Journal of Electrical Engineering, Electronics, Control and Computer Science – JEEECCS, Volume 4, Issue 14, pages 17-22, 2018

CHDL1: Implementing a simplified version of the CompactHDL hardware description language

Florin-Marian Birleanu
Department of Electronics, Computers and Electrical Engineering
Faculty of Electronics, Computer Science and Telecommucations, University of Pitesti
Pitesti, Romania
florin.birleanu@upit.ro

Abstract – **A few years ago an extremely compact hardware description language was proposed. This paper presents the implementation of a subset of that language. For this implementation the JavaCC code generator was used and the resulted application runs on any operating system having Java installed. The application receives the description of the desired logic circuit in the new language and generates the VHDL source files as well as the user constraints file required for implementing the circuit in a FPGA board. The implemented subset of the language allows the user to easily describe any combinatorial logic circuit based on NOT, AND and OR gates. It also makes it very easy to create and use components and to specify pin constraints.**

Keywords – logic design, FPGA, VHDL, CompactHDL, parsing

I. INTRODUCTION

The paper [1] proposed a compact language for describing logic circuts. The people working in this field know very well that the software tools used for configuring FPGAs [2] are based on complex and powerful languages such as VHDL. The new language addressed the fact that VHDL [3] is a very verbose language and that is allows a very wide range of constructs (some of which are not synthesizable [4]) which makes it difficult for beginners. CompactHDL addressed this problems by allowing only a few instructions and a very compact and easy to learn syntax [1].

As we are not aware to this date of any implementation of the CompactHDL language, we present here the results of the first steps in this direction. Our main purpose was to validate the idea of the new language and obtain rapidly a working and useful implementation. With this in mind, we reduced the CompactHDL language proposed in [1] (and we refer to the resulted language using the name CHDL1) and chose to use a compiler compiler software in order to automate as much as possible the implementation of the scanning and parsing parts of the translator [5]. We discuss all these in the following two sections of this paper. We show results and discuss their didactic use in Section IV. In the last section we present the conclusions of the paper and outline a possible continuation of this work.

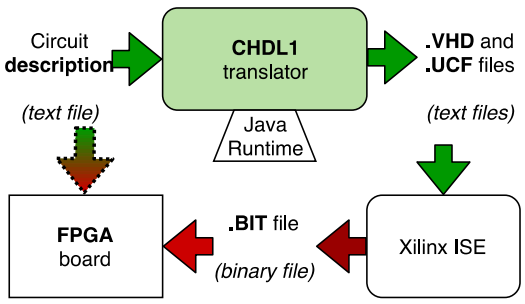
II. LANGUAGES AND TOOLS

We stard by presenting an overview of this work and of the actors involved in it.

A. Overview

Fig. 1 shows the context in which our work is placed.

Figure 1. Overview of the process of configuring a FPGA starting from a CHDL1 description of the logic circuit



The purpose of the language implemented in this paper is to allow the FPGA designer to configure a FPGA board without the need to edit VHDL files, whose syntax is redundant and difficult to memorize. Instead, the designer can use the compact syntax of CHDL1 and obtain the required VHDL files with the aid of the CHDL1 to VHDL translator. This translator is the work that we present in this paper.

In order to benefit from the existing FPGA sotfware tools, the resulted files are then imported into a new Xilinx ISE project and after passing through the necessary steps (synthesis, design implementation, and programming file generation) a binary configuration file is generated. And then this file is downloaded into the FPGA in order to configure it to function according to the original circuit description.

B. The input language

As mentioned before, in establishing the specifications of our input language we started from the description of CompactHDL in [1]. We kept only the basic characteristics required for the description of simple combinatorial circuits and we call the new (sub)language CHDL1.

A CHDL1 source file may contain several module definitions and at most one module instantiation. Each module represents a (sub)circuit and corresponds to an entity-architecture pair in VHDL. The module instantiation may appear only for the top module in the design. Besides specifying which of the defined modules is the top module, this instantiation also specifies how the wires of the top module are associated to actual pins of the FPGA circuit.

