1. VHDL Process

- Contains a set of sequential statements to be executed sequentially
- The whole process is a concurrent statement
- Can be interpreted as a circuit part enclosed inside of a black box
- May or may not be able to be mapped to physical hardware

A process with a sensitivity list

- Syntax
  ```vhdl
  process(sensitivity_list)
  declarations;
  begin
    sequential statement;
    sequential statement;
    ...
  end process;
  ```

- Two types of process
  - A process with a sensitivity list
  - A process with wait statement

- A process is like a circuit part, which can be – active (known *activated*)
  – inactive (known as *suspended*).
- A process is activated when a signal in the sensitivity list changes its value
- Its statements will be executed sequentially until the end of the process
• E.g., 3-input and circuit
  
  signal a,b,c,y: std_logic;
  process(a,b,c)
  begin
    y <= a and b and c;
  end process;

• How to interpret this:
  process(a)
  begin
    y <= a and b and c;
  end process;

• For a combinational circuit, all input should be included in the sensitivity list

A process with wait statement

• Process has no sensitivity list
• Process continues the execution until a wait statement is reached and then suspended
• Forms of wait statement:
  – wait on signals;
  – wait until boolean_expression;
  – wait for time_expression;

2. Sequential signal assignment statement

• Syntax
  signal_name <= value_expression;

• Syntax is identical to the simple concurrent signal assignment
• Caution:
  – Inside a process, a signal can be assigned multiple times, but only the last assignment takes effect

3. Variable assignment statement

• Syntax
  variable_name := value_expression;

• Assignment takes effect immediately
• No time dimension (i.e., no delay)
• Behave like variables in C
• Difficult to map to hardware (depending on context)
• E.g.,
  
  ```vhdl
  process(a,b,c)
  variable tmp: std_logic;
  begin
    tmp := '0';
    tmp := tmp or a;
    tmp := tmp or b;
    y <= tmp;
  end process;
  ```

• What happens if signal is used?

  ```vhdl
  process(a,b,c,tmp)
  begin
    -- tmpentry := tmp
    tmp <= '0';            -- tmpexit := '0';
    tmp <= tmp or a;  -- tmpexit := tmpentry or a;
    tmp <= tmp or b;  -- tmpexit := tmpentry or b;
  end process;
  ```

• Same as:

  ```vhdl
  process(a,b,c,tmp)
  begin
    tmp <= tmp or b;
  end process;
  ```

4. IF statement

• Syntax

```vhdl
if boolean_expr_1 then
  sequential_statements;
elsif boolean_expr_2 then
  sequential_statements;
elsif boolean_expr_3 then
  sequential_statements;
else
  sequential_statements;
end if;
```

• Examples

• Comparison to conditional signal assignment

• Incomplete branch and incomplete signal assignment

• Conceptual Implementation

E.g., 4-to-1 mux

```vhdl
architecture if_arch of mux4 is
begin
  process(a,b,c,d,e)
  begin
    if (a="00") then
      x <= b;
    elsif (a="01") then
      x <= c;
    elsif (a="10") then
      x <= d;
    else
      x <= e;
    end if;
  end process;
end if_arch;
```
E.g., 2-to-2\textsuperscript{2} binary decoder

```vhdl
architecture if_arch of decoder4 is
begin
  process (a)
  begin
    if (a = "00") then
      x <= "0001";
    elsif (a = "01") then
      x <= "0010";
    elsif (a = "10") then
      x <= "0100";
    else
      x <= "1000";
    end if;
  end process;
end if_arch;
```

E.g., 4-to-2 priority encoder

```vhdl
architecture if_arch of prio_encoder42 is
begin
  process (r)
  begin
    if (r(3) = '1') then
      code <= "11";
    elsif (r(2) = '1') then
      code <= "10";
    elsif (r(1) = '1') then
      code <= "01";
    else
      code <= "00";
    end if;
  end process;
active <= r(3) or r(2) or r(1) or r(0);
end if_arch;
```

Comparison to conditional signal assignment

- Two statements are the same if there is only one output signal in if statement
- If statement is more flexible
- Sequential statements can be used in then, elsif and else branches:
  - Multiple statements
  - Nested if statements

