# VHDL:

# Combinational Logic Modeling

## Goals

- To study the details of concurrent signal assignment statements (simple, conditional, and selected).
- To learn basic sequential statements (if, case, null, loop, exit, for, while, and next).
- To learn how to write and use procedures and functions.
- To learn how to use generic constants.
- To get familiar with generate statements (for and if).

## Concurrent Signal Assignment Statements

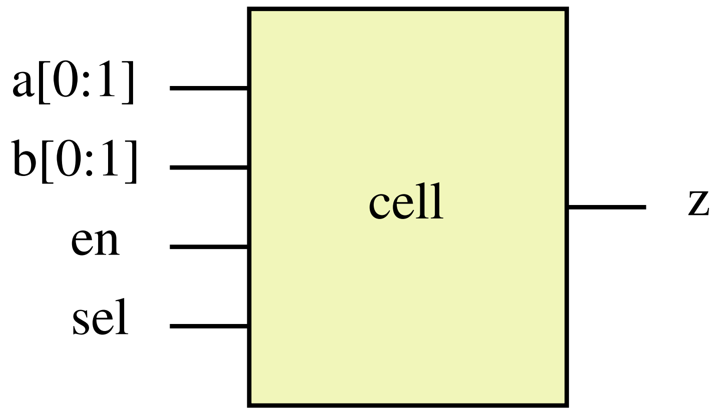
- Signal assignment statements are **concurrent** unless they are declared inside a process.
- Concurrent statements are executed at the same time (not sequentially).
- VHDL is based on an **event driven** simulation kernel. A signal assignment statement is not executed until any of its right hand side signals changes.
- There are three types of concurrent signal assignment statement:
  1. Concurrent simple signal assignment statements
  2. Concurrent conditional signal assignment statements
  3. Concurrent selected signal assignment statements

## Concurrent Simple Signal Assignment Statements

**Example: 1 (Class)** *Show how to model combinational logic using concurrent simple signal assignment statements.*

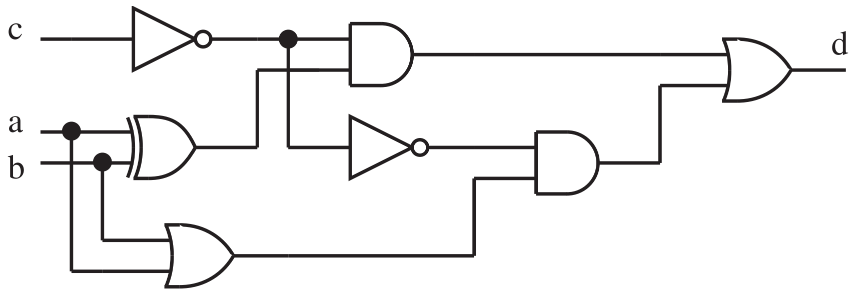
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY cell IS
   PORT( a, b:  IN std_logic_vector (0 TO 1);
         en, sel:  IN std_logic;
         z:  OUT std_logic);
END ENTITY cell;
ARCHITECTURE wires OF cell IS
   SIGNAL s0, s1, s2, s3:  std_logic;
BEGIN
   s0 <= a(0) NAND b(1);
   s1 <= a(1) NAND b(0);
   s2 <= s0 NOR s1;
   s3 <= en AND sel;
   z <= s2 NOR s3;
END ARCHITECTURE wires;
```



## Concurrent Conditional Signal Assignment Statements

**Example: 2 (Class)** *Show how to model the shown combinational network using concurrent conditional signal assignment statement.*



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY condition IS
   PORT( a, b, c:  IN std_logic;
         d:  OUT std_logic);
END ENTITY condition;

ARCHITECTURE test OF condition IS
BEGIN
   d <=  a XOR b WHEN c = '0' ELSE
         a OR b;
END ARCHITECTURE test;
```

## Concurrent Selected Signal Assignment Statements

**Example: 3 (Class)** *The following decoder is an example on using concurrent selected signal assignment statement.*

```
PACKAGE p1 IS
   TYPE states IS (reset, hold, apply, done);
END PACKAGE p1;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.p1.ALL;
ENTITY test IS
   PORT( present_state:  IN states;
         flag:  OUT std_logic_vector (0 TO 1));
END ENTITY test;
ARCHITECTURE rtl OF test IS BEGIN
   WITH present_state SELECT
      flag <=  "00" WHEN reset | hold,
               "10" WHEN apply,
               "--" WHEN OTHERS;
END ARCHITECTURE rtl;
```

## VHDL Sequential Statements

Sequential statements can only be used inside a PROCESS or other sequential bodies (functions, procedures). They are grouped into:

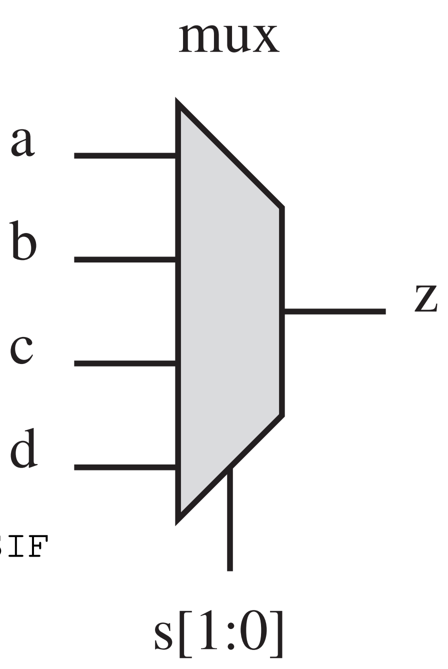
1. Sequential ASSERT (studied before)
2. WAIT (studied before)
3. IF
4. CASE
5. NULL
6. Loop statements. There are three types of loop statements:
   (a) LOOP (infinite loop)
   (b) FOR-LOOP
   (c) WHILE-LOOP
7. EXIT
8. NEXT
9. RETURN

# IF Statements

**Example: 4 (Class)** *Write entity and architecture declarations for a 4-1 mux using an IF statement.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux IS
   PORT( a, b, c, d:  IN std_logic;
         s:  IN std_logic_vector (1 DOWNTO 0);
         z:  OUT std_logic);
END ENTITY mux;
ARCHITECTURE mux OF mux IS BEGIN
   pmux:  PROCESS (a, b, c, d, s) IS BEGIN
      IF    s = "00" THEN     z <= a;
      ELSIF s = "01" THEN     z <= b; -- notice ELSIF
      ELSIF s = "10" THEN     z <= c;
      ELSIF s = "11" THEN     z <= d;
      ELSE                    z <= a;
      END IF;
   END PROCESS pmux;
END ARCHITECTURE mux;
```



## CASE Statements

**Example: 5 (Class)** *Write entity and architecture declarations for an address decoder using a CASE statement.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY decoder IS
   PORT( address:  IN std_logic_vector (2 DOWNTO 0);
         decode:  OUT std_logic_vector (7 DOWNTO 0));
END ENTITY decoder;
ARCHITECTURE behav OF decoder IS BEGIN
   pdec:  PROCESS (address) IS BEGIN
      CASE address IS
         WHEN "001"            => decode <= X"11";
         WHEN "111"            => decode <= X"42";
         WHEN "010"            => decode <= X"44";
         WHEN "101" | "110"    => decode <= X"88";
         WHEN OTHERS           => decode <= X"00";
      END CASE;
   END PROCESS pdec;
END ARCHITECTURE behav;
```