A module definition contains the name of the module, the names of the inputs and outputs, and the body of the module. Two types of instructions may appear inside the body of a module: assignment and instantiation. In an assignment the result of an expression is assigned to a wire (corresponding to a VHDL signal). This wire can be one of the outputs of the module or an internal (locally defined) wire. The expression may contain as operands the inputs of the module, (locally defined) wires, the contants 0 and 1, and, as operators, & (AND), | (OR) and ~ (NOT). It can also contain parentheses to group subexpressions.

An instantiation contains the name of the module to be instantiated and the name of the wires where each of the inputs and outputs should be connected. A new wire is defined (implicitly) by its first appearance either in the lefthand side of an assignment or in the wires list of a module instantiation.

The grammar (expressed in an approximate Extended Backus-Naur Form (EBNF – ISO/IEC 14977 standard)) of a CHDL1 source file is shown in Table I.

TABLE I. THE CHDL1 LANGUAGE GRAMMAR

Start → CHDfile <EOF>
CHDfile → ModuleDefinition* (TopModuleInstantiation ModuleDefinition*)?
ModuleDefinition → <NAME> ":" WireList "→" WireList "{" Instruction* "}"
TopModuleInstantiation → "(" <NAME> ")" "(" WireList "->" WireList ")"
ModuleInstantiation → "(" <NAME> ")" "(" WireList2 "->" WireList ")"
WireList → (<NAME> "," <NAME>)*)?
WireList2 → ((<NAME> | <NUMBER>) "," <NAME>)*)?
Instruction → Assignment | ModuleInstantiation
Assignment → <NAME> "=" Expression
Expression → Term ("|" Term)*
Term → UnaryOp ("&" UnaryOp)*
UnaryOp → ("~" Operand) | Operand
Operand → <NUMBER> | <NAME> | ("(" Expression ")")

Besides the anonymous (single form) tokens marked in the grammar with double quotes, the CHDL1 language contains only three tokens: <EOF> (the (implicit) end of file marker), <NAME> (module or wire name), and <NUMBER> (a numerical constant value). Additionally, a CHDL1 source file may contain blanks (spaces, tabs and new lines) and single-line comments (starting with a double slash). They should be skipped in the scanning phase. The different types of tokens that can be present in a CHDL1 file are shown in Table II, together with their regular expressions.

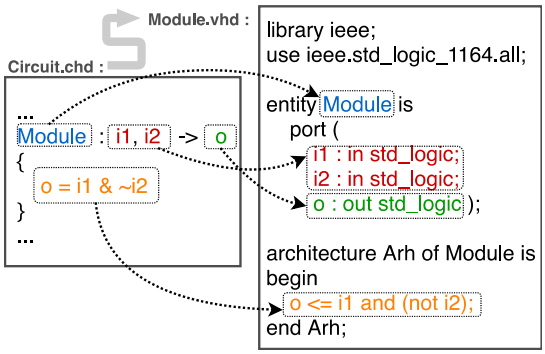
TABLE II. CHDL1 LEXICAL ATOMS

Token type	Name used in the grammar	Regular expression(s)
Blank	(is skipped)	" ", "\t", "\r\n\", "\n" (space, tab, end of line (for Windows, Linux and Mac))
Comment	(is skipped)	<"//" (~["\r", "\n"])*> (Starts with double slash and continues with any string of characters except the end of line markers.)
Separator	anonymous	":", "->", "{", "}", "(", ")", ","
Operator	anonymous	"=", "|", "&", "~"
Number	<NUMBER>	<"0" | "1"> (One of the strings "0" or "1".)
Identifier	<NAME>	<["a"-"z", "A"-"Z"] (("_")? ["a"-"z", "A"-"Z", "0"-"9"])*> (Starts with a letter and continues with letters or numbers (or underscore, but not on the last position).)

