VHDL

Introduction

As digital systems have become more complex, detailed design of the systems at the gate and flip-flop level has become very difficult and time consuming. This has been the reason for the extensive use of Hardware Descriptive Languages (HDLs). A HDL allows the designer/user to model, debug the top level of the digital circuit/system before conversion to the gate level. The two most popular HDLs are VHDL and Verilog.

Definition

VHDL describes the behavior and structure of digital systems. The acronym VHDL stands for VHSIC Hardware Descriptive Language and VHSIC in turn stands for Very High Speed Integrated Circuit. VHDL can describe a digital system at different levels of abstraction:

- Behavioral
- Data flow
- Structural

For instance, consider the case of a binary adder. The binary adder could be described at behavioral in terms of its actual function of adding two numbers without any additional implementation details. The same could be described at the data flow by giving logical equations for the adder (sum & carry). Finally, the adder could be described at the structural level by specifying the basic modules that are used to build the adder and also by specifying the interconnections between the same. The different levels will be dealt in detail with examples in later section.

VHDL adopts a top-down design methodology, in which a system is first specified at a high level and tested. Once debugged, the design can gradually be refined, eventually leading to a structural description closely related to the actual hardware implementation. VHDL is designed to be technology independent.

Benefits of VHDL

VHDL is defined by IEEE. This standard is known by all the VHDL tool developers. So there is only one language to learn. This language is used by all the circuit designers around the world. The life time for this language is assured, since it is an IEEE standards. Any investment or learning is assured for lifetime. Abundance of models available from different sources can be used with ease. Some tools might support Foreign Language Interface, by which you can add your model in C language to the VHDL code. It is a modern language, powerful and general. Other advantages include readability of the code and portability. The code developed is portable to any technology at any time. Time to market is short (leads to leadership in the market). Any error found during the simulation

phase is less expensive than by discovering the errors after making the circuit board (Investment is saved). The great advantage is that the Project Managers can modify the specification without leading to disaster (only the necessary portion of the code need to be changed). It can deliver designs 100% error free at short duration. New Concepts in hardware design (for example, in image processing, DSP, etc.,) can be modeled in VHDL and its efficiency or viability can be proven without doing the hardware.

A large number of ASICs fail to work when plugged into a system even if they meet their specifications first time. VHDL addresses this issue in two ways: A VHDL specification can be executed in order to achieve a high level of confidence in its correctness before commencing design, and may simulate one to two orders of magnitude faster than a gate level description. A VHDL specification for a part can form the basis for a simulation model to verify the operation of the part in the wider system context (e.g. printed circuit board simulation). This depends on how accurately the specification handles aspects such as timing and initialization.

VHDL Program

Entity – Architecture

A VHDL program consists of an entity declaration and an architectural declaration. All the input, output signals, type of each signal are mentioned in the entity whereas the behavior of the digital systems is described in the architecture in terms of logical equations or by mentioning the functional implementation or by any other means. Consider the example of a 1 bit full adder:

```
entity FullAdder is                          --declaration of entity
        port (a, b, cin : in bit;            --Inputs
              sum, cout : out bit);          --Outputs
end FullAdder;                               --End of entity
```

comments

Here:

- entity, port, is, end : key words
- in, out : modes
- FullAdder : identifier
- Bit : data type

The words entity, is, port, in, out, and, end are reserved words (keywords). They all have a special meaning to the VHDL compiler. Anything that follows “--“ is a VHDL comment. These will be not be considered during the compilation. The port declaration specifies that a, b, cin are input signals of type (data type) bit and sum, cout are output signals of type bit.

The operation of the full adder is specified by an architectural declaration:

```
architecture dataflow of  FullAdder is
begin
        sum <= a xor b xor cin;
```

```
    cout <= (a and b) or (b and cin) or (cin or a);
end dataflow;
```

Once again, the words in bold indicate the reserved words (key words). It should be noted that each statement in VHDL must end with a semicolon.

In this example, the architecture name (dataflow) is arbitrary, but the entity name (FullAdder) must match the name used in the associated entity declaration. Signal names and other VHDL identifiers may contain letters, numbers, and the underscore character. An identifier must start with a letter, and it cannot end with an underscore.

Data Types in VHDL

Like a high-level software programming language, VHDL allows data to be represented in terms of high-level data types. A data type is an abstract representation of stored data, such as you might encounter in software languages. These data types might represent individual wires in a circuit, or they might represent collections of wires.

