THESIS ON INFORMATICS AND SYSTEM ENGINEERING C69

# Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models

DMITRI MIHHAILOV

TUT PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering
Chair of Digital Systems Design

**This dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and Systems Engineering on December 15, 2011**

**Supervisors:** Assoc. Prof. Alexander Sudnitson
Tallinn University of Technology, Estonia
Prof. Valeri Sklyarov
University of Aveiro, Portugal
**Advisor:** Assist. Prof. Iouliia Skliarova
University of Aveiro, Portugal

**Opponents:** Prof. Apostolos Dollas
Technical University of Crete, Greece
Dr. Heinz-Dieter Wuttke
Technical University of Ilmenau, Germany

Defence of the thesis: January 20, 2012

Declaration:
*Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted before for any degree or examination.*

/Dmitri Mihhailov/





ISSN 1406-4731
ISBN 978-9949-23-228-4 (publication)
ISBN 978-9949-23-229-1 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C69

# Rekursiivsete sortimisalgoritmide riistvaraline realiseerimine kasutades puulaadseid struktuure ja HFSM mudeleid

DMITRI MIHHAILOV

TTÜ
KIRJASTUS

*To the whole world*

# Abstract

"Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models" aims at exploration of new possibilities to improve known and to develop new sorting techniques that are particularly useful for implementation using Field Programmable Gate Arrays (FPGAs). The challenge is to use cheap reconfigurable devices to design high-performance sorters adaptable to generally unknown number of input data items. The main contributions of this thesis are:

- exploration of a new model of hierarchical finite state machine (HFSM) with implicit modules that is faster and less resource consuming compared to HFSM with explicit modules;
- development of new methods allowing tree-like structures to be represented and processed in hardware;
- application of a multi-level model for data processing;
- proof of advantages for the proposed techniques based on prototyping in FPGA, experiments and comparisons.

Hardware circuits implementing proposed sorting methods are based on the model of HFSM that provides support for modularity, hierarchy and recursion. Such a specification is more readable and provides direct support for reusability. In this work, a new model of HFSMs with implicit modules is applied. It inherits capabilities of the existing models and allows to apply optimization methods developed for conventional finite state machines.

The proposed sorting techniques are based on tree-like data structures as they permit rapid adaptation to eventual modifications in input data. Indeed, any manipulations over tree nodes are simple and fast, while the actual sorting can be done in linear time. The requirement of fast resorting is important, in particular, for the design of priority buffers (queues) and similar devices that are essential for numerous practical applications.

Finally, a multilevel model for data processing has been developed. The advantages of this model are demonstrated on the examples of data sorting. It is shown that combining different sorting algorithms lead to further performance improvements.

Experiments and comparisons demonstrate that the proposed sorting techniques can be used efficiently in low cost FPGAs. The results of this work are not limited to recursive sorting alone but have a wider scope and can be applied effectively to numerous systems that implement recursive algorithms over tree-like structures.

# Kokkuvõte

“Rekursiivsete sortimisalgoritmide riistvaraline realiseerimine kasutades puulaadseid struktuure ja HFSM mudeleid” uurib võimalusi olemasolevate ja uute sortimismeetodite parandamiseks ning väljatöötamiseks, mis oleksid eriti sobilikud realiseerimiseks programmeritavatel loogikamaatriksitel (FPGA). Eesmärgiks on kasutada odavaid FPGA-d kõrgjõudlusega sortimisseadmete projekteerimiseks suvalise mahuga andmete jaoks. Selle töö peamised saavutused on järgmised:

- uue hierarhilise lõpliku automaadi (HFSM) varjatud moodulitega mudeli uurimine, mis on kiirem ja nõuab vähem ressursse kui varjamata moodulitega HFSM;
- uute meetodite väljatöötamine, mis lubavad esitada ja töödelda puulaadseid struktuure riistvaras;
- mitmetasemelise andmetöötluse kasutamine;
- esitatud meetodite FPGA prototüüpine, eksperimendid ja võrdlused.

Loodud sortimismeetodite riistvaraline realiseerimine põhineb HFSM mudelil, mis toetab modulaarsust, hierarhiat ja rekursiivsust. Selline spetsifikatsioon on selgem ja lubab taaskasutatavust. Käesolevas töös uuritakse uut varjatud moodulitega HFSM mudelit, mis pärib kõik olemasoleva mudeli omadused ja lubab kasutada kõiki tavaliste lõplike automaatide jaoks loodud optimeerimismeetodeid.

Esitatud sortimismeetodid põhinevad puulaadsetel struktuuridel sest need suudavad kiiresti adapteeruda sisendandmete muutmisele. Puu tippudega manipuleerimine on lihtne ja kiire ning andmete sortimist saab teostada lineaarse aja jooksul. Kiire ülesortimise omadus on tähtis näiteks prioriteetsete puhverite või sarnaste seadmete projekteerimisel, mis on asendamatud paljudes praktilistes rakendustes.

Välja on töötatud mitmetasemeline andmetöötluse mudel. Selle mudeli eeliseid on demonstreeriti andmete sortimise näidete peal. On näidatud, et mitme sortimisalgoritmi ühendamine viib jõudluse parendamisele.

Eksperimendid ja võrdlused näitavad, et pakutud sortimsmeetodeid on võimalik efektiivselt kasutada odavates FPGA-des. Selle töö resultaadid ei ole piiratud ainult rekursiivse sortimisega, vaid neid saab edukalt rakendada ka muudes süsteemides, mis baseeruvad puulaadsetel struktuuridel.

# Acknowledgements

I would like to thank everyone who has advised me during my studies and/or contributed to this Ph.D. thesis.

First of all, I would like to express a sincere gratitude to my supervisors Assoc. Prof. Alexander Sudnitson (Tallinn University of Technology) and Prof. Valery Sklyarov (University of Aveiro) for their guidance and support. I would also like to thank my advisor Assist. Prof. Iouliia Skliarova (University of Aveiro) for providing valuable comments and remarks.

I would also like to show appreciation to all my colleagues from the Department of Computer Engineering (Tallinn University of Technology). A special thanks to the head of the department Assoc. Prof. Margus Kruus for his help in handling many administrative issues and Prof. Peeter Ellervee for his high expertise in the field of English language transcompilation.

I would like to acknowledge the organizations that have supported my Ph.D. studies: Tallinn University of Technology, Centre of Research Excellence in Dependable Embedded Systems (CREDES), EU Regional Development Fund (project CEBE), National Graduate School in Information and Communication Technologies (IKTDK) and Estonian Information Technology Foundation (EITSA).

Finally, I would like to thank my family for their patience and support during my studies.

Thank you all!

# Table of Contents

# List of Publications

*Journal papers:*

- D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Acceleration of Recursive Data Sorting over Tree-based Structures”, Electronics and Electrical Engineering, 7(113), 2011, pp. 51-56.
- A. Sudnitson, D. Mihhailov, M. Kruus. “Project-Oriented Approach to Low-Power Topics in Advanced Digital Design Course”, Electronics and Electrical Engineering, 6 (102), 2010, pp. 151-154.

*Conference papers:*

- V. Sklyarov, I. Skliarova, D. Mihhailov, A. Sudnitson. “Implementation in FPGA of Address-based Data Sorting”, The 21st International Conference on Field Programmable Logic and Applications (FPL 2011), Chania, Crete, Greece, September 5-7, 2011, pp. 405-410.
- V. Sklyarov, I. Skliarova, D. Mihhailov, A. Sudnitson. “High-performance Hardware Accelerators for Sorting and Managing Priorities”, IEEE Symposium of Design and Diagnostics of Electronic Circuits and Systems (DDECS 2011), Cottbus, Germany, April 13-15, 2011, pp. 313-318.
- V. Sklyarov, I. Skliarova, D. Mihhailov, A. Sudnitson. “Multilevel Models for Data Processing”, The 2011 IEEE GCC Conference and Exhibition (GCC 2011), Dubai, United Arab Emirates, February 19-22, 2011, pp.136-139.
- V. Sklyarov, I. Skliarova, D. Mihhailov, A. Sudnitson. “Processing Tree-like Data Structures for Sorting and Managing Priorities”, The 2011 IEEE Symposium on Computer and Informatics (ISCI 2011), Kuala Lumpur, Malaysia, 2011, pp. 322-327.
- V. Sklyarov, I. Skliarova, D. Mihhailov, A. Sudnitson. “Processing N-ary Trees in Hardware Circuits”, 13th International Symposium on Integrated Circuits (ISIC 2011), Singapore, December 12-14, 2011.
- D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Hardware Implementation of Recursive Sorting Algorithms”, The 2011 International Conference on Electronic Devices, Systems & Applications (ICEDSA 2011), Kuala Lumpur, Malaysia, April 25-27, 2011, pp. 33-38.
- D. Mihhailov, M. Kruus, V. Sklyarov, I. Skliarova, A. Sudnitson. “Recursion and Hierarchy in Digital Design and Prototyping: A Case Study”, International Conference on Computer Systems and Technologies (CompSysTech’2011), Vienna, Austria, June 16-17, 2011, ACM International Conference Proceeding Series, 578, pp. 45-50.
- M. Jenihhin, M. Gorev, V. Pesonen, D. Mihhailov, P. Ellervee, H. Hinrikus, M. Bachmann, J. Lass. “EEG Analyzer Prototype Based on FPGA”, IEEE 7th International Symposium on Image and Signal Processing and Analysis (ISPA 2011), Dubrovnik, Croatia, September 4-6, 2011, pp. 101-106.

- D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Synthesis and Implementation of Hierarchical Finite State Machines with Implicit Modules” The 2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2010), Cancun, Mexico, December13-15, 2010, pp. 436-441.
- D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Parallel FPGA-based Implementation of Recursive Sorting Algorithms”, The 2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2010), Cancun, Mexico, December 13-15, 2010, pp. 121-126.
- D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Application-specific hardware accelerator for implementing recursive sorting algorithms”, The 2010 International Conference on Field-Programmable Technology (FPT 2010), Beijing, China, December 8-10, 2010, pp. 269-272.
- D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Hardware Implementation of Recursive Algorithms”, 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2010), Seattle, WA, USA, August 1-4, 2010, pp. 225-228.
- D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Optimization of FPGA-based Circuits for Recursive Data Sorting” The 12th biennial Baltic Electronics Conference (BEC 2010), Tallinn, Estonia, October 4-6, 2010, pp. 129-132.
- D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Optimization of Recursive Sorting Algorithms for Implementation in Hardware”, 22nd International Conference on Microelectronics (ICM 2010), Cairo, Egypt, December 19-22, 2010.
- D. Mihhailov, A. Sudnitson, K. Tarletski. “Web-Based Tool for FSM Encoding Targeting Low-Power FPGA Implementation”, The 27th International Conference on Microelectronics (MIEL 2010), Nis, Serbia, May 16-19, 2010, pp. 349-352.
- A. Sudnitson, D. Mihhailov, M. Kruus. “Advanced Topics of FSM Design Using FPGA Educational Boards and Web-Based Tools”, IEEE East-West Design & Test Symposium, Moscow, Russia, 2009, pp. 446-449.
- A. Sudnitson, D. Mihhailov, M. Kruus. “Cooperation of FPGA-Based Educational Boards and Web-Based Point Design Tools for Research and Education”, IFIP EduTech’09 International Workshop, Florianopolis, Brazil, October 15-16, 2009.
- A. Sudnitson, D. Mihhailov, M. Kruus, K. Tarletski. “FSM Decomposition with Application to FPGA Synthesis”, International Conference on Computer Systems and Technologies (CompSysTech’09), Ruse, Bulgaria, 2009, ACM International Conference Proceeding Series, 433, pp. IV.4.1-IV.4.6.
- D. Mihhailov, M. Kruus, A. Sudnitson. “FPGA Platform Based Digital Design Education”, International Conference on Computer Systems and Technologies (CompSysTech’2008), Gabrovo, Bulgaaria, 2008, ACM International Conference Proceeding Series, 374, pp. IV.1-IV.6.

# List of Abbreviations

| | |
|---|---|
| ASIC | Application-Specific Integrated Circuit |
| VLSI | Very-Large Scale Integration |
| CPU | Central Processing Unit |
| GPU | Graphics Processing Unit |
| SIMD | Single Instruction, Multiple Data |
| RAM | Random-Access Memory |
| SoC | System-on-Chip |
| FPGA | Field Programmable Gate Array |
| LUT | Lookup Table |
| BRAM | Block RAM |
| HDL | Hardware Description Language |
| VHDL | VHSIC HDL |
| VHSIC | Very-High-Speed Integrated Circuits |
| FSM | Finite State Machine |
| FSMD | FSM with Datapath |
| HGS | Hierarchical Graph Scheme |
| HFSM | Hierarchical FSM |
| RHFSM | Reconfigurable HFSM |
| PHFSM | Parallel HFSM |
| BST | Binary Search Tree |
| PLP | Process-Level Parallelism |
| SLP | Statement-Level Parallelism |
| SLSL | System-Level Specification Languages |
| FP language | Functional Programming language |
| CAD | Computer-Aided Design |
| EEG | Electroencephalography |

# Introduction

The concept of reconfigurable computing has been introduced in the early 1960s [21]. However, only in the middle of the 80th this technology was actually made available. Since then, reconfigurable computing became a subject of intensive research.

With the constant growth of integration level, today's circuits contain way over millions of gates. Therefore, it is quite difficult to know if a designed circuit is correct before fabrication of a test prototype. Construction of a dedicated prototype is quite expensive and also introduces significant delays, which are required for its fabrication. Furthermore, the prototype is made fairly late in the development cycle, as the entire design must be specified first. In such circumstances, the emulation of hardware circuits in programmable logic devices allows to overcome these problems. Recent commercially available field-programmable gate arrays (FPGAs), such as Virtex family from Xilinx [86] and Stratix family from Altera [87], offer large amount of logic, arithmetic units (multipliers), embedded memory blocks and even processor cores, thus becoming an adequate platform for emulating complex systems. Reprogrammability of FPGAs makes it possible not only to implement and verify the full design itself, but also to build early prototypes of the sub-circuits using the same prototyping device. This flexibility is particularly valuable if the system is likely to be modified either to improve performance, or to add new features, or due to the change of standards. Some may argue that developing a prototype for FPGA is also time-consuming and requires additional time and resources, thus delaying the project as well. However, this disadvantage is negligible compared to the amount of time and effort required for manufacturing of a custom VLSI device.

The property of reconfigurability is also essential for such applications as evolvable hardware, which comprises a variety of approaches to the design of electronic circuits using evolutionary techniques [22]. Evolvable hardware is an extremely promising and rapidly developing research direction, which attracts more and more attention (as indicated by the increasing number of publications in this area). The potential of evolvable hardware is quite extensive because theoretically it allows the construction of a circuit with a given specification, whose structure is previously unknown. As it was shown in a number of papers, the results of evolution can be more efficient than any known conventional design [22]. Evolvable techniques are also important for implementation of adaptive hardware. When for some reason such system is impossible to reach (e.g. systems used in space applications), an adaptive circuit can modify its configuration to compensate

for a fault or due to the changing operational conditions in order to retain the original behavior.

Originally FPGAs were primarily used as glue logic. However, due to the constant growth of size and functionality, FPGAs crawled their way into such areas as cryptography, digital signal and image processing to name a few. These applications are characterized by large amounts of data to be processed and are very well suited for parallel implementations. The inherent parallelism of the logic resources in an FPGA makes it an ideal platform for acceleration of such computationally intensive tasks compared to a similar implementation on general-purpose computers.

## 1.1. Motivation

Using and taking advantage of application-specific circuits in general and FPGA-based accelerators in particular have a long tradition in data processing [23]. For example, the system [24] solves a sorting problem over multiple hardware shading units achieving parallelization through using SIMD operations on GPU processors. The algorithms [25-27] are very similar in nature, but target SIMD instruction sets of PowerPC 970MP, Cell and Intel Core 2 Quad processors. The use of FPGAs was studied within projects [28,29] implementing traditional CPU tasks on programmable hardware. In [30] FPGAs are used as co-processors in Altix supercomputer to accelerate XML filtering. Many problems of combinatorial optimization, like Boolean satisfiability, covering of Boolean matrices, graph coloring, can also be efficiently solved with the aid of reconfigurable hardware [31-33].

Reconfigurable systems achieve significant increases in performance by being able to adapt to a particular application, a feature that is not supported by other platforms (e.g. general-purpose multi-core processors, graphics processing units). Firstly, many computationally intensive tasks involve a huge number of similar operations. But as a rule, these operations are not exactly the same for different problems. Thus, it is not easy to construct a universal processing unit, i.e. it has to be customized for a particular problem that is going to be solved. This can easily be done with the aid of FPGA technology. Secondly, different practical applications might require solving tasks with varying complexity. Parameterizable circuits that provide such opportunity can easily be implemented in FPGAs. Thirdly, FPGA enables to build on the same microchip any desired (customized) interface between an accelerator and a general-purpose computational system (or any specialized system that requires an accelerator). Fourthly, the complexity of recent FPGAs allows to construct a complete system-on-chip with application-specific accelerator being part of that system. In this case, the accelerator can be integrated even more efficiently (e.g. memory organization can be tailored to specific data sizes). Lastly,

the recent commercial tools allow reconfigurable digital circuits to be actually synthesized from system-level specification languages (SLSL) such as SystemC [88], thus, allowing to work at a very high level of abstraction. This essentially blurs the line between software and hardware development, which means that designers with a limited knowledge of the targeted FPGA architecture are still capable of producing rapidly functional, algorithmically optimized designs.

Sorting is a traditional data processing technique. There are many methods [1,2] that permit sorting problems to be solved. Notable results have been achieved through applying such techniques as parallelism, pipelining, non-sequential circuits and building specialized blocks in hardware among others. However, any particular technique cannot be seen as a universal approach (producing an optimal result for any set of data). Every method has its own advantages and disadvantages, which can actually depend on the execution platform. A special attention has been paid to such competitive implementation platforms as graphics processing unit (GPU), multi-core CPU and FPGA [34-36]. The appearance of reconfigurable computing provided an attractive option for implementation of data sorting in the context of hardware [23], as it permits the design constraints of multi-core CPU and GPU with predefined architectures to be eliminated.

## 1.2. Thesis contribution

A great deal of research effort in this thesis is aimed at exploration of new possibilities to improve the known and develop new sorting techniques that are particularly useful for implementation in FPGAs. The challenge is to use cheap reconfigurable devices to design high-performance sorters adaptable to generally unknown number of input data items. The proposed techniques are based on tree-like data structures as they permit to execute the required operations faster and allow to apply both sequential and parallel processing. An important advantage of tree sorting compared to other methods is an opportunity of rapid adaptation to eventual modifications in input data. This is basically because a tree built for any number of data items that have already been processed is a part of the tree for new data items. As a result, any manipulations over tree nodes (e.g. insertion of a new node) are simple and fast, while the actual sorting can be done in linear time. The requirement of fast resorting is important, in particular, for the design of priority buffers (queues) and similar devices, which are essential for numerous practical applications.

Many computational algorithms can be implemented using various techniques based on recursive specifications. Experience in software development shows that recursion is not always appropriate, particularly when a clear efficient iterative solution exists. However, even in software applications, applying recursive algorithms for various kinds of binary search is considered to be a notable

exception. In this case a recursive technique is comparable to iterative approach and allows more clear, compact and easily understandable specifications to be produced. Besides recursion can be implemented in hardware more efficiently than in software.

Hardware circuits implementing proposed sorting methods are based on the model of a hierarchical finite state machine (HFSM), which enables recursion to be realized in hardware. HFSM also provides support for modularity that permits to develop any complex algorithm step by step, concentrating efforts at each stage on a specified level of abstraction. Such specification is more readable and provides direct support for reusability. In this work a new model of HFSMs with implicit modules is applied. It inherits capabilities of other existing models (in particular, provides support for modularity, hierarchy, and recursion) and requires a very simple stack memory. In this model the codes of all states are unique and the modules are hidden (implicit). This allows to apply optimization methods developed for conventional finite state machines. Experimental results demonstrate that HFSMs with implicit modules are faster and less resource consuming compared to HFSMs with explicit modules.

This work is focused on improvement of circuits implementing recursive sorting algorithms over N-ary trees by applying both algorithmic and architectural optimization techniques. This was achieved through the use of dual-port memories (available within many commercial FPGAs). Embedded dual-port memory blocks permit simultaneous access to several nodes in a tree. Simultaneous analysis of these nodes and their connectivity allows to cover a larger portion of a tree during traversal. In order to accelerate data processing and reduce memory consumption, a compression method using positional encoding for tree-like structures can be employed.

Another potential improvement in performance can be achieved with introduction of parallelism. It may be possible to put multiple instances of the same algorithm to work on different parts of the tree. The most obvious choice involves parallel traversal of left and right sub-trees beginning from the root and extended to other nodes. Naturally, more parallel branches can be introduced using cascade structures of more than two sorters that are activated for different sub-trees on certain paths from the main root. However, there is one significant limitation. 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. Such dependency can be eliminated if the main sorter activates secondary sorters only when there are a sufficient number of subsequent processing steps for all sorting circuits. Balance dependency can also be eliminated by distributing the incoming data between N>1 parallel HFSM-based circuits. Then each sorter unit

traverses its own independent tree, while the results are mapped from the circuits to a sorted sequence.

This work also describes the hardware implementation and optimization of sorting algorithms that use data items as memory addresses with one-bit flags indicating presence of data (address-based data sorting). The explored technique can be applied either directly or through tree-walk tables permitting number of bits in sorted data items to be increased by constructing and traversing N-ary trees (N>2) that are well balanced and have a fixed depth. It is allowed more than one data item to be assigned to leaves and such sets of items are processed by fast combinational circuits (e.g. sorting networks).

Finally, a multilevel model for data processing is developed. The advantages of this model are demonstrated on examples of data sorting. Different models can be combined, such as the use of the walk technique, binary trees, sorting networks and address-based sorting.

The relevant implementations were verified in commercially available FPGAs. Experiments and comparisons demonstrate that the proposed sorting techniques can be used efficiently in low cost FPGAs. However, in order to process big sets of data either more powerful FPGA or external memory is required.

The results of this work 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-like structures.

## 1.3. Thesis outline

The remainder of this thesis is divided into six chapters:

**Chapter 2**. This chapter gives an overview and comparison of the best known sorting algorithms. It provides necessary definitions and basic classification parameters for sorting algorithms. The brief summaries and comparison for the main classes of sorting algorithms are also presented here.

**Chapter 3.** This chapter describes various techniques to implement recursion in hardware. The brief summaries and comparison of the suggested methodologies are presented here.

**Chapter 4.** This chapter is devoted to the model of hierarchical finite state machine (HFSM). Chapter provides description of hierarchical graph schemes, HFSM models and guidelines for implementation of HFSM in hardware. It is shown on a case study that HFSM model is applicable to reuse techniques. Some practical tasks where the use of HFSM model is advantageous are also mentioned.

**Chapter 5.** This chapter presents a number of simple, but efficient sorting techniques that are particularly useful for implementation in FPGAs. The challenge is to use cheap reconfigurable devices to design high-performance sorters adaptable to generally unknown number of input data items. The proposed sorting techniques are based on tree-like structures and address-based sorting. A number of methods that allow to improve the sequential flow of the sorting algorithms, apply parallel processing and reduce memory requirements are suggested here.

**Chapter 6.** This chapter reports experimental results, comparison and analysis of the proposed methods.

**Chapter 7.** The conclusions are drawn in this chapter.

# 2. Sorting algorithms

Sorting is an important problem of many high performance applications [36]. The main reason for sorting being so useful is that it is much easier for people to handle data when it is sorted than when it's not. For example, finding someone's phone number is trivial when they are sorted by the owner's last name in a phone book or browsing through search engine's results when they are presented based on their relevance. Once the data in the array is sorted it is much easier to find and remove duplicates or perform statistical calculations such as removing outliers, finding the median or computing percentiles. Efficient sorting may be important for optimizing the use of other algorithms that require sorted sequences in order to work correctly, but which may have nothing to do with sorting at all. Good examples are data compression, computer graphics and combinatorial optimization to name a few.

This chapter provides an overview and comparison for the best known sorting algorithms. Section 2.1 provides necessary definitions and basic classification parameters for sorting algorithms. Sections 2.2 through 2.10 give brief summaries and comparison for the main classes of sorting algorithms. The conclusions are drawn in section 2.11.

