SECX1023 PROGRAMMING IN HDL

UNIT II STYLES OF MODELING

Process statement – Wait statement – If statement - Loop statement - Assertion statement – Data flow modeling– Concurrent Signal Assignment statement - Structural modeling – Examples – Component declaration – Component Instantiation – Generate statement – Guarded signals.

Behavioral Modeling

This chapter presents the behavioral style of modeling. In this modeling style, the behavior of the entity is expressed using sequentially executed, procedural type code, A process statement is the primary mechanism used to model the procedural type 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.

Irrespective of the modeling style used, every entity is represented using an entity declaration and at least one architecture body. The first two sections describe these in detail.

An architecture body describes the internal view of an entity. It describes the functionality or the structure of the entity. The syntax of an architecture body is

An architecture body describes the internal view of an entity. It describes the functionality or the structure of the entity. The syntax of an architecture body is

```
architecture architecture-name of
      entity-name is [ architecture-
      item-declarations ]
begin
           statements;
 end architecture-name ;
```

Statements are —>

```
process-statement
block-statement
concurrent-procedure call
concurrent
assertionstatement
concurrent-signal-
assignmentstatement
component-instantiation-
statement generate-statement
```

Process Statement

A process statement contains sequential statements that describe the functionality of a portion of an entity in sequential terms. The syntax of a process statement is

```
[ process-label: ] process [ (
      sensitivity-list ) ] [process-
      item-declarations]
begin
      sequential-statements;
```

```
                    these are -> variable-
                          assignment-statement
                          signal-assignment
                          statement wait-statement
                          if-statement
                          case-
                          statement
                          loop-
                          statement
                          null-
                          statement
                          exit-
                          statement
                          next-statement
                          assertion-statement
                          procedure-call-
                          statement   return-
                          statement.
            end process [process-label];
```

A set of signals that the process is sensitive to 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 on a signal in the sensitivity list. Items declared in the item declarations part are available for use only within the process.

The architecture body, AOI _SEQUENTIAL, presented earlier, contains one process statement. This process statement has four signals in its sensitivity list and has one variable declaration. If an event occurs on any of the signals, A, B, C, or D, the process is executed. This is accomplished by executing statement I first, then statement 2, followed by statement 3, and then statement 4. After this, the process suspends (simulation does not stop, however) and waits for another event to occur on a signal in the sensitivity list

Variable Assignment Statement

Variables can be declared and used inside a process statement. A variable is assigned a value using the variable assignment statement that typically has the form

```
variable-object := expression;
```

The expression is evaluated when the statement is executed and the computed value is assigned to the variable object instantaneously, that is, at the current simulation time.

Variables are created at the time of elaboration and retain their values throughout the entire simulation run (like static variables in C high- level programming language). This is because a process is never exited; it is either in an active state, that is, being executed, or in a suspended state, that is, waiting for a certain event to occur. A process is first entered at the start of simulation (actually, during the initialization phase of simulation) at which time it is executed until it suspends because of a wait statement (wait statements are described later in this chapter) or a sensitivity list.

Consider the following process statement.

```
process (A)
        variable EVENTS_ON_A: INTEGER := 0;
begin
        EVENTS_ON_A                    :=
EVENTS_ON_A+1; end process;
```

At start of simulation, the process is executed once. The variable EVENTS_ON_A gets initialized to 0 and then incremented by 1. After that, any time an event occurs on signal A, the process is activated and the single variable assignment statement is executed. This causes the variable EVENTS_ON_A to be incremented. At the end of simulation, variable EVENTS_ON_A contains the total number of events that occurred on signal A plus one.

Here is another example of a process statement.

```
signal A, Z: INTEGER;
. . .
    PZ: process (A)                       --PZ is a label for the
    process.
            variable V1, V2: INTEGER;
    begin
            V1 := A - V2;                 --statement 1
            Z  <= - V1;                   --statement 2
            V2 := Z+V1 * 2;               -- statement 3
    end process PZ;
```

If an event occurred on signal A at time T1 and variable V2 was assigned a value, say 10, in statement 3, then when the next time an event occurs on signal A, say at time T2, the value of V2 used in statement 1 would still be 10.

Signal Assignment Statement

Signals are assigned values using a signal assignment statement The simplest form of a signal assignment statement is

```
signal-object <= expression [after delay-value ];
```

A signal assignment statement can appear within a process or outside of a process. If it occurs outside of a process, it is considered to be a concurrent signal assignment statement. This' is discussed in the next chapter. When a signal assignment statement appears within a process, it is considered to be a sequential signal assignment statement and is executed in sequence with respect to the other sequential statements that appear within that process.

When a signal assignment statement is executed, the value of the expression is computed and this value is scheduled to be assigned to the signal after the specified delay. It is important to note that the expression is evaluated at the time the statement is executed (which is the current simulation time) and not after the specified delay. If no after clause is specified, the delay is assumed to be a default delta delay.

Some examples of signal assignment statements are

```
COUNTER <= COUNTER+ "0010"; - Assign after a delta delay.
PAR <= PAR xor DIN after 12 ns;
Z <= (AO and A1) or (BO and B1) or (CO and C1) after 6 ns;
```

Wait Statement
As we saw earlier, a process may be suspended by means of a sensitivity list. That is, when a process has a sensitivity list, it always suspends after executing the last sequential statement in the process. The **wait** statement provides an alternate way to suspend the execution of a process. There are three basic forms of the wait statement.

```
wait on sensitivity-list;
wait until boolean-
expression; wait for
time-expression;
```

They may also be combined in a single wait statement. For example

wait on sensitivity-list **until** boolean-expression **for** time-expression-,

Some examples of **wait** statements are

```
wait on A, B, C;              -- statement 1
wait until (A = B);           -- statement 2
wait for 10ns;                -- statement 3
wait on CLOCK for 20ns;       -- statement 4
wait until (SUM > 100) for 50 ms;  -- statement 5
```

