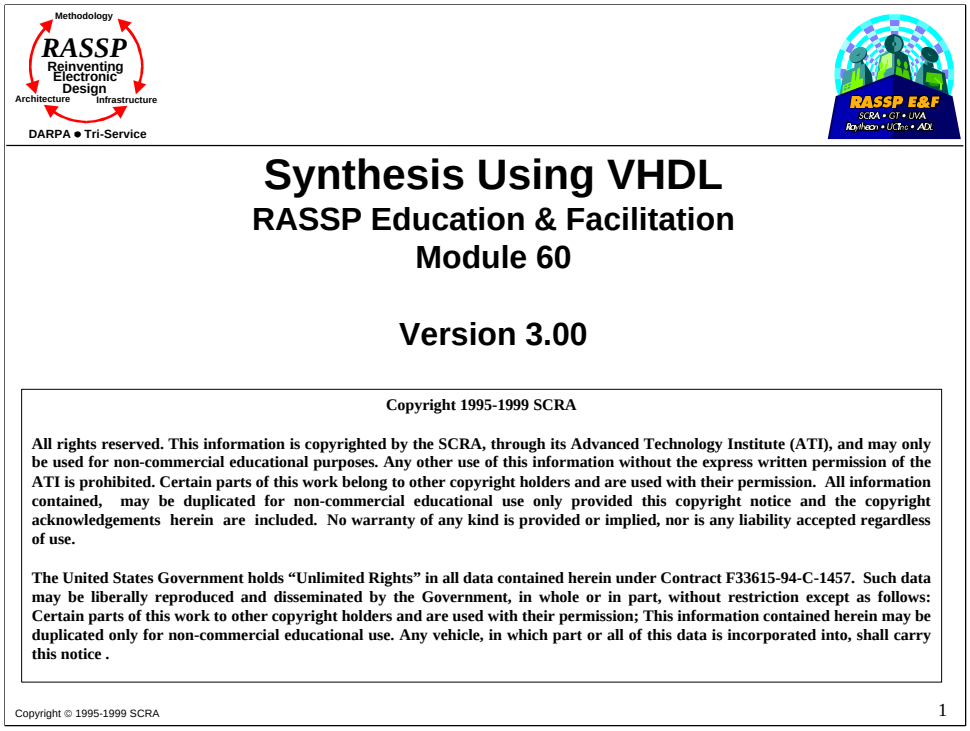
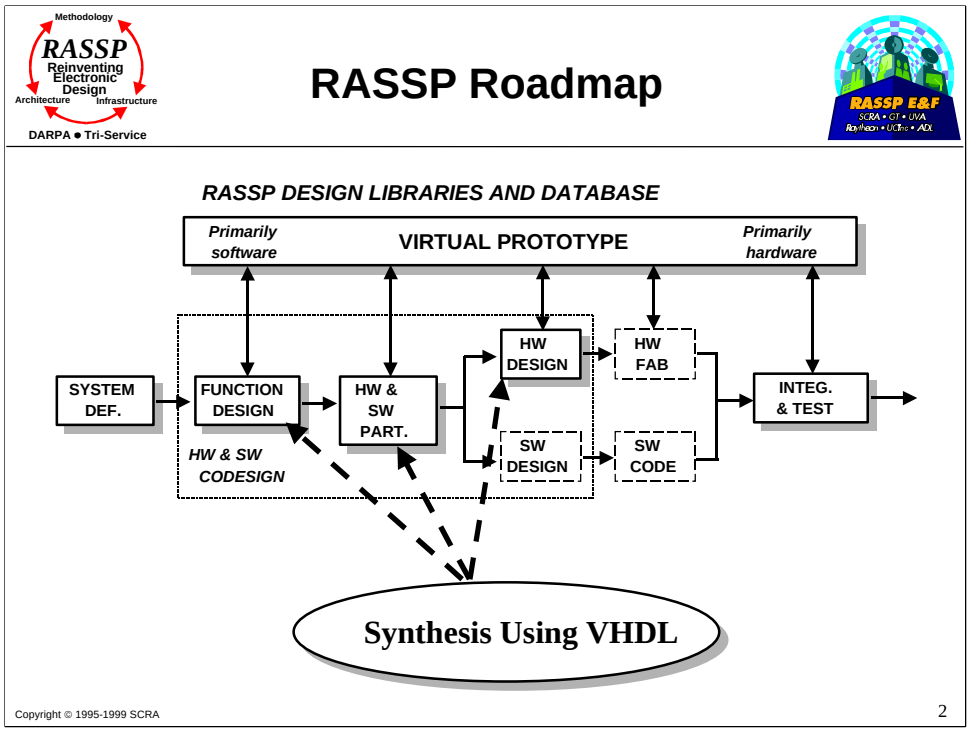
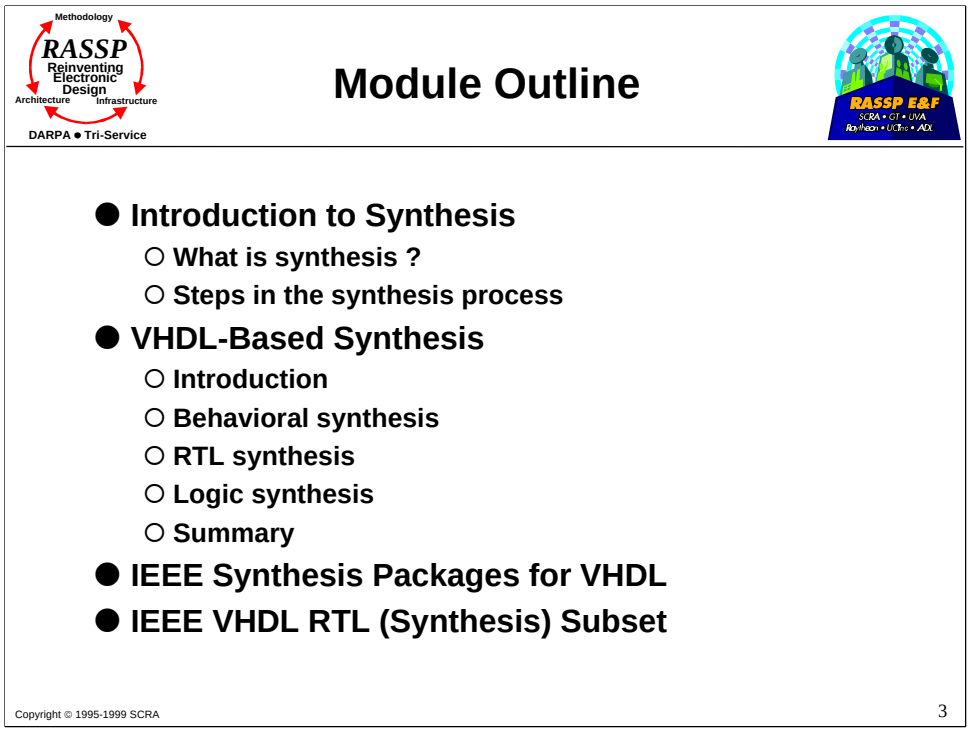


This module describes how one can synthesize digital systems using VHDL. It does not teach VHDL, nor does it teach synthesis. The former is the task of earlier modules, while the latter is the task of various synthesis tools that can take in an input specification in VHDL and process it .



In this module, we describe how VHDL can be used to specify digital circuits in a form that is synthesizable. The basic premise is that the entire VHDL Language Reference Manual (LRM) recommendations for VHDL 1993 cannot be synthesized by commercial tools, and hence the synthesizable subset of VHDL is one that constrains the VHDL 1993 standard to those features that are supported by synthesis tools.

This would have been a very difficult task, given the large number of synthesis tools (non standard) and the variety of target technologies (FPGA, ASIC, etc) available to the designer (which impact the way they design using VHDL), however, the recent proposal for standardization of the so-called RTL subset of VHDL has given us confidence that the material covered in this module is not tool-specific or technology-specific to a large degree, and will be supported by a large number of vendors once the synthesis standard is ratified in 1999.



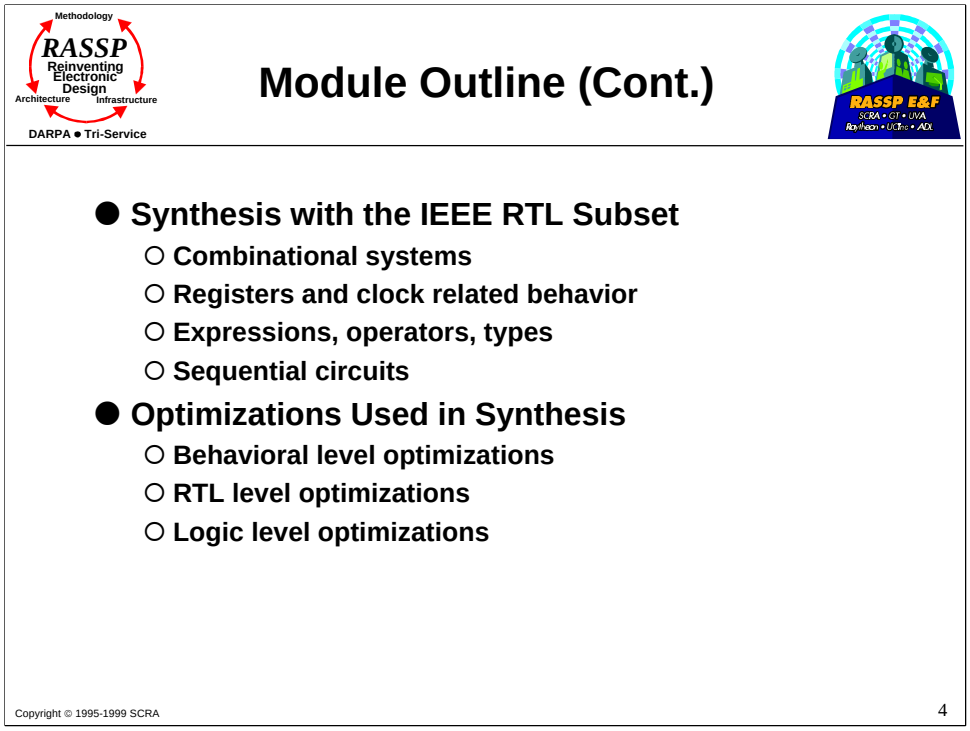
In this module, we describe how VHDL can be used to specify digital circuits in a form that is synthesizable. The basic premise is that the entire VHDL Language Reference Manual (LRM) recommendations for VHDL 1993 cannot be synthesized by commercial tools, and hence the synthesizable subset of VHDL is one that constrains the VHDL 1993 standard to those features that are supported by synthesis tools.

In the fist section we define the synthesis process and various metrics and objectives that characterize it.

Originally, synthesis tools could only synthesis structural descriptions at the logic and gate levels of description. Currently, synthesis tools can support synthesis at higher levels of abstraction in VHDL, promising a greater productivity and efficiency. The second section describes this migration of the synthesis process to higher levels of abstraction.

The third section describes packages, ratified by IEEE, that support the synthesis process.

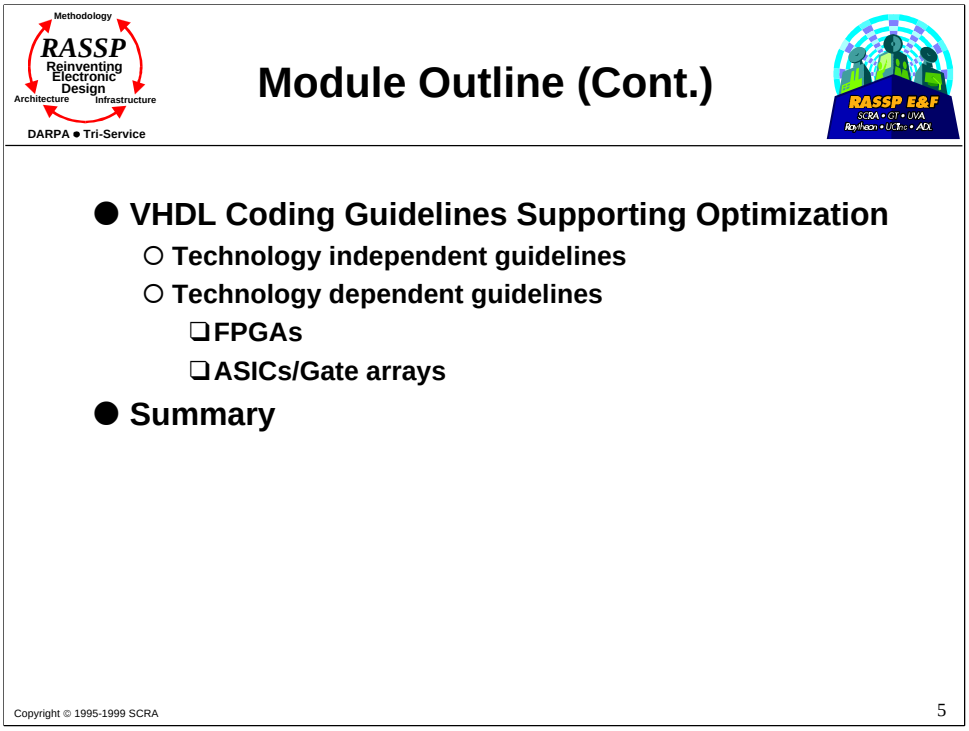
The fourth section describes a recent effort by the Synthesis Interoperability



Working Group (SIWG, http://www.vhdl.org/siwg/) that proposed a subset of VHDL as a candidate for a IEEE Synthesis standard. Clearly, the entire VHDL language cannot be synthesized, thus the proposal represented a giant step forward in the development of portable and efficient synthesis capabilities for digital design using VHDL.

We then describe, with some examples, the use of the IEEE VHDL RTL Subset. It must be noted that the reader is assumed to understand VHDL well, as the focus of the module is not on teaching VHDL, but on the synthesis support for VHDL. Furthermore, this module should also not be considered a tutorial on the synthesis process itself.

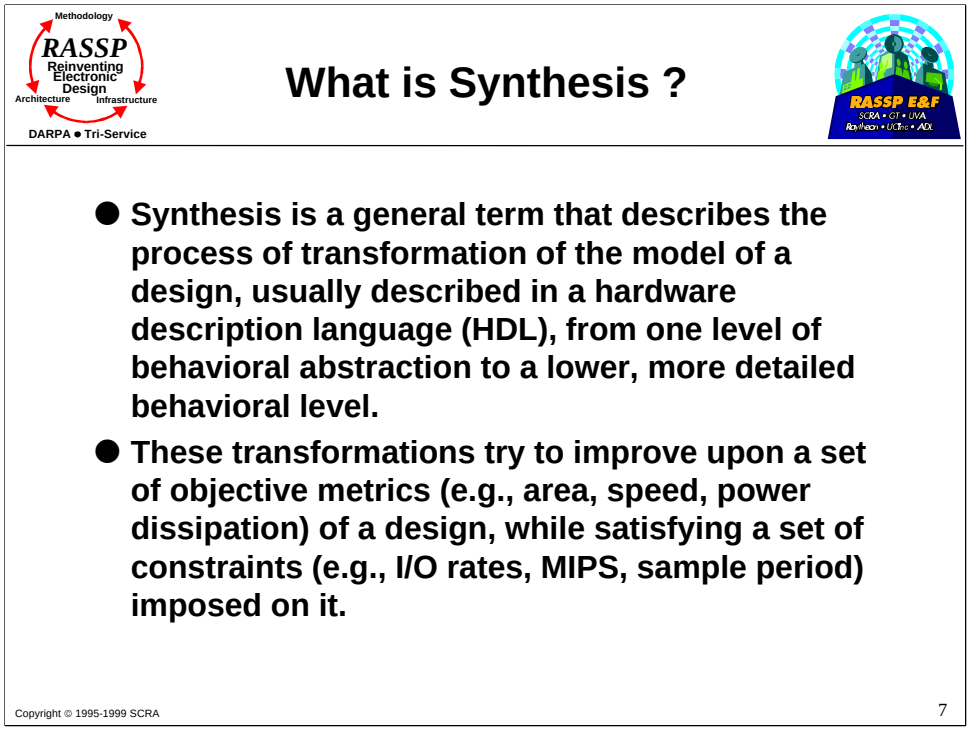
Synthesis optimizations can occur at various levels of abstraction, and the section on optimizations provides a brief introduction to these topics.



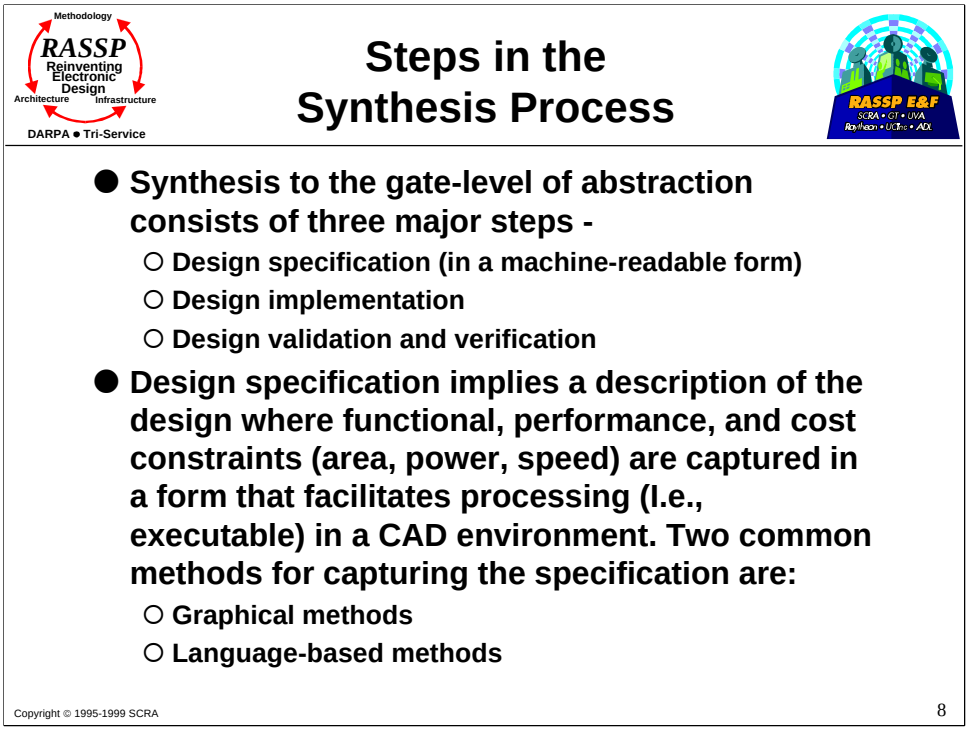
VHDL can be written in a manner that allows the user to take benefit of the optimizations provided and supported by various synthesis tools and target technologies. The section on Coding Guidelines provides this insight via a few well-chosen examples.

Module Outline

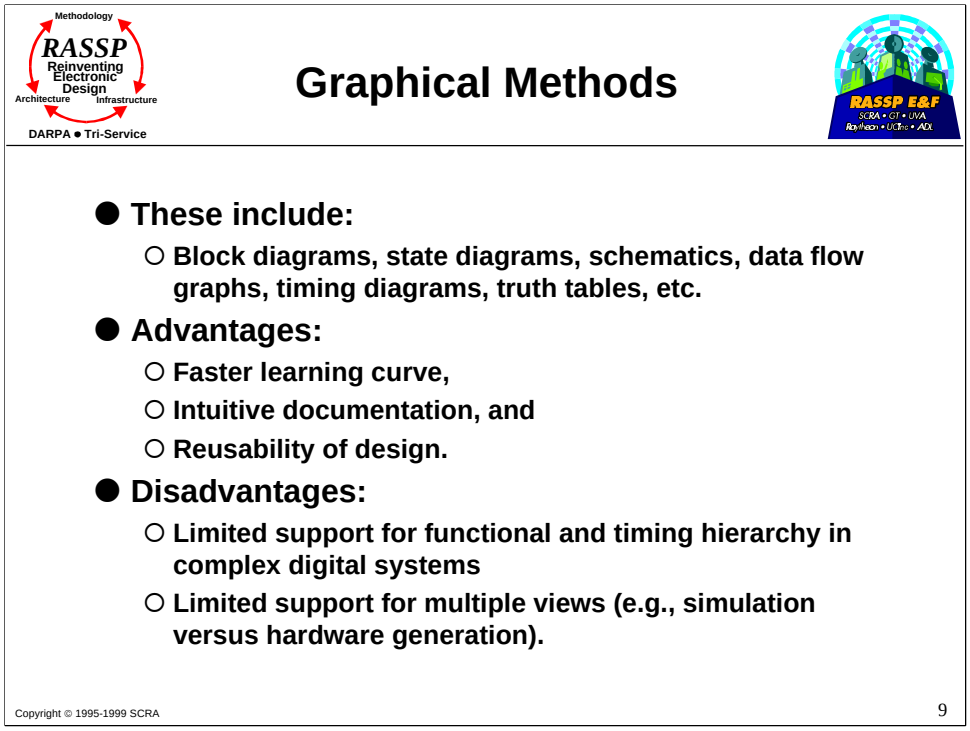
- **Introduction to Synthesis**
- VHDL-Based Synthesis
- IEEE Synthesis Packages for VHDL
- IEEE RTL (Synthesis) Subset for VHDL
- Synthesis with IEEE RTL Subset
- Optimizations Used in Synthesis
- VHDL Coding Guidelines Supporting Optimization
- Summary



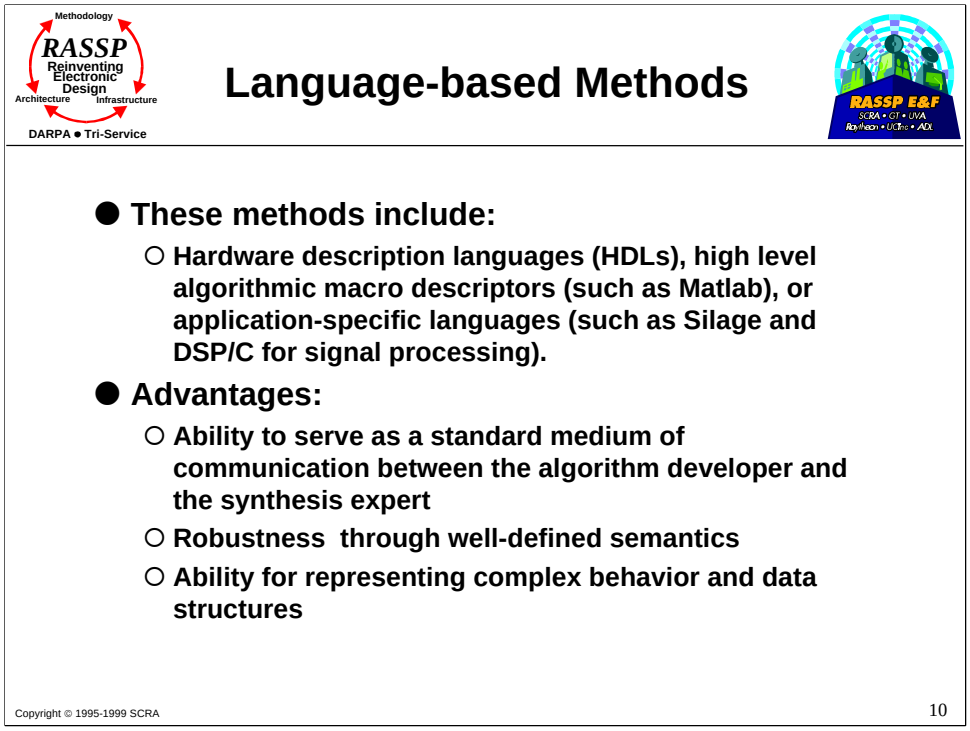
Synthesis in its most generic form simply refers to the incorporation of additional lower level implementation details into a digital design, however, most current tools expect the output to be a net list of gates that are optimized for area, power, or latency.



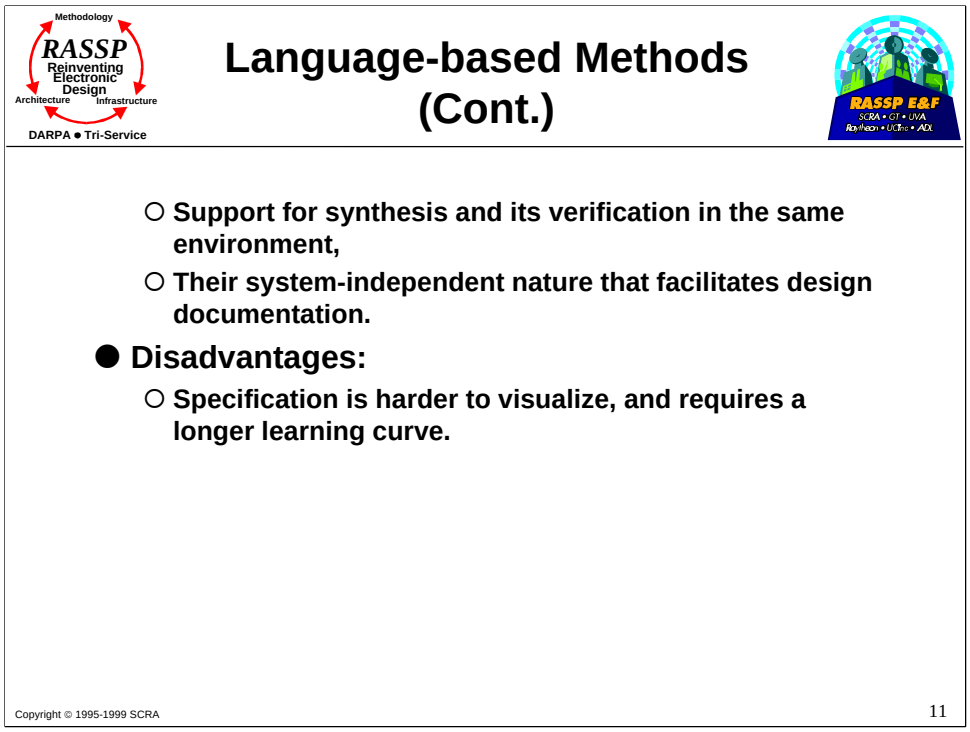
Many consider graphical methods and language methods to be indistinguishable, in that one can be quickly converted to another. It is often the designer's choice as to which mechanism they find convenient when specifying a complex digital design



A number of vendors are providing for mixed graphical and language support, where certain blocks are pieces of VHDL code, while others are graphical descriptions of controllers, etc. The “best choice” appears to depend again on the designer and the domain application.

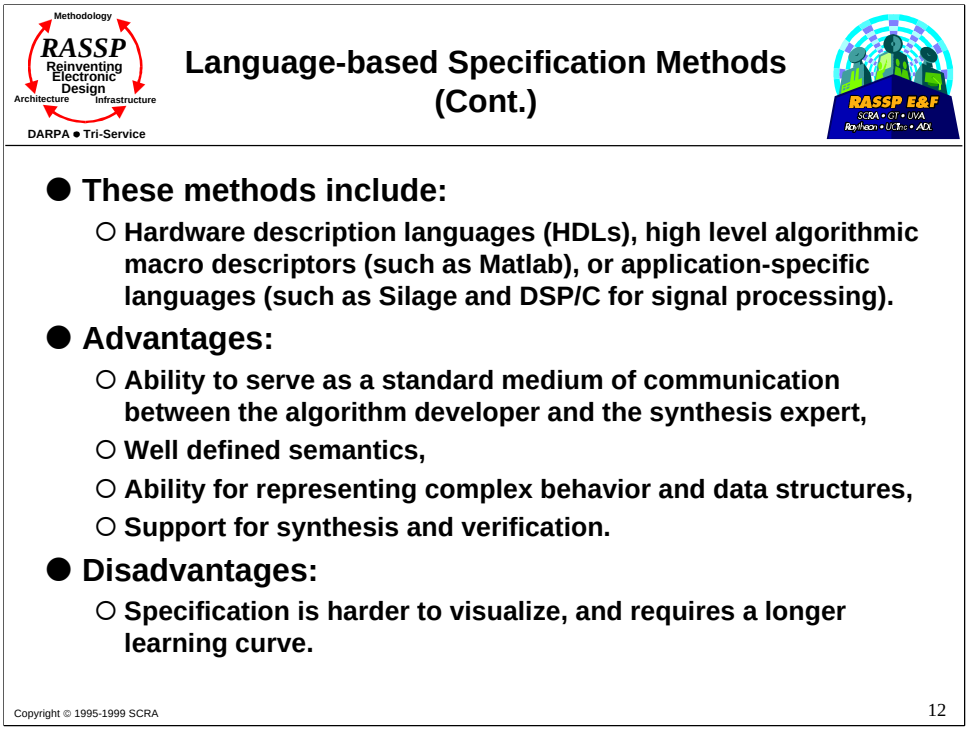


Current popular language-based synthesis methods include the use of VHDL and Verilog, with the market evenly split. Verilog is a concise and restrictive HDL that appears to be a bit more convenient at lower levels of abstraction for some designers, while VHDL appears to be more flexible and powerful at all levels of abstraction. Mixed co-simulation environments exist for mixed Verilog and VHDL designs (the co- in co-simulation refers to the ability to simulate a design description written using a mixture of VHDL and Verilog).

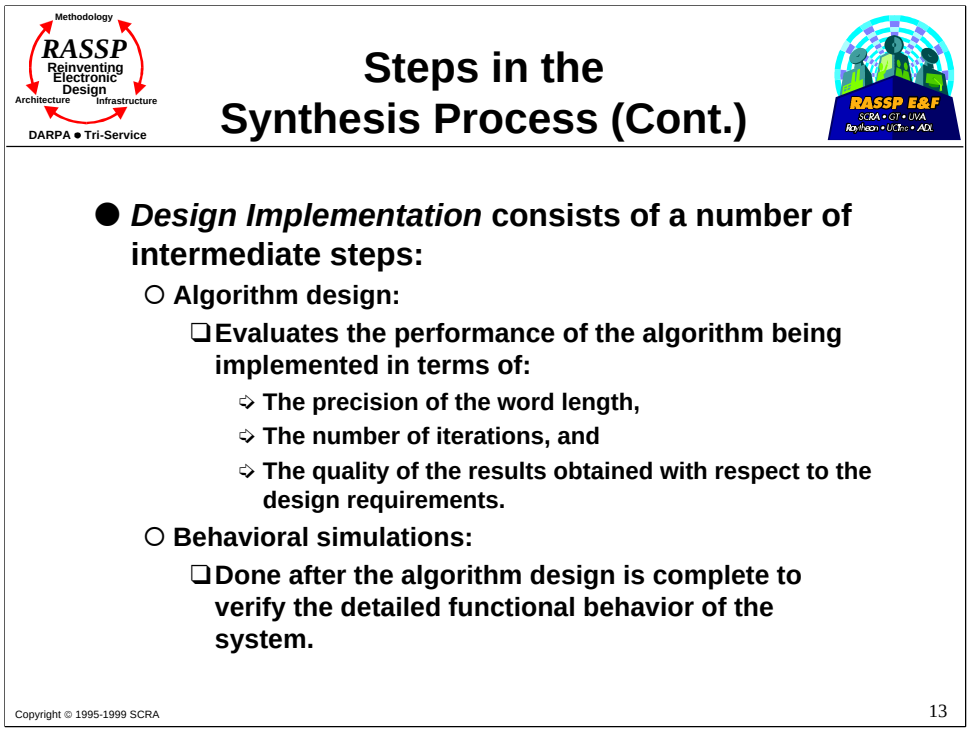


Language-based methods and graphical methods have been studied in other contexts (e.g., software) and their relative merits are still discussed within the community.

Suffice it to say that translators exist from one domain to another.

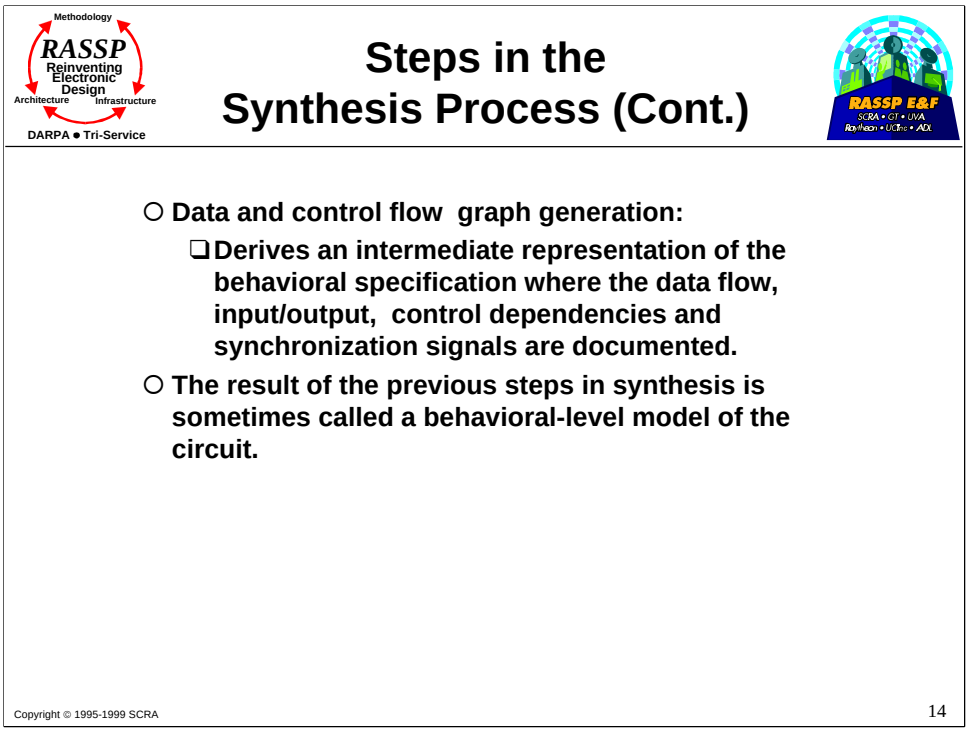


Current popular language-based synthesis methods include the use of VHDL and Verilog, with the market evenly split. Verilog is a concise and restrictive HDL that appears to be a bit more convenient at lower levels of abstraction for some designers, while VHDL appears to be more flexible and powerful at all levels of abstraction. Mixed co-simulation environments exist for mixed Verilog and VHDL designs.

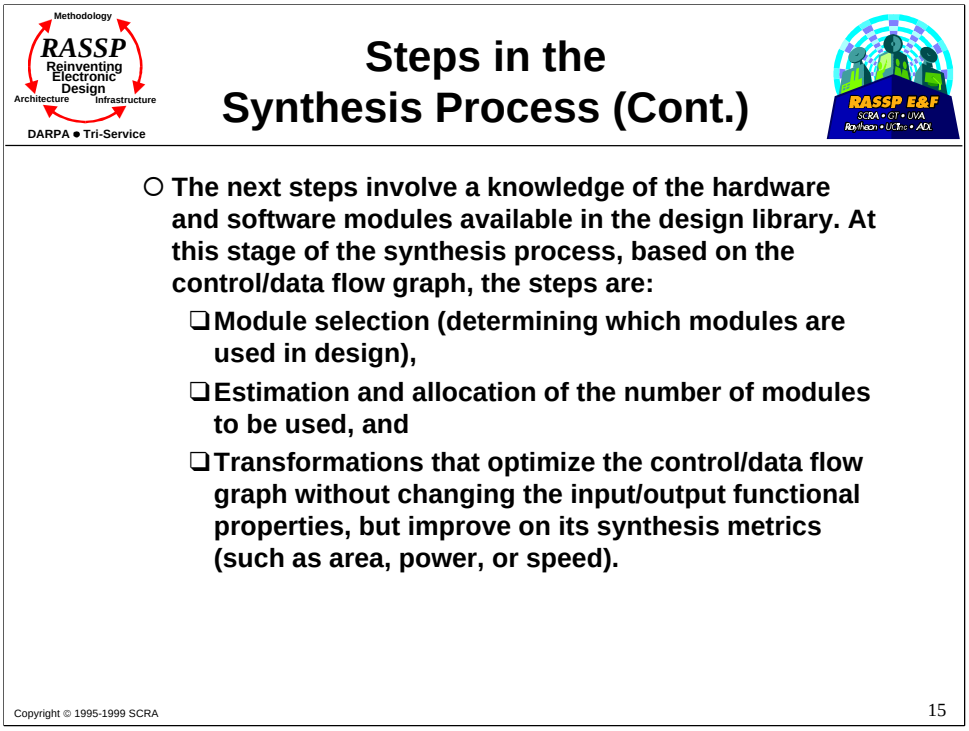


Here we describe the general process of digital design. The top down design process is described, and most VHDL synthesis tools support design entry at one of the steps in the top down design process.

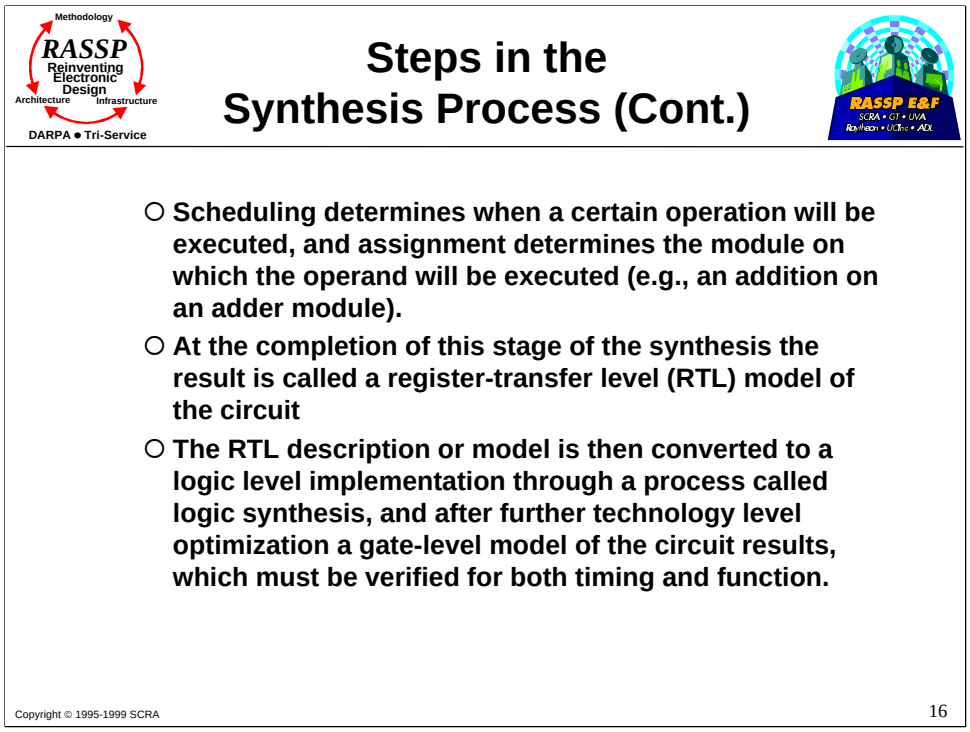
Behavior implies both functional and timing characteristics of a digital system.



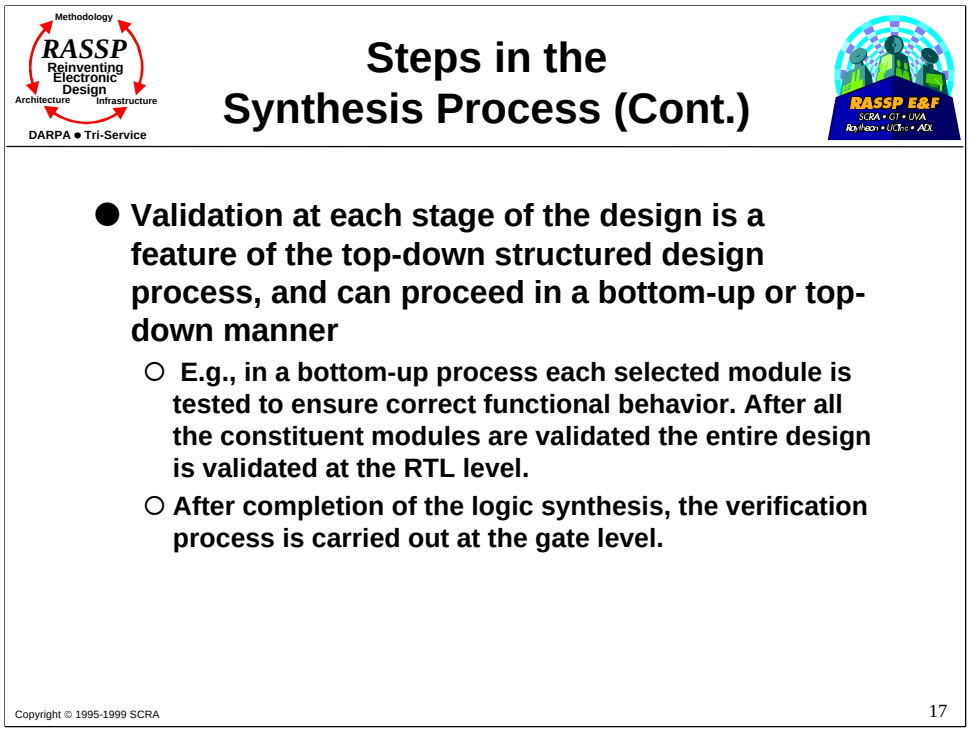
After the behavioral design is completed at the top level, the data and control flow design allows us to capture the concurrent aspects of the specification that drive the implementation aspects of the resource allocation and assignment steps.



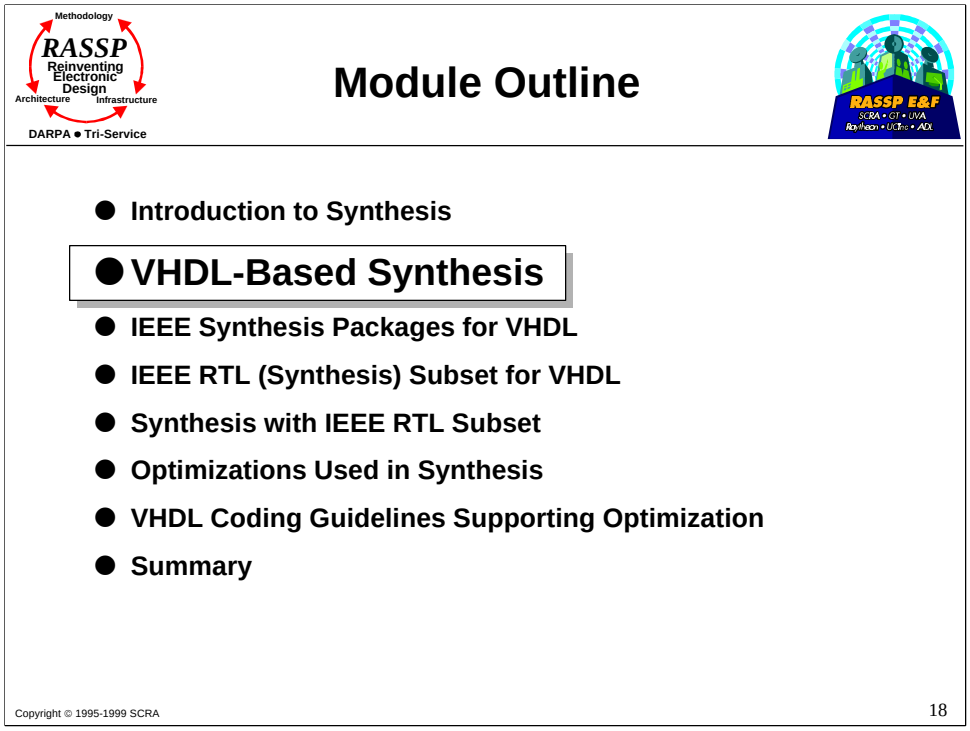
The steps of assignment, allocation, and scheduling should be clearly distinguished. However, the order in which they are done is implementation and constraint dependent, and changing the allocation can result in a change in the assignment and the scheduling steps. The steps of scheduling, allocation, and assignment are interrelated and several iterative algorithms are utilized by CAD environments that attempt to meet the user objectives subject to design constraints.



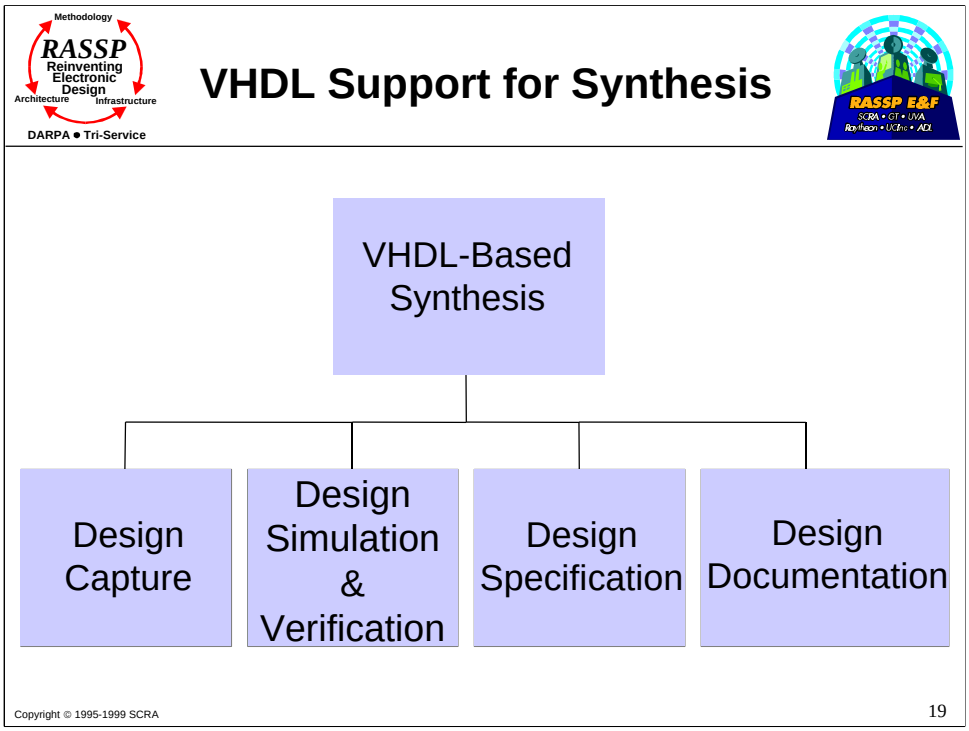
Note that Validation and Verification differ from each other. The difference is captured in the next slide.



Verification and validation differ from each other in some respects. Validation ensures that the implementation matches the design requirements from the customer (i.e., the right system is synthesized), Verification ensures that the implementation is consistent with the specification (i.e., the system is designed right).



Now that we have introduced the process of synthesis, we will now study how VHDL supports this process. It should be noted that VHDL supports this process and does not define the synthesis process.



Synthesis is a process that is independent of VHDL. However, VHDL assists synthesis through its four major roles listed in the slide.

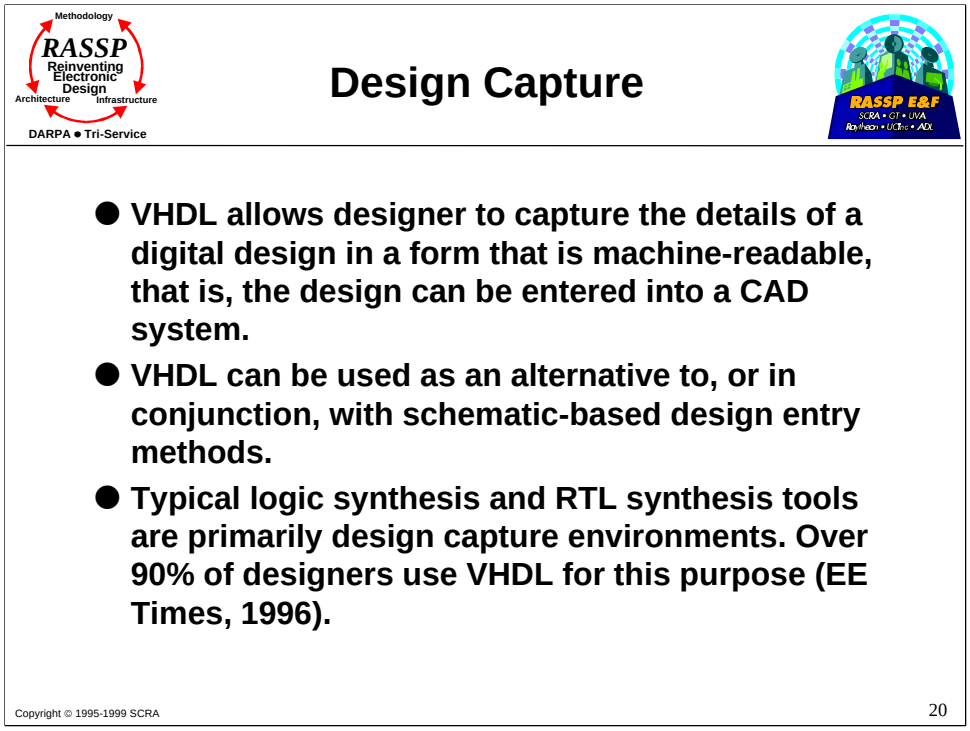
In Design Capture, one already has a digital design in mind, and this design is translated to VHDL so that it can be processed directly by a synthesis tools.

In Design Simulation & Verification, a test bench and a captured design are simulated to ensure that the design works correctly.

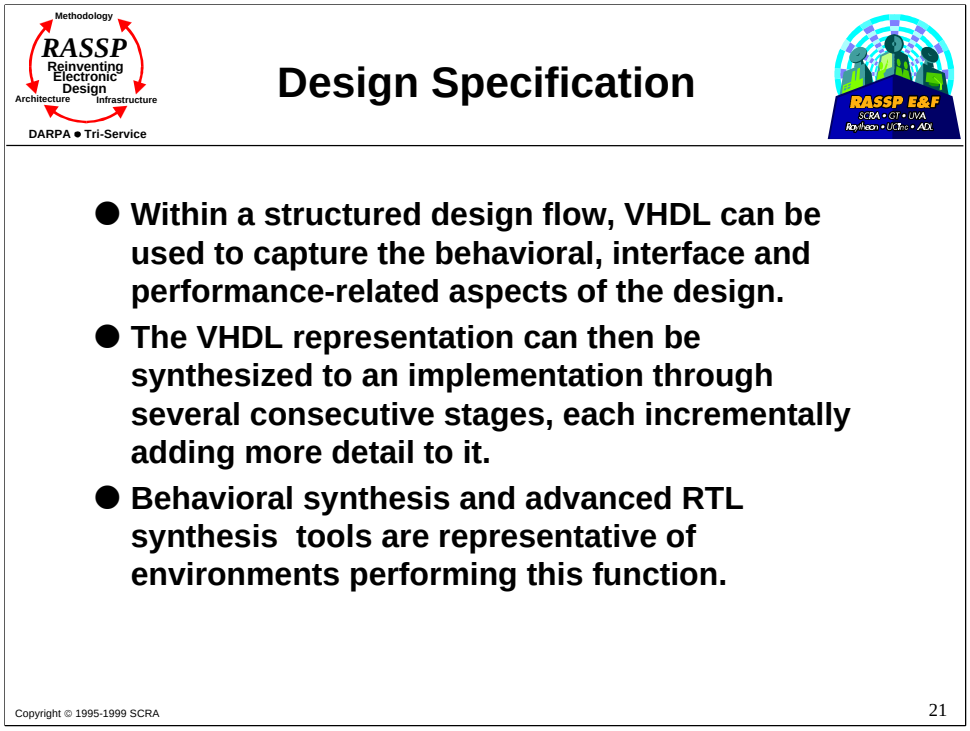
In Design Specification, arguably the most powerful use of VHDL, the design can be specified at a very high level of abstraction, and the synthesis tool can then take this description and translate it to lower levels of design abstraction subject to the enforced constraints.

In Design Documentation one can use VHDL as an executable version of the system that has been designed or is under design.

