VHDL Packages: *standard*

- The *standard* package defines all of the types and associated operator functions for the 'predefined' types
  - Chapter 14 of the IEEE LRM (language reference manual) has a listing of these types
  - Examples of predefined types are BOOLEAN, BIT, CHARACTER, REAL, INTEGER, TIME, etc.
- In Digital Systems, you used the types defined in the IEEE 1164 standard logic package (std_logic, std_logic_vector, etc)
  - VHDL allows users to define their own types, and the *std_logic* types are better suited for digital logic simulation than the predefined types found in the *standard* package

Enumerated Types

An enumerated type is one in which the type definition includes all of the allowed literals for that type

```
type BOOLEAN is (FALSE, TRUE);
```

Boolean types can only take on these two literal values.

The IEEE LRM lists the functions defined for this types: and, or, nand, xor, etc….. See the LRM for a complete list.

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

Type bit was initially provided to perform digital logic simulation but it has been replaced by the 'std_logic' type (more on this later).

Range Types

Types INTEGER and REAL are range types in which the types take on all values within a range:

```
type INTEGER is range -2147483648 to 2147483647;
 type REAL is range -1.7e38 to 1.7e38;
```

The actual range is implementation dependent – all VHDL simulators are supposed to support at least a 32-bit range for integers and single precision (32-bit) IEEE floating point range.

Predefined Type *time*

```
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;
```

Units can also be defined for types – each unit is defined in terms of another unit.

Initial Values of Variables, Signals

Unless overridden in the signal or variable declaration, the initial value is the 'leftmost' value of the type

```
signal  a_boolean: boolean;        <-- Initial value = FALSE
signal  a_integer: integer;        <-- Initial value = -2147483648
signal  a_float: float;            <-- Initial value = -1.7e38
signal  a_bit : bit;               <-- Initial value = '0'

signal another_integer: integer := 0;
                                   <-- Initial value = 0
```


Subtypes

A subtype defines a new type that has a restricted set of literals or values from its associated type.

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
```

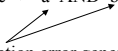
`INTEGER'HIGH` specifies the maximum-valued integer.

Mixing Types

VHDL is a *strongly typed* language. This means that all variables/signals must have declared types (there are no default assumptions) and that you cannot mix types in an expression except for subtypes of the same type.

```
signal  a : std_logic := '0';
signal  b,c : bit :=  '0';

c <= a  AND  b;
```



Compilation error generated - signals 'a' and 'b' are different types. Error message will be:

```
No feasible entries for infix op: "and"
Type error resolving infix expression.
```


Type Conversion Functions

For most cases, need type conversion functions to convert from one type to another. These functions are usually defined in the same package that defines the type. You can also write additional functions yourself.

```
signal  a : std_logic := '0';
signal  b,c : bit :=  '0';

   c <=  TO_BIT(a)  AND  b;
```

TO_BIT is a type conversion function provided in the IEEE 1164 package that converts a *std_logic* type to a *bit* type.

We will talk more about explicit type conversions later.

Number Conversions

For integers and reals, can use explicit conversions:

```
signal a: integer;
signal b: real;

-- converts real to integer
-- rounding rules implementation dependent
a < = integer(b);
-- converts integer to real
b <= real(a);
```

Explicit conversions allowed between *closely related* types (see LRM for this definition, in practice mainly used for integer/real conversion).

Time Conversions

Time to Natural:

```
Natural_var := Time_var  / 1 ns;
```

Natural to Time:

```
Time_var :=  Natural_var * 1 ns
```


Vectors

A vector type is a linear array where the elements are the same type.

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;

   variable  a_vec : bit_vector(0 to 7);
   variable  b_vec : bit _vector(7 downto 0);
```

Note that in the type declaration of type BIT_VECTOR the array range was *unconstrained* ('range <>').

When we declare a variable or signal of that type, must specify the range.

Multi-Dimensional Vectors

To declare a multi-dimensional vector, can do the following:

```
type bytes is array (NATURAL range <> ) of
     bit_vector(7 downto 0);

   variable  some_memory : bytes(0 to 1023);
```

Only one dimension of multi-dimensional vector type can be unconstrained, the other dimensions must be fixed sizes.

It would be a syntax error to declare:

```
type array_2d is array (NATURAL range <>  ) of
      array (natural range <> ) of BIT;
```

Type STRING

```
type STRING is array (POSITIVE range <> ) of CHARACTER;

    variable  a_memory : string(1 to 5) := "Hello";
```

Note that since the range on STRING is of type POSITIVE, then first array index of string will be 1, not 0.

