SECX1023 - PROGRAMMING IN HDL

UNIT- 3

UNIT 3 - ADVANCED FEATURES

3.1 Generics

It is often useful to pass certain types of information into a design description from its environment. Examples of such information are rise and fall delays, and size of interface ports. This is accomplished by using generics. Generics of an entity are declared along with its ports in the entity declaration.

```
entity AND_GATE is
        generic (N: NATURAL);
        port (A: in BIT_VECTOR(1 to N); Z: out BIT);
end AND_GATE;

architecture GENERIC_EX of AND_GATE is
begin
        process (A)
                variable AND_OUT: BIT;
        begin
                AND_OUT := '1';
                for K in 1 to N loop
                        AND_OUT := AND_OUT and A(K);
                end loop;
                Z <= AND_OUT;
        end process;
end GENERIC_EX;
```

In this example, the size of the input port has been modeled as a generic. By doing this, we have modeled an entire class of and gates with a variable number of inputs using a single behavioral description. The AND_GATE entity may now be used with a different number of input ports in different instantiations.

A generic declares a constant object of mode in (that is, the value can only be read), and can be used in the entity declaration and its corresponding architecture bodies. The value of this constant can be specified as a locally static expression in one of the following:

1. entity declaration
2. component declaration
3. component instantiation
4. configuration specification
5. configuration declaration

The value of a generic must be determinable at elaboration time, that is, a value for a generic must be explicitly specified at least once using any of the ones mentioned. The value for a generic may be specified in the entity declaration for an entity as shown in this example. This is the default value for the generic. It can be overridden by others.

```
entity NAND_GATE is
        generic (M: INTEGER := 2);-- M models the number of inputs.
        port (A: in BIT_VECTOR(M downto 1); Z: out BIT);
end NAND_GATE;
```

Two other alternate ways of specifying the value of a generic are in a component declaration and in a component instantiation. The following examples demonstrate these.

```
entity
ANOTHER_GEN_EX is
 end;

architecture GEN_IN_COMP of ANOTHER_GEN_EX is
-- Component declaration for NAND_GATE:
component NAND_GATE
        generic (M: INTEGER);
        port (A: in BIT_VECTOR (M downto 1); Z: out BIT);
end component;
-- Component declaration for AND_GATE:
component AND_GATE
        generic (N: NATURAL := 5);
        port (A: in BIT_VECTOR(1 to N); Z: out BIT);
end component;
signal S1, S2, S3, S4: BIT;
signal SA: BIT_VECTOR (1 to 5);
signal SB: BIT_VECTOR (2 downto 1); signal SC: BIT_VECTOR (1
to 10);
signal SD: BIT_VECTOR (5 downto 0);
begin
      - Component instantiations:
      N1: NAND_GATE generic map (6) port map (SD, S1);
      A1: AND_GATE generic map (N => 10) port map (SC,
      S3); A2; AND_GATE port map (SA, S4);
      -- N2: NAND_GATE port map (SB, S2);
end GEN_IN_COMP;
```

For the purposes of this discussion, we shall assume that the components NAND_GATE and AND_GATE are bound to the entities NAND_GATE and AND_GATE described earlier. The component declaration for AND_GATE specifies a value for the generic. When this component is instantiated and a new generic value is assigned using a generic map as in instance A1, the new value, that is, 10, overrides the value specified in the component declaration, that is, 5. When the AND_GATE component is instantiated and no generic map is specified as in instance A2, the value of the generic specified in the component declaration, that is, 5, is used. In the case of instance N1, again the value supplied by the generic map (i.e., 6) overrides the value assigned to the generic in the entity declaration for NAND_GATE (i.e., 2). The instance N2, shown as a comment, is illegal since neither the instantiation nor the declaration supply the value for the generic.

Values for generics may also be specified in a configuration specification or in a configuration declaration. We shall see this later in the section on configurations. The model of a nor gate with generic rise and fall delays is shown next.

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

architecture NOR2_DELAYS of NOR2 is
        signal TEMP: BIT;
begin
        TEMP <= not (A or B);
        Z <= TEMP after PT_HL when (TEMP = -0') else
```

```
TEMP        after  PT_LH;
            end NOR2_DELAYS;
```

Since no default values were provided for the generics in this case, the values must be provided later when this entity is instantiated or configured.

Consider an or gate constructed using two nor gates; each nor gate has the behavior described previously. The rise and fall delays are specified when the NOR2 component is instantiated. In the following example, different propagation delays are specified in each component instantiation statement.

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

architecture  OR2_NOR2  of
        OR2  is  component
        NOR2
                generic  (PT_HL,  PT_LH:
                TIME);
                port  (A,  B:  in  BIT:  Z:
                out BIT);
        end component;
        signal S1: BIT;
 begin

      N1: NOR2 generic map (5 ns, 3 ns) port map (A, B, S1);
      N2: NOR2 generic map (6 ns, 5 ns) port map (S1, S1, C);
  end;
```

Generics can also be used to control the number of instantiations of a component in a generate statement.

3.2 Configurations

- why are configurations needed? There are two main reasons.

1. Sometimes it may be convenient to specify multiple views for a single entity and use any one of these for simulation. This can be easily done by specifying one architecture body for each view and using a configuration to bind the entity to the desired architecture body. For example, corresponding to an entity FULL_ADDER, there may be three architecture bodies, called FA_BEH, FA_STR, and FA_MIXED. Any one of these can be selected for simulation be specifying an appropriate configuration.
2. Similar to the previous case, it may be desirable to associate a component with any one of a set of entities. The component declaration may have its name and the names, types, and number of ports and generics different from those of its entities. For example, a declaration for a component used in a design may be

```
component OR2
          port (A, B: in BIT; Z:
                out BIT);
end component;
```

and the entities that the above component may possibly be bound to are

```
entity OR_GENERIC is
        port (N: out BIT; L, M: in BIT):
end OR_GENERIC;

entity OR_HS is
        port (X, Y: in BIT; Z: out BIT);
end OR_HS;
```

The component names and the entity names, as well as the port names and their order are different. In one case we may be interested in using the OR_HS entity for the OR2 component, and in another case, the OR_GENERIC entity. This can be achieved by appropriately specifying a configuration for the component. The advantage is that when components are used in a design, arbitrary names for components and their interface ports can be used and these can later be bound to specific entities prior to simulation.

A configuration is, therefore, used to bind

1. An architecture body to its entity declaration,
2. A component with an entity.

Note that a configuration does not have any simulation semantics associated with it; it only specifies how a top-level entity is organized in terms of lower level entities by specifying the bindings between the entities. The language provides two ways of performing this binding:

1. By using a configuration specification,
2. By using a configuration declaration.

3.3 Configuration Specification

A configuration specification is used to bind component instantiations to specific entities that are stored in design libraries. The specification appears in the declarations part of the architecture or block in which the components are instantiated. Binding of a component to an entity can be done on a per instance basis, or for all instantiations of a component, or for a selected set of instantiations of a component. Instantiations of different components can also be bound to the same entity.
Figure 3.1 shows a logic diagram for a 1-bit full-adder. Its structural model is described next.

