



CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF TRANSPORTATION SCIENCES

Jakub Řada

IMPLEMENTATION OF THE MSC COMPRESSION
ALGORITHM IN FIELD PROGRAMMABLE GATE ARRAY

Master's thesis

2016



K614..... Department of Applied Informatics in Transportation

MASTER'S THESIS ASSIGNMENT

(PROJECT, WORK OF ART)

Student's name and surname (including degrees):

Bc. Jakub Řada

Code of study programme code and study field of the student:

N 3710 – IS – Intelligent Transport Systems

Theme title (in Czech): **Implementace kompresního algoritmu MSC v programovatelném hradlovém poli**

Theme title (in English): Implementation of the MSC Compress Algorithm in Field Programmable Gate Arrays

Guides for elaboration

During the elaboration of the master's thesis follow the outline below:

- Get acquainted with principles of logical circuits design in the field programmable gate array architecture
- Read up working principle of the Multistream Compression algorithm
- Design the block architecture of the compression module with regard to the implementation on a hardware platform
- Specify the requirements for selection of the field programmable gate array needed for the implementation
- Implement selected blocks

Graphical work range: according to supervisor's recommendations


Accompanying report length: min. 55 pages including figures, graphs and tables


Bibliography: IEEE Standard VHDL Language Reference Manual, IEEE Std 1076™-2008, New York, 2009
Kochanek J., et.all: Multistream Compression, Data Compression Conference, Snowbird, Utah, USA, 2008
Kilts S.: Advanced FPGA Design: Architecture, Implementation, and Optimization, John Wiley & Sons, 2007, ISBN 978-0470054376

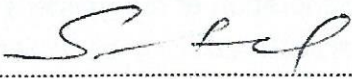
Master's thesis supervisor: **doc. Ing. Vít Fábera, Ph.D.**
Ing. Tomáš Musil, Ph.D.

Date of master's thesis assignment: **May 29, 2015**
(date of the first assignment of this work, that has be minimum of 10 months before the deadline of the theses submission based on the standard duration of the study)


Date of master's thesis submission: **November 30, 2016**
a) date of first anticipated submission of the thesis based on the standard study duration and the recommended study time schedule
b) in case of postponing the submission of the thesis, next submission date results from the recommended time schedule


doc. Dr. Ing. Tomáš Brandejský
head of the Department
of Applied Informatics in Transportation


prof. Dr. Ing. Miroslav Svítek, dr. h. c.
dean of the faculty



I confirm assumption of master's thesis assignment.


Bc. Jakub Řada
Student's name and signature

PragueJune 6, 2016

ACKNOWLEDGEMENT

I would like to thank both supervisors of the thesis – doc. Ing. Vít Fábera Ph.D. and Ing. Tomáš Musil Ph.D. for their constructive advices and suggestions and especially their willingness to spend so much of their valuable time on the project. Furthermore, I would like to thank all the people that supported me during my studies in any way.

DECLARATION

I declare that I have accomplished my final thesis by myself and I have named all the sources I had used in accordance with the guideline about the ethical rules during preparation of university final theses.

I have no relevant reason against using this schoolwork in the sense of §60 of Act No. 121/2000 concerning the authorial law.

Prague 27/11/2016


.....

signature

Title: Implementation of the MSC compression algorithm in Field Programmable Gate Array

Author: Bc. Jakub Řada

Department: Department of Applied Informatics in Transportation

Supervisors: doc. Ing. Vít Fábera, Ph.D., Ing. Tomáš Musil, Ph.D.

Supervisors' email addresses: fabera@fd.cvut.cz, musil@asix.cz

ABSTRACT

The thesis describes the first attempt of hardware implementation of Multistream Compression (MSC) algorithm. The algorithm is transformed to series of Finite State Machines with Data path using Register-Transfer methodology. Those state machines are then implemented in VHDL to selected FPGA platform. The algorithm utilizes a special tree data structure, called MSC tree. The thesis presents new way to store nodes of the tree using the Left Tree Representation. To encode data, the described implementation of the algorithm chooses between two methods – Elias Alpha and ZEBC.

KEYWORDS:

Multistream compression, Field Programmable Gate Array, MSC, FPGA, Compression, Parallel compression, Left Tree Representation

Contents

Contents	5
List of Figures.....	8
List of Tables	9
List of abbreviations	10
1 Introduction.....	11
1.1 Goals of the thesis.....	12
1.2 Organization of the thesis	12
1.3 Initial remarks	12
2 MSC Algorithm	14
2.1 The Steps of the Algorithm	14
2.1.1 Input Data Statistics	14
2.1.2 Creation of Binary Tree	15
2.1.3 Creation of Counter Streams	15
2.1.4 Statistical Analysis of Streams	16
2.1.5 Compression	16
2.2 The Coding Methods	16
2.2.1 Elias α	17
2.2.2 ZEBC	17
2.2.3 Huffman Coding	18
2.3 MSC Properties.....	18
3 RTL Design	19
3.1 Method Overview	19
3.2 RTL Design Steps.....	21
4 FPGA Architecture	22
5 Design Overview	27
5.1 Choice of the Platform.....	29
5.2 Implementation	32
5.3 Memory structure.....	33
5.4 SDRAM Interface	35

5.4.1 Signals.....	36
5.4.2 Pseudocode of SDRAM Controller	37
5.4.3 Block Diagram.....	38
5.5 Memory Arbiter	39
5.6 UART	40
6 Design details.....	41
6.1 Input of Data to FPGA Board.....	41
6.2 Creation of Statistics.....	42
6.3 Building the Tree	44
6.3.1 Representation of Binary Tree.....	44
6.3.2 Process of Tree Building	48
6.3.3 Memory Structure Used for Nodes.....	50
6.4 Determination of Subtrees	51
6.5 Creation of Streams	54
6.5.1 Specification of Counter Position in Thread Streams.....	55
6.5.2 Counter Statistics	57
6.6 Calculation of ZEBC Table	59
6.7 Parallel Analysis	59
6.7.1 Statistics processing.....	60
6.7.2 Coding analysis.....	61
6.7.3 Advance to next node	63
6.8 Buffer scheme.....	63
6.9 Compression	65
6.10 Output of Compressed Data.....	67
7 Results.....	68
8 Future Considerations	71
8.1 Sorting Algorithm.....	71
8.2 Decomposition of Tree to Layers	71
8.3 Huffman Coding	72
9 Conclusion	73

10 References.....	74
Appendix	78

List of Figures

- Figure 1 Example of MSC tree in default setting
- Figure 2 Block diagram of a general system designed by RT methodology
- Figure 3 Architecture of a LUT
- Figure 4 Comparison of Xilinx's and Altera's architecture
- Figure 5 CLB Array and Interconnect channels of Xilinx
- Figure 6 General overview of Xilinx FPGA architecture
- Figure 7 The structure of MSC algorithm hardware implementation
- Figure 8 Xilinx ISE® WebPACK™ development environment
- Figure 9 Xilinx ISim simulation environment
- Figure 10 Architecture of the MSC design
- Figure 11 The SDRAM interface
- Figure 12 Memory arbiter
- Figure 13 Structure of data sent over UART
- Figure 14 Dataflow diagram of SDRAM interface during input data writing
- Figure 15 Dataflow on SDRAM interface during creation of statistics
- Figure 16 Example of MSC tree for "abracadabra"
- Figure 17 Left tree of "brcd" node
- Figure 18 Traversing of tree during transformation from LR to LTR
- Figure 19 Chosen thread root if $no_of_threads \geq 1$
- Figure 20 Alternative choices for thread root 1
- Figure 21 Alternative choices for thread root 2
- Figure 22 Alternative choices for thread root 3
- Figure 23 Selection of subtrees for a random tree
- Figure 24 Dataflow of SDRAM interface during creation of streams
- Figure 25 Example of MSC tree
- Figure 26 STATISTICS manipulation during analysis phase
- Figure 27 Dataflow on SDRAM interface during coding phase
- Figure 28 Workplace

List of Tables

- Table 1 Constructs used in pseudo language
- Table 2 Comparison of FPGA models
- Table 3 Resource estimation
- Table 4 CMD signal
- Table 5 SIZE signal
- Table 6 Attributes of used TYPES of data
- Table 7 Sequence of bytes sent from PC to FPGA
- Table 8 Structure of INPUT DATA STATISTICS in BRAM
- Table 9 a), b) Squeezing statistics
- Table 10 Sequential representation of nodes in memory
- Table 11 Linked representation of nodes in memory
- Table 12 Left tree representation of nodes in memory
- Table 13 First block of memory containing node attributes (NODE1)
- Table 14 Second block of memory containing node attributes (NODE2)
- Table 15 Third block of memory containing node attributes (NODE3)
- Table 16 Fourth block of memory containing node attributes (NODE4)
- Table 17 Types of threads
- Table 18 ZEBC TABLE memory structure
- Table 19 Examples of values coded by ZEBC with different bases
- Table 20 MSC Overhead
- Table 21 Header format of node's first coded counter
- Table 22 Identifier of different thread
- Table 23 FPGA resource utilization
- Table 24 IOB utilization

List of abbreviations

ALM	Adaptive Logic Module
ALUT	Adaptive Look-Up Table
ASIC	Application-Specific Integrated Circuit
ASMD	Algorithmic State Machine with Data path
BC	Binary Complement
BRAM	Block Random-Access Memory
BWT	Burrows–Wheeler transform
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
DCM	Digital Clock Manager
DLL	Delay Locked Loop
DSP	Digital Signal Processing
DSRC/WAVE	Dedicated Short Range Communications/Wireless Access in Vehicular Environments
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FSMD	Finite State Machine with Data path
IO	Input/Output
IOB, IOE	Input/Output Block, Element
JPEG	Joint Photographic Experts Group
JTAG	Joint Test Action Group
LAB	Logic Array Block
LC/LE	Logic Cell/Logic Element
LR	Linked Representation
LSB	Least Significant Bit
LTR	Left Tree Representation
LUT	Look-Up Table
MSB	Most Significant Bit
MSC	Multi Stream Compression
PC	Personal Computer
PLD	Programmable Logic Device
PLL	Phase Locked Loop
RAM	Random-Access Memory
RT	Register Transfer
RTL	Register Transfer Level
SDRAM	Synchronous Dynamic Random-Access Memory
SRAM	Static Random-Access Memory
SRL16	16-bit shift register
SRL32	32-bit shift register
UART	Universal Asynchronous Receiver/Transmitter
VHDL	VHSIC Hardware Description Language
XML	Extensible Markup Language
ZE	Zero Ending
ZEBC	Zero Ending & Binary Complement

1 Introduction

Information science has affected nearly all levels of modern society. Nowadays, the world is approaching the stage in which every device is going to be connected to a global network. Indispensable amount of research concerning the so-called Internet of Things conducted so far provides an opportunity for customized manufacturing, safer transport and more comfortable living with ever-increasing automation. These put above a big demand on communication infrastructure to cope with an enormous volume of data.

Compression is a way to lower this volume. During the last few decades, numerous compression algorithms were invented, ranging from the universal ones (such as Huffman Coding or Arithmetic coding) to those designed for specific data (for example, JPEG, MP3). In pursuit of maximum compression ratio and speed, new compression methods are still being developed.

One of the recently developed compression methods is Multistream Compression (MSC) algorithm [1]. In contrast to other methods, MSC encodes a special stream of counters to achieve decent compression ratio. Its design allows for parallel processing to reduce the processing time. In some applications, this method can be used as a replacement of some existing methods (Huffman or Arithmetic coding) as it often gives better outcome. Besides, this method gives favorable results if applied in conjunction with various transformations (such as Burrows-Wheeler Transform) used for input data preprocessing. Main application areas of the MSC algorithm are text and image compression. One of its utilizations could thus be found in compression of document databases. Analysis of text compression using the MSC algorithm is shown in [2]. The experiment using the MSC algorithm for compression of JPEG images can be found in [3].

Due to its wide area of application, MSC is concern of transportation sciences as well. Nowadays, transport is already extensively computerized – there are information networks in vehicles, communication links between the infrastructure and control centers, and many others. The implementation of cooperative systems is inevitably causing a huge increase of transferred data on the infrastructure. Despite the fact that MSC algorithm does not find use in compressing XML or fixed-length binary encoded messages used by DSRC/WAVE standard [4], there are other possibilities of its utilization such as formerly mentioned database or image compression.

1.1 Goals of the thesis

The goal of this thesis is to design and implement MSC algorithm on FPGA platform. Main features of the algorithm are complex tree structure, high memory demand and parallel processing.

One of the main tasks is to find representation of the used tree structure in memory that would possess advantageous properties for latter stages of the algorithm. Further task is to specify the hardware requirements for implementation of the algorithm with given input parameters. The final objective is to implement the design to highest possible extent into selected platform and verify its functionality.

This thesis should serve as a stepping-stone for further development of the hardware implementation of the algorithm and as an inspiration for building the decompression module which is not subject of this thesis. Obtained results of this implementation shall thus be presented in order to provide objective measure for monitoring future improvements.

1.2 Organization of the thesis

The content of the thesis is conceptually divided into two parts. In first half, the presented information is exclusively theoretical and goes from the ideas to be implemented (chapter 2), through the design methodology that describes how the ideas to be implemented have to be transformed to fit the used platform (chapter 3), to the description of actual hardware platform that physically realizes the ideas (chapter 4). The other half puts emphasis on the practical work although short sections of theory are also contained. The chapter 5 describes mainly the hardware aspects of the overall design and chapter 6 discusses the details of the individual parts with emphasis on the ideas. Chapter 7 analyzes the results of the implementation and chapter 8 gives incentives for further work.

1.3 Initial remarks

Before reading, some features of the thesis should be mentioned. The reader should beware of similar names for different entities. The MSC algorithm for example creates statistics more than once, so terms like ‘statistics’ or ‘number of occurrences’ are used in different contexts.

For an easy orientation, different fonts and capital letters are used in the chapters concerning the algorithm design. CAPITAL LETTERS are used to denote Finite State Machines and memories. *Arial Narrow in italics* is used for attributes in memory.

Throughout the work, snippets of code using pseudo language are used. The occurring constructs are explained in Table 1 below:

Table 1 Constructs used in pseudo language

Construct	Example	Meaning
==/!=	<code>while (end == 0)</code>	equal to / not equal to in conditions
=	<code>last = 1</code>	assignment to variable
(x) / (x downto y)	<code>data(FREE-1 downto 0) = (others => '1')</code>	index / range of indexes
(others => '1')		assignment of specified value to range of indexes
⇒	⇒ <code>read DATA_OUT(TYPE, INDEX, POSITION)</code>	do specified operation
var(x, y, ..)		do specified operation with parameters in brackets (for example: read DATA_OUT from memory address specified by the parameters)
.	<code>CURRENT_NODE.left_child</code>	attributes of memory item (for example: attribute left_child of item CURRENT_NODE)

2 MSC Algorithm

The MSC algorithm is a new lossless compression method invented by Czech scientist Jiří Kochánek [1]. The method is based on the idea that data can be split into different parts. Each of these parts contains own data for compression arranged in streams. For each part of the data, coding method that gives the best compression result is chosen. The MSC algorithm differs from other models based on splitting data into streams by the fact that in this case the streams do not contain symbols but counters. The process of compression is rather complex and requires multiple passes of input data. However, the algorithm presents the possibility of parallel processing.

2.1 The Steps of the Algorithm

The algorithm consists of these five steps [3]:

- ❑ Input data statistics
- ❑ Creation of binary tree based on statistics
- ❑ Transformation of input data into logical streams belonging to single binary node
- ❑ Statistical analysis of each stream ending with selection of the best performance compression method
- ❑ Compression execution

For simplicity, the description will be given only for sequential processing. The details about the parallel processing will be given in further chapters discussing the implementation.

2.1.1 Input Data Statistics

The algorithm starts with reading characters¹ from the input data, one at a time. The input data of the algorithm could be a text or a picture in which case the symbol would be a letter or pixel color, respectively. For each symbol occurring in the input data, its first occurrence as well as number of occurrences is determined. After all input data are read, the statistics is sorted according to the number of occurrences.

¹ Character is one item in the input data. Symbol is a value of the character. Alphabet is a set of all symbols.

2.1.2 Creation of Binary Tree

When the statistics is in the desired form it serves as a basis for building MSC tree. The items of the statistics become the leaves of the binary tree. In this stage, parents of two nodes with the lowest number of occurrences are created iteratively until there is only one unconnected node remaining – the root of the tree. The tree might resemble Huffman tree, however, the MSC tree is governed by additional rule which states that every left child has earlier occurrence in the data than the right one, even if it has lower number of occurrences. Moreover, the nodes are equipped with counters and direction switches.

2.1.3 Creation of Counter Streams

Initially, before the traversing of the tree starts, the counter of each node is set to zero and each switch position set to left as shown on Figure 1. The switch determines the active child of a node and thus one active path is built in a tree from root to leaf. During creation of streams the input data are read for the second time. Knowing the investigated symbol, the tree is traversed from the root node all the way to the leaf node representing this symbol. Each time a node is passed its counter is incremented. If two consecutive symbols are different, the direction switches need to be switched to other direction in those nodes where the current path differs from the previous path. The rule is that when the position of a switch is swapped, the counter value is written into stream of the particular node.

When the reading of input data is finished, there are still nonzero values in node counters which need to be written to streams as well. Therefore, each node is entered once again. The values in streams are actually the data that will be compressed.

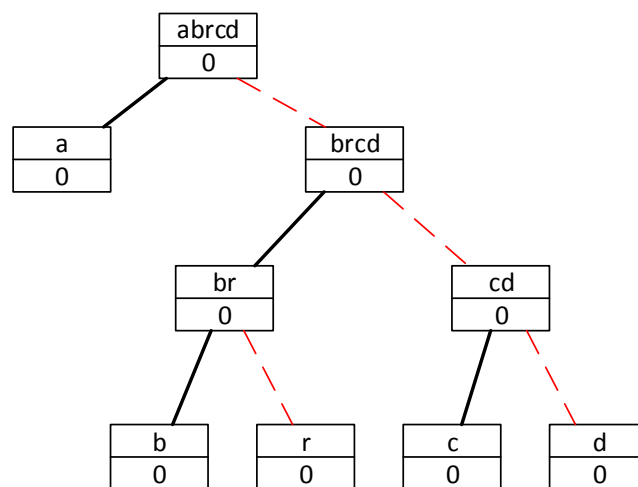


Figure 1 Example of MSC tree in default setting

2.1.4 Statistical Analysis of Streams

After obtaining the streams, the length of the compressed data is calculated for different coding methods and the method giving the best result is selected. The coding methods are specified in advance. The analysis is executed for each node of the tree separately and the best method is selected for every single one. If ZEBC coding (section 2.2.2) is among the investigated methods, the analysis is performed for whole range of bases. In case of Huffman coding, Huffman tree needs to be built for every node of the MSC tree.

2.1.5 Compression

At this stage, the streams of counters are known as well as the method of coding. First, the overhead of the final stream of coded values is written to the output. It carries information required for correct decompression. Some of the items in the overhead are input parameters, others are obtained at the compression module.

For creation of the final compressed stream, traversing of the MSC tree is executed once again. At the beginning, the counter of root node is set to number of characters in input data, counters of other nodes are reset and switches are set to left. The tree is traversed from the root to leaf nodes using the active path until the counter of root node is equal to zero. In case the counter at the entrance of a node has value 0, next value from the stream of counters is read, coded and written to the final stream of coded values. If a node is accessed for the first time, additional information (consisting of leaf node flag, symbol in case of leaf, the chosen coding method and its parameters) is written to the output before the coded counter value. Each time a node is accessed, its counter is decremented. When it reaches zero, the switch position of the parent node is changed.

2.2 The Coding Methods

The previous paragraphs imply that the MSC algorithm chooses among different coding methods. Basically, they can be arbitrary methods. However, the criteria for the convenience of the method can differ. Sometimes, the speed of the decompression is crucial, in other cases the achievement of the maximal compression ratio is desired. Depending on criteria set of methods is adequately adjusted. In the first software application a set of 3 methods that appear to be complementary was tested.

2.2.1 Elias α

Elias Alpha coding is usually used for short or badly compressible streams. If all streams are coded by Elias α , the output data has equal length to the same data compressed by Huffman coding. In the MSC algorithm, the inverted version of this unary method is used. If a number n is coded by this method, it will be of the following form [5]:

$$\alpha(n) = 1_1 1_2 \dots 1_{n-1} 0_n$$

2.2.2 ZEBC

Zero Ending & Binary Complement (ZEBC) is a simple way of coding similar to Elias γ . It was developed specifically for MSC method by the authors of the algorithm [1]. It is parameterized by a selected number, the base – ZEBC(b), which further optimizes the coding of stream. Selection of this number is based on analysis of counters. This method is mostly used for compression of streams of middle length.

The ZEBC coding is a type of variable-length coding that is used for coding natural numbers. The coded number comprises of Zero Ending (ZE) prefix coded by inverted Elias α coding and of a value expressed by the binary complement (BC) but only if the number to be coded is greater than the base value. If the number is lower than the base, it is expressed only in ZE code. The base can be viewed as a number that determines the boundary, from which the coded value is expressed by ZEBC code.

The coding is performed according to ZEBC table of intervals (Appendix A). Construction of the table is governed by a set of elementary rules. Each interval has beginning and end. The values of the first table interval are known – the beginning is 0 and the end is 1. The index i denoting the intervals goes from zero onwards. The calculation of the i -th interval is governed by these rules:

- $i_{k+1} = i_k + 1$
- $beginning(i_{k+1}) = end(i_k) + 1$
- $end(i_{k+1}) = beginning(i_{k+1}) + (2 * (end(i_k) - beginning(i_k) + 1)) - 1$

When number n is coded by ZEBC(b), these rules are followed: if the number n is lower than base b , the number n is expressed only in ZE code. If it is larger or equal to b , then the difference D is calculated as $D = n - b$ and the table interval i into which D belongs is found. The ZEBC(b) code of the number is then expressed by sequence $(b + i)$ in ZE code, followed by number $(D - beginning(i))$ in BC code of length $(i+1)$ bits.

