2010 12th Biennial Baltic Electronics Conference (BEC2010)
Tallinn, Estonia, October 4-6, 2010

Optimization of FPGA-based Circuits for Recursive Data Sorting

D. Mihhailov[1], V. Sklyarov[2], I. Skliarova[2], A. Sudnitson[1]

[1]*Department of Computer Engineering, TUT, Raja 15, 12618 Tallinn, Estonia, E-mail: alsu@cc.ttu.ee*
[2]*DETI/IEETA, University of Aveiro, Portugal, E-mails: skl@ua.pt, iouliia@ua.pt*

ABSTRACT: The paper describes sequential and parallel methods of recursive data sorting that are applied to binary trees. Hardware circuits implementing these methods are based on the model of a hierarchical finite state machine, which provides support for recursion in hardware. It is shown that the considered technique allows the known optimization methods for conventional state machines to be applied directly. The described circuits have been implemented in commercial FPGAs and tested in numerous examples. Analysis and comparison of alternative and competitive techniques is also done in the paper.

1 Introduction

Sorting is an important problem of many high performance applications [1]. Here, we take advantage of specialized (reconfigurable) hardware and demonstrate benefits of recursive technique applied to data structures to be organized in a form of binary trees. The trees are based on links between sorted data items (tree nodes) through pointers and differ from table-based structures. It is known that recursive technique for binary tree based data structures gives numerous advantages [2,3]. This paper evaluates existing and suggests new FPGA-based digital circuits for recursive data sorting using the model of a hierarchical finite state machine (HFSM) [4].

The advantages and disadvantages of recursive techniques in software are well known [2]. It has been shown [3,5] that recursion can be implemented in hardware much more efficiently through parallelisation of algorithms and optimization of recursive calls/returns.

A significant advantage of tree-based data structures is the support for the following two important features: 1) it is not necessary to rebuild the tree in order to insert new data items; and 2) the number of data items to be sorted might be unknown. This is because the tree to be built for any number of data items that have already been processed is a part of the tree for new data items and, as a result, any resorting can be done easier than for non-pointer based data structures. Thus, the pointer-based data structures (such as trees) are better for priority buffers (queues) and similar devices.

Detailed comparison of recursive and iterative processing of binary trees is done in [3,6], where the advantages of recursion were demonstrated that are: clarity of the algorithm; the ease of modifications and improvements (indeed any modification of a recursive module does not change the remainder of the algorithmic specification), better formalization (through reusable models and the relevant design templates and specification methods). The results of experiments presented in [3,6] show that for binary tree based data processing the used recursive technique permits to construct circuits with better performance and smaller hardware resources. This paper concentrates on further improvements of circuits implementing recursive sorting algorithms applying both algorithmic and architectural optimization techniques. The challenges in optimization are capability to use cheap microelectronic devices to design configurable high-performance sorters adaptable to generally unknown number of input data items, such as that are needed for priority-driven buffering.

The remainder of this paper is organized in five sections. Section 2 describes methods for processing binary trees that are essential for recursive sorting algorithms. Section 3 suggests improvements in hardware architectures for the considered methods. Section 4 presents optimization techniques. Section 5 analyzes experimental results, compares known and proposed implementations, and suggests further improvements. The conclusion is given in Section 6.

2 Recursive Processing of Binary Trees

The use of binary trees for sorting data in hardware circuits is considered in [3,5] and it is based on the following technique. Suppose each node of the tree contains 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 and the right sub-tree contains only values that are greater. Repeated values are taken into account in a counter associated with each node. Such a tree can be easily built and traversed recursively.

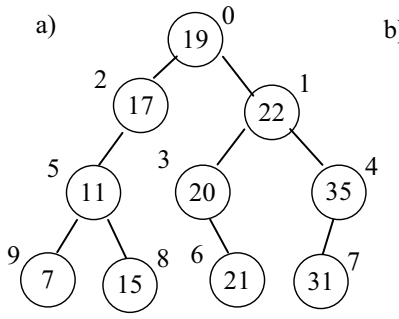
An example of a binary tree is presented in Figure 1.a in form of a graph and in Figure 1.b in form of a linked list that is kept in memory. For each node in Figure 1.a the value and the relevant address in memory are shown. The first column of Figure 1.b specifies memory location, where the node (the list item) is stored. The other columns keep left (LA) and right (RA) addresses according to the mentioned above format. Since the root is always stored

978-1-4244-7358-8/10/$26.00 ©2010 IEEE

at zero address, zero code can be safely used to indicate the absence of a node.

