

# Processing Core for Compressing Wireless Data

The Enhancement of a RISC Microprocessor

**Eskil Viksand Olufsen**

Master of Science in Electronics

Submission date: June 2006

Supervisor: Einar Johan Aas, IET

Co-supervisor: Tor Audun Ramstad, IET  
Robin Osa Hoel, Chipcon AS



# Problem Description

Compression of data in wireless systems is not a well-defined problem. Each application of compression needs to address different data types. This wide variety has made a single, efficient, compression technique hard to find. The only way to approach compression for this type of problem is to involve hybrid techniques or adapt different compression methods for different applications.

Today, most SoC solutions have an embedded microprocessor to handle complex control tasks, and Texas Instruments Norway has developed the NanoRisc microprocessor for this purpose. Texas Instruments Norway wishes to explore the NanoRiscs ability to process lossless data compression algorithms, and examine enhancements to improve its performance on these tasks.

The thesis should include an analysis of the NanoRiscs current ability to process different lossless data compression algorithms, and examine and implement area efficient enhancements to the NanoRisc core.

Assignment given: 16. January 2006  
Supervisor: Einar Johan Aas, IET



## **Abstract**

This thesis explores the ability of the proprietary Texas Instruments embedded 16 bits RISC microprocessor, NanoRisc, to process common lossless compression algorithms, and propose extensions in order to increase its performance on this task.

In order to measure performance of the NanoRisc microprocessor, the existing software tool chain was enhanced for profiling and simulating the improvements, and three fundamentally different adaptive data compression algorithms with different supporting data structures were implemented in the NanoRisc assembly language. On the background of profiling results, some enhancements were proposed:

- Bit field instructions.
- New load and store instructions for table data structures.
- An instruction improving read and writes of variable length codewords from memory.
- An instruction improving CRC-16 checksum calculation.
- Non-blocking load behavior.

These new enhancements improved throughput of the three implemented algorithms by between 18% and 103%, and the code sizes decreased between 6% and 31%. The bit field instructions also reduced RAM access by up to 53%. The enhancements were implemented in the NanoRisc VHDL model and synthesized. Synthesis reports showed an increase in gate count of 30%, but the whole NanoRisc core is still below 7k gates. Power consumption per MIPS increased by 7%, however reduced clock cycle count and memory access decreased the net power consumption of all tested algorithms. It is also shown that data compression with the NanoRisc prior to transmission in a low power RF transceiver may increase battery lifetime 4 times.

Future work should include a comprehensive study of the effect of the proposed enhancements to more common applications for the NanoRisc microprocessor.



## **Acknowledgements**

I want to express my gratitude to Texas Instruments Norway for giving me the opportunity to write this thesis and providing me with a work place.

Special thanks goes to my supervisor Robin Hoel for his guidance, advices and encouraging comments.

Thanks also to Peder Rand for patient guidance when helping me understand the tools and behavior of the NanoRisc core.

I also want to thank the employees at Texas Instrument Norway for a pleasant and educational working environment.





# Table of Contents

<b>INTRODUCTION .....</b>	<b>1</b>
<b>1 THEORY .....</b>	<b>2</b>
1.1 MEASURES.....	2
1.2 ENTROPY CODING .....	3
1.2.1 <i>Entropy Coding Schemes Used for Evaluation</i> .....	4
1.2.1.1 Rice Coding.....	4
1.2.1.2 Huffman Coding.....	5
1.2.1.3 LZ77.....	6
<b>2 RELATED WORKS.....</b>	<b>8</b>
2.1 PHILLIPS TRIMEDIA .....	9
2.2 ARM.....	10
<b>3 THE NANORISC PROCESSOR.....</b>	<b>12</b>
3.1 ARCHITECTURE.....	12
3.2 INSTRUCTION SET.....	14
3.3 TOOLS .....	16
3.3.1 <i>Assembler</i> .....	16
3.3.2 <i>Simulator</i> .....	17
<b>4 IMPLEMENTATION OF THE COMPRESSION ALGORITHMS.....</b>	<b>18</b>
4.1 IMPLEMENTATION OF RICE CODING .....	18
4.1.1 <i>Calculating the K-value</i> .....	19
4.1.2 <i>Modeling Stage</i> .....	20
4.2 IMPLEMENTATION OF HUFFMAN CODING .....	22
4.2.1 <i>Updating the Huffman tree</i> .....	22
4.2.2 <i>Implementing the Tree Data Structure</i> .....	24
4.3 IMPLEMENTATION OF LZ77 .....	25
4.3.1 <i>Simplified Deflate</i> .....	25
4.3.2 <i>Searching for Matches</i> .....	26
4.3.3 <i>The Dictionary Data Structure</i> .....	27
<b>5 ENHANCEMENTS OF EXISTING TOOLS.....</b>	<b>29</b>
5.1 PROFILING .....	29
5.1.1 <i>Profiling With the Assembler</i> .....	29
5.1.1.1 Implementation .....	30
5.1.2 <i>Profiling In the Simulator</i> .....	30
5.1.2.1 Implementation .....	32
5.2 ADDING NEW INSTRUCTIONS.....	33
5.2.1 <i>Adding New Instructions to the Assembler</i> .....	33
5.2.2 <i>Altering the Behavior of the Simulator</i> .....	33
<b>6 FINDING ENHANCEMENTS FOR THE NANORISC PROCESSOR.....</b>	<b>34</b>
6.1 INSTRUCTION LEVEL PROFILING .....	35
6.1.1 <i>Profiling Rice</i> .....	36
6.1.2 <i>Profiling Huffman</i> .....	37
6.1.3 <i>Profiling Deflate</i> .....	38
6.2 PROPOSALS FROM INSTRUCTION LEVEL PROFILING.....	39
6.2.1 <i>Instruction Level Enhancements</i> .....	39
6.2.2 <i>Adding Non-Blocking Load Behavior</i> .....	41
6.2.3 <i>Estimating Speedup</i> .....	42
6.3 ALGORITHMIC LEVEL PROFILING .....	43
6.3.1 <i>Profiling Rice</i> .....	43
6.3.2 <i>Profiling Huffman</i> .....	44
6.3.3 <i>Profiling Deflate</i> .....	45
6.4 PROPOSALS FROM ALGORITHMIC LEVEL PROFILING .....	46

6.4.1	<i>The Stream Function</i> .....	46
6.4.2	<i>The Hash Function</i> .....	48
6.5	PROPOSED ENHANCEMENTS FOR THE NANORISC .....	48
6.6	RESULTS OBTAINED FROM THE PROPOSED ENHANCEMENTS .....	50
<b>7</b>	<b>THE ENHANCED NANORISC PROCESSOR.....</b>	<b>51</b>
7.1	IMPLEMENTATION .....	51
7.1.1	<i>Non-Blocking Load Behavior</i> .....	52
7.1.2	<i>Bit Field Instructions</i> .....	53
7.1.3	<i>Clz Module</i> .....	55
7.1.4	<i>Crc Module</i> .....	56
7.1.5	<i>The Str Instruction</i> .....	57
7.2	SYNTHESIS.....	58
7.2.1	<i>Timing</i> .....	58
7.2.2	<i>Area</i> .....	59
7.2.3	<i>Power</i> .....	61
7.3	PERFORMANCE.....	62
7.3.1	<i>Energy Savings</i> .....	62
7.3.2	<i>Benchmarks</i> .....	64
<b>8</b>	<b>DISCUSSION .....</b>	<b>66</b>
8.1	ENHANCEMENTS .....	66
8.2	POWER.....	67
8.3	TIMING AND THROUGHPUT .....	67
8.4	ASSEMBLY SOURCE CODE .....	68
8.5	AREA.....	68
8.6	A COMPARISON .....	69
8.7	FUTURE WORK.....	70
8.7.1	<i>Processor Core</i> .....	70
8.7.2	<i>Testing</i> .....	70
8.7.3	<i>Tools</i> .....	71
	<b>CONCLUSION .....</b>	<b>72</b>
	<b>REFERENCES .....</b>	<b>73</b>
	<b>APPENDIX .....</b>	<b>74</b>
A.	NEW INSTRUCTIONS .....	75
B.	SYMBOL DISTRIBUTIONS.....	86
C.	INSTRUCTION LEVEL PROFILING .....	89
D.	ALGORITHMIC LEVEL PROFILING .....	109
E.	ZIP-FILE.....	129

## List of Figures

Figure 1, building of a Huffman tree.....	6
Figure 2, video codec.....	9
Figure 3, phillips trimedia architecture [7]. .....	9
Figure 4, ARM9E datapath [8].....	11
Figure 5, simple overwiev of the NanoRisc architecture [reference 8].....	12
Figure 6, memory set up. ....	13
Figure 7, NanoRisc data flow diagram. ....	14
Figure 8, Assembler command line syntax.....	16
Figure 9, NanoRisc ISS screenshot. ....	17
Figure 10, the Rice algorithm flowchart.....	19
Figure 11, calculation of the k-value using equation 6, the JPEG-LS method and the alternative approach. (symbol count is 16) [14].....	20
Figure 12, the Huffman algorithm flowchart.....	22
Figure 13, Huffman tree showing the sibling property. ....	23
Figure 14, updated Huffman tree. ....	23
Figure 15, node memory structure. ....	24
Figure 16, initial Huffman tree.....	24
Figure 17, the simplified Deflate flow chart.....	26
Figure 18, hash table with linked lists.....	28
Figure 19, profile labels syntax.....	29
Figure 20, enhanced NanoRisc ISS GUI.....	31
Figure 21, Profile window.....	32
Figure 22, new ISA dialog window.....	33
Figure 23, example insertion of a bit field (6 clock cycles).....	35
Figure 24, example addition on a bit field (3 clock cycles).....	35
Figure 25, example shift add operation for index storage (3 clock cycles).....	35
Figure 26, timing diagram during load operation. ....	42
Figure 27, old stream function for decoding algorithms. ....	46
Figure 28, new stream function for decoding algorithms.....	48
Figure 29, the enhanced NanoRisc data flow diagram.....	52
Figure 30, state machine during load instructions.....	53
Figure 31, data flow in the bit field example. ....	54
Figure 32, CLZ data flow diagram.....	56
Figure 33, CRC16-CCITT calculation with LFSR.....	57
Figure 34, shift operation during the <i>str</i> instruction.....	58
Figure 35, clock cycles used to decode, switch nodes and inserting nodes during Huffman decoding.....	65



## List of Tables

Table 1, Rice codes ( $k = 2$ ).....	5
Table 2, Encoding steps (LZ77 Example).....	6
Table 3, Incoming symbols (LZ77 Example).....	6
Table 4, original instruction encoding.....	15
Table 5, “pre” types and their encoding.....	16
Table 6, Table example for the implementation of the sorting method.....	21
Table 7, indicators for simplified Deflate.....	25
Table 8, instruction level profiling results from the Rice algorithm.....	36
Table 9, instruction level profiling results from the Huffman algorithm.....	37
Table 10, instruction level profiling results from the simple Deflate algorithm.....	38
Table 11, undefined space in the original ISA.....	39
Table 12, encoding of new instructions.....	39
Table 13, default “pre” values.....	41
Table 14, operations that may be affected by the enhancements.....	43
Table 15, estimated speedup from new instructions and NBL behavior.....	43
Table 16, algorithmic profiling from the Rice decoding algorithm with exponential distributed input stream.....	44
Table 17, algorithmic profiling from the Huffman decoding algorithm with poisson distributed input stream.....	44
Table 18, algorithmic profiling from the Deflate decoding algorithm with text input stream.....	45
Table 19, algorithmic profiling from the Deflate encoding algorithm with text input stream.....	45
Table 20, profiling results from the stream label when decoding the text stream.....	47
Table 21, encoding of the str instruction.....	47
Table 22, profiling results from the stream label when decoding the text stream and using the new str instruction.....	47
Table 23, encoding of the crc instruction.....	48
Table 24, new Instructions.....	49
Table 25, new “pre” types.....	49
Table 26, increase in throughput.....	50
Table 27, RAM access reduction.....	50
Table 28, reduction in code size due to new instructions.....	50
Table 29, compression ratios.....	50
Table 30, Register contents in bit field example.....	54
Table 31, gate count for the original NanoRisc.....	59
Table 32, contributions from each stage and module to the gate count at 25 MHz.....	60
Table 33, contributions from each stage and module to the gate count at 62.5MHz.....	60
Table 34, estimated power consumption at 25 MHz [mW].....	61
Table 35, estimated power consumption at 62.5 MHz [mW].....	61
Table 36, power consumption for 512x16 bits ROM and 2048x16 bits RAM [mW] (global voltage 1.62, 25Mhz operation).....	61
Table 37, power and energy consumption at 25 MHz.....	63
Table 38, energy consumption per bit for the enhanced NanoRisc.....	63
Table 39, energy reduction per bit for the enhanced NanoRisc.....	63
Table 40, reduction in energy consumption due to compression with the enhanced NanoRisc.....	63
Table 41, summary of results from “Energy Aware Lossless Data Compression” [25].....	69
Table 42, results from the implemented algorithms in the enhanced NanoRisc at 25 MHz.....	69



# List of Acronyms and Abbreviations

A list of acronyms and abbreviations that are not explicitly explained in the text.

- ASCII:** American Standard Code for Information Interchange. Standard 8 bits code used in data communications.
- ASIC:** Application Specific Integrated Circuit.
- CCITT:** Consultative Committee on International Telephony and Telegraphy. The international standards-setting organization for telephony and data communications.
- CCSDS:** Consultative Committee for Space Data Systems.
- CPU:** Central Processing Unit. Programmable logic device that performs all the instruction, logic, and mathematical processing in a computer.
- CRC:** Cyclic Redundancy Check. An error checking technique used to ensure the accuracy of transmitting digital data.
- DCT:** Discrete Cosine Transform. Mathematical transform used to convert signals from time domain to frequency domain.
- DEMUX:** De-Multiplexer. Splits a signal to pass over multiple signal paths.
- DSP:** Digital Signal Processor.
- GUI:** Graphical User Interface. A computer terminal interface, such as Windows, that is based on graphics instead of text.
- HW:** Hard Ware.
- I/O:** InOut
- IP:** Internet Protocol. Used for communications across a packet-switched network.
- JPEG:** Joint Photographic Experts Group. JPEG is a standards committee that designed a lossy image compression format.
- JPEG-LS:** A lossless image compression format.
- LAN:** Local Area Network.
- lsb:** Least Significant Bit.
- LSB:** Least Significant Byte
- MPEG:** Motion Picture Expert Group. Group defining the framework for a wide range of video and audio compression standards.
- MS:** Microsoft. Software company.
- msb:** Most Significant Bit
- MSB:** Most Significant Byte
- MUX:** Multiplexer. Allows two or more signals to pass over one signal path.
- NASA:** National Aeronautics and Space Administration. US agency which administer the American space program.
- OPS:** Operations per Second.
- PC:** Personal Computer.
- RAM:** Random Access Memory. Volatile memory used for data storage during operation.
- RISC:** Reduced Instruction Set Computing. Processor architectures where a low amount of instructions are needed to perform necessary tasks.
- ROM:** Read Only Memory. Nonvolatile memory often used as program memory.
- RTL:** Register Transfer Level. Describes logical operation in digital circuits.
- SCSI:** Small Computer System Interface. Parallel interface standard used by Apple Macintosh computers, PCs, and many UNIX systems for attaching peripheral devices to computers.
- SoC:** System on Chip. A chip which constitutes an entire system or major subsystem.
- VHDL:** A hardware modeling language. Commonly used for RTL modeling and synthesis.





## Introduction

Lossless data compression has become a standard feature in most high-speed communications networks. Data compression chipsets have been important for this development, and the significance of the V.42bis compression standard in modems is an example of this. The question is if data compression will play the same role for small wireless networks. If data compression can double or triple network throughput or significantly increase battery lifetime without harmful side effects, then the added complexity is worthwhile.

Not all data types are compressible and there are potential dangers such as data expansion, error propagation and incompatible standards. However, most commonly transmitted data is highly compressible. The aim of data compression for radio transmission is to save power or reduce bandwidth. Bandwidth is a precious commodity, and it is closely related to the bit rate ( $R = \text{bps}$ ). For ordinary binary-phase shift keying the null-to-null bandwidth is given by  $1.0R$ . Thus, if the number of data bits were reduced by half, then one would need only half the bandwidth. With the increase in use of wireless technology, it becomes more and more important that the bandwidth must be used efficiently. However, power can be saved by keeping the bandwidth and reduce airtime. Wireless transmission of one bit typically requires over 1000 times more energy than a single 16 bits computation. It is therefore justifiable to perform significant computation to reduce the number of bits transmitted, but limitations such as memory requirements, area constraints and throughput must be considered.

Today, most SoC transceiver solutions have an embedded microprocessor to handle complex control tasks, and Texas Instruments Norway has developed the NanoRisc microprocessor for this purpose. This thesis will explore the NanoRiscs current ability to process lossless data compression algorithms, and examine enhancements to improve its performance on this task. The work and this report have been divided into five main stages:

- A study of lossless compression algorithms, related works and the NanoRisc microprocessor.
- Implantation of three lossless compression algorithms in the NanoRisc assembly language.
- Enhancements of existing tools in order to measure performance of the NanoRisc and simulate improvements.
- Profile resource use when processing the implemented compression methods and propose improvements based on these results.
- Implement the proposed improvements in the NanoRisc microprocessor core, and synthesize the core in order to estimate changes in area, timing and power due to the implemented improvements.

The scope of this thesis does not include finding suitable data compression methods for wireless data. Only computational requirements have been considered when choosing algorithms for evaluating the NanoRiscs performance on different compression algorithms. Lossy compression methods have also not been considered. The thesis will cover some fundamental information theory, but the reader should be familiar with data compression and integrated circuit design.

# 1 Theory

This chapter will first describe some fundamental measures and terms before entropy coding is briefly explained and three compression algorithms are chosen for evaluating the NanoRisc current ability to process compression algorithms.

## 1.1 Measures

The field of mathematics concerned with data communications and storage is named information theory, and is generally considered to have been founded in 1948 by Claude E. Shannon [1]. He defined the information of a symbol  $x_n$  from the alphabet  $X$  to be:

$$\text{Eq. 1} \quad i(x) = -\log_2 P(x_n) \quad ;[1]$$

Where  $P(x)$  is the probability of the symbol occurring in the data stream. This could be described as how much knowledge is gained due to the observation of the symbol  $X = x_n$ . The logarithmic function can have any base, but by choosing 2 the measure can be translated to bits. An estimation of the average information gained from observing a sequence of symbols  $x_n$  from the alphabet  $X$  is called the Shannon entropy (or just entropy):

$$\text{Eq. 2} \quad H(X) = E[i(X)] = -\sum_{x \in X} P(x_n) \log_2 P(x_n) \quad ;[1]$$

This is an important measure when it comes to compression. For a lossless compression method, the Shannon entropy is the fundamental limit. This means that it is possible to compress the source in a lossless manner down to  $H(X)*n$ , where  $n$  is the number of symbols in the data stream. It is mathematically impossible to do better than  $H(X)*n$ . Equation 2 shows the first order model of the entropy. If there are statistical dependencies between symbols, higher order models can be used [1].

The redundancy of symbol  $x_n$  is:

$$\text{Eq. 3} \quad \rho(x) = l(x_n) - \log_2 \frac{1}{P(x_n)} \quad ;[1]$$

Where  $l(x_n)$  is the length of the symbol  $x_n$  in bits. The expected redundancy of alphabet  $X$  in the data stream is:

$$\text{Eq. 4} \quad E[\rho(X)] = \sum_{x \in X} P(x) \rho(x) = E[l(X)] - H(X) \quad ;[1]$$

There are several quantities used for compression performance. The quantity used in this thesis is the Compression Ratio:

$$\text{Eq. 5} \quad \text{CompressionRatio} = \left(1 - \frac{\text{OutputSize}}{\text{InputSize}}\right) * 100\%$$

When possible enhancements of the original NanoRisc processor are examined, some measures are needed in order to estimate the expected overall improvement. Amdahl's law [2] may be used for just that. This law is named after computer architect Gene Amdahl, and it is used to find the expected improvement to an overall system when only parts of the system are improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors. More technically, the law is concerned with the speedup achievable from an improvement to a computation that affects a proportion  $P$  of that computation where the improvement has a speedup of  $S$ . Amdahl's law states that the overall speedup of applying the improvement will be:

$$\text{Eq. 6} \quad S_{system} = \frac{1}{(1-P) + \frac{P}{S}} \quad ;[2]$$

If the result is e.g. 1.4, the improvement will make the system go 1.4 times faster.

## 1.2 Entropy Coding

Three compression algorithms are chosen to evaluate the NanoRisc processor. This section will give a short theoretical introduction to entropy coding and the coding schemes chosen. The details of the implementations are explained in chapter 4.

There are many known methods of data compression. Often they are suitable for different types of data, and produce different results. Any compression method is based on representing data in a way that reduces the redundancy as much as possible. To achieve this they exploit the statistical properties or the redundancy of the source data. The actual decrease of size is done by representing symbol values in a different way. A symbol that occurs often is encoded with a shorter codeword than a symbol that occurs rarely. Compression is only possible because data is normally represented in a format that is longer than necessary. Samples from a converter or instructions in a computer program often have a fixed length. This is done to make it easier to process data, since processing data is more common than compressing data.

Some compression methods are lossy. They achieve compression by removing non-vital information from the source. Pictures and audio are often compressed with a lossy compression method, since the human eye or ear is still capable of interpreting the information with a reduction of quality. In contrast, a computer program cannot be compressed in a lossy way because the computer will not be able to understand instructions if something are missing. When losing information is not acceptable, the data must be compressed with a lossless compression method. A lossless compression method will completely recover the original data from the compressed data. Entropy coding is defined as a coding scheme that assigns variable length codes to symbols so the code lengths match their probability. Lossless data compression methods are hence often called entropy coders. Entropy coding is often used as the last stage in lossy compression methods. After non-vital information is removed and complex methods have exposed statistical dependencies, entropy coding will make sure this is encoded in the shortest possible way (as close as possible to the entropy).

The process of entropy coding can often be split into modeling and coding. Modeling is a statistical analysis of the input data stream, and coding creates codewords from the statistics. These statistics may be frequencies of occurrence for different symbols, the existence of

repetitive sequences of symbols, dependencies in the frequency contents, etc. Modeling may be either static or adaptive. In static modeling, the same statistics is used every time coding is performed. Static modeling may be a good option if the source is well known and rigid. Adaptive modeling performs a statistical analysis every time coding is carried out. The method may be one-pass or two-pass. One-pass methods gather statistical information as the coding process goes forward and require thus only one pass of the input data stream. Two pass methods do one pass to gather statistical information, and another pass to do the coding. It is therefore necessary that the encoder in a two-pass method must pass the statistical information to the decoder. As established in Shannon's source coding theorem, there exists a relationship between the symbols probability and its shortest corresponding bit sequence. Since the statistical analysis is responsible for the evaluation of each symbols probability, modeling is one of the most important tasks in data compression. It is also important that the coding scheme is able to produce the shortest total output stream from the probability distribution found in the modeling.

### 1.2.1 Entropy Coding Schemes Used for Evaluation

Three fundamentally different entropy coding schemes are chosen to evaluate the NanoRisc processors current ability to process data compression algorithms:

- **Rice Coding** makes codewords directly from a value. These codewords are optimal if the input data stream is modeled to fit a geometrical probability distribution.
- **Huffman Coding** generates codes from a codebook and may fit any probability distribution. The codebook is usually held in a binary tree called a Huffman tree.
- **LZ77** detects patterns in the input stream and code lengths and pointers to where in the stream these patterns are found. (The actual algorithm implemented is called Deflate, and is a version of the LZ77 coding scheme.)

These three algorithms are chosen because they are fundamentally different from each other. Huffman and Rice coding are examples of statistical coding methods. They are heavily dependent on the quality of the modeling process or a precise static model. Even though Huffman and Rice are part of the same family of coding methods, they use very different methods. Huffman uses a codebook built on symbol probability in the data stream, while Rice produces codewords according to symbol value. It is important that the modeling stage produce low symbol values for the Rice encoder, while in Huffman only probabilities matter. The LZ77 coding method is a dictionary method. Dictionary methods utilize repetitive sequences of consecutive symbols in the input data stream. They build dictionaries of these sequences and encode where to find them. If the input stream consists of long and highly repetitive sequences, good compression ratios are achieved.

#### 1.2.1.1 Rice Coding

Rice coding is a selection of those Golomb codes that are easiest to produce in hardware. Golomb codes is a range of codes with a parameter  $m$  which encodes a positive integer  $n$  by encoding  $(n \bmod m)$  in binary followed by encoding  $(n \div m)$  in unary. When the parameter  $m$  is a power of two, the code is extremely efficient for use in computers since the division operation becomes a bitshift operation, and the remainder operation becomes a bitmask operation. This selection of Golomb codes is referred to as Rice codes. The disadvantage of the Rice coding is of course the restricted value of  $m$ , and therefore the compression may be

less effective than that of Golomb codes. In Rice coding the term  $k$ -value is often used, where  $m = 2^k$ . An example of Rice codes with a  $k$ -value of 2 are shown in Table 1.

Symbol Values	4-bit Binary	Quotient	Remainder	Code
0	0000	0	0	1 00
1	0001	0	1	1 01
2	0010	0	2	1 10
3	0011	0	3	1 11
4	0100	1	0	0 1 00
5	0101	1	1	0 1 01
6	0110	1	2	0 1 10
7	0111	1	3	0 1 11
8	1000	2	0	00 1 00
9	1001	2	1	00 1 01
10	1010	2	2	00 1 10
11	1011	2	3	00 1 11
12	1100	3	0	000 1 00
13	1101	3	1	000 1 01
14	1110	3	2	000 1 10
15	1111	3	3	000 1 11

Table 1, Rice codes (k = 2)

When the entropy increases, it is usually the lsbs that becomes more and more random. To deal with this the Rice code just cuts off the lsbs and passes them through without coding, but the msbs that may be less random are coded. It is clear from the table that the Rice code achieves the best compression for an input stream of symbols that have a geometric probability distribution. Rice coding is a widely used technique for entropy coding in image and sound compression methods.

### 1.2.1.2 Huffman Coding

Most variable-sized codes assume a given probability distribution of the symbols in the input data stream. The Huffman code is more general because it does not assume anything about the input symbol distribution, only that all probabilities are non-zero. Huffman was the first to develop an optimal algorithm for arbitrary probability distributions. This is achieved in the way the algorithm builds its codebook. Huffman first described this algorithm in a paper in 1952 [3]. The codebook is built in a binary tree structure (all nodes have only two children), and the algorithm follows these steps:

1. Consider all symbols as individual leaves with their probability as weight.
2. Find the two leaves with the lowest weight.
3. Make a new leaf with the weight of the two probabilities added together, and make the two found leaves children of the new leaf.
4. Repeat from step 2 as long as there are more than one leaf left.

The following example will show the building of a Huffman tree. If the alphabet  $X$  consists of the symbols  $A, B, C, D$  and  $E$ , with a probability distribution  $P(A)=0,42$ ,  $P(B)=0,3$ ,  $P(C)=0,12$ ,  $P(D)=0,09$  and  $P(E)=0,07$ , the Huffman tree would be built as shown in Figure 1.

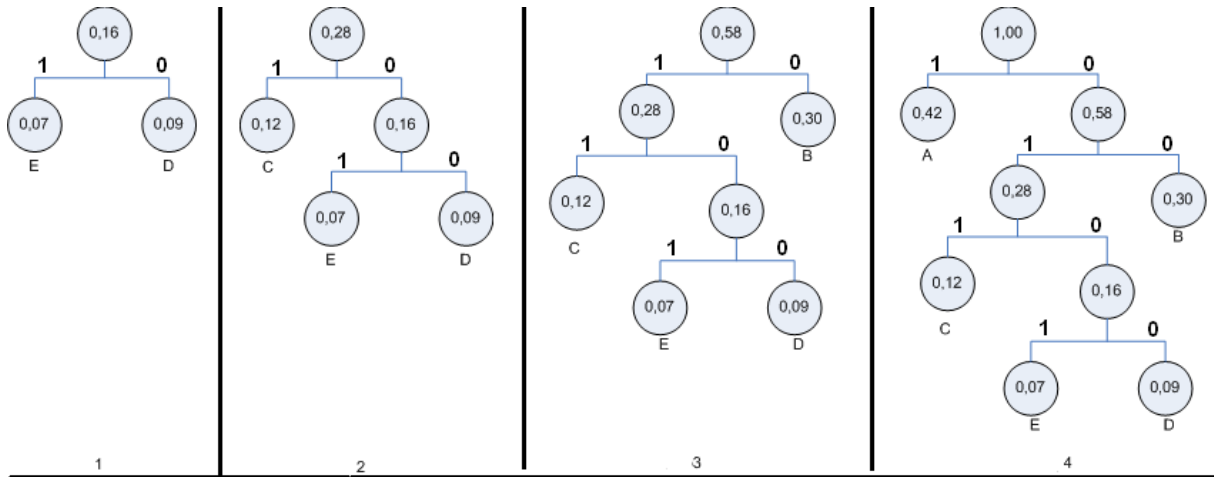


Figure 1, building of a Huffman tree.

When a symbol is read from the input stream, the code is created by traversing the tree from the leaf representing the code and to the root. By traversing right or left, the codeword is created with 1's or 0's. Decoding is done in similar matter, only the tree is traversed from the root to the leaf according to the code read. In the example from Figure 1, the codeword for *E* would be 0101.

### 1.2.1.3 LZ77

This method uses previously observed input data as a dictionary. During encoding, the input stream encoded so far is called the search buffer. New symbols ready to be encoded is called the look-ahead buffer. The search buffer and the look-ahead buffer are often referred to as a window. When new symbols are to be encoded, the method tries to find matches in the search buffer for the pattern of symbols on the input. The window may have finite length, and the method is often called a sliding window (as data is being encoded, the window slide over the data stream). The LZ77 is part of a family of coding methods that is called dictionary methods. One may think of the search buffer as a dictionary of words (where pattern of symbols make up words) and the look-ahead buffer as words needed to be looked up in the dictionary. The method will always try to find the longest match, and when this is found a pointer to the beginning of the match in the search buffer and the length of the match is made. This pair, the pointer and length, is called an index. When encoding, this index together with the first symbol in the input stream that did not match is encoded. Encoding of an incoming data stream, Table 2 , is shown in Table 3.

Pos	Symbol
1	A
2	A
3	B
4	C
5	B
6	B
7	A
8	B
9	C

Table 2, Incoming symbols (LZ77 Example)

Pos	Match	Symbol	Output
1	None	A	(0,0) + A
2	A	B	(1,1) + B
4	None	C	(0,0) + C
5	B	B	(2,1) + B
7	AB	C	(5,2) + C

Table 3, Encoding steps (LZ77 Example)

During decoding, the decoder will build up the same search buffer as the encoder. When the decoder reads a new index, it finds the beginning of the match in the search buffer and outputs the sequence according to its length. After that, it outputs the symbol following the index. The search buffer in the decoder consists of symbols decoded so far. It is evident from this description that looking up indexes (decoding) is much faster than searching for them (encoding), thus the LZ77 is an asymmetrical coding method.

## 2 Related Works

Since the original computer systems were designed for text processing and scientific applications, but not tasks such as audio and video compression, many enhancements of original computer systems have been towards multimedia applications. One of the most common solutions has been to include instructions that are optimized for typical multimedia applications. Almost all major processor manufacturers have developed their own set of media instructions. Examples include Motorola's AltiVec extensions to the PowerPC instruction set, and Intel's MMX, SSE, and SSE2 extensions to the x86 instruction set. The extensions differ in data path width, number and type of registers provided, as well as the availability of specific operations. Motorola's AltiVec and Intel's SSE and SSE2 have 128 bits datapaths, floating point arithmetic, and they support in the region of over 100 instructions. The kinds of extensions these examples represent are of course farfetched for the enhancement of the NanoRisc processor, but the basic idea of adding special instructions and behavior in order to increase processing power for specific tasks is applicable.

Programs that manipulate data at subword level (bit fields smaller than the bit width of the processor core) are common for many embedded applications, e.g. media and network processing. In fact in many cases the input or output of embedded applications consists of packed data, and these applications spend a significant amount of time packing and unpacking narrow width data into memory words. A paper [4] by Bengu Li and Rajiv Gupta at the university of Arizona, showed that by adding a bit section instruction set extension to an ARM processor reduced the instructions executed at runtime between 5% and 28%, while the code size was reduced by between 2% and 21%. These results were gathered from testing the extensions with various benchmark suits from network, media and control applications. They also showed that by adding the extension the register pressure decreased. Before the extension was added the applications needed registers and memory locations to hold values in packed and unpacked form, but with the new bit field extension, this was not necessary. Thus, memory requirements and cache activity decreased as a result of more efficient register use.

Media extensions to microprocessors are also heavily studied. In 1994, a study at Hewlett-Packard by Ruby B. Lee [5] showed that by introducing a small set of new instructions to a PA-RISC microprocessor, enabled for the first time an entry-level workstation to achieve MPEG video decompression and playback at real-time rates. Since this, digital audio and video have made progress and is now the future in all major media broadcast systems. However, the compression methods used require a large amount of processing power. For example, National Television Systems Committee (NTSC) resolution MPEG-2 decoding requires more than 400 MOPS, and 30 GOPS are required for encoding [6]. To meet this task, many microprocessor manufacturers have made special processor cores in order to target real-time processing of multimedia.

Almost every audio and video compression standard has a lossless entropy coding stage. A typical video codec system is shown in Figure 2. The lossy source coder performs filtering, transformation (DCT), subband decomposition, quantization, etc. (these tasks are not covered by this thesis). The output from the source coder still exhibits redundancy, and the lossless entropy coder removes this. The rest of this chapter will describe briefly two popular embedded microprocessor cores for media processing, and how their features may speed up the entropy coder stage.



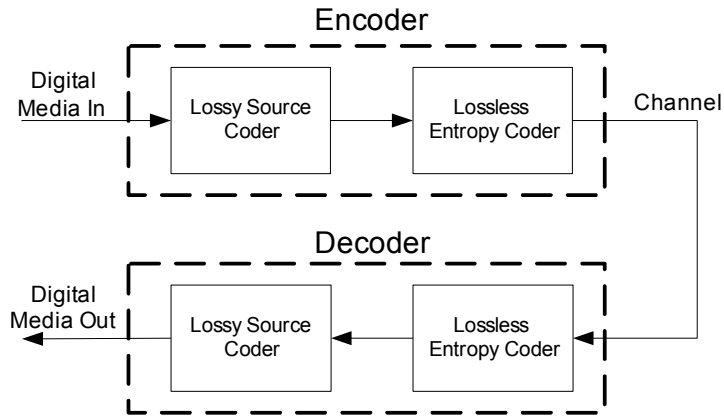


Figure 2, video codec.

### 2.1 Phillips Trimedia

TriMedia/CPU [7] is a VLIW (Very Long Instruction Word) core, and it is optimized for multimedia applications. VLIW means that the core issues one long instruction every clock cycle, and each instruction consists of several operations. Each operation is comparable to a RISC machine instruction. In order to process such instructions, the architecture has five parallel data paths (Figure 3).

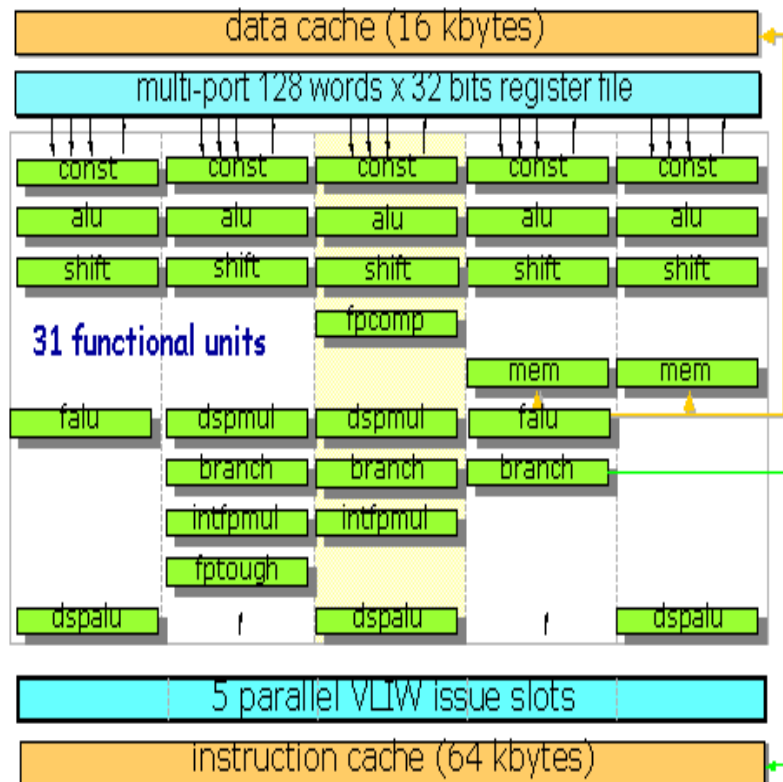


Figure 3, phillips trimedia architecture [7].

The instruction set includes some custom operations that may increase throughput for entropy coders. Among these are 7 instructions for packing, merging and selecting bits. This is an important feature when dealing with variable length codes. Another feature is the VLD

(Variable Length Decoder) which is a coprocessor. This coprocessor is made especially to take care of the Huffman decoding in MPEG1 and MPEG2. The VLD receives as input a pointer to an MPEG1 or MPEG2 bit stream and some configuration information. After the initialization, the TriMedia controls the VLD by a set of five commands:

- Shift bit stream by some number of bits
- Search for the next start of the code
- Reset the VLD
- Flush output fifos

The VLD produces as output a data structure that contains all of the information necessary to complete the video decoding process. This coprocessor is especially developed as a solution to the serial nature of the entropy coder stage in MPEG1 and MPEG2. Even though the encoder stage is not the most complex task in these compression methods, it tends to be a bottleneck because the algorithm makes it difficult to exploit the parallel features of most media processors.

When coprocessors are used, they may do their task in parallel to the main processor, and hence introduce parallelism on an algorithmic level when parallelism on an instruction level is difficult. This is a much used approach, and is often called HW acceleration. When these HW accelerators are used, they are design to do a large part of a task, like the VLD coprocessor. However, the cost is often increased gate count and power consumption. If a HW accelerator is implemented the resulting speed up should be considerable. A big HW accelerator will also make the design closer to an ASIC. Since it takes on a large portion of a certain task, it may become useless in other applications. This is the case for the VLD unit, which is designed especially for the MPEG1 and MPEG2 standard.

## **2.2 ARM**

The ARM (Acorn RISC Machine) is a 32 bits RISC processor architecture that is widely used in a number of embedded designs; in fact the ARM family accounts for over 75% of all 32 bits embedded CPU's. The ARM DSP [8] is developed to meet applications that require a DSP-oriented processor because of their high signal processing content, in addition to handle complex control tasks. The ARM development team claims that the feature of having a microcontroller supporting both control and signal processing has many advantages over traditional solutions based on a separate DSP and control processors. This reasoning also applies for the enhancement of the NanoRisc. If data compression is to be performed in a transceiver SoC, it has many advantages if the data compression algorithm could be processed in the embedded microcontroller. The advantages could be saving power, area, design time, etc. Figure 4 shows the datapath in an ARM DSP microprocessor.

