Brief History of VHDL

The Requirement

The development of VHDL was initiated in 1981 by the United States Department of Defence to address the hardware life cycle crisis. The cost of reprocuring electronic hardware as technologies became obsolete was reaching crisis point, because the function of the parts was not adequately documented, and the various components making up a system were individually verified using a wide range of different and incompatible simulation languages and tools. The requirement was for a language with a wide range of descriptive capability that would *work the same* on any simulator and was independent of technology or design methodology.

Standardization

The standardization process for VHDL was unique in that the participation and feedback from industry was sought at an early stage. A baseline language (version 7.2) was published 2 years before the standard so that tool development could begin in earnest in advance of the standard. All rights to the language definition were *given away* by the DoD to the IEEE in order to encourage industry acceptance and investment.

ASIC Mandate

DoD Mil Std 454 mandates the supply of a comprehensive VHDL description with every ASIC delivered to the DoD. The best way to provide the required level of description is to use VHDL throughout the design process.

VHDL 1993

As an IEEE standard, VHDL must undergo a review process every 5 years (or sooner) to ensure its ongoing relevance to the industry. The first such revision was completed in September 1993, and this is still the most widely supported version of VHDL.

Brief History of VHDL….

VHDL 2000 and VHDL 2002

One of the features that was introduced in VHDL-1993 was shared variables. Unfortunately, it wasn't possible to use these in any meaningful way. A working group eventually resolved this by proposing the addition of protected types to VHDL. VHDL 2000 Edition is simply VHDL-1993 with protected types.

VHDL-2002 is a minor revision of VHDL 2000 Edition. There is one significant change, though: the rules on using buffer ports are relaxed, which makes these much more useful than hitherto.

VHPI

In 2007, an amendment to VHDL 2002 was created. This introduces the VHDL Procedural Interface (VHPI) and also makes a few minor changes to the text of VHDL 2002. Apart from the VHPI itself, no new features were added to VHDL.

The VHPI allows tools programmable access to a VHDL model before and during simulation. In other words, you can write programs in a language such as C that interact with a VHDL simulator.

VHDL 2008

The next revision of VHDL was released in January 2009, and is referred to as "VHDL-2008". F

Summary: History of VHDL

1981	Initiated by US DoD to address hardware life-cycle crisis
1983-85	Development of baseline language by Intermetrics, IBM and TI
1986	All rights transferred to IEEE
1987	Publication of IEEE Standard
1987	Mil Std 454 requires comprehensive VHDL descriptions to be delivered with ASICs
1994	Revised standard (named VHDL 1076-1993)
2000	Revised standard (named VHDL 1076 2000, Edition)
2002	Revised standard (named VHDL 1076-2002)
2007	VHDL Procedural Language Application Interface standard (VHDL 1076c-2007)
2009	Revised Standard (named VHDL 1076-2008)

Describing a Design

- In VHDL an entity is used to describe a hardware module. An entity can be described using,
- Entity declaration
- Architecture
- Configuration
- Package declaration
- Package body
- Let’s see what are these?

Entity Declaration

- Entity Declaration
- It defines the names, input output signals and modes of a hardware module.
- **Syntax** –

```
entity entity_name is
Port declaration;
end entity_name;
```

An entity declaration should start with ‘entity’ and end with ‘end’ keywords. The direction will be input, output or inout.

In	Port can be read
Out	Port can be written
Inout	Port can be read and written
Buffer	Port can be read and written, it can have only one source.

Architecture –

- Architecture can be described using structural, dataflow, behavioral or mixed style.

- **Syntax**

```
architecture architecture_name of entity_name is
architecture_declarative_part;
begin
Statements;
end architecture_name;
```

- Here, we should specify the entity name for which we are writing the architecture body. The architecture statements should be inside the ‘begin’ and ‘énd’ keyword. Architecture declarative part may contain variables, constants, or component declaration.

–

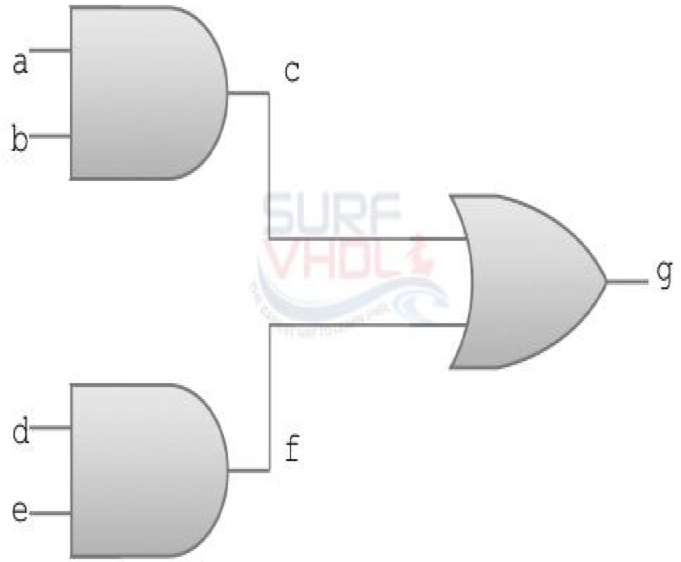
Data Flow Modeling

- In this modeling style, the flow of data through the entity is expressed using concurrent (parallel) signal. The concurrent statements in VHDL are WHEN and GENERATE.
- Besides them, assignments using only operators (AND, NOT, +, *, sll, etc.) can also be used to construct code.
- Finally, a special kind of assignment, called BLOCK, can also be employed in this kind of code.
- In concurrent code, the following can be used −
- Operators
- The WHEN statement (WHEN/ELSE or WITH/SELECT/WHEN);
- The GENERATE statement;
- The BLOCK stateme

- **SYNTAX**

```
architecture architecture_name of entity_name is
architecture_declarative_part;
begin
Statements;
end architecture_name;
```

- **EXAMPLE OF DATAFLOW MODEL**
- **entity** andor **is**
- port(a : **in** std_logic;b : **in** std_logic;d : **in** std_logic;e : **in** std_logic;
- g : **out** std_logic);
- **end** andor;
- **architecture** andor_a **of** and_or **is**
- **begin**
- g <= (a and b) or (d and e);
- **end** andor_a;



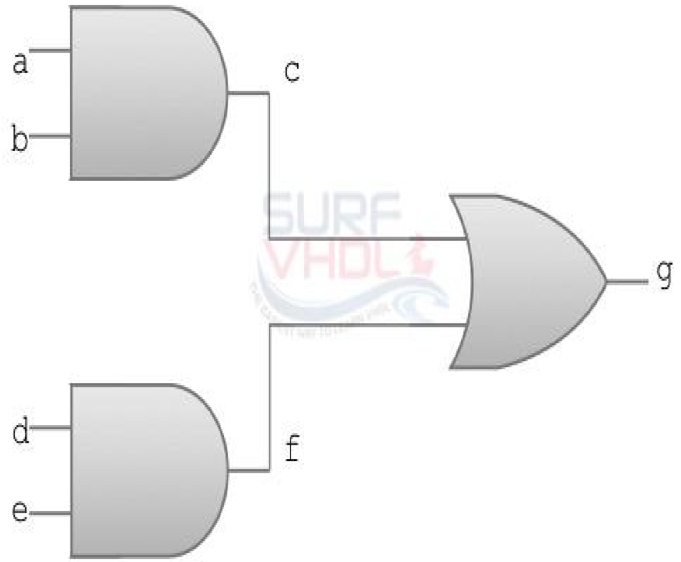
Behavioral Modeling

- In this modeling style, the behavior of an entity as set of statements is executed sequentially in the specified order. Only statements placed inside a PROCESS, FUNCTION, or PROCEDURE are sequential.
- PROCESSES, FUNCTIONS, and PROCEDURES are the only sections of code that are executed sequentially.
- However, as a whole, any of these blocks is still concurrent with any other statements placed outside it.
- One important aspect of behavior code is that it is not limited to sequential logic. Indeed, with it, we can build sequential circuits as well as combinational circuits.
- The behavior statements are IF, WAIT, CASE, and LOOP. VARIABLES are also restricted and they are supposed to be used in sequential code only. VARIABLE can never be global, so its value cannot be passed out directly.

- **SYNTAX**

```
architecture architecture_name of entity_name is
Begin
Process(sensitivity list)
architecture_declarative_part;
begin
Statements;
end architecture_name;
```

- **EXAMPLE OF BEHAVIORAL MODEL**
- **entity** andor **is**
- port(a : **in** std_logic;b : **in** std_logic;d : **in** std_logic;e : **in** std_logic;
- g : **out** std_logic);
- **end** and_or;
- **architecture** and_or_a **of** and_or **is**
- Begin
- process(a,b,d,e,g)
- **begin**
- g <= (a and b) or (d and e);
- end process;
- **end** andor_a;



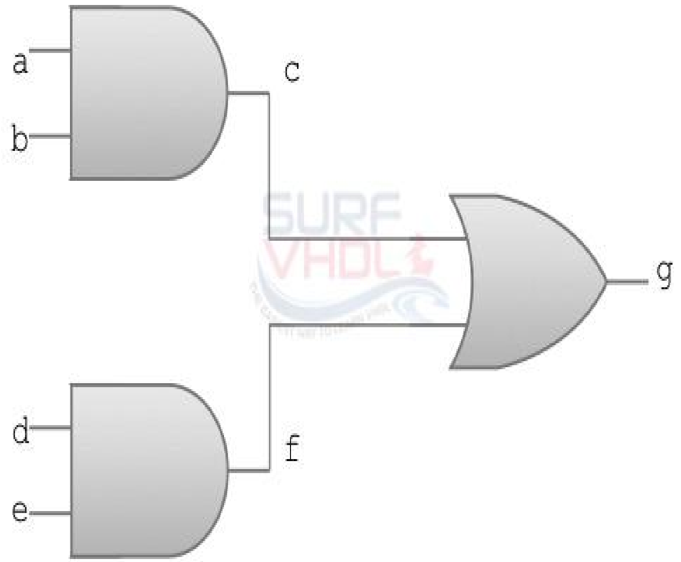
Structural Modeling

- In this modeling, an entity is described as a set of interconnected components. A component instantiation statement is a concurrent statement. 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 Structural modeling, architecture body is composed of two parts − the declarative part (before the keyword begin) and the statement part (after the keyword begin).

