FPGA-BASED IMPLEMENTATION AND COMPARISON OF RECURSIVE AND ITERATIVE ALGORITHMS

Valery Sklyarov, Iouliia Skliarova, Bruno Pimentel

Electronics and Telecommunications Department / University of Aveiro - IEETA
email: skl@det.ua.pt, iouliia@det.ua.pt

ABSTRACT

The paper analyses and compares alternative iterative and recursive implementations of FPGA circuits for various problems. Two types of recursive calls have been examined, namely for cyclic and binary (N-ary) search algorithms. The details of experiments are presented for four different design problems. The relevant comparative data have been obtained as a result of synthesis and implementation in FPGAs of the respective circuits from system-level (Handel-C) and RTL (VHDL) specifications.

1. INTRODUCTION

It is known that many computational algorithms can be implemented using various techniques based on either iterative or recursive specifications. Both types of specifications are very powerful and it is very important to identify some criteria to enable the best type of implementation for any particular problem to be chosen. For example, experience in software development shows that recursion is not always appropriate, particularly when a clear efficient iterative solution exists [1]. On the other hand, some results reported in [2] demonstrate that recursion can be implemented in hardware much more efficiently than in software. This is because any recursive call/return can be combined with the execution of other operations that are required by the respective algorithm. Even in software applications, applying recursive algorithms for various kinds of binary search (such as that can be executed over binary trees) is considered to be a notable exception where a recursive technique is more beneficial than an iterative approach [1].

Note that the majority of selection criteria for applying the alternative techniques considered above in hardware are based on intuitive and unsubstantiated assumptions and the only concrete results are to be found in the context of software development. This paper compares iterative and recursive implementations of similar algorithms in hardware, specifically in FPGAs. Two different levels of specification for such algorithms have been examined: hardware description languages (VHDL, in particular), and system-level specification languages (Handel-C, in particular). The three potential application areas that have been analyzed are: 1) combinatorial optimization; 2) sorting algorithms executing over binary trees; 3) computations that can easily be described using any of the techniques considered above.

The remainder of this paper is organized in four sections. Section 2 discusses some problems that can be solved using either iterative or recursive techniques. Section 3 describes feasible implementations of the techniques considered in hardware. Section 4 presents the implementation details and the results of experiments. The conclusion is in section 5.

2. USING ITERATIVE AND RECURSIVE TECHNIQUES FOR PROBLEM SOLVING

In general, two types of recursion can be examined. The first (trivial) type allows an iterative sequence of operations (i.e. a cycle) to be replaced by a recursive invocation of a procedure that executes each cycle iteration and tests for a condition for exiting from the cycle. Let us consider an example. Fig. 1,a depicts a binary matrix. Suppose we want to find a minimal row cover of the matrix, i.e. such a minimum number of rows that in conjunction have at least one value ‘1’ in each column [3]. The approximate algorithm [3] that allows this problem to be solved requires the following sequence of steps: 1) discovering the best row of the matrix on the basis of some criteria; 2) removing the row and all the columns that contain values ‘1’ in this row; 3) if the solution is not found and we cannot conclude that there is no solution then steps 1) and 2) are repeated for the reduced matrix. Fig. 1,b and 1,c demonstrate iterative and recursive algorithms respectively for the problem considered. Fig. 1,a shows all the steps of the algorithm (see steps 1,2,3) and the result (see the top matrix lines shown in bold font). Rows and columns that have to be removed at each step are indicated by dashed lines. The text in italic font shows masks for rows/columns, which permit a single storage to be used for the initial and all the intermediate matrices. As we can see