For instance, the **bit** data type has only two possible values: ‘1’ or ‘0’. A **bit_vector** is simply an array of **bits**. Every data type in VHDL has a defined set of values, and a defined set of valid operations. Type checking is strict, so it is not possible, for example, to directly assign the value of an **integer** data type to a **bit_vector** data type. But, there are ways to get around this restriction, using what are called **type conversion functions** which are beyond the scope of the material.

The Table below summarizes the fundamental data types available in VHDL.

Data Type	***Values***	***Example***
Bit	‘1’, ‘0’	Q <= ‘1’;
bit_vector	(array of bits)	DataOut <= "00010101";
Boolean	True, False	EQ <= True;
Integer	-2, -1, 0, 1, 2, 3, 4 . . .	Count <= Count + 2;
Real	1.0, -1.0E5	V1 = V2 / 5.3
Time	1 us, 7 ns, 100 ps	Q <= ‘1’ after 6 ns;
Character	‘a’, ‘b’, ‘2, ‘$’, etc.	CharData <= ‘X’;
String	(Array of characters)	Msg <= "MEM: " & Addr

Operators

abs

An absolute value operator can be applied to any numeric type in an expression.
Example: Delta <= **abs**(A-B)

xnor

The logical "both or neither" (equality) operator can be used in an expression. The expression "A xnor B" returns True only when

- A is true and B is true
  OR
- A is false and B is false

and

The logical "and" operator can be used in an expression. The expression "A and B" returns **true** only if <u>both</u> A and B are true.

mod

The modulus operator can be applied to integer types. The result of the expression "A mod B" is an integer type and is defined to be the value such that:

- the sign of (A mod B) is the same as the sign of B, and
- abs (A mod B) < abs (B), and
- (A mod B) = (A * (B - N)) for some integer N.

nand

The logical "not and" operator can be used in an expression. It produces the opposite of the logical "and" operator. The expression "A nand B" returns True only when

- A is false
  OR
- B is false
  OR
- Both A and B are false

nor

The logical "not or" operator can be used in an expression. It produces the opposite of the logical "or" operator. The expression "A nor B" returns True only when both A and B are false.

not

The logical "not" operator can be used in an expression. The expression "not A" returns True if A is false and returns False if A is true.

or

The logical "or" operator can be used in an expression. The expression "A or B" returns True if

- A is true
  OR
- B is true
  OR

- Both A and B are true

rem

The remainder operator can be applied to integer types. The result of the expression "A rem B" is an integer type and is defined to be the value such that:

- The sign of (A rem B) is the same as the sign of A, and
- abs (A rem B) < abs (B), and
- (A rem B) = (A - (A / B) * B).

rol

Rotate left operator.
Example: Sreg <= Sreg **rol** 2;

ror

Rotate right operator.
Example: Sreg <= Sreg **ror** 2;

sla

Shift left arithmetic operator
Example: Addr <= Addr **sla** 8;

sll

Shift left logical operator.
Example: Addr <= Addr **sll** 8;

sra

Shift right arithmetic operator.
Example: Addr <= Addr **sra** 8;

srl

Shift right logical operator.
Example: Addr <= Addr **srl** 8;

xor

The logical "one or the other but not both" (inequality) operator can be used in an expression. The expression "A xor B" returns True only when

- A is true and B is false
  or
- A is false and B is true

: =

The equality operator can be used in an expression on any type except file types. The resulting type of an expression using this operator is Boolean (that is, True or False). The expression "A = B" returns True only if A and B are equal.

/=

The inequality operator can be used in an expression on any type except file types. The resulting type of an expression using this operator is Boolean (that is, True or False). The expression "A /= B" returns True only if A and B are not equal.

:=

This is the assignment operator for a variable. The expression "TEST_VAR := 1" means that the variable TEST_VAR is assigned the value 1.

<

The "less than" operator can be used in an expression on scalar types and discrete array types. The resulting type of an expression using this operator is Boolean (that is, True or False). The expression "A < B" returns True only if A is less than B.