The following will generate a syntax error because '0' is not of type POSITIVE:

```
variable  a_memory : string(0 to 4) := "Hello";
```


Assertion Statements

The SEVERITY_LEVEL type in the *standard* package is used with assertion statements.

```
type SEVERITY_LEVEL is (NOTE, WARNING,ERROR, FAILURE);
```

An assertion statement is a concurrent statement and can appear outside or inside of a process. Assertion statements check a condition, and print a message to the console if the condition is false:

```
ASSERT  condition
  REPORT  some_string
  SEVERITY  some_severity_level;
```

An example:

```
ASSERT  (expected_var = actual_var)
  REPORT  "Incorrect result found!"
  SEVERITY  ERROR;
```

In Modelsim, can mask/unmask assertions of a particular severity.

VHDL Packages: *textio*

Assertion statements are the only method available in the standard package for printing strings to the screen.

Package *textio* provides functions for reading VHDL types from either standard input or a text file, and functions for writing VHDL types as text to a standard output or a text file.

Unfortunately, the capability is *textio* is primitive compared to IO functions in other languages. We will not do much with text input/output in this class.

To use the functions in this package, the following must be included in the VHDL file:

```
LIBRARY std;
use std.textio.all;
```


Types defined in *textio*

```
-- A LINE is a pointer to a String value
type LINE is access string;
-- file of variable length ASCII records
type TEXT is file of string;

type SIDE is (right, left);  -- for justifying output data
subtype WIDTH is natural; -- used for widths of output fields


-- standard input (the keyboard usually)
 file input : TEXT open read_mode is "STD_INPUT";
-- standard output (the console usually)
 file output : TEXT open write_mode is "STD_OUTPUT";
```


Access Types

An access type is the method by which you declare a pointer to a type.

```
type LINE is access string;

 variable ll: line;

  ll = new String'("a new string");
  deallocate(ll);
```

new used for dynamic allocation, returns a pointer

deallocate used to free memory associated with an dynamically allocated object

readline, file_open Procedures

The *readline* procedure reads a line of text from a file into a line variable.

```
procedure READLINE (file F: TEXT; L: out LINE);

variable ll: line;

 readline(input, ll);
```

Read a line of text from file 'input' which is standard input, value returns in *ll* .

To read from a file other than standard input, use 'file_open'.

```
file  inputfile: text;
 process
   variable ll:string;
   begin
     file_open(inputfile, string'("file.txt"), READ_MODE);
      readline(inputfile, ll);
```

File object

filename

Open for reading

read Procedures

The various *read* procedures reads a VHDL object from a line variable. Each successful read modifies the line variable by removing characters, so successive reads to the same line variable resumes where the last left off.

Assume the first line of 'file.txt' contains the line:

56 4.7 20 ns

This has an integer, real, and time type

```
file  inputfile: text;
 process
   variable ll:line; variable a_time: time;
   variable a_int:integer; variable a_real:real;
   begin
     file_open(inputfile, string'("file.txt"), READ_MODE);
     readline(inputfile, ll);
     read (ll, a_int);
     read (ll, a_real);
     read (ll, a_time);
```

Each read picks up where the last read left off.

Detecting formatting errors

Another version of the read procedure is provided for detecting formatting errors:

```
procedure
  READ(L:inout LINE; VALUE:out sometype; GOOD:out Boolean);
```

The variable GOOD will return FALSE if the expected type was not found on the line. Variable L is not modified if the read fails.

```
variable ok:boolean;
 begin
   file_open(inputfile, string'("file.txt"), READ_MODE);
   readline(inputfile, ll);
   read (ll,a_time, ok);
   assert ok
     report "time type not found!!"
     severity error;
```

The first object on the line is an integer and does not have a time unit (I.e. 'ns'), so this will fail.

End of File, Line length

The code below uses the *endfile* function to detect end of file. The *'length* attribute can be used to detect if a line has no characters in it.

free space from previous line. Does not matter if *ll* is initially empty.

```
while (not endfile(stimulusfile)) loop
   deallocate(ll);
   readline(stimulusfile, ll);
   if (ll'length = 0) then -- for blank lines
     next;
   end if;
   --- perform processing on this line
   ---
end loop;
```

Blank line check

endfile returns a boolean – will be TRUE if end of file has been reached.

Opening Files

Must be careful to open a file only once. If the process has no sensitivity list and the process ends with a no-argument *wait* statement, it is easy – just open it at the start of the process:

```
process
 begin
  file_open(inputfile, string'("file.txt"), READ_MODE);
  --- other stuff
  wait;
 end process;
```

A process with a sensitivity list will be triggered multiple times. Use a boolean variable to ensure *file_open* executed only once:

```
process (some_signal, another_signal)
 variable init:boolean;
 begin
   if (not init) then
    file_open(inputfile, string'("file.txt"), READ_MODE);
    init := TRUE;
   end if;
  --- other stuff
  end process;
```


Procedure/Function Overloading

The LRM shows multiple *read* procedures – they are all called *read* with the only difference being the type and number of arguments.

Which read procedure is selected by the compiler is determined by the type and number of arguments passed to it. This is called *operator overloading* (also used in C++).

