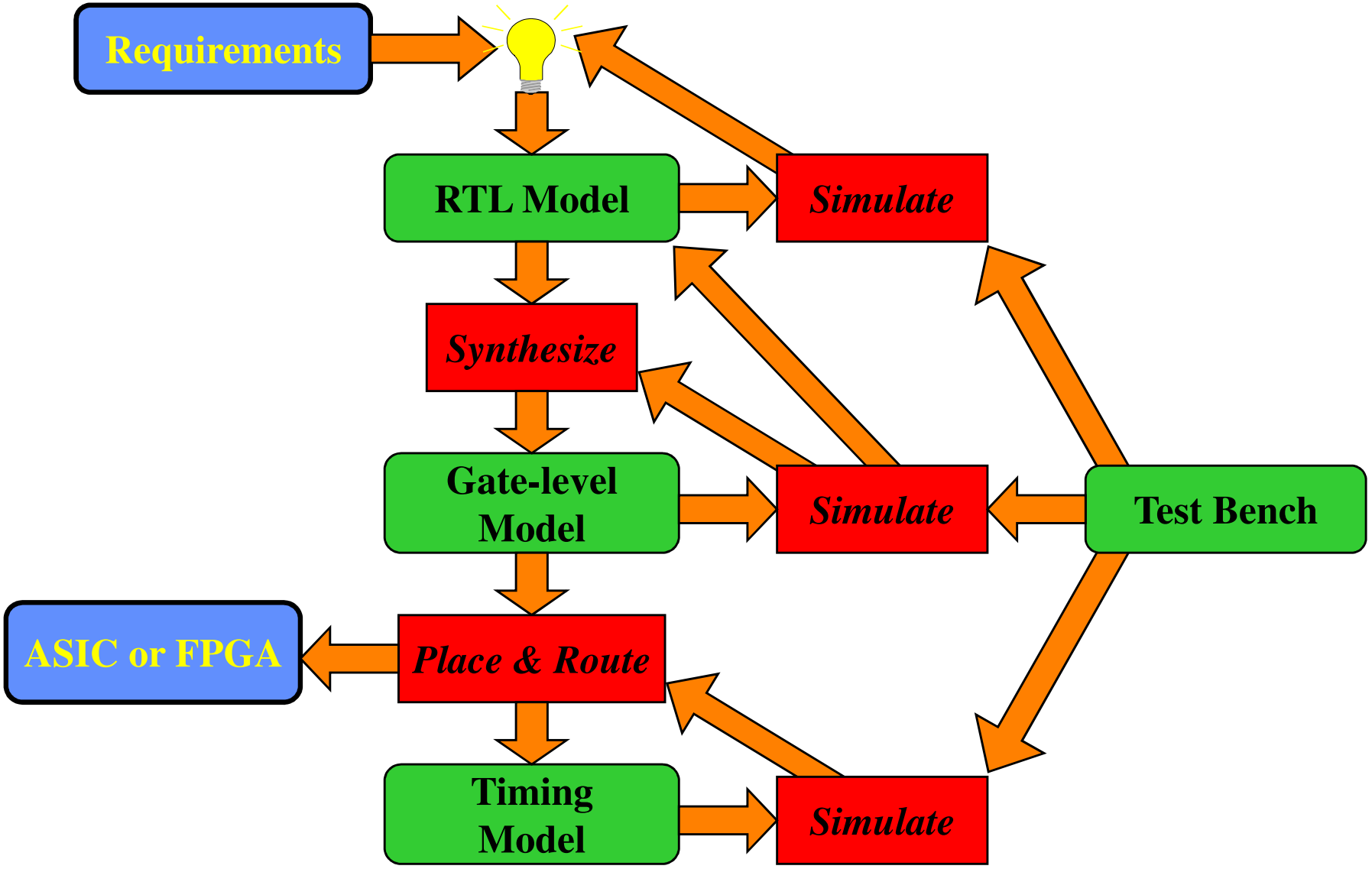
VHDL-based modeling and synthesis (Introduction)

- **Warning !**
  - **Not a course on the full language**
  - **Limited to frequent/recommended constructs and usage**
  - **With special focus on synthesizable RTL descriptions**

Basic design flow – from RTL to silicon



VHDL ?

- HDL = Hardware Description Language
- VHDL = VHSIC (Very High Speed Integrated Circuit) HDL

- Derived from ADA – US DoD project – Strongly typed language !
- IEEE standard (1076) issued in 1987 (ANSI standard in 1988)
- Revised 1993, 2002

- Targets digital circuits (further extension: VHDL-AMS)

- One of the main HDLs used in industry (with Verilog – industry de facto standard), especially for RT-level descriptions

