TbGenerator

Documentation

Content

Table of Contents

Figures

No table of figures entries found.

1 Introduction

1.1 Aim of the Tool

The tool *TbGenerator* automatically generates test-bench skeletons for a VHDL entity based on parsing the entity itself. This does not only save a lot of time that is usually lost on just instantiating the DUT, defining all signals, creating all clocks, etc. but it also leads to a consistent test-bench style. Having such a tool also raises the motivation to do proper testing since the effort (especially for boring work) is drastically reduced.

Whatever output the tool produces can be edited manually easily. The aim of the tool is not to cover each and every special case but to generate a skeleton that can be modified manually.

1.2 Test-bench Styles

The tool supports two different test-bench styles.

1.2.1 Simple Test-benches

By default, *TbGenerator* creates a simple test-bench that consists of one single file. In that file, the DUT is instantiated and all clocks and resets are generated. The actual test-cases (i.e. stimuli generation and response checking) happen in one or more processes within that one test-bench file.

An example for such a test-bench can be found here:

<root>/example/simpleTb → execute *run.bat* to generate the test-bench

1.2.2 Multi-Case Testbenches

For more complex test-benches with multiple test-cases, it is useful to split code into multiple files. Whenever test-bench generator generates a test-bench that contains multiple test-cases (see 4.2.1.2), the test-bench contains multiple files.

The main test-bench file (*xxx_tb.vhd*) is responsible for instantiating the DUT, generating clocks and resets as well as top-level control of the test-bench. The top-level control part is responsible for executing all test-cases, one after the other (only starting the next one if all processes of the last one completed).

The test-bench package file (*xxx_tb_pkg.vhd*) contains some type and constant definitions that are required in both, the main test-bench file and the test-case packages.

The test-case package files (*xxx_tb_case_yyy.vhd*) contain procedures to do stimuli generation and response checking for one specific test-case. This is where the user-code for the test-cases must be added.

An example for such a test-bench can be found here:

<root>/example/multiCaseTb → execute *run.bat* to generate the test-bench

2 Command Line Interface

2.1 Introduction

The tool is written in python, so it is executed as python script. Note that below the Windows syntax is given (“py”). For Linux, replace “py” with “python3”.

The command syntax is given below:

```
py TbGen.py [-h] –src SRC –dst DST [-clear] [-mrg] [-force]
```

2.2 Arguments

2.2.1 Help (-h)

This argument prints out the help text for the CLI that describes all arguments.

2.2.2 Source File (-src)

This argument is required and is the path to the file containing the VHDL entity to be parsed.

2.2.3 Destination Folder (-dst)

This argument is required and contains the path to the folder to place the test-bench in. If the folder does not exist already, it is created.

2.2.4 Clear Destination Folder (-clear)

If this argument is passed, the destination folder is cleared prior to generating the test-bench. “Cleared” means that all files in the folder are deleted (all files, not only .vhd files!). Folders are not deleted since they may contain scripts or stimuli data.

If the destination folder already contains a test-bench, this argument must be passed. Otherwise errors are thrown since *TbGenerator* by default does not overwrite any existing fails to prevent loss of work because of accidental test-bench generator

By default *TbGenerator* asks whether it should really clear the directory if files are present. If this behavior is not required, the *–force* option must be used.

2.2.5 Create Merge Files (-mrg)

If this flag is passed, all generated files have the ending .mrg instead of .vhd. This allows to easily merge changes into an existing (and manually edited) test-bench using a 3rd party merging tool (e.g. WinMerge).

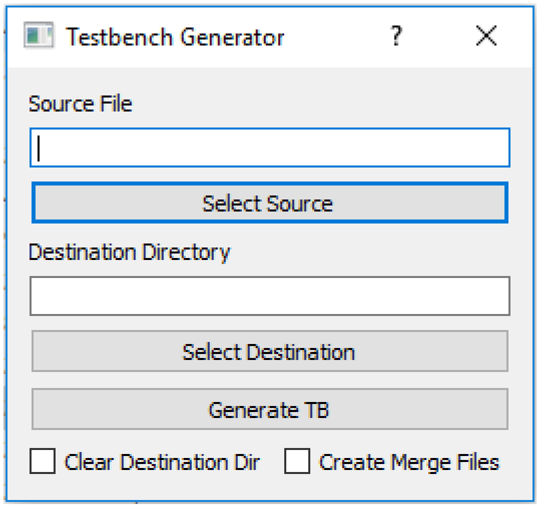
Note that .mrg files are overwritten if they already exist since it is assumed that they are never edited manually and are used temporary for merging into the .vhd files only.

2.2.6 Force Clearing of the Destination Folder (-force)

This flag prevents *–clear* from asking whether existing files should really be deleted.

3 Graphical User Interface

For running the GUI, the file *TbGenGui.pyw* must be executed. The operating system should automatically launch it in python correctly.



The GUI contains the same options as the command line interface, so please refer to 2 for details.

4 VHDL Tags

Information that is not visible from the VHDL alone needs to be passed to *TbGenerator* using tags that are placed as comments within the VHDL file.

4.1 Syntax

Tags can be placed anywhere in VHDL comments (also before or after other text). Tags always stand between "$$" signs. Each tag has a name and is assigned a value in the form "<name>=<value>".