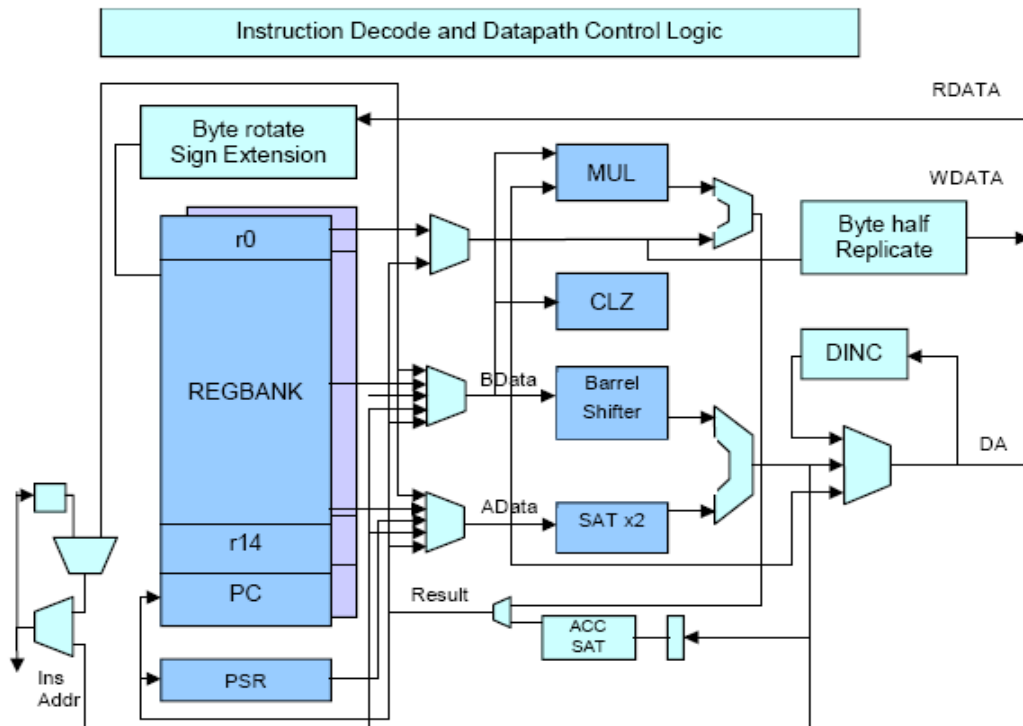


Figure 4, ARM9E datapath [8].

The new modules added to support the DSP enhanced extensions is a 32x16 multiplier, a CLZ (Count Leading Zeroes) module, and two saturation modules. The new arithmetic modules are of little interest to most entropy coders, but the CLZ module may be useful for some algorithms. It is controlled by the “clz” instruction, and it counts the number of leading zeroes in one register and writes the answer to another. ARM has also made additional extensions toward multimedia applications called NEON [8]. This extension uses SIMD instructions (Single Instruction Multiple Data) which include bit field operation such as:

- Bit Field Clear
- Bit Field Insert
- Signed Bit Field Extract
- Unsigned Bit Field Extract

ARM has also developed the Cortex-M3 core [8] that is a powerful processor with some interesting features. In addition to the count leading zero and bit field instructions, its memory map includes something called bit-band regions. These bit-band regions map each word in an alias region of the memory to a bit in a bit-band region of memory, i.e. it can address the memory at bit level in the bit-band regions. The Cortex-M3 core has two bit-bands at the lowest 1MB of the SRAM and peripheral memory regions respectively.

The features mentioned for the different ARM cores and extensions may be very helpful when processing compression algorithms. Bit field operations are important when dealing with variable length codes because they make it possible to access packed codewords within a register. The bit-band regions will ease reading and writing a stream of variable length codewords to or from memory, and the count leading zero instruction should be very useful when decoding Rice-like codewords.

### 3 The NanoRisc Processor

The NanoRisc processor was developed as part of a thesis by Peder Rand [9] for Chipcon AS (now Texas Instruments Norway). Chipcon desired an on-chip firmware processor in order to cope with the increasing complexity of their SoC products. This processor is designed to manage internal control and data processing tasks. It is a compact and effective microcontroller core that can control complex processes and move and process data. The processor features 13 general 16 bits registers, a full 16 bits ALU, an 8x8 multiplier, a 16 bits barrel shifter, and a load/store module with auto increment/decrement. It has up to 32 bit-addressable I/O ports and interrupt handling, which contributes to its easy integration into any design. It is controlled by a compact and comprehensive set of 16 bits instructions, but is still capable of immediate 16 bits memory addressing without the use of paging. This chapter will describe shortly the features of the NanoRisc processor and tools. The interested reader is referred to [9].

#### 3.1 Architecture

The NanoRisc is a simple RISC processor [10]. It is a load/store architecture which means that operations can only be performed on data stored in the registers. It features single cycle execution of all instructions that do not read from memory. A simple overview of the architecture is shown in Figure 5.

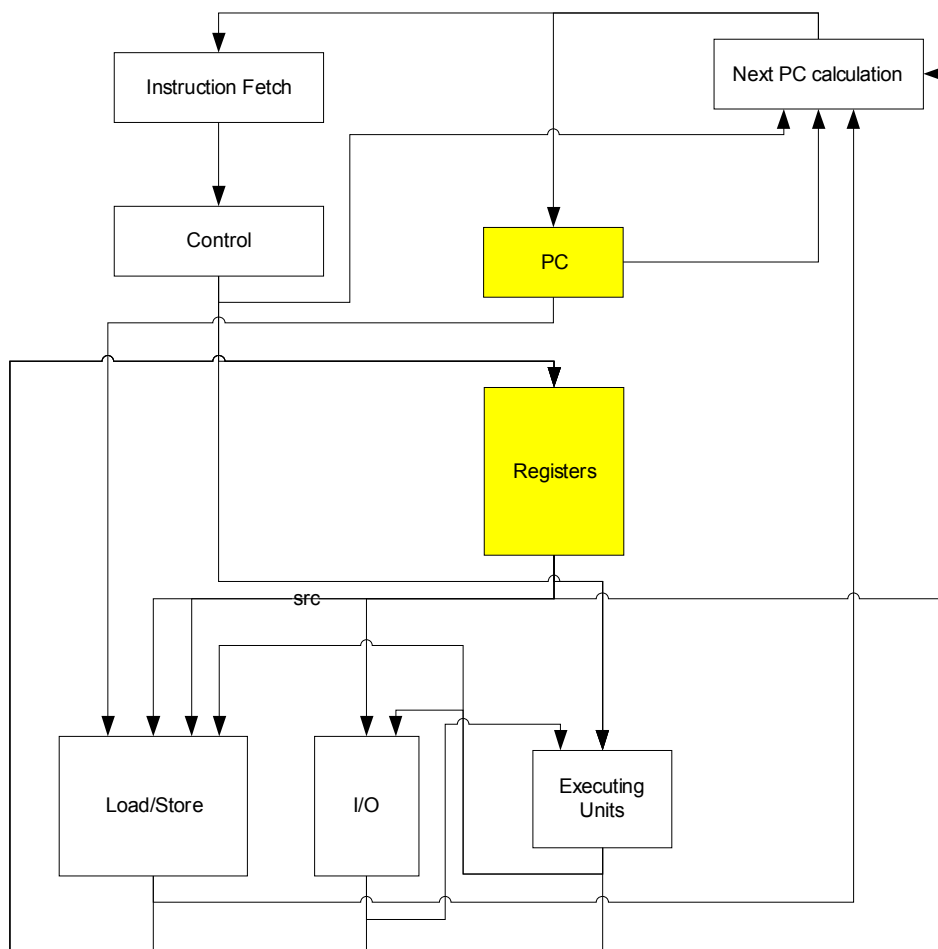


Figure 5, simple overview of the NanoRisc architecture [reference 8].

There are three different memory spaces in the NanoRisc. There are 16 addressable registers (13 general registers, one 16 bits stack pointer, one 7 bits status register, and a 15 bits program counter), 32 bit-addressable I/O ports (16 input ports and 16 output ports), and a combined 16 bits program and data memory space. The 13 16 bits general registers are without any dedicated role, and can be used in any operation where a source and/or a destination register are required. The stack pointer (SP), status register (SR) and program counter (PC) have dedicated roles. The SP and SR can be used in all operations where a source and/or a destination register are required. The I/O ports allow the NanoRisc processor to connect to peripherals or other NanoRisc processors. These ports are accessed and controlled by dedicated instructions. The program and data memory of the NanoRisc share address space, but they have a separate memory bus going out of the NanoRisc. In this thesis, a separate ROM is assumed for the program memory, and a synchronous RAM for the data memory. This setup is shown in Figure 6.

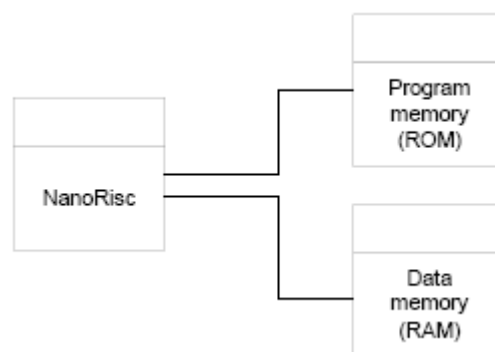


Figure 6, memory set up.

The implementation of the NanoRisc is based on a centralized principle where the instruction is decoded and all control signals are set in the processor control unit (PCU) module. The multiplexers and registers that implement a module are implemented in their respective modules. Figure 7 shows the NanoRisc data flow.

Short description of the modules in Figure 7:

- **PCU**, Processor Control Unit (PCU) decodes instructions and set control signals to the other modules.
- **ALU**, Arithmetic Logical Module has two 16bit operand inputs, and consist of a 16 bits carry propagate adder and “xor”, “and”, “or” and an inverter unit.
- **FETCH**, generates the address of the next instruction to be executed. How this is done depends on the current instruction.
- **I/O**, interface peripheral modules.
- **MEM**, controls the reading and writing of data memory.
- **MUL**, 8x8 bit multiplication module. It is generated by the infix VHDL operator ‘\*’ which produces a multiplier from the Synopsys DesignWare library at synthesis.
- **REG**, register bank which holds the special and general registers. The module has two read ports and one write port.
- **SHIFT**, barrel shifter with one signal path for left shifting and one signal path for right shifting. The value shifted in may be a carry from the status register or the carry from the shift operation (which is done when rotating).
- **SRC**, multiplexer that chooses the source operand for several of the functional modules.

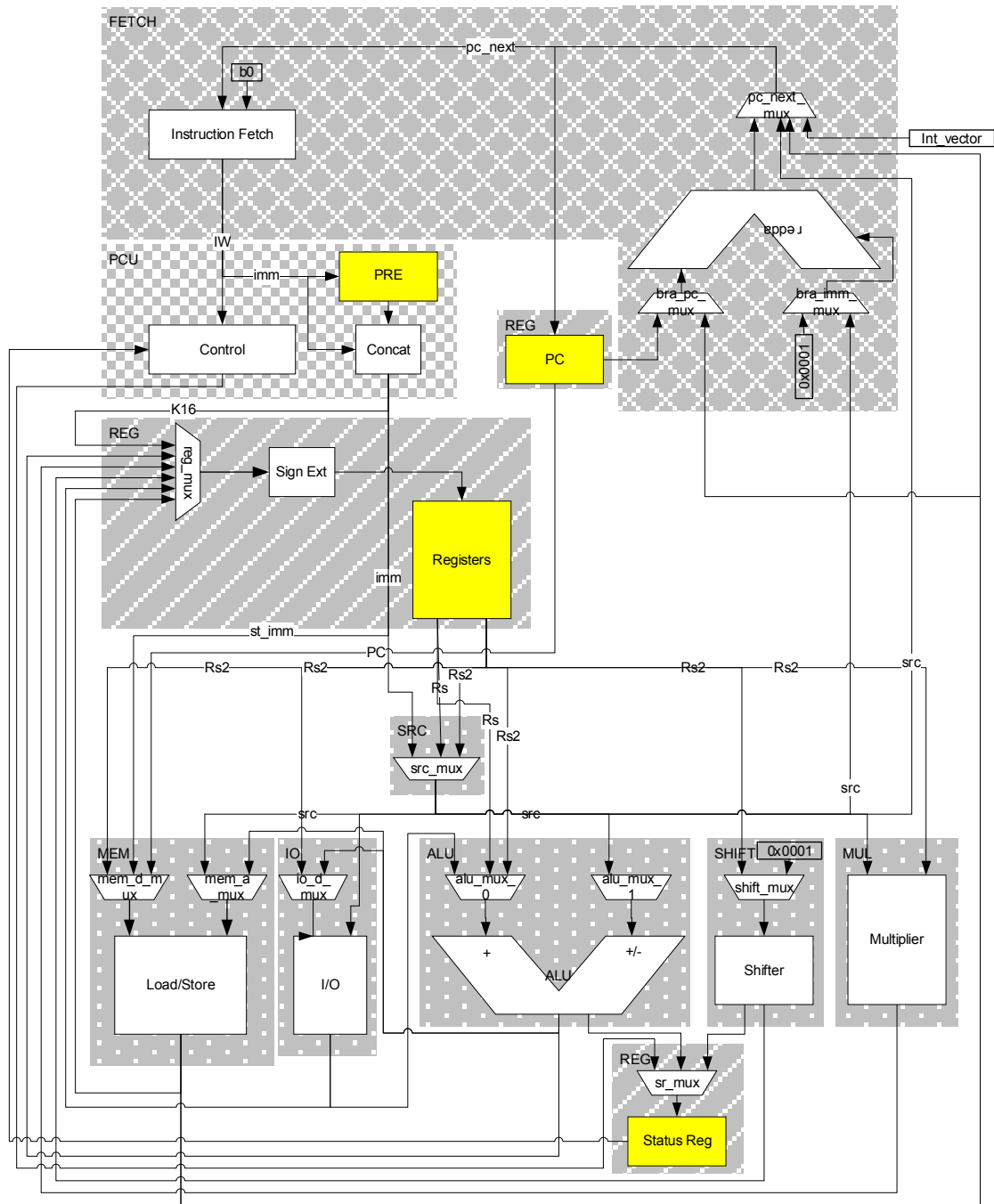


Figure 7, NanoRisc data flow diagram.

### 3.2 Instruction Set

The machine code instruction set of the NanoRisc consists of 55 16 bits instructions. A summary of the instruction encoding, how many cycles for execution and a short description is given in Table 4. A NanoRisc assembly language is made from this instruction set. The default program flow is to execute the next instruction located after the current instruction in the program memory. This default flow can be overridden by either acknowledging an interrupt or changing the flow by a branch, call or return instruction, where the interrupt takes priority.

Mnemonic	Bits															Description	Cycles	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1			0
addi	0	U7							Rd				0	0	0	0	Add Immediate	1
muli	0	U7							Rd				0	0	0	1	Unsigned Mul Immediate of LSB	1
addci	0	U7							Rd				0	0	1	0	Add Immediate with Carry	1
xori	0	U7							Rd				0	0	1	1	Logic XOR Immediate	1
subi	0	U7							Rd				0	1	0	0	Sub Immediate	1
andi	0	U7							Rd				0	1	0	1	Logic AND Immediate	1
subci	0	U7							Rd				0	1	1	0	Sub immediate with borrow	1
ori	0	U7							Rd				0	1	1	1	Logic OR Immediate	1
ld	0	U7							Rd				1	0	0	0	Load word	2
st	0	U7							Rd				1	0	0	1	Store word	2
lds	0	U7							Rd				1	0	1	0	Load word relative to stackpointer	1
sts	0	U7							Rd				1	0	1	1	Store word relative to stackpointer	1
cmpi	0	U7							Rd				1	1	0	0	Compare Immediate	1
tsti	0	U7							Rd				1	1	0	1	Logic Test Immediate	1
movi	0	U7							Rd				1	1	1	0	Load Immediate	1
dbra	0	U7-							Rd				1	1	1	1	Decrement and branch	1
ldo	1	0	1	0	Ra				Rd				U4				Load word with offset	2
sto	1	0	1	1	Ra				Rd				U4				Store word with offset	1
pre	1	1	0		S9[0];Rs2;S9[7:0]				or 0;size;S11 / 1;S12								Load 13-bit PRE register	1
bra	1	1	1	0	S8								0	CC			Conditional branch	1
rol/ror	1	1	1	0	Amount (or Rs)				Rd				1	0	0	dir	Rotate left/right	1
scl/scr	1	1	1	0	Amount (or Rs)				Rd				1	0	1	dir	Shift in from carry left/right	1
sll/slr	1	1	1	0	Amount (or Rs)				Rd				1	1	0	dir	Shift left/right	1
sar	1	1	1	0	Amount (or Rs)				Rd				1	1	1	0	Arithmetic shift right	1
calli	1	1	1	0	S8								1	1	1	1	Call immediate	1
add	1	1	1	1	Rs				Rd				0	0	0	0	Add	1
mul	1	1	1	1	Rs				Rd				0	0	0	1	Unsigned Mul of LSB	1
addc	1	1	1	1	Rs				Rd				0	0	1	0	Add with Carry	1
xor	1	1	1	1	Rs				Rd				0	0	1	1	Logic XOR	1
sub	1	1	1	1	Rs				Rd				0	1	0	0	Subtract	1
and	1	1	1	1	Rs				Rd				0	1	0	1	Logic AND	1
subc	1	1	1	1	Rs				Rd				0	1	1	0	Subtract with borrow	1
or	1	1	1	1	Rs				Rd				0	1	1	1	Logic OR	1
lda	1	1	1	1	Ra				Rd				1	0	d/i	0	Load word with inc/dec	2
sta	1	1	1	1	Ra				Rd				1	0	d/i	1	Store word with inc/dec	2
cmp	1	1	1	1	Rs				Rd				1	1	0	0	Compare	1
tst	1	1	1	1	Rs				Rd				1	1	0	1	Logic Test	1
mov	1	1	1	1	Rs				Rd				1	1	1	0	Copy register content	1
zxt	1	1	1	1	1	0	0	0	Rd				1	1	1	1	Zero-extend LSB into MSB	1
sxt	1	1	1	1	1	0	0	1	Rd				1	1	1	1	Sign-extend LSB into MSB	1
jmp	1	1	1	1	1	0	1	0	R				1	1	1	1	Indirect absolute jump	1
call	1	1	1	1	1	0	1	1	R				1	1	1	1	Indirect absolute call	1
ret	1	1	1	1	1	1	0	0	x	x	x	x	1	1	1	1	Return from subroutine	2
reti	1	1	1	1	1	1	0	1	x	x	x	x	1	1	1	1	Return from interrupt	2
inv	1	1	1	1	1	1	1	0	R				1	1	1	1	Invert	1
nop	1	1	1	1	1	1	1	1	x	x	x	x	1	1	1	1	No-operation	1
rdio	1	0	0	0	Rs				Rd				0	0	0	0	Read from I/O port	1
wrio	1	0	0	1	Rs				Rd				0	0	0	0	Write to I/O port	1
rdioi	1	0	0	U5					Rd				0	0	1	0	Read from I/O port immediate address	1
wrio	1	0	0	U5					Rd				0	1	0	0	Write to I/O port immediate address	1
iosc	1	0	0	U5					Bitnum				0	1	1	0	Set bit in I/O out register to value of carry	1
iobs	1	0	0	U5					Bitnum				1	0	0	0	Set bit in I/O out register	1
iobc	1	0	0	U5					Bitnum				1	0	1	0	Clear bit in I/O out register	1
iots	1	0	0	U5					Bitnum				1	1	0	0	Test bit in I/O in register	1
iotg	1	0	0	U5					Bitnum				1	1	1	0	Toggle bit in I/O in register	1

Table 4, original instruction encoding.

For the NanoRisc to be able to interpret 16 bits immediate values and instructions with 3 operands, a “pre” instruction is used. This instruction will precede the actual instruction, and it is loaded into a dedicated register in the instruction decoder. The “pre” instruction may also hold additional information about the execution of the instruction. An instruction which may use a “pre” instruction will determine how it is interpreted if it is present. Any such instruction will also clear the dedicated “pre” register. There are currently 5 ways of interpreting the “pre” instruction, and the way they are encoded is referred to as types. This is shown in Table 5.

Type	Bits															Description	Cycles	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1			0
Type 1	1	1	0	rs[0]	Rs2			S9[8:1]									PRE format for 2-register/reg+immediate	1
Type 2	1	1	0	imm	Rs2			x	x	x	x	x	x	x	x		PRE format for shift/rotate	1
Type 3	1	1	0	0	size	S10											PRE format for address extension	1
Type 4	1	1	0	1	S12												PRE format for store immediate	1
Type 5	1	1	0	r/a	a/o	CC		U8									PRE format for jmp/calli, r/a=relative/absolute	1
																	a/o="and/or" extra CC, CC=extra CC (111=always)	

Table 5, “pre” types and their encoding.

### 3.3 Tools

There exist two important tools for the NanoRisc processor; an assembler and a simulator. The assembler encodes the NanoRisc assembly language into machine code. The simulator is PC software for a windows platform, and it simulates the behavior of the NanoRisc at a cycle accurate level. It is important to understand how the assembler and simulator are made in order to modify or enhance them. This section will give a short introduction to the existing tools.

#### 3.3.1 Assembler

The NanoRisc assembly language is based on the NanoRisc instruction set (Table 4). Each assembly instruction consists of a mnemonic followed by a possible empty list of arguments, and enables the user to produce all possible machine code instructions from the instruction set. To ease the use of instructions that may need “pre” instructions, the assembler will automatically insert “pre” instructions whenever this is needed.

The GNU Assembly Preprocessor GASP [11] should be used on source code before the NanoRisc assembler is used. This preprocessor includes support for macros with conditional statements, loops, variables, inclusion of files and all other wanted preprocessor functionality. The NanoRisc assembler is implemented as a Windows command-line executable with syntax as seen in Figure 8.

`nr_asm input_file [output_file]`

Figure 8, Assembler command line syntax.

Currently, the only supported output of the assembler is a simple dump of the instruction words in ASCII hex format followed by the word address of the instruction, the originating filename and line number.

The assembler is written in MS Visual C++ using the lexer generator FLEX [12] and parser generator Bison [13] to generate the lexer and parser. The lexer reads the input stream searching for sequences of characters matching the patterns accepted in the programming language it is reading. The Lexer must therefore recognize all instruction mnemonics, directives, register names, alias identifiers, constant values and operators used in the NanoRisc assembly language. When the Lexer is called, it returns a token with corresponding value, an error or end of file. The parser uses a description of the syntax of the programming language to identify constructs of the language that give meaning, and it sees the program as sequence of program lines that can be an instruction, a label definition or a directive. When one of these program line types is identified, the parser expects a list of arguments of the correct type.



### 3.3.2 Simulator

The NanoRisc instruction set simulator (ISS) simulates the behavior of the NanoRisc processor at a cycle accurate level. This means that after execution of one clock cycle of a program, its status is the same as for a processor running with the same input. The NanoRisc ISS is written in MS Visual C++ using the Microsoft Foundation Class (MFC) library for window handling.

The simulator has a graphical user interface (GUI) that provides a simple way to supervise and control the simulation. The most important features of the ISS are listed below, and a screenshot is shown in Figure 9.

- Memory view for viewing specified addresses in memory.
- Possibility to load data memory contents.
- Reload button to quickly restart simulation.
- I/O view of specified I/O ports with hexadecimal and graphical representation.
- Register overview.
- Cycle counter, program counter and current instruction word clearly displayed
- Full disassembler
- Code view showing instruction word, program address, filename, line number, assembly code and the number of times it has been executed for each instruction in the program.
- Highlighting of current instruction and color coding of most visited instructions
- “Run”, “Step” and “Run to cursor” modes with possibility to break execution at any point.
- Unlimited number of user defined breakpoints.

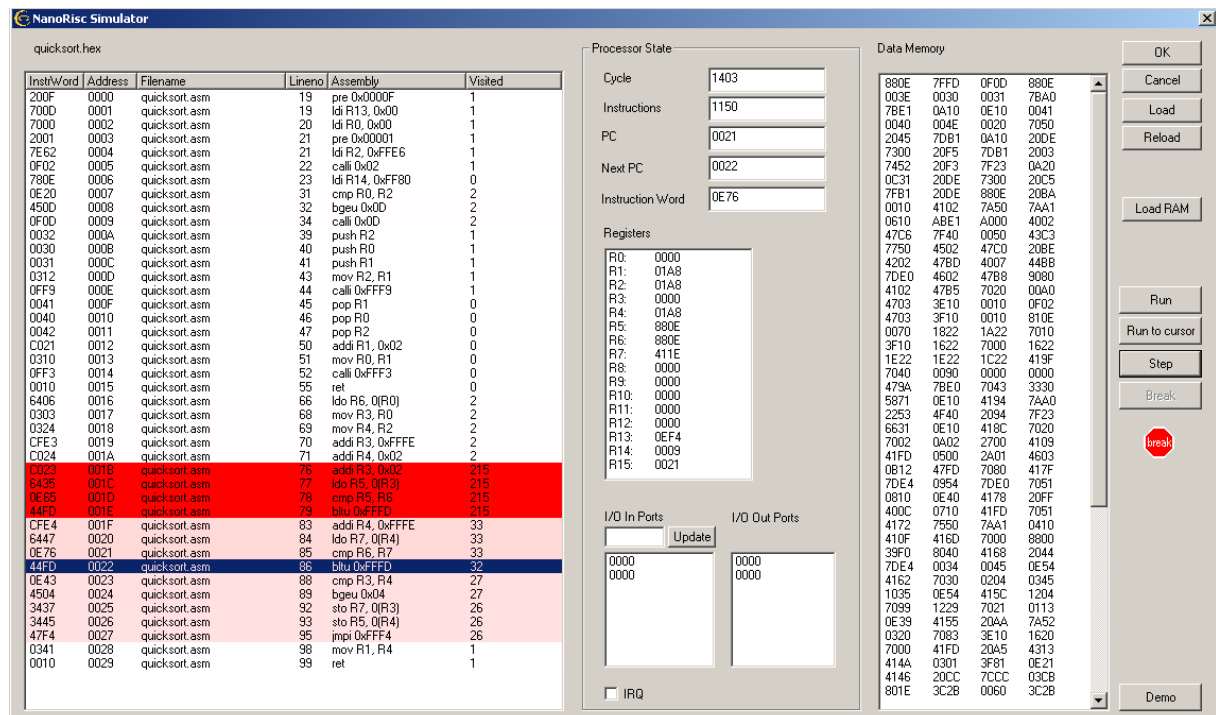


Figure 9, NanoRisc ISS screenshot.

## 4 Implementation of the Compression Algorithms

To identify possible improvements of the NanoRisc processor, three different compression algorithms are implemented with the current instruction set and HW capabilities. A short theoretical introduction to the three algorithms is represented in section 1.2.1. In that section it is also a short discussion of why these compression algorithms were chosen for implementation. The main argument was their fundamentally different methods of encoding and decoding the data stream. In the actual implementation, other important differences become more visible, and may in fact dominate. Since all three methods are heavily dependent on the modeling stage, they require large supporting data structures for this task. To explore the capabilities of the NanoRisc microprocessor different data structures are chosen, and all algorithms are implemented with adaptive one-pass modeling stages. A short summary will be given in here, but the differences will become more evident in the next sections where the implementations are explained in detail.

- **Rice coding** is implemented with sorted tables that gives each symbol a code value according to its index.
- **Huffman coding** is implemented with a binary tree data structure. The tree consists of linked nodes.
- **LZ77** is implemented with a hash table and linked lists. The hash value is made from a CRC hash function.

The algorithms are written in the NanoRisc assembly language, and the enhanced NanoRisc assembler makes the instruction words. All assembly source codes are found in appendix E.

### 4.1 Implementation of Rice Coding

From section 1.2.1.1 it is evident that the most demanding task of Rice coding is making a good modeling stage and the calculation of the  $k$ -value. The encoding stage is just bit shifts and bit masking, and decoding is counting zeroes and bit masking. The implemented  $k$ -value calculation is developed by the author [14], and the modeling stage uses tables to sort the data stream according to frequency counts. The tables will assign low code values to high-sorted symbols. A simple flowchart of the implementation is shown in Figure 10. The  $k$ -value calculation and the modeling stage will be described in detail in the next subsections.

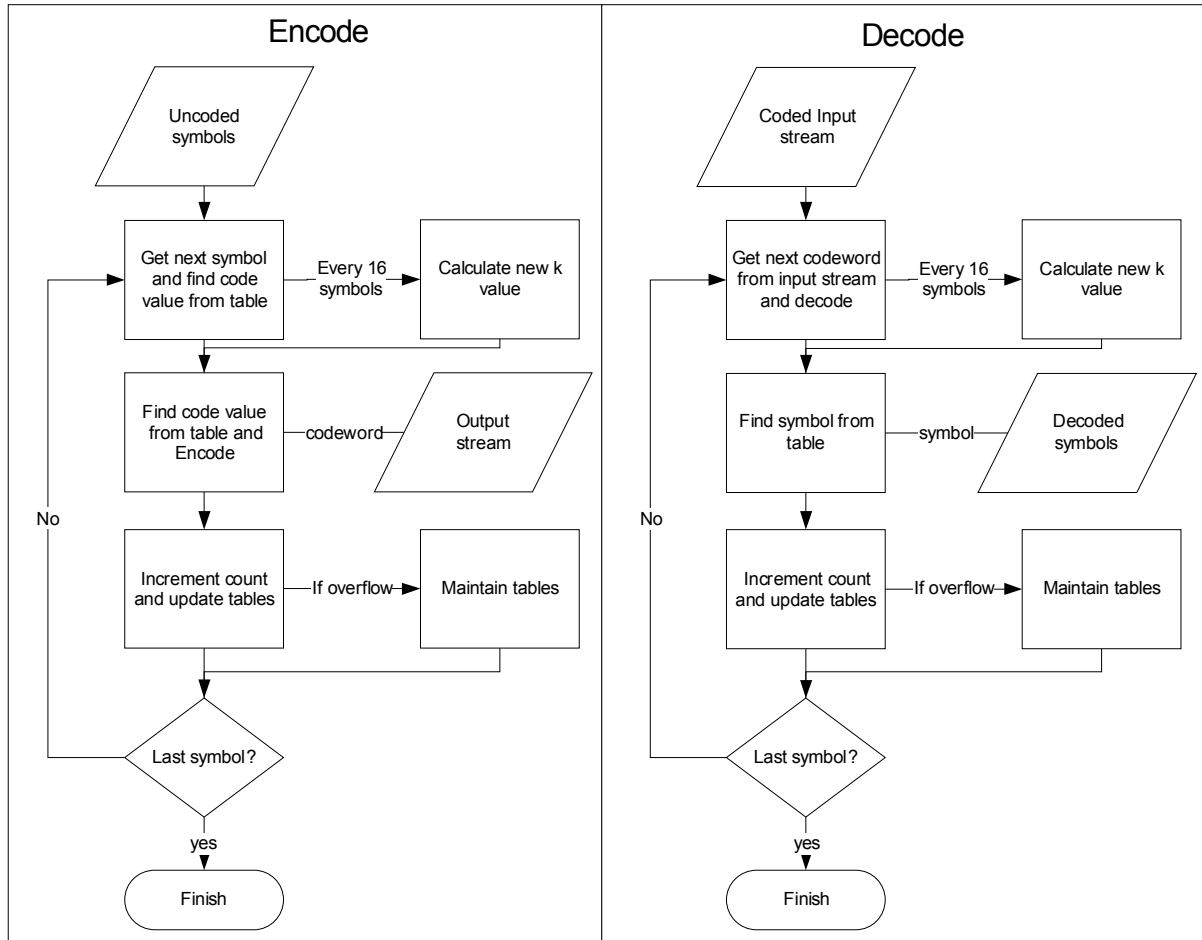


Figure 10, the Rice algorithm flowchart.

### 4.1.1 Calculating the K-value

From the description in section 1.2.1.1 it is clear that the optimal  $k$ -value is dependent on the entropy of the input stream. A mathematical analysis regarding the  $k$ -value was established by Dr. Penshu Yeh, R. F. Rice, and W. Miller at the NASA’s Goddard Space Flight Center (GSFC) in 1993 [15]. The analysis showed that Rice codes with different  $k$ -values basically were Huffman codes for different input streams with different geometrical probability distributions. From this reasoning, it was shown that the optimal  $k$ -value could be found using this equation:

$$\text{Eq. 6} \quad k = \log_2(n) \quad ;[15]$$

Where  $n$  is the average coded symbol value.

Many different methods have been developed to approach this equation. The equation by itself is not complex, but any hardware realization usually requires approximations. One method recommended by the CCSDS sub panel as the standard encoding method for lossless data compression in space applications, avoids any calculation of the optimal  $k$ -value by encoding the input stream in parallel with different  $k$ -values [16]. After encoding, the shortest encoded output stream is chosen for transmission. However, the parallel nature of this method is difficult to implement efficiently with one NanoRisc microprocessor. An approach to the actual equation is used in the JPEG-LS standard for image compression [17]. The method

calculates the  $k$ -value adaptively as it encodes the input data stream. From equation 6 it is clear that the most exhaustive part is averaging over a large amount of symbols. A way to compensate for the large averaging is calculating  $k$  for a given number of past symbols. The method used in JPEG-LS is shown by equation 7.

$$\text{Eq 7} \quad k = \min\{k' \mid 2^{k'} N \geq A\} \quad ;[17]$$

Where  $N$  is the count of symbols used for averaging, and  $A$  is the accumulated sum of values to be encoded.

Even though equation 7 is more hardware friendly than the original equation, it still requires a bit-shift, a comparison and an accumulation in every iteration. Equation 7 is also a bit more optimistic when calculating the  $k$ -value than the original equation (Figure 11). The method used to adaptively calculate the  $k$ -value in the implementation is an approach developed by the author [14]. This approach is a sort of middle course between the CCSDS recommendation and the JPEG-LS method. The method uses a fixed symbol count for averaging which can be described by  $2^b$ , where  $b$  is a positive integer. An approximated averaging of the accumulated symbol values is then made by  $b$  right shifts. The  $k$ -value is further approximated by finding the position of the msb in the shifted value. Figure 11 shows the calculated  $k$ -value using the original equation, the JPEG-LS method and the alternative approach. The calculations are done using a symbol count of 16, and accumulated symbol values from 0 to 240 ( $16 \cdot 15$ ). In the implementation, the  $k$ -value is calculated for every 16 symbol and the initial value is 2.

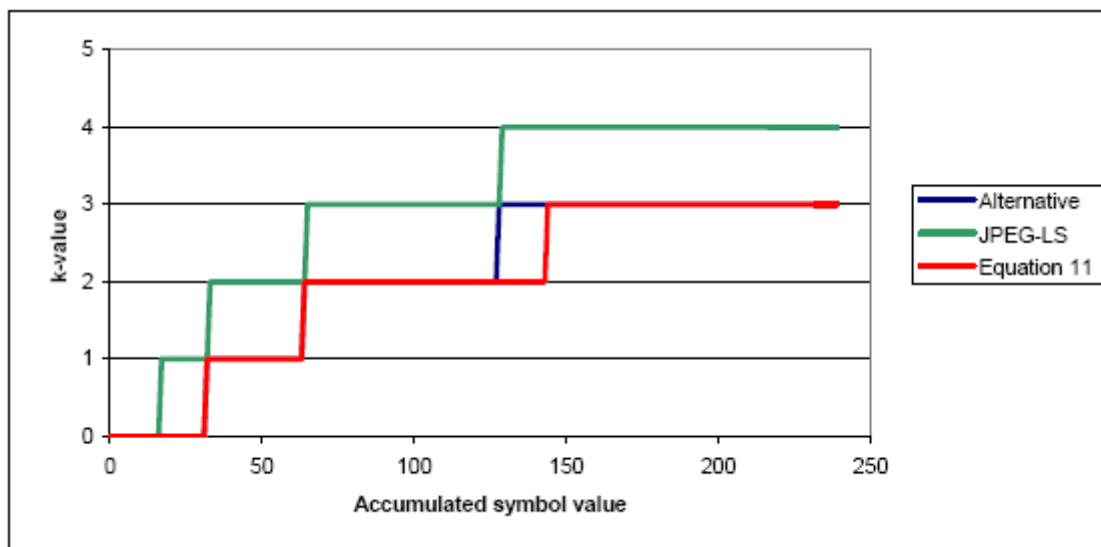


Figure 11, calculation of the  $k$ -value using equation 6, the JPEG-LS method and the alternative approach. (symbol count is 16) [14].

#### 4.1.2 Modeling Stage

Since Rice coding makes codewords directly from the value to encode, it is usually implemented together with a prediction stage. A prediction stage may be described as a digital high pass filter. This method does not require a large supporting data structure, but it requires a source that produces samples that results in small values after the prediction stage. This is

typical for e.g. audio and sensor data. However, in order to make the modeling more general, sorted tables are used.

The table is sorted with descending frequency count. When a symbol has been encoded or decoded, it gets its frequency count incremented. If another symbol value with lower frequency count is placed higher in the table, they will swap places. The symbols placement in the table (index) becomes the code value that is Rice coded. If e.g. symbol 6 is found in index 4, the encoder will encode the value 4. When this code value is decoded, and the decoder has sorted its table in the same way as the encoder, the decoder will find symbol 6 in index 4.

There are many ways of implementing this method, but careful considerations of memory use and run time should be made. The most straightforward approach is to implement one table with symbol values and their frequency count. This implementation is memory efficient, but exhaustive searches for symbol values and frequency counts would result in long runtime. The implemented method use two tables; the “symbol index table” and the “code index table”. An example of these two tables is shown in Table 6. The “symbol index table” is sorted on descending symbol value, while the “code index table” is sorted in descending frequency count. When encoding, the code value is found by using the symbol value as index to the “symbol index table”. After encoding the code value is used as index to the sorted “code index table”, and the frequency count for the encoded symbol is incremented in both tables. Sorting is then performed by comparing frequency counts in the ”code index table”. Since frequency counts are only incremented and the table is sorted for each symbol encoded, the average search for higher frequency counts tends to be short. The search is stopped when a higher or equal frequency count is found, and a swap is performed with the previous index. A swap is of course not needed if the previous index belongs to the current incremented symbol. If a swap is needed, the two rows in the code entry table and the code values in the symbol entry table are swapped.

Index	Symbol Index Table		Code Index Table	
	Frequency Count	Code Value	Frequency Count	Symbol Value
0	13	3	32	5
1	13	4	25	3
2	19	2	19	2
3	25	1	13	0
4	13	5	13	1
5	32	0	13	4
6	4	7	10	8
7	2	8	4	6
8	10	6	2	7

**Table 6, Table example for the implementation of the sorting method**

If e.g. symbol 4 is incremented to 14 in Table 6, the code value in index 4 is used as index to the “code index table”. In this table, the search will begin at index 5 and stop at index 2. From index 3, symbol 0 is found. The swap will hence be performed on symbol 4 and 0. First the row at index 5 and 3 in the “code index table” are swapped, and then the code value in index 4 and 0 in the “symbol index table” are swapped. The symbol size in the implementation is 8 bits and frequency counts are also 8 bits. Total memory use becomes  $2 \cdot 2^8 \cdot 8 = 8192$  bits for the two tables.

## 4.2 Implementation of Huffman Coding

Many implementations of Huffman coding use a static Huffman tree in the encoder and decoder. This is because it requires significant processor power to build and maintain an adaptive Huffman tree. However, to explore the capabilities of the NanoRisc processor, an adaptive Huffman algorithm is chosen for implementation. Figure 12 shows the flowchart of the implemented method. The next subsections will describe the adaptive tree structure and how this is implemented.

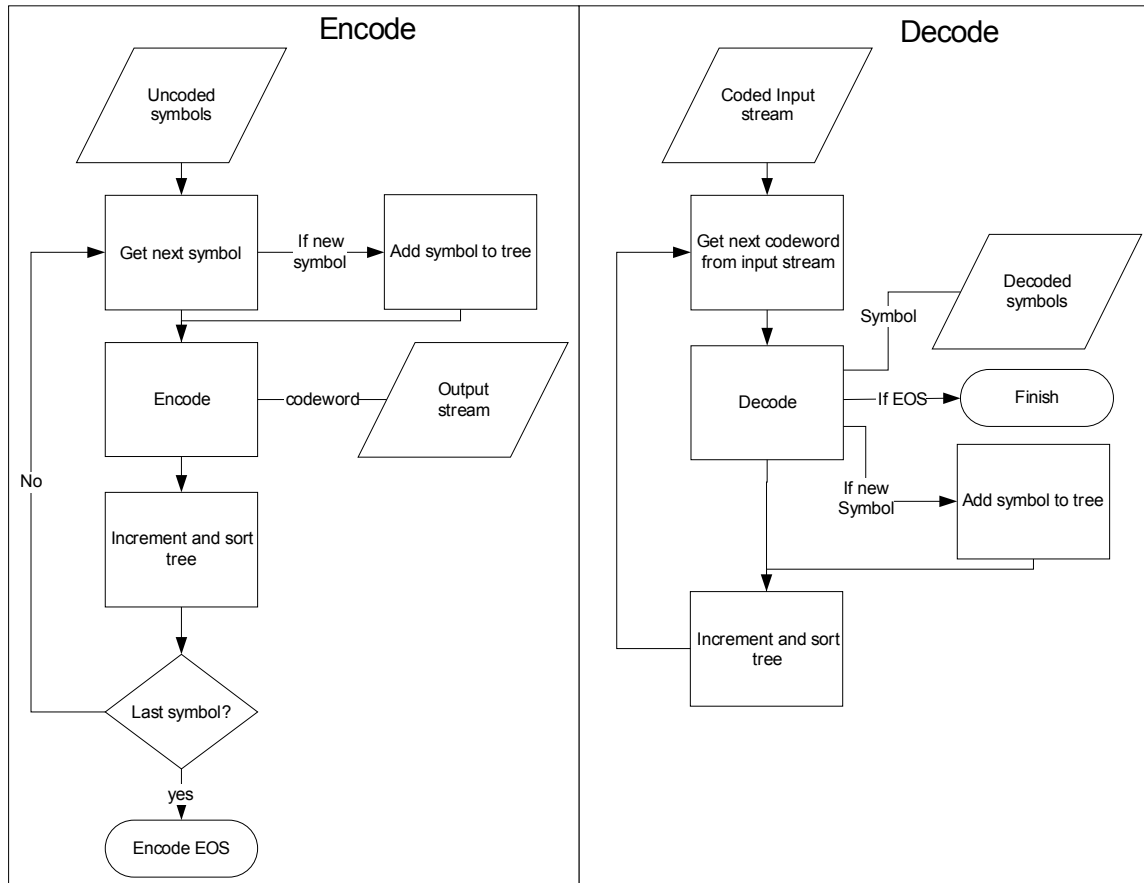


Figure 12, the Huffman algorithm flowchart.