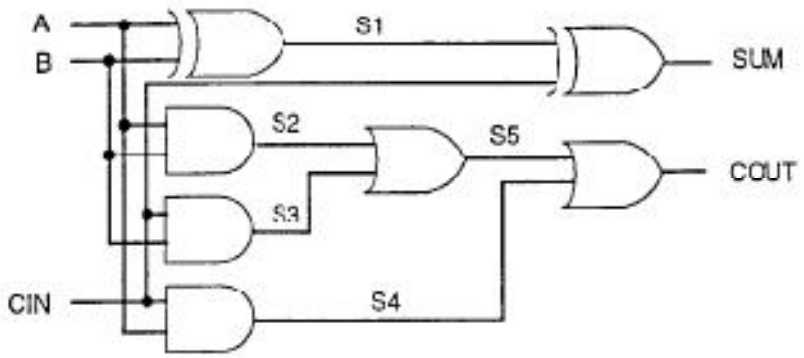


Figure 3.1 A 1 -bit full-adder circuit.

```
library HS_LIB, CMOS_LIB;
entity FULL_ADDER is
```

```
        port (A, B, CIN: in BIT; SUM, COUT: out BIT);
    end;
architecture FA_STR of FULL_ADDER is
        component XOR2
                port (A, B: in BIT; C: out BIT);
        end component;
        component AND2
                port (Z: out BIT; A0, A1: in BIT);
        end component;
        component OR2
                port (A, B: in BIT; C: out BIT);
        end component;
        -- The following four statements are configuration specifications.
        for X1, X2: XOR2
                use entity WORK.XOR2(XOR2BEH); - Binding the entity with
                        more than one instantiation of a component.
        for A3: AND2
                use entity HS_LIB.AND2HS(AND2STR)
                port map (HS_B=>A1, HS_Z=>Z, HS_A=>A0); - Binding the
                        -entity with a single instantiation of a component.

        for all: OR2
                use entity CMOS_LIB.OR2CMOS(OR2STR); - Binding the
                        entity with all instantiations of OR2 component.
        for others: AND2
                use entity WORK.A_GATE(A_GATE_BODY)
                port map (A0, A1, Z);-Binding the entity with all unbound
                        -- instantiations of AND2 component.
        signal S1, S2, S3, S4, S5: BIT;
begin
        X1: XOR2 port map (A, B, S1);
        X2: XOR2 port map (S1, CIN, SUM);
        A1: AND2 port map (S2, A, B);
        A2: AND2 port map (S3, B, CIN);
        A3: AND2 port map (S4, A, CIN);
        O1: OR2 port map (S2, S3, S5);
        O2: OR2 port map (S4, S5, COUT);
end FA_STR;
```

The four for statements appearing in the declarative part of the architecture body are the configuration specifications. The first specification statement indicates that instances XI and X2 of component XOR2 are bound to the entity represented by the entity-architecture pair, XOR2 and XOR2BEH, respectively, that resides in library WORK. The second specification binds the AND2 component with instantiation label A3 to the entity represented by the entity-architecture pair, AND2HS and AND2STR, respectively, that is present in design library HS_LIB. "The mapping of the component (AND2) ports and the entity (AND2HS) ports is specified using named association; for example, port HS_A of the AND2HS entity is mapped to port AO of the AND2 component. The third specification implies that for all instances of component OR2, use the entity represented by the specified entity-architecture pair that is present in the design library, CMOS_LIB. The last specification statement implies that all unbound instances of component AND2, that is, Al and A2, are bound to the entity A_GATE using the architecture A_GATE_BODY, that resides in library WORK.

The previous example showed that different instances of the same component can be bound to different entities. Figure 3.2 depicts this binding. Similarly, it is also possible to bind different components to the same entity. An example is shown in Fig. 3.3. This figure shows that there is nothing special about the component name being AND2. By binding an instance of component AND2 to an entity called OR_GATE, this

instance is being made to behave as specified in the architecture of entity OR_GATE. Using such a binding may cause confusion to the reader, even though it is syntactically correct, and it may be what was intended. Such bindings may sometimes be necessary, for example, while debugging a model, we may want to see the effect of specifying an AND gate to behave like an or gate without changing the rest of the description. This flexibility of being allowed to bind a component instance to any entity may result in a complex maze of bindings. An example is shown in Fig. 3.4.

