Unit V - PRINCIPLES OF HDL
SYLLABUS

• Introduction to VHDL – Sequential and Concurrent descriptions. Signal, port and variable statements. Wait, case and other sequential statements. Block, process, component and generate descriptions. Test bench creation and principles of operation of VHDL simulator. Introduction to Verilog and brief comparison with VHDL
Course Objectives

- explain VHDL background and Design methodology based on VHDL.
- explain modeling of digital systems using VHDL.
- explain sequential and concurrent techniques in VHDL.
- explain data types, subprograms, packages, predefined attributes and configurations of VHDL.
- explain simulation and test bench creation.
- explain verilog (introduction) and comparison with VHDL.
Course outcome

- Understand Design Steps with VHDL
- Understand main programming technique with VHDL.
- Understand simulation techniques and test bench creation.
- Understand how to Design a simple digital circuit Using VHDL.
- Understand the basic concepts of verilog.
- Understand the prose and cons of VHDL and verilog.
VHDL

- What is VHDL?

\[ \text{VHIS C} \rightarrow \text{Very High Speed Integrated Circuit} \]

Hardware

Description
History of VHDL

- Designed by IBM, Texas Instruments, and Intermetrics as part of the DoD funded VHSIC program
- Standardized by the IEEE in 1987: IEEE 1076-1987
- Enhanced version of the language defined in 1993: IEEE 1076-1993
- Additional standardized packages provide definitions of data types and expressions of timing data
  - IEEE 1164 (data types)
  - IEEE 1076.3 (numeric)
  - IEEE 1076.4 (timing)
Traditional vs. Hardware Description Languages

- Procedural programming languages provide the *how* or recipes
  - for computation
  - for data manipulation
  - for execution on a specific hardware model
- Hardware description languages *describe* a system
  - Systems can be described from many different points of view
    - Behavior: what does it do?
    - Structure: what is it composed of?
    - Functional properties: how do I interface to it?
    - Physical properties: how fast is it?
Usage

- Descriptions can be at different levels of abstraction
  - Switch level: model switching behavior of transistors
  - Register transfer level: model combinational and sequential logic components
  - Instruction set architecture level: functional behavior of a microprocessor

- Descriptions can be used for
  - Simulation
    - Verification, performance evaluation
  - Synthesis
    - First step in hardware design
Why do we Describe Systems?

- **Design Specification**
  - unambiguous definition of components and interfaces in a large design

- **Design Simulation**
  - verify system/subsystem/chip performance prior to design implementation

- **Design Synthesis**
  - automated generation of a hardware design
Digital System Design Flow

- Design flows operate at multiple levels of abstraction
- Need a uniform description to translate between levels
- Increasing costs of design and fabrication necessitate greater reliance on automation via CAD tools
  - $5M - $100M to design new chips
  - Increasing time to market pressures
A Synthesis Design Flow

- Automation of design refinement steps
- Feedback for accurate simulation
- Example targets: ASICs, FPGAs

SRM University
The Role of Hardware Description Languages

- Design is structured around a hierarchy of representations
- HDLs can describe distinct aspects of a design at multiple levels of abstraction

[Gajski and Kuhn]
Alternatives

- The Verilog hardware description language
  - Finding increasing use in the commercial world
    - SystemVerilog gaining prominence
  - VHDL dominates the aerospace and defense worlds

- Programming language based design flows
  - SystemC
    - C++ with additional hardware-based language elements
  - C-based design flows
    - C + extensions as well as ANSI C based
  - Other
    - Java, MATLAB, and specialized languages
Role of VHDL

- System simulation
- System description and documentation
- System synthesis
Describing a design

In VHDL an entity is used to describe hardware module.

An entity can be described using,

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

- starts with ‘entity’ and ends with ‘end’ keywords.
- Ports are interfaces through which an entity can communicate with its environment.
- Each port must have a name, direction and a type. The direction will be input, output or inout.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>Port can be read</td>
</tr>
<tr>
<td>Out</td>
<td>Port can be written</td>
</tr>
<tr>
<td>Inout</td>
<td>Port can be read and written</td>
</tr>
<tr>
<td>Buffer</td>
<td>Port can be read and written, it can have only one source.</td>
</tr>
</tbody>
</table>
Architecture

- Describes the internal description of design or it tells what is there inside design.
- Each entity has at least one architecture and an entity can have many architecture.
- Architecture can be described using structural, dataflow, behavioral or mixed style.