### 4.2.1 Updating the Huffman tree

Effective algorithms for constructing Huffman trees are usually fairly simple (section 1.2.1.2), but it is not something that should be done after each symbol has been encoded or decoded. This would slow down the process significantly. Because of this, a method to take an existing Huffman tree and adaptively modify it to account for every symbol in the input stream must be used. A Huffman tree is a binary tree that has a weight assigned to every node, whether an internal node or a leaf node. Each node (except for the root) has a sibling that shares the nodes parent. In order to be a Huffman tree, the tree structure must exhibit something called the sibling property. A tree exhibits sibling property if the nodes can be listed in order of increasing weight, and if every node appears adjacent to its sibling in the list. Figure 13 shows a Huffman tree where every node have a weight  $W$  and a number  $\#$  indicating the nodes order in a sorted list. This arrangement shows that the tree exhibits the sibling property because every node is adjacent to its sibling in a sorted list.

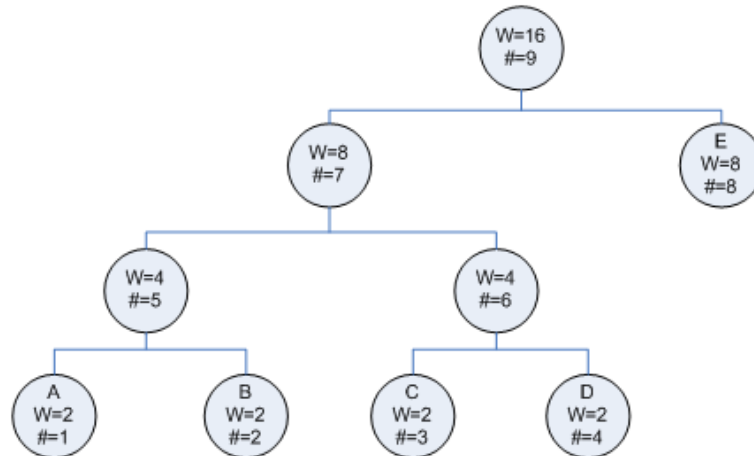


Figure 13, Huffman tree showing the sibling property.

Updating of a Huffman tree is done by incrementing the weight of every node when traversing the tree from the root node representing the current symbol to the root. Maintaining the sibling property when a tree is updated ensures that it is a Huffman tree. Since the tree is incremented for every symbol the parent nodes will always have the accumulated weight of its children.

The increment operation may result in a violation in the sibling property. When this happens, the tree must be rearranged. If a violation occurs, the node being incremented must swap place with a node higher in the sorted list. Swapping an internal node will affect the whole branch. If symbol *A* is encoded or decoded in the Huffman tree in Figure 13, leaf node *A* will first be incremented to  $W=3$ . This increment will violate the sibling property since other nodes higher in the list have less weight. The first swap must be between node #1 and #4 in the list. After the nodes have been swapped, the increment operation must continue with node #6 and #7. When node #7 is incremented it must be swapped with node #8. The resulting Huffman tree with updated weights and order is shown in Figure 14.

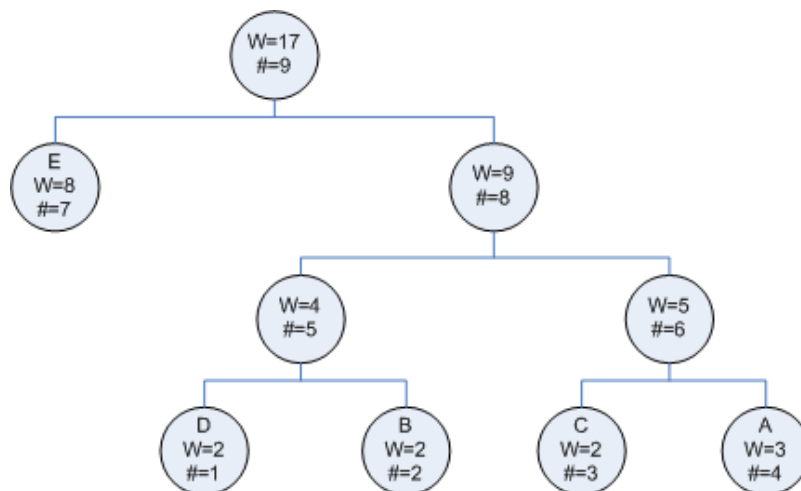


Figure 14, updated Huffman tree.

### 4.2.2 Implementing the Tree Data Structure

The tree is implemented with nodes that each has pointers to its parent and/or children. The use of pointers enables the nodes in the tree to be ordered in a sorted list, since they do not have pre defined memory locations. Traversing the tree is done through pointers, and searching for higher sorted nodes is done by calculating indexes in the sorted list. When choosing this data structure in an environment with strict memory requirements, it is important to hold the memory occupied by a node as small as possible. If the input stream consists of an alphabet of  $k$  symbols, a full Huffman tree requires  $k-1$  internal nodes and  $k$  leaf nodes. For 8 bits symbols, a full Huffman tree will thus require 511 nodes. If any node should be able to have a pointer to any other node in the Huffman tree, the pointer must be at least 9 bits. An internal node must therefore have  $3 \times 9$  bits reserved for pointers, and a leaf node must have one 9 bits pointer to its parent. A leaf node must also have 8 bits for its symbol value. In addition, all nodes must have bits for weight and an indicator telling what kind of node it is (internal, leaf, left or right child).

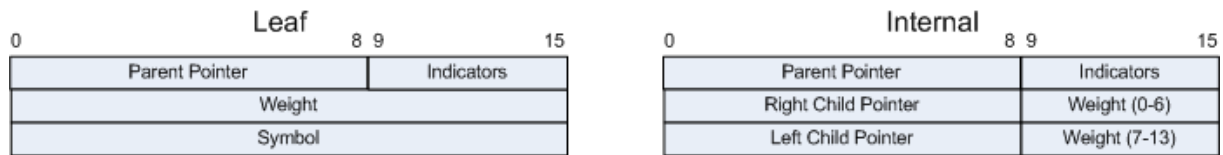


Figure 15, node memory structure.

Figure 15 shows how the node data structure is allocated in the memory. The weight is restricted to 14 bits which makes the maximum number of symbols encoded or decoded before truncating 16384. All nodes allocate 3 fields of 16 bits in the memory. When a node in memory location has its first field at memory address  $a$ , the next higher sorted node is at memory address  $a+3$ . Since there is no way of telling where in the memory a leaf node is situated, the encoder must also have a table of memory addresses to every leaf node. Thus, the static memory allocation required for the encoder is 3578 bytes, while the decoder requires 3066 bytes.

There are three main ways of initiating the Huffman tree when encoding and decoding. The first method is starting with a fully balanced Huffman tree with all symbols, the second is using a predefined weighted Huffman tree. The third and implemented method starts with a Huffman tree of two leaf nodes and the parent (root) node. Figure 16 shows how this tree looks like. When a never before seen symbol is encountered in the input stream at the encoder, it transmits the code found by traversing the tree from the root to the *NS* (New Symbol) leaf node. After this, it can send the new symbol without encoding it. The new symbol must then be inserted as a new leaf node in the Huffman tree. The *EOS* (End Of Stream) leaf is used at the end of the stream. By applying this method, the tree will not occupy more memory than needed and symbols may acquire short codewords earlier in the encoding process. Since the incrementing operation is proportional to the average code length, this method will in most cases require less processing power in the beginning of the stream.



Figure 16, initial Huffman tree.



### 4.3 Implementation of LZ77

The LZ77 compression method was first described in 1977, and during the 1980s and 1990s it was improved several times. Much effort was put into minimizing search time and coding indexes with variable length codes. One of the improved methods is Deflate. This has become a popular compression method that was originally used in the well-known Zip and Gzip software. The method has since been adopted by many applications such as the HTTP protocol, the PNG graphics file format, and Adobe's PDF (Portable Document File). The compression method implemented in the NanoRisc processor is a very simplified version of Deflate developed by the author. This section will first describe the simplified Deflate compression method before describing details of the implemented data structure.

#### 4.3.1 Simplified Deflate

The Deflate compression method is described in detail in [18]. Deflate is a variant of the LZ77 compression method combined with Huffman codes. The original LZ77 method outputs an index to a match and the next symbol in the look-ahead buffer that did not match. By always outputting two components (index and next symbol), the performance of LZ77 is reduced. The Deflate variant eliminates one of the components. If a match was found it outputs the index, or if a match was not found it outputs the next symbol in the look-ahead buffer. Thus, the output stream consists of two types of entities; symbols and indexes. In order to separate these two entities, they are Huffman coded using a static code table. The codewords are prefix codes such that the decoder knows when it reads a symbol or length. If it reads a length, it assumes that a distance will follow.

The reason for making a simplified version of Deflate is mainly to ease memory requirements by not storing large Huffman code tables, and utilizing the fact that most wireless transmission is based on small data packets. The minimum length of a match in Deflate is 3, and the maximum is 257. The maximum distance is 32768. It is evident that a distance of 32768 is not needed in most wireless applications, and a length of 257 would span over most packet lengths. The ZigBee™ standard uses a maximum packet size of 128 byte with 104 bytes of payload [19], and the Bluetooth® standard use a maximum packet size of 359bytes with about 340bytes of payload [19]. In the simplified implementation of Deflate, the maximum length is 150 and the minimum length is 3. The search buffer should cover a number of packets to achieve high compression ratios, but it requires memory to store used packets. A maximum search buffer of 1279 is used in the implemented version. Instead of having prefix Huffman codes, a fixed 3 bits prefix of what to follow in the data stream is used. Each prefix is listed in Table 7. A prefix precedes every symbol or index, except for the *EOS* prefix that follows the last symbol or index in the encoded data stream.

Indicator	Meaning
000	EOS (End of Stream)
001	8bits symbol
010	2bits length, 8bits distance (length:3-6, distance:0-255)
011	4bits length, 8bits distance (length:7-22, distance:0-255)
100	7bits length, 8bits distance (length:23-150, distance:0-255)
101	2bits length, 10bits distance (length:3-6, distance:256-1279)
110	4bits length, 10bits distance (length:7-22, distance:256-1279)
111	7bits length, 10bits distance (length:23-150, distance:256-1279)

Table 7, indicators for simplified Deflate

Figure 17 shows a flowchart of the simplified Deflate compression algorithm. The encoding and decoding is like described above. The next subsections will describe how the encoder search for matches and how different data structures make this process as efficient as possible.

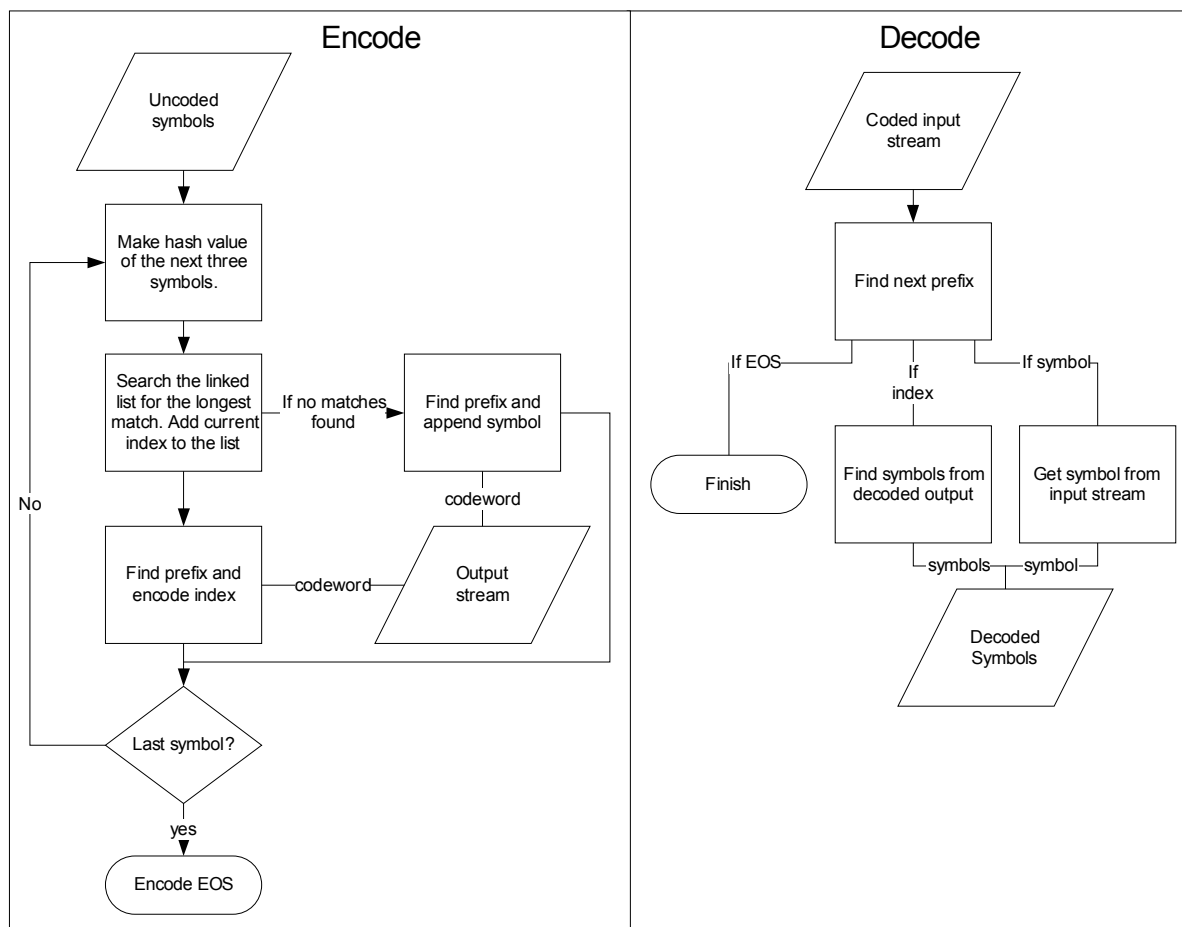


Figure 17, the simplified Deflate flow chart.

### 4.3.2 Searching for Matches

The most time consuming task in the LZ77 coding method is searching for the longest match. This can be shown from the theoretical description of LZ77 in section 1.2.1.3. A straightforward approach could be to exhaustively search for a match by reading the whole search buffer for every new symbol in the input data stream. This approach would require very little memory allocation, but it would have extensive memory access and long run time. This is a common problem for dictionary methods, and is mainly the reason for their highly asymmetrical behavioral. Repetitive sequences of symbols must be replaced by a codeword, and if this is to be done efficiently, sequences must be found through some sort of dictionary data structure. A dictionary data structure, from now on also referred to as the dictionary, helps finding an index for a match in the search buffer for the next sequence to be encoded in the look-ahead buffer. There are many ways of implementing dictionaries, and the throughput and memory usage of dictionary methods are very much dependent on how this is done.

An often used dictionary is the trie data structure. A trie is essentially a tree where each edge is labeled with a symbol, and sibling edges have different symbols. Thus, every sequence in the search buffer can be found by traversing the trie from the root to a leaf. Even though tries

are often used to hold dictionaries, this kind of data structures is not used in the simplified Deflate implementation. The main reason is the heavy use of a tree data structure in the implementation of the adaptive Huffman compression method (section 4.2). Another method of holding the dictionary is therefore preferred.

Hash tables are also used to help the search for sequences in dictionary compression methods, and this is the method chosen for implementation. Hashing is performed by a hash function, and is usually used to associate keys with values. The most common use is hash tables where the hash function transforms a key into an index in a hash table. The hash table is then used to locate the desired value. A good hash functions main features is to produce a hash value of fixed length from an input key, and where two different keys are unlikely to produce the same hash value. If two hash values are different, the two input keys must have been different in some way. This property is a consequence of hash functions being deterministic, mathematical functions. Nevertheless, the feature of producing values of fixed length from a key of arbitrary length implies that different keys may produce the same hash value.

There are many types of hash functions available, and it is important to spend some time choosing a hash function. A good hash function has evenly distributed hash values, and it may be considered as a random number generator. A good hash function will thus minimize the probability of different symbol values generating equal hash values (this is often referred to as collisions). In [20] different hash functions are compared. This paper concludes that checksums generated from standard CRC polynomials provided an excellent hash function. CRC hash values are the remainder of a division based on polynomial arithmetics in a finite field [21]. This hash function is commonly used in packet network traffic or data files to detect errors after transmission or storage. Normally, the result of a CRC hash function is referred to as a checksum, but this is not an accurate term since a checksum would be calculated through addition and not through division, as is the case for a CRC hash function. In this thesis, the result of the CRC function will be referred to as a hash value.

### **4.3.3 The Dictionary Data Structure**

The hash function implemented is based on the CRC8 CCITT standard polynomial [22]. There are several ways of implementing this hash function. The simplest method uses bit shifts and “xor” operations in a Linear Feedback Shift Register (LFSR). This method is simple and easy to implement, but it requires one calculation for every bit. To speed up this process, the implemented method uses look up tables for calculating the hash value for every byte. The look up table contains pre calculated hash values for every byte value, and must be stored in memory. Each pre calculated hash value is 8 bits, so the table requires 256 bytes of memory.

To speed up the search process in the encoding algorithm, every three consecutive symbols in the stream are hashed to an 8 bits value. The 8 bits hash value is used as an index to a hash table. In every index there is a pointer to a linked list. The linked lists contain the pointer to every three-symbol sequence in the search buffer that produced the same hash value. The search is then restricted to finding the longest match starting from any of these pointers. Figure 18 shows how the linked lists are linked to the hash table and how they expand.

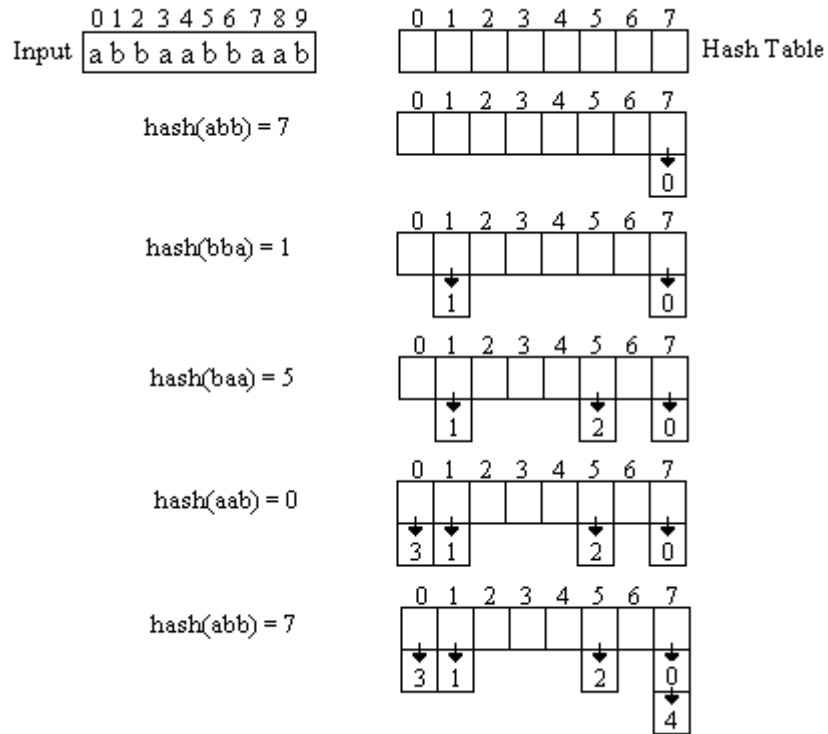


Figure 18, hash table with linked lists

The implemented linked list data structure does not have a limit for its expansion. This is not advised for large search buffers or when dealing with strict memory requirements. A sort of maintenance function should be considered for other implementations. If this is not implemented, the search for matches will be very time consuming in a data stream with many collisions. The memory use for this method is the look up table used in the hash function, the hash table and the linked list. The hash table is 256\*1byte, and the hash table with the linked list could potentially grow as large as 256\*2bytes + 1000\*4bytes = 4512 bytes without a maintenance function. This is in most cases unacceptable for embedded solutions.

## 5 Enhancements of Existing Tools

To measure the current capabilities of the NanoRisc microprocessor and the effect of improvements made, it was necessary to add some functionality to the existing software tools. This section will describe how the profiling tool is added, and how the assembler and ISS are altered in order to simulate the proposed HW enhancements. Both the enhanced NanoRisc assembler and ISS are found as executables in appendix E.

### 5.1 Profiling

After the compression algorithms are implemented in the NanoRisc assembly language, it is necessary to profile the resource use when the source code is processed by a NanoRisc microprocessor. A good profiling tool is vital for exploring where processing power is consumed in the algorithms. Estimations are always possible to do from studying technical descriptions or source code, but the real picture is shown through profiling with realistic input data. Since all compression algorithms are heavily dependent on the source data, profiling could visualize unexpected bottlenecks and show where the best improvement potential is. Nevertheless, estimations must be done to evaluate where in the algorithm attention and profiling should be focused.

When these elements are found, there must be some way of telling the ISS where in the code profiling is wanted. The easiest way to implement this feature would be to use the GUI. The user could mark areas of code for profiling in the source code window of the ISS, but since the code in this window may originate from multiple included files and do not show labels and aliases, this approach would not be easy to use. The best way for the user would be to specify areas for profiling in the original assembly source code. This implies that enhancements of both the NanoRisc assembler and the NanoRisc ISS are necessary.

#### 5.1.1 Profiling With the Assembler

The profile enhancement of the assembler enables the user to specify which areas of the source code that profiling is wanted. Profile labels define the areas, and the syntax is shown in Figure 19.

```
.start_profile identifier [category]
.end_profile identifier
```

Figure 19, profile labels syntax.

The “.start\_profile” label indicates that the start of a profile area begins at next instruction line, and the “.end\_profile” label indicates that the end of a profile area is at the prior instruction line. A pair of “.start\_profile” and “.end\_profile” labels with the same unique identifier defines a specific profile area. Several profile areas may overlap, as long as they don’t start or end at the same instruction line. When an instruction line is being executed, the resource usage is credited to the profile area that is defined for that instruction line. If profile areas overlap, the same resource usage will be credited to all overlapping areas. Every call or branch within a profile area will be credited, and this makes placing “.end\_profile” labels critical. A branch or call inside a profile area could result in undesirable effects such as making the end of the profile area unreachable.

The category part of the “.start\_profile” label is optional. If a category is specified for a profile area, all resource use credited the profile area is also added to the category. Multiple profile areas may use the same category, and the category will hold the sum of the resources used by the profile areas. This is convenient when elements of the same kind are situated in different locations in the source code.

### **5.1.1.1 Implementation**

The enhanced assembler is made by altering or adding to the original source code. The lexer is altered to recognize the “.start\_profile” and “.end\_profile” tokens, and the parser is altered to identify profile labels with identifier and category arguments. Every time the assembly program reads a profile label, it will create a profile object and link it to the correct instruction line. The profile objects are made from a profile class, and a linked list data structure is made to organize these objects. Both the profile class and data structure is added to the original C++ source code. It is a separate linked list data structure for “.end\_profile” objects and “.start\_profile” objects. When the assembler has read the assembly source code, it resolves these two data structures to ensure that every “.start\_profile” object has a corresponding “.end\_profile” object. The output from the original assembler is a line for every instruction word in ASCII hex format followed by the word address of the instruction, the originating filename and line number. The new profile enhancement will output three additional information fields for every instruction line; a number indicating if the instruction is the start or end of a profile area, the identifier of the profile area and the category.

### **5.1.2 Profiling In the Simulator**

As mentioned, the ISS simulates the behavior of the NanoRisc at a cycle accurate level, it is therefore convenient to profile by measuring the amount of clock cycles used within the profiling areas defined in the source code. Another important measure is the amount of memory access, and this is also feasible by enhancing the original ISS. The new GUI of the enhanced ISS is shown in Figure 20. From the figure, one can see the resemblance from the original GUI in Figure 9. Altering the original GUI as little as possible will help users of the original ISS using the new profiling tool. The only profile information visible in the new GUI is the “Profile” button, the “Prof. Identifier” column in the program window, and the count of memory load and store operations. It will only show RAM access since the NanoRisc will fetch a new instruction from the ROM almost every clock cycle.

To do profiling in the enhanced NanoRisc ISS, it must be loaded with a file containing the output from the enhanced NanoRisc assembler described in section 5.1.1. If the source code is made with profiling labels, their identifier will be shown in the “Prof. Identifier” column on the correct instruction line. Instruction lines that correspond to a “.start\_profile” label will have the prefix “S” before its identifier, and instruction lines corresponding to an “.end\_profile” label will have “E” as prefix. The two counters showing memory load and store operations will update for every step in the simulation. They will not show the amount of clock cycles used for load and store operations, but the amount of times the memory is accessed during the simulation. In other words, the load and store counters profile the memory access at an instruction accurate level.

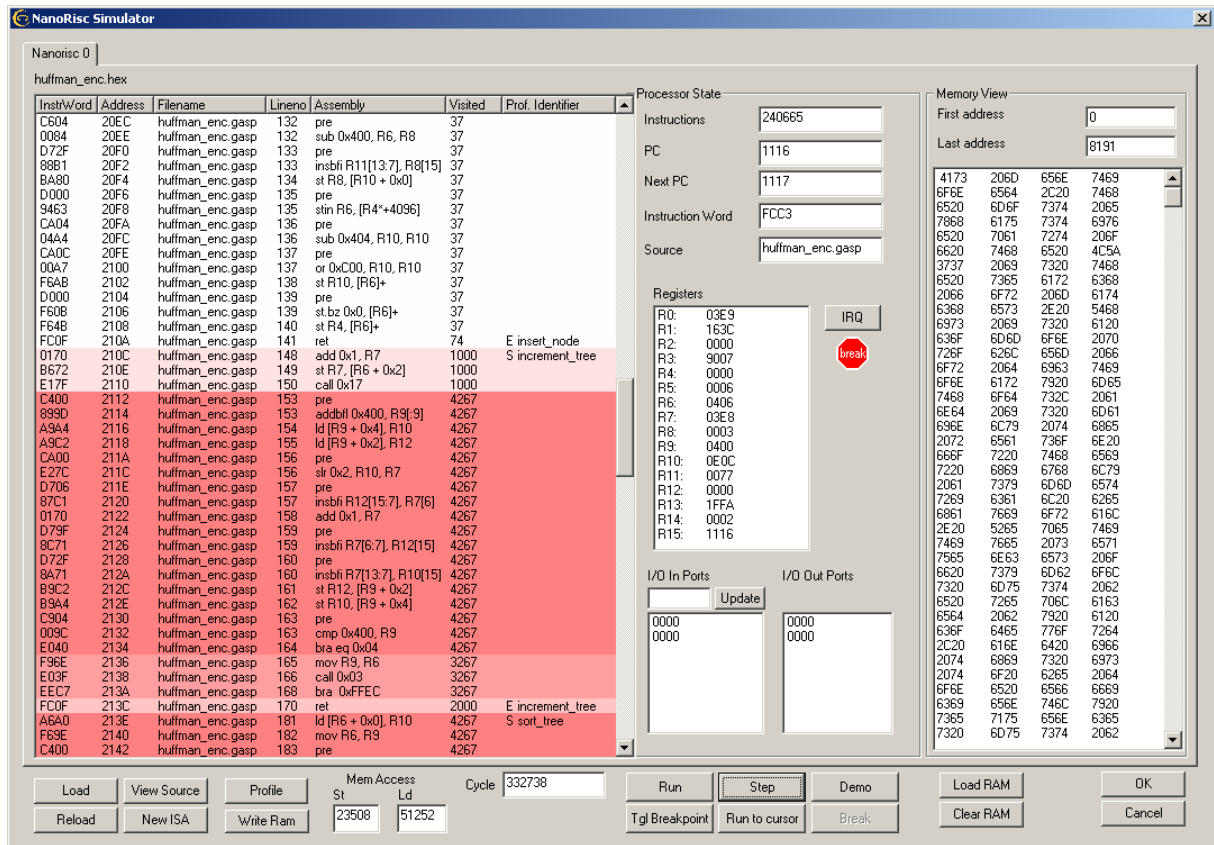


Figure 20, enhanced NanoRisc ISS GUI.

The profile window shown in Figure 21 will appear if the “Profile” button is pushed. Profile areas defined in the assembly source code are shown in two tables, and the categories are shown in the third. This window can be shown by pushing the “Profile” button any time during the simulation, and it will show the profile information gathered so far. Profile areas listed in the “Active Identifiers” table are areas that embraced the instruction line executed at the time the “Profile” button was pushed. The “Passive Identifiers” table show profile areas that are not affected by the current instruction line. Both the “Active Identifiers” table and the “Passive Identifiers” table have four columns; “Identifier” is of course the name of the profile area, “Cycles” is the amount of cycles credited the profile area, “Calls” is the amount of times the instruction line with the profile areas “.start\_profile” label have been executed, and “Per” is the percentage of the total amount of clock cycles which are credited that profile area.

The “Categories” table show profile information for the categories defined in the assembly source code. It is not visible in the profile window which profile areas belong to which category, so profile areas should be defined with identifiers that give a hint of which category they belong to. If the profile area is not defined with a category, a category with the same identifier as the profile area is listed in the table. All tables are ordered by descending cycle count.

The screenshot shows a 'Dialog' window with three columns: Active Identifiers, Passive Identifiers, and Categories. Each column contains a table with columns for Identifier, Cycles, Calls, and Per. The 'Active Identifiers' table shows 'increment\_tree' as the most frequent with 48.21%. The 'Passive Identifiers' table shows 'insert\_node' as the most frequent with 17.26%. The 'Categories' table shows 'increment\_tree' as the most frequent with 48.21%.

Active Identifiers				Passive Identifiers				Categories		
Identifier	Cycles	Calls	Per	Identifier	Cycles	Calls	Per	Category	Cycles	Per
increment_tree	148	1	48.21%	insert_node	53	1	17.26%	increment_tree	148	48.21%
sort_tree	104	2	33.88%	encode	31	1	10.10%	sort_tree	104	33.88%
switch_nodes	34	1	11.07%	shift_ostream	14	2	4.56%	insert_node	53	17.26%
bfo_switch_nodes1	1	1	0.33%	bfo_sort_tree1	8	2	2.61%	bfo	44	14.33%
				sm_main_1	6	1	1.95%	switch_nodes	34	11.07%
				bfo_insert_node2	6	1	1.95%	encode	31	10.10%
				bfo_insert_node1	5	1	1.63%	streams	14	4.56%
				bfo_ext_count2	5	1	1.63%	shift_mul	13	4.23%
				bfo_ins_count1	5	1	1.63%	mem_switch	9	2.93%
				bfo_encode1	4	1	1.30%			
				sm_insert_node2	4	1	1.30%			
				bfo_increment_tree1	4	1	1.30%			
				ms_insert_node1	3	1	0.98%			
				sm_insert_node1	3	1	0.98%			
				bfo_ins_count2	3	1	0.98%			
				bfo_ins_count3	3	1	0.98%			
				ms_switch_nodes2	3	1	0.98%			
				ms_switch_nodes1	3	1	0.98%			
				bfo_switch_nodes2	0	0	0.00%			
				bfo_switch_nodes3	0	0	0.00%			
				ms_switch_nodes4	0	0	0.00%			
				bfo_switch_nodes4	0	0	0.00%			

Figure 21, Profile window

### 5.1.2.1 Implementation

The enhanced NanoRisc ISS is made by altering or adding to the original source code. When the ISS is loaded with a file made from the enhanced NanoRisc assembler, it will look for indicators at every instruction line that tells if the line is the start of a profile area, the end of a profile area, or neither. If the instruction line is the start of a profile area, the simulator will create a profile object. Profile objects are made from a profile class, and are held in a linked list data structure. If the profile area is defined with a category, a category object will be created from a category class if it is not already created by earlier profile areas. If the profile area is not defined with a category, a category object will be created with the profile areas identifier as name. All profile objects are linked to their category objects.

The profile objects are organized in two separate linked list data structures during simulation. One linked list holds the profile objects affected by the current instruction line executed in the simulation, and the other list holds the profile objects not being affected. These lists are called the active and passive list. If the instruction line currently being executed is the start of a profile area, the profile object corresponding to that profile area will be removed from the passive list and added to the active list. When an object is moved in this direction, it will also have its “call” count incremented. If the instruction line is the end of a profile area, the corresponding object is moved from the active list back to the passive list. For every clock cycle of the simulation, all profile objects in the active list will have their clock cycle count incremented. And every time a profile object is incremented, it will also increment the clock cycle count of its linked category object. So if e.g. a category has two overlapping profile areas, it will be incremented twice when both profile areas are “active”.

When the “Profile” button is pushed, the linked list data structures holding the active, passive and category objects are passed to the profile GUI object. The profile GUI class is added to the original C++ source code. Before the data is shown, all objects in their respective lists are sorted with descending cycle count.



## 5.2 Adding New Instructions

In addition to making changes in the hardware module, it is vital that the assembler is able to make code words for the added instructions. This makes it simpler to verify the correct behavior by using actual source code as input to the test bench. To profile and control the behavior, the simulator is also altered in order to simulate the behavior of the NanoRisc according to the changes made.

### 5.2.1 Adding New Instructions to the Assembler

The lexer is altered to recognize the new instruction mnemonics and their operators. The code words bit pattern must also be defined, and the code generator function must be able to insert parameters in the defined fields. In addition, it must be able to detect when the new instructions need “pre” instructions and insert these. The operation of adding new instructions to the assembler is straightforward for readers who are familiar with lexer and parser generators [12, 13].

### 5.2.2 Altering the Behavior of the Simulator

In order to simulate the algorithms with added instructions, the behavior of the simulator must be altered. The simulator must recognize the code words and be able to interpret these into the desired behavior. The first task is to alter the disassembler that interprets the instruction into more human readable assembly language. In order to do this it must be able to retrieve parameters from the instruction itself or the “pre” instruction. After this, new functions must be added in order to simulate the behavior and show the correct response to the user in the GUI. The new instructions and behavior is added as options to the original simulator. A new button (“NewISA”) is added so the user may turn on or off these changes. When this button is clicked, a new dialog window appears so the user can choose the preferred collection of new behavior. Figure 22 shows this dialog window. This is done in order to do controlled simulation with the desired behavior. It is also useful if some of the enhancements are rejected, or if it is desirable to implement different versions of the NanoRisc with only a collection of the enhancements.

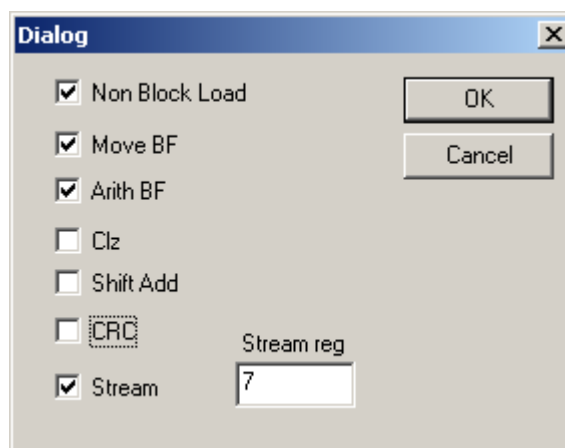


Figure 22, new ISA dialog window.

## 6 Finding Enhancements for the NanoRisc Processor

This chapter will explore the capabilities of the NanoRisc microprocessor when processing the implemented algorithms. Based on these profiling results, new instructions and behavior are proposed and implemented in the simulator. Finding improvements are done in two stages, and after each stage proposals are made to increase performance. In the first stage, blocks of codes that emulate possible new instructions are identified and profiled. This is referred to as instruction level profiling. Instruction level profiling is difficult. Small blocks of instructions that may increase performance by replacing them with new instructions must be identified. In the second stage, the blocks of codes are replaced by proposed instructions, and the source codes are profiled once again to find improvements on an algorithmic level. Algorithmic level profiling is profiling major parts of the compression algorithms, e.g. decoding, encoding, updating data structures, etc. These are easy to identify and is often implemented as functions.

All algorithms are tested on four different data streams. A software program is developed to make three of the data streams. This program creates random numbers from different probability distributions. The source code for this program is found in appendix E. The last stream is made from a section of text in this thesis. The streams generated from the software program have Gamma, Poission and exponential probability distributions. All data streams are 1000 byte, and their symbol distributions are found in appendix B.

Improving throughput of the algorithms may be done in different ways, e.g. parallelization, pipelining, instruction level accelerations, hardware accelerators etc. Parallelization of the NanoRisc would require a redefinition of the NanoRisc architecture and behaviour. The compact original NanoRisc instruction set is not big enough to handle parallel processes, and a VLIW type instruction set must be considered. A parallelization of the NanoRisc will not be considered since this will alter the basic principles of the processor, and it would be hard to support programs made with the original instruction set. Pipelining will improve instructions that use several clock cycles, but since the only instructions that use more than one clock cycle in the original instruction set are load operations, it will have minimal effect on the processing power of NanoRisc. The two main enhancements considered are instruction level acceleration and hardware accelerators.

Instruction level acceleration is about finding ways to add instructions and architectural features that is beneficial for the algorithms, and hopefully also beneficial to other tasks that the processor might be doing. This kind of acceleration is tied very close to the processor core, but it is less intrusive than parallelization and pipelining. Instructions and architectural changes may be added while old programs are still supported. Hardware accelerators are dedicated HW that is designed to do a large portion of a specific task. They are usually detached from the processor core, but still tied closely through memory or buses for low latency communication. Since HW accelerators often do a large portion of a specific task and are heavily optimized, they are usually not very useful for other tasks that the processor might deal with. Other considerations when it comes to HW accelerators are power and area. When accelerators take on a big portion of the total load, it is usually exposed in the power and area consumed. Adding all too specific accelerators will also make the design closer to an ASIC.

## 6.1 Instruction Level Profiling

Since all of the implemented algorithms use variable sized codes to achieve compression, instructions that are able to do bit field operations are something that may increase performance. Bit field operations are operations that use variable or a fixed sized part of a register as input. Operations may be anything from moving or inserting bit fields to arithmetic or logical operations. Examples of blocks of instructions from the source codes that do bit field operations are found in Figure 23 and Figure 24.

```
and    0xFF80, R11    ;Clear 9 msb's
sll    2, R11        ;Shift to wanted position
and    0x01FF, R10   ;Clear 9 lsb's
or     R11, R10     ;Insert bit field
```

Figure 23, example insertion of a bit field (6 clock cycles).

```
and    0x00FF, R5    ;Clear 8 msb's
add    0x1, R5       ;Inc freq
```

Figure 24, example addition on a bit field (3 clock cycles).

