Compact Summary of VHDL

Table of contents

1 Compact Summary of VHDL

Version 1.5, please report any issues to squire@umbc.edu.

From http://www.csee.umbc.edu/portal/help/VHDL/summary_one.html

About VHDL VHDL is case insensitive, upper case letters are equivalent to lower case letters. Reserved words are in lower case by convention and shown in bold in this document.

Identifiers are simple names starting with a letter and may have letters and digits. The underscore character is allowed but not as the first or last character of an identifier.

A comment starts with minus minus, "–", and continues to the end of the line.

Indentation is used for human readability. The language is free form with the characters space, tab and new-line being "white space."

Contents

Notation used in this Compact Summary

```
Each item has:
    a very brief explanation of possible use.
    a representative, possibly not complete, syntax schema
    one or more samples of actual VHDL code.

In the syntax statement  [ stuff ]  means zero or one copy of "stuff".
In some cases "optional" is used to not clutter up the syntax with [[][]].

In the examples, assume the appropriate declarations for identifiers,
appropriate enclosing design unit and appropriate context clauses.
```

1 VHDL Design Units and Subprograms

A design unit may be the entire file or there may be more than one design unit in a file. No less than a design unit may be in a file.

Any design unit may contain a context clause as its initial part. The context clause of a primary unit applies to all of the primary units corresponding secondary units. Architectures and package bodies are the secondary units. Subprograms are not library units and must be inside entities, architectures or packages.

The analysis, compilation, of a design unit results in a library unit is some design library. Predefined libraries typically include but are not limited to: STD, IEEE and WORK. WORK is the default user library.

1 Design Units and Subprograms

- Entity
- Architecture
- Configuration
- Package Declaration
- Package Body
- Subprograms

- Procedure Declaration
- Procedure Body
- Function Declaration
- Function Body
- Context Clause
- Order of Analysis, Compilation

1.1.1.1 Entity The top of every design hierarchy must be an entity. Entities may range from primitive circuits to complex assemblies.

The entity code typically defines just the interface of the entity.

```
entity  identifier is
    generic ( generic_variable_declarations ) ; -- optional
    port ( input_and_output_variable_declarations ) ;
    [ declarations , see allowed list below ]   -- optional
begin                                         \__ optional
    [ statements , see allowed list below ]    /
end entity identifier ;
```

generic*variable*declarations are of the form: variable*name : variable*type := variable*value ; – := variable*value optional

input*and*output*variable*declaration are of the form: variable*name : port*mode variable*type ; port*mode may be in out inout buffer linkage

entity adder is generic (N : natural := 32) ; port (A : in bit*vector(N-1 downto 0); B : in bit*vector(N-1 downto 0); cin : in bit; Sum : out bit_vector(N-1 downto 0); Cout : out bit); end entity adder ;

entity test*bench is – typical top level, simulatable, entity end entity test*bench;

entity Latch is port (Din: in Word; Dout: out Word; Load: in Bit; Clk: in Bit); constant Setup: Time := 12 ns; constant PulseWidth: Time := 50 ns; use WORK.TimingMonitors.all begin assert Clk=‘1’ or Clk’Delayed’Stable(PulseWidth); CheckTiming(Setup, Din, Load, Clk); – passive concurrent procedure end entity Latch;

The allowed declarations are: subprogram declaration subprogram body type declaration subtype declaration constant, object declaration signal, object declaration variable, object declaration - shared file, object declaration alias declaration attribute declaration attribute specification disconnection specification use clause group template declaration group declaration

The allowed statements are: concurrent assertion statements passive concurrent procedure call passive process statement

Architecture

Used to implement a design entity. There may be more than one architecture for a design entity. Typical architectures fall into classes such as functional simulation or detailed logic implementation and may be structural, functional(dataflow) or behavioral.

architecture identifier of entity_name is [declarations , see allowed list below] begin – optional [statements , see allowed list below] end architecture identifier ;

architecture circuits of add4c is signal c : std*logic*vector(3 downto 0); component fadd – duplicates entity port port(a : in std*logic; b : in std*logic; cin : in std*logic; s : out std*logic; cout : out std_logic); end component fadd; begin – circuits of add4c a0: fadd port map(a(0), b(0), cin , sum(0), c(0)); a1: fadd port map(a(1), b(1), c(0), sum(1), c(1)); a2: fadd port map(a(2), b(2), c(1), sum(2), c(2)); a3: fadd port map(a(3), b(3), c(2), sum(3), c(3)); cout <= (a(3) and b(3)) or ((a(3) or b(3)) and ((a(2) and b(2)) or ((a(2) or b(2)) and ((a(1) and b(1)) or ((a(1) or b(1)) and ((a(0) and b(0)) or ((a(0) or b(0)) and cin))))))) after 1 ns; end architecture circuits; – of add4c

The allowed declarations are: subprogram declaration subprogram body 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 disconnection specification use clause group template declaration group declaration

The allowed statements are: concurrent statements

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 , see allowed list below][block configuration , see allowed list below] end architecture identifier ;

– entities and architecture circuits for fadd, add4c and add32 not shown entity add32*test is – test bench end add32*test;

architecture circuits of add32*test is – details implementing test bench deleted end architecture circuits; – of add32*test

configuration add32*test*config of add32*test is for circuits – of add32*test for all: add32 use entity WORK.add32(circuits); for circuits – of add32 for all: add4c use entity WORK.add4c(circuits); for circuits – of add4c for all: fadd use entity WORK.fadd(circuits); end for; end for; end for; end for; end for; end for; end configuration add32*test*config;

Note the architecture name in parenthesis following the entity name.