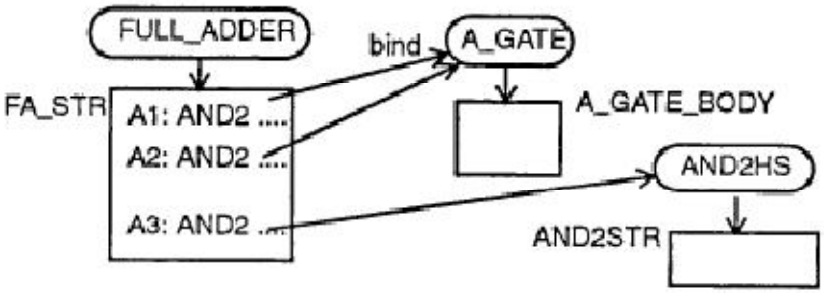


Figure 3.2 Different instances bound to different entities

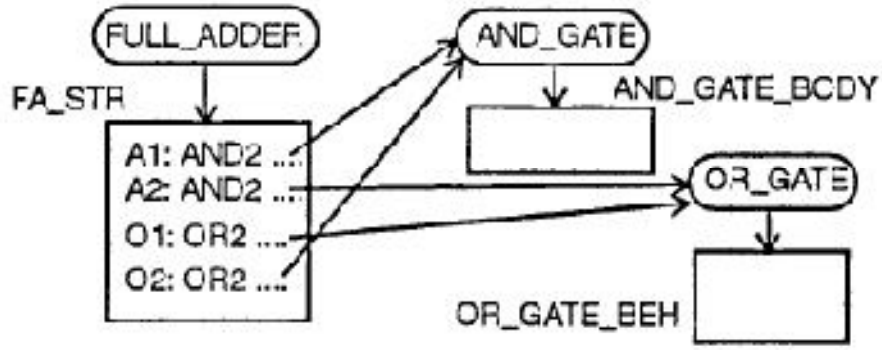


Figure 3.3 Different components bound to same entity

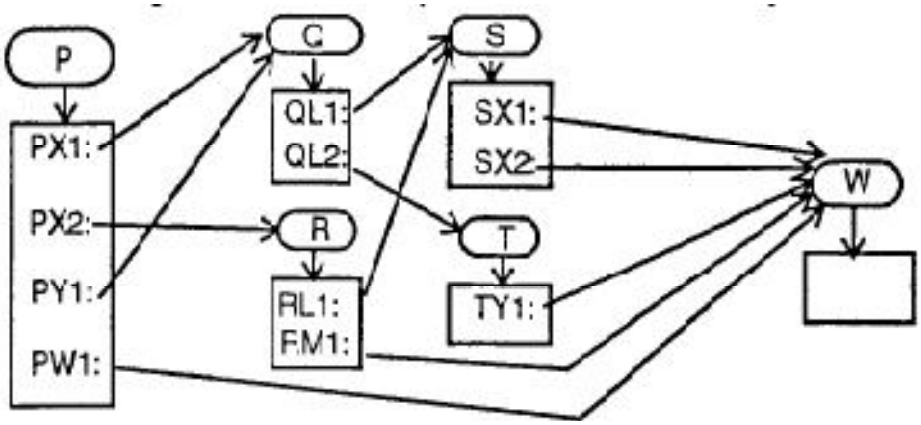


Figure 3.4 A complex maze of bindings

Entity P has four component instances, PXI and PX2 of component type PX, PYI of component type PY, and PWI of component type PW. Component instances PXI and PYI are bound to entity Q while PX2 is bound to R. Component instances QLI and QL2 (of component type QL) in entity Q are bound to entities S and T, respectively. Component instances RLI and RMI in entity R are bound to entities S and W, respectively. All component instances in S and T, and component PWI in entity P are bound to a single entity W. In other words, all the entities, P, Q, R, S, and T have been built hierarchically using a single primitive component, W.

The syntax of a configuration specification is

for *list-ofrcomp-labels: component-name*
use *binding-indication*;

The binding-indication specifies the entity represented by the entity-architecture pair, and the generic and port bindings, and one of its forms is

entity *entity-name* [(*architecture-name*)]
[**generic map** (*generic-association*)]
[**port map** (*port-association*)] -- Form 1

The list of component labels may be replaced with the keyword all to denote all instances of a component; it may also be the keyword others to specie all as yet unbound instances of a component. The generic map is used to specify the values for the generics or provide the mapping between the generic parameters of the component and the entity to which it is bound. The port map is used to specify the port bindings between the component and the bound entity. Additional examples of configuration specifications appear in the following architecture body.

In the binding for N1 and N2, the generic map specifies the mapping of generic names from the entity NOR2 to the component NOR_GATE using named association. The generic values supplied in the instantiations are, therefore, passed to the NOR2 entity through this mapping. The port binding is specified using positional association, that is, ports SO, SI, and Q of NOR_GATE component map to ports A, B, and Z, respectively, of the NOR2 entity. The configuration specification for the AND2_GATE specifies the value, 10, of the generic explicitly (using positional association) which overrides the default value, 5, specified in the entity declaration for the AND2 entity. The component declaration for AND2_GATE, in this case, should not specify any generics since the values are passed directly to the actual generics of the entity. The port mapping for the AND2_GATE is specified using named association.

```
architecture DUMMY of DUMMY is
        component NOR_GATE
                generic (RISE_TIME, FALL_TIME: TIME);
                port (S0, S1: in BIT; Q: out BIT);
        end component;
        component AND2_GATE
                port (DOUT: out BIT; DIN: in BIT_VECTOR);
        end component;
        for M1, N2: NOR_GATE
                use entity WORK.NOR2(NOR2_DELAYS)
                generic map (PT_HL => FALL_TIME, PT_LH => RISE_TIME)
                port map (S0, S1, Q);
        for all: AND2_GATE
                use entity WORK.AND2 (GENERIC_EX)
                generic map (10)
                port map (A => DIN, Z => DOUT);
        signal WR, RD, RW, S1, S2: BIT;
        signal SA: BIT_VECTOR (1 to 10);
begin
        N1: NOR_GATE generic map (2 ns, 3 ns) port map (WR, RD, RW);
        A1: AND2_GATE port map (S1, SA):
        N2: NOR_GATE generic map (4 ns, 6 ns)
                        port map (S1, SA(2), S3);
end DUMMY;
```

The entity declarations for the entities that are bound to components NOR_GATE and AND2_GATE are

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

entity AND2 is
        generic (N: NATURAL := 5);
        port (A: in BIT_VECTOR(1 to N); Z: out BIT);
end AND2;
```

How are the generic map and port map values in a component instantiation passed into its bound entity via the configuration specification? A look at the elaboration of a component instantiation helps us to understand this. Let us take the N1 instantiation in the previous architecture body as an example. Elaboration transforms this component instantiation into the following block statement.

```
N1: block          - A block for the component instantiation.
        generic (RISE_TIME, FALL_TIME: TIME); " Generics of
                --the component.
        generic map (RISE_TIME => 2 ns, FALL_TIME => 3 ns); - Generic
                - map in instantiation.
        port (S0, S1: in BIT; Q: out BIT); - Ports of component.
        port map (S0 => WR, S1 => RD, Q => RW); - Port map in
                -- instantiation.
begin
        NOR2: block          -- A block for the bound entity.

            generic (PT_HL, PT_LH: TIME); - Its generics.
            generic map (PT_HL => FALL_TIME, PT_LH => RISE_TIME);
                    -- Generic map in configuration specification.
            port (A, B: in BIT; Z: out BIT); - Its ports.
            port map (A => S0, B => S1, Z => Q);-- Port map in
                    -- specification.
            -- Other declarations in the architecture
            -- body NOR2_DELAYS appear here.
        begin
            - Statements in architecture body NOR2_DELAYS
            - appear here.
        end block NOR2;
end block M1;
```

The block Nl is created from the component instantiation of NOR_GATE. The generic map and port map for this block are the generic map and port map specified in the component instantiation for Nl, that is, they specify the mapping between the values and signals in the component instantiation statement with the generics and ports of component NOR_GATE. The inner block with label NOR2 represents the entity NOR2 that the component instantiation Nl is bound to in the configuration specification. The generic map and port map of this block specify the generic map and port map that appear in the configuration specification, that is, they specify the mapping between the component NOR_GATE and the entity NOR2.

3.4 Configuration Declaration

Configuration specifications have to appear in an architecture body. Therefore, to change a binding, it is necessary to change the architecture body and re-analyze it. This may sometimes be cumbersome and time consuming. To avoid this, a configuration declaration may be used to specify a binding.

A configuration declaration is a separate design unit, therefore, it allows for late binding of components, that is, the binding can be performed after the architecture body has been written. It is also possible to have more than one configuration declaration for

an entity, each of which defines a different set of bindings for components in a single architecture body, or possibly specifies a unique entity-architecture pair.

The typical format of a configuration declaration is

configuration *configuration-name* **of** *entity-name* ***is***
block-configuration
end [*configuration-name*];

It declares a configuration with the name, *configuration-name,* for the entity, *entity-name.* A *block-configuration* defines the binding of components in a block, where a block may be an architecture body, a block statement, or a generate statement. A block configuration is a recursive structure of the form

for *block-name*
component-configurations
block-configurations
end for;

The *block-name* is the name of an architecture body, a block statement label, or a generate statement label. The top-level block is always an architecture body. A *component-configuration* binds components that appear in a block to entities and is of the form

for *list-of-comp-labels: comp-name* [**use** *binding-indication*;]
[*block-configuration*]
end for;

The block configuration that appears within a component configuration defines the bindings of components at the next level of hierarchy in the entity-architecture pair specified by the binding indication.

There are two other forms of binding indication in addition to the one shown in the previous section. These are

configuration *configuration-name* -- Form 2
open -- Form 3

In form 2, the binding indication specifies that the component instances are to be bound to a configuration of a lower level entity as specified by the configuration name. This implies that a configuration declaration with such a name must exist. In form 3, the binding indication indicates that the binding is not yet specified and that it is to be deferred. Both these forms of binding indication may also be used in a configuration specification.

Here is an example of a configuration declaration that specifies the component configurations for all component instances in architecture FA_STR of entity FULL_ADDER described in the previous section.

