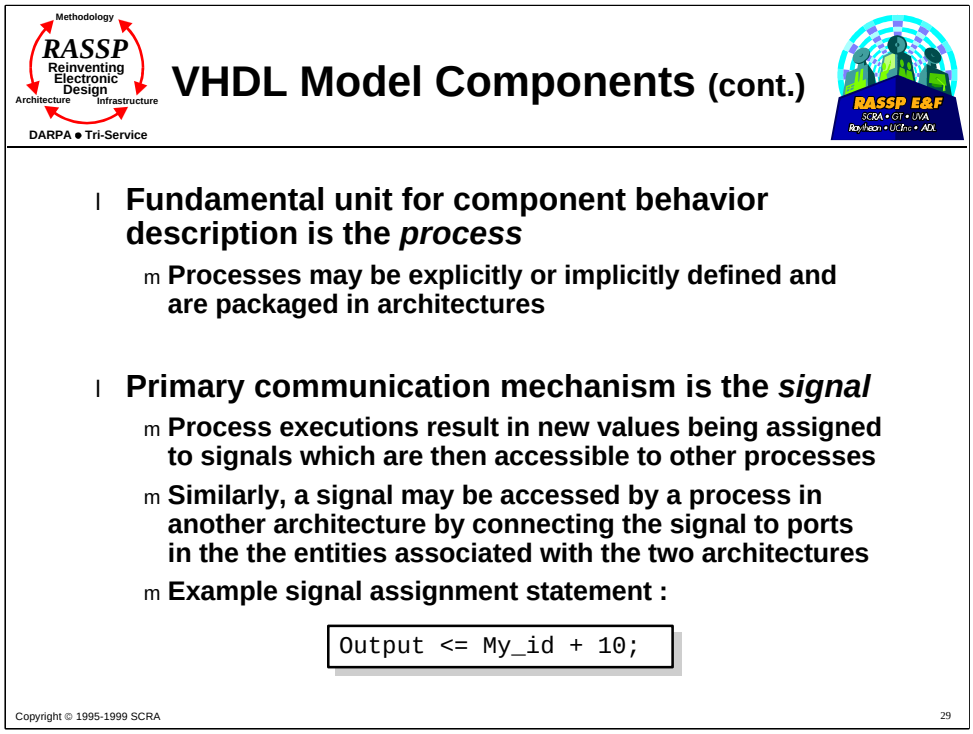
Module Outline

- Introduction
- VHDL Design Example
- **VHDL Model Components**
  - Entity Declarations
  - Architecture Descriptions
  - Timing Model
- Basic VHDL Constructs
- Examples
- Summary



A complete VHDL description consists of an *entity* in which the interface signals are declared and an *architecture* in which the functionality of the component is described.

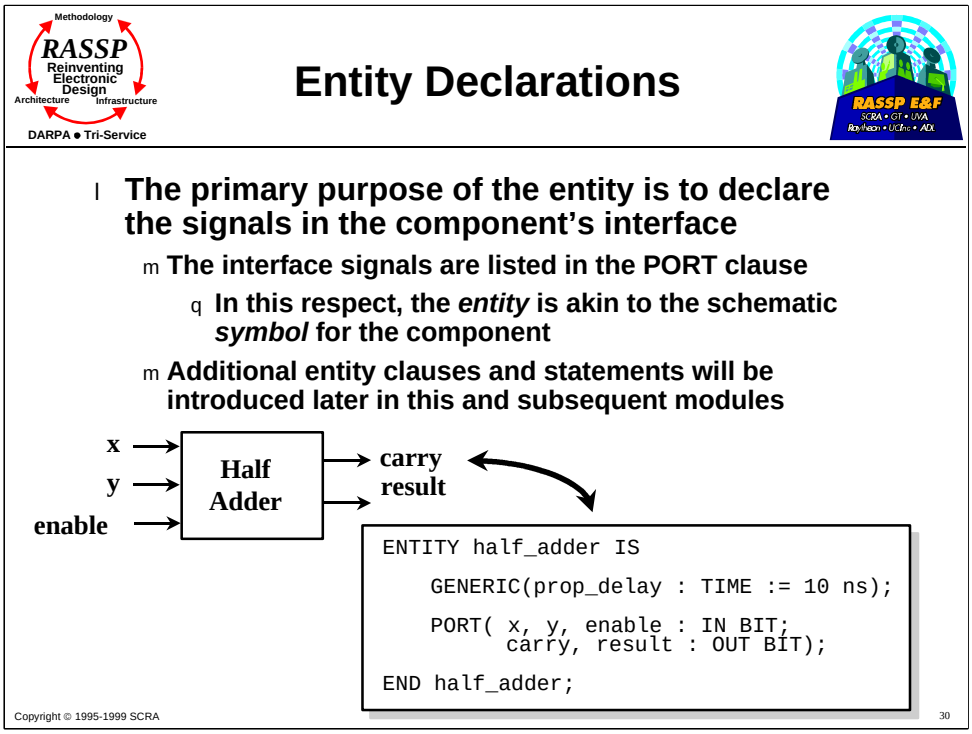
VHDL provides constructs and mechanisms for describing the structure of components that may be constructed from simpler sub-systems. VHDL also provides some high-level description language constructs (e.g. variables, loops, conditionals) to model complex behavior easily. Finally, the underlying timing model in VHDL supports both the concurrency and delay observed in digital electronic systems.



The purpose of this slide is to provide a basic working definition for process and signal. These definitions are not intended to be comprehensive, and detail will be added throughout the presentation of this and subsequent modules.

All behavioral descriptions in VHDL are constructed using processes. Processes may be defined explicitly where complex behavior can be described in a sequential programming style, or they may be implicitly defined in concurrent signal assignment statements. Both of these mechanisms will be covered in more detail in this and subsequent modules.

The primary purpose of the process is to determine new values for signals. Signals are accessible to other processes, and, therefore, provide a mechanism for the results of one process execution to be communicated to other processes. Signals may be made accessible to processes within other VHDL architectures by connecting them to ports of the respective entities.

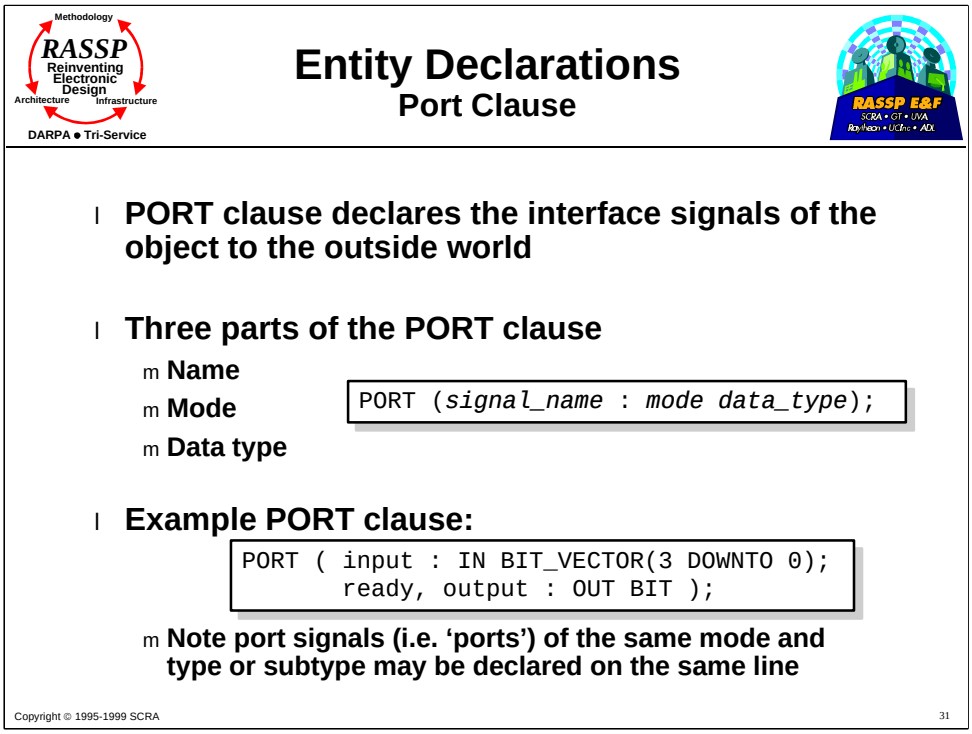


In this slide the term *entity* refers to the VHDL construct in which a component's interface (which is visible to other components) is described. The first line in an entity declaration provides the name of the entity.

Next, an optional GENERIC statement includes value assignments to parameters that may be used in the architecture descriptions of the component.

Following that, the PORT statement indicates the actual interface of the entity. The port statement lists the signals in the component's interface, the direction of data flow for each signal listed, and type of each signal. In the above example, signals x, y, and enable are of direction IN (i.e. inputs to this component) and type bit, and carry and result are outputs also of type bit. Notice that if signals are of the same mode and type, they may be listed on the same line.

Particular attention should be paid to the syntax in that no semicolon is required before the closing parenthesis in PORT or GENERIC declarations. The entity declaration statement is closed with the END keyword, and the name of the entity is optionally repeated.

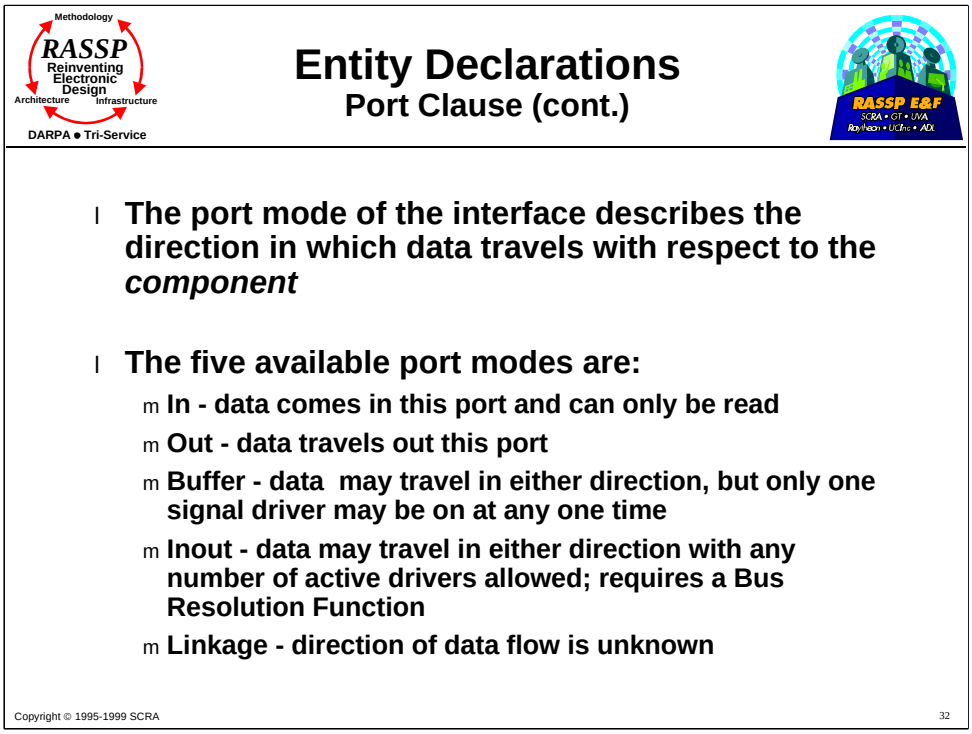


The PORT declaration describes the interface of the component. There are three essential elements to the port declaration: the name, mode, and type of the signals in the interface. An optional fourth element (not shown above) in a port declaration is the signal's initial value which will be assigned to the signal as a default if there are no active drivers on the signal at the start of a simulation.

Note that signals declared in an entity's port declaration may sometimes be referred to as ports.

In the example port declaration above, *input* is an input and can only be read by the device. The ports *ready* and *output* are outputs so that the signals are "driven" by this component. Note that according to the VHDL Standard, a component may not read its own OUT ports.

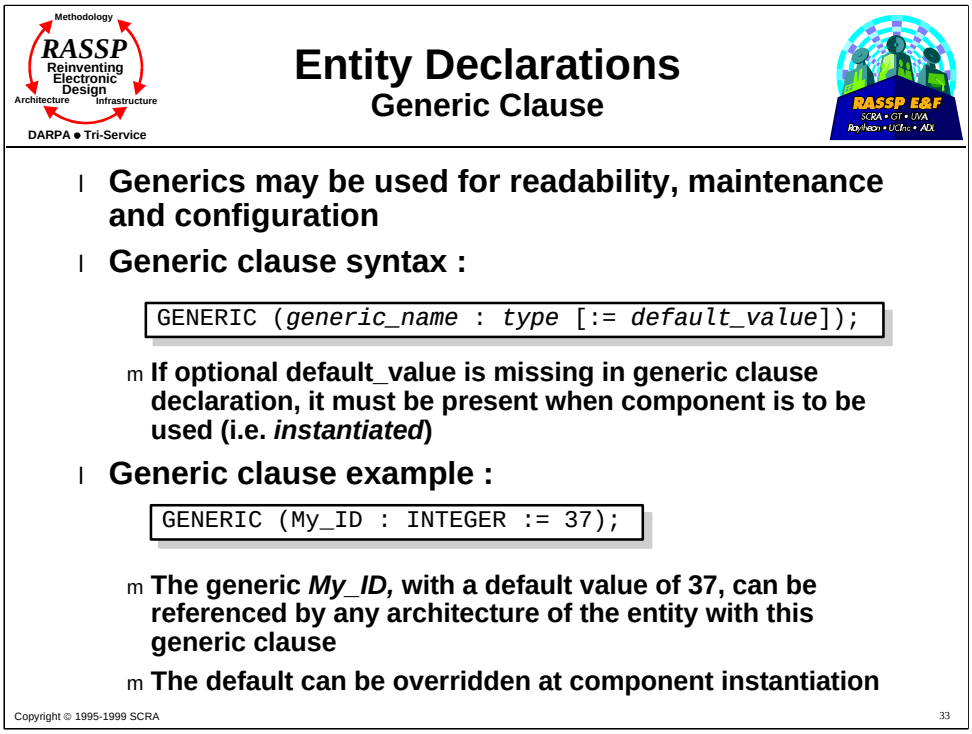
Finally, the port must indicate the type or subtype for port signals. Any VHDL-defined standard or user-defined type or subtype may be used in a port declaration. Note that a range specification may be declared if an unconstrained type is used in the type declaration.



The mode indicates the direction of the flow of data across that port. This flow of data is defined with respect to the component.

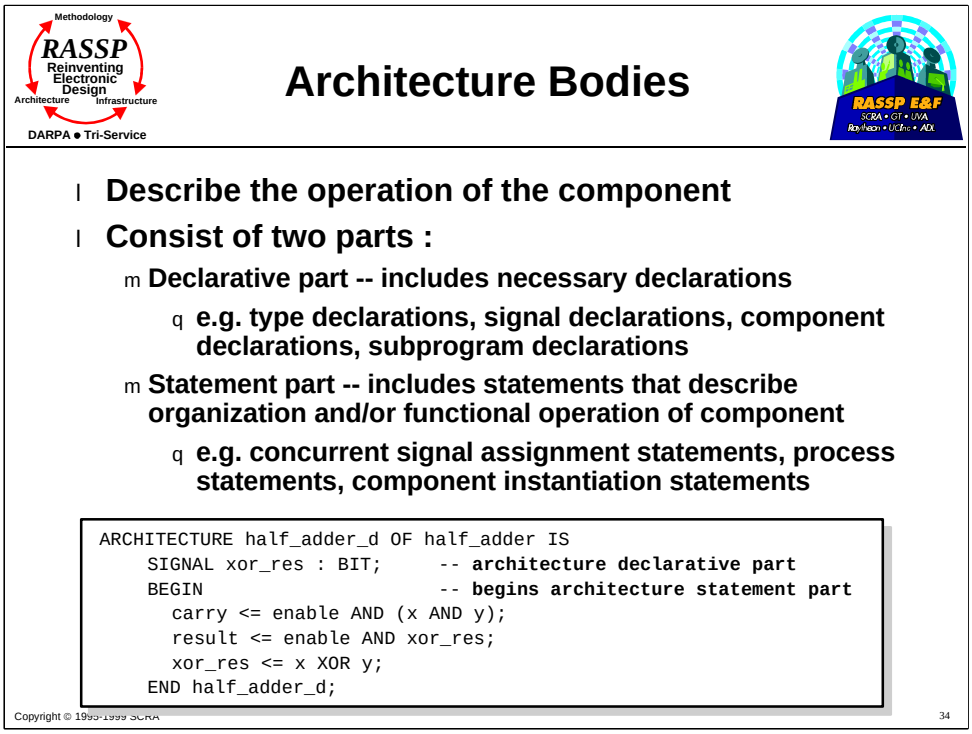
The five port modes available:

1. IN -- indicates that the (only) signal driver is outside this component
2. OUT -- indicates that the (only) signal driver is within this component. Note that a component may not read its own OUT ports.
3. BUFFER -- indicates that there may be signal drivers inside and outside this component. However, only one of these drivers can be driving the signal at any one time.
4. INOUT -- indicates that there may be signal drivers inside and outside this component. Any number of these drivers can be driving the signal simultaneously, but a Bus Resolution Function is then required to determine what values the signal will assume.
5. LINKAGE -- indicates that the location of the signal drivers is not known. This mode type only indicates that a connection exists.



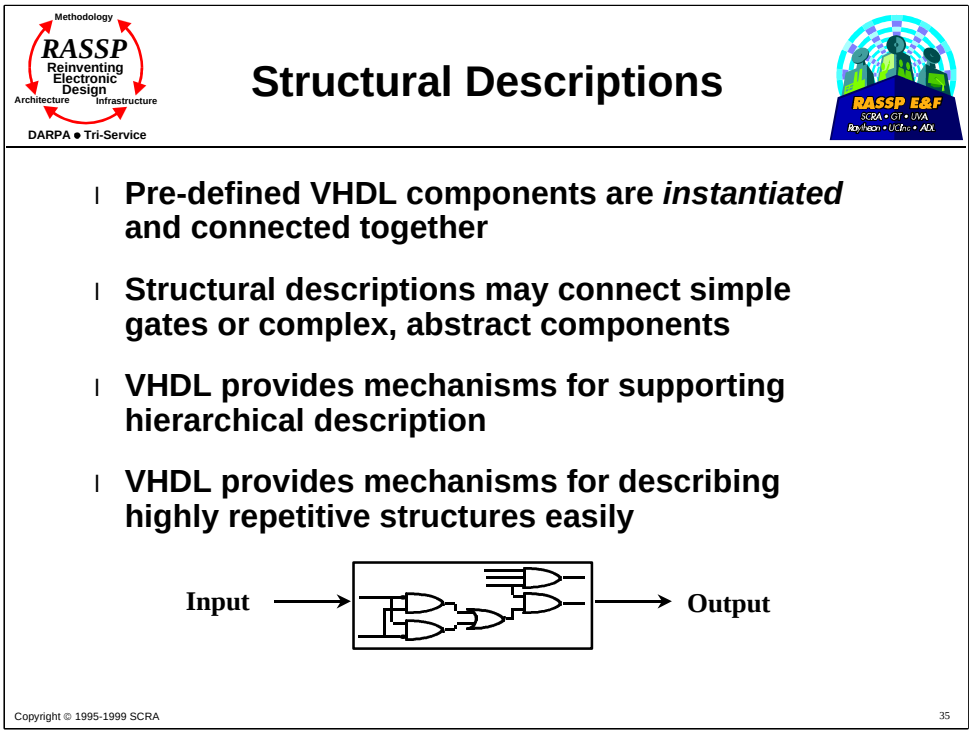
GENERIC statements create parameters to be passed on to the architectures of this entity. These parameters may be used to characterize the component by setting propagation delay, component ids, etc.