<=

This symbol has two purposes. When used in an expression on scalar types and discrete array types, it is the "less than or equal to" operator. The resulting type of an expression using this operator in this context is Boolean (that is, True or False). In this context, the expression "A <= B" returns True only if A is less than or equal to B.
Example: LE := ‘1’ **when** A <= B **else** ‘0’;

In a signal assignment statement, the symbol "<=" is the assignment operator. Thus, the expression "TEST_SIGNAL <= 5" means that the signal TEST_SIGNAL is assigned the value 5.
Example: DataBUS <= 0x"E800";

>

The "greater than" operator can be used in an expression on scalar types and discrete array types. The resulting type of an expression using this operator is Boolean (that is, True or False). The expression "A > B" returns True only if A is greater than B.

>=

The "greater than or equal to" operator can be used in an expression on scalar types and discrete array types. The resulting type of an expression using this operator is Boolean (that is, True or False). The expression "A >= B" returns True only if A is greater than or equal to B.

+

This is the addition operator. Both operands must be numeric and of the same type. The result is also of the same numeric type. Thus, if A = 2 and B = 3, the result of the expression "A + B" is 5.
This operator may also be used as a unary operator representing the identity function. Thus, the expression "+A" would be equal to A.

-

This is the subtraction operator. Both operands must be numeric and of the same type. The result is also of the same numeric type. Thus, if A = 5 and B = 3, the result of the expression "A - B" is 2.
This operator may also be used as a unary operator representing the negative function. Thus, the expression "-A" would be equal to the negative of A.

&

This is the concatenation operator. Each operand must be either an element type or a 1-dimensional array type. The result is a 1-dimensional array type.

*****

This is the multiplication operator. Both operands must be of the same integer or floating point type.
The multiplication operator can also be used where one operand is of a physical type and the other is of an integer or real type. In these cases, the result is of a physical type.

/

This is the division operator. Both operands must be of the same integer or floating point type.
The division operator can also be used where a physical type is being divided by either an integer type or a real type. In these cases, the result is of a physical type. Also, a physical type can be divided by another physical type, in which case the result is an integer.

******

This is the exponentiation operator. The left operand must be of an integer type or a floating point type, and the right operand (the exponent) must be of an integer type. The result is of the same type as the left operand.

Data Objects

Signals

Signals are objects that are used to connect concurrent elements (such as components, processes and concurrent assignments), similar to the way that wires are used to connect components on a circuit board or in a schematic. Signals can be declared globally in an external package or locally within architecture, block or other declarative region.

To declare a signal, you write a **signal** statement such as the following:

```
architecture arch1 of my_design is
   signal Q: std_logic;
begin
   . . .
end arch1;
```

In this simple example, the signal **Q** is declared within the declaration section of the **arch1** architecture. At a minimum, a signal declaration must include the name of the signal (in this case **Q**) and its type (in this case the standard type **std_logic**). If more than one signal of the same type is required, multiple signal names can be specified in a single declaration:

```
architecture arch2 of my_design is
   signal Bus1, Bus2: std_logic_vector(7 downto 0);
begin
   . . .
end declare;
```

In the first example above, the declaration of **Q** was entered in the declaration area of architecture **arch1**. Thus, the signal **Q** will be visible anywhere within the **arch1** design unit, but it will not be visible within other design units. To make the signal **Q** visible to the entire design (a *global signal*), you would have to move the declaration into an external package, as shown below:

```
package my_package is
   signal Q: std_logic;     -- Global signal
end my_package;
. . .
use work.my_package.Q;   -- Make Q visible to the architecture
architecture arch1 of my_design is
begin
   . . .
end arch1;
```

In this example, the declaration for **Q** has been moved to an external package, and a **use** statement has been specified, making the contents of that package visible to the subsequent architecture.

Signal initialization

In addition to creating one or more signals and assigning them a type, the signal declaration can also be used to assign an initial value to the signal, as shown below:

```
signal BusA: std_logic_vector(4 downto 0) := “0000”;
```

Using Variables

Variables are objects used to store intermediate values between sequential VHDL statements. Variables are only allowed in processes, procedures and functions, and they are always local to those functions.

Variables in VHDL are much like variables in a conventional software programming language. They immediately take on and store the value assigned to them and they can be used to simplify a complex calculation or sequence of logical operations.

