Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 530586, 10 pages
http://dx.doi.org/10.1155/2015/530586

Research Article

An Algorithm for Generating Boolean Expressions in VHDL Based on Ladder Diagrams

Hongxia Xie and Zheng-Yun Zhuang

School of Computer and Computing Science, City College, Zhejiang University, 51 Huzhou Street, Hangzhou, Zhejiang 310015, China

Correspondence should be addressed to Zheng-Yun Zhuang; waynemcgwire@yahoo.com

Received 29 September 2014; Revised 23 December 2014; Accepted 5 January 2015

Academic Editor: Ricardo Femat

This study proposes an algorithm for generating the associated Boolean expression in VHDL, given a ladder diagram (LD) as the input. The purpose of the algorithm is to implement of field-programmable gate array- (FPGA-) based programmable logic controllers (PLCs), where an effective conversion from an LD to its associated Boolean expressions seems rarely mentioned. Based on this core thought, the conversion process of the algorithm first involves abstracting and expressing the encountered LD as an activity-on-vertex (AOV) graph. Next, an AND-OR tree in which AND-nodes and OR-nodes connote the series and the parallel relationships between the vertices of the AOV graph is constructed based on the AOV graph. Therefore, by a traversal to the AND-OR tree, the associated Boolean expression, as the output of the algorithm, can be easily obtained in VHDL. The proposed algorithm is then verified with an illustrative example, wherein a complicated LD is given as the input.

1. Introduction

1.1. Background. The emergence of programmable logic controller (PLC) has played a key role in the automatic control field (e.g., for industrial automation) since the 1970s. However, the efficiency of PLC usually relies on the CPU frequency, due to the design of PLC used to adopt the sequential-processor solution. This had limited the application of PLC. For example, in the past, PLC did not serve some application fields like semiconductor processing, laser-based precision cutting, and fault detecting. Apparently, to fulfill the requirements from these fields, the traditional millisecond-level PLC controller, in itself, must be improved to respond to the peripherals rapidly, to have high-speed communications, and to execute the control logics concurrently [1].

Fortunately, with rapid developments of programmable logic device (PLD) and field-programmable gate array (FPGA) nowadays, it is able to have high-speed PLCs which are based on FPGA because of the concurrent execution mechanism. In this manner, the efficiency of the controller can be improved. Therefore, designing high-speed, FPGA-based PLCs becomes a trend.

1.2. Literature Review. Magnussen [2] wrote the first study that proposed the idea of translating PLC programs to VHDL ones. Following the study, Miyazawa et al. [3] discussed the relationships that can associate ladder diagrams (LDs) with Boolean expressions and illustrated two implementation methods of "PLC cyclic scan mechanism" in the VHDL context. Ikeshita et al. [4] studied the mapping relationship between the fundamental elements (i.e., steps, actions, and transitions) in a sequential flow chart (SFC) and the Boolean variables expressed in Verilog-HDL language. Adamski and Monteiro [5] and A. Wegrzyn and M. Wegrzyn [6] discussed the interpretational relationship from Petri-net-based specifications to Boolean expressions in VHDL, but both studies have not proposed a conversion algorithm. Ichikawa et al. [7] summarized the three kinds of operational modes of a PLC process when it is executed by the logic circuits inside the FPGA module. C. Economakos and G. Economakos [8, 9] proposed the idea of converting simple LDs to VHDL programs by the help of medium codes written in C language. Du et al. [10] explained the way to realize concurrent executions of programs in FPGA modules that are LD-based, while Alonso et al. [11] adopted the model-driven engineering

approach and have generated VHDL codes from traditional LDs partially.

As can be reviewed from the literature, most of the existing researches mainly focus either on studying the relationships between the ladder-logic programming language (LAD) elements of PLC and the HDL elements or on proposing the framework to implement and execute relevant PLC processes on the FPGA platform. Nevertheless, a successful translation among the different expressional languages is also an important and fundamental topic for the utilization of FPGA-based PLC.

This study serves this purpose. Since LD language is one major expressional language for PLC designing and VHDL is a main HDL expressional language, the industry still lacks an efficient and well-known algorithm that can convert the LAD-based graphical LDs into Boolean expressions in VHDL directly and completely.

Section 2 proposes the way to abstract the LDs using well-known activity-on-vertex (AOV) graph and the ways and rules to construct an AND-OR tree based on the AOV graph. Section 3 introduces the body of the proposed algorithm in pseudocodes with the traversal method to interpret the associated AND-OR tree to Boolean expressions as the output for VHDL programming. Section 4 illustrates the algorithm with a complicated LD example, in which the applicability of the algorithm is shown. In Section 5, concluding remarks are given.

2. The Abstraction of Ladder Diagrams

The conversion of LDs to text-based descriptions is an important issue for the use of VHDL, as addressed in Section 1.2. LDs are unable to be implemented on or executed by FPGA due to the fact that LAD is a graphical language.

In this section, in order to complete the conversion process, the way to abstract an LD as its corresponding AOV graph is introduced. Next, the way to construct an AND-OR tree based on the AOV graph, while the AOV graph is gradually reduced to be a null graph, is described. To illustrate these ways, an LD instance which is originally designed in LAD, as shown in Figure 1, is used as an example input throughout this section and Section 3.

