ChipTools Documentation

Release 0.0.1

Peter Bennett

Mar 22, 2017

Contents

CHAPTER 1

Introduction

What is ChipTools?

ChipTools is a build and verification framework for FPGA designs.

What can it do?

ChipTools aims to simplify the process of building and testing FPGA designs by providing a consistent interface to vendor applications and automating simulation and synthesis flows.

Key features

- Seamlessly switch between vendor applications without modifying build scripts or project files.
- Enhance testbenches with Python based stimulus generation and checking.
- Automate test execution and reporting using the Python Unittest framework.
- Automatically check and archive build outputs.
- Preprocess and update files before synthesis to automate tasks such as updating version registers.
- Free and open source under the Apache 2.0 License.

Supported Tools

The following tools are currently supported, support for additional tools will be added in the future.

Simulation Tools

- Modelsim (tested with 10.3)
- ISIM (tested with 14.7)
- GHDL (tested with 0.31)
- Vivado (tested with 2015.4)
- Icarus (tested with 0.9.7)

Synthesis Tools

- Xilinx ISE (tested with 14.7)
- Quartus (tested with 13.1)
- Vivado (tested with 2015.4)

CHAPTER 2

Getting Started

Installing ChipTools

Dependencies

ChipTools has the following requirements:

- Python 3.4
- Colorama *(Optional)* to support coloured terminal text on Windows platforms.

Installation

ChipTools should work on any platform that can run Python 3 and your preferred FPGA simulation and synthesis tools.

Clone the ChipTools repository to your system (or download it here):

```
$ git clone --recursive https://github.com/pabennett/chiptools.git
```

Install using the setup.py script provided in the root directory:

```
$ cd chiptools
$ python setup.py install
```

After installation, the ChipTools command line interface can be started with:

```
$ chiptools
```

Configuring ChipTools

.chiptoolsconfig

ChipTools will automatically detect supported simulation and synthesis tools installed on your system by searching the PATH environment variable. If you prefer to explicitly point ChipTools to a specific program you can edit the *.chiptoolsconfig* file which is automatically created by ChipTools in your HOME directory.

The .chiptoolsconfig file uses *INI* format and contains the following:

- **[simulation executables]** Paths to simulation tools
- **[synthesis executables]** Paths to synthesis tools
- **[<toolname> simulation libraries]** Paths to precompiled libraries for the given *<toolname>*

An example .chiptoolsconfig is given below:

```
[simulation executables]
modelsim        = C:\modelsim_dlx_10.3d\win32pe

[synthesis executables]
ise             = C:\Xilinx\14.7\ISE_DS\ISE\bin\nt\
quartus         = C:\altera\13.1\quartus\bin\

[modelsim simulation libraries]
unisim          = C:\modelsim_dlx_10.3d\unisim
xilinxcorelib   = C:\Xilinx\modelsim_10_3de_simlibs\xilinxcorelib
unimacro        = C:\Xilinx\modelsim_10_3de_simlibs\unimacro
secureip        = C:\Xilinx\modelsim_10_3de_simlibs\secureip
```

Tool names under the simulation or synthesis executables categories will only be used if a tool wrapper plugin is available. A list of available plugins can be obtained by launching ChipTools and issuing the **plugins** command.

Loading a design

Project data can be loaded into ChipTools in two ways: using a project file or by importing ChipTools in a Python script and using the **Project** class directly:

```
from chiptools.core.project import Project

# Configure project
project.add_config('simulation_directory': 'path/to/simulation_directory')
project.add_config('synthesis_directory': 'path/to/synthesis_directory')
project.add_config('simulator': 'modelsim')
project.add_config('synthesiser': 'quartus')
project.add_config('part': 'EP3C40F484C6')
# Add constraints
project.add_constraints('path/to/synthesis_constraints.sdc')
# Add source files
project.add_file('path/to/my_top.vhd', library='top')
# Synthesise the project (library and entity)
project.synthesise('top', 'my_top')
```

Project File

ChipTools supports a simple XML file format that can be used to define source files and configuration for your project:

```
<!-- Paths in a project file are relative to the project file location -->
<project>
    <!-- Project Config -->
    <config synthesis_directory='path/to/simulation_directory'/>
    <config simulation_directory='path/to/synthesis_directory'/>
    <config simulator='modelsim'/>
    <config synthesiser='ise'/>
    <config part='xc6slx100t-3-fgg676'/>
    <constraints path='path/to/synthesis_constraints.ucf'/>
    <library name=top>
        <file path='path/to/my_top.vhd'/>
    </library>
</project>
```

The XML file can be loaded into the ChipTools command line interface and operated on interactively.

```
$ chiptools
(cmd) load_project my_project.xml
(cmd) synthesise top.my_top
```

