A METHOD FOR TRANSLATING PSEUDOCODE TO HDL VIA FINITE AUTOMATA

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

Keywords: logic circuit design, finite automata, hardware description language, software algorithms, VHDL

Abstract: *While combinatorial logic circuit design methods are very easy to understand and automate, sequential logic circuit design is difficult for those that are new to the field. Textbooks focus on translating graphs into logic circuits, but very little is said about how these graphs can be easily obtained. We present a method that is intended to help software programmers enter the realm of logic circuit design. At the core of the method is a finite automaton based representation of the algorithm, which is then turned into a high-level logic circuit diagram that is easily translatable to hardware description language (HDL). The paper presents a detailed description of the method. It also contains an example as well as a discussion on benefits and drawbacks*.

1. INTRODUCTION

Digital electronics plays a major role in our lives. It is difficult to imagine life without smartphones, tablets and notebooks computers. But computers are not limited to these programmable devices. Tiny special-purpose computers can be found everywhere, from cars to remote controls and even toys. While general-purpose computer architectures can be programmed to perform specific tasks, another approach is to design custom hardware architectures for those specific tasks. The main benefits of this approach are smaller size and higher speed. On the other hand, the main drawbacks are the lack of reusability and the increased difficulty of hardware design compared to software design. This paper addresses the latter of these issues by proposing a method for easily translating software to hardware. The method makes digital circuit design [1, 2] available to everyone who has basic programming skills. Is is based on a virtual programming language (a pseudocode) with only three types of instructions, i.e. assignment, if, and while. This pseudocode is then translated to a deterministic finite automaton [3, 4] where every assignment instruction becomes a state, and finally the resulting graph is turned into a schematic represented in the VHDL hardware description language [5, 2]. A step-by-step description of the method is presented in Section 2, and an example is shown in Section 3. Section 4 discusses the benefits and the drawbacks of the proposed method, while the overall conclusions of the paper are formulated in Section 5.

2. DESCRIPTION OF THE METHOD

We start this section with an overview of the method and then we present a detailed description of each step.

2.1. Overview

A general overview of the method is shown in Figure 1. The first step in designing a logic circuit is the understanding the functional

specifications. They are usually in the form of a natural language description of what the circuit must do. These specifications must then be detailed until they become a series of clear commands in a virtual programming language that has only the following types of instructions:

- assignment
- if
- while.

A detailed description of the available pseudocode instructions is presented in the following subsection.

The next step of our method is the conversion of the algorithm expressed in pseudocode to an oriented graph where instructions become states and arcs are labeled with the condition that allows that transition. (Unconditional arcs have no label.)

Finally, this graph is turned into a logic circuit schematic in three steps:

1) identification of the memory elements
2) computation of the state transitions
3) assignments to variables

These steps are detailed in the following subsection.

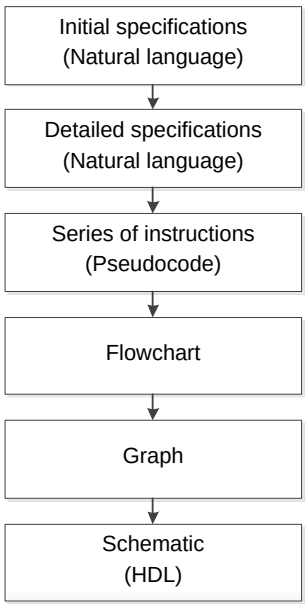

Fig. 1 Overview of the steps of the method.

2.2. Description

1. Pseudocode

We detail next the available pseudocode instructions.

Assigment instruction

The general format of this instruction is:

```
variable = expression
```

The variable is characterized by a name and by the number of bits it uses (which is also the number of flip-flops it will require in hardware). The expression may contain constants, variables, and basic arithmetic and logic operators that are available as combinatorial logic circuits (such as AND, OR, NOT, addition and substraction).

The flowchart equivalent of the assignment instruction is shown in Figure 2.a.

If instruction

The general format of this instruction is:

```
if (expression)
{
  instructions 1
}
else
{
  instructions 2
}
```

If the result obtained after evaluating the expression is nonzero, the instruction block `instructions 1` is executed. Otherwise, the instruction block `instructions 2` is executed.

The flowchart equivalent of the `if` instruction is shown in Figure 2.b.