2.1. The Abstraction of the AOV Graph from the LD. In this study, the following acronyms are used to abstract the LDs.

(i) Symbol of circled-s represents the starting node of the rungs inside an LD, which is drawn as a left-margin connector component in LAD.

(ii) Symbol of circled-b represents the contact component in LAD.

(iii) Symbol of circled-f represents the functional components (e.g., functional blocks or timer modules) in LAD.

(iv) Symbol of directed arrow $\rightarrow$ represents the connection between two components in LAD.

According to these definitions, the LD in Figure 1 can be abstracted as an AOV graph [12], as shown in Figure 2.

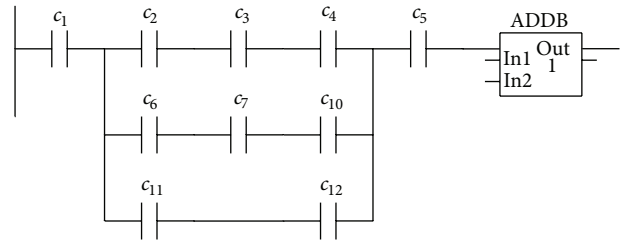


Figure 1: The LD instance designed in LAD.

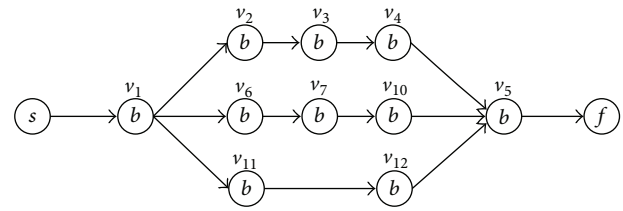


Figure 2: The associated activity-of-vertex (AOV) graph of Figure 1.

2.2. Constructing the AND-OR Tree Based on the AOV Graph

2.2.1. The Definition of AND-OR Tree. An AND-OR tree is a finite set T that includes l ($l \geq 0$) nodes. When T is empty, it is a null tree. Otherwise, it is not a null tree and the following two conditions hold.

Condition (1). There is one and only one "root node" p_0.

Condition (2). Other nodes in T except p_0 can be divided into m subsets $(T_1, T_2, \ldots, T_m)$ which do not intersect with each other, where each subset is an AND-OR tree in itself; these subsets are named as the subtrees of p_0.

2.2.2. The Use of the AND-OR Tree in This Study. AND-OR trees are often used to reduce the problems (or goals) to conjunctions and disjunctions of subproblems (or subgoals). Taking this feature, the definition of the AND-OR tree that appropriates to the conversion problem encountered by this study is extended as follows.

Condition (3). All the leaf nodes of the AND-OR tree is from the vertices in the AOV graph. In addition, all the nonleaf nodes of the AND-OR tree are nodes which are called either AND-nodes or OR-nodes. They will be defined in *Condition (5)* and *Condition (6)*.

Condition (4). If the root node, p_0, of an AND-OR tree has child nodes, then it must have 2 or more child nodes. Note that this rule applies to every level of an AND-OR tree till the leaf since, in fact, any leaf node still holds AND-OR tree properties but they have no child node. This further implies that each nonleaf node must have 2 or more child nodes.

Condition (5). The child nodes (and not the grandchildren nodes) of an AND-node possess a "serial

relationship." This means that the function of an AND-node in such a tree is to conjunct its child nodes and all the child nodes are in a "serial relationship."

Condition (6). The child nodes (and not the grandchildren nodes) of an OR-node possess a "parallel relationship." This means that the function of an OR-node in such a tree is to link its child nodes and all the child nodes are in a "parallel relationship."

2.2.3. Rules to Transform an AOV Graph to an AND-OR Tree. For each vertex v_i in the AOV graph, an "in-out-degrees pair" (in(v_i), out(v_i)) is defined as a vector indicating the number of directional edges entering (pointing inward to) v_i and the number of edges leaving (pointing outward from) v_i. Based on the properties of the AND-OR tree defined in Sections 2.2.1 and 2.2.2, the transformation rules to convert an LD-based AOV graph into the AND-OR tree are as follows.

(i) The AND-transformation rule: given two vertices v_A and v_B in an AOV graph, T, and S_X is a subset of T that has g ($g \geq 0$) vertices. If all the vertices in S_X are reachable from v_A and all of them can reach v_B though the directional edges and if v_A, v_B, and S_X satisfy the following conditions, then an AND-node is established (and an AND-subtree is constructed).

(i.1) The in-out-degrees pair (in(v_A), out(v_A)) satisfies in(v_A) ≥ 0 and out(v_A) $= 1$.

(i.2) The pair (in(v_B), out(v_B)) satisfies in(v_B) $= 1$ and out(v_B) ≥ 0.

(i.3) $\forall v_i$, $v_i \in S_X$ (in(v_i), out(v_i)) satisfies in(v_i) $= 1$ and out(v_i) $= 1$.

