TURQUOISE: A VHDL CODE ANALYZER & COMPILATION TOOLCHAIN

By Trung Truong, ECE '21

Introduction

VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a programming language that describes the logical make-up of Field-Programmable Gate Array (FPGA) and integrated circuits (IC). While VHDL is very efficient in describing hardware, the language can be difficult for beginners to get started due to the concurrent nature of the language, non-elegant syntax, and unclear error messages. Furthermore, as most popular VHDL development toolchains are commercialized software (i.e. Modelsim, Xilinx Vivado, etc.), it is difficult to customize the software to fit beginners' needs (i.e. verbose error message, ease of development, etc.).

To target the issues above, we seek to design a cross-platform, open-source VHDL development tool and create a VHDL compilation pipeline that emphasizes ease of development and safe code. In this paper, the overall architecture of the final product will be discussed. Furthermore, a comprehensive list of features that our implemented tool provides will be listed.

Related work

There are tools on the market that performs static code analysis on VHDL, including VHDL Tool [1], Sigasi, and the SAVE project [2]. However, these tools are not open-source, and it can be difficult to customize these software to integrate in a curriculum targeting beginners (such as ES-4).

Meanwhile, there are open-source tools on the market that perform VHDL compilation, simulation, and synthesis, such as GHDL [3], and Yosys. While these software are open for public usage, they are difficult to configure, and it can be very challenging and time-consuming for beginners to set up the compilation pipeline.

Our approach

To mitigate these problems from available software, we seek to develop an easy-to-use program (`turquoise`) that targets both static code analysis and VHDL compilation. More specifically, this program seeks to provide ease of development by allowing developers to develop VHDL, run simulation from developed code, upload compiled VHDL to the targeted hardware (Upduino v3 FPGA board).

Targeted errors

While there are a lot of mistakes that beginners often make when developing VHDL, we seek to focus on a subset of errors that we believe are the most relevant. More specifically, the following linter features are currently supported with the (`turquoise`) static code analyzer:

<u>Syntax check</u>

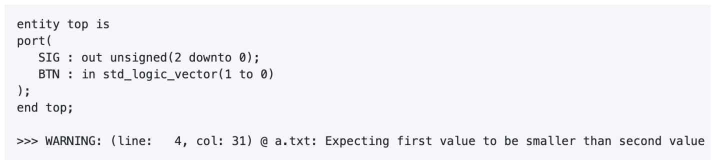
(`turquoise`) supports full syntax check for the 1987, 1993, 2002 versions of the IEEE 1076 VHDL standard (and partially the latest 2008 revision) through (`GHDL`) front-end.

<u>Primitives check</u>

(`turquoise`) performs syntax and semantic check for certain primitive types and displays errors, warning, and info when necessary

Current primitives supported includes `std_logic, std_logic_vector, bit, signed, unsigned, integer, boolean, time, string`

Example: for the following snippet, the linter returns this warning



```
entity top is
port(
   SIG : out unsigned(2 downto 0);
   BTN : in std_logic_vector(1 to 0)
);
end top;

>>> WARNING: (line:   4, col: 31) @ a.txt: Expecting first value to be smaller than second value
```

Fig 1. Primitives warning

<u>Entity/Component typecheck</u>
(`turquoise`) performs signals type check when comparing `entity` declaration and `component` instantiation, and displays errors, warning, and info when necessary

<u>Port map typecheck</u>
(`turquoise`) performs signals type check when mapping signals in `port map` declaration, and displays errors, warning, and info when necessary

<u>Signal check</u>
(`turquoise`) performs the following signals type check: signals declared but not used, signals assigned but not declared

<u>Package check</u>
(`turquoise`) performs the following package check: deprecated package, duplicated package import

Design flow

To achieve the goal of creating a cross-platform static code analyzer and compiler toolchain that eases the VHDL development process, the software architecture has been divided into some key components, as seen in the block diagram.

In the beginning, when developers seek to statically check their code, they can run the static code analysis tool through a command line interface. This linter will take in VHDL files, create an abstract syntax tree using `pyVHDLParse`r (a Python package to parse VHDL), and run subroutines that detect faulty code.

