Content

1. VHDL - Overview and Application Field

1.1 Application of HDLs (1)
1.1.1 Application of HDLs (2)
1.1.2 Range of Use

1.2 VHDL - Overview
1.2.1 VHDL - History
1.2.2 VHDL - Application Field
1.2.3 ASIC Development

1.3 Concepts of VHDL
1.3.1 Abstraction
1.3.2 Abstraction Levels in IC Design
1.3.3 Abstraction levels and VHDL
1.3.4 Description of Abstraction Levels
1.3.5 Behavioural Description in VHDL
1.3.6 RT Level in VHDL
1.3.7 Gate Level in VHDL
1.3.8 Information Content of Abstraction Levels

1.4 Modularity and Hierarchy

1.5 Summary

2. VHDL Language and Syntax

2.1 General
2.1.1 Identifier
2.1.2 Naming Convention

2.2 VHDL Structural Elements
2.2.1 Declaration of VHDL Objects
2.2.2 Entity
2.2.3 Architecture
2.2.4 Architecture Structure
2.2.5 Entity Port Modes
2.2.6 Hierarchical Model Layout
2.2.7 Component Declaration

2.2.8 Component Instantiation
2.2.9 Component Instantiation: Named Signal Asscociation
2.2.10 Configuration
2.2.11 Configuration: Task and Application
2.2.12 Configuration: Example (1)
2.2.13 Configuration: Example (2)
2.2.14 Process
2.2.15 VHDL Communication Model
2.2.16 Signals
2.2.17 Package
2.2.18 Library
2.2.19 Design Structure: Example
2.2.20 Sequence of Compilation
2.2.21 Outlook: Testbench
2.2.22 Simple Testbench Example
2.2.23 Summary
2.2.24 Questions
2.2.25 Questions
2.2.26 Questions

2.3 Data Types
2.3.1 Standard Data Types
2.3.2 Datatype 'time'
2.3.3 Definition of Arrays
2.3.4 'integer' and 'bit' Types
2.3.5 Assignments with Array Types
2.3.6 Types of Assignment for 'bit' Data Types
2.3.7 Concatenation
2.3.8 Aggregates
2.3.9 Slices of Arrays
2.3.10 Questions
2.3.11 Questions

2.4 Extended Data Types
2.4.1 Type Classification
2.4.2 Enumeration Types
2.4.3 Enumeration Types - Example
2.4.4 BIT Type Issues
2.4.5 Multi-valued Types
2.4.6 IEEE Standard Logic Type
2.4.7 Resolved and Unresolved Types
2.4.8 Std_Logic_1164 Package
2.4.9 Resolution Function
2.4.10 STD_LOGIC vs STD_ULOGIC

2.4.11 The NUMERIC_STD Package
2.4.12 Arrays
2.4.13 Multidimensional Arrays
2.4.14 Aggregates and Multidimensional Arrays
2.4.15 Records
2.4.16 Type Conversion
2.4.17 Subtypes
2.4.18 Aliases

2.5 Operators
2.5.1 Logical Operators
2.5.2 Logical Operations with Arrays
2.5.3 Shift Operators: Examples
2.5.4 Relational Operators
2.5.5 Comparison Operations with Arrays
2.5.6 Arithmetic Operators
2.5.7 Questions
2.5.8 Questions

2.6 Sequential Statements
2.6.1 IF Statement
2.6.2 IF Statement: Example
2.6.3 CASE Statement
2.6.4 CASE Statement: Example
2.6.5 Defining Ranges
2.6.6 FOR Loops
2.6.7 Loop Syntax
2.6.8 Loop Examples
2.6.9 WAIT Statement
2.6.10 WAIT Statement: Examples
2.6.11 WAIT Statements and Behavioural Modeling
2.6.12 Variables
2.6.13 Variables vs. Signals
2.6.14 Use of Variables
2.6.15 Variables: Example
2.6.16 Global Variables (VHDL'93)

2.7 Concurrent Statements
2.7.1 Conditional Signal Assignment
2.7.2 Conditional Signal Assignment: Example
2.7.3 Selected Signal Assignment
2.7.4 Selected Signal Assignment: Example
2.7.5 Concurrent Statements: Summary

2.8 RTL-Style

2.8.1 Combinational Process: Sensitivity List
2.8.2 WAIT Statement <-> Sensitivity List
2.8.3 Combinational Process: Incomplete Assignments
2.8.4 Combinational Process: Rules
2.8.5 Clocked Process: Clock Edge Detection
2.8.6 .Detection of a Rising Edge by Use of Functions
2.8.7 Register Inference
2.8.8 Asynchronous Set/Reset
2.8.9 Clocked Process: Rules
2.8.10 Questions
2.8.11 Questions
2.8.12 Questions

2.9 Subprograms
2.9.1 Parameters and Modes
2.9.2 Functions
2.9.3 Procedures

2.10 Subprogram Declaration and Overloading
2.10.1 Overloading Example
2.10.2 Overloading - Illegal Redeclarations
2.10.3 Overloading - Ambiguity
2.10.4 Operator Overloading
2.10.5 Operator Overloading - Example
2.10.6 Questions

3. Simulation

3.1 Sequence of Compilation
Example
Changes in ... recompile files ...

3.2 Simulation Flow
3.2.1 Elaboration
3.2.2 Initialization
3.2.3 Execution

3.3 Process Execution
3.3.1 Concurrent versus Sequential Execution
3.3.2 Signal Update
3.3.3 Delta Cycles (1)
3.3.4 Delta Cycles (2)
3.3.5 Delta Cycles - Example
3.3.6 Process Behaviour

3.3.7 Postponed Processes

3.4 Delay Models
3.4.1 Projected Output Waveforms (LRM)
3.4.2 Transport Delay (1)
3.4.3 Transport Delay (2)
3.4.4 Inertial Delay (1)
3.4.5 Inertial Delay (2)
3.4.6 Inertial Delay (3)

3.5 Testbenches
3.5.1 Structure of a VHDL Testbench
3.5.2 Example
Clock and Reset Generation
Stimuli Generation
Response Analysis

3.6 File I/O
3.6.1 Example for File I/O (1/4)
Example (2/4)
Example (3/4)
Example (4/4)

4. Synthesis

4.1 What is Synthesis?
4.1.1 Synthesizability
4.1.2 Different Language Support for Synthesis
4.1.3 How to Do?
4.1.4 Essential Information for Synthesis
4.1.5 Synthesis Process in Practice
4.1.6 Problems with Synthesis Tools
4.1.7 Synthesis Strategy

4.2 RTL-style
4.2.1 Combinatorics
4.2.2 Complete sensitivity lists
4.2.3 WAIT statement <-> Sensitivity List
4.2.4 Incomplete assignments
4.2.5 Rules for synthesizing combinational logic
4.2.6 Modelling of Flip Flops
4.2.7 Description of a rising clock edge for synthesis
4.2.8 Describing a rising clock edge by means of a function call
4.2.9 Counter synthesis

4.2.10 FF with asynchronous reset
4.2.11 Rules for clocked processes
4.2.12 Questions
4.2.13 Questions
4.2.14 Questions

4.3 Combinational Logic
4.3.1 Coding Style Influence
4.3.2 Source Code Optimization
4.3.3 IF structure <-> CASE structure
4.3.4 Implementation of a Data Bus
Problems with Internal Bus Structures
Portable and Safe Bus Structure
4.3.5 Example of a Multiplier
Multiplier Function Table
Multiplier Minterms -- Karnaugh Diagram
Multiplier: VHDL Code using the Function Table
Multiplier: Minterm Conversion
Multiplier: Integer Realization
4.3.6 Synthesis of Operators
Synthesis Results
4.3.7 Example of an Adder

4.4 Sequential Logic
4.4.1 RTL - Combinational Logic and Registers
4.4.2 Variables in Clocked Processes
Example

4.5 Finite State Machines and VHDL
4.5.1 One "State" Process
4.5.2 Two "State" Processes
4.5.3 How Many Processes?
4.5.4 State Encoding
4.5.5 Extension of Case Statement
4.5.6 Extension of Type Declaration
4.5.7 Hand Coding
4.5.8 FSM: Medvedev
4.5.9 Medvedev Example
4.5.10 Waveform Medvedev Example
4.5.11 FSM: Moore
4.5.12 Moore Example
4.5.13 Waveform Moore Example
4.5.14 FSM: Mealy
4.5.15 Mealy Example

4.5.16 Waveform Mealy Example
4.5.17 Modelling Aspects
4.5.18 Registered Output
4.5.19 Registered Output Example (1)
4.5.20 Waveform Registered Output Example (1)
4.5.21 Registered Output Example (2)
4.5.22 Waveform Registered Output Example (2)

4.6 Advanced Synthesis
4.6.1 Parameterization via Constants
4.6.2 Parameterization via Generics(1)
4.6.3 Parameterization via Generics(2)
4.6.4 GENERATE Statement
4.6.5 Conditional GENERATE Statement
4.6.6 'Parameterization' via Signals

5. Project Management

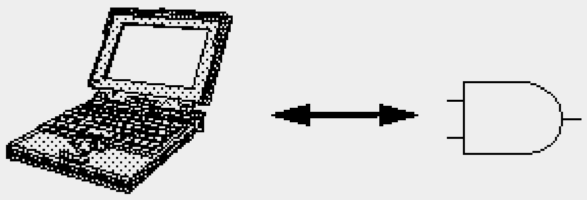
5.1 Design Components
5.1.1 Libraries
5.1.2 The LIBRARY Statement
5.1.3 The USE Statement

5.2 Name Spaces

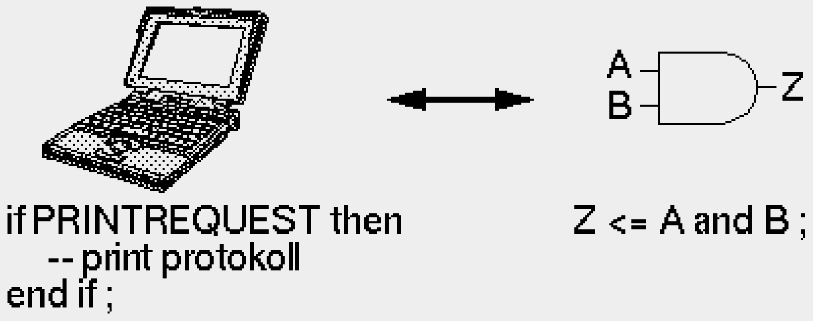
5.3 File Organisation
5.3.1 Packages
5.3.2 Package Syntax
5.3.3 Package Example
5.3.4 Use of Packages
5.3.5 Visibility of Package Contents

1. VHDL - Overview and Application Field

- **What is hardware?**



- **What kind of description?**



```
if PRINTREQUEST then
    -- print protokoll
end if ;
```

```
Z <= A and B ;
```

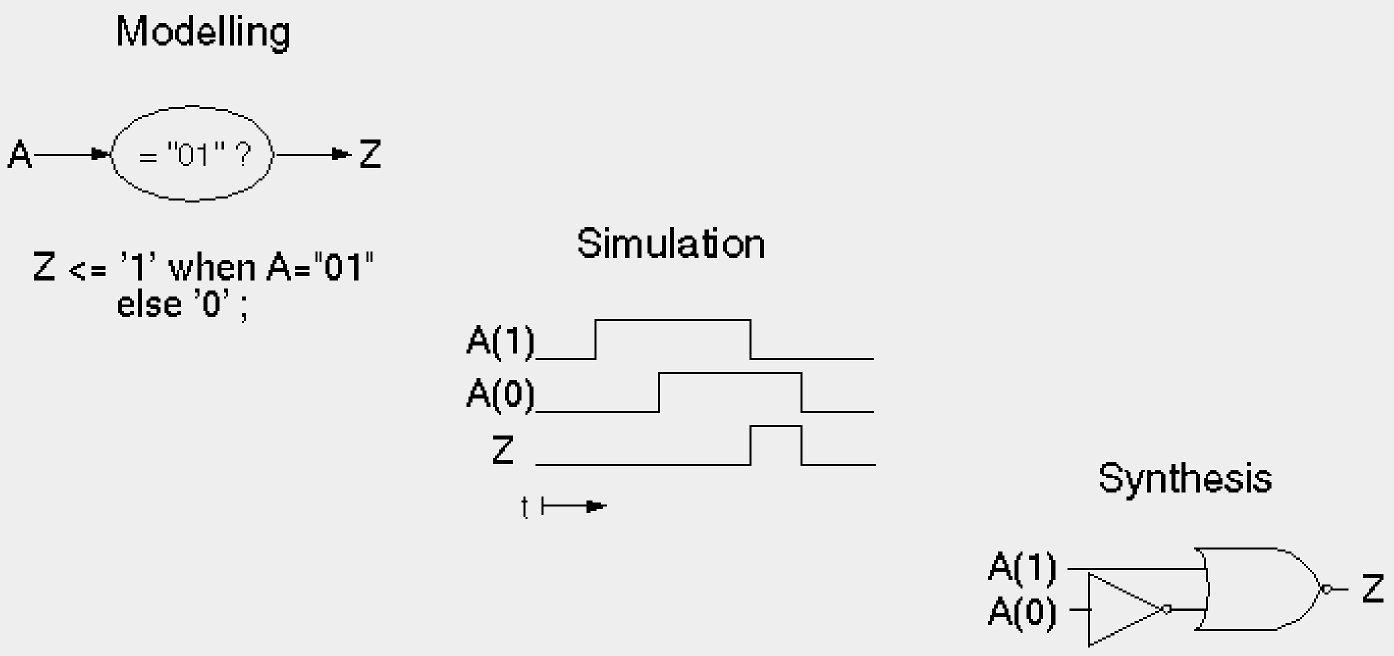
- **Hardware Description Language (HDL) = "Programming"-language for modelling of (digital) hardware**

VHDL is a hardware description language. The word 'hardware', however, is used in a wide variety of contexts which range from complete systems like personal computers on one side to the small logical gates on their internal integrated circuits on the other side.

This is why different descriptions exist for the hardware functionality. Complex systems are often described by the behaviour that is observable from the outside. Abstract behavioural models are used in this case that hide all the implementation details. In this example the print protocol will be executed whenever a PRINTREQUEST occurs. This can be either a pressed key or a software command, etc. The description of a basic logic gate, on the other hand, may consist of only one boolean equation. This is a very short and precise description.

The language VHDL covers the complete range of applications and can be used to model (digital) hardware in a general way.

1.1 Application of HDLs (1)



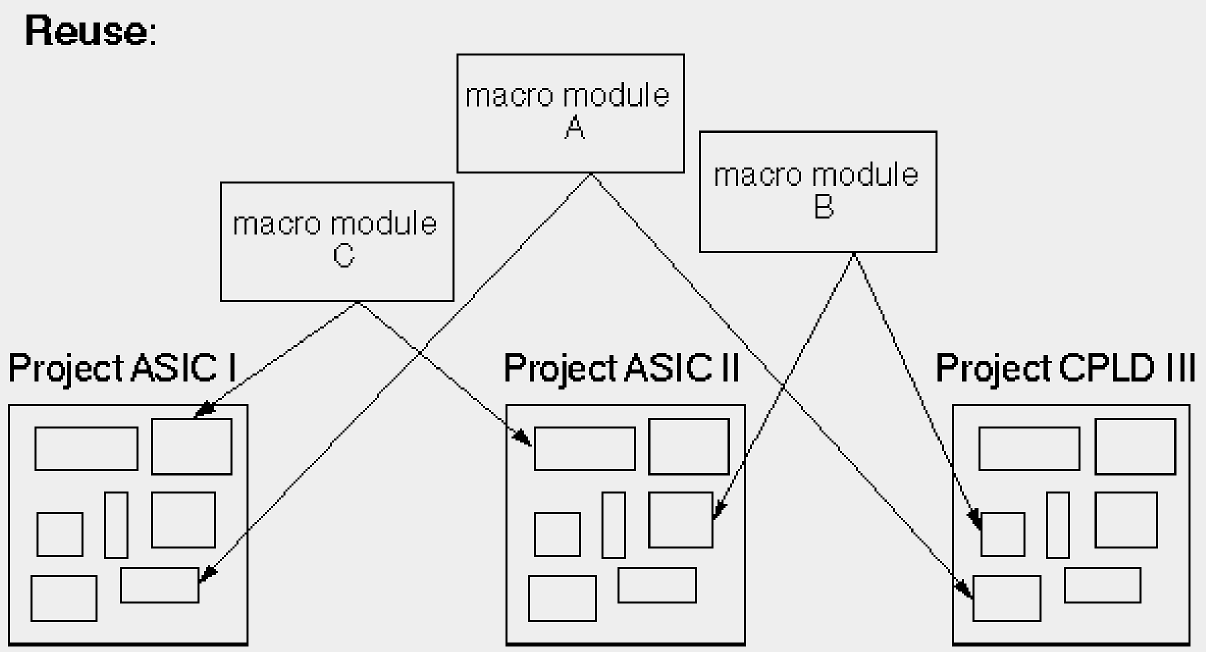
Let's have a look at the field of application for a hardware description language:

The most evident application is probably the development of a formal model of the behaviour of a system. With formality, misunderstandings and misinterpretations can be avoided. Because of the selfdocumenting character of VHDL, a VHDL model can even serve as system documentation to a certain degree.

The big advantage of hardware description languages is the possibility to actually execute the code. In principle, they are nothing else than a specialized programming language. Coding errors of the formal model or conceptual errors of the system can be found by running simulations. There, the response of the model on stimulation with different input values can be observed and analysed.

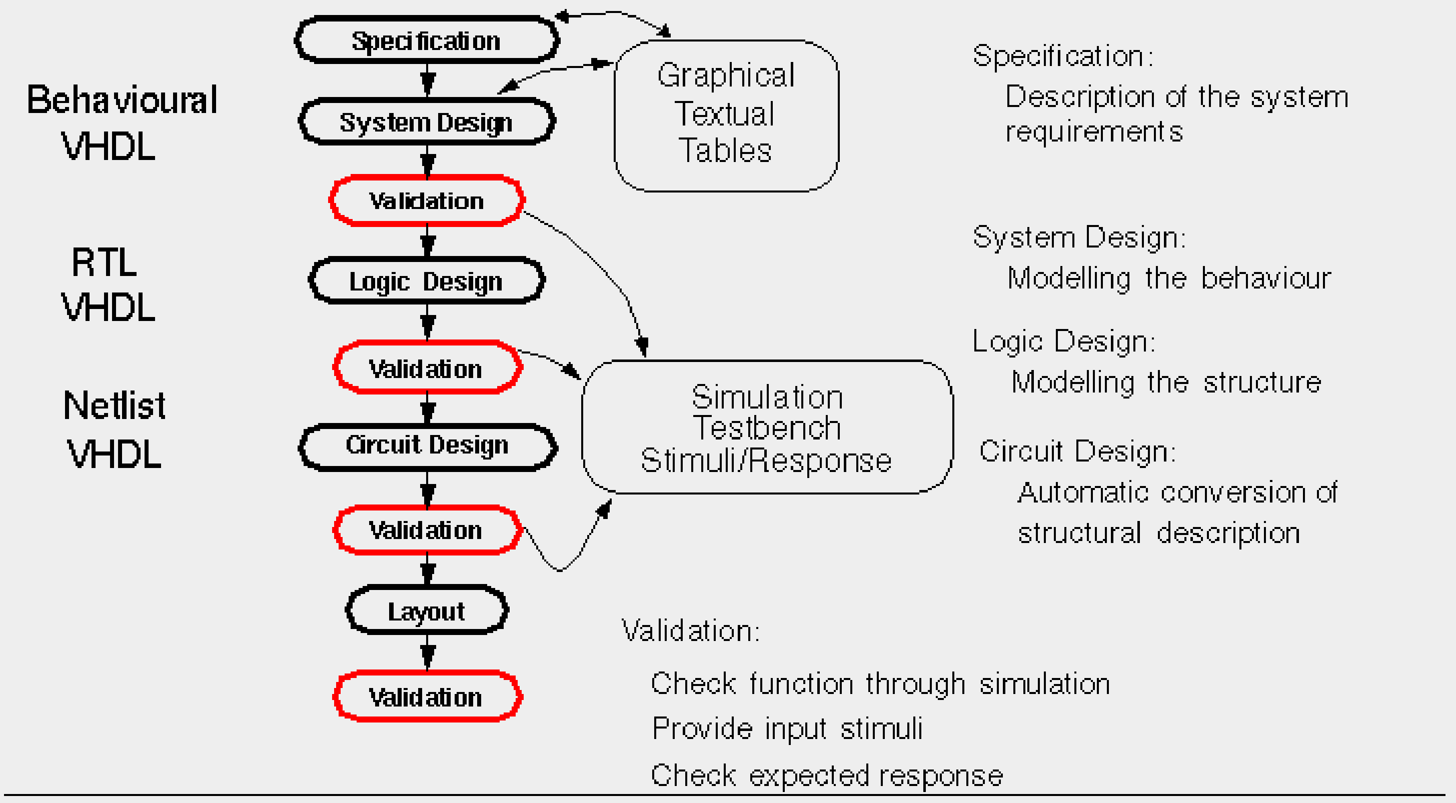
During the development cycle the description has to become more and more precise until it is actually possible to manufacture the product. The (automatic) transformation of a less detailed description into a more elaborated one is called synthesis. Existing synthesis tools are capable of mapping specific constructs of hardware description languages directly to the standard components of integrated circuits. This way, a formal model of the hardware system can be used from the early design studies to the final netlist. Software support is available for the necessary refinement steps.

1.1.1 Application of HDLs (2)



Additionally, hardware description languages offer so called design reuse capabilities. Similar to simple electronic components, like for example a resistor, the corresponding HDL model can be re-used in several designs/projects. It is common use that frequently needed function blocks (macros) are collected in model libraries. The selection of an existing module is not only restricted to the design engineer but can sometimes be performed by a sytnesis tool.

1.1.2 Range of Use



The design process always starts with a specification phase: The component which is to be designed is defined with respect to function, size, interfaces, etc. Despite the complexity of the final product, mainly simple methods, based on paper and pencil most of the time, are being used.

After that, self-contained modules have to be defined on the **system level** . Their interaction is described very precisely and interfaces (inputs, outputs, data formats), clock speed and reset mechanism are specified. With that information at hand, pure simulation models of the circuit can be developed. Behaviour models of standard components can be integrated into the system from libraries of commercial model developers. The overall system can already be simulated.

On the **logic level** , the models that have to be designed are described with all the synthesis aspects in view. As long as only a certain subset of VHDL constructs is used, commercial synthesis programs can derive the boolean functions from this abstract model description and map them to the elements of an ASIC gate library or the configurable logic blocks of FPGAs. The result is a **netlist** of the circuit or of the module on the **gate level** .

Finally, the circuit layout for a specific ASIC technology can be created by means of other tools from the netlist description.

Every transition to a lower abstraction level must be proven by functional **validation** . For this purpose, the description is simulated in such a way that for all stimuli (= input signals for the simulation) the module's responses are compared.

VHDL is suitable for the design phases from **system level** to **gate level** .

1.2 VHDL - Overview

- **Very High Speed Integrated Circuit Hardware Description Language**
  - **Modelling of digital systems**
  - **Concurrent and sequential statements**
  - **Machine-readable specification**
  - **Design lifetime > designer lifetime**
  - **Man- and machine-readable documentation**
- **International Standards**
  - **IEEE Std 1076-1987**
  - **IEEE Std 1076-1993**
- **Analogue- and mixed-signal extension: VHDL-AMS**
  - **IEEE Std 1076.1-1999**
- **Pure definition of language in the LRM (Language Reference Manual)**
  - **No standards for application or methodolgy**

VHDL development was initiated originally from the American Department of Defense (DoD). They requested a language for describing a hardware, which had to be readable for machines and humans at the same time and strictly forces the developer to write structured and comprehensible code, so that the source code itself can serve as a kind of specification document. Most important was the concept of concurrency to cope with the parallelism of digital hardware. Sequential statements to model very complex functions in a compact form were also allowed.

In 1987, VHDL was standardized by the American Institute of Electrical and Electronics Engineers (IEEE) for the first time with the first official update in 1993. Apart from the file handling procedures these two versions of the standard are compatible. The standard of the language is described in the Language Reference Manual (LRM).

A new and difficult stage was entered with the effort to upgrade VHDL with analogue and mixed-signal language elements. The upgrade is called VHDL-AMS (**a** nalogue- **m** ixed- **s** ignal) and it is a superset of VHDL. The digital mechanisms and methods have not been altered by the extension.

For the time being, only simulation is feasible for the analogue part because analogue synthesis is a very complex problem affected by many boundary conditions. The mixed signal simulation has to deal with the problem of synchronizing the digital- and analogue simulators, which has not been solved adequately, yet.

1.2.1 VHDL - History

- **early `70s: Initial discussions**
- **late `70s: Definition of requirements**
- **mid -`82: Contract of development with IBM, Intermetrics and TI**
- **mid -`84: Version 7.2**
- **mid -`86: IEEE-Standard**
- **1987: DoD adopts the standard -> IEEE.1076**
- **mid -`88: Increasing support by CAE manufacturers**
- **late `91: Revision**
- **1993: New standard**
- **1999: VHDL-AMS extension**

VHDL is a language which is permanently extended and revised. The original standard itself needed more than 16 years from the initial concept to the final, official IEEE standard. When the document passed the committee it was agreed that the standard should be revised every 5 years. The first revision phase resulted in the updated standard of the year 1993.

Independently of this revision agreement, additional effort is made to standardize "extensions" of the pure language reference. These extensions cover for examples packages (std_logic_1164, numeric_bit, numeric_std, ...) containing widely needed data types and subprograms, or the definition of special VHDL subsets like the synthesis subset IEEE 1076.6.

The latest extension is the addition of analogue description mechanisms to the standard which results in a VHDL superset called VHDL-AMS.

1.2.2 VHDL - Application Field

- **Hardware design**
  - **ASIC: technology mapping**
  - **FPGA: CLB mapping**
  - **PLD: smaller structures, hardly any use of VHDL**
  - **Standard solutions, models, behavioural description, ...**
- **Software design**
  - **VHDL - C interface (tool-specific)**
  - **Main focus of research (hardware/software co-design)**

VHDL is used mainly for the development of Application Specific Integrated Cicuits (ASICs). Tools for the automatic transformation of VHDL code into a gate-level netlist were developed already at an early point of time. This transformation is called synthesis and is an integral part of current design flows.

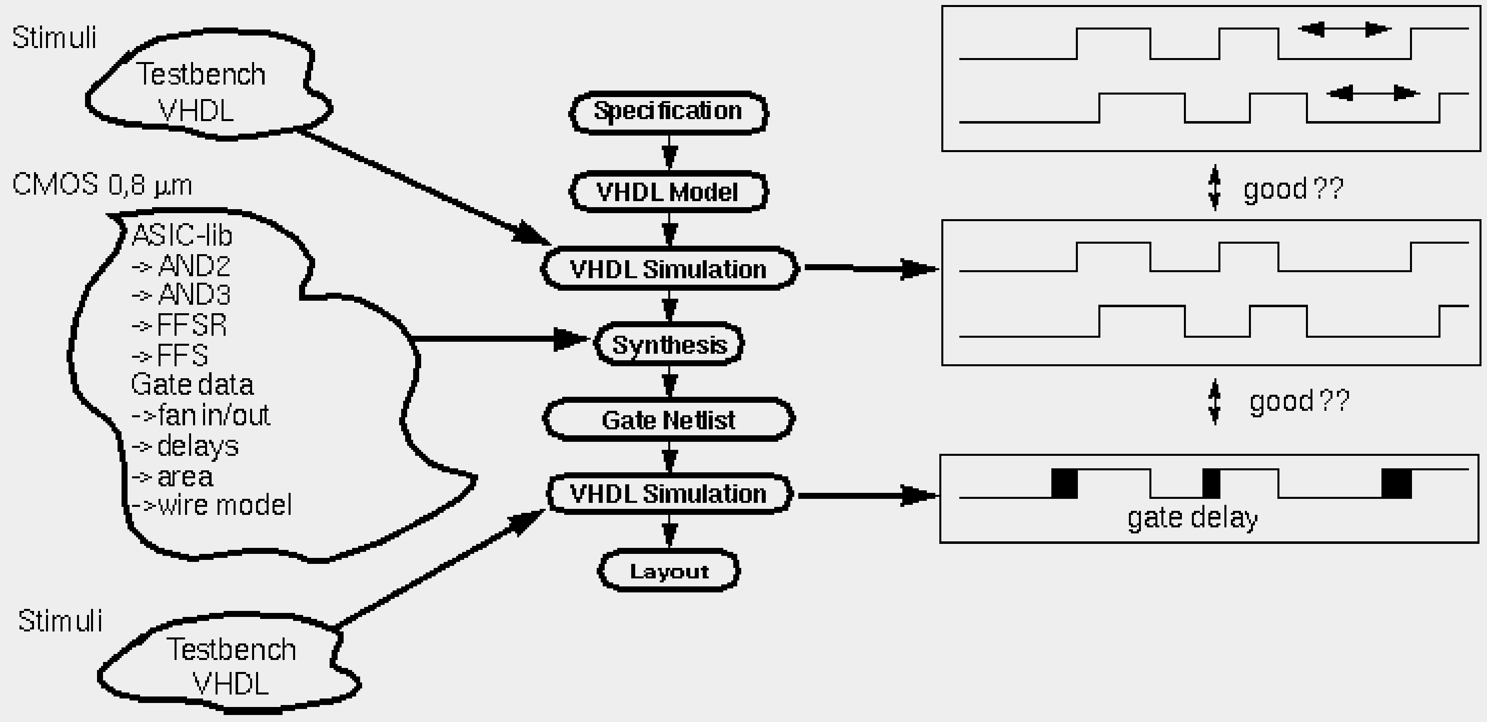
For the use with Field Programmable Gate Arrays (FPGAs) several problems exist. In the first step, boolean equations are derived from the VHDL description, no matter, whether an ASIC or a FPGA is the target technology. But now, this boolean code has to be partitioned into the configurable logic blocks (CLB) of the FPGA. This is more difficult than the mapping onto an ASIC library. Another big problem is the routing of the CLBs as the available resources for interconnections are the bottleneck of current FPGAs.

While synthesis tools cope pretty well with complex designs, they obtain usually only suboptimal results. Therefore, VHDL is hardly used for the design of low complexity Programmable Logic Devices (PLDs).

VHDL can be applied to model system behaviour independently from the target technology. This is either useful to provide standard solutions, e.g. for micro controllers, error correction (de-)coders, etc, or behavioural models of microprocessors and RAM devices are used to simulate a new device in its target environment.

An ongoing field of research is the hardware/software codesign. The most interesting question is which part of the system should be implemented in software and which part in hardware. The decisive constraints are the costs and the resulting perfomance.

1.2.3 ASIC Development



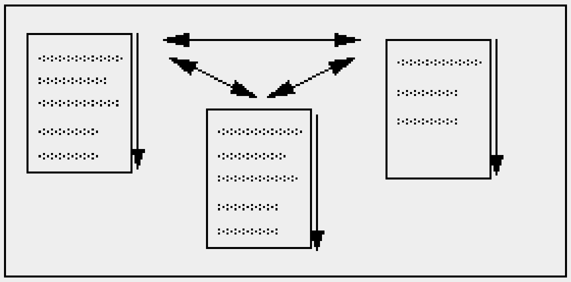
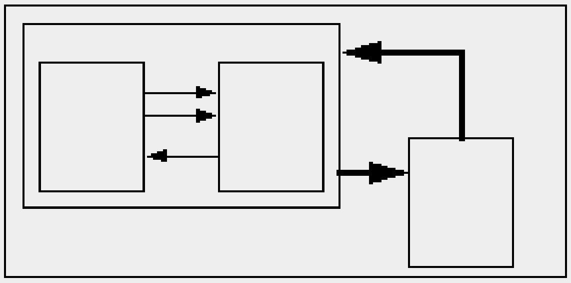
The development of VHDL models starts with their specification which covers functional aspects and the timing behaviour. Sometimes a behavioural VHDL model is derived from there, yet synthesizable code is frequently requested right from the beginning. VHDL code can be simulated and checked for the proper functionality.

If the model shows the desired behaviour, the VHDL description will be synthesized. A synthesis tool selects the appropriate gates and flip-flops from the specified ASIC library in order to reproduce the functional description. It is essential for the synthesis procedure that the sum of the resulting gate delays along the longest paths (from the output to the input of every Flip Flop) is less than the clock period.

As soon as a model built of ASIC librariy elements is available, a simulation on gate level can be performed. Now gate and propagation delays have to be taken into account. Delay values can be included in each VHDL model description, i.e. the designer receives the first clues about maximum clock frequency and critical paths after synthesis, already.

The propagation delay along the signal wires have to be estimated first because the real values are available after the layout is finished. The process of feeding these values back into the VHDL model is called back annotation. Once again it must be checked, whether the circuit fulfills the specified timing constraints.

1.3 Concepts of VHDL

	• **Execution of assignments:** ❍ **Sequential** ❍ **Concurrent**
	• **Methodologies:** ❍ **Abstraction** ❍ **Modularity** ❍ **Hierarchy**

VHDL distinguishes itself from other languages by the way assignments are executed because two basic types of statements are known:

Sequential statements are executed one after another, like in software programming languages. Subsequent statements can override the effects of previous statements this way. The order of the assignment must be considered when sequential statements are used.

Concurrent statements are active continuously. So the order of the statements is not relevant. Concurrent statements are especially suited model the parallelism of hardware.

VHDL features also three important modeling techniques:

Abstraction allows for the description of different parts of a system with different amount of detail. Modules which are needed only for the simulation do not have to be described as detailed as modules that might be synthesized.

Modularity enables the designer(s) to split big functional blocks and to write a model for each part.

Hierarchy lets the designer build a design out of submodules which may consist of several submodules, themselves. Each level of hierarchy may contain modules of different abstraction levels. The submodules of these models are present in the next lower hierarchical level.

1.3.1 Abstraction

- **Abstraction is hiding of details:**
  Differentiation between essential and nonessential information

- **Creation of abstraction levels:**
  On every abstraction level only the essential information is considered, nonessential information is left out

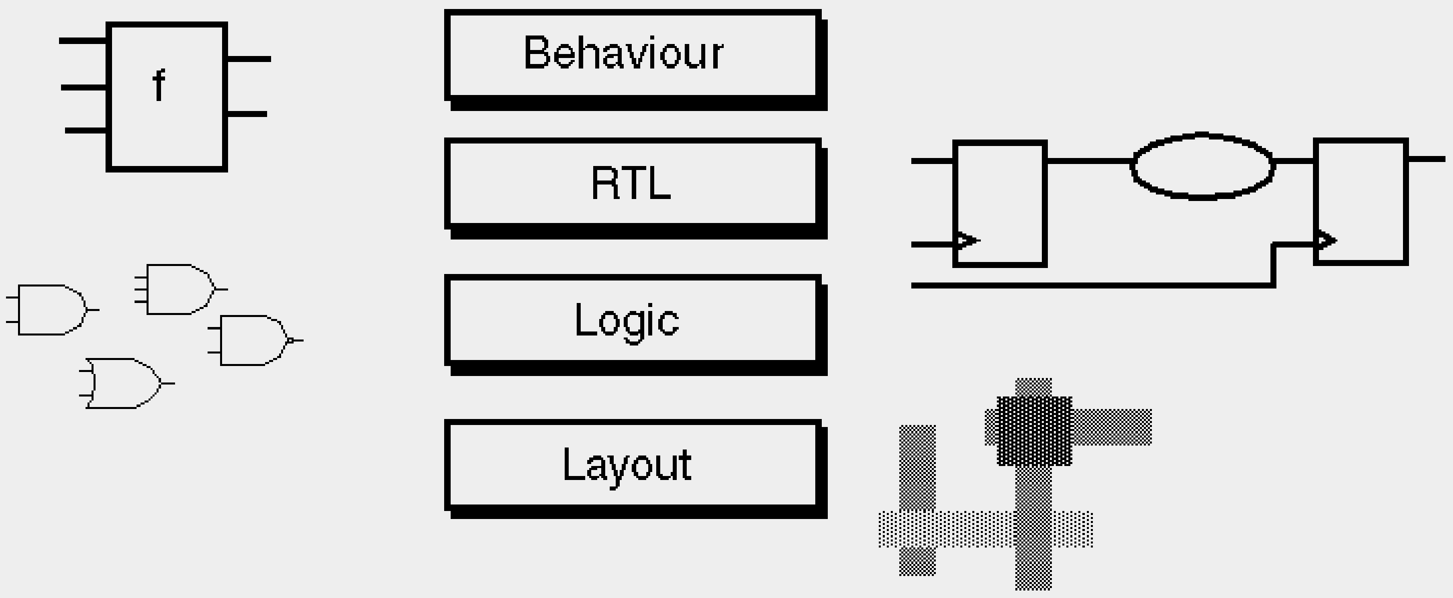
- **Equability of the abstraction:**
  All information of a model on one abstraction level contains the same degree of abstraction

---

Abstraction is defined as the hiding of information that is too detailed. It is therefore necessary to diffrentiate between essential and non-essential information. Information that is not important for the current view of the problem will be left out from the description. Abstraction levels are characterized by the kind of information that is common to all models of this level.

A model is said to be of a certain abstraction level if every module has the same degree of abstraction. If this is not the case than the model will be a mixture of different abstraction levels.

1.3.2 Abstraction Levels in IC Design

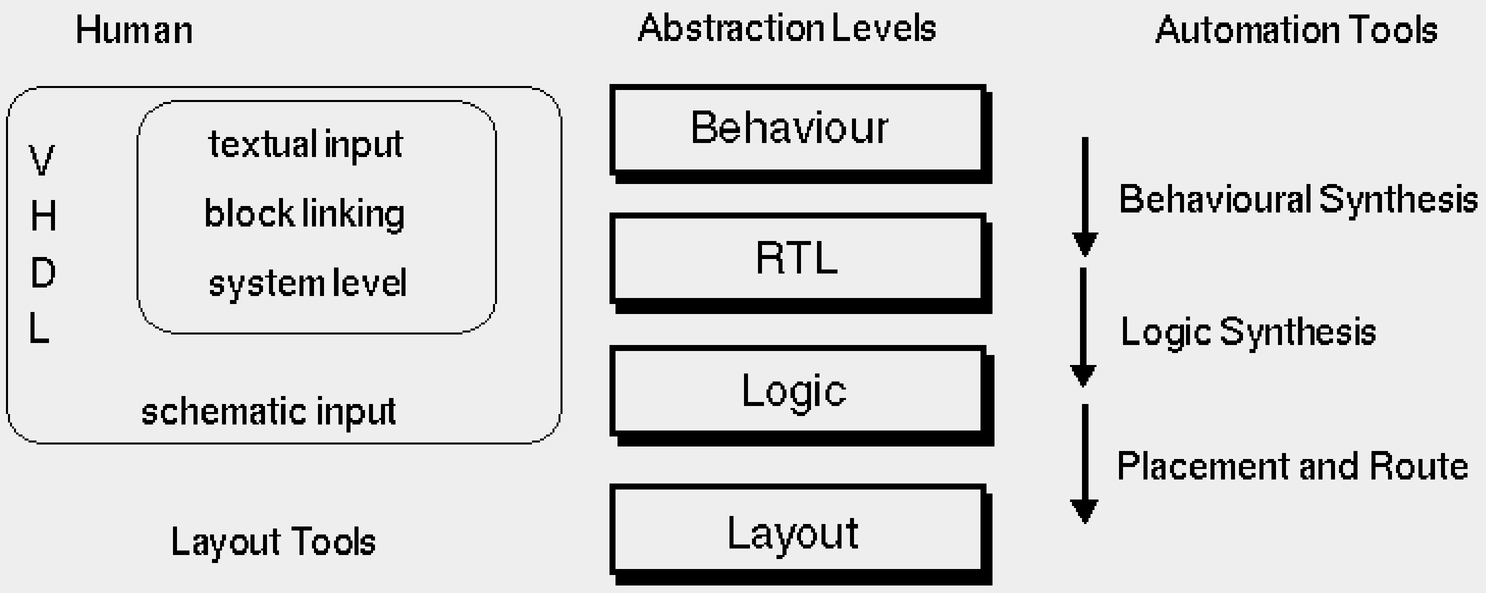


The four abstraction levels of a digital circuit design are shown in the figure. The functional description of the model is outlined in the behavioural level. There is no system clock and signal transitions are asynchronous with respect to the switching time. Usually, such descriptions are simulatable, only, but not synthesizable.

In the next step, the design is divided into combinational logic and storage elements. This is called the Register Transfer Level (RTL). The storage elements (Flip Flops (FFs), latches) are controlled by a system clock. In synchronous designs, FFs should be used (driven by the edge of the clock signal) exclusively, because transparent latches (driven by the level of a control signal) are not spike-proof. For the description on RT level only 10 to 20 percent of all VHDL language constructs are needed and a strict methodology has to be followed. This description on RT level is called synthesizable description.

On the logic level, the design is represented as a netlist with logic gates (AND, OR, NOT, ...) and storage elements. The final layout is at the bottom of the hierarchy. The different cells of the target technology are placed on the chip and the connections are routed. After the layout has been verified, the circuit is ready for the production process.

1.3.3 Abstraction levels and VHDL



VHDL is applicable to the upper three abstraction levels. It is not suitable to describe a layout. The design entry in behavioural and RT level is usually done by text editors. Graphical tools are also available but experienced users often find it easier to write the code by hand. On the gate level, a schematic is modified as VHDL netlist descriptions tend to become too complex pretty soon.

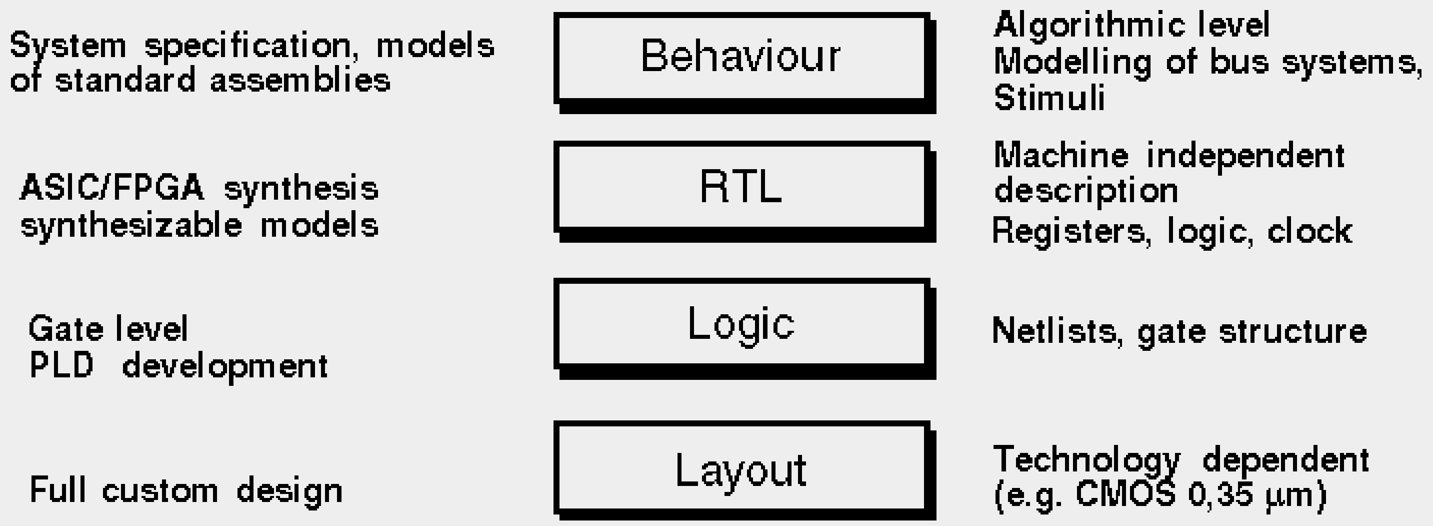
The transition from an upper abstraction level to a lower one is supported more or less efficiently by software.

Behaviourial synthesis is still a dream of many researchers as only very simple behaviour models are synthesizeable. A common application is the design of RAM cells for the target technology, where only the generic parameters (width, depth, number of ports, (a)synchronous, ...) need to be specified.

Logic synthesis, however, has been perfected in recent years. As long as the designer confines himself to certain simple VHDL constructs that are sufficient for RT level descriptions, the synthetis tools will be able to reproduce the behaviour in the logic level.

As a result of the ongoing research in efficient place and route algorithms the step from the logic level to the final layout has been widely automated for digital standard cell designs.

1.3.4 Description of Abstraction Levels



In the behaviour level, complete systems can be modelled. Bus systems or complex algorithms are described without considering synthesizability. The stimuli for simulation of RTL models are described in the behaviour level, for example. Stimuli are signal values of the input ports of the model and are described in the testbench, sometimes called validation bench.

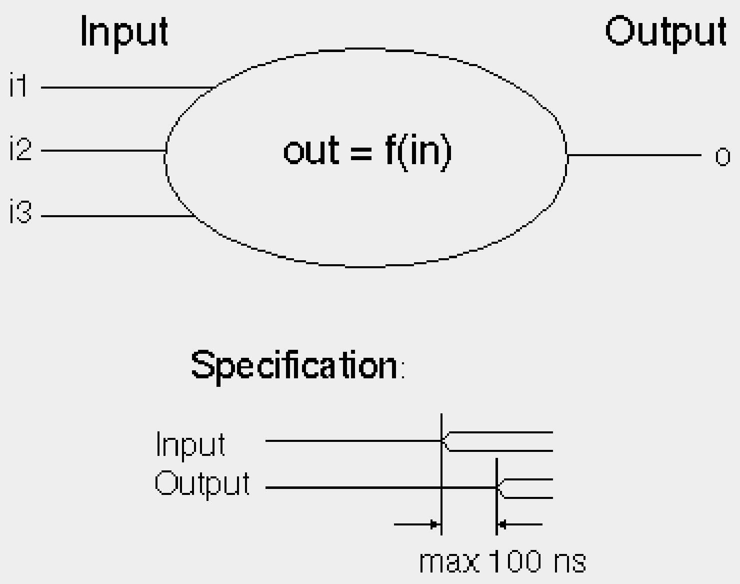
The designer has to take great care to find a consistent set of input stimuli that do not contradict the specification. The responses of the model have to be compared with the expected values which, in the simplest case, can be done with the help of a waveform diagram that shows the simulated signal values.

On the RT level, the system is described in terms of registers and logic that calculates the next value of the storage elements. It is possible to split the code into two blocks (cf. process statement) that contain either purely combinational logic or registers. The registers are connected to the clock signal and provide for synchronous behaviour. In practice, the strict separation of Flip Flops from combinational logic is often annulated and clocked processes describe the registers and the corresponding update functions.

The gate netlist is generated from the RT description with the help of a synthesis tool. For this task, a cell library for the target technology which holds the information about all available gates and their parameters (fan-in, fan-out, delay) is needed.

Based upon this gate netlist the circuit layout is generated. The resulting wire lengths can be converted into propagation delays which can be fed back into the gate level model (back annotation). This allows for thorough timing simulations without the need for additional simulator software.

1.3.5 Behavioural Description in VHDL

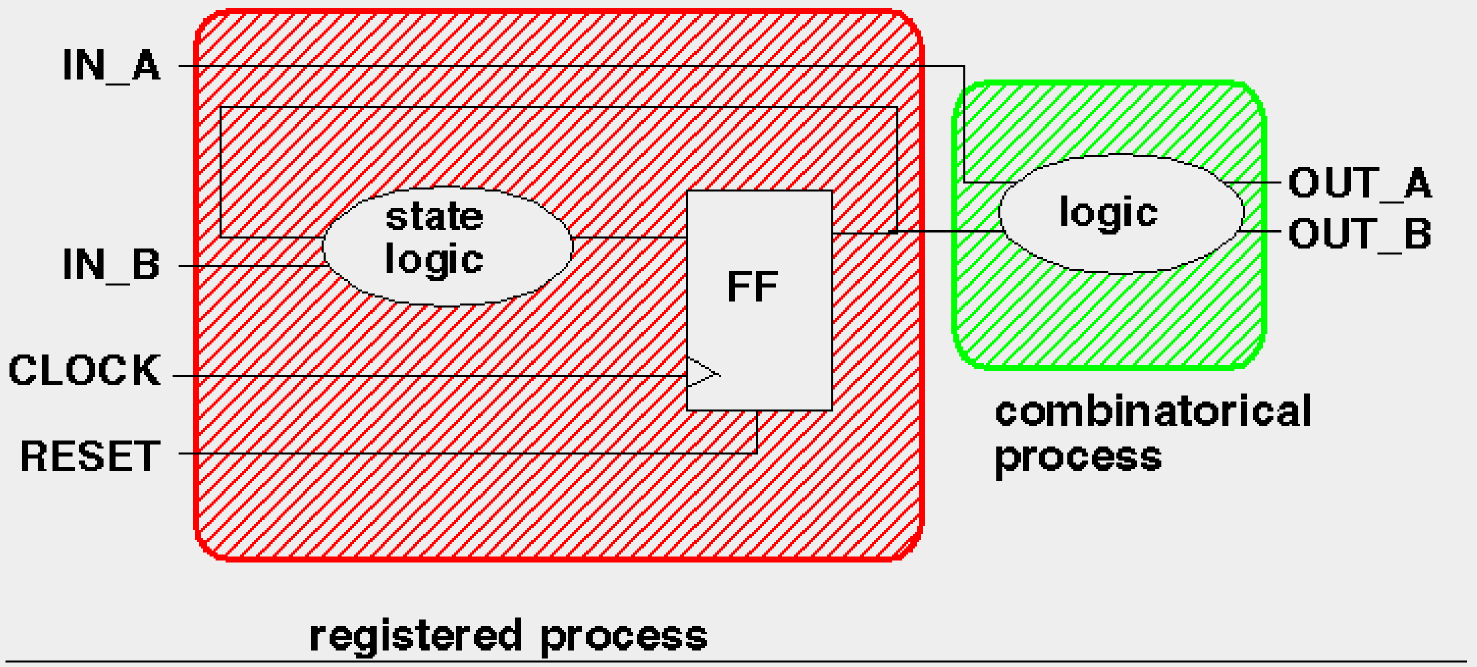


o <= transport i1 + i2 * i3 after 100 ns;

A simple specification of the function of a module is shown. The output o depends upon the three input values i1, i2 and i3. Furthermore it is specified that a new output value must be stable at the latest 100 ns after the input values have changed.

In a behavioural VHDL description, the function can be modeled as a simple equation (eg. i1 + i2 * i3) plus a delay of 100 ns. The worst case, i.e. that 100 ns are needed to calculate a new output value, is assumed here.

1.3.6 RT Level in VHDL

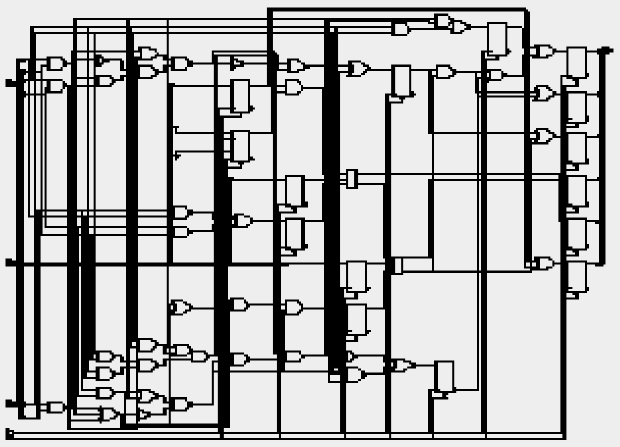


In VHDL functional behaviour is modeled with so called processes. Two different types of processes exist in RT level descriptions: the pure combinational process and the clocked process. All clocked processes infer FlipFlops and can be described in terms of state machine syntax.

In addition to the data input and data output signals, the control signals like the module clock (CLOCK) and reset (RESET) for asynchronously resetable FlipFlops have to be considered in modeling on RT level. When a synchronous reset strategy is employed, the reset input is treated like an ordinary data input.

It follows that RT level VHDL code also contains some sort of structural information in addition to the functional behaviour as storing and non-storing elements are separated. Timing issues in form of when signal values may be updated (eg. synchronously to the clock signal) are also considered.

1.3.7 Gate Level in VHDL

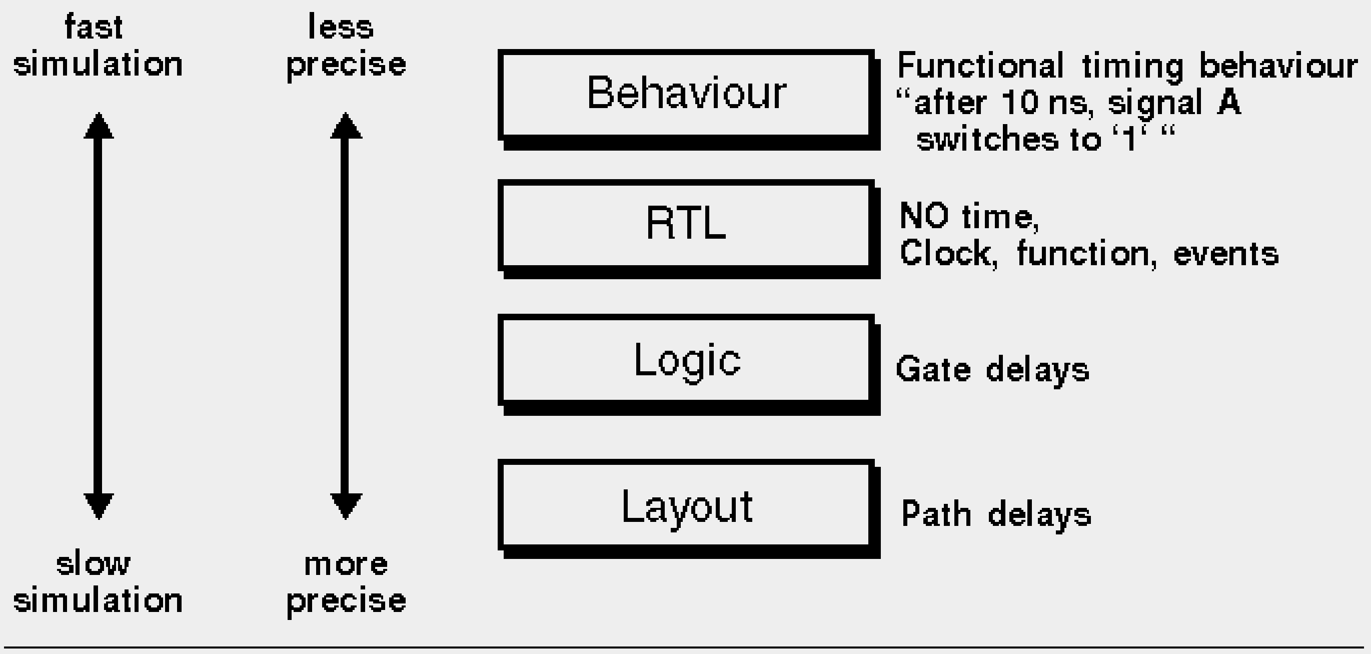


```
U86 : ND2 port map( A => n192, B => n191, Z => n188);
U87 : ND2 port map( A => I3_2, B => I2_0, Z => n175);
U88 : ND2 port map( A => I2_2, B => I3_0, Z => n173);
U89  : NR2 port map( A => mul_36_PROD_not_0,
         B => n174, Z => n185);
U90 : EN port map( A => n181, B => n182, Z => n180);
U91 : ND2 port map( A => I3_2, B => I2_1, Z => n181);
U92 : ND2 port map( A => I2_2, B => I3_1, Z => n182);
U93 : IVP port map( A => n180, Z => n192);
U94  : AO6 port map( A => n173, B => n174, C => n175,
         Z => n172);
U95 : NR2 port map( A => n174, B => n173, Z => n176);
U96 : ND2 port map( A => I3_1, B => I2_1, Z => n174);
U97  : EN port map( A => n183, B => n178,
         Z => product64_4);
U98  : ND3 port map( A => I2_2, B => I3_2, C => n174,
         Z => n183);
```

A VHDL gate level description contains a list of the gates (components) that are used in the design. Another part holds the actual instantiation of the components and lists their interconnection.

A schematic of the gate structure of a digital circuit can be seen on the left side of the picture. The right side shows a part of the corresponding VHDL description. Each single element of the circuit (eg. U86) is instantiated as a component (eg. ND2) and connected to the corresponding signals (n192, n191, n188). All used gates are part of the selected technology library where additional information like area, propagation delay, capacity, etc. is stored.