E.g., find the max of a, b, c

```vhdl
if (a > b) then
  if (a > c) then
    max <= a;  -- a>b and a>c
  else
    max <= c;  -- a>b and c>a
  end if;
else
  if (b > c) then
    max <= b;  -- b>a and b>c
  else
    max <= c;  -- b>a and c>b
  end if;
end if;
```

e.g., 2 conditional sig assignment codes

```vhdl
signal ac_max, bc_max: std_logic;
ac_max <= a when (a > c) else c;
bc_max <= b when (b > c) else c;
max <= ac_max when (a > b) else bc_max;
max <= a when ((a > b) and (a > c)) else c when (a > b) else b when (b > c) else c;
```

• 2 conditional sig assign implementations

```vhdl
signal ac_max, bc_max: std_logic;

ac_max <= a when (a > c) else c;
bc_max <= b when (b > c) else c;
max <= ac_max when (a > b) else bc_max;
```

```vhdl
max <= a when ((a > b) and (a > c)) else
c when (a > b) else
b when (b > c) else
c;
```

• e.g., "sharing" boolean condition

```vhdl
if (a > b and op="00") then
  y <= a - b;
  z <= a - 1;
  status <= '0';
else
  y <= b - a;
  z <= b - 1;
  status <= '1';
end if;
```

Incomplete branch and incomplete signal assignment

• According to VHDL definition:
  – Only the "then" branch is required; "elsif" and "else" branches are optional
  – Signals do not need to be assigned in all branch
  – When a signal is unassigned due to omission, it keeps the "previous value" (implying "memory")

Incomplete branch

• E.g.,

```vhdl
process(a,b)
begin
  if (a=b) then
    eq <= '1';
  else
    eq <= eq;
  end if;
end process;
```

• It implies

```vhdl
process(a,b)
begin
  if (a=b) then
    eq <= '1';
  else
    eq <= eq;
  end if;
end process
```

• fix

```vhdl
process(a,b)
begin
  if (a=b) then
    eq <= '1';
  else
    eq <= '0';
  end if;
end process
```
Incomplete signal assignment

- E.g.,

```vhdl
process(a, b)
begin
    if (a>b) then
gt <= '1';
    elsif (a=b) then
        eq <= '1';
    else
        lt <= '1';
    end if;
end process;
```

Fix #1:

```vhdl
process(a, b)
begin
    if (a>b) then
gt <= '1';
eq <= '0';
lt <= '0';
eq <= '1';
end if;
end process;
```

Fix #2

```vhdl
process(a, b)
begin
    if (a>b) then
gt <= '1';
eq <= '0';
lt <= '0';
eq <= '1';
end if;
end process;
```

Conceptual implementation

- Same as conditional signal assignment statement if the if statement consists of
  - One output signal
  - One sequential signal assignment in each branch
- Multiple sequential statements can be constructed recursively

E.g.

```vhdl
if boolean_expr then
    if boolean_expr_2 then
        signal_a <= value_expr_1;
    else
        signal_a <= value_expr_2;
    end if;
else
    if boolean_expr_3 then
        signal_a <= value_expr_3;
    else
        signal_a <= value_expr_4;
    end if;
end if;
```

E.g.

```vhdl
if boolean_expr then
    sig_a <= value_expr_a_1;
sig_b <= value_expr_a_1;
else
    sig_a <= value_expr_a_2;
sig_b <= value_expr_a_2;
end if;
```
5. Case statement

- Syntax
- Examples
- Comparison to selected signal assignment statement
- Incomplete signal assignment
- Conceptual Implementation

### Syntax

case case_expression is
  when choice_1 =>
    sequential statements;
  when choice_2 =>
    sequential statements;
  ...
  when choice_n =>
    sequential statements;
end case;

#### E.g., 4-to-1 mux

```vhdl
architecture case_arch of mux4 is
begin
  process(a,b,c,d,s)
  begin
    case s is
      when "00" =>
        z <= a;
      when "01" =>
        z <= b;
      when "10" =>
        z <= c;
      when others =>
        z <= d;
    end case;
  end process;
end case_arch;
```

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>c</td>
</tr>
<tr>
<td>0</td>
<td>d</td>
</tr>
</tbody>
</table>

#### E.g., 2-to-2^2 binary decoder

```vhdl
architecture case_arch of decoder4 is
begin
  process(a)
  begin
    case a is
      when "00" =>
        z <= "0001";
      when "01" =>
        z <= "0010";
      when "10" =>
        z <= "0100";
      when others =>
        z <= "1000";
    end case;
  end process;
end case_arch;
```