Or an equivalent configuration of configurations:

configuration add32*test*config of add32*test is for circuits – of add32*test for all: add32 use configuration WORK.add32*config; end for; end for; end configuration add32*test_config;

The allowed declarations are: attribute specification use clause group declaration

The allowed block configurations are: for component*instance*name : component_name – use clause end for;

for all : component_name – use clause end for;

use clauses are of the form: use entity library*name.entity*name[(architecture_name)] use configuration library*name.configuration*name

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

Used to declare the calling interface to a procedure.

procedure identifier [(formal parameter list)] ;

procedure print*header ; procedure build (A : in constant integer; B : inout signal bit*vector; C : out variable real; D : file) ;

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 procedure. The type definitions used in formal parameters must be visible at the place where the procedure is being declared. No semicolon follows the last formal parameter inside the parenthesis.

Formal parameters may be constants, variables, signals or files. The default is variable.

Formal parameters may have modes in, inout and out Files do not have a mode. The default is in .

If no type is given and a mode of in is used, constant is the default.

The equivalent default declaration of "build" is

procedure build (A : in integer; B : inout signal bit_vector; C : out real; D : file) ;

Procedure Body

Used to define the implementation of the procedure.

procedure identifier [(formal parameter list)] is [declarations, see allowed list below] begin sequential statement(s) end procedure identifier ;

procedure print*header is use STD.textio.all; variable my*line : line; begin write (my*line, string'("A B C")); writeline (output, my*line); end procedure print_header ;

The procedure body formal parameter list is defined above in Procedure Declaration. When a procedure declaration is used then the corresponding procedure 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

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

Context Clause

Used to name a library and make library units visible to the design unit that immediately follows.

library library*name ; use library*name.unit_name.all ;

library STD ; use STD.textio.all;

library ieee ; use ieee.std*logic*1164.all; use ieee.std*logic*textio.all; use ieee.std*logic*arith.all; use ieee.numeric*std.all; use ieee.numeric*bit.all; use WORK.my*pkg.s*inc; – select one item from package

Note that the .all makes everything visible. It is optional and when not used the prefix such as ieee.std*logic*1164. must be used on every reference to an item in the library unit. Specific items in the library unit may be listed in place of .all .

The libraries STD and WORK do not need a library specification on most systems. library ieee or equivalent library IEEE is needed on most systems.

Order of Analysis, Compilation

Every design unit must be analyzed, compiled, before it can be used by another design unit. The result of the analysis or compilation results in an analyzed design in a library. The analyzed design goes into the default library WORK unless otherwise specified.

An entity must be analyzed, compiled, before its corresponding architectures or configurations.

A package declaration must be analyzed, compiled, before its corresponding package body.

A package declaration must be analyzed, compiled, before it can be referenced in a context clause. For example:

```
Analyze, compile
     package my_package is
       -- declarations
     end package my_package;

then analyze, compile
     library WORK;  -- this line usually not needed
     use WORK.my_package.all
     entity my_entity is
       -- entity stuff
     end entity my_entity;
```

1 VHDL Sequential Statements

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

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 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 ja, 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 sequence-of-statements when choice2 => _ optional sequence-of-statements / . . .

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

case my*val is when 1 => a:=b; when 3 => c:=d; do*it; when others => null; end case;

loop statement

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] ;

loop input*something; exit when end*file; end loop;

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;

1 VHDL Concurrent Statements

These statements are for use in Architectures.

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 statement

Used to group concurrent statements, possibly hierarchically.

label : block [(guard expression)] [is] [generic clause [generic map aspect ;]][port clause [port map aspect ;]] [block declarative items] begin concurrent statements end block [label] ;

clump : block begin A <= B or C; D <= B and not C; end block clump ;

maybe : block (B'stable(5 ns)) is port (A, B, C : inout std*logic); port map (A => S1, B => S2, C => outp); constant delay: time := 2 ns; signal temp: std*logic; begin temp <= A xor B after delay; C <= temp nor B; end block maybe;

process statement

Used to do have sequential statements be a part of concurrent processing.

label : process [(sensitivity_list)] [is] [process*declarative*items] begin sequential statements end process [label] ;

```
    -- input and output are defined a type 'word' signals
```

reg*32: process(clk, clear) begin if clear='1' then output <= (others=>'0'); elsif clk='1' then output <= input after 250 ps; end if; end process reg*32;

```
      -- assumes  use IEEE.std_logic_textio.all
```

printout: process(clk) – used to show state when clock raises variable my*line : LINE; – not part of working circuit begin if clk='1' then write(my*line, string'("at clock")); write(my*line, counter); write(my*line, string'(" PC=")); write(my*line, IF*PC); writeline(output, my_line); counter <= counter+1; end if; end process printout;

process*declarative*items are any of: subprogram declaration subprogram body type declaration subtype declaration constant, object declaration variable, object declaration file, object declaration alias declaration attribute declaration attribute specification use clause group template declaration group declaration