## CASE Statements (Cont.)

**Example: 6 (Class)** *Here is another address decoder using a CASE statement.*

```
ENTITY decoder IS
   PORT( address:  IN integer;
         decode:  OUT bit_vector (1 DOWNTO 0));
END ENTITY decoder;

ARCHITECTURE another OF decoder IS
BEGIN
   pdec:  PROCESS (address) IS
   BEGIN
      CASE address IS
         WHEN 0 to 7                =>    decode <= "10";
         WHEN 8 to 15               =>    decode <= "01";
         WHEN 16 | 20 | 24 | 28     =>    decode <= "11";
         WHEN OTHERS                => NULL;  -- sequential statement
      END CASE;
   END PROCESS pdec;
END ARCHITECTURE another;
```

## CASE Statements (Cont.)

- A case statement looks similar to a truth table.
- Truth tables could directly translate to CASE statements.

**Example: 7 (Class)** *Write entity and architecture declarations for a full adder using a CASE statement.*

```
ENTITY full_adder IS
   PORT( a, b, c_in:  IN bit;
         s, c_out:  OUT bit);
END ENTITY full_adder;
ARCHITECTURE fa OF full_adder IS
BEGIN
   pa:  PROCESS (a, b, c_in) IS
      VARIABLE input:  bit_vector (0 TO 2);
      VARIABLE output:  bit_vector (0 TO 1);
            ⋮
```

## CASE Statements (Cont.)

**Example 7: (Cont.)**

```
            ⋮
  BEGIN
    input := a & b & c_in;
    CASE input IS
      WHEN "000" => output := "00";
      WHEN "001" => output := "10";
      WHEN "010" => output := "10";
      WHEN "011" => output := "01";
      WHEN "100" => output := "10";
      WHEN "101" => output := "01";
      WHEN "110" => output := "01";
      WHEN "111" => output := "11";
    END CASE;
    s <= output(0);
    c_out <= output(1);
  END PROCESS pa;
END ARCHITECTURE fa;
```

## CASE Statements (Cont.)

**Example: 8 (Class)** *Write entity and architecture declarations for an ALU using a CASE statement.*

```
PACKAGE pack_a IS
   TYPE op_type IS (add, sub, mul, div);
END PACKAGE pack_a;

LIBRARY ieee;
USE ieee.numeric_bit.ALL;
USE WORK.pack_a.ALL;

ENTITY alu IS
   PORT( op:  IN op_type;
         a, b:  IN signed (3 DOWNTO 0);
         c:  OUT signed (3 DOWNTO 0));
END ENTITY alu;
         ⋮
```

## CASE Statements (Cont.)

**Example 8: (Cont.)**

```
          ⋮
ARCHITECTURE alu OF alu IS
BEGIN
   palu:  PROCESS (op, a, b) IS
   VARIABLE temp:  signed (7 DOWNTO 0) := x"00";
   BEGIN
      CASE op IS
         WHEN add => c <= a + b;
         WHEN sub => c <= a - b;
         WHEN mul => temp := a * b; c <= temp (3 DOWNTO 0);
         WHEN div => c <= a / b;
      END CASE;
   END PROCESS palu;
END ARCHITECTURE simple;
```

## LOOP Statements

**Example: 9 (Class)** *Model a free-running counter that starts from 0 and increments mod 16 on each rising edge of the clock.*

```
ENTITY counter IS
   PORT( clk:  IN bit;
         count:  OUT natural RANGE 0 TO 15);
END ENTITY counter;
ARCHITECTURE behav OF counter IS
BEGIN
   pcount:  PROCESS IS
      VARIABLE count_value:  natural RANGE 0 TO 15 := 0;
   BEGIN
      count <= count_value;
      LOOP
         WAIT UNTIL clk = '1';
         count_value := (count_value + 1) MOD 16;
         count <= count_value;
      END LOOP;
   END PROCESS pcount;
END ARCHITECTURE behav;
```

## LOOP and EXIT Statements

**Example: 10 (Class)** *Revise the counter model in Example 9 to include:*

- *Reset input: A high reset input resets the counter to zero. The output stays zero as long as the reset input is high and resumes counting on the next clock rising edge after reset changes to* `'0'`.
- *Stop input: A high stop input stops the counter. The output stays at its current value forever.*

```
ENTITY counter_advanced IS
   PORT( clk, reset, stop:  IN bit;
         count:  OUT natural RANGE 0 TO 15);
END ENTITY counter_advanced;
ARCHITECTURE behav OF counter_advanced IS
BEGIN
   preset:  PROCESS IS
      VARIABLE count_value:  natural RANGE 0 TO 15 := 0;
         ⋮
```

## LOOP and EXIT Statements (Cont.)

**Example 10: (Cont.)**

```
          ⋮
   BEGIN
      count  <= count_value;
      outer:  LOOP
         inner:  LOOP
            WAIT UNTIL clk = '1' or stop = '1' or reset = '1';
            EXIT outer WHEN stop = '1';
            EXIT WHEN reset = '1';     -- EXIT can be used with any loop
            count_value := (count_value + 1) MOD 16;
            count  <= count_value;
         END LOOP inner;
         count_value := 0;
         count  <= count_value;
         WAIT UNTIL reset = '0';
      END LOOP outer;
      WAIT;
   END PROCESS preset;
END ARCHITECTURE behav;
```

## LOOP and EXIT Statements (Cont.)

**Example: 11 (Home)** *State whether each of the following statements is true or false with respect to the counter in Example 10.*

1. *The counter has a synchrounous stop.*
2. *Stop has a priority over reset.*
3. *The counter is sensitive to the clock level.*

# FOR-LOOPS

**Example: 12 (Class)** *Write entity and architecture declarations for a circuit that compresses a bit_vector by bit-wise XORing all its elements.*

```
ENTITY compress IS
   PORT( a:  IN bit_vector (7 DOWNTO 0);
         b:  OUT bit);
END ENTITY compress;
ARCHITECTURE bit_wise OF compress IS
BEGIN
   pcomp:  PROCESS IS
      VARIABLE v:  bit;
   BEGIN
      v := '0';                   -- i is not defined outside the loop
      FOR i IN a'range LOOP       -- i does not need declaration
         v := v XOR a(i);
      END LOOP;
      b <= v;
      WAIT ON a;
   END PROCESS pcomp;
END ARCHITECTURE bit_wise;
```

## WHILE-LOOPS and NEXT Statements

**Example: 13 (Class)** *Model a circuit that counts the ones in a vector.*

```
ENTITY ones IS
   PORT( a:  IN bit_vector (7 DOWNTO 0);
         b:  OUT natural);
END ENTITY ones;
ARCHITECTURE behav OF ones IS BEGIN
   pones:  PROCESS (a) IS
      VARIABLE n, v:  natural;
      VARIABLE x:  bit;
   BEGIN
      n := a'low; v := 0;
      WHILE n <= a'high LOOP
         x := a(n);
         n := n + 1;
         NEXT WHEN x = '0';      -- NEXT can be used with any loop
         v := v + 1;
      END LOOP;
      b <= v;
   END PROCESS pones;
END ARCHITECTURE behav;
```

## Procedures

- Subprograms in VHDL are procedures and functions.
- A procedure consists of a name, a set of formal parameters, a set of declarations, and a set of executable, sequential statements.
- A formal parameter of a procedure is replaced on each call with an actual parameter and so can take different values each time.
- A procedure may read and write the values of signals and variables declared outside itself. However, this is not very flexible. This can be overcome by passing the signals into the procedure as parameters.
- A procedure may contain wait statements. This will cause the calling process to suspend until the wait’s condition is met.
- Parameters can be passed using positional or named association.