A HDL is NOT a programming language

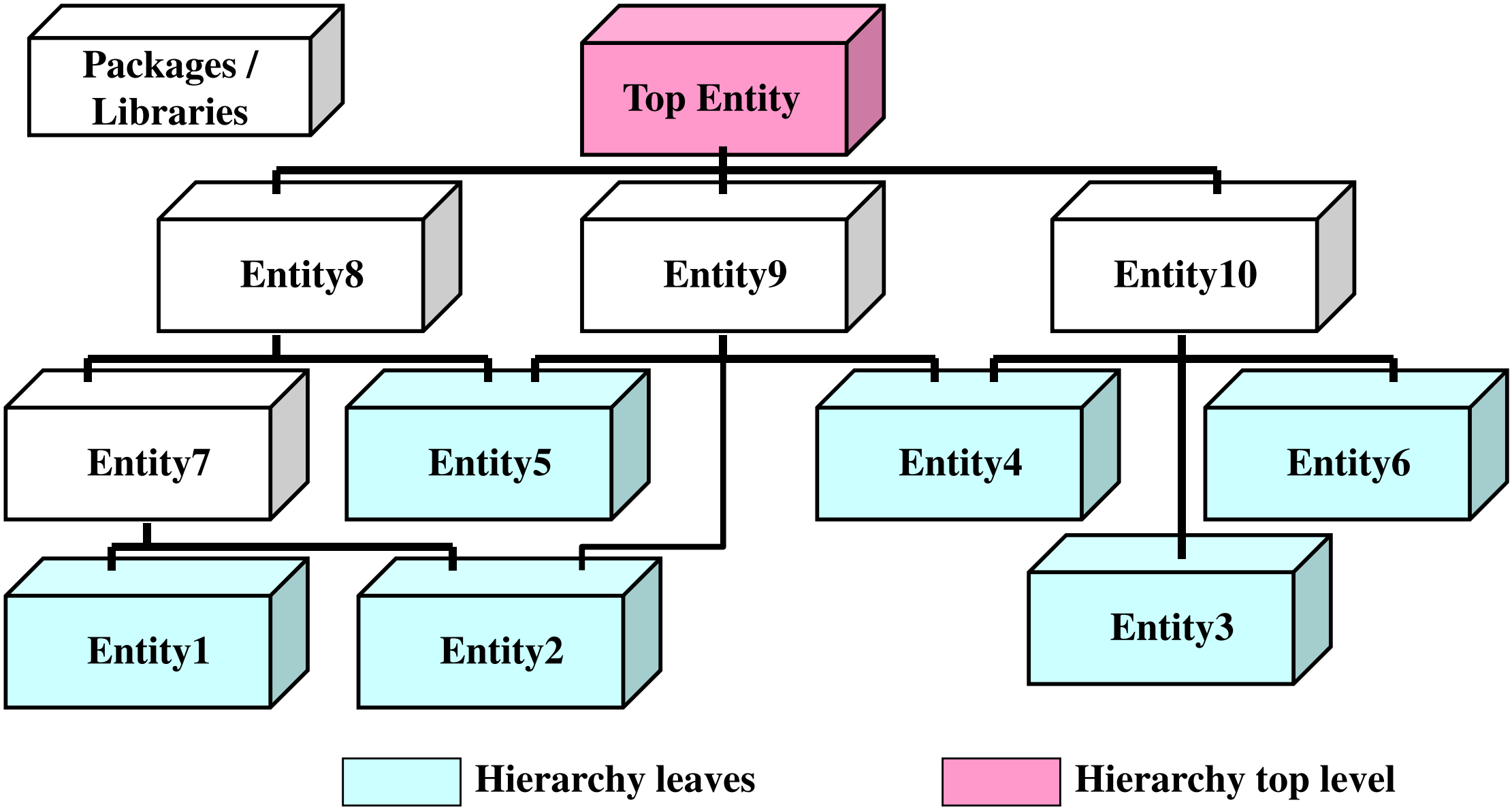
- Concurrency
- Structure and hierarchy specification
- Signals vs. variables
- Time modeling
- "data-flow" behavior: potentially infinite loop of computation sequence (e.g. asynchronous loops…), one event on a signal firing the evaluation of another variable/signal
- Interconnections and bus modeling

