DESIGN AND IMPLEMENTATION OF A TRANSLATOR FOR CONVERTING A MINIMALIST LANGUAGE TO VHDL

Mihai-Ionuț VÎLCU[1], Florin-Marian BÎRLEANU[2]
University of Pitești, Faculty of Electronics, Communications and Computers
[1]mihai.vilcu93@gmail.com, [2]florinmarianb@yahoo.com

Keywords: logic circuit design, high-level synthesis, hardware description languages, web applications

Abstract: *This paper presents the design and implementation of a software application that performs the conversion of an algorithm, written in a high-level language, into a VHDL description of the logic circuits required for the hardware implementation of that algorithm. The minimalist language allows the user to define variables, to test conditions and to implement loops and simple mathematical expressions. The entire application was implemented by using the JavaScript language and runs in a web browser.*

1. INTRODUCTION

FPGA circuits [1] are a very flexible solution for implementing digital circuits that are compact and fast. They have the advantage of not requiring physical connections between components, as these connections are made by writing the contents of a binary file into a configuration memory. The drawback, however, is that constructing the configuration file is not an easy task.

Despite the fact that it is commonly named programming, the configuration of FPGA circuits can not be performed by a software programmer. That is because the FPGA circuit is not a microprocessor that allows the user to load a program for it to run, but is a collection of basic logic circuits that can be interconnected. The configuration file for an FPGA is not made of a series of instructions, but it is made of a series of connections. This file is not the result of compiling a program; it is the result of describing a circuit.

The most flexible method for describing digital circuits is a text description by using the VHDL hardware description language [2]. Despite its similarity with a programming language, VHDL can not be used for writing programs, but for describing circuits. It is true that, given a certain software algorithm, an FPGA expert can design a circuit that can perform the same function as the algorithm (either without any microcontroller involved, or with an ad hoc design of a custom minimalist microcontroller). However, this is not an easy task; and by no means can it be performed by a software programmer with very little hardware design experience.

A solution for allowing the (almost) transparent configuration of FPGA circuits by sofware programmers is high-level synthesis [3]. It consists in writing the desired algorithm in a high-level programming language and then using a software translator that is able to translate the algorithm into a sequential logic circuit that will perform the same function. Such translators are discussed in papers [4] and [5].

This paper presents the design and implementation of a software translator used for processing and transforming a minimal langauge (similar to the language described in [4]) into a VHDL description. The basic principle of this process is a lexico-syntactic analysis of the input source code which transforms this code into a finite automaton that is afterwards translated into VHDL code.

The rest of the paper is organized as follows. The graphical user interface of the application is described in Section 2. The source language of the translator is presented in Section 3. Section 4 describes the structure of the translator and its components. Section 5 shows results obtained by running the application. Conclusions are presented in Section 6.

2. GRAPHICAL USER INTERFACE AND FUNCTIONS

Nowadays web interfaces are more and more used for developing applications in many fields. These interfaces are built with the aid of technologies such as HTML5, CSS3 and JavaScript, which are constantly being developed.

The graphical user interface of our application is split into four sections (illustrated in Fig. 1) that offer an overview of the process of translating the source code to VHDL. The first section (located on the left side of the screen) contains the source code (written in our minimalist programming language) that is input by the user.

The second section (located on the bottom of the screen) contains a list of the tokens indentified (in real time) from the source code.

The third section (located on the right side of the screen) shows the VHDL code generated (in real time) by the translator.

The fourth section (located on the top-right of the screen) is a menu that allows the user to study code samples, as well as to view the syntax tree (corresponding to the source code) and the graph (generated from the syntax tree).