## Procedures (Cont.)

**Example: 14 (Class)** *Write a procedure to be used in comparing two std_logic_vectors.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY comparator IS
   Port( a:  IN std_logic_vector (7 DOWNTO 0);
         b:  IN std_logic_vector (7 DOWNTO 0);
         equal_out:  OUT std_logic;
         not_equal_out:  OUT std_logic);
END ENTITY comparator;
ARCHITECTURE Behavioral OF comparator IS BEGIN
   cp:  PROCESS (a, b) IS
      PROCEDURE compare(  in_1, in_2:  IN std_logic_vector;
                          equal, not_equal:  OUT std_logic) IS
      BEGIN
         ⋮
```

## Procedures (Cont.)

**Example 14: (Cont.)**

```
          .
          .
          .
          IF in_1 = in_2 THEN
             equal := '1';
             not_equal := '0';
          ELSE
             equal := '0';
             not_equal := '1';
          END IF;
       END PROCEDURE compare;

       VARIABLE equal, not_equal:  std_logic;
    BEGIN
       compare(a, b, equal, not_equal);
       equal_out <= equal;
       not_equal_out <= not_equal;
    END PROCESS cp;
END ARCHITECTURE Behavioral;
```

## Procedures (Cont.)

- A parameter could have a default value. In this case, it is optional and may be omitted when the procedure is called.
- Formal parameters of unconstrained array types are useful, as the actual index constraint can be different for each procedure call.
- Procedures need to make use of the attribute ’range to obtain the bounds of the unconstrained array parameter.
- Variables inside a procedure are initialized each time the procedure is called. This is quite different from variables and constants declared inside a process, which are only initialized at time 0, and carry their old values forward between process executions.

# Procedures (Cont.)

$$\begin{array}{ccccc} \left.\begin{array}{l} CONSTANT \\ VARIABLE \\ SIGNAL \end{array}\right\} & N: & \left\{\begin{array}{l} IN \\ OUT \\ INOUT \end{array}\right\} & T := & default \\ \textbf{class} & \textbf{name} & \textbf{mode} & \textbf{type} & \textbf{default} \end{array}$$

- Default mode is IN.
- IN parameters can only be read, OUT parameters can only be updated, and INOUT parameters can be read and updated.
- IN parameters default to CONSTANT, whereas OUT and INOUT parameters default to VARIABLE.
- In a procedure call, a SIGNAL parameter must be associated with a signal, a VARIABLE with a variable, but a CONSTANT may be associated with any expression.
- A **RETURN** (a sequential statement) may appear anywhere in a procedure, but is optional as there is an implied return after the last procedure statement.

## Procedures (Cont.)

**Example: 15 (Class)** *Determine the class and mode of the following procedure parameters. Check whether the procedure call is legal or not.*

```
PROCEDURE demo(  a:   std_logic;
                 b:   OUT std_logic;      c:   INOUT std_logic;
                 SIGNAL d:   std_logic;  SIGNAL e:   OUT std_logic) IS
          .
          .
          .
SIGNAL sig:   std_logic;
VARIABLE var:   std_logic;
          .
          .
          .
   demo(sig NAND var, var, var, sig, sig);
```

| Formal parameter | Class | Mode | Actual parameter | Legal? |
|---|---|---|---|---|
| a | CONSTANT | IN | expression | yes |
| b | VARIABLE | OUT | VARIABLE | yes |
| c | VARIABLE | INOUT | VARIABLE | yes |
| d | SIGNAL | IN | SIGNAL | yes |
| e | SIGNAL | OUT | SIGNAL | yes |

## Procedures (Cont.)

**Example: 16 (Class)** *Write a procedure in a package to find the index of the first one on the right in a vector.*

```
PACKAGE package_procedures IS
   PROCEDURE find_first_one( SIGNAL v:  IN bit_vector;
                             SIGNAL first_one_index:  OUT natural;
                             SIGNAL found:  OUT bit);
END PACKAGE package_procedures;
PACKAGE BODY package_procedures IS
   PROCEDURE find_first_one( SIGNAL v:  IN bit_vector;
                             SIGNAL first_one_index:  OUT natural;
                             SIGNAL found:  OUT bit) IS BEGIN
      FOR index IN v'reverse_range LOOP -- check first one on the right
         IF v(index) = '1' THEN
            found <= '1';
            first_one_index <= index;
            RETURN;
         END IF;
      END LOOP;
      found <= '0';
   END PROCEDURE find_first_one;
END PACKAGE BODY package_procedures;
```

## Procedures (Cont.)

**Example: 17 (Class)** *Write a procedure to be used in defining a set of stimuli to a D-flip-flop. Use positional association for parameters.*

```
ENTITY testbench IS
END ENTITY testbench;

ARCHITECTURE proc_pos OF testbench IS
   COMPONENT d_ff IS
      PORT( d, clk:  IN bit;
            q:  OUT bit);
   END COMPONENT d_ff;
   FOR dut:  d_ff USE ENTITY WORK.d_ff (d_ff);
   SIGNAL d_in, clock, q_out:  bit;
BEGIN
   dut:  d_ff PORT MAP (d_in, clock, q_out);
         ⋮
```

## Procedures (Cont.)

**Example 17: (Cont.)**

```
      ⋮
pd:  PROCESS IS
   PROCEDURE assign( sig:  bit_vector) IS BEGIN
      d_in <= sig(0); clock <= sig(1);
      WAIT FOR 20 ns;
   END PROCEDURE assign;
BEGIN
   assign("10"); assign("11");
   ASSERT q_out= '1'
      REPORT "Problem with capturing a 1 on a rising edge"
      SEVERITY error;
   assign("01");
   ASSERT q_out = '1'
      REPORT "Problem with No Change                      "
      SEVERITY error;
      ⋮
```

## Procedures (Cont.)

**Example 17: (Cont.)**

