



Synthesis Using VHDL

RASSP Education & Facilitation

Module 60

Version 3.00

Copyright 1995-1999 SCRA

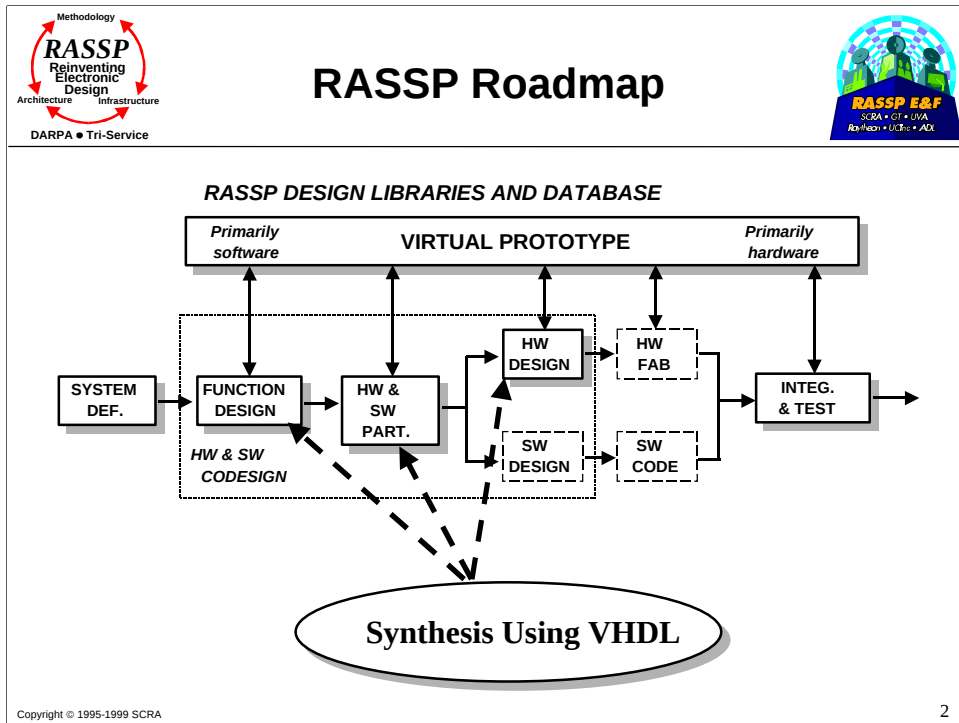
All rights reserved. This information is copyrighted by the SCRA, through its Advanced Technology Institute (ATI), and may only be used for non-commercial educational purposes. Any other use of this information without the express written permission of the ATI is prohibited. Certain parts of this work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

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

Copyright © 1995-1999 SCRA

1

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.



Module Outline



- **Introduction to Synthesis**
 - What is synthesis ?
 - Steps in the synthesis process
- **VHDL-Based Synthesis**
 - Introduction
 - Behavioral synthesis
 - RTL synthesis
 - Logic synthesis
 - Summary
- **IEEE Synthesis Packages for VHDL**
- **IEEE VHDL RTL (Synthesis) Subset**

Copyright © 1995-1999 SCRA
3

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 first 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



Module Outline (Cont.)



- **Synthesis with the IEEE RTL Subset**

- Combinational systems
- Registers and clock related behavior
- Expressions, operators, types
- Sequential circuits

- **Optimizations Used in Synthesis**

- Behavioral level optimizations
- RTL level optimizations
- Logic level optimizations

Copyright © 1995-1999 SCRA

4

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.



Module Outline (Cont.)



- **VHDL Coding Guidelines Supporting Optimization**
 - Technology independent guidelines
 - Technology dependent guidelines
 - FPGAs
 - ASICs/Gate arrays
- **Summary**

Copyright © 1995-1999 SCRA

5

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



What is Synthesis ?



- **Synthesis is a general term that describes the process of transformation of the model of a design, usually described in a hardware description language (HDL), from one level of behavioral abstraction to a lower, more detailed behavioral level.**
- **These transformations try to improve upon a set of objective metrics (e.g., area, speed, power dissipation) of a design, while satisfying a set of constraints (e.g., I/O rates, MIPS, sample period) imposed on it.**

Copyright © 1995-1999 SCRA

7

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.



Steps in the Synthesis Process



- **Synthesis to the gate-level of abstraction consists of three major steps -**
 - Design specification (in a machine-readable form)
 - Design implementation
 - Design validation and verification
- **Design specification implies a description of the design where functional, performance, and cost constraints (area, power, speed) are captured in a form that facilitates processing (i.e., executable) in a CAD environment. Two common methods for capturing the specification are:**
 - Graphical methods
 - Language-based methods

Copyright © 1995-1999 SCRA

8

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

Graphical Methods

- **These include:**
 - Block diagrams, state diagrams, schematics, data flow graphs, timing diagrams, truth tables, etc.
- **Advantages:**
 - Faster learning curve,
 - Intuitive documentation, and
 - Reusability of design.
- **Disadvantages:**
 - Limited support for functional and timing hierarchy in complex digital systems
 - Limited support for multiple views (e.g., simulation versus hardware generation).

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.



Language-based Methods



- **These methods include:**

- Hardware description languages (HDLs), high level algorithmic macro descriptors (such as Matlab), or application-specific languages (such as Silage and DSP/C for signal processing).

- **Advantages:**

- Ability to serve as a standard medium of communication between the algorithm developer and the synthesis expert
- Robustness through well-defined semantics
- Ability for representing complex behavior and data structures

Copyright © 1995-1999 SCRA

10

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 (Cont.)



- Support for synthesis and its verification in the same environment,
- Their system-independent nature that facilitates design documentation.
- **Disadvantages:**
 - Specification is harder to visualize, and requires a longer learning curve.

Copyright © 1995-1999 SCRA

11

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.



Language-based Specification Methods (Cont.)



- **These methods include:**

- Hardware description languages (HDLs), high level algorithmic macro descriptors (such as Matlab), or application-specific languages (such as Silage and DSP/C for signal processing).

- **Advantages:**

- Ability to serve as a standard medium of communication between the algorithm developer and the synthesis expert,
- Well defined semantics,
- Ability for representing complex behavior and data structures,
- Support for synthesis and verification.

- **Disadvantages:**

- Specification is harder to visualize, and requires a longer learning curve.

Copyright © 1995-1999 SCRA

12

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.



Steps in the Synthesis Process (Cont.)



- **Design Implementation** consists of a number of intermediate steps:
 - **Algorithm design:**
 - Evaluates the performance of the algorithm being implemented in terms of:
 - ⇒ The precision of the word length,
 - ⇒ The number of iterations, and
 - ⇒ The quality of the results obtained with respect to the design requirements.
 - **Behavioral simulations:**
 - Done after the algorithm design is complete to verify the detailed functional behavior of the system.

Copyright © 1995-1999 SCRA

13

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.



Steps in the Synthesis Process (Cont.)



- Data and control flow graph generation:
 - Derives an intermediate representation of the behavioral specification where the data flow, input/output, control dependencies and synchronization signals are documented.
- The result of the previous steps in synthesis is sometimes called a behavioral-level model of the circuit.

Copyright © 1995-1999 SCRA

14

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.



Steps in the Synthesis Process (Cont.)



- The next steps involve a knowledge of the hardware and software modules available in the design library. At this stage of the synthesis process, based on the control/data flow graph, the steps are:
 - Module selection (determining which modules are used in design),
 - Estimation and allocation of the number of modules to be used, and
 - Transformations that optimize the control/data flow graph without changing the input/output functional properties, but improve on its synthesis metrics (such as area, power, or speed).

Copyright © 1995-1999 SCRA

15

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.



Steps in the Synthesis Process (Cont.)



- Scheduling determines when a certain operation will be executed, and assignment determines the module on which the operand will be executed (e.g., an addition on an adder module).
- At the completion of this stage of the synthesis the result is called a register-transfer level (RTL) model of the circuit
- The RTL description or model is then converted to a logic level implementation through a process called logic synthesis, and after further technology level optimization a gate-level model of the circuit results, which must be verified for both timing and function.

Copyright © 1995-1999 SCRA

16

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



Steps in the Synthesis Process (Cont.)



- **Validation at each stage of the design is a feature of the top-down structured design process, and can proceed in a bottom-up or top-down manner**
 - E.g., in a bottom-up process each selected module is tested to ensure correct functional behavior. After all the constituent modules are validated the entire design is validated at the RTL level.
 - After completion of the logic synthesis, the verification process is carried out at the gate level.

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).



Module Outline

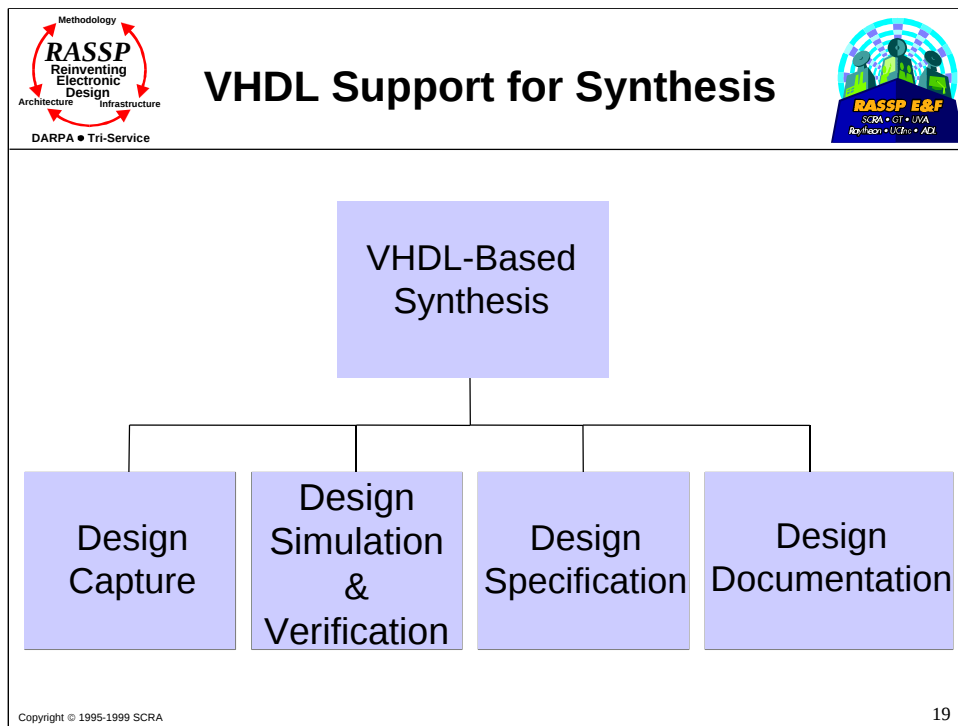


- Introduction to Synthesis
- **VHDL-Based Synthesis**
- IEEE Synthesis Packages for VHDL
- IEEE RTL (Synthesis) Subset for VHDL
- Synthesis with IEEE RTL Subset
- Optimizations Used in Synthesis
- VHDL Coding Guidelines Supporting Optimization
- Summary

Copyright © 1995-1999 SCRA

18

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



- **VHDL allows designer to capture the details of a digital design in a form that is machine-readable, that is, the design can be entered into a CAD system.**
- **VHDL can be used as an alternative to, or in conjunction, with schematic-based design entry methods.**
- **Typical logic synthesis and RTL synthesis tools are primarily design capture environments. Over 90% of designers use VHDL for this purpose (EE Times, 1996).**

Copyright © 1995-1999 SCRA

20

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



Design Specification



- **Within a structured design flow, VHDL can be used to capture the behavioral, interface and performance-related aspects of the design.**
- **The VHDL representation can then be synthesized to an implementation through several consecutive stages, each incrementally adding more detail to it.**
- **Behavioral synthesis and advanced RTL synthesis tools are representative of environments performing this function.**

Copyright © 1995-1999 SCRA

21

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 an optimized design.



Design Simulation and Verification



- **VHDL used to simulate key properties of a design prior to and after synthesis.**
- **VHDL can verify those properties at various levels of the synthesis process.**
- **Various levels of functional and timing simulation can include both technology dependent and independent aspects of the design under synthesis**

Copyright © 1995-1999 SCRA

22

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.



Design Documentation



- A design represented in VHDL is a well-documented design, and is accepted as such by the US Department of Defense.
- VHDL, an IEEE standard, allows designs to be represented in a non-proprietary, technology-independent manner.
- VHDL is capable of representing both the design and its test environment in a tool-environment, technology-neutral manner, adding greatly to design productivity, reuse, and capability for its rapid upgrade.

Copyright © 1995-1999 SCRA

23

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.



Levels of Synthesis



- **Commonly used synthesis tools support**
 - Behavioral Synthesis
 - RTL Synthesis
 - Logic Synthesis

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



Synthesis Categories



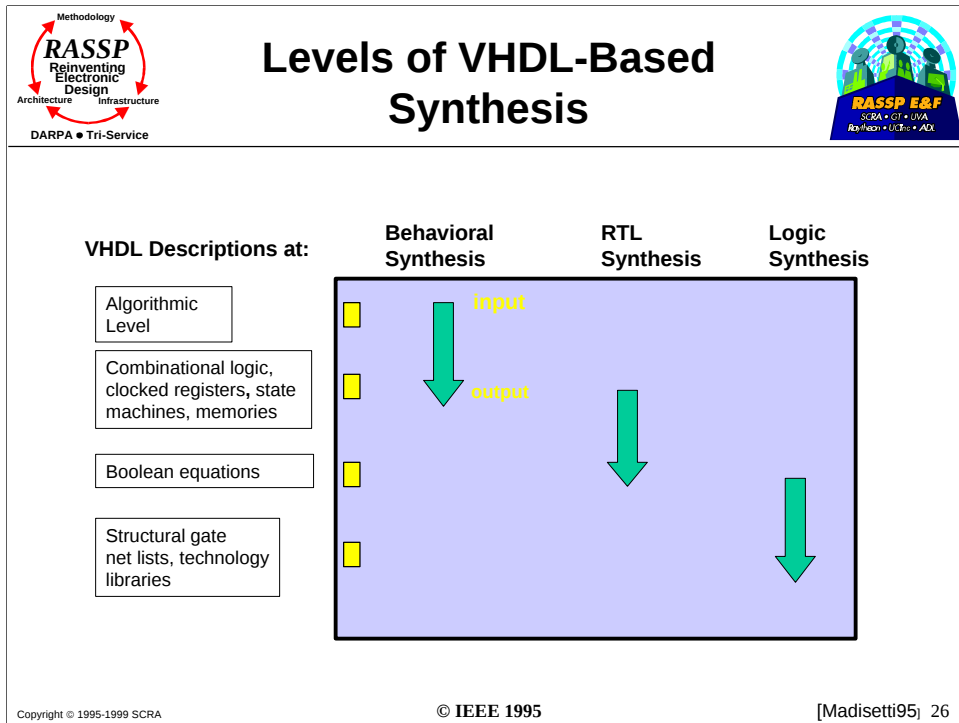
- **Algorithm Synthesis - (also called Behavioral Synthesis) - synthesis of abstract behavior or control-flow behavior from a high level algorithm description**
 - Involves high-level scheduling and assignment
- **Register-Transfer Synthesis - synthesis of register-transfer structure from abstract, control-flow, or register-transfer behavior**
- **Logic Synthesis - synthesis of gate-level logic (an implementation) from register-transfer structure or Boolean equations**

Copyright © 1995-1999 SCRA

© IEEE 1984

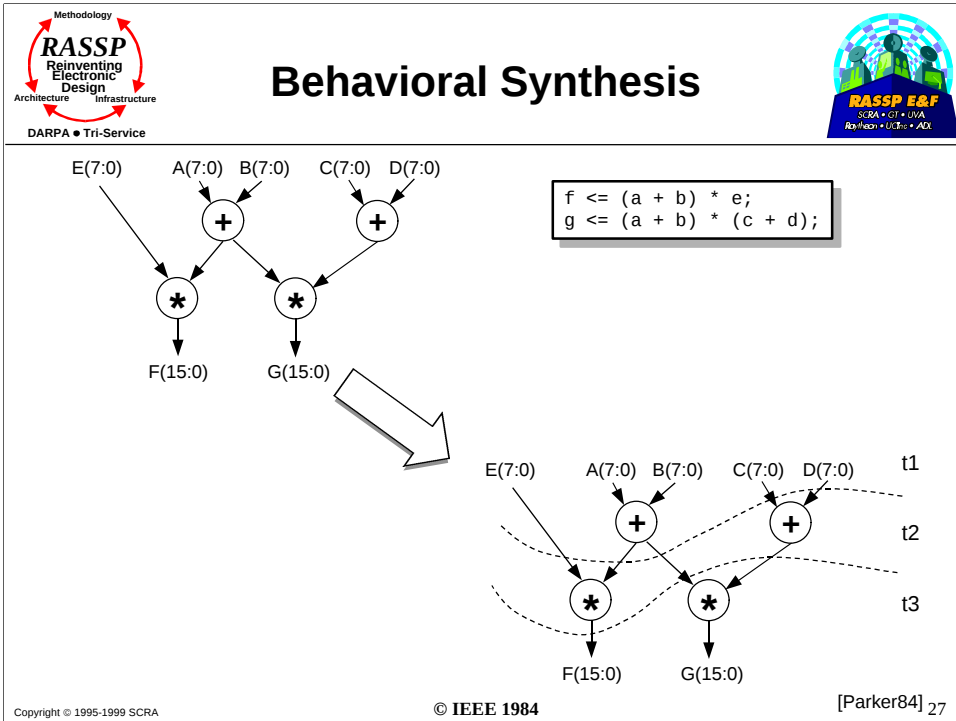
[Parker84] 25

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.