Reasons for using HDLs

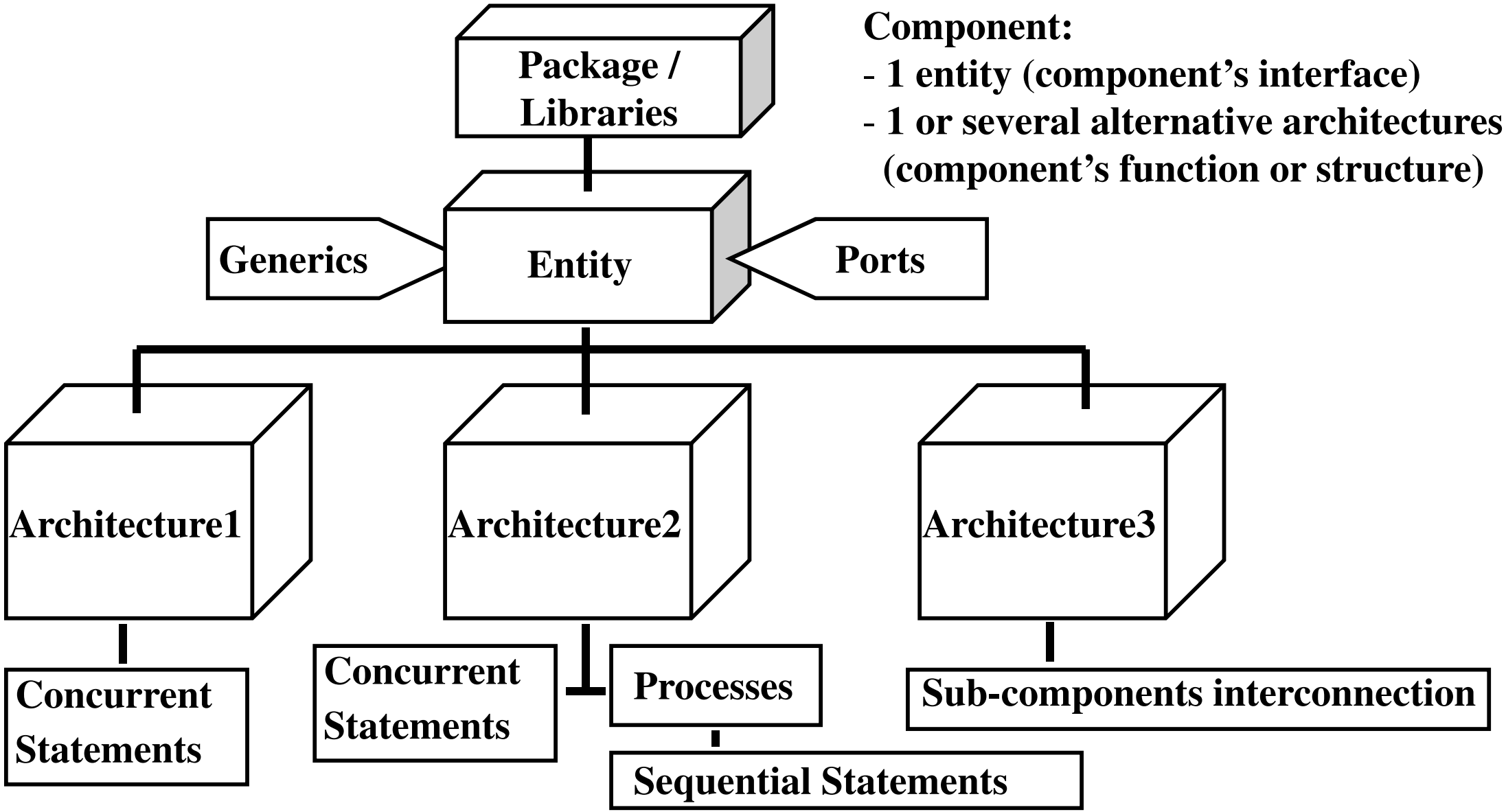
- HDLs enable multi-level hardware modeling (VHDL: from gate to some system level)
  - Progressive description refinements
  - Not all blocks at the same detail level
  - Design productivity improved by higher abstraction (reduced TTM)
  - Technology independence (at least up to some extent …)
  - BUT: don't forget this should still be hardware (at least for synthesizable parts) !
- They allow for various design methodologies, better design management, easier communication (standard languages) and a wide variety of digital hardware
- They provide mechanisms for digital design (simulation, synthesis, …) and design documentation

  => design re-use, pereniality

Circuit description: global view

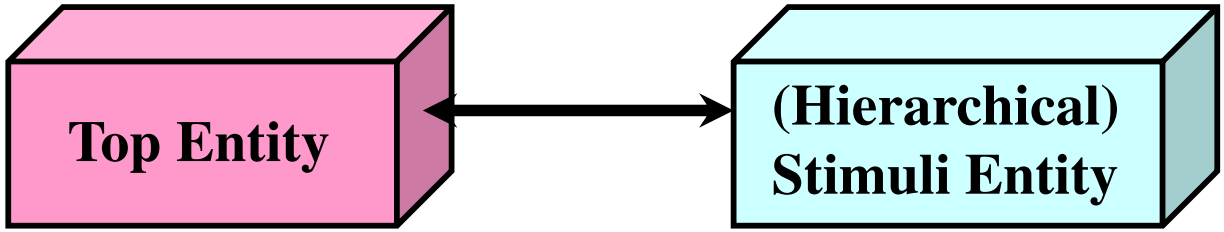


Component description: global view



Component:
- **1 entity (component's interface)**
- **1 or several alternative architectures (component's function or structure)**

System/Environment description: global view



Hierarchy top level
(may only be one block
of the complete designed circuit)

Validation environment (testbench)
- Test I/Os
- or Environment modeling

Synthesis oriented

Simulation oriented

Timing Model (simulation cycle)

- Discrete event simulation
  - Time advances in discrete steps
  - Evaluations when signal values change - *events*
- A process is sensitive to events on input signals
  - Specified in wait statements or sensitivity lists
  - Resumes and schedules new values on output signals
    - Schedules *transactions*
    - New events occur if new values different from old values

Entity declaration

- Corresponds to the schematic symbol (external interface)
  => PORT clause

- PORT (signal_name : mode data_type);

Mode = direction:
In - data can only be read
Out - data travels out

Inout - data may travel in either direction,
with any number of active drivers allowed; requires a Bus Resolution Function
Buffer - data may travel in either direction,
but only one signal driver may be on at any one time
Linkage - direction of data flow is unknown