Generics are discussed further in the Structural VHDL module.



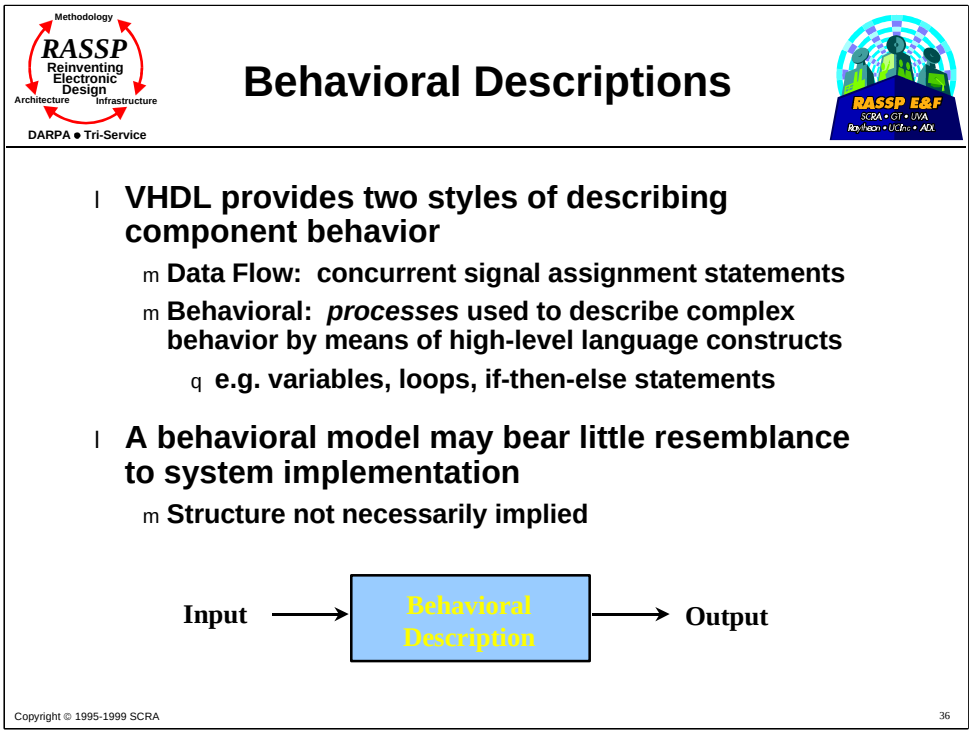
The architecture body describes the operation of the component. There can be many different architectures described for each entity. However, for each instantiation of the entity, one of the possibly several architectures must be selected.

In the above example, the architecture body starts with the keyword ARCHITECTURE followed by the name of the architecture (e.g. *half_adder_d* above) and the name of the entity with which the architecture is associated. The keyword BEGIN marks the beginning of the architecture statement part which may include concurrent signal assignment statements and processes. Any signals that are used internally in the architecture description but are not found in the entity's ports are declared in the architecture's declarative part before the BEGIN statement of the architecture body. The keyword END marks the end of the architecture body.



VHDL also supports descriptions based on a component's underlying internal structure. Structural VHDL allows for sub-components to be instantiated and interconnected. It may be noted that a structural description is similar to a netlist. Of course, the description of the sub-components can themselves be structural and/or behavioral in nature.

RASSP E&F Module 11, Structural VHDL, concentrates on VHDL constructs that support structural descriptions.

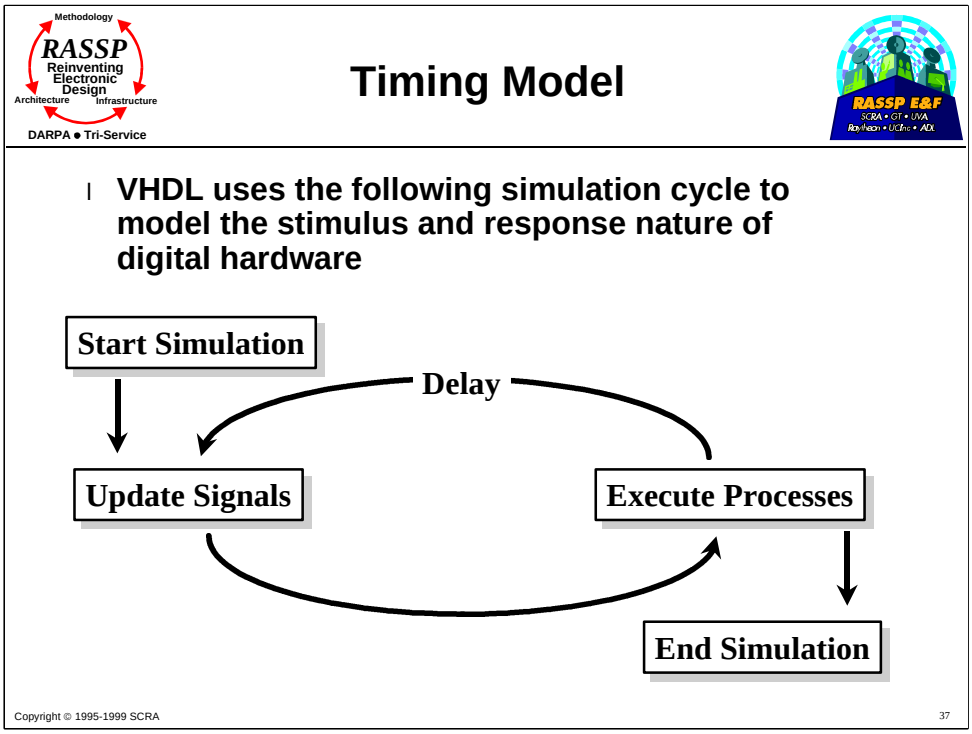


A behavioral description may be relatively abstract in that specific details about a component's internal structure need not be included in the description.

The fundamental unit of behavioral description in VHDL is the *process;* all processes are executed concurrently with each other. In fact, the data flow modeling style is a special case of the general VHDL process mechanism in that each concurrent signal assignment statement is actually a single statement VHDL process that executes concurrently with all other processes.

Within a process, VHDL provides a rich set of constructs to allow the description of complex behavior. This includes loops, conditional statements, variables to control maintaining state information within a process (e.g. loop counters), etc.

Much more information is provided in RASSP E&F Module 12, Behavioral VHDL, which concentrates on VHDL constructs that support behavioral descriptions.



The VHDL timing model controls the stimulus and response sequence of signals in a VHDL model. At the start of a simulation, signals with default values are assigned those values. In the first execution of the simulation cycle, all processes are executed until they reach their first *wait* statement. These process executions will include signal assignment statements that assign new signal values after prescribed delays.

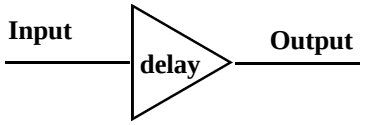
After all the processes are suspended at their respective wait statements, the simulator will advance simulation time just enough so that the first pending signal assignments can be made (e.g. 1 ns, 3 ns, 1 delta cycle).

After the relevant signals assume their new values, all processes examine their wait conditions to determine if they can proceed. Processes that can proceed will then execute concurrently again until they all reach their respective subsequent wait conditions.

This cycle continues until the simulation termination conditions are met or until all processes are suspended indefinitely because no new signal assignments are scheduled to unsuspend any waiting processes.

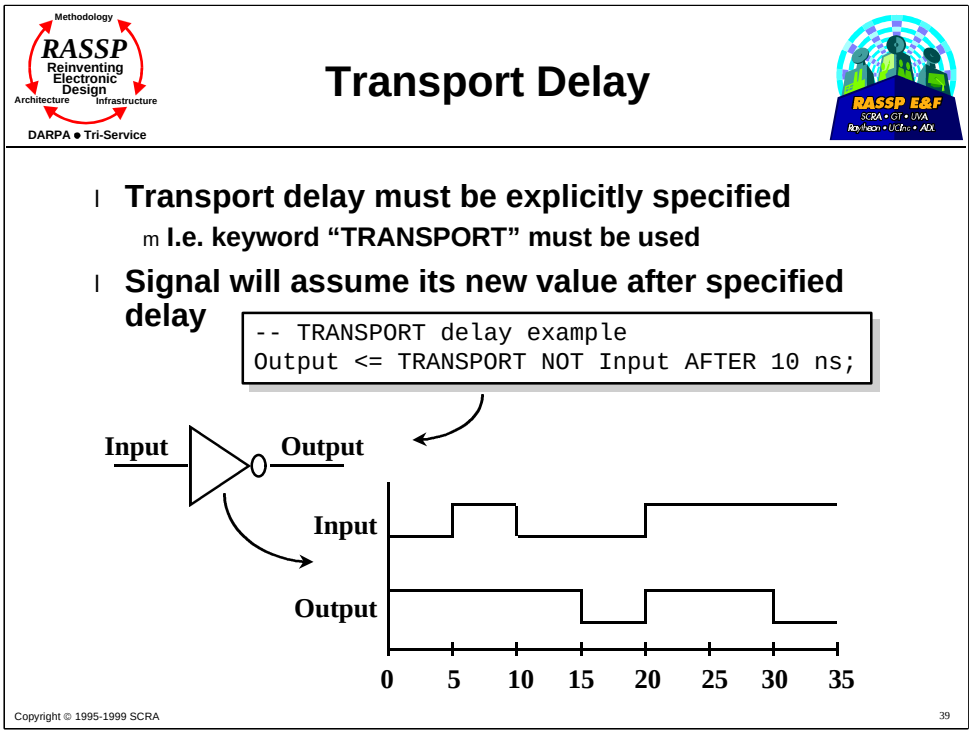
Delay Types

- **All VHDL signal assignment statements prescribe an amount of time that must transpire before the signal assumes its new value**
- **This prescribed delay can be in one of three forms:**
  - **Transport -- prescribes propagation delay only**
  - **Inertial -- prescribes propagation delay and minimum input pulse width**
  - **Delta -- the default if no delay time is explicitly specified**



There are several types of delay in VHDL, and understanding of how delay works in a process is key to writing and understanding VHDL.

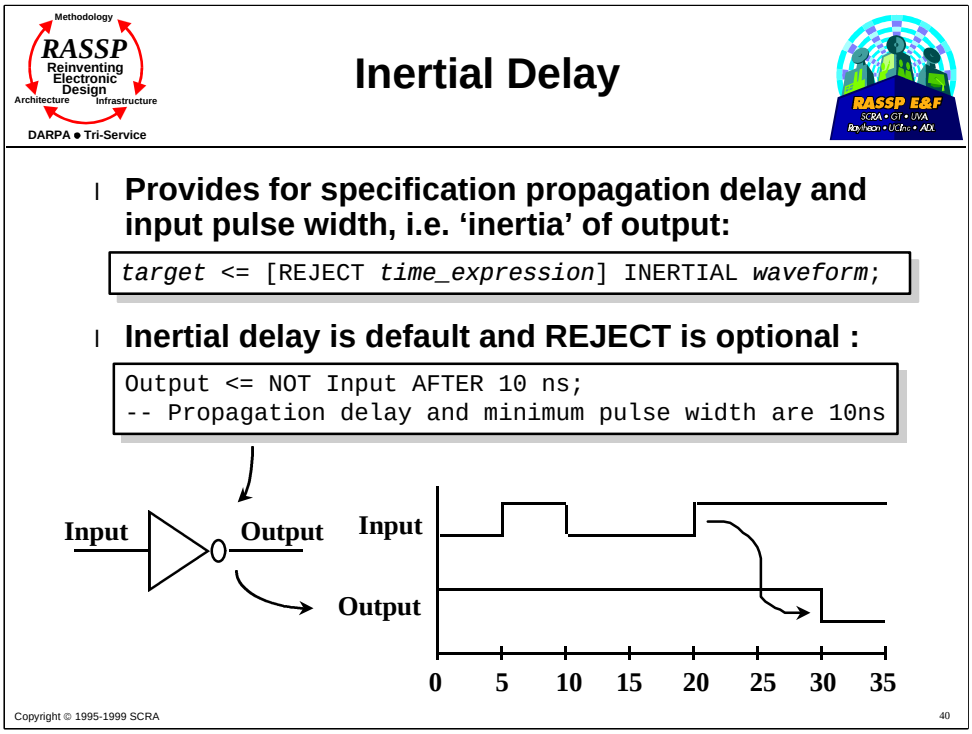
It bears repeating that any signal assignment in VHDL is actually a scheduling for a future value to be placed on that signal. When a signal assignment statement is executed, the signal maintains its original value until the time for the scheduled update to the new value. Any signal assignment statement will incur a delay of one of the three types listed in this slide.



The keyword TRANSPORT must be used to specify a transport delay.

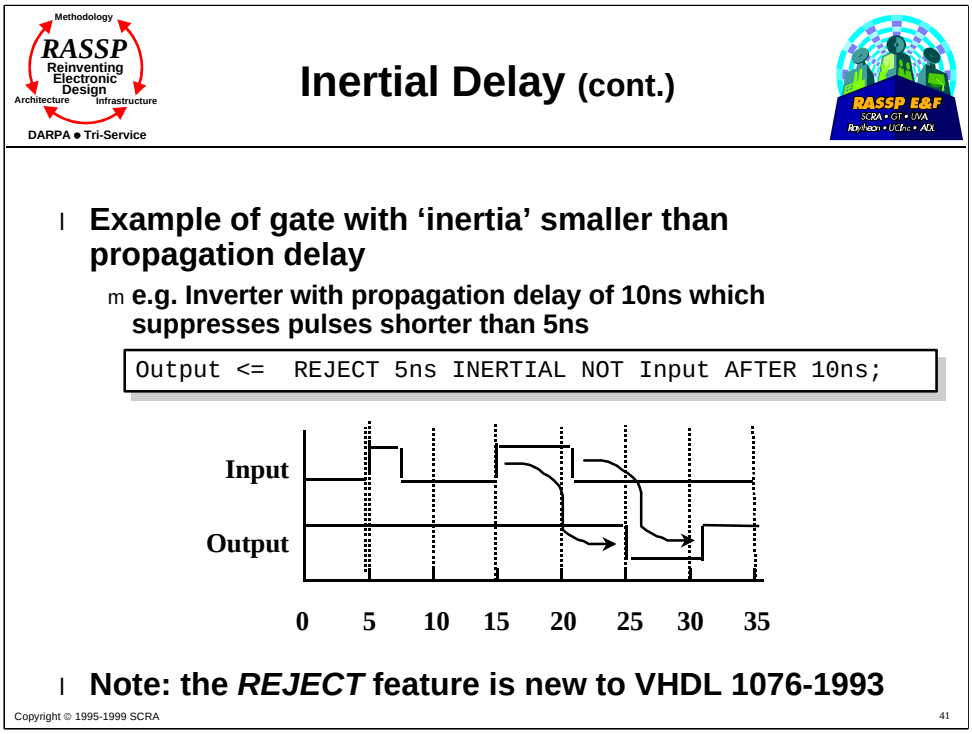
Transport delay is the simplest in that when it is specified, any change in an input signal value may result in a new value being assigned to the output signal after the specified propagation delay.

Note that no restrictions are specified on input pulse widths. In this example, *Output* will be an inverted copy of *Input* delayed by the 10ns propagation delay regardless of the pulse widths seen on *Input* .



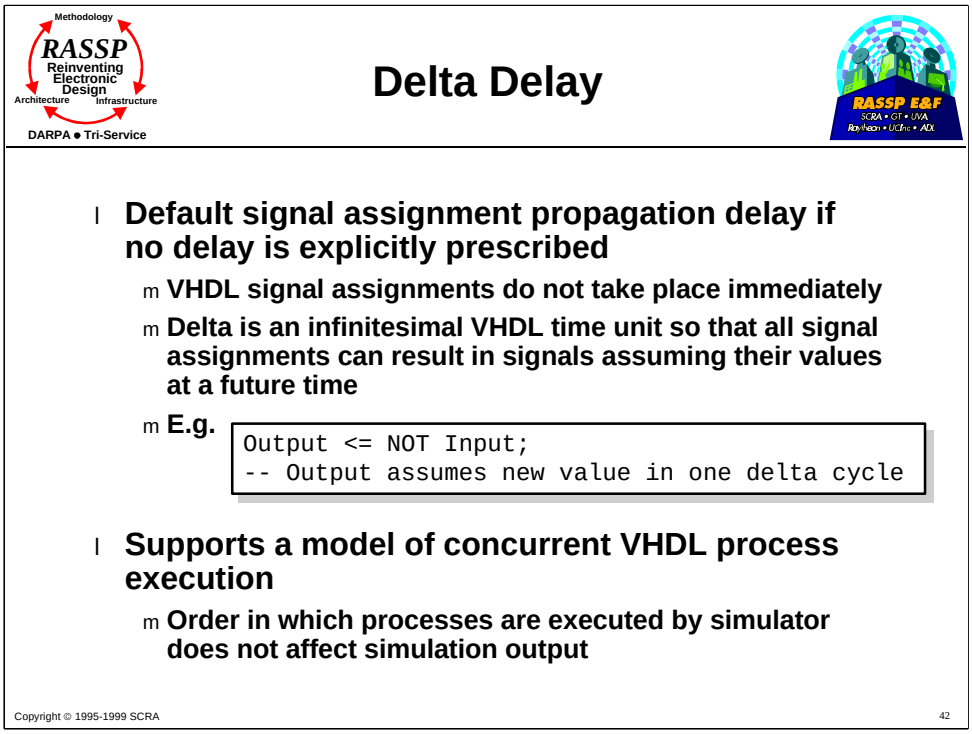
The keyword INERTIAL may be used in the signal assignment statement to specify an inertial delay, or it may be left out because inertial delay is used by default in VHDL signal assignment statements which contain "after" clauses.

If the optional REJECT construct is not used, the specified delay is then used as both the 'inertia' (i.e. minimum input pulse width requirement) and the propagation delay for the signal. Note that in the example above, pulses on *Input* narrower than 10ns are not observed on *Output.*



The REJECT construct is a new feature to VHDL introduced in the VHDL 1076-1993 standard. The REJECT construct can only be used with the keyword INERTIAL to include a time parameter that specifies the input pulse width constraint.

Prior to this, a description for such a gate would have needed the use of an intermediate signal with the appropriate inertial delay followed by an assignment of this intermediate signal's value to the actual output via a transport delay.



VHDL allows the designer to describe systems at various levels of abstraction. As such, timing and delay information may not always be included in a VHDL description.

