SECX1023 PROGRAMMING IN HDL

UNIT I INTRODUCTION TO VHDL

Digital system design process – Levels of Abstraction – Language elements of VHDL- Operators-Data Types – Signal assignments – Inertial delay mechanism – Transport delay mechanism – Concurrent and Sequential assignments – Delta delay

VHDL

VHDL is an acronym for VHSIC Hardware Description Language (VHSIC is an acronym for Very High Speed Integrated Circuits). It is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete digital electronic system, or anything in between. The digital system can also be described hierarchically. Timing can also be explicitly modeled in the same description.
The VHDL language can be regarded as an integrated amalgamation of the following languages:
sequential language
Concurrent language
net-list language
timing specifications
Waveform generation language.

Therefore, the language has constructs that enable you to express the concurrent or sequential behavior of a digital system with or without timing. It also allows you to model the system as an interconnection of components. Test waveforms can also be generated using the same constructs. All the above constructs may be combined to provide a comprehensive description of the system in a single model.

The language not only defines the syntax but also defines very clear simulation semantics for each language construct. Therefore, models written in this language can be verified using a VHDL simulator. It is a strongly typed language and is often verbose to write. It inherits many of its features, especially the sequential language part, from the Ada programming language. Because VHDL provides an extensive range of modeling capabilities, it is often difficult to understand. Fortunately, it is possible to quickly assimilate a core subset of the language that is both easy and simple to understand without learning the more complex features. This subset is usually sufficient to model most applications. The complete language, however, has sufficient power to capture the descriptions of the most complex chips to a complete electronic system.

Digital system design process :-

Digital Systems have conquered the whole world. Every appliances or equipment's we see today are digital. This is because of the very small element called Transistor invented by John Bardeen, Walter Brattain & William Shockley in 1947 at Bell Labs. This tiny and Powerful transistor changed the future of Electronics. Therefore it is our responsibility to study the analysis and design of this digital system as an electronic student. In this chapter we will study the Basic Digital IC Design Flow and then we will study what are the tools available for digital design and synthesis. Later we are going to study a special hardware description language (VHDL) which is used to describe the digital systems.

Digital Design Flow Process:-

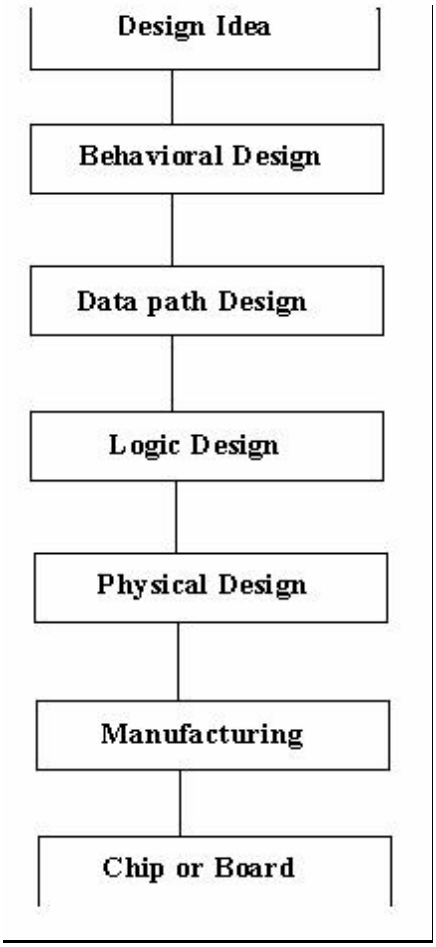

Fig 1.1: Generic IC design flow

Based on the specification given, the design team forms a general idea about the solution to the problem. System level decisions are made regarding the design and a general consensus is reached regarding the major functional blocks that go into the making of the chip. At the end of this stage, a general block diagram solution of the design is agreed upon. CAD tools are generally not needed at this stage.

Behavioral Design:

Hardware Description Languages (HDLs) are used to model the design idea (block diagram). Circuit details and electrical components are not specified. Instead, the behavior of each block at the highest level of abstraction is modeled. Simulations are then run to see if the blocks do indeed function as expected and the whole system performs as a whole. Behavioral descriptions are important as they corroborate the integrity of the design idea. Here we don't have any architectural or hardware details.

Data Path Design:

The next Phase in the design process is the design of the system data path. In this phase, the designer specifies the registers and logic units necessary for implementation of the system. These components may be interconnected using either bidirectional or unidirectional buses. Based on the intended behavior of the system, the procedure of controlling the movement of data between registers and logic units through buses are developed. Data components in the data part of circuit communicate via system busses and the control procedure controls flow of data between these components. This phase results in architectural design of the system with specification of control flow.

Logic Design:

Logic Design is the next phase in the design process and involves the use of primitive gates and flip-flops for the implementation of data registers, busses, logic units, and their controlling hardware. The result of this design stage is a net list of gates and flip-flops. Components used and their interconnections are specified in this net list.

Physical Design:

This stage transforms the net list into transistor list or layout. This involves the replacement of gates and flip-flops with their transistor equivalents or library cells.

Manufacturing:

The final step is manufacturing, which uses the transistor list or layout specification to burn fuses of FPGA or to generate masks for Integrated circuit (IC).

Levels of Abstraction:-

Hardware Abstraction:

VHDL is used to describe a model for a digital hardware device. This model specifies the external view of the device and one or more internal views. The internal view of the device specifies the functionality or structure, while the external view specifies the interface of the device through which it communicates with the other models in its environment. Fig1.2 shows the hardware device and the corresponding software model.

The device to device model mapping is strictly a one to many. That is, a hardware device may have many device models. For example, a device modeled at a high level of abstraction may not have a clock as one of its inputs, since the clock may not have been used in the description. Also the data transfer at the interface may be treated in terms of say, integer values, instead of logical values. In VHDL, each device model is treated as a distinct representation of a unique device, called an entity in this text. Fig1.2 shows the VHDL view of a hardware device that has multiple device models, with each device model representing one entity. Even though entity I through N represent N different entities from the VHDL point of view, in reality they represent the same hardware device.

The entity is thus a hardware abstraction of the actual hardware device. Each entity is described using one model that contains one external view and one or more internal views. At the same time, a hardware device may be represented by one or more entities.