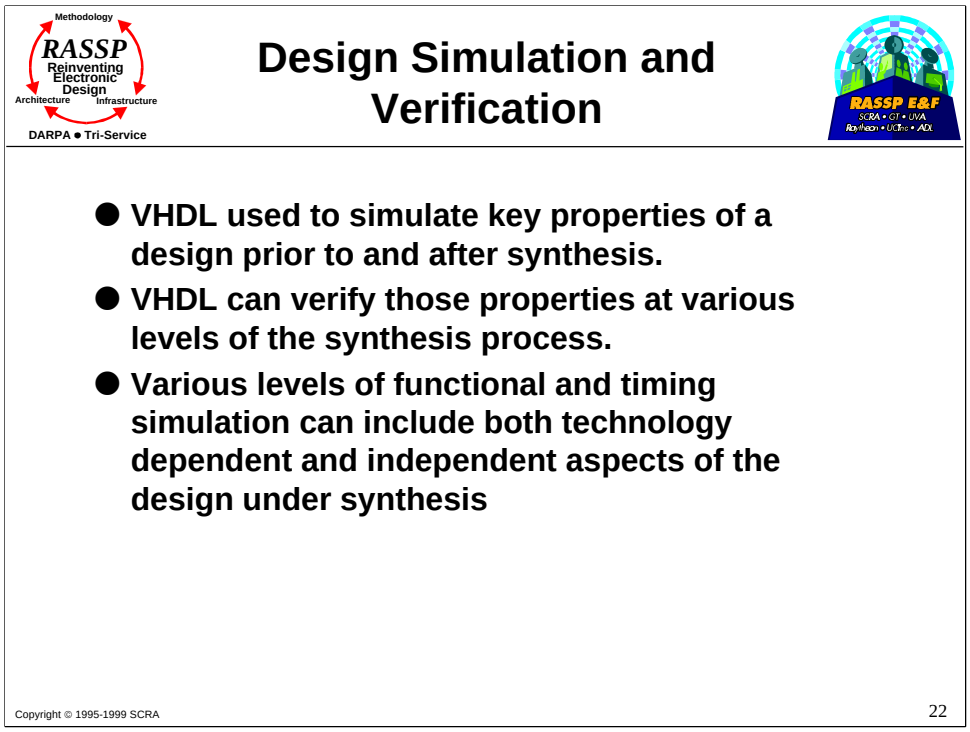
Most of the digital designers (>90%) use VHDL for design capture, while other users (especially for complex applications in telecommunications and military) rely on VHDL for the other three functions it supports.



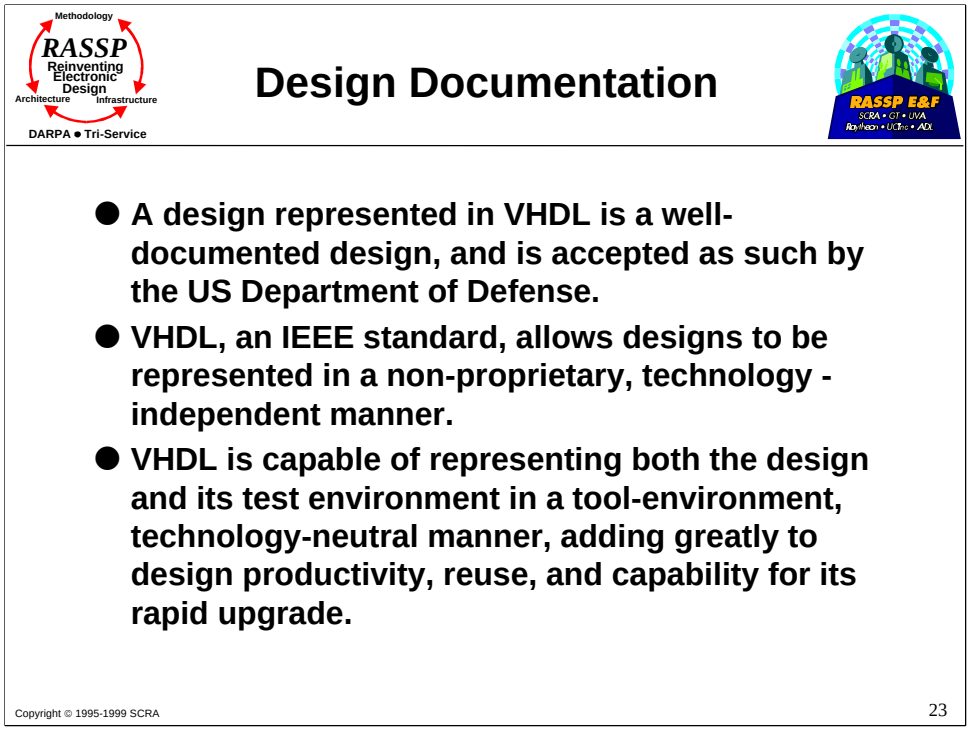
Design Capture, as mentioned earlier, is the most common use of VHDL in the synthesis process.



The terms, behavioral synthesis and RTL synthesis will be explained in the following slides, and refer to a design captured in VHDL at a higher level of abstraction (compared to the gate-level) where the synthesis tools offers some additional assistance in generating a optimized design.

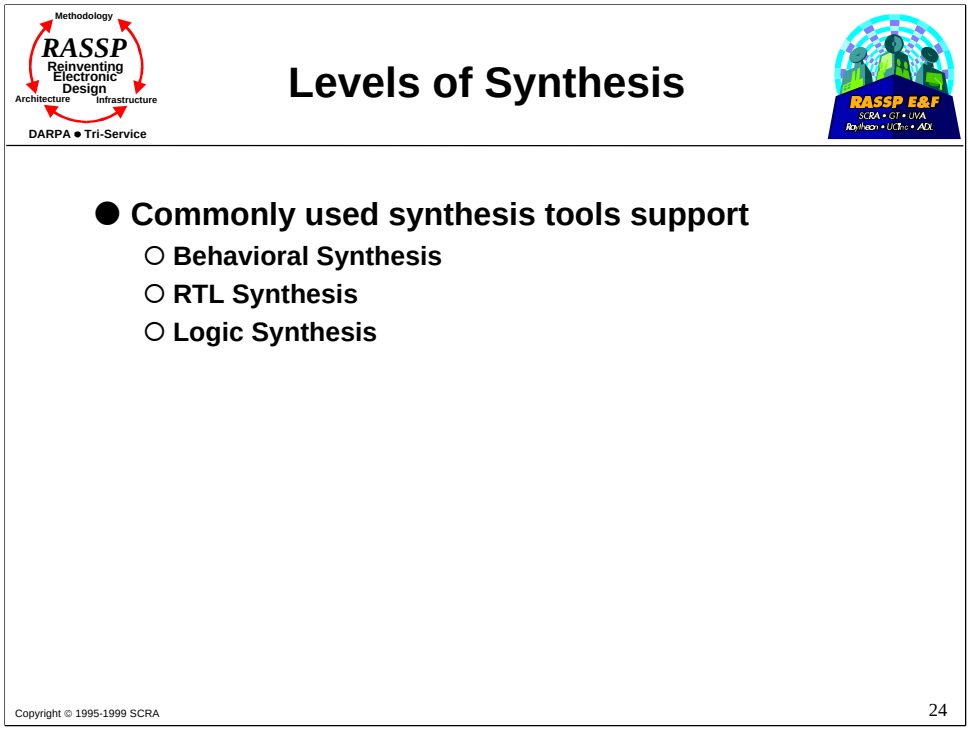


VHDL plays a useful role in providing a verification environment within the design process. Most of the synthesis process is in verifying constraints, functionality, timing, and form, fit and function capabilities of the resulting design.

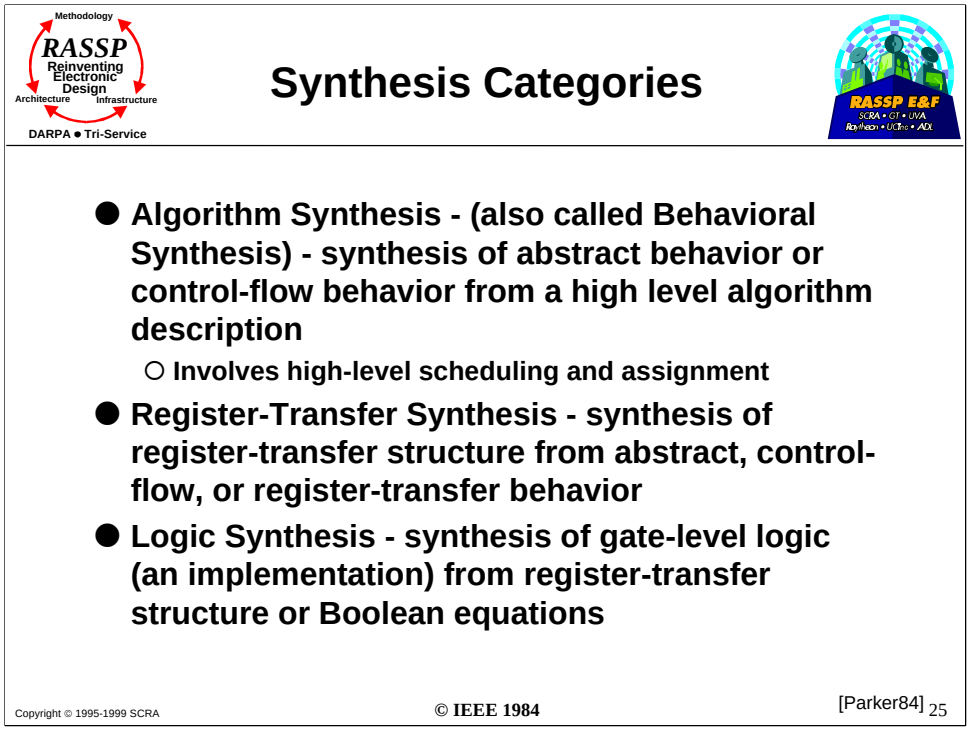


The IEEE Standard 1076-1993 provides the description of the VHDL language.

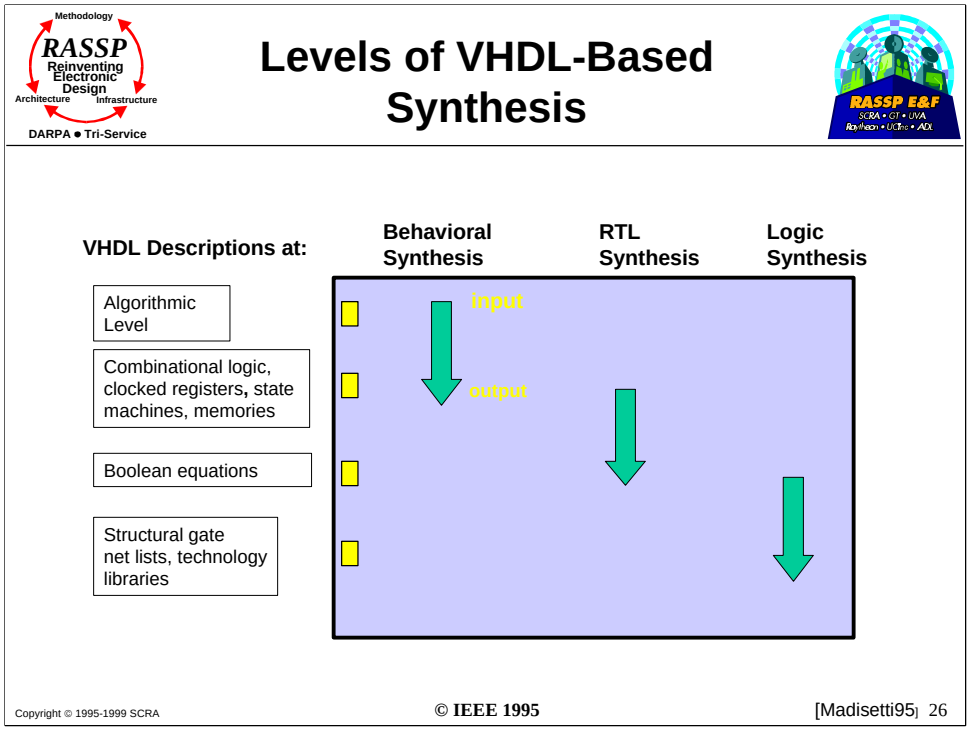
Current practice designs are document using methods that are both technology dependent and tool dependent. For instance, using ABEL one can synthesize a digital design and target it towards a family of FPGAs, but the design documentation (which includes the digital system, its test benches, associated packages) is not portable to other environments. VHDL provides an easy and effective mechanism to document the design at various levels of abstraction in a technology and tool independent manner.



We will now distinguish between each of these levels/categories of synthesis environments that support VHDL.

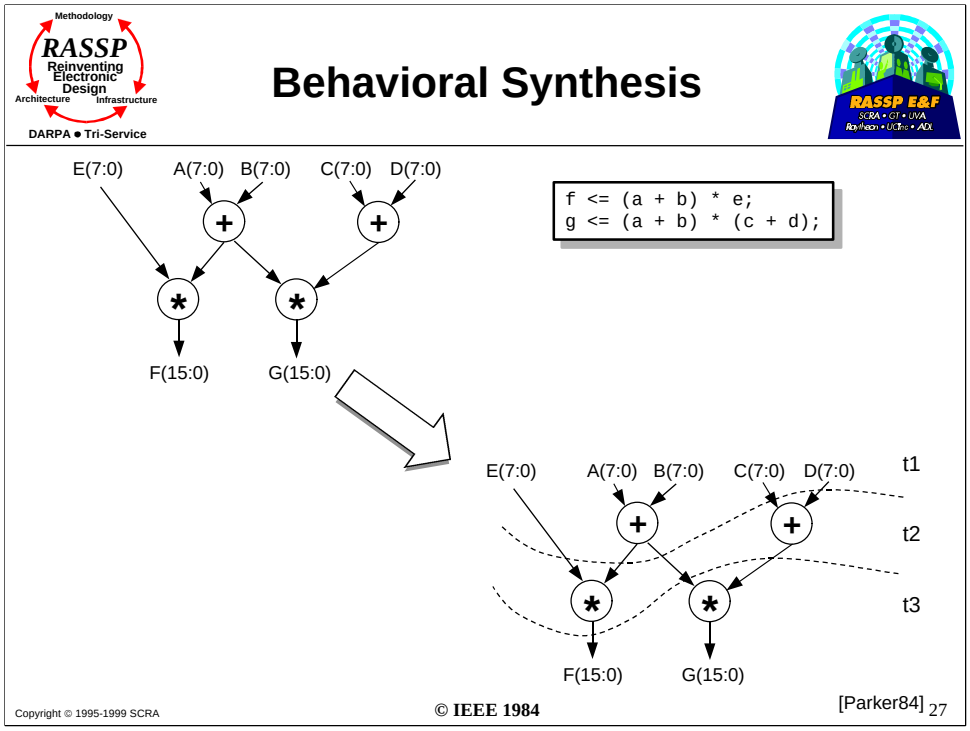


Behavioral synthesis is a relatively new area, where only a small subset of VHDL is supported for synthesis. Register Transfer (Level) Synthesis, also called RTL synthesis, is more common in commercial avenues and will be the main focus of this module.

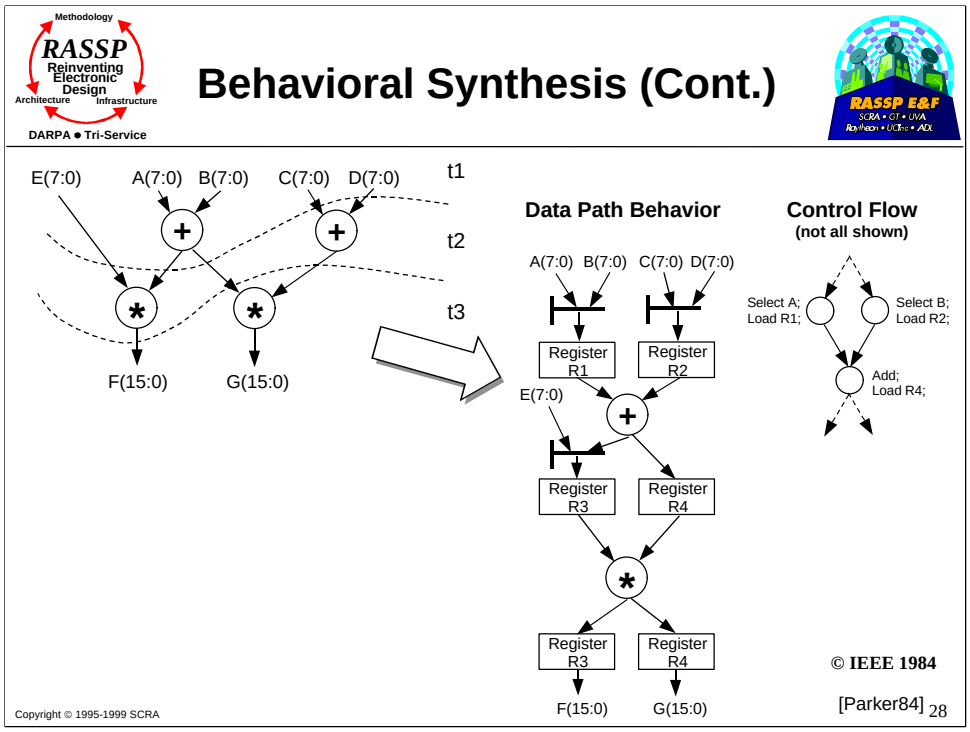


The arrows indicate the starting and end points for each type of synthesis. The tools to the left of the figure would need support from tools towards to the right to ensure continuity of the synthesis to the gate/transistor level implementation.

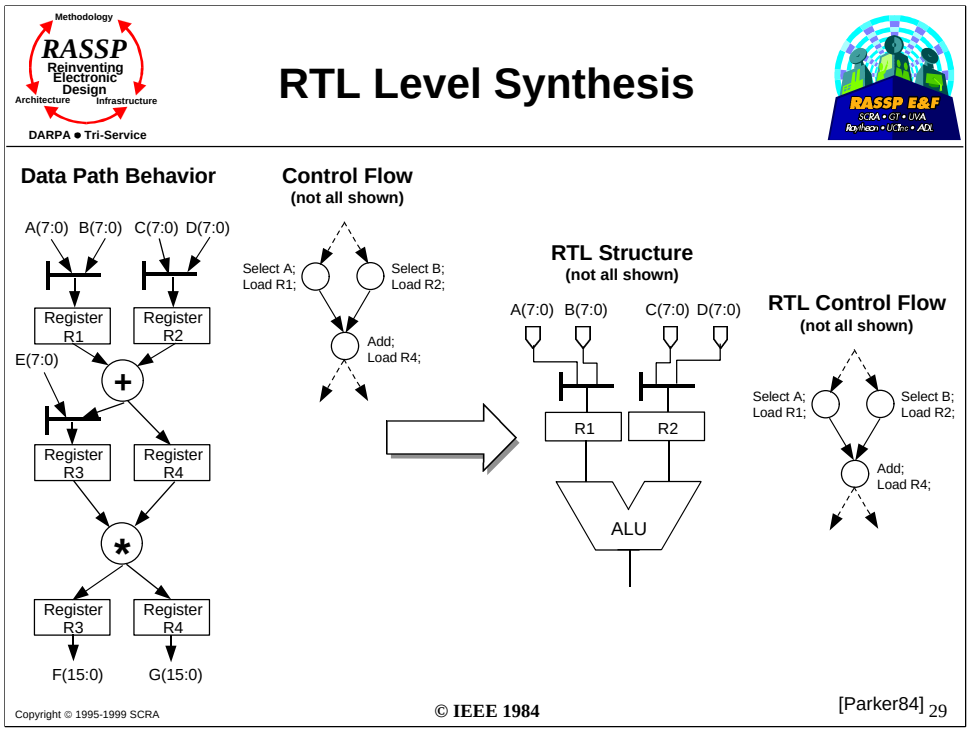
System-level synthesis is an advanced step that includes both hardware and software codesign and synthesis, and is covered in other RASSP modules.



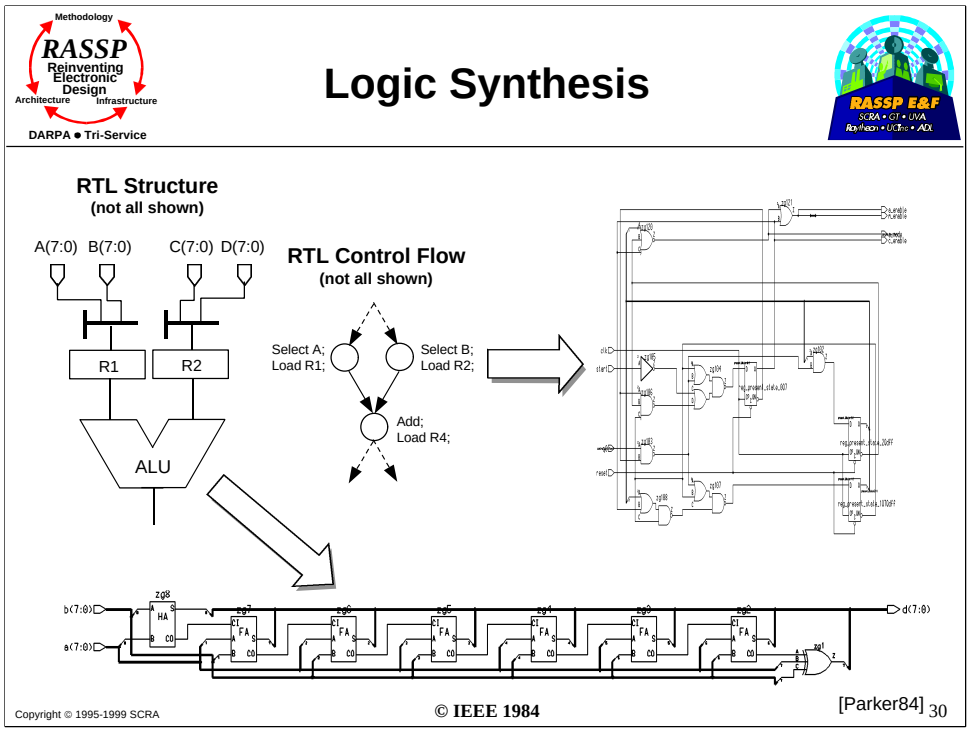
In the graph on the left, there is no notion of time, and the computation is assumed to flow in a data-flow type manner. In the figure on the right, a notion of time is introduced, wherein the operations scheduled in time slot t1 are shown distinct from those scheduled in t2 and t3.



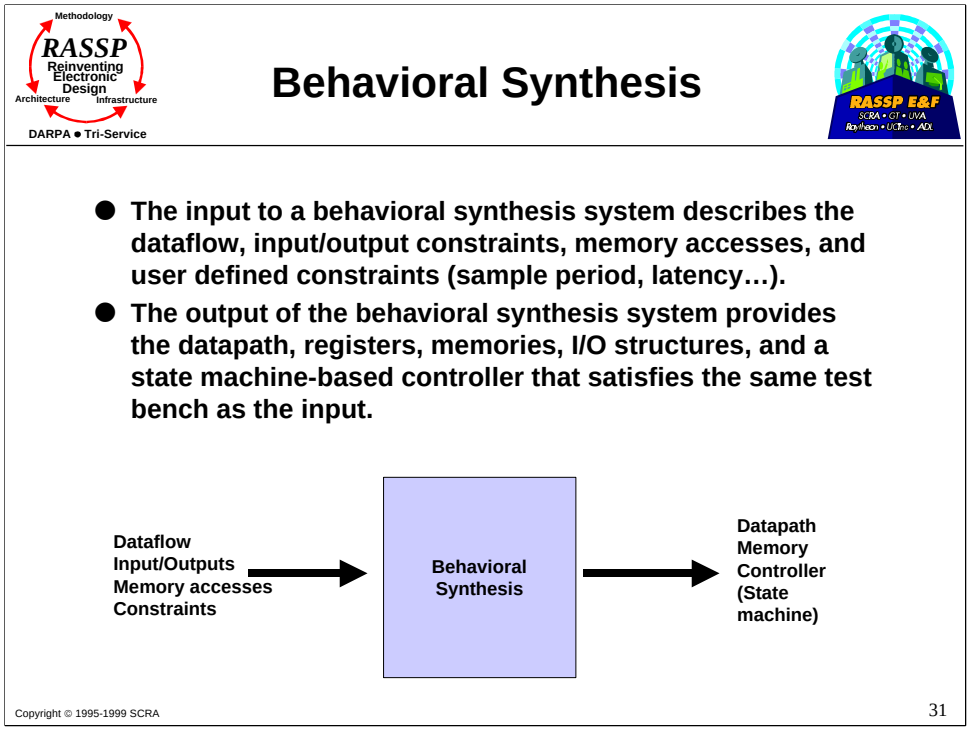
The timing annotated flow graph is now mapped to an RTL structure with an associated data path (the computational portion) and a control flow graph (the controller) that is shown on the right. The exact structure of the computation is not important at this point in the presentation, but it is clear that lower level implementation details have been inserted into the design representation.



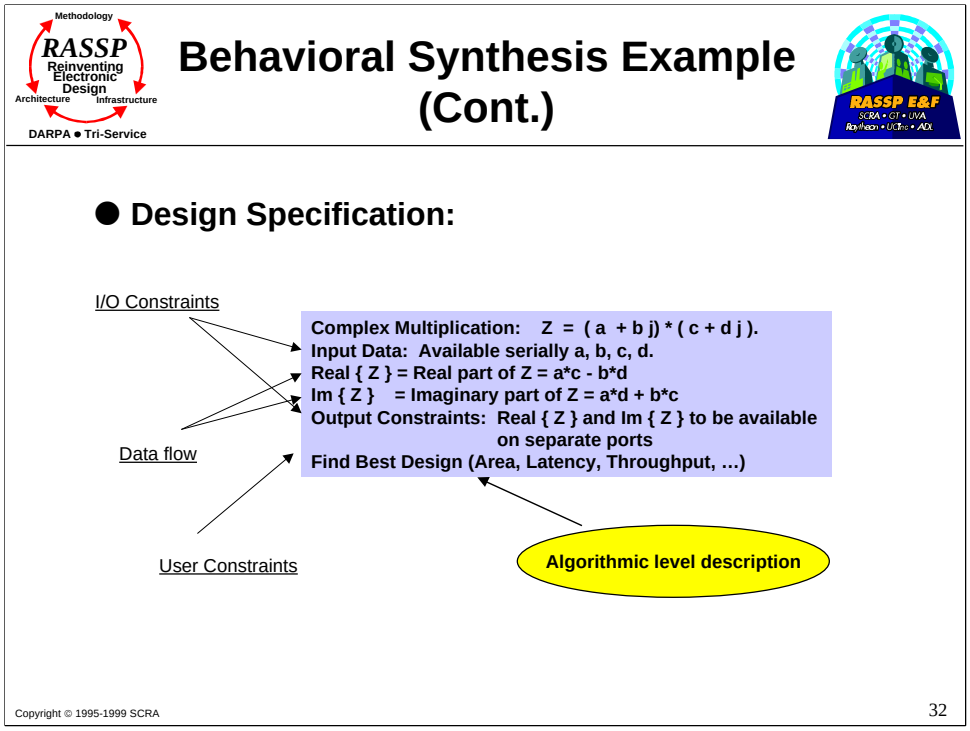
At the RTL level the control/data flow graph is mapped to a computational structure (that consists of one ALU as shown) together with two registers and multiplexors plus some control flow determining the sequence of operations (the scheduling).



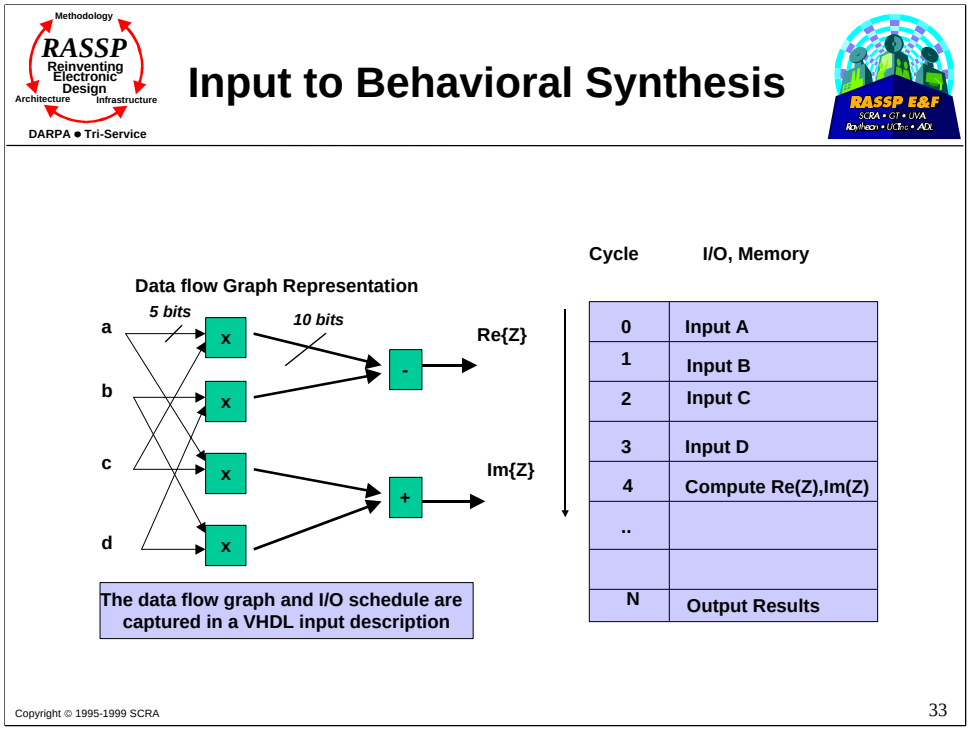
The RTL description is then synthesized to a gate level description shown above to unambiguously describes one possible implementation of the original behavioral algorithm specified as a flow graph.



We now describe the term behavioral synthesis in some detail. We have used the Behavioral Compiler ™ from Synopsys as a reference tool for this description in terms of the scope of behavioral synthesis.

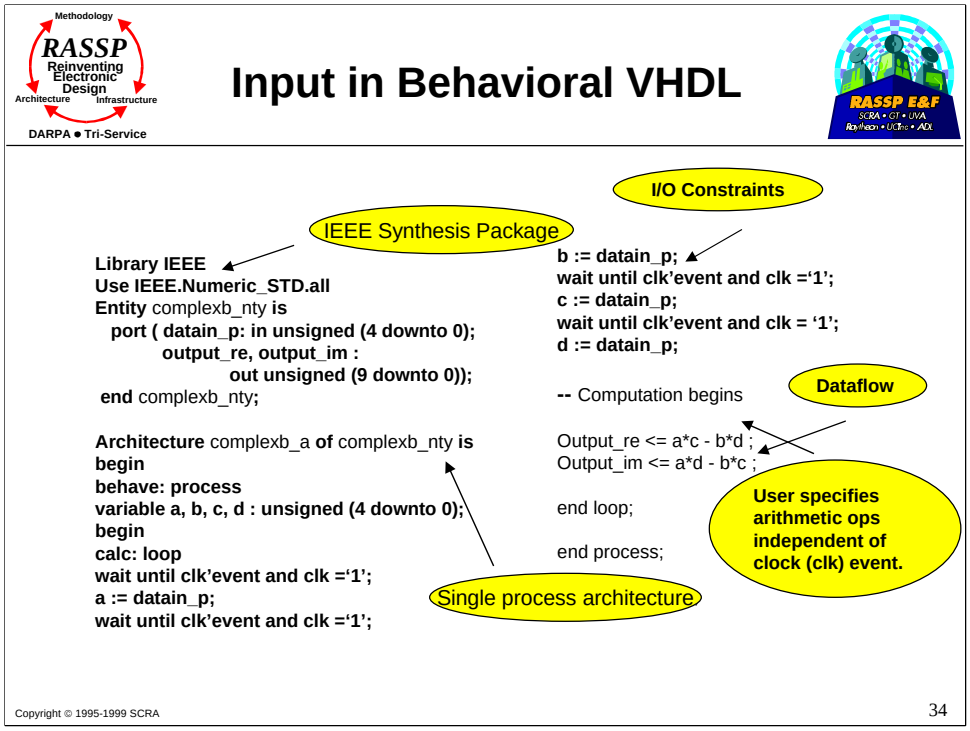


We use a language-based (VHDL) description of the input specification. Note that these tools do not support the entire subset of the language. This is a pseudo-code description of the input specification.

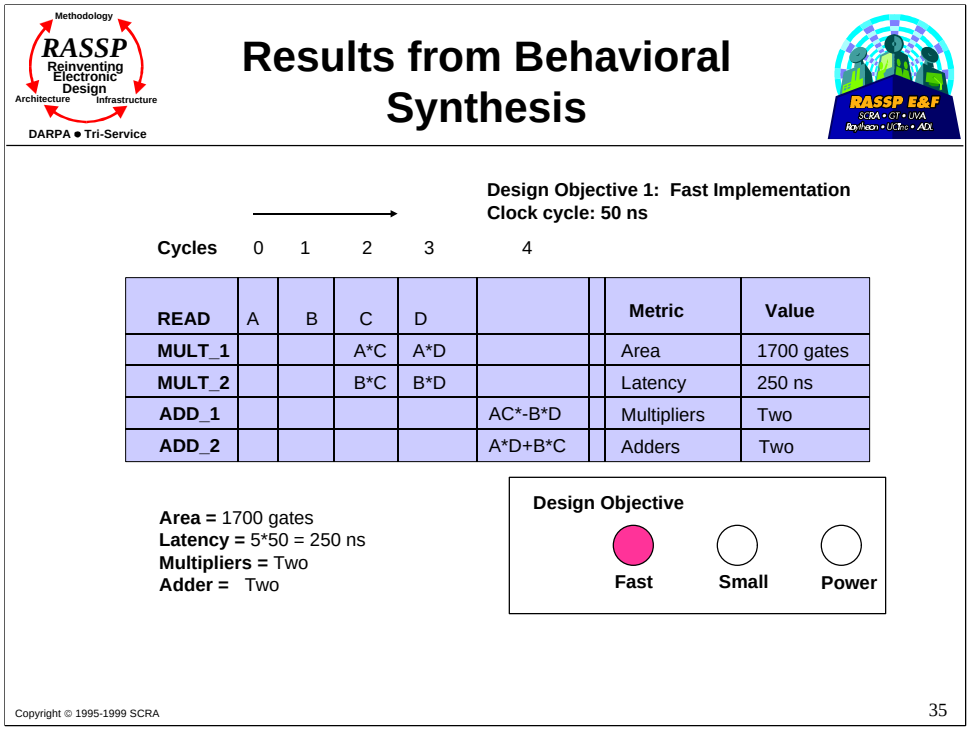


We also have to capture the constraints on the input and the output timing and input it together with the algorithmic description of the previous slide. For simplicity, we assume the input is fed in serially, while the output results occur in parallel.

A dataflow graph is one way of representing the behavior described in VHDL. Neither Behavioral Synthesis or VHDL are limited to what can be expressed in a "simple" dataflow graph, and often an expression of control flow is necessary as well.

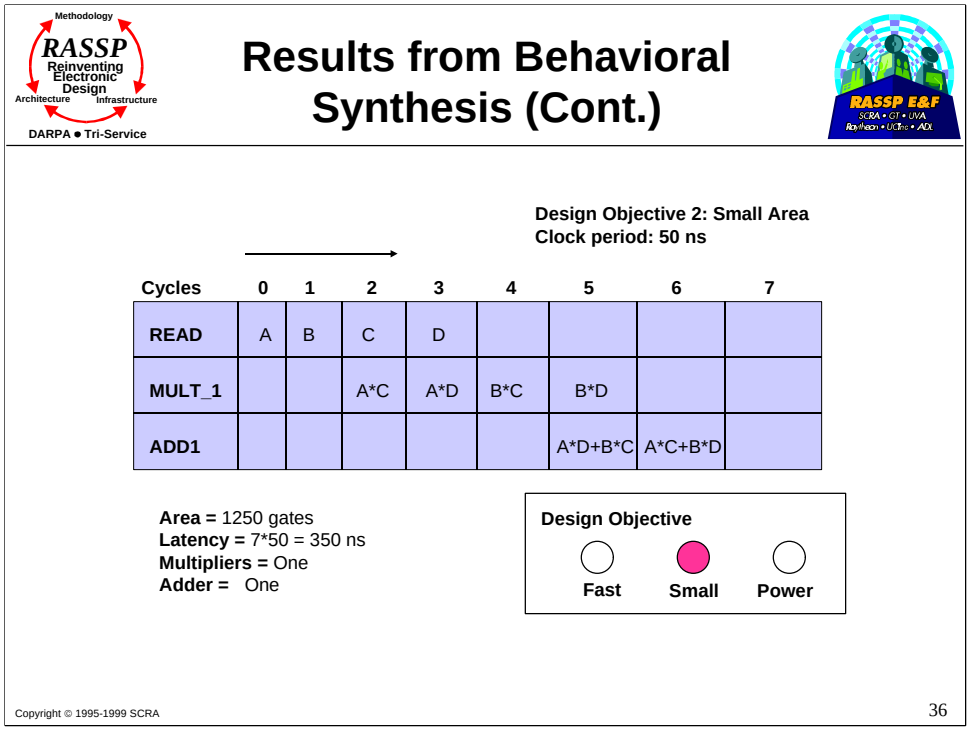


The "English/pictorial" specification of the previous slide is captured in an executable form in the specification shown above that can be input to (read by) a behavioral synthesis tool.



The user has the option of choosing the design objective --- fast design, small area design, or a low power design. When the "fast" design option is chosen, as in this slide, two multipliers and two adders are allocated by the synthesis tools and the algorithm is mapped to this structure resulting in the schedule shown in the figure.

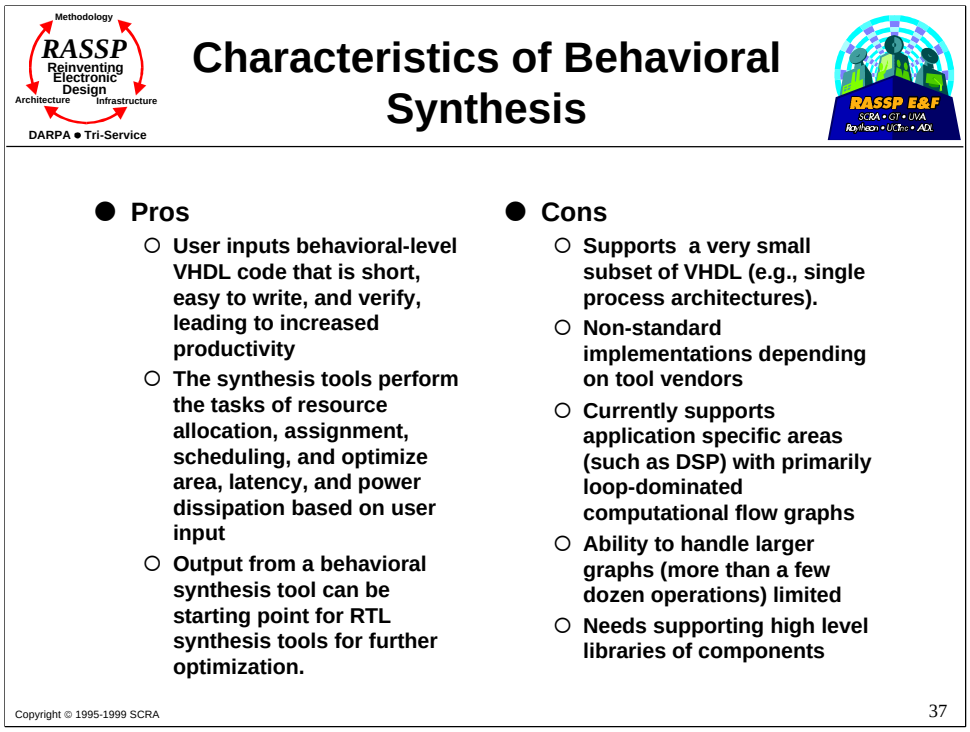
While the output is in gates format (as actually produced by the Synopsys Behavioral Compiler) the metrics are represented on the right corner. The interconnect area is not considered.



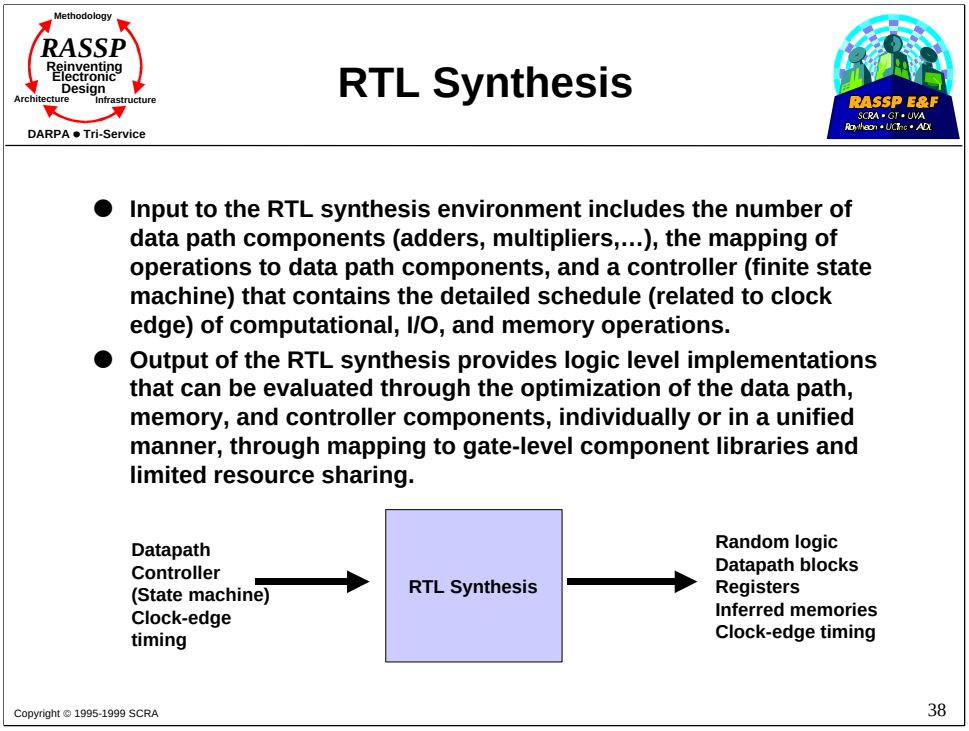
When the same design is input to the behavioral synthesis tool and “small” option is chosen, the synthesis tool attempts to minimize the area by choosing only one multiplier and adder. This results in a small design, albeit a slower one. This exploration is performed by the tool and not by the user, though the user makes a choice of the constraints

These are actual figures obtained from Synopsys Behavioral Compiler tutorial documentation.

[Synopsys97]



We have listed our view of the pros and cons of using a behavioral synthesis tools. Clearly they have the advantage of reducing the burden on the designer when dealing with a domain specific design that has a sufficient number of constraints to provide a limit on the number of implementation choices.



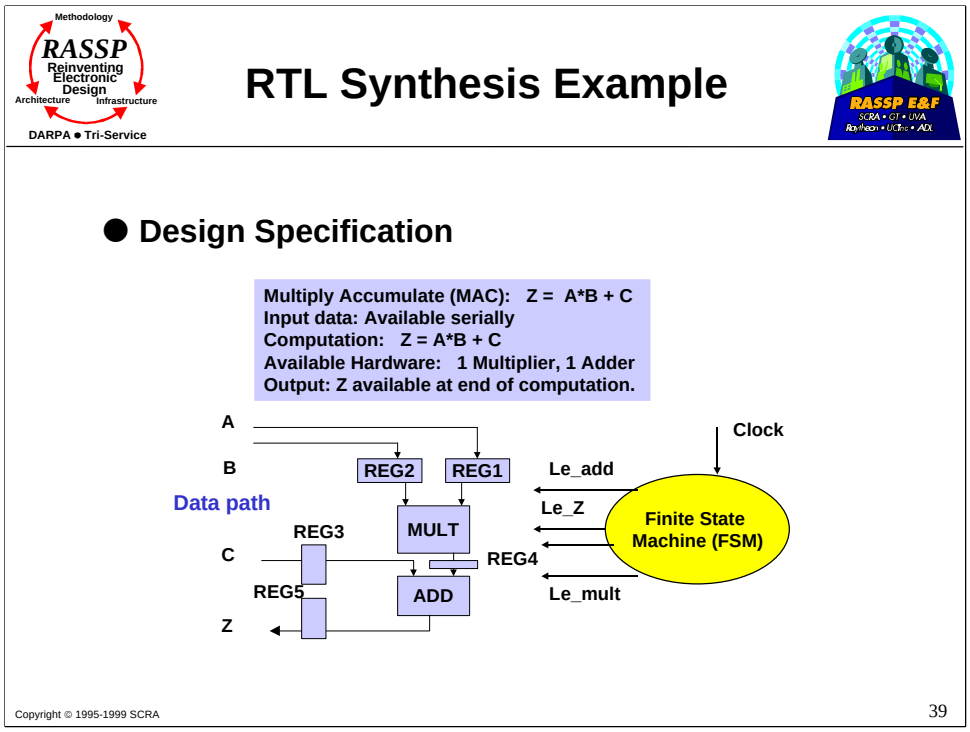
RTL synthesis requires an input description that is more detailed with respect to scheduling and clock-edge related behavior. The number of data path components and the scheduling is performed prior to input to the synthesis tool. This highly constrained design is then converted to lower levels by the synthesis tools.

RTL synthesis tools, typically, operate more like tools that permit capture of the design specification at the RTL level, as opposed to performing extensive optimizations in the quality of the architecture, latency, or area.

Their primary merit is that they raise the design entry abstraction level from the logic/gate-level to the RTL arena.

RTL synthesizers allow the user to benefit from mapping RTL level components directly to optimized RTL components from vendor libraries resulting in faster synthesis.

Most RTL synthesizers allow push-button optimizations for finite state machines (FSM) if they follow one of the allowed templates.



We now describe in "English/pictorial form" the design input specification required by an RTL synthesis tool. Note the requirement for additional detail beyond what is needed for behavioral synthesis in terms of the number of data path units, and the initial scheduling that must be specified up front.

What's in the blue box above would not be input into an RTL synthesis tool, rather the states are defined and what occurs at those states is defined so an RTL description is what is input e.g.,

C1: load A->Reg1, b->Reg2

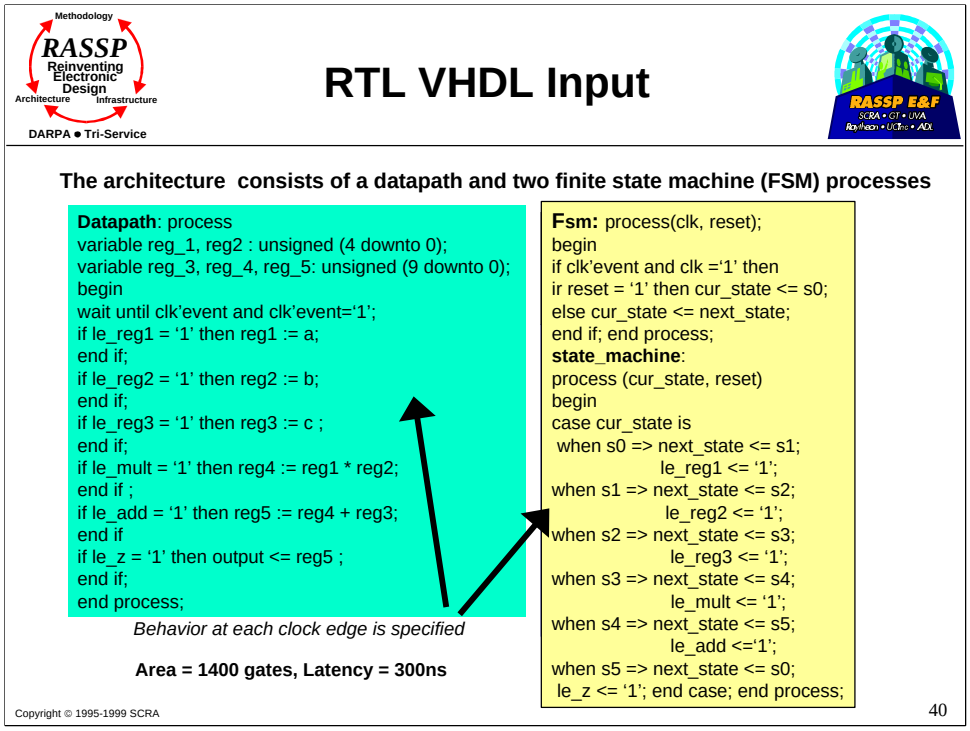
C2: mult(Reg1, Reg2)

C3: store mult_out->Reg4

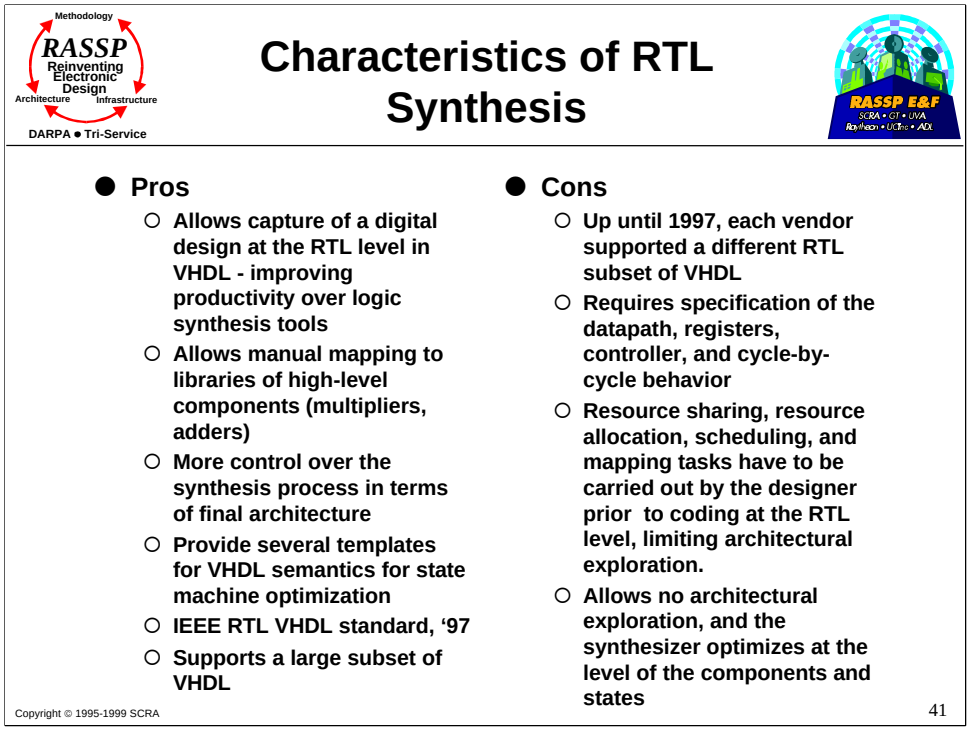
c3: load C->Reg3

C4: add(reg4, reg3)

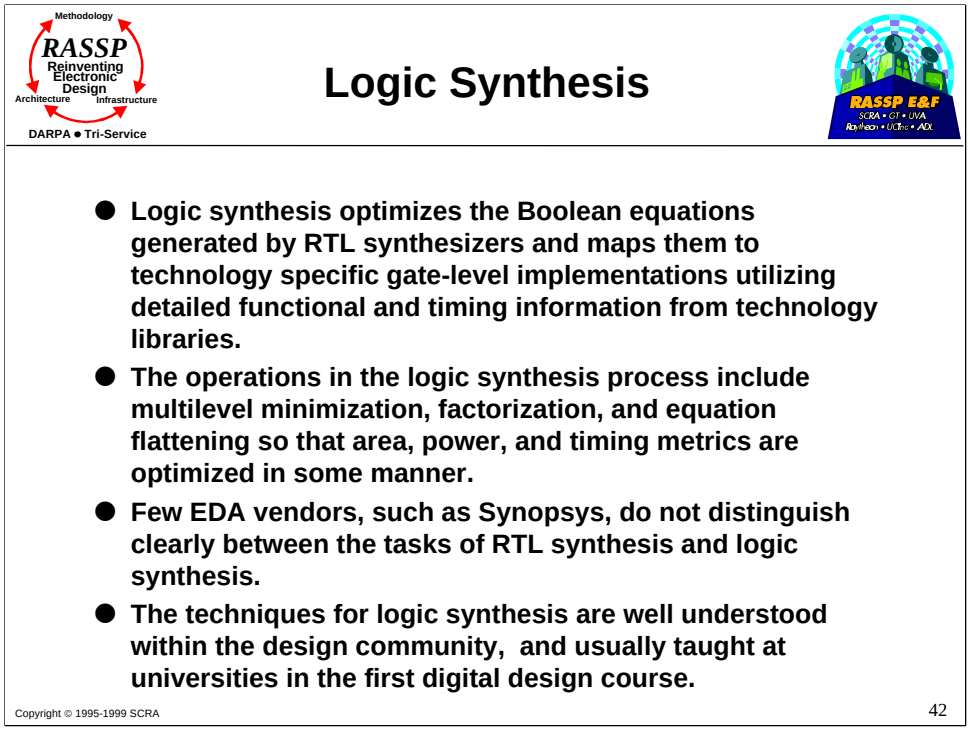
c5: store add_out-reg5



This is a typical input to a RTL synthesis tool (such as Mentor's Autologic or Synopsys' Design Compiler) showing both the control flow and the detailed clock-related timing (data flow).