```
          ⋮
      assign("00");
      ASSERT q_out = '1'
         REPORT "Problem with No Change                     "
         SEVERITY error;
      assign("01");
      ASSERT q_out = '0'
         REPORT "Problem with capturing a 0 on a rising edge"
         SEVERITY error;
      WAIT;                       -- stop simulation run
   END PROCESS pd;
END ARCHITECTURE proc_pos;
```

**Example: 18 (Class)** *In Example 17, what is the index range of the bit string literal `"10"` passed to procedure `assign` as `assign("10")`?*

VHDL assigns the index range `0 TO n-1` to a bit string literal of length n. So, the index range of the bit_vector passed to assign is `(0 TO 1)`.

## Procedures (Cont.)

**Example: 19 (Class)** *Rewrite Example 17 in a more flexible way. Use named association for parameters*

```
ENTITY testbench IS
END ENTITY testbench;

ARCHITECTURE proc_named OF testbench IS
   COMPONENT d_ff IS
      PORT( d, clk:  IN bit;
            q:  OUT bit);
   END COMPONENT d_ff;
   FOR dut:  d_ff USE ENTITY WORK.d_ff (d_ff);
   SIGNAL d_in, clock, q_out:  bit;
BEGIN
   dut:  d_ff PORT MAP (d_in, clock, q_out);
   pd:  PROCESS IS
           ⋮
```

## Procedures (Cont.)

### Example 19: (Cont.)

```
        ⋮
    PROCEDURE assign( sig:  bit_vector;
                      SIGNAL x, y:  OUT bit;
                      pause:  time := 15 ns) IS
    BEGIN
        x <= sig(0);
        y <= sig(1);
        WAIT FOR pause;
    END PROCEDURE assign;
BEGIN
    assign(sig => "10", x => d_in, y => clock);
    assign(sig => "11", x => d_in, y => clock, pause => 20 ns);
    ASSERT q_out = '1'
        REPORT "Problem with capturing a 1 on a rising edge"
        SEVERITY error;
        ⋮
```

## Procedures (Cont.)

**Example 19: (Cont.)**

```
         ⋮
      assign(x => d_in, sig => "01", y => clock, pause => 20 ns);
      ASSERT q_out = '1'
         REPORT "Problem with No Change                      "
         SEVERITY error;
      assign(sig => "00", x => d_in, y => clock, pause => 20 ns);
      ASSERT q_out = '1'
         REPORT "Problem with No Change                      "
         SEVERITY error;
      assign(pause => 20 ns, sig => "01", x => d_in, y => clock);
      ASSERT q_out = '0'
         REPORT "Problem with capturing a 0 on a rising edge"
         SEVERITY error;
      WAIT;                       -- stop simulation run
   END PROCESS pd;
END ARCHITECTURE proc_named;
```

## Functions

- VHDL functions can be used to define any logical, mathematical, algorithmeic, or type-conversion function, which calculates a single result based on the values of a set of parameters.
- A function cannot contain a **wait** statement. It would not make any sense for a function to suspend the process in the middle of evaluating the expression containing the function call. This means that functions always consume **zero simulation time**.
- A function cannot have **OUT** or **INOUT** parameters
- A function may contain any number of **RETURN** statements, each returning a value of the return type. A compile-time error will occur if execution runs off the end of the function without finding a return statement.
- VHDL functions can be declared **pure** or **impure**.

## Functions (Cont.)

**Pure** functions:

- always return the same value for the same set of actual parameters.
- are meant to be **deterministic** and their return values are based only on the function parameter list and visible constants.
- cannot read values from **files**. Since the compiler does not know the contents of a file, it cannot guarantee that the function return value is deterministic when it accesses a file.
- cannot have **side effects**, unlike a procedure. They cannot assign a value to an object declared outside the function.

## Functions (Cont.)

**Impure** functions:

- may return different values for the same set of parameters.
- may be **indeterministic** and their return values are not based only on the function parameter list and visible constants.
- can read values from **files**.
- can read or write any signal within its scope, also those that are not on the parameter list. Therefore, the return value may depend on these shadow parameters as well.
- may have **side effects**, like updating objects outside of their scope (not assigned from its return value).

## Functions (Cont.)

**Example: 20 (Class)** *Write a function to count the ones in a vector.*

```
    ⋮
FUNCTION ones( a:  std_logic_vector) RETURN natural IS
   VARIABLE n:  natural RANGE 0 TO a'length := 0;
BEGIN
   FOR i IN a'range LOOP
      IF a(i) = '1' THEN
         n := n + 1;
      END IF;
   END LOOP;
   RETURN n;
END FUNCTION ones;

    ⋮
VARIABLE v:  std_logic_vector (3 DOWNTO 0); -- v'range=(3 DOWNTO 0)
VARIABLE w:  std_logic_vector (1 TO 8); -- w'range=(1 TO 8)

    ⋮
IF ones(v) > 2 THEN
   y <= ones(v) + ones(w);
END IF;
```

## Functions (Cont.)

**Example: 21 (Class)** *Write a package declaration that defines a function that ands two unsigned numbers.*

```
LIBRARY ieee;
USE ieee.numeric_bit.ALL;
PACKAGE andPackage IS
   FUNCTION newAND(  l, r:  unsigned) RETURN unsigned;
END PACKAGE andPackage;
PACKAGE BODY andPackage IS
   FUNCTION newAND(  l, r:  unsigned) RETURN unsigned IS
      ALIAS l_op:  unsigned (1 TO l'length) is l;
      ALIAS r_op:  unsigned (1 TO r'length) is r;
      VARIABLE res:  unsigned (1 TO l'length);
   BEGIN
      FOR j IN l_op'range LOOP
         res(j) := l_op(j) AND r_op(j);
      END LOOP;
      RETURN res;
   END FUNCTION newAND;
END PACKAGE BODY andPackage;
```

## Functions (Cont.)

**Example: 22 (Class)** *Write a function to return the parity of a vector in a package.*

```
PACKAGE package_functions IS
   FUNCTION parity(  a:  bit_vector) RETURN bit;
END PACKAGE package_functions;

PACKAGE BODY package_functions IS
   FUNCTION parity(  a:  bit_vector) RETURN bit IS
      VARIABLE result:  bit := '0'; -- default value
   BEGIN
      FOR i IN a'range LOOP
         result := result XOR a(i);
      END LOOP;
      RETURN result;
   END FUNCTION parity;
END PACKAGE BODY package_functions;
```

## Functions (Cont.)

**Example: 23 (Class)** *Write a function to locate the index of the first one on the right in a vector. If the vector has no ones, set the function return value to the vector's highest index + 1.*

```
FUNCTION first_one(  a:  bit_vector) RETURN integer IS
   VARIABLE index:  integer RANGE a'low TO a'high + 1;
BEGIN
   FOR index IN a'reverse_range LOOP -- check from the right
      IF a(index) = '1' THEN
         RETURN index;
      END IF;
   END LOOP;
   RETURN a'high + 1;
END FUNCTION first_one;
```

## Functions (Cont.)

**Example: 24 (Class)** *Write a package declaration that defines a function that converts a natural number to unsigned.*

```
LIBRARY ieee;
USE ieee.numeric_bit.ALL;