Syntax:

```plaintext
architecture architecture_name of entity_name
architecture_declarative_part;
begin
  Statements;
end architecture_name;
```
The internal working of an entity can be defined using different modeling styles inside architecture body. They are

- Dataflow modeling.
- Behavioral modeling.
- Structural modeling.
Structure of an entity

Entity

Port declaration
(Entity declaration)

Internal working
(Architecture body)
(Dataflow, Behavioral, Structural or mixed)

Ports
Dataflow modeling

The internal working of an entity can be implemented using concurrent signal assignment. Half adder example

```vhdl
Library IEEE; use IEEE.STD_LOGIC_1164.all;
entity ha_en is
type bit is Boolean;
end ha_en;
architecture ha_ar of ha_en is
begin
S <= A xor B;
C <= A and B;
end ha_ar;
```
Behavioral modeling

The internal working of an entity can be implemented using set of statements. It contains:

- Process statements
- Sequential statements
- Signal assignment statements
- Wait statements

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

entity ha_beha_en is
  port(A : in BIT;  B : in BIT;  S : out BIT;  C : out BIT);
end ha_beha_en;

architecture ha_beha_ar of ha_beha_en is
begin
  process_beh: process(A,B)
  begin
    S<= A xor B;
    C<=A and B;
  end process process_beh;
end ha_beha_ar;
```
Structural modeling

The implementation of an entity is done through set of interconnected components. It contains:
- Signal declaration.
- Component instances
- Port maps.
- Wait statements.

Component declaration:
Syntax:
```
component component_name [is]
  List_of_interface ports;
end component component_name;
```

Before instantiating the component it should be declared using component declaration as shown above. Component Declaration declares the name of the entity and interface of a component.
library IEEE
use IEEE.STD_LOGIC_1164.all;
entity fa_en is
  port(A,B,Cin:in bit; SUM, CARRY:out bit);
end fa_en;
architecture fa_ar of fa_en is
component ha_en
  port(A,B:in bit; S,C:out bit);
end component;
signal C1,C2,S1:bit;
begin
  HA1:ha_en port map(A,B,S1,C1);
  HA2:ha_en port map(S1,Cin,SUM,C2);
  CARRY <= C1 or C2;
end fa_ar;
Component instantiation

Component_label: component_name port map (signal_list);

Signal_list is the architecture signals which we are connecting to component ports. (Above is positional binding - connecting signal and port )

Another method of binding is

HA1:ha_en port map(A => A, B => B, S => S1, C => C1);
HA2:ha_en port map(A => S1, B => Cin, S => SUM, C => C2);
Variables

What are they for:
   Local storage in processes, procedures, and functions

Declaring variables

```variable```
```list_of_variable_names : type_name```
```[ := initial value ];```

Variables must be declared within the process in which they are used and are local to the process.

Note: exception to this is SHARED variables
Signals

Signals must be declared outside a process

Declaration form

signal list_of_signal_names : type_name
[ := initial value ];

Declared in an architecture can be used anywhere within that architecture
architecture RTL of XYZ is
  signal A, B, C : integer range 0 to 7;
  signal Y, Z : integer range 0 to 15;
begin
  process (A, B, C)
    variable M, N : integer range 0 to 7;
  begin
    M := A;
    N := B;
    Z <= M + N;
    M := C;
    Y <= M + N;
  end process;
end RTL;
Ports

The Entity ("BLACK BOX") has PORTS

- PORTS are the points of communication
- PORTS are usually the device pin
- PORTS have an associated name, mode and type
Port Modes

A port’s **MODE** indicates the direction that data is transferred:

- **IN**  Data goes into the entity only
- **OUT** Data goes out of the entity only (and is not used internally)
- **INOUT** Data is bi-directional (goes into and out of the entity)
- **BUFFER** Data that goes out of the entity and is also fed-back internally
There are 3 varieties of operators: logical, relational, and arithmetic

- Logical operators are “and”, “or”, etc.
- Relational operators are used to compare different values
- Arithmetic operators are used for performing mathematics
VHDL Operators - Logical

- Includes AND, OR, NAND, NOR, and XOR
- All have the same precedence
- Execute from left to right
- NOT has a higher precedence, and therefore executes before other operators in an expression
- Logical operations can only be applied to arrays of the same type and length
- Matching elements in arrays is by POSITION!
VHDL Operators - Relational

- Relational operators return a boolean value.
- Most often used within if-then-else statements to control the flow of code depending on different conditions.

