Chapter 5
Sequential Constructs for RT Level Descriptions

Sequential Constructs for RT Level Descriptions

5.1 Process Statement

5.1.1 Declarative Part of a Process

5.1.2 Statement Part of a Process

5.1.3 Process Sensitivity List

5.1.4 Postponed Processes

5.1.5 Passive Processes

5.2 Sequential Wait Statements

5.3 VHDL Subprograms

5.3.1 Function Definition

5.3.2 Procedure Definition

5.3.3 Language Aspects of Subprograms

5.3.4 Nesting Subprograms

Copyright Z. Navabi, 2007

Sequential Constructs for RT Level Descriptions

5.4 VHDL Library Structure
- 5.4.1 Creating Libraries
- 5.4.2 Using Libraries

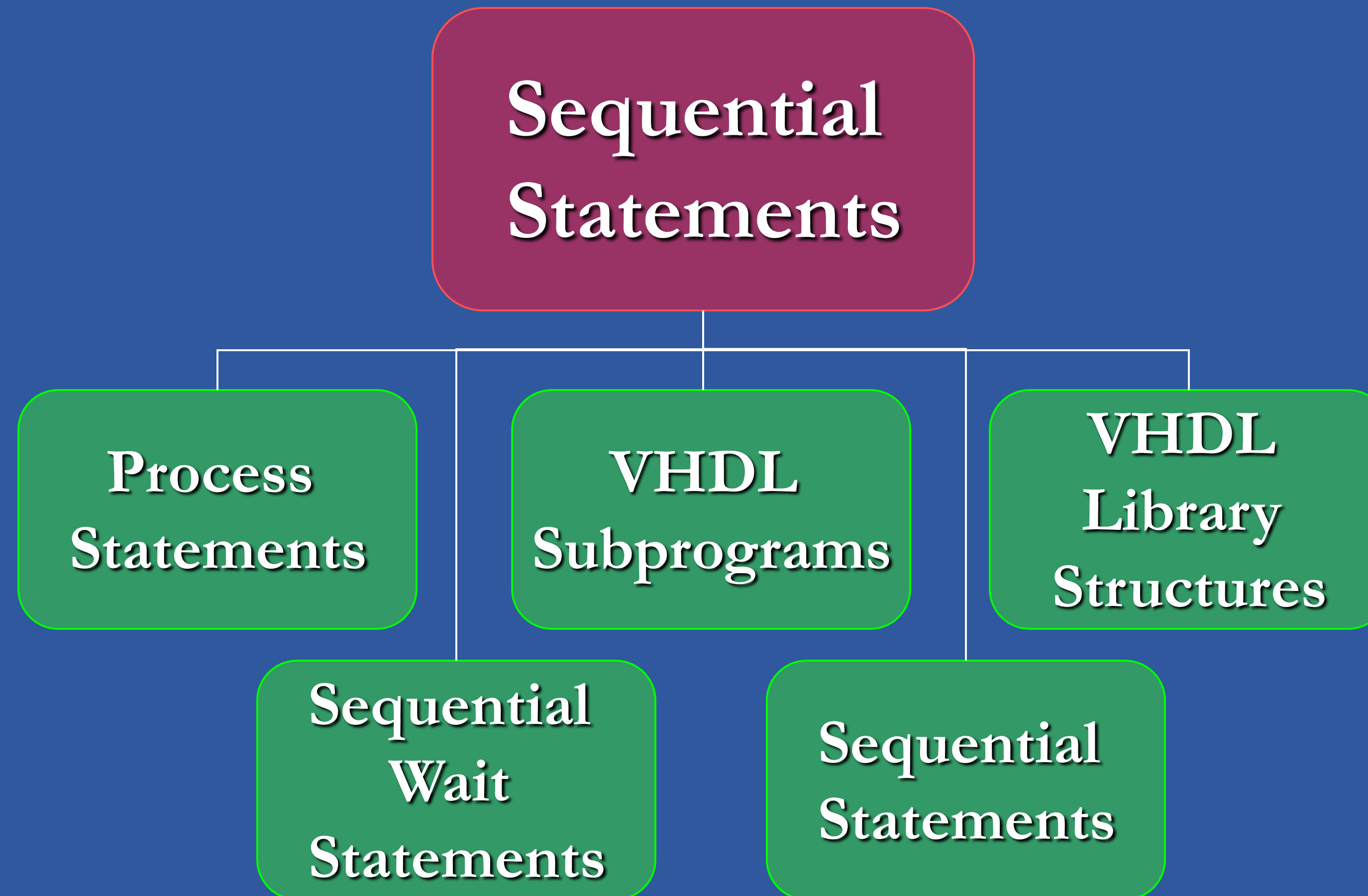
5.5 Packaging Utilities and Components
- 5.5.1 A Package of Utilities
- 5.5.2 A Package of Components