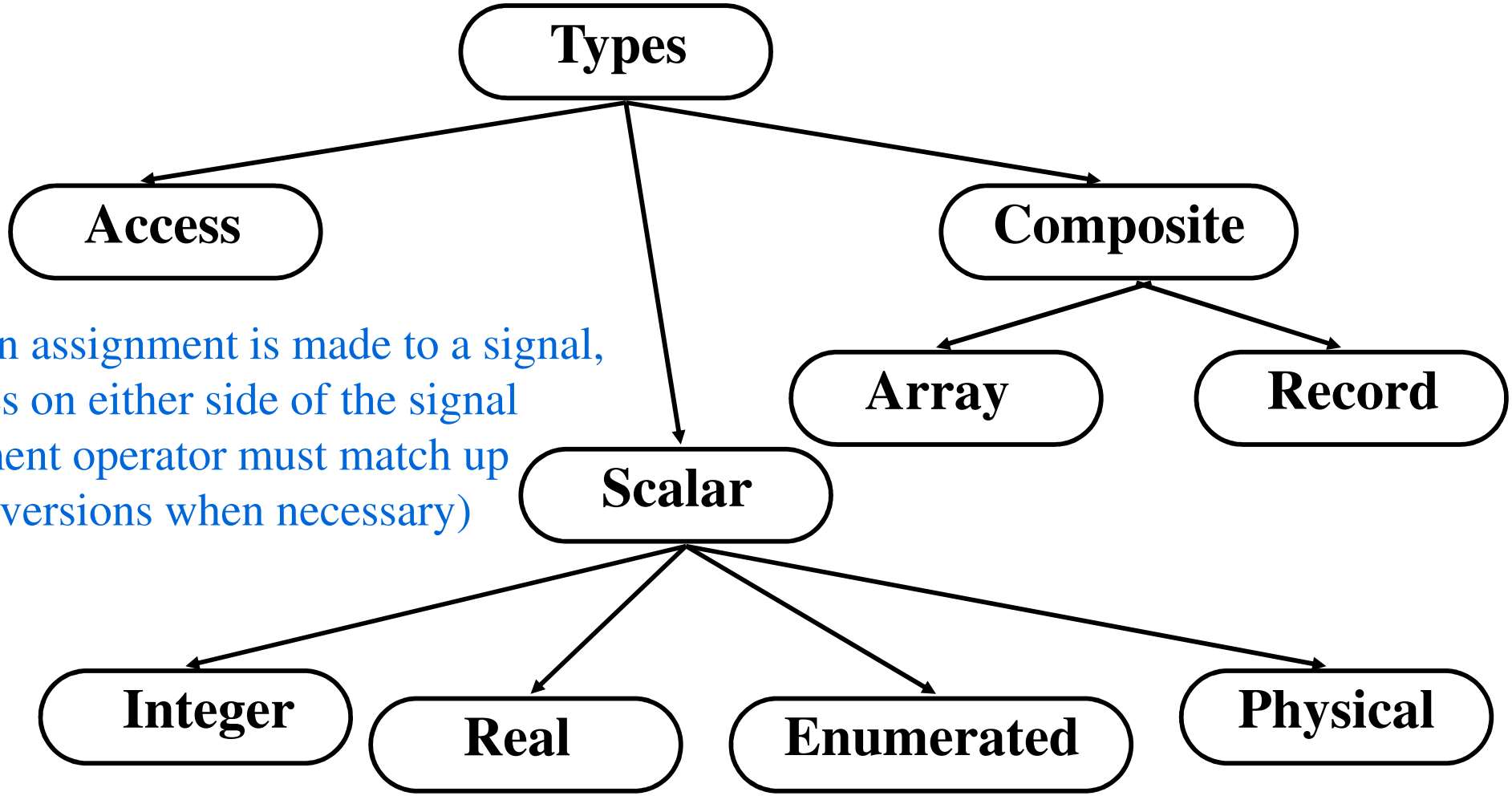
Legal VHDL identifiers

- Letters, digits, and underscores only (first character must be a letter)
- The last character cannot be an underscore
- Two underscores in succession are not allowed
- Using reserved words is not allowed (the VHDL editor will highlight reserved words for this reason)

- Examples
  - Legal: tx_clk, Three_State_Enable, sel7D, HIT_1124
  - Not Legal: _tx_clk, 8B10B, large#num, case, clk_

Data types

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



When an assignment is made to a signal, the types on either side of the signal assignment operator must match up (=> conversions when necessary)

Arrays in VHDL

- Assignment is by position number, not by index number (there is no concept of a most significant bit in the language)

- A "slice" of an array may be referenced, including a single element - the direction of the slice (i.e. to or downto) must match the direction in which the array is declared

```
SIGNAL  z_bus   : ARRAY(3 DOWNTO 0) OF BIT;
SIGNAL  c_bus   : ARRAY(1 TO 4) OF BIT;
```

```
z_bus <= c_bus;
```

is the same as:

```
z_bus(3) <= c_bus(1);
z_bus(2) <= c_bus(2);
z_bus(1) <= c_bus(3);
z_bus(0) <= c_bus(4);
```

Legal:

```
z_bus(3 downto 2) <= "00";
c_bus(2 to 4) <= z_bus(3 downto 1);
```

Illegal:

```
z_bus(0 to 1) <= "11";
```

Other standardized types

- Some limitations to the predefined “standard” types
- More powerful types defined by the IEEE standard 1164 (1993)
  => std_logic, std_logic_vector + functions
  (Warning: avoid std_ulogic, obsolete)