C. *The output languages*

Our translator will generate VHDL source files for every module definition found in the CHDL1 file. Each VHDL file will contain an entity and an architecture, as sketched in Fig. 2.

Figure 2. The correspondence between a module definition in CHDL1 and a VHDL entity-architecture pair



If the top module instantiation is present in the source file, our translator will also generate the corresponding UCF (user constraints file) file containing FPGA pin associations for the top module. For instance, the following top module instantiation:

(Module) (G18, H18 -> J14)

would be translated into:

NET "i1" LOC = "G18";
NET "i2" LOC = "H18";
NET "o" LOC = "J14";

where the module named Module is the one defined in Fig. 2.

D. The software tools

For "compiling" the generated VHDL and UCF files into a bitstream (BIT) configuration file we will use Xilinx ISE 9.1i (running on Linux (lubuntu 14.01)). The resulted configuration file will be transferred to the test board by running (from the command line, in Linux) the Adept2 application. (The test board is a Nexys 2, containing a Spartan 3E XC3S500E FPGA circuit.)

On the other hand, for obtaining the CHDL1 translator we will use the JavaCC code generator [6]. JavaCC is a (command line) software tool that generates Java source code. Based on the specification of the desired language tokens and grammar, it generates Java source code for scanning and parsing the input according to the specified rules. Additional Java code can instruct the generated parser to produce an abstract syntax tree, which can be then traversed in the main program to generate the desired output files.

The resulted Java source file will be compiled with the javac compiler (from Java Development Kit (JDK) 7) and the resulted main class file will be run with java (the Java Virtual Machine (JVM) from the Java Runtime Environment (JRE)).

III. TRANSLATOR DESIGN AND IMPLEMENTATION

A (formal) language translator is usually composed of three parts: the scanner, the parser and the code generator [1, 5]. The first two parts are similar for most languages and can be easily automated with a compiler compiler such as JavaCC. Of course, there is some design effort involved here as well – the effort to design efficient regular expressions for the tokens of the desired input language and a proper context-free grammar for it. (Actually, JavaCC requires a grammar that is LL(k) [6], which is more restrictive than a context-free grammar.) The part that can not be automated (and requires more effort and good programming skills) is the design of the abstract syntax tree and the programming of the code generator.

JavaCC allows us to input all these in a single text file (i.e., a JavaCC specifications file, having a ".jj" extension). The construction of this file will result from combining the three parts of the translator, which are discussed next.

A. The scanner

Scanning is the stage where symbols that are redundant or have no meaning (such as blanks and comments) are removed, while the rest of the symbols from the input source file are grouped into meaningful tokens (such as identifiers, numbers, operators and separators).

Otherwise said, the scanner understands the microsyntax of the input language. We can specify this microsyntax with the aid of the JavaCC keywords SKIP and TOKEN, together with the regular expressions from Table II:

```
SKIP: {" " | "\t" | "\r\n\" | "\n"}
SKIP: {<"//" (~["\r", "\n"])*>}
TOKEN: {":" | "->" | "{" | "}" | "(" | ")" | ","}
TOKEN: {"=" | "|" | "&" | "~"}
TOKEN: {<NUMBER: "0" | "1">}
TOKEN: {<NAME: ["a"-"z", "A"-"Z"] (("_")? ["a"-"z",
      "A"-"Z", "0"-"9"])*>}
```

The scanner skips the specified patterns and outputs a series of token objects (containing an identifier for the token type and the actual value of the found token) that are fed automatically to the parser.

B. The parser

Parsing is the core of language translation. At this stage the series of tokens found by the scanner is checked for syntactical corectness and the relevant tokens are kept and organized into an abstract syntax tree (AST).