In statement 1, the execution of the wait statement causes the process to suspend and then it waits for an event to occur on signals A, B, or C. Once that happens, the process resumes execution from the next statement onwards. In statement 2, the process is suspended until the specified condition becomes true. When an event occurs on signal A or B, the condition is evaluated and if it is true, the process resumes execution from the next statement onwards, otherwise, it suspends again. When the wait statement in statement 3 is executed, say at time T, the process suspends for 10 ns and when simulation time advances to T+10 ns, the process resumes execution from the statement following the wait statement.
The execution of statement 4 causes the process to suspend and then it waits for an event to occur on the signal CLOCK for 20 ns. If no event occurs within 20 ns, the process resumes execution with the statement following the wait. In the last statement, the process suspends for a maximum of 50 ms until the value of signal SUM is greater than 100. The boolean condition is evaluated every time there is an event on signal SUM. If the boolean condition is not satisfied for 50 ms, the process resumes from the statement following the wait statement.
It is possible for a process not to have an explicit sensitivity list. In such a case, the process may have one or more wait statements. It must have at least one wait statement, otherwise, the process will never get suspended and would remain in an infinite loop during the initialization phase of simulation. It is an error if both the sensitivity list and a wait statement are present within a process. The presence of a sensitivity list in a process implies the presence of an implicit "wait on sensitivity-list" statement as the last statement in the process. An equivalent process statement for the process statement in the AOLSEQUENTIAL architecture body is

```
process                              -- No sensitivity list.
        variable TEMP1 ,TEMP2: BIT;
begin
        TEMP1 :=A and B:
        TEMP2 := C and D;
        TEMP1 := TEMP1 or TEMP2;
        Z<= not TEMP1;
        wait on A, B, C, D;           -- Replaces the sensitivity list.
    end process;
```

Therefore, a process with a sensitivity list always suspends at the end of the process and when reactivated due to an event, resumes execution from the first statement in the process.

If Statement

An if statement selects a sequence of statements for execution based on the value of a condition. The condition can be any expression that evaluates to a boolean value. The general form of an if statement is

```
    if boolean-expressionthen
        sequential-statements
    [ elsifboolean-expressionthen         -- elsif clause; if stmt can have 0
or
        sequential-statements ]           -- more elsif clauses.
    [ else                                -- else clause.
        sequential-statements ]
    end if;
```

The if statement is executed by checking each condition sequentially until the first true condition is found; then, the set of sequential statements associated with this condition is executed. The if statement can have zero or more elsif clauses and an optional else clause. An if statement is also a sequential statement, and therefore, the previous syntax allows for arbitrary nesting of if statements. Here are some examples.

```
                                    -- This is a less-than-or-equal-to
    if SUM<=100 then                operator.
        SUM := SUM+10;
        end if;
    if NICKEL_IN then
        DEPOSITED                   --This"<=" is a signal
        <=TOTAL_10;                 assignment
    elsif DIME_IN then              -- operator.
        DEPOSITED <=
    TOTAL_15;              elsif
    QUARTERJN then
        DEPOSITED <= TOTAL_30;
    else
        DEPOSITED <= TOTAL_ERROR;
    end if;

    if CTRLI='1' then
        if CTRL2 = '0' then
            MUX_OUT<= "0010";
        else
            MUX_OUT<= "0001";
        end if;
    else
        if CTRL2 ='0' then
            MUX_OUT <= "1000";
        else
            MUX_OUT <= "0100";
        end if;
```

```
end if;
```

A complete example of a 2-input nor gate entity using an if statement is shown next.

```
entity NOR2 is
port (A, B: in BIT; Z:
out BIT); end NOR2;

architecture NOR2 of NOR2 is -- Architecture body
                                can have-- same name
                                as entity.
begin
    PI: process (A, B)
    constant RISE_TIME: TIME
    :=     10     ns;     constant
    FALL_TIME: TIME := 5 ns:
    variable TEMP: BIT;
    begin
        TEMP := A nor
        B; if (TEMP =
        '1') then
            Z <= TEMP after RISE_TIME;
        else
            Z <= TEMP after FALLJIME;
        end
    if;      end
    process PI;
end NOR2;
```

Case Statement

The format of a case statement is

```
case expression is
    when choices=>sequential-statements       -- branch #1
    when choices=>sequential-statements       -- branch #2
    -- Can have any number of branches.
    [ when others =>sequential-statements ]   -- last branch
end case;
```

The case statement selects one of the branches for execution based on the value of the expression. The expression value must be of a discrete type or of a one-dimensional array type. Choices may be expressed as single values, as a range of values, by using I (vertical bar: represents an "or"), or by using the others clause. All possible values of the expression must be covered in the case statement. "The others clause can be used as a choice to cover the "catch-all" values and, if present, must be the last branch in the case statement. An example of a case statement is

```
type WEEK_DAY is (MON, TUE, WED, THU,
FRI, SAT, SUN); type DOLLARS is range 0 to 10;
variable DAY: WEEK_DAY;
variable POCKET_MONEY: DOLLARS;
case DAY is
    when TUE => POCKET_MONEY := 6;       -- branch 1
    when MON I WED =>POCKET_MONEY := 2;      -- branch 2
    when FRI to SUN=>POCKET_MONEY := 7;       -- branch 3
```

```
        when others =>POCKET_MONEY := 0;     -- branch 4
    end case;
```

Branch 2 is chosen if DAY has the value of either MON or WED. Branch 3 covers the values FRI, SAT, and SUN, while branch 4 covers the remaining value, THU. The case statement is also a sequential statement and it is, therefore, possible to have nested case statements. A model for a 4*1 multiplexer using a case statement is shown next.

```
entity MUX is
    port (A, B, C, D: in BIT; CTRL: in BIT_VECTOR(0 to 1);
        Z: out BIT);
end MUX;

architecture    MUX_BEHAVIOR    of
    MUX          is          constant
    MUX_DELAY: TIME := 10 ns;
begin
    PMUX: process (A, B, C, D,
        CTRL)       variable
        TEMP: BIT;
    begin
        case CTRL is
            when "00" => TEMP
            := A: when "01" =>
            TEMP  :=  B;  when
            "10" => TEMP := C;
            when "11" => TEMP
            := D;
        end case;
        Z    <=    TEMP    after
    MUX_DELAY;   end   process
    PMUX;
end MUX_BEHAVIOR;
```

Null Statement
The statement

```
null;
```

is a sequential statement that does not cause any action to take place and 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.

