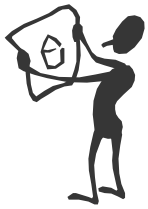
Chapter 7
Behavioral Modeling

VHDL

Outline

- Process Statement
- Signal Assignment Statement
- Variable Assignment Statement
- Wait Statement
- If-Then-Else Statement
- Case Statement
- Null Statement
- Loop Statement
- Exit & Next Statement
- Assertion Statement
- Report Statement



Behavioral Modeling

- **In the behavioral modeling style, the behavior of the entity is expressed using sequentially executed, procedural code, which is very similar in syntax and semantics to that of a high-level programming language like C or Pascal. A process statement is the primary mechanism used to model the behavior of an entity. This chapter describes the process statement and the various kinds of sequential statements that can be used within a process statement to model such behavior.**
- **There is meaning to the order of the statements in the process.**

Process Statement

- **The syntax of the process is:**

```
[P-Label:] PROCESS [(sensitivity-list)][IS]
  [process-item-declarations]
BEGIN
  sequential-statement1;
  sequential-statement2;


END PROCESS [P-Label];
```

- **A set of signals to which the process is sensitive is defined by the sensitivity list. In other words, each time an event occurs on any of the signals in the sensitivity list, the sequential statements within the process are executed in a sequential order, that is, in the order in which they appear (similar to statements in a high-level programming language like C or Pascal). The process then suspends after executing the last sequential statement and waits for another event to occur.**

Sequential Signal Assignment

- **The syntax is:**

target-signal* <= *waveform;

– Examples:

```
MyTest: PROCESS (a, b)
       BEGIN
        z <= a;
        Y <= a AFTER 10 ns;
        X <= (a AND b) AFTER 20 ns;
        W <= a AFTER 10 ns, '1' AFTER 20 ns, '0' AFTER 30 ns;
        END PROCESS MyTest;
```


Variable Assignment

- **The syntax is:**

target-variable := expression;

– Examples:

```
MyTest: PROCESS (a)
          VARIABLE inx: integer := -1; -- only initialize
        BEGIN
          inx := inx + 1;
        END PROCESS MyTest;
```

IF-THEN-ELSE Statement

- Logically equivalent to concurrent conditional signal assignment (WHEN-ELSE), but more versatile.
- Syntax:

```
label:                    -- optional
IF condition1 THEN
  statements1;
ELSIF condition2 THEN     -- optional section
  statements2;
ELSE                      -- optional section
  statements3;
END IF;
```

- Executes the first block of statements following a TRUE condition
- No other statement blocks are executed

IF-THEN-ELSE (example)

- Series of conditions forms a priority encoder structure:

```
PROCESS (x,y,z,a,b,c,d)
BEGIN
   IF x = '1' THEN
      foo <= a;
   ELSIF y = '1' THEN
      foo <= b;
   ELSIF z = '1' THEN
      foo <= c;
   ELSE
      foo <= d;
   END IF;
END PROCESS;
```

- Result:

  foo = x * a + /x * y * b + /x * /y * z * c + /x * /y * /z * d

IF-THEN-ELSE Statement

- **CAUTION** - **Implied memory** (latch) will be created if a signal is not ***always*** given a value each time the process runs
- Example:

```
PROCESS (xbus)
BEGIN
   IF xbus = X"F" THEN
      doitnow <= '1';    -- output of a set-only latch
   END IF;
END PROCESS;
```

- If a latch is NOT desired
  - include a default value assignment
  - or be sure some branch of the IF statement always assigns a value

IF-THEN-ELSE (latch)

```
ENTITY latch IS
  PORT (a,b: IN std_logic;   --inputs
        sel: IN std_logic;
        y,x: OUT std_logic); --output
END latch;
-------------------------------------------------------
ARCHITECTURE behavior OF latch IS
BEGIN
  PROCESS (a,b,sel)
  BEGIN
      IF sel = '0' THEN
            y <=  a;
            x <=  a;
      ELSE
            y <=  '-';
      END IF;
  END PROCESS;
END behavior;
```

```
x =  sel * x.CMB  +  a * /sel
y =  a
```