- `<` less than
- `>` greater than
- `=` equal to
- `<=` less than or equal to
- `>=` greater than or equal to
- `=/=` not equal to
The rules for using relational operators are that the operands must be of the same type.

For an array, the operands can be of different lengths; the operands are aligned to the left, and compared to the right.

This makes 111 greater than 1011!
VHDL Operators - Arithmetic

- Predefined for types integer, real (except for modulus and remainder), and type time.

- As vectors do not represent a numerical value, the arithmetic operators cannot be used with types bit_vector or std_logic_vector.
Legal VHDL Identifiers

- Letters, digits, and underscores only (first character must be a letter)
- The last character cannot be an underscore
- Two underscores in succession are not allowed
- Using reserved words is not allowed (the VHDL editor will highlight reserved words for this reason)

Examples

- Legal
  - tx_clk, Three_State_Enable, sel7D, HIT_1124
- Not Legal
  - _tx_clk, 8B10B, large#num, case, clk_
Predefined VHDL Types

- Variables, signals, and constants can have any one of the predefined VHDL types or they can have a user-defined type

- Predefined Types
  - bit – \{'0', '1'\}
  - boolean – \{TRUE, FALSE\}
  - integer – $[-2^{31} \ldots 2^{31} - 1]$ 
  - real – floating point number in range $-1.0 \times 10^{38}$ to $+1.0 \times 10^{38}$
  - character – legal VHDL characters including lowercase, uppercase letters, digits, special characters, ...
  - time – an integer with units fs, ps, ns, us, ms, sec, min, or hr
**User Defined Type**

Common user-defined type is enumerated

```vhdl
type state_type is (S0, S1, S2, S3, S4, S5);
signal state : state_type := S1;
```

- If no initialization, the default initialization is the leftmost element in the enumeration list (S0 in this example)
- VHDL is strongly typed language => signals and variables of different types cannot be mixed in the same assignment statement, and no automatic type conversion is performed
Arrays

Example

type SHORT_WORD is array (15 downto 0) of bit;
signal DATA_WORD : SHORT_WORD;
variable ALT_WORD : SHORT_WORD := "0101010101010101";
constant ONE_WORD : SHORT_WORD := (others => '1');

- ALT_WORD(0) – rightmost bit
- ALT_WORD(5 downto 0) – low order 6 bits

General form

type arrayTypeName is array index_range of
element_type;
signal arrayName : arrayTypeName [:=InitialValues];
VHDL Statements

- There are two types of statements, **Concurrent** and **Sequential**

- **Concurrent Statements** (means in parallel)
  - Concurrent statements are “executed” concurrently (at the same time)
  - The order of concurrent statements is **not** important
VHDL Statements (cont.)

- **Sequential Statements** (means in series)
  - Sometimes we need to model complex functions. In that case, we can use an “algorithm” or model to describe the function. This is done with Sequential Statements.

- With Sequential statements, the ORDER of the statements is important (example later).

- Therefore, we use a **process** to mark the beginning and end of a block of sequential statements.

- Each completed process is considered to be one big concurrent statement (there can be many processes inside one architecture).
Sequential Statements

- 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
CASE selection_signal IS
  WHEN value_1_of_selection_signal =>
    (do something) -- set of statements 1
  WHEN value_2_of_selection_signal =>
    (do something) -- set of statements 2
  ...
  WHEN value_N_of_selection_signal =>
    (do something) -- set of statements N
  WHEN OTHERS =>
    (do something) -- default action
END CASE ;
ARCHITECTURE archdesign OF design IS
SIGNAL s: std_logic_vector(0 TO 1);
BEGIN
mux4_1: PROCESS (a,b,c,d,s)
BEGIN
CASE s IS
WHEN "00"  => x <= a;
WHEN "01"  => x <= b;
WHEN "10"  => x <= c;
WHEN OTHERS => x <= d;
END CASE;
END PROCESS mux4_1;
END archdesign;
Wait Statements

- an alternative to a sensitivity list
- a process cannot have both wait st. and a sensitivity list
- Generic form of a process with wait statement(s)

**Process**

```plaintext
begin
    sequential-statements
    wait statement
    sequential-statements
    wait-statement
    ...
end process;
```

- How wait statements work?
  Execute seq. statement until wait statement is encountered.
  Wait until the specified condition is satisfied.
  When the end of the process is reached start over again at the beginning.
Forms of Wait Statements

- **wait on** sensitivity-list;
  - until one of the signals in the sensitivity list changes