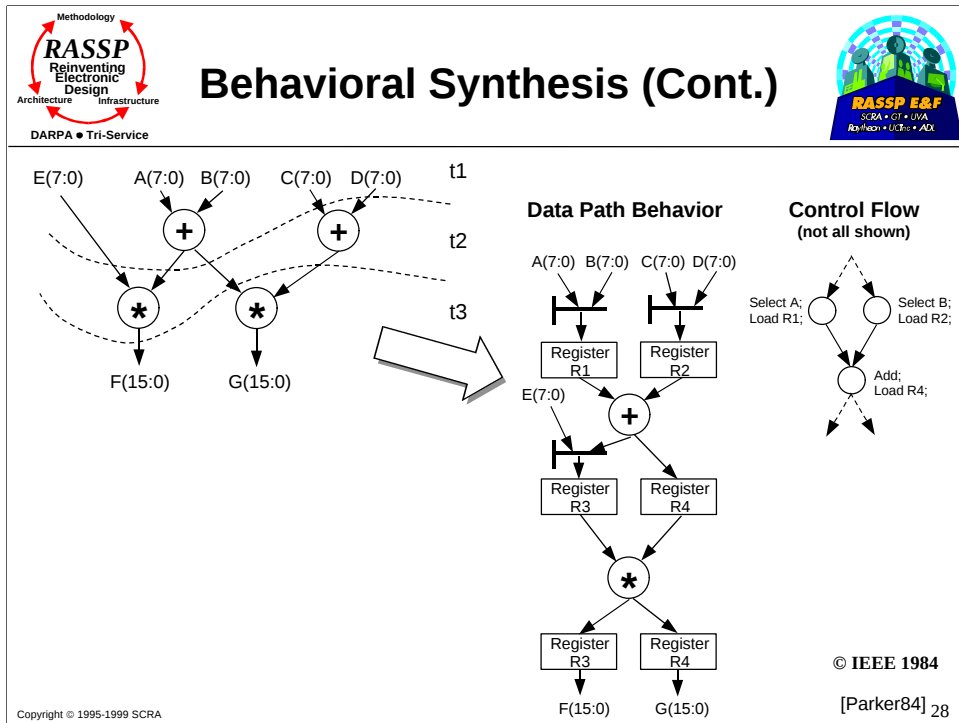


The arrows indicate the starting and end points for each type of synthesis. The tools to the left of the figure would need support from tools towards to the right to ensure continuity of the synthesis to the gate/transistor level implementation.

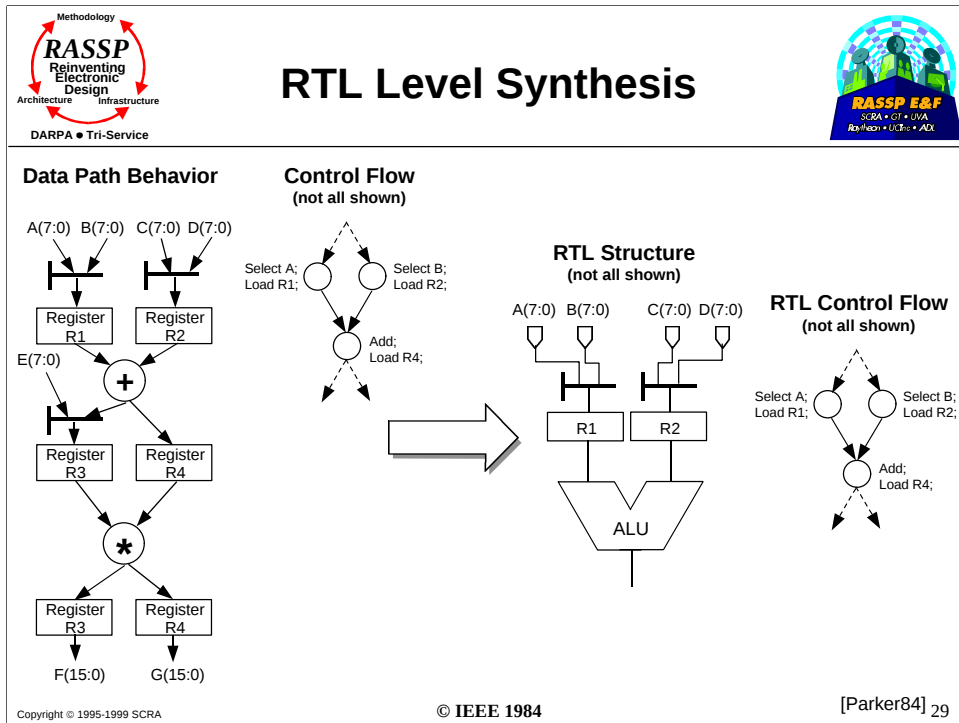
System-level synthesis is an advanced step that includes both hardware and software codesign and synthesis, and is covered in other RASSP modules.



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.

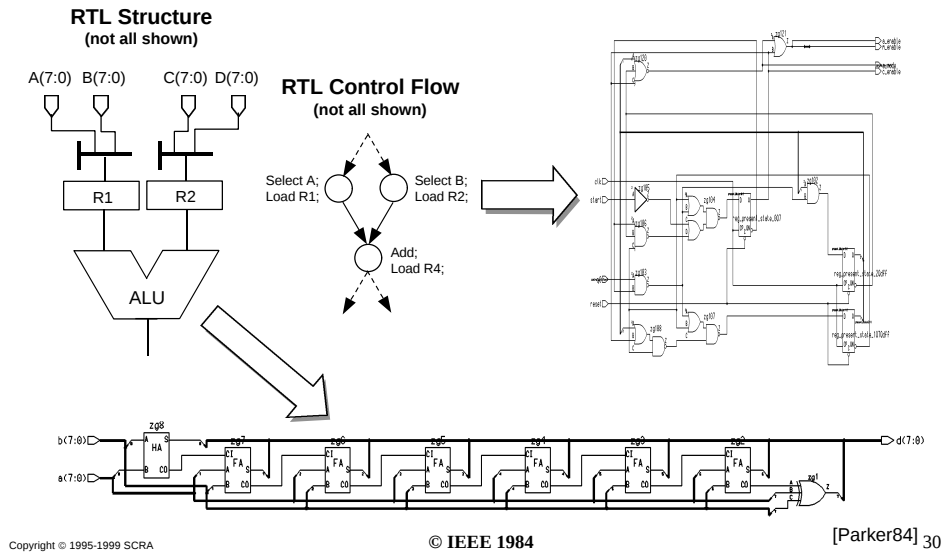


The timing annotated flow graph is now mapped to an RTL structure with an associated data path (the computational portion) and a control flow graph (the controller) that is shown on the right. The exact structure of the computation is not important at this point in the presentation, but it is clear that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

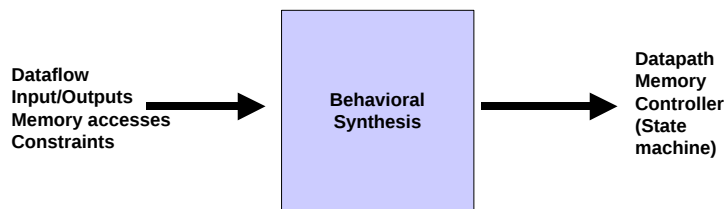
Logic Synthesis



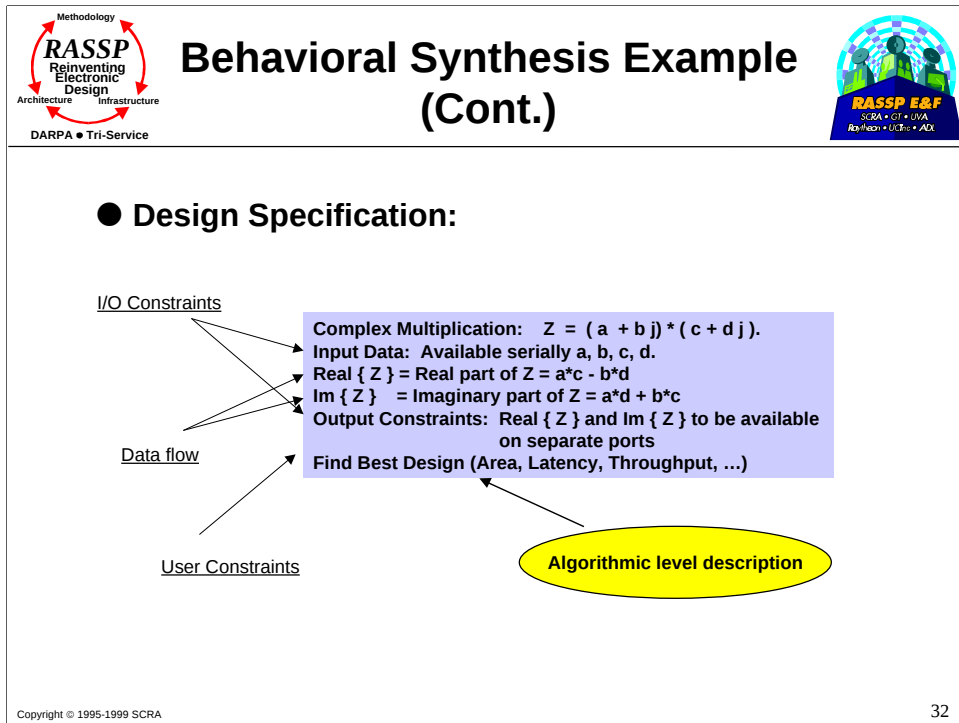
The RTL description is then synthesized to a gate level description shown above to unambiguously describes one possible implementation of the original behavioral algorithm specified as a flow graph.

Behavioral Synthesis

- The input to a behavioral synthesis system describes the dataflow, input/output constraints, memory accesses, and user defined constraints (sample period, latency...).
- The output of the behavioral synthesis system provides the datapath, registers, memories, I/O structures, and a state machine-based controller that satisfies the same test bench as the input.

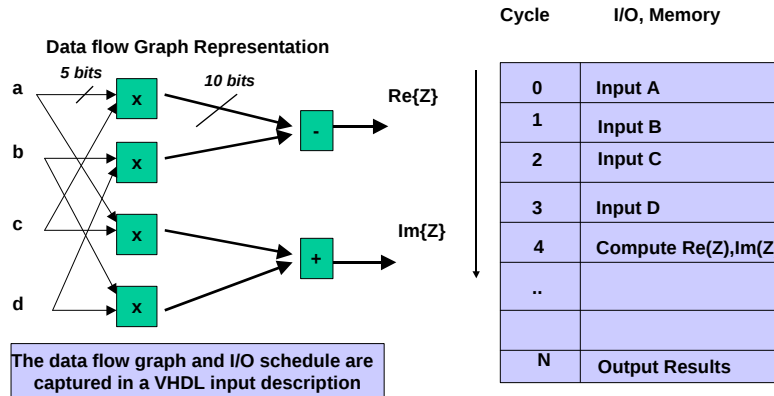


We now describe the term behavioral synthesis in some detail. We have used the Behavioral Compiler™ from Synopsys as a reference tool for this description in terms of the scope of behavioral synthesis.



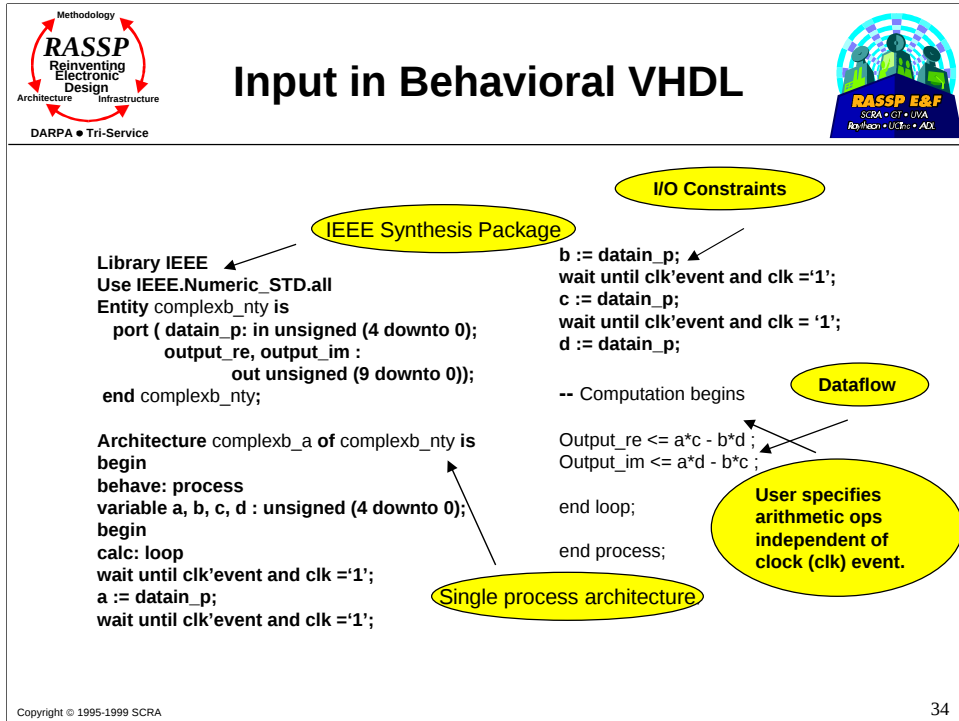
We use a language-based (VHDL) description of the input specification. Note that these tools do not support the entire subset of the language. This is a pseudo-code description of the input specification.

Input to Behavioral Synthesis



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.