from fig. 1,c the recursive invocation of the module Z just enables us to provide a different way for exiting the cycle shown in fig. 1, b. Independently of the implementation (i.e. either in software or in hardware), recursive calls invoke operations over stacks in such a way that the states of the algorithm (where recursive invocations have happened) are saved onto a stack and the stack pointer is incremented to address the storage for a recursively called sub-algorithm. When the recursive sub-algorithm ends, the stack pointer is decremented in order to restore the states of the interrupted algorithm. In case of an iterative algorithm no stack is required. Computations are performed in a cycle, which ends as soon as some conditions are satisfied. From the point of view of functional capabilities (including error handling such as stack overflows or endless loops), iterative and recursive techniques are very similar. However, they are not equal in terms of the required resources and the execution time. Operations with stacks may consume an extra time. The depth of stack required for recursive calls can be large, which would need additional hardware resources. Thus, the first assumption is that replacing cycles with recursive invocations of a function that executes cycle iterations should increase the execution time and the hardware resources required. However, the stacks can easily be implemented in FPGA embedded memory blocks (and this will be demonstrated later in section 4), which have a very regular structure and occupy less space on a chip than irregular blocks built from CLBs with arbitrary interconnections. Thus, intuitive comparison criteria may lead to a wrong conclusion because many different factors have to be taken into account.

Let us analyze a simple cyclic algorithm that enables the maximum common divisor of two unsigned integers to be found. The following two C/C++ functions demonstrate feasible recursive (*RGCD*) and iterative (*IGCD*) implementations.

```
unsigned int RGCD(unsigned int A,
                  unsigned int B)
{   if (B > A) return RGCD(B,A);
    else if (B<=0) return A;
         else return RGCD(B,A%B);   }

unsigned int IGCD(unsigned int A,
                  unsigned int B)
{   unsigned int tmp;
    while (B > 0)
    {     if(B > A)
            { tmp=A; A=B; B=tmp; }
          else
            { tmp=B; B= A%B; A=tmp; }
    }
   return A;
}
```

These functions have also been implemented in Handel-C and in VHDL. Thus, we can compare software and hardware versions.

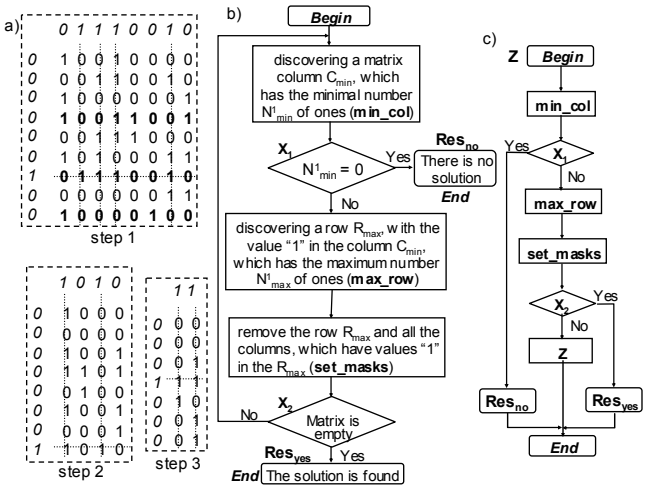

Fig. 1. Steps required for the approximate matrix covering algorithm (a); iterative (b); and recursive (c) implementations of the algorithm

Experiments with the C/C++ functions show better performance for the iterative algorithm by a factor of about 1.5. Experiments with Handel-C/VHDL projects also exhibit advantages of the iterative implementation in terms of both FPGA resources and the execution time (a factor of 1.03/(1.01-1.15) and 0.99/(1.16-1.2), respectively). But, recursive specifications might have other advantages such as clarity of the description, ease of testing (debugging), change and maintenance, etc. Besides, if a hierarchical modular specification [4] is used then recursion does not consume additional resources. This will be shown in section 4. The details of Handel-C and VHDL projects will be given in the next two sections.

The results of analysis of the example considered lead to the following conclusion. Although for the first (cyclic) type of recursion we can expect that iterative solutions providing equal functionality require less hardware resources and attain better execution time, distinctive features of any particular algorithm should be examined in order to choose the best technique. The criteria for such a choice are not the same for different types and levels of languages (i.e. Handel-C and VHDL, etc.).

The second type of recursion is a binary (N-ary, N>2) search, such as that executed over a binary (N-ary) tree. In this particular case, the sequence of recursive invocations also makes it possible to keep track between sub-trees that are examined during forward propagation steps. We assume that the uppermost part of the tree is its root and the bottom part contains leaves. Thus, any sub-sequence of returns from recursive procedures easily enables us to encounter any intermediate point on the way from the upper part to the bottom part of the tree. As a result, we could expect that recursive algorithms of this type would be less complicated and would attain better execution time

than the other algorithms. This is because the latter have to provide additional storage and operations for traversing the tree that make it possible to return to any node along the forward propagation path and to recognize from which branch (for example in case of a binary tree either from the left or from the right branch) the return to the node is being done.