Single VHDL tags can look like this:

```
-- Some comment text     $$ dutlib=psi_lib $$
SomePort : in std_logic; -- $$ type=rst $$
```

Multiple tags can be separted using ";":

```
SomePort : in std_logic; -- $$ type=rst; clk=ClkIn $$
```

Some tags not only take single values but also accept lists. Values of lists are separated using ",":

```
SomePort : in std_logic; -- $$ proc=stimuli,response $$
```

4.2 Tag Descriptions

4.2.1 File Scope Tags

These tags are not related to specific VHDL elements and can be placed as pure comment-lines anywhere before the entity declaration.

4.2.1.1 PROCESSES

Purpose

This tag defines how many independent processes are required for the test-bench and what their names are.

Optional

Yes, by default only one process named “stimuli” is generated.

Example

```
-- $$ processes=config,input,output $$
```

4.2.1.2 TESTCASES

Purpose

This tag defines how many independent test-cases are required for the test-bench and what their names are. This tag also decides whether a single test-case test-bench is generated (if the tag is not given) or a multi-case test-bench is generated (if the tag is given).

Optional

Yes, by default a single file test-bench is generated.

Example

```
-- $$ testcases=normalOp,overflow,underflow $$
```

4.2.1.3 DUTLIB

Purpose

This tag can be used to explicitly specify to which VHDL library the DUT belongs. This is required whenever the test-bench is not compiled into the same library as the DUT.

Optional

Yes, by default it is assumed that the test-bench gets compiled into the same library as the DUT.

Example

```
-- $$ dutlib=psi_lib $$
```

4.2.1.4 TBPKG

Purpose

This tag can be used to specify additional packages that are not required for the implementation of the entity but will be used in the test-bench. These packages will then be added to all generated files.

Packages must be specified in the form *<library>.<package>*.

Optional

Yes.

Example

```
-- $$ tbpkg=work.my_package_pkg,any_lib.blubb_pkg $$
```

4.2.2 Generic Tags

Generic tags are related to specific generics and must be placed as comments on the same line as the generic declaration itself.

4.2.2.1 EXPORT

Purpose

This tag can be used to let *TbGenerator* know that a generic should be exported from the testbench (i.e. be configurable from outside of the testbench).

Optional

Yes, by default generics are not exported.

Example

```
SomeGeneric : integer := 5; -- $$ export=true $$
```

4.2.2.2 CONSTANT

Purpose

This tag can be used to let *TbGenerator* know that a generic should be set to a given value (that can be different from the default value) in the testbench.

Optional

Yes, by default generics are set to their default values.

Example

```
SomeGeneric : integer := 5; -- $$ constant=10 $$
```

4.2.3 Port Tags

Port tags are related to specific ports and must be placed as comments on the same line as the port declaration itself.

4.2.3.1 LOWACTIVE

Purpose

This tag allows specifying that a port is low-active. This leads *TbGenerator* to initialize the related signal with high-level instead of low-level for normal signals. For resets it leads *TbGenerator* to apply ‘0’ to assert the reset and ‘1’ to remove it.

Optional

Yes, by default all ports are regarded as high-active.

Example

```
SomePort : in std_logic; -- $$ lowactive=true $$
```

4.2.3.2 TYPE

Purpose

This tag allows specifying that a port has a special functionality. The following special functionalities are available:

clk The port is a clock input. *TbGenerator* automatically generates the correct clock on this pin. Clock pins must also have the *FREQ* tag to specify the clock frequency.

rst The port is a reset input. *TbGenerator* automatically asserts the reset at the beginning of the test and removes it synchronously to the clock. Reset ports must also have the *CLK* tag to specify the related clock.

Optional

Yes, by default all ports are regarded as signals

Example

```
InClk : in std_logic; -- $$ type=clk; freq=100e6 $$
InRst : in std_logic; -- $$ type=rst; clk=InClk $$
```

4.2.3.3 FREQ

Purpose

This tag allows specifying the frequency of clock inputs. It always appears together with *type=clk*.

Optional

Yes, but mandatory for ports with *type=clk*

Example

```
InClk : in std_logic; -- $$ type=clk; freq=100e6 $$
```

4.2.3.4 CLK

Purpose

This tag allows specifying the clock a reset is related to. It always appears together with *type=rst* and its value must always be the name of a a port with *type=clk*.

Optional

Yes, but mandatory for ports with *type=rst*

Example

```
InRst : in std_logic; -- $$ type=rst; clk=InClk $$
```

4.2.3.5 PROC

Purpose

This tag is only important for multi-case test-benches. In such test-benches, procedures are called for executing the functionality of one process for one specific test-case. Therefore it must be known which signals are driven by which processes. This information can be given with the *PROC* tag.

The value must be one of the process names defined using *PROCESSES*. The value can also be a list of process names. For output signals the signal is an input to all processes in the list case, for input signals, the signal is available to all processes but only driven by the first one in the list.

Optional

Yes, only required for multi-case test-benches.

Example

```
SomePort : in std_logic; -- $$ proc=stimuli,response $$
```

5 Tips & Tricks

5.1 Inter-Procedure Communication in Packages

For multi-case test-benches, all functionality is wrapped in procedures. If communication between two procedures within a test-case package is required (e.g. because the response checking procedure needs to wait until the configuration procedure as completed), *shared variable* are the way to go. *Shared variables* can be defined in the scope of a package and accessed by all procedures within that package.