5.6 Sequential Statements
- 5.6.1 If Statement
- 5.6.2 Loop Statement
- 5.6.3 Case Statement
- 5.6.4 Assertion Statement

5.7 Summary

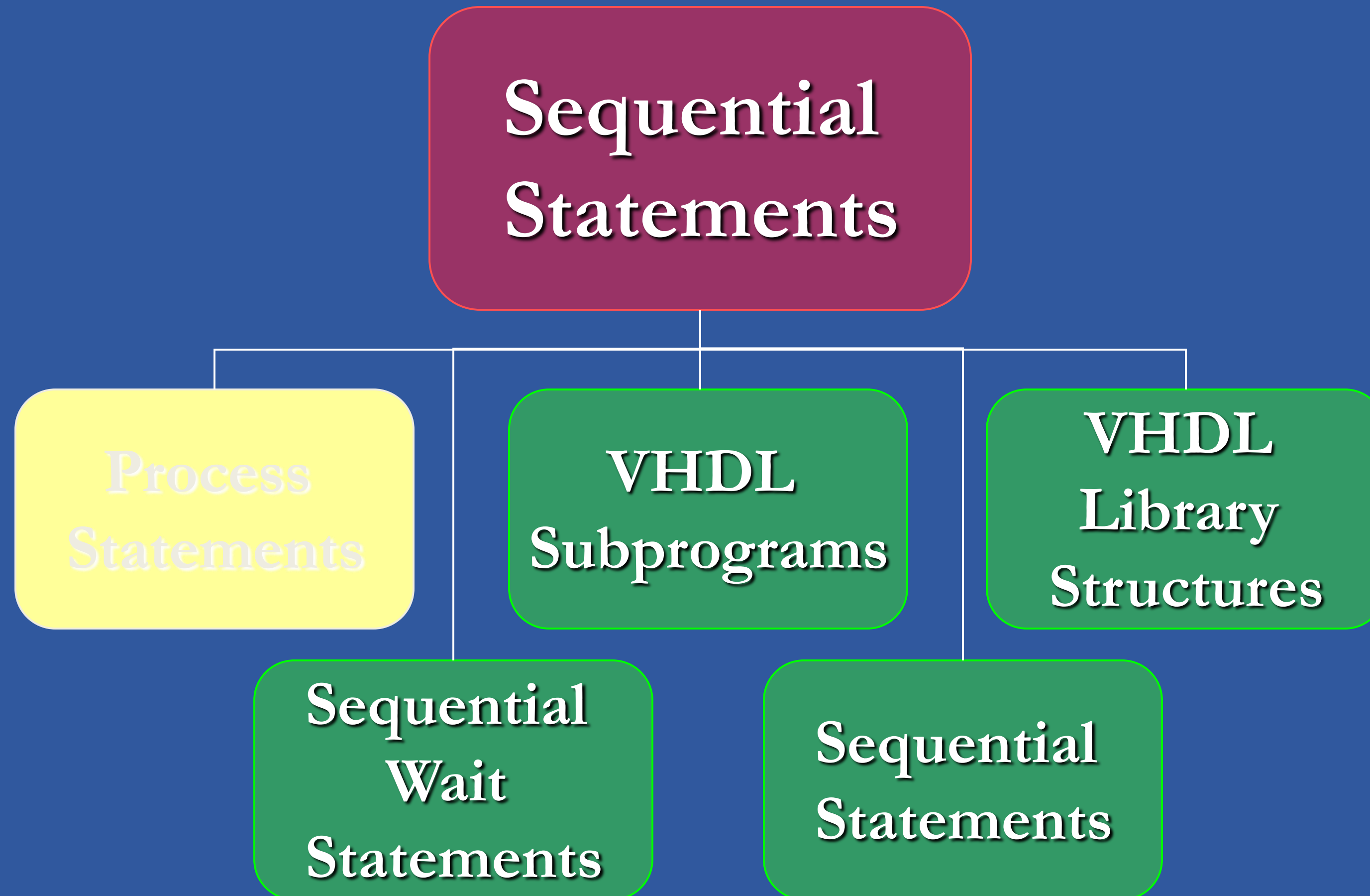
Copyright Z. Navabi, 2007

Sequential Statements



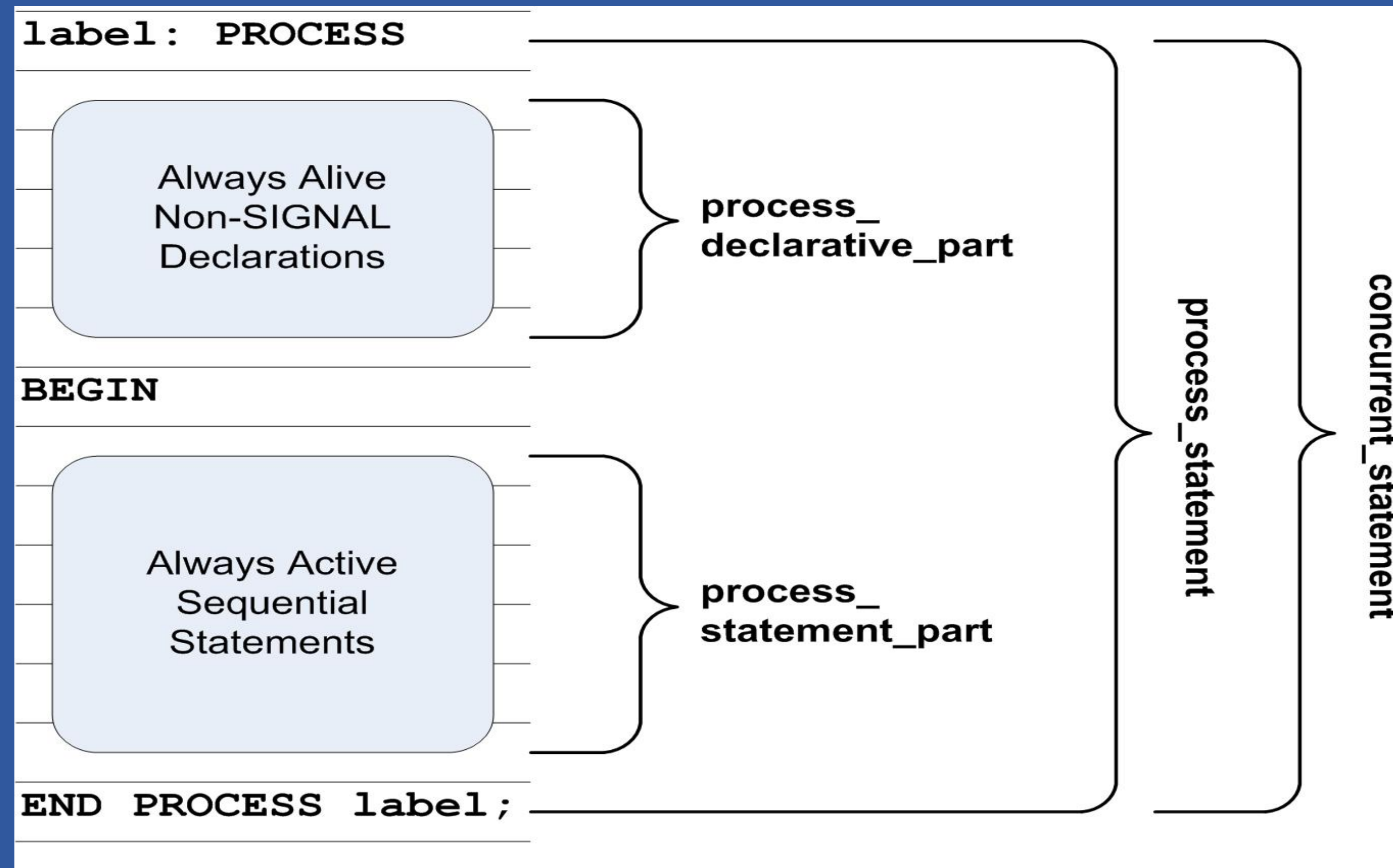
Copyright Z. Navabi, 2007

Process Statements



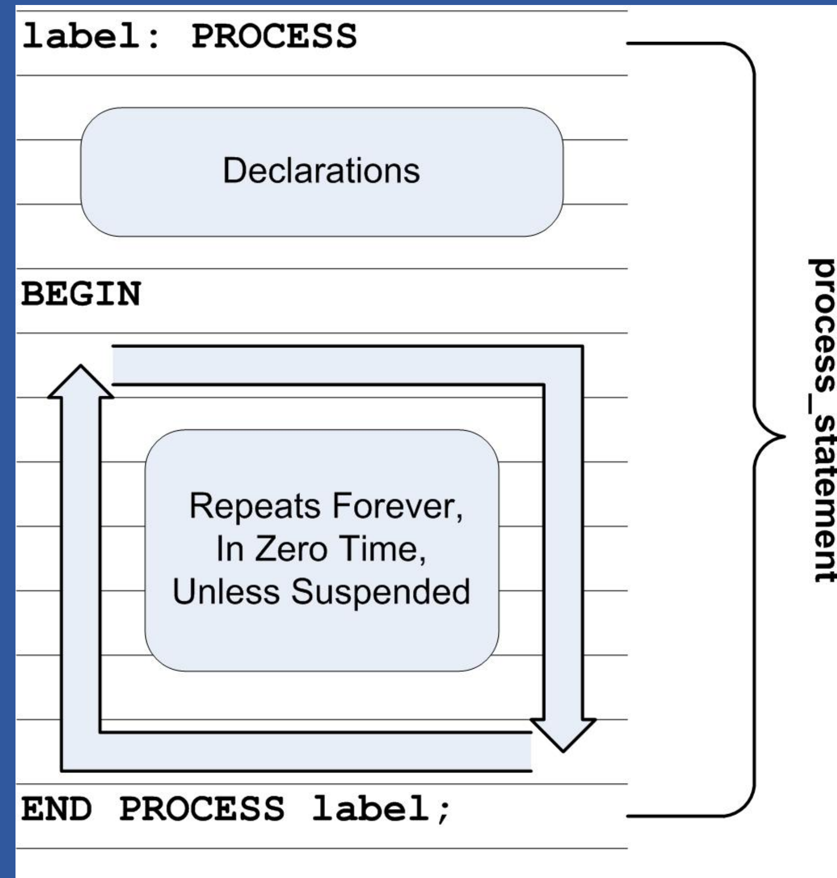
Copyright Z. Navabi, 2007

Process Statements



- **A Process Statement Block Diagram**

Statement Part of a Process



- **A Process Runs in Zero Time, Repeats Forever, Unless Suspended**

Statement Part of a Process

```
ARCHITECTURE sequentiality_demo OF partial_process IS
BEGIN
   PROCESS
   BEGIN
     . . .
     x <= a;
     y <= b;
     . . .
   END PROCESS;
END sequentiality_demo;
```

- Zero Distance Signal Assignments

Statement Part of a Process

```
ARCHITECTURE data_availability_demo OF partial_process
IS
        SIGNAL x : BIT :='0';
BEGIN
    PROCESS BEGIN
        ...
        x <= '1';
        IF x = '1' THEN
            Perform_action_1
        ELSE
            Perform_action_2
        END IF;
        ...
    END PROCESS;
END data_availability_demo;
```

- Partial Code for Demonstrating Delay in Assignment of Values to Signals

Copyright Z. Navabi, 2007

Process Sensitivity List

```
ENTITY multiplexer IS
   PORT (a, b, s : IN BIT; w : OUT BIT);
END ENTITY;
--
ARCHITECTURE processing OF multiplexer IS
BEGIN
   com: PROCESS (a, b, s) BEGIN
      IF s='0' THEN w <= a AFTER 1.4 NS;
      ELSE w <= b AFTER 1.5 NS;
      END IF;
   END PROCESS com;
END ARCHITECTURE processing;
```

- Multiplexer Described Using a Process with Sensitivity List

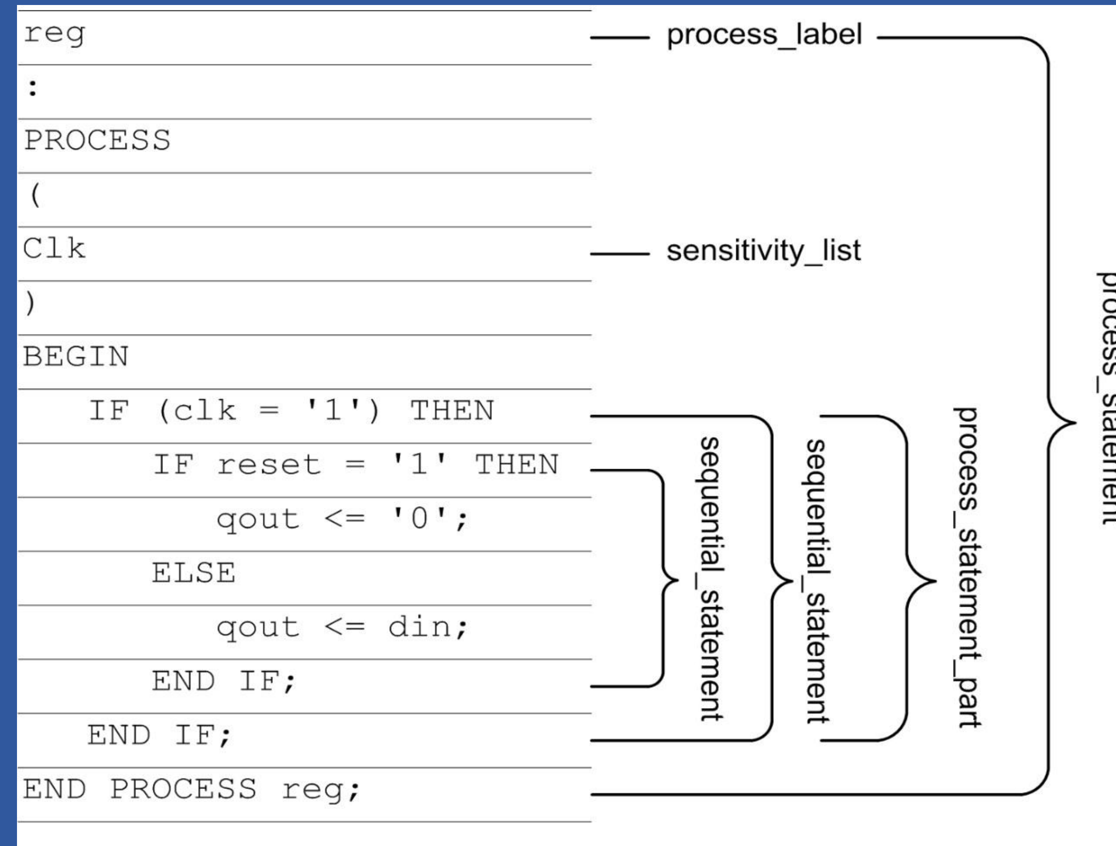
Copyright Z. Navabi, 2007

Process Sensitivity List

```
ENTITY flipflop IS
   PORT (reset, din, clk : IN BIT; qout : OUT BIT);
END ENTITY;
--
ARCHITECTURE synch_process OF flipflop IS BEGIN
   reg: PROCESS (clk) BEGIN
      IF (clk = '1') THEN
         IF reset = '1' THEN qout <= '0';
         ELSE qout <= din;
         END IF;
      END IF;
   END PROCESS reg;
END ARCHITECTURE synch_process;
```

