ELECTRONICS AND ELECTRICAL ENGINEERING

ISSN 1392 – 1215 *2011. No. 7(113)*

ELEKTRONIKA IR ELEKTROTECHNIKA

SYSTEM ENGINEERING, COMPUTER TECHNOLOGY

T 120

SISTEMŲ INŽINERIJA, KOMPIUTERINĖS TECHNOLOGIJOS

Acceleration of Recursive Data Sorting over Tree-based Structures

D. Mihhailov, A. Sudnitson

Department of Computer Engineering, Tallinn University of Technology,
Raja 15, 12618 Tallinn, Estonia, phone: +372 5092356, e-mail: alsu@cc.ttu.ee

V. Sklyarov, I. Skliarova

Department of Electronics, Telecommunications and Informatics, University of Aveiro,
3810-193 Aveiro, Portugal, phone: +351 234401539, e-mail: skl@ua.pt

crossref http://dx.doi.org/10.5755/j01.eee.113.7.612

Introduction

The main objective of this paper is to evaluate and improve FPGA-based digital circuits, which implement recursive specifications. Using and taking advantage of application-specific circuits in general and FPGA-based accelerators in particular have a long tradition in data processing and for solving problems with high computational complexity. FPGA enables faster, less time consuming hardware implementation and finds a lot of practical applications in various fields of digital design. For example, in [1] an FPGA device was successfully used to accelerate performance of word recognition system compared to its software counterpart.

Recursion is a powerful problem-solving technique [2]. It may be applied to problems that are decomposable into smaller sub-problems of exactly the same form as the original. The advantages and disadvantages of recursive techniques in software are well known [3]. It has been shown [4] that recursion can be implemented in hardware more efficiently. This is mainly because any recursive call/return can be combined with the execution of operations that are required by the respective algorithm. Thereby, the number of states needed for the execution of recursion in hardware can be further reduced compared with software analogue. Besides, these states are accumulated on stacks that can be easily constructed on built-in memory blocks of field-programmable gate array (FPGA) devices, which are relatively cheap. The results obtained with some known methods for implementing recursive calls in hardware, reviewed in [5], have shown that many hardware circuits are faster than software programs executing on general-purpose processors.

This paper concentrates on further improvements of digital circuits, which implement recursive sorting algorithms. The proposed improvements include both algorithmic and architectural optimization techniques.

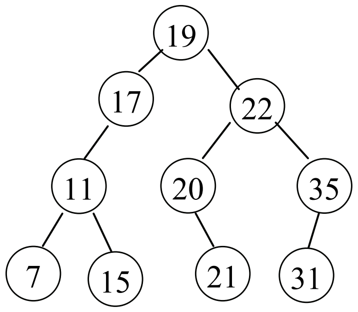
The remainder of this paper is organized in six sections. Section 2 describes sequential methods for processing of binary trees with recursive sorting algorithms, which target FPGA-based implementation. Section 3 presents possibilities for parallel execution of the same task. Section 4 is devoted to binary tree compression technique. Section 5 introduces hardware model, which allows to implement proposed methods. Section 6 analyzes experimental results, compares known and proposed implementations, and suggests some further improvements. The conclusion is given in Section 7.

Recursive sorting of binary trees

Recursive algorithms are most often employed for various kinds of binary search. Consider an example of using a binary tree for sorting data [2]. Suppose each node of the tree contain three fields: a value (e.g. an integer), a pointer to the left child node (LA), and a pointer to the right child node (RA). The nodes are maintained in such a way that for any node the left sub-tree only contains values, which are less than the value of that node. Thus, the right sub-tree would contain only values that are greater. The absence of a node is indicated by a specially allocated code. Such a tree can be easily built and traversed recursively. Sorting of this type will be considered in this paper as a case study to demonstrate the proposed methods and their advantages.

An example binary tree is presented on Fig. 1, a. For each node only its data value is shown. The same tree is shown on Fig. 1, b, but this time as it is stored in a memory. Each row corresponds to a node. The first column specifies memory location, where the node is stored. The other columns represent the node itself according to the above-mentioned format. The actual width of each entry depends on the width of individual field (i.e. Data+LA+RA). The data is stored in the same order as it is supplied to the circuit, which means the root is always stored at zero address. Therefore, all-zero code can be safely used to indicate the absence of a node, as other nodes cannot point to the root.