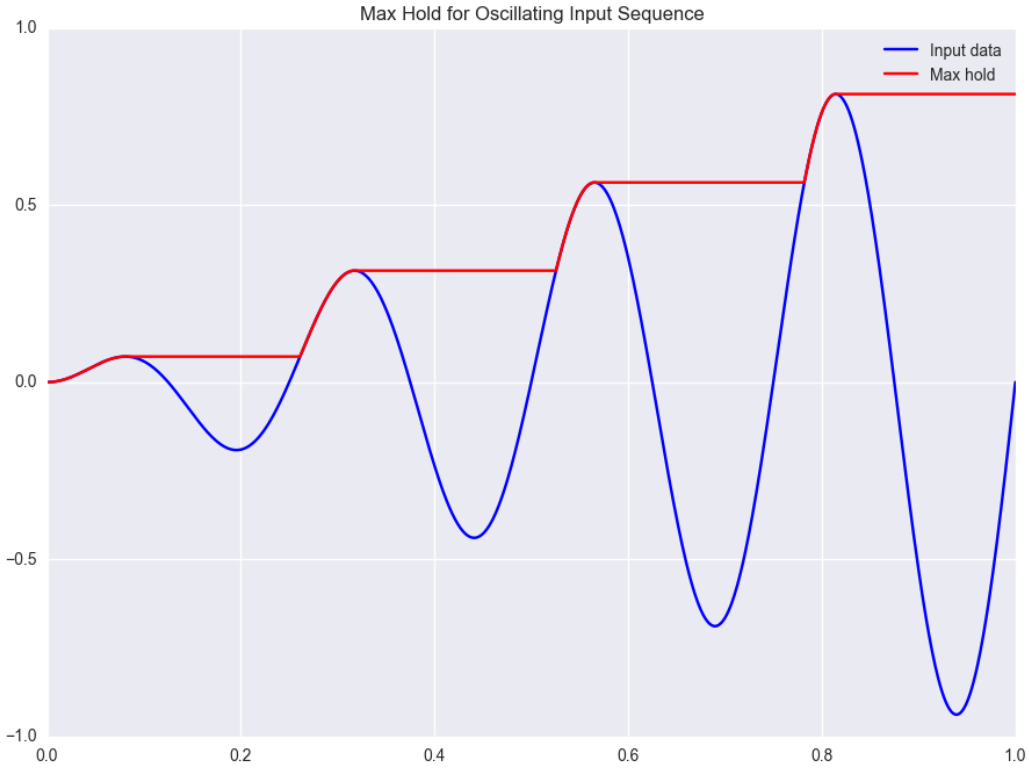
CHAPTER 3

Examples: Max Hold

A demonstration of the ChipTools framework being used to simulate, test and build a simple VHDL component.

Introduction

The **Max Hold** example implements a basic component to output the maximum value of an input sequence until it is reset. For example, if such a component were to be fed an oscillating input with steadily increasing amplitude we would expect the following result:



This example will show you how you can use ChipTools to generate stimulus, check responses, create test reports and generate bit files for the **Max Hold** component.

Source Files

The Max Hold example is located in examples/max_hold.

The following source files belong to the Max Hold example:

Name	Type	Description
max_hold.vhd	**VHDL**	Max Hold component (VHDL).
pkg_max_hold.vhd	**VHDL**	Package for Max Hold component.
tb_max_hold.vhd	**VHDL**	Testbench for Max Hold component (VHDL).
max_hold.sv	**SystemVerilog**	Max Hold component (SystemVerilog).
tb_max_hold.sv	**SystemVerilog**	Testbench for Max Hold component (SystemVerilog).

Support files:

Name	Type	Description
max_hold_tests.py	**Unit Test**	Collection of advanced unit tests.
basic_unit_tests.py	**Unit Test**	Simple unit test.
max_hold.xml	**Project**	Project file (using VHDL sources).
max_hold_sv.xml	**Project**	Project file (using SystemVerilog sources).
max_hold.ucf	**Constraints**	Constraints file when using the ISE synthesis flow.
max_hold.xdc	**Constraints**	Constraints file when using the Vivado synthesis flow.
simulation	**Folder**	Output directory for simulation tasks.
synthesis	**Folder**	Output directory for synthesis tasks.

The Max Hold component has been designed in both VHDL and SystemVerilog so that single-language simulators such as GHDL or Icarus can be used.

Creating the Project

This section will walk through the steps required to load and use the source files with ChipTools. The complete example is available in **max_hold_project.py** in the Max Hold project directory.

Initial Setup

First, import the ChipTools Project wrapper and create a project instance:

```
from chiptools.core.project import Project

# Create a new Project
project = Project()
```

The project wrapper provides a set of functions for loading source files and configuring a project.

Project Configuration

Projects should be configured with the following information before they are used:

Configuration Item	Description
simulation_directory	Output directory for simulation tasks.
synthesis_directory	Output directory for synthesis tasks.
fpga_part	The FPGA part to target when performing a synthesis flow.
simulator	Name of the default simulator to use when performing simulations.
synthesiser	Name of the default synthesiser to use when performing synthesis.

The project wrapper provides two methods for setting configuration data: **add_config**, which accepts a name, value pair as arguments or **add_config_dict**, which accepts a dictionary of name, value pairs.

The following code sample uses the **add_config** method to configure the project wrapper.

```
# Configure project, you may wish to edit some of these settings depending
# on which simulation/synthesis tools are installed on your system.
project.add_config('simulation_directory', 'simulation')
project.add_config('synthesis_directory', 'synthesis')
project.add_config('simulator', 'ghdl')
project.add_config('synthesiser', 'ise')
project.add_config('part', 'xc6slx9-csg324-2')
```

Apply Values to Generic Ports

FPGA designs can be parameterised via the use of a generic port on the top level component. You can assign values to top level port generics by using the **add_generic** method:

```
# Synthesis generics can be assigned via the add_generic command, in this
# example we set the data_width generic to 3:
project.add_generic('data_width', 3)
```

Add Source Files

Add the Max Hold source files to the project and assign them to a library:

```
# Source files for the max_hold component are added to the project. The Project
# **add_file** method accepts a file path and library name, if no library is
# specified it will default to 'work'. Other file attributes are available but
# not covered in this example.
project.add_file('max_hold.vhd', library='lib_max_hold')
project.add_file('pkg_max_hold.vhd', library='lib_max_hold')
```

The testbench is also added to the project under a different library. The optional argument *synthesise* is set to **False** when adding the testbench as we do not want to include it in the files sent to synthesis:

```
# When adding the testbench file we supply a 'synthesise' attribute and set it
# to 'False', this tells the synthesis tool not to try to synthesise this file.
# If not specified, 'synthesise' will default to 'True'
project.add_file(
    'tb_max_hold.vhd',
    library='lib_tb_max_hold',
    synthesise=False
)
```