When the above conditions hold, an AND-node is established with directional links (connoting the parent-child relationships) pointing toward vertices v_A, v_B and $\forall v_i$, $v_i \in S_X$, individually. This yields an AND-tree with the established AND-node as the root which has $(g+2)$ child nodes. In addition, the $(g+2)$ vertices in the AOV graph that correspond to the $(g+2)$ child nodes of this AND-subtree are replaced by one single "&-vertex."

(ii) The OR-transformation rule: if in the AOV graph there exists a set, S_Y, of h vertices, all of which have a same antecedent vertex v_U and a same descendant vertex v_W, an OR-node is established with directional links pointing toward the l vertices in S_Y individually. This yields an OR-tree with the established OR-node as the root which has h child nodes. Note that the OR-tree is an AND-OR tree in itself. In addition, the h vertices, and not v_U and v_W, in the AOV graph which correspond to the h child nodes of the OR-tree are replaced by one single "|-vertex."

2.2.4. A Short Example. By applying the rules defined in Section 2.2.3, the AOV graph in Figure 2 can be stepwise converted into an AND-OR tree. Such process is as shown in Figure 3.

In the AOV graph shown in Figure 3(a), it is observed that "in(v_2) $= 1 >= 0$ and out(v_2) $= 0$" and that "in(v_4) $= 1$ and out(v_4) $= 1 >= 0$." And based on the observation that "v_3 can reach v_4," "v_3 can be reached by v_2," and "in(v_3) $=$ out(v_3) $= 1$," all of the AND-transformation conditions (i.1), (i.2), and (i.3) are held. This means that an AND-translation can be applied to these nodes and an AND-tree can be constructed by establishing an AND-node as the root with v_2, v_3, and v_4 as its child nodes. In such a case, $g = 1$ (i.e., v_3) and the constructed AND-tree has $(g + 3) = 4$ nodes in total. In addition, vertices v_2, v_3, and v_4 in the original AOV graph are substituted by one single &-vertex, which is named as $\&_1$. The above process is shown in Figure 3(b).

Similar transformation processes can be applied to $\{v_6, v_7, v_{10}\}$ and $\{v_{11}, v_{12}\}$, as also shown in Figure 3(b). The process for $\{v_6, v_7, v_{10}\}$ is analogical to that for $\{v_2, v_3, v_4\}$, since these two subsets of vertices have a similar structure. The process for $\{v_{11}, v_{12}\}$ is also similar, but, for the reason that $g = 0$ for this AND-transformation case (i.e., $S_X = \phi$), the established AND-tree only has a total of 3 nodes. Note that after these AND-transformations, the AOV graph is further simplified by two additional &-vertices, that is, $\&_2$ and $\&_3$, as shown in Figure 3(b), wherein no more AND-translations can be performed.

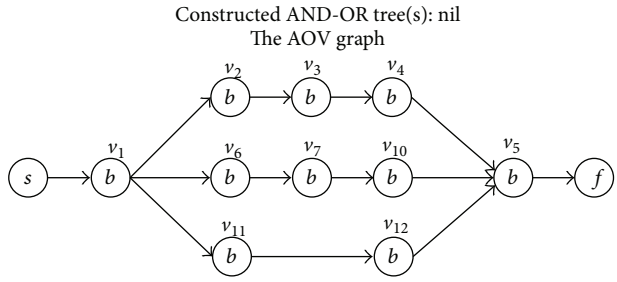
Now the clues for the possible OR-transformations can be searched. As one can observe in Figure 3(b), the &-vertices in the AOV graph perfectly meet the conditions for an OR-transformation. That is, $S_Y = \{\&_1, \&_2, \&_3\}$, $v_U = v_1$, and $v_W = v_5$ where $h = 3$. Then the OR-transformation rule is applied. An OR-subtree which has 3 child nodes (i.e., does not count the grandchildren) is constructed and the relevant &-vertices of the AOV graph in S_Y are replaced with one single |-vertex, named "$|_1$." The above process is illustrated in Figure 3(c).

In Figure 3(c), since no more OR-transformations can be performed, it is the turnback to search for clues for the possible AND-transformations. As can be observed in the AOV result graph in Figure 3(c), another AND-transformation can be performed. Now by observing that $S_X = \{v_1, |_1, v_5\}$, $v_A = s$, $v_B = f$, and $g = 3$, an AND-tree which has $g + 2 = 5$ children nodes (again without counting the root's grand- and grand-grandchildren) is constructed in Figure 3(d). Since after such process the AOV graph becomes empty, the whole conversion algorithm ends here.

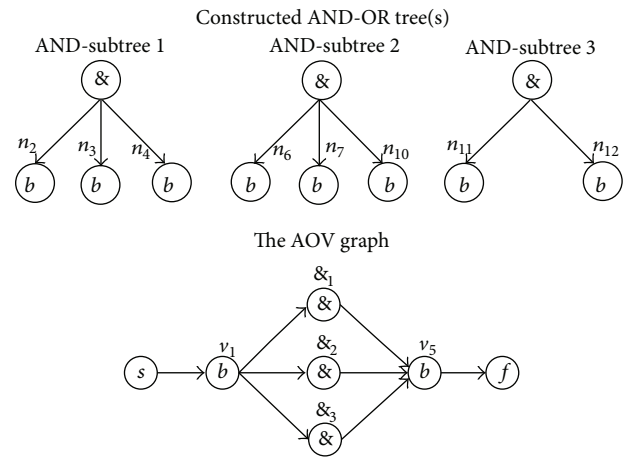
As a short summary, after the "abstraction subalgorithm" and the "transformation subalgorithm" depicted in this section, the final "interpretation subalgorithm" can be easily performed by a traversal to the constructed AND-OR tree. These subalgorithms constitute the proposed algorithm and the bodies of them will be defined in Section 3.