The known method for recursive data sorting [5] (S1) was chosen as a base for comparison. The known algorithm can be improved in hardware through the use of dual-port memories (available within many FPGAs) and algorithmic modifications. Suppose the currently processed node is saved in a buffer register. Then, embedded dual-port memory blocks permit simultaneous access to the left and the right child nodes through LA and RA fields of the buffer register. Analysis of child nodes and their connectivity allows to cover a larger portion of binary tree during traversal.



a)

#	Dat	L	R
0	19	2	1
1	22	3	4
2	17	5	0
3	20	0	6
4	35	7	0
5	11	9	8
6	21	0	0
7	31	0	0
8	15	0	0
9	7	0	0

b)

Fig. 1. Example binary tree: a – visual representation; b – as stored in memory

Consider the same example tree, but this time it is stored in a dual-port memory (Fig. 2). The binary tree is in the middle of the traversal process and node "22" (in a bold circle) is currently being processed. It has been loaded into buffer register and both pointer fields of this node are fed into address inputs of dual-port memory. Each output word stores similar information to the buffer register (i.e. Data+LA+RA) for the left and for the right nodes. Therefore, at each recursive step up to three nodes can be processed at the same time. On Fig. 2 simultaneously processed nodes are enclosed in a dashed circle.

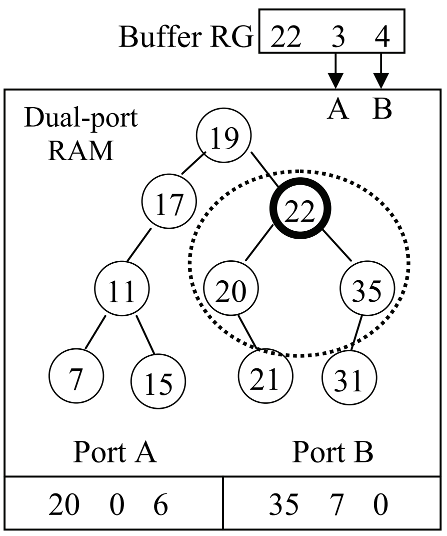


Fig. 2. Example binary tree stored in a dual-port memory

Connectivity of child nodes can be analyzed to reduce the number of recursive calls/returns during traversal compared to the known method. The first improvement considers general connectivity of the child node (does it have any child nodes at all). If the left child node has no child nodes of its own, then its value can be directed to the output, followed by the value of the currently processed node (in case of ascending sorting). Thus, there is no need to call the algorithm to process the left child node. The same applies to the right child node as well.

This approach can be further improved by examining pointers of child nodes individually as well. Consider the situation presented on Fig. 2. The left child node "20" does not have a left child node of its own. Therefore, its value can be directed to the output. As the left child node "20" has already been processed and its right pointer is not zero, the algorithm can be called for the node "24" next. This improved version of the known method will be further referred to as S2.

Parallel sorting architectures

Another potential improvement in performance for the known method can be achieved with introduction of parallelism. Neither known method nor its improved versions permit parallel processing of different branches of the tree and are, therefore, purely sequential. However, it may be possible to put multiple instances of the same algorithm to work on different parts on the same tree. The most obvious choice involves parallel traversal of the sub-trees, which are to the left and to the right of the root.

First of all, consider the known method for recursive data sorting. There are two simultaneously functioning digital circuits that are a main sorter and a secondary sorter (Fig. 3). The tree can be built in a dual-port memory, which would allow simultaneous access for both devices. The main sorter builds the tree, outputs the left sub-tree and the root, and activates the secondary sorter when necessary. The secondary sorter outputs the right sub-tree only. Obviously, more parallel branches can be introduced using cascade structures. This parallel architecture will be further referred to as S3.