There are two unit test files provided for the Max Hold project, these can be added to the project using the **add_unittest** method:

```
# Unit tests have been written for the max_hold component and stored in
# max_hold_tests.py. The Project class provides an 'add_unittest' method for
# adding unit tests to the project, it expects a path to the unit test file.
project.add_unittest('max_hold_tests.py')
project.add_unittest('basic_unit_test.py')
```

Finally, the constraints files can be added to the project using the **add_constraints** method, which takes a **path** argument and an optional **flow** name argument which allows you to explicitly name which synthesis flow the constraints are intended for:

```
# The constraints are added to the project using the add_constraints method.
# The optional 'flow' argument is used to explicitly identify which synthesis
# flow the constraints are intended for (the default is to infer supported
# flows from the file extension).
project.add_constraints('max_hold.xdc', flow='vivado')
project.add_constraints('max_hold.ucf', flow='ise')
```

The project is now fully configured and can be synthesised, simulated or the unit test suite can be executed to check that the requirements are met:

```
# Simulate the project interactively by presenting the simulator GUI:
project.simulate(
    library='lib_tb_max_hold',
    entity='tb_max_hold',
    gui=True,
    tool_name='modelsim'
)
# Run the automated unit tests on the project (console simulation):
project.run_tests(tool_name='isim')
# Synthesise the project:
project.synthesise(
    library='lib_max_hold',
    entity='max_hold',
    tool_name='vivado'
)
```

Alternatively the ChipTools command line can be launched on the project to enable the user to run project operations interactively:

```
# Launch the ChipTools command line with the project we just configured:
from chiptools.core.cli import CommandLine
CommandLine(project).cmdloop()
```

Project (XML) File

The Project configuration can also be captured as an XML file, which provides an alternative method of maintaining the project configuration.

The example project file **max_hold.xml** provides the same configuration as **max_hold_project.py**:

```
<project>
    <config simulation_directory='simulation'/>
    <config synthesis_directory='synthesis'/>
    <config simulator='ghdl'/>
    <config synthesiser='vivado'/>
    <config part='xc7a100tcsg324-1'/>
    <unittest path='max_hold_tests.py'/>
    <unittest path='basic_unit_test.py'/>
    <constraints path='max_hold.ucf' flow='ise'/>
    <constraints path='max_hold.xdc' flow='vivado'/>
    <generic data_width='3'/>
    <library name='lib_max_hold'>
        <file path='max_hold.vhd'/>
        <file path='pkg_max_hold.vhd'/>
    </library>
    <library name='lib_tb_max_hold'>
        <file
            path='tb_max_hold.vhd'
            synthesise='false'
        />
    </library>
</project>
```

The project XML file can be loaded in the ChipTools command line interface using the **load_project** command:

```
$ chiptools
(cmd) load_project max_hold.xml
```

...or in a Python script:

```
from chiptools.core.project import Project

# Create a new Project
project = Project()
# Load a pre-existing project file
project.load_project('max_hold.xml')
```

Simulation and Test

To test the Max Hold component an accompanying testbench, *tb_max_hold.vhd* (VHDL) or *tb_max_hold.sv* (SystemVerilog), is used to feed the component data from a stimulus input text file and record the output values in an output text file. By using stimulus input files and output files we gain the freedom to use the language of our choice to generate stimulus and check results.

A simple stimulus file format is used by the testbench that allows a data write or a reset to be issued to the unit under test:

Stimulus File Format	
Reset (1-bit) (Binary)	Data (N-bit) (Binary)
Reset (1-bit) (Binary)	Data (N-bit) (Binary)
... Repeated	

The width of the binary data field must match the data width on the testbench generic. On each clock cycle a single line should be read from the stimulus file and the supplied values sent to the input of the Max Hold component.

We will use Python to create stimulus files in this format for the testbench.

Unit Tests

Note: The following example can be found in **examples/max_hold/basic_unit_test.py**

We can use Python to define tests for the Max Hold component by first importing the **ChipToolsTest** class from **chiptools.testing.testloader**

```
from chiptools.testing.testloader import ChipToolsTest
```

The **ChipToolsTest** class provides a wrapper around Python's Unittest **TestCase** class that will manage simulation execution behind the scenes while our test cases are executed.

First off, create a ChipToolsTest class and define some basic information about the testbench:

```
class MaxHoldsTestBase(ChipToolsTest):
    # Specify the duration your test should run for in seconds.
    # If the test should run until the testbench aborts itself use 0.
    duration = 0
    # Testbench generics are defined in this dictionary.
    # In this example we set the 'width' generic to 32, it can be overridden
    # by your tests to check different configurations.
```

```
    generics = {'data_width': 32}
    # Specify the entity that this Test should target
    entity = 'tb_max_hold'
    # Specify the library that this Test should target
    library = 'lib_tb_max_hold'
```

These attributes provide the basic information required by ChipTools to execute the testbench.

Tests are executed using the following sequence when using the Python Unittest framework:

1. Execute the unit test class **setUp** function if defined.
2. Execute the test case (a test case is any class method with a 'test prefix').
3. Execute the unit test class **tearDown** function if defined.

If the unit test class provides multiple testcases they can be executed individually or as a batch in ChipTools. The sequence above is executed for each individual test case.

