ADDITIONAL TOPICS IN VHDL

Generics and attributes are useful in describing *parameterized designs.*

An attribute is a value, function, type, range, signal, or a constant that can be associated with certain names within a VHDL description. These names could be among others, an entity name, an architecture name, a label, or a signal. The main two types of attributes are: User Defined Attributes & Predefined Attributes.

A large number of attributes are predefined in the language. These are described later. The language also provides the facility to associate user-defined attributes to names.

User-Defined Attributes

User-defined attributes are constants of any type. They are declared using attribute declarations. An *attribute declaration* declares an attribute name and its type and has the following form:

attribute *attribute-nama: value-type*;

For example,

```
type COMP_LOCATION is
record
X, Y; INTEGER;
end record;

type INCHES is range 0 to 1000
units
micron;
end units;

type FARADS is range 0 to 5000
units
pf;
end units;

attribute PLACEMENT: COMP_LOCATION;
attribute LENGTH: INCHES;
attribute CAPACITANCE: FARADS;
```

These declared attributes have not yet been associated with any name. An *attribute specification* is used to associate an attribute with a name and to assign a value to the attribute. The syntax for an attribute specification is

attribute *attribute-name* **of** *item-names: name-class* **is** *expression*;

The *item-names* is a list of one or more names of an entity, architecture, configuration, component, label, signal, variable, constant, type, subtype, package, procedure, or a function. The *name-class* indicates the class type, that is, whether it is an entity,

architecture, label, or others. The attribute name must have been declared earlier using an attribute declaration. The attribute specification associates an attribute with a group of names that belong to a name class that has the value as specified by the expression (the value of expression must belong to the type of attribute). Some examples of attribute specifications are

```
attribute CAPACITANCE of CLK, RESET: signal is 20 pf;
attribute LENGTH of RX_READY: signal is 3 micron;
```

In the first example, the attribute CAPACITANCE is associated with two sig nals CLK and RESET, each of which has a value of 20 pf.
The item name in the attribute specification can also be replaced witl the keyword all to indicate all names belonging to that name class. In the fol lowing example, the attribute CAPACITANCE is associated with all variable: and the attribute value is set to 0 pf.

```
attribute CAPACITANCE of all: variable is 0 pf;
```

After having created an attribute and then having associated it with a name, the value of the attribute can then be used in an expression by referring to

item-name ' attribute-name -- The single quote is often read as "tick".

For example, RX_READY'LENGTH has the value of 3 micron. Here is a bigger example.

```
architecture NAND_PLACE of NAND_GATE is
component NAND_COMP
port (IN1, IN2: in BIT; OUT1: out BIT);
end component;

type COMP_LOCATION is
record
X, Y: INTEGER;
end record;

attribute PLACEMENT: COMP_LOCATION;
attribute PLACEMENT of N1: label is (50, 45);
signal PERIMETER: INTEGER;
signal A, B, Z: BIT;

begin
N1: NAND_COMP port map (A, B, Z);
PERIMETER <= 2 * (N1'PLACEMENTS +
N1'PLACEMENT.Y);
end NAND_PLACE;
```

User-defined attributes are useful for annotating VHDL models with tool-specific information.

Predefined Attributes

There are five classes of predefined attributes.

1. Value attributes: these return a constant value
2. Function attributes: calls a function that returns a value
3. Signal attributes: creates a new signal
4. Type attributes: returns a type name
5. Range attributes: returns a range

Predefined attributes have a number of very important applications.

- Attributes can be used to detect clock edges,
- perform timing checks in concert with **ASSERT** statements,
- return range information about unconstrained types, and much more.

Function Signal Attributes

Function signal attributes are used to return information about the behavior of signals. These attributes can be used to report whether a signal has just changed value, how much time has passed since the last event transition, or what the previous value of the signal was.

There are five attributes that fall into this category.

- **S'EVENT**, which returns true if an event occurred during the current delta; otherwise, returns false
- **S'ACTIVE**, which returns true if a transaction occurred during the current delta; otherwise, returns false
- **S'LAST_EVENT**, which returns time elapsed since the previous event transition of signal
- **S'LAST_VALUE**, which returns previous value of **S** before the last event
- **S'LAST_ACTIVE**, which returns time elapsed since the previous transaction of signal

Attributes 'EVENT and 'LAST_VALUE

Attribute **'EVENT** is very useful for determining clock edges. By checking if a signal is at a particular value, and if the signal has just changed, It can be deduced that an edge has occurred on the signal.

