Hardware Modeling [VU] (191.011)
– WS24 –
Advanced Testbenches

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

In this lecture we will discuss how testbenches more powerful and with better scalability than the ones we considered so far can be written. In particular, we will look into file I/O, how random values can be generated and the standard environment package.

Hardware Modeling [VU] (191.011)
– WS24 –
Advanced Testbenches

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

Modified: 2025-03-08, 00:15 (b25118c)

More powerful testbenches?

Let us start by addressing the elephant in the room. At this point in the course you have already successfully written testbenches and verified that your designs are correct. So why should you care about advanced testbenches? Is there a need for more powerful testing?

Motivation

- More powerful testbenches?

More powerful testbenches?
Modern designs can be highly complex (e.g., hundreds of I/O pins)
Manually generating and applying stimuli infeasible / impossible
The per-transistor cost of testing is higher than that of designing

Well, basically there is a plethora of reasons why more powerful testbenches are required. Scalability is one obvious problem of the testbenches you have written so far. While they work for many of the rather small examples you deal with during this course, they won't work for complex designs containing hundreds of I/O pins, concurrently handling different interfaces and transmission protocols. Manually applying stimuli on the level of abstraction you did so far is simply not feasible and too error-prone for big designs. With the complexity rising in lockstep with the continuing shrinking of features sizes, this is only becoming worse. As a result, the verification of hardware designs is very demanding and expensive. To give you some vague concept of costs involved, the per-transistor verification costs of a modern chip are higher than the design costs. Therefore, enabling verification engineers and designers to efficiently verify big and complex designs is vital. But how can this be achieved?

Motivation

- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing

- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing
- More powerful testbenches and automation!

Well, ultimately, this is where the more powerful testbenches come into play, that are automated and act on higher levels of abstraction. In this lecture we will introduce language support for some first steps in that direction.

Motivation

- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing
- More powerful testbenches and automation!

- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing
- More powerful testbenches and automation!
  - File I/O

We will start by looking into file handling in VHDL, enabling us to generate inputs for our designs using external tools, and to provide the results of a simulation for postprocessing.

Motivation

- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing
- More powerful testbenches and automation!
  - File I/O

- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing
- More powerful testbenches and automation!
  - File I/O
  - Randomized testing

We will then continue with discussing how random values can be generated in our testbenches, allowing us to apply random test inputs.

Motivation

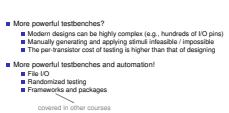
- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing
- More powerful testbenches and automation!
  - File I/O
  - Randomized testing

- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing
- More powerful testbenches and automation!
  - File I/O
  - Randomized testing
  - Frameworks and packages

Finally, there also exist powerful frameworks and useful packages for VHDL simulations.

Motivation

- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing
- More powerful testbenches and automation!
  - File I/O
  - Randomized testing
  - Frameworks and packages



However, for an introduction to such frameworks be referred to other courses. In this lecture we will only briefly introduce one particular VHDL package that can come in handy in testbenches.

Motivation

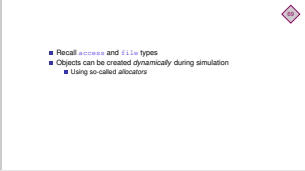
- More powerful testbenches?
  - Modern designs can be highly complex (e.g., hundreds of I/O pins)
  - Manually generating and applying stimuli infeasible / impossible
  - The per-transistor cost of testing is higher than that of designing
- More powerful testbenches and automation!
  - File I/O
  - Randomized testing
  - Frameworks and packages

covered in other courses

Let us begin by discussing how file I/O works in VHDL. However, before we do that, we need to properly introduce two VHDL types we mentioned but neglected so far. In particular, we will now consider access and file types, beginning with the former.

Access Types

- Recall `access` and `file` types



One feature of VHDL we did not mention so far is the possibility to dynamically create objects during simulations. This can be done using so-called allocators, which allocate the required memory and an object of the respective type. We will see an example soon.

Access Types

- Recall `access` and `file` types
- Objects can be created *dynamically* during simulation
  - Using so-called *allocators*

Recall access and file types
Objects can be created *dynamically* during simulation
Using so-called *allocators*
No identifier referring to them

However, naturally objects that are created dynamically cannot be assigned static identifiers that refer to them. Instead, the allocators return a so-called ”access value”.

Access Types

- Recall `access` and `file` types
- Objects can be created *dynamically* during simulation
  - Using so-called *allocators*
  - No identifier referring to them

These access values are of distinct types, so-called access types. As the name suggests, access types provide access to objects of a certain type. The syntax for declaring such an access type is straightforward and shown on the slide. `TYPE_NAME` is the name of the declared access type. The `DESIGNATED_TYPE` defines the type of objects the access type grants access to.

Access Types

- Recall `access` and `file` types
- Objects can be created *dynamically* during simulation
  - Using so-called *allocators*
  - No identifier referring to them
- Access types provide access to objects of certain type

```
type TYPE_NAME is access DESIGNATED_TYPE;
```

Note that objects of an access type must only be of the class `variable`. Thus, it is not possible to have a signal of such a type.