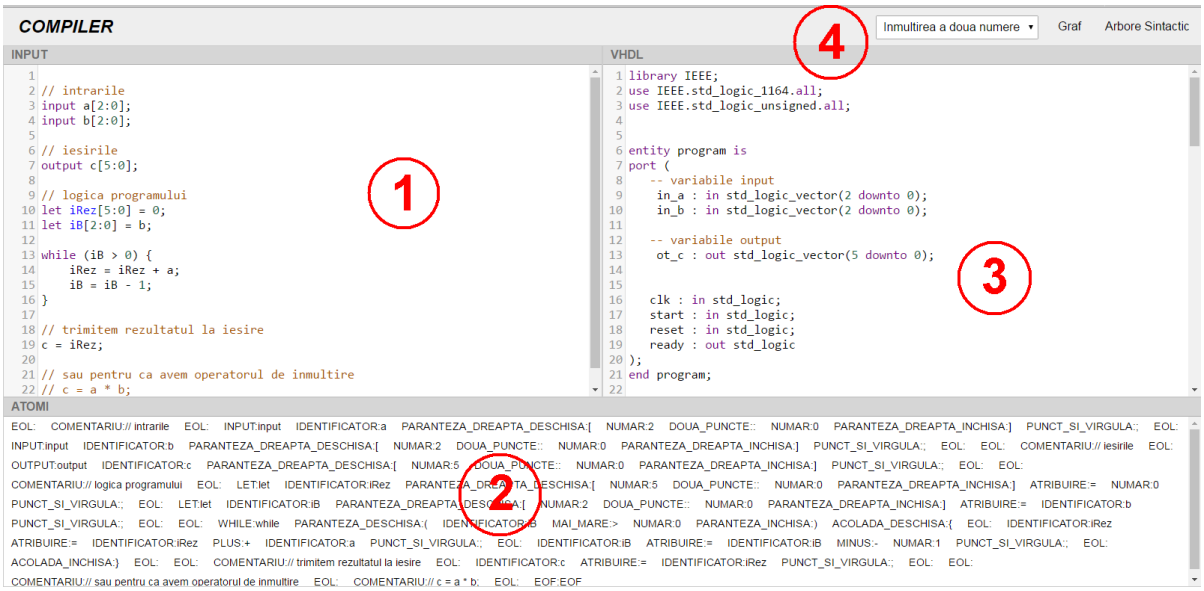

Fig. 1 The graphical user interface of the application and its four components: (1) input, (2) tokens, (3) output, (4) menu.

3. SOURCE LANGUAGE

3.1. Input/output ports

A program written in our minimalist langauge must begin with the declaration of the input/output ports. They can have 1-bit values (of std_logic type, in VHDL) or multi-bit values (of std_logic_vector type, in VHDL). When declaring a multi-bit port, its name must be succeeded by its size written inside square brackets. Ports can be declared only at the beginning of the program. The attempt to declare a port in the middle of the program generates a syntax error.

A port is declared with the **input** (for an input port) or **output** (for an output port) keywords, followed by the name of the port (and, possibly, by its size). Additionaly, the declarations of multiple ports of the same type can be combined into a single declaration that uses the **input**/**otput** keyword followed by the names (and optional dimensions written inside square brackets), separated by commas. The following code shows some examples:

```
// declaraing ports
input c, d[1:0];
output rez, data[2:0];
```

3.2. Variable declarations

Variables can be, as well as ports, single-bit or multi-bit. In order to declare a multi-bit

variable, its name must be followed by its size written inside square brackets.

A variable is declared by using the **let** keyword followed by the name of the variable, its size (optionally) and its initialization with a value (optionally). As well as in the case of port declarations, several variable declarations can be combined into a single instruction, as the following code fragment shows:

```
// declaring variables
let a = 0, b, c[2:0];
```

3.3. Decisions

Conditional instructions allow the execution of a certain code sequence based on the truth value of a condition. In the minimalist language presented in this paper, decisions are made with the **if** keyword followed by a pair of parentheses containing the condition to be tested and by a pair of curly braces containing the set of instructions to be executed if the condition is true and (optionally) by the **else** keyword and by the set of instructions (inside curly braces) to be executed if the condition is false. If the set of instructions contains a single instruction, the curly braces can be omitted. Some examples of use are presented in the following code fragment:

```
// if-else instruction
if (a > 2)
{
    i = i + 1;
}
else
{
    j = j + 1;
    k = k + 1;
}
// if-else with single instructions
if (a > 2)
    i = i + 1;
else
    j = j + 1;
```

