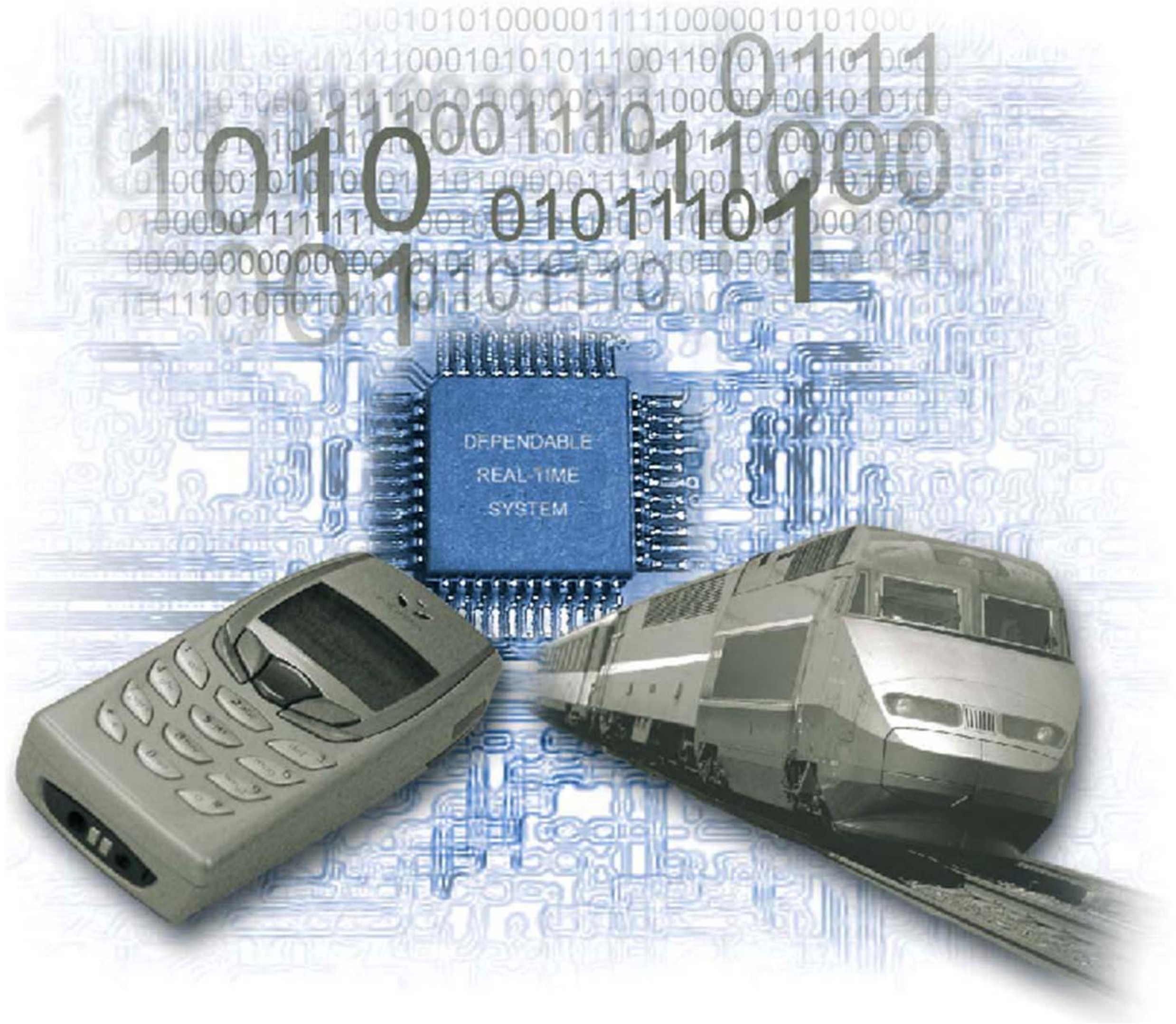
Embedded Systems



Thursday

- Midterm exam: December 16th, 2010, AudiMO,
  16:00 - 19:00

- No lecture on December 16th

- Open book: bring any handwritten or printed notes, or any books you like.

- Please bring your ID.

Exam Policy

- Midterm/End-of-Term Exam/End-of-Semester Exam

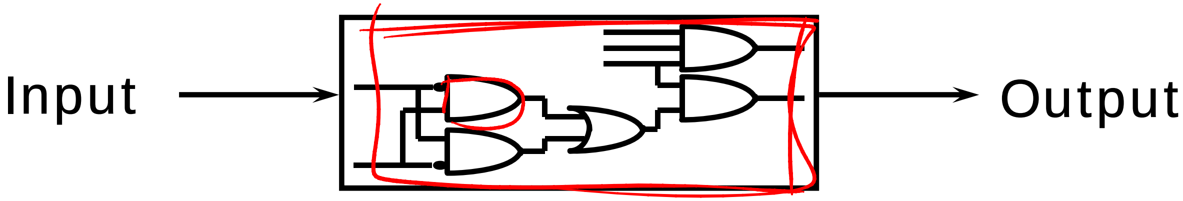
Requirement for admission to end-of-term and end-of-semester exams:

- > 50% of points in problem sets,
- > 50% of points in each project milestone, and
- > 50% of points in midterm exam

- **Final grade:**
- best grade in end-of-term or end-of-semester exam

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



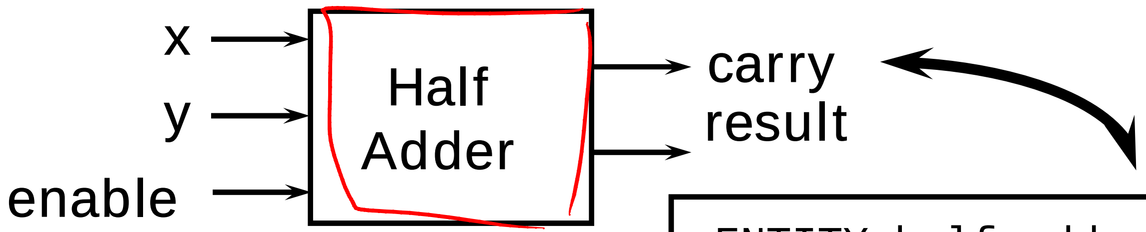
These gates can be pulled from a library of parts

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.

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 half_adder IS
    GENERIC(prop_delay : TIME := 10 ns);
    PORT( x, y, enable : IN BIT;
          carry, result : OUT BIT);
END half_adder;
```

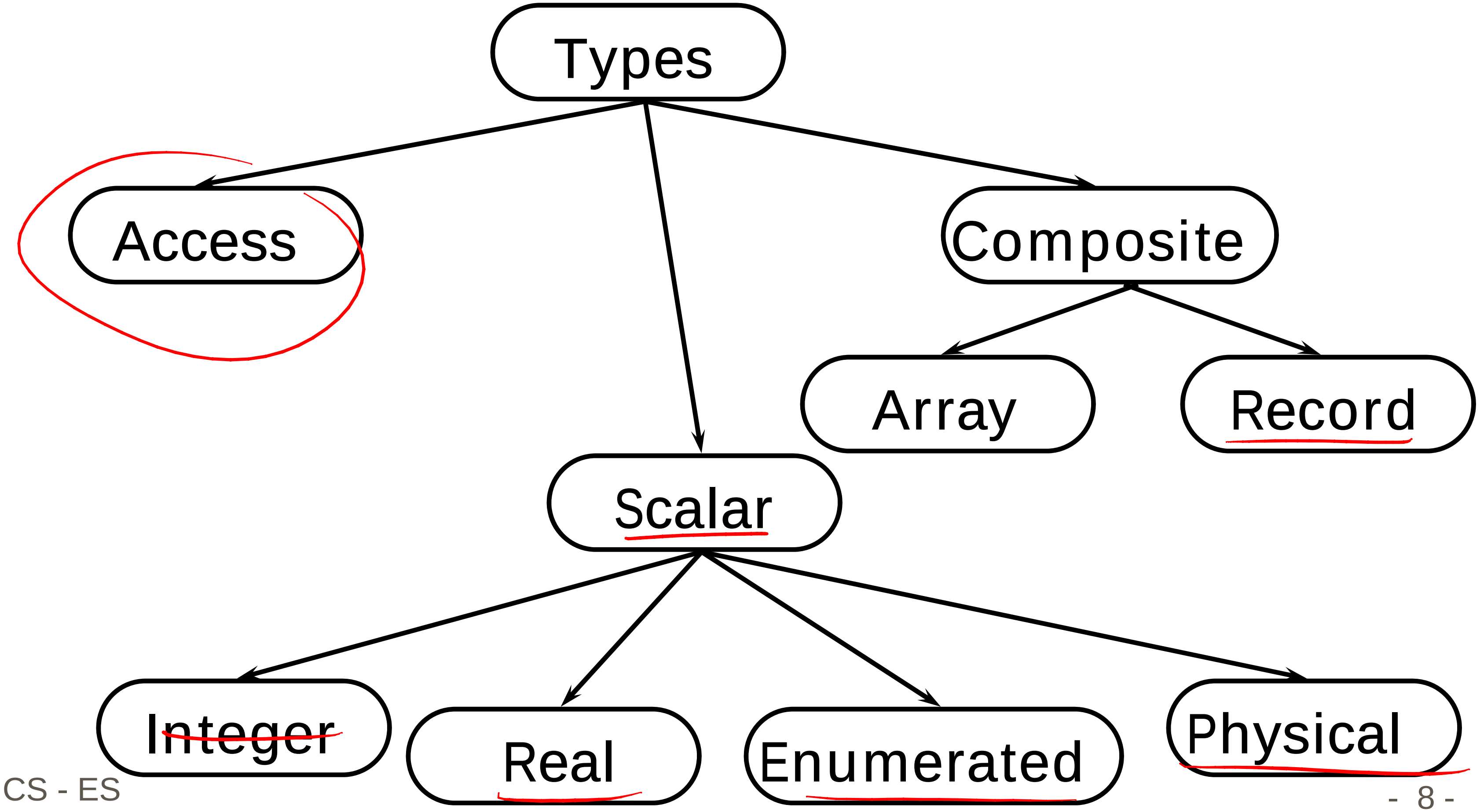
Architecture Bodies

- 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;
```

Data Types

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



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

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

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

REVIEW

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

REVIEW

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
- no VHDL entities, architectures

Libraries

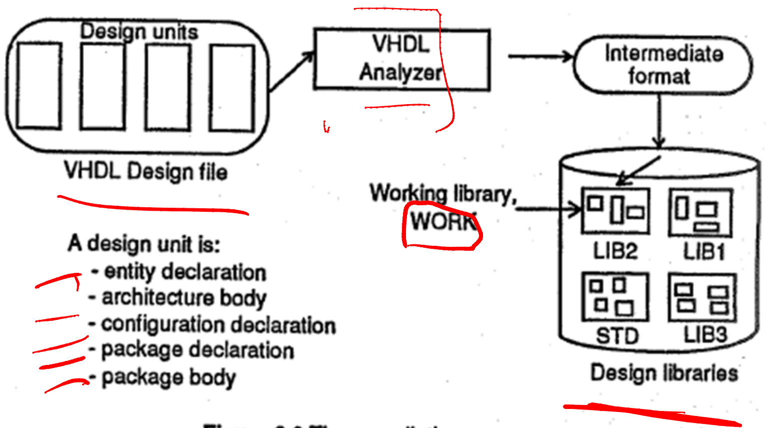


Figure 2.8 The compilation process.