Using Constants and Literals

Constants

Constants are objects that are assigned a value once, when declared, and do not change their value during simulation. Constants are useful for creating more readable design descriptions, and they make it easier to change the design at a later time. The following code fragment provides a few examples of constant declarations:

```
architecture sample1 of consts is
   constant SRAM: bit_vector(15 downto 0) := X"F0F0";
   constant PORT: string  := "This is a string";
   constant error_flag: boolean := True;
begin
   . . .
   process( . . .)
      constant CountLimit: integer := 205;
   begin
      . . .
   end process;

end arch1;
```

Constant declarations can be located in any declaration area in your design description. If you want to create constants that are global to your design description, then you will place the constant declarations into external packages. If a constant will be used only within one segment of your design, you can place the constant declaration within the architecture, block, process or subprogram that requires it.

Literals

Explicit data values that are assigned to objects or used within expressions are called *literals*. Literals represent specific values, but they do not always have an explicit type. (For example, the literal **'1'** could represent either a **bit** data type or a **character**.) Literals do, however, fall into a few general categories.

Character literals

Character literals are 1-character ASCII values that are enclosed in single-quotes, such as the values **'1'**, **'Z'**, **'$'** and **':'**. The data type of the object being assigned one of these values (or the type implied by the expression in which the value is being used) will

dictate whether a given character literal is valid. The literal value **'$'**, for example, is a valid literal when assigned to a **character** type object, but it is not valid when assigned to a **std_logic** or **bit** data type.

String literals

String literals are collections of one or more ASCII characters enclosed in double-quote characters. String literals may contain any combination of ASCII characters, and they may be assigned to appropriately sized arrays of single-character data types (such as **bit_vector** or **std_logic_vector**) or to objects of the built-in type **string**.

Bit string literals

Bit string literals are special forms of string literals that are used to represent binary, octal, or hexadecimal numeric data values.

When representing a binary number, a bit string literal must be preceded by the special character 'B', and it may contain only the characters '0' and '1'. For example, to represent a decimal value of 36 using a binary format bit string literal, you would write **B"100100"**.

When representing an octal number, the bit string literal must include only the characters '0' through '7', and it must be preceded by the special character 'O', as in **O"446"**.

When representing a hexadecimal value, the bit string literal must be preceded by the special character 'X', and it may include only the characters '0' through '9' and the characters 'A' through 'F', as in **X"B295"**. (Lower-case characters are also allowed, so 'a' through 'f' are also valid.)

The underscore character **'_'** may also be used in bit string literals as needed to improve readability. The following are some examples of bit string literals representing a variety of numeric values:

B"0111_1101" (decimal value 253)
O"654" (decimal value 428)
O"146_231" (decimal value 52,377)
X"C300" (decimal value 49,920)

Note: In VHDL standard 1076-1987, bit string literals are only valid for the built-in type **bit_vector**. In 1076-193, bit string literals can be applied to any string type, including **std_logic_vector**.

Numeric literals

There are two basic forms of numeric literals in VHDL, *integer literals* and *real literals*.

Integer literals are entered as you would expect, as decimal numbers preceded by an optional negation character ('-'). The range of integers supported is dependent on your particular simulator or synthesis tool, but the VHDL standard does specify a minimum range of -2,147,483,647 to +2,147,483,647 (32 bits of precision, including the sign bit).

Real literals are entered using an extended form that requires a decimal point. For large numbers, scientific notation is also allowed using the character 'E', where the number to the left of the 'E' represents the mantissa of the real number, while the number to the right of the 'E' represents the exponent. The following are some examples of real literals:

5.0
-12.9
1.6E10
1.2E-20

The minimum and maximum values of real numbers are defined by the simulation tool vendor, but they must be at least in the range of -1.0E38 to +1.0E38 (as defined by the standard). Numeric literals may not include commas, but they may include underscore characters ("_") to improve readability, as in:

1_276_801 -- integer value 1,276,801

Type checking is strict in VHDL, and this includes the use of numeric literals. It is not possible, for example, to assign an integer literal of **9** to an object of type **real**. (You must instead enter the value as **9.0**.)

Based literals

Based literals are another form of integer or real values, but they are written in non-decimal form. To specify a based literal, you precede the literal with a base specification (such as 2, 8, or 16) and enclose the non-decimal value with a pair of '#' characters as shown in the examples below:

2#10010001# (integer value 145)
16#FFCC# (integer value 65,484)
2#101.0#E10 (real value 5,120.0)

Physical literals

Physical literals are special types of literals used to represent physical quantities such as time, voltage, current, distance, etc. Physical literals include both a numeric part (expressed as an integer) and a unit specification. Physical types will be described in more detail later in this chapter. The following examples show how physical literals can be expressed:

300 ns (300 nanoseconds)
900 ps (900 picoseconds)
40 ma (40 milliamps)

A unique feature of VHDL is that it can execute concurrent as well as sequential statements. This feature is explained in the following sections.

