Chapter 6.5. VHDL Design

What does HDL stand for?

HDL is short for Hardware Description Language

(**VHDL** – VHSIC **H**ardware **D**escription **L**anguage)
(Very High Speed Integrated Circuit)

What's VHDL?

- VHDL is a programming language for describing digital system behavior and structure. It can be used to describe, model and design digital systems.
- VHDL is a industry standard language used to describe hardware from the abstract to the concrete level.
- Originally intended for simulation, modeling and documentation
- Later also used for synthesis
- Originally tightly connected with US DoD, but soon found its way to non-military applications
- First standard in 1987, revised in 1993.

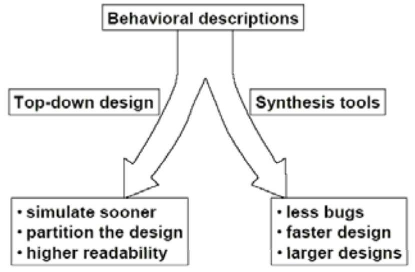
The World Before VHDL

- Polygon pushing
- Transistor level design
- Boolean design
  - One equation for each FF data input
- Schematic Design
  - Allow use of blocks in addition to FFs and gates. Impractical for large designs
- HDL Design, used for
  - Requirement specification
  - Documentation
  - Testing using simulation
  - Verification of correctness before manufacturing
  - Synthesizing digital circuits: implementation at Register Transfer Level → Netlist of elements

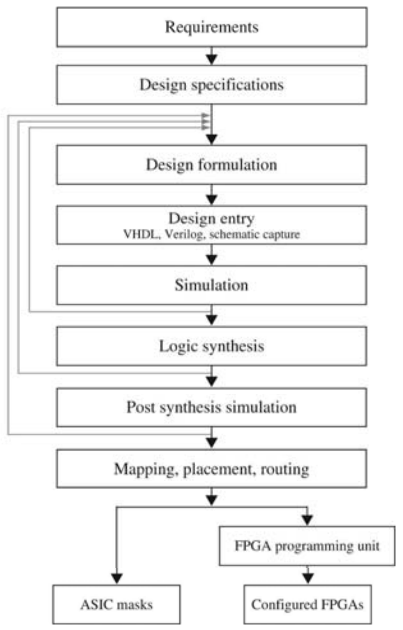
Why VHDL?

- "Universal" Language proposed by USA DoD
- IEEE standard
- Wide industry acceptance
- Supports different methodologies
- Supports different technologies
- Supports technology-independent abstract models
- Helps component re-use
- Self-documenting
- Many tools available

Advantages

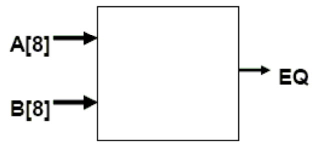


Design Flow



VHDL Design: Entity + Architecture

Simple Example: a Comparator

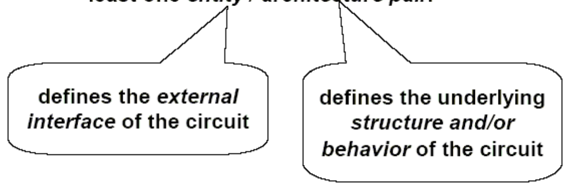


```
Entity Compare is
port(A, B: in std_logic_vector(0 to 7);
         EQ: out std_logic);
end Compare;
architecture comp_1 of Compare is
begin
     process (A, B)
      begin
         if (A=B)
           then  EQ<= '1';
           else  EQ<= '0';
         end if;
     end process;
end comp_1;
```

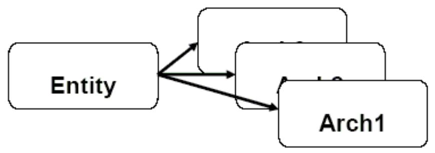
Entities and Architectures

❑ The minimum VHDL design description must include at least one entity and its bounded architecture.

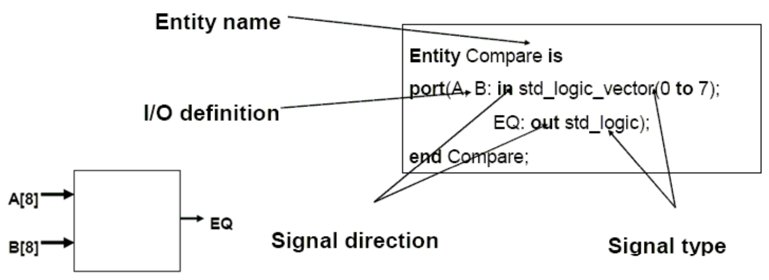
• **Each VHDL design description consists of at least one *entity / architecture pair.***


defines the *external interface* of the circuit
defines the underlying *structure and/or behavior* of the circuit

But VHDL allows the designer to create different alternate architectures for each entity



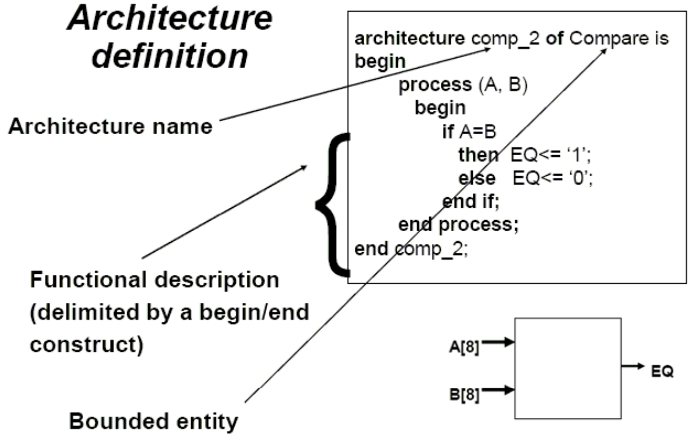
Entity declaration



```
Entity Compare is
port(A, B: in std_logic_vector(0 to 7);
          EQ: out std_logic);
end Compare;
```

- It provides the interface for the circuit
- It does not include the actual circuit behavior
- It allows to connect the circuit into higher level circuits
- VHDL is case-insensitive.

Architecture definition



```
architecture comp_2 of Compare is
begin
    process (A, B)
      begin
         if A=B
           then  EQ<= '1';
           else   EQ<= '0';
         end if;
    end process;
end comp_2;
```

- It is always related to one entity;
- It describes the behavior or the structure of the circuit
- For each entity, it is possible to have more than one architecture.