- 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

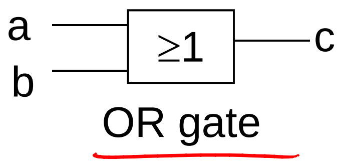
Delay models and VHDL semantics

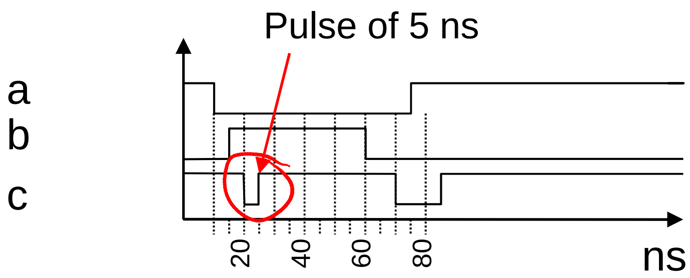
1. Transport delay

- *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) – LRM rule

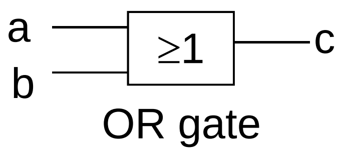
- "All old transactions that are projected to occur at or after the time at which the earliest of the new transactions is projected to occur are deleted from the projected output waveform" [VHDL LRM, chap. 8.4]

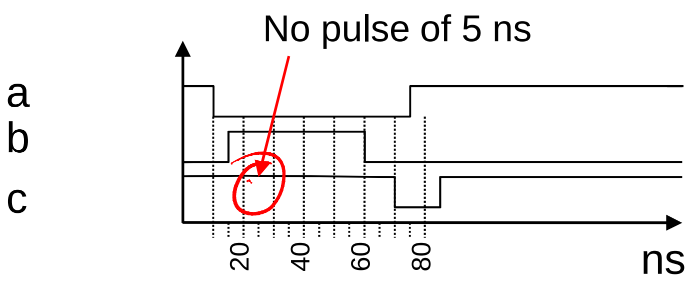
2. Inertial delay

- 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;





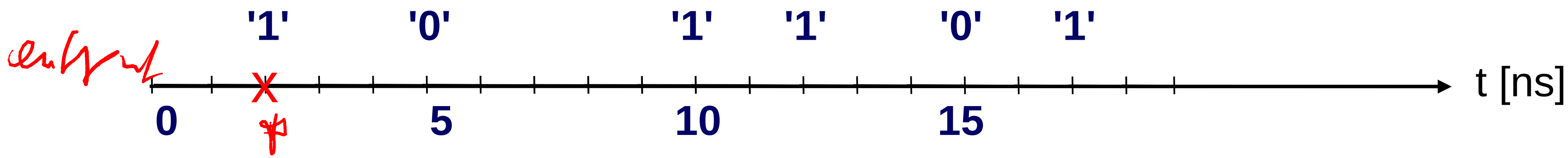
2. Inertial delay (2) – LRM rules

- "*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*"
- The new transactions are then appended
- "*All of the new transactions are marked*
- *An old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction;*
- *The transactions that determines the current value of the driver is marked;*
- *All unmarked transactions … are deleted from the projected output waveform*"

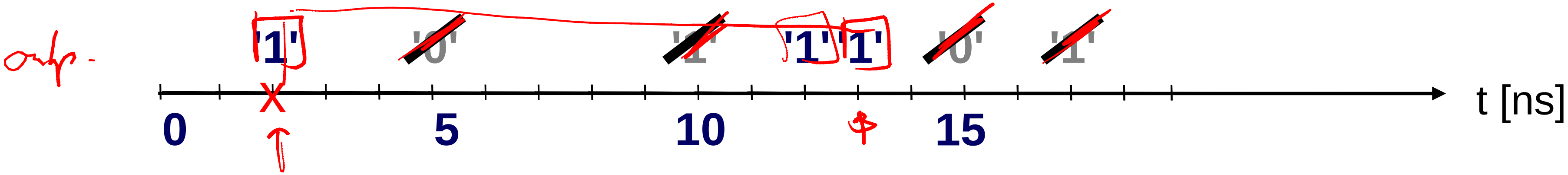
[VHDL LRM, chap. 8.4]

2. Inertial delay (3)

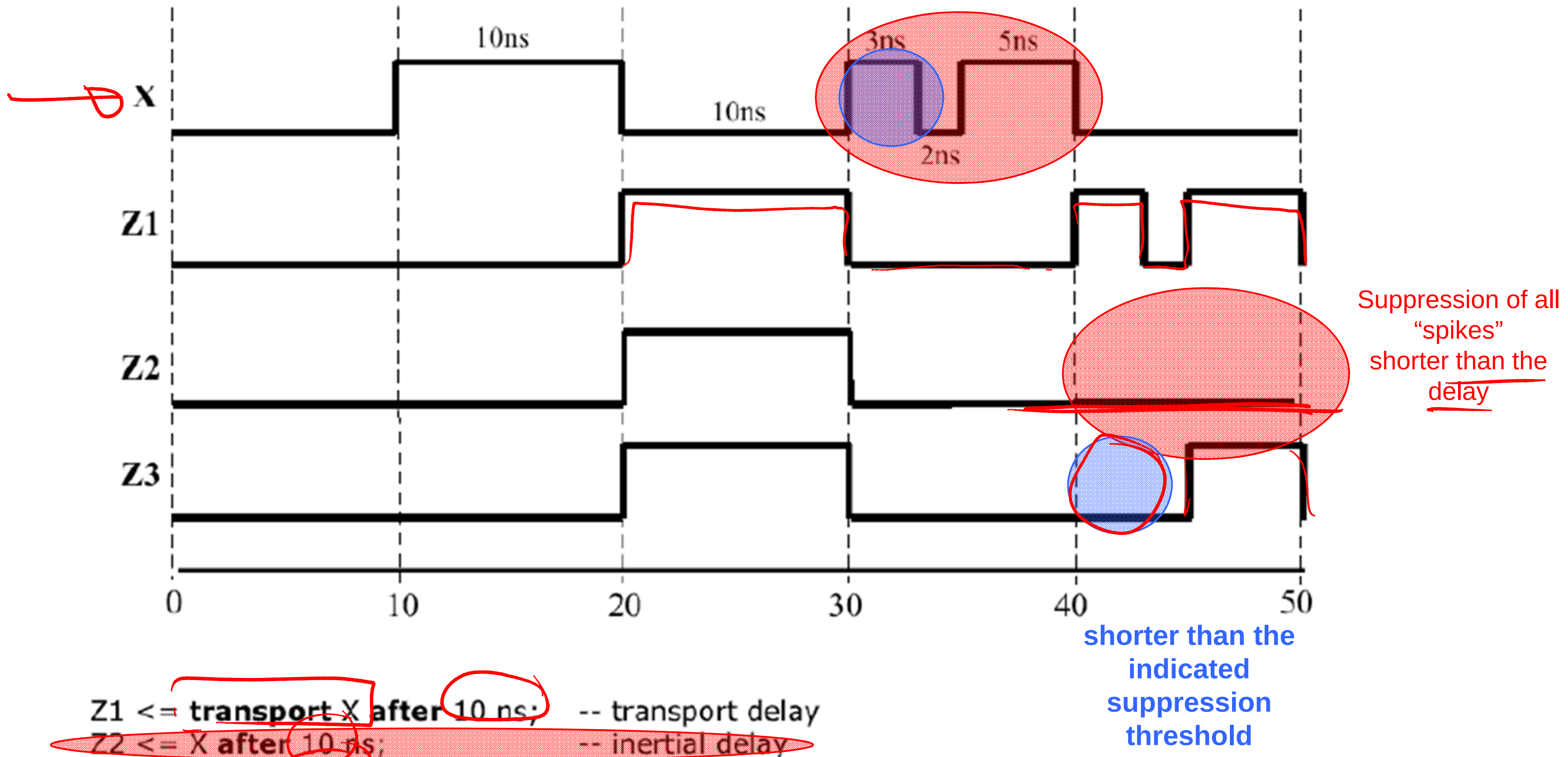
- Assume that we are executing a signal assignment
  output <= ‘1’ **after** 11 ns at time t=2 ns and the projected waveform is:



- Transactions to occur at or after 13 ns are deleted from the output waveform
- The new transactions are then appended
- All of the new transactions are marked
- Transactions immediately preceding a marked transaction and their value component is the same as that of the marked transaction;
- The transactions that determines the current value of the driver is marked;
- All unmarked transactions … are deleted from the projected output waveform

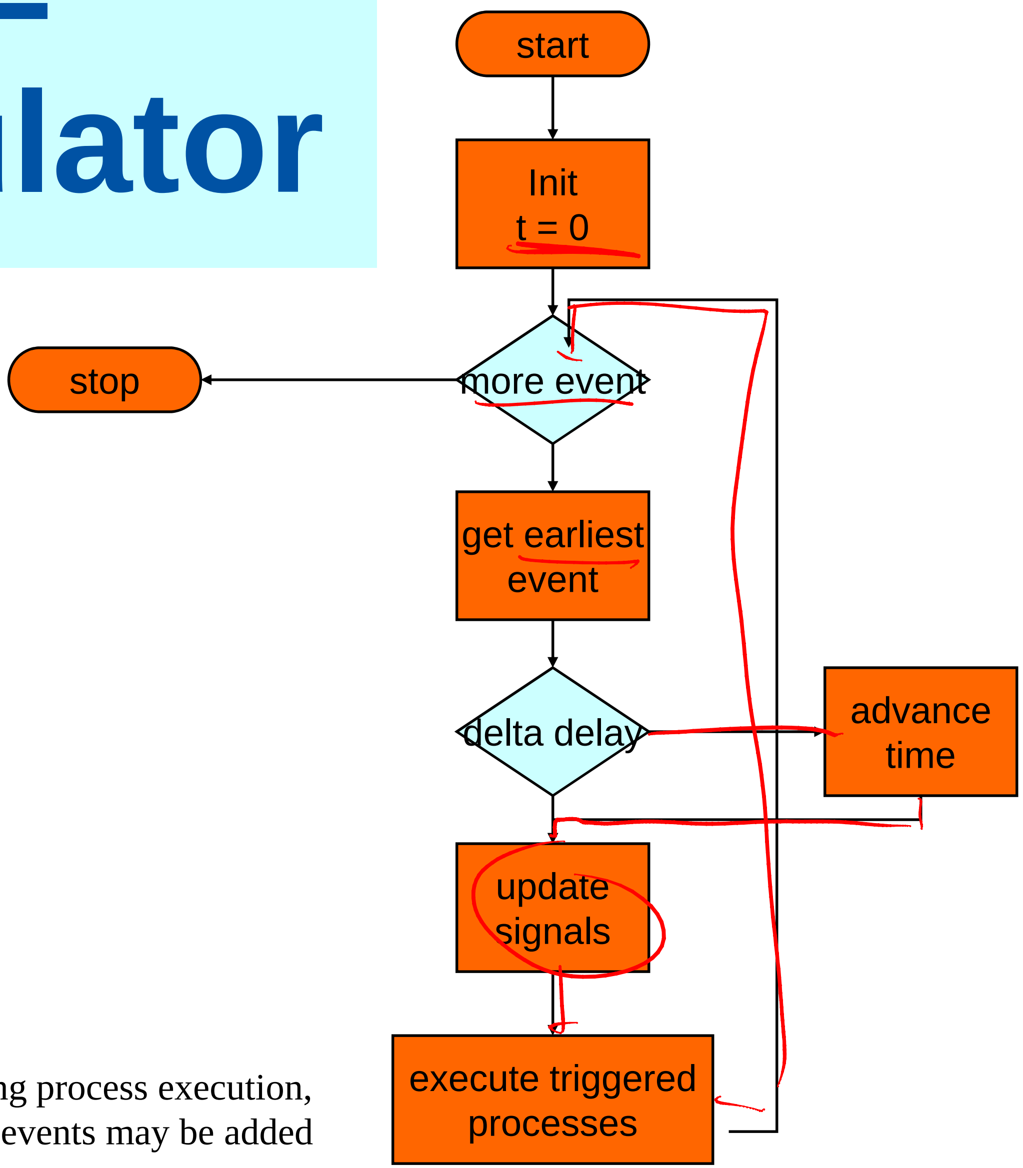


Example



Z1 <= **transport** X **after** 10 ns; -- transport delay
Z2 <= X **after** 10 ns; -- inertial delay
Z3 <= **reject** 4 ns X **after** 10 ns; -- delay with specified rejection pulse width

