Data Objects

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice and Octave.

Data Objects

Data Objects (1)

An object in VHDL is a named item
that holds the value of a specific data type.

There are four types of data objects in VHDL:

- **Signals**
- **Constants**
- **Variables**
- **File**

For describing logic circuits,
the most important data objects are signals.

They represent the logic signals (wires) in the circuit.

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Data Objects (2)

Data Object

- Value of a specific data type
- Operators

*a **type** defines*
a set of <u>values</u> that an object can assume and
a set of <u>operations</u> that can be performed on
objects of that type.

Classes of Data Objects

- Constant – no change
- Variable – change without any delay
- Signal – change with a certain or delta delay
- File

Classes of Data Objects

To model the behavior of a circuit

- Constant – no change

  static : local to process, be held until the next process call

  can be declared in processes, procedures, functions, architectures

- Variable – change without any delay

  dynamic : not be held from one call to the next

  must be declared inside a process

Represents wires in the schematic of a circuit

- Signal – change with a certain / delta delay

  must be declared outside a process

Signal Data Object

Signal Data Object (1)

The signal represents interconnection wires between ports

it may be declared in the declaration part of
- packages
- entities
- architectures
- blocks

The signal declaration is

signal signal_name : signal_type;

Signal assignment: <=

https://surf-vhdl.com/vhdl-syntax-web-course-surf-vhdl/vhdl-types-of-data-object/

Signal Data Objects (2)

An object in VHDL is a named item

A signal is an object that holds
the current and possible future values
of the object.

signals occur
as inputs and outputs in port descriptions,
as signals in architecture, etc.

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Signal Data Objects (3)

Where can signal data objects be declared?

entity declaration (port input, output)
declarative section of an architecture

cannot be in a process

```
entity nand2 is
port (A, B:    in  bit;
        C     :    out  bit );
end entity nand2;
```

```
architecture dataflow  of nand2 is
   signal S: bit;
begin
   S <= A and B;
   C <= not S;
end architecture dataflow;
```

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Signal Declaration

How to declare a signal?

signal signal_name : signal_type [:= initial_value] ;

examples:

signal status : std_logic := ‘0’;
signal data : std_logic_vector (31 downto 0);

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Signal Assignment

A signal assignment schedules a new value to occur at some future time.

The current value of the signal is never changed.

If no specific value of time is specified the default time value is infinitesimally small value of time into the future called delta time.

Signals are assigned using the “<=“ operator. e.g.

```
X1 <= ‘1’ after 10ns;
SR1 <= 5  after 5ns;
X2 <= ‘0’ after 10ns, ‘1’ after 20ns, ‘0’ after 30ns;
X5 <= ‘1’;
```

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Signal Example

```
architecture sarch of sent is
    signal trigger, result : integer := 0;
begin
    process
        signal sig1: integer := 1;
        signal sig2: integer := 2;
        signal sig3: integer := 3;
    begin
        wait on trigger;
        sig1    <=    sig2 ;

        sig2    <=    sig1 + sig3 ;

        sig3    <=    sig2 ;

        result  <=    sig1 + sig2 + sig3 ;
    end process
end sarch
```

Variable Data Object

Variable Data Object

The variable locally stores temporary data
and it is used only inside a sequential statement

process
function
procedures

The variable is visible only inside processes and subprograms
in which it is declared.

The variable declaration is

variable variable_name : variable_type;

Variable assignment: :=

https://surf-vhdl.com/vhdl-syntax-web-course-surf-vhdl/vhdl-types-of-data-object/

Variable Data Objects (1)

Variables are used to hold temporary data.

Where to declare a variable?

within the processes, functions and procedures
in which they are used

How to declare a variable?

variable variable_name : variable_type [:= initial_value];

Examples:

```
variable address : bit_vector (15 downto 0) := x"0000";
variable index : integer range 0 to 10 := 0;
```

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Variable Assignment

In contrast to signal assignment,
a variable assignment takes effect immediately.

Variables are assigned using the “**:=**“ operator. e.g.

A := ‘1’;
ROM_A(5) := ROM_A(0);
STAR_COLOR := GREEN;

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Variable Example

```
architecture varch of vent is
        signal trigger, result : integer := 0;
begin
        process
                variable var1: integer := 1;
                variable var2: integer := 2;
                variable var3: integer := 3;
        begin
                wait on trigger;
                var1 := var2;

                var2 := va1 + var3;

                var3 := var2;

                result <= var1 + var2 + var3;
        end process
end varch
```

Constant Data Object

Constant Data Object

The constant names specific values
to make the model better documented and easy to update.