Following is an example of a rising edge detector:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY dff IS
PORT( d, clk : IN std_logic;
PORT( q : OUT std_logic);
END dff;

ARCHITECTURE dff OF dff IS
```

```
BEGIN
PROCESS(clk)
BEGIN
IF ( clk = '1') AND ( clk'EVENT ) THEN
q <= d;
END IF;
END PROCESS;
END dff;
```

The above code transfers the **d** input to the **q** output, on a rising edge of the **clk** using the **'EVENT** attribute. If the value of the **clk** input is a **'1'** & the value has just changed, then a rising edge must have occurred. An **'X'** value to a **'1'** value also looks like a rising edge when it is not. One more check is made to make certain that the last value of the **clk** input was a **'0'** before the new event occurred. Correct using the **'LAST_VALUE** attribute as shown below

```
IF ( clk = '1' ) AND ( clk'EVENT )
and ( clk'LAST_VALUE = '0') THEN
q <= d;
END IF;
```

Attribute **'LAST_EVENT** returns the time since the previous event occurred on the signal. This attribute is very useful for implementing timing checks, such as setup checks, hold checks, and pulse width checks.

```
IF ( clk = '1' ) and ( clk'EVENT ) THEN
ASSERT ( d'LAST_EVENT >= setup_time )
REPORT "setup violation"
SEVERITY ERROR;
END IF;
```

The **ASSERT** statement checks to see that input **d** has not had an event during the setup time passed in by the generic **setup time**.

Attribute **d'LAST_EVENT** returns the time since the most recent event on signal **d**. If the time returned is less than the setup time, the assertion fails and reports a violation.

Attributes **'ACTIVE** and **'LAST_ACTIVE** trigger on transactions of the signal attached to AND events. A transaction on a signal occurs when a model in or inout port has an event occur that triggers the execution of the model. The model is executed, but the result of the execution produces the same output values. For instance, if an AND gate has a **'1'** value on one input and a **'0'** on the other, the output value is **'0'**. If the input with a **'1'** value changes to a **'0'** value, the output

remains **'0'**; **no event is generated, but a transaction will have been generated on the output of the AND gate.**

These signal attributes create new signals from the signals with which they are associated. These attributes, therefore, create **implicit signals** as compared to **explicit signals** that are created using **signal declarations**.

If S is a signal object, then

- **S'DELAYED (T):**

is a new signal that is the same type as signal S but delayed from S by time T. If T is not specified, a delay of 0ns is assumed.

- **S'STABLE (T):**

is a boolean signal that is true when signal S has not had any event for time T. If T is not specified, it implies current delta.

- **S'QUIET (T):**

creates a boolean signal that is true when S has not been active for time T.

- **S'TRANSACTION:**

creates a bit type signal that toggles its value every time signal S becomes active.

Difference between function attributes for signals and signal attributes

Signal attributes create new signals, and therefore, **cause events** when used in concurrent statements. whereas **function attributes** for signals **do not create** new signals, and therefore, do not create new events.

Signal attributes can, therefore, be used wherever a signal is expected, for example, in the sensitivity list of a process statement or as a signal parameter in a procedure.

For example,

```
PSTABLE: process (A, B, CLK'STABLE)
begin
. . .
end process;
```

It would be illegal to use CLK'EVENT since only signals are allowed in the sensitivity list of a process. Therefore, function attributes of signals should not be used in expressions where the intention is to create new event

Value Attributes

If T is any scalar type or subtype,

- T'LEFT: returns the left bound, that is, the leftmost value, of T.
- T'RIGHT: returns the right bound, that is, the rightmost value, of T.
- T'HIGH: returns the upper bound, that is, the value at the highest position number, of T.
- T'LOW: returns the lower bound, that is, the value at the lowest position number, of T.

For example, if

```
type ALLOWED_VALUE is range 31 downto 0;
type WEEK_DAY is (SUN, MON, TUE, WED, THU, FRI, SAT);
subtype WORK_DAY is WEEK_DAY range FRI downto MON;
```

then the following equivalence relations are true:

- ALLOWED_VALUE'LEFT = 31
- ALLOWED_VALUE'HIGH = 31
- ALLOWED_VALUE'RIGHT = 0
- ALLOWED_VALUE'LOW = ALLOWED_VALUE'RIGHT
-

If A is a constrained array object, then

- A'LENGTH(N) : returns the number of elements in the Nth dimension (N=l if not specified).

For example, if

- **signal** TX_BUS: MVL_VECTOR (7 **downto** 0);
- then TX_BUS'LENGTH = 8.
-

Attributes along with ASSERT statements are used for error checking

