Wavetrace User Manual

Introduction

Wavetrace is a platform independent real time FPGA debug tool. It acts as an internal logic analyzer, allowing one to capture cycle-by-cycle data from internal FPGA nets and registers and display these as a waveform.

Motivation

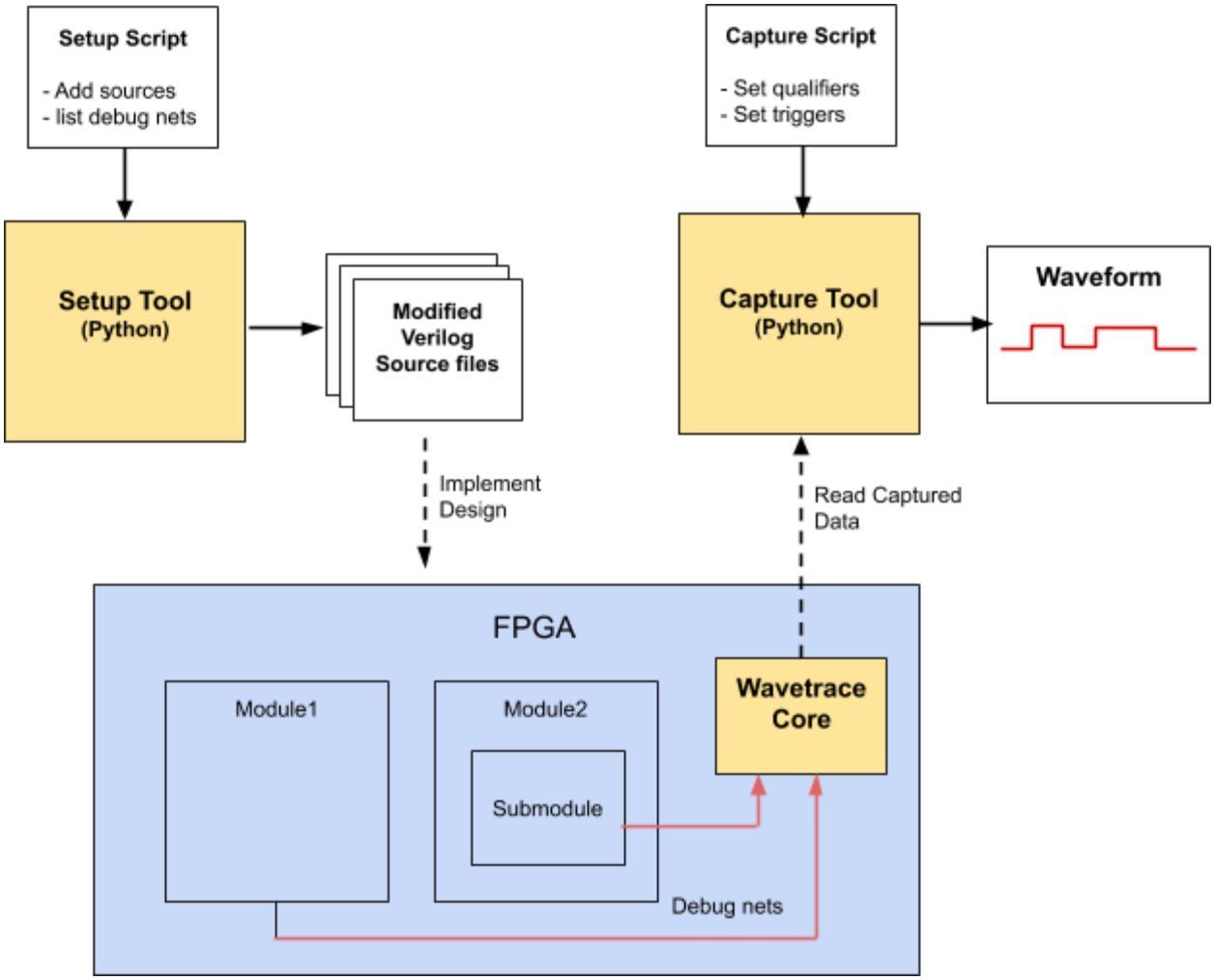
Similar vendor specific tools are typically provided as black boxes with zero transparency. They have limited flexibility, and have proven to be buggy and somewhat unreliable. Having access to the debug tool's source code allows it to be easily extended or modified to support unique debugging situations. Also, having a platform independent debug tool reduces the overhead of transitioning from one FPGA vendor to another.