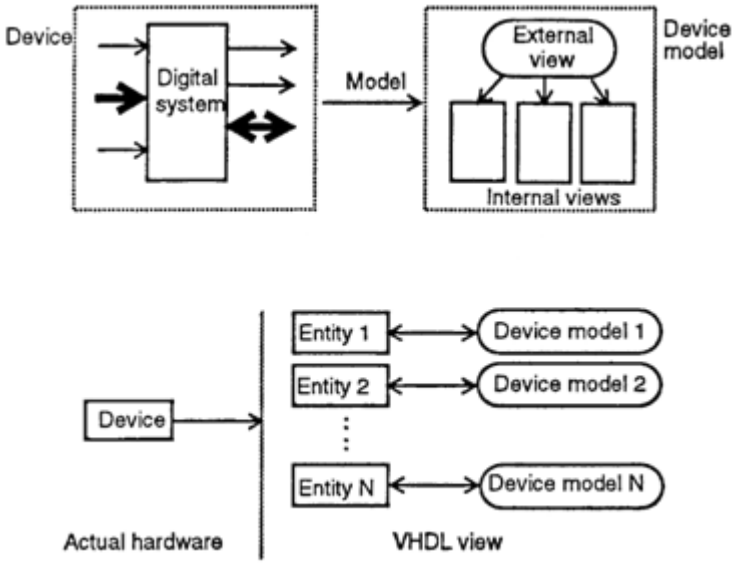

Fig 1.2 A VHDL view of a device

Basic Terminology

VHDL is a hardware description language that can be used to model a digital system. The digital system can be as simple as a logic gate or as complex as a complete electronic system. A hardware abstraction of this digital system is called an entity in this text. An entity X, when used in another entity Y, becomes a component for the entity Y. Therefore, a component is also an entity, depending on the level at which you are trying to model.

To describe an entity, VHDL provides five different types of primary constructs, called design units.

They are

1.Entity declaration
2.Architecture body
3.Configuration declaration
4.Package declaration
5.Package body

An entity is modeled using an entity declaration and at least one architecture body. The entity declaration describes the external view of the entity, for example, the input and output signal names. The architecture body contains the internal description of the entity, for example, as a set of interconnected components that represents the structure of the entity, or as a set of concurrent or sequential statements that represents the behavior of the entity. Each style of representation can be specified in a different architecture body or mixed within a single architecture body Fig1.3 shows an entity and its model.

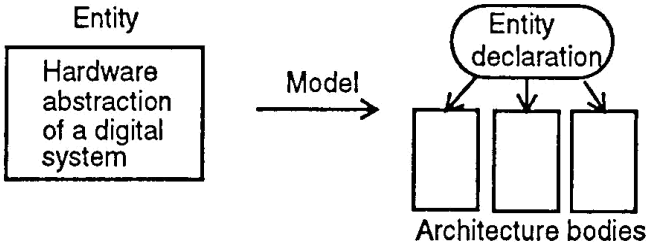

Fig 1.3 An entity and its model.

A configuration declaration is used to create a configuration for an entity. It specifies the binding of one architecture body from the many architecture bodies that may be associated with the entity. It may also specify the bindings of components

used in the selected architecture body to other entities. An entity may have any number of different configurations.

A package declaration encapsulates a set of related declarations such as type declarations, subtype declarations, and subprogram declarations that can be shared across two or more design units. A package body contains the definitions of subprograms declared in a package declaration.

Fig1.4 shows three entities called El, E2, and E3. Entity El has three architecture bodies, EI_AI, EI_A2, and EI_A3. Architecture body EI _AI is a purely behavioral model without any hierarchy. Architecture body EI_A2 uses a component called BX, while architecture body EI_ A3 uses a component called CX. Entity E2 has two architecture bodies, E2_ AI and E2_A2, and architecture body E2_AI uses a component called MI. Entity E3 has three architecture bodies, E3_ AI, E3_A2, and E3_A3. Notice that each entity has a single entity declaration but more than one architecture body.

The dashed lines represent the binding that may be specified in a configuration for entity El. There are two types of binding shown: binding of an architecture body to its entity and the binding of components used in the architecture body with other entities. For example, architecture body, EI_A3, is bound to entity El, while architecture body, E2_AI, is bound to entity E2. Component MI in architecture body, E2_AI, is bound to entity E3. Component CX in the architecture body, EI _A3, is bound to entity E2. However, one may choose a different configuration for entity El with the following bindings:

- Architecture EI_A2 is bound to its entity El
- Component BX to entity E3
- Architecture E3_AI is bound to its entity E3

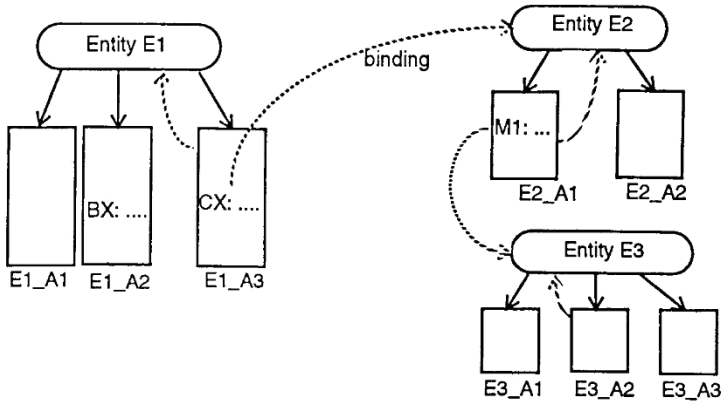

Fig 1.4 A configuration for entity El.

Once an entity has been modeled, it needs to be validated by a VHDL system. A typical VHDL system consists of an analyzer and a simulator. The analyzer reads in one or more design units contained in a single file and compiles them into a design library after validating the syntax and performing some static semantic checks. The design library is a place in the host environment (that is, the environment that supports the VHDL system) where compiled design units are stored.

The simulator simulates an entity, represented by an entity-architecture pair or by a configuration, by reading in its compiled description from the design library and then performing the following steps:

1.Elaboration
2.Initialization
3.Simulation

A note on the language syntax. The language is case insensitive, that is, lower-case and upper-case characters are treated alike. For example, CARRY, CarrY, or

CarrY, all refer to the same name. The language is also free -format, very much like in Ada and Pascal programming languages. Comments are specified in the language by preceding the text with two consecutive dashes (-). All text between the two dashes and the end of that line is treated as a comment.