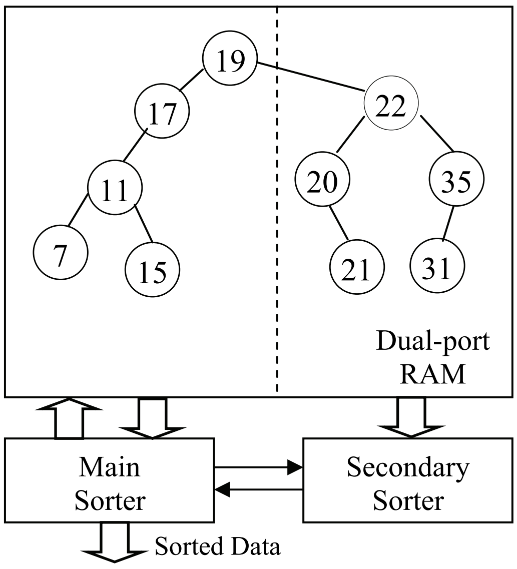


Fig. 3. Parallel architecture S3

However, there is one significant limitation. Although the tree is processed in parallel, the results cannot be output in parallel. All nodes of the left sub-tree have smaller value compared to any node of the right sub-tree. Therefore, temporary storage memory is also required for the right sorter. As soon as the traversal of the left sub-tree is compete, the processed nodes of the right sub-tree can be read from the temporary storage. Intuitively one can guess that the end result would depend considerably on the balance between the left and the right sub-trees of the root. If the tree is completely unbalanced one sorter unit would need significantly more time for data processing than the other. This may completely nullify the advantage of parallel processing compared to its sequential analogue.

Such dependency is most certainly undesirable. In order to eliminate it the Top Level Manager (TLM) is introduced. The circuit distributes input data in such a way that the first item is supplied to the first sorter, the second – to the second sorter, the third – to the third sorter, the fourth – to the first sorter, and so on (if the number of sorting units is three). The process is repeated until all data items are distributed. Thereby, each sorter unit constructs and traverses its own independent tree (Fig. 4). However, now every sorter would require a dual-port output memory to serve as a buffer for its sorted sequence. This parallel architecture will be further referred to as S4.

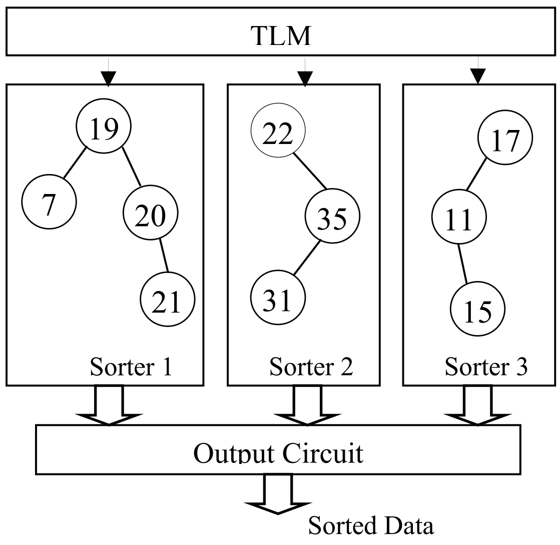


Fig. 4. Parallel architecture S4

As soon as the trees are constructed, the TLM instructs the output circuit to generate the sequence of sorted data. Data items are saved in the output buffer using the first port. Data output is executed in parallel with tree traversal using the second port of the output memories. The output circuit is responsible for merging output sequences. At the beginning, all output buffers are empty. When each buffer contains at least one unprocessed data item, the smallest one (or the greatest one, depending on the sorting strategy) is extracted.

Alternatively, each sorter may be stalled when a new unprocessed data item is found. It waits until the data is read by the output circuit and then continues with tree traversal. This approach eliminates the need for temporary storage. However, the performance is greatly reduced.

Binary tree compression

This paper also suggests compression method for tree-like structures in order to accelerate data processing and reduce memory consumption. The basic idea is the following. Consider it is required to sort m-bit data items. The known method is applied for sorting (m-k) most significant bits. The remaining k bits are encoded for each node using additional “data within group” field, which is 2^k wide. Each bit corresponds to a certain k-bit combination. It is set to “1” if the matching data item has been added to the group. It remains “0”, otherwise.

In this case, the known method will sort up to 2^{m-k} groups of data items. When group is selected, up to 2^k data items within each group can be generated by decoding “data within group” field. Then each clock cycle one generated data item from the selected group can be propagated to the output. Potentially, it is also possible to output data from the selected group and search for the next one in parallel.

A compressed version of the example binary tree from Fig. 1, a is presented on Fig. 5. For the example tree m=6 and k=2. The additional “data within group” field is marked next to each node. The most significant (rightmost) bit of this vector corresponds to binary combination “11”, the next bit – to “10”, etc. The number of “1”s in this vector corresponds to the number of data items each group holds. For example, node “5” holds three data items: 010110(22), 010110(21) and 010100(20), where the decimal value of the relevant binary code is shown in parenthesis.