VHDL Design Units

- Entity: specifies the interface of the system with the environment.
- Architecture: description of the internal part of the system, specifies how the inputs are transformed into outputs.
- Process: Concurrency, event controlled
- Configuration
  - used to combine a component instance to an entity-architecture pair.
- Package
  - Encapsulate elements that can be shared globally among design units.
- Library: Compilation, object code

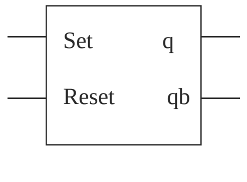
Entity

Interface description :

- Defines connections (ports) that transfer information to and from the system.
- Defines port types : IN, OUT, INOUT
- Architecture only allowed to read IN ports, or write to OUT ports. INOUT ports can be read or written to.

e.g.

```
ENTITY rsff IS
  PORT (set, reset : IN BIT;
               q,qb : INOUT BIT);
END rsff;
```



RSFF

Entity Declaration

entity NAME_OF_ENTITY **is**
  port (*signal_names*: **mode** *type*;
    signal_names: **mode** *type*;
      :
     signal_names: **mode** *type*);
end [NAME_OF_ENTITY] ;

MVL - 9			
Uninitialized	'U'	Weak 1	'H'
Don't Care	'-'	Weak 0	'L'
Forcing 1	'1'	Weak Unknown	'W'
Forcing 0	'0'	High Impedance	'Z'
Forcing Unknown	'X'		

- NAME_OF_ENTITY: user defined
- signal_names: list of signals (both input and output)
- mode: in, out, buffer, inout
- type: boolean, integer, character, std_logic

Architecture

<u>Implementation of the design</u> :

- All entities have one or more architecture
- Describes the functionality of the system.
- Always connected with a specific entity
- entity ports are available as signals within the architecture
- The description can be structural or behavioral.
- **Structural** : Specifies which sub-components are used and how they are connected.
- **Behavioral** : Specifies what the system does, describes the outputs' responses to the inputs' changes.

Architecture for Entity

- Describes an implementation of an entity
- May be several per entity
- Contains concurrent statements

declarative part

definition part

```
architecture Behav of Reg4 is
  component Reg1
    port (...);
  end component;
  signal s1,s2 : std_logic;
begin
  s1 <= s2;
  ...
end Behav;
```

additional signals

concurrent statements

- Structural: describe the design as combination of building blocks
- Behavioral: describe algorithm/function of the design/module
- Mixed structural and behavioral
  - Example: Register Transfer Level (RTL) modeling
    - Data path described structurally
    - Control section described behaviorally

Architecture: Behavioral

Declarative part:

- data types
- constants
- additional signals ("actual" signals)
- components
- ...

Definition part (after 'begin'):

- signal assignments
- processes
- component instantiations
- concurrent statements: order not important

```
architecture EXAMPLE of STRUCTURE is
  subtype DIGIT is integer range 0 to 9;
  constant BASE: integer := 10;
  signal DIGIT_A, DIGIT_B: DIGIT;
  signal CARRY:            DIGIT;
begin
  DIGIT_A <= 3;
   SUM <= DIGIT_A + DIGIT_B;
   DIGIT_B <= 7;
   CARRY <= 0 when SUM < BASE else 1;
end EXAMPLE ;
```

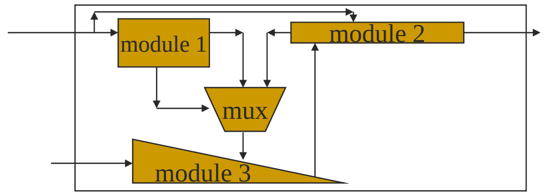
Behavioral Architecture Examples

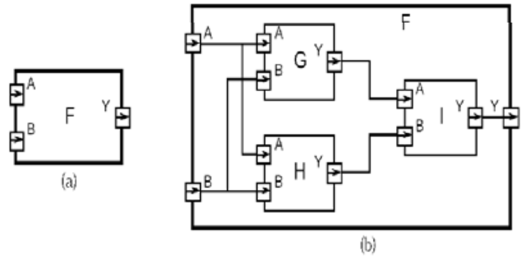
```
entity compare is
 port(A,B: in std_logic_vector(7 downto 0);
      EQ: out std_logic);
end compare;

architecture compare1 of compare is
begin
  EQ <= '1' when (A = B) else '0';
end compare1;
```

```
entity rotate is
  port( Clk, Rst, Load: in std_logic;
  Data: in std_logic_vector(7 downto 0);
  Q: out std_logic_vector(7 downto 0));
end rotate;

architecture rotate1 of rotate is
begin
  reg: process(Rst,Clk)
  variable Qreg:std_logic_vector(7 downto 0);
  begin
    if Rst = '1' then
--Async reset
    Qreg := "00000000";
    elsif (Clk = '1' and Clk'event) then
       if (Load = '1') then Qreg := Data;
       else Qreg:=Qreg(0) & Qreg(7 downto 1);
     end if;
    end if;
    Q <= Qreg;
   end process;
end rotate1;
```

Architecture: Structural

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





Structural Architecture: Example

- In declarative part of architecture.

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

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

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

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

   begin
```

Structural Architectures- **Instantiation**

- **Instantiation in definition part of architecture (after 'begin')**

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

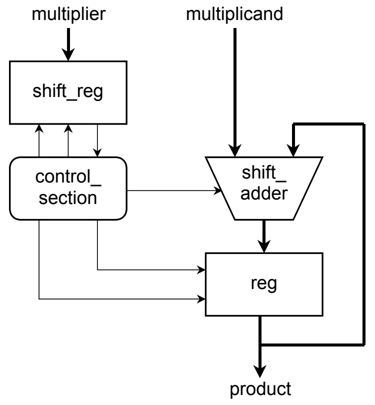
Port Association

- **Two methods of port association are available:**
- **Positional port association**
  - **e.g. port map(A,B,C,open,E);**
    - **order is critical**
  - **Named port association**
    - **e.g port map:**

    (Sum=>S, Carry=>open, IN1=>X, IN2=>Y);
    - **left side: "formals"**
      (port names from component declaration)
    - **right side: "actual“ (architecture signals)**
  - **Independent of order in component declaration**

Mixed Behavior and Structure

- An architecture can contain both behavioral and structural parts
  - process statements and component instances
    - collectively called *concurrent statements*
  - processes can read and assign to signals
- Example: register-transfer-level model
  - data path described structurally
  - control section described behaviorally



Mixed Example