Let us look at an example. Fig. 2,a depicts a binary tree that stores ordered data. Any node of the tree contains four fields: a pointer to the left child node, a pointer to the right child node, a counter, and a value (for example, an integer). The rules for constructing this tree are given in [2,4,5]. To build such a tree for a given set of values, we have to find the appropriate place in the current tree for each incoming integer. In order to sort the data, we can apply a special technique [5] using forward and backward propagation steps that are exactly the same for each node. Thus, a recursive procedure is efficient and the respective algorithm z_2 [2] is shown in fig. 2,b. After a conditional node this algorithm activates itself recursively for any left pointer or terminates when the current pointer is null. After returning (backtracking) from the left sub-tree the algorithm extracts (displays) data from the current node and tests the right pointer using a similar recursive invocation (see fig. 2,b). When the return occurs from the module invoked for the right pointer no operation is executed. Since we can distinguish the returns from the left and right sub-trees, using recursive invocations provides much more algorithmic functionality than just testing some conditions considered at the beginning of this section. The algorithm in fig. 2,b ends as soon as the whole tree is traversed.

The main advantage of the algorithm z_2 is its simplicity and clarity. However, z_2 is not fast at all, mainly due to two unnecessary recursive calls for any leaf of the tree. Indeed, since an absence of a bottom sub-tree is checked in the conditional node at the beginning of z_2, a recursive call has to be done for the left pointer of a leaf (which is null) and for the right pointer of a leaf (which is also null). Consequently two additional returns from z_2 have to be executed for each leaf. Fig. 3,a depicts a modified algorithm z_2 (denoted z^m_2), which does not have this drawback. Fig. 3,b shows the respectively modified algorithm z_1 from [2] (designated z^m_1) and fig. 3,c presents the module z_0, from [2], which calls z^m_1 and z^m_2. The module z^m_1 receives incoming integers and constructs the tree (such as that shown in fig. 2, a). The module z^m_2 sorts data extracted from the tree. Each invocation of z^m_1 enables us to add a new integer to the tree from the input sequence of integers (such as that shown at the bottom of fig. 2,a). The tree is saved in RAM with the following fields: the integer received; a pointer to the previous node; a pointer to the right sub-tree; a pointer to the left sub-tree. Any pointer to an empty sub-tree contains a value that is all 1's, i.e. 1…11.

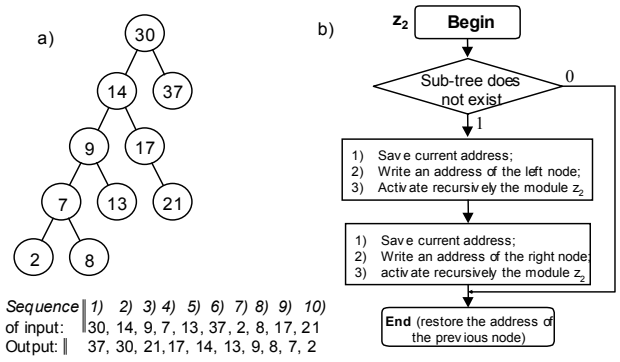

Fig. 2. Binary tree for data sorting (a) and a recursive algorithm of sorting (b)

