Embedded Systems

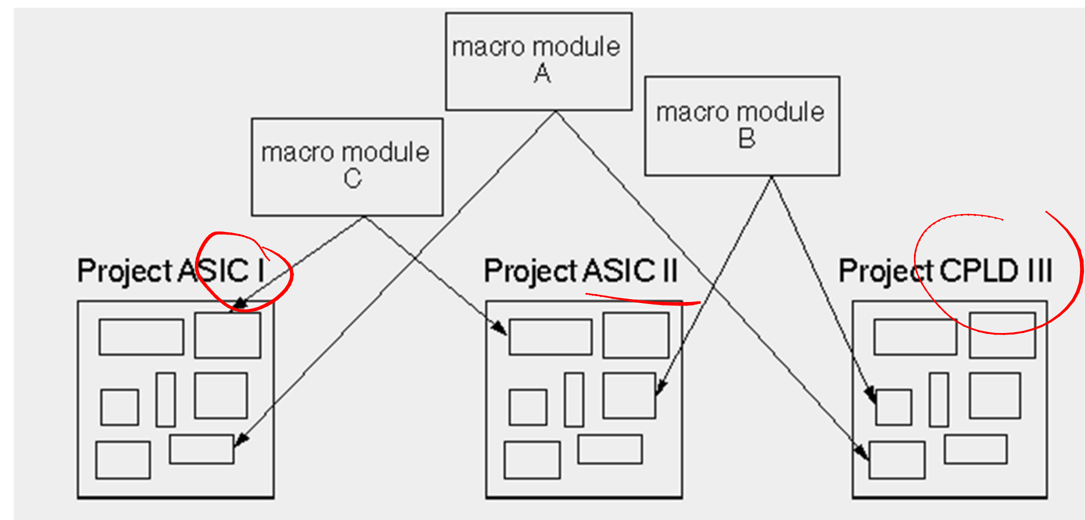
REVIEW

Hardware/System description languages

- VDHL
  - VHDL-AMS
- SystemC
  - TLM

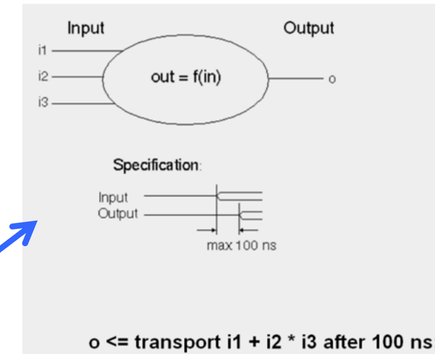
VHDL

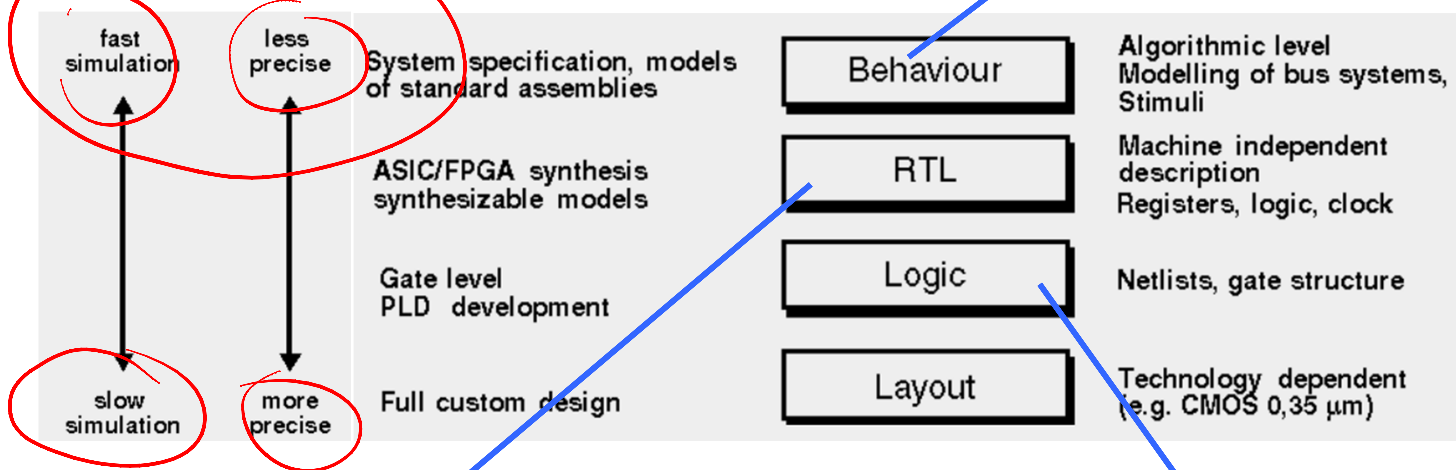
- Main goal was modeling of digital circuits
  - Modelling at various levels of abstraction
  - Technology-independent
    Re-Usability of specifications