While instruction

The general format of this instruction is:

```
while (expression)
{
  instructions
}
```

The instructions inside the curly brackets are executed as long as the result of evaluating the expression is nonzero.

The block of instructions may contain any valid instruction (including `if` and `while`). (The same holds true for the instruction blocks of the `if` instruction.)

The flowchart equivalent of the while instruction is shown in Figure 2.c.

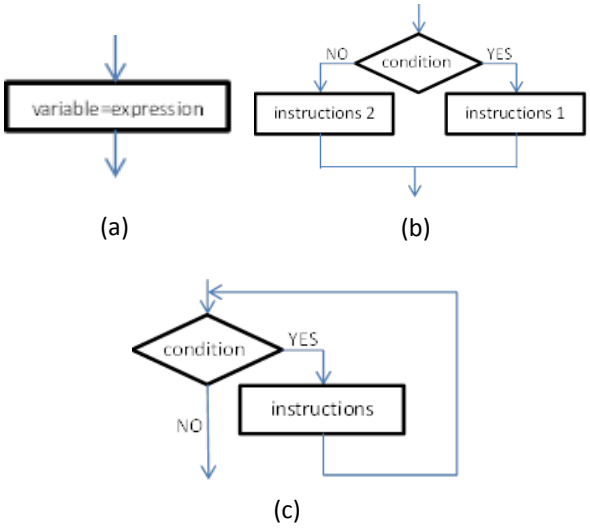

Fig. 2 The flowchart representation of: (a) the assignment instruction; (b) the if *instruction; (c) the* while *instruction.*

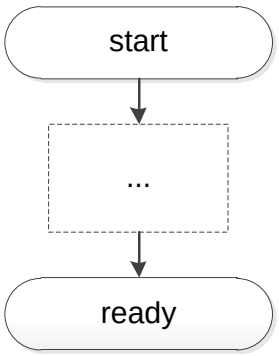

Fig. 3 The generic start/ready framework for algorithms expressed as flowcharts.

2. Graph

In order to obtain the graph for a certain algorithm expressed in pseudocode, we must first obtain the flowchart representation of the algorithm. This step is straighforward, as each instruction is replaced by its flowchart equivalent from Figure 2.

The beginning of the algorithm is marked by a *start* state and the end is marked by a *ready* state, as shown in Figure 3.

Then, every block (except for the rhombuses) is labeled with a state name and becomes a state in the graph.

Next, the states are connected by arcs according to the arrows in the flowchart. When two consecutive states are separated in the flowchart by a conditional instruction (a rhombus block), the arc that connects them is a conditional arc and the condition is determined by the condition in the rhombus and by the branch that is chosen (so, the condition is either (expression is nonzero) or (expression is zero)). (When passing through multiple decision blocks the resulting condition is composed of the conditions of all of those blocks, linked by the AND (&&) logical operator.)

3. Schematic

After having obtained the graph, we can convert it to a digital circuit schematic in three steps that are detailed below.

The generic schematic is shown in Figure 4. In order to simplify things, each input of the circuit should have a variable that should be written only once (at the beginning of the algorithm). This variable will, hence, act as a buffer for that input (whose eventual subsequent oscillations will not matter). (All input variables should be read at once in a single composed assingment instruction which should be the first instruction in the algorithm.) Similarly, a buffer variable should be used for each output of the circuit. This variable should be written only once (at the end of the algorithm) in order to prevent the output from oscillating during the execution of the algorithm.

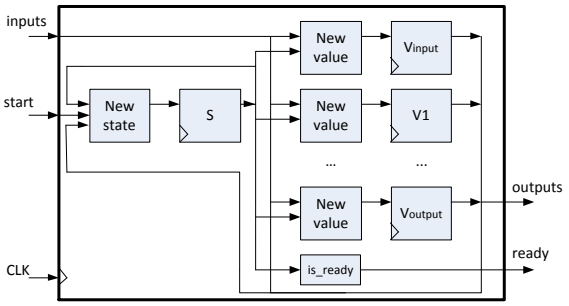

Fig. 4 The generic schematic of the circuit used for implementing a software algorithm.

The three steps for obtaining the schematic starting from the graph representation of the finite automaton are the following.

Identification of the memory elements

We need need memory elements for each of the variables in the algorithm. If a certain variable *V* needs *n* bits, *n* D-type flip-flops will

be used for implementing it. Figure 5 shows an example for a 3-bit variable.