1.3.8 Information Content of Abstraction Levels


fast simulation
slow simulation
less precise
more precise
Behaviour
Functional timing behaviour "after 10 ns, signal A switches to '1' "
RTL
NO time, Clock, function, events
Logic
Gate delays
Layout
Path delays

The behaviour model is a simple way to describe the behaviour of a circuit, similar to usual software programming languages, such as PASCAL or C. With this description, only the functional behaviour can be simulated by a VHDL simulator.

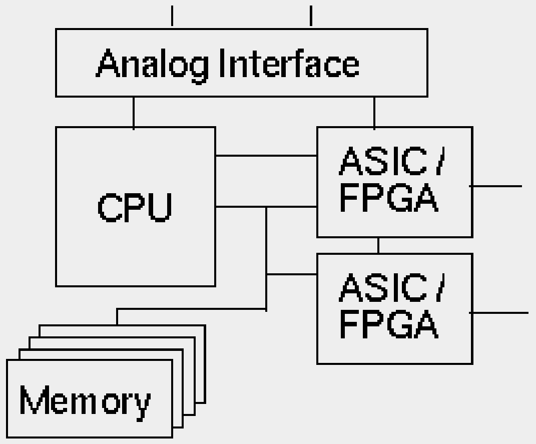
The clock pulse is the distinguishing mark for the RT level description. All operations are related to the clock signal. RT level simulations give no information about the real timing behaviour, which means that is impossible to tell, whether all signals have actually settled to stable values within one clock period or not.

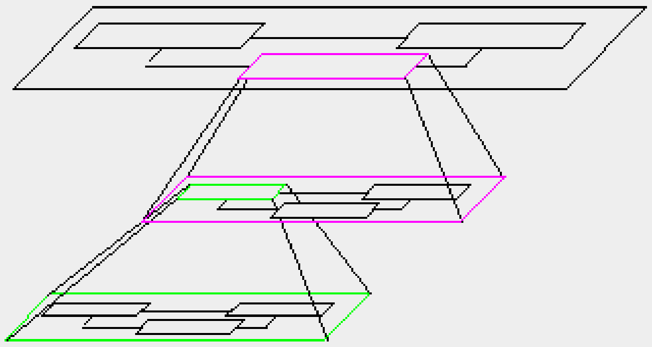
When the model is described on the logic level, delays can be applied to the used gates for simulation. The timing information is part of the synthesis library. This enables a rough validation of the timing behaviour. The uncertainty stems from the propagation delay along the signal wires which has not yet been considered. These delays may very well make up the main part of the entire delay in larger designs.

If the layout is completed, the wire lengths and thus the propagation delays will be known. The design can be simulated on gate level with the additional delay values and consequently the timing behaviour of the entire circuit can be validated. Yet, the simulation time grows considerably with the increased amount of information about the circuit, which restricts timing simulation to small parts of complex designs.

1.4 Modularity and Hierarchy

- **Partitioning in several partial designs**
- **Restrict complexity**
- **Enable teamwork**
- **Study of alternative implementations**
- **Soft macros**
- **Simulation models**





Modularity allows the partitioning of big functional blocks into smaller units and to group closely related parts in self-contained subblocks, so called modules. This way, a complex system can be devided into managable subsystems. The guidelines for partitioning can differ from design to design. Most of the time functional aspects are considered as partitioning constraint. The existance of well defined subsystems allows several designer to work in parallel on the same project as each designer will view his part as a new, complete system.

Hierarchy allows the building of a design out of modules which themselves may be built out of (sub-)modules. One level of a hierarchical description contains one or more modules, each module can even have different degrees of abstraction. These modules can themselves contain submodules which would be present the next lower hierarchical level.

Modularity and hierarchy help to simplify and organize a design project. Additional advantages are that different implementation alternatives can be examined for the modules, eg. in a simulation. Only the corresponding component instantiation needs to be changed for this in the overall model. Also analogues interfaces can be modeled in VHDL and added to the system model for simulation. Sometimes, simulation models of the devices that will be connected to the new design exist and can be used for a simulation of the design under test in its real working environment.

1.5 Summary

- **Hardware and software concepts**
- **Hardware is part of the system design**
- **Behavioural and RTL style**
- **Structure**
- **Concurrence (simultaneity)**
- **Sequential statements**
- **Description of timing behaviour is possible**
- **One language for model development and verification**

Hardware and software concepts are present in VHDL to model a digital system. There is a clear distinction between a pure behavioural model and RT level modeling for synthesis.

VHDL permits a structural (modular) and hierarchical description of a digital system.

Concurrency is an important concept of VHDL: Concurrent statements are executed virtually in parallel. The simulation is event driven. If a certain event occurs (eg. initiated by the stimulus), processes that depend on these events are triggered. These processes contain sequential statements which are evaluated one after another. Each process as a whole can be viewed as a concurrent statement. This way, the changes of signal values caused by the execution of several processes occur at the same time in the simulation.

Furthermore, it is possible to describe timing behaviour in VHDL. This eliminates the need for other languages for stimuli generation for test purposes or timing verification of the final design.

2. VHDL Language and Syntax

- **General**
- **Identifiers**
- **Naming Convention**

2.1 General

```
----------------------------------
-- Example VHDL Code --
----------------------------------

signal mySignal: bit;       -- an example signal

MYsignal <= '0',                    -- start with '0'
            '1' AFTER 10 ns,        -- and toggle after
            '0' after 10 ns,        -- every 10 ns
            '1' afTer 10 ns;
```

- **Case insensitive**
- **Comments: '--' until end of line**
- **Statements are terminated by ';'**
  (may span multiple lines)
- **List delimiter: ','**
- **Signal assignment: '<='**
- **User defined names:**
  - **letters, numbers, underscores**
  - **start with a letter**

VHDL is generally case insensitive which means that lower case and upper case letters are not distinguished. This can be exploited to define own rules for formatting the VHDL source code. VHDL keyword could for example be written in lower case letters and self defined identifiers in upper case letters. This convention is valid for the following slides.

Statements are terminated in VHDL with a semicolon. That means as many line breaks or other constructs as wanted can be inserted or left out. Only the semicolons are considered by the VHDL compiler.

List are normally separated by commas. Signal assignments are notated with the composite assignment operator '<='.

Self defined identifier as defined by the VHDL 87 standard may contain letters, numbers and underscores and must begin with a letter. Further no VHDL keywords may be used. The VHDL 93 standard allows to define identifiers more flexible as the next slide will show.

2.1.1 Identifier

---

mySignal_23 -- normal identifier rdy, RDY, Rdy -- identical identifiers vector_&_vector -- **X** : special character last of Zout -- **X** : white spaces idle__state -- **X** : consecutive underscores 24th_signal -- **X** : begins with a numeral open, register -- **X** : VHDL keywords	• **(Normal) Identifier** ❍ **Letters, numerals, underscores** ❍ **Case insensitiv** ❍ **No two consecutive underscores** ❍ **Must begin with a letter** ❍ **No VHDL keyword**
\mySignal_23\ -- extended identifier \rdy\, \RDY\, \Rdy\ -- different identifiers \vector_&_vector\ -- legal \last of Zout\ -- legal \idle__state\ -- legal \24th_signal\ -- legal \open\, \register\ -- legal	• **Extended Identifier (VHDL93)** ❍ **Enclosed in back slashes** ❍ **Case sensitive** ❍ **Graphical characters allowed** ❍ **May contain spaces and consecutive underscores** ❍ **VHDL keywords allowed**

Simple identifiers as defined by the VHDL 87 standard may contain letters, numbers and underscores. So 'mySignal_23' is a valid simple identifier. Further VHDL is case insensitive that means 'rdy', 'RDY' and 'Rdy' are identical. In particular the identifier has to begin with a letter, so '24th_signal' is not a valid identifier. Also not allowed are graphical characters, white spaces, consecutive underscores and VHDL keywords.

In the VHDL 93 standard a new type of identifiers is defined. They are called extended identifiers and are enclosed in back slashes. Within these back slashes nearly every combination of characters, numbers, white spaces and underscores is allowed. The only thing to consider is that extended identifiers are now case sensitive. So '/rdy/', '/RDY/' and '/Rdy/' are now three different identifiers.

2.1.2 Naming Convention

---

	architecture CONVENTION of NOTATION is end **architecure** CONVENTION ;	• **VHDL keywords are written in lower case letters** • **Importent parts are written in bold letters**
'93	**The keyword 'architecture' may be repeated after the keyword 'end'**	• **Explains syntax of the VHDL'93 standard**
⚠	**Output port modes have to match**	• **Pointing out particular issues to watch out**
	Not generally synthesizable	• **Pointing out synthesis aspects**

	The direction of arrays should always be defined the same way	• **Gives a tip in using the language effectively**

The naming convention are, that VHDL keywords are written in lower case letters while user defined identifiers are written in upper case letters. If something has to be highlighted it is done by writing it in bold letters.

There are several selfexplaining icons. They mark special issues about the VHDL'93 syntax (compared to that of VHDL'87), things to remark, synthesis aspects and special tips.

2.2 VHDL Structural Elements

- **Entity : Interface**
- **Architecture : Implementation, behaviour, function**
- **Configuration : Model chaining, structure, hierarchy**
- **Process : Concurrency, event controlled**
- **Package : Modular design, standard solution, data types, constants**
- **Library : Compilation, object code**

The main units in VHDL are entities, architectures, configurations and packages (together with package bodies).

While an entity describes an interface consisting of the port list most of the time, an architecture contains the description of the function of the corresponding module. In general, a configuration is used for simulation purposes, only. In fact, the configuration is the only simulatable object in VHDL as it explicitly selects the entity/architecture pairs to build the complete model. Packages hold the definition of commonly used used data types, constants and subprograms. By referencing a package, its content can be accessed and used.

Another important construct is the process. While statements in VHDL are generally concurrent in nature, this construct allows for a sequential execution of the assignments. The process itself, when viewed as a whole object, is concurrent. In reality, the process code is not always executed. Instead, it waits for certain events to occur and is suspended most of the time.

A library in VHDL is the logical name of a collection of compiled VHDL units (object code). This logical name has to be mapped by the corresponding simulation or synthesis tool to a physical path on the file system of the computer.

2.2.1 Declaration of VHDL Objects

	Entity	Architecture	Process/Subprogram	Package
Subprogram	x	x	x	x
Component		x		x
Configuration		x		
Constant	x	x	x	x
Datatype	x	x		

			x	x
Port	x			
Signal		x	x1	x
Variable			x2	

The table lists the legal places for the declaration of different objects:

A subprogram is similar to a function in C and can be called many times in a VHDL design. It can be declared in the declarative part of an entity, architecture, process or even another subprogram and in packages. As a subprogam is thought to be used in several places (architectures) it is useful to declare it in a package, always.

Components are necessary to include entity/architecture pairs in the architecture of the next higher hierarchy level. These components can only be declared in an architecture or a package. This is useful, if an entity/architecture pair might be used in several architectures as only one declaration is necessary in this case.

Configurations, themselves, are complete VHDL design units. But it is possible to declare configuration statements in the declarative part of an architecture. This possibility is only rarely used, however, as it is better to create an independent configuration for the whole model.

Constants and data types can be declared within all available objects.

Port declarations are allowed in entities, only. They list those architecture signals that are available as interface to other modules. Additional internal signals can be declared in architectures, processes, subprograms and packages. Please note that signals can not be declared in functions, a special type of a subprogram.

Generally, variables can only be declared in processes and subprograms. In VHDL'93, global variables are defined which can be declared in entities, architectures and packages.

---

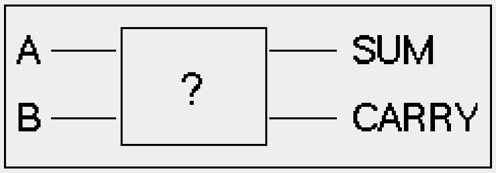
1. Signals may not be declared in functions

2. Global variables (VHDL '93) may also be declared in entities, architectures and packages

2.2.2 Entity

```
entity HALFADDER is
   port(
      A, B:           in   bit;
      SUM, CARRY: out bit);
end HALFADDER;
-- VHDL'93: end entity HALFADDER ;
```

- **Interface description**
- **No behavioural definition**



```
entity ADDER is
    port(
      A, B:           in     integer range 0 to 3;
      SUM:            out integer range 0 to 3;
      CARRY:          out bit );
end ADDER;
```

- **Linking via port signals**
  - **data types**
  - **signal width**
  - **signal direction**

'93 **The keyword 'entity' may be repeated after the keyword 'end'**

On the following pages, a fulladder consisting of two halfadders and an OR gate will be created step by step. We confine ourselves to a purely structural design, i.e. we are using gate level descriptions and do not need any synthesis tools. The idea is to demonstrate the interaction of the different VHDL objects in a straightforward manner.

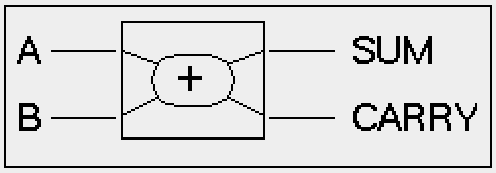
The interface between a module and its environment is described within the entity declaration which is initiated by the keyword ' **entity** '. It is followed by a user-defined, (hopefully) descriptive name, in this case: HALFADDER. The interface description is placed between the keyword ' **is** ' and the termination of the entity statement which consists of the keyword ' **end** ' and the name of the entity. In the new VHDL'93 standard the keyword ' **entity** ' may be repeated after the keyword ' **end** ' for consistency reasons.

The input and output signal names and their data types are defined in the port statement which is initiated by the keyword ' **port** '. The list of ports is enclosed in a '(' ')' pair. For each list element the port name(s) is given first, followed by a ':', the port mode and the data type. Within the list, the ';' symbol is used to separate elements, not to terminate a statement. Consequently, the last list element is not followed by a ';'!

Several ports with the same mode and data type can be declared by a single port statement when the port names are separated by ','. The port mode defines the data flow (in: input, i.e. the signal influences the module behaviour; out: output, i.e. the signal value is generated by the module) while the data type determines the value range for the signals during simulation.

2.2.3 Architecture

```
entity HALFADDER is
  port(
    A, B:            in   bit;
    SUM, CARRY: out bit);
end HALFADDER;

architecture RTL of HALFADDER is

begin

  SUM      <= A xor B;
  CARRY <= A and B;

end RTL;
-- VHDL'93: end architecture RTL ;
```



- **Implementation of the design**
- **Always connected with a specific entity**
  - **one entity can have several architecures**
  - **entity ports are available as signals within the architecture**
- **Contains concurrent statements**

'93 The keyword 'architecture' may be repeated after the keyword 'end'

The architecture contains the implementation for an entity which may be either a behavioural description (behavioural level or, if synthesizable, RT level) or a structural netlist or a mixture of those alternatives..

An architecture is strictly linked to a certain entity. An entity, however, may very well have several architectures underneath, e.g. different implementations of the same algorithm or different abstraction levels. Architectures of the same entity have to be named differently in order to be distinguishable. The name is placed after the keyword ' **architecture** ' which initiates an architecture statement. 'RTL' was chosen in this case.

It is followed by the keyword ' **of** ' and the name of entity that is used as interface ('HALFADDER'). The architecture header is terminated by the keyword ' **is** ', like in entity statements. In this case, however, the keyword ' **begin** ' must be placed somewhere before the statement is terminated. This is done the same way as in entity statements: The keyword ' **end** ', followed by the architecture name. Once again, the keyword ' **architecture** ' may be repeated after the keyword ' **end** ' in VHDL'93.

As the VHDL code is synthesizable, RTL was chosen as architecture name. In case of this simple function, however, there is no difference to behavioural (algorithmic) description. We will use 'BEHAVE', 'RTL', 'GATE', 'STRUCT' and 'TEST' to indicate the abstraction level and the implemented behaviour, respectively. The name 'EXAMPLE' will be used whenever the architecture shows the application of new VHDL elements and is not associated with a specific entity.

2.2.4 Architecture Structure

```
architecture  EXAMPLE  of
 STRUCTURE  is

  subtype DIGIT is integer range 0 to 9;

  constant BASE: integer := 10;

  signal DIGIT_A, DIGIT_B: DIGIT;
  signal CARRY:               DIGIT;

begin

  DIGIT_A <= 3;

  SUM <= DIGIT_A + DIGIT_B;

  DIGIT_B <= 7;

  CARRY <= 0 when SUM < BASE else
                 1;

end EXAMPLE ;
```

- **Declarative part:**
  - **data types**
  - **constants**
  - **additional signals ("actual" signals)**
  - **components**
  - **...**
- **Definition part (after 'begin'):**
  - **signal assignments**
  - **processes**
  - **component instantiations**
  - **concurrent statements: order not important**

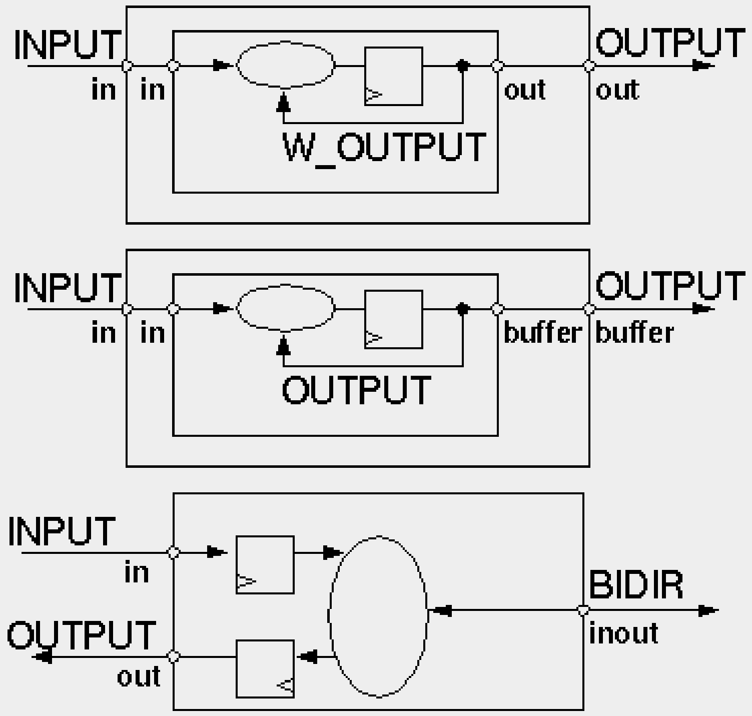
Each architecture is split into an optional declarative part and the definition part.

The declarative part is located between the keywords ' **is** ' and ' **begin** '. New objects that are needed only within the architecture constants, datatypes, signals, subprograms, etc. can be declared here.

The definition part is initiated by the keyword ' **begin** ' and holds concurrent statements. These can be simple signal assignments, process statements, which group together sequential statements, and component instantiations. Concurrency means that the order in which they appear in the VHDL code is not important. The signal SUM, for example, gets always the result of (3 + 7), independently of the location of the two assignments to the signals DIGIT_A and DIGIT_B.

Signal assignments are carried out by the signal assignment operator ' <= '. The symbol represents the data flow, i.e. the target signal whose value shall be updated is placed on the left side of the operator. The right side holds an expression that evaluates to the new signal value. The data types on the left and on the right side have to be identical. Please remember that the signals that are used in this example were defined implicitly by the port declaration of the entity.

2.2.5 Entity Port Modes



- **in:**
  - **signal values are read-only**
- **out:**
  - **signal values are write-only**
  - **multiple drivers**
- **buffer:**
  - **comparable to out**
  - **signal values may be read, as well**
  - **only 1 driver**
- **inout:**
  - **bidirectional port**

Output port modes have to match

The mode of an entity port restricts the direction of the data flow. The port mode ' **in** ' is used to classify those signals that are only read in the underlying architecture. It is not possible to update their values.

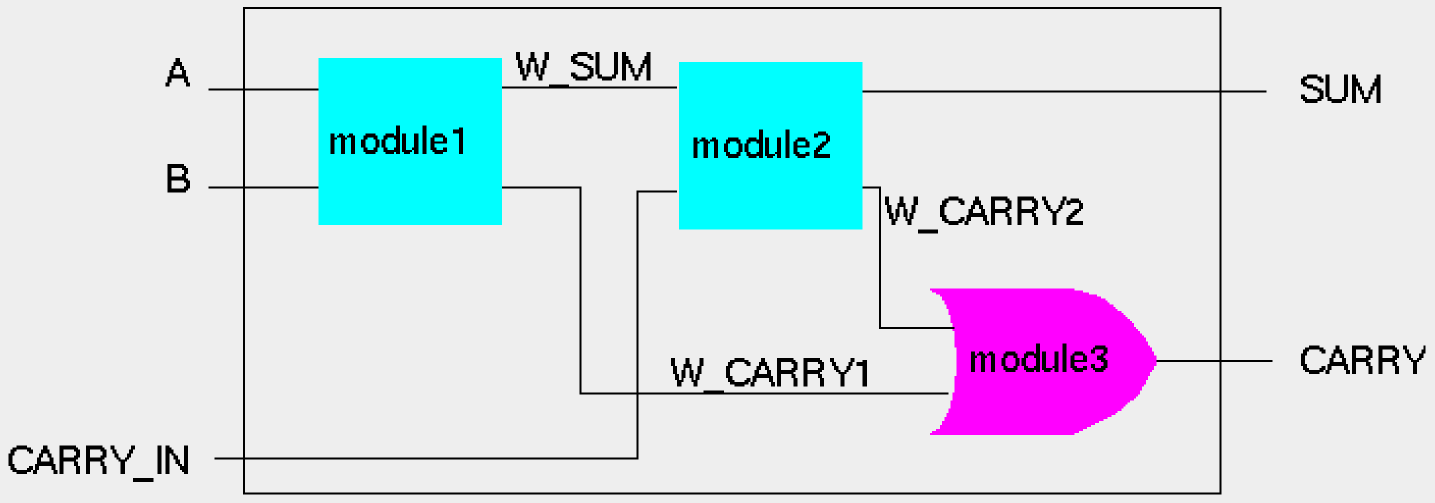
Likewise, the port mode ' **out** ' denotes signals whose values are generated by the architecture. Their values can not be used to influence the behaviour in any form. If the current output value has to be used to calculate the next signal value, e.g. within a counter module, an intermediate signal must be declared. Internal signals do not have a data flow direction associated with them!

Alternatively it is possible to use the port mode ' **buffer** '. This eliminates the need for an additional signal declaration. However, there is just a single source allowed for these signals.

In order to model busses, where multiple units have access to the same data lines, either the port mode ' **out** ' has to be used, if each unit is only writing to this data bus, or the port mode ' **inout** ' which allows a bidirectional data flow.

Please note that the port modes have to match, if the output port of a submodule is connected directly to the output port of the entity on a higher hierarchy level. At the worst, intermediate signals have to be declared to avoid compilation errors.

2.2.6 Hierarchical Model Layout



Full adder: 2 halfadders + 1 OR-gate

VHDL allows for a hierarchical model layout, which means that a module can be assembled out of several submodules. The connections between these submodules are defined within the architecture of a top module. As you can see, a fulladder can be built with the help of two halfadders (module1, module2) and an OR gate (module3).

A purely structural architecture does not describe any functionality and contains just a list of components, their instantiation and the definition of their interconnections.

2.2.7 Component Declaration

```
entity FULLADDER is
   port (A,B, CARRY_IN: in   bit;
           SUM, CARRY:     out bit);
end FULLADDER;

architecture STRUCT of FULLADDER is
   signal W_SUM, W_CARRY1, W_CARRY2 : bit;

    component  HALFADDER
      port (A, B :              in   bit;
              SUM, CARRY : out bit);
    end component;

    component  ORGATE
      port (A, B : in   bit;
              RES : out bit);
    end component;

begin
. . .
```

- **In declarative part of architecture**
- **Comparable to a `socket`-type**

The component port-list does not replace the declaration of connecting signals (local objects, only)

The entity of the fulladder can be derived directly from the block diagram. The inputs A and B, as well as a CARRY_IN input are required, together with the SUM and the CARRY signals that serve as outputs.

As the fulladder consists of several submodules, they have to be "introduced" first. In a component declaration all module types which will be used, are declared. This declaration has to occur before the 'begin' keyword of the architecture stement. Note, that just the interface of the modules is given here and their use still remains unspecified. The component declaration is therefore comparable with a socket definition, which can be used once or several times and into which the appropriate entity is inserted later on. The **port list elements of the component** are called local elements, which means that they **are not signals** !

In this case, only two different sockets, namely the socket HALFADDER and the socket ORGATE are needed. Arbitrary names may be chosen for the components, yet it is advisable to use the name of the entity that will be used later on. Additionally, the port declaration should also be identical. This is absolutely necessary, when the design is to be synthesized, as the software ignores VHDL configuration statements and applies the default rules.

2.2.8 Component Instantiation

```
architecture STRUCT of FULLADDER is
  component HALFADDER
    port (A, B :              in   bit;
            SUM, CARRY : out bit);
  end component;
  component ORGATE
    port (A, B : in   bit;
            RES : out bit);
  end component;

  signal W_SUM, W_CARRY1, W_CARRY2: bit;

begin

    MODULE1: HALFADDER
     port map( A, B, W_SUM, W_CARRY1 );

    MODULE2: HALFADDER
     port map ( W_SUM, CARRY_IN,
                    SUM, W_CARRY2 );

    MODULE3: ORGATE
     port map ( W_CARRY2, W_CARRY1, CARRY );

end STRUCT;
```

- **Socket generation**
- **How many do I need?**
- **Instantiation in definition part of architecture (after 'begin')**
- **Places socket on PCB**
- **Wires signals:**
  - **default: positional association**

If a component has been declared, that means the socket type is fixed, it can be used as often as necessary. This is done in form of component instantiations, where the actual socket is generated. This is comparable to the placement of sockets on a printed circuit board (PCB). The entity/architecture pair that provides the functionality of the component is inserted into the socket at a later time when the configuration of a VHDL design is built.

Each component instance is given a unique name (label) by the designer, together with the name of the component itself. Component instantiations occur in the definition part of an architecture (after the keyword 'begin'). The choice of components is restricted to those that are already declared, either in the declarative part of the architecture or in a package.

As the component ports or socket pins have to be connected to the rest of the circuit, a port map statement is necessary. It has to list the names of the architecture signals that shall be used. As default, the so called positional association rules apply, i.e. the first signal of the port map list is connected to the first port from the component declaration, etc.

2.2.9 Component Instantiation: Named Signal Asscociation

```
entity FULLADDER is
  port (A,B, CARRY_IN: in   bit;
        SUM, CARRY:     out bit);
end FULLADDER;

architecture STRUCT of FULLADDER is

  component HALFADDER
    port (A, B :          in bit;
          SUM, CARRY : out bit);
  end component;
. . .
  signal W_SUM, W_CARRY1, W_CARRY2 : bit;

begin

   MODULE1: HALFADDER
            port map (  A       => A,
                        SUM     => W_SUM,
                        B       => B,
                        CARRY => W_CARRY1 );
  . . .
end STRUCT;
```

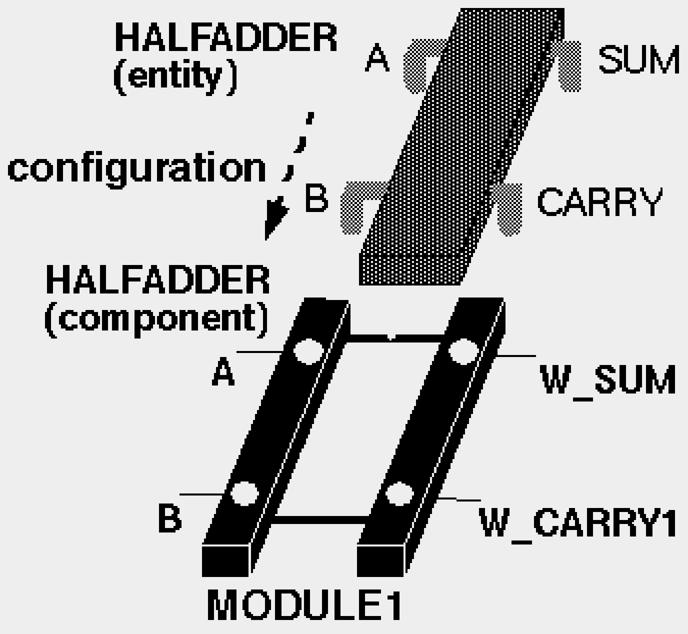
- **Named association:**
  - **left side: "formals" (port names from component declaration)**
  - **right side: "actuals" (architecture signals)**
- **Independent of order in component declaration**

Instead of the positional association that was used in the previous example it is also possible to connect architecture signals directly to specific ports. This is done by the so called named association where the order of the signals is not restricted. The port names from the component declaration, also called "formals", are associated with an arrow ' => ' with the signals of the entity ("actuals").

In the example, the output port SUM is declared third in the component declaration. In the port map statement, however, this port is connected to the signal W_SUM in the second place. Please note that the list elements are separated by ',' symbols in the port map statement unlike the ';' symbols that are used in port declarations.

2.2.10 Configuration

```
entity HALFADDER is
  port(A, B:          in   bit;
       SUM, CARRY: out bit);
end HALFADDER;

. . .

  component HALFADDER
    port(A, B:          in   bit;
         SUM, CARRY: out bit);
  end HALFADDER;

  signal W_SUM, W_CARRY1, W_CARRY2: bit;

. . .

   MODULE1 : HALFADDER
    port map(A, B, W_SUM, W_CARRY1);
```



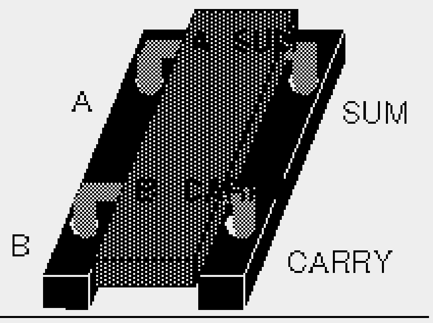
'93 **Entities may be instantiated directly without a preceding component declaration**

Component declaration and instantiation are independet of VHDL models that are actually available. It is the task of the VHDL configuration to link the components to entity/architecture pairs in order to build the complete design. In summary: A component declaration provides a certain kind of socket that can be placed on the circuit as often as necessary with component instantiations. The actual insertion of a device into the instantiated sockets is done by the configuration.

In VHDL'93 it is possible to omit the component declaration and to instantiate entities directly.

2.2.11 Configuration: Task and Application

```
entity FULLADDER is
. . .
end FULLADDER;

architecture STRUCT of FULLADDER is
. . .
end STRUCT;

configuration CFG_FULLADDER of FULLADDER
is
   for STRUCT   -- select architecture STRUCT
     -- use default configuration rules
   end for;
end configuration CFG_FULLADDER ;
```



- **Selects architecture for top-level entity**
- **Selects entity/architecture pairs for instantiated components**
- **Generates the hierarchy**
- **Creates a simulatable object**
- **Default binding rules:**
  - **selects entity with same name as component**
  - **signals are associated by name**
  - **last compiled architecture is used**

The keyword 'configuration' may be repeated after the keyword 'end'

The connection between the entity and the architecture that is supposed to be used for the current simulation is established in the configuration, i.e. it creates the final design hierarchy. This includes the selection of the architecture for the top-level entity. The configuration is the only VHDL object that can be simulated or synthesized. While it is possible to control the configuration process manually for simulation purposes, synthesis tools always apply the default rule set.

For this to succeed, the component names have to match the names of existing entities. Additionally, the port names, modes and data types have to coincide - the order of the ports in the component declaration is ignored. The most recently analysed architecture for the specific entity will be selected as corresponding architecture.

The example shows the default configuration for a structural architecture. Some simulators require an explicit configuration definition of this kind the top-level entity. A configuration refers to a specific entity, which is FULLADDER in this case. The architecture STRUCT is selected with the first 'for' statement. As no additional configuration commands are given, the default rules apply for all other components.

2.2.12 Configuration: Example (1)

```
entity A is
   port(A, B:                in   bit;
          SUM, CARRY: out bit);
end A;

architecture RTL of A is
. . .
```

```
entity B is
   port(U,V: in   bit;
          X,Y: out bit);
end B;

architecture GATE of B is
. . .
```

```
entity FULLADDER is
   port(A, B, CARRY_IN: in   bit;
          SUM, CARRY:      out bit);
end FULLADDER;
architecture STRUCT of FULLADDER is

   component HALFADDER
      port(A, B:                in   bit;
             SUM, CARRY: out bit);
   . . .

   signal W_SUM, W_CARRY1, W_CARRY2: bit;

begin
   MODULE1: HALFADDER
      port map (A, B, W_SUM, W_CARRY1);

   MODULE2: HALFADDER
      port map(W_SUM, CARRY_IN, SUM, W_CARRY2);
   . . .

end STRUCT;
```

Please have a look at the VHDL code fragments in order to understand a more elaborated configuration example:

In the end, a fulladder shall be simulated again. The structure of this fulladder is the same as in the example before, i.e. two halfadders are used. Each halfadder is declared to have two signals of data type 'bit' as input and output, respectively. The component ports are connected to the architecture's signals by position, i.e. the first signal is connected to the first port.

An entity named HALFADDER shall not be available, however, and the two entities A and B that also have different architectures named RTL and GATE, respectively are to be used. Both entities match the ports from the component declaration.

2.2.13 Configuration: Example (2)

```
configuration
CFG_FULLADDER of
FULLADDER is
  for STRUCT
     for MODULE2:
HALFADDER
       use entity
work.B(GATE);
       port map ( U => A,
                  V => B,
                  X => SUM,
                  Y => CARRY );
     end for;

     for others : HALFADDER
       use entity work.A(RTL);
     end for;
  end for;
end CFG_FULLADDER;
```

- **Entity/architecture pairs may be selected by use of**
  - **instance names**
  - **'all': all instances of the specified component**
  - **'others': all instances not explicitly mentioned**
- **If the port names differ => port map clause**
- **Possible to reference an existing configuration of a submodule**

Again, the architecture STRUCT is selected for the FULLADDER entity. Within this for loop, however, the entities and architectures for the subordinated components are selected.

For this, the for statement is used again. The first name after the keyword ' **for** ' names the component instantiation, followed by a ':' and the component name. The keyword 'all' can be used, if all instances of a component shall be adressed. Within the for loop, the use statement selects the entity by specifying the absolute path to that object. Unless explicitly changed, all VHDL objects are compiled into the library work. The architecture for the selected entity is enclosed in a '(' ')' pair.

As the port names of the entity B do not match the port names from the component declaration a port map statement is necessary again.

Again, it is possible to map the names by positional association, yet an explicit names association should always be used to enhance readability. In this case, the formal parameters are the port names of the entity, while the component port names are used as actuals.

It is also possible to address all those components that have not been configured yet with the keyword ' **others** '. This is necessary in this case as there does not exist an entity named HALFADDER. Instead, the entity A and the corresponding architecture RTL is used for all HALFADDER instantiations other than MODULE2. A port map clause is not necessary as the entity port names are equivalent to the names of the component.

All other components that might exist are treated according to the default configuration rules.

In order to simplify the hierarchy definition of large designs it is often useful to define the configuration for the submodules and to reference these configurations from the top level.

2.2.14 Process

```
entity AND_OR_XOR is
  port (A,B :                    in    bit;
        Z_OR, Z_AND, Z_XOR : out bit);
end AND_OR_XOR;

architecture RTL of AND_OR_XOR is
begin

  A_O_X:  process  (A, B) <-- sensitivity list
   begin
     Z_OR   <= A or    B;
     Z_AND <= A and B;
     Z_XOR <= A xor  B;
    end process  A_O_X ;

end RTL;
```

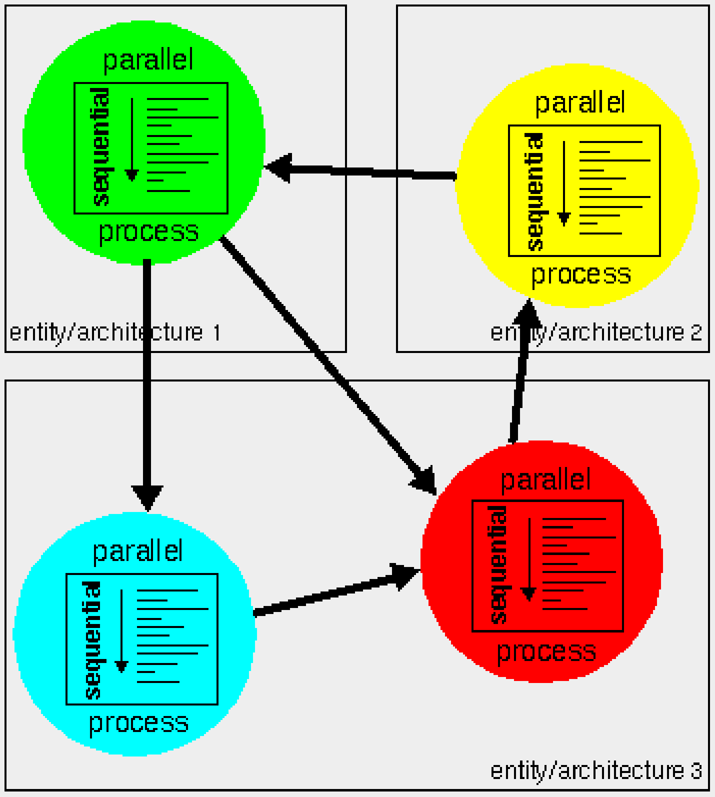
- **Contains sequentially executed statements**
- **Exist within an architecture, only**
- **Several processes run concurrently**
- **Execution is controlled either via**
  - **sensitivity list (contains trigger signals), or**
  - **wait-statements**
- **The process label is optional**

Because the statements within an architecture operate concurrently another VHDL construct is necessary to achieve sequential behaviour. A process, as a whole, is treated concurrently like any other statement in an architecture and contains statements that are executed one after another like in conventional programming languages. In fact it is possible to use the process statement as the only concurrent VHDL statement.

The execution of a process is triggered by events. Either the possible event sources are listed in the sensitivity list or explicit wait statements are used to control the flow of execution. These two options are mutually exclusive, i.e. no wait statements are allowed in a process with sensitivity list. While the sensitivity list is usually ignored by synthesis tools, a VHDL simulator will invoke the process code whenever the value of at least one of the listed signals changes. Consequently, all signals that are read in a purely combinational process, i.e. that influence the behaviour, have to be mentioned in the sensitivity list if the simulation is to produce the same results as the synthesized hardware. Of course the same is true for clocked processes, yet new register values are to be calculated with every active clock edge, only. Therefore the sensitivity list contains the clock signal and asynchronous control signals (e.g. reset).

A process statement starts with an optional label and a ':' symbol, followed by the ' **process** ' keyword. The sensitivity list is also optional and is enclosed in a '(' ')' pair. Similar to the architecture statement, a declarative part exists between the header code and the keyword ' **begin** '. The sequential statements are enclosed between ' **begin** ' and ' **end process** '. The keyword ' **process** ' has to be repeated! If a label was chosen for the process, it has to be repeated in the end statement, as well.

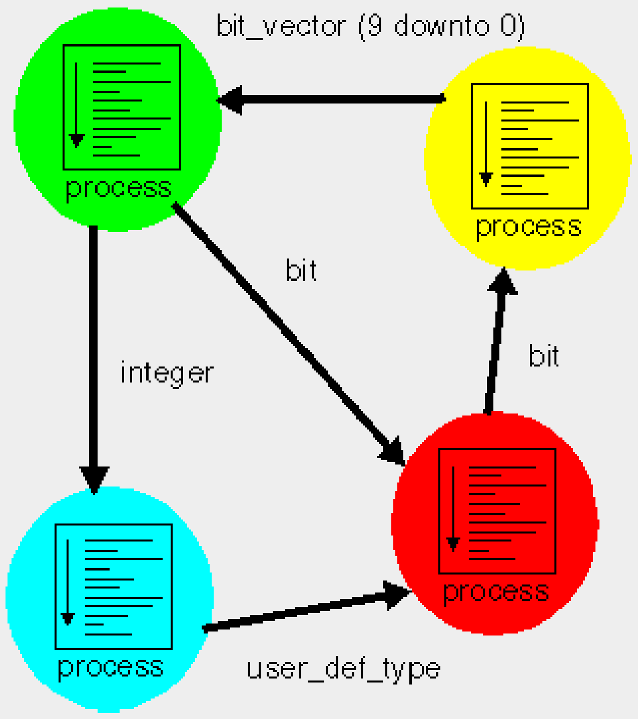
2.2.15 VHDL Communication Model



- Processes are concurrent statements
- Several processes
  - run parallel
  - linked by signals in the sensitivity list
  - sequential execution of statements
- Link to processes of other entity/architecture pairs via entity interface

Process statements are concurrent statements while the instructions within each process are executed sequentially, i.e. one after another. All processes of a VHDL design run in parallel, no matter in which entity or hierarchy level they are located. They communicate with each other via signals. These signals need to be ports of the entities if processes from different architectures depend from another.

2.2.16 Signals



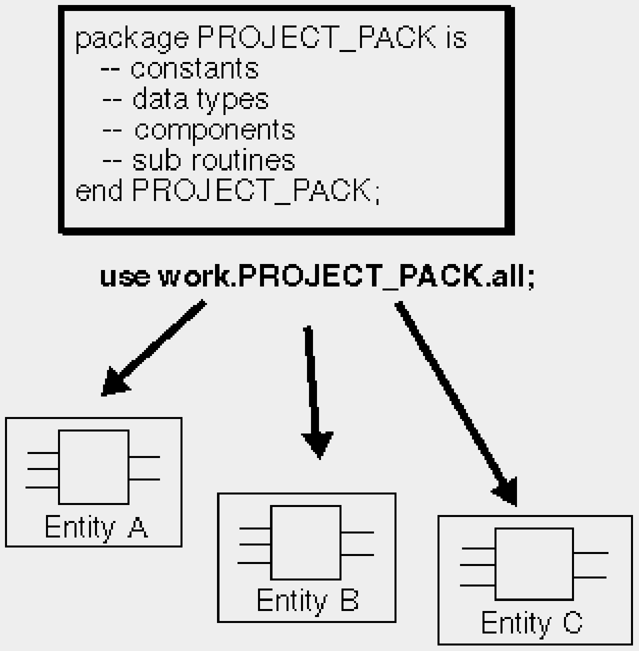
- **Every signal has a specific data type**
  - **number of possible values**
- **Predefined data types**
  - **bit, bit_vector, integer, real, ...**
- **User-defined data types**
  - **more accurate hardware model**
  - **enhanced readability**
  - **improved error detection**

Each signal has a predetermined data type which limits the amount of possible values for this signal. Synthesizable data types offer only a limited number of values, i.e. it is possible to map these values to a certain number of wires. Only the most basic data types are already predefined in VHDL, like bit, bit vectors and integer.

The user can define his own data types which might become necessary to enhance the accuracy of the model (tristate drivers, for example, may be set to high impedance instead of a low or high voltage level), for better readability (e.g. a signal value called "IDLE" tells more about its function than "00101"or "17") and to allow for automatic error detection (e.g. by restricting the range of legal values).

2.2.17 Package

- **Collection of definitions, datatypes, subprograms**
- **Reference made by the design team**
- **Any changes are known to the team immediately**
  - **same data types ("downto vs. to")**
  - **extended functions for all**
  - **clearing errors for all**

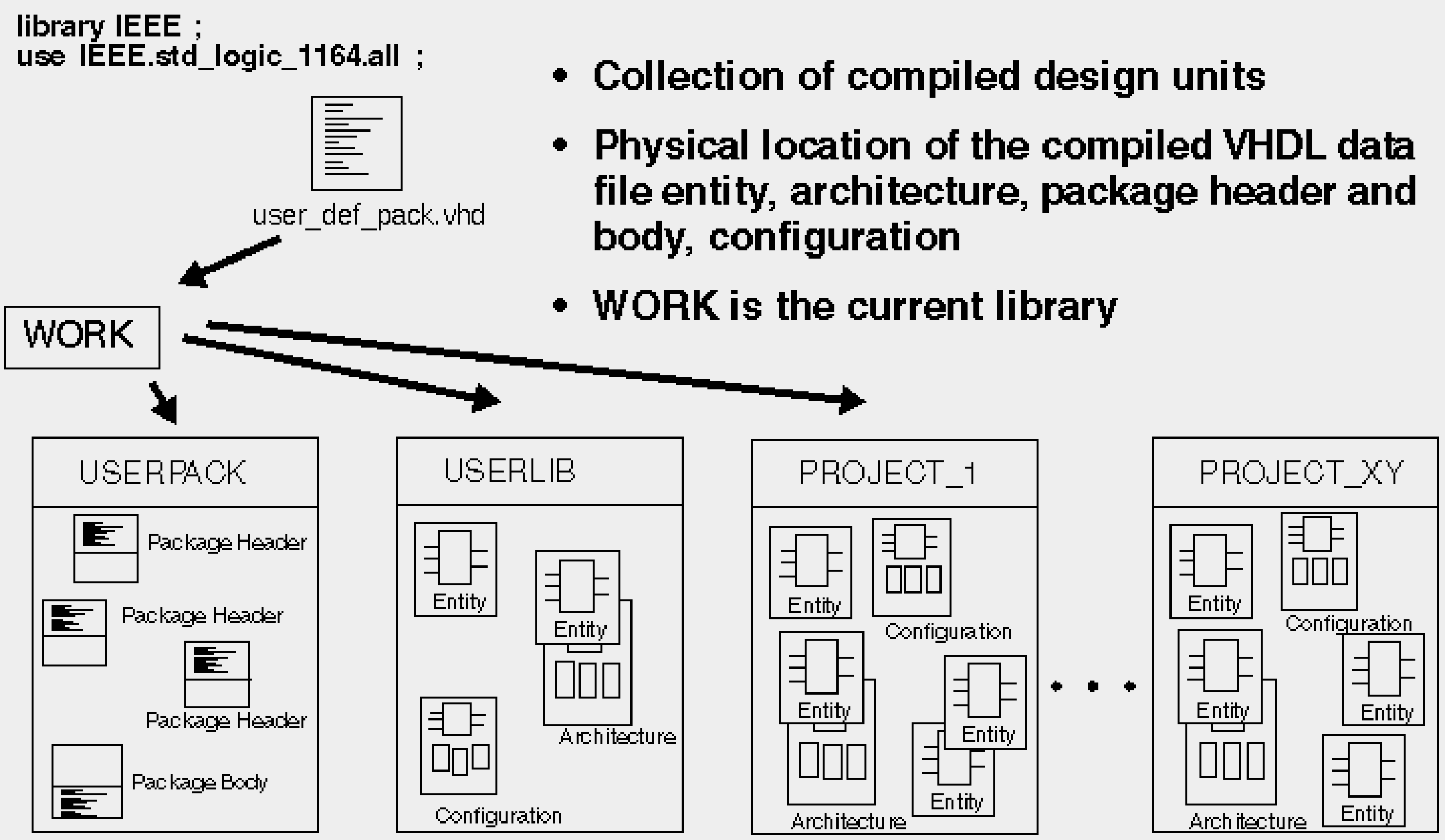


A package is a collection of definitions of data types, subprograms, constants etc. This is especially usefull in teamwork situations where everyone should work with the same data types, e.g. the same orientation of a vector range. This simplifies the connection of the modules of different designers to the complete VHDL model later on. Necessary changes are also circularized immediately to all persons concerned.

It is possible to split a package into a header and a body section. The package header contains prototype declarations of functions or procedures, the definition of all required data types and so on. The actual implementation of the subprograms can be placed in the body section. This simplifies the compilation process, because only the usually rather short package header must be read in order to decide whether the current VHDL code conforms to the previous declarations/definitions.

A package is referenced by a use clause. After the keyword ' **use** ' follows the so called "selected name". This name consists of the library name where the compiled package has been placed, the package name itself and the object name which will be referenced. Usually, the keyword ' **all** ' is used to reference all visible objects of the package.

2.2.18 Library

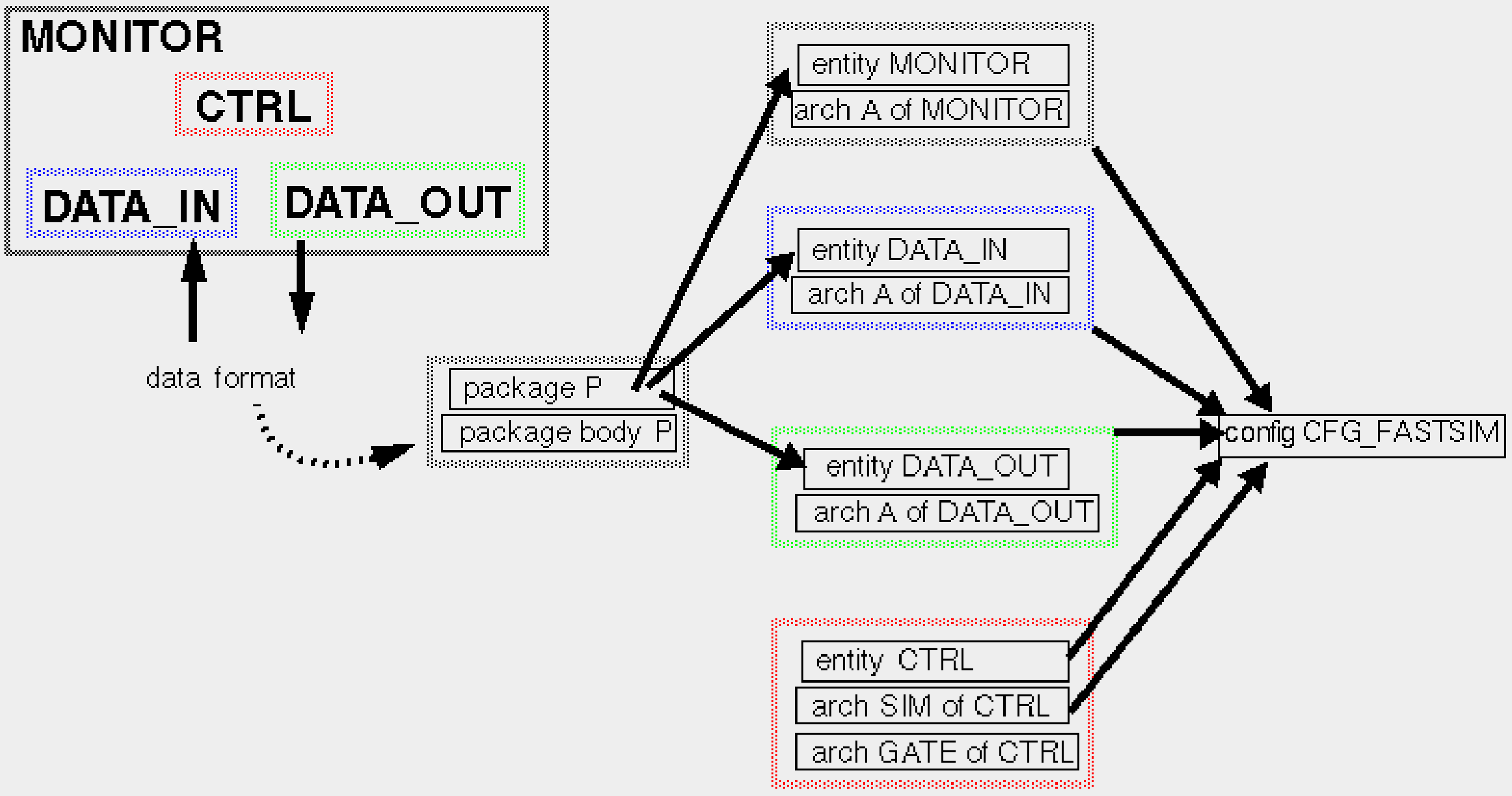


- Collection of compiled design units
- Physical location of the compiled VHDL data file entity, architecture, package header and body, configuration
- WORK is the current library

All analysed objects as there are packages, package bodies, entities, architectures and configurations can be found in a library. In VHDL, the library is a logical name with which compiled objects can be grouped and referenced. The default library is called "work". This logical name can be mapped to another logical library name as shown in the picture, but it has to be mapped to a physical path on a storing device eventually.

Usually, every designer operates within his own work library. Yet he can use units from other libraries which might hold data from former projects (PROJEKT_1 and PROJEKT_XY) or the current project packages (USERPACK). If another library than WORK is to be used, it will have to be made visible to the VHDL compiler. This is done with the library statement that starts with the keyword ' **library** ', followed by the logical name of the library. For example the library IEEE is commonly used because it contains standardized packages.

2.2.19 Design Structure: Example



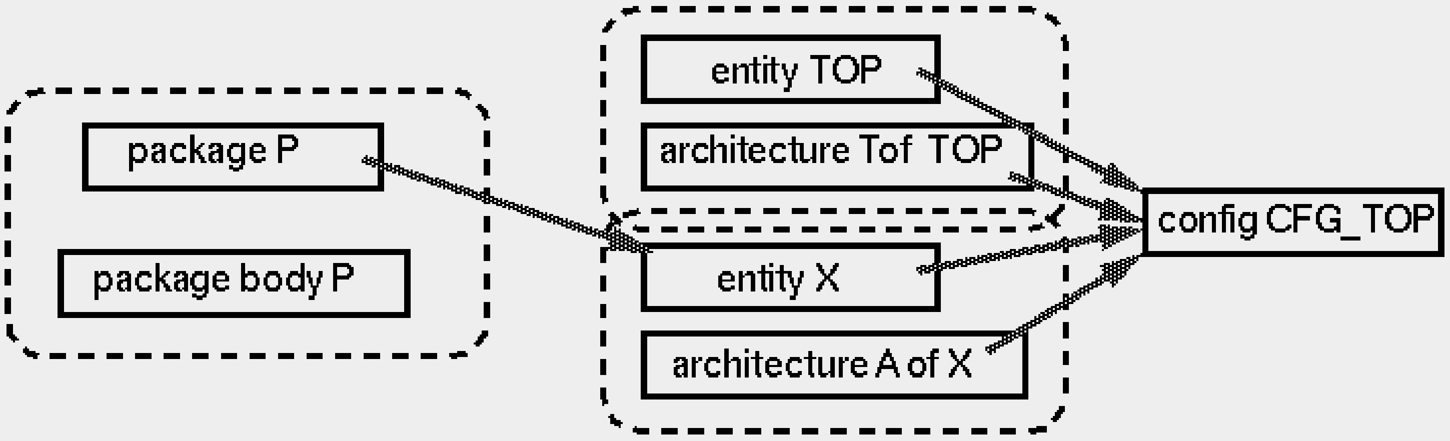
In the example, the design consists of four modules. The top level is the module MONITOR which uses three other submodules. These other modules are called CTRL, DATA_IN and DATA_OUT.

The data types for the data format that will be monitored are defined in a package P as the data types might be used in other design which communicate with this one. A separate package body has also been written. The package P is referenced by the entities MONITOR, DATA_IN and DATA_OUT as these three modules will use the new data types. The CTRL module will process control signals, only, which can be modeled with the predefined data types. The entities MONITOR, DATA_IN and DATA_OUT each have a architecture A. Two different architectures (SIM and GATE) exist for the entity CTRL. A configuration is necessary for the simulation.

Secondary units like package bodies and architectures are linked automatically to their primary units (package and entities). Other links have to be made explicitly. Therefore the package P needs to be referenced with a use clause before the entities are declared, while this is not necessary for the corresponding architectures. The assembly of the final design is made by the configuration, i.e. hierarchy errors like incompatible interfaces will be reported when the configuration is analysed. If several architectures exist for one entity, the configuration will also select the architecture that is to be used for the current simulation.

2.2.20 Sequence of Compilation

- **Primary units are analysed before secondary units**
  - **entity before architecture**
  - **package before package body**
- **All units that are referred to have to be analysed first**
  - **entity after package**
  - **configuration after entity/architecture**

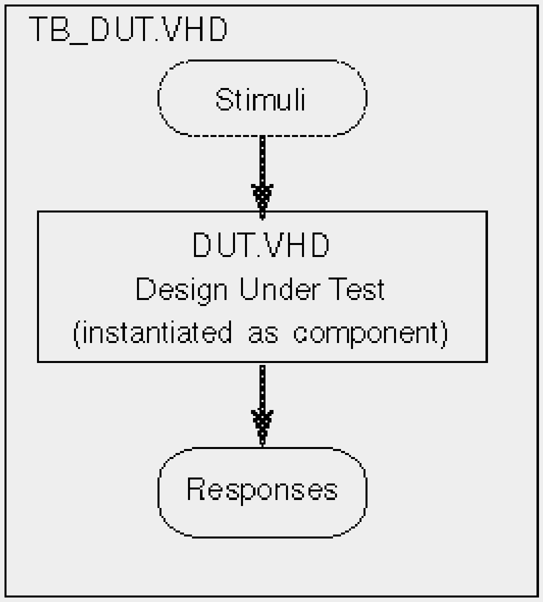


The sequence of compilation is predetermined by the dependency of model parts. If a dependency tree was built, one would have to compile from the lowest level of hierarchy to the top.