## 2.1. Definitions and classification

Informally, an algorithm is any well-defined computational procedure that takes some value (or set of values) as input and produces some value (or set of values) as output. An algorithm is thus a sequence of computational steps that transform the input into the output [2].

The input sequence for the sorting algorithm is usually an n-element array, although it may be represented in some other fashion (such as a linked list). Elements of the array are usually records, each containing key and associated satellite data. The objective of the sorting algorithm is to rearrange the items in such a way, that their keys are ordered in accordance with some well-defined ordering rule [3] (e.g. numerical or alphabetical order). Sorting algorithms are often classified by:

- comparison or non-comparison sorting;
- worst, average and best cases for the array of size n;
- adaptability;

- memory usage;
- stability;
- use of recursion.

The amount of time required to execute sorting depends on the input size: sorting a thousand keys should take significantly longer than sorting ten keys. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input [2]. For the remainder of this work the worst-case running time will be considered, as it provides an upper bound on the running time for any input. Knowing a worst-case running time gives a guarantee that the algorithm will not take longer to execute. In practice, the worst case scenarios occurs fairly often (e.g. searching a database for an absent piece of information). Occasionally the average case will be also considered, when it is not as bad as the worst case. Both worst-case and average running times provide a simple characterization of the algorithm's efficiency and allow to compare the relative performance of algorithms.

The limiting behavior of algorithm's running time function is usually described using a big O notion. When it is said "the running time is O(f(n))", it means that for any value of n the running time is bounded from above by the value of f(n). Although it is often possible to determine the exact function for the running time, such precision is not usually required to describe its growth rate. For large inputs (when n approaches infinity) the term with the largest growth rate becomes dominant making its constant factor and lower order terms insignificant (they are usually omitted in the O-notion). For typical sorting algorithms a good behavior is O(n log n) and bad behavior is $O(n^2)$, while ideal behavior would be O(n).

Sorting two input sequences of the same size may also take different amounts of time due to the different order of elements (e.g. one is nearly sorted, while the other is not). For example, comparison-based sorting algorithms, which evaluate the elements of the array using comparison operation, need at least O(n log n) comparisons for most inputs in the worst case. However, for a sorted input sequence a sorting algorithm like insertion sort would need O(n) comparisons. An algorithm, which takes into account the existing order of items in its input array, is called adaptive. Thereby, it would take an adaptive sorting algorithm less time to sort an input array the closer it is to being already sorted.

Some algorithms require auxiliary memory for data to be temporarily stored. In terms of memory usage the sorting algorithms divide into two basic types: those that sort in-place (the input is overwritten by the output as the algorithm executes) and use no extra memory (except for a small function-call stack or a constant number of instance variables), and those that need enough extra memory to hold at least another copy of the array to be sorted. Another memory related issue is when the size of the array to be sorted approaches or exceeds the size of available primary

memory (RAM), so that much slower memory (hard disk drive) must be employed. In this case the memory usage pattern of a sorting algorithm becomes important. Thus, the number of times sections of memory must be copied or swapped, the number of passes and the localization of array accesses can define the performance. Therefore, an algorithm, which is fairly efficient when the input array can easily fit into RAM, may become impractical if it requires great number of accesses to hard disk drive.

Stable sorting algorithms maintain the relative order of items with equal keys. It means that if two items have the same key and one of them appears before the other in the input, then this order would also be preserved in the sorted output. This property is important when, for example, the input data is timestamped. If the sorting algorithm is unstable, the output may not necessarily be in timestamp order after sorting. In case all keys are unique or when the entire element is considered to be the key, stability is not an issue.

Many sorting algorithms are recursive in nature. The basic idea is to divide a given initial problem into a finite (and usually very small) set of simpler sub-problems in such a way that every sub-problem is of exactly the same type as the original problem [4]. Subsequently, the same decomposition can be applied recursively to each of the sub-problems until newer sub-problems becomes so simple that their solution is known (this situation is identified as a base case). Once a solution to the base case is obtained, the previous sub-problem can also be easily solved by combining base case solutions. By moving gradually from smaller sub-problems to bigger sub-problems, a solution to the original problem is found. This strategy is usually termed as a divide-and-conquer paradigm.

Many examples that demonstrate the advantages of recursion are presented in [2,37-42]. Recursive specifications often produce elegant and easier to understand solutions than the respective iterative specifications. Therefore, practically all modern programming languages provide support for recursion. Recursive functions (as well as functions in general) are implemented in general-purpose computers with the aid of stack memory which keeps all necessary information permitting to return from a recursive call and to restore the state of data as it was before the recursive call. Managing stack (pushing there all the required data before a function call and popping these data as soon as a recursive return is to be done) incurs an overhead for each function call, and recursive functions magnify this overhead because eventually a large number of recursive calls can be generated [4]. But since the use of recursion frequently clarifies complex programs and, in some cases, can be very efficient (e.g. binary and N-ary search), additional overhead may be ignored.

## 2.2. Bubble sort

Bubble sort is a simple comparison-based sorting algorithm. Although bubble sort is one of the simplest sorting algorithms to understand and implement, it is far too inefficient for use on large, unordered data sets. Donald Knuth in [1] concluded that "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems".

The concept of bubble sort is illustrated in Fig.2.1. The algorithm starts at the beginning of the input array by comparing the first two elements. If the first element is found to be greater than the second, they are swapped. The same procedure is applied to each pair of adjacent elements up to the end of the data set. As the result, the greatest element is propagated to the end of the array, thus reaching its final sorted position. The unsorted part is then processed in the similar fashion, repeating until no swaps have occurred during the last pass or unsorted part reaches length of one element.

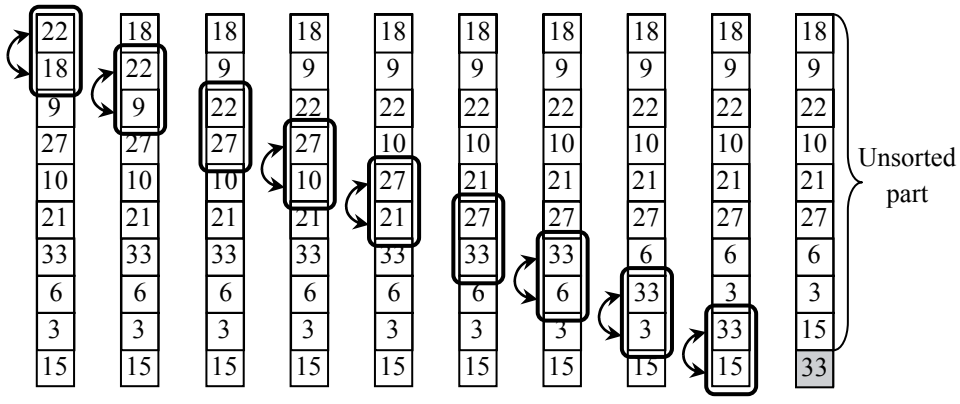



**Fig.2.1.** First pass of bubble sort

For an array of size n, both worst-case and average computational complexity of bubble sort are $O(n^2)$, which makes bubble sort highly impractical for large arrays. However, being an adaptive sorting algorithm, bubble sort may be efficiently used on an array that is already sorted except for a very small number of elements. It should be also noted that bubble sort works in-place.

The positions of the elements in the input array play a large part in determining the performance of bubble sort. Large elements at the beginning of the array are quickly propagated to their final positions and, therefore, do not pose a problem. Small elements at the end, on the other hand, move to their final positions extremely slow. These two types of elements have been termed as rabbits and

turtles, respectively. Various research efforts have been made to improve bubble sort by reducing the number of turtles.

Cocktail sort [1] achieves a fairly good performance compared to bubble sort by trying to solve the turtle problem. The main difference is that cocktail sort moves in both directions on each pass through the array. The first pass moves the largest element to its correct place at the end (just like in the bubble sort). However, the following pass of the unsorted part commences in the opposite direction by moving the smallest element to its correct place at the beginning. Although this results in an undoubtedly better performance, cocktail sort still retains $O(n^2)$ worst-case complexity of a standard bubble sort.

Another algorithm that tries to improve the bubble sort by eliminating the turtles is comb sort. Comb sort was originally designed by Wlodzimierz Dobosiewicz in 1980 [43] and later was rediscovered and popularized by Stephen Lacey and Richard Box with a Byte Magazine article published in April 1991. In bubble sort only adjacent elements are compared. The basic idea of comb sort is that the distance between compared elements (the gap) can be much larger (shell sort also exploits this idea, but, as it is rather a modification of insertion sort, it will be discussed in the next section). The gap starts out as the length of the array to be sorted and gets divided by the shrink factor (generally 1.3) after each pass. This process is repeated until the gap becomes equal to one. At this point, comb sort becomes equivalent to a bubble sort. However, by this time most of the turtles should have been gone, thus making bubble sort efficient. This allows to bring the computational complexity of bubble sort down to O(n log n).

## 2.3. Insertion sort

Insertion sort is a simple comparison-based adaptive sorting algorithm that is relatively efficient for either small or mostly sorted arrays. It is often used as a part of more sophisticated algorithms.

Insertion sorting is typically done in-place within the input array. Array is imaginary divided into two parts: sorted part and unsorted part (Fig.2.2a). At the beginning, sorted part contains only the first element of the array and unsorted one contains the rest. At every step, algorithm takes first element in the unsorted part and inserts it into the correct place of the sorted part (Fig.2.2c). We need to make space to insert the current unsorted item by moving larger items in the sorted part one position to the right (Fig.2.2b). When unsorted part becomes empty, algorithm stops. The sorting procedure is very similar to the way many people would sort a hand of playing cards.

On the whole, insertion sort is an excellent method for either partially sorted or small arrays. Indeed, the best case is an input array that is already sorted. In this scenario insertion sort has a linear running time (O(n)). During each iteration the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array. This fact is important not just because such arrays frequently arise in practice, but also because such arrays also appear at intermediate stages of advanced sorting algorithms. Some divide-and-conquer algorithms sort by dividing the array into smaller sub-arrays. A useful optimization in practice for these algorithms is to use insertion sort for sorting these small sub-arrays, when insertion sort can outperform these usually more complex algorithms.

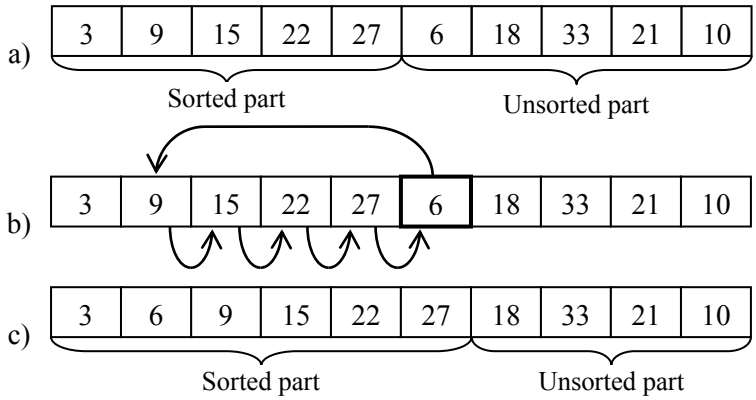



**Fig.2.2.** Insertion sort: a) initial step condition; b) insertion procedure; c) step result

The worst case input is a reverse sorted input array. In this scenario during each iteration the entire sorted part is scanned and shifted before insertion of the element. For this case insertion sort has a quadratic running time ($O(n^2)$). Bubble sort is equivalent in running time to insertion sort in the worst case, but the two algorithms differ greatly in the number of necessary swaps. Experiments by Astrachan [44] sorting strings in Java show bubble sort to be roughly 5 times slower than insertion sort and 40% slower than selection sort (considered in the next section).

Shell sort is a variation of insertion sort that is more efficient for larger arrays, as it exploits the adaptability of the insertion sort. Shell sort was invented by Donald Shell in 1959 [45]. It improves upon insertion sort by allowing the comparison and exchange of elements that are far apart (much like comb sort does for the bubble sort). The sorting algorithm compares elements separated by a distance (the gap) that decreases on each pass (defined with gap sequence). The last pass is actually a pure insertion sort, but by this time the array should be practically sorted. Shell sort has distinctly improved running times in practice over insertion sort. However, while it is easy to implement the shell sort, its analysis is very difficult [46]. The choice of a good gap sequence can also be a challenge.

## 2.4. Selection sort

Selection sort is an in-place comparison-based sorting algorithm. Selection sort is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. It is inefficient for large arrays and generally performs worse than the similar insertion sort.

For the sorting the array is divided into two parts (Fig.2.3a): the already sorted part (which is found at the beginning) and the remaining unsorted part (which occupies the rest of the array). Then the algorithm works as follows:

- Find the minimum unsorted value (Fig.2.3b);
- Swap it with the value in the first unsorted position (itself if the first value is the smallest) (Fig.2.3c);
- Repeat the above steps for the remainder of the unsorted part (Fig.2.3d).

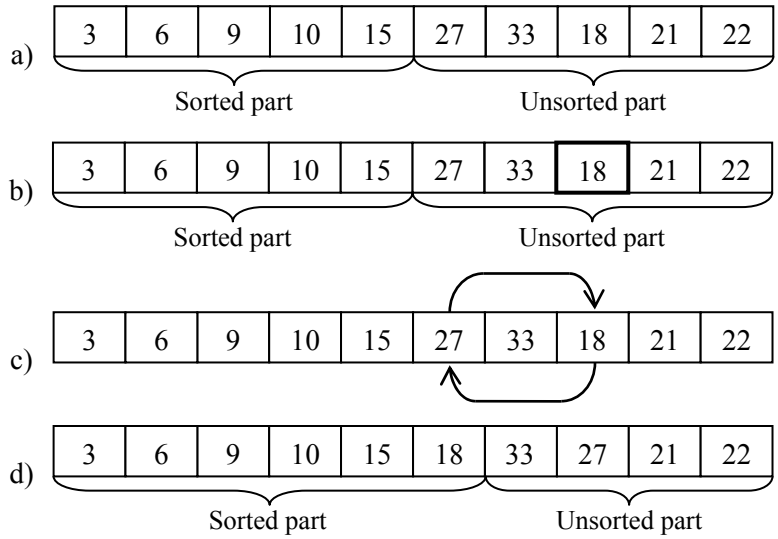



**Fig.2.3.** Selection sort: a) initial step condition; b) selection procedure; c) swap procedure; d) step result

The process of finding the smallest item on one pass through the array does not give much information about where the smallest item might be on the next pass. It takes about as long to run selection sort for an array that is already in order or for an array with all keys equal as it does for a randomly-ordered array. This property can be disadvantageous in some situations, as it means that selection sort is not an adaptive sorting algorithm. However, this also makes selection sort very predictable, which may be good for some real-time applications (running time will be identical for any order). Another useful feature of the selection sort is that it

requires only n swaps of items in the array. Thus the data movement inside the array is minimal (a rather unique property).

Although selection sort is usually greatly outperformed on larger arrays by divide-and-conquer algorithms, it would typically be faster for small arrays. Therefore, it may be useful for the divide-and-conquer algorithms to switch to selecion sort for sorting small sub-arrays.

It is not very difficult to analyze the running time of the selection sort. Finding the smallest element of the array requires scanning of all n elements. Finding the next smallest element requires scanning through the remaining n-1 elements and so on, resulting in a $O(n^2)$ complexity.

Heapsort would be a much more efficient version of selection sort. It was invented by J.W.J. Williams [47] and refined by R.W. Floyd [48] in 1964. The speed-up of the selection procedure is accomplished by using a data structure called a heap. Heap is a special type of a tree-based data structure with the following constraints:

- tree is binary (binary heap);
- tree is complete;
- heap property is satisfied.

In a binary heap each parent node has at most two child nodes. The tree is completely filled on all levels except possibly the lowest, which is filled from left to right. The nodes are ordered in such a way that parent node is greater than or equal to each of its children (max-heap). In this case the largest key is stored at the root. Alternatively, heap can be organized in the opposed way (min-heap), which results in smallest key being stored at the root.

Complete binary trees provide the opportunity to use a compact array representation that does not involve explicit links. No additional pointers are required as the parent and children of each node can be found by arithmetic on array indices. The nodes of the binary tree are stored sequentially within an array in level order. The root is placed at position 1 (this is usually done in order to simplify arithmetic), its children at positions 2 and 3, their children in positions 4, 5, 6, and 7, and so on. Thus, for a node in position k its parent can be found in position k/2, while its children are stored in positions 2k and 2k+1. In this case the division/multiplication by 2 can be replaced with right/left shift respectively. An example of binary tree being represented with an array is shown in Fig.2.4a. Note, that each node of the binary tree is tagged with a corresponding array index.

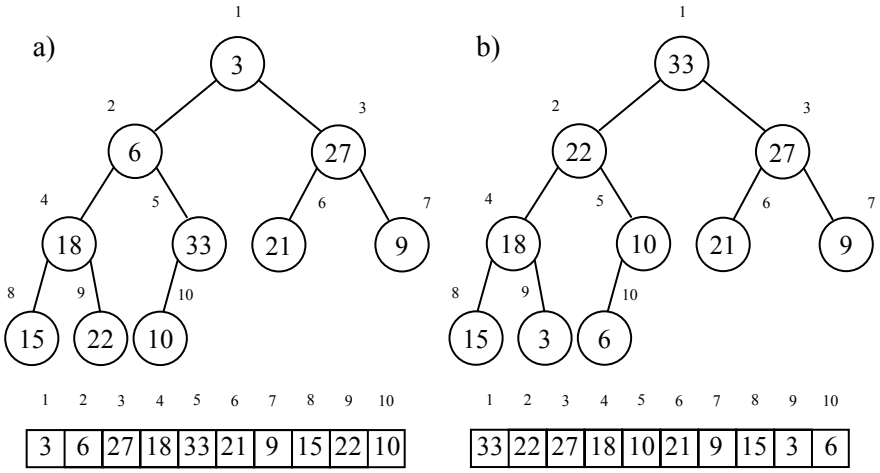



**Fig.2.4.** Binary tree representation: a) of the unordered array; b) of the max-heap

Heapsort breaks into two phases: heap construction (the original array is reorganized into a max-heap) and actual sorting (the items are pulled out of the heap in decreasing order to build the sorted result). Note, that all of these steps can be done within the same memory space, where the input array has been initially stored (sorting is done in-place).

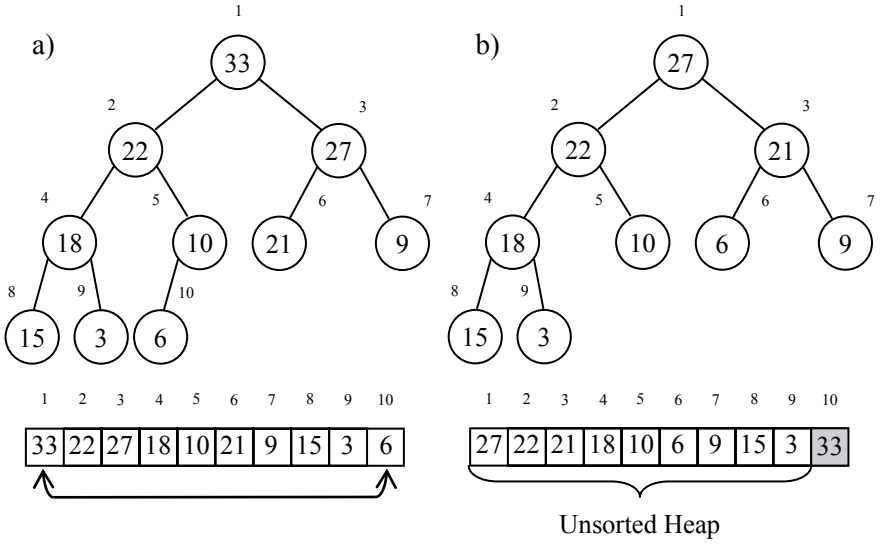



**Fig.2.5.** Heap sort: a) selection and swapping procedure; b) heap is rebuilt

Once the array has been made into a heap (Fig.2.4b), the root node is guaranteed to contain the largest element. It is then selected and swapped with the last unsorted

element in the heap (Fig.2.5a). This process effectively removes the largest element from the heap and places it in the correct sorted position. As a result, at each step the heap becomes smaller. The new root element might violate the max-heap property, so it needs to be restored before the next step (Fig.2.5b). The algorithm stops when the heap is destroyed leaving a sorted array in its place.

Heapsort is slightly adaptive, though not in any particularly useful manner. In the nearly sorted case, the heap construction phase destroys the original order. In the reverse sorted case, the heap construction phase is as fast as possible since the array is ordered according to heap property, but then the sorting phase requires worst time to complete. In 1981 Edsger Dijkstra developed a variation of heapsort called smoothsort [49], which comes closer to O(n) running time if the input is already sorted to some degree. The main difference is that instead of a binary heap, the smoothsort uses a custom structure which basically is a series of heaps with decreasing sizes (none of the heaps have the same size) and whose roots are stored in ascending order. This means that an already sorted array would not require any rearrangements in order to be converted into a valid series of heaps.

## 2.5. Merge sort

Merge sort is a comparison-based divide and conquer sorting algorithm. It was invented by John von Neumann in 1945 [1]. Merge sort takes advantage of the ease of merging two already sorted arrays into a single sorted array. Merge sort has seen a relatively recent surge in popularity for practical implementations, being used for the standard sort routine in the programming languages Perl, Python and Java, among others.

The merge sort algorithm closely follows the divide-and-conquer paradigm. It performs the following steps:

- divide unsorted input array into two sub-arrays of about half the size;
- sort each sub-array recursively using the merge sort;
- merge sorted sub-arrays to produce the result.

The division stops when the length of a sub-array to be sorted reaches one, in which case it is considered to be already in sorted order (Fig.2.6a). The key operation of the merge sort algorithm is the merging itself. In order to merge two sorted sub-arrays into a single sorted output, at each step the smallest element is removed from its array and is sequentially added to the output (Fig.2.6b). As the elements are already ordered, there is no need to search the entire array for the smallest item. When one of the sub-arrays has been completely stored in the output, the remaining elements of the second sub-array are simply copied.

For sorting array of size n, merge sort has an average and worst-case performance of O(n log n). Merge sort's most common implementation does not sort in place (the memory size of the input must be allocated for the sorted output to be stored in). Sorting in-place is possible [50], but is very complicated and offers little performance gains in practice. Also, being a recursive algorithm, merge sort requires memory to store call overhead (although iterative implementation is also possible). Merge sort parallelizes well due to use of divide and conquer approach.

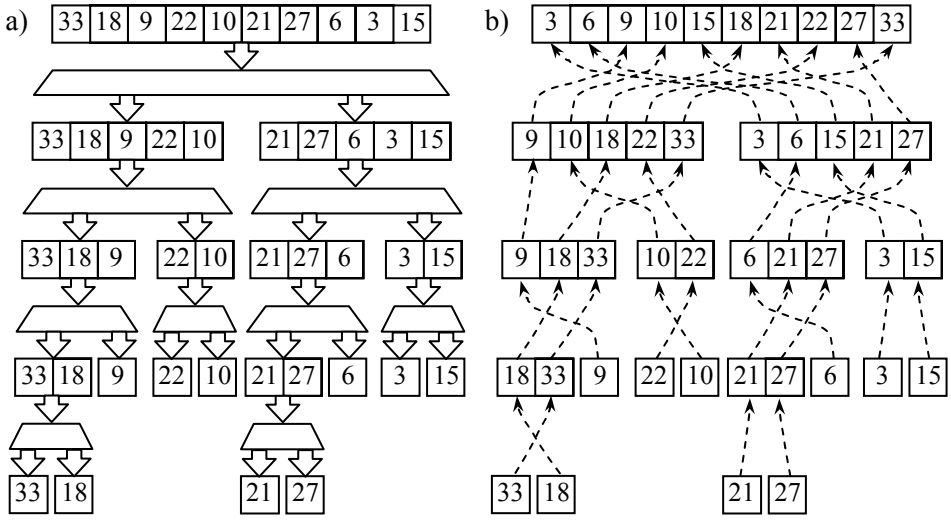



**Fig.2.6.** Merge sort: a) divide phase; b) merge phase

Timsort is a hybrid sorting algorithm derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It was invented by Tim Peters in 2002 [89] for use in the Python programming language. Timsort is based heavily on the fact that most real time data is invariably partially ordered. The algorithm operates by finding runs in the data. A run is a sub-array, where elements are ordered either in ascending or a strictly descending sequence. If it is descending, it must be strictly descending, since these runs are later reversed by a simple swap of elements from both ends converging in the middle. If necessary the runs are created with insertion sort. Then the merge sort is used to produce the final sorted array. Like merge sort, timsort has a worst-case time complexity of O(n log n).