Access Types

- Recall `access` and `file` types
- Objects can be created *dynamically* during simulation
  - Using so-called *allocators*
  - No identifier referring to them
- Access types provide access to objects of certain type

```
type TYPE_NAME is access DESIGNATED_TYPE;
```

  - Can only be used for `variable`

An object of an access type is initially assigned the value null, which means it is not designating any object at all. As mentioned earlier, access values, which actually designate an object, can be assigned to it using an allocator, as shown on the slide. The allocator is the combination of the keyword ”new” and a type name, as shown on the slide via the example of an access type for integer. Note that the allocated object is initialized with the default value of the respective type.

Access Types

- Recall `access` and `file` types
- Objects can be created *dynamically* during simulation
  - Using so-called *allocators*
  - No identifier referring to them
- Access types provide access to objects of certain type

```
type TYPE_NAME is access DESIGNATED_TYPE;
```

  - Can only be used for `variable`
  - Default value `null`; assigned using allocators

```
int_ptr := new integer;
```

In order to access the value of the object referred to by an access type variable, the `all` keyword must be used. The slide shows this for both writing and reading.

Access Types

- Recall `access` and `file` types
- Objects can be created *dynamically* during simulation
  - Using so-called *allocators*
  - No identifier referring to them
- Access types provide access to objects of certain type

```
type TYPE_NAME is access DESIGNATED_TYPE;
```

  - Can only be used for `variable`
  - Default value `null`; assigned using allocators

```
int_ptr := new integer;
```

  - Access to value of designated type object via `all`

```
int_ptr.all := 42; print(to_string(int_ptr.all));
```

If access types and allocators appear a bit strange initially, you can draw a comparison to Java's references to objects created via the new operator.

Access Types

- Recall `access` and `file` types
- Objects can be created *dynamically* during simulation
  - Using so-called *allocators*
  - No identifier referring to them
- Access types provide access to objects of certain type

```
type TYPE_NAME is access DESIGNATED_TYPE;
```

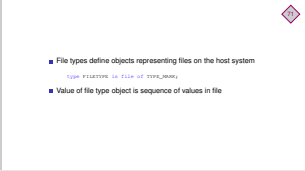
  - Can only be used for `variable`
  - Default value `null`; assigned using allocators
    `int_ptr := new integer;`
  - Access to value of designated type object via `all`
    `int_ptr.all := 42; print(to_string(int_ptr.all));`
- Similar to object references in Java and the `new` operator

Let us now discuss a rather exotic feature of VHDL, namely file types and the `file` class. If you think about it, having a dedicated handling of files via the type system, instead of using existing types, is something rather rare. This is a remnant of VHDL originally being a language for specifying hardware rather than a programming language. The purpose of objects of the `file` class is representing files on the host system where the simulation runs. File types are used to declare the kind of file. The slide shows the syntax for declaring such a type.

File Types

- File types define objects representing files on the host system

```
type FILETYPE is file of TYPE_MARK;
```



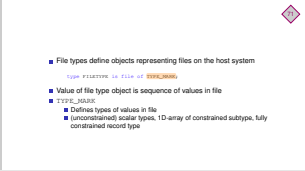
The value of a file object is the sequence of values contained in the file on the host system in the same order as they occur in the file.

File Types

- File types define objects representing files on the host system

```
type FILETYPE is file of TYPE_MARK;
```

- Value of file type object is sequence of values in file



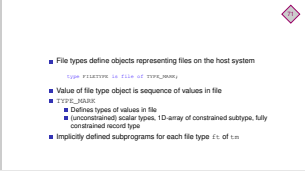
The type of the values in the file is defined by `TYPE_MARK` in the file type declaration. It can be a scalar type, even unconstrained, an unconstrained one-dimensional array of a constrained subtype or a record type of fully constrained elements.

File Types

- File types define objects representing files on the host system

```
type FILETYPE is file of TYPE_MARK;
```

- Value of file type object is sequence of values in file
- `TYPE_MARK`
  - Defines types of values in file
  - (unconstrained) scalar types, 1D-array of constrained subtype, fully constrained record type



For each file type a set of subprograms for handling respective files is implicitly defined. We will discuss them next, referring to the respective file type as `ft` and the type mark as `tm`.

File Types

- File types define objects representing files on the host system

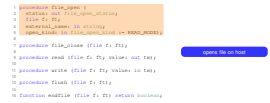
```
type FILETYPE is file of TYPE_MARK;
```

- Value of file type object is sequence of values in file
- `TYPE_MARK`
  - Defines types of values in file
  - (unconstrained) scalar types, 1D-array of constrained subtype, fully constrained record type
- Implicitly defined subprograms for each file type `ft` of `tm`

On this slide you can see all the implicitly defined subprograms for a file type. We will now briefly discuss each of them, for more details we refer you to the standard.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```



The first subprogram we consider is the `file_open` procedure. As the name suggests, it can be used to open a file on the host system via a file type variable. The subprogram takes four parameters, with the first being optional due to an available overload.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

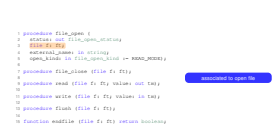
opens file on host

The first is `status`, which is assigned a value of an enumeration type that indicates the result of the file open operation. You can find the particular values in the standard.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

indicates result, optional



The parameter `f` of the file type and the class `file` will be associated with the opened file.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

associated to open file

`external_name` is the path to the file to be opened on the host system.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

name of the host file

Finally, the `open_kind` parameter defines the opening mode. Possible values are `READ_MODE`, `WRITE_MODE` and `APPEND_MODE`, as well as a mode for both, reading and writing, since VHDL 2019.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