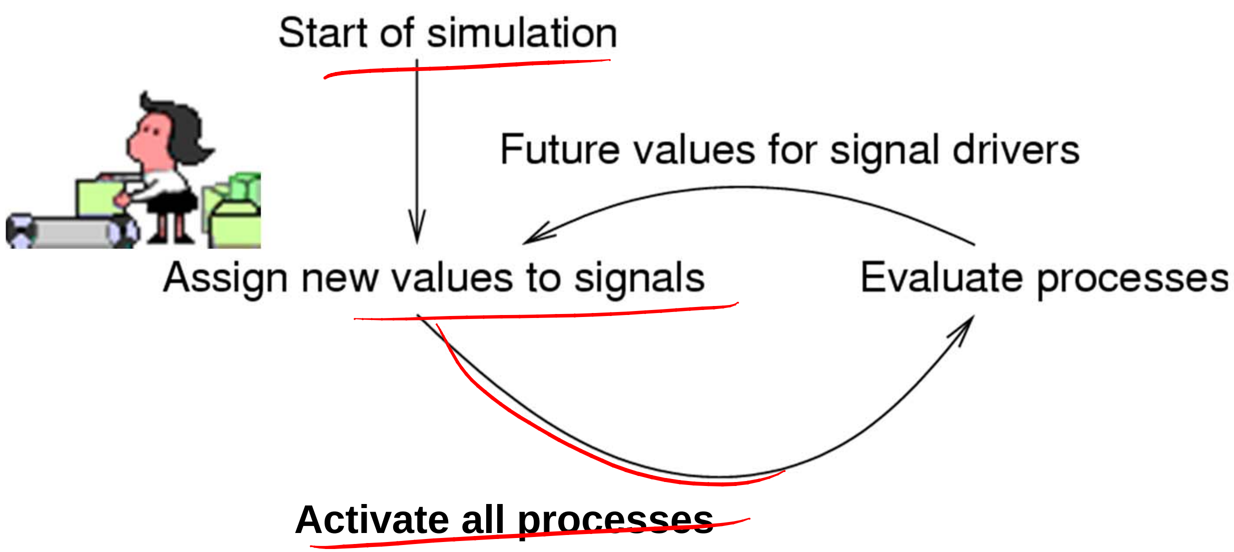
VHDL Simulator



during process execution,
new events may be added

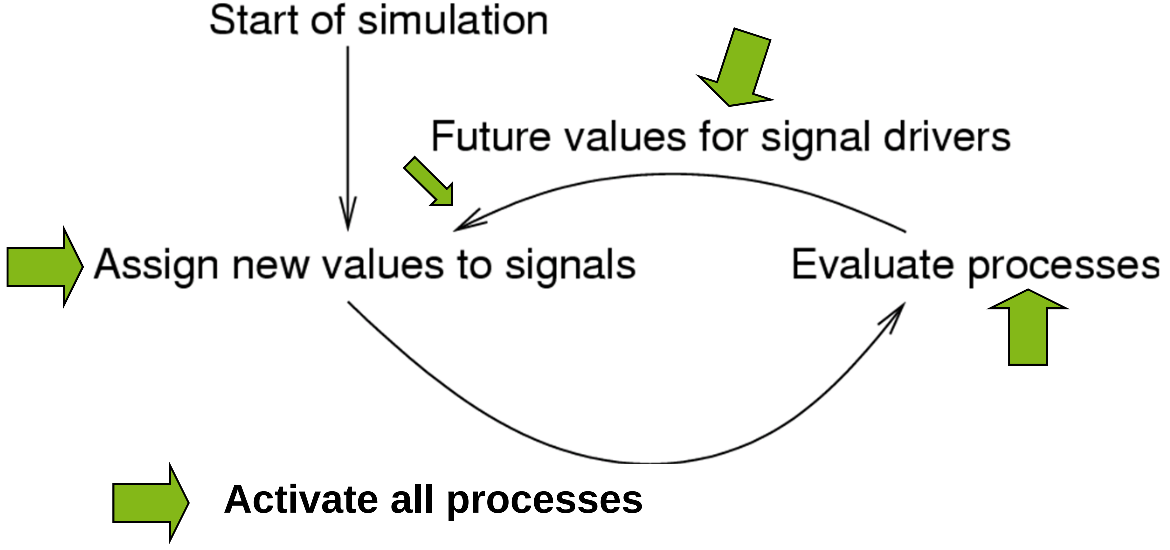
VHDL semantics: global control

- According to the original standards document:
- The execution of a model consists of an initialization phase followed by the repetitive execution of process statements in the description of that model.
- Initialization phase executes each process once.



VHDL semantics: initialization

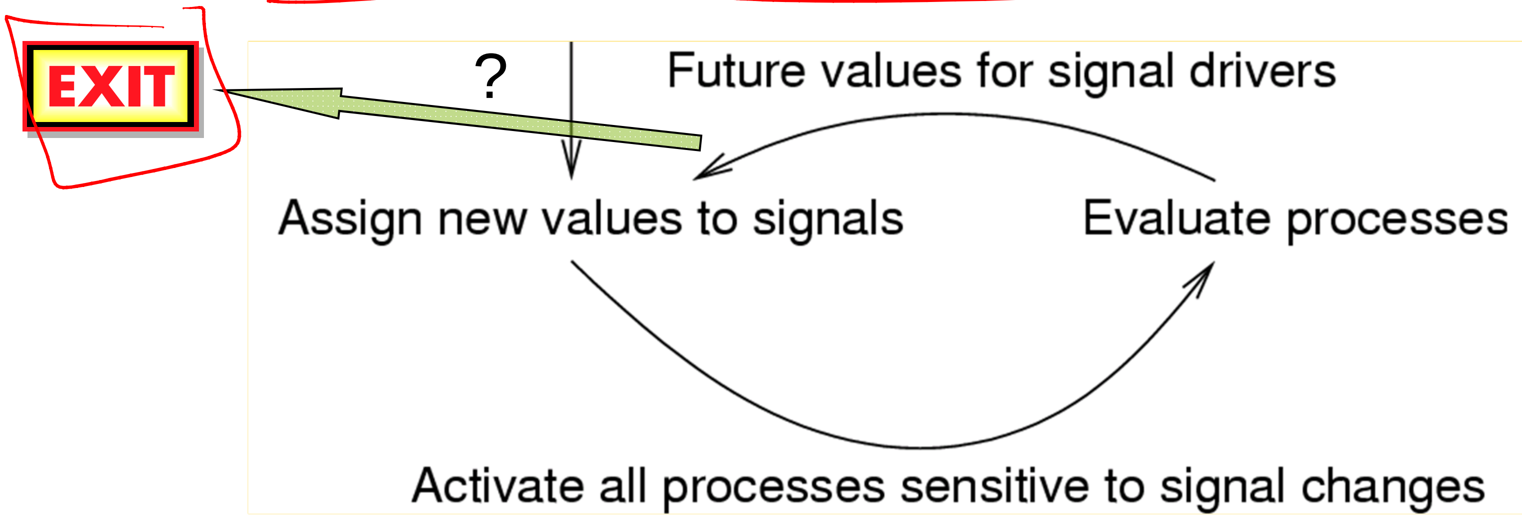
- At the beginning of initialization, the current time, T_c is 0 ns.
  - *The driving value and the effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. …*
  - *Each ... process … is executed until it suspends*.
  - *The time of the next simulation cycle (… in this case … the 1st cycle), T_n is calculated according to the rules of step f of the simulation cycle, below*.



VHDL semantics: The simulation cycle (1)

- Each simulation cycle starts with setting T_c to T_n. T_n was either computed during the initialization or during the last execution of the simulation cycle. Simulation terminates when the current time reaches its maximum, TIME'HIGH. According to the standard, the simulation cycle is as follows:

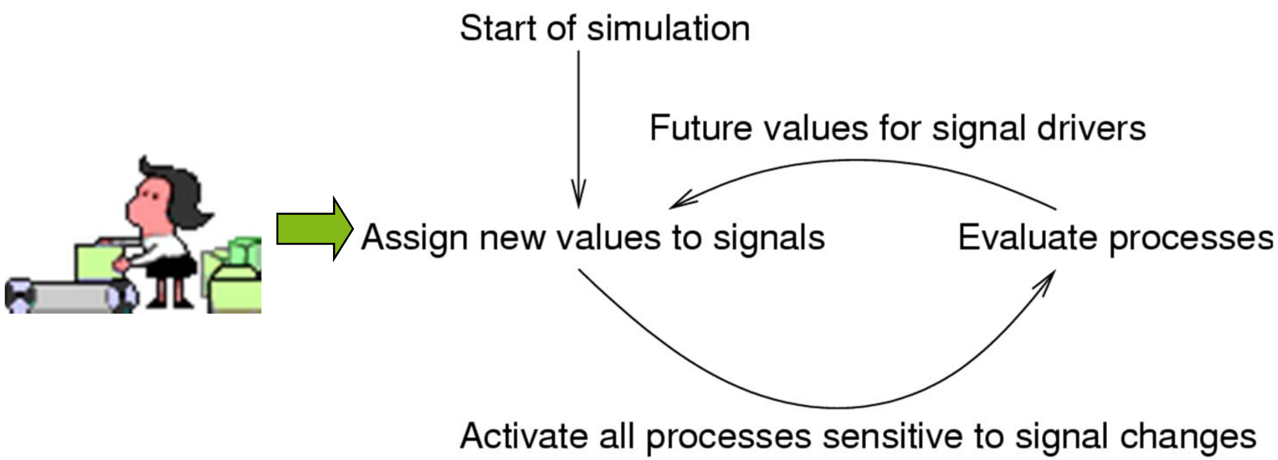
a) The current time, T_c is set to T_n. Stop if T_n= TIME'HIGH and not ∃ active drivers or process resumptions at T_n.



VHDL semantics: The simulation cycle (2)

b) ***Each*** ***active explicit signal in the model is updated. (Events may occur as a result.)***
Previously computed entries in the queue are now assigned if their time corresponds to the current time T_c.
New values of signals are not assigned before the next simulation cycle, at the earliest.
Signal value changes result in events ☞ enable the execution of processes that are sensitive to that signal.

c) ..

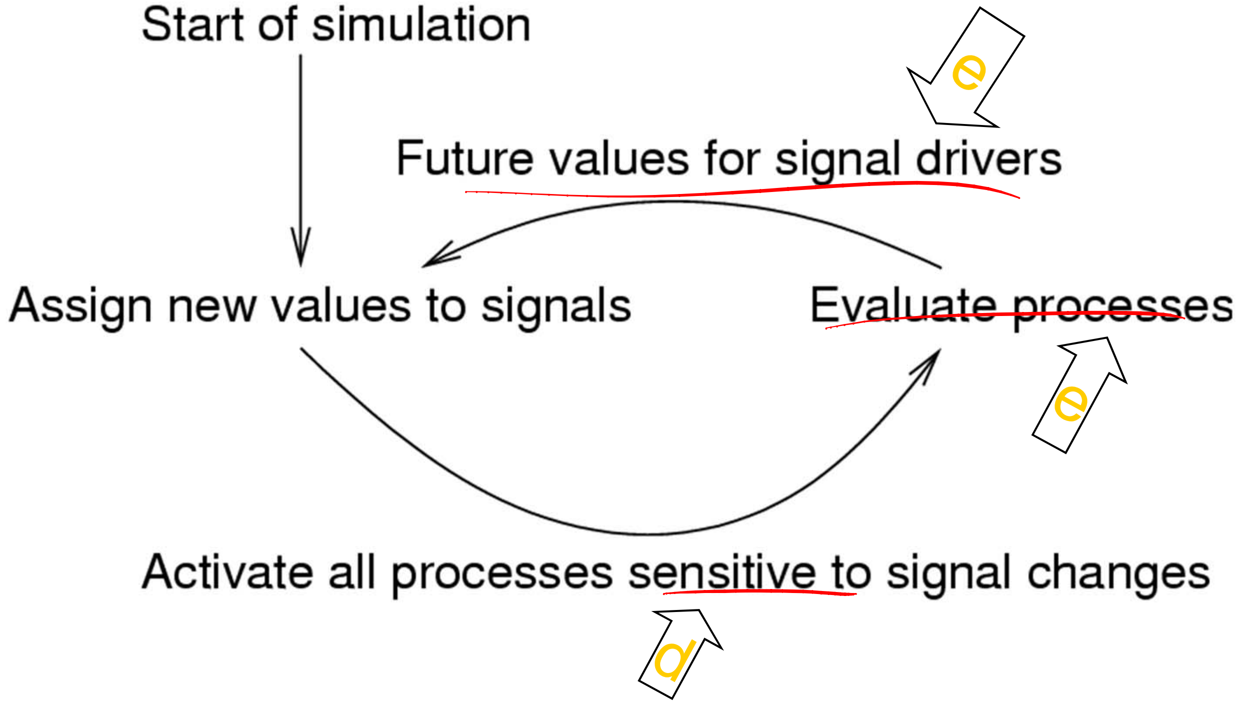


VHDL semantics: The simulation cycle (3)

