Lecture 7

VHDL (Part-2)

Concurrent and Sequential Statements, Loops

Concurrent vs Sequential Statements

Concurrent statements
- Simple signal assignment statement
- Conditional signal assignment statement
- Selected signal assignment statement

Sequential Statements
- VHDL process
- Sequential signal assignment statement
- Variable assignment statement
- If statement
- Case statement
- Simple for loop statement
Concurrent vs Sequential VHDL

**Modeling Style**

<table>
<thead>
<tr>
<th>Location</th>
<th>Concurrent</th>
<th>Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>inside architecture</td>
<td>inside architecture</td>
<td>inside process</td>
</tr>
</tbody>
</table>

**Example statements**

<table>
<thead>
<tr>
<th>Concurrent</th>
<th>Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>process, component instance, concurrent signal assignment</td>
<td>if, for, switch-case, signal assignment</td>
</tr>
</tbody>
</table>

---

Section 1

**CONCURRENT SIGNAL ASSIGNMENT STATEMENT**
Architecture body

Simplified syntax

```vhdl
architecture arch_name of entity_name is
    declarations;
begin
    concurrent statement;
    concurrent statement;
    concurrent statement;
    ...
end arch_name;
```

Simple Signal Assignment

Syntax:

```vhdl
signal_name <= projected_waveform;
```

- E.g.,
  ```vhdl
  y <= a + b + 1 after 10 ns;
  ```
Examples

E.g.,

- status <= '1';
- even <= (p1 and p2) or (p3 and p4);
- arith_out <= a + b + c - 1;

Implementation of last statement

```
  a
  +
  b
  +
  c
  -
  1
  ---
  arith_result
```

Conditional signal assignment statement

Syntax

```
signal_name <= value_expr_1 when boolean_expr_1 else
                value_expr_2 when boolean_expr_2 else
                value_expr_3 when boolean_expr_3 else
                ...
                value_expr_n
```
E.g., 4-to-1 mux

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity mux4 is
  port(
    a, b, c, d: in std_logic_vector(7 downto 0);
    s: in std_logic_vector(1 downto 0);
    x: out std_logic_vector(7 downto 0)
  );
end mux4;
architecture cond_arch of mux4 is
begin
  x <= a when (s="00") else
       b when (s="01") else
       c when (s="10") else
       d;
end cond_arch;
```

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

```vhdl
entity prio_encoder42 is
  port(
    r: in std_logic_vector(3 downto 0);
    code: out std_logic_vector(1 downto 0);
    active: out std_logic
  );
end prio_encoder42;
architecture cond_arch of prio_encoder42 is
begin
  code <= "11" when (r(3)="1") else
          "10" when (r(2)="1") else
          "01" when (r(1)="1") else
          "00";
  active <= r(3) or r(2) or r(1) or r(0);
end cond_arch;
```

<table>
<thead>
<tr>
<th>r</th>
<th>code</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>1---</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01--</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>001-</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>00</td>
<td>0</td>
</tr>
</tbody>
</table>
Conceptual representation

```plaintext
signal_name <= value_expr_1 when boolean_expr_1 else
           value_expr_2;
```

Examples

```plaintext
signal_name <= value_expr_1 when boolean_expr_1 else
           value_expr_2 when boolean_expr_2 else
           value_expr_3 when boolean_expr_3 else
           value_expr_4;
```
E.g.,

```vhdl
signal a, b, r: unsigned(7 downto 0);
signal s: std_logic_vector(1 downto 0);

... with s select
  r <= a+1 when "11",
       a-b-1 when "10",
       a+b when others;
```

Section 2

**SEQUENTIAL SIGNAL ASSIGNMENT STATEMENTS**
Sequential vs. Concurrent

architecture x of entity is
  declaration
  begin
    concurrent statement;
    concurrent statement;
    process()
      begin
        sequential statement;
        sequential statement;
      end
    concurrent statement;
    process()
      begin
        sequential statement;
        sequential statement;
      end
  end x;

Conceptual implementation
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

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

A process with a sensitivity list

Syntax

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

A process is like a circuit part, which can be
- active (known as 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
Example

E.g., 3-input and circuit
```vhdl
signal a, b, c, y: std_logic;
process(a, b, c)
begin
  y <= a and b and c;
end process;
```

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

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

Sequential Signal Assignment Statement

Syntax
```vhdl
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
Examples

