LANCASTER UNIVERSITY
Engineering
Department

Microsystems and SoC Technology
COURSE 410/510 - Lent 2007-08

Dr. Dinesh Pamunuwa

Session Goals

- Introduce basic VHDL constructs
- Introduce the VHDL simulation cycle and timing model
- Illustrate the usage of VHDL as a digital hardware description language

Outline of this session

- Introduction to Modelling with HDLs
  - **Motivation**
  - **Levels of abstraction**
  - **VHDL design example**
- Elements of the VHDL model
  - **Entity Declarations**
  - **Architecture Descriptions**
  - **Timing Model**
- Test Bench
- Basic VHDL Constructs
  - **Data types**
  - **Objects**
  - **Sequential and concurrent statements**
  - **Packages and libraries**
  - **Attributes**
  - **Predefined operators**
- Examples
- Summary

Session outline

- Introduction
  - **Motivation**
  - **Levels of abstraction**
  - **VHDL design example**
- Elements of the VHDL model
- Test Bench
- Basic VHDL Constructs
- Examples
- Summary

Why HDLs

- Requirements specification
  - **Unambiguous capture of system requirements including functionality and performance**
- Design documentation and communication
  - **Standardised HDL facilitates**
    - **Design documentation**
    - **Design communication among large teams**
    - **Simulation of same code in different environments**
- Design methodologies
  - **supports many different design methodologies (top-down, bottom-up) and is very flexible in its approach to describing hardware**

Why HDLs cont.

- Design Management
  - **Usage of VHDL constructs, such as packages and libraries, allows common elements to be shared among members of a design group.**
- Design verification
  - **The initial requirements specification and actual design can both be simulated with same test stimuli.**
    - **Comparison of the resulting outputs verifies the functionality of the design.**
  - **Aids Formal verification (logically proving the equivalence of two specifications with out actual simulation)**
- Automated synthesis through computer-aided design tools
  - **Computer aided design tools automatically synthesize or generate the hardware from the requirement specification.**
  - **Automated synthesis reduces design time and eliminates errors due to manual design**

VHDL: History

- 1960s – 1980s
  - **Many HDLs available, no standardisation.**
- 1983 VHSIC Program from US DoD initiates definition of VHDL.
- 1987 VHDL Standard (IEEE 1076) approved.
- 1992 IEEE 1164 (abstract data types for different signal characteristics, eg 9-valued logic) developed.
- 1993 VHDL modified.
  - **Clarifications and refinements**
- Since 1994
  - **Widespread acceptance of VHDL.**
  - **All existing commercial Electronic Design Automation (EDA) tools from Synopsys, Cadence, Mentor, Xilinx, Altera, etc support specification, simulation and synthesis with VHDL.**

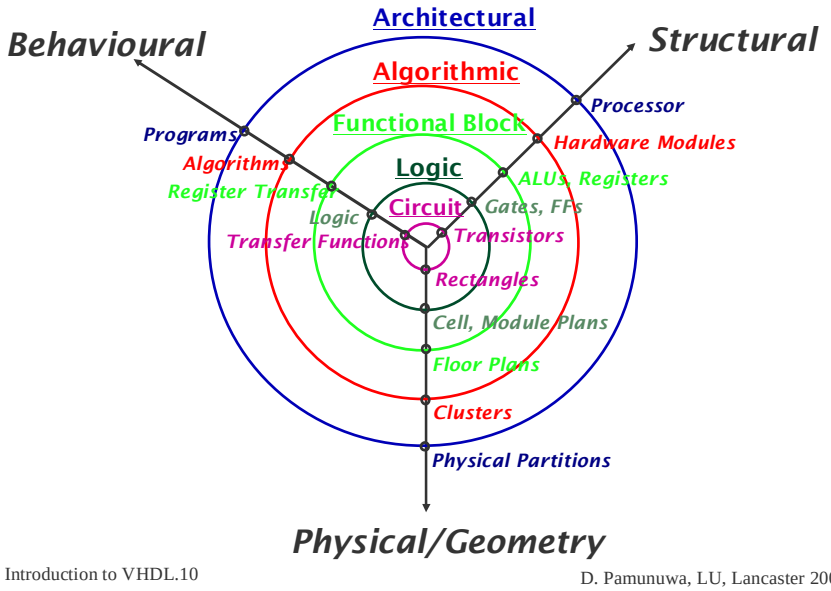
Other HDLs

- Verilog - very popular
  - **PLI (programming language interface) supports simulation models implemented in C.**
  - **Wide support of simulation libraries of ASIC devices.**
  - **Lacks higher-level design management constructs like VHDL's configuration, package and library.**
- SystemC – new Synopsys led initiative
  - **C++ based object oriented framework that supports specification, simulation and synthesis of hardware.**

Design Representations

- Any complex engineering task tackled by “divide and conquer” approach:
  - **Hierarchy and Partitioning**
  - **Different levels of abstraction**
- An integrated electronic system can be represented at different abstraction levels in three domains:
  - **Behavioural**
    - **Specification of system/module functionality**
  - **Structural**
    - **Description of how entities fit together to implement functionality**
  - **Physical/Geometrical**
    - **Physical description/layout of structure with required connectivity**

Digital Design Domains and Levels of Abstraction – Gajski and Kuhn's Y Chart



A designer needs to work at various levels of abstraction. Many of these levels are shown pictorially in the Gajski/Kuhn chart.

Ideally, the designer will have a set of tools at his disposal that will allow him to move between different domains and abstraction levels as and when required. For example, functionality is most easily specified at the highest abstraction level in the behavioural domain. Automatically moving from a behavioural description at the highest level to a physical description at the lowest level of abstraction (a set of polygons depicting the transistors and interconnections), is called BEHAVIOURAL SYNTHESIS.

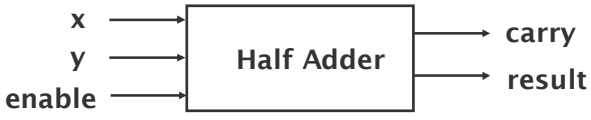
Modelling languages that can be used to represent designs in one or more of these domains and move between them are called Hardware Description Languages (HDLs). Two widely used languages are VHDL (Europe) and Verilog (US). Many design tools can take behavioral or structural VHDL and generate chip layouts. Other languages exist that can be used at higher levels of abstraction more conveniently, such as System C.

As an illustrative example, the next few slides will show a sample VHDL design process to demonstrate how a designer can move from

an algorithmic behavioral description, to a register transfer (or data flow) description, to a gate level description.

Example Design Process

- Problem: Design a single bit half adder with carry and enable
- System Level Behavioural Specifications
  - **Passes results only on enable high**
    - **Result gets x plus y**
    - **Carry gets any carry of x plus y**
  - **Passes zero on enable low**

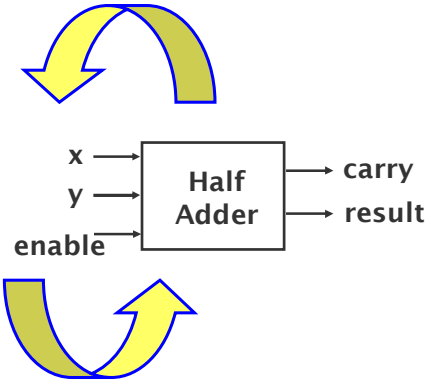


The requirement is to design a single bit adder with carry and enable functions. The inputs x, y, and enable are single bits with enable active high. The output of result is the bit addition of x and y and the carry output is the carry generated by the addition. When the enable line is low, the adder is to output zeroes.

Behavioural Hierarchy

- Proceed to algorithmic level in behavioural domain

```
IF enable = 1 THEN
    result = x XOR y
    carry = x AND y
ELSE
    carry = 0
    result = 0
```



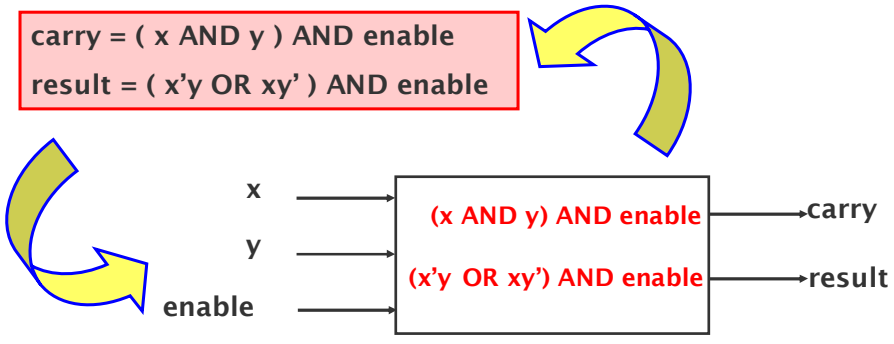
- The model can now be simulated to verify correct understanding of the problem

Note that the code used in the example is "pseudo code" and not intended to follow VHDL syntax.