Concurrent Statements

Figure below describes a simple combinational network in VHDL.

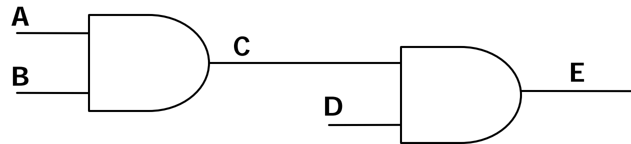

Figure Gate Network

This network can be modeled in the following way:

```
C <= A and B after 10 ns;          (1)
E <= C and D after 10 ns;          (2)
```

where 10 ns is propagation delay of each gate.

The symbol "<=" is the signal assignment operator, which means the values computed on the right side of the equations (1) & (2) is assigned on the left side. When simulated, C changes anytime A or B change their logic values and E changes any time C or D change their values. For instance, if A changes its value at time t = 0, then C changes at time t = 10 ns and E changes at time t = 20 ns.

VHDL signal assignment statements, like (1) & (2) are called concurrent statements when they are not contained in VHDL process or block. Unlike sequential statements the order of the preceding statements is of least significance.

Sequential Statements

A common way of modeling a sequential logic in VHDL is to use a *process* statement. The syntax of the process statement is as follow:

```
process (sensitivity list)
begin
        sequential statements
end process;
```

The sensitivity list includes the various signals that when changes its state will update the signals computed in sequential statements. This update of signals is done in a sequential manner, that is, the first statement is executed before executing the next statement as in any other programming language. A sample VHDL code that makes use of process statements is shown below:

```
process (A, B, C, D)
begin
        A <= B; -- sequential statement 1
        B <= C*2; -- sequential statement 2
```

```
        C <= D*3; -- sequential statement 3
end process;
```

Let the signals be initialized to A = 1, B = 2, C = 3 and D = 0. At time t = 10, the signal D is changed to 1, this will cause the sequential statements to be executed one after the other. The sequence of events is summarized as follows:

Time	Delta	A	B	C	D
0	+0	1	2	3	0
10	+0	1	2	3	1
10	+1	2	3	1	1
10	+2	3	1	1	1
10	+3	1	1	1	1

Compare the execution of the above sequential statements against the following concurrent statements:

```
        A <=B; -- concurrent statement 1
        B <=C*2; -- concurrent statement 2
        C <=D*3; -- concurrent statement 3
```

The sequence of events when the above assumptions are made as follows:

Time	Delta	A	B	C	D
0	+0	1	2	3	0
10	+0	1	2	3	1
10	+1	1	2	1	1
10	+2	1	1	1	1
10	+3	1	1	1	1

The following are the common sequential statements used in VHDL:

- if – then – end if
- case
- while – loop
- for – next
- exit
- null
- wait

IF

The syntax of the **if** statement has the form

```
        if condition then
                        Sequential statement 1
                else sequential statement 2
        end if;
```

The condition used here is a boolean expression to evaluate a TRUE or FALSE. If the boolean expression results in TRUE, the sequential statements following the then keyword (sequential statement 1) is executed. However, if the condition results in FALSE, the sequential statements following the else keyword (sequential statement 2) is executed. The end if statement indicates the end of the if condition. Consider the following VHDL code that describes the behavior of D Flip-Flop:

```
entity DFF is
port ( D, CLK : in bit;
        Q : out bit; QN : out bit := '1');
-- initialize QN to '1' since bit signals are initialized to '0' by default
end DFF;

architecture bhv of DFF is
begin
        process (CLK)                   -- process is executed when CLK changes
        begin
                if CLK = '1' then       -- rising edge of clock
                        Q <= D after 10 ns;
                        QN <= not D after 10 ns;
                end if;
        end process;
end bhv;
```

The above model of D Flip-Flop changes state on the rising edge of the clock input. The signal QN represents the Q' output of the flip-flop. In the entity declaration, signals D and CLK are declared as inputs to the D Flip-Flop, and signals Q and QN are declared as outputs to the D Flip-Flop. In the architecture 'bhv', a process statement is included with CLK signal in the sensitivity list. The statements within the body of the process are to be executed sequentially. The CLK signal is tested within the process and if CLK = '1' means the rising edge has occurred on CLK. Q is now set equal to D, and QN is set to the complement of D. The 10 ns delay represents the propagation delay between the time the clock changes and the flip-flop outputs change.