CASE-WHEN Statement

- Logically equivalent to concurrent selected signal assignment (WITH-SELECT-WHEN), but more versatile
- Syntax:

```
label:                          -- optional
CASE selector_expression IS
  WHEN choice1 => statements1;
  WHEN choice2 => statements2;
  WHEN choice3 [| choice4] => statements3;
  [WHEN OTHERS  => statements4;]     -- optional
END CASE;
```

- Executes the single block of statements following a valid choice, or following OTHERS
- Choices must be mutually exclusive and all inclusive
- Forms a multiplexer-based logic structure

CASE-WHEN (example)

- Example:

```
CASE State IS
  WHEN “000” => x <= x + 1;
  WHEN “001” => x <= x - 1;
  WHEN “010” | “110” => y <= x;
  WHEN “011” =>
    IF (z = ‘1’) THEN
      y <= “00”;
    ELSE
      y <= “11”;
    END IF;
  WHEN OTHERS  => Temp <= (OTHERS => ‘0’);
END CASE;
```

NULL Statement

- The **NULL** is a sequential statement that does not cause any action to take place; execution continues with the next statement. One example of this statement's use is in an if statement or in a case statement where, for certain conditions, it may be useful or necessary to explicitly specify that no action needs to be performed.
- For example:

```
CASE State IS
  WHEN "00" => x <= x + 1;
  WHEN "01" => x <= x - 1;
  WHEN "10" => y <= x;
  WHEN OTHERS => NULL;
END CASE;
```


LOOP Statements

- **Sequential statements for repetitive, iterative operations**
- **NOT executed iteratively in synthesized hardware!**
- **Simply a way of calculating new signal values in a process**
- **General syntax is:**

```
[loop-label:] iteration-scheme LOOP
  sequential-statement1;
  sequential-statement2;
END LOOP [loop-label];
```

- **FOR-LOOP executes specific number of iterations**
- **WHILE-LOOP executes as long as Boolean condition is TRUE**
- **LOOP executes as long as no EXIT appear.**

FOR LOOP

- For-Loop general format:

```
FOR index IN looprange LOOP
  statements;   -- generally using the index
END LOOP;
```

- Index variable automatically declared
- Index variable auto-increments or decrements at end of loop
- The loop index can not be assigned any value inside the loop
- The index name is local to the loop (if another variable with the same name exist outside the loop)
- The range in for loop can also be a range of an enumeration type

FOR LOOP (example)

- For-Loop example 1

```
FOR j IN 7 DOWNTO 0 LOOP
  array(j) <= X"00";  -- clear a signal array
  x(j) := y(j) + 5;   -- integer array variable
END LOOP;
```

- For-Loop example 2

```
fact := 1;
FOR j IN 2 TO N LOOP
  fact := fact * j;
END LOOP;
```


FOR LOOP (synthesis example)

```
--   Example - 2 Level Full Adder
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
-----------------------------------------------------
ENTITY adder4 IS
  GENERIC (N: integer := 2);
  PORT ( a:   IN  unsigned(N-1 DOWNTO 0);     --inputs
         cin: IN  unsigned(N/2-1 DOWNTO 0);   --carry in
         co: OUT  unsigned(N/2-1 DOWNTO 0);   --carry out
         y:  OUT  unsigned(0 DOWNTO 0));      --output
END adder4;
-----------------------------------------------------
ARCHITECTURE behavior OF adder4 IS
BEGIN
  PROCESS (a)
   VARIABLE Temp: unsigned(N/2 DOWNTO 0);
  BEGIN
     Temp := '0' & cin;
   FOR i IN 0 TO N-1 LOOP
        Temp := Temp + (0=>a(i), OTHERS=>'0');
   END LOOP;
   y(0) <= Temp(0);
   co <= Temp(N/2 DOWNTO 1);
  END PROCESS;
END behavior;
```