The **setUp** function executes before each test and can be used to prepare any inputs that do not change for each test. In this example we will simply use the setUp function to prepare the test environment by defining paths to the input and output files to be used by the testbench:

```
def setUp(self):
    """Place any code that is required to prepare simulator inputs in this
    method."""
    # Set the paths for the input and output files using the
    # 'simulation_root' attribute as the working directory
    self.input_path = os.path.join(self.simulation_root, 'input.txt')
    self.output_path = os.path.join(self.simulation_root, 'output.txt')
```

Similarly, the **tearDown** function executes at the end of each test, so we can use this to remove any files that were generated during the test:

```
def tearDown(self):
    """Insert any cleanup code to remove generated files in this method."""
    os.remove(self.input_path)
    os.remove(self.output_path)
```

To execute our tests we will create a function that performs the following operations:

1. Create an array of N random integers
2. Write the array of integers to the input stimulus file
3. Execute the simulation and check that the return code is 0
4. Read the output data generated by the simulation
5. Compare the output data to our Python model of the Max Hold function.

```
def run_random_data_test(self, n):

    # Generate a list of n random integers
    self.values = [random.randint(0, 2**32-1) for i in range(n)]

    # Write the values to the testbench input file
    with open(self.input_path, 'w') as f:
        for value in self.values:
            f.write(
                '{0} {1}\n'.format(
```

```
                    '0',  # Reset status (0)
                    bin(value)[2:].zfill(32),  # write 32bit data
                )
            )

    # Run the simulation
    return_code, stdout, stderr = self.simulate()
    self.assertEqual(return_code, 0)

    # Read the simulation output
    output_values = []
    with open(self.output_path, 'r') as f:
        data = f.readlines()
    for valueIdx, value in enumerate(data):
        # testbench response
        output_values.append(int(value, 2))  # Binary to integer

    # Use Python to work out the expected result from the original imput
    max_hold = [
        max(self.values[:i+1]) for i in range(len(self.values))
    ]

    # Compare the expected result to what the Testbench returned:
    self.assertListEqual(output_values, max_hold)
```

Now we can add extra functions to our class using the 'test' prefix to execute **run_random_data_test** with different parameters:

```
def test_10_random_integers(self):
    """Check the Max hold component using 10 random integers."""
    self.run_random_data_test(10)

def test_100_random_integers(self):
    """Check the Max hold component using 100 random integers."""
    self.run_random_data_test(100)
```

The above example is saved as **basic_unit_test.py** in the Max Hold example folder. We can run this test by invoking ChipTools in the example folder, loading the **max_hold_basic_test.xml** project and then adding and running the testsuite (simulator output has been hidden for presentation purposes):

```
$ chiptools
(Cmd) load_project max_hold_basic_test.xml
(Cmd) run_tests
ok test_100_random_integers (chiptools_tests_basic_unit_test.MaxHoldsTestBase)
ok test_10_random_integers (chiptools_tests_basic_unit_test.MaxHoldsTestBase)
Time Elapsed: 0:00:11.967197
(Cmd)
```

Unit Test Report

When ChipTools has finished running a test suite invoked with the **run_tests** command it will place a report called **report.html** in the simulation directory. The unit test report indicates which tests passed or failed and provides debug information on tests that have failed. A sample report for the full Max Hold unit test suite is given below:

Unit Test Report

Start Time: 2016-01-22 10:08:28
Duration: 0:02:14.934493
Status: Pass 17

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	Error	View
chiptools_tests_basic_unit_test.MaxHoldsTestBase	2	2	0	0	Detail
test_100_random_integers: Check the Max hold component using 100 random integers.			pass		
test_10_random_integers: Check the Max hold component using 10 random integers.			pass		
chiptools_tests_max_hold_tests.MaxHoldTests	15	15	0	0	Detail
test_max_hold_constant_data_0: Test the max hold function with constant 0 data.			pass		
test_max_hold_constant_data_1: Test the max hold function with constant 1 data.			pass		
test_max_hold_constant_data_100: Test the max hold function with constant 100 data.			pass		
test_max_hold_impulse: Test the max hold function with impulses.			pass		
test_max_hold_ramp_down: Test the max hold function with negative gradient ramps.			pass		
test_max_hold_ramp_up: Test the max hold function with positive gradient ramps.			pass		
test_max_hold_random_100bit: Test the max hold function with 100bit random data.			pass		
test_max_hold_random_128bit: Test the max hold function with 128bit random data.			pass		
test_max_hold_random_1bit: Test the max hold function with 1bit random data.			pass		
test_max_hold_random_32bit: Test the max hold function with 32bit random data.			pass		
test_max_hold_random_8bit: Test the max hold function with 8bit random data.			pass		
test_max_hold_random_single_sequence: Test the max hold function with a single random sequence.			pass		
test_max_hold_sinusoid: Test the max hold function with sinusoids.			pass		
test_max_hold_sinusoid_single_sequence: Test the max hold function with a single sinusoid sequence.			pass		
test_max_hold_square: Test the max hold function with square waves.			pass		
Total	**17**	**17**	**0**	**0**	