We also need an additional variable (let us call it *S*) of $\lceil \log_2 n_S \rceil$ bits, where n_S is the number of nodes (states) in the graph.

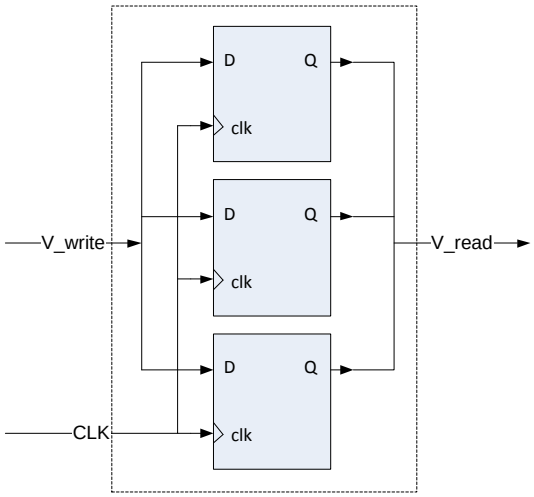

Fig. 5 A 3-bit variable implemented in hardware as a register composed of three D-type flip-flops.

Computation of the state transitions

For an algorithm that contains no decisions that are conditioned by variable values, the *New state* block in Figure 4 has only one input, which is the current state (i.e., the current value of the implicit variable *S*). Otherwise, the current state of some of the variables may be needed as input.

The *New state* block is a combinatorial block, and therefore in order to implement it we must first determine its truth table. We fill in this table by considering each arc in the graph in turn and then putting the destination state in the right-hand side of the table for the row (or rows, if there are multiple arcs pointing to the same destination state) where the source state is located in the left-hand side of the table.

For conditional arcs we must locate in the left-hand side of the truth table the row (or rows) where the source state of the arc is present and also the variables have such values that the condition on the arc is true (i.e., nonzero).

Table 1 shows some examples. The outputs for all remaining lines in the truth table should leave the current state unchanged.

After having obtained the truth table, turning it into a schematic can be done easily, as shown in [1, 2].

Table 1 Construction of the truth table for computing the new state: (top) for an unconditional arc; (bottom) for a conditional arc.

Arc	Current state, S	Variable V		New state
…	…	…		…
$s_i \longrightarrow s_j$	s_i	x		s_j
…	…	…		…
$s_x \xrightarrow{V==3} s_y$	s_x	3		s_y
…	…	…		…

Assignments to variables

In order to obtain the schematics for the *New value* blocks in Figure 4 (i.e., the blocks that compute the new values for the variables), we take one variable at a time and perform the same sequence of steps. For instance, let us take a variable *V* as an example.

First, we check the graph and the algorithm and determine the states where *V* is assigned a value. Let us imagine that *V* must take a certain value v_i in state s_i, another value v_j in state s_j, and another value v_k in state s_k. (For all other states, the value of *V* must remain unchanged.) Hence, the generic schematic for a *New value* block is that shown in Figure 6.

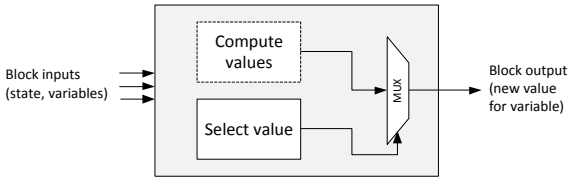

Fig. 6 The generic schematic of a New value *block from Figure 4.*

Next, we must identify for each of the states s_i, s_j and s_k the arcs that have that state as destination. This will help us obtain the truth table for the *Select value* combinatorial block. Each of the identified arcs will become one entry (or more entries, if there are additional inputs that are not used in the condition of that arc) in the table. For each arc, the right-hand side of the table will be filled in with the multiplexer input number of the value that variable *V* must take in the destination state of the arc, and the left-hand side of the table will be filled in with the source state of the arc and, if it is the case, the combination of input variables values for which

the condition of the arc is fulfilled (i.e., it is true, or nonzero). Note that this may result into multiple rows in the truth table for a single conditional arc.

The same operations must be performed for all the other variables.

4. HDL

While by using the method presented so far the schematic that corresponds to a certain algorithm an be computed by hand and represented graphically, is can as well be represented directly in a hardware description language such as VHDL [2, 5].