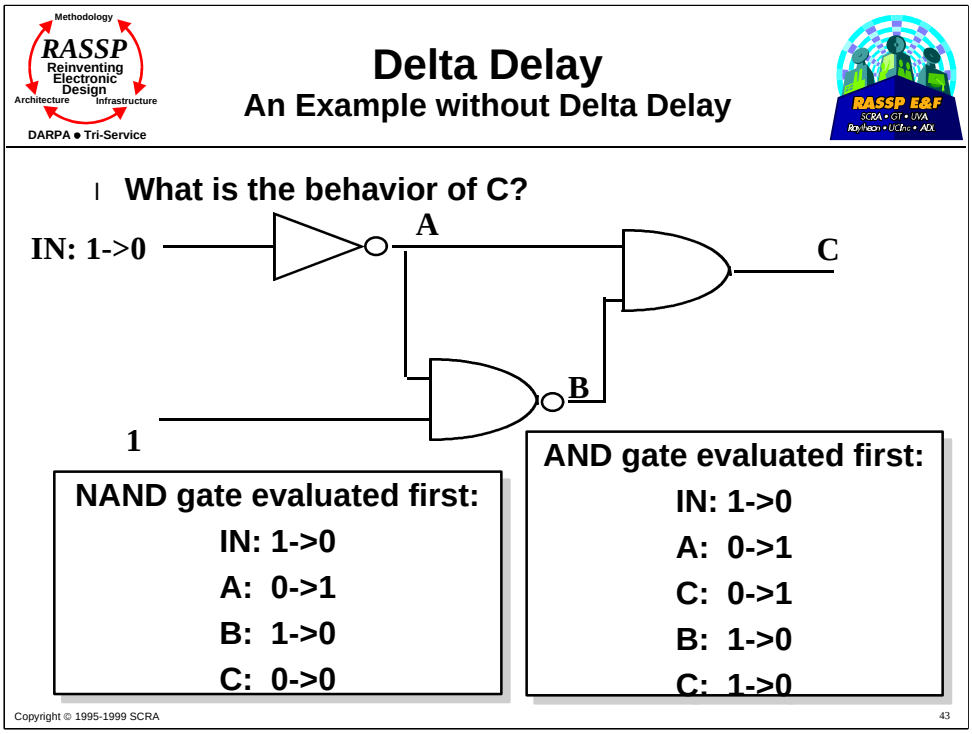
A delta (or delta cycle) is essentially an infinitesimal, but quantized, unit of time. The delta delay mechanism is used to provide a minimum delay in a signal assignment statement so that the simulation cycle described earlier can operate correctly when signal assignment statements do not include explicitly specified delays. That is:

1) all active processes can execute in the same simulation cycle

2) each active process will suspend at wait statement

3) when all processes are suspended simulation is advanced the minimum time necessary so that some signals can take on their new values

4) processes then determine if the new signal values satisfy the conditions to proceed from the wait statement at which they are suspended



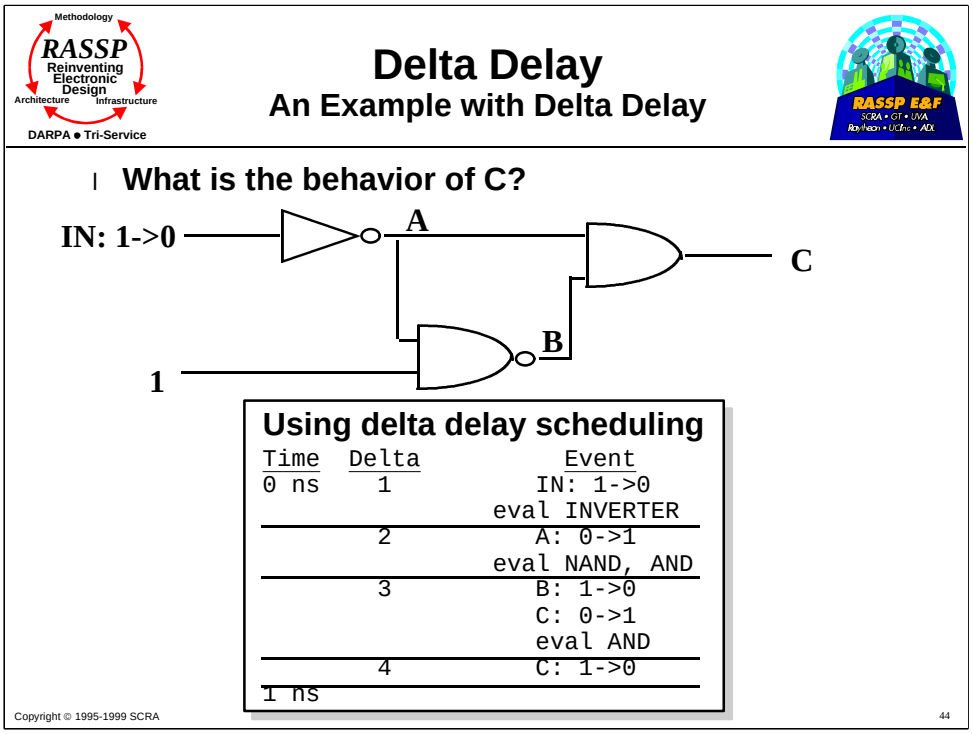
For this discussion, assume that the above circuit does not specify any delays, and that there is no delta delay mechanism. In such a case, the order in which model processes (or components) are executed will affect the model outputs. Consider the example above in which there is a 1 to 0 transition at the input of the AND while the other input to the NAND gate is a constant 1. What is the behavior of *C*?

Note that in the case on the left where the NAND gate is evaluated before the AND gate, *C* can remain at its quiescent value of 0.

However, in the case on the right we see that if the AND gate is evaluated before the NAND gate, a glitch is seen at *C* (i.e. a static-0 hazard is observed). It is generated because the NAND gate has not yet been updated to its new value which will subsequently cause *C* to become 0. Therefore, *C* initially goes to 1 and will only go to 0 after the NAND gate drives its output to 0.

Therefore, if the order of execution is arbitrary, the behavior of the system may be unpredictable.

[Perry94], pp. 22-24.



In this example, each signal assignment requires one delta cycle delay before the signal assumes its new value. Also note that more than one process can be executed in the same simulation cycle (e.g. both the NAND process and the AND process are executed during delta 2).

Following the sequence of events defined by the VHDL simulation cycle, the 1-0 transition on IN allows the INVERTER process to be executed which results in a 0-1 transition being scheduled on *A* one delta cycle in the future. The INVERTER process then suspends. Since all process are suspended, simulation time advances by one delta cycle so that *A* can assume its new value.

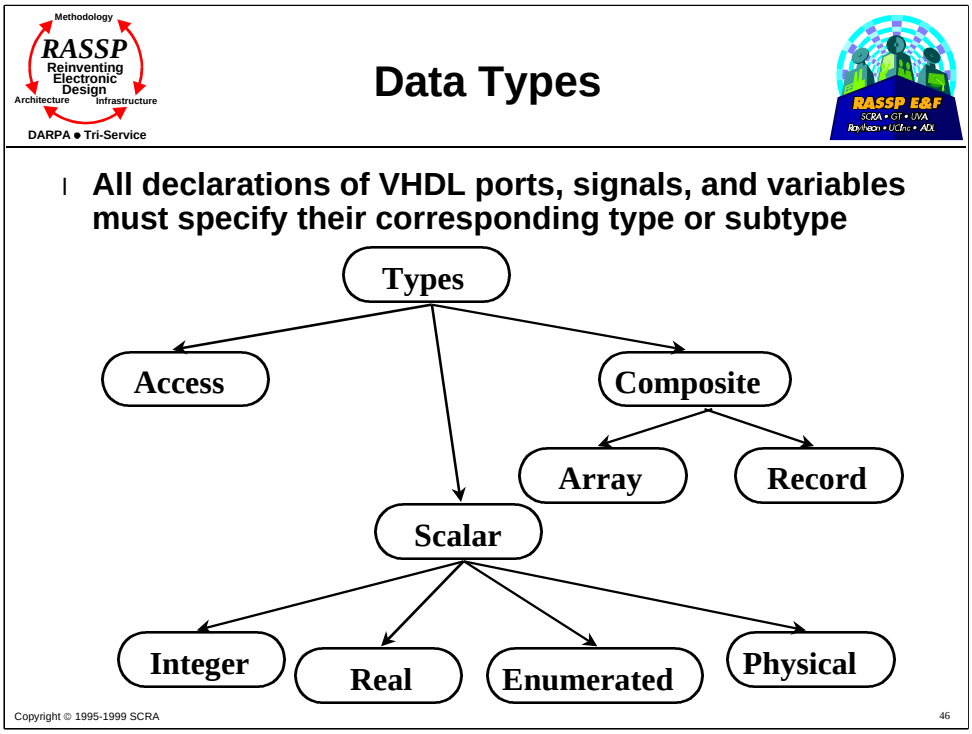
The new value of *A* allows the NAND and AND processes to be executed. Because the value of *A* will not change again during simulation time delta 2, it doesn't matter whether NAND or AND is evaluated first. In either case, the NAND process leads to a 1 being scheduled for *C* and a 0 being scheduled for *B*, both one delta cycle in the future. After the assignments are scheduled, NAND and AND suspend again. Again, simulation time advances by one delta cycle so that *B* and *C* can assume their new values.

The new value of *B* causes the AND process to be evaluated again. This time, a 0 value is scheduled to be assigned to *C* one delta cycle in the future, and the AND process can then suspend. Finally, simulation time advances by one delta cycle so that *C* can assume its new, and final value.

Based on [Perry94], pp 22-24

Module Outline

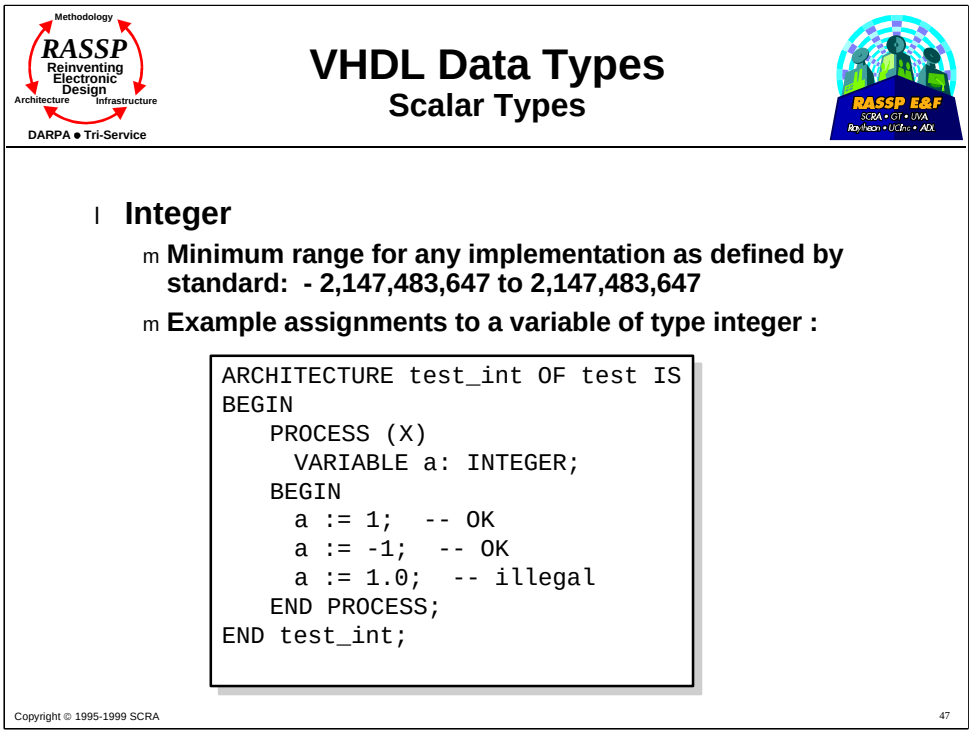
- Introduction
- VHDL Design Example
- VHDL Model Components
- **Basic VHDL Constructs**
  - Data types
  - Objects
  - Sequential and concurrent statements
  - Packages and libraries
  - Attributes
  - Predefined operators
- Examples
- Summary



The three defined data types in VHDL are access, scalar, and composite. Note that VHDL 1076-1987 defined a fourth data type, file, but files were reclassified as objects in VHDL 1076-1993. In any case, files will not be discussed in this module but will be covered in RASSP E&F Module 13, 'Advanced Concepts in VHDL', included in this collection of educational modules.

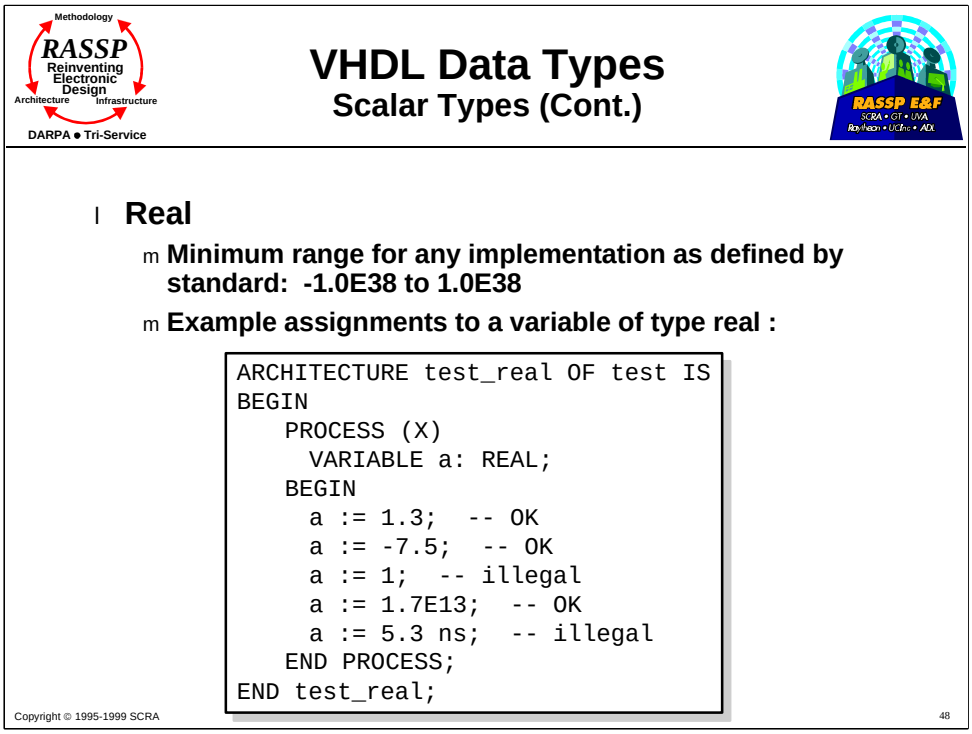
Simply put, access types are akin to pointers in other programming languages, scalar types are atomic units of information, and composite types are arrays and/or records. These are explained in more detail in the next few slides. In addition, subtypes will also be introduced.

[Perry94], pp 74

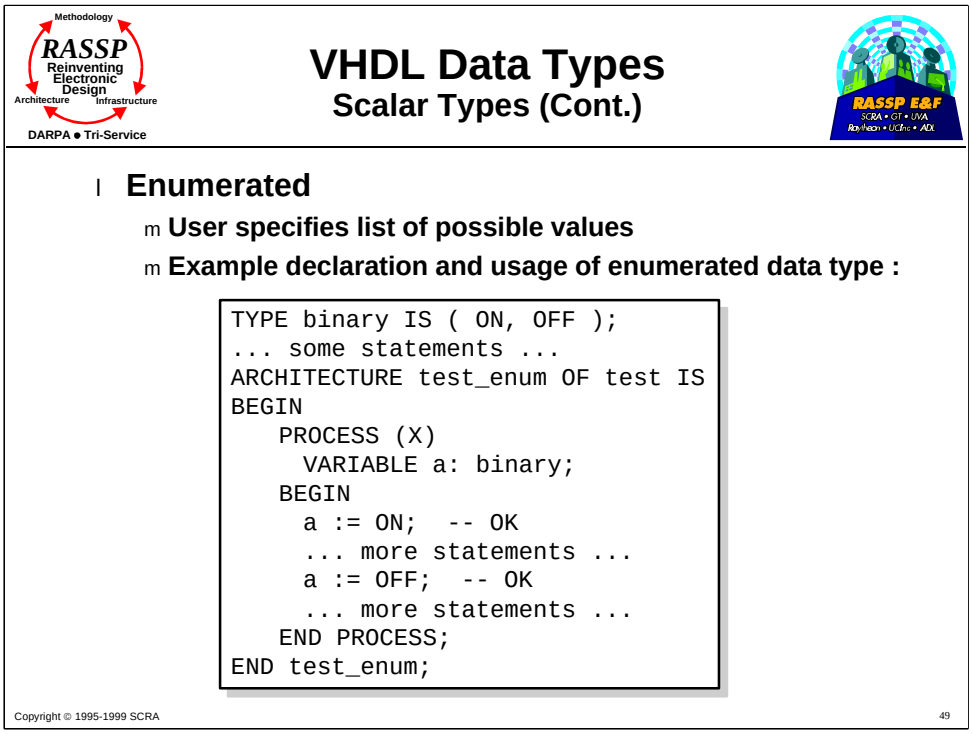


Scalar objects can hold only one data value at a time. A simple example is the integer data type. Variables and signals of type integer can only be assigned integers within a simulator-specific range, although the VHDL standard imposes a minimum range.

In the above example, the first two variable assignments are valid since they assign integers to variables of type integer. The last variable assignment is illegal because it attempts to assign a real number value to a variable of type integer.



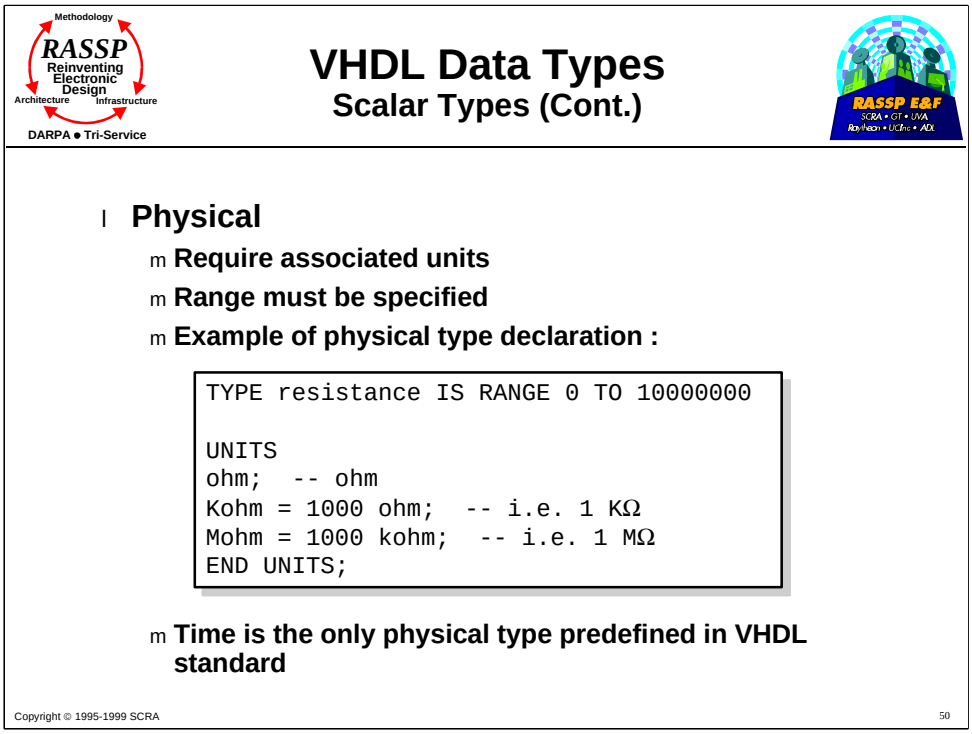
A second simple example is the real data type. This type consists of the real numbers within a simulator-specific (but with a VHDL standard imposed minimum) range. The variable assignment lines marked OK are valid assignments. The first illegal statement above attempts to assign an integer to a real type variable, and the second illegal statement is not allowed since the unit “ns” denotes a physical data type.