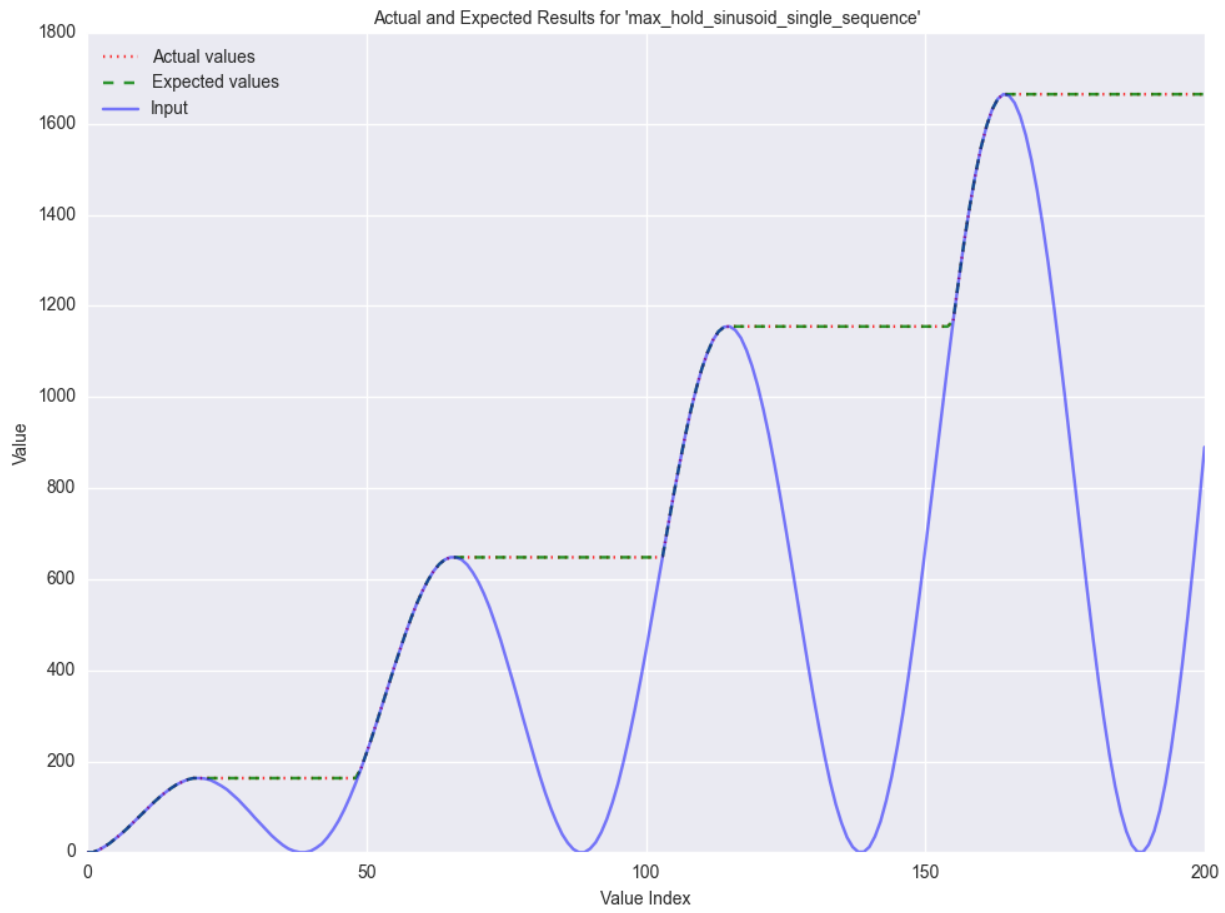
Note: The test report is overwritten each time the unit test suite is executed, so backup old reports if you want to keep them.

Advanced Unit Tests

The previous example showed how a simple unit test can be created to test the Max Hold component with random stimulus. This approach can be extended to produce a large set of tests to thoroughly test the component and provide detailed information about how it is performing. The **max_hold_tests.py** file in the Max Hold example folder implements the following tests:

Test Name	Data Width	Description
max_hold_constant_data_0	32	Continuous data test using zero
max_hold_constant_data_1	32	Continuous data test using 1
max_hold_constant_data_100	32	Continuous data test using 100
max_hold_impulse_test	32	The first data point is nonzero followed by constant zero data.
max_hold_ramp_down_test	32	Successive random length sequences of reducing values.
max_hold_ramp_up_test	32	Successive random length sequences of increasing values.
max_hold_random_single_sequence	32	Single sequence of 200 random values.
max_hold_random_tests_100bit	100	Successive random length sequences of 100bit random values.
max_hold_random_tests_128bit	128	Successive random length sequences of 128bit random values.
max_hold_random_tests_1bit	1	Successive random length sequences of 1bit random values.
max_hold_random_tests_32bit	32	Successive random length sequences of 32bit random values.
max_hold_random_tests_8bit	8	Successive random length sequences of 8bit random values.
max_hold_sinusoid_single_sequence	12	Single sinusoidal sequence.
max_hold_sinusoid_test	12	Multiple sinusoidal sequences of random length.
max_hold_square_test	8	Multiple toggling sequences of random length.

If **Matplotlib** is installed the Unit Test will also create an output image for each test in the simulation folder to show a graph of the input data with the model data and the Max Hold component output data. For example, the max_hold_sinusoid_single_sequence test produces the following output:



Note: For this example, graph generation requires Matplotlib (optionally with Seaborn)

Plots such as these provide a powerful diagnostic tool when debugging components or analysing performance.

Synthesis and Build

Warning: The **Max Hold** example is provided to demonstrate the ChipTools build process, do not attempt to use the bitfiles generated from this project on an FPGA as the IO constraints are not fully defined and have not been checked. Using the bitfiles generated from this project may cause damage to your device.

The Max Hold example includes the files necessary for it to be built using the Xilinx ISE, Vivado and Quartus synthesis flows; the project files provided in the example are configured to use the Vivado synthesis flow by default.

Building with the Command Line Interface

To build the design using the ChipTools command line, first open a terminal in the Max Hold example directory and invoke the ChipTools command line:

```
$ chiptools
--------------------------------------------------------------------------------
ChipTools (version: 0.1.50)

Type 'help' to get started.
Type 'load_project <path>' to load a project.
The current directory contains the following projects:
        1: max_hold.xml
        2: max_hold_basic_test.xml
--------------------------------------------------------------------------------
(cmd)
```

Two projects should be listed by ChipTools in the current directory, load the **max_hold.xml** project by using the **load_project** command:

```
(Cmd) load_project max_hold.xml
[INFO] Loading max_hold.xml in current working directory: max_hold
[INFO] Loading project: max_hold.xml
[INFO] Parsing: max_hold.xml synthesis=None
(Cmd)
```

We can check which files will be sent to the synthesis tool by using the **show_synthesis_fileset** command:

```
(Cmd) show_synthesis_fileset
[INFO] Library: lib_max_hold
[INFO]          max_hold.vhd
[INFO]          pkg_max_hold.vhd
[INFO] Library: lib_tb_max_hold
```

Note that the Max Hold testbench **tb_max_hold.vhd** is excluded from synthesis, this is due to the *synthesis='false'* attribute on the testbench file tag in the **max_hold.xml** project file.

