ECE 3401 Lecture 7

Concurrent Statements & Sequential Statements (Process)

Concurrent Statements

- VHDL provides four different types of concurrent statements namely:
  - Signal Assignment Statement
    - Simple Assignment Statement
    - Selected Assignment Statement
    - Conditional Assignment Statement
  - Component Instantiation Statement
  - Generate Statement
  - Assert Statement

Generate Statements

- **Generate Statements**: describe regular and/or slightly irregular structure by automatically generating component instantiations instead of manually writing each instantiation.
  - E.g., if we implement the three-state buffers for a 32-bit bus using component instantiation statement, we will have to instantiate the three-state buffer component 32 times. In such cases, a ***generate*** statement is preferred.
- There are two variants of the generate statement:
  - FOR GENERATE statement
    - Provides a convenient way of repeating either a logic equation or a component instantiation.
  - IF GENERATE statement

Example 1 – 16-bit register

A 16-bit wide bus is to be connected to a 16-bit register. Create such a register from a single-bit FF using the GENERATE VHDL construct ?

```
LIBRARY work;
USE WORK.ECE3401_package.all;

ENTITY reg16 IS
  PORT ( input  : IN STD_LOGIC_VECTOR (0 to 15);
       clock  : IN STD_LOGIC;
       output : OUT STD_LOGIC_VECTOR (0 to 15);
END reg16;
```

```
ARCHITECTURE bus16_wide OF reg16 IS
 COMPONENT dff
        PORT ( d, clk : IN STD_LOGIC,
                      q     : OUT STD_LOGIC);
 END COMPONENT;

BEGIN

G1 : FOR  i  IN  0 to 15  GENERATE
        dff1: dff PORT MAP (input (i), clock, output(i));

END GENERATE G1;

END bus16_wide;
```

i - is the counter and does not need to be declared. It will automatically increase by 1 for each loop through the generate statement.

When 16 loops have been completed, generation will stop.

Example 2 – 16-bit full adder

```
Entity adder is
       port( a, b : in bit_vector (15 downto 0);
                      cin : in bit;
                      s : out bit_vector (15 downto 0);
                      cout : out bit );
end entity adder;

architecture behavioral of adder is
       signal c : bit_vector (15 downto 0);
begin
       array : for i in 0 to 15 generate
       begin
              first : if i=0 generate
              begin
                     cell : component full_adder
                            port map(a(i), b(i), cin, s(i), c(i));
              end generate first;
              other : if i/=0 generate
              begin
                     cell : component full_adder
                            port_map(a(i), b(i), c(i-1), s(i), c(i));
              end generate other;
       end generate array;
cout <= c(15);
end architecture behavioral;
```

Generate Statements Formats

- The syntax for the GENERATE statement:

```
Label :  generation_scheme GENERATE
        [concurrent_statements]
END GENERATE [label];
```

  Where generation_scheme ::=

```
FOR generate_specification
or
IF condition
```

- The beginning delimiter: GENERATE.
- The ending delimiter: END GENERATE.
- *A label is required for the generate statement* and is optional when used with the END GENERATE statement.

Sequential Statements

- Executed according to the order in which they appear.
- Permitted only within processes.
- Used to describe algorithms.

Sequential Statements

- There are six variants of the sequential statement, namely:
  - PROCESS Statement
  - IF-THEN-ELSE Statement
  - CASE Statement
  - LOOP Statement
  - WAIT Statement
  - ASSERT Statement

Process Statement

- **PROCESS** statement:
  - basic building block for behavioral modeling of digital systems.
  - concurrent shell in which a sequential statement can be executed.
    - appears inside an architecture body, and it encloses other statements within it.
    - IF, CASE, LOOP, and WAIT statements can appear only inside a process.
    - All statements with a process are executed sequentially when the process becomes active.

Process Statement Format

```
[Process_label]  : PROCESS [(sensitivity_list)] [is]
   Process_declarative_region
 BEGIN
        process_statement_region
 END PROCESS [Process_label]
```

- The optional label allows for a user_defined name for the process.
- The keyword **PROCESS** is the beginning delimiter of the process.
- The **END PROCESS** is the ending delimiter of the process statement.