The terms introduced in this section are described in greater detail in the following sections.

Entity Declaration

The entity' declaration specifies the name of the entity being modeled and lists the set of interface ports. Ports are signals through which the entity communicates with the other models in its external environment.

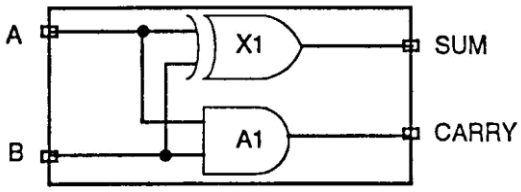

Fig 1.5 A half-adder circuit.

Here is an example of an entity declaration for the half-adder circuit shown in Fig. 1.5.

```
entity HALF_ADDER is
port (A, B: in BIT; SUM, CARRY: out BIT);
end HALF_ADDER;
```

The entity, called HALF_ADDER, has two input ports, A and B (the mode in specifies input port), and two output ports, SUM and CARRY (the mode out specifies output port). BIT is a predefined type of the language; it is an enumeration type containing the character literals '0' and '1'. The port types for this entity have been specified to be of type BIT, which means that the ports can take the values, '0' or '1'.

The following is another example of an entity declaration for a 2-to-4 decoder circuit shown in Fig. 1.6.

```
entity DECODER2x4 is
        port(A, B, ENABLE: in SIT: Z: out
    BIT_VECTOR(0 to 3)); end DECODER2x4;
```

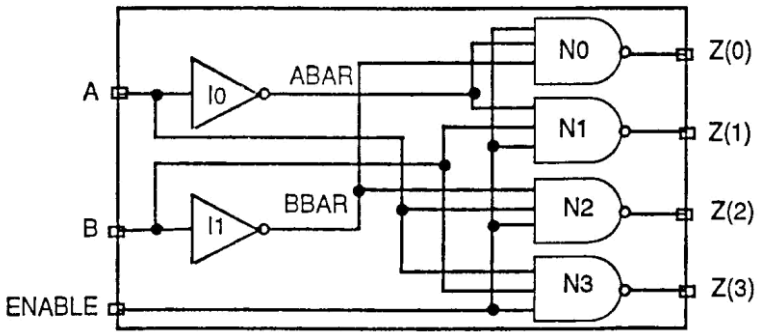

Fig 1.6 :A 2-to-4 decoder circuit.

This entity, called DECODER2x4, has three input ports and four output ports. BIT_VECTOR is a predefined unconstrained array type of BIT. An unconstrained array type is a type in which the size of the array is not specified. The range "0 to 3" for port Z specifies the array size.

From the last two examples of entity declarations, we see that the entity declaration does not specify anything about the internals of the entity. It only specifies the name of the entity and the interface ports.

Architecture Body

The internal details of an entity are specified by an architecture body using any of the following modeling styles:

1. As a set of interconnected components (to represent structure),
2. As a set of concurrent assignment statements (to represent dataflow),
3. As a set of sequential assignment statements (to represent be-hav.ior),
4. Any combination of the above three.

Structural Style of Modeling

In the structural style of modeling, an entity is described as a set of interconnected components. Such a model for the HALF_ADDER entity, shown in Fig. 1.5, is described in an architecture body as shown below.

```
architecture HA_STRUCTURE of
    HALF_ADDER 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 (A, B,
        SUM); A1: AND2 port map
        (A, B, CARRY);
    end HA_STRUCTURE;
```

The name of the architecture body is HA_STRUCTURE. The entity declaration for HALF_ADDER (presented in the previous section) specifies the interface ports for this architecture body. The architecture body is composed of two parts: the declarative part (before the keyword begin) and the statement part (after the keyword begin). Two component declarations are present in the declarative part of the architecture body. These declarations specify the interface of components that are used in the architecture body. The components XOR2 and AND2 may either be predefined components in a library, or if they do not exist, they may later be bound to other components in a library.

The declared components are instantiated in the statement part of the architecture body using component instantiation statements. XI and A1 are the component labels for these component instantiations. The first component instantiation statement, labeled XI, shows that signals A and B (the input ports of the HALF_ADDER), are connected to the X and Y input ports of a XOR2 component, while output port Z of this component is connected to output port SUM of the HALF_ADDER entity.

Similarly, in the second component instantiation statement, signals A and B are connected to ports L and M of the AND2 component, while port N is connected to the CARRY port of the HALF_ADDER. Note that in this case, the signals in the port map of a component instantiation and the port signals in the component declaration are associated by position (called positional association). The structural representation for the HALF_ADDER does not say anything about its functionality. Separate entity

models would be described for the components XOR2 and AND2, each having its own entity declaration and architecture body.

A structural representation for the DECODER2x4 entity, shown in Fig. 1.6, is shown next.

```
architecture DEC_STR of
        DECODER2x4 is component
        INV
                port (A: in BIT; Z: out
        BIT); end component;
        component NAND3
                port (A, B, C: in BIT; Z:
        out BIT); end component;
        signal ABAR, BBAR: BIT;
begin
        I0: INV port map
        (A,ABAR); I1: INV
        port map (B, BBAR);
        N0: NAND3 port map (ABAR, BBAR,
        ENABLE, Z(0)); N1: NAND3 port map
        (ABAR, B, ENABLE, Z(1)); N2: NAND3
        port map (A, BBAR, ENABLE, Z(2)); N3:
        NAND3 port map (A, B, ENABLE, Z(3));
end DEC_STR;
```

In this example, the name of the architecture body is DEC_ STR, and it is associated with the entity declaration with the name DECODER2x4; therefore, it inherits the list of interface ports from that entity declaration. In addition to the two component declarations (for INV and NAND3), the architecture body contains a signaldeclaration that declares two signals, ABAR and BBAR, of type BIT. These signals, that represent wires, are used to connect the various components that form the decoder. The scope of these signals is restricted to the architecture body, and therefore, these signals are not visible outside the architecture body. Contrast these signals with the ports of an entity declaration that are available for use within any architecture body associated with the entity declaration.