- *flipflop* Using Process with Sensitivity List

Process Sensitivity List



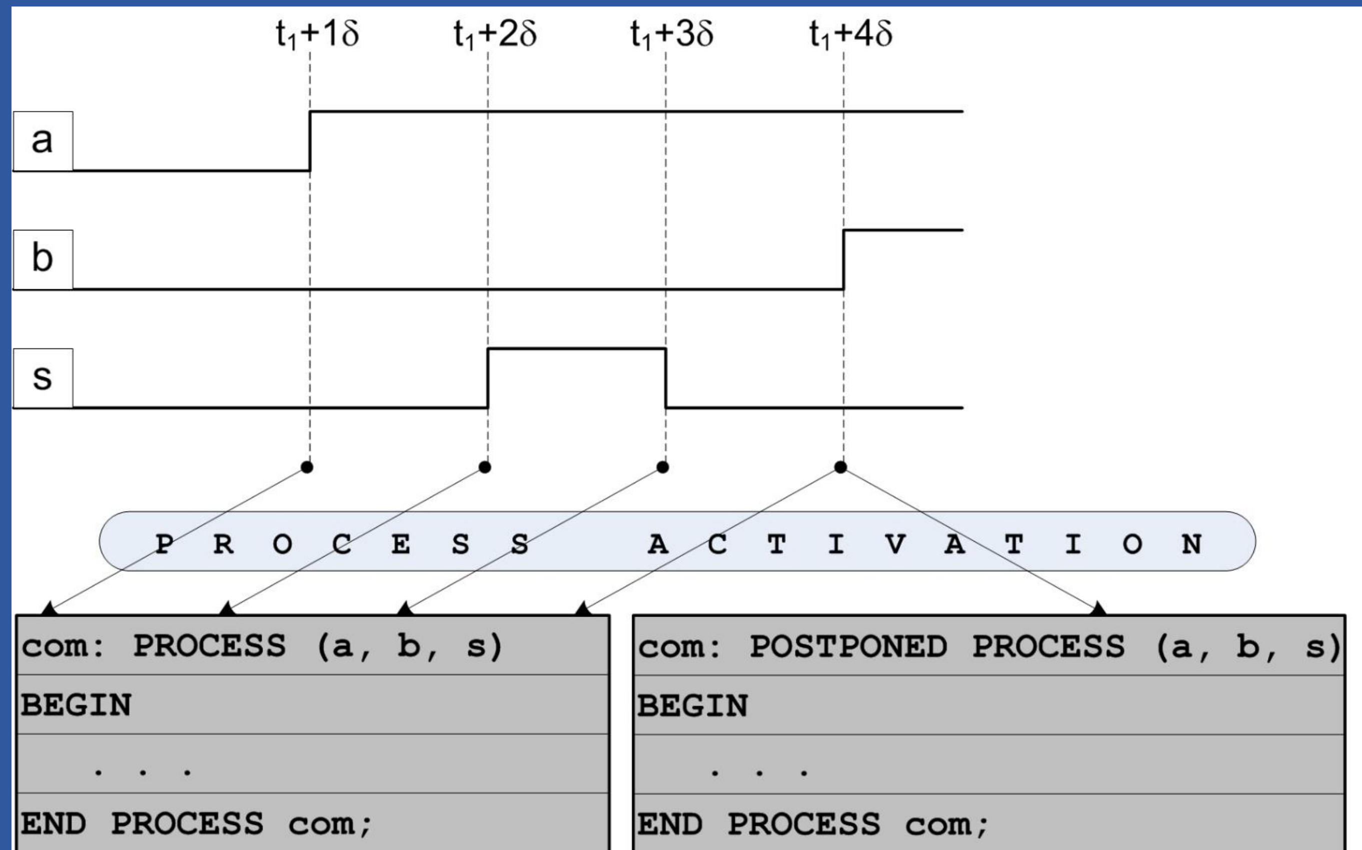
- **Syntax of Process with Sensitivity List**

Process Sensitivity List

```
ARCHITECTURE asynch_process OF flipflop IS
BEGIN
   reg: PROCESS (clk, reset) BEGIN
      IF reset = '1' THEN
         qout <= '0' AFTER 1.2 NS;
      ELSIF (clk = '1' AND clk'EVENT) THEN
         qout <= din AFTER 1.3 NS;
      END IF;
   END PROCESS reg;
END ARCHITECTURE asynch_process;
```

- Process Statement Implementing Asynchronous Control

Postponed Processes

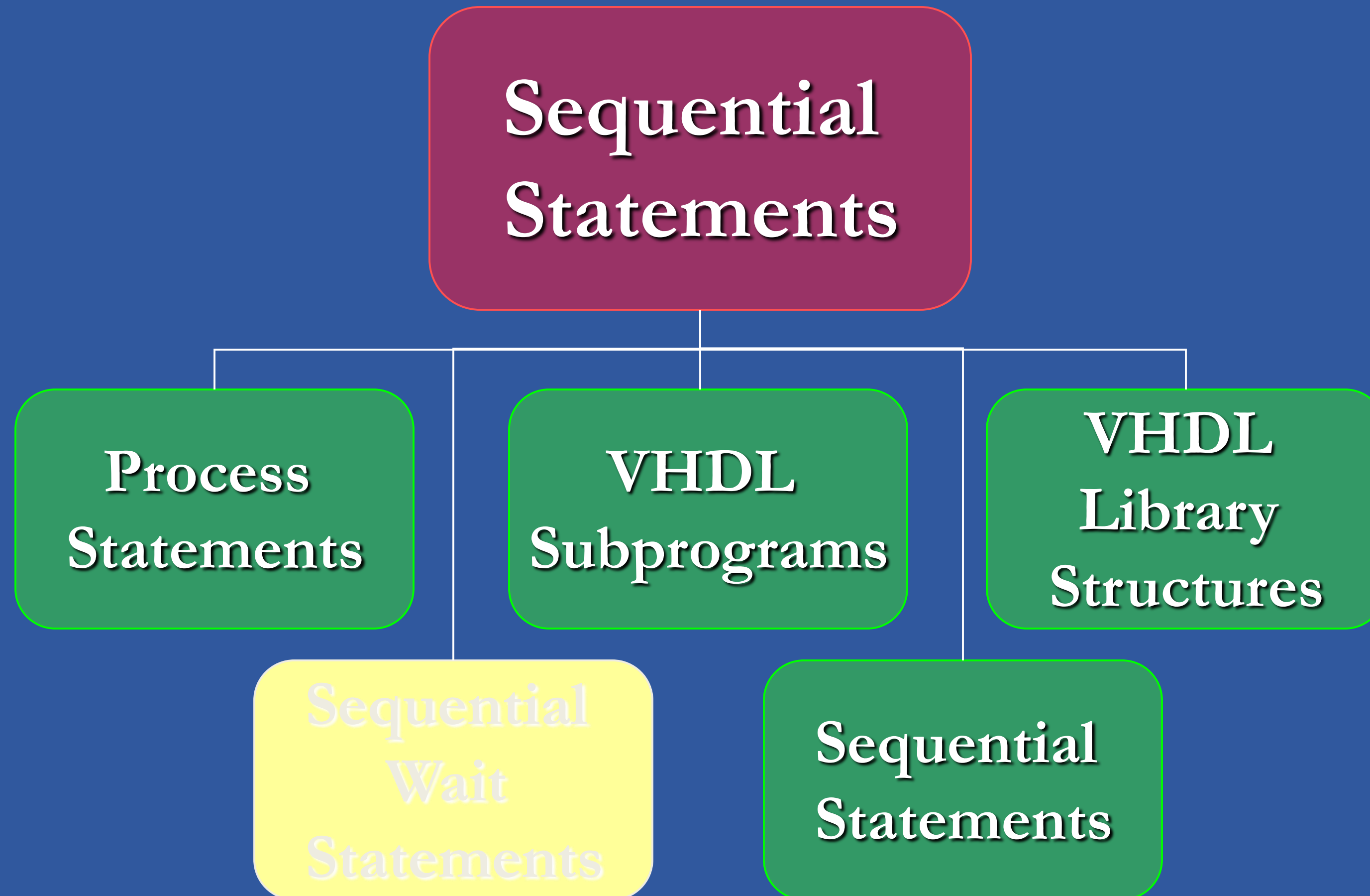


- **Activation of a Postponed Process List**

Passive Processes

```
ENTITY flipflop IS
   PORT (reset, din, clk : IN BIT; qout : OUT BIT);
BEGIN
   timing: PROCESS (clk, reset, din)
      VARIABLE t_clk1, t_clk0 : TIME := 0 NS;
      VARIABLE t_clkon, t_clkoff : TIME := 0 NS;
   BEGIN
      IF clk'EVENT THEN
         IF clk = '1' THEN --rising edge
            t_clk1 := NOW;
            t_clkoff := t_clk1 - t_clk0;
         ELSE              --faling edge
            t_clk0 := NOW;
            t_clkon := t_clk0 - t_clk1;
         END IF;
      END IF;
      IF t_clkon /= t_clkoff THEN
         REPORT "Not 50% duty cycle: On:"
                & TIME'IMAGE(t_clkon) & "Off:"
                & TIME'IMAGE(t_clkoff);
      END IF;
      IF clk = '1' AND din'EVENT THEN
         REPORT "The din input changed while clk was '1'";
      END IF;
   END PROCESS timing;
END ENTITY;
```