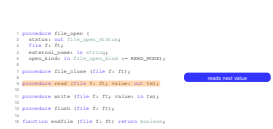
READ_MODE, WRITE_MODE
APPEND_MODE
2019: READ_WRITE_MODE

The `file_close` procedure is pretty self-explanatory, as it simply closes the given file. In general, most simulators will flush and close opened files at the end of the simulation. However, it is still highly recommended to manually close a file for several reasons. First, the standard does not require simulators to close opened files. Second, if the simulation terminates abruptly, for example due to a crash, you risk losing buffered but not yet written data. Furthermore, closing files that are no longer needed is more resource efficient and also mitigates mistakenly opening a still opened file again.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

closes opened file

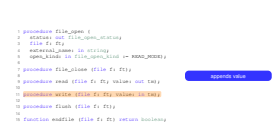


The procedure called `read` also does exactly what its name suggests. That is, it returns the next element of the sequence of values contained in the file via the `value` parameter.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

reads next value

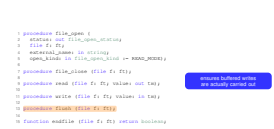


Likewise, the `write` procedure appends the passed `value` to the given file.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

appends value



A call to the `flush` procedure leads to all values previously appended to the file via the `write` procedure to be actually written to the file. It ensures that the effects of previous writes are not just held in a buffer but also actually committed to the file system.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

ensures buffered writes
are actually carried out

Finally, the `endfile` function can be used to determine if further values can be read from a file. It returns false for files in read-only mode if a call to the `read` procedure is able to read another value. For files in write-only mode it always returns true.

File Operations


```
1  procedure file_open (
2    status: out file_open_status;
3    file f: ft;
4    external_name: in string;
5    open_kind: in file_open_kind := READ_MODE);
6
7  procedure file_close (file f: ft);
8
9  procedure read (file f: ft; value: out tm);
10
11 procedure write (file f: ft; value: in tm);
12
13 procedure flush (file f: ft);
14
15 function endfile (file f: ft) return boolean;
```

returns true if `read` can read another value

Types and subprograms for formatted operations on text files

As being restricted to reading and writing single values can be a bit cumbersome, the `TextIO` package provides type and subprogram declarations that allow formatted operation on text files.

Text IO Package

- Types and subprograms for formatted operations on text files

The package essentially revolves around two types called `line` and `text`. Their declarations are shown on the slide.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types

```
type line is access string;
type text is file of string;
```

The purpose of the access type `line` is to provide a dynamically resizable buffer for strings. We will see this in use later.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types

```
type line is access string;  -- dynamically resizable buffer
type text is file of string;
```

The other type, `text`, is a file type with `string` as type mark.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

We want to stress that the `TextIO` package is solely for text-based files. Hence, it is not possible to read or write binary data using it. This becomes clear if we observe that both types are based on strings.

Text IO Package

- Types and subprograms for formatted operations on **text files**
- Revolves around two new types IEEE SA OPEN

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

The package further declares subprograms to read from and write to `line` buffers.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

- Subprograms for formatted manipulation of `line` buffers

```
procedure read(l: inout line; value: out <type>; good: out boolean);
procedure write(l: inout line; value: in <type>;
                justified: in side:=right; field: in width:=0);
```

The `read` procedure reads the line currently held by the given `line` buffer and provides the result via the `value` variable. Note that the procedure only reads the characters from the line until either the end of the line is reached, the amount of read characters surpasses the length of the `value` variable, or an invalid character is encountered. You can find details on this in the standard.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types IEEE SA OPEN

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

- Subprograms for formatted manipulation of `line` buffers

```
procedure read(l: inout line; value: out <type>; good: out boolean);
procedure write(l: inout line; value: in <type>;
                justified: in side:=right; field: in width:=0);
```

The so-called `good` parameter is optional and returns true if the read operation succeeded.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

- Subprograms for formatted manipulation of `line` buffers

```
procedure read(l: inout line; value: out <type>; good: out boolean);
procedure write(l: inout line; value: in <type>;
                justified: in side:=right; field: in width:=0);
```

The `write` procedure can be used to append formatted text to a `line` buffer

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

- Subprograms for formatted manipulation of `line` buffers

```
procedure read(l: inout line; value: out <type>; good: out boolean);
procedure write(l: inout line; value: in <type>;
                justified: in side:=right; field: in width:=0);
```

using the optional `justified` and `field` parameters.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

- Subprograms for formatted manipulation of `line` buffers

```
procedure read(l: inout line; value: out <type>; good: out boolean);
procedure write(l: inout line; value: in <type>;
                justified: in side:=right; field: in width:=0);
```

The `field` parameter can be used to specify a desired field width for the appended string. If the desired field is shorter than the string, the length of the string is instead taken as field width. Therefore, the default value 0 has the effect of the field being exactly as long as the string.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types IEEE SA OPEN

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

- Subprograms for formatted manipulation of `line` buffers

```
procedure read(l: inout line; value: out <type>; good: out boolean);
procedure write(l: inout line; value: in <type>;
                justified: in side:=right; field: in width:=0);
```