The condition in z_0 permits z^m_1 to be called for all integers and finally to invoke the module z^m_2 to sort the data. Since the modules z_0, z^m_1, z^m_2 will be used for comparisons with iterative algorithms, all their micro-operations ($y_1,\ldots,y_{10}$) and logical conditions ($x_1,\ldots,x_7$) are described below:

x_1=1 if there was no left tree node in the previous call of z^m_1 and x_1=0 if there was no right tree node in the previous call of z^m_1 (for the root x_1 can get any value);

x_2=1 if incoming value has already been added to the tree (in this case just a counter for the respective node has to be incremented), else x_2=0 (for brevity in the lines that follow, the *else* statement is not shown);

x_3=1 if a new node has to be added to the tree;

x_4=1 if an incoming value is less than the value in the current node;

x_5=1 if new incoming values are available;

x_6=1 if a left sub-tree exists for the current node;

x_7=1 if a right sub-tree exists for the current node;

y_1=1 – store the address of the current node in the signal *previous address*;

y_2=1 – set the address of the left tree node;

y_3=1 – send the current (sorted) value to the output;

y_4=1 – set the address of the right tree node;

y_5=1 – restore the address of the previous tree node from the respective field of RAM for the current node;

y_6=1 – write the address of the new node to the left pointer of the previous node;

y_7=1 – write the address of the new node to the right pointer of the previous node;

y_8=1 – write the incoming value to the new node; write null values to the left and right pointers of the new node; write the address of the previous node to the new node;

y_9=1 – increment the counter for the current node;

y_{10}=1 – set the address of the root tree node, which enables a new incoming value to be added.

Fig. 4 shows a non-recursive (iterative) algorithm for sorting, which provides the same functionality as the algorithm in fig. 3.

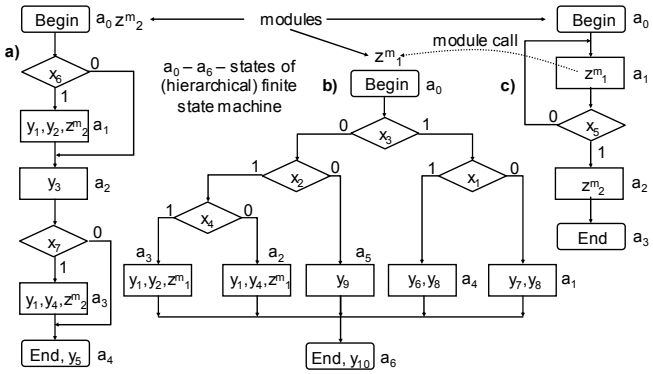

Fig. 3. Recursive modules for experiments

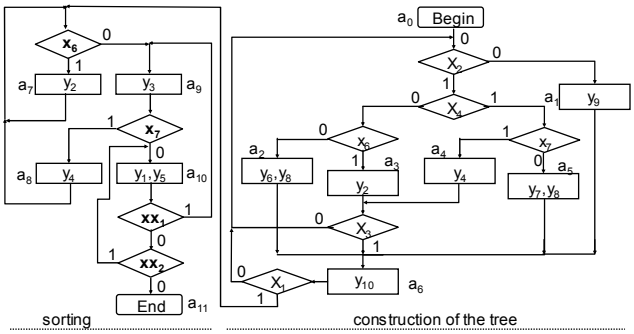

Fig. 4. Iterative non-modular algorithm

Fig. 4 has the following two additional logic conditions: xx_1=1 if the path to the current node is via the left pointer of the previous node; xx_2=1 if there is a previous node, i.e. the current node is not the root node.

Comparing fig. 3 and fig. 4 shows that the modular specification (see fig. 3) is easier to understand and it can be verified, modified and reused more easily.

3. IMPLEMENTATION OF ITERATIVE AND RECURSIVE ALGORITHMS IN HARDWARE

This section provides just very brief details about the specification, synthesis, and implementation issues for recursive and iterative algorithms that are required to understand which types of circuits have been examined. It is known that recursive algorithms can be implemented with the aid of hierarchical finite state machines (HFSM) [4]. Using HFSMs enables us to provide recursive calls in any language and we will demonstrate this possibility on some examples for discovering the greatest common divisor of two unsigned integers (see also section 2). Fig. 5,a shows the specification of the recursive algorithm. The module z_1 describes the operation A%B, i.e. the residue of division (A/B). The implementation of this operation (like any other similar operation) has been chosen to be exactly the same for iterative and recursive algorithms.