FOR LOOP (synthesis result)

co(0) =
a(1) * cin(0)
+ a(0) * cin(0)
+ a(0) * a(1)

y(0) =
a(0) * a(1) * cin(0)
+ /a(0) * /a(1) * cin(0)
+ /a(0) * a(1) * /cin(0)
+ a(0) * /a(1) * /cin(0)

FOR LOOP (synthesis example)

```
--   Example - 4 Level Full Adder
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
-------------------------------------------------------
ENTITY adder4 IS
  GENERIC (N: integer := 4);
  PORT ( a:   IN  unsigned(N-1 DOWNTO 0);     --inputs
         cin: IN  unsigned(N/2-1 DOWNTO 0);   --carry in
         co: OUT  unsigned(N/2-1 DOWNTO 0);   --carry out
         y:  OUT  unsigned(0 DOWNTO 0));      --output
END adder4;
-------------------------------------------------------
ARCHITECTURE behavior OF adder4 IS
BEGIN
  PROCESS (a)
   VARIABLE Temp: unsigned(N/2 DOWNTO 0);
  BEGIN
     Temp := '0' & cin;
   FOR i IN 0 TO N-1 LOOP
         Temp := Temp + (0=>a(i), OTHERS=>'0');
   END LOOP;
   y(0) <= Temp(0);
   co <= Temp(N/2 DOWNTO 1);
  END PROCESS;
END behavior;
```


FOR LOOP (synthesis result)

```
S_1 =
      /a(0) * /a(1) * /a(3) * /cin(0) * cin(1)
    + /a(0) * a(1) * a(2) * /cin(0) * /cin(1)
    + a(0) * /a(1) * a(3) * /cin(0) * /cin(1)
    + a(1) * /a(2) * a(3) * /cin(0) * /cin(1)

 S_2 =
      a(1) * a(2) * a(3) * cin(0) * cin(1)
    + a(0) * a(2) * a(3) * cin(0) * cin(1)
    + a(0) * a(1) * a(3) * cin(0) * cin(1)
    + a(0) * a(1) * a(2) * cin(0) * cin(1)
    + a(0) * a(1) * a(2) * a(3) * cin(1)
    + /a(0) * /a(1) * /a(2) * /a(3) * cin(1)
    + /a(0) * /a(1) * /a(2) * /cin(0) * cin(1)
    + /a(0) * /a(2) * /a(3) * /cin(0) * cin(1)
    + /a(1) * /a(2) * /a(3) * /cin(0) * cin(1)
    + /a(0) * /a(1) * a(2) * a(3) * /cin(1)
    + /a(0) * a(1) * /a(2) * cin(0) * /cin(1)
    + /a(1) * /a(2) * a(3) * cin(0) * /cin(1)
    + /a(0) * a(2) * /a(3) * cin(0) * /cin(1)
    + a(0) * /a(1) * /a(3) * cin(0) * /cin(1)
    + a(0) * a(1) * /a(2) * /a(3) * /cin(1)
    + a(0) * a(2) * /a(3) * /cin(0) * /cin(1)
```


FOR LOOP (synthesis result)

```
\MODULE_1:g1:a0:g0:u0:ga:g1:ua0\ =
      a(0) * cin(0) * cin(1)

 \MODULE_2:g1:a0:g0:u0:ga:g1:ua0\ =
      /a(0) * a(1) * cin(0) * cin(1)
    + a(0) * a(1) * /cin(0) * cin(1)

 \MODULE_3:g1:a0:g0:u0:ga:g1:ua0\ =
      /a(0) * /a(1) * a(2) * cin(0) * cin(1)
    + /a(0) * a(1) * a(2) * /cin(0) * cin(1)
    + a(0) * /a(1) * a(2) * /cin(0) * cin(1)
    + a(0) * a(1) * a(2) * cin(0) * /cin(1)

 \MODULE_4:g1:a0:g0:u0:ga:g1:ua0\ =
      /a(0) * /a(1) * /a(2) * a(3) * cin(0) * cin(1)
    + /a(0) * /a(1) * a(2) * a(3) * /cin(0) * cin(1)
    + /a(0) * a(1) * /a(2) * a(3) * /cin(0) * cin(1)
    + a(0) * /a(1) * /a(2) * a(3) * /cin(0) * cin(1)
    + /a(0) * a(1) * a(2) * a(3) * cin(0) * /cin(1)
    + a(0) * /a(1) * a(2) * a(3) * cin(0) * /cin(1)
    + a(0) * a(1) * /a(2) * a(3) * cin(0) * /cin(1)
    + a(0) * a(1) * a(2) * a(3) * /cin(0) * /cin(1)
```