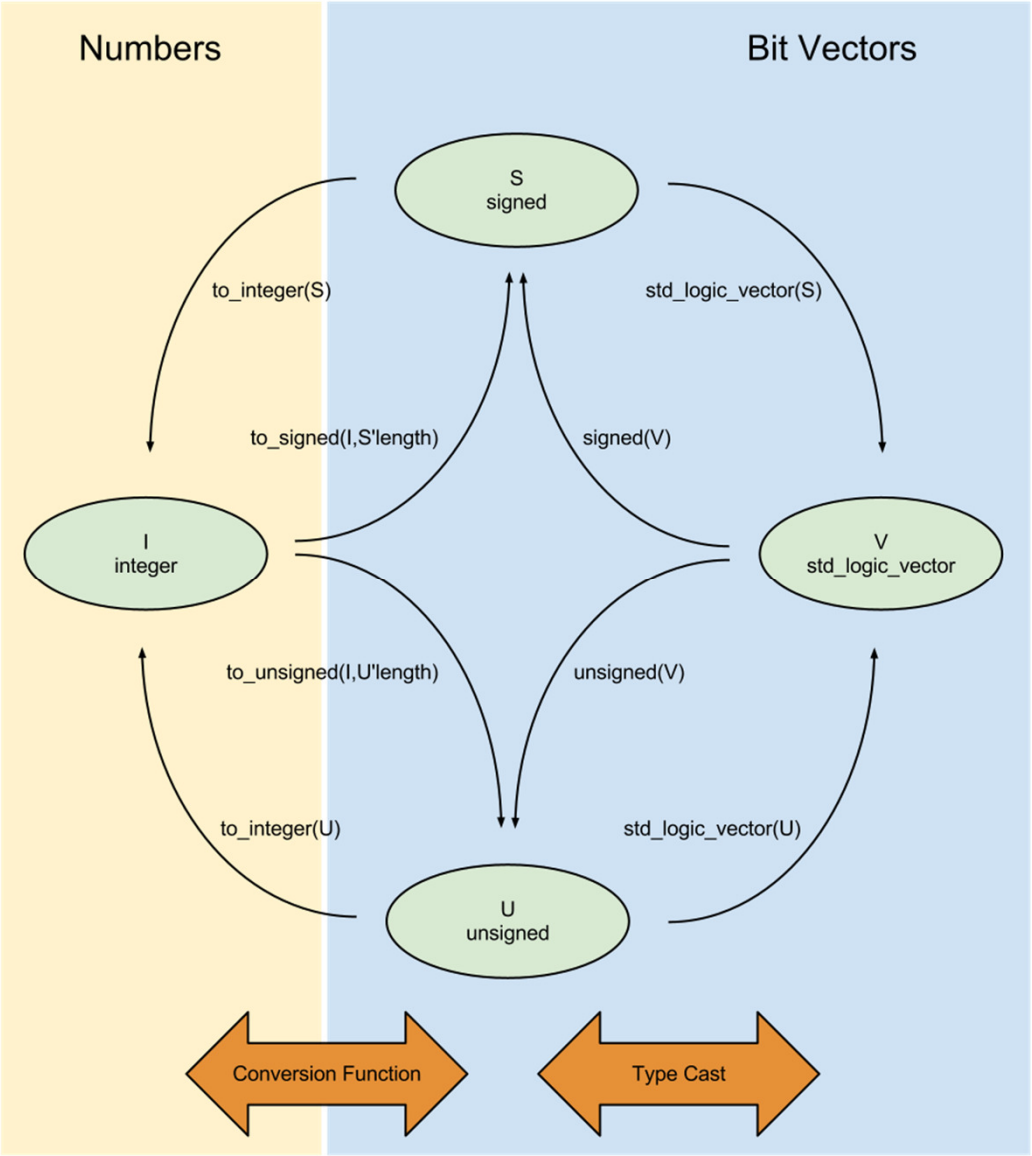
```
TYPE std_logic IS (
        'U'     uninitialized
        'X'     unknown
        '0'     logic 0          | Strong drive
        '1'     logic 1          |
        'Z'     high impedence
        'W'     unknown          |
        'L'     logic 0          | Weak drive (not
        'H'     logic 1          | usually used)
        '-');   don't care
```

  std_logic should be used ALL THE TIME for binary signals

- VHDL package NUMERIC_STD in IEEE Std 1076.3 (1997)
  => signed, unsigned + functions

Type conversions

- Strongly typed language => conversions often necessary (think about the initial type chosen for a given piece of data !)
- Close (compatible) types: possibility to cast a type
  - Example: [un]signed and std_logic_vector are essentially the same
- Conversion functions also available in packages



Architecture body

- Describes the operation of the component
- Consists of two parts:
  - Declarative part -- includes necessary declarations

    e.g. type declarations, signal/variable 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

VHDL objects

- Four types of VHDL objects
  - Constants
  - Signals
  - Variables
  - Files

- Scope:
  - 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

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: SIGNAL signal_name : type_name [:= value];
- Example signal assignment statement:

  Output <= My_id + 10;

- Note - compact assignment (e.g. initializations):

  my_reg <= (others => '0');

