Skeletor Connector Language: Hierarchy Specification to HDL development made easy

Ivan Rodriguez-Ferrandez[1,2], Guillem Cabo[1,2], Javier Barrera[1,2], Jeremy Giesen[1,2], Alvaro Jover-Alvarez [1,2], Leonidas Kosmidis[2,1]
[1]Universitat Politècnica de Catalunya [2]Barcelona Supercomputing Center (BSC)

***Abstract*—In this paper, we present Skeletor, an open-source EDA tool which can be used in the early design phases of large RTL designs in order to increase both individual and team productivity and minimise programmer mistakes by generating structures of Verilog projects from a specification. With a single description file, Skeletor generates templates of the main modules of a design, their correct interconnection and the files associated with the workflow such as templates for test benches, scripts, and headers. This is achieved with a simple syntax similar to Verilog and C++.**

Skeletor does not aim to replace any HDL but instead offers the capability to automate repetitive tasks such as the generation of workflow associated files or redundant code associated with hard-typed language, while minimizing the friction between the user and the tool.

I. Introduction

When it comes to the design of digital circuits using hardware description languages (HDL), developers usually experience several problems. Traditional HDLs are strongly typed and verbose, characteristics that slow down developers' work. On top of that, each implemented module requires additional files such as test benches, compilation and simulation scripts, header files for parameters and constants. Most of these files contain duplicated information which could be extrapolated from the top file if only some small changes would be added. In addition to the problem of overlapping information between files, writing by hand all of them could lead to human errors, slowing down development even further.

The idea of abstracting traditional HDL languages has been around for a long time. High-level HDL languages exist [1][2] and are becoming more important, especially for FPGA development, but when considering ASIC design, most companies prefer to stick with VHDL or Verilog since using them doesn't incur any loss of control over the resultant hardware.

Our proposal aspires to become a robust automation tool which helps to reduce as much as possible repetitive tasks, allow for better work distribution among the team members, improve coding style and set good coding practices without constraining the programmer with an additional layer of abstraction.

II. Tool Overview

Skeletor, is an open-source EDA tool which can be used to bootstrap large RTL projects, frequently implemented by a team of several developers. In such scenarios, the hardware product which is going to be implemented is divided in several hardware modules, which can be developed and tested in parallel by different people. Usually, each module is implemented in a single file, or it is hierarchically composed by other modules implemented in other source files.

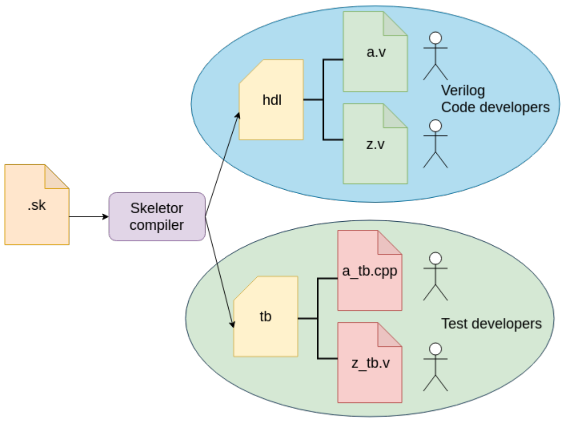


Figure 1. Generation of files by the Skeletor compiler.

Such a parallel development is possible by agreeing the functionality of each module, as well as their interfaces in order to be connected with other modules.

As shown in Figure 1, Skeletor allows the automatic generation of all the infrastructure required for the development and testing of the product from a specification file. In particular, Skeletor generates templates of the main modules of a design, their correct interconnection and the files associated with the workflow such as templates for test benches, scripts, and headers. This is achieved with a simple syntax similar to Verilog and C++. This way, it saves time from tedious module instantiations in HDL and their error prone interconnections, as well as from the testing and infrastructure, such as the different scripts required to launch simulations. Consequently it increases both the individual and the overall team productivity.

It is important to note that Skeletor does not aim to replace any HDL, but instead to automate repetitive tasks as the generation of workflow associated files or redundant code associated with the use of hard-typed languages, while minimizing the friction between the user and the tool.

III. Skeletor compiler

Skeletor is implemented as a compiler based on Flex [3] and Bison [4][5]. This tool is intended to generate the *skeleton* of Verilog projects. We define as skeleton the set of files required in any HDL project including Verilog files, test benches, configuration files and scripts.