Since many of these blocks use several clock cycles when processed, one can expect that new instructions may be implemented to reduce the amount of clock cycles. Many existing processors have bit field instructions. As mentioned in chapter 2, both Phillips TriMedia and ARM – DSP have bit field instructions for extracting, inserting and bit-packing. In addition to saving clock cycles, bit field operations may also save RAM access. When instructions are able to work with bit fields instead of whole registers, the information held in the general-purpose registers in the processor can be packed more efficiently and RAM usage may decrease. In addition to bit field operations there are also other instructions that may help increase processing power. Such instructions may be shift-add instructions, detection of the position of the most or least significant bit in a register, instructions for making bit masks, etc. Shift-add instructions are common in most digital signal processor. Since the NanoRisc processor can address the memory at byte level, loading and storing words must be done with even memory addresses. Most shift-add operations are hence used to make memory addresses for table indexes, an example is shown in Figure 25. Intel 80x86, Motorola 680x0 and many ARM cores (to name a few) have instructions for finding leading or trailing ones or zeroes. Such an instruction is especially useful for the decoding stage in the Rice algorithm.

```
sll    1, R5          ;Shift index
st.b   R6, [R5 + 0x0020] ;store index information
```

Figure 25, example shift add operation for index storage (3 clock cycles).

The new potential instructions identified in the assembly source code are gathered in three main categories; Bit Field Operations, Shift-Add Operations and Count Leading Zeroes. The rest of this section will show the result from profiling these operations. Detailed profiling results are found in appendix C.

### 6.1.1 Profiling Rice

Table 8 shows the results from the Rice algorithm. The table shows the total amount of cycles used by the operations per total amount of cycles used by the algorithm. In addition to the Bit Field, Shift-Add operations and the Count Leading Zeroes (CLZ) operation, the RAM access is shown as load/store operations. The cycles/call column shows how many clock cycles the profiling area use on average every time it is moved to the active list. Even though the basic algorithm is symmetrical, the decoding use more clock cycles than the encoding. This is mostly because the use of the CLZ operation in the decoder. The CLZ operation is used when the zeroes in the rice codeword is counted. This is done through shifting the codeword until a one is detected. The operation could be done more effectively by a more recursive operation or by using look up tables, but this would increase the program size and use more memory. The CLZ operation is also used when calculating the k-value.

	Encode		Decode	
	Cycles/Tot.Cycles	Cycles/Call	Cycles/Tot.Cycles	Cycles/Call
<b>Distribution:</b>	<b>Poisson (Tot. Cycles: 93275)</b>		<b>Poisson (Tot. Cycles: 112490)</b>	
Bit Field Operations	9.60%	4.50	8.50%	4.47
ShiftAdd Operations	9.70%	3.00	6.60%	3.47
Count Leading Zeroes	3.40%	51.68	15.60%	16.56
Store Operations	8.60%	1.00	7.90%	1.00
Load Operations	8.90%	2.00	8.10%	2.00
<b>Distribution:</b>	<b>Gamma (Tot. Cycles: 91322)</b>		<b>Gamma(Tot.Cycles: 110930)</b>	
Bit Field Operations	9.80%	4.50	8.50%	4.48
ShiftAdd Operations	9.60%	3.00	6.60%	3.48
Count Leading Zeroes	3.60%	52.84	16.20%	16.95
Store Operations	8.60%	1.00	7.80%	1.00
Load Operations	8.90%	2.00	8.10%	2.00
<b>Distribution:</b>	<b>Exponential (Tot.Cycles: 78729)</b>		<b>Exponential (Tot.Cycles: 97912)</b>	
Bit Field Operations	11.50%	4.50	9.30%	4.49
ShiftAdd Operations	9.20%	3.00	7.30%	3.54
Count Leading Zeroes	0.60%	7.00	13.50%	12.48
Store Operations	9.20%	1.00	8.30%	1.00
Load Operations	9.50%	2.00	8.40%	2.00
<b>Distribution:</b>	<b>Text (Tot. Cycles: 121546)</b>		<b>Text (Tot. Cycles: 141604)</b>	
Bit Field Operations	7.40%	4.50	7.00%	4.45
ShiftAdd Operations	7.60%	3.00	5.40%	3.45
Count Leading Zeroes	2.50%	49.45	12.70%	16.75
Store Operations	7.00%	1.00	6.70%	1.00
Load Operations	9.30%	2.00	7.50%	2.00

Table 8, instruction level profiling results from the Rice algorithm.

### 6.1.2 Profiling Huffman

The profiling results from the Huffman algorithm are found in Table 9. Since the adaptive Huffman algorithm is a symmetric compression algorithm, similar results are found for the encoding and decoding algorithm. It is clear from the results that there is much to gain from bit field operations, in addition to the possible reduction of load/store operations. Shift-Add operations are mostly used when the encoder must find the address to a leaf node from a table (the symbol value is shifted left once before it is added to the base address of the table). The total clock cycle count shows that the implementation is slightly asymmetric. The reason for this is that the decoder spends more clock cycles in the decoding function.

	Encode		Decode	
	Cycles/Tot.Cycles	Cycles/Call	Cycles/Tot.Cycles	Cycles/Call
<b>Distribution:</b>	<b>Poisson (Tot. Cycles: 450780)</b>		<b>Poisson (Tot. Cycles: 459299)</b>	
Bit Field Operations	26.30%	3.37	22.50%	3.29
ShiftAdd Operations	1.20%	3.70	0.30%	3.04
Store Operations	8.20%	1.00	8.20%	1.00
Load Operations	14.90%	2.00	15.10%	2.00
<b>Distribution:</b>	<b>Gamma (Tot. Cycles: 411086)</b>		<b>Gamma(Tot.Cycles: 418697)</b>	
Bit Field Operations	26.30%	3.38	22.50%	3.30
ShiftAdd Operations	1.20%	3.78	0.20%	3.06
Store Operations	8.20%	1.00	8.20%	1.00
Load Operations	14.90%	2.00	15.00%	2.00
<b>Distribution:</b>	<b>Exponential (Tot.Cycles: 195059)</b>		<b>Exponential (Tot.Cycles: 199180)</b>	
Bit Field Operations	23.50%	3.49	20.30%	3.41
ShiftAdd Operations	2.20%	3.95	0.10%	3.11
Store Operations	8.70%	1.00	9.10%	1.00
Load Operations	14.50%	2.00	14.40%	2.00
<b>Distribution:</b>	<b>Text (Tot. Cycles: 539648)</b>		<b>Text (Tot. Cycles: 550573)</b>	
Bit Field Operations	25.90%	3.36	25.40%	3.36
ShiftAdd Operations	1.10%	3.61	1.10%	3.61
Store Operations	8.20%	1.00	8.10%	1.00
Load Operations	15.10%	2.00	15.20%	2.00

Table 9, instruction level profiling results from the Huffman algorithm.

### 6.1.3 Profiling Deflate

The profiling results from the simple Deflate algorithm are found in Table 10. The table clearly shows how the encoder and decoder are asymmetric. It is also interesting to see how the total amount of used clock cycles increase when the encoder algorithm finds many matches in the exponential distributed input stream. It is evident that the encoding algorithm will have limited gain from bit field and shift-add operations. The decoding algorithm is very simple, and only a small increase in throughput is expected.

	Encode		Decode	
	Cycles/Tot.Cycles	Cycles/Call	Cycles/Tot.Cycles	Cycles/Call
<b>Distribution:</b>	<b>Poisson (Tot. Cycles: 138101)</b>		<b>Poisson (Tot. Cycles: 33516)</b>	
Bit Field Operations	0.20%	2.00	13.90%	3.39
ShiftAdd Operations	3.70%	3.00	0.00%	0.00
Store Operations	7.20%	1.00	7.50%	1.00
Load Operations	15.20%	2.00	7.40%	2.00
<b>Distribution:</b>	<b>Gamma (Tot. Cycles: 137268)</b>		<b>Gamma(Tot.Cycles: 31946)</b>	
Bit Field Operations	0.30%	2.00	12.80%	3.35
ShiftAdd Operations	3.50%	3.00	0.00%	0.00
Store Operations	7.00%	1.00	7.50%	1.00
Load Operations	15.10%	2.00	7.60%	2.00
<b>Distribution:</b>	<b>Exponential (Tot.Cycles: 515066)</b>		<b>Exponential (Tot.Cycles: 30513)</b>	
Bit Field Operations	0.10%	2.00	7.20%	3.12
ShiftAdd Operations	0.70%	3.00	0.00%	0.00
Store Operations	2.30%	1.00	5.20%	1.00
Load Operations	13.50%	2.00	5.60%	2.00
<b>Distribution:</b>	<b>Text (Tot. Cycles: 140460)</b>		<b>Text (Tot. Cycles: 30513)</b>	
Bit Field Operations	0.20%	2.00	13.90%	3.41
ShiftAdd Operations	3.50%	3.00	0.00%	0.00
Store Operations	6.90%	1.00	7.70%	1.00
Load Operations	15.30%	2.00	7.50%	2.00

Table 10, instruction level profiling results from the simple Deflate algorithm.

## 6.2 Proposals from Instruction Level Profiling

The results from the profiling show that it is possible to increase the throughput of the original algorithms by finding new instructions. Bit field operations may be needed for moving bit fields between registers or performing arithmetic or logical operations on bit fields in a register. The most common bit field operations in the source codes involve moving, inserting and adding bit fields. Clock cycles used in the blocks of instructions that perform bit field operations range from 3 to 6. Shift-Add operations are mostly used when finding memory addresses for table look-ups. The Count Leading Zeroes operation will also result in savings for the Rice algorithm. Before finding new instructions, the original instruction set (Table 4) must be analyzed to find space for the new instructions. The undefined space in the instruction set is shown in Table 11. From this, it is possible to make 16 two-register instructions. New “pre” types may also be defined. It is important to keep in mind the limitations in the NanoRisc architecture. The register bank has one write port and two read ports, and adding more ports to the register bank will increase the gate count considerably.

Bits															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	1
1	1	1	1	0	x	x	x	x	x	x	x	1	1	1	1

Table 11, undefined space in the original ISA.

### 6.2.1 Instruction Level Enhancements

Even though there is undefined space in the original instruction set for 16 two-register instructions, it is not preferable to use more space than necessary. An argument for this limitation is the gate cost when adding more instructions than strictly necessary. Adding instructions will also block future enhancements. If an instruction turn out to be less useful than anticipated, it still is hard to remove it from the instructions set. The removal of an instruction could result in rewriting of a number of programs. The encoding of the new proposed instructions is shown in Table 12. These are found by detailed examination of the source codes and profiling results.

Name	Code																Pre
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
insbfi	1	0	0	0	Rd				Rs				0	0	0	1	6
movbfi	1	0	0	0	Rd				Rs				0	1	0	1	6
movbf	1	0	0	0	Rd				Rs				0	1	1	1	7
clz	1	0	0	0	Rs				Rd				1	0	1	1	none
addbfi	1	0	0	0	Len				Rd				1	1	0	1	8
addbfhi	1	0	0	0	Len				Rd				1	1	1	1	8
ldin	1	0	0	1	Ra				Rd				0	0	0	1	8
stin	1	0	0	1	Ra				Rd				0	0	1	1	8
Pre																	
Type 6	1	1	0	1	Len				Pos Rs				Pos Rd				
Type 7	1	1	0	1	Rlen				Pos Rs				x	x	x	x	
Type 8	1	1	0	Imm													

Table 12, encoding of new instructions.

A more detailed description of these instructions is found in appendix A, while a short explanation will be given here. There are five bit field instructions:

- *insbfi* – takes a bit field from the source register (*Rs*) and inserts it in to the contents of the destinations register (*Rd*). All parameters (positions and length) are defined by immediate values in the new “pre” type 6 instruction.
- *movbfi* – has the same immediate parameters given by the new “pre” type 6 instruction as the “insbfi” instruction, but the bit field overwrites the contents of the destination register.
- *movbf* – has the same functionality as the “movbfi” instruction, but it uses a variable length given by a register (*Rlen*) in the new “pre” type 7 instruction and the bit field is moved to the lsbs of the destination register.
- *Addbfli* – adds an immediate value given in the new “pre” type 8 instruction to the lsbs of the source register given by the length parameter
- *Addbfhi* – adds an immediate value given in the new “pre” type 8 instruction to the msbs given by the length parameter.

An instruction that inserted a variable sized bit field may in many situations be preferable, but this would require one more read port than available in the architecture. Since the bit field in the “movbf” instruction is not inserted, the instruction needs only two read ports. A side effect from the new bit field instructions is that it becomes easier to pack information in the registers. This may reduce RAM access and hence contribute further to an increase in throughput. Apart from the bit field instructions there are three more instructions:

- *clz* – will write the amount of leading zeroes in the source register to the destination register. This instruction does not need a “pre” instruction.
- *ldin* – will load the contents of the memory of the address made when the *Ra* register is shifted left and added to the immediate value in the new “pre” type 8 instruction. The instruction will always load 16 bits.
- *stin* – stores a 16 bits value in the memory address made in the same way as for the “ldin” instruction.

From the descriptions of the new instructions above there are defined three new “pre” type instructions:

- *Type 6* – has three parameters; the length of the bit field, the position of the start bit of the bit field in the source register, and the position of the start bit in the destination register where the bit field is inserted.
- *Type 7* – also has three parameters; the register address holding the length of the bitfield, the position of the start bit of the bit field in the source register, and the position of the start bit in the destination register where the bit field is inserted.
- *Type 8* – gives a 12 bit immediate value.

Since there is little space in the original instruction set, there is an extensive use of “pre” instructions in the proposed instructions. This is a major disadvantage. A “pre” instruction will add one clock cycle of processing for every instruction that needs one. The original instruction set also has instructions that use “pre” instructions when the immediate values are big, or when three operands are used. However, the assembler will limit the use of “pre”



instructions by inserting them only when it is necessary. Because of limited space in the original instruction set, it is not possible to have small immediate values or other information within the new instructions. Another way of limit the use of “pre” instructions is therefore necessary. Since the instruction itself cannot hold the required information for the operation, default values may be the solution. All instructions in the proposed instructions, except “clz”, requires a “pre” instruction, this can be exploited by using default values in the instruction decoder if the instruction processed is not preceded with a “pre” instruction. The default values are chosen by examining the source code, and the result is shown in Table 13.

Instruction	Default values
Insbfi	Len = 8, PosRs = 7, PosRd = 7
Movbfi	Len = 8, PosRs = 7, PosRd = 7
Movbf	None
Clz	None
Addbfli	Imm = 1
Addbfhi	Imm = 1
Ldin	Imm = 0
Stin	Imm = 0

Table 13, default “pre” values.

### 6.2.2 Adding Non-Blocking Load Behavior

In addition to new instructions and HW accelerators, other enhancements of the NanoRisc processor could also increase the throughput of the algorithms. The RAM access during load operations will halt the processor until the result is retrieved from the RAM. This is shown in the timing diagram in Figure 26. When the processor is given access to the memory it will use two clock cycles until the retrieved data is in the register bank and can be used by other instructions. The first clock cycle is used for memory address calculation and set up, and in the second the result is written to the register and a new instruction is loaded. This is implemented in the instruction decoder as a 1-bit state machine. A way to avoid this extra clock cycle every load operation is to add a non-blocking behavior, from now on also referred to as NBL, to the load instructions. This means that the load instruction will not halt the processor until the data is retrieved, but go on processing the next instruction. However, the return operation from a function will still use two clock cycles when the instruction pointer is loaded from the stack pointer. Further enhancement could be to add NBL when the memory is busy and more clock cycles are wasted awaiting access to the memory, but this may be costly in terms of gates in the instruction decoder. Extra registers must be added in order to hold the load instruction and a possible “pre” instruction over several clock cycles. It would also in many cases save a limited amount of clock cycles, because loaded information is often used as quickly as possible to save register space.

Since the data from the memory is written to the register bank when another instruction is processed, it must have a separate write port. This will be costly in terms of gates. When this is implemented, it is important that the instruction after a load instruction is not dependent on the data retrieved from the memory, or that it does not cause a write conflict. This may be ensured by the programmer, the assembler software or in hardware. The best way of utilizing the NBL is that the programmer is aware of this, and avoids instructions dependent on a load operation to be processed immediately after the actual load operation. This may in many cases be tedious and result in difficult debugging if the programmer does not keep this in mind. A way of preventing such cases could be to enhance the assembler either by inserting “nop”

instructions where needed, or issue warnings or error messages where problems may occur. This would require no extra hardware except for the dedicated write port. The disadvantage of this method is that the program size grows when “nop” instructions are inserted, or in cases where it is difficult to do other tasks while awaiting the retrieved memory data. Such a method would also increase the complexity of the assembler, and may prevent others from enhancing or making their own assembler tools. The method chosen is to stall the processor in hardware when the *Rs* or *Rd* field of an instruction is equal to the *Rd* field of the proceeding load instruction. The extra cost in gate count in the instruction decoder will be small, and it is more user friendly.

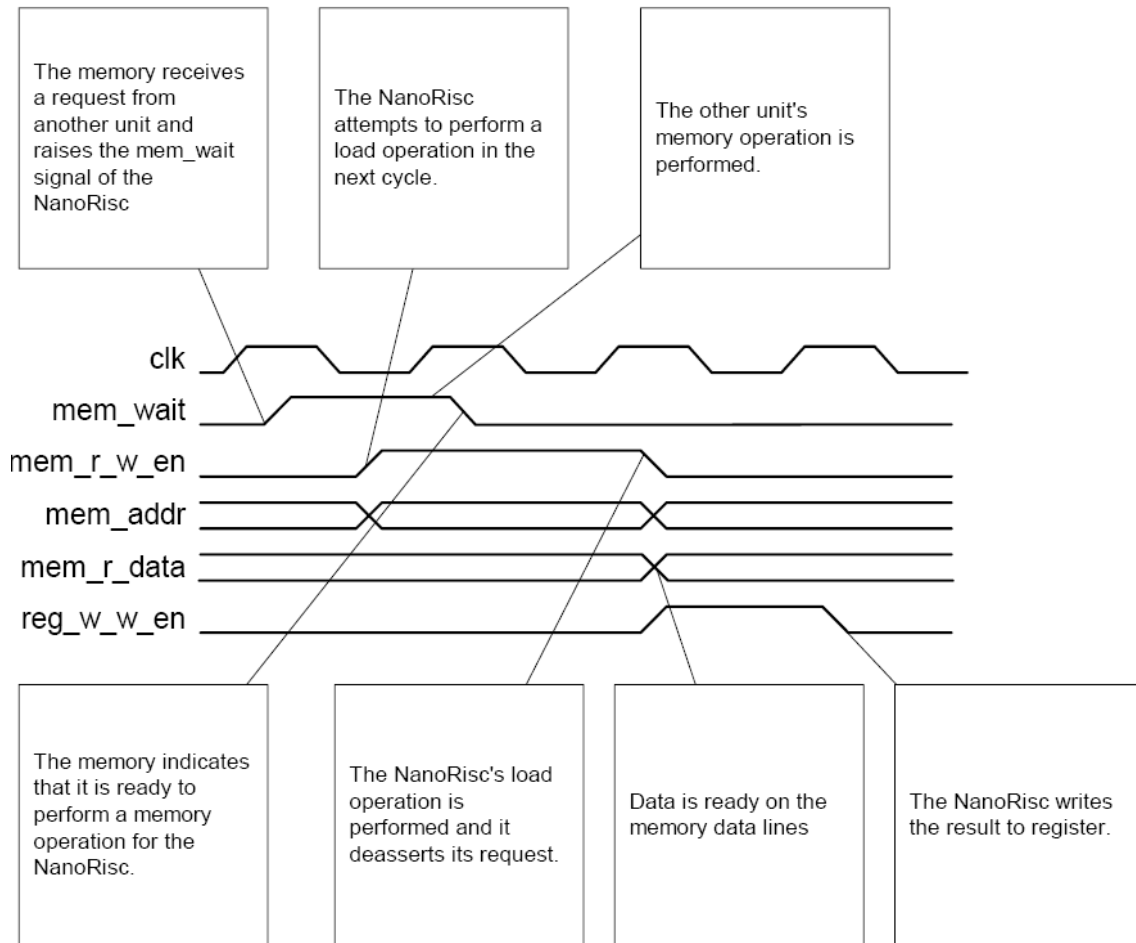


Figure 26, timing diagram during load operation.

### 6.2.3 Estimating Speedup

From the profiling results, Ahmdahl’s law (Eq. 6) may be applied to estimate how the new enhancements may affect the total speedup. The calculation is done by summing all operations that may be affected by the new instructions and the NBL, and estimating a general 2 times speedup for all. An example of this is shown in Table 8 where operations from the Huffman encoding algorithm are summed. The estimation of a general speedup of 2 is a ball park estimate from the profiling result. If all profiled operations are replaced by a “pre” instruction in addition to the new instruction, a speedup of 2 may seem a bit optimistic. However, taken into consideration a possible decrease in RAM access due to new bit field instructions, and that some of the operations will not need “pre” instructions due to default values, it may be a reasonable approximation.

<b>Poisson</b>	
<b>Operation</b>	<b>Cycles/Tot.Cycles</b>
Bit Field Operations	0.263
ShiftAdd Operations	0.012
Load Operations	0.149
Sum:	0.424

Table 14, operations that may be affected by the enhancements

Table 15 shows the estimated speedup when applying the considerations explained above and Ahmdal's law. The results vary greatly from about 10% increase up to 25%.

<b>Ditribution</b>	<b>Rice</b>		<b>Huffman</b>		<b>Deflate</b>	
	<b>Decode</b>	<b>Encode</b>	<b>Decode</b>	<b>Encode</b>	<b>Decode</b>	<b>Encode</b>
<b>Poisson</b>	1.240	1.188	1.234	1.269	1.119	1.106
<b>Gamma</b>	1.245	1.190	1.233	1.269	1.113	1.104
<b>Exponential</b>	1.238	1.182	1.211	1.251	1.068	1.077
<b>Text</b>	1.195	1.154	1.264	1.267	1.120	1.104

Table 15, estimated speedup from new instructions and NBL behavior.

### 6.3 Algorithmic Level Profiling

In section 6.1, the profiling areas contained blocks of instructions that could be replaced by new instructions. This section will show how the performance is increased when using the new enhancements proposed in section 6.2. The profiling areas from the instruction level profiling are hence replaced by new instructions and it would make little sense to use the same profiling areas. This section will describe the enhancements by defining new profiling areas that shows how clock cycles are used on an algorithmic level. As mentioned, algorithmic level profiling is profiling major parts of the compression algorithms, e.g. decoding, encoding, updating data structures, etc. It is therefore important that the reader keep in mind the description of the implementations of the different algorithms in chapter 4. Only a selection of the profiling results is shown in this section, but all profiling results are found in appendix D.

#### 6.3.1 Profiling Rice

From Table 8 in section 6.1.1 it was evident that the decoder algorithm had much to gain from the new "clz" instruction. Both the encoder and decoder had limited use of bit field and shift-add instructions. Table 16 shows the results of the algorithmic profiling with the NanoRisc processor using the exponentially distributed data stream. Only the decoding algorithm is shown due to the symmetrical behavior of the Rice compression method. The label column shows the label name of the different profiling areas. The resources used by the different profiling areas are shown as clock cycles used in the profiling area, and cycles used in the profile area divided by the total amount of cycles. The total number of cycles shown in the table is not a sum of the clock cycles used in the profile areas, but is the total clock cycles used by the algorithm. The "streams" label shows the resource used when reading variable length codewords from RAM. This operation is used in decoding, and is therefore sorted under the "decode" label in the table. The clock cycles used by the "decode" label thus includes the clock cycles used in the "streams" label. The RAM access is given in operations because it is measured at an instruction accurate level. In most cases, load operations will only use one clock cycle every operations due to the NBL, but function returns will still use two clock cycles.

Label	Original		Enhanced		
	Cycles	Cycles/Tot. Cycles	Cycles	Cycles/Tot. Cycles	Speed Up
<b>Decode</b>	<b>41790</b>	<b>42.68%</b>	<b>25116</b>	<b>36.97%</b>	<b>1.66</b>
<i>Streams</i>	25050	25.58%	23158	34.09%	1.08
<b>Calc K</b>	<b>930</b>	<b>0.95%</b>	<b>434</b>	<b>0.64%</b>	<b>2.14</b>
<b>Maintain Tables</b>	<b>346</b>	<b>0.35%</b>	<b>238</b>	<b>0.35%</b>	<b>1.45</b>
<b>Update Table</b>	<b>7934</b>	<b>8.10%</b>	<b>7818</b>	<b>11.51%</b>	<b>1.01</b>
<b>Total Number of Cycles</b>	<b>97912</b>	<b>100.00%</b>	<b>67941</b>	<b>100.00%</b>	<b>1.44</b>
<b>Memory Access [operations]</b>	<b>14758</b>		<b>8612</b>		
<i>Store Operations</i>	8156		5069		
<i>Load Operations</i>	6602		3543		

Table 16, algorithmic profiling from the Rice decoding algorithm with exponential distributed input stream.

Table 16 shows results from the profiling when the new instructions and the non-blocking load behavior are implemented. The “decode” label has reduced its resource use considerably, and it is the main contributor to the total speed up. Another contribution to the increased throughput is the decrease in RAM access due to more effective register usage. The throughput is increased by 44%.

### 6.3.2 Profiling Huffman

The profiling results from the original Huffman algorithm showed that bit field instructions and a non-blocking load behavior should increase the throughput considerably. Table 17 shows how the different algorithmic elements consume processing power with the NanoRisc processor. The “Sort Tree” and “Switch Node” are called from the “Increment Tree”, and the recurses used by the “Increment Tree” include these two functions. Results found in the table are retrieved from decoding the data stream with a Poisson distribution. Only the decoding algorithm is shown due to the symmetric behavior of the Huffman compression method.

Label	Original		Enhanced		
	Cycles	Cycles/Tot. Cycles	Cycles	Cycles/Tot. Cycles	Speed Up
<b>Increment Tree</b>	<b>345777</b>	<b>75.28%</b>	<b>189617</b>	<b>65.01%</b>	<b>1.82</b>
<i>Sort Tree</i>	174197	37.93%	101965	34.96%	1.71
<i>Switch Nodes</i>	15840	3.45%	10982	3.77%	1.44
<b>Insert Node</b>	<b>1040</b>	<b>0.23%</b>	<b>880</b>	<b>0.30%</b>	<b>1.18</b>
<b>Decode</b>	<b>100900</b>	<b>21.97%</b>	<b>86629</b>	<b>29.70%</b>	<b>1.16</b>
<b>Streams</b>	<b>14924</b>	<b>3.25%</b>	<b>14148</b>	<b>4.85%</b>	<b>1.05</b>
<b>Total Number of Cycles</b>	<b>459299</b>	<b>100.00%</b>	<b>291673</b>	<b>100.00%</b>	<b>1.58</b>
<b>Memory Access [operations]</b>	<b>107116</b>		<b>62811</b>		
<i>Store Operations</i>	37888		19307		
<i>Load Operations</i>	69228		43504		

Table 17, algorithmic profiling from the Huffman decoding algorithm with poisson distributed input stream.

It is clear from the table that updating the Huffman tree is the most time consuming task. The throughput is increased when the new instructions and the non-blocking load behaviour is implemented. Most of the savings are found in the “Increment Tree” label. This is mostly because of the implemented data structure (Figure 15). The throughput is increased by 58%. Another important reason for the increase in throughput is the decrease in RAM access.

### 6.3.3 Profiling Deflate

Section 6.1.3 showed that only a limited increase in throughput because of the enhancements is expected. The main contributor should be the new non-blocking load behavior.

Label	Original		Enhanced		
	Cycles	Cycles/Tot. Cycles	Cycles	Cycles/Tot. Cycles	Speed Up
<b>Decode</b>	<b>29258</b>	<b>95.89%</b>	<b>27238</b>	<b>95.60%</b>	<b>1.07</b>
<i>streams</i>	13747	45.05%	13462	47.25%	1.02
<b>Total Number of Cycles</b>	<b>30513</b>	<b>100.00%</b>	<b>28491</b>	<b>100.00%</b>	<b>1.07</b>
<b>Memory Access [operations]</b>	<b>4654</b>		<b>4654</b>		
<i>Store Operations</i>	2357		2357		
<i>Load Operations</i>	2297		2297		

Table 18, algorithmic profiling from the Deflate decoding algorithm with text input stream.

Table 18 shows the profiling results from the Deflate decoding algorithm, and Table 19 shows the result from the original Deflate encoding algorithm. The enhanced Deflate decoding algorithm source code shows a small increase in the throughput, about 7%. Because the decoding algorithm has limited use of bit field instructions, the RAM access is not reduced. From Table 19 it is evident that the two most time consuming tasks in the encoding algorithm is making the hash value and controlling matches. There is some gained from the non-blocking load behavior in the hash and control match labels, but since these function is already implemented in tight loops, it is difficult to take full advantage of this behavior. The increase in throughput is 16%.

Label	Original		Enhanced		
	Cycles	Cycles/Tot. Cycles	Cycles	Cycles/Tot. Cycles	Speed Up
<b>Encode</b>	<b>21828</b>	<b>15.54%</b>	<b>19859</b>	<b>16.41%</b>	<b>1.10</b>
<i>streams</i>	11452	8.15%	10102	8.35%	1.13
<b>Add Match</b>	<b>10000</b>	<b>7.12%</b>	<b>8000</b>	<b>6.61%</b>	<b>1.25</b>
<b>Control Match</b>	<b>40604</b>	<b>28.91%</b>	<b>36418</b>	<b>30.09%</b>	<b>1.11</b>
<b>CRC</b>	<b>32000</b>	<b>22.78%</b>	<b>26000</b>	<b>21.48%</b>	<b>1.23</b>
<b>Total Number of Cycles</b>	<b>140460</b>	<b>100.00%</b>	<b>121038</b>	<b>100.00%</b>	<b>1.16</b>
<b>Memory Access [ops.]</b>	<b>31169</b>		<b>30907</b>		
<i>Store Operations</i>	9685		9423		
<i>Load Operations</i>	21484		21484		

Table 19, algorithmic profiling from the Deflate encoding algorithm with text input stream.

## 6.4 Proposals from Algorithmic Level Profiling

Profiling on an algorithmic level showed that some parts of the algorithms could be further improved by adding enhancements that are more complex. This section will show how the stream functions and the hash function is speeded up.

### 6.4.1 The Stream Function

Common for all variable length coding algorithms is the task of reading or writing a stream of variable codes to or from RAM. From section 2.1, it was described how the Phillips TriMedia core used a coprocessor to deal with this task. A viable alternative for further improvement of the implemented algorithms could therefore be a hardware accelerator. It would speed up all of the implemented algorithms, and could be useful for other applications as well, but the gate cost will be considerable. For maximum speed up, the hardware accelerator must access memory, calculate memory addresses and length of the current stream, do shift operations, and have registers in order to hold initialization parameters, memory addresses, and parts of the stream. This sum up to a 16 bits adder, a 32 bits barrel shifter, memory interface, at least 4 registers and additional control logic. The gate count would be in the area of 1500 gates. This is rather expensive. Another approach described in section 2.2 is the bit-band regions used by the Atmel Cortex core. The bit-band regions enable the load and store operations in the Cortex core to address a stream of variable length codewords on a bit level. However, if this shall reduce the amount of clock cycles it must be able to address bit fields that expand over two memory addresses. In the NanoRisc core, this will require a state machine in the memory access module, and hence making these load and store operations require several clock cycles.

```

shift_istream:
    sub    R8, R_Count                ;Check if there is enough bits left in R_Res
    bra.n  get_packet
    movbf  R_Res[15:R8], R12         ;Extract bits R_Res
    sll   R8, R_Res                  ;Shift out bits from R_Res
    sll   R8, R_IStr                ;Make room in R_IStr for bits from R_Res
    or    R12, R_IStr               ;Mask in bits in R_IStr from R_Res
    ret
;Return from call

get_packet:
    add   R8, R_Count               ;Calculate bits left in R_Res
    movbf R_Res[15:R_Count], R12   ;Extract the remaining bits from R_Res
    sll   R_Count, R_IStr          ;Make room in R_IStr for bits from R_Res
    or    R12, R_IStr              ;Mask in remaining bits
    ld.w  [R_IStr_Addr]+, R_Res     ;Load new bits to R_Res
    sub   R_Count, R8              ;Calculate bits left to stream in to R_Istr
    mov   0x10, R_Count            ;Reset R_Count to 16
    bra  shift_istream

```

Figure 27, old stream function for decoding algorithms.

Figure 27 show the assembly code for the stream function in decoding algorithms, using the new instructions proposed in section 6.2. The function uses two registers to hold a part of the stream. The *R\_Istr* must always hold 16 bits from the stream in order to simplify the decoding process. To reduce memory access, the *R\_Res* holds a reservoir of bits to be shifted in to *R\_Istr*. When *R\_Res* is empty, the next 16 bits of the stream is loaded from RAM. *R\_Count* is the register that holds the amount of bits left in the *R\_Res* register. When calling this function, the amount of bits to be shifted into the stream is held in *R8*. The memory address for the input stream is held by register *R\_Istr\_Addr*. When the reservoir stream has enough bits for the operation, it will use 11 clock cycles, but if not it will use 23 clock cycles. Table 20 shows data from the stream profiling label when decoding the text stream. The Rice and Deflate

decoding algorithms have much to gain from enhancements affecting the stream function, but the Huffman decoding algorithm will not see that much improvement compared to the total clock cycle use.

	Calls	Cycles/Call	Cycles/Tot.Cycles
<b>Rice</b>	2010	12.87	23.37%
<b>Huffman</b>	1037	14.31	3.96%
<b>Deflate</b>	734	18.34	44.67%

Table 20, profiling results from the stream label when decoding the text stream.

A stream instruction is proposed for speeding up the streaming function. This instruction will shift in bits from a register into the register holding the stream according to a register holding the length. The instruction needs three read ports from the register bank, but the NanoRisc processor has only two. An extra read port will be expensive in terms of gates. To avoid this, a dedicated register is chosen to hold the bits to be shifted in. The dedicated register is one of the general registers, so it may still be used as a general register for all other instructions. This implies that when the instruction is executed, the bits intended to be shifted in must have been written to the dedicated register. The dedicated register must be chosen before synthesis. The encoding of the new instruction is shown in Table 21.

Name	Code														Pre	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2		1
<b>str</b>	1	0	0	1	Rlen			Rd				1	1	1	1	

Table 21, encoding of the str instruction.

When using this instruction in the stream function, we get the profiling results in Table 22. The table shows that the amount of clock cycles used by the function has been reduced, but it is still a major contributor to the total cycle count in the Rice and Deflate decoding algorithm.

	Calls	Cycles/Call	Cycles/Tot.Cycles
<b>Rice</b>	2010	8.25	17.70%
<b>Huffman</b>	1037	9.21	2.78%
<b>Deflate</b>	734	11.89	36.65%

Table 22, profiling results from the stream label when decoding the text stream and using the new str instruction.

The resulting assembly code when the “str” instruction is used is shown in Figure 23. The “str” instruction will only use one clock cycle because it needs no “pre” instruction. Table 22 and Figure 28 clearly show that it is still room for improvements in the stream function, but at a fairly high implementation cost as explained in the beginning of this section. The “str” instruction will be cheap in terms of gate count and easy to implement.

```

shift_istream:
  sub    R8, R_Count          ;Check if there is enough bits left in R_Res
  bra.n  get_packet
  str    R8, R_IStr          ;Stream in bits from special register R_Res
  sll    R8, R_Res           ;Shift R_Res accordingly
  ret
  ;Return from Call

get_packet:
  add    R8, R_Count         ;Calculate bits left in R_Res
  str    R_Count, R_IStr     ;Stream in bits from special register R_Res
  ld.w   [R_IStr_Addr]+, R_Res ;Load new bit to R_Res
  sub    R_Count, R8         ;Calculate bits left to stream in to R_Istr
  mov    0x10, R_Count       ;Reset R_Count to 16
  bra    shift_istream
  
```

Figure 28, new stream function for decoding algorithms.

### 6.4.2 The Hash Function

Even though the hash function in the Deflate encoding algorithm is implemented with the use of look up tables, it contributes greatly to the overall clock cycle use. A solution to this is to implement the hash function in hardware. This may be done through a separate hardware accelerator. Another method is to add a hash module in the NanoRisc architecture and use it with a new instruction. The latter method is chosen, and a new “crc” instruction is introduced.

The implemented hash function in the assembly source code is based on the 8 bits CRC8-CCITT standard. If a hash function is implemented in hardware, a simpler method could be used to reduce the gate cost. The drawback of a simpler hash function could be that the probability of collisions will increase. This will again decrease the throughput of the encoding algorithm because more potential matches would have to be checked out. Another consideration when deciding the hash function for a hardware implementation is the portability for other applications. Since the NanoRisc is likely to be implemented in a transceiver SoC, some sort of error detection is often implemented. One of the most frequently used error detection method is the CRC16-CCITT. This method will cost more gates than a simple hash function or the CRC8-CCITT. A way to decrease the gate cost is to limit the amount of bits used in the CRC calculation every clock cycle. By not using the whole 16 bits register as input, but still only calculating the hash value for 8 bits at a time (as for the old LUT-version), the gate count will be halved. The CRC16-CCITT will produce 16 bits hash values, but the table has only 256 entries. In [20] it is shown that all bits in a hash value from a CRC function have high information content. Thus, it does not matter which of the bits in the hash value that are chosen for the final 8 bits table index. The lower byte is used. When the “crc” instruction is implemented, the amount of clock cycles in the CRC function is halved, and the increase in throughput is now 31%. The 256 bytes look up table is made obsolete and removed. The CRC module is used with the instruction encoding in Table 23.

Name	Code															Pre
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
crc	1	0	0	0	Rs			Rd					1	0	0	1

Table 23, encoding of the crc instruction.

### 6.5 Proposed Enhancements for the NanoRisc

From the last sections, it is evident that the new instructions and the non-blocking load behavior have shown promise by increasing the throughput for all algorithms. Two new



instructions were identified through the algorithmic profiling, “str” and “crc”. The “str” instruction improves throughput for all algorithms, especially for the decoding algorithms. Its contribution is mainly through the stream function, but it has been useful in other parts of the algorithms as well. In the Deflate encoding algorithm, a new “crc” instruction was identified. The “crc” instruction has limited use in the implemented algorithms, except for the Deflate encoding algorithm, but for other programs it may increase processing power. Especially when taking into account that the NanoRisc is most likely to be used in a transceiver SoC. It will provide an effective instruction for error detection, generating pseudo random sequences, or for making hash values.

The enhancements chosen for implementation is adding the new instructions in Table 24, and implementing the non-blocking behavior. The new instructions are encoded as shown in Table 12, Table 21 and Table 23. The default values from Table 13 (section 6.2) are used to decrease the amount of “pre” instructions. The new “pre” types are shown in table Table 25. A more detailed description of the proposed instructions is found in appendix A.

Instruction	Cycles	Description	Pre
<i>addbfl</i>	1	Adds an immediate value to a bit field in the lsb's	8
<i>addbfhi</i>	1	Adds an immediate value to a bit field in the msb's	8
<i>clz</i>	1	Count the amount of leading zeroes	none
<i>crc</i>	1	Calculate CRC16-CCITT for 8 bits.	none
<i>inbfi</i>	1	Inserts a bitfield from a register into another register. Bit field given by immediate values.	6
<i>ldin</i>	1 or 2	Load word with left shifted address register and offset	8
<i>movbf</i>	1	Extracts a bitfield from a source register. Bit field given by immediate values.	7
<i>movbfi</i>	1	Extracts a bitfield from a source register. Length of bit field given by a register.	6
<i>stin</i>	1	Store word with left shifted address register and offset	8
<i>str</i>	1	Shifts in bits from a dedicated register into the destination register. The amount of bits shifted in is given by a register.	none

**Table 24, new Instructions.**

The most costly enhancements in terms of gates will be the non-blocking load behavior and the bit field instructions. The non-blocking load behavior will require an additional write port in the register bank, and the bit field instructions will require additional shifters and mask operations. Both the “clz” and “crc” instruction will also require additional modules, but the cost of these modules is moderate. The “stin” and “ldin” instructions will only require architectural changes by adding a path from the barrel shifter to the ALU. This will change the critical path, and may in turn increase the gate count in order to meet timing constraints.

Pre type	Cycles	Description
6	1	Immediate Rs pos, Rd pos and length for bit field operations
7	1	Immediate Rs pos and length register for bit field operations
8	1	12 bit immediate value

**Table 25, new “pre” types.**

## 6.6 Results Obtained From the Proposed Enhancements

Due to the new instruction introduced in section 6.4, the throughput has increased further compared to the profiling results in section 6.3. A summary of the increase in throughput for all algorithms when all enhancements are included are shown in Table 26. When comparing the results with the estimated speedup in Table 15 (section 6.2.3), it is evident that an approximate general speedup of 2 gave conservative results, even when taking into account that these estimation was made without considering the “str” and “crc” instruction. While the estimated speedup varied between 10% and 25%, the real speedup is between 18% and 103%.