Process Statements



Sequential Wait Statements

```
WAIT FOR waiting_time;
WAIT ON waiting_sensitivity_list;
WAIT UNTIL waiting_condition;
WAIT FOR 0 any_time_unit;
WAIT;
```

Sequential Wait Statements

```
ARCHITECTURE process_wait OF multiplexer IS
BEGIN
    com: PROCESS
    BEGIN
        IF s='0' THEN
            w <= a AFTER 1.4 NS;
        ELSE
            w <= b AFTER 1.5 NS;
        END IF;
        WAIT ON a, b, s;
    END PROCESS com;
END ARCHITECTURE process_wait;
```

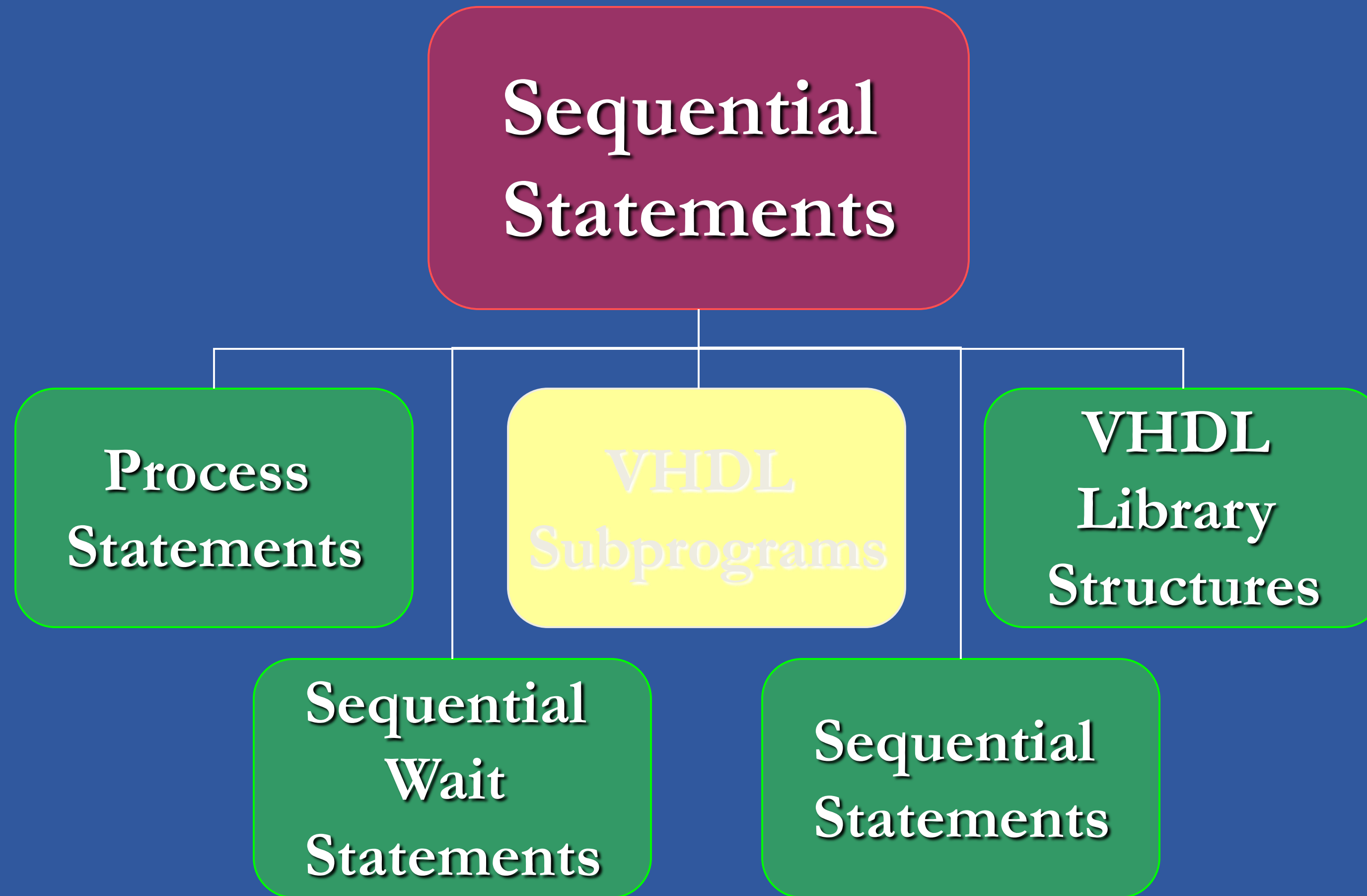
- Process with WAIT

Sequential Wait Statements

```
ARCHITECTURE synch_waituntil OF flipflop IS
BEGIN
   reg: PROCESS
   BEGIN
      IF reset = '1' THEN
         qout <= '0' AFTER 1.2 NS;
      ELSE
         qout <= din AFTER 1.3 NS;
      END IF;
      WAIT FOR 1.5 NS;
      WAIT UNTIL clk = '1';
   END PROCESS reg;
END ARCHITECTURE synch_waituntil;
```

- Multiple WAIT Statements

Process Statements

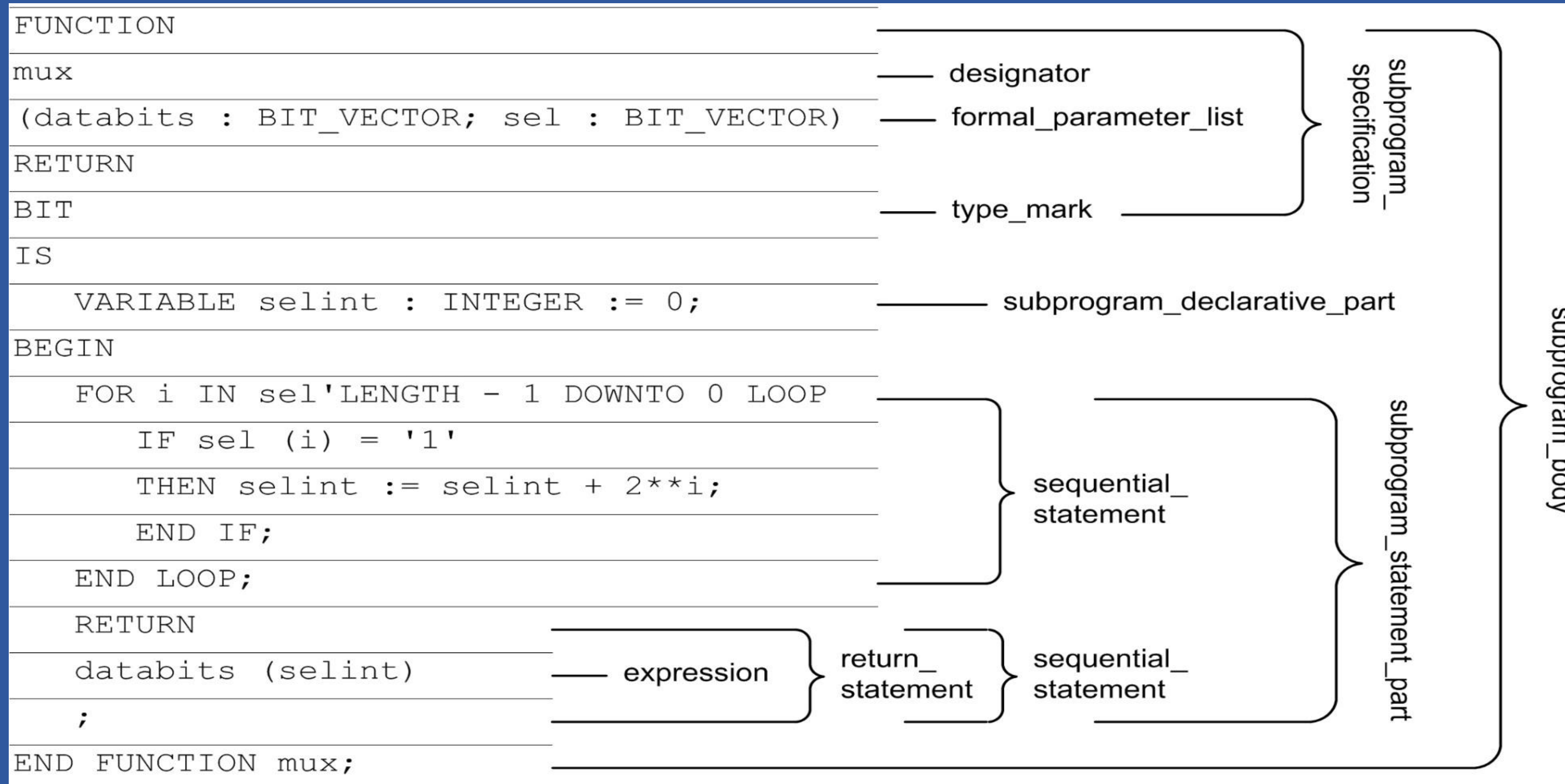


Function Definition

```
FUNCTION mux
    (databits : BIT_VECTOR; sel : BIT_VECTOR)
RETURN BIT IS
    VARIABLE selint : INTEGER := 0;
BEGIN
    FOR i IN sel'LENGTH - 1 DOWNTO 0 LOOP
        IF sel (i) = '1' THEN
            selint := selint + 2**i;
        END IF;
    END LOOP;
    RETURN databits (selint);
```