- **SYNTAX**

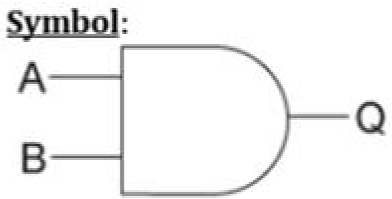
architecture architecture_name of entity_name is
Component declaration;
begin
Statements;
end architecture_name;

- **EXAMPLE OF STRUCTURE MODEL**
- **entity** andor **is**
- port(a : **in** std_logic;b : **in** std_logic;d : **in** std_logic;e : **in** std_logic;
- g : **out** std_logic);
- **end** and_or;
- **architecture** and_or_a **of** and_or **is**
- **Component and2**
- **Port(in0,in1:in** std_logic;out0:out std_logic);
- End component:
- **Component or2**
- **Port(in2,in3:in** std_logic;out1:out std_logic);
- End component;
- **begin**
- g <= (a and b) or (d and e);
- end process;
- **end** andor_a;



Logic Operation – AND GATE

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1




```
VHDL Code:
Library ieee;
use ieee.std_logic_1164.all;
 entity and1 is
port(A,B:in bit ; Q:out bit); end and1;
architecture virat of and1 is
 begin
 Q<=A and B;
end virat;
```

Configuration

Used to bind component instances to design entities and collect architectures to make, typically,
a simulatable test bench. One configuration could create a functional simulation while another configuration
could create the complete detailed logic design.
With an appropriate test bench the results of the two configurations can be compared.
Note that significant nesting depth can occur on hierarchal designs.
There is a capability to bind various architectures with instances of components in the hierarchy. To avoid nesting depth use a configuration for each architecture
level and a configuration of configurations. Most VHDL compilation/simulation systems allow the top level configuration name to be elaborated and simulated.

configuration identifier **of** entity_name **is**
[declarations]
[block configuration ,]
end architecture identifier ;

Package Declaration

Used to declare types, shared variables, subprograms, etc.
package identifier **is**
[declarations, see allowed list below]
end package identifier ;
The example is included in the next section, Package Body. The allowed declarations are:
subprogram declaration type declaration subtype declaration constant, object declaration signal, object declaration variable, object declaration - shared file, object declaration alias declaration component declaration attribute declaration attribute specification use clause
group template declaration group declaration Declarations not allowed include: subprogram body A package body is unnecessary if no subprograms
or deferred constants are declared in the package declaration.

Package Body

Used to implement the subprograms declared in the package declaration.
package body identifier **is**
[declarations, see allowed list below]
end package body identifier ;
package my_pkg **is** -- sample package declaration
type small **is range** 0 **to** 4096;
procedure s_inc(A : **inout** small);

function s_dec(B : small) **return** small;
end package my_pkg;
package body my_pkg **is**
-- corresponding package body **procedure** s_inc(A : **inout** small) **is begin** A := A+1;
end procedure s_inc; **function** s_dec(B : small) **return** small is
begin return B-1;
end function s_dec;
end package body my_pkg;
The allowed declarations are:
subprogram declaration subprogram body type declaration subtype declaration constant, object declaration variable, object declaration – shared file, object declaration alias declaration use clause group template declaration group declaration
Declarations not allowed include: signal, object declaration

Subprograms

There are two kinds of subprograms: procedures and functions. Both procedures and functions written in VHDL must have a body and may have declarations.

Procedures perform sequential computations and return values in global objects or by storing values into formal parameters. Functions perform sequential computations and return a value as the value of the function. Functions do not change their formal parameters. Subprograms may exist as just a procedure body or a function body.

Subprograms may also have a procedure declarations or a function declaration. When subprograms are provided in a package, the subprogram declaration is placed in the package declaration and the subprogram body is placed in the package body.

Procedure Declaration

- *A procedure is a subprogram that defines algorithm for computing values or exhibiting behavior. Procedure call is a statement.*
- **Simplified Syntax**
- **procedure** procedure_name (formal_parameter_list)
- **procedure** procedure_name (formal_parameter_list) **is**
- procedure_declarations
- **begin**
- sequential statements
- **end procedure** procedure_name;
- arameters of the file type have no mode assigned.
- There are three modes available: **in**, **out**, and **inout**. When **in** mode is declared and object class is not defined, then by default it is assumed that the object is a constant. In case of **inout** and **out** modes, the default class is variable. When a procedure is called, formal parameters are substituted by actual parameters. If a formal parameter is a constant, then actual parameter must be an expression. In case of formal parameters such as signal, variable and file, the actual parameters must be objects of the same class. Example 2 presents several procedure declarations with parameters of different classes and modes.
- A procedure can be declared also without any parameters.

Procedure Body

- Procedure body defines the procedure's algorithm composed of sequential statements. When the procedure is called it starts executing the sequence of statements declared inside the procedure body.
- The procedure body consists of the subprogram declarative part After the reserved word **is** and the subprogram statement part placed between the reserved words **begin** and **end**. The key word **procedure** and the procedure name may optionally follow the **end** reserved word.
- Declarations of a procedure are local to this declaration and can declare subprogram declarations, subprogram bodies, types, subtypes, constants, variables, files, aliases, attribute declarations, attribute specifications, use clauses, group templates and group declarations (Example 3).
- A procedure can contain any sequential statements (including **wait** statements). A wait statement, however, cannot be used in procedures which are called from a process with a sensitivity list or from within a function. Examples 4 and 5 present two sequential statements specifications.
- **PROCEDURE CALL**
- A procedure call is a sequential or concurrent statement, depending on where it is used. A sequential procedure call is executed whenever control reaches it, while a concurrent procedure call is activated whenever any of its parameters of **in** or **inout** mode changes its value.
- All actual parameters in a procedure call must be of the same type as formal parameters they substitute.

- **OVERLOADED PROCEDURES**
- The overloaded procedures are procedures with the same name but with different number or different types of formal parameters. The actual parameters decide which overloaded procedure will be called (Example 6).
- **Examples**
- Example 1
- **procedure** Procedure_1 (**variable** X, Y: **inout** Real);
- The above procedure declaration has two formal parameters: bi-directional variables X and Y of the real type.
- Example 2
- **procedure** Proc_1 (**constant** In1: **in** Integer; **variable** O1: **out** Integer);
  procedure Proc_2 (**signal** Sig: **inout** Bit);
- Procedure Proc_1 has two formal parameters: the first one is a constant and it is of mode in and of the integer type, the second one is an output variable of the integer type.
- Procedure Proc_2 has only one parameter, which is a bi-directional signal of the type BIT.
- Example 3
-
```
procedure Proc_3 (X,Y : inout Integer) is
  type Word_16 is range 0 to 65536;
  subtype Byte is Word_16 range 0 to 255;
  variable Vb1,Vb2,Vb3 : Real;
  constant Pi : Real :=3.14;
  procedure Compute (variable V1, V2: Real) is
  begin
    -- subprogram_statement_part
  end procedure Compute;
begin
    -- subprogram_statement_part
end procedure Proc_3;
```
- The example above present different declarations which may appear in the declarative part of a procedure.

- Example 4
- ```
  procedure Transcoder_1 (variable Value: inout bit_vector (0 to 7)) is
  begin
    case Value is
      when "00000000" => Value:="01010101";
      when "01010101" => Value:="00000000";
      when others => Value:="11111111";
    end case;
  end procedure Transcoder_1;
  ```
- The procedure Transcoder_1 transforms the value of a single variable, which is therefore a bi-directional parameter.
- Example 5
- ```
  procedure Comp_3(In1,R:in real; Step :in integer; W1,W2:out real) is
  variable counter: Integer;
  begin
    W1 := 1.43 * In1;
    W2 := 1.0;
    L1: for counter in 1 to Step loop
      W2 := W2 * W1;
      exit L1 when W2 > R;
    end loop L1;
    assert ( W2 < R )
      report "Out of range"
        severity Error;
  end procedure Comp_3;
  ```

- The Comp_3 procedure calculates two variables of mode out: W1 and W2, both of the REAL type. The parameters of mode in: In1 and R constants are of real type and Step of the integer type. The W2 variable is calculated inside the loop statement. When the value of W2 variable is greater than R, the execution of the loop statement is terminated and the error report appears.
- example 6
- **procedure** Calculate (W1,W2: **in** Real; **signal** Out1:**inout** Integer);
  procedure Calculate (W1,W2: **in** Integer; **signal** Out1: **inout** Real);
  -- calling of overloaded procedures:
  Calculate(23.76, 1.632, Sign1);
  Calculate(23, 826, Sign2);
- The procedure Calculate is an overloaded procedure as the parameters can be of different types. Only when the procedure is called the simulator determines which version of the procedure should be used, depending on the actual parameters.