3.4. Loops

Loops allow the execution of a certain code sequence repeatedly while the tested condition is true. In our minimalist language loops are written as the while keyword followed by the condition to be tested (placed inside a pair of parentheses) and by the code sequence to be repeated (placed inside a pair of curly braces). If the code sequence contains a single instruction, the curly braces can be omitted. Inside loops forced continuation (**continue**) of interruption (**break**) instruction can be used. The following code fragment shows some examples:

```
// while
while (j >= 1)
{
    j = j - 1;
}
// while with a single intruction
while (i >= 1)
    i = i - 1;
// while with break and continue
while(2 > 1)
{
    if (i == 0) {
        break;
    }
    i = i + 1;
    if (i == 9) {
        continue;
    }
    i = i * 5;
}
```

3.5. Assignments

An assignment instruction is composed of a variable name followed by the assignments operator and by a mathematical expression followed by a semicolon. The role of such an instruction is to assign the computed value of the expression to the variable. Some examples are presented here:

```
// assignments
i = a[0] + b - 1;
// unary operators
d = !b;
d = -b;
// binary operators
d = a[0] * b / (4 + 5 - 2);
// concatenation operator
c[2:0] = a[2:1] & b;
```

4. APPLICATION DEVELOPMENT

The process of source code translation from the minimalist language to VHDL consists of a series of successive transforms that are performed by the four components shown in Fig. 2 – Lexer, Parser, Grapher and Texter. The role

of the Lexer block is to process the text received as input in order to generate tokens for each lexical atom recognized. The Parser has the role to process the series of tokens received from the Lexer in order to generate the abstract syntax tree.

The Grapher processes the syntax tree received from the Parser in order to generate an oriented graph that represents the logic of the algorithm described with the minimalist language. The Texter generates source code in the target language (VHDL) based on the graph received from the Grapher.

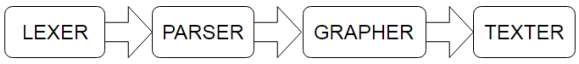

Fig. 2 The block diagram of the translator.

4.1. Lexical analysis

The Lexer processes the source code (written in the minimalist language) one character at a time. When it identifies a lexical atom it outputs it and then it resumes the process [6]. The output is, hence, a list of tokens.

A token is a data structure that contains information about the type of the lexical atom, its actual value and its position in the initial text.

The identification of the lexical atoms is performed by the **getToken** method, which recognizes and returns the next token found in the source text. This is performed by analysing the current character and the next one.

The algorithm used by the getToken method for processing the source code and returning the next token is as follows:

1. Jump over whitespaces
2. If reached the end of the source code, return **EOF**
3. Get current character
4. If beginning of a comment, process the contents of the comment and build a token of type **COMMENT**; return the token
5. If current character and the next one form a special operator (e.g.: ==, != etc.), build a token of the corresponding type; return the token
6. If current character is a simple operator (=, <,> etc.), build a token of the corresponding type; return the token
7. If current character combined with the next one is the beginning of a hexadecimal number, process the contents of the number, build a token of type **NUMBER_HEX**; return the token
8. If current character is a number, process the contents of the number, build a token of type **NUMBER**; return the token
9. If current character is the beginning of an identifier, process characters until the end of the identifier
   a. If the obtained identifier is a keyword, build a token of the corresponding type; return the token
   b. Otherwise, build a token of type **IDENTIFIER**; return the token
10. If current character is a newline, build a token of type **EOL**; return the token
11. And go to the next character
12. Otherwise, if no valid token found, build a token of type **NOTKNOWN**; return the token

4.2. Syntax analysis

The Parser processes the tokens generated by the Lexer and generates the abstract syntax tree based on a context-free grammar. During this process the syntactic corectness of the source code is also performed. Table 1 describes the grammar [7] used for the minimalist language implemented in this paper.