A component instantiation statement is a concurrent statement, as defined by the language. Therefore, the order of these statements is not important. The structural style of modeling describes only an interconnection of components (viewed as black boxes) without implying any behavior of the components themselves, nor of the entity that they collectively represent. In the architecture body DEC_STR, the signals A, B, and ENABLE, used in the component instantiation statements are the input ports declared in the DECODER2x4 entity declaration. For example, in the component instantiation labeled N3, port A is connected to input A of component NAND3, port B is connected to input port B of component NAND3, port ENABLE is connected to input port C, and the output port Z of component NAND3 is connected to port Z(3) of the DECODER2x4 entity. Again positional association is used to map signals in a port map of a component instantiation with the ports of a component specified in its declaration. The behavior of the components NAND3 and INV are not apparent, nor is the behavior of the decoder entity that the structural model represents.

Configuration Declaration

A configuration declaration is used to select one of the possibly many architecture bodies that an entity may have, and to bind components, used to represent structure in that architecture body, to entities represented by an entity-architecture pair or by a configuration, that reside in a design library. Consider the following configuration declaration for the HALF_ADDER entity.

```
library CMOS_LIB, MY_LIB;
configuration HA_BINDING of
    HALF_ADDER is for HA-
    STRUCTURE
        for X1:XOR2
            use entity
            CMOS_LIB.XOR_GATE(DATAFLOW);
        end for;
        for A1:AND2
            use configuration MY_LIB.AND_CONFIG;
        end for;

        end for; end HA_BINDING;
```

The first statement is a library context clause that makes the library names CMOS_LIB and MY_LIB visible within the configuration declaration. The name of the configuration is HA _BINDING, and it specifies a configuration for the HALF_ADDER entity. The next statement specifies that the architecture body HA_STRUCTURE (described in Sec. 23.1) is selected for this configuration. Since this architecture body contains two component instantiations, two component bindings are required. The first statement (for XI: . . . end for) binds the component instantiation, with label XI, to an entity represented by the entity-architecture pair, XOR_GATE.

The architecture body consists of one signal declaration and six concurrent signal assignment statements. The signal declaration declares signals ABAR and BBAR to be used locally within the architecture body. In each of the signal assignment statements, no after clause was used to specify delay. In all such cases, a default delay of 0ns is assumed. This delay of 0ns is also known as delta delay, and it represents an infinitesimally small delay. This small delay corresponds to a zero delay with respect to simulation time and does not correspond to any real simulation time.

To understand the behavior of this architecture body, consider an event happening on one of the input signals, say input port B at time T. This would cause the concurrent signal assignment statements 1,3, and 6, to be triggered. Their right -hand-side expressions would be evaluated and the corresponding values would be scheduled to be assigned to the target signals at time (T+A). When simulation time advances to (T+A), new values to signals Z(3), BBAR, and Z(1), are assigned. Since the value of BBAR changes, this will in turn trigger signal assignment statements, 2 and 4. Eventually, at time (T+2A), signals Z(0) and Z(2) will be assigned their new values.

The semantics of this concurrent behavior indicate that the simulation, as defined by the language, is event-triggered and that simulation time advances to the next time unit when an event is scheduled to occur. Simulation time could also advance a multiple of delta time units. For example, events may have been scheduled to occur at times 1,3,4,4+A, 5,6,6+A, 6+2A, 6+3A, 10,10+A, 15, 15+A time units.

Entity declaration and the DATAFLOW architecture body, that resides in the CMOS_LIB design library. Similarly, component instantiation Al is bound to a configuration of an entity defined by the configuration declaration, with name AND_CONFIG, residing in the MY_LIB design library.

There are no behavioral or simulation semantics associated with a configuration declaration. It merely specifies a binding that is used to build a configuration for an entity. These bindings are performed during the elaboration phase of simulation when the entire design to be simulated is being assembled. Having defined a configuration for the entity, the configuration can then be simulated.

When an architecture body does not contain any component instantiations, for example, when dataflow style is used, such an architecture body can also be selected to create a configuration. For example, the DEC_DATAFLOW architecture body can be selected for the DECODER2x4 entity using the following configuration declaration.

```
configuration DEC_CONFIG of
DECODER2x4 is for
DEC_DATAFLOW
end for;
end DEC_CONFIG ;
```

DEC_CONFIG defines a configuration that selects the DEC_DATAFLOW architecture body for the DECODER2x4 entity. The configuration DEC_CONFIG, that represents one possible configuration for theDECODER2x4 entity, can now be simulated.

Package Declaration

A package declaration is used to store a set of common declarations like components, types, procedures, and functions. These declarations can then be imported into other design units using a context clause. Here is an example of a package declaration.

```
package EXAMPLE_PACK is
        type SUMMER is (MAY, JUN, JUL,
        AUG, SEP); component
        D_FLIP_FLOP
                port (D, CK: in BIT; Q, QBAR:
        out BIT); end component;
        constant PIN2PIN_DELAY: TIME :=
        125 ns; function INT2BIT_VEC
        (INT_VALUE: INTEGER)
                return
BIT_VECTOR;
end EXAMPLE_PACK;
```

The name of the package declared is EXAMPLE_PACK. It contains type, component, constant, and function declarations. Notice that the behavior of the function INT2BIT _VEC does not appear in the package declaration; only the function interface appears. The definition or body of the function appears in a package body (see next section).

Assume that this package has been compiled into a design library called DESIGN_LIB. Consider the following context clauses associated with an entity declaration.

```
library DESIGN_LIB;
useDESIGN_LIB.EXAMPLE_P
```

```
ACK.all; entity RX is . . .
```

The library context clause makes the name of the design library DESIGN_LIB visible within this description, that is, the name DESIGN_LIB can be used within the description. This is followed by a use context clause that imports all declarations in package EXAMPLE_PACK into the entity declaration of RX.