- A Simple Function Definition

Function Definition



- **Function Syntax Details**

Function Definition

```
ARCHITECTURE functional OF multiplexer IS
   FUNCTION mux (databits : BIT_VECTOR; . . .
       .
       .
       .
   END FUNCTION mux;

   SIGNAL sel : BIT_VECTOR (0 DOWNTO 0);
BEGIN
   sel(0) <= s;
   w <= mux ((a,b), sel) AFTER 8 NS;
END ARCHITECTURE functional;
```

- Calling a Function

Procedure Definition

```
PROCEDURE consecutive_data
    (SIGNAL target : OUT BIT_VECTOR;
     CONSTANT ti : TIME; CONSTANT n : INTEGER)
IS
    VARIABLE data : BIT_VECTOR (target'RANGE);
    VARIABLE sum, carry : BIT;
BEGIN
    FOR i IN 1 TO n LOOP
        carry := '1';
        FOR j IN data'REVERSE_RANGE LOOP
            sum := data (j) XOR carry;
            carry := data (j) AND carry;
            data (j) := sum;
        END LOOP;
        target <= TRANSPORT data AFTER ti * i;
    END LOOP;
END PROCEDURE consecutive_data;
```

Concurrent Procedure Calls

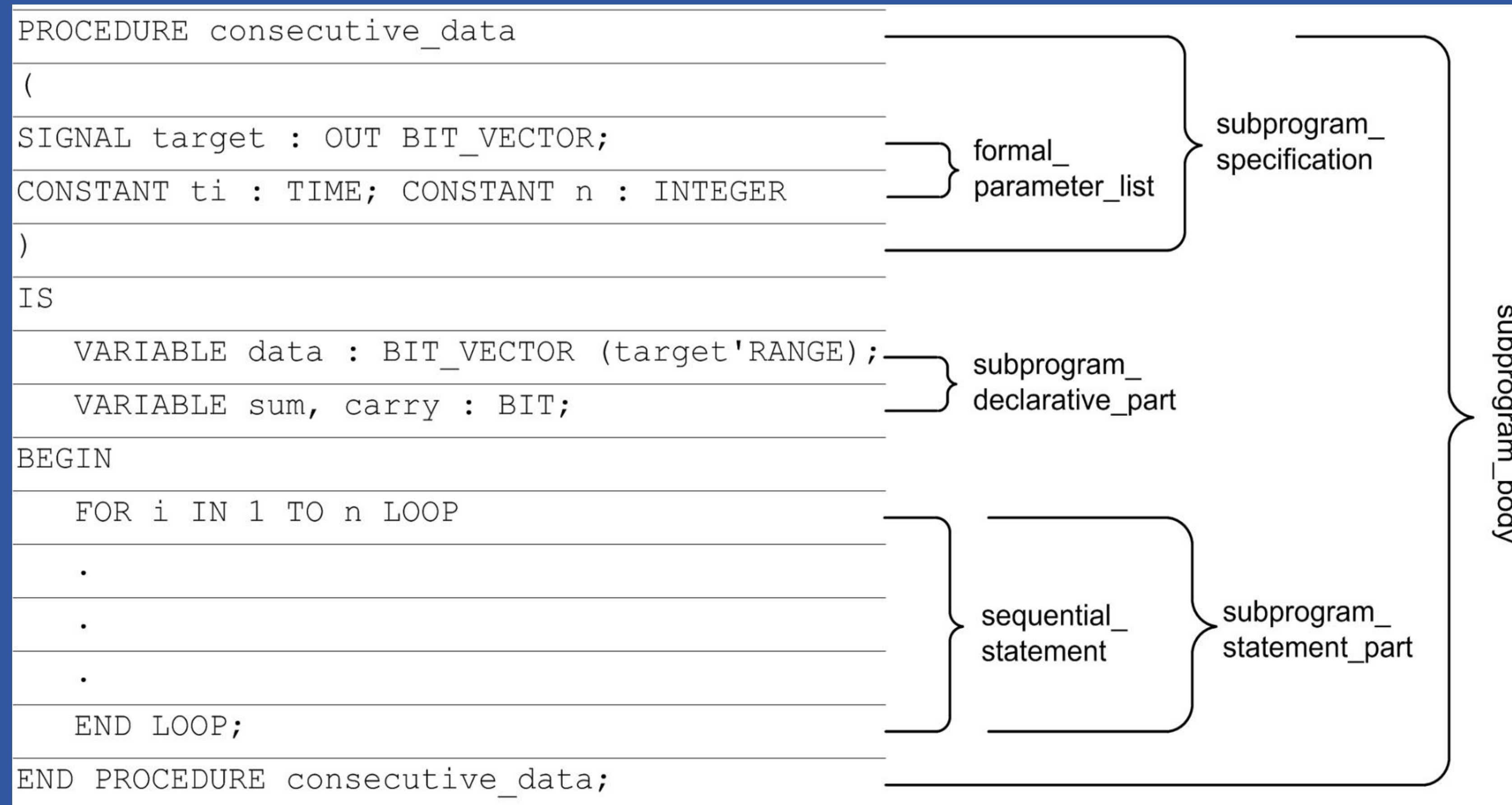
```
ARCHITECTURE procedural OF multiplexer8_tester IS
   PROCEDURE consecutive_data
      (SIGNAL target : OUT BIT_VECTOR;
       CONSTANT ti : TIME; CONSTANT n : INTEGER)
   IS      .
           .
   END PROCEDURE consecutive_data;
   SIGNAL a, b, w2 : BIT_VECTOR (7 DOWNTO 0);
   SIGNAL s : BIT;
   SIGNAL sel : BIT_VECTOR (0 TO 0);
BEGIN
   UUT2: ENTITY WORK.multiplexer8 (direct)
         PORT MAP (a, b, s, w2);
   consecutive_data (a, 123 NS, 6);
   consecutive_data (b, 79 NS, 6);
   consecutive_data (sel, 119 NS, 8);
   s <= sel(0);
END ARCHITECTURE procedural;
```

Procedure Definition

```
PROCEDURE onehot_data
    (SIGNAL target : OUT BIT_VECTOR;
     CONSTANT ti : TIME; CONSTANT n : INTEGER)
IS
   VARIABLE data : BIT_VECTOR (target'RANGE);
   VARIABLE i : INTEGER := 0;
BEGIN
   data (0) := '1';
   WHILE i < n LOOP
      data := data ROR 1;
      target <= TRANSPORT data AFTER ti * i;
      i := i + 1;
   END LOOP;
END PROCEDURE onehot_data;
```

- Using While Loop

Language Aspects of Subprograms



- **Details of a Subprogram Body**

Nesting Subprograms

```
FUNCTION int (invec : BIT_VECTOR) RETURN INTEGER IS
   VARIABLE tmp : INTEGER := 0;
BEGIN
   FOR i IN invec'LENGTH - 1 DOWNTO 0 LOOP
      IF invec (i) = '1' THEN
         tmp := tmp + 2**i;
      END IF;
   END LOOP;
   RETURN tmp;
END FUNCTION int;
FUNCTION mux (databits : BIT_VECTOR; sel : BIT_VECTOR)
RETURN BIT IS
BEGIN
   RETURN databits (int(sel));
END FUNCTION mux;
```

- Using a Function in Another

Nesting Subprograms

```
PROCEDURE inc (VARIABLE invec : INOUT BIT_VECTOR) IS
   VARIABLE sum, carry : BIT;
BEGIN
   carry := '1';
   FOR j IN invec'REVERSE_RANGE LOOP
      sum := invec (j) XOR carry;
      carry := invec (j) AND carry;
      invec (j) := sum;
   END LOOP;
END PROCEDURE inc;
--
PROCEDURE consecutive_data
   (SIGNAL target : OUT BIT_VECTOR;
    CONSTANT ti : TIME; CONSTANT n : INTEGER) IS
    VARIABLE data : BIT_VECTOR (target'RANGE);
BEGIN
   FOR i IN 1 TO n LOOP
      inc (data);
      target <= TRANSPORT data AFTER ti * i;
   END LOOP;
END PROCEDURE consecutive_data;
```

Nesting Subprograms

```
FUNCTION dcd (bin : BIT_VECTOR) RETURN BIT_VECTOR IS
   VARIABLE tmp : BIT_VECTOR(0 TO 2**bin'LENGTH - 1);
BEGIN
   tmp := (OTHERS => '0');
   tmp (int(bin)) := '1';
   RETURN tmp;
END FUNCTION dcd;
```