2.2.3 Huffman Coding

This method is intended for coding of largest streams. It is because the first counter of each tree node coded by this method contains a large header containing all the necessary information for reconstruction of the Huffman tree. Therefore, if a large stream is coded, the percentage of the header in the final compressed stream is low.

The method uses the static Huffman coding for coding of counters, meaning the number of occurrences of each symbol is known in advance.

2.3 MSC Properties

Since the invention of the first compression methods, many new methods were discovered. It is because none of the methods is the best one for all types of data and for various requirements and every single one has its benefits and drawbacks. MSC is not an exception. The list of the advantages and disadvantages is below [6].

ADVANTAGES

- Higher compression efficiency than pure entropy methods for many types of data. Particularly suitable as replacement of Huffman and arithmetic coding in codecs, endecs (MPEG and JPEG types) and after BWT
- Decompression is markedly fast with minimum requirements for memory size
- Method allows both sequential and parallel processing
- Parallel processing can fully exploit the nowadays progress in component base (multi-core processors, FPGA, CPLD,..)
- The data is processed without multiplication and division (unlike arithmetic method), which is particularly advantageous for small devices lacking mathematic co-processors (e.g. mobile phones)
- The development of this method is only in its early stage with still huge potential for improvement

DISADVANTAGES

- Multiple passing - compression consist of 2 passes through input data and one pass through transformed data
- High requirements on system components (memory size) during compression phase
- Slower compression phase than current entropy methods

3 RTL Design

As hardware systems nowadays are very complex, it is impossible to create a design considering the connection among transistors. In the modern era, logic gates as blocks of higher abstraction level are used to realize logic functions. For large designs, however, those are too detailed as there is a possibility in hardware to realize commonly used higher level operations that can be created from the logic gates. The example of such high abstraction blocks are adders, comparators, multiplexers or registers. This level of abstraction is called Register-Transfer Level (RTL).

3.1 Method Overview

In the area of informatics, complex processes are usually described by algorithms, as sequences of steps or actions. Nowadays, algorithms are generally implemented by software and executed in a general purpose computer. However, to obtain better performance and efficiency, it is sometimes beneficial or even necessary to realize them in hardware. The Register-Transfer (RT) methodology is a design methodology that describes the system operation by a sequence of data transfers and manipulations among the registers.

To realize an algorithm in hardware, general hardware constructs that resemble the variable and sequential execution model are needed. The RT methodology is aimed for this purpose. The key characteristics of this methodology are:

- ❑ Registers are used to store the intermediate data and are equivalent to variables used in an algorithm
- ❑ Data path is used to realize all the required register operations
- ❑ Control path is used to specify the order of the register operations

When an algorithm is realized by RT methodology, the necessary data manipulation and data routing are performed by dedicated hardware blocks. The data manipulation circuit, routing network and the registers together are known as the data path. Since an algorithm is described as a sequence of actions, a circuit to control the RT operations flow is needed, and it is known as the control path. Both blocks with connections between them are shown on Figure 2.

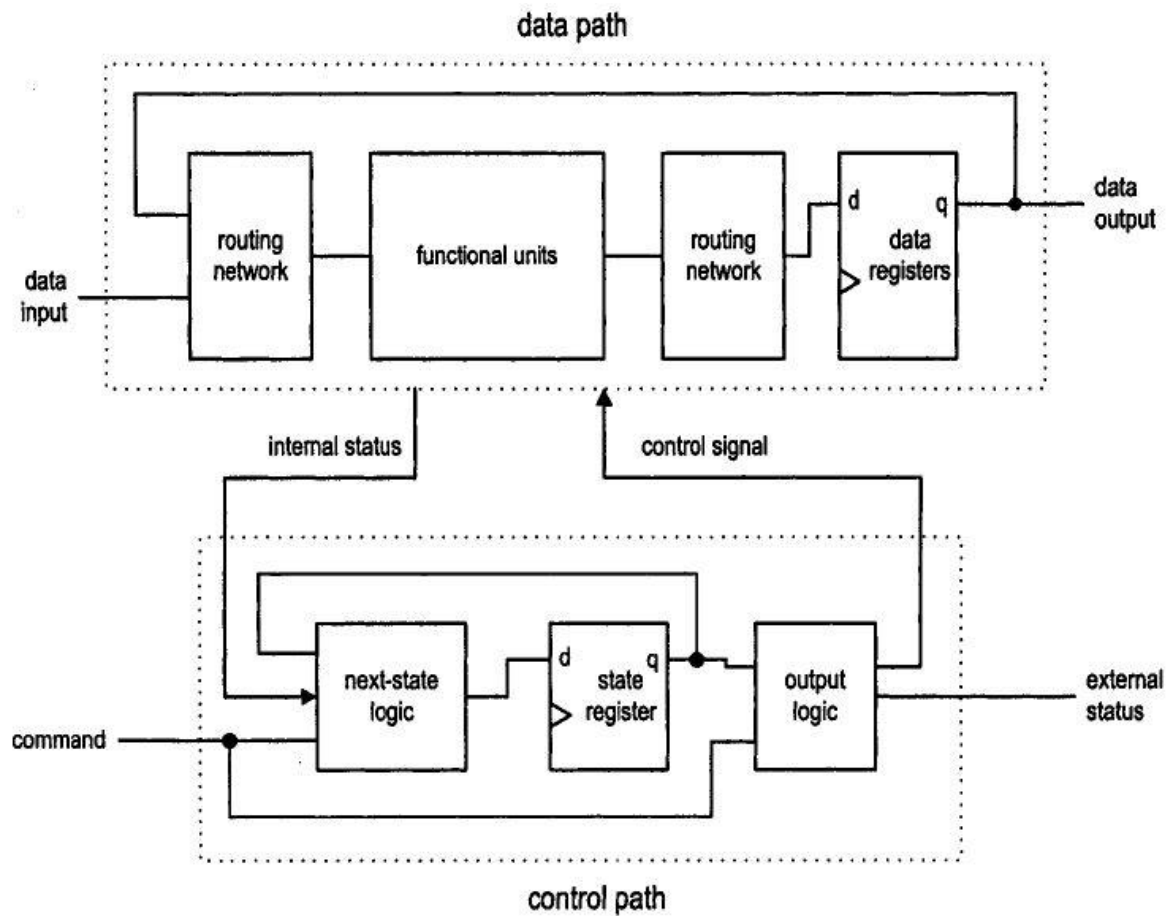


Figure 2 Block diagram of a general system designed by RT methodology (source: [7])

A control path is usually realized by a Finite State Machine (FSM), which uses states to enforce the order of the desired steps and branches, which are equivalent to conditions and loops in an algorithm. A FSM consists of two elements: combinatorial logic and registers. The registers are used to store the state of the machine. The combinatorial logic can be viewed as two distinct functional blocks: the next state logic and the output logic. There are two widely known types of state machines: Mealy and Moore. The output function which specifies the value of the output signals is where the two types differ. If it is a function of the state only, the output is known as a Moore output. On the other hand, if it is a function of the state and input signals, the output is known as a Mealy output. A complex FSM normally has both types of outputs.

Since an RT operation is performed in a state of the FSM, the FSM is extended to Finite State Machine with Data path (FSMD) to indicate the desired RT operation in each state. The state representation and state transition of an FSMD are similar to those of an FSM. An Algorithmic state machine with data path (ASMD) chart is a usual way of describing FSMD [7].

Most architectures require both control and some data. Only very few of them are either all control (for example, simple communications protocols) or all data (digital filters). Separating the design into a controller and a data path helps to think about the operation of the system [8].

3.2 RTL Design Steps

Frank Vahid [9] recognizes 4 steps in the RTL design.

1. The desired behavior of the system shall be described as high-level state machine. The state machine is high-level as the transition conditions and the state actions are more than just Boolean operations on bit inputs and outputs.
2. A data path to carry out the data operations of the high-level state machine is created.
3. Data path, specifically the external Boolean inputs and outputs are connected to a controller block.
4. High-level state machine is converted to FSM for the controller, by replacing data operations with setting and reading of control signals to and from data path.

In case of FPGAs (and other Programmable Logic Devices) the hardware description of the circuit is the first step in the implementation, the description is coded and after that synthesized, technology mapped and packed usually by using software tools. At the end, the circuit is placed and routed to the hardware platform to complete the design flow.

4 FPGA Architecture

Field Programmable Gate Arrays (FPGAs) are pre-fabricated silicon devices that can be electrically programmed. FPGAs consist of an array of different blocks, including general purpose logic blocks and specific purpose hard blocks, such as memory or DSP blocks. General purpose logic blocks are programmable and along with specific purpose hard blocks they are surrounded by a programmable routing fabric that allows these blocks to be programmably interconnected. The array of blocks and the routing fabric are then surrounded by programmable input/output blocks that connect the chip to the outside world. In the following text, architectures of two biggest FPGA vendors Xilinx and Altera will be described. The other manufacturers use similar architecture.

A configurable logic block (CLB) used by Xilinx is a basic component of their FPGAs that further divides into slices. The slices contain blocks performing combinatorial logic, register resources and multiplexors [10]. The elementary building block of Altera's devices is called Logic Array Block (LAB) and it consists of ten Adaptive Logic Modules (ALM) or 16 logic elements.

Commercial vendors use Look-up Tables (LUTs) to provide basic logic and storage functionality. LUT-based blocks provide a good trade-off between too fine-grained and too coarse-grained logic blocks, ensuring that the wasting of resources is minimal while maintaining high performance. The capacity of a LUT is limited by the number of inputs, not by the complexity of a function. Unlike the logic realized by gates, the delay through the LUT is constant, regardless of what logic function is being performed inside.

A LUT is typically built out of SRAM bits to hold the configuration memory LUT-mask and a set of multiplexers to select the bit of the SRAM that matches the inputs. To implement a k-input LUT (k-LUT), 2^k SRAM bits and a $(2^k - 1)$ multiplexers are needed. Figure 3 shows a 4-LUT, which consists of 16 bits of SRAM and a 16:1 multiplexer implemented as a tree of 2:1 multiplexers [11].

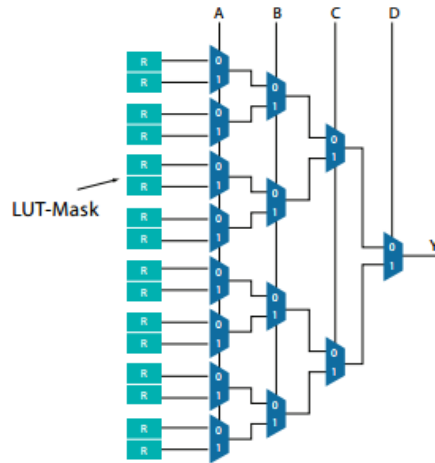


Figure 3 Architecture of a LUT (source [11])

The parameters of various FPGAs differ beyond the names of different vendor components. The Xilinx devices encompass 6-input LUTs and two or three types of slices depending on platform. The LUTs of the most basic slice type can be configured as either a 6-input LUT with one output, or as dual 5-input LUTs with identical 5-bit addresses (shared inputs) and two independent outputs. In contrast, the LUTs in the most complex slice type can also be used as distributed 64-bit RAM with 64 bits or two times 32 bits per LUT, as a single 32-bit shift register (SRL32), or as two 16-bit shift registers (SRL16s) with addressable length [10].

Unlike Xilinx, Altera uses two types of architecture in their model series. The architecture used in majority of their devices utilizes a component called Adaptive LUT (ALUT) with 8 inputs that provides resources for realizing either 6-input LUT or two LUTs that can work independently in case the total number of inputs is less or equal to 8, otherwise the inputs need to be shared. The number of shared inputs depends on widths of the realized LUTs. Two extra adders are included to enhance the arithmetic capability of the ALM, allowing for two 2-bit addition or two 3-bit addition per ALM. The comparison of the two architectures is shown on simplified schemes on Figure 4.

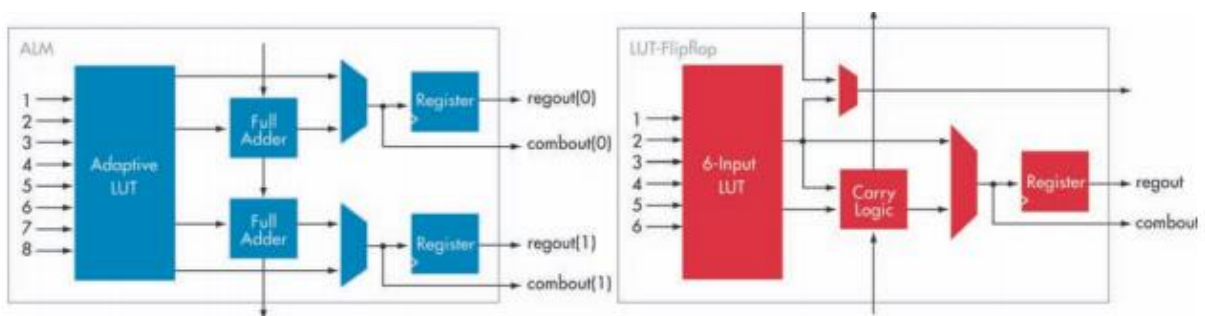


Figure 4 Comparison of Xilinx's and Altera's architecture (source [11])

The other Altera's architecture utilizes a 4-input LUT, however, a gradual shift to the formerly mentioned architecture is visible [11].

Besides LUTs and registers, basic building blocks of devices from both vendors also have carry logic resources which are designed to implement arithmetic logic functions with high speed performance due to their dedicated carry chain which runs vertically in columns.

The input/output blocks or elements (IOBs/IOEs), depending on a vendor, make an interface between the FPGA and the outside world. In Xilinx devices, IOBs are grouped into IO banks located on each edge of the device. The IOBs contain registers and some specialized resources. One of the purposes of the IOBs is to clock data in and out of the FPGA.

Besides registers, the IOBs contain interface logic that is designed to translate the internal voltage domain of the FPGA to any used I/O standard.

Spartan 6 has 4-6 IO banks, depending on the device density and each IO bank has between 30 and 83 IO pins [12].

Interconnect is a programmable network of signal pathways between the inputs and outputs of functional elements within the FPGA, such as IOBs, CLBs, DSP slices, and block RAMs. Interconnect, also called routing, is segmented for optimal connectivity. The interconnect (besides the carry logic resources) is routed through the switch matrices. The switch matrices are in between each function block and they are designed to connect to other switch matrices as well as neighboring blocks as shown on Figure 5 [10].

The implementation tools select an appropriate routing that may include different kinds of routing resources. The result of the routing is designed to try and meet the timing needs of design. It is important that timing constraints are specified so that the timing objectives can be communicated to the implementation tools, allowing the software to choose the smartest routing solution to meet the needs.

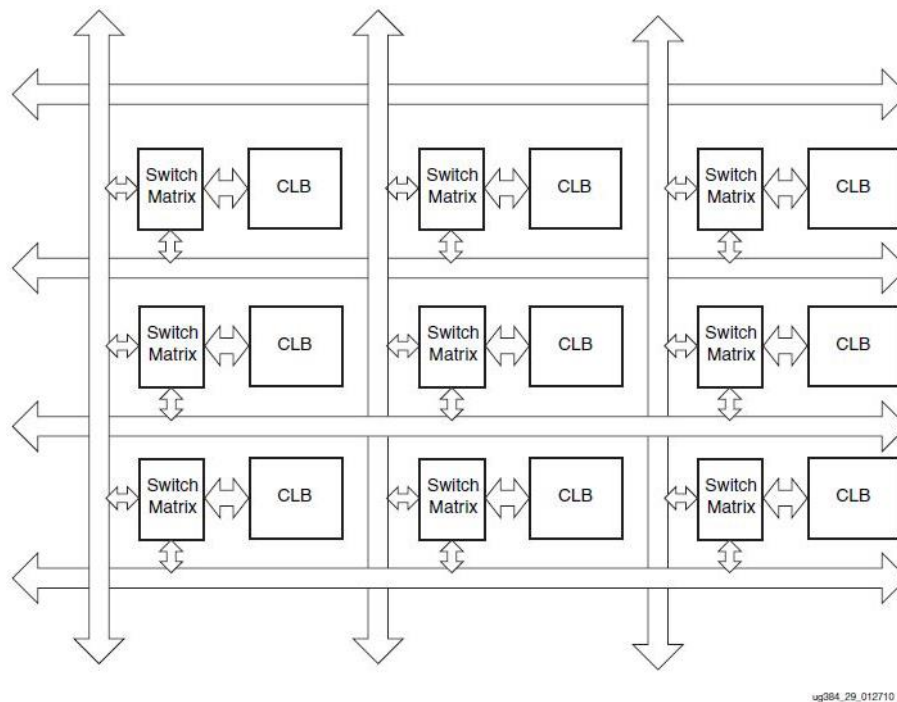


Figure 5 CLB Array and Interconnect channels of Xilinx (source: [10])

Besides the LUTs, which can be configured as small memory blocks for coefficient storage and low capacity buffering, FPGAs contain also dedicated blocks of RAM memory. These memories can operate in true dual-port mode, meaning they have two ports A and B and either of these ports can be a WRITE or READ port, independent of the function on the other port, sharing only the stored data. In the dedicated memories, WRITE and READ are synchronous operations. Data bus of each port can be configured in one of the available widths, independent of the other port.

The Xilinx block RAM (BRAM) is an 18 or 36 Kb block of memory, depending on the device, which can also be configured as two independent 9/18 Kb blocks. The maximum width of the memory data bus is 36/72 bits containing a parity bit for each byte [13]. Each Altera device uses one of the memories labeled as M20K and M10K with maximum data bus width of 40/20 bits (20-bit width applies for true dual port mode) same for both types and capacities of 20 Kb and 10 Kb, respectively. Unlike the Xilinx block, the M20K and M10K blocks cannot be split [14], [15]. High-end platforms from Xilinx also contain dedicated logic for cascading BRAMs thus creating Ultra RAM blocks of maximum capacity 288 Kb.

Today's FPGAs contain more types of clock networks that include at least global and regional networks. The networks are designed to minimize clock skew by optimized integrated circuit layout using buffers and multiplexors, and also some further techniques are employed such as Phase Locked Loop (PLL).

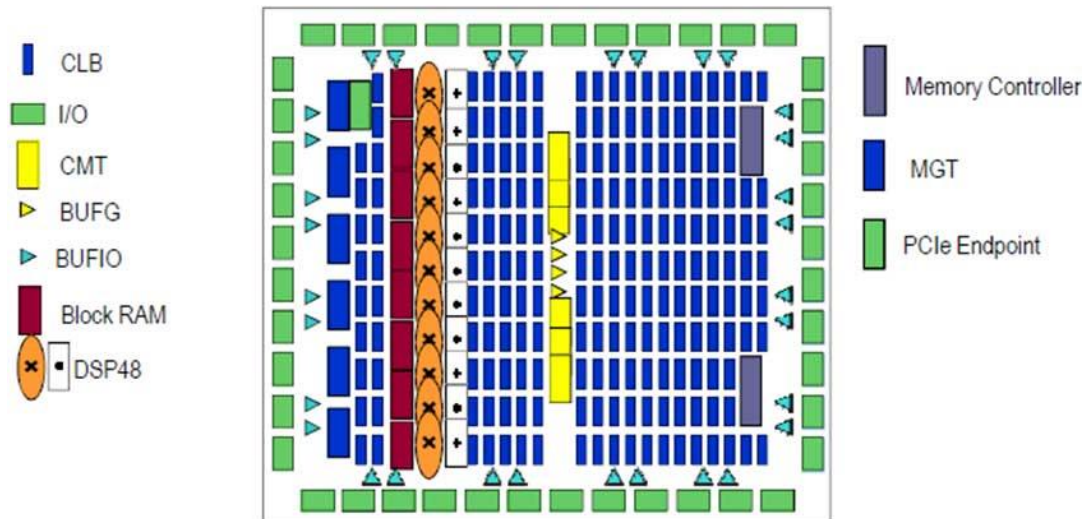


Figure 6 General overview of Xilinx FPGA architecture (source: <https://www.pantechsolutions.net>)

5 Design Overview

Chapter 3 showed the difference between the way an algorithm is implemented in software and hardware. Before the hardware implementation, described in this thesis, was performed, there had already been software implementation of the MSC compression algorithm in C language. Analysis of its source code was performed as a first step of the hardware implementation. It was necessary to familiarize with the algorithm very closely and to understand it to smallest details.

The analysis served as a basis for rewriting the code into pseudocode that removed all the features that are used in software but cannot be used in hardware, such as pointers. The pseudocode was written in a form that would allow direct inference of a Finite State Machines and that considered division of data to memory structures. Besides, new features had to be introduced, such as new representation of tree, for convenient mapping to hardware. Example of such pseudocode is shown at the end of this page. Pseudocodes for all parts of the algorithm are attached in Appendix B.

At this point, memory demand of the design was already known so it was required to map these memory requirements into the resources, as described in section 5.1.

At the end, Finite State Machines (with data path) were inferred for logically divided parts of the algorithm, according to the pseudocode. The FSMs aimed at minimizing number of states and using the minimum number of clock cycles so that each part of the algorithm took as little time as possible. They were then translated into VHDL code, continuously simulated to verify functionality and synthesized. The results of the synthesis served as a basis for choosing a suitable FPGA. At the end, the algorithm was programmed into the chosen FPGA.