Function Declaration

Used to declare the calling and return interface to a function.
function identifier [(formal parameter list)]
return a_type ;
function random **return** float ;
function is_even (A : integer) **return** boolean ;
Formal parameters are separated by semicolons in the formal parameter list.
Each formal parameter is essentially a declaration of an object that is local to the function.
The type definitions used in formal parameters must be visible at the place where the function is being declared.
No semicolon follows the last formal parameter inside the parenthesis.
Formal parameters may be constants, signals or files.
The default is constant. Formal parameters have the mode **in**. Files do not have a mode. Note that **inout** and **out** are not allowed for functions.
The default is **in** . The reserved word **function** may be preceded by nothing, implying **pure** , **pure** or **impure** . A **pure function** must not contain
a reference to a file object, slice, subelement, shared variable or signal with attributes such as 'delayed, 'stable, 'quiet, 'transaction and must not be a parent of an impure function.

Function Body

Used to define the implementation of the function.
function identifier [(formal parameter list)]
return a_type **is**
[declarations, see allowed list below]
begin
sequential statement(s) return some_value; -- of type a_type
end function identifier ;
function random **return** float **is** variable X : float;
begin
-- compute X **return** X;
end function random ;
The function body formal parameter list is defined above in Function Declaration. When a function declaration
is used then the corresponding function body should have exactly the same formal parameter list.
The allowed declarations are: subprogram declaration subprogram body type declaration subtype declaration constant, object declaration variable, object declaration file, object declaration alias declaration use clause group template declaration group declaration
Declarations not allowed include: signal, object declaration

Identifiers

- *Identifiers* are used both as names for VHDL objects, procedures, functions, processes, design entities, etc., and as reserved words. There are two classes of identifiers: basic identifiers and extended identifiers.
- The *basic identifiers* are used for naming all named entities in VHDL. They can be of any length, provided that the whole identifier is written in one line of code. Reserved words cannot be used as basic identifiers (see *reserved words* for complete list of reserved words). Underscores are significant characters in an identifier and basic identifiers may contain underscores, but it is not allowed to place an underscore as a first or last character of an identifier. Moreover, two underscores side by side are not allowed as well. Underscores are significant characters in an identifier.
- The *extended identifiers* were included in VHDL '93 in order to make the code more compatible with tools which make use of extended identifiers. The extended identifiers are braced between two backslash characters. They may contain any graphic character (including spaces and non-ASCII characters), as well as reserved words. If a backslash is to be used as one of the graphic characters of an extended literal, it must be doubled. Upper- and lower-case letters are distinguished in extended literals.

Important Notes

- A basic identifier must begin with a letter.
- No spaces are allowed in basic identifiers.
- Basic identifiers are not case sensitive, i.e. upper- and lower-case letters are considered identical.
- Basic identifiers consist of Latin letters (a..z), underscores (_) and digits (0..9). It is not allowed to use any special characters here, including non-Latin (language-specific) letters.

VHDL Data Types

- This is a classification objects/items/data that defines the possible set of values which the objects/items/data belonging to that type may assume.
- E.g. (VHDL) integer, bit, std_logic, std_logic_vector
- Other languages (float, double, int , char etc)

VHDL Data Types

Every data object in VHDL can hold a value that belongs to a set of values, specified by using a *type declaration.*

A *type is* a name that has associated with it a set of values and a set of operations. Certain types, and operations that can be performed on objects of these types, are predefined in the language.

Eg., INTEGER is a predefined type with the set of values being integers in a specific range provided by the VHDL system i.e., from $-(2^{31} - 1)$ to $+(2^{31} - 1)$.

Some of the allowable and frequently used predefined operators are +, for addition, -, for subtraction, /, for division, and *, for multiplication.

BOOLEAN is predefined type that has the values FALSE and TRUE, and some of its predefined operators are and, or, nor, nand, and not.

The declarations for the predefined types of the language are contained in package STANDARD.

The language also provides the facility to define new types by using type declarations and also to define a set of operations on these types by writing functions that return values of this new type.

Four major categories of types exist. They are

1. *Scalar* types: Values belonging to these types appear in a sequential order.

2. *Composite* types: These are composed of elements of a single type (an array type) or elements of different types (a record type).

3. *Access* types: These provide access to objects of a given type (via pointers).

4. *File* types: These provide access to objects that contain a sequence of values of a given type.

Scalar types

The values belonging to this type are ordered, i.e., relational operators can be used on these values. Eg., BIT is a scalar type and the expression '0' < 1' is valid and has the value TRUE.

There are four different kinds of scalar types. They are
1. enumeration,
2. integer,
3. physical,
4. floating point.

Integer types, floating point types, and physical types are classified as *numeric* types since the values associated with these types are numeric.

Enumeration and integer types are called *discrete* types since these types have discrete values associated with them.

Every value belonging to an enumeration type, integer type, or a physical type has a *position number* associated with it. This number is the position of the value in the ordered list of values belonging to that type.

ENUMERATION TYPES

An enumeration type declaration defines a type that has a set of user-defined values consisting of identifiers and character literals.

Eg.,

Type MVL **is** ('U','0','1','Z');
type MICRO_OP **is** (LOAD, STORE, ADD, SUB, MUL, DIV);
subtype ARITH_OP **is** MICRO_OP **range** ADD **to** DIV;

MVL is an enumeration type that has the set of ordered values, 'U', '0', '1', and 'Z'.

ARITH_OP is a subtype of the base type MICRO_OP and has a range constraint specified to be from ADD to DIV, i.e., the values ADD, SUB, MUL, and DIV belong to the subtype ARITH_OP.

A range constraint can also be specified in an object declaration as shown in the signal declaration for CLOCK; here the value of signal CLOCK is restricted to '0' or 1'.

INTEGER TYPES

An integer type defines a type whose set of values fall within a specified integer range.

Eg.,

type INDEX **is range** 0 **to** 15;
type WORD_LENGTH **is range** 31 **downto** 0;
subtype DATA_WORD **is** WORD_LENGTH **range** 15 **downto** 0;
type MY_WORD **is range** 4 **to** 6;

Values belonging to an integer type are called *integer literals*. Examples of integer literals are 56349 6E2 0 98_71_28

Physical Types

A physical type contains values that represent measurement of some physical quantity, like time, length, voltage, and current. Values of this type are expressed as integer multiples of a base unit.

Eg.,

```
type CURRENT is range 0 to 1E9
units
nA; -- (base unit) nano-ampere
uA = 1000 nA; -- micro-ampere
mA = 1000 µA; --milli-ampere
Amp = 1000 mA; -- ampere
end units;
subtype FILTER_CURRENT is CURRENT range 10 µA to 5 mA;
```

CURRENT is defined to be a physical type that contains values from 0nA to 1nA.

COMPOSITE TYPES

A composite type represents a collection of values. There are two composite types: an array type and a record type.

An array type represents a collection of values all belonging to a single type; on the other hand, a record type represents a collection of values that may belong to same or different types.

An object belonging to a composite type represents a collection of subobjects, one for each element of the composite type. An element of a composite type could have a value belonging to either a scalar type, a composite type, or an access type.

Eg., a composite type may be defined to represent an array of an array of records. This provides the capability of defining arbitrarily complex composite types.

ARRAY TYPES

An object of an array type consists of elements that have the same type.

Eg.,

type ADDRESS_WORD **is array** (0 **to** 63) **of** BIT;
type DATA_WORD **is array** (7 **downto** 0) **of** MVL;
type ROM **is array** (0 **to** 125) **of** DATA_WORD;

ADDRESS_BUS is a one-dimensional array object that consists of 64 elements of type BIT.

ROM_ADDR is a one-dimensional array object that consists of 126 elements, each element being another array object consisting of 8 elements of type MVL. Hence an array of arrays is created.

RECORD TYPES

An object of a record type is composed of elements of same or different types.

It is analogous to the record data type in Pascal and the struct declaration in C.

Eg.,
type PIN_TYPE **is range** 0 **to** 10;
type MODULE **is**
record
SIZE: INTEGER **range** 20 **to** 200;
CRITICAL_DLY: TIME;
NO_INPUTS: PIN_TYPE:
NO_OUTPUTS: PIN_TYPE;
end record;

Values can be assigned to a record type object using aggregates.

ACCESS TYPES

Values belonging to an access type are pointers to a dynamically allocated object of some other type. They are similar to pointers in Pascal and C languages.

Eg.,

```
-- MODULE is a record type declared in the previous sub-section.
type PTR is access MODULE;
type FIFO is array (0 to 63, 0 to 7) of BIT;
type FIFO_PTR is access FIFO;
```

PTR is an access type whose values are addresses that point to objects of type MODULE. Every access type may also have the value null, which means that it does not point to any object.

File Types

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

Syntax of a file type declaration

type *file-type-name* **Is file of** *type-name,*

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

Eg.,
type VECTORS **is file of** BIT_VECTOR;
type NAMES **is file of** STRING;

.

File Types

A file of type VECTORS has a sequence of values of type BIT_VECTOR; a file of type NAMES has a sequence of strings as values in it.

A file is declared using a file declaration.

Syntax of a file declaration

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.

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.