```
entity multiplier is
    port ( clk, reset : in bit;
            multiplicand, multiplier : in integer;
            product : out integer );
end entity multiplier;

architecture mixed of mulitplier is
    signal partial_product, full_product : integer;
    signal arith_control, result_en, mult_bit, mult_load : bit;
begin
    arith_unit : entity work.shift_adder(behavior)
        port map ( addend => multiplicand,  augend => full_product,
                    sum => partial_product,
                    add_control => arith_control );
    result : entity work.reg(behavior)
        port map ( d => partial_product,  q => full_product,
                    en => result_en,  reset => reset );
    ...
```

Mixed Example

```
    …
    multiplier_sr : entity work.shift_reg(behavior)
        port map ( d => multiplier,  q => mult_bit,
                    load => mult_load,  clk => clk );
    product <= full_product;
    control_section : process is
        -- variable declarations for control_section
        -- …
    begin
        -- sequential statements to assign values to control signals
        -- …
        wait on clk, reset;
    end process control_section;
end architecture mixed;
```

Data Types

❑ Like a high-level software programming language, VHDL supports different data types.
❑ Data types allow the user to represent
- high-level data (real, integer, string, …)
- values got by individual wires in a circuit

❑ Every data type can get a defined set of values.
❑ VHDL is strongly-typed: strong restrictions on how operations involving different data-types can be intermixed.

Data Type	Values	Example
std_logic	'0','1','-','X','U','Z','L','H','W'	SUM <= '1';
std_logic_vector	(array of std_logic)	Data_out <= "0010";
boolean	True, False	EQ <= True
Integer	-2, -1, 0, 1 ,2 ...	Count <= Count+2;
Real	1.0, -1.0E5	V = W / 5.3;
Time	7 ns, 100ps	Q <= '1' after 6 ns;
Character	'a', '2', '$'	CharData <= 'x';
String	(array of characters)	Msg <= "Error";

VHDL Data Types

- Bit
  - '0' or '1'
- Bit_Vector
  - "00", "01", "10", ...
- Boolean
  - FALSE or TRUE
- Time
  - integer with units
  - fs, ps, ns, us, ms, ...
- Integer
- Real
- Character
  - 'a', 'b', '1', '2', ...
- Enumeration Type
  - User defined

Data Types

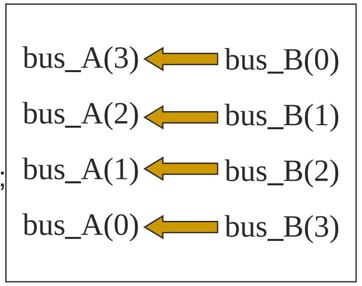
- Two main data types are:
- Scalar Types
  - integer, real, enumerated
    - e.g. type byte is range 255 downto 0;
    - type colors is (red, green, yellow); - (enumerated data type)
- Composite Types
  - arrays and records
    - arrays : regular structures consisting of elements of same type
    - user may define his own arrays or
    - use some predefined arrays e.g. bit_vector, string
    - Records: values of different types

Definition of Arrays

- **Collection of signals of the same type**

Predefined arrays :
- **bit_vector (array of bit)**
- **string (array of character)**
- Example:

signal bus_A : bit_vector(3 downto 0);
Signal bus_B: bit_vector(0 to 3);
bus_A <= bus_B



Types of Assignment for 'bit' Data Types

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

```
signal BUS_A :    bit_vector (3 downto 0);


BUS_A(3) <= '1';

BUS_A <= "0011";

BUS_A<=x"C"
```

Data Type

Type checking is strict!

```
architecture xyz of My_Entity is
      signal Count : integer;
      signal Data_bus : std_logic_vector(7 downto 0);
begin
      ….
      Count <= Data_bus;      WRONG!
      ….
```

Solution: *type conversion functions (later…)*

VHDL Operators

- Logical
  - and, or, nand, nor, xor
- Relational
  - =, /=, <, <=, >, >=
- Shift
  - sll, srl, sla, sra, rol, ror
- Addition
  - +, -
- Concatenation
  - &
- Unary Sign
  - +, -
- Multiplication
  - *, /, mod, rem
- Miscellaneous
  - not, abs, **
- …and other more complex functions included in libraries IEEE standard logic 1164 and IEEE standard logic arithmetic.

VHDL Operators

- "A+B" means "A add B" (A, B: integers, bits or bit-vectors, etc.). If you want logical "OR" operation, you should use "A or B".
- "A*B" means "A multiply B" (A, B: integers, bits or bit-vectors, etc.). If you want logical "AND" operation, use "A and B".
- Expression consist of operands and operators. Following is a list of VHDL operators:

priority

logical	not					
	And	or	Nand	nor	xor	Xnor
relational	=	/=	<	<=	>	>=
shift	Sll	srl	Sla	sra	rol	ror
arithmetic	+	-				
	*	/	mod	rem	**	abs

Concatenation operator: &

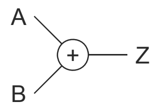
- The concatenation operator '&' is allowed on the right side of the signal assignment operator '<=', only.
- architecture CLASS1 of CONCAT is
  signal BYTE : bit_vector (7 downto 0);
  signal A_BUS, B_BUS : bit_vector (3 downto 0);
  begin

  BYTE <= A_BUS **&** B_BUS;

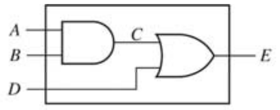
  end CLASS1;

Combinational Logic

```
entity ADD is
  port ( A, B : in  std_logic_vector( 7 downto 0);
         Z    : out std_logic_vector(15 downto 0));
end ADD;

architecture ARITHMETIC of ADD is
begin
    Z <= A + B;
end ARITHMETIC;
```



VHDL Modules

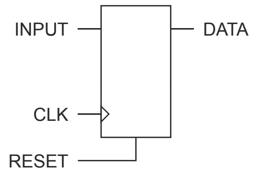


```
entity two_gates is
  port(A, B, D: in bit; E: out bit);
end two_gates;

architecture gates of two_gates is
signal C: bit;
begin
  C <= A and B; -- concurrent
  E <= C or D;  -- statements
end gates;
```

Sequential Logic

- A general term regarding design containing flip-flops, i.e. clocked
- Explicit reset is necessary to guarantee the design during power-up

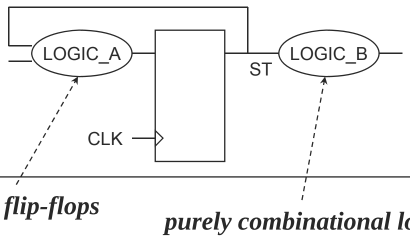
```
process (CLK,RESET)
begin
  if (RESET = `1`) then
    DATA <= `0` ;
  elsif (CLK`event and CLK=`1`) then
    DATA <= INPUT ;
  end if ;
end process ;
```



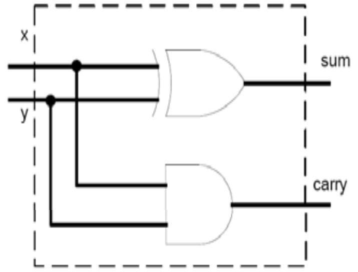
RTL: Combinational Logic and Registers

- Signal assignments in clocked processes infer flip-flops