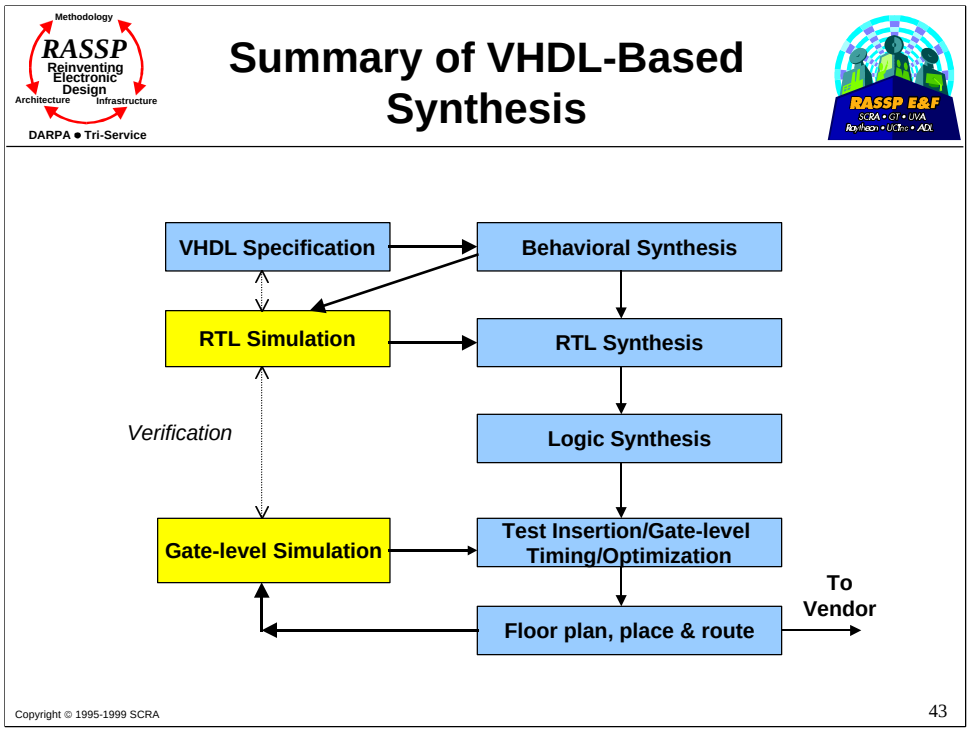


These are viewpoints of this module developer (Madisetti) and do not reflect the views of DARPA or the RASSP program.



Several undergraduate texts cover the area of logic synthesis in some detail.

See for instance: [Brayton84]



We have briefly shown the primary plusses and minuses of the three main types of synthesis tools - behavioral, RTL and logic synthesis. The above flow chart describes one way in which these different levels of abstraction can be used within a top down design process. One of the benefits of VHDL is its executable nature that permits validation and verification of the design at one or more levels of abstraction. Note that the VHDL Specification can be fed into one of the upper three levels on the right (Behavioral, RTL or Logic) depending on the tool being used.

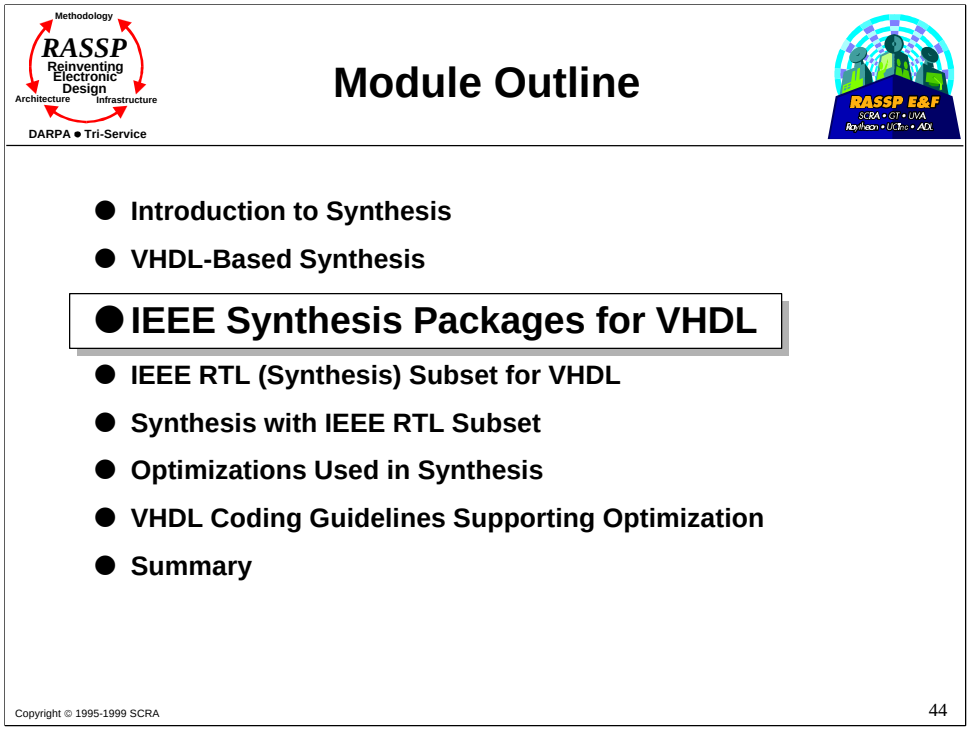
VHDL supports synthesis in many ways - design entry, specification, documentation, simulation, and verification

Synthesis tools do not support all features described in VHDL LRM

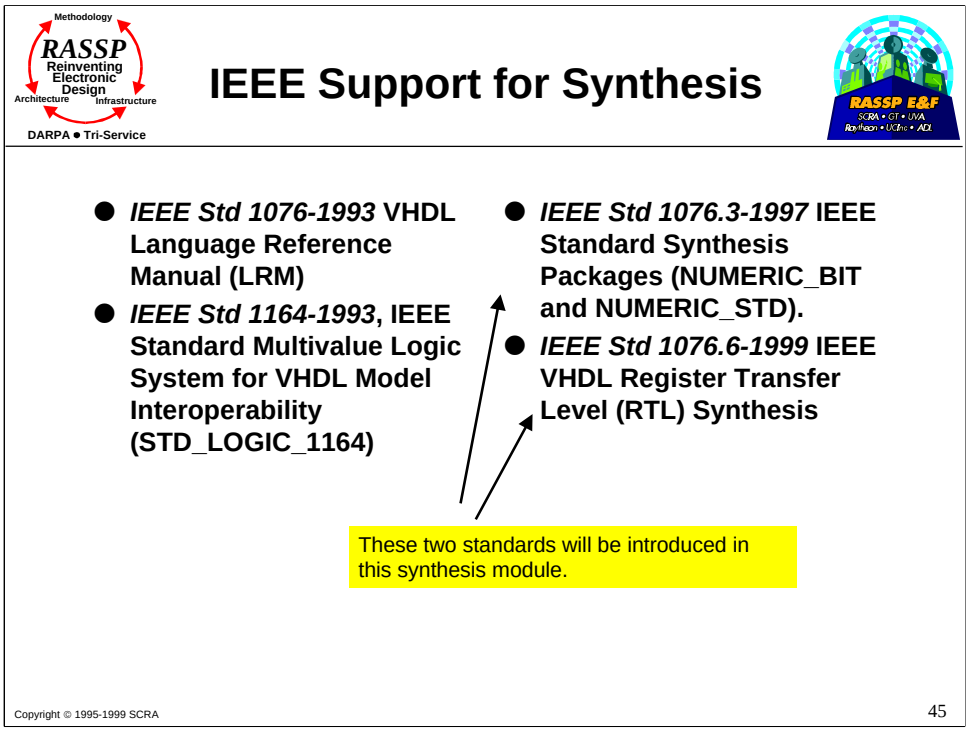
VHDL provides an environment for both design and its test, unlike proprietary languages for synthesis .

Synthesis at higher levels of abstraction, e.g., behavioral synthesis, improves productivity for smaller (few thousand gates) application-specific domains (e.g., DSP). Non-standard support for behavioral synthesis by EDA vendors and less control over process is a problem

RTL synthesis provides a good compromise between raising the level of abstraction and also providing control over the synthesis process and the optimization of the result. In addition, RTL synthesis is now supported by IEEE standardization efforts.

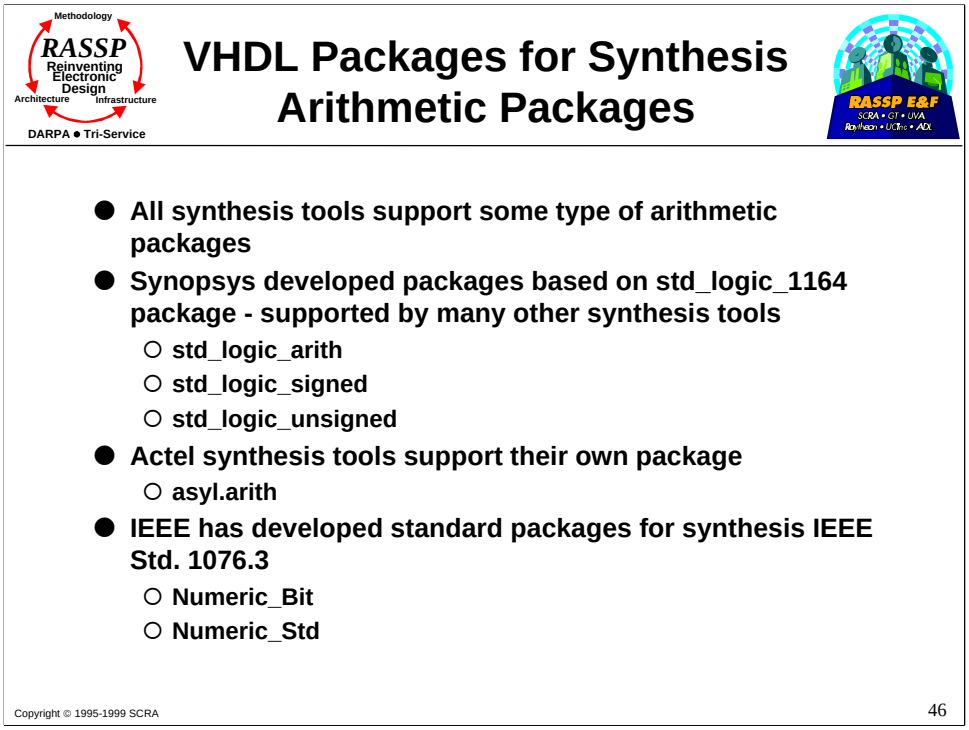


We will now describe VHDL Packages standardized by IEEE standards community. In the past, each vendor used (and supported) a different set of packages for synthesis, leading to variety of migration problems.



Modules 10 through 13 introduce IEEE Std 1164-1993 and IEEE Std 1076-1993 which is the VHDL LRM.

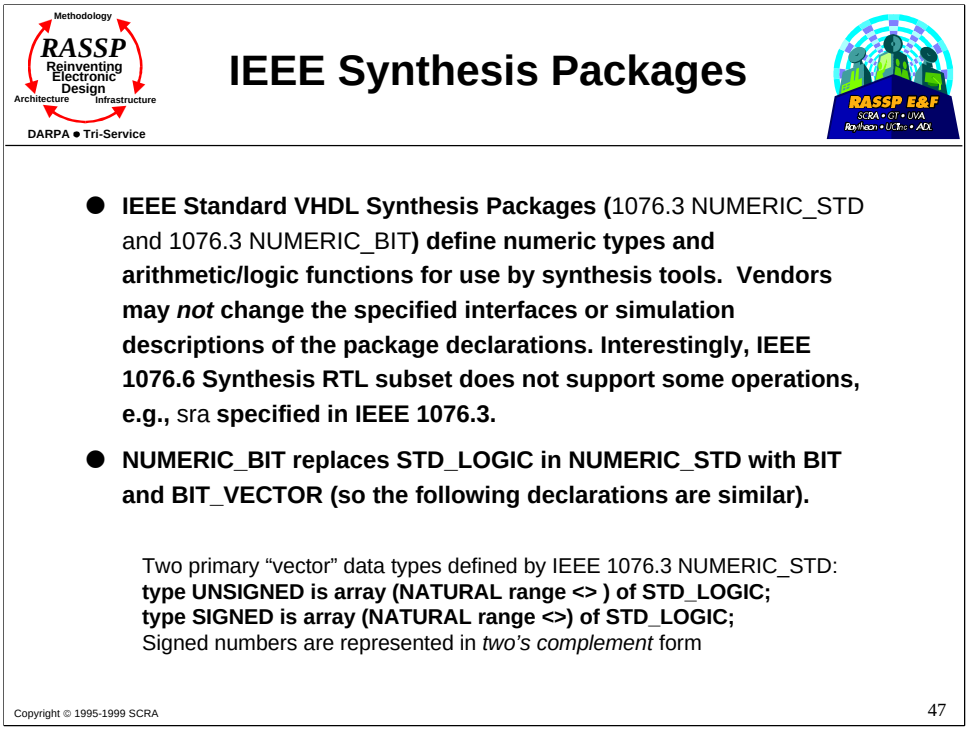
In the remainder of this module, we'll cover the two developments shown on the right - IEEE Std 1076.3-1997, a set of synthesis packages, and the other, IEEE Std 1076.6-1999, a standard for standardization of the synthesis subset.



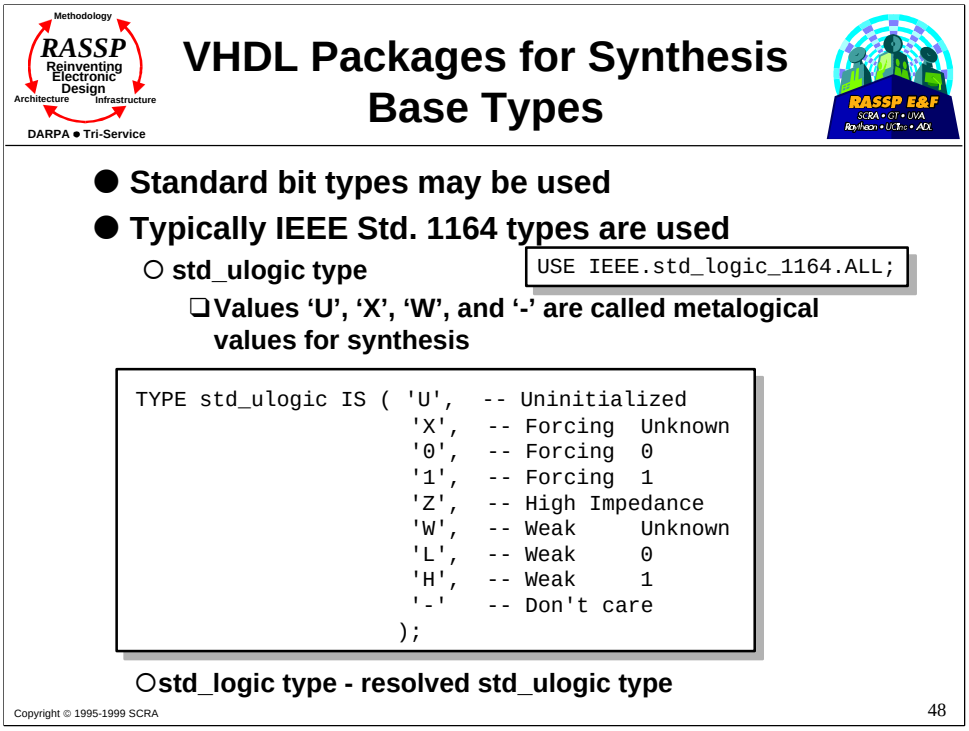
All synthesis tools support some type of arithmetic package - that is, a package that contains operators like addition, subtraction, multiplication, etc.

When synthesis tools began to appear, each tool vendor created their own arithmetic package. Synopsis created packages for general arithmetic, and signed and unsigned arithmetic called std_logic_arith, std_logic_unsigned, and std_logic_signed. These became “de facto” standards and other tool vendors began to support them. In 1997, the IEEE came out with a set of standard packages for synthesis. These are defined in IEEE std. 1076.3-1997 IEEE Standard VHDL Synthesis Packages.

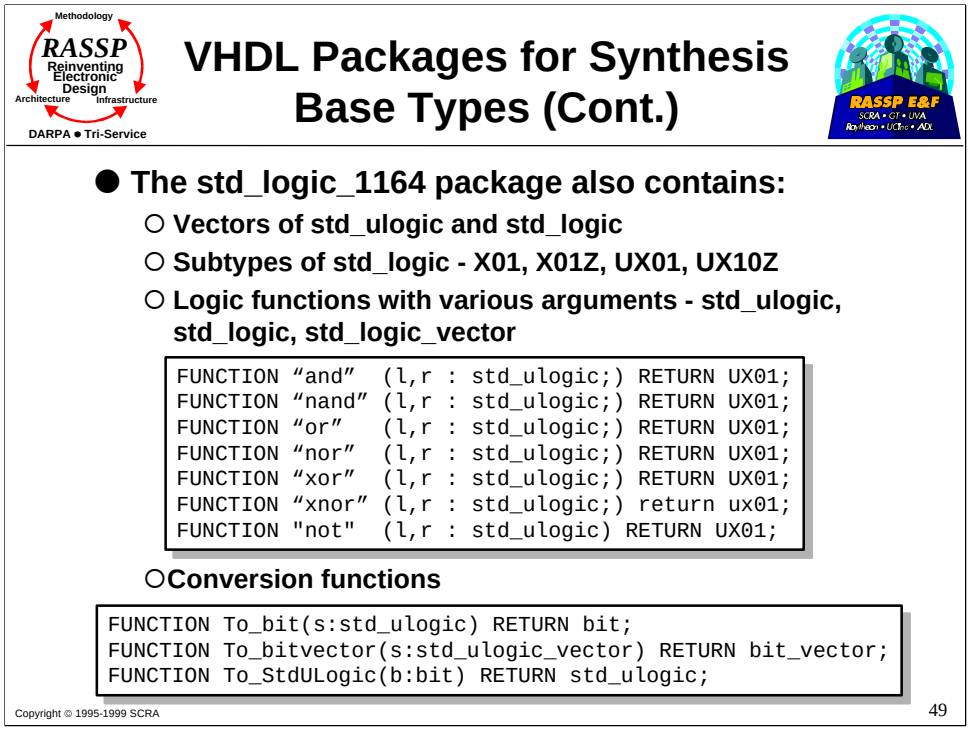
These packages are called numeric_bit, for arithmetic operations on standard VHDL BIT types, and numeric_std for arithmetic operations on std_logic types.



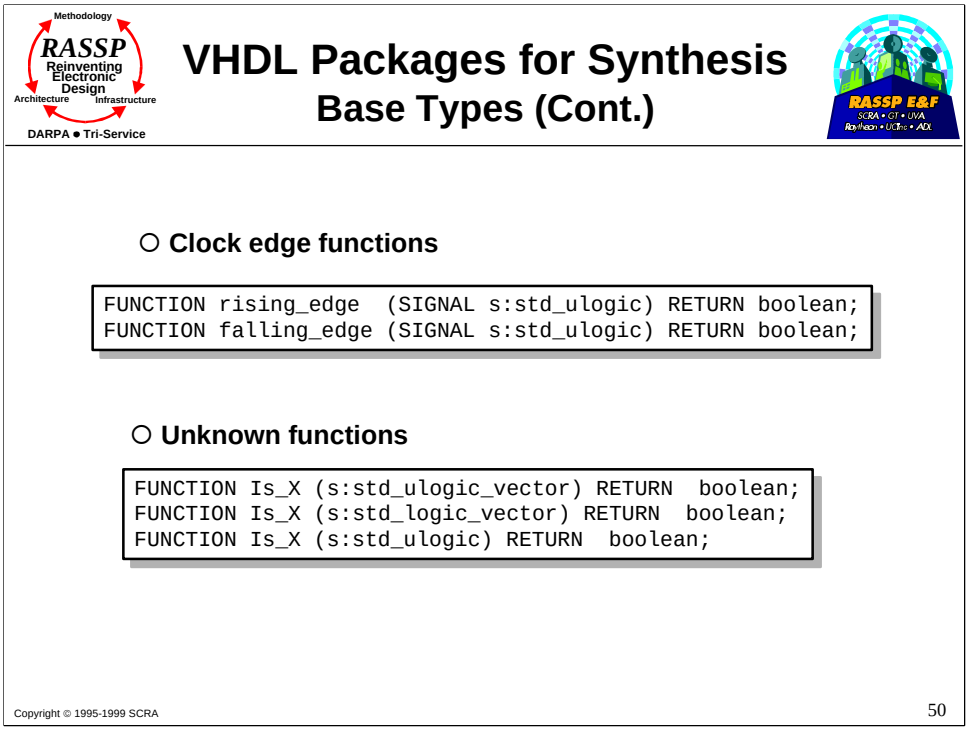
We now list the package (and not the package body) in the slides that ensure. One motivation for this module is to teach the reader how to write synthesizable VHDL, and the availability of the functions in the package is necessary (admittedly, dry reading). The beginner reader may skip the next few slides.



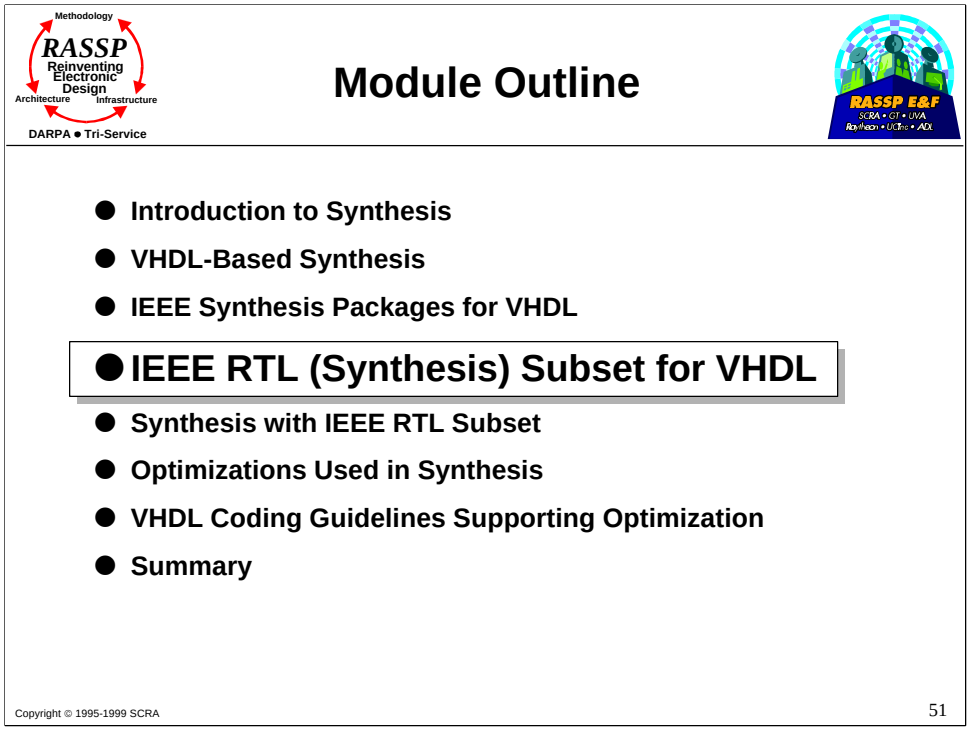
Not all types are supported in synthesis, and thus the Synthesis Packages ensure that designers use data types that can be synthesized. For instance, floating point support has not been standardized.



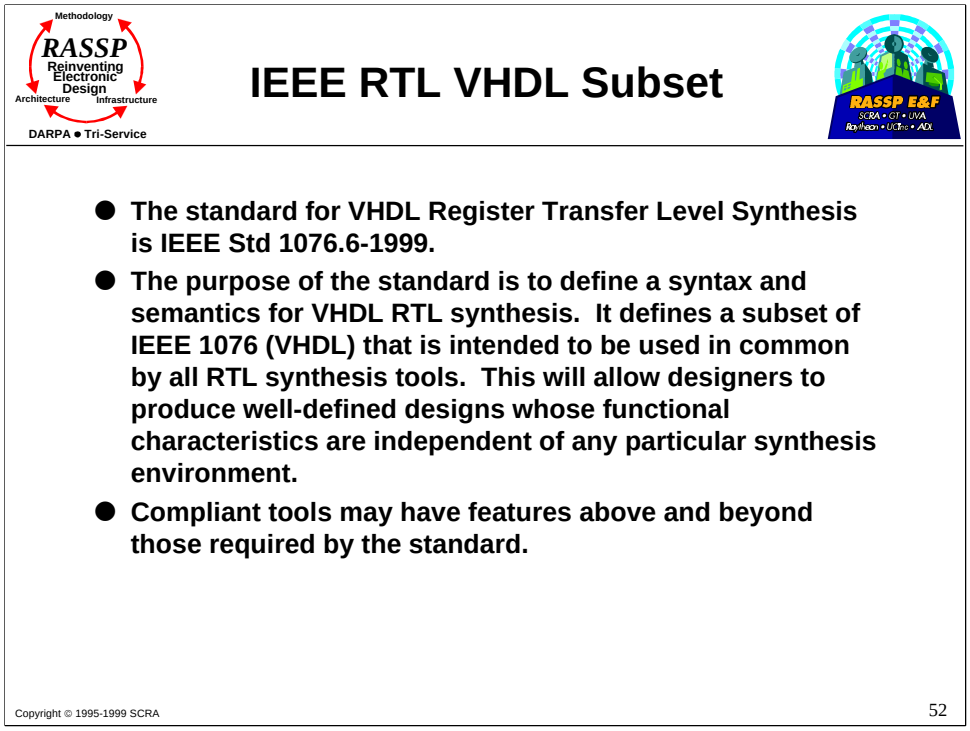
We continue with a description of supported types.



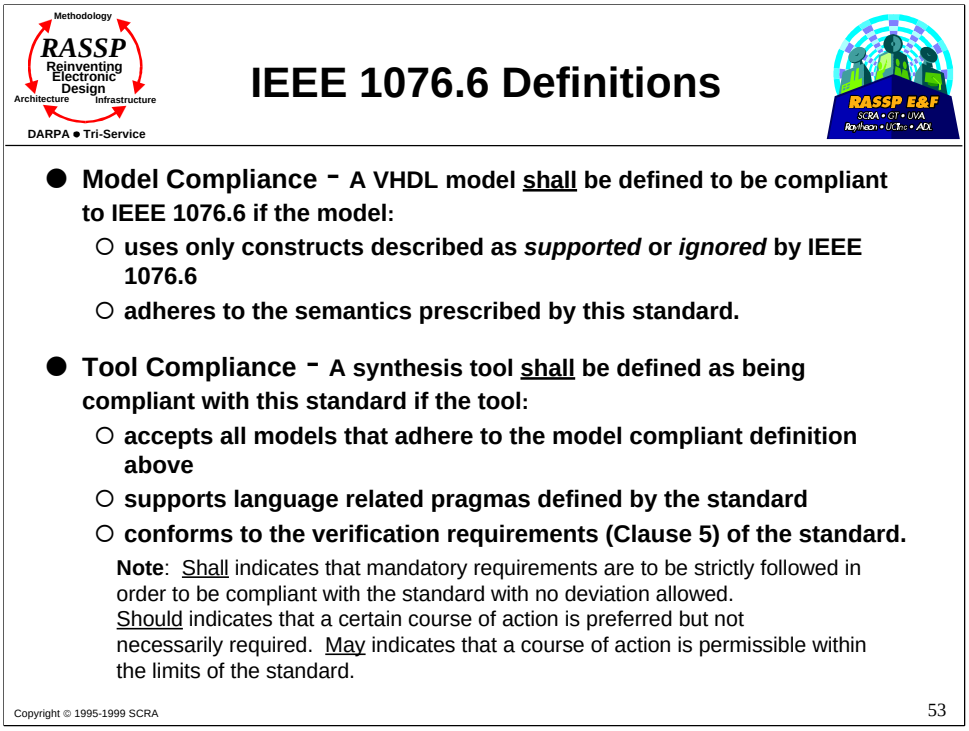
A number of vendors used to support a variety of functions (defined within packages) that were related to clock-edge behavior. The IEEE VHDL package has defined the functions “rising_edge” and “falling_edge” that can now ensure that RTL code is portable from one synthesis environment to other.



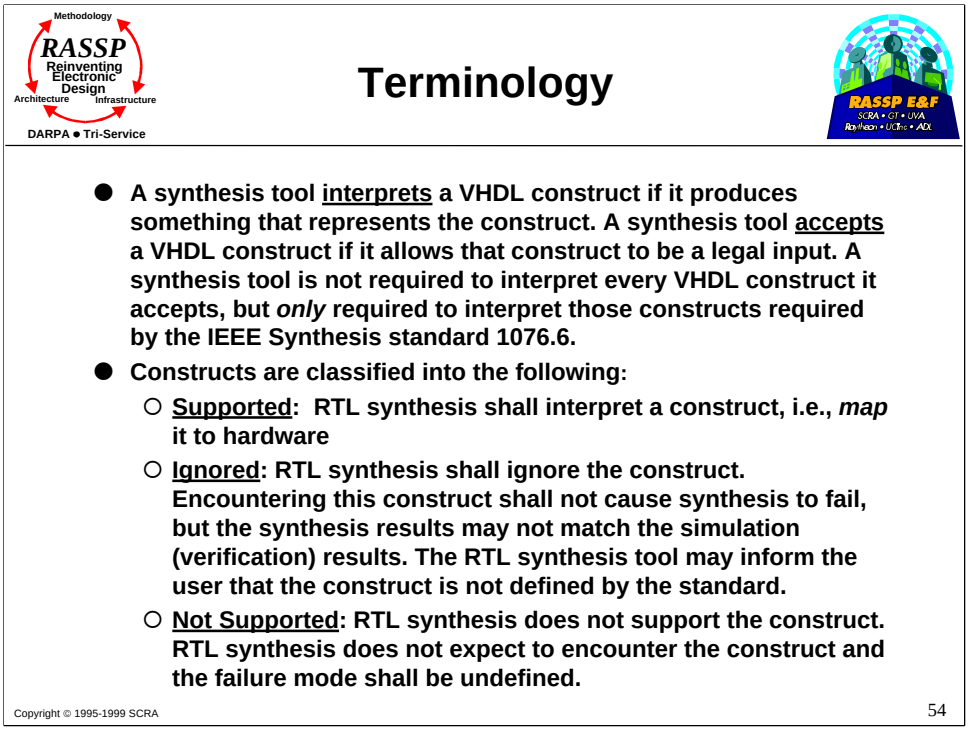
We have introduced the synthesis process, various types of synthesis, support for synthesis using VHDL, and IEEE packages for synthesis. We are now ready to understand the IEEE RTL Synthesis, IEEE 1076.6-1999, subset that became an IEEE standard in 1999. We will describe the syntax and the semantics of the IEEE VHDL RTL Synthesis Subset. We again assume that the reader is familiar with VHDL.



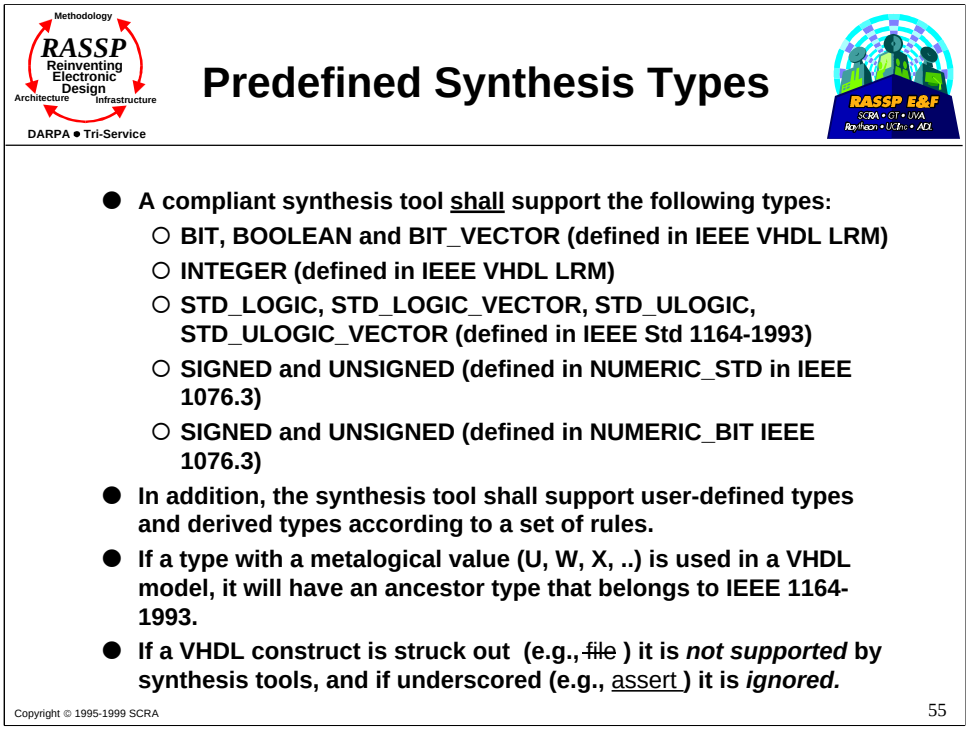
The slide describes the history of the Synthesis Subset (http://www.vhdl.org/siwg/)



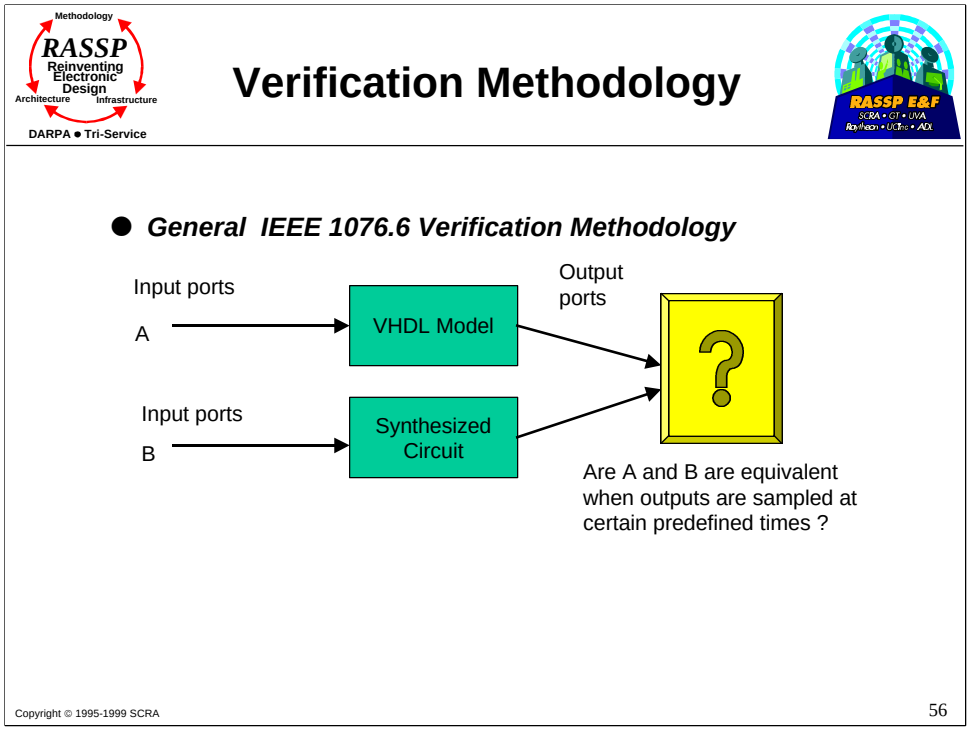
It is important to note the difference between “shall”, “should” and “may” at this point in the module. One should proceed further only after noting the importance of the distinction between these closely related terms and their impact on the tools being developed that support the RTL Synthesis standard.



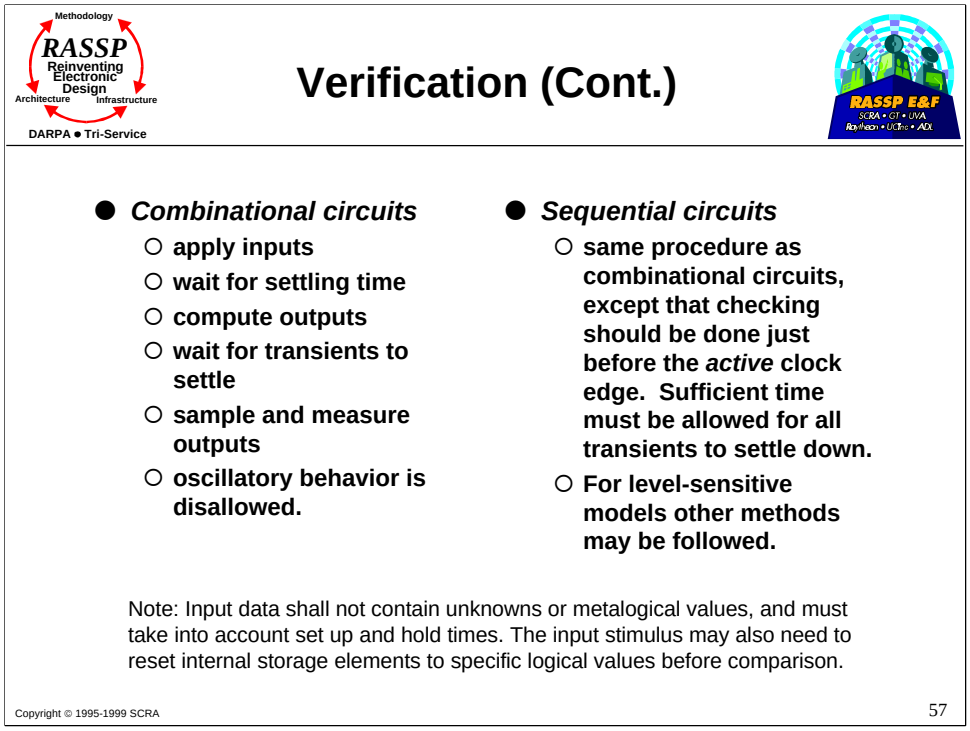
This slide describes the context within which the IEEE RTL Synthesis subset standard operates.



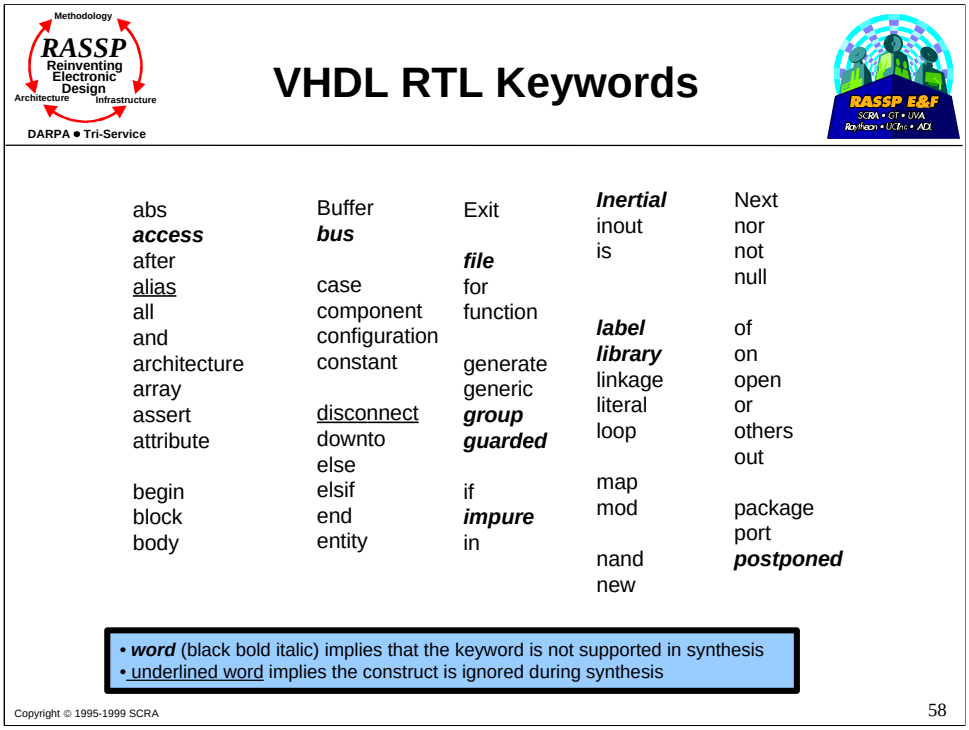
Information on the IEEE Synthesis Standard draft document can be obtained from http://vhdl.org/siwg.



The IEEE standard requires that the VHDL model and the resultant synthesized circuit be equivalent as described by the slide.



The “equivalence” relation of the previous slide is interpreted for combinational and sequential circuits differently, as noted above.

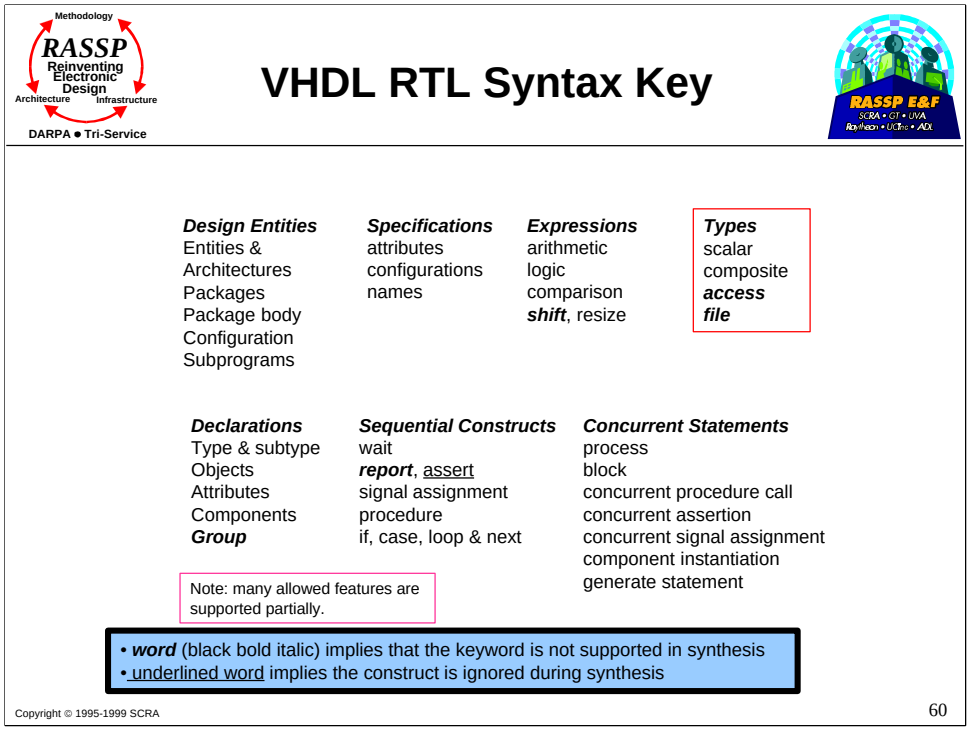


All the keywords used in VHDL are depicted in this slide and the following slide. The supported keywords for synthesis (1076.6) are show in normal font, while the ignored keywords are shown underlined. The keywords that are not supported are in ***bold italics***.

VHDL RTL Keywords (cont.)

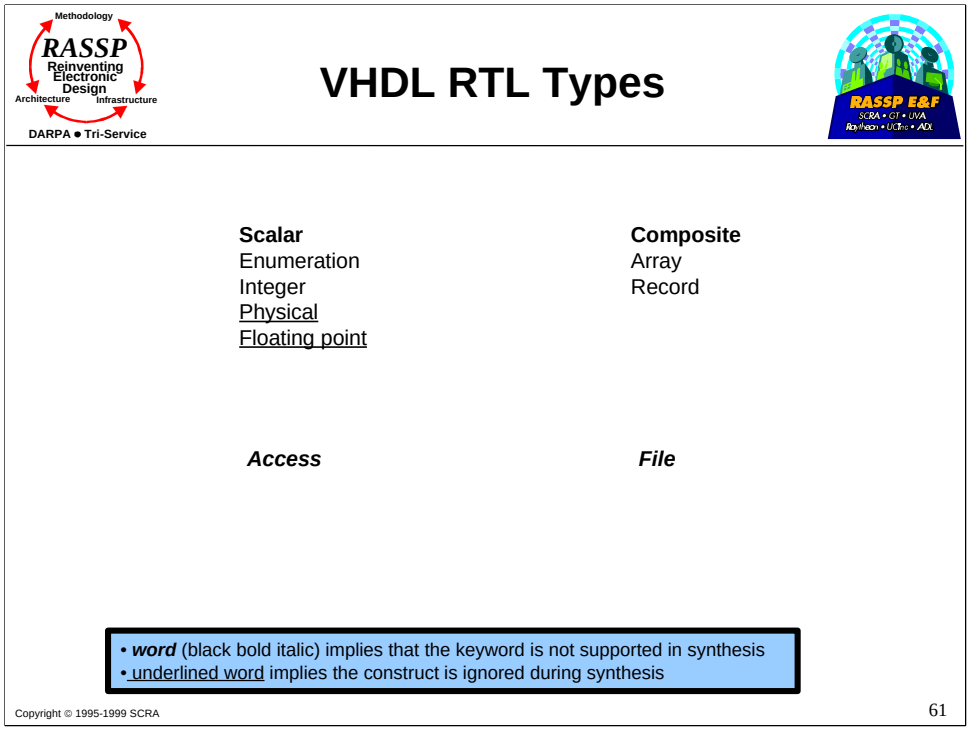
procedure	***shared***	use
process	signal	
pure	***sla***	variable
	sll	
range	***sra, srl***	wait
record		when
register	subtype	
reject		
rem	then	***while (in loop)***
report	to	***with***
return	transport	
rot	type	***xnor***
ror		xor
	unaffected	
select	units	
severity	until	

- ***word*** (black bold italic) implies that the keyword is not supported in synthesis
- underlined word implies the construct is ignored during synthesis

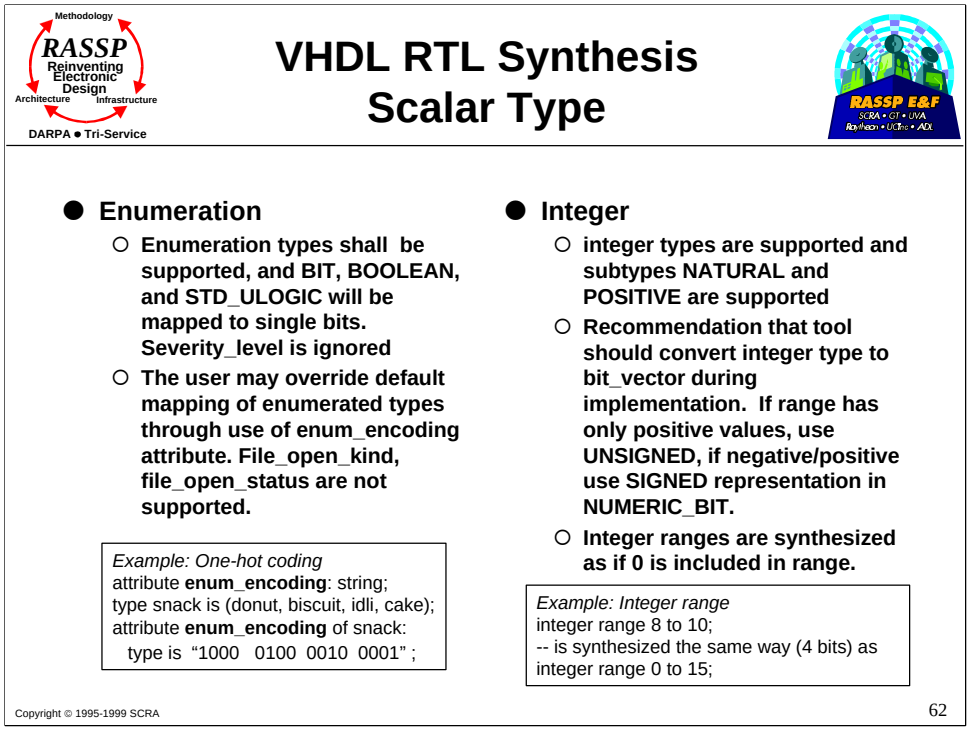


While many features of VHDL are supported, not all of them are supported fully. Thus it is important to rely on the syntax descriptions to see which features are supported, as trivial examples cannot convey these attributes in detail.

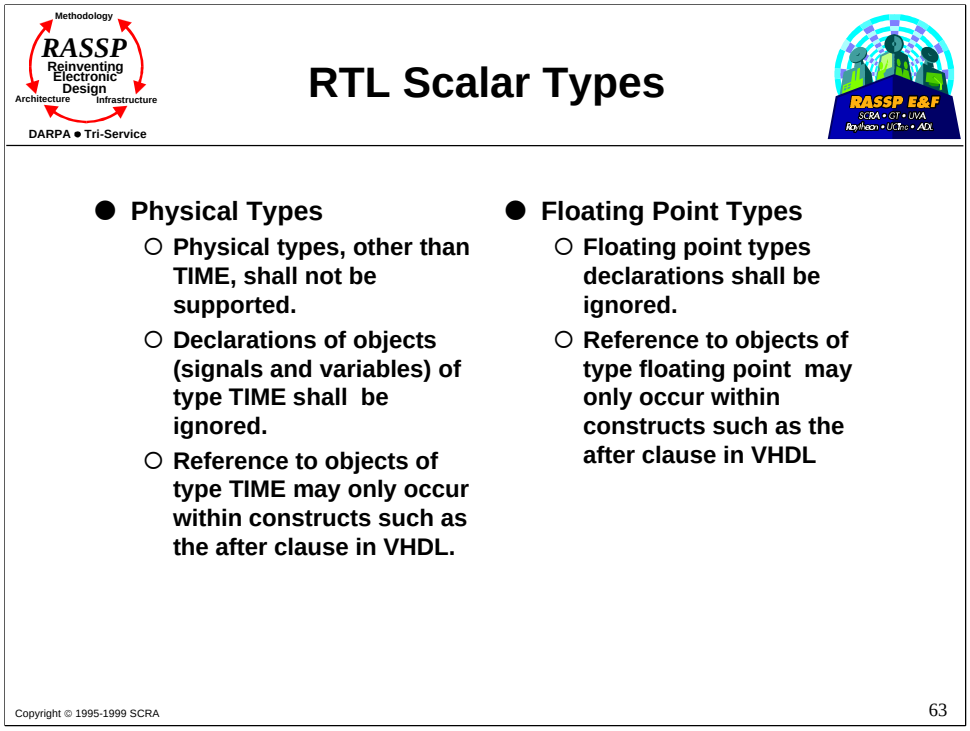
While the following set of slides may appear formal and too detailed, the reader would agree that this level of understanding is necessary for any serious design exploration using VHDL.



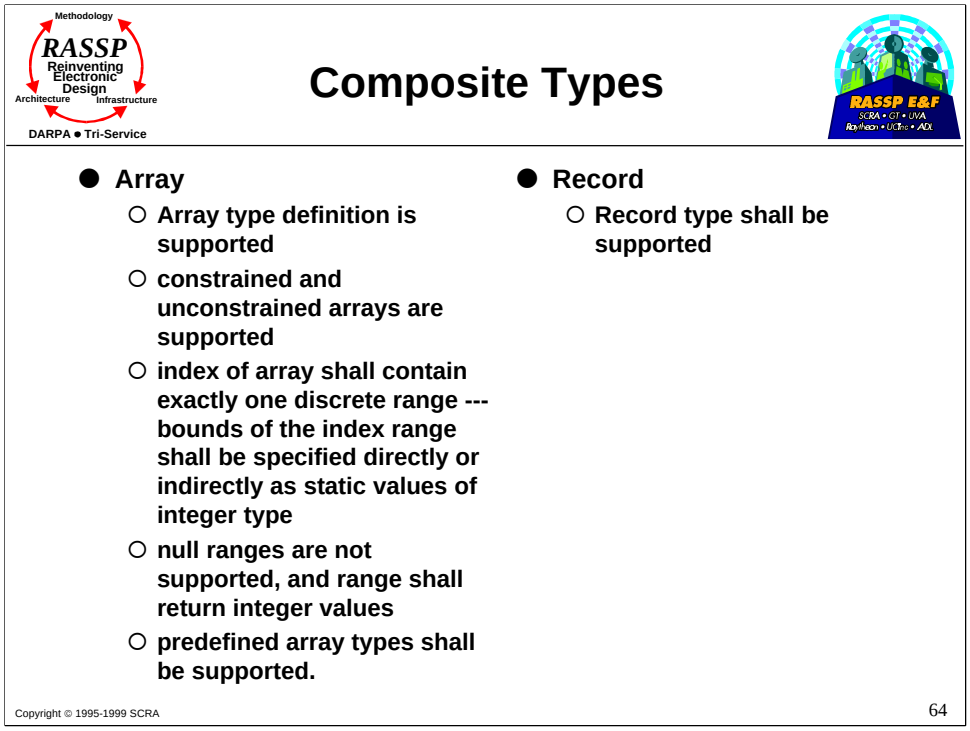
The reader may note that the physical and floating point types are ignored, while the access and file types are not supported in synthesis.