An FPGA build can be initiated by using the **synthesise** command, which accepts the following arguments:

Argument	Description
target	The library and entity to synthesise, using the format *library.entity*
flow	The synthesis flow to use. The default value is taken from the project config.
part	The fpga part to use. The default value is taken from the project config.

To build the Max Hold project using the default synthesis flow (Vivado) for the default FPGA part (xc7a100tcsg324-1) simply issue the synthesise command with the target library and entity:

```
(Cmd) synthesise lib_max_hold.max_hold
```

To build the Max Hold project using *Altera Quartus*, issue the synthesise command with the flow set to 'quartus' and the part set to 'EP3C40F484C6'.

```
(Cmd) synthesise lib_max_hold.max_hold quartus EP3C40F484C6
```

To build the Max Hold project using *Xilinx ISE*, issue the synthesise command with the flow set to 'ise' and the part set to 'xc6slx9-csg324-2'.

```
(Cmd) synthesise lib_max_hold.max_hold ise xc6slx9-csg324-2
```

While the build is running any messages generated by the synthesis tool will be displayed in the ChipTools command line. When the build has completed ChipTools will store any build outputs in a timestamped archive in the synthesis output directory specified in the project settings:

```
[INFO] Build successful, checking reports for unacceptable messages...
[INFO] Synthesis completed, saving output to archive...
[INFO] Added: max_hold_synth_151215_134719
[INFO] ...done
(cmd)
```

If there is an error during build, ChipTools will store any outputs generated by the synthesis tool in a timestamped archive with an 'ERROR' name prefix.

CHAPTER 4

Project Format

ChipTools supports an XML project format that is intended to allow the designer to express an FPGA project configuration in a tool-agnostic manner. This means that the same project file should work for different synthesis or simulation tools when used with ChipTools. Certain *tool specific* data is unavoidable in FPGA designs, so the project format allows argument-passing to specific tools or inclusion of tool-specific data such as constraints or netlists without affecting other tools that may be supported.

The *XmlProjectParser* class implements the XML parser for ChipTools project files. It can be used either through the ChipTools command line interface via the load project command, or via a Python script by importing it directly.

Command line interface:

```
$ chiptools
(cmd) load_project my_project.xml
```

Python import:

```
from chiptools.core.project import Project

my_project = Project()
my_project.load_project('my_project.xml')
```

class chiptools.parsers.xml_project.**XmlProjectParser**

The XmlProjectParser class implements an XML parser for project files using the following format.

Note: All paths appearing in a project file are relative to the location of the project file.

<project>

The project parent tag encapsulates all configuration and file tags belonging to a project file. Existing project files may be imported into a project by including a project tag with the *path* attribute pointing to the existing XML file.

Attribute	Value	Description
synthesise	True	(default) These files are included for synthesis.
	False	Exclude these files from synthesis.
path	*string*	Path to existing project file to include.

<library>

The *library* tag is used to group all child file tags into the same library. If a file is not associated with a library it will default to the *work* library.

Attribute	Value	Description
synthesise	True	(default) These files are included for synthesis.
	False	Exclude these files from synthesis.
name	*string*	(required) Name of the HDL library for these files.

<file>

The *file* tag is used to define a source file to include in the project. Source files can either be text based HDL source files (VHDL or Verilog) or they can be netlists. Tool wrapper plugins will check the file extension to determine how they should process the file, for example .VHD and .V files will be processed with vcom and vlog respectively by Modelsim and .ngc files will be copied into the synthesis folder by ISE.

Attribute	Value	Description
synthesise	True	(default) This file is included for synthesis.
	False	Exclude this file from synthesis.
path	*string*	(required) Path to the source file.

Note: If a file tag is used outside of a library tag the file will automatically be associated with the *work* library and a warning will be displayed.

Note: File tags support additional optional attributes of the form *args_toolname_compile* where *toolname* is the name of a specific tool wrapper (for example: *modelsim*). The attribute value is passed to the simulation tool during compilation if it is the selected tool. *args_modelsim_compile='-2008'* would pass the command line arg '-2008' to Modelsim when it compiles the file.

<constraints>

The *constraints* tag defines the path to a constraints file to be included when performing synthesis on the project. Constraints can be limited to a specific synthesis tool via use of the *flow* attribute.

Attribute	Value	Description
path	*string*	(required) Path to the constraints file.
flow	*string*	(optional) Name of the associated synthesis tool.

<unittest>

The *unittest* tag defines the path to a Python based unit test suite to be included in the project test suite. Unit tests must be valid Python files with a .py extension. If the file is invalid or contains syntax errors it will be excluded from the project test suite. Runtime errors occurring from a unit test will result in that test failing.

Attribute	Value	Description
path	*string*	(required) Path to the unit test file.

<generic>

The *generic* tag defines a generic value setting for the top level entity during synthesis. Generic attribute names map to the name of a generic on the top level entity and the associated value is passed as the generic value.

Attribute	Value	Description
(name)	(value)	Set top level generic *name* to *value* at synthesis.

<config>

The *config* tag defines a config value setting for the project. Config attribute names map to the name of a configuration item in the project and the associated value is passed as the config value.

Attribute	Value	Description
(name)	(value)	Set the configuration item *name* to *value*.

The following configuration items can be set in a project:

Config	Description
simulation_directory	Directory to use as simulation working directory.
synthesis_directory	Directory to use as synthesis working directory.
simulator	Default simulator to use for this project.
synthesiser	Default synthesiser to use for this project.
part	FPGA part to target when performing synthesis.

In addition to the above configuration items, the *config* tag also allows tool-specific argument passing through the use of config attributes using the following naming convention: *args_toolname_flowname*, where *toolname* is the name of the tool to target and *flowname* is the name of a specific tool flow stage. For example: args_ise_par='-mt 4 -ol high -xe n' would pass the arguments *-mt 4 -ol high -xe n* to the place and route stage of an ISE synthesis flow. Each tool wrapper implements its own specific flow stage names.