Using the `justified` parameter one can then either right or left align the string within this field.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

- Subprograms for formatted manipulation of `line` buffers

```
procedure read(l: inout line; value: out <type>; good: out boolean);
procedure write(l: inout line; value: in <type>;
                justified: in side:=right; field: in width:=0);
```

Furthermore, the `readline` and `writeline` procedures read a line from a file into a buffer, respectively append the content of a buffer to a file as a new line.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types IEEE SA OPEN

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

- Subprograms for formatted manipulation of `line` buffers

```
procedure read(l: inout line; value: out <type>; good: out boolean);
procedure write(l: inout line; value: in <type>;
                justified: in side:=right; field: in width:=0);
```

- Subprograms for reading/writing `line` buffers to file

```
procedure readline(file f: text; l: inout line);
procedure writeline(file f: text; l: inout line);
```

Finally, we want to mention that the `TextIO` package provides further functionality in the form of procedures for reading and writing values of the type `bit_vector` in binary, octal or hexadecimal format. These subprograms can prove quite handy, as overloads for them are defined in various packages such as `std_logic_1164` or `numeric_std`.

Text IO Package

- Types and subprograms for formatted operations on text files
- Revolves around two new types IEEE SA OPEN

```
type line is access string;  -- dynamically resizable buffer
type text is file of string; -- text-file type
```

- Subprograms for formatted manipulation of `line` buffers

```
procedure read(l: inout line; value: out <type>; good: out boolean);
procedure write(l: inout line; value: in <type>;
                justified: in side:=right; field: in width:=0);
```

- Subprograms for reading/writing `line` buffers to file

```
procedure readline(file f: text; l: inout line);
procedure writeline(file f: text; l: inout line);
```

- Further procedures `[BINARY|OCTAL|HEX]_[READ|WRITE]` for multiple types (e.g., `bit_vector`, `std_[u]logic[_vector]`, `[un]signed`)

Let us now have a look at an example where we want to read bytes given in hexadecimal format from a file into a `std_ulogic_vector` variable. The file and its content are shown on the slide.

Read from file


```
1  [...]
2  use std.textio.all;
3  [...]
4  begin
5    main: process is
6      file f : text open READ_MODE is "data.txt";
7      variable l : line;
8      variable x : std_ulogic_vector(7 downto 0);
9    begin
10     while not endfile(f) loop
11       readline(f, l);
12       hex_read(l, x);
13       report to_string(x);
14     end loop;
15     file_close(f);
16     wait;
17   end process;
```

```
1 00
2 11
3 AA
```

The first thing we do is including the `TextIO` package.

Read from file


```
1  [...]
2  use std.textio.all;
3  [...]
4  begin
5    main: process is
6      file f : text open READ_MODE is "data.txt";
7      variable l : line;
8      variable x : std_ulogic_vector(7 downto 0);
9    begin
10     while not endfile(f) loop
11       readline(f, l);
12       hex_read(l, x);
13       report to_string(x);
14     end loop;
15     file_close(f);
16     wait;
17   end process;
```

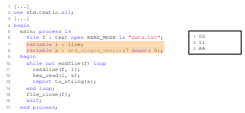
```
1 00
2 11
3 AA
```

Next, we see something we have not mentioned before. Namely, we open the desired file in read mode directly at the declaration of a file instead of opening it later using the procedure previously mentioned.

Read from file


```
1  [...]
2  use std.textio.all;
3  [...]
4  begin
5    main: process is
6      file f : text open READ_MODE is "data.txt";
7      variable l : line;
8      variable x : std_ulogic_vector(7 downto 0);
9    begin
10     while not endfile(f) loop
11       readline(f, l);
12       hex_read(l, x);
13       report to_string(x);
14     end loop;
15     file_close(f);
16     wait;
17   end process;
```

```
1 00
2 11
3 AA
```

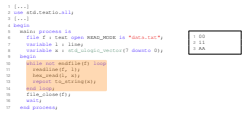


We then create a variable of type `line`, acting as a buffer for the read values and one target variable for the read operation.

Read from file


```
1  [...]
2  use std.textio.all;
3  [...]
4  begin
5    main: process is
6      file f : text open READ_MODE is "data.txt";
7      variable l : line;
8      variable x : std_ulogic_vector(7 downto 0);
9    begin
10     while not endfile(f) loop
11       readline(f, l);
12       hex_read(l, x);
13       report to_string(x);
14     end loop;
15     file_close(f);
16     wait;
17   end process;
```

```
1 00
2 11
3 AA
```



Finally, we read the file line by line until the end of the file is reached, where reading happens by using the previously introduced subprograms. The respective calls first read a line from the file into a buffer variable and then convert it from hexadecimal and assign the result to the target variable.

Read from file


```
1  [...]
2  use std.textio.all;
3  [...]
4  begin
5    main: process is
6      file f : text open READ_MODE is "data.txt";
7      variable l : line;
8      variable x : std_ulogic_vector(7 downto 0);
9    begin
10     while not endfile(f) loop
11       readline(f, l);
12       hex_read(l, x);
13       report to_string(x);
14     end loop;
15     file_close(f);
16     wait;
17   end process;
```

```
1 00
2 11
3 AA
```

Lastly, the opened file is closed.