It is also possible to selectively import declarations from a package declaration into other design units. For example,

```
library DESIGN_LIB;
use
DES[GN_LIB.EXAMPLE_PACK.D_FLIP_
FLOP;                                   use
DESIGN_LIB.EXAMPLE_PACK.PIN2PIN
_DELAY; architecture RX_STRUCTURE of
RX is . . .
```

The two use context clauses make the component declaration for D_FLIP_FLOP and the constant declaration for PIN2PIN_DELAY, visible within the architecture body.
Another approach to selectively import items declared in a package is by using selected names. For example,

```
library DESIGN_LIB;
package
        ANOTHER_PACKAG
        E       is      function
        POCKET_MONEY
                (MONTH:DESIGN_LIB.EXAMPLE_PAC
                K.SUMMER) return INTEGER;
        constant TOTAL_ALU: INTEGER;  -- A deferred constant.
end ANOTHER_PACKAGE;
```

The type SUMMER declared in package EXAMPLE_PACK is used in this new package by specifying a selected name. In this case, a use context clause was not necessary. Package ANOTHER_PACKAGE also contains a constant declaration with the value of the constant not specified; such a constant is referred to as a deferred constant. The value of this constant is supplied in a corresponding package body.

Package Body

A package body is primarily used to store the definitions of functions and procedures that were declared in the corresponding package declaration, and also the complete constant declarations for any deferred constants that appear in the package declaration. Therefore, a package body is always associated with a package declaration; furthermore, a package declaration can have at most one package body associated with it. Contrast this with an architecture body and an entity declaration where multiple architecture bodies may be associated with a single entity declaration. A package body may contain other declarations as well.
Here is the package body for the package EXAMPLE_PACK declared in the previous section.

```
package body EXAMPLE_PACK is
        function INT2BIT_VEC (INT_VALUE:
                INTEGER)                return
```

```
                BIT_VECTOR is
        begin
    end INT2BIT_VEC;
end EXAMPLE_PACK;
```

The name of the package body must be the same as that of the package declaration with which it is associated. It is important to note that a package body is not necessary if the corresponding package declaration has no function and procedure declarations and no deferred constant declarations. Here is the package body that is associated with the package ANOTHER_PACKAGE that was declared in the previous section.

```
package body ANOTHER_PACKAGE is
    constant TOTAL_ALU: INTEGER := 10;
    function POCKET_MONEY
        (MONTH:
        DESIGN_UB.EXAMPLE_PACK.SUMMER)
        return INTEGER is
    begin
    case MONTH is
    when MAY => return 5;
    when JUL I SEP => return 6;
    when others => return 2;
    end case; end POCKET_MONEY;
    end ANOTHER_PAC
```

Data Operators

VHDL will support different types of operations. The following are the types of operators available in VHDL

1. Assignment operator
2. Logical Operator
3. Relational Operator
4. Shift operator
5. Arithmetic operator

    5.1 Addition Operator
    5.2 Multiplication Operator
    5.3 Miscellaneous operator

Assignment Operator

This operator is used to assign values to signals, variables, and constants. They are

1. <= Used to assign a value to signal
2. := Used to assign a variable, constant or generic, used for also establishing initial values.
3. => Used to assign values to individual vector or with others.

Logical Operators

Used to perform to logical operations. The data must be of type Bit, Std_logic or std_ulogic. The logical operators are:

1. NOT
2. AND
3. OR
4. NAND
5. NOR , XOR & XNOR

Relational Operators

Used for making comparisons. The data can be of any types listed above. The relational (Comparison) operators listed below:

1. = Equal to
2. /= not equal to
3. < Greater than
4. > Lesser than
5. <= Greater than
6. >= Lesser than

Shift Operators

Used for shifting data.

1. Sll: Shift left logic
2. Sla: shift left arithmetic
3. Srl: Shift right logic
4. Sra: Shift right arithmetic
5. Rol:Rotateleft
6. Ror: Rotate right

Arithmetic Operators

Used to perform arithmetic operations. The data can be of integer, signed, Unsigned or a real.

The different types of arithmetic operations are:

1. Addition operator (+)
2. Subtract Operator (-)
3. Multiplication operator (*)
4. Division Operator (/)
5. Modulus (MOD)
6. Remainder (REM)

Miscellaneous Operator

Uses as special cases in VHDL

1. Absolute (ABS):
2. Exponentiation (**)

DATA TYPES

All of the objects that are discussed in previous section—the signal, the Variable, and the constant—can be declared using a type specification to specify the characteristics of the object. VHDL contains a wide range of types that can be used to create simple or complex objects. To define a new type, you must create a type declaration. A type declaration defines the name of the type and the range of the type. Type declarations are allowed in package declaration sections, entity declaration sections, architecture declaration sections, subprogram declaration sections, and process declaration sections.