Results from Behavioral Synthesis

Design Objective 1: Fast Implementation
Clock cycle: 50 ns

Cycles → 0 1 2 3 4

READ	A	B	C	D		Metric	Value
MULT_1			A*C	A*D		Area	1700 gates
MULT_2			B*C	B*D		Latency	250 ns
ADD_1					AC*-B*D	Multipliers	Two
ADD_2					A*D+B*C	Adders	Two

Area = 1700 gates
Latency = $5 \times 50 = 250$ ns
Multipliers = Two
Adder = Two

Design Objective



Fast



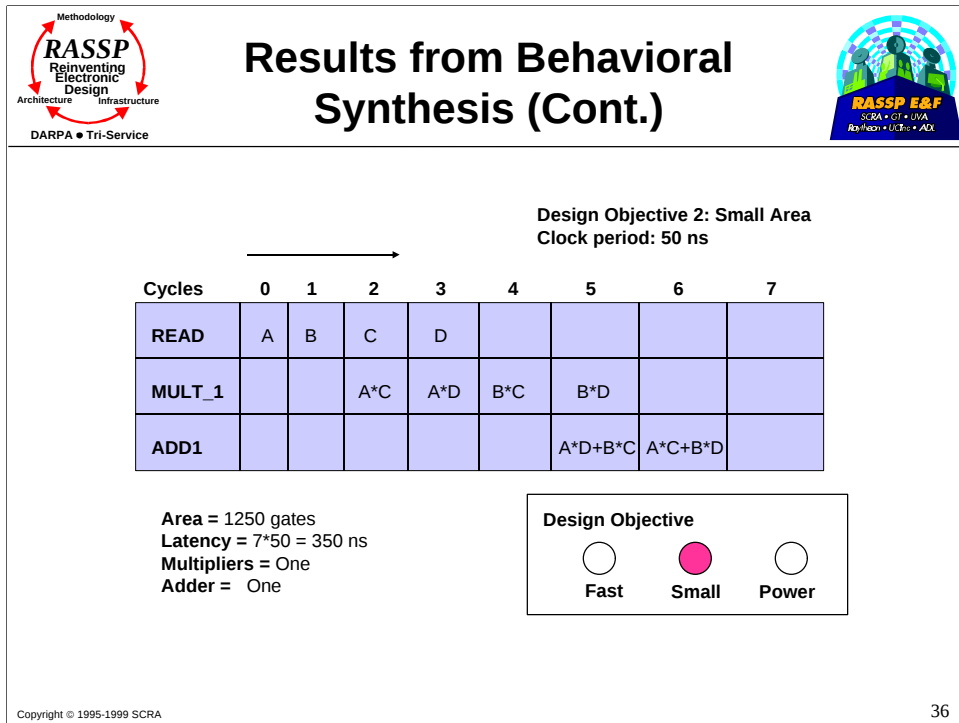
Small



Power

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]



Characteristics of Behavioral Synthesis



● Pros

- User inputs behavioral-level VHDL code that is short, easy to write, and verify, leading to increased productivity
- The synthesis tools perform the tasks of resource allocation, assignment, scheduling, and optimize area, latency, and power dissipation based on user input
- Output from a behavioral synthesis tool can be starting point for RTL synthesis tools for further optimization.

● Cons

- Supports a very small subset of VHDL (e.g., single process architectures).
- Non-standard implementations depending on tool vendors
- Currently supports application specific areas (such as DSP) with primarily loop-dominated computational flow graphs
- Ability to handle larger graphs (more than a few dozen operations) limited
- Needs supporting high level libraries of components

Copyright © 1995-1999 SCRA

37

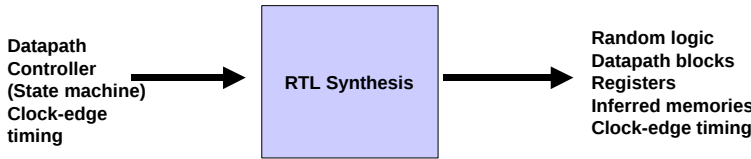
We have listed our view of the pros and cons of using a behavioral synthesis tools. Clearly they have the advantage of reducing the burden on the designer when dealing with a domain specific design that has a sufficient number of constraints to provide a limit on the number of implementation choices.



RTL Synthesis



- Input to the RTL synthesis environment includes the number of data path components (adders, multipliers,...), the mapping of operations to data path components, and a controller (finite state machine) that contains the detailed schedule (related to clock edge) of computational, I/O, and memory operations.
- Output of the RTL synthesis provides logic level implementations that can be evaluated through the optimization of the data path, memory, and controller components, individually or in a unified manner, through mapping to gate-level component libraries and limited resource sharing.



```

graph LR
    A["Datapath  
Controller  
(State machine)  
Clock-edge  
timing"] --> B[RTL Synthesis]
    B --> C["Random logic  
Datapath blocks  
Registers  
Inferred memories  
Clock-edge timing"]
  
```

Copyright © 1995-1999 SCRA
38

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.

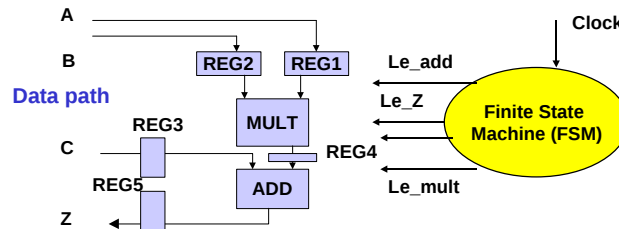
RTL synthesizers allow the user to benefit from mapping RTL level components directly to optimized RTL components from vendor libraries resulting in faster synthesis.

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

RTL Synthesis Example

● Design Specification

Multiply Accumulate (MAC): $Z = A*B + C$
Input data: Available serially
Computation: $Z = A*B + C$
Available Hardware: 1 Multiplier, 1 Adder
Output: Z available at end of computation.



Copyright © 1995-1999 SCRA

39

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

RTL VHDL Input

The architecture consists of a datapath and two finite state machine (FSM) processes

```
Datapath: process
variable reg_1, reg2 : unsigned (4 downto 0);
variable reg_3, reg_4, reg_5: unsigned (9 downto 0);
begin
wait until clk'event and clk'event='1';
if le_reg1 = '1' then reg1 := a;
end if;
if le_reg2 = '1' then reg2 := b;
end if;
if le_reg3 = '1' then reg3 := c ;
end if;
if le_mult = '1' then reg4 := reg1 * reg2;
end if ;
if le_add = '1' then reg5 := reg4 + reg3;
end if
if le_z = '1' then output <= reg5 ;
end if;
end process;
```

Behavior at each clock edge is specified

Area = 1400 gates, Latency = 300ns

```
Fsm: process(clk, reset);
begin
if clk'event and clk ='1' then
ir reset = '1' then cur_state <= s0;
else cur_state <= next_state;
end if; end process;
state_machine:
process (cur_state, reset)
begin
case cur_state is
when s0 => next_state <= s1;
le_reg1 <= '1';
when s1 => next_state <= s2;
le_reg2 <= '1';
when s2 => next_state <= s3;
le_reg3 <= '1';
when s3 => next_state <= s4;
le_mult <= '1';
when s4 => next_state <= s5;
le_add <= '1';
when s5 => next_state <= s0;
le_z <= '1'; end case; end process;
```

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).



Characteristics of RTL Synthesis



● Pros

- Allows capture of a digital design at the RTL level in VHDL - improving productivity over logic synthesis tools
- Allows manual mapping to libraries of high-level components (multipliers, adders)
- More control over the synthesis process in terms of final architecture
- Provide several templates for VHDL semantics for state machine optimization
- IEEE RTL VHDL standard, '97
- Supports a large subset of VHDL

● Cons

- Up until 1997, each vendor supported a different RTL subset of VHDL
- Requires specification of the datapath, registers, controller, and cycle-by-cycle behavior
- Resource sharing, resource allocation, scheduling, and mapping tasks have to be carried out by the designer prior to coding at the RTL level, limiting architectural exploration.
- Allows no architectural exploration, and the synthesizer optimizes at the level of the components and states

Copyright © 1995-1999 SCRA

41

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



Logic Synthesis



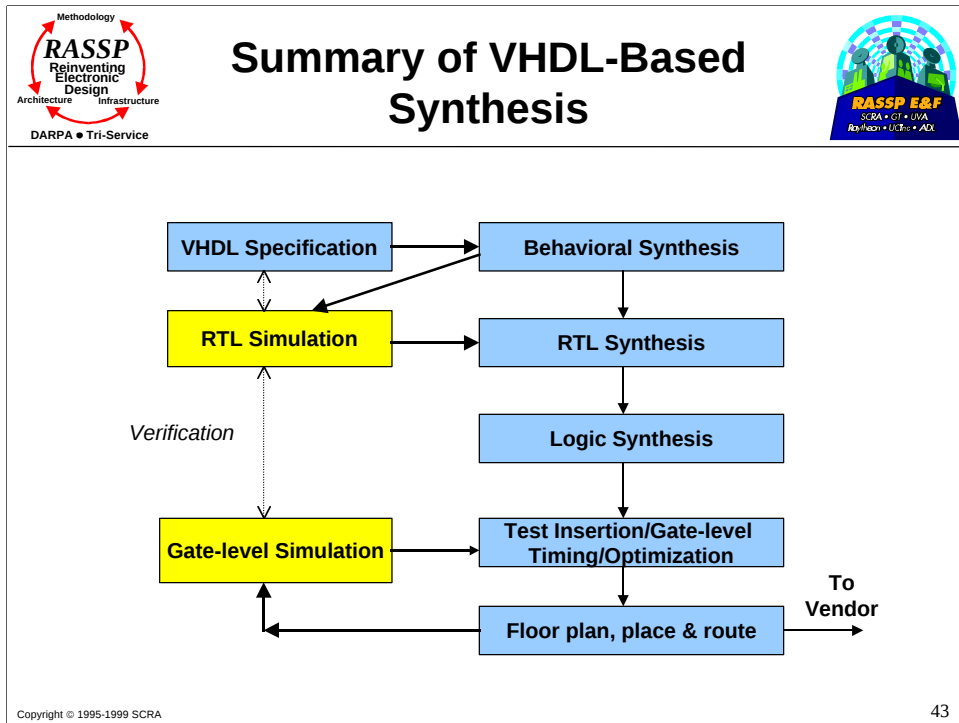
- **Logic synthesis optimizes the Boolean equations generated by RTL synthesizers and maps them to technology specific gate-level implementations utilizing detailed functional and timing information from technology libraries.**
- **The operations in the logic synthesis process include multilevel minimization, factorization, and equation flattening so that area, power, and timing metrics are optimized in some manner.**
- **Few EDA vendors, such as Synopsys, do not distinguish clearly between the tasks of RTL synthesis and logic synthesis.**
- **The techniques for logic synthesis are well understood within the design community, and usually taught at universities in the first digital design course.**

Copyright © 1995-1999 SCRA

42

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.

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

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.

IEEE Support for Synthesis

- **IEEE Std 1076-1993** VHDL Language Reference Manual (LRM)
- **IEEE Std 1164-1993**, IEEE Standard Multivalued Logic System for VHDL Model Interoperability (STD_LOGIC_1164)
- **IEEE Std 1076.3-1997** IEEE Standard Synthesis Packages (NUMERIC_BIT and NUMERIC_STD).
- **IEEE Std 1076.6-1999** IEEE VHDL Register Transfer Level (RTL) Synthesis

These two standards will be introduced in this synthesis module.

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.



VHDL Packages for Synthesis Arithmetic Packages



- All synthesis tools support some type of arithmetic packages
- Synopsys developed packages based on std_logic_1164 package - supported by many other synthesis tools
 - std_logic_arith
 - std_logic_signed
 - std_logic_unsigned
- Actel synthesis tools support their own package
 - asyl.arith
- IEEE has developed standard packages for synthesis IEEE Std. 1076.3
 - Numeric_Bit
 - Numeric_Std

Copyright © 1995-1999 SCRA
46

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.



IEEE Synthesis Packages



- **IEEE Standard VHDL Synthesis Packages (1076.3 NUMERIC_STD and 1076.3 NUMERIC_BIT) define numeric types and arithmetic/logic functions for use by synthesis tools. Vendors may *not* change the specified interfaces or simulation descriptions of the package declarations. Interestingly, IEEE 1076.6 Synthesis RTL subset does not support some operations, e.g., sra specified in IEEE 1076.3.**
- **NUMERIC_BIT replaces STD_LOGIC in NUMERIC_STD with BIT and BIT_VECTOR (so the following declarations are similar).**

Two primary “vector” data types defined by IEEE 1076.3 NUMERIC_STD:
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
Signed numbers are represented in *two’s complement* form

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.

VHDL Packages for Synthesis Base Types

- Standard bit types may be used
- Typically IEEE Std. 1164 types are used

○ std_ulogic type

USE IEEE.std_logic_1164.ALL;

□ Values 'U', 'X', 'W', and '-' are called metalogical values for synthesis

```
TYPE std_ulogic IS ( 'U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0
                    '1', -- Forcing 1
                    'Z', -- High Impedance
                    'W', -- Weak Unknown
                    'L', -- Weak 0
                    'H', -- Weak 1
                    '-' -- Don't care
                    );
```

○ std_logic type - resolved std_ulogic type

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.

VHDL Packages for Synthesis Base Types (Cont.)

- The `std_logic_1164` package also contains:
 - Vectors of `std_ulogic` and `std_logic`
 - Subtypes of `std_logic` - `X01`, `X01Z`, `UX01`, `UX10Z`
 - Logic functions with various arguments - `std_ulogic`, `std_logic`, `std_logic_vector`

```
FUNCTION "and" (l,r : std_ulogic;) RETURN UX01;  
FUNCTION "nand" (l,r : std_ulogic;) RETURN UX01;  
FUNCTION "or" (l,r : std_ulogic;) RETURN UX01;  
FUNCTION "nor" (l,r : std_ulogic;) RETURN UX01;  
FUNCTION "xor" (l,r : std_ulogic;) RETURN UX01;  
FUNCTION "xnor" (l,r : std_ulogic;) return ux01;  
FUNCTION "not" (l,r : std_ulogic) RETURN UX01;
```

- Conversion functions

```
FUNCTION To_bit(s:std_ulogic) RETURN bit;  
FUNCTION To_bitvector(s:std_ulogic_vector) RETURN bit_vector;  
FUNCTION To_StdULogic(b:bit) RETURN std_ulogic;
```

We continue with a description of supported types.

VHDL Packages for Synthesis Base Types (Cont.)

○ Clock edge functions

```
FUNCTION rising_edge (SIGNAL s:std_ulogic) RETURN boolean;  
FUNCTION falling_edge (SIGNAL s:std_ulogic) RETURN boolean;
```

○ Unknown functions

```
FUNCTION Is_X (s:std_ulogic_vector) RETURN boolean;  
FUNCTION Is_X (s:std_logic_vector) RETURN boolean;  
FUNCTION Is_X (s:std_ulogic) RETURN boolean;
```

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.



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

Copyright © 1995-1999 SCRA

51

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.



IEEE RTL VHDL Subset



- The standard for VHDL Register Transfer Level Synthesis is IEEE Std 1076.6-1999.
- The purpose of the standard is to define a syntax and semantics for VHDL RTL synthesis. It defines a subset of IEEE 1076 (VHDL) that is intended to be used in common by all RTL synthesis tools. This will allow designers to produce well-defined designs whose functional characteristics are independent of any particular synthesis environment.
- Compliant tools may have features above and beyond those required by the standard.

Copyright © 1995-1999 SCRA

52

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



IEEE 1076.6 Definitions



- **Model Compliance** - A VHDL model shall be defined to be compliant to IEEE 1076.6 if the model:
 - uses only constructs described as *supported* or *ignored* by IEEE 1076.6
 - adheres to the semantics prescribed by this standard.
- **Tool Compliance** - A synthesis tool shall be defined as being compliant with this standard if the tool:
 - accepts all models that adhere to the model compliant definition above
 - supports language related pragmas defined by the standard
 - conforms to the verification requirements (Clause 5) of the standard.

Note: Shall indicates that mandatory requirements are to be strictly followed in order to be compliant with the standard with no deviation allowed. Should indicates that a certain course of action is preferred but not necessarily required. May indicates that a course of action is permissible within the limits of the standard.

Copyright © 1995-1999 SCRA

53

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.