The enumerated data type allows a user to specify the list of legal values that a variable or signal of the defined type may be assigned. As an example, this data type is useful for defining the various states of a FSM with descriptive names.

The designer first declares the members of the enumerated type. In the example above, the designer declares a new type binary with two legal values, ON and OFF.

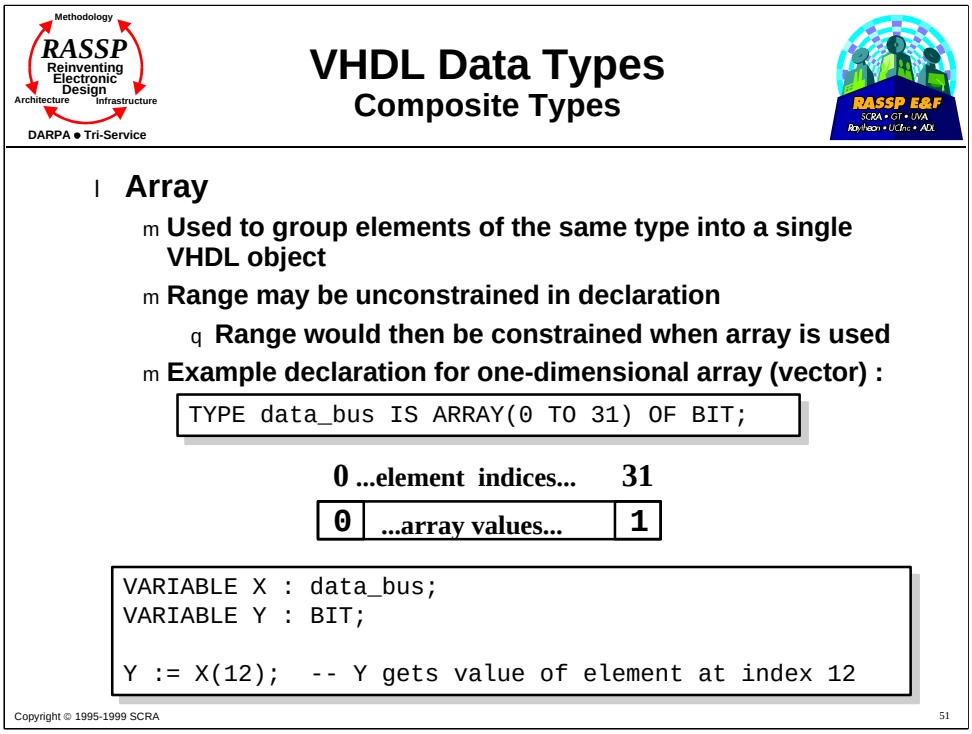
Note that VHDL is not case sensitive. Typing reserved words in capitals and variables in lower case may enhance readability, however.



The physical data type is used for values which have associated units. The designer first declares the name and range of the data type and then specifies the units of the type. Notice there is no semicolon separating the end of the TYPE statement and the UNITS statement. The line after the UNITS line states the base unit of of the type. The units after the base unit statement may be in terms of the base unit or another already defined unit.

Note that VHDL is not case sensitive so Kohm and kohm refer to the same unit.

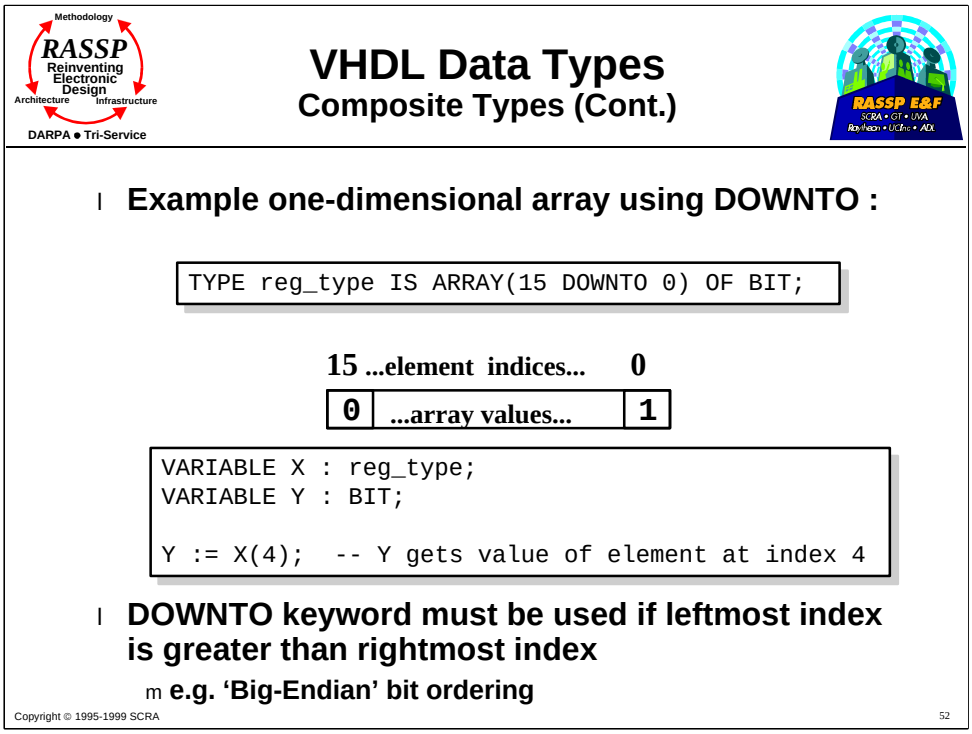
The only predefined physical type in VHDL is time.



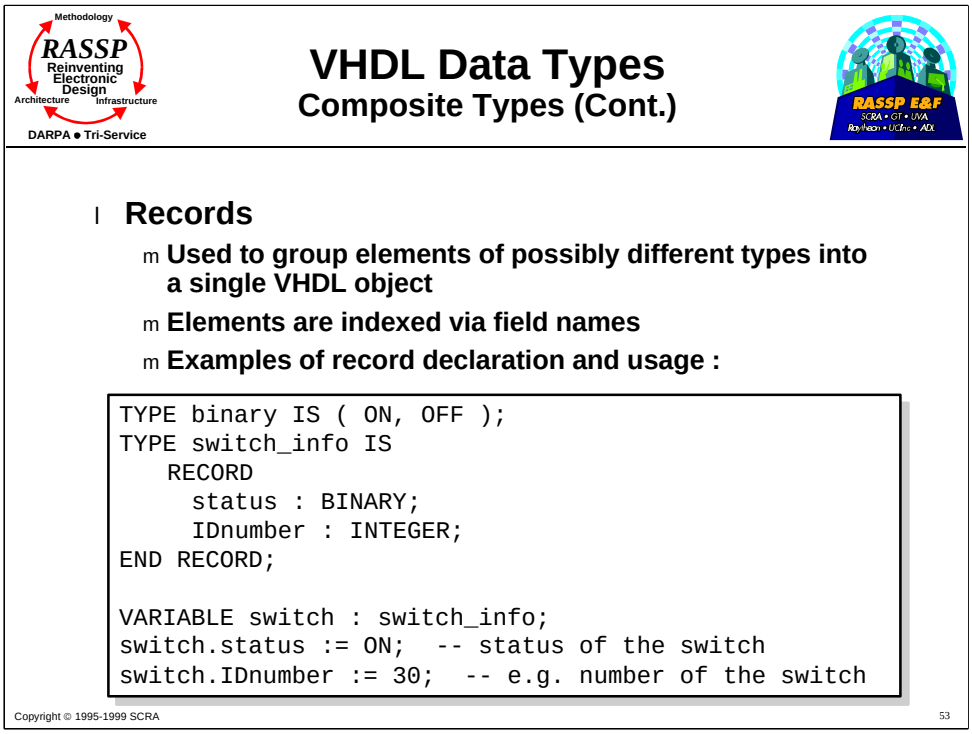
VHDL composite types consists of arrays and records. Each object of this data type can hold more than one value.

Arrays consist of many similar elements of any data type, including arrays. The array is declared in a TYPE statement. There are numerous items in an array declaration. The first item is the name of the array. Second, the range of the array is declared. The keywords TO and DOWNTO designate ascending or descending indices, respectively, within the specified range. The third item in the array declaration is the specification of the data type for each element of the array.

In the example above, an array consisting of 32 bits is specified. Note that individual elements of the array are accessed by using the index number of the element as shown above. The index number corresponds to where in the specified range the index appears. For example, X(12) above refers to the thirteenth element from the left (since the leftmost index is 0) in the array.



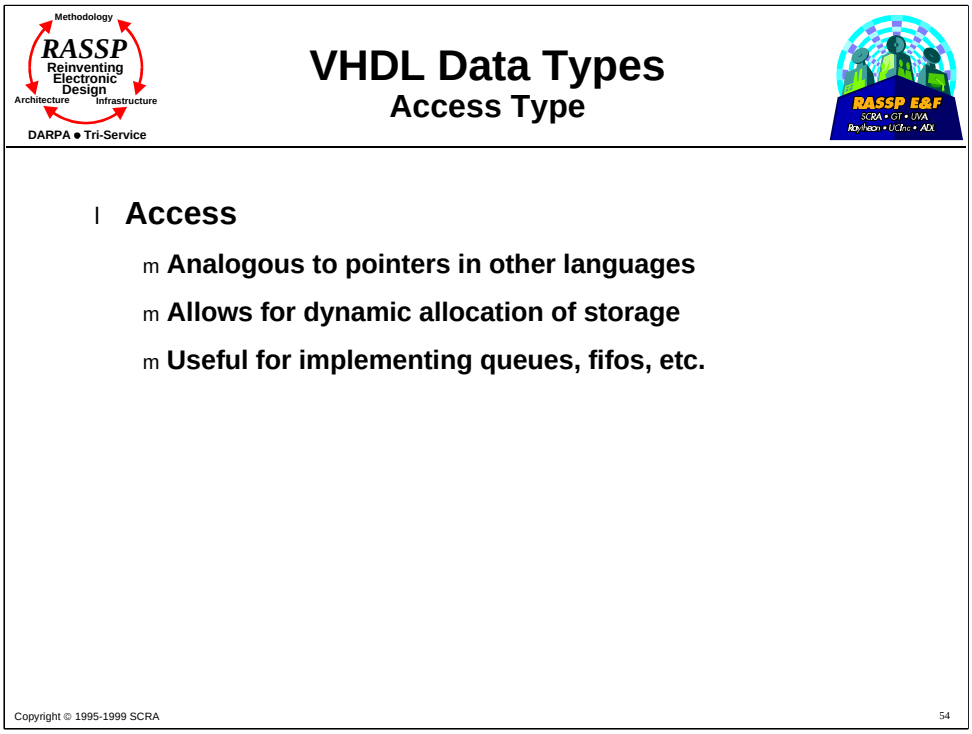
This example illustrates the use of the DOWNTO designator in the range specification of the array. DOWNTO specifies a descending order in array indices so that in the example above, X(4) refers to the fifth element from the right in the array (with 0 being the index for the element furthest to the right in this case).



The second VHDL composite type is the record. An object of type record may contain elements of different types. Again, a record element may be of any data type, including another record.

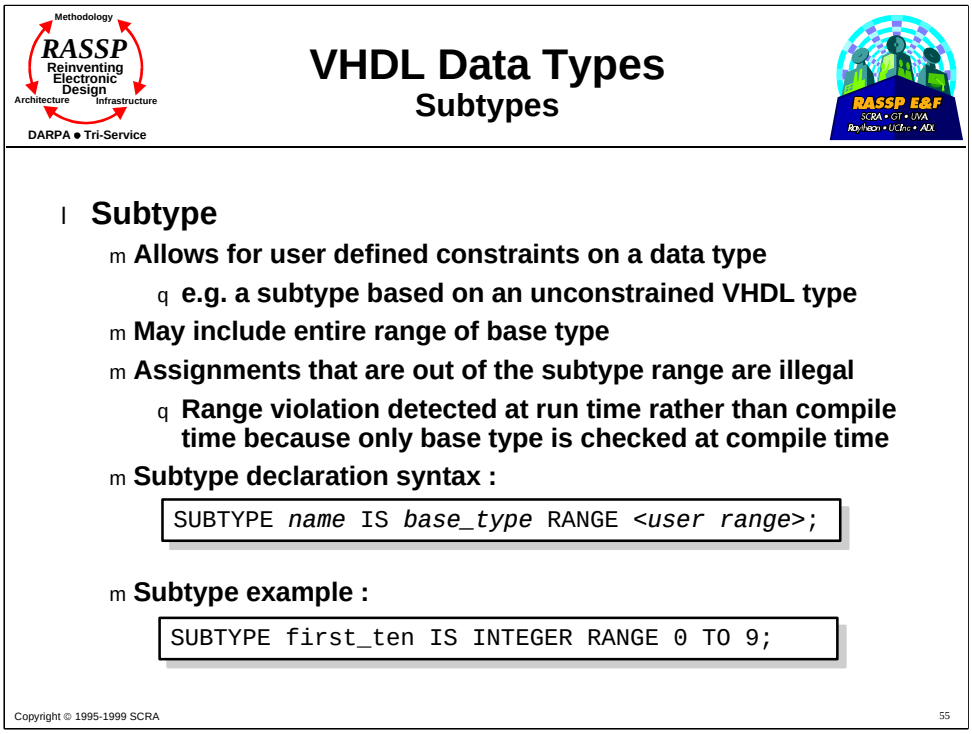
A TYPE declaration is used to define a record. Note that the types of a record's elements must be defined before the record is defined. Also notice that there is no semi-colon after the word RECORD. The RECORD and END RECORD keywords bracket the field names. After the RECORD keyword, the record's field names are assigned and their data types are specified.

In the above example, a record type, switch_info, is declared. This example makes use of the binary enumerated type declared previously. Note that values are assigned to record elements by use of the field names.



The VHDL access type will not be discussed in detail in this module; it will be covered more thoroughly in the 'Advanced Concepts in VHDL' module appearing in this collection of modules.

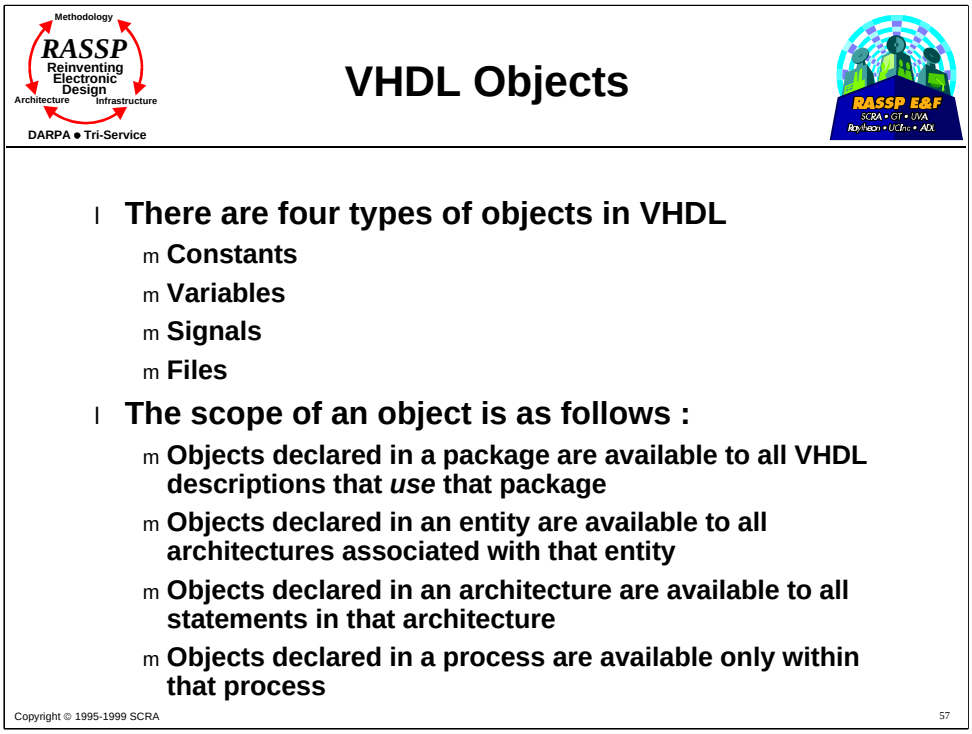
In brief, the access type is similar to a pointer in other programming languages in that it dynamically allocates and deallocates storage space to the object. This capability is useful for implementing abstract data structures (such as queues and first-in-first-out buffers) where the size of the structure may not be known at compile time.



VHDL subtypes are used to constrain defined types. Constraints take the form of range constraints or index constraints. However, a subtype may include the entire range of the base type. Assignments made to objects that are out of the subtype range generate an error at run time. The syntax and an example of a subtype declaration are shown above.