The constant can be declared
in all the declarative VHDL statement,

- sequential
- concurrent

that means it may be declared in the declaration section of

- packages
- entities
- architectures
- processes
- subprograms
- blocks

The constant declaration is

constant constant_name : constant_type := value;

https://surf-vhdl.com/vhdl-syntax-web-course-surf-vhdl/vhdl-types-of-data-object/

Constant Data Objects

A constant is an object which is initialized to a specific value when it is created, and which cannot be subsequently modified.

Where can constants be declared?

- Declarative section of an architecture
- Declarative section of a process

How to declare a constant?

```
constant constant_name : constant_type [:= initial_value];
```

examples:

```
constant yes : boolean := TRUE;
constant msb : integer := 5;
```

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

File Data Object

File Data Object

The File type is used to access File on disk.

It is used only in test bench;
in fact File type cannot be implemented in hardware.

In order to use the FILE type
you shall include the TextIO package
that contains all procedures and functions
that allow you to read from and write to formatted text files.

Input ASCII files are handled as file of lines,
where a line is a string, terminated by a carriage return.

TextIO package declares a type line used to hold

a line read from an input file
a line to write to an output file

https://surf-vhdl.com/vhdl-syntax-web-course-surf-vhdl/vhdl-types-of-data-object/

File Examples

```
process_write_file : process(...)

  file file_name_write        : text;      -- declare file
  variable row                : line;      -- line to access file
  variable integer_value      : integer;   -- integer value write to file

begin

      -- open file write mode
      file_open(file_name_write, “file_to_write.txt”, write_mode);

      -- write line from file and then write integer value
      writeline(file_name_write, row);
      write(row, integer_value);
      ...

end process process_write_file;
```

https://surf-vhdl.com/vhdl-syntax-web-course-surf-vhdl/vhdl-types-of-data-object/

Write

write procedure writes data to a line variable
without automatically appending a new line character.

It allows for building up a line of text
by concatenating multiple write calls
before finally writing the complete line to a file.

```
-- Example using write
variable my_line : line;
write(my_line, "Hello");
write(my_line, " World!");
-- The content of my_line is now "Hello World!"
```

Google AI Overview

Writeln

writeline procedure writes the content of a line variable
to a specified text file and automatically appends a line feed (LF) character,
effectively moving the cursor to the beginning of the next line in the file.

```
-- Example using writeline
variable my_line : line;
file output_file : text open write_mode is "output.txt";

write(my_line, "This is a line.");
writeline(output_file, my_line); -- Writes "This is a line." followed by a new line
```

```
file output_file          : text;        -- declare file
file_open(output_file, “output.txt”, write_mode);
```

Google AI Overview

Readln

readline procedure is used to read an entire line of text from a file and store it into a variable of type LINE.

The LINE type is a pre-defined access type (pointer) to a string, effectively acting as a buffer for the line's content.

```
readline(File_Variable, Line_Variable);
```

```
variable my_line : line;
readline(input_file, my_line);
 -- Reads a line from 'input_file' into 'my_line'
```

Google AI Overview

Read

read procedure is used to extract specific data elements
(like integers, characters, strings, etc.) from a LINE variable
and assign them to a VHDL variable of the corresponding type.

You call read multiple times on the same LINE variable
to extract different data elements within that line.

Syntax: read(Line_Variable, Data_Variable);

```
variable my_line : line;
variable my_integer : integer;
variable my_char : character;

readline(input_file, my_line);      -- First, read the line
read(my_line, my_integer);          -- Then, read an integer from that line
read(my_line, my_char);             -- And then, read a character from the same line
```

Google AI Overview

Data Types

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Types of Signal, variable, constant objects

The type of a signal, variable, or constant object specifies:

the *range* of values it may take
the *set of operations* that can be performed on it.

The VHDL language supports
a predefined standard set of type definitions
the definition of new types by users.

8 types commonly used:

bit
bit_vector
integer
boolean
array
enumeration
std_logic
std_logic_vector

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Bit and Bit_vector

Bit type has two values, ‘0’ and ‘1’.

Example:

signal a : bit := ‘0’;
variable b : bit ;

Bit_vector is an array where each element is of type bit.

Example:

signal c : bit_vector (3 downto 0) := “1000”; -- recommended
signal d : bit_vector (0 to 3) := “1000”;

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Integer type

INTEGER type represents positive, negative numbers and 0.

By default, an INTEGER signal has 32 bits
and can represent numbers from -2^{31} to $2^{31}-1$.