BUT NOT signal_declaration, all signals must be declared outside the process. sig1 <= sig2 and sig3; – considered here as a sequential statement – sig1 is set outside the process upon exit or wait

A process may be designated as postponed in which case it starts in the same simulation cycle as an equivalent non postponed process, yet starts after all other non postponed processes have suspended in that simulation cycle.

concurrent procedure call statement

A sequential procedure call statement may be used and its behavior is that of an equivalent process.

[label :] [postponed] procedure name [(actual_parameters)] ;

trigger*some*event ;

Check*Timing(min*time, max*time, clk, sig*to_test);

Note that a procedure can be defined in a library package and then used many places. A process can not be similarly defined in a package and may have to be physically copied. A process has some additional capability not available in a concurrent procedure.

concurrent assertion statement

A sequential assertion statement may be used and its behavior is that of an equivalent process.

[label :] [postponed] assertion_statement ;

concurrent signal assignment statement

A sequential signal assignment statement is also a concurrent signal assignment statement. Additional control is provided by the use of postponed and guarded.

[label :] sequential signal assignment statement

[label :] [postponed] conditional*signal*assignment_statement ;

[label :] [postponed] selected*signal*assignment_statement ;

The optional guarded causes the statement to be executed when the guarded signal changes from False to True. conditional signal assignment statement A conditional assignment statement is also a concurrent signal assignment statement.

target <= waveform when choice; – choice is a boolean expression target <= waveform when choice else waveform;

sig <= a*sig when count>7; sig2 <= not a*sig after 1 ns when ctl='1' else b_sig;

"waveform" for this statement seems to include [delay_mechanism] See sequential signal assignment statement

selected signal assignment statement

A selected assignment statement is also a concurrent signal assignment statement.

with expression select target <= waveform when choice [, waveform when choice] ;

with count/2 select my_ctrl <= '1' when 1, – count/2 = 1 for this choice '0' when 2, 'X' when others;

component instantiation statement

Get a specific architecture-entity instantiated component.

part*name: entity library*name.entity*name(architecture*name) port map (actual arguments) ;

```
                                        optional (architecture_name)
```

part*name: component*name port map (actual arguments) ;

Given entity gate is port (in1 : in std*logic ; in2 : in std*logic ; out1 : out std_logic) ; end entity gate; architecture circuit of gate is ... architecture behavior of gate is ...

A101: entity WORK.gate(circuit) port map (in1 => a, in2 => b, out1 => c);

```
    -- when gate has only one architecture
```

A102: entity WORK.gate port map (in1 => a, in2 => b, out1 => c);

```
    -- when order of actual arguments is used
```

A103: entity WORK.gate port map (a, b, c);

Given an entity entity add*32 is – could have several architectures port (a : in std*logic*vector (31 downto 0); b : in std*logic*vector (31 downto 0); cin : in std*logic; sum : out std*logic*vector (31 downto 0); cout : out std*logic); end entity add*32;

Create a simple component interface component add*32 – use same port as entity port (a : in std*logic*vector (31 downto 0); b : in std*logic*vector (31 downto 0); cin : in std*logic; sum : out std*logic*vector (31 downto 0); cout : out std*logic); end component add*32;

Instantiate the component 'add*32' to part name 'PC*incr' PC*incr : add*32 port map (PC, four, zero, PC_next, nc1);

Create a component interface, changing name and renaming arguments component adder – can have any name but same types in port port (in1 : in std*logic*vector (31 downto 0); in2 : in std*logic*vector (31 downto 0); cin : in std*logic; sum : out std*logic*vector (31 downto 0); cout : out std*logic); end component adder;

Instantiate the component 'adder' to part name 'PC*incr' PC*incr : adder – configuration may associate a specific architecture port map (in1 => PC, in2 => four, cin => zero, sum => PC_next, cout => nc1);

generate statement

Make copies of concurrent statements

label: for variable in range generate – label required block declarative items ___ optional
begin / concurrent statements – using variable end generate label ;

label: if condition generate – label required block declarative items ___ optional
begin / concurrent statements end generate label ;

band : for I in 1 to 10 generate b2 : for J in 1 to 11 generate b3 : if abs(I-J)<2 generate part: foo port map (a(I), b(2*J-1), c(I, J)); end generate b3; end generate b2; end generate band;

1 VHDL Predefined Types from the package standard

The type and subtype names below are automatically defined. They are not technically reserved words but save yourself a lot of grief and do not re-define them.

Note that enumeration literals such as "true" and "false" are not technically reserver words and can be easily overloaded, but save future readers of your code the confusion. It is confusing enough that '0' and '1' are enumeration literals of both type Character and type Bit. "01101001" is of type string, bit*vector, std*logic_vector and more.