3. The Body of the Algorithm

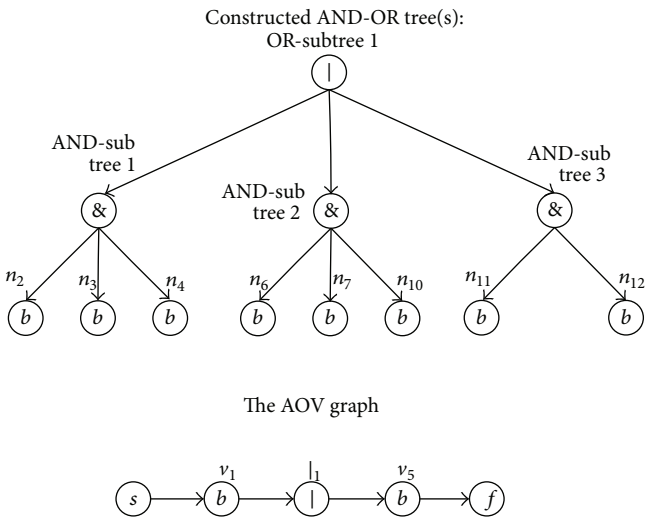
In this section, relevant data structures and methods are introduced first, followed by the body of the proposed algorithm, including the functions to construct the final AND-OR tree and to obtain the Boolean interpretations by traversing the tree. Note that, in this algorithm, because the final AND-OR tree is to be constructed "on the fly" (which means



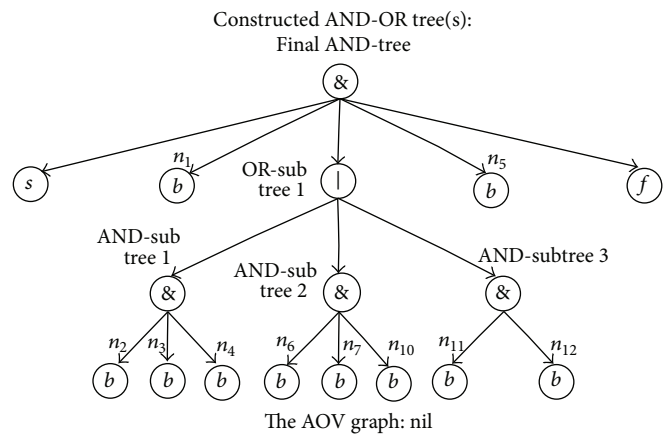
(a) The original AOV graph with null AND-OR tree(s)



(b) Results of first stage AND-translation: the constructed AND-subtrees and the reduced AOV graph



(c) Results of second stage OR-translation: the constructed OR-subtree and the reduced AOV graph



(d) Results of third stage AND-translation: the constructed final AND-tree and the reduced AOV graph

Figure 3: The transformation process from an AOV graph to an AND-OR tree.

```
public class Node {
        string name; // stores the name of the node
        List<Node> preNodes; // stores the predecessor nodes of this node
        List<Node> postNodes; // stores the successor nodes of this node
        ...
}.
```

Algorithm 1

the nodes in the AND-OR tree are constructed by relinking the vertices in the AOV graph, using the same data structure) on such a base, the term "node" is used to denote not only AOV vertex in the AOV graph but also node in the AND-OR tree. And since an original AOV graph fully describes an LD, the input variable named "LD" is, in fact, some specific LD in AOV-graph representation format.

3.1. Data Structure and AND-OR Tree Conversion Algorithms

3.1.1. Data Structure. The classes, in C#, are defined as shown in Algorithm 1.

The classes for AND-nodes and OR-nodes are derived from the Node class, which are defined as shown in Algorithm 2.

```
public class AndNode: Node {
        List<Node> children; // stores the child nodes of this AND-node
        ...
}.
And,
public class OrNode: Node {
        List<Node> children; // stores the child nodes of this OR-node
        ...
}.
```

Algorithm 2

```
Function-1: ConvertRungsIntoAndOrTree
Input: a ladder diagram LD
Output: an AND-OR tree that corresponds to LD
for rung: LD.rungs
    currentNode ← rung.nodes[0]
    currrentDic ← new Dictionary<Node, List<Node>>( )
    ConstructAndOrTree(currentNode, currentDic)
end for
```

Algorithm 3: Function-1: main body of the algorithm.

3.1.2. Algorithm. The main algorithmic body to convert the rungs in LD represented in an AOV graph to an AND-OR tree, named "Function-1," is shown in Algorithm 3. This function utilizes the generic class Dictionary$\langle K, V\rangle$ in C#, wherein K and V are a key-value pair and they can be differently typed. In Algorithm 3, K is defined as the type of a node and V is defined as the list that stores the data of the node.