## 2.6. Quicksort

Quicksort is a comparison-based divide and conquer sorting algorithm. It was developed in 1960 by Tony Hoare [51]. Quicksort is popular because it works well for a variety of different kinds of input data and is substantially faster than any other sorting method in typical applications [3]. Its primary drawback is that although

quicksort is not difficult to implement, it is still fragile in the sense that some care is involved in the implementation to be sure to avoid bad performance [3].

Quicksort relies on a partitioning operation. In order to partition an array the following steps are performed:

- select pivot element;
- rearrange the array in such a way that all elements which are smaller than pivot appear to the left and all greater elements appear to the right of the pivot element (equal elements can go either way);
- recursively sort left and right sub-arrays using quicksort.

The partitioning procedure is shown in Fig.2.7. Firstly, the pivot element is selected (Fig.2.7a). It is placed to the rightmost (leftmost) position in order to eliminate excessive movement of that element during partitioning procedure. Next, the elements of the array are scanned from both ends (excluding pivot itself). If during the scan from the left an element is found, which is greater than pivot, and during the scan from the right an element is found, which is smaller than pivot, they are swapped, as they are obviously out of place (Fig.2.7b). When the whole array has been scanned in the similar manner, the pivot element is swapped with the leftmost element of the right sub-array (rightmost element of the left sub-array), thus placing it to the final sorted position (Fig.2.7c). The left and the right sub-arrays are then sorted by applying the same procedure (Fig.2.7d).

The most complex issue in quicksort is obviously the selection of a good pivot element. Consistently poor choices of pivots can result in drastically slower performance. This problem is usually solved by choosing either a middle element, a random element or (especially for longer partitions) the median of a small sample of items for the pivot. For the later approach it turns out that most of the improvement comes from choosing a sample of size three and then partitioning on the middle item [52]. Doing so will give a slightly better partition at the cost of computing the median.

The running time of quicksort depends on whether the partitioning is balanced or not. If the partitioning is balanced the algorithm runs as fast as possible and exhibits performance of O(n log n). If the partitioning is unbalanced, however, the worst-case performance of $O(n^2)$ can be expected. In practice, the average-case running time of quicksort tends to be much closer to the best case than to the worst case [3].

One significant drawback of quicksort is that it is not adaptive. One particular situation when quicksort exhibits its worst-case performance is when input array consists solely of items that are equal. In this case quicksort would still proceed with partitioning the array into sub-arrays, although there is actually no need for

that. In practice, sorting arrays with large number of duplicate items is quite common (e.g. sorting large group of people by their year of birth). Thus, it is an issue which should be definitely dealt with. One approach is to partition the array not in two, but rather into three parts with respect to the value of the selected pivot: items with smaller value, items with larger value and items with equal value (three-way partitioning [52]). Implementing such partitioning is definitely more complicated, but it results in significant improvement of performance as quicksort becomes adaptive to sorting arrays with duplicate data.

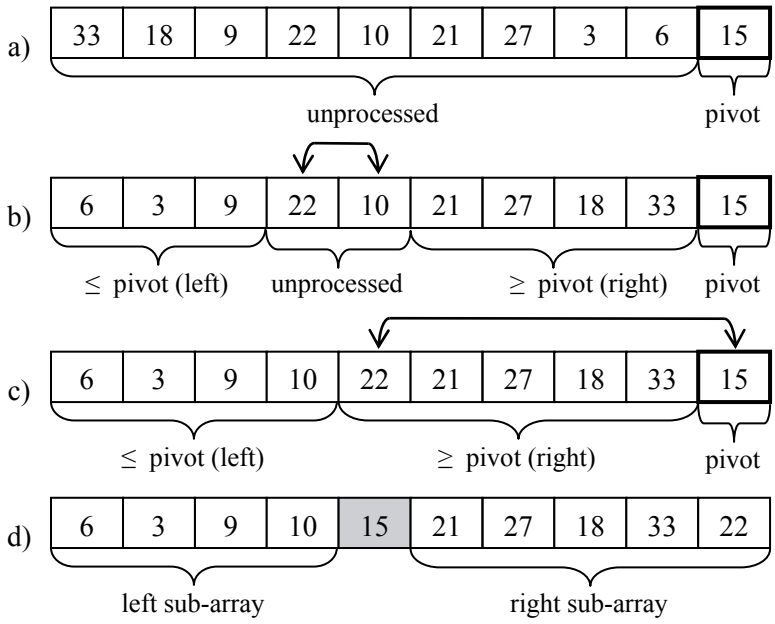



**Fig.2.7.** Quick sort partitioning: a) pivot selection; b) partitioning in progress; c) partitioning is finished; d) pivot is placed to its final position

Other notable features of the quicksort are that it sorts in-place (only a small auxiliary stack for recursion calls is required) and it can also be parallelized due to its divide and conquer nature. Another way to improve the performance of quicksort is to switch to a faster sorting algorithm (e.g. insertion sort) when sub-array size reaches a certain threshold value (usually between 5 and 15) [52].

Introsort is a variation of quicksort designed by David Musser in 1997 [53]. It begins with quicksort and switches to heapsort when the recursion depth exceeds a threshold value based on (the logarithm of) the number of elements being sorted. The switch to heapsort should occur only in case of unbalanced partitioning, thus quicksort's worst-case running time is avoided.

## 2.7. Non-comparison sorting algorithms

All previously discussed sorting algorithms have been comparison-based. So far, the best performance that has been achieved equals to O(n log n). For merge sort and heapsort this actually corresponds to the worst case bound, while quicksort exhibits such performance on average. However, as it turns out, in a worst case scenario there is no comparison-based sorting algorithm that would perform faster than O(n log n) [2]. The reason for this is that a very limited amount of information about the sorted sequence can be gained by using comparisons alone. A good example of a comparison-based sorting is when it is needed to order a set of unlabelled weights using only a scale. In this case information can only be obtained by placing two weights on the scale to see which one is heavier (or the weigh is the same). But there is another group of sorting algorithms, which use information other than that gained from comparisons to determine the sorted order. They are called non-comparison sorting algorithms and the O(n log n) worst case bound does not apply to them.

One such non-comparison sorting algorithm is a counting sort, which was invented by Harold H. Seward in 1954 [54]. The basic idea is to determine for each input element how many items come before it. This information is then used to find a suitable place for the input element in the output array. The algorithm makes no comparisons between input elements, but uses actual values of the elements themselves to produce sorted array. However, this makes counting sort suitable only when the maximum value of keys is not significantly larger than the number of items.

The algorithm operates in three steps as shown in Fig.2.8. In addition to the input array (array A in Fig.2.8) two more arrays are required: one to provide temporary working space (array B in Fig.2.8) and one to store the output (array C in Fig.2.8). Both of the additional arrays are zero-filled in the beginning (Fig.2.8a). The first step is to count the number of times each key value appears in the input array. These values are then saved to the temporary array in positions, which correspond to the value of the keys. For example, key value 7 appears two times. In this case 2 should be saved in the seventh position of the temporary array (Fig.2.8b). This means that the size of temporary array B is determined by the maximum value of keys (in example in Fig.2.8 this is 10). On the next step for each element of the input array its position in the output array is determined. Indexes are calculated by adding to each element of the temporary array B the sum of previous elements (prefix sum) as shown in Fig.2.8c. Finally, elements of the input array can be copied to the output array based on the indexes from the temporary array B. The traversal of the input array begins from the right in order to ensure stability. For example, the rightmost element of the input array A is 7. The seventh element of the temporary array B points to the fifth location in the output array C indicating a

sorted position for element 7 from the input array A (Fig.2.8d). The index in temporary array B is then decremented in case there will be a duplicate item with equal key value, so it would not be saved to the same location.

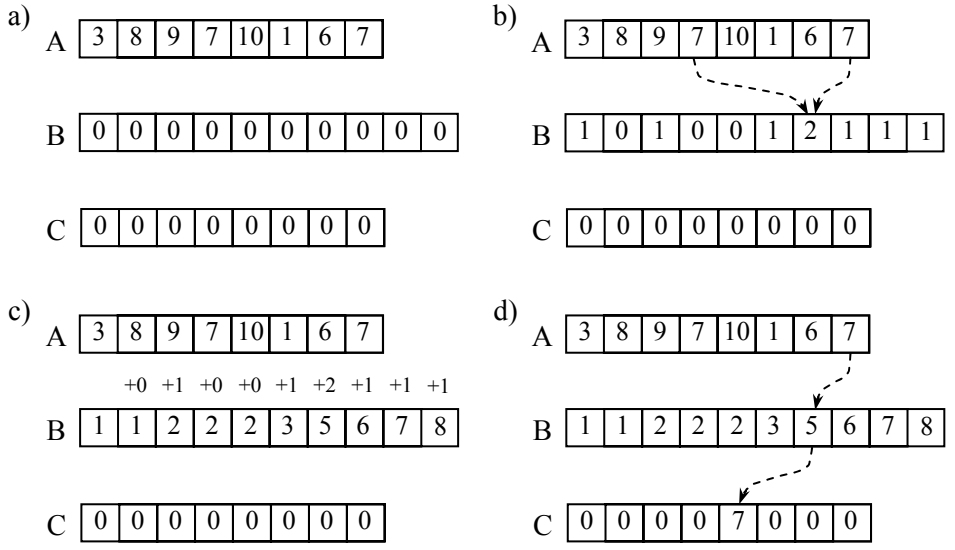



**Fig.2.8.** Counting sort: a) initial condition; b) counting elements; c) calculation of sorted indexes; d) sorting of input array

In contrast to comparison-based sorting algorithms, counting sort does not gain information about the data from comparisons, but by actually studying the data itself (e.g. by knowing the maximum value of keys k). This allows to bring its running time down to O(n+k) (O(n) when n=k). The downside of counting sort is that it does not sort in-place, as in addition to the input array two other arrays are required (of size n and k). However, counting sort is often used as a subroutine for other non-comparison sorting algorithm, radix sort, which can handle larger keys more efficiently.

The radix sort was described in 1954 by Harold H. Seward [54], who proposed to use it in conjunction with counting sort. However, the principle itself dates way back to 1929 [1], when it was used to sort punched cards. The basic idea behind radix sort is to group keys by individual digits, which share the same position. There are two ways to implement radix sort: least significant digit (LSD) radix sort and most significant digit (MSD) radix sort.

LSD radix sort processes keys starting from the least significant digit and proceeds to the most significant digit (while MSD radix sort works the other way around). In order to sort the input array the following steps are performed:

- select the least (most) significant digit;
- group the keys based on that digit by applying any sorting algorithm;
- repeat grouping process for the next more (less) significant digit.

Both radix sort approaches are illustrated in Fig.2.9. For each step the examined digits are highlighted. For each group the radix value is shown below. In each case it takes two passes to sort the input array. In case of LSD radix sort (Fig.2.9a) the grouping should be done using stable sorting algorithm, as the original order of keys must be preserved (in each group the order of elements is exactly the same as in the previous step). The MSD radix sort (Fig.2.9b) allows application of parallel computing, as at each step every group can be sorted independently.

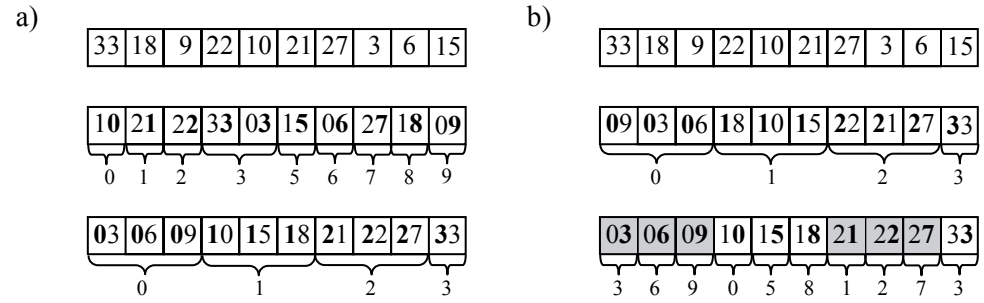



**Fig.2.9.** Radix sort: a) LSD; b) MSD

Radix sort efficiency for input array of size n with length of a key in digits d is O(dn), if an algorithm with linear time is used for grouping step (e.g. counting sort). However, if the counting sort is used as the intermediate sorting algorithm, radix sort obviously does not sort in-place as well.

In 1956 E. J. Isaac and R. C. Singleton proposed another non-comparison sorting algorithm named bucket sort [55]. The basic idea behind this algorithm is to divide the whole set of values that each key can assume into a number of subsets called buckets. Elements of the input array are distributed into these buckets in accordance with their value (Fig.2.10a). Then each non-empty bucket is sorted either using a different sorting algorithm or by recursively applying the bucket sort itself. Finally, the sorted data is gathered from the buckets and is used to replace the original input array with sorted output (Fig.2.10b). Note, that the way bucket sort is implemented in Fig.2.10, it actually corresponds to MSD radix sort.

If the input data is generated by a random process, thus being distributed uniformly, each bucket would contain only a small number of elements, which can be sorted very fast. This scenario would be the most beneficial for bucket sort, as it would have an average running time of O(n). However, the performance of bucket sort degrades if there are many values that are close together, as in this case some

buckets would most likely be completely filled slowing the performance down to as high as $O(n^2)$. The need for auxiliary memory to store the buckets means that sorting is not done in-place.

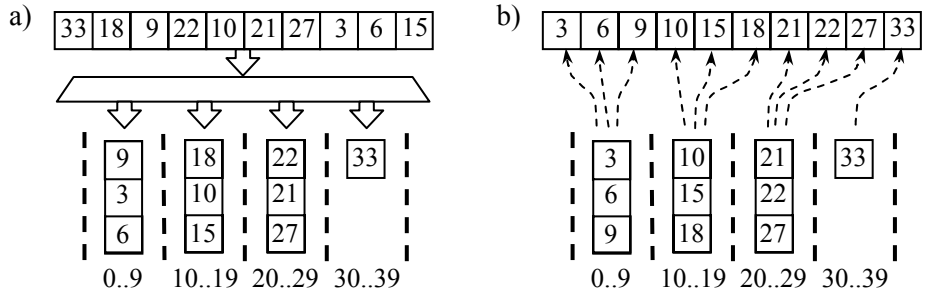



**Fig.2.10.** Bucket sort: a) distribution to buckets; b) gathering of sorted data

One interesting variation of the bucket sort is called pigeonhole sort [90]. It is efficient when the range of keys is approximately equal to the number of items. The main idea is that one bucket is allocated for each possible key value. This way there is no need to apply sorting to buckets after distribution step, as each bucket contains a single element (or a number of duplicate elements). Then buckets can be sequentially read to generate the sorted output. This idea is very similar to the counting sort (but in this case the data itself is moved to the auxiliary array), therefore their performance is also quite the same.

## 2.8. Tree sort

A tree sort is a comparison-based sorting algorithm that uses a tree-like data structure called binary search tree (BST) to produce the sorted output [4,5]. BST is a binary tree, which satisfies the following properties:

- the left sub-tree of each node contains nodes with smaller value of keys;
- the right sub-tree of each node contains nodes with greater value of keys.

The duplicate keys can be treated in different ways. If there is no satellite data associated with the key, it is possible to simply count the number of times a particular key has occurred and save this information in the additional field for each node. Recurring key can also be ignored if they are of no importance. In case equal keys do come with satellite data, the BST property can be modified to place them either to the left or to the right. To preserve stability equal keys should be placed to the right for sorting in ascending order and to the left for sorting in descending order. This actually allows to retain the original order of keys (as in the input array).

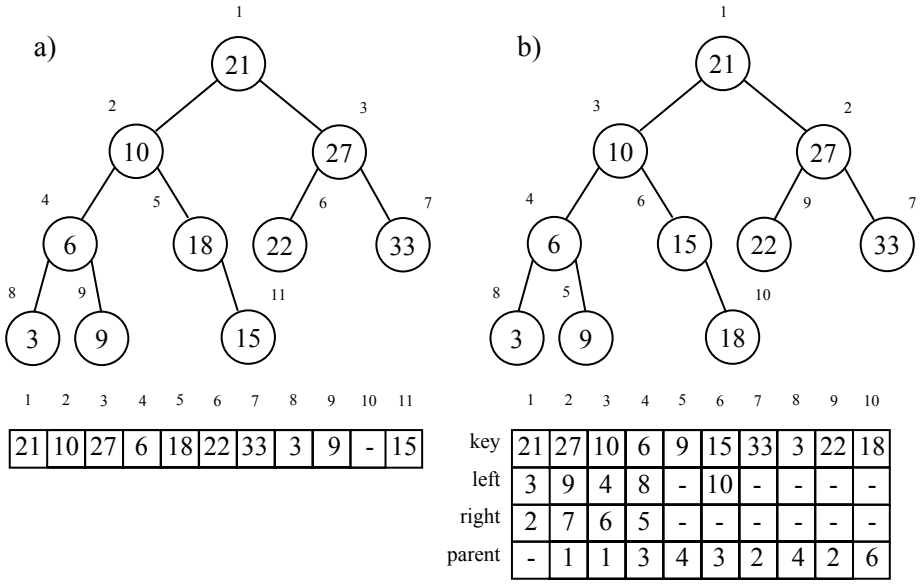



**Fig.2.11.** Binary search tree: a) array representation; b) linked representation

An example of binary search tree is shown in Fig.2.11. BST can be represented as an array (Fig.2.11a). However, unlike heap structure, which is complete by definition, BST does not have this requirement (tenth element of the array is empty). Therefore, for binary search trees a linked representation may be preferable (Fig.2.11b). This requires three pointers to be associated with each key to maintain parent-child relation: pointer to parent node (parent), pointer to left child node (left) and pointer to right child node (right). The absence of a pointer should be indicated with a special code (empty pointer).

After binary search tree has been constructed, the sorting procedure comes down to a simple in-order traversal. The tree sort works as follows:

- recursively apply tree sort to left child node (if it exists);
- output key value;
- recursively apply tree sort to right child node (if it exists).

Note, that sorting procedure does not actually destroy the BST (unlike the destruction of heap during heap sort). This property makes tree sort one of the most efficient methods for incremental sorting, when there is a need to add more items to input array and resort the whole data again. But this also means that sorting is not done in-place.

Binary search tree construction is the dominant process in the tree sort, thus it defines the overall performance of the algorithm (the traversal is done in linear

time). The time required to build a BST actually depends on its balance. Trying to make a binary search tree out of already sorted data results in a worst case performance of $O(n^2)$. On the other hand, a construction of a well-balanced tree is on average a O(n log n) process. The latter case can be achieved by employing a self-balancing binary search tree, which automatically keeps its depth as small as possible during insertions and deletions. However, in practice, tree sort is usually outperformed by other sorting algorithms like heap sort or quicksort when dealing with static arrays.

## 2.9. Sorting networks

A sorting network is a network of wires and comparator modules, which can sort a sequence of numbers. Wires carry values from input to comparator (input wire), from comparator to output (output wire) and from one comparator to the other. The comparator sorts two input values by propagating a smaller value to the upper output and a greater value to the lower output. It is assumed that a comparison is performed in a constant time. The Knuth notation [1] is used to represent the comparator in Fig.2.12.

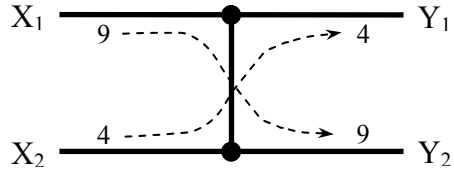



**Fig.2.12.** Comparator element

Sorting networks differ from many comparison-based sorting algorithms due to the fact that the sequence of comparisons is firmly set in advance and they are always executed regardless of the outcome of previous comparisons. Therefore, sorting networks are suitable for relatively short sequences whose length is known a priori.

There are many ways to construct a sorting network. One possibility is to use a traditional comparison-based sorting algorithm. Fig.2.13a shows a sorting network, which is built based on a principle of the bubble sort. By sequentially comparing adjacent numbers the sorting of eight input elements can be done in 28 steps. However, if up to four concurrent comparisons are allowed, then the sorting can be completed in 13 steps, as illustrated in Fig.2.13b. This result is achieved by exploiting the independence of comparison sequences, which enables parallel execution of the algorithm. Therefore, sorting networks are often employed to model sorting algorithms, which actually relay on possibility of parallel comparisons.

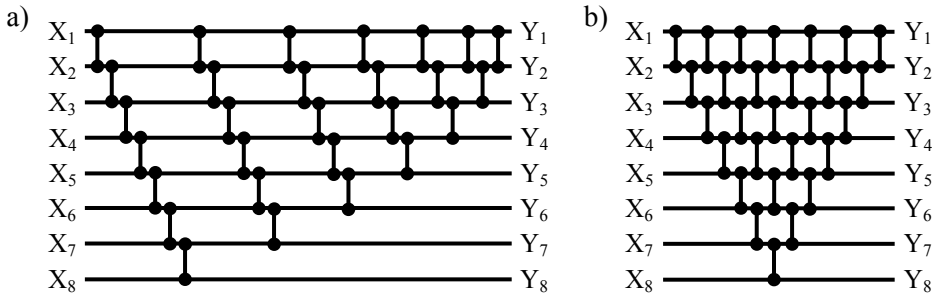



**Fig.2.13.** Sorting network representation of the bubble sort: a) sequential; b) parallel

A more efficient way to implement a sorting network called bitonic sort was proposed by Batcher in 1968 [56]. The core concept of the bitonic sort is the notion of bitonic sequence. A sequence is bitonic if it is a concatenation of two monotonic sequences: one ascending and one descending. Note, that monotonically increasing or monotonically decreasing sequences are also considered bitonic.

The main reason for bitonic sequence being applicable to sorting is the fact that two sorted sequences can be easily merged into a single bitonic sequence, which in turn can be rearranged into a sorted sequence. Suppose there is a sequence $a_1,a_2,\ldots,a_{2n}$ that is bitonic. If it is divided into two sequences $\min(a_1,a_{n+1})$, $\min(a_2, a_{n+2}),\ldots$, $\min(a_n,a_{2n})$ and $\max(a_1,a_{n+1})$, $\max(a_2, a_{n+2}),\ldots$, $\max(a_n,a_{2n})$, then each of these sequences is bitonic and no number of the first sequence is greater than any number in the second sequence [56]. Then by recursively applying the same procedure to each of the halves, a bitonic sequence can be transformed into sorted sequence.

Consider an example of an 8-input bitonic sorter presented in Fig.2.14. Each input can be considered a sorted sequence. Then two inputs are merged into a bitonic sequence and sorted using a 2-input merger (a single comparator). Then two 2-element sorted sequences are merged and sorted using a 4-input merger and so on until the full sorted sequence is generated on the outputs. This provides a systematic method to build a bitonic sorting network of arbitrary sizes.

Bitonic sorting network requires $O(n \log^2 n)$ comparators to implement, has a delay of $O(\log^2 n)$ and is undoubtedly more efficient than the bubble sort implementation discussed earlier. Indeed, an 8-input bitonic sorting network requires 24 comparators and 6 steps, as opposed to 28 comparators and 13 steps required for the parallel bubble sort. Bitonic sorters are frequently used in practice because they have two important properties:

- all signal paths have the same length;
- the number of comparators for each stage is constant.

This is especially useful for implementation in hardware or in parallel processor arrays, as it makes bitonic sorters very predictable, regular, offers good utilization of resources (at each step there is a constant number of required comparisons) and allows easy pipelining.

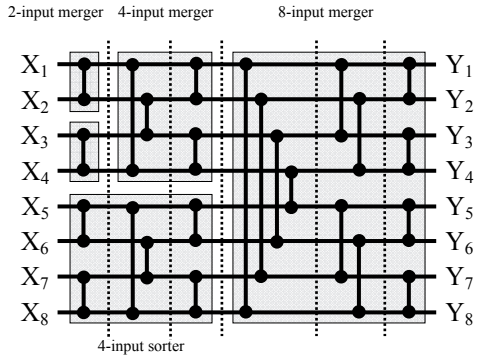



**Fig.2.14.** 8-input bitonic sorting network

An interesting result was published in [57] that proposed a sorting network with a better size and delay (O(n log n) and O(log n) respectively). However, the constants hidden inside the O-notations are actually too big, which makes such networks unsuitable for practical applications [1].

## 2.10. Comparison

In sections 2.2 through 2.9 the brief summaries for the main classes of sorting algorithms have been provided. The most important characteristics of the discussed algorithms are reviewed in Table 2.1. The estimation for the worst case is provided in the column “Running time”.