Read from file


```
1  [...]
2  use std.textio.all;
3  [...]
4  begin
5    main: process is
6      file f : text open READ_MODE is "data.txt";
7      variable l : line;
8      variable x : std_ulogic_vector(7 downto 0);
9    begin
10     while not endfile(f) loop
11       readline(f, l);
12       hex_read(l, x);
13       report to_string(x);
14     end loop;
15     file_close(f);
16     wait;
17   end process;
```

```
1 00
2 11
3 AA
```

Simulating this code results in the output shown on the right, where all three values contained in the file are reported.

Read from file


```
1  [...]
2  use std.textio.all;
3  [...]
4  begin
5    main: process is
6      file f : text open READ_MODE is "data.txt";
7      variable l : line;
8      variable x : std_ulogic_vector(7 downto 0);
9    begin
10     while not endfile(f) loop
11       readline(f, l);
12       hex_read(l, x);
13       report to_string(x);
14     end loop;
15     file_close(f);
16     wait;
17   end process;
```

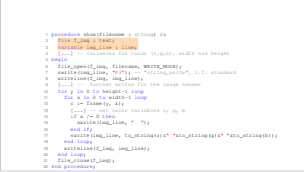
```
1 00
2 11
3 AA
```

```
[...]: 00000000
[...]: 00010001
[...]: 10101010
```

We will now conclude the part about file operations by considering an example for writing to a file. In particular, we are looking at the `show` procedure of `vhdldraw` that creates a text-based image file.

Example: VHDLDraw `show`


```
1  procedure show(filename : string) is
2    file f_img : text;
3    variable img_line : line;
4    [...] -- variables for color (r,g,b), width and height
5  begin
6    file_open(f_img, filename, WRITE_MODE);
7    swrite(img_line, "P3"); -- "string_write", c.f. standard
8    writeline(f_img, img_line);
9    [...] --  further writes for the image header
10   for y in 0 to height-1 loop
11     for x in 0 to width-1 loop
12       c := frame(y, x);
13       [...] -- set color variables r, g, b
14       if x /= 0 then
15         swrite(img_line, "  ");
16       end if;
17       swrite(img_line, to_string(r)&" "&to_string(g)&" "&to_string(b));
18     end loop;
19     writeline(f_img, img_line);
20   end loop;
21   file_close(f_img);
22 end procedure;
```



Just as in the previous example we start by declaring an object of the class `file` and type `text`, and a variable of type `line`. However, this time we are not opening the target file directly at the file declaration.

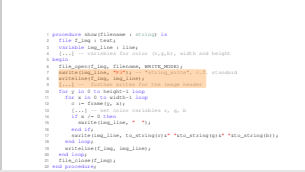
Example: VHDLDraw `show`


```
1  procedure show(filename : string) is
2    file f_img : text;
3    variable img_line : line;
4    [...] -- variables for color (r,g,b), width and height
5  begin
6    file_open(f_img, filename, WRITE_MODE);
7    swrite(img_line, "P3"); -- "string_write", c.f. standard
8    writeline(f_img, img_line);
9    [...] --  further writes for the image header
10   for y in 0 to height-1 loop
11     for x in 0 to width-1 loop
12       c := frame(y, x);
13       [...] -- set color variables r, g, b
14       if x /= 0 then
15         swrite(img_line, "  ");
16       end if;
17       swrite(img_line, to_string(r)&" "&to_string(g)&" "&to_string(b));
18     end loop;
19     writeline(f_img, img_line);
20   end loop;
21   file_close(f_img);
22 end procedure;
```

Instead, this is done in the procedure's statement body, using the `file_open` procedure. Note that we open the file in write mode.

Example: VHDLDraw `show`


```
1  procedure show(filename : string) is
2    file f_img : text;
3    variable img_line : line;
4    [...] -- variables for color (r,g,b), width and height
5  begin
6    file_open(f_img, filename, WRITE_MODE);
7    swrite(img_line, "P3"); -- "string_write", c.f. standard
8    writeline(f_img, img_line);
9    [...] --  further writes for the image header
10   for y in 0 to height-1 loop
11     for x in 0 to width-1 loop
12       c := frame(y, x);
13       [...] -- set color variables r, g, b
14       if x /= 0 then
15         swrite(img_line, "  ");
16       end if;
17       swrite(img_line, to_string(r)&" "&to_string(g)&" "&to_string(b));
18     end loop;
19     writeline(f_img, img_line);
20   end loop;
21   file_close(f_img);
22 end procedure;
```



The next thing the procedure does, is to write the special file header required by the particular image format we use. This is done first by writing to the line buffer and then appending it to the file.

Example: VHDLDraw show


```
1  procedure show(filename : string) is
2    file f_img : text;
3    variable img_line : line;
4    [...] -- variables for color (r,g,b), width and height
5  begin
6    file_open(f_img, filename, WRITE_MODE);
7    swrite(img_line, "P3"); -- "string_write", c.f. standard
8    writeline(f_img, img_line);
9    [...] --  further writes for the image header
10   for y in 0 to height-1 loop
11     for x in 0 to width-1 loop
12       c := frame(y, x);
13       [...] -- set color variables r, g, b
14       if x /= 0 then
15         swrite(img_line, "  ");
16       end if;
17       swrite(img_line, to_string(r)&" "&to_string(g)&" "&to_string(b));
18     end loop;
19     writeline(f_img, img_line);
20   end loop;
21   file_close(f_img);
22 end procedure;
```

Next, the red, green and blue color components of each pixel are written to the file.