Terminology



- A synthesis tool interprets a VHDL construct if it produces something that represents the construct. A synthesis tool accepts a VHDL construct if it allows that construct to be a legal input. A synthesis tool is not required to interpret every VHDL construct it accepts, but *only* required to interpret those constructs required by the IEEE Synthesis standard 1076.6.
- Constructs are classified into the following:
 - Supported: RTL synthesis shall interpret a construct, i.e., *map* it to hardware
 - Ignored: RTL synthesis shall ignore the construct. Encountering this construct shall not cause synthesis to fail, but the synthesis results may not match the simulation (verification) results. The RTL synthesis tool may inform the user that the construct is not defined by the standard.
 - Not Supported: RTL synthesis does not support the construct. RTL synthesis does not expect to encounter the construct and the failure mode shall be undefined.

Copyright © 1995-1999 SCRA

54

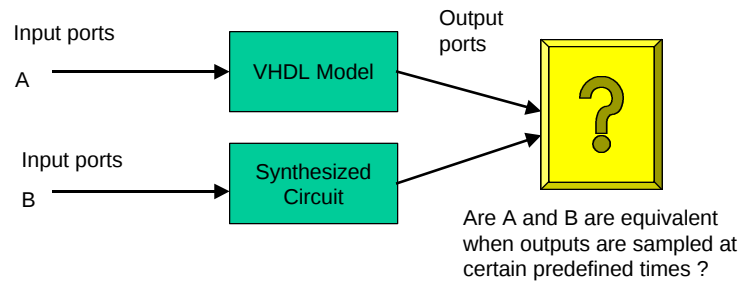
This slide describes the context within which the IEEE RTL Synthesis subset standard operates.

Predefined Synthesis Types

- A compliant synthesis tool shall support the following types:
 - BIT, BOOLEAN and BIT_VECTOR (defined in IEEE VHDL LRM)
 - INTEGER (defined in IEEE VHDL LRM)
 - STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR (defined in IEEE Std 1164-1993)
 - SIGNED and UNSIGNED (defined in NUMERIC_STD in IEEE 1076.3)
 - SIGNED and UNSIGNED (defined in NUMERIC_BIT IEEE 1076.3)
- In addition, the synthesis tool shall support user-defined types and derived types according to a set of rules.
- If a type with a metalogical value (U, W, X, ..) is used in a VHDL model, it will have an ancestor type that belongs to IEEE 1164-1993.
- If a VHDL construct is struck out (e.g., ~~file~~) it is *not supported* by synthesis tools, and if underscored (e.g., assert) it is *ignored*.

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

● General IEEE 1076.6 Verification Methodology



Verification (Cont.)

- **Combinational circuits**

- apply inputs
- wait for settling time
- compute outputs
- wait for transients to settle
- sample and measure outputs
- oscillatory behavior is disallowed.

- **Sequential circuits**

- same procedure as combinational circuits, except that checking should be done just before the *active* clock edge. Sufficient time must be allowed for all transients to settle down.
- For level-sensitive models other methods may be followed.

Note: Input data shall not contain unknowns or metalogical values, and must take into account set up and hold times. The input stimulus may also need to reset internal storage elements to specific logical values before comparison.

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

VHDL RTL Keywords

abs	Buffer	Exit	<i>Inertial</i>	Next
access	<i>bus</i>		inout	nor
after		<i>file</i>	is	not
<u>alias</u>	case	for		null
all	component	function	<i>label</i>	of
and	configuration		<i>library</i>	on
architecture	constant	generate	linkage	open
array		generic	literal	or
assert	<u>disconnect</u>	<i>group</i>	loop	others
attribute	downto	<i>guarded</i>		out
	else		map	
begin	elsif	if	mod	package
block	end	<i>impure</i>		port
body	entity	in	nand	<i>postponed</i>
			new	

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

All the keywords used in VHDL are depicted in this slide and the following slide. The supported keywords for synthesis (1076.6) are shown 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	<i>shared</i>	use
process	signal	
<i>pure</i>	<i>sla</i>	variable
	<i>sll</i>	
range	<i>sra, srl</i>	wait
record		when
register	subtype	
<i>reject</i>		
rem	then	<i>while (in loop)</i>
report	to	<i>with</i>
return	<u>transport</u>	
<i>rot</i>	type	<i>xnor</i>
<i>ror</i>		xor
	<i>unaffected</i>	
select	units	
<u>severity</u>	until	

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



VHDL RTL Syntax Key



Design Entities Entities & Architectures Packages Package body Configuration Subprograms	Specifications attributes configurations names	Expressions arithmetic logic comparison shift , resize	Types scalar composite access file
Declarations Type & subtype Objects Attributes Components Group	Sequential Constructs wait report , <u>assert</u> signal assignment procedure if, case, loop & next	Concurrent Statements process block concurrent procedure call concurrent assertion concurrent signal assignment component instantiation generate statement	

Note: many allowed features are supported partially.

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

Copyright © 1995-1999 SCRA

60

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.

VHDL RTL Types

Scalar

Enumeration
Integer
Physical
Floating point

Composite

Array
Record

Access

File

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

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

VHDL RTL Synthesis Scalar Type

● Enumeration

- Enumeration types shall be supported, and BIT, BOOLEAN, and STD_ULOGIC will be mapped to single bits. Severity_level is ignored
- The user may override default mapping of enumerated types through use of enum_encoding attribute. File_open_kind, file_open_status are not supported.

Example: One-hot coding
 attribute enum_encoding: string;
 type snack is (donut, biscuit, idli, cake);
 attribute enum_encoding of snack:
 type is "1000 0100 0010 0001" ;

● Integer

- integer types are supported and subtypes NATURAL and POSITIVE are supported
- Recommendation that tool should convert integer type to bit_vector during implementation. If range has only positive values, use UNSIGNED, if negative/positive use SIGNED representation in NUMERIC_BIT.
- Integer ranges are synthesized as if 0 is included in range.

Example: Integer range
 integer range 8 to 10;
 -- is synthesized the same way (4 bits) as
 integer range 0 to 15;

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

RTL Scalar Types

● Physical Types

- Physical types, other than TIME, shall not be supported.
- Declarations of objects (signals and variables) of type TIME shall be ignored.
- Reference to objects of type TIME may only occur within constructs such as the after clause in VHDL.

● Floating Point Types

- Floating point types declarations shall be ignored.
- Reference to objects of type floating point may only occur within constructs such as the after clause in VHDL

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

Composite Types

● Array

- Array type definition is supported
- constrained and unconstrained arrays are supported
- index of array shall contain exactly one discrete range --- bounds of the index range shall be specified directly or indirectly as static values of integer type
- null ranges are not supported, and range shall return integer values
- predefined array types shall be supported.

● Record

- Record type shall be supported

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

RTL VHDL Declarations

- Type declarations
 - Subtype declarations
- Object declarations
 - Constant declarations
 - Variable declarations
 - Signal declarations
 - File declarations
 - Interface declarations
 - Alias declarations
- Attribute declarations
- Component declarations
- **Group template declarations**
- **Group declarations**
- Configuration declaration
- Package declaration
- Entity declaration
- Subprogram declaration

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

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.



Type Declarations



- **Declarations of type and subtype shall be supported**
- **Incomplete type declarations shall not be supported**
- **User-defined resolution functions shall be ignored.**
- **Full access type declarations or file type definition shall not be supported.**

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

Object Declarations

- **Constant declarations**
 - Constant declaration shall be supported
 - Deferred constant declaration shall not be supported
- **Signal declarations**
 - Signal declarations shall be supported
 - signal_kind (register, bus) shall not be supported
 - initial value expressions shall be ignored
 - if signal is declared in package, it must have an initial value
 - assignment to signal declared in package shall not be supported
- **Variable declarations**
 - Variable declaration shall be supported
 - initial values shall be ignored
 - shared variables shall not be supported
- **Interface (port) declarations**
 - interface signal, constant and variable declarations supported
 - interface file declarations ignored
 - linkage and bus modes not supported
- **Alias declarations**
 - the following attribute declarations will be 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.

Component Declarations

- **Component declarations shall be supported with the exception of the reserved word “is” as follows:**

```
Component identifier is  
    [ local_generic_clause]  
    [ local_port_clause]  
end component [ component_simple_name ];
```

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

RTL VHDL Expressions & Operators

● Operators

- Logical Operators: `and`, `or`, `nand`, `nor`, `xor`, `xnor`
- Relational Operators: `=`, `/=`, `<`, `<=`, `>`, `>=`
- Shift Operators: `sll`, `srl`, `sla`, `sra`, `rol`, `ror`
- Adding Operators: `+`, `-`, `&`
- Sign Operators: `+`, `-`
- Multiplying Operators: `*`, `/`, `mod`, `rem`
- Miscellaneous Operators: `**`, `abs`, `not`

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

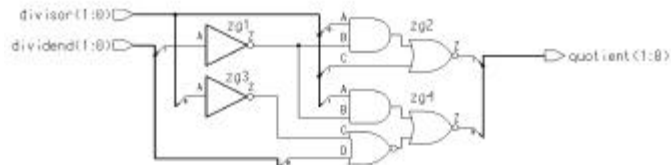
- Generally, if the `numeric_bit` and `numeric_std` packages are supported, the operators within them are supported

- 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;
```



Copyright © 1995-1999 SCRA

71

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

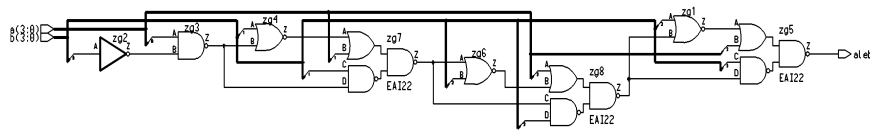
Operators (Cont.)

● Comparison operators - ">", "<", "<=", ">=", "=", "/="

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

ENTITY compare is
  PORT(a : IN unsigned(3 DOWNT0 0);
       b : IN unsigned(3 DOWNT0 0);
       aleb : OUT boolean);
END compare;

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



Copyright © 1995-1999 SCRA

72

The VHDL code describes how a comparison operator is synthesized.

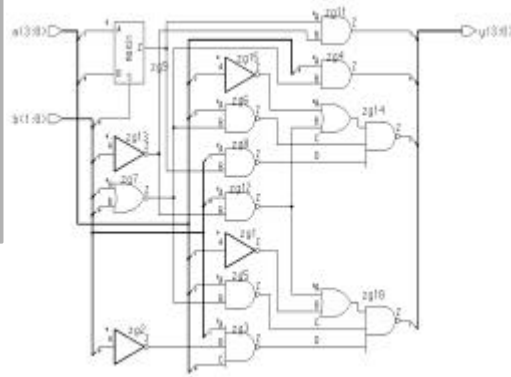
Operators (Cont.)

- 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 DOWNT0 0);
       b : IN unsigned(1 DOWNT0 0);
       y : OUT unsigned(3 DOWNT0 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.

Sequential Wait Statement

The wait statement

```
wait on signal_name ;  
  
wait until boolean_condition ;  
  
wait for time_expression;  
  
label: wait until boolean_condition;
```

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

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.

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

Variable Assignment

Variable_assignment_statement ::= [***label*** :] target := expression;

- ***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.

If Statement in Synthesis

```
If_statement :: = [ if_label: ]  
               if condition then  
                 sequence_of_statements  
               { elsif condition then  
                 sequence_of_statements }  
               [ elsif  
                 sequence_of_statements ]  
               end if [ if_label ] ;
```

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

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.

Case Statement in Synthesis

```
Case_statement ::= [ case_label : ]  
                 case expression is  
                 case_statement_alternative  
                 { case_statement_alternative }  
                 end case [ case_label ] ;  
  
case_statement_alternative ::=  
    when choices => sequence_of_statements
```

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

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.

Loop Statement in Synthesis

```
Loop_statement ::= [ loop_label: ]  
                 [ iteration_scheme ] loop  
                 sequence_of_statements  
                 end loop [ loop_label ] ;  
  
iteration_scheme ::= while condition  
                   | for loop_parameter_specification  
  
parameter_specification ::= identifier is discrete_range  
  
discrete_range ::= discrete_subtype_indication | range
```

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

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

Loop Examples in Synthesis

```
-- following errors are examples of correct VHDL code
-- that is not supported for RTL synthesis (IEEE 1076.6)
Type states_typ is (idle, busy, waiting);
process:
variable count v : natural := 0;
begin
-- loop counter is implicitly defined of states_typ ;
-- implicit definition is not supported for synthesis
for states_i in states_typ loop
-- loop statements here
end loop;
-- max_v is example of dynamic range - not supported
for count_i in 1 to max_v loop
-- loop statements here
-- loop parameters should be of integer type
end loop;
end process;
```

Errors

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

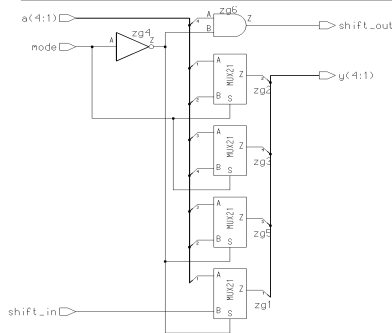
The errors are produced due to incorrect specification in the synthesis subset (even though the simulation may work correctly). The reasons for the errors are described in the comments within the code.

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

Concurrent Statements

- Block **statement**
- Process **statement**
- **Concurrent** procedure call
- **Concurrent** assertion
- **Concurrent** signal assignment **statement**
- Component instantiation **statement**
- Generate **statement**

• ***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.

Block Statement in Synthesis

```
Block_statement ::= block_label: block [ (guard_expression) ] [ is ]  
                block_header  
                block_declarative_part  
                begin  
                block_statement_part  
                end block [ block_label ];  
  
block_header ::= [ generic_clause [ generic_map_clause; ] ]  
                [ port_clause [ port_map_clause; ] ]  
block_declarative_part ::= { block_declarative_item }  
block_statement_part ::= { concurrent_statement }
```

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

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

Process Statement in Synthesis

```

Process_statement ::=
[ process_label: ] [ postponed ] process [ (sensitivity_list) ] [ is ]
  process_declarative_part
begin
  process_statement_part;
end process [ process_label ];
process_declarative_part ::= { process_declarative_item }
process_declarative_item ::=
  | subprogram_declaration
  | subprogram_body
  | type_declaration | subtype_declaration | constant_declaration
  | variable_declaration | file_declaration | alias_declaration
  | attributed_declaration | attribute_specification | use_clause
  | group_template_declaration | group_declaration
process_statement_part ::=
  { sequential_statement }

```

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

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

Process Examples

The use of the process statement in the synthesis subset follows the template below

Example: Process with clock and asynchronous set-reset

```
process ( <clock_signal> , <asynchronous_signals> )  
  <declarations>  
  if <condition1> then <sequential_statements>  
  elsif <condition2> then <sequential_statements>  
  else if <clock_edge> then <sequential_statements>  
  end if ;  
end process;
```

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.

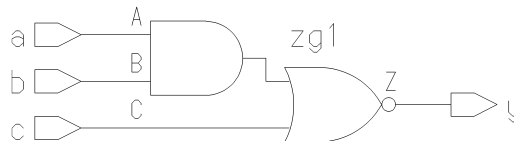
Note: asynchronous signals have higher priority and are level sensitive. A clock edge appears only in the *last else if* statement. The sequential statements cannot contain any **if** statement sensitive to the clock, or any **wait** statement.

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;
```



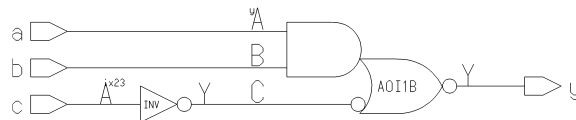
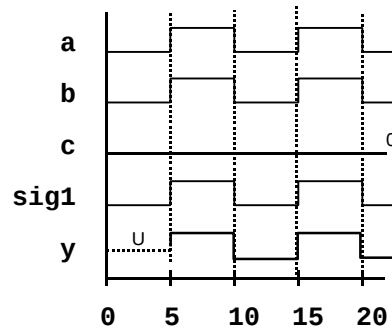
This example shows the code containing a synthesizable process statement and the result of the synthesis process (in this case a combinational circuit).