- **wait for** time-expression;
  - waits until the time specified by the time expression has elapsed
  - What is this:
    wait for 0 ns;

- **wait until** boolean-expression;
  - the boolean expression is evaluated whenever one of the signals in the expression changes, and the process continues execution when the expression evaluates to TRUE
library BITLIB;
use BITLIB.Bit_pack.all;
entity SM1_2 is port(X, CLK: in bit; Z: out bit); end SM1_2;
architecture Table of SM1_2 is signal State, Nextstate: integer := 0;
begin
  process
  begin
  case State is
    when 0 =>
      if X='0' then Z<='1'; Nextstate<=1; end if;
      if X='1' then Z<='0'; Nextstate<=2; end if;
    when 1 =>
      if X='0' then Z<='1'; Nextstate<=3; end if;
      if X='1' then Z<='0'; Nextstate<=4; end if;
    when 2 =>
      if X='0' then Z<='0'; Nextstate<=4; end if;
      if X='1' then Z<='1'; Nextstate<=4; end if;
    when 3 =>
      if X='0' then Z<='0'; Nextstate<=5; end if;
      if X='1' then Z<='1'; Nextstate<=5; end if;
    when 4 =>
      if X='0' then Z<='1'; Nextstate<=5; end if;
      if X='1' then Z<='0'; Nextstate<=6; end if;
    when 5 =>
      if X='0' then Z<='0'; Nextstate<=0; end if;
      if X='1' then Z<='1'; Nextstate<=0; end if;
    when 6 =>
      if X='0' then Z<='1'; Nextstate<=0; end if;
    when others => null; -- should not occur
  end case;
  wait on CLK, X;
  if rising_edge(CLK) then -- rising_edge function is in BITLIB
    State <= Nextstate;
    wait for 0 ns; -- wait for State to be updated
  end if;
  end process;
end table;
**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;```
Procedure call statement

- Call a procedure.

```plaintext
[ 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 then else

- Used to select a set of statements to be executed
- Selection based on a boolean evaluation of a condition or set of conditions. Absence of ELSE results in implicit memory

```
IF condition(s) THEN
    do something;
ELSIF condition_2 THEN  -- optional
    do something different;
ELSE                      -- optional
    do something completely different;
END IF;
```
If then else example

4-1 mux shown below
mux4_1: process (a, b, c, d, s)
    begin
        if  s = “00″ then x <= a ;
            elsif  s = “01″ then x <= b ;
            elsif  s = “10″ then x <= c ;
            else x <= d ;
        end if;
    end process mux4_1 ;
CASE selection_signal IS
    WHEN value_1_of_selection_signal  =>
      (do something) -- set of statements 1
    WHEN value_2_of_selection_signal  =>
      (do something) -- set of statements 2
    ...
    WHEN value_N_of_selection_signal  =>
      (do something) -- set of statements N
    WHEN OTHERS  =>
      (do something) -- default action
END CASE;
ARCHITECTURE archdesign OF design IS
  SIGNAL s: std_logic_vector(0 TO 1);
BEGIN
  mux4_1: PROCESS (a,b,c,d,s)
  BEGIN
    CASE s IS
      WHEN "00"  => x <= a;
      WHEN "01"  => x <= b;
      WHEN "10"  => x <= c;
      WHEN OTHERS => x <= d;
      END CASE;
  END PROCESS mux4_1;
END archdesign;
loop statement

The loop label is optional. By defining the range the direction as well as the possible values of the loop variable are fixed. The loop variable is only accessible within the loop. For synthesis the loop range has to be locally static and must not depend on signal or variable values. While loops are not generally synthesizable.

Three kinds of iteration statements.

[ 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 ] ;
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;
Concurrent Statements

- block statement
- process statement
- concurrent procedure call statement
- concurrent assertion statement
- concurrent signal assignment statement
- conditional signal assignment statement
- selected signal assignment statement
- component instantiation statement
- generate statement
Block Statements

- A block statement provides a way to combine a group of concurrent statements together.
- A group of statements can be placed under a guard.
- FORMAT
  
  label: block (guard expression)
  -- declarative part
  begin
  -- statement part
  end block label
- A guard is a boolean expression that evaluates to true or false.
- Concurrent statements in block execute if guard is true.
A guarded assignment statement executes if either
- (1) the guard expression changes from FALSE to TRUE
- (2) The guard expression is TRUE and one of the signals appearing on the right hand side of the signal assignment changes value

Example:
B1 : block (CONTROL_SIGNAL = ‘1’)
begin
   X <= guarded A or B after 5 min;
   Y <= A or B after 5 min;
end block B1
Process

- Processes are either awake or asleep (active or inactive)
- A process normally has a sensitivity list
  - When a signal in that sensitivity list changes value, the process wakes up and all of the sequential statements are “executed”
  - For example, a process with a clock signal in its sensitivity list will become active on changes of the clock signal
  - At the end of the process, all outputs are assigned and the process goes back to sleep until the next time a signal changes in the sensitivity list.