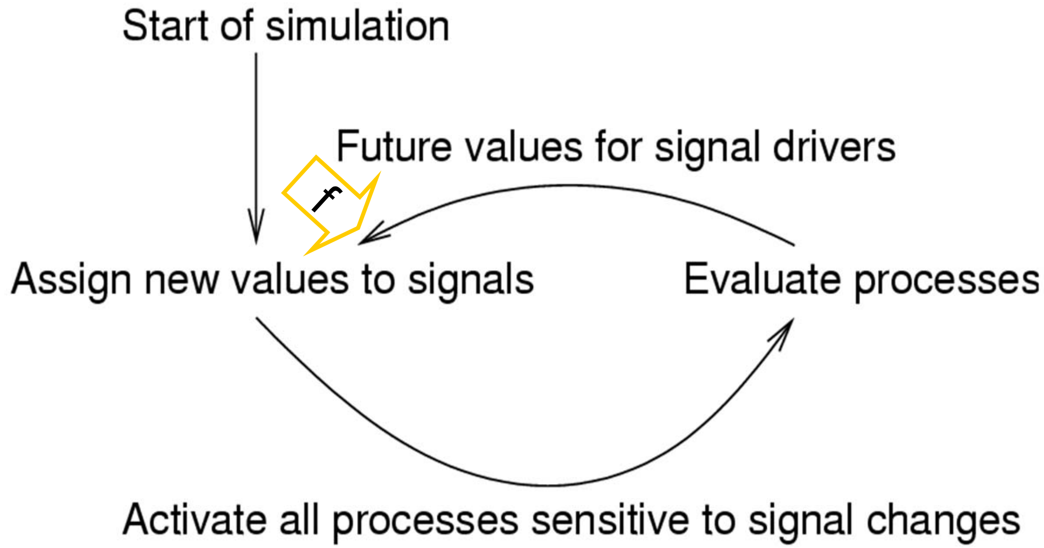
d) ∀ P sensitive to s: if event on s in current cycle: P resumes.

e) Each ... process that has resumed in the current simulation cycle is executed until it suspends*.

*Generates future values for signal drivers.



VHDL semantics: The simulation cycle (4)



f) Time T_n of the next simulation cycle = earliest of

1. TIME'HIGH (end of simulation time).
2. The next time at which a driver becomes active
3. The next time at which a process resumes (determined by **wait for** statements).

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 betwe
  the assignment delay

```
ARCHITECTURE sig_ex OF test
    SIGNAL a, b, c, out_1, (
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;
```

```
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_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

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

Behavioral vs. Structural

Modeling Interfaces

- *Entity* declaration reg4
  - describes the input/output *ports* of a module

entity name **port names** **port mode (direction)**

```
entity reg4 is
    port ( d0, d1, d2, d3, en, clk : in bit;
                q0, q1, q2, q3 : out bit );
end entity reg4;
```

punctuation

reserved words **port type**

Modeling Behavior

- *Architecture body*
  - describes an implementation of an entity
  - may be several per entity
- *Behavioral* architecture
  - describes the algorithm performed by the module
  - contains
    - *process statements*, each containing
    - *sequential statements*, including
    - *signal assignment statements* and
    - *wait statements*

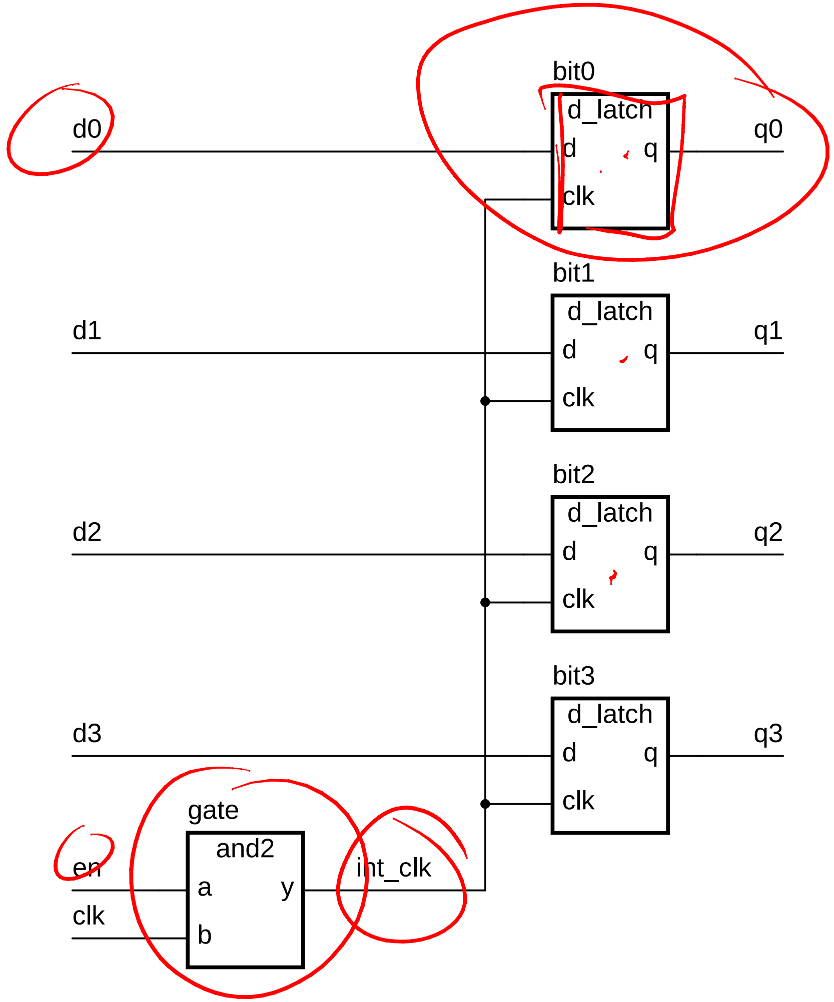
Behavior Example

```
architecture behav of reg4 is
begin
    storage : process is
        variable stored_d0, stored_d1, stored_d2, stored_d3 : bit;
    begin
        if en = '1' and clk = '1' then
            stored_d0 := d0;
            stored_d1 := d1;
            stored_d2 := d2;
            stored_d3 := d3;
        end if;
        q0 <= stored_d0 after 5 ns;
        q1 <= stored_d1 after 5 ns;
        q2 <= stored_d2 after 5 ns;
        q3 <= stored_d3 after 5 ns;
        wait on d0, d1, d2, d3, en, clk;
    end process storage;
end architecture behav;
```

Modeling 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
    - *wait statements*

Structure Example



Structure Example

- First declare D-latch and and-gate entities and architectures

```
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

- Now use them to implement a register

```
architecture struct of reg4 is
    signal int_clk : bit;
begin
    bit0 : entity work.d_latch(basic)
        port map ( d0, int_clk, q0 );
    bit1 : entity work.d_latch(basic)
        port map ( d1, int_clk, q1 );
    bit2 : entity work.d_latch(basic)
        port map ( d2, int_clk, q2 );
    bit3 : entity work.d_latch(basic)
        port map ( d3, int_clk, q3 );
    gate : entity work.and2(basic)
        port map ( en, clk, int_clk );
end architecture struct;
```

Mechanisms for Incorporating VHDL Design Objects

- VHDL mechanisms to incorporate design objects
  - Using direct instantiation (not available prior to VHDL-93) (slide 44)
  - Using *component* declarations and instantiations
    - Create idealized local *components* (i.e. declarations) and connect them to local signals (i.e. instantiations)
    - *Component* instantiations are then bound to VHDL design objects either :
      - Locally -- within the architecture declaring the component
      - At higher levels of design hierarchy, via *configurations*
- Consider structural descriptions for the following entity :

```
USE work.resources.all;

ENTITY reg4 IS -- 4-bit register with no enable
  GENERIC(tprop : delay := 8 ns;
          tsu   : delay := 2 ns);
  PORT(d0,d1,d2,d3 : IN level;
       clk    : IN level;
       q0,q1,q2,q3 : OUT level);
END reg4;
```

4-Bit Register as Running Example

- First, need to find the building block(s)

```
USE work.resources.all;

ENTITY dff IS

  GENERIC(tprop : delay := 8 ns;
          tsu   : delay := 2 ns);

  PORT(d      : IN level;
       clk    : IN level;
       enable : IN level;
       q      : OUT level;
       qn     : OUT level);

END dff;
```

```
ARCHITECTURE behav OF dff IS
  BEGIN
    one : PROCESS (clk)
      BEGIN
                                        -- rising clock edge
        IF ((clk = '1' AND clk'LAST_VALUE = '0')
             AND enable = '1') THEN     -- ff enabled

                                        -- check setup
          IF (d'STABLE(tsu)) THEN
                                        -- check valid input data
            IF (d = '0') THEN
              q <= '0' AFTER tprop;
              qn <= '1' AFTER tprop;
            ELSIF (d = '1') THEN
              q <= '1' AFTER tprop;
              qn <= '0' AFTER tprop;
            ELSE                        -- else invalid data
              q <= 'X';
              qn <= 'X';
            END IF;
          ELSE                          -- else no setup
            q <= 'X';
            qn <= 'X';
          END IF;
        END IF;
    END PROCESS one;
END behav;
```

General Steps to Incorporate VHDL Design Objects

- A VHDL design object to be incorporated into an architecture must *generally* be :
  - declared -- where a local interface is defined
  - instantiated -- where local signals are connected to the local interface
    - Regular structures can be created easily using *GENERATE* statements in component instantiations
  - bound -- where an entity/architecture object which implements it is selected for the instantiated object

Using Component Declarations and Local Bindings

- Component declaration defines interface for idealized local object
  - Component declarations may be placed in architecture declarations or in package declarations
- Component instantiation connects local signals to component interface signals

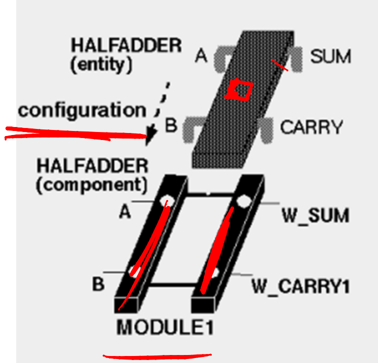
```
USE work.resources.all;

ARCHITECTURE struct_2  OF reg4 IS
  COMPONENT reg1 IS
    PORT (d, clk : IN level;
          q : OUT level);
  END COMPONENT reg1;
  CONSTANT enabled : level := '1';
  FOR ALL : reg1 USE work.dff(behav)
     PORT
   MAP(d=>d,clk=>clk,enable=>enabled,q=>q,qn=>OPEN);
 BEGIN
    r0 : reg1 PORT MAP (d=>d0,clk=>clk,q=>q0);
    r1 : reg1 PORT MAP (d=>d1,clk=>clk,q=>q1);
    r2 : reg1 PORT MAP (d=>d2,clk=>clk,q=>q2);
    r3 : reg1 PORT MAP (d=>d3,clk=>clk,q=>q3);
END struct_2;
```

declared -- where a local interface is defined

instantiated -- where local signals are connected to the local interface

bound -- where an entity/architecture object which implements it is selected for the instantiated object

Using Component Declarations and Configurations




```
USE work.resources.all;

ARCHITECTURE struct_3 OF reg4 IS
  COMPONENT reg1 IS
    PORT (d, clk : IN level;
          q : OUT level);
  END COMPONENT reg1;
  CONSTANT enabled : level := '1';
 BEGIN
    r0 : reg1 PORT MAP (d<=d0,clk<=clk,q<=q0);
    r1 : reg1 PORT MAP (d<=d1,clk<=clk,q<=q1);
    r2 : reg1 PORT MAP (d<=d2,clk<=clk,q<=q2);
    r3 : reg1 PORT MAP (d<=d3,clk<=clk,q<=q3);
END struct_3;
```

```
USE work.resources.all;

CONFIGURATION reg4_conf_1 OF reg4 IS
  CONSTANT enabled : level := '1';
  FOR struct_3
    FOR all : reg1 USE work.dff(behav)
       PORT MAP(d=>d,clk=>clk,enable=>enabled,q=>q,qn=>OPEN);
    END FOR;
  END FOR;
END reg4_conf_1;
```

```
--   Architecture in which a COMPONENT for reg4 is declared
                                ...
   FOR ALL : reg4_comp USE CONFIGURATION work.reg4_conf_1;
                                ...
```