The known methods [3] permit to construct a binary tree from incoming data items and to output the sorted data from the tree. The first improvement is achieved through the use of dual-port memories and algorithmic modifications. Suppose the currently processed node is saved in a buffer register. Dual-port memory permits simultaneous access to the left and to the right child nodes through the LA and RA. Analysis of both child nodes and their further descendants allows a larger fragment of the binary tree to be processed within the same time slot.

Let us assume that the tree in Figure 1.a is stored in a dual-port memory as it is shown in Figure 2. Suppose the fragment of the tree enclosed in a dashed circle is currently being processed (see the contents of the buffer register in Figure 2). Each output word selected by the addresses A and B of the dual port memory keeps the same data as the buffer register (i.e. *Data+LA+RA*). Therefore, at each recursive step up to three nodes, enclosed in Figure 2 by a dashed circle, can be processed within the same time slot. Thus, descendants of child nodes can be analyzed to reduce the number of recursive calls/returns during the traversal procedure compared to the known method. Indeed, if the left child node does not have child nodes then its value can be sent 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.



b)

#	Data	LA	RA
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

Figure 1. An example of a binary tree in form of a graph (a) and linked list (b) that is stored in a memory

Another potential improvement in the performance can be achieved with the introduction of parallelism that is not supported by the known methods. One way is to traverse sub-trees of the root node in parallel. However, there is one significant limitation. Although the sub-trees are processed in parallel, the results cannot be output in parallel. 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 counterpart.

Such dependency is most certainly undesirable. It can be eliminated with distribution of the incoming data between N>1 parallel HFSM-based circuits. Each sorter unit traverses its own independent tree, while the results are mapped from the circuits to a sorted sequence.

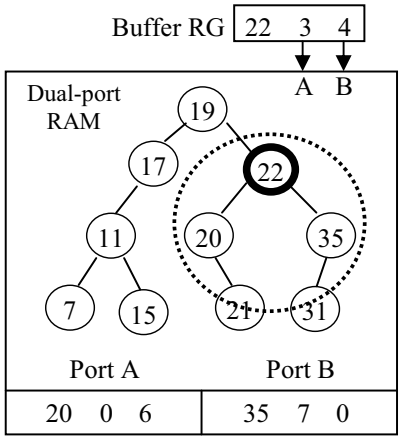


Figure 2. An example of a binary tree, stored in a dual-port memory

3 Hardware Implementation

Figure 3 depicts the proposed top-level architecture of a circuit for data sorting.

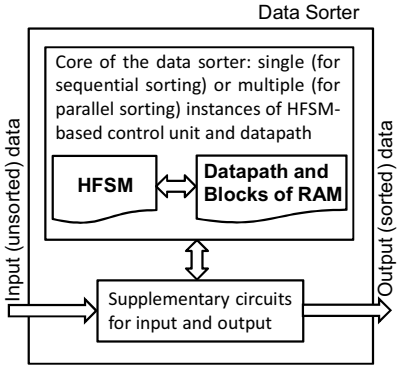


Figure 3. Top-level structure of the circuit for data sorting

The core of the data sorter is composed of a HFSM and a datapath. For sequential data sorting one instance of the data sorter is created. For parallel data sorting N>1 instances are generated.

Two different HFSM models were examined. The first model is very similar to [3] and it has the following distinctive features. The global HFSM state (GS) is defined by the active module and the state in the active module. As a result, the states in different modules can be assigned the same labels (i.e. the same codes). There are two stack memories for storing codes of modules and states. In case of hierarchical call, the GS (codes of the current module and the current state) is pushed into the stacks. When a module is terminated, the HFSM performs a hierarchical return (the GS is restored from the stacks). We will call this model *HFSM with explicit modules*.

The second model (*HFSM with implicit modules*) behaves similarly to a conventional finite state machine (FSM). It has a state register and a single stack. In this case each state has to be assigned a different label (code). The stack is needed just to know which state has to be the target of the transition when a called module is terminated. 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 the stack, it can be encoded with a smaller code. Similarly, during hierarchical return the content of the stack is decoded before being placed into the state register.