```
assert (NOW <= 1000 ns)
report "Simulation completed successfully"
severity ERROR;
```

Assertion would fail when simulation time exceeds 1000 ns and simulation would stop (assuming that the simulator stops on getting a severity level of ERROR).
Severity – 4 levels – note, warning, error, failure

assert *boolean-expression*
[**report** *string-expression*]
[**severity** *expression]*:

If the value of the boolean expression is false, the report message is printed along with the severity level. Here is a model of a D-type rising-edge-triggered flip-flop that uses assertion statements to check for setup and hold times.

Function Attributes

These attributes represent functions that are called to obtain a value. They are often used to convert values from an enumeration or physical type to an integer type. If T is a discrete type, a physical type, or a subtype, then

• T'POS(V): returns the position number of the value V in the ordered list of values of T.
• T'VAUP): returns the value of the type that corresponds to position P.
• T'SUCC(V): returns the value of the parameter whose position is one larger than the position of value V in T.
• T'PRED(V): returns the value of the parameter whose position is one less than the position of value V in type T.
• T'LEFTOF(V): returns the value of the parameter that is to the left of value V in type T.
• T'RIGHTOF(V): returns the value of the parameter that is to the right of value V in type T.

For ascending ranges,

T'SUCC(X) = T'RIGHTOF(X)
T'PRED(X) = T'LEFTOF(X)

For descending ranges,

T'SUCC(X) = T'LEFTOF(X)
T'PRED(X) = T'RIGHTOF(X)

For example, if

```
type STATUS is (SILENT, SEND, RECEIVE);
subtype DELAY_TIME is TIME range 50 ns downto 10 ns;
```

then the following equivalence relations are true:

```
STATUS'POS(SEND) = 1
STATUS'VAL(2) = RECEIVE
DELAY_TIME'SUCC(21 ns) = 22 ns
DELAY_TIME'PRED(10 ns) is an error
DELAY_TIME'LEFTOF(29 ns) = 30 ns

DELAY_TIME'SUCC(29 ns) = DELAY_TIME'LEFTOF(29 ns)
DELAY_TIME'RIGHTOF(11 ns) = 10ns
DELAY_TIME'PRED(11 ns) = DELAY_TIME'RIGHTOF(11 ns)
STATUS'SUCC(RECEIVE) will cause a run-time error
STATUS'PRED(RECEIVE) = SEND
STATUS'LEFTOF(RECEIVE) = STATUS'PRED(RECEIVE)
STATUS'SUCC(SILENT) = SEND
STATUS'RIGHTOF(SILENT) = STATUS'SUCC(SILENT)
```

Inertial and transport Delay Model

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 *delta delay is* a very small delay (infinitesimally small). It does not correspond to any real delay and actual simulation time does not advance. This delay models hardware where a minimal amount of time is needed for a change to occur, for example, in performing zero delay simulation. Delta delay allows for ordering of events that occur at the same simulation time during a simulation. Each unit of simulation time can be considered to be composed of an infinite number of delta delays. Therefore, an event always occurs at a real simulation time plus an integral multiple of delta delays. For example, events can occur at 15 ns, 15 ns+1A, 15 ns+2A, 15 ns+3A, 22 ns, 22 ns+A, 27 ns, 27 ns+A, and so on.

Aside from the delta delay model, there are two other types of delay models that can be used with signal assignments, inertial and transport.

Inertial Delay Model

Inertial delay models the delays often found in switching circuits. It is the default delay model in digital circuits. This delay model is often used to filter out unwanted spikes and transients on signals. It represents the time for which an input value must be stable before the value is allowed to propagate to the output. In addition, the value appears at the output after the specified delay. If the input is not stable for the specified time, no output change occurs.

Z <= A after 10ns:

is an example of a non-inverting buffer with an inertial delay of 10 ns.

Transport delay model

Transport delay models the delays in hardware that do not exhibit any inertial delay. This delay represents pure propagation delay, that is, any changes on an input is transported to the output, no matter how small, after the specified delay. To use a transport delay model, the **keyword transport** must be used in a signal assignment statement.

Effect of Inertial Delay on Signal Drivers

When inertial delays are used, both the signal value being assigned and the delay value affect the deletion and addition of transactions. If the delay of the new transaction is earlier than an existing transaction, the latter is deleted and the new one is added at the end of the driver, regardless of the signal values of the two transactions. Note that this is the same as rule 2 for the transport delay case. On the other hand, if the delay of the new transaction is greater than an already existing one, the signal values of the two transactions are compared. If they are the same, the new transaction is simply added at the end of the driver, if not, the existing one is deleted

before adding the new transaction. Deletion occurs for every existing transaction with a signal value that is different from the new transaction.

Effect of Transport Delay on Signal Drivers

1. If the delay time of the new transaction is greater than those of all the transactions already present on the driver, then the new transaction is added at the end of the driver.
2. If the delay time of the new transaction is earlier than or equal to one or more transactions on the driver, then these transactions are deleted from the driver and the new transaction is added at the end of the driver.

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. Need for operator overloading -- 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.

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.

After declaring the overloaded functions, the operators can now be called using two different types of notations: standard operator notation, & standard function call notation.