Now the main question would be which sorting technique is the best? However, the answer actually depends on the particular application and implementation details. For example, such simple algorithm as insertion sort or selection sort obviously should not be used for large arrays, while they are quite effective for short sequences. The performance of shell sort heavily depends on the gap sequence and is very hard to predict. The heapsort always performs in an optimal time for a comparison-based sorting algorithm. However, it has poor locality, thus is not very suitable when caching is involved. The merge sort is also optimal in terms of running time, but it does not sort in-place and is unusable when memory space is very tight. A carefully tuned version of quicksort runs in O(n log n) time on average and is considered the fastest general-purpose sorting algorithm [3], but it cannot be used if stability is important. The effectiveness of non-comparison algorithms may

actually depend more on the nature of input data (e.g. size of keys or data distribution) than on its size.

**Table 2.1.** Summary of the sorting algorithms

| Name | Running time | In-place | Adaptive | Stable |
|---|---|---|---|---|
| bubble sort | $O(n^2)$ | yes | yes | yes |
| cocktail sort | $O(n^2)$ | yes | yes | yes |
| comb sort | $? < O(n^2)$ [(1)] | yes | yes | no |
| insertion sort | $O(n^2)$ | yes | yes | yes |
| shell sort | $? < O(n^2)$ [(1)] | yes | yes | no |
| selection sort | $O(n^2)$ | yes | no | no |
| heapsort | O(n log n) | yes | no | no |
| smoothsort | O(n log n) | yes | yes | no |
| merge sort | O(n log n) | no | no | yes |
| timsort | O(n log n) | no | yes | yes |
| quicksort | $O(n^2)$ | yes | no | no |
| 3-way quicksort | $O(n^2)$ | yes | yes | no |
| introsort | O(n log n) | yes | no | no |
| counting sort | O(n+k) | no | no | yes |
| radix sort | O(dn) | no | no | depends [(2)] |
| bucket sort | $O(n^2)$ | no | no | yes |
| pigeonhole sort | O(n+k) | no | no | yes |
| tree sort | $O(n^2)$ | no | no | yes |

[(1)] For comb sort and shell sort the exact estimation depends on the gap sequence, but the performance is usually better than $O(n^2)$

[(2)] The stability of the radix sort depends on its version: LSD is always stable, while for MSD this is not necessarily true

An important advantage of tree sort compared to other methods is an opportunity of rapid adaptation to eventual modifications of the input array. This is basically 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 item. As a result, any manipulations over tree nodes (e.g. insertion of a new node) are simple and fast [4], while the actual sorting can be done in linear time. The requirement of fast resorting is important, in particular, for the design of priority buffers (queues) and similar devices, which are essential for numerous practical applications [81]. Tree sort is stable, which is essential for maintaining the correct order of the prioritized data. The sorting is not done in-place and requires stack (if sorting is done recursively), but modern FPGA

devices contain plenty of embedded memory blocks that can provide enough space to hold the tree and implement stack. Furthermore, these memory blocks allow to set up arbitrary word length, which simplifies access to data. Also, reconfigurable hardware provides inherent parallelism, which is well suited for implementation of sorting networks.

## 2.11. Chapter summary

In this chapter an overview of the best known sorting techniques was given. The brief summaries and comparison for the main classes of sorting algorithms were presented.

It was shown that it is more advantageous to base the proposed sorting methods on tree-like structures, because they possess a very important advantage of rapid adaptation to eventual modifications. Any manipulations over tree nodes (e.g. insertion of a new node) are simple and fast, while the actual sorting can be done in linear time. This property is very important for fast resorting as it is an essential requirement.

Some ideas from non-comparison based sorting algorithms (especially pigeonhole sort) will be considered as well because of their linear sorting time. Finally, sorting networks will also be employed as the inherent parallelism of FPGAs is very well suited for their implementation.

# 3. Hardware implementation of recursive algorithms

It is known that many computational algorithms can be implemented using various techniques based on recursive specifications. The advantages and disadvantages of recursive techniques in software are well known [2]. Experience in software development shows that recursion is not always appropriate, particularly when a clear efficient iterative solution exists [4]. However, even in software applications, applying recursive algorithms for various kinds of binary search is considered to be a notable exception. In this case a recursive technique is more beneficial than an iterative approach [4]. The advantages of recursion are: clarity of the algorithm, the ease of modifications and improvements (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 recursion can be implemented in hardware much more efficiently. This work concentrates on improvements of circuits implementing recursive sorting algorithms over N-ary trees applying both algorithmic and architectural optimization techniques. The challenge is to use cheap reconfigurable devices to design high-performance sorters adaptable to generally unknown number of input data items.

This chapter describes various techniques to implement recursion in hardware. Section 3.1 provides a general overview. Sections 3.2 through 3.8 give brief summaries and comparison of the suggested methodologies. The conclusions are drawn in section 3.9.

## 3.1. Recursion in hardware

Taking into account the main advantages of recursive specifications it would be worthwhile to use recursion in reconfigurable hardware. Reconfigurable systems are widely used nowadays to increase performance of computationally intensive applications. There exist a lot of synthesis tools that automatically generate customized hardware circuits from specifications in both high-level and hardware description languages. However, such tools have a limited applicability because they are unable to handle recursive functions whereas it is known that recursion is a powerful problem-solving method widely used in computer science. Therefore a

great deal of research effort is aimed at efficient implementation of recursion in reconfigurable hardware as recursive functions are the most time consuming parts in many algorithms and accelerating their execution with reconfigurable hardware would be very beneficial.

Hardware description languages (such as VHDL) as well as system-level specification languages (such as SystemC) that are usually employed for specifying the required functionality of reconfigurable systems do not provide a direct support for recursion. This can be explained by two reasons. First of all, HDL and SLSL descriptions can be synthesized and further implemented over different hardware platforms. These platforms, as a rule, do not possess a dedicated stack memory which could be used for supporting recursive calls. So, the first reason is the lack of hardware support. Obviously, stack could be synthesized specifically for each problem but it is not easy to calculate recursion depth (and, consequently, the required stack size). One alternative solution would be to substitute recursion automatically by the respective iterative specification. However, commercial synthesis tools do not follow this approach because of its inherent complexity. Therefore, the second reason of not synthesizing recursion is the associated complexity. Moreover, for some algorithms, iteration requires more data movement, is less clear, and is often slower [58].

Nevertheless a number of techniques have been suggested aimed at implementing recursion in reconfigurable hardware. An in-depth review and comparison of different approaches to hardware implementation appears in [42]. It has been shown [39] that recursion can be implemented in hardware more efficiently than in software. This is because any activation/termination of a recursive sub-sequence of operations can be combined with the execution of operations that are required by the respective algorithm. The number of states needed for the execution of recursion in hardware can be further reduced compared with software. Besides, such states are accumulated on stacks that can be constructed on built-in memory blocks, which are relatively cheap. The results obtained with some known methods for implementing recursive calls in hardware, reviewed in [42], have shown that many hardware circuits are faster than software programs executing on general-purpose computers.

Basically, all proposals fall into one of two broad categories: unroll recursive calls into a pipelined circuit or implement it with a stack. In the following sections some of the suggested methodologies will be reviewed.

## 3.2. Embedded processor

Making CPU and its various peripheral devices as dedicated chips costs a lot, occupies board space and affects reliability as every solder joint is a potential

source of failure [6]. One way to solve this problem is to embed the whole system on a single chip, for example in FPGA. The processor can be implemented as a hard or a soft core. Hard processor core is a dedicated block within FPGA fabric (e.g. IBM's PowerPC in Xilinx's Virtex-II Pro, Virtex-4, Virtex-5 or Intel Atom E600C paired with Altera's Arria-II). Alternatively, part of FPGA itself can act as a soft processor core (e.g. Altera's Nios-II or Xilinx's MicroBlaze). Both approaches utilize embedded RAM blocks for local memory and use FPGA resources to implement peripherals (timers, buses, controllers, etc.). Soft cores typically have lower performance than hard cores and consume more power, but they can be instantiated as many times as required (or until the FPGA runs out of resources). Soft and hard processor cores can actually co-exist in the design to complement each other, thus providing greater levels of integration and parallelism.

Having a processor core embedded in FPGA has many advantages. First of all, this permits to reuse or port software code in case of design migration (e.g. from systems with discrete processor or to newer FPGA device) to ensure a greater lifespan for a product. Also, the designer has complete freedom to choose any combination of peripherals and controllers, or even to create a custom one when dealing with non-standard requirements of the specification. It is possible to attach several supporting co-processors that can accelerate execution of certain functions in order to enhance performance and efficiency. This provides a great flexibility for hardware/software co-design, which allows to achieve tighter integration between hardware and software.

The major downside of an FPGA-based embedded system is that it may not be ideal for a battery-powered application due to the FPGA being less energy efficient than a fixed low-power processor. Also, FPGAs have a limited support for analog functionality. The system debugging becomes more complicated, as is much harder to figure out the cause of a fault when software and hardware are being developed simultaneously. The FPGA may not be suitable when designing a low-cost product. If the specification requirements can be met using a simple microcontroller, it will be definitely a less expensive alternative. Therefore, the best applications for an FPGA-based embedded system are those that already include or require FPGA functionality [7].

Embedded processors are capable of executing software code that incorporates recursive functions much like a general-purpose CPUs and, thus, face practically the same issues. Stack overflow is the most common problem, because embedded processors have a very limited memory size. This puts an extra pressure on the designer as stack management becomes significantly more complex. The slowness of function calls and returns may be further increased, as embedded processors run on a much slower frequencies than general-purpose CPUs. Many embedded systems are subjected to real-time constraints (must guarantee response within a

strict period). The use of recursion may actually compromise this property when recursion depth cannot be calculated in advance.

## 3.3. Maruyama et al.

One of the first works on implementing recursion in reconfigurable hardware was done by Maruyama et al. [37,38]. In particular, two techniques have been proposed: multi-thread execution and speculative execution.

The first method is applicable to combinatorial optimization problems, which require traversal of the whole search space (the knapsack problem was selected as a case study). It is based on the multi-tread execution approach, a common technique for computer architecture design. For that a given algorithm should be decomposed into a number of pipeline stages. Operations within each stage are executed in parallel. As all pipeline stages become active at the same time, the maximum attainable speedup is limited by the depth of the pipeline. When a recursive call needs to be executed, the arguments are either forwarded directly to the respective stage in the pipeline or pushed into a stack if the stage is occupied. The arguments can then be retrieved when the pipeline stage becomes available.

The second method is aimed at solving combinatorial search when only one (not necessarily optimal) solution has to be found as fast as possible. It is more suitable for loops that include recursive calls (knight's tour problem was considered as an example). The main idea is to speculatively execute consecutive loop iterations in parallel assuming that none of them will make a recursive call. As soon as this assumption fails for iteration i, the computations for subsequent iterations i+1, i+2,..., i+n (n is the number of pipeline stages) are cancelled. The current data are pushed into stack and the loop is restarted from the beginning (simulating a recursive call). If none of the iterations make a recursive call, then data are popped from the stack and execution is resumed at the previously interrupted stage (simulating a recursive return).

In order to automate the process of generating circuits for speculative execution from high-level programming languages, a compiler was developed and reported in [38]. The compiler accepts C code augmented with special notations (such as for specifying the size of data in bits and for identifying statements to be executed in parallel) and generates synthesizable HDL code (based on speculative execution). All recursive calls are previously transformed into iterative loops by a pre-processor. When memory holding data is accessed more than once by different pipeline stages, the pipeline has to be stalled and memory access operations are executed sequentially. The generated circuit speculatively executes next loop iterations and resets/restarts them when data feedback dependencies are detected [38].

The application of multi-thread execution and speculative execution methods seems to be very limited since, in a general case, all pipeline stages can potentially need reading and writing data from/to memory and supporting parallel accesses to different locations of the data memory in parallel is not feasible. Moreover, it is not clear how to proceed if there is a data dependency between the results of a recursive call and subsequent steps of the algorithm.

## 3.4. Sklyarov et al.

Sklyarov et al. proposed a technique for implementing recursive functions in reconfigurable hardware with the aid of hierarchical finite state machines (HFSM) [39,40,59]. For that the algorithm should be divided into a discrete number of functions. The main idea is to implement function calls in hardware in the same manner as it is done in software. Each function is executed by a specific hardware module, which is designed to execute in parallel as many of the required operations as possible. Only one module can be active at a time. During execution a module can transfer control to any other module (perform a call) by pushing its ID, return state and necessary data into stack. When the called module ends, the interrupted state of the system is restored from the stack [59]. This architecture provides a natural support for recursion, as modules are obviously allowed to call themselves as well.

An example of a simple HFSM is shown in Fig.3.1. It has three modules and Module 2 is currently active (it receives inputs and drives the outputs). The preset state of the HFSM is saved on top of two stacks (module stack and state stack). The data stack is not shown in Fig.3.1 as it is not part of the HFSM. It can be seen that currently executing Module 2 was recursively called by itself, as indicated by the last return entry in the stack. The HFSM model will be discussed in more details in chapter 4.

The suggested technique has three main advantages. First of all, this method can be applied for implementing a recursive function with an arbitrary number of calls. Obviously, the maximum depth of recursion has to be known in advance. The only condition that has to be assured of is that a function will not contain an infinite recursion (the same mast be guaranteed in software as well). The second advantage is that parameterizable VHDL templates (discussed in section 4.3) have been developed for the stacks and the combinational circuit composing an HFSM [39,40,59]. Consequently the design process is very easy: it is only necessary to customize the templates for a particular algorithm. And, finally, the same synthesized and implemented circuit can be used for solving various problem instances. This is not possible with the majority of other approaches that require the circuit to be resynthesized (for example, when initial problem data influence the number of times that a function has to be unrolled).

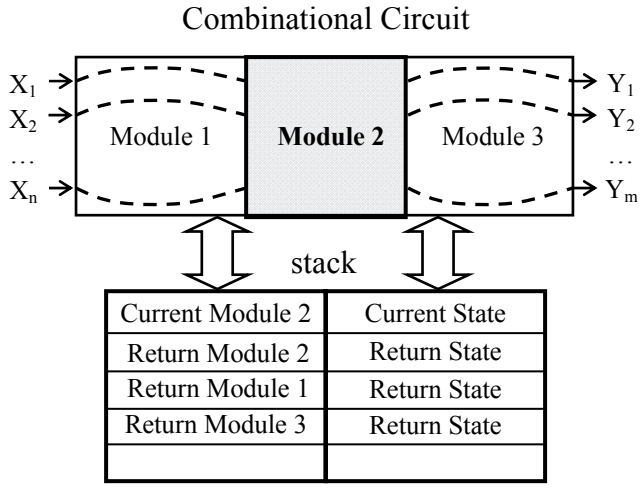



**Fig.3.1.** Hierarchical finite state machine

The main disadvantage of the method is that the parallelism is limited to executing in parallel operations that occur in between recursive calls. If the amount of work in these operations is high, then the HFSM can outperform significantly the corresponding software implementation. Otherwise, if there is a limited number of operations whose execution can be parallelized, then the resulting circuit will require roughly the same number of clock cycles as software running on a single-core microprocessor (actually, the number of clock cycles in hardware will be smaller because invocation of new modules can be overlapped with execution of other algorithm's operations [59]). Since the clock frequency of microprocessors is generally higher than that of FPGA-based systems, the resulting speedup would most likely be negative.

## 3.5. Ninos et al.

The work of Ninos et al. suggests a data-oriented approach [41]. This approach relies on a recursion simplification. The authors claim that this operation can be expanded to as many recursive calls within a function as necessary [41]. Recursion simplification identifies states and conditions which trigger recursive calls, local data to be pushed into stacks and essentially transforms recursion to iteration. If the condition for recursion is met then local data are pushed into the data stack and the function execution is restarted. Otherwise, the function simply continues its execution. When the last statement within the function is reached and the data stack is empty, the execution ends. Otherwise, a recursive return is performed by popping the previous data from the stack. The main idea is that a return state (the one which performed a recursive call) does not have to be saved explicitly as it can be derived

from the restored local data. The basic architecture of a simplified recursive FSM is shown in Fig.3.2.

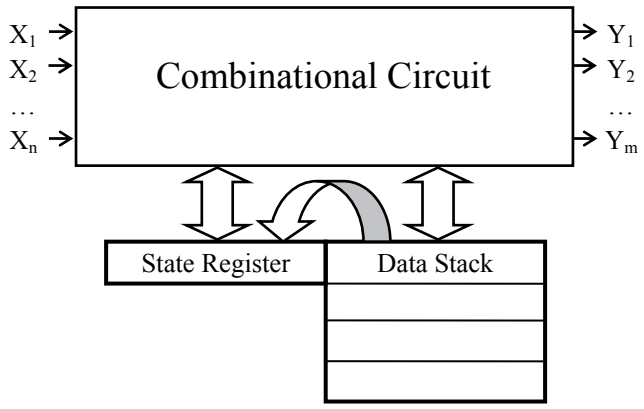



**Fig.3.2.** Simplified recursive FSM

Although in this approach the return state is not explicitly saved, it seems that for most cases in addition to local data some extra information that would allow to identify the point of return from recursive calls should still be stored in the data stack. Actually, this assumption is proved correct in an example for knight's tour problem presented in [41]. Another thing is that on recursive return the control always goes back to the state from which a recursive call has been done. It basically means that this state has to be repeated in order to evaluate the next state to follow, which results in performance degradation.

## 3.6. Stitt et al.

Stitt et al. proposed a new synthesis optimization technique, called recursion flattening, which eliminates recursive function invocations by unrolling and inlining recursive calls [60]. Recursion flattening is realized in two steps. First, the maximum depth of recursion is calculated as a function of a given set of inputs. Then recursive calls are inlined until the required depth of the recursion is reached. A high-level synthesis tool was developed that performs recursion flattening and outputs register-transfer level VHDL code [60].

The suggested technique is not capable of eliminating all recursion but succeeds for some recursive algorithms. Actually, the main difficulty is detecting recursion depth that can only be done if a certain set of conditions is satisfied (such as that every recursive call must modify at least one variable that is used in checking if a base case has been reached). An example of algorithm whose recursion depth cannot be determined statically (before runtime) is quick sort [60]. Another problem

with this method is that various instances of a recursive function (flattened recursion) are synthesized to parallelized circuits [60]. In this case it is not clear how to deal with different instances of inlined functions that require simultaneous access to memory.

## 3.7. Ferreira et al.

All previous methods required either the use of stack, or recursion unrolling before synthesis. Ferreira et al. propose to transform a recursive specification into a tail recursive one [61]. This allows to avoid the need for a stack and does not require recursion depth determination.

The approach relays on the use of a concurrency oriented functional programming (FP) language Erlang [8] to specify the recursive hardware. The system is described with concurrent independent processes, which communicate by message passing. Each process corresponds to a specific hardware block, which is mapped into a finite state machine with datapath (FSMD) model. The use of FP languages for hardware description has been investigated in [62-64], but these studies focus rather on generating a circuit that executes the whole code than providing a separate implementation for each process.

Tail recursion is a special case of recursion when recursive call comes at the end of the function (just before return). In [65,66] it has been proved that all recursive algorithms can be rearranged into a tail-recursive format. The most important property of tail recursion is that it may only require a constant memory space for accumulator variables during execution. This property is essential for making a FSMD version of the algorithm, because it can be represented in an iterative form. Thus, the need for stack is eliminated. The basic architecture of a tail-recursive FSMD is shown in Fig.3.3. Then, the whole system can be represented as a network of communicating FSMDs.

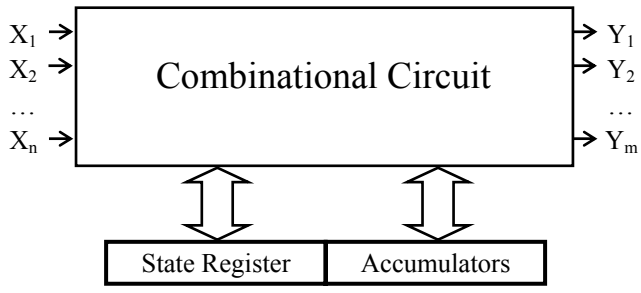



**Fig.3.3.** Tail-recursive FSMD

The elaboration of FSMD from a tail-recursive Erlang functions is simple and has been partially automated. A custom preprocessor that automatically generates a Verilog code from a fully parametrized FSMD template was implemented. However, it is a designer's responsibility to partition the algorithm into concurrent Erlang processes and to organize the communication, to convert recursion to a tail-recursive format and then to customize the FSMD template accordingly.

## 3.8. Comparison

As it was shown in the previous sections all of the suggested methods for implementing recursion in reconfigurable hardware have their advantages and disadvantages. Recursion in an embedded processor is subjected to the same limitations as in case of a general-purpose CPU. Efficiency of multi-thread execution and speculative execution of Maruyama et al. greatly depends on particular problem and is limited by memory bandwidth. The methods that rely on stack (Sklyarov et al. and Ninos et al.) have a restricted parallelism. On the other hand, they are very flexible and can be easily used for implementation of any algorithm. The recursion flattening method of Stitt et al. is not suitable to all recursive algorithms because the maximum recursion depth cannot always be determined. Simultaneous access to memory by different instances of inlined functions can also be a problem. The Ferreira et al. approach relays heavily on the designer to produce the appropriate description in Erlang. Practically all the methods were tested on a relatively small problems and it is very hard to draw any conclusions about their scalability. However, it seems that only the stack-based methods are fully scalable.

The methods differ in the level of supported parallelism [67]. Maruyama et al. [38] explore process-level parallelism (PLP) augmented with pipelining, where multiple instances of recursive function are dispatched simultaneously. The efficiency of this approach is algorithm-dependant and is very restricted by inter-process dependencies and memory bandwidth. Moreover, PLP is difficult to identify automatically [67]. Sklyarov et al. and Ninos et al. explore statement-level parallelism (SLP), where sets of nearby statements are processed simultaneously. The amount of SLP is limited by characteristics of a particular algorithm. Stitt et al. explore pipelining with statements being executed in an overlapped sequence. The maximum amount of achievable parallelism is limited by inter-statement dependencies. The approach of Ferreira et al. actually exploits both PLP and SLP, as the algorithm is implemented as a network of communicating FSMDs. Proposals of Sklyarov et al., Ninos et al. and Ferreira et al. force the designer to identify explicitly the statements/functions to be executed in parallel, while Maruyama et al. and Stitt et al. provide automated high-level synthesis compilers that generate synthesizable HDL code.

This work focuses on circuits that implement recursive sorting algorithms over N-ary trees, which generally should be adaptive to the unknown number of input data items. This makes the Stitt et al. approach unsuitable, as in this case the recursion depth cannot be calculated before runtime. For N-ary trees both the construction and traversal algorithms are quite simple, thus proposals of Maruyama et al. would be ineffective due to the potentially small number of pipeline stages. This also eliminates the embedded processor, as its use would not be justified. Tree maintenance and sorting involve a number of simple operations (e.g. node insertion, node deletion), thus modularity support is preferable. Ninos et al. and Ferreira et al. techniques do not provide this feature. Therefore, the proposed sorting methods rely on the model of HFSM to implement recursion in reconfigurable hardware.

## 3.9. Chapter summary

In this chapter various techniques to implement recursion in hardware were described. Brief summaries and comparison of the suggested methods were presented.

It was shown that it is more advantageous to base the proposed methods on the model of HFSM as it is fully scalable and provides direct support for modularity, hierarchy and reusability, which is especially useful for implementation of tree maintenance functions.

# 4. Hierarchical finite state machine

In order to describe the behavior of a control unit we can apply various forms of behavioral model. Finite state machines (FSM) are probably the most widely used components in digital circuits and systems. That is why almost all the available automatic design tools that are included in industrial CAD systems allow FSMs to be synthesized from their formal specifications [9], such as state diagrams, state transition tables, HDL descriptions, etc. All these specifications are appropriate for the design of relatively simple circuits. For more complex circuits it is very important to provide support for enhanced features, namely hierarchical description of FSM functionality at different levels of abstraction.

The model of hierarchical finite state machine (HFSM) can be considered as an example of such more advanced FSMs. Its specification is based on the hierarchical graph-schemes (HGS), which can be seen as an extension of graph schemes. Any HGS provides support for modularity (each individual HGS is considered to be a module that, in general, can be reused), hierarchy and parallelism. HGSs permits to develop any complex algorithm step by step, concentrating efforts at each stage on a specified level of abstraction [68]. Such specification is more readable and provides direct support for reusability.