PACKAGE convertPackage IS
   FUNCTION natural2unsigned(arg, size:  natural) RETURN unsigned;
END PACKAGE convertPackage;
         ⋮
```

## Functions (Cont.)

**Example 24: (Cont.)**

```
         ⋮
PACKAGE BODY convertPackage IS
   FUNCTION natural2unsigned(arg, size:  natural) RETURN unsigned IS
      VARIABLE result:  unsigned (size - 1 DOWNTO 0);
      VARIABLE i_val:  natural := arg;
   BEGIN
      FOR i IN 0 to result'left LOOP
         IF (i_val MOD 2) = 0 THEN
            result (i) := '0';
         ELSE
            result (i) := '1';
         END IF;
         i_val := i_val/2;
      END LOOP;
      ASSERT i_val = 0
         REPORT "Vector truncated"
         SEVERITY warning;
      RETURN result;
   END FUNCTION natural2unsigned;
END PACKAGE BODY convertPackage;
```

# Declarative Part Usage

| | entity | architecture | process | package | package body | configuration |
|---|---|---|---|---|---|---|
| configuration | | X | | | | |
| use clause | X | X | | X | X | X |
| component | | X | | X | | |
| variable | | | X | | | |
| signal | X | X | | X | X | |
| constant | X | X | X | X | X | |
| type and subtype | X | X | X | X | X | |
| subprogram declaration | X | X | X | X | X | |
| subprogram body | X | X | X | | X | |
| generic | X | X | | X | | |

## Generic Constants

- Generics are channels for static information to be communicated to a block from its environment, whereas ports are dynamic communication links between the block and its environment.
- Thus, generics declaration must be placed within the system entity.
- Generics are used to pass parameters to an instance of an entity, as seen from the outside of the system.
- Parameters passed to an instance can be: signal delay, load capacitance, number of bits of a bus, number of bits of a register, etc.
- Objects in a generic list must be constants.
- The main difference between generics and constants is in that generics can be used dynamically while constants are purely static. You can change the value of a generic without changing the code. However, constants cannot be changed without changing the code.

## Generic Constants (Cont.)

**Example: 25 (Class)** *Define a two-input and gate with a specified delay and show how to instantiate it in building a four-input and gate.*

```
ENTITY and2 IS
   GENERIC( delay:  time := 1 ns);
   PORT( a, b:  IN bit;
         c:  OUT bit);
END ENTITY and2;

ARCHITECTURE simple OF and2 IS
BEGIN
   c <= a AND b AFTER delay;
END ARCHITECTURE simple;
```

## Generic Constants (Cont.)

### Example 25: (Cont.)

```
ENTITY and4 IS
   PORT(  x1, x2, x3, x4:  IN bit;
          y:  OUT bit);
END ENTITY and4;
ARCHITECTURE struct OF and4 IS
   COMPONENT and2 IS
      GENERIC(  delay:  time := 1 ns);
      PORT(  a, b:  IN bit;
             c:  OUT bit);
   END COMPONENT and2;
   FOR ALL: and2 USE ENTITY work.and2 (simple);
   SIGNAL z1, z2:  bit;
BEGIN
   gate1: and2  GENERIC MAP(  delay => 1 ns)
                PORT MAP(  x1, x2, z1);
   gate2: and2  GENERIC MAP(  delay => 1 ns)
                PORT MAP(  x3, x4, z2);
   gate3: and2  GENERIC MAP(  delay => 1 ns)
                PORT MAP(  z1, z2, y);
END ARCHITECTURE struct;
```

## Generic Constants (Cont.)

**Example: 26 (Class)** *Model an and gate with different loading using selected signal assignment statement. Include a spikes-rejection property.*

```
ENTITY and2 IS
   GENERIC( delay:  time := 2 ns;
            rej:  time := 1 ns;
            load:  positive);
   PORT( a, b:  IN bit;
         c:  OUT bit);
END ENTITY and2;

ARCHITECTURE basic OF and2 IS
   SIGNAL int:  bit;
BEGIN
   int <= a AND b;
   c <=  REJECT rej INERTIAL
            int AFTER delay          WHEN load = 1 ELSE
            int AFTER (5 * delay)    WHEN load = 2 ELSE
            int AFTER (10 * delay)   WHEN load = 3 ELSE
            int AFTER (load**2 * delay);
END ARCHITECTURE basic;
```

## Generic Constants (Cont.)

**Example 26: (Cont.)**

```
ENTITY test IS
   PORT( x1, x2, x3, x4:  IN bit;
         y1, y2:  OUT bit);
END ENTITY test;
ARCHITECTURE test OF test IS
   COMPONENT and2 IS
      GENERIC(  delay:  time := 2 ns;
                rej:  time := 1 ns;
                load:  positive);
      PORT( a, b:  IN bit;
            c:  OUT bit);
   END COMPONENT and2;
   FOR ALL: and2 USE ENTITY work.and2 (basic);
BEGIN
   gate1:  and2 GENERIC MAP( delay => 4 ns, load => 2)
                PORT MAP(    a => x1, b => x2, c => y1);
   gate2:  and2 GENERIC MAP( 4 ns, 2 ns, 4)
                PORT MAP(    x3, x4, y2);
END ARCHITECTURE test;
```

## Generic Constants (Cont.)

**Example: 27 (Class)** *Describe an and gate with rise and fall times, which depend on loading.*

```
ENTITY and2 IS
   GENERIC( rise, fall:  time;
            load:  positive);
   PORT( a, b:  IN bit;
         c:  OUT bit);
END ENTITY and2;

ARCHITECTURE load_dep OF and2 IS
   SIGNAL int:  bit;
BEGIN
   int <= a AND b;
   c <=  int AFTER (rise + (load * 2 ns)) WHEN int = '1' ELSE
         int AFTER (fall + (load * 3 ns));
END ARCHITECTURE load_dep;
```

## Generic Constants (Cont.)

**Example: 28 (Class)** *Describe an and gate with rise and fall times and a spikes-rejection property.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY and2 IS
   GENERIC( rise:  time := 1.5 ns;
            fall:  time := 1.2 ns;
            x:  time := 500 ps;
            rise_rej:  time := 400 ps;
            fall_rej:  time := 300 ps;
            x_rej:  time := 300 ps);
   PORT( a, b:  IN std_logic;
         c:  OUT std_logic);
END ENTITY and2;
            ⋮
```

## Generic Constants (Cont.)

**Example 28: (Cont.)**

```
                ⋮
ARCHITECTURE detailed_delay OF and2 IS
   SIGNAL result:  std_logic;
BEGIN
   gate:  PROCESS (a, b) IS BEGIN
      result <= a AND b;
   END PROCESS gate;
   delay:  PROCESS (result) IS BEGIN
      IF result THEN
         c <= REJECT rise_rej INERTIAL '1' AFTER rise;
      ELSIF NOT result THEN
         c <= REJECT fall_rej INERTIAL '0' AFTER fall;
      ELSE
         c <= REJECT x_rej INERTIAL 'X' AFTER x;
      END IF;
   END PROCESS delay;
END ARCHITECTURE detailed_delay;
```