```
LOGIC_A: process
begin
  wait until CLK`event and CLK=`1`;
  -- Logic A
end process LOGIC_A;

LOGIC_B: process (ST)
begin
  -- Logic B
end process LOGIC_B;
```

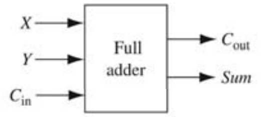


Half Adder



```
library ieee;
use  ieee.std_logic_1164.all;
entity half_adder is
port(
        x,y: in std_logic;
        sum, carry: out std_logic);
end half_adder;

architecture myadd of half_adder is
        begin
                sum <= x xor y;
                carry <= x and y;
end myadd;
```

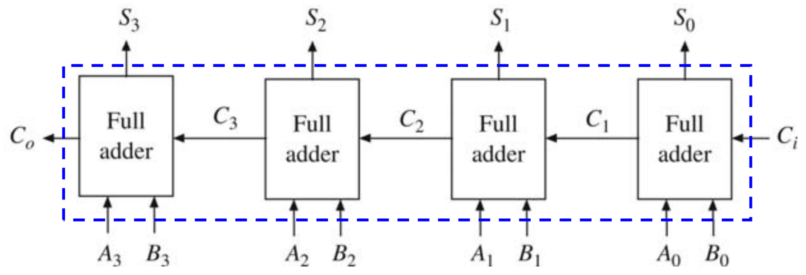
Full Adder (Dataflow)



```
entity FullAdder is
  port(X, Y, Cin: in bit;     --Inputs
          Cout, Sum: out bit);  --Outputs
end FullAdder;
```

```
architecture Dataflow of FullAdder is
begin        -- concurrent assignment statements
  Sum  <= X xor Y xor Cin after 2 ns;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin)
            after 2 ns;
end Dataflow;
```

4-bit Ripple-Carry Adder



```
entity Adder4 is
  port(A, B: in bit_vector(3 downto 0);
       Ci: in bit;
       S: out bit_vector(3 downto 0);
       Co: out bit);
end Adder4;
```

4-bit Adder (Structural)

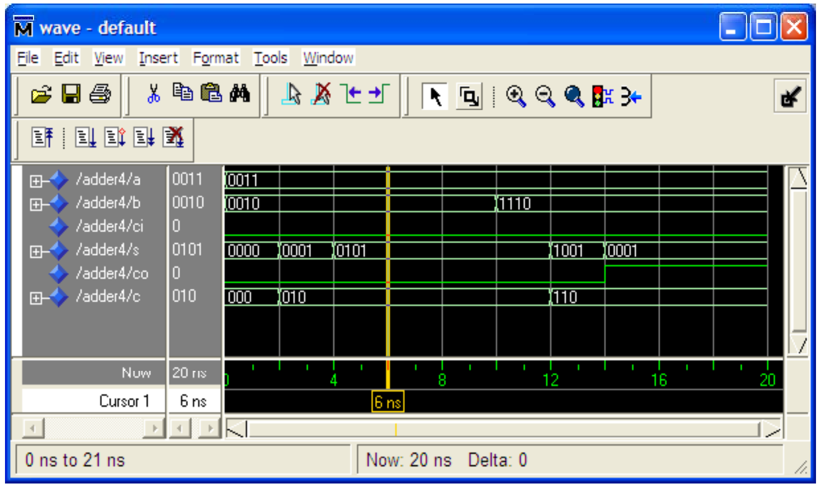
```
architecture Structure of Adder4 is
  component FullAdder
    port(X, Y, Cin: in bit;          -- Inputs
         Cout, Sum: out bit);        -- Outputs
  end component;
  signal C: bit_vector(3 downto 1); -- internal signal
begin     --instantiate four copies of the FullAdder
  FA0: FullAdder port map(A(0),B(0),Ci,C(1),S(0));
  FA1: FullAdder port map(A(1),B(1),C(1),C(2),S(1));
  FA2: FullAdder port map(A(2),B(2),C(2),C(3),S(2));
  FA3: FullAdder port map(A(3),B(3),C(3),Co,S(3));
end Structure;
```

ModelSim VHDL Simulation #1: Based on Commands (*.do file)

- Simulator Commands

```
add list A B Ci S Co C
force A 0011
force B 0010
force Ci 0
run 10 ns

force B 1110
run 10 ns
```

ModelSim VHDL Simulation (Waveforms)

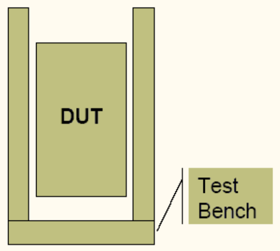


ModelSim VHDL Simulation #2: Based on Test Benches

- Testing a design by simulation
- Use a *test bench* model
  - an architecture body that includes an instance of the design under test
  - applies sequences of test values to inputs
  - monitors values on output signals
    - either using simulator
    - or with a process that verifies correct operation

Test Benches

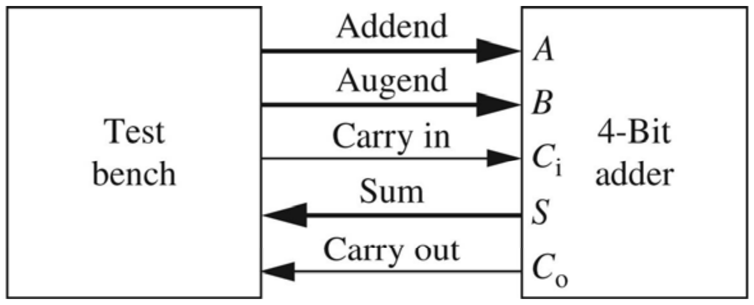
- VHDL can capture performance specification for a circuit, in the form of a test bench.
- Test benches are VHDL descriptions of circuit stimuli and corresponding expected outputs that verify the behavior of a circuit over time. Test benches should be an integral part of any VHDL project and should be created in tandem with other descriptions of the circuit.



Test Bench Example

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

4-bit Adder Test Bench



4-bit Adder Test Bench