Features

- Parses and modifies pre-synthesis Verilog code
- Single or continuous capture modes
- Supports multiple trigger conditions
- Supports multiple conditional storage qualifiers
- Supports subsampling of captured data to trade-off capture resolution versus duration
- Run time configuration of triggers, subsampling, and storage qualifiers
- Arbitrary capture depth using any number of available BlockRAMs

How it Works

Wavetrace can be broken into three components: a Python based setup tool, a Python based capture tool, and a Verilog core. The user defines a set of nets within their Verilog design and provides these as an input to the setup tool. The setup tool analyzes and parses the Verilog code, identifies the nets within the hierarchy, and generates modified Verilog code with these nets routed up to the top level. It also instantiates the Wavetrace debug Verilog core with the debug nets as inputs. The Capture tool interacts with the Verilog core via a UART and is responsible for configuring the core to perform a specific type of capture, it sets up trigger conditions, and monitors the capture status. When a data set has been captured, it reads back the data for each debug net and displays this on a waveform.



Overview of the three Wavetrace components showing the Setup Tool, the Capture Tool, and the Verilog Core instantiated inside an FPGA design

Installation

Wavetrace requires Python 2.7 with PySerial installed as well as GTKWave. The source code for Wavetrace resides in the FPGA Common repository. The repository can be cloned with:

```
> git clone sso://user/obowen/fpga-wavetrace
```

Once the repository is installed, Wavetrace should be added to the local Python path. On Ubuntu systems, the following can be added to the .bashrc file assuming the repo was cloned in a 'src' directory:

```
PYTHONPATH=$PYTHONPATH:~/src/fpga-wavetrace
```

Hardware Requirements

Wavetrace makes use of a UART module instantiated in the FPGA logic. This requires access to two pins for TX and RX, which should be routed to 3.3V TTL pins. A USB to Serial converter cable such as FTDI can be used to connect the UART to a host PC. The three required connections are Ground, TX, and RX.

Using Wavetrace

Wavetrace makes use of two python scripts, one for setup, and one for capture.

The Setup Script

The setup script defines the top level, points Wavetrace to the verilog sources, and lists the nets for debug. Below is an example setup script called.

```
"""Wavetrace debug setup script."""

from wt_setup import WTSetup

# Create wavetrace instance and provide  configuration
# NOTE: The clock frequency parameter must match the design's clock speed!
wt = WTSetup(clock_freq=100.0, pre_trig_depth=1024, capt_depth=4096)

# Point wavetrace to the source code for this project
wt.add_sources("src/hdl/")
wt.add_sources("project/top.v")

# Specify the top level, which is where the debug core gets inserted
wt.top("top")

# Tell wavetrace which nets to use for its clock and reset
wt.clk("clk")
wt.rst("rst")

# Select the nets for debug using the module instantiation hierarchy
wt.net("top.module1.count[7:0]")
wt.net(“top.module1.din_valid")
wt.net(“top.module1.din_ready")
wt.net(“top.module1.din_data[7:0]”)

# generate the modified source code with the debug nets hooked up
wt.generate()
```

Example wavetrace setup script

The setup script can then be run from the command line with

```
> python debug_setup.py
```

The script will output results to the console and report any errors. The script parses the Verilog code and generates a modified version of the sources with the debug nets routed up to the top level. The setup tool also instantiates the Wavetrace core and connects the UART pins to top level ports.