```
library CMOS_UB;
configuration FA_CON of FULL_ADDER is
        for FA_STR
                use WORK.all;
                for A1,A2,A3:AND2
                        use entity CMOS_LIB.BIGAND2 (AND2STR);
                end for;
                for others: OR2          --use defaults, i.e. use OR2 from
                                         -- library WORK.
                end for ;
                for all: XOR2
                        use configuration WORK.X6R2CON;
                end for;
        end for;
end FA_CON;
```

The configuration with name, FA_CON, binds architecture FA_STR with the FULL_ADDER entity. For components within this architecture body, instances Al, A2, and A3, are bound to the entity, BIGAND2, that exists in the design library, CMOS_LIB. For all instances of component OR2, the default bindings are used; these are the entities in the working library with the same names as the component names. The last component configuration shows a different type of binding indication. In this case, all component instances are bound to a configuration instead of an entity-architecture pair. All instances of component XOR2 are bound to a configuration with name XOR2CON, that exists in the working library. This type of binding may also be specified in a configuration specification.

The power of the configuration declaration lies in the fact that the subcomponents in an entire hierarchy of a design can be bound using a single configuration declaration. For example, consider a full-adder circuit composed of two half-adders and an or gate. The half-adder circuit is in turn composed of xor and and gates. The hierarchy for this full-adder is shown in Fig. 3.5.

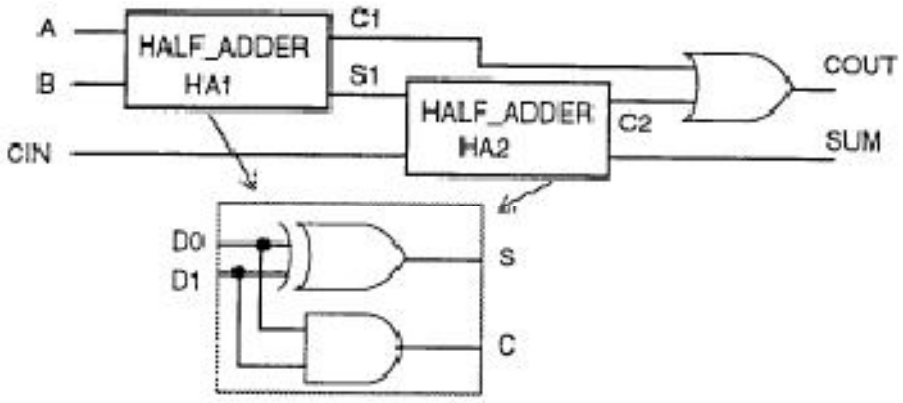


Figure 3.5 A hierarchical 1 -bit full-adder

The structural models for the full-adder and half-adder circuits are described next. A configuration declaration which specifies the bindings for components used in the entire hierarchy of the full-adder is also shown.

The top-level block configuration specifies the bindings of component instances present in the architecture body FA_WITH_HA. Instances HAI and HA2 are bound to an entity specified by the entity-architecture pair, entity HA and architecture HA_STR. The nested block configuration specifies the binding of component instances present in the architecture body HA_STR. In this way, a configuration can be nested to any arbitrary depth and can be used to bind all components in a hierarchy.

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

architecture FA_WITH_HA of FULL_ADDER is
        component HALF_ADDER
                port (A, B : in BIT; SUM, CARRY: out BIT);
        end component;
        component OR2
                port (A, B: in BIT; Z: out BIT);
        end component;
        signal S1, C1, C2: BIT;
begin
        HA1: HALF_ADDER port map (A, B, S1, C1);
        HA2: HALF_ADDER port map (S1, CIN, SUM, C2);
        O1: OR2 port map (C1, C2, COUT);
end FA_WITH_HA;

entity HA is
        port (D0, D1: in BIT; S, C: out BIT);
end HA;

architecture HA_STR of HA is
        component XOR2
                port (X, Y: in BIT; Z: out BIT);
        end component;
        component AND2
                port (L, M: in BIT; N: out BIT);
        end component;
begin
        X1: XOR2 port map (D0, D1, S);
        A1: AND2 port map (D0, D1, C);
end HA_STR;

-- A configuration for the FULL_ADDER that is built using half-adders.
library ECL;
configuration FA_HA_CON of FULL_ADDER is
        for FA_WITH_HA        - Top-level block configuration.
                for HA1, HA2: HALF_ADDER
                        use entity WORK.HA(HA_STR)
                        port map (DO=>A, D1=>B, S=>SUM, C=>CARRY);
                        for HA_STR      - Nested block configuration.
                                for all: XOR2
                                        use entity WORK.XOR2(XOR2);
                                end for;
                                for A1: AND2
                                        use configuration ECL.AND2CON;
                                end for;
                        end for;
                end for;
                for O1: OR2
                        use configuration WORK.OR2CON;
                end for;
        end for;
end FA_HA_CON;
```

The previous example shows that when components in a hierarchy are bound, a single configuration declaration may be used to replace a set of configuration specifications. If configuration specifications were used in the earlier example, they would have to be included separately in the architecture bodies, FA_WITH_HA and HA_STR, and then these bodies would have to be recompiled every time a binding is changed. Note that a component instance must not be bound both in a configuration specification and in a configuration declaration. Configurations, therefore, provide the mechanism by which architecture bodies may contain technology-independent

components. Technology-specific mappings can be specified separately using configuration declarations.

3.5 Subprograms

A subprogram defines a sequential algorithm that performs a certain computation and executes in zero simulation time. There are two kinds of subprograms:

1. *Functions:* These are usually used for computing a single value.
2. *Procedures:* These are used to partition large behavioral descriptions. Procedures can return zero or more values.

A subprogram is defined using a *subprogram body*. The typical format for a subprogram body is

subprogram-specification **is**
subprogram-item-declarations
begin
subprogram-statements -- Same as sequential-statements.
end [*subprogram-name*]**;**

The *subprogram-specification* specifies the name of a subprogram and defines its interface, that is, it defines the formal parameter names, their class (i.e., signal, variable, or constant), their type, and their mode (whether they are in, out, or inout). Parameters of mode in are read-only parameters; these cannot be updated within a subprogram body. Parameters of mode out are write-only parameters; their values cannot be used but can only be updated within a subprogram body. Parameters of mode inout can be read as well as updated.