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.

Sequential Signal Assignment Statements

- **Various types of signal assignment statements inside a process statement (sequential signal assignment statements) are supported**
 - IF statements
 - Case statements
 - Loop statement
 - Only For loops supported
 - Bounds must be specified as static values of an integer type
 - Exit and Next statements supported (without labels)

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

Concurrent Signal Assignment

Concurrent signal assignments shall be supported with the following restrictions. Any **after** clauses shall be ignored. Multiple waveforms are not supported, and clock-related edge specifications shall not be allowed in concurrent signal assignments. **Unaffected** is not supported. **Transport**, **guarded** and other delay_mechanisms are ignored. Two types of concurrent assignments are allowed with some restrictions: *conditional* and *selected*. A conditional select is equivalent to a “**if process**”, while a selected signal assignment is equivalent to a “**case process**”.

Example: Signal assignments

```
architecture sig_a of sig_nty is  
begin  
  B(6) <= A(4);  
  B(2 downto 0) <= A(8 downto 6);  
  B(8) <= A(0) after 10 ns;  
end sig_a;
```

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

Signal Assignment Examples

● Conditional assignment

Conditional waveforms should not reference elements of target signal, and the *last when* condition is not supported.

Example: "Last when" statement
architecture examp_a **of** examp_nty **is**
begin
 siga_s <= B **when** A(0) = '1' **else**
 <= not B **when** A(1) = '1' **else**
 "0000" **when** A(2) = '1' **and** reset = '1' **else**
 (others => ('1'));
end examp_a;

● Selected assignment

Selected signal assignments allow a *last when*, but the other restrictions are similar to conditional assignment.

Example: "selected signal"
architecture examp_a **of** examp_nty **is**
begin
with A **select**
 siga_s <= B **when** "000000",
 not B **when** "101010",
 (others => ('1')) **when** "111111",
 not A **when** others;
end examp_a;

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;

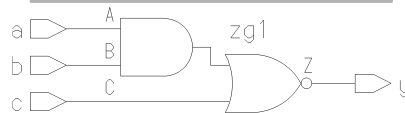
ARCHITECTURE behavior OF csa IS
    BEGIN
        y <= a NOR b;
    END behavior;
```



```
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;
```



Copyright © 1995-1999 SCRA

94

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

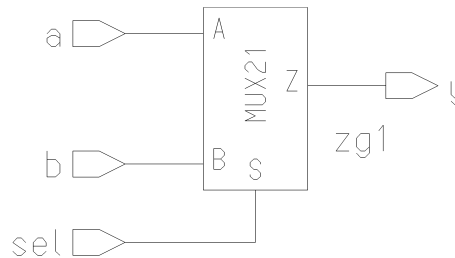
Conditional Signal Assignment Statements

- **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.

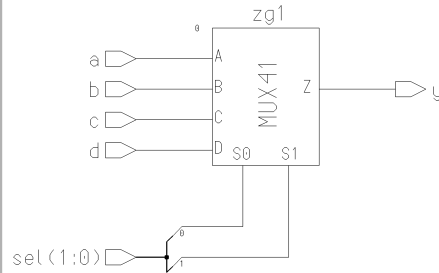
Selected Signal Assignment Statements

● 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.

Concurrent Procedure Call

Concurrent procedure call statement ::=
[label:] [***postponed***] procedure call;

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

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

- **Structural VHDL is supported**
 - Component declarations
 - Binding Indications
 - Component instantiations
- **Lower level components must be made of synthesizable constructs**

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.

Component Instantiation

```
Component_instantiation_statement ::=
    instantiation_label: [ component ] component_name
    [ generic_map_aspect ]
    [ port_map_aspect ] ;
```

Example: Use of components in synthesis

```
architecture version_a of version_nty is
    component counter_nty
    generic
        (mod_g : integer;
         countdly_g : integer);
    port (reset_p : in bit;
          clock : in bit) ;
    end component;
    signal clock, reset_p: bit;
```

```
Begin
    c1_counter: counter_nty
    generic map
        (mod_g => 4, countdly_g =>
         6);
    port map
        (reset_p => reset_p,
         clock => clock);
    end version_a;
```

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

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

Structural VHDL And-Or-Invert Example

● And gate, Or gate, and Inverter leaf cells

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY and2 is
  PORT(a : IN std_logic;
        b : IN std_logic;
        c : OUT std_logic);
END and2;

ARCHITECTURE behav OF and2 IS
BEGIN
  c <= a and b;
END behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY or2 is
  PORT(a : IN std_logic;
        b : IN std_logic;
        c : OUT std_logic);
END or2;

ARCHITECTURE behav OF or2 IS
BEGIN
  c <= a or b;
END behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

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

ARCHITECTURE behav OF inv IS
BEGIN
  b <= not a;
END behav;
```

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

Structural VHDL And-Or-Invert Example

- **And-Or-Invert structural description**
 - Entity
 - Architecture - component declarations

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY aoi2_str is
  PORT(a : IN std_logic;
        b : IN std_logic;
        c : IN std_logic;
        d : OUT std_logic);
END aoi2_str;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY aoi2_str is
  PORT(a : IN std_logic;
        b : IN std_logic;
        c : IN std_logic;
        d : OUT std_logic);
END aoi2_str;

ARCHITECTURE structural OF aoi2_str IS

  COMPONENT and2
    PORT(a : IN std_logic;
          b : IN std_logic;
          c : OUT std_logic);
  END COMPONENT;

  COMPONENT or2
    PORT(a : IN std_logic;
          b : IN std_logic;
          c : OUT std_logic);
  END COMPONENT;

  COMPONENT inv
    PORT(a : IN std_logic;
          b : OUT std_logic);
  END COMPONENT;
```

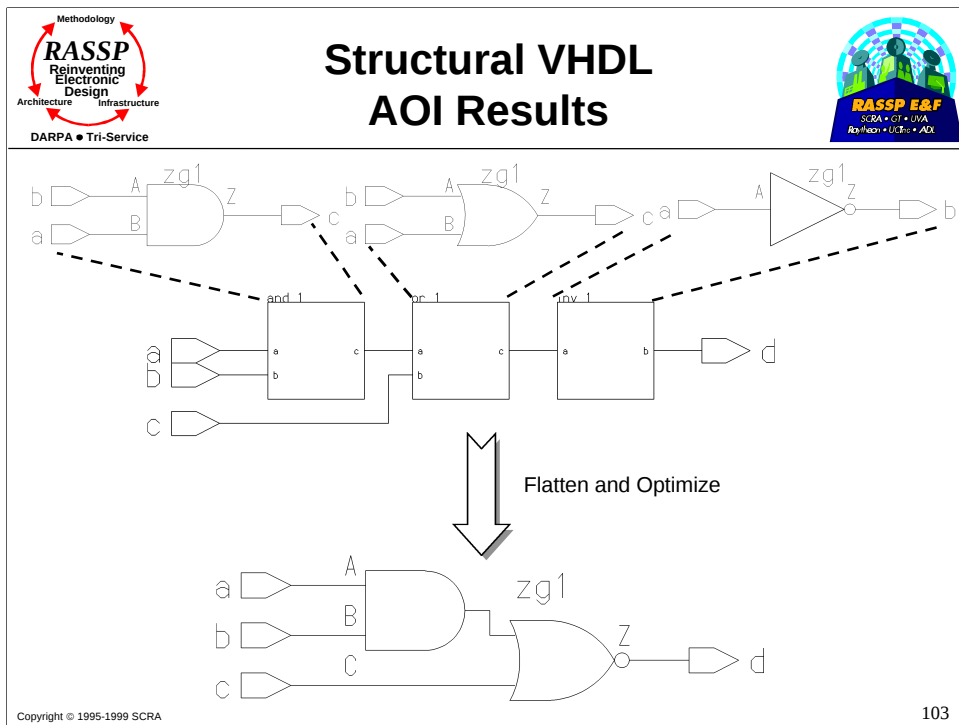
The components described in the previous slide are used in this slide as an example of structural VHDL.

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

Entity Syntax in Synthesis

```
Entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin entity_statement_part ] end [ entity ]
        entity_simple_name;
```

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 **constants**)

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

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

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 **constants**)

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

Architecture Syntax

```
Architecture_body ::= architecture identifier of entity_name is
    architecture_declarative_part
begin
    [ architecture_statement_part ]
end [ architecture ] [ architecture_simple_name ] ;
```

```
Architecture_declarative_part ::= { block_declarative_item }
block_declarative_item ::=
    subprogram_declaration | subprogram_body | type_declaration
    constant_declaration | signal_declaration | shared_variable_declaration
    file_declaration | alias_declaration | component_declaration | attribute_declaration
    configuration_specification | disconnect_specification | group_template_declaration
    group_declaration
```

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

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

Configuration Syntax

```
Configuration_declaration ::=  
  configuration identifier of entity_name is  
    configuration_declarative_part  
  block_configuration  
end [ configuration ] [ configuration_simple_name ] ;
```

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.

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

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.

Subprograms

```
Subprogram_declaration ::=  
    subprogram_specification ;  
subprogram_specification ::=  
    procedure_designator [ { formal_parameter_list } ]  
    | [ pure | impure ] function_designator [ { formal_parameter_list } ]  
    return_type_mark  
designator ::= identifier | operator_symbol  
operator_symbol ::= string_literal
```

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

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

- **Procedures and Functions are supported - with limitations to allowed statement types**
 - Procedures and functions may be in a package or in the declarative part of the architecture

```
LIBRARY ieee;  
USE ieee.std_logic_1164.all;  
USE ieee.numeric_std.all;  
  
PACKAGE logic_package IS  
  
    FUNCTION majority(in1, in2, in3 : std_logic) RETURN std_logic;  
  
    PROCEDURE decode(SIGNAL input  : IN std_logic_vector(1 DOWNTO 0);  
                     SIGNAL output : OUT std_logic_vector(3 DOWNTO 0));  
  
END logic_package;
```

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

Procedures and Functions (Cont.)

```
PACKAGE BODY logic_package IS

FUNCTION majority(in1, in2, in3 : std_logic) RETURN std_logic IS
VARIABLE result : std_logic;
BEGIN
  IF((in1 = '1' and in2 = '1') or (in2 = '1' and in3 = '1') or
    (in1 = '1' and in3 = '1')) THEN
    result := '1';
  ELSIF((in1 = '0' and in2 = '0') or (in2 = '0' and in3 = '0') or
    (in1 = '0' and in3 = '0')) THEN
    result := '0';
  ELSE
    result := 'X';
  END IF;
  RETURN result;
END majority;
```

Example of a majority function.

Procedures and Functions (Cont.)

```
PROCEDURE decode(SIGNAL input : IN std_logic_vector(1 DOWNTO 0);  
                  SIGNAL output : OUT std_logic_vector(3 DOWNTO 0)) IS  
BEGIN  
  CASE input IS  
    WHEN "00" =>  
      output <= "0001";  
    WHEN "01" =>  
      output <= "0010";  
    WHEN "10" =>  
      output <= "0100";  
    WHEN "11" =>  
      output <= "1000";  
    WHEN OTHERS =>  
      output <= "XXXX";  
  END CASE;  
END decode;  
  
END logic_package;
```

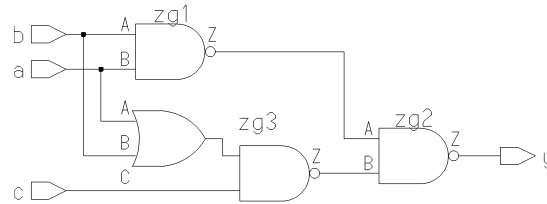
Example of a decode function within a package.

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.)

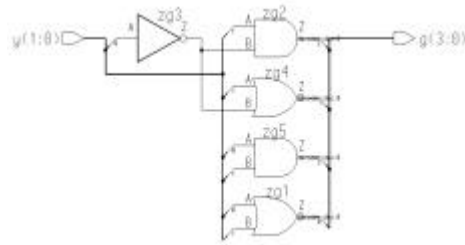
```

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.

Package Syntax

```
Package declaration ::=  
  package identifier is package_declarative_part  
  end [ package ] [ package_simple_name ] ;  
  
package_declarative_part ::= { package_declarative_item }  
package_declarative_item ::=  
  subprogram declaration | type declaration | subtype declaration  
  constant declaration | signal declaration | shared variable declaration  
  file declaration | alias declaration | component declaration |  
  attribute declaration | attribute specification | disconnect specification  
  use clause | group declaration | group template declaration
```

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

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.



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

Copyright © 1995-1999 SCRA

116

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.



Combinational Circuits and Logic



- This section describes the use of the RTL subset in synthesis of
 - **Combinational circuits and logic**
 - Register definition and synchronous behavior
 - Expressions, operators, and types
 - Synchronous sequential circuits
 - State machines and controllers
 - Memories

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.

VHDL in Combinational Circuits

- **Concurrent constructs**

- Selected signal assignment
- Conditional signal assignment

- **Sequential constructs**

- Process **statement**
- case **statement**
- if **constructs**
- loop (**mainly** for-loop)
- while (**not common**)

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..).

Examples of Logic/Arithmetic

```

Architecture logic_a of logic_e is
  signal s1_s, s2_s: std_logic;
begin
  process (A_p, B_p)
    -- A_p and B_p are input ports of type unsigned
    variable v1_v: unsigned (1 downto 0);
  begin
    v1_v := ((A_p(0) nor A_p(1), B_p(0) nor B_p(1));
    s1_s <= v1_v (0) nor v1_v(1);
  end process;

  process (A_p, B_p)
  begin
    Y2 <= A_p + B_p + (A_p * B_p);
  end process;
  Y1 <= s1_s;
end logic_a;

```

sensitivity list includes
all objects read in process

Arithmetic is inferred

Concurrent assignment

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.

Multiplexer/Demultiplexer (Cont.)

- Using selected signal assignment

```
Architecture sel_a of mux_e is
begin
with sel_p select
Y <= in1_p when 0,
Y <= in2_p when 1;
end sel_a;
```

- Using conditional signal assignment

```
Architecture cond_a of mux_e is
begin
Y <= in1_p when sel_p = "0" else
in2_p when sel_p = "1" else
(others => ('1'));
end cond_a;
```

Last **when** statement is not supported, hence "others" is used.

Note two equivalent realization of the mux/demux pairs.

Arithmetic Logic Units (ALU)

```
Architecture vjalu_a of vjalu_e is
begin
  process (sel_p, ain_p, bin_p, cin_p)
    variable logic_v, arith_v: unsigned (4 downto 0);

    begin
      -- logic unit
      case sel_p(1 downto 0) is
        when "00" => logic_v := ain_p AND bin_p;
        when "01" => logic_v := ain_p OR bin_p;
        when others => logic_v := (others => 'X');
      end case;
      -- arithmetic unit
      case sel_p (3 downto 2) is
        when "00" => arith_v := ain_p + bin_p;
        when "01" => arith_v := ain_p * bin_p;
        when others => arith_v := (others => 'X');
      end case;
    end vjalu_a;
```

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.

Synthesis with RTL Subset (Contd.)

- This section describes the use of the RTL subset in synthesis of
 - Combinational circuits and logic
 - Register definition and synchronous behavior
 - Synchronous sequential circuits
 - State machines and controllers
 - Sequential Datapath

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_ULONGIC 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 Edge Specification

● Positive/negative edge clocks

○ if statement can be used as follows

1. If **RISING_EDGE** (clk_signal_name)
2. If clk_signal_name'EVENT **and** clk_signal_name = '1'
3. If clk_signal_name = '1' **and** clk_signal_name'EVENT
4. If **not** clk_signal_name'STABLE **and** clk_signal_name = '1'
5. If clk_signal_name = '1' **and not** clk_signal_name'STABLE

Note: The negative clock edge can be modeled similarly using **FALLING_EDGE** and clk_signal_name = '0' in the above statements.