Eg.,
file VEC_FILE: VECTORS **is in** "/usr/home/jb/uart/div.vec";
file OUTPUT: NAMES **is out** "stdout";

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 "/usr/home/jb/uart/div.vec" in the host environment

VHDL Data objects

- There are four types of data objects in VHDL:
- signals
- variables
- constants
- files

Signal

- The signal represents interconnection wires between ports
- it may be declared in the declaration part of
- Packages,entities,architectures,blocks
- The signal declaration is

```
signal signal_name : signal_type;
 Signal assignment: <=
```

Variable

- The variable locally stores temporary data and it is used only inside a sequential statement that means
- Process,function,procedures
- The variable is visible only inside processes and subprograms in which it is declared.
- The variable declaration is

```
variable variable_name : variable_type;
Variable assignment: :=
```

Constant

- The constant names specific values to make the model better documented and easy to update.
- The constant can be declared in all the declarative VHDL statement,
- sequential
- concurrent
- that means it may be declared in the declaration section of packages, entities, architectures, processes, subprograms and blocks
- The constant declaration is

```
constant constant_name : constant_type := value;
```

File

- The File type is used to access File on disk.
- It is used **only in test bench**; in fact File type cannot be implemented in hardware.
- In order to use the FILE type you shall include the **TextIO** package that contains all procedures and functions that allow you to read from and write to formatted text files.
- Input ASCII files are handled as file of lines, where a line is a string, terminated by a carriage return.
- TextIO package declares a type line used to hold
- a line read from an input file
- a line to write to an output file

Operators

- *Operators are means for constructing expressions.*
- VHDL has a wide set of different operators, which can be divided into groups of the same precedence level (priority). The table below lists operators grouped according to priority level, highest priority first.

miscellaneous operators	**** | abs | not**
multiplying operators	*** | / | mod | rem**
sign operators	**+ | -**
adding operators	**+ | - | &**
shift operators	**sll | srl | sla | sra | rol | ror**
relational operators	**= | /= | < | <= | > | >=**
logical operators	**and | or | nand| nor | xor | xnor**

Libraries and Packages in VHDL

- **Built-in Libraries and Packages.**
- In most vhdl programs you have already seen examples of packages and libraries. Here are two:
- library ieee;
- use ieee.std_logic_1164.all;
- use ieee.std_logic_signed.all;
- The packages are "std_logic_1164" and "std_logic_signed" and the library is "ieee". Since the "scope" of the library statement extends over the entire file, it is not necessary to repeat that for the second package.
- It's instructive to show where the packages are physically located. For Altera Max+2 and Xilinx Foundation these locations typically are:
- Altera: ~\maxplus2\vhdl93\ieee\std1164.vhd
- Xilinx: ~\fndtn\synth\lib\packages\ieee\src\std_logic_1164.vhd
- It is thus tempting to come to the conclusion that the "library ieee;" statement indicates the "directory" in which the std_logic_1164 package is located.

User Libraries and Packages.

- User libraries and packages are setup very similarly to the built-in ones. However, in that case, the user is responsible for the directory structure, the contents of the files, etc. Note that the user must then also set up the pointer to the package. The following shows a complete example of this arrangement. There are two ways to do this: 1) with the "work" directory; 2) with a user library.
- **With the "work" library (Max+2-specific)**
- First we define the "work" library . This is the *pointer* to the working directory, i.e. where all the design files are kept and the software "knows" which one that is, as it is set up by the project definition. Thus if the user were to put a .vhd file, containing a package, in the current working directory, the statement referencing that package would be:
- library work; -- not needed, but OK to include.
- work.usr_package_name.all;

SECTION- B

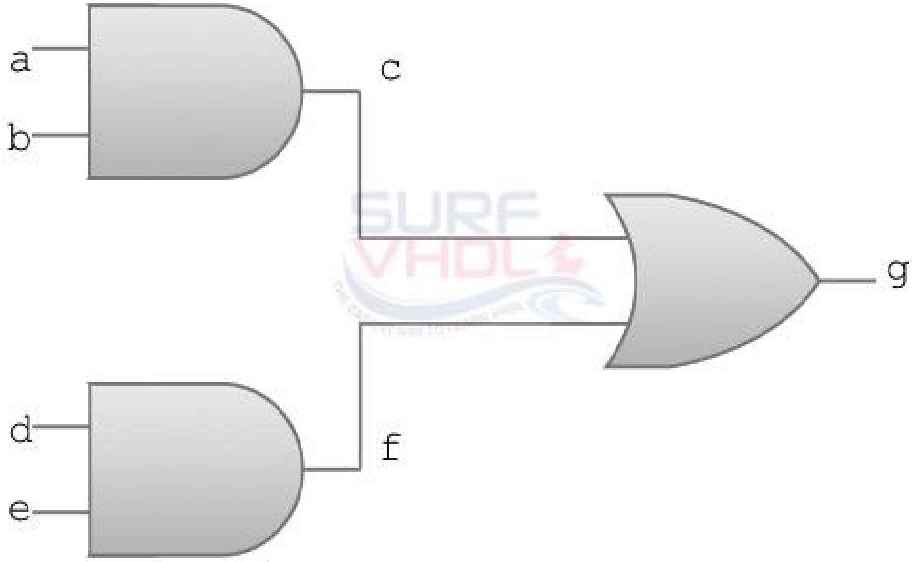
VHDL Process Statement

- A process statement is concurrent statement itself
- The **VHDL process** syntax contains:
- sensitivity list
- declarative part
- sequential statement section
- The process statement is very similar to the classical programming language. The code inside the process statement is executed sequentially. The process statement is declared in the concurrent section of the architecture, so two different processes are executed concurrently.
- The declaration process statement is

```
process_label : process(sensitivity_list)
 -- declarative part
 begin
-- sequential statement
 end process process_label;
```

- The process label is optional, you can avoid using the label. Labeling all process you use, the code will be clear and it will be simple to arrange the simulation environment.
- We will address the process statement in the next lessons. Here there is a simple example of the *and_or2* entity implemented with a process.

```
entity and_or is
 port( a: in std_logic; b: in
std_logic; d: in std_logic; e: in
std_logic; g : out std_logic);
 end and_or;
architecture and_or_a of and_or is
 -- declarative part: empty
 begin
 process_and_or : process(a,b,d,e)
 -- declarative part: empty
begin g <= (a and b) or (d and e);
 end process process_and_or;
end and_or_a;
```



VHDL Sequential Statements

These statements are for use in Processes, Procedures and Functions. The signal assignment statement has unique properties when used sequentially.

These Sequential Statements are

- wait statement
- assertion statement
- report statement
- signal assignment statement
- variable assignment statement
- procedure call statement
- if statement
- case statement
- loop statement
- next statement
- exit statement
- return statement
- null statement

wait statement

Cause execution of sequential statements to wait.
[label:] **wait** [sensitivity clause] [condition clause] ;
wait for 10 ns; -- timeout clause, specific time delay.
wait until clk='1'; -- condition clause, Boolean condition
wait until A>B **and** S1 **or** S2; -- condition clause, Boolean condition
wait on sig1, sig2; -- sensitivity clause, any event on any -- signal terminates wait

assertion statement

Used for internal consistency check or error message generation.
[label:] **assert** boolean_condition [**report** string] [**severity** name] ;
assert a=(b **or** c); **assert** j<i **report** "internal error, tell someone";
assert clk='1' **report** "clock not up" **severity** WARNING;
predefined severity names are: NOTE, WARNING, ERROR, FAILURE default severity for assert is ERROR

report statement

Used to output messages.

[label:] **report** string [**severity** name] ;

report "finished pass1"; -- default severity name is

NOTE **report** "Inconsistent data." **severity** FAILURE;

signal assignment statement

The signal assignment statement is typically considered a concurrent statement rather than a sequential statement.

It can be used as a sequential statement but has the side effect of obeying the general rules for when the target actually gets updated.

In particular, a signal can not be declared within a process or subprogram but must be declared is some other appropriate scope.

Thus the target is updated in the scope where the target is declared when the sequential code reaches its end or encounters a 'wait' or other event that triggers the update.

here value is assigned to signal using symlole <=,when it is used in data flow model then it is concurrent in naturs and

when used in behavioral model then sequential in nature

Examples

sig1 <= sig2;

Sig <= Sa **and** Sb **or** Sc **nand** Sd **nor** Se **xor** Sf **xnor** Sg;

sig1 <= sig2 after 10 ns;

variable assignment statement

Assign the value of an expression to a target variable.

[label:] target := expression ;

A := -B + C * D / E **mod** F **rem** G **abs** H;

Sig := Sa **and** Sb **or** Sc **nand** Sd **nor** Se **xor** Sf **xnor** Sg;

procedure call statement

Call a procedure.

[label:] procedure-name [(actual parameters)] ;

do_it; -- no actual parameters

compute(stuff, A=>a, B=>c+d); -- positional association first,

-- then named association of

-- formal parameters to actual parameters

if statement

Conditional structure. [label:]
if condition1 **then** sequence-of-statements
elsif condition2 **then** _ optional sequence-of-statements /
elsif condition3 **then** _ optional sequence-of-statements /
... **else** _ optional sequence-of-statements / **end if** [label] ;

```
if a=b then
 c:=a;
elsif b<c then
 d:=b;
 b:=c;
else
do_it;
 end if;
```

case statement

Execute one specific case of an expression equal to a choice.
The choices must be constants of the same discrete type as the expression.

[label:] **case** expression **is**
when choice1 => sequence-of-statements
when choice2 => _ optional sequence-of-statements / ...

when others => _ optional if all choices covered sequence-of-statements /
end case [label] ;

Example

case my_val **is**
when 1 => a:=b;
when 3 => c:=d;
when others => **null**;
end case;

loop statement

Three kinds of iteration statements.
1)Loop;
2)For loop;
3)While loop;
Syntax:-

[label:] **loop** sequence-of-statements -- use exit statement to get out
end loop [label] ;

[label:]
 for variable **in** range **loop**
sequence-of-statements;
 end loop [label] ;

 [label:] **while** condition **loop**
sequence-of-statements ;
end loop [label] ;

loop input_something;
 exit when end_file;
 end loop;

Example

 for I **in** 1 **to** 10 **loop**
AA(I) := 0;
end loop;

while not end_file **loop**
 input_something;
 end loop;

all kinds of the loops may contain the 'next' and 'exit' statements.

next statement

A statement that may be used in a loop to cause the next iteration.