The **IF** statement can be implemented in another form. The syntax of this alternative way is given as follow:

```
if condition then
        sequential statements
{elsif condition then sequential statements }
        -- 0 or more elsif clauses may be included
{elsif sequential statements}
end if;
```

When

Conditional statement can be used with the concurrent statements using the **WHEN** keyword. The syntax of the WHEN statement has the form:

```
signal_name <= expression1 when condition1
               else expression2 when condition2
               ….
               {else expression};
```

This concurrent statement is executed whenever an event occurs on a signal used in one of the expressions or conditions. If condition1 is true, signal_name is set equal to the value of expression1, else if condition2 is true, signal_name is set equal to the value of expression2, etc. For example, consider the following VHDL code for 4-to-1 MUX:

```
    F <= I0 when Sel = 0
        else I1 when Sel = 1
        else I2 when Sel = 2
        else I3;
```

In the above concurrent statement, Sel is an integer equivalent to 2-bit binary number, and the output signal F is set to values I0, I1, I2 and I3 values when conditions associated with Sel signal becomes true.

Case

The case statement is used an alternative to multiple if statements within the process. The syntax of such statements has the following form:

```
case expression1 is
    when value1 => assignment statement 1;
    when value2 => assignment statement 2;
    when valuen => assignment statement n;
end case;
```

The same 4-to-1 MUX model is modeled using the case statement as follows:

```
process (Sel)
begin
    case Sel is
        when 0 => F <= I0;
        when 1 => F <= I1;
        when 2 => F <= I2;
        when others => F <= I3;
    end case;
end process;
```

In the above code, if Sel equals 0 then I0 is assigned to F, else if equals 1 then I1 is assigned to F, else if equals 2 then I2 is assigned to F. If however Sel does not equal any values I3 is assigned to F.

Null keyword

The null implies no action, which is appropriate, since the values of State should never occur. The following VHDL code shows the usage of null keyword:

```
process (Sel)
begin
        case Sel is
                when 0 => F <= I0;
                when 1 => F <= I1;
                when 2 => F <= I2;
                when 3 => F <= I3;
                when others => null;
        end case;
end process;
```

While-loop

The syntax of while loop statement is of the following form:

```
[loop-label:] while boolean-expression loop
                        Sequential statements
                end loop [loop-label];
```

An example of the while loop statement is shown below:

```
Label1: while C = '1' loop
                F <= A xor B; -- loop statements
        end loop Label1;
```

In the above VHDL code, a loop named Label1 exists as long as signal C equals 1. The sequential loop statements is executed each time the loop condition holds TRUE.

For – Loop

The general form of a for loop is given as follows:

```
[loop-label:] for loop-index in range loop
                Sequential statements
              end loop [loop-label];
```

This for loop is executes a set of sequential statements for a finite number of iterations defined in the range field. For example, consider the following example of 4-bit full adder

```
loop1: for i in 0 to 3 loop
```

```
        cout := (A(i) and B(i) or (A(i) and cin) or (B(i) and cin);
        sum(i) := A(i) xor B(i) xor cin;
        cin := cout;
end loop loop1;
```

Here A, and B are input bus signals, each 4-bit wide. In the above code, the ‘cout’, ‘sum’, and ‘cin’ signals are computed for each iteration of for –loop. The number of times the sequential statements are iterated is decided by the range specified in the **for** statement.

Exit

The exit statement has the syntax form as follows:

```
exit;    or      exit when condition;
```

This statement is generally included in a loop. The execution of this statement terminates the loop, and also terminates the loop in the second case when the condition is TRUE.

Wait

An alternative form of process uses wait statements instead of sensitivity list. The syntax of process with statement is shown below:

```
process
begin
        sequential statement1
        wait – statement 2
        sequential statement 3
        wait – statement 4
        ….
end process;
```

As noticed in the above syntax, a process cannot have both the wait and sensitivity list. In the above process, the sequential statement 1 is executed and waits for the statement 2. Once the condition statement 2 is met, the sequential statement 3 is executed and then waits for statement 4.