## Generic Constants (Cont.)

**Example: 29 (Class)** *Write entity and architecture declarations for an ALU with a variable word width that performs add, sub, mul, and div.*

```
LIBRARY ieee;
USE ieee.numeric_bit.ALL;
ENTITY alu_variable IS
   GENERIC( width:  positive := 8);
   PORT( a, b:  IN signed (width-1 DOWNTO 0);
         control:  IN unsigned (1 DOWNTO 0);
         c:  OUT signed (width-1 DOWNTO 0));
END ENTITY alu_variable;
ARCHITECTURE alu_variable OF alu_variable IS BEGIN
   Palu:  PROCESS (control, a, b) IS
      VARIABLE temp:  signed (2*width -1 DOWNTO 0) := (OTHERS => '0');
   BEGIN
      CASE control IS
         WHEN "00"   => c <= a + b;
         WHEN "01"   => c <= a - b;
         WHEN "10"   => temp := a * b; c <= temp(width-1 DOWNTO 0);
         WHEN "11"   => c <= a / b;
      END CASE;
   END PROCESS Palu;
END ARCHITECTURE alu_variable;
```

## Generic Constants (Cont.)

**Example: 30 (Class)** *Write entity and architecture declarations for an ALU with a variable word width. The ALU is to perform an add operation with a carry in and out and three logical operations: and, or, and xor.*

```
LIBRARY ieee;
USE ieee.numeric_bit.ALL;

ENTITY alu_carry IS
   GENERIC( width:  positive := 8);
   PORT( a, b:  IN signed (width-1 DOWNTO 0);
         cin:  IN bit;
         control:  IN unsigned (1 DOWNTO 0);
         d:  OUT signed (width-1 DOWNTO 0);
         cout:  OUT bit);
END ENTITY alu_carry;
            ⋮
```

## Generic Constants (Cont.)

**Example 30: (Cont.)**

```
                ⋮
ARCHITECTURE behave OF alu_carry IS BEGIN
   Calu:  PROCESS (a, b, cin, control) IS
      VARIABLE temp, temp_cin:  signed (width DOWNTO 0);
   BEGIN
      temp_cin := (OTHERS => '0'); temp_cin (0) := cin;
      cout <= 0;
      CASE control IS
         WHEN "00"   => d <= a AND b;
         WHEN "01"   => d <= a OR b;
         WHEN "10"   => d <= a XOR b;
         WHEN "11"   =>
            temp := ('0' & a) + ('0' & b) + temp_cin;
            d <= temp(width-1 DOWNTO 0);
            cout <= temp(width);
      END CASE;
   END PROCESS Calu;
END ARCHITECTURE alu_behave;
```

# Declarative Part Usage

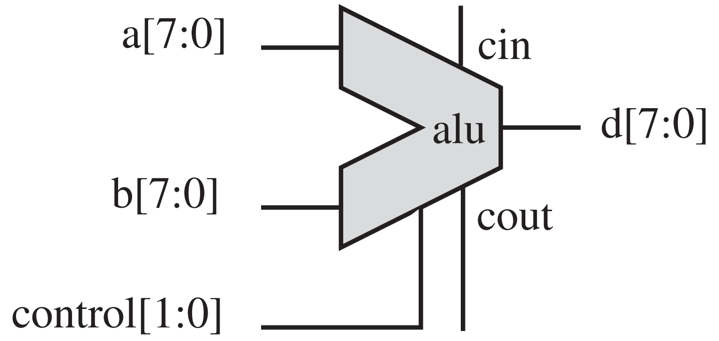
| | entity | architecture | process | package | package body | configuration |
|---|---|---|---|---|---|---|
| configuration | | X | | | | |
| use clause | X | X | | X | X | X |
| component | | X | | X | | |
| variable | | | X | | | |
| signal | X | X | | X | X | |
| constant | X | X | X | X | X | |
| type and subtype | X | X | X | X | X | |
| subprogram declaration | X | X | X | X | X | |
| subprogram body | X | X | X | | X | |
| generic | X | X | | X | | |

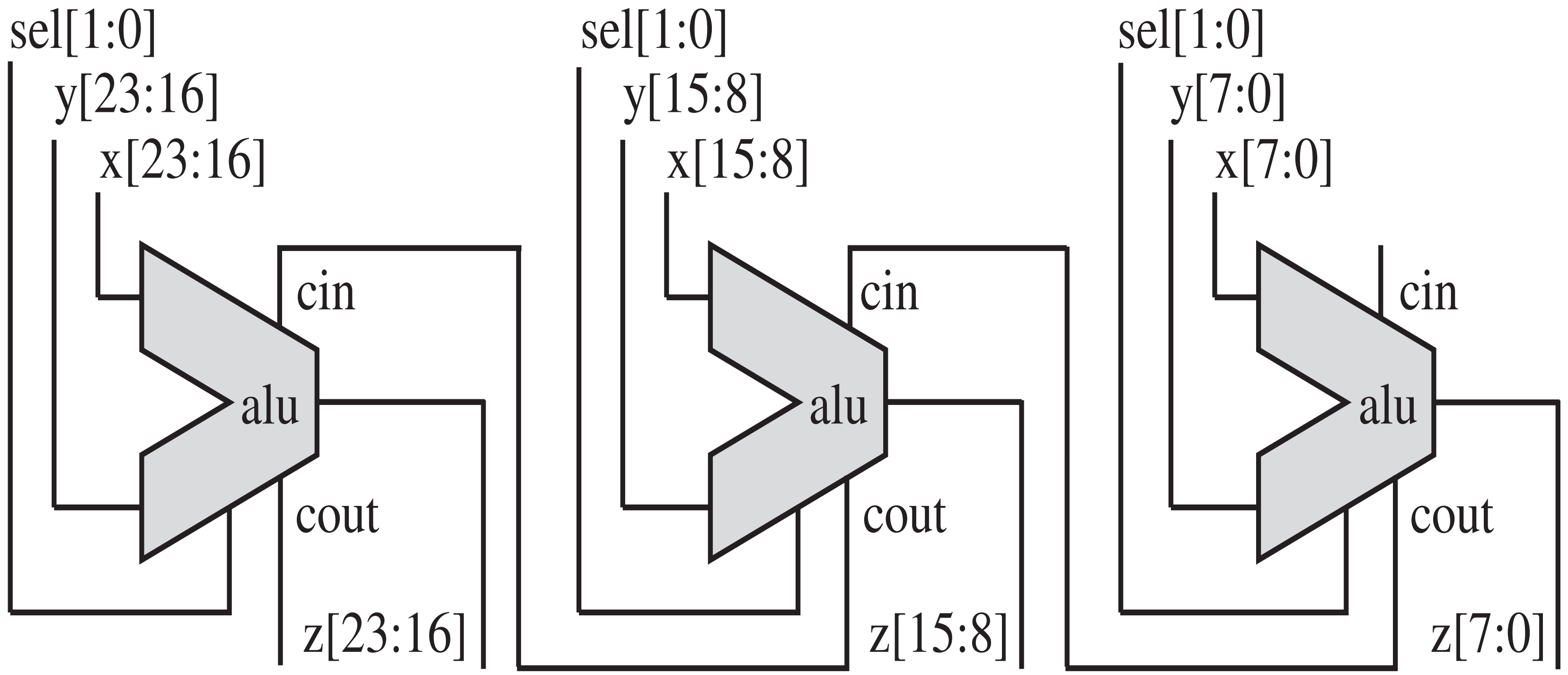
## Generate Statements

- Generate statements cause the enclosed concurrent statements to be replicated.
- There are two types of generate statements:
  - **for-generate**: replicates the enclosed statements for a given number of times.
  - **if-generate**: conditionally replicates the enclosed statements.
- Generate statements are concurrent ones.