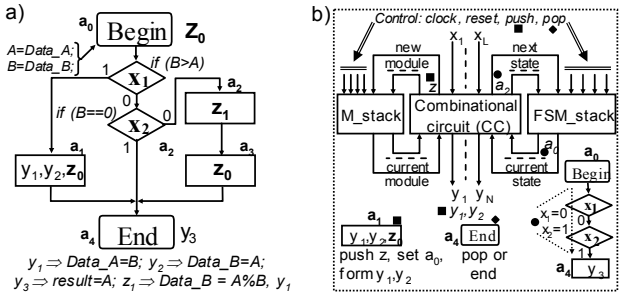

Fig. 5. Specification of the function RGCD (a) and its implementation in hardware (b)

An HFSM (see Fig. 5,b) permits execution of the algorithm in fig. 5,a (or any similar algorithm) and contains two stacks: one for states (the *FSM_stack*) and the other for modules (the *M_stack*). The stacks are managed by a *combinational circuit* (CC) that is responsible for new module invocations and state transitions in any active module that is designated by outputs of the *M_stack*. Any non-hierarchical transition is performed through a change of a state code only on the top register of the *FSM_stack* (see the example marked with •). Any hierarchical call alters the states of the both stacks in such a way that the *M_stack* will store the code for the new module and the *FSM_stack* will be set to the initial state of the module (see the example marked with ■). Any hierarchical return just activates a pop operation without any change in the stacks (see the example marked with ♦). The stack pointer *stack_ptr* is common to both stacks. If the *End* node is reached when *stack_ptr*=0, the algorithm terminates execution. The synthesis method for HFSMs has been considered in detail in [4].

Reusable VHDL specifications for the *M_stack* and the *FSM_stack* are given in [2]. CC has been constructed on the basis of the VHDL template available in [2] for which two levels of transitions are executed: a) transitions between the modules (z_0 and z_1 for our example); and b) transitions between the states within each module (a_0,…,a_4 for our example; the module z_1 has just 1 state a_0). Similarly, HFSMs can be described in Handel-C [6].

The iterative algorithm (such as that shown in fig. 4) is implemented through the design of a finite state machine (FSM) described in VHDL. For example, the FSM for fig. 4 has 12 states (a_0-a_{11}). A synthesizable VHDL specification of the FSM permits the proper FPGA circuits to be built. Handel-C code for the iterative implementation has been derived directly from the respective C language specification. The VHDL projects use identical descriptions of the required operations (such as data swapping, A%B, etc.) for the iterative and recursive algorithms. In Handel-C code, C/C++ operations (such as A%B) have been applied directly because Handel-C specifications can be taken from high-level programming language specifications with minimal changes. Thus, the Handel-C synthesizer takes responsibility for final implementations.

4. IMPLEMENTATION DETAILS AND THE RESULTS OF EXPERIMENTS

Experiments have been carried out for the following problems: P_1 - sorting based on a binary tree, such as that shown in fig. 2, a; P_2 - approximate method for discovering a minimal row cover of a binary matrix (see fig. 1); P_3 - exact binary search algorithm for solving the knapsack problem; P_4 - computation of the greatest common divisor.

The synthesis and implementation of circuits from the specification in VHDL (see table 1) were done in Xilinx ISE 7.1 for xc2s400e-6ft256 FPGA (Spartan-IIE family of Xilinx) available on the prototyping board TE-XC2Se [7]. Synthesis from the specification in Handel-C was done in Celoxica DK3 design suite and implementation was finalized in Xilinx ISE 7.1 for xc2s200-5fg456 FPGA (Spartan-II family of Xilinx) available on the prototyping board RC100 [8] (the projects P_3, P_4 in table 2) and for xc2v1000-4fg456 FPGA (Virtex-II family of Xilinx) available on the prototyping board RC200 [8] (the projects P_1, P_2 in table 2). Note that each circuit includes not only components that are needed for comparison, but also auxiliary blocks for entering input data and visualizing the results on an LCD monitor/panel. For each particular problem described in the same language (i.e. in either VHDL or Handel-C) all the auxiliary components are exactly the same for the iterative and recursive implementations. So, the difference in resources and in execution time provides correct data for a comparison.