The wait statement can be of three different forms:

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

Example of VHDL Modeling of 2:1 Mux at Different Levels of Abstraction

The various architectures namely behavioral, dataflow, and structural can be best distinguished by considering the following example of a 2:1 multiplexer.

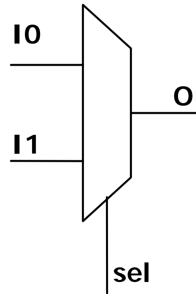

Figure 2:1 Multiplexer

The entity of 2:1 mux can be declared in the following way:

entity mux _21 **is**
port (I0, I1, sel : **in** bit;
        O: **out** bit);
end mux_21;

Behavioral

The behavioral architecture of 2:1 Mux can be declared in the following way:

architecture concurrentbehav of mux_21 **is**
begin
O <= I0 **when** sel = 0
else I1;
end behav;

This concurrent statement is executed whenever an event occurs on a signal used in one of the expressions or conditions. In the above concurrent statement, sel represents integer equivalent of the binary number.

The behavioral architecture can also be declared using sequential statements

architecture sequentialbehav of mux_21 **is**
begin
process (sel)

case sel **is**

when ‘0’ => O <= I0;
when ‘1’ => O <= I1;
when others => null;
end behav;

Dataflow:

In the following figure, 2:1 Mux is realized at the gate level. The data flow architecture as mentioned earlier can be declared using the logic equations that describe the system (2:1 Mux). The entity of the system will be the same as in the case of behavioral architecture. Only the architectural declaration will vary.

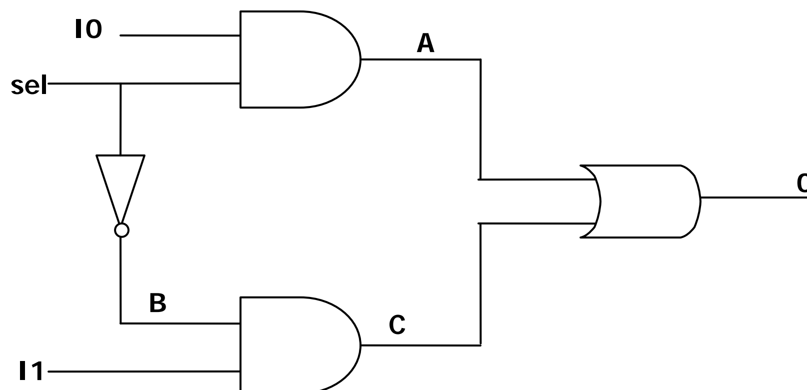

Figure Gate Level Implementation of 2:1 Multiplexer

architecture DF **of** mux_21 **is**
begin
O <= (I0 and sel) or (I1 and (not sel));
end DF;

Structural

The same, 2:1 Mux can also be described at the structural level. Again, considering the gate level implementation of the Mux the structural architecture can be declared in the following way:

architecture structural **of** mux_21 **is**

component and2
port (x, y : in bit; z : out bit);
end component;

```
component Inv
port (x : in bit; y : out bit);
end component;

component or2
port (x, y: in bit; z: out bit);
end component;

signal A, B : bit;

begin

A0 : and2 port map (I0, sel, A);
A1 : and2 port map (I1, C, B);

I0: Inv port map (sel, C);
0o: or2 port map (A, C, O);

end structure;
```

It should be carefully noted that before using the components and2, or2, and Inv in the above structural description these components should already be declared and compiled. The component name should match exactly with the entity name of the compone t being used.

For instance, and gate can be declared in the following way:

```
entity and2 is
port (x, y : in bit; z : out bit);
end and2;

architecture and_df of and2 is
begin
z <= x and y;
end and_df;
```

In the structural architecture, the and gate, inverter, and or gate specified as a component within the architecture. The component specification is similar to the entity declaration for the and, inverter, and or gates. The input and output port signals correspond to those declared for the gates. Then we declare the internal signals after the component declaration.

In the body of the architecture, we create instances of the and gate, inverter, and or gate. Each instance has a name (A0, A1, I0, O0) and a port map. The signal names following the port map correspond one-to-one with the signals in the component port. Thus, for the instance A0, I0, sel, and A correspond to x, y, and z in the actual entity/component of AND gate.