VHDL operators

- **Logical operators**
  - Include and, or, nand, nor, xor, xnor - All have the same precedence
  - Execute from left to right
  - NOT has a higher precedence, and therefore executes before other operators in an expression
  - Logical operations can only be applied to arrays of the same type and length (matching elements in arrays is by POSITION!)
  - Concatenation: &
- **Relational operators** (used to compare values - return a boolean)
  - < <= > >= = /=
  - For an array, the operands can be of different lengths; the operands are aligned to the left, and compared to the right - This makes 111 greater than 1011 !
- **Arithmetic operators** - predefined for types integer, real (except mod, rem), time
  - + - * / ** (exponential) abs (absolute) mod (modulus) rem (remainder)

Attributes

- Attributes provide information about items
- General form of attribute use:

  name_item'attribute_identifier (read as: “tick”)

- Predefined attributes (examples):
  - X'EVENT -- TRUE when there is an event on signal X (see register definition)
  - X'LAST_VALUE -- returns the previous value of signal X
  - X'HIGH -- returns the highest value in the range of X
  - X'STABLE(t) -- TRUE when no event has occurred on signal X in the past ‘t’ time (often used to verify timing characteristics, e.g. setup time violations)
  - T’left -- first (leftmost) value in type T
  - T’right -- last (righmost) value in type T
  - + array attributes, etc.

Architectures: description levels/styles

- **Behavioral**
  - Signal/variable declarations and affectations
  - Concurrent statements (data flow description – logic equations)
  - Sequential statements / processes

  => Synthesizable (circuit)

  => Pure behavior (testbench)
- **Structural**
  - Component declarations (from packages/libraries or user-specified)
  - Instances
  - Connections (port maps)

  => Netlist
  (down to gate level)
- **Mixte …**
- **+ timing/delays (only for testbench or simulation models)**

Structural descriptions

- Structural architecture = composition of subsystems
- Contents:
  - Signal declarations, for internal interconnections (the entity ports are also treated as signals)
  - Component declarations (basically corresponds to a copy of the entity declarations, replacing "entity" by "component")
  - Component instances
  - (Optional) generic maps
  - Port maps in component instances (connect signals to component ports) May be implicit (positional association - by signal ordering) or explicit (named association - by affectations)

```
U1:my_and
generic map (tpd => 5 ns)
port map (x => a, y => b, z => temp);
```

```
U1: my_and
generic map(5 ns)
port map(a, b, temp);
```

Structural description: example

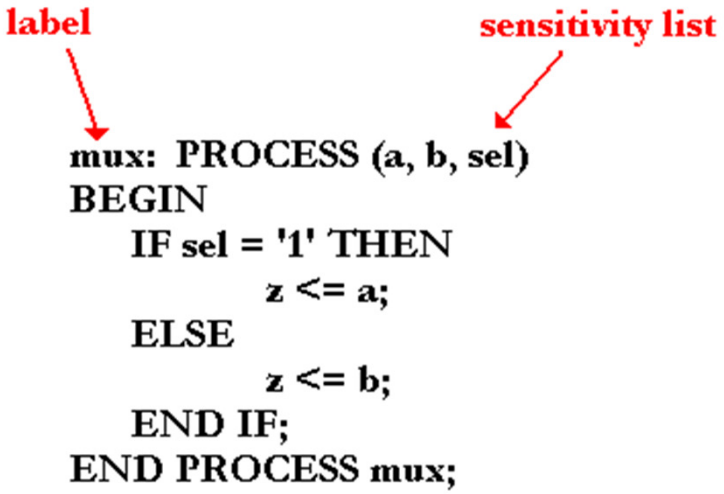
```
architecture struct of myCore is

  component DPCore
  port(
    BusIn : in std_logic_vector(15 downto 0);
    Clk : in std_logic;
    InitACC : in std_logic;
    …
    MQ0 : out std_logic);
  end component;

  component FSMCore
  port(
    Clk, Reset : in std_logic ;
    …
    Mq0 : in std_logic ;
    InitG : out std_logic ;
    …) ;
  end component;

signal Mq0 : std_logic ;
signal InitG : std_logic ; -- linked to InitACC
… ;
begin
  DPInst : DPCore
  port map (
    BusIn,
    Clk,
    InitG,
    …
    MQ0);
  FSMInst : FSMCore
  port map (
    Clk, Reset,
    …
    mq0,
    InitG,
    …) ;
end myCore ;
```

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)
- Concurrent VHDL statements include:
  - Block, process, assert, signal assignment, procedure call, component instantiation

Processes

- Syntax: optional label, the key word “PROCESS”, and a sensitivity list



- A process does execute continuously
- It is invoked when one of the signals in its sensitivity list changes value, or has an “event”
- Sensitivity lists MUST BE COMPLETE so that all processes execute properly!