In other words, the parser "understands" the macrosyntax of the input language. In JavaCC this macrosyntax is specified in a manner that is very similar to the ENBF grammar specification of the language (see Table I). For instance, the first three variables of the grammar would be specified as follows:

```
void Start() : {}
{     CHDfile() <EOF> }
void CHDfile() : {}
{     ( ModuleDefinition() )*
      ( TopModuleInstantiation() ( ModuleDefinition() )* )?
}
void ModuleDefinition() : {}
{     <NAME> ":" WireList() "->" WireList()
      "{" ( Instruction() )* "}"
}
```

However, such a parser (to be invoked in the main program by calling the method Start) would only be useful to check that the output of the scanner does not contain any syntax errors. (Otherwise, exceptions are thrown.)

We can modify this parser specification such that it also generates an AST.

C. The abstract syntax tree

In order to design the structure of the AST we look at the grammar variables (turned into parser methods) and decide what kind of data each of it should return (to make sure we retain from the series of scanned tokens all information that would be relevant for the code generator). Based on this we can construct a class inheritance hierarchy for the nodes of the AST. The results of this design effort are summarized in Table III.

TABLE III. AST NODE DATA FOR OUR CHDL1 TRANSLATOR

Grammar variable	**Returns object of class**	**Node data**	**Node parent class**
Start	NodeStart	type=ANY (int) file (Node)	Node
CHDfile	NodeCHDfile	type=ANY (int) declarations (List<Node>)	Node

ModuleDefinition	NodeDefiniton	type=DEFINITION (int) name (String) inputs (List<String>) outputs (List<String>) body (List<Node>)	Node
TopModuleInstantiation	NodeInstantiationTop	type=INSTANTIATIONTOP (int) name (String) inputs (List<String>) outputs (List<String>)	Node
ModuleInstantiation	NodeInstantiation	type=INSTANTIATIONTOP (int) name (String) inputs (List<String>) outputs (List<String>)	Node
WireList	List<String>		
WireList2	List<String>		
Instruction	NodeAssignment or NodeInstantiation		Node
Assignment	NodeAssignment	type=ASSIGNMENT (int) left (String) right (NodeExpression)	Node
Expression	NodeBinOp	op (String) expr1 (NodeExpression) expr2 (NodeExpression)	NodeExpression
Term	NodeBinOp	op (String) expr1 (NodeExpression) expr2 (NodeExpression)	NodeExpression
UnaryOp	NodeUnOp	op (String) expr (NodeExpression)	NodeExpression
Operand	NodeOperand	opd (String)	NodeExpression

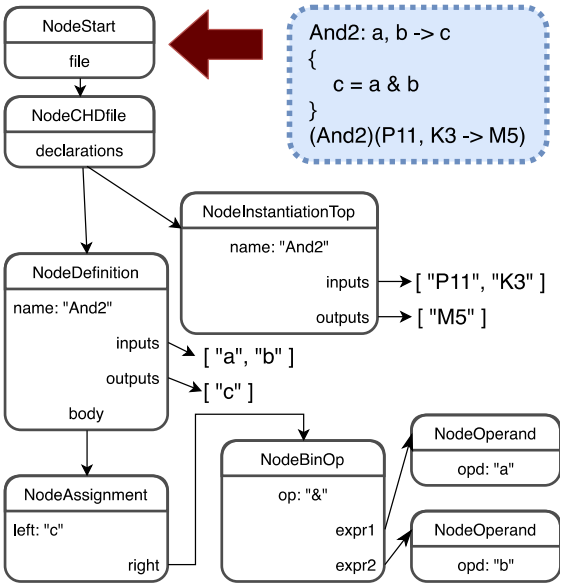
After defining the AST node classes (in the PARSER section of the JavaCC specifications file) we can go back to the parser and make the necessary changes:

```
Node Start() : { NodeStart nodeS; Node node; }
{       { nodeS = new NodeStart(); }
        node=CHDfile() <EOF>
        { nodeS.file = node; return nodeS; }
}
Node CHDfile() : { NodeCHDfile nodeF; Node node; }
{       { nodeF = new NodeCHDfile(); }
        ( node=ModuleDefinition()
        { nodeF.declarations.add(node); } )*
        ( node=TopModuleInstantiation()
        { nodeF.declarations.add(node); }
        ( node=ModuleDefinition()
        { nodeF.declarations.add(node); } )* )?
        { return nodeF; }
}
```

```
Node ModuleDefinition() : { NodeDefinition nodeD;
        Token tok; String name; List<String> inputs;
        List<String> outputs; Node node; }
{       { nodeD = new NodeDefinition(); }
        tok=<NAME> ":" inputs=WireList() "->"
        outputs=WireList()
        "{" ( node=Instruction()
        { nodeD.body.add(node); } )* "}"
        { nodeD.name = tok.image; nodeD.inputs = inputs;
        nodeD.outputs = outputs; return nodeD; }
}
```