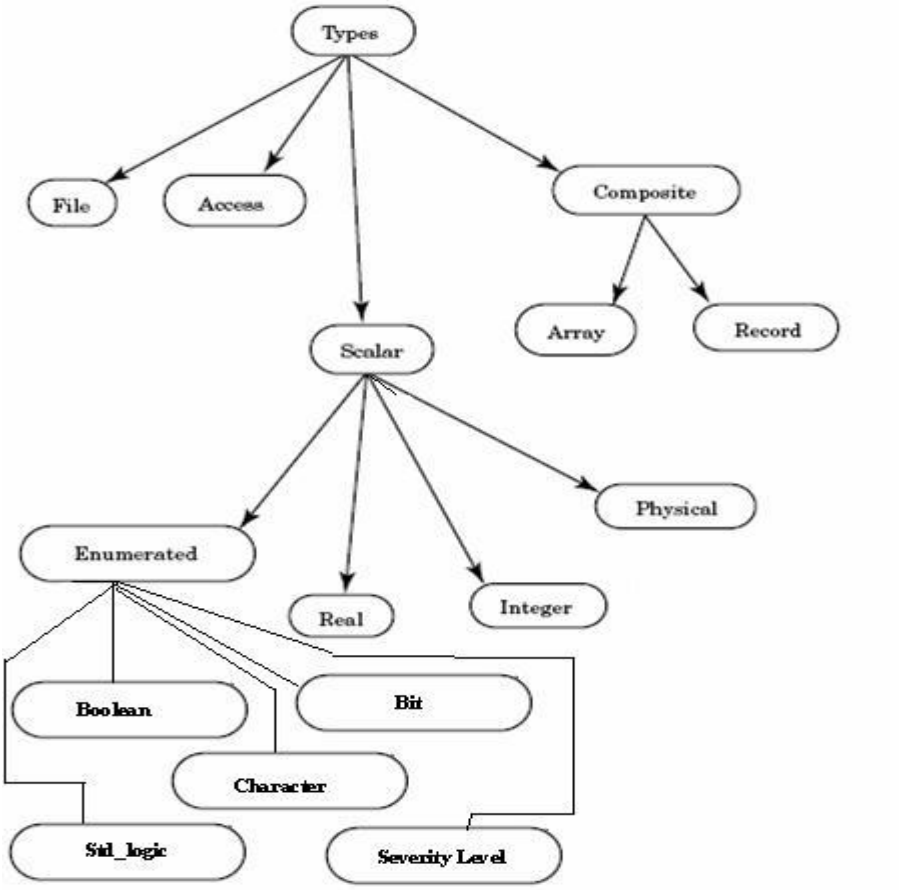

Fig 1.7: Data Types in VHDL

Scalar Types

Scalar types describe objects that can hold, at most, one value at a time. The type itself can contain multiple values, but an object that is declared to be a scalar type can hold, at most, one of the scalar values at any point in time. Referencing the name of the object references the entire object. Scalar types encompass these four classes of types.

1. Integer types
2. Real types
3. Enumerated types
4. Physical types
5. Floating Point

Enumerated Data Types

An enumerated type is a very powerful tool for abstract modeling. A designer can use an enumerated type to represent exactly the values required for a specific operation. All of the values of an enumerated type are user-defined. These values can be identifiers or single-character literals. An identifier is like a name.
These are further classified as the following:

1. Boolean
2. Character
3. Bit
4. Std_logic
5. Severity Level

Boolean

This data type is used when we need to convey some *true or false* conditions. For example

```
Architecture ……..
Begin
Process
(….)
Variable        temp
:boolean Begin
if a < b then

temp <= True;
Else
temp <= False;
end if;
end process;
```

Character

This daa type is used when we need to use all alpha numeric and special characters.

Bit

This data type is used when we need to represent binary values (‘0’ and ‘1’)

Severity Level

This data type is used in Complex projects where we need to show warnings, errors in runtime,
Failures in runtime.

Std_ulogic;

This data types are declared in std_logic_1164.all package of IEEE Library

U →Uninitialized
X →Forcing unknown
Z →High Impedence
W →Weak unknown
‘-‘→don’t care
0 →Forcing 0
1 →Forcing 1
L → Weak 0
H →Weak 1

A typical enumerated type for a four-state simulation value system looks like this:

Type fourval is (‘x’, ‘0’, ‘1’, ‘z’);

Character literals are needed for values ‘1’ and ‘0’ to separate these values from the integer values 1 and 0. It would be an error to use the values 1 and 0 in an enumerated type, because these are integer values. The characters X and Z do not need quotes around them because they do not represent any other type, but the quotes were used for uniformity.

Integer Data type

Integers are exactly like mathematical integers. All of the normal predefined mathematical functions like add, subtract, multiply, and divide apply to integer types. The VHDL LRM does not specify a maximum range for integers, but does specify the minimum range: from -2,147,483,647 to 12,147,483,647. The minimum range is specified by the Standard package contained in the Standard Library. The Standard package defines all of the predefined VHDL

types provided with the language. The Standard Library is used to hold any packages

or entities provided as standard with the language.

There are two types of declaration for Integer Data type

1. Type_integer declaration
   Ex: *type <word lengt> is range 0 to 31;*
2. Object_integer declaration
   Ex: *constant <loop number>: <integer><=345;*

Real Data Type

Real types are used to declare objects that emulate mathematical real numbers. They can be used to represent numbers out of the range of integer values as well as fractional values. The minimum range of real numbers is also specified by the Standard package in the Standard library, and is from _1.0E_38 to _1.0E_38.

Following are a few examples of some real numbers:

```
Architecture test of test is
Signal a: real;
begin
a <= 1.0;         --ok 1
a <= 1;           --error 2
a <= -1.0e10;   --ok 3
a <= 1.5e-20;   --ok 4
a <= 5.3 ns;    --error 5
End test;
```

Line 1 shows how to assign a real number to a signal of type **REAL**. All real numbers have a decimal point to distinguish them from integer values. Line 2 is an example of an assignment that does not work. Signal **a** is of type **REAL**, and a real value must be assigned to signal **a**. The value 1 is of type **INTEGER**, so a type mismatch is generated by this line. Line 3 shows a very large negative number. The numeric characters to the left of the character **E** represent the mantissa of the real number, while the numeric value to the right represents the exponent. Line 4 shows how to create a very small number. In this example, them exponent is negative so the number is very small. Line 5 shows how a type **TIME** cannot be assigned to a real signal. Even though the numeric part of the value looks like a real number, because of the units after the value, the value is considered to be of type **TIME**.

Physical Data types

Physical types are used to represent physical quantities such as distance, current, time, and so on. A physical type provides for a base unit, and successive units are then defined in terms of this unit. The smallest unit represent able is one base unit; the largest is determined by the range specified in the physical type declaration. An example of a physical type for the physical quantity current is shown here:

```
Type current is range 0 to 1000000000
Units
na; --nano amps
ua = 1000 na; --micro amps
ma = 1000 ua; --milli amps
a = 1000 ma; --amps
end units;
```

The type definition begins with a statement that declares the name of the type (**current**) and the range of the type (0 to 1,000,000,000). The first unit declared in the **UNITS** section is the base unit. In the preceding example, the base unit is **na**. After

the base unit is defined, other units can be defined in terms of the base unit or other units already defined. In the preceding example, the unit **ua** is defined in terms of the base unit as 1000 base units. The next unit declaration is **ma**. This unit is declared as 1000 **ua**. The units declaration section is terminated by the **END UNITS** clause. More than one unit can be declared in terms of the base unit. In the preceding example, the **ma** unit can be declared as 1000 **ma** or 1,000,000 **na**. The range constraint limits the minimum and maximum values that the physical type can represent in base units. The unit identifiers all must be unique within a single type. It is illegal to have two identifiers with the same name.

Signal Assignment Statement

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

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

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

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

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

Inertial Delay Model

Inertial delay models the delays often found in switching circuits. It represents the time for which an input valuemust 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. When used with signal assignments, the input value is represented by the value of the expression on the right-hand-side and the output is represented by the target signal.
Fig1.8 shows a simple example of a noninverting buffer with an inertial delay of 10 ns.