Process Statement Sensitivity List

- Sensitivity list: an optional, contains the signals that trigger the process.
- The process statement begins to execute if any of the signals sensitivity list contains an event.
- Once activated by a sensitivity list event, the process statement executes statements in a sequential manner.
- Upon reaching the end of the process, execution suspends until another event occurs from the sensitivity list.

- **Process_declarative_region** may include:

  - type declaration
  - constant declaration
  - variable declaration

(Note: no signal declaration)

- **Process_Statement region** may include:

  -- signal assignment statement

` -- variable assignment statement

  -- IF statement

  -- CASE statement

  -- LOOP statement

  -- WAIT statement

Example

```
do_nothing : PROCESS
BEGIN
END PROCESS do_nothing;
```

The above example is a process statement that uses the label do_nothing.

Example

```
PROCESS (clock)
BEGIN
  -- toggles clock every 50ns
  Clock <= not clock after 50 ns;
END PROCESS;
```

- This process is sensitive to the signal "clock".
  - When an event occurs on clock, the process will execute.
- Within the process_statement_region of the process is a simple signal assignment statement. This statement inverts the value of clock after 50 ns.
- Basically this process toggles clock every 50 ns.

Variables

```
architecture RTL of XYZ is
  signal A, B, C : integer range 0 to 7;
  signal Y, Z    : integer range 0 to 15;
begin
  process (A, B, C)
    variable M, N : integer range 0 to 7;
  begin
    M := A;
    N := B;
    Z <= M + N;
    M := C;
    Y <= M + N;
  end process;
end RTL;
```

- **Variables** can be only defined in a **process or subprogram**.
  - Variables are only accessible within this process.
- In a process, the last signal assignment to a signal is carried out when the process execution is suspended. Value assignments to variables, however, are carried out immediately.
- '**<=**' : signal assignment
- '**:=**' : variable assignment.

Variables vs. Signals

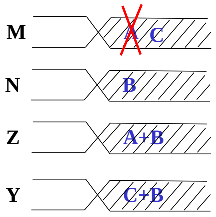
- There are three main differences between variable and signal assignment
  1) Syntax: for the variable assignment ':=', for the signal assignment operator '<='
  2) Timing: Variables are assigned immediately, while signals are assigned at a future delta time.
  3) Range: Variables are used for local processes and signals are used to pass information among concurrent statements.

Variables vs. Signals

```
signal A, B : integer;
signal C    : integer;
signal Y, Z : integer;

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

M,N: variables

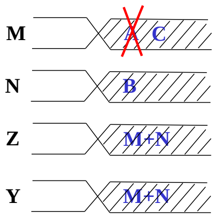


- The 2nd adder input is connected to C

Variables vs. Signals

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

M,N: signals



- Signal values are assigned after the process execution
- Only the last signal assignment is carried out
  - M <= A is overwritten by M <= C;
- The intermediate signals have to be added to the sensitivity list, as they are read during process execution.

Use of Variables

- Variables are suited for the implementation of algorithms.
- Variables store their value until the next process call, i.e., if a variable is read before a value has been assigned, the variable will have to show storage behavior. That means it will have to be synthesized to a latch or flip-flop respectively.

Variables: Example

- **Parity Calculation;**

```
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;
```

- While a scalar signal can always be associated with a wire, this is not valid for variables.
- In the example, FOR LOOP is executed four times. Each time the variable TMP describes a different line of the resulting hardware. The different lines are the outputs of the corresponding XOR gates.

1. IF Statement

```
if CONDITION then
  -- sequential statements
end if;

if CONDITION then
  -- sequential statements
else
  -- sequential statements
end if;

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

- Condition is a boolean expression
- Optional elsif sequence
  - Conditions may overlap
  - priority
- Optional else path
  - executed, if all conditions evaluate to false