As secondary units rely on information given in their primary units (e.g. the interface signals have to be known for an architecture), they can only be compiled when the corresponding primary unit has already been compiled before. Consequently, primary units have to be analysed before their secondary units (entity before architecture, package header before package body).

The same reason applies for references. A package, for example, has to be analysed before an entity which references this package can be compiled, because the entity or its architecture(s) need the information about the data types, etc. The second rule is to compile modules which are referenced by others before the modules that are actually referencing it. Therefore the configuration, which builds up the design hierarchy, has to be analysed at last.

2.2.21 Outlook: Testbench



- **VHDL code for top level**
- **No interface signals**
- **Instantiation of design**
- **Statements for stimuli generation**
- **Simple testbenches: response analysis by waveform inspection**
- **Sophisticated testbenches may need >50% of complete project ressources**

The most commonly used method to verify a design is simulation. As VHDL was developed for the simulation of digital hardware in the first place, this is well supported by the language.

A new top level, usually called testbench, is created which instantiates Design Under Test (DUT) and models its environment. Therefore, the entity of this top level has no interface signals. The architecture will also contain some processes or submodules which generate the stimuli for the DUT and sometimes addtional processes or submodules which simplify the analysis of the responses of the DUT.

The effort which is invested into the creation of a testbench varies considerably and can cost the same amount of the time as the modeling of the DUT. It depends on the type of testbench, i.e. how much functionality (stimuli generation, response analysis, file I/O, etc.) has to be supplied.

2.2.22 Simple Testbench Example

```
entity ADDER IS
port (A,B :              in   bit;
CARRY,SUM : out bit);
end ADDER;
architecture RTL of ADDER is
begin
ADD: process (A,B)
begin
SUM <= A xor B;
CARRY <= A and B;
end process ADD;
end RTL;
```

```
entity TB_ADDER IS
end TB_ADDER;

architecture TEST of TB_ADDER is
  component ADDER
port (A, B:               in   bit;
CARRY, SUM: out bit);
  end component;
  signal A_I, B_I, CARRY_I, SUM_I : bit;
begin
  DUT: ADDER port map (A_I, B_I, CARRY_I, SUM_I);

  STIMULUS: process
  begin
    A_I <= ´1´; B_I <= ´0´;
    wait for 10 ns;
    A_I <= ´1´; B_I <= ´1´;
    wait for 10 ns;
    -- and so on ...
  end process STIMULUS;
end TEST;

configuration CFG_TB_ADDER of TB_ADDER is
for TEST
end for;
end CFG_TB_ADDER;
```

The example shows the VHDL code for a simple design and its corresponding testbench. The design to be tested is the ADDER which implements a halfadder. The architecture RTL contains one pure combinational process which calculates the results for the SUM and CARRY signals whenever the input signals A or B change.

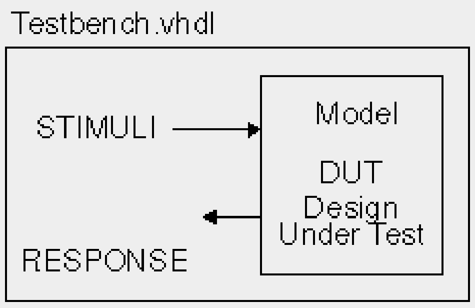
The testbench is shown on the right side. First the empty entity TB_ADDER is defined. There is no need for an interface, so no port list is present. In the architecture TEST of the testbench the component ADDER and the intermediate signals are declared. The intermediate signals (*_I) are connected to the component in the port map of the component instantiation. The signals that are connected to the input ports of the component ADDER get their values assigned in the process STIMULUS. New values are set every 10 ns. The reaction of the DUT can be observed in a waveform display of the simulator.

At the bottom of the VHDL source code, the configuration is listed. Only the architecture TEST for the TB_ADDER entity is specified and the rest is left to the default rules as the name of the component and the entity are identical.

2.2.23 Summary

- VHDL is a very precise language
  - Signal types and directions have to match
  - All objects have to be declared
  - One language for design and validation

- Validation by means of a TESTBENCH
  - Provides stimuli and expected response
  - Top hierarchy level
  - No in-/output ports



VHDL is a very strict language in which hardly a cryptic programming style is possible (as it is the case with the programming language C). Every signal, for example, has to possess a certain data type, it has to be declared at a certain position, and it only accepts assignments from the same data type.

To make a functional test of a VHDL model, a testbench can be written also in VHDL, which delivers the verifiaction environment for the model. In it, stimuli are described as input signals for the model, and furthermore the expected model responses can be checked. The testbench appears as the top hierarchy level, and therefore has neither input- nor output ports.

2.2.24 Questions

	1. An architecture...
○ yes ○ no	**1.1. ... can exist on its own**
○ yes ○ no	**1.2. ... can exist only together with its dedicated design entity.**
○ yes ○ no	**1.3. ... contains a description of the module`s behaviour.**
	2. In VHDL, sequential statements ...
○ yes ○ no	**2.1. ... are defined in the architecture**
○ yes ○ no	**2.2. ... are defined in the process**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.2.25 Questions

	3. The configuration...
○ yes ○ no	**3.1. ... generates the simulateable object.**
○ yes ○ no	**3.2. ... is declared within the architecture.**
○ yes ○ no	**3.3. ... chooses the entity for a certain architecture.**
	4. The component declaration ...
○ yes ○ no	**4.1. ... integrates a certain entity into a preset ‚socket".**
○ yes ○ no	**4.2. ... defines only the socket type.**
○ yes ○ no	**4.3. ... should have the same name as the dedicated entity for default-configuration**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.2.26 Questions

	5. Component instantiation ...
○ yes ○ no	**5.1. ... generates the socket of the component type.**
○ yes ○ no	**5.2. ... wires the socket (=component) to the PCB (=entity).**
	6. Concurrent statements ...
○ yes ○ no	**6.1. ... are declared only in the architecture.**
○ yes ○ no	**6.2. ... are declared only in subprograms.**
○ yes ○ no	**6.3. ... are executed consecutively.**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.3 Data Types

```
entity FULLADDER is
  port(A, B, CARRY_IN: in
 bit;
       SUM, CARRY:     out
bit);
end FULLADDER;

architecture MIX of
FULLADDER is
  component HALFADDER
    port(A, B:              in   bit;
          SUM, CARRY: out
bit);

  signal W_SUM,
W_CARRY1, W_CARRY2: bit;

begin
  HA1: HALFADDER
    port map(A, B, W_SUM,
W_CARRY1);
  HA2: HALFADDER
    port map(CARRY_IN,
W_SUM, SUM, W_CARRY2);

  CARRY <= W_CARRY1 or
W_CARRY2;

end MIX;
```

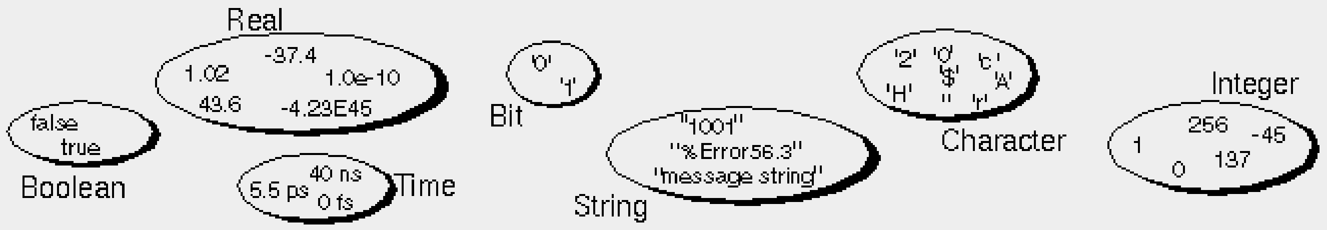
- **Every signal has a type**
- **Type specifies possible values**
- **Type has to be definined at signal declaration...**
- **...either in**
  - **entity: port declaration, or in**
  - **architecture: signal declaration**
- **Types have to match**

In VHDL, signals must have a data type associated with them that limits the number of possible values. This type has to be fixed when the signal is declared, either as entity port or an internal architecture signal, and can not be changed during runtime. Whenever signal values are updated, the data types on both sides of the assignment operator '<=' have to match.

2.3.1 Standard Data Types

```
package STANDARD is
   type BOOLEAN is (FALSE,TRUE);
   type BIT is (`0`,`1`);
   type CHARACTER is (-- ascii set);
   type INTEGER is range
                              -- implementation_defined
   type REAL is range
                             --  implementation_defined
   -- BIT_VECTOR, STRING, TIME
end STANDARD;
```

- **Every type has a number of possible values**
- **Standard types are defined by the language**
- **User can define his own types**



'93 **New types added to the ``standard`` package,**
e.g. umlauts

In VHDL, signals must have a data type associated with them that limits the number of possible values. This type has to be fixed when the signal is declared, either as entity port or an internal architecture signal, and can not be changed during runtime. Whenever signal values are updated, the data types on both sides of the assignment operator ' <= ' have to match.

A number of data types are already defined in the standard package which is always implicitly referenced. ' **boolean** ' is usually used to control the flow of the VHDL execution while ' **bit** ' uses level values (' **0** ', ' **1** ') instead of truth values (' **false** ', ' **true** ') and is therefore better suited to model wires. Number values can be communicated via signals of type ' **integer** ' or ' **real** '. The actual range and accuracy depends on the platform implementation and only lower bounds are defined, e.g. integers are guaranteed to be at least 32 bits wide. Floating point operations can not be synthesized automatically, yet, i.e. the use of ' **real** ' data types is restricted to testbench applications. The same applies to ' **character** ' and ' **time** '.

' **time** ' is a special data type as it consists out of a numerical value and a physical unit. It is used to delay the execution of statements for a certain amount of time, e.g. in testbenches or to model gate and propagation delays. Signals of data type ' **time** ' can be multiplied or divided by ' **integer** ' and ' **real** ' values. The result of these operations remains of data type ' **time** '. The internal resolution of VHDL simulators is set to femto seconds (fs).

2.3.2 Datatype 'time'

```
architecture EXAMPLE of TIME_TYPE is
  signal CLK : bit := `0';
  constant PERIOD : time := 50 ns;

begin
  process
  begin
    wait for 50 ns;
    . . .
    wait for  PERIOD ;
    . . .
    wait for 5 *  PERIOD ;
    . . .
    wait for  PERIOD  * 5.5;
  end process;

. . .

  -- concurrent signal assignment
  CLK <= not CLK after 0.025 us;
  -- or with constant time
  -- CLK <= not CLK after PERIOD/2;
end EXAMPLE;
```

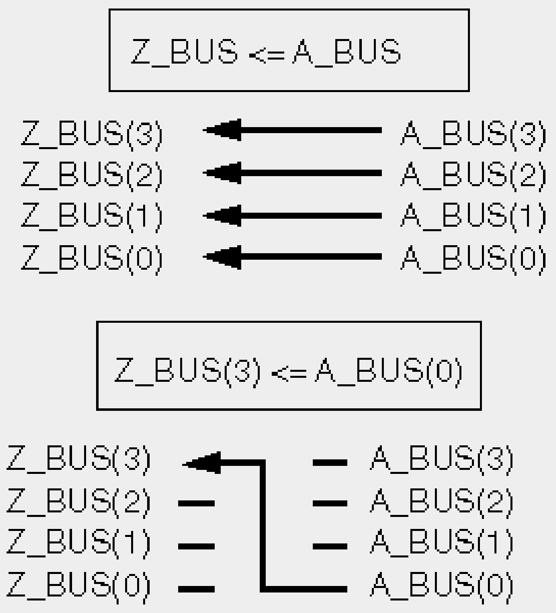
- **Usage**
  - **testbenches**
  - **gate delays**
- **Multiplication/division**
  - **mutliplied/divided by integer/real**
  - **returns TIME type**
  - **internally in smallest unit (fs)**
- **Available time units**
  fs, ps, ns, us, ms, sec, min, hr

'time' is a special data type as it consists out of a numerical value and a physical unit. It is used to delay the execution of statements for a certain amount of time, e.g. in testbenches or to model gate and propagation delays. Signals of data type 'time' can be multiplied or divided by 'integer' and 'real' values. The result of these operations remains of data type 'time'. The internal resolution of VHDL simulators is set to femto-seconds (fs).

2.3.3 Definition of Arrays

```
signal A_BUS, Z_BUS : bit_vector (3 downto 0);
```

- **Collection of signals of the same type**
- **Predefined arrays**
  - **bit_vector (array of bit)**
  - **string (array of character)**
- **Unconstrained arrays: definition of actual size during signal/port declaration**



Arrays are useful to group signals of the same type and meaning. Two unconstrained array data types, i.e. whose range is not limited, are pre-defined in VHDL: ' **bit_vector** ' and ' **string** ' are arrays of ' **bit** ' and ' **character** ' values, respectively. Please note that the array boundaries have to be fixed during signal declarations, e.g. 'bit_vector(3 downto 0)'. Only constrained arrays may be used as entity ports or architecture signals.

Integer signals will be mapped to a number of wires during synthesis. These wires could be modeled via bit vectors as well, yet ' **bit_vector** ' signals do not have a numerical interpretation associated with them. Therefore the synthesis result for the two example architectures would be the same. The process models a simple multiplexer which selects the input A as source for its output Z when the select signal SEL is '1' and the input B otherwise. Please note that the multiplexer process is exactly the same for both data types!

Special care is necessary when signal assignments with arrays are carried out. Although the data type and the width of the signals have to match, this is not true for the order of the array elements. The values are assigned according to their position within the array, not according to their index. Therefore it is highly recommended to use only one direction (usually ' **downto** ' in hardware applications) throughout your designs.

2.3.4 'integer' and 'bit' Types

```
architecture EXAMPLE_1 of DATATYPES is

   signal SEL :      bit ;
   signal A, B, Z :  integer  range 0 to 3;

begin
   A <= 2;
   B <= 3;

   process(SEL,A,B)
   begin
      if SEL = '1' then
         Z <= A;
      else
         Z <= B;
      end if;
   end process;
end EXAMPLE_1;
```

OR:

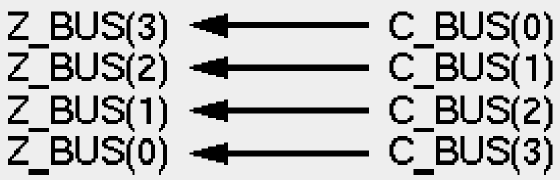
```
architecture EXAMPLE_2 of DATATYPES is

   signal SEL :      bit ;
   signal A, B, Z:    bit_vector (1 downto 0);

begin
   A <= "10";
   B <= "11";

   process(SEL,A,B)
   begin
      if SEL = '1' then
         Z <= A;
      else
         Z <= B;
      end if;
   end process;
end EXAMPLE_2;
```

- **Example for using ' bit' and ' integer'**

Integer signals will be mapped to a number of wires during synthesis. These wires could be modelled via bit vectors as well, yet 'bit_vector' signals do not have a numerical interpretation associated with them. Therefore the synthesis result for the two example architectures would be the same. The process models a simple multiplexer which selects the input A as source for its output Z when the select signal SEL is '1' and the input B otherwise. Please note that the multiplexer process is exactly the same for both data types!

2.3.5 Assignments with Array Types

```
architecture EXAMPLE of ARRAYS is
  signal Z_BUS : bit_vector (3 downto 0);
  signal C_BUS : bit_vector (0 to 3);
begin
  Z_BUS <= C_BUS;
end EXAMPLE;
```



Elements are assigned according to their position, not their number

The direction of arrays should always be defined the same way

Special care is necessary when signal assignments with arrays are carried out. Although the data type and the width of the signals have to match, this is not true for the order of the array elements. The values are assigned according to their position within the array, not according to their index. Therefore it is highly recommended to use only one direction (usually 'downto' in hardware applications) throughout your designs.

2.3.6 Types of Assignment for 'bit' Data Types

---

architecture EXAMPLE of ASSIGNMENT is

```
   signal Z_BUS :     bit_vector (3 downto 0);
   signal BIG_BUS : bit_vector (15 downto 0);

begin

   -- legal assignments:
   Z_BUS(3) <= `1`;
   Z_BUS      <= ``1100``;

   Z_BUS      <=  b`` 1100 ``;
   Z_BUS      <=  x`` c ``;
   Z_BUS      <=  X`` C ``;
   BIG_BUS  <=  B` `0000 _ 0001_0010_0011 ``;

end EXAMPLE;
```

- **Single bit values are enclosed in '.'**
- **Vector values are enclosed in "..."**
  - **optional base specification (default: binary)**
  - **values may be separated by underscores to improve readability**

Different specification of single bits and bit vectors

Valid assignments for the datatype 'bit' are also valid
for all character arrays, e.g.

'std_(u)logic_vector'

The specification of signal values is different for the base types 'character' and 'bit' and their corresponding array types 'string' and 'bit_vector'. Single values are always enclosed in single quotation marks ('), while double quotation marks (") are used to specify array values.

As bit vectors are often used to represent numerical values, VHDL offers several possibilities to increase the readability of bit vector assignments. First, a base for the following number may be specified. Per default binary data consisting of '0's and '1's is assumed. Please note that the values have to enclosed in double quotation marks even though only a single symbol might be necessary when another base is used! Additionally, underscores (_) may be inserted at will to split long chains of numbers into smaller groups in order to improve readability.

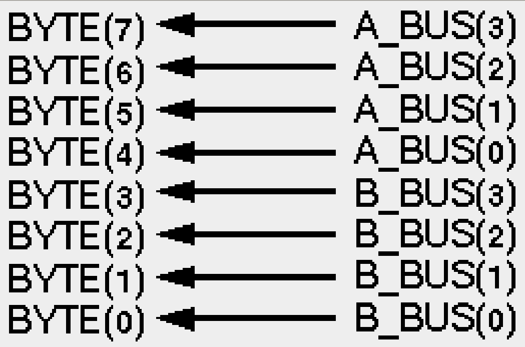
Since VHDL'93, the same rules apply to the enhanced bit vector types ' **std_(u)logic_vector** ', which will be discussed later on, as well.

2.3.7 Concatenation

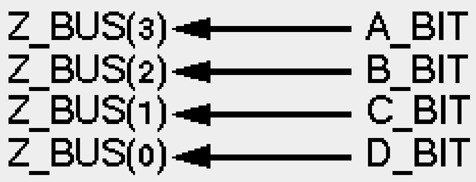
- **Concatenation operator: &**

- **Resulting signal assignment:**

```
architecture EXAMPLE_1 of CONCATENATION is
   signal BYTE             : bit_vector (7 downto 0);
   signal A_BUS, B_BUS : bit_vector (3 downto 0);
begin

   BYTE   <= A_BUS & B_BUS;

end EXAMPLE;
```



```
architecture EXAMPLE_2 of CONCATENATION is
   signal Z_BUS : bit_vector (3 downto 0);
   signal A_BIT, B_BIT, C_BIT, D_BIT : bit;
begin

   Z_BUS <= A_BIT & B_BIT & C_BIT & D_BIT;

end EXAMPLE;
```



The concatenation operator '&' is allowed on the right side of the signal assignment operator '<=', only

As signal assignments require matching data types on both sides of the operator it is sometimes necessary to assemble an arrays in the VHDL code. The concatenation operator ' **&** ' groups together the elements on its sides which have to be of the same data type, only. Again, the array indices are ignored and only the position of the elements within the arrays is used. The concatenation operator may be used on the right side of signal assignments, only!

Another way of assigning signals which does not suffer from this limitation is via the aggregate construct. Here, the signals that are to build the final array are enclosed in a '(' ')' pair and separated by ','. Instead of a simple concatenation, it is also possible to address the array elements explicitly by their corresponding index, as shown in the last signal assignment statement of the aggregate example. The keyword ' **others** ' may be used to select those indices that have not been addressed, yet.

2.3.8 Aggregates

```
architecture EXAMPLE of
AGGREGATES is
  signal BYTE :  bit_vector (7 downto 0);
  signal Z_BUS : bit_vector (3 downto 0);
  signal A_BIT, B_BIT, C_BIT, D_BIT :
bit;
begin
  Z_BUS <= ( A_BIT, B_BIT, C_BIT,
D_BIT ) ;
   ( A_BIT, B_BIT, C_BIT, D_BIT ) <=
bit_vector'(``1011``);
   ( A_BIT, B_BIT, C_BIT, D_BIT ) <=
BYTE(3 downto 0);

  BYTE <= ( 7 => `1`, 5 downto 1 => `1`,
6 => B_BIT, others => `0`) ;
end EXAMPLE;
```

- **Aggregates bundle signals together**
- **May be used on both sides of an assignment**
- **keyword 'other' selects all remaining elements**

**Some aggregate constructs may not be supported
by your synthesis tool**

**Assignment of `0` to all bits of a vector regardless of the width:
VECTOR <= (others => '0');**

Another way of assigning signals which does not suffer from this limitation is via the aggregate construct. Here, the signals that are to build the final array are enclosed in a '(' ')' pair and separated by ','. Instead of a simple concatenation, it is also possible to address the array elements explicitly by their corresponding index, as shown in the last signal assignment statement of the aggregate example. The keyword 'others' may be used to select those indices that have not been addressed, yet.

2.3.9 Slices of Arrays

```
architecture EXAMPLE of SLICES is
  signal BYTE : bit_vector (7 downto 0);
  signal A_BUS, Z_BUS : bit_vector (3 downto 0);
  signal A_BIT : bit;
begin
  BYTE (5 downto 2) <= A_BUS;
  BYTE (5 downto 0) <= A_BUS;
          -- wrong

  Z_BUS (1 downto 0) <= `0` & A_BIT;
  Z_BUS              <= BYTE (6 downto 3);
  Z_BUS (0 to 1)     <= `0` & B_BIT;
    -- wrong

  A_BIT   <= A_BUS (0);
end EXAMPLE;
```

- **Slices select elements of arrays**

The direction of the "slice" and of the array must match

The inverse operation of concatenation and aggregation is the selection of slices of arrays, i.e. only a part of an array is to be used. The range of the desired array slice is specified in brackets and must match the range declaration of the signal! Of course, it is possible to select only single array elements.

2.3.10 Questions

	7. Array assignments are made ...
○ yes ○ no	**7.1. ...according to the position.**
○ yes ○ no	**7.2. ...according to the index.**
○ yes ○ no	**7.3. ...at option.**
	8. A BIT_VECTOR assignment is, e.g.,...
○ yes ○ no	**8.1. ... <= "000111XXUUU00";**
○ yes ○ no	**8.2. ...<= "0000111000111";**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.3.11 Questions

9. Which assignments are correct with the following definitions?

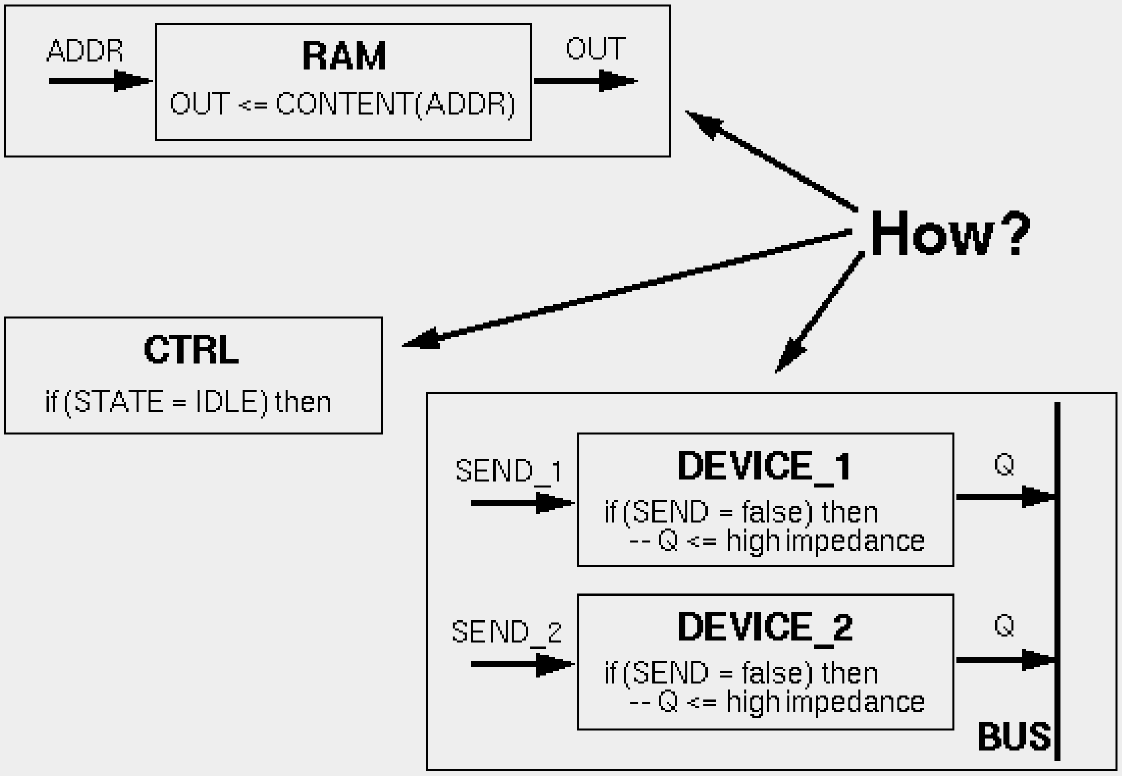
```
signal A_BUS, B_BUS, Z_BUS :        bit_vector (3 downto 0);
signal A_BIT, B_BIT, C_BIT, D_BIT :  bit;
signal BYTE :                       bit_vector (7 downto 0);
```

○ yes ○ no	**9.1. BYTE <= (others => `1`);**
○ yes ○ no	**9.2. BYTE(7 downto4) <= A_BIT & B_BIT & A_BIT & B_BIT;**
○ yes ○ no	**9.3. Z_BUS <= A_BIT & B_BIT;**
○ yes ○ no	**9.4. BYTE (3 downto 0) <= (`1`, B_BIT, `0`, D_BIT);**
○ yes ○ no	**9.5. A_BUS (0 to 1) <= (others => `0`);**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.4 Extended Data Types

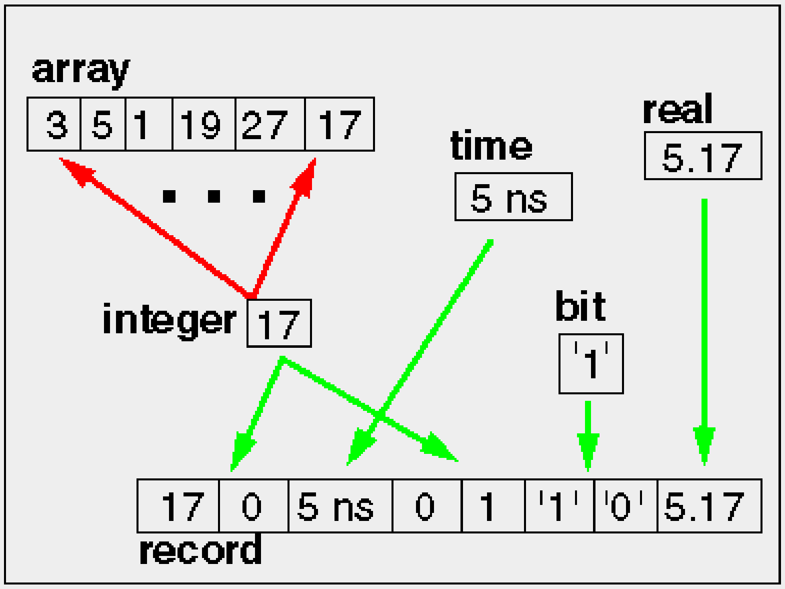


The standard data types are of limited use for practical modeling tasks. Of course, it would be possible to implement the behaviour of RAM or ROM cells via case constructs, yet the resulting VHDL would look pretty awkward. Coding would be much easier, if arrays of other data types were available.

Another case for additional types is based on readability issues. VHDL code should be usable as documentation as well. Symbolic, descriptive names for signal values usually lead to self-documenting code. This does not mean that comments are no longer necessary, though!

While relatively easy workarounds exist for these two problems, the modeling of bus systems is a nightmare when only predefined data types are to be used. The main characteristics of a data bus is the existence of multiple bus participants, i.e. the same signal wires are used by multiple modules. If such a device does not transmit anything, its output driver must be set to a value that does not destroy other data values.

2.4.1 Type Classification



- **Scalar types:**
  - **contain exactly one value**
  - **integer, real, bit, enumerated, physical**
- **Composite types:**
  - **may contain more than one value**
  - ***array* : values of one type, only**
  - ***record* : values of different types**
- **Other types:**
  - **"file"**

- ## "access"

In VHDL, data types are classified into three categories: scalar types, composite types and other types. Scalar types have exactly one value. Examples are the predefined integer, real and bit values. Physical data types (e.g. time) fall also into this category, as well as user defined data types whose range of values is declared via an enumeration. Scalar types do not have distinguishable elements. Composite types, on the other side, usually hold a collection of values. This collection is called an array if all values are of the same type; different data types may be present in records.

FILE and ACCESS types are data types that provide access to other objects. The FILE type is used to read or store data in a file; ACCESS types are comparable with pointers.

2.4.2 Enumeration Types

```
architecture EXAMPLE of ENUMERATION is

   type T_STATE is (RESET, START, EXECUTE, FINISH);

   signal CURRENT_STATE, NEXT_STATE : T_STATE ;
   signal TWO_BIT_VEC : bit_vector(1 downto 0);

begin

   -- valid signal assignments
   NEXT_STATE       <= CURRENT_STATE;
   CURRENT_STATE <= RESET;

   -- invalid signal assignments
   CURRENT_STATE <= "00";
   CURRENT_STATE <= TWO_BIT_VEC;

end EXAMPLE;
```

- **Designer may define their own types**
  - **enhanced readability (commonly used to describe the states of a state maschine)**
  - **limitted legal values**

Synthesis tools map enumerations onto a suitable bit pattern

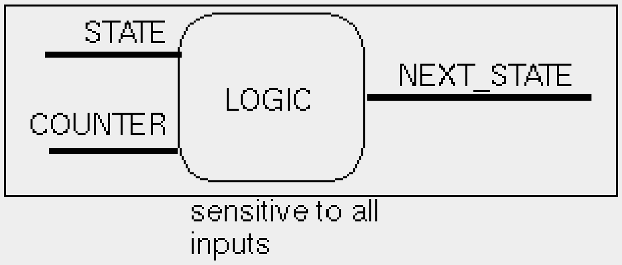
It is possible to define new scalar data types in VHDL. They are called enumeration types because all possible object (constant, signal, variable) values have to be specified in a list at type declaration. User defined data types are frequently used to enhance readability when dealing with so called state machines, i.e. modules that behave differently, depending on the state of internal storage elements. Instead of fixed bit patterns, the symbolic names of the data type values are used which will be mapped to a bit level representation automatically during synthesis.

In the example, a new data type with values denoting four different states (RESET, START, EXECUTE, FINISH) is defined. This type is used for the two signals CURRENT_STATE and NEXT_STATE and the four declared type values can be assigned directly. Of course, after synthesis, there will be two bits for each signal, but a direct assignment of a two bit vector is not allowed. In a signal assignment only those values may be used which are enumerated in the type declaration.

Some synthesis tools allow the designer to map the different values onto specific bit patterns.

2.4.3 Enumeration Types - Example

```
architecture RTL of TRAFFIC_LIGHT is

  type  T_STATE is
       ( INIT,RED,REDYELLOW,GREEN,YELLOW );
  signal STATE, NEXT_STATE : T_STATE;

  signal COUNTER: integer;
  constant END_RED      : integer := 10000;
  constant END_GREEN : integer := 20000;

begin

  LOGIC : process (STATE, COUNTER)
  begin
    NEXT_STATE <= STATE;
    case STATE is
      when  RED                    =>
        if COUNTER = END_RED then
          NEXT_STATE <=  REDYELLOW ;
        end if;
      when  REDYELLOW     => -- statements
      when  GREEN              => -- statements
      when  YELLOW          => -- statements
      when  INIT                 => -- statements
    end case;
  end process LOGIC;
end RTL;
```



The example demonstrates the impact of user defined data types on code readability. The LOGIC block shall implement the behaviour of a traffic light controller. The data type T_STATE is defined in the declarative part of the architecture and is used for the signals STATE and NEXT_STATE. The functional behaviour of the algorithm should be pretty obvious as symbolic names are used, only. For this purpose, additional constants were defined for the specific counter values that are checked within the code.

2.4.4 BIT Type Issues

type BIT is ('0', '1')	• **Values \`0\` and \`1\`, only** ○ **default value \`0\`** • **Additional requirements for simulation and synthesis** ○ **uninitialized** ○ **high impedance** ○ **undefined** ○ **\`don\`t care\`** ○ **different driver strengths**

The ' **bit** ' type has only the two values '0' and '1'. While this is enough to model simple logic where the wires are driven either high or low level, further options are desirable, especially for simulation purposes. VHDL objects are initialised with their default value which is the leftmost value from the type declaration. Therefore, every variable/signal of type ' **bit** ' would be set to '0' at the beginning of each simulation. This makes it impossible to verify the proper reset behaviour of a design, if the reset value of a register is also '0'.

Additional legal wire conditions are necessary, if different driver strengths or high impedance outputs of real hardware drivers are to be modeled. It depends on the synthesis tool whether these additional logic values can be mapped to the corresponding hardware cells. In order to avoid hardware overhead one might think of designating bit positions that may safely be ignored during synthesis ("don't care"). For simulation, the opposite is desirable, i.e. a value which indicates that something went wrong and needs to be inspected ("undefined").

In the beginning, several incompatible multi-valued logic systems were defined by the different software companies. In order to solve the resulting problems, a standardized 9-valued logic system was defined and accepted by the IEEE in 1992.

2.4.5 Multi-valued Types

- **Multi-valued logic-systems are declared via new datatypes**
  - **uninitialized**
  - **unknown**
  - **high impedance**
  - **...**
- **Manufacturer dependent implementation**
  - **mvl4, mvl7, mvl9, ..., mvl46**
- **No common standard before 1992**
- **IEEE-standard**
  - **9-valued logic-system defined and accepted by the IEEE**
  - **standard IEEE 1164 (STD_LOGIC_1164)**

Additional legal wire conditions are necessary, if different driver strengths or high impedance outputs of real hardware drivers are to be modelled. It depends on the synthesis tool whether these additional logic values can be mapped to the corresponding hardware cells. In order to avoid hardware overhead one might think of designating bit positions that may safely be ignored during synthesis ("don't care"). For simulation, the opposite is desirable, i.e. a value which indicates that something went wrong and needs to be inspected ("undefined"). In the beginning, several incompatible multi-valued logic systems were defined by the different software companies. In order to solve the resulting problems, a standardized 9-valued logic system was defined and accepted by the IEEE in 1992.

© LRS - UNI Erlangen-Nuremberg

2.4.6 IEEE Standard Logic Type

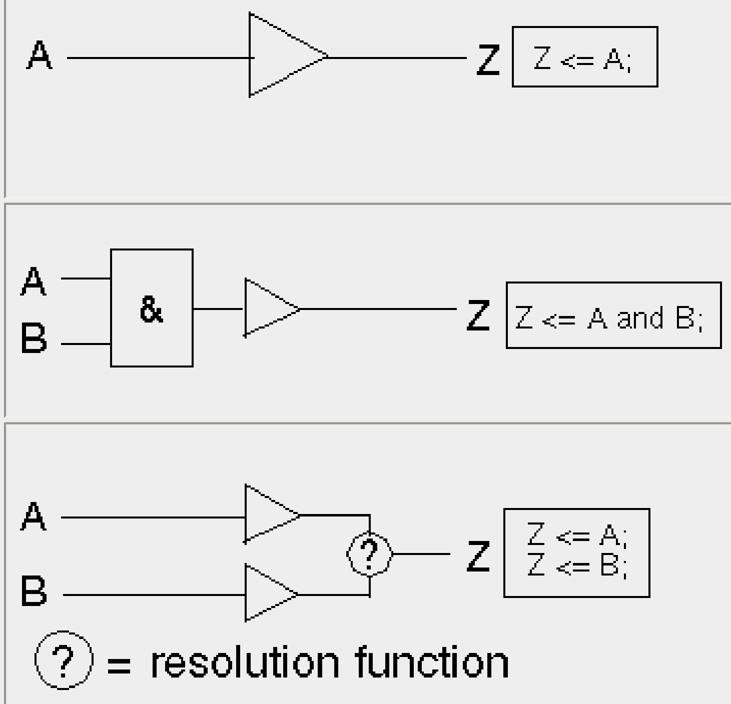
type STD_ULOGIC is (`**U**`, -- uninitialized `**X**`, -- strong 0 or 1 (= unknown) `**0**`, -- strong 0 `**1**`, -- strong 1 `**Z**`, -- high impedance `**W**`, -- weak 0 or 1 (= unknown) `**L**`, -- weak 0 `**H**`, -- weak 1 `**-**`, -- don`t care);	• **9 different signal states** • **Superior simulation results** • **Bus modeling** • **"ASCII-characters"**

- **Defined in package 'IEEE.std_logic_1164'**
- **Similar data type 'std_logic' with the same values**
- **Array types available: 'std_(u)logic_vector', similar to 'bit_vector'**
- **All 'bit' operators available**

The IEEE standard should be used in VHDL designs

The new data type is called 'std_ulogic' and is defined in the package 'std_logic_1164' which is placed in the library IEEE (i.e. it is included by the following statement: 'use IEEE.std_logic_1164.all'). The new type is implemented as enumerated type by extending the existing '0' and '1' symbols with additional ASCII characters. The most important ones are probably 'u' (uninitialized) and 'x' (unknown value). The 'u' symbol is the leftmost symbol of the declaration, i.e. it will be used as initial value during simulation.

2.4.7 Resolved and Unresolved Types



- **Signal assignments are represented by drivers**
- **Unresolved data type: only one driver**
- **Resolved data type: possibly several drivers per signal**

```
architecture EXAMPLE of ASSIGNMENT is
  signal A, B, Z: bit;
  signal INT:      integer;
begin
  Z <= A;
  Z <= B;
  Z <= INT;      -- wrong
end EXAMPLE;
```

- **Conditions for valid assignments**
  - **types have to match**
  - **resolved type, if more than 1 concurrent assignment**

Besides the type definition of 'std_ulogic' the 'std_logic_1164' package contains also the definition of a similar type called 'std_logic' which has the same value set as 'std_ulogic'. Like 'bit_vector', array data types 'std_(u)logic_vector' are also available. Additionally, all operators that are defined for the standard type 'bit' are overloaded to handle the new replacement type.
As mentioned before, the 'bit' data type can not be used to model bus architectures. This is because all signal assignments are represented by drivers in VHDL. If more than one driver try to force the value of a signal a resolution will be needed to solve the conflict. Consequently, the existence of a resolution function is necessary for legal signal assignments. Please note, resolution conflicts are detected at run-time and not during compilation!

2.4.8 Std_Logic_1164 Package

```
PACKAGE std_logic_1164 IS

  -------------------------------------------------
  -- logic state system  (unresolved)
  -------------------------------------------------

   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);

  -------------------------------------------------
  -- unconstrained array of std_ulogic for use
  -- with the resolution function
  -------------------------------------------------
   TYPE   std_ulogic_vector  IS
    ARRAY ( NATURAL RANGE <> ) OF std_ulogic;

  -------------------------------------------------
  -- resolution function
  -------------------------------------------------
   FUNCTION resolved ( s : std_ulogic_vector )
                                RETURN std_ulogic;
  -------------------------------------------------
  -- *** industry standard logic type ***
  -------------------------------------------------
   SUBTYPE std_logic IS resolved std_ulogic;
  -------------------------------------------------
  -- unconstrained array of std_logic
  -- for use in declaring signal arrays
  -------------------------------------------------
   TYPE std_logic_vector  IS
     ARRAY ( NATURAL RANGE <>) OF std_logic;
END std_logic_1164;
```

It is recommended to use the multi-valued logic system from the IEEE instead of the standard 'bit' data type. The new type is called 'std_ulogic' and is defined in the package 'std_logic_1164' which is placed in the library IEEE (i.e. it is included by the following statement: 'use IEEE.std_logic_1164.all'.

It can be seen from the type definition that the most desirable signal values are defined. The '0' and '1' symbols that are used as 'bit' values were extended by additional ASCII characters, the most important ones being probably 'u' (uninitialized) and 'x' (unknown value). The 'u' symbol is the leftmost symbol of the delcaration, i.e. it will be used as initial value during simulation.

Besides the type definition of 'std_ulogic' the 'std_logic_1164' package contains also the definition of a similar type called 'std_logic' which has the same value set as 'std_ulogic'. Like 'bit_vector', array data types 'std_(u)logic_vector' are also available. Additionally, all operators that are defined for the standard type 'bit' are overloaded to handle the new replacement type.

As mentioned before, the 'bit' data type can not be used to model bus architectures. This is because all signal assignments are represented by drivers in VHDL. If more than one driver try to force the value of a signal a resolution will be needed to solve the conflict. Consequently, the existence of a resolution function is necessary for legal signal assignments. Please note, resolution conflicts are detected at run-time and not during compilation!

Predefined VHDL data types do not possess a resolution function because the effects of multiple signal drivers depend on the actual hardware realization. The 'std_ulogic' data type ("u" = "unresolved") is the basis for the resolved data type 'std_logic'. The 'resolved' function that is also defined in the 'std_logic_1164' package gets called whenever signal assignments involving 'std_logic' based data types are carried out.

2.4.9 Resolution Function

```
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
   CONSTANT  resolution_table  : std_logic_table := (
   --     ---------------------------------------------------------------
   --      U      X      0      1      Z      W      L      H      -
   --     ---------------------------------------------------------------
          ( 'U',   'U',   'U',   'U',   'U',   'U',   'U',   'U',   'U' ),    -- U
          ( 'U',   'X',   'X',   'X',   'X',   'X',   'X',   'X',   'X' ),    -- X
          ( 'U',   'X',   '0',   'X',   '0',   '0',   '0',   '0',   'X' ),    -- 0
          ( 'U',   'X',   'X',   '1',   '1',   '1',   '1',   '1',   'X' ),    -- 1
          ( 'U',   'X',   '0',   '1',   'Z',   'W',   'L',   'H',   'X' ),    -- Z
          ( 'U',   'X',   '0',   '1',   'W',   'W',   'W',   'W',   'X' ),    --W
          ( 'U',   'X',   '0',   '1',   'L',   'W',   'L',   'W',   'X' ),    -- L
          ( 'U',   'X',   '0',   '1',   'H',   'W',   'W',   'H',   'X' ),    -- H
          ( 'U',   'X',   'X',   'X',   'X',   'X',   'X',   'X',   'X' )     --  -  );
   VARIABLE result : std_ulogic := 'Z';  -- weakest state default
BEGIN
   IF    (s'LENGTH = 1) THEN
      RETURN s(s'LOW);
   ELSE
      FOR i IN s'RANGE LOOP
        result := resolution_table(result, s(i));
     END LOOP;
   END IF;
   RETURN result;
END resolved;
```

- **All driving values are collected in a vector**
- **The result is calculated element by element according to the table**
- **Resolution function is called whenever signal assignments involving resolved types are carried out**

The conflict resolution process itself, i.e. the decision about the final signal value in case of multiple drivers, is based upon a resolution table. All driving values are collected in an array and handed to the resolution function, even if only a single driver is present! The result is calculated element by element: the current result selects the row of the resolution table and the value of the next signal driver selects the column of the resulting signal value.

2.4.10 STD_LOGIC vs STD_ULOGIC

STD_ULOGIC **STD_ULOGIC_VECTOR**	**Benefit** • **Error messages in case of multiple concurrent signal assignments**
STD_LOGIC **STD_LOGIC_VECTOR**	**Benefit** • **Common industry standard** ❍ **gate level netlists** ❍ **mathematical functions** • **Required for tri-state busses**

STD_LOGIC(_VECTOR) is recommended for RT level designs

Use port mode 'buffer' to avoid multiple signal assignments

By far most of the connections in a design, either as abstract signals or later on as real wires, are from one point to another, i.e. multiple signal drivers would indicate an error. This kind of error will be easily detected if just unresolved data types are used.

Yet, the resolved counterpart 'std_logic' has been established as de facto industry standard despite of some shortcomings. As all kind of hardware structures, including bus systems, can be modeled with this data type it is used by synthesis tools for the resulting gate level description of a design. Even a workaround exists, so that multiple signal assignments can still be detected: the port mode 'buffer' allows for a single signal driver, only. According to the VHDL language definition, however, the resolution function has to be called when resolved signals are assigned. The impact on simulation performance depends on the compiler/simulator implementation.

The last, but certainly not the least advantage of 'std_logic' based designs is the existence of standard packages defining arithmetical operations on vectors. This eliminates the need for complex type conversion functions and thus enhances the readability of the code.

The ' **numeric_std** ' package is located in the library IEEE and provides two numerical interpretations of the bit vector, either as signed or as unsigned integer value. Overloaded operators to mix vectors and integers in expressions are also available. Please note, that it is impossible to overload the signal assignment operator, i.e. a function must be called in this case. Conversion functions ' **to_integer** ' and ' **to_(un)signed** ' are also defined in the package.

The equivalent of ' **numeric_std** ' for ' **bit** ' based operations is called ' **numeric_bit** '. The use of ' **bit** ' based signals is not recommended, however, due to the disadvantages of a 2-valued logic system.

2.4.11 The NUMERIC_STD Package

- Provides numerical interpretation for 'std_logic' based vectors
  - signed: 2-complement (sign+absolute value)
  - unsigned: binary representation of positive integers
- Overloaded mathematical operators
  - allow mixture of vector and integer values (vector <= vector + 1)
- Overloaded relational operators
  - avoid problems when dealing with different vector lengths
  - comparison of vector with integer values
- NUMERIC_BIT package with 'bit' as basis data type

The use of 'bit' and 'bit_vector' is not recommended

The 'numeric_std' package is located in the library IEEE and provides two numerical interpretations of the bit vector, either as signed or as unsigned integer value. Overloaded operators to mix vectors and integers in expressions are also available.

Please note, that it is impossible to overload the signal assignment operator, i.e. a function must be called in this case. Conversion functions 'to_integer' and 'to_(un)signed' are also defined in the package.

The equivalent of 'numeric_std' for 'bit' based operations is called 'numeric_bit'. The use of 'bit' based signals is not recommended, however, due to the disadvantages of a 2-valued logic system.

2.4.12 Arrays

```
type STD_ULOGIC_VECTOR is
    array (natural range <>) of STD_ULOGIC;
type MY_BYTE is
    array(7 downto 0) of STD_ULOGIC;

signal BYTE_BUS : STD_ULOGIC_VECTOR(7 downto 0);
signal TYPE_BUS : MY_BYTE;
```

- **Definition of an array type**
  - **constrained or unconstrained size**
- **Declaration of a signal of that type**
  - **range specification necessary**

```
architecture EXAMPLE of ARRAY is
   type CLOCK_DIGITS is
      (HOUR10,HOUR1,MINUTES10,MINUTES1);
   type  T_TIME is
      array(CLOCK_DIGITS) of integer  range 0 to 9;

   signal ALARM_TIME : T_TIME := (0,7,3,0);

begin
   ALARM_TIME(HOUR1)                    <= 0;
   ALARM_TIME(HOUR10 to MINUTES10) <= (0,7,0);
end EXAMPLE;
```

- **The index set can be of any type**

Only integer index sets are supported by all synthesis tools

Arrays are a collection of a number of values of a single data type and are represented as a new data type in VHDL. It is possible to leave the range of array indices open at the time of definition. These so called unconstrained arrays can not be used as signals, however, i.e. the index range has to be specified in the signal declaration then. The advantage of unconstrained arrays is the possibility to concatenate objects of different lengths, for example, because they are still of the same data type. This would not be allowed if each array length was declared as separate data type.

VHDL does not put any restrictions on the index set of arrays, as long it is a descrete range of values. It is even legal to use enumeration types, as shown in the code example, although this version is not generally synthesizable.

2.4.13 Multidimensional Arrays

```
architecture EXAMPLE of
ARRAY is

   type INTEGER_VECTOR is
     array (1 to 8) of integer;

   -- 1 --
   type MATRIX_A is
     array(1 to 3) of
INTEGER_VECTOR ;
  -- 2 --
  type MATRIX_B is
     array(1 to 4, 1 to 8) of integer ;

   signal MATRIX_3x8 : MATRIX_A;
   signal MATRIX_4x8 : MATIRX_B;

begin

   MATRIX_3x8(3)(5) <= 10; --array of array

   MATRIX_4x8(4, 5) <= 17; -- 2 dim array

end EXAMPLE;
```

- **2 possibilities**
  - **array of array**
  - **multidimensional array**
- **Different referencing**
- **Barely supported by synthesis tools**

Multidimensional arrays can simply be obtained by defining a new data type as array of another array data type (1). When accessing its array elements, the selections are processed from left to right, i.e. the leftmost pair of brackets selects the index range for the "outermost" array. Thus ' **MATRIX_3x8(2)** ' selects the second ' **INTEGER_VECTOR** ' of ' **MATRIX_A** '. The range enclosed in the next pair applies to the array that is returned by the previous slice selection, i.e. ' **MATRIX_3x8(2)(4)** ' returns the fourth integer value of this ' **INTEGER_VECTOR** '.

Multiple dimensions can also be specified diretly within a new array definition (2). The ranges of the different dimensions are separated by ',' symbols. If a whole row or column is to be selected, the range has to be provided in the slice selection. Multidimensional arrays are generally synthesizable up to dimension 2, only.

The most convenient way to assign values to multiple array elements is via the aggregate mechanism. Aggregates can also be nested for this purpose.

2.4.14 Aggregates and Multidimensional Arrays

```
architecture EXAMPLE of AGGREGATE is

    type INTEGER_VECTOR is
        array (1 to 8) of integer;
    type MATRIX_A is
        array(1 to 3) of INTEGER_VECTOR;

    type MATRIX_B is
        array(1 to 4, 1 to 8) of integer;

   signal MATRIX3x8 : MATRIX_A;
   signal MATRIX4x8 : MATIRX_B;
   signal VEC0, VEC1,
           VEC2, VEC3  :  INTEGER_VECTOR;
begin

   MATRIX3x8 <= (VEC0, VEC1, VEC2);
   MATRIX4x8 <= (VEC0, VEC1, VEC2, VEC3);

   MATRIX3x8 <= (others => VEC3);
   MATRIX4x8 <= (others => VEC3);

   MATRIX3x8 <= (others => (others => 5));
   MATRIX4x8 <= (others => (others => 5));

end EXAMPLE;
```

- **Aggregates may be nested**
- **Aggregates can be used to make assignments to all elements of a multidimensional array**

With an aggregate one can assign to all elements of an array a specific value in a clear fashion.

2.4.15 Records

```
architecture EXAMPLE of AGGREGATE is
   type MONTH_NAME is (JAN, FEB, MAR, APR,
                              MAY, JUN, JUL, AUG,
                              SEP, OCT, NOV, DEC);
   type DATE is
        record
           DAY:       integer range 1 to 31;
           MONTH: MONTH_NAME;
           YEAR:     integer range 0 to 4000;
       end record;

   type PERSON is
       record
         NAME:        string (0 to 8);
         BIRTHDAY: DATE;
       end record;

   signal TODAY:      DATE;
   signal STUDENT_1: PERSON;
   signal STUDENT_2: PERSON;
begin
  TODAY      <= (26, JUL, 1988);
  STUDENT_1 <= ("Franziska", TODAY);

  STUDENT_2 <= STUDENT_1;
  STUDENT_2.BIRTHDAY.YEAR <= 1974;
end EXAMPLE;
```

- **Elements of different type**
- **Possible assignments**
  - **record <= record**
  - **record <= aggregate**
  - **record.element <= value**

In contrast to array types, records admit different data types within the newly created structure. Three choices exist for value assignments: The most obvious method is to assign one record to another (' **STUDENT_2 <= STUDENT_1** '). This does not allow to set individual values, however. Again, aggregates are commonly used for this purpose, i.e. the different element values are grouped together (e.g. ' **TODAY <= (26, JUL, 1988)** '). The single elements are addressed via RECORD.ELEMENT constructs, like ' **STUDENT_2.BIRTHDAY.YEAR <= 1974** '. As can be seen, this syntax applies also to nested records.

2.4.16 Type Conversion

```
architecture EXAMPLE of CONVERSION is
   type MY_BYTE is array (7 downto 0) of
std_logic;

   signal VECTOR:      std_logic_vector(7
downto 0);
   signal SOME_BITS:  bit_vector(7 downto 0);
   signal BYTE:           MY_BYTE;

begin

   SOME_BITS <= VECTOR;                        --
wrong
   SOME_BITS <= Convert_to_Bit( VECTOR ) ;

   BYTE <= VECTOR;                             --
wrong
   BYTE <= MY_BYTE( VECTOR ) ;

end EXAMPLE;
```

- **Data types have to match for assignments**
  - **type conversion functions**
  - **type cast**
- ***Closely related types***
  - **integer <-> real**
  - **arrays with the same length, index set and element types**

Matching data types are a strict language requirement in assignment operations. This can always be achieved via type conversion functions that have to be defined by the user for all necessary pairs of data types.

If the data types in question are so called "closely related", the call of a conversion function can be replaced by a more simple type cast. 'integer' and 'real' are closely related, for example, i.e. the following code line represents legal VHDL: ' **REAL_SIGNAL <= real(INTEGER_SIGNAL)** '. The syntax is similar to a function call, except that the desired data type is used directly as prefix.

Arrays are also called closely related when they are built of the same data type for their elements and coincide in their length and index set. Consequently, type casts occur frequently when dealing with vector operations as bit vectors (' **bit_vector** ', ' **std_(u)logic_vector** ') themselves do not have a numerical interpretation associated with them.

The arithmetic and relational operators for bit vectors operate with ' **signed** ' or ' **unsigned** ' data types that interpret the bit values as 2-complement (sign bit + absolute value) of an arbitrary integer number or binary representation of a positive integer value, respectively. These new data types are built of the same basic types (' **bit** ', ' **std_logic** '), i.e. whenever numerical operations are to be carried out with vectors, their interpretation is provided via the corresponding type cast expression. Of course, the type and operator definitions must be made available first (' **use IEEE.numeric_bit/std.all** ').

2.4.17 Subtypes

```
architecture EXAMPLE of SUBTYPES is
   type MY_WORD is array (15 downto 0) of std_logic;
   subtype SUB_WORD is std_logic_vector (15 downto 0);

   subtype MS_BYTE is integer range 15 downto 8;
   subtype LS_BYTE is integer range 7 downto 0;

   signal VECTOR:    std_logic_vector(15 downto 0);
   signal SOME_BITS: bit_vector(15 downto 0);
   signal WORD_1: MY_WORD;
   signal WORD_2: SUB_WORD;

begin
   SOME_BITS <= VECTOR;                    -- wrong
   SOME_BITS <= Convert_to_Bit(VECTOR);

   WORD_1 <= VECTOR;                       -- wrong
   WORD_1 <= MY_WORD(VECTOR);

   WORD_2 <= VECTOR;                       -- correct!

   WORD_2(LS_BYTE) <= "11110000";
end EXAMPLE;
```

- **Subsets of existing types**
- **Same type as the original type**
- **More readable signal assignments**
  - **Eliminates type casts and type conversions**
  - **Symbolic names for array ranges**

Instead of declaring a completely new data type it is possible to declare so called "subtypes", if the new type that is required is just a somewhat restricted version of another type. Subtypes have the same type as their original type and are therefore compatible to the basic type without the need for type casts or conversion functions. Subtypes can also be used to create symbolic names for array ranges.

This is synthesizable alternative to the ' **alias** ' construct in VHDL. Aliases are used to give another name to already existing objects. This way it is possible to break down complex data structures into simpler parts that can be accessed directly.

2.4.18 Aliases

```
architecture EXAMPLE of ALIAS is
    signal DATA is bit_vector(9 downto 0);

     alias STARTBIT :   bit is DATA(9) ;
    alias MESSAGE:  bit_vector(6 downto 0) is DATA (8 downto 2);
    alias PARITY:       bit is DATA(1);
    alias STOPBIT:    bit is DATA(0);
    alias REVERSE:  bit_vector(1 to 10) is DATA;

    function calc_parity(data: bit_vector) return bit is
    . . .

begin
    STARTBIT       <= '0';
    MESSAGE       <= "1100011";
    PARITY           <= calc_parity(MESSAGE);
    REVERSE(10) <= '1';

end EXAMPLE;
```

- **Give new names to already existing objects**
- **Make it easier to handle complex data structures**

Aliases are not always supported by synthesis tools

With an alias statement an object can be referenced by a new name. This way it is possible to break down complex data structures into simpler parts that can be accessed directly.