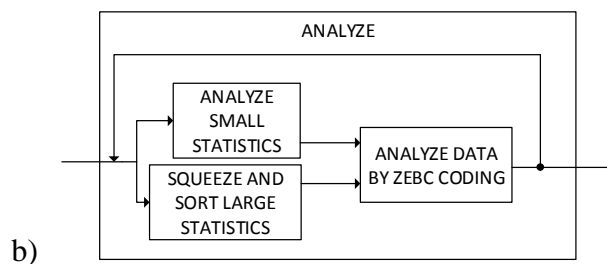
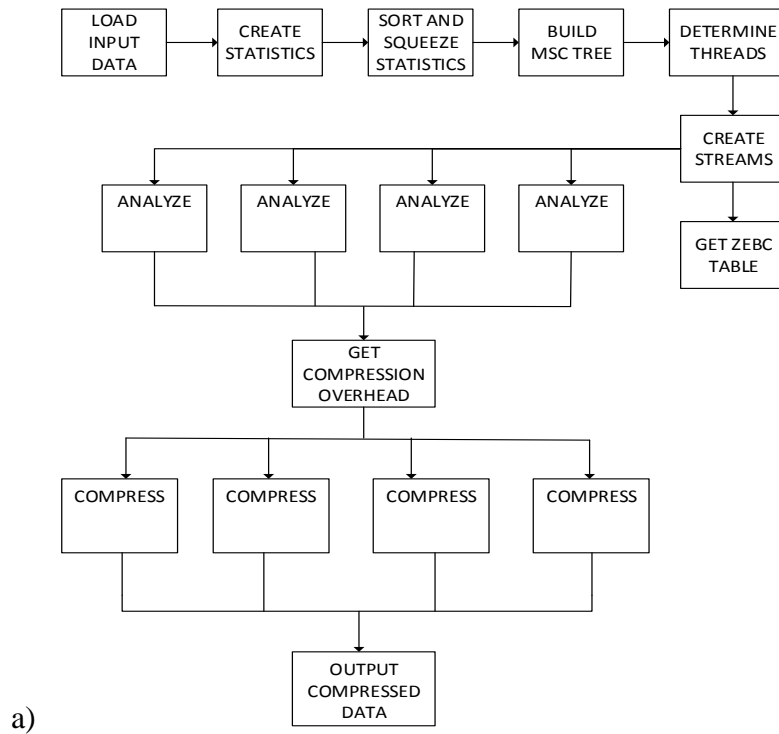
```
i = 0
while (i < SID)
    symbol = INPUT_DATA(i)
    no_of_occ = STAT(symbol).no_of_occ
    no_of_occ = no_of_occ + 1
    if (no_of_occ == 1)
        STAT(symbol).symbol = symbol
        STAT(symbol).first_occ = i
        STAT(symbol).sum = 1
    STAT(symbol).no_of_occ = no_of_occ
    i = i + 1
```

The figure 7a) shows the structure and flow of the hardware implementation of MSC algorithm. Rectangles symbolize FSMs, which are instantiated as components in VHDL language. The arrows symbolize the transition among the FSMs implemented by control signals. The FSMs that follow after parallel processing are launched as soon as processing in the slowest parallel block is finished.

The parallel blocks run concurrently, however, all of them need not be utilized. The number of used blocks is fed into the algorithm as one of the input parameters.

Some of the FSMs are further divided into subcomponents as figures 7b) and 7c) suggest. The ANALYZE block utilizes the subcomponents as steps, whereas the COMPRESS block uses its subcomponents when they are required.

The algorithm is implemented completely in parallel, which means that no resources besides memory are explicitly shared.



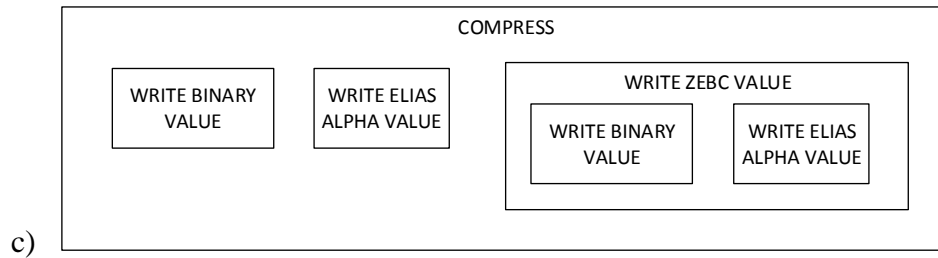


Figure 7 The structure of MSC algorithm hardware implementation

5.1 Choice of the Platform

There is a variety of FPGA manufacturers that offer a broad range of products with different number of resources suitable for diverse applications. The number of logic cells in contemporary FPGAs lies in the range from few thousand to several million [16], [17], therefore the designer needs to make a selection of appropriate density as an integral part of a design. Even the features (such as LUT width) of similar sized platforms are different.

Generally, the choice of the FPGA platform is constrained by cost and physical requirements. There is not one specific method that is used for selection of the right-sized FPGA, however, in the design process, as [18] suggests, one shall estimate the number of:

- ❑ Look-up Tables, Flip-flops, RAM bits, DSP Blocks
- ❑ IOs for each voltage level (3.3, 2.5, 1.8V...)
- ❑ Clock pins, PLL/DLL/DCM, Clock buffers required (regional or global)

In case an FPGA board is used, the electrical wiring is already provided. The pins of the FPGA dedicated for power supply and configuration are on the board wired to connectors that form the interface with outside world. The same applies for communication interfaces.

With utilization of two clock domains, FPGAs provides enough clock resources for the implemented design of the algorithm, which leaves us with only need for specification of memory size, number of registers and logic elements, when there are no further requirements.

In this section, numbers of these resources that are needed for implementation of MSC algorithm are estimated. The estimate will be very rough due to different physical implementations on different platforms, various optimizations, signal trimming, using dedicated components (e.g. DSP blocks) or using synthesis tools² from different vendors.

² Synthesis tool is a computer program that executes logic synthesis. Logic synthesis is process of converting a high-level design description into an optimized gate level representation.

In the first step of estimation process, the architectures of currently used FPGAs from two major manufacturers Xilinx and Altera were analyzed. It was assumed that the algorithm implementation should not require using high-end platforms that are used for the largest designs. This premise already significantly reduced the number of possible choices. To substantiate this assertion, Table 2 shows comparison of 6 FPGA models from Xilinx and Altera that cover the entire range that today's market can offer. Choosing an oversized platform results in unnecessary expenses, which can eventually make the implementation of design unfeasible.

Table 2 Comparison of FPGA models (source: <http://www.digikey.com/>, prices from 08/16/2016)

Vendor	Device family, model	Price in USD	# of LCs/LEs	RAM bits	# of IOs
Xilinx	Kintex Ultrascale KU115	5,635.00	1,451,100	77,721,600	338
Xilinx	Kintex 7 XC7K410T	1,301.25	406,720	29,306,880	400
Xilinx	Spartan 6 XC6SLX45	54.74	43,661	2,138,112	218
Altera	Stratix V 5SEEB	7,309.97	952,000	65,561,600	696
Altera	Arria V 5AGXB5	1,256.63	420,000	23,625,728	544
Altera	Cyclone V 5CEA4	49.40	49,000	3,464,192	224

Xilinx offers three and Altera two lower-end FPGA families, from which the final platform suitable for this application will be picked. As the official materials say, the selected families of FPGAs put the main emphasis on low price [19] [20].

In the next step, after the description of all the state machines in HDL language, the code, which was written independently of any platform, was synthesized to different Xilinx platforms in WebPack ISE. From the synthesis report, it was found that the synthesizer infers the same RTL components for all tested FPGAs (with the same architecture) even for those that could not fit the design. Initially, the calculation of quantity of LUTs required for implementation of RTL components inferred by the synthesizer was performed. However, the estimated number of used LUTs, which is also provided at the end of synthesis, was very far from the calculated number, due to various optimizations and signal trimmings. It equaled approximately one third of the obtained result. Similar outcomes were obtained for only parts of the whole design. The calculation could be performed for the Altera architecture in the same manner, but it seems that the estimation would become more of a guess.

Xilinx and Altera as two main rivals like to compare the performance of their products. They came up with a metrics that allows comparison of logic capacity of their FPGAs [21] [22]. This metric, however, mainly seems as a way to advertise their products. Not only there is a big dispersion of values for various designs, but the average value of 1.2 that expresses the logic capacity ratio of 6-input LUT/ALM claimed by the Xilinx is rather different from value 1.8 claimed by the Altera.

These considerations leave the impression that the best way to find appropriate size of FPGA from the perspective of LUTs is to synthesize the design, which needs to be built anyway, for particular architecture and use the result of estimated resources for decision. The same way can be used with advantage in case of register count determination.

Unlike the previous case, the memory requirements are already precisely determined during the design phase. After the memory bits are known, they need to be fitted to a particular platform.

The MSC compression algorithm is memory demanding. The algorithm needs to store diverse types of data throughout the process. For some data, this demand increases exponentially with symbol size in input data.

There are 3 structures that consume the majority of memory. In case of implemented 8-bit input symbols those structures take up approximately 6.5 Mb. There are FPGAs on the market that could fit this amount of data to internal memory. However, generally FPGAs with bigger memory are greater in all respects. So, it needs to be investigated, whether other resources would not stay unused, which is the case here. An alternative option is to use an external RAM memory.

Besides the large data structures, the rest of data can fit into approximately 25 blocks of M20K memories or 18 Kb BRAMs. The Xilinx BRAM comes out better from the comparison to Altera memories as they offer wider data bus and thus possibility to retrieve more data in one clock cycle.

The estimation of the component utilization by the synthesis tool is in Table 3.

Table 3 Resource estimation

NUMBER OF LUTS	~17 000
NUMBER OF REGISTERS	~8 500
NUMBER OF MEMORY BLOCKS	~25

The numbers consider the algorithm without the mentioned large data structures, FSM for external memory controller and UART receiver and transmitter. Also it is advised to utilize an FPGA not more than of 80% of its full capacity [18]. Based on the gathered information, a development board with the XC6SLX45 FPGA type from Spartan 6 family connected to external SDRAM memory was selected.

5.2 Implementation

The description of the hardware (FSMs) was performed in one of the two most used Hardware description Languages – VHSIC Hardware Description Language (VHDL), defined in standard IEEE 1076 [23]. The language is platform-independent, meaning it can be used for not only FPGAs but also other programmable logic devices (PLD) and Application-specific integrated circuits (ASIC).

Because Xilinx platform was chosen, the design was implemented in the development environment of the same company – Xilinx ISE Design Suite 14.7, more specifically the free version ISE® WebPACK™ [24]. WebPACK provides FPGA and CPLD logic design solution offering HDL synthesis and simulation, implementation, device fitting, and JTAG programming. Therefore, it delivers a complete, front-to-back design flow providing instant access to the ISE features and functionality. Screenshot of the environment can be seen on Figure 8.

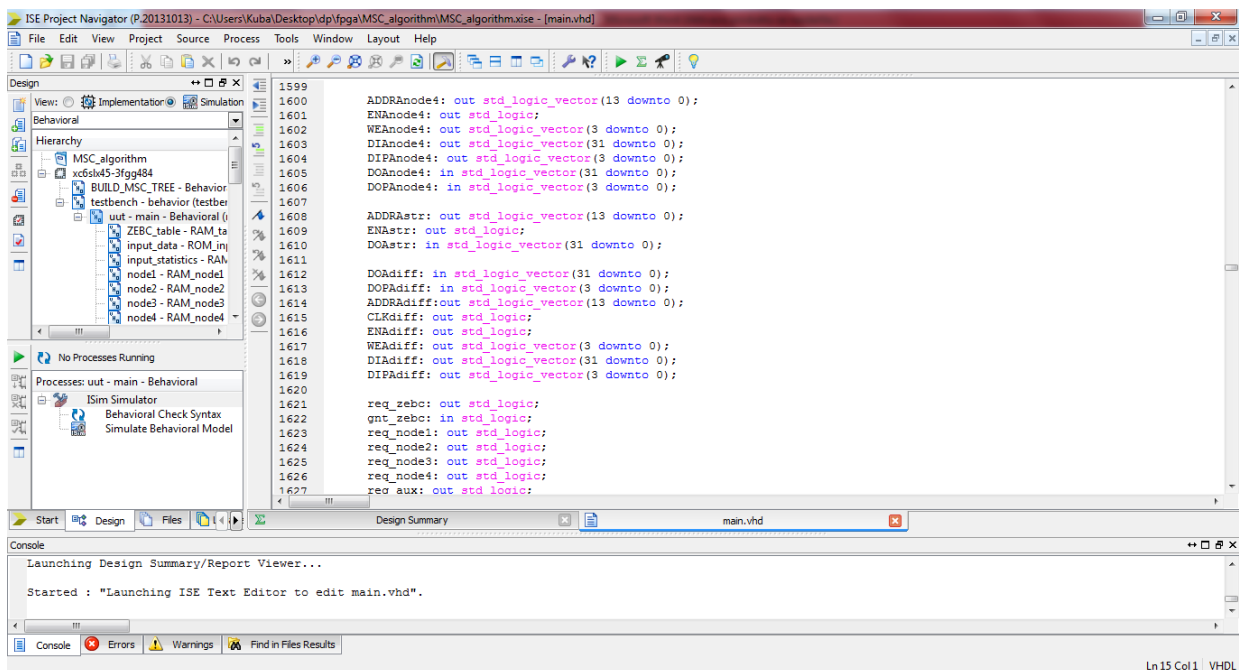


Figure 8 Xilinx ISE® WebPACK™ development environment

During the design, the ISim [25] simulation environment, included in the WebPACK, was used. The ISim is capable of performing behavioral and timing simulations for designs described in Hardware Description Languages – VHDL, Verilog, and mixed VHDL/Verilog language designs. Its user interface is shown on Figure 9.

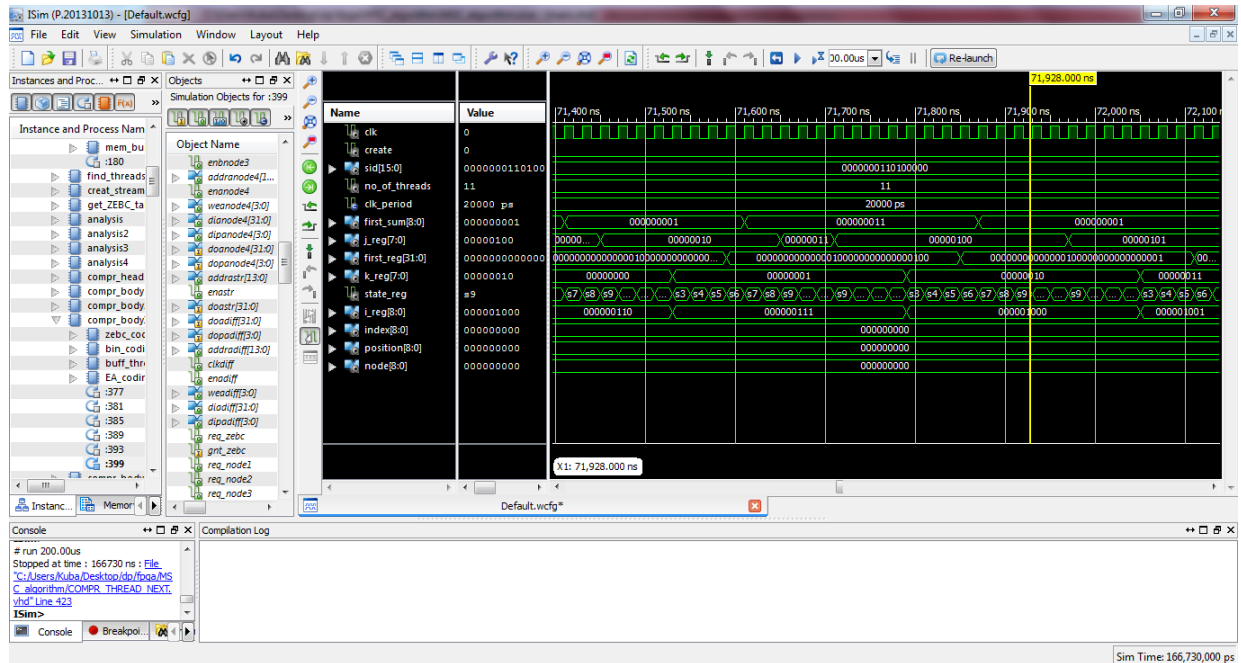


Figure 9 Xilinx ISim simulation environment

5.3 Memory structure

It is obvious from previous sections that the implementation of the MSC algorithm uses internal (BRAMs) as well as external (SDRAM) memories to chosen platform. The Figure 10 shows simplified structure of the MSC algorithm implementation, including memories. The memories are pictured as rectangles with rounded edges. FSMs are represented by rectangles with sharp edges and for easier orientation they are highlighted by grey background. For simplification, some of the memories in the scheme are grouped, thus scaling down the accuracy of the scheme.

As different FSMs share the memories, memory inputs need to be multiplexed, otherwise they would be driven by more sources. The scheme also shows direction of data flow between memory and FSMs (whether data are read, written or both). For illustration purposes, sacrificing the accuracy, both of these phenomena are drawn in the same scheme.

The parallel segments of the algorithm are also illustrated in the scheme. Some of the memories are parallelized as well. Besides, there are nonparallel memories that are accessed from parallel blocks. To prevent collisions caused by simultaneous access of memory, an arbiter needs to be present in the design, which decides which parallel blocks are prioritized in the access over others (section 5.4).

Due to the presence of external SDRAM in the architecture, a memory controller has to be part of the design serving as an interlink between the memory and the compression algorithm. This thesis tackles only the design of the interface between the controller and algorithm (section 5.3). The interaction with the SDRAM memory is elaborated by Ing. Tomáš Musil, Ph.D. who is one of the supervisors of the thesis.

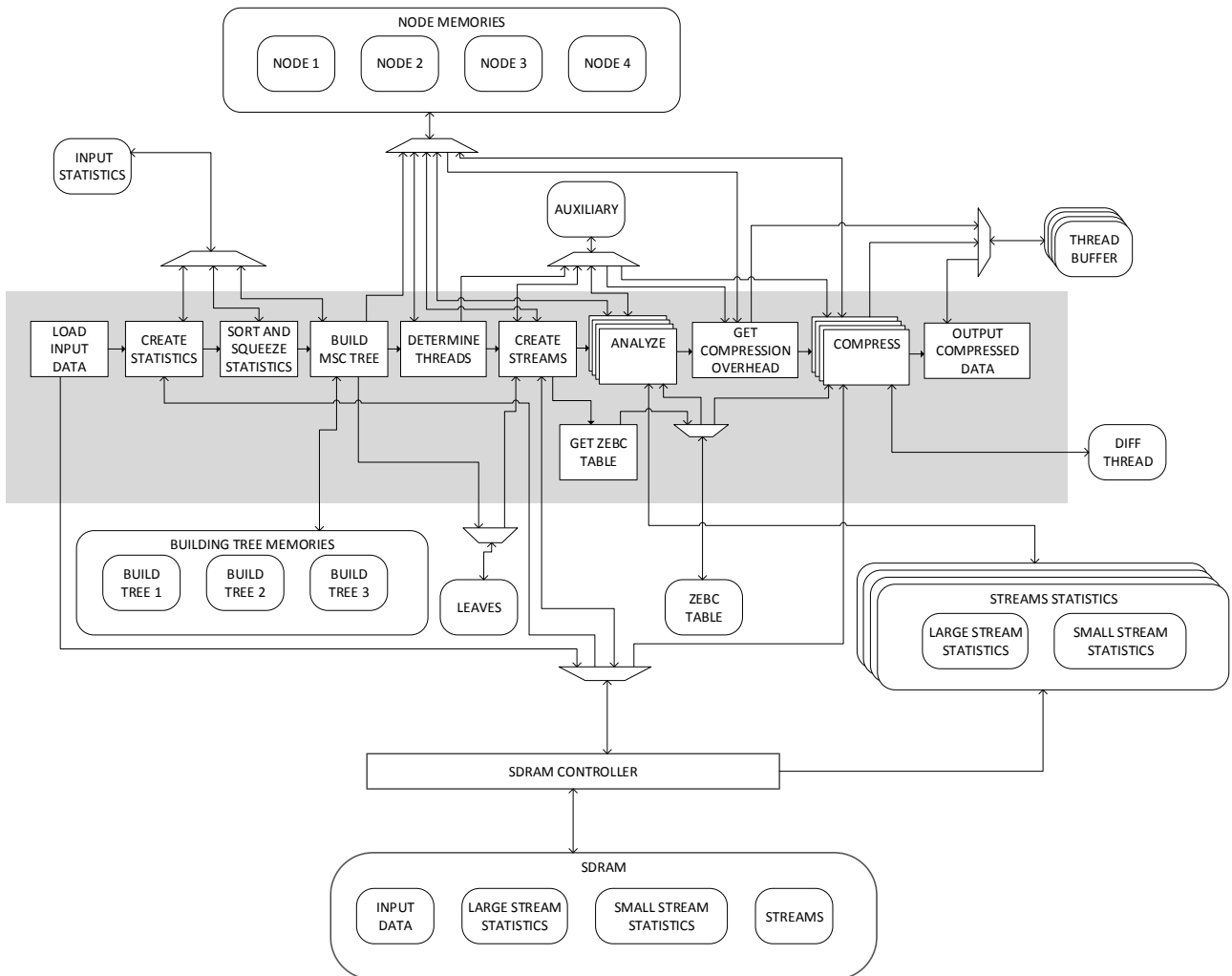


Figure 10 Architecture of the MSC design

At closer inspection of the scheme, it can be seen that the **STREAM STATISTICS** are in the scheme more than once – in **SDRAM** and in **BRAMs**. That is not a mistake, it is an intention, because in the **ANALYZE** stage of the algorithm, part of the entire statistics, which is stored in **SDRAM**, is needed. Therefore, it is copied to **BRAMs** when needed for easier manipulation with the data. For more information see sections 5.4 and 6.7.