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.

Clock Edge Specification (Cont.)

● Positive/negative clock edge specification (cont.).

- Using the wait until statement in VHDL, the following statements are equivalent

1. Wait until **RISING_EDGE** (clk_signal_name)
2. Wait until Clk_signal_name = '1'
3. Wait until Clk_signal_name'EVENT and clk_signal_name = '1'
4. Wait until clk_signal_name = '1' and clk_signal_name'EVENT
5. Wait until not clk_signal_name'STABLE and clk_signal_name = '1'
6. Wait until clk_signal_name = '1' and not clk_stable_name'STABLE

The wait statement should be the first statement in the process, which can cause problems when the process is required to be sensitive to other asynchronous inputs as well.

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.

Synthesizing Flip Flops using “if”

● Guidelines for modeling FFs using “if” statement

The template for modeling a positive-edge FF with an “if” statement is

Template:

```
process (<clock signal>)
<declarations>
if <clock edge> then
<sequential statements>
end if;
end process
```

Example:

```
process(clock)
<declarations>
if rising_edge(clock) then
Q <= D;
end if;
end process;
```

Only clock signal
in sensitivity list

Note: D is read before it is written to, so this infers storage required.

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.

Inferring FFs Using “if”

An edge sensitive FF is inferred under the following conditions.

1. Synchronous assignment is defined for *both* signals and variables. However, a synchronous assignment to a signal should model one or more edge-sensitive storage elements. While, for variables, a synchronous assignment *may* (*not* shall or should) model an edge-sensitive FF. Typically, if the value of the variable is read before write, then an edge sensitive FF may be modeled. Note the distinction between *should*, *shall*, and *may*.
2. In case of the if statement, only the clock signal shall be specified in the sensitivity list of the process. No other signals (except asynchronous signals) shall be included in the sensitivity list (as shown in the preceding example).
3. No statements shall precede or succeed the “if” statement in the process.

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.

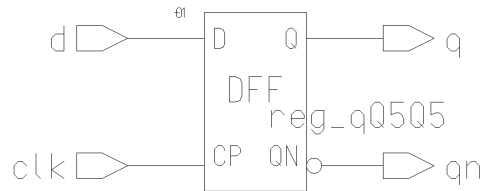
Edge Sensitive D Flip-Flop

- 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;
```



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.)

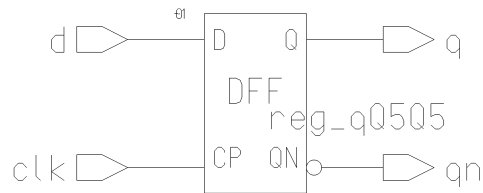
● Wait statement can be used

```
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.

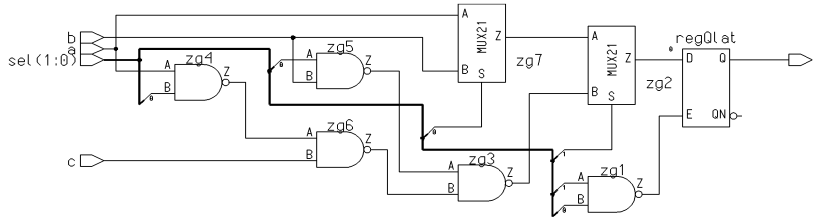
Inferring Latches

- 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;
```



Copyright © 1995-1999 SCRA

133

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

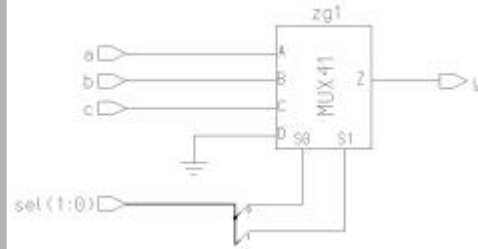
Avoiding Latches

● 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;
```



Copyright © 1995-1999 SCRA

134

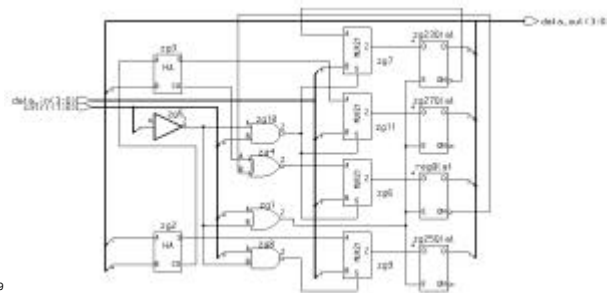
We avoid a latch here (and substitute it with a mux) by enumerating all possibilities for the CASE statement. No storage is necessary as a consequence.

Inferring Latches in Complex Behaviors

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

ENTITY pc_comb IS
    PORT(data_in : IN unsigned(3 DOWNTO 0);
          cntrl : IN unsigned(1 DOWNTO 0);
          data_out : OUT unsigned(3 DOWNTO 0));
END pc_comb;
```

```
ARCHITECTURE rtl OF pc_comb IS
    SIGNAL pc : unsigned(3 DOWNTO 0);
    BEGIN
        one : PROCESS(data_in, cntrl, pc)
            BEGIN
                CASE cntrl IS
                    WHEN "01" => pc <= (pc + "0001");
                    WHEN "10" => pc <= pc;
                    WHEN OTHERS => pc <= data_in;
                END CASE;
            END PROCESS one;
        DATA_OUT <= pc;
    END rtl;
```



Copyright © 1995-1999

135

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 at 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.

Inferring Level-Sensitive Logic

Level sensitive logic *shall* be synthesized under the following conditions:

1. A signal or variable is assigned in a process that does not contain clock edges, and
2. There are executions of the process that do not execute an *explicit* assignment to the signal or variable (requiring it to store older values), and
3. All signals and variables read by the process have well defined values.

Note: When the executions of the process require reading before assignment, then level sensitive logic *may* be synthesized. The process sensitivity list will contain *all* signals read within the process statement.

Lev_sen: **process** (en, D)

```
begin  
if en = '1' then  
  Q <= D;  
end if ;  
end process;
```

Note: The code does not specify what is to be done if en is not '1'.

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

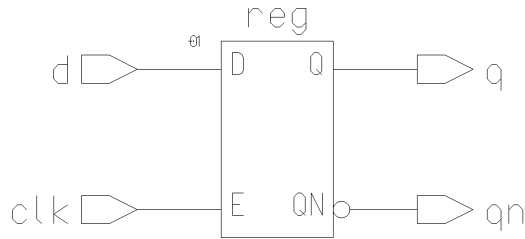
Level Sensitive D Latch

● 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;
```

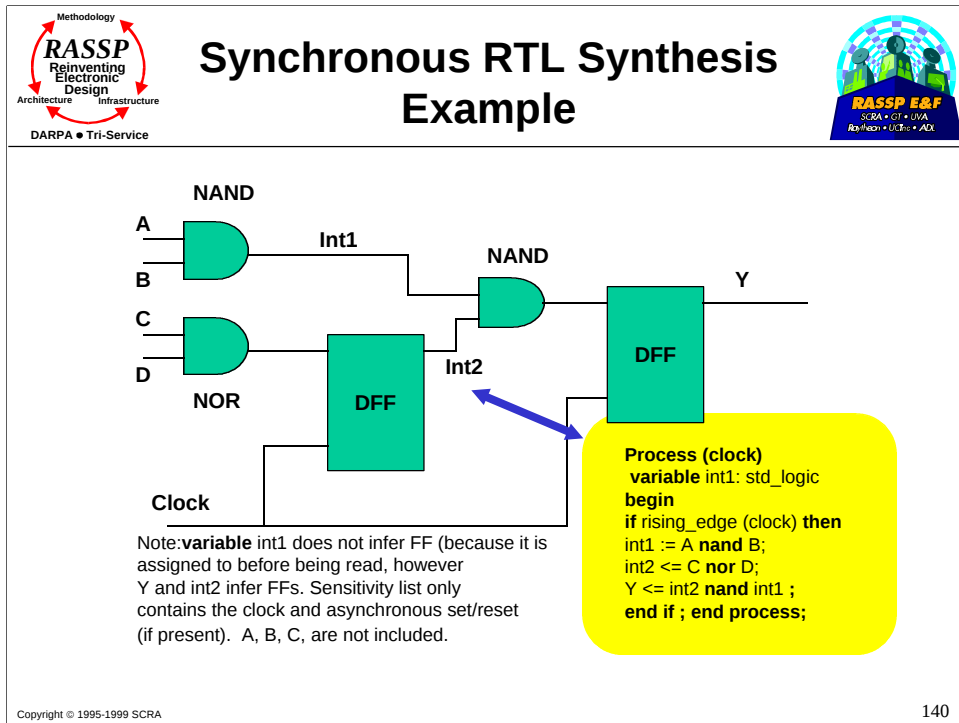


The if statement without the else results in a latch because the complete behavior is not specified, and storage is inferred.

Synthesis with RTL Subset (Contd)

- This section describes the use of the RTL subset in synthesis of
 - Combinational circuits and logic
 - Register definition and synchronous behavior
 - Synchronous sequential circuits
 - State machines and controllers
 - Sequential Datapath

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.

Synthesis with RTL Subset (Contd.)

- This section describes the use of the RTL subset in synthesis of
 - Combinational circuits and logic
 - Register definition and synchronous behavior
 - Synchronous sequential circuits
 - State machines and controllers
 - Sequential Datapath

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

Synthesis of State Machines

- **State machines require specification of the following**
 - state of the machine
 - a clock
 - specification of state transitions
 - inputs and outputs
 - reset and set conditions (synchronous or asynchronous)

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.

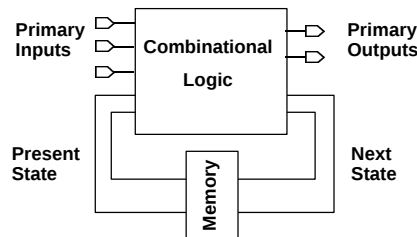
Finite State Machines (FSMs) are widely used to represent designs of controllers, and avionics industry surveys have shown that FSMs with about 20 - 40 states are typical of the design complexities of ASICs

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.

Finite State Machine Synthesis

- State machines can be modeled as a combinational portion and a sequential portion
- Both Mealy and Moore type state machines can be described
- Most synthesis tools employ special algorithms to minimize state machines - thus standard procedures must be used to enable the tool to recognize the state machine

Huffman FSM Model



Copyright © 1995-1999 SCRA

143

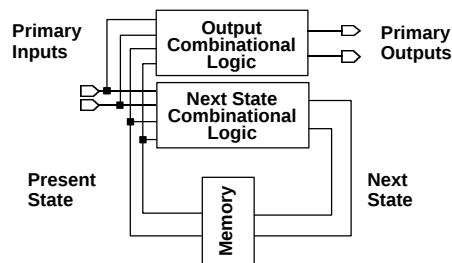
Next, the description of synthesizable state machines will be covered. Synthesis tools typically have special algorithms built in for state machine minimization, so “templates” must be used to ensure that the tool recognizes the state machine and the state variables. Synthesis tools can construct both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

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

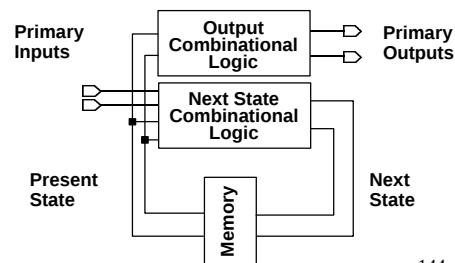
Mealy and Moore State Machine Models

- A state machine has three basic parts
 - Next State Logic
 - Output Logic
 - Memory
- The most straight-forward way to code a synthesizable state machine is to use one process per function

Mealy Machine Model



Moore Machine Model



Copyright © 1995-1999 SCRA

144

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).

State Machine State Encoding

- Use enumerated type to encode state variables

```
TYPE state_type IS (idle,init,test,add,shift);  
SIGNAL present_state, next_state : state_type;
```

- User can choose state encoding type in synthesis tool

- ☐ Sequential
- ☐ One Hot
- ☐ Gray
- ☐ Transition Optimized
- ☐ Others

- enum_encoding attribute can be used to specify encoding directly

```
ATTRIBUTE enum_encoding:STRING;  
TYPE state_type IS (idle,init,test,add,shift);  
ATTRIBUTE enum_encoding OF state_type:TYPE IS "000 100 110 001 011";
```

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 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.

State Machine Memory Process

- **Use flip-flop type process with asynchronous or synchronous reset**
 - State machine **MUST** have reset to function correctly
- **Rising or falling edge may be used**

```
clocked : PROCESS(clk, reset)
BEGIN
  IF (reset = '0') THEN
    present_state <= idle;
  ELSIF (clk'EVENT AND clk = '1') THEN
    present_state <= next_state;
  END IF;
END PROCESS clocked;
```

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.

State Machine Next State Process

```
nextstate : PROCESS(present_state,start,q0)
BEGIN
  CASE present_state IS
    WHEN idle =>
      IF(start='1') THEN
        next_state <= init;
      ELSE
        next_state <= idle;
      END IF;
    WHEN init =>
      next_state <= test;
    WHEN test =>
      IF(q0='1') THEN
        next_state <= add;
      ELSIF(q0='0') THEN
        next_state <= shift;
      ELSE
        next_state <= test;
      END IF;
    WHEN add =>
      next_state <= shift;
    WHEN shift =>
      next_state <= test;
  END CASE;
END PROCESS nextstate;
```

- Next state process forms the heart of the state machine
- Process is sensitive to present state and any necessary inputs
- Most tools prefer a Case statement syntax on which to run their state machine minimization algorithms

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.

State Machine Output Process

```
output : PROCESS(present_state,start,q0)
BEGIN
  -- Default Assignment
  a_enable <= '0' after delay;
  a_mode <= '0' after delay;
  c_enable <= '0' after delay;
  m_enable <= '0' after delay;
  -- State Actions
  CASE present_state IS
    WHEN init =>
      a_enable <= '1' after delay;
      c_enable <= '1' after delay;
      m_enable <= '1' after delay;
    WHEN add =>
      a_enable <= '1' after delay;
      c_enable <= '1' after delay;
      m_enable <= '1' after delay;
    WHEN shift =>
      a_enable <= '1' after delay;
      a_mode <= '1' after delay;
      m_enable <= '1' after delay;
    WHEN OTHERS =>
      NULL;
  END CASE;
END PROCESS output;
```

- Output process computes output values based on the present state and the inputs (Mealy machine) or just on the present state (Moore machine)
- IF or Case statement may be used

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.

State Machine

Putting it all together

```

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY control_unit IS
  PORT(clk      : IN  std_logic;
        q0      : IN  std_logic;
        reset   : IN  std_logic;
        start   : IN  std_logic;
        a_enable : OUT std_logic;
        a_mode  : OUT std_logic;
        c_enable : OUT std_logic;
        m_enable : OUT std_logic);
END control_unit;

```

- Entity
- Architecture declarative part
- Memory process

```

ARCHITECTURE fsm OF control_unit IS
  CONSTANT delay : time := 5 ns ;
  TYPE state_type IS ( idle, init, test, add, shift);
  SIGNAL present_state, next_state : state_type ;
  BEGIN

  clocked : PROCESS(clk, reset)
  BEGIN
    IF (reset = '0') THEN
      present_state <= idle;
    ELSIF (clk'EVENT AND clk = '1') THEN
      present_state <= next_state;
    END IF;
  END PROCESS clocked;

```

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.

