TOP-DOWN DESIGN AND VHDL BASICS

* Supports and stimulates top-down design (separate declarations for communication and content)

* Communication: entity declaration.
* Content: architecture declaration; multiple architectures can be associated with one entity.
* A configuration declaration specifies which of the alternative architectures are actually selected for some design task (mostly simulation).

* Only one architecture per entity allowed (and no configuration).
* Support for hierarchical descriptions (in one file or multiple files).
* Multiple processes in one architecture are allowed (good practice: separate combinational and sequential logic).
* Timing through synchronization with clock (no references to absolute time).
* Top-level I/O always of type std_logic or std_logic_vector.

VHDL SYNTHESIS WITH SYNOPSYS

IEEE standards: packages for synthesizable data types already available; standards for synthesizable VHDL subsets are in preparation. General rules:

* Multiple processes in one architecture are allowed (good practice: separate combinational and sequential logic).
* Timing through synchronization with clock (no references to absolute time).
* Top-level I/O always of type std_logic or std_logic_vector.

THE TESTBENCH CONCEPT

Write stimuli and simulation output processing in VHDL (portability).

* Verify all intermediate designs (presynthesis and postsynthesis) with the same testbench.
* The entire system (testbench + DUT) is a configuration!
MORE ON COMBINATIONAL LOGIC

* Alternative to truth-table specification: if-then-else constructs.
* Important issues:
  + process should be sensitive to all inputs;
  + specify assignment to outputs for all possible input value combinations (latches will be inserted otherwise!)
  + think of specifying the *don't care* set to save logic.
* Word-level descriptions are supported.

WORD-LEVEL COMBINATIONAL LOGIC EXAMPLE: CONDITIONAL XOR

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity cond_xor is
  port (a, b, c: in std_logic; result: out std_logic_vector(1 downto 0));
end cond_xor;
architecture behavioral of cond_xor is
begin
  react: process (a, b, c)
  begin
    if c = '1' then
      result <= a xor b;
    else
      result <= (not a) xor b;
    end if;
  end process react;
end behavioral;
```

YOU GET WHAT YOU WRITE

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

if (x = '1') then
  y := a+b;
else
  y := c+d;
end if.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

if (x = '1') then
  op1 := a;
  op2 := b;
else
  op1 := c;
  op2 := d;
end if;
y := op1 + op2;

SEQUENTIAL LOGIC SPECIFICATION

* Only clocked logic is discussed.
* A sequential process is sensitive to a clock signal only.
* Clocked signals only change on the *rising edge* of the clock.
STATE MACHINE EXAMPLE (I)

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity gcd_controller is
    port (in_valid, reset, equal, greater, clock: in std_logic;
          sel1_2, sel3_4, enable1, enable2, ready: out std_logic);
end gcd_controller;
architecture synthesizable of gcd_controller is
    type state is
        (start, read_input, compare, finished, greater1, greater2);
    signal current_state: state;
    signal compare_results: std_logic_vector(1 downto 0);
begin
    new_state: process (clock)
    begin
        ...
    end process new_state;
    outputs: process(current_state)
    begin
        compare_results <= equal & greater;
    end process outputs;
end synthesizable;
```

STATE MACHINE EXAMPLE (II)

```vhdl
new_state: process (clock)
begin
    if rising_edge(clock) then
        if reset = '1' then
            current_state <= start;
        else
            case current_state is
                when start =>
                    if in_valid = '1' then
                        current_state <= read_input;
                    else
                        current_state <= current_state;
                    end if;
                when read_input =>
                    current_state <= compare;
                end case;
        end if;
    end if;
end process new_state;
```

STATE MACHINE EXAMPLE (III)

```vhdl
outputs: process(current_state)
begin
    compare_results <= equal & greater;
    case current_state is
        when start =>
            sel2_2 <= '-'; sel3_4 <= '-'; enable1 <= '0'; enable2 <= '0'; ready <= '0';
        when read_input =>
            sel2_2 <= '0'; sel3_4 <= '-'; enable1 <= '1'; enable2 <= '1'; ready <= '0';
        end case;
    end process outputs;
```

DATA TYPES FOR VHDL SYNTHESIS

The data types supported by Synopsys include:

* **integer** (default 32 bits, limited range can be declared, e.g. signal a: integer range -5 to 10);
* **std_logic** and **std_logic_vector** (vector of std_logic);
* **unsigned** and **signed** (vector of std_logic; for arithmetic);
* new data types based on the above, e.g. an array of std_logic_vector.
* be aware of the distinction between the data type **boolean** (required e.g. in the condition of an if statement) and **std_logic**.
FUNCTIONS FOR VHDL SYNTHESIS

* Functions (‘+’, and, rising_edge, etc.) supported by a VHDL synthesis tool are normally collected into packages.
* The descriptions in these packages provide an appropriate model for use in presynthesis simulation.
* The power of the synthesis tool lies in the generation of appropriate lower-level hardware for some function call.
* Conversion functions convert types at the language level; nothing happens in hardware. Example: a <= to_integer(b).
* Sometimes a type cast is sufficient. Example: a <= unsigned(b); b <= std_logic_vector(a).

INTEGERS VS. STD_LOGIC_VECTOR

For arithmetic, you can choose between the integer and unsigned data types.
* Results after synthesis will be the same.
* Choose description that results in best readability.

Example: suppose that a, b and c are integer variables and d, e and f unsigned variables. You can write:
  * a := b + c;
  * a := b + 1;
  * d := e + f;
  * d := e + 1;
  * a := to_integer(d);
  * d := to_unsigned(a,5);

MEMORY EXAMPLE (FRAGMENT)

```vhdl
architecture behavioral of shift_in is
  type memory is array (1 to 10) of unsigned (7 downto 0);
  signal local_memory: memory;
begin
  shift: process (clock)
  variable counter: integer range 1 to 10;
  begin
    if rising_edge(clock) then
      if (reset = '1') then
        for counter in 1 to 10 loop
          local_memory(counter) <= to_unsigned(0, 8);
        end loop;
      else
        if (read_mode = '1') then
          for counter in 2 to 10 loop
            local_memory(counter) <= local_memory(counter - 1);
          end loop;
        end if;
      end if;
    end if;
  end process;
end architecture;
```

THE VHDL SYNTHESIS DESIGN FLOW

* Always use the same testbench!
* The possibility to guide Synopsys should be used wherever needed (e.g. for maximal speed rather than minimal area; indicate critical timing).

1. Write/modify presynthesis VHDL.
2. Simulate presynthesis VHDL; if OK, continue, else go to 1.
3. Run Synopsys; if “code is synthesizable” and performance constraints are met, continue, else go to 1.
4. Simulate postsynthesis VHDL; if OK, continue, else go to 1.