5.4 SDRAM Interface

The **SDRAM** memory is high capacity memory that is needed for the capacity-demanding data structures in the **MSC** algorithm. Namely, those demanding structures are **INPUT DATA**, **STREAMS**, **SMALL STATISTICS** AND **LARGE STATISTICS**.

The interface between the algorithm Finite State Machines and the **SDRAM** controller contains 4 equivalent ports. Each port is formed by 3 buses. The first bus is used for control of the interface as well as for manipulation of a single item of data. The two remaining buses control one memory port of two **BRAMs** contained in one port. It is because both blocks that form the **SDRAM** interface (**MSC** algorithm and **SDRAM** controller) possess one memory port of **LARGE STATISTICS BRAM** and one memory port of **SMALL STATISTICS BRAM**, which is shown in scheme on Figure 11.

The complexity of the interface is given by the variety of data that are transferred via this interface and also due to parallel processing. Up to four blocks can run concurrently doing the same thing for different data – different nodes in the **MSC** tree, so each block owns one port and thus speed of the algorithm is not reduced by constant waiting for data retrieval due to queues on the port.

The high count of signals can also be taken as an advantage in times when there is only one sequential block running, during the stage when the counter **STREAMS** are created. In this stage, **INPUT DATA** are read and one value of counter is written into **STREAMS** and, depending on the value of the counter, to **SMALL STATISTICS** or **LARGE STATISTICS**. With this interface, each structure can have its own port for transfer.

Parallel processing is used for analysis and compression. During analysis, both **SMALL** and **LARGE STATISTICS** are needed. In this case, it is not only one value but the whole block for one particular node. For this purpose, both statistics are copied into dedicated **BRAMs**. The **LARGE STATISTICS** is modified in the process but need not be copied back to **SDRAM**. When the compression is being carried out, values from **STREAMS** are read one by one.

The external memory should not present a bottleneck when single values are read. In case of reading the whole STATISTICS block into BRAMs a delay has to be expected. This delay can be lowered by clocking the SDRAM and the ports of the BRAMs, controlled by the SDRAM controller, to the highest clock frequency allowed by the design.

5.4.1 Signals

The interface is designed with respect to the above mentioned parameters. The description of signals and data buses of one port follows.

- ❑ CMD launches the interaction with the SDRAM. As the Table 4 shows, if CMD = “00”, the memory waits for data transfer. If CMD = “01”, specified data are read and if CMD = “11”, specified data are written into SDRAM.

Table 4 CMD signal

CMD	bit representation
IDLE	00
READ	01
WRITE	11

- ❑ SIZE specifies the amount of transferred data as shown in Table 5. If SIZE = ‘0’, only one item is read from/written to an address specified by INDEX and POSITION. The size of an item depends on the TYPE as can be seen in Table 6. Else if SIZE = ‘1’, items on all POSITIONS from particular INDEX are transferred between SDRAM and particular BRAM. This option applies only for SMALL STATISTICS and LARGE STATISTICS.

Table 5 SIZE signal

SIZE	bit representation
ONE	0
ALL	1

- ❑ TYPE determines which data are manipulated. The list of used four types is in Table 6.
- ❑ INDEX specifies exact position for block of data and rough position for specific item.

Table 6 Attributes of used TYPES of data

TYPE	bit representation	INDEX	POSITION	data_size
INPUT DATA	00	0	0-65534	8
STREAM	01	0-3	0-65535	16
SMALL STATISTICS	10	0-510	0-360	16
LARGE STATISTICS	11	0-510	0-180	32

- ❑ POSITION specifies exact position of specific item
- ❑ DATA_IN bus specifies data to be written into SDRAM in case of writing one specific item.
- ❑ READY is output signal to announce that DATA_OUT is ready or that writing is finished in case SIZE = '0'. If SIZE = '1' then the BRAM that is paired with SDRAM is not accessed by the MSC algorithm side, until READY is equal to 1.
- ❑ DATA_OUT outputs the read data in case of reading one specific item.
- ❑ ADDR_x is address for accessing BRAM
- ❑ EN_x is enable signal of BRAM that permits to read/write data
- ❑ WE_x is write enable of BRAM that allows writing into memory. Its functionality is determined by active EN_x signal.
- ❑ DI_x is input data bus of BRAM
- ❑ DO_x is output data bus of BRAM
- ❑ CLK_MSC/CLK_SDRAM are two different clock domains used by the two interface sides

5.4.2 Pseudocode of SDRAM Controller

- lower index determines to/from which BRAM the data are copied. There are 2 paired BRAMs for each interface port: BRAM_{LARGE_STATISTICS} with 32 bit wide data bus and BRAM_{SMALL_STATISTICS} with 16 bit data bus.
- READY is in both READ and WRITE branch to determine if the SDRAM is occupied

```

if (CMD == IDLE)
    ⇒ go to beginning
elseif (CMD == READ)
    if (SIZE == ONE)
        ⇒ read DATA_OUT(TYPE, INDEX, POSITION)
        READY = 1

```

```

    ⇒ go to beginning
elseif (SIZE == ALL)
    ⇒ copy to BRAMTYPE_n(TYPE, INDEX)
    READY = 1
    ⇒ go to beginning
elseif (CMD == WRITE)
    if (SIZE == ONE)
        ⇒ write DATA_IN(TYPE, INDEX, POSITION)
        READY = 1
        ⇒ go to beginning
    elseif (SIZE == ALL)
        ⇒ copy from BRAMTYPE_n(TYPE, INDEX)
        READY = 1
        ⇒ go to beginning

```

5.4.3 Block Diagram

The block diagram of SDRAM interface captures only 1 of 4 ports. The n in the names of the data buses ranges from 0 to 3 and so the interface is 4 times larger than Figure 11 shows. The FSMs of the MSC algorithm decide when and how many ports are needed.

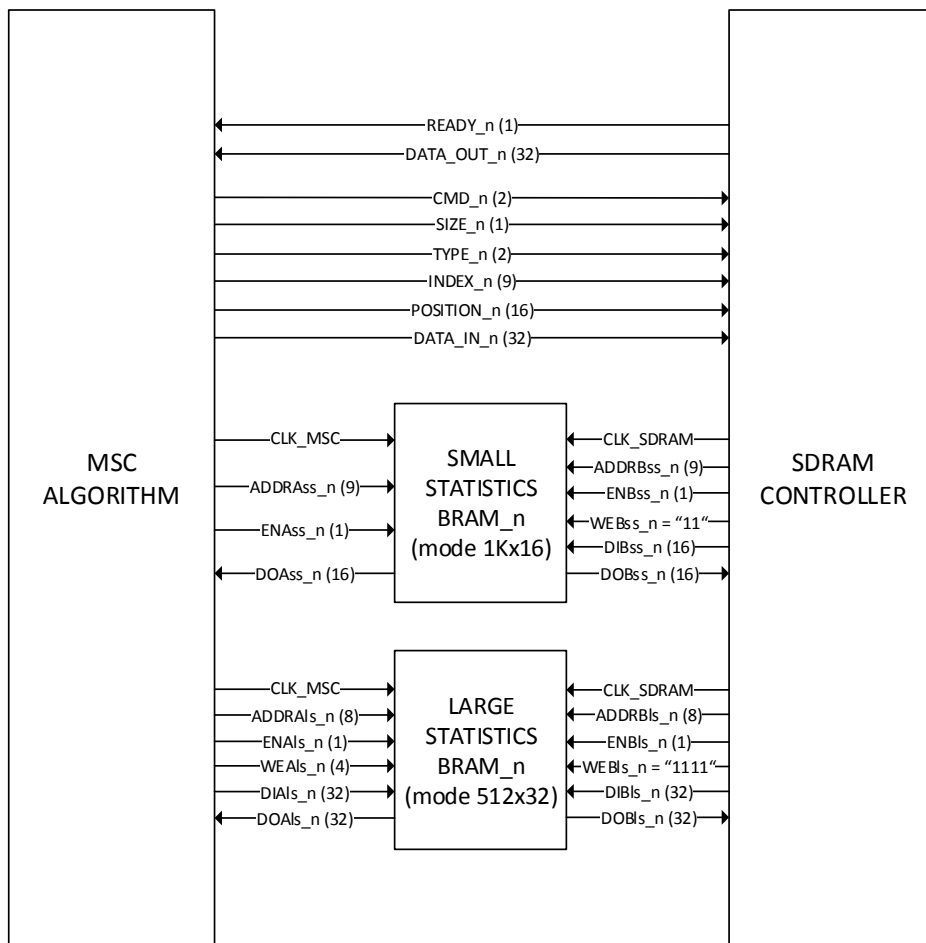


Figure 11 The SDRAM interface

5.5 Memory Arbiter

An arbiter is generally an element that governs the access to shared resources according to a set priority [26]. In case of the described implementation, memories are shared among different parallel blocks. All of the parallel blocks have equal priority, so the access is decided according to the index of parallel blocks. The memory requesting FSMs generally need to use the memory resources quite a low percentage of time so the priority of the blocks is set constant. Otherwise, a token would have to be passed among parallel blocks prioritizing the ones that did not use the memory for a longer time than the other ones.

In this setting, the memory arbiter is realized in hardware by a set of two input multiplexers. When a FSM requests a memory, it sets request signal *req* of particular memory to 1 and if no other parallel block with lower index requests the same memory, the appropriate grant signal *gnt* is set to 1.

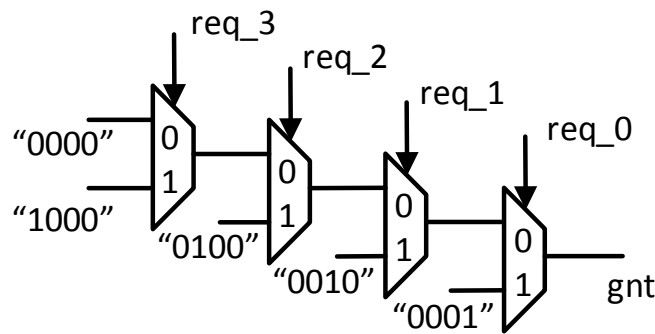


Figure 12 Memory arbiter

Sharing of the memory increases complexity of FSMs. New states are introduced causing slightly longer duration of FSM execution. Further delay might be introduced by the fact that blocks sometimes wait few clock cycles for the grant signal. However, this hold up is minor in comparison with the time saved due to parallel processing.

As the parallel blocks access memories in random times, the memory output needs to be stored to registers as soon as it appears. It ensures that the FSM can operate this data many clocks after they were read.

5.6 UART

The objective of FPGA board in this implementation is to perform the compression of input data. This data need to be loaded from computer (section 6.1). Similarly, the compressed data are transferred back to the computer after the compression is finished (section 6.10). For this purpose, the Universal Asynchronous Receiver/Transmitter (UART) is used as a transfer protocol between the computer and the FPGA.

The UART is a protocol that sends bytes of data through a serial line one by one. The transmission of a single byte is shown in Figure 13. The serial line is in the '1' state when no data are transferred. The transmission is started with a start bit that has value '0', followed by five to eight data bits and ended with a stop bit of value '1'. It is also possible to append an optional parity bit to the end of the data bits. The transmission is defined by a set of parameters in advance, which include the baud rate (i.e., number of bits per second), the number of data bits, and use of the parity bit. The bits for each transmitted byte are sent from least significant bit (LSB) to most significant bit (MSB) [27].

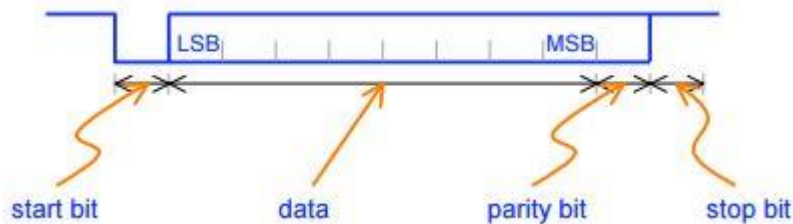


Figure 13 Structure of data sent over UART (source: <http://islwww.epfl.ch/>)

6 Design details

The steps performed in MSC algorithm were already described in chapter 2. In this section, detailed explanation of the algorithm implementation is provided.

Unlike the software implementation, the algorithm presented in this thesis makes selection only between two coding methods – Elias Alpha and ZEBC. Besides those two methods and Huffman coding, no other methods were considered. The Huffman coding was eliminated during design phase for two reasons. The first reason is that it brought quite a significant simplification of the algorithm and rise in speed of its execution. And the second has to do with the size of the compressed files. In this implementation the number of characters in input file is limited to 65 535. The Huffman coding introduces large header for each node coded by this method, meaning that the header itself could form considerable part of the final compressed stream. The large size is a significant handicap for coding of short input files, so the Huffman method may possibly not get used. The header contains all the information needed for reconstruction of Huffman tree, in contrast with no parameters in case of Elias Alpha coding and one single parameter – the base in ZEBC coding.

Besides, the Huffman coding is very demanding as far as the resources are concerned. Its implementation would occupy large amount of memory and it would require some change in the memory structure as well.

6.1 Input of Data to FPGA Board

The input data need to be loaded from computer to the SDRAM memory of the FPGA board. The data flow diagram of the SDRAM interface is shown on Figure 14.

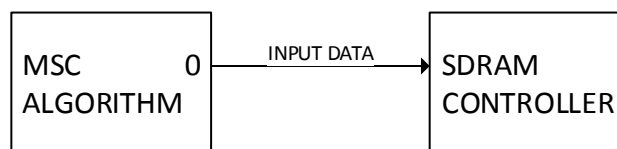


Figure 14 Dataflow diagram of SDRAM interface during input data writing

Besides those data, the compression algorithm also needs to receive compression parameters. The described implementation requires two parameters – a) number of used parallel blocks in range 1 to 4 and b) size of input data or in other words the number of input data characters.

The first parameter is sent as a first byte and two subsequent bytes in Big endian format are occupied by the second parameter. In order to lower memory requirements, the size of symbols in input data is limited to 8 bits, therefore one character is transferred in one byte. The transferred sequence is shown in Table 7. The N denotes number of input characters. The transfer follows the UART protocol described in section 5.6. The communication uses 8 data bits without parity and the transfer rate is fixed to 115200 Bd.

Table 7 Sequence of bytes sent from PC to FPGA

Byte	Type of data
0	Number of parallel blocks
1	Size N of input data (15 downto 8)
2	Size N of input data (7 downto 0)
3	Input data
:	
N + 2	Input data

6.2 Creation of Statistics

To obtain statistics, the entire input data need to be read one by one from SDRAM as shown in dataflow diagram on Figure 15. For each read *symbol*, if it has not occurred in the input data yet (in other words if *number of occurrences* obtained from memory is zero) it is written into the memory along with its attributes. If it is not first occurrence of *symbol*, *number of occurrences* is incremented and the new value is written into the memory. It is obvious, that each *symbol* has a specified place in memory, which is determined by the value of symbol.

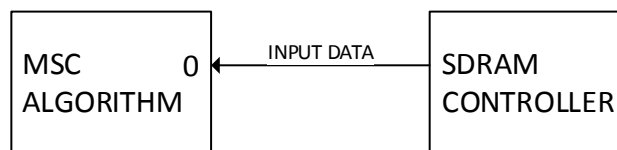


Figure 15 Dataflow on SDRAM interface during creation of statistics

The INPUT DATA STATISTICS is stored in one 18Kbit BRAM. The size of alphabet (maximum variety of symbols) is $2^8 = 256$, due to the restriction to only 8bit symbols. Each item of statistics has five attributes and is represented by 58 bits in total. Because the width of the structure is higher than the memory data bus of width 32 bits, bit representation of *symbol*, *sum of left subtree's nodes* and *index* (more on meaning of the values in paragraph 6.3) are stored on different address than *number of occurrences* and *first occurrence* of symbol.

The position in memory of the first group of attributes is $[2*symbol]$ and it is $[2*symbol + 1]$ for the other group. So in reality, one item takes up 64 bits of memory for their easy addressing in memory. The structure of the stored data can be seen in Table 8.

Table 8 Structure of INPUT DATA STATISTICS in BRAM

address/data	32	24	16	8
0	symbol (31 downto 24)	sum (23 downto 15)	index (14 downto 6)	
1	no_of_occ (31 downto 16)		first_occ (15 downto 0)	
:				
510	symbol (31 downto 24)	sum (23 downto 15)	index (14 downto 6)	
511	no_of_occ (31 downto 16)		first_occ (15 downto 0)	

Generally, not all symbols from alphabet are present in the input data, which means there will be unused addresses in the memory block. The statistics in this format is not convenient for next stages. So, when reading of the input file is finished, it is needed to prepare the STATISTICS for building the MSC tree. The items of STATISTICS are copied to the lowest addresses one after another so that they make one block, omitting the unused symbols. In this process, the STATISTICS array is traversed from the end to the beginning and if a *symbol* used in the input data is encountered, it is copied along with other attributes to the first unoccupied position (from the beginning) in the same array as shown in Tables 9. The number of used symbols (*no_of_symbols*), important for next processing, is obtained. At the end, the items are sorted in increasing order according to the *number of occurrences*.

Table 9 a), b) Squeezing statistics

index	0	1	2	3	4	5	6	7	8	9	10	11	12
symbol	x		z		f		a	b					c

index	0	1	2	3	4	5
symbol	x	c	z	b	f	a

Although, a variety of sophisticated sorting methods have been elaborated in hardware, one of the simplest algorithms – the modified Bubble sort [28] was converted into FSM. The Bubble sort was selected due to its simplicity and its low resource demand. This method compares each two subsequent items in an array and if they are in wrong order (depending on type of sorting), they are swapped. Each iteration ends when the last swapped item in previous iteration is reached. Sorting ends, when no items are swapped in one iteration.

6.3 Building the Tree

In this stage the STATISTICS becomes the array of unconnected nodes, which will be step by step overwritten by currently unconnected nodes. It is obvious that the initially unconnected nodes form leaves of MSC tree.

Before the process of tree building is described, it is necessary to make introduction into representation of binary tree in memory.

6.3.1 Representation of Binary Tree

When it comes to representing a tree in the standard programming language, it can be done so quite easily by using pointers or objects, telling the program, which nodes are connected to a particular node. In case of hardware implementation, however, when the designer works directly with memory, there is not such an option. Moreover, the MSC tree is not ordinary binary tree, but it has some special properties, which play a big role in choosing the right representation.

The important feature for selection of suitable tree representation is that the MSC algorithm requests traversing of the tree from root to a particular leaf, which is known in advance. The properties of the representation required by the algorithm are:

- ❑ each parent knows the position of left and right child
- ❑ each node is capable of determining the direction to a particular leaf
- ❑ the option to traverse only specified subtree of the MSC tree (for parallel processing)

There are two widely used models of binary tree representation in memory [29]. The two following sections will present some theory on how they work and why they are not suitable for MSC tree representation.

All of the tree representations will be demonstrated on the example of coding a word “abracadabra”. This tree can be seen on the Figure 16.

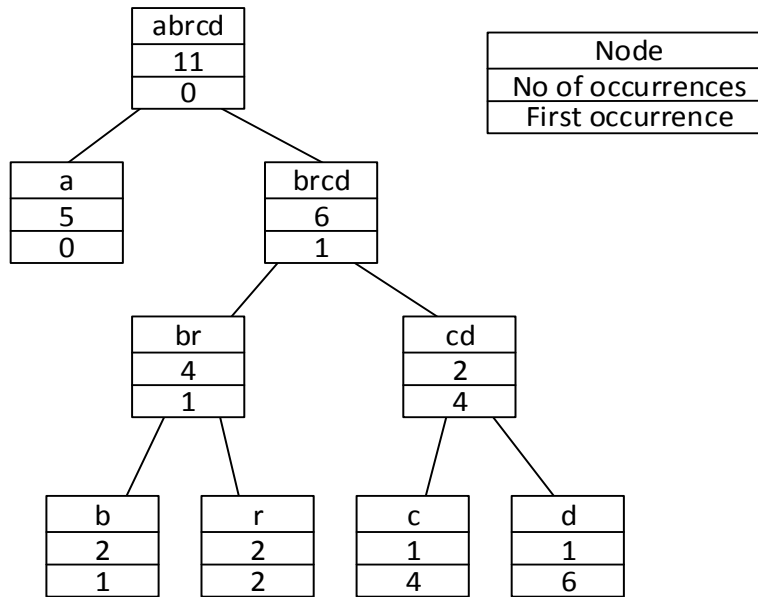


Figure 16 Example of MSC tree for "abracadabra"

6.3.1.1 Sequential Representation

In this representation, the position of all nodes in memory, which can be seen as an array, is fixed and is the same for all binary trees. The root is always stored at index 0, the left child of any node with index n is stored on position $2n + 1$ and the index of right child of the same node is calculated as $2n + 2$. At first glance, it seems like a perfect representation – no additional data required for finding both children. On closer inspection, it is apparent that in case of uneven tree (tree where the depth of the leaves varies considerably), a lot of memory is needed for the representation and also big percentage ends up unused. Another problem is that when the tree is to be traversed from the root to a particular leaf, additional information is required in order to determine the correct path.

For the example shown on Figure 16, fifteen cells are needed for storage of nine nodes, leaving 40% of memory wasted.

Table 10 Sequential representation of nodes in memory

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node	abr cd	a	br cd	-	-	br	cd	-	-	-	-	b	r	c	d

6.3.1.2 *Linked Representation*

In linked representation, all nodes can be stored in cells one after another, but it does not come for free - some additional information is needed. Each node must store the index of left and right child in the array, which is not so memory consuming considering each node in the MSC algorithm possesses plenty of other information. In case of 8-bit input symbols, the indexes take up 9 + 9 bits. The obstacle with this representation is that only with this information, the path from root to particular node cannot be found due to the random position of each node. Each leaf of the tree would require some string of bits that would hold the information about which way to go from every node on the path. As depths in MSC trees can become quite high, these strings can be rather long, taking up plenty of memory and possibly requiring multiple reads from the memory.