This chapter is devoted to the model of hierarchical finite state machine. Section 4.1 provides description of hierarchical graph schemes. Section 4.2 gives an overview of the HFSM model. Section 4.3 suggests guidelines for synthesis of HFSM model. Section 4.4 shows applicability of HFSM model to reuse techniques. Section 4.5 presents some practical tasks where the use of HFSM model is advantageous. The conclusions are drawn in section 4.6.

## 4.1. Hierarchical graph schemes

An HGS is a directed connected graph containing rectangular, rhomboidal and triangular nodes. Each HGS has one entry point, which is a rectangular node named “Begin”, and one exit point, which is a rectangular node named “End”. Other rectangular nodes contain either micro instructions or macro instructions (hierarchical calls), or both. Assignment of micro instructions to the nodes “Begin” and “End” is also allowed if necessary. Any micro instruction $Y_j$ includes a subset of micro operations (output signal, which causes a simple action in the datapath) from the set $Y=\{y_1,...,y_N\}$. Any macro instruction incorporates a subset of macro operations from the set $Z=\{z_1,...,z_F\}$. Each macro operation is described by another

HGS of a lower level considered as a module. These modules are usually very simple, and can be tested and debugged independently. If a macro instruction includes more than one macro operation than all these macro operations have to be executed in parallel (in this work it is assumed that each macro instruction includes just one macro operation). Each logic function is calculated by performing some predefined set of sequential steps that are described by an HGS of a lower level. Each rhomboidal node contains one logical condition from the set $X=\{x_1,...,x_L\}$. A logic condition is an input signal, which communicates the result of a test. All other details can be found in [68,69].

An example of hierarchical graph schemes is shown in Fig.4.1. The algorithm is divided into three different modules ($Z_0$, $Z_1$, and $Z_2$). An execution starts from the node "Begin" of the main module $Z_0$. Module $Z_1$ can be called if condition $x_1$ is met. Similarly, module $Z_1$ can activate module $Z_2$ when both conditions $x_2$ and $x_3$ are true. Note, that module $Z_1$ can also recursively call itself. The execution of a module is terminated when the node "End" is reached. At this time the control is returned to the calling module and it continues execution from the interrupted step. The algorithm terminates at the node "End" of $Z_0$.

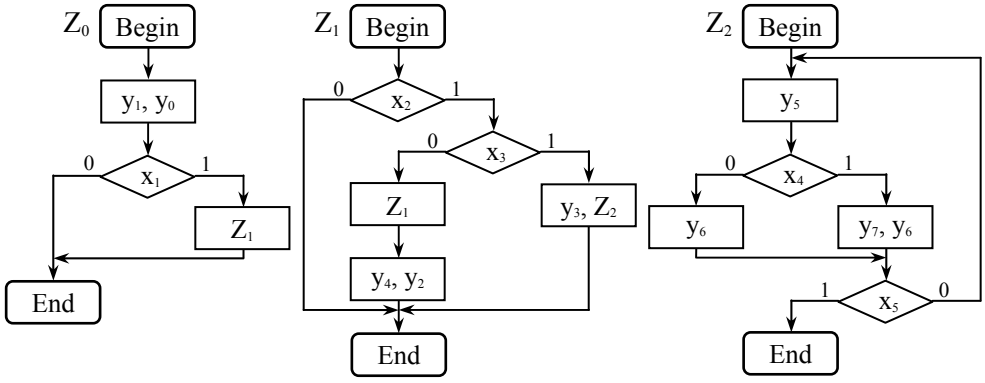



**Fig.4.1.** Hierarchical graph schemes

For a number of practical applications HGS can easily be constructed from specifications in general-purpose languages, such as C/C++ [68]. An HGS can be converted to the respective control circuits modeled by HFSM and then formally described in a hardware description language. The resulting HDL code can be synthesized in any commercially available CAD system.

## 4.2. Models of hierarchical finite state machine

The model of HFSM [68,69] permits the modules described using HGS to be implemented in hardware. Furthermore, this model provides direct support for

modularity and recursion. There are two types of HFSMs [69]: HFSMs with explicit modules and HFSMs with implicit modules.

The model of HFSM with explicit modules (Fig.4.2) has the following distinctive features. The current state of the system is defined with active module and its state (global 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 global state of control unit is restored from the stacks). This model does not permit to apply the majority of optimization methods developed for conventional FSMs.

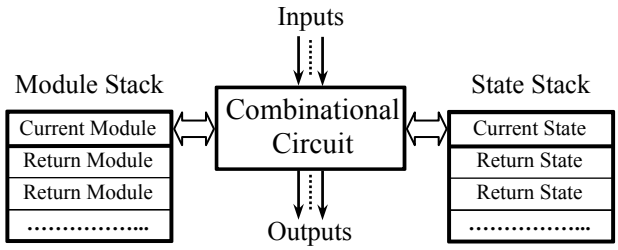



**Fig.4.2.** HFSM with explicit modules

The second model of HFSM with implicit modules (Fig.4.3) 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). In any module all the necessary state transitions are executed through a register, much like it is done in a conventional FSM. The stack is needed just to know which state has to be the target of the transition when a called module is terminated [11].

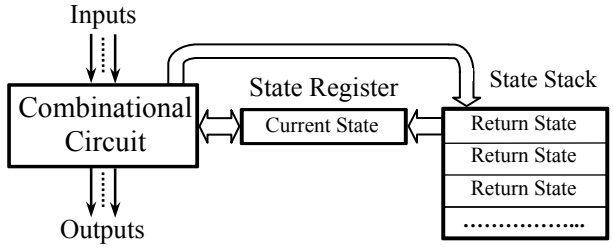



**Fig.4.3.** HFSM with implicit modules

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 (compared to the length of the original state code). Similarly, during hierarchical return the content of the stack is decoded before being placed into the state register. This is especially useful, as this allows to implement the optimized stack using LUTs of FPGA device and conserve block memory. Since the codes of all states are unique and the modules are hidden (implicit), all known optimization methods that are used for conventional FSMs can be applied directly. Experimental results (see section 6.1) demonstrate that the HFSMs with implicit modules are faster and less resource consuming than HFSMs with explicit modules.

Practicability of these models can be further extended by applying either reconfiguration capabilities (the same hardware is used for implementing different algorithms) or parallel execution of macro operations (multiple modules are working at the same time). Reconfigurable hierarchical finite state machine (RHFSM) [70] can be reconfigured both statically and dynamically. The modules are implemented on the basis of memory blocks. The reconfiguration is done by reloading the content of the memory blocks, thus altering the functionality of RHFSM. Parallel hierarchical finite state machine (PHFSM) [71] permit to implement algorithms composed of modules that can be activated in parallel. There are K stacks connected to a common combinational circuit (the number K is equal to the maximum number of modules running in parallel). If two or more modules are called in parallel from the module $Z_a$, the module $Z_a$ is allowed to continue its execution if and only if all called parallel modules have been terminated. If any of parallel modules is still functioning, the module $Z_a$ has to remain suspended. Both of these models fall beyond the scope of this work and will not be considered.

## 4.3. HFSM implementation using HDLs

Synthesis and implementation of HFSM from HGS includes the following steps:

- dividing functionality of the algorithm into modules $Z_0,\ldots,Z_{F-1}$, where F is the number of required modules;
- marking the states for each module with labels $a_0,\ldots,a_{H-1}$, where H is the maximum number of labels that were used for any individual HGS;
- describing HFSM state transitions, output signals and stacks using a hardware description language (VHDL will be used);
- synthesis of the circuit from a hardware description language description using any commercially available synthesis software;
- implementation of the circuit in hardware (e.g. in FPGA).

The design flow will be demonstrated on an example. Note that provided VHDL descriptions are used to show only the basic principles of HFSM specification with

many implementation details being omitted for the purpose of simplicity. The model of HFSM with explicit modules will be discussed first.

Suppose the algorithm is transformed into HGSs as shown in Fig.4.1. There are three modules in total ($Z_0$, $Z_1$, $Z_2$), but in this example only $Z_1$ will be considered (other modules can be implemented in exactly the same way). Suppose the states of $Z_1$ are labeled as shown in Fig.4.4. Then both transition and output functions for this module can be described in almost exactly the same way as it is done for a conventional FSM (Fig.4.4). For transitions that involve solely the change of state inside the module, only the next state is generated (states $a_0$ and $a_3$ in Fig.4.4). In case of hierarchical call the return state (specified as next state), the next module and push request to the stacks are generated (states $a_1$ and $a_2$ in Fig.4.4). When the last state (state $a_4$ in Fig.4.4) is reached, the execution of the module is terminated and the pop request is issued in order to restore HFSM back to the interrupted state. The pop request is generated only if the stacks are not empty, so it would not be issued when the algorithm terminates. The module $Z_1$ does not terminate the execution of the algorithm ($Z_0$ does), thus this condition can be basically omitted. When all modules are described, they are grouped together with two-level VHDL case statement (Fig.4.4).

```
case current_module is
  when Z0 => ...
  when Z1 => case current_state is
               when a0     => if X(2) = '0' then
                                next_state <= a4;
                              elsif X(3) = '0' then
                                next_state <= a1;
                              else
                                next_state <= a2;
                              end if;
               when a1     => next_state <= a3;
                              next_module <= Z1;
                              push <= '1';
               when a2     => next_state <= a4;
                              next_module <= Z2;
                              push <= '1';
                              Y(3) <= '1';
               when a3     => next_state <= a4;
                              Y(4) <= '1'; Y(2) <= '1';
               when a4     => if stack_empty = '0' then
                                pop <= '1';
                              end if;
               when others => NULL;
             end case;
  when Z2 => ...
end case;
```

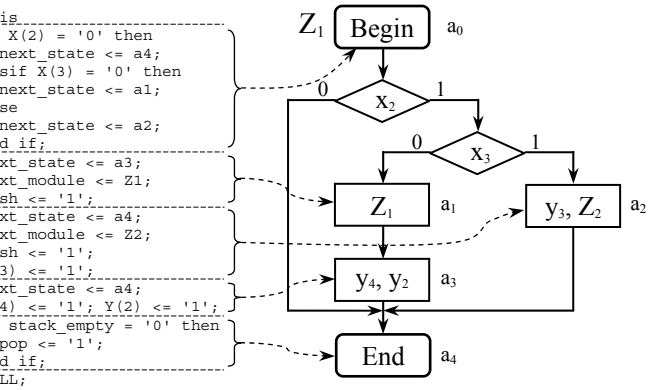



**Fig.4.4.** Description of explicit HFSM module in VHDL

Note that in the model of HFSM with explicit modules the states of all modules are assigned the same labels. Therefore, in case when not all of them have been used, it should be explicitly indicated that for the remaining labels no action is needed (requirement of synthesizable VHDL).

In order to complete the specification of an HFSM, the stacks are needed to be described. In this work the tops of the module and state stacks (where the current global state is stored) have been implemented as separate registers. The VHDL description of both registers and stacks is shown in Fig.4.5. In case of hierarchical call (unless the stacks are full) the code of the next module is saved in the module register and the current state is set to $a_0$ (the "begin" state in all modules should be labeled with $a_0$). The return state (specified as next state in the combinational circuit) and the return module (the module that made a call) are pushed into stacks. The stack pointer is also incremented at the same time (not shown in Fig.4.5). Therefore the return global state should be saved in the location that will be pointed to by the incremented stack pointer (stack_pointer+1). When the module terminates, the global state is restored from the stacks (unless the stacks are empty). At the same time the stack pointer is decremented (not shown in Fig.4.5). For transitions that involve solely the change of state inside the module only the state register is updated with the next state. This description of stacks is valid for any algorithm (unlike combinational circuit that is basically unique for each algorithm). The only variation may be the size of stacks that can be different depending on the depth of hierarchy (recursion).

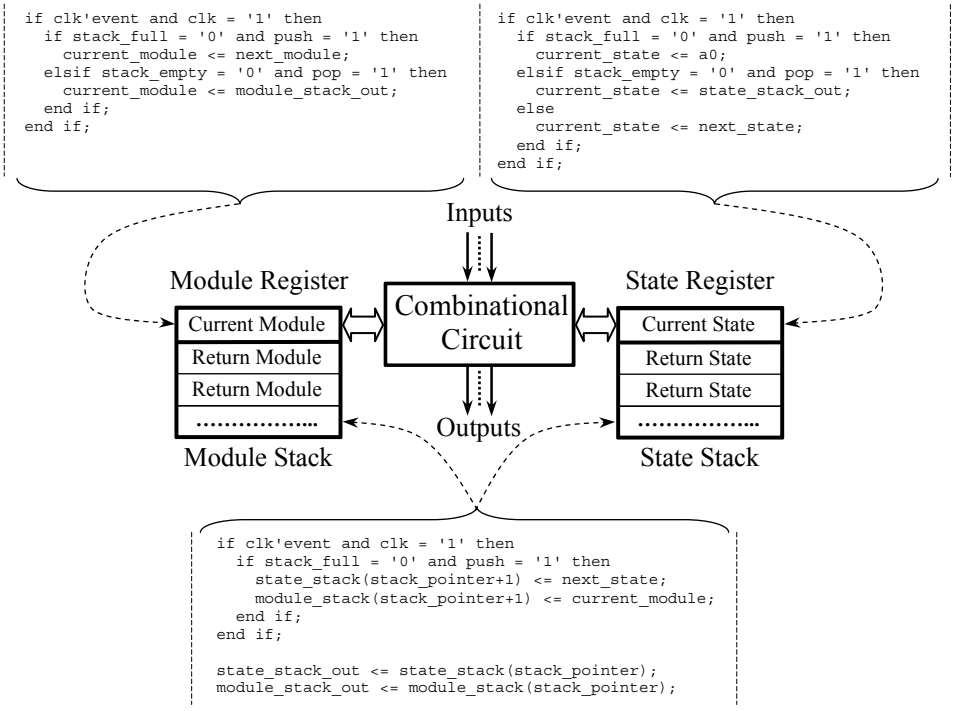



**Fig.4.5.** Description of stacks for HFSM with explicit modules in VHDL

The model of HFSM with implicit modules will be discussed next. The same example (module $Z_1$ from Fig.4.1) will be used.

The main difference is that now each state in each module is assigned a unique code that is a concatenation of the module and state codes. Therefore, both transition and output functions for the whole algorithm can be described in almost exactly the same way as it is done for a conventional FSM (Fig.4.6). For transitions that involve solely the change of state inside the module, only the next state is generated (states $Z_1a_0$ and $Z_1a_3$ in Fig.4.6). In case of hierarchical call the return state, the next state and push request to the stack are generated (states $Z_1a_1$ and $Z_1a_2$ in Fig.4.6). When the last state (state $Z_1a_4$ in Fig.4.6) is reached, the execution of the module is terminated and the pop request is issued in order to restore HFSM back to the interrupted state. The pop request is generated only if the state stack is not empty, so it would not be issued when the algorithm terminates. The state $Z_1a_4$ does not terminate the execution of the algorithm, thus this condition can be basically omitted.

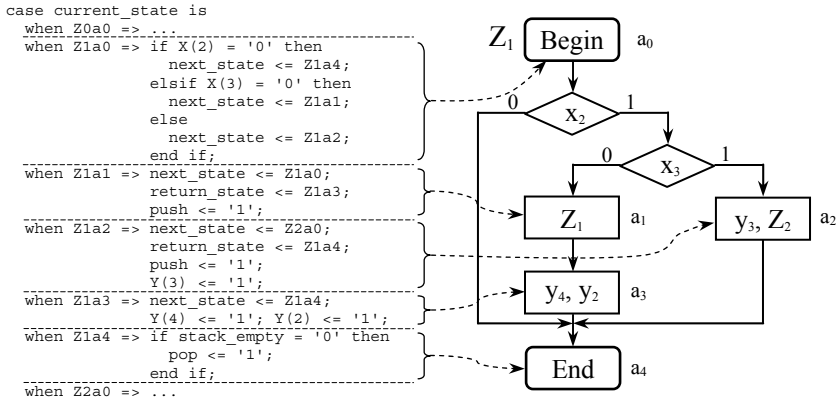



**Fig.4.6.** Description of implicit HFSM module in VHDL

Note that for both models during hierarchical return the control is passed to the state that follows the calling state. This is because the transition from the calling state to the next state is unconditional. In case this transition actually depends on a certain condition, the control should be passed back to the calling state. In order to avoid the second activation of the same operations, the return flag must be employed. This technique is thoroughly described in [39].

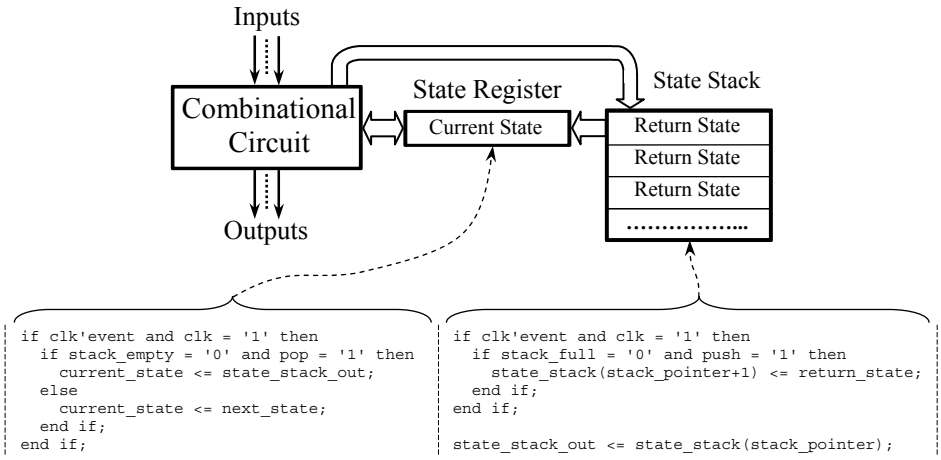



**Fig.4.7.** Description of stacks for HFSM with explicit modules in VHDL

HFSM with implicit states has one state register and a single stack. The VHDL description of both register and stack is shown in Fig.4.7. In case of hierarchical call (unless the stack is full) the return state is pushed into stack and the next state is saved in the state register. The stack pointer is also incremented At the same time (not shown in Fig.4.7). Therefore the return state should be saved in the location that will be pointed to by the incremented stack pointer (stack_pointer+1). When the module terminates, the state is restored from the stack (unless the stack is empty). At the same time the stack pointer is decremented (not shown in Fig.4.7). For transitions that involve solely the change of state inside the module only the state register is updated with the next state. This description of register and stack is valid for any algorithm. The only variation may be the size of stack that can be different depending on the depth of hierarchy (recursion).

## 4.4. Reuse technique with HFSMs

The design of a modern digital circuit is not an easy task, as complexity of such devices is constantly increasing. Developing engineering systems on the basis of high capacity FPGAs puts forward a fundamental question – how to cope with rapidly growing complexity and how to efficiently use enormous and continuously rising hardware resources in the design process in particular [72]. This is actually very important because according to the Moore's law every two years the density of microelectronic devices is generally doubled. The problem is that the number of available transistors grows faster than the ability to meaningfully design with them. This situation is a well known design productivity gap, which was inherited by FPGA from ASIC and which is increasing continuously. Therefore the design productivity will be the real challenge for future systems.

One possible answer to the above question is to apply a reuse technique and an evolutionary strategy in such a way that permits parameterizable and highly optimized project components to be repeatedly utilized in the scope of the same project or possibly in future projects as well. By employing the HFSM model reusability can be achieved at the level of specifications. This allows to describe fragments (modules) in such a way that the developed algorithm can be composed of either novel or previously designed modules providing reuse on project scale (design hierarchy).

For example, the SD card controller, which has been developed for EEG Analyzer [12] project, has been implemented using HFSM. The Secure Digital Memory Card is the de facto standard memory card for mobile equipments. The SD card standard is designed and licensed by the SD Card Association [91]. The functionality of the developed controller is limited to card's identification (MMC, version 1.0, version 2.0 and high capacity cards are supported), initialization and capability of reading a text file with EEG data. The SD card controller hardware design greatly benefited from HFSM-based implementation as:

- it allowed to describe the system incrementally beginning with simple modules and continuing with creating more complicated modules from the modules that have already been developed, verified, and tested in hardware (Fig.4.8, from Level 2 up to Level 1);
- it permitted to employ the ideas of software solutions in hardware implementations due to support of similar execution mechanisms;
- it enabled to design simpler and faster hardware circuits, which would require less resources.

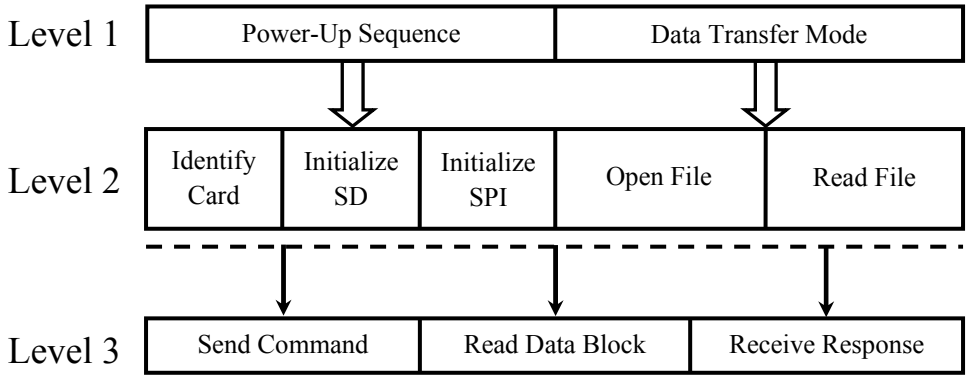



**Fig.4.8.** Hierarchical implementation of SD card controller

Frequently the same sequence of operations needs to be reused in different specifications. In such cases the sequence has been assigned to a reusable module that was executed when necessary through hierarchical calls (Fig.4.8, Level 3

functions). For example, the procedure of sending a command to SD card was implemented as a separate reusable module. The command code and its argument were supplied as parameters during the call.

## 4.5. Practical examples

Practical applications of HFSMs were considered in numerous publications.

The results of [68] were applied to the design of hardware and software for complex data processing operations in [10] (chapter on hierarchical finite state machines and their use in hardware and software design), for models of computations described in [73], for control system in Medusa instrument [74].

HFSMs can be used at different levels, for example for local control in [75] and for implementation of relatively complex embedded systems, like a garage controller that supports automatic parking of arriving cars and driving them to the garage exit on requests [76].

A number of practical applications require software components to be implemented in hardware circuits. In [77] HFSMs are used for reconfigurable SoC design where hierarchy is important at system level.

Many papers are dedicated to synthesis of HFSMs from scenario-based notations such as UML (e.g. [78,79]) for software development. Statecharts [80], which can be seen as another type of specification for HFSMs, were adapted for object-oriented programming and used as a part of unified modeling language (UML).

## 4.6. Chapter summary

This chapter was devoted to the model of hierarchical finite state machine (HFSM). It was shown that the model of HFSM can be effectively used to implement recursion in hardware.

Chapter provided description of hierarchical graph schemes, HFSM models and guidelines for implementation of HFSM in hardware. Known model of HFSM with explicit modules and a new model of HFSM with implicit modules were described. A number of optimization techniques for the new model of HFSM were suggested.

It was also demonstrated on a case study that HFSM model is applicable to reuse techniques. Some practical tasks where the use of HFSM model is advantageous were also mentioned.

# 5. Hardware implementation of sorting algorithms

Among numerous tasks that need to be solved, sorting is considered to be one of the most important [2]. Since it is time consuming for large volumes of data, acceleration is greatly required for many practical applications. It is also important to discover such methods that take advantage of the implementation platform (due to its uniqueness) and consider not only the number of the required operations, but also efficiency of these operations in hardware circuits [36]. Reconfigurable hardware provides inherent parallelism, which is well suited for implementation of multiple processing units that are working together. FPGAs allow to instantiate the same sorter (or different sorters) as many times as required (or until it runs out of resources). Thus, it is very important to develop such methods that take full advantage of this particular capability. This chapter presents a number of simple, but efficient sorting techniques that are particularly useful for implementation in FPGAs. The emphasis is done on applications that involve fast processing of new incoming data items, such as resorting.