Modifying the FPGA Project

Once the setup script has run, the FPGA project should be modified to reference the generated output files and new top level. The following .tcl commands show an example of how this can be accomplished in Xilinx Vivado:

```
# define path to common FPGA library
set fpga_common_dir "~/src/fpga/common"

# remove top level file
Remove_files top.v

# read sources required for the wavetrace core
read_verilog $fpga_common_dir/hdl/stream/stream_serializer.v
read_verilog $fpga_common_dir/hdl/stream/stream_deserializer.v
read_verilog $fpga_common_dir/hdl/fifo/stream_fifo_1clk.v
read_verilog $fpga_common_dir/hdl/fifo/fifo_1clk.v
read_verilog $fpga_common_dir/hdl/ram/ram_1c_1r_1w.sv
read_verilog [glob $fpga_common_dir/tools/wavetrace/verilog/*.v]

# read modified source files generated by the Wavetrace setup tool
read_verilog [glob wavetrace/output/*.v]
```

Example .tcl command to modify an FPGA design to use Wavetrace sources

The Capture Script

The capture script interacts with the FPGA Wavetrace core and sets up trigger conditions and storage qualifiers. It then monitors the core, waits for the trigger conditions to be met, and reads the contents of the capture buffer and displays the results as a waveform using GTKWave. Below is an example capture script:

```
"""Wavetrace debug capture Script."""

import os
import sys
from wt_capture import WTCapture

if len(sys.argv) < 2:
```

```
  print "Usage: python debug_capture.py [SERIAL_PORT]"
  exit(1)
port = sys.argv[1]

wt = WTCapture(port, "output/signal_list.txt")

# Specify the trigger(s) to use for this capture
wt.add_trigger("sys_i.framer.fsync_rx.dout_valid", wt.RISE_EDGE)

# perform the capture
wt.capture()

# launch GTKWave to display the results
os.system("gtkwave output/wavedump.vcd -a output/wavedump.gtkw")
```

Example Wavetrace capture script

Wavetrace WTSetup Module

For exact function details, see the WTSetup source code. At some point, the documentation should be generated from the source code… For now, here is an overview of the module's functions:

Function	Description
`__init__()`	Constructor. Arguments: `clock_freq`: clock frequency of the capture clock `uart_baud`: baud rate of the uart connecting to the debug module `pre_trig_depth`: depth of the pre-trigger capture memory `capt_depth`: depth of the post-trigger capture memory
`add_sources()`	Adds verilog sources and searches them for module definitions. Arguments: `path`: this can be a single Verilog file, or a directory. If it is a directory, all verilog files within it, and any subdirectories, are added too.
`rm_sources()`	Removes verilog sources that have been added via add_sources. This can be useful if there are multiple definitions for certain modules and entire directories were added with add_sources. Arguments: `path`: this can be a single Verilog file, or a directory. If it is a directory,

	all verilog files within it, and any subdirectories, are added too.
top()	Sets the top-level module for the wavetrace debugger. Arguments: `module_type`: The type of the module eg. 'sys_top'. This module should only be instantiated once in the design.
clk()	Selects the capture clock. Arguments: `clk_net`: the top-level net to be used as the capture clock.
rst()	Selects the reset net for the wavetrace module. Arguments: `rst_net`: the top-level net to be used as the reset
net()	Adds one or more debug nets. Arguments: `base`: (Optional) A hierarchical base path to be prefixed to all nets `nets`: A single hierarchical path for a net, or a list of net paths. Net paths must include the bit range, eg. "instance1.instance2.somenet[3:0]"