The Table 11 shows one of the possible layouts of tree nodes in memory. The path is expressed by directions from root to leaf which are represented by bits, where 0 means – go left and 1 means – go right. It can also be seen that there is a lot of unused memory space as leaves do not have children and the path is needed for leaves only.

Table 11 Linked representation of nodes in memory

index	0	1	2	3	4	5	6	7	8
node	d	c	r	b	a	cd	br	brcd	abrcd
left_ch	-	-	-	-	-	1	3	6	4
right_ch	-	-	-	-	-	0	2	5	7
path	111	110	101	100	0	-	-	-	-

6.3.1.3 *Left Tree Representation*

As the previous representations are barely usable for the MSC tree, a new representation was invented for this purpose – Left Tree Representation. This representation is a compromise between the two representations and it has some added value.

It does not waste memory but each item has the deterministic position in the memory, which however depends on structure of the tree. Furthermore, it allows the tree traversing from root to leaves without any added information.

Due to this representation, each node (if not leaf) knows where to find its descendants in memory and it can find a path from root to leaf due to smart organization of nodes, which is governed by following rules.

- ❑ left child of node is stored on a subsequent position:

$$left_ch_index = parent_index + 1$$

- ❑ right child is stored on a position that is computed in a following way:

$$right_ch_index = parent_index + no_of_nodes_in_left_tree$$

where *no_of_nodes_in_left_tree* stands for number of nodes in left tree of the active node. To explain further, each node X of a tree is a root of a subtree (subtree of leaf contains only one node – the leaf). This subtree can be more decomposed to left tree and right tree with X as a root, each including only one direct descendant of the node. For example, as Figure 17 shows, left tree of “brcd” node contains 4 nodes.

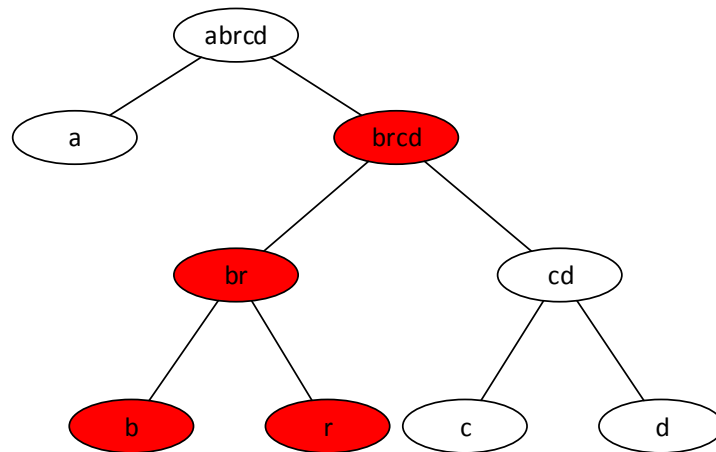


Figure 17 Left tree of "brcd" node

The tree in the example above would be represented as node’s array in following way:

Table 12 Left tree representation of nodes in memory

index	0	1	2	3	4	5	6	7	8
node	abrcd	a	brcd	br	b	r	cd	c	d
number of nodes in left tree	2	1	4	2	1	1	2	1	1

The final test for the left tree representation is to find the path from root to a particular leaf. The only thing that is necessary for this task is the index of the leaf in node’s array. Then, the task comes down to comparing the leaf’s index with active node’s right child index. If leaf’s index is smaller than node’s right child index, we proceed to left child, else proceed to right child. This is carried out until required index is equal to active node’s index.

Let's demonstrate this on the example. Say, we want to get to leaf "r" with index 5.

1. active node is "abrcd" (root) with index 0
2. index of right child is $(0+2) = 2$, which is less than 5 -> proceed to "brcd" (right child) with index 2
3. index of right child is $(2+4) = 6$, which is more than 5 -> proceed to "br" (left child) with index 3
4. index of right child is 5, which is equal to leaf's node index

What more, at the end of writing to streams (section 6.5), the counters of all nodes are written. This task needs to be carried out from top left to bottom right node in tree. The software implementation used recursion for this task, in this new hardware representation; the nodes are already arranged in exactly this order, so the node's array is only passed from the first to the last index.

6.3.2 Process of Tree Building

The construction of MSC tree is governed by two restrictive requirements, which causes that the process has to take place in two steps to achieve the desired representation.

As a first step, the linked representation (LR) of tree is created. The leaves are stored at lowest indexes from index 0 and newly created nodes are stored behind them one after another. The root is thus stored at last index equal to $(2 * no_of_symbols - 2)$. For the LR representation, three auxiliary BRAM memories are used storing the statistics (symbol, number of occurrences and first occurrence), number of nodes in left tree (needed for transformation to the desired representation) and data of linked representation of tree (direction from parent node and indexes of left child, right child and parent) for each node. The memories structure can be found in Appendix C.

In the second step, the linked representation is transformed into the left tree representation (LTR) according to following algorithm:

```

end = 0
index = 0
⇒ read ROOT from LR(2*no_of_symbols - 2)
⇒ write ROOT to LTR(index)
CURRENT_NODE = ROOT
index = index + 1
while (end == 0)
    ⇒ read CURRENT_NODE.left_child
    ⇒ read CURRENT_NODE.right_child
    ⇒ write CURRENT_NODE.left_child to LTR(index)
    index = index + 1
    if CURRENT_NODE.left_child.type != LEAF

```

```

CURRENT_NODE = CURRENT_NODE.left_child
parent = CURRENT_NODE
else
  while(1)
    ⇨ write CURRENT_NODE.right_child to LTR(index)
    index = index + 1
    if CURRENT_NODE.right_child.type != LEAF
      CURRENT_NODE = CURRENT_NODE.right_child
      ⇨ break
    else
      ⇨ read parent
      CURRENT_NODE = parent
      do
        desc_dir = CURRENT_NODE.dir
        parent = CURRENT_NODE.parent
        CURRENT_NODE = parent
      while(desc_dir == 1)
        if index == 2*no_of_symbols - 2
          ⇨ break
        end = 1

```

The attribute denoting type of node (*type*) can take following values: root, middle and leaf. The attribute *desc* stands for descendant and denotes the direction from parent node to the CURRENT_NODE.

The algorithm traverses the nodes in the same order – from left top to right bottom, they are ordered in LTR and stores the attributes of nodes, concerning the structure of the tree and the input statistics into the NODE memories. Besides that, the indexes of leaves in NODE memories are written to LEAVES memory. Traversing of the tree is shown on Figure 18.

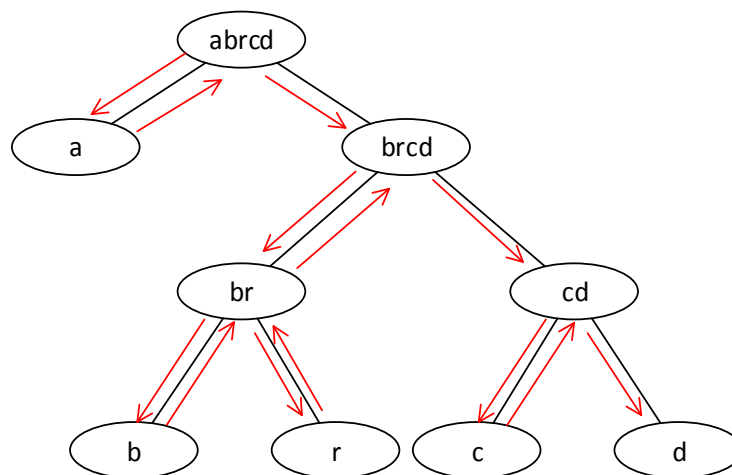


Figure 18 Traversing of tree during transformation from LR to LTR

6.3.3 Memory Structure Used for Nodes

After building the node structure of the Left Tree Representation in memory, it is kept there until the very end of the algorithm as it is needed in almost all following stages. Each node is an individual entity that needs some attributes in each stage of the algorithm, possessing quite a large number of them in total. Some of these attributes are even large arrays of values, which are stored outside the FPGA due to high memory demand. The single value attributes of one node take up 138 bits in total. They were mapped into memories with 36-bit data bus width. As it is much easier not to mix attributes of different nodes on same address, the aim was to fit each node into n addresses, occupying n*36 bits. All the attributes are stored on four addresses and take up 144 bits, leaving 6 bits unused for one node. If Huffman coding was considered, there would be two more attributes – interval counter and max depth of Huffman tree.

As the maximum number of nodes in a tree is 511 ($2^{\text{alphabet_size}} - 1$), it is obvious that the single value attributes will need $511 \times 144 = 73\,584$ bits, which is nearly the maximum capacity of 4 block RAMs. As the attributes of one node are stored in 4 addresses, it is advantageous to store them to separate memories. Due to this separation, all of the attributes are advantageously stored on the same address, only in different memory. Another advantage is that there is a possibility to retrieve all the attributes of a node in one clock cycle.

The structure of the NODE memories can be seen in Tables 13, 14, 15 and 16. The purpose of some attributes will be explained in further sections. The range in brackets denotes, which bits of data bus is occupied by the particular value. The tendency was to group attributes that are used in same stages together.

Table 13 First block of memory containing node attributes (NODE1)

0	no_of_occ (35 - 20)	first_occ (19 - 4)	par_thr (3 - 2)	
:				
510	no_of_occ (35 - 20)	first_occ (19 - 4)	par_thr (3 - 2)	

Table 14 Second block of memory containing node attributes (NODE2)

0	counter (35 - 20)	symbol (19 - 12)	type (11-10)	left_tree (9 - 1)	dir (0)
:					
510	counter (35 - 20)	symbol (19 - 12)	type (11-10)	left_tree (9 - 1)	dir (0)

Table 15 Third block of memory containing node attributes (NODE3)

0	point_str (35 - 20)	point_par_thr (19 - 4)	thr_ind (3-2)	flg1 (1)	
:					
510	point_str (35 - 20)	point_par_thr (19 - 4)	thr_ind (3-2)	flg1 (1)	

Table 16 Fourth block of memory containing node attributes (NODE4)

0	ls_items (35 - 28)	base (27 - 22)	flg2 (21)		sum_counters (19 - 4)	best_m (3 - 2)	
:							
510	ls_items (35 - 28)	base (27 - 22)	flg2 (21)		sum_counters (19 - 4)	best_m (3 - 2)	

6.4 Determination of Subtrees

As some steps of the algorithm are carried out in parallel, each node needs to be assigned to a particular block. To get those blocks, a tree needs to be divided into subtrees or layers. In this implementation, the first option was chosen. It is the easier version because the interface between 2 different blocks is made up by only one node. The number of blocks that will process the data in parallel depends on the user, as it is one of the input parameters to the compression module, but the maximum number was set to 4. However, if number of nodes in a tree is lower than selected number of parallel blocks, then each node is assigned to its own block.

The subtrees are selected in following way. Root of the tree is root of thread³ 0. If the data is to be processed by more than one thread, the child of the root with higher number of occurrences is chosen as root of thread 1.

³ The software implementation used threads to implement parallelism in the algorithm. The word thread became almost a synonym of one instance of parallel processing in MSC algorithm, so the thesis uses interchangeably names “thread” and “parallel block”.

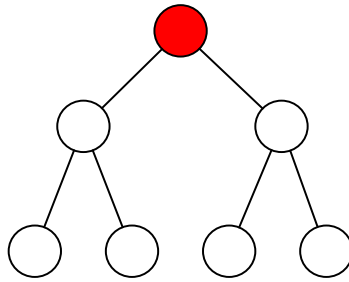


Figure 19 Chosen thread root if no_of_threads ≥ 1

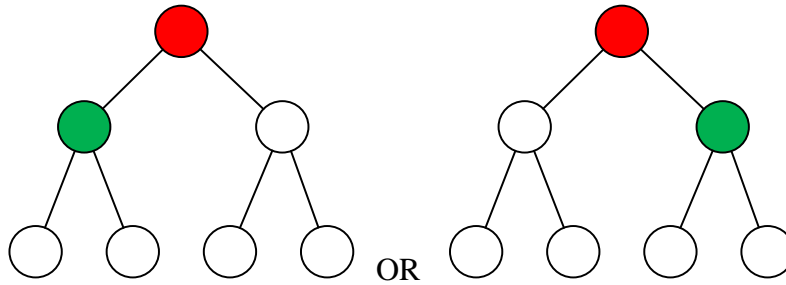


Figure 20 Alternative choices for thread root 1

If number of threads is higher than 2, the child with higher number of occurrences of the tree root's child with lower number of occurrences is chosen, but only if this root's child is not a leaf. Otherwise, the tree root's child with lower number of occurrences is labeled as root of thread 2. In case of 4 threads, the root of thread 3 is either the child with higher number of occurrences of root of thread 1 or pair node of root of thread 2 (the other child of parent of thread 2) if root of thread 1 is a leaf. The selection of the threads is shown on figures below. To save space, they show only cases when roots of thread 1 and 2 are right children. The same applies for left child.

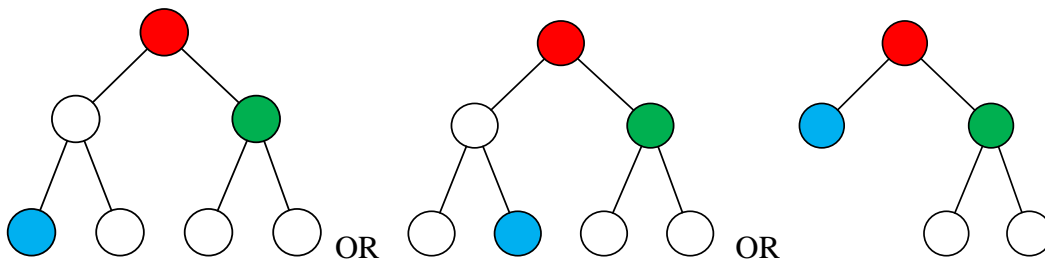


Figure 21 Alternative choices for thread root 2

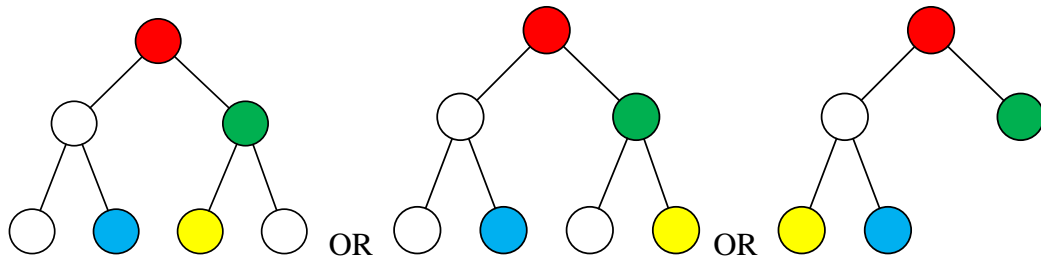


Figure 22 Alternative choices for thread root 3

Three attributes of nodes – parent thread index (*par_thr*), thread index (*thr_i*) and flag of thread root (*flg1*) are stored to NODE memories for thread root nodes. Besides, index of each root in NODE memories and type of thread is stored to AUXILIARY memory for each thread. There are four possible types of thread as shown in Table 17. The representing values must be non-zero, because they are written into the final compressed stream by Elias Alpha coding. Also, the values are selected with respect to occurrence of each type in compressed data - the most used threads are coded by the lowest value.

Table 17 Types of threads

TYPE	REPRESENTING VALUE
MAIN THREAD WITHOUT CHILDREN	4
MAIN THREAD WITH CHILDREN	1
CHILD THREAD WITHOUT CHILDREN	2
CHILD THREAD WITH CHILDREN	3

NOTE:

Selection of the subtrees influences the processing time of stages of the algorithm that run in parallel. The execution time of the ANALYZE phase depends mainly on total number of counters of all nodes in particular block. Duration of the COMPRESS phase then depends considerably on number of occurrences of root node of subtree.

We choose to select threads from the top of the tree as the nodes have higher number of occurrences and thus higher number of counters in a stream that will be processed. However, the root of the tree has only one counter in stream, so we generally do not want it to have both children in different threads, because in the ANALYZE stage of the algorithm, there would be no nodes to analyze. For the sake of simplicity of this FSM, there is one case in which this occurs as shown on Figure 21.

If all leaves of MSC tree are in depth higher than two, then all subtrees consist of more than one node. Example of possible subtree selection is shown on Figure 23. It is apparent that in this particular case every subtree contains between 7 and 9 nodes and those with less nodes tend to have higher number of occurrences, thus maximizing the effort to make the processing time uniform.

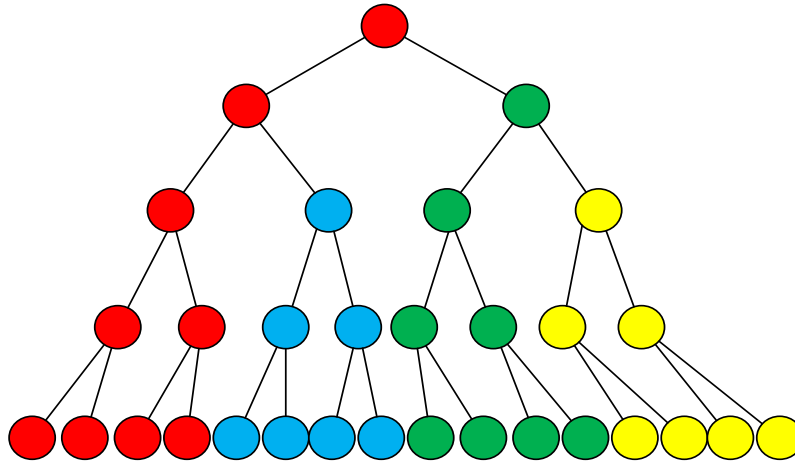


Figure 23 Selection of subtrees for a random tree

6.5 Creation of Streams

To create streams, the INPUT DATA need to be read for second time and even though the blocks for parallel processing have already been determined, this stage still runs sequentially, but the information about parallel block indexes is used.

On the beginning, all the counters *counter* are reset and all switches *dir* in the MSC tree are set to 0 – pointing to left side. For each symbol, the tree is traversed from root to leaf, which is based on calculating child's index of each node until the desired index is reached as specified in section 6.3.1.3. To find an index of particular leaf in the NODE memories, an auxiliary memory structure LEAVES is used, which contains indexes that leaves occupy in the NODE memories. All of the leaves are contained and they are stored at an address that has the value of the leaf's *symbol*.

During traversing of the tree, the position where the *counter* of node will be written into thread STREAMS is specified as soon as the *counter* starts incrementing. The process is described in section 6.5.1.

If the investigated node is not a leaf of tree, then we proceed to a child lying on the path from root to the particular leaf. If switch from the investigated node is not pointing to child on the path, counter of the node lying on the switch (pair node) is written into the thread STREAM (to specified position by *str_pos* and *str_par_pos*) and the counter STATISTICS. The *counter* is written to both memories, because they have different structure. The thread STREAM is used for compression, whereas the counter STATISTICS is intended for analysis. The *counter* of the pair node is then reset and the switch direction is changed to follow the path. Each time a node is entered during traversing, its *counter* is read, incremented by one and written back to the memory.

When the end of input data is reached, there are still nonzero values in node counters, which need to be written to STREAMS as well. As explained earlier (section 6.3.1.3), the array of nodes are accessed in order specified by LTR. The interaction with SDRAM controller during the phase of creating streams is shown on Figure 24.

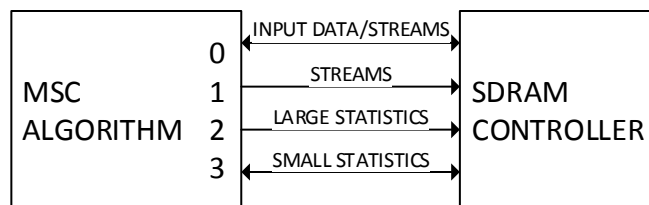


Figure 24 Dataflow of SDRAM interface during creation of streams

6.5.1 Specification of Counter Position in Thread Streams

Each thread has its own thread STREAM. Thread STREAM is an array of numbers that contains all of the *counters* of all of the nodes in one thread but also *counters* of root nodes of child threads if the thread has any. The order in which *counters* are stored is specified and is crucial for the compression phase. A large number of *counters* might need to be stored in each STREAM, thus they are stored into external memory.

The writing into the thread streams has following rules. If counter of investigated node is 0, then the node remembers the first free position in the respective thread stream. The counter value of node will later be written to this position. The position is stored in node attribute *str_pos*. Moreover, if the current node is a root of child thread, the node stores the first free position in parent thread stream into *str_par_pos*. To determine the index of parent thread, the NODE1 memory contains *par_thr*, which stores this value for each thread root node. At the end, the first free position in the current thread stream is incremented. In case of thread root, the same thing happens for parent thread.

To illustrate the process of storing stream positions, an example is presented on tree shown on Figure 25. The tree is divided into 4 threads. For illustration purposes let's say, symbol C in read from input data and the thread streams are empty.