Actual values are passed to and from a subprogram via a subprogram call. Only a signal object may be used to pass a value to a parameter of the signal class. Only a variable object may be used to pass a value to a parameter of the variable class. A constant or an expression may be used to pass a value to a parameter of constant class. When parameters are of a variable or constant class, values are passed to the subprogram by value. Arrays may or may not be passed by reference. For signal objects, the reference to the signal, its driver, or both are passed into the subprogram. What this means is that any assignment to a signal in a procedure (signals cannot be assigned values in a function because the parameters are restricted to be of input mode) affects the actual signal driver immediately and is independent of whether the procedure terminates or not. For a signal of any mode, the signal-valued attributes, STABLE, QUIET, DELAYED, and TRANSACTION (attributes are discussed in Chap. 10), cannot be used in a subprogram body.

The type of an actual value in a subprogram call must match that of its corresponding formal parameter. If the formal parameter belongs to an unconstrained type, the size of this parameter is determined from the actual value that is passed in.
The *subprogram-item-declarations* part contains a set of declarations (e.g., type and object declarations) that are accessible for use locally within the subprogram. These declarations come into effect every time the subprogram is called. Variables are also created and initialized every time the subprogram is called. They remain in existence until the subprogram completes. This is in contrast with declarations in a process statement that get initialized only once, that is at start of simulation, and any declared variables persist throughout the entire simulation run.

The *subprogram-statements* part contains sequential statements that define the computation to be performed by the subprogram. A return statement, which is also a sequential statement, is a special statement that is allowed only within subprograms. The format of a return statement is

return *{ expression]*;

The return statement causes the subprogram to terminate and control is returned back to the calling object. All functions must have a return statement and the value of the expression in the return statement is returned to the calling program. For procedures, objects of mode out and inout return their values to the calling program.

The *subprogram-name* appearing at the end of a subprogram body, if present, must be the same as the function or procedure name specified in the subprogram specification part.

Functions :

Functions are used to describe frequently used sequential algorithms that return a single value. This value is returned to the calling program using a return statement. Some of their common uses are as resolution functions, and as type conversion functions. The following is an example of a function body.

```
function LARGEST (TOTAL_NO: INTEGER; SET: PATTERN)
        return REAL is
        -- PATTERN is defined to be atype of 1-D array of
        -- floating-point values, elsewhere.
            variable RETURN_VALUE: REAL := 0.0;
begin
            for K in 1 to TOTAL_NO loop
                    if SET(K) > RETURN_VALUE then
                            RETURN_VALUE := SET(K);
                    end if;
            end loop;
            return RETURN_VALUE;
end LARGEST;
```

Variable RETURN_VALUE comes into existence with an initial value of 0.0 every time the function is called. It ceases to exist after the function returns back to the calling program.

The general syntax of a subprogram specification for a function body is function *function-name (parameter-list)* **return** *return-type*

Procedures :

Procedures allow decomposition of large behaviors into modular sections. In contrast to a function, a procedure can return zero or more values using parameters of mode out and inout. The syntax for the subprogram specification for a procedure body is

procedure *procedure-name* (*parameter-list)*

The *parameter^ist* specifies the list of formal parameters for the procedure. Parameters may be constants, variables, or signals and their modes may be in, out, or inout. If the object class of a parameter is not explicitly specified, then the object class is by default a constant if the parameter is of mode in, else it is a variable if the parameter is of mode out or inout.

A procedure can normally be used simultaneously as a concurrent and a sequential statement. However, if any of the procedure parameters are of the variable class, the procedure would be restricted to be used as a sequential procedural call, since variables

can only be defined inside of a process. Concurrent procedure calls are useful in representing frequently used processes.

A procedure body can have a wait statement while a function cannot. Functions are used to compute values that are available instantaneously. Therefore, a function cannot be made to wait, for example, it cannot call a procedure with a wait statement in it. A process that calls a procedure with a wait statement cannot have a sensitivity list. This follows from the fact that a process cannot be sensitive to signals and also be made to wait simultaneously. Since a procedure can have a wait statement, any variables declared in the procedure retain their values through this wait period and cease to exist only when the procedure terminates.

Declarations :

A subprogram body may appear in the declarative part of the block in which a call is made. This is not convenient if the subprogram is to be shared by many entities. In such cases, the subprogram body can be described at one place, possibly in a package body and then in the package declaration, the corresponding *subprogram declaration* is specified. If this package declaration is included in other design units using context clauses, the subprograms can then be used in these design units. A subprogram declaration describes the subprogram name and the list of parameters without describing the internal behavior of the subprogram, that is, it describes the interface for the subprogram. The syntax of a subprogram declaration is

*subprogram-specification***;**

Two examples of procedure and function declarations are shown next.

```
procedure ARITH_UNIT (A, B: in INTEGER; OP: in OP_CODE;
                      Z: out INTEGER; ZCOMP; out BOOLEAN);
function VRISE (signal CLOCK_NAME: BIT) return BOOLEAN;
```

Another reason subprogram declarations are necessary is to allow two subprograms to call each other recursively, for example,

```
procedure P ( ) . . .
begin
        A := Q (B);  -- illegal function call.
end P;

function Q ( ) . . .
begin
        P( );
end Q;
```

The call to function Q in procedure P is illegal since Q has not yet been declared. This can be corrected by writing the function declaration for Q either before procedure P or inside the declarative part of procedure P.

3.6 Subprogram Overloading

Sometimes it is convenient to have two or more subprograms with the same name. In such a case, the subprogram name is said to be *overloaded.* For example, consider the following two declarations.

function COUNT (ORANGES: INTEGER) **return** INTEGER;
function COUNT (APPLES: BIT) **return** INTEGER;

Both functions are overloaded since they have the same name, COUNT, and have different parameter types. When a call to either function is made, it is easily possible to identify the exact function to which the call was made from the type of the actual parameters passed. For example, the function call

COUNT(20)

refers to the first function since 20 is of type INTEGER, while the function call

COUNT('1')

refers to the second function, since the type of actual parameter is BIT. If two overloaded subprograms have the same parameter types and result types, then it is possible for one subprogram to hide the other subprogram. This can happen, for example, if a subprogram is declared within another subprogram's scope. Here is an example.

```
architecture HIDING of DUMMY_ENTITY is
        function ADD (A, B: BIT_VECTOR) return BIT_VECTOR is
        begin
                -- Body of function here.
        end ADD;

begin
        SUM_IN_ARCH<=ADD(IN1, IN2);
        process
                function ADD (C, D: BIT_VECTOR) return BIT_VECTOR is
                begin
                        -- Body of function here.
                end ADD;
        begin
                SUM_IN_PROCESS <= ADD (IN1, IN2);
                SUM_IN_ARCH <= HIDING.ADD(IN1, IN2);
        end process;
end HIDING;
```

The function ADD declared in the architecture body is hidden within the process because of the second function ADD that is declared within the declarative part of the process. This function can still be accessed by qualifying the function name with the architecture name as shown in the second statement in the process.