E.g.,

```
process(a,b,c,d)
Begin y <= a or c;
    y <= a and b;
    y <= c and d;
end process;
```

It is same as

```
process(a,b,c,d)
begin
    y <= c and d;
end process;
```

IF statement

Syntax

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

```vhdl
architecture if_arch of mux4 is
begin
  process (a, b, c, d, s)
  begin
    if (s = "00") then
      x <= a;
    elsif (s = "01") then
      x <= b;
    elsif (s = "10") then
      x <= c;
    else
      x <= d;
    end if;
  end process;
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

```vhdl
sig <= value_expr_1 when boolean_expr_1 else
      value_expr_2 when boolean_expr_2 else
      value_expr_3 when boolean_expr_3 else
      value_expr_n;
```

It can be written as:

```vhdl
process(
  ...
)
begin
  if boolean_expr_1 then
    sig <= value_expr_1;
  elsif boolean_expr_2 then
    sig <= value_expr_2;
  elsif boolean_expr_3 then
    sig <= value_expr_3;
  else
    sig <= value_expr_n;
  end if;
end process;
```
Incomplete branch and incomplete signal assignment

According to VHDL definition:

- Only the “then” branch is required; “elsif” and “else” branches are optional
- When a signal is unassigned due to omission, it keeps the “previous value” (implying “memory”)

Incomplete branch

E.g.,

It implies

```vhdl
process(a, b)
begin
  if (a=b) then
    eq <= '1';
  else
    eq <= eq;
  end if;
end process;
```
Incomplete branch (cont’d)

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;
```
Fixes

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

```vhdl
if boolean_expr then
  sig_a <= value_expr_a_1;
  sig_b <= value_expr_b_1
else
  sig_a <= value_expr_a_2;
  sig_b <= value_expr_b_2;
end if;
```
Case statement

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

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>c</td>
</tr>
<tr>
<td>11</td>
<td>d</td>
</tr>
</tbody>
</table>
Comparison to selected signal assignment

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

with sel_exp select
  sig <= value_expr_1 when choice_1,
       value_expr_2 when choice_2,
       value_expr_3 when choice_3,
       ...,
       value_expr_n when choice_n;

It can be rewritten as:

case sel_exp is
  when choice_1 =>
    sig <= value_expr_1;
  when choice_2 =>
    sig <= value_expr_2;
  when choice_3 =>
    sig <= value_expr_3;
  ...
  when choice_n =>
    sig <= value_expr_n;
end case;

Incomplete signal assignment

According to VHDL definition:
- When a signal is unassigned, it keeps the “previous value” (implying “memory”)

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

```vhdl
process(a)
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;
```

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

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

VHDL provides a variety of loop constructs

Only a restricted form of loop can be synthesized

Syntax of simple for loop:
   for index in loop_range loop
       sequential statements;
   end loop;

loop_range must be static

Index assumes value of loop_range from left to right

Loops: example

Hardware:
   add_i: for i in 0 to max_c-1 generate
       b(i) <= a(i) + i;
   end generate add_i;

Generates <max_c> parallel computation units
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 demo_arch of wide_xor is
constant WIDTH: integer := 4;
begin
  process(a, b)
  begin
    for i in (WIDTH-1) downto 0 loop
      y(i) <= a(i) xor b(i);
    end loop;
  end process;
end demo_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) xor b(3);
y(2) <= a(2) xor b(2);
y(1) <= a(1) xor b(1);
y(0) <= a(0) xor b(0);
```
E.g., reduced-xor

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

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

architecture demo_arch of reduced_xor_demo 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 demo_arch;
```

Summary

Concurrent vs. sequential statement

**Concurrent**
- simple assignment
- selected assignment
- conditional assignment

**Sequential**
- Processes
- If statement
- Case statement
- Loop statement