The code does not specifically give the number of bits in the signal.

Integers with fewer bits than 32 can be declared,
using the RANGE keyword.

Example:

```
signal x : integer range -128 to 127;
```

This defines x as an eight-bit signed number.

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Boolean type

An object of type BOOLEAN can have
the values TRUE or FALSE,
where TRUE is equivalent to 1 and FALSE to 0.

Example:

```
signal flag : boolean;
constant correct : boolean := TRUE;
```

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Enumeration type

An ENUMERATION type is defined
by listing all possible values of that type.

All of the values of an enumeration type are user-defined.

```
type enumerated_type_name is  (name {, name});
```

The most common example of using the ENUMERATION type is
for specifying the states for a finite-state machine.

Example:

```
type State_type is (stateA, stateB, stateC);
signal y : State_type := stateB ;
```

When the code is translated by the VHDL compiler,
it automatically assigns bit patterns (codes)
to represent stateA, stateB and stateC.

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Array type

ARRAY types group
one or more elements of the same type together
as a single object.

```
type array_type_name is array (index_range) of element_type;

type byte is array (7 downto 0) of bit;
type word is array (15 downto 0) of bit;
type memory is array ( 0 to 4095 ) of word;

signal program_counter: word := ”0101010101010101”;
variable data_memory: memory;
```

To refer individual elements of array:

program_counter(5 downto 0)	accesses the 6 LSBs of program_counter.
data_memory(0)	accesses the first record in memory.

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

Standard Logic Data Type

std_logic
std_ulogic

std_logic_vector
std_ulogic_vector

std_logic (1)

std_logic represents a single digital signal — similar to a wire carrying a bit.

unlike the basic bit type (only '0' and '1'),

std_logic supports *nine* distinct values

to model real-world digital behavior.

std_logic value

‘U’,	uninitialized
‘X’,	unknown
‘0’,	forcing 0
‘1’,	forcing 1
‘Z’,	high impedance
‘W’,	weak unknown
‘L’,	weak 0
‘H’,	weak 1
‘-’	don’t care

MicroSoft Copilot : vhdl std_logic

std_logic (2)

Why Use std_logic?

- Simulation accuracy:

  Models real-world conditions like floating signals ('Z') or unknown s tates ('X').

- Synthesis compatibility:

  Widely supported by synthesis tools.

- Design clarity:

  Helps identify issues like uninitialized signals ('U') during simulation.

Use std_logic_vector for multi-bit signals.

Convert between types using functions like to_stdlogicvector() or to_integer() depending on your library.

MicroSoft Copilot : vhdl std_logic

std_logic (3)

std_logic is essentially

a resolved subtype of std_ulogic,

meaning it can handle

multiple drivers on a signal line

using a resolution function.

this makes it ideal

for modeling real-world digital circuits

where multiple sources might

drive the same signal.

Supports multiple logic states:

Useful for simulation and debugging.

Resolved type:

Automatically resolves conflicts

when multiple drivers are present.

Used in synthesis and simulation:

Compatible with most synthesis tools and simulators.

MicroSoft Copilot : vhdl std_logic definition

std_logic and std_logic_vector (1)

std_logic provides more flexibility than the bit.

To use, you must include the two statements:

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

std_logic_vector type represents
an array of std_logic objects.

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

std_logic and std_logic_vector (2)

Example

```
signal x1, x2, Cin, Cout, Sel   : std_logic;
signal C     : std_logic_vector (1 to 4);
signal X, Y, S      : std_logic_vector (3 downto 0);
```

std_logic objects are often used in logic expressions.

std_logic_vector objects can be used
as binary numbers in arithmetic circuits
by including in the code the following statement

```
use ieee.std_logic_signed.all;       or
use ieee.std_logic_unsigned.all;
```

To use std_logic

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

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

std_logic vs. bit

VHDL is a strongly type-checked language.

Even for objects that intuitively seem compatible,
like bit and std_logic, one cannot be assigned to another.

use std_logic and std_logic_vector types
(Recommendation)

the bit type is defined in the STANDARD package,
which is implicitly available in all VHDL designs.

```
library std;
use std.standard.all

type bit is ('0', '1');
```

https://sustechvhdl.readthedocs.io/lecture/chapter2.html#id1

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

type std_ulogic is
(
    'U',        -- uninitialized
    'X',        -- unknown
    '0',        -- forcing 0
    '1',        -- forcing 1
    'Z',        -- high impedance
    'W',        -- weak unknown
    'L',        -- weak 0
    'H',        -- weak 1
    '-'         -- don't care
);
```

```
SUBTYPE std_logic IS
    resolved std_ulogic;
```

std_logic and std_ulogic (1)

std_logic type is not a basic type
but a subtype defined in the IEEE standard
std_logic_1164 package.

The source code for this standard defines
an unresolved base type std_ulogic and
then uses a resolution function
to create the final std_logic subtype.