```
entity TestAdder4 is
end TestAdder4;

architecture Test of TestAdder4 is

component Adder4
  port(A, B: in bit_vector(3 downto 0); Ci: in
   bit;
       S: out bit_vector(3 downto 0); Co: out
   bit);
end component;

signal addend, augend, sum: bit_vector(3 downto
   0);
signal cin, cout: bit;
```

4-bit Adder Test Bench

```
constant N: integer := 6;
type bv_arr is array(1 to N) of
               bit_vector(3 downto 0);
type bit_arr is array(1 to N) of bit;
constant addend_array: bv_arr :=
  ( "0011", "0011", "0011", "1101", "1110", "1110");
constant augend_array: bv_arr :=
  ( "0010", "1110", "1101", "0010", "1101", "1100");
constant cin_array: bit_arr :=
  (    '0',    '0',    '1',    '0',    '0',    '1');
constant cout_array: bit_arr :=
  ('0',    '1',    '1',    '0',    '1',    '1');
constant sum_array: bv_arr :=
  ( "0101", "0001", "0001", "1111", "1011", "1011");
```

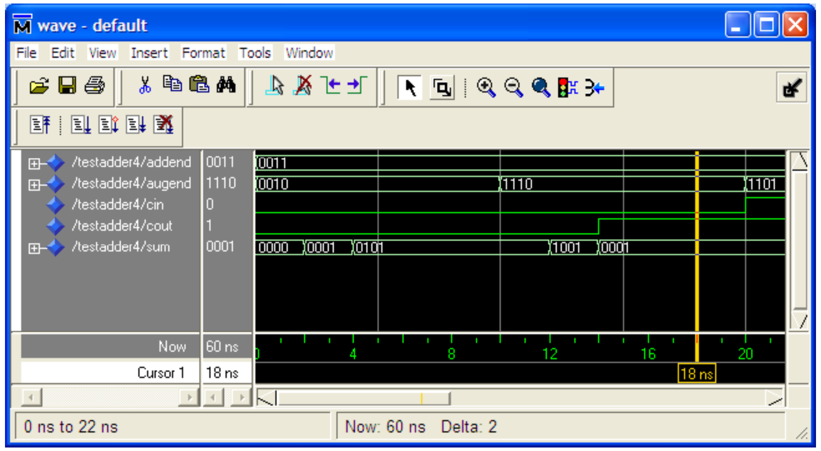
4-bit Adder Test Bench

```
begin
  add1: Adder4 port map (addend, augend, cin,
                        sum, cout);

  process
  begin
    for i in 1 to N loop
      addend <= addend_array(i);
      augend <= augend_array(i);
      cin <= cin_array(i);
      wait for 10 ns;
```

4-bit Adder Test Bench

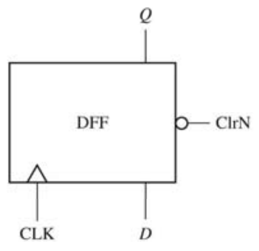
```
      assert (sum = sum_array(i) and
              cout = cout_array(i))
        report "Wrong Answer"
        severity error;
    end loop;
    report "Test Finished";
  end process;
end Test;
```

ModelSim VHDL Simulation: Based on Test Bench



VHDL Processes (Behavioral)

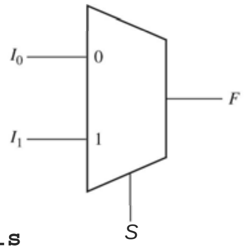
- D Flip-Flop with Asyncronous Clear



```
process(CLK, ClrN)
begin
  if CLRn = '0' then Q <= '0';
  else if CLK'event and CLK = '1'
    then Q <= D;
  end if;
  end if;
end process;
```

Multiplexers: 2-to-1

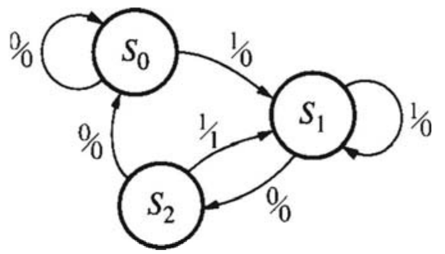
```
entity MUX2to1 is
  port(I1, I0, S: in bit;
       F: out bit);
end MUX2to1;

architecture Dataflow of MUX2to1 is
begin
  F <= I0 when S = '0' else I1;
end Dataflow;
```



Multiplexer: 4-to-1

```
entity MUX4to1 is
  port(I: in bit_vector(3 downto 0);
       S: in bit_vector(1 downto 0);
       F: out bit);
end MUX4to1;
architecture Dataflow of MUX4to1 is
begin
  with S select
    F <= I(0) when "00",
         I(1) when "01",
         I(2) when "10",
         I(3) when "11";
end Dataflow;
```

Sequential Machine



Present State	Next State		Present Output	
	$X = 0$	$X = 1$	$X = 0$	$X = 1$
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_1	0	1

Behavioral Model

```
entity Sequence_Detector is
  port(X, CLK: in bit;
       Z: out bit);
end Sequence_Detector;

architecture Behave of Sequence_Detector is
signal State: integer range 0 to 2 := 0;
begin
  process(CLK)
  begin
    if CLK'event and CLK = '1' then
```

Behavioral Model

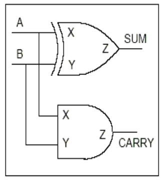
```
case State is
  when 0 =>
    if X = '0' then
      State <= 0;
    else
      State <= 1;
    end if;
  when 1 =>
    if X = '0' then
      State <= 2;
    else
      State <= 1;
    end if;
```

Behavioral Model

```
        when 2 =>
          if X = '0' then
            State <= 0;
          else
            State <= 1;
          end if;
      end case;
    end if;
  end process;
  Z <= '1' when (State = 2 and X = '1')
       else '0';
end Behave;
```

Exercise

❑ Exercise: Define the entity and architecture of a HALF ADDER where the two input signals are A and B, and the two outputs are SUM and CARRY.



Solution

```
Entity Half_Adder is
    port(A, B: in std_logic;
         SUM, CARRY: out std_logic);
end Half_Adder;
architecture Add of Half_Adder is
begin
    SUM <= A xor B;
    CARRY <= A and B;
end Add;
```

Concurrent statements

Another possible architecture

```
architecture Add2 of Half_Adder is
begin
    -- another solution (this is a comment!)
    SUM <= '1' when (A /= B) else '0';
    CARRY <= '1' when (A=B and A='1') else '0';
end Add2;
```

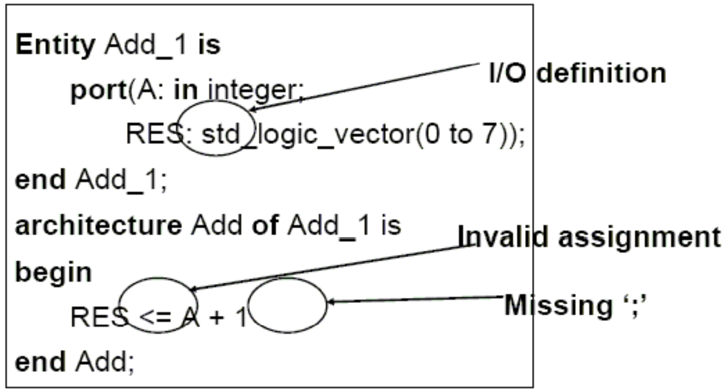
Exercise

Define a circuit with two integer inputs A and B. The output EQ is equal to '1' iff A=B+1.