FOR LOOP (synthesis result)

```
/co(0) =
     /S_1.CMB * /S_2.CMB

co(1) =
     /\MODULE_1:g1:a0:g0:u0:ga:g1:ua0\.CMB *  \MODULE_2:g1:a0:g0:u0:ga:g1:ua0\.CMB *
      \MODULE_3:g1:a0:g0:u0:ga:g1:ua0\.CMB *   \MODULE_4:g1:a0:g0:u0:ga:g1:ua0\.CMB
    + \MODULE_1:g1:a0:g0:u0:ga:g1:ua0\.CMB *   /\MODULE_2:g1:a0:g0:u0:ga:g1:ua0\.CMB *
      \MODULE_3:g1:a0:g0:u0:ga:g1:ua0\.CMB *   \MODULE_4:g1:a0:g0:u0:ga:g1:ua0\.CMB
    + \MODULE_1:g1:a0:g0:u0:ga:g1:ua0\.CMB *    \MODULE_2:g1:a0:g0:u0:ga:g1:ua0\.CMB *
      /\MODULE_3:g1:a0:g0:u0:ga:g1:ua0\.CMB *   \MODULE_4:g1:a0:g0:u0:ga:g1:ua0\.CMB
    + /\MODULE_1:g1:a0:g0:u0:ga:g1:ua0\.CMB *   /\MODULE_2:g1:a0:g0:u0:ga:g1:ua0\.CMB *
      /\MODULE_3:g1:a0:g0:u0:ga:g1:ua0\.CMB *   \MODULE_4:g1:a0:g0:u0:ga:g1:ua0\.CMB
    + \MODULE_1:g1:a0:g0:u0:ga:g1:ua0\.CMB *    \MODULE_2:g1:a0:g0:u0:ga:g1:ua0\.CMB *
      \MODULE_3:g1:a0:g0:u0:ga:g1:ua0\.CMB *    /\MODULE_4:g1:a0:g0:u0:ga:g1:ua0\.CMB
    + /\MODULE_1:g1:a0:g0:u0:ga:g1:ua0\.CMB *   /\MODULE_2:g1:a0:g0:u0:ga:g1:ua0\.CMB *
      \MODULE_3:g1:a0:g0:u0:ga:g1:ua0\.CMB *    /\MODULE_4:g1:a0:g0:u0:ga:g1:ua0\.CMB
    + /\MODULE_1:g1:a0:g0:u0:ga:g1:ua0\.CMB *   \MODULE_2:g1:a0:g0:u0:ga:g1:ua0\.CMB *
      /\MODULE_3:g1:a0:g0:u0:ga:g1:ua0\.CMB *   /\MODULE_4:g1:a0:g0:u0:ga:g1:ua0\.CMB
    + \MODULE_1:g1:a0:g0:u0:ga:g1:ua0\.CMB *    /\MODULE_2:g1:a0:g0:u0:ga:g1:ua0\.CMB *
      /\MODULE_3:g1:a0:g0:u0:ga:g1:ua0\.CMB *   /\MODULE_4:g1:a0:g0:u0:ga:g1:ua0\.CMB
```