- ❑ Initially, the node 0 (root) is investigated
 - value 0 is stored into *str_pos* as it is first free position in red thread stream
- ❑ The node 1 is investigated
 - value 0 is stored into *str_pos* as it is first free position in green thread stream
 - value 1 is stored into *str_par_pos* as it is first free position in red thread stream (red thread is parent thread of green thread)
- ❑ The node 3 is investigated
 - value 0 is stored into *str_pos* as it is first free position in blue thread stream
 - value 1 is stored into *str_par_pos* as it is first free position in green thread stream (green thread is parent thread of blue thread)
- ❑ The node C is investigated
 - value 1 is stored into *str_pos* as it is first free position in blue thread stream

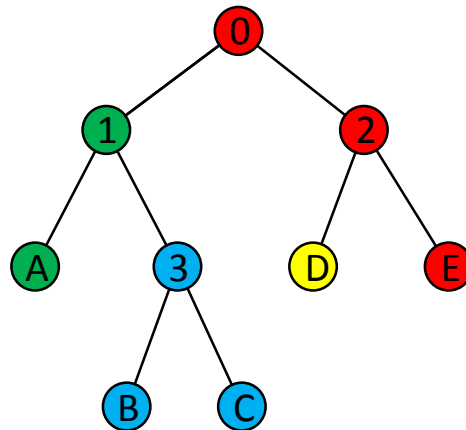


Figure 25 Example of MSC tree

6.5.2 Counter Statistics

The counter STATISTICS is another attribute of node. It also needs to be stored outside the FPGA due to high memory requirements. It is divided into two parts by delimiter. Delimiter is a defined number that separates values of counters to statistics of low value counters SMALL STATISTICS and statistics of large value counters LARGE STATISTICS. The value of the delimiter is computed with respect to minimization of memory demand for statistics storage. The delimiter is computed in a following way, which applies for the worst case scenario.

$$\min \left((\text{delimiter} - 1) * 16 + \left\lfloor \frac{\text{size of input data}}{\text{delimiter} + 1} \right\rfloor * 32 \right)$$

For this particular configuration, the delimiter is equal to 361. To understand the formula above, the structure of both of the statistics have to be understood first. One item in the SMALL STATISTICS occupies 16 bits and stores only the number of occurrences of counter with particular value as the counter value is specified by the position in the array.

For large statistics this approach would not be very convenient. The occurrence of high values of counters is very sparse, which would leave us with a lot of unused memory. Instead, the value of counter as well as its number of occurrences is stored taking up 32 bits of memory and the items of large statistics are stored one after another. The number of occurrences is, however, determined in next phase, at this stage the values higher or equal to delimiter are just stored on first free position in memory.

For computation of memory needed for large statistics, the worst case scenario needs to be determined. It is the scenario when the investigated node is accessed delimiter number of times (in our case 361) without resetting the counter and then the pair node is accessed once. It means that after each $\text{delimiter} + 1$ number of times the tree is traversed, the counter is written into statistics. To get the maximum number of the items in LARGE STATISTICS, $\text{delimiter} + 1$ must divide the maximum number of items in the input data. For this implementation:

$$\left\lfloor \frac{65535}{361 + 1} \right\rfloor = 181$$

The statistics for one node will therefore take up

$$(360 * 16 + 181 * 32) = 5760 + 5792 = 11\,552 \text{ bits.}$$

As the statistics needs to be stored for each node separately, it consumes big amount of memory.

Example:

Counters 128, 131, 570, 128, 391, 128, 417, 570, 391 of one node shall be stored to node's statistics.

1) The initial state is:

SMALL STATISTICS

position	0	127	128	129	130	131	132	360
number of occurrences	0	0	0	0	0	0	0	0

LARGE STATISTICS

position	0	1	2	3	4	5	180
counter	0	0	0	0	0	0	0
number of occurrences	0	0	0	0	0	0		0

2) After storing the counters

SMALL STATISTICS

position	0	127	128	129	130	131	132	360
number of occurrences	0	0	3	0	0	1	0	0

LARGE STATISTICS

position	0	1	2	3	4	5	180
counter	570	391	417	570	391	0	0
number of occurrences	1	1	1	1	1	0		0

NOTE:

In the described implementation, the value of delimiter was selected with objective to minimize the memory demand of the streams STATISTICS. If memory is not a concern and the goal is to maximize speed of the algorithm, the delimiter value should be determined differently. It could be determined empirically by observation for various input data. The lower counters are generally more tightly packed. The values of counters from the lowest values to some set threshold after which the values become dispersed shall be stored to SMALL STATISTICS and the values above the threshold shall be stored to LARGE STATISTICS.

6.6 Calculation of ZEBC Table

The ZEBC table is calculated for indexes 0 to 15 according to the rules described in section 2.2.2. The table is stored in one BRAM. Its structure can be seen in Table 18.

Table 18 ZEBC TABLE memory structure

DATA		PARITY			
31 downto 16	15 downto 0	3	2	1	0
beginning	end				
x 16					

6.7 Parallel Analysis

The purpose of the analysis is to determine a method for each node, by which the counter values will be coded. Besides, the parameters of the selected method are defined.

The analysis starts with reading the index of thread root and type of thread from AUXILIARY memory. The root of the MSC tree (node with index 0) is not analyzed, because its statistics always contains only one value and it is already written in the header of the compressed string, so there is no need to analyze it. Other thread roots are processed equally to ordinary nodes.

The analysis is conducted for each node separately. In the analysis, the stream STATISTICS as a whole is processed. To decrease the number of interactions with the external memory, the entire STATISTICS of the processed node is copied into the BRAMs before the node is analyzed. Manipulation of STATISTICS during this phase is shown on dataflow diagram on Figure 26.

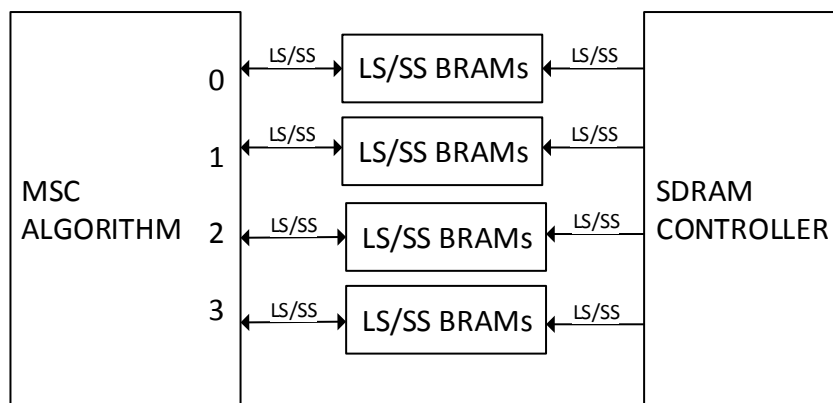


Figure 26 STATISTICS manipulation during analysis phase

The analysis of each node is performed in four steps

1. Statistics processing
2. Elias Alpha coding analysis
3. ZEBC coding analysis
4. Advance to next node

In reality, the component ANALYZE contains three subcomponents as shown on Figure 7b). The first step is divided into two components running in parallel that separately process SMALL and LARGE STATISTICS. The last component is realization of the third step. Steps 2 and 4 are not in subcomponents as they present just few lines of code.

6.7.1 Statistics processing

The processing time of SMALL STATISTICS depends on the value of delimiter. Logic demand for realization the FSM processing SMALL STATISTICS is quite simple and so one iteration is quite short. The number of iterations of LARGE STATISTICS processing depends on its number of items, which is generally not very high if the delimiter is chosen reasonably, but average processing time of one item is much higher with more associated logic. Also, no resources are shared for processing of the both STATISTICS. This resulted in decision to put processing of SMALL STATISTICS and LARGE STATISTICS in parallel.

The SMALL STATISTICS is not changed during the processing. The whole array is only read and two values are determined – total number of counters and maximum counter.

The LARGE STATISTICS is processed in two steps. First, it is sorted in increasing order according to the counter values. The Modified bubble sort is used in the same form it was used earlier. In second step, the LARGE STATISTICS is squeezed in order to merge same counter values to one with appropriate number of occurrences. Thus, the LARGE STATISTICS is changed before the coding analysis is conducted. As the statistics is no longer needed in subsequent stages of the algorithm, it does not have to be stored back into the SDRAM. Similarly to SMALL STATISTICS, the number of counters of LARGE STATISTICS is determined. The number of counters of both STATISTICS are added together in the higher module and stored to NODE4 memory along with other attributes.

Example (continued from section 6.5.2):

LARGE STATISTICS

position	0	1	2	3	4	5	180
counter	391	417	570	0	0	0	0
number of occurrences	2	1	2	0	0	0		0

6.7.2 Coding analysis

During the coding analysis stage, the length of counter values coded by all chosen coding methods is calculated for one node at a time. At the end, the method giving the best result is chosen and additional values are added to this length. The calculation of length is very simple in case of Elias Alpha coding. The number of bits needed for coding all counter values of a node is equal to number of occurrences of the same node.

The analysis of ZEBC coding is more difficult. Computation of length interacts with the ZEBC table of intervals (Appendix A). The FSM calculates the lengths of coded data for different ZEBC bases and the aim is to find the minimum length value. The algorithm can be well understood from the Table 19.

The first column in the table contains the value to be coded. The second column contains indices of ZEBC table intervals to which values belong. From third column on, the values in cells contain the length of coded value by ZEBC(b), where for illustration base $b = 1..5$. The numbers in the brackets are differences of coded values' lengths comparing to coding with $(b - 1)$ considering same counter value (basically it tells us how much the length differs from the length in previous column).

Table 19 Examples of values coded by ZEBC with different bases

Value	Index in ZEBC table	ZEBC(1)	ZEBC(2)	ZEBC(3)	ZEBC(4)	ZEBC(5)
1	0	2	1 (-1)	1	1	1
2	1	2	3 (+1)	2 (-1)	2	2
3	1	4	3 (-1)	4 (+1)	3 (-1)	3
4	1	4	5 (+1)	4 (-1)	5 (+1)	4 (-1)
5	1	4	5 (+1)	6 (+1)	5 (-1)	6 (+1)
6	2	4	5 (+1)	6 (+1)	7 (+1)	6 (-1)
7	2	6	5 (-1)	6 (+1)	7 (+1)	8 (+1)

First, the benchmark is set by computing the length of coded data by ZEB(1). On a closer look of the table above, a pattern can be seen. In ZEB(1) column, we see that the minimum length is 2 and the length increases by two every time a second value in ZEB table interval (underlined) is reached. The total length of one node's counter values is calculated by formula:

$$total\ length = \sum_{i=0}^{mc} nocc(i) * len(i)$$

where:

mc – maximum counter of analyzed node of MSC tree

nocc – number of occurrences of counter with value *i*

len – length of counter with value *i* coded by ZEB(1)

For bases higher than 1, we can see in the table above that when the value of counter ($b + beginning_of_interval - 1$) is reached the length of coded value by ZEB(*b*) is lower by one and the length of all the other values is higher by one comparing to ZEB(*b*-1). In this manner, the length is calculated for counters in SMALL STATISTICS.

In case of dispersed LARGE STATISTICS, each investigated counter value is firstly decremented by the current ZEB base *b* and then the *index* of interval where this value belongs is found. The length is then determined from the formula:

$$(b + index + index + 1)$$

At the end of the analysis for each base, the obtained total lengths of both STATISTICS are added to get the final length, which is compared to so far lowest length (calculated for lower bases) and if the lowest length is higher than the current length, the current length becomes the lowest and the current base is stored as a best base.

When the length for all investigated coding methods is obtained, some values from node's header, that is included in the compressed data, are added to this length. The header will be described later, but it contains two items important for now – best method and coding parameters. Because length of those items differs for different methods, it needs to be included into the decision of best method. Each method has a value assigned to it, which is coded by Elias alpha in the final stream, thus its length is equal to its value. In this implementation, Elias alpha has value 1 and ZEB has value 2 assigned. As far as the compression parameters are concerned, Elias Alpha does not have any and ZEB has one – the base (giving the lowest length) coded by Elias Alpha.

6.7.3 Advance to next node

During analysis, each node of subtree is entered and analyzed. This stage determines the sequence, in which the nodes are analyzed. Also, the total length of coded data is calculated for the parallel block.

The subtree of one parallel block is traversed from left top to right bottom. To obtain attributes of a particular node, which are stored in LTR, it means that index of the node is just incremented by one in order to proceed into the next node. Sometimes, it happens that the node on the next index is from a different parallel block. If this node is left child of its parent, then right child is entered instead. If the node is right child, the analysis is terminated (right child is at the subsequent index in LTR if the node at the current index is leaf node).

The total length of coded data is incremented after analysis of each node by the obtained minimum length plus the remaining items of the header (will be specified later) that have not yet been added. When a node from different thread is found during traversing of the subtree, the length of different parallel block identifier (described in section 6.9) is added to the total length.

6.8 Buffer scheme

Before talking about the compression phase, let us mention the storage of compressed data into an intermediate buffer. The buffer is necessary as the output data cannot be transferred to PC directly. The binary values that appear on the output from the algorithm can be either values coded by inverted Elias Alpha coding that have defined structure and variable length, which can be in some cases quite high. Or it can be arbitrary binary coded value of length between 1 and 32 bits. Some of the values in the MSC overhead are represented by 32 bits otherwise the length does not exceed 16 bits.

As the coded values appear on the output in random time and with random lengths, an intermediate element that would buffer those values in correct format and order is needed. A combination of 8-bit register and BRAM were selected for this task.

The register stacks and splits the variable length values so that they form 8-bit blocks. When the register is full, its content is copied into first unoccupied position in BRAM, thus buffering the values. The parity bit is also used; it is set to '0' for all data. After the last byte of compressed data is stored, one special byte with any value and parity bit set to '1' is stored to buffer, which is used when sending data to computer to recognize last byte of data.

Generally, the data are compressed in parallel, so each parallel block has its own buffer and the last byte in the each buffer is marked by the parity bit. Also the overhead has its own buffer, but due to its known length, there is no need to denote the last byte.

The pseudocodes for producing 8-bit values from Elias Alpha coded and binary values are below.

WRITE ELIAS ALPHA(FREE, LEN)

```

end = 0
last = 0
while (end == 0)
  if (LEN > FREE)
    data(FREE-1 downto 0) = (others => '1');
    last = 1
  elseif (LEN == FREE)
    data(0) = '0';
    data(FREE-1 downto 1) = (others => '1');
    end = 1
    last = 1
  else
    data(FREE-LEN) = '0';
    data(FREE-1 downto FREE-LEN+1) = (others => '1');
    FREE = FREE - LEN
    end = 1
  if (last == 1)
    last = 0
    FREE = 8
    LEN = LEN - FREE
    ⇨ write data to BUFFER
    data(7 downto 0) = (others => '0');

```

WRITE BINARY(FREE, LEN, VALUE)

```

end = 0
last = 0
while (end == 0)
  if (LEN > FREE)
    i = 8 - FREE
    while (i < 8)
      data(7-i) = VALUE(LEN-i-1+8-FREE);
      i = i + 1
    LEN = LEN - FREE
    last = 1
  elseif (LEN == FREE)
    i = 8 - FREE
    while (i < 8)
      data(7-i) = VALUE(LEN-i-1+8-FREE);
      i = i + 1
    last = 1
    end = 1
  else
    i = 8 - FREE
    while (i < 8-FREE+LEN)
      data(7-i) = VALUE(LEN-i-1+8-FREE);

```

```

end = 1
    i = i + 1
if (last == 1)
    last = 0
    FREE = 8
    ⇨ write data to BUFFER
    data(7 downto 0)=(others => '0');

```

6.9 Compression

The compression starts with specification of the overhead. It contains information necessary for decoding. The length of the overhead differs with number of used parallel blocks. It contains general parameters regardless of the number of parallel blocks and specification of each parallel block. Table 20 shows the sequence of values in MSC overhead, their size in bytes and their offset in the final stream of compressed data for different number of parallel blocks.

Table 20 MSC Overhead

Offset				Size (bytes)	Value			
1	2	3	4					
0	0	0	0	1	Size of one symbol in bits			
1	1	1	1	1	Number of threads			
2	2	2	2	4	Length of compressed data			
				6	4	Number of occurrences of thread root 3		
				10	4	Number of counters of thread root 3		
				14	4	Compressed data offset of thread 2		
				6	18	4	Number of occurrences of thread root 2	
				10	22	4	Number of counters of thread root 2	
				14	26	4	Compressed data offset of thread 1	
				6	18	30	4	Number of occurrences of thread root 1
				10	22	34	4	Number of counters of thread root 1
				14	26	38	4	Compressed data offset of thread 0
6	18	30	42	4	Number of occurrences of thread root 0			

After the MSC overhead is written, the parallel compression of counters in subtrees is launched. Initially, the counter of each node in MSC tree is reset and direction switches are set to left. The index of subtree root in NODE memories along with thread type are read from AUXILIARY memory. Initially, thread type is written to the appropriate output buffer. Knowing the index of the root, its *number of occurrences* in the input data is read, defining the number of times the subtree is traversed.

Then, the traversing is started and each node besides the root of MSC tree is investigated. This time, the traversing is governed solely by the switches. The compressed data are composed of counters coded by specified method for each node. The first compressed counter of each node has a header attached to it. The counters are read from STREAMS, which are stored in SDRAM. Each thread uses one port of the SDRAM interface as shown on Figure 27.

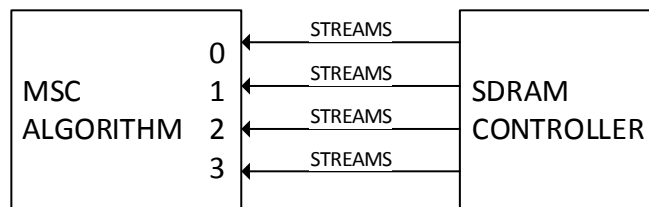


Figure 27 Dataflow on SDRAM interface during coding phase

The header differs for leaves and non-leaf nodes as shown in Table 21. First bit distinguishes between the leaf and non-leaf node. In case of leaf its symbol is included after the first bit. The selected best method, chosen for coding the particular node is the next item in the header, followed by coding parameters of the coding method. The best method identifier is 1 for Elias Alpha coding and 2 for ZEBC coding. The best method identifier is coded by Elias Alpha. Each method for coding counters contains a different number of coding parameters. Elias alpha coding does not have any and ZEBC coding stores the best base, which is also coded by Elias Alpha.

Table 21 Header format of node's first coded counter

LEAF	1	symbol (8 bits)	best method identifier	coding parameters
NON-LEAF	0	best method identifier	coding parameters	

In case a node is from different thread, only identifier of the thread, shown in Table 22, is written the first time the node is accessed and the counters of this node are not written to buffer of this thread. The first bit of the identifier is 0, which suggests that this bit is followed by identifier, not a symbol. In this case it is not the identifier of a method, but of CHILD THREAD that is written by Elias Alpha and which must be different from values of coding methods. It has value 4 to comply with the software implementation. It is followed by ID of thread, coded by Elias Alpha as well.

Table 22 Identifier of different thread

0	CHILD THREAD	ID of thread
---	--------------	--------------

The management of storing output values into output buffers is performed by the tree and counters themselves. If counter of an entered node is 0, new value from first unread position in thread STREAM is obtained and it is written to appropriate buffer. Each time a node is entered, its counter value is decremented by one. If the counter value after decrementing is 0, then direction of parent’s switch is changed.

6.10 Output of Compressed Data

The 8-bit values representing the output data, stored in the output buffers are transmitted via UART to computer as specified in section 5.6. The UART uses identical parameters as in case of loading the input data. Initially, the entire overhead is transferred to the computer. After that, the buffers are read in the order in which the compressed data are stored in the final stream, which is from the highest used thread index to 0. Each buffer is read until last byte with parity bit of value 1 is reached. This last byte is not transmitted to the computer.

The receiver in computer receives all data that are transmitted by the FPGA. The total number of received bytes is determined after first six bytes of overhead are received. The computer program remembers the length of compressed data, which is sent in last four bytes of these six bytes in Big Endian format. Hence, after these six bytes, the computer receives number of bytes that is equal to obtained length subtracted by 6.

One computer program is built for both parts – transferring of input data into an FPGA (described in section 6.1) and receiving the compressed data. The program is not part of this thesis. It was developed by one of the supervisors – doc. Ing. Vít Fábera, Ph.D.

7 Results

The task to implement the algorithm to greatest extent allowed by available resources was accomplished. There was only insufficient amount of time to finish pairing of the SDRAM memory with the FPGA. Despite the seemingly correct functioning, the read data on the output did not correspond to those on input. However, the functionality of the MSC algorithm was tested with success in a version utilizing only BRAM memories. This version allowed simulation in iSim.

To visualize signals of the implementation using SDRAM memory for testing purposes, a logical analyzer Omega from Asix company was used. Forte programmer from the same company was then utilized for the FPGA programming. The workplace layout is shown on Figure 28.