The remainder of this chapter is organized as follows. Section 5.1 provides description of hardware implementation approaches for sorters. Section 5.2 gives an overview of the proposed sorting techniques over binary trees. Section 5.3 is devoted to hardware architectures that process binary trees in parallel. Section 5.4 suggests a compression method for binary trees. Section 5.5 introduces address-based sorting technique. Section 5.6 deals with sorting over N-ary trees. Section 5.7 presents the concept of multi-level sorting. The conclusions are drawn in section 5.8.

## 5.1. Hardware implementation of a sorter

The sorter architectures basically fall into two categories depending on the source of input data: memory-based and stream-based.

In a memory-based architecture (Fig.5.1) the input data are usually kept in memory common to traditional computers. In this case the sorted sequence either replaces the original (unsorted) data in memory, or is saved in a separate memory segment.

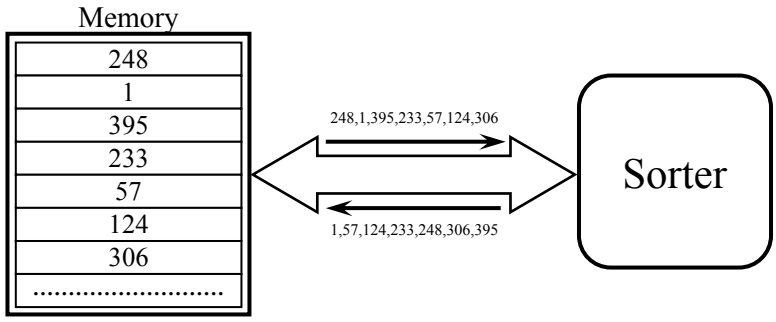



**Fig.5.1.** Memory-based sorter architecture

The input data can also be represented in form of incoming streams (Fig.5.2) that are dynamically generated from different sources (e.g. distributed sensors in networked embedded systems). In this case the sorted sequence is usually transmitted to another device.

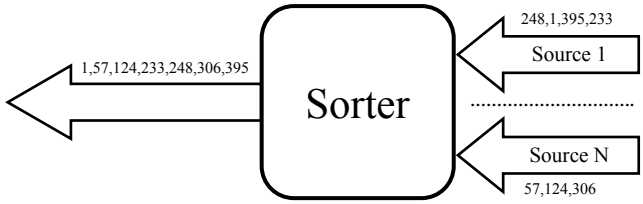



**Fig.5.2.** Stream-based sorter architecture

Physically sorters can be used differently. For example, they can be connected through a system bus of a general-purpose computer and access computer memory (that is a source of data) through allocated windows in memory space. Alternatively, they can be seen as standalone accelerators getting external packages of unsorted data and outputting sorted sequences. In some practical applications data have to be resorted dynamically as soon as a new data package/item is received [81].

## 5.2. Sorting over binary trees

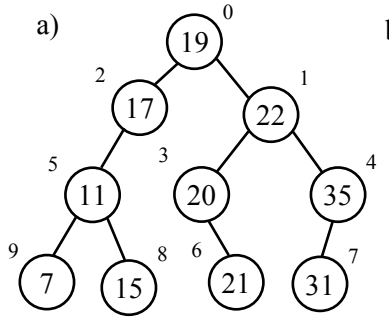
The use of binary trees for sorting data in hardware circuits is considered in [39] and it is based on the following technique. Suppose each node of the tree contains three fields: a value (e.g. an integer), a pointer to the left child node (LA), and a pointer to the right child node (RA). The absence of a node is indicated by a specially allocated code. 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. For the purpose

of simplicity the equal values are ignored. The support for duplicate data items can be easily introduced (section 2.8).

An example of a binary tree is presented in Fig.5.3a in form of a graph and in Fig.5.3b in form of a linked list (as it is kept in memory). For each node in Fig.5.3a the value and the relevant address in memory are shown. The first column of Fig.5.3b specifies memory location, where the node (the list item) is stored. The other columns keep value (Data), left (LA) and right (RA) addresses according to the format mentioned above. 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 methods [39] permit to construct a binary tree from incoming data items and to output the sorted data from the tree. The search for the proper place for a new data item is done as follows:

- compare the new data item with the value of the root node to determine whether it should be placed in the sub-tree headed by the left node or the right node;
- check for the presence of the selected node and if it is absent, create and insert a new node for the data item and end the process;
- repeat previous steps with the selected node as the root (clearly recursion can naturally be applied here).




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 |

**Fig.5.3.** Binary tree represented as (a) graph and (b) linked list

When the binary tree construction is complete, the ascending sorted sequence can be generated by applying inorder traversal. The traversal starts from the root. Its left sub-tree is examined first. If there is a left child node, then it is set as a root and the whole procedure is recursively reapplied to it. Otherwise, the root is examined and its data value is propagated to the output. Finally, the right sub-tree is

examined. If there is a right child node, then it is set as a root and the whole procedure is recursively reapplied to it. Otherwise, the processing of the previous node is resumed. This method will be further referred to as the known algorithm and it will serve as the 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.

Suppose that the tree in Fig.5.3a is stored in a dual-port memory as it is shown in Fig.5.4. The binary tree is in the middle of the traversal process and node "22" (in a bold circle) is currently being processed. 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 with a dashed circle in Fig.5.4) 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.

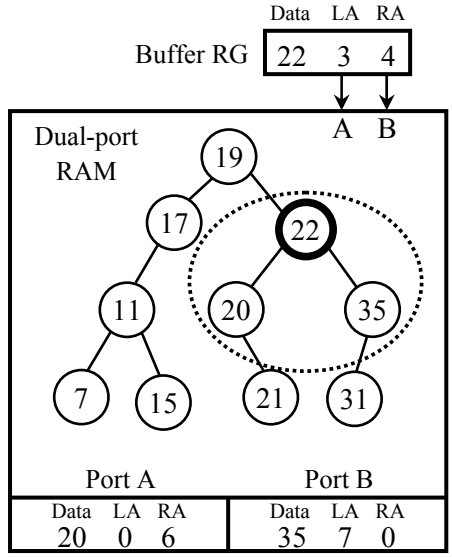



**Fig.5.4.** Binary tree in a dual-port memory

This approach can be further improved by examining pointers of child nodes individually as well. Consider the situation presented in Fig.5.4. 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 “21” next.

## 5.3. Parallel sorting over binary trees

Consider the known method for recursive data sorting over binary tree described in the previous section. FPGAs allow to put multiple instances of the same algorithm to work on different parts of the tree. Thus, the most obvious choice to parallelize the known method involves parallel traversal of the sub-trees, which are to the left and to the right of the root. Naturally, more parallel branches can be introduced using cascade structures of more than two sorters that are activated for different sub-trees on certain paths from the main root.

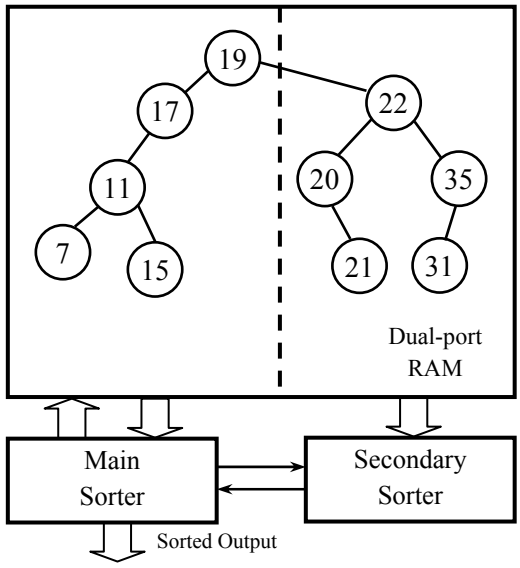



**Fig.5.5.** Main/secondary sorter architecture

The idea behind main/secondary sorter architecture is the following. There are two simultaneously functioning digital circuits that are a main sorter and a secondary sorter (Fig.5.5). 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. The tree itself can be built in a dual-port memory, which would allow simultaneous access for both devices. However, this makes such architecture not well-suited for improved versions of the known method

described in previous section, as they also relay on the use of dual-port embedded memory blocks to speed-up the tree traversal efficiency (sharing of ports is required).

Main/secondary sorter architecture has 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 completed, the processed nodes of the right sub-tree can be read from the temporary storage.

Intuitively one can guess that the performance of main/secondary sorter architecture depends 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 counterpart.

The balance dependency can be eliminated using the following technique. The main sorter activates the secondary sorter only if there is a sufficient number of processing steps. For that purpose each node of the tree is provided with two additional fields indicating the number of nodes in the left and in the right sub-trees accordingly (such fields can easily be filled in during the construction of the tree). Both sorters begin their job at the same time and repeat the same steps to remember the way from the root for backward propagation. For example, if the number of nodes in the left sub-tree is greater than the number of nodes in the right sub-tree, than main sorter begins a standard inorder traversal. In each tree node it evaluates the number of nodes for forward propagation to the left and to the right. As soon as the difference reaches some predefined value, the main sorter takes responsibility for sorting of the last root and the left sub-tree and instructs the secondary sorter to continue sorting with the remainder of the tree. A similar procedure is used when the right sub-tree contains more nodes than the left sub-tree. However, this time the main sorter examines the right sub-tree first. As soon as the difference reaches some predefined value, the main sorter instructs the secondary sorter to continue sorting from this point and takes responsibility of sorting remainder of the tree by backtracking to the root [13].

Balance dependency can also be eliminated with distribution of the incoming data between N>1 parallel sorting circuits. Each sorter unit traverses its own independent tree, while the results are mapped from the circuits to a sorted sequence. This architecture can be easily customized for different values of N [14].

For example, let N be equal to three (N=3). In this case the input data is distributed in such a way that the first, the fourth (N+1), the seventh (2N+1), etc. incoming data items are included into the first tree. Consequently, the second, the

fifth (N+1)+1, the eighth (2N+1)+1, etc. incoming data items are included into the second tree and the third, the sixth (N+1)+2, the ninth (2N+1)+2, etc. incoming data items are included into the third tree. The process is repeated until all data items are distributed. Thereby, each sorter unit constructs and traverses its own independent tree. This procedure is handled by a top-level manager (TLM). Fig.5.6 shows N=3 trees that are built for the same data set as in example from Fig.5.3.

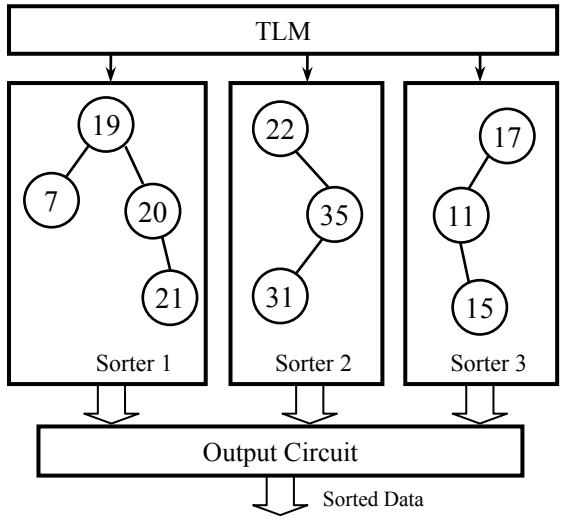



**Fig.5.6.** Parallel sorting of N independent binary trees

Compare binary trees of Fig.5.3a and Fig.5.6. The maximum depth of the tree in Fig.5.3a is four and the maximum depth of the trees in Fig.5.6 is three. Since the time needed for construction and sorting depends on the depth of the trees [39], the performance of both building and sorting algorithms that process data represented by a set of trees (Fig.5.6) is expected to be better.

As soon as the trees are constructed and each tree is stored in the relevant processing memory (Fig.5.6), the TLM instructs the output circuit to generate the sorted data sequence using the following method:

- all trees are traversed in parallel using any method. Each sorter is connected to a dual-port output buffer. Sorted data items are saved in the output buffer using the first port;
- the output circuit checks each output buffer for the presence of unprocessed data items using the second port. When each buffer contains at least one unprocessed data item, the smallest one (or the greatest one, depending on the sorting strategy) is extracted.

In this way the data output can be executed in parallel with tree traversal. 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.

The proposed strategy bears similarities to the merge sort discussed in section 2.5. In the same manner the input data is divided into separate sets (trees) that are processed independently and then merged together to produce a single sorted output. However, proposed implementation results in the loss of stability due to the employed method of data distribution (unlike merge sort that is stable).

## 5.4. Binary tree compression

In order to accelerate data processing and reduce memory consumption, a compression method using positional encoding for tree-like structures can be employed [15]. 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. The flag is set to "1" if the matching data item has been added to the group. It remains "0", otherwise.

A compressed version of the example binary tree from Fig.5.3a is presented in Fig.5.7. 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 indicates the number of data items each group holds. For example, node "5" holds three data items: 010110(22), 010101(21) and 010100(20), where the decimal value of the relevant binary code is shown in parenthesis. These numbers are grouped according to the four most significant bits (M-K) that are common for all three data items (0101(5)). The remaining K bits of each number (10,01,00) are encoded in "data within group" field (0111) that is marked next to the node (Fig.5.7).

Note that compared to the original binary tree from Fig.5.3a, which has ten nodes, the compressed binary tree in Fig.5.7 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, than finding a new place for it in the tree each time. However, such compression eliminates the possibility of having more than one entry with the same data.

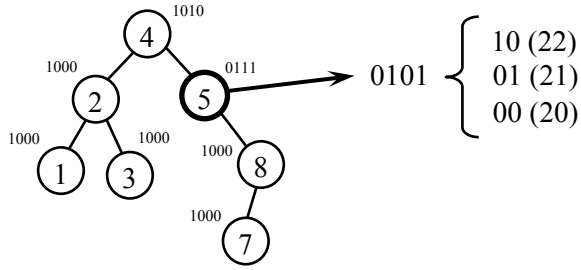



**Fig.5.7.** Compressed binary tree

In case of the compressed binary tree it is required to sort only up to $2^{M-K}$ groups of data items. When a 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.

Alternatively, bits in "data within group" field can be decoded in parallel. The parallel decoding can be applied either to the whole field or the vector can be fragmentized. In the latter case, fragments can be processed either individually or also together in parallel. The parallel decoding can be done using simple and fast combinational circuit. However, as the K grows, the complexity of decoder circuit also increases (requires more resources, introduces longer delays). An example of combinational data decoder is presented in Fig.5.8. Suppose the root node ("4") of the compressed tree from Fig.5.7 is currently being processed.

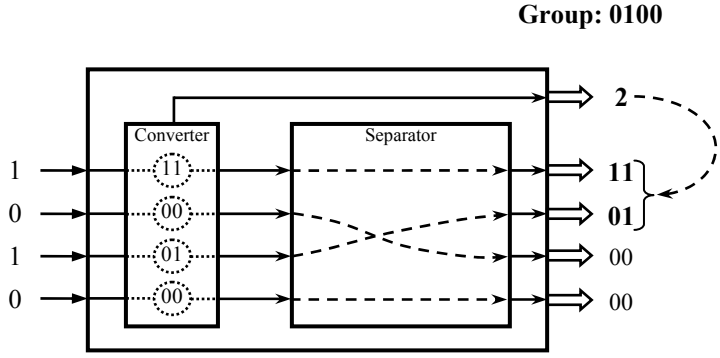



**Fig.5.8.** Combinational data decoder

The first step is to convert input flags to the respective binary values. Empty flags are converted to all-zeros (or all-ones). Convertor circuit also counts the

number of ones in the input vector to determine how many valid data items it contains. Next step is to separate valid data from the empty positions by ordering converted values in descending (ascending) order, i.e. by sorting them. This task can be easily accomplished with sorting networks described in section 2.9. The resultant network should be simple and fast as it is not supposed to sort the whole numbers, but only K least significant bits. Finally, the ordered converted values are propagated to the outputs (Fig.5.8).

The sorted sequence is generated by concatenating the group value (0100) with valid converted values that can be found (based on the number of ones in the input vector) on the top (at the bottom). The result can be transmitted either sequentially (one by one) or in parallel (all at once within a single clock cycle). If the sorted sequence is forwarded to an external device, it is also possible to send the data unprocessed (group, number of ones, ordered converted values). This allows to minimize the number of pins that are required for communication and is especially useful for small and cheap FPGAs that do not have a lot of general purpose I/Os. However, in this case the recipient device is left responsible for further processing of the data.

## 5.5. Address-based sorting

This work also describes the hardware implementation and optimization of sorting algorithms that use data items as memory addresses with one-bit flags indicating presence of data (address-based data sorting). The method is similar to the non-comparison sorting algorithm pigeonhole sort described in section 2.7. The proposed technique can be applied either directly or through tree-walk tables permitting number of bits in sorted data items to be increased (discussed in the next section).

The main idea is rather simple. As soon as a new data item is received, its value is considered to be an address of memory to record a flag. It is assumed that memory is zero filled at the beginning. Fig.5.9 shows a simple example for the same set of data that was used to build a binary tree in Fig.5.3. The positions of the flags in the memory for data items “11” and “21” are highlighted.

When all input data are recorded in memory, the sorted sequence can be immediately generated. Consider the content of the memory in Fig.5.9. It is easy to see that the data have already been organized as a sorted chain during saving. It means that in the proposed address-based sorting technique the process of saving data is actually combined with its processing. Then in order to output the sorted sequence it is simply required to sequentially examine each memory bit (e.g. starting from the lowest for ascending sorting). Empty flags (zeros) are skipped, while set flags (ones) are converted to integers. The conversion can be done using

the same simple combinational circuit that has been discussed in the previous section, thus, a delay is minimal. Besides, the vector can be fragmentized in such a way that fragments (segments) are processed (converted to the respective numbers) in parallel.

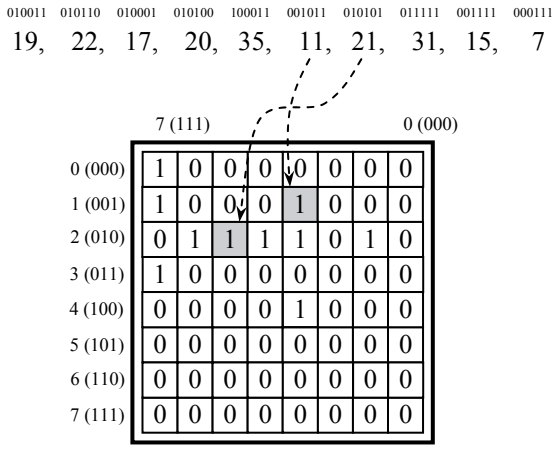



**Fig.5.9.** Positional encoding of the data for address-based sorting

The proposed method is obviously simple and effective, but there are some drawbacks as well. First, the size of memory grows very fast. The problem is that it is always necessary to provide a flag for each possible data item. Thus, for sorting M-bit data items $2^M$ flags are required. For example, if M equals 32...64, then the number of one-bit flags becomes $2^{32}...2^{64}$. Relying on the Moore law cheaper and larger memory is expected to be available on the market, but the required size ($2^{32}...2^{64}$) is still quite huge. Second, when sorting M-bit data, in practice the number of input data items Q is significantly less than $2^M$ ($Q<<2^M$) especially for large values of M. Thus, a huge number of empty flags in memory space can be expected (easily seen in Fig.5.9). Therefore, direct application of address-based sorting is effective when the range of data items is small. For example, when the value of M is small, or when all values belong to a limited group that is know in advance (flag memory requires translation of addresses). This situation is somehow similar to the SAT problem where a formula with Q clauses and M variables is considered and $Q<<2^M$. Thus, it is possible to apply some ideas inherited from the SAT such as the tree-walk tables proposed in [82].

## 5.6. Sorting over N-ary trees

Address-based data sorting can be combined with tree-like structures in order to eliminate some of its limitations. The idea is to divide the whole set of flags into segments (data segments). Then each data segment is created and stored in the

memory only when the corresponding data item (that should be saved in that segment) actually appears. The data segments can be created in any order, which cannot be known in advance. Thus, their addresses in the memory cannot be used to identify the data that is stored in these segments. Therefore, some overhead structure is also required to maintain the actual position of each data segment (most significant bits of associated data items). The N-ary (N>2) trees that are stored in tree-walk tables [82] appear to be more advantageous for this particular application. In this work only such N-ary trees for which N is a power of 2 (i.e. 2,4,8,16, etc) will be considered.

The use of tree-walk tables allows to organize the N-ary tree as a well balanced tree with a fixed depth. For a non-leaf node, the address of its leftmost child in the tree-walk table is called the base index. The rest of the children are ordered sequentially, following the leftmost child. For example, in order to locate i-th child, the index can be calculated by adding i to the base index. The absence of a child node is indicated with a special no-match tag. Data segments correspond to the leaves of this tree-like structure. Then the path from the root to each data segment would define its actual position (most significant bits of associated data items).

Consider example from Fig.5.9. Lets divide the whole set of flags into eight data segments. Thus each segment equals to one memory word and it will hold three least significant bits of input data items. The remaining three most significant bits will be encoded using N-ary (N=4) tree that is stored in tree-walk tables. Each non-leaf node of such tree can be used to store two bits ($\log_2 N=2$). The bit combination is defined as the offset of a child node pointer (in relation to the base index of the node) that points to the next non-leaf node in the path to the corresponding data segment or to the segment itself. Thus, in order to encode three bits two levels of non-leaf nodes are required.

Let's take the first data item in the input sequence from Fig.5.9. In order to encode this number, its binary value is divided into three parts: 0, 10 and 011. The first two parts will be represented with non-leaf nodes and the third part will be stored in the data segment. The first four memory words are reserved for the tree-walk table of the root node. Each word holds a pointer (base index) to the child node. During creation all pointers are assigned a no-match (NM) value. The most significant bit (complemented with additional zero) forms the offset. This offset is then added to the base index of the root (0) in order to find the base index of the next non-leaf node in the path to the corresponding data segment. Currently that location contains a no-match tag (as this is the first data item), thus creation of the new node is required. The no-match tag is replaced with the base address of the new non-leaf node (4) and tree-walk table for the new node is created. Now the second part of the input data item is used to determine the location of the data segment itself. Currently that location contains a no-match tag (as this is the first data item), thus creation of the new data segment is required. The no-match tag is replaced with

the address of the new data segment (8) and the last part of the data item is used to set the appropriate flag. N-ary tree for data item “19” (010011) and its memory representation are shown in Fig.5.10.

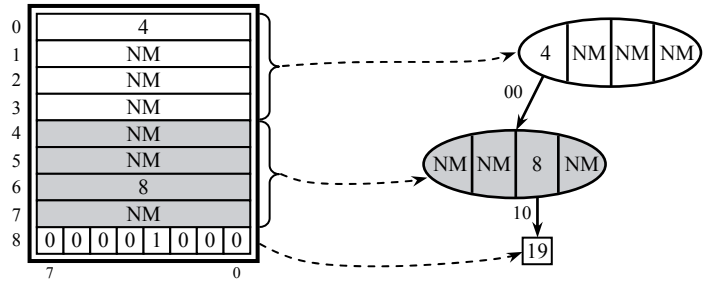



**Fig.5.10.** N-ary tree (N=4) for data item “19”

Fig.5.11 depicts the whole N-ary tree (Fig.5.11b) and its memory representation (Fig.5.11a) that is built for the example from Fig.5.9. The tree has a fixed depth of three levels and it can be used to efficiently locate segments associated with any new incoming data item within a constant number of processing steps.

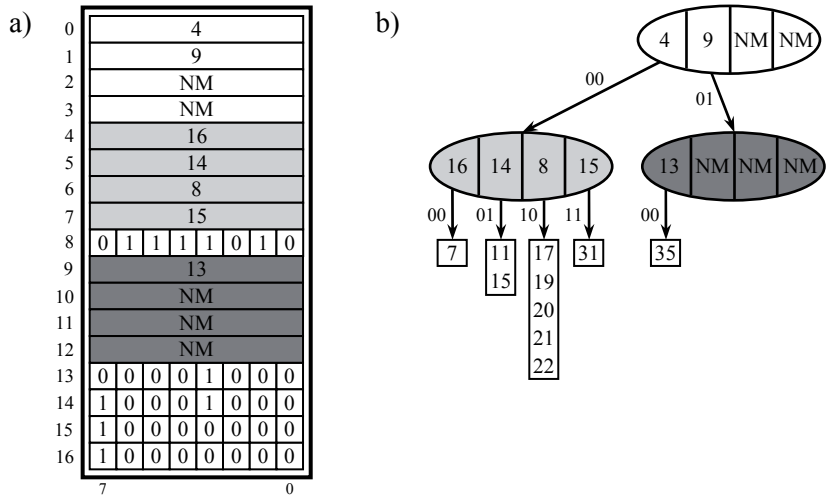



**Fig.5.11.** (a) N-ary tree (N=4) and (b) its memory representation for the whole data set

Note that the last three empty memory words from the example in Fig.5.9 are not present in memory in Fig5.11a. However, as the example is too small, the introduced tree overhead is far greater, thus the memory consumption is actually increased. Therefore, this technique works best for larger data items. The number of bits in sorted data items can be increased by using trees with bigger values of N, by introducing additional levels to the tree or choosing larger size for data segments.