It is also possible for two overloaded subprograms to be directly visible within a region, for example, caused by using use clauses. In such a case, a subprogram call may be ambiguous, and hence an error, if it is not possible to determine which of the overloaded subprograms is being called. Here is an example.

```
package P1 is
        function ADD (A, B: BIT_VECTOR) return BIT_VECTOR;
end P1;

package P2 is
        function ADD (X, Y: BIT_VECTOR) return BIT_VECTOR;
end P2;

use WORK.P1. all, WORK.P2.all;
architecture OVERLOADED of DUMMY_ENTITY is
begin
        SUM_CORRECT <= ADD (X => IN1, Y => IN2);
        SUM_ERROR <= ADD (IN1, IN2);   -- An error.
end OVERLOADED;
```

The function call in the first signal assignment statement is not an error since it refers to the function declared in package P2, while the call in the second signal assignment statement is ambiguous and hence an error.

It is also possible for two subprograms to have the same parameter types and result types but have a different number of parameters. In this case, the number of actual values supplied in the subprogram call identifies the correct subprogram. Here is an example of such a set of functions that determine the smallest value from a set of 2,4, or 8 integers.

```
function SMALLEST (A1, A2: INTEGER) return INTEGER;
function SMALLEST (A1, A2, A3, A4: INTEGER) return INTEGER;
function SMALLEST (A1, A2, A3, A4, A5, A6, A7, A8: INTEGER)
        return INTEGER;
```

A call such as

. . . SMALLEST (4, 5) . . .

refers to the first function, while the function call

. . . SMALLEST (20, 45,52,1,89,67,91,22)...

refers to the third function. This flexibility helps in writing code that is easy to decipher since the same subprogram name can be made to serve differently when used with a different set of inputs.

3.7 Operator Overloading

Operator overloading is one of the most useful features in the language. When a standard operator symbol is made to behave differently based on the type of its operands, the operator is said to be overloaded. The need for operator overloading arises from the fact that the predefined operators in the language are defined for operands of certain predefined types. For example, the and operation is defined for arguments of type BIT and BOOLEAN only. What if the arguments were of type MVL (where MVL is a user-defined enumeration type with values 'U', '0', '1' and 'Z')? In such a case, it is possible to redefine the and operation as a function that operates on arguments of type MVL. The and operator is then said to be overloaded. The operator in the expression

S1 and S2

where S1 and S2 are of type MVL, would then refer to the and operation that was defined by the model writer as a function. The operator in the expression

CLK1 **and** CLK2

where CLK1 and CLK2 are of type BIT, would refer to the predefined and operator. Function bodies are written to define the behavior of overloaded operators. Such a function has, at most, two parameters; the first one refers to the left operand of the operator and the second parameter, if present, refers to the second operand. Here are some examples of function declarations for such function bodies.

```
type MVL is ('U', '0', '1', 'Z');
function "and" (L, R: MVL) return MVL;
function "or" (L, R: MVL) return MVL;
function "not" (R: MVL) return MVL;
```

Since the **and, or,** and **not** operators are predefined operator symbols, they have to be enclosed within double quotes when used as overloaded operator function names. Having declared the overloaded functions, the operators can now be called using two different types of notations:

1. standard operator notation,

2. standard function call notation.

Here are some examples of these two types of notations based on the overloaded operator function declarations that appeared earlier.

```
signal A, B, C: MVL;
signal X, Y, Z: BIT;

A <= 'Z' or '1';                    -- #1: standard operator notation.
B <= "or" ('0', 'Z);                -- #2: function call notation.
X <= not Y;                         --#3
Z<= X and Y;                        --#4
C <= (A or B) and (not C);          -- #5
Z<= ( X and Y) or A;                --#6
```

The or operator in the first statement refers to the overloaded operator because the type of the left operand is MVL. This is the standard operator notation since the overloaded operator symbol appears just like the standard operator symbol. An example of the function call notation is shown in the second statement in which the overloaded function, or, is explicitly called. The operators in the third and fourth statements refer to the predefined operators since their operands are of type BIT. The sixth statement would be an error assuming that there are no other overloaded or operators defined with the first parameter type of BIT and the second parameter type of MVL.

The last example brings up a very interesting point. In overloaded operator functions, it is not necessary for both operands to have the same type. In the previous case, if another or overloaded function with a declaration such as

function "or" (L: BIT; R: MVL) **return** BIT:

were defined, the sixth assignment statement would not be an error.

3.8 Package Declaration

A package declaration contains a set of declarations that may possibly be shared by many design units. It defines the interface to the package, that is, it defines items that can be made visible to other design units, for example, a function declaration. A package body, in contrast, contains the hidden details of a package, for example, a function body.

The syntax of a package declaration is

package *package-name* **is**
package-item-declarations "> These may be:
- subprogram declarations ~ type declarations
- subtype declarations
- constant declarations
- signal declarations
- file declarations
- alias declarations
- component declarations
- attribute declarations
- attribute specifications
- disconnection specifications
- use clauses

end [*package-name*] ;

An example of a package declaration is given next.

```
package SYNTH_PACK is
        constant LOW2HIGH: TIME := 20ns;
        type ALU_OP is (ADD, SUB, MUL, DIV, EQL);
        attribute PIPELINE: BOOLEAN;
        type MVL is ('U', '0', '1', 'Z');
        type MVL_VECTOR is array (NATURAL range <>) of MVL;
        subtype MY_ALU_OP is ALU_OP range ADD to DIV;
        component NAND2
                port (A, B: in MVL; C: out MVL);
        end component;
end SYNTH_PACK;
```

Items declared in a package declaration can be accessed by other design units by using the library and use context clauses. The set of common declarations may also include function and procedure declarations and deferred constant declarations. In this case, the behavior of the subprograms and the values of the deferred constants are specified in a separate design unit called the package body. Since the previous package example did not contain any subprogram declarations and deferred constant declarations, a package body was not necessary.

Consider the following package declaration.

```
use WORK.SYNTH_PACK.all;
package PROGRAM_PACK is
        constant PROP_DELAY: TIME;      -A deferred constant.
        function "and" (L, R: MVL) return MVL;
        procedure LOAD (signal ARRAY_NAME: inout MVL_VECTOR;
                START_BIT, STOP_BIT, INT_VALUE: in INTEGER);
end PROGRAM_PACK;
```

In this case, a package body is required.