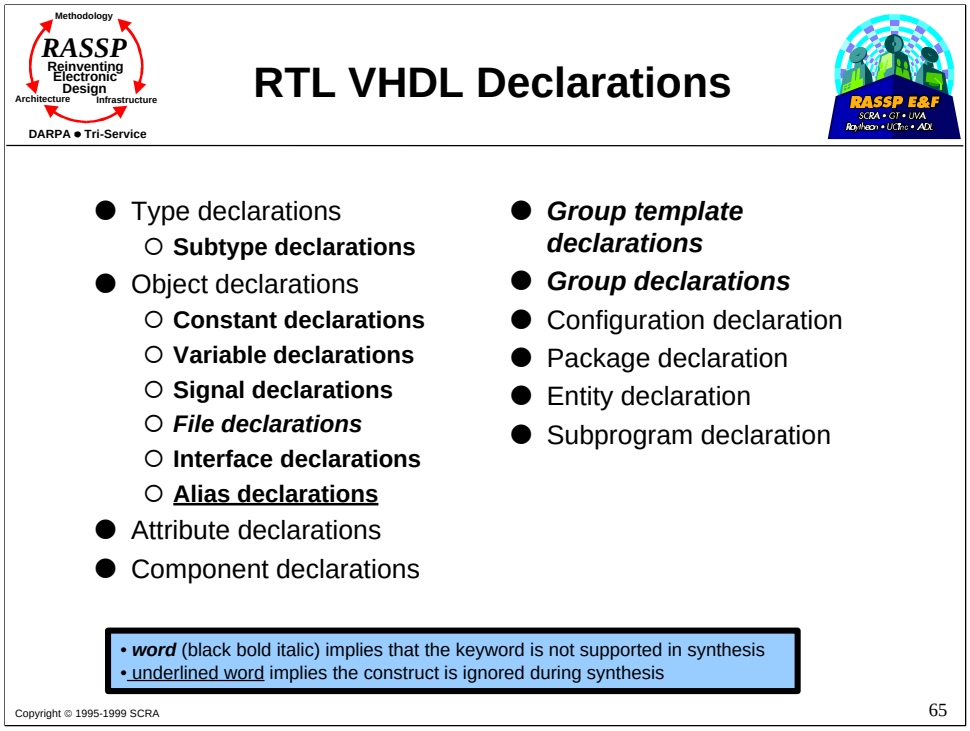
We now describe how the scalar types that are supported in synthesis.



Continuation of the description of scalar types that are supported in VHDL .

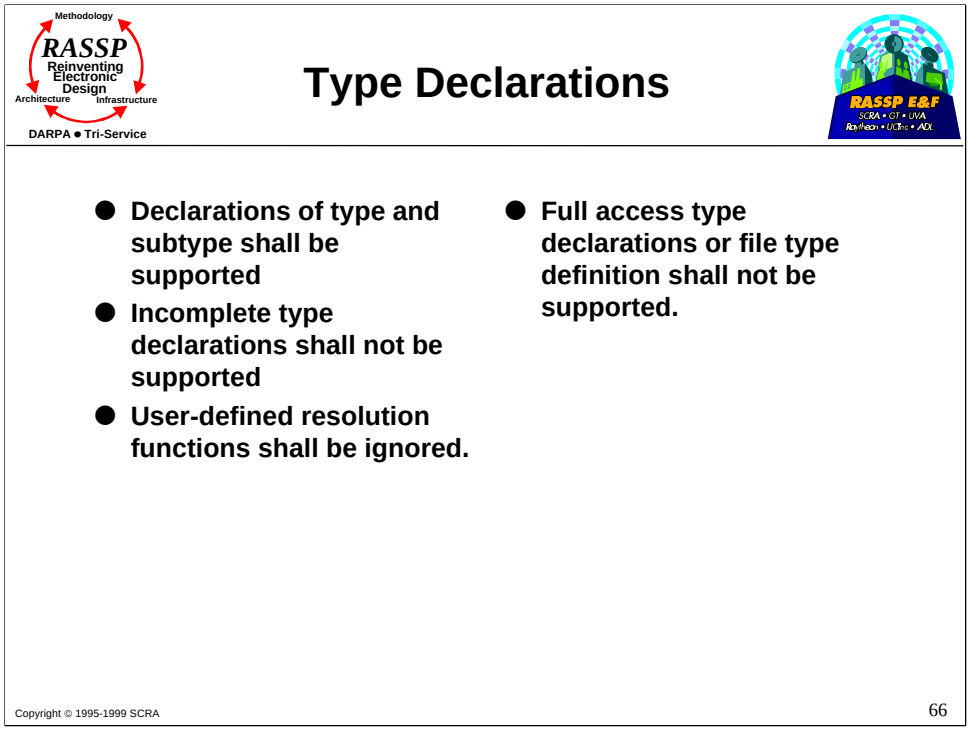


We describe the composite types that are supported in Synthesis. Both array (matrices) and record types are supported.

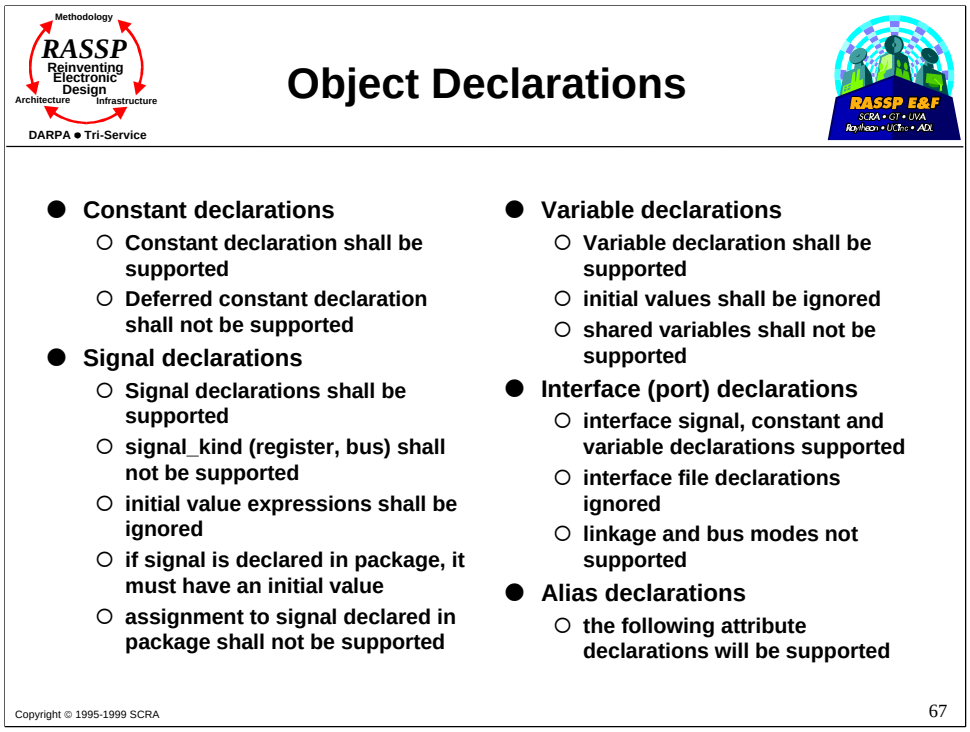


This slide describes the supported and unsupported (and ignored) declaration constructs in the RTL Synthesis Subset.

The following sections describe each of the declarations in further detail.



We describe in detail which features of the type declaration are supported.



Object declarations include constant, signal, variable and port declarations. Some features of these declarations are supported and some are not as shown above.

Supported Attribute Declarations

- Supported pre-defined attribute designators

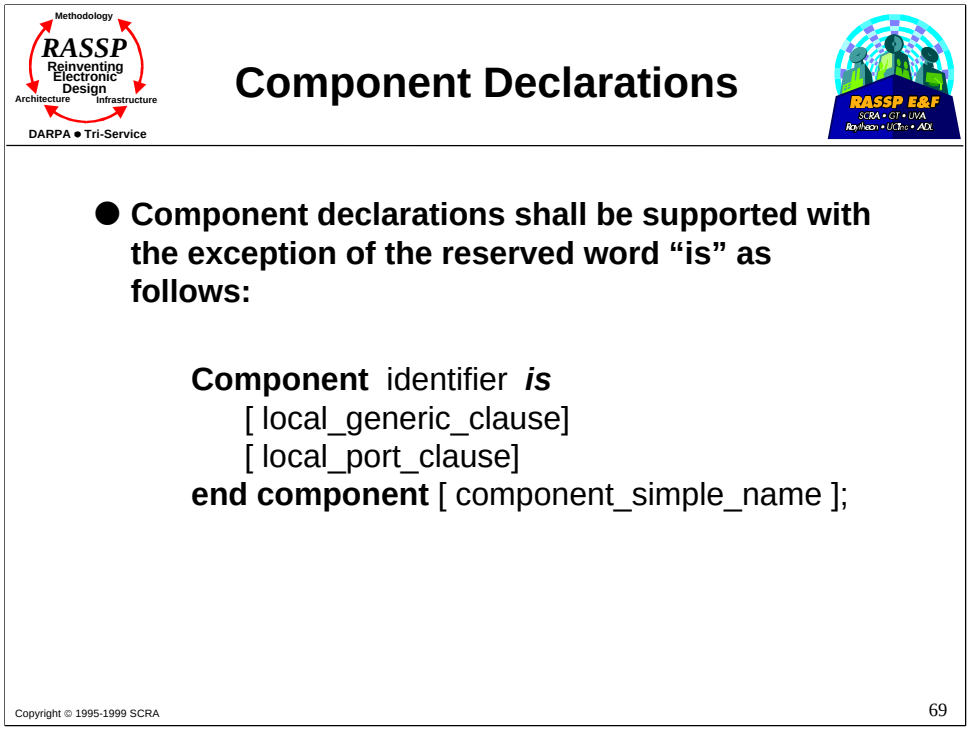
  'BASE, 'LEFT, 'RIGHT, 'RANGE, 'HIGH, 'LOW, 'REVERSE_RANGE, 'LENGTH, 'EVENT, 'STABLE

  'EVENT and 'STABLE shall only be used in the context of clock edge sensitive statements in VHDL

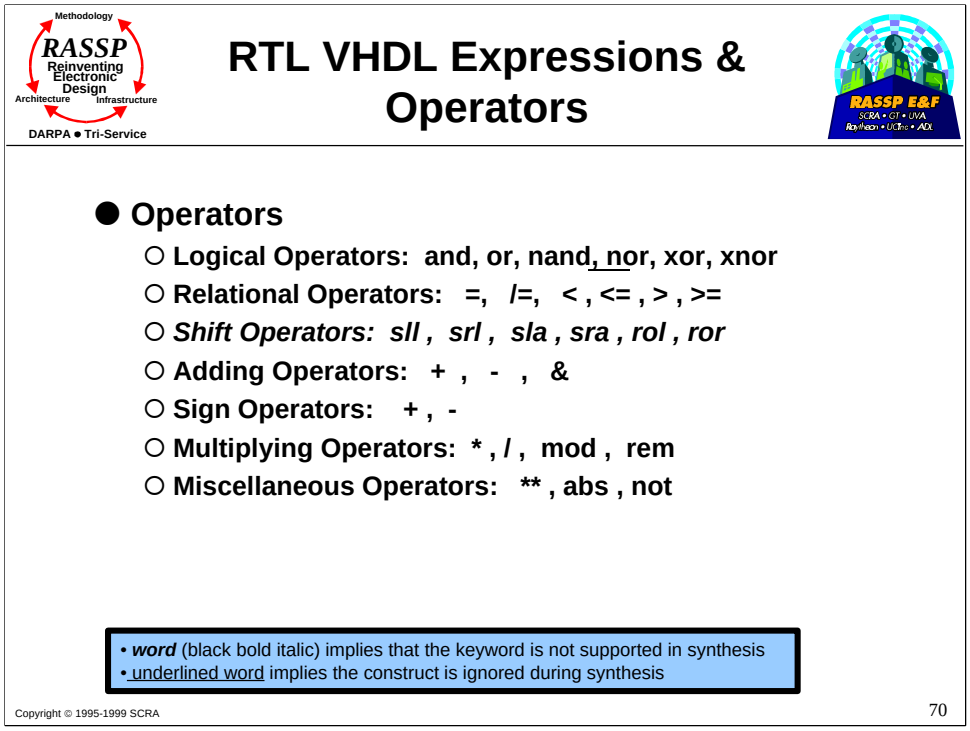
  Attributed
  signal'LENGTH [(n)] is not supported, but signal'LENGTH is supported.

  User defined attributes shall not be supported.

'Event is preferred over 'Stable, as the latter is active all the time in both simulation and synthesis.



The component statement (without the “is”) is supported in Synthesis.



The division (/), mod and rem operators are only supported when both operands are static or when the right operand is a static power of 2.

The exponentiation (**) operand is supported when both operands are static or when the left operand is a static power of 2.

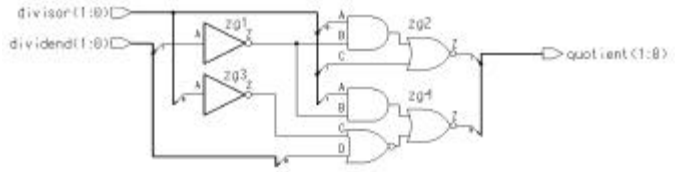
Element and array aggregates shall be permitted, record aggregates are not supported.

Precision is limited to 32 bits, and floating point expressions shall not be supported.

Operators

- **Generally, if the numeric_bit and numeric_std packages are supported, the operators within them are supported**
  - **Arithmetic operators - “abs”, “+”, “-”, “*”, “/”, “rem”, “mod”**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

ENTITY divider is
  PORT(divisor  : IN  unsigned(1 DOWNTO 0);
       dividend : IN  unsigned(1 DOWNTO 0);
       quotient : OUT unsigned(1 DOWNTO 0));
END divider;

ARCHITECTURE behavior OF divider IS
  BEGIN
    quotient <= dividend / divisor;
END behavior;
```



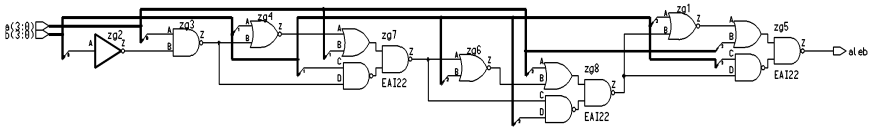
We show the synthesis of various operators in VHDL. These results are outputs of Mentor’s Autologic Synthesis tool.

Methodology
RASSP
Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

Operators (Cont.)

- **Comparison operators - “>”, “<“, “<=“, “>=“, “=“, “/=“**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

ENTITY compare is
  PORT(a    : IN  unsigned(3 DOWNTO 0);
       b    : IN  unsigned(3 DOWNTO 0);
       aleb : OUT boolean);
END compare;

ARCHITECTURE behavior OF compare IS
  BEGIN
    aleb <= (a <= b);
END behavior;
```

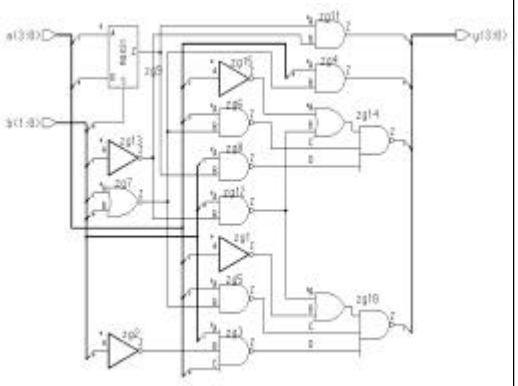


The VHDL code describes how a comparison operator is synthesized.

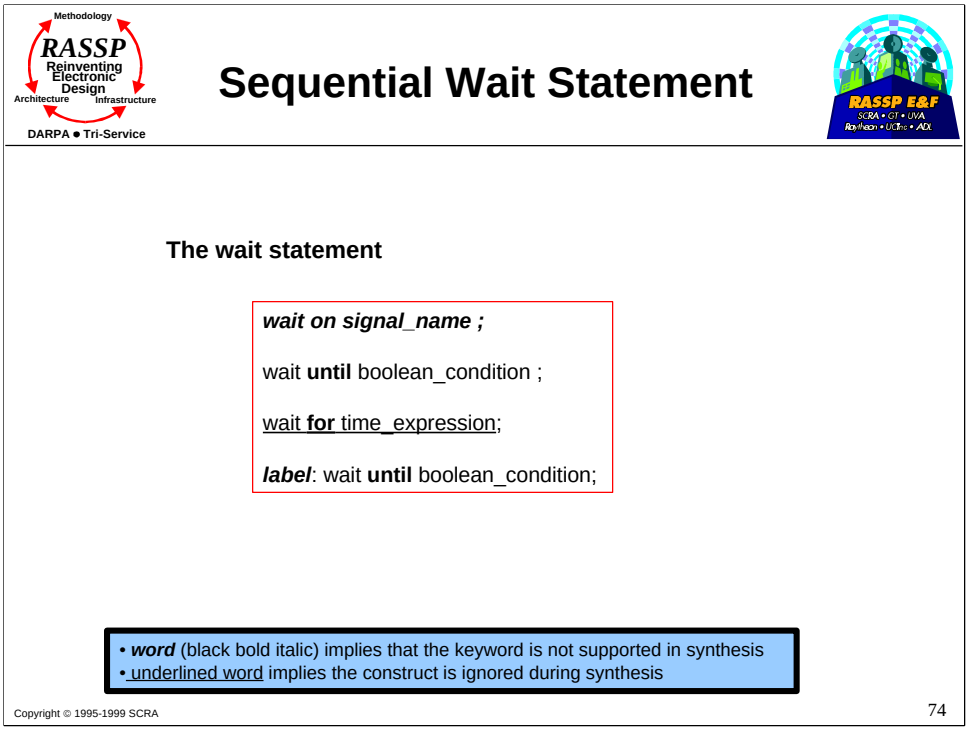
Operators (Cont.)

- **Shift and conversion operators - “shift_left”, “shift_right”, “rotate_left”, “rotate_right”, “resize”, “to_integer”, “to_unsigned”**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

ENTITY shift_4 is
  PORT(a    : IN  unsigned(3 DOWNTO 0);
       b    : IN  unsigned(1 DOWNTO 0);
       y    : OUT unsigned(3 DOWNTO 0));
END shift_4;

ARCHITECTURE behavior OF shift_4 IS
  BEGIN
    y <= shift_left(a,to_integer(b));
END behavior;
```



This slide describes the input VHDL code for the synthesis of the shift operator.



Note that only one wait per process is supported in synthesis. Thus we may not be able to support asynchronous set/reset inputs if a process is also sensitive to the clock. Wait for time_expression is ignored by the synthesis tools.

Sequential Assert/Report

Assert and Report Statements

Label: **assert** condition [report expression] [**severity** expression]

Label: ***report expression*** [**severity** expression]

Assert statement is ignored, and the label statement is not supported.
The report statement is not supported.

- ***word*** (black bold italic) implies that the keyword is not supported in synthesis
- underlined word implies the construct is ignored during synthesis

Sequential Signal Assignments

Syntax of the signal assignment statement

label: target <= [delay_mechanism] waveform ;

delay_mechanism :: = **transport** |
[reject time expression] inertial
waveform :: = waveform_element ***(, waveform_element)*** **|**
unaffected

Supported: target, waveform
Ignored: delay_mechanism, reserved word **after**
Not supported: label, **reject**, **inertial**, **unaffected**, time_expression, multiple waveform_elements, **null**.

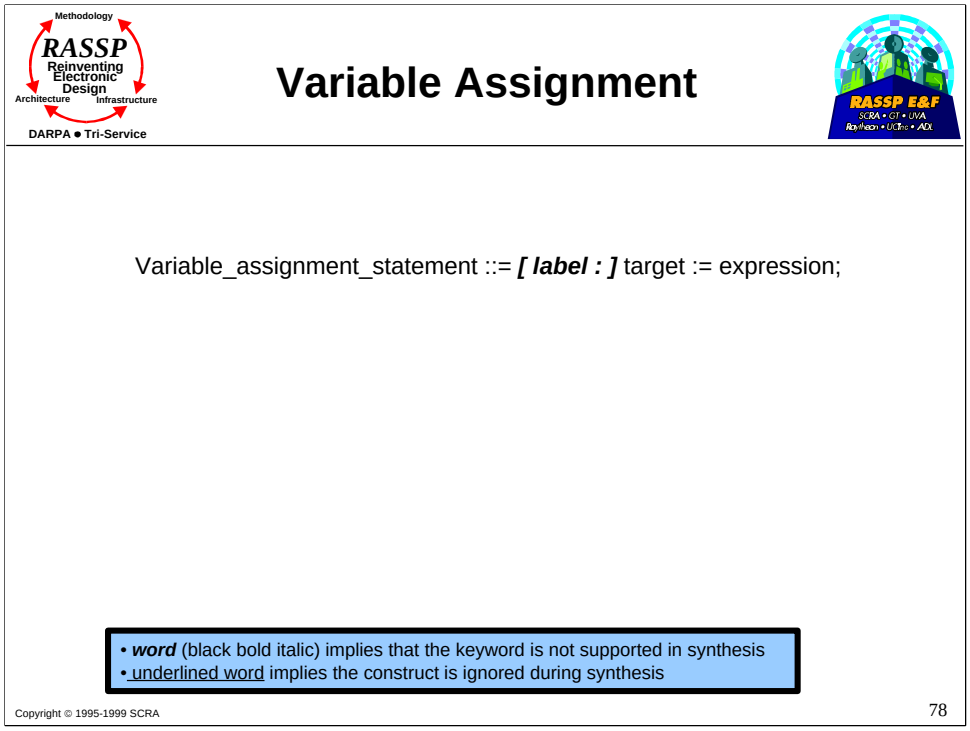
- ***word*** (black bold italic) implies that the keyword is not supported in synthesis
- underlined word implies the construct is ignored during synthesis

Signal Assignment Examples

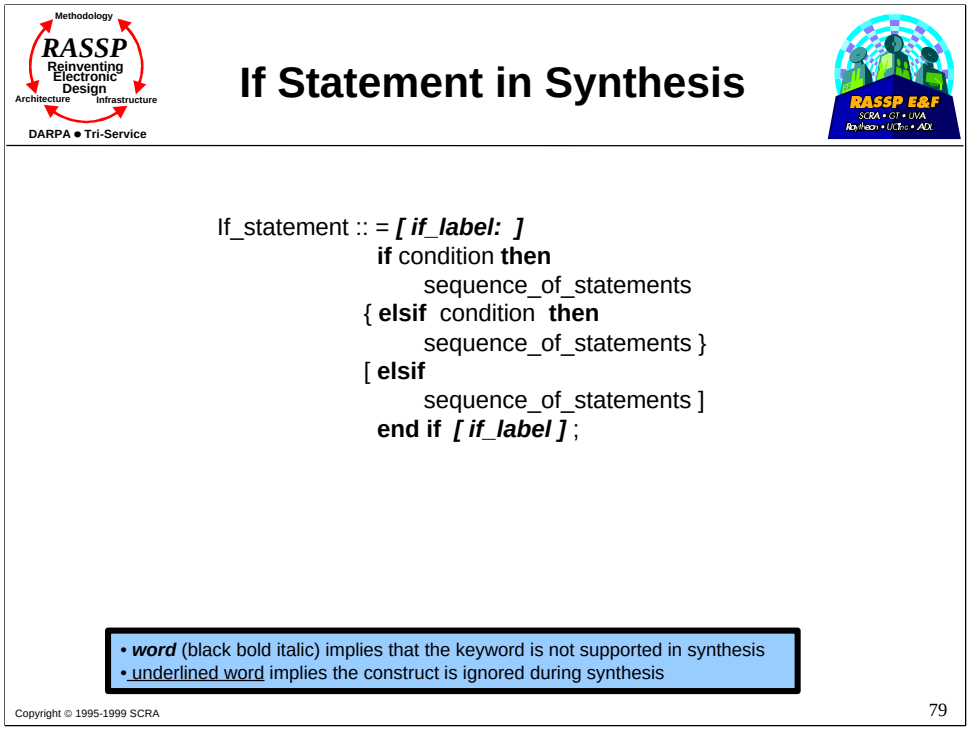
Examples of signal assignments in synthesis

label: **process** -- label is supported for process
begin
wait until clk'event and clk= '1'; -- only one wait statement
s1_s <= 10;
s2_s <= 1 **after** 4 ns; -- ***"after" is ignored***
s3_s <= 1 **after** 5 ns, 25 after 54 ns; -- two expressions
-- not allowed
s4_s <= **transport** 10 ; -- ***transport is ignored***
s5_s <= **transport** 5 **after** 4 ns, 24 after 10 ns;
-- inertial and reject not supported
s6_s <= ***reject 10 ns*** **inertial** 10 **after** 5 ns;
end process label;

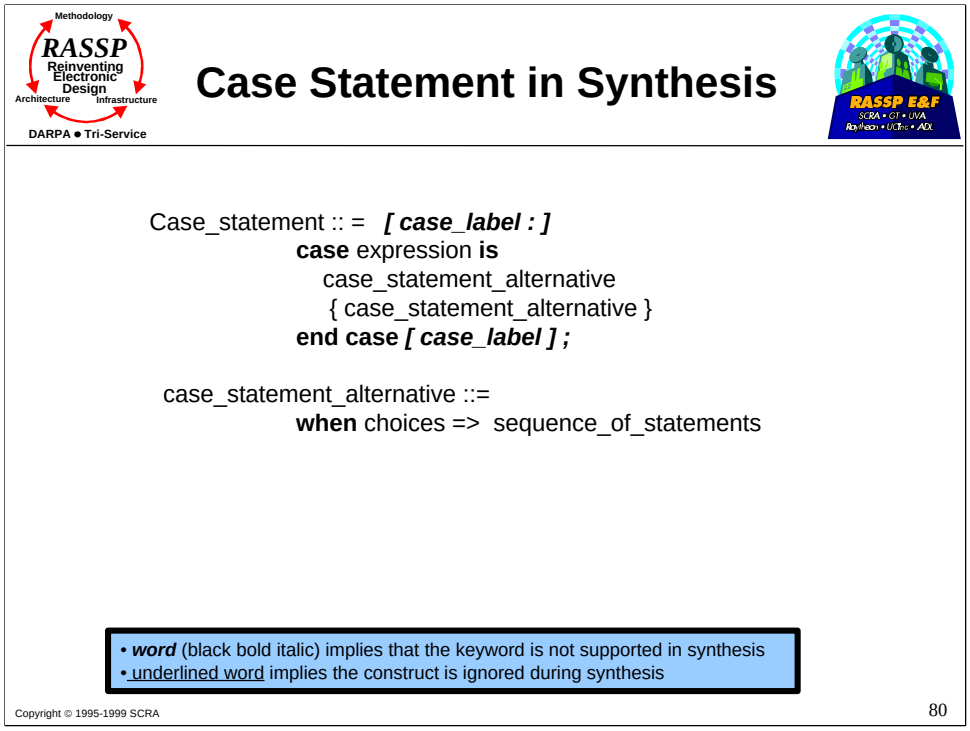
- ***word*** (black bold italic) implies that the keyword is not supported in synthesis
- underlined word implies the construct is ignored during synthesis



Note: Labels are not supported. Array variable assignments are supported.

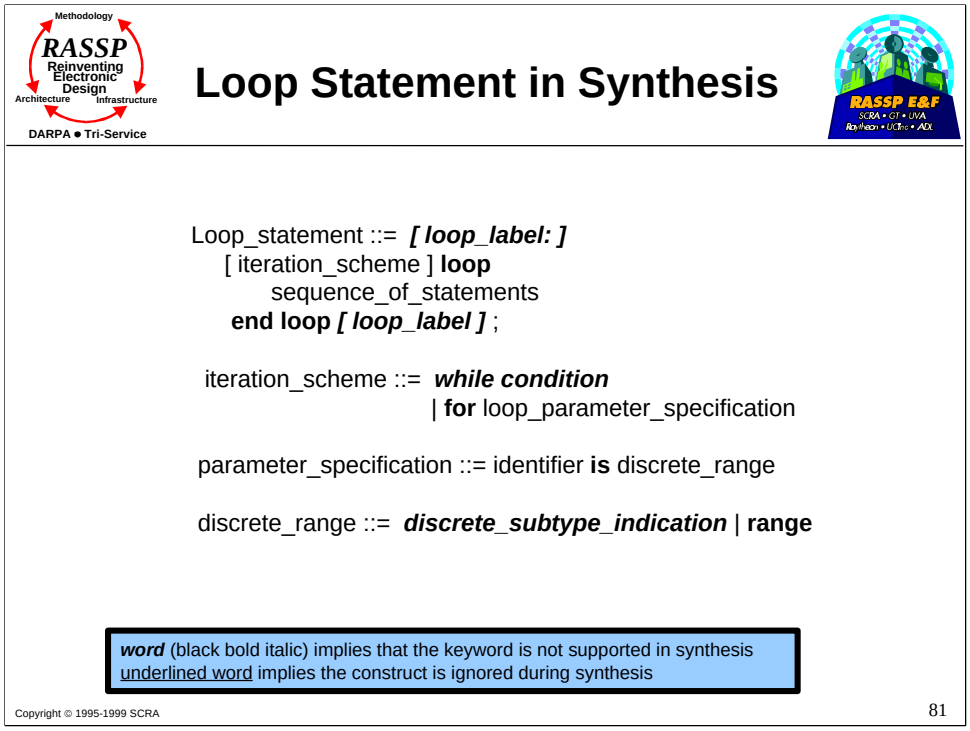


Note: If a signal or variable is assigned under some values of the conditional expressions in the if statement, but not for all values, level-sensitive logic (latches) may result in the synthesis result. Latches may also be synthesized if signal or variable is assigned in all conditional expressions, especially if the variable is read *before* it is written.

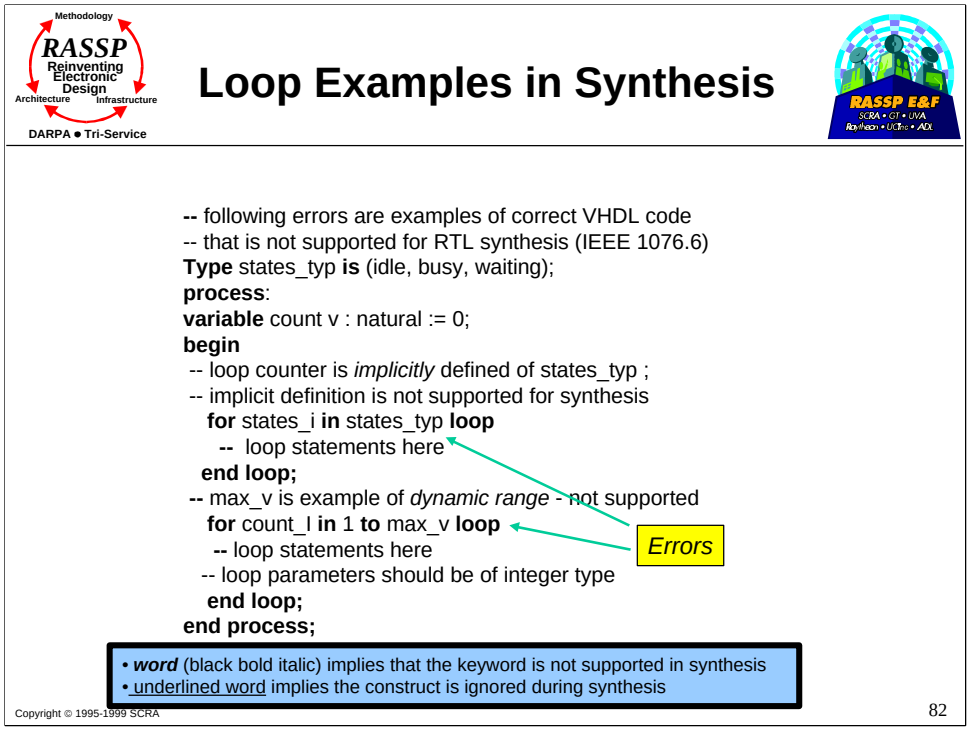


Note: If a signal or variable is assigned in some branches of the case statement, but not in all, level-sensitive sequential logic may result.

If a metalogical value occurs as a choice, or as an element of a choice, the synthesis tool may interpret it as a choice that may never occur. If only a certain metalogical value occurs in a case statement, the others choice must cover the missing values.



Note: Bounds of the discrete shall be specified directly or indirectly as static values belonging to the integer type.



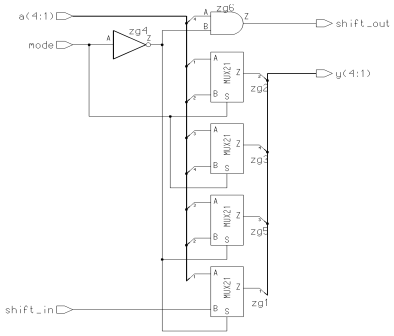
The errors are produced due to incorrect specification in the synthesis subset (even though the simulation may work correctly). The reasons for the errors are described in the comments within the code.

Methodology RASSP Reinventing Electronic Design Architecture Infrastructure DARPA ● Tri-Service

Sequential Loop Statements

● **Only `For` loops with integer range are supported**

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY shift4 is
  PORT(mode      : IN  std_logic;
       shift_in  : IN  std_logic;
       a         : IN std_logic_vector(4 DOWNTO 1);
       y         : OUT std_logic_vector(4 DOWNTO 1);
       shift_out : OUT std_logic);
END shift4;
```




```
ARCHITECTURE behavior OF shift4 IS
  SIGNAL in_temp  : std_logic_vector(5 DOWNTO 0);
  SIGNAL out_temp : std_logic_vector(5 DOWNTO 1);
  BEGIN
   in_temp(0) <= shift_in;
   in_temp(4 DOWNTO 1) <= a;
   in_temp(5) <= '0';
    comb : PROCESS(mode,in_temp,a)
      BEGIN
        FOR i IN 1 TO 5 LOOP
          IF(mode = '0') THEN
            out_temp(i) <= in_temp(i-1);
          ELSE
            out_temp(i) <= in_temp(i);
          END IF;
        END LOOP;
    END PROCESS comb;
    y <= out_temp(4 DOWNTO 1);
    shift_out <= out_temp(5);
END behavior;
```

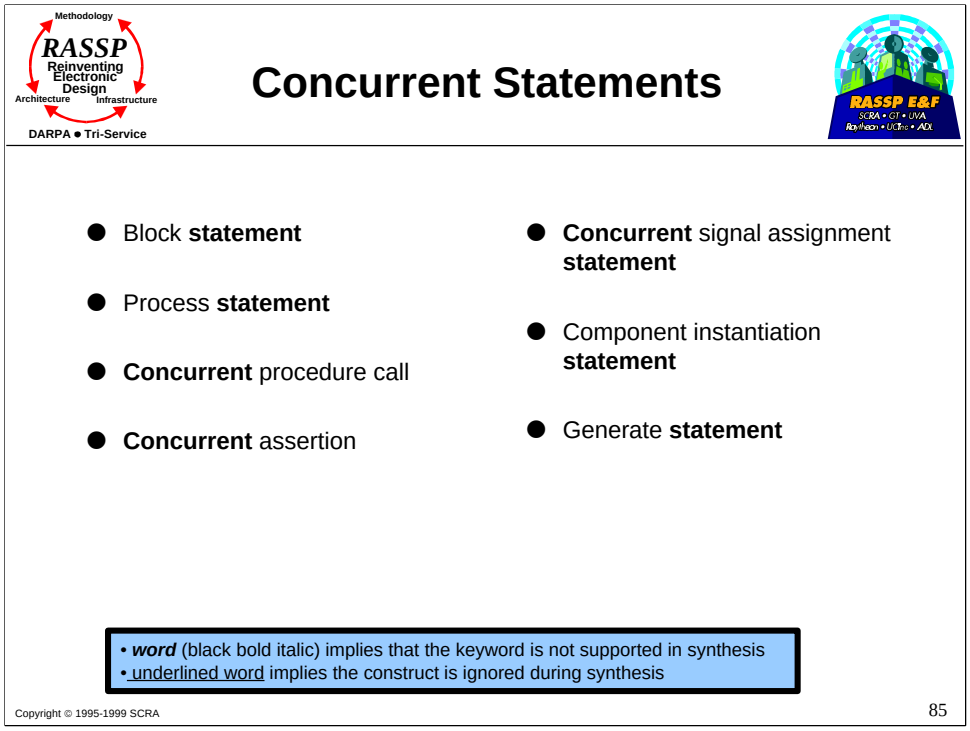
83

A loop conforming to the IEEE VHDL RTL syntax is synthesized using Mentor's Autologic tool as shown above.

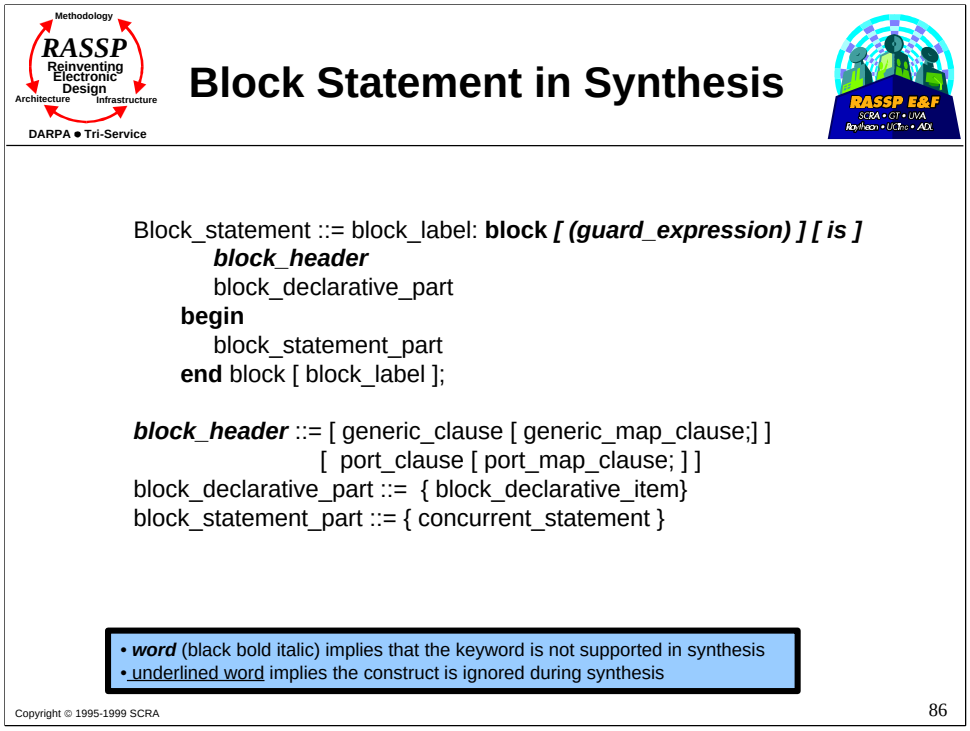
Miscellaneous Sequential RTL Statements

- Next
  - **next_statement ::=** ***[label:]*** next **[loop_label] [when condition];**
- Exit
  - **exit_statement ::=** ***[label:]*** exit **[loop_label] [** when **condition] ;**
- Null
  - **null_statement ::=** ***[label:]*** null **;**
- Return
  - **return_statement ::=** ***[label]*** return **[expression] ;**

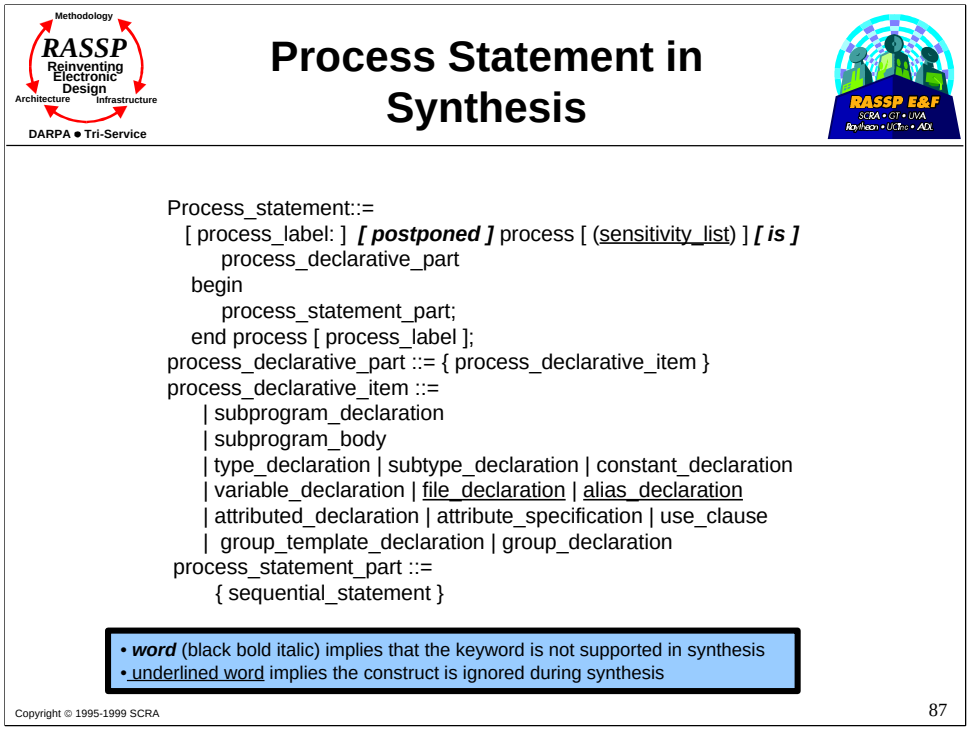
• ***word*** (black bold italic) implies that the keyword is not supported in synthesis
• underlined word implies the construct is ignored during synthesis



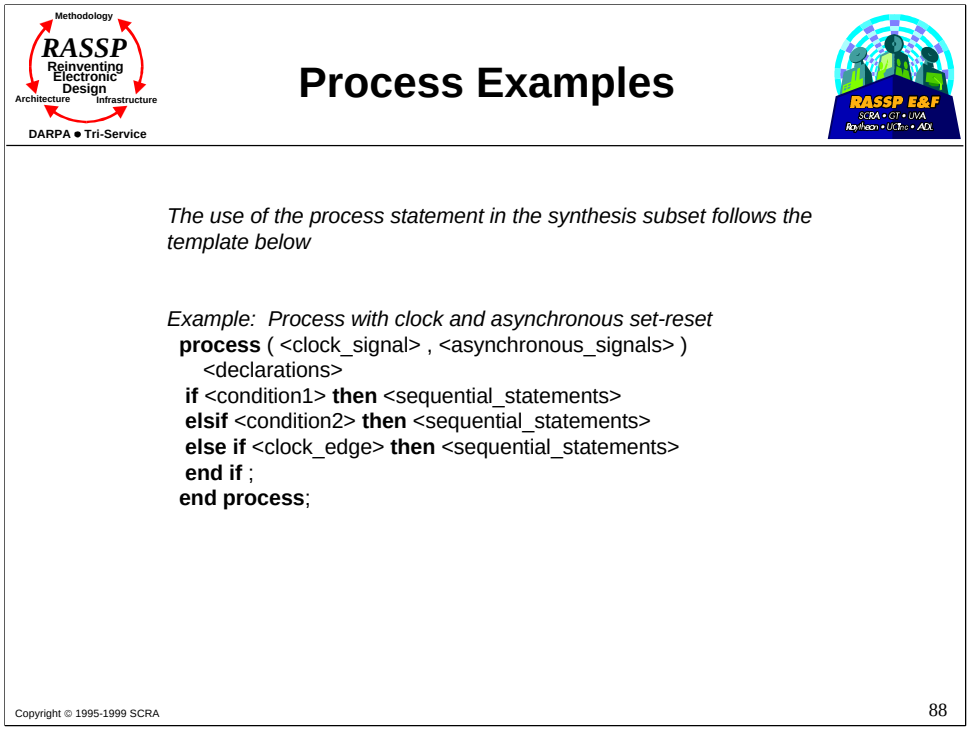
All the concurrent statements in VHDL are supported, though some of them are only supported partially. This is the reason why we have to study the syntax diagrams in detail, or our knowledge of the synthesis subset will be superficial at best.



Note: the component instantiation statement is more powerful and useful than the block statement.



As can be observed, most features of the process statement are supported including sensitivity lists.

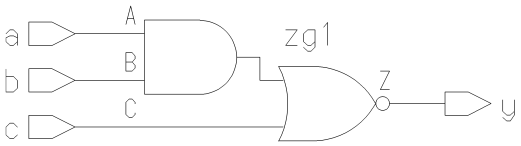


Note: the sensitivity_list must include those signals that are *read* by the process, except for those signals read only under the control of a clock edge . The sensitivity_list usually contains the clock and the asynchronous signals read by the process.

Note: asynchronous signals have higher priority and are level sensitive. A clock edge appears only in the *last* **else if** statement. The sequential statements cannot contain any **if** statement sensitive to the clock, or any **wait** statement.

Process Statements Example

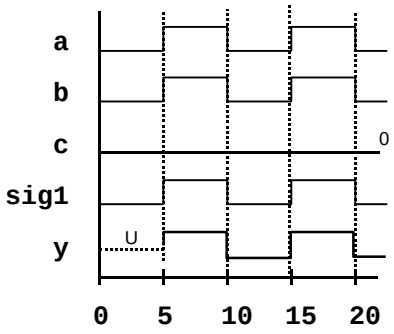
```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY aoi_process is
  PORT(a : IN  std_logic;
       b : IN  std_logic;
       c : IN  std_logic;
       y : OUT std_logic);
END aoi_process;

ARCHITECTURE behavior OF aoi_process IS
  SIGNAL zg1 : std_logic;
  BEGIN
    comb : PROCESS(a,b,c,zg1)
      BEGIN
        zg1 <= a AND b;
        y <= not(zg1 or c);
    END PROCESS comb;
END behavior;
```

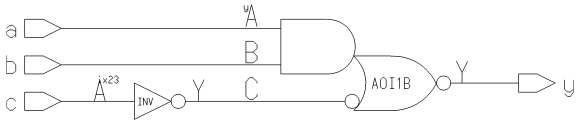


This example shows the code containing a sythesizable process statement and the result of the synthesis process (in this case a combinational circuit).

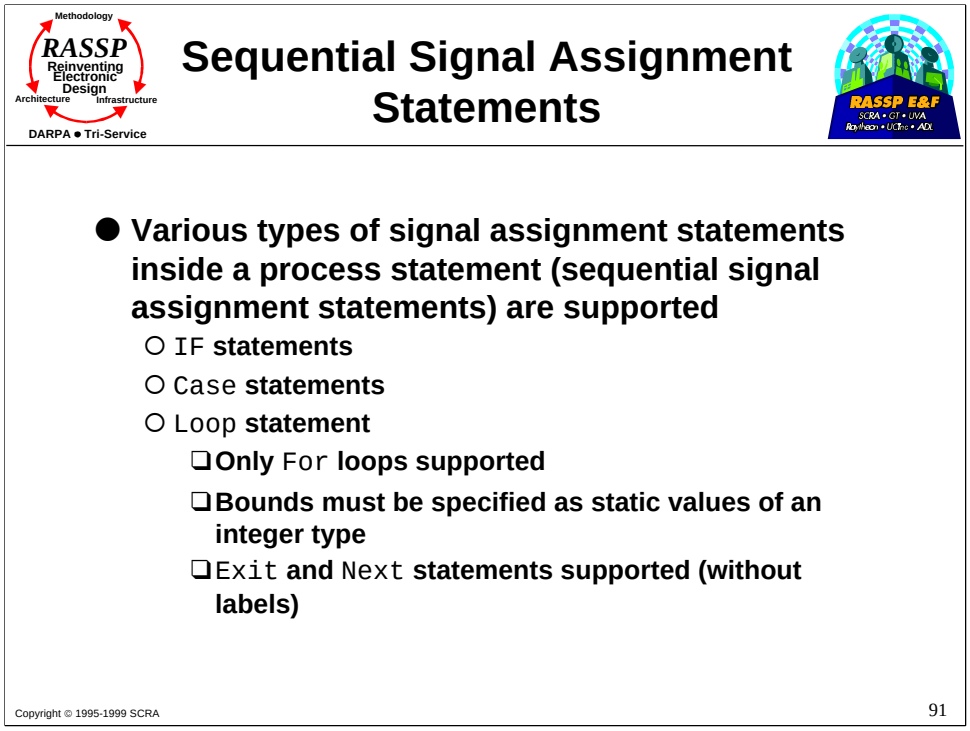
Process Statements
Incomplete Sensitivity List

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY aoi_process is
  PORT(a : IN  std_logic;
       b : IN  std_logic;
       c : IN  std_logic;
       y : OUT std_logic);
END aoi_process;

ARCHITECTURE behavior OF aoi_process IS
  SIGNAL sig1 : std_logic;
  BEGIN
    comb : PROCESS(a,b,c)
      BEGIN
        sig1 <= a AND b;
        y <= not(sig1 or c);
    END PROCESS comb;
END behavior;
```

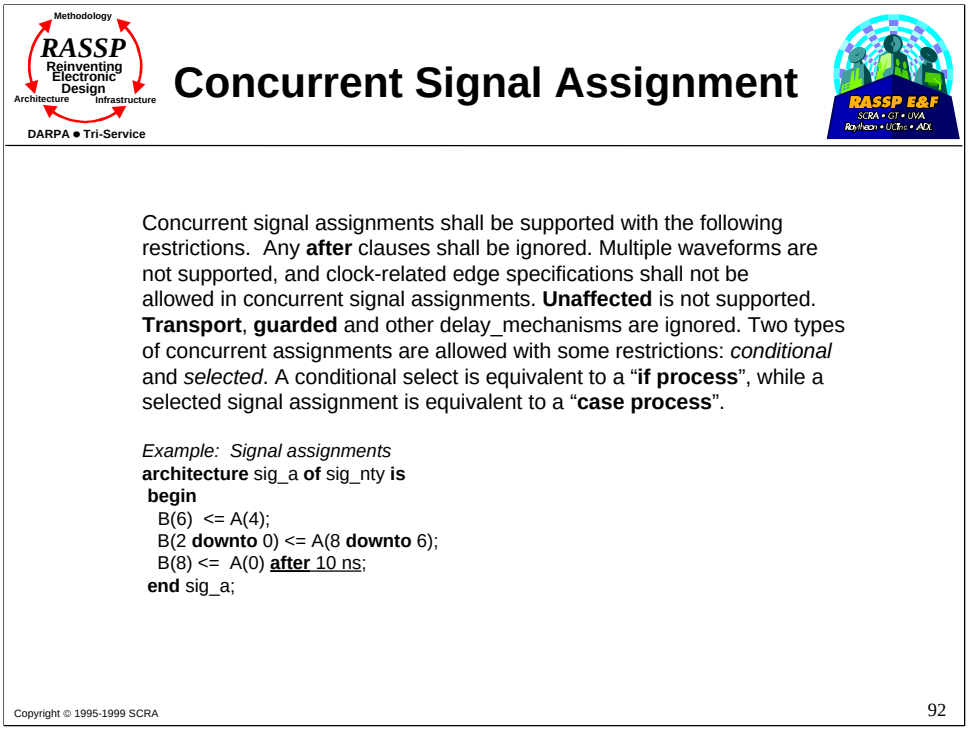




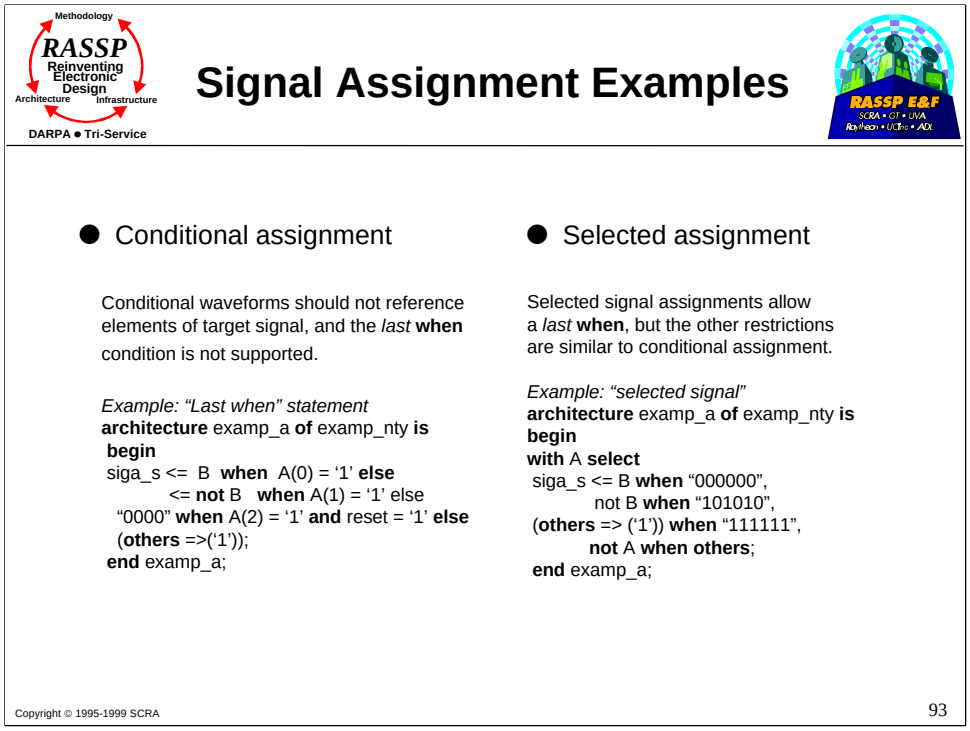
Note that the sensitivity list did not contain sig1, and is thus incomplete, resulting in possible difference between the synthesis and its simulation in VHDL.



We describe a few sequential assignment statements with examples in this and the following slides.



The "after" keyword is ignored in synthesis. Concurrent signal assignments are fully supported.

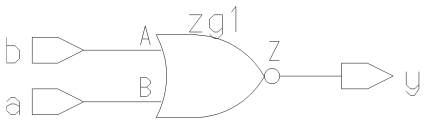


Note the presence and absence, respectively, of the "when" keyword in the above examples.

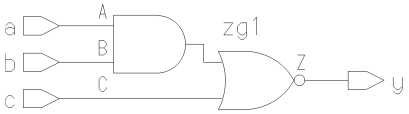
Concurrent Signal Assignment Statements

- **Simple concurrent signal assignment statements are supported**
  - **Multiple waveform elements are not allowed**

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY csa is
  PORT(a : IN  std_logic;
       b : IN  std_logic;
       y : OUT std_logic);
END csa;

ARCHITECTURE behavior OF csa IS
  BEGIN
    y <= a NOR b;
END behavior;
```




```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY aoi_csa is
  PORT(a : IN  std_logic;
       b : IN  std_logic;
       c : IN  std_logic;
       y : OUT std_logic);
END aoi_csa;

ARCHITECTURE behavior OF aoi_csa IS
  SIGNAL sig1,sig2 : std_logic;
  BEGIN
    sig1 <= a AND b;
    sig2 <= c OR sig1;
    y <= NOT sig2;
END behavior;
```

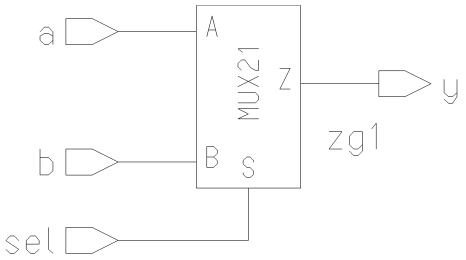


Two examples of combinational synthesis without the use of processes are illustrated in this slide.