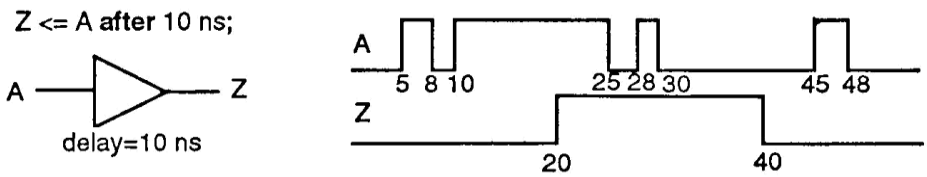

Fig 1.8: Inertial delay example.

Events on signal A that occur at 5 ns and 8 ns are not stable for the inertial delay duration and hence do not propagate to the output. Event on A at 10ns remains stable for more than the inertial delay, and therefore, the value is propagated to the target

signal Z after the inertial delay; Z gets the value 1' at 20 ns. Events on signal A at 25ns and 28 ns do not affect the output since they are not stable for the inertial delay duration. Transition 1' to '0' at time 30 ns on signal A remains stable for at least the inertial delay duration, and therefore, a '0' is propagated to signal Z with a delay of 10 ns; Z gets the new value at 40 ns. Other events on A do not affect the target signal Z. Since inertial delay is most commonly found in digital circuits, it is the default delay model. This delay model is often used to filter out unwanted spikes and transients on signals.

Transport Delay Model

Transport delay models the delays in hardware that do not exhibit any inertial delay. This delay represents purepropagation 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. Fig 1.9 shows an example of a non inverting buffer using a transport delay of 10 ns.

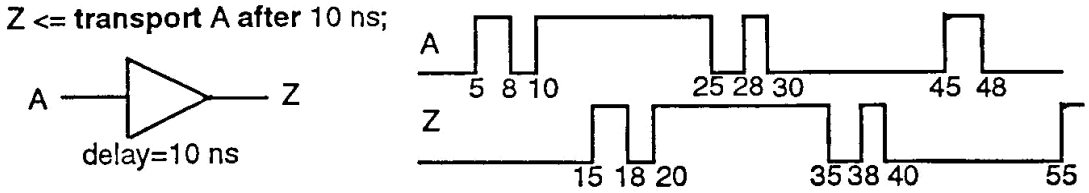

Fig 1.9: Transport delay example.

Ideal delay modeling can be obtained by using this delay model. In this case, spikes would be propagated through instead of being ignored as in the inertial delay case. Routing delays can be modeled using transport delay. An example of a routing delay model is

```
        entity WIRE14 is
port(A: in BIT; Z: out BIT);
        endWIRE14;

        architedtureWIRE14_TRANSPORT    of
        WIRE14 is begin
                process(
                A) begin
                        Z <= transport A after
                0.1 ms; end process;
        endWIRE14_TRANSPORT;
```

Concurrent and Sequential assignments

1. As a set of concurrent assignment statements (to represent dataflow),
2. As a set of sequential assignment statements (to represent be-hav.ior),

Dataflow Style of Modeling(Concurrent assignment)

In this modeling style, the flow of data through the entity is expressed primarily using concurrent signal assignment statements. The structure of the entity is not explicitly specified in this modeling style, but it can be implicitly deduced. Consider the following alternate architecture body for the HALF..ADDER entity that uses this style.

```
        architectureHA_CONCURRENTofHALF_
        ADDER is begin
```

```
        SUM <= A xor B after 8
        ns; CARRY <= A and B
        after 4 ns;
end HA_CONCURRENT;
```

The dataflow model for the HALF_ADDER is described using two concurrent signal assignment statements (sequential signal assignment statements are described in the next section). In a signal assignment statement, the symbol <= implies an assignment of a value to a signal. The value of the expression on the right-hand-side of the statement is computed and is assigned to the signal on the left-hand-side, called the *target signal.* A concurrent signal assignment statement is executed only when any signal used in the expression on the right-hand-side has an event on it, that is, the value for the signal changes.

Delay information is included in the signal assignment statements using after clauses. If either signal A or B, which are input port signals of HALF_ADDER entity, has an event, say at time T, the right-hand-side expressions of both signal assignment statements are evaluated. Signal SUM is scheduled to get the new value after 8 ns while signal CARRY is scheduled to get the new value after 4 ns. When simulation time advances to (T+4) ns, CARRY will get its new value and when simulation time advances to (T+8) ns, SUM will get its new value. Thus, both signal assignment statements execute concurrently.

Concurrent signal assignment statements are concurrent statements, and therefore, the ordering of these statements in an architecture body is not important. Note again that this architecture body, with name HA_CONCURRENT, is also associated with the same HALF_ADDER entity declaration.

Here is a dataflow model for the DECODER2x4 entity.

```
        architscture dec_dataflgw of DECODER2x4
        is signal ABAR, BBAR: BIT;
        begin
Z(3) <=not (A and B and ENABLE);                -- Statement 1
Z(0) <=not (ABAR and BBAR and ENABLE);    --- Statement 2
BBAR <= not B;                                  -- Statement 3
Z(2) <= not (A and BBAR and ENABLE);       -- Statement 4
ABAR <= not A;                                  -- Statement 5
Z(1 ) <= not (ABAR and B and ENABLE);         -- Statement 6
end DEC_DATAFLOW;
```

The architecture body consists of one signal declaration and six concurrent signal assignment statements. The signal declaration declares signals ABAR and BBAR to be used locally within the architecture body. In each of the signal assignment statements, no after clause was used to specify delay. In all such cases, a default delay of 0ns is assumed. This delay of 0ns is also known as delta delay, and it represents an infinitesimally small delay. This small delay corresponds to a zero delay with respect to simulation time and does not correspond to any real simulation time.

To understand the behavior of this architecture body, consider an event happening on one of the input signals, say input port B at time T. This would cause the concurrent signal assignment statements 1,3, and 6, to be triggered. Their right - hand-side expressions would be evaluated and the corresponding values would be scheduled to be assigned to the target signals at time (T+A). When simulation time advances to (T+A), new values to signals Z(3), BBAR, and Z(1), are assigned. Since the value of BBAR changes, this will in turn trigger signal assignment statements, 2 and 4. Eventually, at time (T+2A), signals Z(0) and Z(2) will be assigned their new values.