3.9 Package Body

A package body primarily contains the behavior of the subprograms and the values of the deferred constants declared in a package declaration. It may contain other declarations as well, as shown by the following syntax of a package body.

```
package body package-name is
        package-body-item-daclarations "> These are:
                - subprogram bodies         -- complete constant declarations
                - subprogram declarations
                - type and subtype declarations
                - file and alias declarations
                - use clauses
end [ package-name ];
```

The package name must be the same as the name of its corresponding package declaration. A package body is not necessary if its associated package declaration does not have any subprogram or deferred constant declarations. The associated package body for the package declaration, PROGRAM_PACK, described in the previous section is

```
package body PROGRAM_PACK is
        constant PROP_DELAY: TIME := 15ns;
        function "and" (L, R: MVL) return MVL is
        begin
                return TABLE_AND(L, R);
                -- TABLE_AND is a 2-D constant defined elsewhere.
        end "and";
        procedure LOAD (signal ARRAY_NAME: inout MVL_VECTOR;
                START_BIT, STOP_BIT, INT_VALUE: in INTEGER) is
                -- Local declarations here.
        begin
                -- Procedure behavior here.
        end LOAD;
end PROGRAM_PACK;
```

An item declared inside a package body has its scope restricted to be within the package body and it cannot be made visible in other design units. This is in contrast to items declared in a package declaration that can be accessed by other design units. Therefore, a package body is used to store private declarations that should not be visible, while a package declaration is used to store public declarations which other design units can access. This is very similar to declarations within an architecture body which are not visible outside of its scope while items declared in an entity declaration can be made visible to other design units. An important difference between a package declaration and an entity declaration is that an entity can have multiple architecture bodies with different names, while a package declaration can have exactly one package body, the names for both being the same.

A subprogram written in any other language can be made accessible to design units by specifying a subprogram declaration in a package declaration without a subprogram body in the corresponding package body. The association of this subprogram with its declaration in the package is not defined by the language and is, therefore, tool implementation-specific.

3.10 Design Libraries

A compiled VHDL description is stored in a design library. A design library is an area of storage in the file system of the host environment. The format of this storage is not defined by the language. Typically, a design library is implemented on a host system as a file directory and the compiled descriptions are stored as files in this directory. The management of the design libraries is also not defined by the language and is again tool implementation-specific.

An arbitrary number of design libraries may be specified. Each design library has a logical name with which it is referenced inside a VHDL description. The association of the logical names with their physical storage names is maintained by the host environment. There is one design library with the logical name, STD, predefined in the language; this library contains the compiled descriptions for the two predefined packages, STANDARD and TEXTIO. Exactly one design library must be designated as the working library with the logical name, WORK. When a VHDL description is compiled, the compiled description is always stored in the working library. Therefore, before compilation begins, the logical name WORK must point tonne of the design libraries. Figure 3.6 shows a typical compilation scenario.

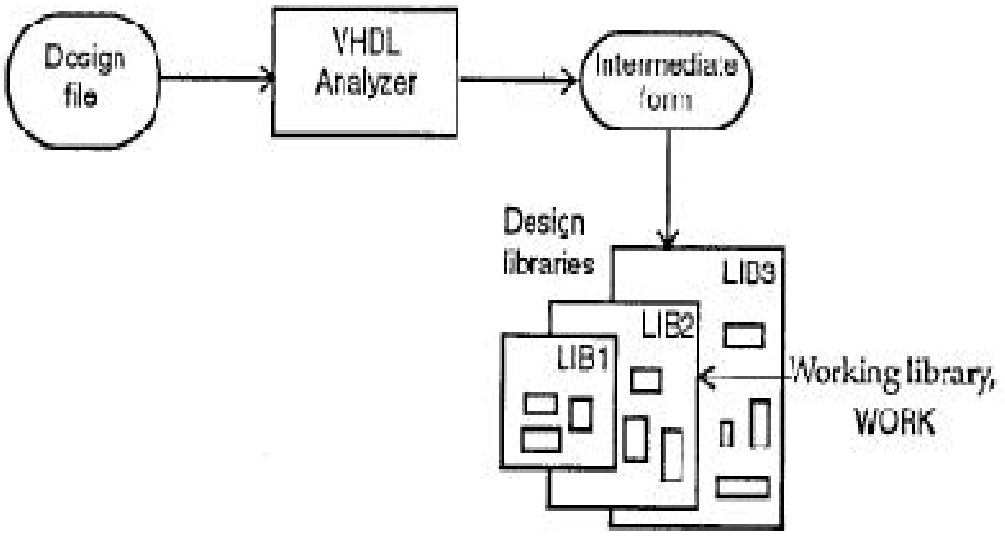


Figure 3.6 Atypical compilation process.

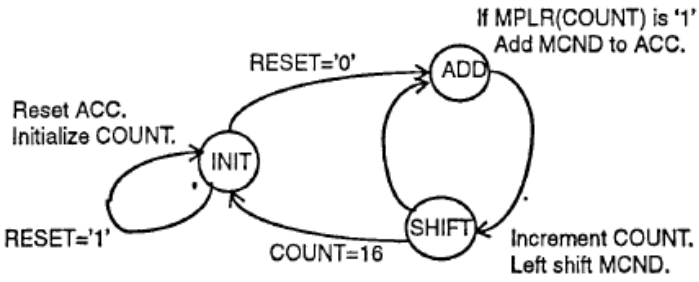
The VHDL source is present in an ASCII file called the *design file.* This is processed by the VHDL analyzer, which after verifying the syntactic and semantic correctness of the source, compiles it into an intermediate form. The intermediate form is stored in the design library that has been designated as the working library.

3.11 State Machine Modeling

State machines can usually be modeled using a case statement in a process. The state information is stored in a signal. The multiple branches of the case statement contain the behavior for each state. Here is an example of a simple multiplication algorithm represented as a state machine. When RESET signal is high, the accumulator ACC and the counter COUNT are initialized. When RESET goes low, multiplication starts. If the bit of the multiplier MPLR in position COUNT is 1', the multiplicand MCND is added to the accumulator. Next, the multiplicand is left-shifted by one bit and the counter is incremented. If COUNT is 16, multiplication is complete and the DONE signal is set high. If not, the COUNT bit of the multiplier MPLR is checked and the process repeated. The state diagram is shown in Fig. 12.11 and the corresponding state machine model is shown next.

The signal MPY_STATE holds the state of the model. Initially the model is in state INIT and stays in this state as long as signal RESET is '1'. When RESET gets the value '0', the accumulator ACC is cleared, the counter COUNT is reset and the multiplicand MCND is loaded into a temporary variable MCND_TEMP, and model advances to state ADD. When model is in ADD state, the multiplicand in MCND_TEMP is added to ACC only if the bit at the COUNT position of the multiplier is a '1', and then the model advances to state SHIFT. In this state, the multiplier is left shifted once, counter is incremented and if the counter value is 16, signal DONE is set to '1' and model returns to state INIT. At this time, ACC contains the result of the multiplication. If the counter value was less than 16, the mode? repeats itself going through states ADD and SHIFT until the counter value becomes 16.