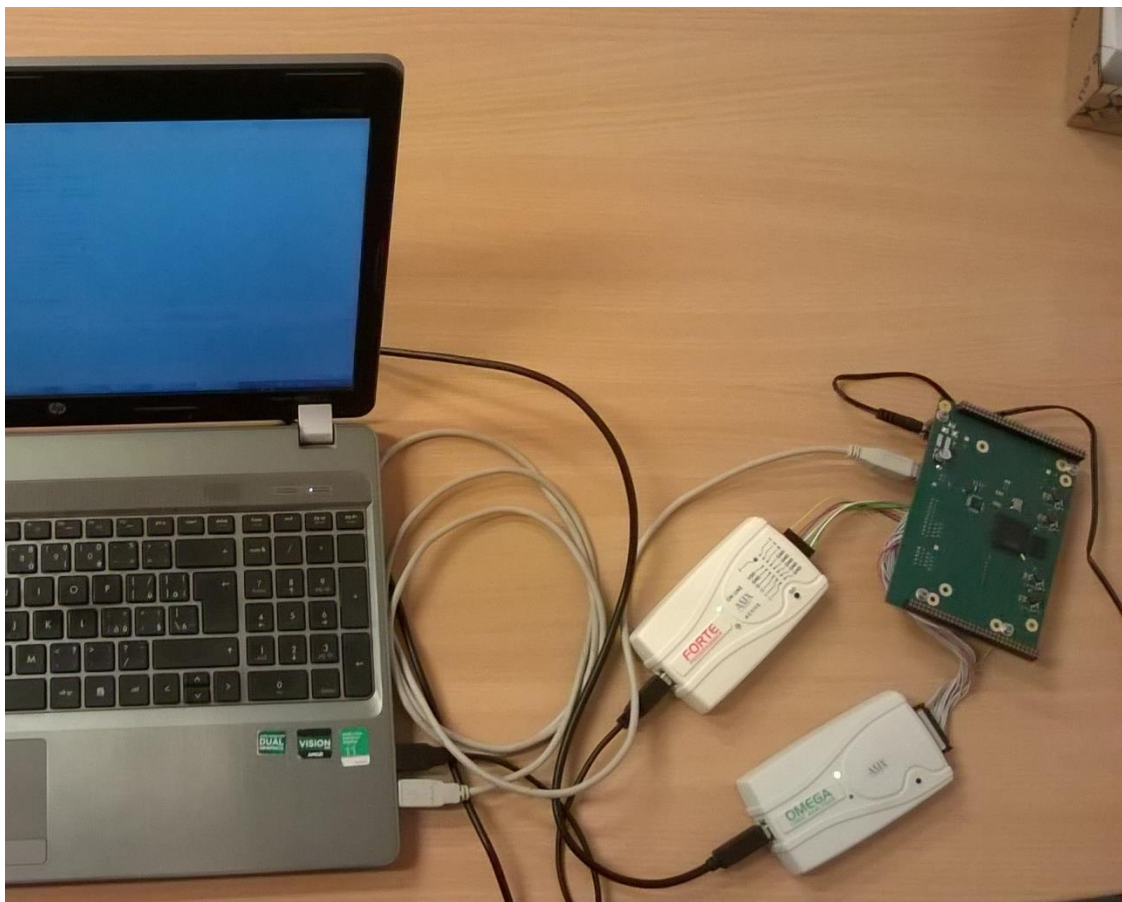


Figure 28 Workplace

The result of the thesis consists in evaluation of utilized resources of the used FPGA platform. Performance comparison of software and hardware implementation is not presented as there are not any applicable results of software implementation available and the other reason is the unfinished pairing of SDRAM and FPGA.

Table 23 shows the utilization of FPGA resources. The values are taken for the algorithm utilizing SDRAM memory and connected logic analyzer. Those values were recorded in the moment of finishing the thesis, so after completion of the project, they are supposed to change only minimally.

Table 23 FPGA resource utilization

Resource	Absolute usage	Relative usage in XC6SLX45 ⁴
Number of Slice Registers	9,265	16%
Number of Slice LUTs	14,346	52%
Number of occupied Slices	4,786	70%
Number of bonded IOBs	69	21%
Number of used BRAMs	25	21%

The chosen platform seems to be a reasonable choice. However, as it seems, Xilinx offers even more suitable XC7A35T model from Artix 7 family, which contains approximately 75% resources and the price is by 20 USD⁵ lower comparing to the selected platform.

For information, table 24 shows the utilization of IOBs. The majority of IOB pins is utilized for SDRAM memory and logic analyzer.

Table 24 IOB utilization

Purpose	Number of IOBs
Clock	1
UART	2
Logic Analyzer	16
SDRAM memory	50

⁴ XC6SLX45 is the chosen platform of Spartan 6 family

⁵ The prices are taken on 26/11/2016

The maximum frequency of the design that is to be programmed into the FPGA is provided at an output of the design synthesis. This frequency is determined by critical path delay incremented by periods needed to prevent timing violation. The synthesis tool calculated the maximum frequency of the design to be equal to 50.521MHz. Due to further optimizations after the synthesis process the frequency could be increased to 80 MHz. The SDRAM memory uses another clock domain of 320 MHz, which is easily obtained from the base frequency.

8 Future Considerations

The described implementation presents the first trial to code MSC algorithm to hardware platform. Some of the algorithm features had to be elaborated from the beginning. Some features were omitted or used in simple, non-optimized way. The implementation features can be divided into two not strictly separated groups – hardware platform and algorithm features. Both of them present big space for various modifications and improvements. Examples of the former group can be optimization for specific architecture, timing and area optimization techniques such as pipelining or resource sharing, and others. In following sections three algorithm modifications are proposed for future work.

8.1 Sorting Algorithm

In the described implementation, sorting algorithm is used twice. If Huffman coding was included this number would raise to four. If a more efficient algorithm was introduced, the compression time might get lower. The implemented sorter compares only two values at a time, however, the hardware implementation allows sorting of more items at once. For illustration, as [30] shows a big improvement in speed of parallel sorting compared to quicksort implemented in software can be achieved.

8.2 Decomposition of Tree to Layers

Besides the implementation described in this thesis, a MSC tree can be also broken into layers. A research shall be conducted, if this alternative way presents some advantageous properties. The designer must, however, bear in mind that the left tree representation does not allow direct traversing of the tree in layers and that in the compression phase of the algorithm the tree is traversed from top to bottom.

As the information about the parallel blocks is stored in the final compressed stream, the decompression module at the receiver side cannot use different parallel blocks for processing, so it is necessary to resolve this feature for both modules simultaneously.

8.3 Huffman Coding

To exploit full potential of MSC algorithm, Huffman coding should be considered as one of the coding options, especially if big files are compressed. With Huffman coding, a larger HW platform must be considered, because the coding requires considerable amount of additional logic and memory resources.

Especially, the analysis becomes much more complex, because in order to make decompression possible, a node compressed by Huffman method must include all the necessary information for reconstruction of the Huffman tree in the header. Therefore, besides the length for determination of the best method, the values needed for representation of Huffman tree are computed. At the end of analysis, the Huffman tree is built to get the binary representation of each counter. And this is performed for each node of the MSC tree. Fortunately, nodes are analyzed sequentially in parallel block, so some values are discarded after analysis of node is done and so the memory demand is lowered.

For further illustration, it was computed that in order to store 4 Huffman trees (because the analysis can run concurrently in 4 threads) twelve 18 Kb BRAMs would be required. Another auxiliary BRAM would be needed for building the trees and further memories for representation of the Huffman trees.

9 Conclusion

The thesis describes the first hardware implementation of MSC algorithm. The implementation is described in VHDL language and utilizes two methods for streams coding – Elias Alpha and ZEBC. The implementation limits the size of symbols in input data to 8 bits and number of characters to 65,535 in order to lower the memory requirements.

During the design phase, software implementation, written by the author of the algorithm, served as a main inspiration for building Finite State Machines that realize this compression algorithm in hardware. However, new features had to be introduced, because in hardware the memory is accessed directly without a support of pointers or objects. One of the outcomes is the invention of Left Tree Representation of MSC tree.

The implementation confirmed high memory demand of the algorithm comparing to other compression methods. It is because the algorithm encodes counters instead of data, which need to be stored in two memory structures in different forms – one for analysis, the other for compression.

The implemented design uses clock with frequency of 80 MHz without any explicit optimizations for the particular platform. The overall synthesis results confirm that the choice of the platform was reasonable.

The interface with PC, used for transfer of input and output of data utilizes UART protocol with 8 data bits and Baud rate of 115,200 Bd. A logic analyzer was used for visualization of signals during testing phase.

There were some problems that occurred during the implementation. In the process of simulation, some bugs were found in the design that required their correction. For example, a process of MSC tree building had to be adjusted by inserting creation of linked representation of tree. Most of the minor problems that were being solved resulted from overlooking important details in datasheets of the FPGA.

Owing to the shortage of time, the pairing of SDRAM memory with FPGA platform was unsuccessful. However, the functionality of the algorithm implementation for limited input data was tested using internal memories of the FPGA. Work on the project is planned to continue.

At the end of the thesis, possible modifications are introduced. Those should be considered during future implementations.

10 References

- [1]. **KOCHÁNEK, J.** Způsob transformace a bezztrátové komprimace dat v elektronické podobě. Czech Republic : Patent Application, 2007. Appl. no. 2007-114.
- [2]. **UNGER, L.** *Improvements of Multistream compression*. Prague : UK 2009, Diploma thesis, UK, Faculty of Mathematics and Physics, Department of Software Engineering.
- [3]. **KOCHÁNEK J., LÁNSKÝ J., UZEL P., ŽEMLIČKA M.** *The New Statistical Compression Method: Multistream Compression*. First International Conference on the Applications of Digital Information and Web Technologies [online], Ostrava : IEEE, 2008 [cit: 07/17/16]. <Available on: <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4664366>>.
- [4]. **SAYEED S., LISCANO R., AHMAD A.** Performance Measurement of XML Compression Algorithms for DSRC Messages. *32nd IEEE Conference on Local Computer Networks (LCN 2007) [online], Dublin, 2007*. 2007 [cit: 08/20/16], pp. 677-684. <Available on: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4367901&isnumber=4367785>>.
- [5]. TRIVIÁLNÍ ALGORITMUS PRO VYHLEDÁVÁNÍ VZORU. *Dokumentografické informační systémy - Komprese*. [Online] [cit: 07/17/16]. < Available on: http://www.ms.mff.cuni.cz/~kopecky/vyuka/dis/11/dis11_v1.html>.
- [6]. **KOCHÁNEK, J.** *New loseless compression method „MultiStream Compression“ (MSC) [Presentation]*.
- [7]. **CHU, P. P.** *RTL Hardware Design Using VHDL*. Cleveland : John Wiley & Sons, Inc, 2006. ISBN: 978-0-471-72092-8.
- [8]. **WOLF, W.** *Modern VLSI Design*. Upper Saddle River: Prentice Hall : Prentice Hall modern semiconductor design series., 2009. ISBN: 978-0-13-714500-3.
- [9]. **VAHID, F.** Presentations of Digital design [Presentation]. *University of California*. [Online] 2006 [online] [cit: 06/15/16]. <Available on: <http://www.ics.uci.edu/~harris/cs151/slides/>>.
- [10]. **XILINX [online]**. *Spartan-6 FPGA, Configurable Logic Block: User Guide*. 2010 [07/19/16]. <available on: http://www.xilinx.com/support/documentation/user_guides/ug384.pdf>.
- [11]. **ALTERA [online]**. *FPGA Architecture*. 2006 [cit: 08/15/16]. <Available on: https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf>.

- [12]. **XILINX.[online]**. *Spartan-6 FPGA SelectIO Resources*. 2015 [cit: 07/17/16]. <Available on: http://www.xilinx.com/support/documentation/user_guides/ug381.pdf>.
- [13]. **XILINX [online]**. *Spartan-6 FPGA, Block RAM Resources*. 2011 [07/19/16]. <available on: http://www.xilinx.com/support/documentation/user_guides/ug383.pdf>.
- [14]. **ALTERA [online]**. *Cyclone V Device Overview*. 2016 [cit: 08/15/16]. <Available on: https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf>.
- [15]. **XILINX [online]**. *User Guide: Xilinx Design Flow for Altera Users*. 2015 [cit: 08/16/16]. <Available on: http://www.xilinx.com/support/documentation/sw_manuels/ug1192-xilinx-design-for-altera.pdf>.
- [16]. **XILINX [online]**. *7 Series Product Selection Guide*. 2014 - 2015 [cit: 07/17/16]. <available on: <http://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf>>.
- [17]. **XILINX. [online]**. *Spartan-6 Family Overview*. 2011 [cit: 07/19/16]. <available on: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf>.
- [18]. **FPGA CENTRAL**. *FPGA Device Selection [Presentation]*. Published on: 2009 [online] [cit: 08/15/16]. <Available on: <http://www.slideshare.net/vkr101/fpga-device-selection>>.
- [19]. **XILINX [online]**. *User Guide: 7 Series FPGAs Configurable Logic Block*. 2014 [cit: 07/19/16]. <Available on: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf>.
- [20]. **ALTERA [online]**. *Altera Product Catalog, Version 16.0*. 2016 [cit: 08/13/16]. <Available on: https://www.altera.com/en_US/pdfs/literature/sg/product-catalog.pdf>.
- [21]. **ALTERA. [online]**. *Stratix III FPGAs vs. Xilinx Virtex-5 Devices :Architecture and Performance Comparison*. 2007. <Available on: https://www.altera.com/en_US/pdfs/literature/wp/wp-01007.pdf>.
- [22]. **XILINX [online]**. *Advantages of the Virtex-5 FPGA 6-Input LUT Architecture*. 2007 [cit: 08/15/16]. <Available on: http://www.xilinx.com/support/documentation/white_papers/wp284.pdf>.
- [23]. IEEE Standard VHDL Language Reference Manual," in IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) , vol., no., pp.c1-626, Jan. 26 2009 [online][cit: 07/15/16]. <Available on: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4772740>>.

- [24]. **XILINX [online]**. ISE In-Depth Tutorial. 2012 [cit: 08/25/16]. <Available on: http://www.xilinx.com/support/documentation/sw_manuels/xilinx14_1/ise_tutorial_ug_695.pdf>.
- [25]. **XILINX [online]**. ISim User Guide. 2012 [cit: 08/25/16]. <Available on: http://www.xilinx.com/support/documentation/sw_manuels/xilinx14_1/plugin_ism.pdf>.
- [26]. **DALLY W. J., HARTING R. C., AAMODT T. M.** *Digital Design Using VHDL*. Illustrated edition. Cambridge : Cambridge University Press, 2015. ISBN: 978-1-107-09886-2.
- [27]. **FISCHER, J.** *Přednáška UART, RS232, 422, 485 [Presentation]*. Praha : autor neznámý, 2013 [online] [cit: 08/17/16]. <Available on: http://measure.feld.cvut.cz/system/files/files/cs/vyuka/predmety/A4M38AVS/Pred_AV_S_2013_UART_RS232.pdf>.
- [28]. **CORMEN T. H. et al.,.** *Introduction to Algorithms 2nd ed.* Cambridge : MIT Press, 2002. ISBN: 978-0-262-03293-7.
- [29]. **ROHINI, S.** *Representation of binary tree in memory [Presentation]*. Published on: 2013 [online] [cit: 06/27/16]. <Available on: <http://www.slideshare.net/rrohinishinde/representation-of-binary-tree-in-memory>>.
- [30]. **MARCELINO R., NETO H. CARDOSO M. P.** *Sorting Units for FPGA-Based Embedded Systems*. In: KLEINJOHANN B., KLEINJOHANN L., WOLF M. *Distributed Embedded Systems: Design, Middleware and Resources [online]* : IFIP 20th World Computer Congress, TC10 Working Conference on Distributed and Parallel Embedded Systems (DIPES 2008), September 7-10, 2008, Milano, Italy, pp. 11 - 22. ISBN 978-0-387-09661-2. [seen on: 08/24/16]. <Available on: <http://link.springer.com/book/10.1007%2F978-0-387-09661-2>>.

List of appendices

Appendix A – ZEBC Table

Appendix B - Pseudocodes

Appendix C – Memory Structures

Appendices

APPENDIX A – ZEBC TABLE

Table of intervals of ZEBC coding		
Index	Beginning of interval	End of interval
0	0	1
1	2	5
2	6	13
3	14	29
4	30	61
5	62	125
6	126	253
7	254	509
8	510	1021
9	1022	2045
10	2046	4093
11	4094	8189
12	8190	16381
13	16382	32765
14	32766	65533
15	65534	131069
16	131070	262141
17	262142	524285
18	524286	1048573
19	1048574	2097149
20	2097150	4194301
21	4194302	8388605
22	8388606	16777213
23	16777214	33554429
24	33554430	67108861
25	67108862	134217725
26	134217726	268435453
27	268435454	536870909
28	536870910	1073741821
29	1073741822	2147483645
30	2147483646	4294967293

APPENDIX B - PSEUDOCODES

CREATE STAT

```
i = 0
while (i < SID)
    symbol = INPUT_DATA(i)
    no_of_occ = STAT(symbol).no_of_occ
    no_of_occ = no_of_occ + 1
    if (no_of_occ == 1)
        STAT(symbol).symbol = symbol
        STAT(symbol).first_occ = i
        STAT(symbol).sum = 1
    STAT(symbol).no_of_occ = no_of_occ
    i = i + 1
```

SQUEEZE AND SORT STAT

```
front = 0x00
back = 0xFF
do
    stat_front = STAT(front)
    stat_back = STAT(back)
    if(STAT (back).no_of_occ != 0)
        while(STAT(front).no_of_occ != 0 AND front < back)
            front = front + 1
            STAT(front) = stat_back
            front = front + 1
        back = back - 1
while(front <= back)
no_of_symbols = front
end = no_of_symbols - 1
do
    cnt = 0
    i = 0
    while(i < end)
        stat_i = STAT(i)
        stat_i1 = STAT(i+1)
        if(stat_i.no_of_occ > stat_i1.no_of_occ)
            tmp = stat_i1
            STAT(i+1) = stat_i
            STAT(i) = tmp
            cnt = cnt + 1
        i = i + 1
    end = end - 1
while(cnt > 0)
```


BUILD TREE

```

i = 0
j = 0
k = 0
while(i < no_of_symbols)
    stat_i = STAT(i)
    BUILD_TREE(i).symbol = stat_i.symbol
    BUILD_TREE(i).sum = stat_i.sum
    BUILD_TREE(i).no_of_occ = stat_i.no_of_occ
    BUILD_TREE(i).first_occ = stat_i.first_occ
    STAT(i).index = i
    i = i + 1
while(i < 2*no_of_symbols-2)
    stat_k = STAT(k)
    stat_k1 = STAT(k+1)
    if(stat_k1.first_occ < stat_k.first_occ)
        swap = 1
        BUILD_TREE(stat_k).parent = i
        BUILD_TREE(stat_k1).parent = i
    if (swap == 1)
        BUILD_TREE(i).left_tree = stat_k1.sum + 1
        BUILD_TREE(i).no_of_occ = stat_k1.no_of_occ + stat_k.no_of_occ
        BUILD_TREE(i).first_occ = stat_k1.first_occ
        BUILD_TREE(i).left_ch = stat_k1.index
        BUILD_TREE(i).right_ch = stat_k.index
        BUILD_TREE(stat_k).dir = 1
        BUILD_TREE(stat_k1).dir = 0
    else
        BUILD_TREE(i).left_tree = stat_k.sum + 1
        BUILD_TREE(i).no_of_occ = stat_k1.no_of_occ + stat_k.no_of_occ
        BUILD_TREE(i).first_occ = stat_k.first_occ
        BUILD_TREE(i).left_ch = stat_k.index
        BUILD_TREE(i).right_ch = stat_k1.index
        BUILD_TREE(stat_k).dir = 0
        BUILD_TREE(stat_k1).dir = 1
    j = k + 2
    stat_j = STAT(j)
    while(stat_j.no_of_occ < (stat_k1.no_of_occ + stat_k.no_of_occ) AND j
    < no_of_symbols)
        STAT(j-1) = stat_j
        stat_j = STAT(j)
        j = j + 1
    STAT(j-1).sum = stat_k.sum + stat_k1.sum + 1
    STAT(j-1).index = i
    STAT(j-1).no_of_occ = stat_k1.no_of_occ + stat_k.no_of_occ
    if swap == 1
        STAT(j-1).first_occ = stat_k1.first_occ
    else
        STAT(j-1).first_occ = stat_k.first_occ
    i = i + 1
    k = k + 1
index = 0
bt_i = BUILD_TREE(i)
NODE(index).no_of_occ = bt_i.no_of_occ
NODE(index).first_occ = bt_i.first_occ

```

```
NODE(index).counter = 0
NODE(index).type = ROOT
NODE(index).left_tree = bt_i.left_tree
NODE(index).dir = 0
index = index + 1
position = bt_i.left_ch
position2 = bt_i.right_ch
while(index < 2*no_of_symbols-2)
    bt_pos = BUILD_TREE(position)
    bt_pos2 = BUILD_TREE(position2)
    NODE(index).no_of_occ = bt_pos.no_of_occ
    NODE(index).first_occ = bt_pos.first_occ
    NODE(index).counter = 0
    NODE(index).left_tree = bt_pos.left_tree
    NODE(index).symbol = bt_pos.symbol
    NODE(index).dir = 0
    if(bt_pos.sum > 1)
        parent = position
        position = bt_pos.left_ch
        position2 = bt_pos.right_ch
        NODE(index).type = MIDDLE
        index = index + 1
    else
        NODE(index).type = LEAF
        LEAVES(bt_pos.symbol) = index
        index = index + 1
        while(1)
            NODE(index).no_of_occ = bt_pos2.no_of_occ
            NODE(index).first_occ = bt_pos2.first_occ
            NODE(index).counter = 0
            NODE(index).left_tree = bt_pos2.left_tree
            NODE(index).symbol = bt_pos2.symbol
            NODE(index).dir = 0
            if(bt_pos2.sum > 1)
                NODE(index).type = MIDDLE
                index = index + 1
                position = bt_pos2.left_ch
                position2 = bt_pos2.right_ch
                break
            else
                position = parent
                NODE(index).type = LEAF
                LEAVES(bt_pos2.symbol) = index
                index = index + 1
                do
                    desc_dir = BUILD_TREE(position).dir
                    parent = BUILD_TREE(position).parent
                    position = parent
                while(desc_dir == 1)
                bt_pos2 = BUILD_TREE(parent)
            if(index == 2*no_of_symbols)
                break
```