There is no automatic type conversion in VHDL, yet users and libraries may provide almost any type conversion. For numeric types integer(X) yields the rounded value of the real variable X as an integer, real(I) yields the value of the integer variable I as a real.

Predefined type declarations

Notes: Reserver words are in bold type, Type names are alphabetical and begin with an initial uppercase letter. Enumeration literals are in plain lower case.

type Bit is ('0', '1');

type Bit_vector is array (Natural range <>) of Bit;

type Boolean is (false, true);

type Character is (–256 characters–);

subtype Delay_length is Time range 0 fs to Time'high;

type File*open*kind is (read*mode, write*mode, append_mode);

type File*open*status is (open*ok, status*error, name*error, mode*error);

type Integer is range –usually typical integer– ;

subtype Natural is Integer range 0 to Integer'high;

subtype Positive is Integer range 1 to Integer'high;

type Real is range –usually double precision floating point– ;

type Severity_level is (note, warning, error, failure);

type String is array (Positive range <>) of Character;

type Time is range –implementation defined– ; units fs; – femtosecond ps = 1000 fs; – picosecond ns = 1000 ps; – nanosecond us = 1000 ns; – microsecond ms = 1000 us; – millisecond sec = 1000 ms; – second min = 60 sec; – minute hr = 60 min; – hour end units;

attribute Foreign : String ;

impure function Now return Delay_length;

The type classification of VHDL is shown below. Users can declare their own types and subtypes. A type statement is used to declare a new type. A subtype statement is used to constrain an existing type.

types-+-scalar—-+-discrete——-+-integer——-+-integer | | | +-natural | | | +-positive | | | | | +-enumeration— +-boolean | | +-bit | | +-character | | +-file*open*kind | | +-file*open*status | | +-severity*level | | | +-floating point-+—————real | | | +-physical——-+—————delay*length | +————time | +-composite-+-array— ——-+-constrained- | | | | | +-unconstrained-+-bit_vector | | +-string | | | +-record- | +-access- | +-file-

VHDL Declaration Statements

Various declarations may be used in various design units. Check the particular design unit for applicability.

Declaration Statements

incomplete type declaration scalar type declaration composite type declaration access type declaration file type declaration subtype declaration constant, object declaration signal, object declaration variable, object declaration file, object declaration alias declarations attribute declaration attribute specification component declaration group template declaration group declaration disconnect specification incomplete type declaration

Declare an identifier to be a type. The full type definition must be provided within this scope.

type identifier ;

type node ;

scalar type declaration

Declare a type that may be used to create scalar objects.

type identifier is scalar*type*definition ;

type my*small is range -5 to 5 ; type my*bits is range 31 downto 0 ; type my_float is range 1.0 to 1.0E6 ;

composite type declaration

Declare a type for creating array, record or unit objects.

type identifier is composite*type*definition ;

type word is array (0 to 31) of bit; type data is array (7 downto 0) of word; type mem is array (natural range <>) of word; type matrix is array (integer range <>, integer range <>) of real;

type stuff is record I : integer; X : real; day : integer range 1 to 31; name : string(1 to 48); prob : matrix(1 to 3, 1 to 3); end record;

type node is – binary tree record key : string(1 to 3); data : integer; left : node*ptr; right : node*ptr; color : color_type; end record;

type distance is range 0 to 1E16 units Ang; – angstrom nm = 10 Ang; – nanometer um = 1000 nm; – micrometer (micron) mm = 1000 um; – millimeter cm = 10 mm; – centimeter dm = 100 mm; – decameter m = 1000 mm; – meter km = 1000 m; – kilometer

```
  mil  = 254000 Ang;   -- mil (1/1000 inch)
  inch = 1000 mil;     -- inch
  ft   = 12 inch;      -- foot
  yd   = 3 ft;         -- yard
  fthn = 6 ft;         -- fathom
  frlg = 660 ft;       -- furlong
  mi   = 5280 ft;      -- mile
  lg   = 3 mi;         -- league
end units;
```

access type declaration

Declare a type for creating access objects, pointers. An object of an access type must be of class variable. An object

type identifier is access subtype_indication;

type node_ptr is access node;

variable root : node_ptr := new node'("xyz", 0, null, null, red); variable item : node := root.all;

file type declaration

Declare a type for creating file handles.

type identifier is file of type_mark ;

type my_text is file of string ;

type word_file is file of word ;

file output : my*text; file*open(output, "my.txt", write*mode); write(output, "some text"&lf); file*close(output);

file test*data : word*file; file*open(test*data, "test1.dat", read*mode); read(test*data, word_value);

subtype declaration

Declare a type that is a subtype of an existing type. Note that type creates a new type while subtype creates a type that is a constraint of an existing type.

subtype identifier is subtype_indication ;

subtype name*type is string(1 to 20) ; variable a*name : name_type := “Doe, John”;

subtype small*int is integer range 0 to 10 ; variable little : small*int := 4;

subtype word is std*logic*vector(31 downto 0) ; signal my_word : word := x“FFFFFFFC”;

constant, object declaration

Used to have an identifier name for a constant value. The value can not be changed by any executable code.