```
SUBTYPE std_logic IS resolved std_ulogic;
```

Google AI Overview

std_logic and std_ulogic (2)

std_ulogic (Unresolved Logic)

a type where multiple drivers on a signal
would result in an error during elaboration,
as it lacks a built-in resolution mechanism.

```
SUBTYPE std_logic IS resolved std_ulogic;
```

std_logic (Standard Logic)

a resolved subtype of std_ulogic.

it includes a predefined resolution function

Also named “resolved“
in the IEEE.std_logic_1164

a predefined resolution function defines
how conflicting values from multiple drivers
are combined into a single, resolved value.

For instance,
if one driver attempts to set a std_logic signal
to '0' and another to '1', the resolution function
determines the outcome (which might be 'X' for
unknown, or based on a defined priority).

Google AI Overview

std_logic and std_ulogic (3)

std_ulogic (Unresolved Logic)

-- from an implementation of the IEEE 1164 package
-- the base, unresolved logic type

```
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
);
```

std_logic (Standard Logic)

-- the resolution function that determines the final value
-- when multiple drivers conflict.

```
FUNCTION resolved (s : std_ulogic_vector)
        RETURN std_ulogic;
```

-- The std_logic subtype,
-- which uses the resolution function.

```
SUBTYPE std_logic IS resolved std_ulogic;
```

Google AI Overview

std_ulogic

std_ulogic is the underlying enumerated type, which defines the nine possible logic states.

A signal of this type cannot have multiple drivers, and the simulator will report an error if they occur.

The initial value for any std_ulogic signal is 'U' (Uninitialized) because it is the first value in the enumeration.

```
TYPE std_ulogic IS (
    'U',        -- Uninitialized      → default
    'X',        -- Forcing Unknown
    '0',        -- Forcing 0
    '1',        -- Forcing 1
    'Z',        -- High Impedance
    'W',        -- Weak Unknown
    'L',        -- Weak 0
    'H',        -- Weak 1
    '-'         -- Don't Care
);
```

Google AI Overview

Subtype std_logic (1)

```
SUBTYPE std_logic IS resolved std_ulogic;
```

std_logic is the final subtype

The resolved keyword
automatically associates
the resolved function
with this subtype,

to handle multiple drivers gracefully.

the resolved keyword is used
in the declaration of a subtype

to indicate that
a resolution function should be applied

when multiple drivers are connected
to a signal of that subtype.

Google AI Overview

Subtype std_logic (2)

The std_logic subtype is defined
as a resolved version of std_ulogic,

the resolution function is automatically called
whenever a signal assignment occurs.

```
SUBTYPE std_logic IS resolved std_ulogic;
```

```
-- the resolution function that determines the final value
-- when multiple drivers conflict.
```

```
FUNCTION resolved (s : std_ulogic_vector)  RETURN std_ulogic;
```

```
-- the std_logic subtype,
-- which uses the resolution function.
```

Google AI Overview : vhdl stdlogic_table type in ieee library package

Resolution steps

When multiple sources drive a std_logic signal, the VHDL simulator performs these steps:

Collects all driver values: It gathers all the values that are being driven onto the signal by different processes.

`s : std_ulogic_vector`

Calls the resolution function: It passes the collection of values (as a std_ulogic_vector) to the internal resolved function.

`resolved (s)`

Applies the table: The resolved function iterates through the input vector, using the stdlogic_table to resolve the final value.

`result := resolution_table(result, s(i));`

Returns the resolved value: The final single std_ulogic value from the resolution table is returned and assigned to the std_logic signal.

This entire process is transparent to the VHDL developer, who only needs to declare and use the std_logic type.

Google AI Overview : vhdl stdlogic_table type in ieee library package

The function resolved (1)

the function **resolved**

takes an array of std_ulogic values
(representing all the drivers on a signal)

returns a single std_ulogic value
according to a predefined resolution table.

for example, if one driver is '0' and another is '1',
the resolution function returns 'X' (Unknown).