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
</tr>
</tbody>
</table>

### E.g., 4-to-2 priority encoder

```vhdl
architecture case_arch of pri_encoder42 is
begin
  process(z)
  begin
    case z is
      when "1000","1001","1010","1011" =>
        code <= "11";
      when "0100","0101","0110","0111" =>
        code <= "10";
      when "0010","0011" =>
        code <= "01";
      when others =>
        code <= "00";
    end case;
  end process;
end case_arch;
```

- Two statements are the same if there is only one output signal in case statement
- Case statement is more flexible
- Sequential statements can be used in choice branches

### Comparison to selected signal assignment

#### E.g.

```vhdl
process(a,b,c,d)
begin
  case a is
    when 0 =>
      ...;
    when 1 =>
      ...;
    when 10 =>
      ...;
  end case;
end process;
```
Incomplete signal assignment

• According to VHDL definition:
  – Signals do not need to be assigned in all choice branch
  – When a signal is unassigned, it keeps the “previous value” (implying “memory”)

• E.g.,

```vhdl
process (a)
  begin
    case a is
      when "100" | "101" | "110" | "111" =>
        high <= '1';
      when "010" | "011" =>
        middle <= '1';
      when others =>
        low <= '0';
    end case;
  end process;
```

• Fix #1:

```vhdl
process (a)
  begin
    case a is
      when "100" | "101" | "110" | "111" =>
        high <= '1';
        middle <= '0';
        low <= '0';
      when "010" | "011" =>
        high <= '0';
        middle <= '1';
        low <= '0';
      when others =>
        high <= '0';
        middle <= '0';
        low <= '1';
    end case;
  end process;
```

• Fix #2:

```vhdl
process (a)
  begin
    case a is
      when "100" | "101" | "110" | "111" =>
        high <= '1';
        middle <= '1';
        low <= '1';
      when others =>
        high <= '0';
        middle <= '0';
        low <= '0';
    end case;
  end process;
```

Conceptual implementation

• Same as selected signal assignment statement if the case statement consists of
  – One output signal
  – One sequential signal assignment in each branch
• Multiple sequential statements can be constructed recursively
6. Simple for loop statement

- **Syntax**
- **Examples**
- **Conceptual Implementation**

- **E.g., bit-wide xor**

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity wide_xor is
  port(a, b : in std_logic_vector(3 downto 0);
      y : out std_logic_vector(3 downto 0));
end wide_xor;

architecture wide_arch of wide_xor is
begin
  process(a, b)
  begin
    for i in (3 downto 0) loop
      y(i) <= a(i) xor b(i);
    end loop;
  end process;
end wide_arch;
```

- **E.g., reduced-xor**

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity reduced_xor is
  port(a : in std_logic_vector(3 downto 0);
       y : out std_logic);
end reduced_xor;

architecture reduced_arch of reduced_xor is
constant WIDTH: integer := 4;
signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
  process(a, tmp)
  begin
    tmp(0) <= a(0); -- boundary bit
    for i in (1 to WIDTH-1) loop
      tmp(i) <= a(i) xor tmp(i-1);
    end loop;
  end process;
  y <= tmp(WIDTH-1);
end reduced_arch;
```
Conceptual implementation

• “Unroll” the loop
• For loop should be treated as “shorthand” for repetitive statements
• E.g., bit-wise xor
  \[
  y(3) <= a(3) \text{ xor } b(3);
  y(2) <= a(2) \text{ xor } b(2);
  y(1) <= a(1) \text{ xor } b(1);
  y(0) <= a(0) \text{ xor } b(0);
  \]

• E.g., reduced-xor
  \[
  \text{tmp}(0) <= a(0);
  \text{tmp}(1) <= a(1) \text{ xor } \text{tmp}(0);
  \text{tmp}(2) <= a(2) \text{ xor } \text{tmp}(1);
  \text{tmp}(3) <= a(3) \text{ xor } \text{tmp}(2);
  y <= \text{tmp}(3);
  \]

Synthesis of sequential statements

• Concurrent statements
  – Modeled after hardware
  – Have clear, direct mapping to physical structures
• Sequential statements
  – Intended to describe “behavior”
  – Flexible and versatile
  – Can be difficult to be realized in hardware
  – Can be easily abused

• Think hardware
• Designing hardware is not converting a C program to a VHDL program