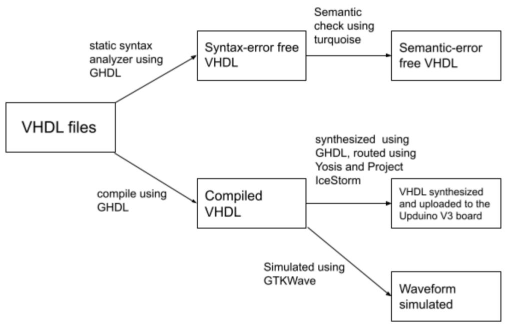

VHDL files
static syntax analyzer using GHDL
Syntax-error free VHDL
Semantic check using turquoise
Semantic-error free VHDL
compile using GHDL
Compiled VHDL
synthesized using GHDL, routed using Yosis and Project IceStorm
VHDL synthesized and uploaded to the Upduino V3 board
Simulated using GTKWave
Waveform simulated

Fig 2. Design flow

Developers can also opt out of the static code analysis and compile the VHDL code directly. In order to compile and synthesize VHDL, (`turquoise`) is using GHDL, an open-source, cross-platform VHDL compiler. In order to flash to Upduino v3 board (which is the targeted FPGA board for ES4), the `GHDL Yosys` plugins and `Project IceStorm` are used to generate bitstreams for the Lattice ICE40 chip from developed VHDL code.

In addition to the ability to flash programs on FPGA, the GHDL compiler also supports waveform generation, which gives developers the ability to simulate the behavior of logic circuits before flashing the code onto the board. To display the result of the simulation, our tool will use `GTKWave`.

Components

1. Compilation Toolchain

To construct the compilation toolchain, we use an open-source compiler called `GHDL`. `GHDL` is cross-platform, meaning it supports the most mainstream operating systems, including Linux, Windows and Apple OS X. It is also much more efficient than any other interpreted simulator, as it uses a code generator (llvm, gcc or a builtin one).

`GHDL` is also the most used open-source VHDL compiler and actively maintained, making it a good choice as it has been thoroughly battle-tested. Lastly, although GHDL only compiles and simulates VHDL, it provides a set of tools that integrate nicely with flashing bitstreams onto the Lattice ICE40 chip family, including the `GHDL Yosys` plugins and `Project IceStorm` [4].

2. Waveform simulation

To display the simulated waveforms, we use a tool called `GTKWave`. `GTKWave` is a graphical user interface application that displays simulated output generated by `GHDL`. More specifically, `GHDL` outputs `.vcd` files after simulation, and `GTKWave` reads in such files and outputs the simulated waveforms on the graphical user interface.

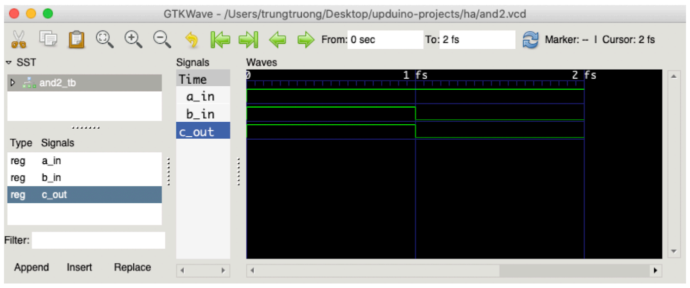


Fig 3. GTKWave GUI

3. Tokenizer

In order to perform static code analysis, we must break a VHDL file into small tokens, or tokenize, for parsing. This is done through the `Tokenize.py` module, which can be found at the project's source code [5]. This module uses `pyVHDLParser` (a 3rd -party Python package) for tokenization.

4. State machine

In order to parse through complicated syntax, we design a state machine or Deterministic Finite Automata (DFA) module `State.py` that triggers a state transition whenever it encounters a known input. This allows the static code analyzer to step through the tokenized VHDL file, and statically analyze the VHDL code snippet by cross-referencing the parsed output to see if it conforms to VHDL semantics.

5. Type check and error generation

As the state machine finishes parsing through generated VHDL tokens, we designed a module called `TypeCheck.py` and `UsedCheck.py` that perform entity-component and port map type check, as well as signal check and package check as listed in the *Targeted Errors* section.

Conclusion

In conclusion, our group has successfully implemented a cross-platform, open-source VHDL toolchain that provides interactive warning and error messages, as well as simulation and synthesis capabilities. Although there have been solutions on the market for linting and compiling VHDL, our tool has an edge when it comes to being beginner-friendly as it focuses on ease of usage and targets relevant mistakes. In the future, we hope to see this tool being used in both classroom setting (such as in ES-4) and intermediate VHDL projects.

References

1. VHDL-Tool. (2021). Retrieved May 4 from https://www.vhdltool.com

2. Mastretti M., Busi M.L., Sarvello R., Sturlesi M., Tomasello S. (1996) Static Analysis of VHDL Source Code: the SAVE Project. In: Bologna S., Bucci G. (eds) Achieving Quality in Software. IFIP — The International Federation for Information Processing. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-34869-8_11

3. Gingold, T. (2020). GHDL Documentation Release 1.0 - dev https://ghdl.readthedocs.io/_/downloads/en/latest/pdf/

4. Wolf, C. (n.d.). Yosys Manual. Retrieved May 4, 2021, from http://www.clifford.at/yosys/files/yosys_manual.pdf

5. Turquoise team. Turquoise source code. Retrieved May 4, 2021 from https://github.com/ttrung149/turquoise