```
FUNCTION resolved (s : std_ulogic_vector)
        RETURN std_ulogic;
```

in the function body of resolved

```
        ...
        result := resolution_table(result, s(i));
        ...,
        RETURN result;
```

resolution_table is of the type stdlogic_table

```
constant resolution_table : stdlogic_table
        := (... );
```

Google AI Overview

The function resolved (2)

```
FUNCTION resolved (s : std_ulogic_vector)
         RETURN std_ulogic;
```

```
FUNCTION resolved (s : std_ulogic_vector) RETURN std_ulogic IS
   VARIABLE result : std_ulogic := 'Z'; -- weakest state default
BEGIN
   IF (s'LENGTH = 1) THEN
      RETURN s(s'LOW);
   ELSE
      FOR i IN s'RANGE LOOP
      -- Logic to resolve values based on a resolution table
      -- (e.g., combining '0' and '1' results in 'X')
         result := resolution_table(result, s(i));
      END LOOP;
      RETURN result;
   END IF;
END FUNCTION resolved;
```

```
constant resolution_table : stdlogic_table
         := (... );
```

```
type stdlogic_table is array(STD_ULOGIC,
         STD_ULOGIC) of STD_ULOGIC;
```

```
SUBTYPE std_logic IS resolved std_ulogic;
```

'LOW attribute, when applied to a vector or an array, returns the lowest legal index of that vector or array.

this attribute is particularly useful when working with arrays or vectors whose ranges might be defined using either **to** or **downto**.

Google AI Overview : vhdl resolved function body

resolution_table (1)

```
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')  -- | -
);
```

To use the table,
find the intersection
of the two driving signals.

The rows represent the first signal and
the columns represent the second.

For example, to find the result of a '1' and an 'L':

Find the row for '1'.
Find the column for 'L'.
The intersection is '1'.

Google AI Overview : vhdl stdlogic_table type in ieee library package

resolution_table (2)

the resolution_table which is
of the type stdlogic_table

defines the behavior when multiple drivers attempt to assign different values to the same std_logic signal.

VHDL *determines* the resolved value, which is particularly important for modeling buses and tri-state logic.

defined in the package body std_logic_1164 and used exclusively by the resolved function, which converts an array of std_ulogic values into a single std_ulogic value.

```
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')  -- | -
);
```

Google AI Overview : vhdl stdlogic_table type in ieee library package

stdlogic_table type

```
type stdlogic_table is array(STD_ULOGIC, STD_ULOGIC) of STD_ULOGIC;
```

The table is an array of std_ulogic values
indexed by std_ulogic values.

The std_ulogic type is
the base, unresolved, 9-value logic type.

Each cell in the stdlogic_table contains
the resolved value for a specific pair of logic states.

The row and column indices correspond
to the two values being resolved.

For example, the entry at resolution_table('0', 'Z') would be '0',
since a strong '0' value dominates a high-impedance 'Z'.

Google AI Overview : vhdl stdlogic_table type in ieee library package

Overloading

Overloading

VHDL overloading refers to the ability
to define multiple subprograms
(functions or procedures) or
operators with the same name,
provided they can be distinguished
by their parameter and/or return type profiles.

This allows for more flexible and readable code,
enabling the same operation or function name
to be applied to different data types.

Google AI Overview : vhdl overloading

Overloading

Benefits of Overloading:

Readability:

allows the use of intuitive names and symbols for operations, making the code easier to understand.

Reusability:

promotes the creation of generic subprograms and operators that can be applied to different data types.

Flexibility:

enables the extension of VHDL's built-in functionality to custom data types.

Google AI Overview : vhdl overloading

Overloading

Considerations:

Ambiguity:

Care must be taken to ensure that
overloaded subprograms or operators
should be distinguishable by the compiler.

If two overloaded entities have identical parameter
and/or return type profiles, it can lead to ambiguity errors.

Visibility and **Scoping**:

Standard VHDL rules for visibility and scoping apply
to overloaded entities, which can sometimes affect
how they are resolved.

Google AI Overview : vhdl overloading

Subprogram Overloading

Multiple functions or procedures can share the same name.

The VHDL compiler distinguishes between them
based on the number, types, and order of their parameters,
and for functions, also by their return type.

This allows you to create generic operations
that work on various data types
without needing unique names
for each type-specific implementation.

Google AI Overview : vhdl overloading

Subprogram Overloading

```
-- Example of procedure overloading

procedure add(a, b: in integer; result: out integer) is
begin
        result := a + b;
end procedure add;

procedure add(a, b: in real; result: out real) is
begin
        result := a + b;
end procedure add;
```

Google AI Overview : vhdl overloading

Operator Overloading (1)

VHDL allows you to redefine
the behavior of existing operators
(like +, -, *, /, and, or, etc.) for user-defined data types.

This is achieved by declaring a function
whose designator is the operator symbol.

Operator overloading makes code more intuitive
when working with custom data types,
as you can use familiar operator symbols
instead of explicit function calls.

Google AI Overview : vhdl overloading

Operator Overloading (2)

