Top-Leve Entity in a simulateable model

- Is usually "empty", i.e. has no ports. Its architecture is usually the "test bench":

```
entity TB_DISPLAY is
end TB_DISPLAY;

architecture TEST of TB_DISPLAY is
-- signal declarations
-- component declaration(s)
begin
-- component instance(s)
-- test processes
end TEST;
```

Modeling

- VHDL modules is divided into two parts
  - External view: describes the interface
    - Entity declaration
  - Internal view: describes module internal implementation
    - Architecture declaration

Entity Declarations

- Syntax

  entity entity_name **is**
  [**generic** (generic_list);]
  [**port** (port_list);]
  end entity_name;

- The port list must define the name, the mode (i.e. direction) and the type of each port on the **entity**:

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

entity COUNTER is
port (CLK  : in std_ulogic;
      RESET: in std_ulogic;
      Q    : out integer
             range 0 to 15);
end COUNTER;
```

VHDL Basic Modeling Constructs

VLSI CAD
Case Western Reserve University

Dan Saab

Generics

- By declaring generics of type time, delays may be programmed on an instance-by-instance basis. Generics may be given a default value, in case a value is not supplied for all instances:

```
entity AN2_GENERIC is
  generic (DELAY: time := 1.0 ns);
  port    (A,B  : in std_ulogic;
           Z  : out std_ulogic);
end AN2_GENERIC;

architecture BEH of AN2_GENERIC is
begin
  Z <= A and B after DELAY;
end A;
```

Entity Declarations

- Generics must be declared before the ports. They do not have a mode, they can only pass information into the entity:

```
entity AN2_GENERIC is
  generic (DELAY: time := 1.0 ns);
  port (A,B : in std_ulogic := '1';
        Z : out std_ulogic);
end AN2_GENERIC;
```

Generics

- pass specific information into an entity.

```
entity PARITY is
  generic (N : integer);
  port    (A : in std_ulogic_vector (N-1 downto 0);
        ODD : out std_ulogic);
end PARITY;
```

- An instance of a component with generics, has a generic map declared before the port map

```
U1: PARITY
  generic map (N   => 8)
  port map    (A   => DATA_BYTE,
               ODD => PARITY_BYTE);
```

Entity Declarations

- Each entity port acts like a signal
  - visible in the architecture(s) of the entity.
  - The mode (i.e.direction) of each port determines whether it may be read from or written to in the architecture:

Mode	Read in Arch	Write in Arch
In	Yes	No
Out	No	Yes
Inout	Yes	Yes
Buffer	Yes	Yes

Signal declaration

- When we need to provide internal signals in an architecture body, we must define them using signal declarations
- The syntax for a signal declaration is very similar to that for a variable declaration:

signal identifier:subtype_indication [:= expression];

  - They can be used by processes within the architecture body but are not accessible outside

Architecture

- The internal operation of a module is described by an architecture body.
  - These operations are applied to values on input ports
  - generates values to be assigned to output ports.
- Syntax:

```
architecture identifier of entityname is
block declarative_item;
begin
        concurrent statement
end architecture identifier ;
```

- Entity name specifies which module has its operation described by this architecture body.

Declarations

- Declarations may typically be any of the following:
  - type, subtype,
  - signal, constant,
  - file, alias,
  - component, attribute,
  - function, procedure,
  - configuration specification.
- Items declared in an architecture are visible in any process or block within it.

```
architecture TB of TB_CPU is
    component CPU_IF
        port   -- port list
    end component;
    signal CPU_DATA_VALID: std_ulogic;
    signal CLK, RESET: std_ulogic := '0';
    constant PERIOD : time := 10 ns;
    constant MAX_SIM: time := 50 *
PERIOD;
begin
        -- concurrent statements
end TB;
```

Entity Declarations

- An entity may also contain declarations. Items declared are visible within the arhitecture(s) of the entity.

```
entity ROM is
        port ( address: in std ulogic veotor(14 downto 0);
                data : out std ulogic vector(7 downto 0);
        enable : in std ulogic);

        subtype inst_byte is bit vector(7 downto 0);
        type romst is array (0 to 2**14 - 1) of inst_byte;
        constant program : romst :=( X"32", X"3F", X"03", -- LDA 3F03
                                     X"71", X"23", -- BLT 23
                                     … );
end entity ROM;
```

Signal declaration

- A signal which is driven by more than one process, concurrent statement or component instance, must be declared with a resolved type, e.g. std_logic or std_logic_vector:

```
architecture COND of TRI_STATE is
  signal TRI_BIT: std_logic;
begin
  TRI_BIT <= BIT_1 when EN_1 = '1'
       else 'Z';
  TRI_BIT <= BIT_2 when EN_2 = '1'
       else 'Z';
end COND;
```

Signal declaration

```
entity acell is
        port ( al, a2, bl, b2 : in bit := '1'; y : out bit);
end entity acell;

architecture primitive of acell is
signal z1, z2: bit;
signal z3: bit;
begin
andgatea: process (al, a2) is
begin
        z1 <= al and a2;
end process andgatea;
andgateb process (bl, b2) is
begin
        z2 <= bl and b2;
end process andgateb;
orgate: process (z1, z2) is
begin
        z3 <= z1 or z2;
end process orgate;
inv: process (z3) is
begin
        y<=not z3;
end process inv;
end architecture primitive;
```