2.5 Operators

logical	not					
	and	or	nand	nor	xor	xnor
relational	=	/=	<	<=	>=	>
shift	sll	srl	sla	sra	rol	ror
arithmetic	+	-				
	*	/	mod	rem		
	**	abs				

sorted in order of increasing precedence (top->down)

New operators: xnor, shift operators

The table shows the operators predefined in VHDL. They are ordered by kind and precedence. Please note the new operators defined in VHDL'93.

2.5.1 Logical Operators

```
entity LOGIC_OP is
   port (A, B, C, D : in   bit;
            Z1:            out bit;
            EQUAL :    out boolean);
end LOGIC_OP;

architecture EXAMPLE of LOGIC_OP is
begin

   Z1 <= A  and (B or (not C xor D)));

   EQUAL <= A xor B;                -- wrong

end EXAMPLE;
```

- **Priority**
  - **not (top priority)**
  - **and, or, nand, nor, xor, xnor (equal priority)**
- **Predefined for**
  - **bit, bit_vector**
  - **boolean**
- **Data types have to match**

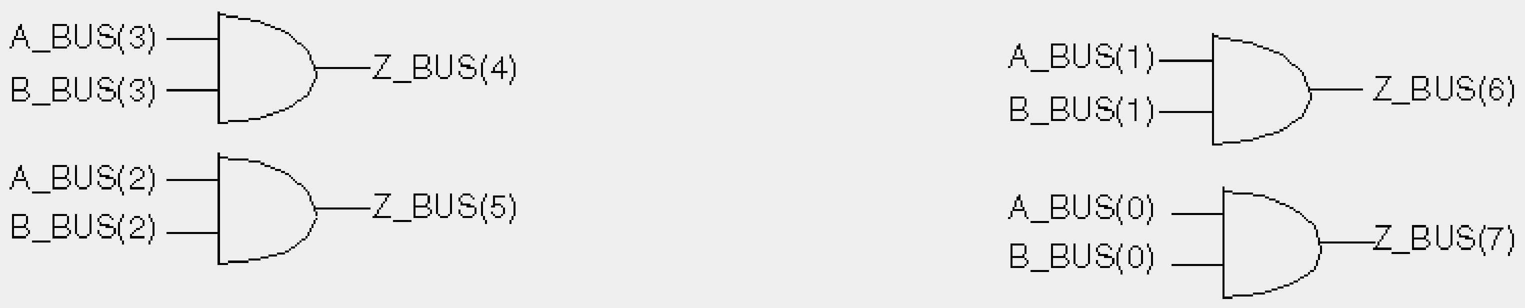
Brackets must be used to define the order of evaluation

Logical operations with arrays require operands of the same type and the same length. The operation is carried out element by element, then.

2.5.2 Logical Operations with Arrays

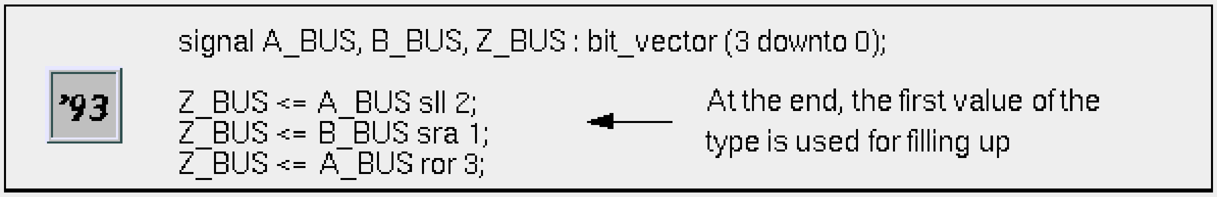
```
architecture EXAMPLE of LOGICAL_OP is
   signal A_BUS, B_BUS : bit_vector (3 downto 0);
   signal Z_BUS :             bit_vector (4 to 7);
begin
   Z_BUS <= A_BUS and B_BUS;
end EXAMPLE;
```

- **Operands of the same length and type**
- **Assignment via the position of the elements (according to range definition)**



2.5.3 Shift Operators: Examples

Defined only for one-dimensional arrays of bit of boolean!



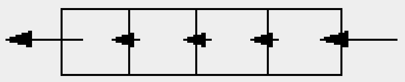
'93

```
signal A_BUS, B_BUS, Z_BUS : bit_vector (3 downto 0);

Z_BUS <= A_BUS sll 2;
Z_BUS <= B_BUS sra 1;
Z_BUS <= A_BUS ror 3;
```

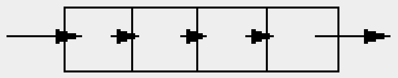
At the end, the first value of the type is used for filling up

Logical shift

sll

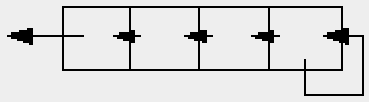


srl

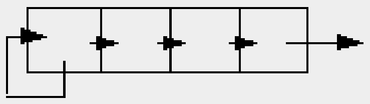


Arithmetic shift

sla

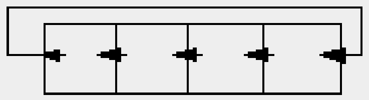


sra

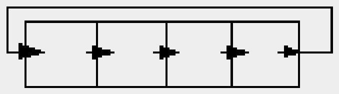


Rotation

rol



ror



2.5.4 Relational Operators

```
architecture EXAMPLE of
            RELATIONAL_OP is

  signal NR_A, NR_B:  integer;

  signal A_EQ_B1, A_EQ_B2:  bit;
  signal A_LT_B:            boolean;

begin
  -- A,B may be of any standard data type
  process (A, B)
  begin
    if (A = B) then
      A_EQ_B1 <= '1';
    else
      A_EQ_B1 <= '0';
    end if;
  end process;

  A_EQUAL_B2 <= A = B;      -- wrong

  A_LT_B <= B <= A;
end EXAMPLE
```

- **Predefined for all standard data types**
- **Result: boolean type (true, false)**

<	less than
<=	less or equal
=	equal
/=	unequal
>=	greater or equal
>	greater

2.5.5 Comparison Operations with Arrays

```
architecture EXAMPLE of
COMPARISON is
  signal NIBBLE: bit_vector(3
downto 0);
  signal BYTE:    bit_vector(0 to 7);
begin
  NIBBLE <= "1001";
  BYTE    <= "00001111";

  COMPARE: process (NIBBLE,
BYTE)
  begin
    if (NIBBLE < BYTE) then
      -- evaluated as:
      -- if (NIBBLE(3) < BYTE(0)) or
      --    ((NIBBLE(3) = BYTE(0))
and
      --     (NIBBLE(2) < BYTE(1)))
or
      --    ((NIBBLE(3) = BYTE(0))
and
      --     (NIBBLE(2) = BYTE(1))
and
      --     (NIBBLE(1) < BYTE(2)))
or
      ...
    -- better:
    if (( "0000"& NIBBLE) <= BYTE)
then
      ...

end EXAMPLE;
```

- **Operands of the same type**
- **Arrays:**
  - **may differ in length**
  - **left-alignment prior to comparison**
  - **are compared element after element**
- **No numerical interpretation (unsigned, 2-complement, etc.)**

Adjust the length of arrays prior to comparison

When comparing array types it must be considerd that the comparison is carried out from the left side to the right. That means for arrays of varying length that "111" is greater than "1001", for example. If it is not desired, hence shall be compared right-aligned, then the arrays will have to be brought upon the equal length by hand, e.g. by means of concatenation: '0'&"111" is then smaller than "1001".

2.5.6 Arithmetic Operators

+ addition	- substraction
* multiplication	** exponentiation
/ division	mod modulo
abs absolute value	rem remainder

```
signal A, B, C:    integer;
signal RESULT: integer;

RESULT<= -A + B * C;
```

- **Operands of the same type**
- **Predefined for**
  - **integer**
  - **real (except mod and rem)**
  - **physical types (e.g. time)**
- **Not defined for bit_vector (undefined number format: unsigned, 2-complement, etc.)**
- **Conventional mathematical meaning and priority**
- **'+' and '-' may also be used as unary operators**

The VHDL operators are rather self-explanatory. While relational operators are available for all pre-defined data types, the logical, shift and arithmetical operators may only be used with bit and numerical values, respectively.

Logical operations with arrays require operands of the same length. The operation is carried out element by element, then. This requirement does not exist for comparison operations with arrays. The arrays are left-aligned prior to comparison instead. Therefore it is recommended to adjust the length of the operands with the help of the concatenation operator.

Shift and rotation operations on arrays were introduced with the VHDL'93 standard. Rotation means that none of the element values is lost as the value that is rotated out of the array on one side will be used for the vacant spot on the other side. This is different from the shift operations where the value is discarded. During so called arithmetic shift operations the vacant spot receives its previous value; in case of logical shift operations the default value of the signal, i.e. the left-most value from the type declaration, will be used.

Please note, that two operations to calculate the remainder of an integer division are defined. The sign of the result of ' **rem** ' and ' **mod** ' operations is equal to the sign of the first and second operand, respectively.

Examples:

```
  5  rem   3  =  2,   5  mod   3  =  2 ( 5 =   1 *  3  +   2  )
(-5) rem   3  = -2, (-5) mod   3  =  1 (-5 = (-1)*  3  + (-2)
                                        = (-2)*  3  +   1) )
(-5) rem (-3) = -2, (-5) mod (-3) = -2 (-5 =   1 *(-3) + (-2) )
  5  rem (-3) =  2,   5  mod (-3) = -1 ( 5 = (-1)*(-3) +   2
                                        = (-2)*(-3) + (-1) )
```

2.5.7 Questions

	10. Which operators can be used on the std_ulogic_vector type?
○ yes ○ no	**10.1. ... Logic-operators**
○ yes ○ no	**10.2. ... Arithmetic-operators**
○ yes ○ no	**10.3. ... Comparison-operators**
	11. Arrays are compared...
○ yes ○ no	**11.1. ... according to their position.**
○ yes ○ no	**11.2. ... from left to right.**
	12. The array elements are assigned ...
○ yes ○ no	**12.1. ... automatically, according to their position.**
○ yes ○ no	**12.2. ... specifically, via the index (e.g.: Z_BUS(1) <= `3`;)**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.5.8 Questions

13. Which instructions represent correct VHDL?

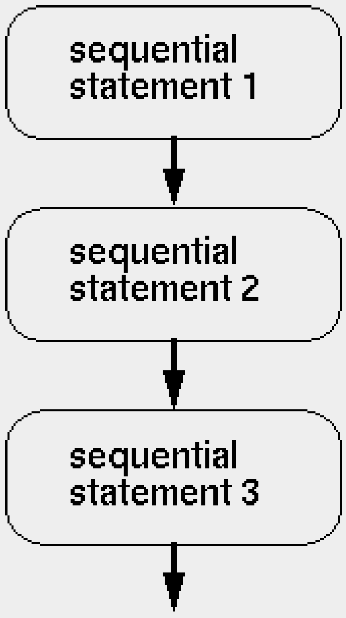
```
signal BOOL :                       boolean;
signal A_INT, B_INT, Z_INT:         integer range 0 to 15;
signal Z_BIT :                      bit;
signal A_VEC, B_VEC, Z_VEC:         bit_vector (3 downto 0);
signal A_VEC2, B_VEC2, Z_VEC2:      bit_vector (7 downto 0);
```

○ yes ○ no	**13.1. Z_BIT <= A_INT = B_INT;**
○ yes ○ no	**13.2. BOOL <= A_INT > B_VEC;**
○ yes ○ no	**13.3. Z_INT <= A_INT + B_INT;**
○ yes ○ no	**13.4. Z_INT <= A_INT = ``0001``;**
○ yes ○ no	**13.5. Z_BIT <= A_VEC and B_INT;**
○ yes ○ no	**13.6. Z_VEC <= A_VEC2 and B_VEC2;**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.6 Sequential Statements



- **exectuted according to the order in which they appear**
- **permitted only within processes**
- **used to describe algorithms**

All statements in processes or subprograms are processed sequentially, i.e. one after another. Like in ordinary programming languages there exist a variety of constructs to control the flow of execution. The if clause is probably the most obvious and most frequently used.

The IF condition must evaluate to a boolean value (true or false). After the first IF condition, any number of ELSIF conditions may follow. Overlaps may occur within different conditions. An ELSE branch, which combines all cases that have not been covered before, can optionally be inserted last. The IF statement is terminated with END IF.

The first IF condition has top priority: if this condition is fulfilled, the corresponding statements will be carried out and the rest of the IF - END IF block will be skipped.

The example code shows two different implementations of equivalent behaviour. The signal assignment to the signal Z in the first line of the left process (architecture EXAMPLE1) is called a default assignment, as its effects will only be visible if it is not overwritten by another assignment to Z. Note that the two conditions of the if and elsif part overlap, because X="1111" is also true when X>"1000". As a result of the priority mechanism of this if construct, Z will receive the value of B if X="1111".

2.6.1 IF Statement

```
if CONDITION then
   -- sequential statements
end if;

if CONDITION then
   -- sequential statements
else
   -- sequential statements
end if;

if CONDITION then
   -- sequential statements
elsif CONDITION then
   -- sequential statements
. . .
else
   -- sequential statements
end if;
```

- **Condition is a boolean expression**
- **Optional elsif sequence**
  - **conditions may overlap**
  - **priority**
- **Optional else path**
  - **executed, if all conditions evaluate to false**

Attention: elsif but end if

The if condition must evaluate to a boolean value ('true' or 'false'). After the first if condition, any number of elsif conditions may follow. Overlaps may occur within different conditions. An else branch, which combines all cases that have not been covered before, can optionally be inserted last. The if statement is terminated with 'end if'.

The first if condition has top priority: if this condition is fulfilled, the corresponding statements will be carried out and the rest of the 'if - end if' block will be skipped.

2.6.2 IF Statement: Example

```
entity IF_STATEMENT is
   port (A, B, C, X : in   bit_vector (3 downto 0);
         Z          : out bit_vector (3 downto 0);
end IF_STATEMENT;
```

```
architecture EXAMPLE1 of IF_STATEMENT is
begin
   process (A, B, C, X)
   begin
      Z <= A;
       if  (X = "1111")  then
         Z <= B;
       elsif  (X > "1000")  then
         Z <= C;
       end if;
   end process;
end EXAMPLE1;
```

```
architecture EXAMPLE2 of IF_STATEMENT is
begin
   process (A, B, C, X)
   begin

       if  (X = "1111")  then
         Z <= B;
       elsif  (X > "1000")  then
         Z <= C;
       else
         Z <= a;
       end if;
   end process;
end EXAMPLE2;
```

The example code shows two different implementations of equivalent behaviour. The signal assignment to the signal Z in the first line of the left process (architecture EXAMPLE1) is called a default assignment, as its effects will only be visible if it is not overwritten by another assignment to Z. Note that the two conditions of the 'if' and 'elsif' part overlap, because X="1111" is also true when X>"1000". As a result of the priority mechanism of this if construct, Z will receive the value of B if X="1111".

2.6.3 CASE Statement

```
case EXPRESSION is

   when VALUE_1 =>
    -- sequential statements

   when VALUE_2 | VALUE_3 =>
    -- sequential statements

   when VALUE_4 to VALUE_N
=>
    -- sequential statements

   when others =>
    -- sequential statements

end case ;
```

- **Choice options must not overlap**
- **All choice options have to be covered**
  - **single values**
  - **value range**
  - **selection of values ("|" means "or")**
  - **"when others" covers all remaining choice options**

While the priority of each branch is set by means of the query's order in the IF case, all branches are equal in priority when using a CASE statement. Therefore it is obvious that there must not be any overlaps. On the other hand, all possible values of the CASE EXPRESSION must be covered. For covering all remaining, i.e. not yet covered, cases, the keyword ' **others** ' may be used.

The type of the EXPRESSION in the head of the CASE statement has to match the type of the query values. Single values of EXPRESSION can be grouped together with the '|' symbol, if the consecutive action is the same. Value ranges allow to cover even more choice options with relatively simple VHDL code.

Ranges can be defined for data types with a fixed order, only, e.g. user defined enumerated types or integer values. This way, it can be decided whether one value is less than, equal to or greater than another value. For ARRAY types (e.g. a BIT_VECTOR) there is no such order, i.e. the range "0000" TO "0100" is undefined and therefore not admissible.

2.6.4 CASE Statement: Example

```
entity CASE_STATEMENT is
   port (A, B, C, X : in   integer range 0 to 15;
            Z             : out integer range 0 to 15;
end CASE_STATEMENT;

architecture EXAMPLE of CASE_STATEMENT is
begin
   process (A, B, C, X)
   begin
       case  X  is
          when  0  =>
            Z <= A;
          when  7  |  9  =>
            Z <= B;
          when  1 to 5  =>
            Z <= C;
          when others =>
            Z <= 0;
       end case;
   end process;
end EXAMPLE;
```

2.6.5 Defining Ranges

```
entity RANGE_1 is
port (A, B, C, X : in   integer range 0 to 15;
      Z          : out integer range 0 to 15;
end RANGE_1;

architecture EXAMPLE of RANGE_1 is
begin
  process (A, B, C, X)
  begin
    case X is
      when 0 =>
        Z <= A;
      when 7 | 9 =>
        Z <= B;
      when 1 to 5 =>
        Z <= C;
      when others =>
        Z <= 0;
    end case;
  end process;
end EXAMPLE;
```

```
entity RANGE_2 is
port (A, B, C, X : in    bit_vector(3 downto 0);
      Z          : out  bit_vector(3 downto 0);
end RANGE_2;

architecture EXAMPLE of RANGE_2 is
begin
  process (A, B, C, X)
  begin
    case X is
      when "0000" =>
        Z <= A;
      when "0111" | "1001" =>
        Z <= B;
      when "0001" to "0101" =>       -- wrong
        Z <= C;
      when others =>
        Z <= 0;
    end case;
  end process;
end EXAMPLE;
```

The sequence of values is undefined for arrays

Ranges can be defined for data types with a fixed order, only, e.g. user defined enumerated types or integer values. This way, it can be decided whether one value is less than, equal to or greater than another value. For array types (e.g. a 'bit_vector') there is no such order, i.e. the 'range "0000" to "0100"' is undefined and therefore not admissible.

2.6.6 FOR Loops

```
entity FOR_LOOP is
  port (A : in    integer range 0 to 3;
        Z  : out bit_vector (3 downto 0));
end FOR_LOOP;

architecture EXAMPLE of FOR_LOOP is
begin
  process (A)
  begin
    Z <= "0000";
     for  I  in  0 to 3  loop
       if (A = I) then
          Z(I) <= `1`;
       end if;
     end loop;
  end process;
end EXAMPLE;
```

- **Loop parameter is implicitly declared**
  - **can not be declared externally**
  - **read only access**
- **The loop parameter adopts all values from the range definition**
  - **integer ranges**
  - **enumerated types**

Loops operate in the usual way, i.e. they are used to execute the same some VHDL code a couple of times. Loop labels may be used to enhance readability, especially when loops are nested or the code block executed within the loop is rather long. The loop variable is the only object in VHDL which is implicitly defined. The loop variable can not be declared externally and is only visible within the loop. Its value is read only, i.e. the number of cycles is fixed when the execution of the for loop begins.

If a for loop is to be synthesized, the range of the loop variable must not depend on signal or variable values (i.e., it has to be locally static). By means of the range assignment, both the direction and the range of the loop variable is determined. If a variable number of cycles is needed, the while statement will have to be used. While loops are executed as long as CONDITION evaluates to a 'true' value. Therefore this construct is usually not synthesizable.

2.6.7 Loop Syntax

```
[LOOP_LABEL :]
for IDENTIFIER in
DISCRETE_RANGE loop
  -- sequential statements
end loop [LOOP_LABEL] ;

[LOOP_LABEL :]
while CONDITION loop
  -- sequential statements
end loop [LOOP_LABEL] ;
```

- **Optional label**
- **FOR loop identifier**
  - **not declared**
  - **read only**
  - **not visible outside the loop**
- **Range attributes**
  - **`low**
  - **`high**
  - **`range**

Synthesis requirements:
- Loops must have a fixed range
- 'while' constructs usually cannot be synthesized

The loop labe is optional. By defining the range the direction as well as the possible values of the loop variable are fixed. The loop variable is only accessable within the loop.
For synthesis the loop range has to be locally static and must not depend on signal or variable values. While loops are not generally synthesizable.

2.6.8 Loop Examples

```
entity CONV_INT is
   port (VECTOR: in   bit_vector(7 downto 0);
            RESULT:  out integer);
end CONV_INT;
```

```
architecture A of CONV_INT is
begin
   process(VECTOR)
      variable TMP: integer;

   begin
      TMP := 0;

      for I in 7 downto 0 loop
         if (VECTOR(I)='1') then
            TMP := TMP + 2**I;
         end if;
      end loop;

      RESULT <= TMP;
   end process;
end A;
```

```
architecture B of CONV_INT is
begin
   process(VECTOR)
      variable TMP: integer;

   begin
      TMP := 0;

      for I in VECTOR'range loop
         if (VECTOR(I)='1') then
            TMP := TMP + 2**I;
         end if;
      end loop;

      RESULT <= TMP;
   end process;
end B;
```

```
architecture C of CONV_INT is
begin
   process(VECTOR)
      variable TMP: integer;
      variable I      : integer;
   begin
      TMP := 0;
      I := VECTOR'high;
      while (I >= VECTOR'low) loop
         if (VECTOR(I)='1') then
            TMP := TMP + 2**I;
         end if;
         I := I - 1;
      end loop;
      RESULT <= TMP;
   end process;
end C;
```

The three loop example architectures are functionally equivalent. The difference lies in the specification of the loop range. Architectures A and B use the for statement. Instead of a hard coded range specification, signal attributes that are dependent on the signal type and are therefore fixed during runtime in architecture B. Architecture C shows an equivalent implementation using a while construct. Please note, that an additional loop variable I has to be declared in this case.

Range attributes are used to make the same VHDL code applicable to a number of signals, independent of their width. They are especially usefull when dealing with integer or array types.The following lines represent equal functionality, provided that Z's range is from 0 to 3.

```
for I in 0 to 3 loop
for I in Z'low to Z'high loop
for I in Z'range loop
```

2.6.9 WAIT Statement

- 'wait' statements stop the process execution
  - The process is continued when the instruction is fulfilled
- Different types of wait statements:

❍ wait for a specific time	`wait for SPECIFIC_TIME;`
❍ wait for a signal event	`wait on SIGNAL_LIST;`
❍ wait for a true condition (requires an event)	`wait until CONDITION;`
❍ indefinite (process is never reactivated)	`wait;`

⚠ **Wait statements must not be used in processes with sensitivity list**

As mentioned before, processes may be coded in two flavours. If the sensitivity list is omitted, another method will be needed to to stop process execution. Wait statements put the process execution on hold until the specified condition is fullfilled. If no condition is given, the process will never be reactivated again.

Wait statements must not be combined with a sensitivity list, independent from the application field.

2.6.10 WAIT Statement: Examples

- **Flip Flop model**

```
entity FF is
     port (D, CLK : in   bit;
             Q          : out bit);
end FF;
```

```
architecture BEH_1 of FF is
begin
   process
   begin
      wait on CLK;
      if (CLK = '1') then
         Q <= D;
      end if;
   end process;
end BEH_1;
```

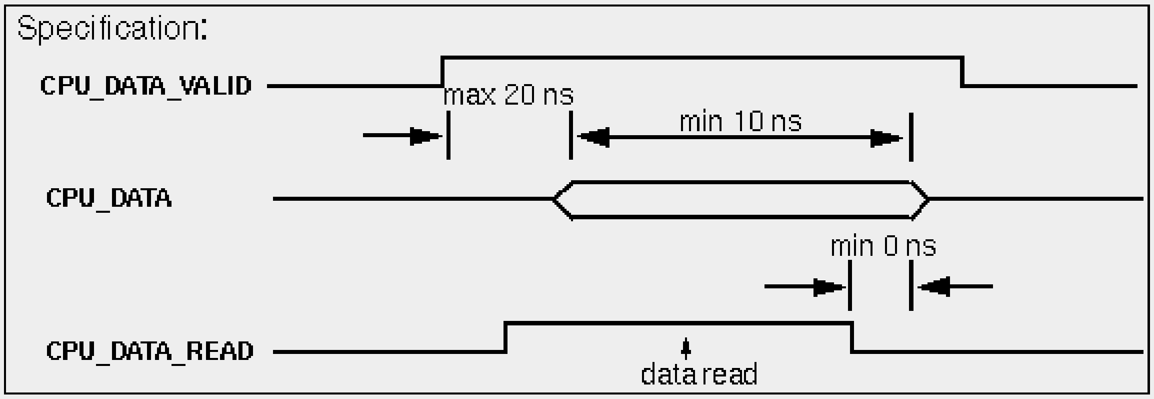
```
architecture BEH_2 of FF is
begin
   process
   begin
      wait until CLK=`1`;

      Q <= D;

   end process;
end BEH_2;
```

- **Testbench: stimuli generation**

```
STIMULUS: process
begin
      SEL      <= `0`;
      BUS_B <= "0000";
      BUS_A <= "1111";
      wait for 10 ns;

      SEL <= `1`;
      wait for 10 ns;

      SEL <= `0`;
      wait for 10 ns;

      wait;
end process STIMULUS;
```

Processes without sensitivity list are executed until a wait statement is reached. In the example architecture BEH_1 of a Flip Flop, the execution resumes as soon as an event is detected on the CLK signal ('wait on CLK). The following if statement checks the level of the clock signal and a new output value is assigned in case of a rising edge. In BEH_2, both checks are combined in a single 'wait until' statement. The evaluation of the condition is triggered by signal events, i.e. the behaviour is the same. Via 'wait for' constructs it is very easy to generate simple input patterns for design verification purposes.

2.6.11 WAIT Statements and Behavioural Modeling



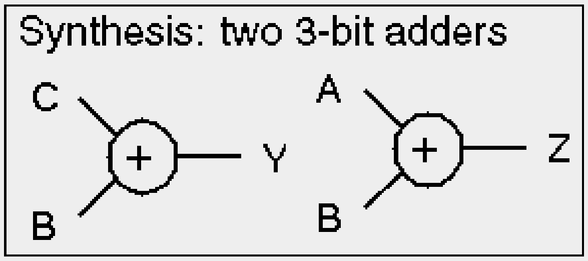
- **Timing behaviour from specification**
- **Translation into VHDL**
- **Based on time**

```
READ_CPU : process
begin
    wait until CPU_DATA_VALID = `1`;
    CPU_DATA_READ <= `1`;
    wait for 20 ns;
    LOCAL_BUFFER <= CPU_DATA;
    wait for 10 ns;
    CPU_DATA_READ <= `0`;
end process READ_CPU;
```

Wait constructs, in general, are an excellent tool for describing timing specifications. For example it is easy to implement a bus protocol for simulation. The timing specification can directly be translated to simulatable VHDL code. But keep in mind that this behavioural modelling can only be used for simulation purposes as it is definitely not syntheziable.

2.6.12 Variables

```
architecture RTL of XYZ is
     signal A, B, C : integer range 0 to 7;
     signal Y, Z :       integer range 0 to 15;
begin
     process (A, B, C)
       variable M, N : integer range 0 to 7;
     begin
        M := A;
        N := B;
        Z <= M + N;
        M := C;
        Y <= M + N;
     end process;
end RTL;
```



- **Variables are available within processes, only**
  - **named within process declarations**
  - **known only in this process**
- **VHDL 93: shared variables**
- **immediate assignment**
- **keep the last value**
- **Possible assignments**
  - **signal to variable**
  - **variable to signal**
  - **types have to match**

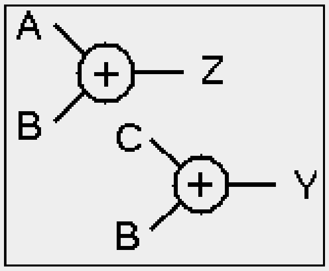
Variables can only be defined in a process and they are only accessible within this process.

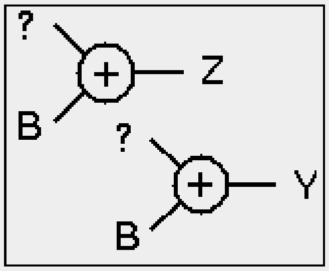
Variables and signals show a fundamentally different behaviour. In a process, the last signal assignment to a signal is carried out when the process execution is suspended. Value assignments to variables, however, are carried out immediately. To distinguish between a signal and a variable assignment different symbols are used: ' **<=** ' indicates a signal assignment and ' **:=** ' indicates a variable assignment.

2.6.13 Variables vs. Signals

```
signal A, B, C, Y, Z : integer;

begin
   process (A, B, C)
       variable M, N : integer;
   begin
      M := A;
      N := B;
      Z <= M + N;
      M := C;
      Y <= M + N;
   end process;
```

```
signal A, B, C, Y, Z : integer;
signal M, N          : integer;
begin
    process (A, B, C, M, N)

    begin
       M <= A;
       N <= B;
       Z <= M + N;
       M <= C;
       Y <= M + N;
    end process;
```

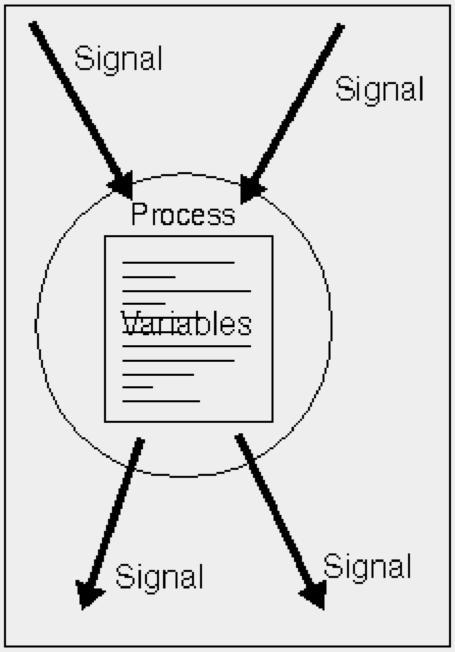




- **Signal values are assigned after the process execution**
- **Only the last signal assignment is carried out**
- **M <= A;**
  is overwritten by
  M <= C;
- **The 2nd adder input is connected to C**

The two processes shown in the example implement different behaviour as both outputs X and Y will be set to the result of B+C when signals are used instead of variables. Please note that the intermediate signals have to added to the sensitivity list, as they are read during process execution.

2.6.14 Use of Variables



- **Intermediate results of algorithm implementations**
  - **signal to variable assignment**
  - **execution of algorithm**
  - **variable to signal assignment**
- **No access to variable values outside their process**
- **Variables store their value until the next process call**

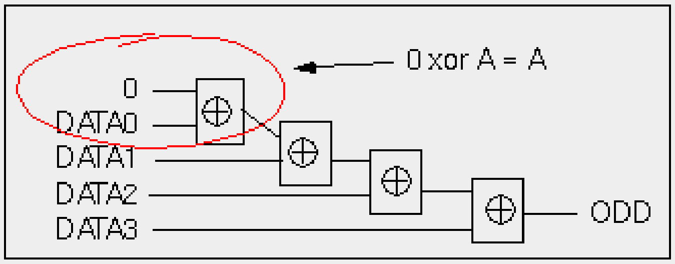
Variables are especially suited for the implementation of algorithms. Usually, the signal values are copied into variables before the algorithm is carried out. The result is assigned to a signal again afterwards. Variables keep their value from one process call to the next, i.e.if a variable is read before a value has been assigned, the variable will have to show storage behaviour. That means it will have to be synthesized to a latch or flip-flop respectively.

2.6.15 Variables: Example

- **Parity calculation**

```
entity PARITY is
     port (DATA: in   bit_vector (3 downto 0);
            ODD : out bit);
end PARITY;


architecture RTL of PARITY is
begin
    process (DATA)
      variable TMP : bit;
    begin
      TMP := `0`;

      for I in  DATA`low to DATA`high  loop
        TMP := TMP xor DATA(I);
      end loop;

      ODD <= TMP;
    end process;
end RTL;
```

Synthesis result:



In the example a further difference between signals and variables is shown. While a (scalar) signal can always be associated with a line, this is not valid for variables. In the example the for loop is executed four times. Each time the variable TMP describes a different line of the resulting hardware. The different lines are the outputs of the corresponding XOR gates.

2.6.16 Global Variables (VHDL'93)

```
architecture BEHAVE of SHARED is
        shared variable S  : integer;
begin
        process (A, B)
        begin
                S :=  A + B;
        end process;

        process (A, B)
        begin
                S :=  A - B;
        end process;
end BEHAVE;
```

- **Accessible by all processes of an architecture (shared variables)**
- **Can introduce non-determinism**

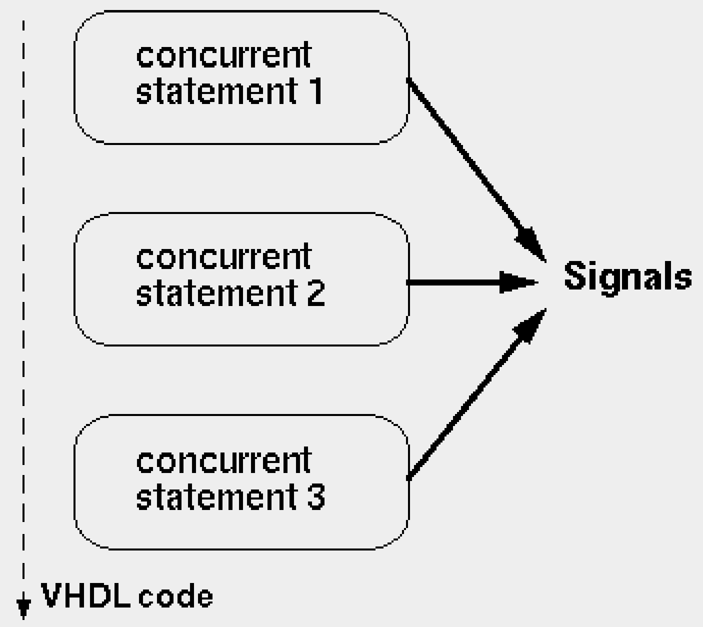
Not to be used in synthesizable code

In VHDL 93, global variables are allowed. These variables are not only visible within a process but within the entire architecture. The problem may occur, that two processes assign a different value to a global variable at the same time. It is not clear then, which of these processes assigns the value to the variable last. This can lead to a non-deterministic behaviour!

In synthesizabel VHDL code global variables must not be used.

2.7 Concurrent Statements

- **Concurrent statements are exectuted at the same time, independent of the order in which they appear**



All statements within architectures are executed concurrently. While it is possible to use VHDL processes as the only concurrent statement, the necessary overhead (process, begin, end, sensitivity list) lets designer look for alternatives when the sequential behaviour of processes is not needed.
The signal assignment statement was the first VHDL statement to be introduced. The signal on the left side of the assignment operator '<=' receives a new value whenever a signal on the right side changes. The new value stems from another signal in the most simple case (i.e. when an intermediate signal is necessary to match different port modes) or can be calculated from a number of signals.

2.7.1 Conditional Signal Assignment

```
TARGET <= VALUE;
TARGET <= VALUE_1 when
CONDITION_1 else
                VALUE_2 when
CONDITION_2 else
                . . .
                VALUE_n;
```

- **Condition is a boolean expression**
- **Mandatory else path, unless unconditional assignment**
  - **conditions may overlap**
  - **priority**
- **Equivalent of if ..., elsif ..., else constructs**

The signal assignment can be extended by the specification of conditions. The condition is appended after the new value and is introduced by the keyword ' **when** '. The keyword ' **else** ' is also strictly necessary after each condition as an unconditional signal assigment has to be present. Consequently, it is not possible to generate storage elements with an conditional signal assignment. Otherwise the behaviour is equivalent to the if ..., elsif ..., else ... construct that is used within processes.

2.7.2 Conditional Signal Assignment: Example

```
entity CONDITIONAL_ASSIGNMENT is
  port (A, B, C, X : in   bit_vector (3 downto 0);
        Z_CONC : out bit_vector (3 downto 0);
        Z_SEQ    : out bit_vector (3 downto 0));
end CONDITIONAL_ASSIGNMENT;

architecture EXAMPLE of CONDITIONAL_ASSIGNMENT is
begin
  -- Concurrent version of conditional signal assignment
  Z_CONC <= B when X = "1111" else
               C when X > "1000" else
               A;

  -- Equivalent sequential statements
  process (A, B, C, X)
  begin
     if  (X = "1111")  then
       Z_SEQ <= B
     elsif  (X > "1000")  then
       Z_SEQ <= C;
    else
       Z_SEQ <= A;
     end if;
  end process;
end EXAMPLE;
```

In the example, two equivalent descriptions of a simple multiplexer are given. Please note that all signals appearing on the right side of the signal assignment operator are entered into the process' sensitivity list. The unconditional else path could be replaced by an unconditional signal assignment in front of the if statement. This assignments would be overwritten, if any of the conditions were true.

2.7.3 Selected Signal Assignment

```
with EXPRESSION select

  TARGET <= VALUE_1 when CHOICE_1,

            VALUE_2 when CHOICE_2 | CHOICE_3,

            VALUE_3 when CHOICE_4 to CHOICE_5,

            . . .

            VALUE_n when others;
```

- **Choice options must not overlap**
- **All choice options have to be covered**
  - **single values**
  - **value range**
  - **selection of values ("|" means "or")**
  - **"when others" covers all remaining choice options**
- **Equivalent of case ..., when ... constructs**

The behaviour of the so called selected signal assignment is similar to the case statement. It suffers from the same restrictions as its sequential counterpart, namely that all possible choice options have to be covered and none of the choice options may overlap with another.

2.7.4 Selected Signal Assignment: Example

```
entity SELECTED_ASSIGNMENT is
   port (A, B, C, X : in   integer range 0 to 15;
            Z_CONC : out integer range 0 to 15;
            Z_SEQ    : out integer range 0 to 15);
end SELECTED_ASSIGNMENT;


architecture EXAMPLE of SELECTED_ASSIGNMENT is
begin
   -- Concurrent version of selected signal assignment
   with X select
      Z_CONC <= A when 0,
                     B when 7 | 9,
                     C when 1 to 5,
                     0 when others;

   -- Equivalent sequential statements
   process (A, B, C, X)
   begin
      case X is
         when 0         => Z_SEQ <= A;
         when 7 | 9     => Z_SEQ <= B;
         when 1 to 5    => Z_SEQ <= C;
         when others => Z_SEQ <= 0;
   end process;
end EXAMPLE;
```

Like with conditional signal assignments, the signal assignment operator '<=' can be seen as the core of the construct. Again, the choice options are appended after the keyword 'when', yet the different assignment alternatives are separated by ',' symbols. The equivalent of the 'case EXPRESSION is' construct from the case statement must be placed as header line in front of the actual assignment specification. The keywords have to be translated, however, to 'with EXPRESSION select'.

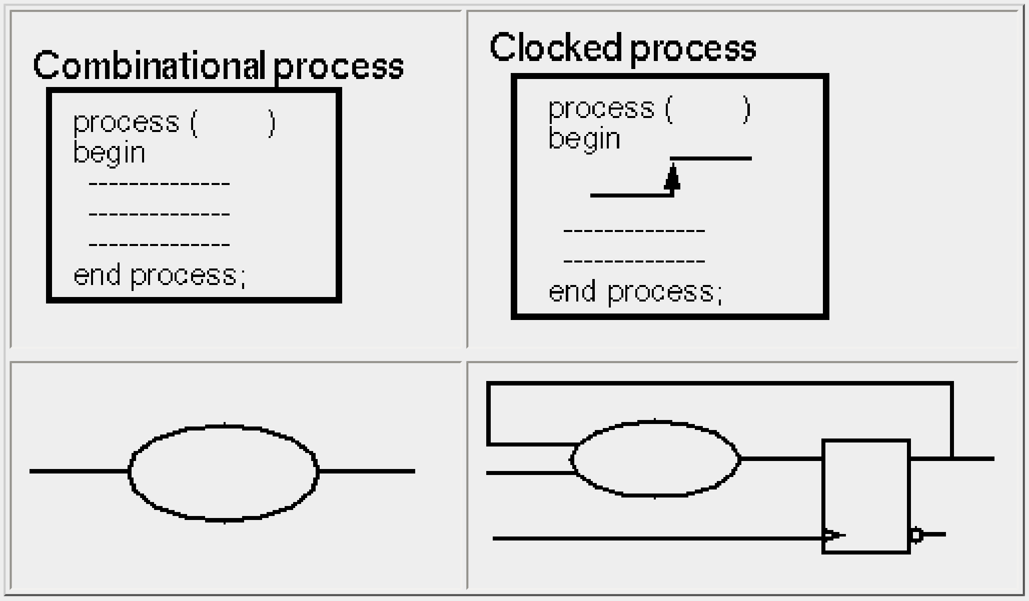
2.7.5 Concurrent Statements: Summary

- **Modelling of multiplexers**
  - **conditional signal assignment: decision based upon several signals**
  - **selected signal assignment: decision based upon values of a single signal**
- **"Shortcuts" for sequential statements**
  - **conditional signal assignment <=> if ..., elsif ..., else ..., end if**
  - **selected signal assignment <=> case ..., when ..., end case**

⚠ **Unconditional else path is mandatory in conditional signal assignments**

All concurrent statements describe the functionality of multiplexer structures. It is impossible to model storage elements, like Flip Flops with concurrent statements, only. Consequently, the unconditional else path is necessary in conditional signal assignments. Every concurrent signal assignment, whether conditional or selected, can be modeled with a process construct, however. As sequentially executed code is easier comprehensible, the concurrent versions should be used as shortcut when simple functionality would be obfuscated by the process overhead, only.

2.8 RTL-Style



In RTL (Register Transfer Level) style modeling, the design is split up into storing elements, i.e. Flip Flops or often simply called registers, and combinatorics which constitute the transfer function from one register to the succeeding register. A process is required for each of them: a combinational process, which describes the functionality, and a clocked process, which generates all storing elements. Of course, it is possible to combine these two processes into a single clocked one which models the complete functionality.

2.8.1 Combinational Process: Sensitivity List

```
process ( A, B, SEL )
begin
  if (SEL = '1') then
    OUT <= A;
  else
    OUT <= B;
  end if;
end process;
```

- **Sensitivity list is usually ignored during synthesis**
- **Equivalent behaviour of simulation model and hardware:**

  sensitivity list has to contain all signals that are read by the process

What kind of hardware is modelled?

What will be the simulation result if SEL is missing in the sensitivity list?

The sensitivity list of a combinational process consists of all signals which will be read within the process. It is especially important not to forget any signals, because synthesis tools generally ignore sensitivity lists in contrast to simulation tools. During simulation, a process will only be executed, if there an event occurs on at least one of the signals of the sensitivity list. During synthesis, VHDL code is simply mapped to logic elements. Consequently a forgotten signal in the sensitivity list will most probably lead to a difference in behaviour between the simulated VHDL model and the synthesized design. Superflouos signals in the sensitivity list will only slow down simulation speed.

The code example models a multiplexer. If the signal SEL was missing, synthesis would create exactly the same result, namely a multiplexer, but simulation will show a completely different behaviour. The multiplexer would work properly as long as an event on SEL would coincide with events on A or B. But without an event on A or B the process would not be activated and thus an event exclusively on SEL would be ignored in simulation. Consequently, during simulation, the output value OUT would only change, if the the input signals A or B were modified.

2.8.2 WAIT Statement <-> Sensitivity List

```
process
begin
  if SEL = `1` then
    Z <= A;
  else
    Z <= B;
  end if;
   wait on A, B, SEL;
end process;
```

```
process  (A, B, SEL)
begin
  if SEL = `1` then
    Z <= A;
  else
    Z <= B;
  end if;

end process;
```

- **equivalent Processes**
- **mutually exclusive:**
  - **either sensitivity list**
  - **or wait statements**

Instead of using a sensitivity list, it is possible to model the same behaviour by the use of a WAIT ON statement. It should be placed as last statement in the process and should quote the same signals of course.

In case of a sensitivity list, the process is started whenever an event occurs on one of the signals in the list. All sequential statements are executed and after the last sequential statement the process is suspended until the next event. In case of a wait statement, the process runs through the sequential statements to the wait statement and suspends until the condition of the wait statement is fulfilled. Process execution must be interrupted via wait statements if no sensitivity list is present as the simulator would be stuck in an endless loop otherwise.

Please remember again, that it is not allowed to use a sensitivity list and a wait statement simultanously in the same process.

2.8.3 Combinational Process: Incomplete Assignments

```
entity MUX is
port (A, B, SEL : in    std_logic;
         Z              : out  std_logic);
end MUX;
```

```
architecture WRONG of MUX is
begin
  process (A, B, SEL)
  begin

     if SEL = `1` then
        Z <= A;
     end if;
  end process;
end WRONG;
```

```
architecture OK_1 of MUX is
begin
  process (A, B, SEL)
  begin
     Z <= B;
     if SEL = `1` then
        Z <= A;
     end if;
  end process;
end OK_1;
```

```
architecture OK_2 of MUX is
begin
  process (A, B, SEL)
  begin

     if SEL = `1` then
        Z <= A;
     else
        Z <= B;
     end if;
  end process;
end OK_2;
```

What is the value of Z if SEL = `0` ?

What hardware would be generated during synthesis ?

Special care is necessary when modeling combinational hardware in order to avoid the generation of latches. The leftmost code example lacks an unconditional else branch. Therefore the value of Z is preserved in case of SEL='0', even if the input signals change. Synthesis would have to generate an adequate storing element, i.e. a latch which is transparent whenever the level of SEL is '1'.

This kind of storing elements is **not** recommended for synchronous designs. Edge triggered Flip Flops are prefered because possibly illegal intermediate signal values are filtered out as long as the combinational logic settles to its final state before the next active clock edge. Additionally latches cannot be tested by a scan test. In scan test mode all Flip Flops are combined to a single shift register the so called scan path . They are all supplied with the same clock signal. This maked it possible to set all registers to specific values by shifting them into the chip using an additional input pin (scan_in). After one system clock period the registers contain new values which are shifted out using an additional output pin (scan_out). This way, scan test provides access to otherwise invisible internal states. Scan test is current state of the art technology to improve testability for production tests.

The two coding alternatives are functionally identical and are mapped to purely combinational logic (multiplexer) by synthesis tools. The difference lies in the implementation of the default assignment. Please remember that signal values are updated at the end of the process execution, only! This way the default assignment of B to Z in the architecture OK_1 will be overwritten if the if condition is true.

2.8.4 Combinational Process: Rules

- Complete sensitivity list
  - RTL behaviour identical with hardware realization
  - incomplete sensitivity lists can cause warnings or errors
- No incomplete if statements
  - inference of transparent latches

2.8.5 Clocked Process: Clock Edge Detection

- **New standard for synthesis: IEEE 1076.6**

- ***if***
  - ***clock_signal_*name'EVENT and *clock_signal_*name='1'**
  - ***clock_signal_*name='1' and *clock_signal_*name'EVENT**
  - **not *clock_signal_*name'STABLE and *clock_signal_*name='1'**

- ***wait until***
  - ***clock_signal_*name'EVENT and *clock_signal_*name='1'**
  - ***clock_signal_*name='1' and *clock_signal_*name'EVENT**
  - **not *clock_signal_*name'STABLE and *clock_signal_*name='1'**
  - ***clock_signal_*name='1' and**

❍ *clock_signal_* name='1' and not *clock_signal_* name'STABLE ❍ RISING_EDGE (*clock_signal_* name)	not *clock_signal_* name'STABLE ❍ RISING_EDGE (*clock_signal_* name) ❍ *clock_signal_* name='1'

IEEE 1076.6 is not fully supported by all tools, yet

As the sensitivity list is usually ignored by synthesis tools and wait statements are not synthesizable in general, a solution to the problem of modeling storage elements has to be found. Synthesis tools solved this issue by looking for certain templates in the VHDL code, namely the first option ('if/wait until X'event and X='1' then') of the two process styles. All alternatives show the same behaviour during simulation, however. Please note that the event detection in the 'wait until' statement is redundant as an event is implicitly required by the 'wait until' construct.

In the meantime, the IEEE standard 1076.6 was passed that lists that the VHDL constructs that should infer register generation. As this standard is not fully supported by synthesis tools, yet, the first option is still the most common way of describing a rising/falling clock edge for synthesis. When asynchronous set or reset signals are present, only the IF variant is applicable.

2.8.6 Detection of a Rising Edge by Use of Functions

- **Defined in std_logic_1164 package**

```
process
begin
   wait until  RISING_EDGE (CLK);
   Q <= D;
end process;
```

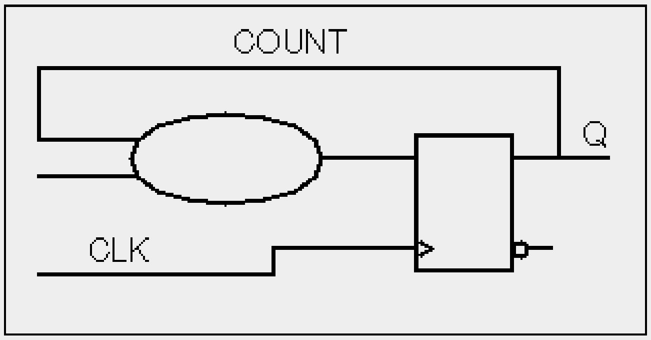
```
function RISING_EDGE (signal CLK : std_ulogic)
             return boolean is
begin
     if (CLK`event and CLK = `1` and CLK`last_value = `0`) then
         return true;
     else
         return false;
     end if;
end RISING_EDGE;
```

The RISING_EDGE function is just mentioned for sake of completeness as it is not supported by synthesis tools. Nevertheless it may be useful for simulation.

2.8.7 Register Inference

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

entity COUNTER is
port (CLK:  in std_logic;
      Q   :  out integer range 0 to 15 );
end COUNTER;

architecture RTL of COUNTER is
  signal COUNT : integer
range 0 to 15 ;
begin
  process (CLK)
  begin
    if CLK`event and CLK = `1` then
      if (COUNT >= 9) then
        COUNT <= 0;
      else
        COUNT <= COUNT +1;
      end if;
    end if;
  end process;


  Q <=  COUNT ;
end RTL;
```

- **Storage elements are synthesized for all signals that are driven within a clocked process**
  - **COUNT: 4 flip flops**
  - **Q: not used in clocked process**



The example shows the VHDL model of a simple 1 digit decimal counter. Several things are worth mentioning:

First, the design is not resetable. This is not a problem for simulation as initial values can be assigned to the signals. Real world hardware does not behave this nicely, however, and the state of internal storage elements is unknown after power up. In order to avoid strange and inexplicable behaviour it is recommended to provide a reset feature that brings the design into a well defined state.

Second, the range of the integer signals has been restricted. Synthesis tools have to map all data types onto a bit pattern that can be transported via wires. Without explicit range definition, the range for integers would be from -2,147,483,647 to +2,147,483,647, which is equivalent of 32 bits. As the maximum counter value is 9 it would be natural to specify a valid range for signal values from 0 to 9. During synthesis, however, 4 bits will be needed to represent the number 9, i.e. the theoratical maximum value of the signal would be 15. In order to avoid possible shortcomings of synthesis tools and to make sure that the counter restarts with 0, indeed, the range is set to match the synthesized hardware.

Third, the internal signal COUNT has been declared in addition to the output port Q. This is due to the fact that Q is declared with port mode ' **out** ', i.e. its value can not be read within the architecture. As its next value depends on the previous one, however, it is necessary to declare the intermediate signal COUNT which is used within the counter process. The process itself is a clocked process without any asynchronous control signals and thus the CLK signal is the only signal in the sensitivity list.

Flip Flops are infered by clocked processes, only. Every signal that might be updated in a clocked process receives a register. Therefore, four storage elements will be created for the COUNT signal. The assignment of the output value is done concurrently, i.e. the outputs of the Flip Flops will be connected directly to the outputs of the COUNTER module.

2.8.8 Asynchronous Set/Reset

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

entity ASYNC_FF is
port (D, CLK, SET, RST : in   std_logic;
       Q                       : out std_logic);
end ASYNC_FF;

architecture RTL of ASYNC_FF is
begin
   process  (CLK, RST, SET)
   begin
      if (RST = `1`) then
        Q <= `0`;

     elsif SET ='1' then

        Q <= '1';
      elsif (CLK`event and CLK = `1`) then
        Q <= D;
      end if;
   end process;
end RTL;
```

- **Only possible in processes with sensitivity list**
- **If / elsif - structure**
  - **clock edge detection as last condition**
  - **no unconditional else branch**

As noted before, it is advisable to provide each clocked design with a reset capability. If a synchronous reset strategy is employed, the reset signal is treated just like any other control signal, i.e. the clock signal will be still the only signal in the process' sensitivity list.

While purely synchronous clocked processes can also be described with the 'wait until' construct, asynchronous control signals can be modeled with processes with sensitivity list, only. All signals that might trigger the process execution have to be listed again, i.e. the asynchronous signals (usually reset, only) are added. The process itself consists of an if-construct, where the asynchronous signals are checked first, followed by the detection of the active clock edge.

The condition for synchronous actions has to be the last condition of the if structure because asynchronous control signals are usually treated with higher priority by the underlying hardware cells. An unconditional else path is strictly forbidden as statements that have to be processed whenever the active clock edge is not present do not have a physical representation.

It is very important not to forget any of these asynchronous signals. Otherwise simulation will differ from synthesis results, as simulation tools base their simulation on the sensitivity list and synthesis tools usually ignore the sensitivity list completely.

2.8.9 Clocked Process: Rules

```
process
begin
   wait until CLK'event and CLK='1';
   if RESET = '1' then
     -- synchronous register reset
   else
     -- combinatorics
   end if;
end process;
```
	• **Wait-form: no sensitivity list**
	• **Synchronous reset**

```
process(CLK, RST)
begin
   if (RST = `1`) then
     -- asynchronous register reset
   elsif (CLK`event and CLK=`1`) then
     -- combinatorics
   end if;
end process;
```
	• **If-form: only clock and asynchronous signals (reset) in sensitivity list**
	• **Synchronous and asynchronous reset**

Registers for all driven signals

All registers should be resetable

2.8.10 Questions

	14. Which signals in clocked processes are used for deducing registers ?
○ yes ○ no	**14.1. Temporary signals.**
○ yes ○ no	**14.2. Signals which contain an assignment.**
○ yes ○ no	**14.3. Signals which have been read.**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.8.11 Questions

	15. Which two process types are permitted in the RTL - style ?
○ yes ○ no	**15.1. Mixed and analog processes.**
○ yes ○ no	**15.2. Combinational and sequential processes.**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.8.12 Questions

	16. What causes latches to be created in the synthesized design ?
○ yes ○ no	**16.1. Concurrent IF-statements.**
○ yes ○ no	**16.2. Forgotten else-paths.**
○ yes ○ no	**16.3. Signal assignments which are not executed in all paths of an IF- or CASE-statement .**
○ yes ○ no	**16.4. Incomplete sensitivity lists.**
	17. How can unwanted latches be prevented most efficiently ?
○ yes ○ no	**17.1. By replacing IF-statements by CASE-statements.**
○ yes ○ no	**17.2. By giving default assignments to all signals which contain an assignment in one path, before the IF- or CASE-statement.**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

2.9 Subprograms

- **Functions**
  - **function name can be an operator**
  - **arbitrary number of input parameters**
  - **exactly one return value**
  - **no WAIT statement allowed**
  - **function call <=> VHDL expression**
- **Procedures**
  - **arbitrary number of parameters of any possible direction (input/output/inout)**
  - **RETURN statement optional (no return value!)**
  - **procedure call <=> VHDL statement**
- **Subprograms can be overloaded**
- **Parameters can be constants, signals, variables or files**

'93 **'impure' functions are allowed in VHDL'93**

The term subprogram is used as collective name for functions, procedures and operators. Operator definitions are treated as a special case of function definitions where the name is replaced by the operator symbol, enclosed by quotation marks ("). Please note that it is not permitted to declare new operators, i.e. it is just possible to provide a function with a different set of input parameters. This feature is called overloading (different subprograms differ by their parameters, only) and may be applied to all subprograms.