The main advantage consists in flexibility. For instance, variable *V* from Figure 5 may be implemented in VHDL as follows:

```
entity Variable_V is
   port ( clk: in std_logic;
   V_write: in std_logic_vector(2 downto 0);
   V_read: out std_logic_vector(2 downto 0));
end entity;

architecture Arch_V of Variable_V is
begin
   process(clk)
   begin
      if (clk'event and clk='1') then
         V_read <= V_write;
      end if;
   end process;
end Arch_V;
```

This VHDL code template remains the same for all the other variables in the algorithm. The only thing that changes is the definition of the *V_write* and *V_read* ports in the entity. Instead of 2 we must put the value of *n-1*, where *n* is the size in bits of the variable.

Another advantage of representing the schematic as HDL code is that by doing this we can automate the design of the combinatorial blocks (and of the entire schematic as well). For instance, the truth table in Table 1 could be implemented with the following VHDL code sequence:

```
...
architecture Arch_NewState of NewState is
begin
   process(S_in, V_in)
   begin
      if...
      elsif (S_in == s_i) then
         S_out <= s_j;
      elsif ((S_in == s_x) and (V_in == 3)) then
         S_out <= s_y;
      ...
      else
         S_out <= S_in;
      end if;
   end process;
end Arch_NewState;
```

A similar and easily customizable VHDL code template can be constructed for the *New value* and *Select value* blocks. All of these VHDL files can be created automatically by a software program that follows the steps of the method described above.

3. EXAMPLE

In this section we show an example of the method described previously. Let us consider that we want to implement a circuit for computing the product *c* of two 8-bit numbers, *a* and *b*.

Implementing this circuit in the combinatorial way requires dealing with a 2^{2x8}-rows truth table, which is impractical. A sequential implementation is, hence, the only viable solution. But while for a digital circuit designer this would be an easy task, a software designer would not know from where to start. However, it would be very easy for him to write the following pseudocode:

```
// Pseudocode for (c) = Product(a, b)
A = a
B = b
C = 0
while (B>0)
{
   C = C+A
   B = B-1
}
// c = C
```

(Note that we did not consider here an efficient algorithm for computing the product of two numbers, as our purpose is only to illustrate our method for converting software to hardware.)

Having this pseudocode algorithm, converting it to a flowchart representation is straightforward and leads to the result in Figure 7. Note that (as discussed before), the instructions `A = a` and `B = b` that buffer the input data of the algorithm into the variables *A* and *B*

can be considered (from the point of view of the method) to be a single composed instruction.

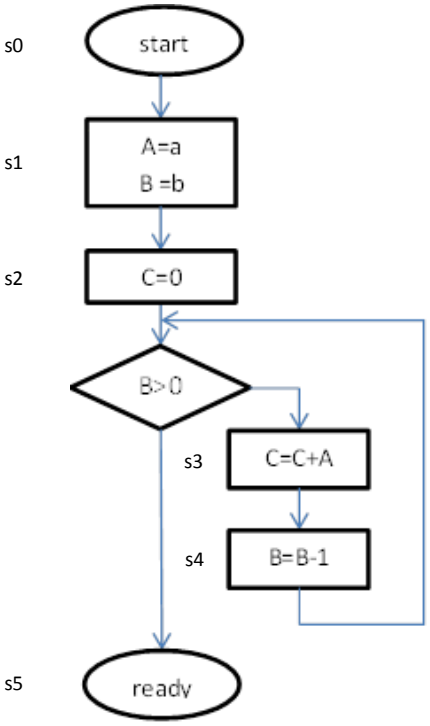

Fig. 7 The flowchart representation of the algorithm for computing the product of two numbers.

The next step is to label each nondecisional block in the flowchart with a state name. We already did that in Figure 7. Hence, we have 6 states: *s0*, *s1*, *s2*, *s3*, *s4*, and *s5*.

Next, we obtain the state transition graph of the finite automaton by first drawing 6 labeled circles (one for each of the 6 states) and then by connecting them with arrows as indicated by the flowchart.

The only possible difficulty here is the B>0 decision block. In this step of our method, it acts as being transparent, but affects the arcs passing through it. Any unconditional arc that passes through it turns into two conditional arcs – one (B>0) conditional arc, and one (!(B>0)) conditional arc. (Note that if the entering arc already had a condition (cond), the arcs that exit have conditions ((cond) and (B>0)) and ((cond) and (!(B>0))), respectively.)