"Function-2" shown in Algorithm 4(a) lists the pseudocodes of a ConstructAndOrTree() function mentioned in Algorithm 3. Its algorithm is done by calling another ConstructAndOrTreeHelper() function, whose pseudocodes are listed in Algorithm 4(b) as "Function-3," after a children list (i.e., "childrenList") is created (i.e., newed). These two functions, together, construct the AND-OR tree for one single rung, given the first node of a stair rung of a ladder. As can be seen, Function-2 mainly focuses on transforming the nodes that meet the AND-transformation rule (in Section 2.2.3) into an AND-subtree with an additional AND-node established. Function-3 is in charge of converting the parallel nodes that meet the OR-transformation rule into an OR-subtree, wherein the additional OR-node is established and the tree structure is organized by calling ConstructOrNode(), whose algorithm is named as "Function-4" as shown in Algorithm 4(c).

3.2. The Boolean Expression Interpretation Algorithm. As the AND-OR tree construction algorithm shown in Algorithm 4, after an AND-OR tree is constructed as an output, the tree can be traversed to have the text-based Boolean expression interpretations. "Function-5," as shown in Algorithm 5, traverses the AND-OR tree in order and at the same time generates the associated Boolean expression in VHDL code format.

By taking the AND-OR tree shown in Figure 3 as an example input, if Function-5 is called, the interpreted Boolean expression, which is able to be coded directly in VHDL programs, is as follows:

> (1 AND X(1) AND ((X(2) AND X(3) AND X(4)) OR (X(6) AND X(7) AND X(10)) OR (X(11) AND X(12))) AND X(5)).

Note that the constant "1" represents the circled-s vertex in the AOV graph, which is the left-margin connector in the original LD.

4. Illustrative Example

This section offers a more complicated example to verify the proposed conversion algorithm. Figure 4(a) shows the LD designed in LAD originally, while Figure 4(b) is its corresponding AOV graph, obtained according to the abstraction subalgorithm introduced in Section 2.1.

Figure 5 illustrates the process of building the AND-OR tree. With Figure 4(b) as its input, this is done by the algorithm proposed in Section 3 which is designed according to the conversion logics discussed in Section 2.2.

At first, by Function-3 and Function-4, the two rungs which are attached to the left-margin connector (i.e., the vertex "circled-s") and are delimited by vertex v_4, rooted from the contacts v_{11} and v_{16}, are converted and reorganized into an OR-subtree, as shown in Figure 5(a). Next, the rungs which are attached to v_4 in Figure 5(b) are converted and reorganized into the other OR-subtree, as shown in Figure 5(c). This yields an AOV graph in Figure 5(d).

As can be observed in Figure 5(d), no more OR-transformations can be performed and the rest nodes (vertices) satisfy the conditions for an AND-transformation. Therefore, an AND-transformation is performed by Function-2. The final AND-OR tree with an established AND-node as the root is constructed, as shown in Figure 5(e).

Finally Function-5 takes place. It traverses the AND-OR tree in Figure 5(e) and obtains the Boolean expression interpretation in VHDL, which is as follows:

> (1 AND ((X(1) AND ((X(2) AND X(3)) OR (X(11) AND X(12)))) OR (X(16) AND X(17) AND X(20))) AND X(4) AND ((X(21) AND X(22) AND X(23) AND X(24)) OR (((X(5) AND X(6)) OR (X(13) AND X(14))) AND X(7) AND (X(10) OR X(15))))).

(a) Function-2: ConstructAndOrTree()

```
Function-2: ConstructAndOrTree
Input: first node of a rung named "nd" and a dictionary typed Dictionary<Node, List<Node>> named "dic"
Output: an And-Or tree
childrenList ← new List<Node>()
ConstructAndOrTreeHelper(nd, dic, childrenList)
if (childrenList.Count > 1)
    andNode ← new AndNode();
    andNode.preNodes ← childrenList[0].preNodes
    andNode.postNodes ← childrenList[childrenList.Count − 1].postNodes
    andNode.children.Add(childrenList)
    Update postNodes list of node of which postNodes contains nd
    Update preNodes list of node of which preNodes contains nd
    Remove nodes included in childrenList in the rung
    Update values which contain nd in the dic
end // if
```

(b) ConstructAndOrTreeHelper(), called by ConstructAndOrTree()

```
Function-3: ConstructAndOrTreeHelper
Input: a node typed Node named "node", a dictionary typed Dictionary<Node, List<Node>> named "dic", a list typed
List<Node> named "childrenList"
childrenList.add(node)
if (node.postNodes.count == 0)
      Return
end // if
if (node.postNodes.count == 1)
      postNode ← node.postNodes[0]
      if (postNode.preNodes.count == 1)
         ConstructAndOrTreeHelper(postNode, dic, childrenList)
      End
      Else
         insert node into dic[postNode]
      end
else
      subDic ← new Dictionary<Node, List<Node>>()
      for currentNode: node.postNodes
         ConstructAndOrTree(currentNode, subDic)
      end // for
      while (subDic.Key.count > 1)
      begin
         for keyValuePair: subDic.keyValuePairs
         begin
            if (keyValuePair.Key.preNodes.count == keyValuePair.Value.count)
               orNode ← ConstructOrNode(keyValuePair, node)
               remove keyValuePair in the subDic
               ConstructAndOrTree(orNode, subDic)
         end // for
      end // while
      orNode ← ConstructOrNode(keyValuePair which is the only item in the subDic, node)
      ConstructAndOrTreeHelper(orNode, dic, childrenList)
```