```
-- Example of operator overloading for a custom 'bit_vector' type

function "+" (left, right: in bit_vector) return bit_vector is
        variable temp_result: bit_vector(left'range);
begin
        -- Logic to perform bit-wise addition
        for i in left'range loop
                -- Simple XOR for illustration
                temp_result(i) := left(i) xor right(i);
        end loop;
        return temp_result;
end function "+";
```

Google AI Overview : vhdl overloading

Operator Overloading (3)

Function Declaration: Operator overloading is achieved
by declaring a function whose designator is an operator symbol.

This function defines the specific behavior of the operator
for the given operand types.

```
function "operator_symbol" (p1_name: p1_type; p2_name: p2_type) return result_type is
    -- Function body defining the operator's behavior
  end function "operator_symbol";
```

Existing Operators Only:
VHDL allows overloading of existing predefined operators
(e.g., +, -, *, /, =, <, >, and, or, not, etc.).

It does not permit the creation of entirely new operator symbols.

Google AI Overview : VHDL Operator Overloading

Operator Overloading (4)

Type-Dependent Behavior:

The VHDL compiler determines
which overloaded operator function to use
based on the types of the operands involved in an expression.

If multiple *overloaded functions* exist for the same operator,
the compiler selects the one whose parameter types match the operand types.

Enhancing Readability and Usability:

Overloading operators can make VHDL code
more intuitive and readable, especially
when working with complex or user-defined data types,
as it allows for natural-looking expressions.

Google AI Overview : VHDL Operator Overloading

Operator Overloading (5)

Common uses include

defining arithmetic operations
for custom numeric types
(e.g., std_logic_vector interpreted
as unsigned or signed numbers,
often handled by packages
like ieee.numeric_std),

or defining comparison operations
for complex data structures.

Google AI Overview : VHDL Operator Overloading

Operator Overloading (6)

Important Note:

When using overloaded operators,
especially with std_logic_vector types,

it is crucial to use *standard* and
well-defined packages
like ieee.numeric_std
for arithmetic operations,

rather than *non-standard* or
deprecated packages
like std_logic_unsigned
or std_logic_arith.

Google AI Overview : VHDL Operator Overloading

Overloaded operators (1)

```
-------------------------------------------------------------------
-- common subtypes
-------------------------------------------------------------------
SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')
SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')
SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')

-------------------------------------------------------------------
-- overloaded logical operators
-------------------------------------------------------------------

FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xnor" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;
```

```
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
);
```

X01
X01Z

```
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
);
```

UX01
UX01Z

https://portal.cs.umbc.edu/help/VHDL/packages/std_logic_1164.vhd

Overloaded operators (2)

```
-------------------------------------------------------------------
  -- vectorized overloaded logical operators
  -------------------------------------------------------------------
  FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
  FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

  FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
  FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

  FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN std_logic_vector;
  FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

  FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
  FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

  FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
  FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
```

https://portal.cs.umbc.edu/help/VHDL/packages/std_logic_1164.vhd

Overloaded Operator AND (1)

```
FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
  -- pragma built_in SYN_AND
  -- pragma subpgm_id 184
  BEGIN
  --synopsys synthesis_off
     RETURN and_table(l, r));
  --synopsys synthesis_on
  END "and";
```

https://portal.cs.umbc.edu/help/VHDL/packages/std_logic_1164.vhd

Overloaded Operator AND (2)

```
-------------------------------------------------------------------
-- and
-------------------------------------------------------------------
FUNCTION "and"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
    -- pragma built_in SYN_AND
        -- pragma subpgm_id 198
    --synopsys synthesis_off
    ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
    ALIAS rv : std_logic_vector ( 1 TO r'LENGTH ) IS r;
    VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    --synopsys synthesis_on
BEGIN
    --synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments of overloaded 'and' operator " &
                    "are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := and_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
    --synopsys synthesis_on
END "and";
```

```
-------------------------------------------------------------------
FUNCTION "and"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
    -- pragma built_in SYN_AND
        -- pragma subpgm_id 191
    --synopsys synthesis_off
    ALIAS lv :std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
    ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
    VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    --synopsys synthesis_on
BEGIN
    --synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments of overloaded 'and' operator " &
                    "are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := and_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
    --synopsys synthesis_on
END "and";
-------------------------------------------------------------------
```

https://portal.cs.umbc.edu/help/VHDL/packages/std_logic_1164.vhd

Overloaded Operator AND (3)