The Verilog files which are generated with our tool do not contain logic, they only specify interconnections between modules and instances. The generated files can then be modified and completed by the programmer to give to each module the required functionality.

Skeletor syntax is quite similar to Verilog. The key idea behind it is allowing the developer to specify which modules need to be designed and how they are going to be connected, in a similar way that it is specified in the top level file of any design.

A. *Lexical specification*

The language is composed of a small set of keywords, which are listed in Table I along with their brief description. Using only a few keywords was an intentional decision to simplify and ease the usage of the tool. Despite that, our configuration language is expressive enough to allow describing most of the typical interconnections found on HDL.

Lexic token	***Function***
$$$ reg a;$$$	Verilog Code to be dumped
module	Starts definition of a module
in	Input of a module
out	Output of a module
inout	Input Output of a module
'define	Define which will be translated to Verilog
#define	Define for pseudo code
#function	Short description of function
#description	detailed description
#coder	Name of the assigned programmer
#references	Documentation references for the design
wire	Connection between modules
->	Assign connections between ports and wires
top	Top module of the project
$\sim$	Negation of a signal

Table I
SKELETOR CONFIGURATION LANGUAGE KEYWORDS.

Since the configuration language is based in C we also have a lot of the standard operators and comments. The list of the available lexicon from C is shown in the listing of Table II.

Lexic token	***Function***
//	One line comment
*/ * My comment */*	Multi-line comment
+, -	Sum and subtraction operand
(,)	Open and close parenthesis
";", ":", ","	Punctuation
true, false	True and false bool state
==	Equality
!=	Non-equality
<, >, <=, >=	Less than, more than, less or equal than, more or equal than
**, /*	Multiplication and division
=	Assignation
[,], {, }	Brackets
&&	And operator
||	Or operator
!	Negation operator

Table II
SKELETOR CONFIGURATION LANGUAGE OPERATORS.

B. *Grammar specification*

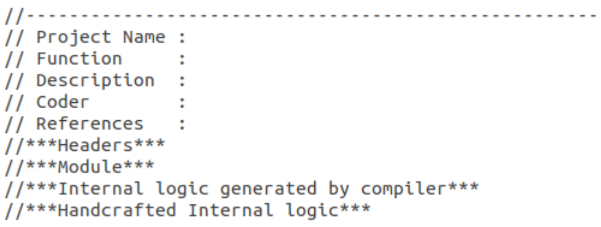
In this part we are going to explain the specific grammar of this language.

- The main structure in Skeletor is the *module*. All modules are considered equal in the hierarchy, but always one of them is marked as top, and as such it must be specified last in the list of modules.
- Each module can have between 0 parameters and *N* parameters and each one can be initialised with a define which will be translated to Verilog or with a define in Skeletor pseudocode which will put the corresponding value in Verilog.
- Inside of each module are the in, out, inout definitions which defines the inputs and outputs of each individual module.
- Inside a module it is possible to call an instance of another module and make the connection. When a module instantiates another module, the connections between the two must be of the same type. That is an input can be connected only with an input and an output can be connected only with an output. Obviously the inout connection type can be used for both inputs and outputs.
- Inside a module if there are multiple instances and there is a need to connect them, the -> operator can be used. This operator makes connection with opposites types of connections. That is between an input with an output and vice versa. As in the previous case, the inout can be used without constraints.

C. *Code Conventions*

We understand that each programmer or organisation have their own preferences regarding coding conventions, which should be consistent in a large project to enhance readability and maintainability. In the current tool version we only support a default set of coding rules which is described below. However, we plan to support more flexibility in the future.

- Inputs are suffixed with _i, while outputs are suffixed with _o.
  E.g.: input clk_i,
- Wires which connect two modules are suffixed with the name of the modules that connect, an additional comment is introduced in order to keep the information of the signals that it connects.
  E.g.: wire rdy_a_d; // wiring between rdy_o of module a and clk_i of module d
- Defines are written in capital letters.
- Module names are in lowercase letters and with underscores between words.
  E.g.: module simple_example #(
- Parameters are written without underscores and only the first letter of each word in capitals.
  E.g.: TransAddrSize
- Project defines will be placed into a file called "defines.vh"
- All the files have five regions: a File description, Headers, Modules, Generated logic and Handcrafted logic. This file structure can be seen in Figure 2.



```
//-----------------------------------------------------
// Project Name :
// Function     :
// Description  :
// Coder        :
// References   :
//***Headers***
//***Module***
//***Internal logic generated by compiler***
//***Handcrafted Internal logic***
```

Figure 2. Default Sections of generated Verilog files for modules and top.