[label:] **next** [label2] [**when** condition] ;

next;

next outer_loop;

next when A>B;

next this_loop **when** C=D **or** done; -- done is a Boolean variable

exit statement

A statement that may be used in a loop to immediately exit the loop.

[label:] **exit** [label2] [**when** condition] ;

exit;

exit outer_loop;

exit when A>B;

exit this_loop **when** C=D **or** done; -- done is a Boolean variable

return statement

Required statement in a function, optional in a procedure.

[label:] **return** [expression] ;

return; -- from somewhere in a procedure

return a+b; -- returned value in a function

null statement

Used when a statement is needed but there is nothing to do.
[label:] **null** ;

null;

EXAMPLE

```
process

variable count: unsigned (7 downto 0);
 begin

wait until clk = '1';

 if reset = '1' then count := "00000000";
else

count := count + 1;
 end if;

result <= count;
end process;
```

VHDL Generics

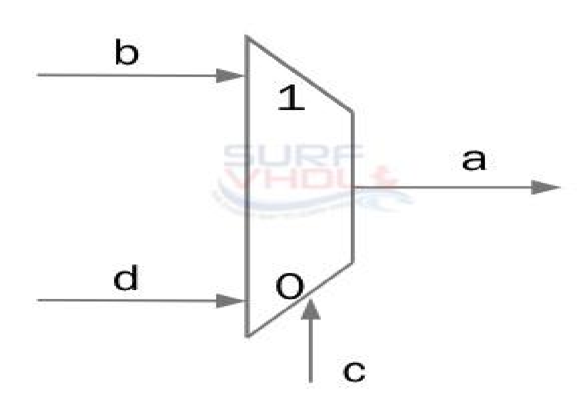
- The RAMs are similar. Have the same interface in terms of signal but **different** access time address and BUS width. In this case, there is no need to write twice the same module. It should be possible to **parameterize** the component during the instantiation. In order to implement parameterization of an entity VHDL introduce the **generic** clause.
- In the entity declaration, all the values that have to be customized can be passed using **generic** clause.
- In the component instantiation, the **generic map** statement can map the new values in the component.
- You should notice that in the entity declaration the generic parameters can have a default values.

```
entity RAM is
generic( data_width : integer := 64; addr_width : integer := 12; Taa :
time := 50; Toe : time := 40);
 port( oeb, wrb, csb : in std_logic; data : inout
std_logic_vector(data_width-1 downto 0);

 addr : in std_logic_vector(addr_width-1 downto 0)); end RAM;
```

VHDL Concurrent Conditional Assignment

- The **Conditional Signal Assignment** statement is concurrent because
- it is assigned in the concurrent section of the architecture. It is possible to
- implement the same code in a sequential version, as we will see next.
- The conditional signal assignment statement is a process that assigns values to a signal.
- It is a concurrent statement; this means that you must use it only in concurrent code sections.
- The statement that performs the same operation in a sequential environment is the "**if**" statement.
- The syntax for a conditional signal assignment statement is:

```
a <= b when c='1' else d;
```



Concurrent Conditional Signal Assignment

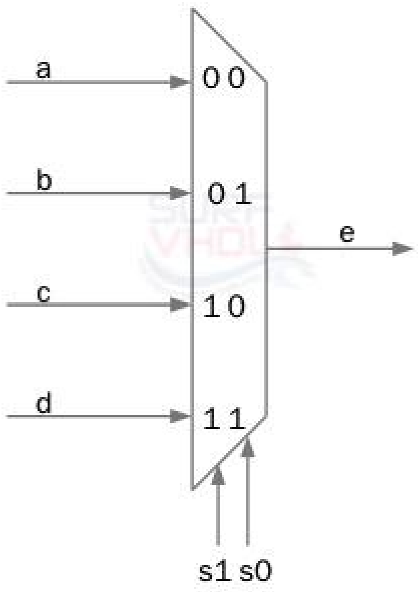
- This example extends the previous one. This is a 4-way mux, implemented as concurrent code.
- The architecture declarative section is empty. As you can notice, we don’t care about how the mux is implemented.
- In this moment we don’t’ talk about logic gate, and or nand ect, we are describing the **behavior** of circuit using a high level description.

Concurrent Conditional Signal Assignment

- This example is the same 4-way mux as the previous one, in which we used a different syntax to implement the selector. In this case, we have introduced the statement “with select”.
- In the architecture declarative section, we declared a signal “***sel***” of type **integer** used to address the mux. The signal “***sel***” is coded as binary to integer.
- The statement “**with select**” allows compacting the syntax of the mux code.
- Vhdl code given below

```
entity mux4_select is
port(
a : in bit;
b : in bit;
c : in bit;
d : in bit;
s0 : in bit;
s1 : in bit;
e : out bit);
end mux4_select;
architecture mux4_select_a of mux4_select is
signal sel : integer;
begin
sel <= 0 when (s1='0' and s0='0') else
1 when (s1='0' and s0='1') else
2 when (s1='1' and s0='0') else
3;
with sel select
e <= a when 0,
b when 1,
c when 2,
d when others;
end mux4_select_a;
```

```vhdl
entity mux4 is
port(
a : in bit;
b : in bit;
c : in bit;
d : in bit;
s0 : in bit;
s1 : in bit;
e : out bit);
end mux4;
architecture mux4_a of mux4 is
-- declarative part: empty
begin
e <= a when (s1='0' and s0='0') else
b when (s1='0' and s0='1') else
c when (s1='1' and s0='0') else
d;
end mux4_a;
```

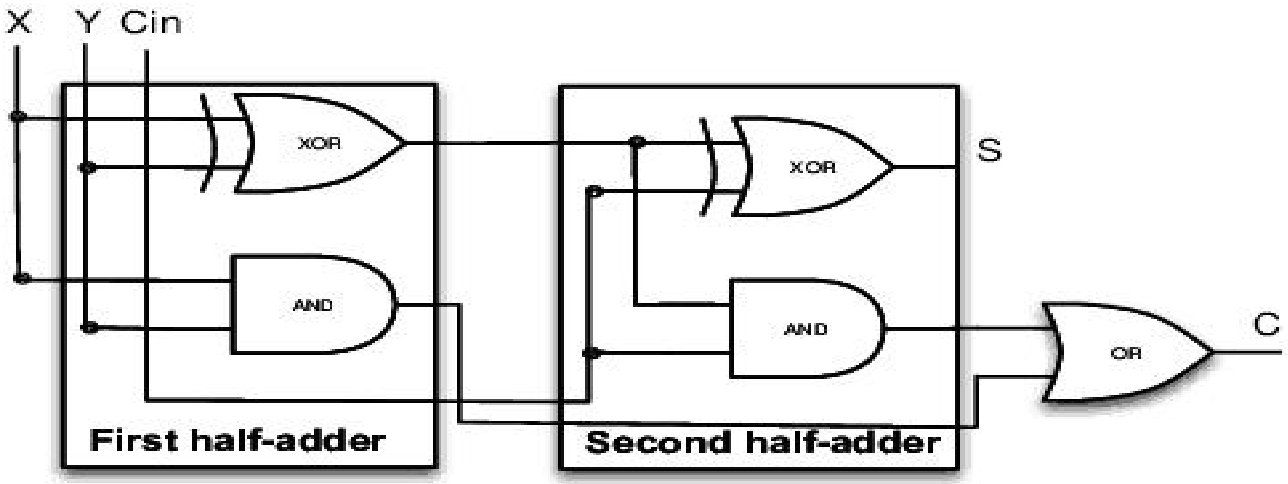


SECTION- C

VHDL code for all logic gates using data flow modeling

```
entity ALLGATES_SOURCE is
 Port ( A,B : in  STD_LOGIC; P, Q, R, S, T, U, V : out  STD_LOGIC);
end ALLGATES_SOURCE;
 architecture dataflow of ALLGATES_SOURCE is
 begin

     P  <= A and B;
    Q <= A nand B;
    R  <= A or B;
    S <= A nor B;
    T <= not A;
    U <= A xor B;
     V <= A xnor B;
  end dataflow;
```

VHDL code for full adder using structure modeling



Vhdl code for full adder

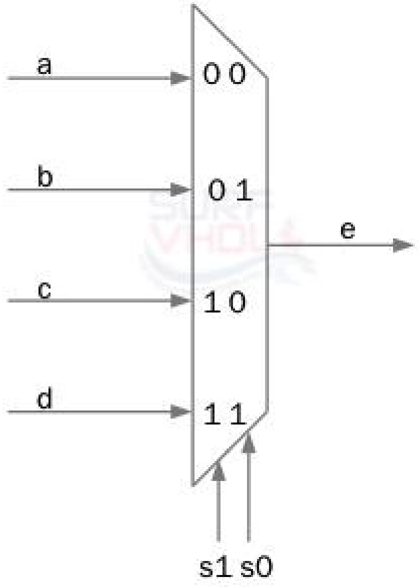
```
library IEEE;
 use IEEE.STD_LOGIC_1164.ALL;
 use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity FAdder is
 Port ( FA, FB, FC : in STD_LOGIC; FS, FCA : out
STD_LOGIC);
end FAdder;
architecture structural of FAdder is
component HA is
Port ( A,B : in STD_LOGIC; S,C : out STD_LOGIC);
end component;
 component ORGATE is
Port ( X,Y: in STD_LOGIC; Z: out STD_LOGIC);

end component;
 SIGNAL S0,S1,S2:STD_LOGIC;
 begin
 U1:HA PORT MAP(A=>FA,B=>FB,S=>S0,C=>S1);
 U2:HA PORT MAP(A=>S0,B=>FC,S=>FS,C=>S2);
 U3:ORGATE PORT MAP(X=>S2,Y=>S1,Z=>FCA);
 end structural;
```