## FOR-Generate Statements

**Example: 31 (Class)** *Assume we have configured an instance type of the entity alu_carry in Example 30 as shown to the right and would like to use it in generating the wider ALU shown below.*





## FOR-Generate Statements (Cont.)

### Example 31: (Cont.)

```
LIBRARY ieee;
USE ieee.numeric_bit.ALL;

ENTITY alu_wide1 IS
      PORT( x, y:  IN signed (23 DOWNTO 0);
            cin:  IN bit;
            sel:  IN unsigned (1 DOWNTO 0);
            z:  OUT signed (23 DOWNTO 0);
            cout:  OUT bit);
END ENTITY alu_wide1;
ARCHITECTURE wide1 OF alu_wide1 IS
   COMPONENT alu_carry IS
      GENERIC(  width:  positive := 8);
      PORT( a, b:  IN signed (width-1 DOWNTO 0);
            cin:  IN bit;
            control:  IN unsigned (1 DOWNTO 0);
            d:  OUT signed (width-1 DOWNTO 0);
            cout:  OUT bit);
   END COMPONENT alu_carry;
            ⋮
```

## FOR-Generate Statements (Cont.)

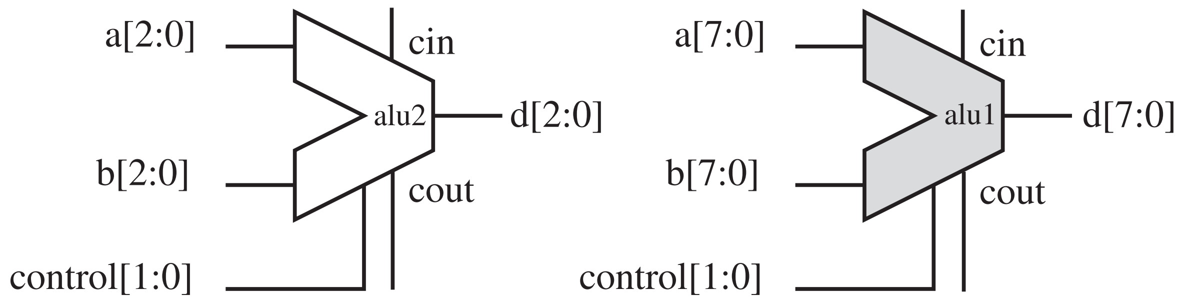
**Example 31: (Cont.)**

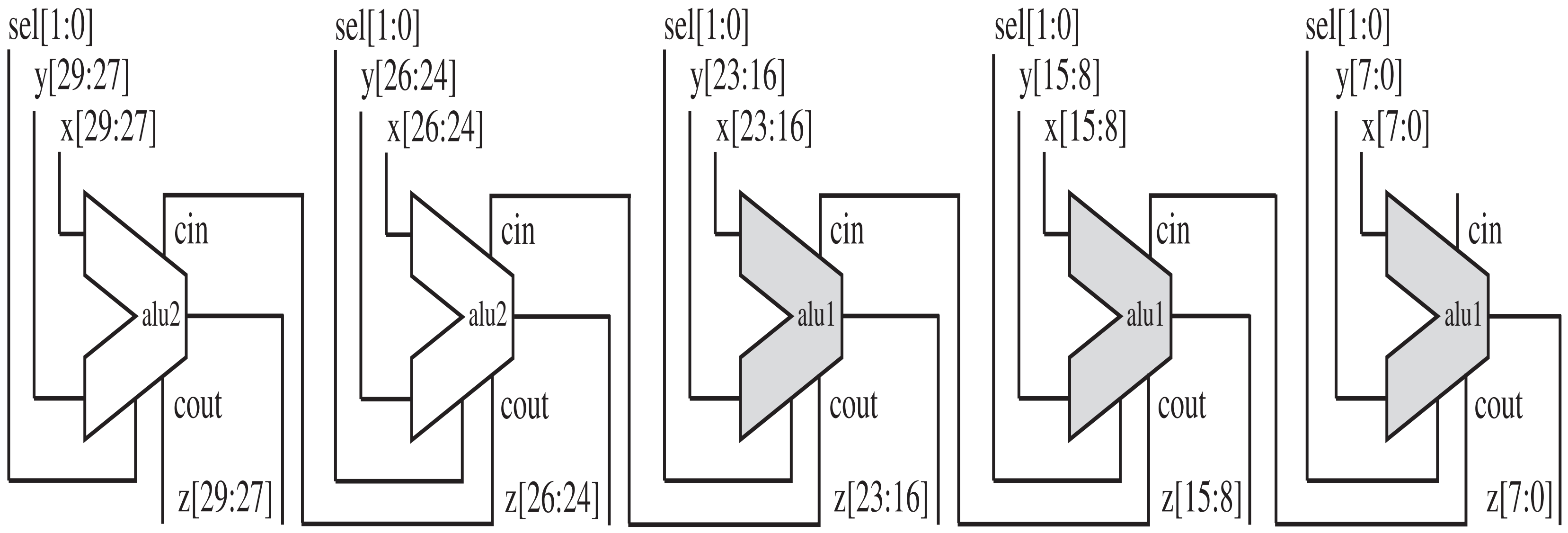
```
              .
              .
              .
   SIGNAL carry:  bit_vector (3 DOWNTO 0);
BEGIN
   carry (0)  <= cin;
   cout  <= carry (3);

   wide:  FOR n IN 0 TO 2 GENERATE
      FOR alu:  alu_carry USE ENTITY work.alu_carry (behave);
   BEGIN
      alu:  alu_carry
         GENERIC MAP(   width => 8)
         PORT MAP(   x(7+n*8 DOWNTO n*8),
                     y(7+n*8 DOWNTO n*8),
                     carry(n),
                     sel,
                     z(7+n*8 DOWNTO n*8),
                     carry(n+1));
   END GENERATE wide;
END ARCHITECTURE wide1;
```

## IF-Generate Statements

**Example: 32 (Class)** *Assume we have configured two instance types of the entity alu_carry in Example 30 as shown and would like to use them in generating the wider ALU shown below.*