State Machine

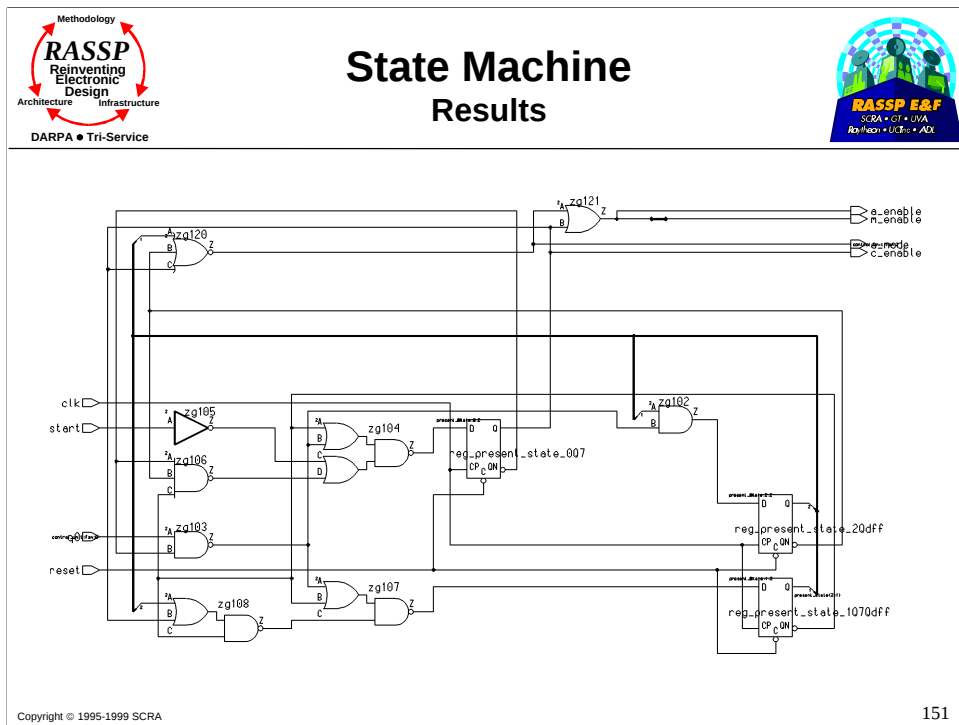
Putting it all together (Cont.)

- Next state process
- Output process

```
nextstate : PROCESS(present_state,start,q0)
BEGIN
  CASE present_state IS
    WHEN idle =>
      IF(start='1') THEN
        next_state <= init;
      ELSE
        next_state <= idle;
      END IF;
    WHEN init =>
      next_state <= test;
    WHEN test =>
      IF(q0='1') THEN
        next_state <= add;
      ELSIF(q0='0') THEN
        next_state <= shift;
      ELSE
        next_state <= test;
      END IF;
    WHEN add =>
      next_state <= shift;
    WHEN shift =>
      next_state <= test;
  END CASE;
END PROCESS nextstate;
```

```
output : PROCESS(present_state,start,q0)
BEGIN
  -- Default Assignment
  a_enable <= '0' after delay;
  a_mode <= '0' after delay;
  c_enable <= '0' after delay;
  m_enable <= '0' after delay;
  -- State Actions
  CASE present_state IS
    WHEN init =>
      a_enable <= '1' after delay;
      c_enable <= '1' after delay;
      m_enable <= '1' after delay;
    WHEN add =>
      a_enable <= '1' after delay;
      c_enable <= '1' after delay;
      m_enable <= '1' after delay;
    WHEN shift =>
      a_enable <= '1' after delay;
      a_mode <= '1' after delay;
      m_enable <= '1' after delay;
    WHEN OTHERS =>
      NULL;
  END CASE;
END PROCESS output;
END fsm;
```

This is the next state process and output process.



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.

VHDL Coding of an FSM: Example

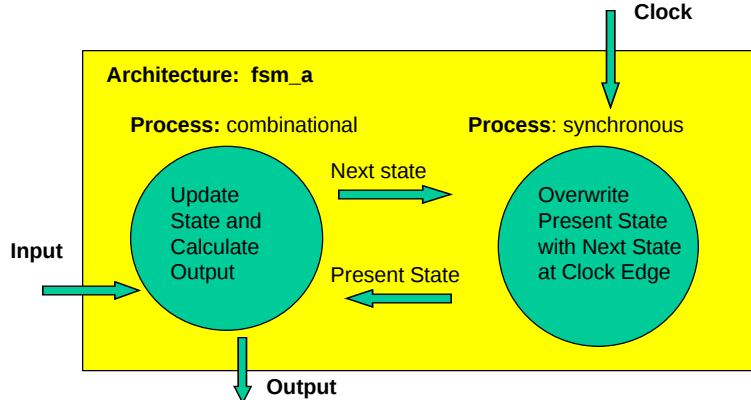
Input	Present State	Next State	Output
0	Q_i	Q_i	0
1	Q_i	Q_i+1	1 (when Q2) else 0

The state machine has four states, one input X, and one output Z

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

Structure of the VHDL Description

The architecture consists of two concurrent processes as shown



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.

Example of an FSM in IEEE RTL VHDL Subset

```

Package states is
type state_typ is (Q0,Q1,Q2,Q3)
end states;
use work.states.all;
entity fsm_nty is
  port (X, clock: in bit; Z: out bit);
end fsm_nty;
architecture fsm_a of fsm_nty is
  signal present_state, next_state: state_typ;
begin
  combinational: process (present_state, X)
  begin
    case present_state is
      when Q0 =>
        if X = '0' then Z <= '0'; next_state <= Q0;
        else Z <= '0'; next_state <= Q1;
        end if;
      when Q1 =>
        if X = '0' then Z <= '0'; next_state <= Q1;
        else Z <= '0'; next_state <= Q2;
        end if;
      when Q2 =>
        if X = '0' then Z <= '0'; next_state <= Q2;
        else Z <= '1'; next_state <= Q3;
        end if;
      when Q3 =>
        if X = '0' then Z <= '0'; next_state <= Q3;
        else Z <= '0'; next_state <= Q0;
        end if;
    end case;
  end process;
  synchronous: process
  begin
    wait until rising_edge(clock);
    present_state <= next_state;
  end process;
end fsm_a;

```

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

State Machines Variations

- **It is possible to have additional state variables in a synthesizable FSM**
 - E.g., a counter to count the number of cycles through a certain state
- **It is also possible to have more than one state machine in a given architecture**
 - Separate enumerated state variables
 - Separate or combined state transition, clock and output 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.

Synthesis with RTL Subset (Contd.)

- This section describes the use of the RTL subset in synthesis of
 - Combinational circuits and logic
 - Register definition and synchronous behavior
 - Synchronous sequential circuits
 - State machines and controllers
 - **Sequential Datapath**

Datpath 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.

Sequential Datapaths

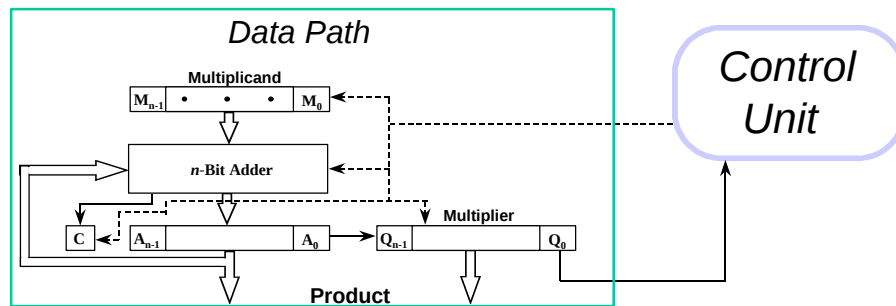
- **RTL descriptions that describe datapaths and control units can be synthesized**
- **Individual VHDL processes are used to describe combinational logic blocks, registers, and state machines**
- **Layout of the datapath is “implied” by the interconnection of the processes via signals**

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



Copyright © 1995-1999 SCRA

158

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

Sequential RTL Datapaths Example - 8 Bit Multiplier

● Entity

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

ENTITY multiplier8 IS
  PORT(start      : IN  std_logic;      -- input to indicate start
        start_ack : OUT std_logic;      -- output to acknowledge data
        clk       : IN  std_logic;
        reset     : IN  std_logic;
        a_input   : IN  unsigned(7 DOWNTO 0);
        b_input   : IN  unsigned(7 DOWNTO 0);
        product   : OUT unsigned(15 DOWNTO 0);
        data      : OUT std_logic;      -- data present on outputs
        data_ack  : IN  std_logic);      -- data acknowledgement
END multiplier8;
```

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

Sequential RTL Datapaths Example - 8 Bit Multiplier

- Architecture declarative part
- Concurrent signal assignment statements

```
ARCHITECTURE synthesizable_rtl OF multiplier8 IS

  SUBTYPE count_integer IS integer RANGE 0 to 8;
  TYPE states IS (idle, initialize, test, shift, add, stop);
  SIGNAL present_state : states := idle;
  SIGNAL next_state    : states := idle;
  SIGNAL present_count : count_integer;
  SIGNAL next_count    : count_integer;
  SIGNAL a_enable      : std_logic;
  SIGNAL a_reset       : std_logic;
  SIGNAL a_mode        : std_logic;
  SIGNAL c_enable      : std_logic;
  SIGNAL c_reset       : std_logic;
  SIGNAL m_enable      : std_logic;
  SIGNAL q_enable      : std_logic;
  SIGNAL q_mode        : std_logic;
  SIGNAL e_enable      : std_logic;
  SIGNAL c              : std_logic;
  SIGNAL c_out         : std_logic;
  SIGNAL multiplier    : unsigned(7 DOWNTO 0);
  SIGNAL multiplicand  : unsigned(7 DOWNTO 0);
  SIGNAL accumulator   : unsigned(7 DOWNTO 0);
  SIGNAL adder_result  : unsigned(7 DOWNTO 0);

  BEGIN
    product(15 DOWNTO 8) <= accumulator;
    product(7 DOWNTO 0)  <= multiplier;
```

The signals are declared in this slide.

Sequential RTL Datapaths Example - 8 Bit Multiplier

- **Multiplicand (M)**
register process
- **Multiplier (Q)** shift
register process

```
m_reg : PROCESS(clk)
BEGIN
  IF(clk'EVENT AND clk = '1') THEN
    IF(m_enable = '1') THEN
      multiplicand(7 DOWNTO 0) <= a_input;
    END IF;
  END IF;
END PROCESS m_reg;

q_reg : PROCESS(clk)
BEGIN
  IF(clk'EVENT AND clk = '1') THEN
    IF(q_enable = '1') THEN
      IF(q_mode = '1') THEN
        multiplier(7 DOWNTO 0) <= b_input;
      ELSIF(q_mode = '0') THEN
        multiplier(6 DOWNTO 0) <= multiplier(7 DOWNTO 1);
        multiplier(7) <= accumulator(0);
      END IF;
    END IF;
  END IF;
END PROCESS q_reg;
```

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

Sequential RTL Datapaths Example - 8 Bit Multiplier

- **Accumulator (A) shift register process**
- **Carry (C) register process**

```
a_reg : PROCESS(clk)
BEGIN
  IF(clk'EVENT AND clk = '1') THEN
    IF(a_reset = '0') THEN
      accumulator <= "00000000";
    ELSIF(a_enable = '1') THEN
      IF(a_mode = '1') THEN
        accumulator <= adder_result;
      ELSIF(q_mode = '0') THEN
        accumulator(6 DOWNTO 0) <= accumulator(7 DOWNTO 1);
        accumulator(7) <= c_out;
      END IF;
    END IF;
  END IF;
END PROCESS a_reg;

c_reg : PROCESS(clk)
BEGIN
  IF(clk'EVENT AND clk = '1') THEN
    IF(c_reset = '0') THEN
      c_out <= '0';
    ELSIF(c_enable = '1') THEN
      c_out <= c;
    END IF;
  END IF;
END PROCESS c_reg;
```

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

Sequential RTL Datapaths Example - 8 Bit Multiplier

- **Adder
combinational
logic block**

```
adder : PROCESS (multiplicand,accumulator)

    VARIABLE a_temp : unsigned(8 downto 0);
    VARIABLE b_temp : unsigned(8 downto 0);
    VARIABLE result : unsigned(8 downto 0);

    BEGIN
        a_temp(7 downto 0) := multiplicand;
        a_temp(8) := '0';
        b_temp(7 downto 0) := accumulator;
        b_temp(8) := '0';
        result := a_temp + b_temp;
        c <= result(8);
        adder_result <= result(7 downto 0);
    END PROCESS adder;
```

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

Sequential RTL Datapaths Example - 8 Bit Multiplier

- Control unit memory process
- Control unit next state process

```
controller_state_reg : PROCESS(clk,reset)
BEGIN
  IF(reset = '1') THEN
    present_state <= idle;
    present_count <= 0;
  ELSIF(clk'EVENT AND clk = '1') THEN
    present_state <= next_state;
    present_count <= next_count;
  END IF;
END PROCESS controller_state_reg;
```

```
controller_state_trans : PROCESS(present_state,
  present_count,start,multiplier(0),data_ack)
BEGIN
  next_state <= present_state; -- default case
  next_count <= present_count; -- default case
  CASE present_state IS
    WHEN idle =>
      IF(start = '1') THEN
        next_state <= initialize;
      ELSE
        next_state <= idle;
      END IF;
      next_count <= 0;
    WHEN initialize =>
      next_state <= test;
      next_count <= present_count;
```

```
    WHEN test =>
      IF(present_count < 8) THEN
        IF(multiplier(0) = '0') THEN
          next_state <= shift;
        ELSE
          next_state <= add;
        END IF;
      ELSE
        next_state <= stop;
      END IF;
      next_count <= present_count;
    WHEN add =>
      next_state <= shift;
      next_count <= present_count;
    WHEN shift =>
      next_state <= test;
      next_count <= present_count+1;
    WHEN stop =>
      IF(data_ack = '1') THEN
        next_state <= idle;
      ELSE
        next_state <= stop;
      END IF;
    WHEN OTHERS =>
      next_state <= idle;
      next_count <= present_count;
    END CASE;
  END PROCESS controller_state_trans;
```

The multiplier sequencer and controller are described in this slide.

Sequential RTL Datapaths Example - 8 Bit Multiplier

● Control unit output process

```
controller_output : PROCESS(present_state)
BEGIN
  CASE present_state IS
    WHEN idle =>
      a_enable <= '0';
      a_reset <= '1';
      a_mode <= '1';
      c_enable <= '0';
      c_reset <= '1';
      m_enable <= '0';
      q_enable <= '0';
      q_mode <= '1';
      e_enable <= '0';
      start_ack <= '0';
      data <= '0';
    WHEN initialize =>
      a_enable <= '1';
      a_reset <= '0';
      a_mode <= '1';
      c_enable <= '0';
      c_reset <= '0';
      m_enable <= '1';
      q_enable <= '1';
      q_mode <= '1';
      e_enable <= '1';
      start_ack <= '1';
      data <= '0';
```

```
    WHEN test =>
      a_enable <= '0';
      a_reset <= '1';
      a_mode <= '1';
      c_enable <= '0';
      c_reset <= '1';
      m_enable <= '0';
      q_enable <= '0';
      q_mode <= '1';
      e_enable <= '0';
      start_ack <= '0';
      data <= '0';
    WHEN add =>
      a_enable <= '1';
      a_reset <= '1';
      a_mode <= '1';
      c_enable <= '1';
      c_reset <= '1';
      m_enable <= '0';
      q_enable <= '0';
      q_mode <= '0';
      e_enable <= '0';
      start_ack <= '0';
      data <= '0';
```

Continuation of the controller description.

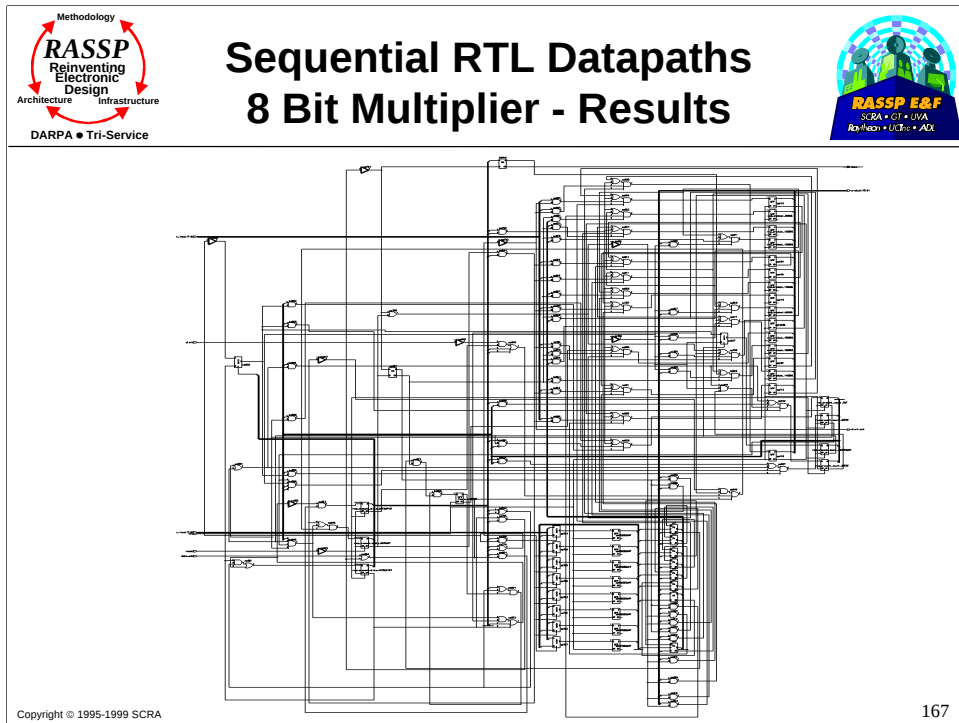
Sequential RTL Datapaths Example - 8 Bit Multiplier

● Control unit output process (Cont.)