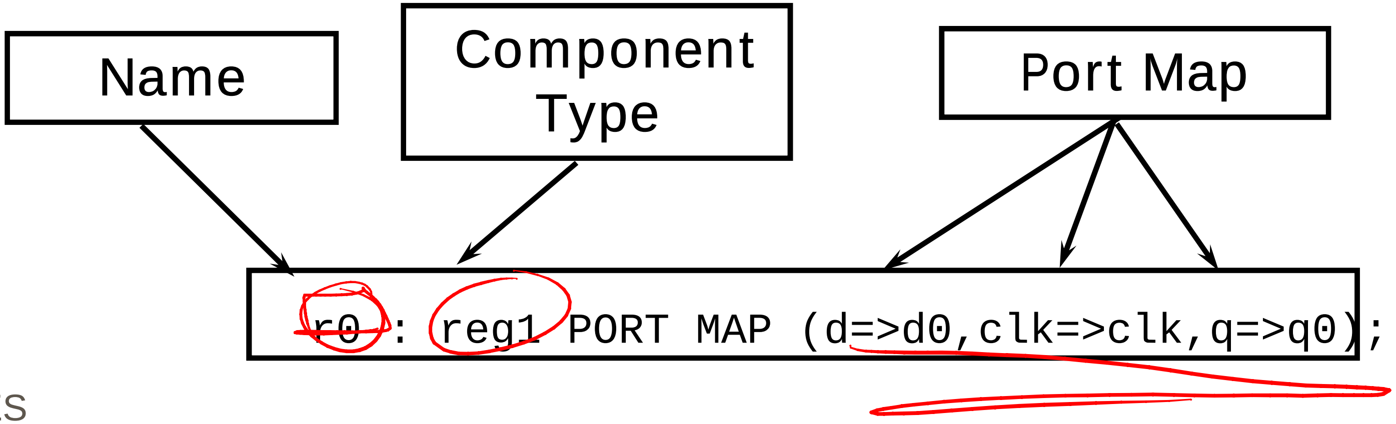
- three separate VHDL files
- first file above shows the architecture description in which the *reg1* component is declared and instantiated.
- second file shows a configuration declaration in which the *reg1* components in the *struct_3* architecture of entity *reg4* are bound to dff(behav).
- The third example shows a small excerpt from an architecture description in which a locally visible component named *reg4_comp* is bound to a VHDL design object via the configuration declaration *reg4_conf_1* found in the *work* library (i.e. this is configuration declaration shown in the middle section of this slide).

Power of Configuration Declarations

- Reasons to use *configuration declarations* :
  - Large design may span multiple levels of hierarchy
  - When the architecture is developed, only the component interface may be available
  - Mechanism to put the pieces of the design together
- Configurations can be used to customize the use VHDL design objects interfaces as needed :
  - Entity name can be different than the component name
  - Entity of incorporated design object may have more ports than the component declaration
  - Ports on the entity declaration of the incorporated design object may have different names than the component declaration

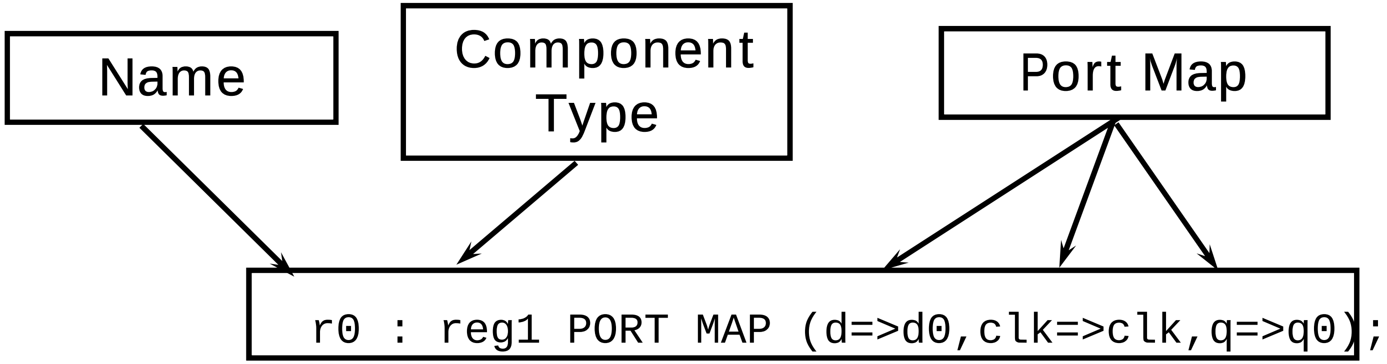
Instantiation Statement

- The instantiation statement connects a declared component to signals in the architecture
- The instantiation has 3 key parts
  - Name -- to identify unique *instance* of component
  - Component type -- to select one of the declared components
  - Port map -- to connect to signals in architecture
    - Along with optional Generic Map presented on next slide



Instantiation Statement

- The instantiation statement connects a declared component to signals in the architecture
- The instantiation has 3 key parts
  - Name -- to identify unique *instance* of component
  - Component type -- to select one of the declared components
  - Port map -- to connect to signals in architecture
    - Along with optional Generic Map presented on next slide



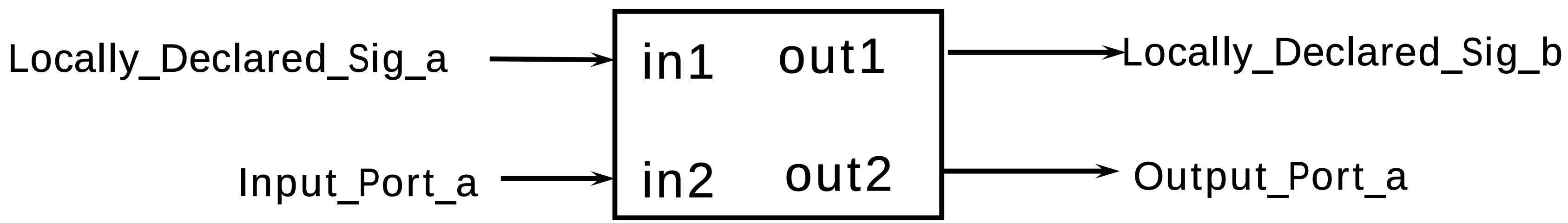
Generic Map

- Generics allow the component to be customized upon instantiation
  - Entity declaration of design object being incorporated provides default values
- The GENERIC MAP is similar to the PORT MAP in that it maps specific values to the generics of the component

```
USE Work.my_stuff.ALL
ARCHITECTURE test OF test_entity
      SIGNAL S1, S2, S3 : BIT;
BEGIN
    Gate1 : my_stuff.and_gate -- component found in package
      GENERIC MAP (tplh=>2 ns, tphl=>3 ns)
      PORT MAP (S1, S2, S3);
END test;
```

Rules for Actuals and Locals

- An *actual* is either signal declared within the architecture or a port in the entity declaration
  - A port on a *component* is known as a *local* and must be matched with a compatible *actual*
- VHDL has two main restrictions on the association of *locals* with *actuals*
  - Local and actual must be of same data type
  - Local and actual must be of compatible modes
    - Locally declared signals do not have an associated mode and can connect to a local port of any mode



Generate Statement

- VHDL provides the GENERATE statement to create well-patterned structures easily
  - Some structures in digital hardware are repetitive in nature (e.g. RAMs, adders)
- Any VHDL concurrent statement may be included in a GENERATE statement, including another GENERATE statement
  - Specifically, component instantiations may be made within GENERATE bodies

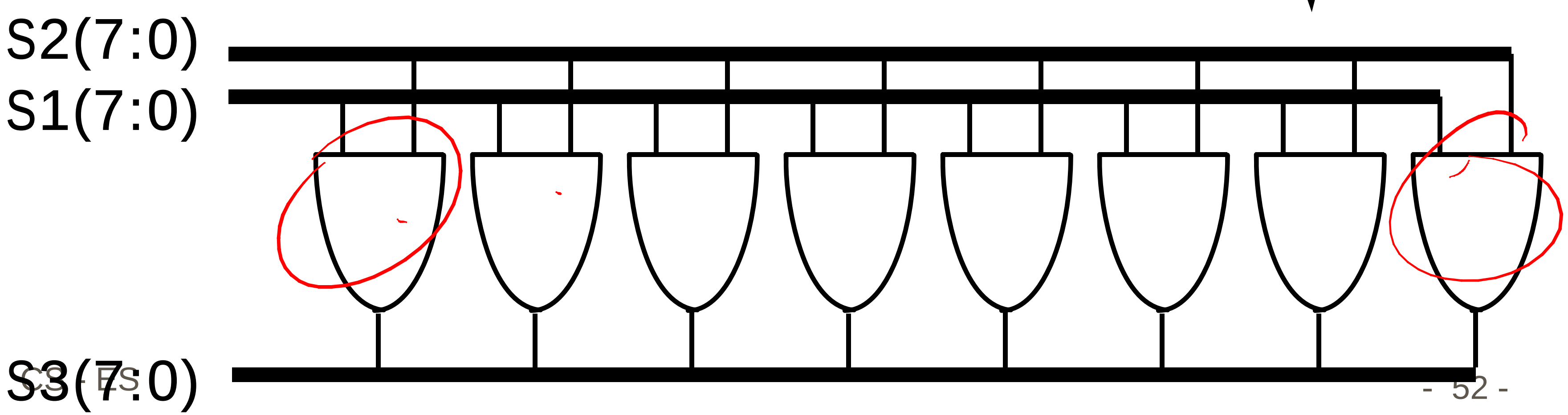
Generate Statement
FOR-scheme

- All objects created are similar
- The GENERATE parameter must be discrete and is undefined outside the GENERATE statement
- Loop cannot be terminated early

```
name : FOR N IN 1 TO 8 GENERATE
    concurrent-statements
END GENERATE name;
```

FOR-scheme Example

```
-- this uses the and_gate component from before
ARCHITECTURE test_generate OF test_entity IS
      SIGNAL S1, S2, S3: BIT_VECTOR(7 DOWNTO 0);
BEGIN
    G1 : FOR N IN 7 DOWNTO 0 GENERATE
      and_array : and_gate
        GENERIC MAP (2 ns, 3 ns)
        PORT MAP (S1(N), S2(N), S3(N));
    END GENERATE G1;
END test_generate;
```



Generate Statement
IF-scheme

- Allows for conditional creation of components
- Can not use ELSE or ELSIF clauses with the IF-scheme

```
name : IF (boolean expression) GENERATE
   concurrent-statements
END GENERATE name;
```

IF-scheme Example

```
ARCHITECTURE test_generate OF test_entity
      SIGNAL S1, S2, S3: BIT_VECTOR(7 DOWNTO 0);
BEGIN
    G1 : FOR N IN 7 DOWNTO 0 GENERATE

      G2 : IF (N = 7) GENERATE
        or1 : or_gate
          GENERIC MAP (3 ns, 3 ns)
          PORT MAP (S1(N), S2(N), S3(N));
      END GENERATE G2;

      G3 : IF (N < 7) GENERATE
        and_array : and_gate
          GENERIC MAP (2 ns, 3 ns)
          PORT MAP (S1(N), S2(N), S3(N));
      END GENERATE G3;

    END GENERATE G1;
END test_generate;
```

Summary

- Structural VHDL describes the arrangement and interconnection of components
- Components can be of any level of abstraction -- low level gates or high level blocks of logic
- Generics are inherited by every architecture or component of that entity
- Generate statements automatically create large, regular blocks of logic

Behavioral VHDL

Process Syntax

```
[ process_label : ] PROCESS
[( sensitivity_list )]

  process_declarations

BEGIN

  process_statements

END PROCESS [ process_label ] ;
```

NO
SIGNAL
DECLARATIONS!