```
-------------------------------------------------------------------
-- tables for logical operations
-------------------------------------------------------------------

--synopsys synthesis_off
-- truth table for "and" function
CONSTANT and_table : stdlogic_table := (
--      ----------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -        |   |
--      ----------------------------------------------------
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
);
```

```
type stdlogic_table is array(STD_ULOGIC, STD_ULOGIC)
      of STD_ULOGIC;
```

https://portal.cs.umbc.edu/help/VHDL/packages/std_logic_1164.vhd

std_ulogic (Unresolved Logic)
std_logic (Standard Logic)
a resolved subtype of std_ulogic.

The subpgm_id pragma assigns
a *unique identification number*
to a subprogram (a procedure or function).

This is typically used by synthesis tools
to differentiate between overloaded subprograms
that have the same name
but different parameter types.

Pragma built_in SYN_AND (1)

A pragma built_in SYN_AND command
does not exist in standard VHDL and
is not part of any built-in VHDL package.

It is a non-standard, vendor-specific **compiler directive**
used by some *older* synthesis tools.

These "**pragmas**" are special comments
that instruct the synthesis tool on
how to interpret certain parts of the code.

How vendor-specific pragmas work

Pragmas like built_in SYN_AND are added
as comments to a VHDL function declaration.

When the synthesis tool encounters this pragma,
it ignores the VHDL code in the function body
and substitutes it with a pre-optimized,
"hard-wired" hardware representation of the logic.

This was done to improve synthesis run times, which were
significantly slower on older toolsets.

```
-------------------------------------------------------------------
-- and
-------------------------------------------------------------------
FUNCTION "and"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
    -- pragma built_in SYN_AND
        -- pragma subpgm_id 198
    --synopsys synthesis_off
```

Google AI Overview : pragma built_in SYN_AND in vhdl package

Pragma built_in SYN_AND (2)

How to implement logic in standard VHDL

Instead of using vendor-specific pragmas,
you should write standard, synthesizable VHDL.

A synthesizer will automatically recognize the intent of the logic
and produce an optimized gate-level representation.

Example: Implementing a standard AND function

A standard AND operation is a fundamental and
built-in operator in VHDL that does not require a special pragma.

For boolean types, the and operator is predefined.

For std_logic and std_logic_vector types,
you must use the ieee.std_logic_1164 package,
which defines the and operator for these types.

Google AI Overview : pragma built_in SYN_AND in vhdl package

Bitwise AND Function (1)

how to implement a function
that performs a standard bitwise AND
without any special directives.

```
-- Include the standard logic package, which defines the
AND operator for std_logic_vector
library ieee;
use ieee.std_logic_1164.all;
```

```
-- A package to hold the function
package my_logic_pkg is

  -- Declare a function that performs
  -- a bitwise AND on two std_logic_vectors
  function bitwise_and (
    a : std_logic_vector;
    b : std_logic_vector
  ) return std_logic_vector;

end package my_logic_pkg;
```

```
package body my_logic_pkg is

  function bitwise_and (
    a : std_logic_vector;
    b : std_logic_vector
  ) return std_logic_vector is
    -- The result vector must have a defined size
    variable result : std_logic_vector(a'range);
  begin
    -- Iterate through the vectors and
    -- perform the AND operation
    for i in a'range loop
      result(i) := a(i) and b(i);
    end loop;
    return result;
  end function bitwise_and;

end package body my_logic_pkg;
```

Google AI Overview : pragma built_in SYN_AND in vhdl package

Bitwise AND Function (2)

```
-- Example usage of the function within an architecture
use work.my_logic_pkg.all;

entity and_example is
  port (
    input_a  : in  std_logic_vector(3 downto 0);
    input_b  : in  std_logic_vector(3 downto 0);
    output_y : out std_logic_vector(3 downto 0)
  );
end entity and_example;

architecture RTL of and_example is
begin
  -- Use the standard VHDL operator to perform the AND
  output_y <= input_a and input_b;