Example: VHDLDraw show


```
1  procedure show(filename : string) is
2    file f_img : text;
3    variable img_line : line;
4    [...] -- variables for color (r,g,b), width and height
5  begin
6    file_open(f_img, filename, WRITE_MODE);
7    swrite(img_line, "P3"); -- "string_write", c.f. standard
8    writeline(f_img, img_line);
9    [...] --  further writes for the image header
10   for y in 0 to height-1 loop
11     for x in 0 to width-1 loop
12       c := frame(y, x);
13       [...] -- set color variables r, g, b
14       if x /= 0 then
15         swrite(img_line, "  ");
16       end if;
17       swrite(img_line, to_string(r)&" "&to_string(g)&" "&to_string(b));
18     end loop;
19     writeline(f_img, img_line);
20   end loop;
21   file_close(f_img);
22 end procedure;
```

After all pixels were written to the file, the procedure finishes by closing the file.

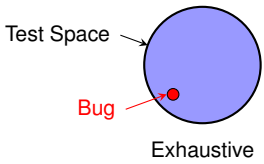
Example: VHDLDraw show


```
1  procedure show(filename : string) is
2    file f_img : text;
3    variable img_line : line;
4    [...] -- variables for color (r,g,b), width and height
5  begin
6    file_open(f_img, filename, WRITE_MODE);
7    swrite(img_line, "P3"); -- "string_write", c.f. standard
8    writeline(f_img, img_line);
9    [...] --  further writes for the image header
10   for y in 0 to height-1 loop
11     for x in 0 to width-1 loop
12       c := frame(y, x);
13       [...] -- set color variables r, g, b
14       if x /= 0 then
15         swrite(img_line, "  ");
16       end if;
17       swrite(img_line, to_string(r)&" "&to_string(g)&" "&to_string(b));
18     end loop;
19     writeline(f_img, img_line);
20   end loop;
21   file_close(f_img);
22 end procedure;
```

Exhaustive testing infeasible (exponential in state and input space)

Let us now talk about the support for random testing in VHDL. But first, we will address the question of why random testing is even required. In general, you are likely familiar with the problems involved in properly testing software. As these problems are inherent to verification though, they are also relevant for verifying hardware. In principle, we would of course like to exhaustively test our designs. That is, we want to ensure that for each possible input and system state the correct output is produced. This is illustrated by the image on the slide, which shows the overall test space, as a circle that contains all combinations of states and inputs that can occur during the proper operation of the design. For some of these combinations our design might behave erroneously, which is represented by the red dot. All tested combinations are shaded blue. Thus, we can clearly observe that exhaustive testing will find all bugs contained in a design. However, due to the testing effort growing exponentially in the state and input space, this quickly becomes infeasible even for simple systems.

- Exhaustive testing infeasible (exponential in state and input space)

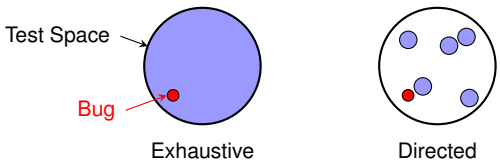


- Exhaustive testing infeasible (exponential in state and input space)
- Directed Testing
  - Apply predefined stimuli
  - Requires solid knowledge of UUT
  - Only finds “anticipated” bugs

Test Space
Bug
Exhaustive
Directed

An alternative is directed testing, where predefined stimuli are applied to the unit-under-test. This is only effective when detailed knowledge about the design and the specific scenarios to target is available, often resulting in the designers writing the tests themselves. However, this comes with an inherent limitation, as only bugs that are already anticipated can be found. In particular, designers are often aware of problematic parts of their designs and target these to find respective problems. Errors resulting from a designer misinterpreting the specification cannot be found using this method, as the designer will never create a test case that triggers this behavior. Furthermore, directed testing usually only covers a small part of the test space.

- Exhaustive testing infeasible (exponential in state and input space)
- Directed Testing
  - Apply predefined stimuli
  - Requires solid knowledge of UUT
  - Only finds “anticipated” bugs

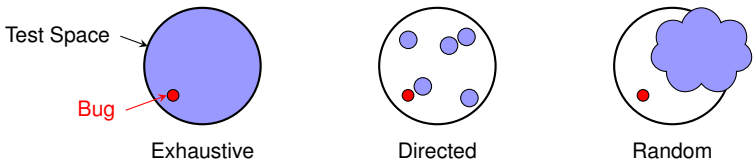


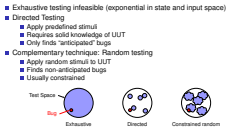
- Exhaustive testing infeasible (exponential in state and input space)
- Directed Testing
  - Apply predefined stimuli
  - Requires solid knowledge of UUT
  - Only finds “anticipated” bugs
- Complementary technique: Random testing
  - Apply random stimuli to UUT
  - Finds non-anticipated bugs

Test Space
Bug
Exhaustive
Directed
Random

This is where random testing, as a complementary technique to directed testing, comes in handy. This testing approach involves applying randomly generated stimuli to the UUT. A key advantage of this is that bugs that are not expected by the designer can be uncovered. However, just blindly applying random stimuli is usually not of much use, as the overall test space might not be covered very well, or inputs that cannot even occur during the operation of the design are applied.