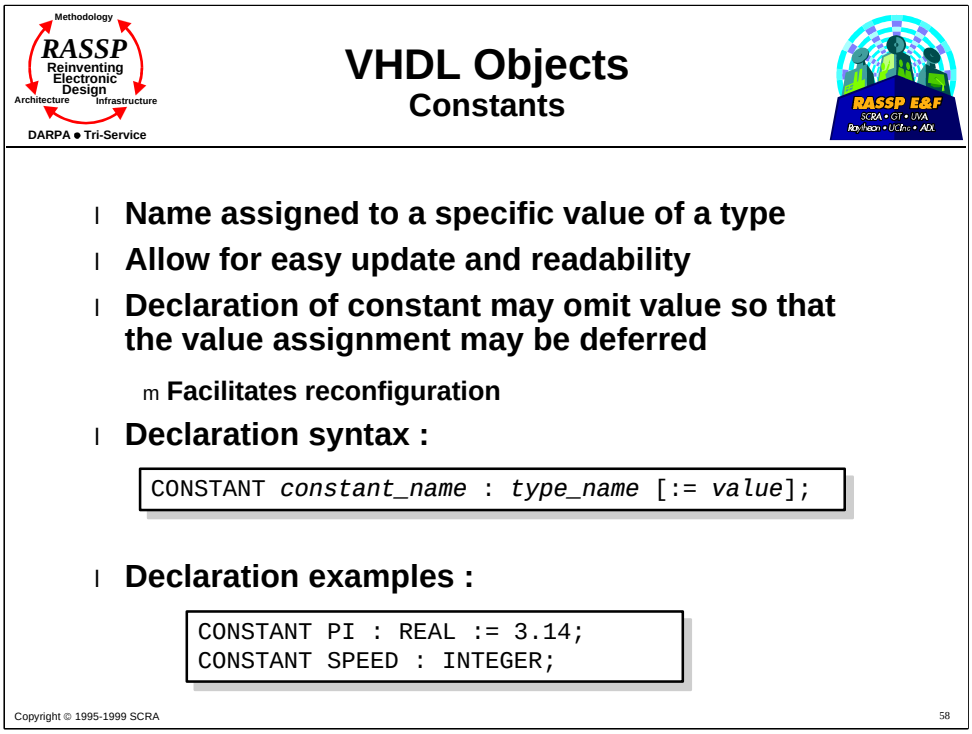
VHDL Data Types
Summary

- **All declarations of VHDL ports, signals, and variables must include their associated type or subtype**
- **Three forms of VHDL data types are :**
  - **Access -- pointers for dynamic storage allocation**
  - **Scalar -- includes Integer, Real, Enumerated, and Physical**
  - **Composite -- includes Array, and Record**
- **A set of built-in data types are defined in VHDL standard**
  - **User can also define own data types and subtypes**



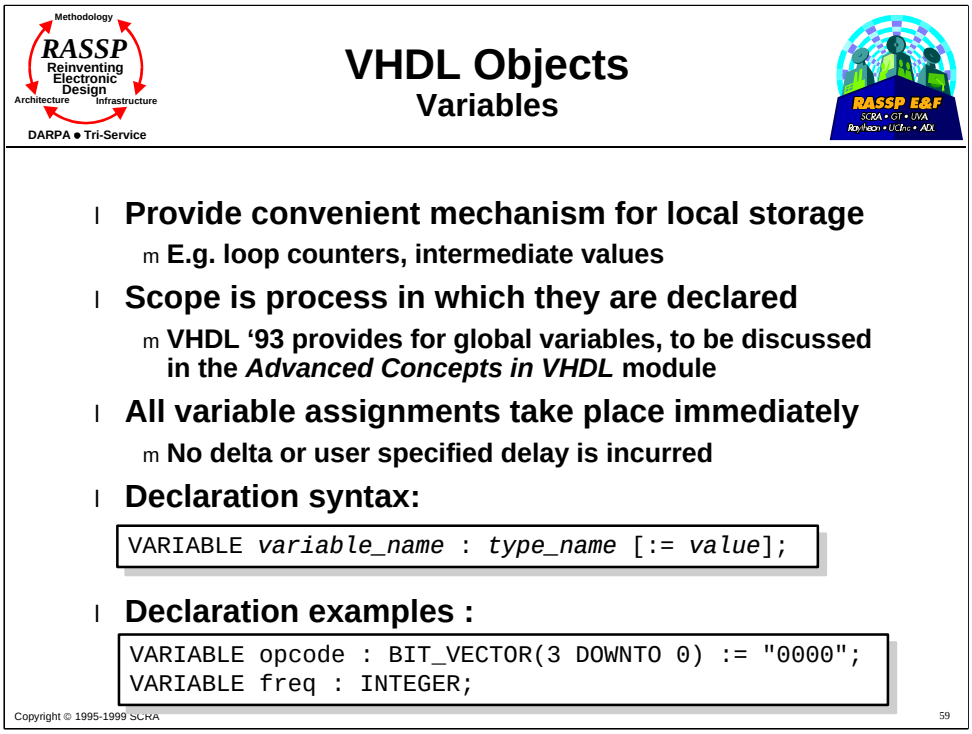
VHDL 1076-1993 defines four types of objects, files, constants, variables, and signals. Simple scoping rules determine where object declarations can be used. This allows the reuse of identifiers in separate entities within the same model without risk of inadvertent errors.

For example, a signal named data could be declared within the architecture body of one component and used to interconnect its underlying subcomponents. The identifier data may also be used again in a different architecture body contained within the same model.



VHDL constants are objects whose values do not change. The value of a constant, however, does not need to be assigned at the time the constant is declared; it can be assigned later in a package body if necessary, for example.

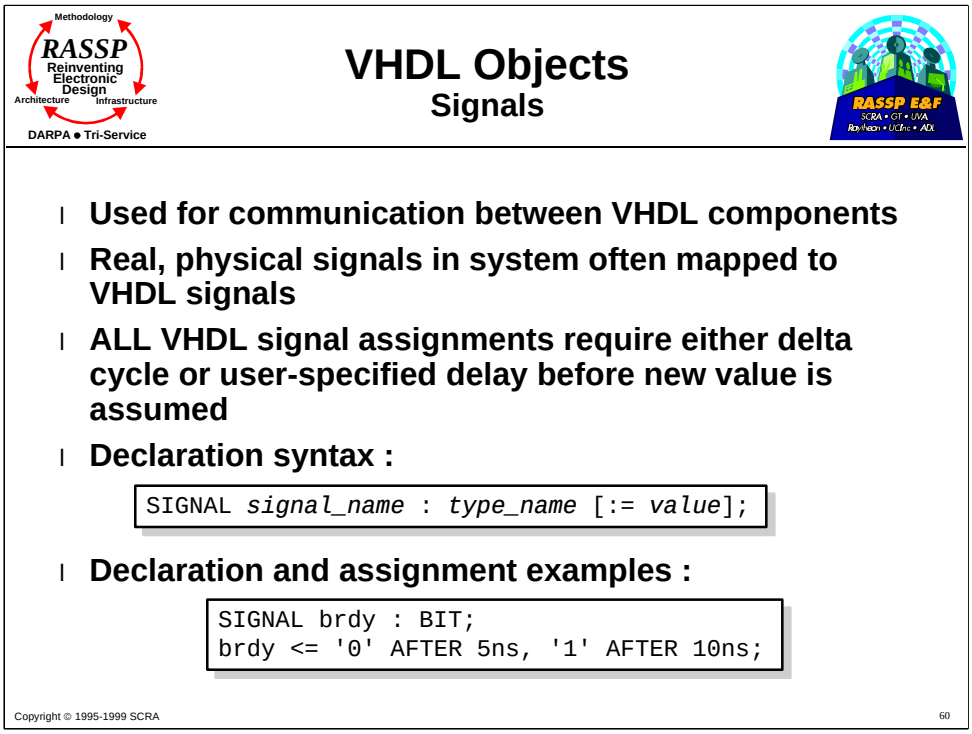
The syntax of the constant declaration statement is shown above. The constant declaration includes the name of the constant, its type, and, optionally, its value.



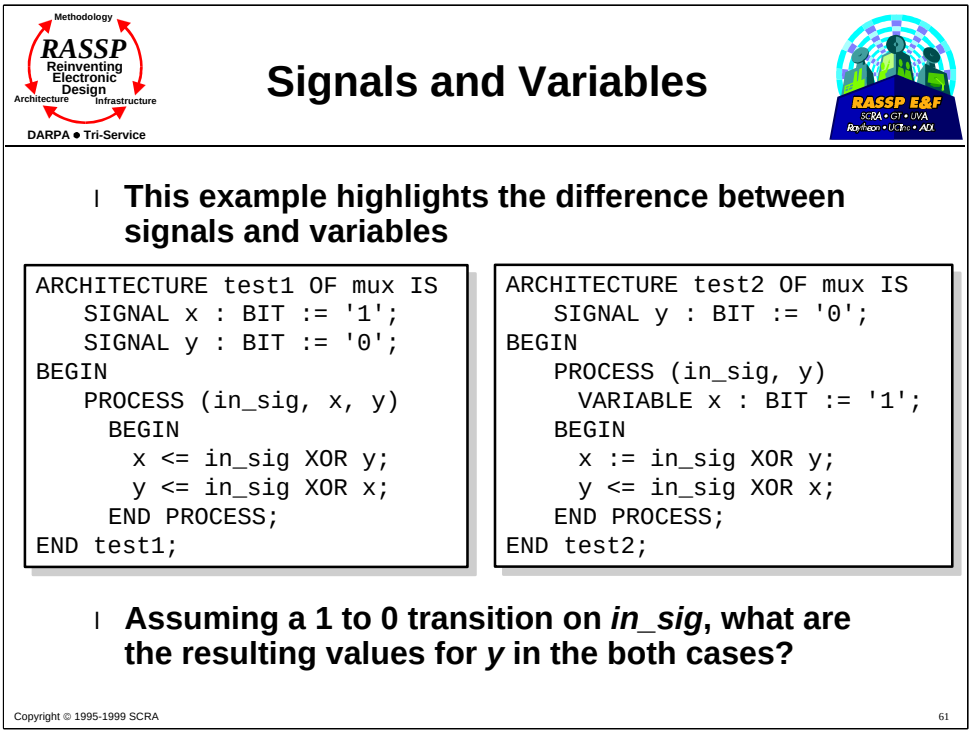
This discussion about VHDL variables does not include global (aka shared) variables which were introduced in the 1076-1993 standard. The discussion of shared variables is deferred until Module 13, "Advanced Concepts in VHDL."

An important feature of the behavior of VHDL variables is that an assignment to a VHDL variable results in the variable assuming its new value immediately (i.e. no simulation time or delta cycles must transpire as is the case for VHDL signals). This feature allows the sequential execution of statements within VHDL processes where variables are used as placeholders for temporary data, loop counters, etc.

Examples of variable declarations and assignments are shown above. Note that when a variable is declared, it may optionally be given an initial value as well.



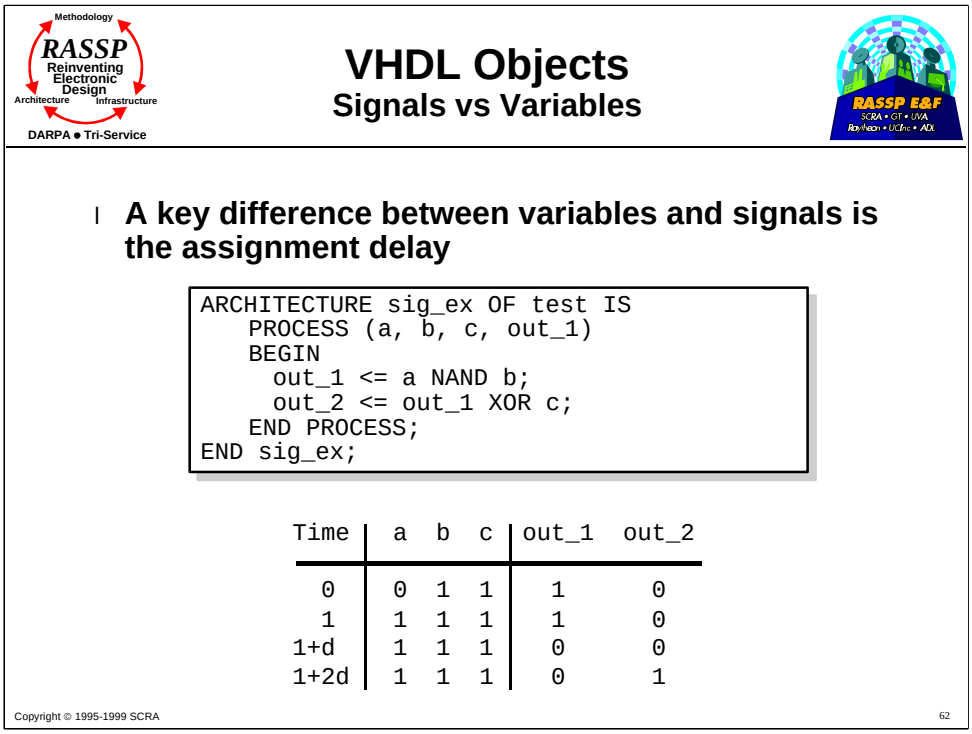
Signals are used to pass information directly between VHDL processes and entities. As has already been said, signal assignments require a delay before the signal assumes its new value. In fact, a particular signal may have a series of future values with their respective timestamps pending in the signal's *waveform*. The need to maintain a waveform results in a VHDL signal requiring more simulator resources than a VHDL variable.



Note that signal assignments require that a delay be incurred before the signals assume their new values.

In the example on the left, the signal assignment for *x* leads to a '0' being scheduled on *x* one delta cycle in the future. Note that *x* still holds its original value of '1', however, when the signal assignment for *y* is evaluated. Thus, the signal assignment statement for *y* evaluates to '1', and *y* will assume this new value one delta cycle in the future. This contrived example actually leads to *x* and *y* swapping values in delta time while *in_sig* has a value of '0'.

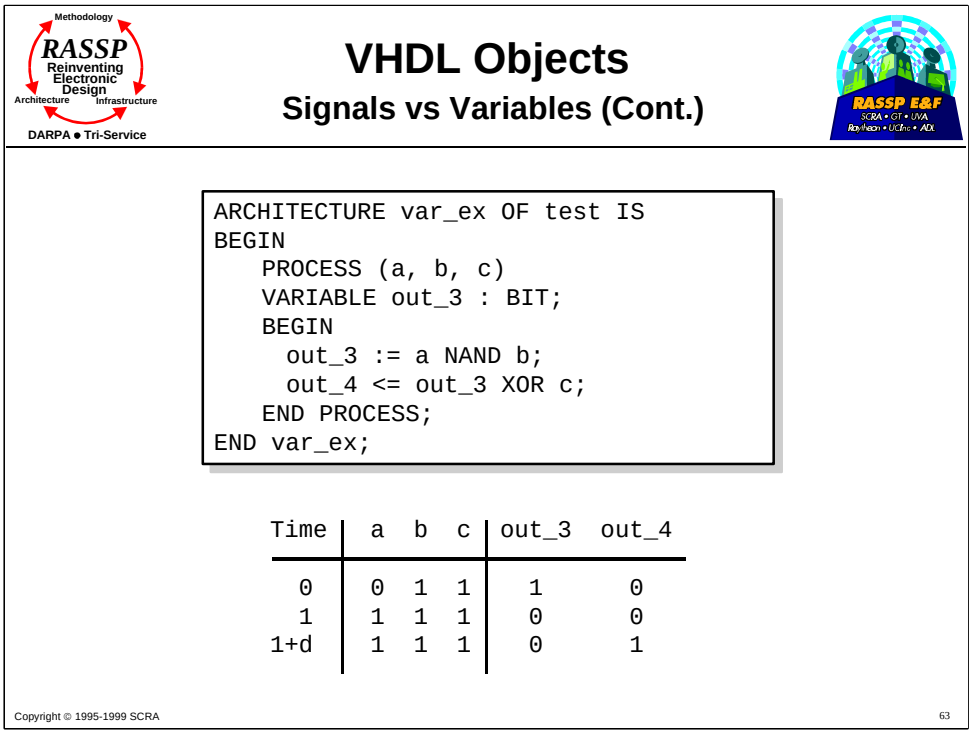
In the example on the right, the variable assignment for *x* leads to *x* assuming a '0' immediately. Thus, when the signal assignment for *y* is evaluated, *x* already has its new value and the statement evaluates to a '0', resulting in *y* retaining its original value. This example does not perform the swapping in delta time that would be performed by the example on the left.



To review, note that some delay must transpire after a VHDL signal assignment statement before the signal assumes its new value. Examples will be used in this and the next slide to illustrate the difference between signals and variables. The example shown above utilizes signals. Note that in this example, a, b, c, out_1, and out_2 are signals that are declared elsewhere, e.g. in the component's *entity.*

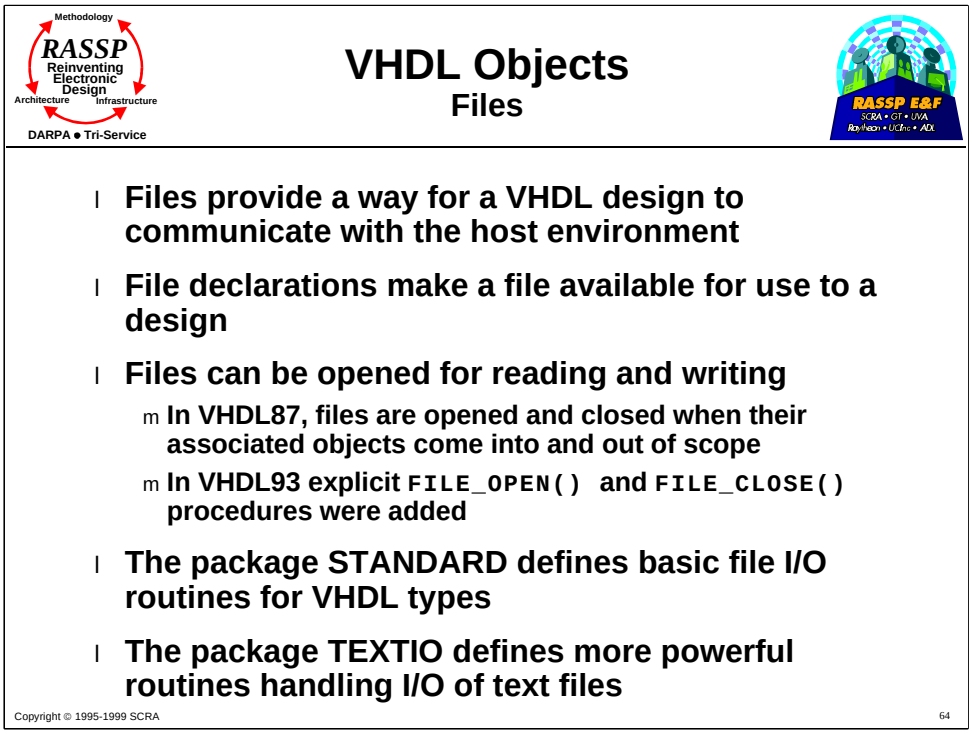
The table indicates the values for the various signals at the key times in the example. At time 1, a new value of 1 is observed on *a*. This causes the process sensitivity list to fire and results in a 0 being assigned to *out_1.* The signal assignment statement for *out_2* will also be executed but will not result in a new assignment to *out_2* because neither *out_1* nor *c* will be changed at this time. At time 1+d (i.e. 1 plus 1 delta cycle), *out_1* assumes its new value causing the process sensitivity list to fire again. In this process execution, the statement for *out_1* will be executed again but no new assignment will be made because its right hand side parameters have not changed. The *out_2* assignment statement, however, results in a 1 being assigned to *out_2*. At time 1+2d, *out_2* assumes its new value of 1. This example, then, requires 2 delta cycles and two process executions to arrive at its quiescent state following a change to *a* (or *b*, for that matter).

[MG90]

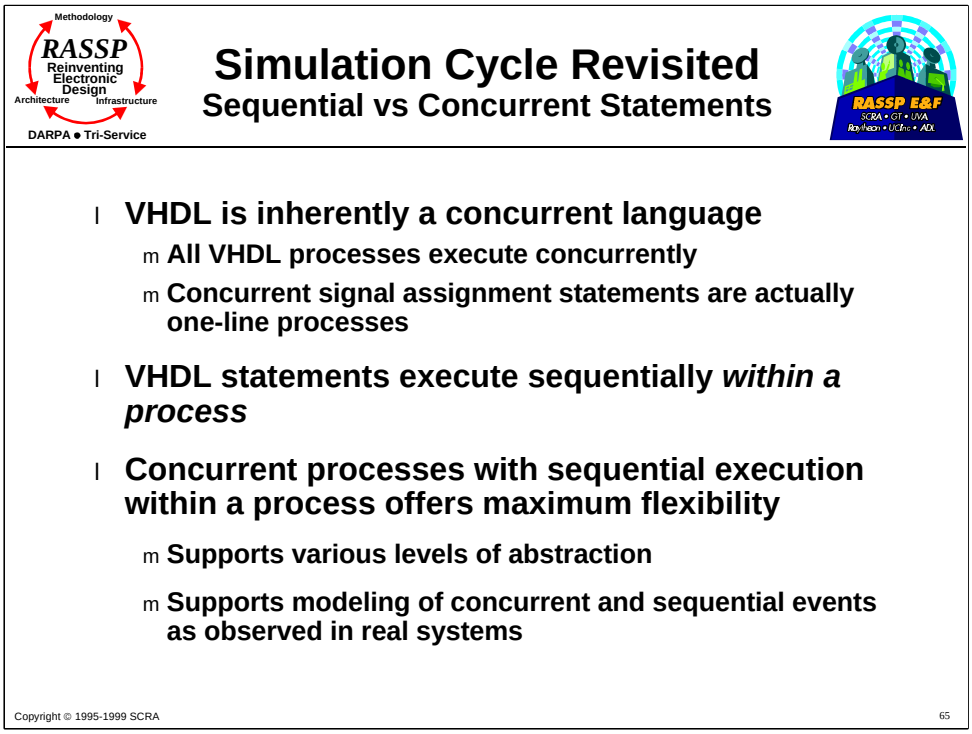


In this example, variables are used to achieve the same functionality as the example in the previous slide. In this example, however, when there is a change in *a* at time 1, *out_3* will assume its new value at time 1 because it is a variable, and VHDL variable assignment statements result in the new values being assumed immediately. The new value for *out_4*, therefore, will be calculated with the new *out_3* value and results in an assignment to a value of '1' being scheduled for one delta cycle in the future.