In the first stage of the design process, a high level behavior of the adder is considered. This level uses abstract constructions (such as the IF-THEN-ELSE statement) to make the model more readable and comprehensible. Simulation of the adder at this level proves correct understanding of the problem specifications of the adder.

Behavioural Hierarchy *cont.*

- With algorithmic description confirmed, proceed to RTL/Logic level: Boolean logic equations describing the data flow are created

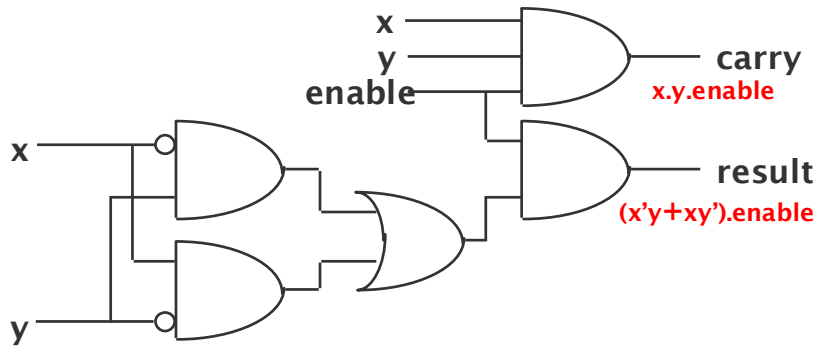


- Again, the model can be simulated at this level to confirm the logic equations

After the behavioral model is confirmed, a more specific model is formed. This model uses logic equations to describe the flow of data inside the model. Notice that the higher level construct of the IF-THEN-ELSE statement is gone.

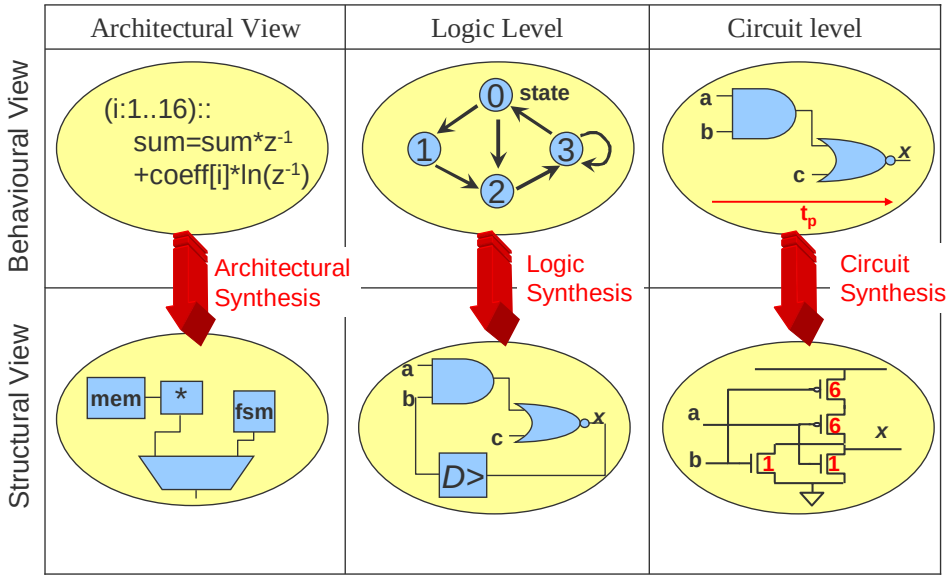
Structural Hierarchy

- Then a structural description is created at the logic level – consists of interconnected gates



Finally, the logic equations are mapped to specific logic gates. The models for these gates can come from many different libraries, and use specific or generic technologies. Upto this point, the description has been TECHNOLOGY INDEPENDENT, which is very important, as the same design can be REUSED. The logic gate mapping can be done for different technologies, by performing synthesis for the target library, or otherwise.

Common Synthesis Tasks

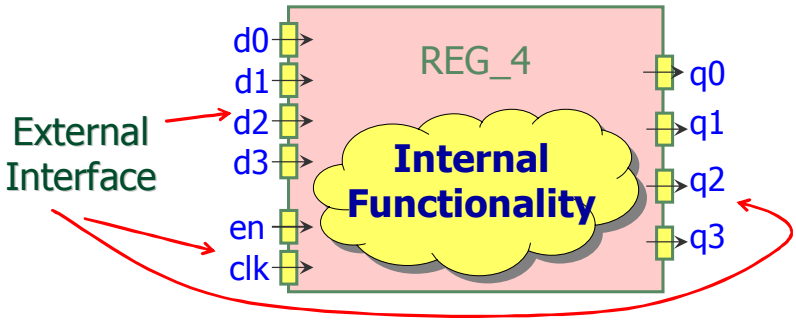


Session outline

- Introduction
- Elements of the VHDL Model
  - **Entity Declarations**
  - **Architecture Descriptions**
  - **Timing Model**
- Test Bench
- Basic VHDL Constructs
- Examples
- Summary

VHDL Model Components

- A complete VHDL component description requires a VHDL *entity* and a VHDL *architecture*
  - **The entity defines a component's interface**
  - **The architecture defines a component's function**



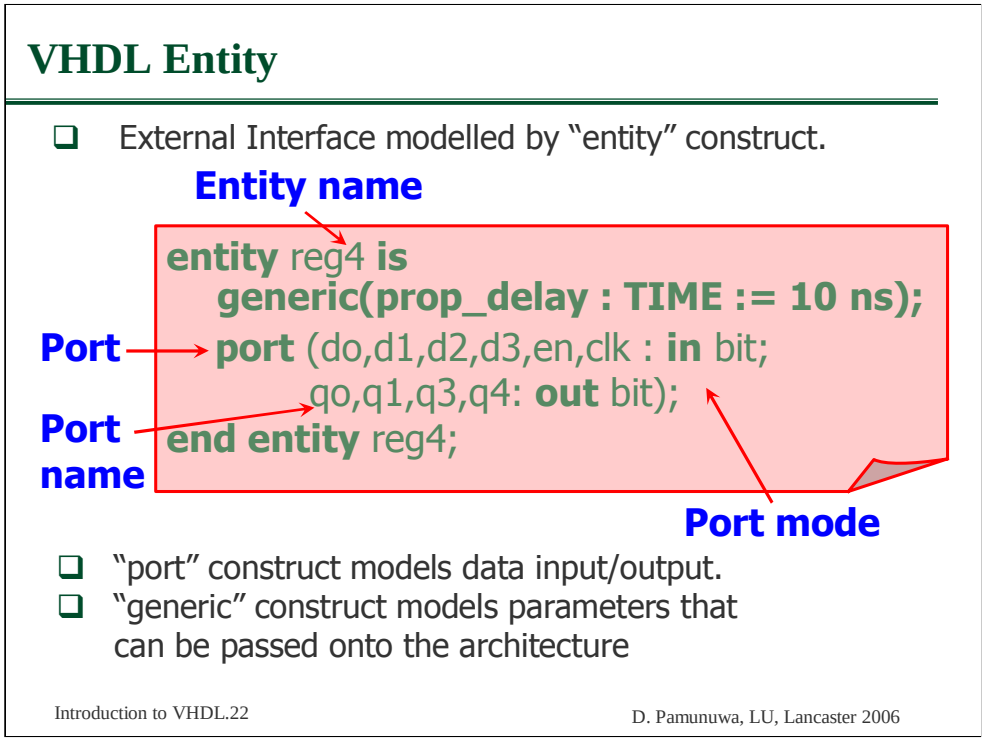
Introduction to VHDL.21

D. Pamunuwa, LU, Lancaster 2006

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

Lets consider an example piece of hardware, which is a register. There are pins or ports which communicate with the outside world, and they will have some direction associated with them. These pins define the interface to the external world.