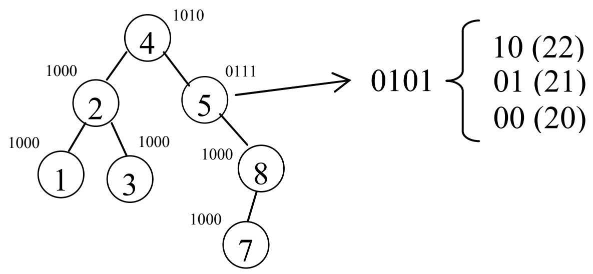


Fig. 5. Compressed binary tree

Note, that compared to the original binary tree from Fig. 1, a, which has ten nodes, the compressed binary tree on Fig. 5 has only seven. It should be also mentioned, that as a side effect the construction of the compressed binary tree itself is also accelerated when compared to the original. This is due to the fact that addition of a data item to the existing group can be done a lot faster, then finding a new place for it in the tree each time.

Hardware implementation

The main feature of any recursive algorithm is the capability to activate (to call) itself. Different methods for implementation of recursive algorithms in hardware are considered in [4,5,6]. The model of a hierarchical finite state machine (HFSM) [7] was chosen to provide recursion support. The choice is mainly based on the primary objective to implement such circuits using FPGA-based prototyping boards.

There are many synthesizable specifications for recursive algorithms and hierarchical graph-schemes (HGS) is one of them. Such specification is more readable and provides direct support for reusability. Furthermore, this model allows to efficiently implement recursion in hardware (FPGA device, in particular). A HGS can be easily converted to a HFSM and then formally coded in a hardware description language (e.g. VHDL). The coding is done using the VHDL templates proposed in [7], which are customizable for the any set of HGSs. The resulting VHDL code is synthesizable and permits the hardware to be designed in commercially available CAD systems

The known model of HFSM has the following distinctive features. The current state of the system is defined with active module and its state. Therefore, states in different modules can be assigned the same labels (the same codes). Any non-hierarchical transition is performed through a change of a state code only (just like in conventional finite state machine). However, hierarchical transition would alter both module code and state code.

There are two stack memories for storing modules and states. In case of hierarchical call, the state of the control unit (active module and its current state) is pushed into these stacks. When the execution flow of module is terminated, the HFSM performs hierarchical return (the state of control unit is restored from the stacks). Both, hierarchical calls and hierarchical returns, can be combined with the execution of micro-operations. This known model will be further referred to as HFSM with explicit modules (Fig. 6).

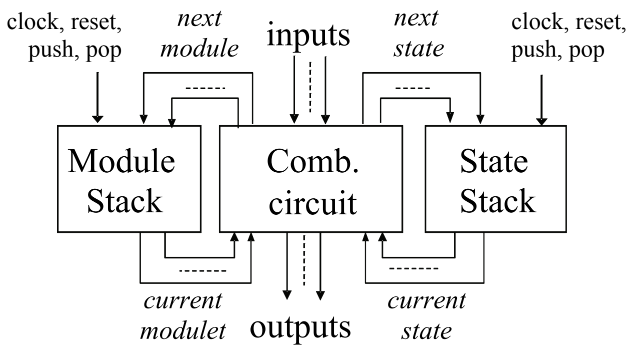


Fig. 6. Hierarchical Finite State Machine with explicit modules

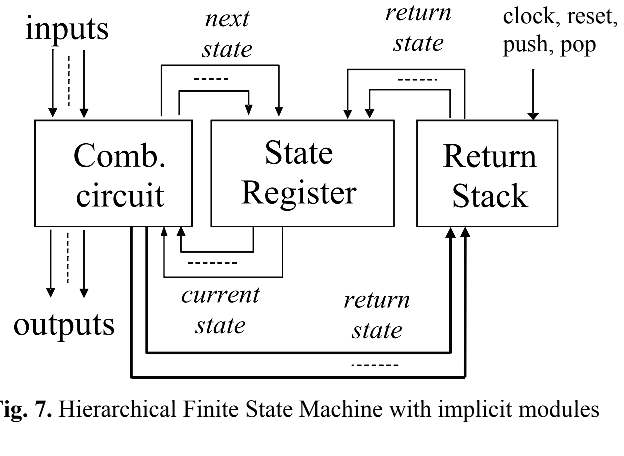


Fig. 7. Hierarchical Finite State Machine with implicit modules

A new proposed HFSM model is called HFSM with implicit modules (Fig. 7). It essentially behaves like a conventional finite state machine (FSM). It has a state register and a single stack of states. In this case each state has to be assigned a different label (code). A major advantage of the new model is that it is directly applicable to all known optimization techniques that have been proposed for conventional FSMs (like state encoding).