```plaintext
label: PROCESS (sensitivity list)
    -- variable declarations
BEGIN
    -- sequential statements
END PROCESS label ;
```

- The process label and variable declarations are optional.
- The process executes when one of the signals in the sensitivity list has an event.
D-Flipflop:

entity DFF is
    -- Signals are initialized to 0 by default.
    -- To make QN a 1, it has to be initialized
    port ( D, CLK : in std_logic;
           Q : out std_logic;
           QN : out std_logic := '1');
end DFF;
architecture data_flip of DFF is
begin
    process ( CLK )
    begin
        if (CLK = '1' and CLK'event ) then
            Q <= D after 10ns;
            QN <= not D after 10ns;
        end if;
    end process;
end data_flip;
Selective Signal Assignment: with-select-when

- Assignment based on a selection signal
- WHEN clauses must be mutually exclusive
- Use a WHEN OTHERS when all conditions are not specified
- Only one reference to the signal, only one assignment operator (<=)

```
WITH selection_signal SELECT
  signal_name <= value_1 WHEN value_1 of selection_signal,
                 value_2 WHEN value_2 of selection_signal,
                 ...
                 value_n WHEN value_n of selection_signal,
                 value_x WHEN OTHERS;
```
Conditional Signal Assignment: when-else

- Signal is assigned a value based on conditions
- Any simple expression can be a condition
- Priority goes in order of appearance
- Only one reference to the signal, only one assignment operator (<=)
- Use a final ELSE to avoid latches

```vhdl
signal_name <= value_1 WHEN condition1 ELSE value_2 WHEN condition2 ELSE ...
                value_n WHEN condition N ELSE value_x ;
```
The 4-1 multiplexer is shown below

```
x <= a when (s = "00") else
    b when (s = "01") else
    c when (s = "10") else
    d ;
```
Selective Signal Assignment

The 4-1 multiplexer is shown below

with s select

\[ x \leq a \text{ when } "00", \]
\[ b \text{ when } "01", \]
\[ c \text{ when } "10", \]
\[ d \text{ when others}; \]
The Generate Statement

- The generate statement
  - A parameterized approach to describing the regular interconnection of components

a: for i in 1 to 6 generate
  a1: one_bit generic map (gate_delay)
  port map(in1=>in1(i), in2=> in2(i), cin=>carry_vector(i-1),
  result=>result(i), cout=>carry_vector(i),opcode=>opcode);
end generate;
The Generate Statement: Example

- Instantiating an register

```vhdl
entity dregister is
    port ( d : in std_logic_vector(7 downto 0);
          q : out std_logic_vector(7 downto 0);
          clk : in std_logic);
end entity dregisters;