Table 1 The context-free grammar of our minimalist language

<Program>	→	ε | <Instructions>
<Instructions>	→	<Instruction> <Instruction>*
<Instruction>	→	(ε | <Port> | <Assignment> | <Decision> | <Repetition>) ”;”
<Assignment>	→	(ε | ”let”) <Variable> ”=” <Expression> (”,” <Variable> ”=” <Expression>)*
<Variable>	→	<Identifier> (”[”<Number>”:”<Number> ”]” | ”[”<Number>”]” | ε)
<Identifier>	→	<Letter> (<Letter> | <Digit>)*
<Letter>	→	a | b … | z | A | B … | Z | _

<Digit>	→	0 | 1 | 2 … | 9
<Number>	→	<Digit> <Digit>*
<Expression>	→	<Term> (<Operator> <Term>)*
<Term>	→	<Op_unary> <Number> | <Number_hex> | <Variable> | ”(” <Expression> ”)”
<Op_unary>	→	”-” | ”!”
<Number_hex>	→	”0x” <Digit><Digit>*
<Operator>	→	”*” | ”/” | ”+” | ”-” | ”&” | ”>” | ”>=” | ”<” | ”<=” | ”==” | ”=” | ”&&” | ”||”
<Decision>	→	”if” ”(” <Expression> ”)” <Instr_block> (”else” <Instr_block>)*
<Instr_block>	→	<Instruction> | ”{” <Instructions> ”}”
<Repetition>	→	”while” ”(” <Expression> ”)” <Instr_block>
<Port>	→	(”input” | ”output”) <Variable> (, <Variable>)*

The abstract syntax tree is generated by using a recursive descent parser [6] for parsing instructions and the Shunting-yard [8] algorithm for parsing expressions.

During the generation of the syntax tree, the Parser also checks that all the instructions conform to the grammar shown in Table 1.

If the syntactic structure of the code is not valid, the Parser block throws an error that contains information about the cause of the syntax error and its location in the source code.

Besides checking the syntactic structure, the Parser also performs additional checks in order to avoid common errors.

4.3. Graph generation

The Grapher has the role to process the abstract syntax tree generated by the Parser and to produce an oriented graph representing the logic of the source program.

The graph starts with a node of type NODE_START. From this node goes out a transition on the condition "(start == 1)", which allows the start of graph processing. The start node is always named s0 (state number 0). This node is followed by other types of nodes (NODE_EXPRESSION, NODE_IF, NODE_WHILE), used for representing expressions, decisions and loops.

4.4. VHDL code generation

The target code generator (i.e. the Texter) is composed of a series of static methods that are used for processing the finite state automaton generated by the graph generator block.

The main method of the code generator is **getVHDL**. It receives as parameters the previously generated graph, the data regarding the input/output ports, as well as the variables declared in the source program. By using these data, it generates the final program with the aid of a set of templates for the various parts of the program.

The use of templates allows the separation of the graph processing logic from the logic that generates the final text. A sample template looks like this:

```
when "{{state}}" =>
  {{expression}}
  next_s <= "{{futureState}}";
```

4.5. Unit testing

During the development of a software application, the source code evolves due to changes of the specifications, redesign or refactorization. All of these modifications can introduce different problems in the operation of the application, problems that are not always obvious during manual testing. Automated testing has the role to discover this type of problems and to guarantee the correct operation of the modified program.

Unit testing verifies each program unit separately. A unit may be a class, a method or a sequence of method calls.

Tests were implemented using the mocha library (from mochajs.org), which offers the possibility to test the JavaScript code on the server as well as in the browser. It comes with a set of tools that allow the rapid creation and execution of the desired tests. In order to run the written tests, a platform that is able to interpret and execute the JavaScript code is required. In this paper the NodeJS platform was used for running tests.

5. RESULTS

For testing the corectness of the VHDL code generated by the translator, a circuit that performs the multiplication of two 3-bit numbers was built. The circuit has two inputs (named a and b) of 3 bits, and an output (named rezultat) of 6 bits. Therefore, in the program the input ports a[2:0] and b[2:0] and the output port rezultat[5:0] must be defined. The translator automatically adds the inputs reset, clk and start, as well as the output ready.