```
Entity Compare_int is
      port(A, B: in integer;
            EQ: out std_logic);
end Compare_int;
architecture comp of Compare_int is
begin
      EQ<= '1' when (A=B+1) else '0';
end comp;
```

Exercise

Locate the errors present in this VHDL code:



```
Entity Add_1 is
      port(A: in integer;
            RES: std_logic_vector(0 to 7));
end Add_1;
architecture Add of Add_1 is
begin
      RES <= A + 1
end Add;
```

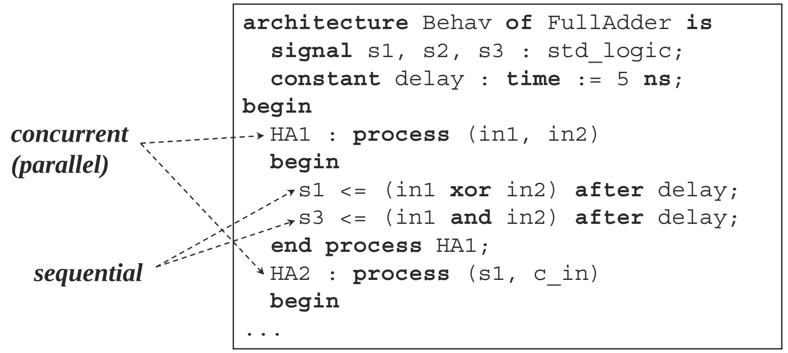
Processes

- Used in behavioral modeling that allows you to use sequential statements to describe the behavior of a system over time

[*process_label*:] process [(*sensitivity_list*)]
begin
list of sequential statements such as:
signal assignments
variable assignments
case statement
exit statement
if statement
loop statement
next statement
null statement
procedure call
wait statement
end process [*process_label*];

Process

- Statements within an architecture operate concurrently; statements within a process execute sequentially
- Processes themselves are concurrent statements



```
architecture Behav of FullAdder is
  signal s1, s2, s3 : std_logic;
  constant delay : time := 5 ns;
begin
  HA1 : process (in1, in2)
  begin
    s1 <= (in1 xor in2) after delay;
    s3 <= (in1 and in2) after delay;
  end process HA1;
  HA2 : process (s1, c_in)
  begin
...
```

Process Statement

- All statements in an architecture are concurrent
- Process statements exist with an architecture
- Process statements are concurrent
- Sequential statements exist only within process statements
- All statements in the process are executed when the process is invoked
- A process consists of a sensitivity list, a declarative part and a statement part:

```
name : process(sensitivity list)
              declarations;
          begin
               statements;
          end process name;
```

Process Execution

- A real physical system the logic is “always active”
- Processes behave similarly, after executing last statement they immediately go back to first statement
- Process execution is suspended by wait statements -
- Execution resumes when wait condition is met

```
Example:
     process
        begin
            statements;
            wait <condition>;
            statements;
            wait <condition>;
     end process;
```

```
Examples of wait statements:

wait until EN=‘1’;
wait for 50ns;
wait on a,b;
 - a and b are actually a
              sensitivity  list
```

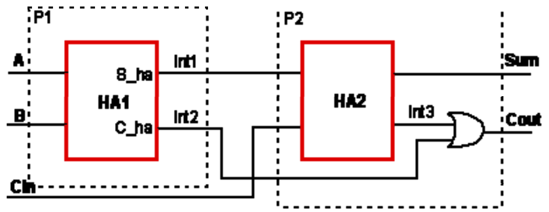
Process Execution

- All invoked processes are executed in parallel and the order in which they appear in the code is unimportant
- All processes are invoked at the start of a simulation
- If wait condition is the first statement, execution is immediately suspended
- If wait condition is last statement, the process is executed once then waits till condition is met
- Sensitivity list that appear in process statement i.e. process(a,b,c), are equivalent to “wait on” statement at end of process
- Process is invoked if a signal in the sensitivity list changes its value
- A process with no sensitivity list is re-invoked immediately after last statement is executed

Signals in Processes

- signals cannot be declared within a process
- signals are declared within an architecture and are recognized by all processes
- **signal assignments within a process, only take effect when process suspends, till then all signals retain their previous values**
- all signal assignments occur concurrently
- only last assignment of a signal is effective

Full Adder



- HA1
  - S_ha = (A xor B) = int1
  - C_ha = (A and B) = int2
- HA2
  - (A xor B) xor Cin = int1 xor Cin = Sum
  - (A xor B) and Cin = int 1 and Cin = int3
  - int2 or int3 = Cout

Full Adder – using Processes

```
library ieee;
    use ieee.std_logic_1164.all;
    entity FULL_ADDER is
            port (A, B, Cin : in std_logic;
            Sum, Cout : out std_logic);

end FULL_ADDER;

architecture BEHAV_FA of
    FULL_ADDER is
    signal int1, int2, int3: std_logic;
    begin
-- Process P1 that defines the first half
    adder
    P1: process (A, B)
            begin
                        int1<= A xor B;
                        int2<= A and B;
        end process;
```

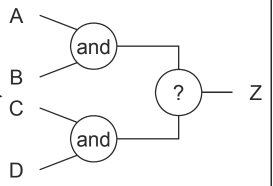
```
-- Process P2 that defines the
    second half adder and the
    OR -- gate
    P2: process (int1, int2, Cin)
        begin
                    Sum <= int1
    xor Cin;
                    int3 <= int1
    and Cin;
                    Cout <= int2 or
    int3;
        end process;
    end BEHAV_FA;
```

Concurrent vs. Sequential Execution

❑ Following two VHDL codes lead to different results for output Z

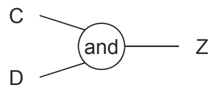
```
architecture CONCURRENT of MULTIPLE is
  signal Z, A, B, C, D: std_logic;
begin
  Z <= A and B;
  Z <= C and D;
end CONCURENT;
```

leads to Z=unknown



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

leads to Z=C*D



Concurrent Statements

- Executed at the same time, independent of statement order
- Signal assignment "<="
  - Left side receives a new value whenever the right side changes
- Conditional signal assignment

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

- Selected signal assignment