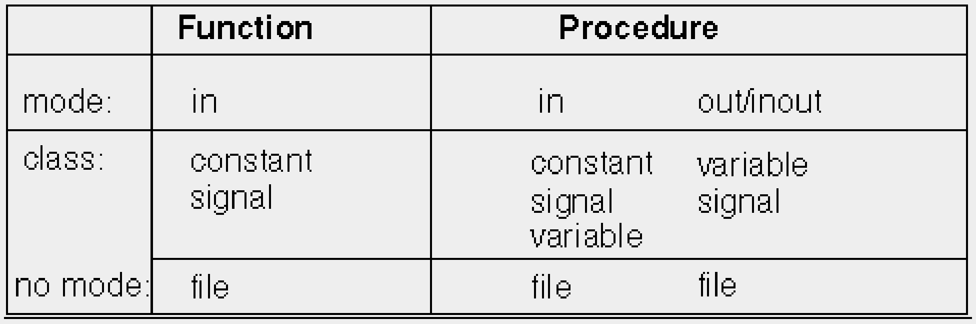
Subprogram definitions consist of the subprogram *declaration,* where the identifier and parameter list are defined, and the subprogram *body,* defining the behaviour. The *statements* in the subprogram body are executed *sequential* ly.

A function call is an expression (like 'a + b') that can exist within a statement, only. A procedure call, on the other hand, is a statement (like 'c := a + b;') and therefore it can be placed inside a process where it is executed sequentially or inside an architecture where it acts like any other concurrent statement. The return value is given after the keyword ' **return** ' which may be placed several times within a subprogram body. Please note that procedures do not have a return value!

IMPURE functions can have access to *external objects* outside of their scope. This allows returning different values, even when called with the same parameters. If, for example, a function call was used to read input values from a file, the file would have to be declared inside the function in VHDL'87. Consequently, the file would be opened with each function call and closed whenever at the end of the execution of the function. Therfore, always the first character would be read! It was impossible to read character after character via a function call in VHDL'87, unless the file itself was provided with each function call.

The IMPURE mechanism and the existance of global objects in VHDL'93 allows to open the file somewhere in the VHDL code and to read character after character by a function call. Another good example is the implementation of a random number generator which returns a different value each time it is called. Of course, the random numbers could be read from a file, then.

2.9.1 Parameters and Modes



	Function	Procedure	
mode:	in	in	out/inout
class:	constant signal	constant signal variable	variable signal
no mode:	file	file	file

- **Formal parameters (parameter declaration)**
  - **default mode: IN**
  - **default parameter class for mode IN: constant, for mode OUT/INOUT: variable**
  - **file parameters have no mode**
- **Actual parameters (subprogram call)**
  - **must match classes of formal parameter**
  - **class constant matches actual constants, signals or variables**

Function parameters are always of mode IN and can not be declared as variables

Analogous to entity ports, subprogram parameters have a certain mode associated with them. The default mode is IN, i.e. these objects can be read only. Signal attributes (' **event** *', '* **last_value** *', etc.) are only available if the formal parameter (i.e.* in the declaration part of the subprogram) *has been declared as signal. Please note that the use of the signal attributes '* **stable** *', '* **quiet** *', '* **transaction** *' and '* **delayed** *'* *is forbidden.*

Actual parameters are the ones used in the subprogram call and are treated as constants by default. The classes of the formal and actual parameters must match. So it is an error if a parameter is declared as signal and a variable is used as actual parameter in the call. Parameters of type ' **constant** ' are an exception as they match all possible types.

2.9.2 Functions

```
architecture EXAMPLE of FUNCTIONS is
  [(im)pure] function COUNT_ZEROS (A : bit_vector)
                                          return integer is
    variable ZEROS : integer;
  begin
    ZEROS := 0;
    for I in A'range loop
      if A(I) = '0' then
        ZEROS := ZEROS +1;
      end if;
    end loop;
    return ZEROS;
  end COUNT_ZEROS;

  signal WORD:     bit_vector(15  downto 0);
  signal WORD_0: integer;
  signal IS_0:       boolean;
begin
  WORD_0 <= COUNT_ZEROS(WORD);
  process
  begin
    IS_0 <= true;
    if COUNT_ZEROS("01101001") > 0 then
      IS_0 <= false;
    end if;
    wait;
  end process;
end EXAMPLE;
```

- **(Im)pure declaration optional (default: 'pure' = VHDL'87)**
- **Body split into declarative and definition part (cf. process)**
- **Unconstrained parameters possible (array size remains unspecified)**
- **Are used as expression in other VHDL statements (concurrent or sequential)**

The keyword *'* **pure** *'* can be used for "oldstyled" functions, the keyword *'* **impure** *'* declares the new VHDL'93 function type. By default, i.e. if no keyword is given, functions are declared as PURE. A *PURE* function does not have access to a *shared variable* , because *shared* *variables* are declared in the declarative part of an architecture and *PURE* functions do not have access to objects outside of their scope.

The code example shows a function that counts the number of '0's in a bit vector. Only parameters of mode ' **in** ' are allowed in function calls and are treated as ' **constant** ' by default. The size of the bit vector is not declared, i.e. a so called unconstrained formal parameter is used. Consequently, the subprogram code has to be written independently of the actual vector width. This can be done with the help of predefined attributes, like the attribute ' **range** ' in the example. This way, the for loop works for any vector size passed to the function. If a constrained array is used instead (e.g. bit_vector(15 downto 0)), the actual parameter will have to be of the same type, i.e. the type of the array elements and the size of the array must match. The directions of the array ranges may differ.

Functions may be used wherever an expression is necessary within a VHDL statement. Subprograms themselves, however, are executed sequentially like processes. Similar to a process, it is also possible to declare local variables. These variables are initialised with every function call with the leftmost element of the type declaration (boolean: false, bit: '0'). The leftmost value of integers is guaranteed to be at least -(2^31)-1, i.e. ZEROS must be initialised to 0 at the beginning of the function body. It is recommended to initialise all variables in order to enhance the clarity of the code.

2.9.3 Procedures

```
architecture EXAMPLE of PROCEDURES is
  procedure COUNT_ZEROS
                        (A: in bit_vector;
                         signal Q: out integer) is
    variable ZEROS : integer;
  begin
    ZEROS := 0;
    for I in A'range loop
      if A(I) = '0' then
        ZEROS := ZEROS +1;
      end if;
    end loop;
    Q <= ZEROS;

  end COUNT_ZEROS;

  signal COUNT: integer;
  signal IS_0:     boolean;
begin
  process
  begin
    IS_0 <= true;
    COUNT_ZEROS("01101001", COUNT);
    wait for 0 ns;
    if COUNT > 0 then
      IS_0 <= false;
    end if;
    wait;
  end process;
end EXAMPLE;
```

- **No return value**
- **Parameter values may be updated (mode out/inout)**
- **Body split into declarative and definition part (cf. process)**
- **Unconstrained parameters possible (array size remains unspecified)**
- **Are used as VHDL statements (concurrent or sequential)**

Procedures, in contrast to functions, are used like any other statement in VHDL. Consequently, they do not have a return value, although the keyword ' **return** ' may be used to indicate the termination of the subprogram. Depending on their position within the VHDL code, either in an architecture or in a process, the procedure as a whole is executed concurrently or sequentially, respectively. The code within all subprograms is always executed sequentially.

Procedures can feed back results to their environment via an arbitrary number of output parameters. As the default mode of a parameter is ' **in** ', the keyword ' **out** ' or ' **inout** ' is necessary to declare output signals/variables. Per default, output parameters are of the class variable.

The VHDL compiler will report an error message if a function declared with a signal as parameter is called with a variable and vice versa.

Since the class of the parameters have to match, one might think of overloading a procedure, i.e. by writing procedures that differ in the class declaration of the parameters and the corresponding assignment operators, only. Yet this is **not** possible because the parameter class is ignored when the appropriate subprogram is selected!

The example procedure is the equivalent of the previously presented function for counting the '0' elements within a bit_vector. Again, all internal variables should be initialized because subprograms do not store variable values and initialize them with type'left at each call instead.

2.10 Subprogram Declaration and Overloading

- Subprograms may be declared/defined in any declaration part
  - package
  - entity
  - architecture
  - process
  - subprogram
- Overloading of subprograms possible
  - identical name
  - different parameters
  - works with any kind of subprogram
- During compilation/runtime that subprogram is called whose formal parameter match the provided actuals

Subprograms may be declared/defined in any declarative part of a VHDL object. The actual definition of the behaviour may also be separated from the declaration, which is often the case when packages are split into package and package body. The usual object visibility rules apply, e.g. a subprogram which is declared in a package may be used in all units that reference this package. Subprograms that are declared within another subprogram are available within this "parent" subprogram, only.

It is legal to declare subprograms with identical names, as long as they are distinguishable by the compiler. Thus, if the two subprogram names match, the parameter set/return values have to differ. This is called overloading and is allowed for all subprograms. It is especially useful when applied to operators, which can be seen as functions with a special name. This allows, for example, to use the conventional '+' symbol for the addition of integer values and, likewise, with bit vectors that should be interpreted as numbers.

2.10.1 Overloading Example

```
procedure READ ( L         : inout line;
                 VALUE : out    character;
                 GOOD : out     boolean );

procedure READ (L          : inout line;
                VALUE : out    character );

procedure READ ( L         : inout line;
                 VALUE : out    integer;
                 GOOD : out    boolean );

procedure READ ( L          : inout line;
                  VALUE : out integer );

. . .
```

- **Input routines from TEXTIO package**
  - **extract different datatypes from a line**
  - **identical names**
  - **different number of parameters**
  - **different parameter types**

The file I/O procedures read, write, readline, writeline and the line types are predefined in the standard TEXTIO package. Several overloaded read/write procedures for the standard data types are declared. They extract a value of the desired type from a file line. The line itself is modified, as indicated by the mode declaration ' **inout** ', i.e. several values may be read from a single line. During compilation that procedure is chosen whose formal parameters match the actual parameters in the procedure call.

2.10.2 Overloading - Illegal Redeclarations

```
package P_EXAMPLE is
  -- 1 --
  procedure TEST (A: bit;
                  variable X_VAR: out
integer);
  -- 2 --
  procedure TEST (B: bit;
                  variable X_VAR: out
integer);
  -- 3 --
  procedure TEST (variable X_VAR: out
integer;
                  A: bit);
  -- 4 --
  procedure TEST (A: bit;
                  variable X_VAR: in
integer);
  -- 5--
  procedure TEST (A: bit;
                  signal X_SIG: out
integer);
  -- 6--
  procedure TEST (A: bit;
                  signal X_SIG: out
integer;
                  FOO: boolean := false);
end P_EXAMPLE;
```

- **VHDL compiler ignore the following differences in parameter declarations:**
  - **names of formal parameters (2)**
  - **order of parameter declaration (3)**
  - **mode/class of formal parameter (4/5)**
- **Default values may be assigned to input parameters**

Default parameters should not be used in synthesizable code

It is not possible to declare two subprograms which have the same number of parameters and the same types but *different names, modes* or classes. The compiler will report an error message similar to "illegal redeclaration". The example package P_EXAMPLE will not compile successfully unless the declarations 2-5 are removed, e.g. by marking them as comments. These four procedures have the same name as the first one and all of them need a bit and an integer parameter.

If a subprogram is necessary to deal with signal and variable parameters, it is possible to declare an additional dummy input parameter and assign it a default value. This way, the declared subprograms differ in the number of parameter and a redeclaration error is avoided.

2.10.3 Overloading - Ambiguity

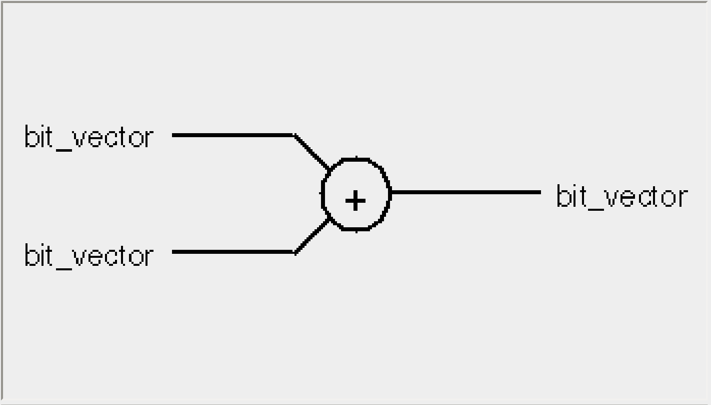
```
-- Declarations 2-4 need to be removed from the
-- package P_EXAMPLE in order to compile!
use work.P_EXAMPLE.all;

entity AMBIGUITY is
end AMBIGUITY;

architecture EXAMPLE of AMBIGUITY is
  signal A:        bit;
  signal X_SIG: integer;
begin
  process
    variable X_VAR: integer;
  begin
    -- 1 --
    TEST(A, X_VAR);
    -- 2--
    TEST(A, X_SIG);
    -- 3--
    TEST(A => A, X_SIG => X_SIG);
    -- 4 --
    TEST(A => A, X_VAR => X_VAR);
    wait;
  end process;
end EXAMPLE;
```

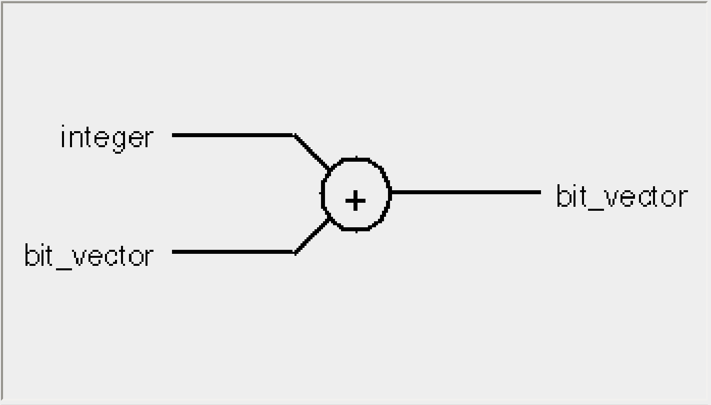
- **Ambiguous calls occur, if it is not posible to find a unique subprogram by**
  - **name**
  - **number of formal parameters**
  - **types and order of actual parameters (1/2)**
  - **names of formal parameters (named association, only)**

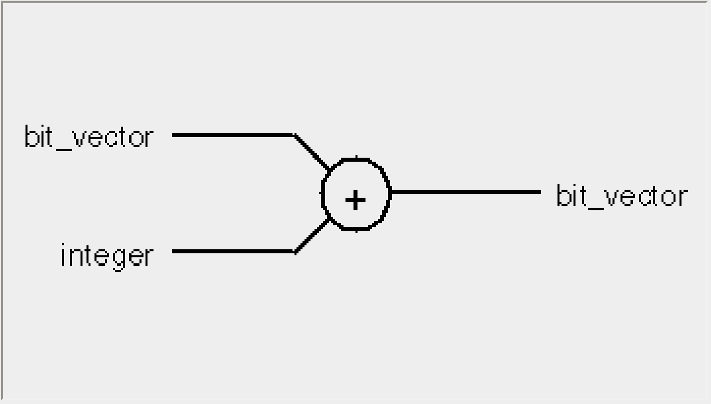
For ambigues overloading it is necessary to use the named association mechanism to map the actual parameters to the formal ones that were declared. Otherwise, the compiler will try to find a unique subprogram of the given name which has the same number and type of parameters as in the call. If this fails, an error message about an ambiguous expression will be generated (TEST statements 1, 2).

Input parameters with a default value assigned to them need not be present in the subprogram call. Their use is not recommended, however, as some synthesis tools map absent parameters to the default value of the data type (type'left) which may lead to a different behaviour, if the parameter is actually used in the body.

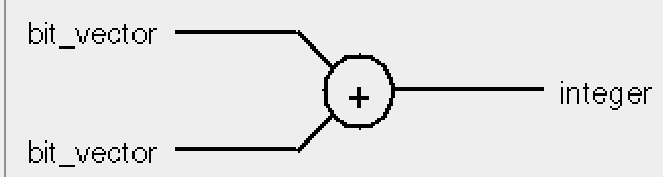
2.10.4 Operator Overloading







- **Similar to function declarations**
  - **name = existing operator symbol, enclosed in quotation marks**
  - **operand left/right of operator are mapped to first/second parameter**
- **Extends operator functionality to new data types**
- **Operator call according to the individual context**
- **Definition of new operators is not allowed**



Arithmetic operations with 'bit_vector' are not defined

All standard VHDL operators can be overloaded but is not allowed to define new operators. Operator declarations are equivalent to function declarations apart from the name which must be placed in quotation marks ("). The number of parameters is also fixed and can not be modified. In case of binary operators, i.e. operators with two operands, the left/right operand are mapped to the left/rightmost parameter, respectively.

2.10.5 Operator Overloading - Example

```
package P_BIT_ARITH is
  function "+" (L: bit_vector; R: bit_vector) return bit_vector; -- 1
  function "+" (L: integer; R: bit_vector) return bit_vector;     -- 2
  function "+" (L: bit_vector; R: integer) return bit_vector;     -- 3
  function "+" (L: bit_vector; R: bit_vector) return integer;     -- 4
end P_BIT_ARITH;

use work.P_BIT_ARITH.all;
entity OVERLOADED is
  port(A_VEC, B_VEC: in bit_vector(3 downto 0);
        A_INT, B_INT: in integer range 0 to 15;
        Q_VEC: out bit_vector(3 downto 0);
        Q_INT: out integer range 0 to 15);
end OVERLOADED;

architecture EXAMPLE of OVERLOADED is
begin
  Q_VEC <= A_VEC + B_VEC; -- a
  Q_VEC <= A_INT + B_VEC; -- b
  Q_VEC <= A_VEC + B_INT; -- c
  Q_VEC <= A_INT + B_INT; -- d
  Q_INT <= A_VEC + B_VEC; -- e
  Q_INT <= A_INT + B_INT; -- f
end EXAMPLE;
```

The code example shows just the operator declarations and their use. The behaviour has to be defined in a package body, if the design is to be simulated. During compilation, the VHDL compiler searches its list of operator declarations for a parameter list with matching data types. This way, the function bodies of the operator declarations (1)-(3) will be used in the signal assignments (a)-(c). Signal assignment (d) will result in an error message as the addition of two integer values to obtain a bit_vector has not been defined, yet. Assignment (e) matches the parameter list of declaration (4) and the last assignment will use the standard VHDL operator.

Arithmetic operations based on the data type 'bit' are defined in the standard package 'numeric_bit' (library IEEE). In practice, this data type should be avoided, however, as standard packages are available that define more powerful bit vector types and the corresponding operations.

2.10.6 Questions

	18. What is subprogram overloading ?
○ yes ○ no	**18.1. Same parameters but different function name.**
○ yes ○ no	**18.2. The right function is chosen from the context.**
○ yes ○ no	**18.3. Same names but different parameters.**
	19. What is important for the use of overloading?
○ yes ○ no	**19.1. Subprograms with the same name should do the same.**
○ yes ○ no	**19.2. A good subprogram documentation in the package header.**

submit reset

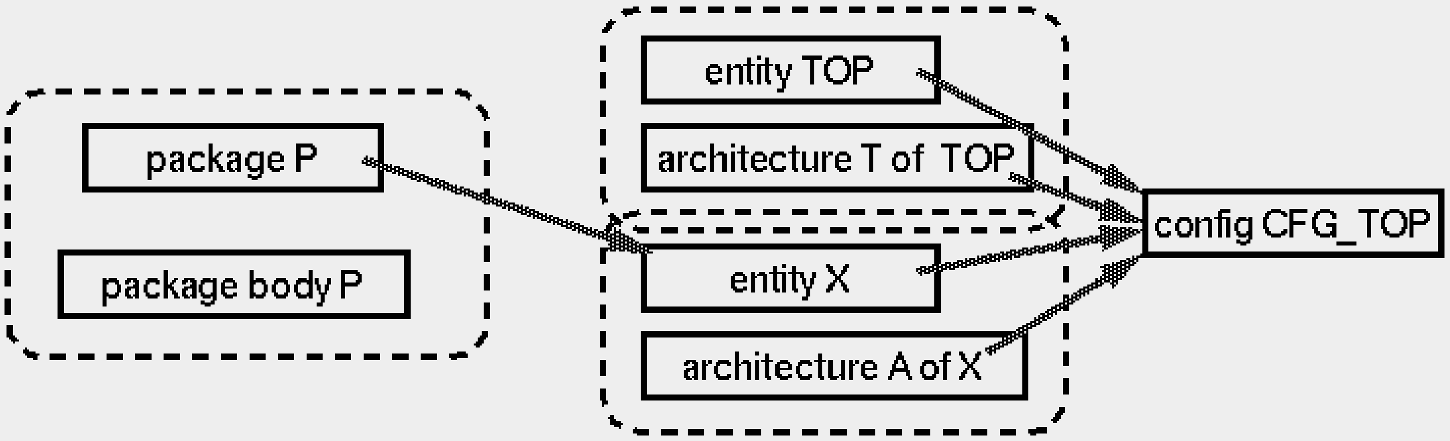
Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

3. Simulation

- **Sequence of Compilation**
- **Simulation Flow**
- **Process Execution**
- **Delay models**
- **Testbenches**
- **File IO**

3.1 Sequence of Compilation

- **Main components are analysed before side- or sub-components**
  - ***entity before architecture***
  - ***package before package body***
- **The component which is referred to another one has to be analysed first**
  - ***package before entity/architecture***
  - ***configuration after entity/architecture***

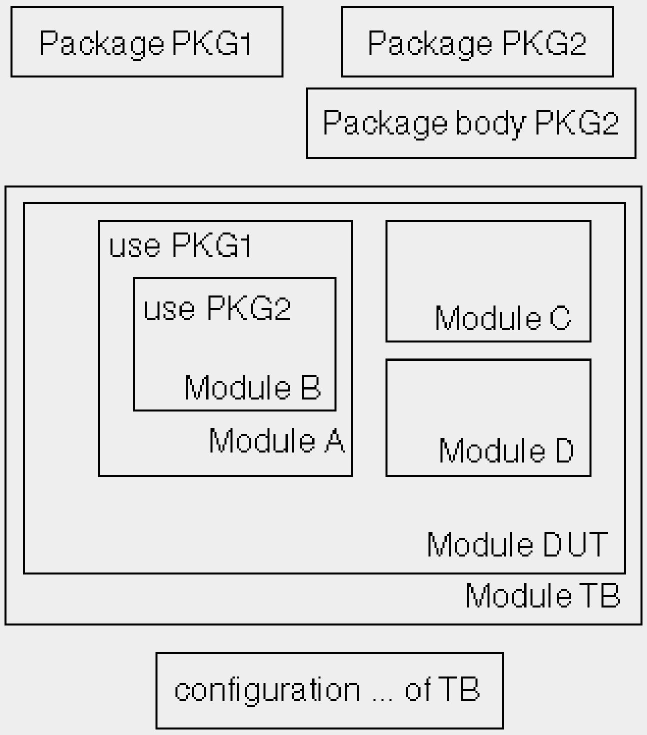


The sequence of compilation is determined by the interdependence of the single parts. Primary units have to be compiled before secondary ones, because secondary units need some information from their corresponding primary unit for the compliation process (e.g. entity ports are available as architecture signals). Thus, the entity is compiled before its architecture(s), a package header before the package body and the configuration at the end.

General rule: When a module is referenced, it must be compiled before.

Example

Structure of a design



- **Which files will have to be recompiled if there is a change in the following files? (only minor changes, i.e. comments)**
    - **Entity of module C**
    - **Architecture of module D**
    - **Package PKG1**
    - **Package PKG2**
    - **Package body PKG2**
    - **Architecture of TB**
    - **Configuration of TB**

The design DUT which is to be simulated consists of several modules (A,B,C,D). The packages PKG1 and PKG2 are referenced in module A and B, respectively. The package PKG 1 is split into the package header containing declarations of subprograms, data types and constants and a package body with the corresponding definitions.

The following order is suitable for the initial compilation: PKG1, PKG2, body of PKG2, module D, C, B, A, DUT, TB (entity first) and finially the testbench configuration. As the configuration creates the simulatable object, is has to recompiled whenever something is changed.

If entity C is changed, the corresponding architecture and the configuation have to be recompiled. A modification of D's architecture does not require any additional recompilations apart from the configuration. A recompilation of PKG1 leads to recompilations of the complete module A and the configuration.

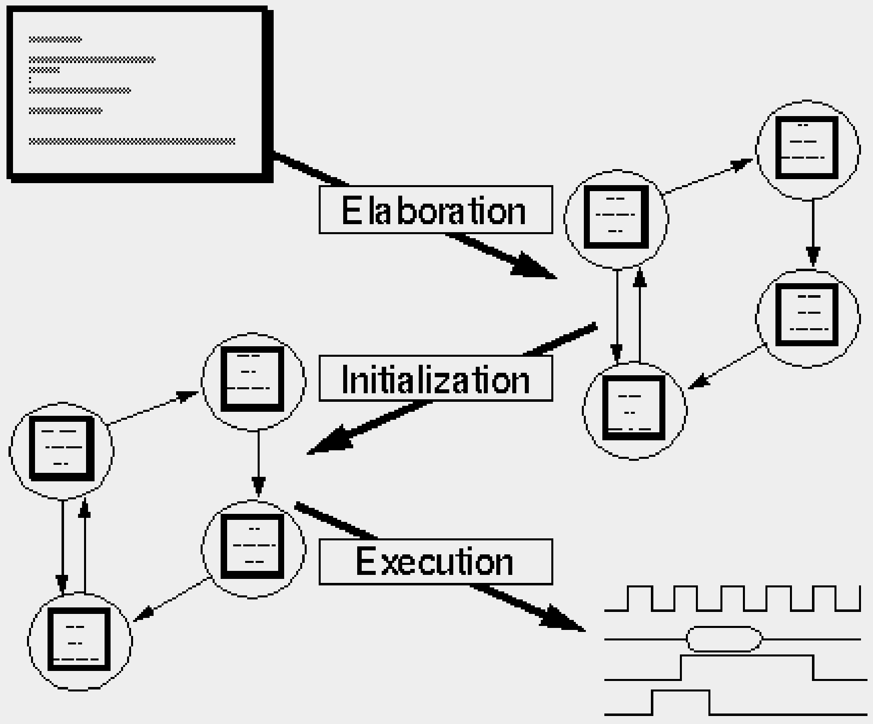
Changes in PKG2 imply that the package body and module B have to be recompiled. If the changes are restricted to the package body, only the configuration, as always, remains to be updated. The same applies to modified testbench (TB) architectures and changes to the configuration itself.

Please note, that only minor modifications, e.g. in the VHDL comments where considered here. If entity ports are changed, for example, a simple recompilation is not enough and the VHDL code of the modules on all hierarchy levels might have to be adjusted accordingly. The interface mismatch is detected when compiling the configuration.

Changes in ... recompile files ...

• **Entity of module C** ❍	• **Package PKG2** ❍
• **Architecture of module D** ❍	• **Package body PKG2** ❍
• **Package PKG1** ❍	• **Architecture of TB** ❍
• **Configuration of TB** ❍	

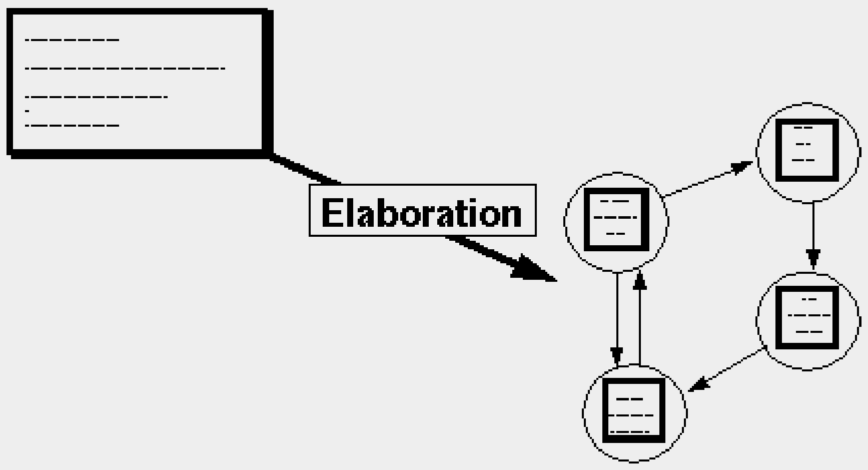
3.2 Simulation Flow



- **Design elaboration**
  - **Specified elements are created**
- **Signal initialisation**
  - **Starting values are assigned**
- **Simulation is executed on command**

The simulation of a VHDL model operates in three phases. First, the simulation model is created in the elaboration phase. In the initialisation phase a starting value is assigned to all signal. The model itself is executed in the execution phase.

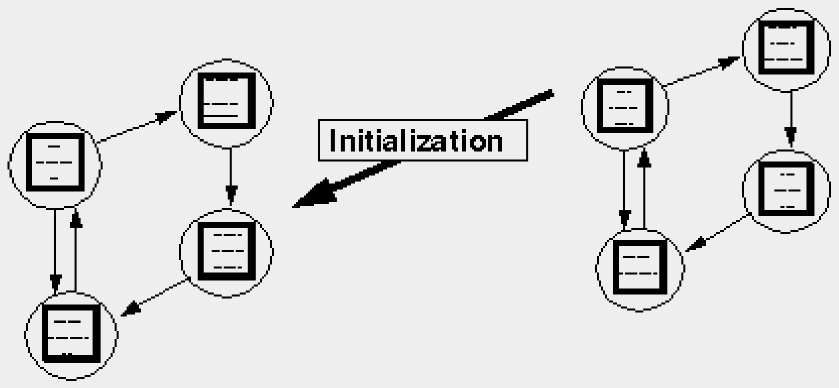
3.2.1 Elaboration



- **During elaboration design elements are created**
- **All design objects are elaborated before the simulation**
- **Except "for loop"- variables and objects which are defined in subprograms**

The simulation of a VHDL model operates in three phases. First, the simulation model is created in the elaboration phase. The processes and concurrent statements of the whole design are combined in a communication model. This model lists, which process can be activated by which one, i.e. some sort of netlist is created. All objects are converted to an executable form residing in the simulator memory. Loop variables and subprograms are the only exception as they are elaborated dynamically during the execution of the simulation.

3.2.2 Initialization



type std_ulogic is (`u` ,`x`,`0`.....) -- First value is `u` !!!

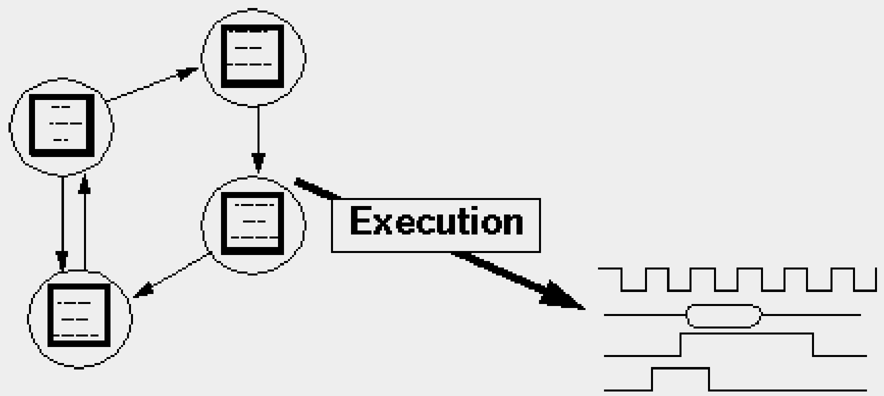
signal clock: std_ulogic := `0`;
signal reset: std_ulogic;

- **Initial values:**
  - **Start values from declaration**

    OR
  - **First value from type definition (type`left)**
- **Every process is executed until it is suspended**
  - **...without signal values being updated**

The initial values of all signals are assigned in the initialization phase. Either the initial value specified in the signal declaration or the first value in the type definition (="data type'left") is used. In the case of the type STD_ULOGIC based types this is an 'u' for uninitialized. Hence the designer can deduce from an 'u' in the simulation waveform, that there has never been assigned a value to the corresponding signal. Signal values do not return to 'u' except an 'u' is directly assigned. At the end of the initialization phase every process is executed once until it is suspended. The signal values, however, are not updated.

3.2.3 Execution



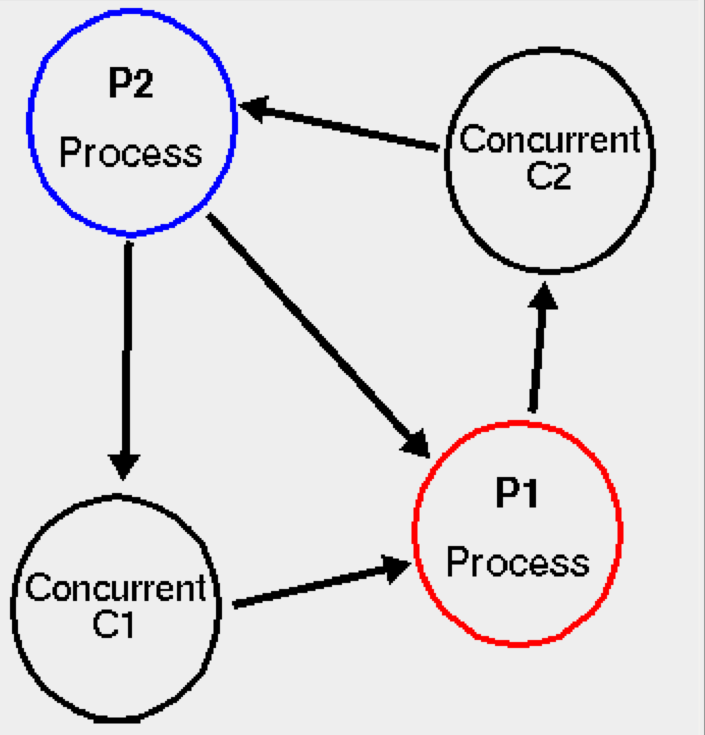
- **Simulation is actually executed**
- **Signal values are evaluated**

The actual simulation of the design behaviour takes place in the execution phase. By means of testbench processes, the VHDL model is provided with stimuli. The individual signals of the model can then be viewed and checked in the waveform window (stimuli and responses of the model). The actual responses can be compared automatically with the expected values by adequate VHDL statements, as well.
For instance you can compare actual and expected responses at time OCCURRING_TIME by an assertion in an 'if' statement:

```
if now=OCCURRING_TIME then
  assert EXPECTED_RESPONSE=RECEIVED_RESPONSE
  report "unexpected behaviour"
  severity error;
end if;
```

3.3 Process Execution

```
architecture A of E is
begin
  P1 : process
  begin
    -- sequential statements
  end process P1;

  -- C2: concurrent statements

  P2 : process
  begin
    -- sequential statements
  end process P2;

  -- C1: concurrent statements

end A;
```



An architecture can contain processes and concurrent statements which are all active in parallel. Statements within processes are sequential ones and are executed one after another.

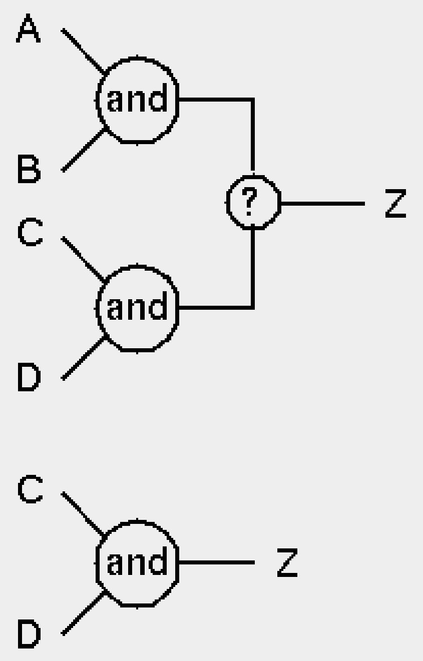
The connection of the parallel parts is established via signals and sensitivity lists. Concurrent statements can be interpreted as functionally equivalent processes with a sensitivity list including all those values that are going to be read in this process. Concurrent statements have to be transferred to sequential ones of course (conditional signal assignment -> if statement, selected signal assignment -> case statement).

If, for example, the process P1 was triggered, e.g. by a clock edge, its statements are executed one after another. This way it is possible to execute parts of the code after an active edge has occured. Let us assume that a couple of signals were modified. Their value is modified after the process execution has finished. According to the schematic, these updated values will trigger C1 and P1 which will trigger P1 and C2 in turn, and so on. The execution of the statements continues until a stable state is reached, i.e. no events are generated anymore.

3.3.1 Concurrent versus Sequential Execution

```
architecture CONCURRENT of MULTIPLE is
        signal A, B, C, D : std_ulogic;
        signal Z : std_logic
begin
        Z <= A and B;
        Z <= C and D;
end CONCURENT;
```

```
architecture SEQUENTIAL of MULTIPLE is
         signal Z, A, B, C, D : std_ulogic;
begin
        process (A, B, C, D)
        begin
              Z <= A and B;
              Z <= C and D;
          end process;
end SEQUENTIAL;
```



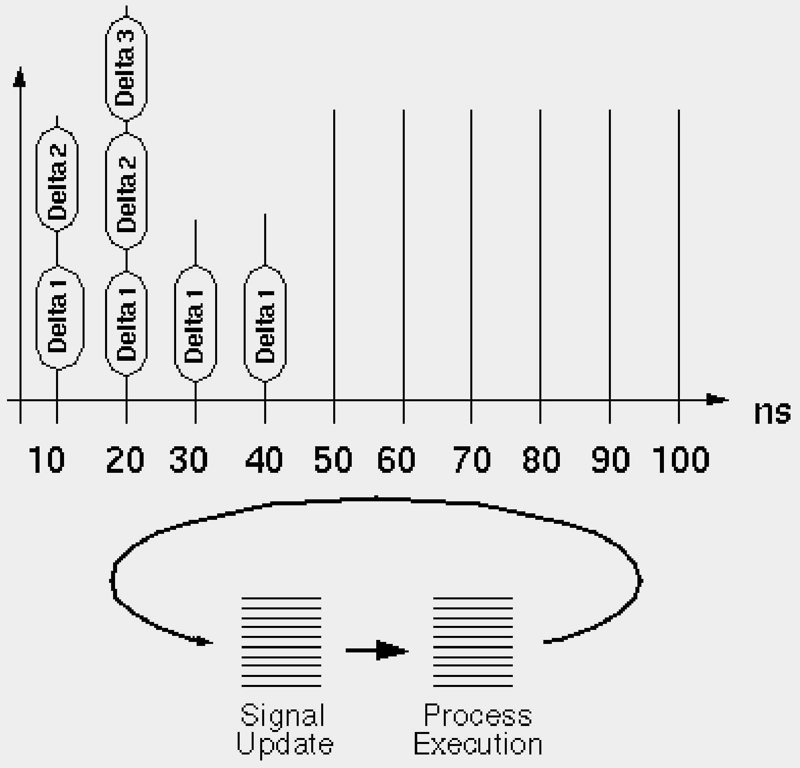
If the same two signal assignments appear in the VHDL code, once as concurrent statement in the architecture and once in a process, the result will differ substantially: In the first case, two parallel signal assignments are actually made to the signal. This is only allowed for resolved types, for which a resolution functions is present to decide which value is actually driven. In the second case, the first assignment is executed and its result is stored. Afterwards, this result is overwritten by another assignment, so that only the last signal assignment is carried out when updating the signal.

3.3.2 Signal Update

- **Signals have a past value, a current value and a future value**
  - **future value used within the simulator core, only**
  - **past value ≠ current value: event**
- **Signal values are updated at the end of a process execution:**
  the old current value of a signal is overwritten by the future value
- **Several process calls at one single moment of the simulation are possible**

The signal update mechanism is essential for a VHDL simulator. Signals possess a past, a current and a future value within the simulators signal management functions. Signal assignments in a process always assign the value to the future value of the signal. The future value is copied to the current value in the signal update phase after the process execution is finished i.e. the process is suspended.

3.3.3 Delta Cycles (1)



- **One moment of simulation**
- **One loop cycle = ``delta cycle"**
- **Delta time is orthogonal to simulation time**
- **Signals are updated**
- **All processes are initiated**
    - **signal assignments are stored**
- **New signal assignments**
    - **to execute**

further processes

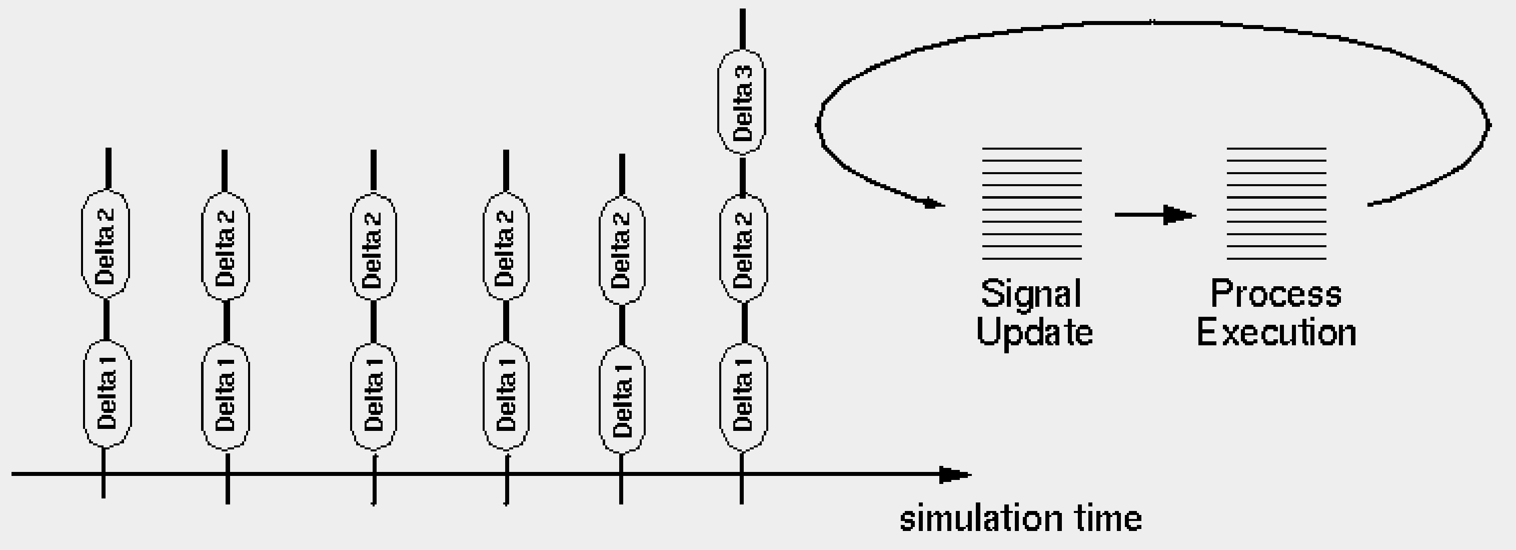
A simulation cycle always consists of a signal update and a phase process execution. Several of these so called delta cycles might have to be carried out at one point simulation time in order to achieve a stable state of the system. The number of delta cycles does not affect the time used in the simulation! It just affects the time that is necessary to carry out the simulation.

At the beginning, all signals are updated and a list of all processes that are triggered by the signal changes is created. All the processes of this list are executed one after another in delta cycle 1. When the execution is finished, a signal update will be carried out. Again, a new process list is created containing those processes whose sensitivity list signal values have changed.

This continues until the process list remains empty, that means no further processes are triggered by the signal events. Now, statements which induce a real time step ('wait for ...', '... after ...') are carried out and the simulation time advances for the specified amount of time. Afterwards the delta cycles for the new simulation time are started. The delta cycles are lined up orthogonally to the simulation time, i.e. several delta cycles can be carried out at one simulation time.

3.3.4 Delta Cycles (2)

- **Several delta cycles at any moment of the simulation**



The delta cycles are orthogonal to the simulation time. So at a fixed simulation time several delta cycles can be executed.

3.3.5 Delta Cycles - Example

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity DELTA is
  port (A, B :  in std_ulogic;
          Y, Z :   out std_ulogic);
end DELTA;

architecture EXAMPLE of DELTA is
  signal X : std_ulogic;
begin
  process (A, B, X)
  begin
    Y <= A;
    X <= B;
    Z <= X;
  end process;
end EXAMPLE;
```

Event on B (first delta cycle)

future value of

Y receives the **current value of A** (no change)

X receives the **current value of B (new value)**

Z receives the **current value of X** (no change)

signal update

Event on X (second delta cycle)

future value of

Y receives the **current value of A** (no change)

X receives the **current value of B** (no change)

Z receives the **current value of X (new value)**

signal update

`

No further events on A, B, X

Let us assume that the value of signal B changes. The process of architecture A is triggered by this event on B and is activated for the first time. The future value of X is given the current value of B. At the end of the process, the future value of X is transferred to the current value. This change of value on X results in an event that calls the process for the second time. Now, the current value of X is written to the future value of Z (B and consequently X remain the same). At the end of the process the signal update is carried out once again, so the future value of Z is transferred to the current value of Z. As no change has appeared in the current values of the signals A, B, or X that are listed in the sensitivity list, the process will not be called again. Both signals X and Z have obtained the value from B this way.

This example is for demonstrative purposes, only. The intermediate signal X conceals the functionality of the process and would not be used in practice. Generally, variables, which are not subject to the update mechanism, should be used instead.

3.3.6 Process Behaviour

```
process  (A, B)
begin
    if (A=B) then
      Z <= \`1\`;
    else
       Z <= \`0\`;
    end if;
end process;
```
	```
	process
	begin
	    if (A=B) then
	       Z <= \`1\`;
	    else
	       Z <= \`0\`;
	    end if;
	     wait on A, B;
	end process;
	```
		• **The process is an endless loop**
		• **It is stopped by a wait-statement**
		• **The sensitivity list is equivalent to a wait-statement**
		• **A process with a sensitivity list must not contain any wait statements**

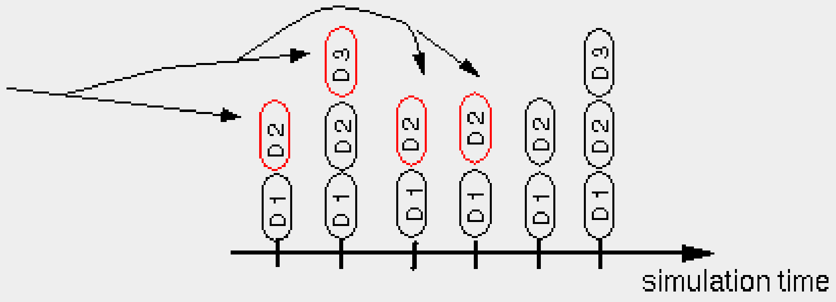
Basically, a process has to be considered as an endless loop. The continuous process execution can be interrupted via wait statements. The use of a sensitivity list is equivalent to a WAIT ON-statement. If a sensitivity list is present, WAIT statements must not appear in the process.

3.3.7 Postponed Processes

```
postponed process
begin
    Z <= `0` ;       -- wrong
    wait for 0 ns;
    Z <= `1` ;       -- wrong
    wait on A, B;    -- wrong
end process;
```

- **Processes which are executed in the last delta cycle of a certain moment**
- **The following is not permitted:**
    - **Wait-statements of the time 0**
    - **Signal assignments without delay (for 0 ns)**

```
postponed
 process (A, B)
begin
  if (A=B) then
    Z <=  `1`  after 5 ns;
  else
    Z <= `0`  after 4 ns;
  end if;
end process;
```

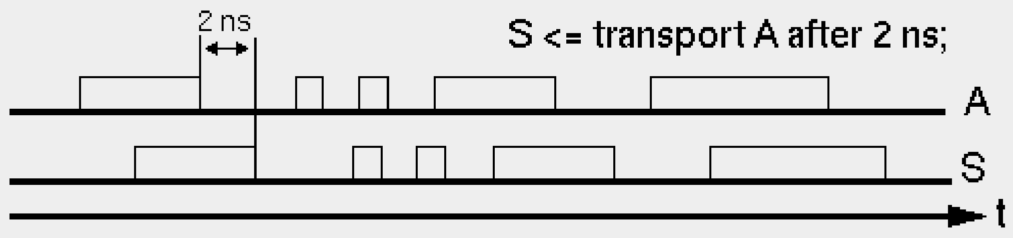
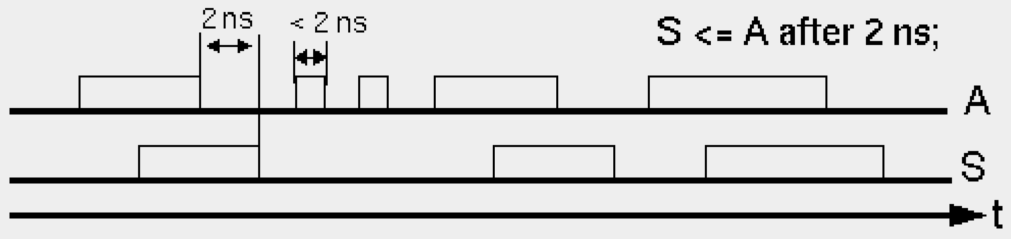


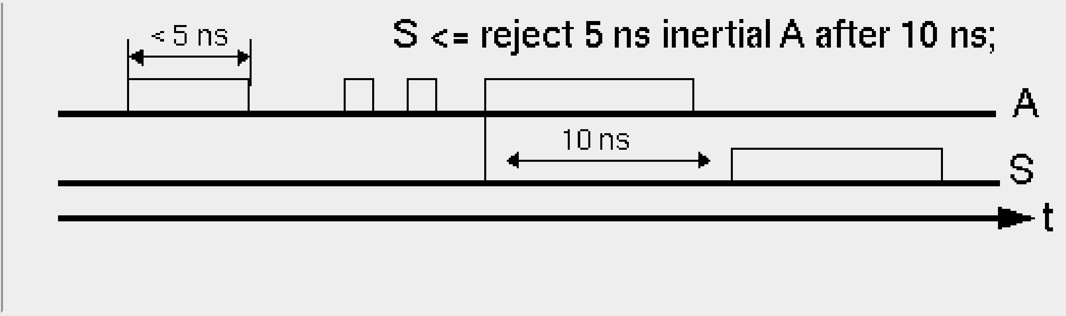
'93 Postponed processes are a new feature of VHDL'93

Postponed processes are always carried out in the last delta cycle. This means that this process can access already stable signals at this point of simulation time. In postponed processes, WAIT statements with 0 ns and signal assignments without delay are not permitted.

Please note that postponed processes can only be used in simulation, not in synthesis.

3.4 Delay Models

	**Transport delay:** *models the current flow through a wire (everything is transferred)*
	**Inertial delay: (default delay mechanism)** *models spike-proof behaviour => a value is transferred only if it is active for at least 2 ns*
	Inertial delay with pulse rejection limit *models spike-*



proof behaviour => a value is transferred only if it is active for at least 5 ns

There are two different delay models in VHDL: transport and inertial, which is used per default.

In the transport delay model, everything is transferred via the signal, as can be seen in the upper example where signal A is an exact copy of signal S, delayed by 2 ns. Transport delay models signal transfers by wire with pure propagation delay, thus spikes are not filtered out.

When using inertial delay, signal transitions are only transferred when the new value remains constant for a minimum amount of time, thus spikes are suppressed.

"S <= A after 2 ns" filters out all spikes of less than 2 ns and delays signal values which remain constant for a longer period of time for 2 ns.
"S <= reject 5 ns inertial A after 10 ns" requires a minimum pulse width of 5 ns and copies all other signal values from A to S with 10 ns delay.

Inertial delay is characteristic of switching circuits. Spikes which are shorter than the necessary specific switching time of the circuit have no effect on the succeeding switch and will not be transmitted.

3.4.1 Projected Output Waveforms (LRM)

- **Transport and inertial delay:**
  - **(1) All old transactions that are projected to occur at or after the time at which the earliest new transaction is projected to occur are deleted from the projected output waveform.**
  - **(2) The new transactions are then appended to the projected output waveform in the order of their projected occurrence.**
- **For inertial delay projected output waveform is further modified:**
  - **(3) All of the new transactions are marked.**
  - **(4) An old transaction is marked if the time at which it is projected to occur is less than the time at which the first new transaction is projected to occur minus the pulse rejection limit.**
  - **(5) For each remaining unmarked, old transaction, the old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction**

- **(6) The transaction that determines the current value of the driver is marked.**
- **(7) All unmarked transactions (all of which are old transactions) are deleted from the projected output waveform.**

The definition of the delay mechanism is quoted from the VHDL language reference manual. As conclusion, all signal assignments can be brought into the following format:
T <= reject TIME_1 inertial VALUE after TIME_2;

The following assignments are equivalent:

```
T <=                         VALUE after TIME_1; -- (default inertial delay)
T <=                inertial VALUE after TIME_1;
T <= reject TIME_1 inertial VALUE after TIME_1;
```

Also equivalent are:

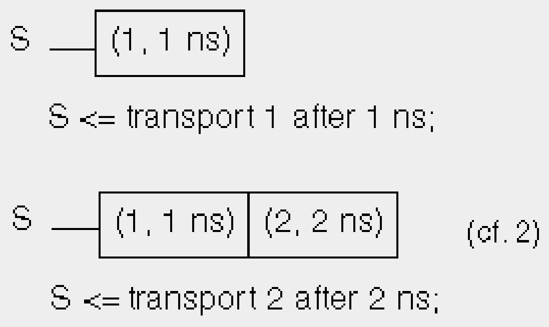
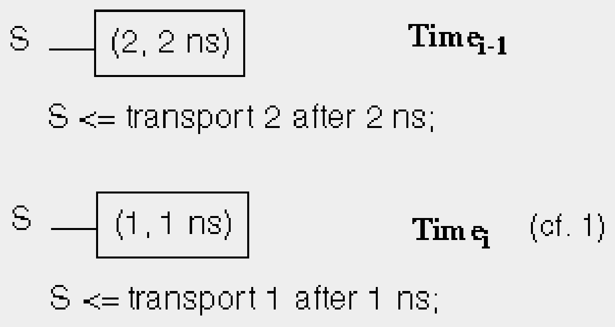
```
T <= transport          VALUE after TIME_1;
T <= reject 0 ns inertial VALUE after TIME_1;
```

This is because a pulse rejection limit of 0 ns makes all transaction being marked (step 3) and thus none of the transactions will be deleted in step 7. Consequently steps 3 to 7 have no effect and the delay model is actually equivalent to transport delay.

Furthermore "T <= VALUE" is just a shortcut for "T <= VALUE after 0 ns".

3.4.2 Transport Delay (1)

signal S : integer := 0;
process
begin
    S <= transport 1 after 1 ns;
    S <= transport 2 after 2 ns;
    wait;
end process;
	signal S : integer := 0;
	process
	begin
	    S <= transport 2 after 2 ns;
	    S <= transport 1 after 1 ns;
	    wait;
	end process;

 S (1, 1 ns) S <= transport 1 after 1 ns; S (1, 1 ns) (2, 2 ns) (cf. 2) S <= transport 2 after 2 ns;	 S (2, 2 ns) Time$_{i-1}$ S <= transport 2 after 2 ns; S (1, 1 ns) Time$_i$ (cf. 1) S <= transport 1 after 1 ns;

A signal driver manages value/time pairs
It is not possible to assign a new value for Timei < Timei-1

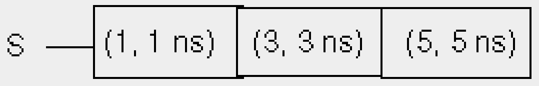
Within a signal driver, the actual values are always associated with an activation time. The initial value/time (0, 0 ns) pair for the integer signal of the example is left out in the figures as this pair remains unaffected at any time.

Whenever a signal assignments leads to a new pair, it must be decided, whether the time in the value/time pair is chronologically after the time of the last list entry or not. The first signal assignment is the most simple one as it is just appended to the list. The same applies to pairs that occur later on in time (left example; step 2). Otherwise the new value/time pair will be inserted chronologically after the pair with the next previous time specification and all succeeding pairs will be deleted (right example; step 1).

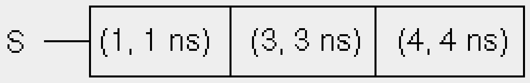
It is not possible to insert a new value/time pair before an already existing one without deleting all succeeding pairs!

3.4.3 Transport Delay (2)

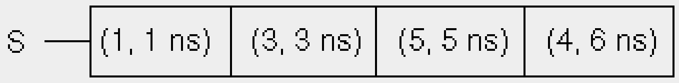
signal S : integer := 0;
process
begin
  S <= transport 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;
  S <= transport 4 after 4 ns;
wait;
end process;
	signal S : integer := 0;
	process
	begin
	  S <= transport 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;
	  S <= transport 4 after 6 ns;
	wait;
	end process;



S <= transport 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;



S <= transport 4 after 4ns;



S <= transport 4 after 6 ns;

New pairs are either appended to the list or overwrite the remaining elements

After the first signal assignment the driver of S contains three value/time pairs.

The signal assignment occuring after 4 ns (left example: "S <= transport 4 after 4 ns") is prior to the last assignment that was specified before. Thus, the last entry is overwritten: In step 1, the last list element is deleted; in step 2, the new pair is added. If the list had been longer, then the following list entries would have been deleted too.

In the second example ("S <= transport 4 after 6 ns"), the pair is attached to the list, because the time entry of this pair follows chronologically the time entry of the last list element (nothing to do in step 1, only step 2). The time entry ti of the additional value/time pair decides on whether the list is overwritten and the following entries have to be deleted (ti-1 >= ti), or whether the pair will be attached to the list (ti-1 < ti).

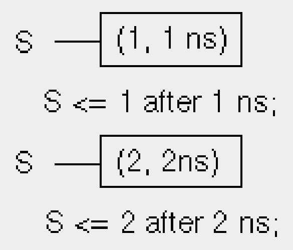
3.4.4 Inertial Delay (1)

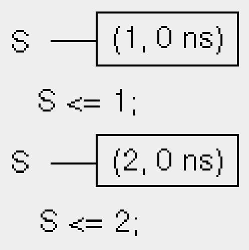
```
signal S : integer := 0;
process
begin
    S <= 1 after 1 ns;
    S <= 2 after 2 ns;
    wait;
end process;
```