Example 1

Write VHDL IF-THEN-ELSE code to model a D Flip-flop (input and output D and Q, respectively)

```
IF (clock’event and clock = 1)  THEN
    Q <=  D AFTER 5 ns;
END IF;
```

Example 2: Clocked 4-to-1 MUX

```
ENTITY clocked_mux IS
 PORT (inputs  :  IN  BIT_VECTOR (0 to 3);
      sel     :  IN  BIT_VECTOR (0 to 1);
      clk     :  IN  BIT;
      output  : OUT  BIT);
END clocked_mux;
```

```
ARCHITECTURE example OF clocked_mux IS
BEGIN
    PROCESS (clk)
        VARIABLE temp : BIT;
         BEGIN
            IF (clk = ‘1’) THEN
                IF  sel = “00” THEN
                        temp := inputs (0)
                ELSIF sel = “01” THEN
                        temp := inputs(1)
                ELSIF sel = “10” THEN
                        temp := inputs (2)
                ELSE
                        temp := inputs(3)
                END IF;
                output <= temp AFTER 5 ns;
            END IF;
    END PROCESS;
END example;
```

Example 3

```
entity IFSTMT is
   port (A, B, C, X : in bit_vector (3 downto 0);
                 Z : out bit_vector (3 downto 0);
end IFSTMT;
```

```
architecture EX1 of IFSTMT is
begin
   process (A, B, C, X)
    begin
      Z <= A;
      if (X = "1111") then
        Z <= B;
      elsif (X > "1000") then
        Z <= C;
      end if;
   end process;
end EX1;
```

=

```
architecture EX2 of IFSTMT is
begin
   process (A, B, C, X)
    begin
      if (X = "1111") then
        Z <= B;
      elsif (X > "1000") then
        Z <= C;
           else
        Z <= A;
      end if;
   end process;
end EX2;
```

2. Case Statement

VHDL syntax is :

```
CASE expression IS
        WHEN constant_value => sequential statements
        WHEN constant_value => sequential statements
        WHEN others => sequential statements
END CASE;
```

- The keyword **WHEN** is used to identify constant values that the expression might match. The expression evaluates a choice, and then the associated statements will be executed.
- The **CASE** statement will exit when all statements associated with the first matching constant value are executed.

Example 1

```
CASE vect IS
     WHEN “00” => int := 0;
     WHEN “01” => int := 1;
     WHEN “10” => int := 2;
     WHEN “11’ => int := 3;
END CASE;
```

- vect is a two element bit-vector. By evaluating vect and the matching **WHEN** value or choice causes the variable int to be assigned the matching integer value.

Example 2: Clocked 4-to-1 MUX

```
ENTITY clocked_mux IS
    PORT ( inputs : IN BIT_VECTOR (0 to 3);
        sel    : IN BIT_VECTOR (0 to 1);
        Clk    : IN BIT;
        output : OUT BIT);
END clocked-mux;

ARCHITECTURE behave OF clocked-mux IS
 BEGIN
    PROCESS (clk)
        VARIABLE temp : BIT;
      BEGIN
         CASE clk IS
            WHEN ‘1’ =>
                              CASE sel  IS
                 WHEN “00” => temp := inputs(0);
                 WHEN “01” => temp := inputs(1);
                 WHEN “10” => temp := inputs(2);
                 WHEN “11” => temp := inputs(3);
               END CASE;
               output <= temp AFTER 5 ns;
            WHEN OTHERS => NULL;
         END CASE;
      END PROCESS;
 END behave;
```

Example 3

```
entity CASE_STATEMENT is
     port (A, B, C, X :   in   integer range 0 to 15;
               Z      :  out   integer range 0 to 15;
end CASE_STATEMENT;
architecture EXAMPLE of CASE_STATEMENT is
begin
   process (A, B, C, X)
    begin
     case X is
       when 0 =>
                Z <= A;
       when 7 | 9 =>
                Z <= B;
       when 1 to 5 =>
                Z <= C;
       when others =>
                Z <= 0;
    end case;
  end process;
end EXAMPLE;
```