constant identifier : subtype*indication := constant*expression;

constant Pi : real := 3.14159; constant Half*Pi : real := Pi/2.0; constant cycle*time : time := 2 ns; constant N, N5 : integer := 5;

A deferred constant has no := constant_expression can only be used in a package declaration and a value must appear in the package body.

signal, object declaration

Used to define an identifier as a signal object. No explicit initialization of an object of type T causes the default initialization at time zero to be the value of T’left

signal identifier : subtype_indication [signal_kind] [:= expression];

signal a*bit : bit := ‘0’; a*bit <= b_bit xor ‘1’; – concurrent assignment

signal my*word : word := X“01234567”; my*word <= X“FFFFFFFF”; – concurrent assignment

signal foo : word register; – guarded signal signal bar : word bus; – guarded signal signal join : word wired*or; – wired*or must be a resolution function

signal_kind may be register or bus.

A simple signal of an unresolved type can have only one driver. Note that “bit” is an unresolved type as is “std*ulogic“, but, ”std*logic” is a resolved type and allows multiple drivers of a simple signal. variable, object declaration

Used to define an identifier as a variable object. No explicit initialization of an object of type T causes the default initialization at time zero to be the value of T’left

variable identifier : subtype_indication [:= expression];

variable count : integer := 0; count := count + 1;

A variable may be declared as shared and used by more than one process, with the restriction that only one process may access the variable in a single simulation cycle.

shared variable identifier : subtype_indication [:= expression];

shared variable status : status_type := stop; status := start;

Note: Variables declared in subprograms and processes must not be declared shared. Variables declared in entities, architectures, packages and blocks must be declared shared. Some analysers/compilers may require shared variables to be ‘protected’.

Note: Both signal and variable use := for initialization. signal uses <= for concurrent assignment variable uses := for sequential assignment file, object declaration

Used to define an identifier as a file object.

file identifier : subtype_indication [file*open*information]

file*open*information [open file*open*kind] is file*logical*name

file*open*kind from use STD.textio.all read*mode write*mode append_mode

use STD.textio.all; – declares types ‘text’ and ‘line’ file my*file : text open write*mode is “file5.dat”; variable my_line : line;

write(my*line, string’(“Hello.”); – build a line writeline(my*file, my_line); – write the line to a file

Note: The file*logical*name is a string in quotes and its syntax must conform to the operating system where the VHDL will be simulated. The old DOS 8.3 format in lower case works on almost all operating systems.

alias declarations

Used to declare an additional name for an existing name.

alias new*name is existing*name*of*same*type ; alias new*name [: subtype_indication] : is [signature];

```
new_name may be an indentifier, a character literal or operator symbol
```

alias rs is my*reset*signal ; – bad use of alias alias mantissa:std*logic*vector(23 downto 0) is my*real(8 to 31); alias exponent is my*real(0 to 7); alias "<" is my_compare [my*type, my*type, return boolean] ; alias 'H' is STD.standard.bit.'1' [return bit] ;

attribute declaration

Users may define attributes to be used in a local scope. Predefined attributes are in the Predefined Attributes section

attribute identifier : type_mark ;

attribute enum*encoding : string; – user defined type my*state is (start, stop, ready, off, warmup); attribute enum*encoding of my*state : type is "001 010 011 100 111"; signal my*status : my*state := off; – value "100"

attribute specification

Used to associate expressions with attributes. Predefined attributes are in the Predefined Attributes section

attribute identifier of name : entity_class is expression ;

entity_class architecture component configuration constant entity file function group label literal package procedure signal subtype type variable units

attribute enum*encoding : string; type my*state is (start, stop, ready, off, warmup); attribute enum*encoding of my*state : type is "001 010 011 100 111"; signal my*status : my*state := off; – value "100"

component declaration

Used to define a component interface. Typically placed in an architecture or package declaration. The component or instances of the component are related to a design entity in a library in a configuration.

component component*name is generic (generic*variable*declarations) ; – optional port (input*and*output*variable*declarations) ; end component component*name ;

generic*variable*declarations are of the form: variable*name : variable*type := value ;

input*and*output*variable*declaration are of the form: variable*name : port*mode variable*type ; port*mode may be in out inout buffer linkage

component reg32 is generic (setup*time : time := 50 ps; pulse*width : time := 100 ps); port (input : in std*logic*vector(31 downto 0); output: out std*logic*vector(31 downto 0); Load : in std*logic*vector; Clk : in std*logic*vector); end component reg32;

Then an instantiation of the reg32 component in an architecture might be:

RegA : reg32 generic map (setup*time => global*setup, pulse_width => 150 ps) – no semicolon port map (input => Ainput, output => Aoutput, Load => Aload, Clk => Clk);

An alternative to the component declaration and corresponding component instantiation above is to use a design entity instantiation.