Such N-ary trees can be easily stored either in on-chip embedded memory blocks of FPGAs, or in external memory devices.

Data sorting can be performed using inorder traversal of the N-ary tree, thus going through all data segments. Each data segment can then be processed as described in the previous section. As opposed to the binary search tree where all the nodes can be handled in the same manner, the proposed N-ary trees have two types of nodes that require different processing. This is because non-leaf nodes contain only pointers, while leaf nodes (data segments) contain data items. Also the size of a non-leaf and a leaf node may be different (as it was shown in the example). While it is quite easy to distinguish between the node types (the tree has a fixed depth that should be known in advance), the need for different treatment may introduce additional complexity to the algorithm.

## 5.7. Multi-level data processing

In order to improve performance of data processing different models can be combined, such as those which have been discussed in this chapter (the use of the walk technique, binary trees, sorting networks and address-based sorting). Obviously, it is not necessary to use all these models together at the same time. If the number of data items is limited (for instance, less than $2^8$), then sorting networks would allow a nearly optimal solution to be produced. If there are additional requirements like fast resorting, then tree-like structures can provide a significant assistance in this respect. Thus linking of these different models can allow both properties of fast sorting and resorting to be achieved. Some examples of such multi-level data processing have been presented in the previous sections of this chapter:

- combining compressed binary trees with sorting networks;
- combining address-based sorting with sorting networks;
- combining N-ary trees with address-based sorting (that in turn can be also combined with sorting networks).

It is also possible to design such a multi-level data processing system that incorporates different models and can adapt itself based on the nature of the incoming data. Then each model can be used either autonomously or in combinations with the other methods. However, such circuit most certainly needs a complex decider that would select the most appropriate combination depending on input.

## 5.8. Chapter summary

This chapter presented a number of simple, but efficient sorting techniques that are particularly useful for implementation in FPGAs. The challenge was to use cheap reconfigurable devices to design high-performance sorters adaptable to generally unknown number of input data items.

The proposed sorting techniques were based on tree-like structures and address-based sorting. A number of methods that allowed to improve the sequential flow of the sorting algorithms, apply parallel processing and reduce memory requirements were suggested here.

It was also shown that in order to improve performance of data processing various models can be combined to produce an even better solution. Such linking permits to exploit advantages of different methods within a single device. For example, sorting networks with tree-like structures allows both properties of fast sorting and resorting to be achieved.

# 6. Experiments and results

The methods considered in the previous chapter were implemented and tested. Five types of experiments have been performed:

- the proposed methods were verified in software (C/C++) [20] running on general-purpose computer (Intel Core 2 Duo CPU, 1.87 GHz) and embedded processor (PowerPC PPC405 in Virtex-4 available on the prototyping board FX12 [92]);
- the synthesis and implementation of the circuits from the specification in VHDL were done in Xilinx ISE 13.2 [93] for Spartan3E-1200E-FG320 FPGA available on NEXYS-2 prototyping board of Digilent [94];
- the considered above circuits have been implemented and tested using the different models of HFSM (with explicit and implicit modules);
- comparison to alternative recursive and iterative methods reported in previously published papers was made;
- optimization of power consumption based on state encoding was applied;
- applicability of the proposed methods for resorting of newly arriving data items together with the previously received portions of data was investigated.

Data for the experiments were acquired from the LFSR-based pseudo-random-number generator. Since it is very difficult or even impossible to take into account the performance of input/output operations, it is assumed that all data are available inside FPGA either preliminarily copied to built-in FPGA memory, or represented in form of incoming streams produced by pseudo-random-number generator. Thus, all the considered circuits were entirely implemented in a single FPGA and no external resources were used at all.

The remainder of this chapter is organized as follows. Section 6.1 provides results for sequential sorting over binary trees. Section 6.2 gives an overview of the results for hardware architectures that process binary trees in parallel. Section 6.3 deals with results for address-based sorting technique. Section 6.4 suggests a power consumption optimization technique. In section 6.5 the comparison of the received results is made. The conclusions are drawn in section 6.6.

## 6.1. Results for sequential sorting over binary trees

The experiments were carried out using LFSR-based pseudo-random-number generator that produced 16-bit data items. All values were unique and the number of data items varied between 1200-1300. The generator and the circuits were built within the same FPGA. The items were sorted using the known method (S1) and improved method (S2) that have been described in section 5.2. Also the same data sets were used to construct compressed binary trees (section 5.4) for M=16 and K=4 ($S1_{K4S}$). The tree was sorted using the known method S1, while the "data within group" field was sorted sequentially (the data items were extracted one by one each clock cycle).

The sorting time for S1, S2 and $S1_{K4S}$ in clock cycles is presented in Table 6.1. The *Number of Data Items* column shows the total number of data items in the test data set. 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 and S2 on the balance between the left and right sub trees (not important for $S1_{GR4}$ as it does not use the original binary tree for sorting data). Columns S1, S2 and $S1_{K4S}$ show results for the corresponding methods.

**Table 6.1.** Sorting time for sequential algorithms (in clock cycles)

| Number of Data Items | Balance (Left/Right) | S1 | S2 | $S1_{K4S}$ |
|---|---|---|---|---|
| 1211 | 185/1025 | 4843 | 3373 | 2830 |
| 1216 | 266/949 | 4863 | 3393 | 2856 |
| 1248 | 332/915 | 4991 | 3486 | 2921 |
| 1203 | 460/742 | 4811 | 3350 | 2804 |
| 1228 | 528/699 | 4911 | 3432 | 2901 |
| 1212 | 556/655 | 4847 | 3350 | 2831 |
| 1230 | 623/606 | 4919 | 3470 | 2915 |
| 1259 | 822/436 | 5035 | 3496 | 2920 |
| 1230 | 799/430 | 4919 | 3419 | 2834 |
| 1304 | 849/454 | 5215 | 3610 | 2977 |
| 1276 | 963/312 | 5103 | 3564 | 2931 |
| 1225 | 958/266 | 4899 | 3417 | 2886 |
| 1225 | 986/238 | 4899 | 3420 | 2889 |
| 1199 | 1051/147 | 4795 | 3319 | 2824 |

The sorting performance of the known method S1 is the worse. The average number of clock cycles per data item is approximately 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. Methods S1 and S2 exhibit the same performance if the number of data items is increased as reported in [16]. Sorting of the compressed tree using known method S1 and sequential decoder comes down to 2.3 clock cycles per data item.

All methods have been implemented in Spartan3E-1200E-FG320 FPGA device. The results are presented in Table 6.2. The circuits are based on the known model of HFSM with explicit modules described in section 4.2 (column $HFSM_{explicit}$). Methods S1 and S2 have also been implemented using the new optimized model of HFSM with implicit modules also in section 4.2 (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. Column F presents the maximum attainable clock frequency in MHz, column S - the number of slices, column L - the number of LUTs and column B - the number of block RAMs.

**Table 6.2.** Implementation results for sequential algorithms

| Sorting 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 |
| $S1_{K4S}$ | 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 the highest maximum clock frequency. As more complexity is introduced to improve the performance (sequential improvements, tree compression) of 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 sequential decoder 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 (section 4.2) 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 [17,18] as will be shown in section 6.4.

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 6.2) must also be taken into account. The sorting performance of the known method S1 remains the worse. The average time per data item is approximately 40 ns. The sequentially improved method S2 still provides a better result with the average time of 34 ns. Sorting of the compressed tree using known method S1 and sequential decoder comes down to 30 ns. It should be noted that performance distribution remains the same as for clock cycles. However, in this case the results are much closer due to the difference in the maximum attainable clock frequency (more complex processing introduces longer delays).

Applicability of the tree-like structures to resorting was also investigated. For example, such resorting is important for management of priorities considered in [81]. After sorting of 1200-1300 data items a new portion that included from 10 to 120 data items was added to the binary tree and the new tree was sorted again. The results were compared to the full construction and sorting times for a new data sequences as if the binary tree was built from a scratch. The acceleration from 400 (10 new data items) to 50 (120 new data items) times has been achieved.

## 6.2. Results for parallel sorting over binary trees

The experiments were carried out using LFSR-based pseudo-random-number generator that produced 16-bit data items. All values were unique and the number of data items varied between 1200-1300. The generator and the circuits were built within the same FPGA. The items were sorted using two parallel approaches that have been described in section 5.3: main/secondary architecture (P1) and parallel traversal of independent trees for N=2 ($P2_{N2}$) and N=4 ($P2_{N4}$). In all approaches the sorting units were based on the known method S1. Also the same data sets were used to construct compressed binary trees (section 5.4) for M=16 and K=4 ($S1_{K4P}$). The tree was sorted using the known method S1, while the "data within group" field was processed using decoder based on sorting network. It was assumed that the host device could handle sorted data that were sent in parallel (process up to 16 data items in parallel within one clock cycle).

The sorting time for P1, $P2_{N2}$, $P2_{N4}$ and $S1_{K4P}$ in clock cycles is presented in Table 6.3. The *Number of Data Items* column shows the total number of data items in the test data set. 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 P1 on the balance between the left and right sub trees (not important for $P2_{N2}$, $P2_{N4}$ and $S1_{K4P}$ as they do not use the original binary tree for sorting data). Columns P1, $P2_{N2}$, $P2_{N4}$ and $S1_{K4P}$ show results for the corresponding methods.

**Table 6.3.** Sorting time for parallel approaches (in clock cycles)

| Number of Data Items | Balance (Left/Right) | P1 | $P2_{N2}$ | $P2_{N4}$ | $S1_{K4P}$ |
|---|---|---|---|---|---|
| 1211 | 185/1025 | 5129 | 2474 | 1298 | 2159 |
| 1216 | 266/949 | 4749 | 2462 | 1296 | 2187 |
| 1248 | 332/915 | 4579 | 2522 | 1326 | 2231 |
| 1203 | 460/742 | 3714 | 2420 | 1290 | 2135 |
| 1228 | 528/699 | 3499 | 2490 | 1295 | 2231 |
| 1212 | 556/655 | 3279 | 2440 | 1291 | 2159 |
| 1230 | 623/606 | 3101 | 2479 | 1302 | 2247 |
| 1259 | 822/436 | 3727 | 2541 | 1329 | 2215 |
| 1230 | 799/430 | 3629 | 2507 | 1325 | 2139 |
| 1304 | 849/454 | 3853 | 2632 | 1414 | 2231 |
| 1276 | 963/312 | 4167 | 2573 | 1399 | 2207 |
| 1225 | 958/266 | 4101 | 2479 | 1312 | 2215 |
| 1225 | 986/238 | 4185 | 2479 | 1305 | 2219 |
| 1199 | 1051/147 | 4354 | 2432 | 1281 | 2167 |

Not surprisingly, parallel method P1 gives 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 $P2_{N2}$ improves performance to approximately 2 clock cycles per data item. Increasing the number of parallel sorters from 2 to 4 ($P2_{N4}$) decreases the time of sorting to almost 1 clock cycle per data item. Parallel architecture P2 (both versions) exhibit the same performance if the number of data items is increased as reported in [16]. Sorting of the compressed tree using known method S1 and parallel decoder comes down to 1.8 clock cycles per data item, which is better than in case of sequential decoding. Acceleration is much less than $2^K$ because of the small number of data items that are uniformly distributed by pseudo-random-number generator (groups contain too little data items for parallel processing).

All methods have been implemented in Spartan3E-1200E-FG320 FPGA device. The results are presented in Table 6.4. The circuits are based on the known model

of HFSM with explicit modules described in section 4.2. Column F presents the maximum attainable clock frequency in MHz, column S - the number of slices, column L - the number of LUTs and column B - the number of block RAMs.

**Table 6.4.** Implementation results for parallel approaches

| Sorting Method | F | S | L | B |
|---|---|---|---|---|
| P1 | 102 | 1115 | 2203 | 8 |
| $P2_{N2}$ | 90 | 1297 | 2480 | 8 |
| $P2_{N4}$ | 83 | 1707 | 3209 | 12 |
| $S1_{K4P}$ | 88 | 623 | 1858 | 4 |

Parallel processing of trees is definitely faster than sequential, but hardware resources are also greatly increased (most notably the number of embedded memory blocks as temporary storage is required). Parallel architecture P1, being the simplest, requires medium FPGA resources and has the highest maximum clock frequency. TLM and output circuit of parallel architecture P2 further increase consumption of FPGA recourses and introduce additional delays (especially true for $P2_{N4}$). Compared to sequential decoding of the compressed binary tree parallel decoder requires more resources for combinational logic (due to the use of sorting network).

In order to estimate the performance for a particular implementation device, the clock cycle period (Table 6.4) must also be taken into account. The sorting performance of the parallel architecture P1 remains the worse. The average time per data item is approximately 25-40 ns (depending on the tree balance). The parallel architecture P2 still provides a better result with the average time of 22 ns for $P2_{N2}$ and 12 ns for $P2_{N4}$. Sorting of the compressed tree using known method S1 and parallel decoder comes down to 20 ns. It should be noted that performance distribution remains the same as for clock cycles. However, in this case the results are much closer due to the difference in the maximum attainable clock frequency (more complex processing introduces longer delays).

## 6.3. Results for address-based sorting

The experiments were carried out using LFSR-based pseudo-random-number generator that produced a sequence of data items with the required length (18 bits, 19 bits, 20 bits). The generator and the circuits were built within the same FPGA. The items were sorted using two approaches: direct address-based sorting (section 5.5) and combination of address-based sorting with N-ary trees (section 5.6). The

latter was implemented using model of HFSM with explicit modules. The length of a memory word was set to 16 bits. Each 16-bit flag vector was processed using decoder based on sorting network (section 5.4). It was assumed that the host device could handle sorted data that were sent in parallel (process up to 16 data items in parallel within one clock cycle).

Direct implementation of address-based method (Direct (18-bit)) using Spartan3E-1200E-FG320 FPGA device [19] permits any set of 18-bit data to be sorted (up to $2^{18}$ items). The results in Table 6.5 permit the complexity and performance to be evaluated. The number of clock cycles needed to fill in BRAM is equal to the number of data items (assuming that each data item can be saved in BRAM during one clock cycle). For this particular implementation the number of clock cycles needed to sort data is equal to $2^{14}$ (one clock cycle is used to read 16-bit word from which up to 16 data items can be extracted during the same clock cycle). The sorter works very fast and requires very few resources as the processing is trivial.

The first implementation of the method based on tree-walk tables (TW (18-bit)) for N-ary tree (N=4) also permits any set of 18-bit data to be sorted (up to $2^{18}$ numbers). The results in Table 6.5 permit the complexity and performance to be evaluated. For different data sets the actual number of clock cycles required for sorting varies from 7000 to 53556 as shown in Table 6.6. The problem is that data items are uniformly distributed by pseudo-random-number generator. Thus for storing 6000 or more data items the complete N-ary tree is required. Compared to the direct address-based sorting approach N-ary tree implementation needs more hardware resources and operates at lower clock frequency making it inferior.

However, the advantage of combining address-based data sorting with N-ary trees lies in the possibility of sorting data with greater length (19 bits, 20 bits, etc.) within the same Spartan3E-1200E-FG320 FPGA. In such case direct address-based sorting in not possible, as this particular device cannot provide enough embedded memory. The second implementation of the method based on tree-walk tables (TW (20-bit)) for N-ary tree (N=4) allows to sort sets of 20-bit data (Table 6.6). The results in Table 6.5 permit the complexity and performance to be evaluated. Note, that for sorting 20-bit (also 19-bit) data items the circuit itself was left practically the same. This is because the previously described sorter was in fact implemented for data items with the length of 20 bits and was simply supplied numbers that began with “00”. The only difference is that now all available embedded memory blocks of Spartan3E-1200E-FG320 FPGA are in use.

**Table 6.5.** Implementation results for address-based sorting

| Sorting Method | F | S | L | B |
|---|---|---|---|---|
| Direct (18-bit) | 155 | 326 | 578 | 16 |
| TW (18-bit) | 77 | 562 | 1048 | 18 |
| TW (20-bit) | 78 | 586 | 1109 | 28 |

The maximum number of sorted data depends on the distribution of data within the interval from 0 to $2^{20}$-1. A pseudo-random-number generator that was used in experiments produced sequences of uniformly distributed data items. Thus whenever a data item could not be saved the experiment was aborted. The results are shown in Table 6.6. The examination of the constructed N-ary trees revealed that the data segments were poorly filled. Therefore, the maximum number of sorted data can be greatly increased if there are many large clusters (groups) of data items that can efficiently fill the created data segments.

**Table 6.6.** Sorting time for address-based sorting (in clock cycles)

| Number of Data Items | Direct (18-bit) | TW (18-bit) | TW (19-bit) | TW (20-bit) |
|---|---|---|---|---|
| 100 | 16384 | 7000-7700 | 8500-8900 | 9300-10000 |
| 300 | 16384 | 16000-16600 | 19000-20000 | 22000-25000 |
| 500 | 16384 | 20000-24000 | 27000-29000 | 34000-36000 |
| 1000 | 16384 | 33000-36000 | 44000-47000 | 55000-58000 |
| 2000 | 16384 | 45000-47000 | 68000-70000 | 82000-83000 |
| 3000 | 16384 | 49000-51000 | 83000-86000 | - |
| 5000 | 16384 | 52000-53000 | - | - |
| 6000 | 16384 | 53556 | - | - |

In order to estimate the performance for a particular implementation device, the clock cycle period (Table 6.5) must also be taken into account. Sorting time per data item for both direct and combined approaches is shown in Fig.6.1. The performance distribution remains practically the same as for clock cycles.

For small sets of data (e.g. hundred 18-bit data items) the combined approach is actually able to outperform direct address-based sorting. However, as the N-ary tree grows larger, the time required for its traversal also increases and occasionally begins to dominate in the sorting procedure. Therefore, direct address-based sorting approach can be considered preferable when FPGA posses sufficient memory

resources. When address-based sorting is not possible, the combined approach can still be used. However, due to the uniform data distribution the current preliminary implementation provides rather small data capacity as it has been shown in Table 6.6. Thus, additional research effort to improve the capabilities of the combined approach is definitely required.

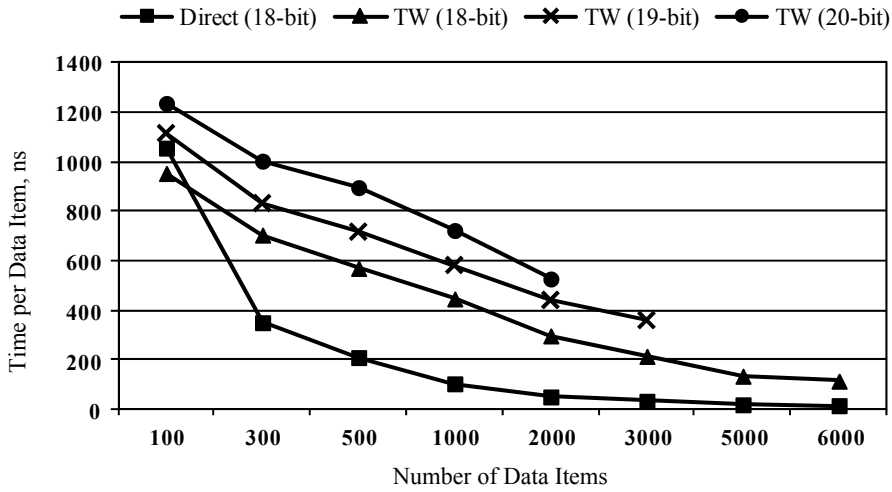



**Fig.6.1.** Sorting time per data item for address-based sorting (in ns)

It should be also noted that address-based sorting is not effective for very small number of data items (Fig.6.1). In fact the direct address-based sorting works most efficiently when the number of data items is close to maximum. The combined approach performs best when data items comes in large clusters (groups) that can efficiently fill the created data segments.

## 6.4. Optimization of power consumption

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 behavior of a 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 [83]. By labeling 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

behavior 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 behavior. 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 [84]. Therefore, reduction of switching activity in the state register was chosen as the primary optimization goal. Based on stochastic model of the 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 [95], which had been developed at Tallinn University of Technology (TUT). In order to estimate the impact of the encoding on power consumption, Xilinx ISE 13.2 was used for carrying out FPGA design flow with Spartan3E-1200E-FG320 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. Experiments have shown the decreasing of power consumption in about 5%. Note that this research is still a work in progress and the received results should be considered preliminary.

## 6.5. Comparison

Experimental results [20] show that hardware implementations are definitely faster than software implementations (either in general purpose or embedded processors) in all cases even though the clock frequencies of the FPGA and the PC

differ significantly. This is because the proposed optimization techniques are valid just for hardware circuits and cannot be implemented in software.

The known method S1 exhibits a very steady performance. However, it falls short before other methods in this respect. Sequentially improved method S2 and sorting of the compressed binary tree provide approximately the same speedup 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. Also parallel processing of the "data within group" field using sorting network makes sorting of the compressed binary tree even faster.

The dependency of the parallel architecture P1 on tree balance limits its practical usability although for a well-balanced tree the performance improvement is quite significant. However, parallel architecture $P2_{N2}$ for two instances of S1 delivers peak performance of P1 for any input data sequence with basically the same area overhead (at the cost of stability). The best performance is achieved by $P2_{N4}$ 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. Increasing the number of sorters beyond four seems irrelevant as $P2_{N4}$ already requires practically one clock cycle per data item.

The experiments have been done for relatively small sets of data items (1200-1300). The main restriction that limits the number of data items is the amount of available embedded block memories on the FPGA microchip, as they are used to store the binary tree [16]. Therefore, the number of data items can be significantly increased by replacing cheaper Spartan 3E family device with mode advanced FPGA such as Spartan-6 or Virtex-7 family. This would also increase the performance, as these devices are generally faster. It is also possible to employ the external memory, but it may actually introduce additional complexity to the proposed techniques due to a fixed size of memory words and the inevitable need for sharing. The proposed tree compression technique can also allow to sort more data on the same FPGA device.

The results of experiments demonstrate that the considered address-based sorting provide a number of advantages compared to data sort over binary trees (e.g. the required hardware resources). Most importantly, the complexity of problems that can be solved in a single FPGA Spartan3E-1200E-FG320 can be sufficiently increased. Combining address-based sorting with N-ary trees permits the length of data to be increased even further. Besides, because there is no data dependency between tree branches of N-ary tree, the individual sub-trees can potentially be processed with any desired level of parallelism. The experiments have shown that the main restriction that limits the number of data items is the available embedded block RAMs on the FPGA microchip. Therefore, the number of data items can be

significantly increased by replacing the cheap Spartan-3E FPGA with a more advanced FPGA (Spartan-6 or Virtex7) or by using external memory. The algorithms themselves are easily scalable.

In other works the known methods were often either modeled or just partially tested in available prototyping systems. Frequently, external onboard memories were used. Thus, the exact comparison in hardware is indeed very difficult. However, the performance of the proposed methods is found to be comparable with known results obtained for significantly more advanced FPGAs [36,72,85].

## 6.6. Chapter summary

This chapter reported experimental results and comparison for the proposed methods. All methods were designed, implemented in hardware and tested entirely inside a single FPGA microchip.

The advantages of the proposed methods that allowed to improve the sequential flow of the sorting algorithms, apply parallel processing and reduce memory requirements were demonstrated here. It was also shown that in order to improve performance of data processing various models can be combined to produce an even better solution as such linking permitted to exploit advantages of different methods within a single sorting device.

It was determined that the main restriction limiting the number of data items is the amount of available embedded block memories in the FPGA microchip. Therefore, the number of data items can be significantly increased by replacing cheaper Spartan 3E family device with mode advanced FPGA such as Spartan-6 or Virtex-7 family. This would also increase the performance, as these devices are generally faster.

Experimental results showed that hardware implementations were definitely faster than software implementations. The performance of the proposed methods was found to be comparable with known results.

# Conclusions

The main contributions of this thesis are:

- exploration of a new model of HFSM with implicit modules that is faster and less resource consuming compared to HFSM with explicit modules;
- development of new methods allowing tree-like structures to be represented and processed in hardware;
- application of a multilevel model for data processing;
- proof of advantages for the proposed techniques based on prototyping in FPGA, numerous experiments and comparisons.