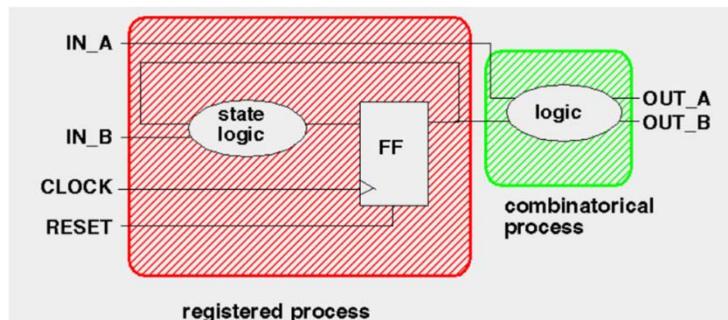


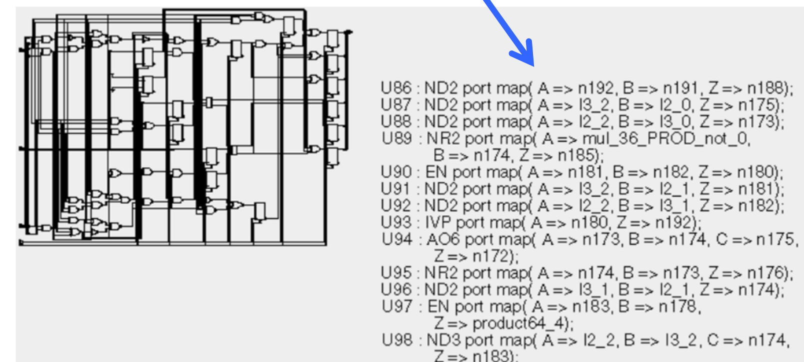
Abstraction Levels

REVIEW



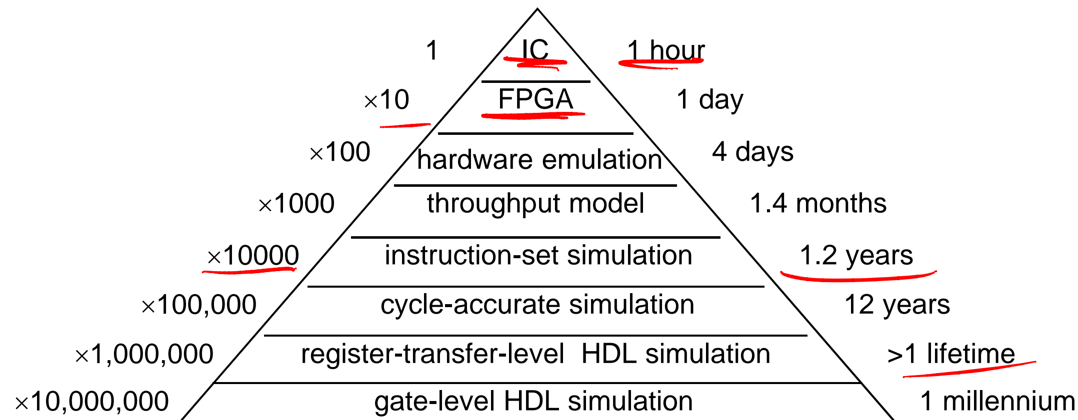




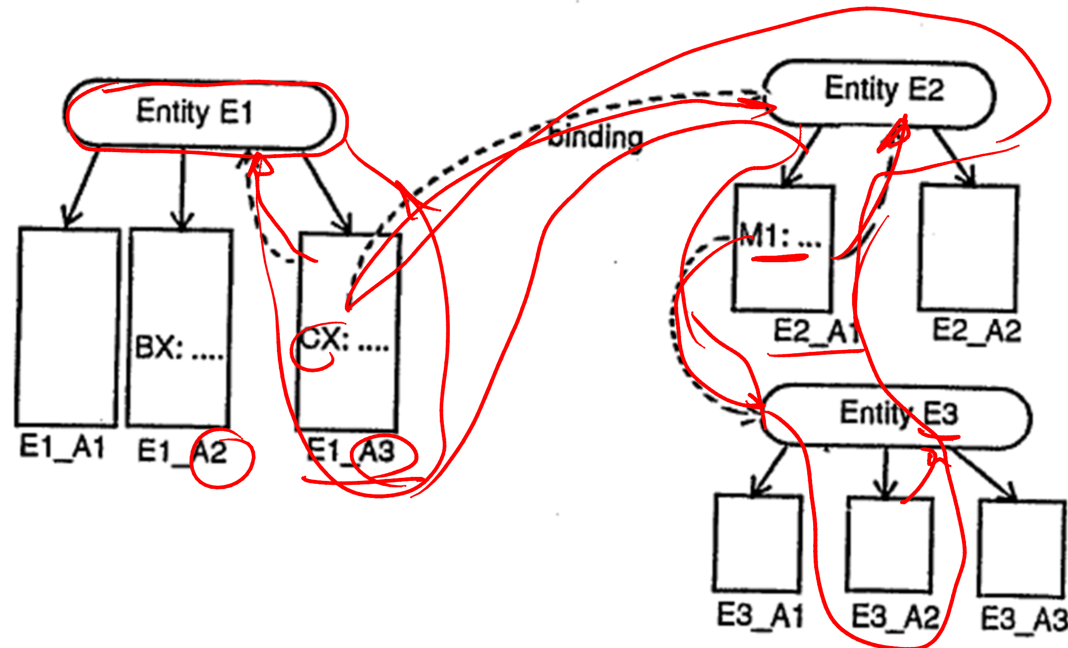


Simulation speed

- Relative speeds of different types of simulation/emulation
  - 1 hour actual execution of SOC
    - = 1.2 years instruction-set simulation
    - = 10,000,000 hours gate-level simulation



Entity/Architectures



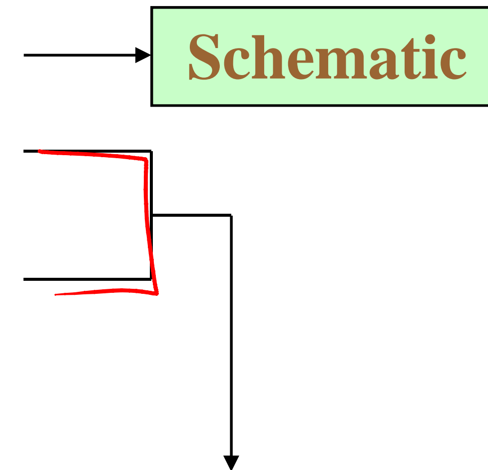
- Architecture body E1_A3 is bound to entity E1
- Architecture body E2_A1 is bound to entity E2
- Component M1 in the AB E2_A1 is bound to entity E3
- Component CX in the AB E1_A3 is bound to entity E2

Definitions of the Description Methods

- *Structural Description Method:* expresses the design as an arrangement of interconnected components
  - It is basically schematic
- *Behavioral Description Method:* describes the functional behavior of a hardware design in terms of circuits and signal responses to various stimuli
  - The hardware behavior is described algorithmically
- *Data-Flow Description Method:* is similar to a register-transfer language
  - This method describes the function of a design by defining the flow of information from one input or register to another register or output

Design Description Methods

- *Structural Description Method*
- *Behavioral Description Method*
- *Data-Flow Description Method*



- These two are similar in that both use a process to describe the functionality of a circuit

VHDL Mixed Modeling

- The three types of modeling can be combined within an architecture body
  - component instantiation – structural
  - concurrent signal assignments – dataflow
  - process statements – behavior
- This is called the *mixed* style of modeling

VHDL Mixed Modeling

```
entity FULL_ADDER is
     port (A, B, CIN: in BIT; SUM, COUT: out BIT);
end FULL_ADDER;

architecture FA_MIXED of FULL_ADDER is
     component XOR2
          port (P1, P2: in BIT; PZ: out BIT);
     end component;
     signal S1: BIT;
begin
     X1: XOR2 port map (A, B, S1);              -- structure
     process (A, B, CIN)                        -- behavior
          variable T1, T2, T3: BIT;
     begin
          T1 := A and B;
          T2 := B and CIN;
          T3 := A and CIN;
          COUT <= T1 or T2 or T3;
     end process;
     SUM <= S1 xor CIN;                         -- dataflow
end FA_MIXED;
```