Mathematically, the multiplication of two numbers (a x b) is a repeated summing of b with itself a times. Starting from this idea, we can write (by using the minimalist language described in this paper) the following algorithm for computing the result of multiplying the two values applied on the circuit inputs:

```
// the inputs of the circuit
input a[2:0];
input b[2:0];
// the output of the circuit
output rezultat[5:0];
// the program logic
let iRez[5:0] = 0;
let iA[2:0] = a;
while (iA > 0) {
    iRez = iRez + b;
    iA = iA - 1;
}
// send computed result to the output
result = iRez;
```

The VHDL code generated by our application for the above program looks like this:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity program is
port (
   -- input variables
    in_a : in std_logic_vector(2 downto
0);
    in_b : in std_logic_vector(2 downto
0);
   -- output variables
   ot_rezultat: out std_logic_vector(5
downto 0);
   clk : in std_logic;
   start : in std_logic;
   reset : in std_logic;
   ready : out std_logic
);
end program;
architecture program_a of program is
  -- signals used for current and next
states
  signal current_s, next_s :
std_logic_vector(3
  downto 0);
begin
process (clk, reset)
begin
    if (reset = '1') then
        current_s <= "0000";
    elsif (rising_edge(clk)) then
        current_s <= next_s;
    end if;
end process;
process (current_s, start)
  -- internal variables
  variable var_iRez:std_logic_vector(5
downto 0);
  variable var_iA:std_logic_vector(2
downto 0);
begin
  case current_s is
    when "0000" =>
        ready <= '0';
        if (start = '1') then
           next_s <= "0001";
        end if;
    when "0001" =>
        var_iRez(5 downto 0) :=
"000000";
        next_s <= "0010";
    when "0010" =>
        var_iA(2 downto 0) := in_a;
        next_s <= "0011";
    when "0011" =>
        if (var_iA > 0) then
            next_s <= "0100";
        else
            next_s <= "0111";
        end if;
    when "0100" =>
        var_iRez := var_iRez + in_b;
        next_s <= "0101";
    when "0101" =>
        var_iA := var_iA - "001";
        next_s <= "0110";
    when "0110" =>
        next_s <= "0011";
    when "0111" =>
        ot_rezultat <= var_iRez;
        next_s <= "1000";
    when "1000" =>
        -- final state
        ready <= '1';
```