RegA : entity WORK.reg32(behavior) – library.entity(architecture) generic map (global_setup, 150 ps) – no semicolon port map (Ainput, Aoutput, Aload, Clk);

There is no requirement that the component name be the same as the design entity name that the component represents. Yet, the component name and design entity name are often the same because some systems automatically take the most recently compiled architecture of a library entity with the same name as the component name.

group template declaration

A group template declaration declares a group template, which defines the allowable classes of named entities that can appear in a group.

group identifier is (entity*class*list) ;

entity*class*list entity_class [, entity_class] [<>]

entity_class architecture component configuration constant entity file function group label literal package procedure signal subtype type variable units

– a group of any number of labels group my_stuff is (label <>) ;

group declaration

A group declaration declares a group, a named collection of named entities.

group identifier : group*template*name (group_member [, group member]) ;

group my*group : my*stuff (lab1, lab2, lab3) ;

disconnect specification

A disconnect specification applies to a null transaction such as a guard becoming false.

disconnect signal*name : type*mark after time*expression ; disconnect others : type*mark after time*expression ; disconnect all : type*mark after time_expression ;

disconnect my*sig : std*logic after 3 ns;

VHDL Resolution and Signatures

Contents

Resolution Functions Signatures ’left ’right vs ’high ’low Resolution Functions

A resolution function defines how values from multiple sources, multiple drivers, are resolved into a single value.

A type may be defined to have a resolution function. Every signal object of this type uses the resolution function when there are multiple drivers.

A signal may be defined to use a specific resolution function. This signal uses the resolution function when there are multiple drivers.

A resolution function must be a pure function that has a single input parameter of class constant that is a one dimensional unconstrained array of the type of the resolved signal.

An example is from the package std*logic*1164 :

```
type std_ulogic is ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );

type std_ulogic_vector is array ( natural range <> ) of std_ulogic;


-- resolution function

function resolved ( s : std_ulogic_vector ) return std_ulogic;
    variable result : std_ulogic := 'Z';  -- weakest state default
begin
    -- the test for a single driver is essential otherwise the
    -- loop would return 'X' for a single driver of '-' and that
    -- would conflict with the value of a single driver unresolved
    -- signal.
    if  s'length = 1 then
        return s(s'low);
    else
        for i in s'range loop
            result := resolution_table(result, s(i));
        end loop;
    end if;
    return result;
end resolved;
```

```
constant resolution_table : stdlogic_table := (
--      ---------------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -        |   |
--      ---------------------------------------------------------
         ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
         ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
         ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
         ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
         ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
         ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
         ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
         ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
         ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
    );


subtype std_logic is resolved std_ulogic;


type std_logic_vector is array ( natural range <>) of std_logic;

signal xyz : std_logic_vector(0 to 3);

xyz <= -- some expression ;

xyz <= -- some other expression ; -- a second driver
                       -- each bit of  xyz  comes from function "resolved"
```

Signatures

A signature distinguishes between overloaded subprograms and enumeration literals based on their parameter and result type profiles. A signature may be used in an attribute name, entity designator, or alias declaration.

The syntax of the signature is

[type*mark, type*mark, ... , type*mark return type*mark]

A signature used in an alias statement to give a shorthand to a textio procedure is:

```
alias swrite is write [line, string, side, width] ;
```

allowing swrite(output, “some text”); in place of write(output, string’(“some text”));

The “[line, string, side, width]” is the signature to choose which of the overloaded ‘write’ procedures to alias to ‘swrite’.

No return is used for procedures. The type marks are the parameter types in their defined order. The square brackets at beginning and end are part of the signature. The signature is used immediately after the subprogram or enumeration literal name.

’left ’right vs ’high ’low

This is just a specific example to help understand ‘to’ vs ‘downto’ and how the values of attributes such as ’left ’right and ’high ’low are determined.

A : std*logic*vector(31 downto 0) := x“FEDCBA98”; – ‘downto’ B : std*logic*vector(4 to 27) := x“654321”; – ‘to’ C a literal constant x“321”

Name bitstring (attributes on following lines)

A 11111110110111001011101010011000 A’left=31 A’right=0 A’low=0 A’high=31 A(A’left)=1 A(A’right)=0 A(A’low)=0 A(A’high)=1 A’range=(31 downto 0) A’reverse_range=(0 to 31) A’length=32 A’ascending=false

B 011001010100001100100001 B’ascending=true B’left=4 B’right=27 B’low=4 B’high=27 B(B’left)=0 B(B’right)=1 B(B’low)=0 B(B’high)=1 B’range=(4 to 27) B’reverse_range=(27 downto 4) B’length=24 B’ascending=true

C 001100100001 C’left=0 C’right=11 C’low=0 C’high=11 C(C’left)=0 C(C’right)=1 C(C’low)=0 C(C’high)=1 C’range=(0 to 11) C’reverse_range=(11 downto 0) C’length=12 C’ascending=true

```
Notice the default values of attributes on literal constants.
Always a range of (0 to 'length-1)  'left  = 'low  = 0
                                    'right = 'high = 'length-1
```