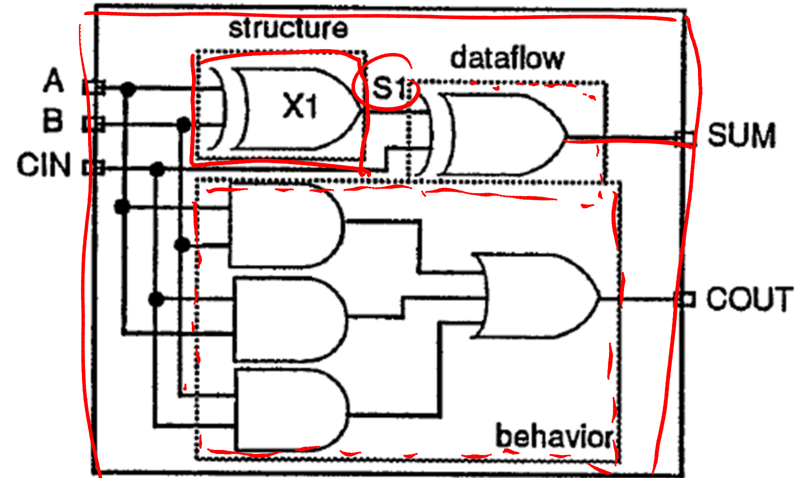
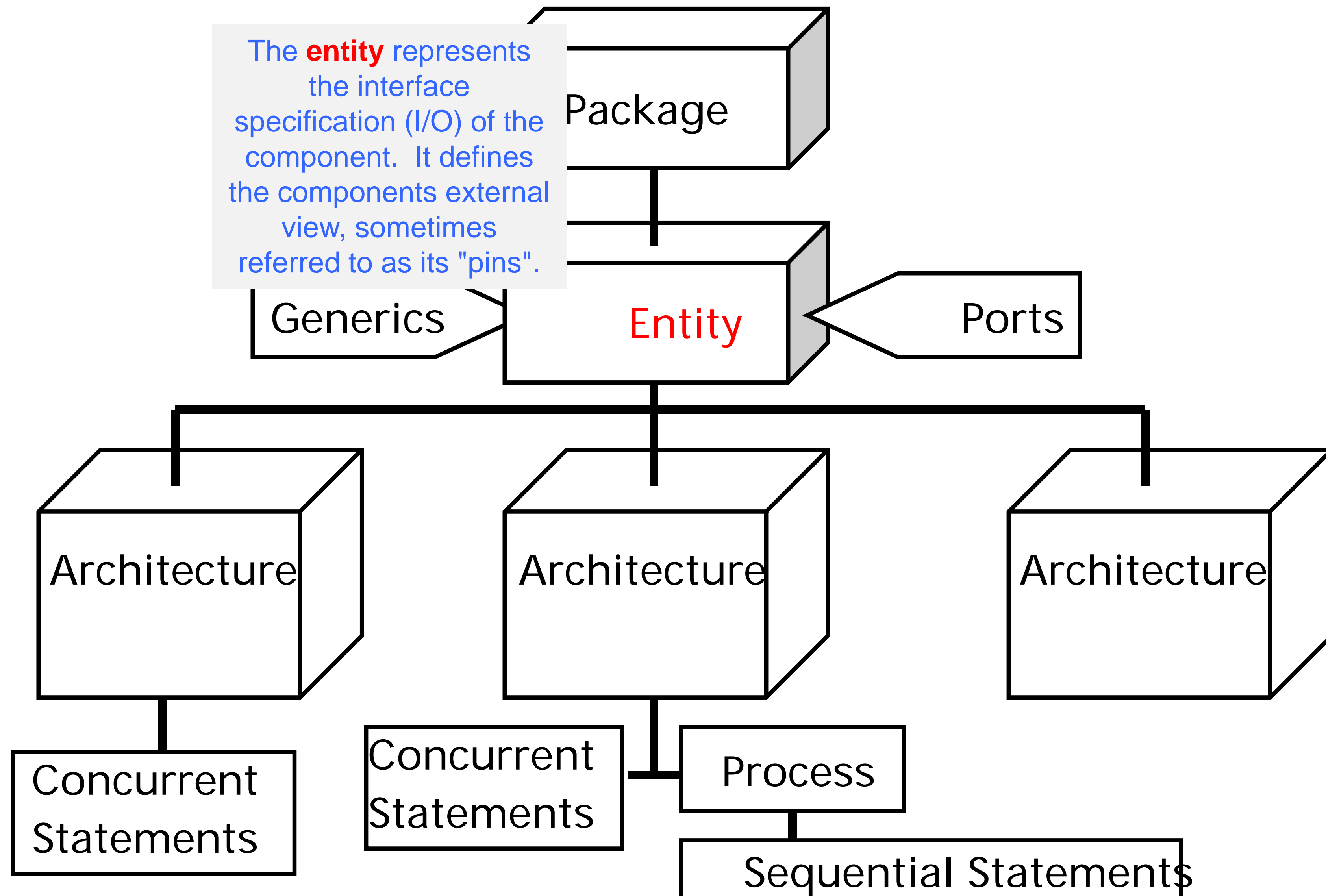


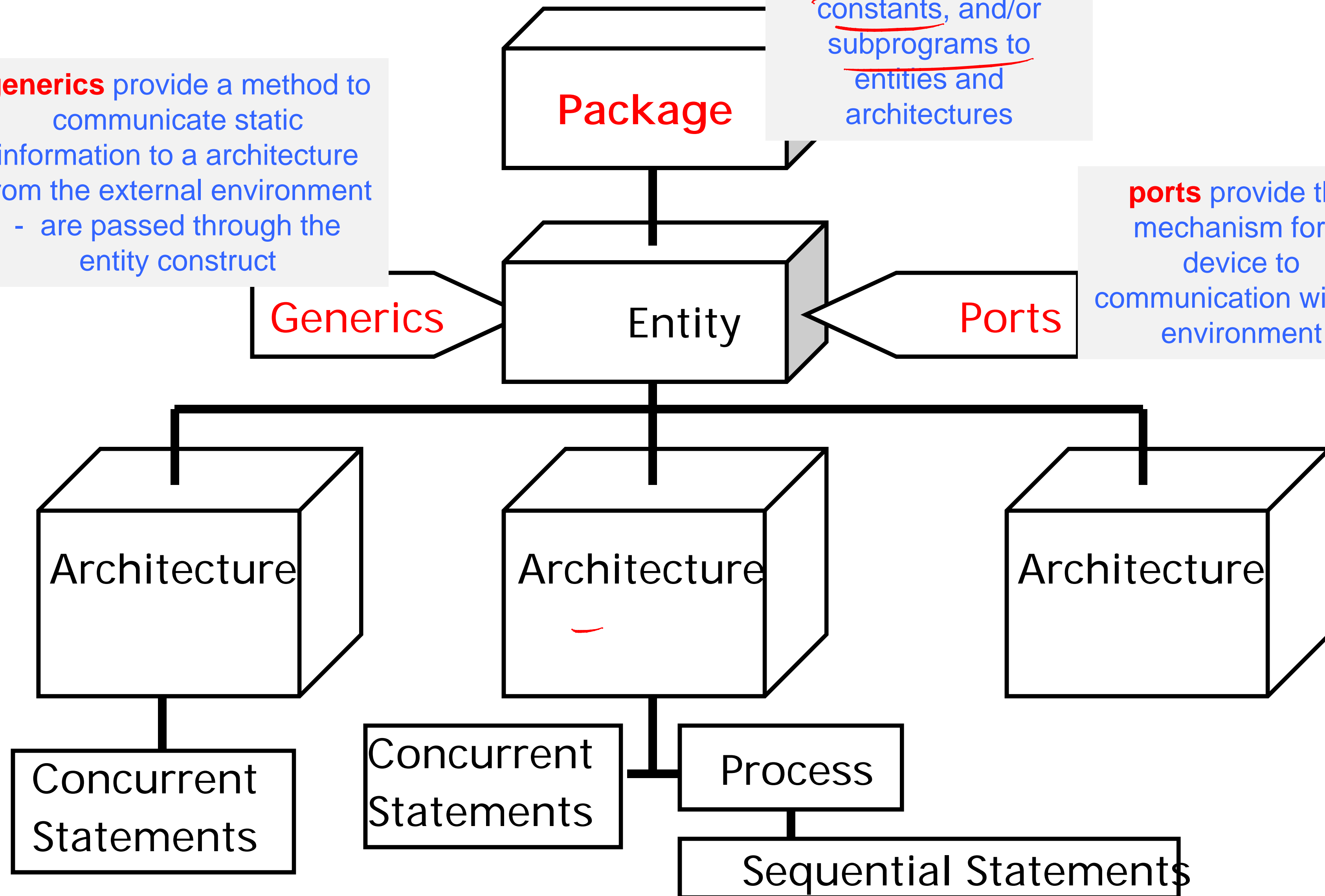
Figure 2.7 A 1-bit full-adder.