  -- The function from the package can also be used,
  – but is not necessary
  -- output_y <= bitwise_and(input_a, input_b);
end architecture RTL;
```

Built-in operator:
The and operator in VHDL works directly
on std_logic_vector types
without needing a for-loop or special directives.

Function vs. direct assignment:
While you could perform the AND
in a single assignment statement
(e.g., Result <= A and B;),
using a function is a good practice
for modularity and reusability.

Vector length:
This method requires that both input vectors
have the same length and index range.
If the lengths differ, VHDL will report an error.

std_logic library:
The std_logic_1164 library is required
for the std_logic_vector type.

Google AI Overview : pragma built_in SYN_AND in vhdl package

Bitwise AND Function (3)

Special pragmas are not necessary
for optimized AND logic.

Standard VHDL operators are sufficient.

Definitions for and, or, xor, and other logical operators
for std_logic and std_logic_vector types
are in the ieee.std_logic_1164.all package.

Ensure portability
by writing clear, standard-compliant VHDL.

This also helps with correct interpretation
by modern synthesis tools.

Google AI Overview : pragma built_in SYN_AND in vhdl package

Conversion functions using numeric_std (1)

To convert std_logic_vector in VHDL, you typically use functions from the numeric_std package, which is the recommended standard for arithmetic operations. These functions allow you to convert between std_logic_vector, integer, signed, and unsigned types.

Common Conversion Functions Using numeric_std

Here are the most widely used conversion functions:

Microsoft Copilot : std_logic_vector converting functions

Conversion functions using numeric_std (2)

From std_logic_vector to Numeric Types

To Unsigned:
unsigned_val := to_unsigned(integer_val, vector_length);

To Signed
signed_val := to_signed(integer_val, vector_length);

To integer
int_val := to_integer(unsigned(std_logic_vector_val));

From Numeric Types to std_logic_vector

Unsigned to std_logic_vector:
std_logic_vector_val := std_logic_vector(unsigned_val);

Signed to std_logic_vector:
std_logic_vector_val := std_logic_vector(signed_val);

Microsoft Copilot : std_logic_vector converting functions

Pure and Impure Functions (1)

Functions can be either pure (which is default) or impure.

Pure functions always return the same value
for the same set of parameters.

Impure functions may return different values
for the same set of parameters.

Additionally, an impure function may have „side effects”,
like updating objects outside of their scope,
which is not allowed for pure functions.

https://peterfab.com/ref/vhdl/vhdl_renerta/mobile/source/vhd00032.htm

Pure and Impure Functions (2)

The function declaration can be preceded
by an optional reserved word pure or impure,
denoting the character of the function.

If the reserved word is omitted
it is assumed by default that the function is pure.

The function name, which appears after the reserved word function,
can be either an identifier or an operator symbol
(if the function specifies the operator).

Specification of new functions for existing operators
is allowed in VHDL and is called operator overloading.

https://peterfab.com/ref/vhdl/vhdl_renerta/mobile/source/vhd00032.htm

Pure and Impure Functions (2)

a pure function always returns the same output
for the same input and has no side effects,

an impure function may access or modify
external signals or variables,
making its behavior potentially non-deterministic.

MS Copilot :in vhdl, pure, impure function

Pure Functions

A pure function in VHDL is
deterministic and side-effect-free.

It adheres to these rules:

- only uses its input parameters to compute the result.
- cannot read or write to signals or variables outside its scope.
- always returns the same result for the same input values.
- can be used in constant declarations, concurrent statements, and synthesis.

```
function add(a, b: integer) return integer is
begin
  return a + b;
end function;
```

MS Copilot :in vhdl, pure, impure function

Impure Functions

An impure function may read from or write to objects outside its parameter list.

- declared with the impure keyword:
- van access signals, variables, or files not passed as parameters.
- may produce different results for the same inputs
- typically used in testbenches, file I/O, or random number generation.
- not synthesizable in most cases.

```
impure function read_signal return integer is
begin
  return some_signal;  -- Accesses external signal
end function;
```

MS Copilot :in vhdl, pure, impure function

Pure and Impure Functions (2)

Feature	Pure Function	Impure Function
Access external data	No	Yes
Deterministic	Yes	Not guaranteed
Synthesis-friendly	Usually	Rarely
Use in constants	Yes	No
Side effects	None	Possible (e.g., file I/O, signal read)

MS Copilot :in vhdl, pure, impure function

References

[1] http://en.wikipedia.org/
[2] J. V. Spiegel, VHDL Tutorial,
http://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html
[3] J. R. Armstrong, F. G. Gray, Structured Logic Design with VHDL
[4] Z. Navabi, VHDL Analysis and Modeling of Digital Systems
[5] D. Smith, HDL Chip Design
[6] http://www.csee.umbc.edu/portal/help/VHDL/stdpkg.html
[7] VHDL Tutorial - VHDL onlinewww.vhdl-online.de/tutorial/