Conditional Signal Assignment Statements

- **Concurrent conditional signal assignment statements are supported - must end in `else` clause**

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY mux2 is
  PORT(a   : IN  std_logic;
       b   : IN  std_logic;
       sel : IN std_logic;
       y   : OUT std_logic);
END mux2;

ARCHITECTURE behavior OF mux2 IS

  BEGIN
   y <= a WHEN (sel = '0') ELSE
        b WHEN (sel = '1') ELSE
        'X';
END behavior;
```

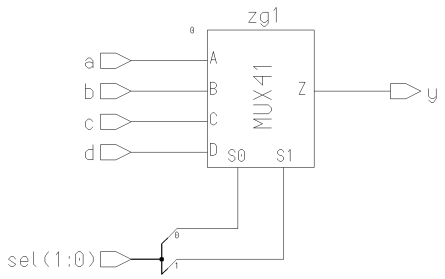


The conditional if statement results in a multiplexor as shown in the example above.

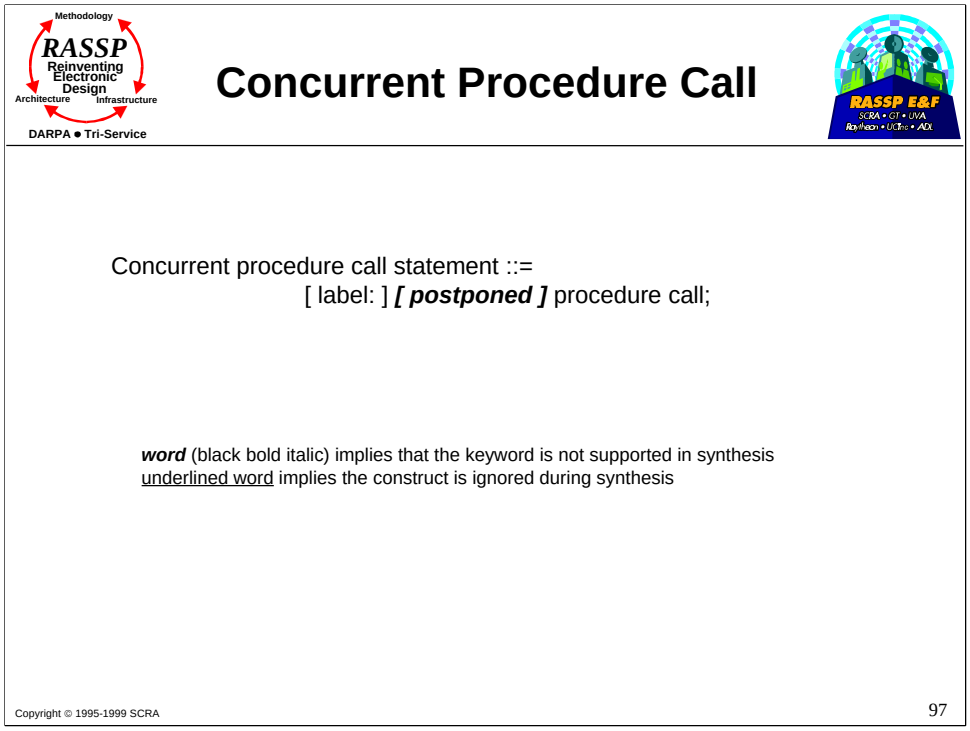
Selected Signal Assignment Statements

- **Concurrent selected signal assignment statements are supported**

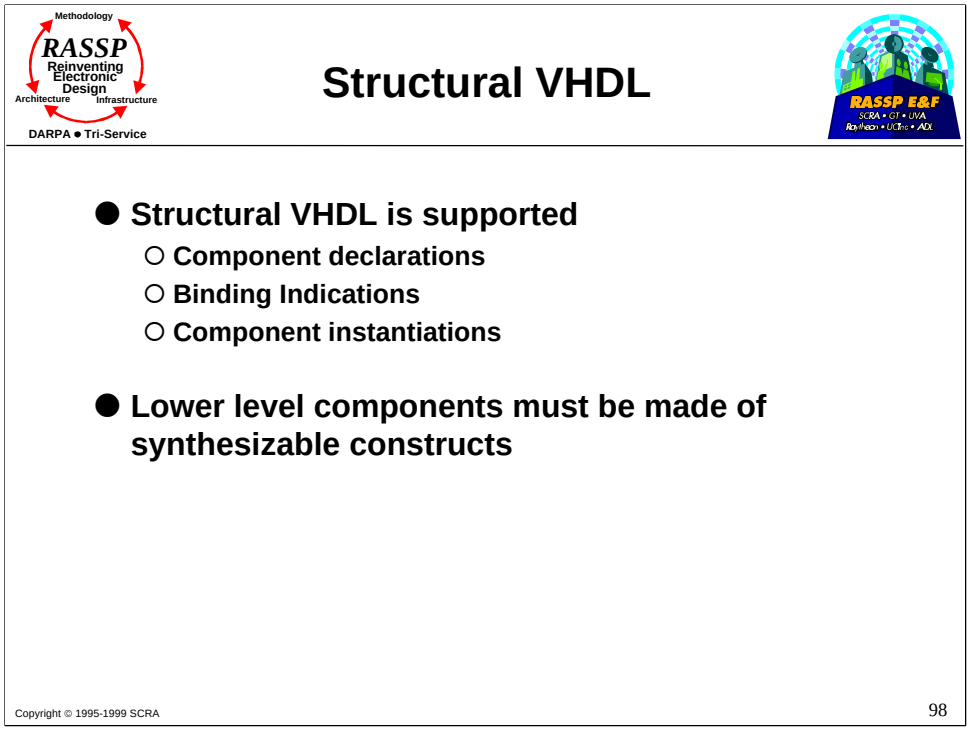
```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY mux4 is
  PORT(a   : IN  std_logic;
       b   : IN  std_logic;
       c   : IN  std_logic;
       d   : IN  std_logic;
       sel : IN std_logic_vector(1 DOWNTO 0);
       y   : OUT std_logic);
END mux4;

ARCHITECTURE behavior OF mux4 IS
  BEGIN
    WITH sel SELECT
      y <= a WHEN "00",
           b WHEN "01",
           c WHEN "10",
           d WHEN "11",
         'X' WHEN OTHERS;
END behavior;
```



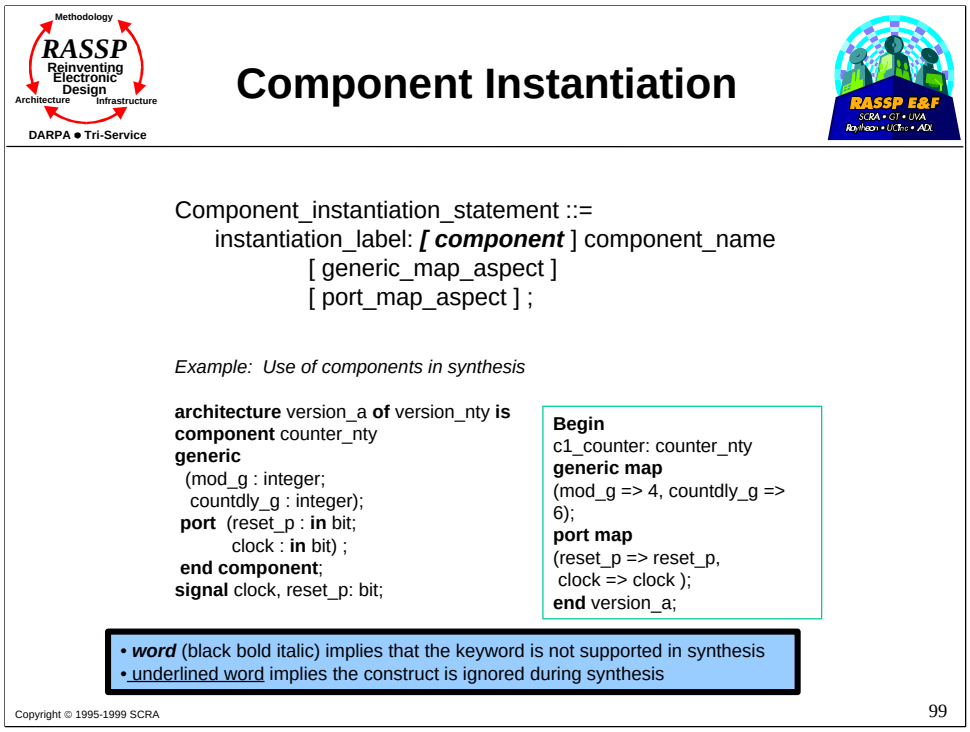
The selected signal assignment is an alternative realization of a multiplexor.



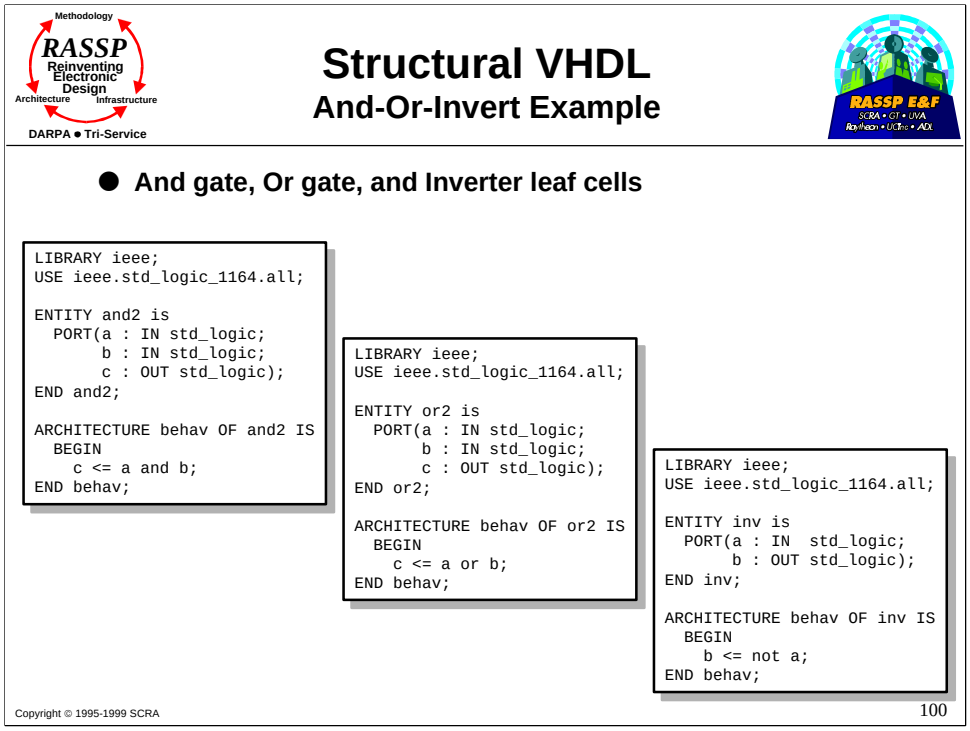
Note: Different instantiations of the concurrent procedure call make different associations between actual and formal parameters, but it is important for the calling architecture to have visibility over the variables, signals being associated. Files parameters are not supported by the RTL synthesis subset.



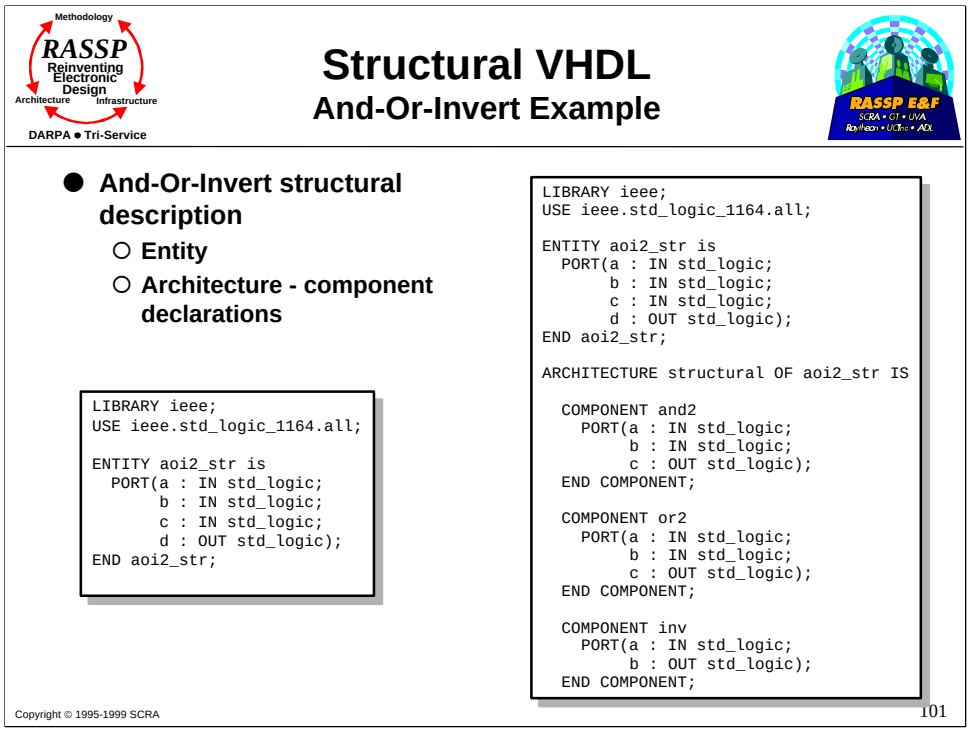
Structural VHDL is used to describe netlists and interconnection of components within VHDL, and is very useful within the digital design process. A number of examples will illustrate VHDL's support for structural synthesis.



Note: only constants can have an initial value (of integer subtype), other initial values (of signals and variables in generics) are ignored.



This slide describes the components used as entity/architecture pairs.



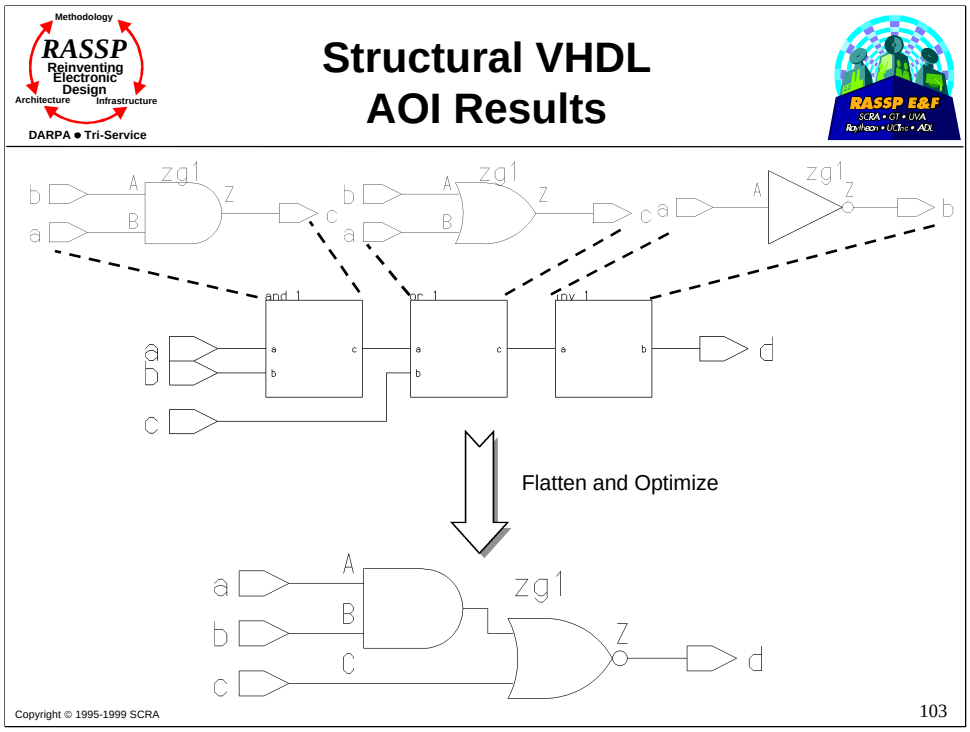
The components described in the previous slide are used in this slide as an example of structural VHDL.

Structural VHDL
And-Or-Invert Example

- **Architecture - binding indications, signal declarations, component instantiations**

```
  FOR ALL : and2 USE ENTITY work.and2(behav);
  FOR ALL : or2  USE ENTITY work.or2(behav);
  FOR ALL : inv  USE ENTITY work.inv(behav);

  SIGNAL and_out : std_logic; -- signal for output of and gate
  SIGNAL or_out  : std_logic; -- signal for output of or gate

  BEGIN
    AND_1 : and2 PORT MAP(a => a, b => b, c => and_out);

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

    INV_1 : inv  PORT MAP(a => or_out, b => d);
END structural;
```


Port map declarations are supported as shown in this slide.



The structural description is optimized and flattened further by logic synthesis tools and results in the final implementation depicted in the bottom half of the slide.

Generate Statement

Both the *if-generate* and *for-generate* forms shall be supported.

Example: generate in synthesis

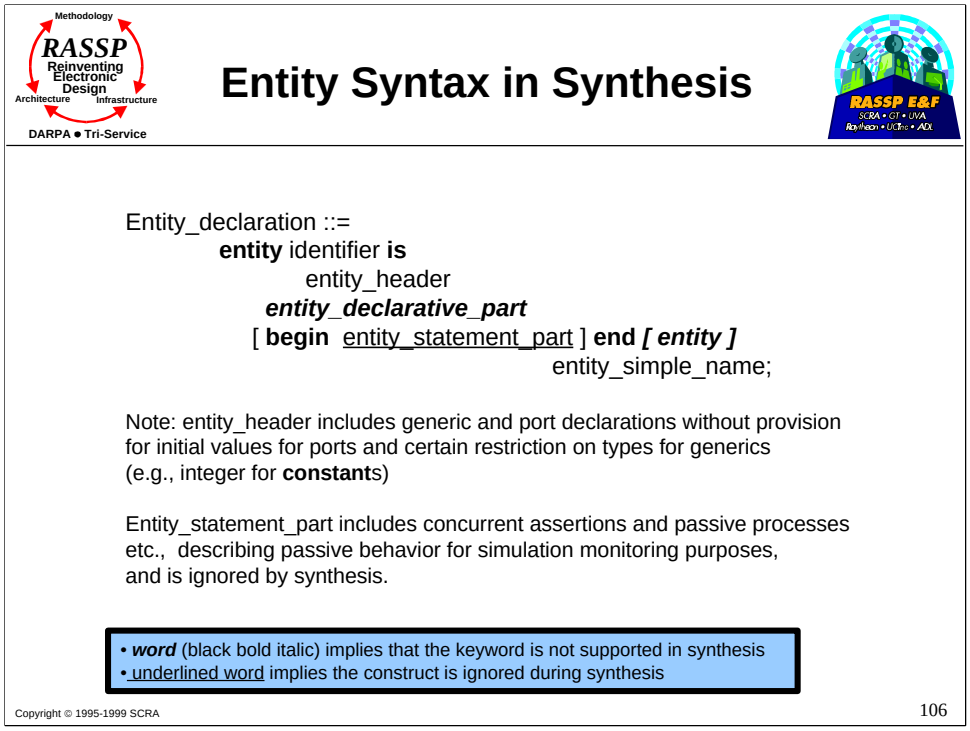
```
…
begin
G1: for J in 0 to 3 generate
 G2: if J = 0 generate
    J1: d_ff port map (clear, count, ct(j));
  end generate G2;
G3: if J > 0 generate
   J2: d_ff port map (clear, ct(j-1), ct(j));
 end generate G3;
 q(j) <= ct(j);
 end generate G1;
```

Generate parameter should be statically computable.

The generate statement is very useful in the construction of regular structures consisting of a number of lower - level building blocks.

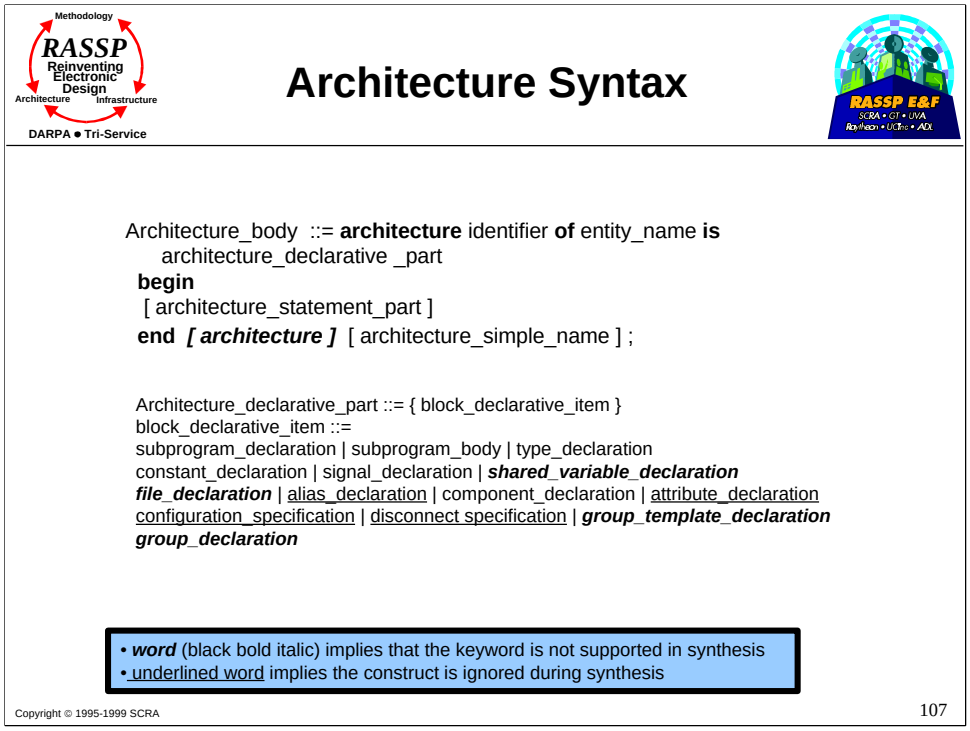
Design Entities in Synthesis

- Entity
- Architecture
- Configuration
- *Component configuration*
- Subprograms
- Package
- Package body

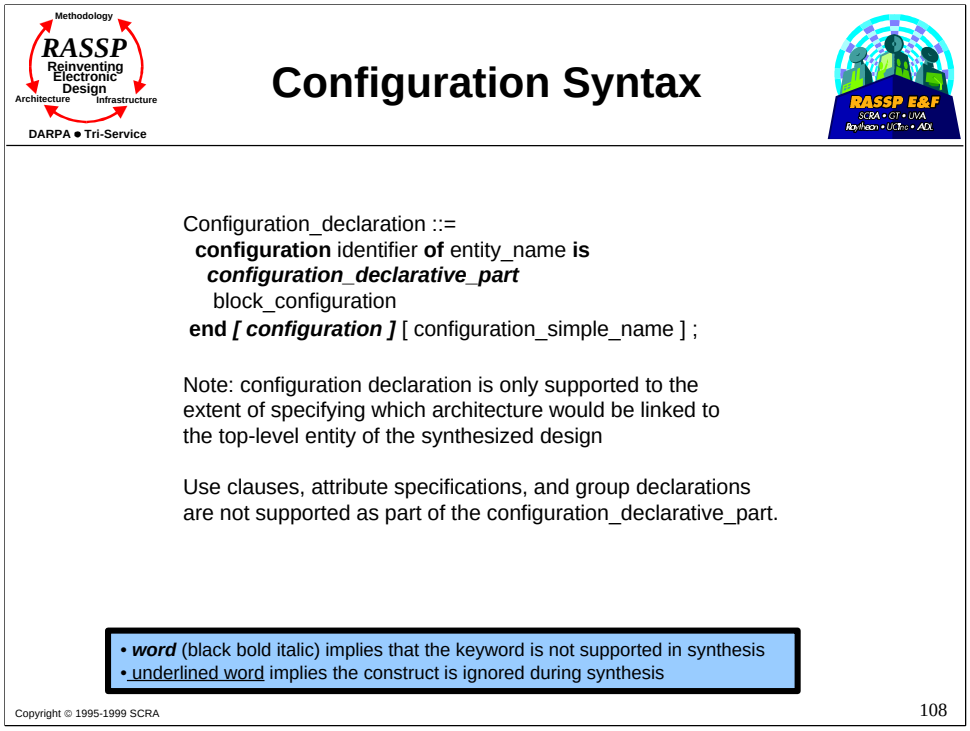


Note: entity_header includes generic and port declarations without provision for initial values for ports and certain restriction on types for generics (e.g., integer for **constant**s)

Entity_statement_part includes concurrent assertions and passive processes etc., describing passive behavior for simulation monitoring purposes, and is ignored by synthesis.

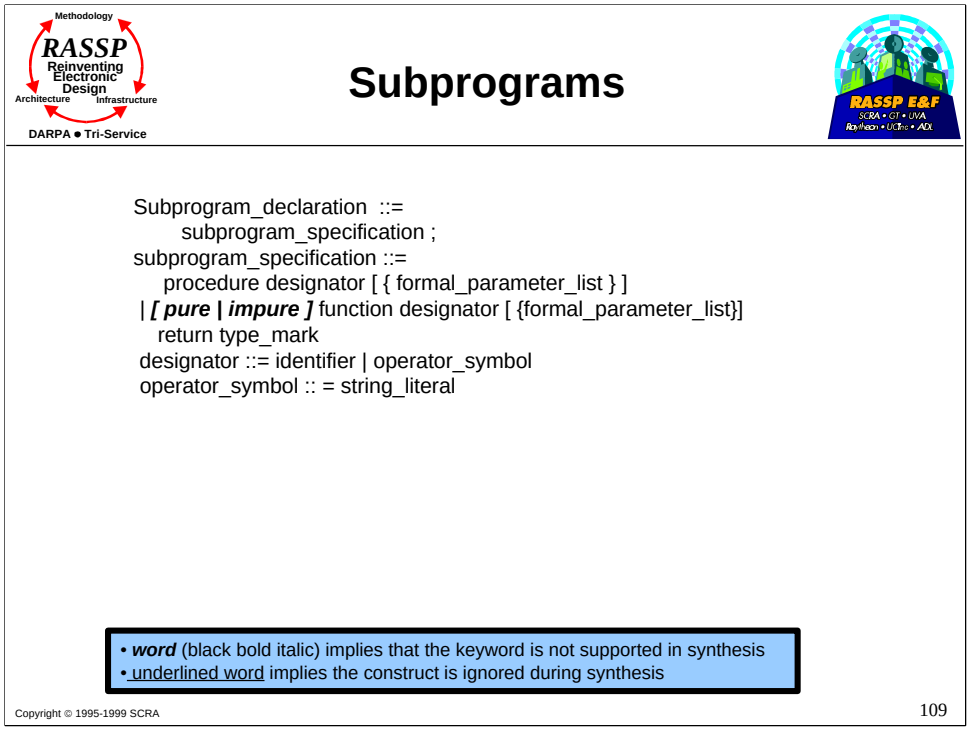


Note: user-defined attributes are not supported. Multiple architectures are supported.



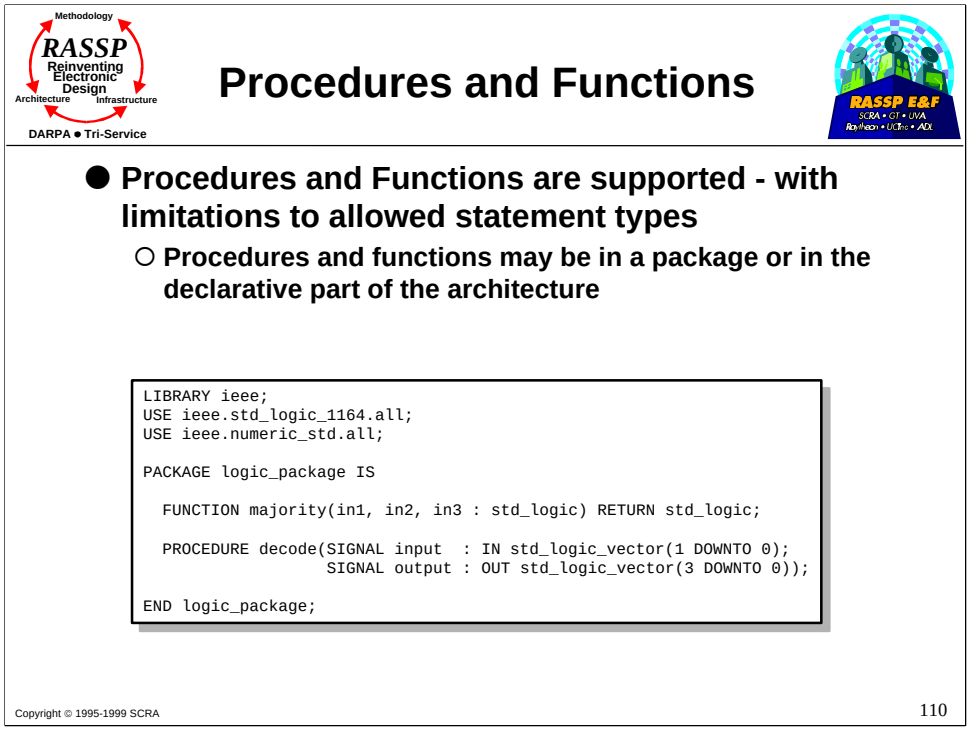
Note: configuration declaration is only supported to the extent of specifying which architecture would be linked to the top-level entity of the synthesized design.

Use clauses, attribute specifications, and group declarations are not supported as part of the configuration_declarative_part.

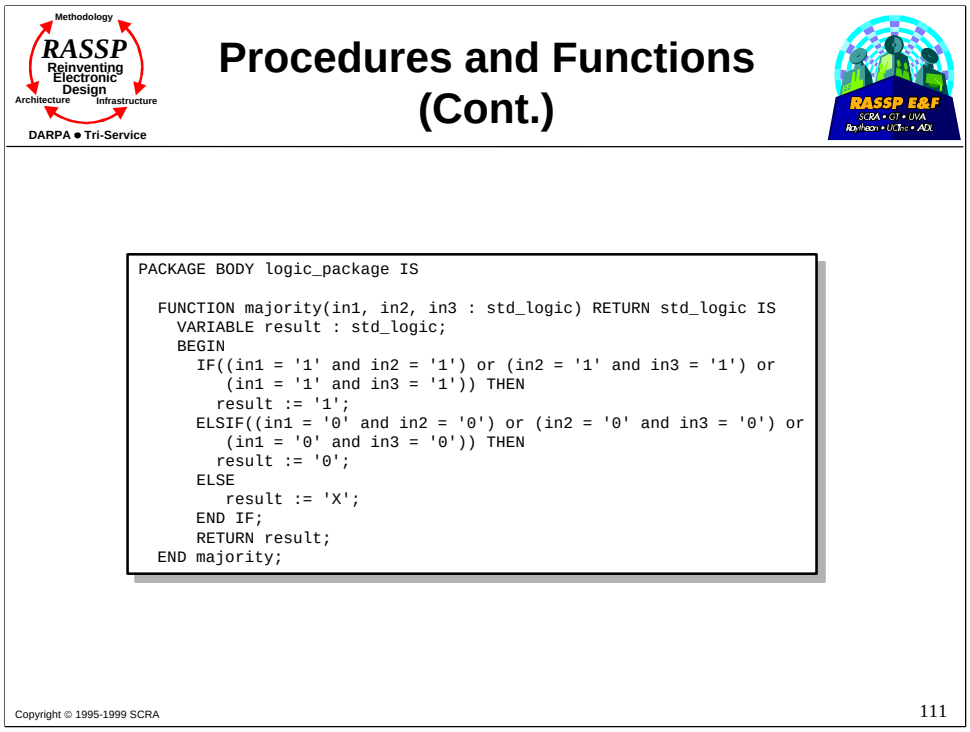


Note: constant, variable, and signal parameters shall be supported, subject to earlier restrictions. File parameters are not supported. Alias declarations shall be ignored, and group declarations are not supported.

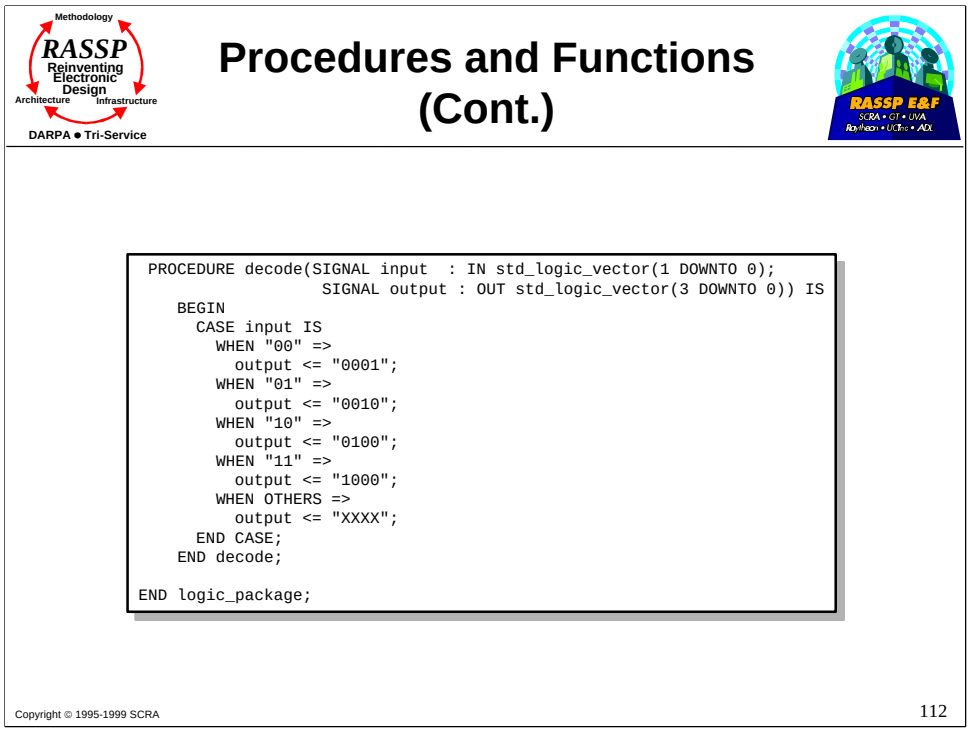
Resolution functions are ignored, with the exception of RESOLVED in subtype STD_LOGIC. Operator overloading shall also be supported.



Procedures and functions have restrictions on the VHDL syntax they can use, similar to the restriction placed on VHDL architectures.



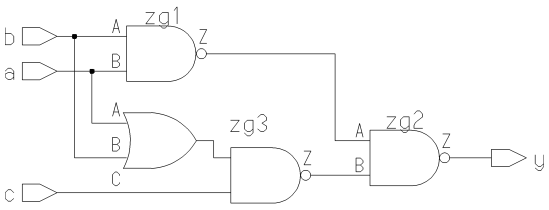
Example of a majority function.



Example of a decode function within a package.

Using Procedures and Functions

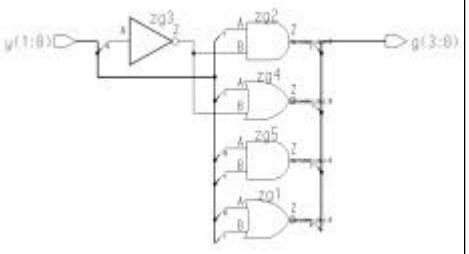
```
 LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE work.logic_package.all;

ENTITY voter IS
   PORT(a : IN  std_logic;
        b : IN  std_logic;
        c : IN  std_logic;
        y : OUT std_logic);
END voter;

ARCHITECTURE maj OF voter IS
  BEGIN
    y <= majority(a,b,c);
END maj;
```



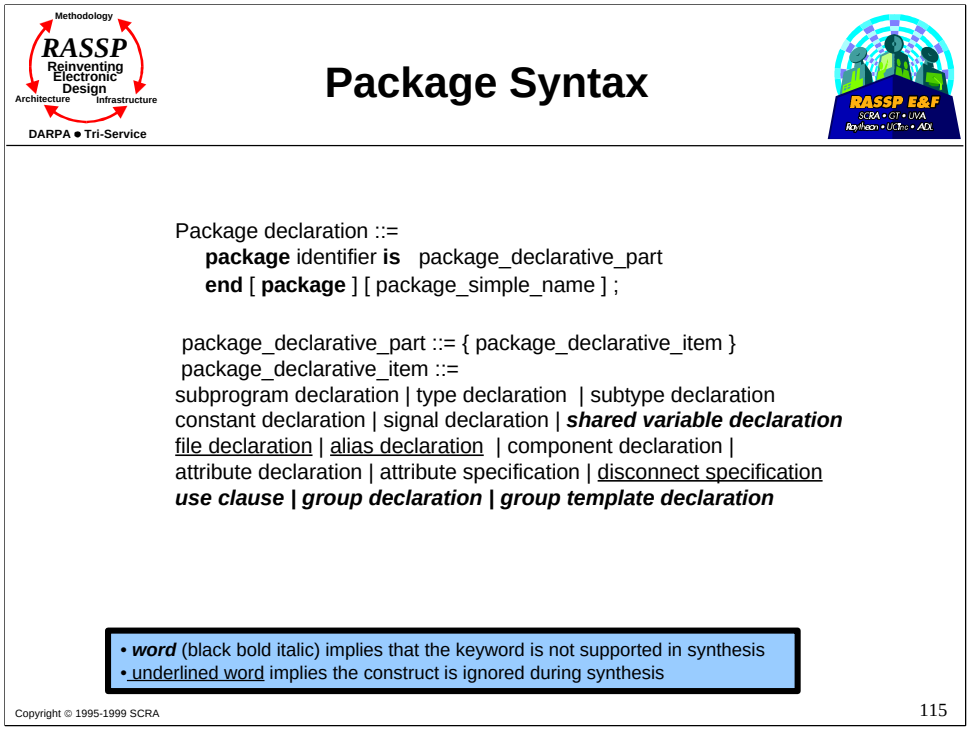
The architecture calls the majority function described in a package. The resulting synthesis is shown in the right part of the slide.

Using Procedures and Functions (Cont.)

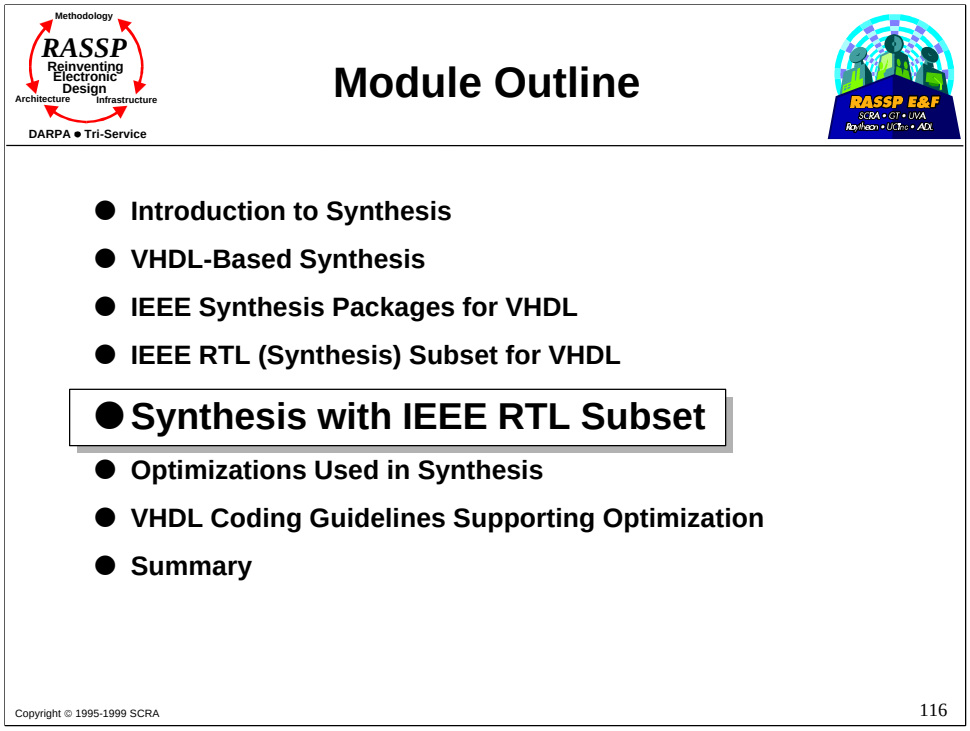
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE work.logic_package.all;

ENTITY decoder IS
  PORT(y : IN std_logic_vector(1 DOWNTO 0);
       g : OUT std_logic_vector(3 DOWNTO 0));
END decoder;

ARCHITECTURE dec OF decoder IS
  BEGIN
    comb : PROCESS(y)
      BEGIN
        decode(y,g);
    END PROCESS comb;
END dec;
```



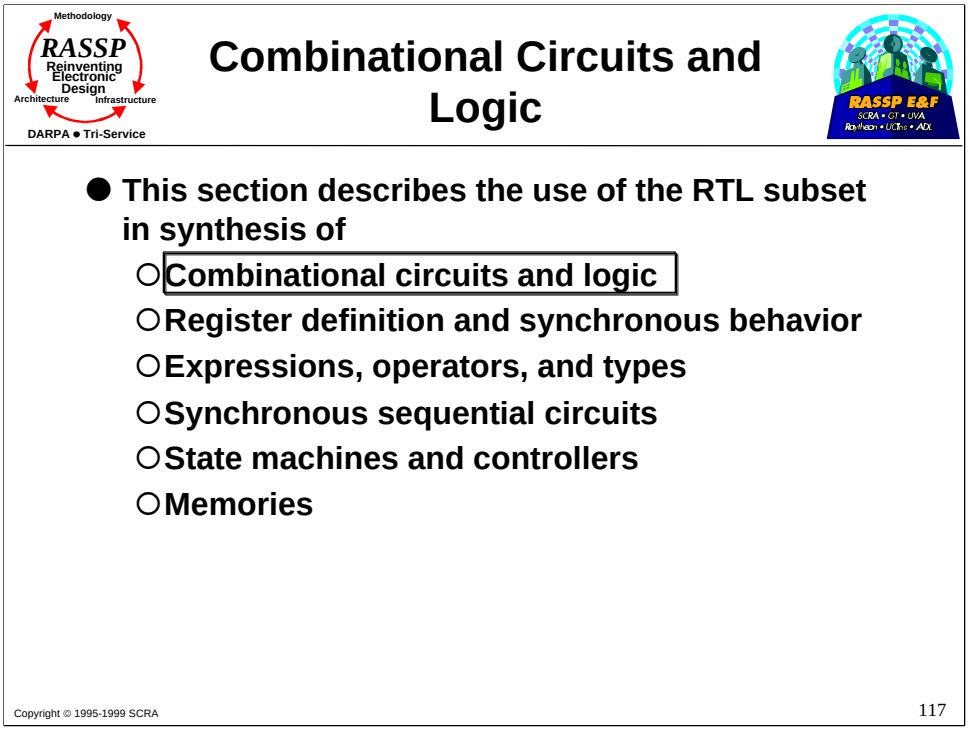
Similar example calling the decode functional block.



Note: signal and constant declarations shall have an initial value expression. Deferred constants are not allowed. Package bodies are similar with respect to the features permitted for the package.



We have learnt much about the RTL Synthesis subset syntax in the previous section, and now we will explore its use through some detailed examples.

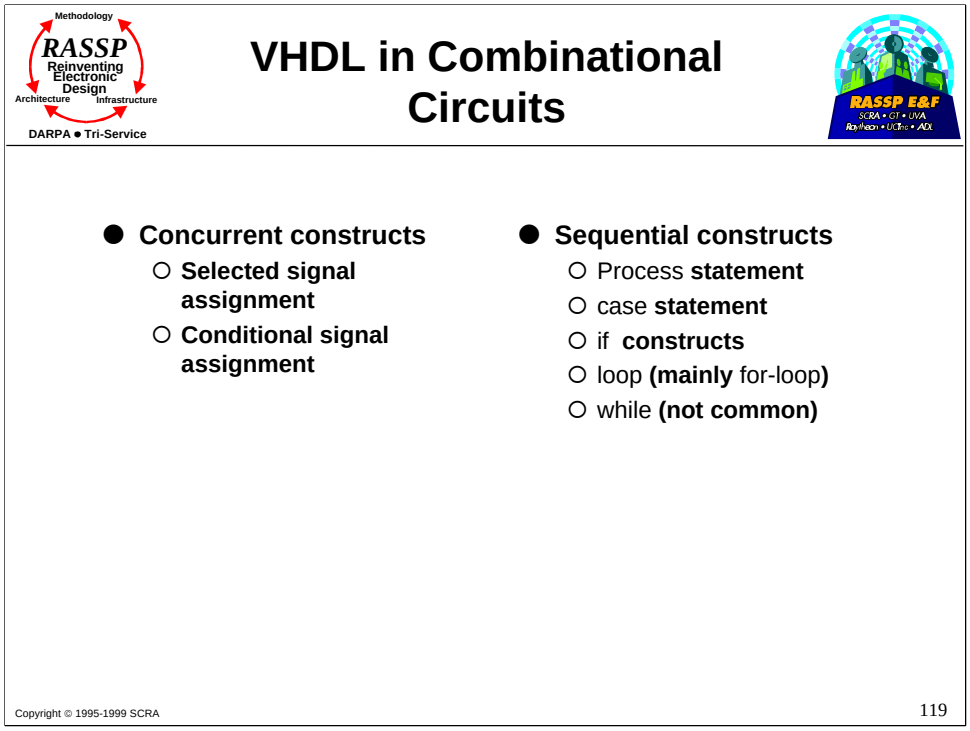


We will describe some examples of synthesis for each of the above classes of digital circuits.

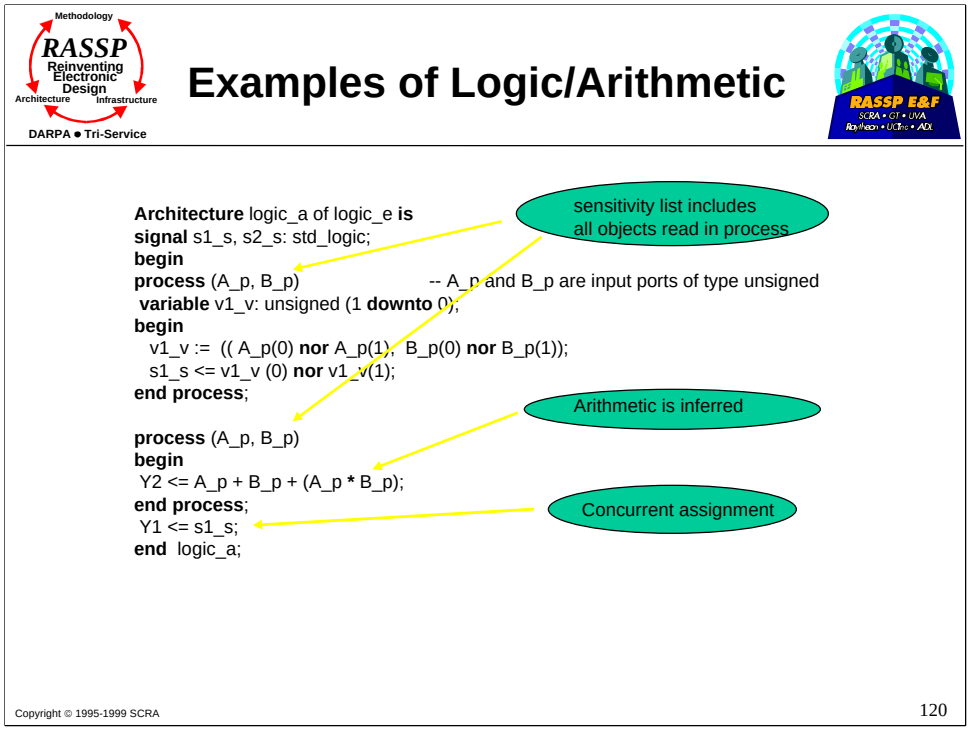
Combinational Circuits

- Combinational circuits include
  - logic/arithmetic
  - multiplexers and demultiplexers
  - encoders and decoders
  - comparators
  - Arithmetic Logic Units (ALUs)
- Both concurrent statements and sequential VHDL statements can be used to model combinational circuits as following examples demonstrate.

Combinational circuits can be realized in a number of ways in VHDL.



At the level of combinational logic there is little in the form of a logical pattern or structure that assists a designer, so optimizers rely on other techniques to optimize area, power, speed, etc. (e.g., Boolean optimizations, flattening..).



Note:

1. The absence of the clock signal.

2. The sensitivity list does not affect synthesis, but simulation will *differ* from synthesis if sensitivity_list guidelines are not followed (verification tough!).