Putting It All Together

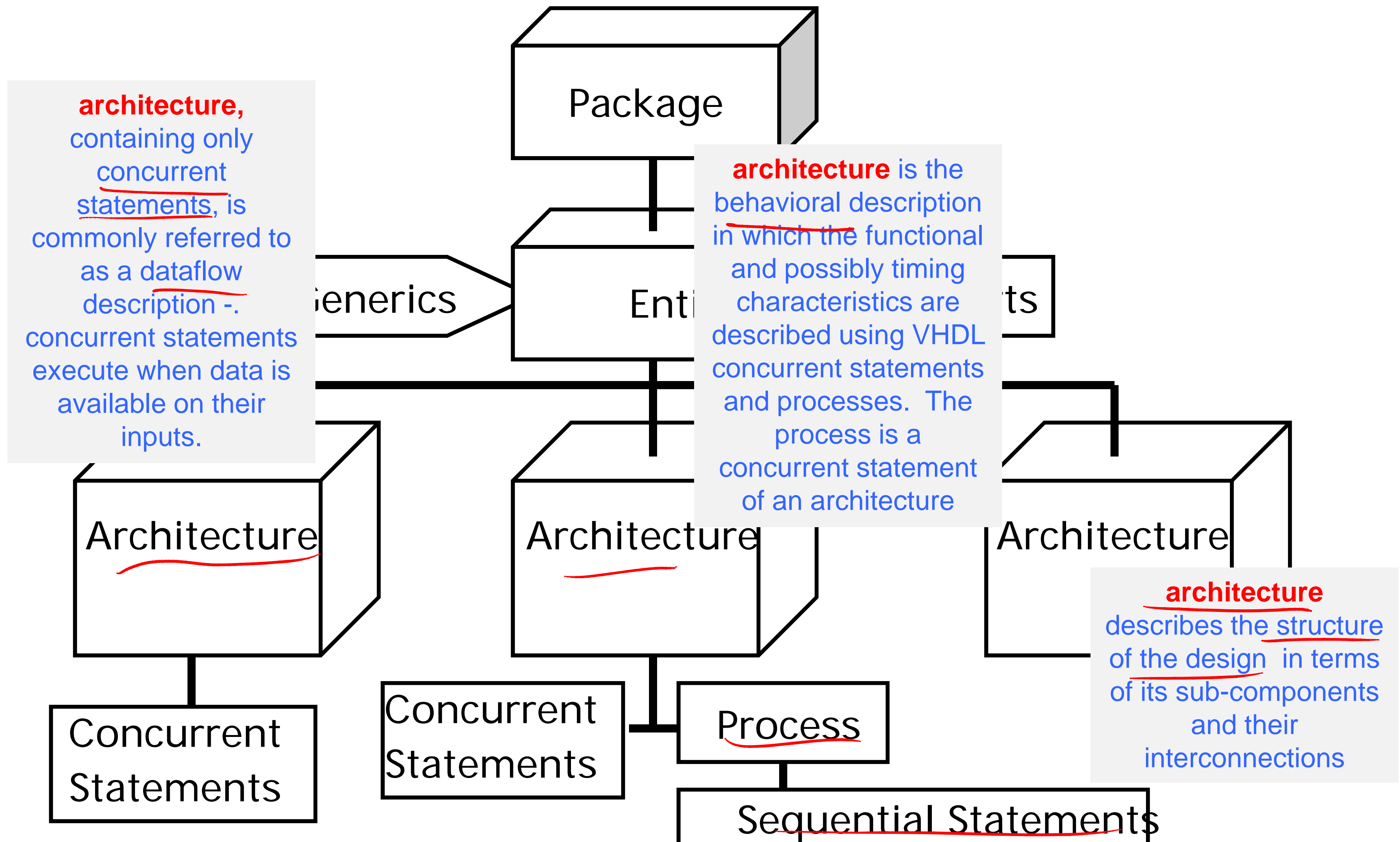


REVIEW

Putting It All Together

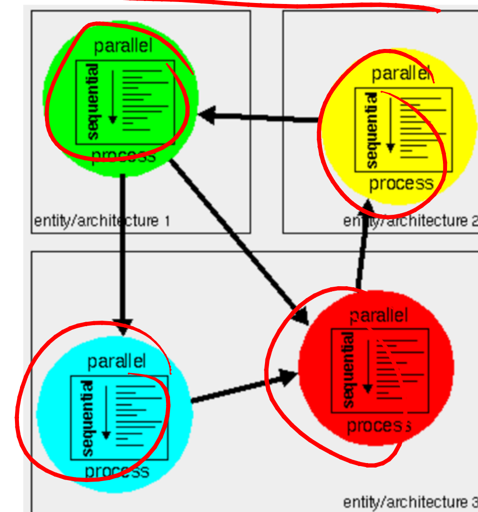


Putting It All Together



Concurrent Statements

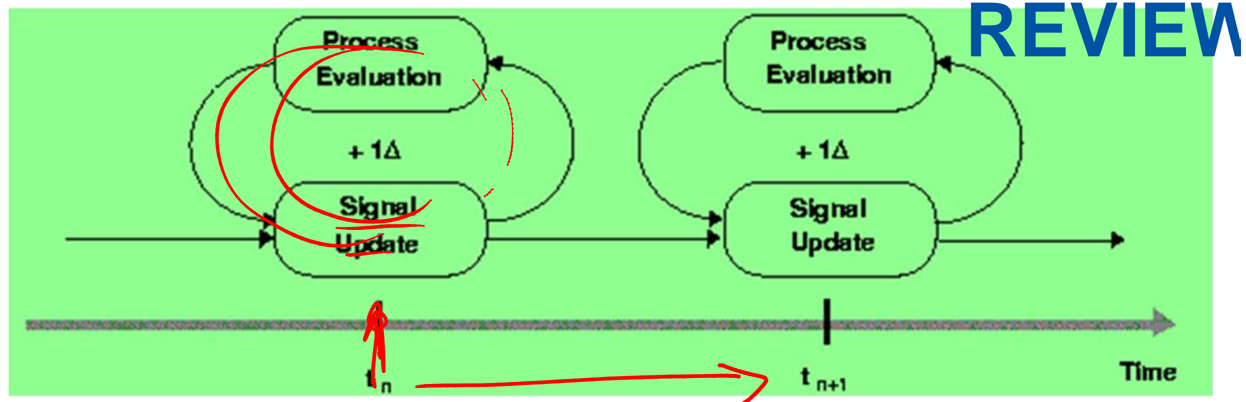
- Basic granularity of concurrency is the *process*
  - Processes are executed concurrently
  - Concurrent signal assignment statements are one-line processes



- Mechanism for achieving concurrency :
  - Processes communicate with each other via signals
  - Signal assignments require delay before new value is assumed
  - Simulation time advances when all active processes complete
  - Effect is concurrent processing
    - I.e. order in which processes are actually executed by simulator does not affect behavior