Loop Statement

A loopstatement is used to iterate through a set of sequential statements.Thesyntax of a loop statement is

```
[   loop-label   :   ]   iteration-
    schemeloopsequential
    -statements
end loop [loop-label] ;
```

There are three types of iteration schemes. The first is the for iteration scheme that has the form

```
for identifier in range
```

An example of this iteration scheme is

```
FACTORIAL := 1;
for NUMBER in 2 to N loop
        FACTORIAL := FACTORIAL *
NUMBER; end loop;
```

The body of the for loop is executed (N-1) times, with the loop identifier, NUMBER, being incremented by I at the end of each iteration. The object NUMBER is implicitly declared within the for loop to belong to the integer type whose values are in the range 2 to N. No explicit declaration for the loop identifier is, therefore, necessary. The loop identifier, also, cannot be assigned any value inside the for loop. If another variable with the same name exists outside the for loop, these two variables are treated separately and the variable used inside the for loop refers to the loop identifier.
The range in a for loop can also be a range of an enumeration type such as

```
type HEXA is ('0', '1', '2', '3', '4', ' 5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E',
'F'):
. . .
for NUM in HEXA'('9') downto HEXA'('0') loop
        -- NUM will take values in type HEXA from '9' through '0'.
        . . .
end loop;

for CHAR in HEXA loop
        -- CHAR will take all values in type HEXA from '0' through 'F'.
        . . .
end loop;
```

Notice that it is necessary to qualify the values being used for NUM [e.g., HEXA'('9')] since the literals '0' through '9' are overloaded, once being defined in type HEXA and the second time being defined in the predefined type CHARACTER. Qualified expressions are described in Chap. 10.
The second form of the iteration scheme is the while scheme that has the form

```
while boolean-expression
```

An example of the while iteration scheme is

```
J:=0;SUM:=10;
WH-LOOP: while J < 20 loop - This loop has a label,
        WH_LOOP. SUM := SUM * 2;
        J:=J+
3; end loop;
```

The statements within the body of the loop are executed sequentially and repeatedly as long as the loop condition, J < 20, is true. At this point, execution continues with the statement following the loop statement.

The third and final form of the iteration scheme is one where no iteration scheme is specified. In this form of **loop** statement, all statements in the loop body are repeatedly executed until some other action causes it to exit the loop. These actions can be caused by an **exit** statement, **a next** statement, or a return statement. Here is an example.

```
SUM:=1;J:=0;
        L2: loop        -- This loop also has a label.
        J:=J+21;
        SUM := SUM* 10;
        exit when SUM > 100;
end loop L2;            -- This loop label, if present, must be the same
                        -- as the initial loop label.
```

In this example, the exit statement causes the execution to jump out of loop L2 when SUM becomes greater than 100. If the exit statement were not present, the loop would execute indefinitely.

Exit Statement

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 is used, the specified loop is exited only if the given condition is true, otherwise, execution continues with the next statement. An alternate form for loop L2 described in the previous section is

```
SUM := 1; J
:= 0; L3: loop
        J:=J+21;
        SUM := SUM*
        10; if (SUM >
        100) then
                exit L3;    -- "exit;" also would have been sufficient.
        end
if; end loop
L3;
```

Next Statement

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 and execution resumes with the first statement in the next iteration of this loop. If no loop label is specified, the innermost loop is assumed. In contrast to the exit statement that causes the loop to be terminated (i.e., exits the specified loop), the next statement causes the current loop iteration of the specified loop to be prematurely terminated and execution resumes with the next iteration. Here is an example.

```
for J in 10 downto 5 loop
        if      (SUM        <
                TOTAL_SUM)
                then   SUM   :=
                SUM +2;
```

```
elsif       (SUM        =
        TOTAL_SUM)
        then next;
else
        null;
end if;
        K:=K
+1; end loop;
```

When the next statement is executed, execution jumps to the end of the loop (the last statement, K := K+1, is not executed), decrements the value of the loop identifier, J, and resumes loop execution with this new value of J.

The next statement can also cause an inner loop to be exited. Here is such an example.

```
L4: for K in 10 downto 1 loop
        --statements
        section 1 L5: loop
                -- statements section 2
                next L4 when WR_DONE = '1';
                --statements
        section 3end loop L5;
                --statements section 4
end loop L4;
```

When WR_DONE = 1' becomes true, statements sections 3 and 4 are skipped and execution jumps to the beginning of the next iteration of loop L4. Notice that the loop L5 was terminated because of the result of **next** statement.

Assertion Statement

Assertion statements are useful in modeling constraints of an entity. For example, you may want to check if a signal value lies within a specified range, or check the setup and hold times for signals arriving at the inputs of an entity. If the check fails, an error is reported. The syntax of an assertion statement is

```
assert       boolean-
expression[
reportstring-
expression     ]     [
severityexpression ]:
```

If the value of the boolean expression is false, the report message is printed along with the severity level. The expression in the severity clause must generate a value of type SEVERTTY_ LEVEL (a predefined enumerated type in the language with values NOTE, WARNING, ERROR, and FAILURE). The severity level is typically used by a simulator to initiate appropriate actions depending on its value. For example, if the severity level is ERROR, the simulator may abort the simulation process and provide relevant diagnostic information. At the very least, the severity level is displayed.