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

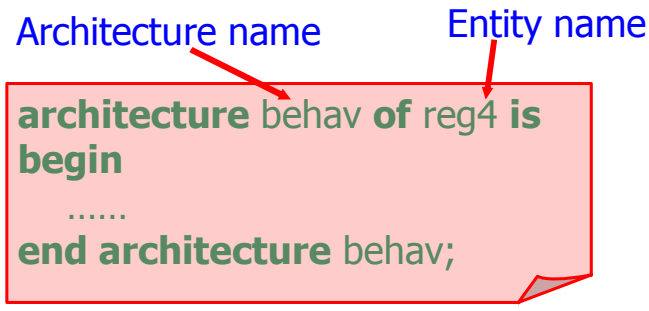


The VHDL key word *entity* refers to the VHDL construct in which a component's interface (which is visible to other components) is described. The first line in an entity declaration provides the name of the entity. Next, an optional GENERIC statement includes value assignments to parameters that may be used in the architecture descriptions of the component. GENERIC statements create parameters to be passed on to the architectures of this entity. These parameters may be used to characterize the component by setting propagation delay, component ids, etc. It is not a data interface.

Following that, the PORT statement indicates the actual interface of the entity. The port statement lists the signals in the component's interface, the direction of data flow for each signal listed, and type of each signal. In terms of syntax, no semicolon is required before the closing parenthesis in PORT or GENERIC declarations. The entity declaration statement is closed with the END keyword, and the name of the entity is optionally repeated.

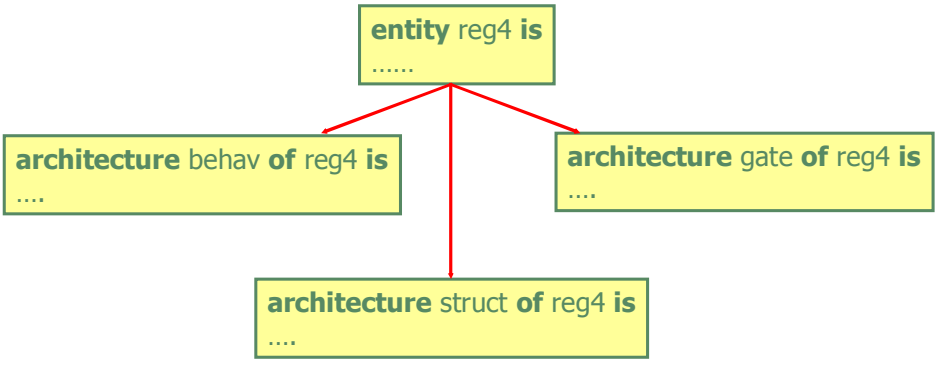
VHDL Architecture

- Internal functionality modelled by "architecture" construct



One Entity, Many Architectures

- There can be more than one "architecture" bodies (internal functionality descriptions) associated with one "entity"



Architecture Bodies

- Describes the component's operation
- Consist of two parts :
  - **Declarative part -- includes necessary declarations**
    - **e.g. type declarations, signal declarations, component declarations, subprogram declarations**
  - **Statement part -- includes statements that describe organization and/or functional operation of component**
    - **e.g. concurrent signal assignment statements, process statements, component instantiation statements**

```
ARCHITECTURE dataflow OF half_adder IS
    SIGNAL xor_res : BIT;      -- declarative part
BEGIN                        -- start of statement part
      carry <= enable AND (x AND y);
      result <= enable AND xor_res;
      xor_res <= x XOR y;
END dataflow;
```


The architecture body describes the operation of the component. There can be many different architectures described for each entity. However, for each instantiation of the entity, a specific architecture has to be bound to it.

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

Describing Operation in Architecture Body

- Fundamental unit for component behavior description is the *process*
  - **Processes may be explicitly defined – using keyword *process***
  - **Processes may be implict – a signal assignment**
- Primary communication mechanism is the *signal*
  - **Process executions result in new values being assigned to signals which are then accessible to other processes**
  - **Similarly, a signal may be accessed by a process in another architecture by connecting the signal to ports in the entities associated with the two architecture**

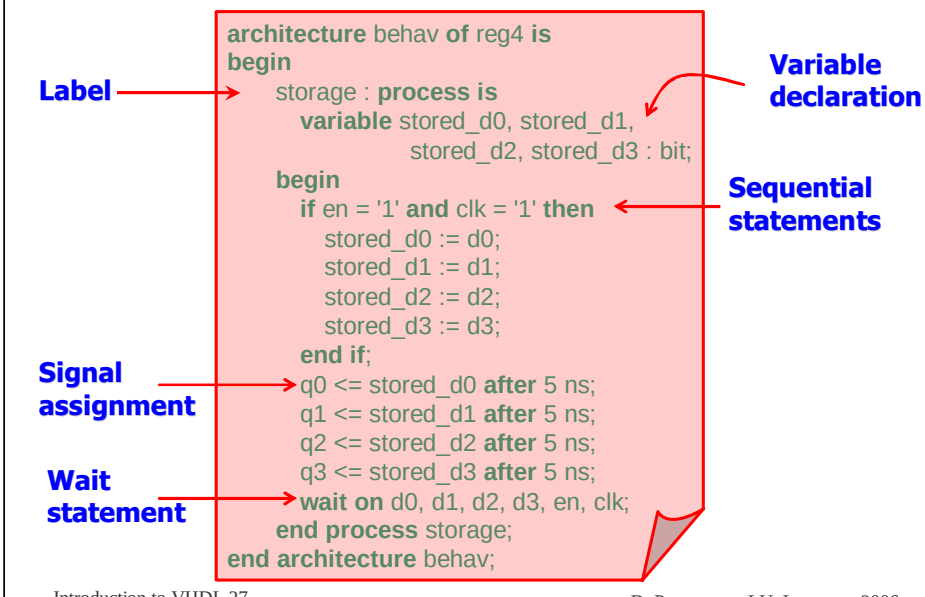
A signal is the primary communication mechanism in VHDL, and models a carrier. In hardware terms, we can start by thinking of it as a wire. All behavioral descriptions in VHDL are constructed using processes.

Processes may be defined explicitly where complex behavior can be described in a sequential programming style, or they may be implicitly defined in concurrent signal assignment statements. The primary purpose of the process is to determine new values for signals.

Signals are accessible between processes, and therefore provide a mechanism for the results of one process execution to be communicated to other processes. Signals may be made accessible to processes within other VHDL architectures by connecting them to ports of the respective entities.

More detail will be added as we continue.

Behavior Example



Introduction to VHDL.27 D. Pamunuwa, LU, Lancaster 2006

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

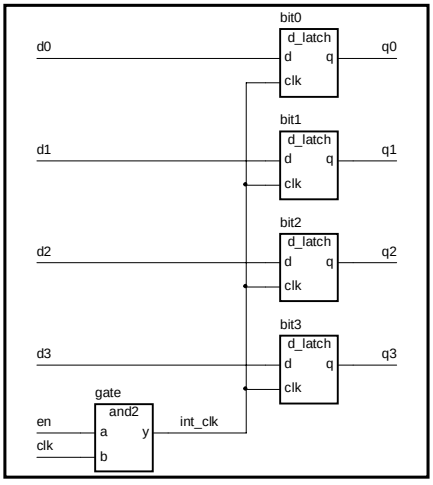
As mentioned earlier, the fundamental unit of behavioral description in VHDL is the *process;* all processes are executed concurrently with each other. The data flow modeling style (with concurrent signal assignments) is a special case of the general VHDL process mechanism in that each concurrent signal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

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

Modelling Structure

- Structural architecture implements the module as a composition of subsystems
- Contains:
  - **signal declarations, for internal interconnections**
    - **the entity ports are also treated as signals**
  - **component instances**
    - **instances of previously declared entity/architecture pairs**
  - **port maps in component instances**
    - **connect signals to component ports**
- VHDL provides mechanisms for:
  - **describing highly repetitive structures easily**
  - **supporting hierarchical description**

Structure Example



Structure Example

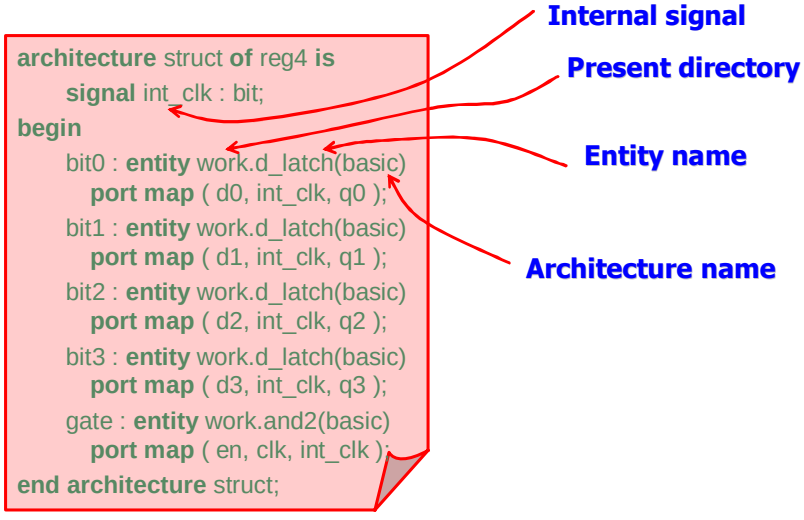
- First declare entities and architectures for D-latch, AND-gate

```
entity d_latch is
      port ( d, clk : in bit;  q : out bit );
end entity d_latch;

architecture basic of d_latch is
begin
      latch_behavior : process is
      begin
         if clk = '1' then
            q <= d after 2 ns;
         end if;
         wait on clk, d;
      end process latch_behavior;
end architecture basic;
```

```
entity and2 is
      port ( a, b : in bit;  y : out bit );
end entity and2;

architecture basic of and2 is
begin
      and2_behavior : process is
      begin
         y <= a and b after 2 ns;
         wait on a, b;
      end process and2_behavior;
end architecture basic;
```