Delta Delay

REVIEW



1) all active processes can execute in the same simulation cycle e.g., t_n

2) each active process will suspend at wait statement (sensitive list → process finish)

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

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

5) all processes are suspended and no signal update:

$t_n \rightarrow t_{n+1}$ (new entries in the event queue)

The Full Adder

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Truth Table

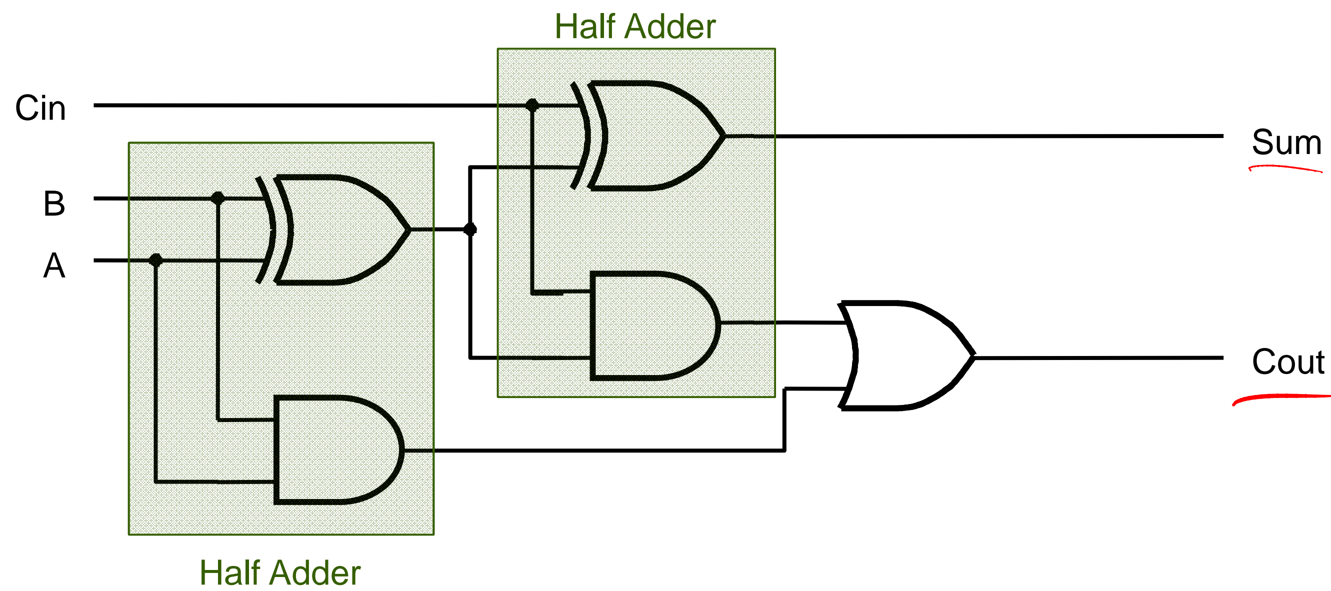
```
  0      0      0      0
  0      0      1      1
+ 0    + 1    + 0    + 1
  0      1      1     10

  1      1      1      1
  0      0      1      1
+ 0    + 1    + 0    + 1
  1     10     10     11
```

Carry-in

Carry-out Sum

The Full Adder



The Full Adder in VHDL

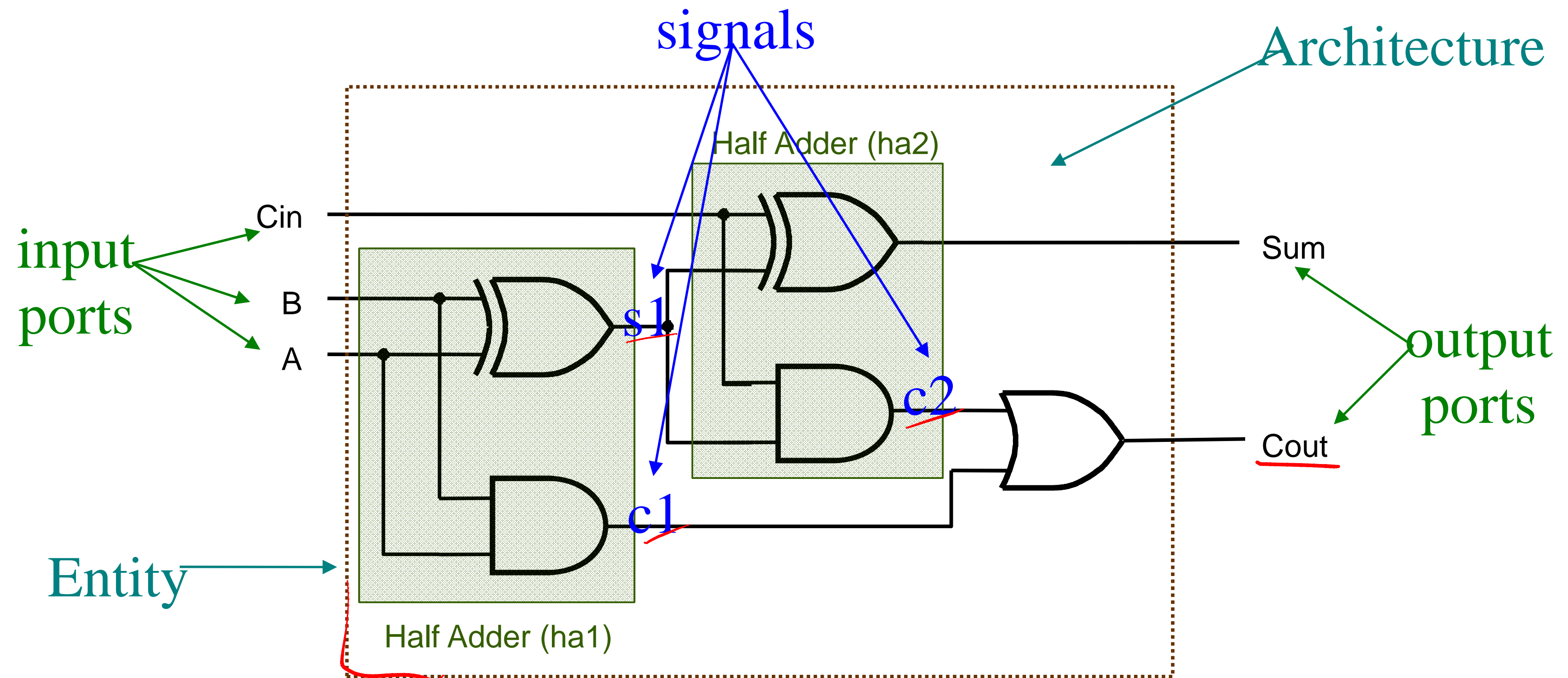
```
19  ----------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  -- Uncomment the following library declaration if using
24  -- arithmetic functions with Signed or Unsigned values
25  --use IEEE.NUMERIC_STD.ALL;
26
27  -- Uncomment the following library declaration if instantiating
28  -- any Xilinx primitives in this code.
29  --library UNISIM;
30  --use UNISIM.VComponents.all;
31
32  entity full_adder_1 is
33      Port ( A,B,Cin : in  STD_LOGIC;
34             Sum,Cout : out  STD_LOGIC);
35  end full_adder_1;
36
37  architecture Bool_exp of full_adder_1 is
38
39  begin
40
41     Sum <= A xor B xor Cin;
42     Cout <= (A and B) or (A and Cin) or (B and Cin);
43
44  end Bool_exp;
```