Multiplexer/Demultiplexer

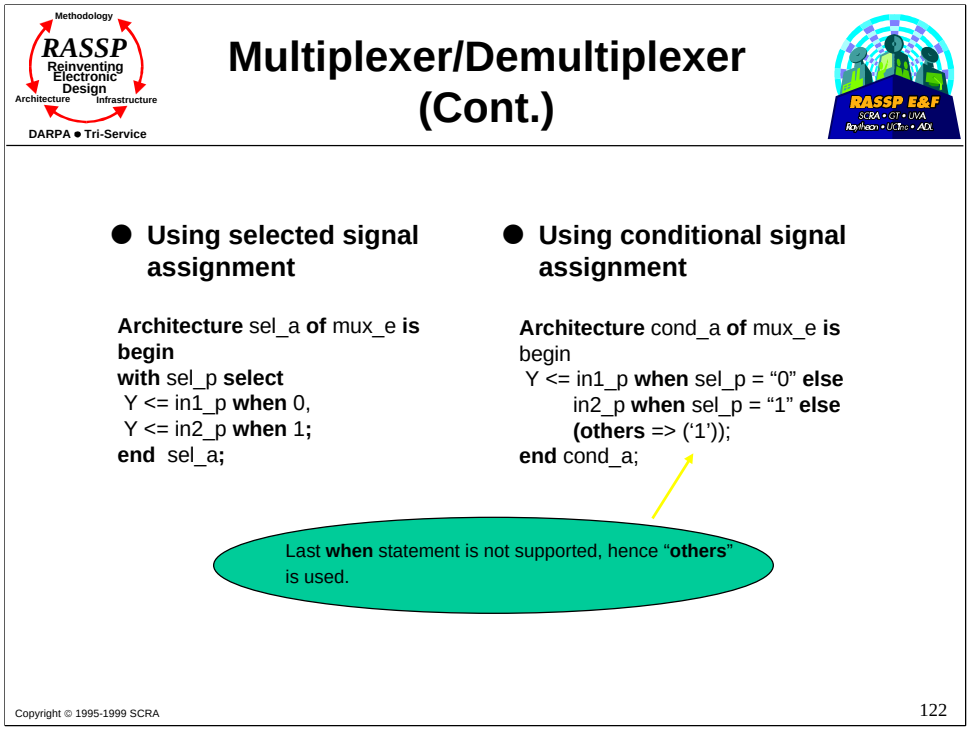
- **Using** if **statement**

```
Process (sel_p, in1_p, in2_p)
begin
if (sel_p =“0”) then
   Y_p <= in1_p;
 elsif (sel_p =“1”) then
   Y_p <= in2_p;
 end if;
end process;
```

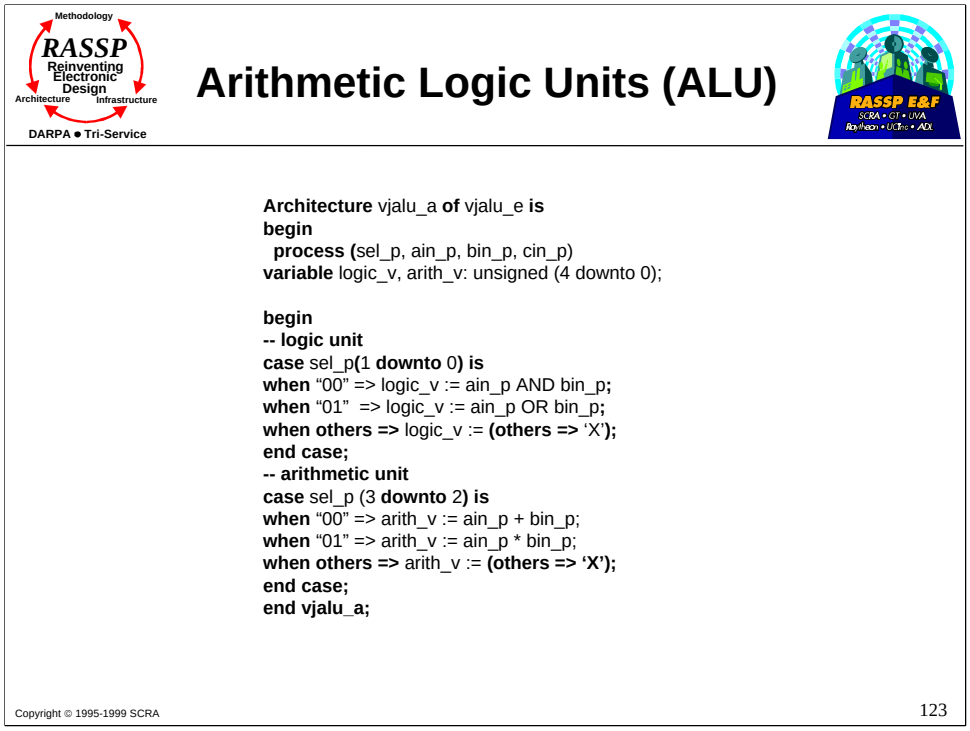
- **Using** case **statement**

```
Process (sel_p, in1_p, in2_p)
 case sel_p is
   when 0 => Y_p <= in1_p;
   when 1 => Y_p <= in2_p;
end case;
end process;
```


Note: Case statement is usually more readable, though both can be used within the context of the process.

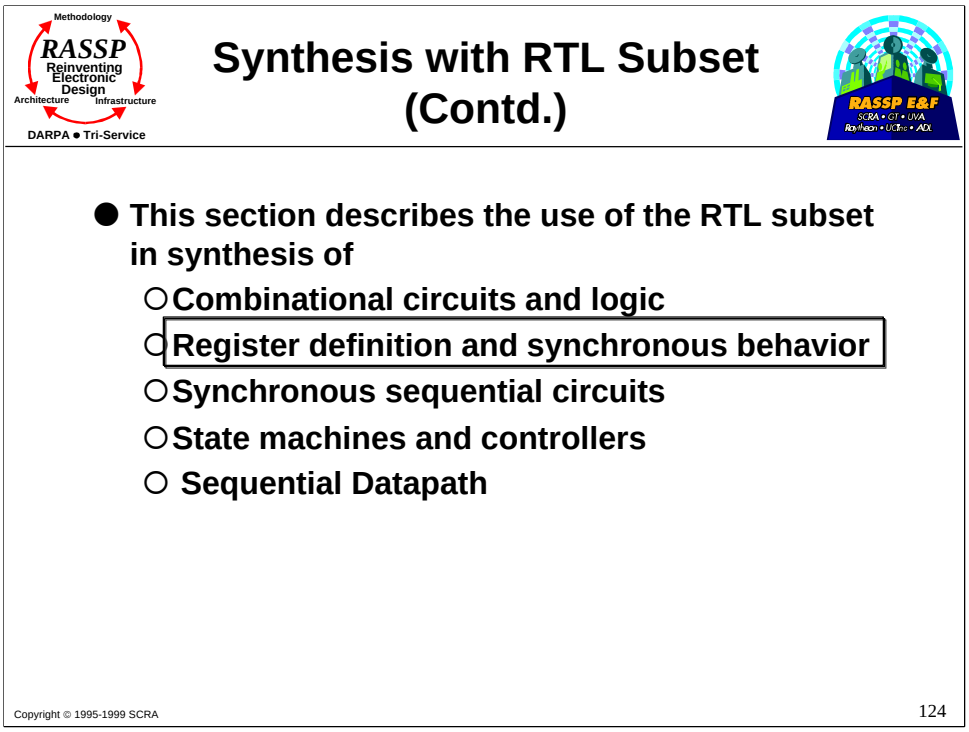


Note two equivalent realization of the mux/demux pairs.



Both floating and integer data path examples are presented later in the module.

This ALU describes a simpler data path supporting a number of logical operations.

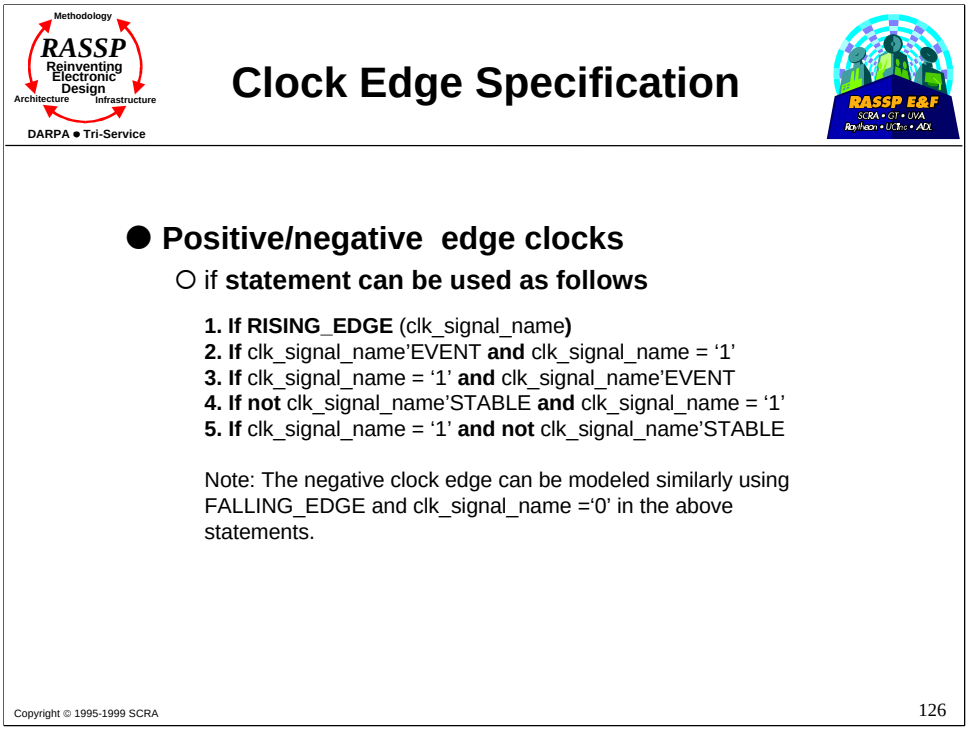


Register definition and expressing synchronous behavior is a very important function in the specification and design of digital circuits. We show how the RTL subset can be used in this phase of synthesis.

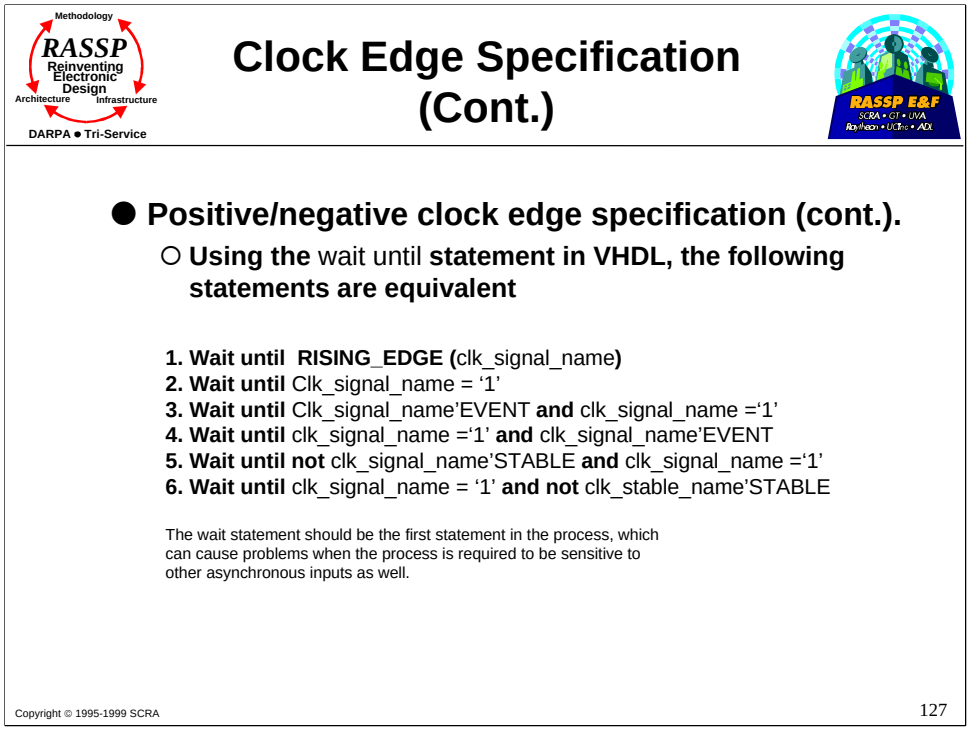
Registers and Clocks

- Clock types: **Allowed types for clock signals --** BIT, STD_ULOGIC **and their subtypes (**STD_LOGIC**), with a minimum subset of '0' and '1', without the metalogical values (**U, X, W, or "-"**).**
- **Scalar elements of vectors of above types are also allowed, e.g., bus8(0), as clocks.**
- **Clock *edges* can be specified either using the VHDL *functions* FALLING_EDGE and RISING_EDGE declared in** STD_LOGIC_1164 **or by** NUMERIC_BIT**, or through the use of** if **and** wait **statements (with some *restrictions*).**

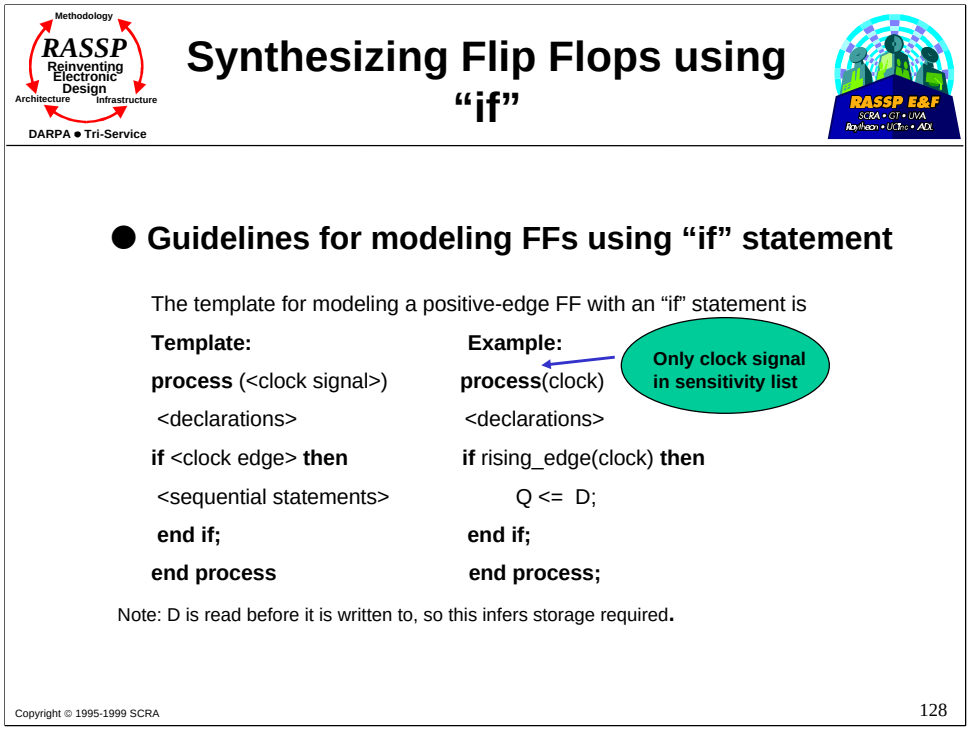
Clocks play an important role in synthesis, and the Synthesis subset has taken special efforts to ensure their correct use and interpretation in the synthesis as the following slides will show.



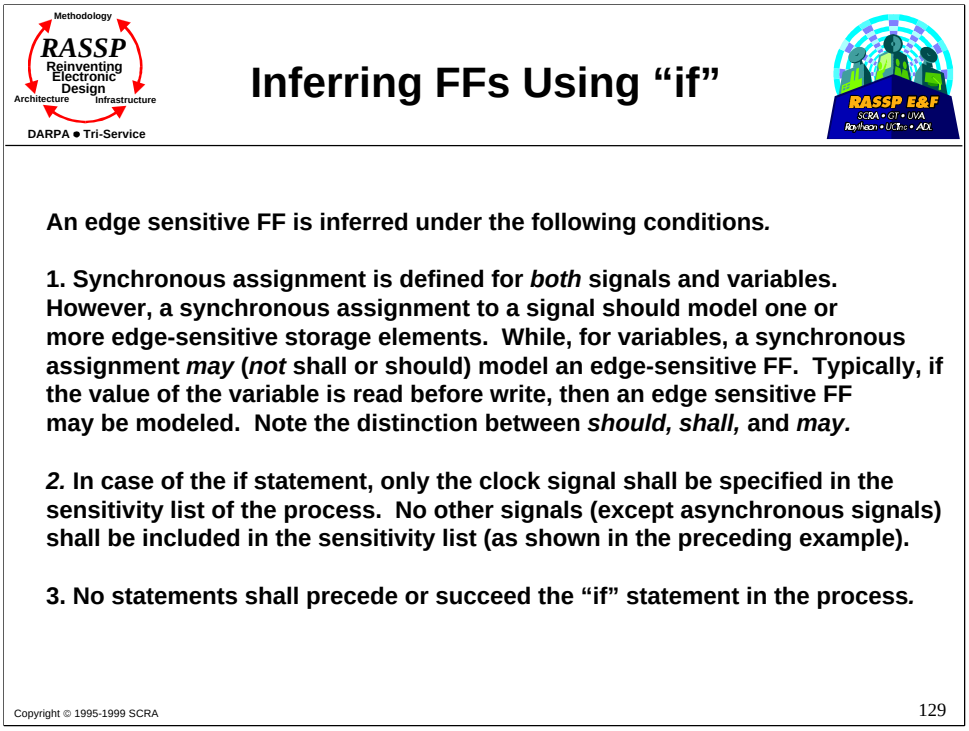
Clock edges (rising or falling) can be defined using the if-then-else construct or the wait-until construct in VHDL. The former is described in this slide in five different options. The if-then-else construct is preferred over the wait-until construct as will be clear from the next slide.



Note: Only *one* clock edge is allowed per process (though many processes can exist in an architecture**.** This does not mean that we are restricted to one of rising or falling edges, as *both* can be used, but only once in each process.



Note that flip flops and latches are usually inferred when an object is read before it is written to, and also in cases when some memory is inferred.



Both signals and variables can result in flip flops depending on the VHDL description of the function. Note that a synchronous process contains only one "if" statement, and no other statement.

Inferring FFs Using“wait”

The wait until statement can be used with the following conditions:

1. Only one **wait** statement is allowed per process, and it shall be the first statement in the process. This implies that asynchronous conditions cannot be modeled in the same process using the wait until construct.

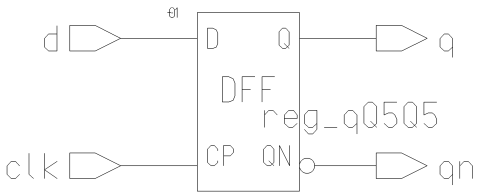
```
Label:  Process  -- note labels are allowed for processes
begin
wait until CLOCK = ‘0’;
COUNT := COUNT + 1; -- COUNT may model edge-sensitive FF
 VAR := COUNT
 VAR := VAR + 1; -- VAR is written to before it is read - no FF
 Q <= D;  -- Q also infers FF
end process;
```


As mentioned earlier, the wait statement is restrictive when used for synthesis purposes.

Edge Sensitive D Flip-Flop

- **Clocks must be of BIT or STD_LOGIC type - metalogical values not allowed**
- **Rising_edge() and Falling_edge() functions can be used to specify clock edge**

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY d_ff is
  PORT(d   : IN  std_logic;
       clk : IN  std_logic;
       q   : INOUT std_logic;
       qn  : OUT std_logic);
END d_ff;

ARCHITECTURE behavior OF d_ff IS
  BEGIN
    seq : PROCESS(clk)
      BEGIN
        IF(rising_edge(clk)) THEN
          q <= d;
        END IF;
    END PROCESS seq;
    qn <= not q;
END behavior;
```



131

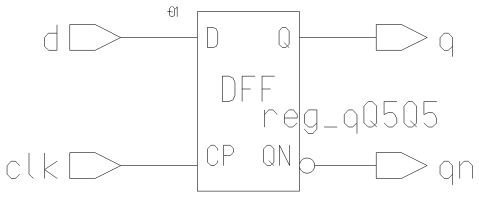
Note the inferring of a Flip Flop because "d" is not assigned for all cases of input. This implies that the previous value of "d" has to be stored (in a FF).

Edge Sensitive D Flip-Flop (Cont.)

- **Wait statement can be used**

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY d_ff_w is
  PORT(d   : IN  std_logic;
       clk : IN  std_logic;
       q   : INOUT std_logic;
       qn  : OUT std_logic);
END d_ff_w;

ARCHITECTURE behavior OF d_ff_w IS
  BEGIN
    seq : PROCESS
      BEGIN
        WAIT UNTIL rising_edge(clk);
        q <= d;
    END PROCESS seq;
    qn <= not q;
END behavior;
```

```
ARCHITECTURE behavior OF d_ff_w IS
  BEGIN
    seq : PROCESS
      BEGIN
        WAIT UNTIL clk = '1';
        q <= d;
    END PROCESS seq;
    qn <= not q;
END behavior;
```



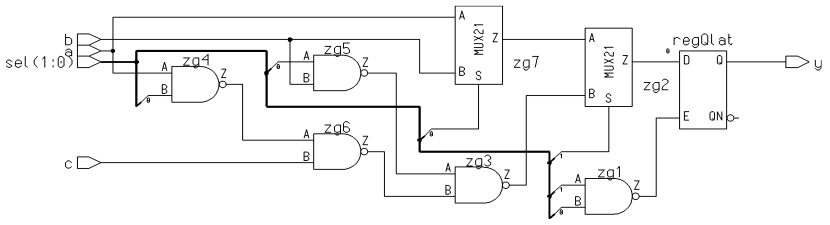
This slide shows how a wait statement can be used to assert synchronous behavior in a circuit.

Inferring Latches

- **If signals or variables are not assigned values in some conditional expressions of `IF` or `Case` statements, level-sensitive sequential logic might result**

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY mux3_seq is
  PORT(a   : IN  std_logic;
       b   : IN  std_logic;
       c   : IN  std_logic;
       sel : IN std_logic_vector(1 DOWNTO 0);
       y   : OUT std_logic);
END mux3_seq;
```

```
ARCHITECTURE behavior OF mux3_seq IS
  BEGIN
    comb : PROCESS(a,b,c,sel)
      BEGIN
        CASE sel IS
          WHEN "00" => y <= a;
          WHEN "01" => y <= b;
          WHEN "10" => y <= c;
          WHEN OTHERS => --empty
        END CASE;
    END PROCESS comb;
END behavior;
```

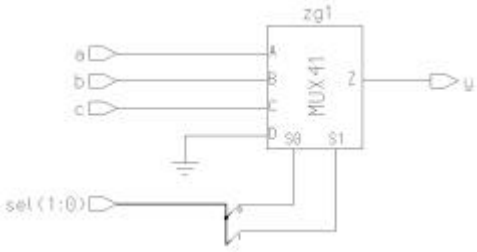


No clock edge is included in the design resulting in a latch being inferred. Storage is implied but no edge sensitivity is declared.

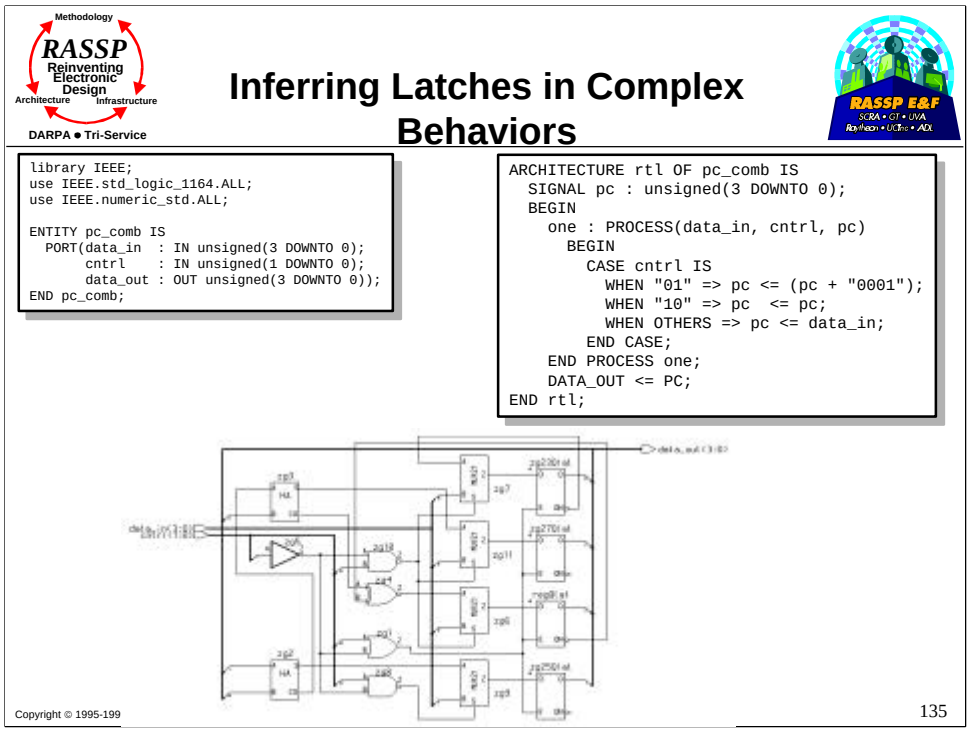
Avoiding Latches

- **Assigning values of “don’t care” (‘X’ or ‘-’) in these cases can avoid this**

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY mux3 is
  PORT(a   : IN  std_logic;
       b   : IN  std_logic;
       c   : IN  std_logic;
       sel : IN std_logic_vector(1 DOWNTO 0);
       y   : OUT std_logic);
END mux3;

ARCHITECTURE behavior OF mux3 IS

  BEGIN
    comb : PROCESS(a,b,c,sel)
      BEGIN
        CASE sel IS
          WHEN "00" => y <= a;
          WHEN "01" => y <= b;
          WHEN "10" => y <= c;
          WHEN OTHERS => y <= 'X';
        END CASE;
    END PROCESS comb;
END behavior;
```



We avoid a latch here (and substitute it with a mux) by enumerating all possibilities for the CASE statement. No storage is necessary as a consequence.



A latch is inferred as it is not clear to the synthesizer whether it has to store the value of the PC. In general, synthesis tools are not very good and understanding code behavior over a large number of statements.

Synthesizing Asynchronous Signals

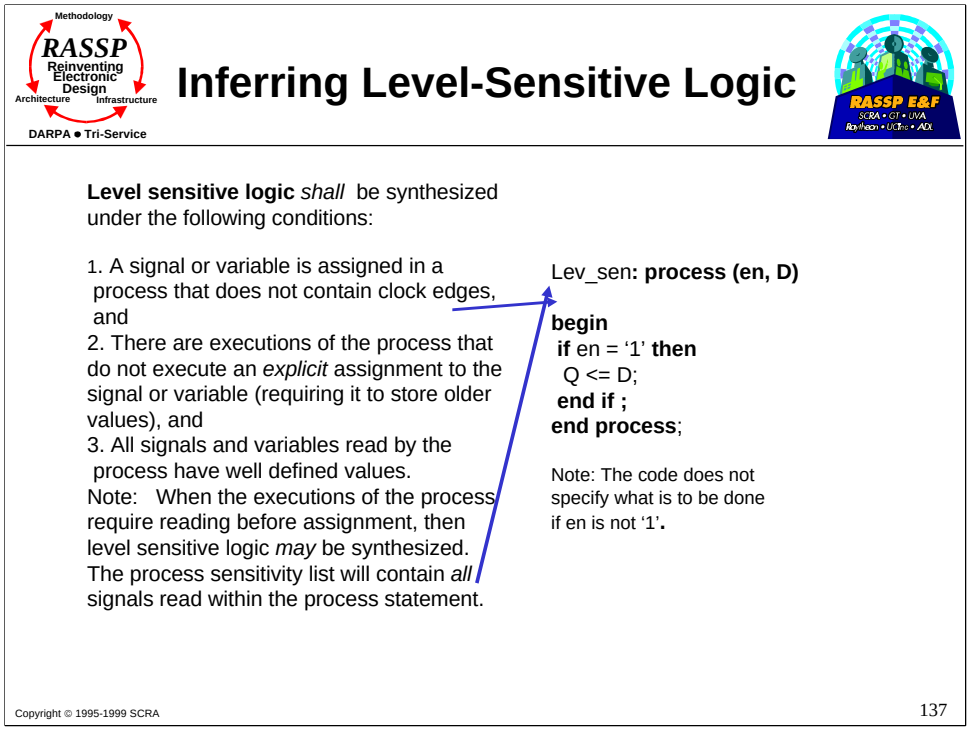
Asynchronous signals such as SET/RESET are frequently included in synchronous circuits. The **wait until** construct does not support this feature, however the **if** construct is useful in including these signals in the following format.

```
Async_dff: process (clock,  reset, set)
 begin
 if reset = '1' then Q <= '0';
 elsif set = '1' then  Q <= '1';
 elsif clock'event and clock = '1' then  Q <= D;
 end if ;
end process;
```


Note:

1. It is difficult to include more than one wait statement in a process designed for synthesis, so asynchronous inputs would require an if-then-else structure.

2. All the asynchronous signals and the clock signals are included in the sensitivity list of the process. In the combinational process, even A would have been included in the sensitivity list, and D would have been written before being read. All asynchronous signals are level sensitive and cannot contain clock expressions**.**

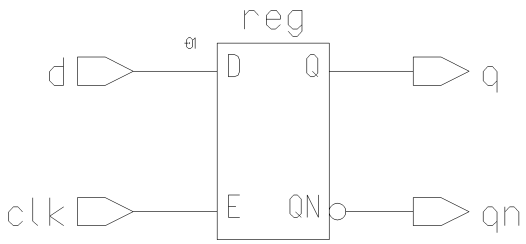


The example shows that some behavior is unspecified, and level-sensitive logic (e.g., latches) are inferred.

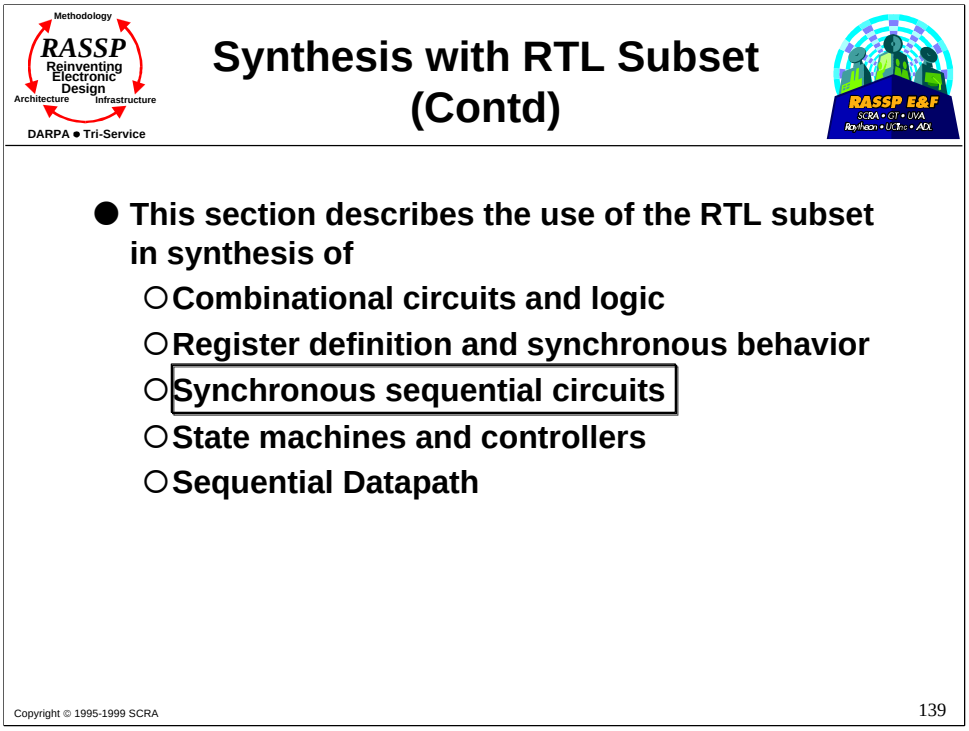
Level Sensitive D Latch

- **Use `IF` statement without `Else` clause**

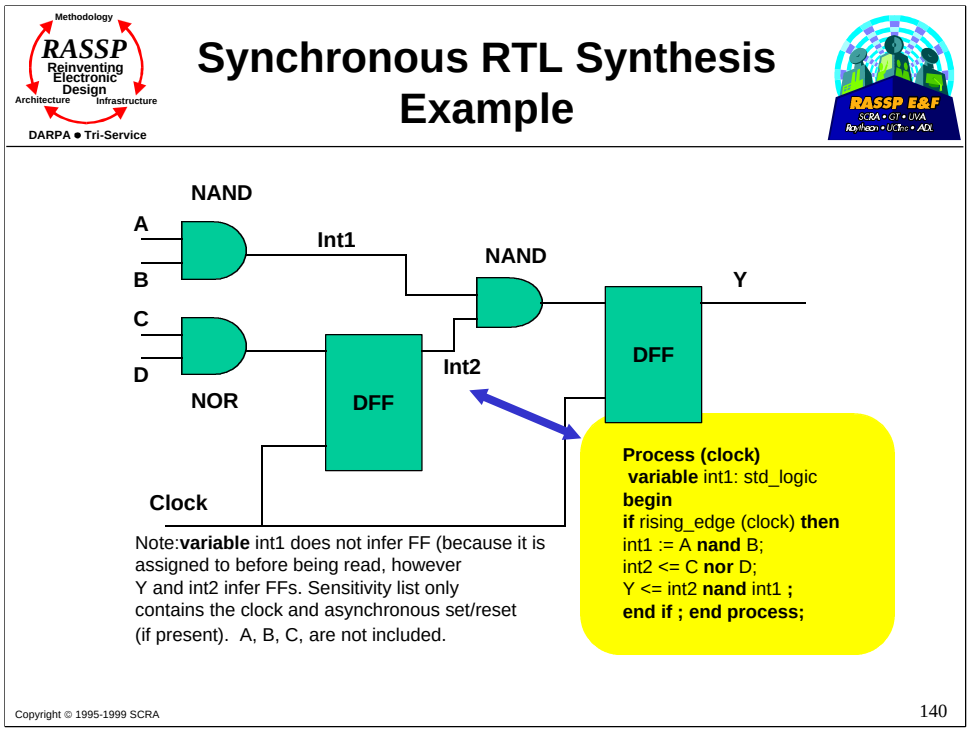
```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY d_latch is
  PORT(d   : IN  std_logic;
       clk : IN  std_logic;
       q   : INOUT std_logic;
       qn  : OUT std_logic);
END d_latch;

ARCHITECTURE behavior OF d_latch IS
  BEGIN
    seq : PROCESS(d,clk)
      BEGIN
        IF(clk = '1') THEN
          q <= d;
        END IF;
    END PROCESS seq;
    qn <= not q;
END behavior;
```



The if statement without the else results in a latch because the complete behavior is not specified, and storage is inferred.

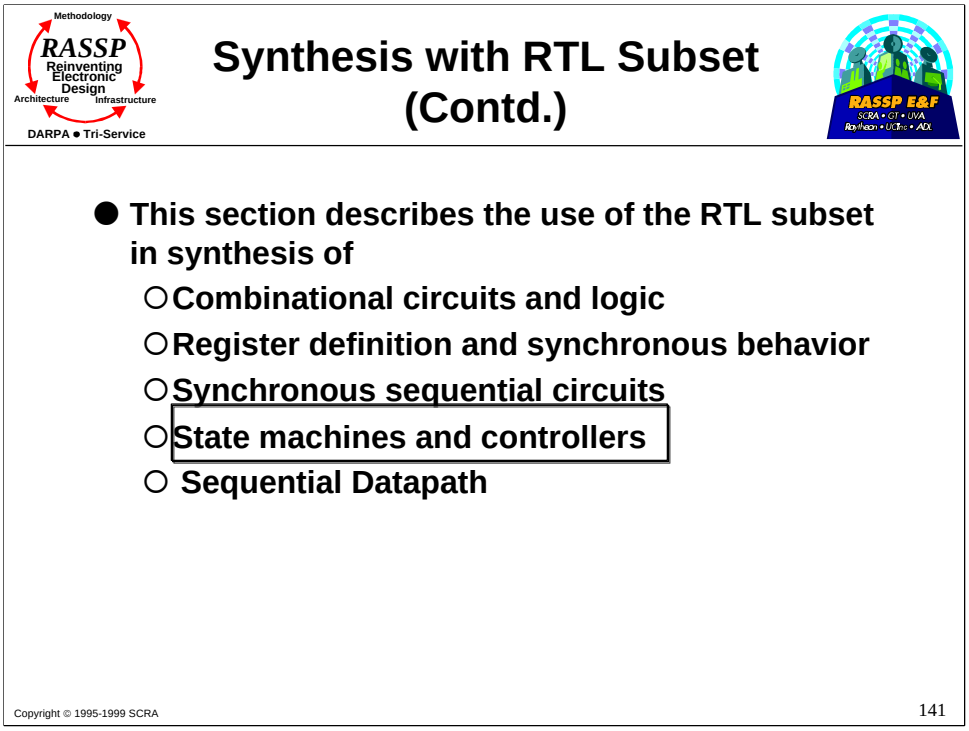


We now describe an example of a synchronous sequential circuit and its synthesis in VHDL.

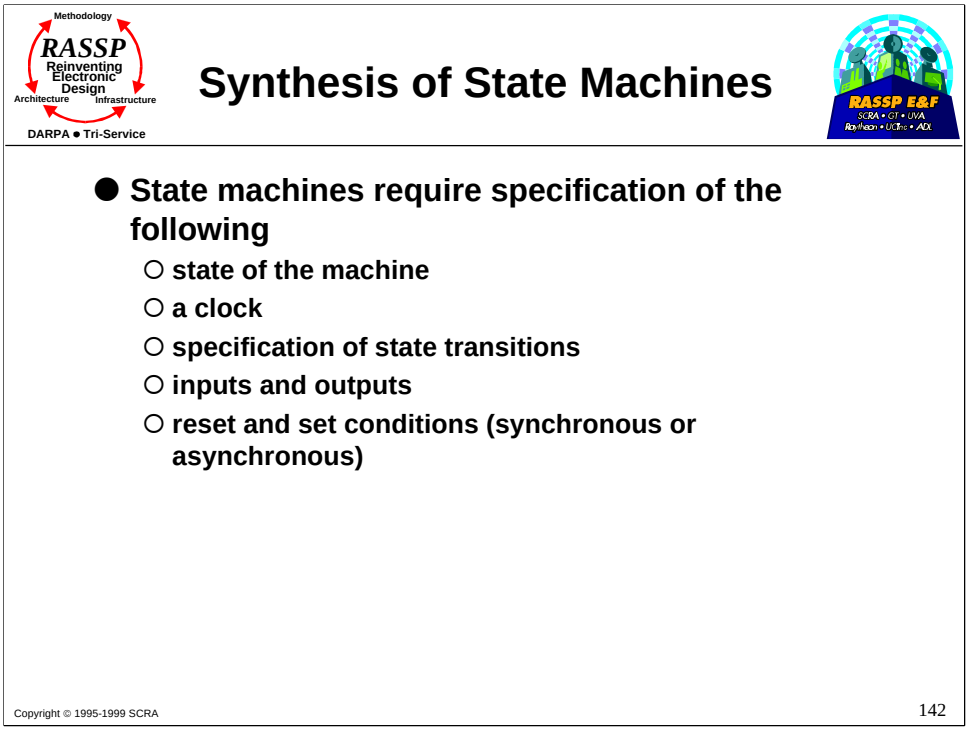


The FF's are realized because the clock signal is included in the sensitivity list. If we wanted to realize combinational logic, we should have included all signals (A, B, C, D) in the sensitivity list and not mentioned the clock signals.

Note that variables can also result in flip-flops if they are read before they are assigned. Thus, in the most general case, variables and signals may infer latches.



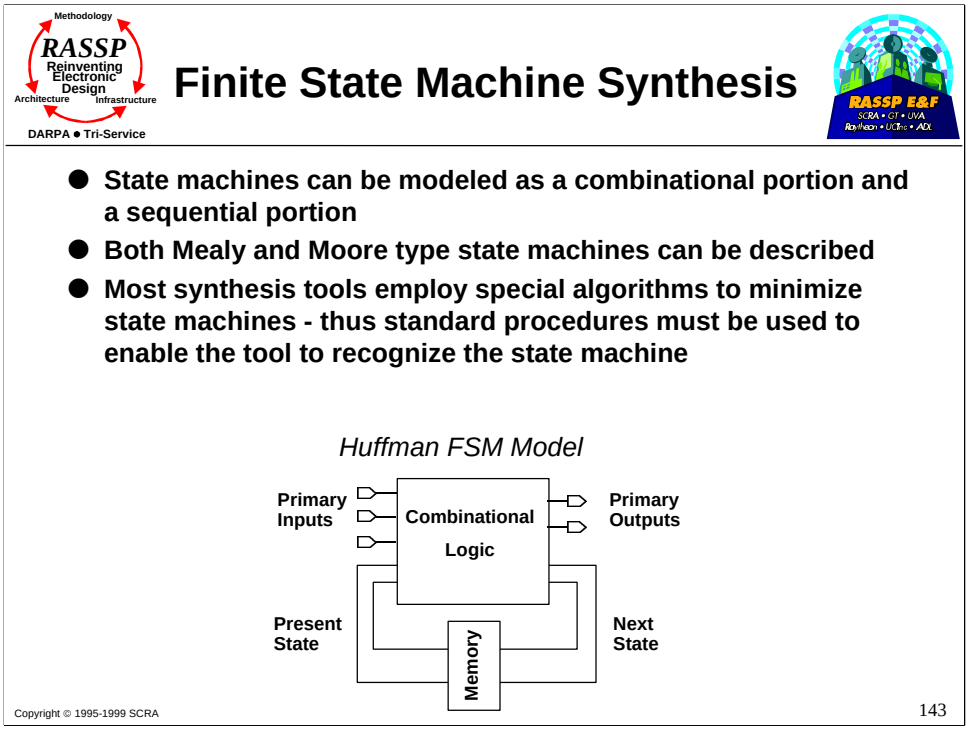
We will now describe the use of the RTL subset in the synthesis of finite state machines (controllers).



Note: Many synthesis tools have optimized the synthesis of state machines provided certain templates are followed. Examples of optimizations include one-hot, random, gray, binary, Mustang, Spectral, and Nova to name just a few. The IEEE RTL standard does not specify such a template, so we provide an example of a popular coding style (Synopsys) for state machine synthesis that is consistent with the IEEE RTL level 1 standard.

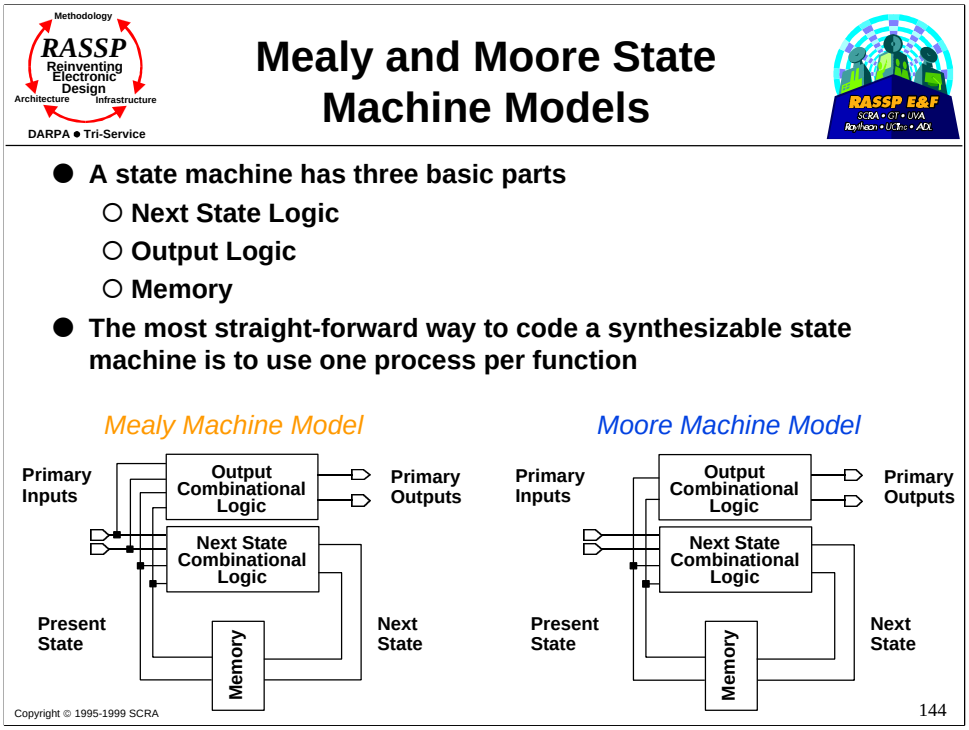
Finite State Machines (FSMs) are widely used to represent designs of controllers, and avionics industry surveys have shown that FSMs with about 20 - 40 states are typical of the design complexities of ASICs

FSMs can be of the Moore (output depends on current state alone) and Mealy (output depends on input and current state) types. State encoding is also used to optimize FSM synthesis and has been incorporated into the IEEE RTL standard.



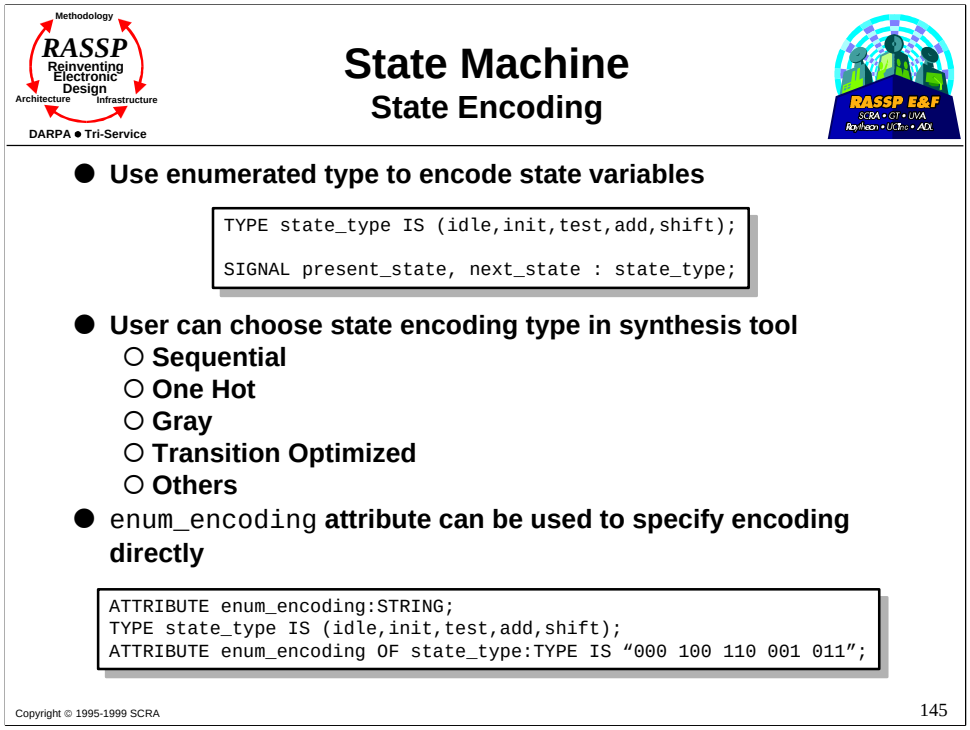
Next, the description of synthesizable state machines will be covered. Synthesis tools typically have special algorithms built in for state machine minimization, so “templates” must be used to ensure that the tool recognizes the state machine and the state variables. Synthesis tools can construct both Mealy and Moore state machines.

Considering the traditional Huffman model, state machines are comprised of a combinational portion and a memory portion.



The memory portion of the state machine updates the present state of the state machine with the next state on a clock edge.

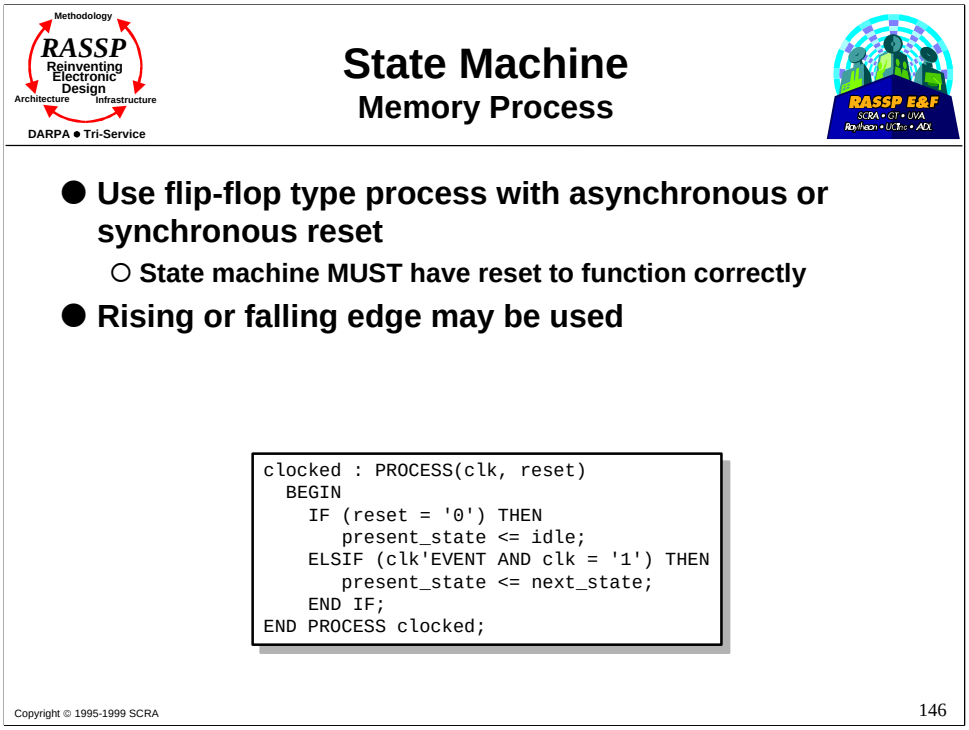
The combinational portion of the state machine actually performs two functions. First, it calculates the next state based on the present state and the inputs to the state machine. Second it calculates the outputs of the state machine based on the present state and the inputs (Mealy machine) or only on the present state (Moore machine).



Normally, the state variables, present state and next state, are defined to be of some enumerated type. Using an enumerated type has two benefits, first, it facilitates easier debugging in simulating the behavioral VHDL description before synthesis as the value of the state variables are easily seen during simulation. Second, the use of an enumerated type allows different encodings for the state variable to be generated by the tool during synthesis.

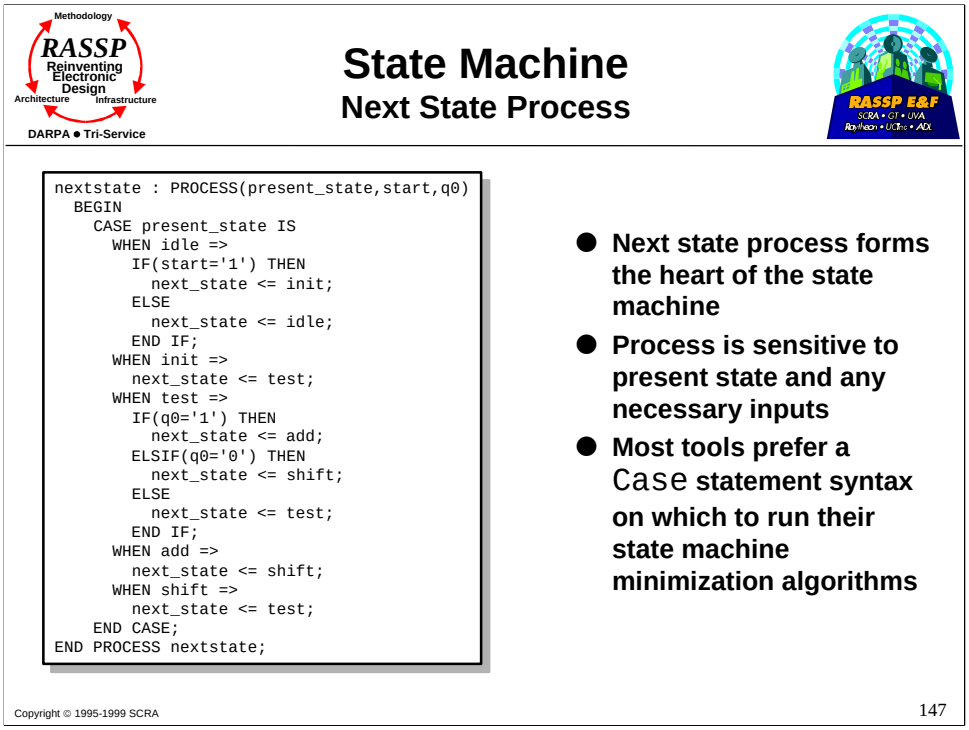
Various possible encodings such as one hot, sequential or gray encoding can be used. Several synthesis tools have the ability to to pick “optimum” encodings based on minimizing the number of state variables that change when transitioning to adjacent states.