Sequential statements

- Statements inside a process execute sequentially
- Major examples: If Then Else, If Then Elsif … Else, Case (with optional "when others" clause)

```
IF condition THEN
        -- sequential statements
END IF;

IF condition THEN
        -- sequential statements
ELSE
        -- sequential statements
END IF;
```

```
IF condition THEN
        -- sequential statements
ELSIF condition THEN
        -- sequential statements
ELSIF condition THEN
        -- sequential statements
ELSE
        -- sequential statements
END IF;
```

```
CASE object IS
    WHEN value_1 =>
        -- statements
    WHEN value_2 =>
        -- statements
    WHEN value_3 =>
        -- statements

    -- etc...
END CASE;
```

Signals vs. variables

- Different temporal behavior after affectations (driver modification)
  - Signal affectations scheduled during execution
  - Variable affectations have immediate effect in the order they appear in the VHDL code
- Synthesizable descriptions: signals recommended (closer to after synthesis behavior)
- Significant improvement of simulation time may justify using variables (one should make sure thet the behavior does not depend on the affection ordering)

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

Libraries

- Accessed via an assigned logical name, contain a package or a collection of packages (compiled !)
  - Analogous to directories of files
  - Libraries contain analyzed (i.e. *compiled*) entities, architectures, and packages
- Resource libraries
  - Standard package
  - IEEE developed packages
  - Vendor proprietary component (or type) packages
- Working libraries
  - Default Work library
  - Any library of (pre-existing) design units that are referenced in a design
  - Library into which the unit is being compiled (many libraries can be created for a given circuit: blocks, testbenches, top level …)

Referencing libraries

- All packages must be compiled
- Implicit libraries (items in these packages do not need to be referenced, they are implied)
  - Work
  - Std - contains packages Standard (types Bit, Boolean, Integer, Real, and Time + all operator functions to support types) and Textio (file operations)
- LIBRARY clause
  - Defines the referenced library name (symbolic name to path/directory)
  - Name defined at compile time
- USE clause
  - Defines the library elements used in the design (may be a superset)
- Example:

```
Library ieee;
Use IEEE.std_logic_1164.all;
```

Configurations

- Mechanism to define correspondences between one entity and one (out of several) architecture/implementation
  - Alternative descriptions of the same component (e.g. behavioral or structural; implementation_1 or implementation_2; …)
  - Several description levels (e.g. behavioral and netlist after synthesis)
- Can be defined in a separate file or on the fly (i.e. in the file describing the entity and the architectures)
- Flexibility
  - Several configurations can be specified for a given component (e.g. to simulate before or after synthesis of a given block)
  - Possibility to configure all sub-blocks in the comonent hierarchy with a single configuration … or to create a hierarchy of configurations

Configurations: syntax

```
for architecture_name
  for instance_name : component_name  use entity lib_name.entity_name(archi_name);
  end for ;
 end for ;
```

Can be hierarchical - example:

```
Library lib_myCore ;
configuration cfg_myCore_Bench of myCore_Bench is
  for behav
    for BlockTest : myCore  use entity lib_myCore.myCore(struct) ;
          for struct
             for DPInst : DPCore use entity lib_myCore.DPCore(RTL16b) ;
             end for;
             for FSMInst : FSMCore use entity lib_myCore.FSMCore(behav) ;
             end for;
          end for ;
    end for ;
  end for ;
end cfg_myCore_Bench ;
```

Writing testbenches

- Testbench: test component for a given design
  - An architecture body that includes an instance of the design under test
  - Applies sequences of test values to inputs (or model environment behavior)
  - Allows for generating waveforms or (better !!) monitors values on output signals

- Self-testing testbenches
  - Either using Assert clauses
    e.g. assert BusOut="00000000" report "Erroneous MSBs" severity Warning;
  - Or with a process that verifies correct operation
  - Or by comparing with a reference component (e.g. non regression test)
    - stimulates both components with same inputs
    - compares outputs for equality, potentially taking into account timing differences

RTL modeling (for synthesis) – main guidelines

- Hierarchy: don't mix structure and behavior in the same architecture
  - Hierarchy leaves: behavioral descriptions
  - Other levels: structural descriptions
- Architecture bodies: separate sequential and combinatorial logic parts ("two process" description style)
  - Sequential processes depend on the clock and define the global circuit memory elements and synchronization (memorization conditions)
  - Combinatorial processes define the logic/arithmetic functions (next value computations of memorized signals) – They should not infer any memory element

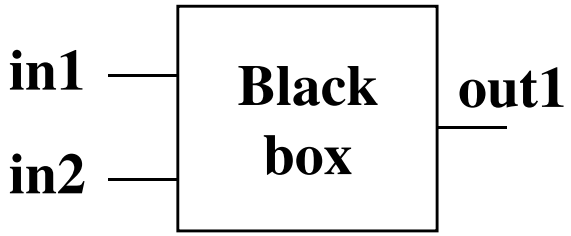
RTL modeling (for synthesis) – pitfalls

- Don't use std_ulogic types …
- Take care of obsolete libraries, e.g. arithmetic (to be replaced by numeric_std, implies different description constraints)
- Avoid confusions
  - Naming conventions to be applied
  - Only one entity per vhd file, with file name = entity name
  - Organization of files in several directories (vhd, bench, synth, P&R, …)
  - Only one architecture per file unless a real use of alternative descriptions
  - Management of versions (e.g. behavioral or netlist description of the same block) through configurations, specified in SEPARATE files

A AND gate in VHDL

```
entity AND2 is
port (
  in1, in2 : in std_logic;
  out1 : out std_logic
);
end AND2;
```

std_logic rather than BIT




```
architecture behavConcur of AND is
begin
 out1 <= in1 and in2 ;
end behav;
```

Concurrent affectations (equations)

```
architecture behavSeq of AND is
  signal temp : std_logic;
begin
process (in1, in2 )
  begin
    temp <= in1 and in2;
  end process;
  s <= temp ;
end behav;
```