	Rice		Huffman		Deflate	
	Decode	Encode	Decode	Encode	Decode	Encode
<b>Poisson</b>	63%	36%	60%	62%	29%	28%
<b>Gamma</b>	65%	36%	60%	61%	28%	24%
<b>Exponential</b>	64%	32%	56%	52%	103%	18%
<b>Text</b>	51%	30%	60%	62%	28%	31%

Table 26, increase in throughput.

Another effect, especially due to the bit field instructions, is that RAM access has been reduced. Table 27 show how much the RAM access has decreased in all algorithms. The only algorithm that shows no reduction is the Deflate decoding algorithm. This is because of its limited use of bit field instructions.

	Rice				Huffman				Deflate			
	Decode		Encode		Decode		Encode		Decode		Encode	
	Load	Store	Load	Store	Load	Store	Load	Store	Load	Store	Load	Store
<b>Poisson</b>	-30%	-30%	-37%	-40%	-37%	-49%	-37%	-47%	0%	0%	-13%	-1%
<b>Gamma</b>	-40%	-38%	-38%	-41%	-37%	-49%	-37%	-47%	0%	0%	-11%	-1%
<b>Exponential</b>	-53%	-43%	-41%	-42%	-35%	-42%	-31%	-39%	0%	0%	-5%	-5%
<b>Text</b>	-27%	-35%	-27%	-39%	-37%	-48%	-37%	-47%	0%	0%	-15%	-3%

Table 27, RAM access reduction.

In addition to the increased throughput and reduced memory access, the code size for all algorithms has been reduced because new instructions have replaced blocks of old instructions. Table 28 shows the code size and the reduction.

	Decode			Encode		
	Original [byte]	Enhanced [byte]	Reduction	Original [byte]	Enhanced [byte]	Reduction
<b>Rice</b>	358	246	-31%	336	246	-27%
<b>Huffman</b>	686	540	-21%	684	556	-19%
<b>Deflate</b>	172	156	-9%	360	340	-6%

Table 28, reduction in code size due to new instructions.

The compression ratios achieved by the profiled compression methods and the different input streams are found in Table 29. The streams symbol distributions are found in appendix B.

	Poisson	Gamma	Exp	Text
<b>Rice</b>	51.30%	54.70%	79.80%	37.20%
<b>Huffman</b>	51.40%	55.20%	79.40%	43.00%
<b>Deflate</b>	0.20%	10.20%	64.30%	10.20%

Table 29, compression ratios.

## 7 The Enhanced NanoRisc Processor

In order to make the original NanoRisc capable of behaving according to the new instructions found in chapter 6, new modules and signal paths are added while some of the original modules are altered. This chapter will describe how this is implemented in HW. First a quick overview of the changes in the data flow is given. In the following sections, some of the changes are described in detail. In the end, synthesis results concerning area, timing and power are described.

### 7.1 Implementation

The simple overview of the architecture in chapter 3 is still valid, but the data flow diagram has changed. New modules and data paths are added in order to support the new instructions and the non-blocking load behavior. Figure 29 shows the new data flow diagram where new modules are filled blue and new signal paths are marked red. As mentioned, a more detailed description of the most important new modules and implementation is given in the next sections, while a quick description will be given here:

- **Sign Extm**, has the same function as the original Sign Ext module. This new module is needed since the non-blocking load behavior needs a dedicated write port in the register bank.
- **Mask** produces two bit masks. A length indicator tells the module the amount of ones that it shall append to the two bit masks. One bit mask appends ones from the msb, while the other appends them from the lsb.
- **Extr** uses one of the bit masks produced by the Mask module to extract a bit field. It can extract a bit field from the shifter module or SRC MUX output. The bit field is used by the BFU or ALU module.
- **BFU** shifts and if necessary inserts the bit field from the Extr module to a given position.
- **CLZ** gives the amount of leading zeroes in the input.
- **CRC** calculates the new CRC value from the LSB of the input. It uses the standard CRC16-CCITT polynomial.

Some of the original modules are altered. Those altered have new signal lines to or from them, and in all cases, except for the register module and shifter module, it is only the MUX units inside the modules that are altered because of the new signals. The shifter module is also moved in front of the ALU module. This is done in order to do addition on shifted values for the new “ldin” and “stin” instructions. The VHDL source code for the CLZ, CRC, BFU, Extr and Mask modules are found in appendix E.

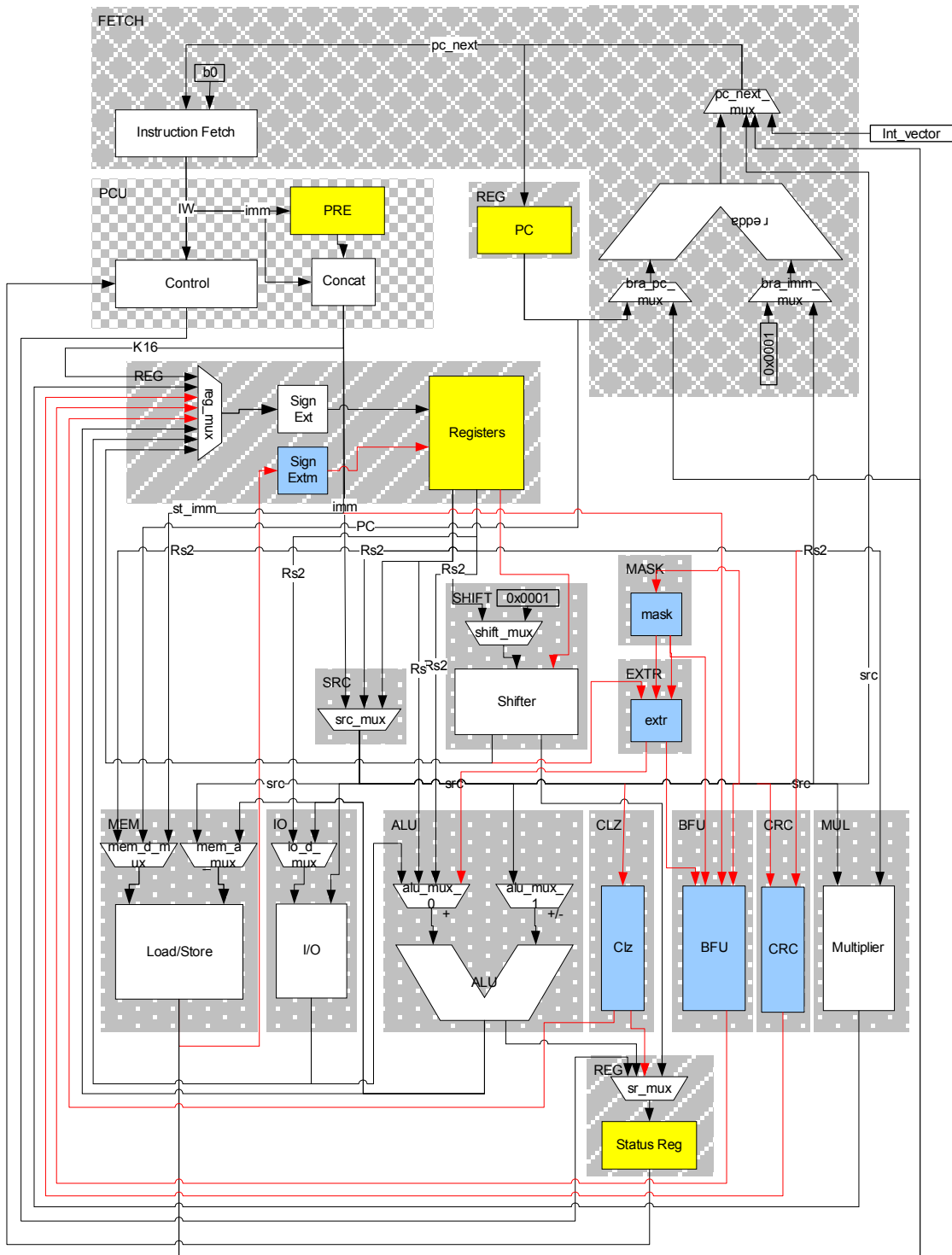


Figure 29, the enhanced NanoRisc data flow diagram.

### 7.1.1 Non-Blocking Load Behavior

As described in section 6.2.2, the original load instructions uses two clock cycles. This can be described by the 1-bit state machine in Figure 30. This state machine is controlled in the PCU, and it is reused when the new non-blocking behavior (NBL) is implemented. Both the NBL and the original state machine stall the next instruction fetch in the idle state if the memory is not ready for access. When the memory is ready, the idle state will set the address on the

memory address bus before making the transition to the load state in the next clock cycle. When making this transition, the original idle state will stall the next instruction fetch while the new NBL idle state will not. The load states purpose is to load the data on the memory data bus into the register bank. Since the original idle state stalled the instruction fetch, the original load state will not have a new instruction to decode, but the new NBL load state will. However, if the source or destination register in the new instruction is the same as the destination register in the load instruction, it will decode a “nop” instruction and stall the next instruction fetch. The new instruction is then decoded in the next clock cycle instead. If the source or destination register is not the same as the destination register in the load instruction, the new instruction will be decoded in the same clock cycle as the data is loaded from the memory data bus into the register bank.

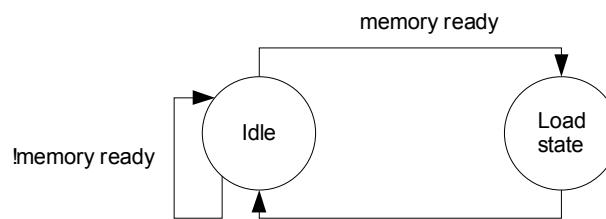


Figure 30, state machine during load instructions.

In order to fetch a new instruction in the idle state, a register is needed to hold the load instruction. Fortunately, this can be done through an extended use of a register already present in the PCU. The PCU has a dedicated register which holds the last instruction word in cases where a new instruction fetch is not performed. The use of this register is now extended to decode the load instruction during the load state. The changes in the *PCU* should not contribute much to the increase in gate count caused by the NBL behavior. The main contributor is expected to be the new dedicated write port in the register bank. This will add a DEMUX in order to be able to write all registers. Additional MUXes are also added by the synthesis tool since both write ports have access to the same registers. However, since the load state will decode a “nop” instruction in cases where the new instruction has the same source or destination register as the destination register in the load instruction, there can be no write conflicts.

### 7.1.2 Bit Field Instructions

The modules implemented for the bit field instructions are best explained through an example. The example will show the data flow when the “insbf” instruction is decoded with these parameters:

Len = 5  
 PosRs = 12  
 PosRd = 9

The content of the source register (*Rs*), the original destination register before the instruction, (*Rd*) and the new contents written to the destination register after the instruction (*New Rd*) are shown in Table 30. The bit field inserted from *Rs* to the new *Rd* is marked by a red outline.

Register	Binary														Hexadecimal		
	15	14	13	12	11	10	9	8	7	6	5	4	3	2		1	0
Rs	1	0	1	0	1	0	1	1	1	1	0	0	1	1	0	1	ABCD
Rd	1	1	0	1	1	1	0	0	1	0	1	1	1	0	1	0	DCBA
New Rd	1	1	0	1	1	1	0	1	0	1	1	1	1	0	1	0	DD7A

Table 30, Register contents in bit field example.

Figure 31 shows the data flow in the example. First, the contents of the *Rs* register is fed to the shift register. The shift registers control lines are set to left shift by the amount given by the inverted *PosRs*. This will shift the bit field to the msbs. The length is given to the Mask module, and it produces a bit mask with ones in the 5 msbs (zeroes in the rest). The result from the shifter and mask module is then given to the Extr module that simply do a bit-wise “and” operation on them. The BFU will take the results from the Mask and Extr module and right shift them according to the inverted *PosRd*. When this is done, both the bit mask and bit field is positioned correctly according to the insert position. In order to remove the bits in *Rd* which is to be replaced, the shifted bit mask is inverted and a bitwise “and” operation is performed on the inverted mask and *Rd*. A bitwise “or” operation between this result and the shifted bit field is then enough to insert the bit field.

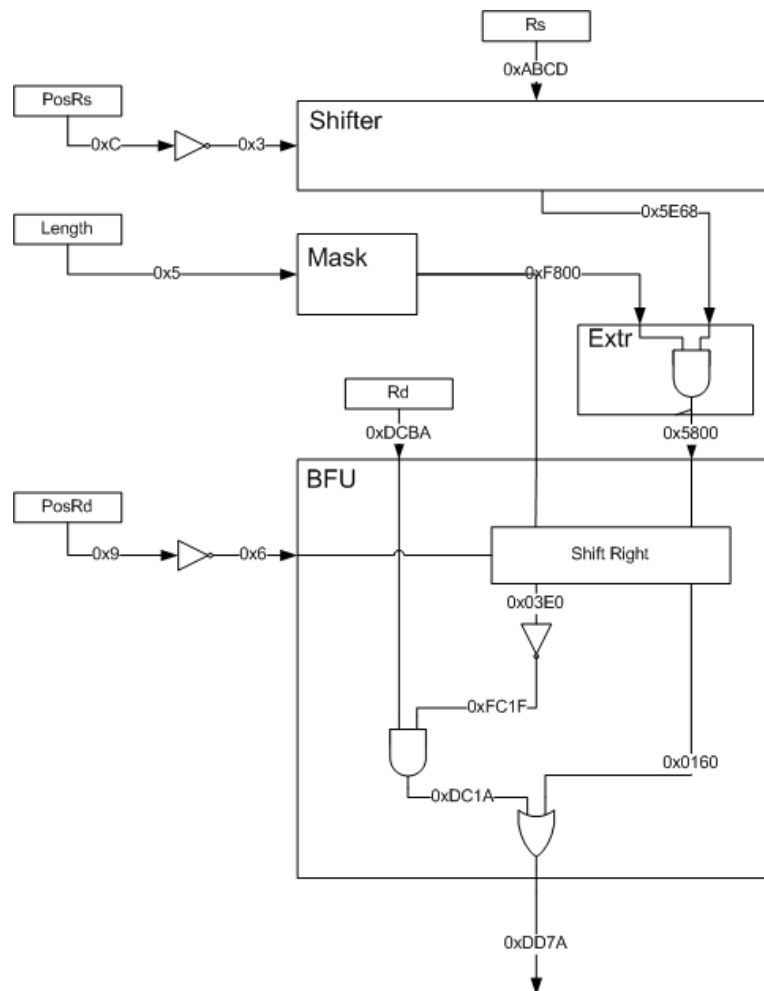


Figure 31, data flow in the bit field example.

In the “movbfi” and “movbf” instructions, the contents of the *Rd* register are not read since no insertion is needed. The “movbf” instruction will use the inverted *Len* as the amount of right shifts. Both the shift operations in the Shifter module and in the BFU module use inverted values for the amount of shifts. Because of this, the *PosRs* is inverted in the assembler. Thus no additional inverter is needed in the original Shifter module. The shift right operation in the BFU module will interpret the amount of right shift as an inverted value, so no inverter is needed in this case either. The inverters in Figure 31 are only illustrative. The “addbfli” and “addbfhi” instruction do not use the BFU module. For both instructions, the mask module will produce a mask with ones in the lsbs according to *Len*. The content of the destination register in the “addbfli” instruction is just fed to the Extr module together with the bit mask, and the result from this module is sent to the ALU. In the case of the “addbfhi” instruction, the content of the destination register is rotated left in the Shifter module according to *Len* before the same procedure is performed.

The most expensive part of this implementation is the Shift right operation in the BFU module. Since this operation must shift an arbitrary value in less than one clock cycle, it is implemented as a barrel shifter. The original Shifter module is also a barrel shifter and is implemented at the cost of 430 gates. This module has some additional features that are not used in the BFU, such as shift left and rotate operations. The gate count of the BFU module should be slightly less than that of the Shifter. However, the cost of the Mask and Extr modules also adds to the total costs of implementing bit field instructions. Taken this into account, the total cost will probably pass the gate count of the Shifter module.

### 7.1.3 Clz Module

There are several possibilities for implementing the “clz” function in HW. One method could be to add more logic to the shifter such that it could output information on the amount of leading zeroes. However, since the shifter is part of the most critical path after moving it in front of the ALU, adding more logic could further increase area to meet timing constraints. A separate CLZ module is therefore implemented.

The method used in the module can be described as a simple recursive function. In the first stage, the 16 bits input bits are divided in to the most and least significant byte. If the bits in the most significant byte are all zeroes, one is appended to the result and the least significant byte is multiplexed to the next stage. In each stage, the result from the last stage is divided into the most and least significant halves. As described for the first stage; if the most significant halves of bits being considered are all zeroes, one is appended to the result and the least significant bits are multiplexed to the next stage. If they are not all zeroes, zero is appended to the result and the most significant bits are multiplexed to the next stage. This is repeated until just a single bit remains.

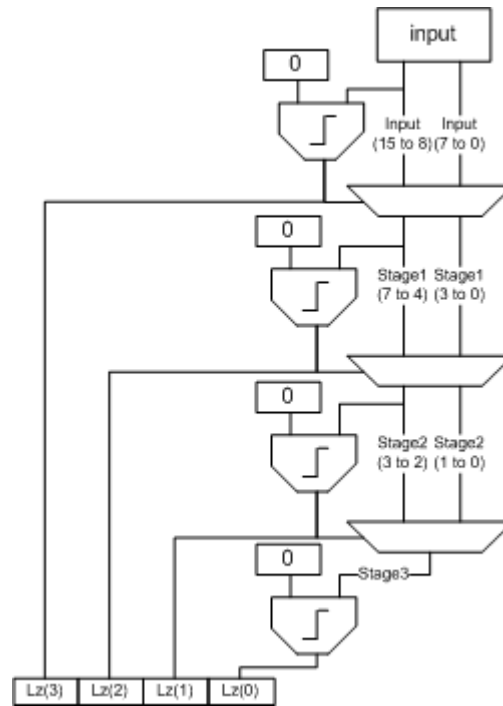


Figure 32, CLZ data flow diagram.

Figure 32 shows the data flow in this method.  $Lz$  is the result. The flow in the figure limits the result to 15, but an additional comparator not shown sets the result to 16 if all bits in the input are zero. If the result is 16 the zero flag in the status register is set, if the result is zero the negative flag in the status register is set. The module in itself should be synthesized to a small amount of gates. An estimate would be in the area of 50 gates.

#### 7.1.4 Crc Module

As explained in section 4.3.2, all CRCs are binary polynomials that are divided with the data. In general, division of large numbers is hard to implement in HW efficiently. Therefore it is more convenient to convert the binary information into a more appropriate form. The string of bits to be verified is represented as the coefficient of a large polynomial, rather than as a large binary number. The conversion of the CRC16-CCITT polynomial will be as follow:

$$G = 10001000000100001 = x^{16} + x^{12} + x^5 + 1$$

The CRC hash value is based on polynomial arithmetic. More accurate, the hash value is the remainder of dividing the polynomial in a Galois field with two elements ( $GF(2)$ ). A polynomial in  $GF(2)$  is a polynomial in a single variable  $x$  whose coefficients are 0 or 1. Addition and subtraction are done modulo 2, i.e. both operations are the same as the exclusive or operator (“xor”). Partial sums in division and multiplication are “xor’ed”. Using this kind of arithmetic, any remainder of a polynomial of  $n$  bits is no more than  $n-1$  bits long.  $n$  is referred to as the order of the polynomial. Another term often used is that the CRC hash value is the remainder of a binary division with no carry. As mentioned in section 4.3.2, CRC calculations are often implemented as liner feedback shift registers (LFSR). Figure 33 illustrates CRC16-CCITT calculation with an LFSR. The flip-flops are shift registers which store the remainder after every clock cycle. When the whole message has been shifted through, the LFSR will hold the final CRC hash value.



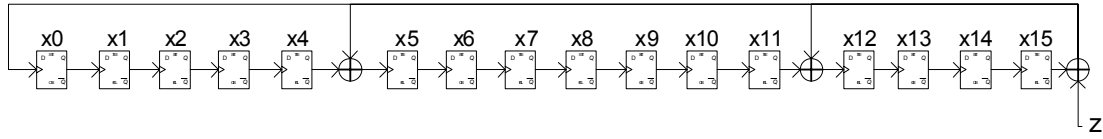


Figure 33, CRC16-CCITT calculation with LFSR.

In many systems where CRC is used for error detection, data is transferred as a serial bit stream. This is true for floppy disks, hard disks, modems, and as well as for newer optical disks. When the data stream is serial, the LFSR is trivial to implement and can operate at high speed. However, for this application (the “crc” instruction) it must operate at the clock speed used in the NanoRisc processor. The LFSR method is therefore too slow, and will not save many clock cycles in the compression algorithm. A method of calculating the hash value from more than one bit each clock cycle is necessary. Fortunately, there are many other applications that benefit from parallel CRC calculation (e.g. RAM disks and SCSI devices) and many methods have been developed to meet their need. One method has already been presented and implemented in this thesis; parallel CRC calculation with look up tables. This method is mostly used in software applications, and was proven too slow. The implemented HW method uses a “xor” network. The “xor” network express the contents of the shift register after 8 shifts as a function of the initial contents of the shift register and the 8 data bits shifted in. The resulting “xor” network using the CRC16-CCITT standard polynomial is shown below. The bits in the byte shifted in are denoted  $z_0$  through  $z_7$ .

$$\begin{aligned}
 x_0 &= z_0 \oplus z_4 \oplus x_8 \oplus x_{12} & x_1 &= z_1 \oplus z_5 \oplus x_9 \oplus x_{13} \\
 x_2 &= z_2 \oplus z_6 \oplus x_{10} \oplus x_{14} & x_3 &= z_3 \oplus z_7 \oplus x_{11} \oplus x_{15} \\
 x_4 &= z_4 \oplus x_{12} & x_5 &= z_0 \oplus z_4 \oplus z_5 \oplus x_8 \oplus x_{12} \oplus x_{13} \\
 x_6 &= z_1 \oplus z_5 \oplus z_6 \oplus x_9 \oplus x_{13} \oplus x_{14} & x_7 &= z_2 \oplus z_6 \oplus z_7 \oplus x_{10} \oplus x_{14} \oplus x_{15} \\
 x_8 &= x_0 \oplus z_7 \oplus z_3 \oplus x_{11} \oplus x_{15} & x_9 &= x_1 \oplus z_4 \oplus x_{12} \\
 x_{10} &= x_2 \oplus z_5 \oplus x_{13} & x_{11} &= x_3 \oplus z_6 \oplus x_{14} \\
 x_{12} &= x_4 \oplus z_0 \oplus z_4 \oplus z_7 \oplus x_8 \oplus x_{12} \oplus x_{15} & x_{13} &= x_5 \oplus z_1 \oplus z_5 \oplus x_9 \oplus x_{13} \\
 x_{14} &= x_6 \oplus z_6 \oplus z_2 \oplus x_{10} \oplus x_{14} & x_{15} &= x_7 \oplus z_3 \oplus z_7 \oplus x_{11} \oplus x_{15}
 \end{aligned}$$

These “xor” operations are then simply implemented in VHDL. The CRC module uses the standard CRC16-CCITT polynomial. This makes the module useless when other polynomials are required, but it limits the gate count considerably. A module with an optional polynomial will have the same complexity as a division module (600-700 gates), while this module with one fixed polynomial will have less than 100 gates.

### 7.1.5 The Str Instruction

Since the “str” instruction need three registers (a register that holds the stream, a register that holds the length to shift the stream and a register that holds the bits to be shifted into the stream) an extra read port in the register bank is needed. This is discussed in section 6.4.1. To avoid an extra read port a dedicated register is chosen to hold the bits to be shifted into the stream. This register must be chosen from one of the general-purpose registers before synthesizing, and it will not be possible change it during operation. The chosen register may

still be used as a general-purpose register, but when processing the “str” instruction it must hold the bits to be shifted in.

The original shifter module is enhanced in order to shift in bits to the stream register. The contents of the dedicated register are shifted left as many times as the destination register, and the bits falling off the dedicated register are shifted in to the destination register. The “str” instruction is only able to do left shifts. Hence, the original shifter is extended with one input port that is shifted equal amounts of times to the left as the other input port. This is a small operation, and the total cost in gates should be below 100. Figure 34 shows how the bits are shifted in to the stream register from the dedicated register. The amount of bits shifted in is given by the 4 lsb's of the length register.

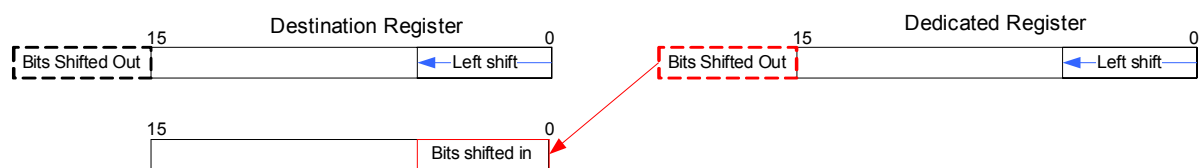


Figure 34, shift operation during the *str* instruction.

## 7.2 Synthesis

The synthesis of the design was done using the Synopsys Design Compiler with the Virage Logic TSMC 0.18um FSG DUS Standard Cell Library. The synthesis was done at two clock speeds: 25MHz for low power applications and 62.5 MHz for high performance applications. The synthesis was performed by scripts.

Since the NanoRisc microprocessor has no internal instruction register, the timing of the arrival from the program ROM is crucial. The timing data used to model this is taken from a high-speed single-port synchronous diffusion ROM made with the Artisan Rom generator for the TSMC 0.18um process. In a typical process with typical conditions, it has an address setup time of 0.31 ns and an access time of 1.29 ns. The address setup is the time the instruction address has to be stable on the address lines, and access time is the time from the rising clock to the instruction is stable on the data lines. These two constraints were given to the synthesis tool by using the “set\_output\_delay” and “set\_input\_delay” statements. The enhancements are added to the original NanoRisc processor core in six stages:

1. Non-Blocking Load Behavior (NBL).
2. The Count Leading Zeroes instruction.
3. Bit field instructions.
4. Load Index and Store Index instructions.
5. The CRC instruction.
6. The Stream instruction.

The stages are cumulative, so e.g. in stage number 6 all enhancements are added.

### 7.2.1 Timing

Timing reports show the estimated propagation delay through different paths in the circuit. The difference between this propagation delay and the timing constraint is called slack. If the slack is positive, the timing constraints are met, if it is negative, the timing constraints are not met. The synthesis tool will always try to meet the timing constraints by increasing drive

strength, insert clock buffers or inverter chains until the slack is zero. The path with least slack is often referred to as the critical path.

At 25 MHz the slack of the original NanoRisc is 20.54 ns, while after all enhancements are implemented (stage 6), the slack is 17.51 ns. This is a fairly large slack at 25 MHz, and the clock frequency may be increased by a factor of two without making noticeable changes in drive strength. The critical path in the original NanoRisc is from the program data lines, through the PCU, through the MUL module, and to the register bank. In stage 3, the shifter module and the Extr module is moved in front of the ALU, and this changes the critical path. The critical path after stage 3 is from the program data lines, through the PCU, through the shifter, through the Extr module, through the ALU and to the register bank.

At about 62.5 MHz, the synthesis tool starts to insert buffers and inverter chains to reduce fan out. The slack is zero for the original NanoRisc and all enhancement stages. The critical path changes much more between the stages depending on the choices the synthesis tool makes when trying to meet timing constraints. The original NanoRisc, stage 1, 4 and 5 has the same critical path as before (at 25 MHz), while stage 2 has a new critical path going from the program data lines, through the PCU, through the I/O module, through the ALU, and to the register bank. Stage 3 and 6 has the same critical path as the original NanoRisc and stage 1.

### 7.2.2 Area

The area of a circuit will often grow larger at higher speed. This is because of timing constraints that need to be met by the synthesis tool. Drive strength of components have to be increased to reduce propagation delay, and a unit with greater drive strength use more area. The synthesis tool may also insert buffers and inverter chains in order to meet timing constraints, and this will also add to the total gate count. Table 31 shows the gate count when the original NanoRisc is synthesized at 25 MHz and 62.5MHz. The difference in gate count is 323 gates.

	25 MHz	62.5 MHz
<b>nb_alu</b>	429.00	455.50
<b>nb_const_gen</b>	28.00	28.00
<b>nb_fetch</b>	252.50	252.50
<b>nb_int</b>	6.25	6.25
<b>nb_io</b>	170.00	195.25
<b>nb_mem</b>	218.50	218.50
<b>nb_mul</b>	408.50	425.75
<b>nb_pcu</b>	862.25	980.25
<b>nb_reg</b>	2299.27	2424.01
<b>nb_shift</b>	426.75	426.75
<b>nb_src</b>	69.00	80.25
<b>Total</b>	5170.02	5493.01

Table 31, gate count for the original NanoRisc.

As mentioned in section 7.2.1, the slack at 25 MHz was high. This implies that the gate count reported at 25 MHz should be near the minimum for the circuit. Table 32 shows how much each stage and module contributes to the total increase in gate count at 25 MHz. The extra cost in gates when implementing all enhancements is 1565 gates, making the total cost of the

enhanced NanoRisc 6735 gates at 25 MHz. The two major contributors to the total gate cost of the enhancements are the NBL and the modules needed for bit field instructions.

Module	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Total increase
nb_alu	0.50	-0.50	26.00	-0.50	0.00	-0.50	<b>25.00</b>
nb_const_gen	0.00	0.00	0.00	0.00	0.00	0.00	<b>0.00</b>
nb_fetch	0.00	0.00	0.00	0.00	0.00	0.00	<b>0.00</b>
nb_int	0.00	0.00	0.00	0.00	0.00	0.00	<b>0.00</b>
nb_io	0.00	0.00	0.00	0.00	0.00	0.00	<b>0.00</b>
nb_mem	0.00	0.00	0.00	0.25	-0.25	0.00	<b>0.00</b>
nb_mul	0.00	0.00	0.00	0.00	0.00	0.00	<b>0.00</b>
nb_pcu	59.75	13.25	125.75	40.00	6.25	6.00	<b>251.00</b>
nb_reg	573.03	19.50	20.50	2.50	41.00	-0.50	<b>656.03</b>
nb_shift	-0.50	0.50	0.00	-0.50	0.50	57.00	<b>57.00</b>
nb_src	0.00	0.00	0.00	0.00	0.00	0.00	<b>0.00</b>
nb_clz	0.00	39.75	0.00	0.00	0.00	0.00	<b>39.75</b>
nb_extr	0.00	0.00	48.00	0.00	0.00	0.00	<b>48.00</b>
nb_mask	0.00	0.00	67.00	0.00	0.00	0.00	<b>67.00</b>
nb_bfu	0.00	0.00	357.75	0.00	0.50	0.00	<b>358.25</b>
nb_crc	0.00	0.00	0.00	0.00	63.50	0.00	<b>63.50</b>
<b>Total increase</b>	<b>632.78</b>	<b>72.50</b>	<b>645.00</b>	<b>41.75</b>	<b>111.50</b>	<b>62.00</b>	<b>1565.53</b>

Table 32, contributions from each stage and module to the gate count at 25 MHz.

Table 33 shows how much each stage and module contributes to the increased gate count at 62.5 MHz. The penalty of moving the shifter in front of the ALU, and thereby changing the critical path, is clearly shown from stage 3 in the table. The resulting increase in gate count is more than double the increase from the non-blocking load in stage 1. The cost in gates for all enhancements is 1868, making the total cost of the enhanced NanoRisc 7361 at 62.5 MHz.

Module	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Total increase
Sum buffers	-4.00	8.75	42.00	-26.25	-5.00	32.00	<b>47.50</b>
nb_alu	0.25	5.25	118.25	-16.25	5.00	0.00	<b>112.50</b>
nb_const_gen	0.00	0.00	0.00	0.00	0.00	0.00	<b>0.00</b>
nb_fetch	0.00	0.00	-1.50	3.50	-3.50	0.00	<b>-1.50</b>
nb_int	0.00	0.00	0.00	0.00	0.00	0.00	<b>0.00</b>
nb_io	-21.75	1.00	-4.50	0.00	0.00	0.00	<b>-25.25</b>
nb_mem	0.00	0.00	0.00	0.00	0.00	0.00	<b>0.00</b>
nb_mul	-4.00	12.50	-9.00	-0.50	-3.50	4.25	<b>-0.25</b>
nb_pcu	44.50	-3.75	288.50	122.50	-142.50	81.75	<b>391.00</b>
nb_reg	472.29	11.25	134.26	-17.50	-4.25	26.50	<b>622.55</b>
nb_shift	0.00	0.00	111.75	-36.00	19.00	48.00	<b>142.75</b>
nb_src	-7.50	3.00	5.00	-4.00	0.25	-2.25	<b>-5.50</b>
nb_clz	0.00	39.50	0.00	0.00	0.00	0.00	<b>39.50</b>
nb_extr	0.00	0.00	48.00	0.00	0.25	-0.25	<b>48.00</b>
nb_mask	0.00	0.00	78.75	2.50	-1.75	-4.75	<b>74.75</b>
nb_bfu	0.00	0.00	358.75	-0.50	0.50	-0.50	<b>358.25</b>
nb_crc	0.00	0.00	0.00	0.00	64.00	-0.50	<b>63.50</b>
<b>Total increase</b>	<b>479.79</b>	<b>77.50</b>	<b>1170.26</b>	<b>27.50</b>	<b>-71.50</b>	<b>184.25</b>	<b>1867.80</b>

Table 33, contributions from each stage and module to the gate count at 62.5MHz.

An interesting effect is shown in stage 5 when the CRC module is introduced; the gate count is reduced. Because of the heavy timing constraints, the synthesis tool is forced to make decisions that may result in an area that is not optimal. By introducing the CRC module, it chooses a different approach to meet the timing constraints, and the result is reduced gate count.

### 7.2.3 Power

The power consumption was calculated in running mode without memories. The calculation was done by Synopsys Power Compiler. To estimate switching activity, the VHDL files are analyzed and elaborated in Design Compiler before creating a SAIF (Switching Activity Interchange Format) forward annotation file. This file tells the simulation tool which ports to log switching activity during simulation. ModelSim is used for simulation with the SAIF file, and it creates a SAIF back annotation file with the measured switching activity during simulation. This file contains average switching activity for all modules on RTL level, and is included in the synthesis before the design is compiled in Design Compiler. Power Compiler is then used to estimate power consumption. The global operating voltage is 1.62 V.

	Switch Power	Int Power	Leak Power	Total Power
<b>Original [mW]</b>	0.362	0.561	6.79E-03	0.929
<b>Enhanced [mW]</b>	0.427	0.632	8.61E-03	1.068

Table 34, estimated power consumption at 25 MHz [mW].

Table 34 show the estimated power at 25 MHz, while Table 35 show estimated power at 62.5 MHz. Both tables show values in mW. The total power consumption of the enhanced NanoRisc is 1.068 mW at 25 MHz, while at 62.5 MHz it is 2.816 mW. When the processor is in halted mode, no signals are toggling and the only power consumption should be leakage power.

	Switch Power	Int Power	Leak Power	Total Power
<b>Original [mW]</b>	0.928	1.433	7.19E-03	2.369
<b>Enhanced [mW]</b>	1.117	1.689	9.38E-03	2.816

Table 35, estimated power consumption at 62.5 MHz [mW].

The increase in power consumption for the enhanced NanoRisc is 15% at 25 MHz, and 19% at 62.5 MHz. However, these power estimations do not include the power consumption for ROM and RAM access. To investigate this, a 512x16 bits diffusion ROM and a 2048x16 bits single port synchronous SRAM is generated in an Artisan ROM and RAM generator. These generators use a TSMC 0.18um process. The results are found in Table 36.

Module	Read Power	Write Power	Standby Power
ROM	3.323	x	0.006
RAM	2.561	3.041	0.024

Table 36, power consumption for 512x16 bits ROM and 2048x16 bits RAM [mW] (global voltage 1.62, 25Mhz operation)

The power consumption for the ROM should be fairly accurate since the NanoRisc fetches a new instruction every clock cycle, but the power consumption in the RAM is too much since the access rate will be far less than 25 MHz.

### 7.3 Performance

Choosing measurements for comparison of the performance of processors are difficult. Many x86 based PC manufacturers tend to use clock frequency as a measure, but this is inaccurate when processors have different architectures and instruction sets. MIPS (Millions of Instructions per Second) is another measure that is frequently used among microcontroller manufacturers, but this may again be very misleading because of different instruction sets and architectures. Some instructions may be processed in one clock cycle in one microprocessor core, while two or three may be spent in another.

$$\text{Eq. 8} \quad \text{MIPS} = \frac{\text{InstructionCount}}{\text{ExecutionTime} * 10^6}$$

A more comparable measure is CPI (Cycles per Instruction). It measures how effective the cycles in a processor core are.

$$\text{Eq. 9} \quad \text{CPI} = \frac{\text{ClockCycles}}{\text{InstructionCount}}$$

Even though the NanoRisc is a RISC processor, the CPI deviates from 1 because of RAM reads may need more than one clock cycles and “pre” instructions do not count as an instruction. CPI can be calculated by running a program in the simulator and using the cycle and instruction count. When weighing the results from all input streams and algorithms equally, the CPI becomes 1.44 for the original NanoRisc and 1.37 with all enhancements. The difference would have been greater if only the non-blocking load behavior had been implemented, but since the new instructions are heavy users of “pre” instructions, the difference becomes smaller. Calculating MIPS at 25 MHz, the original NanoRisc get a value of 17.36 while the enhanced NanoRisc gets 18.25.

However, the NanoRisc processor is likely to be implemented in a transceiver SoC where the performance is much more than speed. A more appropriate measurement in this setting is a function of speed, size and power consumption. These factors should be weighted according to the application. Another measurement is speed compared with power consumption. From the power synthesis report, the power consumption in the original NanoRisc was 0.929 mW, while the enhancements increased it to 1.068 mW at 25 MHz. The original NanoRisc will hence consume approximately 0.054 mW per MIPS, while the enhanced NanoRisc will consume approximately 0.059 mW per MIPS. This is an increase of 7% in power consumption per MIPS. However, considering the reduction in number of clock cycles used to process the compression algorithms and the reduced RAM access, the net power consumption for the algorithms will be decreased. As shown in section 7.2.3, memory access costs more power than computations. From Table 27 in section 6.6 it is shown that all algorithms, except the Deflate decoding algorithm, have less RAM access due to the enhancements.

#### 7.3.1 Energy Savings

Section 7.2.3 showed the power consumption in the original NanoRisc, and the NanoRisc with all enhancements. A RAM and ROM module was also generated using the Artisan RAM and ROM generator. These modules were generated in a similar process as used when synthesizing the NanoRisc. A summary of these power estimations and the resulting energy consumption per clock cycle at 25 MHz are found in Table 37.

Module	Power Consumption [mW]	Energy per clock cycle [pJ]
Original NanoRisc	0.929 Average	37 Average
Enhanced NanoRisc	1.068 Average	42 Average
1 kB ROM	3.320 Read	132 Read
2 kB RAM	2.560 Read, 3.040 Write	102 Read, 121 Write

Table 37, power and energy consumption at 25 MHz.

From the profiling results presented in chapter 5 and found in appendix C and D, the energy consumed per bit for all algorithms can be calculated. The results for enhanced NanoRisc with the RAM and ROM module in Table 37 are shown in Table 38.

	Poisson [nJ]		Gamma [nJ]		Exponential [nJ]		Text [nJ]	
	Encode	Decode	Encode	Decode	Encode	Decode	Encode	Decode
<b>Rice</b>	1.65	1.66	1.61	1.61	1.43	1.43	2.23	2.25
<b>Huffman</b>	9.69	7.14	6.36	6.51	3.23	3.20	8.32	8.57
<b>Deflate</b>	2.76	0.64	2.81	0.61	10.60	0.38	2.72	0.59

Table 38, energy consumption per bit for the enhanced NanoRisc.

The original NanoRisc consumes less power, but uses more clock cycles and memory access than the enhanced. This results in total savings in energy for all algorithms. Reduced energy consumption for the enhanced NanoRisc compared to the original is shown in Table 39.