Structure Example

- Wire them together to implement a register

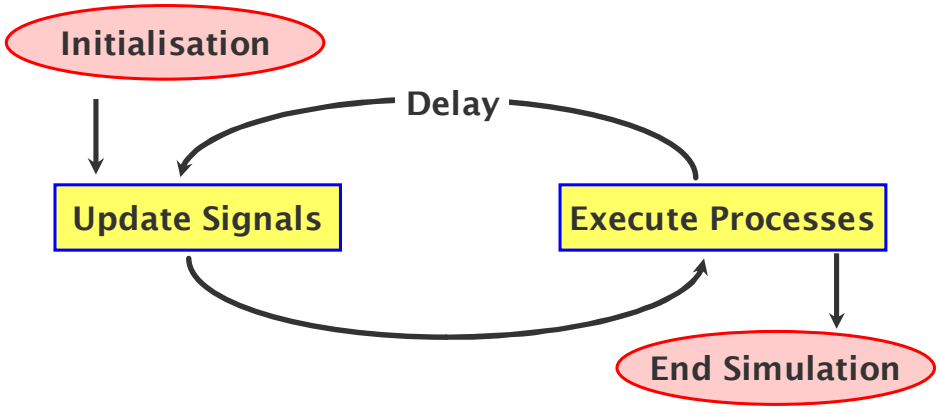


Session outline

- Introduction
- VHDL Model Components
  - **Entity Declarations**
  - **Architecture Descriptions**
  - **Timing Model**
- Test Bench
- Basic VHDL Constructs
- Examples
- Summary

Simulation and Timing Model

- VHDL uses a 2-step simulation cycle to model the stimulus and response nature of digital hardware



Introduction to VHDL.34

D. Pamunuwa, LU, Lancaster 2006

The VHDL simulation model is essentially a 2-step process. At the start of a simulation, signals with default values are assigned those values. In the first execution of the simulation cycle, all processes are executed until they reach their first *wait* statement, or the end of the process. These process executions will include signal assignment statements that assign new signal values after prescribed delays.

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

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

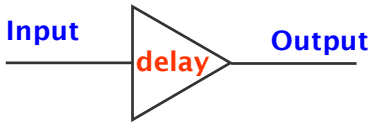
This cycle continues until the simulation termination conditions are met or until all processes are suspended indefinitely because no new signal assignments are scheduled to unsuspend any waiting processes.

Simulation Algorithm

- Simulation cycle
  - **advance simulation time to time of next transaction**
  - **for each transaction at this time**
    - **update signal value**
      - **event if new value is different from old value**
  - **for each process sensitive to any of these events, or whose "wait for …" time-out has expired**
    - **resume**
    - **execute until a wait statement, then suspend**
- Simulation finishes when there are no further scheduled transactions

Delay Types

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



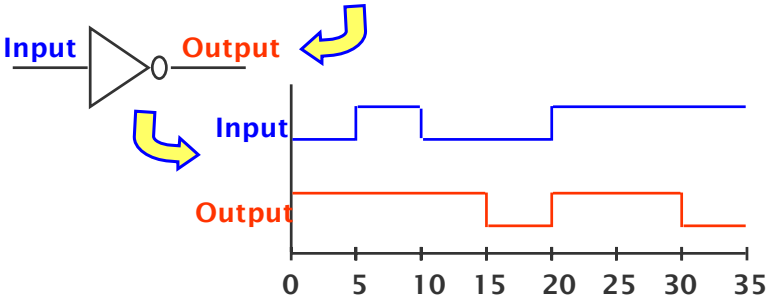
The different delay types in VHDL can be quite confusing at the start. Hence we will spend some time on it, as understanding them properly is the first step in designing logic with VHDL.

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

Transport Delay

- Signal will assume new value after specified delay
- Transport delay must be explicitly specified
  - **keyword "TRANSPORT" must be used**

```
-- Modelling Inverter delay with TRANSPORT delay
Output <= TRANSPORT NOT Input AFTER 10 ns;
```



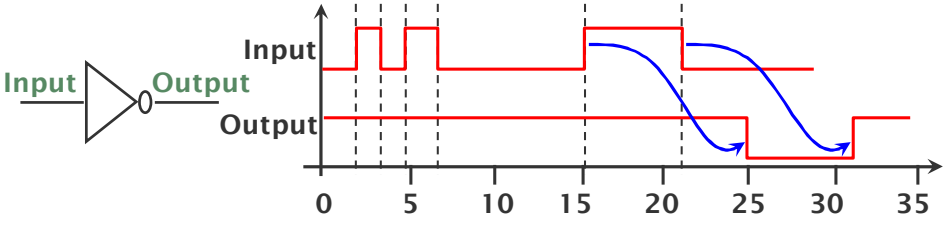
Transport delay is the simplest in that when it is specified, any change in an input signal value may result in a new value being assigned to the output signal after the specified propagation delay.

In this example, *Output* will be an inverted copy of *Input* delayed by the 10ns propagation delay regardless of the pulse widths seen on *Input* .

Inertial Delay

- Specifies:
  - **Signal assignment delay**
  - **Minimum pulse width to suppress pulses of shorter duration**
  - **e.g. Inverter with propagation delay of 10ns which suppresses pulses shorter than 5ns**

```
Output <=  REJECT 5ns INERTIAL NOT Input AFTER 10ns;
```



Note: the *REJECT* feature is new to VHDL 1076-1993

The keyword INTERTIAL is used to specify the propagation delay for the signal, while the keyword REJECT is used to specify a minimum duration for the input pulse width. In this example, any pulse on *Input* narrower than 5ns is not observed on *Output*.

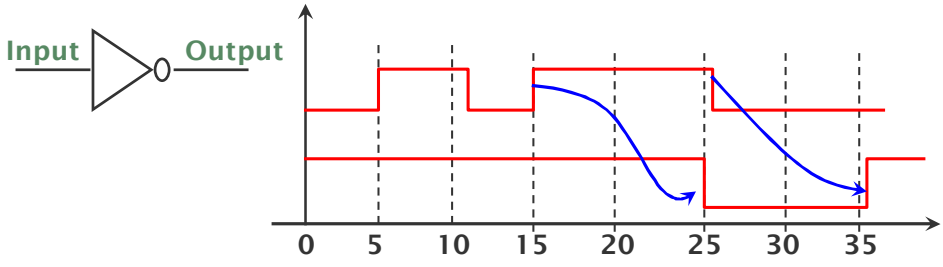
The REJECT construct is a new feature to VHDL introduced in the VHDL 1076-1993 standard. The REJECT construct can only be used with the keyword INERTIAL to include a time parameter that specifies the input pulse width constraint.

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

Inertial Delay (cont.)

- INERTIAL delay is default
  - **Keyword INERTIAL may be left out**
- REJECT is optional

```
Output <= NOT Input AFTER 10 ns;
-- Propagation delay and minimum pulse width are 10ns
```



The keyword INERTIAL may be used in the signal assignment statement to specify an inertial delay, or it may be omitted because inertial delay is the default delay type.