## IF-Generate Statements (Cont.)

**Example 32: (Cont.)**

```
LIBRARY ieee;
USE ieee.numeric_bit.ALL;
ENTITY alu_wide2 IS
      PORT( x, y:  IN signed (29 DOWNTO 0);
            cin:  IN bit;
            sel:  IN unsigned (1 DOWNTO 0);
            z:  OUT signed (29 DOWNTO 0);
            cout:  OUT bit);
END ENTITY alu_wide2;

ARCHITECTURE wide2 OF alu_wide2 IS
   COMPONENT alu_carry IS
      GENERIC(  width:  positive := 8);
      PORT(  a, b:  IN signed (width-1 DOWNTO 0);
             cin:  IN bit;
             control:  IN unsigned (1 DOWNTO 0);
             d:  OUT signed (width-1 DOWNTO 0);
             cout:  OUT bit);
   END COMPONENT alu_carry;
            .
            .
            .
```

## IF-Generate Statements (Cont.)

**Example 32: (Cont.)**

```
        ⋮
  SIGNAL carry:  bit_vector (5 DOWNTO 0);
BEGIN
  carry (0) <= cin;
  cout <= carry (5);
  outer:  FOR n IN 0 TO 4 GENERATE
    inner_1:  IF n < 3 GENERATE
      FOR alu:  alu_carry USE ENTITY work.alu_carry (behave);
    BEGIN
      alu:  alu_carry
        GENERIC MAP( width => 8)
        PORT MAP( x(7+n*8 DOWNTO n*8),
                  y(7+n*8 DOWNTO n*8),
                  carry(n),
                  sel,
                  z(7+n*8 DOWNTO n*8),
                  carry(n+1));
    END GENERATE inner_1;
        ⋮
```

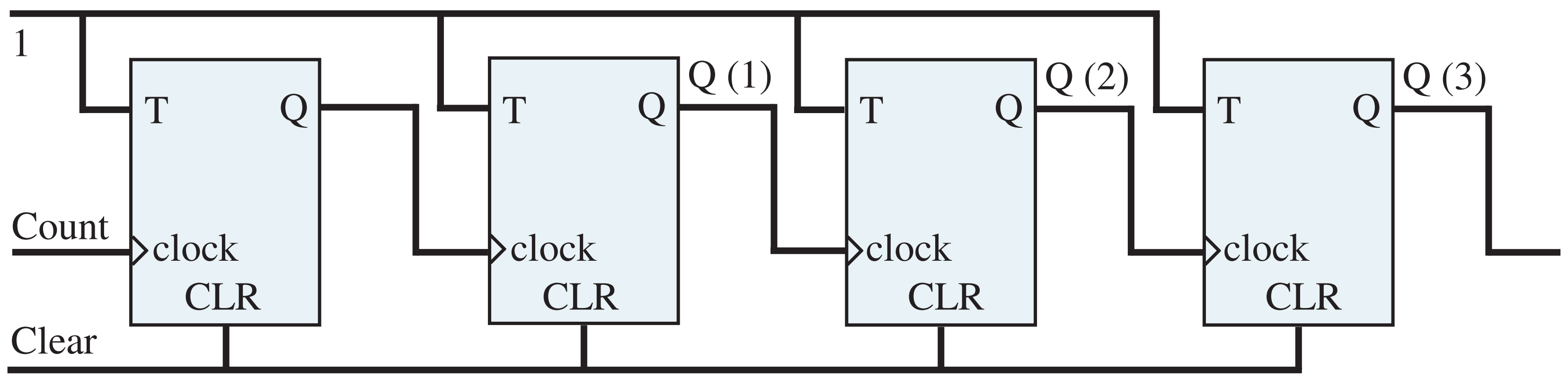
## IF-Generate Statements (Cont.)

**Example 32: (Cont.)**

```
          ⋮
    inner_2:  IF n >= 3 GENERATE
      FOR alu:  alu_carry USE ENTITY work.alu_carry (behave);
    BEGIN
      alu:  alu_carry
        GENERIC MAP( width => 3)
        PORT MAP( x(26+(n-3)*3 DOWNTO 24+(n-3)*3),
                  y(26+(n-3)*3 DOWNTO 24+(n-3)*3),
                  carry(n),
                  sel,
                  z(26+(n-3)*3 DOWNTO 24+(n-3)*3),
                  carry(n+1));
    END GENERATE inner_2;
  END GENERATE outer;
END ARCHITECTURE wide2;
```

## IF-Generate Statements (Cont.)

**Example: 33 (Class)** *Write entity and architecture declarations for the shown ripple counter.*



## IF-Generate Statements (Cont.)

**Example 33: (Cont.)**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY counter IS
   GENERIC (width:  positive := 4);
   PORT (count, clear:  IN std_logic;
         q:  OUT std_logic_vector (0 TO width-1));
END ENTITY counter;

ARCHITECTURE behavioral OF counter IS
   COMPONENT t_ff IS
      PORT(    t, clk, clr:  IN std_logic;
               q:  OUT std_logic);
   END COMPONENT t_ff;
   SIGNAL s:  std_logic_vector(0 TO width-1);
BEGIN
            ⋮
```

## IF-Generate Statements (Cont.)

**Example 33: (Cont.)**

```
              ⋮
  outer:  FOR j IN 0 TO width-1 GENERATE
    inner_1:  IF j = 0 GENERATE
      FOR t1:  t_ff USE ENTITY WORK.t_ff (behav);
    BEGIN
      t1:  t_ff PORT MAP ( '1', count, clear, s(j));
    END GENERATE inner_1;

    inner_2:  IF j > 0 GENERATE
      FOR t1:  t_ff USE ENTITY WORK.t_ff (behav);
    BEGIN
      t1:  t_ff PORT MAP ( '1', s(j-1), clear, s(j));
    END GENERATE inner_2;
  END GENERATE outer;

  q <= s;
END ARCHITECTURE behavioral;
```

## IF-Generate Statements (Cont.)

**Example: 34 (Home)** *Given an n-bit binary number (I), show that the logic equations to convert it to its equivalent gray number (O) are as follows:*

$$\begin{aligned} O_{n-1} &= I_{n-1} \\ O_i &= I_{i+1} \oplus I_i, \quad 0 \leq i < n-1 \end{aligned}$$

**Example: 35 (Class)** *Write entity and architecture declarations for a circuit that converts a binary input to its gray equivalent.*

```
ENTITY bin2gray IS
   GENERIC (n:  positive := 4);
   PORT (input:  IN bit_vector (n-1 DOWNTO 0);
         output:  IN bit_vector (n-1 DOWNTO 0));
END ENTITY bin2gray;
            ⋮
```

## IF-Generate Statements (Cont.)

**Example 35: (Cont.)**

```
        ⋮
ARCHITECTURE gen_process of bin2gray IS BEGIN
  outer:  FOR index IN n-1 DOWNTO 0 GENERATE
    msb:  IF index = n-1 GENERATE
      msb_p:  PROCESS (input) IS BEGIN
        output(n-1) <= input(n-1);
      END PROCESS msb_p;
    END GENERATE msb;
    bits:  IF index < n-1 GENERATE
      bits_p:  PROCESS (input) IS BEGIN
        output(index) <= input(index + 1) XOR input(index);
      END PROCESS bits_p;
    END GENERATE bits;
  END GENERATE outer;
END ARCHITECTURE gen_process;
```