The Full Adder in VHDL

- Construct Full Adder from two Half Adders
- Use Structural VHDL
- Realize using hierarchical design
  - Design half adder
  - Interconnect half adders
  - Include any additional logic

The Full Adder in VHDL

(Using the structural model)



20

The Full Adder in VHDL

(The Package File)

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

PACKAGE halfadd_package IS
        COMPONENT halfadd
                PORT ( A, B: IN STD_LOGIC ;
                        Sum, Cout: OUT  STD_LOGIC ) ;
        END COMPONENT ;
END halfadd_package ;
```

The Full Adder in VHDL

(The Design File)

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

ENTITY fulladd IS
        PORT ( Cin, A, B : IN STD_LOGIC ;
               Sum, Cout :  OUT    STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE Structure OF fulladd IS
    SIGNAL s1, c1, c2:        STD_LOGIC ;
BEGIN
    ha1 : halfadd PORT MAP ( A => A, B => B, Sum => s1, Cout => c1 ) ;
    ha2 : halfadd PORT MAP ( s1, Cin, Sum, c2 );
    Cout <= c1 OR c2 ;
END Structure ;
```

Module Outline

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

- Several alternative architectures may be developed for use with the same entity

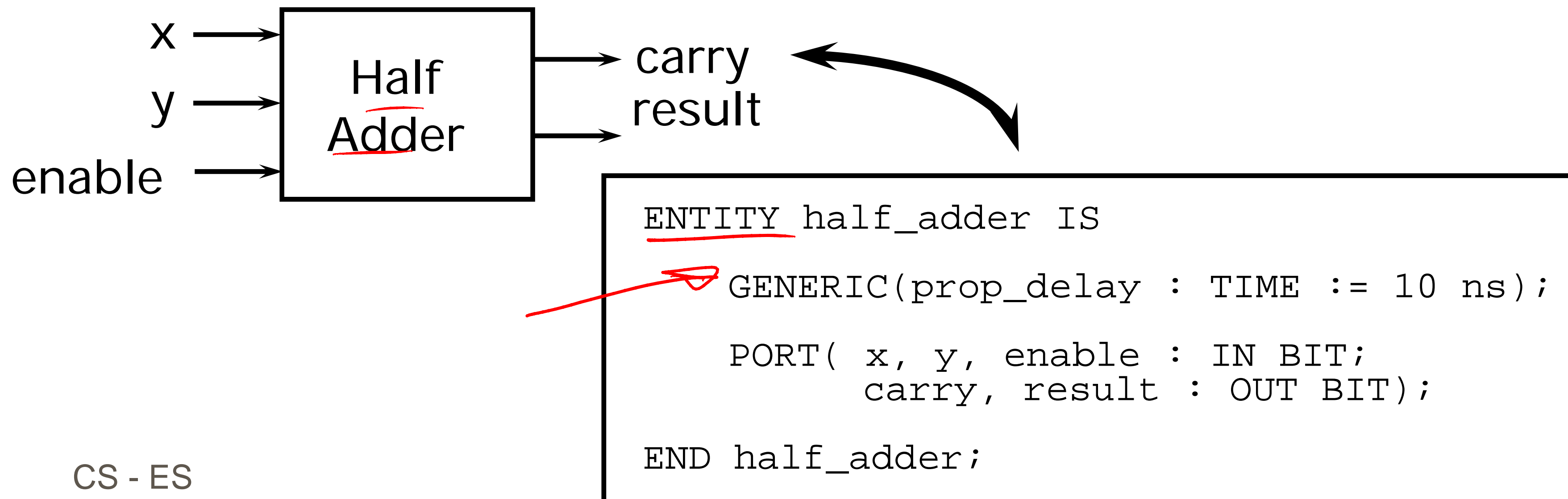
VHDL Model Components (cont.)

- Fundamental unit for component behavior description is the *process*
  - Processes may be explicitly or implicitly defined and are packaged in architectures
- 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 the entities associated with the two architectures
  - Example signal assignment statement :

```
Output <= My_id + 10;
```

Entity Declarations

- The primary purpose of the entity is to declare the signals in the component's interface
  - The interface signals are listed in the PORT clause
    - In this respect, the *entity* is akin to the schematic *symbol* for the component



Entity Declarations
Port Clause

- PORT clause declares the interface signals of the object to the outside world

- Three parts of the PORT clause
  - Name
  - Mode
  - Data type

```
PORT (signal_name : mode data_type);
```

- Example PORT clause:

```
PORT ( input : IN BIT_VECTOR(3 DOWNTO 0);
       ready, output : OUT BIT );
```

  - Note port signals (i.e. ‘ports’) of the same mode and type or subtype may be declared on the same line

Entity Declarations
Port Clause (cont.)

- The port mode of the interface describes the direction in which data travels with respect to the *component*
- The five available port modes are:
  - In - data comes in this port and can only be read
  - Out - data travels out this port
  - Buffer - data may travel in either direction, but only one signal driver may be on at any one time
  - Inout - data may travel in either direction with any number of active drivers allowed; requires a Bus Resolution Function

Entity Declarations
Generic Clause

- Generics may be used for readability, maintenance and configuration
- Generic clause syntax :

```
GENERIC (generic_name : type [:= default_value]);
```

- Generic clause example :

```
GENERIC (My_ID : INTEGER := 37);
```

  - The generic *My_ID*, with a default value of 37, can be referenced by any architecture of the entity with this generic clause
  - The default can be overridden at component instantiation