Also note, however, that in this example, the order in which the statements appear within the process is important because the two statements are executed sequentially, and the process will only be executed once as a result of the single change in *a*.



The VHDL file object is introduced above. Files may be opened in read or write mode, and once a file is opened, its contents may only be accessed sequentially. A detailed description of the use of file objects is beyond this module and will be discussed further in the 'Advanced Concepts in VHDL' module.

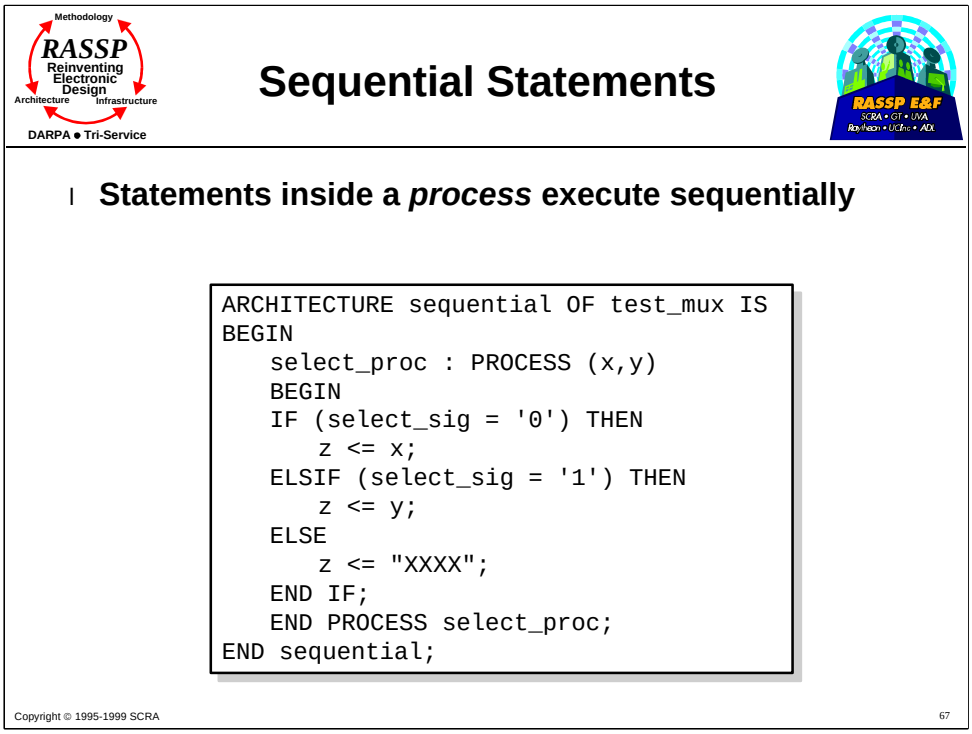


In essence, VHDL is a concurrent language in that all processes execute concurrently. All VHDL execution can be seen as taking place inside processes; concurrent signal assignment statements have already been described as being equivalent to one-line processes. Within a process, however, VHDL adheres to a sequential mode of execution where statements within a process are executed in "top-to-bottom" fashion until the process suspends at a *wait* statement.

This simultaneous support of concurrent and sequential modes allows great flexibility in modeling systems at multiple levels of design and description abstraction.

Concurrent Statements

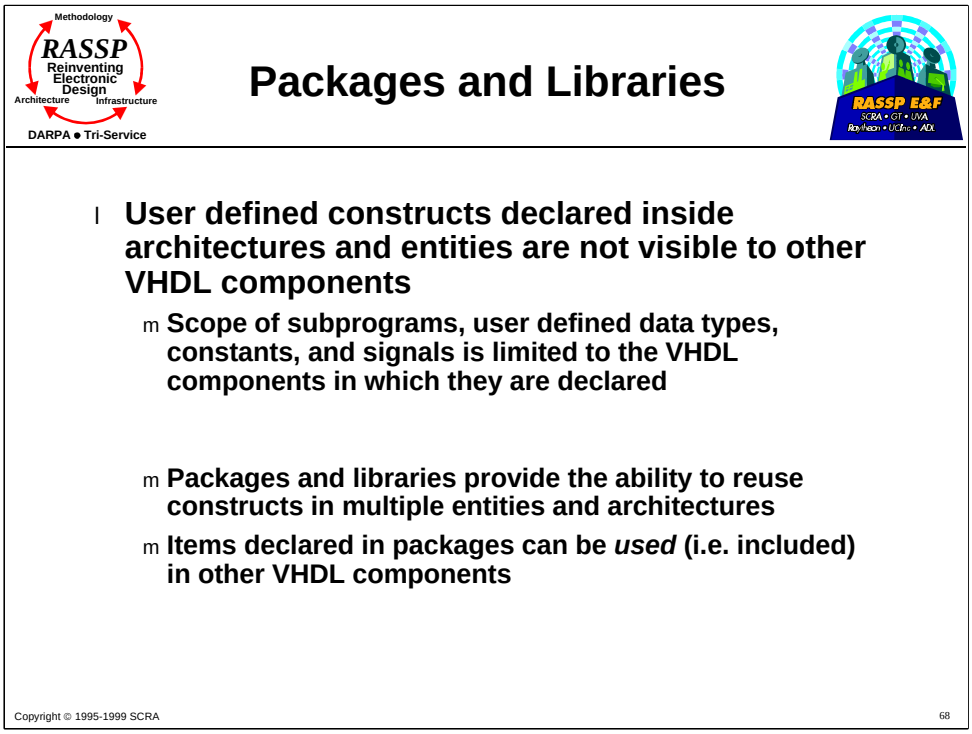
- **Basic granularity of concurrency is the *process***
- **Mechanism for achieving concurrency :**
  - **Processes communicate with each other via signals**
  - **Signal assignments require delay before new value is assumed**
  - **Simulation time advances when all active processes complete**
  - **Effect is concurrent processing**
    - **I.e. order in which processes are actually executed by simulator does not affect behavior**
- **Concurrent VHDL statements include :**
  - **Block, process, assert, signal assignment, procedure call, component instantiation**



Statements in a VHDL process are executed sequentially. A process may also include a sensitivity list which is declared immediately after the PROCESS keyword. The process executes when there is a transition on any of the specified signals. Alternatively, a process would include at least one *wait* statement to control when and where a process may suspend so that signals with pending signal assignments may assume their new values. Actually, a sensitivity list is equivalent to a *wait* statement at the bottom of a process which suspends execution until there is a transition on one of the signals on the sensitivity list.

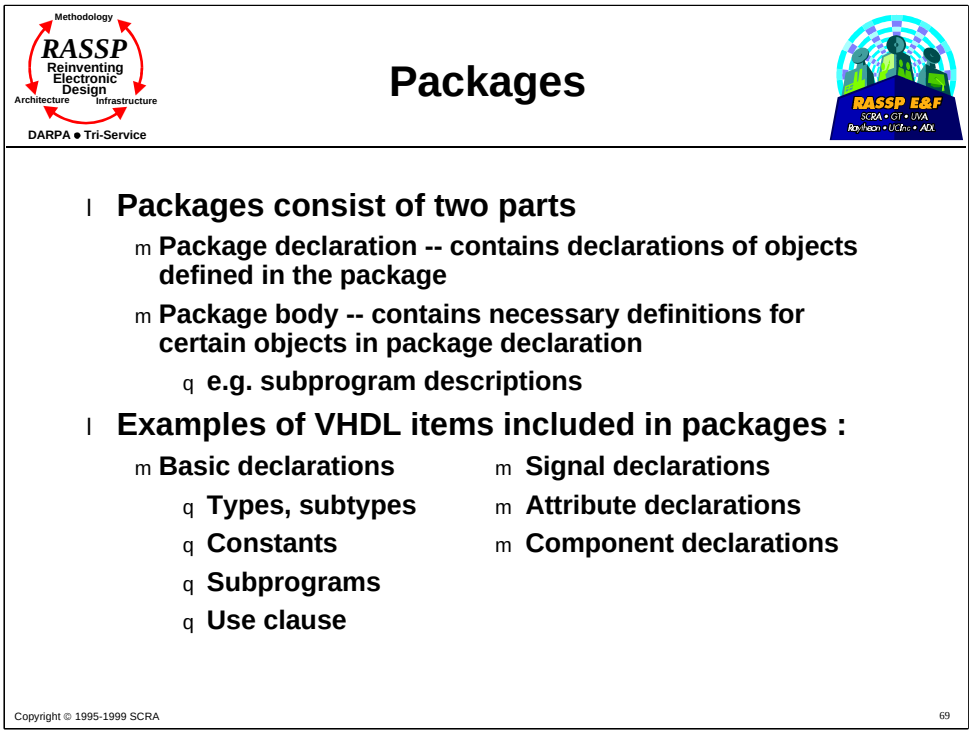
The *wait* statement will be covered in detail in the Behavioral VHDL module.

In the example above, the sensitivity list includes signals *x* and *y*. The process can also be named; the process in the example above is named *select_proc*.



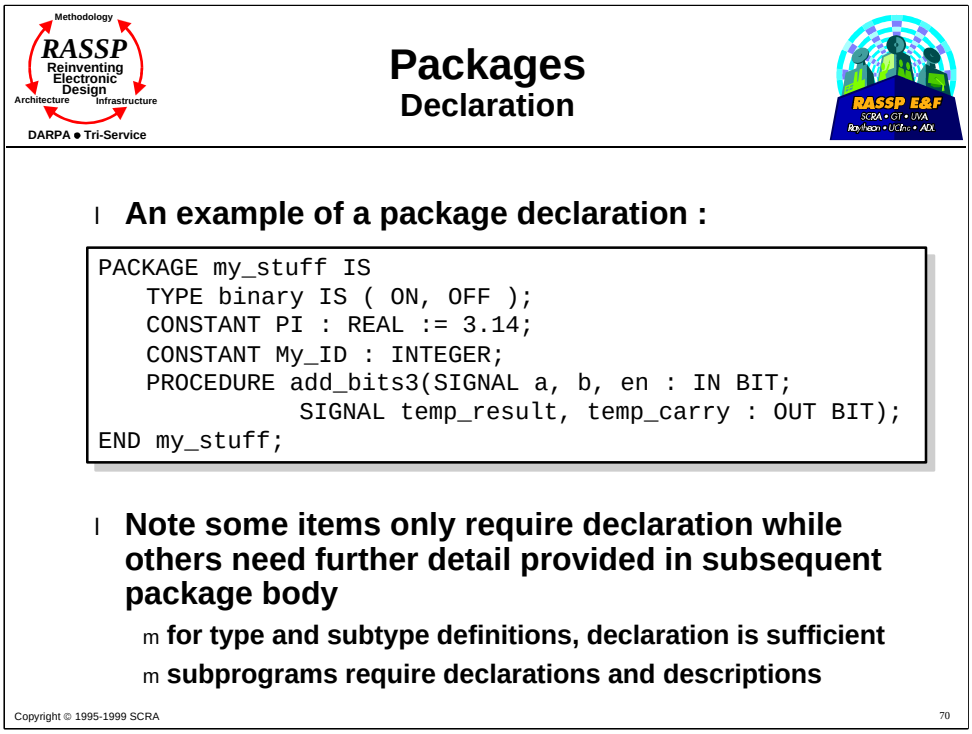
VHDL provides the package mechanism so that user-defined types, subprograms, constants, aliases, etc. can be defined once and reused in the description of multiple VHDL components.

VHDL libraries are collections of packages, entities, and architectures. The use of libraries allows the organization of the design task into any logical partition the user chooses (e.g. component libraries, package libraries to house reusable functions and type declarations).

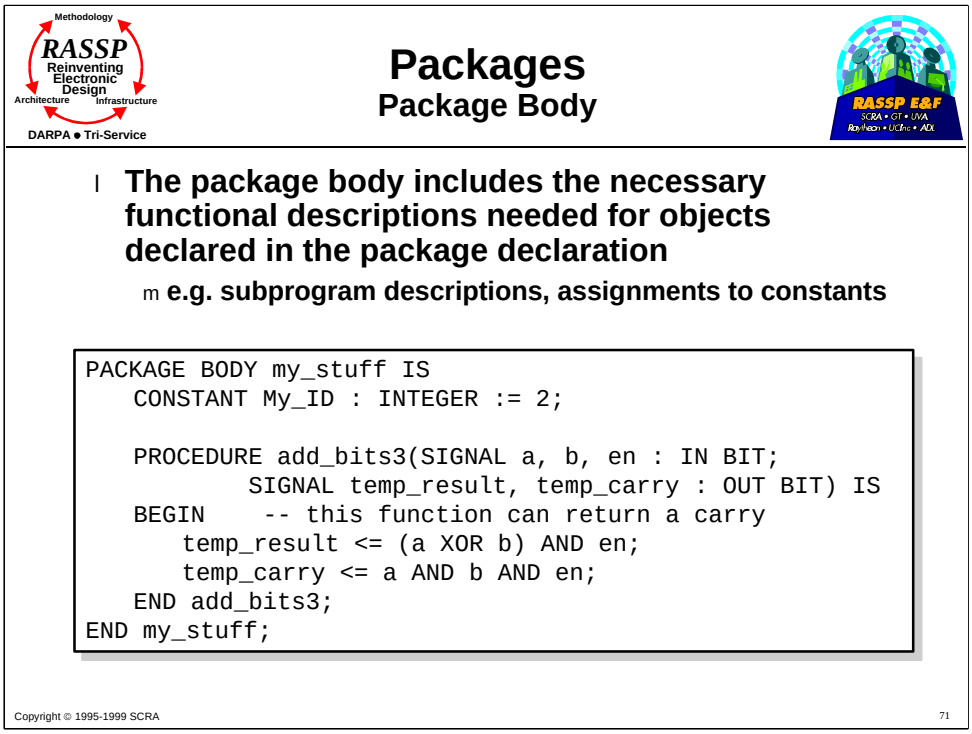


A package contains a collection of user-defined declarations and descriptions that a designer makes available to other VHDL entities. Items within a package are made available to other VHDL entities (including other packages) with a *use* clause. Some examples of possible package contents are shown above.

The next two slides will describe the two parts of a VHDL package, the package declaration and the package body.

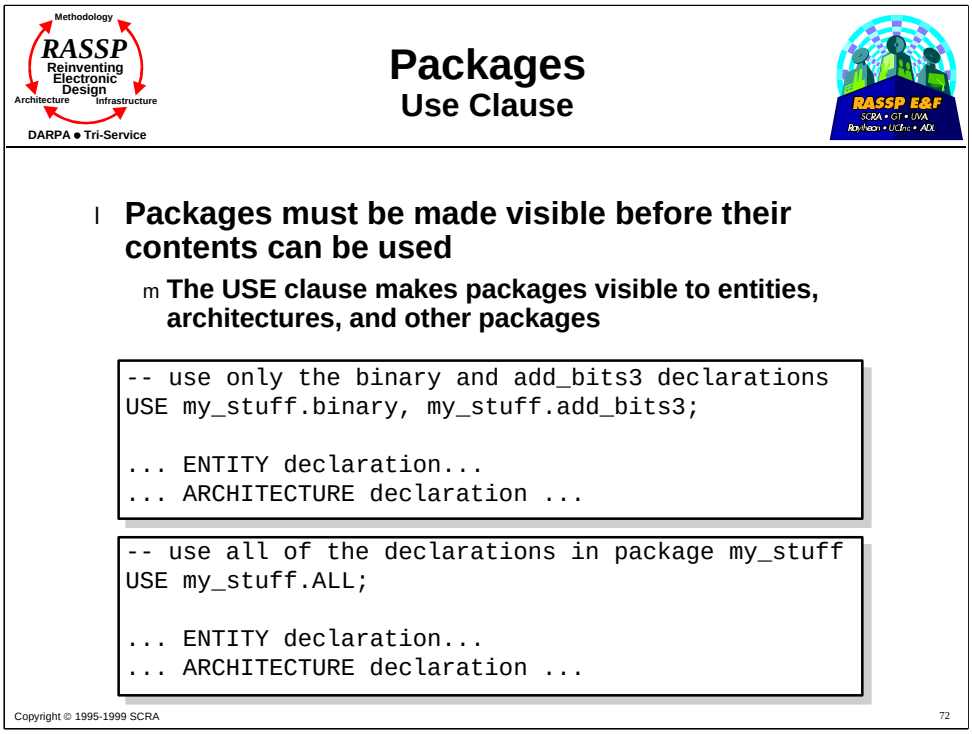


This is an example of a package declaration. The package declaration lists the contents of the package. The declaration begins with the keyword PACKAGE and the name of the package followed by the keyword IS. VHDL declaration statements are then included, such as type declarations, constant declarations, and subprogram declarations. For many VHDL constructs, such as types, declarations are sufficient to fully define them. For a subprogram, however, the declaration only specifies the parameters required by the function or procedure; the operation of the subprogram appears later in the package body. The package declaration ends with END and the package name.



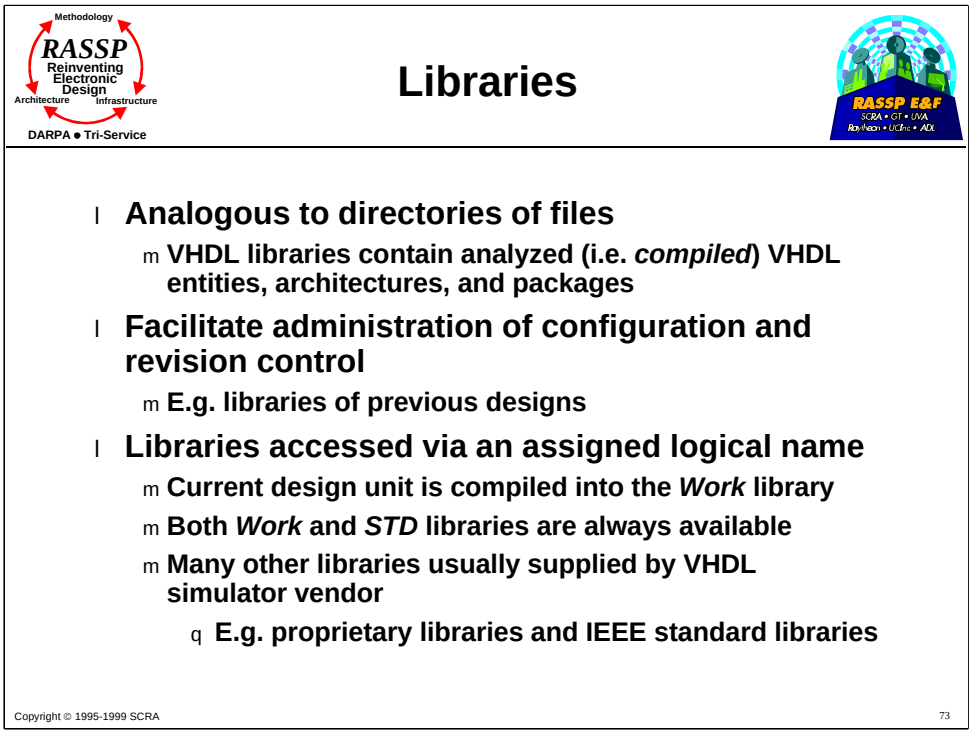
The package body contains the functional descriptions for the subprograms and other items declared in the corresponding package declaration.