```
signal A, B, C: MVL;
signal X, Y, Z: BIT;
-- are examples of standard operator notation

A <= 'Z' or '1';
```

The or operator 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.

-- A function call notation is one in which the overloaded function, **or**, is explicitly called.

B <= **"or"** ('0', 'Z);

In overloaded operator functions, it is not necessary for both operands to have the same type. For example if another or overloaded function with a declaration such as

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

is used we can use the below statement

signal A, B, C: MVL;
signal X, Y, Z: BIT;
Z<= (X **and** Y) **or** A;

VHDL arithmetic operators, + and – defined to operate on integers not on bit vectors Use a function or a procedure for adding 2 bit vectors. Hence using operator overloading, we extend definition of "+".Overloading can be applied to procedures & functions. Several procedures can have the same name. Type of actual parameters in the procedure call determines which version of procedure is called.

MULTIVALUED LOGIC & SIGNAL RESOLUTION

Generally we use 2-VALUED BIT logic i.e., 0 & 1. A 3rd value say 'Z' – represents high impedance state useful to represent tristate buffers & busses. Similarly a 4th value say 'X'– represents unknown state which may occur if initial value of a signal is unknown or a signal is driven simultaneously to 2 conflicting values 0 & 1, the output may assume X.

IEEE-1164 Standard logic

It is a 9-Valued Logic. Here Unknown, 0, 1 come in 2 strengths-forcing & weak. When Forcing & weak tied – forcing dominates. Eg. 0 & H tied – result is 0. H represented by o/p of a pull-up resistor. One of the Subset is Z01X

'U'	Uninitialized
'X'	Forcing unknown
'0'	Forcing 0
'1'	Forcing 1
'Z'	High impedance
'W'	Weak unknown
'L'	Weak 0
'H'	Weak 1
'-'	Don't care

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.

An example of a generic N-input and gate is sho\vn next.

```
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 example demonstrates 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;
```

Other uses of generics include modeling ranges of subtypes, for example,

subtype ALUBUS **is** INTEGER **range** TOP **downto** 0;
-- TOP is a generic.

Generics can also be used to control the number of instantiations of a component in a generate statement (generate statements are discussed next)

Generate statements

Generate statements give the designer the ability to create replicated structures, or select between multiple representations of a model. Concurrent statements can be conditionally selected or replicated during the elaboration phase using the generate statement.

There are two forms of the generate statement.

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

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** *identifier* **in** *discrete-range* **generate**
[begin]
concurrent-statement(s)
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 below Fig., 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;
```

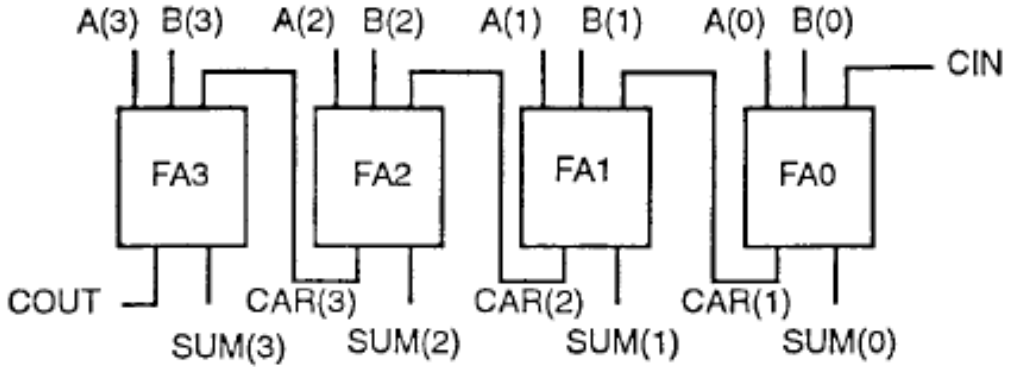

Fig. A 4-bit full-adder

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));
```

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, shown in below Fig., 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;
```