architecture behavioral of dregister is
begin
    d: for i in dreg’range generate
        reg: dff port map( (d=>d(i), q=>q(i), clk=>clk;
    end generate;
end architecture register;
```

- Instantiating interconnected components

Declare local signals used for the interconnect
Component Instantiation Statement

The syntax for the components instantiation is as follows,

instance_name : component name
port map (port1=>signal1, port2=> signal2,... port3=>signaln);

cOMPONENT NAND2
    port (in1, in2: in std_logic;
          out1: out std_logic);
END COMPONENT;

signal int1, int2, int3: std_logic;

architecture struct of EXAMPLE is
    U1: NAND2 port map (A,B,int1);
    U2: NAND2 port map (in2=>C, in2=>D, out1=>int2);
    U3: NAND3 port map (in1=>int1, int2, Z);  .....
VHDL Subprograms (Function & procedure)

- Classes of subprograms
  - Function
    - Computes and returns a value
    - Does not modify any arguments
    - Used only in expressions
  - Procedure
    - May modify its arguments
    - Does not return a value
    - Sequential or Concurrent Statement
Formals vs. Actual in Subprograms

- **Formals:**
  - part of function declaration
  - placeholders which describe type of object to be passed to function

- **Actual:**
  - objects which are passed into a function in specific instantiations
  - Formal types must match actual types.
Properties of Modes

- Parameters declared as IN can only be read:
- Parameters declared as OUT can only be written to.
- Parameters declared as INOUT can be read and assigned new values.
- Default mode is IN.
Constraints on Declarations of Parameters

for Functions

- **Allowed Modes**
  - In

- **Allowed Object Classes**
  - Constants
  - Signals

for procedure

- **Allowed Modes**
  - in
  - Out
  - Inout

- **Allowed Object Classes**
  - constant
  - variable
  - signal
Functions

To declare a function in VHDL, specify:

- the name of the function
- the input parameters (if any) (formal parameters)
- the type of the returned value
- any declarations required by the function
- an algorithm for the computation of the returned value

Declare in architecture declaration section:

architecture WITHFUNCTION of WHAT is
  <declare function(s) here>
begin
  <call functions here>
end;
Structure for Function Body Specification

function function_name ( function_formal_parameter_list )
return return_type is
  function_declaration_part
begin
  sequential_statements
return ( return_value );
end function_name;

- CONSTRAINT: All parameters in the function formal parameter list must have mode IN.
- No signals allowed in the function declaration part.
- No WAIT statements allowed either directly or indirectly.
Function Example

Type conversion: std_logic_vector to Integer:

```vhdl
function BV2INT (BVIN: std_logic_vector) -- Unconstrained
return Integer is
  variable BV_INT: std_logic_vector (BVIN'length –1 downto 0);
  variable TEMP: Integer := 0;
begin
  BV_INT := BVIN;
  for i in 0 to BVIN'length-1 loop
    if BV_INT(i) = '1' then
      TEMP := TEMP + 2**i;
    end if;
  end loop;
  return TEMP;
end BV2INT;
```

Variables in functions are dynamic.
function DECODE3TO8 ( V: std_logic_vector (2 downto 0))
  return std_logic_vector is
  variable RESULT:std_logic_vector(7 downto 0) := "00000001";
begin
  if v(0) = '1' then   RESULT := RESULT(6 downto 0) & '0';
  end if;
  if v(1) = '1' then   RESULT := RESULT(5 downto 0) & "00";
  end if;
  if v(2) = '1' then   RESULT := RESULT(3 downto 0) & "0000";
  end if;
return RESULT;
end DECODE3TO8;
Procedures

- PROCEDURES are subprograms that can modify one or more of the formal parameters.
- Parameters may be of mode IN, OUT or INOUT.
- If the mode is not specified, it is assumed to be IN.
- If the class of a parameter is not explicitly declared, it is assumed that:
  - INs are assumed to be of class CONSTANT
  - OUTs and INOUTs are assumed to be VARIABLE
- As with functions, type of formal in declaration must match type of actual when called.
- Variables declared within a procedure are initialized on each procedure call and values do not persist across invocations of the procedure.
- Signals cannot be declared in procedures.
- Ports are visible within procedures.
- Can have WAITs in procedure.
- Procedures can be called as concurrent statements or as sequential statements within a process.
Procedures

To declare a PROCEDURE in VHDL, specify:

- the name of the procedure
- the input and output parameters (if any)
- any declarations required by the procedure itself
- an algorithm

Structure for Procedure Body Specification