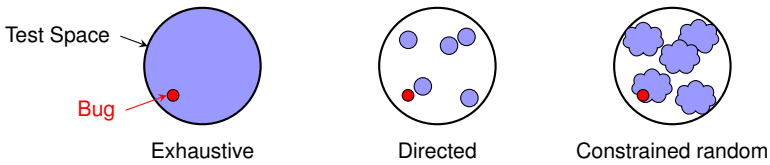
- Exhaustive testing infeasible (exponential in state and input space)
- Directed Testing
  - Apply predefined stimuli
  - Requires solid knowledge of UUT
  - Only finds “anticipated” bugs
- Complementary technique: Random testing
  - Apply random stimuli to UUT
  - Finds non-anticipated bugs





Therefore, we typically bound the randomly generated inputs by specific constraints, ensuring that only meaningful and relevant random data is applied. This can achieve a better coverage of the test space and a higher testing efficiency. Let us continue by discussing how VHDL facilitates random testing.

- Exhaustive testing infeasible (exponential in state and input space)
- Directed Testing
  - Apply predefined stimuli
  - Requires solid knowledge of UUT
  - Only finds “anticipated” bugs
- Complementary technique: Random testing
  - Apply random stimuli to UUT
  - Finds non-anticipated bugs
  - Usually constrained



uniform procedure of math_real package
Generates real in 0.0 < x < 1.0

In VHDL, random testing can be implemented using the `uniform` procedure provided in the `math_real` package. This procedure takes two seed values and generates a pseudo-random real number in the interval of 0 and 1, excluding the interval border values.

Random Testing in VHDL

- `uniform` procedure of `math_real` package

```
1 procedure uniform(variable seed1, seed2: inout positive;
2                   variable x : out real);
```

- Generates `real` in $0.0 < x < 1.0$

- `uniform` procedure of `math_real` package
- Generates `real` in $0.0 < x < 1.0$
- Seeds allow repetition of generated sequence

Internally, this procedure implements a particular pseudo-random-number-generator that mimics a uniform distribution of generated values. Using the two seeds, the generated sequence is controllable. This is not only paramount for reproducing test cases, but also allows easily changing the sequence of generated stimuli. Note that the procedure modifies the passed seed variables and that you should not manually change them after the first call to `uniform`.

Random Testing in VHDL

- `uniform` procedure of `math_real` package

```
1 procedure uniform(variable seed1, seed2: inout positive;
2                   variable x : out real);
```

- Generates `real` in $0.0 < x < 1.0$
- Seeds allow repetition of generated sequence

uniform procedure of math_real package
Generates real in $0.0 < x < 1.0$
Seeds allow repetition of generated sequence
Manual conversion to other types / ranges required

Note that `uniform` is the only built-in way to generate random values in VHDL. Therefore, we manually have to adapt the random floating point numbers for specific use cases, such as other types and ranges. We will now look at two examples.

Random Testing in VHDL

- `uniform` procedure of `math_real` package

```
1 procedure uniform(variable seed1, seed2: inout positive;
2                   variable x : out real);
```

- Generates `real` in $0.0 < x < 1.0$
- Seeds allow repetition of generated sequence
- Manual conversion to other types / ranges required

The first example we consider is an impure function that pseudo-randomly generates a `std_ulogic` value of zero or one. The subprogram signature, as well as its declarative section, are shown on the slide. Inside this section, we declare a variable for the result of the random number generation. In addition to that, we require two seed variables to be visible within the scope of the procedure and that are initialized to some arbitrary value. Note that we cannot simply declare the seed variables in the function's declarative section as well, as they must keep their values between calls of the function.

Random Testing in VHDL (cont'd)

- Generation of random `std_ulogic` value

```
1 impure function rand_sul return std_ulogic is
2   variable rand : real;
3 begin
```

Next, we use `uniform` to generate a pseudo-random real value, which will be contained in the variable `rand`.

Random Testing in VHDL (cont'd)

- Generation of random `std_ulogic` value

```
1 impure function rand_sul return std_ulogic is
2   variable rand : real;
3 begin
4   uniform(seed1, seed2, rand);
```

Based on this variable we can then either return a zero or a one. Next, let us look at how a random integer inside a certain interval can be generated based on `uniform`.

Random Testing in VHDL (cont'd)

- Generation of random `std_ulogic` value

```
1 impure function rand_sul return std_ulogic is
2   variable rand : real;
3 begin
4   uniform(seed1, seed2, rand);
5   if rand < 0.5 then
6     return '0';
7   end if;
8   return '1';
9 end function;
```

The respective subprogram takes two integer values, `start` and `stop`, which define the interval of the generated integer.

Random Testing in VHDL (cont'd)

- Generation of random `std_ulogic` value

```
1 impure function rand_sul return std_ulogic is
2   variable rand : real;
3 begin
4   uniform(seed1, seed2, rand);
5   if rand < 0.5 then
6     return '0';
7   end if;
8   return '1';
9 end function;
```

- Generation of `integer` range

```
1 impure function rand_int(start, stop : integer) return integer is
```

We start by generating a random `real` number just as in the previous example.