VHDL Reserved Words

abs operator, absolute value of right operand. No () needed. access used to define an access type, pointer after specifies a time after NOW alias create another name for an existing identifier all dereferences what precedes the .all and operator, logical "and" of left and right operands architecture a secondary design unit array used to define an array, vector or matrix assert used to have a program check on itself attribute used to declare attribute functions begin start of a begin end pair block start of a block structure body designates a procedure body rather than declaration buffer a mode of a signal, holds a value bus a mode of a signal, can have multiple drivers case part of a case statement component starts the definition of a component configuration a primary design unit constant declares an identifier to be read only disconnect signal driver condition downto middle of a range 31 downto 0 else part of "if" statement, if cond then ... else ... end if; elsif part of "if" statement, if cond then ... elsif cond ... end part of many statements, may be followed by word and id entity a primary design unit exit sequential statement, used in loops file used to declare a file type for start of a for type loop statement function starts declaration and body of a function generate make copies, possibly using a parameter generic introduces generic part of a declaration group collection of types that can get an attribute guarded causes a wait until a signal changes from False to True if used in "if" statements impure an impure function is assumed to have side effects in indicates a parameter in only input, not changed inertial signal characteristic, holds a value inout indicates a parameter is used and computed in and out is used as a connective in various statements label used in attribute statement as entity specification library context clause, designates a simple library name linkage a mode for a port, used like buffer and inout literal used in attribute statement as entity specification loop sequential statement, loop ... end loop; map used to map actual parameters, as in port map mod operator, left operand modulo right operand nand operator, "nand" of left and right operands new allocates memory and returns access pointer next sequential statement, used in loops nor operator, "nor" of left and right operands not operator, complement of right operand null sequential statement and a value of used in type declarations, of Real ; on used as a connective in various statements open initial file characteristic or operator, logical "or" of left and right operands others fill in missing, possibly all, data out indicates a parameter is computed and output package a design unit, also package body port interface definition, also port map postponed make process wait for all non postponed process to suspend procedure typical programming procedure process sequential or concurrent code to be executed pure a pure function may not have side effects range used in type definitions, range 1 to 10; record used to define a new record type register signal parameter modifier reject clause in delay mechanism, followed be a time rem operator, remainder of left operand divided by right op report statement and clause in assert statement, string output return statement in procedure or function rol operator, left operand rotated left by right operand ror operator, left operand rotated right by right operand select used in selected signal assignment statement severity used in assertion and reporting, followed by a severity signal declaration that an object is a signal shared used to declare shared objects sla operator, left operand shifted left arithmetic by right op sll operator, left operand shifted left logical by right op sra operator, left operand shifted right arithmetic by right srl operator, left operand shifted right logical by right op subtype declaration to restrict an existing type then part of if condition then ... to middle of a range 1 to 10 transport signal characteristic type declaration to create a new type unaffected used in signal waveform units used to define new types of units until used in wait statement use make a package available to this design unit variable declaration that an object is a variable wait sequential statement, also used in case statement when used for choices in case and other statements while kind of loop statement with used in selected signal assignment statement
xnor operator, exclusive "nor" of left and right operands xor operator, exclusive "or" of left and right operands

VHDL Operators

Highest precedence first, left to right within same precedence group, use parenthesis to control order. Unary operators take an operand on the right. "result same" means the result is the same as the right operand. Binary operators take an operand on the left and right. "result same" means the result is the same as the left operand.

** exponentiation, numeric ** integer, result numeric abs absolute value, abs numeric, result numeric not complement, not logic or boolean, result same

- multiplication, numeric * numeric, result numeric / division, numeric / numeric, result numeric mod modulo, integer mod integer, result integer rem remainder, integer rem integer, result integer
- unary plus, + numeric, result numeric
- unary minus, - numeric, result numeric
- addition, numeric + numeric, result numeric
- subtraction, numeric - numeric, result numeric & concatenation, array or element & array or element, result array

sll shift left logical, logical array sll integer, result same srl shift right logical, logical array srl integer, result same sla shift left arithmetic, logical array sla integer, result same sra shift right arithmetic, logical array sra integer, result same rol rotate left, logical array rol integer, result same ror rotate right, logical array ror integer, result same

= test for equality, result is boolean /= test for inequality, result is boolean < test for less than, result is boolean <= test for less than or equal, result is boolean > test for greater than, result is boolean >= test for greater than or equal, result is boolean

and logical and, logical array or boolean, result is same or logical or, logical array or boolean, result is same nand logical complement of and, logical array or boolean, result is same nor logical complement of or, logical array or boolean, result is same xor logical exclusive or, logical array or boolean, result is same xnor logical complement of exclusive or, logical array or boolean, result is same

VHDL Predefined Attributes

The syntax of an attribute is some named entity followed by an apostrophe and one of the following attribute names. A parameter list is used with some attributes. Generally: T represents any type, A represents any array or constrained array type, S represents any signal and E represents a named entity.