In any module all the necessary state transitions are executed through a register, much like it is done in a conventional FSM. Suppose that a new module Z has to be called in state a_j. In this case the following operations are executed at the same time: 1) the next state of the current module is saved in the stack; 2) the stack pointer is incremented; and 3) the transition from a_j to the first state of Z is performed in the state register. When the called module Z is terminated, the stack pointer is decremented and the state from the stack has to be selected for the next state transition.

There exist two modes of returns. In the first mode there are no conditional transitions from the state like a_j. Thus, we can explicitly save in the stack the target state for transition after return from the called module. In the second (more complicated) mode there are conditional transitions from the states where we have to call other modules and conditions for such transitions can be influenced by the called modules. In this case the method based on the use of a special return flag (described in [3] with all the necessary details) can be applied directly. Ordinary transitions are executed in the register similarly to conventional FSMs.

A major advantage of the second model is that it is directly applicable to all known optimization techniques that have been proposed for conventional FSMs.

4 Optimization Technique

In the last decade, probabilistic approaches have received a lot of attention as viable techniques for analyzing complex digital systems. As a rule, the control part in the high-level representation of a digital system is considered to be a FSM. Given the FSM description and the input probabilities, the probabilistic behaviour of the FSM can be studied regarding to its transition structure as a Markov chain. The input probability distribution can be obtained by simulating the FSM at a higher level of abstraction in the context of its environment or by direct knowledge from the designer [7]. By labelling each outgoing edge of each state with the probability for the FSM to make that particular transition, a finite state model, that matches the definition of a discrete parameter Markov chain, can be obtained. Analyzing the behaviour of such Markov chain allows the reachability analysis of the FSM to be performed. Using steady state probabilities, which are received as the result of such analysis, it is possible to build different kinds of quantitative estimations of FSM's stochastic behaviour. These stochastic estimations can then be successfully applied to solving various problems in the field of low-power logic synthesis.

In a high-level specification, states of the FSM are represented with variables in symbolic form. As current digital circuits employ bi-stable storage elements, which can hold one of only two possible values, transformation of these abstract variables to physical implementation requires binary encoding. In other words, each symbolic variable should be replaced with a binary vector. The resultant circuit is dependent on the selected encoding, which may affect area, performance, testability and power consumption among others.

The hardware implementation of the FSM generally consists of a register, where binary state codes are held, and combinational logic, which computes the next state and outputs. Both parts serve as power dissipation sources, whereas power is consumed during charging and discharging of load capacitances. The dynamic power dissipation in the combinational part of the circuit is very difficult to estimate, even after the state encoding is determined [8]. Therefore, reduction of switching activity in the state register was chosen as the primary optimization goal. Based on the stochastic model of FSM, the state assignment is obtained by minimizing the Hamming distance (number of bits by which two codes differ) between adjacent states with higher transition probability.

The encoding for the second HFSM model was obtained with a special CAD tool called Stochastic FSM Encoder [9]. To estimate the impact of the encoding on power consumption, Xilinx ISE 11 was used for carrying out FPGA design flow with Spartan-3E family FPGA being set as the target device. Power consumption estimation was received using XPower Analyzer tool. The default settings for the switching rate of inputs were used. The frequency of clock signal was set to 50MHz. Only the dynamic power component was considered, as it has been the target of optimization.

5 Experimental Results

Circuits that implement methods of section 2 and possess architectures of section 3 were described in VHDL. The synthesis and implementation of these circuits were done in Xilinx ISE 11 for FPGA Spartan3E-1200E-FG320 of Xilinx available on the prototyping board NEXYS2 from Digilent.

A random-number generator produces items of data with a length of 14 bits (i.e. values in an interval between 0 and 16383). These data are sorted by the following methods that were implemented and tested using two models of HFSM briefly characterized in section 3:

1. Known sequential method that was described in [3];
2. Sequential method described in section 2 and based on the use of dual-port memories;
3. Parallel method described in section 2 and based on simultaneous processing of N trees.