DETERMINE THREADS

note: memory ROOTS became part of memory AUXILIARY

```

if(no_of_threads > 2*no_of_symbols - 1)
    no_of_threads = 2*no_of_symbols - 1
thread = 0
index = 0
ROOTS(thread).index_node = index
NODE(index).index_thread = thread
if(no_of_threads == 1)
    ROOTS(thread).type = MAIN_WITHOUT
else
    ROOTS(thread).type = MAIN_WITH
    thread = 1
    node = NODE(index)
    NODE(index).par_thread = 0
    left_ch = NODE(index + 1)
    right_ch = NODE(index + node.left_tree)
    if(left_ch.no_of_occ > right_ch.no_of_occ)
        index = index + 1
        pair_index = index + node.left_tree
    else
        index = index + node.left_tree
        pair_index = index + 1
    ROOTS(thread).index_node = index
    NODE(index).index_thread = thread
    NODE(index).flag = 1
    ROOTS(thread).type = CHILD_WITHOUT
if(no_of_threads >= 3)
    thread = 2
    node = NODE(pair_index)
    if(node.type != LEAF)
        left_ch = NODE(pair_index + 1)
        right_ch = NODE(pair_index + node.left_tree)
        if(left_ch.no_of_occ > right_ch.no_of_occ)
            index2 = pair_index + 1
            pair_index2 = pair_index + node.left_tree
        else
            index2 = pair_index + node.left_tree
            pair_index2 = pair_index + 1
        ROOTS(thread).index_node = index2
        NODE(index2).index_thread = thread
        NODE(index2).flag = 1
    else
        ROOTS(thread).index_node = pair_index
        NODE(pair_index).index_thread = thread
        NODE(pair_index).flag = 1
    ROOTS(thread).type = CHILD_WITHOUT
    par_threads(thread) = 0
if(no_of_threads == 4)
    thread = 3
    node = NODE(index)
    if(node.type != LEAF)
        left_ch = NODE(index + 1)

```

```
right_ch = NODE(index + node.left_tree)
if(left_ch.no_of_occ > right_ch.no_of_occ)
    index = index + 1
else
    index = index + node.left_tree
par_threads(thread) = 1
ROOTS(1).type = CHILD_WITH
else
    index = pair_index2
par_threads(thread) = 0
ROOTS(thread).index_node = index
NODE(index).index_thread = thread
NODE(index).flag = 1
ROOTS(thread).type = CHILD_WITHOUT
```

CREATE STREAMS

note: memory STR_POS became part of memory AUXILIARY

```

i = 0
delimiter = 361
ls_items = 0
while(i < SID)
  n_index = 0
  symbol = INPUT_DATA(i)
  des_index = LEAVES(symbol)
  do
    index = n_index
    write = 0
    node = NODE(index)
    if(node.counter == 0)
      NODE(index).point_str = STR_POS(node.thread)
      str_pos(node.thread) = STR_POS(node.thread) + 1
      if(node.flag == 1)
        par_thr = NODE(index).par_thr
        NODE(index).point_par_str = STR_POS(par_thr)
        str_pos(par_thr) = STR_POS(par_thr) + 1
      NODE(index).counter = node.counter + 1
    if(node.type != LEAF)
      if(des_index < index + node.left_tree)
        n_index = index + 1
        pair_index = index + node.left_tree
        if(node.dir == 1)
          write = 1
        else
          n_index = index + node.left_tree
          pair_index = index + 1
          if(node.dir == 0)
            write = 1
    if(write == 1)
      pair_child = NODE(pair_index)
      STREAM(pair_child.thread, pair_child.point_str) = pair_child.counter
      if(pair_child.flag == 1)
        STREAM(node.thread, pair_child.point_par_str) =
          pair_child.counter
      if(pair_child.counter < delimiter)
        no_of_occ = SMALL_STAT(pair_index, pair_child.counter).no_of_occ
        SMALL_STAT(pair_index, pair_child.counter).no_of_occ = no_of_occ
          + 1
      else
        LARGE_STAT(pair_index, pair_child.ls_items).counter =
          pair_child.counter
        LARGE_STAT(pair_index, pair_child.ls_items).no_of_occ = 1
        NODE(pair_index).ls_items = pair_child.ls_items + 1
        NODE(pair_index).counter = 0
        NODE(index).dir = NOT node.dir
  while(index != des_index)
  i = i + 1

```

```
i = 0
while(i < 2*no_of_symbols - 1)
  node = NODE(i)
  if(node.counter != 0)
    STREAM(node.thread, node.point_str) = node.counter
    if(node.flag == 1)
      par_thr = NODE(index).par_thr
      STREAM(par_thr, node.point_par_str) = node.counter
    if(node.counter < delimiter)
      no_of_occ = SMALL_STAT(i, node.counter).no_of_occ
      SMALL_STAT(i, node.counter).no_of_occ = no_of_occ + 1
    else
      LARGE_STAT(i, node.ls_items).counter = node.counter
      LARGE_STAT(i, node.ls_items).no_of_occ = 1
      NODE(i).ls_items = node.ls_items + 1
  NODE(i).counter = 0
  NODE(i).dir = 0
  i = i + 1
```

GET_ZEBC_TABLE

```
ZEBC_TAB(0).begin = 0
ZEBC_TAB(0).end = 1
i = 1
while(i < 16)
    zebc = ZEBC_TAB(i-1)
    last_diff = zebc.end - zebc.begin + 1
    ZEBC_TAB(i).begin = zebc.end + 1
    ZEBC_TAB(i).end = zebc.begin + (last_diff*2) - 1
    if(delimiter < zebc.end)
        large_stat_ind = i
    i = i + 1
```

ANALYZE

note: METH_EA and METH_ZEBC are identifiers of coding methods, CHILD_THREAD is identifier of child thread

```

ln_anal = 0
end = 0
index = ROOTS(ind_of_thread).index_node
while(end != 1)
    node = NODE(index)
    if(index != 0)
        SQUEEZE_AND_SORT_LARGE_STAT
        GET_STAT_ITEMS_AND_MAX_SMALL_COUNTER
        ANALYZE_EA
        best_meth = METH_EA
        best_len_head = len_head
        best_len_body = len_body
        best_len_ = len_head + len_body
        ANALYZE_ZEBC
        if(best_len > len_head + len_body)
            best_meth = METH_ZEBC
            best_len_head = len_head
            best_len_body = len_body
            best_len_ = len_head + len_body
        NODE(index).best_meth = best_meth
        ln_anal = ln_anal + 1 + best_len
        if(node.type == LEAF)
            ln_anal = ln_anal + 8
        NODE(index).ls_items = ls_items
        NODE(index).best_base = best_base
        NODE(index).first_cntr_flag = 1
        NODE(index).sum_cntrs = sum_cntrs_ss + sum_cntrs_ls
        NODE(index).best_meth = best_meth
    if(index < 2*no_of_symbols - 2)
        left_ch = node(index + 1)
        right_ch = node(index + node.left_tree)
        if(left_ch.thread != ind_of_thread)
            if(right_ch.thread != ind_of_thread)
                end = 1
                if ((index + 1) != (index + node.left_tree))
                    ln_anal = ln_anal + 1 + CHILD_THREAD + left_ch.thread + 1 +
                        CHILD_THREAD + right_ch.thread
                    len_thread(ind_of_thread) = ROOTS(ind_of_thread).type_thread +
                        ln_anal
                else
                    index = index + node.left_tree
                    ln_anal = ln_anal + 1 + CHILD_THREAD + left_ch.thread
                    LOAD STATISTICS FROM SDRAM
            else
                index = index + 1
                if(right_ch.thread != ind_of_thread)
                    ln_anal = ln_anal + 1 + CHILD_THREAD + right_ch.thread
                    LOAD STATISTICS FROM SDRAM
        else
            end = 1
            len_thread(ind_of_thread) = ROOTS(ind_of_thread).type_thread + ln_anal

```


SQUEEZE_AND_SORT_LARGE_STAT

```
end = node.ls_items
sum_cntrs_ls = node.ls_items
do
  cnt = 0
  i = 0
  while(i < (end-1))
    large_stat_i = LARGE_STAT(i)
    large_stat_i1 = LARGE_STAT(i+1)
    if(large_stat_i.counter > large_stat_i1.counter)
      tmp = large_stat_i1
      LARGE_STAT(i+1) = large_stat_i
      LARGE_STAT(i) = tmp
      cnt = cnt + 1
    i = i + 1
  end = end - 1
while(cnt > 0)

old_beg = 0
ls_items = 0
last_cntr = 0
do
  cntr = LARGE_STAT(index, old_beg).counter
  if(cntr != last_cntr)
    ls_items = ls_items + 1
    LARGE_STAT(index, ls_items).counter = cntr
    last_cntr = cntr
  else
    no_of_occ = LARGE_STAT(index, ls_items).no_of_occ
    LARGE_STAT(index, ls_items).no_of_occ = no_of_occ + 1
  old_beg = old_beg + 1
while(old_beg < end)
```

GET_STAT_ITEMS_AND_MAX_SMALL_COUNTER

```
i = 0
max_small_cntr = 0
while(i < delimiter)
  ss_item = small_stat(index, i).no_of_occ
  if(ss_item != 0)
    sum_cntrs_ss = sum_cntrs_ls + ss_item
    max_small_cntr = i
  i = i + 1
```

ANALYZE_EA

```
len_head = METH_EA
len_body = node.no_of_occ
```

ANALYZE_ZEBC

```

best_base = 1
volume_stat_s = 0, volume_stat_l = 0
add = 2
ind = 0
end = ZEBC_TAB(ind).end
endl = end + 1
cntr = 1
i_b = 2
break = 0
while(cntr <= max_small_cntr)
    if(cntr > endl)
        ind = ind + 1
        end = ZEBC_TAB(ind).end
        endl = end + 1
        add = add + 2
    occ = SMALL_STAT(index, cntr).no_of_occ
    while(occ > 0)
        volume_stat_s = volume_stat_s + add
        occ = occ - 1
    cntr = cntr + 1
cntr = 0
while(cntr < ls_items)
    value = LARGE_STAT(index, cntr).counter - 1
    end = ZEBC_TAB(large_stat_ind).end
    while(value > end)
        ind = ind + 1
        end = ZEBC_TAB(ind).end
    occ = LARGE_STAT(index, cntr).no_of_occ
    while(occ > 0)
        volume_stat_l = volume_stat_l + ind + ind + 2
        occ = occ - 1
    cntr = cntr + 1
best_volume = volume_stat_s + volume_stat_l
while(i_b <= 50)
    ind = 0
    while(break != 1)
        zebc = ZEBC_TAB(ind)
        uli = i_b + zebc.begin - 1
        if(uli <= max_small_cntr)
            puli = SMALL_STAT(index, uli)
            volume_stat_s = volume_stat_s - puli
            uli_end = i_b + zebc.end - 1
            if(uli_end > max_small_cntr)
                cntr = max_small_cntr - uli
                uli = uli + 1
                while(cntr != 0)
                    puli = SMALL_STAT(index, uli).no_of_occ
                    uli = uli + 1
                    volume_stat_s = volume_stat_s + puli
                    cntr = cntr - 1
                break = 1
            else
                cntr = uli_end - uli
                uli = uli + 1

```

```
        while(cntr != 0)
            puli = SMALL_STAT(index, uli).no_of_occ
            uli = uli + 1
            volume_stat_s = volume_stat_s + puli
            cntr = cntr - 1
    else
        break = 1
        ind = ind + 1
    volume_stat_l = 0
    ind = large_stat_ind
    cntr = 0
    while(cntr < ls_items)
        value = LARGE_STAT(index, cntr).counter - i_b
        zebc = ZEBC_TAB(ind)
        while(value > zebc.end)
            ind = ind + 1
            zebc = zebc_tab(ind)
        while(occ > 0)
            volume_stat_l = volume_stat_l + i_b + ind + ind + 1
            occ = occ - 1
            cntr = cntr + 1
        volume = volume_stat_s + volume_stat_l
        if(best_volume > volume)
            best_volume = volume
            best_base = i_b
        i_b = i_b + 1
    len_head = METH_ZEBC + best_base
    len_body = best_volume
```

GET_COMPRESSION_OVERHEAD

note: WR_BIN and MSC_WR_ALPHA are writing functions described in section 6.8

```
size = 8
WR_BIN(size)
WR_BIN(no_of_threads)
buff_len = 0
i = 0
while(i < no_of_threads)
    x = len_thread(i)/8
    y = len_thread(i) - 8*x
    if(y != 0)
        x = x + 1
    buff_len = buff_len + x
    i = i + 1
i = 0, z = 0
while(i < no_of_threads)
    x = len_thread(i)/8
    y = len_thread(i) - 8*x
    if(y != 0)
        x = x + 1
    z = z + x
    buff_offset(i) = buff_len - z
    i = i + 1
i = 0
j = no_of_threads - 1
file_offset = 2 + ((no_of_threads*3) - 1)*4
while(i < no_of_threads)
    if(i == 0)
        x = buff_len + file_offset
        WR_BIN(x)
    else
        WR_BIN(buff_offset(j))
    WR_BIN(NODE(root(j).index).no_of_occ)
    if(i != (no_of_threads - 1))
        counters = node(root(j).index).sum_cntrs
        WR_BIN(counters)
    i = i + 1
    j = j - 1
```

COMPRESS

```

type = ROOTS(thread_ind).type_of_thread
index = ROOTS(thread_ind).index
root_ind = ROOTS(thread_ind).index
root_node = NODE(index)
cntr = root_node.no_of_occ
STR_POS(thread_ind) = 0
MSC_WR_ALPHA(type)
pos = STR_POS(thread_ind)
root_node.counter = STREAM(thread_ind, pos)
STR_POS(thread_ind) = pos + 1
if(type == MAIN_WITHOUT)
    while(cntr != 0)
        node = root_node
        parent = root_node
        do
            if(node.direction == 0)
                index = index + 1
            else
                index = index + node.left_tree
            node = NODE(index)
            if(node.counter == 0)
                if(node.first_cntr_flag == 1)
                    node.first_cntr_flag = 0
                    INIT_COMPR_NODE_STREAM(node)
                pos = STR_POS(thread_ind)
                node.counter = STREAM(thread_ind, pos)
                STR_POS(thread_ind) = pos + 1
                if(node.best_meth == METH_ELIAS_ALPHA)
                    MSC_WR_ALPHA(node.counter)
                elseif(node.best_meth == METH_ZEBC)
                    MSC_WR_ZEBC(node.best_base, node.counter)
                NODE(index).counter = node.counter - 1
            if(node.counter == 0)
                parent.direction = NOT parent.direction
                parent = node
        while(node.type != LEAF)

if(type == MAIN_WITH)
    while(cntr != 0)
        node = root_node
        parent = root_node
        break = 0
        do
            if(node.direction == 0)
                index = index + 1
            else
                index = index + node.left_tree
            node = NODE(index)
            if(node.index_thread != thread_ind)
                diff_thr = DIFF_THREAD(node.index_thread)
                if(diff_thr.counter == 0)
                    if(diff_thr.flag == 0)
                        DIFF_THREAD(node.index_thread).flag = 1
                        INIT_COMPR_THREAD_STREAM(node)

```

```

        pos = STR_POS(thread_ind)
        diff_thr.counter = STREAM(thread_ind, pos)
        STR_POS(thread_ind) = pos + 1
        DIFF_THREAD(node.index_thread).counter = diff_thr.counter - 1
        if((diff_thr.counter - 1) == 0)
            parent.direction= NOT parent.direction
        break
    else
        if(node.counter == 0)
            if(node.first_cntr_flag == 1)
                node.first_cntr_flag = 0
                INIT_COMPR_NODE_STREAM(node)
            pos = STR_POS(thread_ind)
            node.counter = STREAM(thread_ind, pos)
            STR_POS(thread_ind) = pos + 1
            if(node.best_meth == METH_ELIAS_ALPHA)
                MSC_WR_ALPHA(node.counter)
            elseif(node.best_meth == METH_ZEBC)
                MSC_WR_ZEBC(node.best_base, node.counter)
            NODE(index).counter = node.counter - 1
            if(node.counter == 0)
                parent.active = NOT parent.active
            parent = node
            if(node.type == LEAF)
                break
    while(1)

if(type == CHILD_WITH)
    while(cntr != 0)
        node = root_node
        parent = root_node
        break = 0
    do
        if(node.index_thread != thread_ind)
            diff_thr = DIFF_THREAD(node.index_thread)
            if(diff_thr.counter == 0)
                if(diff_thr.flag == 0)
                    DIF_THREAD(node.index_thread).flag = 1
                    INIT_COMPR_THREAD_STREAM(node)
                pos = STR_POS(thread_ind)
                diff_thr.counter = STREAM(thread_ind, pos)
                STR_POS(thread_ind) = pos + 1
                DIFF_THREAD(node.index_thread).counter = diff_thr.counter - 1
                if((diff_thr.counter - 1) == 0)
                    parent. active = NOT parent. active
                break
        else
            if(node.counter == 0)
                if(node.first_cntr_flag == 1)
                    node.first_cntr_flag = 0
                    INIT_COMPR_NODE_STREAM(node)
                pos = STR_POS(thread_ind)
                node.counter = STREAM(thread_ind, pos)
                STR_POS(thread_ind) = pos + 1
                if(node.best_meth == METH_ELIAS_ALPHA)
                    MSC_WR_ALPHA(node.counter)

```

```
        elseif(node.best_meth == METH_ZEBC)
            MSC_WR_ZEBC(node.best_base, node.counter)
        NODE(index).counter = node.counter - 1
        if(node.counter == 0 AND index != root_ind)
            parent.active = NOT parent.active
        parent = node
        if(node.type == LEAF)
            break
        if(node.direction == 0)
            index = index + 1
        else
            index = index + node.left_tree
        node = NODE(index)
    while(1)

if(type == CHILD_WITHOUT)
    while(cntr != 0)
        node = root_node
        parent = root_node
        do
            if(node.counter == 0)
                if(node.first_cntr_flag == 1)
                    node.first_cntr_flag = 0
                    INIT_COMPR_NODE_STREAM(node)
                pos = STR_POS(thread_ind)
                node.counter = STREAM(thread_ind, pos)
                STR_POS(thread_ind) = pos + 1
                if(node.best_meth == METH_ELIAS_ALPHA)
                    MSC_WR_ALPHA(node.counter)
                elseif(node.best_meth == METH_ZEBC)
                    MSC_WR_ZEBC(node.best_base, node.counter)
            NODE(index).counter = node.counter - 1
            if(node.counter == 0 AND index != root_ind)
                parent.active = NOT parent.active
            parent = node
            if(node.type == LEAF)
                break
            if(node.direction == 0)
                index = index + 1
            else
                index = index + node.left_tree
            node = NODE(index)
        while(node.type != LEAF)
```

INIT_COMPR_THREAD_STREAM

```
WR_BIN(0)
MSC_WR_ALPHA(CHILD_THREAD)
MSC_WR_ALPHA(node.index_thread)
```

INIT_COMPR_NODE_STREAM

```
if(node.type == LEAF)
    WR_BIN(1)
    WR_BIN(node.symbol)
else
    WR_BIN(0)
MSC_WR_ALPHA(node.best_meth)
if(node.best_meth == ZEB)
    MSC_WR_ALPHA(node.best_base)
```


APPENDIX C – MEMORY STRUCTURES

STATISTICS

address/data	32	24	16	8
0	symbol (31 downto 24)	sum (23 downto 15)	index (14 downto 6)	
1	no_of_occ (31 downto 16)		first_occ (15 downto 0)	
:				
510	symbol (31 downto 24)	sum (23 downto 15)	index (14 downto 6)	
511	no_of_occ (31 downto 16)		first_occ (15 downto 0)	

BUILD TREE1

address/data	32	24	16	8
0	symbol (63 downto 56)	sum (55 downto 47)		
:				
510	symbol (63 downto 56)	sum (55 downto 47)		

BUILD TREE2

address/data	32	24	16	8
0	no_of_occ (31 downto 16)		first_occ (15 downto 0)	
:				
510	no_of_occ (31 downto 16)		first_occ (15 downto 0)	

BUILD TREE3

address/data	32	24	16	8
0	left_ch (31 downto 23)	right_ch(22 downto 14)	parent(13 downto 5)	dir(4)
:				
510	left_ch (31 downto 23)	right_ch(22 downto 14)	parent(13 downto 5)	dir(4)

NODE1

address/data	32	24	16	8	3	2	1	0
0	no_of_occ (31 downto 16)		first_occ (15 downto 0)		par_thr			
:								
510	no_of_occ (31 downto 16)		first_occ (15 downto 0)		par_thr			

NODE2

address/data	32	24	16	8	3	2	1	0
0	counter (16)		symbol (8)	type (2)	left_tree (9)		dir	
:								
510	counter (16)		symbol (8)	type (2)	left_tree (9)		dir	

NODE3

address/data	32	24	16	8	3	2	1	0
0	point_str (35 downto 20)		point_par_thr (19 downto 4)		thr_i (3-2)	flg		
:								
510	point_str (35 downto 20)		point_par_thr (19 downto 4)		thr_i (3-2)	flg		

NODE4

address/data	32	24	16	8	3	2	1	0
0	ls_items	base (6)	flg	sum_counters (16)		best_m		
:								
510	ls_items	base (6)	flg	sum_counters (16)		best_m		

AUXILIARY

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
index_of root 0															type_thr		
index_of root 1															type_thr		
index_of root 2															type_thr		
index_of root 3															type_thr		
stream_position 0																	
stream_position 1																	
stream_position 2																	
stream_position 3																	
temp_counter 0																	
temp_counter 1																	
temp_counter 2																	
temp_counter 3																	
flag																	
flag																	
flag																	
flag																	
len_thread 0																	
len_thread 1																	
len_thread 2																	
len_thread 3																	

ZEBC TABLE

address/data	32	16	0	3	2	1	0
0	beginning	ending					
:							
15	beginning	ending					

LEAVES

address/data	16	8
0	index_of_node(15 downto 7)	
:		
255	index_of_node(15 downto 7)	

LARGE STATISTICS

address/data	32	24	16	8
0	no_of_occ		counter	
:				
180	no_of_occ		counter	

SMALL STATISTICS

address/data	16	8
0	no_of_occ	
:		
359	no_of_occ	

DIFF THREAD

address/data	16	8	1	0
0	counter0		flg	
1	counter1		flg	
2	counter2		flg	
3	counter3		flg	