The results of experiments are presented in tables 1, 2. Rows/columns of the tables contain the following data: MI – *M*odular *I*terative implementation; BM/DM - using *B*lock *M*emory/*D*istributed *M*emory; N_s – the number of FPGA slices; F – the maximum allowed clock frequency (in MHz) calculated by the implementation tools; N_{clock} – the number of clock cycles required for solving a given problem instance; ET – execution time in *ns* using the frequency F.

The size of the binary matrices for the covering problem P_2 was chosen to be *128 rows * 128 columns*. In total, 9 different randomly generated problem instances were tested but they gave nearly the same results. That is why just one of these results is shown in table 2. The problem P_4 was tested for many pairs of numbers and data in the table are given for unsigned integers 189 and 135 (the result is 27). Although different numbers produce results that are not equal, the ratio between parameters in the table for recursive and iterative algorithms is nearly the same. A similar approach was used for the problem P_1 and data in tables 1, 2 are given for the sequence shown in fig. 2,a. Data for the problem P_3 are given for 5 objects, which have to be packed in a knapsack.

The projects provide generic parameters for initial data and can therefore be customized. Cyclic/tree-based algorithms are marked with *(c)/(t)*, respectively.

Table 1. The results of experiments (VHDL projects)

Problem/ MI/BM/DM	N_s/F/N_{clock}/ET	
	Recursive	Iterative
P1 *(t)*		443/35.1/70/1994
P1/MI *(t)*	623/74.8/72/963	599/76.0/87/1145
P1/MI/BM *(t)*	474/52.2/72/1379	473/70.2/87/1239
P1/MI/DM *(t)*	477/58.8/72/1224	
P3 *(t)*		153/59.9/88/1469
P3/MI *(t)*	165/37.3/62/1662	
P3/MI/BM *(t)*	149/40.3/62/1538	
P3/MI/DM *(t)*	150/43.1/62/1438	
P4 *(c)*		448/41.3/9/217
P4/MI *(c)*	515/42/11/261	
P4/MI/BM *(c)*	454/43.5/11/252	
P4/MI/DM *(c)*	454/42.4/11/259	

Table 2. The results of experiments (Handel-C projects)

Problem	N_s/F/N_{clock}/ET	
	Recursive	Iterative
P1 *(t)*	1293/37.3/73/1953	750/45.5/61/1340
P2 *(c)*	5118/25.1/18270/ $7.28*10^6$	5118/25.1/182688/ $7.28*10^6$
P3 *(t)*	624/31.5/265/8407	228/36.7/474/12915
P4 *(c)*	242/16.4/6/365	234/16.3/6/367

Modular iterative implementations MI are based on HFSM (see fig. 5,b) for executing control sequences (such as shown in fig. 4) but these sequences are decomposed on modules (recursive calls in the modules are prohibited). For example, the algorithm for the problem P_1 has been decomposed in the next three modules: the main module (which is the same as in fig. 3); the module for binary tree construction (see the right hand part of fig. 4); and the module for sorting (see the left hand part of fig. 4). Block memory (namely one ISE 7.1 template component RAMB4_S8_S8 for device primitive instantiation) has been used for constructing the blocks *M_stack* and *FSM_stack* (see fig. 5,b) in implementations marked with BM. Distributed memory (namely a set of ISE 7.1 components RAM16X1S) has been used for constructing the blocks *M_stack* and *FSM_stack* in implementations marked with DM (see table 1). Note, that iterative and iterative modular implementations for the problem P_1 are based on slightly different FSM behaviors. That is why the relevant values of N_{clock} are also different.

Note that comparison of Handel-C and VHDL projects for similar problems is not a target of the paper. The auxiliary circuits for VHDL and Handel-C projects are very different. Thus, just the comparison of recursive-iterative algorithms for the same specification tool can be meaningful. Due to space limitations, some implementation issues of the algorithms in fig. 3-5 used for experiments have not been described in detail, but they permit execution time and FPGA resources to be minimized.

We can summarize the results of experiments and analysis of alternative specifications as follows:

1. Using recursive algorithms of the first type (see section 2 and the rows of tables 1, 2 marked with *c*) that replace conditions for exiting from cycles with recursive returns increases the execution time and the hardware resources required. Thus, this kind of recursion can be recommended only when it permits a clearer specification to be provided and if it might benefit from a hierarchical specification (see point 3 below for additional details). For the majority of practical applications the respective iterative implementations are better.