The semantics of this concurrent behavior indicate that the simulation, as defined by the language, is event-triggered and that simulation time advances to the next time unit when an event is scheduled to occur. Simulation time could also advance a multiple of delta time units. For example, events may have been scheduled to occur at times 1,3,4,4+A, 5,6,6+A, 6+2A, 6+3A, 10,10+A, 15, 15+A time units.

The after clause may be used to generate a clock signal as shown in the following concurrent signal assignment statement

CLK <= **not** CLK **after** 10 ns;

This statement creates a periodic waveform on the signal CLK with a time period of 20 ns as shown in Fig. 1.10.

Fig 1.10: A clock waveform with constant on-off period.

Behavioral Style of modeling (Sequential assignment)

In contrast to the styles of modeling described earlier, the behavioral style of modeling specifies the behavior of an entity as a set of statements that are executed sequentially in the specified order. This set of sequential statements, that are specified inside a process statement, do not explicitly specify the structure of the entity but merely specifies its functionality. A process statement is a concurrent statement that can appear within an architecture body. For example, consider the following behavioral model for the DECODER2x4 entity.

```
architecture DEC_SEQUENTIAL of DECODER2x4 is
 begin
 process (A, B, ENABLE)
variable ABAR, BBAR: BIT;
begin
ABAR := not A;      BBAR := not B;                                  if (ENABLE = '1')
Then          Z(3) <= not (A and B):              Z(0) <= not (ABAR and BBAR);
Z(2) <= not (A and BBAR);              Z(1 ) <= not (ABAR and B);
Else
Z<="1111";          end if; end process; end;
```

A process statement, too, has a declarative part (between the keywords process and begin), and a statement part (between the keywords begin and end process). The statements appearing within the statement part are sequential statements and are executed sequentially. The list of signals specified within the parenthesis after the keyword process constitutes a sensitivity list and the process statement is invoked whenever there is an event on any signal in this list. In the previous example, when an event occurs on signals A, B, the statements appearing within the process statement are executed sequentially.

Signal assignment statements appearing within a process are called *sequential signal assignmentstatements.* Sequential signal assignment statements, including variable assignment statements, are executed sequentially independent of whether an event occurs on any signals in its right-hand-side expression or not; contrast this with the execution of concurrent signal assignment statements in the dataflow modeling style. In the previous architecture body, if an event occurs on any signal. A, B, statement I which is a variable assignment statement, is executed, then statement 2 is executed, and so on. Execution of the third statement, an if statement, causes control to jump to the appropriate branch based on the value of the signal, ENABLE. If the value of ENABLE is 1', the next four signal assignment statements, 4 through 7, are executed independent of whether A, B, ABAR, or BBAR changed values, and the

target signals are scheduled to get their respective values after delta delay. If ENABLE has a value '0', a value of 'V is assigned to each of the elements of the output array, Z. When execution reaches the end of the process, the process suspends itself, and waits for another event to occur on a signal in its sensitivity list.

It is possible to use case or loop statements within a process. The semantics and structure of these statements are very similar to those in other high-level programming languages like C or Pascal. An explicit wait statement can also be used to suspend a process. It can be used to wait for a certain amount of time or to wait until a certain condition becomes true, or to wait until an event occurs on one or more signals. Here is an example of a process statement that generates a clock with a different on-off period. Fig1.11 shows the generated waveform.

```
proces
begin
        CLK <= '0'
        ; wait for 20
        ns; CLK <=
        '1' ; wait for
        12 ns;
end process;
```

Fig 1.11: A clock waveform with varying on-off period.

This process does not have a sensitivity list since explicit wait statements are present inside the process. It is important to remember that a process never terminates. It is always either being executed or in a suspended state. All processes are executed once during the initialization phase of simulation until they get suspended. Therefore, a process with no sensitivity list and with no explicit wait statements will never suspend itself.

A signal can represent not only a wire but also a place holder for a value, that is, it can be used to model a flip-flop. Here is such an example. Port signal Q models a level-sensitive flip-flop.

```
entityLS_DFF is
        port(Q: out BIT; D, CLK:
in BIT): end LS_DFF;

architectureLS_DFF_BEH      of
LS_DFF is begin
        process(D,
        CLK) begin
                if(CLK  =  '1')
                        then Q
                        <= D;
                end
        if;        end
        process;
endLS_DFF_BEH;
```

Delta Delay

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+IA, 15 ns+2A, 15 ns+3A, 22 ns, 22 ns+A, 27 ns, 27 ns+A, and so on.

Consider the AOI_SEQUENTIAL architecture body. Let us assume that an event occurs on input signal D (i.e., there is a change of value on signal D) at simulation time T. Statement I is executed first and TEMPI is assigned a value immediately since it is a variable. Statement 2 is executed next and TEMP2 is assigned a value immediately. Statement 3 is executed next which uses the values of TEMPI and TEMP2 computed in statements I and 2, respectively, to determine the new value for TEMPI. And finally, statement 4 is executed that causes signal Z to get the value of its right -hand-side expression after a delta delay, that is, signal Z gets its value only at time T+A; this is shown in Fig. 1.12

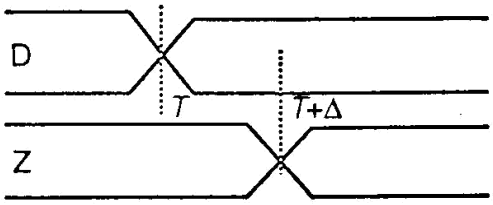

Fig 1.12: Delta delay.

Consider the process PZ described in the previous section. If an event occurs on signal A at time *T,* execution of statement I causes VI to get a value, signal Z is then scheduled to get a value at time T+A, and finally statement 3 is executed in which the old value of signal Z is used, that is, its value at time T, not the value that was scheduled to be assigned in statement 2. The reason for this is because simulation time is still at time T and has not advanced to time T+A. Later when simulation time advances to T+A, signal Z will get its new value. This example shows the important distinction between a variable assignment and a signal assignment statement. Variable assignments cause variables to get their values instantaneously while signal assignments cause signals to get their values at a later time (at least a delta delay later).

So far we have seen two examples of sequential statements, the variable assignment statement and the signal assignment statement. Other kinds of sequential statements are described next.