VHDL code for half adder using data flow modeling

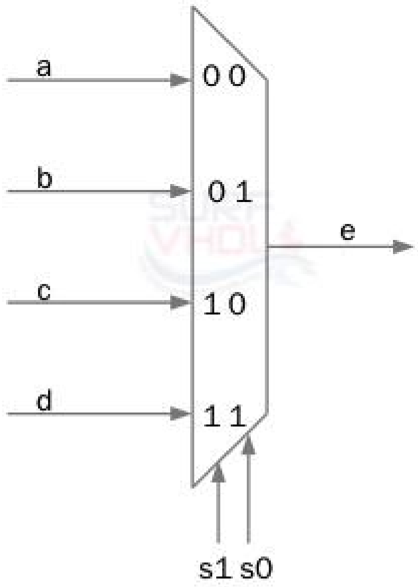
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
 use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity HA is Port ( A,B : in  STD_LOGIC; S,C : out  STD_LOGIC);
 end HA;
architecture dataflow of HA is
 begin
S <= A XOR B;
 C <= A AND B;
 end dataflow;
```

VHDL Code for 4:1 Mux:

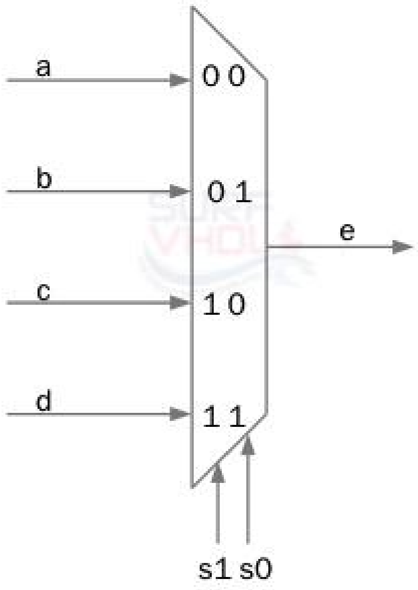
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity mux_4to1 is
port(A,B,C,D : in STD_LOGIC;
S0,S1: in STD_LOGIC;Z: out STD_LOGIC);
end mux_4to1;
architecture bhv of mux_4to1 is
begin
process (A,B,C,D,S0,S1) is
begin
if (S0 ='0' and S1 = '0') then
Z <= A;
elsif (S0 ='1' and S1 = '0') then
Z <= B;
elsif (S0 ='0' and S1 = '1') then
Z <= C;
else
Z <= D;
end if;
end process;
end bhv;
```



- **4:1 MUX USING CONDITIONAL SIGNAL ASSIGNMENT STATEMENT and select signal assignment**
- library IEEE;
- use IEEE.STD_LOGIC_1164.all;
- **entity** mux4_select **is**
- port(a : **in** bit;b : **in** bit;c : **in** bit;d : **in** bit;s0 : **in** bit;s1 : **in** bit; e : **out** bit);
- **end** mux4_select;
- **architecture** mux4_select_a **of** mux4_select **is**
- **signal** sel : integer;
- **begin**
- sel <= 0 when (s1='0' and s0='0') **else**
- 1 when (s1='0' and s0='1') **else**
- 2 when (s1='1' and s0='0') **else**
- 3;
- **with** sel **select**
- e <= a **when** 0,
- b **when** 1,
- c **when** 2,
- d **when others**;
- **end** mux4_select_a;



- **4:1 MUX USING CONDITIONAL SIGNAL ASSIGNMENT STATEMENT entity** mux4 **is**
- port(
- a : **in** bit;
- b : **in** bit;
- c : **in** bit;
- d : **in** bit;
- s0 : **in** bit;
- s1 : **in** bit;
- e : **out** bit);
- **end** mux4;
- **architecture** mux4_a **of** mux4 **is**
- -- declarative part: empty
- **begin**
- e <= a when (s1='0' and s0='0') **else**
- b when (s1='0' and s0='1') **else**
- c when (s1='1' and s0='0') **else**
- d;
- **end** mux4_a;



VHDL Code for 1:4 Demux:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity demux_1to4 is
port(
F : in STD_LOGIC;
S0,S1: in STD_LOGIC;
A,B,C,D: out STD_LOGIC
);
end demux_1to4;
architecture bhv of demux_1to4 is
begin
process (F,S0,S1) is
begin
if (S0 ='0' and S1 = '0') then
A <= F;
elsif (S0 ='1' and S1 = '0') then
B <= F;
elsif (S0 ='0' and S1 = '1') then
C <= F;
else
D <= F;
end if;
end process;
end bhv;
```

VHDL CODE FOR 4:2 ENCODER

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
 entity ENCODER_SOURCE is
 Port ( I : in  STD_LOGIC_VECTOR (3 downto 0);

  Y : out STD_LOGIC_VECTOR (1 downto 0)); end
ENCODER_SOURCE;
architecture Behavioral of ENCODER_SOURCE is
 begin
process (I)
begin
case I is
when "0001" => Y <= "00" ;
 when "0010" => Y <= "01" ;
when "0100" => Y <= "10" ;
 when others => Y <= "11" ;
end case; end process; end Behavioral;
```

VHDL CODE FOR 2:4 DECODER

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
 entity DECODER_SOURCE is
 Port ( Y : OUT  STD_LOGIC_VECTOR (3 downto 0);
  I : IN STD_LOGIC_VECTOR (1 downto 0));
end DECODER_SOURCE;
architecture Behavioral of DECODER_SOURCE is
begin
process (Y)
begin
case 1 is
 when "00" => Y <= "0001" ;
 when "01" => Y <= "0010" ;
when "10" => Y <= "0100" ;
 when others => Y <= "1000" ;
end case;
end process;
 end Behavioral;
```

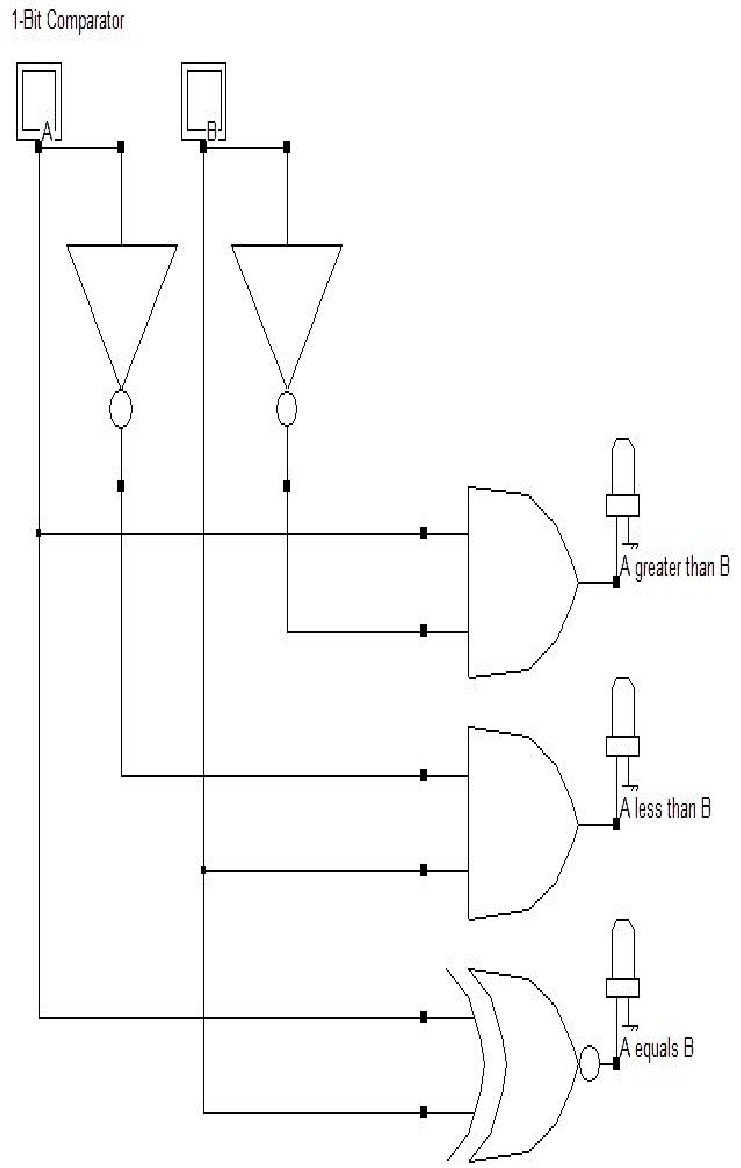
BCTO 7 SEGMENT DISPLAYD

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
 entity BCD_7 is
 Port ( LED : OUT  STD_LOGIC_VECTOR (6 downto 0);
  LED_BCD : IN STD_LOGIC_VECTOR (3 downto 0));
end BCD_7;
architecture Behavioral of BCD_7 is
begin
process(LED_BCD)
begin case LED_BCD is
 when "0000" => LED_out <= "0000001"; -- "0"
 when "0001" => LED_out <= "1001111"; -- "1"
when "0010" => LED_out <= "0010010"; -- "2"
when "0011" => LED_out <= "0000110"; -- "3"
when "0100" => LED_out <= "1001100"; -- "4"
when "0101" => LED_out <= "0100100"; -- "5"
 when "0110" => LED_out <= "0100000"; -- "6"
when "0111" => LED_out <= "0001111"; -- "7"
when "1000" => LED_out <= "0000000"; -- "8"
when "1001" => LED_out <= "0000100"; -- "9"
when "1010" => LED_out <= "0000010"; -- a
when "1011" => LED_out <= "1100000"; -- b
when "1100" => LED_out <= "0110001"; -- C
when "1101" => LED_out <= "1000010"; -- d
when "1110" => LED_out <= "0110000"; -- E
when "1111" => LED_out <= "0111000"; -- F
end case;
end process;
end Behavioral
```

Decimal Digit	Input lines				Output lines							Display pattern
	A	B	C	D	a	b	c	d	e	f	g	
0	0	0	0	0	1	1	1	1	1	1	0	0
1	0	0	0	1	0	1	1	0	0	0	0	1
2	0	0	1	0	1	1	0	1	1	0	1	2
3	0	0	1	1	1	1	1	1	0	0	1	3
4	0	1	0	0	0	1	1	0	0	1	1	4
5	0	1	0	1	1	0	1	1	0	1	1	5
6	0	1	1	0	1	0	1	1	1	1	1	6
7	0	1	1	1	1	1	1	0	0	0	0	7
8	1	0	0	0	1	1	1	1	1	1	1	8
9	1	0	0	1	1	1	1	1	0	1	1	9

Comparator circuit

- A comparator is a combinational logic circuit that compares two inputs and gives an output that indicates the relationship between them. There are three outputs.
- An output that indicates if number A is greater than number B.
- An output that indicates if it's smaller.
- And finally, an output that indicates if the two numbers are equal.
- Let's take a look at its logic circuit for some clarity.