```
signal S : integer := 0;
process
begin
    S <= 1;
    S <= 2;
    wait;
end process;
```

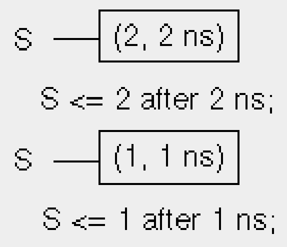
```
signal S : integer := 0;
process
begin
    S <= 2 after 2 ns;
    S <= 1 after 1 ns;
    wait;
end process;
```

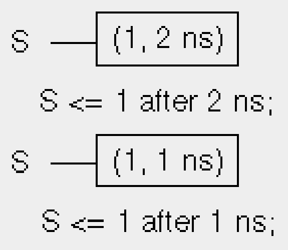
```
signal S : integer := 0;
process
begin
    S <= 1 after 2 ns;
    S <= 1 after 1 ns;
    wait;
end process;
```





```
S <= 1; -- equivalent to
S <= 1 after 0 ns;
```



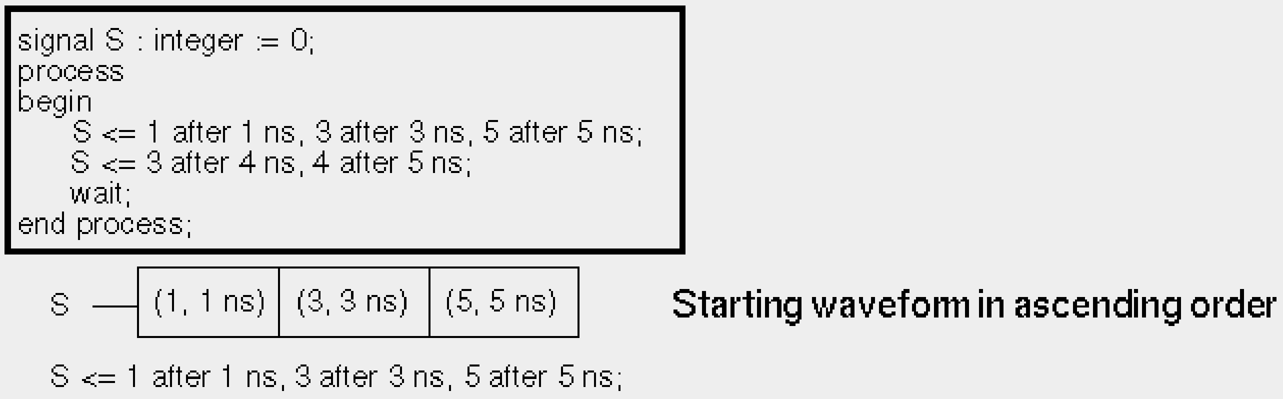


The last assignment to a signal in a process takes effect

In the left example, the pulse rejection limit of the signal assignment "S <= 2 after 2 ns" equals 2 ns. Therefore, all transactions are marked, whose recurrence time is smaller than 0 ns (2 ns minus pulse rejection time; step 4). Accordingly the pair (1, 1 ns) is not marked and deleted in the final step (step 7).

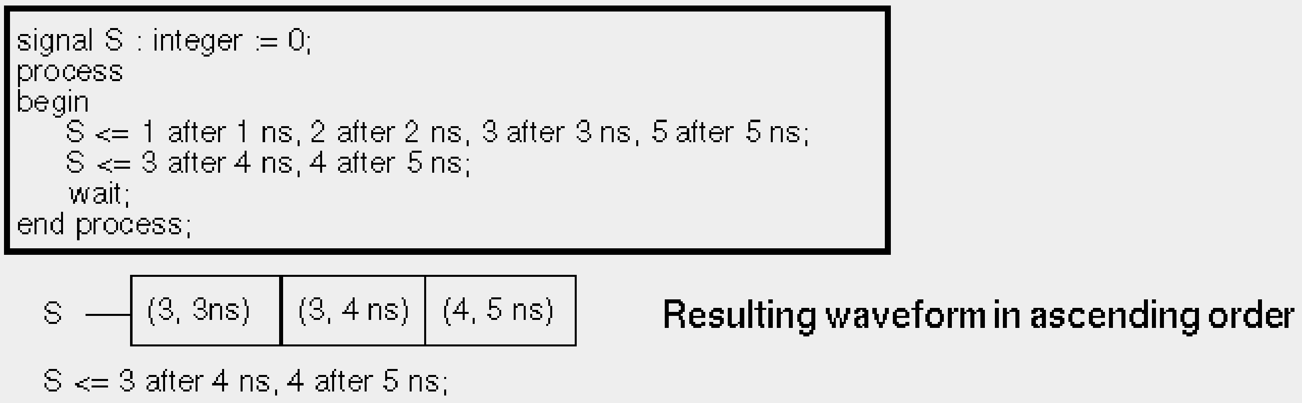
The time in the new value/time pair in the three right hand side examples is at most as big as the time of the pair in the list, thus the old entry is overwritten by the new one (step 1 deletes, step 2 adds new pair), independent from rejection limit values.

3.4.5 Inertial Delay (2)



```
signal S : integer := 0;
process
begin
     S <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;
     S <= 3 after 4 ns, 4 after 5 ns;
     wait;
end process;
```

S
(1, 1 ns)
(3, 3 ns)
(5, 5 ns)
Starting waveform in ascending order
S <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;



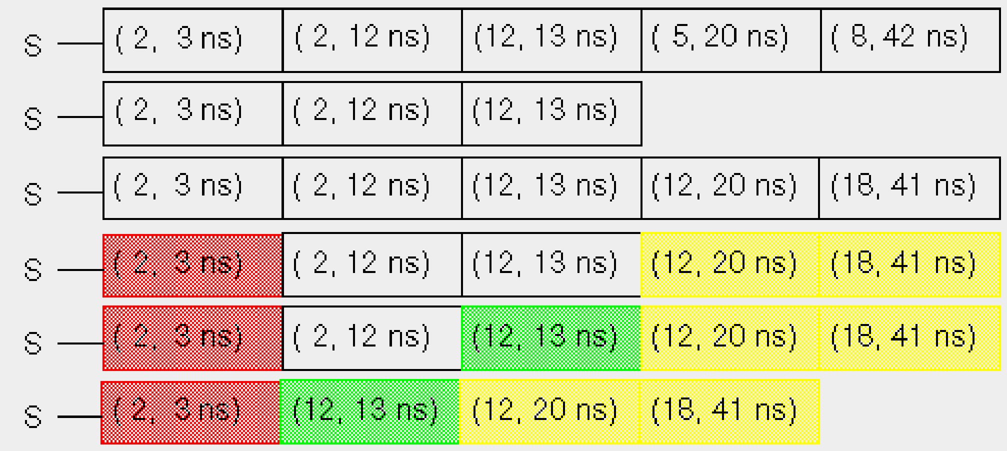
```
signal S : integer := 0;
process
begin
     S <= 1 after 1 ns, 2 after 2 ns, 3 after 3 ns, 5 after 5 ns;
     S <= 3 after 4 ns, 4 after 5 ns;
     wait;
end process;
```

S
(3, 3ns)
(3, 4 ns)
(4, 5 ns)
Resulting waveform in ascending order
S <= 3 after 4 ns, 4 after 5 ns;

After the first signal assignment "S <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;" the list contains three value/time pairs.
The second signal assignment "S <= 3 after 4 ns, 4 after 5 ns;" deletes the last entry from the list because 4 ns <= 5 ns (step 1) and appends two entries (step 2). Then, all new transactions (step 3) and also (3, 3 ns) will be marked, because this transaction is a direct predecessor of a marked transaction and it has the same value as this marked transaction, namely 3 (step 5). The unmarked entries will be deleted, that is the pair (1, 1 ns) and (2, 2 ns) in the second case respectively (step 7).

3.4.6 Inertial Delay (3)

```
signal S : integer := 0;
process
begin
    S <= 2 after 3 ns, 2 after 12 ns, 12 after 13 ns, 5 after 20 ns, 8 after 42 ns;
    S <= reject 15 ns inertial 12 after 20 ns, 18 after 41 ns;
    wait;
end process;
```

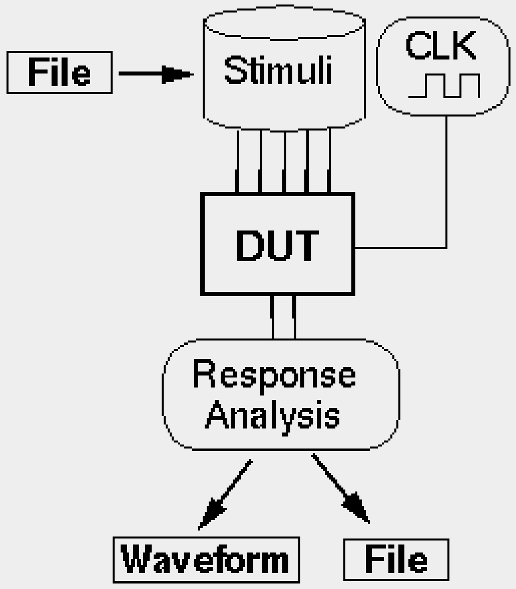

S (2, 3 ns) (2, 12 ns) (12, 13 ns) (5, 20 ns) (8, 42 ns)
S (2, 3 ns) (2, 12 ns) (12, 13 ns)
S (2, 3 ns) (2, 12 ns) (12, 13 ns) (12, 20 ns) (18, 41 ns)
S (2, 3 ns) (2, 12 ns) (12, 13 ns) (12, 20 ns) (18, 41 ns)
S (2, 3 ns) (2, 12 ns) (12, 13 ns) (12, 20 ns) (18, 41 ns)
S (2, 3 ns) (12, 13 ns) (12, 20 ns) (18, 41 ns)

The signal assignment "S <= 2 after 3 ns, 2 after 12 ns, 12 after 13 ns, 5 after 20 ns, 8 after 42 ns;" builds the first list.
The second signal assignment "S <= reject 15 ns inertial 12 after 20 ns, 18 after 41 ns;" modifies this list in the following way:

step 1 : all pairs with time values greater than or equal to 20 ns will be removed
step 2 : the new pairs will be attached
step 3 : all new transactions will be marked (light gray);
step 4 : old transactions with a time value smaller than the time value of the first new transaction (20 ns) minus the reject limit (15 ns), i.e. 5 ns, will be marked (dark gray)
step 5 : still unmarked transactions will be marked (gray), if they are direct predecessor of a marked transaction and contain the same value as the already marked transaction
step 6 : the current value/time pair will be marked; as this pair has to be marked anyway it is not displayed in the list
step 7 : all unmarked transactions will be deleted

3.5 Testbenches

Example of a testbench



- Stimuli transmitter to DUT (testvectors)
- Needs not to be synthesizable
- No ports to the outside
- Environment for DUT
- Verification and validation of the design
- Several output methods
- Several input methods

A testbench is used to verify the specified functionality of a design. It provides the stimuli for the Device Under Test (DUT) and analyses the DUT's respones or stores them in a file. Information necessary for generating the stimuli can be integrated directly in the testbench or can be loaded from an external file. Simulation tools visualize signals by means of a waveform which the designer compares with the expected response. In case the waveform does not match the expected response, the designer has to correct the source code. When dealing with bigger designs, this way of verification proofs impractical and will likely become a source of errors. The only way out would be a widely automated verification, but this is still a dream for the future.

3.5.1 Structure of a VHDL Testbench

entity TB_TEST is end TB_TEST;	• **Empty entity**
architecture BEH of TB_TEST is -- component declaration of the DUT -- internal signal definition begin -- component instantiation of the DUT -- clock generation -- stimuli generation end BEH;	• **Declaration of the DUT** • **Connection of the DUT with testbench signals** • **Stimuli and clock generation (behavioural modelling)** • **Response analysis**

configuration CFG_TB_TEST of TB_TEST is
  for BEH;
    -- customized configuration
  end for;
end CFG_TB_TEST;
	• **default or customized configuration to simulate the testbench**

The entity of a testbench is completely empty, because all necessary stimuli are created standalone. Otherwise a testbench for the testbench would be needed. As the DUT's inputs cannot be stimulated directly, internal temporary signals have to be defined. In order to distinguish between port names and internal signals, prefixes can be employed, for instance "W_" to indicate a wire.

The declaration part of the testbench's architecture consists of the definition of the internal signals, a component declaration of the DUT or DUTs and perhaps some constant definitions e.g to set the clock period duration. A stimuli process provides the values for the DUT's input ports, in which behavioural modelling can be employed as there is no need to synthesize it. Sometimes, models of external components are available (probably behavioural description, only) and can be used similar to create a whole system.

A configuration is used to pick the desired components for the simulation.

3.5.2 Example

```
entity TB_TEST is
end TB_TEST;

architecture BEH of TB_TEST is
  component TEST
    port(CLK      : in std_logic;
         RESET    : in std_logic;
         A        : in integer range 0 to 15;
         B        : in std_logic;
         C        : out integer range 0 to 15);
end component;

  constant PERIOD  : time := 10 ns;
  signal W_CLK     : std_logic := '0';
  signal W_A, W_C  : integer range 0 to 15;
  signal W_B       : std_logic;
  signal W_RESET   : std_logic;
begin
  DUT : TEST
    port map(CLK      => W_CLK,
             RESET    => W_RESET,
             A        => W_A,
             B        => W_B,
             C        => W_C);
  . . .
```

- **Declaration part of the architecture**
  - **component**
  - **internal signals**
  - **subprograms**
  - **constants**

- ***Instantiation of the DUT***
- **Connecting internal signals with the module ports**

Initial clock signal value set to '0'

The example shows a VHDL testbench for the design TEST. The design is declared as component in the declaration part of the architecture BEH. A constant PERIOD is defined to set the clock period. Internal signals that are needed as connections to the DUT are also declared. It is important to initialize the clock signal either to '0' or '1' instead of its default value 'u' because of the clock generation construct that is used later on.

Clock and Reset Generation

```
  W_CLK <= not W_CLK after
PERIOD/2;
```

-- complex version

```
  W_CLK <= '0' after PERIOD/4
when W_CLK='1' else
                '1' after 3*PERIOD/4
when W_CLK='0' else
                '0';
```

- **Simple signal assignment**
  - **endless loop**
  - **W_CLK must be initialized to '0' or '1' (not 'u' = 'u')**
  - **symmetric clock, only**
- **Conditional signal assignment**
  - **Complex clocking schemes**
- **Realization as process introduces huge overhead**

```
  W_RESET <= '0',
                   '1' after 20 ns,
                   '0' after 40 ns;
```

- **Reset generation**

```
  assert now>100*PERIOD
         report "End of simulation"
         severity failure;
```

- **Simulation termination via assert**

The clock stimulus is the most important one for synchronous designs. It can be created either with a concurrent signal assignment or within a clock generation process. As a process requires a lot of "overhead" when compared to the implemented functionality, the concurrent version is recommended.

In the most simple form shown on top, the clock runs forever and is symmetric. As the signal value is inverted after half of the clock period, the initial signal value must not be 'u', i.e. its start value has to be explicitly declared. The more elaborated example below shows the generation of an asymmetric clock with 25% duty cycle via conditional signal assignments. Please note that the default signal value needs not to be specified because of the unconditional else path that is required by the conditional signal assignment.

The complete simulation is stopped after 100 clock cycles via the ASSERT statement. Of course, the time check can be included in a conditional signal assignment as well.

Clocks with a fixed phase relationship are modeled best with the ' **delayed** ' attribute similar to the following VHDL statement: "CLK_DELAYED <= W_CLK'delayed(5 ns);"

The reset realization is straight forward: It is initialized with '0' at the beginning of the simulation, activated, i.e. set to '1', after 20 ns and returns to '0' (inactive) after an additional 20 ns for the remainder of the simulation.

Stimuli Generation

```
. . .
  STIMULI : process
  begin
     W_A    <= 10;
     W_B    <= '0';
     wait for 5*PERIOD;
     W_B <= '1';
     wait for PERIOD;
     . . .
     wait;
  end process STIMULI;
. . .
```
	• **Simple stimuli generation**

```
. . .
process(W_C)
begin
  case W_C is
    when 3       => W_B <= '1'
after 10 ns;
    when others => W_B <= '0'
after 10 ns;
  end case;
end process;
. . .
```
	• **Dynamically generated stimuli from DUT response**

All other DUT inputs can be stimulated the same way. Yet, the pattern generation via processes is usually preferred because of its sequential nature. Please note that a wait statement is required to suspend a process as otherwise it would restart. More complex testbenches will show dynamic behaviour, i.e. the input stimuli will react upon DUT behaviour. This may lead eventually to a complete behavioural model of the DUT environment.

Response Analysis

```
. . .
process(W_C)
begin
  assert W_C > 5 -- message, if false
  report "WRONG RESULT!!"
  severity ERROR;
end process;
. . .
```

- **Assertion with severity level**
  - **Note**
  - **Warning**
  - **Error (default)**
  - **Failure**

```
. . .
process(W_C)
begin
  assert W_C > 5
  report "WRONG RESULT in " &
            W_C'path_name &
         "Value: " &
            integer'image(W_C)
   severity error;
. . .
```

- **Additional attributes in VHDL'93**
  - **'path_name**
  - **'inst_name**
  - **'image**

```
WRONG RESULT in TB_TEST:W_C Value: 2
```

- **Report in simulator**

'93 **Additional attributes for debugging purposes**

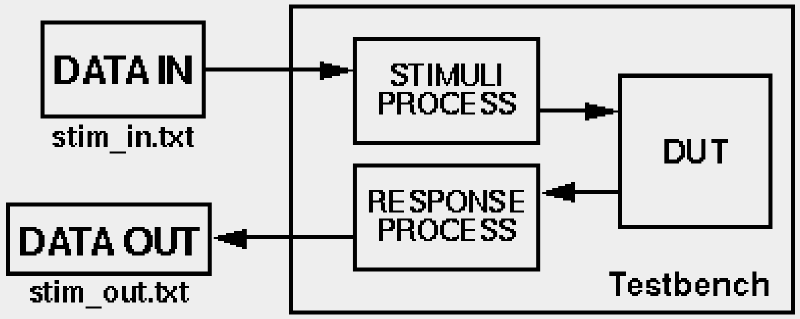
'93 **Report without assert statement**

The ' **assert** ' statement is suited best for performing automatic response analysis. An assertion checks a condition and will report a message, if the condition is false. Depending on the chosen severity level and the settings in the simulation tool, the simulation either resumes (e.g. note, warning) or stops (e.g. error, failure) after reporting that the assertion fails. The default severity level is ' **error** '.

The message that is reported is defined by the the designer. In order to achieve dynamic reports that offer more detailed debugging information in case of errors several new attributes were defined in VHDL'93. They provide additional information about the circumstances that lead to the failing assertion: ' **instance_name** ' and ' **path_name** ' may be used to locate the erronous module. The path information is necessary when one component is instantiated at various places within a complex design. The ' **image** ' attribute looks like a function call. Its argument is returned as string representation and the data type is the prefix of the attribute.

As a report is generated whenever the assertion condition evaluates to ' **false** ', it can be forced by setting this condition to the fixed value ' **false** '. This construction is no longer necessary in VHDL'93, i.e. the 'report' keyword may be used without a preceding assertion.

3.6 File I/O



- **Package TEXTIO of STD library**
- **Important functions and procedures:**
  - **readline(...), read(...),**
  - **writeline(...), write(...),**
  - **endfile(...)**
- **Additional data types (text, line)**

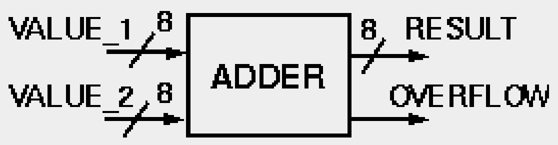
- **READ / WRITE overloaded for all predefined data types:**
  - **bit, bit_vector**
  - **boolean**
  - **character, string**
  - **integer, real**
  - **time**

In VHDL, the designer is allowed to load data from or save data to a file. To do this, it is necessary to include the TEXTIO package of the STD library which contains basic functions and procedures. These subprograms (write, writeline, read, readline, ...) facilitate the file I/O mechanism and are defined for the predefined VHDL data types. The main application of file I/O is simulation flexibility. This means that simulation stimuli and response analysis are left to specialized software, which is often easier than writing the corresponding VHDL models.

To get the information from a file, the user reads it line by line (READLINE procedure) and stores the data in a variable of the data type ' **line** '. Afterwards it is possible to access the data from this line with the READ command. Usually a line contains information of different data types. To interpret the data in a correct way, i.e. a string as a string and an integer value as an integer, you have to employ actual parameters of the same data type. This fact will become clearer in the following example. Besides it is necessary to read the space character between the information in a separate read statement!

The file output works in a similar way, i.e. a line is assembled first via WRITE commands and is finally written to the file with a WRITELINE statement.

3.6.1 Example for File I/O (1/4)



- **ADDER module**
  - **Adds two 8 bit vectors and provides an 8 bit result vector**
  - **Generates an overflow signal**

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

entity ADDER is
  port(VALUE_1      : in   std_logic_vector(7 downto 0);
       VALUE_2      : in   std_logic_vector(7 downto 0);
       OVERFLOW : out std_logic;
       RESULT      : out std_logic_vector(7 downto 0));
end ADDER;

architecture RTL of ADDER is
  signal INT_RES    : std_logic_vector(8 downto 0);
  signal INT_VAL_1 : std_logic_vector(8 downto 0);
  signal INT_VAL_2 : std_logic_vector(8 downto 0);
begin
  INT_VAL_1   <= '0' & VALUE_1;
  INT_VAL_2   <= '0' & VALUE_2;
  INT_RES      <= INT_VAL_1 + INT_VAL_2;
  RESULT       <= INT_RES(7 downto 0);
  OVERFLOW <= INT_RES(8);
```

- **Entity and architecture of the ADDER module**
- **std_logic_unsigned package**
  - **copyright by Synopsys (EDA software company)**
  - **not standardized by IEEE**
  - **overloaded**

end RTL;	**mathematical operators where std_logic_vector is treated as unsigned number**

The first part of the file I/O example shows a design of an adder (entity and architecture). Please note the use of the STD_LOGIC_UNSIGNED package. Although this package is located in the IEEE library it is not standardized by the institute. The package was created by Synopsys Inc., an EDA software company. Their packages, however, have achieved the level of a quasi-standard in industry.

In the architecture, the operands are extended to 9 bits in order to avoid an overflow during the actual addition. The most significant bit of the resulting vector is then used as OVERFLOW signal and the lower 8 bits represent the result of the addition.

Example (2/4)

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_textio.all;
use STD.textio.all;
	• **Add package std.textio and IEEE.std_logic_textio for file I/O functions and procedures**

entity TB_ADDER is
end TB_ADDER;
 	
architecture BEH of TB_ADDER is
  component ADDER
    port(VALUE_1 : in
  std_logic_vector(7 downto 0);
        VALUE_2 : in
  std_logic_vector(7 downto 0);
        OVERFLOW : out std_logic;
        RESULT : out
std_logic_vector(7 downto 0));
  end component;
 	
  signal W_VALUE_1 :
std_logic_vector(7 downto 0);
  signal W_VALUE_2 :
std_logic_vector(7 downto 0);
  signal W_OVERFLOW : std_logic;
  signal W_RESULT :
std_logic_vector(7 downto 0);
 	
begin
  DUT : ADDER
    port map(VALUE_1 =>
W_VALUE_1,
            VALUE_2 =>
W_VALUE_2,
            OVERFLOW =>
W_OVERFLOW,
            RESULT =>
W_RESULT);
	• **Common testbench structure**
	    ❍ **Empty entity; no external interface**
	    ❍ **Component declaration and instantiation**
	    ❍ **Definition of internal signals to connect the input/output ports with the stimuli/response analysis processes**

The testbench for the ADDER design follows the usual structure. The entity remains empty as no higher hierarchy level exists and in the declarative part of the architecture the design under test and signals for its interface ports are declared. In the component instantiation statement, these signals are connected to the DUT ports.

Please note that two additional packages are used to handle the file I/O. The standard TEXTIO package provides the basic functionality. The STD_LOGIC_TEXTIO, again a Synopsys package that is not standardized by the IEEE, provides overloaded subprograms to handle STD_ULOGIC based data types.

Example (3/4)

```
STIMULI : process
   variable L_IN : line;
  variable CHAR : character;
  variable DATA_1 : std_logic_vector(7 downto 0);
  variable DATA_2 : std_logic_vector(7 downto 0);
   file STIM_IN : text is in "stim_in.txt";
begin
  W_VALUE_1 <= (others => '0');
  W_VALUE_2 <= (others => '0');
  wait for PERIOD;
  while not endfile (STIM_IN) loop
     readline (STIM_IN, L_IN);
     hread (L_IN, DATA_1);
    W_VALUE_1 <= DATA_1;
     read (L_IN, CHAR);
     hread (L_IN, DATA_2);
    W_VALUE_2 <= DATA_2;
    wait for PERIOD;
  end loop;
  wait;
end process STIMULI;
```

- **STIMULI process**
  - **File access is limited to only one line at a certain time**
  - **Only variables are allowed for the parameters of the read functions**
  - **The function hread(...) is defined in the IEEE.std_logic_textio package; it reads hex values and transforms them into a binary vector**

```
00 A1
FF 01
FF 00
11 55
0F 01
1F 05
AA F3
```

- **Stimuli file "stim_in.txt"**
  - **Each line contains two hex values to stimulate the inputs of the ADDER module**

After having instantiated the entity ADDER, a stimuli process is needed that provides the test patterns. Different from previous examples the stimuli are read from the file "stim_in.txt". Some example content is shown below the process code. The first hex value of each line is the stimulus for the port VALUE_1 (2 hex values -> 8 bits), the second hex value is for the port VALUE_2. Each line will be used as input vector for a duration of PERIOD.

The signals W_VALUE_1 and W_VALUE_2 are initialized with the all zeros vector. After one clock cycle a loop is started where it is checked, whether the stumli file STIM_IN still contains data. If so, the following statements will be executed, otherwise they will be skipped and the process will suspend at the last ' **wait** ' statement of this process.

Reading the file works as follwos: First one line L_IN is read from the file STIM_IN by the READLINE command. Afterwards, three different values, an 8 bit standard logic vector, followed by a single character and another 8 bit vector are extracted from this line L_IN. The data type is selected by the data type of the variable used in the READ procedure. HREAD is also a kind of read command which transforms the value read from hexadecimal to STD_LOGIC_VECTOR automatically. HREAD is contained in the STD_LOGIC_TEXTIO package.

After the values from the file are stored in the variables, they have to be assigned to the signals that are connected to the input ports of the DUT. After finishing one line, the process waits for one PERIOD before checking again, whether the stimuli file STIM_IN still contains some data.

Example (4/4)

```
RESPONSE :
process(W_RESULT)
  variable L_OUT : line;
  variable CHAR_SPACE :
character := ' ';
  file STIM_OUT : text is out
"stim_out.txt";
begin
   write (L_OUT, now);
   write (L_OUT,
CHAR_SPACE);
  write (L_OUT, W_RESULT);
  write (L_OUT,
CHAR_SPACE);
  hwrite (L_OUT,
W_RESULT);
  write (L_OUT,
CHAR_SPACE);
   write (L_OUT,
W_OVERFLOW);
   writeline (STIM_OUT,
L_OUT);
end process RESPONSE;
```

- **Response process of the testbench**
  - **'NOW' is a function returning the current simulation time**
  - **Several write commands assemble a line**
  - **Writeline saves this line in the file**
  - **The function hwrite(...) is defined in the IEEE.std_logic_textio package; it transforms a binary vector to a hex value and stores it in the line**

```
0 NS UUUUUUUU 00 U
0 NS XXXXXXXX 00 X
0 NS 00000000 00 0
20 NS 10100001 A1 0
40 NS 00000000 00 1
60 NS 11111111 FF 0
80 NS 01100110 66 0
100 NS 00010000 10 0
120 NS 00100100 24 0
140 NS 10011101 9D 1
```

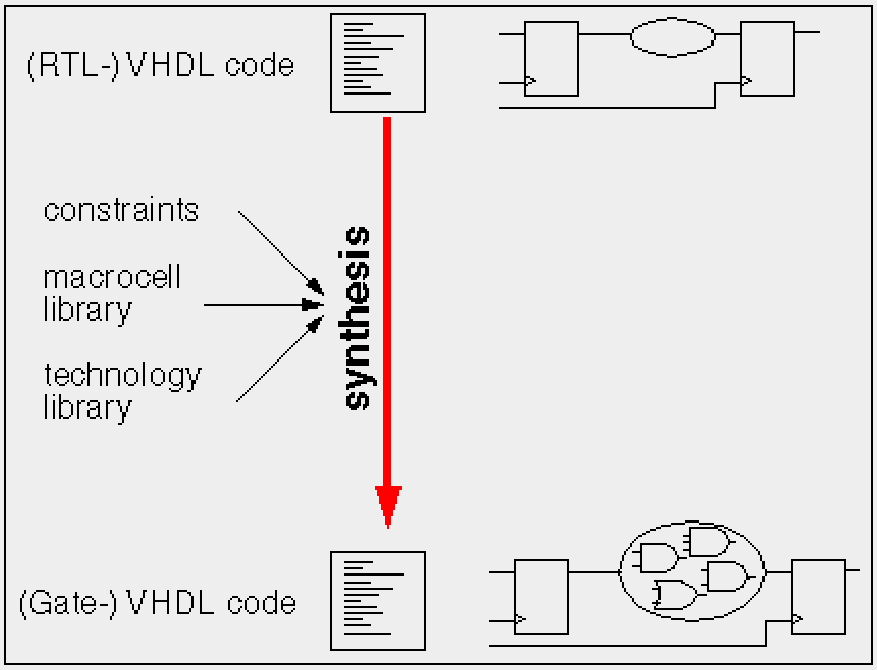
- **Response file "stim_out.txt"**
  - **4 columns containing:**
    - Simulation time
    - 8 bit result value (binary and hex)
    - Overflow bit

The file output is just the inverse of the file input presented before: A line has to be composed and written to the file when finished. The RESPONSE process stores the current simulation time, the RESULT as STD_LOGIC_VECTOR and in hexadecimal format and the OVERFLOW signal. The process is activated whenever an event occurs at W_RESULT (sensitivity list). This is the reason why three lines are written to the fileat 0 NS. First, W_RESULT is uninitialized because no initial value is specified explicitly in the signal definition. So W_RESULT consists of 'U's only, which HWRITE transforms to '0'. The 'X's in the second line occur because the overloaded operator returns 'X' (unknown) when undefined values are added. 'X' and 'U' are treated just same, i.e. the result is also 0.

4. Synthesis

- What is Synthesis?
- RTL-style
- Cominatorical Logic
- Sequential Logic
- Finite State Machines and VHDL
- Advanced Synthesis

4.1 What is Synthesis?

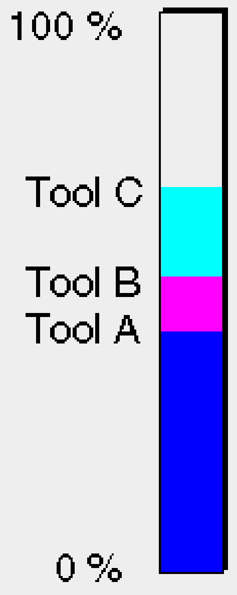


- **Transformation of an abstract description into a more detailed descrition**
  - **"+" operator is transformed into a gate netlist**
  - **"if (VEC_A = VEC_B) then" is realized as a comparator which controlls a multiplexer**

- **Transformation depends on several factors**

In general, the term "synthesis" is used for the automated transformation of RT level descriptions into gate level representations. This transformation is mainly influenced by the set of basic cells that is available in the target technology. While simple operations like comparisons and either/or decisions are easily mapped to boolean functions, more complex constructs like mathematical operators are mapped to a tool specific macro cell library first. This means that a number of adder, multiplier, etc. architectures are known to the synthesis tool and these designs are treated as if they were designed by the user.

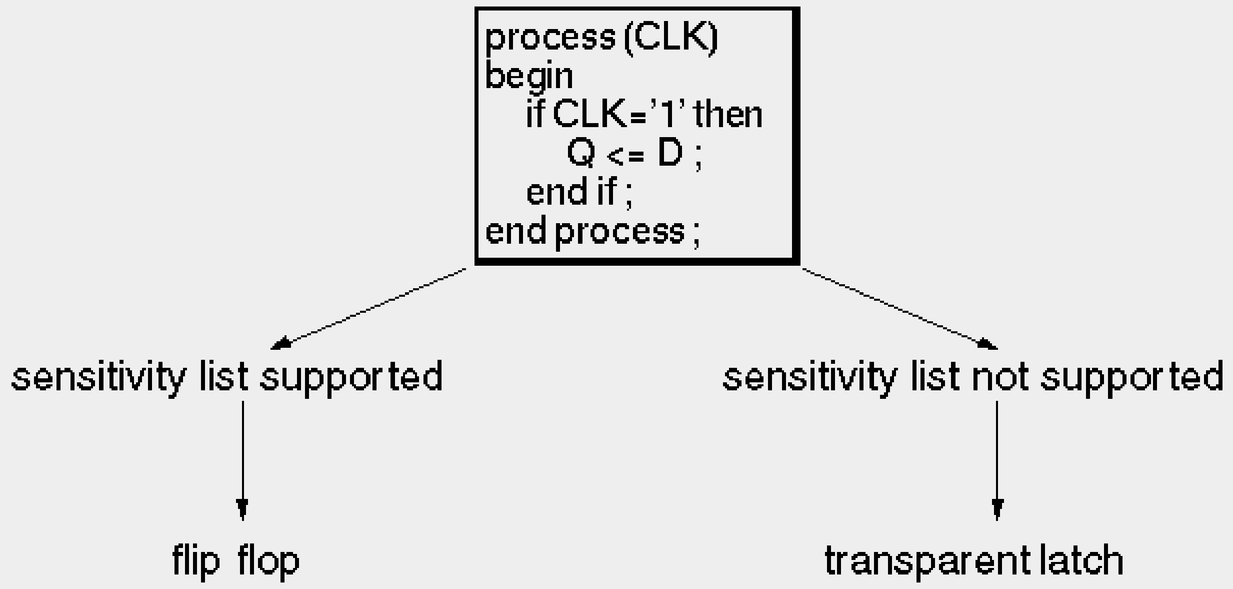
4.1.1 Synthesizability



- **Only a subset of VHDL is synthesizable**
- **Different Tools support different subsets**
  - **records?**
  - **arrays of integers?**
  - **clock edge detection?**
  - **sensitivity list?**
  - **...**

The macro cell library is just one distinguishing feature of synthesis software. VHDL itself is not fully synthesizable and the available tools differ in the language subset that is supported. Complex user defined data structures like records and multidimensional arrays (e.g. simple arrays of integers) turn out to be the most problematic cases.

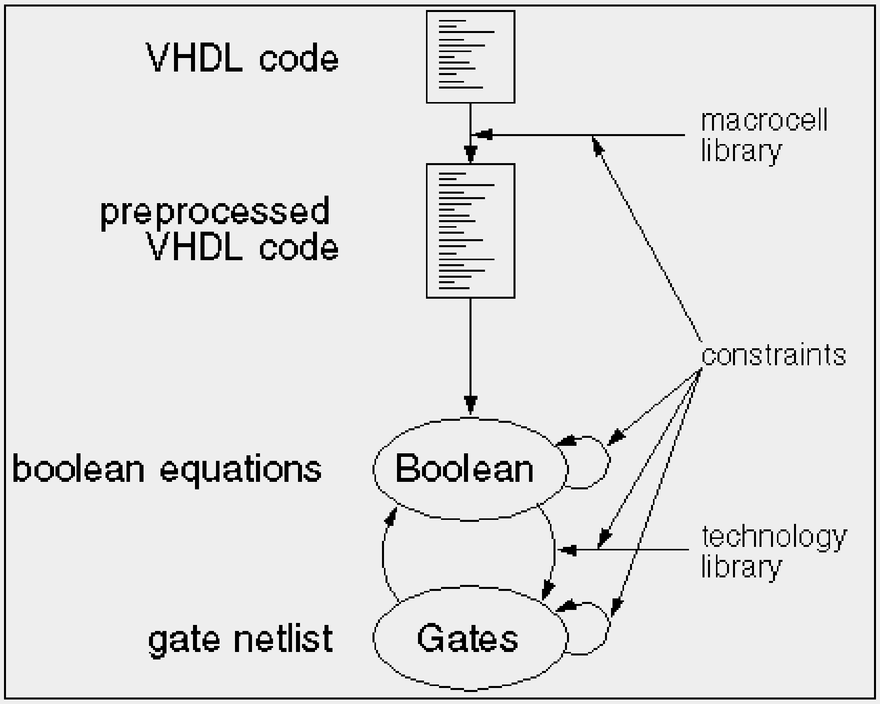
4.1.2 Different Language Support for Synthesis


```
process (CLK)
begin
    if CLK='1' then
        Q <= D ;
    end if ;
end process ;
```

The consequences of different language support on the resulting hardware are demonstrated at the example of a clocked process. In case the synthesis tool supports sensitivity lists the result is a flip flop because the process is triggered with every event at CLK and the value of D will be assigned to Q, if CLK='1' as a result of this event. Thus, the behaviour of a rising edge triggered flip flop is modeled here.

Even if synthesis tools do not support sensitivity lists in general, they often look for templates that describe the behaviour of registers. Usually, the check for the CLK event has to be part of the if condition, as well. If the sensitivity list is ignored and the code can not be matched to a register template, a level triggered latch will be generated!

4.1.3 How to Do?



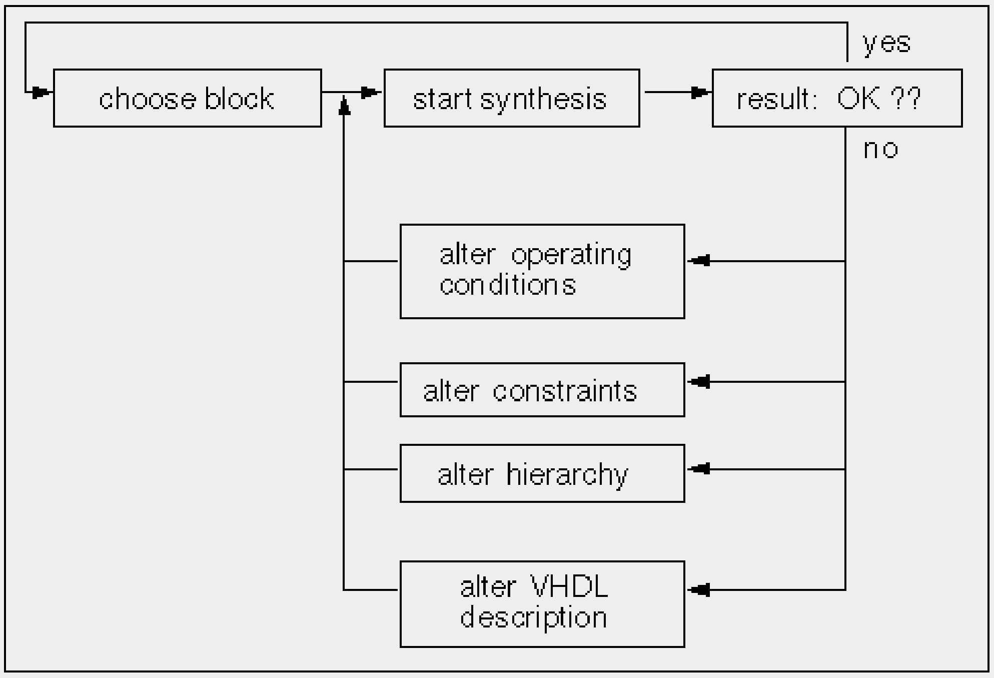
- Contraints
  - speed
  - area
  - power
- Macrocells
  - adder
  - comparator
  - businterface
- Optimizations
  - boolean: mathematic
  - gate: technological

Besides the fixed synthesis constraints set by the target technology and the tool capabilities, "soft" constraints that are imposed by the designer have to be considered as well. Maximum operating speed and required hardware resources are usually the main targets for netlist optimization. This is possible either on a purely abstract mathematical model or by different mappings of the boolean functions on the available technology cells. Due to the complexity, the optimization phase requires quite a lot of iterations before the software reports its final result.

4.1.4 Essential Information for Synthesis

- **Load values**
- **Path delays**
- **Driver strengths**
- **Timing**
- **Operating conditions**

4.1.5 Synthesis Process in Practice



- **In most cases synthesis has to be carried out several times in order to achieve an optimal synthesis result**

Even after extensive optimizations by the synthesis tool, the result is pretty often not compliant with the system requirements. In this case, the input to the software has to be modified. Several parameters may be modified by the designer: The block operating conditions includes environmental conditions like operating temperature as well as settings like necessary driver strength (fan-out) or capacitance of wire connections. They have a direct impact on the actual wire delays.

Hierarchy alterations can simply be performed by selecting a bigger block and allowing the tool to break up the hierarchy definitions from the VHDL source code. If the repeated attempts still fail to produce the desired result, modifications of the original VHDL code become the last way out.

4.1.6 Problems with Synthesis Tools

- Timing issues
  - layout information is missing during the synthesis process
  - clock tree must be generated afterwards
- Complex clocking schemes (inverted clocks, multiple clocks, gated clocks)
- Memory
  - synthesis tools are not able to replace register arrays with memory macro cells
- Macro cells
  - no standardized way for instantiation of existing technology macro cells
- IO-pads
  - ASIC-libraries have several different IO-pads
  - selection by hand, either within the synthesis tool or in the top level entity

While the algorithms have matured considerably there exist still a number of problems and pitfalls for the users of synthesis tools. Many issues are related to the separation of netlist and layout generation. Therefore the length of the interconnections can only be estimated during synthesis and critical nets have to modified by hand afterwards. The clock tree, for example, requires extensive buffering in order to distribute the clock signal evenly on the chip and has to be generated by hand.

While synthesis of synchronous designs with a single clock source is fairly simple, practical systems, unfortunately, often require additional clock signals. This introduces asynchronous behaviour which is very complex to handle as long as the exact propagation delays are unknown. Macro cells that are available in the target technology are also hard to use. This is especially true for memory cells that can not used by synthesis tools automatically. The same applies to the I/O cells of ASIC libraries that have to chosen by hand.

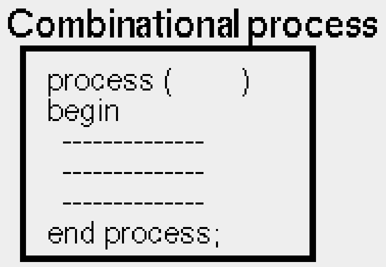
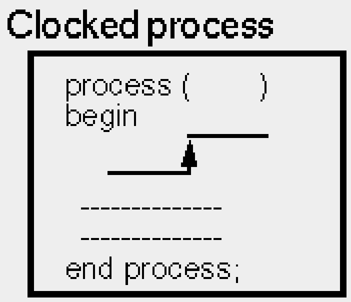
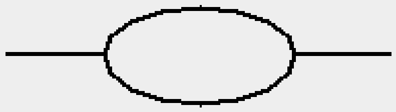
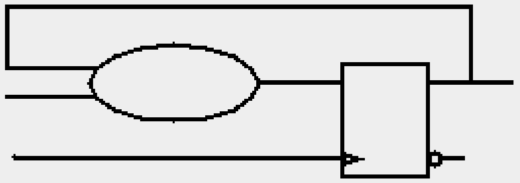
4.1.7 Synthesis Strategy

- **Consider the effects of different coding styles on the inferred hardware structures**
- **Appropriate design partitioning**
    - **critical paths should not be distributed to several synthesis blocks**
    - **automatic synthesis performs best at module sizes of several 1000 gates**
    - **different optimization constraints may be used for separate blocks**

The VHDL coding style itself has a rather big impact on the synthesis result. Therefore it is necessary to keep this in mind even if the model is to be synthesized at the last step of the development cycle.

The design partitioning should be reviewed prior to the synthesis runs. This is mainly due to the fact that the algorithms perform best at module sizes of several thousand gates. It is not necessary to rewrite the RTL description as submodules can be grouped together during synthesis. This allows for different optimization settings, i.e. high speed parts can be synthesized with very stringent timing constraints while non critical parts should consume the least amount of ressources (area) possible.

4.2 RTL-style

Combinational process	Clocked process
 Combinational process process ( ) begin -------------- -------------- -------------- end process;	 Clocked process process ( ) begin -------------- -------------- end process;
• Complete sensitivity list • Complete IF-statements or default-assignment	• if-elsif-elsif-end if structure • The first IF has a reset • The last elsif has the CLK poll • No else branch
	

RTL - register transfer level. In variation to the modelling on the behaviour level, where VHDL can be used like any other programming language, the RT level is one stage nearer to the hardware. By that the algorithm is partitioned into pure combinational groups and in clocked groups with storage ability, that is flip-flops. In a pure combinational group no storage element should be contained. On the other hand, IF assignments must be written completely, that means that the ELSE branch must not be forgotten, because otherwise unintended latches can be generated by the synthesis tool. Another possibility is to assign default values before the IF condition, which can be changed

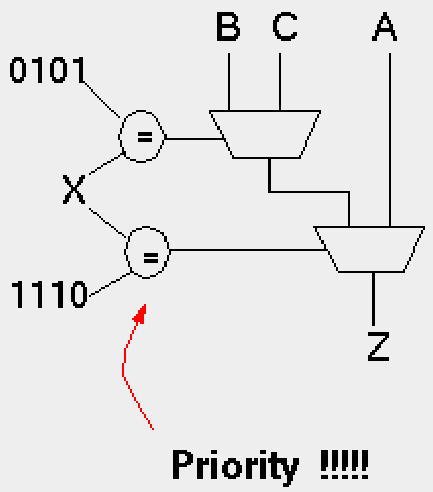
in the IF branch, if necessary.
A clocked process with a asynchronous reset has the following form:

```
process(clock, reset)
begin
if reset = '1' then ...(Reset-assignmnet)
elsif clock'event and clock = '1' then ...(assignment to FFs) ;
end if ;
end process;
```

4.2.1 Combinatorics

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity IF_EXAMPLE is
port (A, B, C, X : in std_ulogic_vector(3 downto 0);
        Z          : out std_ulogic_vector(3 downto 0));
end IF_EXAMPLE;

architecture A of IF_EXAMPLE is
begin
    process (A, B, C, X)
    begin
      if ( X = "1110" ) then
        Z <= A;
      elsif (X = "0101") then
        Z <= B;
      else
        Z <= C;
      end if;
    end process;
end A;
```

Hardware realisation



An IF assignment is always implemented as one or several mutliplexers in the synthesis. With it, every condition corresponds to a multiplexer (IF or ELSIF branch).
Please note the priority of the single branches. In the simulation of an IF assignment, when entering a condition, the others following will not be worked off.
This branches' order must be kept in the synthesis, as well. Yet, it is only possible by means of series connected multiplexers, whereas the FIRST condition is attached to the SELECT entry of the LAST multiplexer.

4.2.2 Complete sensitivity lists

?

- If SEL is missing in the sensitivity list, what will the behaviour be?

```
process (A, B, SEL)
begin
  if SEL = `1` then
    Z <= A;
  else
    Z <= B;
  end if;
end process;
```

- All signals which are read are entered into the sensitivity list
- Complete if-statement for the synthesis of combinational logic

A process is activated in the simulation, when on one of its sensitivity list's signal an event occures.

When the SEL signal is not contained in the list - as in above question - the process would be activated only by means of events on A or B.
This does not correspond to the desired behaviour of the described multiplexer on one side.
On the other side, most synthesis tools do not react on sensitivity lists, but handle the VHDL code contained in the process. In our case it is an IF assignment.
This IF assignment is always realised as multiplexer.
Result: If SEL is not contained in the sensitivity list, the component part will not be simulated, which is usually synthesised.
Upshot: All inputs - signals which are read - should always reside in the sensitivity list for the description of pure combinatorics.

4.2.3 WAIT statement <-> Sensitivity List

```
process
begin
  if SEL = `1` then
    Z <= A;
  else
    Z <= B;
  end if;
WAIT ON A,B,SEL;
end process;
```
	• **equivalent Processes**

```
process (A, B, SEL)
begin
  if SEL = `1` then
    Z <= A;
  else
    Z <= B;
  end if;
end process;
```

The sensitivity list can be replaced with a WAIT ON statement at the end of the process. The behviour of the process does not change with this replacement.

4.2.4 Incomplete assignments

- **What is the value of Z, if SEL = `0` ?**
- **What hardware would be generated during synthesis ?**

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity INCOMP_IF is
port (A, B, SEL :in std_ulogic;
        Z             : out std_ulogic);
end INCOMP_IF;

architecture RTL of INCOMP_IF is
begin
process (A, B, SEL)
begin
      if SEL = `1` then
          Z <= A;
      end if;
end process;
end RTL;
```

In this example, the 'else' branch has been left out.
If SEL='0', the old value of Z will be maintained in the simulation, that means no change will be carried out on Z.

For this it must be implemented with a storage element in the according hardware. Therefore, the synthesis tools create a latch, in which the SEL signal ia connected with the clock entry. It is an element very difficult to test in the synchronous design, and therefore it should not be used.
Normally only edge-triggered FF are used, which are ALL connected to one and the same clock signal.
Then the possibility exists to combine all FF with special Scan FF to a scan path; with an additional entry pin, the chip can be put into a scan test modus and internal values can be read out.

4.2.5 Rules for synthesizing combinational logic

- Complete sensitivity list
    - RTL behaviour has to be identical with the IC
    - an incomplete sensitiviy list can cause errors or warnings
- No incomplete If-statements are allowed
    - transparent latches

4.2.6 Modelling of Flip Flops

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity FLOP is
port (D, CLK        : in std_ulogic;
         Q                  : out std_ulogic);
end FLOP;

architecture A of FLOP is
begin
      process
      begin
          wait until CLK`event and CLK=`1`;
          Q <= D;
      end process;
end A;
```

Here, a D-flip-flop controlled by a clock pulse edge is described. If an event occurs at the clock signal and this event has the value ONE, the value of the pin D will be transferred to the pin Q.
(You could also await the negative clock pulse edge, then must be: CLK='0').

4.2.7 Description of a rising clock edge for synthesis

- **New standard for synthesis: IEEE 1076.6**

... if condition	***... wait until condition***
RISING_EDGE (*clock_signal_* name)	RISING_EDGE (*clock_signal_* name)
*clock_signal_* name'EVENT *and* *clock_signal* _name='1'	*clock_signal_* name'EVENT *and* *clock_signal* _name='1'
clock_signal _name='1' *and* *clock_signal_* name'EVENT	*clock_signal* _name='1' *and* *clock_signal_* name'EVENT
*not clock_signal_* name'STABLE *and* *clock_signal_* name='1'	*not clock_signal_* name'STABLE *and* *clock_signal_* name='1'
clock_signal _name='1' *and not* *clock_signal_* name'STABLE	*clock_signal* _name='1' *and not* *clock_signal_* name'STABLE
	clock_signal _name='1'

As the sensitivity list is usually ignored by synthesis tools and wait statements are not synthesizable in general, a solution to the problem of modelling storage elements has to be found. Synthesis tools solved this issue by looking for certain templates in the VHDL code, namely the first option ('if/wait until X'event and X='1' then') of the two process styles. All alternatives show the same behaviour during simulation, however. Please note that the event detection in the 'wait until' statement is redundant as an event is implicitly required by the 'wait until' construct.

In the meantime, the IEEE standard 1076.6 that lists the VHDL constructs that should infer register generation was passed. As this standard is not fully supported by synthesis tools, yet, the first option is still the most common way of describing a rising/falling clock edge for synthesis. When asynchronous set or reset signals are present, only the IF variant is applicable.

4.2.8 Describing a rising clock edge by means of a function call

- **In Std_Logic_1164 package**

```
process
begin
  wait until RISING_EDGE(CLK);
  Q <= D;
end process;
```

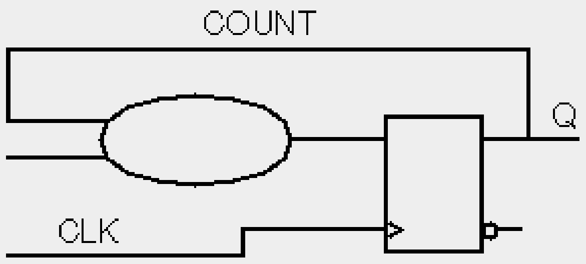
```
function RISING_EDGE (signal CLK : std_ulogic)
            return boolean is
begin
    if (  CLK`event and CLK =`1`
                            and CLK`last_value=`0`) then
        return true;
    else
        return false;
    end if;
end RISING_EDGE;
```

The function rising_edge is only mentioned here to complete things. It is not accepted by the most synthesis tools (because of 'last_value), but it can be used in the simulation.

4.2.9 Counter synthesis

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity COUNTER is
port ( CLK    : in std_ulogic;
       Q    : out integer  range 0 to 15 );
end COUNTER;

architecture A of COUNTER is
  signal COUNT : integer
range 0 to 15 ;
begin
  process (CLK)
  begin
    if CLK`event and CLK = `1` then
      if (COUNT >= 9) then
        COUNT <= 0;
      else
        COUNT <= COUNT +1;
      end if;
    end if;
  end process;
   Q <= COUNT;
end A;
```

- **For all signals which receive an assignment in clocked processes memory is synthesized**



This is the description of a counter without resetting line. We have to point out some peculiarities:
At first the range assignment in the port declaration of the output. Here, the digits from 0 to 15 are only allowed, that means 4 bit are sufficient for a binary representation. The port signal Q is replaced by means of the synthesis tool by a 4 bit signal (ultimately all types are transferred by means of the synthesis tools into std_logic types).
So a 4 bit counter is realised.
Another peculiarity is, that the port modus 'out' of the signal Q can be only written on, it cannot be read. Therefore, a (temporary) signal COUNT must be declared within the architecture, in order to be able to implement the query COUNT >= 9.
The result of the counting is then transferred into Q in a concurrent signal assignment (Q <= COUNT) as an additional process. That means every result at COUNT triggers the assignment Q <= COUNT.
The internal IF assignment thus describes the combinatoric before the FF. The number of FF is derived from the width of the signal, which receive an assignment inside the outer IF assignment ('event and so on). In our case, it is only the signal COUNT of 4 bit (because of the range 0 to 15).
Important : In the sensitivity list there is only the signal CLK!!!