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

Delta Delay

- Default signal assignment propagation delay if no delay is explicitly prescribed
  - **VHDL signal assignments do not take place immediately**
  - **Delta is an infinitesimal VHDL time unit so that all signal assignments can result in signals assuming their values at a future time**

```
Output <= NOT Input;
-- Output assumes new value in one delta cycle
```

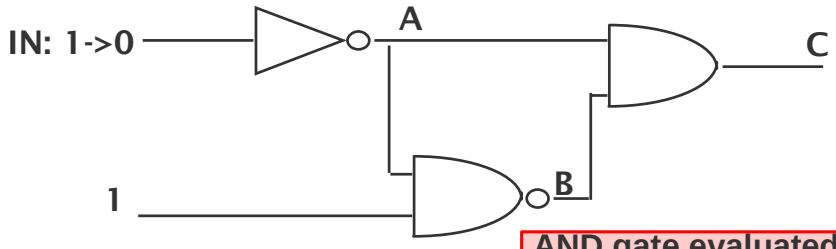
- Supports the model of concurrent VHDL process execution and the two-step simulation model
  - **Order in which processes are executed by simulator does not affect simulation output**

D. Pamunuwa, LU, Lancaster 2006

The delta delay models the fact that any value updates in hardware have some associated physical delay. Since VHDL doesn't require that a delay is always associated with a signal assignment, the delta delay is used to provide a minimum delay in a signal assignment statement. A delta (or delta cycle) is an infinitesimal, but quantized, unit of time. This supports the simulation cycle described earlier.

An Example without Delta Delay

What is the behavior of C?



NAND gate evaluated first:

IN: 1->0
A: 0->1
B: 1->0
C: 0->0

AND gate evaluated first:

IN: 1->0
A: 0->1
C: 0->1
B: 1->0
C: 1->0

If we assume no delta delay mechanism. Then the order in which processes (or components) are executed will affect the model outputs. Consider the example above in which there is a 1 to 0 transition at the input of the Inverter while the other input to the NAND gate is a constant 1. What is the behavior of *C*?

Note that if the NAND gate is evaluated before the AND gate, *C* can remain quiescent at its final value of 0.

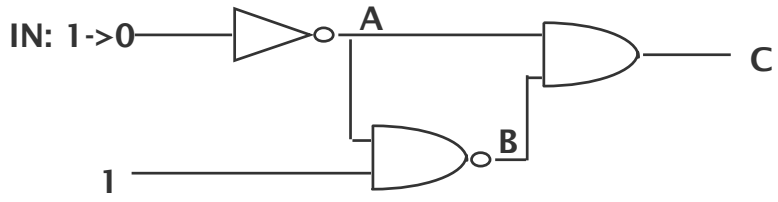
However if the AND gate is evaluated before the NAND gate, a glitch is seen at *C* (i.e. a static-0 hazard is observed). It is generated because the NAND gate has not yet been updated to its new value which will subsequently cause *C* to become 0. Therefore, *C* initially goes to 1 and will only go to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

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

[Perry94], pp. 22-24.

An Example with Delta Delay

- What is the behavior of C?



Using delta delay scheduling

Time	Delta	Event
0 ns	0	IN: 1->0
		eval INVERTER
	1	A: 0->1
		eval NAND, AND
	2	B: 1->0
		C: 0->1
		eval AND
	3	C: 1->0
1 ns		

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

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

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

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

Based on [Perry94], pp 22-24

Signals and Variable Assignments

- A process executes sequential statements that include variable and signal assignment statements
- Signal assignment is not effected immediately
  - **There will be at least a delta delay associated with each signal assignment**
  - **Process has to suspend before signal is updated**
    - **First wait statement or end of process**
  - **The transaction is actually processed in the next simulation cycle**
- Variable assignments are updated immediately

Simulation Example

```
architecture behav of top is
signal x,y,z : integer := 0;
begin
  p1 : process is
  variable a, b : integer := 0;
  begin
        a := a + 20;
        b := b + 10;
        x <= a + b after 10 ns;
        y <= a - b after 20 ns;
        wait for 30 ns;
 end process;

 p2: process is
 begin
       z <= (x + y);
       wait on x,y;
 end process;
end behav;
```

	0 ns	10 ns	20 ns	30 ns
a	20	20	20	40
b	10	10	10	20
x	0	30	30	30
y	0	0	10	10
z	0	30	40	40

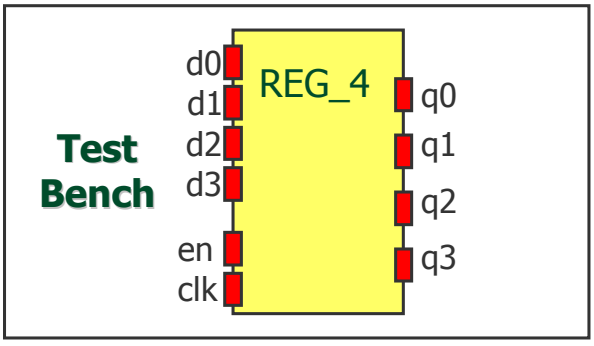
For lecture

Session outline

- Introduction
- VHDL Model Components
- Test Bench
- Basic VHDL Constructs
- Examples
- Summary

VHDL Test Bench

- A VHDL Test bench is a module that is used for testing the functionality of a design module.



- Entity declarations of test bench modules have no input and output ports.

Test Bench

- Testing a design by simulation using a test bench model
  - **Contains an architecture body that includes an instance of the design under test**
  - **Applies sequences of test values to inputs**
  - **Monitors values on output signals**
    - **Either manually**
    - **Or with a process that verifies correct operation**

Test Bench Example

```
entity test_bench is
end entity test_bench;

architecture test_reg4 of test_bench is
    signal d0, d1, d2, d3, en, clk, q0, q1, q2, q3 : bit;
begin
    dut : entity work.reg4(behav)
      port map ( d0, d1, d2, d3, en, clk, q0, q1, q2, q3 );
    stimulus : process is
    begin
      d0 <= '1';  d1 <= '1';  d2 <= '1';  d3 <= '1';  wait for 20 ns;
      en <= '0';  clk <= '0';  wait for 20 ns;
      en <= '1';  wait for 20 ns;
      clk <= '1';  wait for 20 ns;
      d0 <= '0';  d1 <= '0';  d2 <= '0';  d3 <= '0';  wait for 20 ns;
      en <= '0';  wait for 20 ns;
      …
      wait;
    end process stimulus;
end architecture test_reg4;
```

No I/O Ports

Regression Testing

- Test that a refinement of a design is correct
  - **lower-level structural model behaves the same as a behavioral model**
- Test bench includes two instances of design under test
  - **behavioral and lower-level structural model**
  - **stimulates both with same inputs**
  - **compares outputs for equality**
- Need to take account of timing differences

Regression Test Example

```
architecture regression of test_bench is
    signal d0, d1, d2, d3, en, clk : bit;
    signal q0a, q1a, q2a, q3a, q0b, q1b, q2b, q3b : bit;
begin
    dut_a : entity work.reg4(struct)
      port map ( d0, d1, d2, d3, en, clk, q0a, q1a, q2a, q3a );
    dut_b : entity work.reg4(behav)
      port map ( d0, d1, d2, d3, en, clk, q0b, q1b, q2b, q3b );
    stimulus : process is
    begin
      d0 <= '1';  d1 <= '1';  d2 <= '1';  d3 <= '1';  wait for 20 ns;
      en <= '0';  clk <= '0';  wait for 20 ns;
      en <= '1';  wait for 20 ns;
      clk <= '1';  wait for 20 ns;
      …
      wait;
    end process stimulus;
```

Regression Test Example

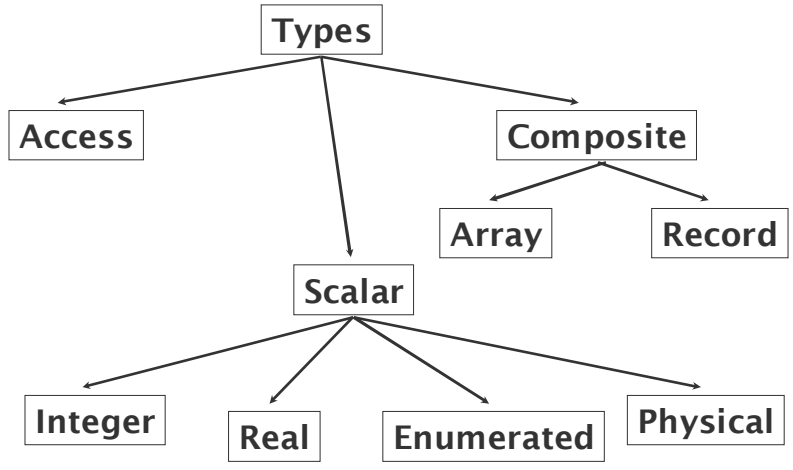
```
…
      verify : process is
      begin
         wait for 10 ns;
         assert q0a = q0b and q1a = q1b and q2a = q2b and q3a = q3b
            report ”implementations have different outputs”
            severity error;
         wait on d0, d1, d2, d3, en, clk;
      end process verify;
end architecture regression;
```

Session outline

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

Data Types

- All declarations of VHDL ports, signals, and variables must specify their corresponding type or subtype



The three defined data types in VHDL are access, scalar, and composite. Note that VHDL 1076-1987 defined a fourth data type, file, but files were reclassified as objects in VHDL 1076-1993. In a nutshell, access types are similar to pointers in other programming languages, scalar types are atomic units of information, and composite types are arrays and/or records.