Here is a model of a D-type rising-edge-triggered flip-flop that uses assertion statements to check for setup and hold times.

```
entity DFF is
        port (D, CK: in BIT: Q, NOTQ:
out BIT); end DFF;
```

```
architecture CHECK_TIMES of DFF
        is constant HOLD_TIME:
        TIME := 5 ns; constant
        SETUP_TIME: TIME := 3 ns;
begin
        process (D, CK)
                variable LastEventOnD, LastEventOnCk: TIME;
        begin
                --Check for hold
                time: if D'
                EVENT then
                        assert (NOW = 0ns) or
                                    ((NOW - LastEventOnCk) >=
                                HOLD_TIME) report "Hold time too
                                short!"
                                severity
                        FAILURE;LastEventO
                        nD := NOW;
                end if;

                -- Check for setup time:
                if (CK = '1') and
                        CK'EVENT then
                        assert (NOW =
                        0ns) or
                                    ((NOW - LastEventOnD) >=
                                SETUP_TIME) report "Setup time too
                                short!"
                                severity
                        FAILURE;LastEventO
                        nCk := NOW;
                end if;
                -- Behavior of FF:
                if (CK = '1' ) and
                        CK'EVENT then
                        Q<=D;
                        NOTQ <= not D;
                end
        if; end
        process;
end CHECK_TIMES;
```

EVENT is a predefined attribute of a signal and is true if an event (a change of value) occurred on that signal at the time the value of the attribute is determined. Attributes are described in greater detail in Chap. 10. NOW is a predefined function that returns the current simulation time. In the previous example, the process is sensitive to signals D and CK. When an event occurs on either of these signals, the first if statement is executed. This checks to see if an event occurred on D. If so, the assertion is checked to make sure that the difference between the current simulation time and the last time an event occurred on signal CK is greater than a constant HOLD_TIME delay. If not, a report message is printed and the severity level is returned to the simulator. Similarly, the next if statement checks for the setup time.

The last if statement describes the latch behavior of the D-type flip-flop. The setup and hold times have been modeled as constants in this example. These could also be modeled as generic parameters of the flip-flop. Generics are discussed in Chap. 7.

Here is another example that uses an assertion statement to check for spikes at the input of an inverter.

```
package PACK1 is
        constant MIN_PULSE: TIME := 5
        ns; constant PROPAGATE_DLY:
        TIME := 10 ns;
end PACK1;

use
WORK.PACK1.a
ll; entity INV is
        port (A: in BIT; NOT_A:
out BIT): end INV;

architecture CHECK_INV
of INV is begin
        process (A)
                variable LastEventOnA: TIME := 0 ns;
        begin
                assert (NOW = 0ns) or
                                ((NOW - LastEventOnA) >=
                        MIN_PULSE) report "Spike detected on
                        input of inverter" severity WARNING;
                LastEventOnA := NOW:
                NOT_A <= not A after
        PROPAGATE_DLY; end process;
end CHECK_INV;
```

Some other examples of assertion statements are

```
assert (DATA <= 255)
        report "Data out of range.';

assert (CLK = '0') or (CLK = '1');  --CLK is of type ('X', '0', 'I ', 'Z').
```

In the last assertion statement example, the default report message "Assertion violation" is printed. The default severity level is ERROR if the severity clause is not specified as in the previous two examples.

Dataflow Modeling

This chapter presents techniques for modeling the dataflow of an entity. A dataflow model specifies the functionality of the entity without explicitly specifying its structure. This functionality shows the flow of information through the entity, which is expressed primarily using concurrent signal assignment statements and block statements. This is in contrast to the behavioral style of modeling described in the previous chapter, in which the functionality of the entity is expressed using procedural type statements that are executed sequentially. This chapter also describes resolution functions and their usage.

Concurrent Signal Assignment Statement

One of the primary mechanisms for modeling the dataflow behavior of an entity is by using the concurrent signal assignment statement. An example of a dataflow model for a 2-input or gate, shown in Fig.2.1, follows.

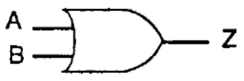

Fig 2.1 An or gate

```
entity OR2 is
        port (signal A, B: in BIT; signal Z:
out BIT); end OR2;

architecture  OR2  of
OR2 is begin
        Z <= A or B after
9 ns; end OR2;
```

The architecture body contains a single concurrent signal assignment statement that represents the dataflow of the or gate. The semantic interpretation of this statement is that whenever there is an event (a change of value) on either signal A or B (A and B are signals in the expression for Z), the expression on the right is evaluated and its value is scheduled to appear on signal Z after a delay of 9 ns. The signals in the expression, A and B, form the "sensitivity list" for the signal assignment statement.
There are two other points to mention about this example. First, the input and output ports have their object class "signal" explicitly specified in the entity declaration. If it were not so, the ports would still have been signals, since this is the default and the only object class that is allowed for ports. The second point to note is that the architecture name and the entity name are the same. This is not a problem since architecture bodies are considered to be secondary units while entity declarations are primary units and the language allows secondary units to have the same names as the primary units.
An architecture body can contain any number of concurrent signal assignment statements. Since they are concurrent statements, the ordering of the statements is not important. Concurrent signal assignment statements are executed whenever events occur on signals that are used in their expressions. An example of a dataflow model for a 1-bit full-adder, whose external view is shown in Fig. 5.2, is presented next.

```
entity FULL_ADDER is
        port  (A,  B,  CIN:  in  BIT;  SUM,
COUT: out BIT); end FULL_ADDER;

architecture          FULL_ADDER          of
        FULL_ADDER  is  begin  SUM<=(A
        xor B) xor CIN after 15 ns;
        COUT <= (A and B) or (B and CIN) or (CIN and
A) after 10 ns; end FULL_ADDER;
```