Random Testing in VHDL (cont'd)

- Generation of random `std_ulogic` value

```
1 impure function rand_sul return std_ulogic is
2   variable rand : real;
3 begin
4   uniform(seed1, seed2, rand);
5   if rand < 0.5 then
6     return '0';
7   end if;
8   return '1';
9 end function;
```

- Generation of `integer` range

```
1 impure function rand_int(start, stop : integer) return integer is
2   variable rand : real;
3 begin
4   uniform(seed1, seed2, rand);
```

We then scale this random number by the size of the interval plus one, offset it by the lower interval bound and subtract 0.5 before converting it to an integer.

Random Testing in VHDL (cont'd)

- Generation of random `std_ulogic` value

```
1 impure function rand_sul return std_ulogic is
2   variable rand : real;
3 begin
4   uniform(seed1, seed2, rand);
5   if rand < 0.5 then
6     return '0';
7   end if;
8   return '1';
9 end function;
```

- Generation of `integer` range

```
1 impure function rand_int(start, stop : integer) return integer is
2   variable rand : real;
3 begin
4   uniform(seed1, seed2, rand);
5   return integer(rand * real(stop-start+1)+real(start)-0.5);
6 end function;
```

The plus one and minus 0.5, highlighted in red, are necessary to give the border values the same probability as the other values inside the interval. This works because VHDL rounds to the nearest integer when converting a `real` to an `integer`.

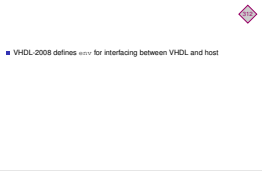
Random Testing in VHDL (cont'd)

- Generation of random `std_ulogic` value

```
1 impure function rand_sul return std_ulogic is
2   variable rand : real;
3 begin
4   uniform(seed1, seed2, rand);
5   if rand < 0.5 then
6     return '0';
7   end if;
8   return '1';
9 end function;
```

- Generation of `integer` range

```
1 impure function rand_int(start, stop : integer) return integer is
2   variable rand : real;
3 begin
4   uniform(seed1, seed2, rand);
5   return integer(rand * real(stop-start+1)+real(start)-0.5);
6 end function;
```



Finally, we briefly want to introduce the `std.env` package that is present since VHDL-2008. The purpose of this package is to provide an interface between VHDL and the simulation host.

Standard Environment Package

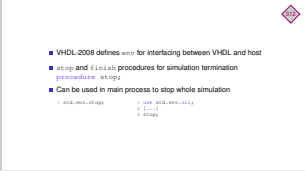
- VHDL-2008 defines `env` for interfacing between VHDL and host

In the original version, supported by the tools, the package essentially consists of the two procedures `stop` and `finish`. Both procedures can be used to terminate a simulation. According to the standard, the difference is that `finish` does not allow the simulation to be continued after the procedure call, whereas `stop` only pauses the simulation, thus allowing it to be resumed. However, we have not seen the tools we use react differently to the two calls and will hence only use `stop` in this course. In a nutshell, as long as some process eventually calls the subprogram, your simulation terminates regardless of whether all signals are stable and all processes contain a wait statement. The slide shows the declaration of `stop`.

Standard Environment Package

- VHDL-2008 defines `env` for interfacing between VHDL and host
- `stop` and `finish` procedures for simulation termination

```
procedure stop;
```



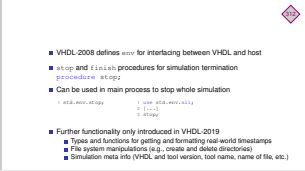
To use the subprogram of the `std.env` package, you can either directly call it using `std.env.stop`, shown in the left code snippet, or by adding a use statement for the `std.env` package and then calling the subprogram using only its identifier. This is shown in the right code snippet.

Standard Environment Package

- VHDL-2008 defines `env` for interfacing between VHDL and host
- `stop` and `finish` procedures for simulation termination
  `procedure stop;`
- Can be used in main process to stop whole simulation

```
1 std.env.stop;
```

```
1 use std.env.all;
2 [...]
3 stop;
```



Finally, we want to give an outlook into the functionality provided by the 2019 version of the `std.env` package, which will likely be supported by tools in the future. In this version of VHDL, the package was significantly extended in order to provide useful types and functions for logging, like a datetime type and functions that get real-world timestamps from the host. Furthermore, the newest version of the package allows testbenches to manipulate the host file system, such as creating or deleting directories. In addition to that, there are also subprograms defined that allow to get meta information about simulations, like the VHDL and tool version, the name of the current file and more. While these are particularly noteworthy changes, the package comes with further functionality we did not mention here. As always, you can find details in the standard.

Standard Environment Package

- VHDL-2008 defines `env` for interfacing between VHDL and host
- `stop` and `finish` procedures for simulation termination
  `procedure stop;`
- Can be used in main process to stop whole simulation

```
1 std.env.stop;
```

```
1 use std.env.all;
2 [...]
3 stop;
```

- Further functionality only introduced in VHDL-2019
  - Types and functions for getting and formatting real-world timestamps
  - File system manipulations (e.g., create and delete directories)
  - Simulation meta info (VHDL and tool version, tool name, name of file, etc.)

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

Lecture Complete!