- Using *int* Function

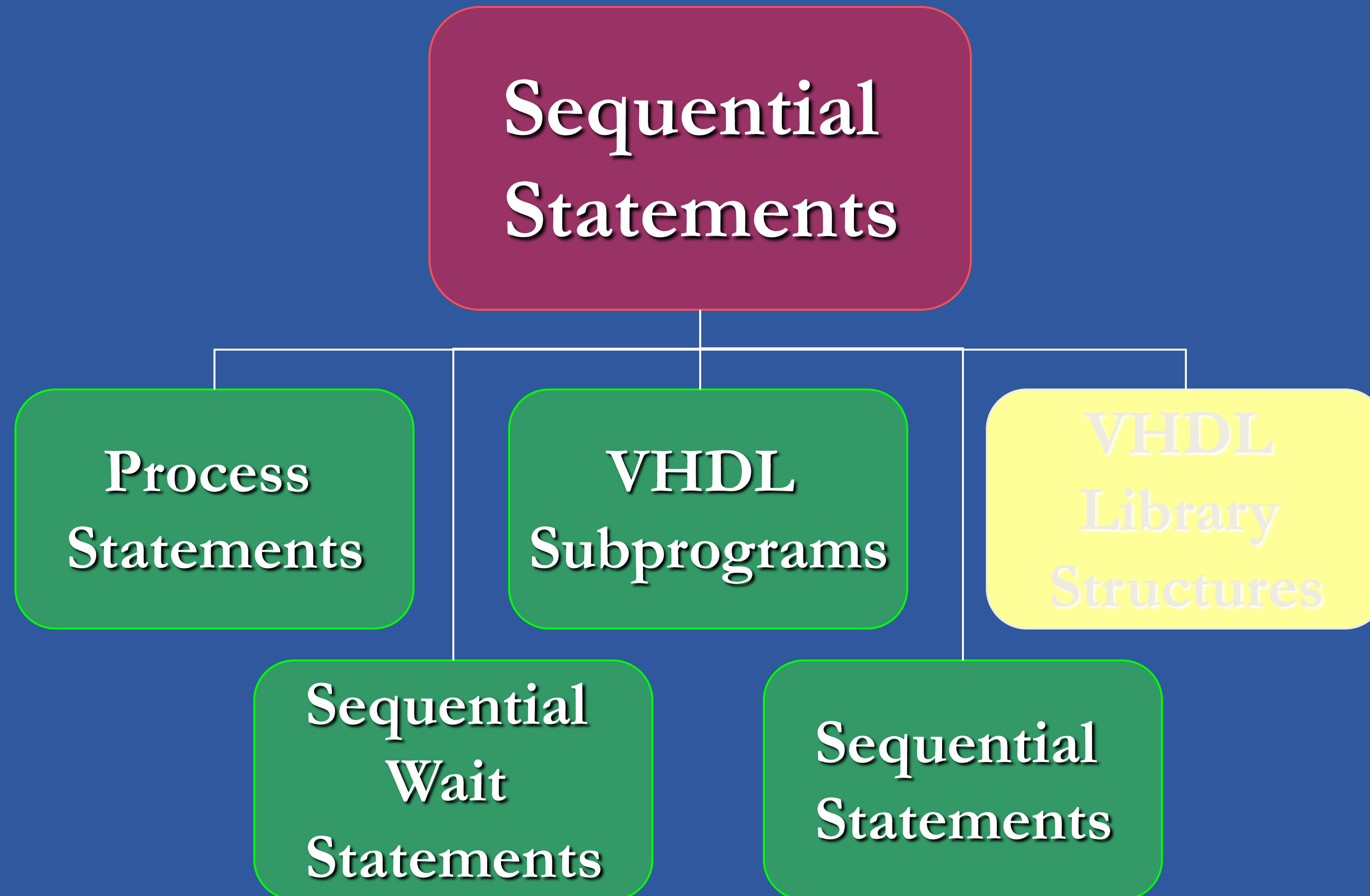
Nesting Subprograms

```
ENTITY decoder IS
   PORT (bin_in : IN BIT_VECTOR; en : IN BIT;
         dcd_ou : OUT BIT_VECTOR);
END ENTITY decoder;
--
ARCHITECTURE functional OF decoder IS
BEGIN
   dcd_ou <= dcd (bin_in) WHEN en = '1'
                          ELSE (OTHERS => '0');
END ARCHITECTURE functional;
```

- Using dcd Function in a Concurrent Statement

Copyright Z. Navabi, 2007

Process Statements



A Package of Utilities

```
PACKAGE BasicUtilities IS
   FUNCTION int (invec : BIT_VECTOR) RETURN INTEGER;
   FUNCTION mux (databits : BIT_VECTOR;
                      sel : BIT_VECTOR) RETURN BIT;
   FUNCTION bin (inint, size : INTEGER)
                                 RETURN BIT_VECTOR;
   FUNCTION dcd (bin : BIT_VECTOR) RETURN BIT_VECTOR;
   PROCEDURE consecutive_data
      (SIGNAL target : OUT BIT_VECTOR;
       CONSTANT ti : TIME; CONSTANT n : INTEGER);
END PACKAGE BasicUtilities;
```

- An Example Package Declaration

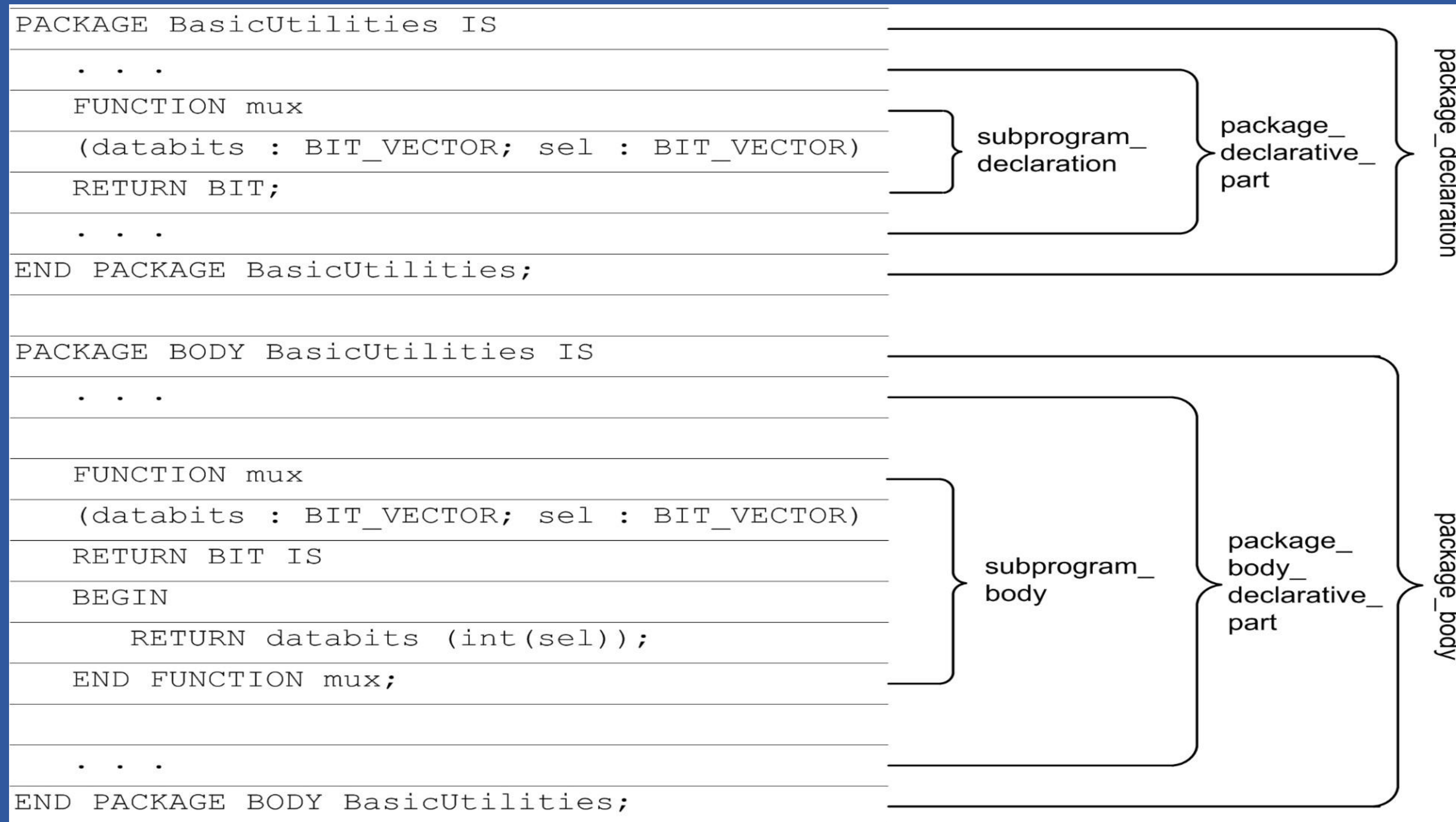
Copyright Z. Navabi, 2007

Subprogram Definition in Package Body

```
PACKAGE BODY BasicUtilities IS
FUNCTION int : see Figure 4.37
FUNCTION mux : see Figure 4.31
FUNCTION bin (inint, size : INTEGER) RETURN BIT_VECTOR IS
   VARIABLE tmpi : INTEGER := inint;
   VARIABLE tmpb : BIT_VECTOR (size - 1 DOWNTO 0);
BEGIN
   tmpb := (OTHERS => '0');
   FOR i IN 0 TO size - 1 LOOP
      IF ((tmpi MOD 2) = 1) THEN
         tmpb(i) := '1';
      END IF;
      tmpi := tmpi / 2;
   END LOOP;
   RETURN tmpb;
END FUNCTION bin;
--   PROCEDURE inc: see Figure 5.21
--   PROCEDURE consecutive_data: see Figure 5.21
END PACKAGE BODY BasicUtilities;
```


Copyright Z. Navabi, 2007

A Package of Utilities



- Package Declaration and Body Syntax

Copyright Z. Navabi, 2007

A Package of Components

```
LIBRARY utilities;
USE utilities.BasicUtilities.ALL;
ENTITY alu4function IS
   PORT (ai, bi : IN BIT_VECTOR;
         mode : IN BIT_VECTOR (1 DOWNTO 0);
         aluout : OUT BIT_VECTOR);
END ENTITY;
```

- A Design Unit Compiled in our *GenericParts* Library (continued)

Copyright Z. Navabi, 2007

A Package of Components

```
ARCHITECTURE customizable OF alu4function IS
   CONSTANT size : INTEGER := ai'LENGTH;
BEGIN
   PROCESS (ai, bi, mode) BEGIN
      CASE BIT_VECTOR (mode) IS
         WHEN "00" =>
            aluout <= bin(int(ai) + int(bi), size);
         WHEN "01" =>
            aluout <= bin(int(ai) - int(bi), size);
         WHEN "10" =>
            aluout <= ai AND bi;
         WHEN "11" =>
            aluout <= ai OR bi;
         WHEN OTHERS =>
            aluout <= bin(0, size);
      END CASE;
   END PROCESS;
END ARCHITECTURE customizable;
```