```
entity acell is
        port ( al, a2, b1, b2 : in bit := '1'; y : out bit);
end entity acell;

architecture primitive of acell is
signal z1, z2: bit;
signal z3: bit;
begin
        z1 <= a1 and a2;
        z2 <= b1 and b2;
        z3 <= z1 or z2;
        y<=not z3;
end architecture primitive;
```

Signal declaration

- During elaboration, each signal is set to an initial value. If a signal is not given an explicit initial value, it will default to the leftmost value ('left) of its declared type:

```
signal I : integer range 0 to 3;
-- I will initialise to 0
signal X : std_logic;
-- X will initialise to 'U'
```

Concurrent Statements

- They describe the module's operation.
- They can be activated and perform their actions concurrently.
  - Order is not important
- Example:

```
architecture EX1 of CONC is
signal Z, A, B, C, D : integer;
begin
        D <= A + B;
        Z <= C + D;
end EX1;
```

Equivalent

```
architecture EX2 of CONC is
signal Z, A, B, C, D : integer;
begin
        Z <= C + D;
        D <= A + B;
end EX2;
```

- **The following code checks that an active low static ram write strobe (ramwe) pulse meets the minimum width requirements and will report any violation**:

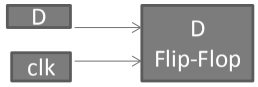
```
ram_we_check : PROCESS
 BEGIN

       -- wait until end of write pulse
       WAIT UNTIL (ramwe'EVENT AND ramwe='1');

        -- check ramwe stable for time Twe
        IF (NOT(ramwe'DELAYED'STABLE(Twe))) THEN
              ASSERT false REPORT "RAM: Min write strobe width violation"
              SEVERITY error;
        END IF;

END PROCESS ram_we_check;
```

Note that the ramwe signal is not directly tested for stability, instead the 'DELAYED attribute is used to delay it by 1 delta period, otherwise the 'STABLE attribute always returns false since there is an event on ramwe in the current delta period.

Signal Attributes:Signal S; interval T:

- S'active
  - True if there is a transaction on S in the current simulation cycle, false otherwise.
- S'last_event
  - Time interval since last event on S.
- S'last_active
  - Time interval since last transaction on S.
- S'last_value
  - S value just before last event on S.

Example:Setup time check




```
signal clk std ulogic;
...
if clk'event and (clk = '1' or clk = 'H')
   and (clk'last_value = '0' or clk'last_value = 'L') then
   assert d'last_event >= Tsetuptime
     report "Timing error: d changed within setup time of clk";
end if;
```

- How can you check the pulse width?
- How can you check the clock frequency?

Signal Attributes:Signal S; interval T:

- S'delayed(T)
  - A signal that takes on the same values as S but is delayed by time T.
- S'stable(T)
  - A Boolean signal that is true if there has been no event on S in the time interval T up to the current time, otherwise false.
- S'quiet(T)
  - A Boolean signal that is true if there has been no transaction on S in the time interval T up to the current time, otherwise false.
- S'transaction
  - A signal of type bit that changes value from '0' to '1' or vice versa each time there is a transaction on S.
- S'event
  - if there is an event on S in the current simulation cycle, false otherwise.

Architecture

- An architecture can contain any mix of component instances, processes or other concurrent statements:

```
architecture TEST of TB_DFF is
component DFF port (CLK, D: in  std_ulogic;
                   Q    : out std_ulogic);
end component;
signal CLK, D, Q : std_ulogic := '0';
begin
        UUT: DFF port map (CLK, D, Q);
        CLK <= not (CLK) after 25 ns;
        STIMULUS: process
        begin
                wait for 50 ns;
                D <= '1';
                wait for 100 ns;
                D <= '0';
                wait for 50 ns;
        end process STIMULUS;
end TEST;
```

Flip-flop data setup

```
flip_flop : PROCESS (clk)
 BEGIN
        IF (clk'EVENT AND clk='1') THEN   -- on clock rising edge..

         -- check d has been stable for time Tsu
                IF (d'STABLE(Tsu)) THEN
                        q <= d;                 --..set q output to d..
                ELSE
                        q <= 'X';               -- ..otherwise, set q unknown
                        ASSERT false REPORT "FF: Data setup violation"
                        SEVERITY error;
                END IF;

        ELSE
                q <= q;                 -- ..otherwise, no change in q
        END IF;
 END PROCESS flip_flop;
```

Flip-flop data hold

```
flip_flop : PROCESS
 BEGIN
        -- wait until a clock rising edge
        WAIT UNTIL (clk'EVENT AND clk='1');
        WAIT FOR (Thld);                -- wait hold time
        -- check d has been stable for time Thld

        IF (d'STABLE(Thld)) THEN
                q <= d;                 -- ..set q output to d..
        ELSE
                q <= 'X';               -- ..otherwise, set q unknown
                ASSERT false REPORT "FF: Data hold violation"
                SEVERITY error;
        END IF;