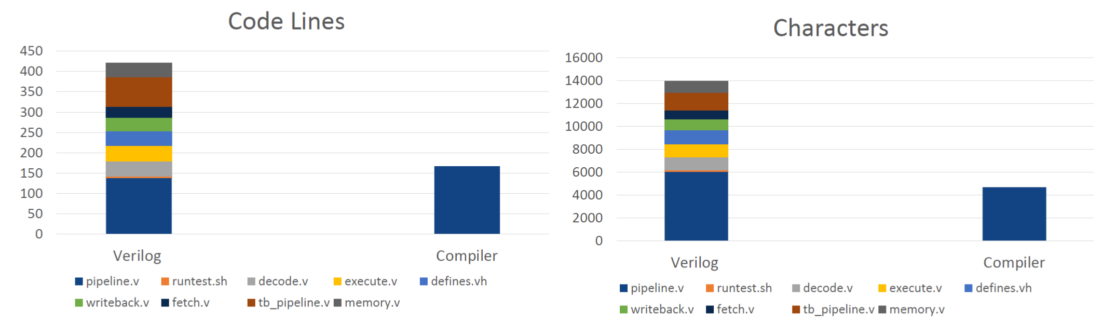
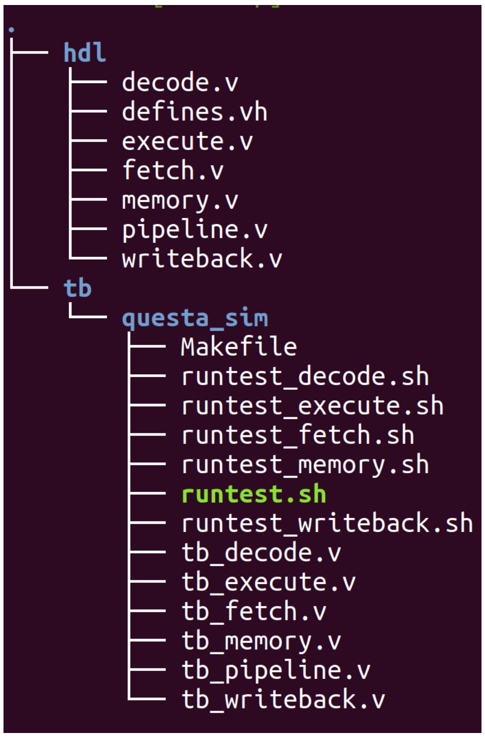


Figure 3. Code size comparison between Skeletor-generated code and the entire Verilog code of the complete processor pipeline project of Figure 6 in terms of LOC and characters.



```
.
├── hdl
│   ├── decode.v
│   ├── defines.vh
│   ├── execute.v
│   ├── fetch.v
│   ├── memory.v
│   ├── pipeline.v
│   └── writeback.v
└── tb
    └── questa_sim
        ├── Makefile
        ├── runtest_decode.sh
        ├── runtest_execute.sh
        ├── runtest_fetch.sh
        ├── runtest_memory.sh
        ├── runtest.sh
        ├── runtest_writeback.sh
        ├── tb_decode.v
        ├── tb_execute.v
        ├── tb_fetch.v
        ├── tb_memory.v
        ├── tb_pipeline.v
        └── tb_writeback.v
```

Figure 4. Generated files for the example of Figure 6.

D. Usage

Skeletor can be used as a command line tool for Linux-based systems. Our code [6] has minimal dependencies and can be compiled in any distribution.

Figure 4 shows the files generated for a simple processor pipeline, described in Figure 6 when invoking Skeletor with the flags -t -v -i. This means that Skeletor creates an individual testbench for each module compatible with the Verilator Verilog Simulator [7]. Skeletor can also generate testbenches for use with Mentor Graphic's QuestaSim [8]. All the available flags are shown in the Figure 5.