VHDL Sequential Statements

- Assignments executed sequentially in processes
- Sequential statements
  - {Signal, variable} assignments
  - Flow control
    - IF <condition> THEN <statements> [ELSIF] <statements> ELSE <statements> END IF;
    - FOR <range> LOOP <statements> END LOOP;
    - WHILE <condition> LOOP <statements> END LOOP;
    - CASE <condition> IS WHEN <value> => <statements>
WHEN <value> => <statements>
WHEN others => <statements>
END CASE;
  - WAIT ON <signal> UNTIL <expression> FOR <time> ;
  - ASSERT <condition> REPORT <string> SEVERITY <level> ;

The Wait Statement

- The wait statement causes the suspension of a process statement or a procedure
- wait [sensitivity_clause] [condition_clause] [timeout_clause] ;
  - sensitivity_clause ::= ON signal_name { , signal_name }

```
WAIT ON clock;
```

  - condition_clause ::= UNTIL boolean_expression

```
WAIT UNTIL clock = '1';
```

  - timeout_clause ::= FOR time_expression

```
WAIT FOR 150 ns;
```

Equivalent Processes

- “Sensitivity List” vs “wait on”

```
Summation:
  PROCESS( A, B, Cin)
  BEGIN
    Sum <= A XOR B XOR Cin;
END PROCESS Summation;
```

=

```
Summation:  PROCESS
  BEGIN
    Sum <= A XOR B XOR Cin;
      WAIT ON A, B, Cin;
END PROCESS Summation;
```

if you put a sensitivity list in a process,
you can’t have a wait statement!

if you put a wait statement in a process,
you can’t have a sensitivity list!

“wait until” and “wait for”

- What do these do?

```
Summation:  PROCESS
  BEGIN
    Sum <= A XOR B XOR Cin;
    WAIT UNTIL A = ‘1’;
END PROCESS Summation;
```

```
Summation:  PROCESS
  BEGIN
    Sum <= A XOR B XOR Cin;
  WAIT FOR 100 ns;
END PROCESS Summation;
```

Subprograms

- Similar to subprograms found in other languages
- Allow repeatedly used code to be referenced many times without duplication
- Break down large chunks of code in small, more manageable parts
- VHDL provides functions and procedures for use

Functions

- Produce a single return value
- Called by expressions
- Cannot modify the parameters passed to it
- Requires a RETURN statement

```
FUNCTION add_bits (a, b : IN BIT) RETURN BIT IS
BEGIN  -- functions cannot return multiple values
      RETURN (a XOR b);
END add_bits;
```

```
FUNCTION add_bits2 (a, b : IN BIT) RETURN BIT IS
 VARIABLE result : BIT;  -- variable is local to function
   BEGIN
     result := (a XOR b);
     RETURN result;  -- the two functions are equivalent
END add_bits2;
```

Functions

```
ARCHITECTURE behavior OF adder IS
BEGIN
    PROCESS (enable, x, y)
    BEGIN
    IF (enable = '1') THEN
      result <= add_bits(x, y);
      carry  <= x AND y;
    ELSE
      carry, result <= '0';
    END PROCESS;
END behavior;
```

```
FUNCTION add_bits

(a, b : IN BIT)
```

- Functions must be called by other statements
- Parameters use positional association

Procedures

- Produce many output values
- Are invoked by statements
- May modify the parameters

```
PROCEDURE add_bits3 (SIGNAL a, b, en : IN BIT;
           SIGNAL temp_result, temp_carry : OUT BIT) IS

BEGIN -- procedures can return multiple values
     temp_result <= (a XOR b) AND en;
     temp_carry <= a AND b AND en;

END add_bits3;
```

- Do not require a RETURN statement

Procedures (Cont.)

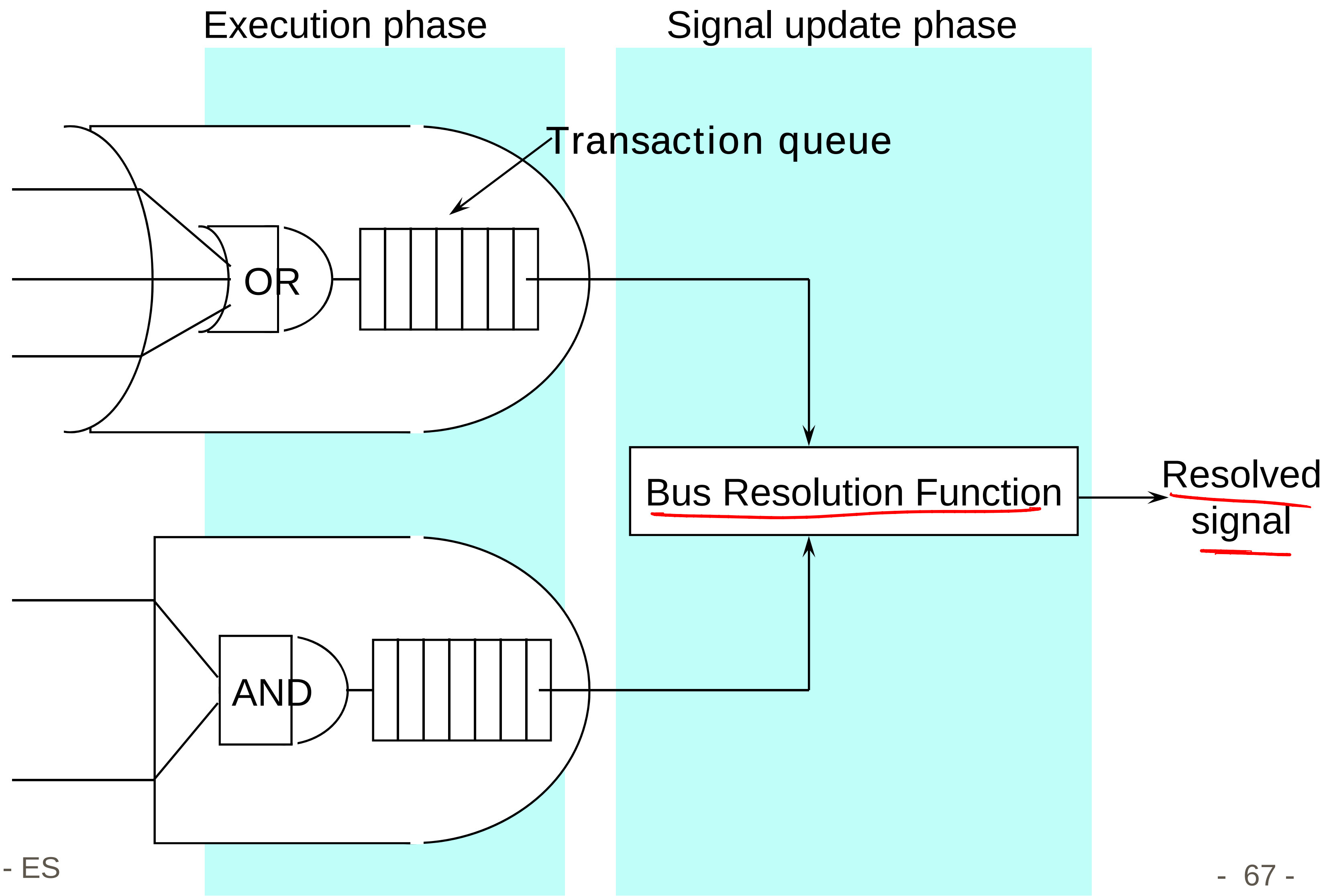
```
ARCHITECTURE behavior OF adder IS
              BEGIN
       PROCESS (enable, x, y)
             BEGIN
   add_bits3(x, y, enable,
                result, carry);
            END PROCESS;
          END behavior;
```

- With parameter passing, it is possible to further simplify the architecture

```
PROCEDURE add_bits3

(SIGNAL a, b, en : IN BIT;
 SIGNAL temp_result,
      temp_carry : OUT BIT)
```

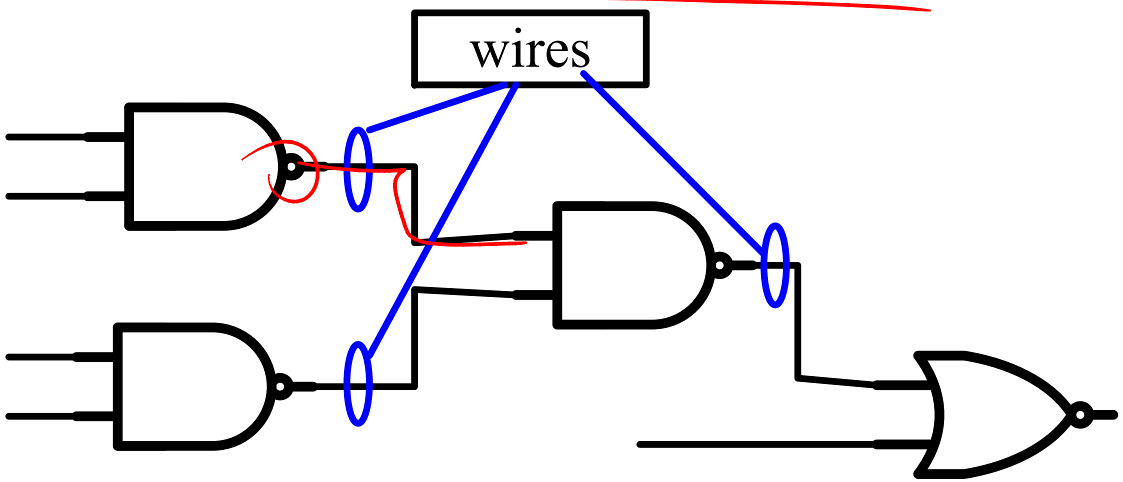
- The parameters must be compatible in terms of data flow and data type

Signal Resolution and Buses



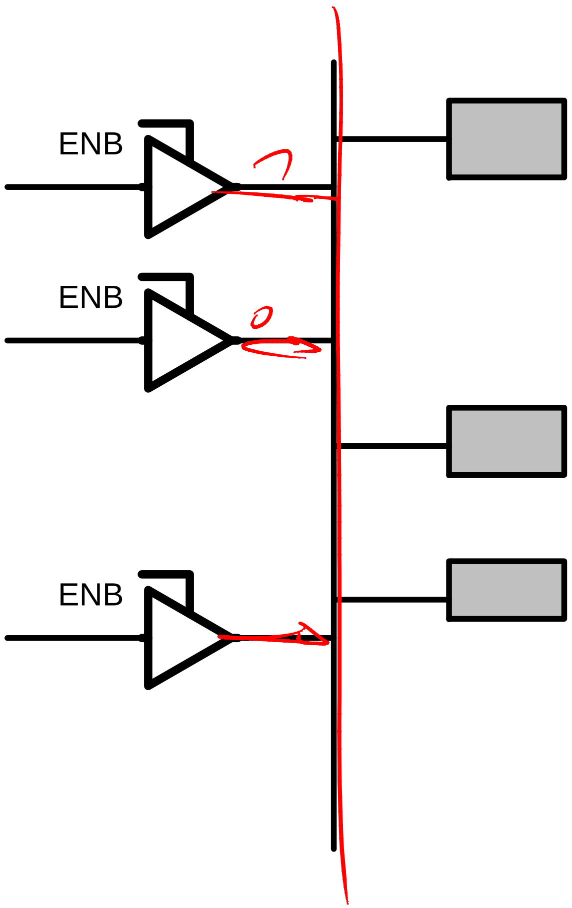
Busses and Wires

- What is the difference between a bus and a wire?
- <u>Wires</u> – have only one driving source



Busses and Wires

- Busses on the other hand can be driven by one or more sources
- In both cases there can be more than one destination for the signal
- With busses, only the device acting as source will actually drive a value. All others will have their output set at high impedance (Z).