Concurrent versus Sequential Signal Assignment

In the previous chapter, we saw that signal assignment statements can also appear within the body of a process statement. Such statements are called sequential signal assignment statements, while signal assignment statements that appear outside of a process are called concurrent signal assignment statements. Concurrent signal assignment statements are event triggered, that is, they are executed whenever there is an event on a signal that appears in its expression, while sequential signal assignment statements are not event triggered and are executed in sequence in relation to the other sequential statements that appear within the process. To further understand the difference between these two kinds of signal assignment statements, consider the following two architecture bodies.

```
architecture      SEQ_SIG_ASG      of
        FRAGMENT1 is - A, B and Z
        are signals.
begin
        process (B)
        begin -- Following are sequential signal assignment
                statements:A<=B;
                Z<=
        A;        end
        process;
end;

architecture CON_SIG_ASG of FRAGMENT2 is
begin -- Following are concurrent signal assignment
        statements:A<=B;
        Z<=A;
end;
```

In architecture SEQ_SIG_ASG, the two signal assignments are sequential signal assignments. Therefore, whenever signal B has an event, say at time T, the first signal assignment statement is executed and then the second signal assignment statement is executed, both in zero time. However, signal A is scheduled to get its new value of B only at time T+Δ (the delta delay is implicit), and Z is scheduled to be assigned the old value of A (not the value of B) at time T+Δ also.
In architecture CON_SIG_ASG, the two statements are concurrent signal assignment statements. When an event occurs on signal B, say at time T, signal A gets the value of B after delta delay, that is, at time T+Δ. When simulation time advances to T+Δ, signal A will get its new value and this event on A (assuming there is a change of value on signal A) will trigger the second signal assignment statement that will cause the new value of A to be assigned to Z after another delta delay, that is, at time T+2Δ. The delta delay model is explored in more detail in the next section.
Aside from the previous difference, the concurrent signal assignment statement is identical to the sequential signal assignment statement.
For every concurrent signal assignment statement, there is an equivalent process statement with the same semantic meaning. The concurrent signal assignment statement:

```
CLEAR <= RESET or PRESET
after 15 ns; -- RESET and PRESET
are signals.
```

is equivalent to the following process statement:.

```
proces
begin
        CLEAR <= RESET or PRESET
        after 15 ns; wait on RESET,
          PRESET;
end process;
```

An identical signal assignment statement (this is now a sequential signal assignment) appears in the body of the process statement along with a wait statement whose sensitivity list comprises of signals used in the expression of the concurrent signal assignment statement.

Conditional Signal Assignment Statement

The conditional signal assignment statement selects different values for the target signal based on the specified, possibly different, conditions (it is like an if statement). A typical syntax for this statement is

```
Target - signal <=[ waveform-elements when
          condition else][ waveform-
          elementswhenconditionelse ]
          . . .
          waveform-elements;
```

The semantics of this concurrent statement are as follows. Whenever an event occurs on a signal used either in any of the waveform expressions (recall that a waveform expression is the value expression in a waveform element) or in any of the conditions, the conditional signal assignment statement is executed by evaluating the conditions one at a time. For the first true condition found, the corresponding value (or values) of the waveform is scheduled to be assigned to the target signal. For example,

```
Z  <=   IN0 after 10ns when S0 = '0' and S1 = '0' else
        IN1 after 10ns when S0 = '1' and S1 = '0' else
        IN2 after 10ns when S0 = '0' and S1 = '1' else
        IN3 after 10 ns;
```

In this example, the statement is executed any time an event occurs on signals IN0, IN1, IN2, IN3, S0, or S1. The first condition (S0='0' and S1='0') is checked; if false, the second condition (S0='1' and S1='0') is checked; if false, the third condition is checked; and so on. Assuming S0='0' and S1='1', then the value of IN2 is scheduled to be assigned to signal Z after 10 ns.

For a given conditional signal assignment statement, there is an equivalent process statement that has the same semantic meaning. Such a process statement has exactly one if statement and one wait statement within it. The signals in the sensitivity list for the wait statement is the union of signals in all the waveform expressions and the signals referenced in all the conditions. The equivalent process statement for these conditional signal assignment statement example is

```
proces
begin
if S0 = '0' and S1 = '0'
        then   Z<=   IN0
        after 10 ns;
elsif S0='1'and S1='0' then
Z<= IN1 after 10ns;
        elsif S0='0' and S1 = '1' then Z<= IN2 after 10 ns;
else
        Z<= INS after 10 ns;
end if;
```

```
wait on IN0, IN1, IN2, IN3,
S0, S1; end process;
```

Selected Signal Assignment Statement

The selected signal assignment statement selects different values for a target signal based on the value of a select expression (it is like a case statement). A typical syntax for this statement is

```
with expression select —This is the select
        expression.target-signal <= waveform-
        elements when choices,
                waveform-elements when choices,
                …
                waveform-elements when choices ;
```

The semantics of a selected signal assignment statement are very similar to that of the conditional signal assignment statement. Whenever an event occurs on a signal in the select expression or on any signal used in any of the waveform expressions, the statement is executed. Based on the value of the select expression that matches the choice value specified, the value (or values) of the corresponding waveform is scheduled to be assigned to the target signal. Note that the choices are not evaluated in sequence. All possible values of the select expression must be covered by the choices that are specified not more than once. Values not covered explicitly may be covered by an "others" choice, which covers all such values. The choices may be a logical "or" of several values or may be specified as a range of values.

Here is an example of a selected signal assignment statement.

```
type OP is (ADD, SUB,
MUL, DIV); signal
OP_CODE: OP;
. . .
with OP_CODE select
        Z <= A+B after ADD_PROP_DLY
        when ADD, A - B after
        SUB_PROP_DLY when SUB,
        A * B after MUL_PROP_DLY
        when MUL, A / B after
        DIV_PROP_DLY when DIV;
```

In this example, whenever an event occurs on signals, OP_CODE, A, or B, the statement is executed. Assuming the value of the select expression, OP_CODE, is SUB, the expression "A - B" is computed and its value is scheduled to be assigned to signal Z after SUB_PROP_DLY time.