VHDL code for comparator using behavioral method

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
 use IEEE.STD_LOGIC_ARITH.ALL;
 use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity COMPARATOR_SOURCE is
  Port ( A : in  STD_LOGIC_VECTOR (1 downto 0);
 G,L,E : out  STD_LOGIC);
end COMPARATOR_SOURCE;
 architecture Behavioral of COMPARATOR_SOURCE is
Begin
 process (A)
 begin
G <= '0';
 L <= '0';
 E <= '0';
case A is
 when (A(0)<=A(1)) => E <= '1';
when "01" => L <= '1';
 when others => G <= '1';
end case;
end process;
end Behavioral;
```

Truth table for 1-bit comparator

A	B	A>B	A<B	A=B
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	1

VHDL code for Parallel In Parallel Out Shift Register

```
library ieee;
 use ieee.std_logic_1164.all;
 entity pipo is port( clk : in std_logic;
 D: in std_logic_vector(3 downto 0);
 Q: out std_logic_vector(3 downto 0) );
end pipo;
 architecture arch of pipo is
begin process (clk)
 begin
 if (CLK'event and CLK='1') then
 Q <= D;
 end if;
 end process;
 end arch;
```

For parallel in – parallel out shift registers, all data bits appear on the parallel outputs immediately following the simultaneous entry of the data bits. This code is a four-bit parallel in – parallel out shift register constructed by D flip-flops.

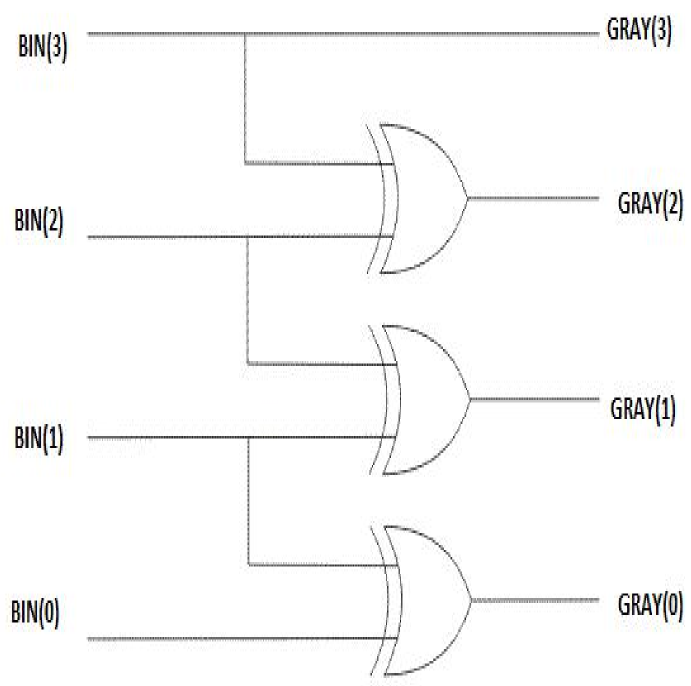
Serial In – Parallel Out Shift Registers

For Serial in – parallel out shift registers, all data bits appear on the parallel outputs following the data bits enters sequentially through each flipflop.
The following code is a four-bit Serial in – parallel out shift register constructed by D flip-flops.

```
library ieee;
 use ieee.std_logic_1164.all
entity sipo is
port( clk, clear : in std_logic; Input_Data: in std_logic; Q: out std_logic_vector(3 downto 0) );
end sipo;
 architecture arch of sipo is
 begin process (clk)
 begin
 if clear = '1' then
 Q <= "0000";
elsif (CLK'event and CLK='1') then
 Q(3 downto 1) <= Q(2 downto 0);
Q(0) <= Input_Data;
end if;
 end process;
end arch;
```

VHDL Code for Gray code to Binary conversion:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity gray2bin is
port(
G : in std_logic_vector(3 downto
0);     --gray code input

bin : out std_logic_vector(3 downto
0)   --binary output
        );
end gray2bin;

architecture gate_level of gray2bin
is

begin

--xor gates.
bin(3) <= G(3);
bin(2) <= G(3) xor G(2);
bin(1) <= G(3) xor G(2) xor G(1);
bin(0) <= G(3) xor G(2) xor G(1) xo
r G(0);

end;
```



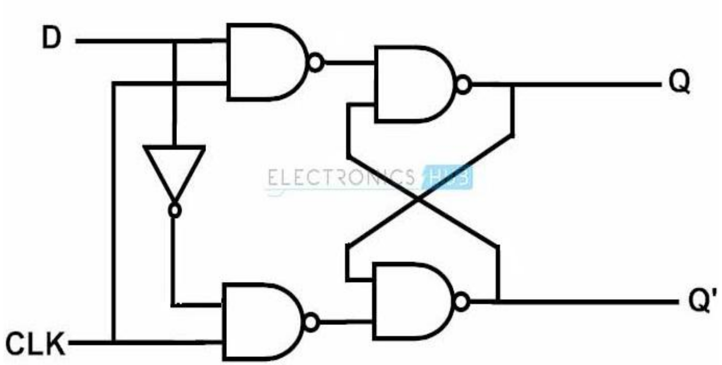
VHDL Code for a D Flip Flop

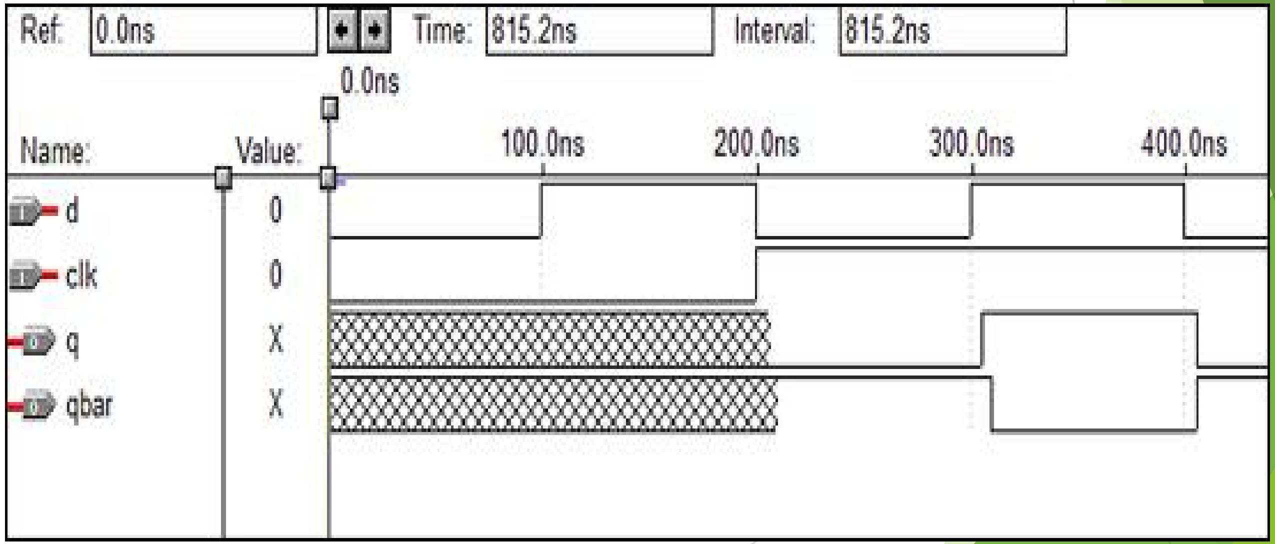
- Library ieee;
- use ieee.std_logic_1164.all;

- entity dflip is
-    port(d,clk:in bit; q,qbar:out bit);
- end dflip;
-
- architecture virat of dflip is
- begin

q<=d when (ckl='1' and clk'event)else
'0';
qbar<= not d;

- end virat;