In addition, the enum_encoding attribute can be defined for the state variable as shown and the synthesis tool will use the specified encodings.



For the memory process, a rising edge or falling edge flip flop should be used. A reset function, either synchronous, or asynchronous MUST be included to ensure that the state machine can be brought to a known state.

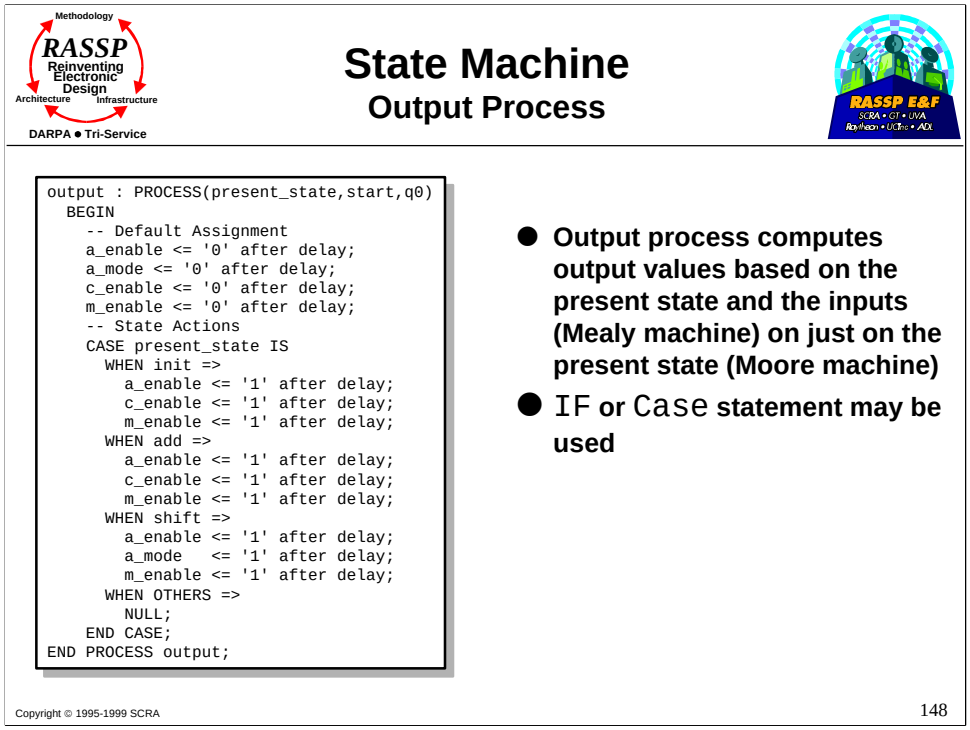
Note that during behavioral simulations before synthesis, the state machine will appear to simulate correctly, even without a reset function, because the state variable will default to the lowest value of the enumerated type (idle in the present example). Post synthesis simulation will show however, that the state variables will remain “unknown” unless a reset function is used.



The next state process is the heart of the state machine and calculates the value of the next state based on the present state and the inputs to the state machine. The next state process should be sensitive to the present state and the state machine's inputs.

Most synthesis tools prefer that the next state calculation be based on a CASE statement. Be sure that values of next state are calculated for all possible values of the present state or latches will be inferred in the combinational next state process.

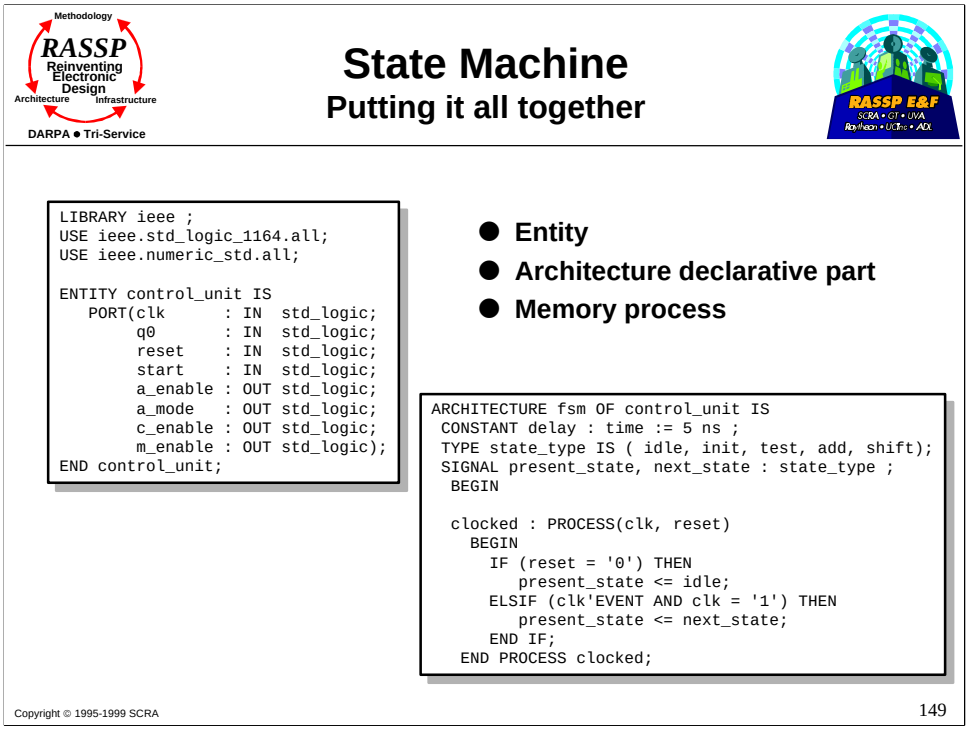
When synthesizing a state machine, most tools will report that they have recognized the state variables and will list the encodings, in terms of bits, that have been given to each literal (state such as idle, add, shift, etc.) in the enumerated type. These encodings should be noted as they are useful in debugging the state machine after synthesis.



The output process should be sensitive to the the present state only (Moore machines) or the present state and the inputs (Mealy machines).

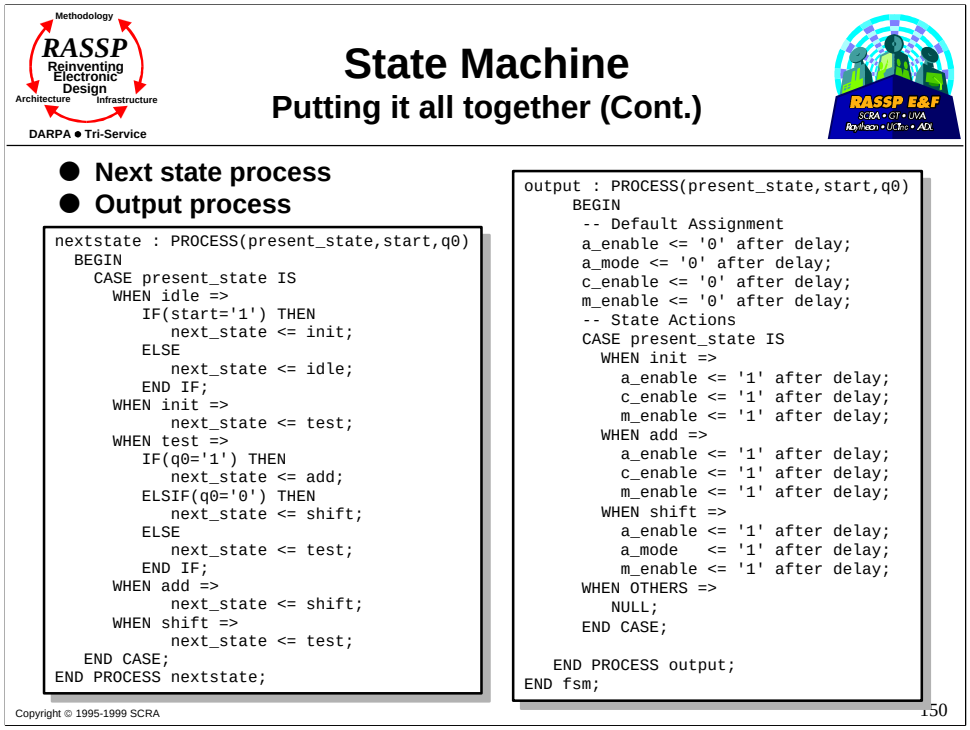
A CASE statement or an IF statement can be used to calculate the output values.

Note here that there is a default assignment for the outputs that takes effect if the outputs are not assigned a value in the CASE statement. Also note that the “idle” state is not specifically listed as a condition clause in the case statement. It is covered by the OTHERS condition and in that case, the default values will be set for all outputs.



Here we will show the entire synthesizable description of the state machine and the results of synthesis. This slide shows the entity description, the architecture declarative part, and the memory process. Notice that the architecture declarative process includes the enumerated type declaration for the state variables, the declaration of the present_state and next_state variables, and a declaration of a constant delay of type time. The delay constant will be used to delay the assignment of the outputs after the clock cycle to make the simulation easier to interpret as previously discussed.

The memory process includes an asynchronous reset function.

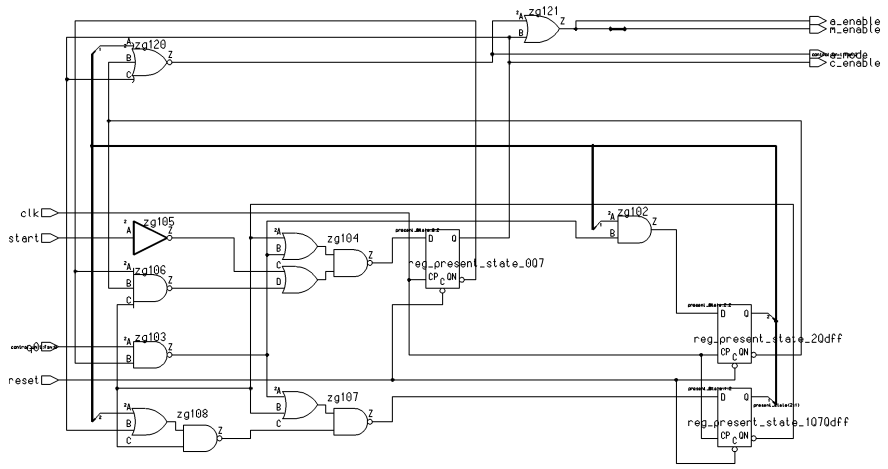


This is the next state process and output process.

Methodology
RASSP
Reinventing Electronic Design
Architecture
Infrastructure
DARPA ● Tri-Service

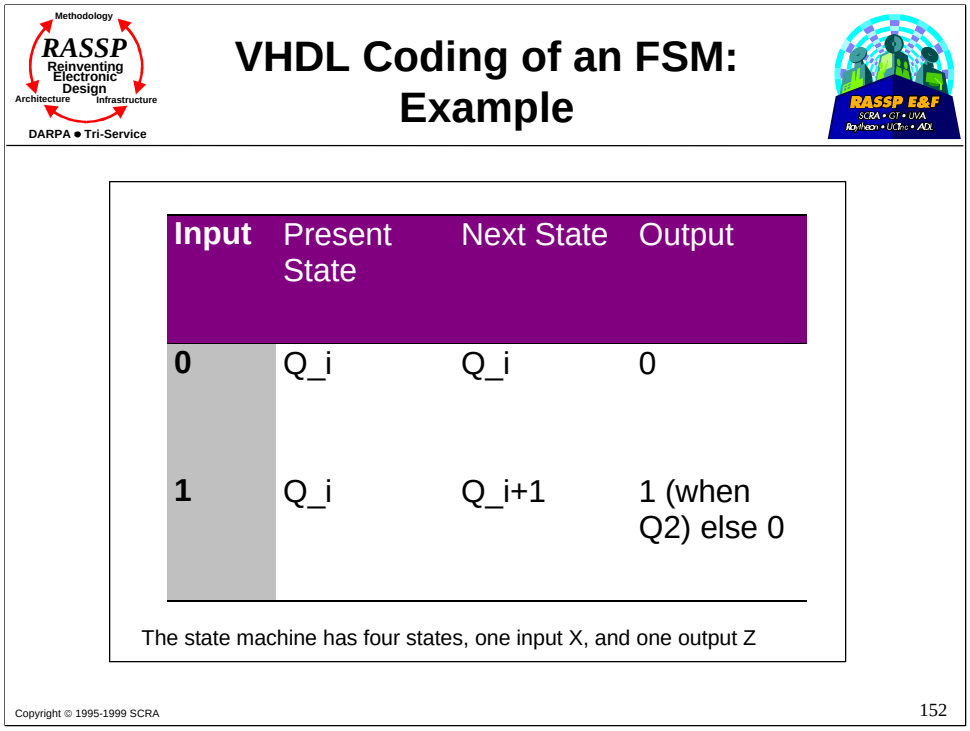
State Machine
Results

RASSP E&F
SCRA • GT • UVA
Raytheon • UCB • ADL

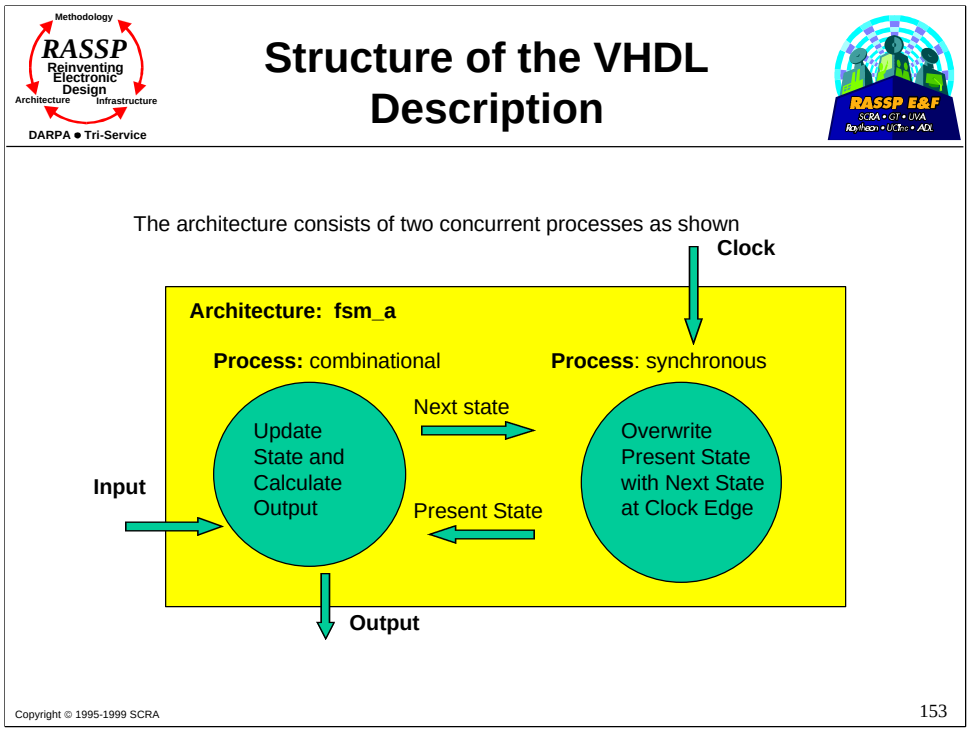


This is the result of the state machine synthesis. The flip flops for the state variables (3) can clearly be seen.

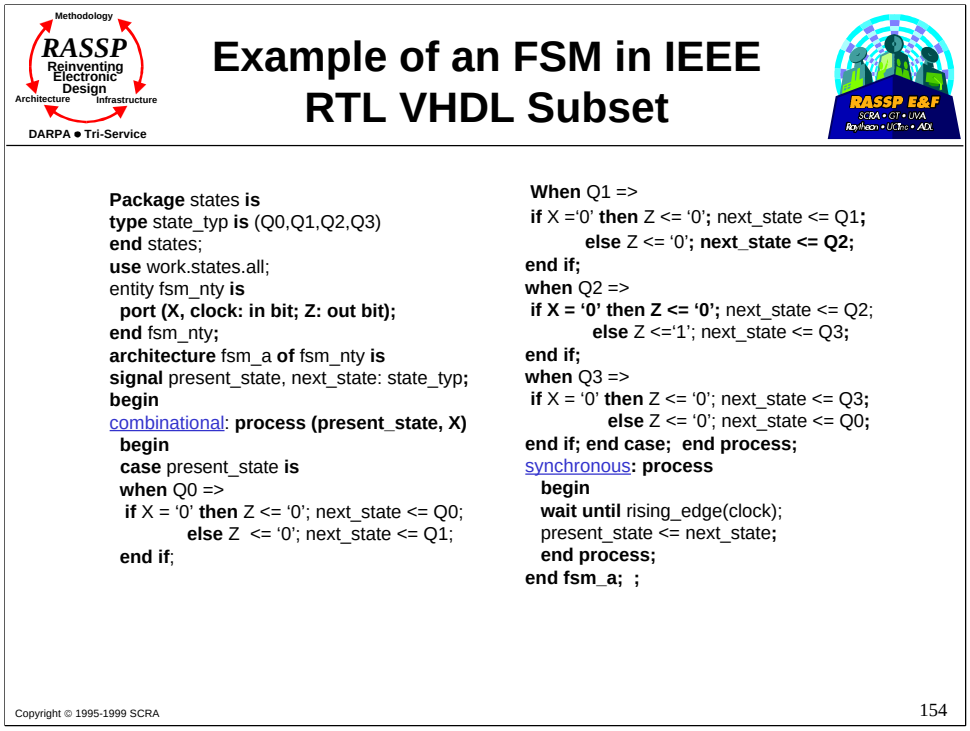
In the following, we will look at another example of FSM synthesis starting with the state table specification.



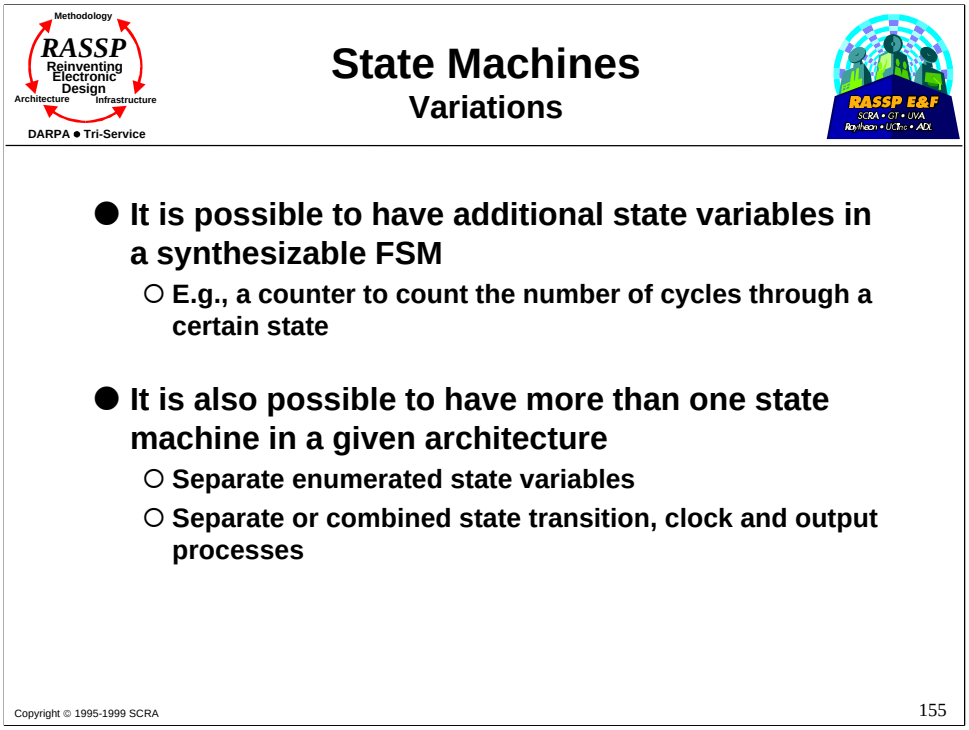
This is a simple transition diagram that we'll code in VHDL as a synthesizable template for a finite state machine.



FSMs are best synthesized through the use of templates of the architecture, such as shown in this slide, where a multi-process structure is followed. The clock-related dependencies are captured in the synchronous process, while the state update is calculated in the combinational process.

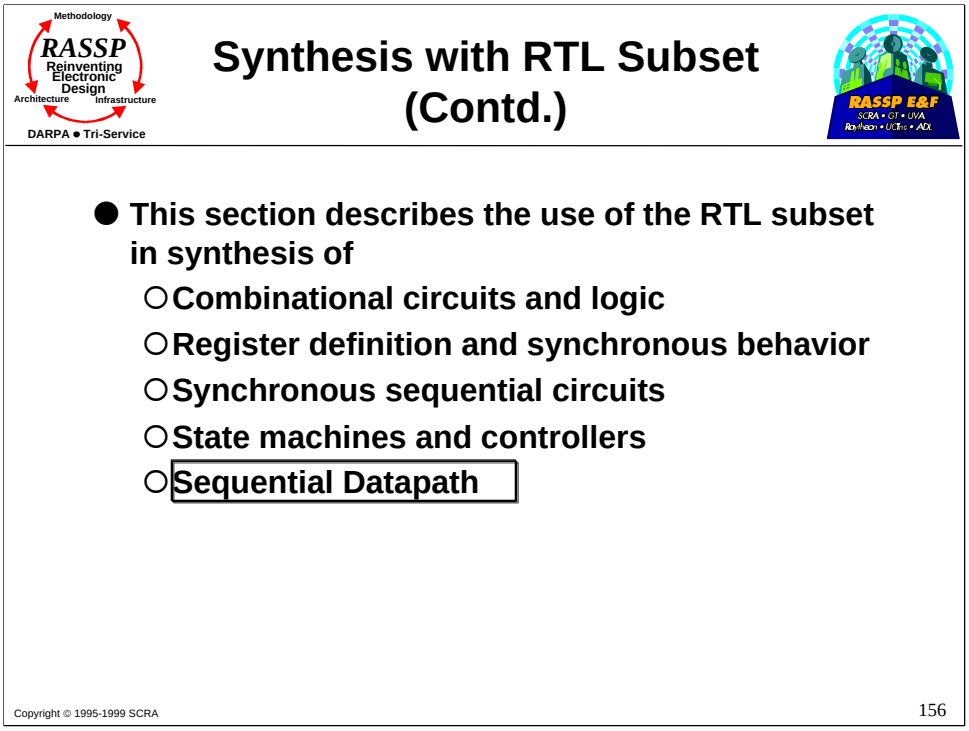


The reader may note the partitioning of the state calculation and the state update. Reset/Set may also be included within the two processes.

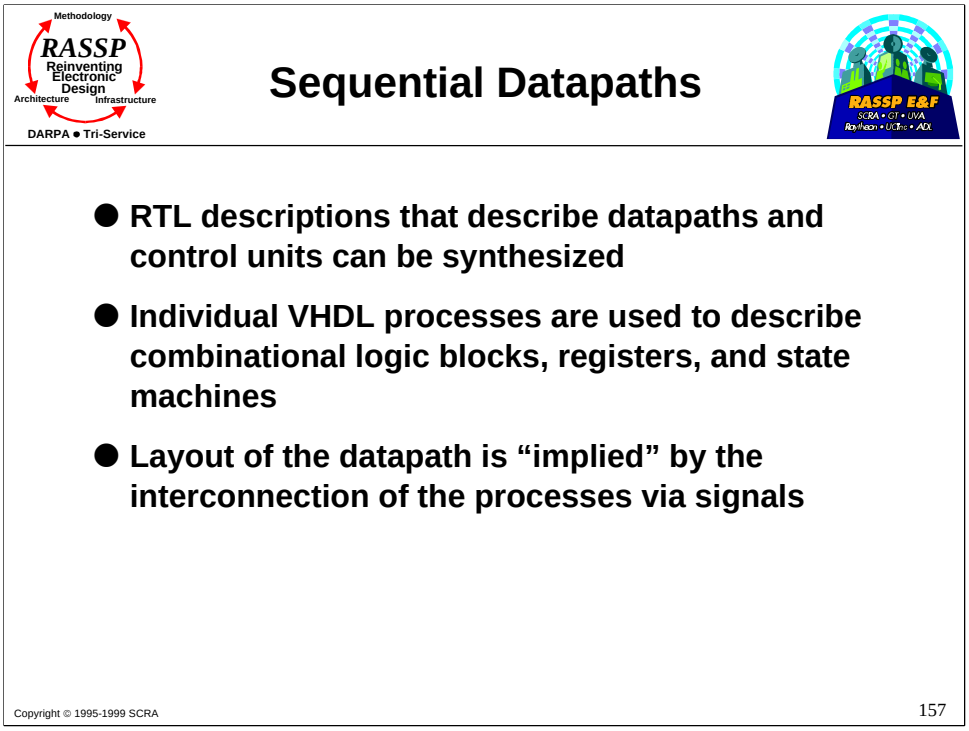


It is possible to have additional state variables in a state machine. An example is a counter that will count the number of transitions through a given state.

It is also possible to have more than one state machine in a given architecture.



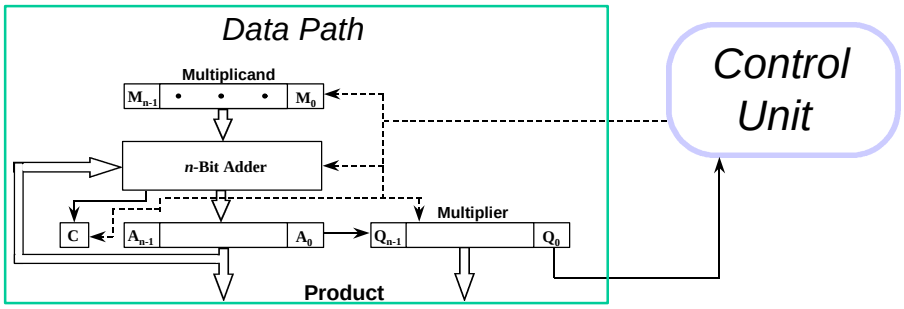
Datapath elements, such as the multiplier, adder, divider, constitute an important function in digital circuits, and we describe a synthesizable example of a multiplier in the following slides.



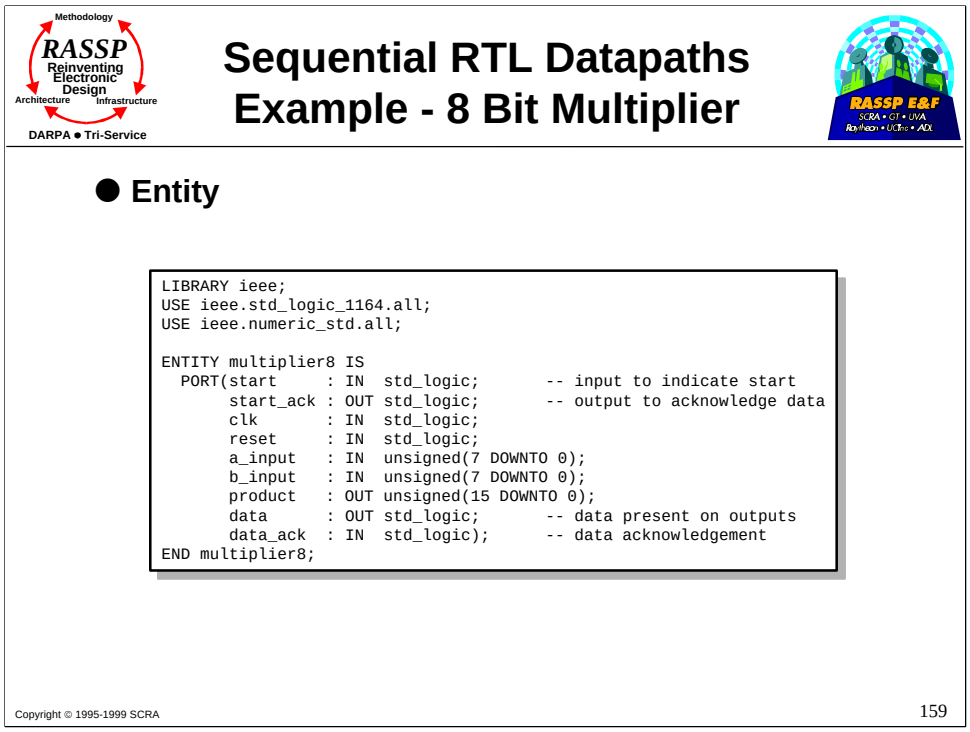
We will now describe a more complex example - a multiplier using some of the VHDL syntax we have learned so far.

Sequential (RTL) Datapaths
Example - 8 Bit Multiplier

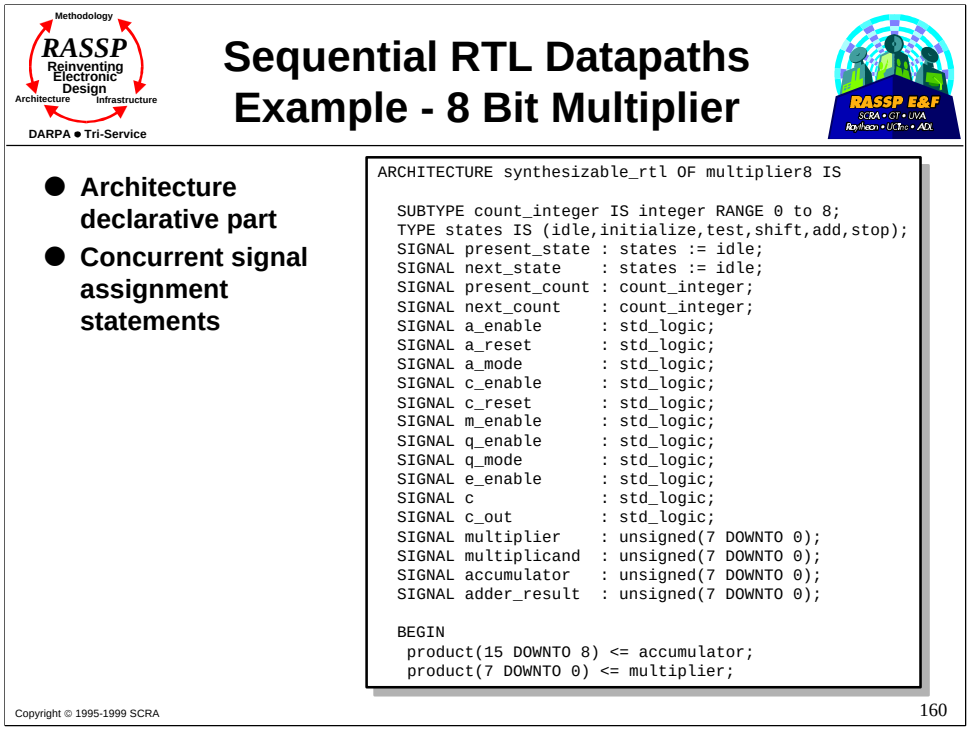
- **Unsigned 8 bit multiplier - datapath and control unit are similar to the example from the Structural and Behavioral VHDL modules**
- **Separate processes are used to model the M, A, Q, and C registers, the adder, and the output, memory, and next state portions of the control unit**



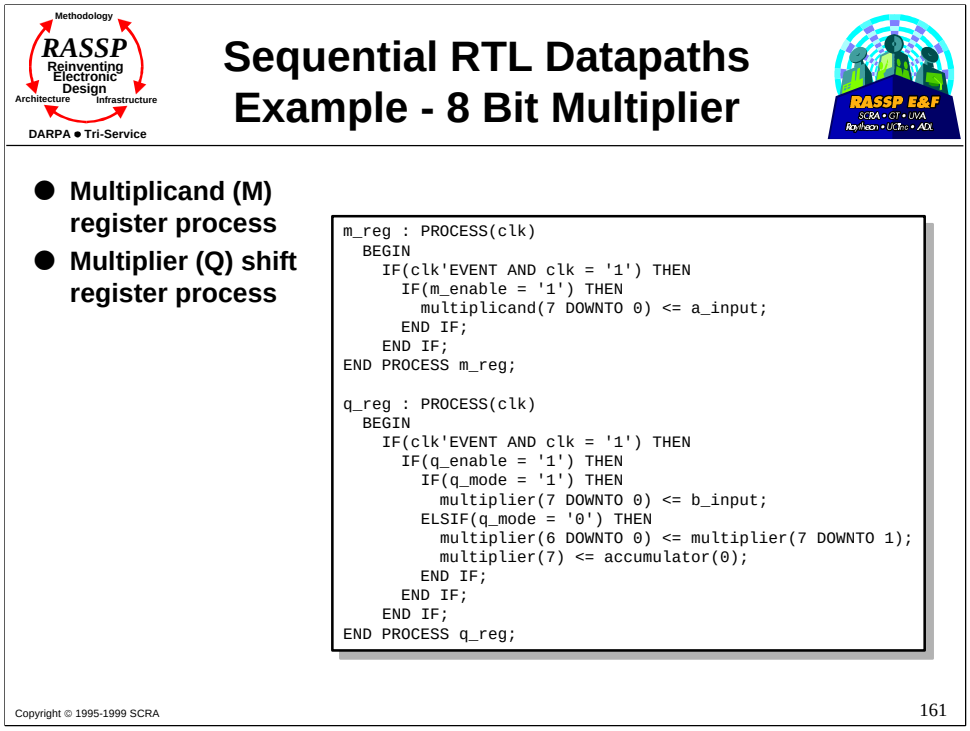
This slide describes the operation of an unsigned 8-bit multiplier.



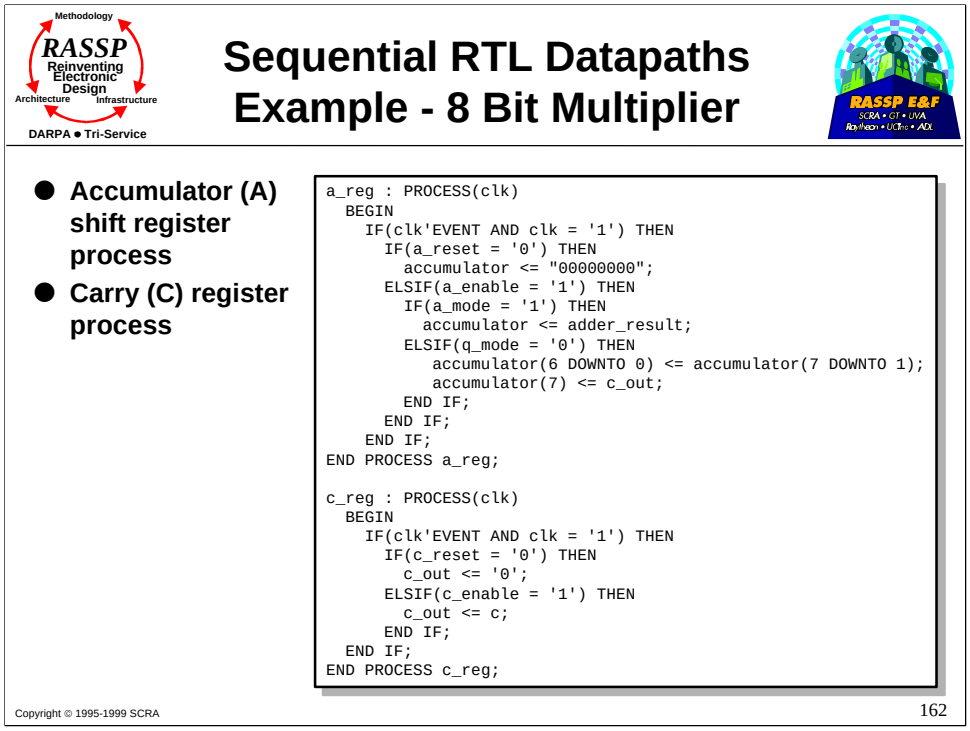
The entity represents the input/output ports of the multiplier.



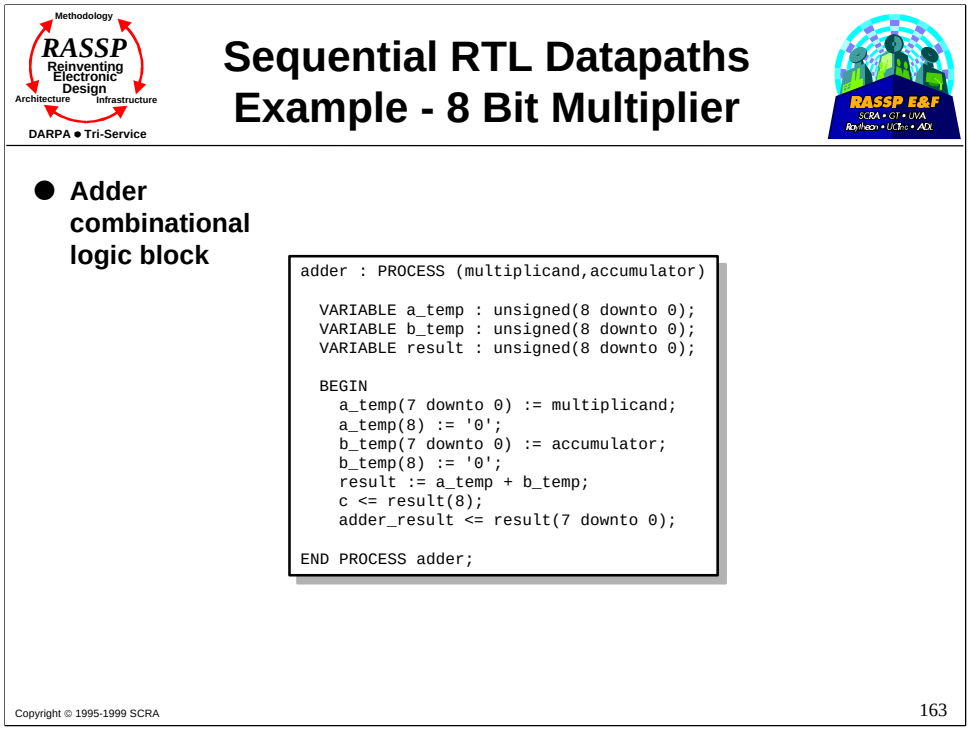
The signals are declared in this slide.



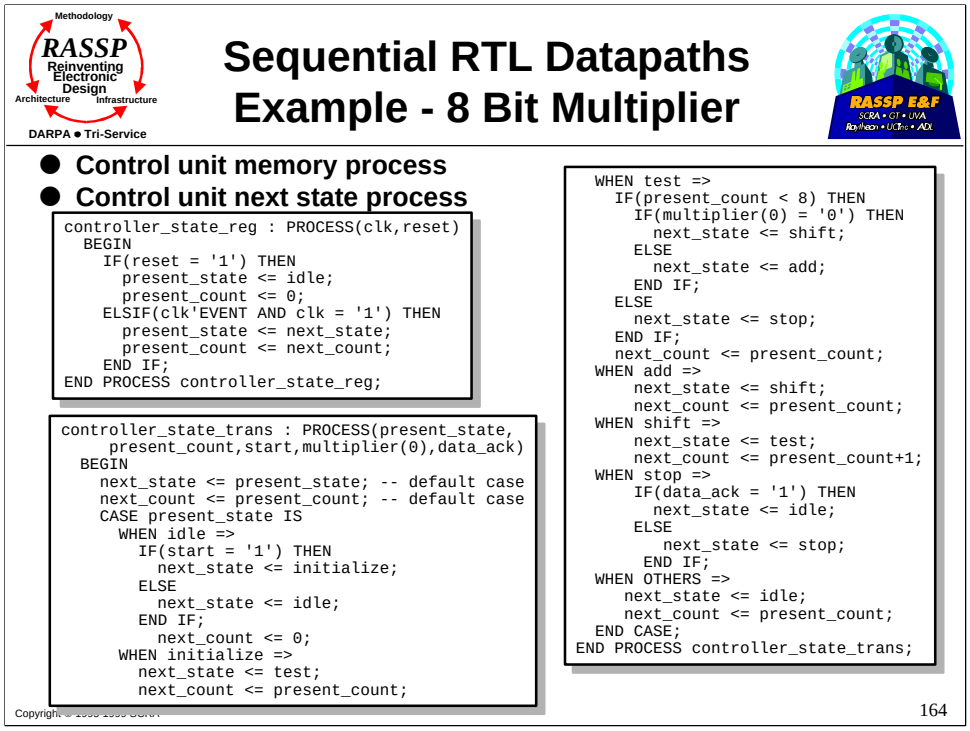
The loading of the input registers (to the multiplier) is described in this slide.



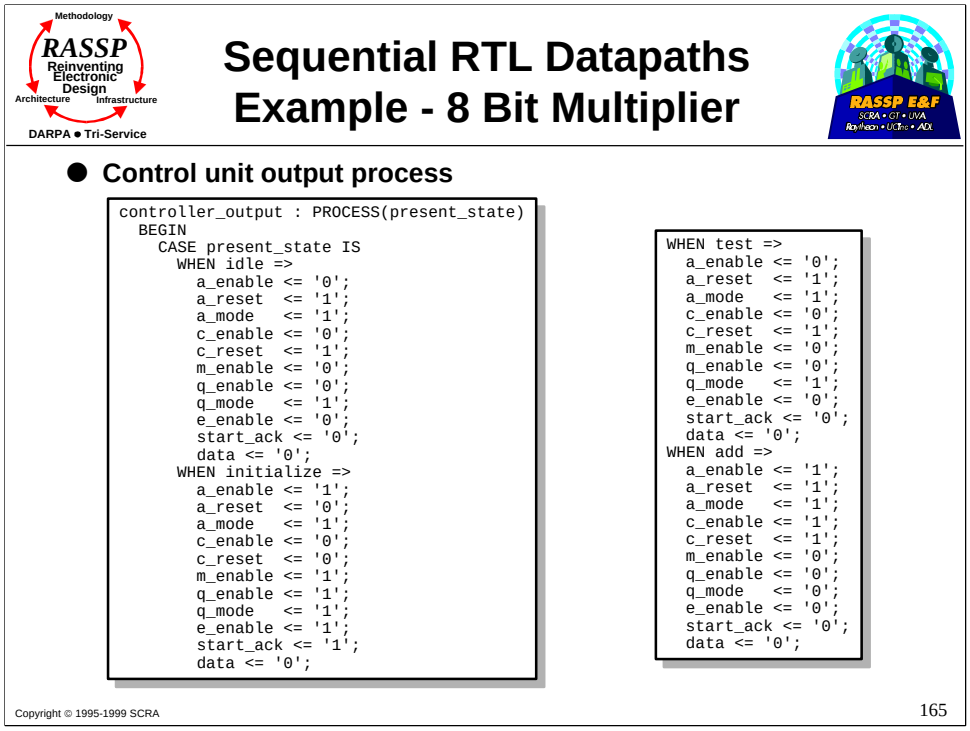
The accumulator and carry register processes are described in this slide.



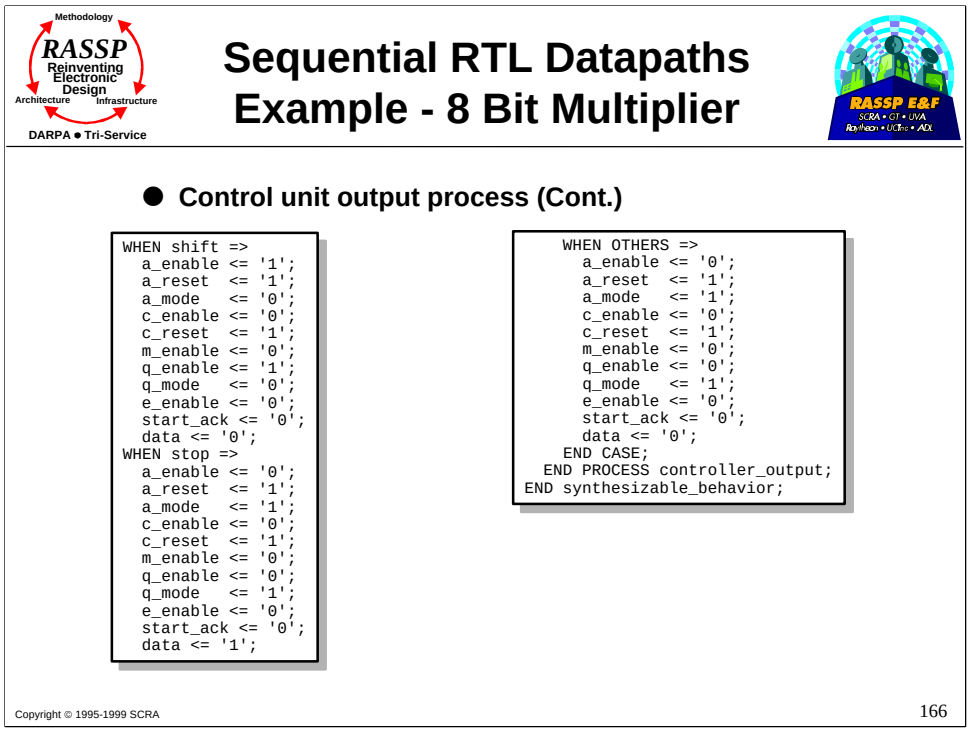
The combinational activity within the multiplier is described in this slide.



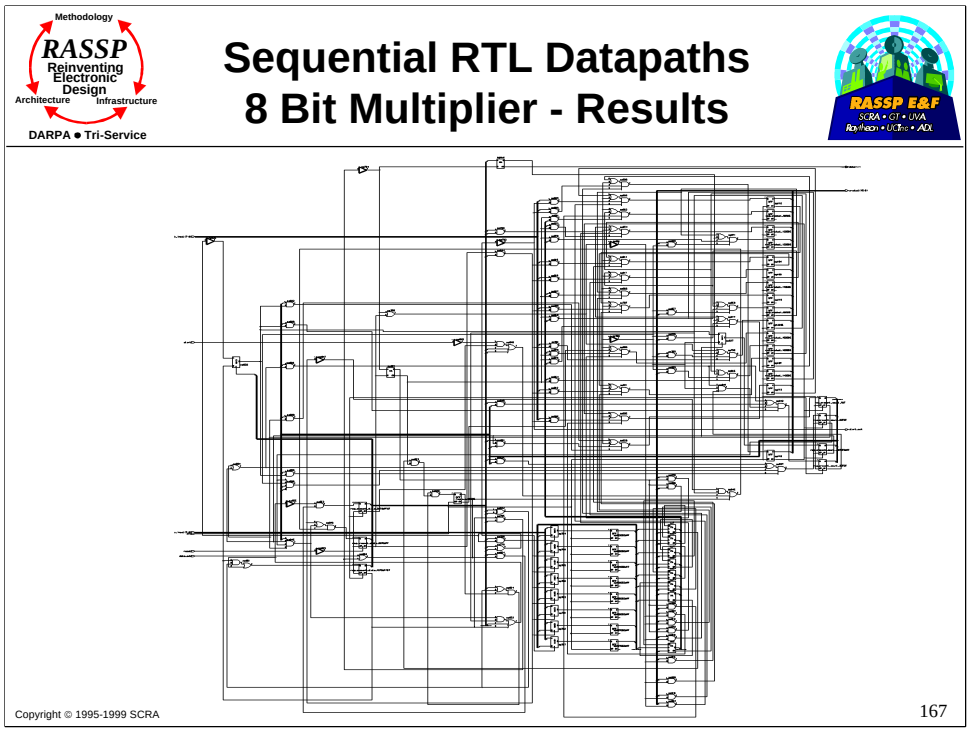
The multiplier sequencer and controller are described in this slide.



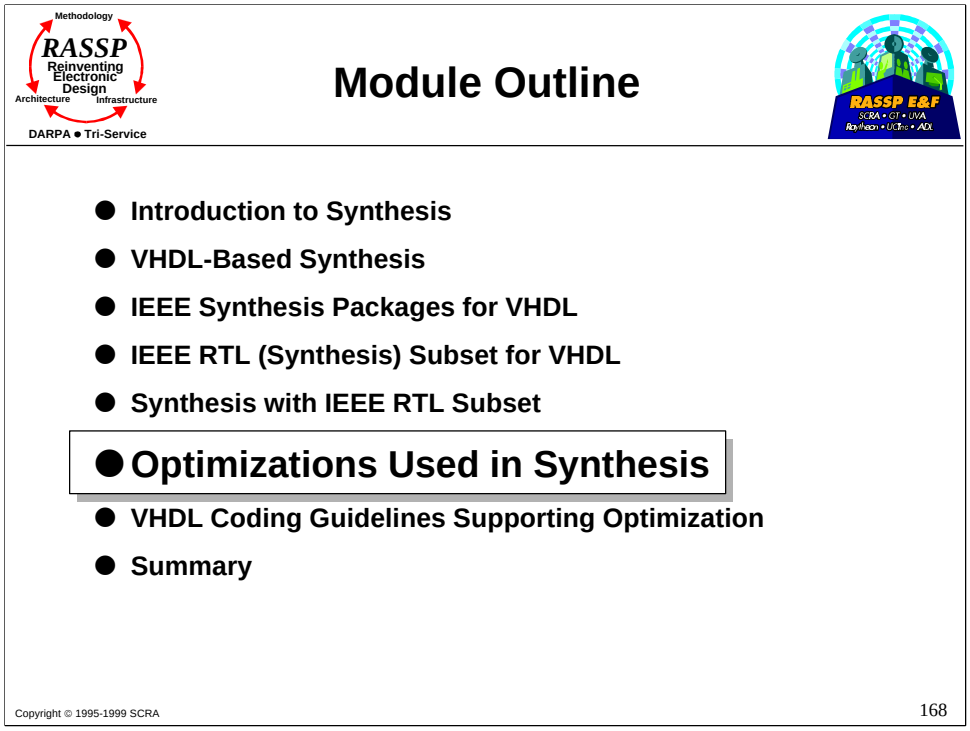
Continuation of the controller description.



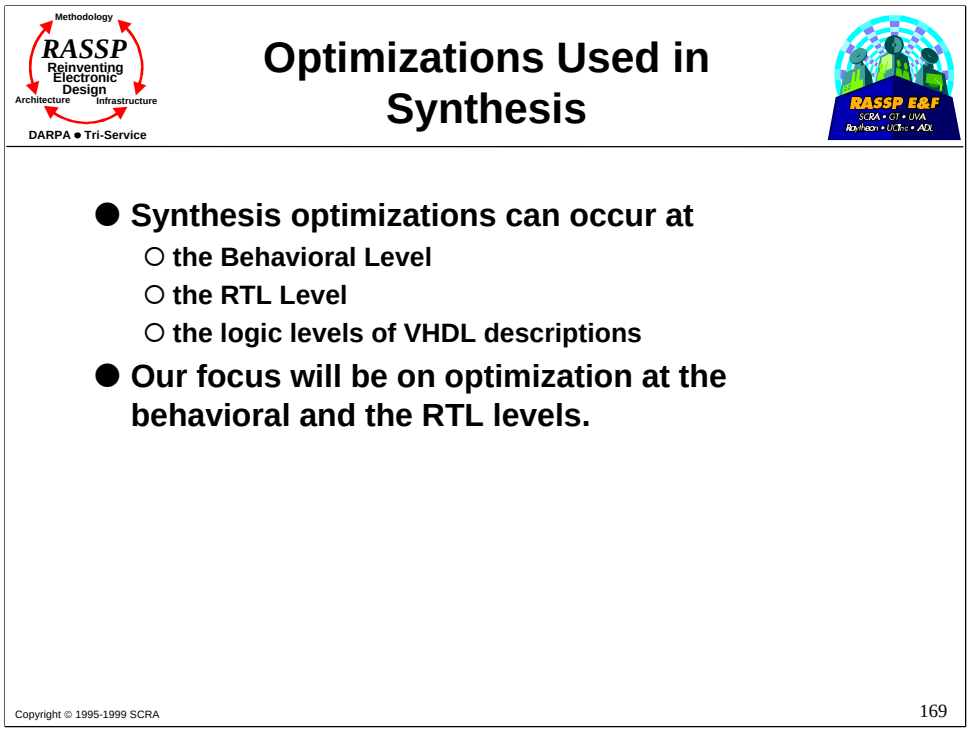
Continuation of the multiplier controller.