When a new type is defined, a new read procedure must be written for that type if ASCII string conversion for the type is desired. Typically, this read procedure is defined in the package that defines that type.

More on *file_open*

Chapter 3 of the LRM discusses file operations in detail.

Default is to open for reading

```
procedure FILE_OPEN (file F: FT;
                     fname: in STRING,
                     open_type: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_OPEN (status: out FILE_OPEN_STATUS;
                     file F: FT;
                     fname: in STRING,
                     open_type: in FILE_OPEN_KIND := READ_MODE);
```

Use this version to check for errors on open. Status will be either *open_ok*, *status_error*, *name_error*, or *mode_error*.

FILE_OPEN_KIND can be either *read_mode*, *write_mode* or *append_mode*.

write, *writeline* Procedures

Use the *write* procedure to write the ASCII representation of a VHDL type to a line variable. Use the *writeline* procedure to write a line variable to a file.

```
file  outputfile: text;
 process
   variable ll:line;
   variable a_time: time := 10 ns;
   variable a_int:integer := 53;
   variable a_real:real := 34.7;
   begin
     file_open(outputfile, string'("file.txt"), WRITE_MODE);
     write (ll, a_int); write (ll, string'(" "));
     write (ll, a_real); write (ll, string'(" "));
     write (ll, a_time, RIGHT, 19, NS);
     write (ll, string'(LF));
     writeline (outputfile, ll);
     --- other operations..
```

Add whitespace between values

Justification, field width, time units

Newline character

More on *write*

For all predefined types except *time*, the *write* procedure is defined as:

```
procedure WRITE (L :inout LINE;  VALUE: in sometype,
                 JUSTIFIED: in SIDE:=RIGHT;
                 FIELD: in WIDTH := 0);
```

For the time type, the *write* procedure is defined as:

```
procedure WRITE (L :inout LINE;  VALUE: in sometype,
                 JUSTIFIED: in SIDE:=RIGHT;
                 FIELD: in WIDTH := 0;
                 UNIT: in TIME := ns);
```


Aside: Functions vs Procedures

We will study procedure and function syntax in more detail later. For now, be aware that a procedure can modify a parameter if its mode is either *inout* or *out*, and that a procedure does not return a value.

```
procedure
  READ(L:inout LINE; VALUE:out sometype; GOOD:out Boolean);
```

Reads values from L and also modifies it by removing characters from L. Also modifies parameters VALUE, GOOD.

A function can never modify its parameters (mode of function parameters always *in*), and will always return a value:

```
function ENDFILE (file F: TEXT) return BOOLEAN;
```


stim_readfile.vhd

Can look at *stim_readfile.vhd* in the exam1 library for examples of reading input values from a file.

CAUTION! *stim_readfile.vhd* is somewhat complicated because it supports an input format that requires some character-by-character processing. It reads input from *src/stimfile.stm* .

```
# stimulus file for D-latch gate
#          D  G  R
## FF reset
0 ns  ==>  0  0  0
## negate reset
10 ns ==>  0  0  1
## D had no effect, clock low
20 ns ==>  1  0  1
```

If line starts with '#', then it is a comment line.

Must read this as complete string or character-by-character.

Time at which vector is to be applied, values for vector.

Simplifying *stim_readfile.vhd*

Could simplify *readfile* architecture if each line of the input file used the following format (assume values are either '1' or '0'):

```
time_val  d_val   g_val   r_val
```

Can read the value into a *bit* type, then convert the bit type into a std_logic type when applying the value.

Code for doing this is shown on the next page.

There is no error checking in this code. Possible errors would be: file does not exist, time_val is less than previous time_val (can't go backward in time!), incorrect formatting of input values (not a bit type), empty lines).


```
 file stimulusfile: text;
process
 variable time_val: time;
 variable inbit: bit;
 variable ll: line;
 begin
  file_open(stimulusfile, filename);
  while (not endfile(stimulusfile)) loop
     deallocate(ll);
     readline(stimulusfile, ll);
     read(ll,time_val);
     wait for (time_val - NOW);
     read(ll,inbit);
     d <= To_stdulogic(inbit);
     read(ll,inbit);
     g <= To_stdulogic(inbit);
     read(ll,inbit);
     q <= To_stdulogic(inbit);
  end loop;
  wait;
 end process;
```

Suspends process until NOW equals the time value just read.

Apply d, g, r signals at this time.

to_stdulogic converts a bit type to a std_logic type.

Library *utilities*, Package *std_utils*

In the *utilities* directory are several packages that we will make use of over the semester.

The *std_utils* package contains some type conversion functions between pre-defined types.

I will not cover these functions/procedures in detail, you might want to peruse this package.

As we look at examples that uses a particular package from the *utitilies* directory, I will only cover a particular procedure or function from a package if it illustrates some functionality that has not been demonstrated elsewhere. You should look at the contents of the packages as the semester progresses; you will find some of them useful.