The width of a stack entry can be also minimized, as the number of return states is limited. When a state code is pushed into stack, it can be encoded with a smaller code (compared to the length of the original state code). Similarly, during hierarchical return the content of stack is decoded before being placed into state register. This is especially useful, as this allows to implement the optimized stack using DistributedRAM feature of FPGA device and conserve block memory.

Experimental results

Circuits that implement proposed methods and architectures were described in VHDL. The synthesis and implementation of these circuits were done in Xilinx [9] ISE 11 for FPGA Spartan3E 1200E-FG320 of Xilinx available on the prototyping board NEXYS2 from Digilent [10]. A pseudo-random-number generator produces data items with a length of 16 bits. These data items are sorted by the methods and are implemented using hardware models described in the previous sections.

The sorting time for the know and proposed methods in clock cycles is presented in Table 1. The *Number of Data Items* column shows the total number of data items that is sorted. An additional column Balance (Left/Right) shows the number of nodes in the left and right sub-trees from the root. It is needed to examine the dependency of methods S1, S2 and S3 on the balance between the left and right sub-trees (not valid for other methods, as they do not use the original binary tree for sorting data). Columns *S1*, *S2*, *S3* show results for the corresponding methods. Column $S4_{N2}$ present results for parallel architecture S4 with two instances of the known method S1 working in parallel ($S4_{N4}$ - with four instances of S1). Column $S1_{G4}$ shows results for a compressed binary tree (parameters are as follow: m=16 and k=4), which is sorted using known method S1.

The performance of the known method (S1) is the worse. The average number of clock cycles per data item is 4. The sequentially improved method S2, which is based on the use of dual port memories, provides a better result with the average number of clock cycles of 2.8 per data item. Sorting of the compressed tree using known method S1 comes down to 2.3 clock cycles per data item. Not surprisingly, parallel method S3 give the best performance if the tree balance is good (up to 2.5 clock cycles per data item). However, sorting of unbalanced trees reduces performance greatly (down to 4 clock cycles per data item). Processing of 2 independent binary trees using parallel architecture S4 improves performance to approximately 2 clock cycles per data item. Increasing the number of parallel sorters from 2 to 4 decreases the time of sorting to almost 1 clock cycle per data item.

Table 1. Sorting time (in clock cycles)

Number of Data Items	Balance (Left/ Right)	S1	S2	S3	S4$_{N2}$	S4$_{N4}$	S1$_{G4}$
1211	185/1025	4843	3373	5129	2474	1298	2830
1216	266/949	4863	3393	4749	2462	1296	2856
1248	332/915	4991	3486	4579	2522	1326	2921
1203	460/742	4811	3350	3714	2420	1290	2804
1228	528/699	4911	3432	3499	2490	1295	2901
1212	556/655	4847	3350	3279	2440	1291	2831
1230	623/606	4919	3470	3101	2479	1302	2915
1259	822/436	5035	3496	3727	2541	1329	2920
1230	799/430	4919	3419	3629	2507	1325	2834
1304	849/454	5215	3610	3853	2632	1414	2977
1276	963/312	5103	3564	4167	2573	1399	2931
1225	958/266	4899	3417	4101	2479	1312	2886
1225	986/238	4899	3420	4185	2479	1305	2889
1199	1051/147	4795	3319	4354	2432	1281	2824

The implementation results for circuits, which implement proposed methods, using Spartan3E 1200E-FG320 FPGA device are presented in Table 2. All methods have been implemented using the known model of HFSM with explicit modules (column *HFSM$_{explicit}$*). Methods S1 and S2 have also been implemented using the new optimized model of HFSM with implicit modules (column *HFSM$_{implicit}$*). It should be noted that there is no difference for these models in the number of clock cycles required for data sorting by the same algorithm, as this number depends only on the sorting algorithm itself. Subcolumn *F* presents the maximum attainable clock frequency in MHz, subcolumn *S* - the number of slices, subcolumn *L* - the number of LUTs and subcolumn *B* - the number of block RAMs.