T’BASE is the base type of the type T T’LEFT is the leftmost value of type T. (Largest if downto) T’RIGHT is the rightmost value of type T. (Smallest if downto) T’HIGH is the highest value of type T. T’LOW is the lowest value of type T. T’ASCENDING is boolean true if range of T defined with to . T’IMAGE(X) is a string representation of X that is of type T. T’VALUE(X) is a value of type T converted from the string X. T’POS(X) is the integer position of X in the discrete type T. T’VAL(X) is the value of discrete type T at integer position X. T’SUCC(X) is the value of discrete type T that is the successor of X. T’PRED(X) is the value of discrete type T that is the predecessor of X. T’LEFTOF(X) is the value of discrete type T that is left of X. T’RIGHTOF(X) is the value of discrete type T that is right of X. A’LEFT is the leftmost subscript of array A or constrained array type. A’LEFT(N) is the leftmost subscript of dimension N of array A. A’RIGHT is the rightmost subscript of array A or constrained array type. A’RIGHT(N) is the rightmost subscript of dimension N of array A. A’HIGH is the highest subscript of array A or constrained array type. A’HIGH(N) is the highest subscript of dimension N of array A. A’LOW is the lowest subscript of array A or constrained array type. A’LOW(N) is the lowest subscript of dimension N of array A. A’RANGE is the range A’LEFT to A’RIGHT or A’LEFT downto A’RIGHT . A’RANGE(N) is the range of dimension N of A. A’REVERSE*RANGE is the range of A with to and downto reversed. A’REVERSE*RANGE(N) is the REVERSE*RANGE of dimension N of array A. A’LENGTH is the integer value of the number of elements in array A. A’LENGTH(N) is the number of elements of dimension N of array A. A’ASCENDING is boolean true if range of A defined with to . A’ASCENDING(N) is boolean true if dimension N of array A defined with to . S’DELAYED(t) is the signal value of S at time now - t . S’STABLE is true if no event is occurring on signal S. S’STABLE(t) is true if no even has occurred on signal S for t units of time. S’QUIET is true if signal S is quiet. (no event this simulation cycle) S’QUIET(t) is true if signal S has been quiet for t units of time. S’TRANSACTION is a bit signal, the inverse of previous value each cycle S is active. S’EVENT is true if signal S has had an event this simulation cycle. S’ACTIVE is true if signal S is active during current simulation cycle. S’LAST*EVENT is the time since the last event on signal S. S’LAST*ACTIVE is the time since signal S was last active. S’LAST*VALUE is the previous value of signal S. S’DRIVING is false only if the current driver of S is a null transaction. S’DRIVING*VALUE is the current driving value of signal S. E’SIMPLE*NAME is a string containing the name of entity E. E’INSTANCE*NAME is a string containing the design hierarchy including E. E’PATH*NAME is a string containing the design hierarchy of E to design root.

Standard VHDL Packages

VHDL standard packages and types

The following packages should be installed along with the VHDL compiler and simulator. The packages that you need, except for “standard”, must be specifically accessed by each of your source files with statements such as:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_textio.all;
use IEEE.std_logic_arith.all;
use IEEE.numeric_bit.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_signed.all;
use IEEE.std_logic_unsigned.all;
use IEEE.math_real.all;
use IEEE.math_complex.all;

library STD;
use STD.textio;
```

A version of these packages, declaration and body, are in this directory

The package standard is predefined in the compiler. Types defined include: bit bit*vector typical signals integer natural positive typical variables boolean string character typical variables real time delay*length typical variables Click on standard to see the functions defined Note: This package must be provided with compiler, do not use this one.

The package textio provides user input/output Types defined include: line text side width Functions defined include: readline read writeline write endline Click on textio to see how to call the functions

The package std*logic*1164 provides enhanced signal types Types defined include: std*ulogic std*ulogic*vector std*logic std*logic*vector Click on std*logic*1164 to see available functions

The package std*logic*textio provides input/output for 1164 types Functions defined include: readline read writeline write endline Click on std*logic*textio to see how to call the functions

The package std*logic*arith provides numerical computation This package name unfortunately seams to have several definitions:

std*logic*arith*syn.vhd defines types signed and unsigned and has arithmetic functions that operate on signal types signed and unsigned and std*logic*vector and std*ulogic*vector, but adding A to B of std*logic*vector type, needs unsigned(A) + unsigned(B). Click on std*logic*arith*syn to see the functions defined

std*logic*arith*ex.vhd has arithmetic functions that operate on signal types std*logic*vector and std*ulogic*vector Click on std*logic*arith*ex to see the functions defined

The package numeric*bit provides numerical computation Types defined include: unsigned signed arrays of type bit for signals Click on numeric*bit to see the functions defined

The package numeric*std provides numerical computation Types defined include: unsigned signed arrays of type std*logic for signals Click on numeric_std to see the functions defined

The package std*logic*signed provides signed numerical computation on type std*logic*vector

Click on std*logic*signed to see the functions defined

The package std*logic*unsigned provides unsigned numerical computation on type std*logic*vector Click on std*logic*unsigned to see the functions defined

The package math_real provides numerical computation on type real

Click on math_real to see the functions defined This declaration and body are in mathpack

The package math*complex provides numerical computation Types defined include: complex, complex*vector, complex_polar

Click on math_complex to see the functions defined This declaration and body are in mathpack