Information on a Bus

- Possible state for a BUS
  - Driven high (driven to a 1)
  - Driven low (driven to a 0)
  - No driving value (Z or high impedance)
  - Capacitive high (H)
  - Capacitive low (L)
  - Conflict (one driver driving it to a 1, another a 0) (X)
  - Conflict of capacitive values (W)
- And other useful values
  - U – Uninitialized
  - – - a Don't Care

→ Definition of standard value set according to standard IEEE 1164: {'0', '1', 'Z', 'X', 'H', 'L', 'W', 'U', '-'}

Bus Resolution

- VHDL does not allow multiple concurrent signal assignments to the same signal
  - Multiple sequential signal assignments are allowed

```
LIBRARY attlib; USE attlib.att_mvl.ALL;
-- this code will generate an error
ENTITY bus IS
   PORT (a, b, c : IN MVL; z : OUT MVL);
END bus;

ARCHITECTURE smoke_generator OF bus IS
   SIGNAL circuit_node : MVL;
BEGIN
   circuit_node <= a;
   circuit_node <= b;
   circuit_node <= c;
   z <= circuit_node;
END smoke_generator;
```

Bus Resolution Functions

- VHDL uses bus resolution functions to resolve the final value of multiple signal assignments

```
FUNCTION wired_and (drivers : MVL_VECTOR) RETURN MVL IS
    VARIABLE accumulate : MVL := '1';
BEGIN
    FOR i IN drivers'RANGE LOOP
      accumulate := accumulate AND drivers(i);
    END LOOP;
    RETURN accumulate;
END wired_and;
```

- Bus resolution functions may be user defined or called from a package

Bus Resolution

- If a signal has a bus resolution function associated with it, then the signal may have multiple drivers

```
LIBRARY attlib; USE attlib.att_mvl.ALL;
USE WORK.bus_resolution.ALL;

ENTITY bus IS
   PORT (a, b, c : IN MVL; z : OUT MVL);
END bus;

ARCHITECTURE fixed OF bus IS
   SIGNAL circuit_node : wired_and MVL;
BEGIN
   circuit_node <= a;
   circuit_node <= b;
   circuit_node <= c;
   z <= circuit_node;
END fixed;
```

Assert Statement

- ASSERT statements are used to print messages at the simulation console when specified runtime conditions are met
- ASSERT statements defined one of four severity levels :
  - Note -- relays information about conditions to the user
  - Warning -- alerts the user to conditions that are not expected, but not fatal
  - Error -- relays conditions that will cause the model to work incorrectly
  - Failure -- alerts the user to conditions that are catastrophic

Assert Statements

- Syntax of the ASSERT statement

```
ASSERT condition
  REPORT "violation statement"
  SEVERITY level;
```

- **When the specified condition is *false*, the ASSERT statement triggers and the report is issued**
  - **The violation statement is enclosed in quotes**

```
ASSERT NOT((s='1') AND (r='1'))
  REPORT "Set and Reset are both 1"
  SEVERITY ERROR;
```

Testbenches

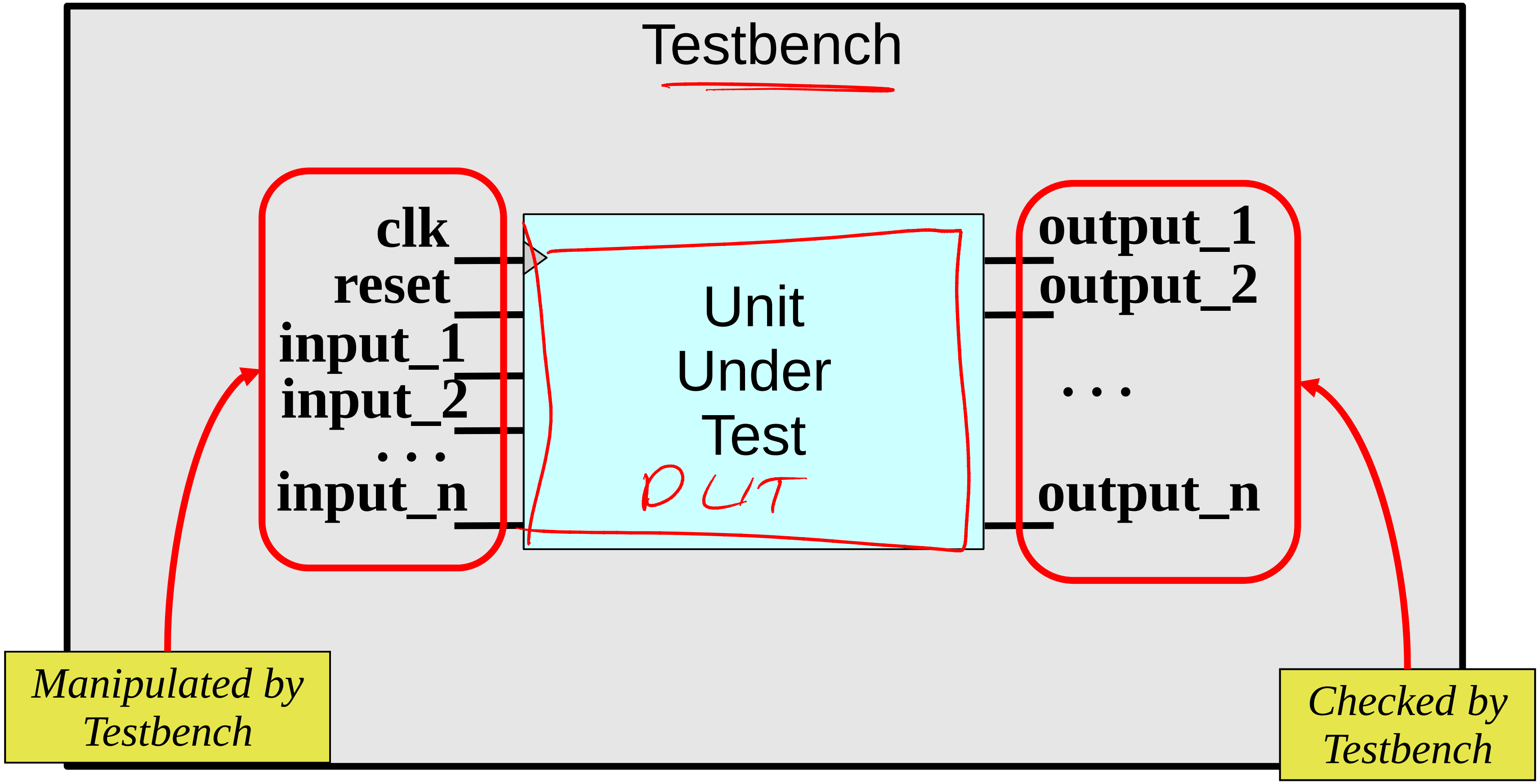
Why Use Testbenches?

- specify inputs and observe outputs

- advantages
  - test design without downloading to board
  - can program a test of all inputs as well automatically check expected behaviour

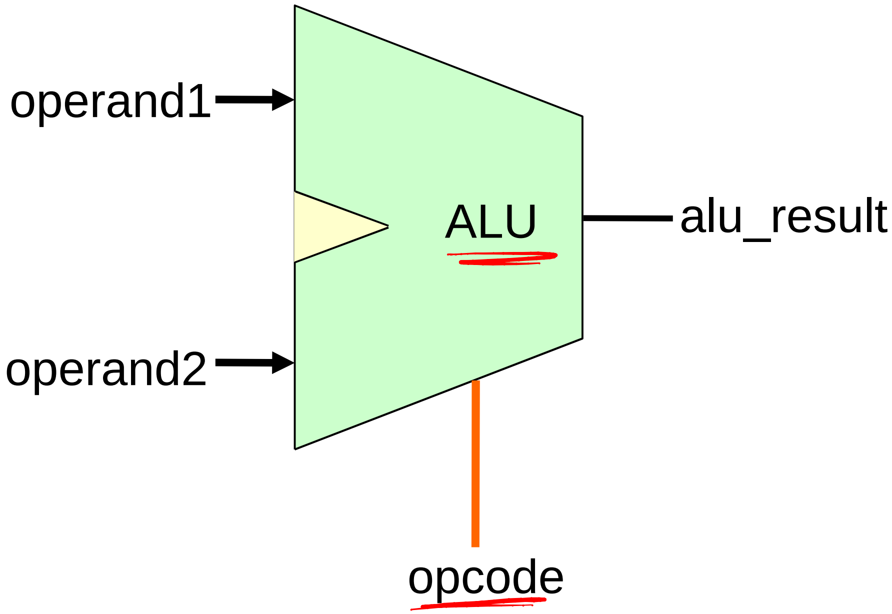
- disadvantages
  - cannot test all functionality (e.g. keyboard)
  - cannot determine/resolve timing issues
  - can only test a few combinations of input

Testbench Structure



Manipulating Input

- Input set as normal signal assignment




```
input: process()
begin
        operand1 <= "000";
        operand2 <= "101";
        opcode <= "100";
end process;
```

Checking Output

- Output read as a normal signal read
  - Checking done with assert-report-severity

Syntax:

assert *condition*
report *debug-string*
severity note | warning | error | failure;

Checking Output

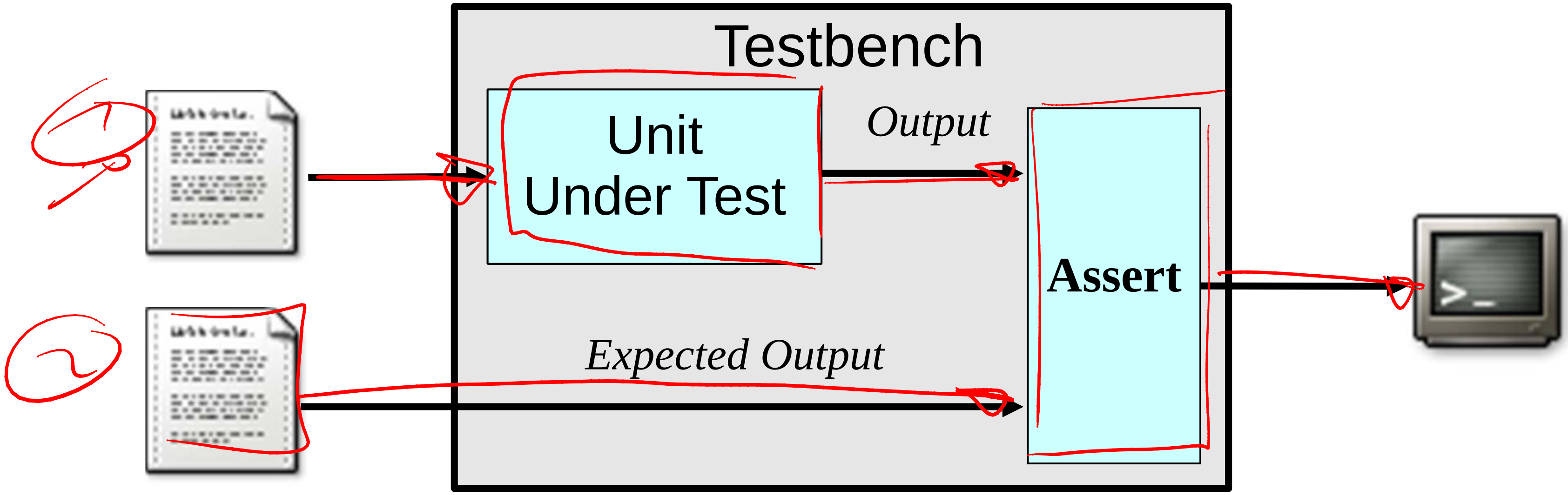
- Output read as a normal signal read
  - Checking done with assert-report-severity

Falling edge allows signal to be checked

```
verify: process(output)
begin
  if clk'event and clk = '0' then
    if (opcode = AND_OP)
        assert (alu_result = operand1 and operand2)
        report "Output is incorrect"
        severity error;
    end if;
  end if;
end process;
```

File Input

- Using assert to check behaviour implies writing correct behaviour twice
- Instead, write expected behaviour in file:



File Input

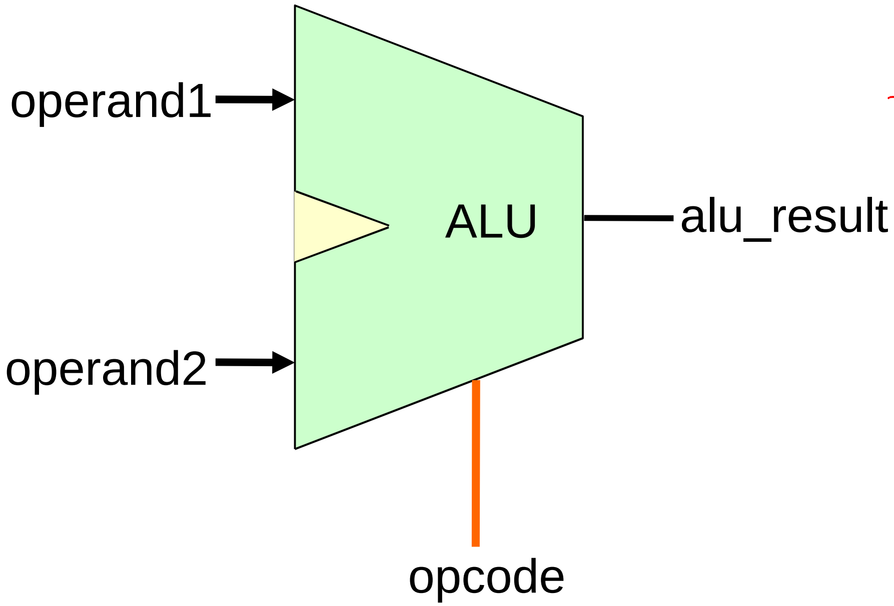
- Require `file` and `line` variables:

```
std.textio.all;
ieee.std_logic_textio.all;
```

```
InputProcess: process (clk)
  file input : text is "input.txt";
  variable line_in: line;
  variable op1_in, op2_in :
                 std_logic_vector(3 downto 0);
begin
  if clk'event and clk='1' then
         if not endfile(input) then
                readline(input, line_in);
                read(line_in, op1_in);
                read(line_in, op2_in);
                operand1 <= op1_in;
                operand2 <= op2_in;
         end if;
  end if;
end process;
```