```
procedure procedure_name ( formal_parameter_list ) is
    procedure_declaration_part
begin
    sequential_statements
end procedure_name;
```

- Parameters in the procedure formal parameter list may have mode IN, OUT, or INOUT.
- No signals allowed in the procedure declaration part.
- Concurrent procedures are executed once at the beginning of simulation and thereafter at any time that there is an event on any parameter of mode in or inout.
procedure ARITH_UNIT ( A, B : in integer; op : in op_code; Z : out integer; ZCOMP : out boolean ) is
begin
  case op is
    when ADD => Z := A + B;
    when SUB => Z := A - B;
    when MUL => Z := A * B;
    when DIV => Z := A / B;
    when LT => ZCOMP := A < B;
    when LE => ZCOMP := A <= B;
    when EQ => ZCOMP := A = B;
    when others => Z := Z;
  end case;
end ARITH_UNIT;
Test Bench

- Test bench is a module that is used for testing the functionality of a design module by simulation.

- Entity declarations of test bench modules have no input and output ports.

- Using a test bench model an architecture body that includes an instance of the design under test applies sequences of test values to inputs monitors values on output signals either using simulator or with a process that verifies correct operation.
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity tb_en is
end tb_en;

architecture tb_ar of tb_en is
signal a_i,b_i,c_i,sum_i,carry_i:bit;
begin
eut: entity work.fa_en(fa_ar)
port map(A=>a_i,B=>b_i,Cin=>c_i,SUM=>sum_i,CARRY=>carry_i);
stimulus: process
begin
a_i<='1';b_i<='1';c_i<='1';
wait for 10ns;
a_i<='0';b_i<='1';c_i<='1';
wait for 10ns;
a_i<='1';b_i<='0';c_i<='0';
wait for 10ns;
if now=30ns then
wait;
end if;
end process stimulus;
end tb_ar;
Once we have a VHDL program whose syntax and semantics are correct, a simulator can be used to observe its operation.

Simulator operation begins at simulation time of zero.

At this time, the simulator initialises all signals to a default value.

It also initialises any signals and variables for which initial values have been explicitly declared.

Next, the simulator begins the execution of all processes (and concurrent statements) in the design.

The simulator uses a time-based event list and a signal-sensitivity matrix to simulate the execution of all the processes.
VHDL Simulation (2)

- At simulation time zero, all processes are scheduled for execution.
- One of them is selected and all of its sequential statements are executed, including any looping behaviour that is specified.
- When the execution of this process is completed, another one is selected, and so on, until all processes have been executed.
- This completes one simulation cycle.
- During its execution, a process may assign new values to signals.
- The new values are not assigned immediately. They are placed on the event list and scheduled to become effective at a certain time.
If the assignment has an explicit simulation time (after clause), then it is scheduled on the event list to occur at that time.

Otherwise, it is supposed to occur “immediately”.

It is actually scheduled to occur at the current simulation time plus one delta delay.

The delta delay is an infinitesimally short time, such that the current simulation time plus any number of delta delays still equals the current simulation time.

The delta delay concept allows processes to execute multiple times (if necessary) in zero simulated time.

After a simulation cycle completes, the event list is scanned for the signals that change at the next earliest time on the list.
This may be as little as one delta delay, or it may be a real delay, in which case the simulation time is advanced.

In any case, the scheduled signal changes are made.

Some processes may be sensitive to the changing signals.

All the processes that are sensitive to a signal that just changed are scheduled for execution in the next simulation cycle (begins now).

The simulator’s operation goes on indefinitely until the list is empty.

The event list mechanism makes it possible to simulate the operation of concurrent processes in a uni-processor system.

The delta delay mechanism ensures correct operation even though a set of processes may require multiple executions.
Introduction to Verilog

A Hardware Description Language
- VHDL, ABEL, Verilog
- Pascal or C-like syntax

Open standard
- introduced in 1985 by Gateway Design System, now owned by Cadence

Advantages of Verilog HDL
- allows hardware designer to describe designs at various levels of abstraction
- behavioral vs. structural or lower-level implementations
- allows simulation of designs before fabrication
- can be synthesized to actual hardware
## VHDL & VERILOG

<table>
<thead>
<tr>
<th></th>
<th>VHDL</th>
<th>Verilog HDL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td>Public (IEEE Standard 1076)</td>
<td>Public (IEEE Standard 1364)</td>
</tr>
<tr>
<td><strong>Levels of modeling (mixing allowed?)</strong></td>
<td>system, functional, structural, RTL, gate (yes)</td>
<td>functional, structural, RTL, gate, switching (yes)</td>
</tr>
<tr>
<td><strong>Looks like</strong></td>
<td>Ada</td>
<td>c</td>
</tr>
<tr>
<td><strong>Level of complexity</strong></td>
<td>difficult</td>
<td>average</td>
</tr>
<tr>
<td><strong>Easy to read</strong></td>
<td>no</td>
<td>average</td>
</tr>
<tr>
<td><strong>Predefined features</strong></td>
<td>poor</td>
<td>good</td>
</tr>
<tr>
<td><strong>Designer friendly</strong></td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Designers Acceptance</strong></td>
<td>widely used</td>
<td>widely used</td>
</tr>
<tr>
<td><strong>Tool independent</strong></td>
<td>yes</td>
<td>more or less</td>
</tr>
<tr>
<td><strong>Straightforward hardware meaning</strong></td>
<td>not really</td>
<td>partially</td>
</tr>
<tr>
<td><strong>Clock model</strong></td>
<td>asynchronous, multi-phase, multiple clocks</td>
<td>asynchronous, multi-phase, multiple clocks</td>
</tr>
<tr>
<td><strong>Timing properties</strong></td>
<td>sequential and concurrent</td>
<td>sequential and concurrent</td>
</tr>
<tr>
<td><strong>Finite state machine</strong></td>
<td>implicit, verbose</td>
<td>implicit, verbose</td>
</tr>
<tr>
<td><strong>Semantics for simulation (speed)</strong></td>
<td>yes (average)</td>
<td>yes (fast)</td>
</tr>
<tr>
<td><strong>Semantics for synthesis</strong></td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Verilog Module

In Verilog, a circuit is a module.