Wavetrace WTCapture Module

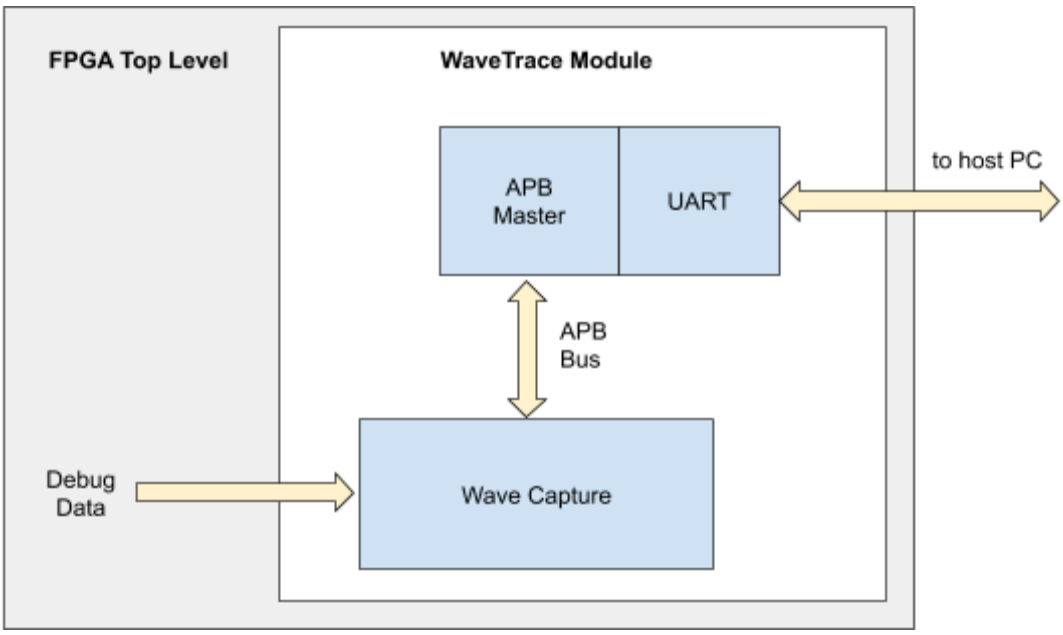
For exact function details, see the WTCapture.py source code. At some point, the documentation should be generated from the source code… For now, here is an overview of the module's functions:

Function	**Description**
__init__()	Constructor Arguments: `serial_port`: the serial port connected to the FPGA's wavetrace uart, example "/dev/ttyUSB2" `signal_list_filename`: the filename containing the list of signals being debugged, this should be generated by the setup tool.

add_trigger()	Adds a trigger to be used for the next capture. Arguments: `trig_net`: the net name and bits for the trigger, for example "inst_0.counter[7:4]" `trig_value`: either a value, or a string constant, indicating the value required on the 'trig_net' for the trigger to fire. A 'rising_edge' or 'falling_edge' string can be used to trigger on either a rising or falling edge instead of a value.
add_qualifier()	Adds a storage qualifier to be used for the next capture Arguments: `qual_net`: the net name and bits for the storage qualifier, for example "inst_0.counter[0]" `qual_value`: the value the 'qual_net' must match in order for the sample to be stored.
set_sub_sampling()	Set ratio of sub-sampling for data captured after all storage qualifiers have been applied. This allows a trade-off between captured depth and captured resolution. Arguments: `ratio`: the subsampling ratio, '1' will capture all samples, a ratio of '2' will capture every second sample, etc.
capture()	Performs the Wavetrace capture sequence: - stops any pending captures - sets the trigger conditions - enables the capture - waits for the trigger to be hit and the capture to complete - reads the captured data - writes waveform file and GTKWave config file Arguments: trig_mode: trigger mode, either none, single, or continuous

Wavetrace FPGA Core Architecture

The Wavetrace core gets inserted in the top level by the setup script. Internally, it makes use of a capture submodule and an APB to UART conversion to get data to and from the host PC.



The capture submodule consists of trigger detection, storage qualifier, and down-sampling logic. A fifo is used to hold data before a trigger event and a configurable depth capture buffer stores cycle-by-cycle data after trigger conditions have been met.