```
WHEN shift =>
  a_enable <= '1';
  a_reset <= '1';
  a_mode <= '0';
  c_enable <= '0';
  c_reset <= '1';
  m_enable <= '0';
  q_enable <= '1';
  q_mode <= '0';
  e_enable <= '0';
  start_ack <= '0';
  data <= '0';
WHEN stop =>
  a_enable <= '0';
  a_reset <= '1';
  a_mode <= '1';
  c_enable <= '0';
  c_reset <= '1';
  m_enable <= '0';
  q_enable <= '0';
  q_mode <= '1';
  e_enable <= '0';
  start_ack <= '0';
  data <= '1';
```

```
WHEN OTHERS =>
  a_enable <= '0';
  a_reset <= '1';
  a_mode <= '1';
  c_enable <= '0';
  c_reset <= '1';
  m_enable <= '0';
  q_enable <= '0';
  q_mode <= '1';
  e_enable <= '0';
  start_ack <= '0';
  data <= '0';
END CASE;
END PROCESS controller_output;
END synthesizable_behavior;
```

Continuation of the multiplier controller.



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



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

Copyright © 1995-1999 SCRA

168

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 Used in Synthesis



- **Synthesis optimizations can occur at**
 - the Behavioral Level
 - the RTL Level
 - the logic levels of VHDL descriptions
- **Our focus will be on optimization at the behavioral and the RTL levels.**

Copyright © 1995-1999 SCRA

169

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.



Optimization of Behavioral VHDL



- **Scheduling and Assignment**
 - Input/output scheduling
 - Operation scheduling and assignment
 - Register allocation
- **Loop pipelining**
- **Memory and I/O inferencing**
- **Controller (FSM) generation**

Copyright © 1995-1999 SCRA

170

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.



Operation Scheduling and Assignment



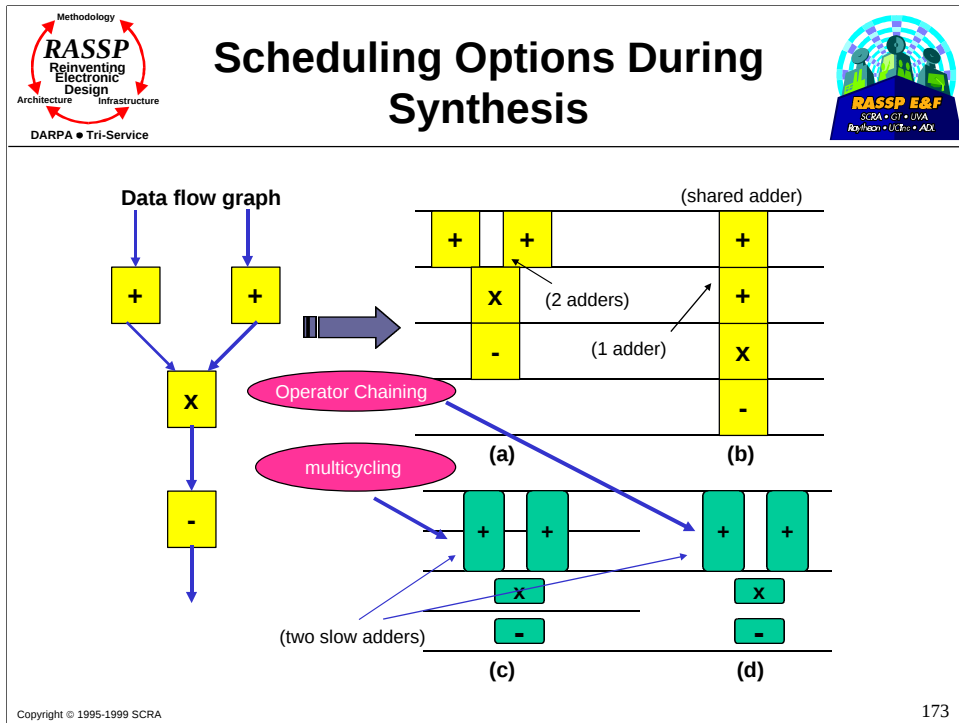
The scheduling activity tries to assign operations to clock cycles so that:

1. Data and control dependencies are satisfied,
2. High level constraints are met (e.g., latency, throughput, clock period)
3. Minimum resources are used by flattening the profile of operations across multiple cycles (subject to (1)).
4. Minimizing the number of registers used in producing and consuming the variables through efficient register sharing.

The typical objective functions that are optimized during this step are

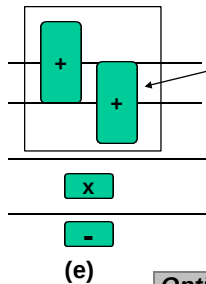
- Functional cost
- Interconnection cost (muxes, busses..)
- Register costs

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.

Scheduling Options (Cont.)



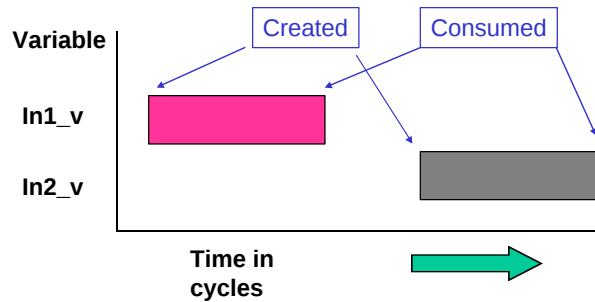
2-stage pipelined
adder

Note: operator chaining (d) allows use of slower adders and fewer cycles. Multicycling (c) allows use of slower adders but faster clock rate. Sharing (b) reduces area, and pipelining (e) reduces area and increases throughput, but increases latency. Also, these optimizations are carried out by the behavioral synthesis tool, without need for changing VHDL input code (constraints need be changed).

Options	(a)	(b)	(c)	(d)	(e)
Clock	50ns	50ns	50ns	100ns	50ns
Cycles	3	4	4	2	5
Adders	2 fast	1 fast	2 slow	2 slow	1 pipelined

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.

Register Allocation

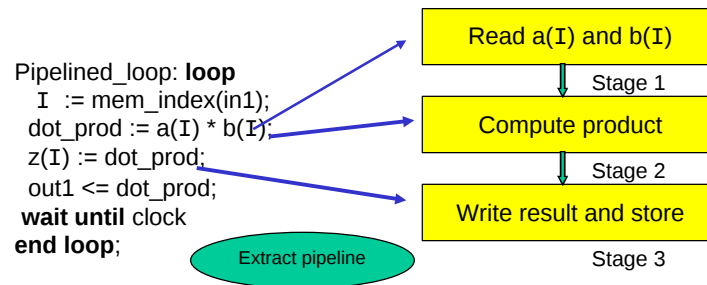


In1_v and **in2_v** are two variables that do *not* overlap in their lifetimes, and hence can be mapped to a *single* register for storage. Since a register is about a third the size of an adder, this can lead to savings in area and power consumption.

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.



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

- **Memory and I/O inferencing extracts the scheduling of memory accesses and the dependencies between memory accesses and other operations automatically from the VHDL code.**
- **The user can tradeoff between single port and multiple port memories, and also the mapping of several variables to the same RAM.**
- **Generation of addresses for multicycle operations is also handled automatically.**

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

RTL Level Optimization

- Expansion of subprograms by in-lining in the main program and then optimization
- Constant folding to remove unnecessary computation, e.g., `out1 <= a + 3 + 2` becomes `out1 <= a + 5`.
- Dead code is removed
- Bit minimization by procedures such as state encoding (using `ENUM_ENCODING`), etc.

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

Logic Level Optimizations

Common operations at this level include *factorization* and *flattening*, to name a few. Extensive literature on optimization at this level exists.

Original equation

$$\begin{aligned} Y &= A.B.C \\ Y2 &= Y + A.B.D \\ Y3 &= A.B + C + D \end{aligned}$$

Factorize

$$\begin{aligned} M &= A.B \\ Y &= M.C \\ Y2 &= Y + M.D \\ Y3 &= M + C + D \end{aligned}$$

Six gates

Flatten

$$\begin{aligned} Y2 &= A.B.C + A.B.D \\ Y3 &= A.B + C + D \end{aligned}$$

Factorize

$$\begin{aligned} M &= A.B, \\ N &= C + D, \quad Y2 = M.N \\ Y3 &= M + N \end{aligned}$$

Four gates

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.



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**
- Conclusions

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.



Coding Guidelines for Synthesis



- **Coding guidelines are organized as:**

- Design process issues
- HDL modeling issues
- Simulation speed issues
- Synthesis modeling issues
- Technology dependent issues

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

Design Process Issues

- A top down design methodology, with a bottom up verification methodology is preferred.
- The design specification is defined as clearly as possible in form of an executable specification that matches the executable requirement
- Global clock and reset signals are recommended
- Testability issues are to be considered early on in the design process. Partial scan and Built-in-Self-Test (BIST) are suitable options that tradeoff between speed and area.

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

VHDL Modeling Issues

- Before coding, the RTL architecture should be clearly defined with respect to controller, datapath and memories,
- The HDL code should reflect the RTL architecture, and the partitioning of the architecture into blocks of approximately 5000 gates (2-3 arithmetic units plus logic) is carried out to facilitate rapid synthesis. Most synthesis tools work well in this range of complexity. The process statement allows partitioning of the design into concurrent structures

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**

Copyright © 1995-1999 SCRA

185

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.

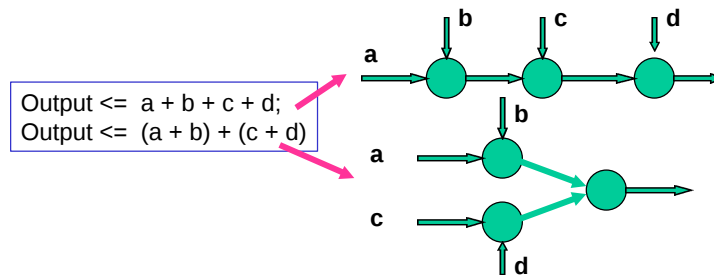
Synthesis Modeling Issues

- In modeling combinational logic, signals should be assigned in all branches to prevent inferring latches
- Objects assigned to in a for loop should be assigned a default value prior to execution of the loop
- Case statement is preferred over the if statement for its readability, and sometimes efficient synthesis
- Unbounded integer types are to be avoided since they can default to the 32 bit integer type. This can cause problems with the optimization stage
- IEEE standard packages should be used

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.

Synthesis Modeling Issues (Cont.)

- Ensuring compatibility between simulation and synthesis. Both VHDL descriptions below generate the same synthesis output, but differ in their simulation.

Full sensitivity list

```
Process (enable, in1, in2)
begin
  if (enable = '0') then
    output <= (in1 and in2) or in3;
  end if ;
end process;
```

```
Process (enable, in1, in2, in3)
begin
  if (enable = '0') then
    output <= (in1 and in2) or in3;
  end if;
end process;
```

Note: most synthesis tools ignore the sensitivity list, but IEEE RTL standard recommends using a full sensitivity list to ensure compatibility with the simulation.

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

Synthesis Modeling Issues (Cont.)

- **Order of signal assignments does not affect synthesis and simulation as long as the sensitivity list is complete. However, if using signals *and* variables the order may affect simulation and synthesis**

```
Test1_v := Test2_v and B and C;  
Sig_s <= not Test1_v;  
Test2_v := D and E;
```

Swapping these statements
only affects simulation (not synthesis)

```
Var1_v := A and B;  
Var1_v := B and C;  
output <= Var1_v;
```

Swapping these statements
affects synthesis and simulation

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

Technology Dependence in Synthesis

- The VHDL coding should sometimes (unfortunately) take into account whether the target is a FPGA or an ASIC. For example, in FSM coding, a one-hot coding method is area efficient in FPGAs, but *not* for ASICs.

FSM Encoding	FPGA Area (CLBs)	FPGA Time (ns)	ASIC Area (units)	ASIC Time (ns)
One-Hot Encoding	8	18	73	17
Binary Encoding	13	43	54	12

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).



Implementation Technology Considerations



- **The developer of a synthesizable behavioral VHDL description MUST consider the implementation technology when writing the code**
 - **Implementability reasons**
 - ❑ Some FPGA families do not have internal tri-state devices
 - behavioral descriptions that are based on internal tri-state busses will fail in the technology mapping phase
 - ❑ Many PLDs have flip flops only on the I/O pins - state machines with too many I/O ports and state variables simply won't fit
 - **Efficiency reasons**
 - ❑ Some FPGA technologies do not have built-in flip-flops, so registers or state variables are expensive
 - ❑ In some ASIC technologies multiplexors may in fact be faster than tri-states

Copyright © 1995-1999 SCRA

193

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

Signal Selection in VHDL (for FPGAs)

- Signal selection in VHDL for some FPGA targets could be different from that used for ASICs, as shown below:

General signal selection in VHDL

```
if (sel = '0') then output <= signal_0;
else
    output <= signal_1;
end if;
```

Signal selection is implemented here using muxes

In some FPGAs, muxes can be expensive, and tri-state buffers are cheaper

Preferred signal selection for FPGAs using tri-state busses

```
output <= signal_0 when (EnA = '1') else 'Z';
output <= signal_1 when (EnB = '1') else 'Z';
```

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

Memory Design for in VHDL FPGAs

- The memory address decode should be implemented with tri-state busses for some FPGA families, *and*
- The RAM itself should instantiate library cells provided, e.g., RAM16X1 in Xilinx XACT library, as shown below:

```
Architecture tech_depend_ram of scratch_pad is
...
component ram16x1
port (D, A3, A2, A1, A0, WE: in std_logic; O : out std_logic);
end;
begin
for I in 0 to width -1 generate
cell_ram16x1: ram16x1 port map ( D => value_in(I),
A3 => addr(3), A2 => addr(2), A1 => addr(1), A0 => addr(0),
WE => write, O => value_out (I);
end generate;
end tech_depend_ram;
```

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>