ECE 3401 Lecture 8

Sequential Statements

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

3. Loop Statement

- The LOOP statement provides a mechanism to repeatedly execute a sequence of statements. VHDL provides two types of loop statements:
  - FOR LOOP
  - WHILE LOOP.

FOR LOOP Statement

- *FOR LOOP syntax:*

```
[loop_label :]
FOR variable_name IN range LOOP
        sequential_statements
END LOOP [loop_label];
```

- The sequential_statements within the loop will be repeatedly executed within the range specified.

Example

```
FOR i IN 0 to 3 LOOP
   IF vect(i) = '1' THEN
       value := value + 2**i;
   ENDIF;
END LOOP;
```

- After the fourth pass, the loop range will be exceeded and the loop will terminate.
- A feature of VHDL: unlike most programming languages, the range variable i was not declared. Any range variable used within the FOR construct does not have to be declared. The same range identifier can be used repeatedly from one loop statement to the next.

Example

```
FOR i IN 3 downto 0 LOOP
   IF vect(i) = '1' THEN
       value := value + 2**i;
   ENDIF;
END LOOP;
```

- Has the same behavior as previous example. Only difference is that the range is descending.

WHILE LOOP Statement

- WHILE LOOP Syntax:

  [loop_label :]

  WHILE boolean_expression **LOOP**

  sequential_statements

  END LOOP [loop_label];
- The boolean_expression condition is evaluated, and if it is true the sequential_statements within the loop statement are evaluated until the condition is no longer true.

NEXT & EXIT Loop Termination Statements

- **NEXT** and **EXIT** statements are the two statements that can be used inside the loop statement
  - NEXT: terminate a loop iteration
  - EXIT: completely terminate the loop statement

4. Sensitivity List vs. Wait Statement

- The process statement contains only one sensitivity list. A process with a sensitivity list can only be triggered by an event on a signal in the list.
- Once triggered, the process will sequentially execute all of statements in the statement region and then suspend until another event is detected on those signals.
- If multiple signals are included in the sensitivity list, any one of those signals in the list can trigger the process. Therefore, the use of sensitivity list in a process is fairly limited.
  - To provide greater flexibility for the control of execution of a process, a **WAIT** statement can be used.

Wait Statement

- The **WAIT** statement provides the user with more options than the process sensitivity list.

- **Advantage:**
  - It can be placed anywhere within the process body.
    - With the process sensitivity list the process suspends at the end of the process.
    - With the **WAIT** statement, the suspension occurs where a **WAIT** statement is encountered.
  - There is no limitation to the number of **WAIT** statements within a process.
  - **WAIT** statements are more flexible.

Wait Statement

- **WAIT** statements stop the process execution.
  - The process is continued when the instruction is fulfilled
- Four types of wait statements:
  - wait on signal_list; -- wait for a signal event

    WAIT ON clock, clear, reset, D;
  - wait until condition; -- wait for true condition (requires an event)

    WAIT UNTIL (clock = ‘1’);

    WAIT UNTIL (clock =‘1’) or (clear = ‘0’);
  - wait for specific_time; -- wait for a specific time

    WAIT FOR 10ns;
  - wait; -- indefinite (process is never reactivated)
- Wait statements must not be used in processes with sensitivity list

Sensitivity List & Wait Statement

A process with sensitivity is functionally equivalent to a process statement with a WAIT statement as the last statement within the process.

PROCESS (clk)
BEGIN
clk <= NOT (clk) AFTER 50ns;
END PROCESS;
	PROCESS
	BEGIN
	clk <= NOT (clk) AFTER 50ns;
	WAIT ON clk;
	END PROCESS;

If a process does not have a sensitivity list and does not have a WAIT statement contained within it, the process will loop forever during initialization.

This is important to remember !

Example: D Flip-Flop Model

```
entity FF is
   port (D, CLK : in bit;
           Q    : out bit);
end FF;
```

```
architecture BEH_1 of FF is
   begin
     process
     begin
       wait on CLK;
       if (CLK = '1') then
         Q <= D;
       end if;
     end process;
   end BEH_1;
```

=

```
architecture BEH_2 of FF is
   begin
     process
     begin
       wait until CLK='1';
           Q <= D;
     end process;
   end BEH_2;
```

Example: Testbench Stimuli Generation

```
STIMULUS: process
   begin
     SEL <= `0`;
     BUS_B <= "0000";
     BUS_A <= "1111";
     wait for 10 ns;

     SEL <= `1`;
     wait for 10 ns;

     SEL <= `0`;
     wait for 10 ns;

     wait;
   end process STIMULUS;
```

- Via '**wait for**' construct it is very easy to generate simple input patterns for design verification purposes.
- Wait for constructs are excellent tool for describing timing specifications.

WAIT Statements and Behavioral Modeling

```
READ_CPU : process
begin
    wait until CPU_DATA_VALID = `1`;
      CPU_DATA_READ <= `1`;
    wait for 20 ns;
      LOCAL_BUFFER <= CPU_DATA;
    wait for 10 ns;
      CPU_DATA_READ <= `0`;
end process READ_CPU;
```

- It is easy to implement a bus protocol for simulation.
- The timing specification can directly be translated to simulatable VHDL code.
  - This behavioral modeling can only be used for simulation purposes as it is definitely not synthesizable.

5. Assertion Statement

- Check that expected conditions are met within the model
- Both concurrent and sequential statement, can be included anywhere in a process body
- [label :]**ASSERT** boolean_expression

  [**REPORT expression]**

  [**SEVERITY severity_level**];
- Severity_level: predefined enumeration type
  - TYPE severity_level IS (note, warning, error, failure)

Example

```
assert (last_position-first_position + 1) = number_of_entries
report "inconsistency in buffer model"
severity failure;
```

- Both report and severity clauses are optional
  - Default report string is: "Assertion violation"
  - Default severity level is: error

Concurrent Assertion Statement Example

```
architecture functional of S_R_flipflop is
begin
   q<='1' when s='1' else
     '0' when r='1';
   q_n<='0' when s='1' else
     '1' when r='1';
   check: assert not (s='1' and r='1')
          report "Incorrect use of S_R_flip_flop: s and r both
   '1'";
End architecture functional;
```

Process Using Signals and Corresponding Simulation Output

```
entity dummy is
end dummy

architecture sig of dummy is
signal trigger, sum : integer :=0;
signal sig1: integer :=1;
signal sig2: integer :=2;
signal sig3: integer :=3;
begin
    process
    begin
      wait on trigger;
      sig1 <= sig2+sig3;
      sig2 <= sig1;
      sig3 <= sig2;
     sum <= sig1 + sig2 + sig3;
   end process;
end sig;
```

At 10ns, trigger changes to ‘1’

Process Using Variables and Corresponding Simulation Output

```
entity dummy is
end dummy

architecture var of dummy is
signal trigger, sum : integer :=0;
begin
    process (trigger)
    variable var1: integer :=1;
    variable var2: integer :=2;
    variable var3: integer :=3;
    begin
      var1 := var2+var3;
      var2 := var1;
      var3 := var2;
     sum <= var1 + var2 + var3;
   end process;
end var;
```

At 10ns, trigger changes to ‘1’