Note: If a configuration item is already defined any new definitions will be ignored. A warning will be displayed if a redefinition is attempted.

CHAPTER 5

Testing

The Python Unittest module provides access to a powerful unit testing framework that can be extended to test FPGA firmware designs. ChipTools provides the **ChipToolsTest** class which extends **unittest.TestCase** to allow the automated execution and checking of FPGA firmware simulations. Any tests that you define should inherit the ChipToolsTest class, which can be found in **chiptools.testing.testloader**. ChipTools is re-using the existing Python Unittest framework which means that the rich ecosystem of testing tools provided by the Python community can now be used on your FPGA designs.

Before attempting to create your own FPGA unit tests in ChipTools you should first acquaint yourself with the Python Unittest framework to understand how the test flow and assertions work.

Test Flow

The Unittest TestCase defines a set of pre-defined functions that are called during different points in the test, this allows the designer to prepare the test environment and inputs, execute the tests and then clean up temporary files and close unused processes. The full detail of the pre-defined setUp/tearDown functions can be found in the Python Unittest docs, a typical test flow is given below:

1. setUpModule
2. setUpClass
3. setUp
4. *<user_test_1>*
5. tearDown
6. setUp
7. *<user_test_2>*
8. tearDown
9. tearDownClass
10. tearDownModule

Using ChipToolsTest

Assertion Based Tests

You may already have some testbenches for a design that use a self-checking approach that prints messages to the simulator transcript. Incorporating tests like these into the unit test framework is a simple exercise as ChipTools provides access to the simulator stdout and stderr streams as well as the simulator return code:

```
import re
from chiptools.testing.testloader import ChipToolsTest


class TestSimulatorStdout(ChipToolsTest):
    duration = 0  # Run forever
    library = 'my_test_lib'  # Testbench library
    entity = 'my_testbench'  # Entity to simulate

    def test_simulator_stdout(self):
        # Run the simulation
        return_code, stdout, stderr = self.simulate()

        # Check return code
        self.assertEquals(return_code, 0)

        # Check stdout for 'Error:' using regex
        errors = re.search('.*Error:.*', stdout)
        self.assertIsNone(errors)
```

This is one of the simplest tests you can define although it is also fairly limited. ChipTools allows you to make this approach slightly more flexible by providing a way to override generics/parameters before the test is run:

```
import re
from chiptools.testing.testloader import ChipToolsTest


class TestSimulatorStdout(ChipToolsTest):
    duration = 0  # Run forever
    library = 'my_test_lib'  # Testbench library
    entity = 'my_testbench'  # Entity to simulate
    generics = {'width' : 3) # Default generic width to 3

    def check_simulator_stdout(self):
        # Run the simulation
        return_code, stdout, stderr = self.simulate()

        # Check return code
        self.assertEquals(return_code, 0)

        # Check stdout for 'Error:' using regex
        errors = re.search('.*Error:.*', stdout)
        self.assertIsNone(errors)

    def test_width_5(self):
        self.generics['width'] = 5
        self.check_simulator_stdout()

    def test_width_12(self):
```

```
        self.generics['width'] = 12
        self.check_simulator_stdout()
```

By using simple test cases like these you are able to re-use your existing self-checking testbenches and define new test cases for them by modifying parameters/generics or stimulus files through ChipTools.

Model Based Tests

One of the big benefits of using Python is that you have access to a wide range of open source libraries that can assist with test development; for example you could use Python to model the expected behavior of a system such as a signal processing pipeline or cryptographic core. You can incorporate such models into the ChipTools test framework and use them to generate sets of stimulus which can be fed into your testbench, and you can then check the simulation response against the model response to determine whether or not the implementation is correct:

```
import numpy as np
from chiptools.testing.testloader import ChipToolsTest

class TestFastFourierTransform(ChipToolsTest):

    duration = 0  # Run forever
    library = 'my_test_lib'  # Testbench library
    entity = 'my_testbench'  # Entity to simulate
    N = 1024  # Our fixed FFT size
    generics = {'n' : N)

    def test_noise(self):
        values = np.random.randint(0, 2**16-1, self.N)
        self.run_fft_simulation(values)

    def test_sinusoid(self):
        f = 10
        values = np.sin(2*np.pi*f*np.linspace(0, 1, self.N))
        self.run_fft_simulation(values)

    def run_fft_simulation(self, values):
        out_path = os.path.join(self.simulation_root, 'fft_out.txt')
        in_path = os.path.join(self.simulation_root, 'fft_in.txt')

        # Create the stimulus file
        with open(in_path, 'w') as f:
            for value in values:
                f.write('{0}\n'.format(value))

        # Run the simulation
        return_code, stdout, stderr = self.simulate()

        # Check return code
        self.assertEquals(return_code, 0)

        # Open the simulator response file that our testbench created.
        with open(out_path, 'r') as f:
            actual = [float(x) for x in f.readlines()]

        # Run the FFT model to generate the expected response
        expected = np.fft.fft(values)
```

```
        # (compare our actual and expected values)
        self.compare_fft_response(actual, expected)
```

The example above demonstrates how you might check a common signal processing application using a Fast Fourier Transform. By using this approach a large suite of stimulus can be created to thoroughly check the functionality of the design.

External Test Runners

Perhaps you would like to set up a continous integration system such as Jenkins to execute your tests on a nightly basis. ChipTools makes this easy to do by allowing your unit tests to be run using external test runners like Nosetests or Pytest. To enable a unit test to be run using an external test runner simply add a *project* attribute to the test class which provides a path to a valid ChipTools XML project file defining the files and libraries required by the simulation environment:

```
import numpy as np
import os
from chiptools.testing.testloader import ChipToolsTest

class TestFastFourierTransform(ChipToolsTest):

    duration = 0
    library = 'my_test_lib'
    entity = 'my_testbench'
    base = os.path.dirname(__file__)
    project = os.path.join(base, 'my_project.xml')
```

Test cases that do not provide a *project* attribute will not be able to be run using an external runner.