```
with EXPRESSION select
TARGET <= VALUE_1 when CHOICE_1,
          VALUE_2 when CHOICE_2 | CHOICE_3,
          VALUE_3 when CHOICE_4 to CHOICE_5,
          ...
          VALUE_n when others;
```

Block Statements

- A block statement provides a way to combine a group of concurrent statements together
- A group of statements can be placed under a guard
- FORMAT

  label: **block** (guard expression)
  -- declarative part
  begin
  -- statement part
  end block label
- A guard is a boolean expression that evaluates to true or false.
- Concurrent statements in block execute if guard is true

Sequential Statements

- Sequential statements can only exist with a process
- if , case , for loops are examples of sequential statements
- examples:

```
CASE sel IS
 WHEN "10" =>
    a :=  1;
  WHEN "01" =>
    a :=  2;
   WHEN OTHERS =>
    a := 3;
 END CASE;
```

```
IF set = '1' AND reset = '0' THEN
      q <= '0' ;
      qn <=  '1' ;
ELSIF set = '0' AND reset = '1' THEN
      q <= '1' ;
      qn <=  '0' ;
ELSIF set = '0' AND reset = '0' THEN
      q <= q ;
      qn <=  qn ;
ENDIF;
```

```
FOR I IN 0  TO 3 LOOP
    s(i) <= a(i) xor b(i);
END LOOP;
```

Sequential Statements

- Executed according to the order in which they appear
- Permitted only within processes, used to describe algorithms
- IF statement

```
if CONDITION then
   -- seq. statements
end if;
```

```
if CONDITION then
   -- seq. statements
else
   -- seq. statements
end if;
```

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

- CASE statement

```
case EXPRESSION is
    when VALUE_1 =>
      -- sequential statements
    when VALUE_2 | VALUE_3 =>
      -- sequential statements
    when VALUE_4 to VALUE_N =>
      -- sequential statements
    when others =>
      -- sequential statements
end case;
```

Sequential Statements (contd.)

- FOR loop
  - Loop variable
    - Declared implicitly
    - Local
    - Read only
    - Must be locally static if the loop is to be synthesized (must not depend on signal or variable values)

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

- WAIT statement
  - Stops process execution
  - Generally not synthesizable
  - An excellent tool in test bench and simulation

Wait for a specific time	`wait for specific_time;`
Wait for a signal event	`wait on signal_list;`
Wait for a true condition	`wait until condition;`
Wait indefinitely	`wait;`

Sequential Statements (contd.)

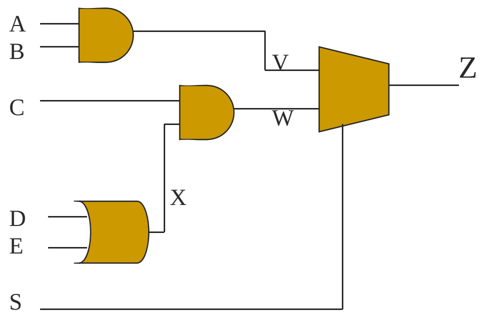
- Variables are available within processes
  - Named within process declarations
  - Known only in this process
- Immediate assignment
- Keep the last value
- Possible assignments
  - Signal to variable
  - Variable to signal
  - Types have to match

Signals in VHDL

- Signals carry information.
- Allow analysis of timing relationships in a VHDL system.
- Unlike variables in C, signals contain information both on current and previous values
- Signals can have different types, e.g. bit, bit_vector(0 to 7)
- External signals : signals which connect entity to outside world (i.e. ports of the entity).
  - Have mode associated with them:
    - IN mode : data can only be read from signal (port)
    - OUT mode : data can only be written to signal (port)
    - INOUT mode : data can be read and written to signal (port)
- Internal signals : signals which are not visible outside the architecture.
  - Declared in declarative part of architecture
  - Have no mode associated with them

Signal Assignment

- Y <= (A and B) or C -- simple assignment
- Y <= (A and B) or C after 10ns -- delayed assignment
- Y <= ‘0’ when a = b else ‘1’; -- conditional signal assignment



Write VHDL code for the schematic.
Write a statement for V,W,X and Z

Variables

- A method is needed for immediately storing temporary data within a process
- Immediate storage done with the aid of variables - variable assignments take effect immediately as they are sequential statements
- Variables can only be defined with a process and are not recognized outside the process
- Variable declaration very similar to signal declaration
  - e.g variable A : bit_vector(0 to 7);
- Variable assignment done with “:=“ e.g. A := “00110011”
- Signal has 3 properties : type, value and time, variable only has 2: type and value
- Signals and variables of same type can be assigned to each other

Variable example

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


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

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

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

Variables vs. Signals

- Signals
  - In a process, only the last signal assignment is carried out
  - Assigned when the process execution is suspended
  - “<=“ to indicate signal assignment
- Variables
  - Assigned immediately
  - The last value is kept
  - “:=“ to indicate variable assignment

Variables vs. Signals

- **Signal values are assigned when the process execution is suspended.**
- variable assignments take effect immediately.

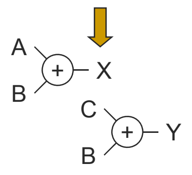
- **Only the last signal assignment is carried out.**
- variable immediate assignment

- **Signals are know in the architecture.**
- Variables are known only in the process

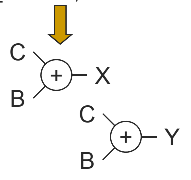
Variables vs. Signals (contd.)

- Depending on M and N are variables or signals, the following two VHDL codes lead to different results for X.
- ✓ Left code: A (B) is assigned to variables M (N) immediately → X=A+B.
- ✓ Right code: signal M takes the last assigned value of C → X=C+B.

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

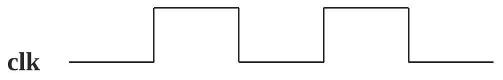