The resulting graph is shown in Figure 8. Also note that the arc exiting the *start* state has condition (start == '1') and that from the *ready* state the circuit must move unconditionally to the *start* state at the next positive edge of the clock input.

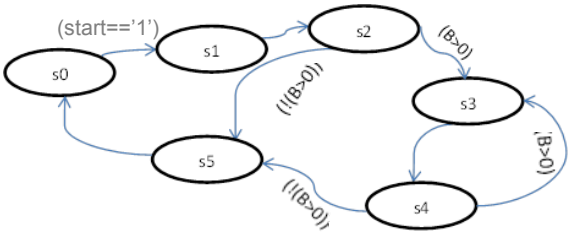

Fig. 8 The state transition graph for the finite automaton that corresponds to the flowchart in Figure 7.

If we now observe Figures 7 and 8, we notice that we have three variables (*A* - 8 bits, *B* - 8 bits, and *C* - 16 bits) and 6 states (and hence, the state variable *S* should be $\lceil \log_2 6 \rceil = 3$ bits wide). Therefore, the global schematic of the circuit for computing the product *c* = *a*x*b* according to the algorithm represented in Figure 7 looks like in Figure 9.

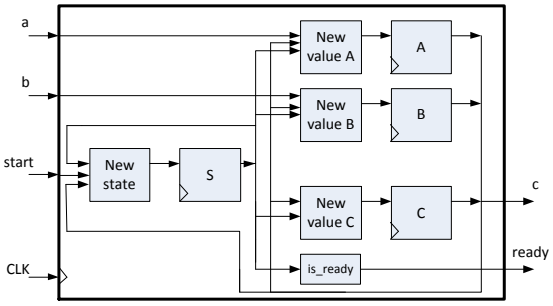

Fig. 9 The global schematic of the circuit for computing the product of two numbers.

For computing the *New state* block in Figure 9 we use the arcs of the graph in Figure 8, as discussed in Section 2.2.3. The resulting table is shown in Table 2. Each row is actually another form of representing the arcs from Figure 8.

Table 2 The truth table for the New state *block in Figure 9.*

start input	**Current state, S**	**Variable B**		**New state**
1	s0	x		s1
x	s1	x		s2
x	s2	(B > 0)		s3
x	s2	(!(B>0))		s5
x	s3	x		s4
x	s4	(B > 0)		s3
x	s4	(!(B>0))		s5
x	s5	x		s0
otherwise				same state

Now let us consider the implementation of block *New value A* from Figure 9.

Variable *A* is assigned only one value, *a*, in state *s1*. In all other cases, it is assigned its current value, *A*, as it must stay unchanged. Hence, the multiplexer in block *New value A* should have two inputs: 0 - for *a*, and 1 - for *A*. Input 0 must be selected in all the cases when the automaton is about to enter state *s1*, and input 1 in all the other cases. If we check Figure 8, we notice that there is only one arc coming to state *s1*, i.e., the conditional (start == '1') arc coming from *s0*. Hence, the truth table for the *Select value* block in block *New value A* has only two rows, as shown in Table 3.

Table 3 The truth table for the multiplexer selection in block New value A *from Figure 9.*

Current state, S		**Multiplexer selection**
s0		0
other		1

Variable *B* is assigned two different values in the program: value *b* in state *s1*, and value *B-1* in state *s4*. Plus implicit value, *B*, in all other cases. Hence, we need a multiplexer with 3 inputs (0: *b*; 1: *B-1*; 3: *B*). There is one unconditional arc that arrives in state *s1* (from *s0*), and one unconditional arc that arrives in state *s4* (from *s3*). Hence the circuit for choosing the proper selection for the multiplexer in block *New value B* can be implemented in a similar manner to that shown above.

For variable *C* we have three different assignments: *0* - for state *s2*, *C+A* - for state *s3*, and *C* - for all other states. State *s2* is reached through the unconditional arc coming from *s1*, and state *s3* can be reached through two arcs: the conditional arc (B>0) coming from *s2*, and the conditional arc (B>0) coming from *s4*. Hence, the truth table for the *Select value* block in *New value C* is that shown in Table 4.

Table 4 The truth table for the multiplexer selection in block New value C *from Figure 9.*

Current state, S	**Variable B**		**Multiplexer selection**
s1	x		0
s2	(B > 0)		1
s4	(B > 0)		1
otherwise			2

The VHDL descriptions of all the blocks discussed here can be obtained easily, as discussed in Section 2.2.4.