State transitions occur at every falling clock edge; this is specified by the wait statement. The mode of signal ACC is set to buffer since the value is read and updated within the model




```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity MULTIPLY is
        port (MPLR, MCND: in MVL_VECTOR(15 downto 0);
                -- MPLR is multiplier, MCND is multiplicand.
                CLOCK, RESET: in MVL;
                DONE: out MVL; ACC: buffer MVL_VECTOR(31 downto 0));
end MULTIPLY;

architecture STATE_MACHINE of MULTIPLY is
        type STATE_TYPE is (INIT, ADD, SHIFT);
        signal MPY_STATE: STATE_TYPE; -- Initial state is INIT
begin
        process
                variable COUNT: NATURAL ;
                variable MCND_TEMP: MVL_VECTOR(31 downto 0);
        begin
                wait until (CLOCK = '0' and CLOCK'EVENT);
                -- Alternately, "wait until (CLOCK = '0' and
                -- not CLOCK'STABLE);";
                --Both are equivalent in this context.
                case MPY_STATE is
                        when INIT=>
                                if RESET = '1' then


                                        MPY_STATE <= INIT;
                                        -- The above statement is not really
                        necessary
                                        --since MPY_STATE will retain its old value.
                                else
                                        ACC <= (others => '0');
                                        COUNT := 0;
                                        MPY_STATE <= ADD;
                                        DONE <= '0';
                                        MCND_TEMP(15 downto 0) := MCND;
                                        MCND_TEMP(31 downto 16) := (others =>
                                '0');
                                end if;
                        when ADD =>
                                if (MPLR(COUNT) = '1') then
                                        ACC <= ACC + MCND_TEMP;
                                end if;
                                MPY_STATE <= SHIFT;
                        when SHIFT =>
                                - Left-shift MCND:
                                MCND_TEMP := MCND_TEMP(30 downto 0) &
                                '0';
                                COUNT := COUNT+1;
                                if (COUNT = 16) then
                                        MPY_STATE <= INIT;
                                        DONE <= '1';
                                else
                                        MPY_STATE <= ADD;
                                end if;
                end case;
        end process;
end STATE_MACHINE;
```

3.12 Modeling Moore FSM

The output of a Moore finite state machine (FSM) depends only on the state and not on its inputs. This type of behavior can be modeled using a single process with a case statement that switches on the state value. An example of a state transition diagram for a Moore finite state machine is shown in Fig. 12.16 and its corresponding behavior model appears next.

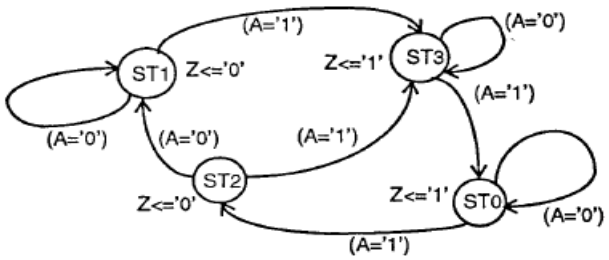


Figure: State diagram of a Moore machine

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity MOORE_FSM is
        port (A, CLOCK: in BIT; Z: out MVL);
end MOORE_FSM;

architecture FSM_EXAMPLE of MOORE_FSM is
        type STATE_TYPE is (ST0, ST1, ST2, ST3);
        signal MOORE_STATE: STATE_TYPE;
begin
        process (CLOCK)
        begin
                if (CLOCK = '0' and CLOCK'EVENT) then
                        case MOORE_STATE is
                                when ST0 =>
                                        Z<='1';
                                        if (A = '1') then
                                                MOORE_STATE <= ST2;
                                        end if;
                                when ST1 =>
                                        Z<= '0';
                                        if (A = '1') then
                                                MOORE_STATE <= ST3;
                                        end if;
                                when ST2 =>
                                        Z <= '0';
                                        if (A = '0') then
                                                MOORE_STATE <= ST1:
                                        else
                                                MOORE_STATE <= ST3;
                                        end if;
                                when ST3 =>
                                        Z<='1';
                                        if (A = '1') then
                                                MOORE_STATE <= ST0;


                                                        end if;
                                end case;
                        end if;
                end process;
        end FSM_EXAMPLE;
```

3.13 Modeling Mealy FSM

In a Mealy type finite state machine, the outputs not only depend on the state of the machine but also on its inputs. This type of finite state machine can also be modeled in a similar style as the Moore example case, that is, using a single process. To show the variety of the language, a different style is used to model a Mealy machine. In this case, we use two processes, one process that models the synchronous aspect of the finite state machine and the second process models the combinational part of the finite state machine. Here is an example of a state transition table, shown in Fig. 12.17, and its corresponding behavior model.

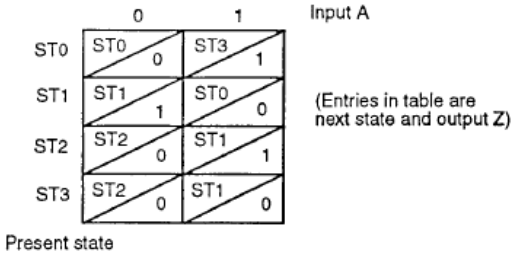


Figure: State transition table for a Mealy machine

```
library ATTLIB; use ATTUB.ATT_PACKAGE.all;
entity MEALY_FSM is
        port (A, CLOCK: in BIT; Z out MVL);
end MEALY_FSM;

architecture YET_ANOTHER_EXAMPLE of MEALY_FSM is
        type MEALY_TYPE is (ST0, ST1, ST2, ST3);
        signal P_STATE, N_STATE: MEALY_TYPE;
begin
        SEQ_PART: process (CLOCK)
        begin
                -- Synchronous section:
                if (CLOCK = '0' and CLOCK'EVENT) then
                        P_STATE <= N_STATE;
                end if;
        end process SEQ_PART;
        COMB_PART: process (P_STATE, A)
        begin
                case P_STATE is
                        when ST0 =>
                                if (A = '1') then
                                        Z <= '1 ': N_STATE <= ST3:
                                else
                                        Z<='0';
                                end if;
                        when ST1 =>
                                if (A = '1') then
                                        Z <= '0';
                                        N_STATE <= ST0;
                                else
                                        Z<='1' ;
                                end if;
                        when ST2 =>
                                if (A = '0') then
                                        Z <= '0';
                                else
```

```
                    Z <=' 1'; N_STATE <= ST1;
                end if;
            when ST3 => Z<='0';
                if (A = '0') then
                    N_STATE <= ST2;
                else
                    N_STATE <= ST1;
                end if;
        end case;
    end process COMB_PART;
end YET_ANOTHER_EXAMPLE;
```

In this type of finite state machine, it is important to put the input signals in the sensitivity list for the combinational part process, since the outputs may directly depend on the inputs independent of the clock. Such a condition does not occur in a Moore finite state machine since outputs depend only on states and state changes occur synchronously.