The proposed methods are based on tree-like structures that possess a very important advantage of rapid adaptation to eventual modifications. Any manipulations over tree nodes (e.g. insertion of a new node) are simple and fast, while the actual sorting can be done in linear time. This property is very important for fast resorting that is essential for priority buffers (queues) and similar devices, which are widely used in practice.

Hardware circuits implementing proposed sorting methods are based on the model of HFSM. It was shown that the new model of HFSM with implicit modules is faster and less resource consuming. Also it allows to apply optimization methods developed for conventional finite state machines, such as power consumption optimization based on state encoding.

A number of sorting techniques and optimizations that are particularly useful for implementation in FPGAs have been suggested:

- sorting over binary trees can be greatly improved through the use of dual-port memories available in most modern FPGAs;
- a significant improvement in performance can be achieved with introduction of parallelism;
- the memory requirement for storing a binary tree can be significantly reduced by applying positional encoding to the least significant bits of data items;
- the compressed data can be efficiently sorted with parallel decoders that are based on sorting networks;
- address-based data sorting, which is effective when the range of data items is small;

- combining address-based data sorting with N-ary trees permits to extend the length of sorted data.

Some of the abovementioned suggestions involve the concept of multi-level data processing in order to improve the performance. This concept combines different sorting techniques in such a way that permits to produce even better results. The effectiveness of such approach was demonstrated on the examples of data sorting. However, in order to process big sets of data either more powerful FPGA or external memory is required.

All the proposed methods were designed, implemented in hardware and tested entirely inside a single FPGA microchip. The emphasis was done on implementation in cheap FPGA circuits that can be used for different embedded systems. Thus, it can be concluded that the proposed techniques are widely applicable and quite complex problems can be processed in relatively simple FPGAs of Spartan-3E family. It was determined that the main restriction limiting the number of data items is the amount of available embedded block memories in the FPGA microchip. Therefore, the size of data sets can be significantly increased by replacing cheaper Spartan 3E family device with mode advanced FPGA such as Spartan-6, Virtex-6 or Virtex-7 family. This would also increase the performance, as these devices are generally faster.

Experimental results show that hardware implementations are definitely faster than software implementations (either in general purpose or embedded processors) in all cases even though the clock frequencies of the FPGA and the PC differ significantly. This is because the proposed optimization techniques are valid just for hardware circuits and cannot be implemented in software. Also the performance of the proposed methods is found to be comparable with known results obtained for significantly more advanced FPGAs.

The results of this work are not limited to sorting alone. They have a wider scope and can be applied effectively to numerous systems that implement recursive algorithms over tree-like structures.

# References


*Books:*

[1] D.E. Knuth. “The Art of Computer Programming, Volume 3: Sorting and Searching”, 2nd Edition, Addison-Wesley, 1998.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stain. “Introduction to Algorithms”, 2nd Edition, MIT Press, 2002.

[3] R. Sedgwick, K. Wayne. “Algorithms”, 4th Edition, Addison-Wesley, 2011.

[4] F.M. Carrano, “Data abstraction and problem solving with C++: walls and mirrors”, 5th Edition, Addison-Wesley, 2007.

[5] B.W. Kernighan, D.M. Ritchie. “The C Programming Language”, 2nd Edition, Prentice Hall, 1988.

[6] C. Maxfield. “The Design Warrior's Guide to FPGAs: Devices, Tools and Flows”, Newnes, 2004.

[7] R.C. Cofer, B.F. Harding. “Rapid System Prototyping with FPGAs: Accelerating the design process”, Newnes, 2006.

[8] J. Armstrong. “Programming Erlang: Software for a Concurrent World”, Pragmatic Bookshelf, 2007.

[9] G. de Micheli. “Synthesis and Optimization of Digital Circuits”, McGraw-Hill, 1994.

[10] S.E. Lyshevski, G.J. Iafrate, W.A. Goddard, D.W.Brenner. “Handbook of Nanoscience, Engineering, and Technology”, CRC Press, 2003.

*Co-authored papers:*

[11] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Synthesis and Implementation of Hierarchical Finite State Machines with Implicit Modules” The 2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2010), Cancun, Mexico, December 13-15, 2010, pp. 436-441.

[12] M. Jenihhin, M. Gorev, V. Pesonen, D. Mihhailov, P. Ellervee, H. Hinrikus, M. Bachmann, J. Lass. “EEG Analyzer Prototype Based on FPGA”, IEEE 7th International Symposium on Image and Signal Processing and Analysis (ISPA 2011), Dubrovnik, Croatia, September 4-6, 2011, pp. 101-106.

[13] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Hardware Implementation of Recursive Algorithms”, 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2010), Seattle, WA, USA, August 1-4, 2010, pp. 225-228.

[14] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Parallel FPGA-based Implementation of Recursive Sorting Algorithms”, The 2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2010), Cancun, Mexico, December 13-15, 2010, pp. 121-126.

[15] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Acceleration of Recursive Data Sorting over Tree-based Structures”, Electronics and Electrical Engineering, 7(113), 2011, pp. 51-56.

[16] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. “Optimization of FPGA-based Circuits for Recursive Data Sorting” The 12th biennial Baltic Electronics Conference (BEC 2010), Tallinn, Estonia, October 4-6, 2010, pp. 129-132.

[17] A. Sudnitson, D. Mihhailov, M. Kruus. “Project-Oriented Approach to Low-Power Topics in Advanced Digital Design Course”, Electronics and Electrical Engineering, 6 (102), 2010, pp. 151-154.

[18] D. Mihhailov, A. Sudnitson, K. Tarletski. “Web-Based Tool for FSM Encoding Targeting Low-Power FPGA Implementation”, The 27th International Conference on Microelectronics (MIEL 2010), Nis, Serbia, May 16-19, 2010, pp. 349-352.

[19] V. Sklyarov, I. Skliarova, D. Mihhailov, A. Sudnitson. “Implementation in FPGA of Address-based Data Sorting”, The 21st International Conference on Field Programmable Logic and Applications (FPL 2011), Chania, Crete, Greece, September 5-7, 2011, pp. 405-410.

[20] V. Sklyarov, I. Skliarova, D. Mihhailov, A. Sudnitson. “High-performance Hardware Accelerators for Sorting and Managing Priorities”, IEEE Symposium of Design and Diagnostics of Electronic Circuits and Systems (DDECS 2011), Cottbus, Germany, April 13-15, 2011, pp. 313-318.

*Papers:*

[21] G. Estrin. “Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer”, IEEE Annals of the History of Computing, Volume 24 Issue 4, October 2002, pp. 3-9.

[22] J. F. Miller, D. Job, V. K. Vassilev. “Principles in the Evolutionary Design of Digital Circuits – Part I”, Genetic Programming and Evolvable Machines, Volume 1 Issue 1-2, April, 2000, pp. 7-35.

[23] R. Mueller, J. Teubner, G. Alonso. “Data processing on FPGAs”, Proceedings of the VLDB Endowment, Volume 2 Issue 1, August 2009, pp. 910-921.

[24] N.K. Govindaraju, J. Gray, R. Kumar, D. Manocha. “GPUTeraSort: High performance graphics co-processor sortingfor large database management”, The 2006 ACM SIGMOD International Conference on Management of Data, 2006, pp. 325-336.

[25] H. Inoue, T. Moriyama, H. Komatsu, T. Nakatani. “AA-Sort: A new parallel sorting algorithm for multi-core SIMD processors”, 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), Brasov, Romania, September 15-19, 2007, pp. 189-198.

[26] B. Gedik, R.R. Bordawekar, P.S. Yu. “CellSort: Highperformance sorting on the Cell processor”, 33rd International Conference on Very Large Data Bases (VLDB’07), Vienna, Austria, September 23-27, 2007, pp. 1286-1297.

[27] J. Chhugani, A.D. Nguyen, V.W. Lee, W. Macy, M. Hagog, Y.K. Chen, A. Baransi, S. Kumar, P. Dubey. “Efficient implementation of sorting on multi-core SIMD CPU architecture”, Proceedings of the VLDB Endowment, Volume 1 Issue 2, August 2008, pp. 1313-1324.

[28] D.J. Greaves, S. Singh. “Kiwi: Synthesis of FPGA circuits from parallel programs”, 16th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2008), Stanford, Palo Alto, California, USA, April 14-15, 2008, pp. 3-12.

[29] S.S. Huang, A. Hormati, D.F. Bacon, R. Rabbah. “LiquidMetal: Object-oriented programming across the hardware/software boundary”, 22nd European Conference on Object-Oriented Programming (ECOOP 2008), Paphos, Cyprus, 2008, pp. 76-103.

[30] A. Mitra, M.R. Vieira, P. Bakalov, V.J. Tsotras, W. Najjar. “Boosting XML Filtering through a scalable FPGA-based architecture”, 4th Biennial Conference on Innovative Data Systems Research (CIDR 2009), Asilomar, CA, USA, January 4-7, 2009.

[31] I. Skliarova, A.B. Ferrari. “Reconfigurable Hardware SAT Solvers: A Survey of Systems”, IEEE Transactions on Computers, Volume 53 Issue 11, November 2004, pp. 1449-1461.

[32] V. Sklyarov, I. Skliarova, B. Pimentel. "FPGA-based Implementation of Graph Colouring Algorithms", Studies in Computational Intelligence (SCI) 76, Autonomous Robots and Agents, Springer-Verlag Berlin Heidelberg, 2007, ch. 26, pp. 225-231.

[33] I. Skliarova, V. Sklyarov. "Design Methods for FPGA-based implementation of combinatorial Search Algorithms", International Workshop on SoC and MCSoC Design (IWSOC'2006), 4th International Conference on Advances in Mobile Computing and Multimedia (MoMM'2006), Yogyakarta, Indonesia, December 2006, pp. 359-368.

[34] X. Ye, D. Fan, W. Lin, N. Yuan, P. Ienne. “High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs”, 4th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010), Atlanta, Georgia, USA, 19-23 April 2010.

[35] S. Chey, J. Liz, J.W. Sheaffery, K. Skadrony, and J. Lach. “Accelerating Compute-Intensive Applications with GPUs and FPGAs”, IEEE Symposium on Application Specific Processors (SASP 2008), Anaheim, USA, June 2008, pp. 101-107.

[36] R.D. Chamberlain, N. Ganesan. “Sorting on Architecturally Diverse Computer Systems”, 3rd International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA'09), Portland, Oregon, USA, November 2009, pp. 39-46.

[37] T. Maruyama, M. Takagi, T. Hoshino. “Hardware implementation techniques for recursive calls and loops”, 9th International Workshop on Field-Programmable Logic and Applications (FPL'99), Glasgow, UK, August 30 - September 1, 1999, pp. 450-455.

[38] T. Maruyama, T. Hoshino. “A C to HDL compiler for pipeline processing on FPGAs”, 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’2000), Napa Valley, CA , USA, April 17-19, 2000, pp. 101-110.

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

[40] V. Sklyarov, I. Skliarova, B. Pimentel, "FPGA-based Implementation and Comparison of Recursive and Iterative Algorithms", 15th International Conference on Field-Programmable Logic and Applications (FPL'05), Finland, August 24-26, 2005, pp. 235-240.

[41] S. Ninos, A. Dollas. “Modeling recursion data structures for FPGA-based implementation”, 18th International Conference on Field Programmable Logic and Applications (FPL’08), Heidelberg, Germany, September 8-10, 2008, pp. 11-16.

[42] I. Skliarova, V. Sklyarov. "Recursion in Reconfigurable Computing: a Survey of Implementation Approaches", 19th International Conference on Field Programmable Logic and Applications (FPL’09), Prague, Czech Republic, August 31 - September 2, 2009, pp. 224-229.

[43] W. Dobosiewicz. “An efficient variation of bubble sort”, Information Processing Letters, Volume 11, Number 1, August 29, 1980, pp. 5–6.

[44] O. Astrachan. “Bubble Sort: An Archaeological Algorithmic Analysis”, 34th SIGCSE technical symposium on Computer science education (SIGCSE'03), Reno, Navada, USA, January 2003, pp. 1-5.

[45] D.L. Shell “A high-speed sorting procedure”, Communications of the ACM, Volume 2 Issue 7, July 1959, pp. 30-32.

[46] R.Sedgwick. “Analysis of Shellsort and Related Algorithms””, 4th Annual European Symposium on Algorithms (ESA'96), Barcelona, Spain, September 25-27, 1996, pp. 1-11.

[47] J.W.J. Williams. “Algorithm 232 - Heapsort”, Communications of the ACM, Volume 7 Issue 6, June 1964, pp. 347–348.

[48] R.W. Floyd. “Algorithm 245 - Treesort 3”, Communications of the ACM, Volume 7 Issue 12, December 1964, pp. 701-702.

[49] E.W. Dijkstra. “Smoothsort, an alternative for sorting in situ”, Science of Computer Programming, Volume 1 Issue 3, 1982, pp. 223–233.

[50] J. Katajainen, T. Pasanen, J. Teuhola. “Practical In-Place Mergesort”, Nordic Journal of Computing, Volume 3, Number 1, Spring 1996, pp. 27-40

[51] C.A.R. Hoare. “Quicksort”, The Computer Journal, Volume 5 Issue 1, 1962, pp. 10-15.

[52] R. Sedgwick. “Implementing Quicksort programs”, Communications of the ACM, Volume 21 Issue 10, October 1978, pp. 847-857.

[53] D.R. Musser. “Introspective Sorting and Selection Algorithms”, Software - Practice and Experience, Volume 27 Number 8, August 1997, pp. 983-993.

[54] H.H. Seward. "2.4.6 Internal Sorting by Floating Digital Sort", Information sorting in the application of electronic digital computers to business operations, Master's thesis, Report R-232, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954, pp. 25–28.

[55] E.J. Isaac, R.C. Singleton. “Sorting by Address Calculation”, Journal of the ACM, Volume 3 Issue 3, July 1956, pp. 169-174.

[56] K.E. Batcher. “Sorting networks and their applications”, American Federation of Information Processing Societies Spring Joint Computer Conference (AFIPS'68), Atlantic City, NJ, USA, 30 April - 2 May, 1968, pp. 307-314.

[57] M. Ajtai, J. Komlos, E. Szemeredi. “An O(n log n) sorting network”, 15th annual ACM symposium on Theory of computing (STOC'83), Boston, Massachusetts, USA, April 25-27, 1983, pp. 1-9.

[58] J.V. Nobble. “Recurses!”, Computing in Science &Engineering, Volume 5 Issue 3, May/June 2003, pp. 76-81.

[59] I. Skliarova, V. Sklyarov. "Recursive versus Iterative Algorithms for Solving Combinatorial Search Problems in Hardware", 21st International Conference on VLSI Design (VLSI Design’2008), Hyderabad, India, January 4-8, 2008, pp. 255-260.

[60] G. Stitt, J.R. Villarreal, “Recursion flattening”, 18th ACM Great Lakes Symposium on VLSI, Orlando, Florida, USA, May 4-6, 2008, pp. 131-134.

[61] P. Ferreira, J.C. Ferreira, J.C. Alves. “Tail recursion in Hardware”, 25th Conference on Design of Circuits and Integrated Systems, Canary Islands, Spain, November 17-19, 2010.

[62] S. Vegdahl. “A survey of proposed architectures for the execution of functional languages”, IEEE Transactions on Computers, Volume C-33, Number 12, 1984, pp. 1050–1071.

[63] M. Sheeran. “Hardware design and functional programming: a perfect match”, Journal of Universal Computer Science, Volume 11, Number 7, 2005, pp.1135–1158.

[64] R. Sharp. “Higher-level hardware synthesis”, PhD Thesis, University of Cambridge, 2000-2002.

[65] M.P. Ward. “Proving program refinements and transformations”, Ph.D. Thesis, St. Annes College, Oxford, UK, 1989.

[66] M.P. Ward, K.H. Bennett, “Recursion Removal/Introduction by formal transformation: An aid to program development and program comprehension”, The Computer Journal, Volume 42 Number 8, August 1999, pp. 650–673.

[67] S.A. Edwards. “The Challenges of Synthesizing Hardware from C-Like Languages”, IEEE Design & Test of Computers, Volume 23 Issue 5, September-October 2006, pp.375-386.

[68] V. Sklyarov. “Hierarchical Finite-State Machines and their Use for Digital Control”, IEEE Transactions on VLSI Systems, Volume 7, Number 2, 1999, pp. 222-228.

[69] V. Sklyarov. “Synthesis of Circuits and Systems from Hierarchical and Parallel Specifications”, 12th Biennial Baltic Electronics Conference (BEC’10), Tallinn, Estonia, October 4-6, 2010, pp. 37-48.

[70] V. Sklyarov, I. Skliarova. "Reconfigurable Hierarchical Finite State Machines", 3rd International Conference on Autonomous Robots and Agents (ICARA'2006), Palmerston North, New Zealand, December 2006, pp. 599-604.

[71] V. Sklyarov, I. Skliarova. "Design and Implementation of Parallel Hierarchical Finite State Machines", 2nd International Conference on Communications and Electronics (HUT-ICCE’2008), Hoi An, Vietnam, June 2008, pp. 33-38.

[72] R. Mueller. “Data Stream Processing on Embedded Devices”, Ph.D. Thesis, ETH, Zurich, 2010.

[73] C.A.M. Marcon, N.L.V. Calazans, F.G.Moraes. “Requirements, Primitives and Models for Systems Specification”, 15th Symposium on Integrated Circuits and Systems Design, Brazil, 2002, pp. 323-328.

[74] B. Aparicio del Moral, J.M. Jerónimo Zafra, J.F. Rodríguez Gómez, R. Sanz Mesa, R. Morales Muñoz, A. Rodríguez Trinidad, J.J. López Moreno, The international Medusa team. “New Control System for Space Instruments. Application for Medusa Experiment”, 7th International Planetary Probe Workshop, Barcelona, Spain, June, 2010.

[75] D.M. Muñoz, C.H. Llanos, M. Ayala-Rincón, R.H. van Els. “Distributed approach to group control of elevator systems using fuzzy logic and FPGA implementation of dispatching algorithms”, Engineering Applications of Artificial Intelligence, Volume 21 Number 1, February 2008, pp. 1309-1320.

[76] V. Sklyarov, I. Skliarova, A. Neves. "Modeling and Implementation of Automatic System for Garage Control", ICCAS-SICE'2009, Fukuoka, Japan, August, 2009, pp. 4295-4300.

[77] S. Lee, S. Yoo, K. Shoi. “Reconfigurable SoC Design with Hierarchical FSM and Synchronous Dataflow Model”, 10th International Symposium on Hardware/software codesign, Estes Park, USA, May, 2002, pp. 199-204.

[78] S. Uchitel, J. Kramer, J. Magee. “Synthesis of Behavorial Models from Scenarios”, IEEE Transactions on Software Engineering, Volume 29 Issue 2, February 2003, pp. 99-115.

[79] J. Whittle, P.K. Jayaraman. “Generating Hierarchical State Machines from Use Case Charts”, 14th IEEE International Requirements Engineering Conference, Minneapolis, USA, September, 2006, pp. 16-25.

[80] D. Harel. “Statecharts: A visual formalism for complex systems”, Science of Computer Programming, Volume 8 Issue 3, June 1987, pp. 231-274.

[81] V. Sklyarov, I. Skliarova. "Modeling, Design, and Implementation of a Priority Buffer for Embedded Systems", 7th Asian Control Conference (ASCC’2009), Hong Kong, 2009, pp. 9-14.

[82] J.D. Davis, Z. Tan, F. Yu, L. Zhang. “A practical reconfigurable hardware accelerator for Boolean satisfiability solvers”, 45th Design Automation Conference, Anaheim, CA, USA, June 8-13, 2008, pp. 780-785.

[83] G.D. Hachtel, E. Macii, A. Pardo, F. Somenzi. "Markovian Analysis of Large Finite State Machines", IEEE Transactions on Computer-Aided Design, Volume 15, 1996, pp.1479-1493.

[84] W. Nőth, R. Kolla. “Spanning Tree Based State Encoding for Lower Power Dissipation”, Technical report, Department of Computer Science, University of Wűrzburg, 1998.

[85] X. Ye, D. Fan, W. Lin, N. Yuan, P. Ienne. "GPU-Warpsort: A Fast Comparison-based Sorting Algorithm on GPUs", IEEE International Parallel & Distributed Processing Symposium (IPDPS 2010), Atlanta, USA, April, 2010.

*Web resources (all listed URLs are valid as of December 2010)*

[86] Xilinx. Silicon Devices. FPGAs.
[*http://www.xilinx.com/products/silicon-devices/fpga/index.htm*]

[87] Altera. Altera Devices. FPGAs.
[*http://www.altera.com/devices/fpga/fpga-index.html*]

[88] SystemC.
[http://www.systemc.org/]

[89] listsort.txt.
[http://svn.python.org/projects/python/trunk/Objects/listsort.txt]

[90] Paul E. Black. U.S. National Institute of Standards and Technology. Dictionary of Algorithms and Data Structures. pigeonhole sort.
[http://xlinux.nist.gov/dads//HTML/pigeonholeSort.html]

[91] SD Card Association.
[http://www.sdcard.org]

[92] Nu Horizons Electronics.
[http://www.nuhorizons.com/]

[93] Xilinx. Software and Design Tools. ISE Design Suite.
[http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm]

[94] Digilent Inc. FPGA Boards. Nexys-2.
[http://digilentinc.com/Products/Detail.cfm?NavPath=2,400,789&Prod=NEXYS2]

[95] Java Applet on Stochastic FSM Encoding.
[http://www.pld.ttu.ee/applets/state/]

# Curriculum Vitae

**Personal data**

| | |
|---|---|
| Name | Dmitri Mihhailov |
| Date and place of birth | 01.11.1982<br>Tallinn, Estonia |
| Citizenship | Estonian |

**Contact data**

| | |
|---|---|
| Address | Raja 15, Tallinn 12618, Estonia |
| Phone | +372 620 2265 |
| E-mail | d.mihhailov@ttu.ee |

**Education**

| | |
|---|---|
| 2007 - ... | Ph.D. in Information and Communication Technology,<br>Tallinn University of Technology |
| 2005 - 2007 | M.Sc. in Computer and Systems Engineering,<br>Tallinn University of Technology |
| 2000 - 2005 | B.Sc. in Computer and Systems Engineering,<br>Tallinn University of Technology |

**Career**

| | |
|---|---|
| 2010 - ... | Researcher,<br>Chair of Digital Systems Design,<br>Department of Computer Engineering,<br>Faculty of Information Technology,<br>Tallinn University of Technology |
| 2007 - 2010 | Assistant,<br>Chair of Digital Systems Design,<br>Department of Computer Engineering,<br>Faculty of Information Technology,<br>Tallinn University of Technology |

**Honours & Awards**

"Tiger University" grant for ICT PhD students, Estonian Information Technology Foundation (EITSA), 2010

# Elulookirjeldus

**Isikuandmed**

| | |
|---|---|
| Nimi | Dmitri Mihhailov |
| Sünniaeg ja koht | 01.11.1982<br>Tallinn, Eesti |
| Kodakondsus | Eesti |

**Kontaktandmed**

| | |
|---|---|
| Aadress | Raja 15, Tallinn 12618, Eesti |
| Telefon | +372 620 2265 |
| E-post | d.mihhailov@ttu.ee |

**Hariduskäik**

| | |
|---|---|
| 2007 - ... | doktoriõpe, info- ja kommunikatsioonitehnoloogia õppekava, Tallinna Tehnikaülikool |
| 2005 - 2007 | tehnikateaduse magistri kraad; arvuti- ja süsteemitehnika õppekava, Tallinna Tehnikaülikool |
| 2000 - 2005 | tehnikateaduste bakalaureuse kraad; arvuti- ja süsteemitehnika õppekava, Tallinna Tehnikaülikool |

**Teenistuskäik**

| | |
|---|---|
| 2010 - ... | Teadur,<br>Digitaaltehnika õppetool,<br>Arvutitehnika instituut,<br>Infotehnoloogia teaduskond,<br>Tallinna Tehnikaülikool |
| 2007 - 2010 | Assistent,<br>Digitaaltehnika õppetool,<br>Arvutitehnika instituut,<br>Infotehnoloogia teaduskond,<br>Tallinna Tehnikaülikool |

**Teaduspreemiad ja -tunnustused**

"Tiigriülikooli" stipendium IKT doktorantidele (EITSA), 2010.a

# DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo.** Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.