4. DISCUSSION

The method presented here for converting software algorithms to sequential digital logic circuits is intended to facilitate digital design for people that are familiar with software design. The method consists in a series of well-defined steps that can be either followed manually in order to obtain a schematic, or implemented as a software that is able to automatically translate an algorithm (written in the pesudocode presented in Section 2.2.1) to VHDL.

Its simplicity make this method a good didactic tool that helps with understanding the relationship between software and hardware. The resulting circuit architecture differs from the classic datapath-controlpath architecture, being more distributed and data-oriented.

The main drawback of the method is that it generates suboptimal circuits, as our focus was on functionality and simplicity, and not on optimization.

Our method starts from software and advances to hardware in order to obtain a sequential logic circuit that is able to perform the functionality described by the algorithm. The available algorithms do not take time into consideration, the purpose being to compute output data from input data (ideally, in zero-time). That means that our method can implement so far only circuits that could be (at least theoretically) implemented as combinatorial circuits (such as circuits for computing square roots, trigonometic functions, binary to binary-coded-decimal conversions, etc.) For instance, the multiplier we implemented as an example in Section 3 could be implemented combinatorially as well, but it would require a very large number of logic gates. Hence, our method is particularly suited for sequentializing impractically-large combinatorial circuits (or for reducing size at the expense of increasing the computation time).

We must note as well that the period for the clock signal must be chosen to be greater than the maximum propagation time of all combinatorial paths between register outputs and register inputs. Transitions from one state to another in the graph that corresponds to the

algorithm are performed only right after the positive edges of the clock signal (which go to the *clock* inputs of all the flip-flops in the design).

The roles of the *start* input and *ready* output are to command the start of the operation and to signal its end, respectively (as shown in Figure 10).

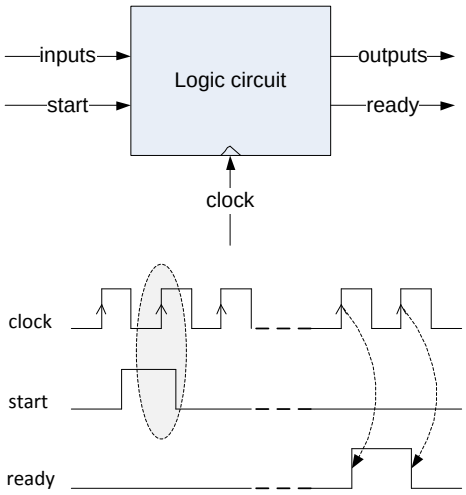

Fig. 10 The controlled operation mode for the circuits implemented with our method.

Continuous operation circuits could be obtained by slight modifications of the method, but care should be taken when designing the algorithm in order to avoid the oscillation of the outputs when the inputs stay unchanged. More precisely, the buffer variables for the outputs of the circuit should be written to only once during the algorithm.

Although not presented in this paper, function calls can be introduced quite easily in our method. This would require the introduction of the function call as a new instruction recognized by the pseudocode and the insertion of the graph of the function into the graph of the algorithm to be implemented.

5. CONCLUSIONS

We introduced in this paper a method for designing digital circuits from a software perspective. After writing the algorithm in the virtual programming language defined in the paper, the presented method can be applied in order to automatically obtain a sequential logic circuit that performs the same function as the algorithm.

The method is simple and didactic and provides a good starting point for designing a logic circuit whose desired function is known. Further, optimizations can subsequently be performed (such as rearranging instructions and performing them in parallel, which would lead to fewer states in the graph and, thus, fewer elementary logic circuits).

6. REFERENCES

[1]. Katz, Randy H., Borriello, Gaetano, "*Contemporary Logic Design*", 2nd ed., Prentice Hall, 2004.

[2]. Hwang, Enoch O., "*Digital Logic and Microprocessor Design with VHDL*", Cengage Learning, 2005.

[3]. Hopcroft, John E., Motwani, Rajeev, Ullman, Jeffrey D., "*Introduction to Automata Theory, Languages, and Computation*", 2nd ed., Addison-Wesley, 2001.

[4]. Aho, Alfred V., Lam, Monica S., Sethi, Ravi, Ullman, Jeffrey D., "*Compilers. Principles, Techniques, & Tools*", 2nd ed., Addison-Wesley, 2007.

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