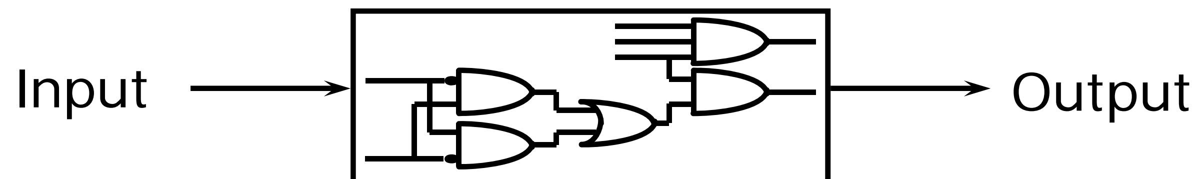
Architecture Bodies

- Describe the operation of the component
- 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 half_adder_d OF half_adder IS
    SIGNAL xor_res : BIT;       -- architecture declarative part
    BEGIN                       -- begins architecture statement part
      carry <= enable AND (x AND y);
      result <= enable AND xor_res;
      xor_res <= x XOR y;
    END half_adder_d;
```

Structural Descriptions

- Pre-defined VHDL components are 'instantiated' and connected together
- Structural descriptions may connect simple gates or complex, abstract components
- Mechanisms for supporting hierarchical description
- Mechanisms for describing highly repetitive structures easily



Behavioral Descriptions

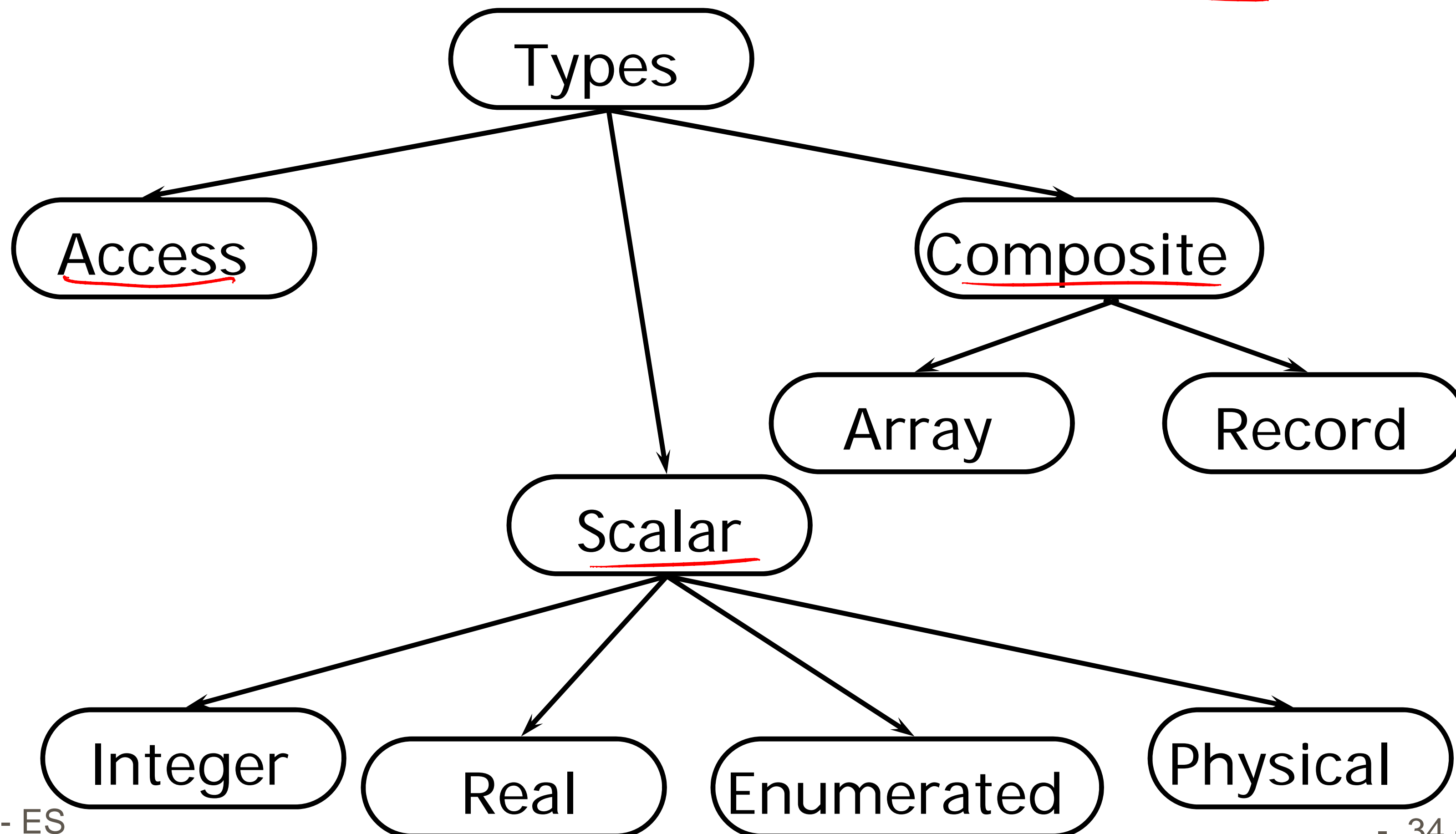
- VHDL provides two styles of describing component behavior
  - Data Flow: concurrent signal assignment statements
  - Behavioral: *processes* used to describe complex behavior by means of high-level language constructs
    - variables, loops, if-then-else statements, etc.

Module Outline

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

Data Types

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



VHDL Data Types
Scalar Types

- Integer
  - Minimum range for any implementation as defined by standard: - 2,147,483,647 to 2,147,483,647
  - 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;
```

VHDL Data Types
Scalar Types (Cont.)

- 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;
```

VHDL Data Types
Scalar Types (Cont.)

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

```
TYPE binary IS ( ON, OFF );
... some statements ...
ARCHITECTURE test_enum OF test IS
BEGIN
    PROCESS (X)
      VARIABLE a: binary;
    BEGIN
      a := ON;   -- OK
      ... more statements ...
      a := OFF;   -- OK
      ... more statements ...
    END PROCESS;
END test_enum;
```

VHDL Data Types
Scalar Types (Cont.)

- 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

VHDL Data Types
Composite Types

- Array
  - Used to group elements of the same type into a single VHDL object
  - Range may be unconstrained in declaration
    - Range would then be constrained when array is used
  - 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 Data Types

Composite Types (Cont.)

- Example one-dimensional array using DOWNTO :

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

15 element indices... 0

DOWNTO keyword must be used if leftmost index is greater than rightmost index

- ‘Big-endian’ bit ordering, for example

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

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

VHDL Data Types
Composite Types (Cont.)

- 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
```

VHDL Data Types
Access Type

- Access
  - Analogous to pointers in other languages
  - Allows for dynamic allocation of storage
  - Useful for implementing queues, fifos, etc.

VHDL Data Types
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 Data Types
Summary

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

VHDL 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
- Allow for easy update and readability
- Declaration of constant may omit value so that the value assignment may be deferred
  - Facilitates reconfiguration
- Declaration syntax :

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

- Declaration examples :

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

VHDL Objects
Variables

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

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

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

VHDL Objects
Signals

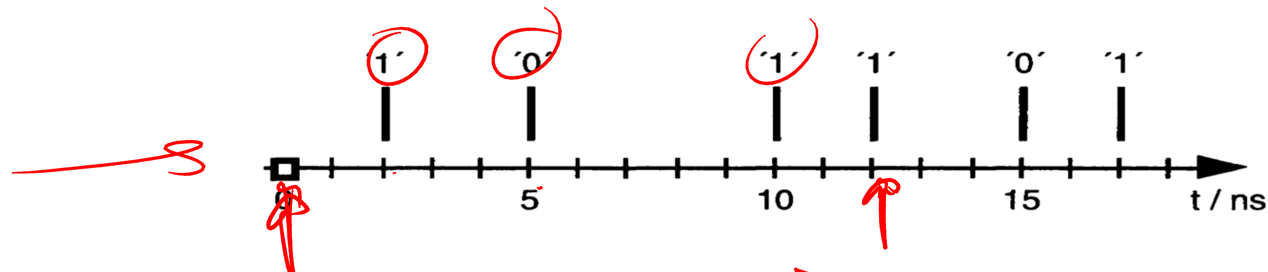
- Used for communication between VHDL components
- Real, physical signals in system often mapped to VHDL signals
- ALL VHDL signal assignments require either delta cycle or user-specified delay before new value is assumed
- Declaration syntax :