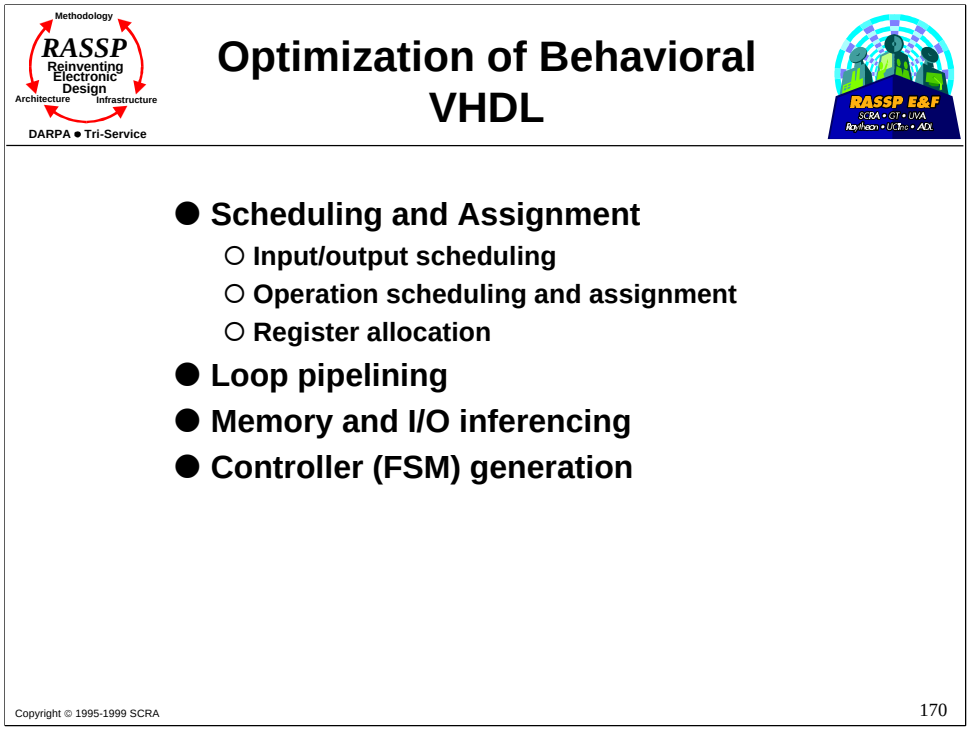
The multiplier that is synthesized using the code in the previous slides is shown here.



While our focus is on using VHDL for synthesis, often understanding the synthesis process/tools will help us write VHDL in a way to take advantage of the synthesis process. This section provides some insight into how one may write VHDL so that synthesis tools are able to exploit some the coding style to optimize the synthesis result.



Optimizations can result in higher payoffs at the higher levels of abstraction, as opposed to optimizations at the gate level. Gains of around 10-40% in metrics such as area and power are typical values that could be expected.



The optimizations are performed automatically by the synthesis tool and usually are transparent to the user.

Input/Output Scheduling

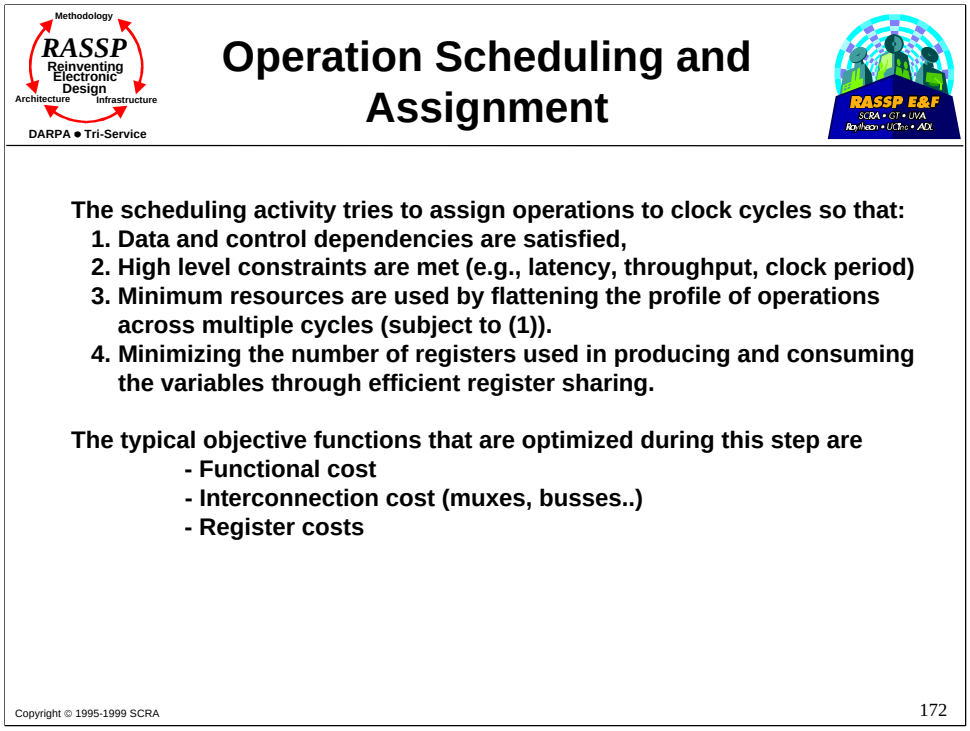
Input/Output can be *implicitly* defined in VHDL, or the protocol for I/O transfer can be *explicitly* defined by the user constraints.

The behavioral optimization (e.g., in Synopsys' Behavioral Compiler) tries to:

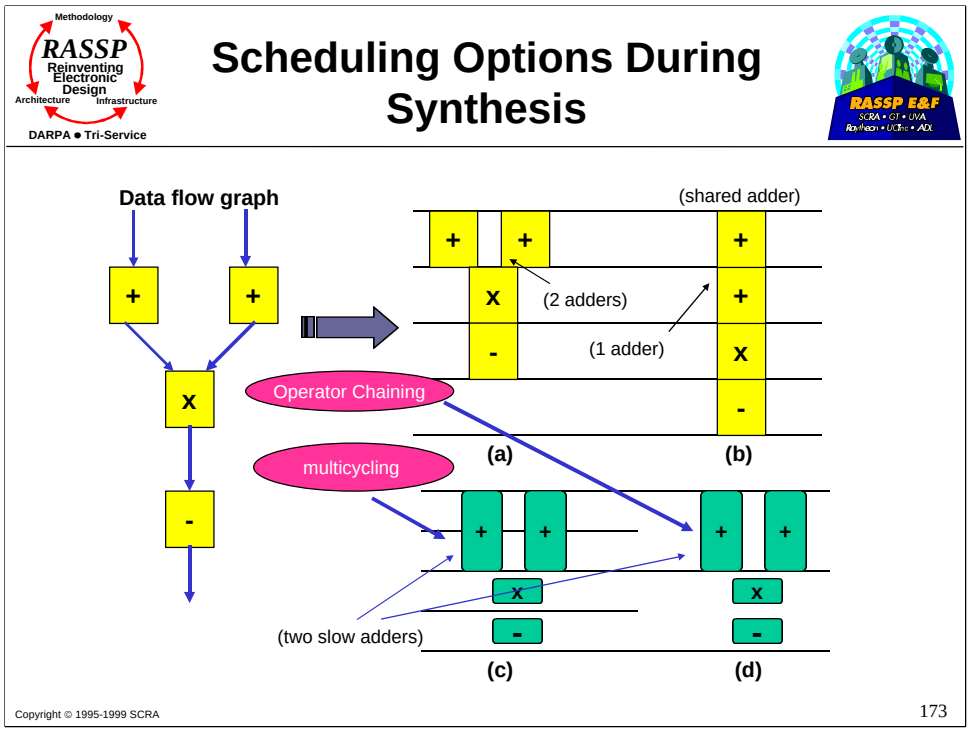
1. Delay **READ** operations as late as possible (to minimize registers for storage) by
   - Chaining the input ports directly to the arithmetic operations removing the need to register inputs, and
   - Moving the read operation to a time where it is consumed immediately to ensure that other variables can share the register.

2. Pulling **WRITE** operations as early as possible by
   - Chaining the write operations directly to the producer (no extra registers required to hold the result), and
   - putting the write operation as close in time to the producer as possible, increasing the chance of sharing registers.

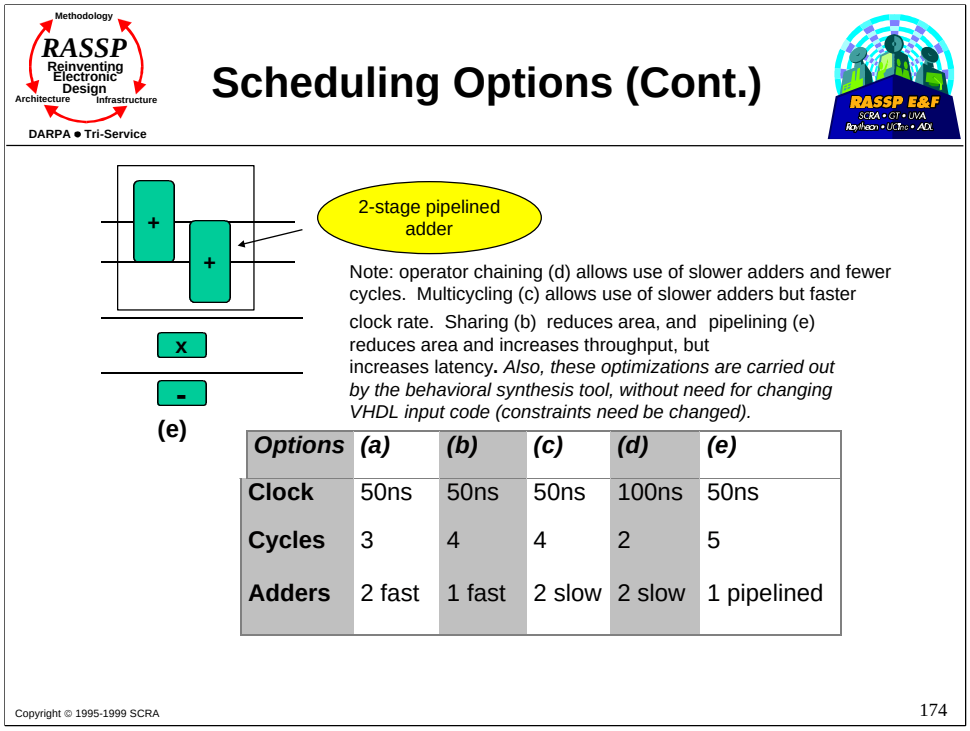
This slides presents guidelines on read and write operations within the behavioral level of synthesis. Writing as per these guidelines assists the synthesis tool in optimization of the implementation.



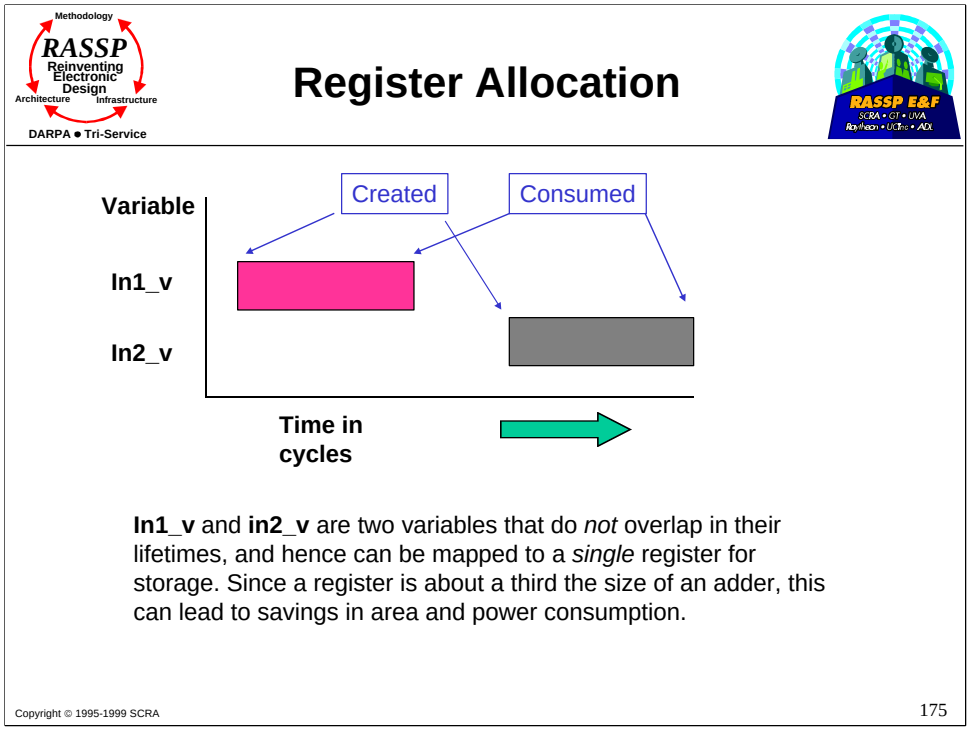
This optimization may require a coupling between the RTL and logic synthesis phases to obtain the best accuracy on the cost information (in terms of information fed back on exact costs in terms of area and power).



In multicycling an operation can stretch across multiple cycles, while in operator chaining two operations or more operations are chained within the same clock cycle, as in the multiply and subtract example above. Synopsys' documentation on their Behavioral Compiler (1997) provides similar material on their tool's behavior.



In the pipelined data path scheduling, the arithmetic unit is partitioned into a number of pipeline stages that can be used to operate on consecutive input data. The latency may be increased, but the sample period improves with a higher clock cycle. This is one possible optimization that can be done at this level. Detailed guidelines from tool vendors provide other tips.



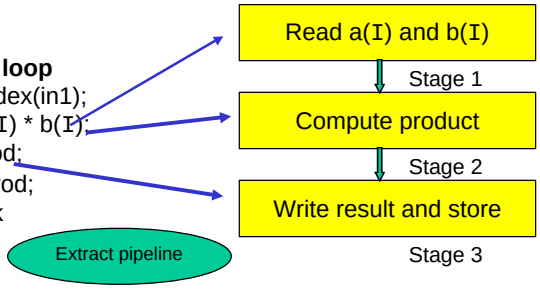
Registers can be shared across multiple variables minimizing the area of the synthesized circuit.

Loop Pipelining

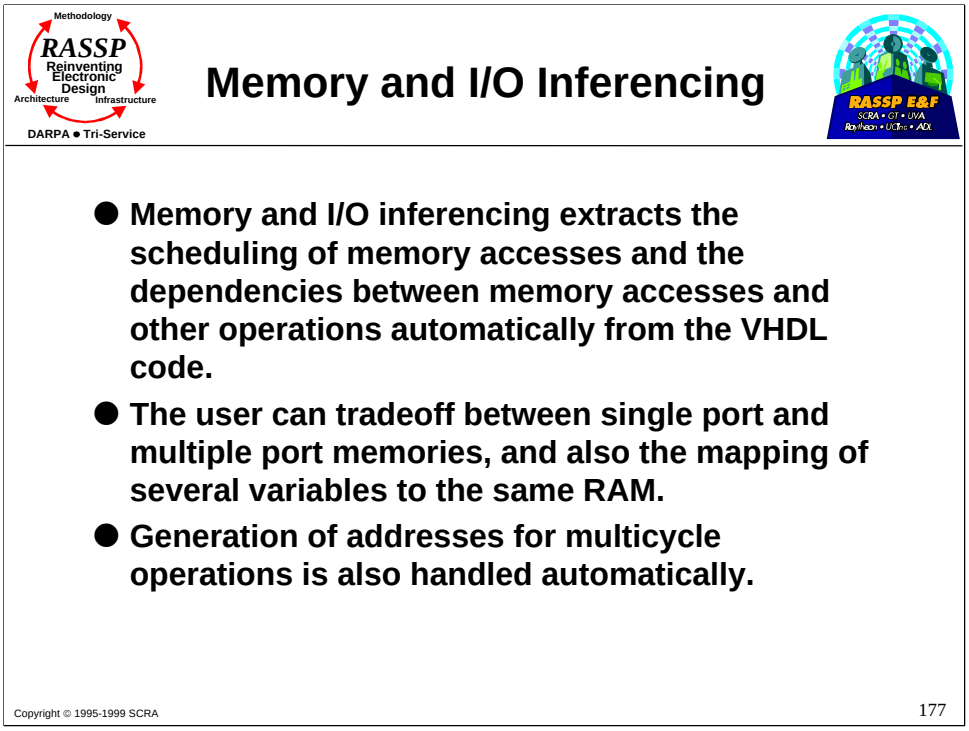
In some loops, data required for the next loop iteration is available before the current loop is completed. The loops can thus be pipelined with the second loop initiated while the first loop is completing.

```
Pipelined_loop: loop
   I  := mem_index(in1);
  dot_prod := a(I) * b(I);
  z(I) := dot_prod;
  out1 <= dot_prod;
 wait until clock
end loop;
```

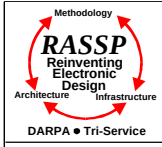
Extract pipeline



The earliest time the second loop can begin depends on the dependencies between the loop iterations and the availability of the resources to carry out the concurrent computation.



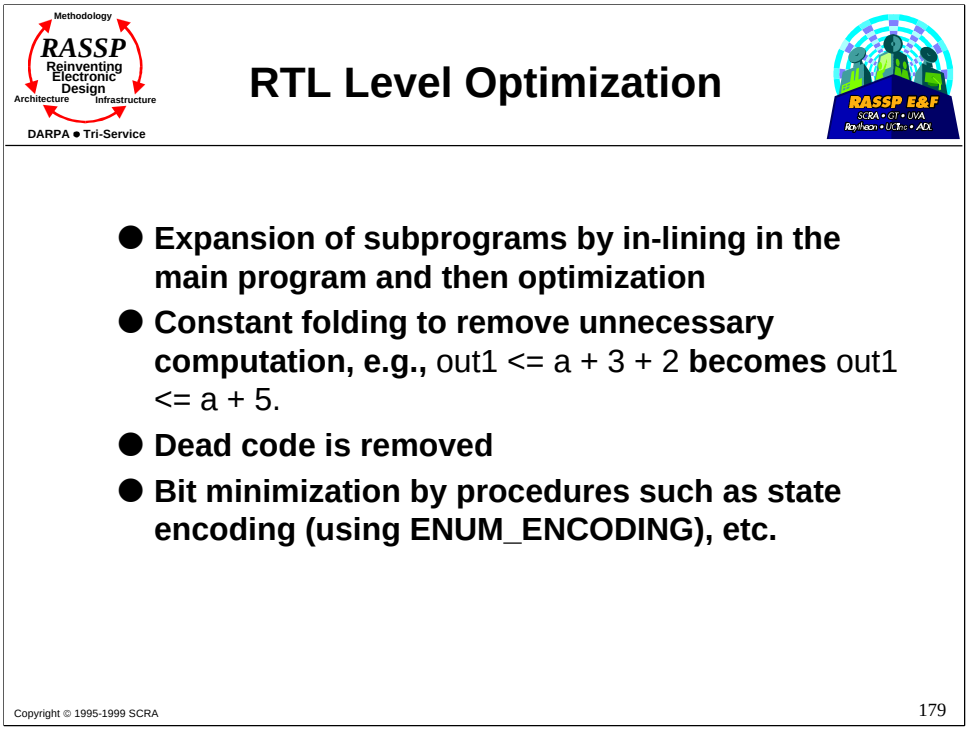
Memory and I/O inferencing is critical in any behavioral synthesis tool, but not essential for RTL level tools that would allow these structures to be specified with reference to the clock cycle.



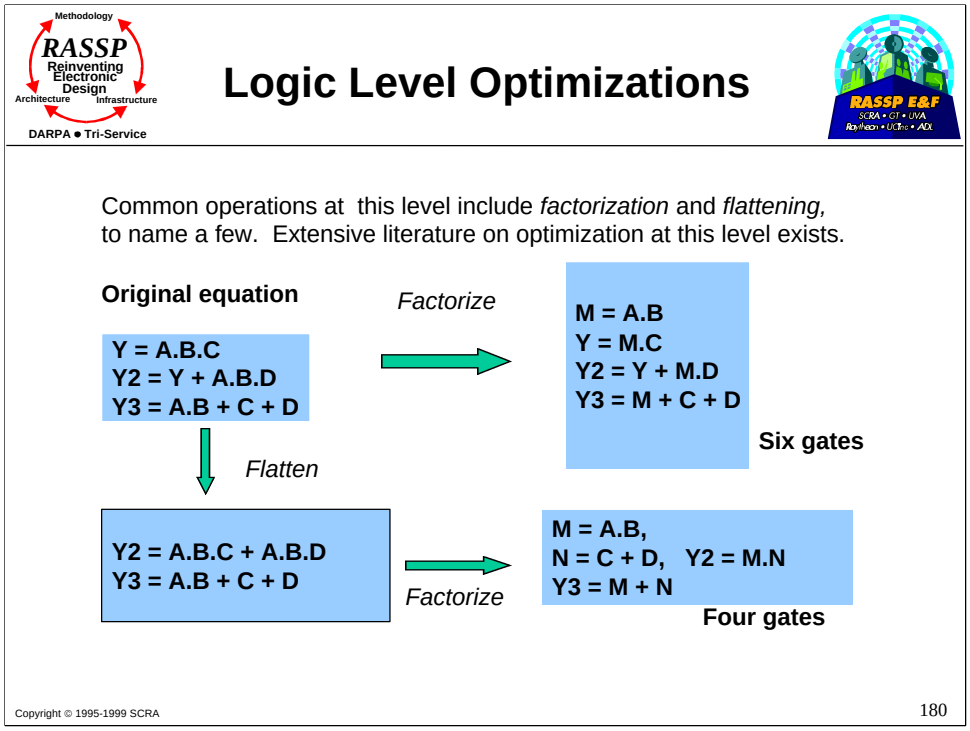
FSM (Controller) Generation

- The behavioral synthesis tool maps the VHDL input to a controller/datapath/memory architecture. Thus, after the data path scheduling, assignment and allocation is completed, a FSM is needed to :
  - Control inputs and outputs to and from the data path
  - Define state transitions
  - Define actions needed at each cycle
  - Generate control signals to control the data path elements, memories, muxes, busses, and other components of the synthesized architecture.

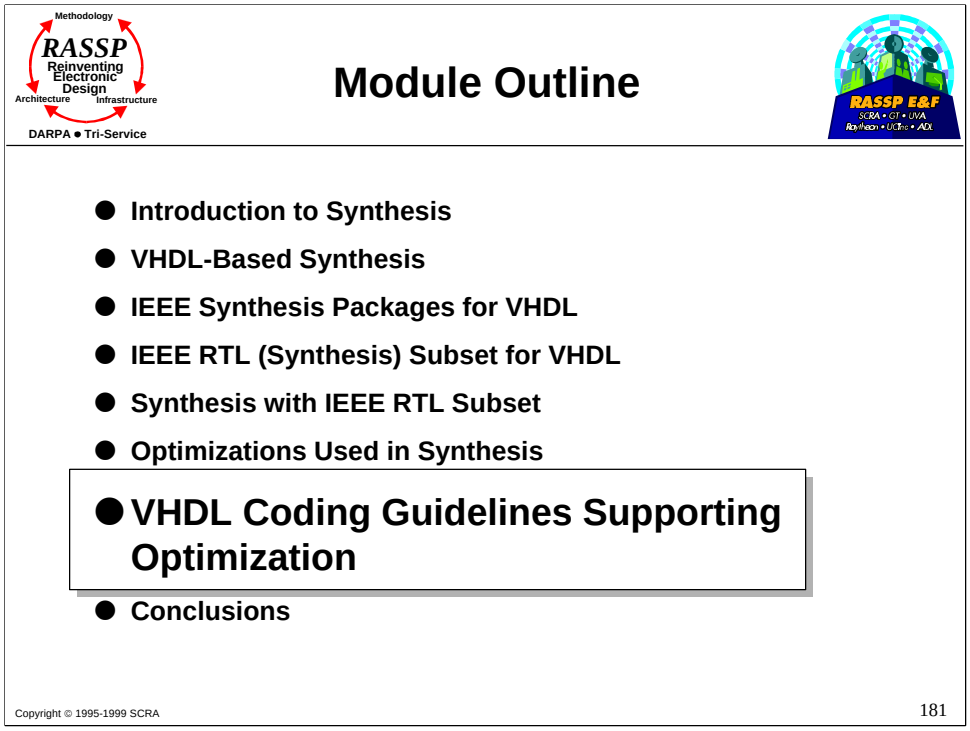
The FSM follows a template typical to one shown earlier in this module



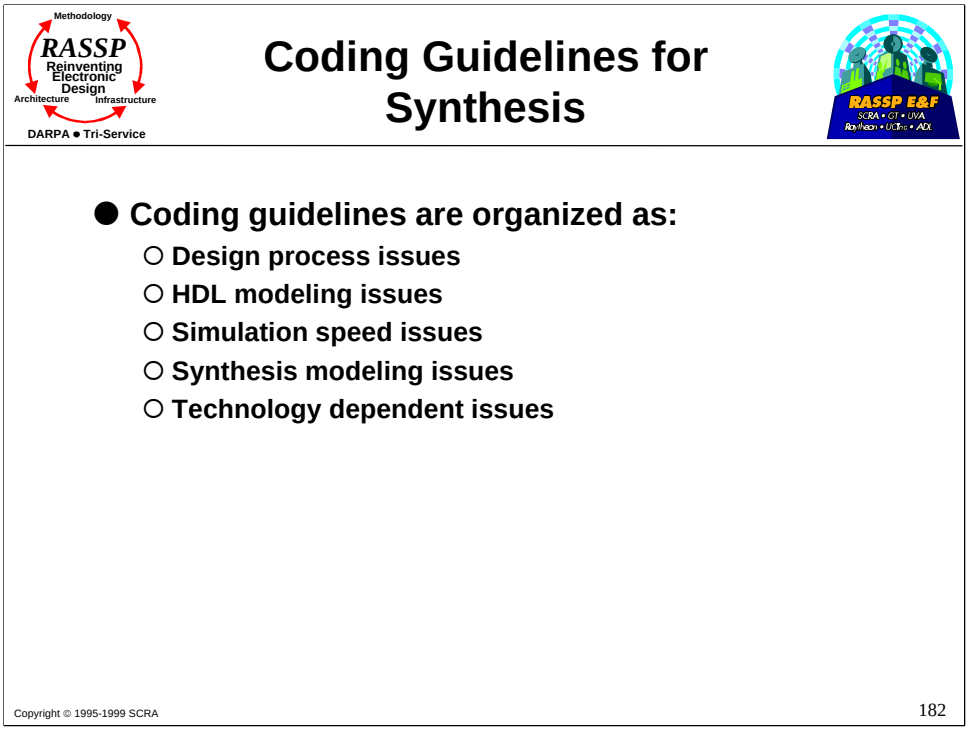
These optimizations are very similar to those used in code optimization in software.



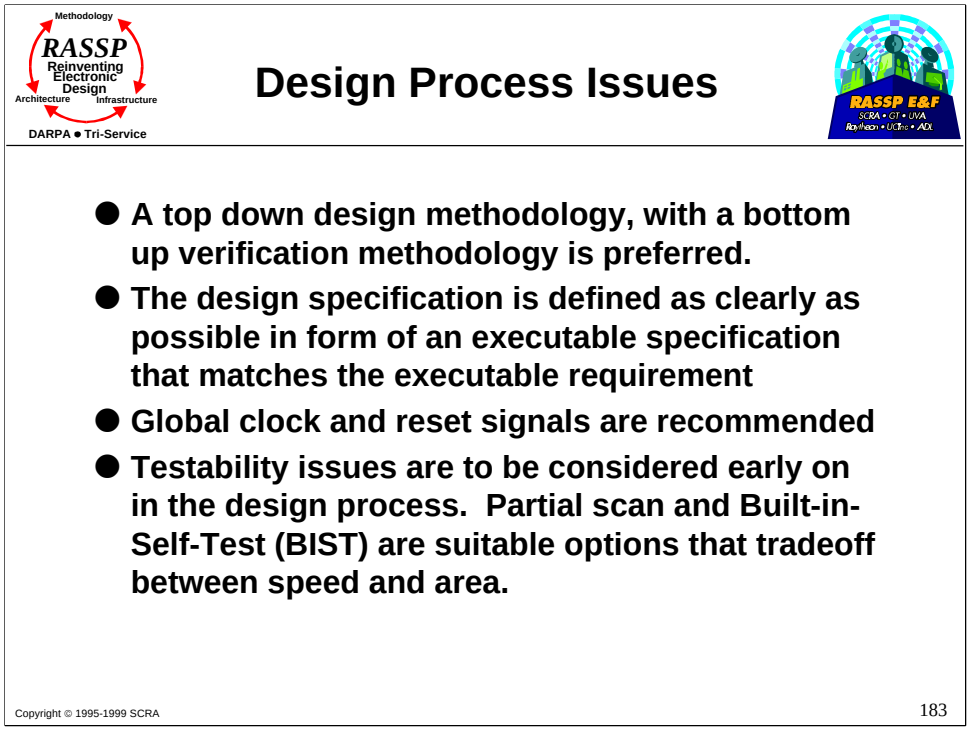
Logic level optimizations were subject of much interest in the early 1980s where a number of algorithms for multilevel logic optimization were proposed and implemented. Such optimizations are now routinely implemented by the synthesis tool, and the VHDL based designer is not expected to have familiarity with them.



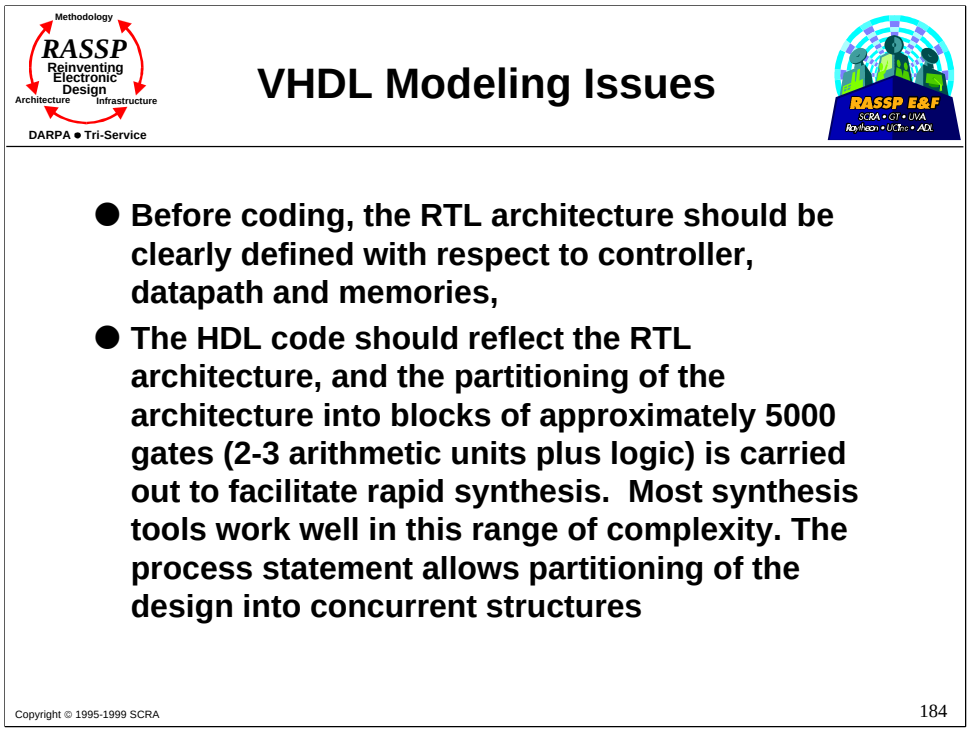
Coding guidelines are not standards, but are prescriptions that allow the designer to achieve a higher efficiency in coding, reduce the number of errors, and also ensure greater productivity.



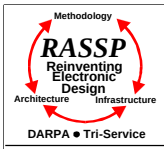
Note that the recommendations that follow are general guidelines, and are easier said than done.



Reset signals are important, as synthesis ignores initial values on variables and signals. Constants can be initialized with integers.



We are suggesting that “template” -based coding styles are useful both in enhancing productivity, taking advantage of synthesis process, and also in reducing errors in the coding process.



VHDL Modeling Guidelines (Cont.)

- Models should be as generic as possible to facilitate reuse
- Guidelines such as the use of meaningful signal and variable names facilitate readability and reduce errors, an active low signal may be called control_n, for instance.
- Comments and the use of packages allow code maintainability and modularity
- Simulation and synthesis should be considered together to facilitate verification of the design process

These are representative of good coding practice.

Simulation Speed Issues

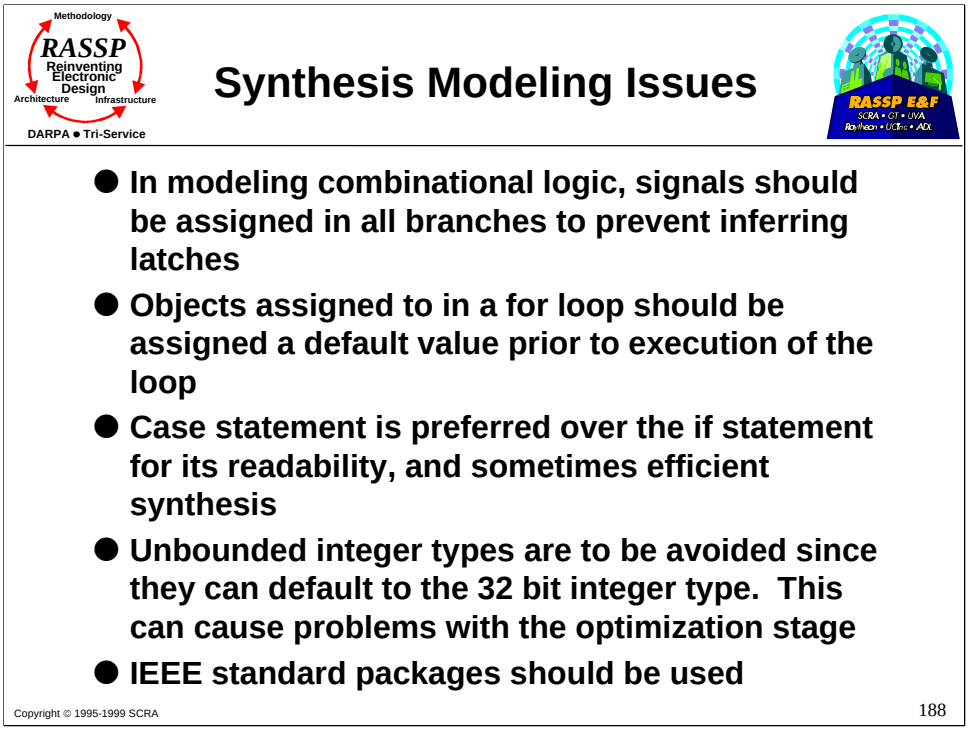
- Process statements are favored over concurrent signal assignments. This reduces the overhead that results from a large number of signals that may need monitoring
- The number of signals in the sensitivity list should be minimized to improve simulation speed
- Process statements should be of sufficient granularity (e.g., a few thousand gates), as too many processes overload the simulation
- Block statement is not recommended for synthesis, and process can be used wherever a block can.

These guidelines allow improvement on the speed of simulation.

Simulation Speed Issues

- Use variables instead of signals in a process, wherever possible
- Vectored data types can be converted to integer data type to improve simulation speed
- The 'EVENT attribute is recommended over 'STABLE, as the former is not always active. The RISING_EDGE and FALLING_EDGE defined in IEEE 1076.3 packages are preferred over 'EVENT.

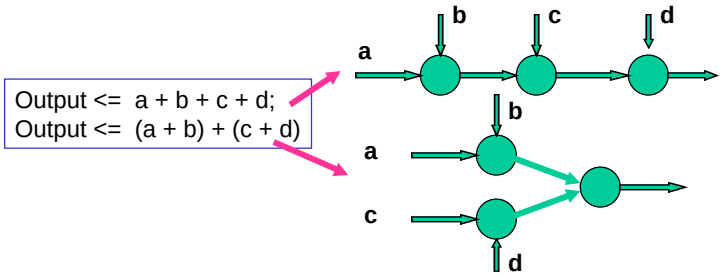
Continuation of mechanisms to improve simulation speed. Simulation is important because of the synthesize-simulate-verify cycle used in the synthesis process.



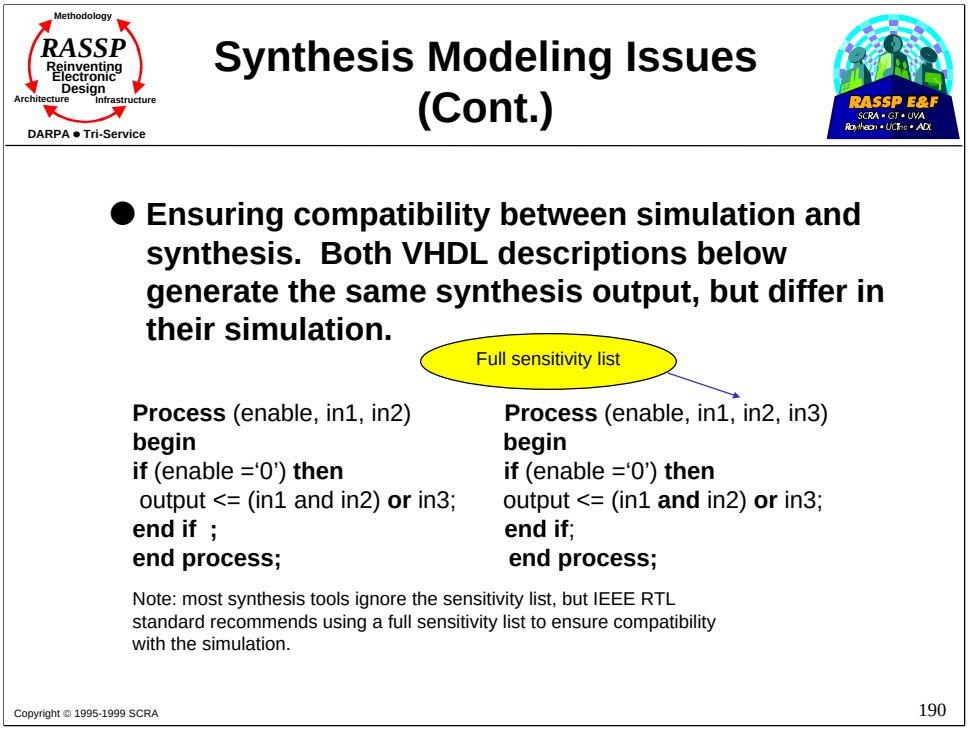
These are general guidelines on the relative advantages of using one VHDL construct over another.

Synthesis Modeling Issues

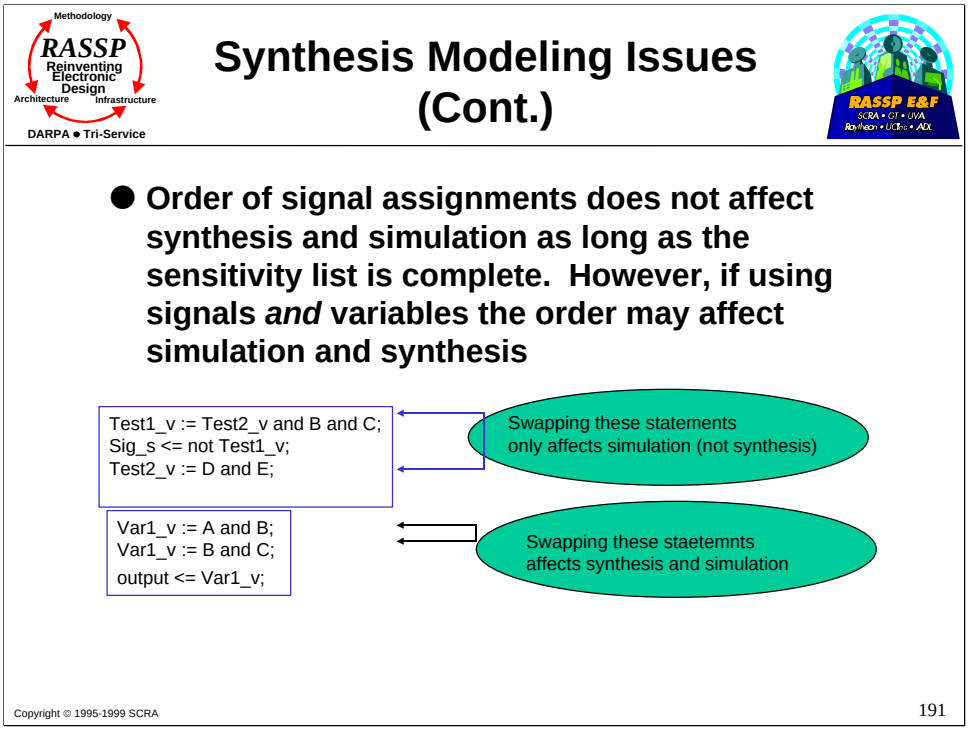
- **The** if **statement is preferred over the** wait **statement, and the former also allows the use of asynchronous signals, such as RESET/SET.**
- **Using parentheses in arithmetic expressions facilitates optimization through resource sharing, and specification of concurrency, and improves critical path. The second statement below has a shorter critical path, for instance,**



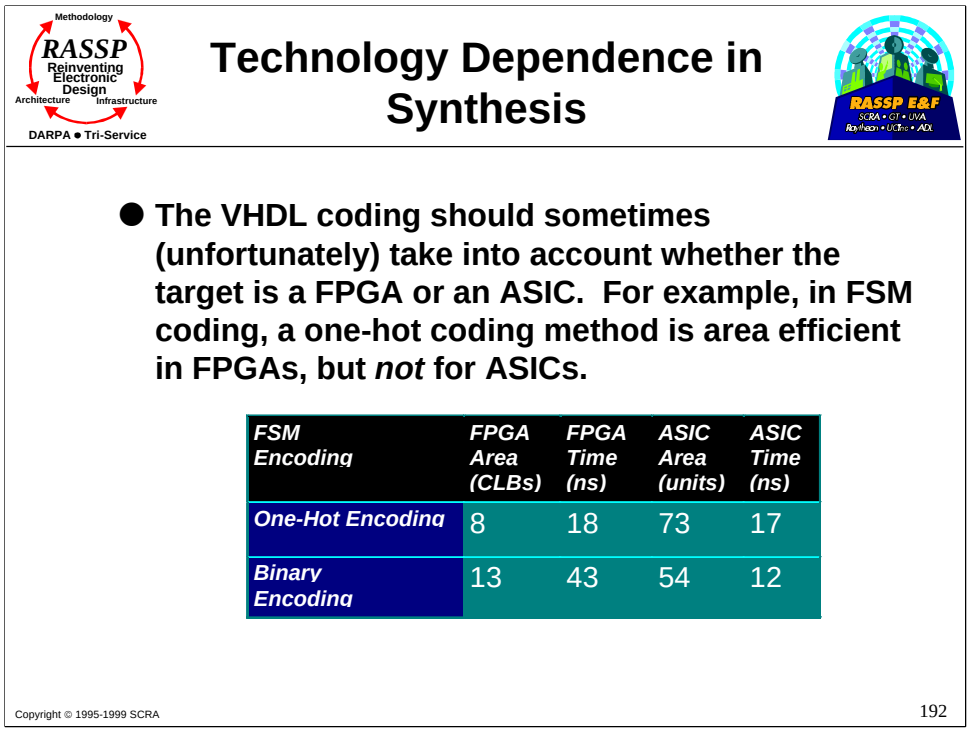
These optimizations are dependent on support from the synthesis tools.



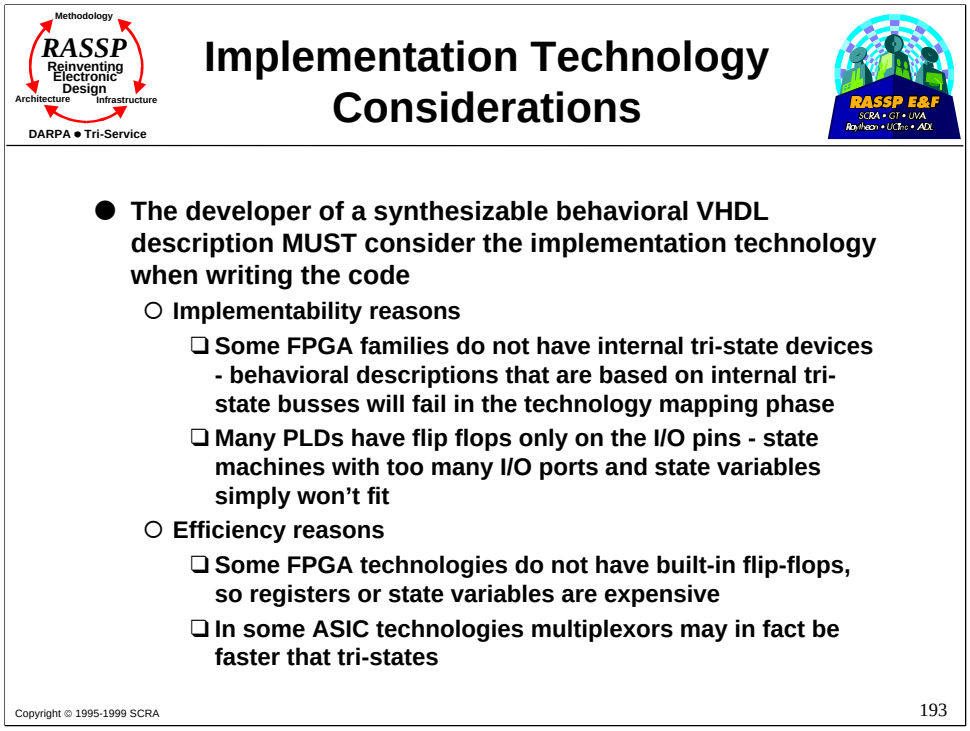
It is safe to include all signals at the right hand side in the sensitivity list to maintain compatibility between the simulation and synthesis.



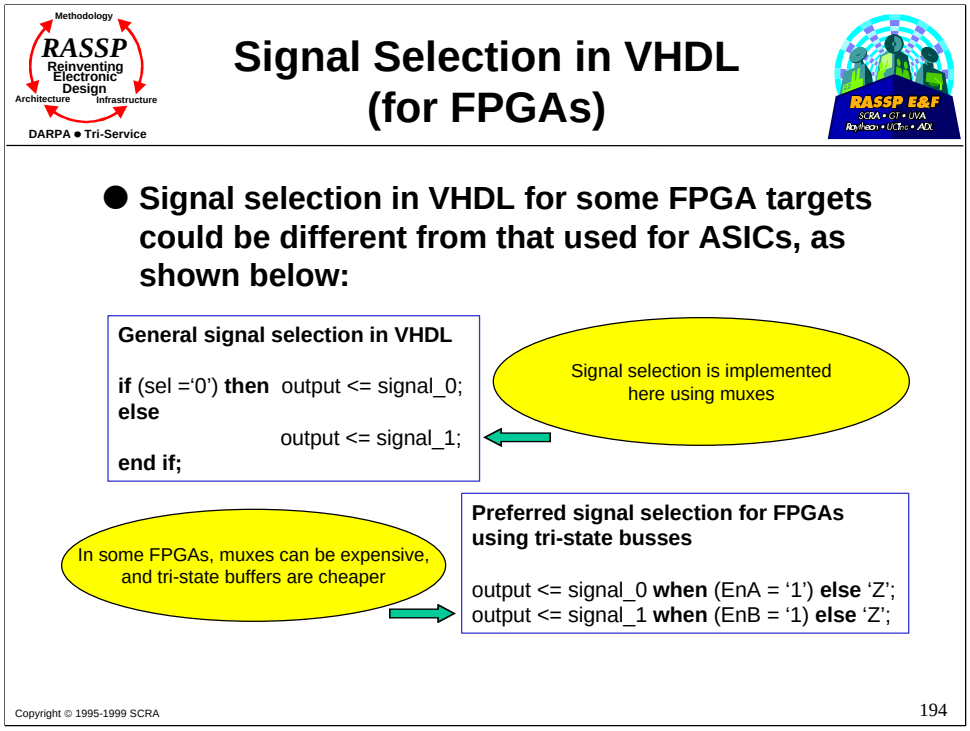
This addresses the problems of verification of the synthesis results via simulation.



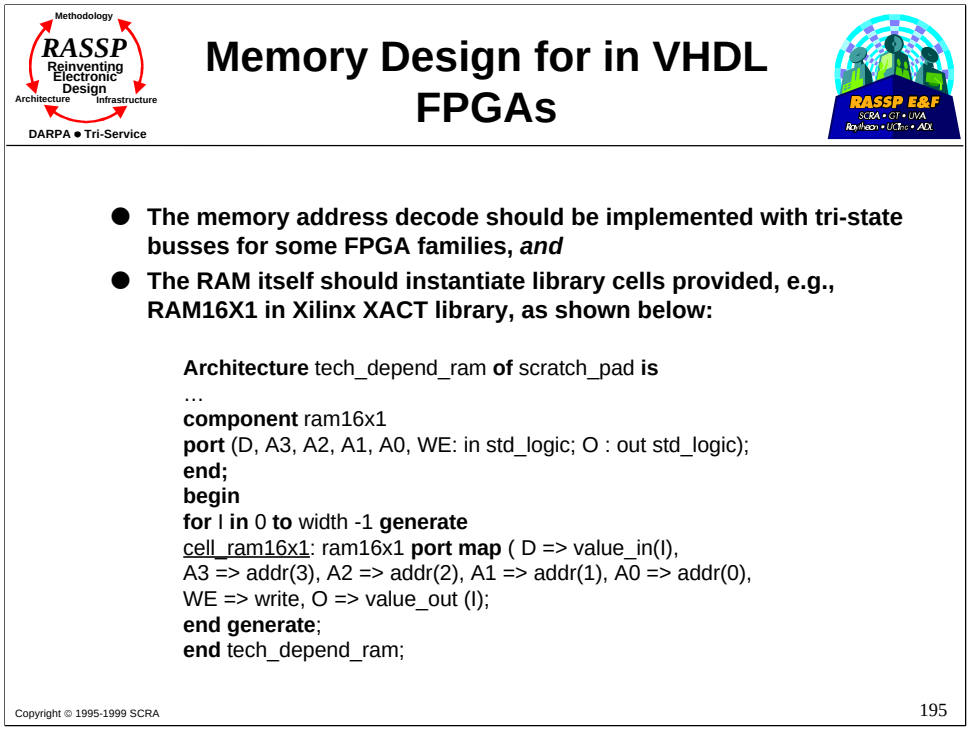
This example is true for certain families of FPGAs, as not all FPGAs support efficient synthesis of one hot encoding (especially if flip flops are not provided in quantity).



These guidelines are only true for certain FPGA families. Data books for FPGAs are the best sources of this information.



This is true for some FPGA families (e.g., Xilinx), but not for all.



These recommendation only apply to Xilinx, for instance. Other FPGA or ASIC families may have similar guidelines.

Module Outline

- Introduction to Synthesis
- VHDL-Based Synthesis
- IEEE Synthesis Packages for VHDL
- IEEE RTL (Synthesis) Subset for VHDL
- Synthesis with IEEE RTL Subset
- Optimizations Used in Synthesis
- VHDL Coding Guidelines Supporting Optimization
- **Summary**

Summary

- This module has introduced the use of VHDL in the synthesis of digital circuits
- A discussion of relevant IEEE VHDL packages and standards that support the synthesis process has been provided
- Examples have been provided to illustrate the synthesis process using VHDL
- Coding styles and optimizations and their effect on the synthesis result are also discussed.

References

[Brayton84] R. Brayton, et al, ***Logic Synthesis Algorithms for VLSI Synthesis***, Kluwer Academic Publishers, Boston, 1984.

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

[Madisetti95] Madisetti, Vijay K, ***VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis***, IEEE Press, 1995 ; © IEEE 1995

[Mentor97] Mentor QuickVHDL Simulator Tutorial and Autologic Synthesis Tools, http://www.mentor.com.

[Parker84] Parker, Alice C., "Automated Synthesis of Digital Systems," IEEE Design and Test of Computers, November 1984, pp. 75-8; © IEEE 1984

[Synopsys97] Behavioral Compiler Tutorials, http://www.synopsys.com