[Perry94], pp 74

Scalar Types – Integer

- Minimum range for any implementation as defined by standard:
  - **-2,147,483,647 to 2,147,483,647 ($-2^{31}+1$ to $2^{31}-1$)**
- Example assignments to a variable of type integer :

```
ARCHITECTURE test_int OF test IS
BEGIN
       PROCESS (X)
              VARIABLE a: INTEGER;
       BEGIN
              a := 1;  -- OK
              a := -1;  -- OK
              a := 1.0;  -- illegal
       END PROCESS;
END test_int;
```

- Use defined integer types
  - **TYPE month IS RANGE 1 TO 12 ;**
  - **TYPE placing IS RANGE 10 DOWNTO 0;**

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

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

Operators

- Defined precedence levels in decreasing order :
  - **Miscellaneous operators: **, abs, not**
  - **Multiplication operators: *, /, mod, rem**
  - **Sign operator: +, -**
  - **Addition operators: +, -, &**
  - **Shift operators: sll, slr, sal, sar, rol, ror**
  - **Relational operators: =, /=, <, <=, >, >=**
  - **Logical operators: AND, OR, NAND, NOR, XOR, XNOR**
- Operators can be chained to form complex expressions, e.g. :

```
res <= a AND NOT(B) OR NOT(a) AND b;
```

  - **Can use parentheses for readability and to control the association of operators and operands**

The predefined operators in VHDL is shown above.

Integer Type : operations

- Addition: +
- Subtraction or negation: -
- Multiplication: *
- Division: /
- Remainder: rem
- Modulo: mod
- Absolute value: abs
- Exponentiation: **
- Logical: =, /=, <, >, <=, >=

7 rem 4 = 3	**7 mod 4 = 3**
7 rem -4 = 3	**7 mod -4 = -1**
-7 rem 4 = -3	**-7 mod 4 = 1**
-7 rem -4 = -3	**-7 mod -4 = -3**

$a \text{ rem } b = a - rnd(a/b) * b$

$a \text{ mod } b = a - rnd(a/b) * b$ → **if a and b are both of the same sign (same as *rem*)**

$a \text{ mod } b = a + ceil(|a/b|) * b$ → **if a and b are of opposite sign**

Scalar Types – Real

- Minimum range for any implementation as defined by standard: -1.0E38 to 1.0E38
- Example assignments to a variable of type real :

```
ARCHITECTURE test_real OF test IS
BEGIN
    PROCESS (X)
      VARIABLE a: REAL;
    BEGIN
      a := 1.3;  -- OK
      a := -7.5;  -- OK
      a := 1;  -- illegal
      a := 1.7E13;  -- OK
      a := 5.3 ns;  -- illegal
    END PROCESS;
END test_real;
```


The REAL type consists of the real numbers within a simulator-specific (but with a VHDL standard imposed minimum) range. The first illegal statement above attempts to assign an integer to a real type variable, and the second illegal statement is not allowed since the unit “ns” denotes a physical data type.

Real Type: Operations

- Addition: +
- Subtraction or negation: -
- Multiplication: *
- Division: /
- Absolute value: abs
- Exponentiation: **
- Logical: =, /=, <, >, <=, >=

```
x := 5**5  -- 5^5, OK
y := 0.5**3 -- 0.5^3, OK
x := 4**0.5 -- 4^0.5, Illegal
y := 0.5**(-2) -- 0.5^(-2), OK
```


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

Operator Overloading

```
ARCHITECTURE test_overLoad OF test IS
BEGIN
    PROCESS (X)
      VARIABLE a, b: REAL;
      VARIABLE p, q: INTEGER;
    BEGIN
      a := b + 1.3;  -- OK
      p := q + 5;  -- OK
    END PROCESS;
END test_overLoad;
```

How does this work?

Look up Operator Overloading in textbook!

Scalar Types – Enumerated

- User specifies list of possible values
- Example declaration and usage of enumerated data type :

```
TYPE binary IS ( ON, OFF );
......
VARIABLE a: binary;
a := ON;   -- OK
a := OFF;   -- OK
```

- Predefined enumerated types
  - **Boolean**

```
TYPE boolean IS (false, true);
```

  - **Character**

```
TYPE character is ( 'a', 'b', 'c', …….);
```


The enumerated data type is a user specified list of values that a variable or signal of the defined type may take. Its particularly useful in defining the different states of a FSM with descriptive names such ‘init’, ‘reset’ etc.

The members of the enumerated type are explicitly defined.

Scalar Types – Physical

- Require associated units
- Range must be specified
- Example of physical type declaration :

```
TYPE resistance IS RANGE 0 TO 10000000

UNITS
ohm;  -- ohm
Kohm = 1000 ohm;  -- i.e. 1 KΩ
Mohm = 1000 kohm;  -- i.e. 1 MΩ
END UNITS;
```

- Time is the only physical type predefined in VHDL standard

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

One thing to be careful about is that VHDL is not case sensitive so Kohm and kohm refer to the same unit.

The only predefined physical type in VHDL is time.

Attributes

- Attributes provide information about certain items in VHDL
  - **E.g. types, subtypes, procedures, functions, signals, variables, constants, entities, architectures, configurations, packages, components**
  - **General form of attribute use :**

```
name'attribute_identifier  -- read as "tick"
```

- VHDL has several predefined attributes :
  - **X'EVENT -- TRUE when there is an event on signal X**
  - **X'LAST_VALUE -- returns the previous value of signal X**
  - **Y'HIGH -- returns the highest value in the range of Y**
  - **X'STABLE(t) -- TRUE when no event has occurred on signal X in the past 't' time**

D. Pamunuwa, LU, Lancaster 2006

Attributes convey information about different items in VHDL. For example, an attribute can be used to determine the depth of an array, its range, its leftmost index, etc. Additionally, the user may define new attributes. This capability allows user-defined constructs and data types to use attributes. Note that, by convention, the apostrophe marking the use of an attribute is pronounced tick (e.g. 'EVENT is pronounced "tick EVENT").

Attributes of the Scalar Type

- T'left : Left most value of T
- T'right : Right most value of T
- T'low : Least value of T
- T'high : Highest value of T
- T'ascending : true if T is ascending, false otherwise
- T'image(x) : A string representing the value of 'x'
- T'value(s) : The value in T represented by string 's'

Example of Scalar Attributes

```
type minIndex is range 21 downto 11;
```

- minIndex**'left** = 21
- minIndex**'right** = 11
- minIndex**'low** = 11
- miniIndex**'high** = 21
- minIndex**'ascending** = false
- minIndex**'image**(14) = "14"
- minIndex**'value**("20") = 20

Attribute of Discrete Scalar Type

- Discrete types are integers, and all enumerated types
- T'pos(x): position of x in T
- T'val(n): value in T at position n
- T'succ(x): successor of x in T
- T'pred(x): predecessor of x in T
- T'leftof(x): value in T at position one left of x
- T'rightof(x): value in T at position one right of x

Example of Discrete Type Attributes

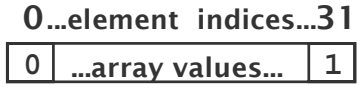
```
type logic_level is (dontCare, low, high, open);
```

- logic_level**'pos**(low) = 2
- logic_level**'val**(1) = dontCare
- logic_level**'succ**(high) = open
- logic_level**'pred**(high) = low

Composite Types – Array

- Used to group elements of the same type into a single VHDL object
- Example declaration for one-dimensional array (vector) :

```
TYPE data_bus IS ARRAY(0 TO 31) OF BIT;
```




```
VARIABLE X : data_bus;
VARIABLE Y : BIT;

Y := X(12);  -- Y gets value of element at index 12
```

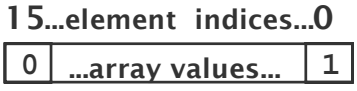

VHDL composite types are arrays and records. Each object of this data type can hold more than one value.

Arrays consist of many similar elements of any data type, including arrays.