- A Design Unit Compiled in our *GenericParts* Library

A Package of Components

```
LIBRARY utilities;
USE utilities.BasicUtilities.ALL;
ENTITY dregister IS
   PORT (rst, clk : IN BIT; regin : IN BIT_VECTOR;
         regout : OUT BIT_VECTOR);
END ENTITY;
--
ARCHITECTURE synchronous OF dregister IS
   CONSTANT size : INTEGER := regin'LENGTH;
BEGIN
   reg: PROCESS (clk) BEGIN
      IF (clk = '1') THEN
         IF rst = '1' THEN regout <= bin (0, size);
         ELSE regout <= regin;
         END IF;
      END IF;
   END PROCESS reg;
END ARCHITECTURE synchronous;
```

- D-Register Compiled in *GenericParts*

A Package of Components

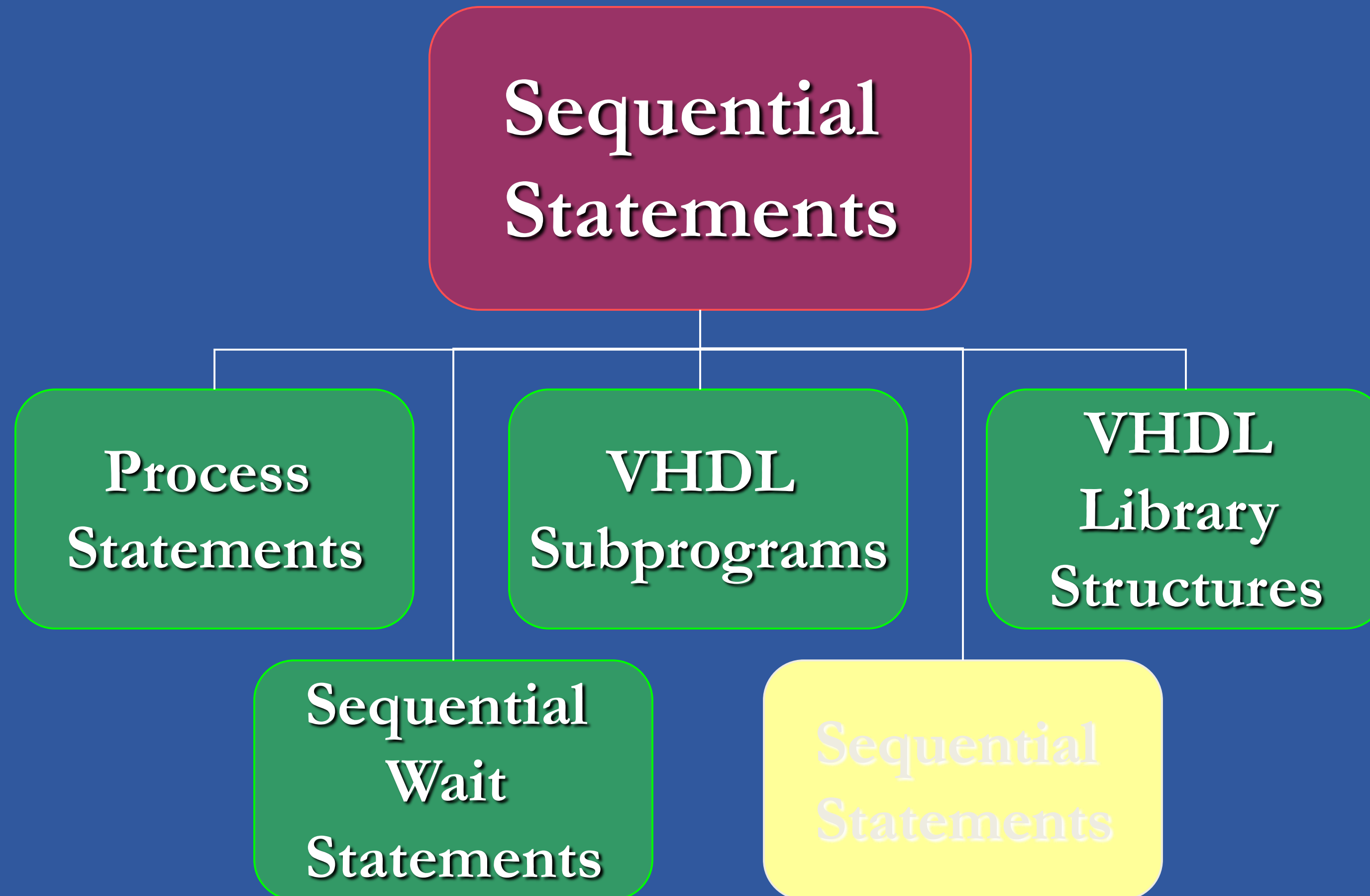
```
PACKAGE GenericParts IS
   COMPONENT dec_n PORT
      (bin_in : IN BIT_VECTOR; en : IN BIT;
       dcd_ou : OUT BIT_VECTOR);
   END COMPONENT;
   COMPONENT alu_n PORT
      (ai, bi : IN BIT_VECTOR;
       mode : IN BIT_VECTOR (1 DOWNTO 0);
       aluout : OUT BIT_VECTOR);
   END COMPONENT;
   COMPONENT mux_n PORT
      (ins : IN BIT_VECTOR;
       s : IN BIT_VECTOR; w : OUT BIT);
   END COMPONENT;
   COMPONENT ssd_f PORT
      (bcd : IN BIT_VECTOR (3 DOWNTO 0);
       display : OUT BIT_VECTOR (1 TO 7));
   END COMPONENT;
   COMPONENT dreg_n PORT
      (rst, clk : IN BIT; regin : IN BIT_VECTOR;
       regout : OUT BIT_VECTOR);
   END COMPONENT;
END PACKAGE GenericParts;
```

A Package of Components

```
LIBRARY utilities;                                  -- Line 1
USE utilities.BasicUtilities.ALL;                   -- Line 2
LIBRARY components;                                 -- Line 3
USE components.GenericParts.ALL;                    -- Line 4
USE components.ALL;                                 -- Line 5
ENTITY alu_n_tester IS END ENTITY;
--
ARCHITECTURE timed OF alu_n_tester IS
   SIGNAL m : BIT_VECTOR (1 DOWNTO 0) := "00";
   SIGNAL li,ri,ao : BIT_VECTOR (7 DOWNTO 0) := "00000100";
   FOR UUT1 : alu_n USE ENTITY
      components.alu4function (customizable);
BEGIN
   UUT1: alu_n PORT MAP (li, ri, m, ao);
   consecutive_data (m, 123 NS, 13);
   consecutive_data (li, 223 NS, 9);
   consecutive_data (ri, 257 NS, 9);
END ARCHITECTURE timed;
```

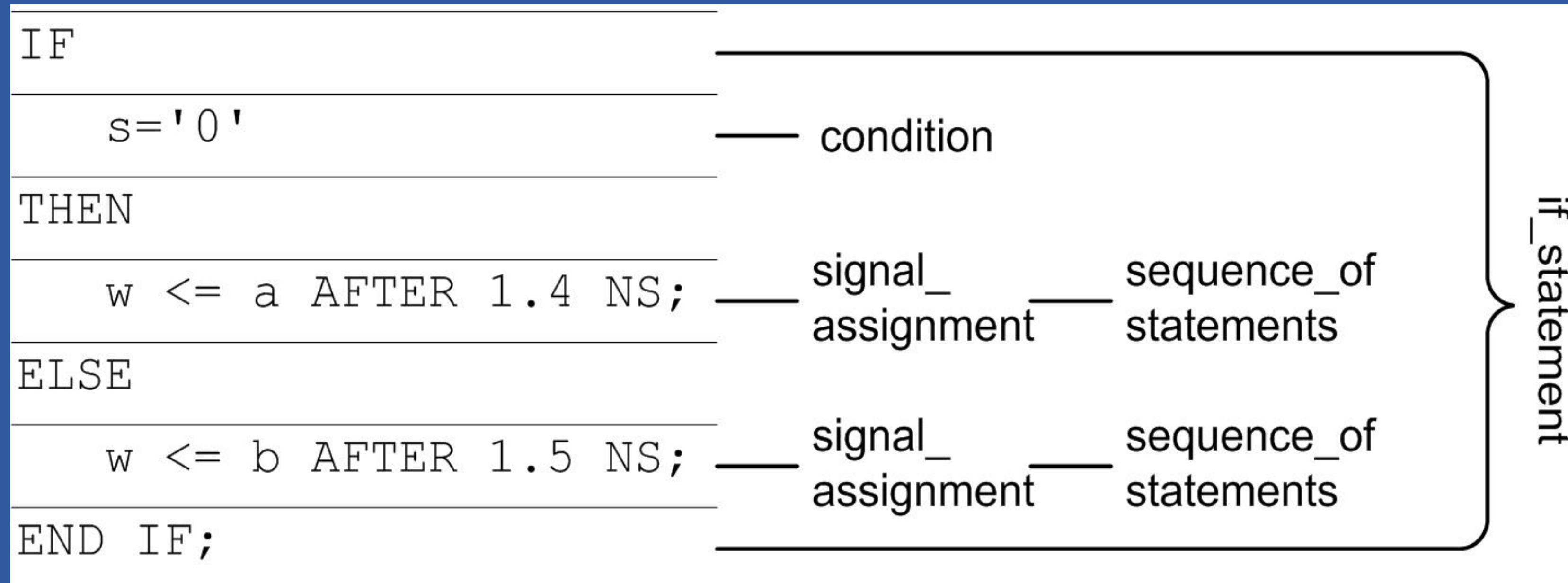
- Using Components and their Declarations