For every selected signal assignment statement, there is also an equivalent process statement with the same semantics. In the equivalent process statement, there is one case statement that uses the select expression to branch. The list of signals in the sensitivity list of the wait statement comprises of all signals in the select expression and in the waveform expressions. The equivalent process statement for the previous example is

```
proces
begin
        case OP_CODE is
```

```
                when ADD => Z<= A +B after
                ADD_PROP_DLY; when SUB =>Z <=
                A-B after SUB_PROP_DLY; when
                MUL =>Z<= A * B after
                MUL_PROP_DLY; when DIV => Z <=
                A /B after DIV_PROP_DLY;
            end case;
            wait on OP_CODE,
        A, B; end process;
```

Structural Modeling

This chapter describes the structural style of modeling. An entity is modeled as a set of components connected by signals, that is, as a netlist. The behavior of the entity is not explicitly apparent from its model. The component instantiation statement is the primary mechanism used for describing such a model of an entity.

An Example

Consider the circuit shown in Fig. 2.2 and its VHDL structural model.

```
        entity GATING is
            port (A, CK, MR, DIN: in BIT; RDY,
        CTRLA: out BIT); end GATING;

        architecture STRUCTURE_VIEW of
            GATING is component AND2
                port (X, Y: in BIT; Z:
            out BIT); end component;
\           component DFF
                port (D, CLOCK: in BIT; Q,
            QBAR: out BIT); end component;
            component NOR2
                port (A, B: in BIT; Z:
            out BIT); end component;
            signal SI, S2: BIT;
        begin
            D1: DFF port map (A, CK, SI, S2);
            A1: AND2 port map (S2, DIN,
            CTRLA); N1: NOR2 port map
            (SI, MR, RDY);
end STRUCTURE_VIEW
```

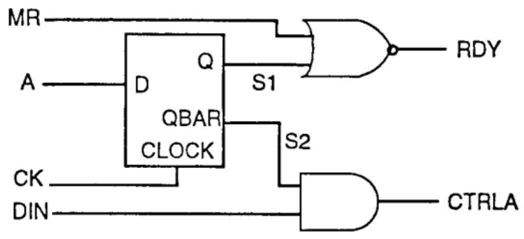

Fig 2.2: A circuit generating control signals

Three components, AND2, DFF, and NOR2, are declared. These components are instantiated in the architecture body via three component instantiation statements, and the instantiated components are connected to each other via signals SI and S2. The component instantiation statements are concurrent statements, and therefore, their order of appearance in the architecture body is not important. A component can, in general, be instantiated any number of times. However, each instantiation must have a unique component label; as an example, A1 is the component label for the AND2 component instantiation.

Component Declaration

A component instantiated in a structural description must first be declared using a component declaration. A component declaration declares the name and the interface of a component. The interface specifies the mode and the type of ports. The syntax of a simple form of component declaration is

```
component component-name
        port    (list-of-interface-
ports ) ; end component;
```

The component-name may or may not refer to the name of an already ex-isfing entity in a library. If it does not, it must be explicitly bound to an entity; otherwise, the model cannot be simulated. This is done using a configuration. Configurations are discussed in the next chapter.
The list-of-interface-ports specifies the name, mode, and type for each port of the component in a manner similar to that specified in an entity declaration. "The names of the ports may also be different from the names of the ports in the entity to which it may be bound (different port names can be mapped in a configuration). In this chapter, we will assume that an entity of the same name as that of the component already exists and that the name, mode, and type of each port matches the corresponding ones in the component. Some examples of component declarations are

```
component NAND2
        port (A, B: in MVL; Z:
out MVL); end component;

component MP
        port (CK, RESET, RON, WRN: in BIT;
                DATA_BUS: inout INTEGER range
                0  to  255;  ADDR_BUS:  in
                BIT_VECTOR(15 downto 0));
end component;

component RX
        port (CK, RESET, ENABLE, DATAIN, RD:
                in BIT;DATA_OUT: out INTEGER
                range  0  to  (2**8  -  1);
                PARITY_ERROR,  FRAME_ERROR,
                OVERRUN_ERROR: out BOOLEAN);
end component;
```

Component Instantiation

A component instantiation statement defines a subcomponent of the entity in which it appears. It associates the signals in the entity with the ports of that subcomponent. A format of a component instantiation statement is

component-label: component-name port **map**(association-list) ',

The component-label can be any legal identifier and can be considered as the name of the instance. The component-name must be the name of a component declared earlier using a component declaration. The association-list associates signals in the entity, called actuals, with the ports of a component, called locals. An actual must be an object of class signal. Expressions or objects of class variable or constant are not allowed. An
actual may also be the keyword open to indicate a port that is not connected. There are two ways to perform the association of locals with actuals:

1. positional association,
2.named association.

In positional association, an association-list is of the form

actuali, actualg, actual3, . . ., actual

Each actual in the component instantiation is mapped by position with each port in the component declaration. That is, the first port in the component declaration corresponds to the first actual in the component instantiation, the second with the second, and so on. Consider an instance of a NAND2 component.

--Component
declaration:
component NAND2
port (A, B: **in** BIT; **Z:**
out BIT); **end component;**

--Component instantiation:
N1: NAND2 **port map** (S1, S2, S3);

N1 is the component label for the current instantiation of the NAND2 component. Signal S1 (which is an actual) is associated with port A (which is a local) of the NAND2 component, S2 is associated with port B of the NAND2 component, and S3 is associated with port Z. Signals S1 and S2 thus provide the two input values to the NAND2 component and signal S3 receives the output value from the component. The ordering of the actuals is, therefore, important.