(c) ConstructOrNode(), called by ConstructAndOrTreeHelper()

```
Function-4: ConstructOrNode
Input: a key-value-pair named "kvp" of which key is the successor node of the constructed orNode and value is the children
of the orNode, the predecessor node named "nd" of the orNode
Output: an orNode
orNode ← new OrNode()
orNode.children ← kvp.value
orNode.preNodes ← nd
orNode.postNodes ← kvp.key
delete elements included in kvp.value in kvp.key.preNodes
```

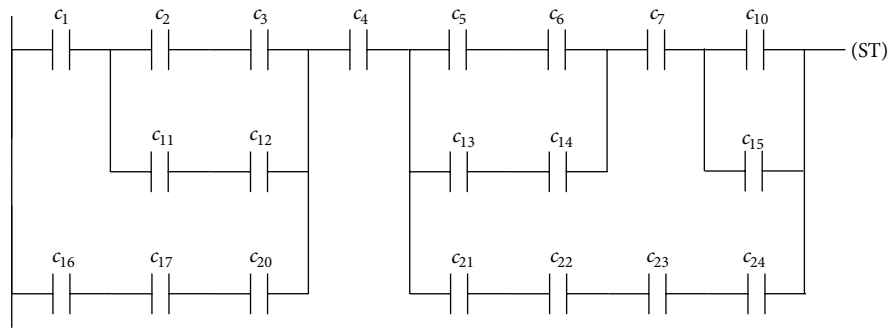
Algorithm 4: Continued.

```
kvp.key.preNodes.Add(orNodes)
delete elements included in kvp.value in nd.postNodes
nd.postNodes.append(orNode)
delete elements includes in kvp.value in ladderdiagram and insert orNode into ladderdiagram
return orNode
```

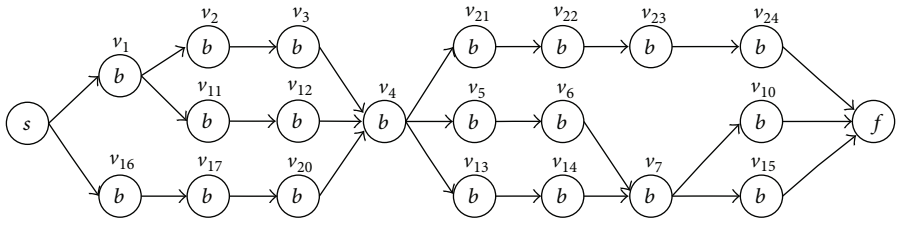
Algorithm 4: The AND-OR tree construction algorithm.

```
Function-5: TranslateAndOrTreeIntoBooleanExpression
Input: the root node of the And-Or tree named "root"
Output: the Boolean expression of the And-Or Tree
expression ← new string()
if (root.children.count == 0)
    Return expression ← root.toString()
end // if
if (the type of root is AND-Node)
    for childNode: root.children
        subExpression ← TranslateAndOrTreeIntoBooleanExpression(childNode)
        Expression ← expression + "AND" + subExpression
    end for
else
    for childNode: root.children
        subExpression ← TranslateAndOrTreeIntoBooleanExpression(childNode)
        Expression ← expression + "OR" + subExpression
    end // for
end // if
Expression ← "("+ expression +")"
return expression
```

Algorithm 5: The interpretation algorithm.