For each method the maximum number *n* was determined, where 2^n is the maximum number of data that can be sorted in a single FPGA Spartan3E-1200E-FG320. Table 1 presents number *n* together with number of slices (S), block RAMs (B) and the maximum attainable clock frequency F_{max} (here HFSM(explicit)/HFSM(implicit) indicates the model with explicit/implicit modules).

As can be seen from Table 1 the main restriction that limits the number of data items is the size of available embedded Block RAMs (B). Thus, the number of data items can be significantly increased if we replace the cheap Spartan-3E FPGA with a more advanced FPGA such as that are available from Virtex-5/6 families. We have used the Spartan-3E because one of the paper objectives is the low cost of data sorters.

Table 1. Implementation results for the methods 1-3

Method	n	HFSM(explicit)			HFSM(implicit)		
		S	F_{max}	B	S	F_{max}	B
1	12	2415	100	13	1175	107	13
1	13	4407	100	27	1965	103	27
2	12	2609	77	20	1146	65	20
3 (N=2)	12	2812	75	20	-	-	-
3 (N=4)	12	3343	73	20	-	-	-

Table 2 presents the results of sorting by different methods characterized in section 2. The number of data items is approximately 2^{12} as it is suitable for all methods.

Table 2. The results of experiments for the methods 1-3

Number of data items	Number of clock cycles required for sorting			
	Method 1	Method 2	Method 3 (N=2)	Method 3 (N=4)
4019	16075	11242	5622	4047
3985	15939	11175	5553	4013
3945	15779	10964	5541	3969
3969	15875	11168	5531	4001
3979	15915	11043	5587	4013
3963	15851	11087	5540	3989
4010	16039	11095	5622	4037
3977	15907	11110	5553	4013
4048	16191	11235	5706	4072
3988	15951	11028	5555	4014

Our results demonstrate that the known method [3] allows more data items to be sorted on a single FPGA. This is because this method requires less number of memory blocks. However, the performance of the known method is the worst (see Table 2). The average number of clock cycles per data item is 4. The known method exhibits the same performance if the number of data items is increased to 2^{13}.

The sequential method based on the use of dual port memories provides better result with the average number of clock cycles of 2.8 per data item.

Parallel methods give the best performance. Processing of 2 independent binary trees improves performance in approximately 1.4 clock cycles per data item (i.e. it is nearly proportional to the square root of N). The same tendency has taken place for N=4. Increasing the number of parallel sorters from 2 to 4 decreases the time of sorting to almost 1 clock cycle per data item. However, the relevant circuits require more FPGA resources (see Table 1).

The model of HFSM with implicit modules requires less FPGA resources and permits the optimization methods of section 4 to be applied. Experiments with these methods have shown the decreasing of power consumption in about 5%.

6 Conclusions

The paper suggests new hardware-oriented sequential and parallel implementations of recursive sorting algorithms. It clearly demonstrates the advantages of the proposed innovations based on prototyping in FPGA and abundant experiments. The results of the paper are not limited to just recursive sorting. They have a wide scope and can be applied effectively to numerous systems that implement recursive algorithms over tree-like structures.

Acknowledgment

This research was supported by the European Union through the European Regional Development Fund.

References

[1] R.D. Chamberlain, N. Ganesan, "Sorting on Architecturally Diverse Computer Systems", Proc. of the 3d International Workshop on High-Performance Reconfigurable Computing Technology and Applications, Portland, 2009, pp. 39-46.

[2] F.M.Carrano, Data Abstraction and Problem Solving with C++, The Benjamin/Cumming Publishing Company, Inc., 2006.

[3] 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.

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

[5] S. Ninos, A. Dollas, "Modeling recursion data structures for FPGA-based implementation", Proc. 18th Int. Conference FPL'08, Germany, 2008, pp. 11-16.

[6] V. Sklyarov, I. Skliarova, and B. Pimentel, FPGA-based implementation and comparison of recursive and iterative algorithms, Proceeding of FPL'2005, Tampere, Finland, 2005, pp. 235-240.

[7] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Markovian Analysis of Large Finite State Machines", IEEE Trans. Computer-Aided Design, Vol. 15, pp.1479-1493, 1996.

[8] W. Nőth, R. Kolla, "Spanning Tree Based State Encoding for Lower Power Dissipation", Technical report, Dept. of Computer Science, University of Wűrzburg, 1998.

[9] Available at: http://www.pld.ttu.ee/applets/.