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

- Declaration and assignment examples :

```
SIGNAL sig_a : BIT;
```

```
sig_a <= '1' AFTER 2 ns,  '0' AFTER 5 ns,  '1' AFTER 10 ns,
         '1' AFTER 12 ns, '0' AFTER 15 ns, '1' AFTER 17 ns;
```



event queue for sig_a

Variables vs. Signal

Variables	Signals
Has no time dimension	Has a time dimension
Values are updated immediately	Values can be changed only at future times
Cannot schedule events for some future time	Can schedule events for some future time
Should be visible only within a process	Used for communication, hence has global visibility
Can never appear in the process sensitivity list	Only signals can appear in the sensitivity list

Signals and Variables

- This example highlights the difference between signals and variables

```
ARCHITECTURE test1 OF mux IS
    SIGNAL x : BIT := '1';
    SIGNAL y : BIT := '0';
BEGIN
    PROCESS (in_sig, x, y)
      BEGIN
        x <= in_sig XOR y;
        y <= in_sig XOR x;
      END PROCESS;
END test1;
```

```
ARCHITECTURE test2 OF mux IS
    SIGNAL y : BIT := '0';
BEGIN
    PROCESS (in_sig, y)
      VARIABLE x : BIT := '1';
    BEGIN
      x := in_sig XOR y;
      y <= in_sig XOR x;
    END PROCESS;
END test2;
```

- Assuming a 1 to 0 transition on *in_sig*, what are the resulting values for *y* in the both cases?

VHDL Objects
Signals vs Variables

- A key difference between variables and signals is the assignment delay

```
ARCHITECTURE sig_ex OF test IS
    SIGNAL a, b, c, out_1, out_2 : BIT;
BEGIN
    PROCESS (a, b, c, out_1)
    BEGIN
      out_1 <= a NAND b;
      out_2 <= out_1 XOR c;
    END PROCESS;
END sig_ex;
```

Time	a	b	c	out_1	out_2
0	0	1	1	1	0
1	1	1	1	1	0
1+d	1	1	1	0	0
1+2d	1	1	1	0	1

VHDL Objects

Signals vs Variables (Cont.)

```
ARCHITECTURE var_ex OF test IS
    SIGNAL a,b,c,out_4 : BIT;
BEGIN
    PROCESS (a, b, c)
    VARIABLE out_3 : BIT;
    BEGIN
      out_3 := a NAND b;
      out_4 <= out_3 XOR c;
    END PROCESS;
END var_ex;
```

Time	a	b	c	out_3	out_4
0	0	1	1	1	0
1	1	1	1	0	0
1+d	1	1	1	0	1

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
- The package STANDARD defines basic file IO routines for VHDL types
- The package TEXTIO defines more powerful routines handling IO of text files

Packages and Libraries

- 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
- Packages and libraries provide the ability to reuse constructs in multiple entities and architectures
  - Items declared in packages can be *used* (i.e. included) in other VHDL components

Packages

- Packages consist of two parts
  - Package declaration -- contains declarations of objects defined in the package
  - Package body -- contains necessary definitions for 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

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

Packages
Package Body

- The package body includes the necessary functional descriptions 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;
```

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 ...
```

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

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, e.g :
  - 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

Operators

- Operators can be chained to form complex expressions, e.g. :

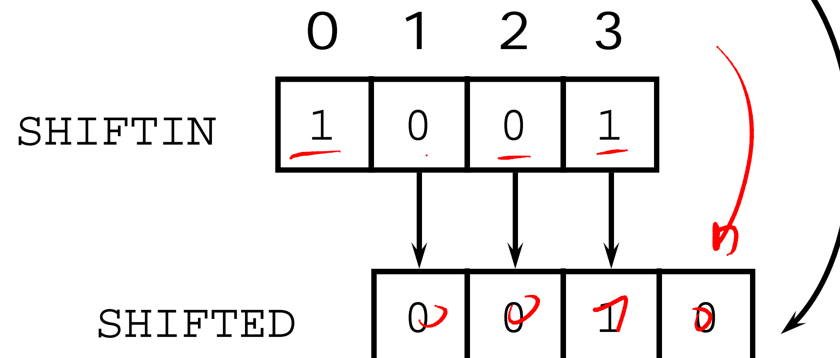
```
res <= a AND NOT(B) OR NOT(a) AND b;
```

  - Can use parentheses for readability and to control the association of operators and operands
- Defined precedence levels in decreasing order :
  - Miscellaneous operators -- **, abs, not
  - Multiplication operators -- *, /, mod, rem
  - Sign operator -- +, -
  - Addition operators -- +, -, &
  - Shift operators -- sll, srl, sla, sra, rol, ror
  - Relational operators -- =, /=, <, <=, >, >=
  - Logical operators -- AND, OR, NAND, NOR, XOR, XNOR

Operators
Examples

- The concatenation operator &

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



- The exponentiation operator **

```
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
```

Module Outline

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

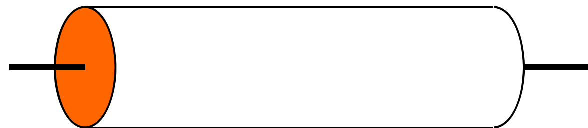
Summary

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

Delay models and VHDL semantics

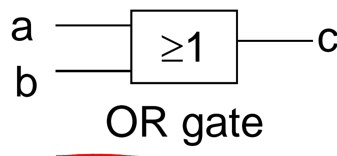
1. Transport delay

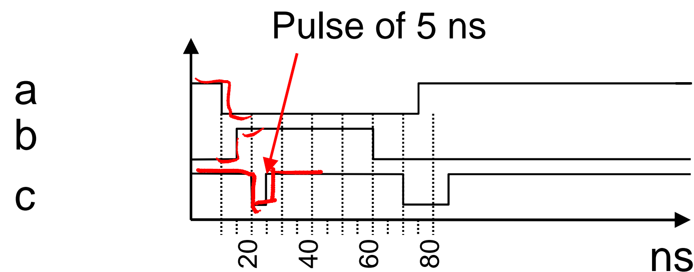
- *signal* <= **transport** expression **after** delay;
- This corresponds to models for *simple wires*



Pulses will be propagated, no matter how short they are - idealized wire.

c <= **transport** a or b **after** 10 ns;





1. Transport delay (2)

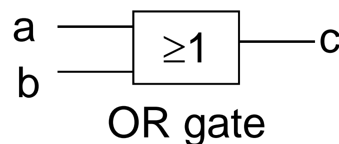
- “All old transactions that are projected to occur at or after the time at which the earliest of the new transactions is projected to occur are deleted from the projected output waveform” [VHDL LRM, chap. 8.4]

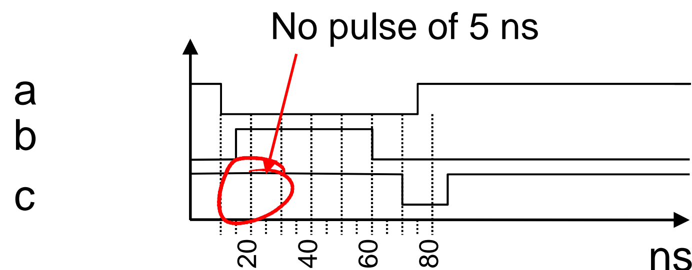
2. Inertial delay

- By default, inertial delay is assumed.
- Suppression of all "spikes" shorter than the delay, resp. shorter than the indicated suppression threshold.
- Inertial delay models the behavior of gates.

- Example:

  c <= a or b **after** 10 ns;





- Tricky rules for removing events from projected waveform (☞)