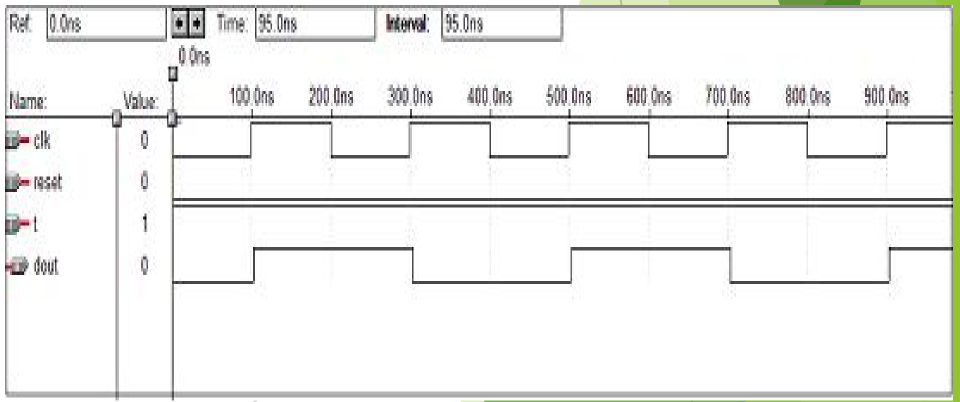




VHDL Code for a T Flip Flop

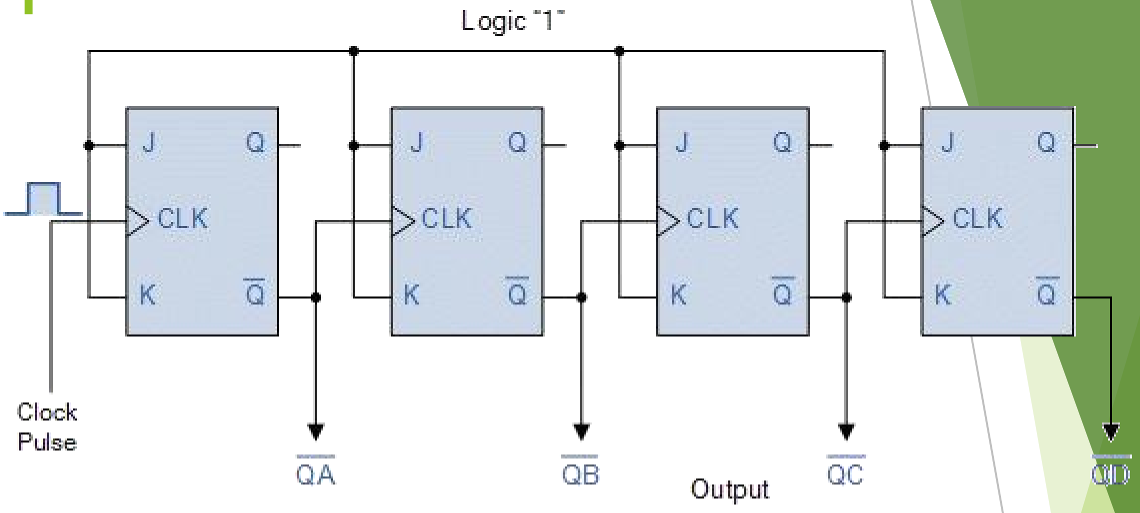
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Toggle_flip_flop is
  port(

    t : in STD_LOGIC;
    clk : in STD_LOGIC;
    reset : in STD_LOGIC;
    Q : out STD_LOGIC
  );
end Toggle_flip_flop;
```

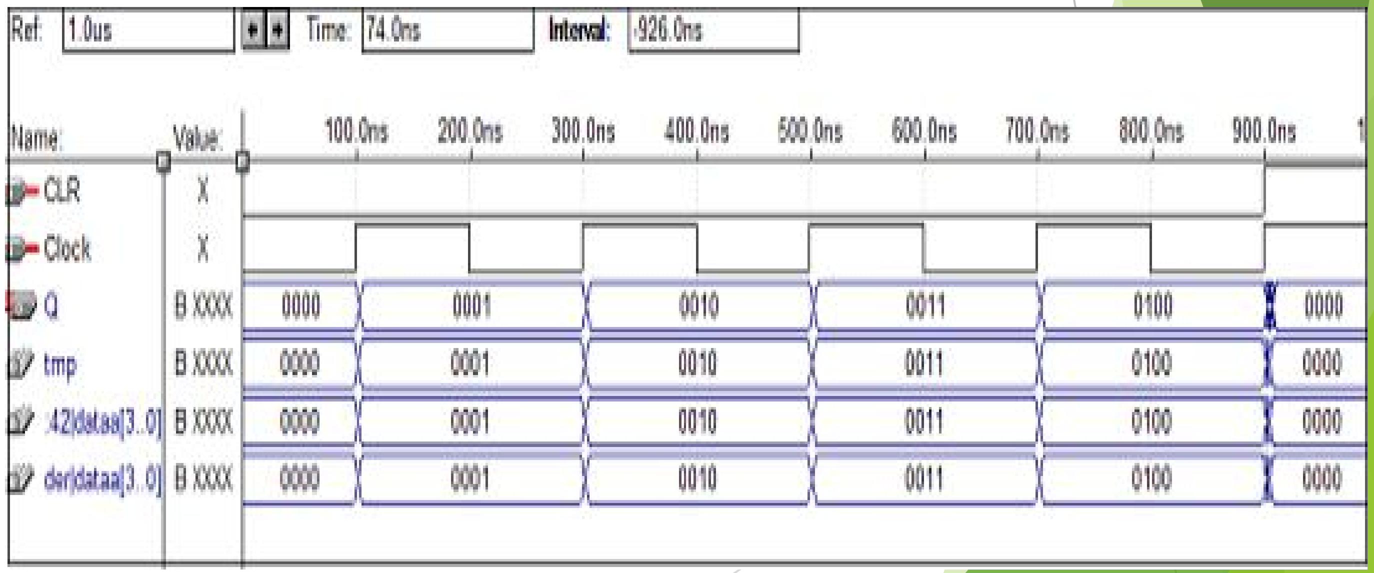
```
architecture virat of Toggle_flip_flop is
begin
   process (t,clk,reset)
  variable temp : std_logic ;

begin
    if (reset = '1') then
      Q : = '0';
    elsif (clk=‘1’and clk’event) then

        temp : = not t;
      end if;

    Q < = temp;
  end process tff;
end virat;
```



VHDL Code for a 4 - bit Up Counter

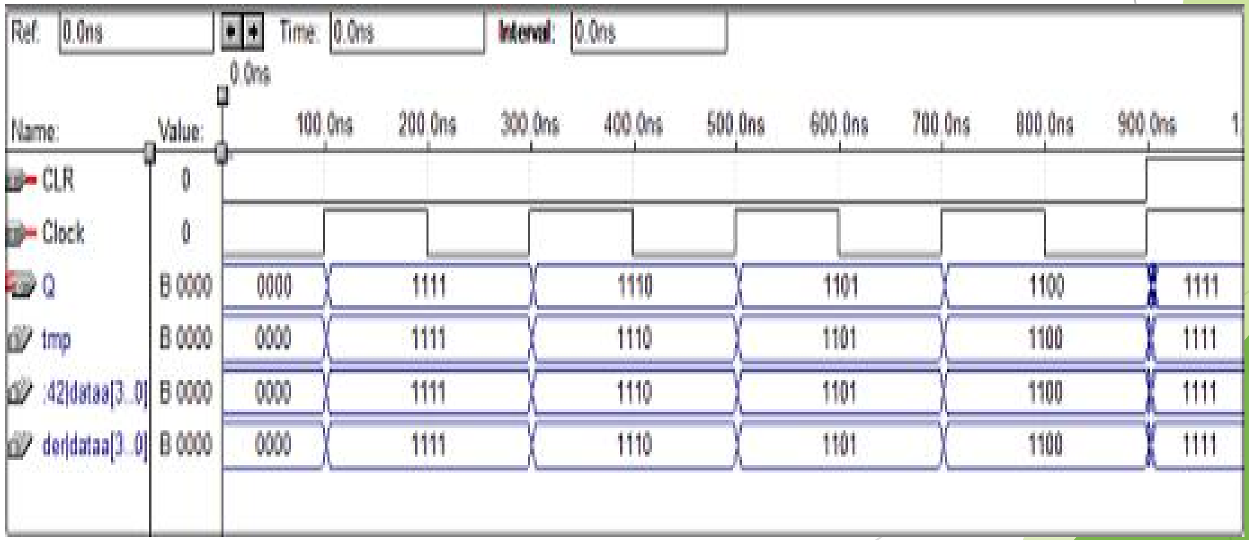
```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
  port(Clock, CLR : in std_logic;
    Q : out std_logic_vector(3 downto 0)
  );
end counter;

architecture virat of counter is
  signal tmp: std_logic_vector(3 downto 0);
begin
  process (Clock, CLR)

  begin
    if (CLR = '1') then
      tmp < = "0000";
    elsif (Clock'event and Clock = '1') then
      temp <= tmp + 1;
    end if;
  end process;
  Q <= tmp;
end virat;
```





VHDL Code for a 4-bit Down Counter

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity dcounter is
  port(Clock, CLR : in std_logic;
    Q : out std_logic_vector(3 downto 0));
end dcounter;

architecture virat of dcounter is
  signal tmp: std_logic_vector(3 downto 0);

begin
  process (Clock, CLR)
  begin
    if (CLR = '1') then
      tmp <= "1111";
    elsif (Clock'event and Clock = '1') then
      tmp <= tmp - 1;
    end if;
  end process;
  Q <= tmp;
end virat;
```



4 bit up down counter VHDL source code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter_VHDL is
port( d: in std_logic_vector(0 to 3);
Clock: in std_logic;
Load: in std_logic;
Reset: in std_logic;
Direction: in std_logic;
Output: out std_logic_vector(0 to 3) );
end Counter_VHDL;

architecture Behavioral of
Counter_VHDL is
signal temp: std_logic_vector(0 to 3);
begin
process(Clock,Reset)
begin
if Reset='1' then
temp <= "0000";
elsif ( Clock'event and Clock='1')
then
if Load='1' then
temp <= d;
elsif (Load='0' and Direction='0')
then
temp <= temp + 1;
elsif (Load='0' and Direction='1')
then
temp <= temp - 1;
end if;
end if;
end process;
Output <= temp;
end Behavioral;
```