Process Statements



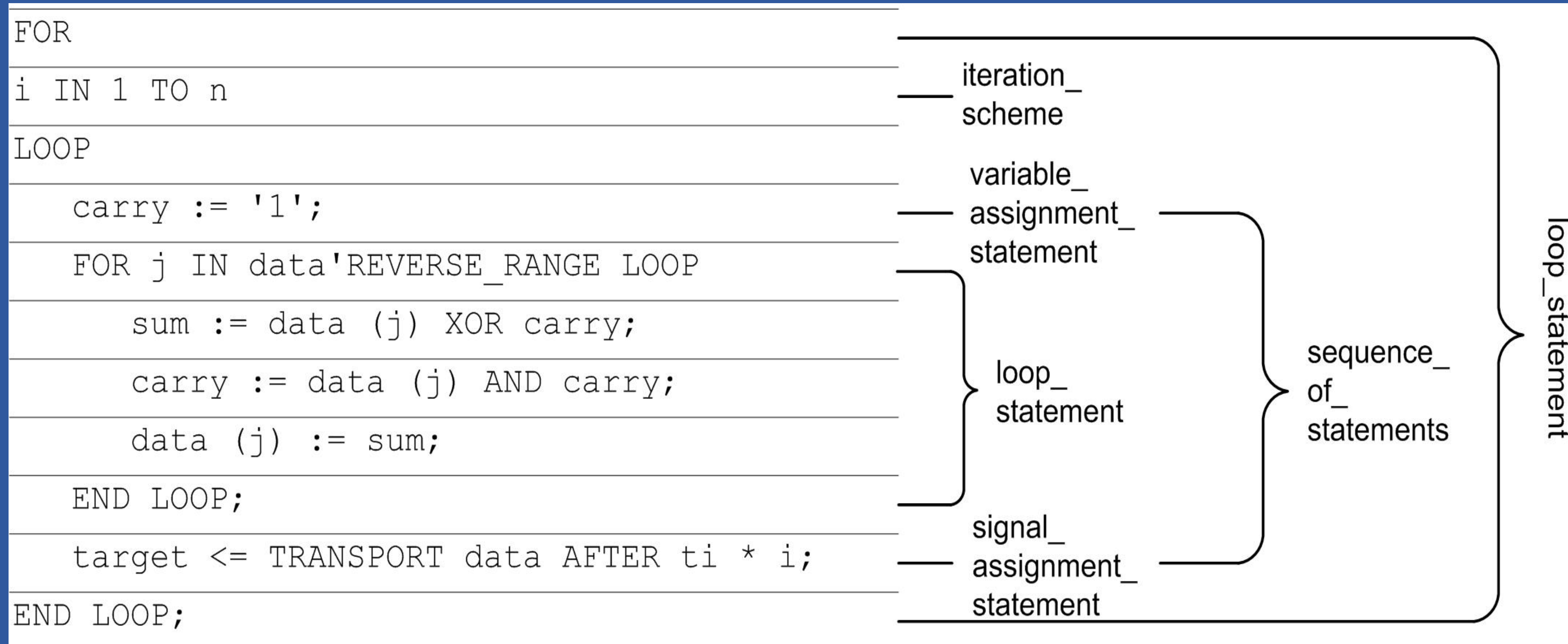
Copyright Z. Navabi, 2007

If Statement



- **Simple if Statement Syntax**

Loop Statement



- **Loop Statement with a FOR Iteration Scheme**

Loop Statement

```
Long_runing : LOOP
    .
    .
    .
    IF x = 25 THEN
       EXIT;
    END IF;
    .
    .
    .
END LOOP long_runing;
```

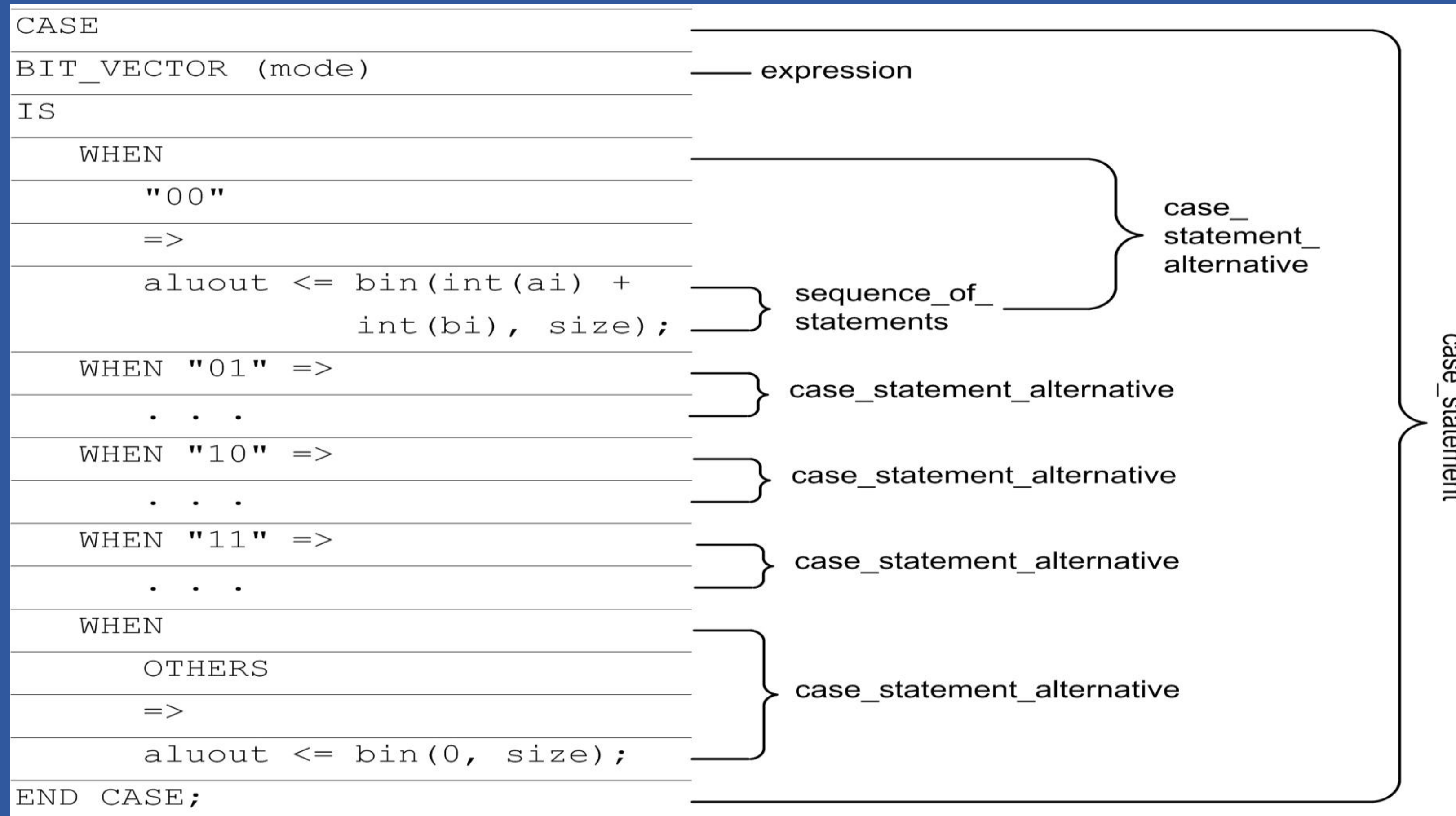
- Partial Code for Demonstrating Exiting from a Potentially Infinite Loop

Loop Statement

```
loop_1 : FOR i IN 5 TO 25 LOOP
   . . .
   sequential_statement_1;
   . . .
   sequential_statement_2;
   . . .
   loop_2 : WHILE j <= 90 LOOP
   . . .
      sequential_statement_3;
      sequential_statement_4;
      . . .
      NEXT loop_1 WHEN condition_1;
      . . .
      sequential_statement_5;
      sequential_statement_6;
      . . .
   END LOOP loop_2;
   . . .
END LOOP loop_1;END LOOP long_runing;
```

- Partial Code for Demonstrating Conditional Next Statements in a Loop

Case Statement



- **Syntax Details of *Case Statement***

Assertion Statement

- **general format :**

```
ASSERT assertion_condition
REPORT "reporting_message"
SEVERITY severity_level;
```

Assertion Statement

```
ARCHITECTURE sync_timed OF dregister IS
   CONSTANT size : INTEGER := regin'LENGTH;
BEGIN
   reg: PROCESS (clk)
      VARIABLE last_edge, duration : TIME := 0 NS;
   BEGIN
      duration := NOW - last_edge;
      last_edge := NOW;
      ASSERT NOT (duration < 3 NS)
         REPORT "Clock Width Too Short"
         SEVERITY NOTE;
      IF (clk = '1') THEN
         IF rst = '1' THEN regout <= bin (0, size);
         ELSE regout <= regin;
         END IF;
      END IF;
   END PROCESS reg;
END ARCHITECTURE sync_timed;
```

- Architecture for Dregister Using Sequential ASSERT

Copyright Z. Navabi, 2007

Summary

- The focus of this chapter was on description of hardware using sequential statements
- A sequential statement offers a convenient way of describing behavior of a hardware component.
- VHDL bodies for inclusion of sequential statements are
  - process statements
  - subprograms.
- Following subjects were discussed in this chapter
  - Details of these constructs and various forms of their utilizations
  - VHDL library structures and packages
- how packages can be used for inclusion of subprograms and component declarations was also showed.

Acknowledgment

Slides developed by:

Nadereh Hatami