If a port in a component instantiation is not connected to any signal, the keyword open can be used to signify that the port is not connected. For example,

N3: NAND2 **port map** (S1, **open,** S3);

The second input port of the NAND2 component is not connected to any signal. An input port may be left open only if its declaration specifies an initial value. For the previous component instantiation statement to be legal, a component declaration for NAND2 may appear like

component NAND2
port (A, B: **in** BIT := '0'; **Z: out** BIT);
1 Both A and B have an initial value of '0'; however, only
2 the initial value of B is necessary in this case.
end component;

A port of any other mode may be left unconnected as long as it is not an unconstrained array. In named association, an association-list is of the form

locale => actual1, local2 => actual2, ..., localn => actualn

For example, consider the component NOR2 in the entity GATING described in the first section. The instantiation using named association may be written as

N1: NOR2 **port map** (B=>MR, Z=>RDY, A=>S1);

In this case, the signal MR (an actual), that is declared in the entity port list, is associated with the second port (port B, a local) of the NOR2 gate, signal RDY is associated with the third port (port Z) and signal S1 is associated with the first port (port A) of the NOR2 gate. In named association, the ordering of the associations is not important since the mapping between the actuals and locals are explicitly specified. An important point to note is that the scope of the locals is restricted to be within the port map part of the instantiation for that component; for example, the locals A, B, and Z of component NOR2 are relevant only within the port map of instantiation of component NOR2.
For either type of association, there are certain rules imposed by the language. First, the types of the local and the actual being associated must be the same. Second, the modes of the ports must conform to the rule that if the local is readable, so must the actual and if the local is writable, so must the actual. Since a signal locally declared is considered to be both readable and writable, such a signal may be associated with a local of any mode. If an actual is a port of mode in, it may not be associated with a local of mode out or inout; if the actual is a port of mode out, it may not be associated with a local of mode in or inout; if the actual is a port of mode inout, it may be associated with a local of mode in, out, or inout.

Generate Statements

Concurrent statements can be conditionally selected or replicated during the elaboration phase using the generate statement. There are two forms of the generate statement.

1. Using the for-generaHon scheme, concurrent statements can be replicated a predetermined number of times.

2. With the if-generation scheme, concurrent statements can be conditionally selected for execution.

The generate statement is interpreted during elaboration, and therefore, has no simulation semantics associated with it. It resembles a macro expansion. The generate statement provides for a compact description of regular structures such as memories, registers, and counters.

The format of a generate statement using the for-generation scheme is

generate-label: **for** generale-identifierin discrete-range **generate** concurrent-statements **end** **generate**[generate-label];

The values in the discrete range must be globally static, that is, they must be computable at elaboration time. During elaboration, the set of concurrent statements are replicated once for each value in the discrete range. These statements can also use the generate identifier in their expressions and its value would be substituted during elaboration for each replication. There is an implicit declaration for the generate identifier within the generate statement, and therefore, no declaration for this identifier is required. The type of the identifier is defined by the discrete range.
Consider the following representation of a 4-bit full-adder, shown in Fig. 2.3, using

the generate statement.

```
entity FULL_ADD4 is
        port (A, B: in BIT_VECTOR(3 downto 0); CIN: in BIT;
                SUM: out BIT_VECTOR(3 downto 0); COUT: out
                BIT);
end FULL_ADD4:

architecture    FOR_GENERATE    of
        FULL_ADD4    is    component
        FULL_ADDER
                port (A, B, C: in BIT; COUT,
        SUM: out BIT); end component;
        signal CAR: BIT_VECTOR(4 downto 0);
begin
        CAR(0) <= CIN;
        GK: for K in 3 downto 0 generate
        FA: FULL_ADDER port map (CAR(K),
                                A(K),       B(K),
                                CAR(K+1),SUM(
                                K));
        end  generate
        GK;COUT  <=
        CAR(4);
end FOR_GENERATE
```

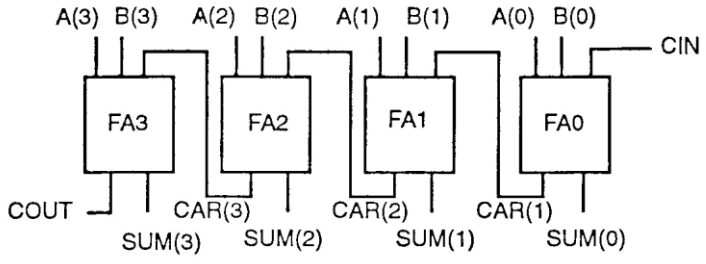

Fig.2.3: A 4-bit full-adder.

After elaboration, the generate statement is expanded to

```
FA(3): FULL_ADDER port map (CAR(3), A(3), B(3), CAR(4),
SUM(3));
FA(2): FULL_ADDER port map (CAR(2), A(2), B(2), CAR(3),
SUM(2));
FA(1): FULL_ADDER port map (CAR(1), A(1), B(1), CAR(2),
SUM(1));
FA(0): FULL_ADDER port map (CAR(0), A(0), B(0), CAR(1),
SUM(0));
```

Components in a generate statement can be bound to entities using a generate block configuration. A block configuration is defined for each range of generate labels. Here is an example of such a binding using a configuration declaration.

```
configuration GENERATE_BIND of FULL_ADD4 is
        use WORK.all;   -- Example of a declaration in the
                                -- configuration declarative part.
        for FOR_GENERATE        -- An architecture body block
        configuration.
            forGK(1)      --A generate block configuration.
                for FA: FULL_ADDER
                                use                 configuration
                                WORK.FA_HA_CON;
                end for;
            end for;
            for GK(2 to 3)
                for FA: FULL_ADDER - No explicit binding.
                                -- Use defaults, i.e., use entity
                                FULL_ADDER -- in working
                                library.
                end for;
            end
            for; for
            GK(0)
                for FA: FULL_ADDER
                                use entity
                WORK.FULL_ADDER(FA_DA
                TAFLOW); end for;
            end for;
        end for;
    end GENERATE_BIND;
```

There are three generate block configurations, one each for GK(1), GK(2 to 3), and for GK(0). Each of these block configurations define the bindings for the components valid for that generate index.