Once a package is defined, its contents are made visible to VHDL entities and architectures via a USE clause which is analogous to the *include* statement of some other programming languages.



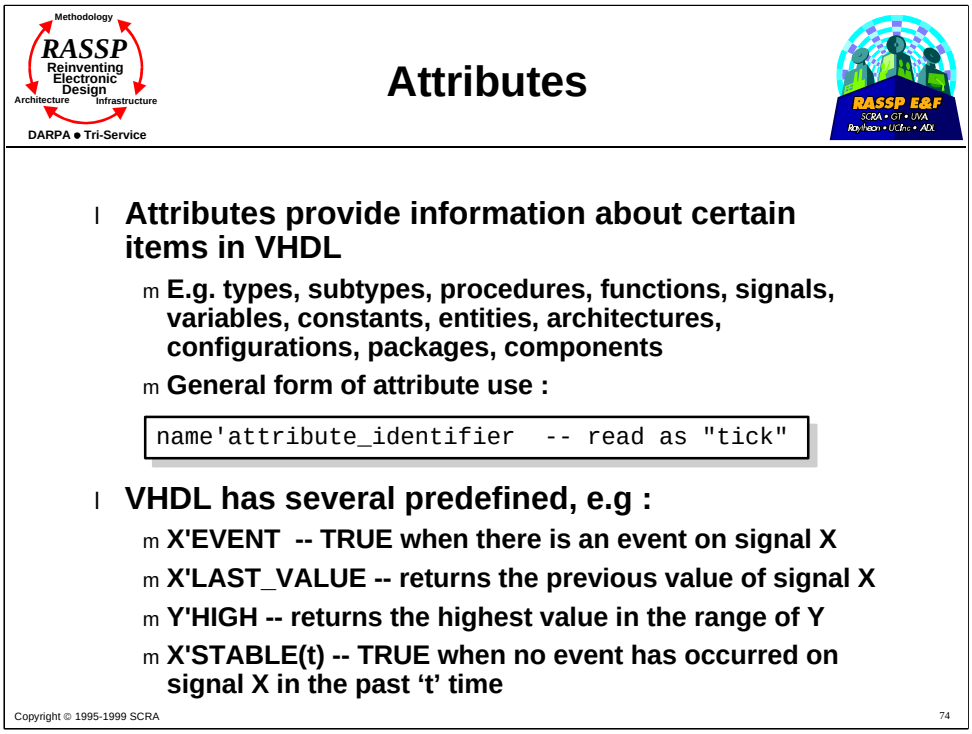
Packages are made visible to a VHDL description through the *use* of the USE clause. This statement comes at the beginning of the entity or architecture file and makes the contents of a package available within that file.

The USE clause can select all or part of a particular package. In the first example above, only the *binary* data type and *add_bits3* procedure are made visible. In the second example, the full contents of the package are made visible by use of the keyword ALL in the use clause.



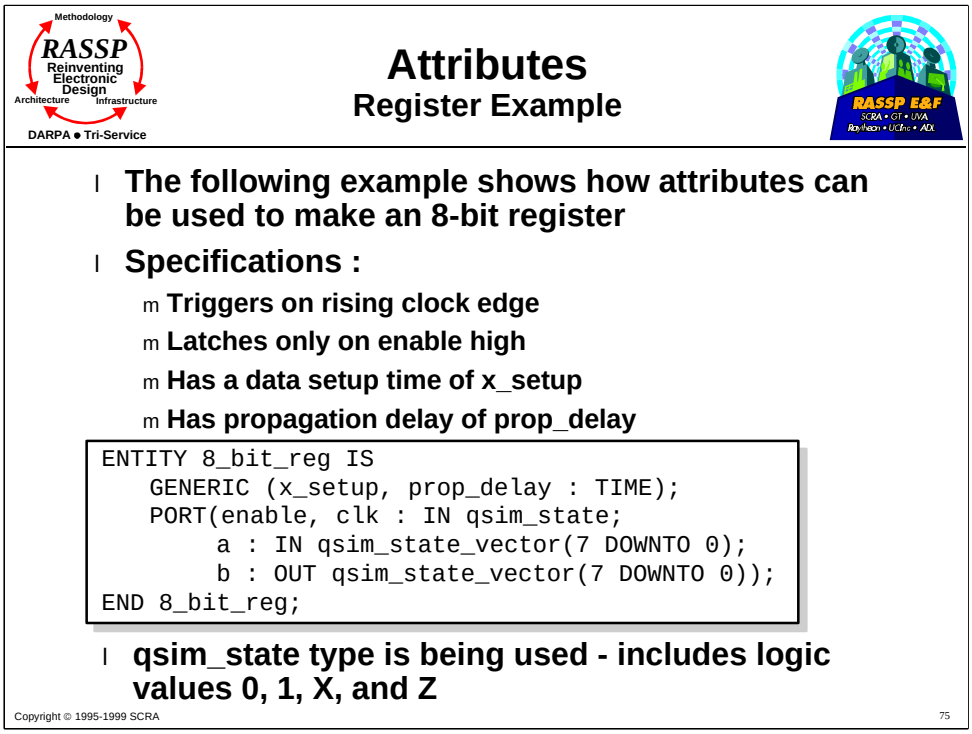
Increasingly complex VLSI technology requires configuration and revision control management. Additionally, efficient design calls for reuse of components when applicable and revision of library components when necessary.

VHDL uses a library system to maintain designs for modification and shared use. VHDL refers to a library by an assigned logical name; the host operating system must translate this logical name into a real directory name and locate it. The current design unit is compiled into the Work library by default; Work is implicitly available to the user with no need to declare it. Similarly, the predefined library STD does not need to be declared before its packages can be accessed via *use* clauses. The STD library contains the VHDL predefined language environment, including the package STANDARD which contains a set of basic data types and functions and the package TEXTIO which contains some text handling procedures.



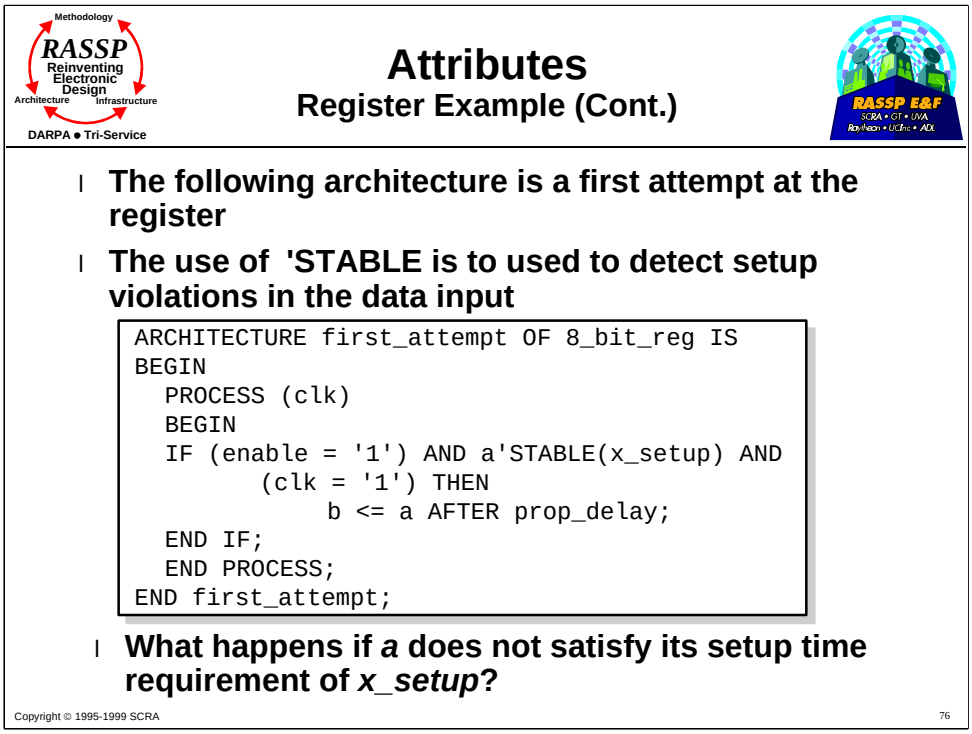
Attributes may be used to communicate information about many different items in VHDL. Similarly, attributes can return various types of information. For example, an attribute can be used to determine the depth of an array, its range, its leftmost index, etc. Additionally, the user may define new attributes to cover specific situations. This capability allows user-defined constructs and data types to use attributes. An example of the use of attributes is in assigning information to a VHDL construct, such as board location, revision number, etc.

A few examples of predefined VHDL attributes are shown above. Note that, by convention, the apostrophe marking the use of an attribute is pronounced tick (e.g. 'EVENT is pronounced "tick EVENT").



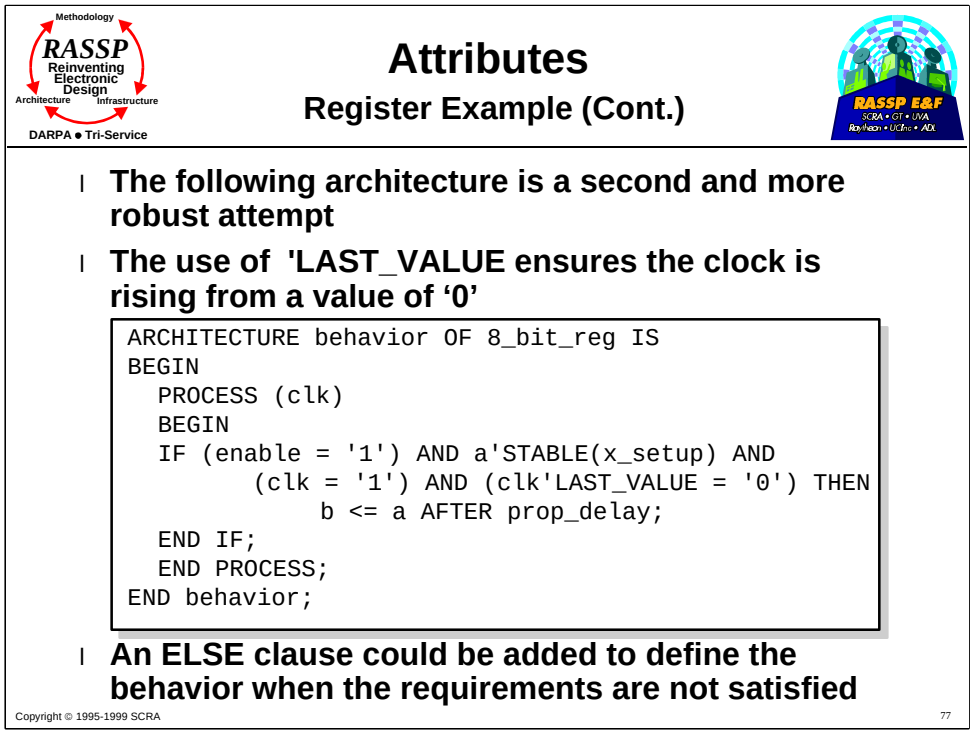
The example presented on this and the next three slides is a simple rising clock edge triggered 8-bit register with an active-high enable. The register has a data setup time of x_setup and a propagation delay of prop_delay.

The input and output signals of this register use the QSIM_STATE logic values. These values include logic 0, 1, X and Z. The *a* and *b* signals use the QSIM_STATE_VECTOR type which is an array of QSIM_STATE type vectors.

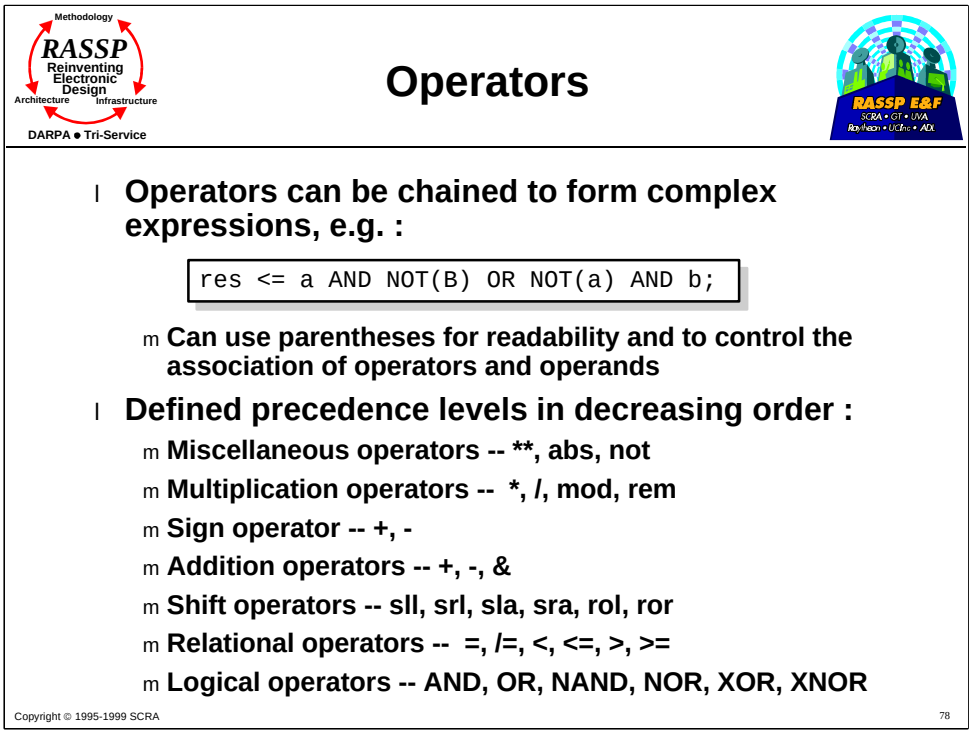


This implementation of the 8-bit register uses the 'STABLE attribute to determine if the input satisfies the setup time requirement of the register. If the setup requirement is not met, the body of the IF statement will not execute, and the value on *a* will not be assigned to *b*.

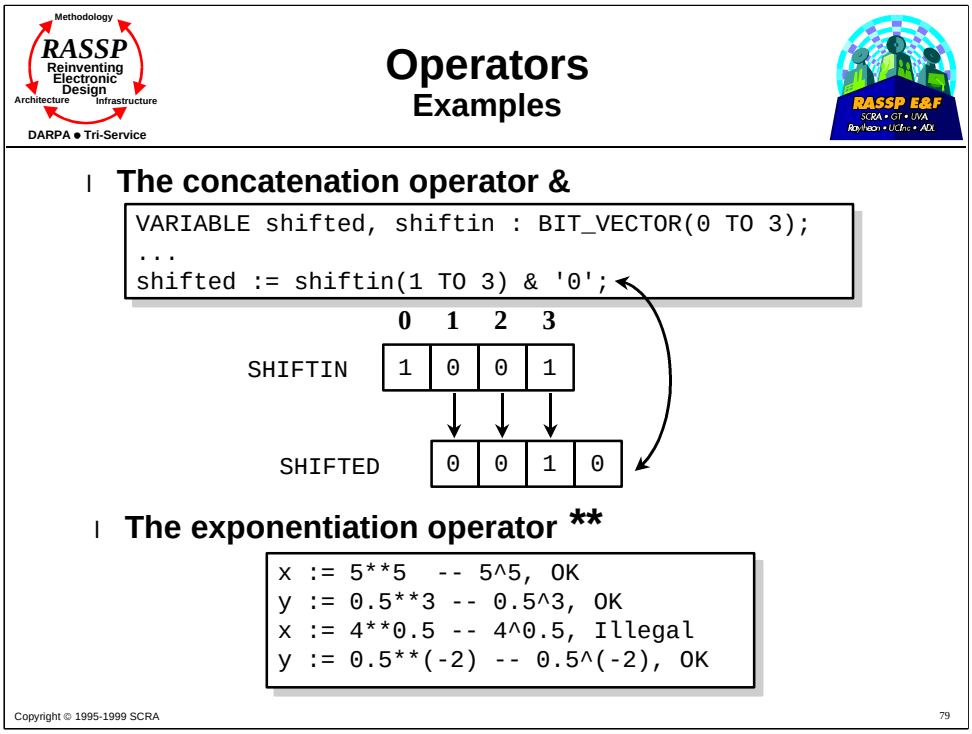
Note that although the process checks that *clk* and *enable* are '1' to store the data, it does not consider the possibility that clk may have transitioned to '1' from either 'X' or 'Z'.



This implementation adds a check for '0' to '1' transitions on clk by using the 'LAST_VALUE attribute on the signal clk.



The list of predefined operators in VHDL is shown above. The logical and relational operators are similar to those in other languages. The addition operators are also familiar except for the concatenation operator which will be discussed in the next slide. The multiplication operators are also typical (e.g. the mod operator returns the modulus of the division and the rem operator returns the remainder). Finally, the miscellaneous operators provides some useful frequently used functions.



The concatenation operator joins two vectors together. Both vectors must be of the same type. The example given above performs a logical shift left for a four bit array.

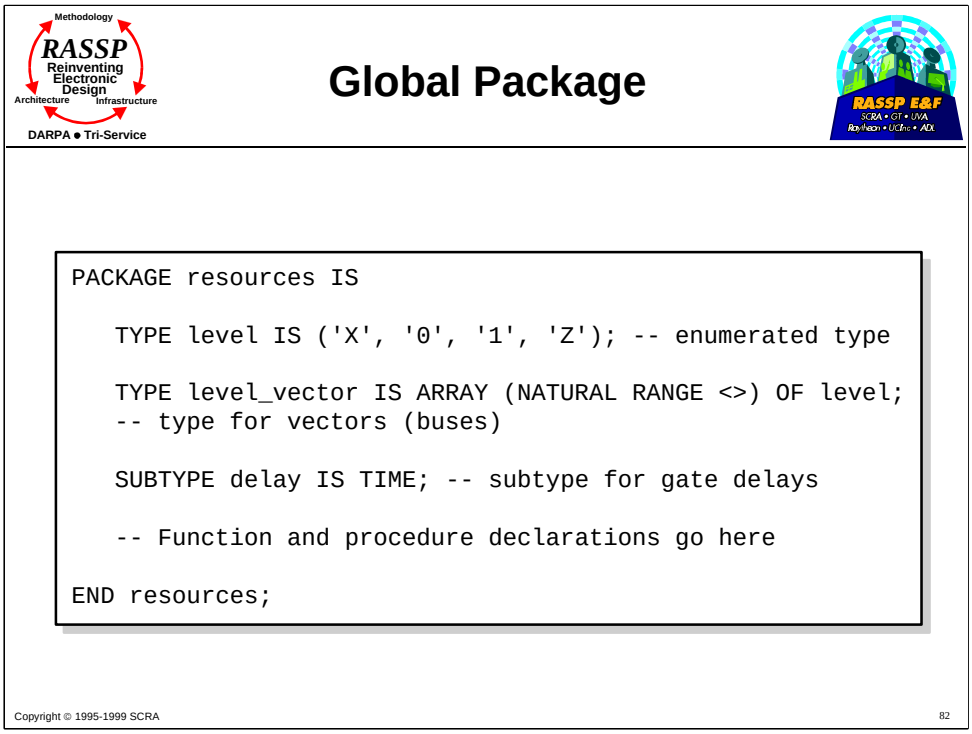
For the exponentiation operator ** from the package STD, the exponent must be an integer; no real exponents are allowed. Negative exponents are allowed only with real numbers. Other packages can be found that include overloaded operators (discussed in Module 12) for exponentiation with real and negative arguments.