END PROCESS flip_flop;
```

Flip-flop

```
flip_flop : PROCESS (clk)
 BEGIN
        IF (clk'EVENT AND clk='1') THEN  -- on clock rising edge..
                q <= d;                 -- ..set q output..
        ELSE
                q <= q;                 -- ..otherwise, no change in q
        END IF;
END PROCESS flip_flop;
```

Clock generation

```
clock: process (clk) is
begin
   if clk = '0' then
      clk <= '1' after
      Tpw, '0' after
      2*Tpw;
   end if;
end process clock;
```

More than one signal assignment statement for a given signal

```
mux:process (a, b, sel) is
begin
    case sel is
    when '0' => z <= a after
    delay1;
    when '1' =>z <= b after
    delay2;
    end case;
end process mux;
```

Signal Assignment

- Example:

```
-- signal y is to take on the new value at a time 5 ns
-- later than that at which the statement executes.
y<= not z1 after 5 ns;
```

- if this assignment is executed at time 50 ns, and z1 has the value '1' at that time, then the signal y will take on the value '0' at time 55 ns.
- The execution of assignment schedules a transaction for the signaly which consists of the new value ('0') and the simulation time (55ns) at which it is to be applied.
  - Many transactions may be scheduled for a signal.

Signal declaration

- This signal assignment schedule a number of transactions for signal clk
- Delays are measured from the current time

  clk <= '1' after Tpw, '0' after 2*Tpw;
- Assume
  - statement execute at time 40 ns
  - value of clk at time 40 ns is '0'
  - Tpw is a contant 10 ns
- Two transactions are scheduled:
  - one for time 50 ns to set clk to '1',
  - and a second for time 60 ns to set clk to '0'.

Signal Assignment

- Provides a new value for a signal
  - The new value is determined by evaluating an expression
  - Determine when the signal take on the new value (when events occur over time)
    - This is fundamental to modeling hardware
- Syntax:

```
label : name <= delaymechanism waveform;
...
waveform (expression [after time expression] )  { ,... }
```

Examples

An Adder:

```
halfadd : process is
begin
      sum <= a xor b after Tpd;
      carry <= a and b after Tpd;
      wait on a, b;
end process halfadd;
```

Equivalent Adder:

```
halfadd : process (a, b) is
begin
      sum <= a xor b after Tpd;
      carry <= a and b after Tpd;
end process halfadd;
```

DFF: Example

```
entity DFF is
port ( D : in bit; clk: in bit; clr: in bit; Q : out bit);
end entity DFF;
architecture behavioral of DFF is
begin
      State_change : process (clk, clr) is
      begin
            if clr= '1' then
                  Q <= '0' after 2 ns;
            elsif clk'event and clk = '1' then
                  Q <= D after 2 ns;
            end if;
      end process state_change;
end architecture behavioral;
```

Wait Statements

- specify when processes respond to changes in signal values
- wait statement causes a process to suspend execution.
  - The sensitivity clause, condition clause and timeout clause specify when the process is subsequently to resume execution.
- A wait statement is a sequential statement with the following syntax rule:

```
label: wait on signal name
            until BooleanExpression
            for TimeExpression
```

Delta delays

- Executing a signal assigment statement without delay specification
  - the signal value does not change as soon as the signal assignment statement is executed.
  - the assignment schedules a transaction for the signal, which is applied after the process suspends.
- Process does not see the effect of the assignment until the next time it resumes.
  - No zero delay in VHDL

```
A <= '1';
...
if A ='1' then
```

Next time: Delay Mechinsm

wait

condition clause in a wait

```
clock:process is
Begin
   clk <= ’1’ after Tpw, ’0’ after 2*Tpw;
   wait until clk = ’0’;
end process clock;

clock: process is
Begin
   clk <= ’1’ after Tpw, ’0’ after 2*Tpw;
   wait for 2*Tpw;
end process clock;
```

Wait can have sensitivity as well as condition

```
-- condition (reset=’0’) is only tested when
-- an event  occurs on clk
wait on clk until reset = ’0’;

-- suspend until trigger changes to ’1’ or
-- until 1 ms of simulation time has elapsed
wait until trigger = ’1’ for 1 ms;
```

Wait suspending a process for the remainder of the simulation

```
testbench: process is
begin
        test0 <= ’0’ after 1O ns, ’1’ after 20 ns,
                  ’0’ after 30 ns, ’1’ after 40 ns;
        test1 <= ’0’ after 1O ns, ’1’ after 30 ns;
wait;
end process testbench;
```

Wait: Change sensitivity variables

```
entity mux2 is
            port ( a, b, sel in bit;z out bit);
end entity mux2;
architecture behavioral of mux2 is
constant propdelay time := 2 ns;
begin
      slick mux process is
      begin
            case sells
                  when ’0’ =>
                  z <= a after propdelay;
                  wait on sel, a;
                  when ’1’ =>
                  z <= b after propdelay;
                  wait on sel, b;
            end case;
      end process slick mux
end architecture behavioral;
```