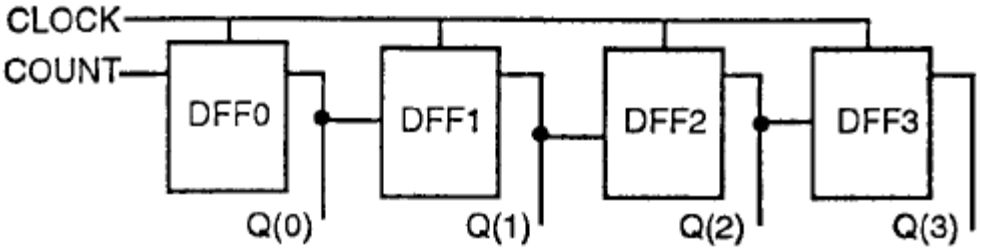

Fig. A 4-bit counter

File Types

Objects of file types represent files in the host environment. They provide a mechanism by which a VHDL design communicates with the host environment.

The syntax of a file type declaration is

type *file-type-name* **Is file of** *type-name;*

The *type-name* is the type of values contained in the file.

Few examples of file type declarations

type VECTORS **is file of** BIT_VECTOR;

A file of type VECTORS has a sequence of values of type BIT_VECTOR;

type NAMES **is file of** STRING;

is a file of type NAMES and has a sequence of strings as values in it .

A file is declared using a file declaration. The syntax of a file declaration is:

file *file-name: file-type-name* **is** *mode string-expression ;*

The *string-expression* is interpreted by the host environment as the physical name of the file i.e. file path name.The mode of a file, in or out, specifies whether it is an input or an output file, respectively. Input files can only be read while output files can only be written to. Two examples of declaring files are given below.

file VEC_FILE: VECTORS **is in** "/bt/uart/div.vec";

VEC_FILE is declared to be a file that contains a sequence of bit vectors and it is an input file. It is associated with the file "/bt/uart/div.vec" in the host environment.

file OUTPUT: NAMES **is out** "stdout";

A file belongs to the **variable** class of objects. However, a file cannot be assigned values using a variable assignment statement. It can be read, written to, or tested for an end-of-file condition, only by using special procedures and functions that are implicitly declared for every file type. These are as follows

procedure READ (F: **in** *file-type-name ;* VALUE: **out** *type-name) ;*
-- Gets the next value in VALUE from file F.

procedure WRITE (F: **out** *file-type-name ;* VALUE: **in** *type-name) ;*
-- Appends a given value in VALUE to file F.

function ENDFILE (F: **in** *file-type-name)* **return** BOOLEAN;
-- Returns false if a read on an input file F will be successful in getting another value, otherwise it returns true.

If *type-name* is an unconstrained array type, a different READ procedure is implicitly declared, which is of the form

procedure READ (F: **in** *file-type-name,* VALUE: **out** *type-name,* LENGTH: **out** NATURAL);

-- LENGTH returns the number of elements of the array that was read.
(**procedure** READ (F: **in** *file-type-name ;* VALUE: **out** *type-name) ;)*

A file cannot be opened or closed explicitly and values within a file can only be accessed sequentially. One of the predefined packages that is supplied with VHDL is the Textual Input and Output (TextIO) package. The TextIO package contains procedures and functions that give the designer the ability to read from and write to formatted text files. These text files are ASCII files of any format that the designer desires. TextIO treats these ASCII files as files of lines, where a line is a string, terminated by a carriage return. There are procedures to read a line and write a line and a function that checks for end of file. The TextIO package also declares a number of types that are used while processing text files. Type **line** is declared in the TextIO package and is used to hold a line to write to a file or a line that has just been read from the file. The line structure is the basic unit upon which all TextIO operations are performed. When reading from a file, the first step is to read in a line from the file into a structure of type **line**. Then the line structure is processed field by field.