The call of this new Start method return the AST that we can traverse in order to generate the desired output files (in ".vhd" and ".ucf" format). An example of a simple CHDL1 source file and its corresponding AST is shown in Fig. 3.

Figure 3. The abstract syntax tree for the CHDL1 description of an AND gate



D. The code generator

We call the "Start" method in the "main" method of our parser class (defined between PARSER_BEGIN(…) and PARSER_END(…) in the ".jj" file). Its output is a reference to the root of the abstract syntax tree.

Starting from the root of the AST we access the list of declarations from the source file. For each such declaration that is of type module definition we create a new VHDL source file. If a top module instantiation is present, we create for it a ".ucf" file. We check for redefinition of modules by inserting each module definition into a hash map. Then, we create a new ".vhd" file for the module and we fill it with the definition for the corresponding module entity in VHDL. Each input or output is a single-bit std_logic value.

In order to also generate the VHDL architecture for the module, we create a hash map into which we

insert all the wires encountered in the module definition (starting with the inputs and the outputs). A new wire is added if a new wire name is encountered on the left side of an assignment or in the wires list of an instantiation. Then, for each of these new wires (that are neither inputs nor outputs of the module) we add a signal declaration in the architecture of the VHDL module. For constructing the body of the architecture in the VHDL file, we replace each instruction found (in the body of the CHDL1 module) with the corresponding VHDL construct. For assignment instructions the translation to VHDL is straightforward (as we only need to replace the assignment operator and the CHDL1 logic operators with the corresponding operators in VHDL). Instantiations, on the other hand, require a little more attention (in order to generate a unique label for each one and to properly generate the port map of the instantiated entity).

It should be mentioned that we did not check whether the module being instantiated in the CHDL1 source was also defined there. (Hence, we did not check that the numbers of inputs and outputs in the instantiation correspond with the numbers of inputs and outputs from the definition.) We omitted to do that in order to facilitate the splitting of the CHDL1 description into multiple source file. This facilitates also the interoperability between CHDL1 and VHDL, by allowing the designer to use inside a CHDL1 source file a module defined in a VHDL source file.

As we said earlier, if a top module instantiation is present in the CHDL1 source file, we must generate a ".ucf" file for assinging module pins to actual FPGA pins. In this case, the module definition must be found in the CHDL1 source file above this instantiation (or otherwise an error will be generated). Of course, the numbers of inputs and outputs in this instantiation must be equal to the numbers of inputs and outputs in the module definition. A "NET … LOC … ;" line is added in the ".ucf" file for each of these inputs and outputs.

IV. RESULTS

A. *Compilation and use*

The parts presented in the previous section were put together into a JavaCC specification file (that we called "CompactHDL.jj" with the following structure:

```
PARSER_BEGIN(CHDL)
public class CHDL { // The main class of the parser
        public static void main (String args[]) { // Main program
                … // Parser instantiation
                … // Call of Start() method => the AST root
                … // AST traversal and code generation
        }
}
PARSER_END(CHDL)
… // Scanner specifications (Language tokens)
… // Parser specifications (Language syntax)
```

The resulted file was compiled (in the command terminal of the operating system) with javacc in order to generate the missing Java code for the scanner and the parser:

```
>> javacc CompactHDL.jj
```

(We used version 5.0 of JavaCC and the "bin" folder was added to the system PATH variable.)