```verilog
module decoder_2_to_4 (A, D) ;
input [1:0] A ;
output [3:0] D ;
assign D = (A == 2'b00) ? 4'b0001 :
           (A == 2'b01) ? 4'b0010 :
           (A == 2'b10) ? 4'b0100 :
           (A == 2'b11) ? 4'b1000 ;
endmodule
```
module decoder_2_to_4 (A, D) ;

input [1:0] A ;
output [3:0] D ;

assign D = (A == 2'b00) ? 4'b0001 :
            (A == 2'b01) ? 4'b0010 :
            (A == 2'b10) ? 4'b0100 :
            (A == 2'b11) ? 4'b1000 ;

endmodule
Declaring A Module

- Can’t use keywords as module/port/signal names
  - Choose a descriptive module name

- Indicate the ports (connectivity)

- Declare the signals connected to the ports
  - Choose descriptive signal names

- Declare any internal signals

- Write the internals of the module (functionality)
Declaring Ports

- A signal is attached to every port
- Declare type of port
  - input
  - output
  - inout (bidirectional)
- Scalar (single bit) - don’t specify a size
  - input cin;
- Vector (multiple bits) - specify size using range
  - Range is MSB to LSB (left to right)
  - Don’t have to include zero if you don’t want to...
    (D[2:1])
  - output OUT [7:0];
  - input IN [0:4];
Module Styles

- Modules can be specified different ways
  - Structural – connect primitives and modules
  - RTL – use continuous assignments
  - Behavioral – use initial and always blocks

- A single module can use more than one method
Structural

- A schematic in text form
- Build up a circuit from gates/flip-flops
  - Flip-flops themselves described behaviorally

Structural design

- Create module interface
- Instantiate the gates in the circuit
- Declare the internal wires needed to connect gates
- Put the names of the wires in the correct port locations of the gates

- For primitives, outputs always come first
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

wire N1, N2, N3;

and A0 (N1, V1, V2),
A1 (N2, V2, V3),
A2 (N3, V3, V1);

or Or0(major, N1, N2, N3);

endmodule
module majority (major, V1, V2, V3) ;
output major ;
input V1, V2, V3 ;
assign major = V1 & V2 | V2 & V3 | V1 & V3;
endmodule
module majority (major, V1, V2, V3) ;
output reg major ;
input V1, V2, V3 ;
always @(V1, V2, V3) begin
    if (V1 && V2 || V2 && V3 || V1 && V3) major = 1;
    else major = 0;
end
endmodule
QUESTIONS

PART - A

1. What are the important features of VHDL?
2. Differentiate a signal and variable?
3. What are the different types of modeling in VHDL?
4. Explain ‘case’ statement in VHDL with an Example.
5. Explain ‘BLOCK’ statement in VHDL with an Example.
7. Explain ‘Generate’ statement in VHDL with an Example.
8. What is Test Bench?
10. Give the behavioral model for JK flipflop.
11. Give the behavioral model for D & T flipflop.
12. Give the data flow model for half adder and half subtractor.
13. Give the dataflow model for full adder.
14. Give the dataflow model for full subtractor.
15. Write short notes on delays in VHDL.
16. What is component instantiation?
17. Differentiate sequential from concurrent signal assignment statements.
18. Write short notes on “Wait” statement.

Part – B
1. Explain the various modeling methods used in VHDL with an example.
2. Explain in detail about the principal of operation of VHDL Simulator.
3. Write the VHDL program for 4 bit counter.
4. Write the VHDL program for full adder in all three types of modeling?
5. Write VHDL program for 4:1 MUX and 1:4 DEMUX using behavioral modeling.
6. Write VHDL program for encoder and decoder using structural modeling.
7. With an example explain in detail the test bench creation.
8. Write a verilog program for 1) Full Adder 2) Shift Register.