FOR LOOP (synthesis result)

```
y(0) =
     a(0) * a(1) * a(2) * a(3) * cin(0)
   + /a(0) * /a(1) * a(2) * a(3) * cin(0)
   + /a(0) * a(1) * /a(2) * a(3) * cin(0)
   + a(0) * /a(1) * /a(2) * a(3) * cin(0)
   + /a(0) * a(1) * a(2) * /a(3) * cin(0)
   + a(0) * /a(1) * a(2) * /a(3) * cin(0)
   + a(0) * a(1) * /a(2) * /a(3) * cin(0)
   + /a(0) * /a(1) * /a(2) * /a(3) * cin(0)
   + /a(0) * a(1) * a(2) * a(3) * /cin(0)
   + a(0) * /a(1) * a(2) * a(3) * /cin(0)
   + a(0) * a(1) * /a(2) * a(3) * /cin(0)
   + /a(0) * /a(1) * /a(2) * a(3) * /cin(0)
   + a(0) * a(1) * a(2) * /a(3) * /cin(0)
   + /a(0) * a(1) * a(2) * /a(3) * /cin(0)
   + /a(0) * a(1) * /a(2) * /a(3) * /cin(0)
   + a(0) * /a(1) * /a(2) * /a(3) * /cin(0)
```


WHILE LOOP

- While-Loop general format:

```
WHILE boolean-expression LOOP
  statements;
END LOOP;
```

- No automatic index declaration or modification

WHILE LOOP (example)

- While-Loop example 1

```
j := 1; x:= 0;
WHILE j < 5 LOOP
  x := x + array(j);
  j := j + 1;
END LOOP;
```

- While-Loop example 2

```
j := 2; fact:= 1;
WHILE j <= N LOOP
  fact := fact * j;
  j := j + 1;
END LOOP;
```


LOOP

- Loop general format:

```
LOOP
  statements;
END LOOP;
```

- No automatic index declaration or modification
- The loop is infinite (if no exit, next or return)
- Actually it is equivalent to WHILE TRUE LOOP

EXIT Statements

- The exit statement is a sequential statement that can be used only inside a loop. It causes execution to jump out of the innermost loop or the loop whose label is specified. The syntax for an exit statement is:

```
EXIT [loop-label] [WHEN condition];
```

- If no loop label is specified, the innermost loop is exited. If the when clause ("when condition") is used, the specified loop is exited only if the given condition is true; otherwise, execution continues with the next statement.

EXIT (example)

- Exit example 1

```
FOR j IN 1 TO 10 LOOP
  array(j) <= X"00";
  IF j+i > 4 THEN
    EXIT; -- stop loop if j+i > 4
  END IF;
END LOOP;
```

- Exit example 2

```
Lp1: FOR j IN 1 TO 10 LOOP
       array(j) <= X"00";
       EXIT Lp1 WHEN j+i > 4;
     END LOOP Lp1;
```


NEXT Statements

- The next statement is also a sequential statement that can be used only inside a loop. The syntax is the same as that for the exit statement except that the keyword next replaces the keyword exit. Its syntax is:

```
NEXT [loop-label] [WHEN condition];
```

- The next statement results in skipping the remaining statements in the current iteration of the specified loop; execution resumes with the first statement in the next iteration of this loop, if one exists. If no loop label is specified, the innermost loop is assumed. In contrast to the exit statement, which causes the loop to be terminated, the next statement causes the current loop iteration of the specified loop to be prematurely terminated; execution resumes with the next iteration.

Next (example)

- Next example 1

```
FOR opcode IN nop TO jmp LOOP  -- enumerated
  Inx := Inx + 1;
  IF opcode = jmp THEN
    NEXT;   -- skip next line if jump
  ELSE
     pc := pc + 1;
  END IF;
END LOOP;
```

- Next example 2

```
Lp1: FOR i IN 1 TO 10 LOOP
     Lp2: LOOP
            NEXT Lp1 WHEN Done = '1';
            Jnx = Jnx + 1; -- This line is skip
          END LOOP Lp2
     Inx = Inx + 1;        -- This line is skip
END LOOP Lp1;
```