File Input

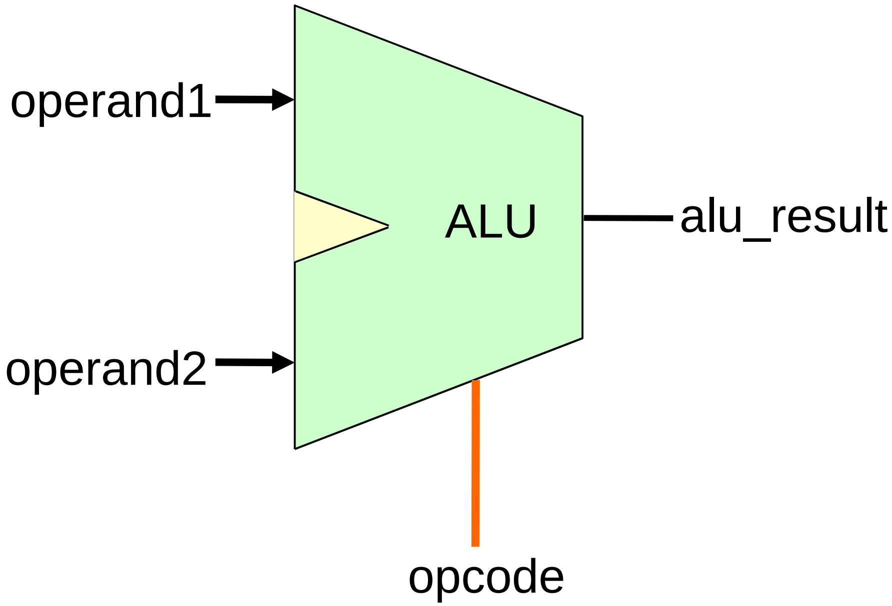
- Require `file` and `line` variables:




```
CheckProcess: process (clkt)
  file output1 : text is "output.txt";
  variable line_in: line;
  variable o_exp : std_logic;
begin
  if clk'event and clk='0' then
        if not endfile(output1) then
                readline(output1, line_in);
                read(line_in, o_exp);
                assert (o_exp = alu_result)
                report "Output incorrect"
                severity error;
        end if;
  end if;
end process;
```

Debug Output

- Lines can also be used to write to std_out:




```
CheckProcess: process(clk)
  variable line_out: line;
begin
  if clk'event and clk = '1' then
          write(line_out, string'("At time "));
          write(line_out, now);
          write(line_out, string'(", output is "));
          write(line_out, alu_result;
          writeline(output, line_out);
  end if;
end process;
```

Summary

- Behavioral VHDL is used to focus on the behavior, and not the structure, of the device
- Several familiar programming constructs, such as CASE and IF-THEN-ELSE statements, are available
- Subprograms allow large parts of code to be broken down into smaller, more manageable parts
- Bus resolution functions decide the final value of multiple signal assignments to one signal

VHDL for Performance Modeling

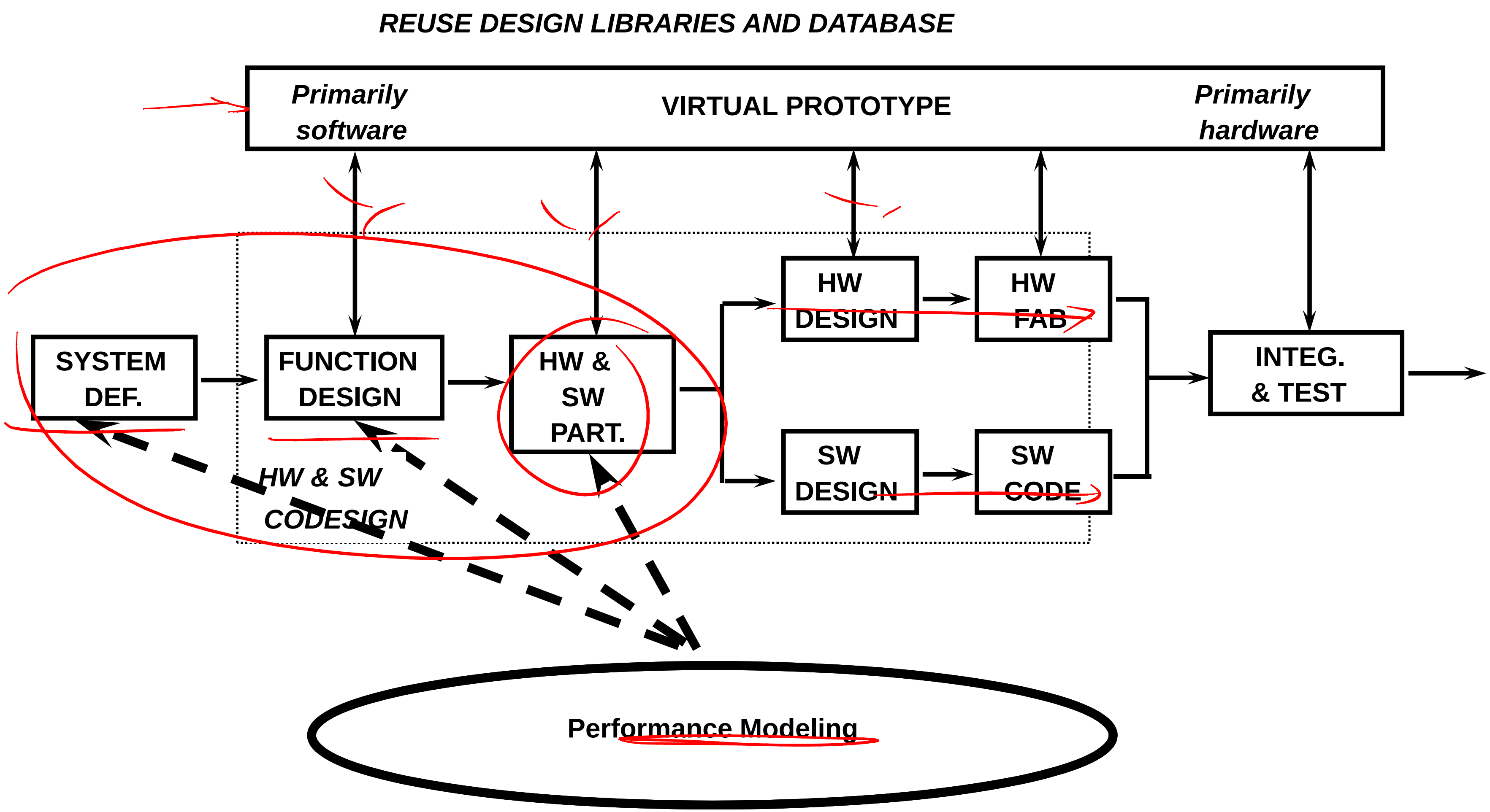
Goals:

- Estimate the performance of a given system by analyzing a high level model of the system
  - Model needs to include as little detail as necessary
    - Shorter model development time
    - Shorter model simulation time
    - Easier interpretation of the results
  - Model needs to produce as accurate results as possible
    - Increasing accuracy usually means increasing detail - a conflict with the goal above
    - Performance models often may not produce accurate absolute results, but will produce accurate comparative results with a similar model of another system alternative
    - Selecting the best candidate architecture can be performed with an abstract performance model, but model must be refined to ensure performance goals are met

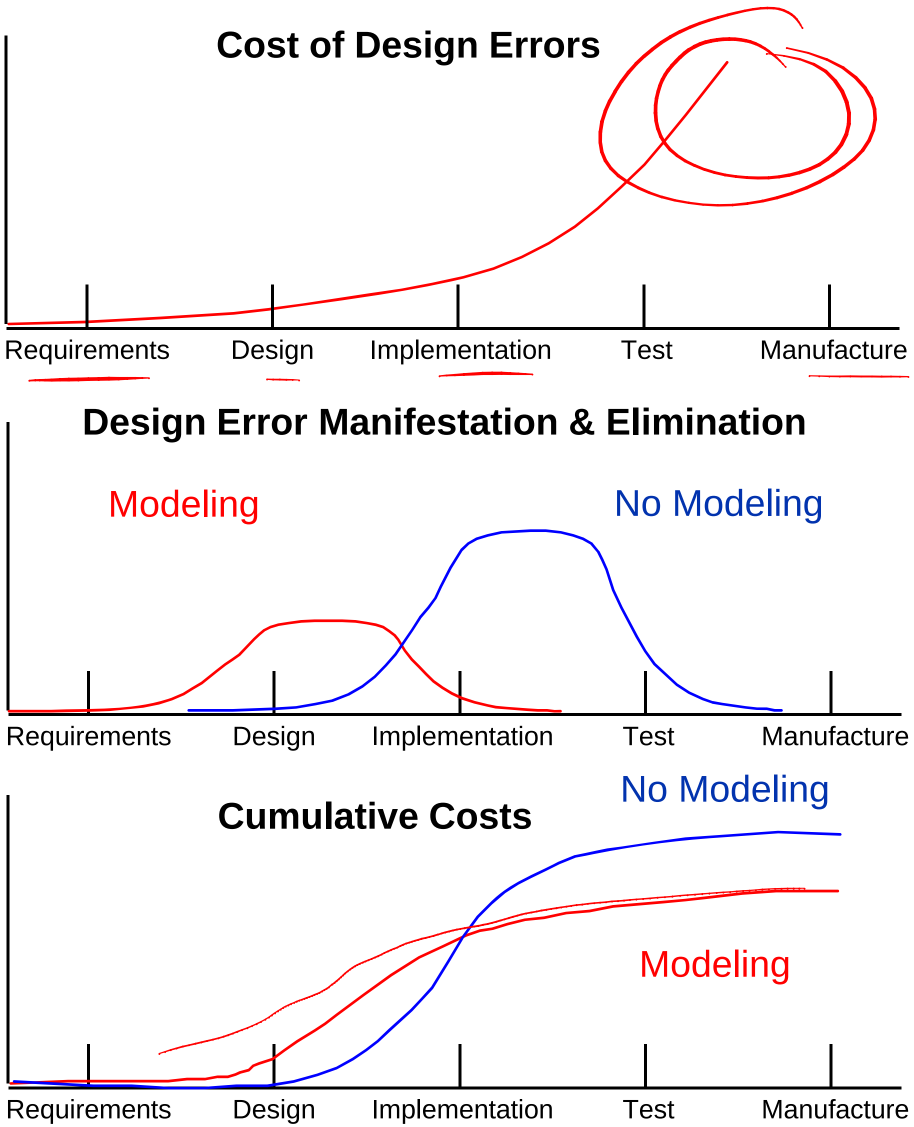
VHDL for Performance Modeling

- Performance models are used for:
  - Evaluating and comparing two or more design alternatives (architecture selection)
    - Hardware configuration
    - Software configuration
    - Hardware/software partitioning
  - Determining the number and size of components (system sizing)
  - Finding the system's performance bottleneck (bottleneck identification)
  - Determining the optimum value of a parameter (system tuning)
  - Characterizing the load on the system (workload characterization)
  - Predicting the system's performance at future loads (forecasting)

Rapid Prototyping Design Process



Performance Modeling Benefits



Performance modeling:

- aids in the evaluation of design alternatives,
- determines bottlenecks, overdesign, etc.,
- captures design decisions and assumptions,
- examines system behavior at boundary conditions,
- provides a focal point for early interaction of system, hardware, and software designers

[Hein96]