	Poisson		Gamma		Exponential		Text	
	Encode	Decode	Encode	Decode	Encode	Decode	Encode	Decode
<b>Rice</b>	-20%	-33%	-21%	-34%	-18%	-35%	-16%	-28%
<b>Huffman</b>	-30%	-29%	-29%	-29%	-25%	-27%	-30%	-29%
<b>Deflate</b>	-8%	-13%	-6%	-13%	-2%	-44%	-11%	-13%

Table 39, energy reduction per bit for the enhanced NanoRisc.

All calculations for the enhanced NanoRisc assume ROM access every clock cycle. This is because of the NBL behavior. In the original NanoRisc ROM, access is not performed during the first clock cycle in the load operation. Because of this and the fact that energy consumed in the NanoRisc core is small compared to ROM and RAM access, makes reduced RAM access the main contributor to the reduced energy.

All the implemented algorithms achieved bit compression. Thus, if the NanoRisc processor was implemented in a SoC to process compression algorithms, energy may be saved because fewer bits must be transmitted or received. For comparison, the CC2400 2.4 GHz low power transceiver [23] consumes 34.2 mW during transmission with a transmit power of 0dBm, while receiving it consumes 41.4 mW. The highest data rate is 1 Mbps. The energy consumed per bit when transmitting or receiving at this rate is hence 32 nJ and 41 nJ. Compared with the energy consumed in the NanoRisc per clock cycle the difference is a factor of 1000.

	Poisson		Gamma		Exponential		Text	
	Encode	Decode	Encode	Decode	Encode	Decode	Encode	Decode
<b>Rice</b>	-46%	-47%	-50%	-51%	-75%	-76%	-30%	-32%
<b>Huffman</b>	-30%	-34%	-35%	-39%	-69%	-72%	-17%	-22%
<b>Deflate</b>	8%	1%	-1%	-9%	-31%	-63%	-2%	-9%

Table 40, reduction in energy consumption due to compression with the enhanced NanoRisc.

Table 40 show the reduction in energy consumption due to compression with the enhanced NanoRisc. Encoding is assumed in the transmitter and decoding is assumed in the receiver. All algorithms except Deflate show energy reduction for all input streams. This is because of its poor compression ratios. For the poisson distributed input stream, the Deflate encode algorithm only removes 16 bits while using 108090 clock cycles and 28152 memory accesses. Still, the excess energy consumption is only 8.

The implemented algorithms are adaptive and they are therefore quite demanding when it comes to processing power. If the source is known and tend to have a rigid probability distribution, a static model may achieve good compression ratios. This is true for Rice and Huffman, but the Deflate algorithm must in all cases be adaptive. A static model will reduce clock cycle count and reduce RAM access significantly.

### 7.3.2 Benchmarks

The test of time is important for all instruction sets. The instruction set is one of the most important design issues for microprocessor core designers. If the core is to be widespread and used in many embedded solutions, a rigid and comprehensive instruction set is important for embedded software developers. Often it is the case that microprocessor core designers feel that some instructions are important, while in real life they are seldom used. Adding instructions may limit future expandability, especially for cores that are widespread. If an instruction proves to be less useful than anticipated, a removal may result in rewriting a number of programs.

A way to measure the effect of the improvements added to the NanoRisc for other types of tasks than data compression, is by using benchmarks. Benchmarks are important for embedded microprocessors in order to compare different microprocessors with different architectures and instruction sets. The Embedded Microprocessor Benchmark Consortium (EEMBC) is a non-profit group formed in 1997 to develop meaningful performance benchmarks for the hardware and software used in embedded systems. Their benchmarks have become an industry standard for evaluating the capabilities of embedded processors. The EEMBC benchmarks reflect real world applications and they are available as different suites targeting telecommunication, networking, network storage, digital entertainment, Java, automotive, industrial, consumer and office equipment products. Among these, the network benchmark suite is the most relevant since the NanoRisc is likely to be embedded in a transceiver SoC. Unfortunately the EEMBC benchmark suites are licensed, but earlier work [24] describes a benchmark, called NetBench, that is similar to the EEMBC network benchmark suite. NetBench is used for evaluating and designing network processors. A short summary of the programs used in the benchmark suite:

- CRC – Calculates the CRC-32 CCITT checksum.
- TL – A table lookup routine using a radix-tree routing table.
- DRR – Deficit-round robin scheduling
- NAT – Network Address Translation for IP address simplification and conservation.
- IPCHAINS – A firewall application that checks the IP source of incoming packets.
- URL – Implements URL-based switching.
- DH – Diffie-Hellman public key encryption.
- MD5 – Message Digest Algorithm creates a cryptographically signature for outgoing packets.



The deficit-round robin routine and the encryption algorithms will probably not gain much from the added instructions. However, for some of the other functions it is possible to estimate or show effects of the new improvements. The radix-tree routing table is very similar to Huffman Trees. A radix-tree is a tree with leafs representing keys, and each key can be found by traversing the path from the root to the leaf. This is in fact the decode routine in the implemented Huffman algorithm. Other important features of a radix-tree are switching nodes, inserting nodes and removing nodes. All except the remove nodes are implemented in the Huffman algorithm. Figure 35 shows the clock cycles used during these operations in the Huffman decoding algorithm. All input distributions show about 20% savings with the new extensions.

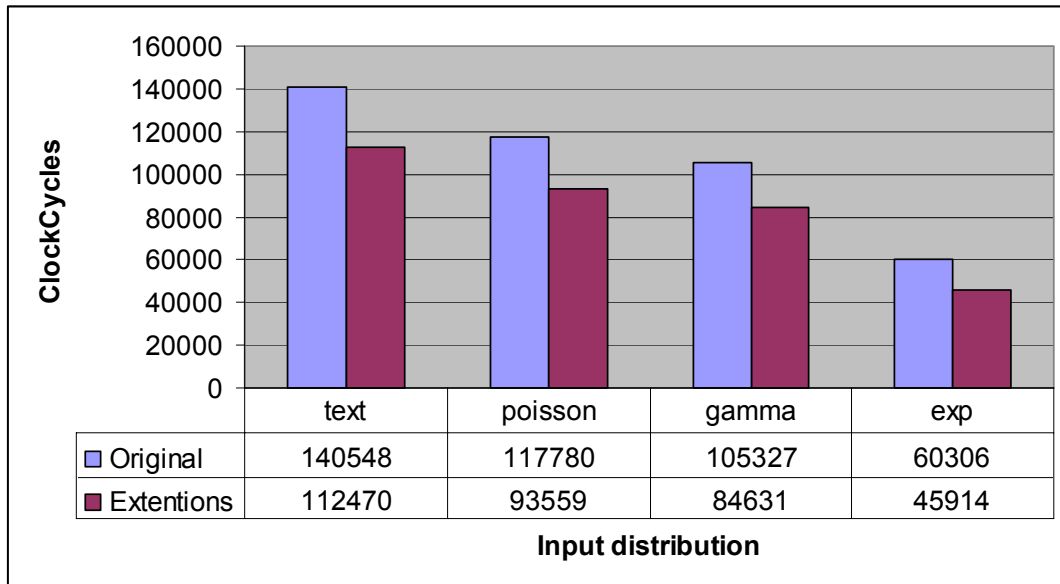


Figure 35, clock cycles used to decode, switch nodes and inserting nodes during Huffman decoding.

A CRC-32 calculation will be difficult since the NanoRisc has 16 bits data width. However, if the CRC-16 CCITT polynomial is used, it will of course gain from the “crc” instruction. This is shown in the Deflate encoding algorithm where the “crc” instruction halved the clock cycles used in the hash function and saved 256 bytes of memory. The other programs include some sort of fragmentation of the IP packet. Packet fragmentation is very typical for network processors or transceivers. When data packets are sent over a transmission medium, it is desirable that each data item in the packet is expressed in its natural size and not expanded to e.g. a 16 bits entity for convenience. This is to make the best use of the communication bandwidth. When packets are received or sent, they must be fragmented or assembled. It is not hard to think that such operations may benefit heavily from bit field operations.

From these estimations, it is probable that network applications would benefit from the improvements added to the original NanoRisc. The non-blocking load behavior will improve performance for applications that load information from RAM. Bit field instructions and the “ldin” and “stin” instructions will improve packet fragmentation and radix-tree routing. If the CRC-16 CCITT polynomial is used for error control, the “crc” instruction will also be a useful addition. In addition, as shown in this thesis, the “crc” instruction will provide a high quality hash function.

## 8 Discussion

In this chapter, some of the most interesting problems encountered in the thesis will be discussed. Also, a comparison is made in order to discuss the performance of the enhanced NanoRisc and the implemented algorithms. The last section will cover some of the future work that could be done.

### 8.1 Enhancements

All enhancements except the non-blocking load behavior has been instruction level enhancements. Instruction level enhancements are more intrusive to the processor core than HW accelerators. When instructions and modules are added to the core, it is important that these improvements are beneficial for a wide range of applications. In this thesis, only lossless compression algorithms are considered, while in a SoC the embedded microcontroller will handle a wide variety of tasks. This reasoning makes HW accelerators seem more suitable because they are easier to change/add between designs. As an example, the “crc” instruction is used to make hash values or checksums for error control, and it uses a fixed polynomial according to the CRC16-CCITT standard. If this function is implemented as a HW accelerator controlled by I/O ports, it will be easier to change polynomials or type of hash function between designs. The tradeoffs in such a HW accelerator would be speed and area. However, the “crc” instruction calculates a new CRC16-CCITT hash value from 8 bits in one clock cycle. If the accelerator were to compete with the implemented instruction in speed, it must have its own memory access module in addition to the hash function. This will increase the area considerably compared with the implemented CRC module, but it will enable the HW accelerator to work in parallel to the NanoRisc.

If the transceiver SoC is specified to transfer large amounts of data, data compression algorithms and the throughput of these may become vital. However, if the transceiver is specified to transfer a small amount of data, data compression is less vital and throughput requirements may be easier to meet. Adding or changing HW accelerators are more flexible between designs, but when implemented they tend to be more rigid (depending on how the module is specified). Another disadvantage of HW accelerators is that they do not usually utilize the capabilities already present in the processor core. If HW accelerators shall provide maximum speed up, they usually must have their own memory access module, ALU, registers etc. This implies a considerable increase in gate count compared to the implemented instructions.

As mentioned in section 2.1, the Phillips Trimedia uses a coprocessor (the VLD) to decode Huffman code words in MPEG1 and MPEG2. It is initialized with pointer to an MPEG1 or MPEG2 data stream. This module is thus responsible for the whole entropy decoding task. Such solutions could also be considered for the NanoRisc. A specialized HW accelerator dedicated for a specific compression algorithm should be superior in throughput compared to a program processed on the NanoRisc. So if speed is a major issue and if targeting a specific standard, such solutions must be taken into consideration. Another solution in high speed and high bit rate applications could be to run several NanoRisc processors in parallel. This would provide full flexibility, but possibly require more power and less speed compared to a specialized HW accelerator.

Another possibility to increase the performance is by altering or redesigning the original instruction set. This could make more room for specialized instructions for compression algorithms, and thereby reducing the number of “pre” instructions needed by the added instructions. Instructions removed or changed could still be performed by inserting macros to emulate the original instructions. A simple example of instructions that could be changed to improve the throughput is the shift and rotate instructions. In the encoding of these instructions there is one bit field that holds the immediate number of shifts/rotates or the register containing the amount of shifts/rotates. Which of the two possibilities that are valid, is decided by a “pre” instruction. If the shift or rotate instruction is preceded by a “pre” instruction, the PCU will know that the field contains a register address. When making a data compression program, using shift or rotate instructions with a register holding the amount of shifts/rotates is more probable than using immediate values. This is because data compression algorithms normally use variable length codes. Changing the decoding of this instruction in the PCU in a way that a “pre” instruction indicated an immediate value, rather than a register address, would be easy and probably have no effect on the gate count. However, since the main purpose of the NanoRisc microprocessor is to handle complex control tasks in SoC solutions, such enhancements are not considered. Changing or altering the original instruction set may have a negative affect on other applications.

## **8.2 Power**

The executing modules in the NanoRisc architecture are implemented without enable signals. This implies that all these modules may get new inputs and toggle accordingly every clock cycle, even though their result is not used. This is done to reduce area by keeping control logic at a minimum. The new modules added to the NanoRisc architecture are inserted in a similar way. The increase in power due to the new modules is in the area of 15%. This could be reduced by adding enable signals to the most power consuming modules. The enable signal would make sure that modules not in use had stable inputs and hence not toggle unnecessarily.

## **8.3 Timing and Throughput**

Through this thesis, improvements regarding throughput has been measured in clock cycles per processed bit of the input stream, but a more real life measurement is bits per second. This measure is similar to clock cycles per bit, except that clock cycles are converted to time units. Hence, timing becomes critical to the throughput. Considering this, improving or restructuring the architecture of the NanoRisc in order to increase the maximum clock frequency is a viable alternative to improving the functionality. This reasoning makes changing the critical path by moving the shifter and extr modules in front of the ALU seem like a bad choice. However, using the NanoRisc at very high clock speeds is not likely. In a transceiver SoC power is a vital measure, and this prevents very high clock speeds. The most likely clock frequency for an implementation with the NanoRisc microprocessor will be closer to 25 MHz than 62.5 MHz.

## 8.4 Assembly Source Code

The task of writing complex adaptive data compression algorithms in assembly proved to be a time consuming task. The author had very little experience in this before writing the test algorithms. The enhancements were proposed based on the profiling results from the original source codes. In the enhanced source codes, the changes were mostly added by replacing blocks of instructions that emulated the proposed instructions. Further increase of throughput could very well be obtained from rewriting the algorithms. This is true for both the original source codes and the enhanced source codes, and it could shift the effect of the enhancements either way. However, a total rewrite of the algorithms with full awareness of the capabilities in the proposed enhancements should in any case result in higher throughput. If this is not true, the proposed instructions are unnecessary and should be dismissed.

As mentioned, large or complex programs like some of the implemented data compression algorithms are time consuming to write in assembly. A “C” compiler would make this task less tedious. “C” has become the standard for writing high-level language programs for small microprocessors. The original NanoRisc architecture is very “C” compiler friendly with a large number of general registers and stack functionality. If a “C” compiler is to be made in the future and the instructions proposed in this thesis become part of the design, a “C” compiler must be able to utilize the proposed enhancements. This will not be straightforward. The bit field instructions may provoke unfamiliar “C” syntax, and the “str” instruction needs a dedicated register.

## 8.5 Area

The increase in area due to the enhancements is in the area of 30% at both clock speeds. This is based on the gate count of the processor core. Other considerations that can be taken into account when measuring the increase in area is the program ROM size. In Table 28, section 6.6 it is shown that all algorithms has reduced their program size.

There are mainly three different types of ROM available for implementation with the NanoRisc microprocessor; Diffusion ROM, Metallic ROM and One-Time Programmable ROM. There is also possible to synthesize the program as a logic gate array. The diffusion ROM is cheapest among the ROM alternatives in terms of bit density, and is the most probable to be chosen for implementations. The bit density varies from 2-4 gates per byte. A logic gate array is even cheaper in terms of area because the synthesis tool will utilize redundancy in the bit pattern of the program. When a logic gate array with random bit patterns are synthesized with the same library used in section 7.2, the bit density is in the area of 1 gates per byte. A logic gate array is very rigid and cannot be altered in the production stage. The contents of a diffusion ROM however can be altered in the production stage, but it is rather expensive and time consuming.

If a diffusion ROM with bit density of 2 gates per byte is chosen, the equivalent gate reduction due to reduced program size is in the area of 300 for Huffman, 200 for Rice and 40 for Deflate. When also taken into consideration the removal of the CRC LUT in the Deflate encoding algorithm, the gate count is further decreased by about 600 gates for this algorithm.

## 8.6 A Comparison

Some studies have been done in the area of lossless data compression for energy savings in wireless LAN networks. At MIT Laboratory for Computer Science, a report [25] has been published that uses lossless data compression software programs for compressing text and web data before transmission. The test setup used in the report is between a stationary PC and a handheld computer. The compression software evaluated in the report is asymmetrical library methods, and they are used on the whole file before transmission. A summary of the most important results are found in Table 41.

	<b>bzip2</b>	<b>compress</b>	<b>lzo</b>	<b>ppmd</b>	<b>zlib</b>
<b>Compression Ratio</b>	70	53	38	72	61
<b>Static memory allocation [Kbyte]</b>	8400	500	16	10000	130
<b>Intructions per bit removed (comp.)</b>	116	10	7	76	74
<b>Intructions per bit removed (decomp.)</b>	31	6	2	10	5
<b>Throughput (comp.) [Mbps]</b>	0.91	3.70	24.22	1.57	0.82
<b>Throughput (decomp.) [Mbps]</b>	2.59	11.65	109.44	1.42	41.15

Table 41, summary of results from "Energy Aware Lossless Data Compression" [25]

Table 42 shows the results obtained by the enhanced NanoRisc at 25 MHz. The results are calculated by averaging the results from all input data streams. Even though measurements concerning lossless data compression algorithms are very dependent on the input data stream, comparing the two tables will give a ball park estimate on the performance. A major difference between the two tables is the processing power. While the results from Table 41 are obtained by processing algorithms on a 233 MHz 32 bits StrongARM SA-110, the enhanced NanoRisc is running at 25 MHz using 16 bits computations. The difference in clock frequency and bit width is of great importance for the throughput and instructions per bit removed. There is also a difference in the file size. In Table 41 a whole data file is used, while all data streams used when testing the NanoRisc where 1000 bytes. This has impact on compression ratios and instructions per bit removed. Adaptive compression algorithms tend to use more computing and achieve less compression ratios in the beginning of the stream. This is because the modeling stage must adjust the incoming probability distribution. Later in the stream, the modeling stage has gathered more information and uses less time updating the memory structure holding the statistics.

	<b>Rice</b>	<b>Huffman</b>	<b>Deflate</b>
<b>Compression Ratio [%]</b>	55.75	57.25	21.23
<b>Static memory allocation [Kbyte]</b>	1	3.5	4.5
<b>Intructions per bit removed (comp.)</b>	16	54	112
<b>Intructions per bit removed (decomp.)</b>	16	55	14
<b>Throughput (comp.) [Mbps]</b>	2.76	0.80	1.05
<b>Throughput (decomp.) [Mbps]</b>	2.76	0.78	8.92

Table 42, results from the implemented algorithms in the enhanced NanoRisc at 25 MHz.

Considering the differences explained above, it seems like the enhanced NanoRisc is a powerful yet small and energy efficient alternative for data compression in small network applications.

## 8.7 Future Work

As mentioned in chapter 2 a paper [4] showed that adding bit field instructions to an ARM processor reduced the instructions executed at runtime between 5% and 28%, while the code size was reduced by between 2% and 21%. These results were gathered from testing the extensions with various benchmark suites. In this thesis, enhancements are added to the NanoRisc processor. Bit field instructions are added together with shift-add memory mode, non-blocking load behavior, and two instructions that are beneficial for streaming variable sized codes and CRC-16 calculations. These enhancements increased the throughput for the three implemented data compression algorithms by between 18% and 103%, while the code size was reduced by between 6% and 31%. In section 7.3.2, it is discussed how the improvements may benefit programs in a network benchmark suite, but the implemented enhancements have only shown their effect on the tested algorithms, so testing on a wide variety of applications should be done before deciding if they should be part of the design.

### 8.7.1 Processor Core

Since the processor core has no enable signals to its internal modules, it consumes power unnecessary when a program is processed. The power consumption during operation becomes very dependent on the gate count in the core, and this has a very negative effect on the added enhancements. To minimize this negative side effect, enable signals providing stable inputs for modules not in use should be considered.

### 8.7.2 Testing

The testing of the enhancements made to the NanoRisc was limited to simulations in the ISS and ModelSim. These tests verified the functionality, but it would also been useful to test the NanoRisc in hardware. An FPGA implementation was too time consuming for this thesis, but it should be considered for further evaluation and verification.

It is important to test the effect of the proposed instructions on other applications as well. If other applications seem to gain little from the enhancements, careful considerations should be made before deciding whether the enhancements should be part of the design. Further testing with a wide variety of programs should be done. It is also important to measure the effect of all proposed instructions individually. As mentioned, every added instruction may block future extensions.

In general, all programs that use information stored in memory will benefit from the non-blocking load behavior, and if they use memory, bit field operations may help them reduce memory access or memory allocation requirements. As shown in section 7.3.2, some programs in the NetBench benchmark suite will probably benefit from the new instructions, but a more comprehensive study should be done in order to establish their effect on common applications for the NanoRisc microprocessor.

Some may also argue that the proposed instructions are inconsistent. The only arithmetic bit field instructions are “addbfhi” and “addbfli”. Further testing could reveal a need for e.g. bit field subtraction instructions. A major drawback in the “addbfhi” and “addbfli” instructions is that the immediate value is limited to 12 bits. This is an effect of the limited space in the original instruction set. Further testing should reveal if this limits the effect of the instructions.

### 8.7.3 Tools

As mentioned in section 8.4 a “C” compiler should be made in order to ease the development of complex algorithms or control programs. In the beginning of this thesis when different data compression algorithms were evaluated for implementation, the complexity of the implementations called for an evaluation of making a “C” compiler. Writing a “C” compiler from scratch would not be feasible within the time limitation, but different toolsets could reduce the development time. One of the most widespread open source compiler framework is GCC [26]. However, even with the help from this toolset a report [27] estimates that 4 man-months are needed in order to port GCC to the OpenRISC architecture [28]. Thus, even with the help of toolsets it would not have been feasible within the time limitations.

The added profile enhancements to the existing tools measures clock cycle use within profile areas in the source code. As seen in section 6.6 and 7.2, there are more effects from adding functionality and instructions than reduced clock cycles. The bit field instructions helped reducing memory access, while all enhancements resulted in increased power consumption by the core. Power is an important measure in many SoC transceiver solutions. The profile tool added in this thesis could be further enhanced so it could estimate power consumption. From synthesis and experience power figures could be linked to instructions, memory access etc. This would help software developers to make power efficient programs.

## Conclusion

The goal of this thesis was to investigate the current capabilities of the NanoRisc microprocessor to process lossless compression algorithms, and find enhancements that improved its performance in this task. In order to measure performance, existing software tools are enhanced for profiling and simulating the improvements. Three fundamentally different data compression algorithms are implemented in the NanoRisc assembly language and simulated with the enhanced tools. On the background of these profiling results, some enhancements to the NanoRisc are proposed:

- Bit field instructions.
- New load and store instructions for table data structures.
- An instruction improving read and writes of variable length codewords from memory.
- An instruction improving CRC-16 checksum calculation.
- Non-blocking load behavior.

The new enhancements have improved throughput of the three implemented algorithms by between 18% and 103%, and the code size has decreased between 6% and 31%. Bit field instruction has also reduced RAM access by up to 53%. Compression ratios for the implemented algorithms on the tested input streams varied from 0.2% to 79.8%. Synthesis reports showed an increase in gate count of 30%, but the whole NanoRisc core is still below 7k gates. Power consumption per MIPS increased by 7%, however reduced clock cycle count and memory access due to bit field operations decreased the net power consumption for all tested algorithms. When calculating the energy used to remove bits in the compression algorithms with the transmit and receive power of the CC2400 2.4 GHz low power RF transceiver, the best case resulted in 76 % energy savings while the worst case resulted in a 8% energy increase.

This thesis has shown that major improvements of throughput for lossless compression algorithms are possible through enhancements of the NanoRisc processor at a fairly low gate cost. It is also shown that data compression with the enhanced NanoRisc may increase battery lifetime in a CC2400 low power transceiver 4 times. However, poor compression ratios may increase power consumption. This makes choosing the right compression algorithm crucial, but even though 108090 clock cycles and 28152 memory accesses are used to remove 2 out of 1000 bytes, the increased power consumption is no more than 8%.

The next step would be to do a more comprehensive study in order to establish the proposed instructions effect on common applications for the NanoRisc microprocessor.



## References

1. C.E. Shannon: "*A Mathematical Theory of Communications*", The Bell System Technical Journal, pp. 379-423, 623-656, July, 1948.
2. G. Amdahl: "*Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*", AFIPS Conference Proceedings, (30), pp. 483-485, April, 1967.
3. D.A. Huffman: "*A Method for the Construction of Minimum-Redundancy Codes*", Proceedings of the I.R.E, September 1952.
4. B. Li, R. Gupta: "*Bit Section Instruction Set Extension of ARM for Embedded Applications*", International Conference on Compilers, Architecture and Synthesis of Embedded Systems (CASES), p.p 69-78, October 2002.
5. R.B. Lee: "*Accelerating Multimedia with Enhanced Microprocessors*", IEEE Micro, vol. 15, no. 2, pp. 22-32, April 1995.
6. I. Kuroda, T. Nishitani: "*Multimedia Processors*", IEEE, vol 86, no. 6, pp. 1203-1221, June 1998.
7. <http://www.semiconductors.philips.com/products/nexperia/>
8. <http://www.arm.com/>
9. P. Rand: "*NanoRisc*", Thesis NTNU, June 2005.
10. C.H. Sequin, D.A. Patterson: "*Design and Implementation of RISC I*", Advanced Course on VLSI Architecture, University of Bristol, July 1982.
11. R. Pesh: "*GASP, an assembly preprocessor*", March 1994.
12. V. Paxson, W.L. Estes, J. Millaway: "*Flex, version 2.5.31*", March 2003.
13. C. Donnelly, R. Stallman: "*Bison*", September 2005.
14. E.V. Olufsen: "*Data Compression for Wireless Systems*", report in course TFE4700 NTNU, December 2005.
15. P.S. Yeh, R.F. Rice, W. Miller: "*On the Optimality of a Universal Noisless Code*", AIAA Computing in Aerospace 9 Conf. San Diego, October 1993.
16. Consultative Committee for Space Data Systems (CCSDS): "*Report Concerning Space Data Systems Standards*", CCSDS 120.0-G-1, Green Book, Issue 1, May 1997.
17. M. Weinberger, G. Seroussi, G. Sapiro: "*A Low-Complexity Context Based, Lossless Image Compression Algorithm*", IEEE Data Compression Conference, 1996.
18. D. Salomon: "*Data Compression, 3<sup>rd</sup> Edition*", Springer – Verlag New York Inc. ISBN 0-387-40697-2
19. R.J. Gorski: "*Hardware Capabilities and Protocols Availabilities*", A24073932, April 2005.
20. R. Jain: "*A Comparison of Hashing Schemes for Address Lookup in Computer Networks*", Digital Equipment Corporation, February 1989.
21. <http://wikipedia.org>
22. <http://www.itu.int/home/index.html>
23. <http://www.chipcon.com/>
24. G. Memik, W.H. Mangione-Smith, W. Hu: "*NetBench: A Benchmark Suite for Network Processors*", IEEE International Conference Computer Aided Design, November 2001.
25. K. Barr, K. Asanovic: "*Energy Aware Lossless Data Compression*", MIT Laboratory of Computer Science, May 2003.
26. <http://gcc.gnu.org>
27. M. Bolado, J. Castillo, H. Posadas, P. Sanchez, E. Villar, C. Sanchez, P. Blasco, H. Fouren: "*Using Open Source Cores in Real Applications*", XVIII Conference on Design of Circuits and Integrated Systems (DCIS2003), November 2003.
28. <http://www.opencores.org>

## Appendix

- A. New Instructions
- B. Symbol Distributions
- C. Detailed Instruction Level Profiling
- D. Detailed Algorithmic Level Profiling
- E. ZIP-File


## A. New Instructions

### addbfhi – Add Bit Field High Immediate

**Description** The bit field in Rs given by position 15 and length *Len* ( $K_1$ ) is added with an immediate value *Imm* ( $K_2$ ). The immediate value is limited to 12 bit and is given by the pre instruction. If the instruction is not preceded by a pre instruction it assumes a default immediate value. The addition is unsigned.

**Syntax** `addbf 1, Rd[K1:]`  
 pre:  
`addbf K2, Rd[K1:]`

**Operation**  $Rs = 1 + Rd[15:K_1]$   
 pre:  
 $Rs = K_2 + Rd[15:K_1]$

**Coding** 
 A diagram showing the bit fields of the instruction. It consists of a sequence of bits: 1, 0, 0, 0, followed by a field labeled 'Len' (blue), then a field labeled 'Rd' (orange), and finally four 1s (green).

**Program counter**  $PC = PC + 1$

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	-	-	-	-

**Pre** Type 8

**Default Value**  $Imm = 1$

**Cycles** 1

### addbfli – Add Bit Field Low Immediate

**Description** The bit field in Rs given by position  $Len (K_1)-1$  and length  $Len (K_1)$  is added with an immediate value  $Imm (K_2)$ . The immediate value is limited to 12 bit and is given by the pre instruction. If the instruction is not preceded by a pre instruction it assumes a default immediate value. The addition is unsigned.

**Syntax**  $addbf \ 1, Rd[:K_1]$   
 pre:  
 $addbf \ K_2, Rd[:K_1]$

**Operation**  $Rs = 1 + Rd[K_1-1:K_1]$   
 pre:  
 $Rs = K_2 + Rd[K_1-1:K_1]$

**Coding** 
 A diagram showing the bit fields of the instruction. It consists of a sequence of bits: 1, 0, 0, 0, followed by a field labeled 'Len' (4 bits), then a field labeled 'Rd' (4 bits), and finally the bits 1, 1, 0, 1. The bits are grouped into colored boxes: the first four bits are green, 'Len' is blue, 'Rd' is orange, and the last four bits are green.

**Program counter**  $PC = PC + 1$

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	-	-	-	-

**Pre** Type 8

**Default Value**  $Imm = 1$

**Cycles** 1

### clz – Count Leading Zeroes

**Description** Rd is given the value of the number of leading zeroes in Rs.

**Syntax** clz Rs, Rd

**Operation**  $Rd = 16 - |\log_2(Rs)|$

**Coding** 
 A bitfield diagram for the clz instruction. It consists of 14 bits. The first four bits are 1, 0, 0, 0. The next five bits are labeled 'Rs'. The next four bits are labeled 'Rd'. The final four bits are 1, 0, 1, 1. The bits are color-coded: the first four are green, 'Rs' is yellow, 'Rd' is orange, and the last four are green.

**Program counter** PC = PC + 1

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	0	*	*	0

The negative flag (N) is set to the msb of register Rs (answer is zero), and the zero flag is set if the result of the instruction is 16 (all zeroes).

**Pre** None

**Default Value** None

**Cycles** 1

**crc – CRC16**

**Description** Updates the value in Rd with a CRC calculation using the LSB of Rs. Will always use the CRC16-CCITT standard polynomial.

**Syntax** crc Rs, Rd

**Operation**  $Rd = CRC(Rd[7:8], Rd)$

**Coding**  A horizontal bar representing the instruction bit pattern. It consists of 14 bits: four green bits (1, 0, 0, 0), a yellow bit (Rs), a yellow bit, an orange bit (Rd), an orange bit, and four green bits (1, 0, 0, 1).

**Program counter** PC = PC +1

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	-	-	-	-

**Pre** None

**Default Value** None

**Cycles** 1

### insbfi – Insert Bit Field Immediate

**Description** The bit field in Rs given by position *PosRs* ( $K_1$ ) and length *Len* ( $K_3$ ) is inserted into Rd from position *PosRd* ( $K_2$ ). Positions *PosRs* and *PosRd* and length *Len* are given the pre instruction. If the instruction is not preceded by a pre instruction it assumes default values.

**Syntax** insbf Rs[7:8], Rd[7]  
 pre:  
 insbf Rs[ $K_1:K_3$ ], Rd[ $K_2$ ]

**Operation** Rd[7:8] = {Rd[15:8], Rs[7:8]}  
 pre:  
 Rd[ $K_2:K_3$ ] = {Rd[15:15- $K_2$ ], Rs[ $K_1:K_3$ ], Rd[ $K_2-K_3:K_2-K_3+1$ ]}

**Coding** 
 A bit field diagram showing 14 bits. The first four bits are 1, 0, 0, 0. The next four bits are labeled 'Rd'. The next four bits are labeled 'Rs'. The last four bits are 0, 0, 0, 1.

**Program counter** PC = PC + 1

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	-	-	-	-

**Pre** Type 6

**Default Value** Len = 8, PosRs = 7, PosRd = 7

**Cycles** 1

## ldin – Load index

**Description** The contents in the memory location specified by the value in Rs shifted left once and added with an immediate value (offset), are loaded into register Rd. This memory instruction has no size indicator, and will always load words. The immediate offset *Imm* is given by the pre instruction, and is limited to 12 bits. If the instruction is not preceded with a pre instruction it assumes a default immediate value.

**Syntax** ldin [Rs\*+0], Rd  
pre:  
ldin [Rs\*+K<sub>1</sub>], Rd

**Operation** Rd = M[Rs<<1+0]  
pre:  
Rd = M[Rs<<1+K<sub>1</sub>]

**Coding**  1 0 0 1 Ra Rd 0 0 0 1

**Program counter** PC = PC + 1

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	-	-	-	-

**Pre** Type 8

**Default Value** Imm = 0

**Cycles** 1



### movbf – Move Bit Field

**Description** The bit field in Rs given by position *PosRd* ( $K_1$ ) and the length in register *Rlen* (Rl) is moved to Rd from position *Rlen*-1. Positions *PosRd* and the length *Rlen* register are given by the pre instruction.

**Syntax** movbf Rs[K<sub>1</sub>:Rl], Rd

**Operation** Rd[K<sub>2</sub>:K<sub>3</sub>] = {0[15:15-Rl], Rs[K<sub>1</sub>: Rl]}

**Coding**  A horizontal bar representing the instruction bit field. It is divided into 14 segments. The first four segments are green and contain the bits 1, 0, 0, 0. The next four segments are orange and contain the label Rd. The next two segments are yellow and contain the label Rs. The final four segments are green and contain the bits 0, 1, 1, 1.

**Program counter** PC = PC + 1

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	-	-	-	-

**Pre** Type 7

**Default Value** None

**Cycles** 1

### movbfi – Move Bit Field Immediate

**Description** The bit field in Rs given by position *PosRs* ( $K_1$ ) and length *Len* ( $K_3$ ) is moved to Rd from position *PosRd* ( $K_2$ ). Positions *PosRs* and *PosRd* and length *Len* are given by the pre instruction. If the instruction is not preceded by a pre instruction it assumes default values.

**Syntax** movbf Rs[7:8], Rd[7]  
pre:  
movbf Rs[ $K_1:K_3$ ], Rd[ $K_2$ ]

**Operation** Rd[7:8] = {0[15:8], Rs[7:8]}  
pre:  
Rd[ $K_2:K_3$ ] = {0[15:15- $K_2$ ], Rs[ $K_1:K_3$ ], 0[ $K_2-K_3:K_2-K_3+1$ ]}

**Coding**  A bit field diagram for the instruction. It consists of 14 bits. The first four bits are 1, 0, 0, 0. The next four bits are labeled 'Rd'. The next four bits are labeled 'Rs'. The last four bits are 0, 1, 0, 1.

**Program counter** PC = PC + 1

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	-	-	-	-

**Pre** Type 6

**Default Value** PosRs = 7, PosRd = 7, Len

**Cycles** 1

## stin – Store Index

**Description** The contents in Rd are stored to the memory location specified by the value in Rs shifted left once and added with an immediate value *Imm* (offset). This memory instruction has no size indicator, and will always store words. The immediate offset is given by the pre instruction, and is limited to 12 bits. If the instruction is not preceded with a pre instruction it assumes a default immediate value.

**Syntax** stin Rd, [Rs\*+0]  
pre:  
stin Rd, [Rs\*+K<sub>1</sub>]

**Operation** M[Rs<<1+0] = Rd  
pre:  
M[Rs<<1+K<sub>1</sub>] = Rd

**Coding**  1 0 0 1 Ra Rd 0 0 1 1

**Program counter** PC = PC + 1

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	-	-	-	-

**Pre** Type 8

**Default Value** Imm = 0

**Cycles** 1

**str – Stream**

**Description** Shifts Rd and the dedicated register Rk equal amounts of times to the left. The Amount of shifts is the value in Rlen. The bits falling off Rk is shifted into Rd.

**Syntax** Str Rlen, Rd

**Operation**  $Rd = \{Rd[15-Rlen:15-Rlen+1], Rk\{15:15-Rlen+1\}\}$

**Coding** 
 A bit field diagram for the instruction. It consists of a sequence of bits: 1, 0, 0, 1, followed by a field labeled 'Rlen', then a field labeled 'Rd', and finally four 1s. The bits 1, 0, 0, 1, and the four 1s are shown in green boxes. The 'Rlen' and 'Rd' fields are shown in blue boxes.