The resulted ".java" file were then compiled with javac (from Java Development Kit (JDK) version 7):

```
>> javac *.java
```

This generated in the current folder ".class" files for the classes in the application. The main class of the application was then run with the Java Virtual Machine:

```
>> java CHDL <circuit.chd
```

This assumed that the digital circuit description in the CHDL1 language was present in the "circuit.chd" text file located in the current folder. As a result of running this command, ".vhd" files were generated for every module definition found in the ".chd" file and an ".ucf" file was generated for the instantiation of the top module.

B. *Example of use*

As an example, let us see how we could use our CHDL1 translator for the implementation of a digital circuit with three inputs that adds the constant "1" to the binary number from its input. its implementation based on a demultiplexer is shown in Fig. 4.

Figure 4. The schematic of the "y =x+1" circuit using a 1:8 DeMUX

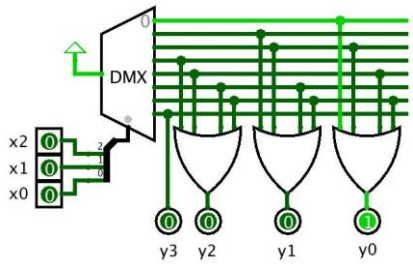
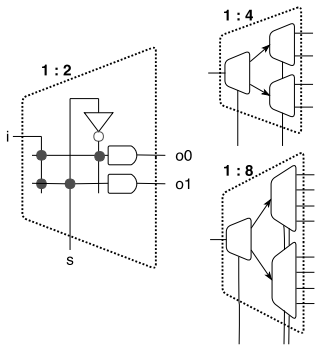


Figure 5. Implementation of demultiplexers – 1:2, 1:4 and 1:8



In order to make things more interesting, the 1:8 demultiplexer can be constructed from 1:2 demultiplexers, as shown in Fig. 5. Despite the complexity of the resulting circuit schematic, its description in CHDL1 is very simple:

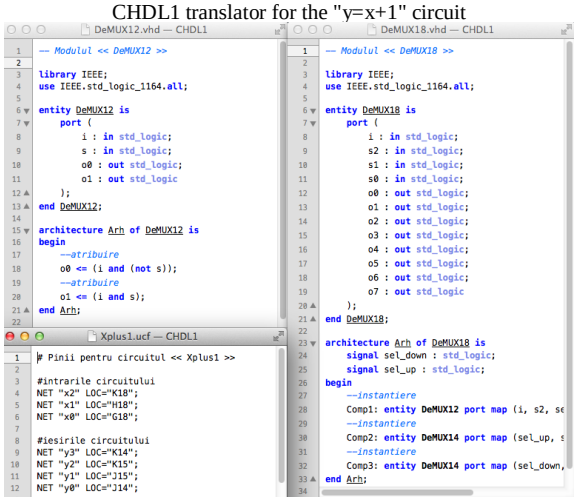
```
// circuit.chd:
Xplus1: x2, x1, x0 -> y3, y2, y1, y0 {
```

```
  (DeMUX18) (1, x2, x1, x0 -> o0, o1, o2, o3, o4, o5, o6, o7)
  y3 = o7
  y2 = o3 | o4 | o5 | o6
  y1 = o1 | o2 | o5 | o6
  y0 = o0 | o2 | o4 | o6 }
DeMUX18: i, s2, s1, s0 -> o0, o1, o2, o3, o4, o5, o6, o7, o8 {
  (DeMUX12) (i, s2 -> sel_up, sel_down)
  (DeMUX14) (sel_up, s1, s0 -> o0, o1, o2, o3)
  (DeMUX14) (sel_down, s1, s0 -> o4, o5, o6, o7) }
DeMUX14: i, s1, s0 -> o0, o1, o2, o3 {
  (DeMUX12) (i, s1 -> sel_up, sel_down)
  (DeMUX12) (sel_up, s0 -> o0, o1)
  (DeMUX12) (sel_down, s0 -> o2, o3) }
DeMUX12: i, s -> o0, o1 {
  o0 = i & ~s
  o1 = i & s }
```