The body of the generate statement can also have other concurrent statements. For example, in the previous architecture body, the component instantiation statement could be replaced by signal assignment statements like this

```
    G2: for M in 3 downto 0 generate
        SUM(M) <= (A(M) xor B(M)) xor
        CAR(M); CAR(M+1 ) <= (A(M) and
        B(M)) and CAR(M);
    end generate G2;
```

The second form of the generate statement uses the if-generation scheme. The format for this type of generate statement is

```
    genarate-label:  H  expression
        generate     concurrent-
        statements
    end generate [generete-label] ;
```

The if-generate statement allows for conditional selection of concurrent statements based on the value of an expression. This expression must be a globally static expression, that is, the value must be computable at elaboration time.

Here is an example of a 4-bit counter, that is modeled using the if-generate statement.

```
    entity COUNTER4 is
```

```
          port (COUNT, CLOCK: in BIT; Q: buffer
BIT_VECTOR(0 to 3)); end COUNTER4;

architecture IF_GENERATE of
          COUNTER4 is component
          D_FLIP_FLOP
                    port (D, CLK: in BIT; Q:
          out BIT); end component;
begin
          GK: for K in 0 to 3
                    generate GKO: if K
                    = 0 generate
                              DFF: D_FLIP_FLOP port map (COUNT,
                    CLOCK, Q(K)); end generate GK0;
                    GK1_3: if K > 0 generate
                              DFF: D_FLIP_FLOP port map (Q(K-1),
                    CLOCK, Q(K)); end generate GK1_3;
          end generate
GK;                 end
IF_GENERATE;
```

Guarded Signals

A guarded signal is a special type of a signal that is declared to be of a register or a bus kind in its declaration. A general form of a signal declaration is

```
signal list-of-signals: resolution-function
            signal-typesignal-kind [
            := expression ];
```

A guarded signal must be a resolved signal, that is, it must have a resolution function associated with it. Also, the signal can only be assigned values under the control of a guard expression, for example, using a guarded assignment (guarded option used in a concurrent signal assignment statement). This implies that guarded signals can only be assigned values within block statements.

A guarded signal behaves differently from other signals in that when the guard expression is false, the driver to the guarded signal becomes disconnected after a specific time, called the disconnect time. On the other hand, in an unguarded signal, if the guard expression is false, any new events on the signals appearing in the expression do not influence the value of the target signal; the driver continues to drive the target signal with the old value. To understand this difference better, consider the following guarded block BL

```
architecture GUARDED_EX of EXAMPLE is
          signal GUARD_SIG: WIRED_OR
          BIT         register;         signal
          UNGUARD_SIG: WIRED_AND BIT;
begin
          B1: block ( guard-
          expression )begin
                    GUARD_SIG                 <=
                    guardedexpression1           ;
                    UNGUARD_SIG               <=
                    guardedexpression2;
          end   block
B1;               end
GUARDED_EX;
```

Transforming the guarded signal assignment statement into its equivalent process statement, the block B1 now looks like this

```
B1:   block   (   guard-
expression )begin
          proces
          s
          begin
                    if GUARD then
                              GUARD_SIG <=expression1;
                    else
                              GUARD_SIG<=null;

end if;
                    wait          on          signals-in-
          expressioni1,GUARD; end process;
          proces
          s
          begin
                    If GUARD then
                              UNGUARD_SIG <= expression2;
                    end if;
                    wait          on          signals-in-
          expressionS,GUARD; end process;
end block B1;
```

The process statement for the guarded signal, GUARD_ SIG, has an explicit signal assignment statement that disconnects its driver, while there is no such statement for the unguarded signal, UNGUARD_SIG. As this example shows, a driver of a guarded signal can be explicitly disconnected by assigning a null value to the signal. Such a statement is called a disconnection statement.

Let us now explore the differences between a register and a bus signal. A bus signal represents a hardware bus in that when all drivers to the signal become disconnected (as might be the case on a real hardware bus), the value of the signal is determined by calling the resolution function with all the drivers off. A register signal, on the other hand, models a storage component (that is multiply driven) in which if all drivers to the signal become disconnected, the resolution function is not called and the value of the last active driver is retained. With a bus signal, the previous value is lost. Also, bus signals may either be ports of an entity or locally declared signals, whereas register signals can only be locally declared signals.

The disconnect time for a guarded signal can be specified using a disconnection specification. The syntax of a disconnection specification is

```
disconnect guarded-signal-name: signal-type after time-expression;
```

This is an example of a disconnection specification.

```
disconnect GUARD_SIG: BIT after 8 ns;
```

This implies that the driver of signal GUARD_SIG will get disconnected 8 ns after the corresponding GUARD goes false.

The disconnection specification is useful in modeling decay times, for example, capacitance delay on buses. An alternate way of specifying disconnect time is by assigning a value null to the signal in a disconnection statement as shown.

```
S1 <= null after 10 ns;
```

This statement specifies that the driver of SI will be disconnected after 10 ns. Thereafter, this driver does not contribute to the resolved value of the signal. However, such a statement can appear only as a sequential statement and the target signal must be a guarded signal.

Here is a more comprehensive example.

```
use WORK.RF.PACK.all;
-- Package RF_PACK contains functions WIRED_AND
and WIRED_OR. entity GUARDED_SIGNALS is
        port (CLOCK: in BIT; N: in INTEGER);
end;

architecture EXAMPLE of GUARDED_SIGNALS is
        signal      REG_SIG:      WIRED_AND
        INTEGER  register;  signal  BUS_SIG:
        WIRED_OR  INTEGER  bus;  disconnect
        REG_SIG:   INTEGER   after   50   ns;
        disconnect BUS_SIG: INTEGER after 20
        ns;
begin
        BX:   block   (CLOCK='1'   and   (not
        CLOCK'STABLE)) begin
                REG_SIG   <=   guarded   N
                after  15  ns;  BUS_SIG  <=
                guarded N after 10 ns;
        end   block
BX;             end
EXAMPLE;
```