```
signal A, B, C, Y, Z : integer;
signal M, N : integer;
begin
  process (A, B, C, M, N)
  begin
    M <= A;
    N <= B;
    X <= M + N;
    M <= C;
    Y <= M + N;
  end process;
```



Asynchronous and Synchronous Processes

- It is good practice to separate logic into asynchronous logic and asynchronous logic



if clock’event and clock = ‘1’ then
   means : if the clock rises from ‘0’ to ‘1’ then …
if clock’event and clock = ‘0’ then
   means : if the clock falls from ‘1’ to ‘0’ then

• These statements cause the creation of edge triggered flipflops during the synthesis process
• No need to add “else” statements to the above “if” statements
• **Never add other conditions to above if statements. Instead of:**
if clock’event and clock = ‘1’ and EN = ‘1’ then

do: **if clock’event and clock = ‘1’ then**
if EN = ‘1’ then

Asynchronous and Synchronous Processes

```
ARCHITECTURE arc_shifter OF shifter
   IS
  SIGNAL shift_val :bit4;
BEGIN
  nxt: PROCESS(load, left_right, din,
    dout)
 BEGIN
   IF (load = ’1’) THEN
     shift_val <= din;
    ELSIF (left_right = ’0’) THEN
      shift_val(2 downto 0) <= dout(3
    downto 1);
      shift_val(3) <= ’0’;
    ELSE
      shift_val(3 downto 1) <= dout(2
    downto 0);
     shift_val(0) <= ’0’;
    END IF;
 END PROCESS;
```

```
current: PROCESS
  BEGIN
    WAIT UNTIL clock’EVENT and clock = ’1’;
    dout <= shift_val;
  END PROCESS;
END arc_shifter;
```

Predefined Signal Attributes

- VHDL provides several predefined attributes which provide information about the signal
- ✓ signal_name’ACTIVE: indicates if a transaction has occurred
- ✓ signal_name’QUITE: indicates that transaction has not occurred
- ✓ signal_name’EVENT : If an event has occurred on signal_name
- ✓ signal_name’STABLE: If an event has not occurred
- ✓ signal_name’LAST_EVENT: Time elapsed since last event has occurred
- ✓ signal_name’DELAYED(T): A signal identical to signal_name but delayed by T units of type TIME;

Configurations

- Used by simulator to bind component instance to entity-architecture pair.
- All designs simulated should have configurations.
- If no instantiated components appear in architecture, configuration is empty

Example 1: empty configuration

```
CONFIGURATION cfg1_rsff OF rsff is
      FOR arc_rsff
      END FOR;
END cfg1_rsff;
```

Example 2:

```
CONFIGURATION cfg2_rsff OF rsff is
      FOR arc2_rsff
        FOR  U1,U2 : nand2 USE ENTITY
                WORK.nand2(arc_nand2);
      END FOR;
END cfg1_rsff;
```

Use of Packages in VHDL

- A VHDL package is simply a way of grouping a collection of related declarations that serve a common purpose
- Can be reused by other designs

```
package identifier is
  {package declaration}
end package identifier;
```

Packages

- Allow user to define elements that are not included in the standard VHDL language.
- A collection of commonly used data types and sub-programs used in a design.
- Packages are defined in two parts:
  - package declaration : includes declaration of all elements defined by the package.
  - package body : includes the implementation of all elements declared in the package.
- Not all packages have bodies, sometimes body not required.

```
PACKAGE days_package IS
   TYPE day_t IS (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday);
END days_package;
```

Predefined Packages

- The most popular packages in VHDL are defined by IEEE.
- Standard : contains all basic declarations and definitions, always included by default.
- Std_logic_1164 : contains many useful language extensions.
- Textio : Contains definitions of all operations on texts.
- To use a the std_logic_1164 package in a design unit, include the following statements:

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

For the previous **user defined** package example use :

```
use WORK.days_package.all;
```

Predefined Packages

- The predefined types in VHDL are stored in a library “std’
- Each design unit is automatically preceded by the following context clause

library std, work; **use** std.standard.all;

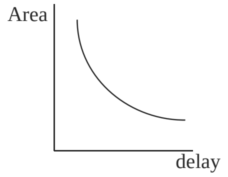
```
package standard is
type boolean is (false, true);  -- defined for operators =, <=,
  >=, ..
type bit is (‘0’, ‘1’); -- defined  for logic operations and, or,
  not…
type character is (..);
type integer is range IMPLEMENTATION_DEFINED;
subtype natural is integer range 0 to integer’high;
type bit_vector is array(natural range <>) of bit;
…
end package standard;
```

VHDL Libraries

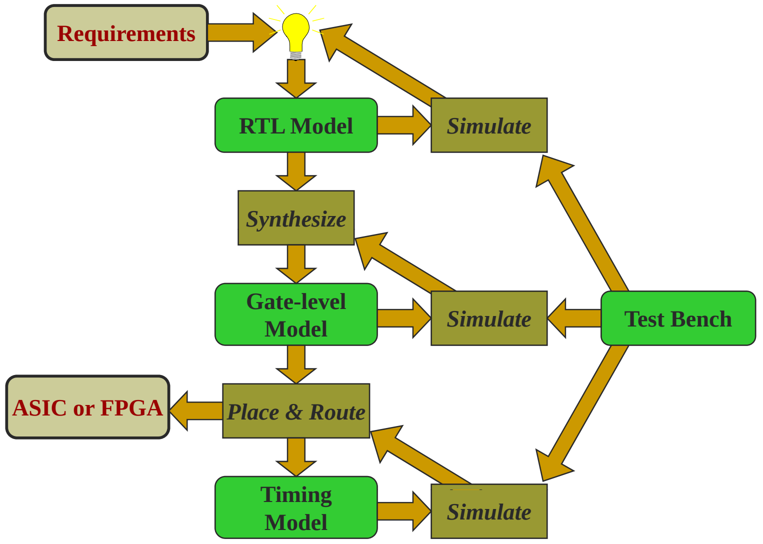
- library IEEE;
- use IEEE.numeric_bit.all;
  - Types signed and unsigned
  - Overloaded operators for signed and unsigned
- library IEEE;
- use IEEE.std_logic_1164.all;
- use IEEE.numeric_std.all;
- use IEEE.std_logic_unsigned.all;
  - Types std_logic and std_logic_vector
  - Overloaded operators for signed and unsigned

Use of VHDL in Synthesis

- Translates register-transfer-level (RTL) design into gate-level netlist
- VHDL was initially developed as a language for SIMULATION
- Recently being used as a language for hardware synthesis from logic synthesis companies
  - Synopsys Design Compiler, Ambit BuildGates, Mentor Graphics Autologic, ..
- Synthesis tools take a VHDL design at behavioral or structural level and generate a logic netlist
  - Minimize number of gates, delay, power, etc.



Basic Design Methodology


Requirements
RTL Model
Simulate
Synthesize
Gate-level Model
Simulate
Test Bench
ASIC or FPGA
Place & Route
Timing Model
Simulate