Simple Array

- Example one-dimensional array using DOWNTO :

```
TYPE reg_type IS ARRAY(15 DOWNTO 0) OF BIT;
```




```
VARIABLE X : reg_type;
VARIABLE Y : BIT;

Y := X(4);  -- Y gets value of element at index 4
```

- DOWNTO keyword must be used if leftmost index is greater than rightmost index
  - **e.g. 'Big-Endian' bit ordering**

Using the DOWNTO designator in the range specification instead of TO. DOWNTO specifies a descending order in array indices

Arrays with Enumerated Types

```
-- enumerated type
TYPE colour IS (red, green, blue);

-- array of positive integers with enumerated indexing
-- 3 element array
TYPE set1 IS ARRAY (colour) OF natural;

-- array of positive integers with restricted
-- enumerated indexing; 2 element array
TYPE set2 IS ARRAY (colour RANGE green
                                TO blue) OF natural;

-- array of enumerated type with integer indexing
TYPE set3 IS ARRAY (natural) OF colour;
```

Multi-Dimensional Arrays

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TABLE5_2D is array (4 downto 0) of WORD8;
type TABLE2_3D is array (1 downto 0) of TABLE5_2D;

-- Array aggregates: Initialising with positional association
constant CNST_A : TABLE2_3D := (
("0000000","0000001","0000010","0000011","0000100"),
("0010000","0010001","0010010","0100011","0010100"));

-- Array aggregates: Initialising with named association
constant CNST_B : TABLE5_2D := (
 (4=>"00000000",3=>"00000001",2=>"00000010",
    1=>"00000011",0=>"00000100"));

constant CNST_C : TABLE5_2D := (
    (4|2=>"00000001",others=>"00000000"));

-- mixing positional and named association is illegal
constant CNST_D : TABLE5_2D := (
    ("00100000", 3|2=>"00000001",others=>"00000000"));
```


Illustrates multidimensional arrays, and also different ways of accessing the array elements. 3D arrays are supported by Xilinx XST for FPGA synthesis.

Accessing Multi-Dimensional Arrays

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TABLE5_2D is array (4 downto 0) of WORD8;
type TABLE2_3D is array (1 downto 0) of TABLE5;

signal WORD_A : WORD8;
signal TABLE_A, TABLE_B : TABLE5_2D;
signal TABLE_C, TABLE_D : TABLE2_3D;

-- using just one index
TABLE_A (4) <= WORD_A;
TABLE_C (1) <= TABLE_A;

-- using maximum number of indices
TABLE_A (4) (0) <= '1';
TABLE_C (1) (3) (0) <= '0';

-- using slices
TABLE_A (4 downto 1) <= TABLE_B (3 downto 0);
TABLE_C (1) (4) (3 downto 0) <= TABLE_B (3) (4 downto 1);
```


Illustrates multidimensional arrays, and also different ways of accessing the array elements. 3D arrays are supported by Xilinx XST for FPGA synthesis.

Accessing Multi-Dimensional Arrays of Matrix Type

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);

type MATRIX_3D is array(4 downto 0, 2 downto 0)
  of STD_LOGIC_VECTOR (7 downto 0);

signal word_a : WORD8;

signal matrix_1: MATRIX_3D;

-- note the different indexing

Matrix_1 (1,4) <= word_a; -- legal

Matrix_1 (1,4)(7) <= '1'; -- legal

Matrix_1 (1,4,6) <= '0'; -- illegal
```


Illustrates multidimensional arrays, and also different ways of accessing the array elements. 3D arrays are supported by Xilinx XST for FPGA synthesis.

Array attributes

- A**'left**(N) – Left bound of index range of dim. N of A
- A**'right**(N) – Right bound of index range of dim. N of A
- A**'low**(N) – Lower bound of index range of dim. N of A
- A**'high**(N) – Upper bound of index range of dim. N of A
- A**'range**(N) – Index range of dim. N of A
- A**'reverse_range**(N) – Reverse index range of dim. N of A
- A**'length**(N) – Length of index range of dim. N of A
- A**'ascending**(N) – true if index range of dim. N of A is ascending, false otherwise.

Unconstrained Array Types

- Possible to declare an unconstrained array type
  - **Constrained at time of object declaration**

```
type studentNo is array (natural range <>) of integer;
...
...
variable MSoC: studentNo (0 to 8);
```

  - **The <> symbols act as placeholders for the range**
- Predefined unconstrained arrays
  - **String – unconstrained array of type character**

```
variable name: string (0 to 11) := (others => " ");
```

  - **bit_vector – unconstrained array of type bit**

```
variable byte: bit_vector(0 to 7);
```

Array operations

- and, or, nand, nor, xor and xnor
  - **one-dimensional equal sized and same type bit or boolean arrays.**
- sll, slr
  - **one-dimensional bit or boolean arrays**
  - **shifting by filling in with zero**
- sal or sar
  - **one-dimensional bit or boolean arrays**
  - **shifting by filling in with copies of element**
- rol and ror
  - **one-dimensional bit or boolean arrays**
  - **rotate**
- Relational operators: <, >, <=, >=, =, /=
- Concatenation: &

There are two basic forms of shift: Shift Logical and Shift Arithmetic, in either direction of left or right.

Shift Logical Left (SLL), Shift Logical Right (SLR), Shift Arithmetic Right (SAR), Shift Arithmetic Left (SAL)

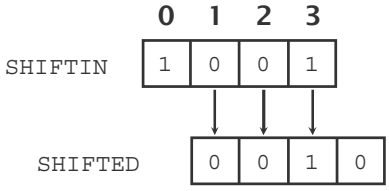
The "Logical" in "Shift Logical Left" and "Shift Logical Right“ refers to the fact that the sign bit of the number is ignored --the number is considered to be a sequence of bits that is either an unsigned quantity or a non-numeric quantity such as a graphic. When a number is shifted in this manner, the sign bit is not preserved. Logical shifts always bring in '0's in the bit positions being "shifted in". The "Arithmetic" in "Shift Arithmetic Right“ and "Shift Arithmetic Left“ refers to the fact that the sign-bit is replicated as the number is shifted, thereby

preserving the sign of the number. This is referred to as "sign extension", and is useful when dividing signed numbers by powers of 2.

Operators – Examples

- The concatenation operator &

```
VARIABLE shifted, shiftin : BIT_VECTOR(0 TO 3);
                    ...
        shifted := shiftin(1 TO 3) & '0';
```



The concatenation operator joins two vectors together. Both vectors must be of the same type. Shown is a logical shift left for a four bit array.

Composite Types – Records

- Used to group elements of possibly different types into a single VHDL object
- Elements are indexed via field names
- Examples of record declaration and usage :

```
TYPE binary IS ( ON, OFF );
TYPE switch_info IS
   RECORD
     status : BINARY;
     IDnumber : INTEGER;
END RECORD;

VARIABLE switch : switch_info;
switch.status := ON;  -- status of the switch
switch.IDnumber := 30;  -- e.g. number of the switch
```


The second VHDL composite type is the record. An object of type record may contain elements of different types. A TYPE declaration is used to define a record. Note that the types of a record's elements must be defined before the record is defined. Note that there is no semi-colon after the word RECORD. The RECORD and END RECORD keywords demarcate the field names.

Access Type

- Analogous to pointers in other languages
- Allows for dynamic allocation of storage
- Useful for implementing queues, fifos, etc.
- Not discussed in this module

The access type is similar to a pointer in other programming languages allowing dynamic allocation and deallocation of storage space to an object. Useful for implementing abstract data structures (such as queues and first-in-first-out buffers) where the size of the structure may not be known at compile time. Not synthesisable, hence we wont discuss this any further.

Subtypes

- Subtype
  - **Allows for user defined constraints on a data type**
    - **e.g. a subtype based on an unconstrained VHDL type**
  - **May include entire range of base type**
  - **Assignments that are out of the subtype range are illegal**
    - **Range violation detected at run time rather than compile time because only base type is checked at compile time**
  - **Subtype declaration syntax :**

```
SUBTYPE name IS base_type RANGE <user range>;
```

  - **Subtype example :**

```
SUBTYPE first_ten IS INTEGER RANGE 0 TO 9;
```


VHDL subtypes are used to constrain defined types. Constraints take the form of range constraints or index constraints. A subtype may include the entire range of the base type. Assignments made to objects that are out of the subtype range generate an error at run time.

Potential Problems to Avoid with Subtype

- Objects defined by subtypes derived from a base type are considered to be of the same type
  - **Example**