```
        next_s <= "0000";
    when others =>
        next_s <= "0000";
  end case;
end process;
end program_a;
```

In order to verify the corectness of this generated VHDL code, a simulation was performed with ActiveHDL Student Edition. The results are shown in Fig. 3. The start of the simulation is on the top, while the end of the simulation is shown on the bottom of the figure.

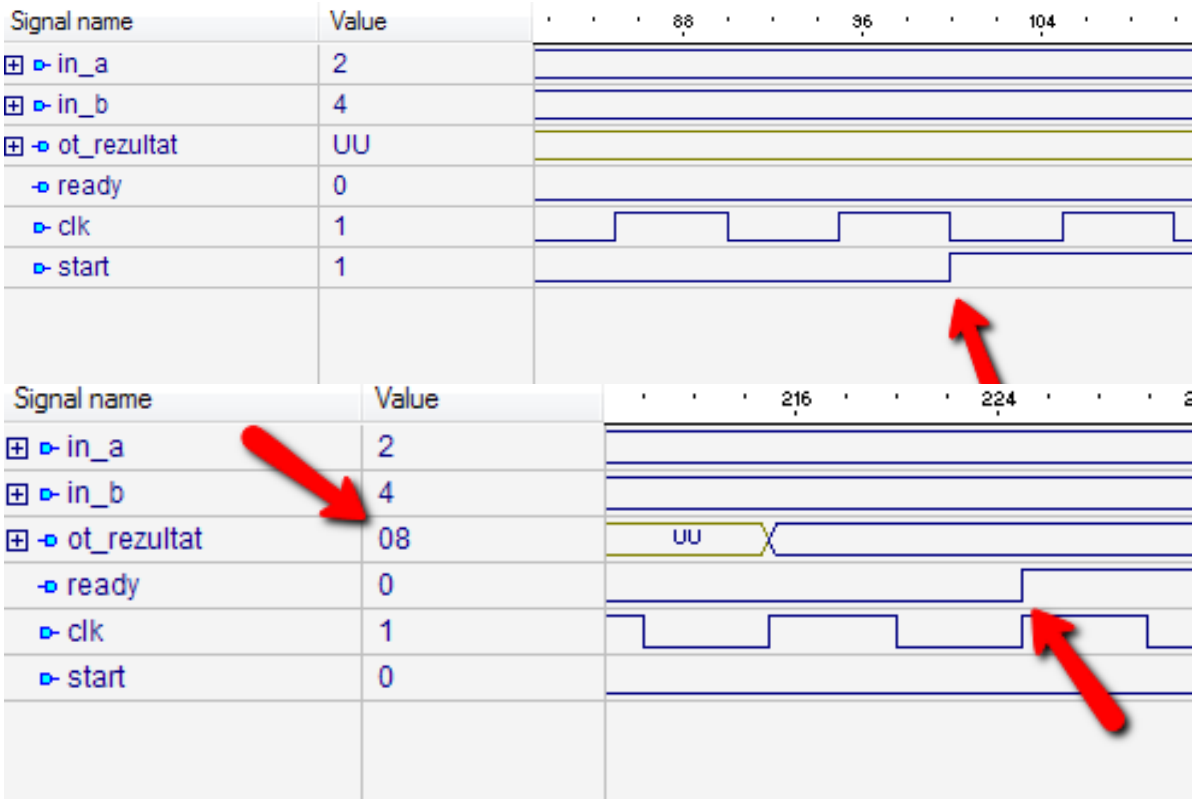

Fig. 3 The results of simulating the VHDL code generated for the multiplication circuit.

6. CONCLUSION

The paper presented a software application that allows the description of logic circuits by using an imperative structured programming language. This is done by converting the source code written in this minimalist programming language (that allows instructions for declaring variables and for performing assignments, decisions and loops) into a circuit description in the VHDL hardware description language.

The method that was used for generating the VHDL code is simple and didactic and it helps to understand the relationship between software and hardware. The main drawback of this method is the efficiency of the generated VHDL code. This efficiency depends on the number of nodes of the graph generated internally by the translator from the abstract syntax tree corresponding to the input source code.

7. REFERENCES

[1]. I. Kuon, R. Tessier, J. Rose, "*FPGA Architecture: Survey and Challenges*," Foundations and Trends in Electronic Design Automation, vol. 2, no. 2 (2007), pp. 135-253.

[2]. P.P. Chu, "*FPGA Prototyping by VHDL Examples. Xilinx Spartan-3 Version*," John Wiley & Sons, 2008.

[3]. W. Meeus, K. Van Beeck, T. Goedeme, J. Meel, D. Stroobandt, "*An overview of today's high level synthesis tools*," Design Automation for Embedded Systems, vol. 16, no. 3 (2012), pp. 31-51.

[4]. F.M. Birleanu, "*A Method for Translating Pseudocode to HDL via Finite Automata*,"

University of Pitesti Scientific Bulletin. Series: Electronics and Computer Science, vol. 14, no. 1 (2014), pp. 11-18.

[5]. F.M. Birleanu, B.A. Enache, M. Alexandru, "*First Steps Towards Designing a Compact Language for the Description of Logic Circuits*," COMM 2016, pp. 357-360.

[6]. K.D. Cooper, L. Torczon, "*Engineering a Compiler*", 2nd ed., Elsevier, 2012.

[7]. J.E. Hopcroft, R. Motwani, J.D. Ullman, "*Introduction to Automata Theory, Languages and Computation*," 2nd ed., Addison-Wesley, 2001.

[8]. E.W. Dijkstra, "*ALGOL 60 Translation: An ALGOL 60 Translator for the X1*," ALGOL Bulletin. Supplement nr. 10, 1961.