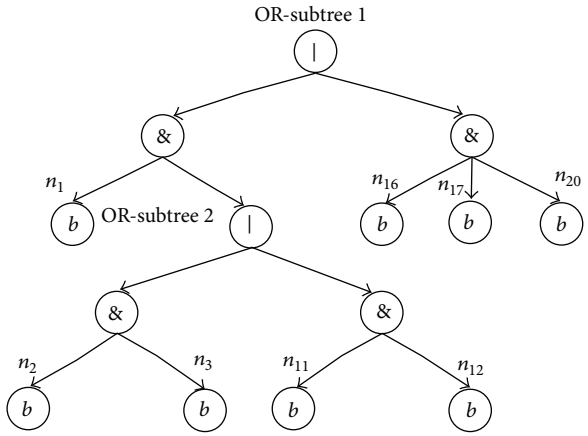


(a) Source LD

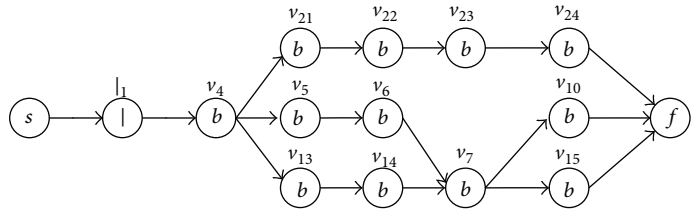


(b) Associated AOV graph

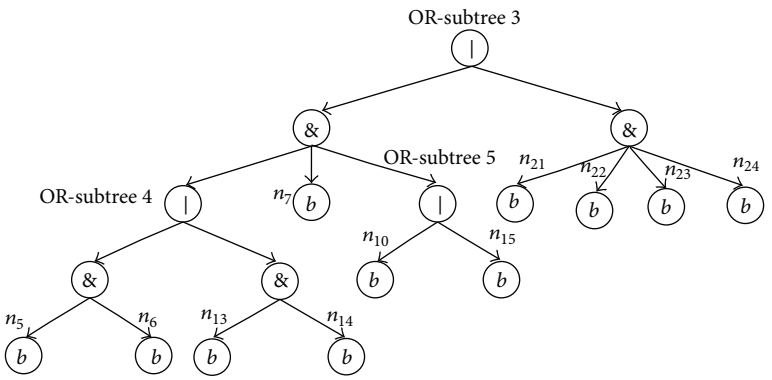
Figure 4: A more complicated example: source LD and its AOV graph.



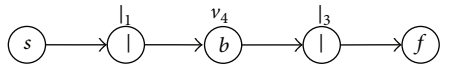
(a) The constructed OR-subtree from Figure 4(b)



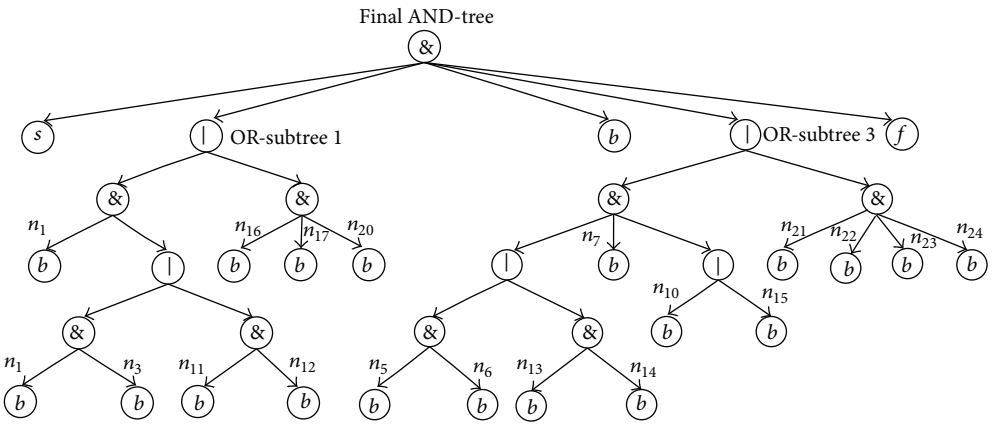
(b) The reduced AOV graph of Figure 4(b) after the OR-subtree is constructed in (a)



(c) The constructed OR-subtree from (b)



(d) The reduced AOV graph of (b) after the OR-subtree is constructed in (c)



(e) The final AND-tree (with a null AOV graph which is not shown)

Figure 5: AND-OR-tree construction for the complicated example: logics performed by the algorithm.

The above expression is equivalent to the AND-OR tree itself and is ready to be programmed.

5. Conclusion

This study proposes an algorithm to convert LDs into its associated VHDL Boolean expressions. With the help of AOV graph, the LD is abstracted as an AOV graph at first and next it is transformed to the associated AND-OR tree. Then, the final Boolean expression in VHDL is interpreted by AND-OR tree traversal. The algorithm is helpful to VHDL programming in dealing with the logics behind an LD drawn in the LAD language. This is a possible contribution of this study in that the algorithm supports the development of next-generation high-speed PLCs based on FPGA.

```
PACKAGE VHDLFUNPACKAGE IS
PROCEDURE MOVB(CONSTANT C1: IN INTEGER RANGE 255 DOWNTO 0;
                    SIGNAL SOUT: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
PROCEDURE ADDB(SIGNAL S1: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                    SIGNAL S2: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                    SIGNAL SOUT: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END VHDLFUNPACKAGE;
PACKAGE BODY VHDLFUNPACKAGE IS
PROCEDURE MOVB(CONSTANT C1: IN INTEGER RANGE 255 DOWNTO 0;
                    SIGNAL SOUT: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)) IS
        BEGIN
            SOUT <= CONV_STD_LOGIC_VECTOR(C1, 8);
        END PROCEDURE MOVB;
PROCEDURE ADDB(SIGNAL S1: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                    SIGNAL S2: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                    SIGNAL SOUT: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)) is
        BEGIN
            SOUT <= S1 + S2;
        END PROCEDURE ADDB;
END VHDLFUNPACKAGE;
```

Algorithm 6

In addition, the proposed algorithm is verified with a complicated LD. In fact, in this study, two LDs, a simpler one and a more complicated one, serve as the inputs of the algorithm. The results of both of which are correct. This lends support to the efficacy of the proposed algorithm. However, for illustration simplicity, the example only demonstrates the cases of normally open and normally closed contacts. About the transformation of special functional blocks, please refer to Appendix.