```
PROCESS
    SUBTYPE smallintA IS INTEGER RANGE 0 TO 10;
    SUBTYPE smallintB IS INTEGER RANGE 0 TO 15;
    VARIABLE A: smallintA := 5;
    VARIABLE B: smallintB := 8;
    VARIABLE C: INTEGER;
BEGIN
    B := B * A;     -- OK
    C := B + 1;     -- OK
END;
```

Compilation OK, run time error! ☹

Though VHDL is strongly typed, but only with respect to the base type, not derived subtypes.

Thus, the VHDL analyzer will not be aware of inconsistent subtypes as shown here, and the simulator will execute the statements as expected. Note, however, that the result after multiplying A and B may be out of the range of B's subtype resulting in a runtime subtype range violation.

VHDL Data Types – Summary

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

VHDL Objects

- There are four types of objects in VHDL
  - **Constants**
  - **Variables**
  - **Signals**
  - **Files**
- The scope of an object is as follows :
  - **Objects declared in a package are available to all VHDL descriptions that *use* that package**
  - **Objects declared in an entity are available to all architectures associated with that entity**
  - **Objects declared in an architecture are available to all statements in that architecture**
  - **Objects declared in a process are available only within that process**

VHDL Objects – Constants

- Name assigned to a specific value of a type
- Declaration of constant may omit value so that the value assignment may be deferred
  - **Facilitates reconfiguration**
  - **Allow for easy update and readability**
- Declaration syntax :

```
CONSTANT constant_name : type_name [:= value];
```

- Declaration examples :

```
CONSTANT PI : REAL := 3.14;
CONSTANT SPEED : INTEGER;
```


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

VHDL Objects – Variables

- Provide convenient mechanism for local storage
  - **E.g. loop counters, intermediate values**
- Scope is process in which they are declared
  - **VHDL '93 provides for global variables**
- All variable assignments take place immediately
  - **No delta or user specified delay is incurred**
- Declaration syntax:

```
VARIABLE variable_name : type_name [:= value];
```

- Declaration examples :

```
VARIABLE opcode : BIT_VECTOR(3 DOWNTO 0) := "0000";
VARIABLE freq : INTEGER;
```


An assignment to a VHDL variable results in the variable assuming its new value immediately (unlike for VHDL signals where simulation time or delta cycles must transpire). This allows sequential execution of statements within VHDL processes where variables are used as placeholders for temporary data, loop counters, etc.

When a variable is declared, it may optionally be given an initial value as well.

VHDL Objects – Signals

- Used for communication between VHDL components
- Real, physical signals in system often mapped to VHDL signals
- ALL VHDL signal assignments require either delta cycle or user-specified delay before new value is assumed
- Declaration syntax :

```
SIGNAL signal_name : type_name [:= value];
```

- Declaration and assignment examples :

```
SIGNAL dir : BIT;
dir <= '0' AFTER 5ns, '1' AFTER 10ns;
```


D. Pamunuwa, LU, Lancaster 2006

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

VHDL Objects – Files

- Files provide a way for a VHDL design to communicate with the host environment
- File declarations make a file available for use to a design
- Files can be opened for reading and writing
  - **In VHDL87, files are opened and closed when their associated objects come into and out of scope**
  - **In VHDL93 explicit `FILE_OPEN()` and `FILE_CLOSE()` procedures were added**
- The package STANDARD defines basic file I/O routines for VHDL types
- The package TEXTIO defines more powerful routines handling I/O of text files

Files may be opened in read or write mode, and once a file is opened, its contents may only be accessed sequentially. For detailed description of the use of file objects refer the textbook.

Packages and Libraries

- Packages and libraries provide the ability to reuse constructs in multiple entities and architectures
  - **User defined constructs declared inside architectures and entities are not visible to other VHDL components**
    - **Scope of subprograms, user defined data types, constants, and signals is limited to the VHDL components in which they are declared**
  - **Items declared in packages can be *used* (i.e. included) in other VHDL components**

VHDL provides the package mechanism so that user-defined types, subprograms, constants, aliases, etc. can be defined once and reused in the description of multiple VHDL components. VHDL libraries are collections of packages, entities, and architectures. The use of libraries allows the organization of the design task into any logical partition the user chooses (e.g. component libraries, package libraries to house reusable functions and type declarations).

Packages

- Packages consist of two parts
  - **Package declaration -- contains declarations of objects defined in the package**
  - **Package body -- contains necessary definitions for certain objects in package declaration**
    - **e.g. subprogram descriptions**
- Examples of VHDL items included in packages :
  - **Basic declarations**
    - **Types, subtypes**
    - **Constants**
    - **Subprograms**
    - **Use clause**
  - **Signal declarations**
  - **Attribute declarations**
  - **Component declarations**

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

Packages – Declaration

- An example of a package declaration :

```
PACKAGE my_stuff IS
       TYPE binary IS ( ON, OFF );
       CONSTANT PI : REAL := 3.14;
       CONSTANT My_ID : INTEGER;
       PROCEDURE add_bits3(SIGNAL a, b, en : IN BIT;
              SIGNAL temp_result, temp_carry : OUT BIT);
END my_stuff;
```

- Note some items only require declaration while others need further detail provided in subsequent package body
  - **for type and subtype definitions, declaration is sufficient**
  - **subprograms require declarations and descriptions**

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

Packages – Package Body

- The package body includes the necessary functional descriptions needed for objects declared in the package declaration
  - **e.g. subprogram descriptions, assignments to constants**

```
PACKAGE BODY my_stuff IS
   CONSTANT My_ID : INTEGER := 2;

   PROCEDURE add_bits3(SIGNAL a, b, en : IN BIT;
      SIGNAL temp_result, temp_carry : OUT BIT) IS
   BEGIN    -- this function can return a carry
      temp_result <= (a XOR b) AND en;
      temp_carry <= a AND b AND en;
   END add_bits3;
END my_stuff;
```


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

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

Packages – Use Clause

- Packages must be made visible before their contents can be used
  - **The USE clause makes packages visible to entities, architectures, and other packages**

```
-- use only the binary and add_bits3 declarations
USE my_stuff.binary, my_stuff.add_bits3;

... ENTITY declaration...
... ARCHITECTURE declaration ...
```

```
-- use all of the declarations in package my_stuff
USE my_stuff.ALL;

... ENTITY declaration...
... ARCHITECTURE declaration ...
```


Once a package is defined, its contents are made visible to VHDL entities and architectures via a USE clause which is analogous to the *include* statement of some other programming languages. This statement comes at the beginning of the entity or architecture file and makes the contents of a package available within that file.

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

Libraries

- Analogous to directories of files
  - **VHDL libraries contain analyzed (i.e. *compiled*) VHDL entities, architectures, and packages**
- Facilitate administration of configuration and revision control
  - **E.g. libraries of previous designs**
- Libraries accessed via an assigned logical name
  - **Current design unit is compiled into the *Work* library**
  - **Both *Work* and *STD* libraries are always available**
  - **Many other libraries usually supplied by VHDL simulator vendor**
    - **E.g. proprietary libraries and IEEE standard libraries**

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

Session outline

- Introduction
- VHDL Model Components
- Test Bench
- Basic VHDL Constructs
- Examples
- Summary

Register Example

- The following example shows a better attempt at a register
- Specifications :
  - **Triggers on rising clock edge**
  - **Latches only on enable high**
  - **Has a data setup time of x_setup**
  - **Has propagation delay of prop_delay**

```
ENTITY 8_bit_reg IS
   GENERIC (x_setup, prop_delay : TIME);
   PORT(enable, clk : IN qsim_state;
      a : IN qsim_state_vector(7 DOWNTO 0);
      b : OUT qsim_state_vector(7 DOWNTO 0));
END 8_bit_reg;
```

qsim_state enumerated type - logic values 0, 1, X, Z

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

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

Attributes – Register Example (cont.)

- The following architecture is a first attempt at the register
- The use of 'STABLE is to used to detect setup violations in the data input

```
ARCHITECTURE first_attempt OF 8_bit_reg IS
BEGIN
    PROCESS (clk)
    BEGIN
    IF (enable = '1') AND a'STABLE(x_setup) AND
        (clk = '1') THEN
       b <= a AFTER prop_delay;
    END IF;
    END PROCESS;
END first_attempt;
```

- What happens if a does not satisfy its setup time requirement of x_setup?

Attributes – Register Example (cont.)

- The following architecture is a second and more robust attempt
- The use of 'LAST_VALUE ensures the clock is rising from a value of ‘0’

```
ARCHITECTURE behavior OF 8_bit_reg IS
BEGIN
    PROCESS (clk)
    BEGIN
    IF (enable = '1') AND a'STABLE(x_setup) AND
        (clk = '1') AND (clk'LAST_VALUE = '0') THEN
       b <= a AFTER prop_delay;
    END IF;
    END PROCESS;
END behavior;
```

- An ELSE clause could be added to define the behavior when the requirements are not satisfied

Summary

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