**Program counter** PC = PC +1

**Status register**

HALT	IRQ	IE	V	N	Z	C
-	-	-	-	-	-	-

**Pre** None

**Default Value** None

**Cycles** 1

### New Prefixes

Type 6	1	1	0	1	Len			Pos Rs			Pos Rd			
Type 7	1	1	0	1	Rlen			Pos Rs			x	x	x	x
Type 8	1	1	0	Imm										

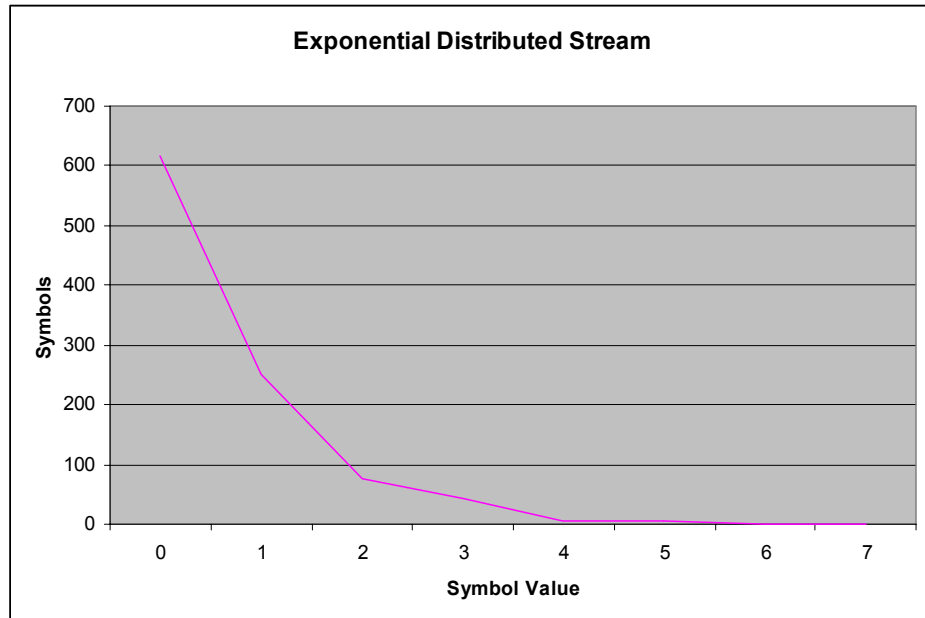
### Unused Space

1	0	0	0	x	x	x	x	x	x	x	x	0	0	1	1
1	0	0	1	x	x	x	x	x	x	x	x	0	1	0	1
1	0	0	1	x	x	x	x	x	x	x	x	0	1	1	1
1	0	0	1	x	x	x	x	x	x	x	x	1	0	0	1
1	0	0	1	x	x	x	x	x	x	x	x	1	0	1	1
1	0	0	1	x	x	x	x	x	x	x	x	1	1	0	1

## B. Symbol Distributions

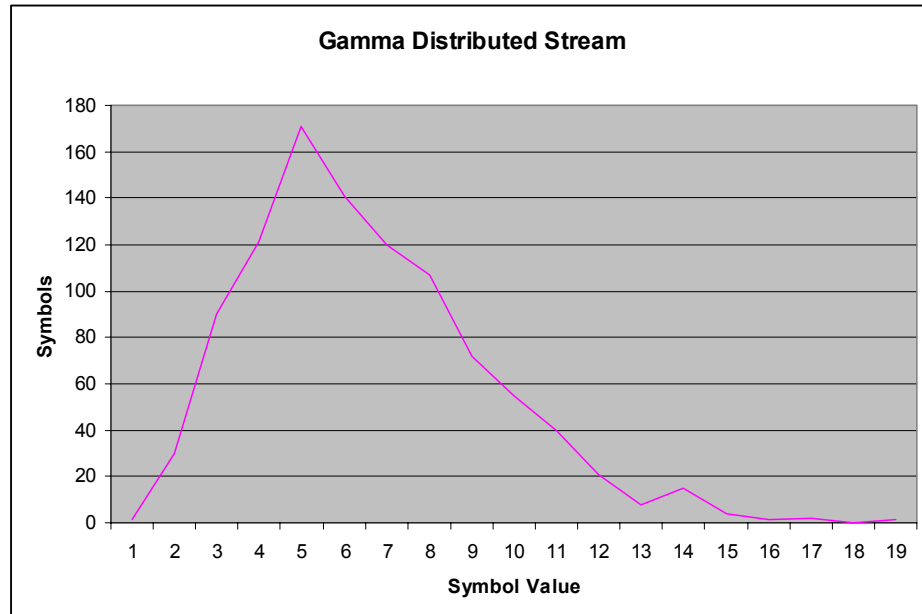
### Exponential

Value	Symbols
0	616
1	250
2	77
3	44
4	6
5	5
6	1
7	1



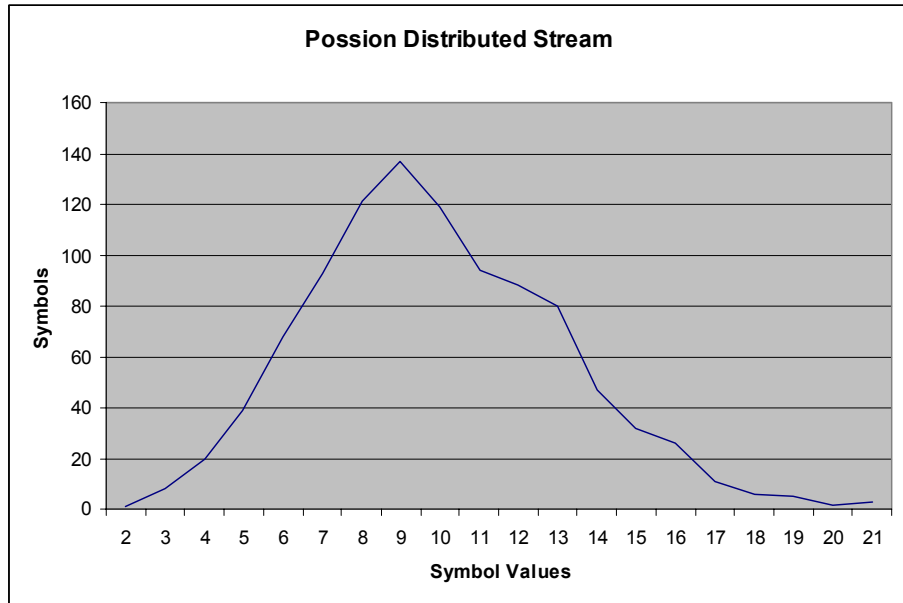
### Gamma

Value	Symbols
1	1
2	30
3	90
4	121
5	171
6	141
7	120
8	107
9	72
10	55
11	40
12	21
13	8
14	15
15	4
16	1
17	2
18	0
19	1



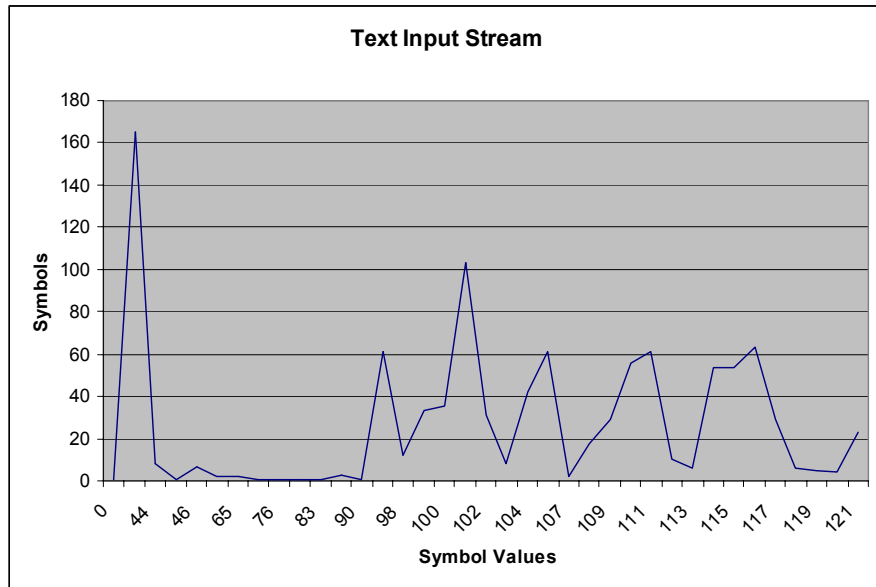
## Poisson

Value	Symbols
2	1
3	8
4	20
5	39
6	68
7	93
8	121
9	137
10	119
11	94
12	88
13	80
14	47
15	32
16	26
17	11
18	6
19	5
20	2
21	3



**Text**

Value	Symbols
0	1
32	165
44	8
45	1
46	7
55	2
65	2
72	1
76	1
82	1
83	1
84	3
90	1
97	61
98	12
99	33
100	35
101	103
102	31
103	8
104	42
105	61
107	2
108	18
109	29
110	56
111	61
112	10
113	6
114	54
115	54
116	63
117	29
118	6
119	5
120	4
121	23





### C. Instruction Level Profiling

Adaptive Huffman Encode (Exponential)				
Label	Cycles	Calls	Cycles/Call	Cycles/Tot.Cycles
<i>bfo1_insert_node1</i>	24	8	3.0	0.0001
<i>bfo1_ext_count2</i>	3384	1692	2.0	0.0171
<b>sum type 1</b>	<b>3408</b>	<b>1700</b>	<b>2.0</b>	<b>0.0172</b>
<i>bfo2_insert_node2</i>	24	8	3.0	0.0001
<i>bfo2_ext_count1</i>	5076	1692	3.0	0.0256
<i>bfo2_ins_count2</i>	4815	1605	3.0	0.0243
<b>sum type 2</b>	<b>9915</b>	<b>3305</b>	<b>3.0</b>	<b>0.0501</b>
<i>bfo3_ins_count1</i>	8025	1605	5.0	0.0405
<b>sum type 3</b>	<b>8025</b>	<b>1605</b>	<b>5.0</b>	<b>0.0405</b>
<i>bfo5_switch_nodes2</i>	48	16	3.0	0.0002
<i>bfo5_switch_nodes4</i>	48	16	3.0	0.0002
<i>bfo5_switch_nodes6</i>	36	12	3.0	0.0002
<i>bfo5_switch_nodes8</i>	36	12	3.0	0.0002
<i>bfo5_ins_count3</i>	4815	1605	3.0	0.0243
<b>sum type 5</b>	<b>4983</b>	<b>1661</b>	<b>3.0</b>	<b>0.0252</b>
<i>bfo9_switch_nodes1</i>	64	16	4.0	0.0003
<i>bfo9_switch_nodes3</i>	64	16	4.0	0.0003
<i>bfo9_switch_nodes5</i>	48	12	4.0	0.0002
<i>bfo9_switch_nodes7</i>	48	12	4.0	0.0002
<i>bfo9_encode1</i>	6444	1611	4.0	0.0325
<i>bfo9_sort_tree1</i>	6420	1605	4.0	0.0324
<i>bfo9_increment_tree1</i>	6420	1605	4.0	0.0324
<b>sum type 9</b>	<b>19508</b>	<b>4877</b>	<b>4.0</b>	<b>0.0985</b>
<b>Sum Bit Field Operations</b>	<b>45839</b>	<b>13148</b>	<b>3.5</b>	<b>0.2315</b>
<i>sa2_insert_node1</i>	24	8	3.0	0.0001
<i>sa2_insert_node2</i>	32	8	4.0	0.0002
<i>sa2_switch_nodes1</i>	63	21	3.0	0.0003
<i>sa2_switch_nodes2</i>	75	25	3.0	0.0004
<b>sum type 2</b>	<b>194</b>	<b>62</b>	<b>3.1</b>	<b>0.0010</b>
<i>sa3_main1</i>	4000	1000	4.0	0.0202
<b>sum type 3</b>	<b>4000</b>	<b>1000</b>	<b>4.0</b>	<b>0.0202</b>
<b>Sum ShiftAdd Operations</b>	<b>4194</b>	<b>1062</b>	<b>3.9</b>	<b>0.0212</b>
<b>Streams</b>	<b>7504</b>	<b>1008</b>	<b>7.4</b>	<b>0.0379</b>

**Total Number of Cycles**                      **198035**

**Memory Access [ops.]**                      **47305**

*Store Operations*                              17955

*Load Operations*                              29350

<b>Adaptive Huffman Decode (Exponential)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo1_insert_node1</i>	24	8	3.0	0.0001
<i>bfo1_ext_count2</i>	3476	1738	2.0	0.0175
<b>Sum type 1</b>	<b>3500</b>	<b>1746</b>	<b>2.0</b>	<b>0.0176</b>
<i>bfo2_insert_node2</i>	24	8	3.0	0.0001
<i>bfo2_ext_count1</i>	5214	1738	3.0	0.0262
<i>bfo2_ins_count2</i>	4932	1644	3.0	0.0248
<b>Sum Type 2</b>	<b>10170</b>	<b>3390</b>	<b>3.0</b>	<b>0.0511</b>
<i>bfo3_ins_count1</i>	8220	1644	5.0	0.0413
<b>Sum Type 3</b>	<b>8220</b>	<b>1644</b>	<b>5.0</b>	<b>0.0413</b>
<i>bfo5_switch_nodes2</i>	54	18	3.0	0.0003
<i>bfo5_switch_nodes4</i>	54	18	3.0	0.0003
<i>bfo5_switch_nodes6</i>	42	14	3.0	0.0002
<i>bfo5_switch_nodes8</i>	42	14	3.0	0.0002
<i>bfo5_ins_count3</i>	4932	1644	3.0	0.0248
<b>Sum type 5</b>	<b>5124</b>	<b>1708</b>	<b>3.0</b>	<b>0.0257</b>
<i>bfo9_switch_nodes1</i>	72	18	4.0	0.0004
<i>bfo9_switch_nodes3</i>	72	18	4.0	0.0004
<i>bfo9_switch_nodes5</i>	56	14	4.0	0.0003
<i>bfo9_switch_nodes7</i>	56	14	4.0	0.0003
<i>bfo9_sort_tree1</i>	6576	1644	4.0	0.0330
<i>bfo9_increment_tree1</i>	6576	1644	4.0	0.0330
<b>Sum type 9</b>	<b>13408</b>	<b>3352</b>	<b>4.0</b>	<b>0.0673</b>
<b>Sum Bit Field Operations</b>	<b>40422</b>	<b>11840</b>	<b>3.4</b>	<b>0.2029</b>
<i>sa2_insert_node1</i>	24	8	3.0	0.0001
<i>sa2_switch_nodes1</i>	75	25	3.0	0.0004
<i>sa2_insert_node2</i>	32	8	4.0	0.0002
<i>sa2_switch_nodes2</i>	87	29	3.0	0.0004
<b>Sum type 2</b>	<b>218</b>	<b>70</b>	<b>3.1</b>	<b>0.0011</b>
<b>Sum ShiftAdd Operations</b>	<b>218</b>	<b>70</b>	<b>3.1</b>	<b>0.0011</b>
<b>Streams</b>	<b>12265</b>	<b>1008</b>	<b>12.2</b>	<b>0.0616</b>

**Total Number of Cycles**                      **199180**

**Memory Access [ops.]**                      **46966**

*Store Operations*                              18205

*Load Operations*                              28761

<b>Adaptive Huffman Encode (Gamma)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo1_insert_node1</i>	54	18	3.0	0.0001
<i>bfo1_ext_count2</i>	10842	5421	2.0	0.0262
<b>sum type 1</b>	<b>10896</b>	<b>5439</b>	<b>2.0</b>	<b>0.0263</b>
<i>bfo2_insert_node2</i>	54	18	3.0	0.0001
<i>bfo2_ext_count1</i>	16263	5421	3.0	0.0393
<i>bfo2_ins_count2</i>	10335	3445	3.0	0.0250
<b>sum type 2</b>	<b>26652</b>	<b>8884</b>	<b>3.0</b>	<b>0.0644</b>
<i>bfo3_ins_count1</i>	17225	3445	5.0	0.0416
<b>sum type 3</b>	<b>17225</b>	<b>3445</b>	<b>5.0</b>	<b>0.0416</b>
<i>bfo5_switch_nodes2</i>	123	41	3.0	0.0003
<i>bfo5_switch_nodes4</i>	123	41	3.0	0.0003
<i>bfo5_switch_nodes6</i>	222	74	3.0	0.0005
<i>bfo5_switch_nodes8</i>	222	74	3.0	0.0005
<i>bfo5_ins_count3</i>	10335	3445	3.0	0.0250
<b>sum type 5</b>	<b>11025</b>	<b>3675</b>	<b>3.0</b>	<b>0.0266</b>
<i>bfo9_switch_nodes1</i>	164	41	4.0	0.0004
<i>bfo9_switch_nodes3</i>	164	41	4.0	0.0004
<i>bfo9_switch_nodes5</i>	296	74	4.0	0.0007
<i>bfo9_switch_nodes7</i>	296	74	4.0	0.0007
<i>bfo9_encode1</i>	13880	3470	4.0	0.0335
<i>bfo9_sort_tree1</i>	13780	3445	4.0	0.0333
<i>bfo9_increment_tree1</i>	13780	3445	4.0	0.0333
<b>sum type 9</b>	<b>42360</b>	<b>10590</b>	<b>4.0</b>	<b>0.1023</b>
<b>Sum Bit Field Operations</b>	<b>108158</b>	<b>32033</b>	<b>3.4</b>	<b>0.2612</b>
<i>sa2_insert_node1</i>	54	18	3.0	0.0001
<i>sa2_insert_node2</i>	72	18	4.0	0.0002
<i>sa2_switch_nodes1</i>	465	155	3.0	0.0011
<i>sa2_switch_nodes2</i>	366	122	3.0	0.0009
<b>sum type 2</b>	<b>957</b>	<b>313</b>	<b>3.1</b>	<b>0.0023</b>
<i>sa3_main1</i>	4000	1000	4.0	0.0097
<b>sum type 3</b>	<b>4000</b>	<b>1000</b>	<b>4.0</b>	<b>0.0097</b>
<b>Sum ShiftAdd Operations</b>	<b>4957</b>	<b>1313</b>	<b>3.8</b>	<b>0.0120</b>
<b>Streams</b>	<b>9258</b>	<b>1037</b>	<b>8.9</b>	<b>0.0224</b>

**Total Number of Cycles**                      **414032**

**Memory Access [ops.]**                      **96823**

*Store Operations*                              34687

*Load Operations*                              62136

<b>Adaptive Huffman Decode (Gamma)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo1_insert_node1</i>	54	18	3.0	0.0001
<i>bfo1_ext_count2</i>	10850	5425	2.0	0.0259
<b>Sum type 1</b>	<b>10904</b>	<b>5443</b>	<b>2.0</b>	<b>0.0260</b>
<i>bfo2_insert_node2</i>	54	18	3.0	0.0001
<i>bfo2_ext_count1</i>	16275	5425	3.0	0.0389
<i>bfo2_ins_count2</i>	10335	3445	3.0	0.0247
<b>Sum Type 2</b>	<b>26664</b>	<b>8888</b>	<b>3.0</b>	<b>0.0637</b>
<i>bfo3_ins_count1</i>	17225	3445	5.0	0.0411
<b>Sum Type 3</b>	<b>17225</b>	<b>3445</b>	<b>5.0</b>	<b>0.0411</b>
<i>bfo5_switch_nodes2</i>	164	41	4.0	0.0004
<i>bfo5_switch_nodes4</i>	123	41	3.0	0.0003
<i>bfo5_switch_nodes6</i>	222	74	3.0	0.0005
<i>bfo5_switch_nodes8</i>	222	74	3.0	0.0005
<i>bfo5_ins_count3</i>	10335	3445	3.0	0.0247
<b>Sum type 5</b>	<b>11066</b>	<b>3675</b>	<b>3.0</b>	<b>0.0264</b>
<i>bfo9_switch_nodes1</i>	164	41	4.0	0.0004
<i>bfo9_switch_nodes3</i>	164	41	4.0	0.0004
<i>bfo9_switch_nodes5</i>	296	74	4.0	0.0007
<i>bfo9_switch_nodes7</i>	296	74	4.0	0.0007
<i>bfo9_sort_tree1</i>	13780	3445	4.0	0.0329
<i>bfo9_increment_tree1</i>	13780	3445	4.0	0.0329
<b>Sum type 9</b>	<b>28480</b>	<b>7120</b>	<b>4.0</b>	<b>0.0680</b>
<b>Sum Bit Field Operations</b>	<b>94339</b>	<b>28571</b>	<b>3.3</b>	<b>0.2253</b>
<i>sa2_insert_node1</i>	54	18	3.0	0.0001
<i>sa2_switch_nodes1</i>	465	155	3.0	0.0011
<i>sa2_insert_node2</i>	72	18	4.0	0.0002
<i>sa2_switch_nodes2</i>	366	122	3.0	0.0009
<b>Sum type 2</b>	<b>957</b>	<b>313</b>	<b>3.1</b>	<b>0.0023</b>
<b>Sum ShiftAdd Operations</b>	<b>957</b>	<b>313</b>	<b>3.1</b>	<b>0.0023</b>
<b>Streams</b>	<b>13673</b>	<b>1018</b>	<b>13.4</b>	<b>0.0327</b>

**Total Number of Cycles**                      **418697**

**Memory Access [ops.]**                      **97328**

*Store Operations*                              34484

*Load Operations*                              62844

<b>Adaptive Huffman Encode (Poisson)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo1_insert_node1</i>	60	20	3.0	0.0001
<i>bfo1_ext_count2</i>	12070	6035	2.0	0.0266
<b>sum type 1</b>	<b>12130</b>	<b>6055</b>	<b>2.0</b>	<b>0.0267</b>
<i>bfo2_insert_node2</i>	60	20	3.0	0.0001
<i>bfo2_ext_count1</i>	18105	6035	3.0	0.0399
<i>bfo2_ins_count2</i>	11181	3727	3.0	0.0246
<b>sum type 2</b>	<b>29346</b>	<b>9782</b>	<b>3.0</b>	<b>0.0647</b>
<i>bfo3_ins_count1</i>	18635	3727	5.0	0.0411
<b>sum type 3</b>	<b>18635</b>	<b>3727</b>	<b>5.0</b>	<b>0.0411</b>
<i>bfo5_switch_nodes2</i>	213	71	3.0	0.0005
<i>bfo5_switch_nodes4</i>	213	71	3.0	0.0005
<i>bfo5_switch_nodes6</i>	324	108	3.0	0.0007
<i>bfo5_switch_nodes8</i>	324	108	3.0	0.0007
<i>bfo5_ins_count3</i>	11181	3727	3.0	0.0246
<b>sum type 5</b>	<b>12255</b>	<b>4085</b>	<b>3.0</b>	<b>0.0270</b>
<i>bfo9_switch_nodes1</i>	284	71	4.0	0.0006
<i>bfo9_switch_nodes3</i>	284	71	4.0	0.0006
<i>bfo9_switch_nodes5</i>	432	108	4.0	0.0010
<i>bfo9_switch_nodes7</i>	432	108	4.0	0.0010
<i>bfo9_encode1</i>	15020	3755	4.0	0.0331
<i>bfo9_sort_tree1</i>	14908	3727	4.0	0.0329
<i>bfo9_increment_tree1</i>	14908	3727	4.0	0.0329
<b>sum type 9</b>	<b>46268</b>	<b>11567</b>	<b>4.0</b>	<b>0.1020</b>
<b>Sum Bit Field Operations</b>	<b>118634</b>	<b>35216</b>	<b>3.4</b>	<b>0.2615</b>
<i>sa2_insert_node1</i>	60	20	3.0	0.0001
<i>sa2_insert_node2</i>	80	20	4.0	0.0002
<i>sa2_switch_nodes1</i>	672	224	3.0	0.0015
<i>sa2_switch_nodes2</i>	561	187	3.0	0.0012
<b>sum type 2</b>	<b>1373</b>	<b>451</b>	<b>3.0</b>	<b>0.0030</b>
<i>sa3_main1</i>	4000	1000	4.0	0.0088
<b>sum type 3</b>	<b>4000</b>	<b>1000</b>	<b>4.0</b>	<b>0.0088</b>
<b>Sum ShiftAdd Operations</b>	<b>0</b>	<b>0</b>	<b>3.8</b>	<b>0.0207</b>
<b>Streams</b>	<b>9536</b>	<b>1020</b>	<b>9.3</b>	<b>0.0210</b>

**Total Number of Cycles**                      **453720**

**Memory Access [ops.]**                      **106383**

*Store Operations*                              38123

*Load Operations*                              68260

<b>Adaptive Huffman Decode (Poisson)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo1_insert_node1</i>	60	20	3.0	0.0001
<i>bfo1_ext_count2</i>	12062	6031	2.0	0.0263
<b>Sum type 1</b>	<b>12122</b>	<b>6051</b>	<b>2.0</b>	<b>0.0264</b>
<i>bfo2_insert_node2</i>	60	20	3.0	0.0001
<i>bfo2_ext_count1</i>	18093	6031	3.0	0.0394
<i>bfo2_ins_count2</i>	11172	3724	3.0	0.0243
<b>Sum Type 2</b>	<b>29325</b>	<b>9775</b>	<b>3.0</b>	<b>0.0638</b>
<i>bfo3_ins_count1</i>	18620	3724	5.0	0.0405
<b>Sum Type 3</b>	<b>18620</b>	<b>3724</b>	<b>5.0</b>	<b>0.0405</b>
<i>bfo5_switch_nodes2</i>	213	71	3.0	0.0005
<i>bfo5_switch_nodes4</i>	213	71	3.0	0.0005
<i>bfo5_switch_nodes6</i>	324	108	3.0	0.0007
<i>bfo5_switch_nodes8</i>	324	108	3.0	0.0007
<i>bfo5_ins_count3</i>	11172	3724	3.0	0.0243
<b>Sum type 5</b>	<b>12246</b>	<b>4082</b>	<b>3.0</b>	<b>0.0267</b>
<i>bfo9_switch_nodes1</i>	284	71	4.0	0.0006
<i>bfo9_switch_nodes3</i>	284	71	4.0	0.0006
<i>bfo9_switch_nodes5</i>	432	108	4.0	0.0009
<i>bfo9_switch_nodes7</i>	432	108	4.0	0.0009
<i>bfo9_sort_tree1</i>	14896	3724	4.0	0.0324
<i>bfo9_increment_tree1</i>	14896	3724	4.0	0.0324
<b>Sum type 9</b>	<b>31224</b>	<b>7806</b>	<b>4.0</b>	<b>0.0680</b>
<b>Sum Bit Field Operations</b>	<b>103537</b>	<b>31438</b>	<b>3.3</b>	<b>0.2254</b>
<i>sa2_insert_node1</i>	60	20	3.0	0.0001
<i>sa2_switch_nodes1</i>	675	225	3.0	0.0015
<i>sa2_insert_node2</i>	80	20	4.0	0.0002
<i>sa2_switch_nodes2</i>	564	188	3.0	0.0012
<b>Sum type 2</b>	<b>1379</b>	<b>453</b>	<b>3.0</b>	<b>0.0030</b>
<b>Sum ShiftAdd Operations</b>	<b>1379</b>	<b>453</b>	<b>3.0</b>	<b>0.0030</b>
<b>Streams</b>	<b>13904</b>	<b>1020</b>	<b>13.6</b>	<b>0.0303</b>

**Total Number of Cycles**                      **459299**

**Memory Access [ops.]**                      **107116**

*Store Operations*                              37888

*Load Operations*                              69228

<b>Adaptive Huffman Encode (Text)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo1_insert_node1</i>	111	37	3.0	0.0002
<i>bfo1_ext_count2</i>	14510	7255	2.0	0.0267
<b>sum type 1</b>	<b>14621</b>	<b>7292</b>	<b>2.0</b>	<b>0.0269</b>
<i>bfo2_insert_node2</i>	111	37	3.0	0.0002
<i>bfo2_ext_count1</i>	21765	7255	3.0	0.0401
<i>bfo2_ins_count2</i>	12801	4267	3.0	0.0236
<b>sum type 2</b>	<b>34677</b>	<b>11559</b>	<b>3.0</b>	<b>0.0639</b>
<i>bfo3_ins_count1</i>	21335	4267	5.0	0.0393
<b>sum type 3</b>	<b>21335</b>	<b>4267</b>	<b>5.0</b>	<b>0.0393</b>
<i>bfo5_switch_nodes2</i>	465	155	3.0	0.0009
<i>bfo5_switch_nodes4</i>	465	155	3.0	0.0009
<i>bfo5_switch_nodes6</i>	648	216	3.0	0.0012
<i>bfo5_switch_nodes8</i>	648	216	3.0	0.0012
<i>bfo5_ins_count3</i>	12801	4267	3.0	0.0236
<b>sum type 5</b>	<b>15027</b>	<b>5009</b>	<b>3.0</b>	<b>0.0277</b>
<i>bfo9_switch_nodes1</i>	620	155	4.0	0.0011
<i>bfo9_switch_nodes3</i>	620	155	4.0	0.0011
<i>bfo9_switch_nodes5</i>	864	216	4.0	0.0016
<i>bfo9_switch_nodes7</i>	864	216	4.0	0.0016
<i>bfo9_encode1</i>	17160	4290	4.0	0.0316
<i>bfo9_sort_tree1</i>	17068	4267	4.0	0.0315
<i>bfo9_increment_tree1</i>	17068	4267	4.0	0.0315
<b>sum type 9</b>	<b>54264</b>	<b>13566</b>	<b>4.0</b>	<b>0.1000</b>
<b>Sum Bit Field Operations</b>	<b>139924</b>	<b>41693</b>	<b>3.4</b>	<b>0.2579</b>
<i>sa2_insert_node1</i>	111	37	3.0	0.0002
<i>sa2_insert_node2</i>	148	37	4.0	0.0003
<i>sa2_switch_nodes1</i>	1041	347	3.0	0.0019
<i>sa2_switch_nodes2</i>	858	286	3.0	0.0016
<b>sum type 2</b>	<b>2158</b>	<b>707</b>	<b>3.1</b>	<b>0.0040</b>
<i>sa3_main1</i>	4000	1000	4.0	0.0074
<b>sum type 3</b>	<b>4000</b>	<b>1000</b>	<b>4.0</b>	<b>0.0074</b>
<b>Sum ShiftAdd Operations</b>	<b>6158</b>	<b>1707</b>	<b>3.6</b>	<b>0.0114</b>
<b>Streams</b>	<b>10226</b>	<b>1037</b>	<b>9.9</b>	<b>0.0188</b>

**Total Number of Cycles**                      **542537**

**Memory Access [ops.]**                      **127339**

*Store Operations*                              45099

*Load Operations*                              82240

<b>Adaptive Huffman Decode (Text)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo1_insert_node1</i>	111	37	3.0	0.0002
<i>bfo1_ext_count2</i>	14512	7256	2.0	0.0264
<b>Sum type 1</b>	<b>14623</b>	<b>7293</b>	<b>2.0</b>	<b>0.0266</b>
<i>bfo2_insert_node2</i>	111	37	3.0	0.0002
<i>bfo2_ext_count1</i>	21768	7256	3.0	0.0395
<i>bfo2_ins_count2</i>	12801	4267	3.0	0.0233
<b>Sum Type 2</b>	<b>34680</b>	<b>11560</b>	<b>3.0</b>	<b>0.0630</b>
<i>bfo3_ins_count1</i>	21335	4267	5.0	0.0388
<b>Sum Type 3</b>	<b>21335</b>	<b>4267</b>	<b>5.0</b>	<b>0.0388</b>
<i>bfo5_switch_nodes2</i>	465	155	3.0	0.0008
<i>bfo5_switch_nodes4</i>	465	155	3.0	0.0008
<i>bfo5_switch_nodes6</i>	648	216	3.0	0.0012
<i>bfo5_switch_nodes8</i>	648	216	3.0	0.0012
<i>bfo5_ins_count3</i>	12801	4267	3.0	0.0233
<b>Sum type 5</b>	<b>15027</b>	<b>5009</b>	<b>3.0</b>	<b>0.0273</b>
<i>bfo9_switch_nodes1</i>	620	155	4.0	0.0011
<i>bfo9_switch_nodes3</i>	620	155	4.0	0.0011
<i>bfo9_switch_nodes5</i>	864	216	4.0	0.0016
<i>bfo9_switch_nodes7</i>	864	216	4.0	0.0016
<i>bfo9_sort_tree1</i>	17068	4267	4.0	0.0310
<i>bfo9_increment_tree1</i>	17068	4267	4.0	0.0310
<b>Sum type 9</b>	<b>37104</b>	<b>9276</b>	<b>4.0</b>	<b>0.0674</b>
<b>Sum Bit Field Operations</b>	<b>122769</b>	<b>37405</b>	<b>3.3</b>	<b>0.2230</b>
<i>sa2_insert_node1</i>	111	37	3.0	0.0002
<i>sa2_switch_nodes1</i>	1041	347	3.0	0.0019
<i>sa2_insert_node2</i>	148	37	4.0	0.0003
<i>sa2_switch_nodes2</i>	858	286	3.0	0.0016
<b>Sum type 2</b>	<b>2158</b>	<b>707</b>	<b>3.1</b>	<b>0.0039</b>
<b>Sum ShiftAdd Operations</b>	<b>2158</b>	<b>707</b>	<b>3.1</b>	<b>0.0039</b>
<b>Streams</b>	<b>14553</b>	<b>1037</b>	<b>14.0</b>	<b>0.0264</b>

**Total Number of Cycles**                      **550573**

**Memory Access [ops.]**                      **128674**

*Store Operations*                              44851

*Load Operations*                              83823



Deflate Encode (Exponential)				
Label	Cycles	Calls	Cycles/Calls	Cycles/Tot.Cycles
<i>bfo1_make_code2</i>	0	0	#DIV/0!	0
<i>bfo1_make_code3</i>	236	118	2	0.0005
<i>bfo1_make_code4</i>	90	45	2	0.0002
<b>Sum Type 1</b>	<b>326</b>	<b>163</b>	<b>2</b>	<b>0.0006</b>
<b>Sum Bit Field Operations</b>	<b>326</b>	<b>163</b>	<b>2</b>	<b>0.0006</b>
<i>sa3_encode1</i>	645	215	3	0.0013
<i>sa3_add_match1</i>	3000	1000	3	0.0058
<b>Sum type 3</b>	<b>3645</b>	<b>1215</b>	<b>3</b>	<b>0.0071</b>
<b>Sum ShiftAdd Operations</b>	<b>3645</b>	<b>1215</b>	<b>3</b>	<b>0.0071</b>
<b>Streams</b>	<b>4800</b>	<b>380</b>	<b>12.6</b>	<b>0.0093</b>
<b>CRC</b>	<b>31000</b>	<b>1000</b>	<b>31.0</b>	<b>0.0602</b>

**Total Number of Cycles**                   **515066**

**Memory Access [ops.]**                   **81537**  
*Store Operations*                           11751  
*Load Operations*                            69786

Deflate Decode (Exponential)				
Label	Cycles	Calls	Cycles/Calls	Cycles/Tot.Cycles
<i>bfo4_dec_stream3</i>	156	52	3	0.0083
<b>Sum Type 4</b>	<b>156</b>	<b>52</b>	<b>3</b>	<b>0.0083</b>
<i>bfo10_dec_stream4</i>	236	118	2	0.0126
<i>bfo10_dec_stream5</i>	90	45	2	0.0048
<i>bfo10_dec_stream6</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 10</b>	<b>326</b>	<b>163</b>	<b>2</b>	<b>0.0173</b>
<i>bfo12_dec_stream1</i>	864	216	4	0.0460
<b>Sum Type 12</b>	<b>864</b>	<b>216</b>	<b>4</b>	<b>0.0460</b>
<b>Sum Bit Field Operations</b>	<b>1346</b>	<b>431</b>	<b>3.1</b>	<b>0.0716</b>
<b>Streams</b>	<b>6116</b>	<b>378</b>	<b>16.2</b>	<b>0.3253</b>

**Total Number of Cycles**                   **18802**

**Memory Access [ops.]**                   **3315**  
*Store Operations*                           1594  
*Load Operations*                            1721

Deflate Encode (Gamma)				
Label	Cycles	Calls	Cycles/Calls	Cycles/Tot.Cycles
<i>bfo1_make_code2</i>	0	0	#DIV/0!	0
<i>bfo1_make_code3</i>	360	180	2	0.0026
<i>bfo1_make_code4</i>	0	0	#DIV/0!	0
<b>Sum Type 1</b>	<b>360</b>	<b>180</b>	<b>2</b>	<b>0.0026</b>
<b>Sum Bit Field Operations</b>	<b>360</b>	<b>180</b>	<b>2</b>	<b>0.0026</b>
<i>sa3_encode1</i>	1824	608	3	0.0133
<i>sa3_add_match1</i>	3000	1000	3	0.0219
<b>Sum type 3</b>	<b>4824</b>	<b>1608</b>	<b>3</b>	<b>0.0351</b>
<b>Sum ShiftAdd Operations</b>	<b>4824</b>	<b>1608</b>	<b>3</b>	<b>0.0351</b>
<b>Streams</b>	<b>11040</b>	<b>790</b>	<b>14.0</b>	<b>0.0804</b>
<b>CRC</b>	<b>31000</b>	<b>1000</b>	<b>31.0</b>	<b>0.2258</b>

**Total Number of Cycles**                   **137286**

**Memory Access [ops.]**                   **30322**

*Store Operations*                       9564

*Load Operations*                       20758

Deflate Decode (Gamma)				
Label	Cycles	Calls	Cycles/Calls	Cycles/Tot.Cycles
<i>bfo4_dec_stream3</i>	1287	429	3	0.0403
<b>Sum Type 4</b>	<b>1287</b>	<b>429</b>	<b>3</b>	<b>0.0403</b>
<i>bfo10_dec_stream4</i>	360	180	2	0.0113
<i>bfo10_dec_stream5</i>	0	0	#DIV/0!	0.0000
<i>bfo10_dec_stream6</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 10</b>	<b>360</b>	<b>180</b>	<b>2</b>	<b>0.0113</b>
<i>bfo12_dec_stream1</i>	2436	609	4	0.0763
<b>Sum Type 12</b>	<b>2436</b>	<b>609</b>	<b>4</b>	<b>0.0763</b>
<b>Sum Bit Field Operations</b>	<b>4083</b>	<b>1218</b>	<b>3.4</b>	<b>0.1278</b>
<b>Streams</b>	<b>13607</b>	<b>788</b>	<b>17.3</b>	<b>0.4259</b>

**Total Number of Cycles**                   **31946**

**Memory Access [ops.]**                   **4816**

*Store Operations*                       2397

*Load Operations*                       2419

Deflate Encode (Poisson)				
Label	Cycles	Calls	Cycles/Calls	Cycles/Tot.Cycles
<i>bfo1_make_code2</i>	0	0	#DIV/0!	0
<i>bfo1_make_code3</i>	292	146	2	0.0021
<i>bfo1_make_code4</i>	0	0	#DIV/0!	0
<b>Sum Type 1</b>	<b>292</b>	<b>146</b>	<b>2</b>	<b>0.0021</b>
<b>Sum Bit Field Operations</b>	<b>292</b>	<b>146</b>	<b>2</b>	<b>0.0021</b>
<i>sa3_encode1</i>	2058	686	3	0.0149
<i>sa3_add_match1</i>	3000	1000	3	0.0217
<b>Sum type 3</b>	<b>5058</b>	<b>1686</b>	<b>3</b>	<b>0.0366</b>
<b>Sum ShiftAdd Operations</b>	<b>5058</b>	<b>1686</b>	<b>3</b>	<b>0.0366</b>
<b>Streams</b>	<b>12004</b>	<b>834</b>	<b>14.4</b>	<b>0.0869</b>
<b>CRC</b>	<b>31000</b>	<b>1000</b>	<b>31.0</b>	<b>0.2245</b>

**Total Number of Cycles**                    **138101**

**Memory Access [ops.]**                    **30941**

*Store Operations*                        9898

*Load Operations*                        21043

Deflate Decode (Poisson)				
Label	Cycles	Calls	Cycles/Calls	Cycles/Tot.Cycles
<i>bfo4_dec_stream3</i>	1620	540	3	0.0483
<b>Sum Type 4</b>	<b>1620</b>	<b>540</b>	<b>3</b>	<b>0.0483</b>
<i>bfo10_dec_stream4</i>	292	146	2	0.0087
<i>bfo10_dec_stream5</i>	0	0	#DIV/0!	0.0000
<i>bfo10_dec_stream6</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 10</b>	<b>292</b>	<b>146</b>	<b>2</b>	<b>0.0087</b>
<i>bfo12_dec_stream1</i>	2748	687	4	0.0820
<b>Sum Type 12</b>	<b>2748</b>	<b>687</b>	<b>4</b>	<b>0.0820</b>
<b>Sum Bit Field Operations</b>	<b>4660</b>	<b>1373</b>	<b>3.4</b>	<b>0.1390</b>
<b>Streams</b>	<b>14630</b>	<b>832</b>	<b>17.6</b>	<b>0.4365</b>

**Total Number of Cycles**                    **33516**

**Memory Access [ops.]**                    **4997**

*Store Operations*                        2519

*Load Operations*                        2478

Deflate Encode (Text)				
Label	Cycles	Calls	Cycles/Calls	Cycles/Tot.Cycles
<i>bfo1_make_code2</i>	0	0	#DIV/0!	0
<i>bfo1_make_code3</i>	194	97	2	0.0014
<i>bfo1_make_code4</i>	30	15	2	0.000213584
<b>Sum Type 1</b>	<b>224</b>	<b>112</b>	<b>2</b>	<b>0.0016</b>
<b>Sum Bit Field Operations</b>	<b>224</b>	<b>112</b>	<b>2</b>	<b>0.0016</b>
<i>sa3_encode1</i>	1866	622	3	0.0133
<i>sa3_add_match1</i>	3000	1000	3	0.0214
<b>Sum type 3</b>	<b>4866</b>	<b>1622</b>	<b>3</b>	<b>0.0346</b>
<b>Sum ShiftAdd Operations</b>	<b>4866</b>	<b>1622</b>	<b>3</b>	<b>0.0346</b>
<b>Streams</b>	<b>10716</b>	<b>736</b>	<b>14.6</b>	<b>0.0763</b>
<b>CRC</b>	<b>31000</b>	<b>1000</b>	<b>31.0</b>	<b>0.2207</b>

**Total Number of Cycles**                   **140460**

**Memory Access [ops.]**                   **31169**

*Store Operations*                       21484

*Load Operations*                       9685

Deflate Decode (Text)				
Label	Cycles	Calls	Cycles/Calls	Cycles/Tot.Cycles
<i>bfo4_dec_stream3</i>	1530	510	3	0.0501
<b>Sum Type 4</b>	<b>1530</b>	<b>510</b>	<b>3</b>	<b>0.0501</b>
<i>bfo10_dec_stream4</i>	194	97	2	0.0064
<i>bfo10_dec_stream5</i>	30	15	2	0.0010
<i>bfo10_dec_stream6</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 10</b>	<b>224</b>	<b>112</b>	<b>2</b>	<b>0.0073</b>
<i>bfo12_dec_stream1</i>	2492	623	4	0.0817
<b>Sum Type 12</b>	<b>2492</b>	<b>623</b>	<b>4</b>	<b>0.0817</b>
<b>Sum Bit Field Operations</b>	<b>4246</b>	<b>1245</b>	<b>3.4</b>	<b>0.1392</b>
<b>Streams</b>	<b>13013</b>	<b>734</b>	<b>17.7</b>	<b>0.4265</b>