Table 2. Implementation results

Method	HFSM$_{explicit}$				HFSM$_{implicit}$			
	F	S	L	B	F	S	L	B
S1	101	714	1391	5	109	355	708	5
S2	82	790	1548	6	89	435	855	6
S3	102	1115	2203	8	-	-	-	-
S4$_{N2}$	90	1297	2480	8	-	-	-	-
S4$_{N4}$	83	1707	3209	12	-	-	-	-
S1$_{G4}$	77	774	1512	4	-	-	-	-

For the original HFSM model with explicit modules the known method S1, being the simplest, requires the least FPGA resources and has one of the highest maximum clock frequency. As more complexity is introduced to improve the performance (sequential improvements, parallel architectures) to the known method, the resource consumption grows, while working clock frequency decreases. Note, that for sorting the compressed binary tree the number of required BRAMs is actually less than needed for sorting of the uncompressed binary tree. Also the decoding area overhead is not that big, but it reduces the maximum clock frequency quite a lot. Optimization of the generator circuit can certainly improve the overall performance.

The new optimized model of HFSM with implicit modules consumes almost two times less hardware resources and has a slightly higher clock frequency. This is mostly due to the fact that the return stack has been optimized and now requires significantly less FPGA resources to implement. Also, the new model is actually recognized as a regular Finite State Machine (FSM) by the design software and therefore can be subjected to various FSM optimization techniques [8].

The performance in clock cycles is a more theoretical measure, as it does not take into account the particular implementation device. In order to estimate the performance for a real hardware implementation, the clock cycle period (Table 2) must also be taken into account. The graph on Fig. 8 summarizes the performance of the proposed methods and parallel architectures for a Spartan3E 1200E-FG320 FPGA device. Horizontal axis lists the pseudo-randomly generated input sequences (the same as in Table 1), also providing the balance of the received binary trees. Vertical axis represents sorting time per data item in nanoseconds.

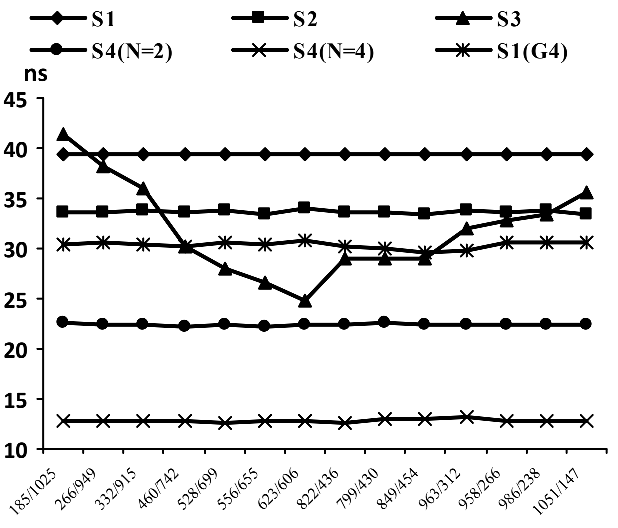


Fig. 8. Sorting time per data item (in ns)

The known method exibits a very steady performance, however, it falls short before other methods in terms of performance. Sequentially improved method S2 and sorting of the compressed binary tree provide approximately the same improvement with practically identical area overhead. However, one significant difference is that storage of a compressed binary tree requires less BRAM memory blocks. Therefore, this approach may be better suited for applications, which require processing of a larger data volumes. The dependency of the parallel arcitecture S3 on tree balance limits its practical usability, also for a well-balanced trees the performance improvement is quite significant. However, parallel architecture S4 for two instances of S1 delivers peak performance of S3 for any intput data sequence with basically the same area overhead. The best performance is archived by S4 with four instances of S1, but the number of required FPGA resources is also the highest. Therefore, this approach may be better suited for applications, which require fast processing of a smaller data volumes.

Conclusions

The paper suggests new hardware-oriented sequential and parallel implementations of recursive sorting algorithms. It demonstrates the advantages of the proposed innovations based on experimental results received from prototyping on FPGA development board NEXYS-2 from Digilent. Obviously, the results of the paper are not limited to just recursive sorting alone. They have a wider scope and can be applied effectively to numerous systems that implement recursive algorithms over tree-based data structures.