ChipToolsTest Class Detail

class chiptools.testing.testloader.**ChipToolsTest**(*methodName='runTest'*)

The *ChipToolsTest* class is derived from unittest.TestCase and provides a base class for your unit tests to allow them to make full use of ChipTools.

When creating a unit test class you should override the *duration*, *generics*, *library* and *entity* attributes to define the test configuration for the simulator. Individual tests can redefine these attributes at run-time to provide a powerful testing mechanism for covering different configurations.

A brief example of a basic unit test case is given below:

```
>>> from chiptools.testing.testloader import ChipToolsTest
>>> class MyBasicTestCase(ChipToolsTest):
...     duration = 0  # Run forever
...     generics = {'data_width' : 3}  # Set data-width to 3
...     library = 'lib_tb_max_hold'  # Testbench library
...     entity = 'tb_max_hold'  # Entity to simulate
...     # Defining a path to a project allows us to run this test case
...     # with Nosetests etc. as well as through ChipTools.
...     project = os.path.join('max_hold.xml')
...     # The setUp method is called at the beginning of each test:
...     def setUp(self):
...         # Do individual test set-up here
...         pass
```

```
...     # Methods starting with 'test_' are considered test cases:
...     def test_max_hold(self):
...         # Run the simulator
...         return_code, stdout, stderr = self.simulate()
...         self.assertEqual(return_code, 0)  # Check error code
...         # More advanced checks could search stdout/stderr for
...         # assertions, or read output files and compare the
...         # response to a Python model.
...         pass
...     # The tearDown method is called at the end of each test:
...     def tearDown(self):
...         # Clean up after your tests here
...         pass
```

For a complete example refer to the Max Hold example in the examples folder.

duration = 0
The duration attribute defines the time in seconds that the simulation should run for if the chosen simulator supports this as an argument during execution. If a time of 0 is specified the simulation will run until it is terminated automatically by the testbench. e.g. To fix simulation time at 10ms set duration to 10e-3

entity = None
The *entity* attribute defines the name of the top level component to be simulated when running this test. The entity should name a valid design unit that has been compiled as part of the project.

generics = {}
The *generics* attribute is a dictionary of parameter/generic names and associated values. These key, value pairs will be passed to the simulator to override top-level generics or parameters to customise the test environment:

```
>>> generics = {
...     'data_width' : 3,
...     'invert_bits': True,
...     'test_string': 'hello',
...     'threshold': 0.33,
>>> }
```

The *generics* attribute can also be used to dynamically specify parameters for individual tests to check different configurations in your testbench:

```
>>> def test_32_bit_bus(self):
...     self.generics['data_width'] = 32
...     self.simulate()
>>> def test_16_bit_bus(self):
...     self.generics['data_width'] = 16
...     self.simulate()
```

static get_environment(*project*, *tool_name=None*)
Return the simulation environment items from the supplied project instance as a tuple of (simulator, simulation_root, libraries).

library = None
The *library* attribute defines the name of the library in which the top level component to be simulated exists.

load_environment(*project*, *tool_name=None*)
Initialise the TestCase simulation environment using the supplied Project reference so that the individual tests implemented in this TestCase are able to compile and simulate the design.

project = None

The *project* attribute is optional, but if used it should supply an absolute path to a valid ChipTools Project XML file that defines the libraries and source files that make up the design that this test case belongs to. This attribute is required when the test case is executed directly by an external test runner instead of ChipTools, as it will be used to prepare the simulation environment for the external test runner.

The following provides a convenient way of setting the project path so that your test can be run from any directory:

```
>>> from chiptools.testing.testloader import ChipToolsTest
>>> class MyUnitTest(ChipToolsTest)
...     base = os.path.dirname(__file__)
...     # Now use os.path.join to build a relative path to the project.
...     project = os.path.join(base, '..', 'my_project.xml')
```

classmethod setUpClass()

The *setUpClass* method prepares the ChipTools simulation environment if it has not already been loaded.

If this test case is loaded via the ChipTools Project API it will be initialised via a call to the *load_environment* method, which pulls the simulation environment information from the parent Project instance.

If this test case is loaded via an external tool such as Nosetests the setUpClass method will attempt to load the project file pointed to by the project path stored in the *project* attribute. When you create your test case you can specify this attribute in your test class to allow an external runner like Nosetests to call your test cases.

If the environment was not already initialised by ChipTools and a valid project file path is not stored in the *project* attribute, this method will raise an EnvironmentError and cause your test to fail.

This method overloads the unittest.TestCase.setUpClass classmethod, which is called once when a TestCase class is instanced.

simulate()

Launch the simulation tool in console mode to execute the testbench.

The simulation tool used and the arguments passed during simulation are defined by the test environment configured by the test case and the *Project* settings. When the simulation is complete this method will return a tuple of (*return_code*, *stdout*, *stderr*) which can be used to determine if the test was a success or failure. For example your testbench may use assertions to print messages during simulation, your Python TestCase could use regex to match success of failure criteria in the stdout string:

```
>>> def test_stdout(self):
...     return_code, stdout, stderr = self.simulate()
...     # Use an assertion to check for a negative result on a search
...     # for 'Error:' in the simulator stdout string.
...     self.assertIsNone(re.search('.*Error:.*', stdout))
```

simulation_root

The *simulation_root* property is an absolute path to the directory where the simulator is invoked when simulating the testbench. Any inputs required by the testbench, such as stimulus files, should be placed in this directory by your TestCase. Similarly, any outputs produced by the testbench will be placed in this directory.

For example, to build paths to a testbench input and output file you could do the following:

```
>>> def setUp(self):
...     self.tb_in = os.path.join(self.simulation_root, 'input.txt')
...     self.tb_out = os.path.join(self.simulation_root, 'output.txt')
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index