2. It seems to be more advantageous to use modular algorithms in general, and recursive algorithms in particular, for problems of binary (N-ary) search compared to iterative implementations. Indeed, although the non-modular iterative implementation for P_1 uses the smallest number of FPGA slices, the execution time is the worst. The recursive implementation MI/DM of P_3 is the best in all aspects. Other records of table 1 show that modular algorithms mostly provide better execution time. For example, for the problem P_1 the best execution time t_{et} = 963 *ns* is achieved by the recursive algorithm (marked with P1/MI), whereas the non-modular iterative algorithm (marked with P1) that relies on a traditional FSM is the slowest (t_{et} = 1994 *ns*). Note, that solving a similar problem in software reveals slightly better performance for the iterative algorithm. From examining the complexity of the resulting circuits we conclude that using embedded memory blocks enables the FPGA resources that are occupied to be significantly reduced. Thus, for recursive and iterative implementations the number of FPGA slices becomes practically the same. Note, that just 4% of memory available for the component RAMB4_S8_S8 is used for solving the problem P_1 and the same component enables much more complicated algorithms to be realized practically without increasing the resources. Additional FPGA slices are needed just for a counter keeping a stack pointer. Thus, we could expect that for very complex algorithms modular implementations would be significantly better. Besides, a highly integrated stack memory for HFSMs might potentially be implemented as an embedded block in future generations of FPGAs and this would attract additional attention to recursive and hierarchical implementations.

3. There are some potential applications where we can benefit from the strategy *divide and conquer*, i.e. additional circuit complexity is not as important as clarity of the algorithm, and this can be achieved through a hierarchical specification. For example, in our opinion the algorithm in fig. 3 is more readable and understandable than the specification in fig. 4. This is a subjective judgment but sometimes this may influence the choice for a particular designer. It is also important that hierarchical modular specifications provide direct support for reusability. Indeed, the same module can be reused in different algorithms and even within the same algorithm, which may lead to a reduction in the design time and even reduced hardware resources for some cases. Obviously, if a hierarchy is done on the basis of HFSM, recursive calls can also be provided without any additional hardware.

ACKNOWLEDGEMENTS

The authors would like to acknowledge Ivor Horton for his very useful comments and suggestions.

5. CONCLUSION

The main contribution of the paper is the presentation of criteria that allow either recursive or iterative implementations to be chosen for a particular problem. These criteria have been obtained from detailed analysis and numerous experiments provided for different design problems. The experiments have been carried out both in software and in hardware and in the last case FPGA implementations have been constructed from system- and RTL-level specifications. Note, that due to space limitations, many previously published results have been referenced without including details here. However all the relevant publications are available online from [9].

6. REFERENCES

[1] F. M. Carrano, "Data Abstraction and Problem Solving with C++," The Benjamin/Cumming Publishing Company, Inc. 1995.

[2] V. Sklyarov, "FPGA-based implementation of recursive algorithms," *Microprocessors and Microsystems. Special Issue on FPGAs: Applications and Designs*, vol. 28/5-6, pp. 197–211, 2004.

[3] A. D. Zakrevskij, "Logical Synthesis of Cascade Networks," Moscow: Science, 1981.

[4] V. Sklyarov, "Hierarchical Finite-State Machines and Their Use for Digital Control," *IEEE Trans. on VLSI Systems*, vol. 7, no. 2, pp. 222–228, 1999.

[5] B. W. Kernighan, D. M. Ritchie, "The C Programming Language," Prentice Hall, 1988.

[6] V. Sklyarov, "Hardware Implementation of Hierarchical FSMs," in *Proc. of WISICT, Workshop on Design and Applications of Reconfigurable Systems*, 2005, pp. 148–153.

[7] Spartan-IIE Development Platform. [Online]. Available: www.trenz-electronic.de.

[8] Celoxica products. [Online]. Available: http://www.celoxica.com.

[9] Additional publications. [Online]. Available: www.ieeta.pt/~iouliia/, www.ieeta.pt/~skl/.