The experiments have been done for a relatively small number of data items (1200-1300). The employed FPGA device is able to handle around 4000 data items. The main restriction that limits the number of data items is the number of available embedded Block RAMs on the FPGA microchip, as they are used to store the binary tree. Thus, the number of data items can be significantly increased if we replace cheaper Spartan-3E family devices with mode advanced FPGAs such as that are available from Virtex-5 or Virtex-6 family. This would also increase the performance, as these devices are generally faster. The proposed tree compression technique can also allow to sort more data on the same FPGA device.

Acknowledgement

This research was supported by the European Union through the European Regional Development Fund and by the Estonian Science Foundation Grant No. 8478.

References

1. **Tamulevičius G., Arminas V., Ivanovas E., Navakauskas D.** Hardware Accelerated FPGA Implementation of Lithuanian Isolated Word Recognition System // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 3(99). – P. 57-62.
2. **Cormen T. H., Leiserson C. E., Rivest R. L., Stain C.** Introduction to Algorithms, 2nd ed. – MIT Press, 2002. -1312 p.
3. **Carrano F. M.** Data Abstraction and Problem Solving with C++: Walls and Mirrors. – Addison Wesley Publishing Company, Inc., 2004. - 992 p.
4. **Ninos S., Dollas A.** Modeling recursiondata structures for FPGA-based implementation // 18th International Conference FPL'08, 2008. – P. 11–16.
5. **Skliarova I., Sklyarov V.** Recursion in Reconfigurable Computing: a Survey of Implementation Approaches // 19th International Conference FPL'09, 2009. – P. 224–229.
6. **Maruyama T., Takagi M., Gishiro T.** Hardware implementation techniques for recursive calls and loops // 9th International Workshop FPL'99, 1999. – P.450-455.
7. **Sklyarov V.** Hierarchical Finite State Machines and their Use for Digital Control // IEEE Transactions on VLSI Systems. – 1999. – Vol. 7. - No. 2. – P. 222–228.
8. **Sudnitson A., Mihhailov D., Kruus M.** Project-Oriented Approach to Low-Power Topics in Advanced Digital Design Course // Electronics and Electrical Engineering. – 2010. – No. 6(102). – P. 151-154.
9. **Xilinx Inc.** FPGA Design Tools. Silicon Devices. Online: http://www.xilinx.com/
10. **Digilent Inc.** Nexys2 FPGA Educational Board. Online: http://www.digilentinc.com/.

Received 2011 02 15

D. Mihhailov, A. Sudnitson, V. Sklyarov, I. Skliarova. Acceleration of Recursive Data Sorting over Tree-based Structures // Electronics and Electrical Engineering. – Kaunas: Technologija, 2011. – No. 7(113). – P. 51–56.

The main objective of this paper is to evaluate and improve FPGA-based digital circuits, which implement recursive specifications. Recursive sorting algorithms over binary trees are considered as a case study to evaluate and demonstrate new techniques and their advantages. Since recursive calls are not directly supported by hardware description languages, they are implemented using the model of a hierarchical finite state machine (HFSM). The paper presents analysis and comparison of alternative and competitive techniques for describing recursive algorithms in hardware. The experimental results demonstrate that the proposed innovations allow to achieve better performance. Obviously, the results of this paper are not limited to recursive sorting alone. They have a wider scope and can be applied effectively to numerous systems that implement recursive algorithms over tree-based data structures. Ill. 8, bibl. 10, tabl. 2 (in English; abstracts in English and Lithuanian).

D. Mihhailov, A. Sudnitson, V. Sklyarov, I. Skliarova. Pasikartojančių duomenų rūšiavimo paspartinimas duomenų masyvuose // Elektronika ir elektrotechnika. – Kaunas: Technologija, 2011. – Nr. 7(113). – P. 51–56.

Analizuojamas FPGA struktūrų tobulinimas, siekiant gerinti pasikartojančių duomenų specifikacijas. Pateikiamos ir demonstruojamos naujos technologijos bei jų pranašumai dvejetainių duomenų masyvuose taikant pasikartojančių duomenų rūšiavimo algoritmus. Straipsnyje pateikiama rekursinių algoritmų analizė, palyginimas ir alternatyvos. Eksperimentiniai rezultatai rodo, kad, įdiegus siūlomas naujoves, galima padidinti algoritmų našumą. Il. 8, bibl. 10, lent. 2 (anglų kalba; santraukos anglų ir lietuvių k.).