E. Syntax highlighting

In addition to the compiler, we provide a syntax highlighter for the Sublime Text editor [9], which can be seen in Figure 6. This facilitates the development of Skeletor configuration files. Files with the extension .sk are automatically recognized after the addition of the syntax package. We are also planning to support other widely used editors such as vim [10], emacs [11] and IDEs like eclipse [12] and Visual Studio [13].

IV. Evaluation

As an indication of the Skeletor compiler advantages, we provide in Figure 3 a comparison between the total amount of lines of code (LOC) and characters used to implement the simple processor pipeline described in Figure 6 and the compiler generated code. As we can see, the generated code by Skeletor accounts for almost half of the Verilog code contained in the final implementation, which is an important step towards productivity. This is not only because the development time is shortened, but also due to the fact that less code to be written means less bugs (especially due to connectivity issues between modules) which is translated to easier debugging and verification.

V. Future Work

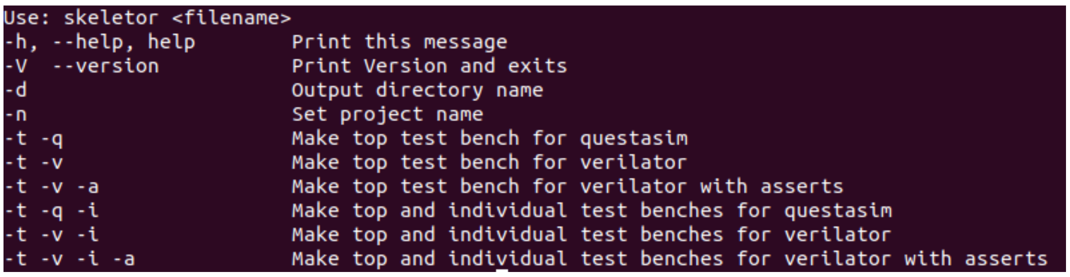
The first version of the tool has been released in our git repository [6] in May 2019, when we considered that the tool supported the minimum set of features to be considered useful. Since then additional features such as support for inline Verilog insertion have been added. Thanks to Flex and Bison, Skeletor is easily extensible. We plan to keep improving its usability and increase its capabilities based on the feedback received by its users. Among the current list of future features we consider as candidates for upcoming releases are: loops, libraries of standard ports such as AXI, Avalon, and wishbone and the ability to instantiate existing Verilog modules.

As a conclusion, even if currently the tool is not entirely polished and it is still under development, it can speed up RTL development. It can be especially useful in several situations such as when several programmers work in the same project, when coding conventions have to be enforced or when new users get started with HDL and require some additional guidance regarding workflow and good coding style.

Currently the tool is being used in both student assignments at the Universitat Politècnica de Catalunya, as well as in research projects at the Barcelona Supercomputing Center (BSC). We encourage anyone interested in the project to check out our repository on [6].