Module Outline

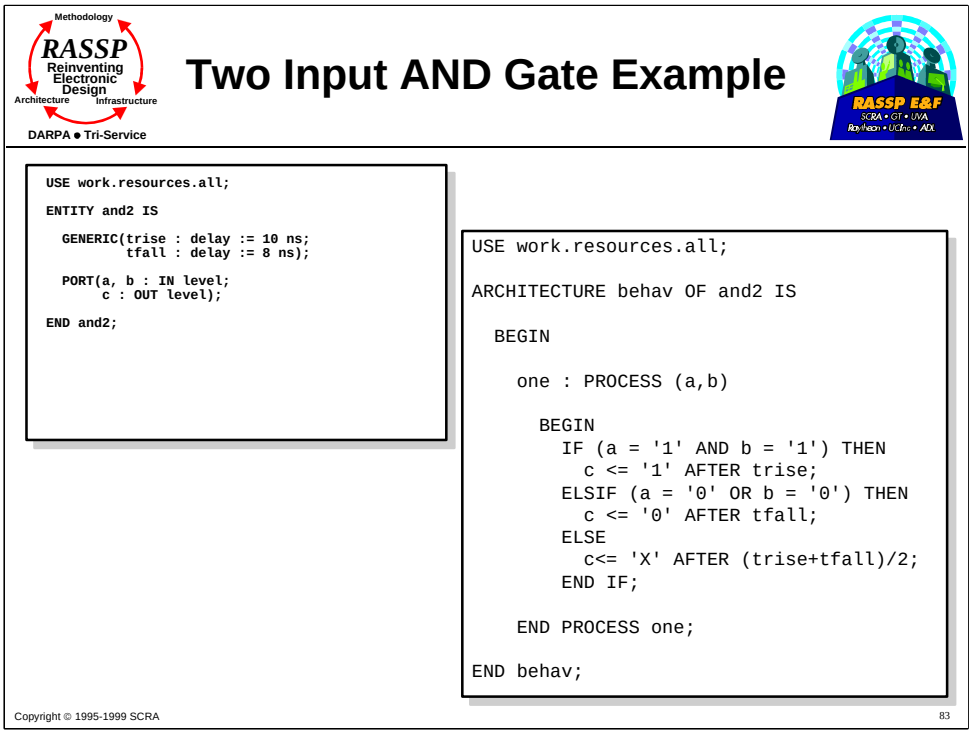
- Introduction
- VHDL Design Example
- VHDL Model Components
- Basic VHDL Constructs
- **Examples**
- Summary

Examples

- **Build a library of logic gates**
  - **AND, OR, NAND, NOR, INV, etc.**
- **Include sequential elements**
  - **DFF, Register, etc.**
- **Include tri-state devices**
- **Use 4-valued logic**
  - **‘X’, ‘0’, ‘1’, ‘Z’**
  - **Encapsulate global declarations in a package**



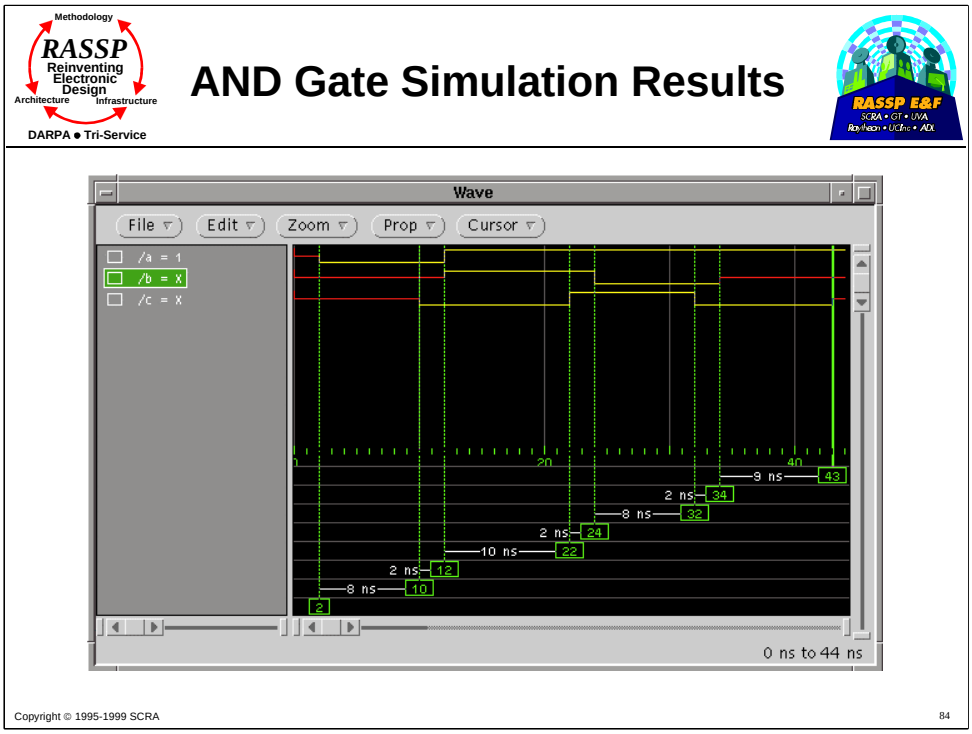
This is the package declaration for the user defined four valued type package. It includes the four valued enumerated type itself, a vector or array of that type, and a subtype of type *time* to be used for delay values. Functions and/or procedures could be declared in the package (with their actual implementation descriptions included in the package body), but that will be deferred until Module 12, Behavioral VHDL.



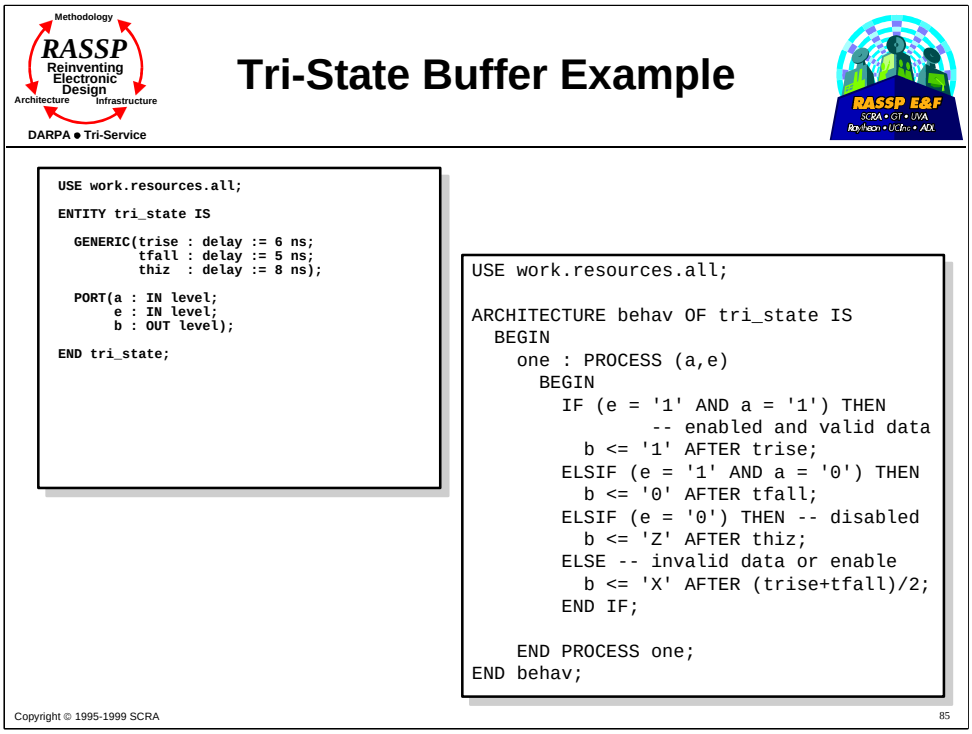
This is a simple 2 input AND gate. Note that the entity includes generics for rise time and fall time, and the two input and one output ports.

The architecture contains the “behavior” of the AND gate. A single process statement is used which executes anytime either the *a* or *b* inputs change (because they are in the process sensitivity list - see module 12). A simple if statement is used to determine what the correct output should be, and the proper delay is inserted by the AFTER clause.

Note that the USE construct is needed in both cases if Entity and Architecture code segments are contained within separate files.



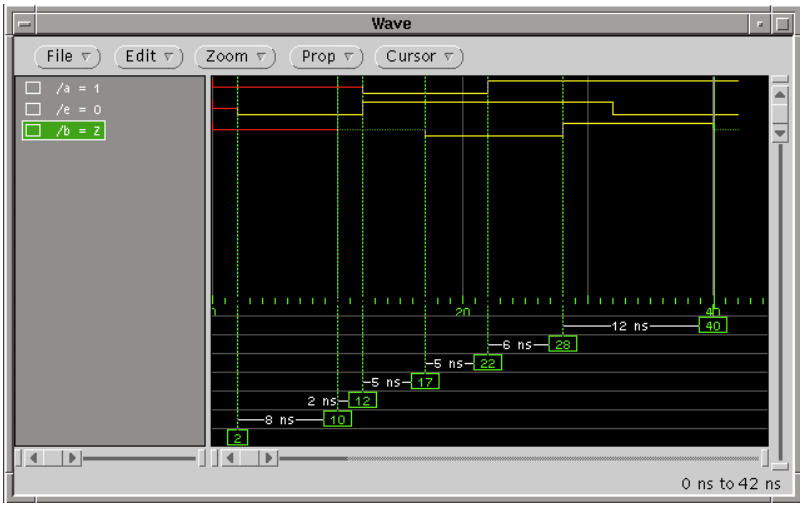
This is the simulation results output for the two input AND gate. The Mentor Graphics' QuickVHDL simulator was used.

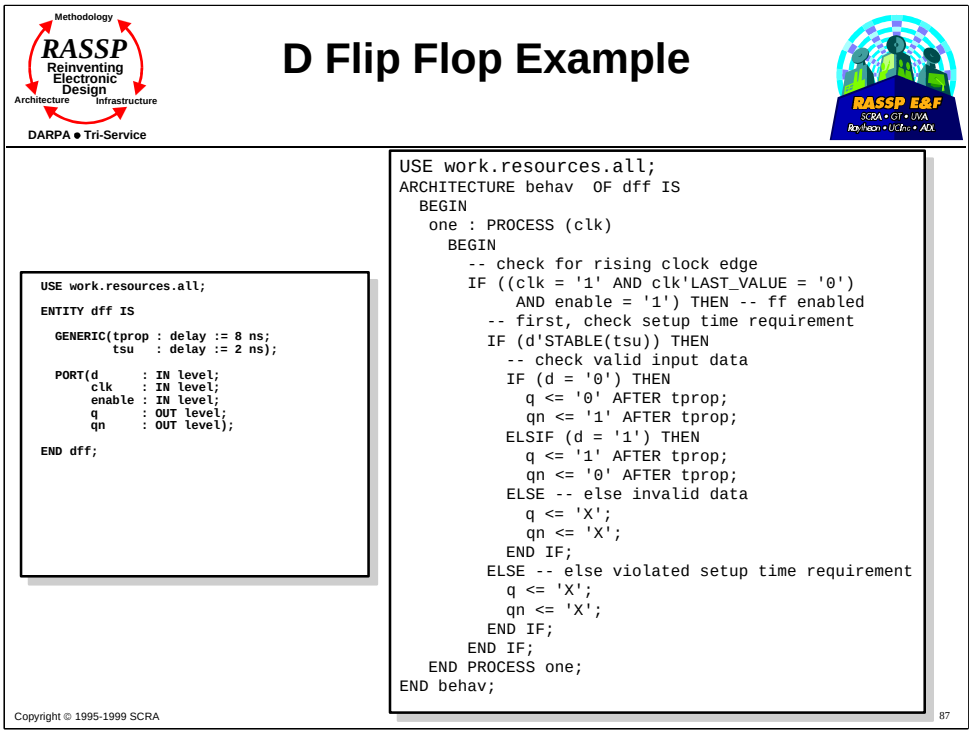


This is a tri-state buffer example. It is similar to the AND gate, but in this case, it uses the ‘Z’ value as well as the ‘X’ value. Also, a *thiz* delay (to indicate a driver “turn off” time) is used in addition to the rise and fall delay times.

Note that the USE construct is needed in both cases if Entity and Architecture code segments are contained within separate files.

Tri-State Buffer Simulation Results



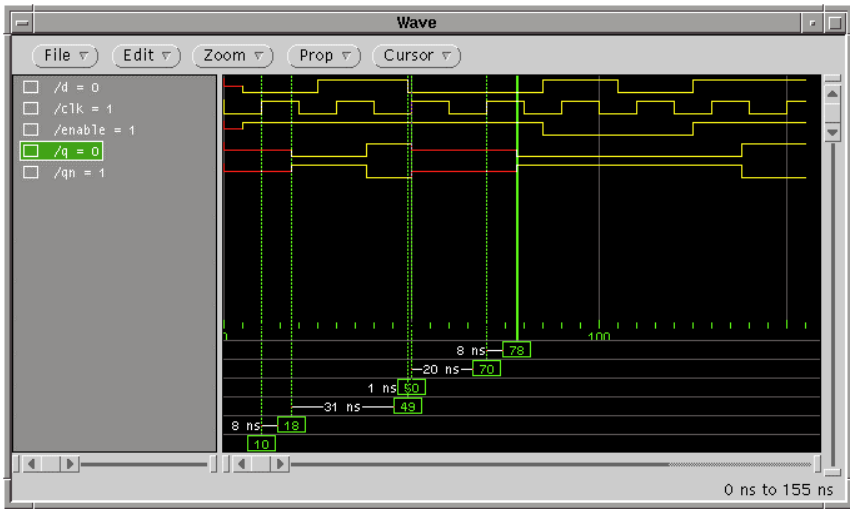


This is a DFF example that illustrates the use of signal attributes. Notice the ‘LAST_VALUE attribute is used in the clock statement to recognize a ‘0’ to ‘1’ rising edge transition (the last value has to be a ‘0’ to avoid triggering on ‘X’ or ‘Z’ to ‘1’ transitions).

Also, the ‘STABLE attribute is used at each rising clock edge to determine if the *d* input has satisfied the setup time requirement.

Note that the USE construct is needed in both cases if Entity and Architecture code segments are contained within separate files.

D Flip Flop Simulation Results



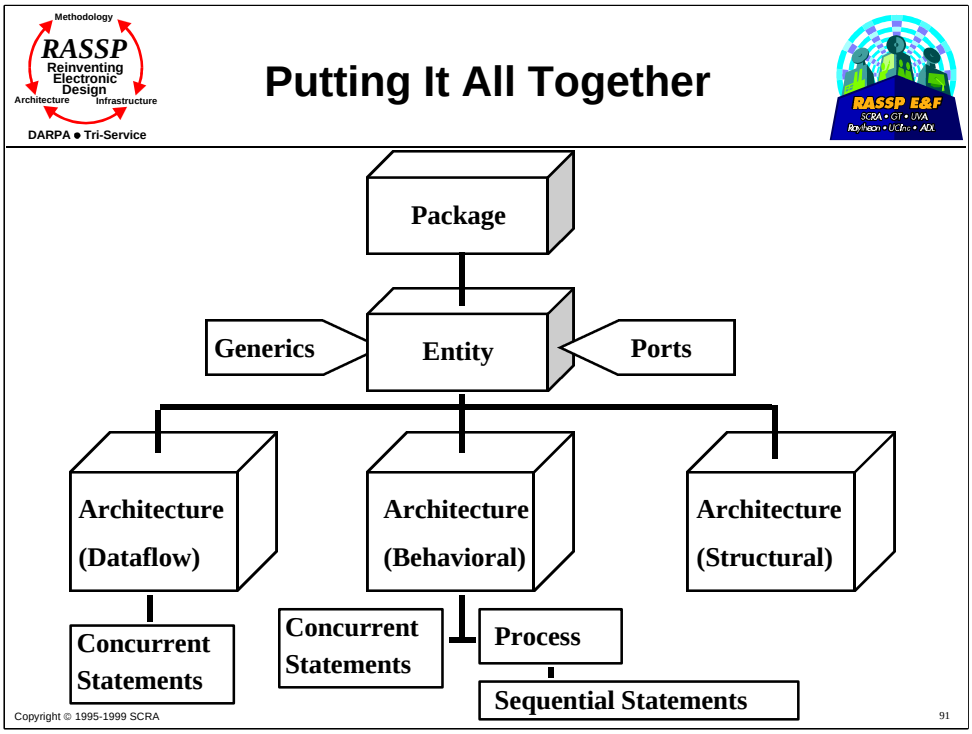
This is the simulation results. Notice that there is a case where an ‘X’ is output when the input fails to satisfy the setup time requirement.

Module Outline

- Introduction
- VHDL Design Example
- VHDL Model Components
- Basic VHDL Constructs
- Examples
- **Summary**

Summary

- VHDL is a worldwide standard for the description and modeling of digital hardware
- VHDL gives the designer many different ways to describe hardware
- Familiar programming tools are available for complex and simple problems
- Sequential and concurrent modes of execution meet a large variety of design needs
- Packages and libraries support design management and component reuse



This diagram is a graphical representation of many of the VHDL constructs talked about in this module. In summary, generics and ports are used in the entity definition which serves as the module's interface to other modules. Each entity can have any number of different descriptions of module behavior included in VHDL architectures (although only one architecture can be instantiated per module use). Architectures use concurrent statements and possibly processes to allow great flexibility in how behavior is described.

[MG93]

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.

[Gajski83] Gajski, Daniel D. and Kuhn, Robert H., "Guest Editors Introduction - New VLSI Tools", IEEE Computer, pp 11-14, IEEE, 1983; © IEEE 1983

[Hein98] Hein, et al, "VHDL Modeling Terminology and Taxonomy," Version3.0, July 29, 1998.

[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.

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

[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

[Smith88] Smith, David, "What is Logic Synthesis", *VLSI Design and Test*, October, 1988.

References, cont.

[USE/DA94] USE/DA Standards Survey, 1994.

[VI93] VHDL International Survey, 1993.

[Walker85] Walker, Robert A. and Thomas, Donald E., "A Model of Design Representation and Syntheses", 22nd Design Automation Conference, pp. 453-459, IEEE, 1985

[Waxman89A] Waxman, R., Saunders, L.F., and Carter, H.C., “Abolishing the Tower of Babel,” Spectrum, Vol. 26, Number 5, May 1989, pp. 40-44.

[Waxman89B] R. Waxman and L. Saunders, The Evolution of VHDL, Invited Paper, INFORMATION PROCESSING '89, G.X. Ritter (ed.), Elsevier Science Publishers B.V. (North Holland), copyright IFIP, 1989, PP. 735-742.

[Williams94] Williams, R. D., "Class Notes for EE 435: Computer Organization and Design", University of Virginia, http://www.ee.virginia.edu/research/CSIS/, 1994.