Thirdly, as an extra function, the algorithm is able to scrutinize the validity of any of an LD instance. This is based on the claim of a previous study [12]: if any rung in an LD instance cannot be successfully converted to an AND-OR tree, the LD is invalid. The algorithm proposed by this study can also well serve such examination purpose.

Finally, future works can involve the practical and economical interests pertaining to the algorithm. Demonstrating the example applications of the algorithm is a future work. An IDE tool for visually designing the LDs and for transforming the LD programs into VHDL programs directly is an implementation issue. As can be imagined, this will involves a data structure that is to save and load the LDs, as well as to organize the LD in another graph data structure to store the AOV graph, so that the subsequent conversion process based on the proposed algorithm can follow. More critically, with one such IDE tool, a new developer can easily develop VHDL programs to run on the FPGAs. In the meanwhile, he/she just needs to concentrate on writing a correct LD program, instead of any new tool or new language, leave the rest jobs for the IDE tool that converts the LD program into a VHDL one, and leave the remaining jobs for the other tool that compiles the VHDL program (e.g., Quartus) as FPGA executables. This is, exactly, the major motivation of this study to design the algorithm, which can serve as a key part of the IDE tool and can be used to convert different LD diagrams to their associated logical expressions.

Appendix

For the special functional blocks, for example, ADD, MUL, and MOV, or other functional blocks, no matter which, the translation processes are identical. The procedure of such a process is described as follows.

(1) Use VHDL to realize the function that is associated with the mentioned instruction (functional block). That is, it is to write the codes for the relevant library functions in VHDL.

(2) When there is any such instruction in the LD diagram, just invoke the associated library function.

For example, for the 8-bit adder block and the 8-bit MOV instruction, we can firstly compose the library functions for them in VHDL as shown in Algorithm 6.

Then when there is the 8-bit adder block or the 8-bit MOV instruction required in the LD program, the translation process just needs to invoke these library functions.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] M. M. Patil, S. Subbaraman, and P. S. Nilkund, "IEC control specification to HDL synthesis: considerations for implementing PLC on FPGA and scope for research," in *Proceedings of the*

International Conference on Control, Automation and Systems (ICCAS '10), pp. 2170–2174, October 2010.

[2] B. Magnussen, "A parallel control computer structure for complex high speed applications," in *Proceedings of 1st IEEE International Conference on Engineering of Complex Computer Systems (Held jointly with 5th CSESAW, 3rd IEEE RTAW and 20th IFAC/IFIP WRTP)*, pp. 385–388, Ft. Lauderdale, Fla, USA, November 1995.

[3] I. Miyazawa, T. Nagao, M. Fukagawa, Y. Itoh, T. Mizuya, and T. Sekiguchi, "Implementation of ladder diagram for programmable controller using FPGA," in *Proceedings of the 7th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '99)*, vol. 2, pp. 1381–1385, 1999.

[4] M. Ikeshita, Y. Takeda, H. Murakoshi, N. Funakubo, and I. Miyazawa, "An application of FPGA to high-speed programmable controller: development of the conversion program from SFC to Verilog," in *Proceedings of the 7th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '99)*, vol. 2, pp. 1386–1390, Barcelona, Spain, October 1999.

[5] M. Adamski and J. L. Monteiro, "From interpreted Petri net specification to reprogrammable logic controller design," in *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE '00)*, pp. 13–19, December 2000.

[6] A. Wegrzyn and M. Wegrzyn, "Petri net-based specification, analysis and synthesis of logic controllers," in *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE '00)*, pp. 20–26, December 2000.

[7] S. Ichikawa, M. Akinaka, R. Ikeda, and H. Yamamoto, "Converting PLC instruction sequence into logic circuit: a preliminary study," in *Proceedings of the International Symposium on Industrial Electronics (ISIE '06)*, pp. 2930–2935, July 2006.

[8] C. Economakos and G. Economakos, "Optimized FPGA implementations of demanding PLC programs based on hardware high-level synthesis," in *Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '08)*, pp. 1002–1009, September 2008.

[9] C. Economakos and G. Economakos, "An architectural exploration framework for efficient FPGA implementation of PLC programs," in *Proceedings of the 17th Mediterranean Conference on Control and Automation (MED '09)*, pp. 1172–1177, Thessaloniki, Greece, June 2009.

[10] D. Du, Y. Liu, X. Guo, K. Yamazaki, and M. Fujishima, "Study on LD-VHDL conversion for FPGA-based PLC implementation," *International Journal of Advanced Manufacturing Technology*, vol. 40, no. 11-12, pp. 1181–1190, 2009.

[11] D. Alonso, J. Suardíaz, P. J. Navarro, P. M. Alcover, and J. A. López, "Automatic generation of VHDL code from traditional ladder diagrams applying a model-driven engineering approach," in *Proceedings of the 35th Annual Conference of the IEEE Industrial Electronics Society (IECON '09)*, pp. 2416–2421, Porto, Portugal, November 2009.

[12] Y. Yan and H. Zhang, "Compiling Ladder Diagram into Instruction List to comply with IEC 61131-3," *Computers in Industry*, vol. 61, no. 5, pp. 448–462, 2010.