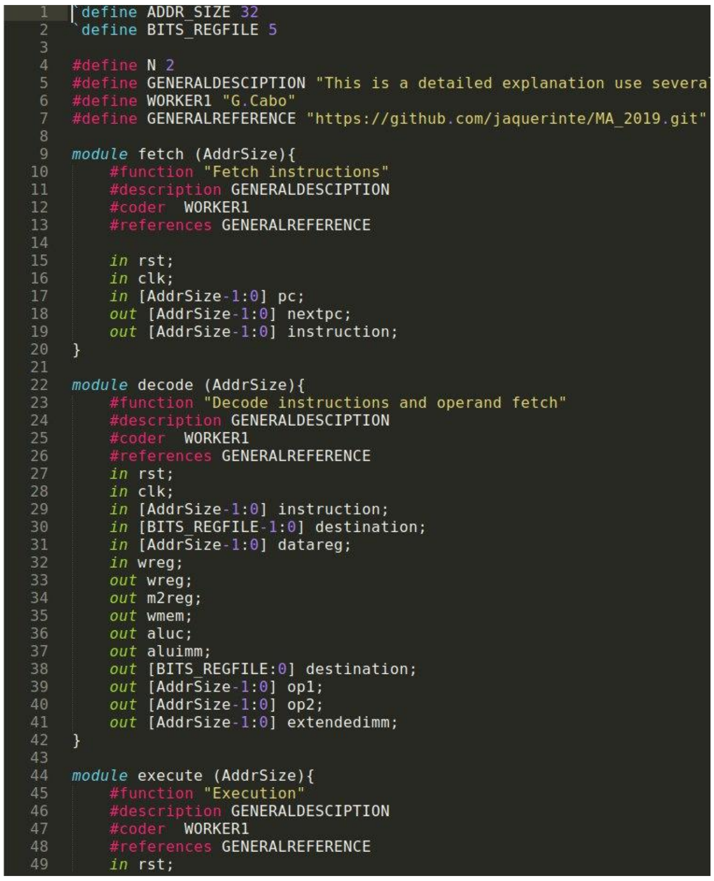
Acknowledgments

This work has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence. Leonidas Kosmidis is also funded by the Spanish Ministry of Economy and Competitiveness (MINECO) under a Juan de la Cierva Formación postdoctoral fellowship (FJCI-2017-34095).



```
Use: skeletor <filename>
-h, --help, help        Print this message
-V  --version           Print Version and exits
-d                      Output directory name
-n                      Set project name
-t -q                   Make top test bench for questasim
-t -v                   Make top test bench for verilator
-t -v -a                Make top test bench for verilator with asserts
-t -q -i                Make top and individual test benches for questasim
-t -v -i                Make top and individual test benches for verilator
-t -v -i -a             Make top and individual test benches for verilator with asserts
```

Figure 5. Command line flags supported by Skeletor.



```
`define ADDR_SIZE 32
`define BITS_REGFILE 5

#define N 2
#define GENERALDESCRIPTION "This is a detailed explanation use severa
#define WORKER1 "G.Cabo"
#define GENERALREFERENCE "https://github.com/jaquerinte/MA_2019.git"

module fetch (AddrSize){
    #function "Fetch instructions"
    #description GENERALDESCRIPTION
    #coder  WORKER1
    #references GENERALREFERENCE

    in rst;
    in clk;
    in [AddrSize-1:0] pc;
    out [AddrSize-1:0] nextpc;
    out [AddrSize-1:0] instruction;
}

module decode (AddrSize){
    #function "Decode instructions and operand fetch"
    #description GENERALDESCRIPTION
    #coder  WORKER1
    #references GENERALREFERENCE
    in rst;
    in clk;
    in [AddrSize-1:0] instruction;
    in [BITS_REGFILE-1:0] destination;
    in [AddrSize-1:0] datareg;
    in wreg;
    out wreg;
    out m2reg;
    out wmem;
    out aluc;
    out aluimm;
    out [BITS_REGFILE:0] destination;
    out [AddrSize-1:0] op1;
    out [AddrSize-1:0] op2;
    out [AddrSize-1:0] extendedimm;
}

module execute (AddrSize){
    #function "Execution"
    #description GENERALDESCRIPTION
    #coder  WORKER1
    #references GENERALREFERENCE
    in rst;
```

Figure 6. Example of syntax highlighting of a Skeletor description file implementing a simple processor pipeline, in the Sublime Text3 editor.

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[2] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, June 2004, pp. 69–70.

[3] V. Paxson, "Flex: The Fast Lexical Analyzer - scanner generator for lexing in C and C++ ." [Online]. Available: https://github.com/westes/flex

[4] R. P. Corbett, "Static semantics and compiler error recovery," Ph.D. dissertation, University of California, Berkeley.

[5] R. P. Corbett, "GNU Bison." [Online]. Available: https://www.gnu.org/software/bison

[6] I. Rodriguez, G. Cabo, J. Giesen, J. Barrera, A. Jover, and L. Kosmidis, "Skeletor," Sep. 2019. [Online]. Available: https://github.com/jaquerinte/Skeletor

[7] W. Snyder, "Verilator and SystemPerl," in *North American SystemC Users' Group, Design Automation Conference (DAC)*, June 2004.

[8] Mentor Graphics, "Questa Advanced Simulator." [Online]. Available: https://www.mentor.com/products/fv/questa

[9] Sublime HQ, "Sublime Text." [Online]. Available: https://www.sublimetext.com

[10] B. Moolenaar, "Vim." [Online]. Available: https://www.vim.org/

[11] R. Stallman, "GNU Emacs." [Online]. Available: https://www.gnu.org/software/emacs/

[12] Eclipse Foundation, "Eclipse." [Online]. Available: https://www.eclipse.org/ide

[13] Microsoft, "Visual Studio." [Online]. Available: https://visualstudio.microsoft.com