4.2.10 FF with asynchronous reset

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity ASYNC_FF is
port (    D, CLK, SET, RST : in std_ulogic;
          Q                : out std_ulogic);
end ASYNC_FF;

architecture A of ASYNC_FF is
begin
   process  (CLK, RST, SET)
   begin
     if (RST = `1`) then
       Q <= `0`;
     elsif SET ='1' then
       Q <= '1';
     elsif (CLK`event and CLK = `1`) then
       Q <= D;
     end if;
   end process;
end A;
```

- **if/elsif - structure**
  - **The last elsif has an edge**
  - **No else**
- **It has a sensitivity list !!**

In order to describe a RESET, you have to proceed as follows:
The RESET signal is also in the sensitivity list. The RESET query comes at first (asynchronous RESET).
The CLK query stands in the following 'elsif' assignment. There is no 'else' branch!!! This would lead to an error message in the synthesis.
The combinatoric for the next conditions' calculation stands as has been before in the CLK branch (here the elsif branch).
Generally it is applicable: the last elsif condition contains the CLK query (exactly NAME'event and NAME='value', it must be added that 'value' can be 0 or 1). All previous 'if' and 'elsif' condition signals have to be also in the sensitivity list, because their queries can occur asynchronous to the cycle.
Important : If these signals are not contained in the sensitivity list, something else will be simulated than that what would be synthesised, because the sensitivity list is on principle all the same to the synthesis tools. They only comply with the structure of the VHDL code within the process.

4.2.11 Rules for clocked processes

```
process
begin
   wait until CLK'event and CLK='1';
   if RESET = '1' then    -- synchron RESET
     -- Register reset
   else
     -- combinatorics
   end if;
end process;
```

Wait-form:

- **no sensitivity list**

Wait and If-form:

- **All signals which receive an assignment -> Register**

```
process(CLK, RST)
begin
  if (RST = `1`) then     -- asynchron RESET
    -- Register reset
  elsif (CLK`event and CLK=`1`) then
    -- combinatorics
  end if;
end process;
```

If-form:

- **Only Clock and Reset in sensitivity list (and all the other asynchronous Signals)**

```
process(CLK)    -- no RESET
begin
  if (CLK`event and CLK=`1`) then
    -- combinatorics
  end if;
end process;
```

4.2.12 Questions

	20. Which signals in clocked processes are used for deducing registers ?
○ yes ○ no	20.1. Temporary signals.
○ yes ○ no	20.2. Signals which contain an assignment.
○ yes ○ no	20.3. Signals which have been read.

submit reset

4.2.13 Questions

	21. Which two process types are permitted in the RTL - style ?
○ yes ○ no	**21.1. Mixed and analog processes.**
○ yes ○ no	**21.2. Combinational and sequential processes.**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

4.2.14 Questions

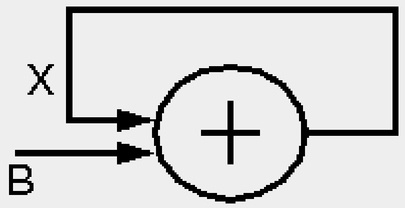
	22. What causes latches to be created in the synthesized design ?
○ yes ○ no	**22.1. Concurrent IF-statements.**
○ yes ○ no	**22.2. Forgotten else-paths.**
○ yes ○ no	**22.3. Signal assignments which are not executed in all paths of an IF- or CASE-statement .**
○ yes ○ no	**22.4. Incomplete sensitivity lists.**
	23. How can unwanted latches be prevented most efficiently ?
○ yes ○ no	**23.1. By replacing IF-statements by CASE-statements.**
○ yes ○ no	**23.2. By giving default assignments to all signals which contain an assignment in one path, before the IF- or CASE-statement.**

submit reset

Please answer the questions by clicking "Yes" or "No". Then press "submit" to verify your answers, or "reset" to clear your selections.

4.3 Combinational Logic

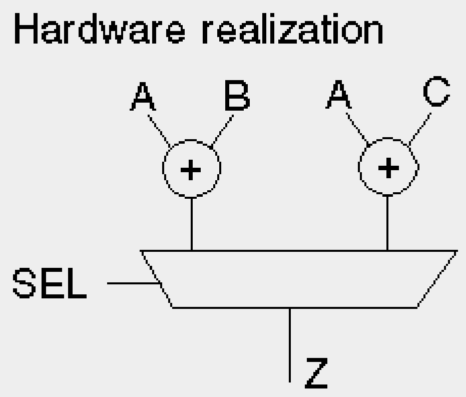
```
architecture EXAMPLE of FEEDBACK is
  signal B,X : integer range 0 to 99;
begin
  process (X, B)
  begin
    X <= X + B;
  end process;

 . . .
end EXAMPLE;
```

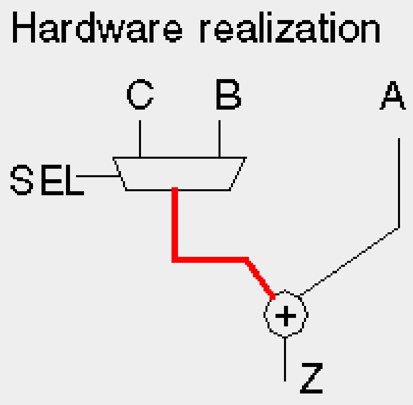


Do not create combinational feedback loops!

When modeling purely combinational logic, it is necessary to avoid combinational feedback loops. A feedback loop triggers itself all the time, i.e. the corresponding process is always active. In the example, this results in a perpetual addition, i.e. X is increased to its maximum value. So simulation quits at time 0 ns with an error message because X exceeds its range. In general, synthesis is possible, yet the hardware is not useable.

4.3.1 Coding Style Influence

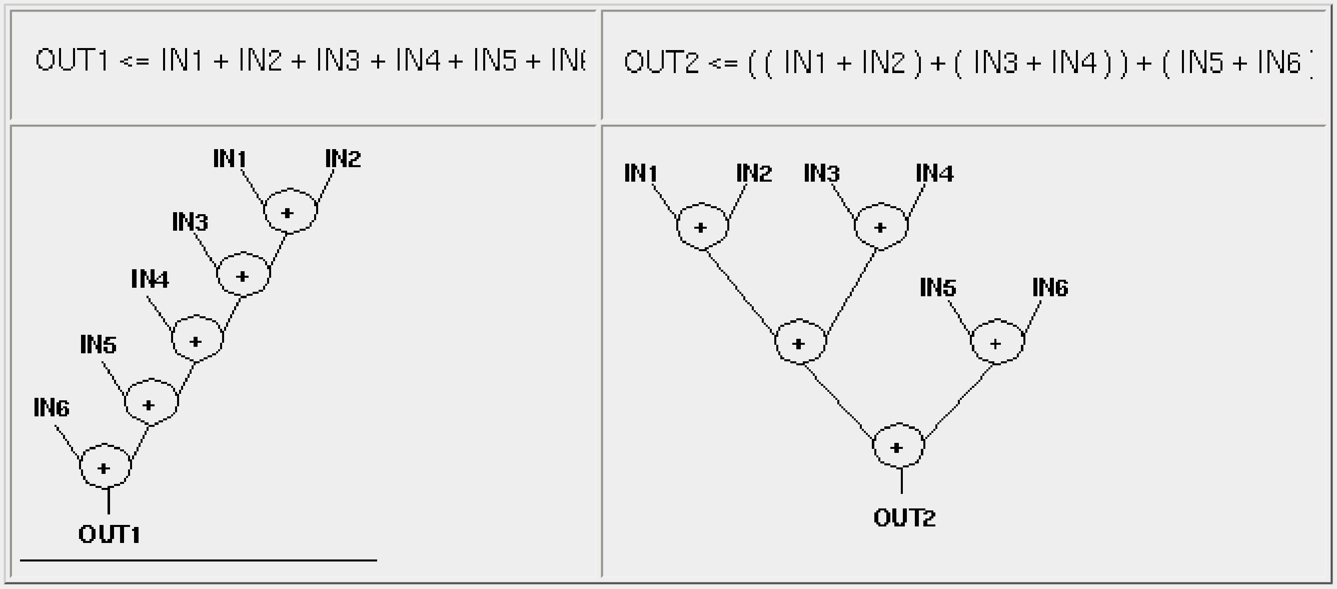
Direct implemen-tation	EXAMPLE1:
	process (SEL,A,B)
	begin
	
	  if SEL = \`1\` then
	    Z <= A + B;
	  else
	    Z <= A + C;
	  end if;
	
	end process EXAMPLE1;
		 Hardware realization
		A
		B
		A
		C
		+
		+
		SEL
		Z

Manual resource sharing	EXAMPLE2:
	process (SEL,A,B)
	  variable TMP : bit;
	begin
	  if SEL = \`1\` then
	    TMP := B;
	  else
	    TMP := C;
	  end if;
	  Z <= A + TMP;
	end process EXAMPLE2;
		

An IF statement is synthesized to a multiplexer with eventual additional logic. That is the reason why the direct implementation of example 1 results in two adders as this is exactly what the VHDL code describes. But it is obvious that one adder is sufficient to implement the desired functionality and good synthesis tools will detect this during their optimization cycles. In example 2 a temporal variable is used to implement a functionally equivalent description which requires only one adder. Manual resource sharing is recommended as it leads to a better starting point for the synthesis process.

4.3.2 Source Code Optimization

- **An operation can be described very efficiently for synthesis, e.g.:**



```
OUT1 <= IN1 + IN2 + IN3 + IN4 + IN5 + IN
```

```
OUT2 <= ( ( IN1 + IN2 ) + ( IN3 + IN4 ) ) + ( IN5 + IN6
```

- **In one description the longest path goes via five, in the other description via three addition components - some optimization tools automatically change the description according to the given constraints.**

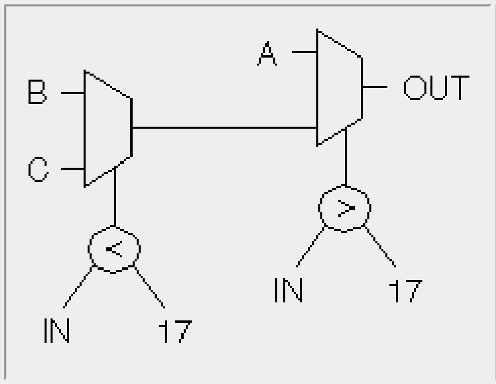
The structure of the generated hardware, at least in the first synthesis iteration, is determined by the VHDL code itself. Consequently, the coding style has a rather big impact on the optimization algorithms. As not all synthesis tools are able to optimize the design structure itself, it is reasonable to ease their task, e.g. by structuring the code for minimum critical paths.

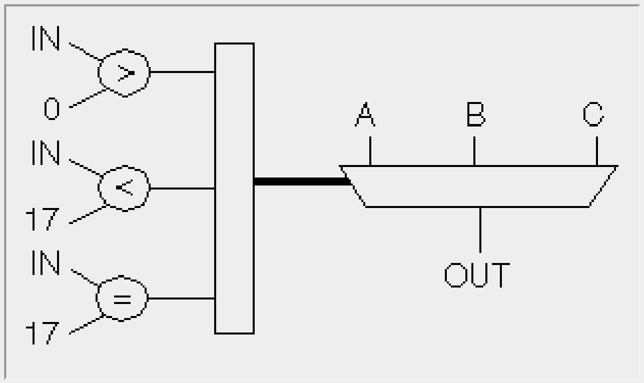
4.3.3 IF structure <-> CASE structure

- **Different descriptions are synthesized differently**

```
. . .
if (IN > 17) then
  OUT <= A ;
elsif (IN < 17) then
  OUT <= B ;
else
  OUT <=  C ;
end if ;
. . .
```

```
. . .
case IN is
  when 0 to 16 =>
      OUT <= B ;
  when 17 =>
      OUT <= C ;
  when others =>
      OUT <= A ;
end case ;
. . .
```

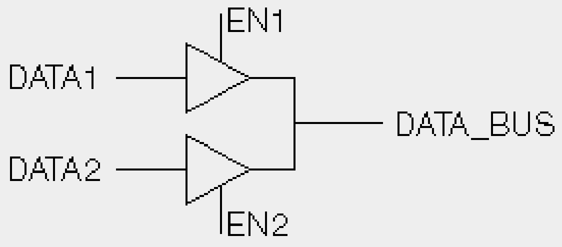




While algorithms can take care of some clumsy VHDL constructs, other model aspects can not be changed during synthesis. The use of IF constructs, for example, implies different levels of priority, i.e. they infer a hierarchical structure of multiplexers. In CASE statements, however, the different choice options do not overlap and a parallel structure with a single switching stage is the result.

4.3.4 Implementation of a Data Bus

```
entity TRISTATE is
  port(DATA1, DATA2 : in   std_ulogic;
       EN1, EN2     : in   std_ulogic;
       DATA_BUS     : out std_logic );
end TRISTATE;
```



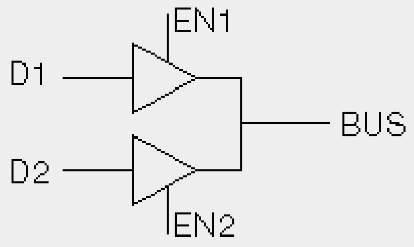
```
architecture RTL1 of TRISTATE is
begin
  process (DATA1, EN1)
  begin
      if EN1 = '1' then
        DATA_BUS <= DATA1;
      else
        DATA_BUS <= 'Z';
      end if;
  end process;

  process (DATA2, EN2)
  begin
      if EN2 = '1' then
        DATA_BUS <= DATA2;
      else
        DATA_BUS <= 'Z';
      end if;
  end process;
end RTL1;
```

```
architecture RTL2 of TRISTATE is
begin
  DATA_BUS <= DATA1 when EN1 = '1' else 'Z';
  DATA_BUS <= DATA2 when EN2 = '1' else 'Z';
end RTL2;
```

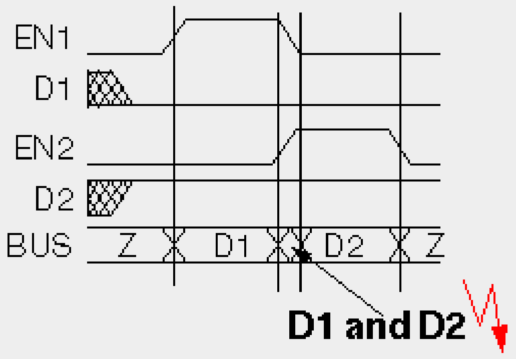
In order to implement a proper internal bus system it must be guaranteed that only one driver is active while all others are set to high impedance, i.e. driving 'Z'. Otherwise, if one bus member drives a logic '1' and another drives a logic '0', the current might, depending on the actual technology, increase beyond acceptable levels and probably result in a permanent damage of the device. This overlap of active drivers may be caused by different propagation delays, that means the deactivation of one driver takes more time than the activation of another one. As a consequence two drivers are active. Even if the delays are balanced so that everything works properly, a change to another technology will likely induce problems with the delays.

Problems with Internal Bus Structures

Bus with tristate drivers



Waveform



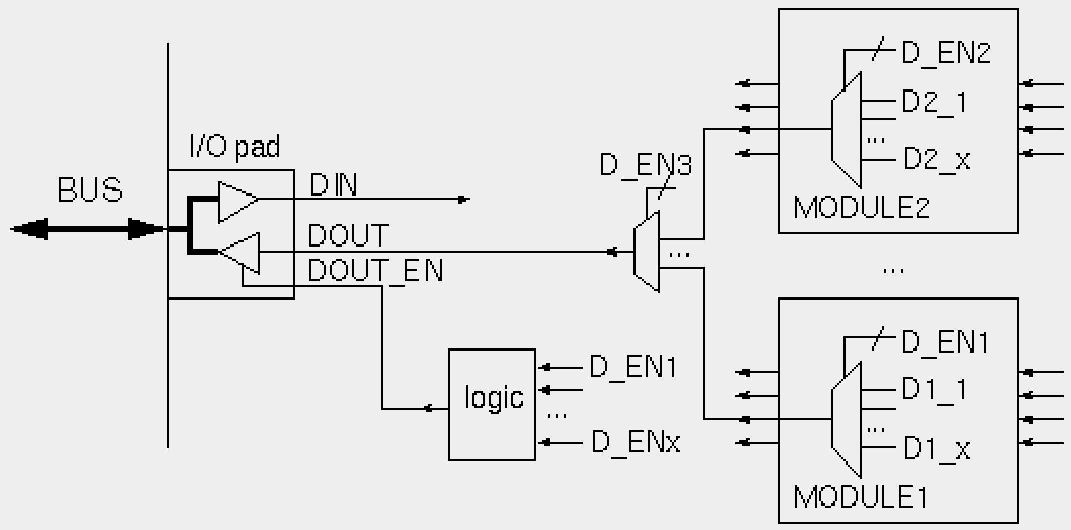
- Different propagation delays
- A bus controller has to guarantee that **at most one driver** is active on the bus
- Technology dependency

Because of the different propagation delays for rising and falling edges one can not assure that only one driver is active at a fixed time. Multiplexers are used to avoid this problem.

Portable and Safe Bus Structure

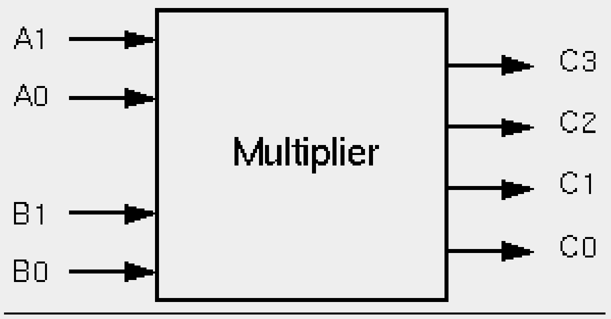
- **Multiplexer instead of tristate driver eliminate internal bus**
- **Three internal signals (DIN, DOUT, DOUT_EN)**

- **Bidirectional I/O pad**
- **Benefit**
  - **Safe circuit**
  - **Portable and testable**



An alternative design structure avoids the problems associated with tristate signals: Multiplexers, driven by the enable signals, guarantee that only one driver exists per signal. Bidirectional signals are eliminated internally by splitting the original bus into two parts. The bidirectional communication with the outside world is done by special I/O pads, i.e. the core structure represents a safe circuit that is fully testable and easily ported to other technologies.

4.3.5 Example of a Multiplier



```
entity MULTIPLIER is
   port (
         A0 : in  bit;
         A1 : in  bit;
         B0 : in  bit;
         B1 : in  bit;
         C0 : out bit;
         C1 : out bit;
         C2 : out bit;
         C3 : out bit);
end MULTIPLIER;
```

- **2 x 2 bit multiplier**
  - **inputs:**
    A1, A0, B1, B0 : 2 bit
  - **outputs:**
    C3, C2, C1, C0 : 4 bit
- **3 different VHDL implementations**
  - **Function table**
  - **Synthesis "by hand" (boolean functions for the outputs)**
  - **Use of VHDL integer types and operators**

Different VHDL coding styles shall be demonstrated with a simple module that has to calculate the result of the multiplication of two 2-bit numbers. The maximum value of each input is 3, i.e. the maximum output value is 9 which needs 4 bits in a binary code. Therefore, four input ports and four output ports of data type 'bit' are required. The same entity shall be used for all different implementations.

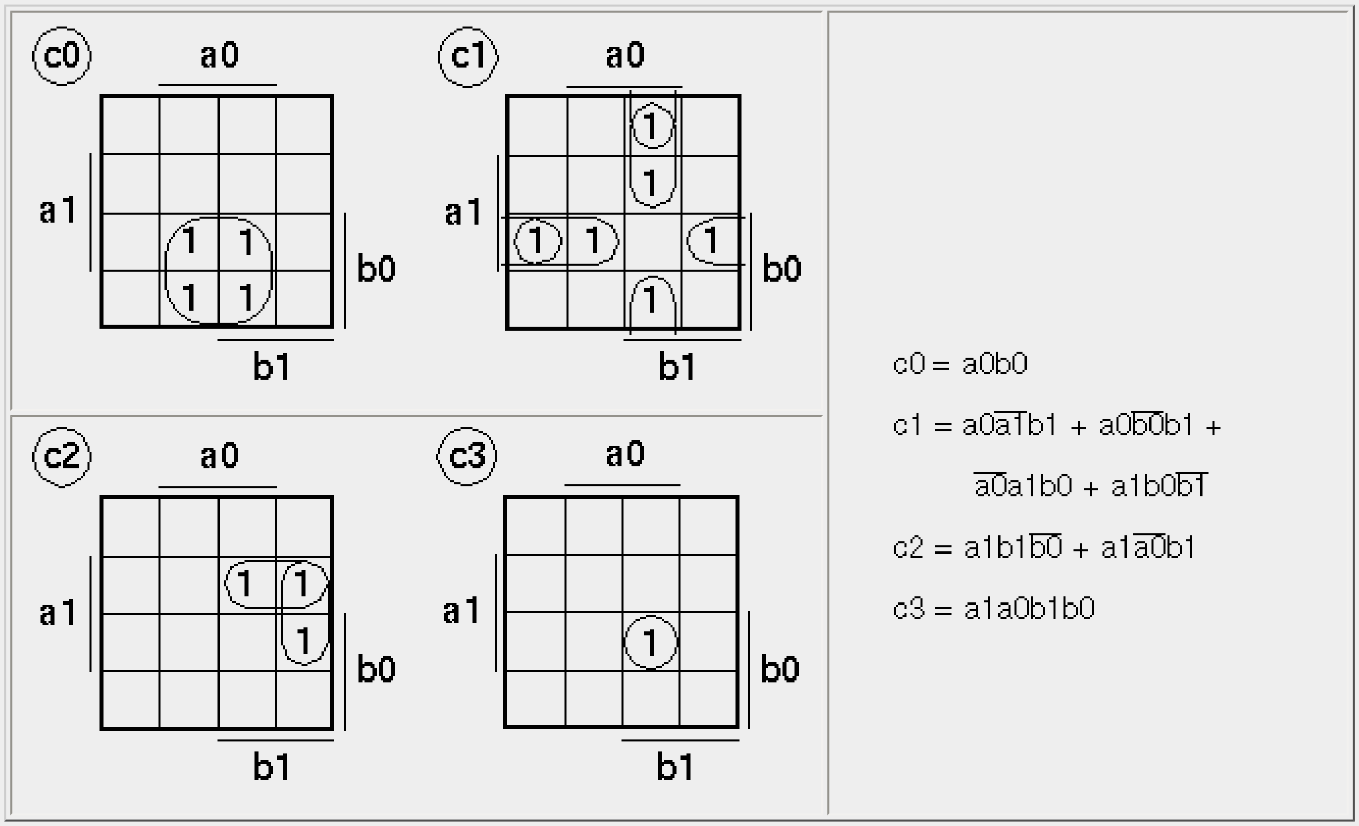
Multiplier Function Table

a1	a0	b1	b0	c3	c2	c1	c0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0

1	1	1	1	1	0	0	1

The most direct approach is via the function table of the multiplications. The behaviour of a combinational logic block is completely defined by listing the result for all possible input values. Of course, the function table is usually no the most compact representation.

Multiplier Minterms -- Karnaugh Diagram



$c0 = a0b0$

$c1 = a0\overline{a1}b1 + a0\overline{b0}b1 +$

$\overline{a0}a1b0 + a1b0\overline{b1}$

$c2 = a1b1\overline{b0} + a1\overline{a0}b1$

$c3 = a1a0b1b0$

The function table of this 2x2 bit multiplier leads directly to the four Karnaugh diagrams of the output signals. The bars on the side of the squares indicate those regions where the corresponding input bit is '1'. All '1's of the output signals are marked in the corresponding diagrams. By combining adjacent '1's, the minimal output function can be derived.

Multiplier: VHDL Code using the Function Table

```
architecture RTL_TABLE of MULTIPLIER is
  signal A_B : bit_vector (3 downto 0);
begin
  A_B <=  A1 & A0 & B1 & B0;

  MULTIPLY : process (A_B)
  begin
    case A_B is
        when "0000" => (C3,C2,C1,C0) <=  "0000";
        when "0001" => (C3,C2,C1,C0) <=  "0000";
        when "0010" => (C3,C2,C1,C0) <=  "0000";
        when "0011" => (C3,C2,C1,C0) <=  "0000";
        when "0100" => (C3,C2,C1,C0) <=  "0000";
        when "0101" => (C3,C2,C1,C0) <=  "0001";
        when "0110" => (C3,C2,C1,C0) <=  "0010";
        when "0111" => (C3,C2,C1,C0) <=  "0011";
        . . .
        when "1100" => (C3,C2,C1,C0) <=  "0000";
        when "1101" => (C3,C2,C1,C0) <=  "0011";
        when "1110" => (C3,C2,C1,C0) <=  "0110";
        when "1111" => (C3,C2,C1,C0) <=  "1001";
    end case;
  end process MULTIPLY;
end RTL_TABLE;
```

- **An internal signal is used that combines all input signals**
- **The internal signal is generated concurrently, i.e. it is updated whenever the input changes**
- **The function table is realized as case statement and thus has to placed within a process. The internal signal is the only signal that controlls the behaviour.**

Here the function table is coded in VHDL within a CASE statement.
To be able to examine all inputs at once in the CASE statement the signal are contatenated to a new signal 'A_B' within a concurrent signal assignment. The process which reads the new signal 'A_B' must have this signal in the sensitivity list. As the concurrent signal assignment to 'A_B' is sensitive to the input signals, the process sensitive to 'A_B' is indirectly sensitiv to the input signals. At least the signal A_B is temporary and will not be visible in the synthesised circuit.

Multiplier: Minterm Conversion

```
architecture RTL_MINTERM
of MULTIPLIER is
begin

    C0 <= A0 and B0;

    C1 <= (A0 and not A1 and B1) or
              (A0 and not B0 and B1) or
              (not A0 and A1 and B0) or
              (A1 and B0 and not B1);

    C2 <= (A1 and B1 and not B0) or
               (A1 and not A0 and B1);

    C3 <= A1 and A0 and B1 and B0;

end RTL_MINTERM;
```

- **The minterm functions are realized directly as concurrent statements**

Here the output functions are written in VHDL. Principle we have synthesised the circuit by ourself as we have executed every step of the synthesis.

Multiplier: Integer Realization

```
library IEEE;
use IEEE.NUMERIC_BIT.all;

architecture RTL_INTEGER
of MULTIPLIER is
    signal A_VEC, B_VEC: unsigned(1
downto 0);
    signal A_INT, B_INT:     integer
range 0 to 3;

    signal C_VEC: unsigned (3 downto
0);
    signal C_INT:   integer range 0 to 9;
begin
    A_VEC <= A1 & A0;
    A_INT <= TO_INTEGER(A_VEC);
    B_VEC <= B1 & B0;
    B_INT <= TO_INTEGER(B_VEC);

    C_INT <= A_INT * B_INT;
    C_VEC <= TO_UNSIGNED(C_INT,
4);

    (C3, C2, C1, C0) <= C_VEC;
end RTL_INTEGER;
```

- **The NUMERIC_BIT package provides all necessary functions to convert bit vectors to integer values and vice versa**
- **Internal signals are used to generate bit vectors and integer representations of the port signals. The bit vectors shall be treated as unsigned binary values.**
- **The single bit input signals are concatenated to vectors and converted to integer data types**
- **The multiplication is realized via the standard VHDL operator**
- **The size of the target vector must be specified when converting integers back to bit vectors**

- Finally, the bit vector elements are assigned to the output ports

Last, the bits are converted to integer values and the multiplication is implemented with the aid of the VHDL multiplication operator '*'. Internal signals are used to hold the data values in different data formats (vectors, integers).

The minterm realization is very tedious and is performed by the synthesis tool automatically when the function table is parsed. The most elegant solution is the integer implementation as the function of the code is clearly visible and not hidden in boolean functions or in hardcoded values like in the other examples. The use of 'bit' type ports, however, is very awkward. It is better style to use 'unsigned' bit vectors or 'integer'. Additionally, the conversion of the bit vectors to 'integer' is not necessary as the arithmetic operators, including '*' are overloaded in the 'numeric_bit' package.

The synthesis result should be identical in all three cases.

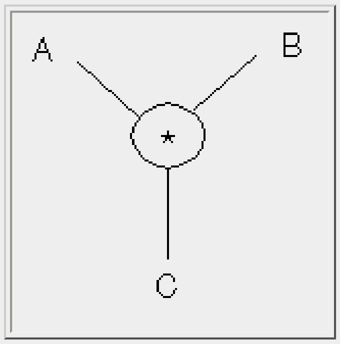
4.3.6 Synthesis of Operators

- **Operator structure**
  - **Discrete gates**
  - **Macro cell from the library**
- **Operator architecture (e.g. ripple-carry, carry-look-ahead etc.)**
  - **Specific comments for the synthesis tool contained in the VHDL code**
  - **Optimization based on time- / surface-defaults**

Based on the operator symbol, the synthesis knows about the desired functionality. Depending on the target technology and the corresponding library elements either a sort of submodule which performs the necessary operations is created out of standard cells, or a macro cell that has already been optimized by the manufacturer is instantiated in the netlist. If alternative implementations exist, e.g. ripple-carry or carry-look-ahead, the decision will be made according to the given speed and area constraints. Sometimes, the user may influence the synthesis process via tool options or special VHDL comments that are evaluated by the software.

Synthesis Results

- **Step 1: Conversion into a generic netlist**
  - **VHDL -> either boolean equations (more or less complicated)**
    or the tool recognizes a complex function
- **Step 2: Optimization**
- **Depends on the tool: all three examples provide the same result**



4.3.7 Example of an Adder

```
entity ADD is
   port (A, B  : in integer range 0 to 7;
            Z      : out integer range 0 to 15);
end ADD;

architecture ARITHMETIC of ADD is
begin
    Z <= A + B;
end ARITHMETIC;
```

Package with "+" functions

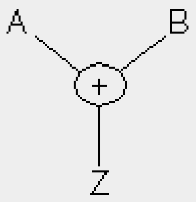
```
library VENDOR_XY;
use VENDOR_XY.p_arithmetic.all;

entity MVL_ADD is
   port (A, B : in mvl_vector (3 downto 0);
            Z : out mvl_vector (4 downto 0) );
end MVL_ADD;

architecture ARITHMETIC of MVL_ADD is
begin
    Z <= A + B;
end ARITHMETIC;
```

Notice:

Advantages of a range declaration with integer types:

a) During simulation: check for "out of range..."

b) During synthesis: only 4 bit bus width



You could follow the same steps shown above with the multiplier when implementing an adder. But as we already know the smartest way of coding an adder in VHDL is to use integer types and the operator '+'. The number of bits per port signal is determined by the range constraint. The operator "+" can be transferred into the function table and into boolean expressions, i.e. gates, by a synthesis tool.
It is more complicated when using self defined data types (non integer). The VHDL standard as stated in the LRM (Language Reference Manual) predefines such arithmetic operators only for INTEGER and REAL data types. Therefore it is necessary to describe a new "+" operator in VHDL.
This new function (with the same name "+", but with new parameters) might look like:

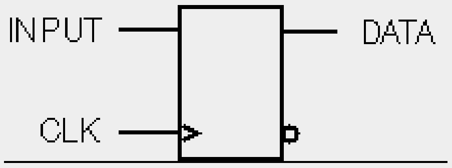
```
function "+"(L: mvl_vector; R: mvl_vector) return mvl_vector is
constant length: INTEGER := R'length + 1;
```

```
begin
     -- calculate result = L + R
    return result;
end;.
```

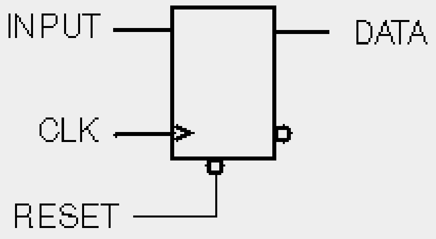
4.4 Sequential Logic

- **A reset mechanism is required in hardware to initialize all registers**
- **Asynchronous reset behaviour can be modeled with processes with sensitivity list, only**

```
process
begin
  wait until CLK`event and CLK=`1`;   -- not recommended
    DATA <= INPUT ;
end process ;
```

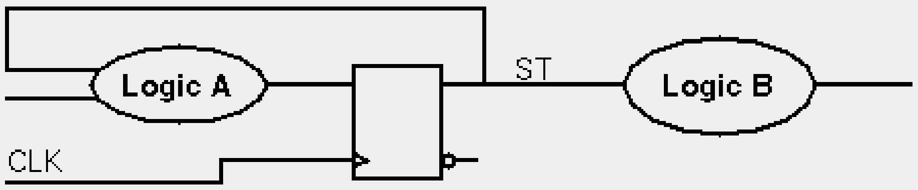


```
process(CLK,RESET)
begin
  if (RESET = `1`) then
    DATA <= `0` ;
  elsif (CLK`event and CLK=`1`) then        -- correct
     DATA <= INPUT ;
  end if ;
end process ;
```



Sequential logic is the general term for designs containing storing elements, especially Flip Flops. While all signals can be initialized prior to simulation by specifying default values, an explicit reset mechanism is necessary to guarantee that the designs behaves the same way whenever it is powered up. Usually, a dedicated reset signal is used for this purpose. Please note that asynchronous behaviour can be modeled with processes with sensitivity list, only, i.e. the process has to react upon the clock and the reset signal.

4.4.1 RTL - Combinational Logic and Registers



```
LOGIC_A: process
begin
wait until CLK`event and CLK=`1`;
--    Logic A
end process LOGIC_A;

LOGIC_B: process (ST)
begin
--  Logic B
end process LOGIC_B;
```

```
LOGIC_AB: process
begin
wait until CLK`event and CLK=`1` ;
-- Logic A and Logic B
end process LOGIC_AB;
```

- **Signal assignments in clocked processes infer Flip Flops**
  - **LOGIC_A: logic + Flip Flops**
  - **LOGIC_B: purely combinational logic**
  - **LOGIC_AB: Flip Flops at the outputs of "Logic A" and "Logic B"**
    => wrong implementation

Additionally, all signals that may receive new values within a clocked process infer Flip Flops. Though it is recommended from a theoretic point of view that registers and combinational logic are modeled with separate processes, it is often convenient to place the calculation of new Flip Flop values in the same process. Postprocessing of register values, however, has to be performed within another process or with concurrent statements.

4.4.2 Variables in Clocked Processes

```
VAR_1: process(CLK)
  variable TEMP : integer;
begin

  if (CLK'event and CLK = '1') then
    TEMP := INPUT * 2;
    OUTPUT_A <= TEMP + 1;
    OUTPUT_B <= TEMP + 2;
  end if;
end process VAR_1;
```

```
VAR_2: process(CLK)
  variable TEMP : integer;
begin

  if (CLK'event and CLK = '1') then
    OUTPUT <= TEMP + 1;
    TEMP := INPUT * 2;
  end if;
end process VAR_2;
```

- **Registers are generated for all variables that might be read before they are updated**
- **How many registers are generated?**

The hardware implementation of variables depends on their use in the process. The VHDL language guarantees that variables still hold their old values when a process is executed again. If this value is not used, however, because it is always updated prior to its use, a storing element will become redundant. Consequently, Flip Flops are infered for variables in clocked processes only if they are read before they will be updated.

In the VAR_1 process, the variable is always set to INPUT*2 whenever an active clock edge is detected. Thus, TEMP is treated as shortcut for the expression and is not visible in the final netlist. In VAR_2, however, the value of TEMP is used to calculate the new value of the OUTPUT signal, i.e. a register bank (integer: at least 32 bit) will be generated.

Example

```
process
   variable LFSR : bit_vector(3 downto 0);
begin
   wait until CLK`event and CLK=`1`;
      LFSR(0) := INPUT;
      LFSR(3) := LFSR(2);
      LFSR(2) := LFSR(1);
      LFSR(1) := LFSR(0);

      OUTPUT <= LFSR(3);
end process;
```

How many registers are generated?

Three FlipFlop will be implemented. One for the signal OUTPUT, which is driven within the clocked process. Two FlipFlops will be implemented for the variables LFSR(2) and LFSR(1), as they are read before they are driven.

4.5 Finite State Machines and VHDL

- **State Processes**
- **State Coding**
- **FSM Types**
  - **Medvedev**
  - **Moore**
  - **Mealy**
  - **Registered Output**

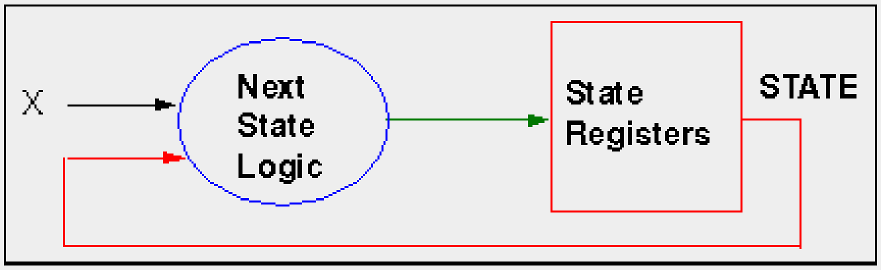
In this chapter, the different types of finite state machines, their graphical representation and ways to model them with VHDL will be shown. Furthermore only synchronous automatons are assumed.

Generally every finite state machine can be described either by one single or by two separated processes. Implementation guidelines and advantages or drawbacks of the different variants will be given.

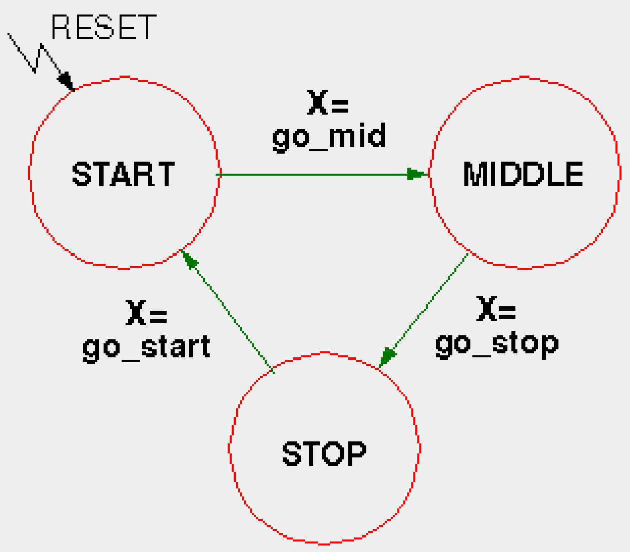
The actual states of a state machine should generally be described by descriptive names. This can be achieved by use of an enumeration type whose values are these names. Later in the synthesis process, these names have to be mapped to a binary representation. This step is called state encoding.

There are several versions of finite state machines. The standard versions known in theory are Medvedev, Moore and Mealy machines. However, there are far more versions than these three. It is for example recommended for several reasons to place storing elements (registers, Flip Flops) at the module outputs. By doing this, additional versions of finite state machines can be build, which will be shown later.

4.5.1 One "State" Process



```
FSM_FF: process (CLK, RESET)
begin
   if RESET='1' then
      STATE <= START  ;
   elsif CLK'event and CLK='1' then
      case  STATE  is
            when  START   => if  X=GO_MID  then
                              STATE <= MIDDLE  ;
                          end if ;
            when  MIDDLE  => if  X=GO_STOP  then
                              STATE <= STOP  ;
                          end if ;
            when  STOP    => if  X=GO_START  then
                              STATE <= START  ;
                          end if ;
            when  others  =>  STATE <= START  ;
         end case ;
   end if ;
end process FSM_FF ;
```



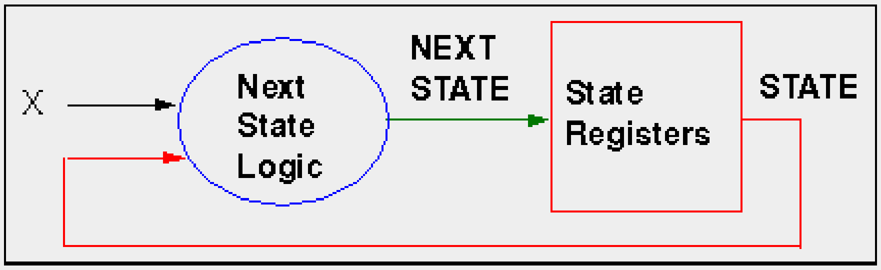
Three different notations of a simple state machine are shown in the picture.

The graphic on the top depicts the automaton as an abstract block diagram which contains only the relevant blocks and signals of interest. The first block (oval) represents the logic of the automaton and the second block (rectangle) the storing elements.

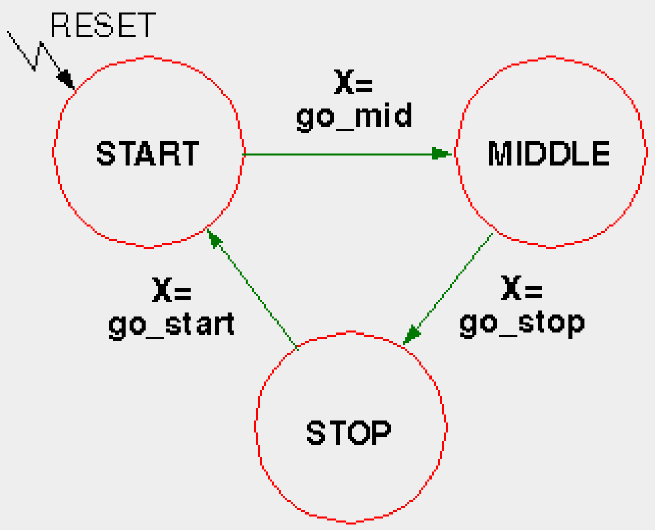
In the bottom right graphic, the automaton is described by a so called bubble diagram. The circles mark the different states of the automaton. If the condition connected to the corresponding transition (arrow) evaluates to 'true' at the time the active clock edge occurs, the automaton will change its state. This is a synchronous behavior. Here the asynchronous reset is the only exception to this behavior. At the time the reset signal gets active, the automat changes to the reset state START immediately.

In the bottom left graphic, the corresponding part of the VHDL source code is shown. The automaton is described in one clocked process. The first IF branch contains the reset sequence. In the second branch, the ELSIF branch, the rest of the automaton is described. In the CASE statement which models the state transitions, the current state of the automaton is detected and it is examined whether input values are present that lead to a change of the state.

4.5.2 Two "State" Processes



```
FSM_FF: process (CLK, RESET)    begin
   if RESET='1' then
       STATE <= START  ;
   elsif CLK'event and CLK='1' then
       STATE  <=  NEXT_STATE  ;
   end if;
end process FSM_FF ;

FSM_LOGIC: process ( STATE , X)
begin
    NEXT_STATE <= STATE ;
   case  STATE  is
        when  START   => if  X=GO_MID  then
                     NEXT_STATE <= MIDDLE  ;
                  end if ;
        when  MIDDLE  => ...
        when  others  =>  NEXT_STATE <= START  ;
      end case ;
end process FSM_LOGIC ;
```



Now, the same automaton is used to show an implementation based on two VHDL processes.

The signal NEXT_STATE is examined explicitly this time. It is inserted in the block diagram between the logic and the storing elements. In the bubble diagram no changes have to be made at this point, as the behaviour remains the same.

The VHDL source code contains two processes. The logic for the NEXT_STATE calculation is described in a separate process. The result is a clocked process describing the storing elements and another purely combinational process describing the logic. In the CASE statement, again, the current state is checked and the input values are examined. If the state has to change, then NEXT_STATE and STATE will differ. With the next occurence of the active clock edge, this new state will be taken over as the current state.

4.5.3 How Many Processes?

- **Structure and Readability**
  - **Asynchronous combinatoric ≠ synchronous storing elements**
    => 2 processes
  - **FSM states change with special input changes**
    => 1 process more comprehensible
  - **Graphical FSM (without output equations) resembles one state process**
    => 1 process
- **Simulation**
  - **Error detection easier with two state processes**
    => 2 processes
- **Synthesis**
  - **2 state processes can lead to smaller generic net list**

and therefore to better synthesis results => 2 processes

Automaton descriptions with either one or two separated processes were shown previously. Depending on the own liking and experiences, either one of the two versions is preferred by desingers.

Generally there are different advantages and disadvantages:

Structure and readability
The VHDL model should represent in a way the hardware which has to be created out of the VHDL source code. So the structure should be mirrored in the VHDL code. As purely combinational logic and storing elements are two different structural elements, these should be separated, i.e. the VHDL source code should be split into two processes.

But one is normally interested in the actual changes of the states of the automaton, only. These changes can then be observed from the outside of the module. The one process description is more adequate for this view. Additionally, the graphical description, which is often used as a specification for the VHDL model, resembles more a one process than a two process description.

Simulation
It will be easier to detect possible errors of the VHDL model in the waveform if one has access to the intermediate signal NEXT_STATE. So the time and location where the error occurs for the first time can be determined exactly and with that the source of the error. The two process version is therefore the better version.

Synthesis
The synthesis algorithms are based on heuristics. Therefore it is impossible to give universally valid statements. But several synthesis tools tend to produce better results (no sophisticated synthesis script assumed), in the meaning of less gate equivalents, when two processes are used to describe the automaton because they are closer related to the hardware structure.

4.5.4 State Encoding

type STATE_TYPE is (**START, MIDDLE, STOP**) ; signal STATE : STATE_TYPE ;	• **State encoding responsible for safety of FSM**
START -> " **00** " MIDDLE -> " **01** " STOP -> " **10** "	• **Default encoding: binary**
START -> " **001** " MIDDLE -> " **010** " STOP -> " **100** "	• **Speed optimized default encoding: one hot**
if {ld(# of states) ≠ ENTIER[ld(# of states)] } => unsafe FSM!	

A finite state machine is an abstract description of digital hardware. It is a synthesis requirement that the states of the automaton are described as binary values or the synthesis tool itself will transform the state names into a binary description on his own. This transformation is called state encoding.

Most synthesis tools select a binary code by default, except the designer specifies another code explicitly. The states of the automaton above could be encoded by a synthesis tool with the values "00", "01" and "10". However other possibilities of state encoding exist. A frequently used code which is needed for speed optimized circuits is the "one-out-of-n" code which is also called one hot code. Here, one bit is used for every state of the automaton. E.g. if the automat has 11 states, then the state vector contains 11 bits. The bit of the vector which is set to '1' represents the current state of the automaton.

A problem arises for the encoding of the states which can not be ignored: If the automat has n states, one needs ENTIER[ld(n)] Flip Flops for a binary code. This is the smallest integer value bigger or equal to the result of the binary logarithm of n. In the example above, two FFs are needed for a binary code. As the automat consists only of 3 states and two FFs can represent up to 4 states ("00", "01", "10", "11"), there is one invalid state which leads to an unsafe state machine, i.e. the behaviour of the design when accidentally entering this state is not determined.

Ususally, a mechanism has to be provided which corrects the erronous entering of an invalid state.

4.5.5 Extension of Case Statement

```
type STATE_TYPE is (START, MIDDLE, STOP) ;
signal STATE : STATE_TYPE ;
. . .
   case STATE is
        when START    => . . .
        when MIDDLE  => . . .
        when STOP     => . . .

        when others     => . . .

   end case ;
```

- **Adding the "when others" choice**

Not simulatable;
in RTL there exist no other values for STATE

Not necessarily safe;
some synthesis tools will ignore "when others" choice

The most obvious method to intercept invalid states is to insert a 'when others' branch in the CASE statement. With this branch, all values of the examined expression (here: STATE) that are not included in the other branches of the CASE statement are covered. The intention is to intercept all illegal states and to restart the automaton with its reset state (here: START).

VHDL is a very strict language. Therefore, during simulation, only the values defined by the type definition of a signal can be accessed. For this reason, invalid states do not exist in the simulation and consequently can not be simulated. Furthermore the synthetisized circuit is also not necessarily safe. Some synthesis tools ignore the 'when others' branch as, by definition, there are no values to cover in this branch.

Inserting a 'when others' branch into the case statement is not a good solution for creating a safe state machine.

4.5.6 Extension of Type Declaration

```
type STATE_TYPE is (START, MIDDLE, STOP, DUMMY) ;
signal STATE : STATE_TYPE ;
...
   case STATE is
        when START    => ...
        when MIDDLE  => ...
        when STOP      => ...

        when DUMMY     => ...      -- or when others

   end case ;
```

- **Adding dummy values**
- **Advantages:**
  - **Now simulatable**
  - **Safe FSM after synthesis**

{2(ENTIER [ld(n)]) -n} dummy states (n=20 => 12 dummy states)**

Changing to one hot coding => unnecessary hardware (n=20 => 12 unnecessary FlipFlops)

The second way is to define additional values for the enumeration type. So many values have to be added that after state encoding invalid values can no longer occur. If an automaton contains for example 20 states, 5 Flip Flops are needed with a binary code. With 5 FFs one can distinguish 32 values (2^5=32). Thus, 12 additional values have to be added to the enumeration type of the state machine.

By adding additional values to the enumeration type, one gets a state machine whose behaviour in case of errors can now be simulated. The synthesis also results in a safe circuit representing the original state machine. However this method is somewhat awkward as one has to insert many so called dummy states eventually. Furthermore, this method is only suitable when binary coding for the states of the automaton is used. If one has added for example these 12 additional values for a safe state machine, he will get 12 redundant Flip Flop when he switches the state encoding to the one hot code as an extra bit is needed for every state.

It is impossible to make a state machine safe by inserting additional values for dummy states if a one hot code is used. Every new value would lead to an additional FF and therefore would noly increase the amount of invalid state values after synthesis.

Therfore the method of inserting additional values into the enumeration type is not a good solution, too, as it is applicable to binary state encoding, only.

4.5.7 Hand Coding

```
subtype STATE_TYPE is std_ulogic_vector (1 downto 0) ;
signal STATE : STATE_TYPE ;

constant START   : STATE_TYPE := "01";
constant MIDDLE : STATE_TYPE := "11";
constant STOP     : STATE_TYPE := "00";
...
   case STATE is
         when START     => ...
         when MIDDLE  => ...
         when STOP      => ...
         when others     => ...
   end case ;
```

- **Defining constants**
- **Control of encoding**
- **Safe FSM**
- **Simulatable**
- **Portable design**
- **More effort**

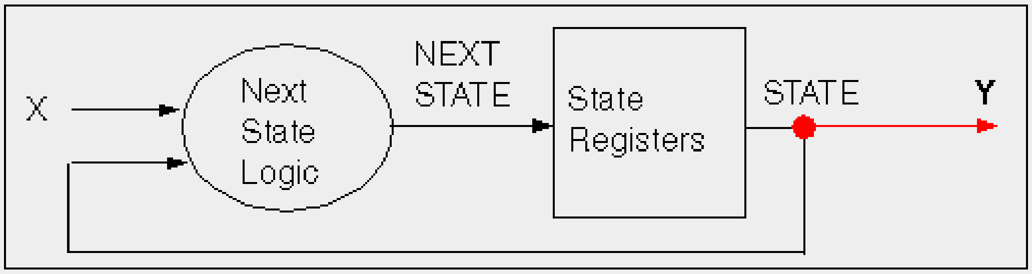
The best method of state encoding is hand coding, i.e. the designer decides by himself which code will be used.

This is done by using a vector type instead of an enumeration type. This vector type can based upon the 'std_(u)logic_vector' type for exmple. The width of this vector depends on the code chosen. The state signal is now of this vector type, which is the reason for the term "state vector".

In the next step, constants are defined which represent the corresponding states of the automaton. These constants are set to the state vector values according to the selected code. With these constants, the code can be fixed by the designer and can not be altered by the synthesis tool. This VHDL model is also 100 percent portable. The behaviour in case of errors can be verified in a simulation as the state vector can assume all values that might occur in real hardware, now.

The only drawback to mention is a little more effort in writing the VHDL code. This is especially true when the code is changed. The hand coding alternative is the best method to design a safe finite state machine and is furthermore portable among different synthesis tools.

4.5.8 FSM: Medvedev



- **The output vector resembles the state vector: Y = S**

Two Processes	One Process

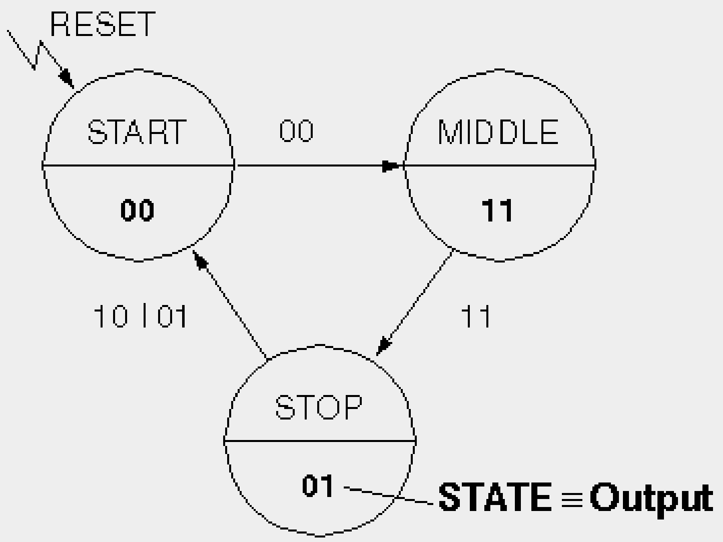
architecture RTL of MEDVEDEV is
  ...
begin
  REG: process (CLK, RESET)
  begin
    -- State Registers Inference
  end process REG ;
 	
  CMB: process (X, STATE)
  begin
    -- Next State Logic
  end process CMB ;
 	
  Y <= S ;
end RTL ;
	architecture RTL of MEDVEDEV is
	  ...
	begin
	
	  REG: process (CLK, RESET)
	  begin
	    -- State Registers Inference with Logic Block
	  end process REG ;
	
	  Y <= S ;
	
	end RTL ;

The difference between the three types of state machines known in theory (Medevedev, Moore and Mealy machines) is the way the output is generated.

In the Medvedev machine the value of the output is identical with the state vector of the finite state machine. That means, the logic for the output consists only out of wires; namely the connection from the state vector registers to the output ports. This is done in VHDL by a simple signal assignment which is shown in the example above. Concurrent assigments are used here.

4.5.9 Medvedev Example

```
architecture RTL of MEDVEDEV_TEST is
   signal STATE,NEXTSTATE : STATE_TYPE ;
begin
    REG: process (CLK, RESET)
   begin
      if RESET=`1` then
         STATE <= START ;
      elsif CLK`event and CLK=`1` then
         STATE <= NEXTSTATE ;
      end if ;
   end process REG;
    CMB: process (A,B,STATE) begin
      NEXT_STATE <= STATE;
      case STATE is
         when START  => if (A or B)=`0` then
                     NEXTSTATE <= MIDDLE ;
                  end if ;
         when MIDDLE => if (A and B)=`1` then
                     NEXTSTATE <= STOP ;
                   end if ;
         when STOP    => if (A xor B)=`1` then
                     NEXTSTATE <= START ;
                   end if ;
         when others => NEXTSTATE <= START ;
      end case ;
   end process CMB ;
   -- concurrent signal assignments for output
    (Y,Z) <= STATE ;
end RTL ;
```



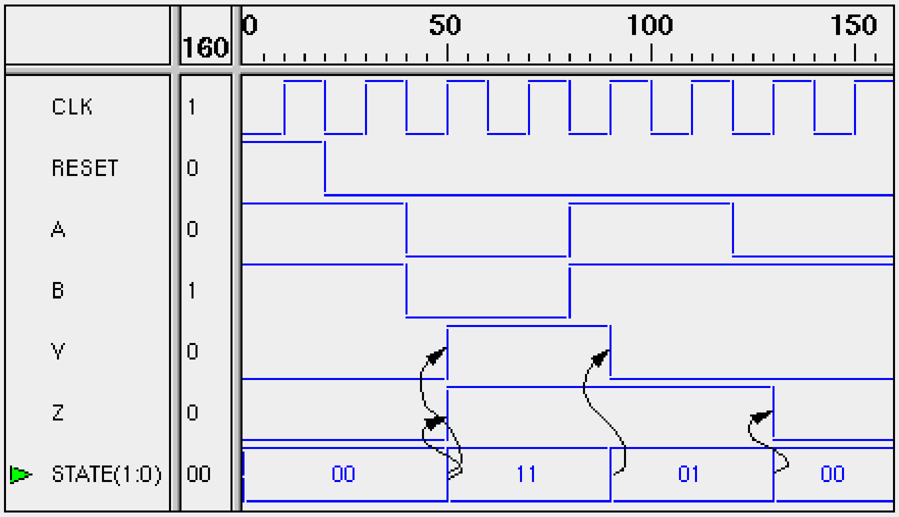
```
subtype STATE_TYPE is std_ulogic_vector(1 downto 0);
constant START    : STATE_TYPE := "00";
constant MIDDLE   : STATE_TYPE := "11";
constant STOP     : STATE_TYPE := "01";
```

Here, an example of a Medvedev machine is shown.

The bubble diagram contains the states of the machine (START, MIDDLE, STOP), the state encoding ("00", "11", "01"; see also the constant declarations) and the state transitions. The so called weights (labels) of the transitions (arrows) determine the value of the input vector (here the signals A and B) for which the corresponding state transition will be executed. For '10 | 01', the state transition is executed when the input vector has either the value "10" or the value "01".

The functionality of the state machine is described in the VHDL source code on the left side. The version with two processes was selected. One can see that the output vector is wired with the state vector by a concurrent signal assignment.

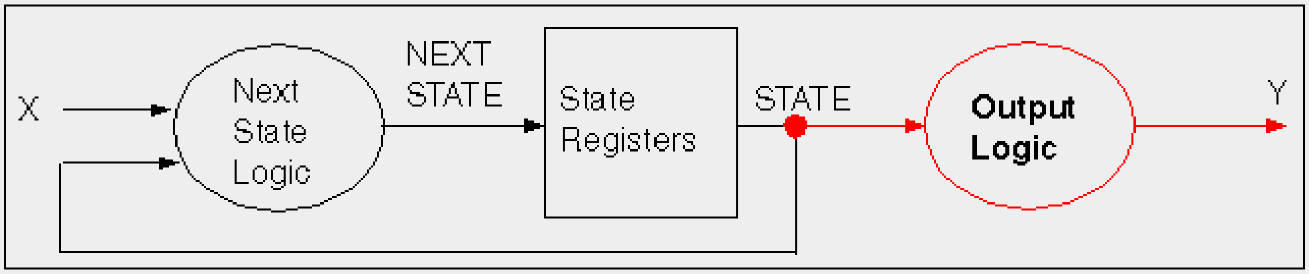
4.5.10 Waveform Medvedev Example



- **(Y,Z) = STATE => Medvedev machine**

In the waveform one can see the progression over time of the signal values of the design during simulation. It is apparent that it must be a Medvedev automaton, because the values of the output vector (represented by the two singals Y and Z) change synchronously with the state vector.

4.5.11 FSM: Moore



- **The output vector is a function of the state vector: Y = f(S)**

Three Processes	**Two Processes**

architecture RTL of MOORE is
...
begin
REG: -- Clocked Process
CMB: -- Combinational Process
OUTPUT: process (STATE)
begin
-- Output Logic
end process OUTPUT ;
end RTL ;
	architecture RTL of MOORE is
	...
	begin
	REG: process (CLK, RESET)
	begin
	-- State Registers Inference with Next State Logic
	end process REG ;
	OUTPUT: process (STATE)
	begin
	-- Output Logic
	end process OUTPUT ;
	end RTL ;

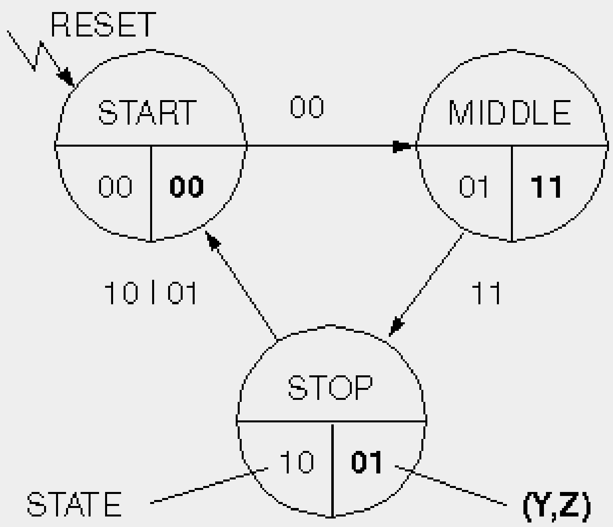
Here, an example of a Moore machine is shown.

The value of the output vector is a function of the current state. This is the reason for the second logic block in the block diagram, located after the storing elements. This logic block holds the hardware which is needed to calculate the output values out of the current state of the automaton.

In the VHDL source code, this logic is implemented with an own combinational process. As the value of the output vector depends on the current value of the state vector, only, no other signals appear in the sensitivity list of the process.

4.5.12 Moore Example