By running into the terminal the command "java CHDL <circuit.chd", the following files were generated: Xplus1.vhd, DeMUX18.vhd, DeMUX14.vhd, DeMUX12.vhd and Xplus1.ucf. The contents of three of these files is shown in Fig. 6.

Afterwards, we opened Xilinx ISE 9.1i and created a new project. We selected the FPGA circuit from the Nexys 2 board (Spartan3E xc3s500e fg456), added the generated files to the project and run the "Generate programming file" command. (Then, we programmed the resulted "Xplus1.bit" file into the Nexys 2 board by running the Adept application.)

Figure 6. The contents of three of the files generated by the CHDL1 translator for the "y=x+1" circuit



C. Discussion

It is not difficult to notice how much the CHDL1 language simplifies the description of basic combinatorial logic circuits compared to VHDL. The CHDL1 source file is a minimalistic direct description (in text form) of the circuit schematic. And it is very important for novice HDL (hardware description language) users to see the HDL source code as a "description of a schematic" , not as a "program". This makes our translator a handy tool for teaching the basics of logic circuit design using hardware description languages.

Our translator is easy to use and runs on virtually any operating system (with Java installed). It is a console application (without a graphical user interface), but it is easy to automate its use through scripting in the operating system terminal. Also, it is easy to integrate with a generic source code editor that has customizable menu commands.

Besides the didactic use of the resulted application for teaching the fundamentals of logic circuit design and hardware description languages, our implementatin of the CHDL1 translator might be a useful case study for Java programmers who want to implement a translator for a custom language with the aid of the JavaCC tool.

Acknowledgment

The author would like to thank his former student Georgiana-Cosmina Ghita for her help during the implementation of the translator presented in this paper.

Conclusion

Logic circuit design is the foundation on which digital computing devices such as CPUs and GPUs were built. FPGAs play a major role in this field, as they allow rapid prototyping of digital systems based on a textual description of the circuit schematic. However, the languages used for this description are not very friendly for beginners. This is why we implemented here a very simple language for describing combinatorial logic circuits using the three basic logic gates ("and", "or" and "not"). The language also enables the designer to easily describe circuits made of subcircuits. With the aid of Java CC we implemented a translator from this language (which we called CHDL1) to the widely used VHDL language. The resulted application is useful as a tool for teaching the basics of logic design and hardware description languages. Its design can also serve as a case study that highlights and briefly explains and illustrates the main steps required for implementing a generic translator.

The work presented here could be further developed by extending the CHDL1 language such that it accepts the use of buses and parameters. This would allow us to describe generic (variable-sized) multiplexers, decoders, adders and other combinatorial logic circuits.

References

[1] F.M. Birleanu, B.A. Enache, M. Alexandru, “First steps towards designing a compact language for the description of logic circuits,” Proceedings of the International Conference on Communications (COMM), 9-10 June 2016.

[2] G. R. Smith, FPGAs 101: Everything You Need to Know to Get Started. Elesevier (Newnes), 2010.

[3] B. J. LaMeres, Introduction to Logic Circuits & Logic Design with VHDL. Springer International Publishing, 2017.

[4] E. Bezerra, D.V. Lettnin, Synthesizable VHDL Design for FPGAs. Springer International Publishing, 2014.

[5] S.C. Reghizzi, L. Breveglieri, A. Morzenti, Formal languages and compilation. Springer-Verlag London, 2013.

[6] T. Copeland, Generating Parsers with JavaCC: An Easy-to-Use Guide for Developers. Alexandria, VA: Centennial Books, 2013.