Process :
- **Sequential statements**
- **Sensitivity list: include all inputs !!**
- **Define outputs in ALL possible cases**

Case and If then else statements …

- Case: better when no required priority (balanced critical path)
- If then else: may be better in some cases, when unbalanced critical paths can be an advantage (ex: late arrival of one of the input signals)
- The case statement is better for simulation speed
- Note:
  - A “If then else” statement encodes a priority scheme - conditions may not be exclusive
  - A “case” statement generates a one-level logic without priority – conditions must be exclusive

A "ROM" in VHDL

```
entity ROM is
      port(  Rom_Address    : in STD_LOGIC_VECTOR(3 downto 0) ;
             Rom_out        : out STD_LOGIC_VECTOR(7 downto 0 )) ;
end ROM;

architecture A of ROM is
type tab_rom is array (0 to 15) of STD_LOGIC_VECTOR(7 downto 0);
constant rom : tab_rom :=
      ( 0  => "00001101" , 1  => "00010101" , 2  => "00011111" , 3  => "00101100" ,
        --  0x0D          0x15          0x1F          0x2C
        4  => "00111100" , 5  => "01001101" , 6  => "01100001" , 7  => "01110101" ,
        --  0x3C          0x4D          0x61          0x75
        8  => "10001010" , 9  => "10011111" , 10 => "10110011" , 11 => "11000101" ,
        --  0x8A          0x9F          0xB3          0xC5
        12 => "11010100" , 13 => "11100001" , 14 => "11101001" , 15 => "11101110" );
        --  0xD4          0xE1          0xE9          0xEE

begin
  Rom_out <= rom(conv_integer(Rom_Address)) ;
end A;
```

A latch in VHDL

D latch:

```
process (clk, D)
  begin
    if (clk='1') then
       Q <= D;
    end if;
end process;
```

Remark:
D must be in the sensitivity list (must be transferred when the latch is transparent, without any event on the clock signal)

```
process (clk, D, reset)
  begin
    if (reset='0') then Q <= '0';
      elsif (clk='1') then
         Q <= D;
    end if;
end process;
```

D latch with asynchronous initialization

```
process (clk, D, reset)
  begin
    if (clk='1') then
      if (reset='0') then  Q <= '0';
        else  Q <= D;
      end if;
    end if;
end process;
```

D latch with synchronous initialization

A flip-flop in VHDL

D flip-flop:

```
process (clk)
  begin
    if (clk='1' and clk'event) then
        Q <= D;
    end if;
end process;
```

Remark:
D is not necessarily in the sensitivity list (sometimes better for synthesis, but longer simulations)

Redundant if D is not in the sensitivity list …
but more readable !

```
process (clk, reset)
  begin
    if (reset='0') then Q <= '0';
      elsif (clk='1' and clk'event) then
          Q <= D;
    end if;
end process;
```

D flip-flop with asynchronous initialization

```
process (clk)
  begin
    if (clk='1' and clk'event) then
      if (reset='0') then  Q <= '0';
        else  Q <= D;
      end if;
    end if;
end process;
```

D flip-flop with synchronous initialization

Finite state machine example

```
entity FSM is
port (
  clk, reset : in std_logic;
  in1, in2   : in std_logic;
  out1, out2 : out std_logic
);
end FSM;

architecture archi of FSM is
  type STATE_TYPE is (state_0, state_1,
                      state_2, state_3, state_4);
  signal state, next_state : STATE_TYPE;
begin
  process (clk, reset)
  begin
    if (clk='1' and clk'event) then
      if (reset='0') then  state <= state_0;
      else  state <= next_state;
      end if; end if; end process;
```

```
process (state, in1, in2)
 begin
   case state is
     when state_0 =>
       if (in1='0') then
         next_state <= state_0;
       else
         next_state <= state_2;
       end if;
     when state_1 =>
     …
   end case;
end process;
```

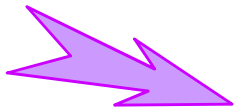
```
process (state)
  begin
    case state is
      when state_0 =>
        out1 <= '0';
        out2 <= '0';
      when state_1 =>
      …
    end case;
  end process;

end archi;
```

Example of Moore machine

Several description levels

```
when multin =>
  Valid_temp <= '1' ;
  if (operandB(counter)='1') then
    nstate <= addit ;
    naccumulation <= accumulation + operandA ;
    noperandA <= operandA ;
    ncounter <= counter ;
  else nstate <= shifting ;
       noperandA <= operandA + operandA ;
       ncounter <= counter + 1 ;
       naccumulation <= accumulation ;
  end if ;
```

Behavioral description:
RTL "control flowchart"



```
when multin =>
  Valid_temp <= '1' ;
  SelMQFF <= '0' ;
  SelD <= '1' ;
  SelB <= '0' ;
  if (Mq0='1') then nstate <= addit ;
        SelACCFF <= '1' ;
        InitD <= '1' ;
        SelACCMux <= '0';
        SelMQMux <= '0' ;
        SelCPTR <= '0' ;
    else nstate <= shifting ;
        SelACCFF <= '0' ;
        InitD <= '0' ;
        SelACCMux <= '1' ;
        SelMQMux <= '1' ;
        SelCPTR <= '1' ;
  end if ;
```

Finite state machine