```
architecture RTL of MOORE_TEST is
   signal STATE,NEXTSTATE : STATE_TYPE ;
begin
   REG: process (CLK, RESET) begin
      if RESET=`1` then    STATE <= START ;
      elsif CLK`event and CLK=`1` then
         STATE <= NEXTSTATE ;
      end if ;    end process REG ;
   CMB: process (A,B,STATE) begin
      NEXT_STATE <= STATE;
      case STATE is
        when START => if (A or B)=`0` then
                         NEXTSTATE <= MIDDLE ;
                      end if ;
        when MIDDLE => if (A and B)=`1` then
                         NEXTSTATE <= STOP ;
                      end if ;
        when STOP    => if (A xor B)=`1` then
                         NEXTSTATE <= START ;
                      end if ;
        when others => NEXTSTATE <= START ;
      end case ;     end process CMB ;
   -- concurrent signal assignments for output
   Y <= ‚1` when STATE=MIDDLE else ‚0` ;
   Z <= ‚1` when STATE=MIDDLE
            or STATE=STOP else ‚0` ;
end RTL ;
```

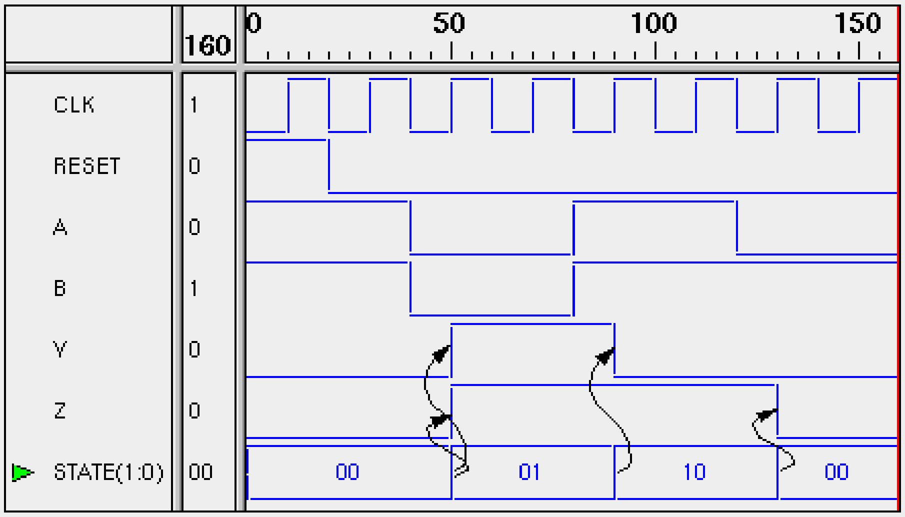


```
subtype STATE_TYPE is std_ulogic_vector(1 downto 0);
constant START    : STATE_TYPE := "00";
constant MIDDLE   : STATE_TYPE := "01";
constant STOP     : STATE_TYPE := "10";
```

Again, the bubble diagram and the corresponding VHDL code are shown; this time for a Moore automaton. The difference to the Medvedev automaton can be recognized in the difference between the state encoding and the corresponding values for the output vector. Both values are specified in the bubbles. The values for the output vector (Y, Z) are the same as in the Medvedev automaton. However the state enconding is now based upon a binary code.

In the VHDL source code, the ouput logic is not contained in a combinational process because of space limitation. Instead, it is implemented via separate concurrent signal assignments. One can see that the output values are calculated out of the state vector values.

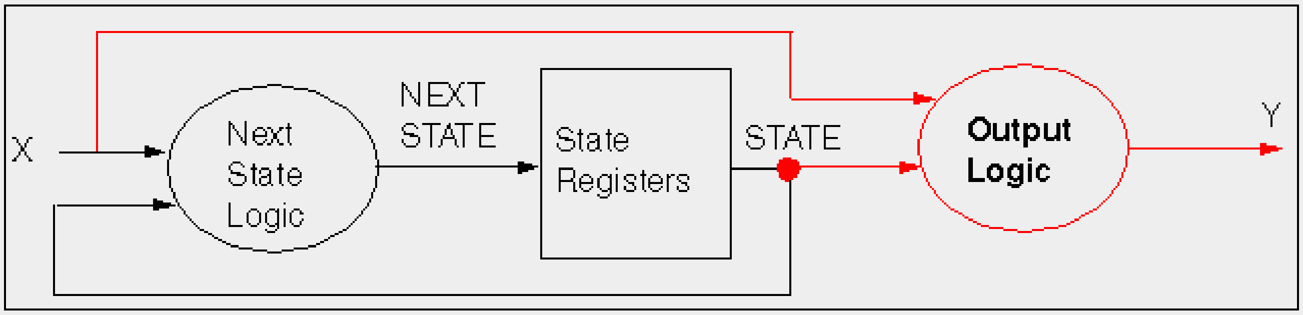
4.5.13 Waveform Moore Example



- **(Y,Z) changes simultaneously with STATE => Moore machine**

Again, the characteristics of the Moore automaton can be seen clearly in the waveform. The values of the output vector change simultaneously with the values of the state vector. But this time the values of the output vector differ from those of the state vector.

4.5.14 FSM: Mealy



- **The output vector is a function of the state vector and the input vector: Y = f(X,S)**

Three Processes	**Two Processes**

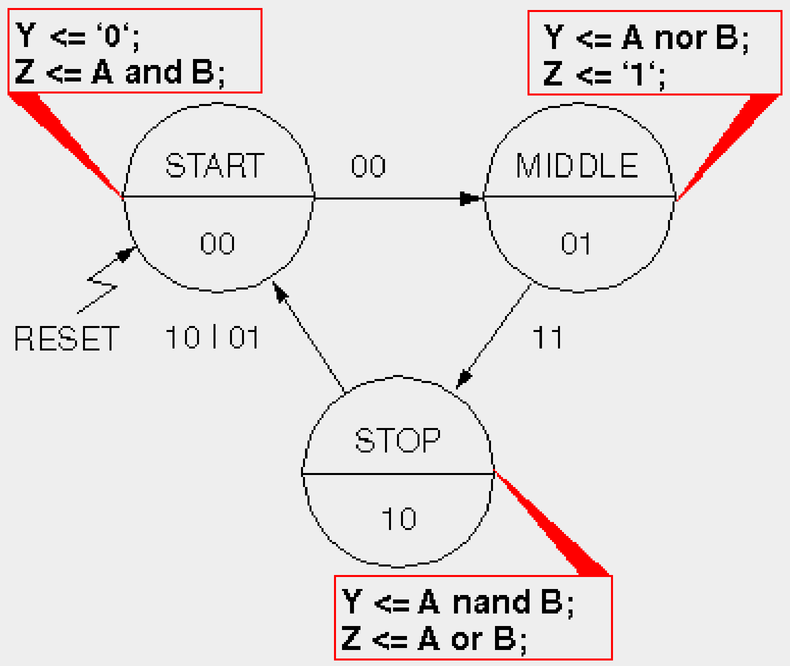
architecture RTL of MEALY is
   ...
begin
   REG: -- Clocked Process
 	
   CMB: -- Combinational Process
 	
   OUTPUT: process (STATE, X)
   begin
      -- Output Logic
   end process OUTPUT ;
end RTL ;
	architecture RTL of MEALY is
	   ...
	begin
	   MED: process (CLK, RESET)
	   begin
	      -- State Registers Inference with Next State Logic
	   end process MED ;
	
	   OUTPUT: process (STATE, X)
	   begin
	      -- Output Logic
	   end process OUTPUT ;
	end RTL ;

Here, a Mealy automat is shown.

The value of the ouput vector is a function of the current values of the state vector and of the input vector. This is why a line is drawn in the block diagram from the input vector to the logic block calculating the output vector. In the VHDL source code, the input vector is now listed in the senisitivity list of the corresponding process.

4.5.15 Mealy Example

```
architecture RTL of MEALY_TEST is
   signal STATE,NEXTSTATE : STATE_TYPE ;
begin
   REG: · · ·   -- clocked STATE process

   CMB: · · ·   -- Like Medvedev and Moore Examples

   OUTPUT: process (STATE, A, B)
   begin
      case STATE is
        when START   =>
                               Y <= `0` ;
                               Z <= A and B ;
        when MIDLLE  =>
                               Y <= A nor B ;
                               Z <= '1' ;
        when STOP    =>
                               Y <= A nand B ;
                               Z <= A or B ;
        when others  =>
                               Y <= `0` ;
                               Z <= '0' ;
      end case;
   end process OUTPUT;
end RTL ;
```

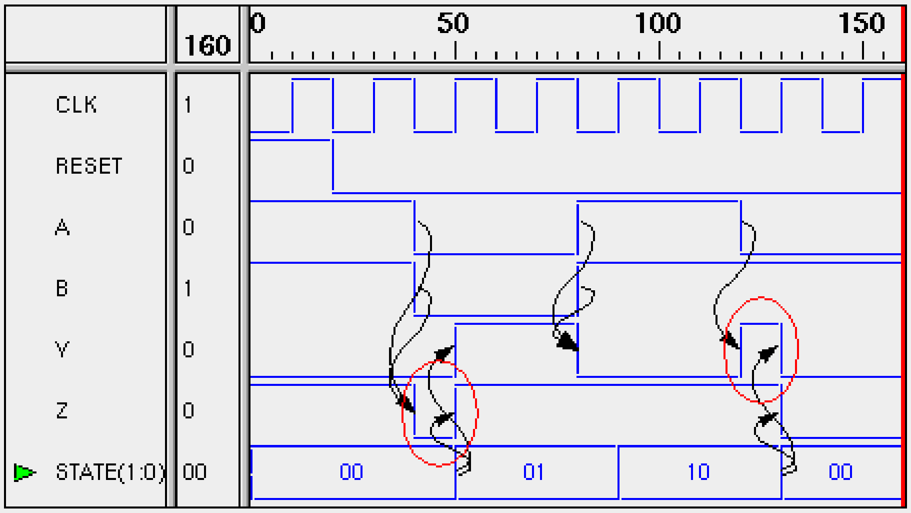


```
subtype STATE_TYPE is std_ulogic_vector(1 downto 0);
constant START     : STATE_TYPE := "00";
constant MIDDLE    : STATE_TYPE := "01";
constant STOP      : STATE_TYPE := "10";
```

In contrast to the other two types of automatons decribed before, the output values can not be simply written into the corresponding state bubble here. Complete functions have to be written down which differ from state to state. These functions are often "hidden" behind the state bubble, instead of explicitly displayed in the graphic.

In the VHDL source code, the calculation of the ouput values is described with concurrent signal assignments, again. One can see that the input signals appear on the right side of the assignments and are therefore part of the output function, now.

4.5.16 Waveform Mealy Example

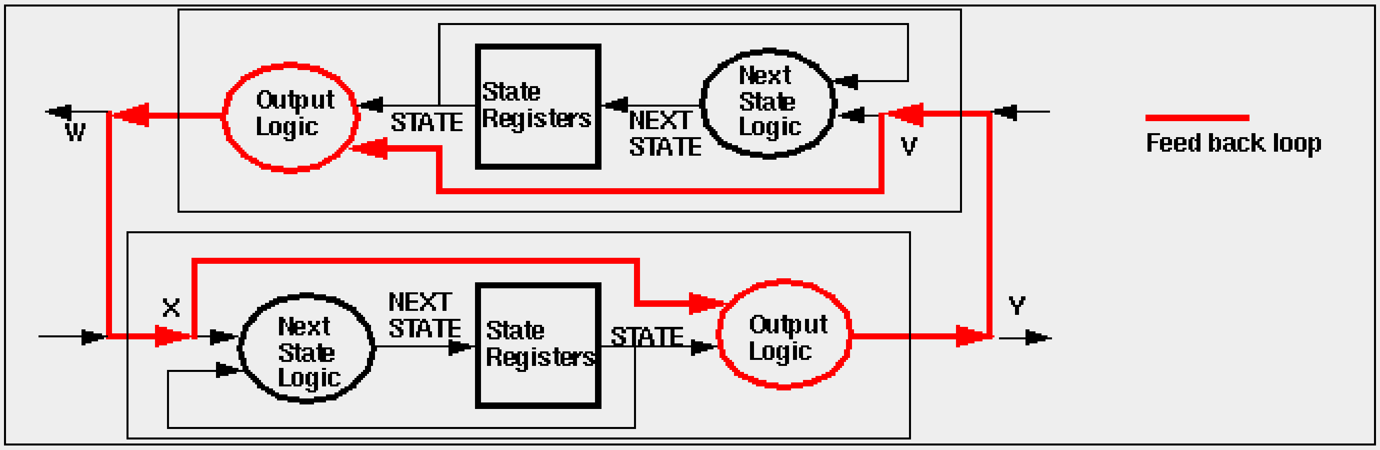


- **(Y,Z) changes with input => Mealy machine**
- **Note the "spikes" of Y and Z in the waveform**
  - **FSM has to be modeled carefully in order to avoid spikes in normal operation.**

Again, one can see the characteristics of the Mealy automaton clearly in the waveform. The most remarkable feature is the fact that the output values change together with the values of the input vector, sometimes. Furthermore, they change together with changes of the state vector values, of course. As one can see, this can lead to so called spikes, i.e. signal pulses with a smaller width than the clock period. This can lead to a misbehaviour in the blocks following thereafter. Of course, this has to be avoided and the designer must take special care when modeling a Mealy automaton in a form similar to the one described here.

4.5.17 Modelling Aspects

- **Medvedev is too inflexible**
- **Moore is preferred because of safe operation**
- **Mealy more flexible, but danger of**
  - **Spikes**
  - **Unnecessary long paths (maximum clock period)**
  - **Combinational feed back loops**



There are different reasons for a designer to use one or other version of the three different automatons.

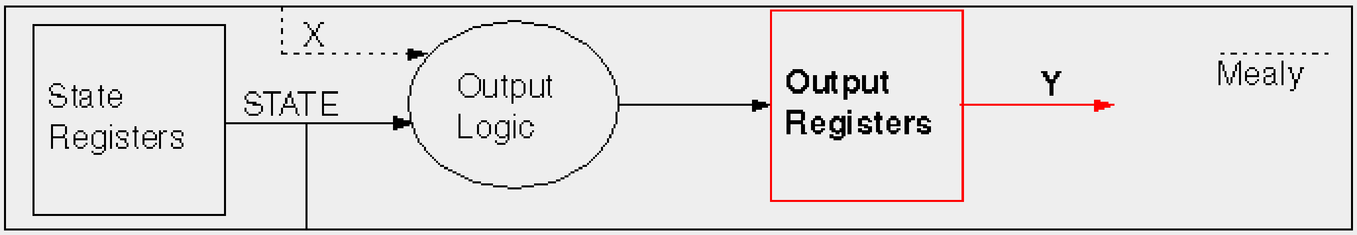
The advantage of the Medvedev automaton is the reduced amount of hardware needed to implement the automaton and the output logic. As the output values are identical with the state vector, no additional combinational logic is needed and the values of the output vector are changed together with the active clock edge. But the designer has to select the code for the state vector by himself. This means that the designer has to put more effort into the design and the design offers no flexibility, which will be negative if the code for the state machine has to be changed.

The Moore automaton is frequently used because this type of automaton is more flexible than the Medvedev automaton and the output calculation depends on the state vector, only. By this pure dependence on the state vector, the output values are calculated in a relatively safe manner, which means, the new values are stable long before the next active clock edge occurs and spikes are avoided. A disadvantage which becomes sometimes relevant is, that a change of the input vector needs on complete clock cycle to affect the output vector (first, the state vector has to change before the output vector can change). Sometimes, this delay of time is unaccetable and consequently the Moore automaton can not be used.

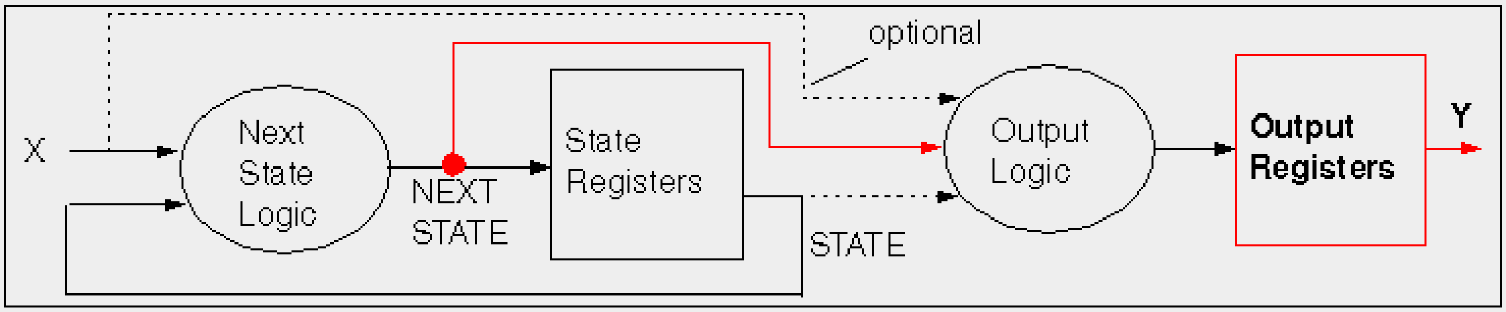
The Mealy automaton is the most flexible of the automatons presented. As the output vector depends on the state vector and the input vector, it can react on every change of a value. But there are also some disadvantages. Spikes can occur, for example, which has been shown on the slide before. If two Mealy automatons are connected in a row then there is the danger of combinational feedback loops, as demonstrated in the picture above. Additionally, long paths are created. Two logic block are connected in a row, so a change of a value needs a relatively long time to propagate through the logic to the next Flip Flops. So one has to choose a relatively large clock period.

4.5.18 Registered Output

- **Avoiding long paths and uncertain timing**
- **With one additional clock period**



- **Without additional clock period (Mealy)**



As shown before, combinational feedback loops can occur if two Mealy automatons are connected in a row. These loops can be avoided if the outputs are "clocked", i.e. if the outputs are connected to Flip Flops. With this, the feedback loop is broken up.

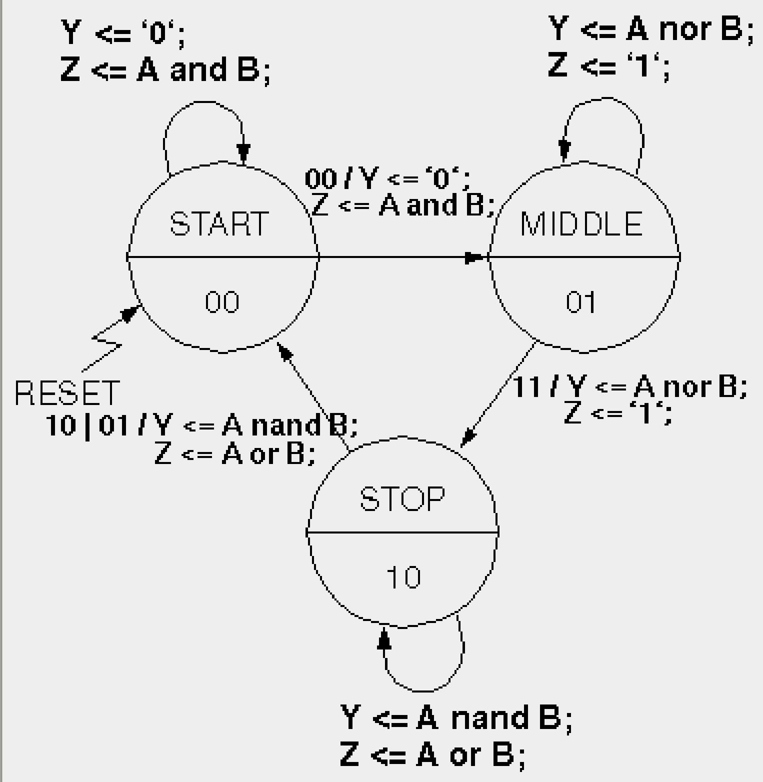
Generally, clocked outputs are used to avoid long paths between Flip Flops and thus assure a safe timing behavior. By clocking all outputs, the output logic is separated from the next logic block of the subsequent module and by this the path through logic elements is shortened and higher clock frequencies are possible. Furthermore, synthesis tools often have problems when optimizing logic paths which pass module boundaries for speed. By clocking the outputs, these problems can be solved. Another advantage is that the successive module can work with safe input data. "Safe" means that the data changes with the active clock edge, only, and thus spikes are ruled out. So the designer of the successive modul has more options in designing the module.

Two versions are known for clocking the output. In the first version, an additional delay of one clock period is created. Here, Flip Flops are simply inserted between the output logic and the state machine outputs without any other changes. By this, the new values of the outputs arrive after the next active clock edge. However, this delay, sometimes, can not be accepted.

In the second version, Flip Flops are present at the state machine output and the output logic uses the NEXT_STATE signal in addition to the state vector. For bigger state machines, this can lead to incomprehensible dependencies and relatively long paths as two logic blocks are connected in a row, again. But this version of a state machine is the most flexible and the fastest (in terms of delay) which provides safe (clocked) output signals.

4.5.19 Registered Output Example (1)

```
architecture RTL of REG_TEST is
   signal Y_I , Z_I : std_ulogic ;
   signal STATE,NEXTSTATE : STATE_TYPE ;
begin

   REG: · · ·   -- clocked STATE process

   CMB: · · ·   -- Like other Examples

   OUTPUT: process (STATE, A, B)
   begin
      case STATE is
        when START   =>
                                    Y_I<= `0` ;
                                    Z_I<= A and B ;
        . . .
   end process OUTPUT

   -- clocked output process
   OUTPUT_REG: process(CLK)    begin
      if CLK'event and CLK='1' then
       Y <= Y_I ;
       Z <= Z_I ;
      end if ;
   end process OUTPUT_REG ;
end RTL ;
```



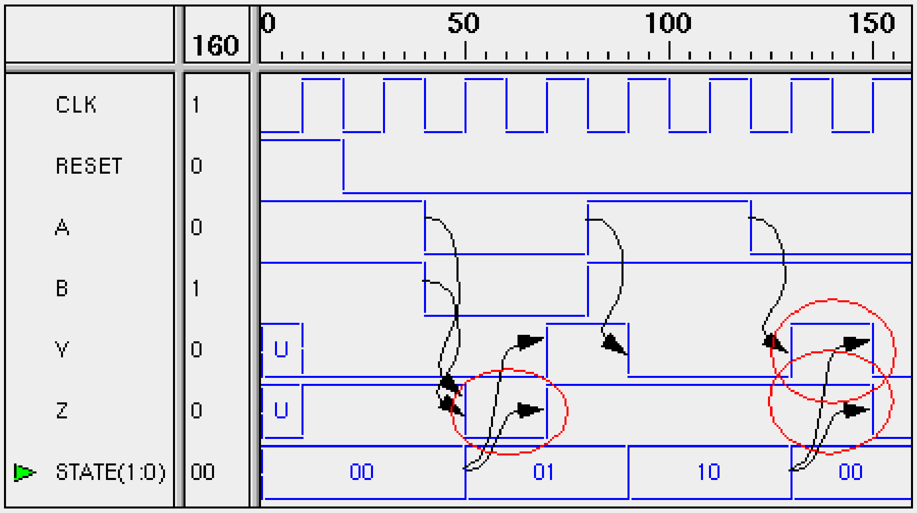
In the picture, the example of the Mealy automat is shown again.

In the bubble diagram, the assignments which were hidden behind the states before are now explicitely connected to self loops. Furthermore, the transitions between states also have signal assignments for the output signals. As transitions take place only when an active clock edge occurs, it is clear that a value is assigned to the output signals with every active clock edge, i.e. Flip Flops have to be provided for the outputs.

As the signal assignments of the old state (which will be exited) are connected to the transitions, the output values depend only on the current (old) state (STATE) but not on the new state (NEXT_STATE). Generally, the signal assignments can be hidden again behind the states in the graphical diagram. For this, so called in-state actions for the self loops and exit-actions for the transitions are used.

In the VHDL source code, intermediate signals (Y_I, Z_I) are evaluated. The values of these signals depend on the input values and the current state. In the following clocked process, the intermediate signals are connected to Flip Flops.

4.5.20 Waveform Registered Output Example (1)



- **One clock period delay between STATE and output changes.**
- **Input changes with clock edge result in an output change.
(Danger of unmeant values**

○)

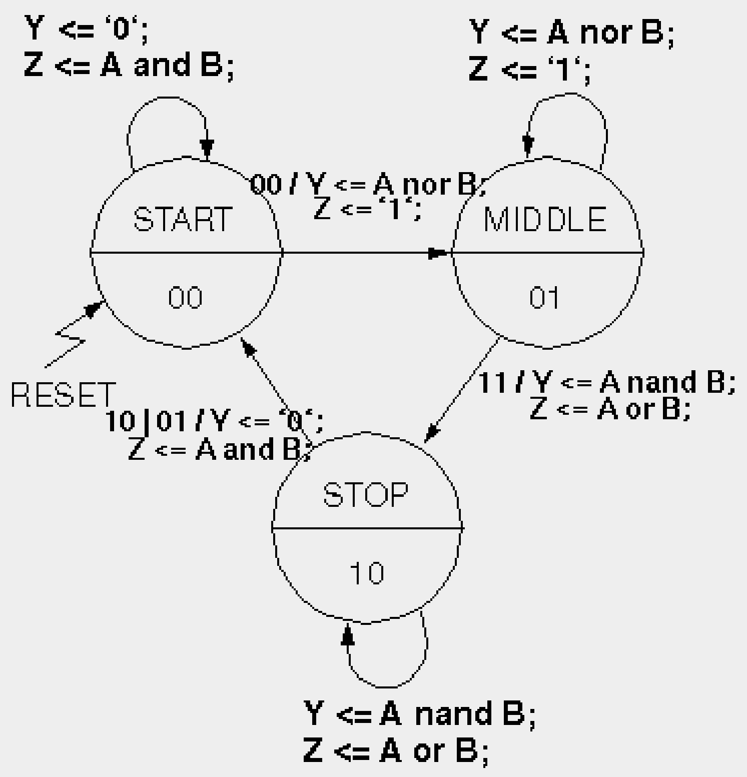
The wavform depicts the simulation results of the clocked Mealy automaton.

It can be sees clearly that the output values change one clock period after the change of the state vector. Furthermore, it can be seen that changes of the input values affect the output values just when the next active clock edge occurs. All value changes of intermediate signals occur only synchronously to the clock so that the state machine has a fixed and well known behaviour.

Still, the modeling has to be carried out carefully. It is essential to consider the delay of one clock period between the changes of the state vector and the output signals. If it is ignored, the values marked in the waveform can lead to undesired behaviour in the subsequent modules.

4.5.21 Registered Output Example (2)

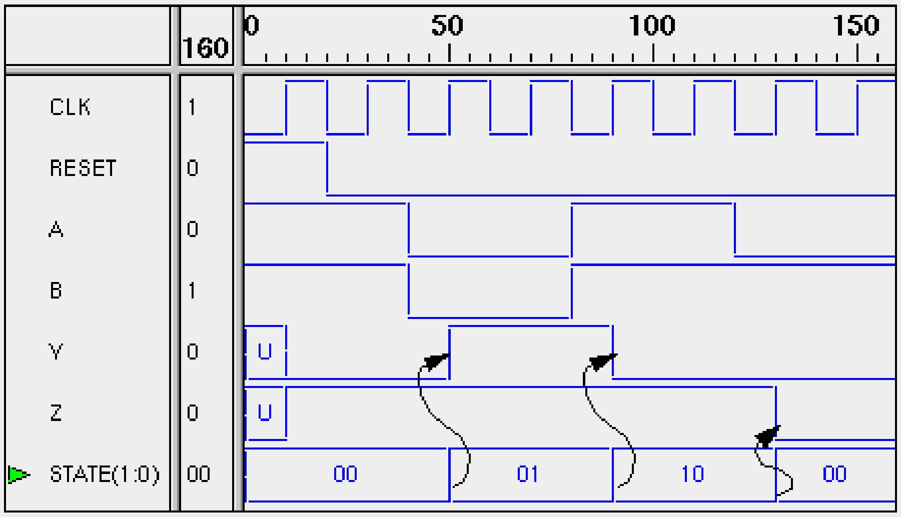
```
architecture RTL of REG_TEST2 is
   signal Y_I , Z_I : std_ulogic ;
   signal STATE,NEXTSTATE : STATE_TYPE ;
begin

   REG: · · ·   -- clocked STATE process

   CMB: · · ·   -- Like other Examples

   OUTPUT: process ( NEXTSTATE , A, B)
   begin
      case NEXTSTATE is
        when START   =>
                                 Y_I<= `0` ;
                                 Z_I<= A and B ;
        . . .
   end process OUTPUT

   OUTPUT_REG: process(CLK)
   begin
      if CLK'event and CLK='1' then
       Y <= Y_I ;
       Z <= Z_I ;
      end if ;
   end process OUTPUT_REG ;
end RTL ;
```



Again, the Mealy automaton from the previous examples is used. Besides the self-loops that were already introduced before, signal assignments for the state machine outputs are connected to the state transitions. By this, assignments to output signals appear only on clocked transitions and Flip Flops have to be provided for the outputs.

The assignments connected to the transitions, however, are equivalent to the assignment that leads to a new target state. Therefore the output values depend on the current input values and on the new state (NEXTSTATE). This affects the VHDL sourcecode: The values of the intermediate signals are calculated from the values of the current inputs and the current successor state. Again, Flip Flops are infered for the intermediate signals with another process.

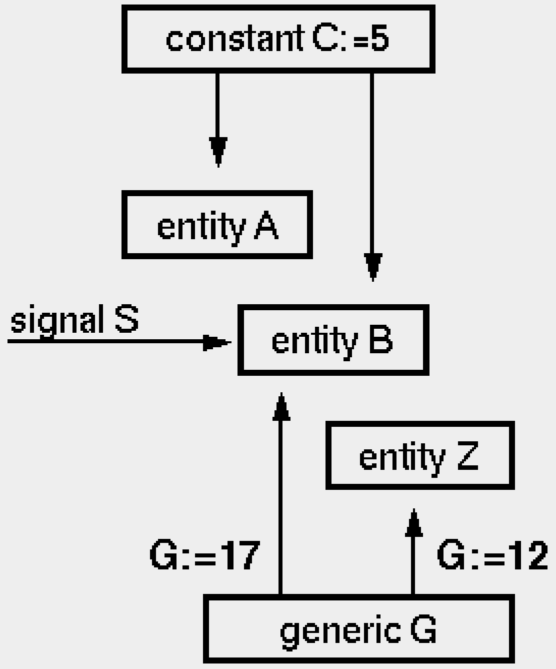
4.5.22 Waveform Registered Output Example (2)



- **No delay between STATE and output changes.**
- **"Spikes" of original Mealy machine are gone!**

The waveform shows the simulation results of the second version of the clocked Mealy automaton. The output values change synchronously with the state changes now and undesired temporary values are eliminated.

4.6 Advanced Synthesis



- **Constant C identical in all referencing units**
- **Generic G different but constant within each entity**
- **Input signal S set/changed in operation (different operation modes)**

Once you have finished a design, you hope that you will be able to use at least parts of the VHDL code in other designs as well. This is certainly possible as long as you can adopt the VHDL code via copy/paste. But if the function has to be changed slightly, the designer will have to adapt the VHDL code. To make this adaption easier and less error-prone, VHDL provides several ways of parameterizing a design or a module, that means the behaviour description depends on some parameters. The value of these parameters can then be set differently in different implementations or even on the fly during operation. The intention is that by changing the parameter the behaviour will change accordingly.

4.6.1 Parameterization via Constants

```
package T_PACK is
  constant MAX_VALUE :integer := 15;
end T_PACK;
```

```
use WORK.T_PACK.all;

entity COUNTER_C is
  port(...,
       COUNT : buffer integer range 0 to MAX_VALUE);
end COUNTER_C;

architecture RTL of COUNTER_C is
begin
  process(CLK)
  begin
    if CLK'event and CLK='1' then
      if RESET='1' then
        COUNT <= 0;
      elsif ENABLE='1' then
        if COUNT < MAX_VALUE then
          COUNT <= COUNT + 1;
        else
          COUNT <= 0 ;
        end if;
      end if;
    end if;
  end process;
end RTL;
```

- **Constants are fixed for the complete design**
- **Instantiations of COUNTER_C produce exactly the same counter**
- **Parametric signals in port map**

One way to parameterize a design is to use constants. The example shows a counter with reset and enable. The counter is free wheeling when enabled. The maximum value (MAX_VALUE) is set by a constant which is defined and given a value in the package T_PACK. Wherever COUNTER_C is built in (by use of a component declaration and component instantiation) the counter range is fixed (from 0 to the value specified in the package).

4.6.2 Parameterization via Generics(1)

```
entity COUNTER_G is
   generic ( MAX_VALUE : integer := 15);
   port(...                          -- default value
          COUNT : buffer integer
                              range 0 to MAX_VALUE );
end COUNTER_G;

architecture RTL of COUNTER_G is
begin
  process(CLK)
  begin
      if CLK'event and CLK='1' then
        if RESET='1' then
           COUNT <= 0;
        elsif ENABLE='1' then
          if COUNT < MAX_VALUE then
             COUNT <= COUNT + 1;
          else
             COUNT <= 0 ;
          end if;
        end if;
      end if;
  end process;
end RTL;
```

- **Generics are defined in the entity declaration**
- **Treated as constants in the architecture**
- **Default values**
- **Parametric signals in port map**

If you want to instantiate counters with different counter ranges in one design you will have to switch to generics. Generics are defined like the ports in the entity definition and receive their values during the step of component instantiation. Therefore, in addition to the port map, a generic map is required to provide these values. Generics may be given a default value in the generic clause which will be used if the generic is not explicitly assigned a value.

4.6.3 Parameterization via Generics(2)

```
entity TWO_COUNTERS IS
port(...);
end entity;

architecture RTL of
TWO_COUNTERS is
component COUNTER_G
   generic (MAX_VALUE:integer :=
15);
  port(...);
end component;

begin
  COUNTER1 : COUNTER_G
    port map (...); -- MAX_VALUE
with default value

  COUNTER2 : COUNTER_G
    generic map (MAX_VALUE =>
31)
    port map (...);
...
end RTL;
```

- **Different values for different instantiations**
  - **Instantiation with default value**
  - **Instantiation with generic map**

Every instantiation needs a label

Only generics of type integer are supported by synthesis tools

The component declaration is as usual except that you must not forget the generic clause.

But how does the instantiation of COUNTER_G work? If default values were defined for the generics, the component instantiation does not need a generic map (cf. COUNTER1). Of course, the default value can be overwritten by setting the generic to an explicit value in the generic map of the component instantiation (cf. COUNTER2).

The most useful feature is that entities using generics can be instantiated with different values for the generics within the same module. However only integer type generics are synthesizable!

© LRS - UNI Erlangen-Nuremberg

4.6.4 GENERATE Statement

```
entity GENERATE_COUNTER IS
  port(...);
end entity;

architecture RTL of
GENERATE_COUNTER is
  component COUNTER_G
    generic (MAX_VALUE:integer :=
15);
    port(...);
  end component;

begin
  GEN: for K in 2 to 5 generate
    COUNTER : COUNTER_G
      generic map (MAX_VALUE =>
2**K-1)
      port map (...);
  end generate;

  . . .
end RTL;
```

- **'for generate' needs a label**
- **"loop" for concurrent statements (component instantiations, signal assignments)**
    - **several instantiations of the same component**
    - **different values for generics**
    - **loop variable implicitly declared**

When a component has to be instantiated several times, the way described above would be exhaustive. The generate statement provides a shortcut to this problem. Within a 'for ... generate ...' loop concurrent statements can be iterated. This applies not only to repeated component instantiations but also to concurrent signal assignments. The loop variable is declared implicitly, again, and can only be read, e.g. as generic value.Value assignments will lead to an error.

4.6.5 Conditional GENERATE Statement

```
entity GENERATE_COUNTER IS
  port(...);
end entity;

architecture RTL of GENERATE_COUNTER is
  component COUNTER_G
    generic (MAX_VALUE:natural := 15);
    port(...);
  end component;

begin
  GEN: for K in 1to 8 generate
      COND1 : if K<=5 generate
        COUNTER : COUNTER_G
          generic map (MAX_VALUE => 2**K-1)
          port map (...);
     end generate;

      COND2: if K>5 and FLAG generate
        MAX : COUNTER_G
          generic map (MAX_VALUE => 31)
          port map (...);
      end generate;
  end generate;

  . . .
end RTL;
```

- **If condition is true then generate ...**
- **No elsif / else paths**
- **Each generate needs a label**

To make the generate statement even more powerful you can use an if-generate statement which allows to execute the concurrent statements subject to the value of a boolean expression. In contrast to conditional signal assignments or the sequential if statement, elsif and else paths are not permitted.

In the example, 8 counters are instantiated. For K=1...5 the maximum counter value is calculated according to the formula 2^K-1, i.e. 1, 3, 7, 15, 31. For higher index values the maximum counter values remains 31. It is also possible to evaluate constants or generics of the own entity in the expression of the if condition.

4.6.6 'Parameterization' via Signals

```
entity COUNTER_S is
  port( **MAX_VALUE** : in integer range 0 to 1023
           . . .);
end COUNTER_S;

architecture RTL of COUNTER_S is
begin
  process(CLK)
  begin
     if CLK'event and CLK='1' then
       if RESET='1' then
          COUNT <= 0;
       elsif ENABLE='1' then
          if COUNT < MAX_VALUE then
            COUNT <= COUNT+1;
          else
            COUNT <= 0 ;
         end if;
       end if;
     end if;
  end process;
end RTL;
```

- **Set different modes in operation (cf. MAX_VALUE)**
- **No parametric signals in port map**

Constants and generics are very profitable when configuring a module before synthesizing. If it is necessary to switch parameters during operation, the only solution is to feed these parameters via signals into the module.

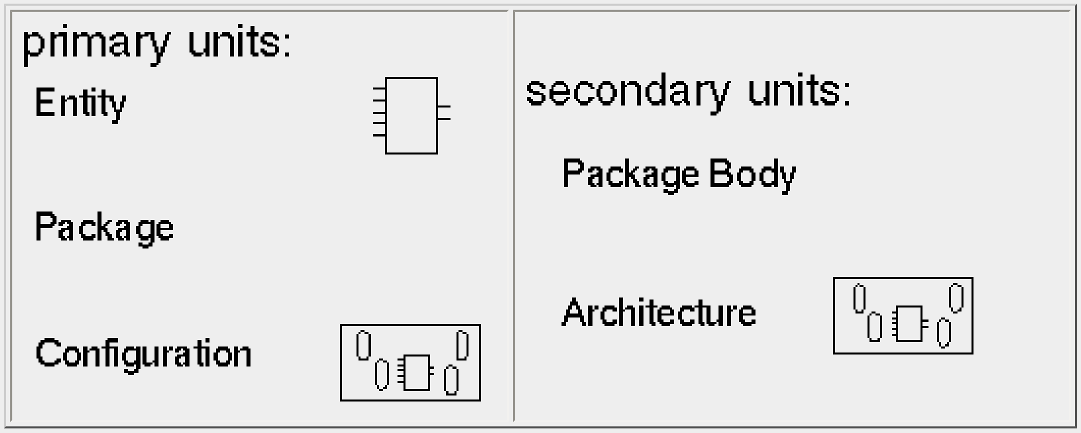
The counter range in the example above can be set to values between 1 and 1023 during operation. For that reason, an overhead of hardware is generated to provide this flexibility in functionality. The signal MAX_VALUE must be stable when the active clock edge occurs. If this cannot be guaranteed, storing elements will have to be created, which hold the current value of MAX_VALUE and are updated at certain times, only.

5. Project Management

- **Design Components**
- **Libraries**
- **File Organisation**

5.1 Design Components

- **Five basic design units, divided into two groups:**

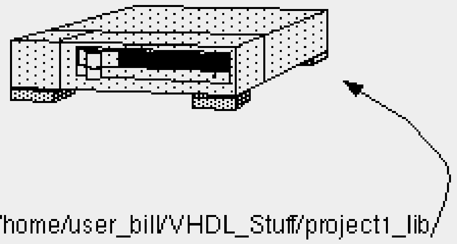


- **Each design unit has to be analysed and stored in a library**
- **Packages may be split into header (declarations) and main part**

All parts of a VHDL design have to be analysed/compiled before they can be used for simulation or synthesis. In total, five different so called design units exist: entity/architecture, package/package body and configuration. A file must contain at least one design unit to be accepted by the compiler. The separation of the package body from the declarative part is not strictly necessary.

5.1.1 Libraries

- **Container for compiled design units**
  - **entities, architectures,**
  - **packages, package bodies,**
  - **configurations**
- **Mapped to a directory on the filesystem**
  - **platform independency**
  - **setup file needed**
- **Different libraries in one design project possible**



Setup file:
PROJECT1: /home/user_bill/VHDL_Stuff/project1_lib/

Commandline:
compile -library **PROJECT1** p_frame_types.vhd

After compilation, the design units are stored in a so called "library". The purpose of the library mechanism is platform independency, i.e. every VHDL tool has to map the logical library name to a physical directory. If the target library is not specified when compiling a VHDL file, the design units are compiled into the default library ' **work** '. It is often mapped to the startup directory of the software. The default settings can be overwritten, e.g. via setup files. Special setup files or option settings are needed to specify alternatives.

In the example, the logical library name PROJECT1 is mapped to the physical directory "/home/user_bill/VHDL_Stuff/project1_lib" in the setup files for the simulator and synthesis tool. This library must be named as target when VHDL files are compiled, i.e. the command will look like "compile -library project1 p_frame_types.vhd". In order to be able to use the information contained in the P_FRAME_TYPES package, the library PROJECT1 has to be made visible with the LIBRARY statement. Afterwards, individual elements of the library are made available with the USE statement.

5.1.2 The LIBRARY Statement

```
library EXAMPLE;
use EXAMPLE.PKG.all;

entity A is
. . .
end A;

library EXAMPLE;
use EXAMPLE.PKG.all;

entity X is
. . .
end X;

architecture BEH of A is
. . .
end BEH;
```

```
architecture BEH of X is
. . .
end BEH;
```

- **Occurrence**
  - **in front of any design unit (entity, architecture, package, ...)**
  - **valid for the next unit, only**
- **Secondary units "inherit" library declarations**
- **Does not declare any VHDL design units / objects**
- **Refers to existing libraries**
- **Easier design management and preparation**
- **May cause visibility problems**

Library statements may only be placed in front of VHDL design units and are valid for the unit immediately following, only. Secondary design units, however, inherit library declarations that apply to their primary counterparts. Thus, a library clause needs not be repeated in front of an architecture or package body, even when they are placed in separate files (cf. "architecture BEH of X"). On the other hand, library EXAMPLE must be declared explicitly for the entity X, again, if definitions from the package PKG are to be used.

The libraries WORK and STD are visible by default and need not be declared via library statements. The library STD contains the STANDARD package, which is also visible by default, and the TEXTIO package that is necessary for file I/O operations.

Libraries can be used advantageously to separate the object code of different design projects. Yet, visibility problems might occur. If objects that are not distinguishable by the compiler exist, none of the eligible objects is used and an error message is generated. Probably the most obvious example is the case of procedures with identical names and parameter lists that are defined in separate packages. Please note that an illegal redeclaration will be reported if both packages reside in the same library.

5.1.3 The USE Statement

```
library EXAMPLE;
library OTHER_1;
library OTHER_2;

use EXAMPLE.PKG_1.CONST_A,
      EXAMPLE.PKG_1.CONST_B;

use EXAMPLE.PKG_2.all;

use EXAMPLE.ENTITY_1;

architecture BEH of ENTITY_1 is
   use OTHER_1.all;
   signal A : integer;
begin
   A <= OTHER_2.PKG.CONST_A;
end BEH;
```

- **Occurrence**
  - **in front of any design unit**
  - **valid for the next unit, only**
  - **may appear in declarative parts, if all items of an object are made visible**
- **Secondary units "inherit" use clauses**
- **Library must be known**
- **Grants access to individual items**
  - **design units (e.g. entities)**
  - **objects from packages (e.g. constants)**
  - **'all' accesses all objects**
- **complete "logical pathname" needed, if 'use' is omitted**

A library statement alone does not give access to the design units and objects contained in this library. An additional use clause is necessary for this purpose. Most of the rules concerning its location and consequences are equivalent to the library statement and consequently both statements appear conjunctively most of the time.

A use clause makes individual elements of a library visible within a design unit. It is even possible to select specific objects from a package. Usually, however, the keyword ' **all** ' is used to make all its declarations available to the user. The keyword ' **all** ' is also applicable to complete library contents. Use clauses with the keyword ' **all** ' may also be placed within the declarative part of entities, architectures, etc.

While the library statement is always necessary, objects can be accessed via their "selected name" instead of a use clause. Then, the complete logical path consisting of the names of the library, design unit and object must be specified.

5.2 Name Spaces

```
package PKG is
  constant C : integer := 1;
end PKG;

entity ENT is
  constant C : integer := 4;
end ENT;

architecture RTL of ENT is
  signal C : integer := 8;
begin
   -- only signal C is visible
  C <= 24;
  process
    variable C : integer := 9;
  begin
     -- only variable C is visible
    C := 42;
    for C in 0 to 5 loop
       -- only loop parameter C is visible
       ENT.RTL.C <= C; -- selected name
    end loop;
     ENT.RTL.C <= work.PKG.C;
  end process;
end RTL;
```

- **Multible objects of same names are allowed:**
  - **Without selected names: Local name overrides**
  - **Assignment with complete selected names: ENT.RTL.C <= 12;**
- **Declarations in a package are visible in all design units which use this package**
- **Declarations in an entity declarative part are visible in all the architectures of this entity**
- **Declarations in an architecture are visible for all processes of this architecture**
- **Declarations in a process are visible only inside this process**
- **Declarations in a loop statement (loop parameter) or in a subrogramm are only visible inside these objects.**

5.3 File Organisation

- Primary and secondary design units can be split into several files
- Advantages of

❍ Several Packages	❍ modularisation and reuse aspects (IEEE, corporate, project packages) ❍ separation of synthesisable from simulation only VHDL
❍ Package / Package body separation	❍ no recompilation of the design hierarchy if body (implementation) changed

❍ **Entity / Architecture separation**	❍ **system design (top level, structural) independent from implementation** ❍ **several modeling alternatives (e.g. behavioural, RTL) possible**
❍ **Several top level configurations**	❍ **adjust design to goal of simulation** ❍ **comparison of alternative architectures**

VHDL offers several possibilities of implementing hierarchy. The language introduces the main hierarchy into the design with the philosophy of entity/architecture pairs and their instantiation as components. This leads to a strict separation of interface and implementation. With the help of packages, libraries and a strict coding style (port, type naming conventions, comments, ...) it is possible to manage huge designs with a plenty of files. Switching between the different implementations during simulation is done with the configuration mechanism.

5.3.1 Packages

- Packages contain VHDL objects that may be shared by many design units
  - (sub-)type declarations
  - constants
  - subprograms
  - components
- Package contents must be made available via use clauses
- Implementation details can be hidden in a package body
- Package body
  - visible only within a package
  - always linked to a package
  - may contain all declarations/definitions that are legal for a package
  - holds definition of previously declared constants and subprograms

Packages are the only language mechanism to share objects among different design units. Usually, they are designed to provide standard solutions for specific problems, e.g. data types and corresponding subprograms like type conversion functions for a certain bus protocol, procedures and components (macros) for signal processing purposes, etc. A body is strictly needed if subprograms are to be placed in packages because only the declaration of functions and procedures may be placed in the package itself. A package body is not needed if no subprograms or deferred constants are used. Please note that only objects declared in a package can be referenced via use statements.

5.3.2 Package Syntax

Package declaration:	Package body declaration:
package IDENTIFIER is -- Declaration of -- types and subtypes -- subprograms -- constants, signals and shared variables -- files -- aliases -- components end [package] [IDENTIFIER] ;	package body IDENTIFIER is -- Definition of previously declared -- constants -- subprograms -- Declaration/definition of additional -- types and subtypes -- subprograms -- constants, signals and shared variables -- files -- aliases -- components end [package body] [IDENTIFIER] ;

Only the package content is visible, NOT the body

Subprogram definitions can not be placed in a package

'93 The keywords 'package' / 'package body' may be repeated after the keyword 'end'

5.3.3 Package Example

```
package PKG is
      type T1 is ...
      type T2 is ...
      constant C : integer;
      procedure P1 (...);
end PKG;
```

```
package body PKG is
     type T3 is ...

     C := 17;

     procedure P1 (...) is
     . . .
     end P1;

     procedure P2 (...) is
     . . .
     end P2;
end PKG;
```

```
library STD;                -- VHDL default
library WORK;               -- VHDL default
use STD.standard.all; -- VHDL default
use work.PKG.all;

entity EXAMPLE is
end EXAMPLE;

architecture BEH of EXAMPLE is
     signal S1 : T1;
     signal S2 : T2;
     signal S3 : T3; -- error: T3 not declared

begin

     P1 (...);
     P2 (...);          -- error: P2 not declared

end BEH;
```

- **Signals or procedures which are declared in the package body cannot be accessed**
- **A specific new data type may be defined in only one of the referenced packages**
- **Deferred constants: actual value assignment is package body**

The libraries WORK and STD and the STANDARD package are available by default. Thus the corresponding VHDL statements are not needed.

If a package body exists, it should be placed in a separate file. Otherwise it would not be possible to compile the body separately. Only the package contnent can be made visible with use statements so changes to constant declarations or subprogram bodies can be made in the package body without implying a recompilation of the complete design. Only the package body has to be recompiled then. A package body is not necessary if no subprograms or deferred constants are declared.

In this example the type T3 and the procedure P2 are declared in the package body, only. This leads to an error during compilation because these two items are not visible in the architecture.

5.3.4 Use of Packages

- **Definitions of: types, subtypes, constants, components and subprograms**
- **Deferred constants:
constants, which are declared in the header of a package and are assigned a value in the body**

package P is
    constant C: integer;
end P;
	package net is
	    subtype p_bit is std_ulogic;
	    subtype p_bit_vector is std_ulogic_vector;
	    subtype p_net_data is p_bit_vector (7 downto 0);
	    type p_net_frame is array (1 downto 0) of p_net_data;
	
	    constant low: integer := 0;
	    constant high: integer :=1;
	
	    constant period: time := 10 ns;

package body P is
    constant C : integer:=200;
end P;	

- **There are low-cost tools which do not support self defined packages.**

It is possible to change the value of the constant C several times without recompiling the complete design. Only the package body has to be recompiled.

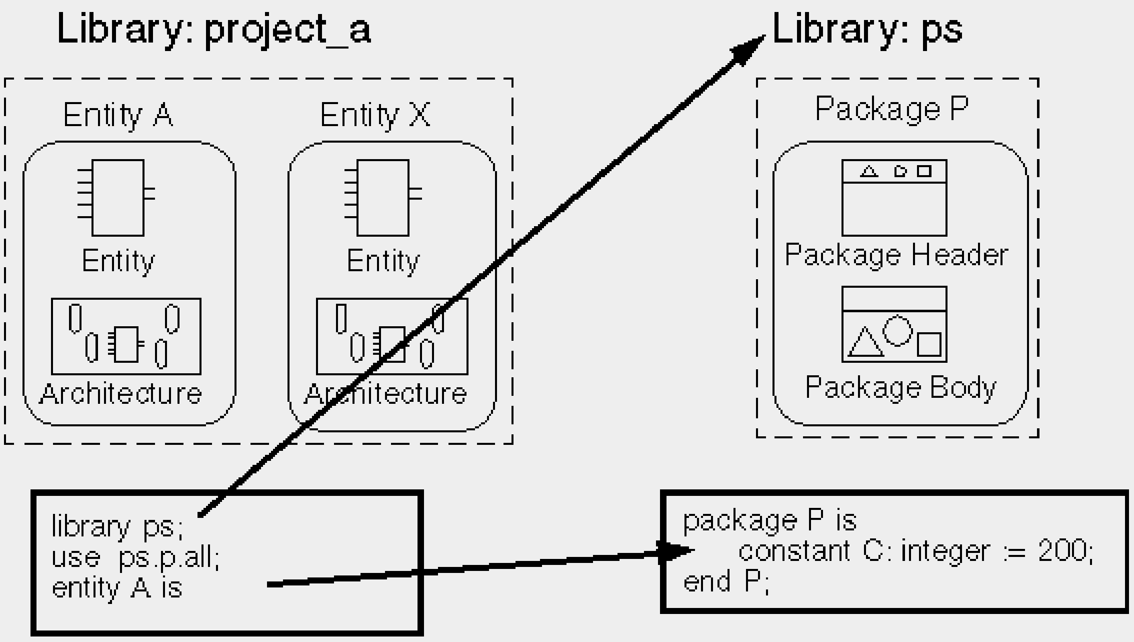
A package body is not necessary if no subprograms or deferred constants are declared.

Example:

```
package SimulationTimes is
      constant tCLK      :      time := 30 ns;
      constant tSetup    :      time := 10 ns;
      constant tHold     :      time := 14 ns;
```

```
    constant tCLK77  :    time := 77 ns;
    constant tWrite   :    time := 8 ns;
    constant tRead    :    time := 8 ns;
    constant tData    :    time := 21 ns;
end SimulationTimes ;
```

5.3.5 Visibility of Package Contents



```
library ps;
use  ps.p.all;
entity A is
```

```
package P is
     constant C: integer := 200;
end P;
```

With the keyword "all" as suffix in the use clause it is possible to make visible all the objects declared in the package header. If you only want to make visible special objects of the package header you have to use their simple name as suffix:

```
Library ps;
use ps.p.C;
```