**Total Number of Cycles**                   **30513**

**Memory Access [ops.]**                   **4654**

*Store Operations*                       2357

*Load Operations*                       2297

<b>Rice Encode (Exponential)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo6_maintain_tables1</i>	64	16	4.0	0.0008
<i>bfo6_get_byte1</i>	3996	999	4.0	0.0508
<b>Sum Type 6</b>	<b>4060</b>	<b>1015</b>	<b>4.0</b>	<b>0.0516</b>
<i>bfo8_encode1</i>	4995	999	5.0	0.0634
<b>Sum Type 8</b>	<b>4995</b>	<b>999</b>	<b>5.0</b>	<b>0.0634</b>
<b>Bit Field Operations</b>	<b>9055</b>	<b>2014</b>	<b>4.5</b>	<b>0.1150</b>
<i>sa1_comp_freq1</i>	1158	386	3.0	0.0147
<b>Sum Type 1</b>	<b>1158</b>	<b>386</b>	<b>3.0</b>	<b>0.0147</b>
<i>sa3_get_byte1</i>	2997	999	3.0	0.0381
<b>Sum Type 3</b>	<b>2997</b>	<b>999</b>	<b>3.0</b>	<b>0.0381</b>
<i>sa4_encode1</i>	2997	999	3.0	0.0381
<i>sa4_switch1</i>	42	14	3.0	0.0005
<b>Sum Type 4</b>	<b>3039</b>	<b>1013</b>	<b>3.0</b>	<b>0.0386</b>
<i>sa5_maintain_tables</i>	42	14	3.0	0.0005
<b>Sum Type 5</b>	<b>42</b>	<b>14</b>	<b>3.0</b>	<b>0.0005</b>
<b>Sum ShiftAdd Operations</b>	<b>7236</b>	<b>2412</b>	<b>3.0</b>	<b>0.0919</b>
<b>CLZ</b>	<b>434</b>	<b>62</b>	<b>7.0</b>	<b>0.0055</b>
<b>Streams</b>	<b>13436</b>	<b>1999</b>	<b>6.7</b>	<b>0.1707</b>

**Total Number of Cycles**                      **78729**

**Memory Access [ops.]**                      **14768**

*Store Operations*                              7259

*Load Operations*                              7509

<b>Rice Decode (Exponential)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo6_switch1</i>	56	14	4.0	0.0006
<i>bfo6_decode_loop1</i>	3988	997	4.0	0.0407
<i>bfo6_maintain_tables1</i>	64	16	4.0	0.0007
<b>Sum Type 6</b>	<b>4108</b>	<b>1027</b>	4.0	0.0420
<i>bfo7_decode1</i>	4985	997	5.0	0.0509
<b>Sum Type 7</b>	<b>4985</b>	<b>997</b>	5.0	0.0509
<b>Bit Field Operations</b>	<b>9093</b>	<b>2024</b>	<b>4.5</b>	<b>0.0929</b>
<i>sa1_comp_freq1</i>	2991	997	3.0	0.0305
<b>Sum Type 1</b>	<b>2991</b>	<b>997</b>	3.0	0.0305
<i>sa4_switch</i>	42	14	3.0	0.0004
<i>sa4_comp_freq2</i>	3988	997	4.0	0.0407
<b>Sum Type 4</b>	<b>4030</b>	<b>1011</b>	4.0	0.0412
<i>sa5_maintain_tables1</i>	142	14	10.1	0.0015
<b>Sum Type 5</b>	<b>142</b>	<b>14</b>	10.1	0.0015
<b>Shift Add Operations</b>	<b>7163</b>	<b>2022</b>	<b>3.5</b>	<b>0.0732</b>
<i>clz1</i>	12785	997	12.8	0.1306
<i>clz2</i>	434	62	7.0	0.0044
<b>CLZ</b>	<b>13219</b>	<b>1059</b>	<b>12.5</b>	<b>0.1350</b>
<b>Streams</b>	<b>23056</b>	<b>1994</b>	<b>11.6</b>	<b>0.2355</b>

**Total Number of Cycles**                    **97912**

**Memory Access [ops.]**                    **14758**

*Store Operations*                        *8156*

*Load Operations*                        *6602*

<b>Rice Encode (Gamma)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo6_maintain_tables1</i>	0	0	#DIV/0!	0.0000
<i>bfo6_get_byte1</i>	3996	999	4	0.0438
<b>Sum Type 6</b>	<b>3996</b>	<b>999</b>	<b>4</b>	<b>0.0438</b>
<i>bfo8_encode1</i>	4995	999	5	0.0547
<b>Sum Type 8</b>	<b>4995</b>	<b>999</b>	<b>5</b>	<b>0.0547</b>
<b>Bit Field Operations</b>	<b>8991</b>	<b>1998</b>	<b>4.5</b>	<b>0.0985</b>
<i>sa1_comp_freq1</i>	2493	831	3	0.0273
<b>Sum Type 1</b>	<b>2493</b>	<b>831</b>	<b>3</b>	<b>0.0273</b>
<i>sa3_get_byte1</i>	2997	999	3	0.0328
<b>Sum Type 3</b>	<b>2997</b>	<b>999</b>	<b>3</b>	<b>0.0328</b>
<i>sa4_encode1</i>	2997	999	3	0.0328
<i>sa4_switch1</i>	321	107	3	0.0035
<b>Sum Type 4</b>	<b>3318</b>	<b>1106</b>	<b>3</b>	<b>0.0363</b>
<i>sa5_maintain_tables</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 5</b>	<b>0</b>	<b>0</b>	<b>#DIV/0!</b>	<b>0.0000</b>
<b>Sum ShiftAdd Operations</b>	<b>8808</b>	<b>2936</b>	<b>3</b>	<b>0.0964</b>
<b>CLZ</b>	<b>3276</b>	<b>62</b>	<b>52.8</b>	<b>0.0359</b>
<b>Streams</b>	<b>15186</b>	<b>1999</b>	<b>7.6</b>	<b>0.1663</b>

**Total Number of Cycles**                      **91322**

**Memory Access [ops.]**                      **15971**

*Store Operations*                              7821

*Load Operations*                              8150

<b>Rice Decode (Gamma)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo6_switch1</i>	428	107	4.0	0.0039
<i>bfo6_decode_loop1</i>	3988	997	4.0	0.0360
<i>bfo6_maintain_tables1</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 6</b>	<b>4416</b>	<b>1104</b>	<b>4.0</b>	<b>0.0398</b>
<i>bfo7_decode1</i>	4985	997	5.0	0.0449
<b>Sum Type 7</b>	<b>4985</b>	<b>997</b>	<b>5.0</b>	<b>0.0449</b>
<b>Bit Field Operations</b>	<b>9401</b>	<b>2101</b>	<b>4.5</b>	<b>0.0847</b>
<i>sa1_comp_freq1</i>	2991	997	3.0	0.0270
<b>Sum Type 1</b>	<b>2991</b>	<b>997</b>	<b>3.0</b>	<b>0.0270</b>
<i>sa4_switch</i>	321	107	3.0	0.0029
<i>sa4_comp_freq2</i>	3988	997	4.0	0.0360
<b>Sum Type 4</b>	<b>4309</b>	<b>1104</b>	<b>3.9</b>	<b>0.0388</b>
<i>sa5_maintain_tables1</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 5</b>	<b>0</b>	<b>0</b>	<b>#DIV/0!</b>	<b>0.0000</b>
<b>Shift Add Operations</b>	<b>7300</b>	<b>2101</b>	<b>3.5</b>	<b>0.0658</b>
<i>clz1</i>	14678	997	14.7	0.1323
<i>clz2</i>	3276	62	52.8	0.0295
<b>CLZ</b>	<b>17954</b>	<b>1059</b>	<b>17.0</b>	<b>0.1618</b>
<b>Streams</b>	<b>24420</b>	<b>1994</b>	<b>12.2</b>	<b>0.2201</b>

**Total Number of Cycles**                      **110930**

**Memory Access [ops.]**                      **16045**

*Store Operations*                              8682

*Load Operations*                              7363



<b>Rice Encode (Poisson)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo6_maintain_tables1</i>	0	0	#DIV/0!	0.0000
<i>bfo6_get_byte1</i>	3996	999	4	0.0428
<b>Sum Type 6</b>	<b>3996</b>	<b>999</b>	<b>4</b>	<b>0.0428</b>
<i>bfo8_encode1</i>	4995	999	5	0.0536
<b>Sum Type 8</b>	<b>4995</b>	<b>999</b>	<b>5</b>	<b>0.0536</b>
<b>Bit Field Operations</b>	<b>8991</b>	<b>1998</b>	<b>4.5</b>	<b>0.0964</b>
<i>sa1_comp_freq1</i>	2619	873	3	0.0281
<b>Sum Type 1</b>	<b>2619</b>	<b>873</b>	<b>3</b>	<b>0.0281</b>
<i>sa3_get_byte1</i>	2997	999	3	0.0321
<b>Sum Type 3</b>	<b>2997</b>	<b>999</b>	<b>3</b>	<b>0.0321</b>
<i>sa4_encode1</i>	2997	999	3	0.0321
<i>sa4_switch1</i>	417	139	3	0.0045
<b>Sum Type 4</b>	<b>3414</b>	<b>1138</b>	<b>3</b>	<b>0.0366</b>
<i>sa5_maintain_tables</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 5</b>	<b>0</b>	<b>0</b>	<b>#DIV/0!</b>	<b>0.0000</b>
<b>Sum ShiftAdd Operations</b>	<b>9030</b>	<b>3010</b>	<b>3</b>	<b>0.0968</b>
<b>CLZ</b>	<b>3204</b>	<b>62</b>	<b>51.7</b>	<b>0.0344</b>
<b>Streams</b>	<b>15424</b>	<b>1999</b>	<b>7.7</b>	<b>0.1654</b>

**Total Number of Cycles**                      **93275**

**Memory Access [ops.]**                      **16296**

*Store Operations*                              7998

*Load Operations*                             8298

<b>Rice Decode (Poisson)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo6_switch1</i>	556	139	4.0	0.0049
<i>bfo6_decode_loop1</i>	3988	997	4.0	0.0355
<i>bfo6_maintain_tables1</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 6</b>	<b>4544</b>	<b>1136</b>	<b>4.0</b>	<b>0.0404</b>
<i>bfo7_decode1</i>	4985	997	5.0	0.0443
<b>Sum Type 7</b>	<b>4985</b>	<b>997</b>	<b>5.0</b>	<b>0.0443</b>
<b>Bit Field Operations</b>	<b>9529</b>	<b>2133</b>	<b>4.5</b>	<b>0.0847</b>
<i>sa1_comp_freq1</i>	2991	997	3.0	0.0266
<b>Sum Type 1</b>	<b>2991</b>	<b>997</b>	<b>3.0</b>	<b>0.0266</b>
<i>sa4_switch</i>	417	139	3.0	0.0037
<i>sa4_comp_freq2</i>	3988	997	4.0	0.0355
<b>Sum Type 4</b>	<b>4405</b>	<b>1136</b>	<b>3.9</b>	<b>0.0392</b>
<i>sa5_maintain_tables1</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 5</b>	<b>0</b>	<b>0</b>	<b>#DIV/0!</b>	<b>0.0000</b>
<b>Shift Add Operations</b>	<b>7396</b>	<b>2133</b>	<b>3.5</b>	<b>0.0657</b>
<i>clz1</i>	14336	997	14.4	0.1274
<i>clz2</i>	3204	62	51.7	0.0285
<b>CLZ</b>	<b>17540</b>	<b>1059</b>	<b>16.6</b>	<b>0.1559</b>
<b>Streams</b>	<b>25036</b>	<b>1994</b>	<b>12.6</b>	<b>0.2226</b>

**Total Number of Cycles**                      **112490**

**Memory Access [ops.]**                      **16402**

*Store Operations*                              8874

*Load Operations*                              7528

<b>Rice Encode (Text)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo6_maintain_tables1</i>	0	0	#DIV/0!	0.0000
<i>bfo6_get_byte1</i>	3996	999	4	0.0329
<b>Sum Type 6</b>	<b>3996</b>	<b>999</b>	<b>4</b>	<b>0.0329</b>
<i>bfo8_encode1</i>	4995	999	5	0.0411
<b>Sum Type 8</b>	<b>4995</b>	<b>999</b>	<b>5</b>	<b>0.0411</b>
<b>Bit Field Operations</b>	<b>8991</b>	<b>1998</b>	<b>4.5</b>	<b>0.0740</b>
<i>sa1_comp_freq1</i>	2520	840	3	0.0207
<b>Sum Type 1</b>	<b>2520</b>	<b>840</b>	<b>3</b>	<b>0.0207</b>
<i>sa3_get_byte1</i>	2997	999	3	0.0247
<b>Sum Type 3</b>	<b>2997</b>	<b>999</b>	<b>3</b>	<b>0.0247</b>
<i>sa4_encode1</i>	2997	999	3	0.0247
<i>sa4_switch1</i>	699	233	3	0.0058
<b>Sum Type 4</b>	<b>3696</b>	<b>1232</b>	<b>3</b>	<b>0.0304</b>
<i>sa5_maintain_tables</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 5</b>	<b>0</b>	<b>0</b>	<b>#DIV/0!</b>	<b>0.0000</b>
<b>Sum ShiftAdd Operations</b>	<b>9213</b>	<b>3071</b>	<b>3</b>	<b>0.0758</b>
<b>CLZ</b>	<b>3066</b>	<b>62</b>	<b>49.5</b>	<b>0.0252</b>
<b>Streams</b>	<b>16514</b>	<b>2015</b>	<b>8.2</b>	<b>0.1359</b>

**Total Number of Cycles**                      **121546**

**Memory Access [ops.]**                      **19801**

*Store Operations*                              8555

*Load Operations*                              11246

<b>Rice Decode (Text)</b>				
<b>Label</b>	<b>Cycles</b>	<b>Calls</b>	<b>Cycles/Call</b>	<b>Cycles/Tot.Cycles</b>
<i>bfo6_switch1</i>	928	232	4.0	0.0066
<i>bfo6_decode_loop1</i>	3988	997	4.0	0.0282
<i>bfo6_maintain_tables1</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 6</b>	<b>4916</b>	<b>1229</b>	<b>4.0</b>	<b>0.0347</b>
<i>bfo7_decode1</i>	4985	997	5.0	0.0352
<b>Sum Type 7</b>	<b>4985</b>	<b>997</b>	<b>5.0</b>	<b>0.0352</b>
<b>Bit Field Operations</b>	<b>9901</b>	<b>2226</b>	<b>4.4</b>	<b>0.0699</b>
<i>sa1_comp_freq1</i>	2991	997	3.0	0.0211
<b>Sum Type 1</b>	<b>2991</b>	<b>997</b>	<b>3.0</b>	<b>0.0211</b>
<i>sa4_switch</i>	696	232	3.0	0.0049
<i>sa4_comp_freq2</i>	3988	997	4.0	0.0282
<b>Sum Type 4</b>	<b>4684</b>	<b>1229</b>	<b>3.8</b>	<b>0.0331</b>
<i>sa5_maintain_tables1</i>	0	0	#DIV/0!	0.0000
<b>Sum Type 5</b>	<b>0</b>	<b>0</b>	<b>#DIV/0!</b>	<b>0.0000</b>
<b>Shift Add Operations</b>	<b>7675</b>	<b>2226</b>	<b>3.4</b>	<b>0.0542</b>
<i>clz1</i>	14944	1013	14.8	0.1055
<i>clz2</i>	3066	62	49.5	0.0217
<b>CLZ</b>	<b>18010</b>	<b>1075</b>	<b>16.8</b>	<b>0.1272</b>
<b>Streams</b>	<b>25036</b>	<b>1994</b>	<b>12.6</b>	<b>0.1768</b>

**Total Number of Cycles**                    **141604**

**Memory Access [ops.]**                    **20025**

*Store Operations*                        *9464*

*Load Operations*                        *10561*

## D. Algorithmic Level Profiling

### With Original Instruction Set

Adaptive Huffman Encode (Exponential)		
Label	Cycles	Cycles/Tot.Cycles
<b>Increment Tree</b>	<b>124230</b>	<b>63.69%</b>
<i>Sort Tree</i>	47400	24.30%
<i>Switch Nodes</i>	2078	1.07%
<b>Insert Node</b>	<b>408</b>	<b>0.21%</b>
<b>Encode</b>	<b>40776</b>	<b>20.90%</b>
<b>Streams</b>	<b>7504</b>	<b>3.85%</b>

**Total Number of Cycles**                    **195059**

**Memory Access [ops.]**                    **45321**  
*Store Operations*                            16963  
*Load Operations*                            28358

Adaptive Huffman Decode (Exponential)		
Label	Cycles	Cycles/Tot.Cycles
<b>Increment Tree</b>	<b>128037</b>	<b>64.28%</b>
<i>Sort Tree</i>	49413	24.81%
<i>Switch Nodes</i>	2402	1.21%
<b>Insert Node</b>	<b>408</b>	<b>0.20%</b>
<b>Decode</b>	<b>57496</b>	<b>28.87%</b>
<b>Streams</b>	<b>12265</b>	<b>6.16%</b>

**Total Number of Cycles**                    **199180**

**Memory Access [ops.]**                    **46966**  
*Store Operations*                            18205  
*Load Operations*                            28761

<b>Adaptive Huffman Encode (Gamma)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>309969</b>	<b>75.40%</b>
<i>Sort Tree</i>	148499	36.12%
<i>Switch Nodes</i>	10208	2.48%
<b>Insert Node</b>	<b>918</b>	<b>0.22%</b>
<b>Encode</b>	<b>68670</b>	<b>16.70%</b>
<b>Streams</b>	<b>9258</b>	<b>2.25%</b>

**Total Number of Cycles**                    **411086**

**Memory Access [ops.]**                    **94859**  
     *Store Operations*                    33705  
     *Load Operations*                    61154

<b>Adaptive Huffman Decode (Gamma)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>310020</b>	<b>74.04%</b>
<i>Sort Tree</i>	148550	35.48%
<i>Switch Nodes</i>	10208	2.44%
<b>Insert Node</b>	<b>918</b>	<b>0.22%</b>
<b>Decode</b>	<b>94201</b>	<b>22.50%</b>
<b>Streams</b>	<b>13673</b>	<b>3.27%</b>

**Total Number of Cycles**                    **418697**

**Memory Access [ops.]**                    **97328**  
     *Store Operations*                    34484  
     *Load Operations*                    62844

<b>Adaptive Huffman Encode (Poisson)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>344957</b>	<b>76.52%</b>
<i>Sort Tree</i>	170515	37.83%
<i>Switch Nodes</i>	15506	3.44%
<b>Insert Node</b>	<b>1020</b>	<b>0.23%</b>
<b>Encode</b>	<b>72970</b>	<b>16.19%</b>
<b>Streams</b>	<b>9536</b>	<b>2.12%</b>

**Total Number of Cycles**                    **450780**

**Memory Access [ops.]**                    **104423**  
     *Store Operations*                    37143  
     *Load Operations*                    67280

<b>Adaptive Huffman Decode (Poisson)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>345777</b>	<b>75.28%</b>
<i>Sort Tree</i>	174197	37.93%
<i>Switch Nodes</i>	15840	3.45%
<b>Insert Node</b>	<b>1040</b>	<b>0.23%</b>
<b>Decode</b>	<b>100900</b>	<b>21.97%</b>
<b>Streams</b>	<b>14924</b>	<b>3.25%</b>
<b>Total Number of Cycles</b> <b>459299</b>		
<b>Memory Access [ops.]</b> <b>107116</b>		
<i>Store Operations</i>	37888	
<i>Load Operations</i>	69228	

**Static Memory Allocation [byte]**                    **3578**  
     *Huffman Tree*                    3066  
     *Symbol Address Table*                    512

<b>Adaptive Huffman Encode (Text)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>423895</b>	<b>78.55%</b>
<i>Sort Tree</i>	224613	41.62%
<i>Switch Nodes</i>	27980	5.18%
<b>Insert Node</b>	<b>1887</b>	<b>0.35%</b>
<b>Encode</b>	<b>81122</b>	<b>15.03%</b>
<b>Streams</b>	<b>10226</b>	<b>1.89%</b>

**Total Number of Cycles**                   **539648**

**Memory Access [ops.]**                   **125413**

*Store Operations*                   44136

*Load Operations*                   81277

<b>Adaptive Huffman Decode (Text)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>423919</b>	<b>77.00%</b>
<i>Sort Tree</i>	224637	40.80%
<i>Switch Nodes</i>	27980	5.08%
<b>Insert Node</b>	<b>1887</b>	<b>0.34%</b>
<b>Decode</b>	<b>110681</b>	<b>20.10%</b>
<b>Streams</b>	<b>14553</b>	<b>2.64%</b>

**Total Number of Cycles**                   **550573**

**Memory Access [ops.]**                   **128674**

*Store Operations*                   44851

*Load Operations*                   83823



Deflate Encode (Exponential)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>10387</b>	<b>0.0202</b>
<i>streams</i>	4800	0.009319194
<b>Add Match</b>	<b>9000</b>	<b>0.0175</b>
<b>Control Match</b>	<b>389452</b>	<b>0.7561</b>
<b>CRC</b>	<b>31000</b>	<b>0.0602</b>

**Total Number of Cycles            515066**

**Memory Access [ops.]            81537**  
*Store Operations                    11751*  
*Load Operations                    69786*

Deflate Decode (Exponential)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>18146</b>	<b>0.594697342</b>
<i>streams</i>	6116	0.2004

**Total Number of Cycles            30513**

**Memory Access [ops.]            3315**  
*Store Operations                    1594*  
*Load Operations                    1721*

Deflate Encode (Gamma)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>22370</b>	<b>0.1629</b>
<i>streams</i>	11040	0.080416066
<b>Add Match</b>	<b>9000</b>	<b>0.0656</b>
<b>Control Match</b>	<b>37051</b>	<b>0.2699</b>
<b>CRC</b>	<b>31000</b>	<b>0.2258</b>

**Total Number of Cycles            137286**

**Memory Access [ops.]            30322**  
*Store Operations                    9564*  
*Load Operations                    20758*

Deflate Decode (Gamma)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>30111</b>	<b>0.942559319</b>
<i>streams</i>	13607	0.4259

**Total Number of Cycles            31946**

**Memory Access [ops.]            4816**  
*Store Operations                    2397*  
*Load Operations                    2419*

Deflate Encode (Poisson)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>23904</b>	<b>0.1731</b>
<i>streams</i>	<i>12004</i>	<i>0.086921891</i>
<b>Add Match</b>	<b>9000</b>	<b>0.0652</b>
<b>Control Match</b>	<b>33809</b>	<b>0.2448</b>
<b>CRC</b>	<b>31000</b>	<b>0.2245</b>

**Total Number of Cycles**            **138101**

**Memory Access [ops.]**            **30941**  
*Store Operations*                    9898  
*Load Operations*                    21043

Deflate Decode (Poisson)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>31447</b>	<i>0.93826829</i>
<i>streams</i>	<i>14630</i>	<i>0.4365</i>

**Total Number of Cycles**            **33516**

**Memory Access [ops.]**            **4997**  
*Store Operations*                    2519  
*Load Operations*                    2478

Deflate Encoding (Text)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>21828</b>	<b>15.54%</b>
<i>streams</i>	<i>11452</i>	<i>8.15%</i>
<b>Add Match</b>	<b>10000</b>	<b>7.12%</b>
<b>Control Match</b>	<b>40604</b>	<b>28.91%</b>
<b>CRC</b>	<b>32000</b>	<b>22.78%</b>

**Total Number of Cycles**            **140460**

**Memory Access [ops.]**            **31169**  
*Store Operations*                    9685  
*Load Operations*                    21484

Deflate Decoding (Text)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>29258</b>	<i>95.89%</i>
<i>streams</i>	<i>13747</i>	<i>45.05%</i>

**Total Number of Cycles**            **30513**

**Memory Access [ops.]**            **4654**  
*Store Operations*                    2357  
*Load Operations*                    2297

<b>Rice Encode (Exponential)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Encode</b>	<b>24363</b>	<b>0.3095</b>
<i>Streams</i>	13436	0.1707
<b>Calc K</b>	<b>930</b>	<b>0.0118</b>
<i>CLZ</i>	434	0.0055
<b>Maintain Tables</b>	<b>338</b>	<b>0.0043</b>
<b>Update Tables</b>	<b>7934</b>	<b>0.1008</b>

**Total Number of Cycles**            **78729**

**Memory Access [ops.]**            **14768**

*Store Operations*                7259

*Load Operations*                7509

<b>Rice Decode (Exponential)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Decode</b>	<b>41790</b>	<b>0.4268</b>
<i>Streams</i>	25050	0.2558
<i>CLZ1</i>	14728	0.1504
<b>Calc K</b>	<b>930</b>	<b>0.0095</b>
<i>CLZ2</i>	434	0.0044
<b>Maintain Tables</b>	<b>346</b>	<b>0.0035</b>
<b>Update Table</b>	<b>7934</b>	<b>0.0810</b>

**Total Number of Cycles**            **97912**

**Memory Access [ops.]**            **14758**

*Store Operations*                8156

*Load Operations*                6602

<b>Rice Encode (Gamma)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Encode</b>	<b>24909</b>	<b>0.2728</b>
<i>Streams</i>	15186	0.1663
<b>Calc K</b>	<b>3772</b>	<b>0.0413</b>
<i>CLZ</i>	3276	0.0359
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Tables</b>	<b>15199</b>	<b>0.1664</b>

**Total Number of Cycles            91322**

**Memory Access [ops.]            15971**  
     *Store Operations*            7821  
     *Load Operations*            8150

<b>Rice Decode (Gamma)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Decode</b>	<b>44101</b>	<b>0.3976</b>
<i>Streams</i>	24420	0.2201
<i>CLZ1</i>	14678	0.1323
<b>Calc K</b>	<b>3772</b>	<b>0.0340</b>
<i>CLZ2</i>	3276	0.0295
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Table</b>	<b>15199</b>	<b>0.1370</b>

**Total Number of Cycles            110930**

**Memory Access [ops.]            16045**  
     *Store Operations*            8682  
     *Load Operations*            7363

<b>Rice Encode (Poisson)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Encode</b>	<b>24909</b>	<b>0.2670</b>
<i>Streams</i>	15424	0.1654
<b>Calc K</b>	<b>3700</b>	<b>0.0397</b>
<i>CLZ</i>	3204	0.0344
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Tables</b>	<b>16956</b>	<b>0.1818</b>

**Total Number of Cycles**            **93275**

**Memory Access [ops.]**            **16296**

*Store Operations*                7998

*Load Operations*                8298

<b>Rice Decode (Poisson)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Decode</b>	<b>43789</b>	<b>0.3893</b>
<i>Streams</i>	26601	0.2365
<i>CLZ1</i>	18629	0.1656
<b>Calc K</b>	<b>3700</b>	<b>0.0329</b>
<i>CLZ2</i>	3204	0.0285
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Table</b>	<b>16956</b>	<b>0.1507</b>

**Total Number of Cycles**            **112490**

**Memory Access [ops.]**            **16402**

*Store Operations*                8874

*Load Operations*                7528

<b>Rice Encode (Text)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Encode</b>	<b>25297</b>	<b>0.2081</b>
<i>Streams</i>	16514	0.1359
<b>Calc K</b>	<b>3562</b>	<b>0.0293</b>
<i>CLZ</i>	3066	0.0252
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Tables</b>	<b>44529</b>	<b>0.3664</b>

**Total Number of Cycles            121546**

**Memory Access [ops.]            19801**  
     *Store Operations*            8555  
     *Load Operations*            11246

<b>Rice Decode (Text)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Decode</b>	<b>44852</b>	<b>0.3167</b>
<i>Streams</i>	25564	0.1805
<i>CLZ1</i>	14944	0.1055
<b>Calc K</b>	<b>3562</b>	<b>0.0252</b>
<i>CLZ2</i>	3066	0.0217
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Table</b>	<b>44529</b>	<b>0.3145</b>

**Total Number of Cycles            141604**

**Memory Access [ops.]            20025**  
     *Store Operations*            9464  
     *Load Operations*            10561

**With All Enhancements**

<b>Adaptive Huffman Encode (Exponential)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>67165</b>	<b>52.57%</b>
<i>Sort Tree</i>	28250	22.11%
<i>Switch Nodes</i>	1453	1.14%
<b>Insert Node</b>	<b>352</b>	<b>0.28%</b>
<b>Encode</b>	<b>32943</b>	<b>25.78%</b>
<b>Streams</b>	<b>8200</b>	<b>6.42%</b>

**Total Number of Cycles**                    **127771**

**Memory Access [ops.]**                    **29921**  
     *Store Operations*                    10406  
     *Load Operations*                    19515

<b>Adaptive Huffman Decode (Exponential)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>69319</b>	<b>52.44%</b>
<i>Sort Tree</i>	29507	22.32%
<i>Switch Nodes</i>	1681	1.27%
<b>Insert Node</b>	<b>352</b>	<b>0.27%</b>
<b>Decode</b>	<b>48301</b>	<b>36.54%</b>
<b>Streams</b>	<b>12372</b>	<b>9.36%</b>

**Total Number of Cycles**                    **132183**

**Memory Access [ops.]**                    **29135**  
     *Store Operations*                    10475  
     *Load Operations*                    18660

<b>Adaptive Huffman Decode (Gamma)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>169678</b>	<b>63.68%</b>
<i>Sort Tree</i>	88443	33.19%
<i>Switch Nodes</i>	7220	2.71%
<b>Insert Node</b>	<b>792</b>	<b>0.30%</b>
<b>Decode</b>	<b>81475</b>	<b>30.58%</b>
<b>Streams</b>	<b>13898</b>	<b>5.22%</b>

**Total Number of Cycles**                    **266454**

**Memory Access [ops.]**                    **56867**  
     *Store Operations*                    17478  
     *Load Operations*                    39389

<b>Adaptive Huffman Encode (Gamma)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>169652</b>	<b>66.66%</b>
<i>Sort Tree</i>	88417	34.74%
<i>Switch Nodes</i>	7220	2.84%
<b>Insert Node</b>	<b>792</b>	<b>0.31%</b>
<b>Encode</b>	<b>55260</b>	<b>21.71%</b>
<b>Streams</b>	<b>9601</b>	<b>3.77%</b>

**Total Number of Cycles**                    **254506**

**Memory Access [ops.]**                    **56410**  
     *Store Operations*                    17705  
     *Load Operations*                    38705



<b>Adaptive Huffman Encode (Poisson)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>189703</b>	<b>68.16%</b>
<i>Sort Tree</i>	101982	36.64%
<i>Switch Nodes</i>	10953	3.94%
<b>Insert Node</b>	<b>880</b>	<b>0.32%</b>
<b>Encode</b>	<b>58705</b>	<b>21.09%</b>
<b>Streams</b>	<b>9824</b>	<b>3.53%</b>

**Total Number of Cycles**                    **278331**

**Memory Access [ops.]**                    **62094**  
     *Store Operations*                    19553  
     *Load Operations*                    42541

<b>Adaptive Huffman Decode (Poisson)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>189617</b>	<b>65.01%</b>
<i>Sort Tree</i>	101965	34.96%
<i>Switch Nodes</i>	10982	3.77%
<b>Insert Node</b>	<b>880</b>	<b>0.30%</b>
<b>Decode</b>	<b>86629</b>	<b>29.70%</b>
<b>Streams</b>	<b>14148</b>	<b>4.85%</b>

**Total Number of Cycles**                    **291673**

**Memory Access [ops.]**                    **62811**  
     *Store Operations*                    19307  
     *Load Operations*                    43504

<b>Adaptive Huffman Encode (Text)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>235689</b>	<b>70.92%</b>
<i>Sort Tree</i>	135548	40.79%
<i>Switch Nodes</i>	19536	5.88%
<b>Insert Node</b>	<b>1628</b>	<b>0.49%</b>
<b>Encode</b>	<b>65252</b>	<b>19.63%</b>
<b>Streams</b>	<b>10405</b>	<b>3.13%</b>

**Total Number of Cycles**                    **332346**

**Memory Access [ops.]**                    **74760**  
     *Store Operations*                    23508  
     *Load Operations*                    51252

<b>Adaptive Huffman Decode (Text)</b>		
<b>Label</b>	<b>Cycles</b>	<b>Cycles/Tot.Cycles</b>
<b>Increment Tree</b>	<b>235703</b>	<b>67.60%</b>
<i>Sort Tree</i>	135562	38.88%
<i>Switch Nodes</i>	19536	5.60%
<b>Insert Node</b>	<b>1628</b>	<b>0.47%</b>
<b>Decode</b>	<b>96366</b>	<b>27.64%</b>
<b>Streams</b>	<b>14839</b>	<b>4.26%</b>

**Total Number of Cycles**                    **348687**

**Memory Access [ops.]**                    **76015**  
     *Store Operations*                    23220  
     *Load Operations*                    52795

Deflate Decode (Exponential)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>16782</b>	<b>0.974507868</b>
<i>streams</i>	6294	0.3655

**Total Number of Cycles            17221**

**Memory Access [ops.]            3315**  
*Store Operations                    1594*  
*Load Operations                    1721*

Deflate Encode (Exponential)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>9853</b>	<b>0.0218</b>
<i>streams</i>	4640	0.010287222
<b>Add Match</b>	<b>8000</b>	<b>0.0177</b>
<b>Control Match</b>	<b>349948</b>	<b>0.7759</b>
<b>CRC</b>	<b>26000</b>	<b>0.0576</b>

**Total Number of Cycles            451045**

**Memory Access [ops.]            77320**  
*Store Operations                    11133*  
*Load Operations                    66187*

Deflate Decode (Gamma)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>28591</b>	<b>0.958914677</b>
<i>streams</i>	14056	0.4714

**Total Number of Cycles            29816**

**Memory Access [ops.]            4816**  
*Store Operations                    2397*  
*Load Operations                    2419*

Deflate Encode (Gamma)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>21023</b>	<b>0.1774</b>
<i>streams</i>	10480	0.088440311
<b>Add Match</b>	<b>8000</b>	<b>0.0675</b>
<b>Control Match</b>	<b>34186</b>	<b>0.2885</b>
<b>CRC</b>	<b>26000</b>	<b>0.2194</b>

**Total Number of Cycles            118498**

**Memory Access [ops.]            27964**  
*Store Operations                    9446*  
*Load Operations                    18518*

Deflate Encode (Poisson)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>22410</b>	<b>0.1882</b>
<i>streams</i>	<i>11338</i>	<i>0.095228496</i>
<b>Add Match</b>	<b>8000</b>	<b>0.0672</b>
<b>Control Match</b>	<b>31440</b>	<b>0.2641</b>
<b>CRC</b>	<b>26000</b>	<b>0.2184</b>

**Total Number of Cycles**            **119061**

**Memory Access [ops.]**            **28152**  
*Store Operations*                    *9782*  
*Load Operations*                    *18370*

Deflate Decode (Poisson)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>29966</b>	<i>0.955944748</i>
<i>streams</i>	<i>15128</i>	<i>0.4826</i>

**Total Number of Cycles**            **31347**

**Memory Access [ops.]**            **4997**  
*Store Operations*                    *2519*  
*Load Operations*                    *2478*

Deflate Encode (Text)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>19859</b>	<b>16.41%</b>
<i>streams</i>	<i>10102</i>	<i>8.35%</i>
<b>Add Match</b>	<b>8000</b>	<b>6.61%</b>
<b>Control Match</b>	<b>36418</b>	<b>30.09%</b>
<b>CRC</b>	<b>26000</b>	<b>21.48%</b>

**Total Number of Cycles**            **121038**

**Memory Access [ops.]**            **27701**  
*Store Operations*                    *9423*  
*Load Operations*                    *18278*

Deflate Decode (Text)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>27238</b>	<i>95.60%</i>
<i>streams</i>	<i>13462</i>	<i>47.25%</i>
<b>Total Number of Cycles</b>	<b>28491</b>	
<b>Memory Access [ops.]</b>	<b>4654</b>	
<i>Store Operations</i>	<i>2357</i>	
<i>Load Operations</i>	<i>2297</i>	

Rice Encode (Exponential)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>30093</b>	<b>0.4707</b>
<i>Streams</i>	15126	<b>0.2366</b>
<b>Calc K</b>	<b>434</b>	<b>0.0068</b>
<b>Maintain Tables</b>	<b>238</b>	0.0037
<b>Update Tables</b>	<b>8156</b>	0.1276

**Total Number of Cycles            63932**

**Memory Access [ops.]            8616**

*Store Operations            4184*

*Load Operations            4432*

Rice Decode (Exponential)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>25116</b>	<b>36.97%</b>
<i>Streams</i>	23158	34.09%
<b>Calc K</b>	<b>434</b>	<b>0.64%</b>
<b>Maintain Tables</b>	<b>238</b>	<b>0.35%</b>
<b>Update Table</b>	<b>7818</b>	<b>11.51%</b>

**Total Number of Cycles            67941**

**Memory Access [ops.]            8612**

*Store Operations            5069*

*Load Operations            3543*

Rice Decode (Gamma)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>25572</b>	<b>0.3360</b>
<i>Streams</i>	24646	0.3238
<b>Calc K</b>	<b>434</b>	<b>0.0057</b>
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Table</b>	<b>15124</b>	<b>0.1987</b>

**Total Number of Cycles**            **76111**

**Memory Access [ops.]**            **9830**  
     *Store Operations*                5413  
     *Load Operations*                4417

Rice Encode (Gamma)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>31468</b>	<b>0.4393</b>
<i>Streams</i>	16501	0.2303
<b>Calc K</b>	<b>434</b>	<b>0.0061</b>
<b>Maintain Tables</b>	<b>0</b>	<i>0.0000</i>
<b>Update Tables</b>	<b>14724</b>	<i>0.2055</i>

**Total Number of Cycles**            **71637**

**Memory Access [ops.]**            **9744**  
     *Store Operations*                4653  
     *Load Operations*                5091

Rice Encode (Poisson)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>31655</b>	<b>0.4325</b>
<i>Streams</i>	16688	0.2280
<b>Calc K</b>	<b>434</b>	<b>0.0059</b>
<b>Maintain Tables</b>	<b>0</b>	<i>0.0000</i>
<b>Update Tables</b>	<b>16097</b>	<i>0.2199</i>

**Total Number of Cycles**            **73197**

**Memory Access [ops.]**            **10037**

*Store Operations*                4798

*Load Operations*                5239

Rice Decode (Poisson)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>25572</b>	<b>0.3286</b>
<i>Streams</i>	24850	0.3193
<b>Calc K</b>	<b>434</b>	<b>0.0056</b>
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Table</b>	<b>16625</b>	<b>0.2136</b>

**Total Number of Cycles**            **77817**

**Memory Access [ops.]**            **10294**

*Store Operations*                5680

*Load Operations*                4614

Rice Encode (Text)		
Label	Cycles	Cycles/Tot.Cycles
<b>Encode</b>	<b>32628</b>	<b>0.3335</b>
<i>Streams</i>	17581	0.1797
<b>Calc K</b>	<b>434</b>	<b>0.0044</b>
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Tables</b>	<b>39763</b>	<b>0.4064</b>

**Total Number of Cycles**            **97836**

**Memory Access [ops.]**            **13448**

*Store Operations*                5261

*Load Operations*                8187

Rice Decode (Text)		
Label	Cycles	Cycles/Tot.Cycles
<b>Decode</b>	<b>26024</b>	<b>0.2528</b>
<i>Streams</i>	25878	0.2513
<b>Calc K</b>	<b>434</b>	<b>0.0042</b>
<b>Maintain Tables</b>	<b>0</b>	<b>0.0000</b>
<b>Update Table</b>	<b>40644</b>	<b>0.3948</b>

**Total Number of Cycles**            **102960**

**Memory Access [ops.]**            **13885**

*Store Operations*                6161

*Load Operations*                7724



## E. Zip-File

This thesis has been delivered with zip-file named Appendix\_E.zip. The folder structure in this file is shown here.

```
Appendix_E
├── Assembly_Source_Code
│   ├── Enhanced
│   │   ├── Deflate
│   │   │   ├── Decode
│   │   │   │   └── RAM
│   │   │   └── Encode
│   │   │       └── RAM
│   │   ├── Huffman
│   │   │   ├── Decode
│   │   │   │   └── RAM
│   │   │   └── Encode
│   │   │       └── RAM
│   │   ├── Rice
│   │   │   ├── Decode
│   │   │   │   └── RAM
│   │   │   └── Encode
│   │   │       └── RAM
│   └── Original
│       ├── Deflate
│       │   ├── Decode
│       │   │   └── RAM
│       │   └── Encode
│       │       └── RAM
│       ├── Huffman
│       │   ├── Decode
│       │   │   └── RAM
│       │   └── Encode
│       │       └── RAM
│       └── Rice
│           ├── Decode
│           │   └── RAM
│           └── Encode
│               └── RAM
├── NanoRisc_Assembler
├── NanoRisc_ISS
├── Stream_Builder_Source_Code
└── VHDL_Source_Code
```